

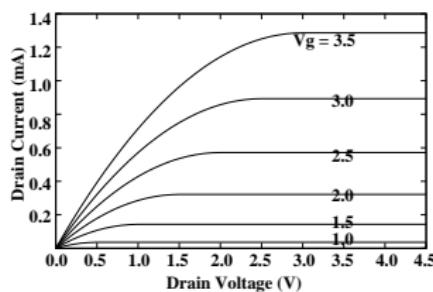
CMOS Logic Design

Dinesh Sharma

EE Department
IIT Bombay, Mumbai

September 9, 2020

A simple model



$K \equiv \mu C_{ox} W/L$ and V_T is the threshold voltage.
for $V_{gs} \leq V_T$, $I_{ds} = 0$

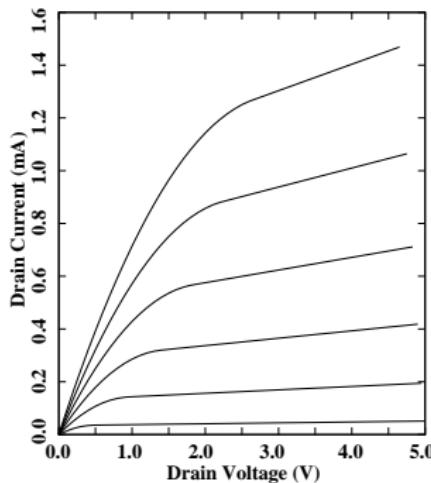
for $V_{gs} > V_T$ and $V_{ds} \leq V_{gs} - V_T$,
 $I_{ds} = K [(V_{gs} - V_T)V_{ds} - \frac{1}{2}V_{ds}^2]$

for $V_{gs} > V_T$ and $V_{ds} > V_{gs} - V_T$,
 $I_{ds} = K \frac{(V_{gs} - V_T)^2}{2}$

This model assumes current to be independent of V_{ds} in the saturation region.

(This is somewhat oversimplified. Better models exist, but we shall use this one for analysing digital circuits.)

A more realistic model



Let 'Early Voltage' $\equiv V_E$

$$\begin{aligned} \text{define } V_{dss} &\equiv V_E \left(\sqrt{1 + \frac{2(V_{gs} - V_T)}{V_E}} - 1 \right) \\ &\simeq (V_{gs} - V_T) \left(1 - \frac{V_{gs} - V_T}{2V_E} \right) \\ \text{and } I_{dss} &\equiv K \left[(V_{gs} - V_T)V_{dss} - \frac{1}{2}V_{dss}^2 \right] \end{aligned}$$

$$\text{for } V_{gs} > V_T \text{ and } V_{ds} \leq V_{dss} \quad I_{ds} = K \left[(V_{gs} - V_T)V_{ds} - \frac{1}{2}V_{ds}^2 \right]$$

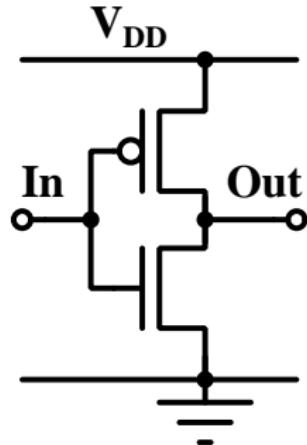
$$\text{for } V_{gs} > V_T \text{ and } V_{ds} > V_{dss} \quad I_{ds} = I_{dss} \frac{V_{ds} + V_E}{V_{dss} + V_E}$$

CMOS Static Logic

- Each logic stage contains pull up and pull down networks controlled by input signals.
- The pull up network contains p channel transistors.
- The pull down network is made of n channel transistors.
- If the pull up network is ‘on’, the pull down network is ‘off’ and *vice versa*.
- Since the pull up and pull down networks are never ‘on’ simultaneously, there is no static power consumption.

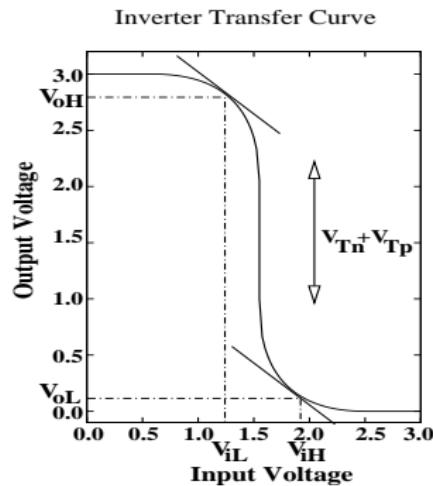
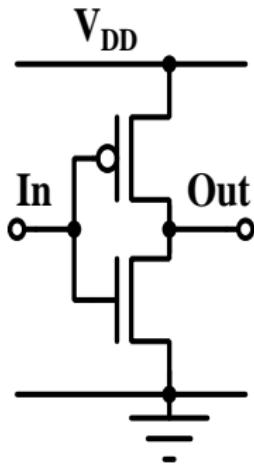
CMOS Inverter

The simplest of CMOS logic structure is the inverter. This is the basic gate of CMOS style of design. More complex gates are designed by mapping them to an 'equivalent' inverter.



- The pull up network of the logic gate is made equivalent to the pMOS of the inverter.
- The pull down network of the logic gate is made equivalent to the nMOS of the inverter.
- Thumb rules are used to map the geometries of the pull up and pull down networks to single transistors.

Static Characteristics

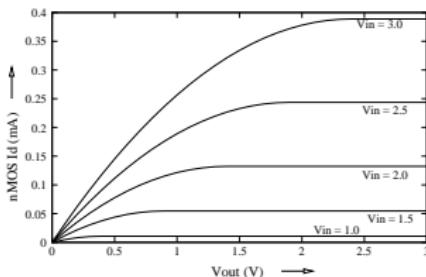
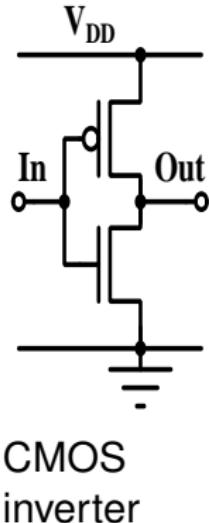


The range of input voltages can be divided into several regions.

- nMOS ‘off’, pMOS ‘on’
- nMOS saturated, pMOS linear
- nMOS saturated, pMOS saturated
- nMOS linear, pMOS saturated
- nMOS ‘on’, pMOS ‘off’

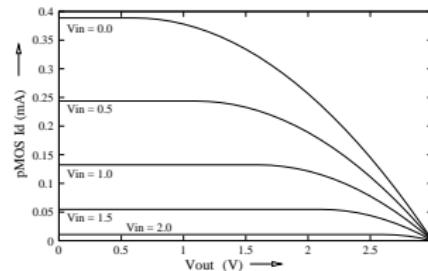
Transistor Currents

As we sweep the input voltage from 0 to V_{DD} , the current through the nMOS and pMOS can be plotted as a function of V_{in} and V_{out} .



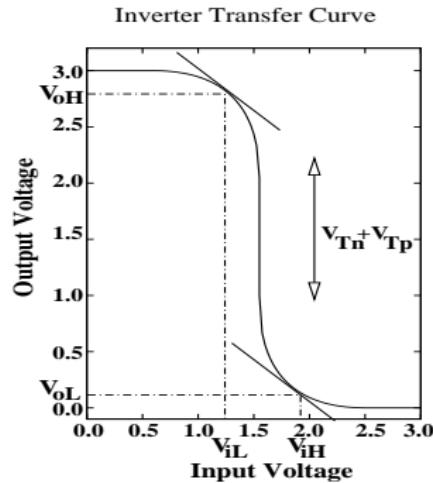
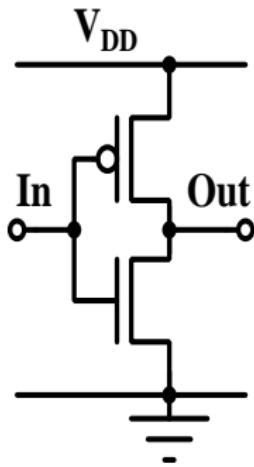
The gate voltage for the nMOS transistor = V_{in} and the drain voltage = V_{out} .

The output voltage for the inverter for a given input voltage is the V_{out} value where the current through the two transistors is equal.



The absolute gate voltage for pMOS is $V_{DD} - V_{in}$ and the absolute drain voltage is $V_{DD} - V_{out}$.

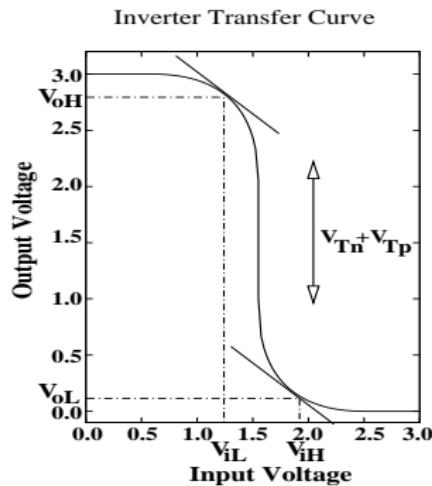
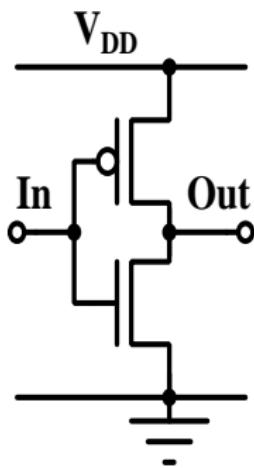
nMOS ‘off’, pMOS ‘on’



For $0 < V_i < V_{Tn}$

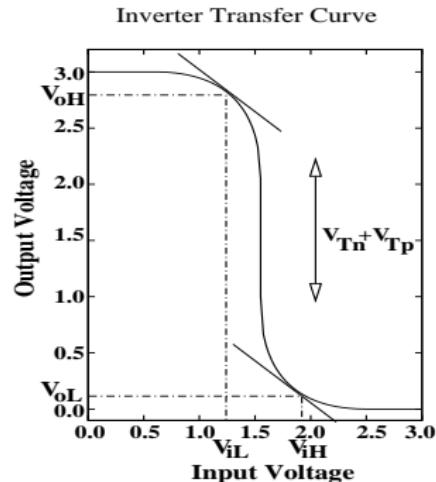
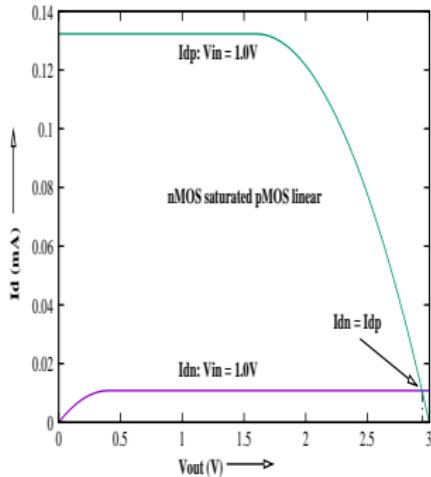
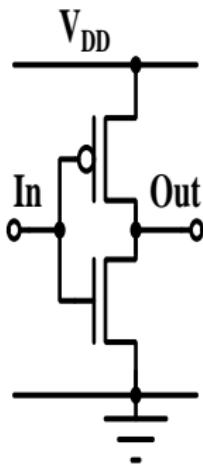
- the n channel transistor is ‘off’,
- the p channel transistor is ‘on’ and the output voltage = V_{DD} .
- This is the normal digital operation range with input = ‘0’ and output = ‘1’.

nMOS saturated, pMOS linear



- In this regime, both transistors are ‘on’.
- The input voltage V_i is $> V_{Tn}$, but is small enough so that the n channel transistor is in saturation, and the p channel transistor is in the linear regime.
- In static condition, the output voltage will adjust itself such that the currents through the n and p channel transistors are equal.

nMOS saturated, pMOS linear



- The absolute value of gate-source voltage on the p channel transistor is $V_{DD} - V_i$, and therefore the “over voltage” on its gate is $V_{DD} - V_i - V_{Tp}$.
- The drain source voltage of the pMOS has an absolute value $V_{DD} - V_o$.

nMOS saturated, pMOS linear

$$\begin{aligned} I_{dp} &= K_p \left[(V_{DD} - V_i - V_{Tp})(V_{DD} - V_o) - \frac{1}{2}(V_{DD} - V_o)^2 \right] \\ I_{dn} &= \frac{K_n}{2}(V_i - V_{Tn})^2 \end{aligned}$$

Where symbols have their usual meanings. Equating currents, we get

$$K_p \left[(V_{DD} - V_i - V_{Tp})(V_{DD} - V_o) - \frac{1}{2}(V_{DD} - V_o)^2 \right] = \frac{K_n}{2}(V_i - V_{Tn})^2$$

We define $\beta \equiv K_n/K_p$ and $V_{dp} \equiv V_{DD} - V_o$. This gives

$$(V_{DD} - V_i - V_{Tp})V_{dp} - \frac{1}{2}V_{dp}^2 = \frac{\beta}{2}(V_i - V_{Tn})^2$$

nMOS saturated, pMOS linear

$$\frac{1}{2}V_{dp}^2 - (V_{DD} - V_i - V_{Tp})V_{dp} + \frac{\beta}{2}(V_i - V_{Tn})^2 = 0$$

Solving the quadratic equation above we get V_{dp} , from which V_o may be written as:

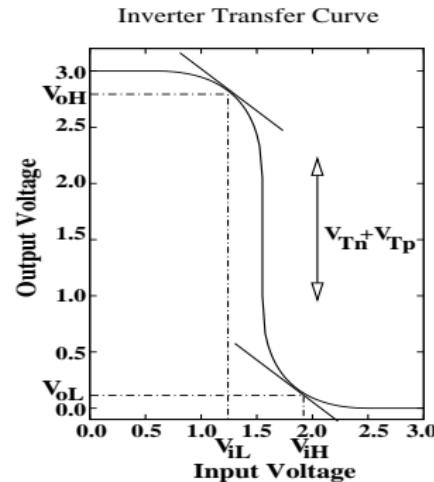
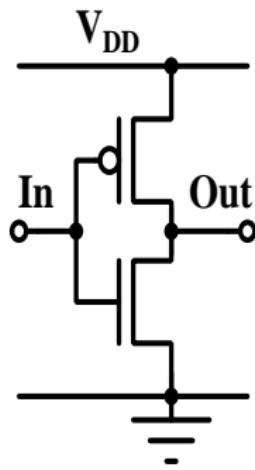
$$V_o = V_i + V_{Tp} + \sqrt{(V_{DD} - V_i - V_{Tp})^2 - \beta(V_i - V_{Tn})^2}$$

If $K_n = K_p$; ($\beta = 1$),

$$V_o = (V_i + V_{Tp}) + \sqrt{(V_{DD} - V_{Tn} - V_{Tp})(V_{DD} - 2V_i + V_{Tn} - V_{Tp})}$$

$$\text{for } V_i \leq \frac{V_{DD} + V_{Tn} - V_{Tp}}{2}$$

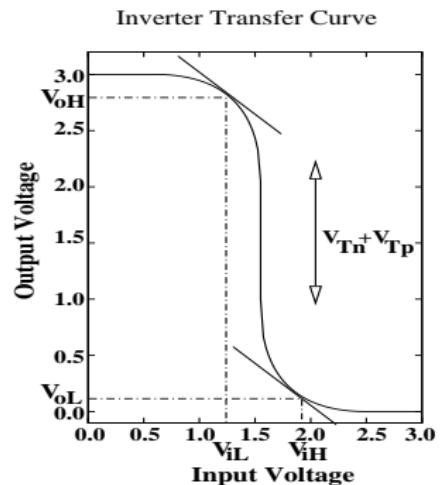
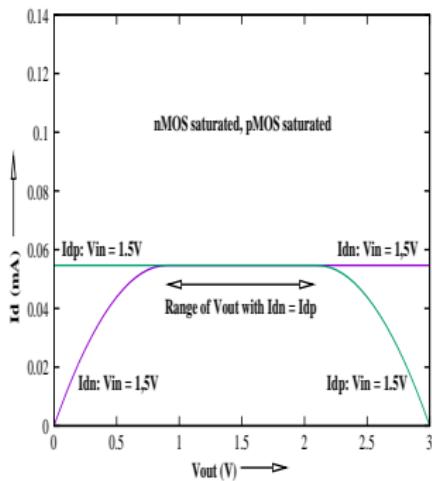
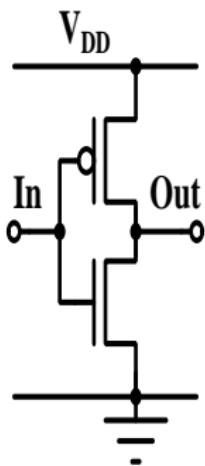
nMOS saturated, pMOS saturated



When the input voltage is $V_i = (V_{DD} + \sqrt{\beta}V_{Tn} - V_{Tp})/(1 + \sqrt{\beta})$, both transistors are saturated.

- Currents of both transistors are independent of their drain voltages.
- we do not get a unique solution for V_o by equating drain currents.

nMOS saturated, pMOS saturated



nMOS saturated, pMOS saturated

When the input voltage is $V_i = (V_{DD} + \sqrt{\beta}V_{Tn} - V_{Tp})/(1 + \sqrt{\beta})$, both transistors are saturated.

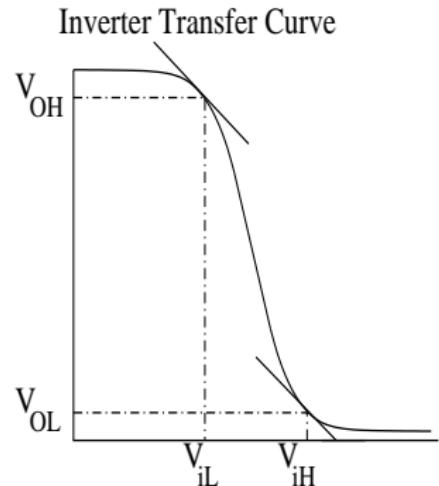
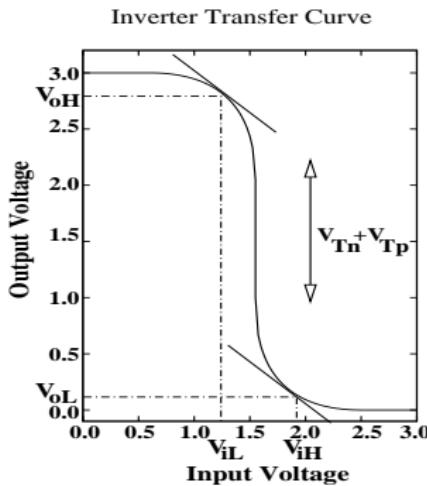
- Currents of both transistors are independent of their drain voltages.
- we do not get a unique solution for V_o by equating drain currents.
- The currents will be equal for all values of V_o in the range
$$V_i - V_{Tn} \leq V_o \leq V_i + V_{Tp}$$

All output voltages in the range $V_i - V_{Tn} \leq V_o \leq V_i + V_{Tp}$ satisfy the drain current equations at this input voltage.

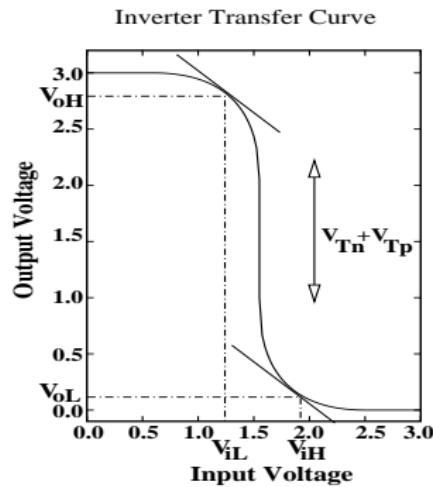
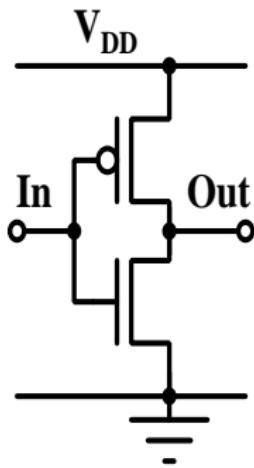
So the transfer curve of an inverter shows a drop of $V_{Tn} + V_{Tp}$ at a voltage near $V_{DD}/2$.

nMOS saturated, pMOS saturated

The sudden drop in V_o for $V_i = (V_{DD} + \sqrt{\beta}V_{Tn} - V_{Tp})/(1 + \sqrt{\beta})$ is an artifact of the simple model we have chosen for transistor currents. In reality, drain currents are weakly dependent on drain-source voltage. The transfer curve then has no discontinuity.



nMOS linear, pMOS saturated

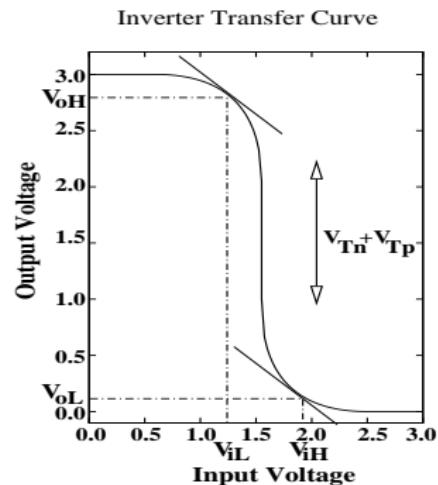
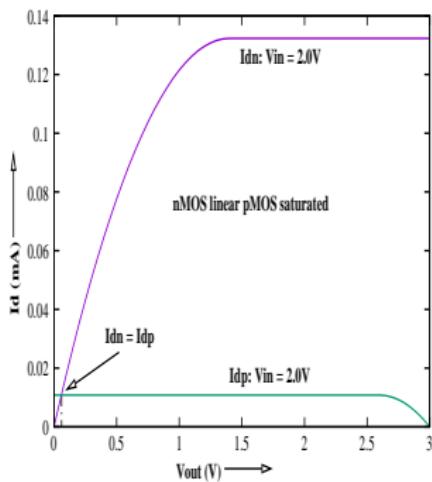
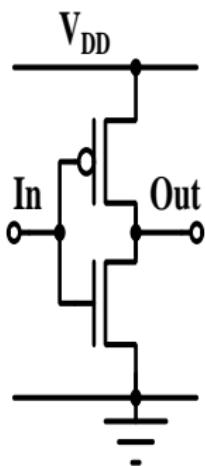


As we increase V_i further, so that

$$\frac{V_{DD} + \sqrt{\beta}V_{Tn} - V_{Tp}}{1 + \sqrt{\beta}} < V_i < V_{DD} - V_{Tp}$$

both transistors are still 'on', but nMOS enters the linear regime while pMOS is saturated.

nMOS linear, pMOS saturated



nMOS linear, pMOS saturated

Equating currents when nMOS is saturated and pMOS is in linear regime,

$$\begin{aligned} I_d &= \frac{K_p}{2}(V_{DD} - V_i - V_{Tp})^2 \\ &= K_n \left[(V_i - V_{Tn})V_o - \frac{1}{2}V_o^2 \right] \end{aligned}$$

From this, we get the quadratic equation

$$\frac{1}{2}V_o^2 - (V_i - V_{Tn})V_o + \frac{(V_{DD} - V_i - V_{Tp})^2}{2\beta} = 0$$

with solutions:

$$V_o = (V_i - V_{Tn}) \pm \sqrt{(V_i - V_{Tn})^2 - \frac{(V_{DD} - V_i - V_{Tp})^2}{\beta}}$$

$$V_o = (V_i - V_{Tn}) \pm \sqrt{(V_i - V_{Tn})^2 - \frac{(V_{DD} - V_i - V_{Tp})^2}{\beta}}$$

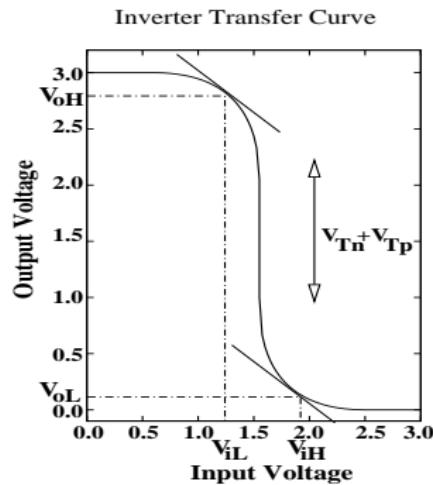
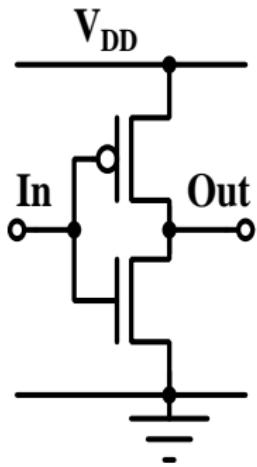
We must choose the negative sign, since $V_o \leq V_i - V_{Tn}$ for nMOS to be in linear regime. So,

$$V_o = (V_i - V_{Tn}) - \sqrt{(V_i - V_{Tn})^2 - \frac{(V_{DD} - V_i - V_{Tp})^2}{\beta}}$$

In the special case where $\beta = 1$, we have

$$V_o = (V_i - V_{Tn}) - \sqrt{(V_{DD} - V_{Tn} - V_{Tp})(2V_i - V_{DD} - V_{Tn} + V_{Tp})}$$

nMOS 'on', pMOS 'off'



- As we increase the input voltage beyond $V_{DD} - V_{Tp}$, the p channel transistor turns 'off', while the n channel conducts strongly.
- As a result, the output voltage falls to zero.
- This is the normal digital operation range with input = '1' and output = '0'.

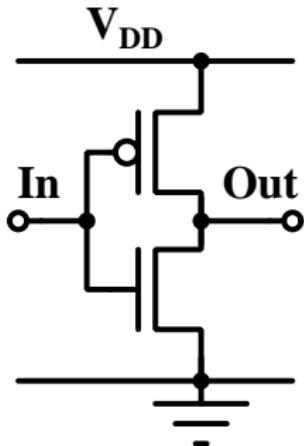
Noise Margins

- For robust design, the output levels must be interpreted correctly at the input of next stage even in the presence of noise.
- For the ‘high’ level, we require that the output of one stage should still be interpreted as ‘high’ at the input of the next gate even when pulled down a little due to noise.
- Therefore V_{oH} should be $> V_{iH}$.
- Similarly V_{oL} should be $< V_{iL}$
- The difference, $V_{iL} - V_{oL}$ is the ‘low’ noise margin. and $V_{oH} - V_{iH}$ is the ‘high’ noise level.

Logic Levels

- A digital circuit should distinguish logic levels, but be insensitive to the exact analog voltage at the input.
- Therefore flat portions of the transfer curve (where $\frac{\partial V_o}{\partial V_i}$ is small) are suitable for digital logic.
- We select two points on the transfer curve where the slope ($\frac{\partial V_o}{\partial V_i}$) is -1.0.
- The coordinates of these two points define the values of (V_{iL}, V_{oH}) and (V_{iH}, V_{oL}) .
- The region to the left of V_{iL} and to the right of V_{iH} has $|\frac{\partial V_o}{\partial V_i}| < 1$, and is suitable for digital operation.

Calculation of Noise Margins



- To evaluate the values of noise margins, we shall use the expressions derived for $\beta = 1$ to keep the algebra simple.
- When the input is low and output high, the n channel transistor is saturated and the p channel transistor is in its linear regime.
- When the input is high and the output is low, the n channel transistor is in its linear regime, while the p channel transistor is saturated.

Calculation of V_{iL} and V_{oH}

for (V_{iL}, V_{oH}) , n channel transistor is saturated, while the p channel transistor is in its linear regime.

$$V_o = (V_i + V_{Tp}) + \sqrt{(V_{DD} - V_{Tn} - V_{Tp})(V_{DD} + V_{Tn} - V_{Tp} - 2V_i)}$$

From this, we evaluate $\frac{\partial V_o}{\partial V_i}$ and set it = -1.

$$\frac{\partial V_o}{\partial V_i} = -1 = 1 - \sqrt{\frac{V_{DD} - V_{Tn} - V_{Tp}}{V_{DD} + V_{Tn} - V_{Tp} - 2V_i}}$$

This gives

$$V_{iL} = \frac{3V_{DD} + 5V_{Tn} - 3V_{Tp}}{8}$$

$$V_{oH} = \frac{7V_{DD} + V_{Tn} + V_{Tp}}{8} = V_{DD} - \frac{V_{DD} - V_{Tn} - V_{Tp}}{8}$$

Calculation of V_{iH} and V_{oL}

When the input is 'high', we should use the equation for nMOS linear and pMOS saturated.

$$V_o = (V_i - V_{Tn}) - \sqrt{(V_{DD} - V_{Tn} - V_{Tp})(2V_i - V_{DD} - V_{Tn} + V_{Tp})}$$

Differentiating with respect to V_i gives

$$\frac{\partial V_o}{\partial V_i} = -1 = 1 - \sqrt{\frac{V_{DD} - V_{Tn} - V_{Tp}}{2V_i - V_{DD} - V_{Tn} + V_{Tp}}}$$

From where, we get

$$V_{iH} = \frac{5V_{DD} + 3V_{Tn} - 5V_{Tp}}{8}$$

$$V_{oL} = \frac{V_{DD} - V_{Tn} - V_{Tp}}{8}$$

Calculation of Noise Margins

The ‘High’ noise margin is given by

$$V_{oH} - V_{iH} = \frac{V_{DD} - V_{Tn} + 3V_{Tp}}{4}$$

Similarly, the ‘Low’ noise margin is

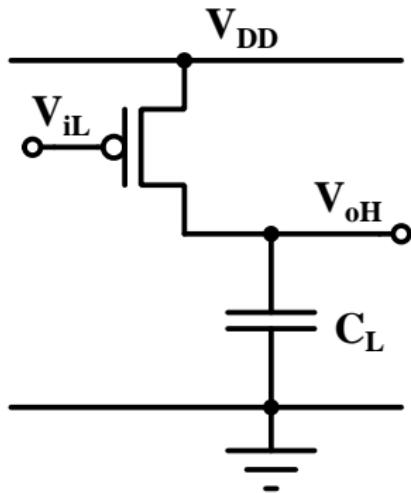
$$V_{iL} - V_{oL} = \frac{V_{DD} + 3V_{Tn} - V_{Tp}}{4}$$

The two noise margins can be made equal by choosing equal values for V_{Tn} and V_{Tp} .

Dynamic Characteristics

- For the calculation of rise and fall times, we shall assume that only one of the two transistors in the inverter is 'on'.
- This is more conservative than the static logic levels calculated by slope considerations.
- We shall use the simple model described at the beginning of this lecture.

Rise time



When the input is low, the n channel transistor is ‘off’, while the p channel transistor is ‘on’. From Kirchhoff’s current law at the output node,

$$I_{dp} = C \frac{dV_o}{dt}$$

so,

$$\frac{dt}{C} = \frac{dV_o}{I_{dp}}$$

Integrating both sides, we get

$$\frac{\tau_{rise}}{C} = \int_0^{V_{oH}} \frac{dV_o}{I_{dp}}$$

$$\frac{\tau_{rise}}{C} = \int_0^{V_{oH}} \frac{dV_o}{I_{dp}}$$

Till the output rises to $V_{iL} + V_{Tp}$, the p channel transistor is in saturation. If $V_{oH} > V_{iL} + V_{Tp}$ (which is normally the case), the integration range can be broken into saturation and linear regimes. Thus

$$\begin{aligned} \frac{\tau_{rise}}{C} &= \int_0^{V_{iL} + V_{Tp}} \frac{dV_o}{\frac{K_p}{2}(V_{DD} - V_{iL} - V_{Tp})^2} \\ &+ \int_{V_{iL} + V_{Tp}}^{V_{oH}} \frac{dV_o}{K_p [(V_{DD} - V_{iL} - V_{Tp})(V_{DD} - V_o) - \frac{1}{2}(V_{DD} - V_o)^2]} \end{aligned}$$

$$\begin{aligned}\tau_{rise} &= \frac{2C(V_{iL} + V_{Tp})}{K_p(V_{DD} - V_{iL} - V_{Tp})^2} \\ &+ \frac{C}{K_p(V_{DD} - V_{iL} - V_{Tp})} \ln \frac{V_{DD} + V_{oH} - 2V_{iL} - 2V_{Tp}}{V_{DD} - V_{oH}}\end{aligned}$$

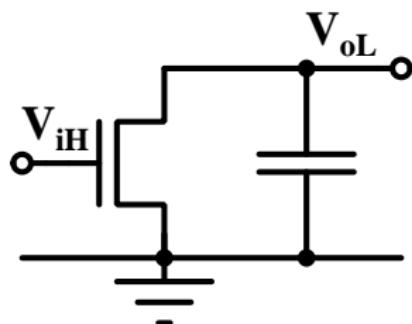
- The first term is just the constant current charging of the load capacitor.
- The second term represents the charging by the pMOS in its linear range.
- This can be compared with resistive charging, which would have taken a charge time of

$$\tau = RC \ln \frac{V_{DD} - V_{iL} - V_{Tp}}{V_{DD} - V_{oH}}$$

to charge from $V_{iL} + V_{Tp}$ to V_{oH} .

Fall Time

V_{DD}



When the input is high, the p channel transistor is ‘off’, while the n channel transistor is ‘on’. From Kirchhoff’s current law at the output node,

$$I_{dn} = -C \frac{dV_o}{dt}$$

Separating variables and integrating from the initial voltage ($= V_{DD}$) to some terminal voltage V_{oL} gives

$$\frac{\tau_{fall}}{C} = - \int_{V_{DD}}^{V_{oL}} \frac{dV_o}{I_{dn}}$$

Fall time

$$\frac{\tau_{fall}}{C} = - \int_{V_{DD}}^{V_{oL}} \frac{dV_o}{I_{dn}}$$

The n channel transistor will be in saturation till the output falls to $V_i - V_{Tn}$. Below this, the transistor will be in its linear regime. We can divide the integration range in two parts.

$$\begin{aligned}\frac{\tau_{fall}}{C} &= - \int_{V_{DD}}^{V_i - V_{Tn}} \frac{dV_o}{I_{dn}} - \int_{V_i - V_{Tn}}^{V_{oL}} \frac{dV_o}{I_{dn}} \\ &= \int_{V_i - V_{Tn}}^{V_{DD}} \frac{dV_o}{\frac{K_n}{2}(V_i - V_{Tn})^2} \\ &\quad + \int_{V_{oL}}^{V_i - V_{Tn}} \frac{dV_o}{K_n[(V_i - V_{Tn})V_o - \frac{1}{2}V_o^2]}\end{aligned}$$

Fall time

$$\frac{\tau_{fall}}{C} = \frac{V_{DD} - V_i + V_{Tn}}{\frac{K_n}{2}(V_i - V_{Tn})^2} + \frac{1}{K_n(V_i - V_{Tn})} \ln \frac{2(V_i - V_{Tn}) - V_{oL}}{V_{oL}}$$

The first term represents the time taken to discharge at constant current in the saturation regime, whereas the second term is the quasi-resistive discharge in the linear regime.

Trade off between power, speed and robustness

Noise margins are given by

$$\begin{aligned}V_{oH} - V_{iH} &= \frac{V_{DD} - V_{Tn} + 3V_{Tp}}{4} \\V_{iL} - V_{oL} &= \frac{V_{DD} + 3V_{Tn} - V_{Tp}}{4}\end{aligned}$$

- As we scale technologies, we improve speed and power consumption. However, the noise margin becomes worse.
- We can improve noise margins by choosing relatively higher threshold voltages. However, this will reduce speeds.
- We could also increase V_{DD} - but that would increase power dissipation.

Thus we have a trade off between power, speed and noise margins.

CMOS Inverter Design Flow

- A common design requirement is symmetric charge and discharge behaviour and equal noise margins for high and low logic values.
- This requires matched values of K_n and K_p and equal values of V_{Tn} and V_{Tp} .
- Rise and fall times depend linearly on K_n and K_p .
- Thus it is a straightforward calculation to determine transistor geometries if speed requirements and technological parameters are given.
- However, as transistor geometries are made larger, self loading can become significant.

CMOS Inverter Design Flow

- For large self-loading, we have to model the load capacitance as

$$C_{Load} = C_{ext} + \alpha K_n$$

where we have assumed that $\beta = K_n/K_p$ is constant. α is a technological constant.

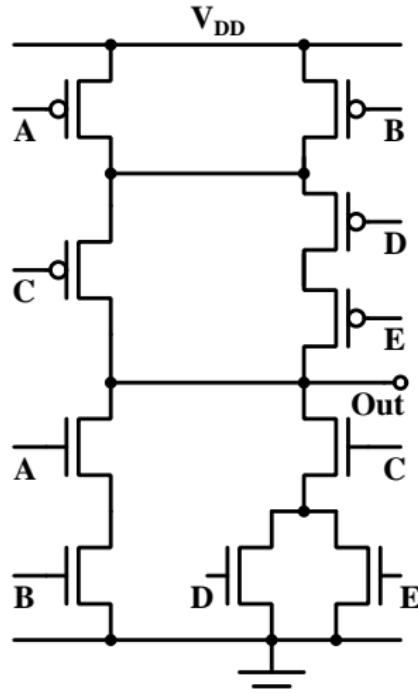
- We use the expressions for K_T/C which depend only on voltages. Once these values are calculated, the geometry can be determined.
- In the extreme case, when self capacitance dominates the load capacitance, K/C becomes constant and τ becomes geometry independent. There is no advantage in using wider transistors in this regime to increase the speed. It is better to use multi-stage logic with tapered buffers in this regime.

From Inverters to Other Logic

Once the basic CMOS inverter is designed, other logic gates can be derived from it. The logic has to be put in a canonical form which is a sum of products with a bar (inversion) on top.

- For every ‘.’ in the expression, we put the corresponding n channel transistors in series and the corresponding p channel transistors in parallel.
- for every ‘+’, we put the n channel transistors in parallel and the p channel transistors in series.
- We scale the transistor widths up by the number of devices (n or p) put in series.
- The geometries are left untouched for devices put in parallel.

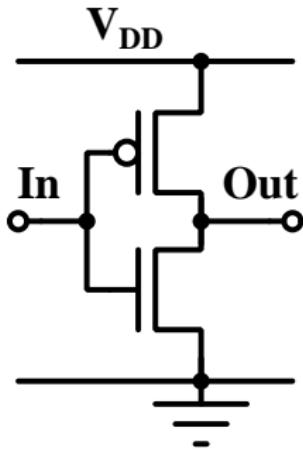
CMOS implementation of $\overline{A} \cdot B + C \cdot (\overline{D} + E)$



- For n channel, A and B are in series, The pair is in parallel with C which is in series with a parallel combination of D and E.
- For p channel, A is in parallel with B, the pair is in series with C which is in parallel with a series combination of D and E.

Implementation of $\overline{A} \cdot B + C \cdot (\overline{D} + E)$ in CMOS logic design style.

CMOS summary



- Logic consumes no static power in CMOS design style.
- However, signals have to be routed to the n pull down network *as well as* to the p pull up network.
- So the load presented to every driver is high.
- This is exacerbated by the fact that n and p channel transistors cannot be placed close together as these are in different wells which have to be kept well separated in order to avoid latch-up.

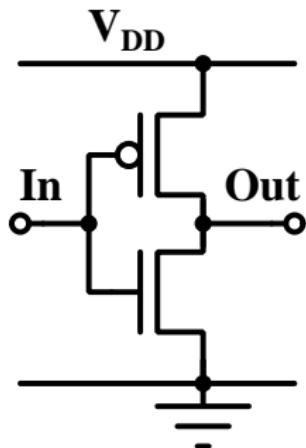
Logic Design Styles: Pseudo nMOS

Dinesh Sharma

EE Department
IIT Bombay, Mumbai

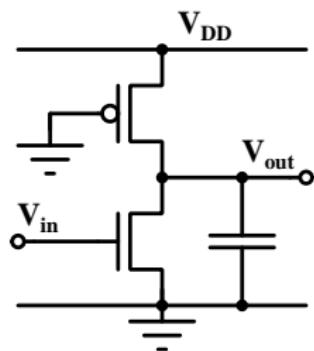
September 2, 2020

CMOS summary



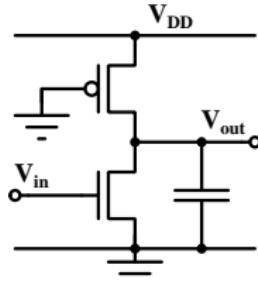
- ▶ Logic consumes no static power in CMOS design style.
- ▶ However, signals have to be routed to the n pull down network *as well as* to the p pull up network.
- ▶ So the load presented to every driver is high.
- ▶ This is exacerbated by the fact that n and p channel transistors cannot be placed close together as these are in different wells which have to be kept well separated in order to avoid latch-up.

Pseudo nMOS Design Style



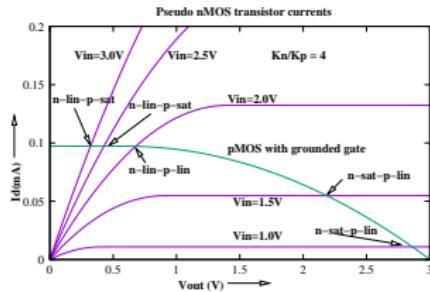
- ▶ The CMOS pull up network is replaced by a single pMOS transistor with its gate grounded.
- ▶ Since the pMOS is not driven by signals, it is always 'on'.
- ▶ The effective gate voltage seen by the pMOS transistor is V_{DD} . Thus the over-voltage on the p channel gate is always $V_{DD} - V_{Tp}$.
- ▶ When the nMOS is turned 'on', a direct path between supply and ground exists and static power will be drawn.
- ▶ However, the dynamic power is reduced due to lower capacitive loading

Static Characteristics

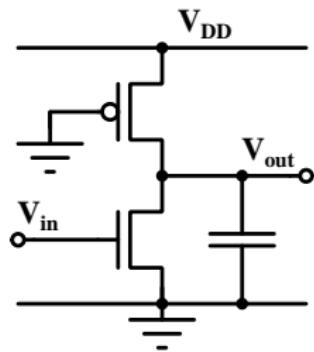


As we sweep the input voltage from ground to V_{DD} , we encounter the following regimes of operation:

- ▶ For $V_i \leq V_{Tn}$, nMOS is ‘off’.
- ▶ For low input voltage ($> V_{Tn}$), nMOS is saturated, pMOS is in linear regime.
- ▶ As V_i is raised further, V_o continues to fall. Eventually we enter the regime where nMOS is linear and pMOS is also linear.
- ▶ Finally, as we keep raising V_i , the output falls below V_{Tp} , provided the nMOS transistor is sufficiently wide. Now the nMOS is in linear regime, while pMOS is saturated.



Low input



- ▶ When the input voltage is less than V_{Tn} , the nMOS transistor is ‘off’.
- ▶ So the output is ‘high’ and no current is drawn from the supply.
- ▶ As we raise the input just above V_{Tn} , the output starts falling.
- ▶ In this region, the output voltage is just below V_{DD} . So the nMOS is saturated, while the pMOS is in linear regime.

nMOS saturated, pMOS linear

The input voltage is assumed to be sufficiently low so that the output voltage exceeds the saturation voltage $V_i - V_{Tn}$. Normally, this voltage will be higher than V_{Tp} , so the p channel transistor is in linear mode of operation.

Equating currents through the n and p channel transistors, we get

$$\frac{K_n}{2}(V_i - V_{Tn})^2 = K_p \left[(V_{DD} - V_{Tp})(V_{DD} - V_o) - \frac{1}{2}(V_{DD} - V_o)^2 \right]$$

defining $\beta \equiv K_n/K_p$, $V_1 \equiv V_{DD} - V_o$ and $V_2 \equiv V_{DD} - V_{Tp}$, we get

$$\frac{1}{2}V_1^2 - V_2 V_1 + \frac{\beta}{2}(V_i - V_{Tn})^2 = 0$$

This quadratic equation can be solved for V_1 , which gives the value of V_o .

nMOS saturated, pMOS linear

$$\frac{1}{2}V_1^2 - V_2 V_1 + \frac{\beta}{2}(V_i - V_{Tn})^2 = 0$$

The solutions are:

$$V_1 = V_2 \pm \sqrt{V_2^2 - \beta(V_i - V_{Tn})^2}$$

substituting the values of V_1 and V_2 and choosing the sign which puts V_o in the correct range, we get

$$V_o = V_{Tp} + \sqrt{(V_{DD} - V_{Tp})^2 - \beta(V_i - V_{Tn})^2}$$

Range of validity

$$V_o = V_{Tp} + \sqrt{(V_{DD} - V_{Tp})^2 - \beta(V_i - V_{Tn})^2}$$

- ▶ This equation is valid as long as the nMOS is in saturation and pMOS is in its linear regime.
- ▶ pMOS will enter saturation when $V_o \leq V_{Tp}$, and this will happen when

$$V_{DD} - V_{Tp} = \sqrt{\beta}(V_i - V_{Tn}) \quad \text{or} \quad V_i = V_{Tn} + \frac{V_{DD} - V_{Tp}}{\sqrt{\beta}}$$

- ▶ For example, for $V_{DD} = 1.8V$, $V_{Tn} = V_{Tp} = 0.4$ and $\beta = 4$,

$$V_{Tn} + \frac{V_{DD} - V_{Tp}}{\sqrt{\beta}} = 1.1V$$

nMOS linear, pMOS linear

- ▶ nMOS will enter linear regime when the output falls to $V_i - V_{Tn}$.
- ▶ The output voltage will fall to $V_i - V_{Tn}$ when

$$V_o = V_i - V_{Tn} = V_{Tp} + \sqrt{(V_{DD} - V_{Tp})^2 - \beta(V_i - V_{Tn})^2}$$

- ▶ This can be solved to give the condition:

$$V_i > V_{Tn} + \frac{V_{Tp} + \sqrt{V_{Tp}^2 + (\beta + 1)V_{DD}(V_{DD} - 2V_{Tp})}}{\beta + 1}$$

- ▶ For $V_{DD} = 1.8V$, $V_{Tn} = V_{Tp} = 0.4V$ and $\beta = 4$, the input voltage which will cause the nMOS to enter linear region is: 1.085V. The output voltage at this input is 0.685V.

nMOS linear, pMOS linear

To determine the output voltage when both transistors are in their linear regimes, We can again equate the nMOS and pMOS currents.

$$I_{dn} = K_n \left[(V_i - V_{Tn})V_o - \frac{1}{2}V_o^2 \right]$$

$$I_{dp} = K_p \left[(V_{DD} - V_{Tp})(V_{DD} - V_o) - \frac{1}{2}(V_{DD} - V_o)^2 \right]$$

Equating the two and defining $\beta \equiv K_n/K_p$, we can write

$$\beta \left[(V_i - V_{Tn})V_o - \frac{1}{2}V_o^2 \right] =$$

$$V_{DD}^2 - V_{Tp}V_{DD} - V_oV_{DD} + V_oV_{Tp} - \frac{1}{2}(V_{DD}^2 + V_o^2 - 2V_{DD}V_o)$$

This leads to the quadratic equation

$$\frac{\beta - 1}{2}V_o^2 - (\beta(V_i - V_{Tn}) - V_{Tp})V_o + \frac{V_{DD}}{2}(V_{DD} - 2V_{Tp}) = 0$$

nMOS linear, pMOS linear

When both transistors are in their linear regime,

$$\frac{\beta - 1}{2} V_o^2 - (\beta(V_i - V_{Tn}) - V_{Tp})V_o + \frac{V_{DD}}{2}(V_{DD} - 2V_{Tp}) = 0$$

We can solve this quadratic equation to get the value of V_o as

$$\frac{\beta(V_i - V_{Tn}) - V_{Tp} - \sqrt{(\beta(V_i - V_{Tn}) - V_{Tp})^2 - (\beta - 1)V_{DD}(V_{DD} - 2V_{Tp})}}{\beta - 1}$$

As the input voltage is raised still further, the output voltage will fall to V_{Tp} . The pMOS transistor will then enter the saturation regime.

nMOS linear, pMOS saturated

As the input voltage is raised still further, the output voltage will fall to V_{Tp} . The pMOS transistor is now in saturation regime.
Equating currents, we get

$$K_n \left[(V_i - V_{Tn})V_o - \frac{1}{2}V_o^2 \right] = \frac{K_p}{2}(V_{DD} - V_{Tp})^2$$

which gives

$$\frac{1}{2}V_o^2 - (V_o - V_{Tn})V_o + \frac{(V_{DD} - V_{Tp})^2}{2\beta}$$

This can be solved to get

$$V_o = (V_i - V_{Tn}) - \sqrt{(V_i - V_{Tn})^2 - (V_{DD} - V_{Tp})^2 / \beta}$$

Noise Margins

We find points on the transfer curve where the slope is -1.
When the input is low and output high, we should use

$$V_o = V_{Tp} + \sqrt{(V_{DD} - V_{Tp})^2 - \beta(V_i - V_{Tn})^2}$$

Differentiating this equation with respect to V_i and setting the slope to -1, we get

$$V_{iL} = V_{Tn} + \frac{V_{DD} - V_{Tp}}{\sqrt{\beta(\beta + 1)}}$$

and

$$V_{oH} = V_{Tp} + \sqrt{\frac{\beta}{\beta + 1}} (V_{DD} - V_{Tp})$$

When the input is high and the output low, we use

$$V_o = (V_i - V_{Tn}) - \sqrt{(V_i - V_{Tn})^2 - (V_{DD} - V_{Tp})^2 / \beta}$$

Differentiating with respect to V_i and setting the slope to -1, we get

$$V_{iH} = V_{Tn} + \frac{2}{\sqrt{3\beta}} (V_{DD} - V_{Tp})$$

and

$$V_{oL} = \frac{(V_{DD} - V_{Tp})}{\sqrt{3\beta}}$$

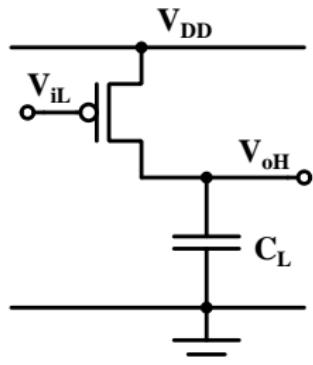
Ratioed Logic

To make the output ‘low’ value lower than V_{Tn} , we get the condition

$$\beta > \frac{1}{3} \left(\frac{V_{DD} - V_{Tp}}{V_{Tn}} \right)^2$$

- ▶ This places a requirement on the ratios of widths of n and p channel transistors. The logic gates work properly only when this equation is satisfied.
- ▶ Therefore this kind of logic is also called ‘ratioed logic’.
- ▶ In contrast, CMOS logic is called ratioless logic because it does not place any restriction on the ratios of widths of n and p channel transistors for static operation.
- ▶ The noise margin for pseudo nMOS can be determined easily from the expressions for V_{iL} , V_{oL} , V_{iH} , V_{oH} .

Rise Time

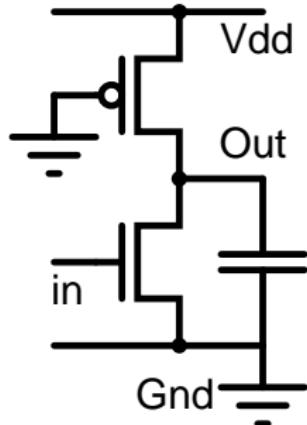


When the input is low, the nMOS is off and the output rises from 'low' to 'high'. The situation is identical to the charge up condition of a CMOS gate with the pMOS being biased with its gate at 0V.

This gives

$$\tau_{rise} = \frac{C}{K_p(V_{DD} - V_{Tp})} \left[\frac{2V_{Tp}}{V_{DD} - V_{Tp}} + \ln \frac{V_{DD} + V_{oH} - 2V_{Tp}}{V_{DD} - V_{oH}} \right]$$

Fall Time



Calculation of fall time is complicated by the fact that the pMOS load continues to dump current in the output node, even as the nMOS tries to discharge the output capacitor.

The nMOS needs to sink the discharge current as well as the drain current of the pMOS transistor.

Simplifying assumption:

pMOS current remains constant at its saturation value through the entire discharge process.

(This will result in a slightly pessimistic value of discharge time).

Fall Time

If we assume that the pMOS current remains constant at its saturation value,

$$I_p = \frac{K_p}{2} (V_{DD} - V_{Tp})^2$$

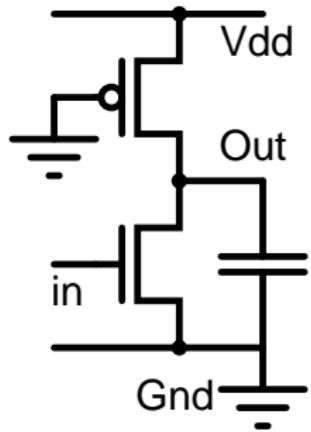
. We can write the KCL equation at the output node as:

$$I_n - I_p + C \frac{dV_o}{dt} = 0$$

which gives

$$\frac{\tau_{fall}}{C} = - \int_{V_{DD}}^{V_{oL}} \frac{dV_o}{I_n - I_p}$$

We define $V_1 \equiv V_i - V_{Tn}$.



The integration range can be divided into two regimes.

- ▶ nMOS is saturated when $V_1 \leq V_o < V_{DD}$.
- ▶ It is in the linear regime when $V_{oL} < V_o < V_1$.

Fall Time

$$\frac{\tau_{fall}}{C} = - \int_{V_{DD}}^{V_1} \frac{dV_o}{\frac{1}{2}K_n V_1^2 - I_p} - \int_{V_1}^{V_{oL}} \frac{dV_o}{K_n(V_1 V_o - \frac{1}{2}V_o^2) - I_p}$$

$$\text{So } \frac{K_n \tau_{fall}}{2C} = \frac{V_{DD} - V_1}{V_1^2 - 2I_p/K_n} + \int_{V_{oL}}^{V_1} \frac{dV_o}{2V_1 V_o - V_o^2 - 2I_p/K_n}$$

The integral can be evaluated using partial fractions.

We define $V_2^2 \equiv 2I_p/K_n$. The denominator of the integral term can then be factorized by adding and subtracting V_1^2 .

$$2V_1 V_o - V_o^2 - V_1^2 + V_1^2 - V_2^2 = -(V_1 - V_o)^2 + \left(\sqrt{V_1^2 - V_2^2} \right)^2$$

$$= \left(\sqrt{V_1^2 - V_2^2} + V_1 - V_o \right) \left(\sqrt{V_1^2 - V_2^2} - V_1 + V_o \right)$$

Fall Time

$$\frac{K_n \tau_{fall}}{2C} = \frac{V_{DD} - V_1}{V_1^2 - 2I_p/K_n} + \int_{V_{oL}}^{V_1} \frac{dV_o}{2V_1 V_o - V_o^2 - V_2^2}$$

The denominator of the integral term can be factorized as

$$\left(\sqrt{V_1^2 - V_2^2} + V_1 - V_o \right) \left(\sqrt{V_1^2 - V_2^2} - V_1 + V_o \right)$$

The integral term can then be written as:

$$\frac{1}{2\sqrt{V_1^2 - V_2^2}} \left(\int_{V_{oL}}^{V_1} \frac{dV_o}{V_1 - V_o + \sqrt{V_1^2 - V_2^2}} + \int_{V_{oL}}^{V_1} \frac{dV_o}{V_o - V_1 + \sqrt{V_1^2 - V_2^2}} \right)$$

Fall Time

Integrating over V_o gives

$$\frac{1}{2\sqrt{V_1^2 - V_2^2}} \ln \left. \frac{V_o - V_1 + \sqrt{V_1^2 - V_2^2}}{V_1 - V_o + \sqrt{V_1^2 - V_2^2}} \right|_{V_{oL}}^{V_1}$$

On putting the limits for the integral, the upper limit gives 0. So the integral can be written as

$$\frac{1}{2\sqrt{V_1^2 - V_2^2}} \ln \frac{V_1 - V_{oL} + \sqrt{V_1^2 - V_2^2}}{V_{oL} - V_1 + \sqrt{V_1^2 - V_2^2}}$$

Fall Time

Thus we can write down the expression for fall time as:

$$\frac{K_n \tau_{fall}}{2C} = \frac{V_{DD} - V_1}{V_1^2 - 2I_p/K_n} + \frac{1}{2\sqrt{V_1^2 - V_2^2}} \ln \frac{V_1 - V_{oL} + \sqrt{V_1^2 - V_2^2}}{V_{oL} - V_1 + \sqrt{V_1^2 - V_2^2}}$$

Substituting back for V_2^2 , we get

$$\frac{K_n \tau_{fall}}{2C} = \frac{V_{DD} - V_1}{V_1^2 - 2I_p/K_n} + \frac{1}{2\sqrt{V_1^2 - 2I_p/K_n}} \ln \frac{V_1 - V_{oL} + \sqrt{V_1^2 - 2I_p/K_n}}{V_{oL} - V_1 + \sqrt{V_1^2 - 2I_p/K_n}}$$

Since $I_p = K_p(V_{DD} - V_{Tp})^2/2$, $2I_p/K_n$ is just $(V_{DD} - V_{Tp})^2/\beta$.

Fall Time

$$\frac{K_n \tau_{fall}}{2C} = \frac{V_{DD} - V_1}{V_1^2 - 2I_p/K_n} + \frac{1}{2\sqrt{V_1^2 - 2I_p/K_n}} \ln \frac{V_1 - V_{oL} + \sqrt{V_1^2 - 2I_p/K_n}}{V_{oL} - V_1 + \sqrt{V_1^2 - 2I_p/K_n}}$$
$$2I_p/K_n = \frac{V_{DD} - V_{Tp}}{\beta} \quad \text{and} \quad V_1 = V_i - V_{Tn}$$

This relation was derived using the pessimistic assumption that the p channel transistor dumps its saturation current over the entire discharge range.

In any case, this relation is not used for designing the inverter because the limit on the size of the n channel transistor is put by static considerations and not by the fall time.

Pseudo nMOS Inverter design

- ▶ We design the basic inverter and then scale device sizes based on the logic function being designed.
- ▶ The load device size is calculated from the rise time.

$$\tau_{rise} = \frac{C}{K_p(V_{DD} - V_{Tp})} \left[\frac{2V_{Tp}}{V_{DD} - V_{Tp}} + \ln \frac{V_{DD} + V_{oH} - 2V_{Tp}}{V_{DD} - V_{oH}} \right]$$

- ▶ Since $K_p = \mu_p C_{ox} W_p / L_p$, Width of the p channel transistor is given by

$$\frac{L_p C}{\mu_p C_{ox} \tau_{rise} (V_{DD} - V_{Tp})} \left[\frac{2V_{Tp}}{V_{DD} - V_{Tp}} + \ln \frac{V_{DD} + V_{oH} - 2V_{Tp}}{V_{DD} - V_{oH}} \right]$$

- ▶ Given a value of τ_{rise} , operating voltages and technological constants, K_p and hence, the geometry of the p channel transistor can be determined.

Pseudo nMOS Inverter design

- ▶ Geometry of the n channel transistor is determined from static considerations.

$$V_{oL} = (V_{iH} - V_{Tn}) - \sqrt{(V_{iH} - V_{Tn})^2 - (V_{DD} - V_{Tp})^2 / \beta}$$

- ▶ Let V_n be the desired static low noise margin. We take $V_{oL} = V_{Tn} - V_n$, and calculate β . (This is conservative. We could have taken V_{oL} to be $(V_{DD} - V_{Tp})/\sqrt{3\beta} - V_n$.
- ▶ The only unknown in the above relation is β . So $\beta \equiv K_n/K_p$ can be evaluated using it.
- ▶ Since K_p has already been evaluated, knowing β , we can evaluate K_n and hence, the geometry of the n channel transistor.

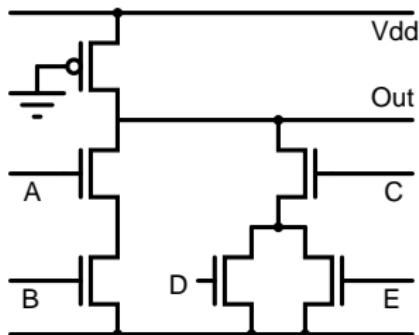
Conversion to other logic

- ▶ Once the basic pseudo nMOS inverter is designed, other logic gates can be derived from it.
- ▶ The procedure is the same as that for CMOS, except that it is applied only to nMOS transistors.
- ▶ The p channel transistor is kept at the same size as that for an inverter.
- ▶ Series-parallel rules are applied to determine the geometry of n channel transistors.

Conversion to other logic

- ▶ The logic is expressed as a sum of products with a bar (inversion) on top.
- ▶ For every ‘.’ in the expression, we put the corresponding n channel transistors in series.
- ▶ For every ‘+’, we put the n channel transistors in parallel. We scale the transistor widths up by the number of devices put in series.
- ▶ The geometries are left untouched for devices put in parallel.

$A \cdot B + C \cdot (D + E)$ in pseudo-nMOS



- ▶ A and B are in series.
- ▶ The pair is in parallel with C which is in series with a parallel combination of D and E .

Implementation of $A \cdot B + C \cdot (D + E)$ in pseudo-nMOS logic design style.

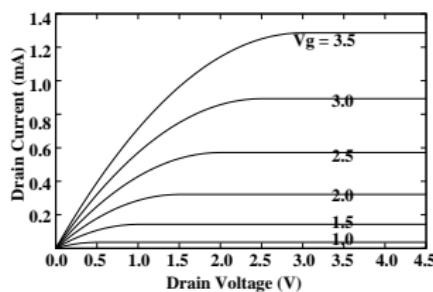
Design of Logic Gates in CMOS Technology

Dinesh Sharma

EE Department
IIT Bombay, Mumbai

September 19, 2020

A simple model



$K \equiv \mu C_{ox} W/L$ and V_T is the threshold voltage.
for $V_{gs} \leq V_T$, $I_{ds} = 0$

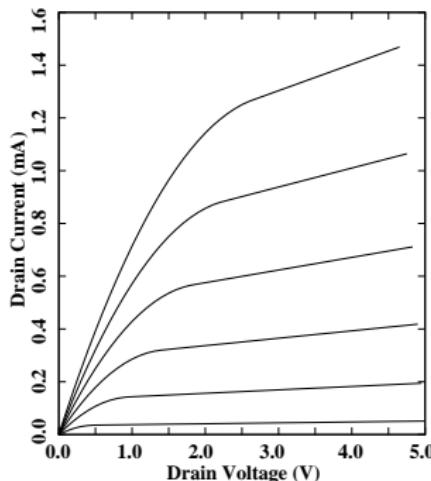
for $V_{gs} > V_T$ and $V_{ds} \leq V_{gs} - V_T$,
 $I_{ds} = K [(V_{gs} - V_T)V_{ds} - \frac{1}{2}V_{ds}^2]$

for $V_{gs} > V_T$ and $V_{ds} > V_{gs} - V_T$,
 $I_{ds} = K \frac{(V_{gs} - V_T)^2}{2}$

This model assumes current to be independent of V_{ds} in the saturation region.

(This is somewhat oversimplified. Better models exist, but we shall use this one for analysing digital circuits.)

A more realistic model



Let 'Early Voltage' $\equiv V_E$

$$\begin{aligned} \text{define } V_{dss} &\equiv V_E \left(\sqrt{1 + \frac{2(V_{gs} - V_T)}{V_E}} - 1 \right) \\ &\simeq (V_{gs} - V_T) \left(1 - \frac{V_{gs} - V_T}{2V_E} \right) \\ \text{and } I_{dss} &\equiv K \left[(V_{gs} - V_T)V_{dss} - \frac{1}{2}V_{dss}^2 \right] \end{aligned}$$

$$\text{for } V_{gs} > V_T \text{ and } V_{ds} \leq V_{dss} \quad I_{ds} = K \left[(V_{gs} - V_T)V_{ds} - \frac{1}{2}V_{ds}^2 \right]$$

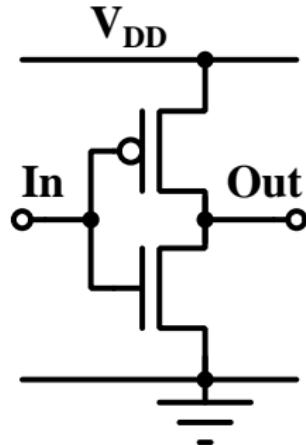
$$\text{for } V_{gs} > V_T \text{ and } V_{ds} > V_{dss} \quad I_{ds} = I_{dss} \frac{V_{ds} + V_E}{V_{dss} + V_E}$$

CMOS Static Logic

- Each logic stage contains pull up and pull down networks controlled by input signals.
- The pull up network contains p channel transistors.
- The pull down network is made of n channel transistors.
- If the pull up network is ‘on’, the pull down network is ‘off’ and *vice versa*.
- Since the pull up and pull down networks are never ‘on’ simultaneously, there is no static power consumption.

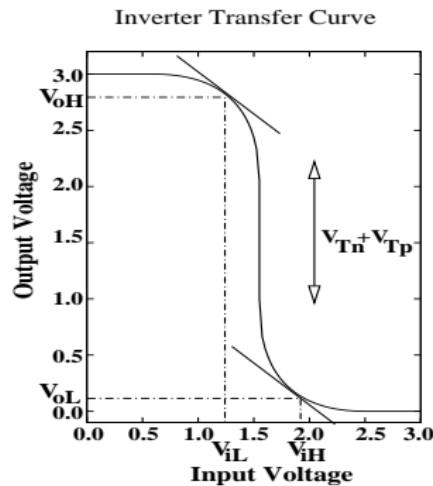
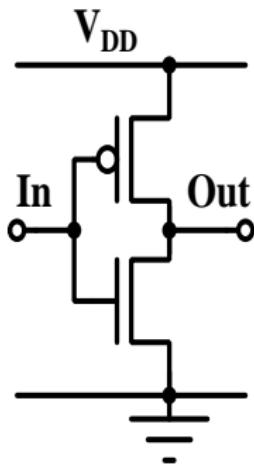
CMOS Inverter

The simplest of CMOS logic structure is the inverter. This is the basic gate of CMOS style of design. More complex gates are designed by mapping them to an 'equivalent' inverter.



- The pull up network of the logic gate is made equivalent to the pMOS of the inverter.
- The pull down network of the logic gate is made equivalent to the nMOS of the inverter.
- Thumb rules are used to map the geometries of the pull up and pull down networks to single transistors.

Static Characteristics

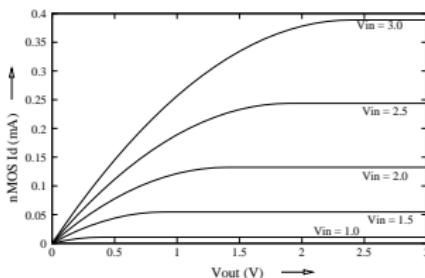
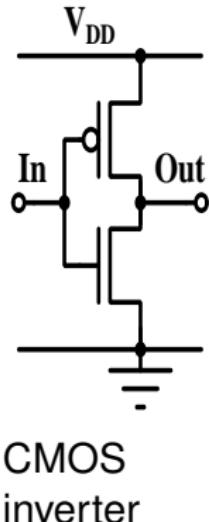


The range of input voltages can be divided into several regions.

- nMOS ‘off’, pMOS ‘on’
- nMOS saturated, pMOS linear
- nMOS saturated, pMOS saturated
- nMOS linear, pMOS saturated
- nMOS ‘on’, pMOS ‘off’

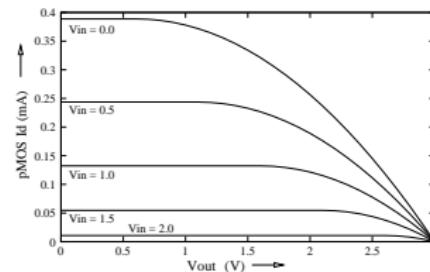
Transistor Currents

As we sweep the input voltage from 0 to V_{DD} , the current through the nMOS and pMOS can be plotted as a function of V_{in} and V_{out} .



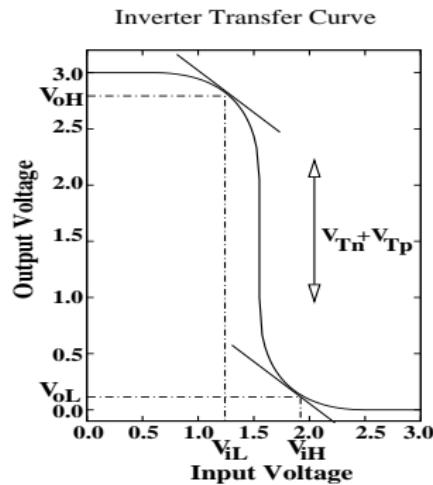
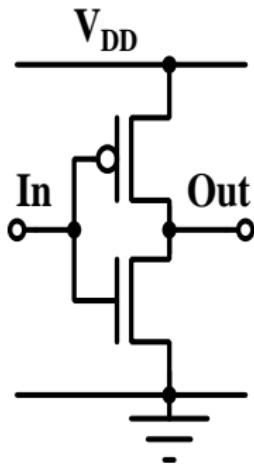
The gate voltage for the nMOS transistor = V_{in} and the drain voltage = V_{out} .

The output voltage for the inverter for a given input voltage is the V_{out} value where the current through the two transistors is equal.



The absolute gate voltage for pMOS is $V_{DD} - V_{in}$ and the absolute drain voltage is $V_{DD} - V_{out}$.

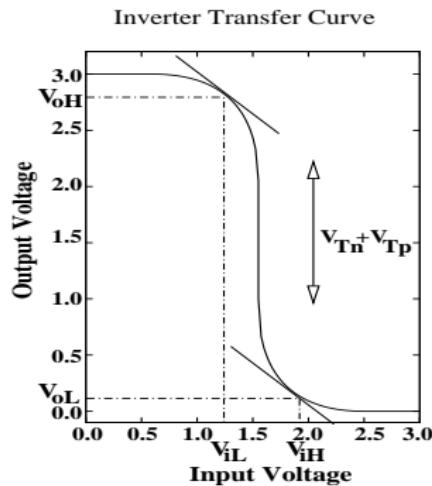
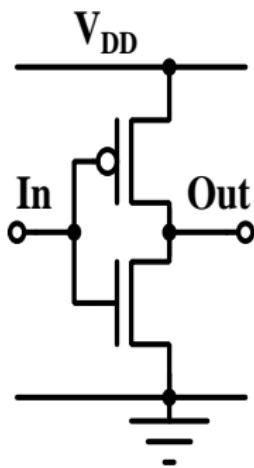
nMOS ‘off’, pMOS ‘on’



For $0 < V_i < V_{Tn}$

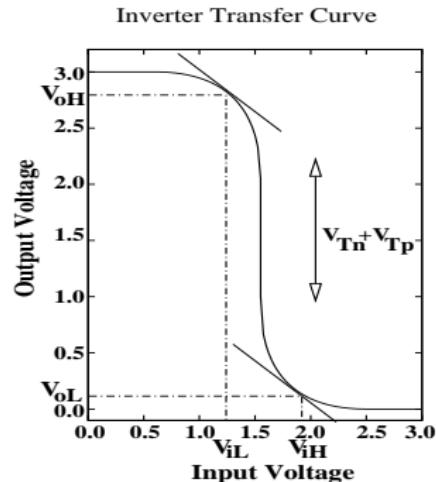
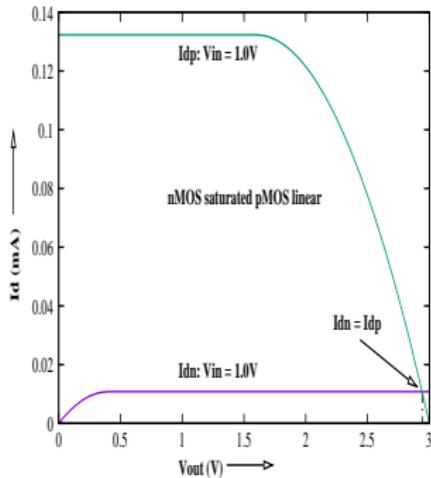
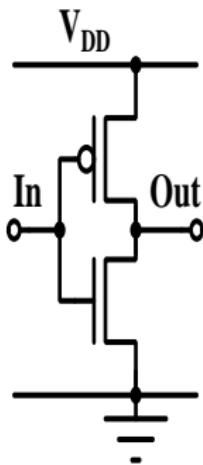
- the n channel transistor is ‘off’,
- the p channel transistor is ‘on’ and the output voltage = V_{DD} .
- This is the normal digital operation range with input = ‘0’ and output = ‘1’.

nMOS saturated, pMOS linear



- In this regime, both transistors are 'on'.
- The input voltage V_i is $> V_{Tn}$, but is small enough so that the n channel transistor is in saturation, and the p channel transistor is in the linear regime.
- In static condition, the output voltage will adjust itself such that the currents through the n and p channel transistors are equal.

nMOS saturated, pMOS linear



- The absolute value of gate-source voltage on the p channel transistor is $V_{DD} - V_i$, and therefore the “over voltage” on its gate is $V_{DD} - V_i - V_{Tp}$.
- The drain source voltage of the pMOS has an absolute value $V_{DD} - V_o$.

nMOS saturated, pMOS linear

$$\begin{aligned} I_{dp} &= K_p \left[(V_{DD} - V_i - V_{Tp})(V_{DD} - V_o) - \frac{1}{2}(V_{DD} - V_o)^2 \right] \\ I_{dn} &= \frac{K_n}{2}(V_i - V_{Tn})^2 \end{aligned}$$

Where symbols have their usual meanings. Equating currents, we get

$$K_p \left[(V_{DD} - V_i - V_{Tp})(V_{DD} - V_o) - \frac{1}{2}(V_{DD} - V_o)^2 \right] = \frac{K_n}{2}(V_i - V_{Tn})^2$$

We define $\beta \equiv K_n/K_p$ and $V_{dp} \equiv V_{DD} - V_o$. This gives

$$(V_{DD} - V_i - V_{Tp})V_{dp} - \frac{1}{2}V_{dp}^2 = \frac{\beta}{2}(V_i - V_{Tn})^2$$

nMOS saturated, pMOS linear

$$\frac{1}{2}V_{dp}^2 - (V_{DD} - V_i - V_{Tp})V_{dp} + \frac{\beta}{2}(V_i - V_{Tn})^2 = 0$$

Solving the quadratic equation above we get V_{dp} , from which V_o may be written as:

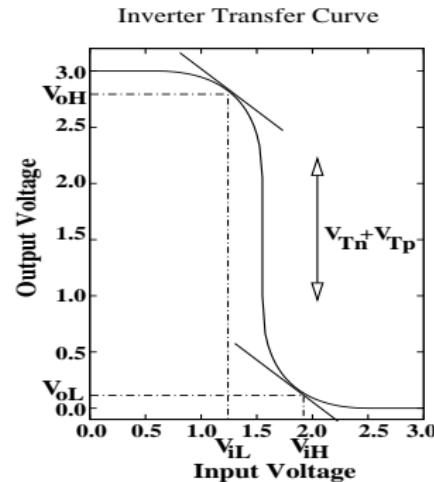
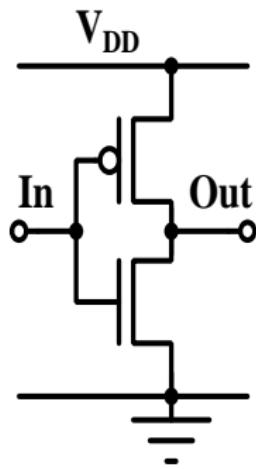
$$V_o = V_i + V_{Tp} + \sqrt{(V_{DD} - V_i - V_{Tp})^2 - \beta(V_i - V_{Tn})^2}$$

If $K_n = K_p$; ($\beta = 1$),

$$V_o = (V_i + V_{Tp}) + \sqrt{(V_{DD} - V_{Tn} - V_{Tp})(V_{DD} - 2V_i + V_{Tn} - V_{Tp})}$$

$$\text{for } V_i \leq \frac{V_{DD} + V_{Tn} - V_{Tp}}{2}$$

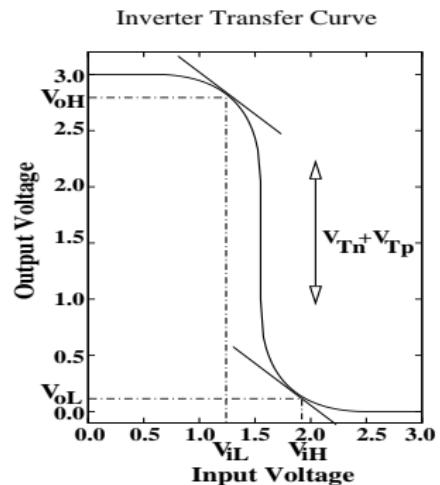
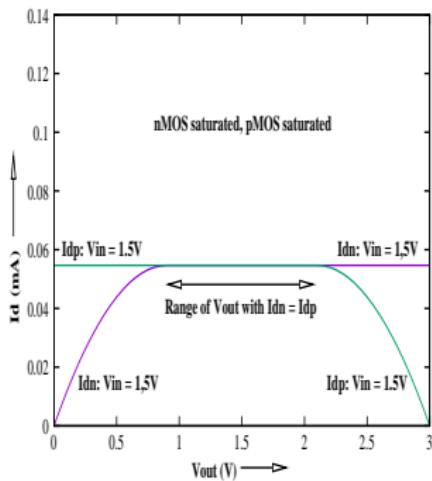
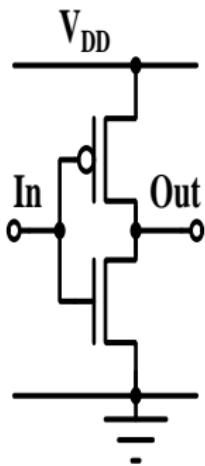
nMOS saturated, pMOS saturated



When the input voltage is $V_i = (V_{DD} + \sqrt{\beta}V_{Tn} - V_{Tp})/(1 + \sqrt{\beta})$, both transistors are saturated.

- Currents of both transistors are independent of their drain voltages.
- we do not get a unique solution for V_o by equating drain currents.

nMOS saturated, pMOS saturated



nMOS saturated, pMOS saturated

When the input voltage is $V_i = (V_{DD} + \sqrt{\beta}V_{Tn} - V_{Tp})/(1 + \sqrt{\beta})$, both transistors are saturated.

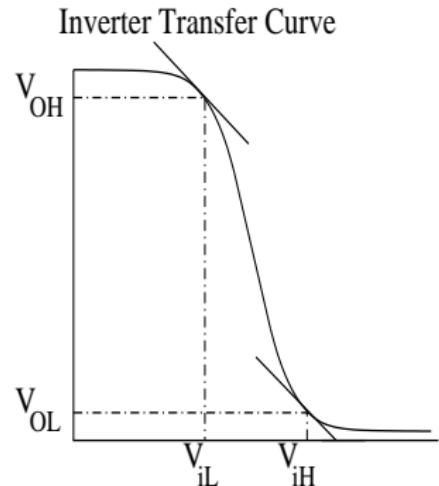
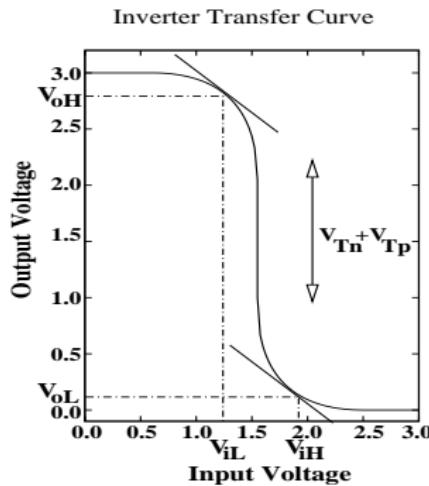
- Currents of both transistors are independent of their drain voltages.
- we do not get a unique solution for V_o by equating drain currents.
- The currents will be equal for all values of V_o in the range
$$V_i - V_{Tn} \leq V_o \leq V_i + V_{Tp}$$

All output voltages in the range $V_i - V_{Tn} \leq V_o \leq V_i + V_{Tp}$ satisfy the drain current equations at this input voltage.

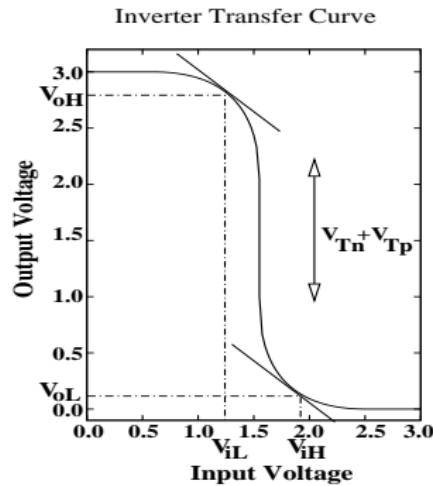
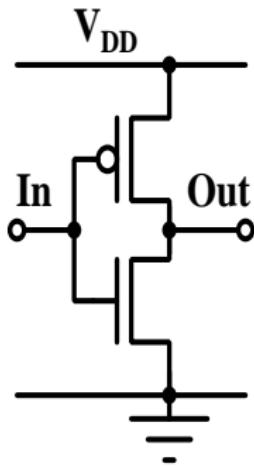
So the transfer curve of an inverter shows a drop of $V_{Tn} + V_{Tp}$ at a voltage near $V_{DD}/2$.

nMOS saturated, pMOS saturated

The sudden drop in V_o for $V_i = (V_{DD} + \sqrt{\beta}V_{Tn} - V_{Tp})/(1 + \sqrt{\beta})$ is an artifact of the simple model we have chosen for transistor currents. In reality, drain currents are weakly dependent on drain-source voltage. The transfer curve then has no discontinuity.



nMOS linear, pMOS saturated

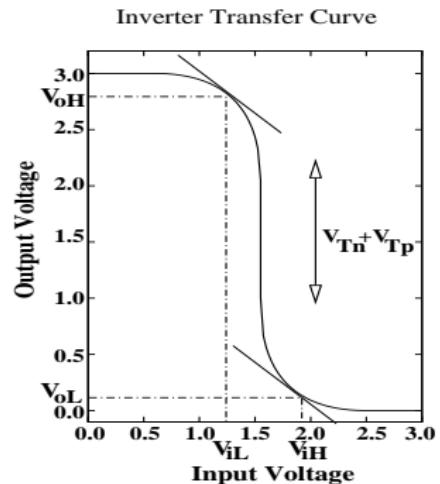
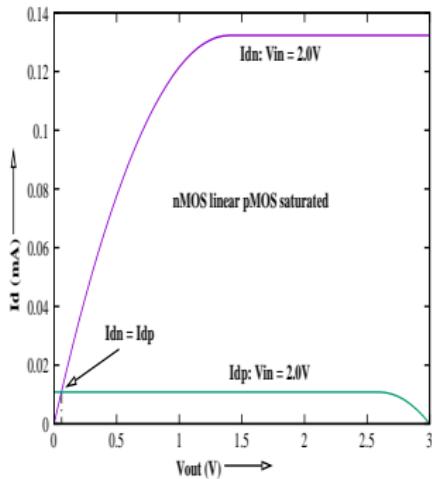
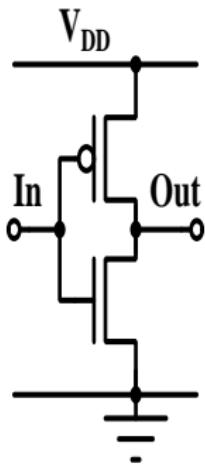


As we increase V_i further, so that

$$\frac{V_{DD} + \sqrt{\beta}V_{Tn} - V_{Tp}}{1 + \sqrt{\beta}} < V_i < V_{DD} - V_{Tp}$$

both transistors are still 'on', but nMOS enters the linear regime while pMOS is saturated.

nMOS linear, pMOS saturated



nMOS linear, pMOS saturated

Equating currents when nMOS is saturated and pMOS is in linear regime,

$$\begin{aligned} I_d &= \frac{K_p}{2}(V_{DD} - V_i - V_{Tp})^2 \\ &= K_n \left[(V_i - V_{Tn})V_o - \frac{1}{2}V_o^2 \right] \end{aligned}$$

From this, we get the quadratic equation

$$\frac{1}{2}V_o^2 - (V_i - V_{Tn})V_o + \frac{(V_{DD} - V_i - V_{Tp})^2}{2\beta} = 0$$

with solutions:

$$V_o = (V_i - V_{Tn}) \pm \sqrt{(V_i - V_{Tn})^2 - \frac{(V_{DD} - V_i - V_{Tp})^2}{\beta}}$$

$$V_o = (V_i - V_{Tn}) \pm \sqrt{(V_i - V_{Tn})^2 - \frac{(V_{DD} - V_i - V_{Tp})^2}{\beta}}$$

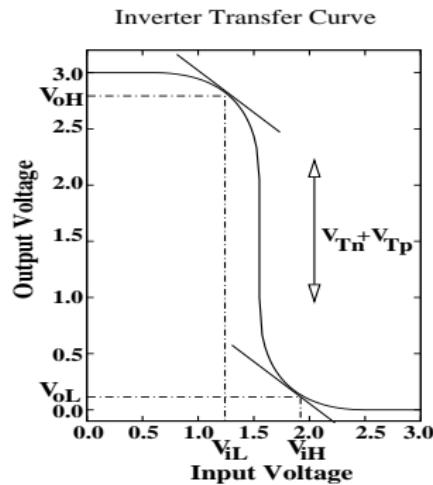
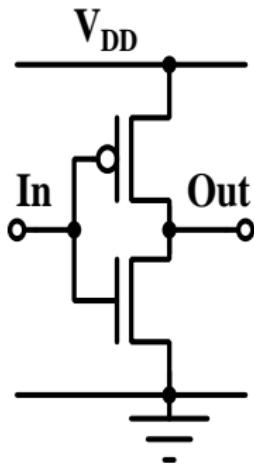
We must choose the negative sign, since $V_o \leq V_i - V_{Tn}$ for nMOS to be in linear regime. So,

$$V_o = (V_i - V_{Tn}) - \sqrt{(V_i - V_{Tn})^2 - \frac{(V_{DD} - V_i - V_{Tp})^2}{\beta}}$$

In the special case where $\beta = 1$, we have

$$V_o = (V_i - V_{Tn}) - \sqrt{(V_{DD} - V_{Tn} - V_{Tp})(2V_i - V_{DD} - V_{Tn} + V_{Tp})}$$

nMOS 'on', pMOS 'off'



- As we increase the input voltage beyond $V_{DD} - V_{Tp}$, the p channel transistor turns 'off', while the n channel conducts strongly.
- As a result, the output voltage falls to zero.
- This is the normal digital operation range with input = '1' and output = '0'.

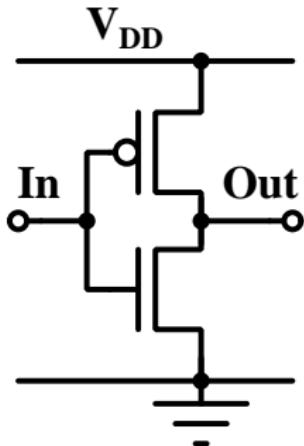
Noise Margins

- For robust design, the output levels must be interpreted correctly at the input of next stage even in the presence of noise.
- For the ‘high’ level, we require that the output of one stage should still be interpreted as ‘high’ at the input of the next gate even when pulled down a little due to noise.
- Therefore V_{oH} should be $> V_{iH}$.
- Similarly V_{oL} should be $< V_{iL}$
- The difference, $V_{iL} - V_{oL}$ is the ‘low’ noise margin. and $V_{oH} - V_{iH}$ is the ‘high’ noise level.

Logic Levels

- A digital circuit should distinguish logic levels, but be insensitive to the exact analog voltage at the input.
- Therefore flat portions of the transfer curve (where $\frac{\partial V_o}{\partial V_i}$ is small) are suitable for digital logic.
- We select two points on the transfer curve where the slope ($\frac{\partial V_o}{\partial V_i}$) is -1.0.
- The coordinates of these two points define the values of (V_{iL}, V_{oH}) and (V_{iH}, V_{oL}) .
- The region to the left of V_{iL} and to the right of V_{iH} has $|\frac{\partial V_o}{\partial V_i}| < 1$, and is suitable for digital operation.

Calculation of Noise Margins



- To evaluate the values of noise margins, we shall use the expressions derived for $\beta = 1$ to keep the algebra simple.
- When the input is low and output high, the n channel transistor is saturated and the p channel transistor is in its linear regime.
- When the input is high and the output is low, the n channel transistor is in its linear regime, while the p channel transistor is saturated.

Calculation of V_{iL} and V_{oH}

for (V_{iL}, V_{oH}) , n channel transistor is saturated, while the p channel transistor is in its linear regime.

$$V_o = (V_i + V_{Tp}) + \sqrt{(V_{DD} - V_{Tn} - V_{Tp})(V_{DD} + V_{Tn} - V_{Tp} - 2V_i)}$$

From this, we evaluate $\frac{\partial V_o}{\partial V_i}$ and set it = -1.

$$\frac{\partial V_o}{\partial V_i} = -1 = 1 - \sqrt{\frac{V_{DD} - V_{Tn} - V_{Tp}}{V_{DD} + V_{Tn} - V_{Tp} - 2V_i}}$$

This gives

$$V_{iL} = \frac{3V_{DD} + 5V_{Tn} - 3V_{Tp}}{8}$$

$$V_{oH} = \frac{7V_{DD} + V_{Tn} + V_{Tp}}{8} = V_{DD} - \frac{V_{DD} - V_{Tn} - V_{Tp}}{8}$$

Calculation of V_{iH} and V_{oL}

When the input is 'high', we should use the equation for nMOS linear and pMOS saturated.

$$V_o = (V_i - V_{Tn}) - \sqrt{(V_{DD} - V_{Tn} - V_{Tp})(2V_i - V_{DD} - V_{Tn} + V_{Tp})}$$

Differentiating with respect to V_i gives

$$\frac{\partial V_o}{\partial V_i} = -1 = 1 - \sqrt{\frac{V_{DD} - V_{Tn} - V_{Tp}}{2V_i - V_{DD} - V_{Tn} + V_{Tp}}}$$

From where, we get

$$V_{iH} = \frac{5V_{DD} + 3V_{Tn} - 5V_{Tp}}{8}$$

$$V_{oL} = \frac{V_{DD} - V_{Tn} - V_{Tp}}{8}$$

Calculation of Noise Margins

The ‘High’ noise margin is given by

$$V_{oH} - V_{iH} = \frac{V_{DD} - V_{Tn} + 3V_{Tp}}{4}$$

Similarly, the ‘Low’ noise margin is

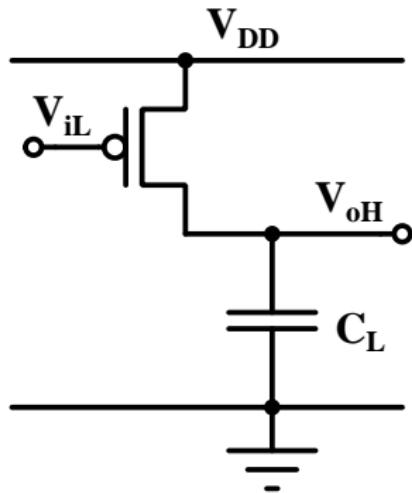
$$V_{iL} - V_{oL} = \frac{V_{DD} + 3V_{Tn} - V_{Tp}}{4}$$

The two noise margins can be made equal by choosing equal values for V_{Tn} and V_{Tp} .

Dynamic Characteristics

- For the calculation of rise and fall times, we shall assume that only one of the two transistors in the inverter is 'on'.
- This is more conservative than the static logic levels calculated by slope considerations.
- We shall use the simple model described at the beginning of this lecture.

Rise time



When the input is low, the n channel transistor is ‘off’, while the p channel transistor is ‘on’. From Kirchhoff’s current law at the output node,

$$I_{dp} = C \frac{dV_o}{dt}$$

so,

$$\frac{dt}{C} = \frac{dV_o}{I_{dp}}$$

Integrating both sides, we get

$$\frac{\tau_{rise}}{C} = \int_0^{V_{oH}} \frac{dV_o}{I_{dp}}$$

$$\frac{\tau_{rise}}{C} = \int_0^{V_{oH}} \frac{dV_o}{I_{dp}}$$

Till the output rises to $V_{iL} + V_{Tp}$, the p channel transistor is in saturation. If $V_{oH} > V_{iL} + V_{Tp}$ (which is normally the case), the integration range can be broken into saturation and linear regimes. Thus

$$\begin{aligned} \frac{\tau_{rise}}{C} &= \int_0^{V_{iL} + V_{Tp}} \frac{dV_o}{\frac{K_p}{2}(V_{DD} - V_{iL} - V_{Tp})^2} \\ &+ \int_{V_{iL} + V_{Tp}}^{V_{oH}} \frac{dV_o}{K_p [(V_{DD} - V_{iL} - V_{Tp})(V_{DD} - V_o) - \frac{1}{2}(V_{DD} - V_o)^2]} \end{aligned}$$

$$\begin{aligned}\tau_{rise} &= \frac{2C(V_{iL} + V_{Tp})}{K_p(V_{DD} - V_{iL} - V_{Tp})^2} \\ &+ \frac{C}{K_p(V_{DD} - V_{iL} - V_{Tp})} \ln \frac{V_{DD} + V_{oH} - 2V_{iL} - 2V_{Tp}}{V_{DD} - V_{oH}}\end{aligned}$$

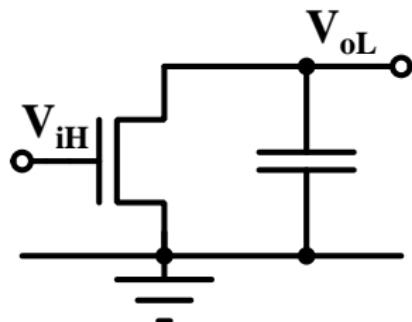
- The first term is just the constant current charging of the load capacitor.
- The second term represents the charging by the pMOS in its linear range.
- This can be compared with resistive charging, which would have taken a charge time of

$$\tau = RC \ln \frac{V_{DD} - V_{iL} - V_{Tp}}{V_{DD} - V_{oH}}$$

to charge from $V_{iL} + V_{Tp}$ to V_{oH} .

Fall Time

V_{DD}



When the input is high, the p channel transistor is ‘off’, while the n channel transistor is ‘on’. From Kirchhoff’s current law at the output node,

$$I_{dn} = -C \frac{dV_o}{dt}$$

Separating variables and integrating from the initial voltage ($= V_{DD}$) to some terminal voltage V_{oL} gives

$$\frac{\tau_{fall}}{C} = - \int_{V_{DD}}^{V_{oL}} \frac{dV_o}{I_{dn}}$$

Fall time

$$\frac{\tau_{fall}}{C} = - \int_{V_{DD}}^{V_{oL}} \frac{dV_o}{I_{dn}}$$

The n channel transistor will be in saturation till the output falls to $V_i - V_{Tn}$. Below this, the transistor will be in its linear regime. We can divide the integration range in two parts.

$$\begin{aligned}\frac{\tau_{fall}}{C} &= - \int_{V_{DD}}^{V_i - V_{Tn}} \frac{dV_o}{I_{dn}} - \int_{V_i - V_{Tn}}^{V_{oL}} \frac{dV_o}{I_{dn}} \\ &= \int_{V_i - V_{Tn}}^{V_{DD}} \frac{dV_o}{\frac{K_n}{2}(V_i - V_{Tn})^2} \\ &\quad + \int_{V_{oL}}^{V_i - V_{Tn}} \frac{dV_o}{K_n[(V_i - V_{Tn})V_o - \frac{1}{2}V_o^2]}\end{aligned}$$

Fall time

$$\frac{\tau_{fall}}{C} = \frac{V_{DD} - V_i + V_{Tn}}{\frac{K_n}{2}(V_i - V_{Tn})^2} + \frac{1}{K_n(V_i - V_{Tn})} \ln \frac{2(V_i - V_{Tn}) - V_{oL}}{V_{oL}}$$

The first term represents the time taken to discharge at constant current in the saturation regime, whereas the second term is the quasi-resistive discharge in the linear regime.

Trade off between power, speed and robustness

Noise margins are given by

$$\begin{aligned}V_{oH} - V_{iH} &= \frac{V_{DD} - V_{Tn} + 3V_{Tp}}{4} \\V_{iL} - V_{oL} &= \frac{V_{DD} + 3V_{Tn} - V_{Tp}}{4}\end{aligned}$$

- As we scale technologies, we improve speed and power consumption. However, the noise margin becomes worse.
- We can improve noise margins by choosing relatively higher threshold voltages. However, this will reduce speeds.
- We could also increase V_{DD} - but that would increase power dissipation.

Thus we have a trade off between power, speed and noise margins.

CMOS Inverter Design Flow

- A common design requirement is symmetric charge and discharge behaviour and equal noise margins for high and low logic values.
- This requires matched values of K_n and K_p and equal values of V_{Tn} and V_{Tp} .
- Rise and fall times depend linearly on K_n and K_p .
- Thus it is a straightforward calculation to determine transistor geometries if speed requirements and technological parameters are given.
- However, as transistor geometries are made larger, self loading can become significant.

CMOS Inverter Design Flow

- For large self-loading, we have to model the load capacitance as

$$C_{Load} = C_{ext} + \alpha K_n$$

where we have assumed that $\beta = K_n/K_p$ is constant. α is a technological constant.

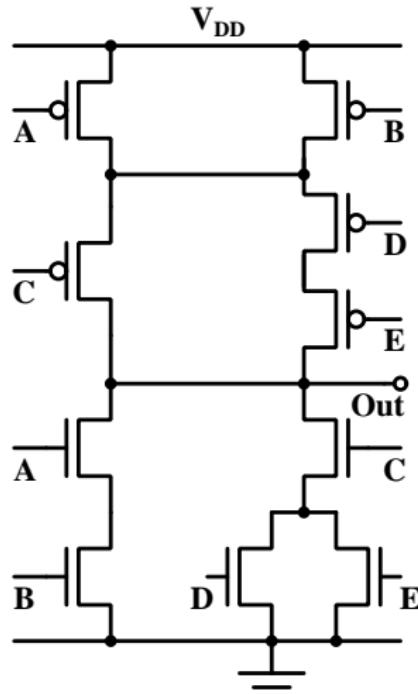
- We use the expressions for K_T/C which depend only on voltages. Once these values are calculated, the geometry can be determined.
- In the extreme case, when self capacitance dominates the load capacitance, K/C becomes constant and τ becomes geometry independent. There is no advantage in using wider transistors in this regime to increase the speed. It is better to use multi-stage logic with tapered buffers in this regime.

From Inverters to Other Logic

Once the basic CMOS inverter is designed, other logic gates can be derived from it. The logic has to be put in a canonical form which is a sum of products with a bar (inversion) on top.

- For every ‘.’ in the expression, we put the corresponding n channel transistors in series and the corresponding p channel transistors in parallel.
- for every ‘+’, we put the n channel transistors in parallel and the p channel transistors in series.
- We scale the transistor widths up by the number of devices (n or p) put in series.
- The geometries are left untouched for devices put in parallel.

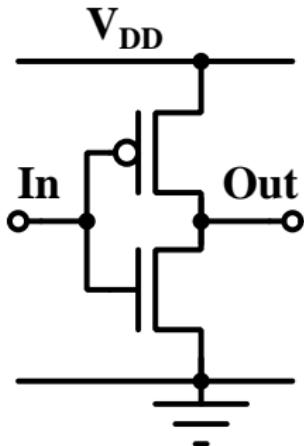
CMOS implementation of $A \cdot B + C \cdot (D + E)$



- For n channel, A and B are in series, The pair is in parallel with C which is in series with a parallel combination of D and E.
- For p channel, A is in parallel with B, the pair is in series with C which is in parallel with a series combination of D and E.

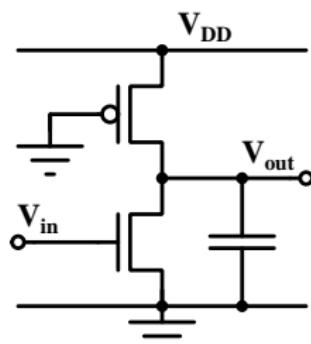
Implementation of $\overline{A} \cdot B + C \cdot (\overline{D} + E)$ in CMOS logic design style.

CMOS summary



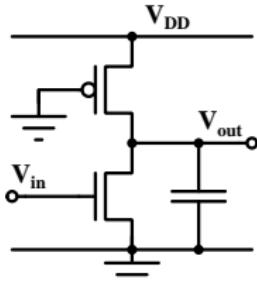
- Logic consumes no static power in CMOS design style.
- However, signals have to be routed to the n pull down network *as well as* to the p pull up network.
- So the load presented to every driver is high.
- This is exacerbated by the fact that n and p channel transistors cannot be placed close together as these are in different wells which have to be kept well separated in order to avoid latch-up.

Pseudo nMOS Design Style



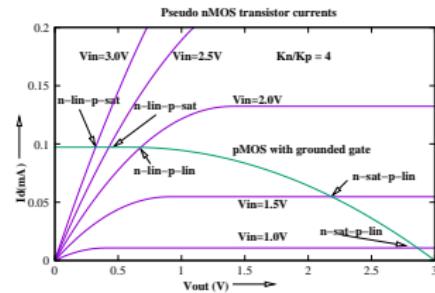
- The CMOS pull up network is replaced by a single pMOS transistor with its gate grounded.
- Since the pMOS is not driven by signals, it is always 'on'.
- The effective gate voltage seen by the pMOS transistor is V_{DD} . Thus the over-voltage on the p channel gate is always $V_{DD} - V_{Tp}$.
- When the nMOS is turned 'on', a direct path between supply and ground exists and static power will be drawn.
- However, the dynamic power is reduced due to lower capacitive loading

Static Characteristics

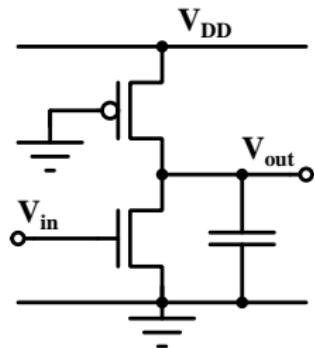


As we sweep the input voltage from ground to V_{DD} , we encounter the following regimes of operation:

- For $V_i \leq V_{Tn}$, nMOS is 'off'.
- For low input voltage ($> V_{Tn}$), nMOS is saturated, pMOS is in linear regime.
- As V_i is raised further, V_o continues to fall. Eventually we enter the regime where nMOS is linear and pMOS is also linear.
- Finally, as we keep raising V_i , the output falls below V_{Tp} , provided the nMOS transistor is sufficiently wide. Now the nMOS is in linear regime, while pMOS is saturated.



Low input



- When the input voltage is less than V_{Tn} , the nMOS transistor is ‘off’.
- So the output is ‘high’ and no current is drawn from the supply.
- As we raise the input just above V_{Tn} , the output starts falling.
- In this region, the output voltage is just below V_{DD} . So the nMOS is saturated, while the pMOS is in linear regime.

nMOS saturated, pMOS linear

The input voltage is assumed to be sufficiently low so that the output voltage exceeds the saturation voltage $V_i - V_{Tn}$.

Normally, this voltage will be higher than V_{Tp} , so the p channel transistor is in linear mode of operation.

Equating currents through the n and p channel transistors, we get

$$\frac{K_n}{2}(V_i - V_{Tn})^2 = K_p \left[(V_{DD} - V_{Tp})(V_{DD} - V_o) - \frac{1}{2}(V_{DD} - V_o)^2 \right]$$

defining $\beta \equiv K_n/K_p$, $V_1 \equiv V_{DD} - V_o$ and $V_2 \equiv V_{DD} - V_{Tp}$, we get

$$\frac{1}{2}V_1^2 - V_2 V_1 + \frac{\beta}{2}(V_i - V_{Tn})^2 = 0$$

This quadratic equation can be solved for V_1 , which gives the value of V_o .

nMOS saturated, pMOS linear

$$\frac{1}{2}V_1^2 - V_2V_1 + \frac{\beta}{2}(V_i - V_{Tn})^2 = 0$$

The solutions are:

$$V_1 = V_2 \pm \sqrt{V_2^2 - \beta(V_i - V_{Tn})^2}$$

substituting the values of V_1 and V_2 and choosing the sign which puts V_o in the correct range, we get

$$V_o = V_{Tp} + \sqrt{(V_{DD} - V_{Tp})^2 - \beta(V_i - V_{Tn})^2}$$

Range of validity

$$V_o = V_{Tp} + \sqrt{(V_{DD} - V_{Tp})^2 - \beta(V_i - V_{Tn})^2}$$

- This equation is valid as long as the nMOS is in saturation and pMOS is in its linear regime.
- pMOS will enter saturation when $V_o \leq V_{Tp}$, and this will happen when

$$V_{DD} - V_{Tp} = \sqrt{\beta}(V_i - V_{Tn}) \quad \text{or} \quad V_i = V_{Tn} + \frac{V_{DD} - V_{Tp}}{\sqrt{\beta}}$$

- For example, for $V_{DD} = 1.8V$, $V_{Tn} = V_{Tp} = 0.4$ and $\beta = 4$,

$$V_{Tn} + \frac{V_{DD} - V_{Tp}}{\sqrt{\beta}} = 1.1V$$

nMOS linear, pMOS linear

- nMOS will enter linear regime when the output falls to $V_i - V_{Tn}$.
- The output voltage will fall to $V_i - V_{Tn}$ when

$$V_o = V_i - V_{Tn} = V_{Tp} + \sqrt{(V_{DD} - V_{Tp})^2 - \beta(V_i - V_{Tn})^2}$$

- This can be solved to give the condition:

$$V_i > V_{Tn} + \frac{V_{Tp} + \sqrt{V_{Tp}^2 + (\beta + 1)V_{DD}(V_{DD} - 2V_{Tp})}}{\beta + 1}$$

- For $V_{DD} = 1.8V$, $V_{Tn} = V_{Tp} = 0.4V$ and $\beta = 4$, the input voltage which will cause the nMOS to enter linear region is: 1.085V. The output voltage at this input is 0.685V.

nMOS linear, pMOS linear

To determine the output voltage when both transistors are in their linear regimes, We can again equate the nMOS and pMOS currents.

$$I_{dn} = K_n \left[(V_i - V_{Tn})V_o - \frac{1}{2}V_o^2 \right]$$

$$I_{dp} = K_p \left[(V_{DD} - V_{Tp})(V_{DD} - V_o) - \frac{1}{2}(V_{DD} - V_o)^2 \right]$$

Equating the two and defining $\beta \equiv K_n/K_p$, we can write

$$\beta \left[(V_i - V_{Tn})V_o - \frac{1}{2}V_o^2 \right] =$$

$$V_{DD}^2 - V_{Tp}V_{DD} - V_oV_{DD} + V_oV_{Tp} - \frac{1}{2}(V_{DD}^2 + V_o^2 - 2V_{DD}V_o)$$

This leads to the quadratic equation

$$\frac{\beta - 1}{2}V_o^2 - (\beta(V_i - V_{Tn}) - V_{Tp})V_o + \frac{V_{DD}}{2}(V_{DD} - 2V_{Tp}) = 0$$

nMOS linear, pMOS linear

When both transistors are in their linear regime,

$$\frac{\beta - 1}{2} V_o^2 - (\beta(V_i - V_{Tn}) - V_{Tp})V_o + \frac{V_{DD}}{2}(V_{DD} - 2V_{Tp}) = 0$$

We can solve this quadratic equation to get the value of V_o as

$$\frac{\beta(V_i - V_{Tn}) - V_{Tp} - \sqrt{(\beta(V_i - V_{Tn}) - V_{Tp})^2 - (\beta - 1)V_{DD}(V_{DD} - 2V_{Tp})}}{\beta - 1}$$

As the input voltage is raised still further, the output voltage will fall to V_{Tp} . The pMOS transistor will then enter the saturation regime.

nMOS linear, pMOS saturated

As the input voltage is raised still further, the output voltage will fall to V_{Tp} . The pMOS transistor is now in saturation regime. Equating currents, we get

$$K_n \left[(V_i - V_{Tn})V_o - \frac{1}{2}V_o^2 \right] = \frac{K_p}{2}(V_{DD} - V_{Tp})^2$$

which gives

$$\frac{1}{2}V_o^2 - (V_o - V_{Tn})V_o + \frac{(V_{DD} - V_{Tp})^2}{2\beta}$$

This can be solved to get

$$V_o = (V_i - V_{Tn}) - \sqrt{(V_i - V_{Tn})^2 - (V_{DD} - V_{Tp})^2 / \beta}$$

Noise Margins

We find points on the transfer curve where the slope is -1.
When the input is low and output high, we should use

$$V_o = V_{Tp} + \sqrt{(V_{DD} - V_{Tp})^2 - \beta(V_i - V_{Tn})^2}$$

Differentiating this equation with respect to V_i and setting the slope to -1, we get

$$V_{iL} = V_{Tn} + \frac{V_{DD} - V_{Tp}}{\sqrt{\beta(\beta + 1)}}$$

and

$$V_{oH} = V_{Tp} + \sqrt{\frac{\beta}{\beta + 1}} (V_{DD} - V_{Tp})$$

When the input is high and the output low, we use

$$V_o = (V_i - V_{Tn}) - \sqrt{(V_i - V_{Tn})^2 - (V_{DD} - V_{Tp})^2 / \beta}$$

Differentiating with respect to V_i and setting the slope to -1, we get

$$V_{IH} = V_{Tn} + \frac{2}{\sqrt{3\beta}} (V_{DD} - V_{Tp})$$

and

$$V_{oL} = \frac{(V_{DD} - V_{Tp})}{\sqrt{3\beta}}$$

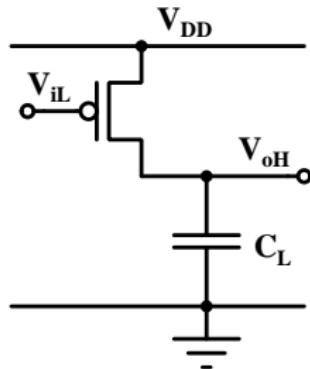
Ratioed Logic

To make the output ‘low’ value lower than V_{Tn} , we get the condition

$$\beta > \frac{1}{3} \left(\frac{V_{DD} - V_{Tp}}{V_{Tn}} \right)^2$$

- This places a requirement on the ratios of widths of n and p channel transistors. The logic gates work properly only when this equation is satisfied.
- Therefore this kind of logic is also called ‘ratioed logic’.
- In contrast, CMOS logic is called ratioless logic because it does not place any restriction on the ratios of widths of n and p channel transistors for static operation.
- The noise margin for pseudo nMOS can be determined easily from the expressions for V_{iL} , V_{oL} , V_{iH} , V_{oH} .

Rise Time

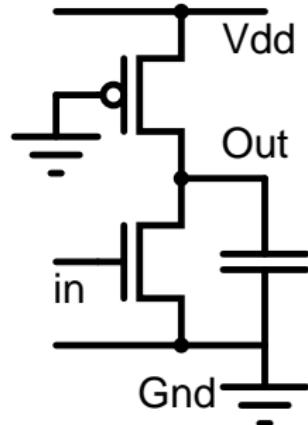


When the input is low, the nMOS is off and the output rises from 'low' to 'high'.
The situation is identical to the charge up condition of a CMOS gate with the pMOS being biased with its gate at 0V.

This gives

$$\tau_{rise} = \frac{C}{K_p(V_{DD} - V_{Tp})} \left[\frac{2V_{Tp}}{V_{DD} - V_{Tp}} + \ln \frac{V_{DD} + V_{oH} - 2V_{Tp}}{V_{DD} - V_{oH}} \right]$$

Fall Time



Calculation of fall time is complicated by the fact that the pMOS load continues to dump current in the output node, even as the nMOS tries to discharge the output capacitor.

The nMOS needs to sink the discharge current as well as the drain current of the pMOS transistor.

Simplifying assumption:

pMOS current remains constant at its saturation value through the entire discharge process.

(This will result in a slightly pessimistic value of discharge time).

Fall Time

If we assume that the pMOS current remains constant at its saturation value,

$$I_p = \frac{K_p}{2} (V_{DD} - V_{Tp})^2$$

. We can write the KCL equation at the output node as:

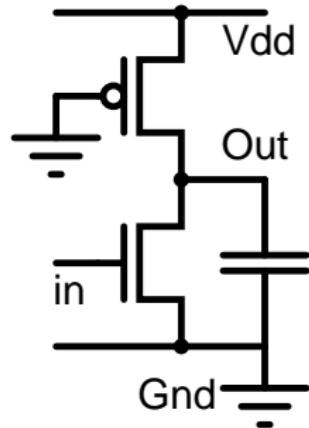
$$I_n - I_p + C \frac{dV_o}{dt} = 0$$

which gives

$$\frac{\tau_{fall}}{C} = - \int_{V_{DD}}^{V_{oL}} \frac{dV_o}{I_n - I_p}$$

We define $V_1 \equiv V_i - V_{Tn}$.

Fall Time



The integration range can be divided into two regimes.

- nMOS is saturated when $V_1 \leq V_o < V_{DD}$.
- It is in the linear regime when $V_{oL} < V_o < V_1$.

Fall Time

$$\frac{\tau_{fall}}{C} = - \int_{V_{DD}}^{V_1} \frac{dV_o}{\frac{1}{2}K_n V_1^2 - I_p} - \int_{V_1}^{V_{oL}} \frac{dV_o}{K_n(V_1 V_o - \frac{1}{2}V_o^2) - I_p}$$

$$\text{So } \frac{K_n \tau_{fall}}{2C} = \frac{V_{DD} - V_1}{V_1^2 - 2I_p/K_n} + \int_{V_{oL}}^{V_1} \frac{dV_o}{2V_1 V_o - V_o^2 - 2I_p/K_n}$$

The integral can be evaluated using partial fractions.

We define $V_2^2 \equiv 2I_p/K_n$. The denominator of the integral term can then be factorized by adding and subtracting V_1^2 .

$$2V_1 V_o - V_o^2 - V_1^2 + V_1^2 - V_2^2 = -(V_1 - V_o)^2 + \left(\sqrt{V_1^2 - V_2^2}\right)^2$$

$$= \left(\sqrt{V_1^2 - V_2^2} + V_1 - V_o\right) \left(\sqrt{V_1^2 - V_2^2} - V_1 + V_o\right)$$

Fall Time

$$\frac{K_n \tau_{fall}}{2C} = \frac{V_{DD} - V_1}{V_1^2 - 2I_p/K_n} + \int_{V_{oL}}^{V_1} \frac{dV_o}{2V_1 V_o - V_o^2 - V_2^2}$$

The denominator of the integral term can be factorized as

$$\left(\sqrt{V_1^2 - V_2^2} + V_1 - V_o \right) \left(\sqrt{V_1^2 - V_2^2} - V_1 + V_o \right)$$

The integral term can then be written as:

$$\frac{1}{2\sqrt{V_1^2 - V_2^2}} \left(\int_{V_{oL}}^{V_1} \frac{dV_o}{V_1 - V_o + \sqrt{V_1^2 - V_2^2}} + \int_{V_{oL}}^{V_1} \frac{dV_o}{V_o - V_1 + \sqrt{V_1^2 - V_2^2}} \right)$$

Fall Time

Integrating over V_o gives

$$\frac{1}{2\sqrt{V_1^2 - V_2^2}} \ln \left. \frac{V_o - V_1 + \sqrt{V_1^2 - V_2^2}}{V_1 - V_o + \sqrt{V_1^2 - V_2^2}} \right|_{V_{oL}}^{V_1}$$

On putting the limits for the integral, the upper limit gives 0. So the integral can be written as

$$\frac{1}{2\sqrt{V_1^2 - V_2^2}} \ln \frac{V_1 - V_{oL} + \sqrt{V_1^2 - V_2^2}}{V_{oL} - V_1 + \sqrt{V_1^2 - V_2^2}}$$

Fall Time

Thus we can write down the expression for fall time as:

$$\frac{K_n \tau_{fall}}{2C} = \frac{V_{DD} - V_1}{V_1^2 - 2I_p/K_n} + \frac{1}{2\sqrt{V_1^2 - V_2^2}} \ln \frac{V_1 - V_{oL} + \sqrt{V_1^2 - V_2^2}}{V_{oL} - V_1 + \sqrt{V_1^2 - V_2^2}}$$

Substituting back for V_2^2 , we get

$$\frac{K_n \tau_{fall}}{2C} = \frac{V_{DD} - V_1}{V_1^2 - 2I_p/K_n} + \frac{1}{2\sqrt{V_1^2 - 2I_p/K_n}} \ln \frac{V_1 - V_{oL} + \sqrt{V_1^2 - 2I_p/K_n}}{V_{oL} - V_1 + \sqrt{V_1^2 - 2I_p/K_n}}$$

Since $I_p = K_p(V_{DD} - V_{Tp})^2/2$, $2I_p/K_n$ is just $(V_{DD} - V_{Tp})^2/\beta$.

Fall Time

$$\frac{K_n \tau_{fall}}{2C} = \frac{V_{DD} - V_1}{V_1^2 - 2I_p/K_n} + \frac{1}{2\sqrt{V_1^2 - 2I_p/K_n}} \ln \frac{V_1 - V_{oL} + \sqrt{V_1^2 - 2I_p/K_n}}{V_{oL} - V_1 + \sqrt{V_1^2 - 2I_p/K_n}}$$

$$2I_p/K_n = \frac{V_{DD} - V_{Tp}}{\beta} \quad \text{and} \quad V_1 = V_i - V_{Tn}$$

This relation was derived using the pessimistic assumption that the p channel transistor dumps its saturation current over the entire discharge range.

In any case, this relation is not used for designing the inverter because the limit on the size of the n channel transistor is put by static considerations and not by the fall time.

Pseudo nMOS Inverter design

- We design the basic inverter and then scale device sizes based on the logic function being designed.
- The load device size is calculated from the rise time.

$$\tau_{rise} = \frac{C}{K_p(V_{DD} - V_{Tp})} \left[\frac{2V_{Tp}}{V_{DD} - V_{Tp}} + \ln \frac{V_{DD} + V_{oH} - 2V_{Tp}}{V_{DD} - V_{oH}} \right]$$

- Since $K_p = \mu_p C_{ox} W_p / L_p$, Width of the p channel transistor is given by

$$\frac{L_p C}{\mu_p C_{ox} \tau_{rise} (V_{DD} - V_{Tp})} \left[\frac{2V_{Tp}}{V_{DD} - V_{Tp}} + \ln \frac{V_{DD} + V_{oH} - 2V_{Tp}}{V_{DD} - V_{oH}} \right]$$

- Given a value of τ_{rise} , operating voltages and technological constants, K_p and hence, the geometry of the p channel transistor can be determined.

Pseudo nMOS Inverter design

- Geometry of the n channel transistor is determined from static considerations.

$$V_{oL} = (V_{iH} - V_{Tn}) - \sqrt{(V_{iH} - V_{Tn})^2 - (V_{DD} - V_{Tp})^2 / \beta}$$

- Let V_n be the desired static low noise margin. We take $V_{oL} = V_{Tn} - V_n$, and calculate β . (This is conservative. We could have taken V_{oL} to be $(V_{DD} - V_{Tp})/\sqrt{3\beta} - V_n$.
- The only unknown in the above relation is β . So $\beta \equiv K_n/K_p$ can be evaluated using it.
- Since K_p has already been evaluated, knowing β , we can evaluate K_n and hence, the geometry of the n channel transistor.

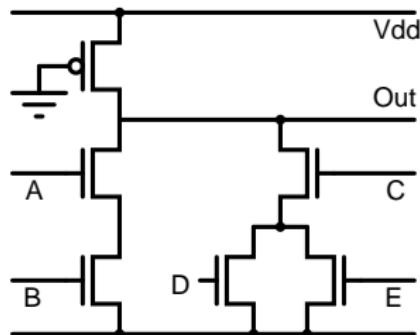
Conversion to other logic

- Once the basic pseudo nMOS inverter is designed, other logic gates can be derived from it.
- The procedure is the same as that for CMOS, except that it is applied only to nMOS transistors.
- The p channel transistor is kept at the same size as that for an inverter.
- Series-parallel rules are applied to determine the geometry of n channel transistors.

Conversion to other logic

- The logic is expressed as a sum of products with a bar (inversion) on top.
- For every ‘.’ in the expression, we put the corresponding n channel transistors in series.
- For every ‘+’, we put the n channel transistors in parallel. We scale the transistor widths up by the number of devices put in series.
- The geometries are left untouched for devices put in parallel.

$A \cdot B + C \cdot (D + E)$ in pseudo-nMOS



- A and B are in series.
- The pair is in parallel with C which is in series with a parallel combination of D and E.

Implementation of $\overline{A} \cdot \overline{B} + C \cdot (\overline{D} + \overline{E})$ in pseudo-nMOS logic design style.

Complementary Pass gate Logic

- This logic family is based on multiplexer logic.
- Given a Boolean function $F(x_1, x_2, \dots, x_n)$, we can express it as:

$$F(x_1, x_2, \dots, x_n) = x_i \cdot f_1 + \overline{x_i} \cdot f_2$$

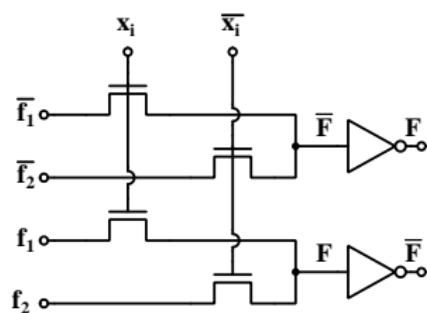
where f_1 and f_2 are reduced expressions for F with x_i forced to 1 and 0 respectively.

- Thus, F can be implemented with a multiplexer controlled by x_i which selects f_1 or f_2 depending on x_i .
- f_1 and f_2 can themselves be decomposed into simpler expressions by the same technique.

Complementary Pass gate Logic

- To implement a multiplexer, we need both x_i and $\overline{x_i}$.
- Therefore, this logic family needs all inputs in true as well as in complement form.
- In order to drive other gates of the same type, it must produce the outputs also in true and complement forms.
- Thus each signal is carried by two wires.
- This logic style is called “Complementary Pass-gate Logic” or CPL for short.

Basic Multiplexer Structure



Pure pass-gate logic contains no ‘amplifying’ elements. Therefore, each logic stage degrades the logic level.

Hence, multiple logic stages cannot be cascaded.

We include conventional CMOS inverters to restore the logic level.

Ideally, the multiplexer should be composed of complementary pass gate transistors.

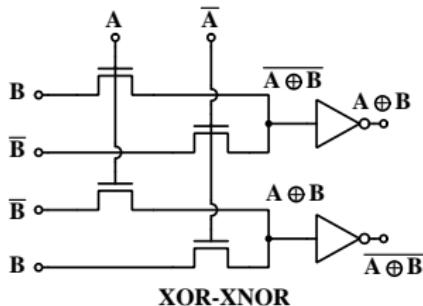
However, we shall use just n channel transistors as switches for simplicity.

Logic Design using CPL

- For any logic function, we pick one input as the control variable.
- Multiplexer inputs are decided by re-evaluating the function, forcing this variable to 1 and zero respectively.
- Since both true and complement outputs are generated by CPL, we need fewer types of gates.
- For example, we do not need separate gates for AND and NAND functions.
- The same applies to OR-NOR, and XOR-XNOR functions.

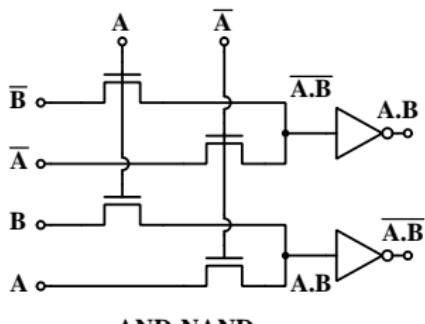
Implementation of XOR and XNOR

To take an example, let us consider the XOR-XNOR functions.

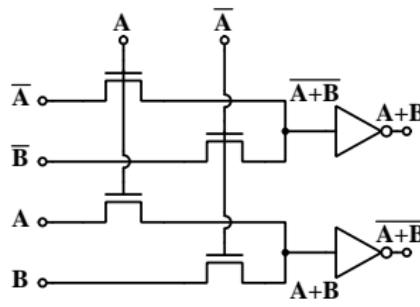


- Because of the inverter, for XOR output, We calculate the XNOR function given by $A \cdot B + \bar{A} \cdot \bar{B}$.
- If we put $A = 1$, this reduces to B and for $A = 0$, it reduces to \bar{B} .
- For the XNOR output, we generate the XOR expression $= A \cdot \bar{B} + \bar{A} \cdot B$
- The expression reduces to \bar{B} for $A = 1$ and to B for $A = 0$.

Implementation of AND-NAND and OR-NOR



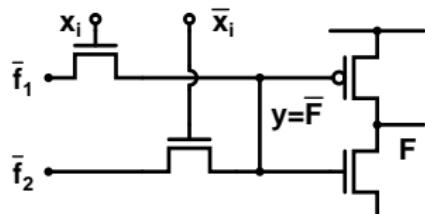
AND-NAND



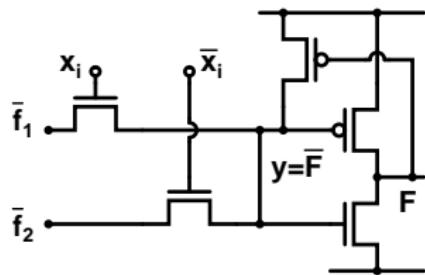
OR-NOR

- For AND, the mux should output $\bar{A} \cdot \bar{B}$ to be inverted by the buffer. This reduces to \bar{B} when $A = 1$ and to 1 ($= \bar{A}$) when $A = 0$.
- Implementation of NAND, OR and NOR functions follows along the same lines.

Buffer Leakage Current

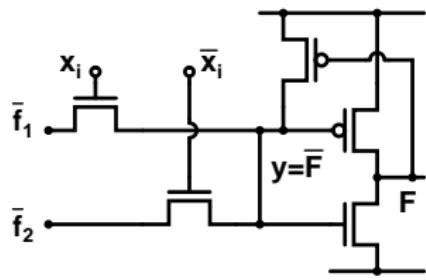


- The high output of the multiplexer (y) cannot rise above $V_{DD} - V_{Tn}$ because we use nMOS multiplexers.
- Consequently, the pMOS transistor in the buffer inverter never quite turns off.
- This results in static power consumption in the inverter.



This can be avoided by adding a pull up pMOS with the inverter.

Use of Pull-up PMOS



- When the multiplexer output (y) is 'low', the inverter output (F) is high. The pMOS is off and has no effect.
- When the multiplexer output (y) goes 'high', the inverter output falls and turns the pMOS on.

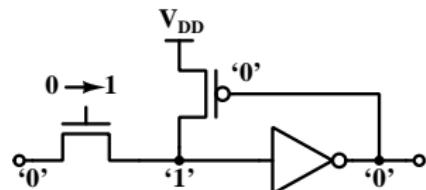
Now, even though the multiplexer nMOS turns 'off' as y approaches $V_{DD} - V_{Tn}$, the pMOS remains 'on' and takes the inverter input (y) all the way to V_{DD} .

This avoids leakage in the inverter.

Need for ratioing

The use of pMOS pull-up brings up another problem.

Consider the equivalent circuit when the inverter output is ‘low’ and the pMOS is ‘on’.

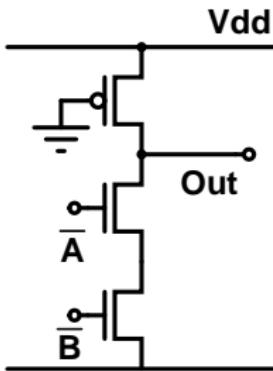
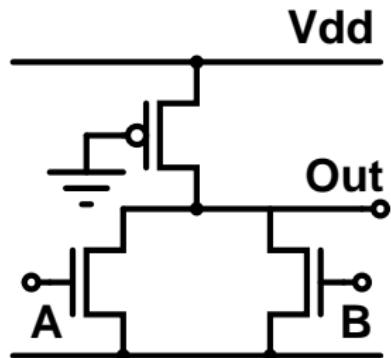


If the final output is ‘low’, the pMOS pull-up is ‘on’. Now if the multiplexer output wants to go ‘low’, it has to fight the pMOS pull-up - which is trying to keep this node ‘high’.

In fact, the multiplexer n transistor and the pull up p transistor constitute a pseudo nMOS inverter.

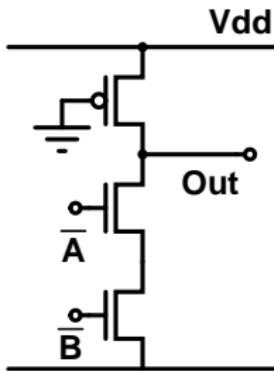
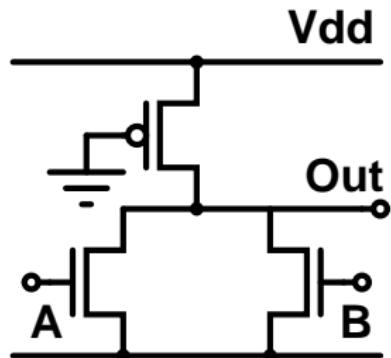
Therefore, the multiplexer output cannot be pulled low unless the transistor geometries are appropriately ratioed.

Improving Pseudo nMOS



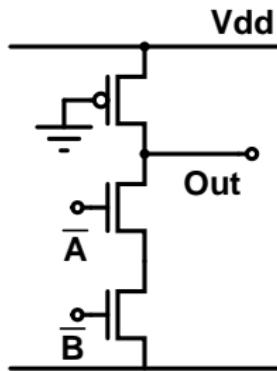
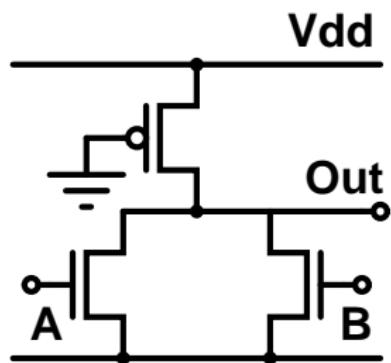
- In the pseudo-nMOS NOR circuit on the left, static power is consumed when the output is 'LOW'
- We would like to turn the pMOS off when A OR B is TRUE.
- The OR logic can be constructed by using a Pseudo-nMOS NAND of \bar{A} and \bar{B} as in the circuit on the right.
- But then what about the pMOS drive of this circuit?

Improving Pseudo nMOS



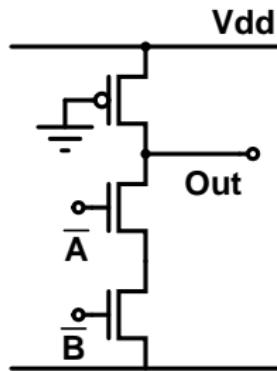
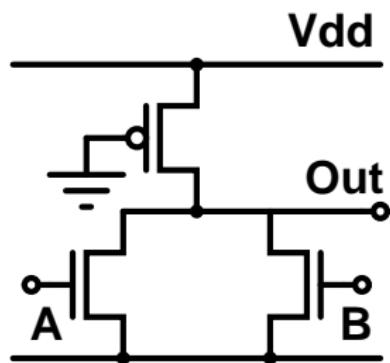
- In the pseudo-nMOS NOR circuit on the left, static power is consumed when the output is 'LOW'
- We would like to turn the pMOS off when A OR B is TRUE.
- The OR logic can be constructed by using a Pseudo-nMOS NAND of \bar{A} and \bar{B} as in the circuit on the right.
- But then what about the pMOS drive of this circuit?

Pseudo nMOS without Static Power



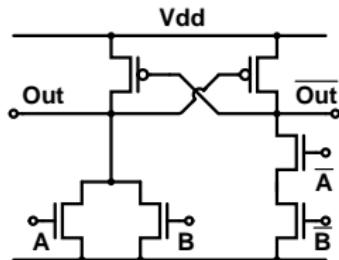
- The output of the circuit on the right is 'LOW' when both \bar{A} and \bar{B} are 'HIGH' ($A = B = 0$).
- We would like to turn *its* pMOS off when NOR of A and B is 'TRUE'
- But this can be provided by the circuit on the left!
- So the two circuits can drive each other's pMOS transistors and avoid static power consumption.

Pseudo nMOS without Static Power



- The output of the circuit on the right is ‘LOW’ when both \bar{A} and \bar{B} are ‘HIGH’ ($A = B = 0$).
- We would like to turn *its* pMOS off when NOR of A and B is ‘TRUE’
- But this can be provided by the circuit on the left!
- So the two circuits can drive each other’s pMOS transistors and avoid static power consumption.

Cascade Voltage Switch Logic



This kind of logic is called Cascade Voltage Switch Logic (CVSL).

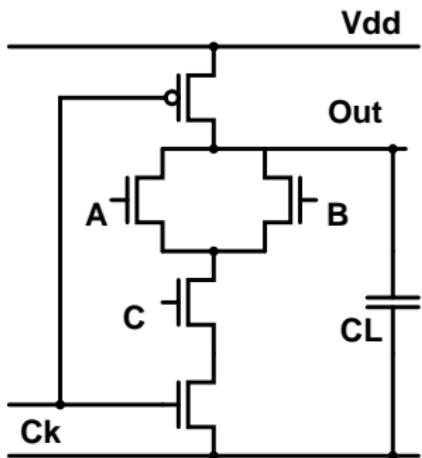
It can use any network f and its complementary network \bar{f} in the two cross-coupled branches.

- Like CMOS static logic, there is no static power consumption.
- Like CPL, this logic requires both True and Complement signals. It also provides both True and complement outputs. (Dual Rail Logic).
- Like pseudo nMOS, the inputs present a single transistor load to the driving stage.
- The circuit is self latching. This reduces ratioing requirements.

Dynamic logic

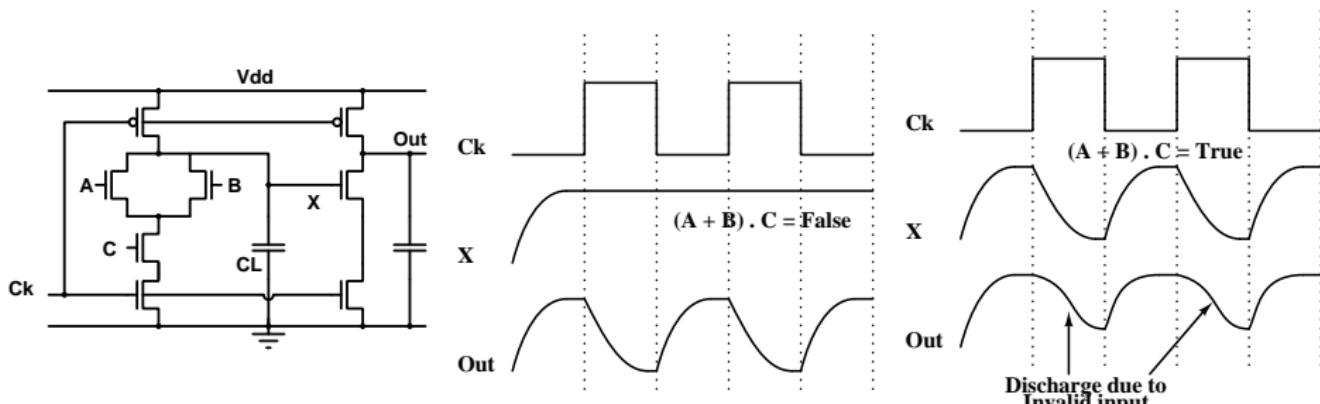
- In this style of logic, some nodes are required to hold their logic value as a charge stored on a capacitor.
- These nodes are not connected to their ‘drivers’ permanently.
- The ‘driver’ places the logic value on them, and is then disconnected from the node.
- Due to leakage etc., the logic value cannot be held indefinitely.
- Dynamic circuits therefore require a *minimum* clock frequency to operate correctly.
- Use of dynamic circuits can reduce circuit complexity and power consumption substantially.

A CMOS dynamic logic circuit



- When the clock is low, pMOS is on and the bottom nMOS is off.
- The output is ‘pre-charged’ to 1 unconditionally.
- When the clock goes high, the pMOS turns off and the bottom nMOS comes on.
- The circuit then conditionally discharges the output node, if $(A+B).C$ is TRUE.
- This implements the function $\overline{(A + B)}.C$.

Problem with Cascading



There is no problem when $(A+B) \cdot C$ is false. X pre-charges to 1 and remains at 1.

When $(A+B) \cdot C$ is TRUE, X takes some time to discharge. During this time, charge placed on the output leaks away as the input to nMOS of the inverter is not 0.

Problem with Cascading

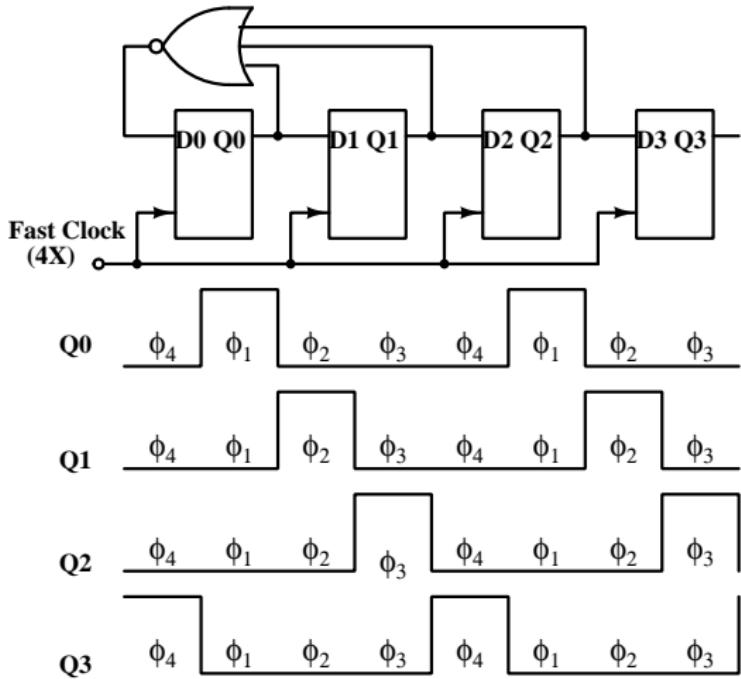
- In the dynamic circuit, the output is pre-charged to ‘1’.
- If the final output is supposed to be ‘0’, there will be some time during which the output will still be at the wrong value of ‘1’.
- This transiently wrong value can discharge the pre-charged output of the next stage and lead to malfunction.
- We need to isolate the output from the next stage till it has acquired the correct value.
- Operation of dynamic logic should proceed in distinct time slots or ‘phases’, with the output disconnected from the next stage in the phase where it is still in the process of evaluating.
- To implement this, we need a multi-phase clock.

Multi-phase clock

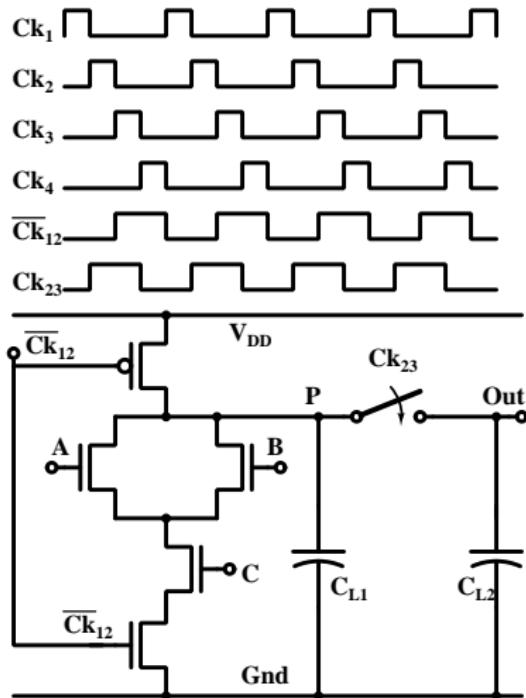
- In a multi-phase implementation, the output is pre-charged during the first phase.
- In the next phase, pre-charging is disabled and logic evaluation is enabled. Output is disconnected from the next stage during pre-charge as well as evaluation phases.
- In the final phase(s), the output holds its correct value. It is connected to the next stage and disconnected from pre-charge and evaluate circuits of the current stage.
- A minimum of 3 phases are required – pre-charge, evaluate and valid.
- In a 4-phase implementation, the valid state holds for a duration of two phases.
- To implement this kind of dynamic logic, we need a multi-phase clock.

Multi-phase clock generation

- We need to generate various clock signals to control connection and disconnection of sub-circuits during different time slots.
- An example scheme for generating a 4 phase clock is shown on the right.
- As can be seen, each Q output is high during a single phase of a 4 phase cycle.

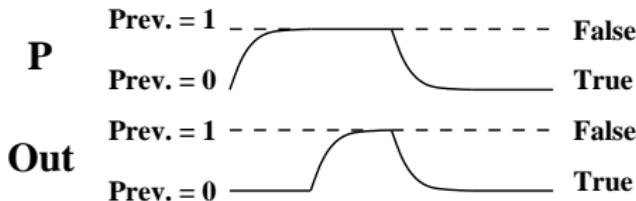


4 Phase Dynamic Logic

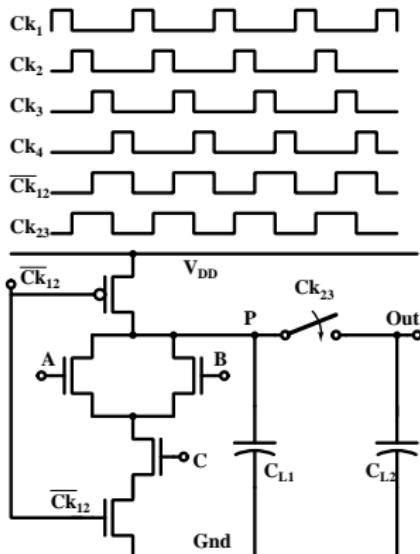


- Ck_{mn} is defined as a clock signal which is 'high' during phase m and phase n of the clock.
- Similarly, \overline{Ck}_{mn} is a clock signal which is 'low' during phase m and phase n of the clock.

Phase → 1 2 3 4

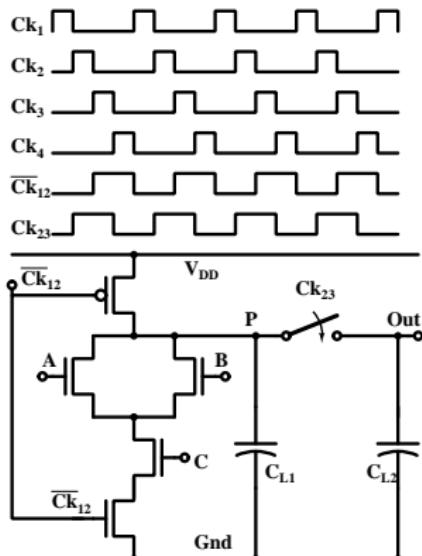


4 Phase Dynamic Logic



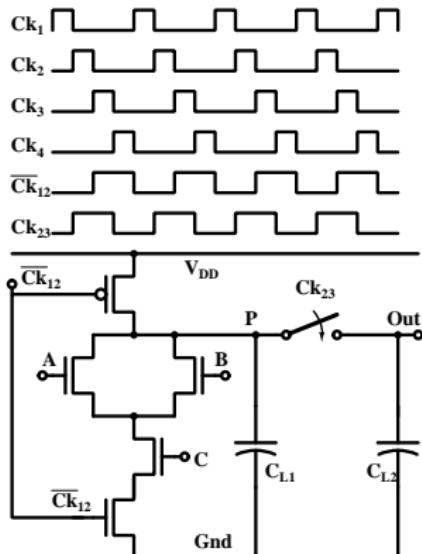
- The circuit on the left shows a 4 phase dynamic gate.
- In phase 1, $\overline{Ck_{12}} = 0$, $Ck_{23} = 0$. So the pMOS is on and the bottom (clocked) nMOS is off. The output is disconnected from transistors.
- Node P pre-charges to '1' while the output holds its old value.
- In phase 2, $\overline{Ck_{12}} = 0$, $Ck_{23} = 1$.
- pMOS is on, bottom (clocked) nMOS is off and the output capacitor is in parallel with the capacitor on node P.
- As a result, node P, as well as the output node pre-charge to 1.

4 Phase Dynamic Logic



- In phase 3, $\overline{Ck}_{12} = 1$, $Ck_{23} = 1$.
- So the pre-charge pMOS is off and the bottom (clocked) nMOS is on. Capacitors at node P and at the output are still in parallel.
- If $(A + B) \cdot C = 1$, both capacitors will discharge to ground through the signal transistors and the bottom (clocked) nMOS. Otherwise the output will remain at 1.
- Thus the gate evaluates in phase 3 and acquires the correct output value.
- In phase 4, $\overline{Ck}_{12} = 1$, $Ck_{23} = 0$. The output is isolated from transistors and will hold its valid value.

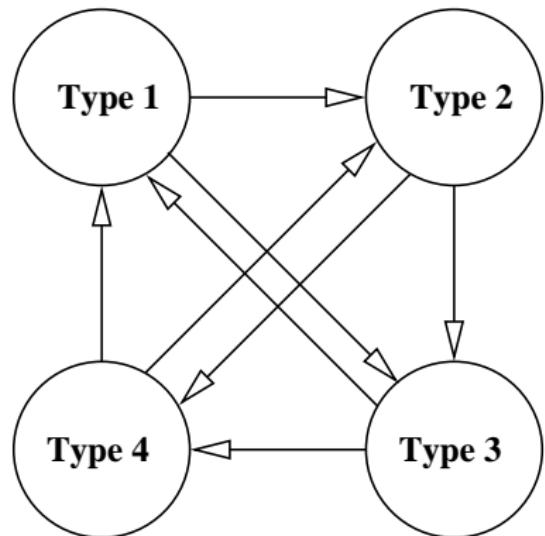
4 Phase Dynamic Logic



- In phases 4 as well as in phase 1, the output is isolated from the driver and retains its valid value.
- Notice that the node P is no more valid in phase 1, since it pre-charges to '1' in phase 1.
- This is called a type 3 gate, and needs the inputs to be valid and stable in phase 3.
- By changing the clocks to \bar{Ck}_{23} and Ck_{34} , we shall get a type 4 gate which evaluates in phase 4 and whose output remains valid in phases 1 and 2.
- Similarly, by cyclic permutation of clock signals, we can get type 1 and type 2 gates.

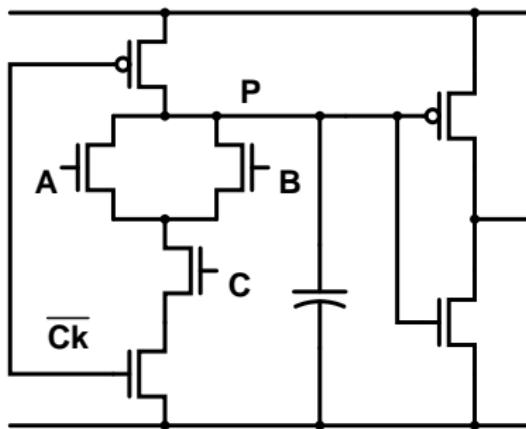
Drive cycles

Drive Sequences



- A type 3 gate can drive a type 4 or a type 1 gate.
- similarly, type 4 will drive types 1 and 2; type 1 will drive types 2 and 3; and type 2 will drive types 3 and 4.
- We can use a 2 phase clock if we stick to type 1 and type 3 gates (or type 2 and type 4 gates) as these can drive each other.

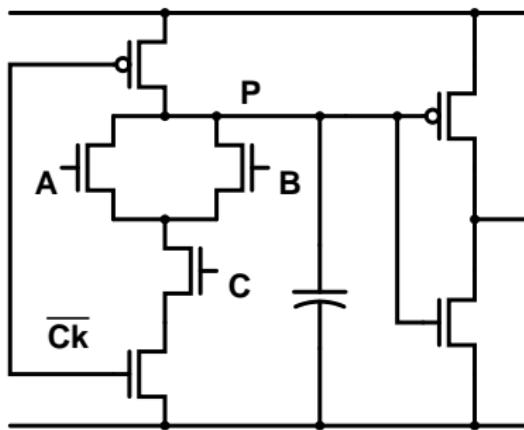
Domino Logic



- Another way to eliminate the problem with cascading logic stages is to use a static inverter after the CMOS dynamic gate.
- Now the output is '0' when it is not valid.
- Therefore, it does not affect the evaluation of the next gate.

This kind of logic is called a domino logic stage.

Domino Logic



- The output is held 'low' before logic evaluation.
- If the final output of this gate is '0', there is no problem anyway.
- If the final output was supposed be '1', the next stage will be erroneously held at zero for some time.

However, this does not result in a false evaluation by the next stage.

The only effect it can have is that the next stage starts its evaluation a little later.

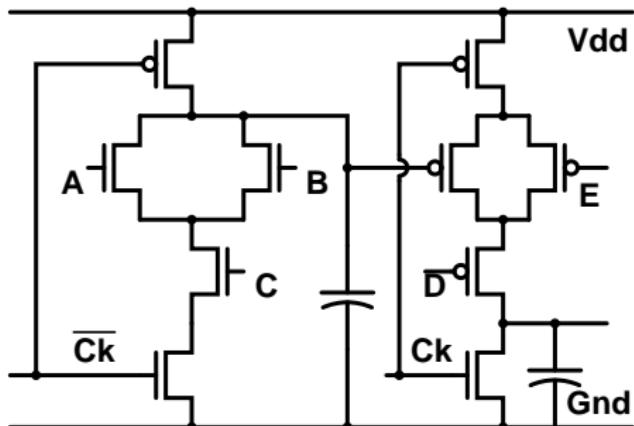
Domino Logic

- Domino logic is fast, because the pre-charge cycle is common to all stages.
- Once pre-charge is done, each stage evaluates one after the other (hence the name - domino logic).
- Thus for n stage logic, there is only one cycle of pre-charge and n cycles of evaluation, rather than n cycles of pre-charge and n of evaluation.

Domino Logic: Non inverting output?

- The addition of an inverter means that the logic is non-inverting.
- Therefore, it cannot be used to implement any arbitrary logic function.
- In synchronous digital circuits, combinational logic alternates with clocked latches.
- If inversion is required, we insert a latch at that place and take the \overline{Q} output of the latch to the next group of domino logic.

Zipper Logic



A, B, C must be from p stages.
D and E must be from n stages.

The circuit above will implement
 $((A + B) \cdot C + \overline{E}) \cdot \overline{D}$

This kind of logic is called *zipper logic*.

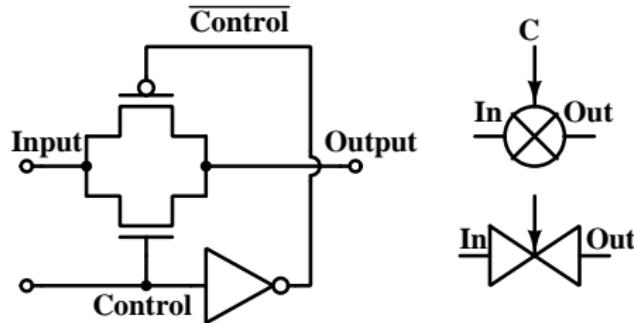
- Instead of using an inverter, we can alternate n and p evaluation stages.
- The n stage is pre-charged high, but it drives a p stage.
- A high pre-charged stage will keep the p evaluation stage off, which will not cause any malfunction.
- The p stage will be pre-discharged to ‘low’, which is safe for driving n stages.

Circuits using Mixed Styles

- Some commonly used circuits use a mixture of styles.
- The availability of transistor switches which can pass signals in either direction provides a unique flexibility to CMOS technology based designs.
- Mixing traditional CMOS logic gates with MOS transistor switches in this way leads to efficient implementation of many useful circuits.
- Examples of this class of circuits are transmission gates, tri-stateable inverters, multiplexers, tiny-xor circuits and D latches and flipflops.

Transmission gate

A transmission gate using two pass transistors and a CMOS inverter is shown below:



Both transistors are on or off simultaneously because of the inverter. The pMOS passes a '1' value efficiently, while the nMOS passes a '0' value efficiently.

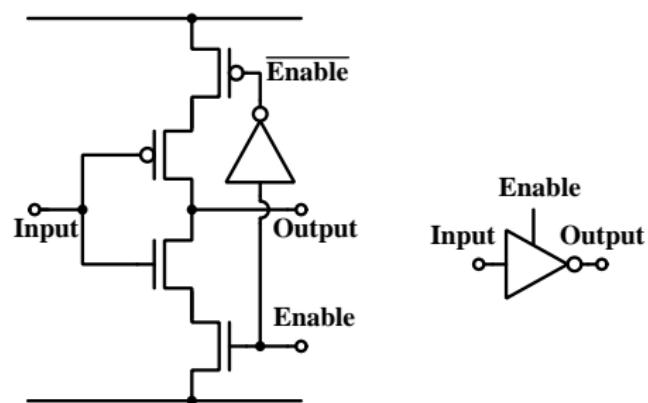
Notice that the input and output are interchangeable.

This circuit is represented by either of the symbols shown on the right.

Tri-stateable Inverter

We sometimes need an inverter which can be connected or disconnected from the output using an enable signal.

- When $\text{Enable} = 0$, both pull up and pull down are off. The inverter then presents a high impedance output.
- When $\text{Enable} = 1$, power is supplied to the inner pMOS and nMOS transistors and the circuit acts like a CMOS inverter.



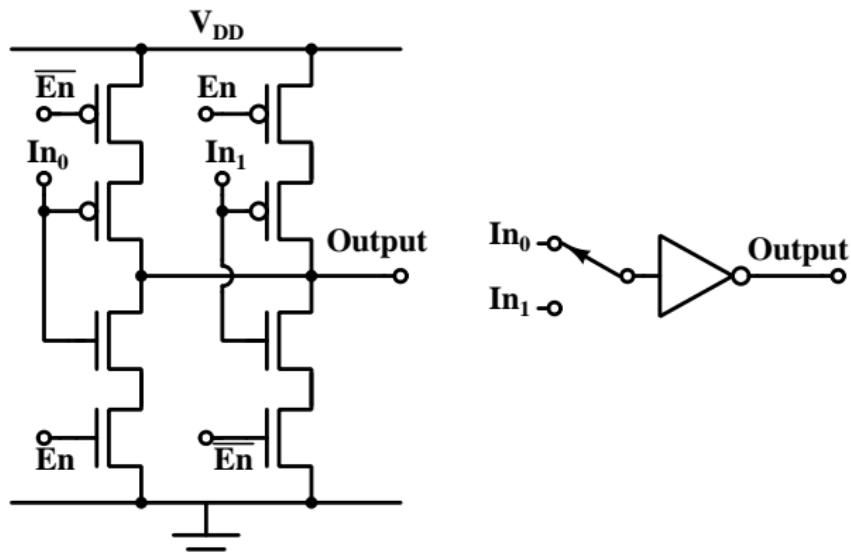
The symbol shown on the right is used to represent a tri-stateable inverter.

2 way multiplexer

By shorting the outputs of two tristateable inverters, we can implement an inverting mux.

The enable signals of the two tri-stateable inverters are complementary.

So only one of these is active at any given time.

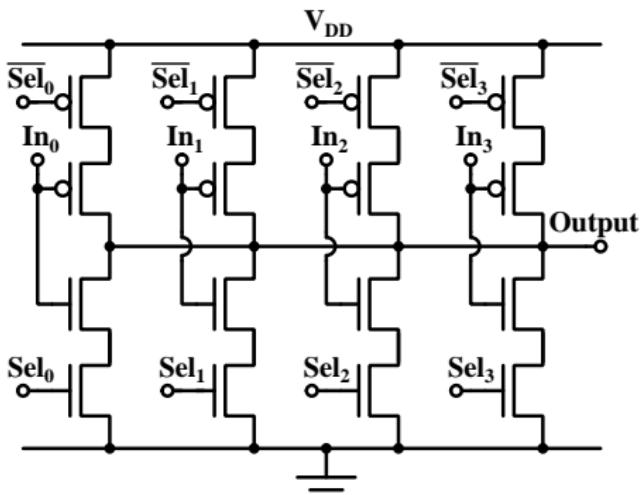


Depending on the state of the enable signal, the input to the enabled inverter appears at the output in inverted form.

Multiple input Multiplexer

We can generalize the design of the 2-way mux to a multiple input multiplexer by adding a decoder.

- The decoder generates Sel and \bar{Sel} for each input, which in turn control a tri-stateable inverter each.
- Only one Sel signal is high and the corresponding \bar{Sel} signal is low.

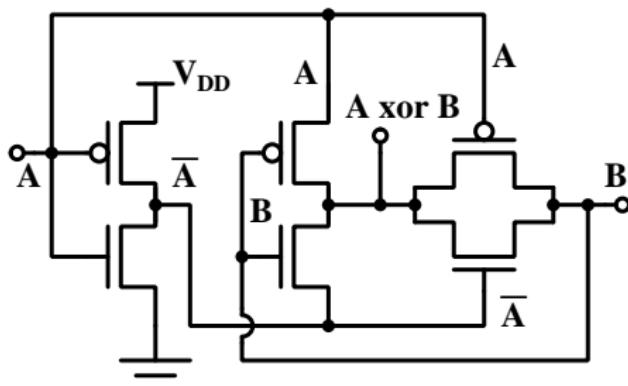


Outputs of all the tri-stateable inverter are shorted together. Input data of the selected inverter appears inverted at the output and all other inverters are disabled.

“tiny” XOR

A compact XOR circuit can be made using pass gates.

- When $A = 0$, the pass gate on the right is on and passes B to the output.
- Also, the inverter like circuit in the middle has 0 applied on top and a 1 applied at the bottom. Both transistors act as source followers and thus, also drive the output to B .



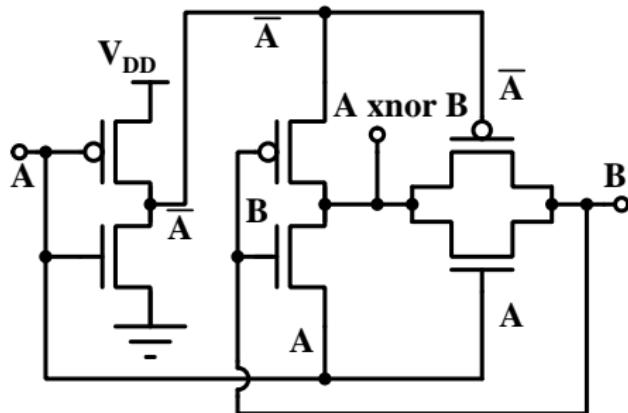
When $A = 1$, the pass gate is open and the middle circuit has 1 at the top and 0 at the bottom. So it acts like a regular inverter and provides \bar{B} at the output.

By Shannon's Boolean expansion theorem, this constitutes the XOR function: $A \cdot \bar{B} + \bar{A} \cdot B$.

“tiny” XNOR

A compact XNOR circuit can also be made similarly.

- When $A = 1$, the pass gate on the right is on and passes B to the output.
- Also, the inverter like circuit in the middle has 0 applied on top and a 1 applied at the bottom. Both transistors act as source followers and also couple B to the output.



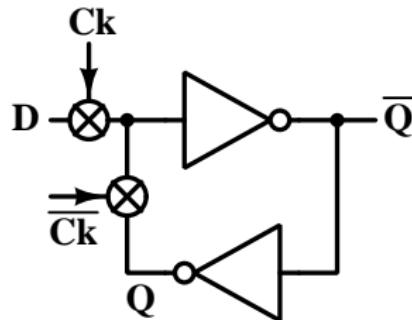
When $A = 0$, the pass gate is open and the middle inverter like circuit has 1 at the top and 0 at the bottom. So it acts like a regular inverter and provides \bar{B} at the output.

By Shannon's Boolean expansion theorem, this constitutes the XNOR function: $A \cdot B + \bar{A} \cdot \bar{B}$

Transparent D Latch

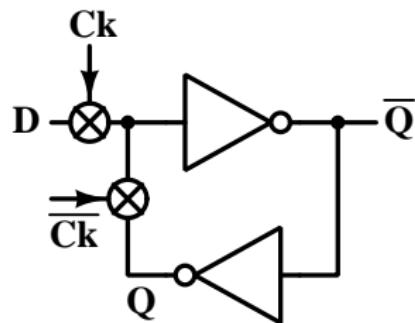
Two inverters and two pass gates can be combined to form a transparent (level sensitive) D latch.

- When clock is high, the input is connected to the two inverters and the feedback switch is open.
- So D is buffered through to Q.
- When clock is low, the input is disconnected, the feedback switch is on and forms a latch.
- Q captures the value of D when the clock goes from high to low.
- When the clock is high, changes in the value of D will result in a corresponding change in the Q outputs. That is why it is called a transparent latch.

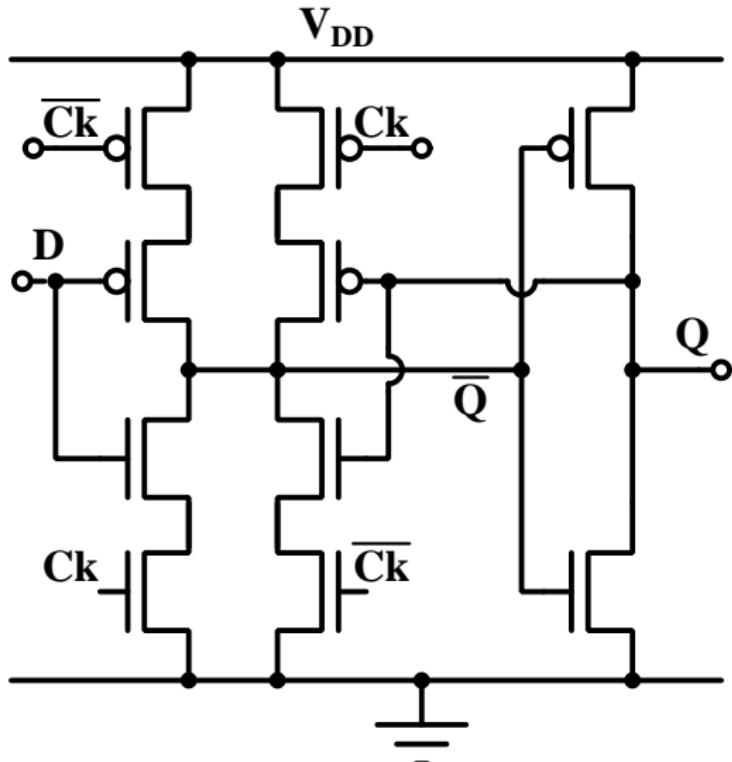


Transparent D Latch using Mux

The two pass gates and the upper inverter in the transparent D latch form a two way multiplexer between D and Q.



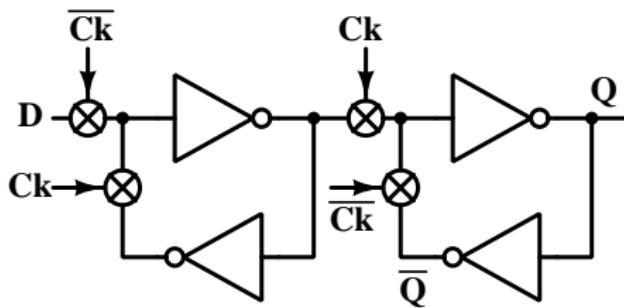
We can replace these by the 2 way mux described earlier.



D Flip Flop

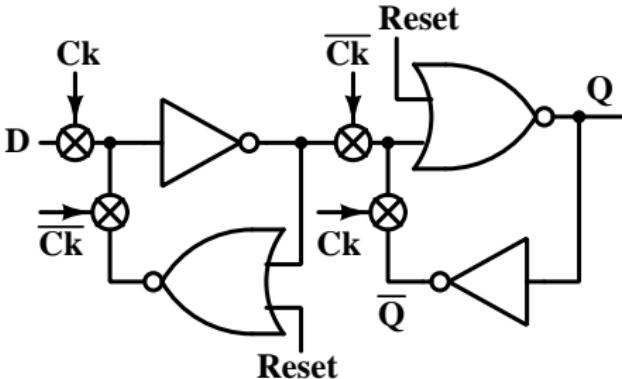
Two transparent D latches with complementary clocks can be connected in a master-slave configuration to form an edge sensitive D flipflop

- When the clock is low, the first latch is transparent but the second latch hold its previous value.
- When the clock transitions to high, the first latch captures the value of D at this instant. The second latch becomes transparent.
- Outputs don't follow D in either state of the clock.
- Q reflects the value of D at the most recent positive transition on Ck.



D Flip Flop with Reset

We sometimes need an edge sensitive D flip flop which can be reset asynchronously.

- Q should go to 0 as soon as reset is applied.
 - To do this, the inverter supplying Q is changed to a NOR, with Reset as its other input.
 - However, this is not sufficient. The flip flop should remain reset even when the reset signal is removed and Ck is at 0.
 - For this, the lower inverter in the first latch should also be changed to a NOR gate, with reset as its other input.
- 

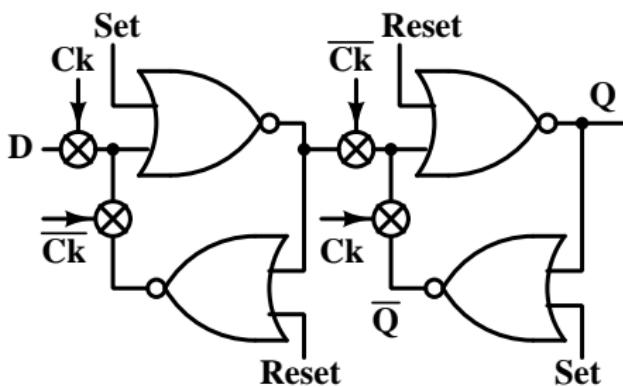
Use of asynchronous reset is not recommended due to testability concerns.

D Flip Flop with Set and Reset

If we need both set and reset for an edge sensitive D flipflop, we should change the other two inverters also to NOR.

- Q should go to 1 as soon as set is applied.
- When $Ck = 1$, a 1 on set (with 0 on reset) will force Q to 1.
- When $Ck = 0$, a 1 on set will force the first latch output to 0. This will be inverted by the NOR in the transparent second latch to produce a 1.
- In common with all asynchronous set and reset circuits, both should not be asserted simultaneously.

Use of asynchronous set or reset is not recommended due to testability concerns.



Semi-custom Design

- In the previous sections, we have looked at design techniques used when we have to design all parts of a VLSI circuit from scratch.
- This includes the choice of logic styles, adjustment of transistor geometries, their layout etc. to meet a given set of specifications.
- This is called full custom design.
- While this permits an optimal trade-off between power, speed and complexity, the whole process is long and laborious.
- The cost of developing a custom designed VLSI circuit is very high and can be justified only by those designs which sell in very large quantities.

Semi-custom Design

- Some applications can afford to compromise on speed, power and complexity in order to achieve a quick design and fabrication time and lower costs.
- These use a design style known as semi-custom design.
- In semi-custom design, part of the design and fabrication is already done for us. We take this pre-fabricated template and customize it to perform the functions that our VLSI needs to perform.
- This approach has several advantages. The prefabricated part can be processed and kept ready for customization.
- Then the time to take an application to the market is defined only by the remaining processing which is necessary after customization.

Semi-custom Design

- Different applications can use the same pre-fabricated template.
- While each application may have a relatively small market, the template will be made in very large numbers, making it economically competitive.
- Obviously, the pre-fabricated template itself is not customized for a particular final circuit. So it will not be an optimal design for a given application.
- However, most applications do not require absolutely the best performance.
- In such cases, the time to market and cost can be drastically reduced by using semi-custom design.

Field Programmable Gate Arrays

- From a time to market point of view, it is preferable to have the customization done as late as possible.
- However, then a higher fraction of the design and fabrication cycle is non-specific to the actual design.
- In Field Programmable Gate Array (FPGA) based design, customization is done *after* the product has actually been delivered to the end user.
- That is what the term Field Programmable refers to.
- The actual VLSI template which can support this late customization is quite complex.
- The area of the chip is much higher and the speed much lower compared to what could have been achieved by a custom designed circuit.

Field Programmable Gate Arrays

- System clock speeds possible with FPGAs are of the order of hundreds of MHz, whereas custom circuits can go to clock speeds of about 4 GHz.
- To implement a given function, the silicon area consumed by an FPGA may be an order of magnitude higher than the area consumed by custom designed circuit performing the same function.
- However, since many applications can use the same FPGA, the FPGA chip is produced in very large quantities, which provides economies of scale.
- Thus, for designs which are not produced in very large quantities, FPGA based design can be the most cost effective solution, in spite of the overhead in complexity, power consumption and delay.

Semi-Custom Design Procedure

There are two components of Semi-custom design technique.

- ① Customization techniques which will be used for adapting the “fabric” to implement the final application on the given template.
- ② The design of the basic template or the “fabric” which can be customized as late as possible and which can be used by a large variety of applications.

During fabrication, transistors are defined much before the interconnects.

Therefore, to be able to customize the circuit as late as possible, customization of a pre-fabricated template is commonly done by changing the interconnects.

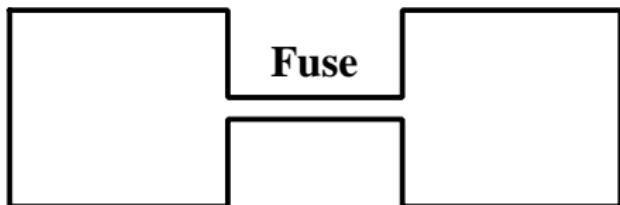
Customizing Interconnects

Customization of interconnects is done by placing programmable interconnect devices at appropriate points in the pre-fabricated template. These include:

- Fuses: Here the connection is a short by default and can be converted to an open by blowing a fuse.
- Anti-fuses: Here the connection is open by default, but can be converted to a short by breaking down an insulator.
- transistor based interconnects: Here a connection is made or broken using transistor switches. The state of the transistor is defined by a memory.

Use of Fuses to Customize Interconnect

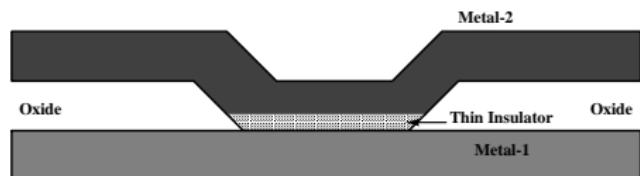
Customization of interconnects can be done through programmable removal of an existing connection. These work like fuses in electrical circuits.



- The connection to be customized is connected in the template through a narrow neck like structure.
- The narrow part of the interconnect is capable of passing normal operating currents of the circuit.
- However, during customization, one can pass a pulse of heavy current through this, which 'blows' the fuse and disconnects the wires leading up to the fuse.

Use of Anti-Fuses to Customize Interconnect

In these structures, there is no connection between the programming nodes in the un-programmed state of the interconnect. The current path is interrupted through a thin layer of insulator.



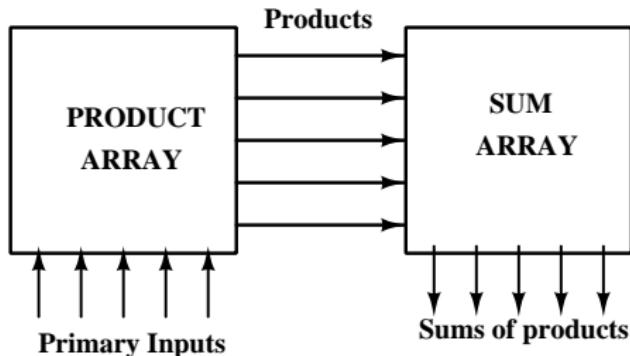
By applying a high voltage pulse, this insulator can be broken down and a short created across it.

This action is opposite to that of a fuse. Therefore such structures are called anti-fuses.

Re-configurable Logic with Programmable Logic Arrays

Logic functions can be expressed in a generic sum of products form.

- We need a circuit which can be re-configured to implement any logic expressed as a sum of products.
- One way is to use a customisable array which produces different product terms and another customisable array which sums the products so produced.



Use of Pseudo-nMOS for Programmable Logic Arrays

- We need an architecture where transistor geometries are not changed with logic and only interconnect needs to be programmed.
- CMOS logic is not convenient for implementing this architecture.
- This is because simultaneous configuration of the p channel pull up network and the n channel pull down network will be needed.
- Pseudo NMOS gates are more suitable in this case, because the pull up is just a single grounded gate pMOS whose geometry remains the same for all logic.

Use of Pseudo-nMOS for Programmable Logic Arrays

- Pseudo NMOS circuits are ratioed.
- Implementing sums is not a problem, since this is done with NOR type gates.
- Transistor geometry remains the same in NOR gates irrespective of how many transistors are put in parallel.
- However, implementing products presents a problem, since NAND gates connect nMOS transistors in series.
- The geometry of series connected pull down devices depends on the number of devices connected in series and hence, on the specific logic being implemented.

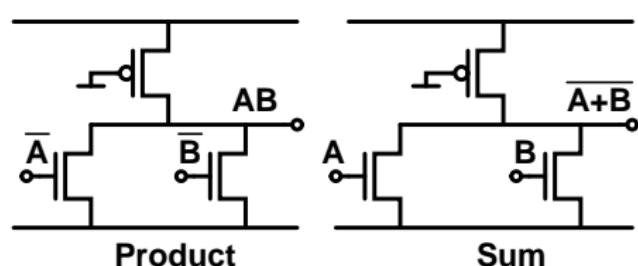
Use of Pseudo-nMOS for Programmable Logic Arrays

How to implement the product function without changing transistor geometry?

We can use the expression :

$$A \cdot B = \overline{\overline{A} + \overline{B}}$$

Now the product of A and B can be implemented as the NOR of \overline{A} and \overline{B} , which does not use series connected transistors.



By adding inverters at the input and output as required, we can implement products or sums using only NOR type of circuits, which use constant geometry NMOS transistors in parallel.

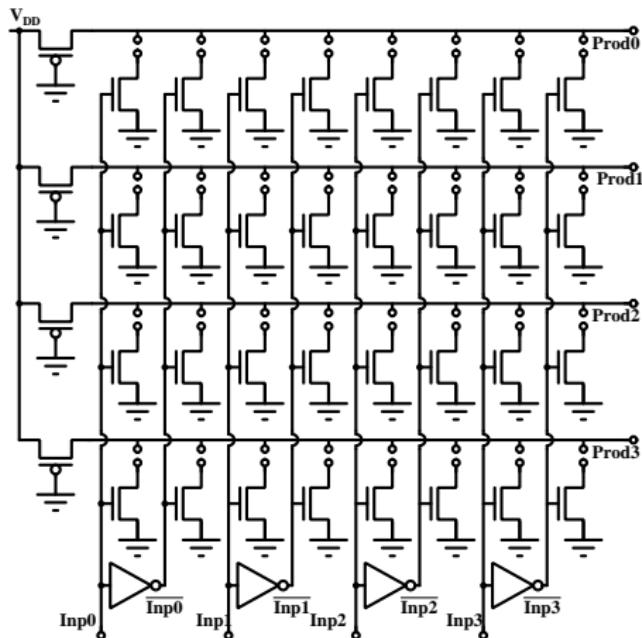
Programmable Logic: Product Array

Suppose we want to generate p distinct products of n inputs.

- We generate complements of all inputs.
- We place pull down nMOS transistors in a matrix of $2n$ columns and p rows.
- Each row is pulled up by a pMOS transistor with grounded gate.
- The row corresponds to the output of a NOR circuit which may have up to n pull down nMOS transistors in parallel.
- Each column is driven by an input or its complement. So there are $2n$ columns.

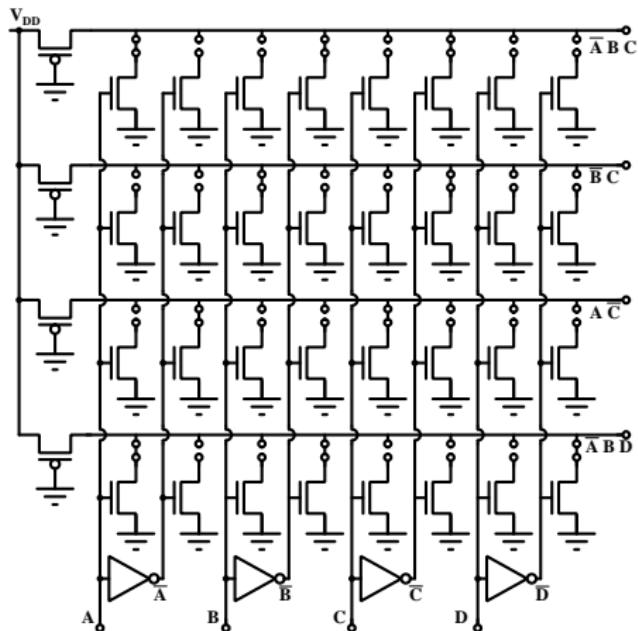
Programmable Logic: Product Array

- Each row generates a distinct product.
- By connecting links selectively at drains of nMOS transistors in any row, chosen transistors can be included in the NOR configuration.
- Connecting a transistor includes the complement of the input driving its gate in the product term.



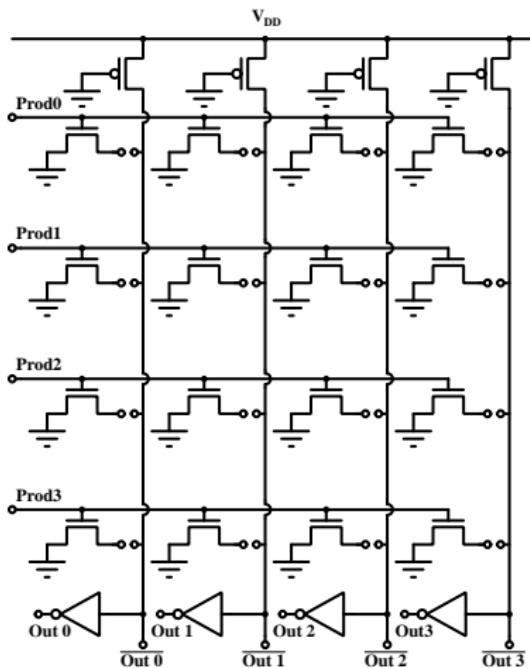
Programmable Logic: Product Array

- In the example shown on the right, we have generated products $\bar{A} \cdot B \cdot C$, $\bar{B} \cdot C$, $A\bar{C}$ and $\bar{A} \cdot B \cdot D$.
- To generate $\bar{A} \cdot B \cdot C$, links from drains of transistors with gates connected to the columns with A , \bar{B} and \bar{C} are connected to the top product row.
- Other products are generated similarly.

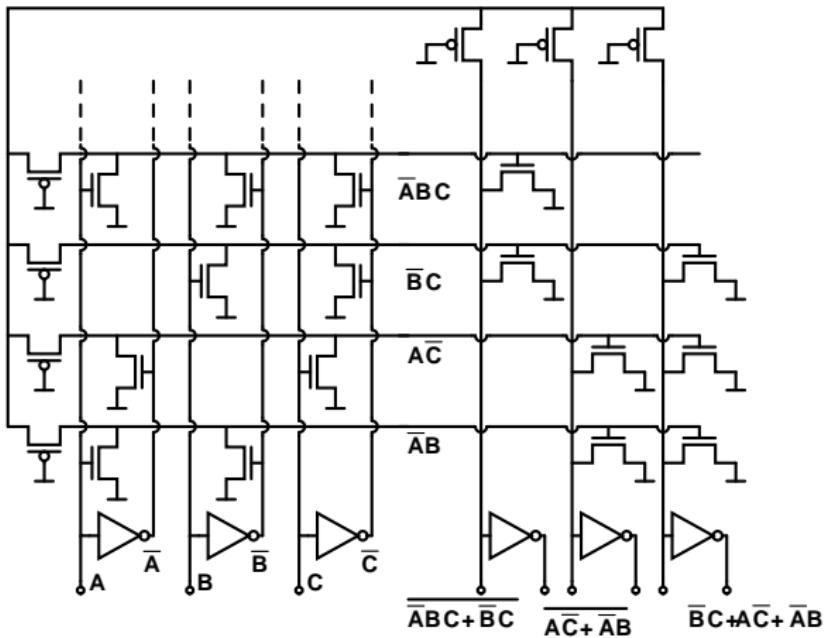


Programmable Logic: Sum Array

- The circuit on the right can produce sums of selected product terms.
- By connecting links at drains of nMOS transistors in a particular column, chosen products can be included in a sum output.
- Several columns can be used to generate different sums of products.
- Outputs are NORs of products. Inverters convert these to ORs of products.



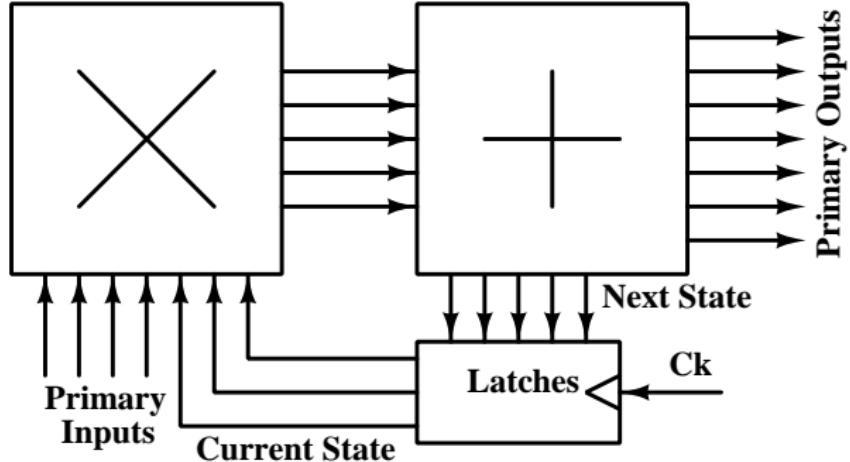
Programmable Logic Arrays



Implementation of Finite State Machines

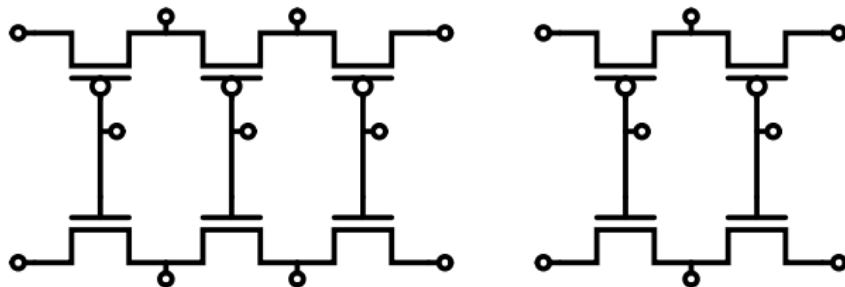
- A finite state machine has the following components:
 - 1 Storage elements to store the current state,
 - 2 Random logic to compute the next state as a function of current state and current inputs, and
 - 3 Random logic to compute the current output as a function of current state and optionally, the current inputs.
- By adding latches at the output of the PLA, we can provide for all of these.
- Output from latches is fed back to the product array.
- The PLA computes both the next state and current outputs.

Implementation of Finite State Machines



Sea of Gates

- This style uses CMOS logic as its base.
- In CMOS logic, each input goes to an nMOS as well as a pMOS.
- Transistors are pre-placed in a matrix of cells like the one shown below. Interconnects determine what kind of logic will be implemented.

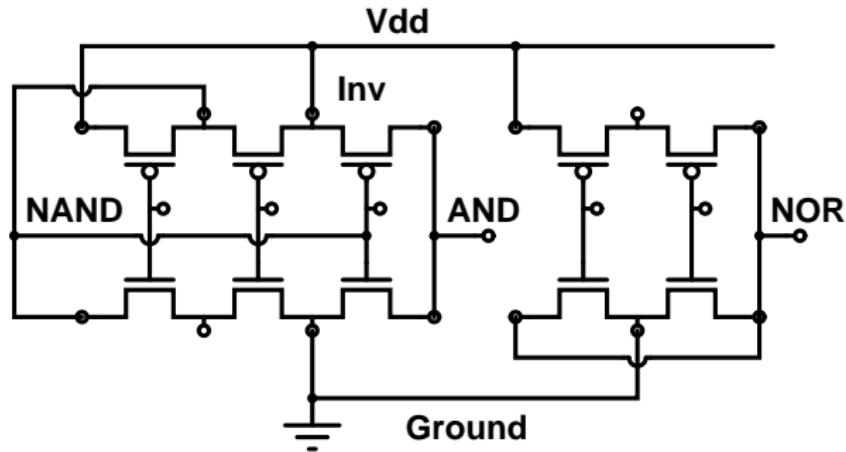


Both n and p channel transistors are in series here!

How do we construct regular CMOS logic gates using these?

Logic from Sea of Gates

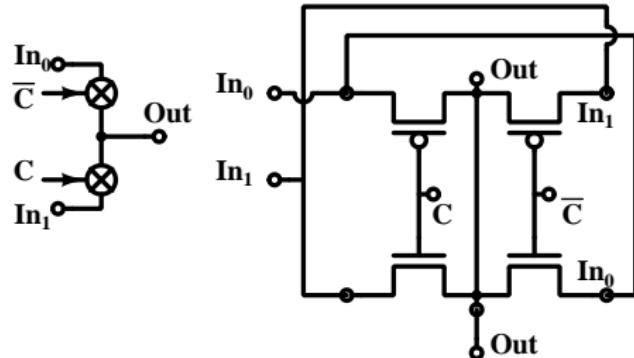
Actually, by wiring the apparently series connected devices, we can convert them to series or parallel as required.



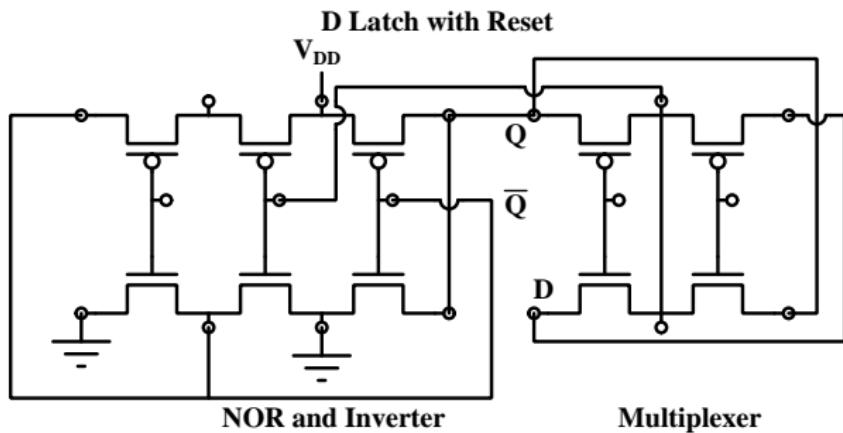
The example above shows a NAND gate, whose output is fed to an inverter to form the AND function. The structure on the right forms a NOR gate.

Sea of Gates Transmission Gate

- The sea of gates template makes use of the fact that all inputs go to an nMOS as well as to a pMOS in CMOS style gates.
- What about structures which don't follow this rule? For example, in a transmission gate, the nMOS and pMOS transistors are driven by complementary signals.
- Transmission gates are often used in pairs with one or the other being on. (Inputs should not be left floating in static CMOS design).
- The pair can be implemented as shown on the right, coupling diagonally opposite transistors in a 2-pair.



Sea of Gates implementation of D latch

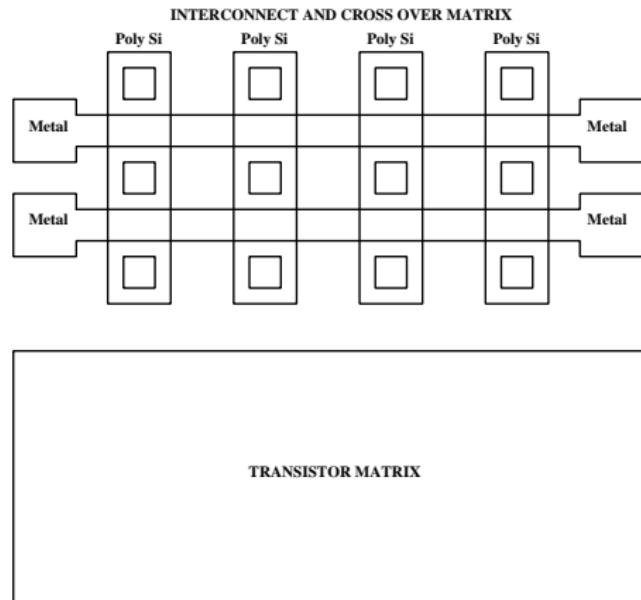


Two such latches can be connected in master-slave configuration to form an edge sensitive D flip-flop.

Interconnect Channels in Semi-Custom Logic

Pre-fabricated interconnect channels alternate with transistor matrices in the fabric of a semi-custom chip.

- The exact shape and composition of these interconnect channels varies from design to design.
- Typically, these will include facilities for local as well as global interconnects.
- These are optimized to provide a sufficient interconnect capacity without wasting too much area.



Using Memories as Logic

- A logic function can be represented by a truth table. This can be stored in memory. Inputs to the logic function act as address pins.
- Pre-computed value of the logic function are stored at the address represented by its input values.
- For any input combination, the address represented by these is looked up and the stored value presented as the output.
- For example, any logic function of 5 inputs can be implemented by a 32×1 bit memory.
- If the same 32 bit memory is organized as 16×2 , it can store two logic functions of up to 4 inputs.

Evolution of FPGAs

Different semi-custom design chips have been introduced with various names. Here is a list:

PLAs: These were the first semi-custom devices to be introduced in the market by Philips in the early 1970's. These are used to implement generic sums of products and have a programmable AND plane as well as a programmable OR plane. Because both AND and OR arrays are programmable in PLAs, these are rather slow.

PALs: A modification of PLAs in which the product array is programmable, but the OR plane is fixed were introduced as Programmable Array Logic or PAL. To cover for the lack of flexibility due to the OR array being fixed, these were introduced in different size combinations and multiple devices were used for different functions.

Evolution of FPGAs

CPLDs As technology progressed, it was possible to put multiple devices on the same chip. Combinations of PALs and PLAs were introduced with programmable interconnect as complex programmable logic devices or CPLDs. Altera was one of the first companies to introduce CPLDs commercially. CPLDs were a huge success and multiple companies introduced CPLDs of different architectures.

Sea of Gates In parallel with field programmable devices, mask programmable devices were manufactured. In these, one (or more) levels of metallization could be customized at the manufacturing site, over a “sea” of pre-fabricated devices. These provided circuits with better performance, but with less flexibility as programming could not be done at the user site.

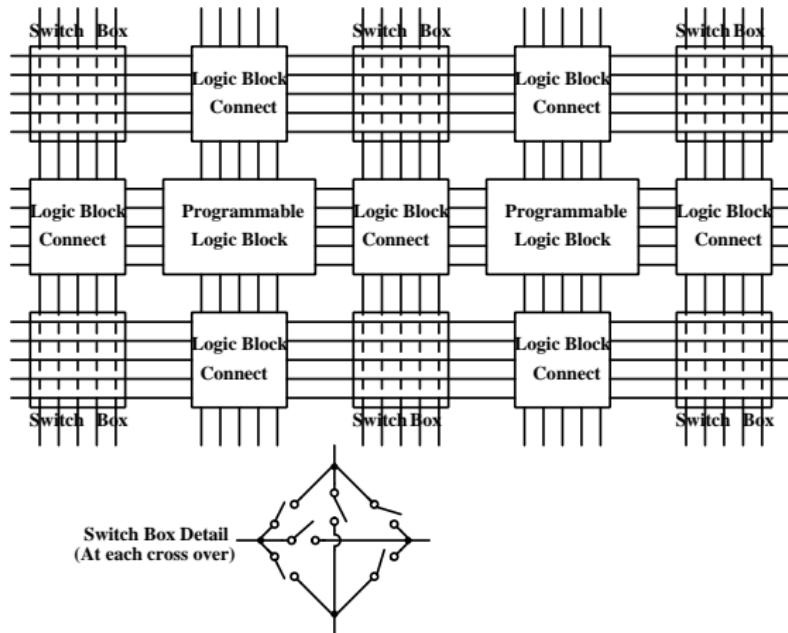
Evolution of FPGAs

FPGAs Field Programmable Gate Arrays borrow features from sea of gates as well as CPLDs. Instead of a “sea” of transistors, these have a “sea” or a repetitive array of combinations of logic elements and memories. In addition, these include a programming infrastructure for interconnects like CPLDs.

On the periphery of these chips, special Input Output devices are fabricated which help in establishing fast interconnection with the external world.

FPGA Architectures

A field Programmable array allows the logic functions as well as the interconnect to be programmed.



Transistors driven by SRAM can be used to program interconnects.

Current FPGA Architecture

Apart from logic blocks and interconnect fabric, modern FPGA's contain other useful structures, such as

- Block RAM,
- Processor cores (Power PC on Xilinx Vertex family),
- Fast multipliers
- I-O Blocks

Types of Reconfigurable Logic

- A typical board, used for implementing a variety of applications will have a micro-controller and a few programmable devices for providing dedicated functions and glue logic.
- This is typically configured once at power on, and then left alone to perform its functions. This is called “static re-configurability”.
- But we may want to re-configure a structure while it is in operation! For example, one can imagine the design of digital filter, which adapts itself during operation itself depending on inputs. This kind of re-configurability is called “dynamic re-configurability”.
- Obviously, dynamic configurability requires that the dead time during re-configuration should be minimized.

Fine and Coarse Grain Reconfigurability

- There is a trade-off between flexibility of re-configuration and the time taken to re-configure.
- In fine grain re-configuration, we can re-program every gate of the design. This is the case for current FPGA and CPLD devices. This gives very good design flexibility, but takes a long time to re-configure.
- In coarse grain re-configuration, relatively large functional blocks are re-configured. For example, by re-connecting a collection of shifters and adders, we can generate any combination of multipliers and adders. The effort to re-configure is smaller, because we do not alter the inner design of shifters and adders. This is compatible with dynamic re-configuration.

Use of Dynamic Re-configuration

Using coarse grain re-configuration, it becomes possible to dynamically change a circuit, *during* its operation.

This has obvious advantages in adaptive circuits.

We normally do not want to re-configure the whole circuit. Indeed the control signals for doing the re-configuration will be generated by a circuit which remains unchanged.

Therefore dynamic re-configuration requires circuits which can be partially re-programmed. Many FPGA are beginning to promise this feature.

Design of Logic Gates in CMOS Technology

Dinesh Sharma
EE Department, IIT Bombay

September 20, 2020

Contents

1 Transistor Models	5
2 Static CMOS Logic Design	8
2.1 Static CMOS Design style	8
2.2 CMOS Inverter	8
2.2.1 Static Characteristics	9
2.2.2 Noise margins	14
2.2.3 Dynamic Considerations	16
2.2.4 Trade off between power, speed and robustness	20
2.2.5 CMOS Inverter Design Flow	20
2.2.6 Conversion of CMOS Inverters to other logic	20
3 Pseudo-nMOS Logic Design	22
3.1 Static Characteristics	23
3.1.1 For $0 \leq V_i \leq V_{Tn}$: nMOS ‘off’	24
3.1.2 nMOS saturated, pMOS linear	24
3.1.3 nMOS linear, pMOS linear	25
3.1.4 nMOS linear, pMOS saturated	25
3.2 Noise margins	26
3.3 Dynamic characteristics	27
3.3.1 Rise Time	27
3.3.2 Fall Time	27
3.4 Pseudo nMOS design Flow	29
3.5 Conversion of pseudo nMOS Inverter to other logic	30
4 Dual Rail Logic Design	31
4.1 Complementary Pass gate Logic	31
4.1.1 Basic Multiplexer Structure	31
4.1.2 Logic Design using CPL	32
4.1.3 Buffer Leakage Current	33
4.2 Cascade Voltage Switch Logic	35

5	Dynamic Logic	37
5.1	Basic CMOS Dynamic Gate	37
5.1.1	Problem with Cascading CMOS dynamic logic	38
5.2	Four Phase Dynamic Logic	40
5.3	Domino Logic	43
5.4	Zipper logic	44
6	Circuits with Mixed Design Styles	46
6.1	Transmission gates	46
6.2	Tri-stateable inverter	47
6.3	Multiplexers	47
6.4	Tiny XOR and XNOR	48
6.5	D latch and flipflop	50
7	Semi-Custom Design	53
7.1	Procedure for Semi-Custom Design	54
7.2	Customization of interconnects	54
7.3	Implementation of Configurable Template	55
7.3.1	Programmable Logic Arrays	55
7.3.2	Sea of Gates	60
7.4	Interconnect Channels in Semi-Custom Logic	62
7.5	Using Memories as Logic	63
7.6	Field Prgrammable Gate Arrays	63

List of Figures

1.1	MOS characteristics according to the simple analytic model	5
1.2	MOS characteristics with non zero conductance in saturation	6
2.1	The basic CMOS inverter	8
2.2	nMOS and pMOS drain currents for different input voltages	9
2.3	Output voltage for nMOS in saturation, pMOS in linear regime	10
2.4	Output voltage when both transistors are saturated	12
2.5	Transfer Curve of a CMOS inverter	12
2.6	Output voltage when nMOS is in linear regime while pMOS is saturated.	13
2.7	CMOS inverter with the nMOS ‘off’	17
2.8	CMOS inverter with the pMOS ‘off’	19
2.9	CMOS implementation of $\overline{A.B + C.(D + E)}$	21
3.1	Pseudo nMOS inverter	22
3.2	Transistor currents in a Pseudo nMOS inverter	23
3.3	‘high’ to ‘low’ transition on the output	27
3.4	Pseudo NMOS implementation of $\overline{A.B + C.(D + E)}$	30
4.1	Basic Multiplexer with logic restoring inverters	32
4.2	Implementation of XOR and XNOR by CPL logic.	32
4.3	Implementation of (a) AND-NAND and (b) OR-NOR functions using complementary pass-gate logic.	33
4.4	High leakage current in inverter	33
4.5	Pull up pMOS to avoid leakage in the inverter	34
4.6	Problem with a low to high transition on the output	34
4.7	Pseudo-nMOS NOR	35
4.8	Pseudo-nMOS OR from complemented inputs	35
4.9	OR-NOR implementation in Cascade Voltage Switch Logic	36
5.1	CMOS dynamic gate to implement $\overline{(A + B).C}$	37
5.2	Dynamic gate for $\overline{(A + B).C}$ cascaded with an inverter.	38
5.3	Generation of phase clocks for dynamic logic	40

5.4	CMOS 4 phase dynamic logic	41
5.5	Operation of a type 3 gate	42
5.6	CMOS 4 phase dynamic logic drive constraints	43
5.7	CMOS domino logic	43
5.8	Zipper logic	45
6.1	a transmission gate	46
6.2	a tri-stateable inverter	47
6.3	a two way mux	48
6.4	4 input multiplexer	48
6.5	“Tiny” XOR circuit using a transmission gate	49
6.6	“Tiny” XNOR circuit using a transmission gate	49
6.7	A transparent D latch using transmission gates	50
6.8	A transparent D latch using a two way multiplexer	51
6.9	Edge sensitive master slave D flipflop	51
6.10	Edge sensitive D flipflop with asynchronous reset	51
6.11	Edge sensitive D flipflop with asynchronous set and reset	52
7.1	fuse structure	55
7.2	An anti-fuse. The insulator could be amorphous silicon.	55
7.3	Architetcture of a Programmable Logic Array	56
7.4	Product and sum implementation in a PLA	56
7.5	A programmable product array	57
7.6	Example for generation of different products from inputs	58
7.7	Sum array for generating sums of products	59
7.8	A finite state machine implemented with a PLA	60
7.9	Pattern of transistors in a sea of gates template	61
7.10	Implementing a NAND, Inverter and NOR gates in sea of gates	61
7.11	A pair of transmission gates with complementary control inputs	62
7.12	A transparent D latch implemented with “Sea of Gates”	62
7.13	An example interconnect channel for semi-custom logic	63
7.14	Alternating logic and interconnect boxes in an FPGA	65

Chapter 1

Transistor Models

We shall use simple analytical models for MOS transistors. We use a sign convention according to which, voltage and current symbols associated with the pMOS transistor (such as V_{Tp}) have positive values. Then, the n channel formulae can be used for both transistors and we shall assign signs to quantities explicitly.

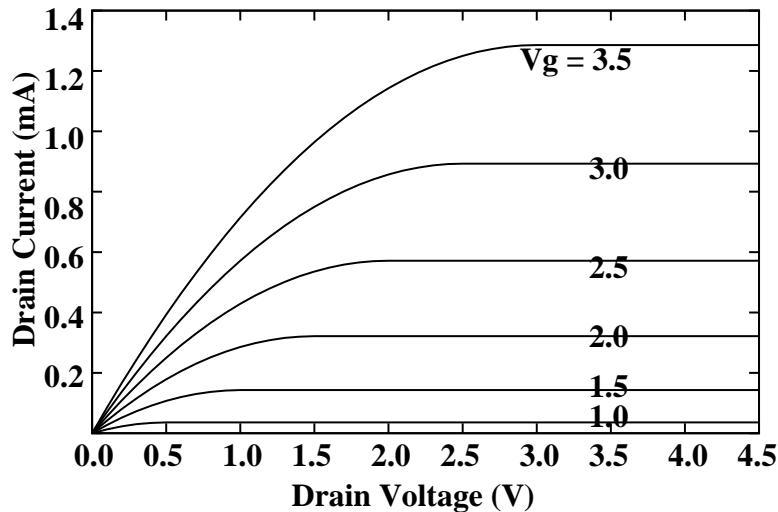


Figure 1.1: MOS characteristics according to the simple analytic model

The model we use is described by the following equations:

for $V_{gs} \leq V_T$,

$$I_{ds} = 0 \quad (1.1)$$

for $V_{gs} > V_T$ and $V_{ds} \leq V_{gs} - V_T$,

$$I_{ds} = K \left[(V_{gs} - V_T)V_{ds} - \frac{1}{2}V_{ds}^2 \right] \quad (1.2)$$

and for $V_{gs} > V_T$ and $V_{ds} > V_{gs} - V_T$,

$$I_{ds} = K \frac{(V_{gs} - V_T)^2}{2} \quad (1.3)$$

The saturation region equation is somewhat oversimplified because it assumes that the current is independent of V_{ds} . In reality, the current has a weak dependence on V_{ds} in this region.

In order to model the saturation region more accurately, we adopt an “Early Voltage” like formalism.

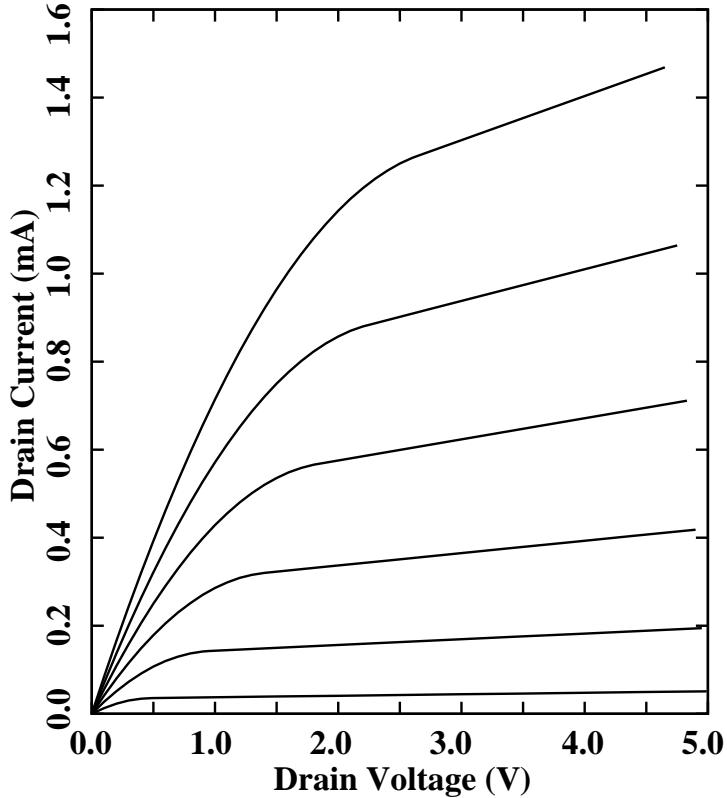


Figure 1.2: MOS characteristics with non zero conductance in saturation

It is assumed that the current increases linearly in the saturation region. All linear characteristics in saturation can be produced backwards towards negative drain voltages and will intersect the drain voltage axis at a single point at $-V_E$. (This is, at best, an approximation). Because the conductance in saturation is now non zero, the onset of saturation has to be redefined, so that the current and its derivative are continuous at the boundary of linear and

saturation regimes. The current equations are given by:

For $V_{gs} > V_T$ and $V_{ds} \leq V_{dss}$,

$$I_{ds} = K \left[(V_{gs} - V_T)V_{ds} - \frac{1}{2}V_{ds}^2 \right] \quad (1.4)$$

and for $V_{gs} > V_T$ and $V_{ds} > V_{dss}$,

$$I_{ds} = I_{dss} \frac{V_d + V_E}{V_{dss} + V_E} \quad (1.5)$$

Where V_E is the ‘Early Voltage’. Here V_{dss} and I_{dss} are saturation drain voltage and drain current respectively. Since the current values must match at either side of $V_{ds} = V_{dss}$, we must have:

$$I_{dss} \equiv K \left[(V_{gs} - V_T)V_{dss} - \frac{1}{2}V_{dss}^2 \right]. \quad (1.6)$$

For the curve to be smooth and continuous at $V_d = V_{dss}$, the value of the first derivative should match on either side of V_{dss} . Therefore,

$$K(V_{gs} - V_T - V_{dss}) = \frac{I_{dss}}{V_{dss} + V_E}$$

So,

$$K(V_{gs} - V_T - V_{dss})(V_{dss} + V_E) = K \left[(V_{gs} - V_T)V_{dss} - \frac{1}{2}V_{dss}^2 \right] \quad (1.7)$$

This leads to a quadratic equation in V_{dss}

$$\frac{1}{2}V_{dss}^2 + V_E V_{dss} - (V_{gs} - V_T)V_E = 0 \quad (1.8)$$

Solving this quadratic, we get

$$V_{dss} = V_E \left(\sqrt{1 + \frac{2(V_{gs} - V_T)}{V_E}} - 1 \right) \quad (1.9)$$

For $V_E \gg V_{gs} - V_T$ this reduces to

$$V_{dss} \simeq (V_{gs} - V_T) \left(1 - \frac{V_{gs} - V_T}{2V_E} \right) \quad (1.10)$$

Characteristics of a MOS transistor using this model are shown in fig.1.2. While accurate modeling of the output conductance is essential for linear design, the simpler model assuming constant I_d in saturation is often adequate for preliminary digital design. In any case, final designs will have to be validated with detailed simulations. In this booklet, we shall use the simple model for MOS devices to keep the algebra simple.

Chapter 2

Static CMOS Logic Design

Static logic circuits are those which can hold their output logic levels for indefinite periods as long as the inputs are unchanged. Circuits which depend on charge storage on capacitors are called dynamic circuits and will be discussed in a later chapter.

2.1 Static CMOS Design style

The most common design style in modern VLSI design is the Static CMOS logic style. In this, each logic stage contains pull up and pull down networks which are controlled by input signals. The pull up network contains p channel transistors, whereas the pull down network is made of n channel transistors. The networks are so designed that the pull up and pull down networks are never ‘on’ simultaneously. This ensures that there is no static power consumption.

2.2 CMOS Inverter

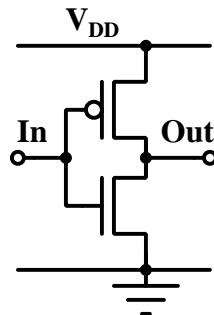


Figure 2.1: The basic CMOS inverter

The simplest of such logic structures is the CMOS inverter. In fact, for any CMOS logic design, the CMOS inverter is the basic gate which is first analyzed and designed in detail. Thumb rules are then used to convert this design to other more complex logic. The basic CMOS inverter is shown in fig. 2.1. We shall develop the characteristics of CMOS logic through the inverter structure, and later discuss ways of converting this basic structure more complex logic gates.

2.2.1 Static Characteristics

The range of input voltages can be divided into several regions.

- nMOS ‘off’, pMOS ‘on’
- nMOS saturated, pMOS linear
- nMOS saturated, pMOS saturated
- nMOS linear, pMOS saturated
- nMOS ‘on’, pMOS ‘off’

Let us compute the output voltage for a series of input voltages from 0V to V_{DD} . We have plotted the individual drain currents of the n and p channel transistors as functions of the output voltage for different input voltages V_{in} .

The gate voltage for the nMOS transistor = V_{in} and the drain voltage = V_{out} while the

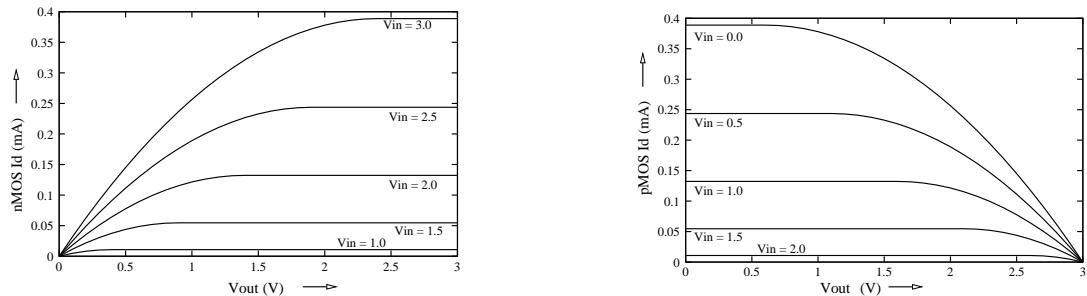


Figure 2.2: nMOS and pMOS drain currents for different input voltages

absolute gate voltage for pMOS is $V_{DD} - V_{in}$ and the absolute drain voltage is $V_{DD} - V_{out}$.

The Output voltage of the inverter will be the value where the two currents are equal for any given V_{in} .

nMOS ‘off’, pMOS ‘on’

For $0 < V_i < V_{Tn}$ the n channel transistor is ‘off’, the p channel transistor is ‘on’ and the output voltage = V_{DD} . This is the normal digital operation range with input = ‘0’ and output = ‘1’.

nMOS saturated, pMOS linear

In this regime, both transistors are ‘on’. The input voltage V_i is $> V_{Tn}$, but is small enough so that the n channel transistor is in saturation, and the p channel transistor is in the linear regime. In static condition, the output voltage will adjust itself such that the currents through the n and p channel transistors are equal.

The absolute value of gate-source voltage on the p channel transistor is $V_{DD} - V_i$, and therefore

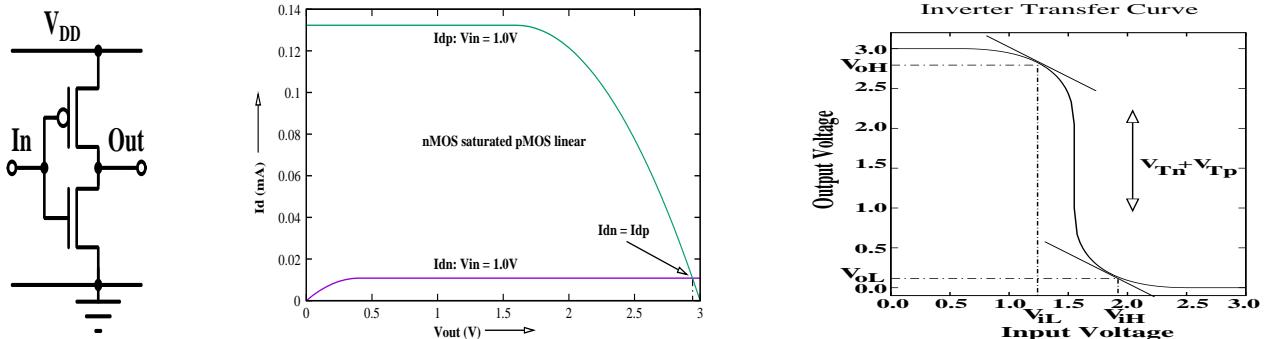


Figure 2.3: Output voltage for nMOS in saturation, pMOS in linear regime

the “over voltage” on its gate is $V_{DD} - V_i - V_{Tp}$. The drain source voltage of the pMOS has an absolute value $V_{DD} - V_o$.

Therefore,

$$I_d = K_p \left[(V_{DD} - V_i - V_{Tp})(V_{DD} - V_o) - \frac{1}{2}(V_{DD} - V_o)^2 \right] = \frac{K_n}{2}(V_i - V_{Tn})^2 \quad (2.1)$$

Where symbols have their usual meanings.

We define $\beta \equiv K_n/K_p$. We make the substitution $V_{dp} \equiv V_{DD} - V_o$, where V_{dp} is the absolute value of the drain-source voltage for the p channel transistor. Then,

$$(V_{DD} - V_i - V_{Tp})V_{dp} - \frac{1}{2}V_{dp}^2 = \frac{\beta}{2}(V_i - V_{Tn})^2 \quad (2.2)$$

Which gives the quadratic

$$\frac{1}{2}V_{dp}^2 - V_{dp}(V_{DD} - V_i - V_{Tp}) + \frac{\beta}{2}(V_i - V_{Tn})^2 = 0 \quad (2.3)$$

Solutions to the quadratic are:

$$V_{dp} = (V_{DD} - V_i - V_{Tp}) \pm \sqrt{(V_{DD} - V_i - V_{Tp})^2 - \beta(V_i - V_{Tn})^2} \quad (2.4)$$

These equations are valid only when the pMOS is in its linear regime. This requires that

$$V_{dp} \equiv V_{DD} - V_o \leq V_{DD} - V_i - V_{Tp}$$

Therefore, we must choose the negative sign. Thus

$$V_{DD} - V_o = (V_{DD} - V_i - V_{Tp}) - \sqrt{V_{DD} - V_i - V_{Tp})^2 - \beta(V_i - V_{Tn})^2} \quad (2.5)$$

Therefore,

$$V_o = V_i + V_{Tp} + \sqrt{V_{DD} - V_i - V_{Tp})^2 - \beta(V_i - V_{Tn})^2} \quad (2.6)$$

Since V_o must be $\geq V_i + V_{Tp}$, the limit of applicability of the above result is given by

$$(V_{DD} - V_i - V_{Tp})^2 = \beta(V_i - V_{Tn})^2$$

That is, the solution for V_o is valid for

$$V_i \leq \frac{V_{DD} + \sqrt{\beta}V_{Tn} - V_{Tp}}{1 + \sqrt{\beta}} \quad (2.7)$$

In the case where we size the n and p channel transistors such that

$$K_n = K_p; \text{ so } \beta = 1$$

we have

$$V_o = (V_i + V_{Tp}) + \sqrt{(V_{DD} - V_{Tn} - V_{Tp})(V_{DD} - 2V_i + V_{Tn} - V_{Tp})} \quad (2.8)$$

with

$$V_i \leq \frac{V_{DD} + V_{Tn} - V_{Tp}}{2}$$

nMOS saturated, pMOS saturated

At the limit of applicability of eq. 2.7, when the input voltage is exactly at

$$V_i = \frac{V_{DD} + \sqrt{\beta}V_{Tn} - V_{Tp}}{1 + \sqrt{\beta}} \quad (2.9)$$

both transistors are saturated. Since the currents of both transistors are independent of their drain voltages in this condition, we do not get a unique solution for V_o by equating drain currents.

The currents will be equal for all values of V_o in the range

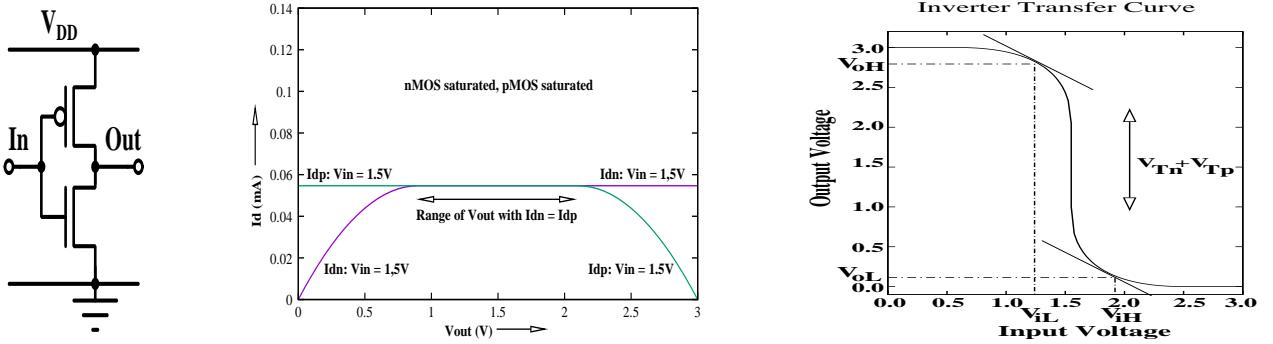


Figure 2.4: Output voltage when both transistors are saturated

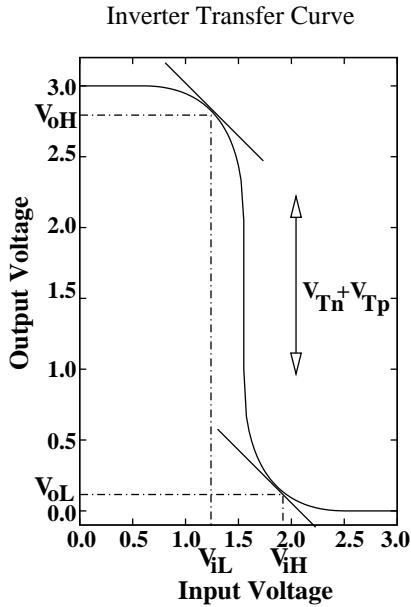


Figure 2.5: Transfer Curve of a CMOS inverter

$$V_i - V_{Tn} \leq V_o \leq V_i + V_{Tp}$$

Thus the transfer curve of an inverter shows a drop of $V_{Tn} + V_{Tp}$ at a voltage near $V_{DD}/2$. This is actually an artifact of the simple transistor model chosen for this analysis, which assumes perfect saturation of drain current. In a real case, the drain current does depend on the drain voltage (albeit weakly) in the saturation region. If the model incorporates an Early Voltage like effect, the drop near the middle of the characteristic is more gradual.

nMOS linear, pMOS saturated

At the gate voltage given by eq. 2.9, both transistors are saturated. As we increase V_i beyond this value, such that

$$\frac{V_{DD} + \sqrt{\beta}V_{Tn} - V_{Tp}}{1 + \sqrt{\beta}} < V_i < V_{DD} - V_{Tp}$$

both transistors are still ‘on’, but nMOS enters the linear regime while pMOS gets saturated. Equating currents in this condition,

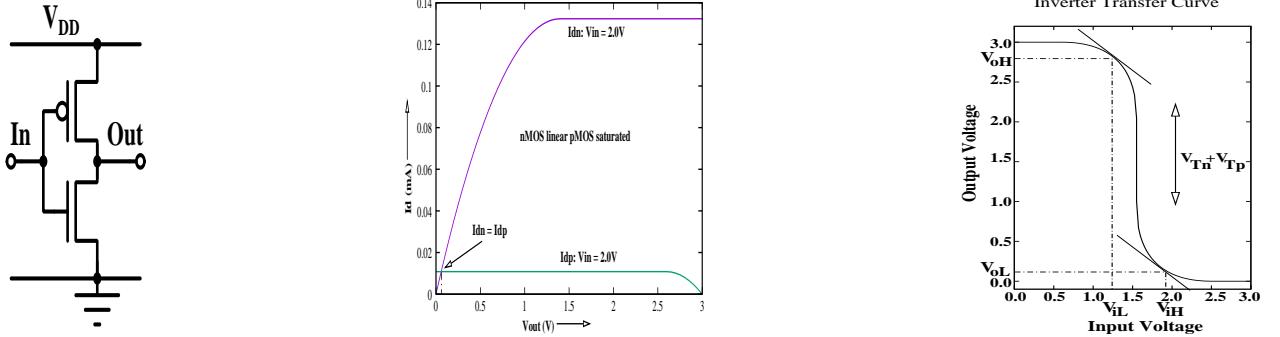


Figure 2.6: Output voltage when nMOS is in linear regime while pMOS is saturated.

$$I_d = \frac{K_p}{2}(V_{DD} - V_i - V_{Tp})^2 = K_n \left[(V_i - V_{Tn})V_o - \frac{1}{2}V_o^2 \right] \quad (2.10)$$

From this, we get the quadratic equation

$$\frac{1}{2}V_o^2 - (V_i - V_{Tn})V_o + \frac{(V_{DD} - V_i - V_{Tp})^2}{2\beta} = 0 \quad (2.11)$$

This has solutions

$$V_o = (V_i - V_{Tn}) \pm \sqrt{(V_i - V_{Tn})^2 - \frac{(V_{DD} - V_i - V_{Tp})^2}{\beta}} \quad (2.12)$$

Since the equations are valid only when the n channel transistor is in the linear regime ($V_o < V_i - V_{Tn}$), we choose the negative sign. This gives,

$$V_o = (V_i - V_{Tn}) - \sqrt{(V_i - V_{Tn})^2 - \frac{(V_{DD} - V_i - V_{Tp})^2}{\beta}} \quad (2.13)$$

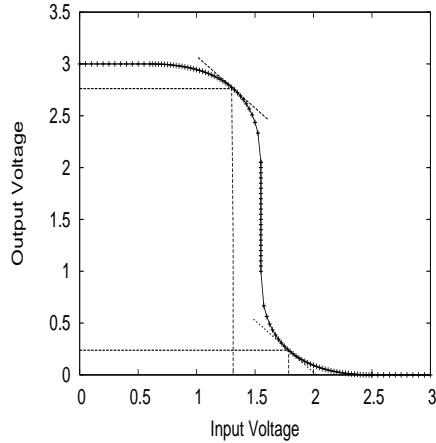
Again, in the special case where $\beta = 1$, we have

$$V_o = (V_i - V_{Tn}) - \sqrt{(V_{DD} - V_{Tn} - V_{Tp})(2V_i - V_{DD} - V_{Tn} + V_{Tp})} \quad (2.14)$$

nMOS ‘on’, pMOS ‘off’

As we increase the input voltage beyond $V_{DD} - V_{Tp}$, the p channel transistor turns ‘off’, while the n channel conducts strongly. As a result, the output voltage falls to zero. This is the normal digital operation range with input = ‘1’ and output = ‘0’.

The figure below shows the transfer curve of an inverter with $V_{DD} = 3V$, $V_{Tn} = 0.6V$ and $V_{Tp} = 0.5V$, and $\beta = 1$.



The plot produced by SPICE for this circuit with realistic models is quite similar.

2.2.2 Noise margins

The requirement from a digital circuit is that it should distinguish logic levels, but be insensitive to the exact analog voltage at the input. This implies that the flat portions of the transfer curve (where $\frac{\partial V_o}{\partial V_i}$ is small) are suitable for digital logic. We select two points on the transfer curve where the slope ($\frac{\partial V_o}{\partial V_i}$) is -1.0. The coordinates of these two points define the values of (V_{iL}, V_{oH}) and (V_{iH}, V_{oL}) . Robust digital design requires that the output high level be higher than what is acceptable as a high level at the input ($V_{oH} > V_{iH}$). The difference between these two levels is the ‘high’ noise margin. This is the amount of noise that can ride on the worst case ‘high’ output and still be accepted as a ‘high’ at the input of the next gate. Similarly, we require $V_{oL} < V_{iL}$. The difference, $V_{iL} - V_{oL}$ is the ‘low’ noise margin. Obviously, it is of interest to evaluate the values of these noise margins. For the discussion which follows, we shall use the expressions derived earlier for $\beta = 1$ to keep the algebra simple.

Calculation of V_{iL} and V_{oH}

from eq. (2.8)

$$V_o = (V_i + V_{Tp}) + \sqrt{(V_{DD} - V_{Tn} - V_{Tp})(V_{DD} + V_{Tn} - V_{Tp} - 2V_i)}$$

From this, we can evaluate $\frac{\partial V_o}{\partial V_i}$ and set it = -1.

$$\frac{\partial V_o}{\partial V_i} = -1 = 1 - \sqrt{\frac{V_{DD} - V_{Tn} - V_{Tp}}{V_{DD} + V_{Tn} - V_{Tp} - 2V_i}} \quad (2.15)$$

This gives

$$V_{iL} = \frac{3V_{DD} + 5V_{Tn} - 3V_{Tp}}{8} \quad (2.16)$$

Substituting this in eq.(2.8), we get

$$\begin{aligned} V_{oH} &= \frac{3V_{DD} + 5V_{Tn} - 3V_{Tp}}{8} + V_{Tp} + \\ &\quad \sqrt{(V_{DD} - V_{Tn} - V_{Tp}) \left(V_{DD} + V_{Tn} - V_{Tp} - \frac{3V_{DD} + 5V_{Tn} - 3V_{Tp}}{4} \right)} \\ &= \frac{3V_{DD} + 5V_{Tn} + 5V_{Tp}}{8} + \sqrt{(V_{DD} - V_{Tn} - V_{Tp}) \left(\frac{V_{DD} - V_{Tn} - V_{Tp}}{4} \right)} \\ &= \frac{3V_{DD} + 5V_{Tn} + 5V_{Tp}}{8} + \frac{V_{DD} - V_{Tn} - V_{Tp}}{2} \\ &= \frac{7V_{DD} + V_{Tn} + V_{Tp}}{8} \end{aligned}$$

So

$$V_{oH} = \frac{7V_{DD} + V_{Tn} + V_{Tp}}{8} = V_{DD} - \frac{V_{DD} - V_{Tn} - V_{Tp}}{8} \quad (2.17)$$

Calculation of V_{iH} and V_{oL}

When the input is ‘high’, we should use eq.(2.14).

$$V_o = (V_i - V_{Tn}) - \sqrt{(V_{DD} - V_{Tn} - V_{Tp})(2V_i - V_{DD} - V_{Tn} + V_{Tp})}$$

Differentiating with respect to V_i gives

$$\frac{\partial V_o}{\partial V_i} = -1 = 1 - \sqrt{\frac{V_{DD} - V_{Tn} - V_{Tp}}{2V_i - V_{DD} - V_{Tn} + V_{Tp}}} \quad (2.18)$$

$$\text{So } \frac{V_{DD} - V_{Tn} - V_{Tp}}{2V_i - V_{DD} - V_{Tn} + V_{Tp}} = 4$$

$$\text{Therefore } V_{DD} - V_{Tn} - V_{Tp} = 8V_i - 4V_{DD} - 4V_{Tn} + 4V_{Tp}$$

From where, we get

$$V_{iH} = \frac{5V_{DD} + 3V_{Tn} - 5V_{Tp}}{8} \quad (2.19)$$

Substituting the value of V_{iH} for V_{in} in eq.2.14), we get

$$\begin{aligned}
V_{oL} &= \frac{5V_{DD} + 3V_{Tn} - 5V_{Tp}}{8} - V_{Tn} - \\
&\quad \sqrt{(V_{DD} - V_{Tn} - V_{Tp}) \left(\frac{5V_{DD} + 3V_{Tn} - 5V_{Tp}}{4} - V_{DD} - V_{Tn} + V_{Tp} \right)} \\
&= \frac{5V_{DD} - 5V_{Tn} - 5V_{Tp}}{8} - \sqrt{(V_{DD} - V_{Tn} - V_{Tp}) \left(\frac{V_{DD} - V_{Tn} - V_{Tp}}{4} \right)} \\
&= \frac{5V_{DD} - 5V_{Tn} - 5V_{Tp}}{8} - \frac{V_{DD} - V_{Tn} - V_{Tp}}{2} = \frac{V_{DD} - V_{Tn} - V_{Tp}}{8}
\end{aligned}$$

So

$$V_{oL} = \frac{V_{DD} - V_{Tn} - V_{Tp}}{8} \quad (2.20)$$

Calculation of Noise Margins

The high noise margin is given by

$$V_{oH} - V_{iH} = \frac{V_{DD} - V_{Tn} + 3V_{Tp}}{4} \quad (2.21)$$

Similarly, the Low noise margin is

$$V_{iL} - V_{oL} = \frac{V_{DD} + 3V_{Tn} - V_{Tp}}{4} \quad (2.22)$$

The two noise margins can be made equal by choosing equal values for V_{Tn} and V_{Tp} .

2.2.3 Dynamic Considerations

In this section, we analyze the dynamic behaviour of the inverter. For the calculation of rise and fall times, we shall assume that only one of the two transistors in the inverter is ‘on’. (Notice that this is more conservative than the input high and low conditions determined by slope considerations in eq.2.19 and 2.16). We shall continue to use the simple model described at the beginning of this booklet.

Rise time

When the input is low, the n channel transistor is ‘off’, while the p channel transistor is ‘on’. The equivalent circuit in this condition is shown in fig. 2.7. From Kirchoff’s current law at the output node,

$$I_{dp} = C \frac{dV_o}{dt}$$

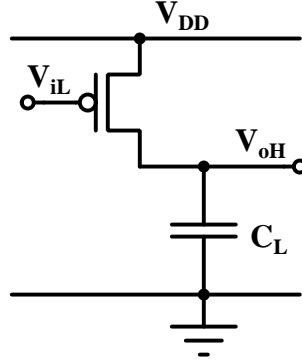


Figure 2.7: CMOS inverter with the nMOS ‘off’

so,

$$\frac{dt}{C} = \frac{dV_o}{I_{dp}}$$

This separates the variables, with the LHS independent of operating voltages and the RHS independent of time. Integrating both sides, we get

$$\frac{\tau_{rise}}{C} = \int_0^{V_{oH}} \frac{dV_o}{I_{dp}}$$

Till the output rises to $V_{iL} + V_{Tp}$, the p channel transistor is in saturation. Since the current is constant, the integration is trivial. If $V_{oH} > V_{iL} + V_{Tp}$ (which is normally the case), the integration range can be broken into saturation and linear regimes. Thus

$$\begin{aligned} \frac{\tau_{rise}}{C} &= \int_0^{V_{iL}+V_{Tp}} \frac{dV_o}{\frac{K_p}{2}(V_{DD} - V_{iL} - V_{Tp})^2} \\ &\quad + \int_{V_{iL}+V_{Tp}}^{V_{oH}} \frac{dV_o}{K_p \left[(V_{DD} - V_{iL} - V_{Tp})(V_{DD} - V_o) - \frac{1}{2}(V_{DD} - V_o)^2 \right]} \end{aligned}$$

We define $V_1 \equiv V_{DD} - V_o$ and $V_2 \equiv V_{DD} - V_{iL} - V_{Tp}$, so $dV_o = -dV_1$.

We get

$$\frac{K_p \tau_{rise}}{2C} = \frac{V_{iL} + V_{Tp}}{V_2^2} - \int_{V_2}^{V_{DD}-V_{oH}} \frac{dV_1}{2V_1 V_2 - V_1^2}$$

The integral can be evaluated as

$$\begin{aligned} I &\equiv - \int_{V_2}^{V_{DD}-V_{oH}} \frac{dV_1}{2V_1 V_2 - V_1^2} \\ &= \frac{1}{2V_2} \int_{V_{DD}-V_{oH}}^{V_2} \left(\frac{1}{V_1} + \frac{1}{2V_2 - V_1} \right) dV_1 \end{aligned}$$

$$\begin{aligned}
&= \frac{1}{2V_2} \left[\ln \frac{V_1}{2V_2 - V_1} \right]_{V_{DD}-V_{oH}}^{V_2} \\
&= \frac{1}{2V_2} \ln \frac{2V_2 - V_{DD} + V_{oH}}{V_{DD} - V_{oH}}
\end{aligned}$$

Therefore,

$$\frac{K_p \tau_{rise}}{2C} = \frac{V_{iL} + V_{Tp}}{V_2^2} + \frac{1}{2V_2} \ln \frac{2V_2 - V_{DD} + V_{oH}}{V_{DD} - V_{oH}}$$

or

$$\frac{K_p \tau_{rise}}{2C} = \frac{V_{iL} + V_{Tp}}{(V_{DD} - V_{iL} - V_{Tp})^2} + \frac{1}{2(V_{DD} - V_{iL} - V_{Tp})} \ln \frac{2V_2 - V_{DD} + V_{oH}}{V_{DD} - V_{oH}}$$

Thus,

$$\begin{aligned}
\tau_{rise} &= \frac{C(V_{iL} + V_{Tp})}{\frac{K_p}{2}(V_{DD} - V_{iL} - V_{Tp})^2} \\
&\quad + \frac{C}{K_p(V_{DD} - V_{iL} - V_{Tp})} \ln \frac{V_{DD} + V_{oH} - 2V_{iL} - 2V_{Tp}}{V_{DD} - V_{oH}}
\end{aligned} \tag{2.23}$$

The first term is just the constant current charging of the load capacitor. The second term represents the charging by the pMOS in its linear range. This can be compared with resistive charging, which would have taken a charge time of

$$\tau = RC \ln \frac{V_{DD} - V_{iL} - V_{Tp}}{V_{DD} - V_{oH}}$$

to charge from $V_{iL} + V_{Tp}$ to V_{oH} .

Fall time

When the input is high, the n channel transistor is ‘on’ and the p channel transistor is ‘off’. If the output was initially ‘high’, it will be discharged to ground through the nMOS. To analysis the fall time, we apply Kirchoff’s current law to the output node. This gives

$$I_{dn} = -C \frac{dV_o}{dt}$$

Again, separating variables and integrating from the initial voltage ($= V_{DD}$) to some terminal voltage V_{oL} gives

$$\frac{\tau_{fall}}{C} = - \int_{V_{DD}}^{V_{oL}} \frac{dV_o}{I_{dn}}$$

The n channel transistor will be in saturation till the output voltage falls to $V_i - V_{Tn}$. Below this voltage, the transistor will be in its linear regime. Thus, we can divide the integration

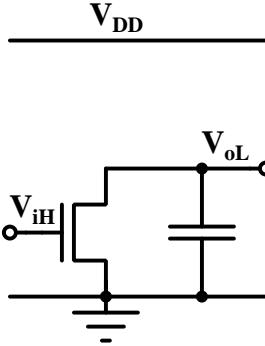


Figure 2.8: CMOS inverter with the pMOS ‘off’

range in two parts.

$$\begin{aligned} \frac{\tau_{fall}}{C} &= - \int_{V_{DD}}^{V_i - V_{Tn}} \frac{dV_o}{I_{dn}} - \int_{V_i - V_{Tn}}^{V_{oL}} \frac{dV_o}{I_{dn}} \\ &= \int_{V_i - V_{Tn}}^{V_{DD}} \frac{dV_o}{\frac{K_n}{2}(V_i - V_{Tn})^2} \\ &\quad + \int_{V_{oL}}^{V_i - V_{Tn}} \frac{dV_o}{K_n [(V_i - V_{Tn})V_o - \frac{1}{2}V_o^2]} \end{aligned}$$

Therefore

$$\begin{aligned} \frac{K_n \tau_{fall}}{2C} &= \frac{V_{DD} - V_i + V_{Tn}}{(V_i - V_{Tn})^2} + \int_{V_{oL}}^{V_i - V_{Tn}} \frac{dV_o}{2V_o(V_i - V_{Tn}) - V_o^2} \\ &= \frac{V_{DD} - V_i + V_{Tn}}{(V_i - V_{Tn})^2} + \frac{1}{2(V_i - V_{Tn})} \int_{V_{oL}}^{V_i - V_{Tn}} dV_o \left(\frac{1}{V_o} + \frac{1}{2(V_i - V_{Tn}) - V_o} \right) \end{aligned}$$

Which gives

$$\begin{aligned} \frac{K_n \tau_{fall}}{2C} &= \frac{V_{DD} - V_i + V_{Tn}}{(V_i - V_{Tn})^2} + \frac{1}{2(V_i - V_{Tn})} \left[\ln \frac{V_o}{2(V_i - V_{Tn}) - V_o} \right]_{V_{oL}}^{V_i - V_{Tn}} \\ &= \frac{V_{DD} - V_i + V_{Tn}}{(V_i - V_{Tn})^2} + \frac{1}{2(V_i - V_{Tn})} \ln \frac{2(V_i - V_{Tn}) - V_{oL}}{V_{oL}} \end{aligned}$$

and therefore

$$\tau_{fall} = \frac{C(V_{DD} - V_i + V_{Tn})}{\frac{K_n}{2}(V_i - V_{Tn})^2} + \frac{C}{K_n(V_i - V_{Tn})} \ln \frac{2(V_i - V_{Tn}) - V_{oL}}{V_{oL}} \quad (2.24)$$

Again, the first term represents the time taken to discharge at constant current in the saturation regime, whereas the second term is the quasi-resistive discharge in the linear regime.

2.2.4 Trade off between power, speed and robustness

As we scale technologies, we improve speed and power consumption. However, as we can see from the expression for noise margins, (eq 2.21 and eq 2.22) the noise margin becomes worse. We can improve noise margins by choosing relatively higher threshold voltages. However, this will reduce speeds. We could also increase V_{DD} - but that would increase power dissipation. Thus we have a trade off between power, speed and noise margins.

This choice is made much more complicated by process variations, because we have to design for the worst case.

2.2.5 CMOS Inverter Design Flow

The CMOS inverter forms the basis of most static CMOS logic design. More complex logic can be designed from it by simple thumb rules. A common (though not universal) design requirement is symmetric charge and discharge behaviour and equal noise margins for high and low logic values. This requires matched values of K_n and K_p and equal values of V_{Tn} and V_{Tp} . For a constant load capacitance, rise and fall times depend linearly on K_n and K_p . Thus it is a straightforward calculation to determine transistor geometries if speed requirements and technological parameters are given. However, as transistor geometries are made larger, self loading can become significant. We now have to model the load capacitance as

$$C_{Load} = C_{ext} + \alpha K_n$$

where we have assumed that $\beta = K_n/K_p$ is kept constant. α is a technological constant. We use the expressions for $K\tau/C$ which depend only on voltages. Once these values are calculated, the geometry can be determined.

In the extreme case, when self capacitance dominates the load capacitance, K/C becomes constant and τ becomes geometry independent. There is no advantage in using wider transistors in this regime to increase the speed. It is better to use multi-stage logic with tapered buffers in this regime. This will be discussed in the module on Logical Effort.

2.2.6 Conversion of CMOS Inverters to other logic

Once the basic CMOS inverter is designed, other logic gates can be derived from it. The logic has to be put in a canonical form which is a sum of products with a bar (inversion) on top. For every ‘.’ in the expression, we put the corresponding n channel transistors in series and the corresponding p channel transistors in parallel. for every ‘+’, we put the n channel transistors in parallel and the p channel transistors in series. We scale the transistor widths up by the number of devices (n or p) put in series. The geometries are left untouched for devices put in parallel. Fig.2.9 shows the implementation of $A \cdot B + C \cdot (\bar{D} + E)$ in CMOS logic design style.

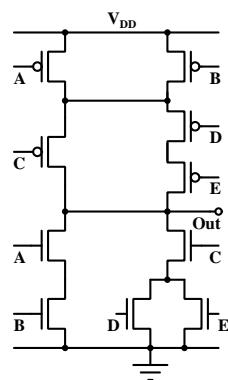


Figure 2.9: CMOS implementation of $\overline{A.B + C.(D + E)}$

Chapter 3

Pseudo-nMOS Logic Design

CMOS design style ensures that the logic consumes no static power. This is because the pull down and pull up networks are never ‘on’ simultaneously. However, this requires that signals have to be routed to the n pull down network *as well as* to the p pull up network. This means that the load presented to every driver is high. This fact is exacerbated by the fact that n and p channel transistors cannot be placed close together as these are in different wells which have to be kept well separated in order to avoid latch-up.

Pseudo nMOS design style reduces dynamic power (by reducing capacitive loading) at the cost of having non-zero static power by replacing the pull up network by a single pMOS transistor with its gate terminal grounded. The pseudo nMOS inverter is shown below. Notice

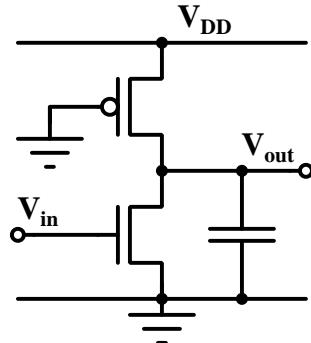


Figure 3.1: Pseudo nMOS inverter

that since the pMOS is not driven by signals, it is always ‘on’. The effective gate voltage seen by the pMOS transistor is V_{DD} . Thus the over-voltage on the p channel gate is always $V_{DD} - V_{T_p}$. This transistor will be saturated for output voltage $\leq V_{T_p}$ and will be in the linear regime for output voltage $\geq V_{T_p}$.

When the nMOS is turned ‘on’, a direct current path exists between supply and ground and so, static power will be dissipated.

3.1 Static Characteristics

As we sweep the input voltage from ground to V_{DD} , we encounter the following regimes of operation:

- For $V_i \leq V_{Tn}$, nMOS is ‘off’.
- For low input voltage ($> V_{Tn}$), nMOS is saturated, pMOS is in linear regime.
- As V_i is raised further, V_o continues to fall. Eventually we enter the regime where nMOS is linear and pMOS is also linear.
- Finally, as we keep raising V_i , the output falls below V_{Tp} , provided the nMOS transistor is sufficiently wide. Now the nMOS is in linear regime, while pMOS is saturated.

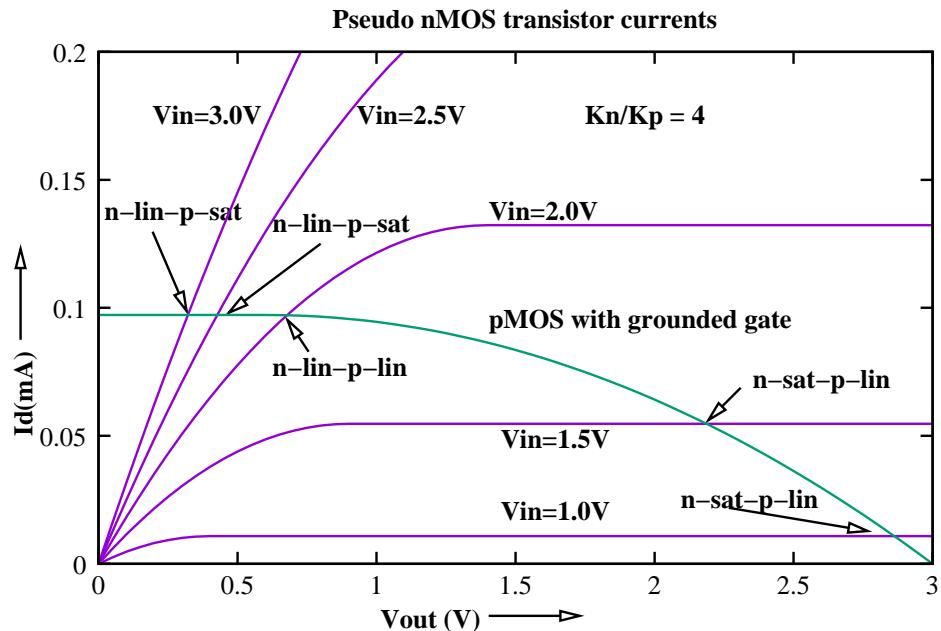


Figure 3.2: Transistor currents in a Pseudo nMOS inverter

3.1.1 For $0 \leq V_i \leq V_{Tn}$: nMOS ‘off’

When the input voltage is less than V_{Tn} , the n channel transistor is off. Since no current flows through the p channel transistor, its drain-source voltage must be zero. Thus the output is ‘high’ (and = V_{DD}). No static power is dissipated in this condition.

3.1.2 nMOS saturated, pMOS linear

As the input voltage is raised above V_{Tn} , we enter this region. At this stage, the output voltage is close to V_{DD} (so more than $V_i - V_{Tn}$). Therefore the n channel transistor will be in saturation. The output voltage is also much higher than V_{Tp} , so the p channel transistor is in linear mode of operation. Equating currents through the n and p channel transistors, we get

$$\frac{K_n}{2}(V_i - V_{Tn})^2 = K_p \left[(V_{DD} - V_{Tp})(V_{DD} - V_o) - \frac{1}{2}(V_{DD} - V_o)^2 \right] \quad (3.1)$$

defining $\beta \equiv K_n/K_p$, $V_1 \equiv V_{DD} - V_o$ and $V_2 \equiv V_{DD} - V_{Tp}$, we get

$$\frac{1}{2}V_1^2 - V_2V_1 + \frac{\beta}{2}(V_i - V_{Tn})^2 = 0 \quad (3.2)$$

with solutions

$$V_1 = V_2 \pm \sqrt{V_2^2 - \beta(V_i - V_{Tn})^2}$$

substituting the values of V_1 and V_2 and choosing the sign which puts V_o in the correct range, we get

$$V_o = V_{Tp} + \sqrt{(V_{DD} - V_{Tp})^2 - \beta(V_i - V_{Tn})^2} \quad (3.3)$$

This expression is valid as long as the n channel transistor is in saturation and the p channel transistor is in linear regime. As we keep increasing the input voltage V_i , the output will keep falling. Eq.3.3 will no more be valid when either the output falls below V_{Tp} , so that the p channel transistor enters saturation, or it falls below $V_i - V_{Tn}$, so that the n channel transistor enters the linear regime.

The output will fall below V_{Tp} when

$$V_{DD} - V_{Tp} = \sqrt{\beta(V_i - V_{Tn})} \quad \text{or} \quad V_i = V_{Tn} + \frac{V_{DD} - V_{Tp}}{\sqrt{\beta}}$$

On the other hand it will fall to $V_i - V_{Tn}$ when

$$V_i - V_{Tn} = V_{Tp} + \sqrt{(V_{DD} - V_{Tp})^2 - \beta(V_i - V_{Tn})^2}$$

This can be solved to show that nMOS will enter its linear regime when

$$V_i > V_{Tn} + \frac{V_{Tp} + \sqrt{V_{Tp}^2 + (\beta + 1)V_{DD}(V_{DD} - 2V_{Tp})}}{\beta + 1}. \quad (3.4)$$

For common values of parameters, this happens before pMOS enters saturation.

3.1.3 nMOS linear, pMOS linear

$$\text{for } V_i > V_{Tn} + \frac{V_{Tp} + \sqrt{V_{Tp}^2 + (\beta + 1)V_{DD}(V_{DD} - 2V_{Tp})}}{\beta + 1}.$$

the nMOS enters its linear mode of operation. At this point the output voltage is $> V_{Tp}$, so the pMOS transistor is also in its linear regime. This combination of nMOS as well as pMOS being in linear regime will continue as we increase V_i , till V_o falls to V_{Tp} .

To determine the output voltage when both transistors are in their linear regimes, We can again equate the nMOS and pMOS currents using the transistor current equations for linear regime. The solution is straightforward, though algebraically tedious.

Equating n and p channel transistor currents,

$$K_n \left[(V_i - V_{Tn})V_o - \frac{1}{2}V_o^2 \right] = K_p \left[(V_{DD} - V_{Tp})(V_{DD} - V_o) - \frac{1}{2}(V_{DD} - V_o)^2 \right] \quad (3.5)$$

Defining $\beta \equiv K_n/K_p$, we can write

$$\begin{aligned} \beta \left[(V_i - V_{Tn})V_o - \frac{1}{2}V_o^2 \right] &= V_{DD}^2 - V_{Tp}V_{DD} - V_oV_{DD} + V_oV_{Tp} - \frac{1}{2}(V_{DD}^2 + V_o^2 - 2V_{DD}V_o) \\ \text{or } \beta(V_i - V_{Tn})V_o - \frac{\beta}{2}V_o^2 &= \frac{1}{2}V_{DD}^2 - V_{Tp}V_{DD} + V_{Tp}V_o - \frac{1}{2}V_o^2 \end{aligned}$$

This gives

$$\frac{\beta - 1}{2}V_o^2 - (\beta(V_i - V_{Tn}) - V_{Tp})V_o + \frac{V_{DD}}{2}(V_{DD} - 2V_{Tp}) = 0 \quad (3.6)$$

Solving this quadratic will give the value of V_o .

$$V_o = \frac{\beta(V_i - V_{Tn}) - V_{Tp} \pm \sqrt{(\beta(V_i - V_{Tn}) - V_{Tp})^2 - (\beta - 1)V_{DD}(V_{DD} - 2V_{Tp})}}{\beta - 1}$$

To keep V_o in the correct operating range, and because V_o is a decreasing function of V_i , we choose the negative sign. Thus,

$$V_o = \frac{\beta(V_i - V_{Tn}) - V_{Tp} - \sqrt{(\beta(V_i - V_{Tn}) - V_{Tp})^2 - (\beta - 1)V_{DD}(V_{DD} - 2V_{Tp})}}{\beta - 1} \quad (3.7)$$

3.1.4 nMOS linear, pMOS saturated

As the input voltage is raised still further, the output voltage will reach and then fall below V_{Tp} . The nMOS continues in its linear regime, and now the pMOS transistor enters its saturation regime.

Equating currents, we get

$$K_n \left[(V_i - V_{Tn})V_o - \frac{1}{2}V_o^2 \right] = \frac{K_p}{2}(V_{DD} - V_{Tp})^2$$

which gives

$$\frac{1}{2}V_o^2 - (V_o - V_{Tn})V_o + \frac{(V_{DD} - V_{Tp})^2}{2\beta}$$

This can be solved to get

$$V_o = (V_i - V_{Tn}) - \sqrt{(V_i - V_{Tn})^2 - (V_{DD} - V_{Tp})^2 / \beta} \quad (3.8)$$

3.2 Noise margins

As in the case of CMOS inverter, we find points on the transfer curve where the slope is -1.

When the input is low and output high, we should use Eq(3.3). Differentiating this equation with respect to V_i and setting the slope to -1, we get

$$V_{iL} = V_{Tn} + \frac{V_{DD} - V_{Tp}}{\sqrt{\beta(\beta + 1)}} \quad (3.9)$$

and

$$V_{oH} = V_{Tp} + \sqrt{\frac{\beta}{\beta + 1}} (V_{DD} - V_{Tp}) \quad (3.10)$$

When the input is high and the output low, we use Eq(3.8). Again, differentiating with respect to V_i and setting the slope to -1, we get

$$V_{iH} = V_{Tn} + \frac{2}{\sqrt{3\beta}} (V_{DD} - V_{Tp}) \quad (3.11)$$

and

$$V_{oL} = \frac{(V_{DD} - V_{Tp})}{\sqrt{3\beta}} \quad (3.12)$$

To make the output ‘low’ value lower than V_{Tn} , we get the condition

$$\beta > \frac{1}{3} \left(\frac{V_{DD} - V_{Tp}}{V_{Tn}} \right)^2$$

This condition on values of β places a requirement on the ratios of widths of n and p channel transistors. The logic gates work properly only when this equation is satisfied. Therefore this kind of logic is also called ‘ratioed logic’. In contrast, CMOS logic is called ratioless logic because it does not place any restriction on the ratios of widths of n and p channel transistors for static operation. The noise margin for pseudo nMOS can be determined easily from the expressions for V_{iL} , V_{oL} , V_{iH} , V_{oH} .

3.3 Dynamic characteristics

In the sections above, we have derived the behaviour of a pseudo nMOS inverter in static conditions. In the sections below, we discuss the dynamic behaviour of this inverter.

3.3.1 Rise Time

When the input is low and the output rises from ‘low’ to ‘high’, the nMOS is off. The situation is identical to the charge up condition of a CMOS gate with the pMOS being biased with its gate at 0V. This gives

$$\tau_{rise} = \frac{C}{K_p(V_{DD} - V_{Tp})} \left[\frac{2V_{Tp}}{V_{DD} - V_{Tp}} + \ln \frac{V_{DD} + V_{oH} - 2V_{Tp}}{V_{DD} - V_{oH}} \right] \quad (3.13)$$

3.3.2 Fall Time

Analytical calculation of fall time is complicated by the fact that the pMOS load continues to dump current in the output node, even as the nMOS tries to discharge the output capacitor.

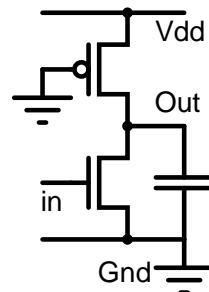


Figure 3.3: ‘high’ to ‘low’ transition on the output

Thus the nMOS should sink the discharge current as well as the drain current of the pMOS transistor. We make the simplifying assumption that the pMOS current remains constant at its saturation value through the entire discharge process. (This will result in a slightly pessimistic value of discharge time). Then,

$$I_p = \frac{K_p}{2}(V_{DD} - V_{Tp})^2$$

. We can write the KCL equation at the output node as:

$$I_n - I_p + C \frac{dV_o}{dt} = 0$$

which gives

$$\frac{\tau_{fall}}{C} = - \int_{V_{DD}}^{V_{oL}} \frac{dV_o}{I_n - I_p}$$

We define $V_1 \equiv V_i - V_{Tn}$ and $V_2 \equiv V_{DD} - V_{Tp}$. The integration range can be divided into two regimes. nMOS is saturated when $V_1 \leq V_o < V_{DD}$ and is in linear regime when $V_{oL} < V_o < V_1$. Therefore,

$$\frac{\tau_{fall}}{C} = - \int_{V_{DD}}^{V_1} \frac{dV_o}{\frac{1}{2}K_n V_1^2 - I_p} - \int_{V_1}^{V_{oL}} \frac{dV_o}{K_n(V_1 V_o - \frac{1}{2}V_o^2) - I_p}$$

$$\text{So } \frac{K_n \tau_{fall}}{2C} = \frac{V_{DD} - V_1}{V_1^2 - 2I_p/K_n} + \int_{V_{oL}}^{V_1} \frac{dV_o}{2V_1 V_o - V_o^2 - 2I_p/K_n}$$

We define $V_2^2 \equiv 2I_p/K_n$. The denominator of the integral term can then be factorized by adding and subtracting V_1^2

$$\begin{aligned} 2V_1 V_o - V_o^2 - V_1^2 + V_1^2 - V_2^2 &= -(V_1 - V_o)^2 + \left(\sqrt{V_1^2 - V_2^2} \right)^2 \\ &= \left(\sqrt{V_1^2 - V_2^2} + V_1 - V_o \right) \left(\sqrt{V_1^2 - V_2^2} - V_1 + V_o \right) \end{aligned}$$

The integral term can then be written as:

$$\frac{1}{2\sqrt{V_1^2 - V_2^2}} \left(\int_{V_{oL}}^{V_1} \frac{dV_o}{\sqrt{V_1^2 - V_2^2} + V_1 - V_o} + \int_{V_{oL}}^{V_1} \frac{dV_o}{\sqrt{V_1^2 - V_2^2} - (V_1 - V_o)} \right)$$

The integration over V_o can now be carried out to get

$$\frac{1}{2\sqrt{V_1^2 - V_2^2}} \ln \frac{\sqrt{V_1^2 - V_2^2} - (V_1 - V_o)}{\sqrt{V_1^2 - V_2^2} + V_1 - V_o} \Big|_{V_{oL}}^{V_1}$$

On putting the limits for the integral, the upper limit gives 0. So the integral can be written as

$$\frac{1}{2\sqrt{V_1^2 - V_2^2}} \ln \frac{\sqrt{V_1^2 - V_2^2} + V_1 - V_{oL}}{\sqrt{V_1^2 - V_2^2} - (V_1 - V_{oL})}$$

Thus we can write down the expression for fall time as:

$$\frac{K_n \tau_{fall}}{2C} = \frac{V_{DD} - V_1}{V_1^2 - 2I_p/K_n} + \frac{1}{2\sqrt{V_1^2 - V_2^2}} \ln \frac{\sqrt{V_1^2 - V_2^2} + V_1 - V_{oL}}{\sqrt{V_1^2 - V_2^2} - (V_1 - V_{oL})}$$

Substituting back for V_2^2 , we get

$$\frac{K_n \tau_{fall}}{2C} = \frac{V_{DD} - V_1}{V_1^2 - 2I_p/K_n} + \frac{1}{2\sqrt{V_1^2 - 2I_p/K_n}} \ln \frac{\sqrt{V_1^2 - 2I_p/K_n} + V_1 - V_{oL}}{\sqrt{V_1^2 - 2I_p/K_n} - (V_1 - V_{oL})}$$

Since $I_p = K_p(V_{DD} - V_{Tp})^2/2$, $2I_p/K_n$ is just $(V_{DD} - V_{Tp})^2/\beta$.

$$\frac{K_n \tau_{fall}}{2C} = \frac{V_{DD} - V_1}{V_1^2 - 2I_p/K_n} + \frac{1}{2\sqrt{V_1^2 - 2I_p/K_n}} \ln \frac{\sqrt{V_1^2 - 2I_p/K_n} + V_1 - V_{oL}}{\sqrt{V_1^2 - 2I_p/K_n} - (V_1 - V_{oL})} \quad (3.14)$$

$$\text{Where } 2I_p/K_n = \frac{(V_{DD} - V_{Tp})^2}{\beta} \quad \text{and} \quad V_1 = V_i - V_{Tn}$$

This relation was derived using the pessimistic assumption that the p channel transistor dumps its saturation current over the entire discharge range.

In any case, this relation is not used for designing the inverter because the limit on the size of the n channel transistor is put by static considerations and not by the fall time.

3.4 Pseudo nMOS design Flow

We design the basic inverter first and then map the inverter design to other logic circuits. The load device size is calculated from the rise time. From Eq. 3.13 we have

$$\tau_{rise} = \frac{C}{K_p(V_{DD} - V_{Tp})} \left[\frac{2V_{Tp}}{V_{DD} - V_{Tp}} + \ln \frac{V_{DD} + V_{oH} - 2V_{Tp}}{V_{DD} - V_{oH}} \right]$$

Given a value of τ_{rise} , operating voltages and technological constants, K_p and hence, the geometry of the p channel transistor can be determined.

Geometry of the n channel transistor in the reference inverter design can be determined from static considerations. Using Eq. 3.8, the output ‘low’ level is given by:

$$V_o = (V_i - V_{Tn}) - \sqrt{(V_i - V_{Tn})^2 - (V_{DD} - V_{Tp})^2/\beta}$$

If the desired value of the output ‘low’ level is given, we can calculate β . But $\beta \equiv K_n/K_p$ and K_p is already known. This evaluates K_n and hence, the geometry of the n channel transistor.

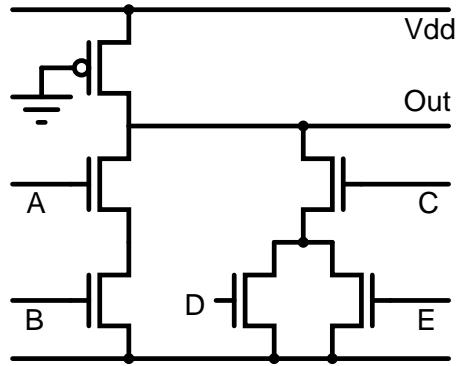


Figure 3.4: Pseudo NMOS implementation of $\overline{A.B + C.(D + E)}$

3.5 Conversion of pseudo nMOS Inverter to other logic

Once the basic pseudo nMOS inverter is designed, other logic gates can be derived from it. The procedure is the same as that for CMOS, except that it is applied only to nMOS transistors. The p channel transistor is kept at the same size as that for an inverter.

The logic is expressed as a sum of products with a bar (inversion) on top. For every ‘.’ in the expression, we put the corresponding n channel transistors in series and for every ‘+’, we put the n channel transistors in parallel. We scale the transistor widths up by the number of devices put in series. The geometries are left untouched for devices put in parallel. Fig.3.4 shows the implementation of $\overline{A.B + C.(D + E)}$ in pseudo NMOS logic design style.

Chapter 4

Dual Rail Logic Design

4.1 Complementary Pass gate Logic

This logic family is based on multiplexer logic and makes use of Shannon's Boolean expansion theorem.

Given a Boolean function $F(x_1, x_2, \dots, x_n)$, we can express it as:

$$F(x_1, x_2, \dots, x_n) = x_i \cdot f_1 + \overline{x_i} \cdot f_2$$

where f_1 and f_2 are reduced expressions for F with x_i forced to '1' and '0' respectively. Thus, F can be implemented with a multiplexer controlled by x_i which selects f_1 or f_2 depending on x_i . f_1 and f_2 can themselves be decomposed into simpler expressions by the same technique.

To implement a multiplexer, we need both x_i and $\overline{x_i}$. Therefore, this logic family needs all inputs in true as well as in complement form. In order to drive other gates of the same type, it must produce the outputs also in true and complement forms. Thus each signal is carried by two wires. This logic style is called "Complementary Pass-gate Logic" or CPL for short.

4.1.1 Basic Multiplexer Structure

Pure pass-gate logic contains no 'amplifying' elements. Therefore, it has zero or negative noise margin. (Each logic stage degrades the logic level). Therefore, multiple logic stages cannot be cascaded. We shall assume that each stage includes conventional CMOS inverters to restore the logic level. Ideally, the multiplexer should be composed of complementary pass gate transistors. However, we shall use just n channel transistors as switches for simplicity. This gives us the multiplexer structure shown in fig.4.1.

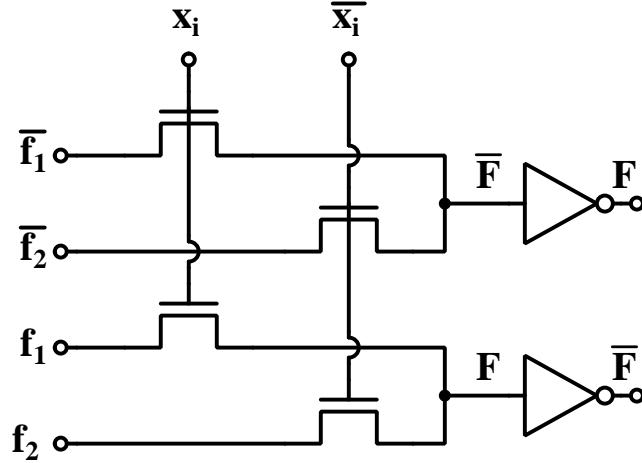


Figure 4.1: Basic Multiplexer with logic restoring inverters

4.1.2 Logic Design using CPL

Since both true and complement outputs are generated by CPL, we do not need separate gates for AND and NAND functions. The same applies to OR-NOR, and XOR-XNOR functions.

To take an example, let us consider the XOR-XNOR functions. Because of the inverter,

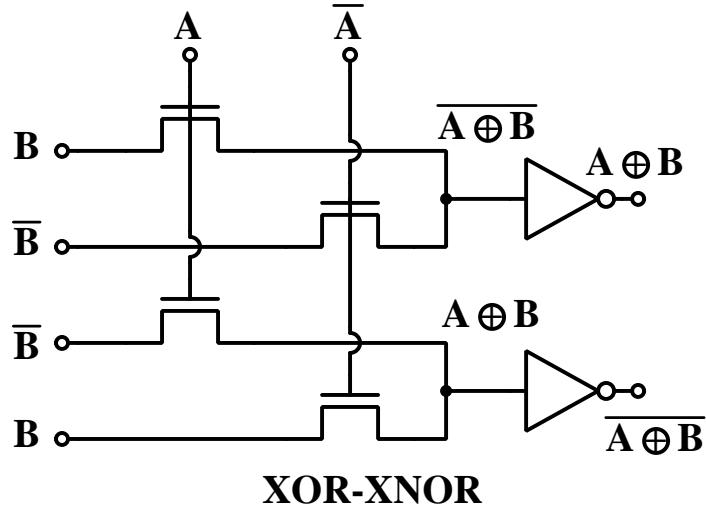


Figure 4.2: Implementation of XOR and XNOR by CPL logic.

the multiplexer for the XOR output first calculates the XNOR function given by $A.B + \bar{A}.\bar{B}$.

If we put $A = '1'$, this reduces to B and for $A = '0'$, it reduces to \bar{B} . Similarly, for the XNOR output, we generate the XOR expression $= A\bar{B} + \bar{A}B$ which will be inverted by the logic level restoring inverter. The expression reduces to \bar{B} for $A = '1'$ and to B for $A = '0'$. This leads to an implementation of XOR-XNOR as shown in fig.4.2

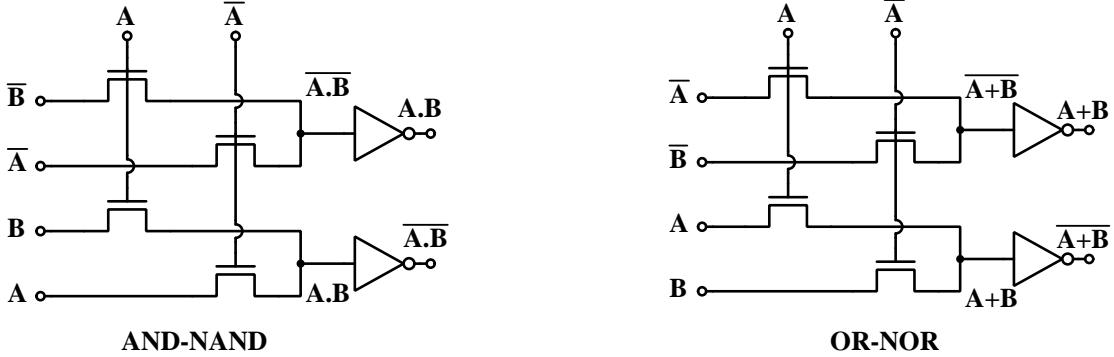


Figure 4.3: Implementation of (a) AND-NAND and (b) OR-NOR functions using complementary pass-gate logic.

Implementation of AND and OR functions is similar. In case of AND, the multiplexer should output $\overline{A \cdot B}$ to be inverted by the buffer. This reduces to \bar{B} when $A = '1'$. When $A = '0'$, it evaluates to $1 = \bar{A}$. For NAND output, the multiplexer should output $A \cdot B$, which evaluates to B for $A = '1'$ and to $'0'$ (or A) when $A = '0'$.

4.1.3 Buffer Leakage Current

The circuit configuration described above uses nMOS multiplexers. This limits the ‘high’

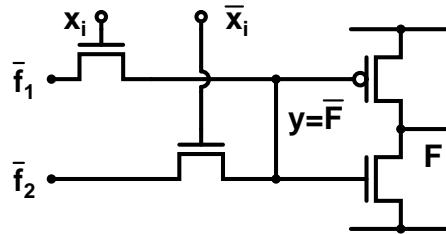


Figure 4.4: High leakage current in inverter

output of the multiplexer (node y - which is the input for the inverter) to $V_{DD} - V_{Tn}$. Consequently, the pMOS transistor in the buffer inverter never quite turns off. This results in static power consumption in the inverter. This can be avoided by adding a pull up pMOS as shown

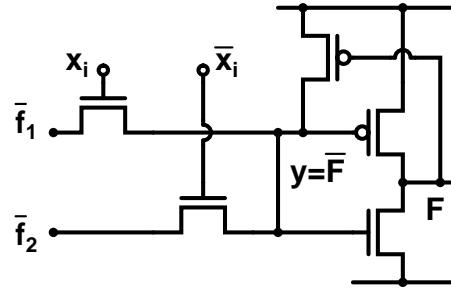


Figure 4.5: Pull up pMOS to avoid leakage in the inverter

in fig. 4.5. When the multiplexer output (y) is ‘low’, the inverter output is high. The pMOS is therefore off and has no effect. When the multiplexer output goes ‘high’, the inverter input charges up, the output starts falling and turns the pMOS on. Now, as the multiplexer output (y) approaches $V_{DD} - V_{Tn}$, the nMOS switch in the multiplexer turn off. However, the pMOS pull up remains ‘on’ and takes the inverter input all the way to V_{DD} . This avoids leakage in the inverter.

However, this solution brings up another problem. Consider the equivalent circuit when the inverter output is ‘low’ and the pMOS is ‘on’. Now if the multiplexer output wants to go

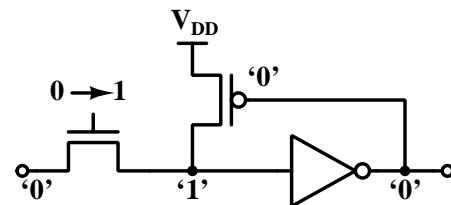


Figure 4.6: Problem with a low to high transition on the output

‘low’, it has to fight the pMOS pull-up - which is trying to keep this node ‘high’.

In fact, the multiplexer n transistor and the pull up p transistor constitute a pseudo nMOS inverter. Therefore, the multiplexer output cannot be pulled low unless the transistor geometries are appropriately ratioed.

4.2 Cascade Voltage Switch Logic

We can understand this logic configuration as an attempt to improve pseudo-nMOS logic circuits. Consider the NOR gate shown below: Static power is consumed by this NOR circuit

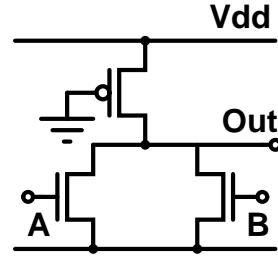


Figure 4.7: Pseudo-nMOS NOR

whenever the output is ‘LOW’. This happens when A OR B is TRUE. We wish that the pMOS could be turned off for just this combination of inputs.

To turn the pMOS transistor off, we need to apply a ‘HIGH’ voltage level to its gate whenever A OR B is true. This obviously requires an OR gate. Non-inverting gates cannot be made in a single stage. However, We can create the OR function by using a NAND of \bar{A} and \bar{B} as shown in figure 4.8. But then what about the pMOS drive of this circuit?

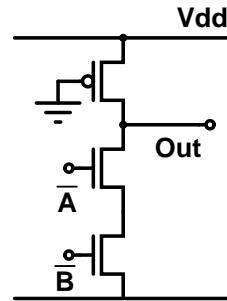


Figure 4.8: Pseudo-nMOS OR from complemented inputs

We want to turn the pMOS of this OR circuit off when both \bar{A} and \bar{B} are ‘HIGH’; i.e. when $A = B = 0$. This means we would like to turn the pMOS of this circuit off when the NOR of A and B is ‘TRUE’.

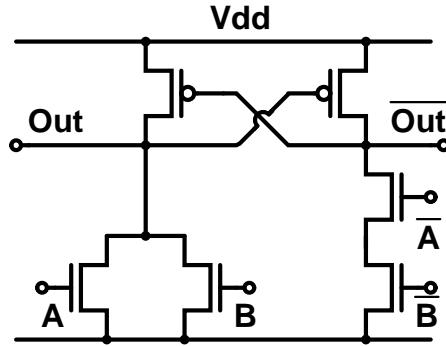


Figure 4.9: OR-NOR implementation in Cascade Voltage Switch Logic

But we already have this signal as the output of the first (NOR) circuit! So the two circuits can drive each other's pMOS transistors and avoid static power consumption. This kind of logic is called Cascade Voltage Switch Logic (CVSL). It can use any network f and its complementary network \bar{f} in the two cross-coupled branches. The complementary network is constructed by changing all series connections in f to parallel and all parallel connections to series, and complementing all input signals.

CVSL shares many characteristics with static CMOS, CPL and pseudo-nMOS.

- Like CMOS static logic, there is no static power consumption.
- Like CPL, this logic requires both True and Complement signals. It also provides both True and complement outputs. (Dual Rail Logic).
- Like pseudo nMOS, the inputs present a single transistor load to the driving stage.
- The circuit is self latching. This reduces ratioing requirements.

Chapter 5

Dynamic Logic

In this style of logic, some nodes are required to hold their logic value as a charge stored on a capacitor. These nodes are not connected to their ‘drivers’ permanently. The ‘driver’ places the logic value on them, and is then disconnected from the node. Due to leakage etc., the logic value cannot be held indefinitely. Dynamic circuits therefore require a *minimum* clock frequency to operate correctly. Use of dynamic circuits can reduce circuit complexity and power consumption substantially, compared to pseudo NMOS.

5.1 Basic CMOS Dynamic Gate

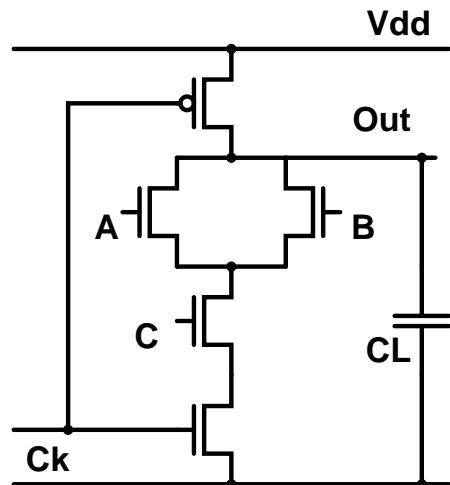


Figure 5.1: CMOS dynamic gate to implement $\overline{(A + B)}.C$.

When the clock is low, pMOS is on and the bottom nMOS is off. The output is ‘pre-

charged' to '1' unconditionally. When the clock goes high, the pMOS turns off and the bottom nMOS comes on. The circuit then conditionally discharges the output node, if $(A+B).C$ is TRUE. This implements the function $(\overline{A} + \overline{B}).\overline{C}$.

5.1.1 Problem with Cascading CMOS dynamic logic

The dynamic circuit described above can run into problems when several dynamic gates are cascaded. Consider the case when the circuit described in Fig. 5.1 is followed by an inverter. Let us take two cases – when $(A+B).C$ is FALSE and when it is TRUE.

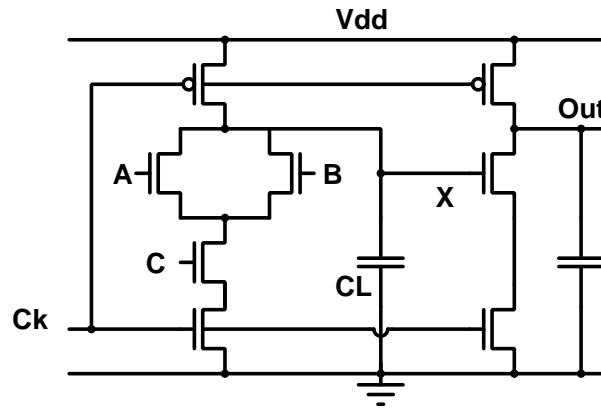
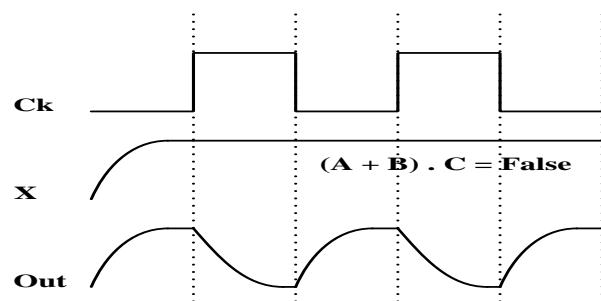
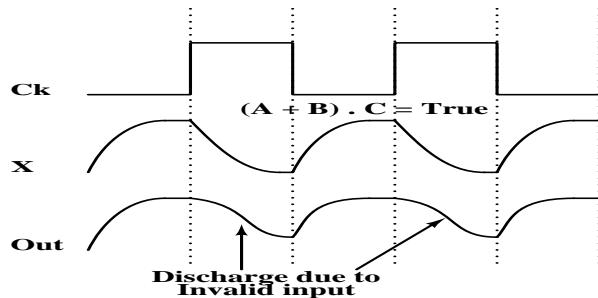


Figure 5.2: Dynamic gate for $(\overline{A} + \overline{B}).\overline{C}$ cascaded with an inverter.

When $(A+B).C$ is FALSE, There is no problem X pre-charges to '1' and remains at '1'. Therefore the inverter sees the correct logic value at its input all the time and discharges the output to '0', which is the expected output.



However, When $(A+B).C$ is TRUE, there is a problem.
The correct value for X is now '0'. After the pre-charge cycle, X takes some time to discharge



to '0'. So for some time after pre-charge, its output is held at the wrong value of '1'. During this time, the charge placed on the output capacitor of inverter is leaked away to ground as the input to nMOS of the inverter is not '0' and the clock input is high. This can lead to a wrong evaluation of the logic function!

In a dynamic logic stage, the output is pre-charged to '1'. If the final output is supposed to be '0', there will be some time during which the output will still be at the wrong value of '1'. This transiently wrong value can discharge the pre-charged output capacitor of the next stage and thus lead to malfunction. So we need to isolate the output of a dynamic logic stage from the input of the next dynamic logic stage till it has acquired the correct value.

This means the operation of dynamic logic should proceed in distinct time slots or 'phases'. The output of the stage should be connected to the input of the next stage only in the phases in which its output is valid and stable. So the sequence of operations should be:

- Pre-charge the output in the first phase.
- Disable pre-charging and carry out logic evaluation in the next stage. The Output should be disconnected from the next stage during pre-charge as well as evaluation phases.
- In the final phase(s), the output holds its correct value. In this phase, connect the output to the next stage and disconnected it from pre-charge and evaluate circuits of the current stage.
- Thus, a minimum of 3 phases are required – pre-charge, evaluate and output-valid.
- In a 4-phase implementation, the valid state holds for a duration of two phases instead of just one.

- To implement this kind of dynamic logic, we need a multi-phase clock.

An example scheme for generating a 4 phase clock is shown below.

The nor gate inserts a 1 in the shift register whenever it sees three 0's at Q0, Q1 and Q2.

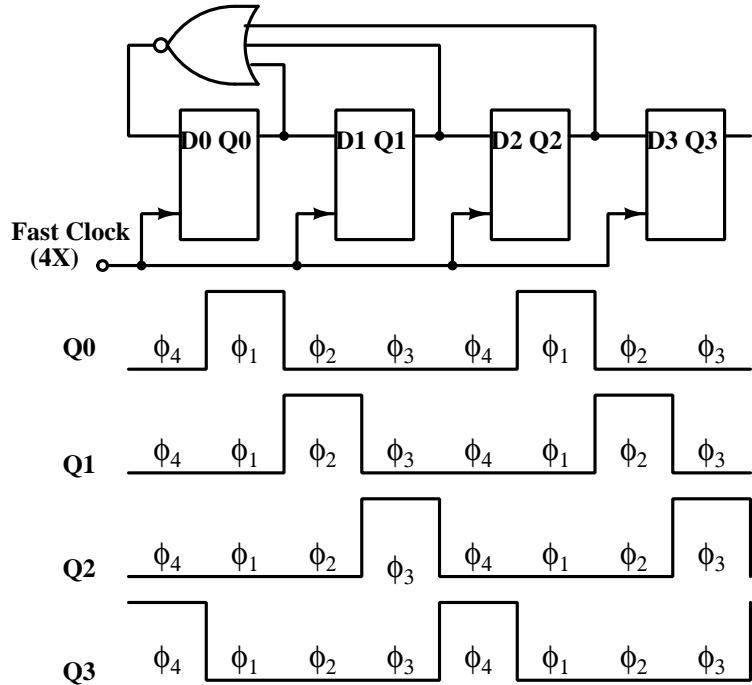


Figure 5.3: Generation of phase clocks for dynamic logic

If any of these is a 1, it inserts 0's. Thus, exactly one D flipflop among the four holds a 1 at any time, all the rest hold 0. Thus, each Q output is high during a single phase of a 4 phase cycle.

5.2 Four Phase Dynamic Logic

As discussed above, We use different phases for pre-charge, evaluation and for holding a valid output.

For the circuit shown in Fig. 5.4, we have a 4 phase clock. Ck_{mn} is a clock which is high during the m and n phases of the clock. Similarly, \overline{Ck}_{mn} is a signal which is low during m and n phases of the clock. Combined clock signals of the type Ck_{mn} can be generated easily using simple logic.

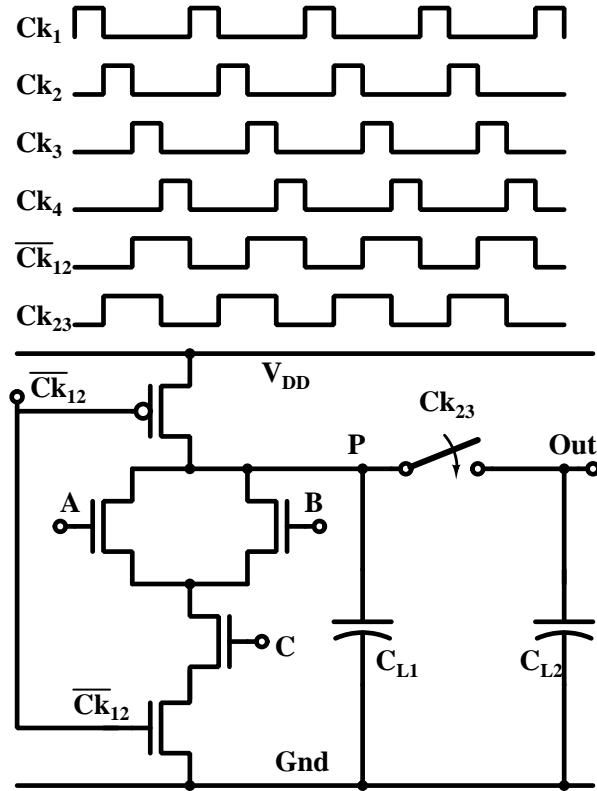


Figure 5.4: CMOS 4 phase dynamic logic

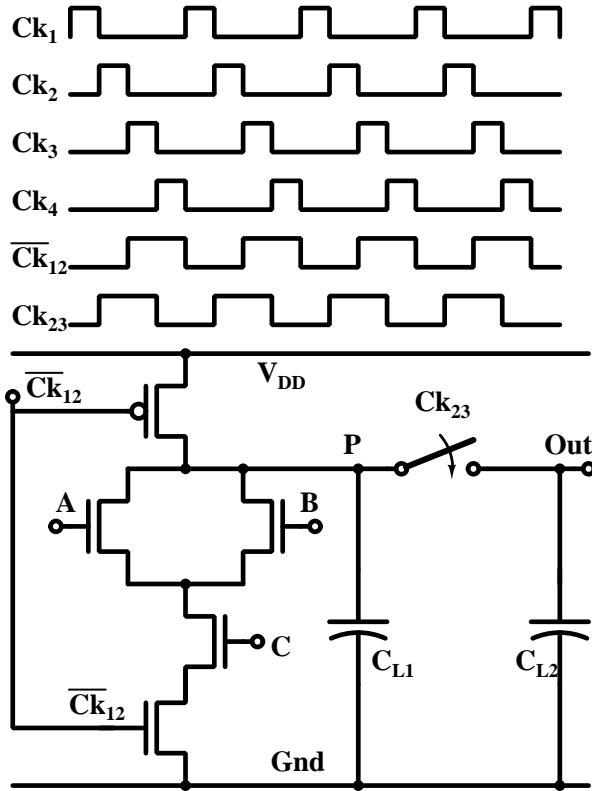
Now, for the gate shown in the figure, In phase 1, $\overline{Ck}_{12} = 0, Ck_{23} = 0$. So the pMOS is on and the bottom (clocked) nMOS is off. The output is disconnected from transistors. Node P pre-charges to ‘1’ while the output holds its old value.

In phase 2, $\overline{Ck}_{12} = 0, Ck_{23} = 1$. So the pMOS is on, bottom (clocked) nMOS is off and the output capacitor is in parallel with the capacitor on node P. As a result, node P, as well as the output node pre-charge to 1.

In phase 3, $\overline{Ck}_{12} = 1, Ck_{23} = 1$. So the pre-charge pMOS is off and the bottom (clocked) nMOS is on. Capacitors at node P and at the output are still in parallel.

If $(A + B) \cdot C = 1$, both capacitors will discharge to ground through the signal transistors and the bottom (clocked) nMOS. Otherwise the output will remain at 1. Thus the gate evaluates in phase 3 and acquires the correct output value at the end of this phase.

In phase 4, $\overline{Ck}_{12} = 1, Ck_{23} = 0$. The output is isolated from transistors and will hold its



- Ck_{mn} is defined as a clock signal which is ‘high’ during phase m and phase n of the clock.
- Similarly, \overline{Ck}_{mn} is a clock signal which is ‘low’ during phase m and phase n of the clock.

Phase → 1 2 3 4

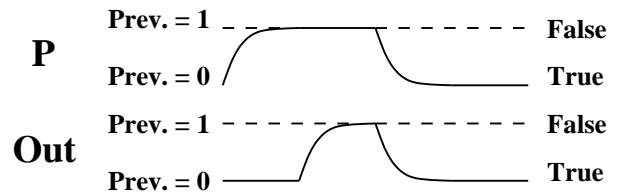


Figure 5.5: Operation of a type 3 gate

valid value. In phase 4 as well as in phase 1, the output is isolated from the driver and retains its valid value. However, Notice that node P no more holds a valid value in phase 1, since it pre-charges to ‘1’ in this phase.

This is called a type 3 gate, and needs the inputs to be valid and stable in phase 3. By changing the clocks to \overline{Ck}_{23} and Ck_{34} , we shall get a type 4 gate which evaluates in phase 4 and whose output remains valid in phases 1 and 2. Similarly, by cyclic permutation of clock signals, we can get type 1 and type 2 gates. A type 3 gate evaluates in phase 3 and is valid in phases 4 and 1. Similarly, we can have type 4, type 1 and type 2 gates. The output of a type 3 gate is correct and stable in phases 4 and 1. Consequently, its output can be used by type 4 and type 1 gates without any malfunction.

Each logic gate type in 4-phase dynamic logic design, needs its inputs to be valid and stable during one phase and provides an output which is valid and stable for two clock phases. A type 3 gate can drive a type 4 or a type 1 gate. Similarly, type 4 will drive types 1 and 2; type 1 will drive types 2 and 3; and type 2 will drive types 3 and 4. We can use a 2 phase clock if we stick to type 1 and type 3 gates (or type 2 and type 4 gates) as these can drive

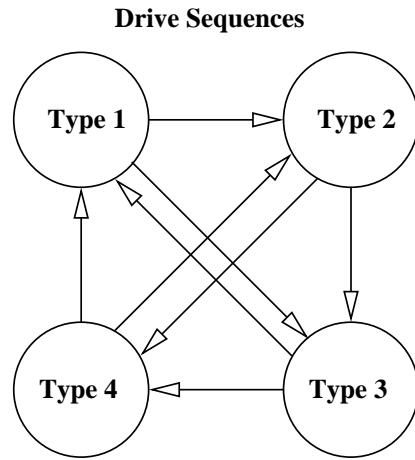


Figure 5.6: CMOS 4 phase dynamic logic drive constraints

each other.

5.3 Domino Logic

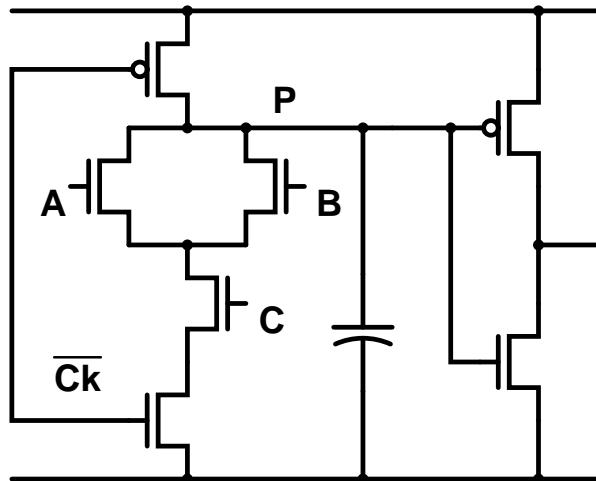


Figure 5.7: CMOS domino logic

Another way to eliminate the problem with cascading logic stages is to use a static inverter after the CMOS dynamic gate. Recall that cascading of dynamic CMOS stage causes problems because the output is pre-charged to V_{DD} . If the final value of a stage is meant to

be zero, the next stage nMOS to which this output is connected erroneously sees a one till the pre-charged output is brought down to zero. During this time, it ends up discharging its own pre-charged output, which it was not supposed to do.

If an inverter is added, the output is held ‘low’ before logic evaluation. Now, if the final output of this gate is ‘0’, there is no problem anyway. If the final output was supposed be ‘1’, the next stage is erroneously held at zero for some time. However, this does not result in a false evaluation by the next stage. The only effect it can have is that the next stage starts its evaluation a little later.

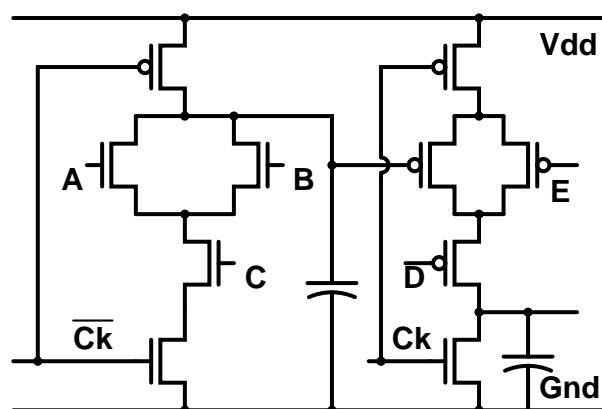
Domino logic is fast, because the pre-charge cycle is common to all stages. Once pre-charge is done, each stage evaluates one after the other (hence the name - domino logic). Thus for n stage logic, there is only one cycle of pre-charge and n cycles of evaluation, rather than n cycles of pre-charge and n of evaluation.

However, the addition of an inverter means that the logic is non-inverting. Therefore, it cannot be used to implement any arbitrary logic function. In synchronous digital circuits, combinational logic alternates with clocked latches. If inversion is required, we insert a latch at that place and take the \bar{Q} output of the latch to the next group of domino logic.

5.4 Zipper logic

Instead of using an inverter, we can alternate n and p evaluation stages. Now each stage is inverting and any logic can be implemented using such stages. The constraint now is that an n stage can only drive a p stage, and a p stage can only drive an n stage. This kind of logic is called *zipper* logic, in analogy with a zipper where links from alternate sides are joined one after the other. The n stage is pre-charged ‘high’. If the output was supposed to be ‘high’, it is anyway correct and cannot cause a malfunction. If the eventual output was supposed to be ‘low’, this transiently wrong ‘high’ output will not harm the next stage whose evaluation transistors are p type. A ‘high’ output will keep the transistors off for some time and no charge will be leaked away from capacitors. When the stage reaches its correct output, the next stage will evaluate its output correctly.

Similarly, the p stage will be pre-discharged to ‘low’. If the output was supposed to be ‘low’, it is anyway correct and cannot cause a malfunction. If the eventual output was supposed to be ‘high’, this transiently wrong ‘low’ output will not harm the next stage whose evaluation transistors are n type. A ‘low’ output will keep the nMOS evaluation transistors of the next stage off for some time and no charge will be leaked away from capacitors. When the stage reaches its correct output, the next stage will evaluate its output correctly.



**A, B, C must be from p stages.
D and E must be from n stages.**

Figure 5.8: Zipper logic

Chapter 6

Circuits with Mixed Design Styles

Some commonly used circuits use a mixture of styles. The availability of transistor switches which can pass signals in either direction provides a unique flexibility to CMOS technology based designs. Mixing traditional CMOS logic gates with MOS transistor switches in this way leads to efficient implementation of many useful circuits. Examples of this class of circuits are transmission gates, tri-stateable inverters, multiplexers, tiny-xor circuits and D latches and flipflops.

6.1 Transmission gates

A single transistor used as a switch has limitations in passing rail to rail signals. An nMOS transistor is inefficient for passing a ‘1’, while a pMOS transistor acting as a switch is inefficient for passing a ‘0’. However, if we use an nMOS and a pMOS in parallel, they provide a good transmission gate with a reasonable on resistance over the entire voltage range. Both the n

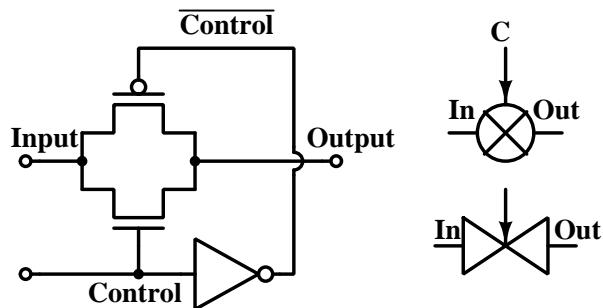


Figure 6.1: a transmission gate

and p channel transistor are on when the control input is high and both are off when it is low. The inverter is a CMOS inverter.

Notice that the input and output are interchangeable.

This circuit is represented by either of the symbols shown on the right.

6.2 Tri-stateable inverter

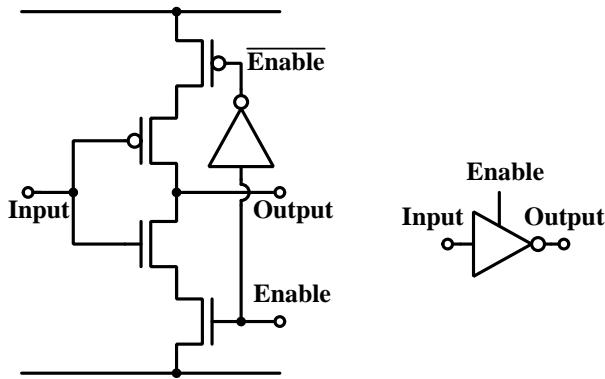


Figure 6.2: a tri-stateable inverter

We often need an inverter which can be put into a high impedance state using a control input. The high impedance state is the third state of the inverter, apart from producing a '0' or a '1' at the output. Such inverters are called tri-stateable inverters. When $\text{Enable} = 0$, both pull up and pull down are off. The inverter then presents a high impedance output.

When $\text{Enable} = 1$, power is supplied to the inner pMOS and nMOS transistors and the circuit acts like a CMOS inverter.

The symbol shown on the right is used to represent a tri-stateable inverter.

6.3 Multiplexers

By shorting the outputs of two tri-stateable inverters of which only one is active, we can implement an inverting mux. The enable signals of the two tri-stateable inverters are complementary. So only one of these is on at any given time. Depending on the state of the enable signal, the input to the enabled inverter appears at the output in an inverted form.

The two way multiplexer can be generalized to an n way multiplexer by adding a decoder, which generates Sel and \overline{Sel} for each input, which in turn control a tri-stateable inverter each. The figure below shows a 4 way multiplexer. The Decoder provides outputs such that only

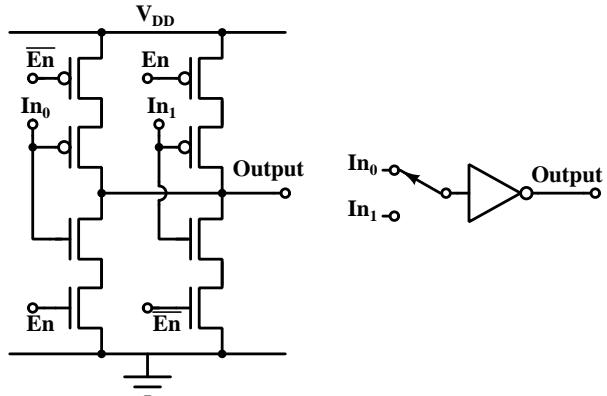


Figure 6.3: a two way mux

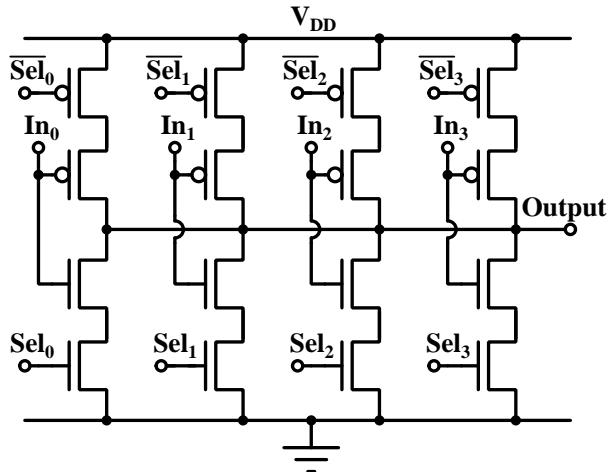


Figure 6.4: 4 input multiplexer

one Sel signal is high and the corresponding \bar{Sel} signal is low. So the selected data appears inverted at the output and all other tri-stateable inverters are disabled.

6.4 Tiny XOR and XNOR

A compact XOR circuit can be made using pass gates.

When $A = 0$, the pass gate on the right is on and passes B to the output. Also, the inverter like circuit in the middle has 0 applied on top and a 1 applied at the bottom. Both transistors act as source followers and thus, also drive the output to B .

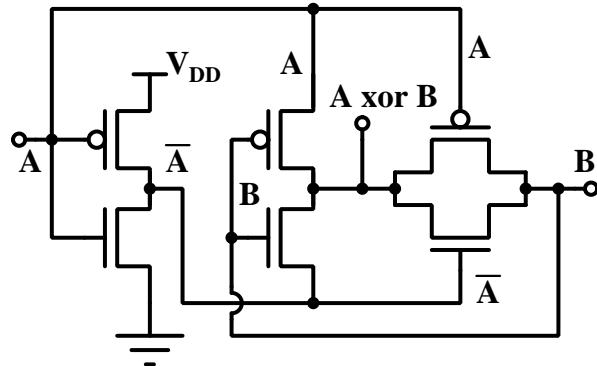


Figure 6.5: “Tiny” XOR circuit using a transmission gate

When $A = 1$, the pass gate is open and the middle circuit has 1 at the top and 0 at the bottom. So it acts like a regular inverter and provides \overline{B} at the output.

Then, by Shannon’s Boolean expansion theorem, this constitutes the XOR function: $A \cdot \overline{B} + \overline{A} \cdot B$.

A compact XNOR circuit can be made similarly.

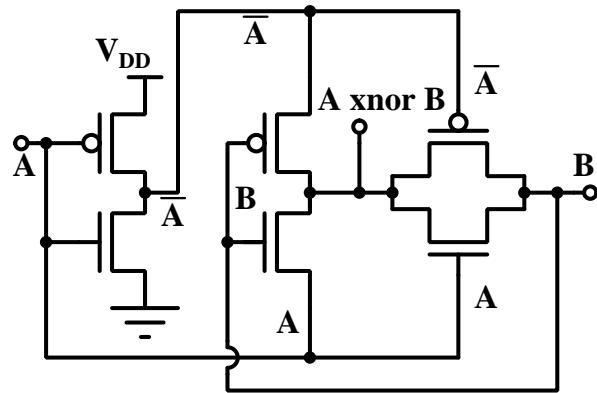


Figure 6.6: “Tiny” XNOR circuit using a transmission gate

When $A = 1$, the pass gate on the right is on and passes B to the output. Also, the inverter like circuit in the middle has 0 applied on top and a 1 applied at the bottom. Both transistors act as source followers and thus, also drive the output to B.

When $A = 0$, the pass gate is open and the middle inverter like circuit has 1 at the top and 0 at the bottom. So it acts like a regular inverter and provides \bar{B} at the output.

By Shannon's Boolean expansion theorem, this constitutes the XNOR function: $A \cdot B + \bar{A} \cdot \bar{B}$

6.5 D latch and flipflop

Two inverters and two pass gates can be combined to form a transparent D latch. This latch is level sensitive.

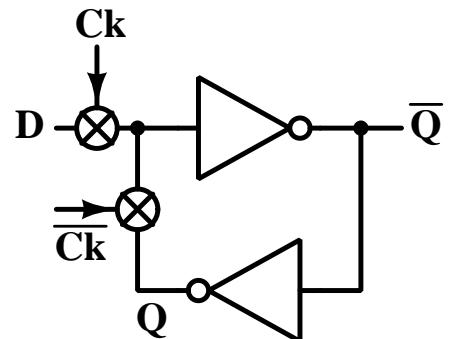


Figure 6.7: A transparent D latch using transmission gates

When the clock signal is high, the input is connected to the two inverters, and the feedback switch is open. Thus, D is buffered through to Q. This latch is called transparent in this stage, because a change in the value of D will result in a corresponding change in the Q outputs.

When the clock goes low, the inverters are disconnected from the D input and the feedback switch is closed. this results in the value of D (which existed when the clock went from high to low) being capture by the latch. In this state, changes in D do not affect the Q outputs.

The two pass gates and the upper inverter in the transparent D latch form a two way multiplexer between D and Q.

Two transparent latches with complementary clocks can be connected in a master slave fashion to form an edge sensitive D flip flop. When the clock is low, the first latch is transparent but the second latch hold its previous value. When the clock transitions to high, the first latch captures the value of D at this instant. The second latch becomes transparent.

Outputs don't follow D in either state of the clock. Q reflects the value of D at the most recent positive transition on Ck.

We sometimes need an edge sensitive D flipflop which can be reset asynchronously. Q

It is possible to replace the two pass gates and the connected inverter as described above by an inverting two way mux. The feedback inverter can continue to be a traditional CMOS inverter.

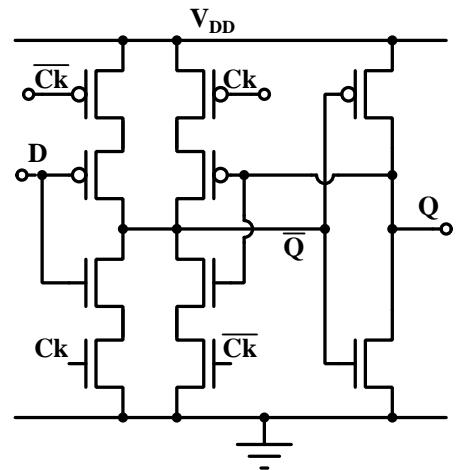


Figure 6.8: A transparent D latch using a two way multiplexer

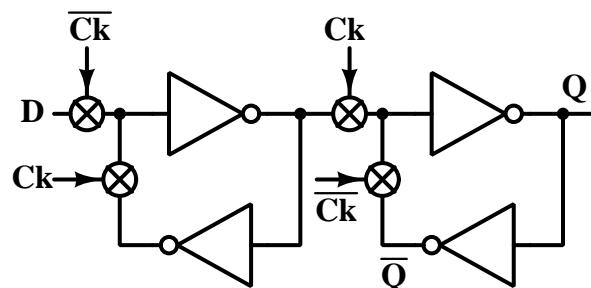


Figure 6.9: Edge sensitive master slave D flipflop

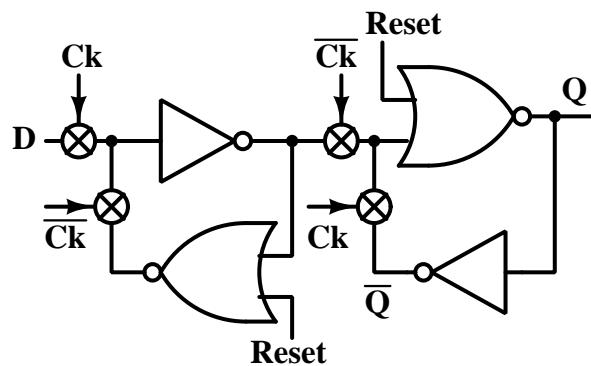


Figure 6.10: Edge sensitive D flipflop with asynchronous reset

should go to 0 as soon as reset is applied. To do this, the inverter supplying Q is changed to a NOR, with Reset as its other input. However, this is not sufficient. The flipflop should remain reset even when the reset signal is removed and Ck is at 0. For this, the lower inverter in the first latch should also be changed to a NOR gate, with reset as its other input.

Use of asynchronous reset is not recommended due to testability concerns.

If we need both set and reset for an edge sensitive D flipflop, we should change the other two inverters also to NOR. Q should go to 1 as soon as Set is applied. As is common in

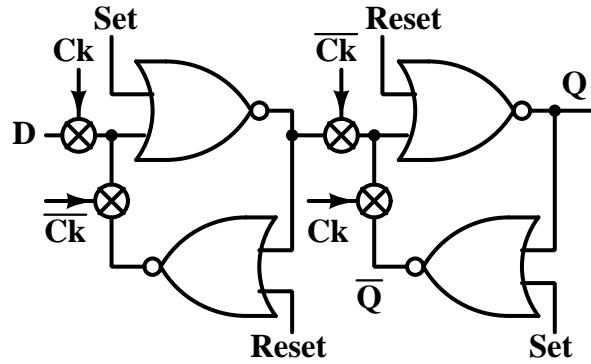


Figure 6.11: Edge sensitive D flipflop with asynchronous set and reset

all asynchronous set and reset controls, the two of these should never be applied simultaneously. so Reset should be 0 when Set goes to 1.

When Ck = 1, a 1 on set (with 0 on reset) will force Q to 1.

When Ck = 0, a 1 on set will force the first latch output to 0. This will be inverted by the NOR in the transparent second latch to produce a 1 at the Q output.

However, the use of asynchronous set or reset is not recommended due to testability concerns.

Chapter 7

Semi-Custom Design

In the previous chapters, we have looked at design techniques used when we have to design all parts of a VLSI circuit from scratch. This includes the choice of logic styles, adjustment of transistor geometries, their layout etc. to meet a given set of specifications. This is called full custom design. While this permits an optimal tradeoff between power, speed and complexity, the whole process is long and laborious. The cost of developing a custom designed VLSI circuit is very high and can be justified only by those designs which sell in very large quantities.

Some applications can afford to compromise on speed, power and complexity in order to achieve a quick design and fabrication time and lower costs. These use a design style known as semi-custom design. In semi-custom design, part of the design and fabrication is already done for us. We take this pre-fabricated template and customize it to perform the functions that our VLSI needs to perform. This approach has several advantages. The prefabricated part can be processed and kept ready for customization. Then the time to take an application to the market is only the processing which is necessary after customization.

Also, different applications can use the same pre-fabricated template. While each application may have a relatively small market, the template will be used in very large numbers, making it economically feasible. Obviously, the pre-fabricated template is not customized for final requirements of a particular final circuit. So it will not be an optimal design for a given application. However, most applications do not require absolutely the best performance. In such cases, the time to market and cost can be drastically reduced by using semi-custom design.

Clearly, if the customization step occurs later in the design and fabrication cycle, the time to market will be shorter. However, the overall implementation may have a lower performance, since a higher fraction of the design and fabrication cycle is non-specific to the actual design.

An example of semi-custom design is Field Programmable Gate Array (FPGA) based design. In this case, customization is done *after* the product has actually been delivered to the end user. (That is what the term Field Programmable refers to). The actual VLSI template which can cater to this late customization is quite complex. Therefore the area is much higher and the speed much lower compared to what could have been achieved by a custom designed circuit. For example, system clock speeds possible with FPGAs are of the order of hundreds of MHz, whereas custom circuits can go to clock speeds of about 4 GHz. To implement a given function, the silicon area consumed by an FPGA may be an order of magnitude higher than the area consumed by custom designed circuit performing the same function. However, since many applications can use the same FPGA, the FPGA chip is produced in very large quantities, which provides economies of scale. Thus, for designs which are not produced in very large quantities, FPGA based design may be the most cost effective solution inspite of the overhead in complexity, power consumption and delay.

7.1 Procedure for Semi-Custom Design

There are two components of Semi-custom design technique.

1. Customization techniques which will be used for adapting the “fabric” to implement the final application on the given template.
2. The design of the basic template or the “fabric” which can be customized as late as possible and which can be used by a large variety of applications.

In the technologies used for fabricating VLSI circuits, transistors are defined much earlier than the interconnects. Interconnects are defined towards the end of the fabrication procedure. Therefore customization of a pre-fabricated template is commonly done by changing the interconnects.

7.2 Customization of interconnects

Customization of interconnects is done by placing programmable interconnect devices at appropriate points in the pre-fabricated template. These include fuses, anti-fuses and transistor based interconnect.

Fuses Customization of interconnects can be done through programmable removal of an existing connection. These work like fuses in electrical circuits. The connection to be customized is connected in the template through a narrow neck like structure. The narrow part of the interconnect is capable of passing normal operating currents of the circuit. However, during customization, one may pass a pulse of heavy current through this, which ‘blows’ the fuse and disconnects the wires leading up to the fuse.

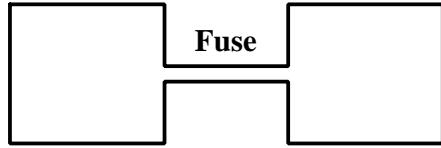


Figure 7.1: fuse structure

Anti-fuses In these structures, there is no connection between the programming nodes in the un-programmed state of the interconnect. The current path is interrupted through a thin layer of insulator. By applying a high voltage pulse, this insulator can be broken

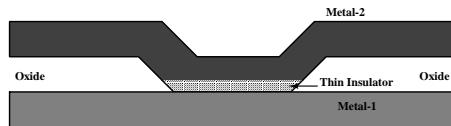


Figure 7.2: An anti-fuse. The insulator could be amorphous silicon.

down and a short created across it. This action is opposite to that of a fuse, so such structures are called anti-fuses.

Transistor based connection In this approach, a transistor is placed in the current path as a switch. When this transistor is on, there is a connection. When it is off, there is no connection. The state of the transistor is decided by a memory, which can be pre-loaded at power on.

7.3 Implementation of Configurable Template

7.3.1 Programmable Logic Arrays

Logic functions can be expressed in a generic sum of products form.

We would like a regular circuit which can be re-configured to implement any sum of products. One way is to use a customisable array which produces different product terms and another customisable array which sums the products so produced. CMOS logic is not convenient for implementing this architecture. This is because simultaneous configuration of the p channel pull up network as well as that of the n channel pull down network is inconvenient. Pseudo NMOS gates are more suitable for re-configuration in this case, as the pull up is just a single grounded gate PMOS whose geometry and interconnection remains the same for all

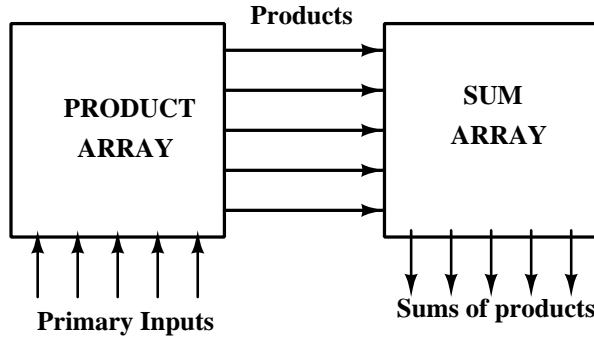


Figure 7.3: Architecture of a Programmable Logic Array

logic.

But Pseudo NMOS circuits are ratioed. Since the geometry of transistors is defined early during fabrication, we would like an architecture where transistor geometry does not depend on the logic being implemented.

Implementing sums is not a problem, since this will be done through a NOR like configura-

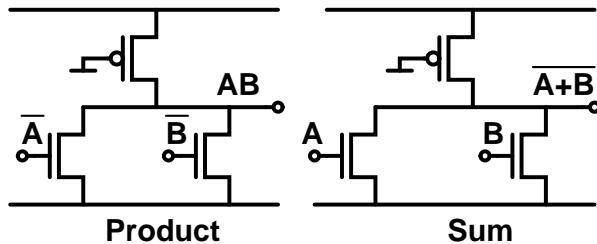


Figure 7.4: Product and sum implementation in a PLA

tion. In NOR configuration, each term contributes a transistor in parallel, whose geometry is the same as that of the reference inverter on which the design is based. Thus, sum of logic terms can be implemented with transistors whose geometry is fixed. However, implementing products presents a problem, since this requires NAND configuration, where the geometry of series connected pull down devices depends on the number of devices connected in series and hence, on the specific logic being implemented.

How do we implement the product function then?

We can use the expression : $A \cdot B = \overline{\overline{A} + \overline{B}}$.

Now the product of A and B can be implemented as the NOR of \overline{A} and \overline{B} , which does not use series connected transistors. By adding inverters at the input and output as required, we can

implement products or sums using only NOR type of circuits, which use constant geometry NMOS transistors in parallel.

Product Array

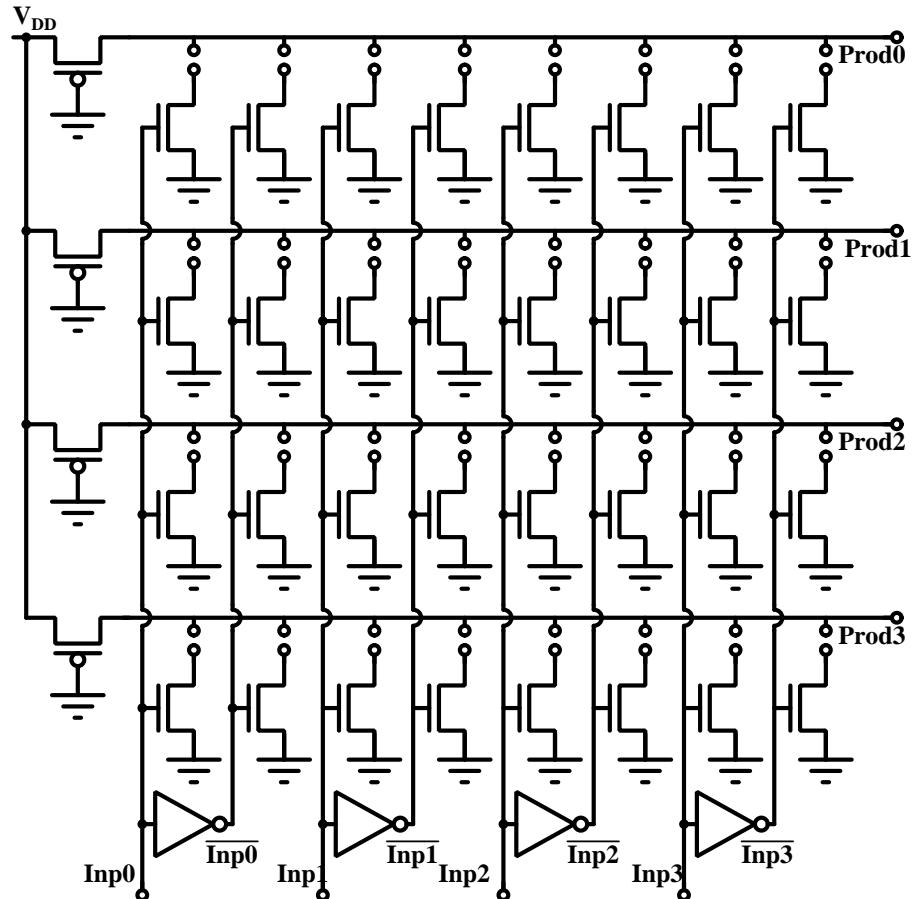


Figure 7.5: A programmable product array

Suppose we want to generate p distinct products of n inputs. Since for product implementation, we need complemented inputs, we connect a static CMOS inverter to each input. Now we have access to complemented as well as uncomplemented inputs. To design the product array, we place nMOS transistors in a matrix of $2n$ columns and p rows where n is the number of inputs and p is the number of distinct products that we want to generate. Since the complement of each of the inputs is produced using inverters, n signals and n complements drive the gates of nMOS transistors in $2n$ columns. Each row generates a

distinct product. It is pulled up by a grounded gate pMOS transistor. So this matrix can produce p distinct products. By connecting links selectively at drains of nMOS transistors in any row, chosen transistors can be included in parallel in the NOR configuration. Connecting a transistor includes the complement of the input driving its gate in the product term.

Each column is driven by an input or its complement. Thus each of these nMOS transistors has its source grounded and its gate connected to an input or its complement. Programming involves either connecting its drain to the row which forms the output or leaving it unconnected. If the drain is connected to the output, the transistor comes in parallel with other connected transistors, as required by the pseudo NMOS configuration.

If we want to include a signal (or its complement) in the product term, we connect the drain of the transistor driven by the complemented input (or uncomplemented input) to the row. If we do not want either the signal or its complement in the product, we simply leave both nMOS transistors unconnected.

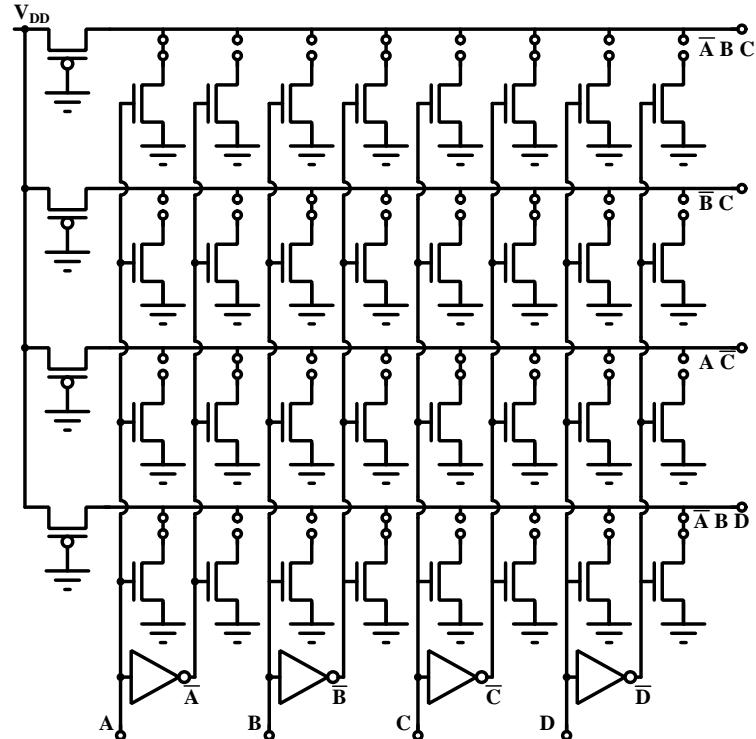


Figure 7.6: Example for generation of different products from inputs

In the example shown in Fig.7.6, we have generated products $\bar{A} \cdot B \cdot C$, $\bar{B} \cdot C$, $A \cdot \bar{C}$ and $\bar{A} \cdot B \cdot D$. To generate $\bar{A} \cdot B \cdot C$, links from drains of transistors driven by A , \bar{B} and \bar{C} are connected to the top product row. For $\bar{B} \cdot C$, drains of transistors driven by B and \bar{C} are

connected to the output line in the second row. Other products are generated similarly.

Sum Array

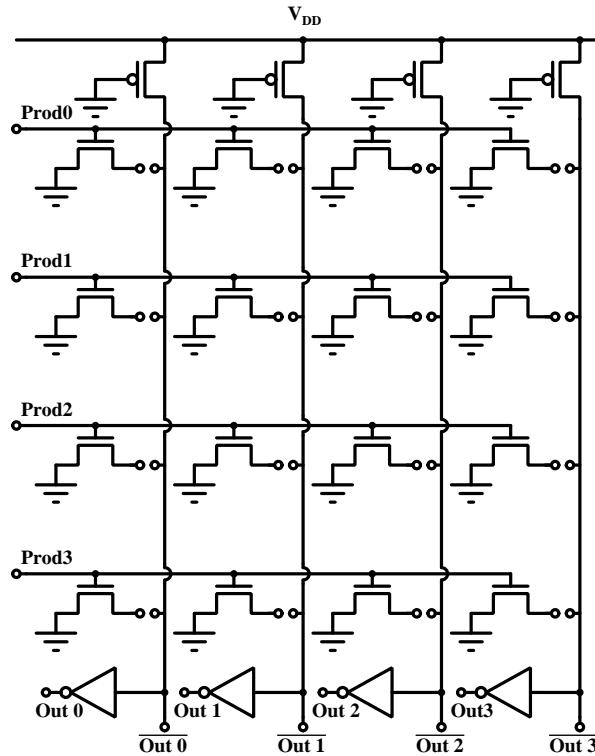


Figure 7.7: Sum array for generating sums of products

We can form the sum matrix similarly. Each of the products generated by the product array drives gates of a nMOS transistors in a row, arranged in a $p \times s$ matrix, where s is the number of different sums of products that we need to generate. The circuit in Fig.7.7 can produce sums of selected product terms. By connecting links at drains of nMOS transistors in a particular column, chosen products can be included in a sum output. Several columns can be used to generate different sums of products. Outputs are NORs of products. Inverters convert these to ORs of products.

Implementation of Finite State Machines

A finite state machine has the following components:

1. Storage elements to store the current state,

2. Random logic to compute the next state as a function of current state and current inputs, and
3. Random logic to compute the current output as a function of current state and optionally, the current inputs.

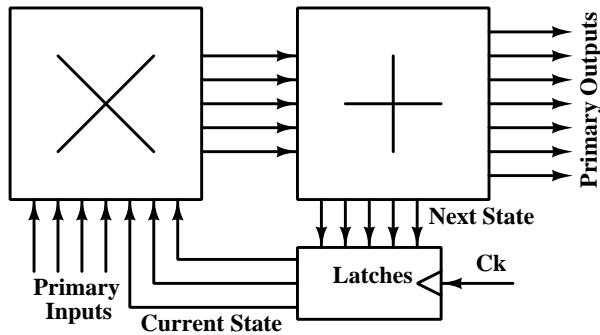


Figure 7.8: A finite state machine implemented with a PLA

By adding latches at the output of the PLA, we can provide for all of these. Output from latches is fed back to the product array. The PLA now computes both the next state and current outputs with primary inputs and current state as its inputs.

7.3.2 Sea of Gates

The programmable logic array uses pseudo nMOS design style as its base. This involves static power consumption. The stray capacitance of a programmable plane is high – which makes PLAs rather slow.

How can we use CMOS style gates in a semi-custom design? The “Sea of Gates” template provides the capability to use CMOS style gates in a semi-custom design. In CMOS logic, each input goes to an nMOS as well as a pMOS. In this style of design, all transistors are pre-placed in a pattern which is optimum for implementing CMOS gates because there are pairs of nMOS and PMOS transistors driven by the same input at their gates. Once an array of such patterns is placed in the chip, interconnects will determine what kind of logic will be implemented. Both n and p channel transistors appear to be in series here! How do we construct regular CMOS logic gates using these?

Actually, by wiring the apparently series connected devices, we can convert them to series or parallel as required. The example in Fig.7.10 shows a NAND gate, whose output is fed to an inverter to form the AND function. The structure on the right forms a NOR gate.

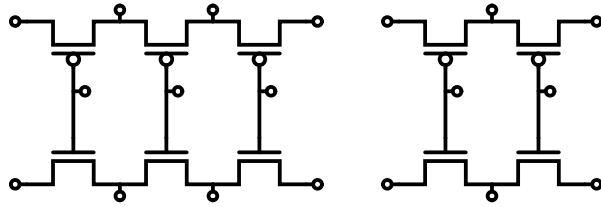


Figure 7.9: Pattern of transistors in a sea of gates template

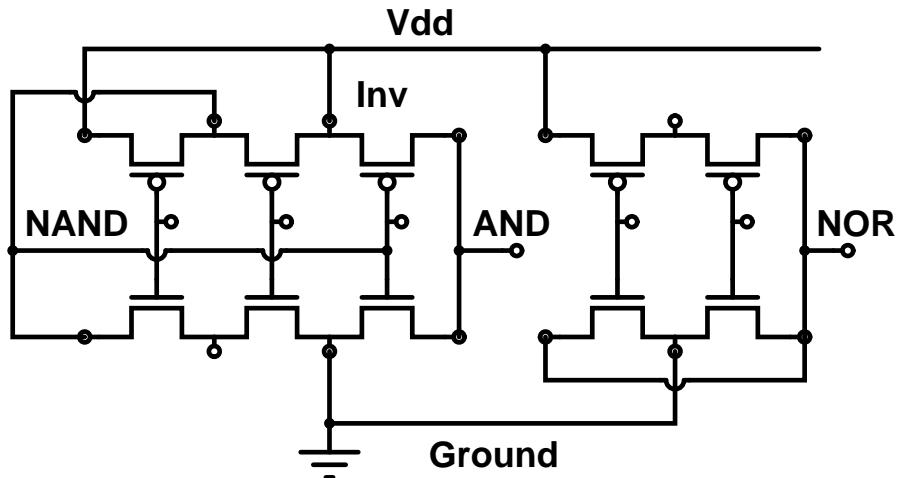


Figure 7.10: Implementing a NAND, Inverter and NOR gates in sea of gates

The sea of gates template makes use of the fact that all inputs go to an nMOS as well as to a pMOS in CMOS style gates. What about structures which don't follow this rule? For example, in a transmission gate, the nMOS and pMOS transistors are driven by complementary signals.

Transmission gates are often used in pairs with one or the other being on. (This is so because Inputs should not be left floating in static CMOS design). A pair of transmission gates can be implemented as shown in Fig.7.11, coupling diagonally opposite transistors in a 2-pair. As an application of this, a transparent D latch uses transmission gates with complementary control inputs. It can be implemented as shown in Fig.7.12. Two such latches can be connected in master-slave configuration to form an edge sensitive D flip-flop.

Sea of gates based designs provide better performance compared to PLA based designs. However, these cannot be field programmed and the interconnect mask has to be customized and used during chip manufacture.

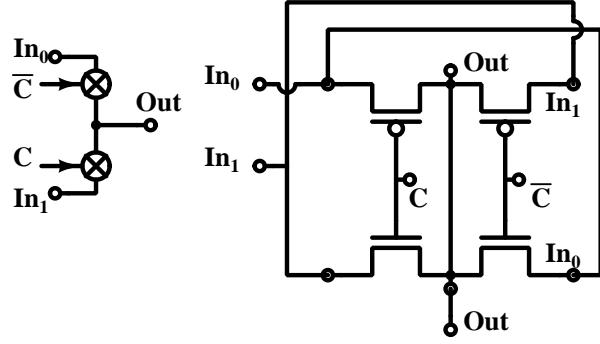


Figure 7.11: A pair of transmission gates with complementary control inputs

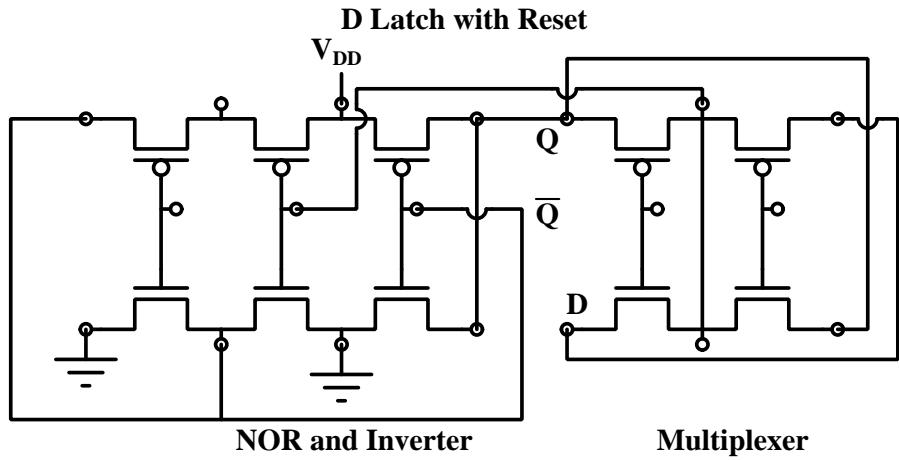


Figure 7.12: A transparent D latch implemented with “Sea of Gates”

7.4 Interconnect Channels in Semi-Custom Logic

Apart from programming the transistor matrix, we need to provide programmable interconnect between different sub-units. therefore in a semi-custom integrated circuit, pre-fabricated interconnect channels alternate with transistor matrices.

The exact shape and composition of these interconnect channels varies from design to design. Typically, these include facilities for local as well as global interconnects. The size of these blocks is optimized to provide a sufficient interconnect capacity without wasting too much area.

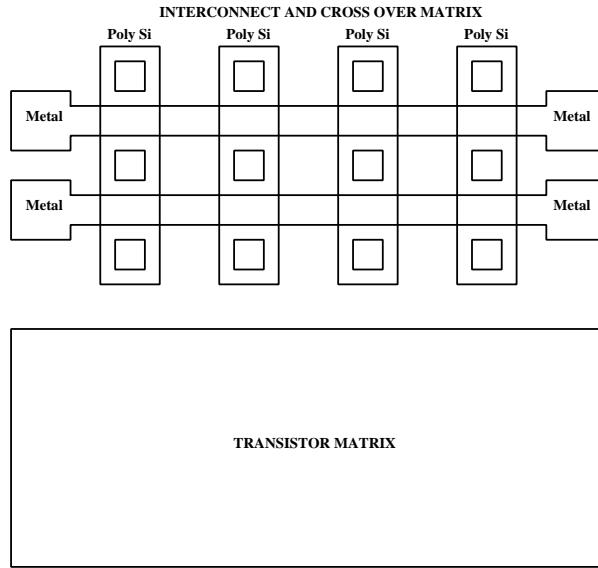


Figure 7.13: An example interconnect channel for semi-custom logic

7.5 Using Memories as Logic

A logic function can be represented by a truth table. This can be stored in memory. Inputs to the logic function act as address lines. Precomputed value of the logic function are stored at the address represented by its input values. For any input combination, the address represented by these is looked up and the stored value presented as the output.

For example, any logic function of 5 inputs can be implemented by a 32×1 bit memory. If the same 32 bit memory is organized as 16×2 , it can store two logic functions of up to 4 inputs.

7.6 Field Prgrammable Gate Arrays

Different types of semi-custom design chips have been introduced with various names. Here is a list:

PLAs: These were the first semi-custom devices to be introduced in the market by Philips in the early 1970's. These are used to implement generic sums of products and have a programmable AND plane as well as a programmable OR plane.

Because both AND and OR arrays are programmable in PLAs, these are rather slow.

PALs: A modification of PLAs in which the product array is programmable, but the OR

plane is fixed were introduced as Programmable Array Logic or PAL. To cover for the lack of flexibility due to the OR array being fixed, these were introduced in different size combinations and multiple devices were used for different functions.

CPLDs As technology progressed, it was possible to put multiple devices on the same chip. Combinations of PALs and PLAs were introduced with programmable interconnect as complex programmable logic devices or CPLDs. Altera was one of the first companies to introduce CPLDs commercially. CPLDs were a huge success and multiple companies introduced CPLDs of different architectures.

Sea of Gates In parallel with field programmable devices, mask programmable devices were manufactured. In these, one (or more) levels of metallization could be customized at the manufacturing site, over a “sea” of pre-fabricated devices. These provided circuits with better performance, but with less flexibility as programming could not be done at the user site.

FPGAs Field Programmable Gate Arrays borrow features from sea of gates as well as CPLDs. Instead of a “sea” of transistors, these have a “sea” or a repetitive array of combinations of logic elements and memories.

In addition, these include a programming infrastructure for interconnects like CPLDs.

On the periphery of these chips, special Input Output devices are fabricated which help in establishing fast interconnection with the external world.

A field Programmable array allows the logic functions as well as the interconnect to be programmed. Transistors driven by SRAM can be used to program interconnects. Apart from logic blocks and interconnect fabric, modern FPGA's contain other useful structures, such as

- Block RAM,
- Processor cores (Power PC on Xilinx Vertex family),
- Fast multipliers
- I-O Blocks

A typical board, used for implementing a variety of applications will have a micro-controller and a few programmable devices for providing dedicated functions and glue logic. This is configured once at power on, and then left alone to perform its functions. This is called “static re-configurability”.

But we may want to re-configure a structure while it is in operation! For example, one can imagine the design of digital filter, which adapts itself during operation itself depending on

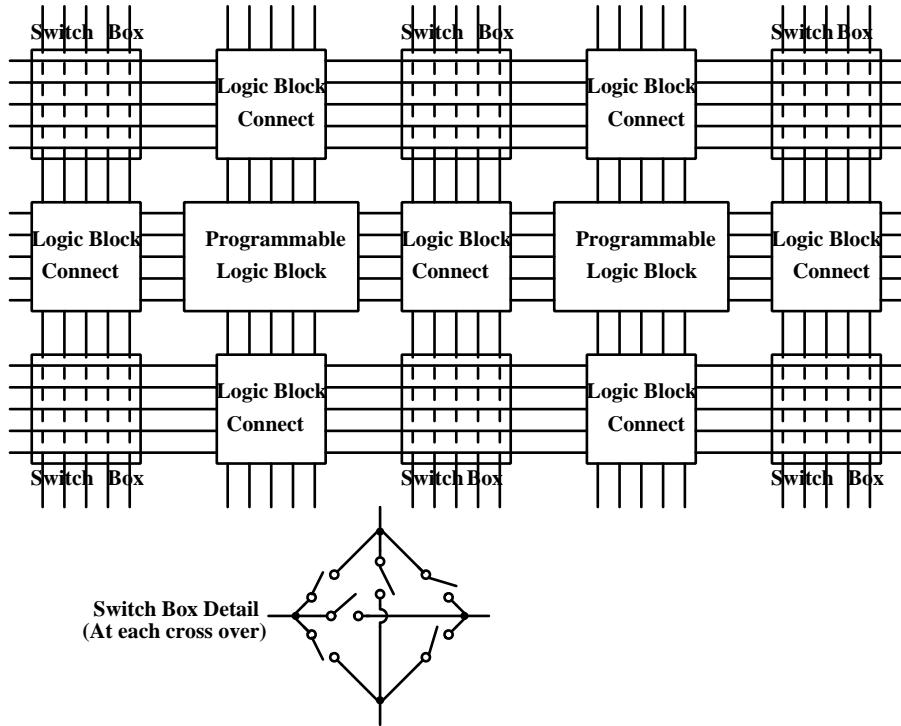


Figure 7.14: Alternating logic and interconnect boxes in an FPGA

inputs. This kind of re-configurability is called “dynamic re-configurability”.

Obviously, dynamic configurability requires that the dead time during re-configuration should be minimized. Thus, there is a trade-off between flexibility of re-configuration and the time taken to re-configure. In fine grain re-configuration, we can re-program every gate of the design. This is the case for current FPGA and CPLD devices. This gives very good design flexibility, but takes a long time to re-configure.

In coarse grain re-configuration, relatively large functional blocks are re-configured. For example, by re-connecting a collection of shifters and adders, we can generate any combination of multipliers and adders. The effort to re-configure is smaller, because we do not alter the inner design of shifters and adders. This is compatible with dynamic re-configuration. Using coarse grain re-configurability, it becomes possible to dynamically change a circuit, *during its operation*. This has obvious advantages in adaptive circuits.

We normally do not want to re-configure the whole circuit. Indeed the control signals for doing the re-configuration will be generated by a circuit which remains unchanged. Therefore

dynamic re-configurability requires circuits which can be partially re-programmed. Many FPGA are beginning to promise this feature.

Semi-custom Design

Dinesh Sharma

EE Department
IIT Bombay, Mumbai

September 19, 2020

Semi-custom Design

- In the previous sections, we have looked at design techniques used when we have to design all parts of a VLSI circuit from scratch.
- This includes the choice of logic styles, adjustment of transistor geometries, their layout etc. to meet a given set of specifications.
- This is called full custom design.
- While this permits an optimal trade-off between power, speed and complexity, the whole process is long and laborious.
- The cost of developing a custom designed VLSI circuit is very high and can be justified only by those designs which sell in very large quantities.

Semi-custom Design

- Some applications can afford to compromise on speed, power and complexity in order to achieve a quick design and fabrication time and lower costs.
- These use a design style known as semi-custom design.
- In semi-custom design, part of the design and fabrication is already done for us. We take this pre-fabricated template and customize it to perform the functions that our VLSI needs to perform.
- This approach has several advantages. The prefabricated part can be processed and kept ready for customization.
- Then the time to take an application to the market is defined only by the remaining processing which is necessary after customization.

Semi-custom Design

- Different applications can use the same pre-fabricated template.
- While each application may have a relatively small market, the template will be made in very large numbers, making it economically competitive.
- Obviously, the pre-fabricated template itself is not customized for a particular final circuit. So it will not be an optimal design for a given application.
- However, most applications do not require absolutely the best performance.
- In such cases, the time to market and cost can be drastically reduced by using semi-custom design.

Field Programmable Gate Arrays

- From a time to market point of view, it is preferable to have the customization done as late as possible.
- However, then a higher fraction of the design and fabrication cycle is non-specific to the actual design.
- In Field Programmable Gate Array (FPGA) based design, customization is done *after* the product has actually been delivered to the end user.
- That is what the term Field Programmable refers to.
- The actual VLSI template which can support this late customization is quite complex.
- The area of the chip is much higher and the speed much lower compared to what could have been achieved by a custom designed circuit.

Field Programmable Gate Arrays

- System clock speeds possible with FPGAs are of the order of hundreds of MHz, whereas custom circuits can go to clock speeds of about 4 GHz.
- To implement a given function, the silicon area consumed by an FPGA may be an order of magnitude higher than the area consumed by custom designed circuit performing the same function.
- However, since many applications can use the same FPGA, the FPGA chip is produced in very large quantities, which provides economies of scale.
- Thus, for designs which are not produced in very large quantities, FPGA based design can be the most cost effective solution, in spite of the overhead in complexity, power consumption and delay.

Semi-Custom Design Procedure

There are two components of Semi-custom design technique.

- ① Customization techniques which will be used for adapting the “fabric” to implement the final application on the given template.
- ② The design of the basic template or the “fabric” which can be customized as late as possible and which can be used by a large variety of applications.

During fabrication, transistors are defined much before the interconnects.

Therefore, to be able to customize the circuit as late as possible, customization of a pre-fabricated template is commonly done by changing the interconnects.

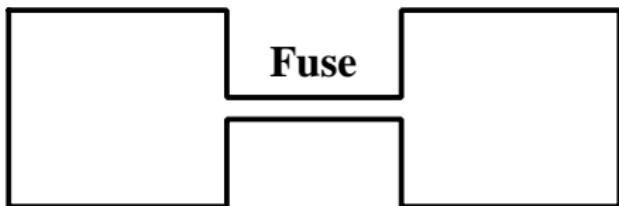
Customizing Interconnects

Customization of interconnects is done by placing programmable interconnect devices at appropriate points in the pre-fabricated template. These include:

- Fuses: Here the connection is a short by default and can be converted to an open by blowing a fuse.
- Anti-fuses: Here the connection is open by default, but can be converted to a short by breaking down an insulator.
- transistor based interconnects: Here a connection is made or broken using transistor switches. The state of the transistor is defined by a memory.

Use of Fuses to Customize Interconnect

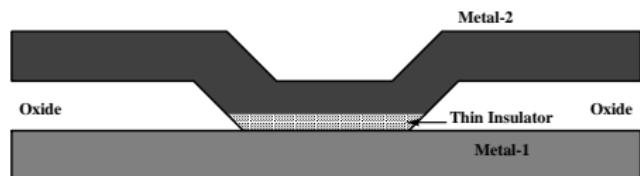
Customization of interconnects can be done through programmable removal of an existing connection. These work like fuses in electrical circuits.



- The connection to be customized is connected in the template through a narrow neck like structure.
- The narrow part of the interconnect is capable of passing normal operating currents of the circuit.
- However, during customization, one can pass a pulse of heavy current through this, which 'blows' the fuse and disconnects the wires leading up to the fuse.

Use of Anti-Fuses to Customize Interconnect

In these structures, there is no connection between the programming nodes in the un-programmed state of the interconnect. The current path is interrupted through a thin layer of insulator.



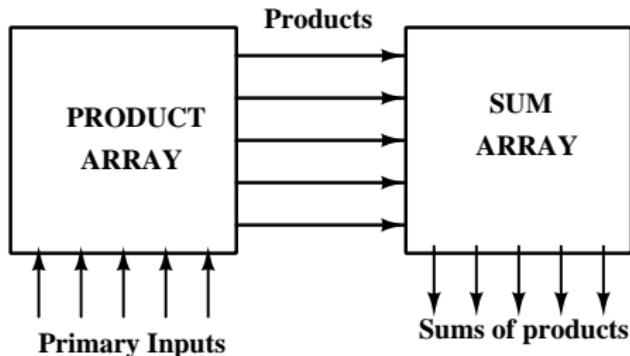
By applying a high voltage pulse, this insulator can be broken down and a short created across it.

This action is opposite to that of a fuse. Therefore such structures are called anti-fuses.

Re-configurable Logic with Programmable Logic Arrays

Logic functions can be expressed in a generic sum of products form.

- We need a circuit which can be re-configured to implement any logic expressed as a sum of products.
- One way is to use a customisable array which produces different product terms and another customisable array which sums the products so produced.



Use of Pseudo-nMOS for Programmable Logic Arrays

- We need an architecture where transistor geometries are not changed with logic and only interconnect needs to be programmed.
- CMOS logic is not convenient for implementing this architecture.
- This is because simultaneous configuration of the p channel pull up network and the n channel pull down network will be needed.
- Pseudo NMOS gates are more suitable in this case, because the pull up is just a single grounded gate pMOS whose geometry remains the same for all logic.

Use of Pseudo-nMOS for Programmable Logic Arrays

- Pseudo NMOS circuits are ratioed.
- Implementing sums is not a problem, since this is done with NOR type gates.
- Transistor geometry remains the same in NOR gates irrespective of how many transistors are put in parallel.
- However, implementing products presents a problem, since NAND gates connect nMOS transistors in series.
- The geometry of series connected pull down devices depends on the number of devices connected in series and hence, on the specific logic being implemented.

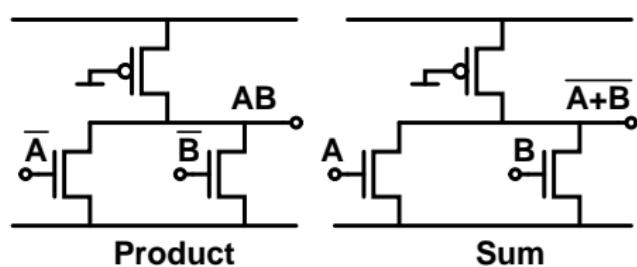
Use of Pseudo-nMOS for Programmable Logic Arrays

How to implement the product function without changing transistor geometry?

We can use the expression :

$$A \cdot B = \overline{\overline{A} + \overline{B}}$$

Now the product of A and B can be implemented as the NOR of \overline{A} and \overline{B} , which does not use series connected transistors.



By adding inverters at the input and output as required, we can implement products or sums using only NOR type of circuits, which use constant geometry NMOS transistors in parallel.

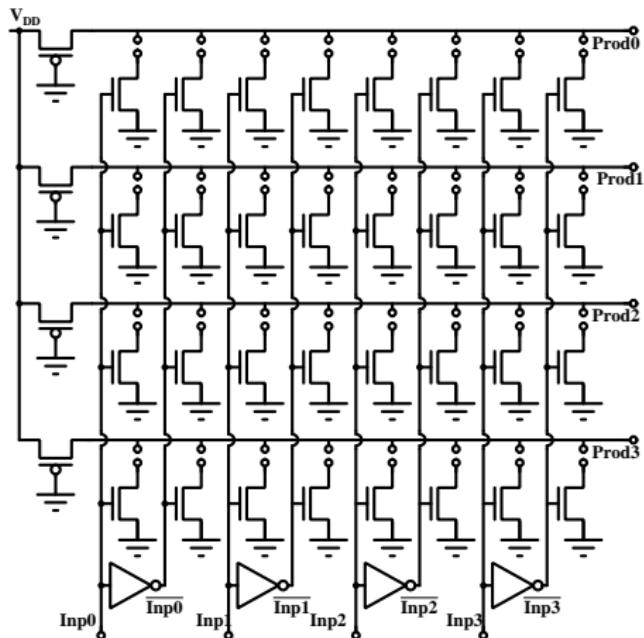
Programmable Logic: Product Array

Suppose we want to generate p distinct products of n inputs.

- We generate complements of all inputs.
- We place pull down nMOS transistors in a matrix of $2n$ columns and p rows.
- Each row is pulled up by a pMOS transistor with grounded gate.
- The row corresponds to the output of a NOR circuit which may have up to n pull down nMOS transistors in parallel.
- Each column is driven by an input or its complement. So there are $2n$ columns.

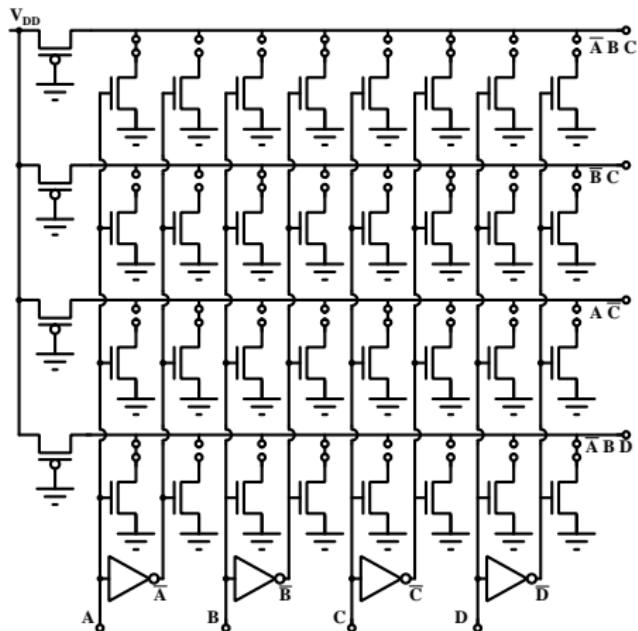
Programmable Logic: Product Array

- Each row generates a distinct product.
- By connecting links selectively at drains of nMOS transistors in any row, chosen transistors can be included in the NOR configuration.
- Connecting a transistor includes the complement of the input driving its gate in the product term.



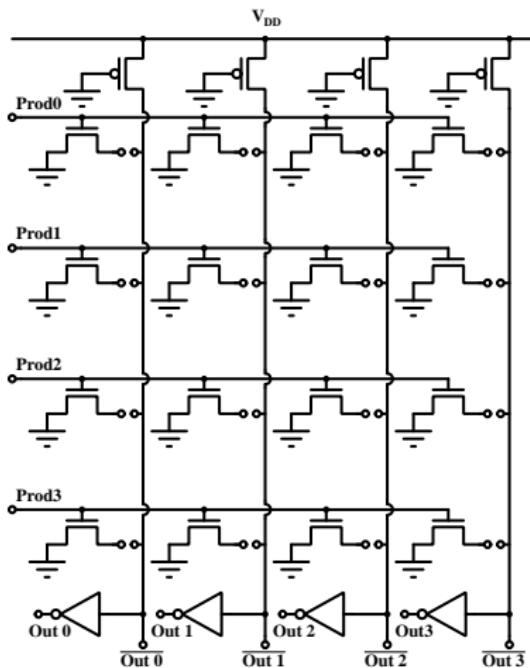
Programmable Logic: Product Array

- In the example shown on the right, we have generated products $\bar{A} \cdot B \cdot C$, $\bar{B} \cdot C$, $A\bar{C}$ and $\bar{A} \cdot B \cdot D$.
- To generate $\bar{A} \cdot B \cdot C$, links from drains of transistors with gates connected to the columns with A , \bar{B} and \bar{C} are connected to the top product row.
- Other products are generated similarly.

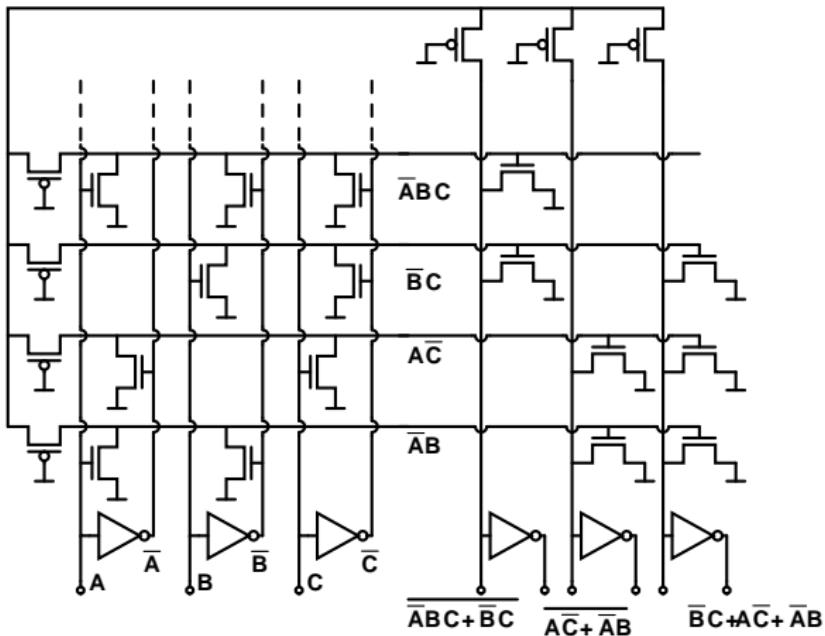


Programmable Logic: Sum Array

- The circuit on the right can produce sums of selected product terms.
- By connecting links at drains of nMOS transistors in a particular column, chosen products can be included in a sum output.
- Several columns can be used to generate different sums of products.
- Outputs are NORs of products. Inverters convert these to ORs of products.



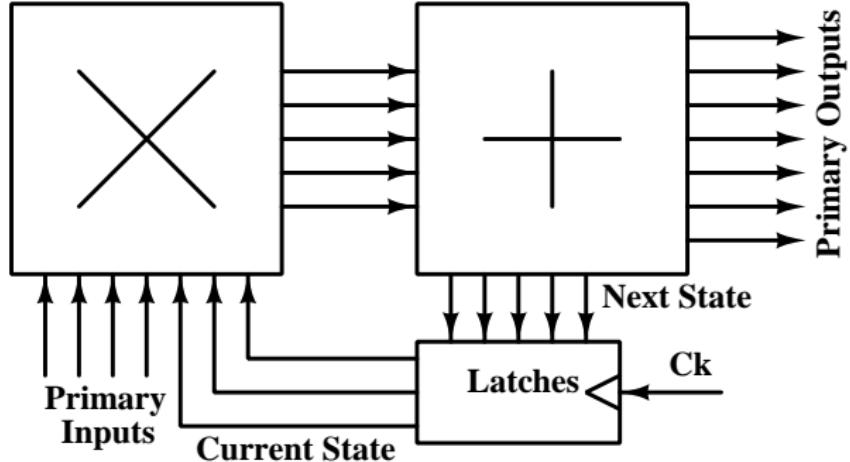
Programmable Logic Arrays



Implementation of Finite State Machines

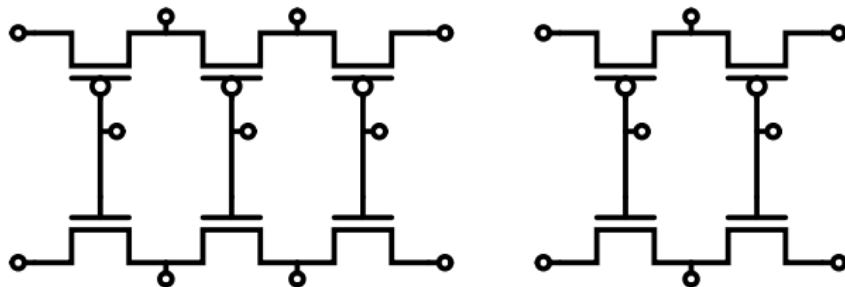
- A finite state machine has the following components:
 - ① Storage elements to store the current state,
 - ② Random logic to compute the next state as a function of current state and current inputs, and
 - ③ Random logic to compute the current output as a function of current state and optionally, the current inputs.
- By adding latches at the output of the PLA, we can provide for all of these.
- Output from latches is fed back to the product array.
- The PLA computes both the next state and current outputs.

Implementation of Finite State Machines



Sea of Gates

- This style uses CMOS logic as its base.
- In CMOS logic, each input goes to an nMOS as well as a pMOS.
- Transistors are pre-placed in a matrix of cells like the one shown below. Interconnects determine what kind of logic will be implemented.

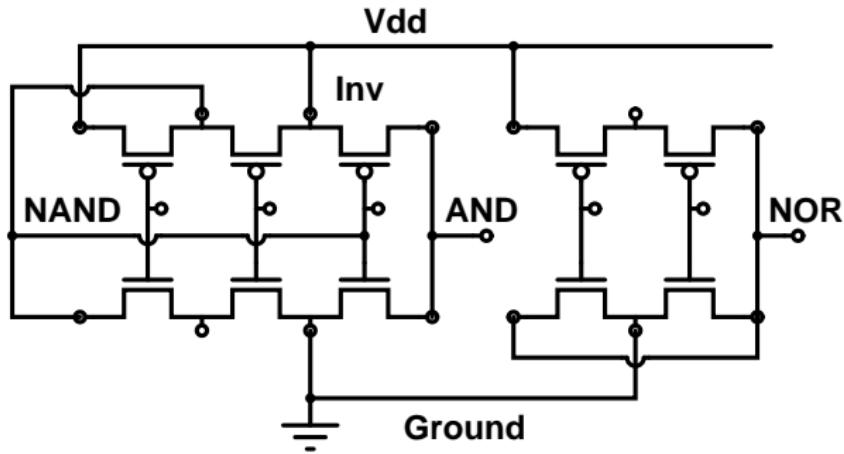


Both n and p channel transistors are in series here!

How do we construct regular CMOS logic gates using these?

Logic from Sea of Gates

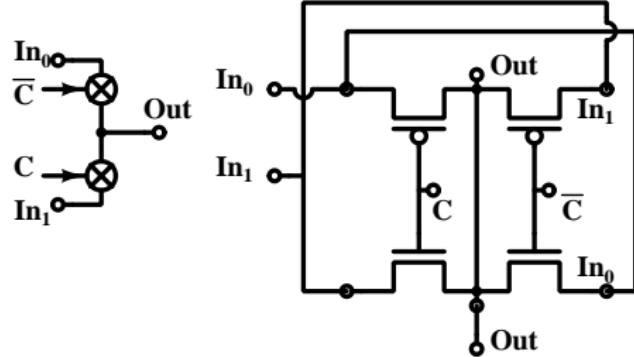
Actually, by wiring the apparently series connected devices, we can convert them to series or parallel as required.



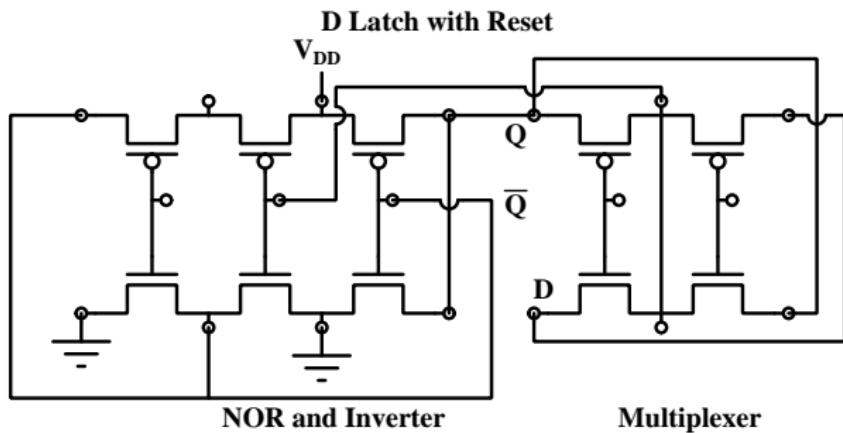
The example above shows a NAND gate, whose output is fed to an inverter to form the AND function. The structure on the right forms a NOR gate.

Sea of Gates Transmission Gate

- The sea of gates template makes use of the fact that all inputs go to an nMOS as well as to a pMOS in CMOS style gates.
- What about structures which don't follow this rule? For example, in a transmission gate, the nMOS and pMOS transistors are driven by complementary signals.
- Transmission gates are often used in pairs with one or the other being on. (Inputs should not be left floating in static CMOS design).
- The pair can be implemented as shown on the right, coupling diagonally opposite transistors in a 2-pair.



Sea of Gates implementation of D latch

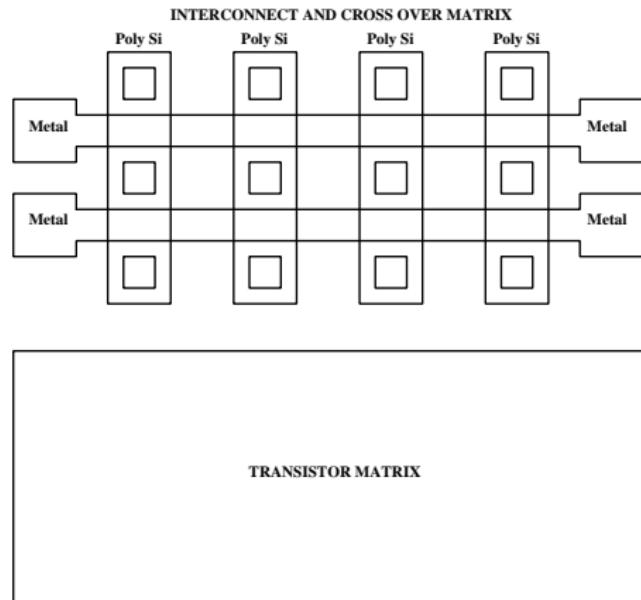


Two such latches can be connected in master-slave configuration to form an edge sensitive D flip-flop.

Interconnect Channels in Semi-Custom Logic

Pre-fabricated interconnect channels alternate with transistor matrices in the fabric of a semi-custom chip.

- The exact shape and composition of these interconnect channels varies from design to design.
- Typically, these will include facilities for local as well as global interconnects.
- These are optimized to provide a sufficient interconnect capacity without wasting too much area.



Using Memories as Logic

- A logic function can be represented by a truth table. This can be stored in memory. Inputs to the logic function act as address pins.
- Pre-computed value of the logic function are stored at the address represented by its input values.
- For any input combination, the address represented by these is looked up and the stored value presented as the output.
- For example, any logic function of 5 inputs can be implemented by a 32×1 bit memory.
- If the same 32 bit memory is organized as 16×2 , it can store two logic functions of up to 4 inputs.

Evolution of FPGAs

Different semi-custom design chips have been introduced with various names. Here is a list:

PLAs: These were the first semi-custom devices to be introduced in the market by Philips in the early 1970's. These are used to implement generic sums of products and have a programmable AND plane as well as a programmable OR plane. Because both AND and OR arrays are programmable in PLAs, these are rather slow.

PALs: A modification of PLAs in which the product array is programmable, but the OR plane is fixed were introduced as Programmable Array Logic or PAL. To cover for the lack of flexibility due to the OR array being fixed, these were introduced in different size combinations and multiple devices were used for different functions.

Evolution of FPGAs

CPLDs As technology progressed, it was possible to put multiple devices on the same chip. Combinations of PALs and PLAs were introduced with programmable interconnect as complex programmable logic devices or CPLDs. Altera was one of the first companies to introduce CPLDs commercially. CPLDs were a huge success and multiple companies introduced CPLDs of different architectures.

Sea of Gates In parallel with field programmable devices, mask programmable devices were manufactured. In these, one (or more) levels of metallization could be customized at the manufacturing site, over a “sea” of pre-fabricated devices. These provided circuits with better performance, but with less flexibility as programming could not be done at the user site.

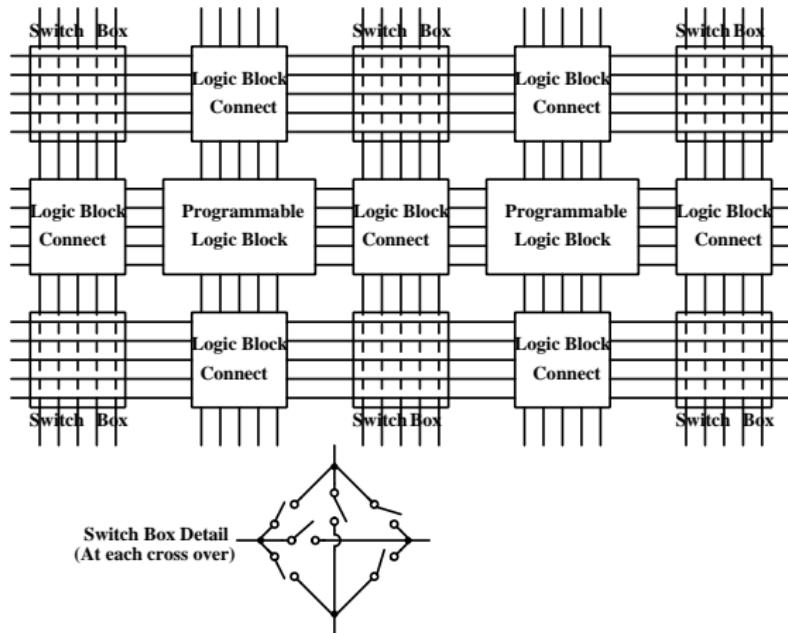
Evolution of FPGAs

FPGAs Field Programmable Gate Arrays borrow features from sea of gates as well as CPLDs. Instead of a “sea” of transistors, these have a “sea” or a repetitive array of combinations of logic elements and memories. In addition, these include a programming infrastructure for interconnects like CPLDs.

On the periphery of these chips, special Input Output devices are fabricated which help in establishing fast interconnection with the external world.

FPGA Architectures

A field Programmable array allows the logic functions as well as the interconnect to be programmed.



Transistors driven by SRAM can be used to program interconnects.



Current FPGA Architecture

Apart from logic blocks and interconnect fabric, modern FPGA's contain other useful structures, such as

- Block RAM,
- Processor cores (Power PC on Xilinx Vertex family),
- Fast multipliers
- I-O Blocks

Types of Reconfigurable Logic

- A typical board, used for implementing a variety of applications will have a micro-controller and a few programmable devices for providing dedicated functions and glue logic.
- This is typically configured once at power on, and then left alone to perform its functions. This is called “static re-configurability”.
- But we may want to re-configure a structure while it is in operation! For example, one can imagine the design of digital filter, which adapts itself during operation itself depending on inputs. This kind of re-configurability is called “dynamic re-configurability”.
- Obviously, dynamic configurability requires that the dead time during re-configuration should be minimized.

Fine and Coarse Grain Reconfigurability

- There is a trade-off between flexibility of re-configuration and the time taken to re-configure.
- In fine grain re-configuration, we can re-program every gate of the design. This is the case for current FPGA and CPLD devices. This gives very good design flexibility, but takes a long time to re-configure.
- In coarse grain re-configuration, relatively large functional blocks are re-configured. For example, by re-connecting a collection of shifters and adders, we can generate any combination of multipliers and adders. The effort to re-configure is smaller, because we do not alter the inner design of shifters and adders. This is compatible with dynamic re-configuration.

Use of Dynamic Re-configuration

Using coarse grain re-configuration, it becomes possible to dynamically change a circuit, *during* its operation.

This has obvious advantages in adaptive circuits.

We normally do not want to re-configure the whole circuit. Indeed the control signals for doing the re-configuration will be generated by a circuit which remains unchanged.

Therefore dynamic re-configuration requires circuits which can be partially re-programmed. Many FPGA are beginning to promise this feature.

Designing Multi-Stage Logic

Dinesh Sharma

EE Department
IIT Bombay, Mumbai

October 16, 2020

Single Stage Delay

In a CMOS gate, for any digital input, either the pull down or the pull up net work is OFF.

When the input is low,

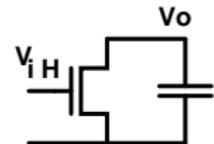
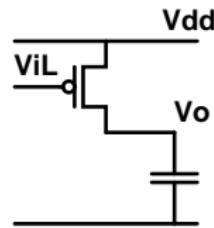
$$I_{dp} = C_L \frac{dV_{out}}{dt} \quad \text{So} \quad dt = C_L \frac{dV_{out}}{I_{dp}} \quad \text{for rise time}$$

When the input is high,

$$I_{dn} = -C_L \frac{dV_{out}}{dt} \quad \text{So} \quad dt = -C_L \frac{dV_{out}}{I_{dn}} \quad \text{for fall time}$$

Where the transistor currents are of the form

$$I_d = K' \frac{W}{L} f(V_{ds}, V_{gs}) \quad \text{Where} \quad K' = \mu C_{ox}$$



Single Stage Delay

- Integrating from the start voltage V_1 to end voltage V_2 , we get expressions of the type

$$\tau_L = \frac{C_L}{K'W/L} \int_{V_1}^{V_2} \frac{dV_{out}}{f(V_{out}, V_{gs})}$$

where τ_L is the time taken to charge/discharge the load capacitor C_L from V_1 to V_2 and K' is the conductance factor given by μC_{ox} .

- The right hand side of this equation is a definite integral. It will evaluate to some constant depending on the voltages defining the 'High' and 'Low' logic levels, the supply voltage, turn on voltages, drain saturation voltage etc.

Single Stage Delay

$$\tau_L = \frac{C_L}{K'W/L} \int_{V_1}^{V_2} \frac{dV_{out}}{f(V_{out}, V_{gs})}$$

In digital design, we keep the channel length at its minimum value, so L is a constant. Let us initially ignore the parasitic capacitances. We can see that

$$\frac{W\tau_L}{C_L} = \text{Constant} \quad \text{so } \tau_L \propto \frac{C_L}{W}$$

This tells us that the delay associated with a gate charging a load capacitor scales directly with C_L and inversely with W , the width of the charging/discharging transistor. This linear dependence permits us to design logic stages easily.

Tapered Buffer

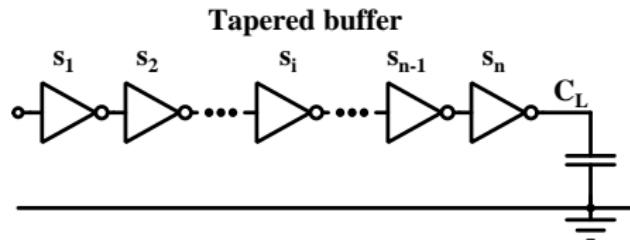
As an example of multi-stage logic, let us take the case when a large capacitor is to be driven by a CMOS circuit.

- A minimum sized inverter will take too long to charge this capacitor.
- Therefore, we would like to scale up the inverter (multiply all transistor widths by a scale factor) in order to drive this large capacitor.
- However, the input capacitance of this scaled up inverter may be too large for a minimum sized inverter to drive!
- Therefore, we need a medium sized inverter to drive the large final inverter.

Tapered Buffer

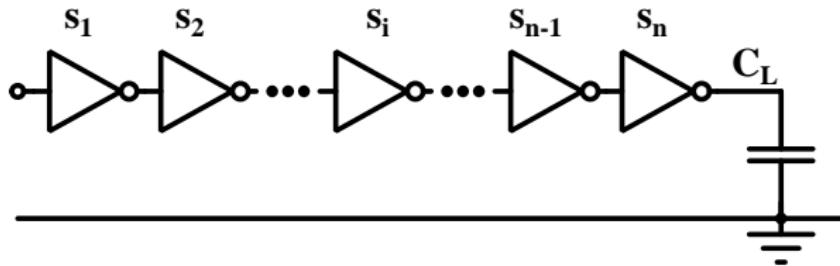
- To drive a large capacitive load, we need a wide transistor, and so we scale up the inverter driving this load.
- To drive this scaled up inverter, we need another inverter of medium size.
- We keep adding inverters, till the first inverter in the chain is small enough to be driven by standard CMOS logic.

This kind of buffering is referred to as a *tapered buffer*.



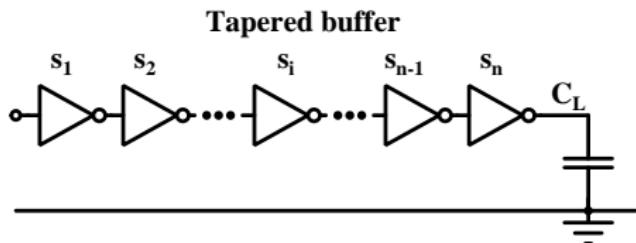
Tapered Buffer

Tapered buffer



- How do we decide the number of inverters to include in this chain?
- What should be the scale factors for each successive stage to minimize the total delay?

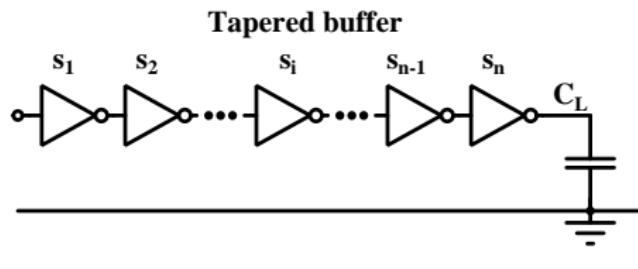
Tapered Buffer



- Let the i 'th inverter in the chain be scaled up by a factor s_i relative to a minimum sized inverter.
- Let the delay of a minimum sized inverter driving another minimum sized inverter be τ .

- The i 'th inverter provides charging current which is s_i times the minimum sized inverter.
- However, the load it sees is s_{i+1} times the input capacitance of a minimum inverter.
- Therefore the delay associated with the i 'th stage is $\frac{s_{i+1}}{s_i} \tau$.

Tapered Buffer



The delay associated with the i 'th stage is $\frac{s_{i+1}}{s_i} \tau$.

So the total delay of the inverter chain is given by

$$d_{total} = \sum_1^n \frac{s_{i+1}}{s_i} \tau = \tau \sum_1^n \frac{s_{i+1}}{s_i}$$

In order to minimize the total delay, we should put the partial derivative with respect to each of the s_i equal to zero. Therefore,

$$\tau \frac{d}{ds_i} \left(\frac{s_2}{s_1} + \dots + \frac{s_i}{s_{i-1}} + \frac{s_{i+1}}{s_i} + \dots \right) = 0$$

Tapered Buffer

Total delay is minimum when

$$\tau \frac{d}{ds_i} \left(\frac{s_2}{s_1} + \dots + \frac{s_i}{s_{i-1}} + \frac{s_{i+1}}{s_i} + \dots \right) = 0$$

Only two terms in the sum contain s_i . Since all scale factors s_i are independent, the derivative of all the rest of the terms is 0. Therefore,

$$\frac{1}{s_{i-1}} - \frac{s_{i+1}}{s_i^2} = 0 \quad \text{Which gives: } \frac{s_i}{s_{i-1}} = \frac{s_{i+1}}{s_i}$$

Tapered Buffer

For minimum delay through the tapered buffer, we must have

$$\frac{s_i}{s_{i-1}} = \frac{s_{i+1}}{s_i}$$

- The *stage ratio*, which is the factor by which an inverter is larger than the previous one, should be **the same** for all stages.
- Let this constant stage ratio for the tapered buffer be ρ .
- The delay contributed by the i^{th} stage is $\frac{s_{i+1}}{s_i} \tau = \rho \tau$.
- The total delay of n stages is then $n\rho\tau$.

Tapered Buffer

- The first stage of the tapered buffer has a size which can be driven by any CMOS gate. Its input capacitance corresponding to this size is C_{in} .
- Each subsequent stage has a drive capability which is ρ times the drive capability of the previous stage.
- Since the drive capability is being stepped up by ρ in n stages, we should have

$$\rho^n = \frac{C_L}{C_{in}} \quad \text{so } \rho = \left(\frac{C_L}{C_{in}} \right)^{1/n}$$

We define the ratio $H \equiv C_L/C_{in}$.

Tapered Buffer

$$\rho^n = \frac{C_L}{C_{in}} = H, \quad \text{so } n = \frac{\ln H}{\ln \rho}$$

$$d_{total} = n\rho\tau = \frac{\ln H}{\ln \rho} \rho \tau = \tau \ln H \frac{\rho}{\ln \rho}$$

In order to minimize d_{total} , we set its derivative with respect to ρ to 0.
This gives

$$\tau \ln H \left(\frac{1}{\ln \rho} - \frac{\rho}{(\ln \rho)^2} \frac{1}{\rho} \right) = 0 \quad \text{which leads to } \frac{1}{\ln \rho} = \frac{1}{(\ln \rho)^2}$$

Thus $\ln \rho = 1$, and therefore $\rho = e$

Thus we obtain the result that the optimum stage ratio for a tapered buffer is e .

The optimum number of stages in the buffer is given by

$$n = \frac{\ln H}{\ln \rho} = \ln H = \ln \frac{C_L}{C_{in}}$$

Obviously, the number of inverters to put in the chain will be given by the nearest integer to this value.

C_L/C_{in} is a given design parameter. We take the integer n closest to $\ln(C_L/C_{in})$ and make a tapered inverter with n stages, with a stage ratio of $e \approx 2.718$.

Generalizing the Tapered Buffer

- We have found that the optimum stage ratio for a tapered buffer is e , and the number of inverters in the chain is $\ln(C_{out}/C_{in})$.
- These results were computed for a situation where the only logic gates used were inverters, and loading due to driver transistors *themselves* in the logic gate was ignored.
- Now we would like to see how to optimize the delay for the general case, where any logic gate can be used in multi-stage logic and the effect of self loading is not ignored.
- We would also like to take the realistic case, where the logic path is not a linear chain and outputs have a fanout $\neq 1$.

Delay in the General Case

To obtain delay for a general chain of logic gates, we need to:

- Take the effects of self-loading into account.
- Consider the delay introduced by logic gates other than inverters.
- Include the effect of branching or fanout, which will occur in a general logic chain.

Effects of Self-Loading

- The input capacitance of the **next** stage is not the only load on a logic gate.
- In addition to the input capacitance of the next stage, a logic gate has to drive the capacitance associated with **its own** output transistors.
- This loading comes from the drain capacitance of the output transistors as well as their drain to gate capacitance.
- This additional capacitance is proportional to width of the driver transistors W . Thus,

$$C_L = C_{ext} + C_p W$$

Where C_p is the parasitic capacitance *per unit width* of driver transistors.

Thus, if we make a logic gate larger, its parasitic load also increases proportionally.

Effects of Self-Loading

- The self loading parasitic capacitance is proportional to W . Thus,

$$C_L = C_{ext} + WC_p$$

Where C_p is the parasitic capacitance *per unit width* of driver transistors.

- If we make a logic gate larger, its parasitic load also increases proportionally.
- Therefore the delay of the stage is

$$\tau_L \propto \frac{C_L}{W} = \frac{C_{ext} + WC_p}{W} \quad \text{Which gives} \quad \tau_L \propto \frac{C_{ext}}{W} + C_p$$

- Thus the parasitic delay associated with self-loading is independent of W .

Effects of Self-Loading

- Our earlier expression for gate delay was $\tau_L \propto C_{next}/W$ where C_{next} was the capacitance presented the next logic gate input.
- For taking the self-loading into account, we just have to add a constant to this delay, so that $\tau_L \propto C_{ext}/W + C_p$ Where C_p is **independent of W**.
- we can write the above expression as

$$\tau_L = \text{const.} \times \frac{C_{ext}}{C_{in}} + \tau_p$$

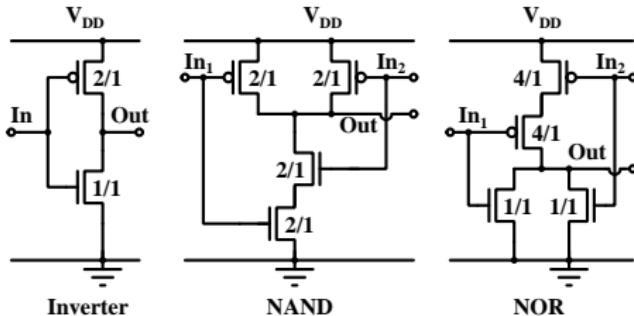
Where τ_p is the parasitic delay associated with a logic gate driving its own output capacitance, and is independent of the size of the driving logic gate or the load.

Multi-stage Logic with Other Gates

Transistor sizes in a gate are adjusted based on several considerations.

- ① Difference in mobility of pMOS and nMOS transistors. A pMOS transistor has to be wider than an nMOS transistor to provide the same drive current because the hole mobility is lower than electrons mobility. For example, we may scale the width of pMOS transistors to be double that of nMOS transistors connected in the same configuration. This ratio is represented by γ .
- ② If there are n series connected transistors, their widths should be scaled up by n . in order to make the output drive of the logic gate equivalent to an unscaled inverter.
- ③ After accounting for mobility differences and series connections, we may scale up *all* transistors by some factor in order to drive larger loads. (This was S_i in our earlier discussion).

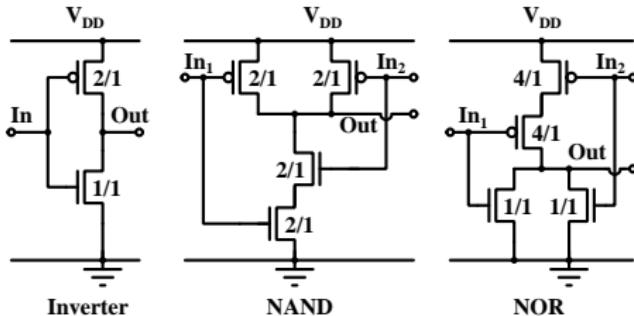
Load presented by 2 input NAND and NOR gates to their drivers.



Here we have assumed $\gamma = 2$.

- Transistor sizes are adjusted to account for mobility differences and series connection.
- Further, we may scale up *all* transistors by some factor in order to drive larger loads.
- This has an impact on the capacitive loading placed on the *previous stage*.

Load presented by 2 input NAND and NOR gates to their drivers.

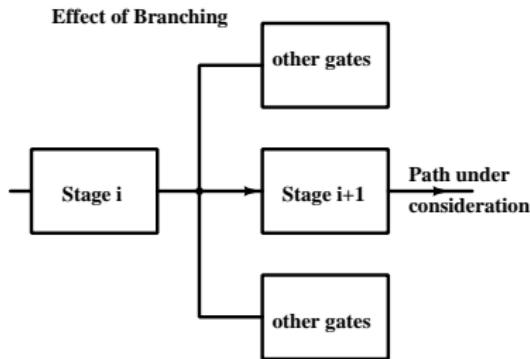


- While an inverter places a load of 3 units on the previous gate, a 2 input NAND gate with the same output drive loads the previous stage with a capacitance of 4 units.
- A 2 input NOR gate loads the previous stage with a capacitance of 5 units.
- We must account for a loading factor for different gate *types* when optimizing multi-stage logic.

Delay of Other Logic Gates

- A minimum sized NAND gate presents a load capacitance which is $4/3$ times that presented by a minimum sized inverter. Similarly, a minimum sized NOR gate loads the previous stage with a capacitance which is $5/3$ times that of a minimum sized inverter.
- These gates then have an output drive which is the same as a minimum inverter.
- It is more convenient to find the delay of a logic gate that has the same input capacitance as an inverter. (This way, the logic type will affect its own delay and not that of the previous gate).
- Therefore the delay of a NAND gate which presents the same input capacitance as an inverter will be $4/3$ times the inverter delay and the NOR gate will have a delay $5/3$ times that of an inverter.

Effect of Branching



- In general, a logic chain will contain points where multiple gates are driven by a stage.
- The effect of this branching (or fanout) must be taken into account while computing delay.

If a stage drives multiple gates, its actual load is the sum of the input capacitances of all branches that it drives.

Thus the delay of this gate is higher by the factor C_{total}/C_{onpath} compared to the delay if it was driving only one path.

Effect of Branching

- The stage driving multiple paths incurs higher delay compared to the case when it drives only a single path.
- We can compute the gate delay as if it was driving only the next gate in the path, and then scale it up by the factor C_{total}/C_{onpath} to obtain the delay with fanout.
- If there is no branching, this factor is just 1, because $C_{total} = C_{onpath}$.
- Except for this correction, we can model the logic chain as if it is just a linear path.

Generalized Logic Chain

We can generalize the optimization made for a tapered buffer to a generic logic path by incorporating the corrections discussed above.

- We can account for the parasitic delay by adding a size independent delay. The parasitic delay depends only on the logic type and not on the size of the gate.
- We can account for different types of logic by scaling up the stage delay of an inverter by the ratio of the input capacitances of the gate and the inverter.
- This correction factor is called the logical effort. This is also size independent and depends only on the type of gate being used.
- We can account for branching by scaling up the charge/discharge time by the ratio of the total capacitance the gate drives with the input capacitance of the next stage.

Logical Effort

- For delay calculation, we can treat a CMOS logic gate as an inverter, with a delay which is scaled up depending on its input capacitance relative to an inverter with the same output drive.
- This correction factor is called the *logical effort* of this gate.
- Logical effort is independent of sizing of the gate. A NAND gate providing twice the output drive of a minimum inverter will have an input capacitance which is $4/3$ times that of an inverter sized up by a factor of 2.
- Thus, the logical effort depends on the logic function provided by a gate and on *nothing else*. (It will depend on γ , of course - but that is a technological constant).

Single gate delay with Logical Effort

We can now handle the delay of a logic gate in a general case.

Self Loading The effect of self loading can be incorporated in delay calculation by using the expression $d = f + p$ where f is the *effort delay*, which depends on transistor currents and load capacitance, while p is the parasitic delay, which is size independent.

Logic type We further express f as a product of two quantities, g and h . Here h is the *electrical effort* of this stage. This is given by the ratio of output capacitance to input capacitance. g is the *logical effort* which accounts for the extra loading caused by a gate as compared to an inverter.

Single gate delay with Logical Effort

- The effort delay is the product of Logical Effort and electrical effort: $f = gh$
- We can express the delay introduced by a logic gate as

$$d = f + p = gh + p$$

- all items in this equation are dimensionless.
- Delay is measured in units of τ , which is the delay of a minimum inverter driving another minimum inverter *not including the parasitic delay*.

This way of expressing delays separates the effects of different components which cause delay, so these can be handled independently.

Single gate delay with Logical Effort

Using the method of logical effort, we separate the effects of different components which cause delay, so these can be handled independently.

- Dependence on technology is encapsulated in τ , which is the delay of a minimum inverter driving another minimum inverter *excluding* delay due to self loading.
- All delays are expressed as a multiple of this quantity.
- Thus delays in this formulation are unitless numbers and must be multiplied with τ to get the absolute value of delay.

Dependence on Gate Sizing

- Dependence on sizing is encapsulated in h . This is the only component of delay which is size dependent.
- h is also a unit less quantity, because it is defined as the ratio of output capacitance to input capacitance.

$$h = \frac{C_{out}}{C_{in}}$$

- These capacitances can be measured in units of the input capacitance of an inverter.
- C_{in} is a measure of the factor by which a gate has been scaled up in order to make it faster.

Dependence on Logic Type

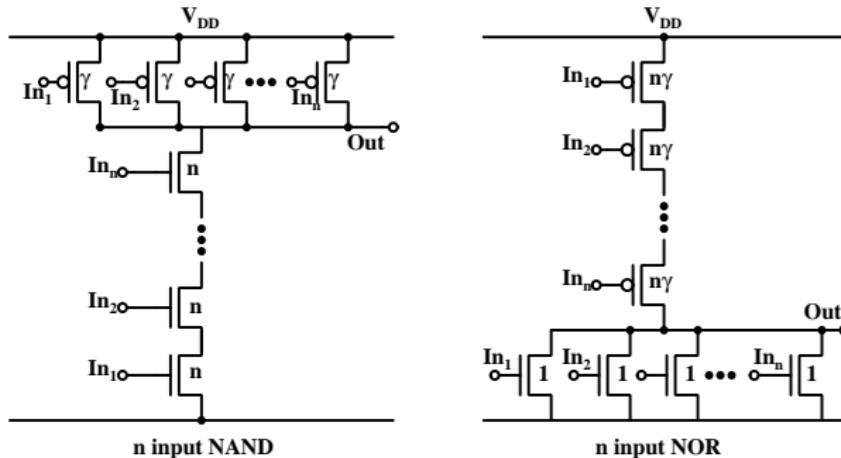
- Dependence on logic type is expressed through g . This accounts for the series/parallel configuration of transistors in a particular gate.
- If the p type transistors are twice the size of n type transistors in minimum inverters, the logical effort of NAND gates is $4/3$, while that of NOR gates is $5/3$.
- Logical effort of other gates can be calculated easily.
- g depends on logic type and *nothing else*.

Effect of Self Loading

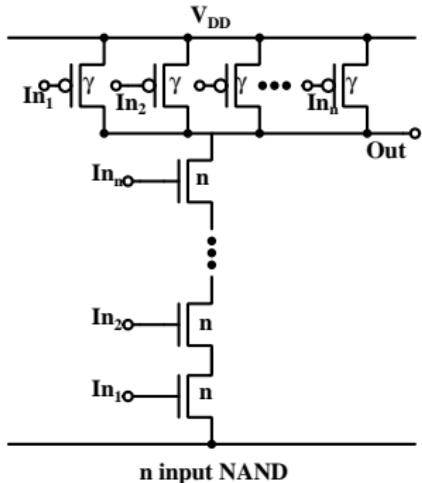
- The effect of parasitic delays due to the load presented by the driver transistors themselves is expressed through p .
- p is size independent.
- It is also a dimensionless quantity as this delay is also expressed in units of τ .

Logical Effort for Common CMOS Gates

- The logical effort of an inverter is, by definition, 1.
- The logical effort of other logic functions depends on their circuit topology.
- N input NAND and NOR gates are shown in the figure below.

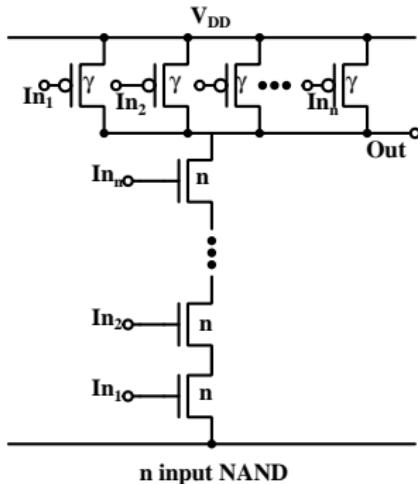


Logical Effort for n input NAND



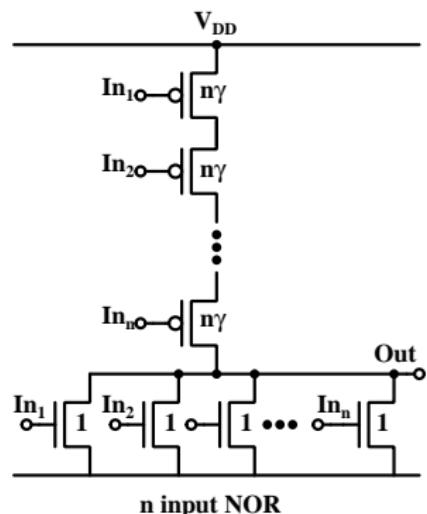
- The n input NAND gate has n nMOS transistors in series and n pMOS transistors in parallel.
- Each nMOS should be n times wider than the nMOS in the minimum inverter.
- Each pMOS will have the same width as that of the minimum inverter – which is γ times the minimum size (to account for lower hole mobility).
- So total capacitive load on any input is $n + \gamma$ units.

Logical Effort for n input NAND



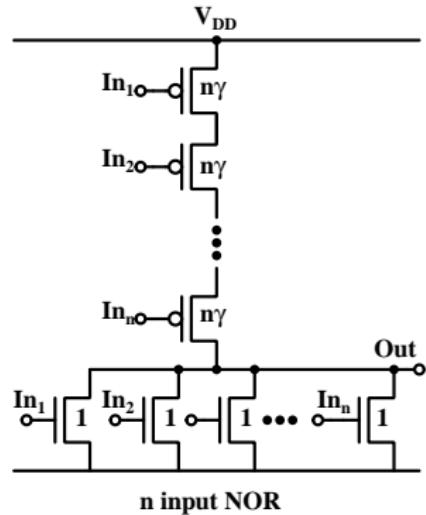
- Each input of the N input NAND gate loads its driver with $(n + \gamma)$ units of capacitance.
- The minimum inverter loads its driver by $(1 + \gamma)$ units.
- So the logical effort of an n input NAND gate is $(n + \gamma)/(1 + \gamma)$.
- This reduces to $4/3$ for a 2 input NAND with $\gamma = 2$, as expected.

Logical Effort for n input NOR



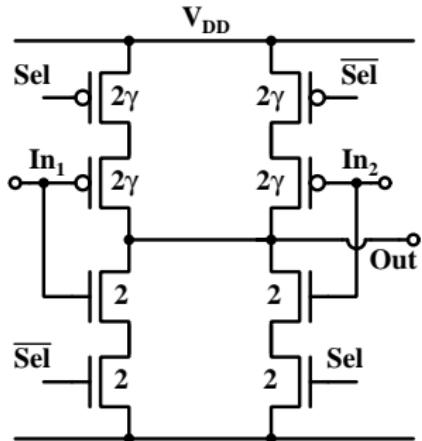
- The n input NOR gate has n pMOS transistors in series and n nMOS transistors in parallel.
- Each pMOS should be n times wider than the pMOS in the minimum inverter, which itself is γ times the minimum size (to account for lower hole mobility).
- Each nMOS will have the same width as that of the minimum inverter.
- So total capacitive load on any input is $1 + n\gamma$ units.

Logical Effort for n input NOR



- Each input of the n input NOR gate loads its driver with $(1 + n\gamma)$ units of capacitance.
- The minimum inverter loads its driver by $(1 + \gamma)$ units.
- So the logical effort of an n input NOR gate is $(1 + n\gamma)/(1 + \gamma)$.
- This reduces to $5/3$ for a 2 input NOR with $\gamma = 2$, as expected.

Logical Effort for a multiplexer



- Each input of the multiplexer is loaded with a capacitance $\propto (2 + 2 \gamma)$.
- The minimum inverter loads its driver by $(1 + \gamma)$ units.
- So the logical effort for the multiplexer is $(2 + 2 \gamma)/(1 + \gamma) = 2$.

It is interesting to see that the logical effort will remain 2 for every data input even when we parallel n tri-stateable inverters to form an n input mux.

Parasitic Delay

Given the series/parallel connections of any logic gate and the rules associated with these connections, we can compute the logical effort of the gate.

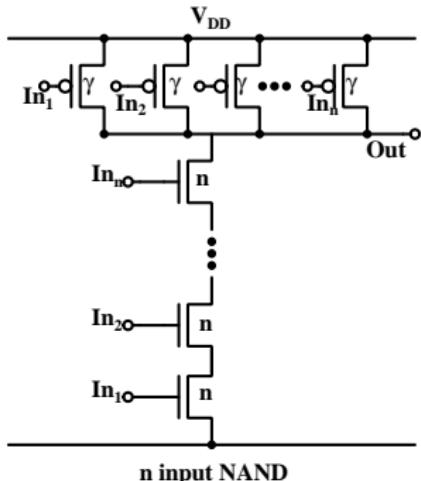
How do we estimate the parasitic delay?

- The parasitic delay of an inverter can be computed through simulation or measurement of delay of an inverter loaded by varying number of inverters. The slope of the delay versus load gives τ , while the intercept will give p_{inv} .
- We can estimate the parasitic delay of a logic gate relative to that of an inverter by looking at the ratio of the diffusion area directly connected to the output node in the gate, as compared to that in an inverter.

Parasitic Delay

- We can approximate the parasitic delay by considering only the self capacitance which is directly connected to the output.
- In case of an inverter, the drain of the n channel transistor (of width 1) and the drain of the p channel transistor (of width γ) are directly connected to the output node.
- Thus the parasitic capacitance for an inverter is $\propto (1 + \gamma)$.
- For any other gate, we find the width of transistors **which are directly connected to the output** node, and find its ratio with $(1 + \gamma)$.
- This permits us to express the parasitic delay of any gate as a multiple of the parasitic delay of the inverter.

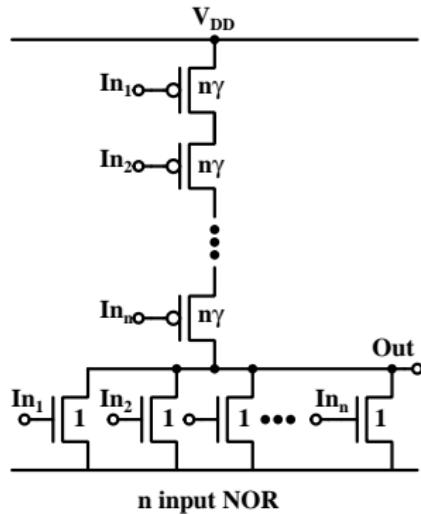
Parasitic Delay of n input NAND



- In case of an n input NAND gate, only one of the series connected n channel transistors (with width = n) is connected to the output node.
- All p channel transistors (each with width γ) are connected to the output node.
- The total parasitic capacitance at the output node is $\propto (n + n \gamma)$

Hence the parasitic delay of an n input NAND gate is
 $(n + n \gamma)/(1 + \gamma) = n$ times the parasitic delay of an inverter.

Parasitic Delay of n input NOR

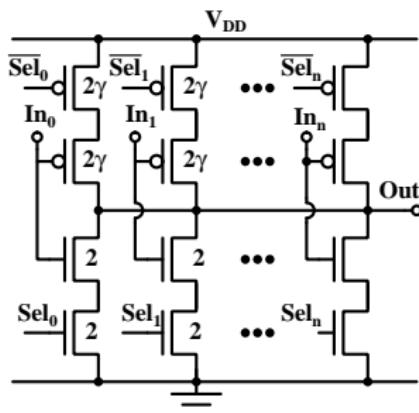


- In case of an n input NOR gate, all n-channel transistors (with width = 1) are connected to the output node.
- Only one of the series connected p channel transistors (with width $n \gamma$) is connected to the output node.
- The total parasitic capacitance at the output node is $\propto (n + n \gamma)$

Hence the parasitic delay of an n input NOR gate is also $(n + n \gamma)/(1 + \gamma) = n$ times the parasitic delay of an inverter.

Parasitic Delay of n input Multiplexer

There are n tri-stateable inverters connected in parallel in an n input multiplexer.



- Each tri-stateable inverter connects an n -channel transistor of width 2 and a p-channel transistor of width 2γ to the output.
- Total parasitic capacitance at the output node is then $2n(1 + \gamma)$.
- Parasitic delay of the n input mux is therefore $2n$ times the parasitic delay of an inverter.

Logical effort and Parasitic Delay of common gates

Gate	Logical Effort	Parasitic Delay
Inverter	1	p_{inv}
2 input NAND	$(2 + \gamma)/(1 + \gamma)$	$2 p_{inv}$
n input NAND	$(n + \gamma)/(1 + \gamma)$	$n p_{inv}$
2 input NOR	$(1 + 2\gamma)/(1 + \gamma)$	$2 p_{inv}$
n input NOR	$(1 + n\gamma)/(1 + \gamma)$	$n p_{inv}$
2 way mux	2	$4 p_{inv}$
n way mux	2	$2n p_{inv}$

The logical effort for an n input mux is independent of n. However, a more complex multiplexer will have a large parasitic delay.

Design of Multi-stage Logic

In multi-stage logic, we consider the logical effort g_i and electrical effort h_i of each stage.

- We define the *path logical effort* as the product of logical effort of all stages on a given path.

$$G \equiv \prod_{i=1}^N g_i$$

- Similarly, we define the *path electrical effort* as the product of electrical effort of all stages on a given path.

$$H \equiv \prod_{i=1}^N h_i, \quad \text{where} \quad h_i = \frac{C_{out_i}}{C_{in_i}} \quad \text{for stage } i.$$

Design of Multi-stage Logic

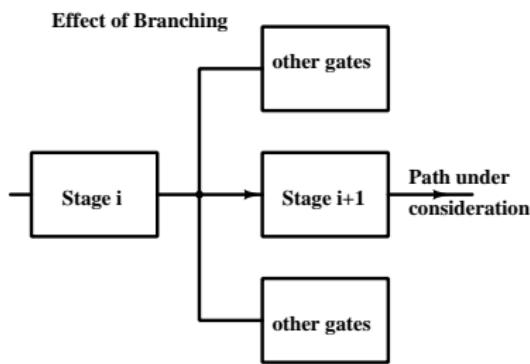
- If there is no branching, the load on a particular stage is just the input capacitance of the next stage, that is: $C_{out_i} = C_{in_{i+1}}$. When we multiply all electrical efforts along a path, all except the first and last capacitances cancel.
- Thus we get

$$H = \frac{C_L}{C_{in_1}}$$

Where C_L is the final load capacitance and C_{in_1} is the input capacitance of the first stage.

Effect of Branching

If, however, there is branching at some stage i , $C_{out_i} \neq C_{in_{i+1}}$.



Now C_{out_i} includes not only $C_{in_{i+1}}$ but also the input capacitance of other logic gates which are not on the path under consideration.

$$C_{total} \equiv C_{onpath} + C_{offpath}$$

$$\text{Where } C_{onpath} = C_{in_{i+1}}$$

We need to introduce a correction factor for this. We define a *branching effort* as

$$b \equiv \frac{C_{onpath} + C_{offpath}}{C_{onpath}}$$

Branching Effort

The branching effort is defined as

$$b \equiv \frac{C_{onpath} + C_{offpath}}{C_{onpath}} \quad \text{while} \quad h \equiv \frac{C_{onpath}}{C_{in}}$$

The C_{onpath} in electrical effort h considers only the on path loading. Because of this, H retains its cancellation property:

$$H = \prod_i h_i = \frac{C_{in_2}}{C_{in_1}} \frac{C_{in_3}}{C_{in_2}} \cdots \frac{C_L}{C_{in_n}} = \frac{C_L}{C_{in_1}}$$

The branching effort b provides the correction for the actual loading seen by a logic stage.

$$b_i h_i = \frac{C_{total_{i+1}}}{C_{onpath_{i+1}}} \frac{C_{onpath_{i+1}}}{C_{in_i}} = \frac{C_{total_{i+1}}}{C_{in_i}}$$

Path Branching Effort

- If there is no branching at a stage, the corresponding b_i value is 1 since $C_{offpath} = 0$. So, multiplication by b_i does not change anything.
- If, however, there is branching at the i 'th stage, multiplying h_i by b_i corrects the output capacitance, because

$$b_i h_i = \frac{C_{total_{i+1}}}{C_{onpath_{i+1}}} \frac{C_{onpath_{i+1}}}{C_{in_i}} = \frac{C_{total_{i+1}}}{C_{in_i}}$$

- We define the branching effort of the whole path, denoted by B , as the product of the branching effort at each stage along the path.

$$B = \prod_{i=1}^N b_i$$

Path Effort

We can now define the *path effort*, F as the product of all logical efforts and branch corrected electrical efforts.

$$F = \prod_{i=1}^N g_i \prod_{i=1}^N b_i h_i = \prod_{i=1}^N g_i \prod_{i=1}^N b_i \prod_{i=1}^N h_i$$

So,

$$F = GBH \quad \text{where } G = \prod_{i=1}^N g_i, B = \prod_{i=1}^N b_i, \text{ and } H = \prod_{i=1}^N h_i$$

The advantage of using branching correction separately from electrical effort is that H retains its cancellation property for capacitance of intermediate stages and is defined only by the final load and the input capacitance. ($H = C_L / C_{in_1}$).

Path Effort

- The equation that defines the path effort looks quite similar to the definition of the stage effort.
- Notice, however, that unlike the stage effort f , the path effort F does not define the delay of the path.
- The total delay is the *sum* of individual delays and not their product.

$$D = \sum d_i = \sum g_i b_i h_i + \sum p_i$$

- Still, F is a useful quantity for optimisation of path delays as we shall see later.

Path Delay

- The path delay, D , is the sum of the delays of each of the N stages of logic in the path.
- As in the expression for delay in a single stage we separate the contribution to path delay from effort and the delay due to parasitic capacitances.

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

where D_F is the path effort delay, while P is the *path parasitic delay*.

- The path effort delay is simply $D_F = \sum g_i b_i h_i$ and the path parasitic delay is $P = \sum p_i$.

Minimizing Path Delay

- The total path delay is given by

$$D = \sum d_i = \sum g_i b_i h_i + \sum p_i = \sum g_i b_i \frac{C_{i+1}}{C_i} + \sum p_i$$

- We need to adjust the size of each stage (and hence C_i) such that D is minimum.
- Therefore we should set the partial derivative of the expression for D with respect to each of C_i to 0.
- All p_i are size independent, and therefore give 0 on differentiating with respect to C_i .
- Only two terms in the first sum involve C_i . These are $g_{i-1} b_{i-1} C_i / C_{i-1} + g_i b_i C_{i+1} / C_i$.

Minimizing Path Delay

- In the expression for path delay, only two terms depend on C_i . These are $g_{i-1}b_{i-1}C_i/C_{i-1} + g_i b_i C_{i+1}/C_i$.
- All other terms will give 0 on differentiation with respect to C_i .

$$\frac{\partial D}{\partial C_i} = 0 = \frac{g_{i-1}b_{i-1}}{C_{i-1}} - \frac{g_i b_i C_{i+1}}{C_i^2}$$

This leads to $g_{i-1}b_{i-1}\frac{C_i}{C_{i-1}} = g_i b_i \frac{C_{i+1}}{C_i}$

So $g_{i-1}b_{i-1}h_{i-1} = g_i b_i h_i$ for all i

Minimizing Path Delay

- The path delay is minimized when each stage in the path has the same stage effort, $f = gbh$.
- Since the Path Effort F is the product of all stage efforts and the stage effort has to be equal for all stages for minimum delay, we must have $\hat{f} = F^{1/N}$. (A hat over a symbol indicates an expression that achieves minimum delay.)
- For this optimum effort, we obtain

$$\hat{D} = NF^{1/N} + P = N(GBH)^{1/N} + P$$

\hat{D} is the minimum delay possible for this path.

- We achieve this minimum delay by appropriate sizing of transistors in each stage of the logic path, so that all stages have the same effort.

Minimizing Path Delay

- Minimization of path delay requires that each logic stage be designed so that the stage effort $f \equiv g_i b_i h_i$ is the same for all stages.
- This gives the optimum value of stage effort as
 $\hat{f} = F^{1/N} = (GBH)^{1/N}$.
- Since $F = GBH$ is known, we can compute \hat{f} . (While all h_i are not known, H is known to be C_L/C_{in_1} because of the cancellation property).
- Since \hat{f} can be calculated, we can compute h for all stages.
- We start with the last stage, where the output capacitance is known ($=C_L$). Knowing h , we can compute C_{in} – and hence the scale factor for this stage.

Sizing for Minimum Path Delay

$$h = \frac{\hat{f}}{gb} = \frac{(GBH)^{1/N}}{gb}$$

$$\text{Since } h_i \equiv \frac{C_{in_{i+1}}}{C_{in_i}}, \quad \frac{C_{in_{i+1}}}{C_{in_i}} = \frac{(GBH)^{1/N}}{gb}$$

This gives

$$C_{in_i} = \frac{g_i b_i}{(GBH)^{1/N}} C_{in_{i+1}}$$

We can use this recursive relation for computing the scale, and hence transistor sizes for all stages, starting with the last one.

Sizing for Minimum Path Delay

- We have the recursive relation

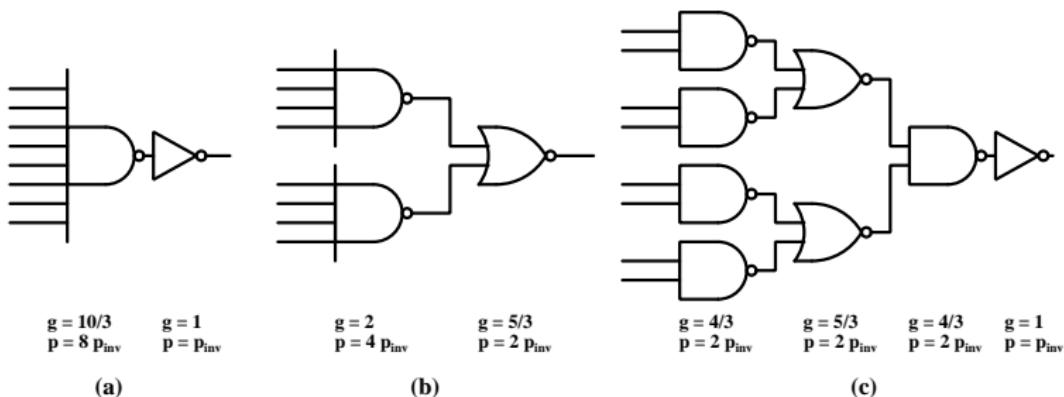
$$C_{in_i} = \frac{g_i b_i}{(GBH)^{1/N}} C_{in_{i+1}}$$

- Knowing C_L , we can compute the input capacitance of the last stage.
- From the input capacitance of the last stage, we can compute the input capacitance of stage preceding it using the recursive relation.
- in this way, C_{in} for every stage can be determined.
- From C_{in} , we can calculate the scale factor, and hence the geometry of transistors for each stage.

Example: An 8-input AND network

When a large number of inputs must be combined, there are several options for the structure of the circuit.

The figure below shows three configurations for computing the AND function of eight inputs. Which one is best?



Example: An 8-input AND network

The path logical effort, G , is the product of the logical efforts of the logic gates along the path. In the following example, we assume $\gamma = 2$ and $p_{inv} = 0.6$.

- $G = 10/3 \times 1 = 3.33$ for configuration a,
- $G = 6/3 \times 5/3 = 3.33$ for case b, and
- $G = 4/3 \times 5/3 \times 4/3 \times 1 = 2.96$ for configuration c.

Since there is no branching, $B = 1$.

We estimate the parasitic delay of n input NANDs and NORs to be $n p_{inv}$.

Example: An 8-input AND network

We can write the total delay for the three configurations as:

$$\text{Configuration a: } D = 2(3.33H)^{1/2} + 5.4$$

$$\text{Configuration b: } D = 2(3.33H)^{1/2} + 3.6$$

$$\text{Configuration c: } D = 4(2.96H)^{1/4} + 4.2$$

It is clear from these equations that case b will always be better than a.

Example: An 8-input AND network

Configuration a: $D = 2(3.33H)^{1/2} + 5.4$

Configuration b: $D = 2(3.33H)^{1/2} + 3.6$

Configuration c: $D = 4(2.96H)^{1/4} + 4.2$

- Choice between cases b and c depends on the electrical effort, H.
- When H = 1, configuration b has a delay of 7.25, while that for configuration c is 9.45. So in this case, b will be best.
- For H = 12, the delays for configurations b and c are 16.25 and 13.97 respectively, so configuration c will be best.

The equations show that for high electrical effort, case c yields least delay because the $H^{1/4}$ factor dominates. (In the current example, configuration c is best for $H > 5.68$).

Example: An 8-input AND network

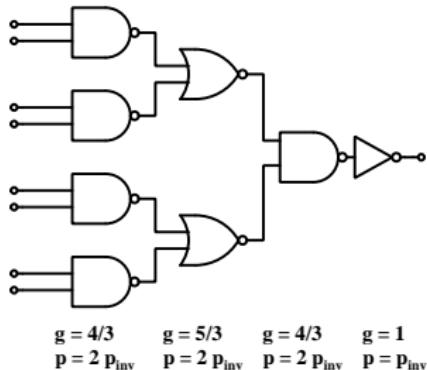
- Let us take the example of 8-input AND circuit to see how all geometries can be calculated using logical effort.
- We take the input capacitance of a minimum inverter as the unit of capacitance.
- The unit of time is τ , the delay of a reference inverter driving another identical reference inverter *excluding* its parasitic delay.
- The unit of transistor width will be the width of the n transistor in the reference inverter.
- We shall take $\gamma = 2$ and $p_{inv} = 0.6$ in this example.

Example: 8-input AND geometry computation

- We are given that the load to be driven is equivalent to 64 reference inverters. ($C_{out} = 64$).
- The input source is designed to drive up to 4 reference inverters. ($C_{in} = 4$).

$H = 64/4 = 16$. As discussed before configuration c, which is a 4 stage design, is best for this value of H

Example: 8-input AND geometry computation



- We can take the logic path from any of the inputs to the final output.
- On this path, we shall encounter:
 - 1 a 2 input NAND ($g = 4/3, b = 1$),
 - 2 a 2 input NOR ($g = 5/3, b = 1$),
 - 3 a 2 input NAND ($g = 4/3, b = 1$), and
 - 4 an inverter ($g = 1, b = 1$)

$$G = \frac{4}{3} \times \frac{5}{3} \times \frac{4}{3} \times 1 = 80/27 = 2.963 \quad B = 1, \quad H = \frac{64}{4} = 16$$

$$F = GBH = 47.4074, \quad \text{So} \quad \hat{f} = 47.4074^{1/4} = 2.624$$

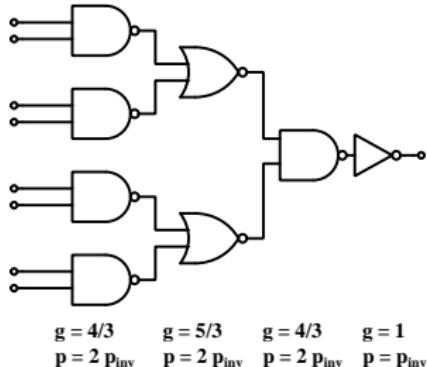
Relating C_{in} to Transistor Geometry

- Unit of capacitance is the input capacitance of the reference inverter.
- Unit of width is the width of the n channel transistor in the reference inverter.
- So $1 + \gamma$ units of width correspond to 1 unit of capacitance.
- The input capacitance of a reference logic gate, obtained from the reference inverter by series parallel rules, is g .
- Thus, $(C_{in} = g) \Rightarrow$ scale factor = 1 over the reference logic gate.
- For any given C_{in} , scale factor = C_{in}/g .

If we know C_{in} , we can find the scale factor by dividing it by g , and then multiply all transistor widths in the reference logic gate by this scale factor to get individual transistor geometries.

Last stage: inverter

We begin from the load end in this example. The last stage is an inverter, with $g = 1, b = 1$.



$$\hat{f} = 2.624 = gbh = 1 \times 1 \times h. \quad \text{So} \quad h = 2.624$$

$$h = \frac{C_{out}}{C_{in}} = \frac{64}{C_{in}} = 2.624$$

$$\text{So} \quad C_{in} = \frac{64}{2.624} = 24.39$$

So the final stage inverter is scaled up by 24.39 compared to the reference inverter. Taking the n channel transistor width in the reference inverter as the unit,

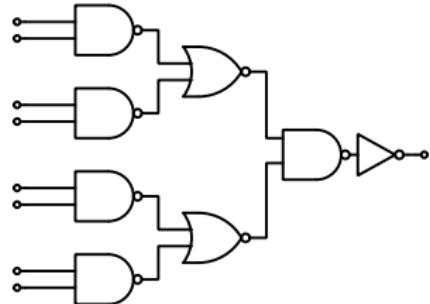
n-channel transistor width = 24.39

p-channel transistor width = $\gamma \times 24.39 = 48.78$.

3rd stage: 2 input NAND

The stage driving the inverter is a 2 input NAND.

So $g = 4/3$, $b = 1$, $C_{out} = 24.39$.



$$\begin{array}{ll} g = 4/3 & \\ p = 2 p_{inv} & \end{array}$$

$$\begin{array}{ll} g = 5/3 & \\ p = 2 p_{inv} & \end{array}$$

$$\begin{array}{ll} g = 4/3 & \\ p = 2 p_{inv} & \end{array}$$

$$\begin{array}{ll} g = 1 & \\ p = p_{inv} & \end{array}$$

$$\hat{f} = 2.624 = g b h = \frac{4}{3} \times 1 \times h$$

$$\text{So } h = \frac{2.624 \times 3}{4} = 1.968$$

$$h = \frac{C_{out}}{C_{in}} = \frac{24.39}{C_{in}} = 1.968$$

$$\text{Therefore } C_{in} = \frac{24.39}{1.968} = 12.3936$$

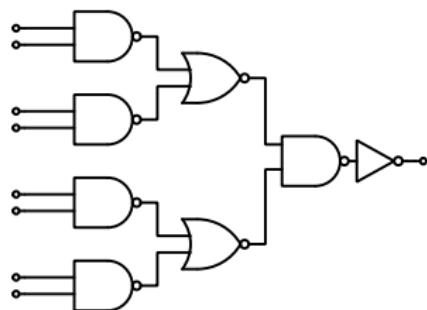
Scale factor for this stage is $12.3936/g = (12.3936 \times 3)/4 = 9.2952$. The reference 2 input NAND gate has n channel transistor width = 2, and p channel transistor width = 2.

Therefore the transistor geometries in this stage will be $2 \times 9.2952 = 18.59$ for both n and p channel transistors.

2nd stage: 2 input NOR

The stage driving the 2 input NAND is a 2 input NOR.

So $g = 5/3$, $b = 1$, $C_{out} = 12.3936$.



$$\hat{f} = 2.624 = g b h = \frac{5}{3} \times 1 \times h. \quad \text{So} \quad h = 1.5744$$

$$h = \frac{C_{out}}{C_{in}} = \frac{12.3936}{C_{in}} = 1.5744$$

$$\text{Therefore } C_{in} = \frac{12.3936}{1.5744} = 7.872$$

Scale factor for this stage is $7.872/g = (7.872 \times 3)/5 = 4.7232$.

The reference 2 input NOR gate has n channel transistor width = 1, and p channel transistor width = 4.

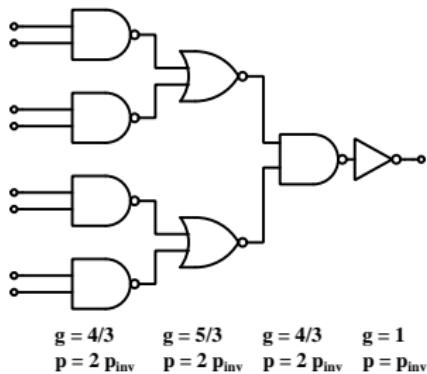
n-channel transistor width = 4.72,

p channel transistor width = $4 \times 4.7232 = 18.89$

First stage: 2 input NAND

Finally, we come to the first stage which is a 2 input NAND.

So $g = 4/3$, $b = 1$, $C_{out} = 7.872$.



$$\hat{f} = 2.624 = gbh = \frac{4}{3} \times 1 \times h. \text{ So } h = 1.9680$$

$$h = \frac{C_{out}}{C_{in}} = \frac{7.872}{C_{in}} = 1.968$$

$$\text{Therefore } C_{in} = \frac{7.872}{1.968} = 4$$

This agrees with our specification that the input capacitance of the first stage should be equivalent to 4 inverters.

The scale factor will be $4/g = 4/(4/3) = 3$. In the reference NAND, all transistors have a width = 2. so in the first stage, all transistors will have a width = $3 \times 2 = 6$.

Design of 8 input AND: Summary of results

The following table gives the geometries of transistors in all stages:

Stage	I	II	III	IV
Logic type	2in NAND	2in NOR	2in NAND	Inverter
g	4/3	5/3	4/3	1
C_{in}	4	7.87	12.39	24.39
Scale Factor	3	4.7232	9.2952	24.39
n width	6	4.72	18.59	24.39
p width	6	18.89	18.59	48.78
Parasitic Delay	1.2	1.2	1.2	0.6

The total delay of the 4 stage implementation is:

$$4\hat{f} + \sum P_i = 4 \times 2.624 + 4.2 = 10.496 + 4.2 = 14.7$$

Optimizing the path length

- The delay optimization procedure discussed earlier assumes that the number of stages N is known.
- However, this may not be the optimum path length and we can sometimes get a faster circuit by buffering intermediate outputs in this logic path. (For example, the delay of the tapered buffer had an optimum number of inverters).
- How do we determine the optimum number of stages in the general case?

Optimizing the path length

- We assume that there is a logic path containing n_1 stages and we are free to add n_2 inverters to this path, if that results in a lower overall delay.
- We now consider the optimization of this logic path containing $N = n_1 + n_2$ stages.
- We shall assume that there is no requirement for n_2 to be even. (This implies that an inverted output is equally acceptable or else, the logic path and inputs can be suitably altered to produce the desired output).
- The optimization problem is to find the scale factors for each of the $N = n_1 + n_2$ stages, such that the delay is minimum.

Optimizing the path length

- The path effort $F = GBH$ for the n_1 stages of logic is known.
- This is because the logical effort of each of the logic gates is known to us, so that G may be evaluated.
- Branching, if any, in the logic chain is also known, so B can be evaluated.
- finally, H depends only on the final load capacitance and input capacitance, which is the starting specification for the optimization.

Optimizing the path length

- Addition of n_2 inverters does not change the value of G , since $g = 1$ for each of the n_2 inverters.
- Similarly, the inverters do not introduce any additional branching, so B remains the same.
- Finally, H is defined by the final load and the input capacitance of the first logic element, which is not changed by the inserted inverters.
- Therefore $F = GBH$ for the $N = n_1 + n_2$ stages (including n_2 inverters) is the same as the F for n_1 logic stages.

Optimizing the path length

- Additional inverters in the logic chain permit sharing the effort over a larger number of stages, which can reduce the total delay.
- We first find the optimum value of N .
- If $N > n_1$, we shall have the opportunity of reducing the delay by adding inverters.

Optimizing the path length

- The total delay in the N stages is the sum of delays of n_1 logic stages and n_2 inverters.
- For optimum delay, the stage effort should be equal for all the N stages.
- Therefore, the stage effort of logic stages as well as the inverters is the same and is $= F^{1/N}$.
- So the total delay is:

$$\hat{D} = NF^{1/N} + \sum_{i=1}^{n_1} p_i + (N - n_1)p_{inv}$$

Optimizing the path length

$$\hat{D} = NF^{1/N} + \sum_{i=1}^{n_1} p_i + (N - n_1)p_{inv}$$

- The first term in the equation above is the effort delay of N stages.
- The sum of parasitic delays of n_1 logic gates gives us the second term.
- Finally, the parasitic delay of $n_2 = N - n_1$ inverters gives the third term.
- We define the optimum stage effort $\rho \equiv F^{1/N}$. Then

$$\hat{D} = N(\rho + p_{inv}) + \sum_{i=1}^{n_1} p_i - n_1 p_{inv}$$

Optimizing the path length

$$\hat{D} = N(\rho + p_{inv}) + \sum_{i=1}^{n_1} p_i - n_1 p_{inv}$$

Since $\rho \equiv F^{1/N}$, $N = \ln F / \ln \rho$. Therefore,

$$\hat{D} = \frac{\ln F}{\ln \rho} (\rho + p_{inv}) + \sum_{i=1}^{n_1} p_i - n_1 p_{inv}$$

- Notice that F , n_1 , p_i and p_{inv} are given constants.
- We can optimize the total delay with respect to ρ by setting the derivative of \hat{D} with respect to ρ to zero.

Optimizing the path length

$$\hat{D} = \frac{\ln F}{\ln \rho} (\rho + p_{inv}) + \sum_{i=1}^{n_1} p_i - n_1 p_{inv}$$

Differentiating by parts, we get

$$\frac{\partial \hat{D}}{\partial \rho} = 0 = -\frac{\ln F}{(\ln \rho)^2} \cdot \frac{1}{\rho} \cdot (\rho + p_{inv}) + \frac{\ln F}{\ln \rho} (1)$$

Therefore,

$$\frac{\ln F}{\ln \rho} = \frac{\ln F}{(\ln \rho)^2} \cdot \frac{1}{\rho} \cdot (\rho + p_{inv})$$

and so,

$$1 = \frac{1}{\rho \ln \rho} (\rho + p_{inv})$$

Optimizing the path length

$$1 = \frac{1}{\rho \ln \rho} (\rho + p_{inv})$$

This gives

$$\rho + p_{inv} = \rho \ln \rho$$

Which can be written as

$$p_{inv} + \rho(1 - \ln \rho) = 0$$

This condition is independent of F and the value of ρ is uniquely defined by p_{inv} .

Solving for Stage Ratio ρ

$$p_{inv} + \rho(1 - \ln \rho) = 0$$

This equation cannot be solved in closed form and either iterative solutions or graphical solutions have to be used to determine ρ from p_{inv} .

In the special case when $p_{inv} = 0$, we have

$$\rho(1 - \ln \rho) = 0 \quad \text{so } \ln \rho = 1 \quad \text{which gives } \rho = e$$

This corresponds to the case of tapered inverter that we had solved before.

Solving for Stage Ratio ρ

$$p_{inv} + \rho(1 - \ln \rho) = 0$$

For non-zero values of p_{inv} , this equation can be solved iteratively using Newton Raphson technique.

We define

$$f(\rho) = \rho(1 - \ln \rho) + p_{inv} = 0$$

$$\text{Then } f'(\rho) = (1 - \ln \rho) + \rho \left(-\frac{1}{\rho} \right) = -\ln \rho$$

Let us illustrate the iterative method by taking $p_{inv} = 1$.

We know that for $p_{inv} = 0$, the value of ρ is e . A guess value to start iterations can be $\rho = 3$.

Solving for Stage Ratio ρ

$$f(\rho) = \rho(1 - \ln \rho) + p_{inv} = 0 \quad \text{and} \quad f'(\rho) = -\ln \rho$$

We illustrate iterative solution of this equation taking $p_{inv} = 1$ and the initial guess for $\rho = 3$.

Each successive guess for ρ can be calculated as

$$\rho_{next} = \rho - \frac{f(\rho)}{f'(\rho)} = \rho + \frac{\rho(1 - \ln \rho) + p_{inv}}{\ln \rho}$$

$$\text{So} \quad \rho_{next} = \frac{\rho + p_{inv}}{\ln \rho}$$

Solving for Stage Ratio ρ

ρ	$\rho_{next} = (\rho + p_{inv}) / \ln \rho$
3.0000	3.6410
3.6410	3.5914
3.5914	3.5911
3.5911	3.5911

The value of ρ converges to four decimal digits within four iterations.

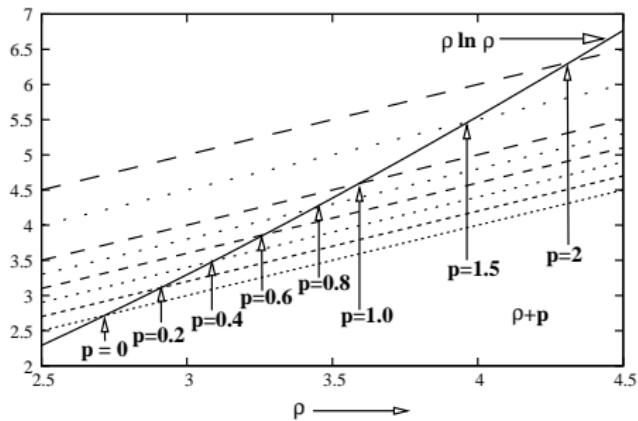
Solving for Stage Ratio ρ

$$\rho(1 - \ln \rho) + p_{inv} = 0 \quad \text{So} \quad \rho \ln \rho = \rho + p_{inv}$$

We can also solve the above equation graphically by plotting $\rho + p_{inv}$ and $\rho \ln \rho$ as functions of ρ . The value of ρ at which these two intersect is the solution of the equation.

The figure on the right shows the solution for different values of p_{inv} .

For a given CMOS process, p_{inv} is fixed. Therefore, the value of ρ needs to be computed only once for a given process.



Stage delay

The table below gives the values of ρ and the corresponding stage delay for several values of parasitic delay p .

p	ρ	$\ln \rho$	$d = \rho + p$
0	2.718 (e)	1.000	2.718
0.2	2.912	1.069	3.11
0.4	3.093	1.129	3.49
0.6	3.266	1.184	3.87
0.8	3.432	1.233	4.23
1.0	3.591	1.278	4.59
1.5	3.967	1.378	5.47
2.0	4.319	1.463	6.32

Finding the optimum number of stages

- Once ρ is known, we can evaluate the optimum number of logic stages for a given path effort as $N = \ln F / \ln \rho$.
- This value will be fractional in general and needs to be zapped to the nearest integer.
- If $N > n_1$ (where n_1 is the number of logic stages required for implementing the desired logic function), we can insert $N - n_1$ inverter stages to optimize the overall delay.
- After this the stage effort can be re-calculated as $f = F(1/N)$.
- Once f and N are known, we can proceed as before to compute the sizes of each stage.

Finding the optimum number of stages

If we have the freedom of inserting a number of inverters in the logic path to optimize delay, we follow the following procedure:

- From p_{inv} , find the ideal stage effort ρ by solving $p_{inv} + \rho(1 - \ln \rho) = 0$. This can be done through iterative solutions or graphically as described earlier.
- Once ρ is known, find the number of stages as $\ln F / \ln \rho$. The nearest integer to the value so found is the optimum number of stages N .
- If $N > n_1$, insert $(N - n_1)$ inverters anywhere in the logic path. If $N \leq n_1$, take $N = n_1$.
- The stage effort is now adjusted to $f = F^{1/N}$.

- Given the value of f , we can start from the last stage and work backwards as earlier to calculate all transistor geometries.
- For the last stage the output capacitance is known ($=C_L$). The input capacitance can be calculated from

$$C_{in_N} = \frac{g_N}{f} C_L$$

This gives the scale factor for this stage from which, geometries of transistors in the last stage can be computed.

- For each preceding stage, we use the recursive relation

$$C_{in_i} = \frac{g_i b_i}{f} C_{in_{i+1}}$$

- From C_{in_i} , we can calculate the scale factor, and hence the geometry of all transistors for this stage.

Sensitivity of Delay to the Number of Stages

We would like to know how much the path delay changes if the number of stages deviates from the optimum.

N/\hat{N}	D/\hat{D}	N/\hat{N}	D/\hat{D}
0.25	7.42	1.4	1.06
0.5	1.46	2.0	1.24
0.7	1.09	3.0	1.62
1/0	1/00	4.0	2.01

This table assumes $p_{inv} = 0.6$.

We can see that delay is quite insensitive to the number of stages, provided the deviation from optimum is not too large. Using 40% higher N increases the delay by just 6%.

Sensitivity of Delay to the Number of Stages

- As we have seen, using 40% higher N increases the delay by just 6%.
- In fact, doubling the number of stages from optimum increases the delay only 24%.
- Using half as many stages as the optimum increases the delay by 46%.
- Delay penalty is smaller for higher than necessary N compared to lower N for the same deviation from the optimum.
- A stage or two more or less in a design with many stages will make little difference, provided proper transistor sizes are used. Only when very few stages are required does a change of one or two stages make a large difference.

Logic Design

Dinesh Sharma
EE Department, IIT Bombay

(Design Examples for Logical Effort have been taken from the paper by Sutherland)

October 17, 2020

Contents

1	Transistor Models	1
2	Static CMOS Logic Design	5
2.1	Static CMOS Design style	5
2.2	CMOS Inverter	5
2.2.1	Static Characteristics	6
2.2.2	Noise margins	11
2.2.3	Dynamic Considerations	13
2.2.4	Trade off between power, speed and robustness	17
2.2.5	CMOS Inverter Design Flow	17
2.2.6	Conversion of CMOS Inverters to other logic	17
2.2.7	The series parallel rule	18
2.2.8	Application of Series Parallel rule to complex logic	22
3	Pseudo-nMOS Logic Design	25
3.1	Static Characteristics	26
3.1.1	For $0 \leq V_i \leq V_{Tn}$: nMOS ‘off’	27
3.1.2	nMOS saturated, pMOS linear	27
3.1.3	nMOS linear, pMOS linear	28
3.1.4	nMOS linear, pMOS saturated	28
3.2	Noise margins	29
3.3	Dynamic characteristics	30
3.3.1	Rise Time	30
3.3.2	Fall Time	30
3.4	Pseudo nMOS design Flow	32
3.5	Conversion of pseudo nMOS Inverter to other logic	33
4	Dual Rail Logic Design	35
4.1	Complementary Pass gate Logic	35
4.1.1	Basic Multiplexer Structure	35
4.1.2	Logic Design using CPL	36

4.1.3	Buffer Leakage Current	37
4.2	Cascade Voltage Switch Logic	39
5	Dynamic Logic	41
5.1	Basic CMOS Dynamic Gate	41
5.1.1	Problem with Cascading CMOS dynamic logic	42
5.2	Four Phase Dynamic Logic	44
5.3	Domino Logic	47
5.4	Zipper logic	48
6	Semi-Custom Design	51
6.1	Procedure for Semi-Custom Design	52
6.2	Customization of interconnects	52
6.3	Implementation of Configurable Template	53
6.3.1	Programmable Logic Arrays	53
6.3.2	Sea of Gates	58
6.4	Interconnect Channels in Semi-Custom Logic	60
6.5	Using Memories as Logic	61
6.6	Field Prgrammable Gate Arrays	61
7	Multi-Stage Logic Design	65
7.1	Stage Delay and Sizing	65
7.2	Tapered Buffer	66
7.3	Using Logic Gates Other Than Inverters	68
7.3.1	Delay model for a single gate	69
7.3.2	Considering the Effects of Self-Loading	69
7.3.3	Gate Delays with Logical Effort	70
7.3.4	Logical Effort for Common CMOS Gates	71
7.4	Design of multi-stage logic	73
7.4.1	Branching Effort	73
7.4.2	An example: 8-input AND network	76
7.5	Optimizing the path length	81

List of Figures

1.1	MOS characteristics according to the simple analytic model	1
1.2	MOS characteristics with non zero conductance in saturation	2
2.1	The basic CMOS inverter	5
2.2	nMOS and pMOS drain currents for different input voltages	6
2.3	Output voltage for nMOS in saturation, pMOS in linear regime	7
2.4	Output voltage when both transistors are saturated	9
2.5	Transfer Curve of a CMOS inverter	9
2.6	Output voltage when nMOS is in linear regime while pMOS is saturated.	10
2.7	CMOS inverter with the nMOS ‘off’	14
2.8	CMOS inverter with the pMOS ‘off’	16
2.9	CMOS implementation of $\overline{A.B + C.(D+E)}$	18
3.1	Pseudo nMOS inverter	25
3.2	Transistor currents in a Pseudo nMOS inverter	26
3.3	‘high’ to ‘low’ transition on the output	30
3.4	Pseudo NMOS implementation of $\overline{A.B + C.(D+E)}$	33
4.1	Basic Multiplexer with logic restoring inverters	36
4.2	Implementation of XOR and XNOR by CPL logic.	36
4.3	Implementation of (a) AND-NAND and (b) OR-NOR functions using complementary pass-gate logic.	37
4.4	High leakage current in inverter	37
4.5	Pull up pMOS to avoid leakage in the inverter	38
4.6	Problem with a low to high transition on the output	38
4.7	Pseudo-nMOS NOR	39
4.8	Pseudo-nMOS OR from complemented inputs	39
4.9	OR-NOR implementation in Cascade Voltage Switch Logic	40
5.1	CMOS dynamic gate to implement $\overline{(A+B).C}$	41
5.2	Dynamic gate for $\overline{(A+B).C}$ cascaded with an inverter.	42
5.3	Generation of phase clocks for dynamic logic	44

5.4	CMOS 4 phase dynamic logic	45
5.5	Operation of a type 3 gate	46
5.6	CMOS 4 phase dynamic logic drive constraints	47
5.7	CMOS domino logic	47
5.8	Zipper logic	49
6.1	fuse structure	53
6.2	An anti-fuse. The insulator could be amorphous silicon.	53
6.3	Architetcture of a Programmable Logic Array	54
6.4	Product and sum implementation in a PLA	54
6.5	A programmable product array	55
6.6	Example for generation of different products from inputs	56
6.7	Sum array for generating sums of products	57
6.8	A finite state machine implemented with a PLA	58
6.9	Pattern of transistors in a sea of gates template	59
6.10	Implementing a NAND, Inverter and NOR gates in sea of gates	59
6.11	A pair of transmission gates with complementary control inputs	60
6.12	A transparent D latch implemented with “Sea of Gates”	60
6.13	An example interconnect channel for semi-custom logic	61
6.14	Alternating logic and interconnect boxes in an FPGA	63
7.1	A tapered buffer	66
7.2	Load presented by 2 input NAND and NOR gates to their drivers.	69
7.3	n input NAND and NOR gates	71
7.4	A 2 way multiplexer	72
7.5	Three circuits that compute the AND function of eight inputs.	76
7.6	4 stage implementation of 8 input AND	78
7.7	Graphical solution of the equation $\rho + p_{inv} = \rho \ln \rho$	83

Chapter 1

Transistor Models

We shall use simple analytical models for MOS transistors. We use a sign convention according to which, voltage and current symbols associated with the pMOS transistor (such as V_{Tp}) have positive values. Then, the n channel formulae can be used for both transistors and we shall assign signs to quantities explicitly.

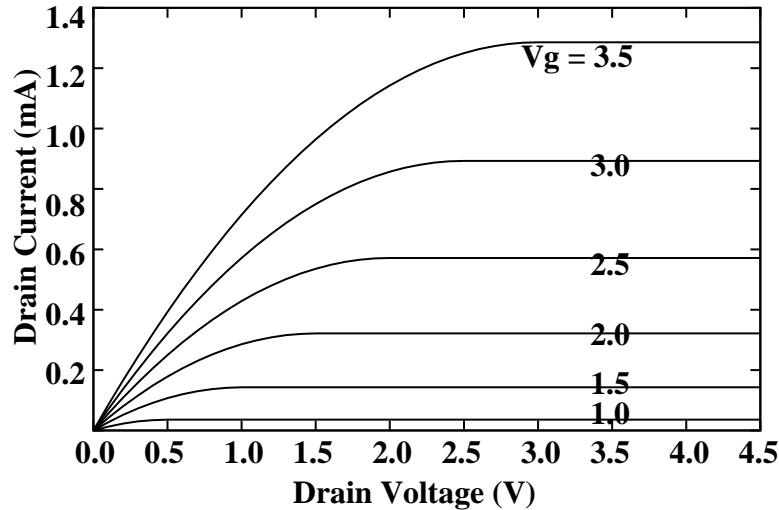


Figure 1.1: MOS characteristics according to the simple analytic model

The model we use is described by the following equations:
for $V_{gs} \leq V_T$,

$$I_{ds} = 0 \quad (1.1)$$

for $V_{gs} > V_T$ and $V_{ds} \leq V_{gs} - V_T$,

$$I_{ds} = K \left[(V_{gs} - V_T)V_{ds} - \frac{1}{2}V_{ds}^2 \right] \quad (1.2)$$

and for $V_{gs} > V_T$ and $V_{ds} > V_{gs} - V_T$,

$$I_{ds} = K \frac{(V_{gs} - V_T)^2}{2} \quad (1.3)$$

The saturation region equation is somewhat oversimplified because it assumes that the current is independent of V_{ds} . In reality, the current has a weak dependence on V_{ds} in this region.

In order to model the saturation region more accurately, we adopt an “Early Voltage” like formalism.

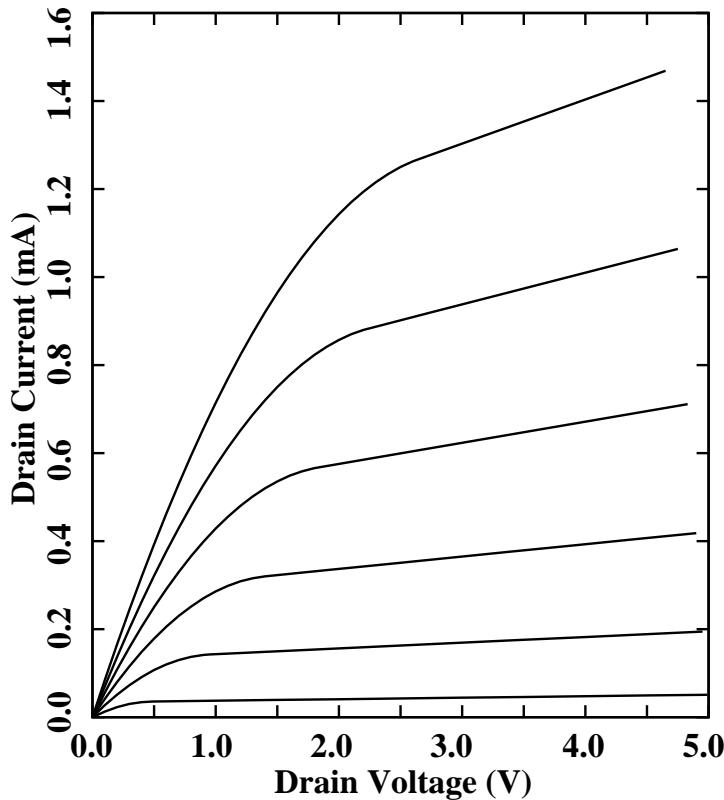


Figure 1.2: MOS characteristics with non zero conductance in saturation

It is assumed that the current increases linearly in the saturation region. All linear characteristics in saturation can be produced backwards towards negative drain voltages and will intersect the drain voltage axis at a single point at $-V_E$. (This is, at best, an approximation). Because the conductance in saturation is now non zero, the onset of saturation has to be redefined, so that the current and its derivative are continuous at the boundary of linear and

saturation regimes. The current equations are given by:

For $V_{gs} > V_T$ and $V_{ds} \leq V_{dss}$,

$$I_{ds} = K \left[(V_{gs} - V_T)V_{ds} - \frac{1}{2}V_{ds}^2 \right] \quad (1.4)$$

and for $V_{gs} > V_T$ and $V_{ds} > V_{dss}$,

$$I_{ds} = I_{dss} \frac{V_d + V_E}{V_{dss} + V_E} \quad (1.5)$$

Where V_E is the ‘Early Voltage’. Here V_{dss} and I_{dss} are saturation drain voltage and drain current respectively. Since the current values must match at either side of $V_{ds} = V_{dss}$, we must have:

$$I_{dss} \equiv K \left[(V_{gs} - V_T)V_{dss} - \frac{1}{2}V_{dss}^2 \right]. \quad (1.6)$$

For the curve to be smooth and continuous at $V_d = V_{dss}$, the value of the first derivative should match on either side of V_{dss} . Therefore,

$$K(V_{gs} - V_T - V_{dss}) = \frac{I_{dss}}{V_{dss} + V_E}$$

So,

$$K(V_{gs} - V_T - V_{dss})(V_{dss} + V_E) = K \left[(V_{gs} - V_T)V_{dss} - \frac{1}{2}V_{dss}^2 \right] \quad (1.7)$$

This leads to a quadratic equation in V_{dss}

$$\frac{1}{2}V_{dss}^2 + V_E V_{dss} - (V_{gs} - V_T)V_E = 0 \quad (1.8)$$

Solving this quadratic, we get

$$V_{dss} = V_E \left(\sqrt{1 + \frac{2(V_{gs} - V_T)}{V_E}} - 1 \right) \quad (1.9)$$

For $V_E \gg V_{gs} - V_T$ this reduces to

$$V_{dss} \simeq (V_{gs} - V_T) \left(1 - \frac{V_{gs} - V_T}{2V_E} \right) \quad (1.10)$$

Characteristics of a MOS transistor using this model are shown in fig.1.2. While accurate modeling of the output conductance is essential for linear design, the simpler model assuming constant I_d in saturation is often adequate for preliminary digital design. In any case, final designs will have to be validated with detailed simulations. In this booklet, we shall use the simple model for MOS devices to keep the algebra simple.

Chapter 2

Static CMOS Logic Design

Static logic circuits are those which can hold their output logic levels for indefinite periods as long as the inputs are unchanged. Circuits which depend on charge storage on capacitors are called dynamic circuits and will be discussed in a later chapter.

2.1 Static CMOS Design style

The most common design style in modern VLSI design is the Static CMOS logic style. In this, each logic stage contains pull up and pull down networks which are controlled by input signals. The pull up network contains p channel transistors, whereas the pull down network is made of n channel transistors. The networks are so designed that the pull up and pull down networks are never ‘on’ simultaneously. This ensures that there is no static power consumption.

2.2 CMOS Inverter

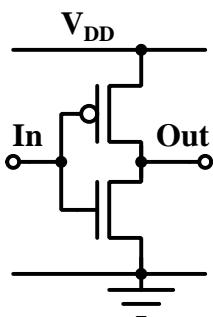


Figure 2.1: The basic CMOS inverter

The simplest of such logic structures is the CMOS inverter. In fact, for any CMOS logic design, the CMOS inverter is the basic gate which is first analyzed and designed in detail. Thumb rules are then used to convert this design to other more complex logic. The basic CMOS inverter is shown in fig. 2.1. We shall develop the characteristics of CMOS logic through the inverter structure, and later discuss ways of converting this basic structure more complex logic gates.

2.2.1 Static Characteristics

The range of input voltages can be divided into several regions.

- nMOS ‘off’, pMOS ‘on’
- nMOS saturated, pMOS linear
- nMOS saturated, pMOS saturated
- nMOS linear, pMOS saturated
- nMOS ‘on’, pMOS ‘off’

Let us compute the output voltage for a series of input voltages from 0V to V_{dd} . We have plotted the individual drain currents of the n and p channel transistors as functions of the output voltage for different input voltages V_{in} .

The gate voltage for the nMOS transistor = V_{in} and the drain voltage = V_{out} while the

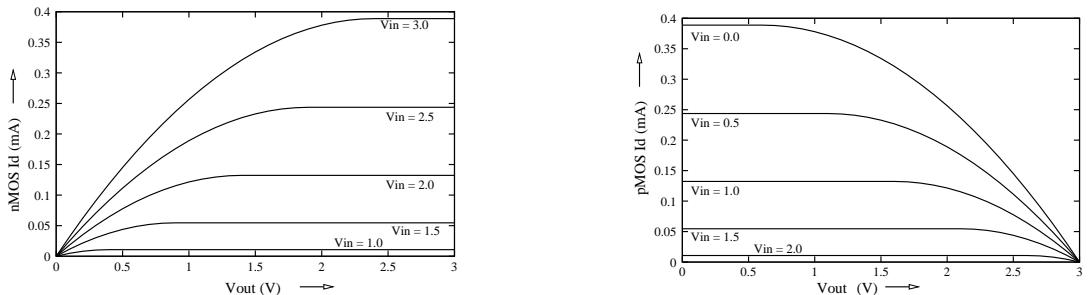


Figure 2.2: nMOS and pMOS drain currents for different input voltages

absolute gate voltage for pMOS is $V_{DD} - V_{in}$ and the absolute drain voltage is $V_{DD} - V_{out}$.

The Output voltage of the inverter will be the value where the two currents are equal for any given V_{in} .

nMOS ‘off’, pMOS ‘on’

For $0 < V_i < V_{Tn}$ the n channel transistor is ‘off’, the p channel transistor is ‘on’ and the output voltage = V_{dd} . This is the normal digital operation range with input = ‘0’ and output = ‘1’.

nMOS saturated, pMOS linear

In this regime, both transistors are ‘on’. The input voltage V_i is $> V_{Tn}$, but is small enough so that the n channel transistor is in saturation, and the p channel transistor is in the linear regime. In static condition, the output voltage will adjust itself such that the currents through the n and p channel transistors are equal.

The absolute value of gate-source voltage on the p channel transistor is $V_{dd} - V_i$, and therefore

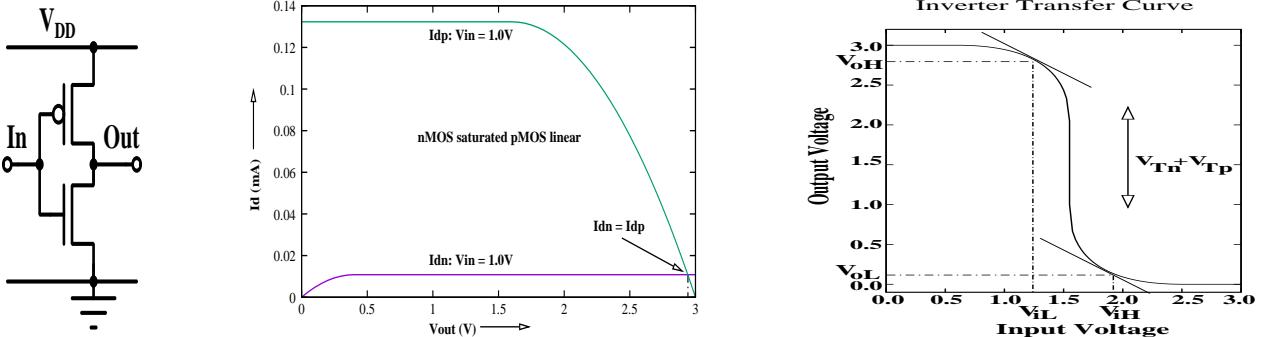


Figure 2.3: Output voltage for nMOS in saturation, pMOS in linear regime

the “over voltage” on its gate is $V_{dd} - V_i - V_{Tp}$. The drain source voltage of the pMOS has an absolute value $V_{dd} - V_o$.

Therefore,

$$I_d = K_p \left[(V_{dd} - V_i - V_{Tp})(V_{dd} - V_o) - \frac{1}{2}(V_{dd} - V_o)^2 \right] = \frac{K_n}{2}(V_i - V_{Tn})^2 \quad (2.1)$$

Where symbols have their usual meanings.

We define $\beta \equiv K_n/K_p$. We make the substitution $V_{dp} \equiv V_{dd} - V_o$, where V_{dp} is the absolute value of the drain-source voltage for the p channel transistor. Then,

$$(V_{dd} - V_i - V_{Tp})V_{dp} - \frac{1}{2}V_{dp}^2 = \frac{\beta}{2}(V_i - V_{Tn})^2 \quad (2.2)$$

Which gives the quadratic

$$\frac{1}{2}V_{dp}^2 - V_{dp}(V_{dd} - V_i - V_{Tp}) + \frac{\beta}{2}(V_i - V_{Tn})^2 = 0 \quad (2.3)$$

Solutions to the quadratic are:

$$V_{dp} = (V_{dd} - V_i - V_{Tp}) \pm \sqrt{(V_{dd} - V_i - V_{Tp})^2 - \beta(V_i - V_{Tn})^2} \quad (2.4)$$

These equations are valid only when the pMOS is in its linear regime. This requires that

$$V_{dp} \equiv V_{dd} - V_o \leq V_{dd} - V_i - V_{Tp}$$

Therefore, we must choose the negative sign. Thus

$$V_{dd} - V_o = (V_{dd} - V_i - V_{Tp}) - \sqrt{V_{dd} - V_i - V_{Tp})^2 - \beta(V_i - V_{Tn})^2} \quad (2.5)$$

Therefore,

$$V_o = V_i + V_{Tp} + \sqrt{V_{dd} - V_i - V_{Tp})^2 - \beta(V_i - V_{Tn})^2} \quad (2.6)$$

Since V_o must be $\geq V_i + V_{Tp}$, the limit of applicability of the above result is given by

$$(V_{dd} - V_i - V_{Tp})^2 = \beta(V_i - V_{Tn})^2$$

That is, the solution for V_o is valid for

$$V_i \leq \frac{V_{dd} + \sqrt{\beta}V_{Tn} - V_{Tp}}{1 + \sqrt{\beta}} \quad (2.7)$$

In the case where we size the n and p channel transistors such that

$$K_n = K_p; \text{ so } \beta = 1$$

we have

$$V_o = (V_i + V_{Tp}) + \sqrt{(V_{dd} - V_{Tn} - V_{Tp})(V_{dd} - 2V_i + V_{Tn} - V_{Tp})} \quad (2.8)$$

with

$$V_i \leq \frac{V_{dd} + V_{Tn} - V_{Tp}}{2}$$

nMOS saturated, pMOS saturated

At the limit of applicability of eq. 2.7, when the input voltage is exactly at

$$V_i = \frac{V_{dd} + \sqrt{\beta}V_{Tn} - V_{Tp}}{1 + \sqrt{\beta}} \quad (2.9)$$

both transistors are saturated. Since the currents of both transistors are independent of their drain voltages in this condition, we do not get a unique solution for V_o by equating drain currents.

The currents will be equal for all values of V_o in the range

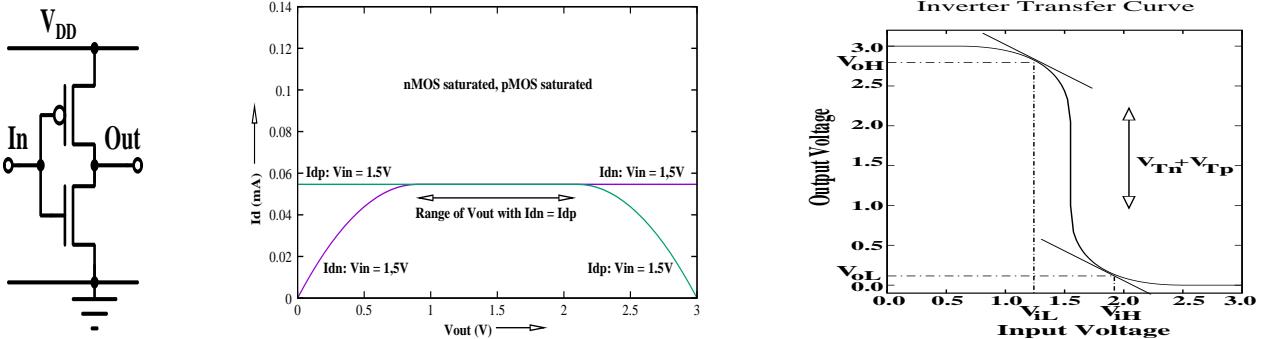


Figure 2.4: Output voltage when both transistors are saturated

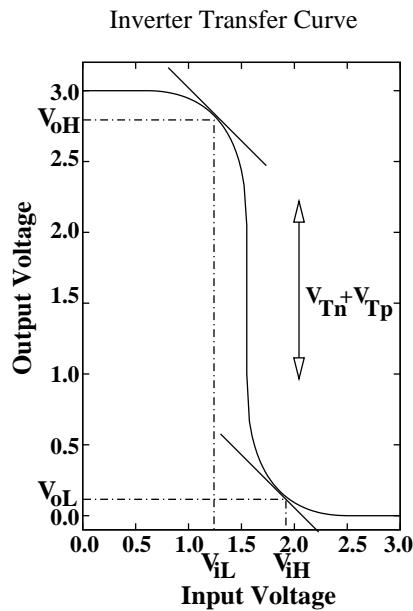


Figure 2.5: Transfer Curve of a CMOS inverter

$$V_i - V_{Tn} \leq V_o \leq V_i + V_{Tp}$$

Thus the transfer curve of an inverter shows a drop of $V_{Tn} + V_{Tp}$ at a voltage near $V_{dd}/2$. This is actually an artifact of the simple transistor model chosen for this analysis, which assumes perfect saturation of drain current. In a real case, the drain current does depend on the drain voltage (albeit weakly) in the saturation region. If the model incorporates an Early Voltage like effect, the drop near the middle of the characteristic is more gradual.

nMOS linear, pMOS saturated

At the gate voltage given by eq. 2.9, both transistors are saturated. As we increase V_i beyond this value, such that

$$\frac{V_{dd} + \sqrt{\beta}V_{Tn} - V_{Tp}}{1 + \sqrt{\beta}} < V_i < V_{dd} - V_{Tp}$$

both transistors are still ‘on’, but nMOS enters the linear regime while pMOS gets saturated. Equating currents in this condition,

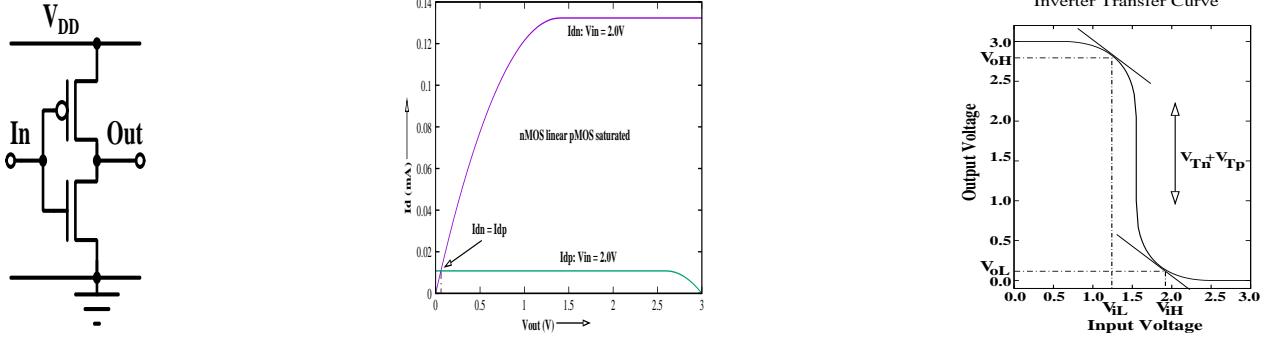


Figure 2.6: Output voltage when nMOS is in linear regime while pMOS is saturated.

$$I_d = \frac{K_p}{2}(V_{dd} - V_i - V_{Tp})^2 = K_n \left[(V_i - V_{Tn})V_o - \frac{1}{2}V_o^2 \right] \quad (2.10)$$

From this, we get the quadratic equation

$$\frac{1}{2}V_o^2 - (V_i - V_{Tn})V_o + \frac{(V_{dd} - V_i - V_{Tp})^2}{2\beta} = 0 \quad (2.11)$$

This has solutions

$$V_o = (V_i - V_{Tn}) \pm \sqrt{(V_i - V_{Tn})^2 - \frac{(V_{dd} - V_i - V_{Tp})^2}{\beta}} \quad (2.12)$$

Since the equations are valid only when the n channel transistor is in the linear regime ($V_o < V_i - V_{Tn}$), we choose the negative sign. This gives,

$$V_o = (V_i - V_{Tn}) - \sqrt{(V_i - V_{Tn})^2 - \frac{(V_{dd} - V_i - V_{Tp})^2}{\beta}} \quad (2.13)$$

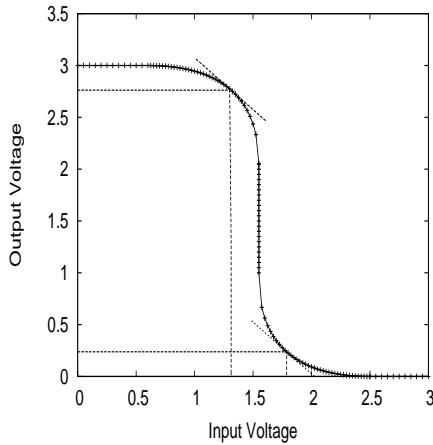
Again, in the special case where $\beta = 1$, we have

$$V_o = (V_i - V_{Tn}) - \sqrt{(V_{dd} - V_{Tn} - V_{Tp})(2V_i - V_{dd} - V_{Tn} + V_{Tp})} \quad (2.14)$$

nMOS ‘on’, pMOS ‘off’

As we increase the input voltage beyond $V_{dd} - V_{Tp}$, the p channel transistor turns ‘off’, while the n channel conducts strongly. As a result, the output voltage falls to zero. This is the normal digital operation range with input = ‘1’ and output = ‘0’.

The figure below shows the transfer curve of an inverter with $V_{dd} = 3V$, $V_{Tn} = 0.6V$ and $V_{Tp} = 0.5V$, and $\beta = 1$.



The plot produced by SPICE for this circuit with realistic models is quite similar.

2.2.2 Noise margins

The requirement from a digital circuit is that it should distinguish logic levels, but be insensitive to the exact analog voltage at the input. This implies that the flat portions of the transfer curve (where $\frac{\partial V_o}{\partial V_i}$ is small) are suitable for digital logic. We select two points on the transfer curve where the slope $(\frac{\partial V_o}{\partial V_i})$ is -1.0. The coordinates of these two points define the values of (V_{iL}, V_{oH}) and (V_{iH}, V_{oL}) . Robust digital design requires that the output high level be higher than what is acceptable as a high level at the input ($V_{oH} > V_{iH}$). The difference between these two levels is the ‘high’ noise margin. This is the amount of noise that can ride on the worst case ‘high’ output and still be accepted as a ‘high’ at the input of the next gate. Similarly, we require $V_{oL} < V_{iL}$. The difference, $V_{iL} - V_{oL}$ is the ‘low’ noise margin. Obviously, it is of interest to evaluate the values of these noise margins. For the discussion which follows, we shall use the expressions derived earlier for $\beta = 1$ to keep the algebra simple.

Calculation of V_{iL} and V_{oH}

from eq. (2.8)

$$V_o = (V_i + V_{Tp}) + \sqrt{(V_{dd} - V_{Tn} - V_{Tp})(V_{dd} + V_{Tn} - V_{Tp} - 2V_i)}$$

From this, we can evaluate $\frac{\partial V_o}{\partial V_i}$ and set it = -1.

$$\frac{\partial V_o}{\partial V_i} = -1 = 1 - \sqrt{\frac{V_{dd} - V_{Tn} - V_{Tp}}{V_{dd} + V_{Tn} - V_{Tp} - 2V_i}} \quad (2.15)$$

This gives

$$V_{iL} = \frac{3V_{dd} + 5V_{Tn} - 3V_{Tp}}{8} \quad (2.16)$$

Substituting this in eq.(2.8), we get

$$\begin{aligned} V_{oH} &= \frac{3V_{DD} + 5V_{Tn} - 3V_{Tp}}{8} + V_{Tp} + \\ &\quad \sqrt{(V_{DD} - V_{Tn} - V_{Tp}) \left(V_{DD} + V_{Tn} - V_{Tp} - \frac{3V_{DD} + 5V_{Tn} - 3V_{Tp}}{4} \right)} \\ &= \frac{3V_{DD} + 5V_{Tn} + 5V_{Tp}}{8} + \sqrt{(V_{DD} - V_{Tn} - V_{Tp}) \left(\frac{V_{DD} - V_{Tn} - V_{Tp}}{4} \right)} \\ &= \frac{3V_{DD} + 5V_{Tn} + 5V_{Tp}}{8} + \frac{V_{DD} - V_{Tn} - V_{Tp}}{2} \\ &= \frac{7V_{DD} + V_{Tn} + V_{Tp}}{8} \end{aligned}$$

So

$$V_{oH} = \frac{7V_{dd} + V_{Tn} + V_{Tp}}{8} = V_{dd} - \frac{V_{dd} - V_{Tn} - V_{Tp}}{8} \quad (2.17)$$

Calculation of V_{iH} and V_{oL}

When the input is ‘high’, we should use eq.(2.14).

$$V_o = (V_i - V_{Tn}) - \sqrt{(V_{dd} - V_{Tn} - V_{Tp})(2V_i - V_{dd} - V_{Tn} + V_{Tp})}$$

Differentiating with respect to V_i gives

$$\frac{\partial V_o}{\partial V_i} = -1 = 1 - \sqrt{\frac{V_{dd} - V_{Tn} - V_{Tp}}{2V_i - V_{dd} - V_{Tn} + V_{Tp}}} \quad (2.18)$$

$$\text{So } \frac{V_{dd} - V_{Tn} - V_{Tp}}{2V_{in} - V_{DD} - V_{Tn} + V_{Tp}} = 4$$

$$\text{Therefore } V_{DD} - V_{Tn} - V_{Tp} = 8V_{in} - 4V_{DD} - 4V_{Tn} + 4V_{Tp}$$

From where, we get

$$V_{iH} = \frac{5V_{dd} + 3V_{Tn} - 5V_{Tp}}{8} \quad (2.19)$$

Substituting the value of V_{iH} for V_{in} in eq.2.14), we get

$$\begin{aligned} V_{oL} &= \frac{5V_{DD} + 3V_{Tn} - 5V_{Tp}}{8} - V_{Tn} - \\ &\quad \sqrt{(V_{DD} - V_{Tn} - V_{Tp}) \left(\frac{5V_{DD} + 3V_{Tn} - 5V_{Tp}}{4} - V_{DD} - V_{Tn} + V_{Tp} \right)} \\ &= \frac{5V_{DD} - 5V_{Tn} - 5V_{Tp}}{8} - \sqrt{(V_{DD} - V_{Tn} - V_{Tp}) \left(\frac{V_{DD} - V_{Tn} - V_{Tp}}{4} \right)} \\ &= \frac{5V_{DD} - 5V_{Tn} - 5V_{Tp}}{8} - \frac{V_{DD} - V_{Tn} - V_{Tp}}{2} = \frac{V_{DD} - V_{Tn} - V_{Tp}}{8} \end{aligned}$$

So

$$V_{oL} = \frac{V_{dd} - V_{Tn} - V_{Tp}}{8} \quad (2.20)$$

Calculation of Noise Margins

The high noise margin is given by

$$V_{oH} - V_{iH} = \frac{V_{dd} - V_{Tn} + 3V_{Tp}}{4} \quad (2.21)$$

Similarly, the Low noise margin is

$$V_{iL} - V_{oL} = \frac{V_{dd} + 3V_{Tn} - V_{Tp}}{4} \quad (2.22)$$

The two noise margins can be made equal by choosing equal values for V_{Tn} and V_{Tp} .

2.2.3 Dynamic Considerations

In this section, we analyze the dynamic behaviour of the inverter. For the calculation of rise and fall times, we shall assume that only one of the two transistors in the inverter is ‘on’. (Notice that this is more conservative than the input high and low conditions determined by slope considerations in eq.2.19 and 2.16). We shall continue to use the simple model described at the beginning of this booklet.

Rise time

When the input is low, the n channel transistor is ‘off’, while the p channel transistor is ‘on’. The equivalent circuit in this condition is shown in fig. 2.7. From Kirchoff’s current law at the output node,

$$I_{dp} = C \frac{dV_o}{dt}$$

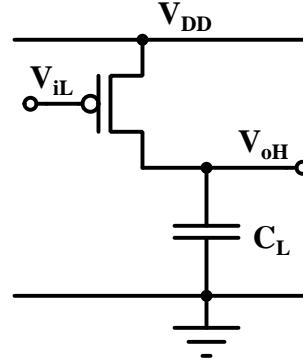


Figure 2.7: CMOS inverter with the nMOS ‘off’

so,

$$\frac{dt}{C} = \frac{dV_o}{I_{dp}}$$

This separates the variables, with the LHS independent of operating voltages and the RHS independent of time. Integrating both sides, we get

$$\frac{\tau_{rise}}{C} = \int_0^{V_{oH}} \frac{dV_o}{I_{dp}}$$

Till the output rises to $V_{iL} + V_{Tp}$, the p channel transistor is in saturation. Since the current is constant, the integration is trivial. If $V_{oH} > V_{iL} + V_{Tp}$ (which is normally the case), the integration range can be broken into saturation and linear regimes. Thus

$$\begin{aligned} \frac{\tau_{rise}}{C} &= \int_0^{V_{iL}+V_{Tp}} \frac{dV_o}{\frac{K_p}{2}(V_{dd} - V_{iL} - V_{Tp})^2} \\ &\quad + \int_{V_{iL}+V_{Tp}}^{V_{oH}} \frac{dV_o}{K_p [(V_{dd} - V_{iL} - V_{Tp})(V_{dd} - V_o) - \frac{1}{2}(V_{dd} - V_o)^2]} \end{aligned}$$

We define $V_1 \equiv V_{dd} - V_o$ and $V_2 \equiv V_{dd} - V_{iL} - V_{Tp}$, so $dV_o = -dV_1$.

We get

$$\frac{K_p \tau_{rise}}{2C} = \frac{V_{iL} + V_{Tp}}{V_2^2} - \int_{V_2}^{V_{dd}-V_{oH}} \frac{dV_1}{2V_1 V_2 - V_1^2}$$

The integral can be evaluated as

$$\begin{aligned} I &\equiv - \int_{V_2}^{V_{dd}-V_{oH}} \frac{dV_1}{2V_1 V_2 - V_1^2} \\ &= \frac{1}{2V_2} \int_{V_{dd}-V_{oH}}^{V_2} \left(\frac{1}{V_1} + \frac{1}{2V_2 - V_1} \right) dV_1 \end{aligned}$$

$$\begin{aligned}
&= \frac{1}{2V_2} \left[\ln \frac{V_1}{2V_2 - V_1} \right]_{V_{dd}-V_{oH}}^{V_2} \\
&= \frac{1}{2V_2} \ln \frac{2V_2 - V_{dd} + V_{oH}}{V_{dd} - V_{oH}}
\end{aligned}$$

Therefore,

$$\frac{K_p \tau_{rise}}{2C} = \frac{V_{iL} + V_{Tp}}{V_2^2} + \frac{1}{2V_2} \ln \frac{2V_2 - V_{dd} + V_{oH}}{V_{dd} - V_{oH}}$$

or

$$\frac{K_p \tau_{rise}}{2C} = \frac{V_{iL} + V_{Tp}}{(V_{dd} - V_{iL} - V_{Tp})^2} + \frac{1}{2(V_{dd} - V_{iL} - V_{Tp})} \ln \frac{2V_2 - V_{dd} + V_{oH}}{V_{dd} - V_{oH}}$$

Thus,

$$\begin{aligned}
\tau_{rise} &= \frac{C(V_{iL} + V_{Tp})}{\frac{K_p}{2}(V_{dd} - V_{iL} - V_{Tp})^2} \\
&+ \frac{C}{K_p(V_{dd} - V_{iL} - V_{Tp})} \ln \frac{V_{dd} + V_{oH} - 2V_{iL} - 2V_{Tp}}{V_{dd} - V_{oH}}
\end{aligned} \tag{2.23}$$

The first term is just the constant current charging of the load capacitor. The second term represents the charging by the pMOS in its linear range. This can be compared with resistive charging, which would have taken a charge time of

$$\tau = RC \ln \frac{V_{dd} - V_{iL} - V_{Tp}}{V_{dd} - V_{oH}}$$

to charge from $V_{iL} + V_{Tp}$ to V_{oH} .

Fall time

When the input is high, the n channel transistor is ‘on’ and the p channel transistor is ‘off’. If the output was initially ‘high’, it will be discharged to ground through the nMOS. To analyse the fall time, we apply Kirchoff’s current law to the output node. This gives

$$I_{dn} = -C \frac{dV_o}{dt}$$

Again, separating variables and integrating from the initial voltage ($= V_{dd}$) to some terminal voltage V_{oL} gives

$$\frac{\tau_{fall}}{C} = - \int_{V_{dd}}^{V_{oL}} \frac{dV_o}{I_{dn}}$$

The n channel transistor will be in saturation till the output voltage falls to $V_i - V_{Tn}$. Below this voltage, the transistor will be in its linear regime. Thus, we can divide the integration

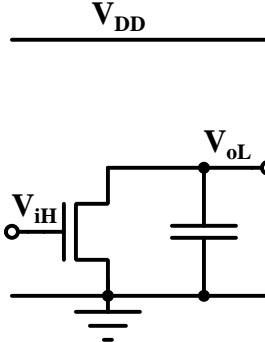


Figure 2.8: CMOS inverter with the pMOS ‘off’

range in two parts.

$$\begin{aligned} \frac{\tau_{fall}}{C} &= - \int_{V_{dd}}^{V_i - V_{Tn}} \frac{dV_o}{I_{dn}} - \int_{V_i - V_{Tn}}^{V_{oL}} \frac{dV_o}{I_{dn}} \\ &= \int_{V_i - V_{Tn}}^{V_{dd}} \frac{dV_o}{\frac{K_n}{2}(V_i - V_{Tn})^2} \\ &\quad + \int_{V_{oL}}^{V_i - V_{Tn}} \frac{dV_o}{K_n [(V_i - V_{Tn})V_o - \frac{1}{2}V_o^2]} \end{aligned}$$

Therefore

$$\begin{aligned} \frac{K_n \tau_{fall}}{2C} &= \frac{V_{dd} - V_i + V_{Tn}}{(V_i - V_{Tn})^2} + \int_{V_{oL}}^{V_i - V_{Tn}} \frac{dV_o}{2V_o(V_i - V_{Tn}) - V_o^2} \\ &= \frac{V_{dd} - V_i + V_{Tn}}{(V_i - V_{Tn})^2} + \frac{1}{2(V_i - V_{Tn})} \int_{V_{oL}}^{V_i - V_{Tn}} dV_o \left(\frac{1}{V_o} + \frac{1}{2(V_i - V_{Tn}) - V_o} \right) \end{aligned}$$

Which gives

$$\begin{aligned} \frac{K_n \tau_{fall}}{2C} &= \frac{V_{dd} - V_i + V_{Tn}}{(V_i - V_{Tn})^2} + \frac{1}{2(V_i - V_{Tn})} \left[\ln \frac{V_o}{2(V_i - V_{Tn}) - V_o} \right]_{V_{oL}}^{V_i - V_{Tn}} \\ &= \frac{V_{dd} - V_i + V_{Tn}}{(V_i - V_{Tn})^2} + \frac{1}{2(V_i - V_{Tn})} \ln \frac{2(V_i - V_{Tn}) - V_{oL}}{V_{oL}} \end{aligned}$$

and therefore

$$\tau_{fall} = \frac{C(V_{dd} - V_i + V_{Tn})}{\frac{K_n}{2}(V_i - V_{Tn})^2} + \frac{C}{K_n(V_i - V_{Tn})} \ln \frac{2(V_i - V_{Tn}) - V_{oL}}{V_{oL}} \quad (2.24)$$

Again, the first term represents the time taken to discharge at constant current in the saturation regime, whereas the second term is the quasi-resistive discharge in the linear regime.

2.2.4 Trade off between power, speed and robustness

As we scale technologies, we improve speed and power consumption. However, as we can see from the expression for noise margins, (eq 2.21 and eq 2.22) the noise margin becomes worse. We can improve noise margins by choosing relatively higher threshold voltages. However, this will reduce speeds. We could also increase V_{dd} - but that would increase power dissipation. Thus we have a trade off between power, speed and noise margins.

This choice is made much more complicated by process variations, because we have to design for the worst case.

2.2.5 CMOS Inverter Design Flow

The CMOS inverter forms the basis of most static CMOS logic design. More complex logic can be designed from it by simple thumb rules. A common (though not universal) design requirement is symmetric charge and discharge behaviour and equal noise margins for high and low logic values. This requires matched values of K_n and K_p and equal values of V_{Tn} and V_{Tp} . For a constant load capacitance, rise and fall times depend linearly on K_n and K_p . Thus it is a straightforward calculation to determine transistor geometries if speed requirements and technological parameters are given. However, as transistor geometries are made larger, self loading can become significant. We now have to model the load capacitance as

$$C_{Load} = C_{ext} + \alpha K_n$$

where we have assumed that $\beta = K_n/K_p$ is kept constant. α is a technological constant. We use the expressions for $K\tau/C$ which depend only on voltages. Once these values are calculated, the geometry can be determined.

In the extreme case, when self capacitance dominates the load capacitance, K/C becomes constant and τ becomes geometry independent. There is no advantage in using wider transistors in this regime to increase the speed. It is better to use multi-stage logic with tapered buffers in this regime. This will be discussed in the module on Logical Effort.

2.2.6 Conversion of CMOS Inverters to other logic

Once the basic CMOS inverter is designed, other logic gates can be derived from it. The logic has to be put in a canonical form which is a sum of products with a bar (inversion) on top. For every ‘.’ in the expression, we put the corresponding n channel transistors in series and the corresponding p channel transistors in parallel. for every ‘+’, we put the n channel transistors in parallel and the p channel transistors in series. We scale the transistor widths up by the number of devices (n or p) put in series. The geometries are left untouched for devices put in parallel. Fig.2.9 shows the implementation of $A \cdot B + C \cdot (D + E)$ in CMOS logic design style.

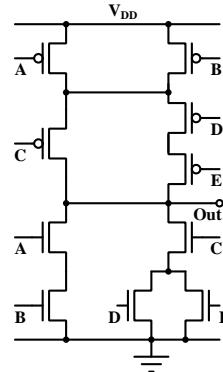


Figure 2.9: CMOS implementation of $\overline{A} \cdot \overline{B} + \overline{C} \cdot (\overline{D} + \overline{E})$

2.2.7 The series parallel rule

While making an equivalent logic gate from an inverter which has been designed to meet given specifications, we try to match the output current of the logic gate (for the worst case combination of inputs to the gate). Geometries of the transistors are derived by using the geometries of the transistors in the inverter as template values.

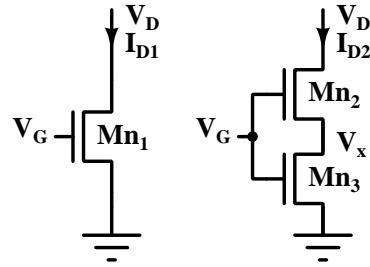
We model each transistor as a switch in series with an ‘on’ resistance. the state of the switch is controlled by the logic input, while the value of the resistance is decided by the geometry of the transistor. In digital design we normally use the minimum channel length for all transistors. Then the equivalent ‘on’ resistance is decided by the width of the transistor. since the current through the transistor is directly proportional to W/L , the equivalent ‘on’ resistance is inversely proportional to its width.

In case of parallel transistors, current may flow through either transistor or through both transistors depending on the logic inputs. Clearly the worst case will be when current flows through only a single transistor. Thus, when using transistors in parallel, the logical input may be such that only one of these is ‘on’. Therefore we use the same width as the template transistor for all transistors in parallel. Now even for the worst case combination of logic inputs, the gate will provide the same drive as the template inverter.

When we use a number of transistors in series, (for a ‘.’ in the logic expression for n transistors and for a ‘+’ for p transistors) we need to scale up the geometries of each series connected transistor, so that they provide the same current drive as the template transistor in the inverter. For current to flow through the series resistance, all of these must be ‘on’. (The ‘on’ resistance is irrelevant if all of these are not ‘on’).

If the ‘on’ resistance of the template transistor in the inverter is R_1 , while that of each of the n series connected transistor is R_2 , we must have $nR_2 = R_1$ or $R_2 = R_1/n$. since the ‘on’ resistance of the transistors is inversely proportional to their width, width of the series connected transistors should be n times the width of the corresponding template transistor.

In fact, we need not depend entirely on this linear resistance analogy for this rule. Let us compare the current through a single template transistor M_{n1} with the current through two identical series connected nMOS transistors. In both cases, the applied gate voltage is such that all transistors are ‘on’.



Let us first determine the mode of operation of these transistors as we sweep the drain voltage V_D from 0 to a voltage much higher than V_G .

- Clearly, M_{n1} is in linear mode till V_D reaches $V_G - V_{Tn}$. Also, M_{n1} will be saturated for $V_D \geq V_G - V_{Tn}$.
- Since both M_{n2} and M_{n3} are on, we must have $V_G - V_x \geq V_{Tn}$. Therefore, $V_x < V_G - V_{Tn}$. But V_x is the drain-source voltage for M_{n3} . So M_{n3} is always in linear mode.
- V_{DS} for M_{n2} is $V_D - V_x$ while its gate-source voltage is $V_G - V_x$. Therefore, it will be saturated whenever

$$V_D - V_x \geq V_G - V_x - V_{Tn} \quad \text{or when} \quad V_D \geq V_G - V_{Tn}$$

. Thus, M_{n2} and M_{n1} are always in the same mode.

Condition	M_{n1}	M_{n2}	M_{n3}
$V_D \leq V_G - V_{Tn}$	Linear	Linear	Linear
$V_D \geq V_G - V_{Tn}$	Saturated	Saturated	Linear

For $0 \leq V_D \leq V_G - V_{Tn}$, all three transistors are in their linear mode. Since M_{n2} and M_{n3} are in series, their currents must be equal. This gives

$$K_n \left((V_G - V_x - V_{Tn})(V_D - V_x) - \frac{1}{2}(V_D - V_x)^2 \right) = K_n \left((V_G - V_{Tn})V_x - \frac{1}{2}V_x^2 \right)$$

We define $V_1 \equiv V_G - V_{Tn}$. Then,

$$(V_1 - V_x)(V_D - V_x) - \frac{1}{2}(V_D - V_x)^2 = V_1 V_x - \frac{1}{2}V_x^2$$

$$\text{Or } V_1 V_D - V_x V_D - V_1 V_x + V_x^2 - \frac{1}{2}(V_D^2 + V_x^2 - 2V_D V_x) = V_1 V_x - \frac{1}{2}V_x^2$$

$$\text{This leads to } V_x^2 - 2V_1 V_x + V_1 V_D - \frac{1}{2}V_D^2 = 0$$

We can solve this quadratic equation to give

$$V_x = \frac{2V_1 \pm \sqrt{4V_1^2 - 4(V_1 V_D - \frac{1}{2}V_D^2)}}{2} = V_1 \pm \sqrt{V_1^2 - V_1 V_D + \frac{1}{2}V_D^2}$$

Since $V_x < V_G - V_{Tn}$, the negative sign must be chosen.

$$\text{Then } V_x = V_1 - \sqrt{V_1^2 - V_1 V_D + \frac{1}{2}V_D^2}$$

$$\text{Or } V_x = V_1 - \sqrt{V_1(V_1 - V_D) + \frac{1}{2}V_D^2}$$

It is interesting to evaluate this at $V_D = V_1 = V_G - V_{Tn}$ when Mn_2 is at the edge of saturation. At this value of V_D , we get

$$V_x = V_1 \left(1 - \frac{1}{\sqrt{2}}\right)$$

For $V_D \geq V_G - V_{Tn}$, Mn_2 is in saturation, while Mn_3 is in linear mode. Since the two are in series, their currents must be equal. Therefore,

$$\frac{K_n}{2} (V_G - V_x - V_{Tn})^2 = K_n \left((V_G - V_{Tn})V_x - \frac{1}{2}V_x^2\right)$$

$$\text{This gives } (V_G - V_x - V_{Tn})^2 = 2 \left((V_G - V_{Tn})V_x - \frac{1}{2}V_x^2\right)$$

$$\text{Defining } V_1 \equiv V_G - V_{Tn}, \text{ we get } (V_1 - V_x)^2 = 2V_1 V_x - V_x^2$$

$$\text{Therefore } V_1^2 + V_x^2 - 2V_1 V_x = 2V_1 V_x - V_x^2 \quad \text{So, } 2V_x^2 - 4V_1 V_x + V_1^2 = 0$$

$$\text{This can be solved to give } V_x = \frac{4V_1 \pm \sqrt{16V_1^2 - 8V_1^2}}{4} = V_1 \pm \sqrt{\frac{V_1^2}{2}}$$

Again, since V_x must be $< V_G - V_{Tn}$, the negative sign should be chosen.

$$\text{So } V_x = V_1 \left(1 - \frac{1}{\sqrt{2}}\right) = \left(1 - \frac{1}{\sqrt{2}}\right) (V_G - V_{Tn})$$

Thus the voltage V_x remains constant at $(1 - 1/\sqrt{2})(V_G - V_{Tn})$ for $V_D \geq V_G - V_{Tn}$. This value matches with the solution for linear mode at the edge of saturation, as indeed it should.

For $V_D \leq V_G - V_{Tn}$, Mn_1 , Mn_2 as well as Mn_3 are in linear mode. So,

$$I_{D1} = K_n(V_1V_D - \frac{1}{2}V_D^2)$$

I_{D2} can be computed as the current through Mn_3 .

$$I_{D2} = K_n(V_1V_x - \frac{1}{2}V_x^2)$$

The quadratic equation for V_x was

$$V_x^2 - 2V_1V_x + V_1V_D - \frac{1}{2}V_D^2 = 0$$

$$\text{Therefore } V_1V_x - \frac{1}{2}V_x^2 = \frac{1}{2} \left(V_1V_D - \frac{1}{2}V_D^2 \right)$$

So, current through Mn_2 and Mn_3 is

$$I_{D2} = K_n(V_1V_x - \frac{1}{2}V_x^2) = \frac{K_n}{2} \left(V_1V_D - \frac{1}{2}V_D^2 \right)$$

The current through Mn_1 in linear mode is

$$I_{D1} = K_n \left(V_1V_D - \frac{1}{2}V_D^2 \right)$$

Thus the current through the series connected transistors Mn_2 and Mn_3 is half of that thorough the single transistor Mn_1 with identical geometry.

This can be shown in another way: Since the currents through Mn_2 and Mn_3 must be equal, the sum of their currents must equal $2I_{D2}$.

$$2I_{D2} = K_n \left((V_1 - V_x)(V_D - V_x) - \frac{1}{2}(V_D - V_x)^2 \right) + K_n \left((V_1V_x - \frac{1}{2}V_x^2) \right)$$

$$\text{Therefore } \frac{2I_{D2}}{K_n} = (V_1 - V_x)(V_D - V_x) - \frac{1}{2}(V_D - V_x)^2 + V_1V_x - \frac{1}{2}V_x^2$$

$$\text{So } \frac{2I_{D2}}{K_n} = V_1V_D - V_xV_D - V_xV_1 + V_x^2 - \frac{1}{2}(V_D^2 + V_x^2 - 2V_DV_x) + V_1V_x - \frac{1}{2}V_x^2$$

All terms involving V_x cancel and we are left with

$$\frac{2I_{D2}}{K_n} = V_1V_D - \frac{1}{2}V_D^2 \quad \text{or} \quad I_{D2} = \frac{K_n}{2} \left(V_1V_D - \frac{1}{2}V_D^2 \right)$$

This establishes that for $V_D \leq V_G - V_{Tn}$, the current through series connected Mn_2 and Mn_3 is half of that through Mn_1 .

For $V_D \geq V_G - V_{Tn}$, Mn_1 and Mn_2 are saturated, while Mn_3 is in linear mode. Under these conditions,

$$I_{D1} = \frac{K_n}{2} V_1^2$$

Since Mn_2 is saturated, $V_x = V_1 \left(1 - \frac{1}{\sqrt{2}}\right)$ and so, $V_1 - V_x = V_1/\sqrt{2}$

Using this, current through Mn_2 can be calculated.

$$I_{D2} = \frac{K_n}{2} (V_1 - V_x)^2 = \frac{K_n}{2} \frac{V_1^2}{2} \quad \text{whereas} \quad I_{D1} = \frac{K_n}{2} V_1^2$$

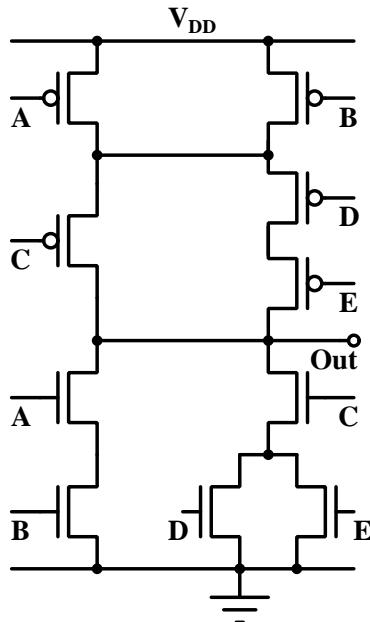
So, in saturation also, the current through Mn_2 and Mn_3 is half as much as that through Mn_1 .

Thus, at all operating voltages, current through series connected Mn_2 and Mn_3 is half as much as that through Mn_1 .

(This establishes the validity of the series rule even for non-linear I-V relationships).

2.2.8 Application of Series Parallel rule to complex logic

Let us see how to apply the series parallel rule when parallel combinations are in series or when series combinations of transistors are in parallel. Let us take the example implementation of $\overline{A.(B + C)} + D.E$ to illustrate successive application of series parallel rules. The circuit is replicated here for easy reference.



n channel transistors driven by A and B are in series. So each should have a width which is twice that of the template nMOS in the inverter. The circuit should provide the same discharge current, even when $A = B = '0'$, $C = '1'$ and only one of D and E is on. This condition can be met when all three transistors C, D, and E have twice the width of nMOS in the inverter.

fixing the geometries of pMOS transistors is more complex. In the case when $A=C='1'$, $B=D=E='0'$, the series connected transistors B,D and E have to pull the output node to '1'. Thus all three transistors should have one third the resistance of the template pMOS transistor in the inverter – and hence, thrice the width. Since C has to be equivalent to D and E in series, it should have a width = 1.5 times the width of the template pMOS transistor.

Finally, A and B should be equivalent, so A should also be thrice as wide as the template transistor. We can verify that in the case when $A=C='0'$, $B=D=1$; the equivalent drive is the same as that of the template inverter. This is because A has one third the resistance of the template transistor, while C has a resistance which is two thirds of the resistance of the template transistor.

There can be other choices which meet the requirements of equivalent drive. In such cases we choose the combination which provides the lowest overall capacitance or area. For example, in this case, we could have started with A and C and made them 2X. B being equivalent to A, should also be 2X. Since D and E are in series, these should be 4X. With this choice also, we can meet the condition of equivalent drive. However, the total capacitance in this case is $(2 + 2 + 2 + 4 + 4 = 14)$ times the capacitance of the template transistor. In the previous choice, the capacitance was $(3 + 3 + 3 + 3 + 1.5 = 13.5)$ times the capacitance of the template transistor.

Chapter 3

Pseudo-nMOS Logic Design

CMOS design style ensures that the logic consumes no static power. This is because the pull down and pull up networks are never ‘on’ simultaneously. However, this requires that signals have to be routed to the n pull down network *as well as* to the p pull up network. This means that the load presented to every driver is high. This fact is exacerbated by the fact that n and p channel transistors cannot be placed close together as these are in different wells which have to be kept well separated in order to avoid latch-up.

Pseudo nMOS design style reduces dynamic power (by reducing capacitive loading) at the cost of having non-zero static power by replacing the pull up network by a single pMOS transistor with its gate terminal grounded. The pseudo nMOS inverter is shown below. Notice

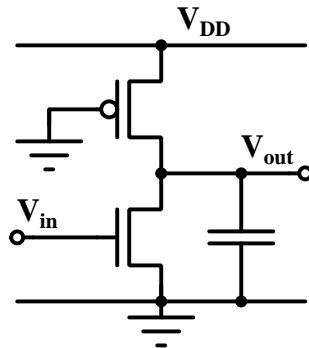


Figure 3.1: Pseudo nMOS inverter

that since the pMOS is not driven by signals, it is always ‘on’. The effective gate voltage seen by the pMOS transistor is V_{dd} . Thus the over-voltage on the p channel gate is always $V_{dd} - V_{Tp}$. This transistor will be saturated for output voltage $\leq V_{Tp}$ and will be in the linear regime for output voltage $\geq V_{Tp}$.

When the nMOS is turned ‘on’, a direct current path exists between supply and ground and so, static power will be dissipated.

3.1 Static Characteristics

As we sweep the input voltage from ground to V_{dd} , we encounter the following regimes of operation:

- For $V_i \leq V_{Tn}$, nMOS is ‘off’.
- For low input voltage ($> V_{Tn}$), nMOS is saturated, pMOS is in linear regime.
- As V_i is raised further, V_o continues to fall. Eventually we enter the regime where nMOS is linear and pMOS is also linear.
- Finally, as we keep raising V_i , the output falls below V_{Tp} , provided the nMOS transistor is sufficiently wide. Now the nMOS is in linear regime, while pMOS is saturated.

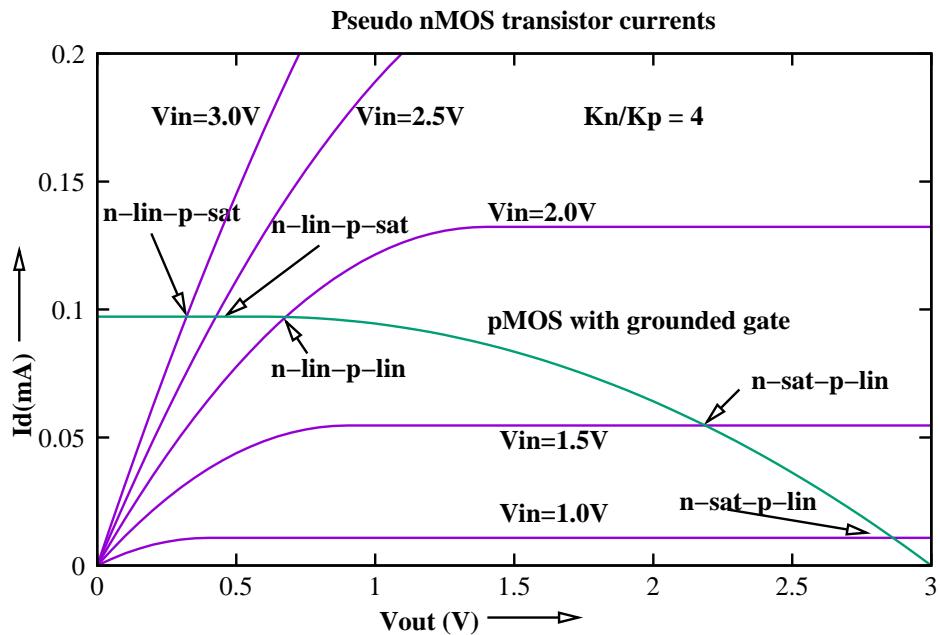


Figure 3.2: Transistor currents in a Pseudo nMOS inverter

3.1.1 For $0 \leq V_i \leq V_{Tn}$: nMOS ‘off’

When the input voltage is less than V_{Tn} , the n channel transistor is off. Since no current flows through the p channel transistor, its drain-source voltage must be zero. Thus the output is ‘high’ (and = V_{dd}). No static power is dissipated in this condition.

3.1.2 nMOS saturated, pMOS linear

As the input voltage is raised above V_{Tn} , we enter this region. At this stage, the output voltage is close to V_{dd} (so more than $V_i - V_{Tn}$). Therefore the n channel transistor will be in saturation. The output voltage is also much higher than V_{Tp} , so the p channel transistor is in linear mode of operation. Equating currents through the n and p channel transistors, we get

$$\frac{K_n}{2}(V_i - V_{Tn})^2 = K_p \left[(V_{dd} - V_{Tp})(V_{dd} - V_o) - \frac{1}{2}(V_{dd} - V_o)^2 \right] \quad (3.1)$$

defining $\beta \equiv K_n/K_p$, $V_1 \equiv V_{dd} - V_o$ and $V_2 \equiv V_{dd} - V_{Tp}$, we get

$$\frac{1}{2}V_1^2 - V_2V_1 + \frac{\beta}{2}(V_i - V_{Tn})^2 = 0 \quad (3.2)$$

with solutions

$$V_1 = V_2 \pm \sqrt{V_2^2 - \beta(V_i - V_{Tn})^2}$$

substituting the values of V_1 and V_2 and choosing the sign which puts V_o in the correct range, we get

$$V_o = V_{Tp} + \sqrt{(V_{dd} - V_{Tp})^2 - \beta(V_i - V_{Tn})^2} \quad (3.3)$$

This expression is valid as long as the n channel transistor is in saturation and the p channel transistor is in linear regime. As we keep increasing the input voltage V_i , the output will keep falling. Eq.3.3 will no more be valid when either the output falls below V_{Tp} , so that the p channel transistor enters saturation, or it falls below $V_i - V_{Tn}$, so that the n channel transistor enters the linear regime.

The output will fall below V_{Tp} when

$$V_{dd} - V_{Tp} = \sqrt{\beta}(V_i - V_{Tn}) \quad \text{or} \quad V_i = V_{Tn} + \frac{V_{dd} - V_{Tp}}{\sqrt{\beta}}$$

On the other hand it will fall to $V_i - V_{Tn}$ when

$$V_i - V_{Tn} = V_{Tp} + \sqrt{(V_{dd} - V_{Tp})^2 - \beta(V_i - V_{Tn})^2}$$

This can be solved to show that nMOS will enter its linear regime when

$$V_i > V_{Tn} + \frac{V_{Tp} + \sqrt{V_{Tp}^2 + (\beta + 1)V_{dd}(V_{dd} - 2V_{Tp})}}{\beta + 1}. \quad (3.4)$$

For common values of parameters, this happens before pMOS enters saturation.

3.1.3 nMOS linear, pMOS linear

$$\text{for } V_i > V_{Tn} + \frac{V_{Tp} + \sqrt{V_{Tp}^2 + (\beta + 1)V_{dd}(V_{dd} - 2V_{Tp})}}{\beta + 1}.$$

the nMOS enters its linear mode of operation. At this point the output voltage is $> V_{Tp}$, so the pMOS transistor is also in its linear regime. This combination of nMOS as well as pMOS being in linear regime will continue as we increase V_i , till V_o falls to V_{Tp} .

To determine the output voltage when both transistors are in their linear regimes, We can again equate the nMOS and pMOS currents using the transistor current equations for linear regime. The solution is straightforward, though algebraically tedious.

Equating n and p channel transistor currents,

$$K_n \left[(V_i - V_{Tn})V_o - \frac{1}{2}V_o^2 \right] = K_p \left[(V_{dd} - V_{Tp})(V_{dd} - V_o) - \frac{1}{2}(V_{dd} - V_o)^2 \right] \quad (3.5)$$

Defining $\beta \equiv K_n/K_p$, we can write

$$\begin{aligned} \beta \left[(V_i - V_{Tn})V_o - \frac{1}{2}V_o^2 \right] &= V_{dd}^2 - V_{Tp}V_{dd} - V_oV_{dd} + V_oV_{Tp} - \frac{1}{2}(V_{dd}^2 + V_o^2 - 2V_{dd}V_o) \\ \text{or } \beta(V_i - V_{Tn})V_o - \frac{\beta}{2}V_o^2 &= \frac{1}{2}V_{dd}^2 - V_{Tp}V_{dd} + V_{Tp}V_o - \frac{1}{2}V_o^2 \end{aligned}$$

This gives

$$\frac{\beta - 1}{2}V_o^2 - (\beta(V_i - V_{Tn}) - V_{Tp})V_o + \frac{V_{dd}}{2}(V_{dd} - 2V_{Tp}) = 0 \quad (3.6)$$

Solving this quadratic will give the value of V_o .

$$V_o = \frac{\beta(V_i - V_{Tn}) - V_{Tp} \pm \sqrt{(\beta(V_i - V_{Tn}) - V_{Tp})^2 - (\beta - 1)V_{dd}(V_{dd} - 2V_{Tp})}}{\beta - 1}$$

To keep V_o in the correct operating range, and because V_o is a decreasing function of V_i , we choose the negative sign. Thus,

$$V_o = \frac{\beta(V_i - V_{Tn}) - V_{Tp} - \sqrt{(\beta(V_i - V_{Tn}) - V_{Tp})^2 - (\beta - 1)V_{dd}(V_{dd} - 2V_{Tp})}}{\beta - 1} \quad (3.7)$$

3.1.4 nMOS linear, pMOS saturated

As the input voltage is raised still further, the output voltage will reach and then fall below V_{Tp} . The nMOS continues in its linear regime, and now the pMOS transistor enters its saturation regime.

Equating currents, we get

$$K_n \left[(V_i - V_{Tn})V_o - \frac{1}{2}V_o^2 \right] = \frac{K_p}{2}(V_{dd} - V_{Tp})^2$$

which gives

$$\frac{1}{2}V_o^2 - (V_o - V_{Tn})V_o + \frac{(V_{dd} - V_{Tp})^2}{2\beta}$$

This can be solved to get

$$V_o = (V_i - V_{Tn}) - \sqrt{(V_i - V_{Tn})^2 - (V_{dd} - V_{Tp})^2/\beta} \quad (3.8)$$

3.2 Noise margins

As in the case of CMOS inverter, we find points on the transfer curve where the slope is -1.

When the input is low and output high, we should use Eq(3.3). Differentiating this equation with respect to V_i and setting the slope to -1, we get

$$V_{iL} = V_{Tn} + \frac{V_{dd} - V_{Tp}}{\sqrt{\beta(\beta + 1)}} \quad (3.9)$$

and

$$V_{oH} = V_{Tp} + \sqrt{\frac{\beta}{\beta + 1}} (V_{dd} - V_{Tp}) \quad (3.10)$$

When the input is high and the output low, we use Eq(3.8). Again, differentiating with respect to V_i and setting the slope to -1, we get

$$V_{iH} = V_{Tn} + \frac{2}{\sqrt{3\beta}} (V_{dd} - V_{Tp}) \quad (3.11)$$

and

$$V_{oL} = \frac{(V_{dd} - V_{Tp})}{\sqrt{3\beta}} \quad (3.12)$$

To make the output ‘low’ value lower than V_{Tn} , we get the condition

$$\beta > \frac{1}{3} \left(\frac{V_{dd} - V_{Tp}}{V_{Tn}} \right)^2$$

This condition on values of β places a requirement on the ratios of widths of n and p channel transistors. The logic gates work properly only when this equation is satisfied. Therefore this kind of logic is also called ‘ratioed logic’. In contrast, CMOS logic is called ratioless logic because it does not place any restriction on the ratios of widths of n and p channel transistors for static operation. The noise margin for pseudo nMOS can be determined easily from the expressions for V_{iL} , V_{oL} , V_{iH} , V_{oH} .

3.3 Dynamic characteristics

In the sections above, we have derived the behaviour of a pseudo nMOS inverter in static conditions. In the sections below, we discuss the dynamic behaviour of this inverter.

3.3.1 Rise Time

When the input is low and the output rises from ‘low’ to ‘high’, the nMOS is off. The situation is identical to the charge up condition of a CMOS gate with the pMOS being biased with its gate at 0V. This gives

$$\tau_{rise} = \frac{C}{K_p(V_{dd} - V_{Tp})} \left[\frac{2V_{Tp}}{V_{dd} - V_{Tp}} + \ln \frac{V_{dd} + V_{oH} - 2V_{Tp}}{V_{dd} - V_{oH}} \right] \quad (3.13)$$

3.3.2 Fall Time

Analytical calculation of fall time is complicated by the fact that the pMOS load continues to dump current in the output node, even as the nMOS tries to discharge the output capacitor.

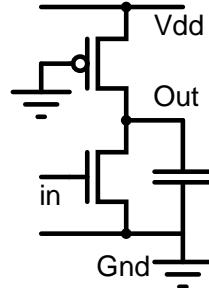


Figure 3.3: ‘high’ to ‘low’ transition on the output

Thus the nMOS should sink the discharge current as well as the drain current of the pMOS transistor. We make the simplifying assumption that the pMOS current remains constant at its saturation value through the entire discharge process. (This will result in a slightly pessimistic value of discharge time). Then,

$$I_p = \frac{K_p}{2}(V_{dd} - V_{Tp})^2$$

. We can write the KCL equation at the output node as:

$$I_n - I_p + C \frac{dV_o}{dt} = 0$$

which gives

$$\frac{\tau_{fall}}{C} = - \int_{V_{dd}}^{V_{oL}} \frac{dV_o}{I_n - I_p}$$

We define $V_1 \equiv V_i - V_{Tn}$ and $V_2 \equiv V_{dd} - V_{Tp}$. The integration range can be divided into two regimes. nMOS is saturated when $V_1 \leq V_o < V_{dd}$ and is in linear regime when $V_{oL} < V_o < V_1$. Therefore,

$$\frac{\tau_{fall}}{C} = - \int_{V_{dd}}^{V_1} \frac{dV_o}{\frac{1}{2}K_n V_1^2 - I_p} - \int_{V_1}^{V_{oL}} \frac{dV_o}{K_n(V_1 V_o - \frac{1}{2}V_o^2) - I_p}$$

$$\text{So } \frac{K_n \tau_{fall}}{2C} = \frac{V_{dd} - V_1}{V_1^2 - 2I_p/K_n} + \int_{V_{oL}}^{V_1} \frac{dV_o}{2V_1 V_o - V_o^2 - 2I_p/K_n}$$

We define $V_2^2 \equiv 2I_p/K_n$. The denominator of the integral term can then be factorized by adding and subtracting V_1^2

$$\begin{aligned} 2V_1 V_o - V_o^2 - V_1^2 + V_1^2 - V_2^2 &= -(V_1 - V_o)^2 + \left(\sqrt{V_1^2 - V_2^2} \right)^2 \\ &= \left(\sqrt{V_1^2 - V_2^2} + V_1 - V_o \right) \left(\sqrt{V_1^2 - V_2^2} - V_1 + V_o \right) \end{aligned}$$

The integral term can then be written as:

$$\frac{1}{2\sqrt{V_1^2 - V_2^2}} \left(\int_{V_{oL}}^{V_1} \frac{dV_o}{\sqrt{V_1^2 - V_2^2} + V_1 - V_o} + \int_{V_{oL}}^{V_1} \frac{dV_o}{\sqrt{V_1^2 - V_2^2} - (V_1 - V_o)} \right)$$

The integration over V_o can now be carried out to get

$$\frac{1}{2\sqrt{V_1^2 - V_2^2}} \ln \frac{\sqrt{V_1^2 - V_2^2} - (V_1 - V_o)}{\sqrt{V_1^2 - V_2^2} + V_1 - V_o} \Big|_{V_{oL}}^{V_1}$$

On putting the limits for the integral, the upper limit gives 0. So the integral can be written as

$$\frac{1}{2\sqrt{V_1^2 - V_2^2}} \ln \frac{\sqrt{V_1^2 - V_2^2} + V_1 - V_{oL}}{\sqrt{V_1^2 - V_2^2} - (V_1 - V_{oL})}$$

Thus we can write down the expression for fall time as:

$$\frac{K_n \tau_{fall}}{2C} = \frac{V_{dd} - V_1}{V_1^2 - 2I_p/K_n} + \frac{1}{2\sqrt{V_1^2 - V_2^2}} \ln \frac{\sqrt{V_1^2 - V_2^2} + V_1 - V_{oL}}{\sqrt{V_1^2 - V_2^2} - (V_1 - V_{oL})}$$

Substituting back for V_2^2 , we get

$$\frac{K_n \tau_{fall}}{2C} = \frac{V_{dd} - V_1}{V_1^2 - 2I_p/K_n} + \frac{1}{2\sqrt{V_1^2 - 2I_p/K_n}} \ln \frac{\sqrt{V_1^2 - 2I_p/K_n} + V_1 - V_{oL}}{\sqrt{V_1^2 - 2I_p/K_n} - (V_1 - V_{oL})}$$

Since $I_p = K_p(V_{dd} - V_{Tp})^2/2$, $2I_p/K_n$ is just $(V_{dd} - V_{Tp})^2/\beta$.

$$\frac{K_n \tau_{fall}}{2C} = \frac{V_{dd} - V_1}{V_1^2 - 2I_p/K_n} + \frac{1}{2\sqrt{V_1^2 - 2I_p/K_n}} \ln \frac{\sqrt{V_1^2 - 2I_p/K_n} + V_1 - V_{oL}}{\sqrt{V_1^2 - 2I_p/K_n} - (V_1 - V_{oL})} \quad (3.14)$$

$$\text{Where } 2I_p/K_n = \frac{(V_{dd} - V_{Tp})^2}{\beta} \quad \text{and} \quad V_1 = V_i - V_{Tn}$$

This relation was derived using the pessimistic assumption that the p channel transistor dumps its saturation current over the entire discharge range.

In any case, this relation is not used for designing the inverter because the limit on the size of the n channel transistor is put by static considerations and not by the fall time.

3.4 Pseudo nMOS design Flow

We design the basic inverter first and then map the inverter design to other logic circuits. The load device size is calculated from the rise time. From Eq. 3.13 we have

$$\tau_{rise} = \frac{C}{K_p(V_{dd} - V_{Tp})} \left[\frac{2V_{Tp}}{V_{dd} - V_{Tp}} + \ln \frac{V_{dd} + V_{oH} - 2V_{Tp}}{V_{dd} - V_{oH}} \right]$$

Given a value of τ_{rise} , operating voltages and technological constants, K_p and hence, the geometry of the p channel transistor can be determined.

Geometry of the n channel transistor in the reference inverter design can be determined from static considerations. Using Eq. 3.8, the output ‘low’ level is given by:

$$V_o = (V_i - V_{Tn}) - \sqrt{(V_i - V_{Tn})^2 - (V_{dd} - V_{Tp})^2/\beta}$$

If the desired value of the output ‘low’ level is given, we can calculate β . But $\beta \equiv K_n/K_p$ and K_p is already known. This evaluates K_n and hence, the geometry of the n channel transistor.

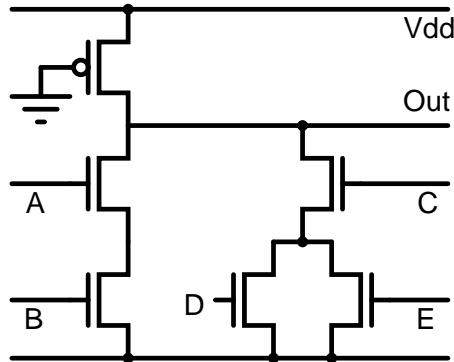


Figure 3.4: Pseudo NMOS implementation of $\overline{A.B + C.(D + E)}$

3.5 Conversion of pseudo nMOS Inverter to other logic

Once the basic pseudo nMOS inverter is designed, other logic gates can be derived from it. The procedure is the same as that for CMOS, except that it is applied only to nMOS transistors. The p channel transistor is kept at the same size as that for an inverter.

The logic is expressed as a sum of products with a bar (inversion) on top. For every ‘.’ in the expression, we put the corresponding n channel transistors in series and for every ‘+’, we put the n channel transistors in parallel. We scale the transistor widths up by the number of devices put in series. The geometries are left untouched for devices put in parallel. Fig.3.4 shows the implementation of $\overline{A.B + C.(D + E)}$ in pseudo NMOS logic design style.

Chapter 4

Dual Rail Logic Design

4.1 Complementary Pass gate Logic

This logic family is based on multiplexer logic and makes use of Shannon's Boolean expansion theorem.

Given a Boolean function $F(x_1, x_2, \dots, x_n)$, we can express it as:

$$F(x_1, x_2, \dots, x_n) = x_i \cdot f_1 + \bar{x}_i \cdot f_2$$

where f_1 and f_2 are reduced expressions for F with x_i forced to '1' and '0' respectively. Thus, F can be implemented with a multiplexer controlled by x_i which selects f_1 or f_2 depending on x_i . f_1 and f_2 can themselves be decomposed into simpler expressions by the same technique.

To implement a multiplexer, we need both x_i and \bar{x}_i . Therefore, this logic family needs all inputs in true as well as in complement form. In order to drive other gates of the same type, it must produce the outputs also in true and complement forms. Thus each signal is carried by two wires. This logic style is called "Complementary Pass-gate Logic" or CPL for short.

4.1.1 Basic Multiplexer Structure

Pure pass-gate logic contains no 'amplifying' elements. Therefore, it has zero or negative noise margin. (Each logic stage degrades the logic level). Therefore, multiple logic stages cannot be cascaded. We shall assume that each stage includes conventional CMOS inverters to restore the logic level. Ideally, the multiplexer should be composed of complementary pass gate transistors. However, we shall use just n channel transistors as switches for simplicity. This gives us the multiplexer structure shown in fig.4.1.

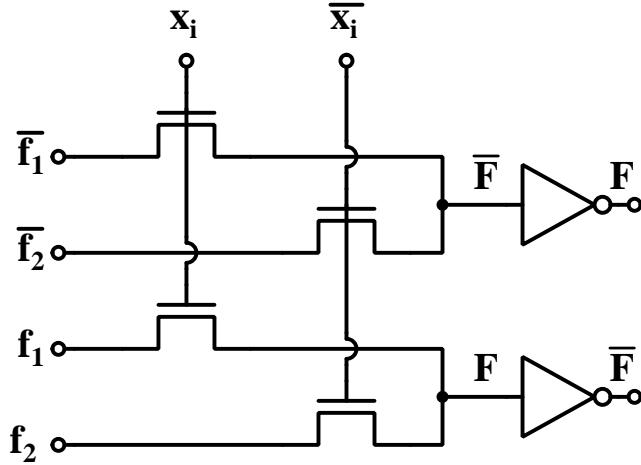


Figure 4.1: Basic Multiplexer with logic restoring inverters

4.1.2 Logic Design using CPL

Since both true and complement outputs are generated by CPL, we do not need separate gates for AND and NAND functions. The same applies to OR-NOR, and XOR-XNOR functions.

To take an example, let us consider the XOR-XNOR functions. Because of the inverter,

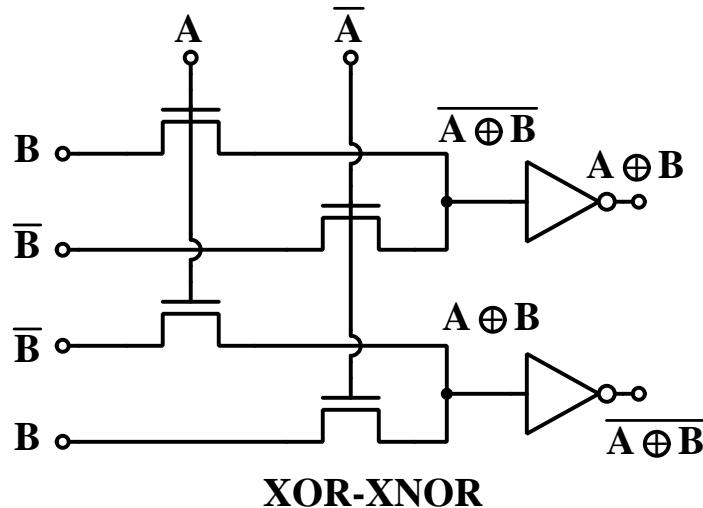
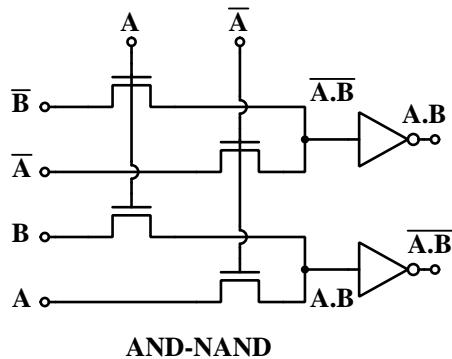


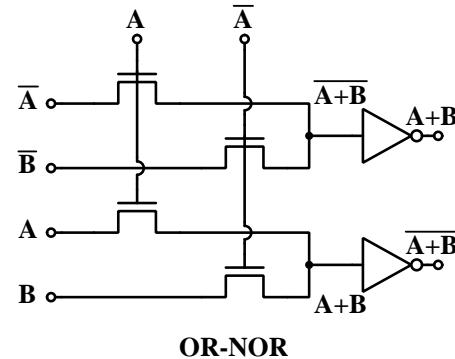
Figure 4.2: Implementation of XOR and XNOR by CPL logic.

the multiplexer for the XOR output first calculates the XNOR function given by $A.B + \bar{A}.\bar{B}$.

If we put $A = '1'$, this reduces to B and for $A = '0'$, it reduces to \bar{B} . Similarly, for the XNOR output, we generate the XOR expression $= A.\bar{B} + \bar{A}.B$ which will be inverted by the logic level restoring inverter. The expression reduces to \bar{B} for $A = '1'$ and to B for $A = '0'$. This leads to an implementation of XOR-XNOR as shown in fig.4.2



AND-NAND



OR-NOR

Figure 4.3: Implementation of (a) AND-NAND and (b) OR-NOR functions using complementary pass-gate logic.

Implementation of AND and OR functions is similar. In case of AND, the multiplexer should output $\bar{A}.B$ to be inverted by the buffer. This reduces to \bar{B} when $A = '1'$. When $A = '0'$, it evaluates to $1 = \bar{A}$. For NAND output, the multiplexer should output $A.B$, which evaluates to B for $A = '1'$ and to $'0'$ (or A) when $A = '0'$.

4.1.3 Buffer Leakage Current

The circuit configuration described above uses nMOS multiplexers. This limits the ‘high’

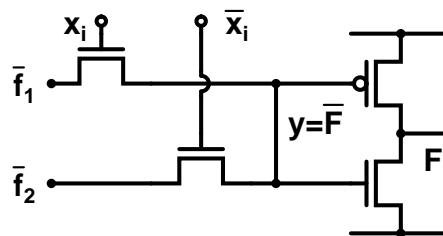


Figure 4.4: High leakage current in inverter

output of the multiplexer (node y - which is the input for the inverter) to $V_{dd} - V_{Tn}$. Consequently, the pMOS transistor in the buffer inverter never quite turns off. This results in static power consumption in the inverter. This can be avoided by adding a pull up pMOS as shown

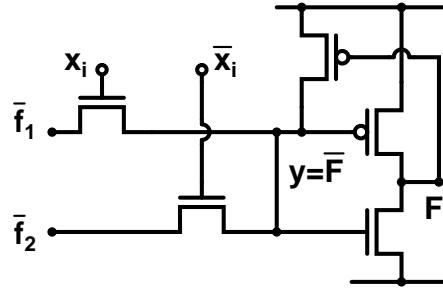


Figure 4.5: Pull up pMOS to avoid leakage in the inverter

in fig. 4.5. When the multiplexer output (y) is ‘low’, the inverter output is high. The pMOS is therefore off and has no effect. When the multiplexer output goes ‘high’, the inverter input charges up, the output starts falling and turns the pMOS on. Now, as the multiplexer output (y) approaches $V_{dd} - V_{Tn}$, the nMOS switch in the multiplexer turn off. However, the pMOS pull up remains ‘on’ and takes the inverter input all the way to V_{dd} . This avoids leakage in the inverter.

However, this solution brings up another problem. Consider the equivalent circuit when the inverter output is ‘low’ and the pMOS is ‘on’. Now if the multiplexer output wants to go

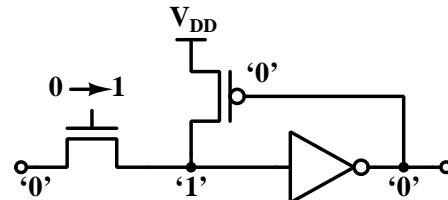


Figure 4.6: Problem with a low to high transition on the output

‘low’, it has to fight the pMOS pull-up - which is trying to keep this node ‘high’.

In fact, the multiplexer n transistor and the pull up p transistor constitute a pseudo nMOS inverter. Therefore, the multiplexer output cannot be pulled low unless the transistor geometries are appropriately ratioed.

4.2 Cascade Voltage Switch Logic

We can understand this logic configuration as an attempt to improve pseudo-nMOS logic circuits. Consider the NOR gate shown below: Static power is consumed by this NOR circuit

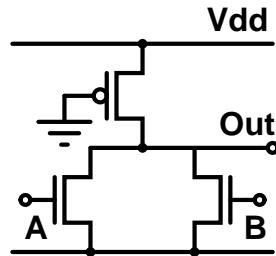


Figure 4.7: Pseudo-nMOS NOR

whenever the output is ‘LOW’. This happens when $A \text{ OR } B$ is TRUE. We wish that the pMOS could be turned off for just this combination of inputs.

To turn the pMOS transistor off, we need to apply a ‘HIGH’ voltage level to its gate whenever $A \text{ OR } B$ is true. This obviously requires an OR gate. Non-inverting gates cannot be made in a single stage. However, We can create the OR function by using a NAND of \bar{A} and \bar{B} as shown in figure 4.8. But then what about the pMOS drive of this circuit?

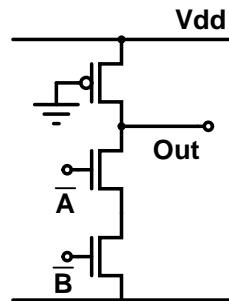


Figure 4.8: Pseudo-nMOS OR from complemented inputs

We want to turn the pMOS of this OR circuit off when both \bar{A} and \bar{B} are ‘HIGH’; i.e. when $A = B = 0$. This means we would like to turn the pMOS of this circuit off when the NOR of A and B is ‘TRUE’.

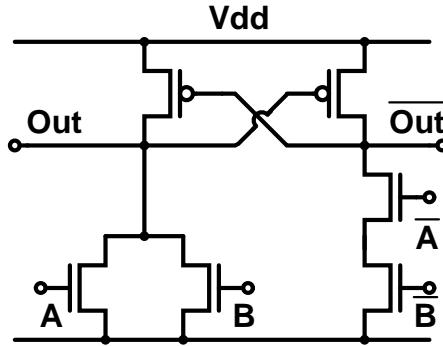


Figure 4.9: OR-NOR implementation in Cascade Voltage Switch Logic

But we already have this signal as the output of the first (NOR) circuit! So the two circuits can drive each other's pMOS transistors and avoid static power consumption. This kind of logic is called Cascade Voltage Switch Logic (CVSL). It can use any network f and its complementary network \bar{f} in the two cross-coupled branches. The complementary network is constructed by changing all series connections in f to parallel and all parallel connections to series, and complementing all input signals.

CVSL shares many characteristics with static CMOS, CPL and pseudo-nMOS.

- Like CMOS static logic, there is no static power consumption.
- Like CPL, this logic requires both True and Complement signals. It also provides both True and complement outputs. (Dual Rail Logic).
- Like pseudo nMOS, the inputs present a single transistor load to the driving stage.
- The circuit is self latching. This reduces ratioing requirements.

Chapter 5

Dynamic Logic

In this style of logic, some nodes are required to hold their logic value as a charge stored on a capacitor. These nodes are not connected to their ‘drivers’ permanently. The ‘driver’ places the logic value on them, and is then disconnected from the node. Due to leakage etc., the logic value cannot be held indefinitely. Dynamic circuits therefore require a *minimum* clock frequency to operate correctly. Use of dynamic circuits can reduce circuit complexity and power consumption substantially, compared to pseudo NMOS.

5.1 Basic CMOS Dynamic Gate

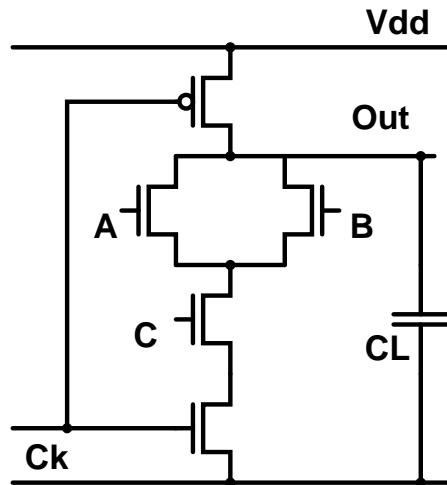


Figure 5.1: CMOS dynamic gate to implement $\overline{(A + B) \cdot C}$.

When the clock is low, pMOS is on and the bottom nMOS is off. The output is ‘pre-

charged' to '1' unconditionally. When the clock goes high, the pMOS turns off and the bottom nMOS comes on. The circuit then conditionally discharges the output node, if $(A+B).C$ is TRUE. This implements the function $\overline{(A+B).C}$.

5.1.1 Problem with Cascading CMOS dynamic logic

The dynamic circuit described above can run into problems when several dynamic gates are cascaded. Consider the case when the circuit described in Fig. 5.1 is followed by an inverter. Let us take two cases – when $(A+B).C$ is FALSE and when it is TRUE.

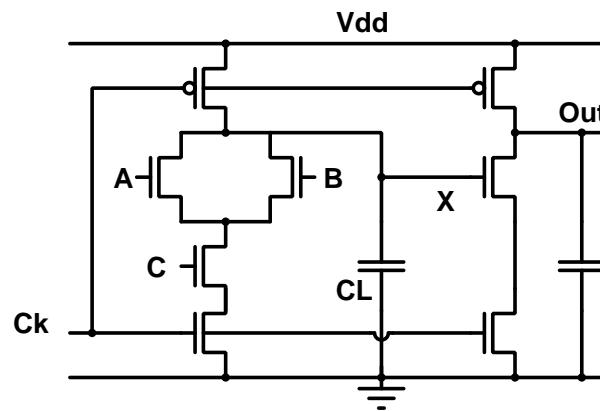
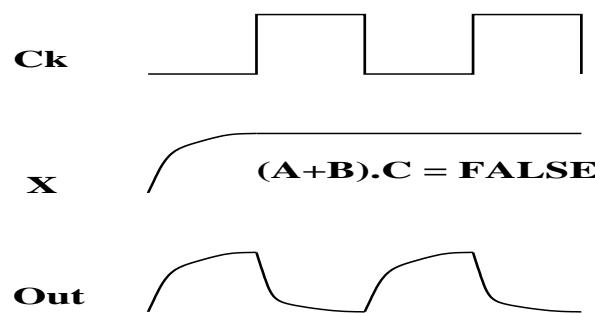
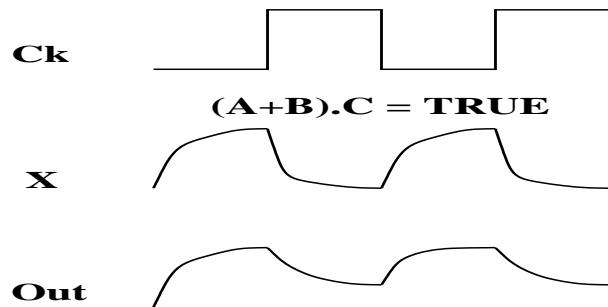


Figure 5.2: Dynamic gate for $\overline{(A+B).C}$ cascaded with an inverter.

When $(A+B).C$ is FALSE, There is no problem X pre-charges to '1' and remains at '1'. Therefore the inverter sees the correct logic value at its input all the time and discharges the output to '0', which is the expected output.



However, When $(A+B).C$ is TRUE, there is a problem.
The correct value for X is now '0'. After the pre-charge cycle, X takes some time to discharge



to '0'. So for some time after pre-charge, its output is held at the wrong value of '1'. During this time, the charge placed on the output capacitor of inverter is leaked away to ground as the input to nMOS of the inverter is not '0' and the clock input is high. This can lead to a wrong evaluation of the logic function!

In a dynamic logic stage, the output is pre-charged to '1'. If the final output is supposed to be '0', there will be some time during which the output will still be at the wrong value of '1'. This transiently wrong value can discharge the pre-charged output capacitor of the next stage and thus lead to malfunction. So we need to isolate the output of a dynamic logic stage from the input of the next dynamic logic stage till it has acquired the correct value.

This means the operation of dynamic logic should proceed in distinct time slots or 'phases'. The output of the stage should be connected to the input of the next stage only in the phases in which its output is valid and stable. So the sequence of operations should be:

- Pre-charge the output in the first phase.
- Disable pre-charging and carry out logic evaluation in the next stage. The Output should be disconnected from the next stage during pre-charge as well as evaluation phases.
- In the final phase(s), the output holds its correct value. In this phase, connect the output to the next stage and disconnected it from pre-charge and evaluate circuits of the current stage.
- Thus, a minimum of 3 phases are required – pre-charge, evaluate and output-valid.
- In a 4-phase implementation, the valid state holds for a duration of two phases instead of just one.

- To implement this kind of dynamic logic, we need a multi-phase clock.

An example scheme for generating a 4 phase clock is shown below. The nor gate inserts a 1 in the shift register whenever it sees three 0's at Q0, Q1 and Q2.

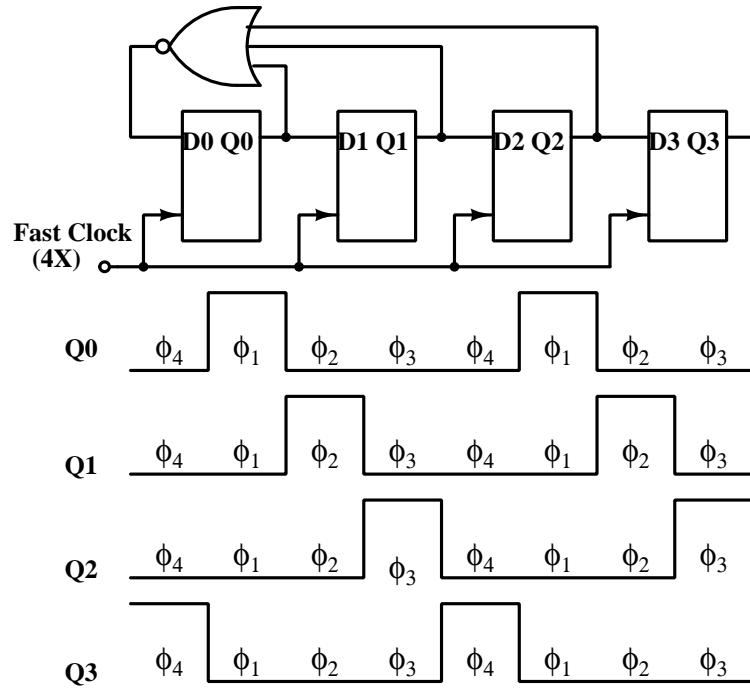


Figure 5.3: Generation of phase clocks for dynamic logic

If any of these is a 1, it inserts 0's. Thus, exactly one D flipflop among the four holds a 1 at any time, all the rest hold 0. Thus, each Q output is high during a single phase of a 4 phase cycle.

5.2 Four Phase Dynamic Logic

As discussed above, We use different phases for pre-charge, evaluation and for holding a valid output.

For the circuit shown in Fig. 5.4, we have a 4 phase clock. Ck_{mn} is a clock which is high during the m and n phases of the clock. Similarly, \overline{Ck}_{mn} is a signal which is low during m and n phases of the clock. Combined clock signals of the type Ck_{mn} can be generated easily using simple logic.

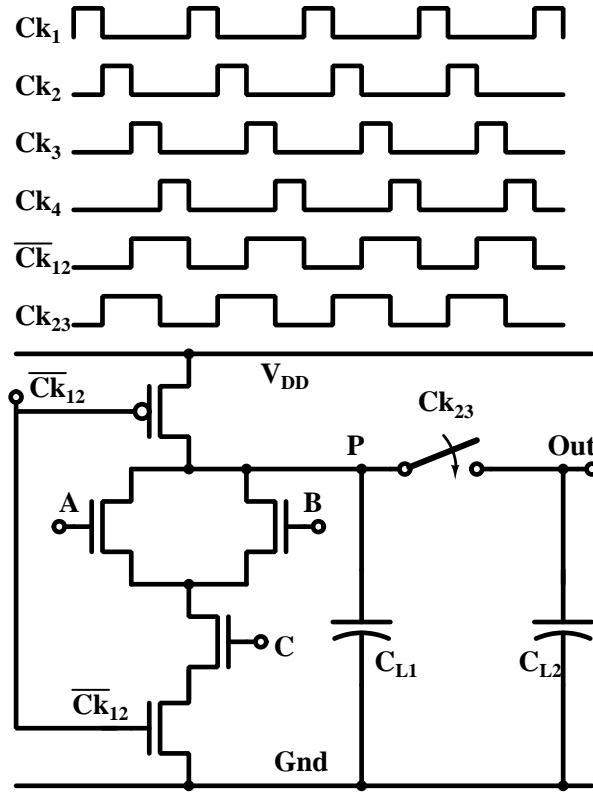


Figure 5.4: CMOS 4 phase dynamic logic

Now, for the gate shown in the figure, In phase 1, $\overline{Ck_{12}} = 0, Ck_{23} = 0$. So the pMOS is on and the bottom (clocked) nMOS is off. The output is disconnected from transistors. Node P pre-charges to ‘1’ while the output holds its old value.

In phase 2, $\overline{Ck_{12}} = 0, Ck_{23} = 1$. So the pMOS is on, bottom (clocked) nMOS is off and the output capacitor is in parallel with the capacitor on node P. As a result, node P, as well as the output node pre-charge to 1.

In phase 3, $\overline{Ck_{12}} = 1, Ck_{23} = 1$. So the pre-charge pMOS is off and the bottom (clocked) nMOS is on. Capacitors at node P and at the output are still in parallel.

If $(A + B) \cdot C = 1$, both capacitors will discharge to ground through the signal transistors and the bottom (clocked) nMOS. Otherwise the output will remain at 1. Thus the gate evaluates in phase 3 and acquires the correct output value at the end of this phase.

In phase 4, $\overline{Ck_{12}} = 1, Ck_{23} = 0$. The output is isolated from transistors and will hold its

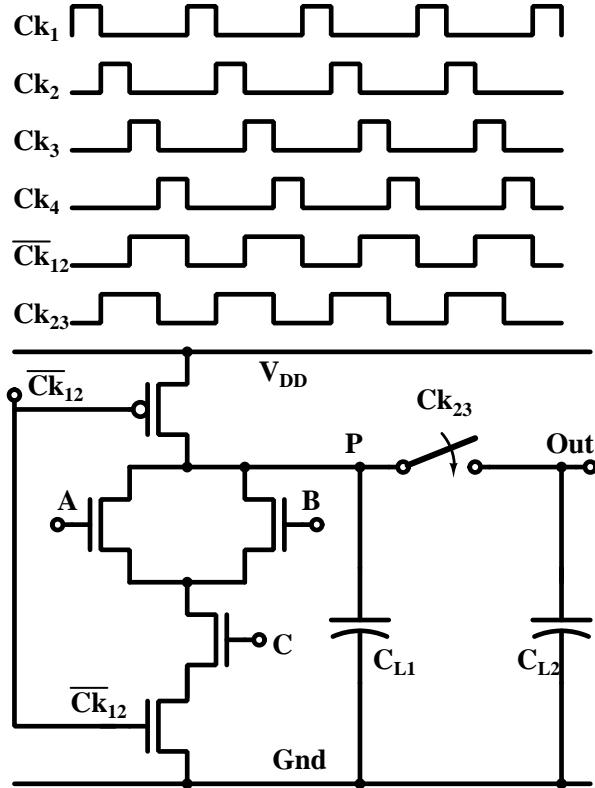


Figure 5.5: Operation of a type 3 gate

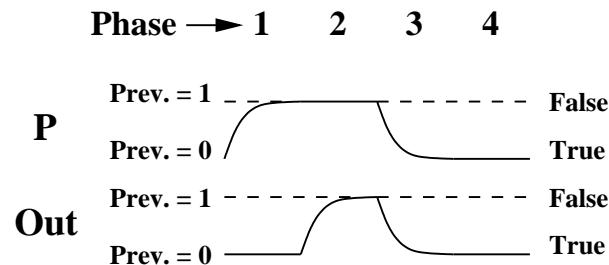
valid value. In phase 4 as well as in phase 1, the output is isolated from the driver and retains its valid value. However, Notice that node P no more holds a valid value in phase 1, since it pre-charges to ‘1’ in this phase.

This is called a type 3 gate, and needs the inputs to be valid and stable in phase 3. By changing the clocks to \overline{Ck}_{23} and Ck_{34} , we shall get a type 4 gate which evaluates in phase 4 and whose output remains valid in phases 1 and 2. Similarly, by cyclic permutation of clock signals, we can get type 1 and type 2 gates. A type 3 gate evaluates in phase 3 and is valid in phases 4 and 1. Similarly, we can have type 4, type 1 and type 2 gates.

The output of a type 3 gate is correct and stable in phases 4 and 1. Consequently, its output can be used by type 4 and type 1 gates without any malfunction.

Each logic gate type in 4-phase dynamic logic design, needs its inputs to be valid and stable during one phase and provides an output which is valid and stable for two clock phases. A type 3 gate can drive a type 4 or a type 1 gate. Similarly, type 4 will drive types 1 and 2; type 1 will drive types 2 and 3; and type 2 will drive types 3 and 4. We can use a 2 phase clock if we stick to type 1 and type 3 gates (or type 2 and type 4 gates) as these can drive

- Ck_{mn} is defined as a clock signal which is ‘high’ during phase m and phase n of the clock.
- Similarly, \overline{Ck}_{mn} is a clock signal which is ‘low’ during phase m and phase n of the clock.



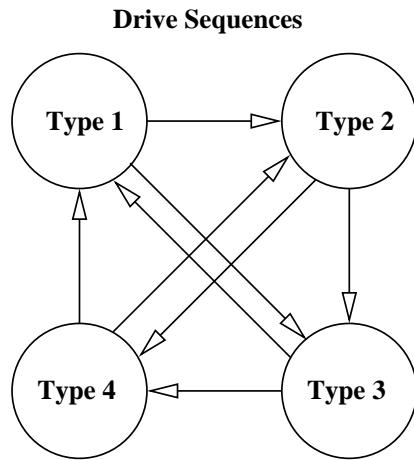


Figure 5.6: CMOS 4 phase dynamic logic drive constraints

each other.

5.3 Domino Logic

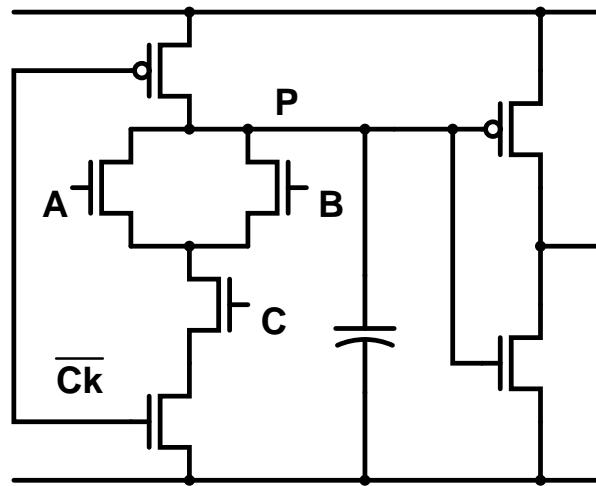


Figure 5.7: CMOS domino logic

Another way to eliminate the problem with cascading logic stages is to use a static inverter after the CMOS dynamic gate. Recall that cascading of dynamic CMOS stage causes problems because the output is pre-charged to V_{dd} . If the final value of a stage is meant to

be zero, the next stage nMOS to which this output is connected erroneously sees a one till the pre-charged output is brought down to zero. During this time, it ends up discharging its own pre-charged output, which it was not supposed to do.

If an inverter is added, the output is held ‘low’ before logic evaluation. Now, if the final output of this gate is ‘0’, there is no problem anyway. If the final output was supposed to be ‘1’, the next stage is erroneously held at zero for some time. However, this does not result in a false evaluation by the next stage. The only effect it can have is that the next stage starts its evaluation a little later.

Domino logic is fast, because the pre-charge cycle is common to all stages. Once pre-charge is done, each stage evaluates one after the other (hence the name - domino logic). Thus for n stage logic, there is only one cycle of pre-charge and n cycles of evaluation, rather than n cycles of pre-charge and n of evaluation.

However, the addition of an inverter means that the logic is non-inverting. Therefore, it cannot be used to implement any arbitrary logic function. In synchronous digital circuits, combinational logic alternates with clocked latches. If inversion is required, we insert a latch at that place and take the \overline{Q} output of the latch to the next group of domino logic.

5.4 Zipper logic

Instead of using an inverter, we can alternate n and p evaluation stages. Now each stage is inverting and any logic can be implemented using such stages. The constraint now is that an n stage can only drive a p stage, and a p stage can only drive an n stage. This kind of logic is called *zipper* logic, in analogy with a zipper where links from alternate sides are joined one after the other. The n stage is pre-charged ‘high’. If the output was supposed to be ‘high’, it is anyway correct and cannot cause a malfunction. If the eventual output was supposed to be ‘low’, this transiently wrong ‘high’ output will not harm the next stage whose evaluation transistors are p type. A ‘high’ output will keep the transistors off for some time and no charge will be leaked away from capacitors. When the stage reaches its correct output, the next stage will evaluate its output correctly.

Similarly, the p stage will be pre-discharged to ‘low’. If the output was supposed to be ‘low’, it is anyway correct and cannot cause a malfunction. If the eventual output was supposed to be ‘high’, this transiently wrong ‘low’ output will not harm the next stage whose evaluation transistors are n type. A ‘low’ output will keep the nMOS evaluation transistors of the next stage off for some time and no charge will be leaked away from capacitors. When the stage reaches its correct output, the next stage will evaluate its output correctly.

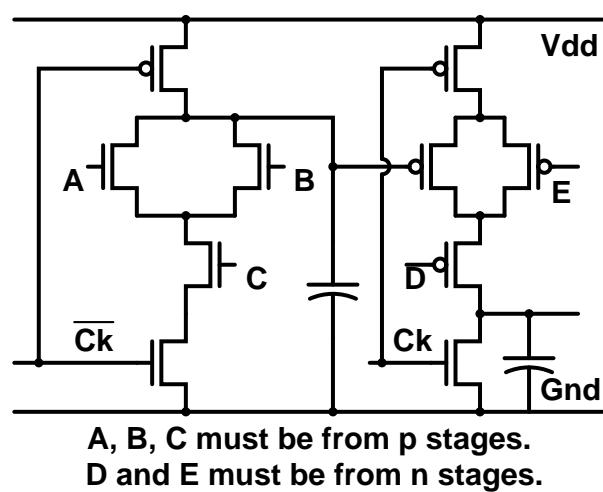


Figure 5.8: Zipper logic

Chapter 6

Semi-Custom Design

In the previous chapters, we have looked at design techniques used when we have to design all parts of a VLSI circuit from scratch. This includes the choice of logic styles, adjustment of transistor geometries, their layout etc. to meet a given set of specifications. This is called full custom design. While this permits an optimal tradeoff between power, speed and complexity, the whole process is long and laborious. The cost of developing a custom designed VLSI circuit is very high and can be justified only by those designs which sell in very large quantities.

Some applications can afford to compromise on speed, power and complexity in order to achieve a quick design and fabrication time and lower costs. These use a design style known as semi-custom design. In semi-custom design, part of the design and fabrication is already done for us. We take this pre-fabricated template and customize it to perform the functions that our VLSI needs to perform. This approach has several advantages. The prefabricated part can be processed and kept ready for customization. Then the time to take an application to the market is only the processing which is necessary after customization.

Also, different applications can use the same pre-fabricated template. While each application may have a relatively small market, the template will be used in very large numbers, making it economically feasible. Obviously, the pre-fabricated template is not customized for final requirements of a particular final circuit. So it will not be an optimal design for a given application. However, most applications do not require absolutely the best performance. In such cases, the time to market and cost can be drastically reduced by using semi-custom design.

Clearly, if the customization step occurs later in the design and fabrication cycle, the time to market will be shorter. However, the overall implementation may have a lower performance, since a higher fraction of the design and fabrication cycle is non-specific to the actual design.

An example of semi-custom design is Field Programmable Gate Array (FPGA) based design. In this case, customization is done *after* the product has actually been delivered to the end user. (That is what the term Field Programmable refers to). The actual VLSI template which can cater to this late customization is quite complex. Therefore the area is much higher and the speed much lower compared to what could have been achieved by a custom designed circuit. For example, system clock speeds possible with FPGAs are of the order of hundreds of MHz, whereas custom circuits can go to clock speeds of about 4 GHz. To implement a given function, the silicon area consumed by an FPGA may be an order of magnitude higher than the area consumed by custom designed circuit performing the same function. However, since many applications can use the same FPGA, the FPGA chip is produced in very large quantities, which provides economies of scale. Thus, for designs which are not produced in very large quantities, FPGA based design may be the most cost effective solution inspite of the overhead in complexity, power consumption and delay.

6.1 Procedure for Semi-Custom Design

There are two components of Semi-custom design technique.

1. Customization techniques which will be used for adapting the “fabric” to implement the final application on the given template.
2. The design of the basic template or the “fabric” which can be customized as late as possible and which can be used by a large variety of applications.

In the technologies used for fabricating VLSI circuits, transistors are defined much earlier than the interconnects. Interconnects are defined towards the end of the fabrication procedure. Therefore customization of a pre-fabricated template is commonly done by changing the interconnects.

6.2 Customization of interconnects

Customization of interconnects is done by placing programmable interconnect devices at appropriate points in the pre-fabricated template. These include fuses, anti-fuses and transistor based interconnect.

Fuses Customization of interconnects can be done through programmable removal of an existing connection. These work like fuses in electrical circuits. The connection to be customized is connected in the template through a narrow neck like structure. The narrow part of the interconnect is capable of passing normal operating currents of the circuit. However, during customization, one may pass a pulse of heavy current through this, which ‘blows’ the fuse and disconnects the wires leading up to the fuse.

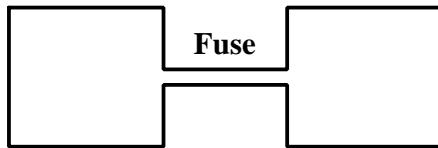


Figure 6.1: fuse structure

Anti-fuses In these structures, there is no connection between the programming nodes in the un-programmed state of the interconnect. The current path is interrupted through a thin layer of insulator. By applying a high voltage pulse, this insulator can be broken

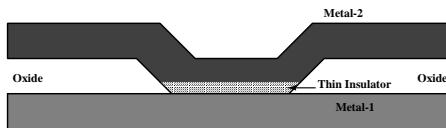


Figure 6.2: An anti-fuse. The insulator could be amorphous silicon.

down and a short created across it. This action is opposite to that of a fuse, so such structures are called anti-fuses.

Transistor based connection In this approach, a transistor is placed in the current path as a switch. When this transistor is on, there is a connection. When it is off, there is no connection. The state of the transistor is decided by a memory, which can be pre-loaded at power on.

6.3 Implementation of Configurable Template

6.3.1 Programmable Logic Arrays

Logic functions can be expressed in a generic sum of products form.

We would like a regular circuit which can be re-configured to implement any sum of products. One way is to use a customisable array which produces different product terms and another customisable array which sums the products so produced. CMOS logic is not convenient for implementing this architecture. This is because simultaneous configuration of the p channel pull up network as well as that of the n channel pull down network is inconvenient. Pseudo NMOS gates are more suitable for re-configuration in this case, as the pull up is just a single grounded gate PMOS whose geometry and interconnection remains the same for all

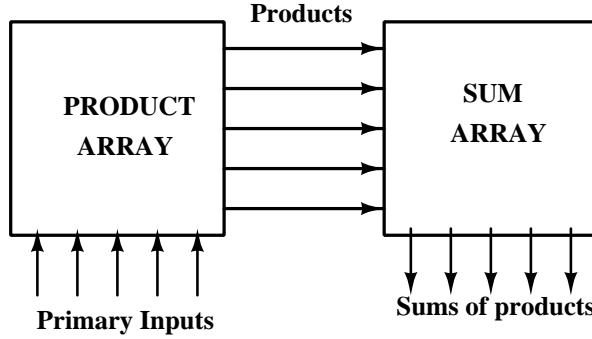


Figure 6.3: Architecture of a Programmable Logic Array

logic.

But Pseudo NMOS circuits are ratioed. Since the geometry of transistors is defined early during fabrication, we would like an architecture where transistor geometry does not depend on the logic being implemented.

Implementing sums is not a problem, since this will be done through a NOR like configura-

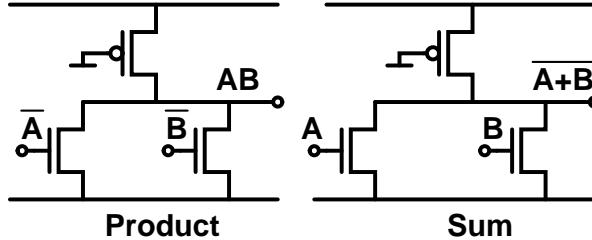


Figure 6.4: Product and sum implementation in a PLA

tion. In NOR configuration, each term contributes a transistor in parallel, whose geometry is the same as that of the reference inverter on which the design is based. Thus, sum of logic terms can be implemented with transistors whose geometry is fixed. However, implementing products presents a problem, since this requires NAND configuration, where the geometry of series connected pull down devices depends on the number of devices connected in series and hence, on the specific logic being implemented.

How do we implement the product function then?

We can use the expression : $A \cdot B = \overline{\overline{A} + \overline{B}}$.

Now the product of A and B can be implemented as the NOR of \overline{A} and \overline{B} , which does not use series connected transistors. By adding inverters at the input and output as required, we can

implement products or sums using only NOR type of circuits, which use constant geometry NMOS transistors in parallel.

Product Array

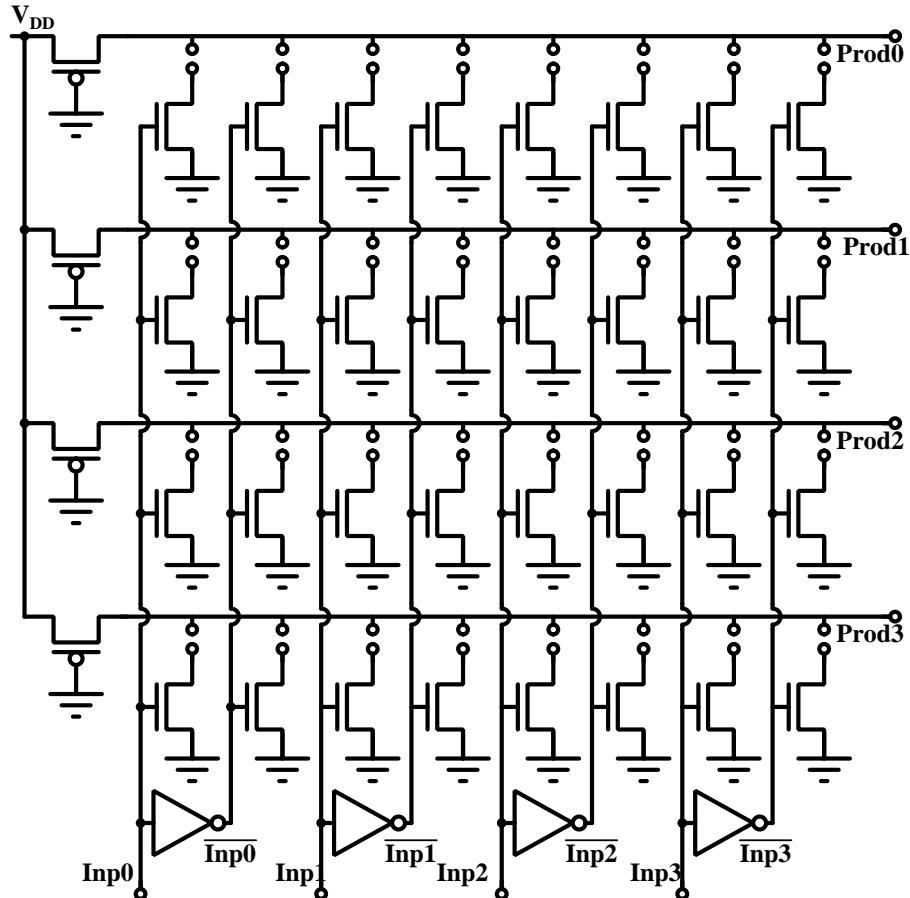


Figure 6.5: A programmable product array

Suppose we want to generate p distinct products of n inputs. Since for product implementation, we need complemented inputs, we connect a static CMOS inverter to each input. Now we have access to complemented as well as uncomplemented inputs. To design the product array, we place nMOS transistors in a matrix of $2n$ columns and p rows where n is the number of inputs and p is the number of distinct products that we want to generate. Since the complement of each of the inputs is produced using inverters, n signals and n complements drive the gates of nMOS transistors in $2n$ columns. Each row generates a

distinct product. It is pulled up by a grounded gate pMOS transistor. So this matrix can produce p distinct products. By connecting links selectively at drains of nMOS transistors in any row, chosen transistors can be included in parallel in the NOR configuration. Connecting a transistor includes the complement of the input driving its gate in the product term.

Each column is driven by an input or its complement. Thus each of these nMOS transistors has its source grounded and its gate connected to an input or its complement. Programming involves either connecting its drain to the row which forms the output or leaving it unconnected. If the drain is connected to the output, the transistor comes in parallel with other connected transistors, as required by the pseudo NMOS configuration.

If we want to include a signal (or its complement) in the product term, we connect the drain of the transistor driven by the complemented input (or uncomplemented input) to the row. If we do not want either the signal or its complement in the product, we simply leave both nMOS transistors unconnected.

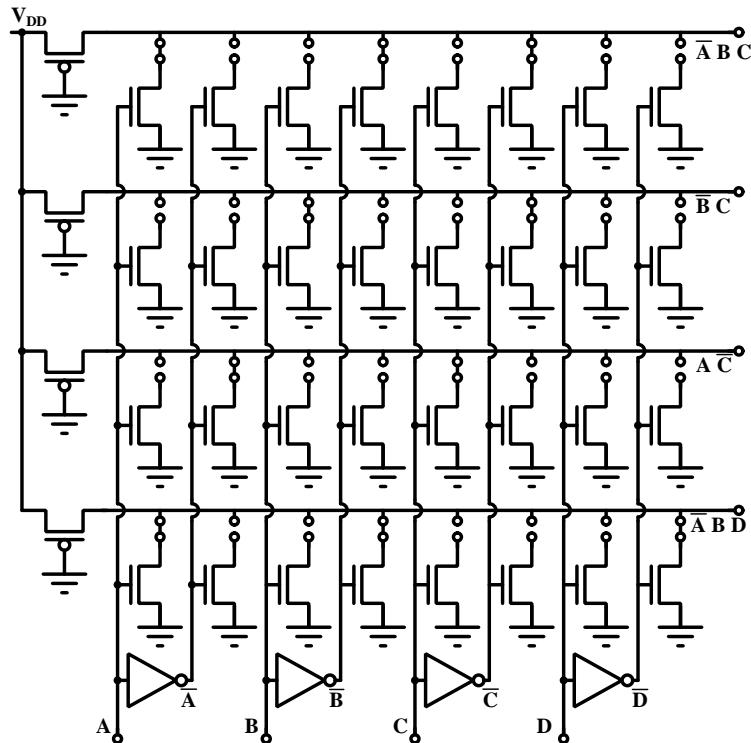


Figure 6.6: Example for generation of different products from inputs

In the example shown in Fig.6.6, we have generated products $\bar{A} \cdot B \cdot C$, $\bar{B} \cdot C$, $A \cdot \bar{C}$ and $\bar{A} \cdot B \cdot D$. To generate $\bar{A} \cdot B \cdot C$, links from drains of transistors driven by A , \bar{B} and \bar{C} are connected to the top product row. For $\bar{B} \cdot C$, drains of transistors driven by B and \bar{C} are

connected to the output line in the second row. Other products are generated similarly.

Sum Array

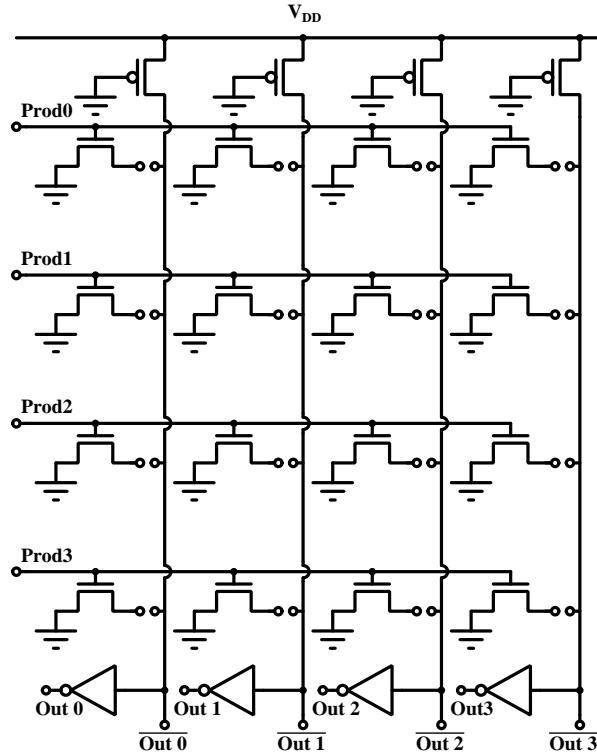


Figure 6.7: Sum array for generating sums of products

We can form the sum matrix similarly. Each of the products generated by the product array drives gates of a nMOS transistors in a row, arranged in a $p \times s$ matrix, where s is the number of different sums of products that we need to generate. The circuit in Fig.6.7 can produce sums of selected product terms. By connecting links at drains of nMOS transistors in a particular column, chosen products can be included in a sum output. Several columns can be used to generate different sums of products. Outputs are NORs of products. Inverters convert these to ORs of products.

Implementation of Finite State Machines

A finite state machine has the following components:

1. Storage elements to store the current state,

2. Random logic to compute the next state as a function of current state and current inputs, and
3. Random logic to compute the current output as a function of current state and optionally, the current inputs.

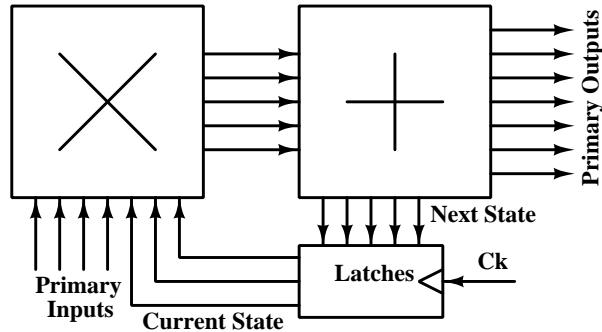


Figure 6.8: A finite state machine implemented with a PLA

By adding latches at the output of the PLA, we can provide for all of these. Output from latches is fed back to the product array. The PLA now computes both the next state and current outputs with primary inputs and current state as its inputs.

6.3.2 Sea of Gates

The programmable logic array uses pseudo nMOS design style as its base. This involves static power consumption. The stray capacitance of a programmable plane is high – which makes PLAs rather slow.

How can we use CMOS style gates in a semi-custom design? The “Sea of Gates” template provides the capability to use CMOS style gates in a semi-custom design. In CMOS logic, each input goes to an nMOS as well as a pMOS. In this style of design, all transistors are pre-placed in a pattern which is optimum for implementing CMOS gates because there are pairs of nMOS and PMOS transistors driven by the same input at their gates. Once an array of such patterns is placed in the chip, interconnects will determine what kind of logic will be implemented. Both n and p channel transistors appear to be in series here! How do we construct regular CMOS logic gates using these?

Actually, by wiring the apparently series connected devices, we can convert them to series or parallel as required. The example in Fig.6.10 shows a NAND gate, whose output is fed to an inverter to form the AND function. The structure on the right forms a NOR gate.

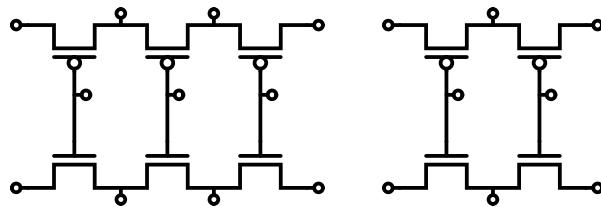


Figure 6.9: Pattern of transistors in a sea of gates template

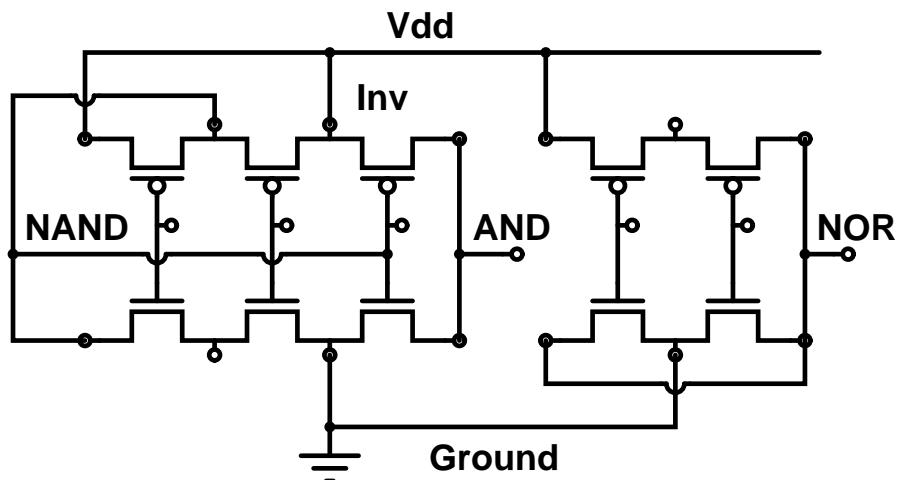


Figure 6.10: Implementing a NAND, Inverter and NOR gates in sea of gates

The sea of gates template makes use of the fact that all inputs go to an nMOS as well as to a pMOS in CMOS style gates. What about structures which don't follow this rule? For example, in a transmission gate, the nMOS and pMOS transistors are driven by complementary signals.

Transmission gates are often used in pairs with one or the other being on. (This is so because Inputs should not be left floating in static CMOS design). A pair of transmission gates can be implemented as shown in Fig.6.11, coupling diagonally opposite transistors in a 2-pair. As an application of this, a transparent D latch uses transmission gates with complementary control inputs. It can be implemented as shown in Fig.6.12. Two such latches can be connected in master-slave configuration to form an edge sensitive D flip-flop.

Sea of gates based designs provide better performance compared to PLA based designs. However, these cannot be field programmed and the interconnect mask has to be customized and used during chip manufacture.

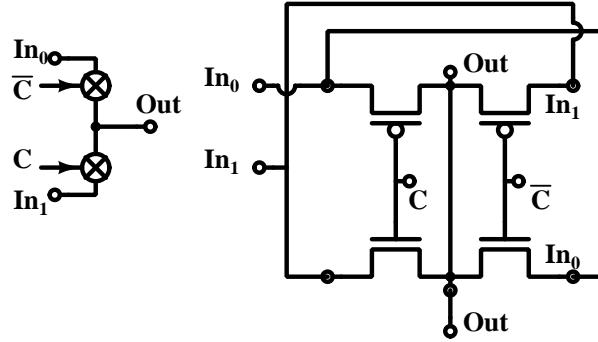


Figure 6.11: A pair of transmission gates with complementary control inputs

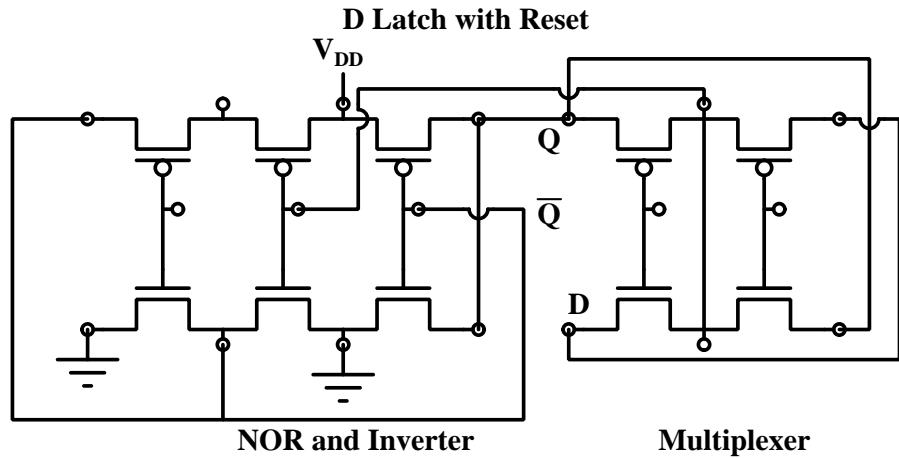


Figure 6.12: A transparent D latch implemented with “Sea of Gates”

6.4 Interconnect Channels in Semi-Custom Logic

Apart from programming the transistor matrix, we need to provide programmable interconnect between different sub-units. therefore in a semi-custom integrated circuit, pre-fabricated interconnect channels alternate with transistor matrices.

The exact shape and composition of these interconnect channels varies from design to design. Typically, these include facilities for local as well as global interconnects. The size of these blocks is optimized to provide a sufficient interconnect capacity without wasting too much area.

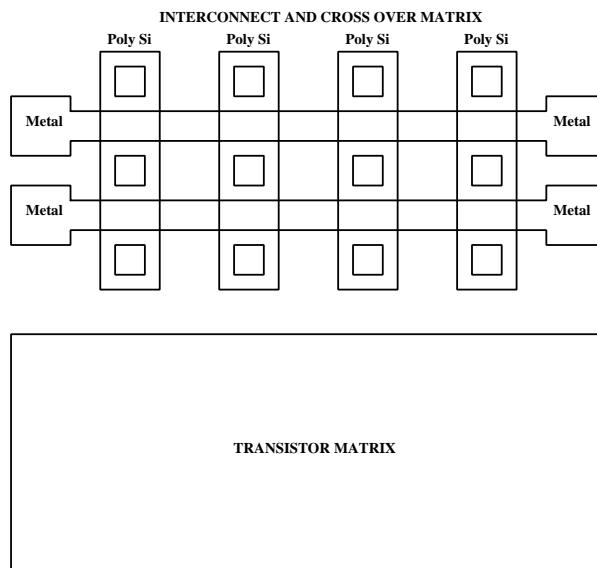


Figure 6.13: An example interconnect channel for semi-custom logic

6.5 Using Memories as Logic

A logic function can be represented by a truth table. This can be stored in memory. Inputs to the logic function act as address lines. Precomputed value of the logic function are stored at the address represented by its input values. For any input combination, the address represented by these is looked up and the stored value presented as the output.

For example, any logic function of 5 inputs can be implemented by a 32×1 bit memory. If the same 32 bit memory is organized as 16×2 , it can store two logic functions of up to 4 inputs.

6.6 Field Prgrammable Gate Arrays

Different types of semi-custom design chips have been introduced with various names. Here is a list:

PLAs: These were the first semi-custom devices to be introduced in the market by Philips in the early 1970's. These are used to implement generic sums of products and have a programmable AND plane as well as a programmable OR plane. Because both AND and OR arrays are programmable in PLAs, these are rather slow.

PALs: A modification of PLAs in which the product array is programmable, but the OR

plane is fixed were introduced as Programmable Array Logic or PAL. To cover for the lack of flexibility due to the OR array being fixed, these were introduced in different size combinations and multiple devices were used for different functions.

CPLDs As technology progressed, it was possible to put multiple devices on the same chip.

Combinations of PALs and PLAs were introduced with programmable interconnect as complex programmable logic devices or CPLDs. Altera was one of the first companies to introduce CPLDs commercially. CPLDs were a huge success and multiple companies introduced CPLDs of different architectures.

Sea of Gates In parallel with field programmable devices, mask programmable devices were manufactured. In these, one (or more) levels of metallization could be customized at the manufacturing site, over a “sea” of pre-fabricated devices. These provided circuits with better performance, but with less flexibility as programming could not be done at the user site.

FPGAs Field Programmable Gate Arrays borrow features from sea of gates as well as CPLDs. Instead of a “sea” of transistors, these have a “sea” or a repetitive array of combinations of logic elements and memories.

In addition, these include a programming infrastructure for interconnects like CPLDs.

On the periphery of these chips, special Input Output devices are fabricated which help in establishing fast interconnection with the external world.

A field Programmable array allows the logic functions as well as the interconnect to be programmed. Transistors driven by SRAM can be used to program interconnects. Apart from logic blocks and interconnect fabric, modern FPGA’s contain other useful structures, such as

- Block RAM,
- Processor cores (Power PC on Xilinx Vertex family),
- Fast multipliers
- I-O Blocks

A typical board, used for implementing a variety of applications will have a micro-controller and a few programmable devices for providing dedicated functions and glue logic. This is configured once at power on, and then left alone to perform its functions. This is called “static re-configurability”.

But we may want to re-configure a structure while it is in operation! For example, one can imagine the design of digital filter, which adapts itself during operation itself depending on

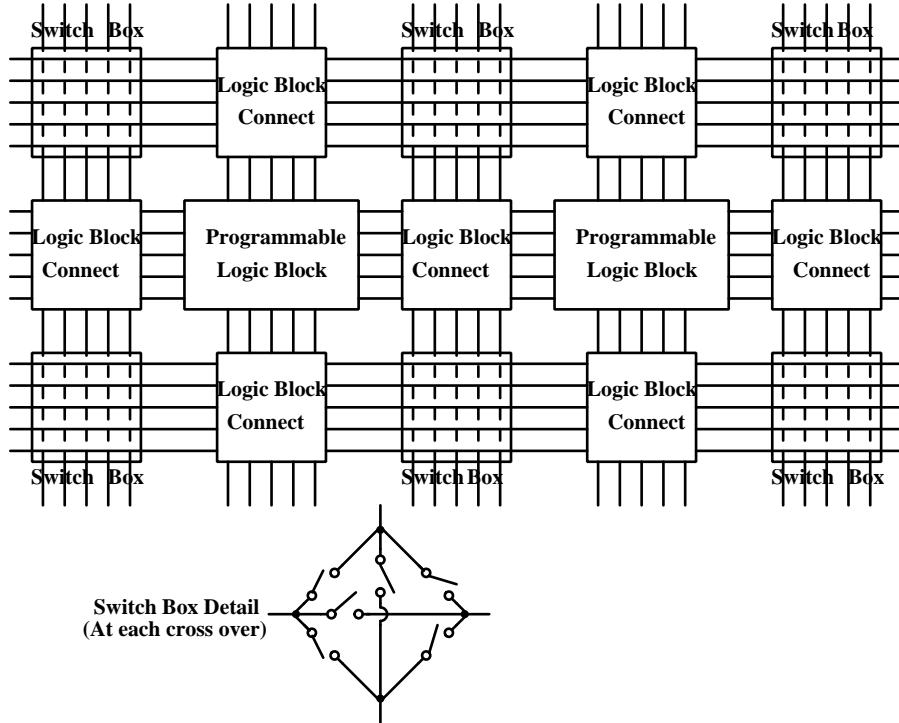


Figure 6.14: Alternating logic and interconnect boxes in an FPGA

inputs. This kind of re-configurability is called “dynamic re-configurability”.

Obviously, dynamic configurability requires that the dead time during re-configuration should be minimized. Thus, there is a trade-off between flexibility of re-configuration and the time taken to re-configure. In fine grain re-configuration, we can re-program every gate of the design. This is the case for current FPGA and CPLD devices. This gives very good design flexibility, but takes a long time to re-configure.

In coarse grain re-configuration, relatively large functional blocks are re-configured. For example, by re-connecting a collection of shifters and adders, we can generate any combination of multipliers and adders. The effort to re-configure is smaller, because we do not alter the inner design of shifters and adders. This is compatible with dynamic re-configuration. Using coarse grain re-configurability, it becomes possible to dynamically change a circuit, *during* its operation. This has obvious advantages in adaptive circuits.

We normally do not want to re-configure the whole circuit. Indeed the control signals for doing the re-configuration will be generated by a circuit which remains unchanged. Therefore

dynamic re-configurability requires circuits which can be partially re-programmed. Many FPGA are beginning to promise this feature.

Chapter 7

Multi-Stage Logic Design

We have seen how to design single stage logic in different design styles. However, typical designs need multi-stage combinational logic. In this chapter, we'll discuss techniques used for the design of multi-stage logic.

7.1 Stage Delay and Sizing

In chapter ??, we had derived expressions of the type

$$\frac{K\tau_L}{C_L} = \int_{V_1}^{V_2} \frac{dV}{f(V)}$$

for the delay of a single logic stage. Here τ_L is the time taken to charge/discharge the load capacitor C_L from V_1 to V_2 and K is the conductance factor given by $\mu C_{ox} \frac{W}{L}$.

The right hand side of this equation is a definite integral. It will evaluate to some constant depending on the voltages defining the ‘High’ and ‘Low’ logic levels, the supply voltage, turn on voltages, drain saturation voltage etc. In digital design, we shall keep the channel length at its minimum value, so L is a constant. Let us initially ignore the parasitic capacitances. We can see that

$$\frac{W\tau_L}{C_L} = \text{Constant} \quad \text{so } \tau_L \propto \frac{C_L}{W}$$

This tells us that the delay associated with a gate charging a load capacitor scales directly with C_L and inversely with W , the width of the charging/discharging transistor. This linear dependence permits us to design logic stages easily.

7.2 Tapered Buffer

As an example of multi-stage logic, let us take the case when a large capacitor is to be driven by a CMOS circuit. A minimum sized inverter will take too long to charge this capacitor. Therefore, we would like to scale up an inverter (multiply all transistor width by a scale factor) in order to drive this large capacitor. However, the input capacitance of this scaled up inverter may be too large for a minimum sized inverter to drive. Therefore, we need a medium sized inverter to drive the large final inverter. We keep adding inverters, till the first inverter in the chain is small enough to be driven by standard CMOS logic. This kind of buffering is referred to as a *tapered buffer*.

How do we decide the number of inverters to include in this chain? And what should be the

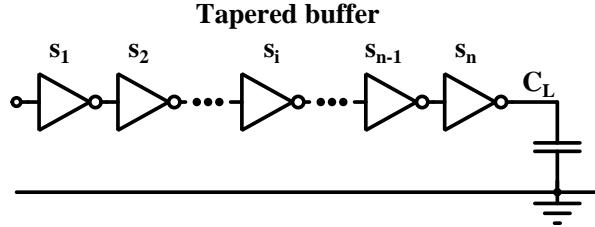


Figure 7.1: A tapered buffer

scale factors for each successive stage to minimize the total delay?

Let the i 'th inverter in the chain be scaled up by a factor s_i relative to a minimum sized inverter. Let the delay of a minimum sized inverter driving another minimum sized inverter be τ . Then the i 'th inverter provides charging currents which are s_i times the minimum sized inverter. However, the load it sees is s_{i+1} times the input capacitance of a minimum inverter. Therefore the delay associated with the i 'th stage is $\frac{s_{i+1}}{s_i}\tau$. The total delay of the inverter chain is given by

$$d_{total} = \sum_1^n \frac{s_{i+1}}{s_i} \tau = \tau \sum_1^n \frac{s_{i+1}}{s_i} \quad (7.1)$$

In order to minimize the total delay, we should put the partial derivative with respect to each of the s_i equal to zero. Therefore,

$$\tau \frac{d}{ds_i} \left(\frac{s_2}{s_1} + \dots + \frac{s_i}{s_{i-1}} + \frac{s_{i+1}}{s_i} + \dots \right) = 0 \quad (7.2)$$

Only two terms in the sum contain s_i . Since all scale factors s_i are independent, the derivative of all the rest of the terms is 0. Therefore,

$$\frac{1}{s_{i-1}} - \frac{s_{i+1}}{s_i^2} = 0 \quad \text{Which gives:} \quad \frac{s_i}{s_{i-1}} = \frac{s_{i+1}}{s_i} \quad (7.3)$$

This means that the *stage ratio*, which is the factor by which an inverter is larger than the previous one, should be the same for all stages.

Let this constant stage ratio for the tapered buffer be ρ . The delay contributed by the i 'th stage is $\frac{s_{i+1}}{s_i}\tau = \rho\tau$. The first stage is a unit inverter. Each subsequent stage has a drive capability which is ρ times the drive capability of the previous stage. Since the drive capability is being stepped up by ρ in n stages, we should have

$$\rho^n = \frac{C_L}{C_{in}} \quad \text{so } \rho = \left(\frac{C_L}{C_{in}}\right)^{1/n} \quad (7.4)$$

We define the ratio $H \equiv C_L/C_{in}$. Then, $\rho^n = H$ and so, $n = \ln H / \ln \rho$.

The total delay is given by

$$d_{total} = \sum_1^n \frac{s_{i+1}}{s_i} \tau = \sum_1^n \rho\tau = n\rho\tau \quad (7.5)$$

We want to find the value of ρ which will minimize the total delay. Note that n and ρ are not independent since $\rho^n = H$. Taking logarithms on both sides, we get

$$n \ln \rho = \ln H, \quad \text{so} \quad n = \frac{\ln H}{\ln \rho} \quad (7.6)$$

The total delay can then be expressed as

$$d_{total} = \frac{\ln H}{\ln \rho} \rho\tau = \tau \ln H \frac{\rho}{\ln \rho} \quad (7.7)$$

Total delay will be minimized with respect to ρ when its derivative with respect to ρ is 0. This gives

$$\tau \ln H \left(\frac{1}{\ln \rho} - \frac{\rho}{(\ln \rho)^2} \frac{1}{\rho} \right) = 0 \quad (7.8)$$

This leads to

$$\frac{1}{\ln \rho} = \frac{1}{(\ln \rho)^2}, \quad \text{so} \quad \ln \rho = 1, \quad \text{or} \quad \rho = e \quad (7.9)$$

$$\text{Since } \rho = e, \quad n = \frac{\ln H}{\ln \rho} = \ln H \quad (7.10)$$

Thus we obtain the result that the optimum stage ratio for a tapered buffer is e , while the optimum number of stages in the buffer is given by $\ln(C_{out}/C_{in})$.

(The optimum stage ratio comes out higher (≈ 3 to 4), if we take self loading into account.)

These results were computed for a situation where capacitors were driven only by inverters and self loading due to transistors in the gate was ignored. The general situation will differ from the tapered inverter chain in several respects:

1. Multi-stage logic can use gates other than inverters.
2. Each gate will have a parasitic delay associated with it due to the capacitance of the driver transistors themselves.
3. In the tapered inverter chain, each stage drives another inverter. In general, we can have branching, where a stage drives multiple gates. (Fanout is greater than 1).

The generalized optimization which removes these restrictions was developed by Ivan E. Sutherland and Robert F Sproull in a paper “Logical Effort: Designing for Speed on the Back of an Envelope.”, published in the Proceedings of the Conference on Advanced Research in VLSI, held in March 1991. The method was later described in much greater detail in a book by Sutherland, Sproull and Davis titled “Logical Effort: Designing Fast CMOS Circuits”, published by Morgan Kaufmann in 1998. The material in this chapter is largely based on that book.

First, Let us see what happens if we replace inverters in the tapered buffer with static CMOS logic gates.

7.3 Using Logic Gates Other Than Inverters

Transistor sizes in a gate are adjusted based on several considerations.

1. Difference in mobility of pMOS and nMOS. A PMOS transistor has to be wider than an NMOS transistor to provide the same drive current because the hole mobility is lower than electrons mobility. For example, we may scale the width of PMOS transistors to be double that of NMOS transistors connected in the same configuration.
2. If there are n series connected transistors, their widths should be scaled up by n . in order to make the output drive of the logic gate equivalent to an unscaled inverter.
3. After accounting for mobility differences and series connections, we may scale up *all* transistors by some factor in order to drive larger loads.

However, this has an impact on the capacitive loading placed on the *previous* stage. Let the ratio of widths of p and n channel transistors in a minimal inverter to get equal rise and fall times be γ . We express capacitive loading in units of capacitance of a minimum sized n channel MOS transistor. As we can see from Fig.7.2, an inverter places a load of $(1+\gamma)$ units on the previous gate, a 2 input NAND gate with the same output drive loads the previous stage with a capacitance of $2 + \gamma$ units and a 2 input NOR gate loads the previous stage with a capacitance of $1 + 2\gamma$ units. For example if $\gamma = 2$, the input capacitance of an inverter is 3 units, that of a 2 input NAND is 4 units, while a 2 input NOR presents a capacitance of 5

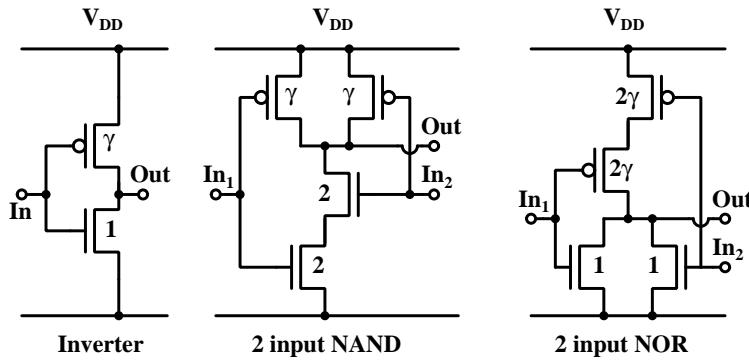


Figure 7.2: Load presented by 2 input NAND and NOR gates to their drivers.

units to its driver.

Thus we must account for a loading factor for different gate types when optimizing multi-stage logic. The ratio of $(2 + \gamma)/(1 + \gamma)$ for a 2 input NAND gate and of $(1 + 2\gamma)/(1 + \gamma)$ for a 2 input NOR gate is independent of eventual scaling for drive capability. A NAND gate scaled to provide the same drive as a scaled up inverter will also present a load which is $(2 + \gamma)/(1 + \gamma)$ times higher on the previous stage as compared to the corresponding scaled up inverter. The same can be said for the NOR gate or in fact, for any other CMOS gate.

This correction factor of $(2 + \gamma)/(1 + \gamma)$ for a 2 input NAND or of $(1 + 2\gamma)/(1 + \gamma)$ for a 2 input NOR gate is called the *logical effort* of this gate. This factor should be multiplied with the scale factor of this gate.

7.3.1 Delay model for a single gate

7.3.2 Considering the Effects of Self-Loading

If we consider the effects of self-loading, the logic has to drive some additional capacitance coming from the drain capacitance and Miller capacitance associated with the driver transistor. This additional capacitance is proportional to W . Thus,

$$C_L = C_{ext} + WC_p$$

Where C_p is the parasitic capacitance per unit width of driver transistors. Therefore the delay of the stage is

$$\tau_L \propto \frac{C_L}{W} = \frac{C_{ext} + WC_p}{W} \quad \text{Which gives} \quad \tau_L \propto \frac{C_{ext}}{W} + C_p \quad (7.11)$$

Thus the parasitic delay associated with self-loading is scale independent.

These refinements for computing the delay of a logic gate are encapsulated in the idea of “Logical Effort”.

7.3.3 Gate Delays with Logical Effort

In the method of Logical Effort, we deal with normalized delays. The unit of time is taken to be the delay of a minimum sized inverter driving another minimum sized inverter *without any parasitic elements*.

As discussed in the section above, the delay of a gate can be expressed as

$$d = f + p \quad (7.12)$$

where f is the *effort delay*, which depends on transistor currents and load capacitance, while p is the parasitic delay, which is size independent. We further express f as a product of two quantities, g and h . Here h is the *electrical effort* of this stage. This is given by the ratio of output capacitance to input capacitance. This is equivalent to the *stage ratio* we had used while discussing the tapered inverter. g is the *logical effort* which accounts for the extra loading caused by a gate as compared to an inverter, as discussed in the previous section. So,

$$f = gh \quad (7.13)$$

Combining Equations 7.12 and 7.13, we can express the delay introduced by a logic gate as

$$d = gh + p \quad (7.14)$$

where the delay is measured in units of τ , which is the delay of a minimum inverter driving another minimum inverter *not including the parasitic delay*.

This way of expressing delays separates the effects of different components which cause delay, so these can be handled independently.

1. Dependence on technology is encapsulated in τ . All delays are expressed as a multiple of this quantity. Thus delays in this formulation are dimensionless numbers and must be multiplied with τ to get the absolute value of delay.
2. Dependence on sizing is encapsulated in h . This is the only component of delay which is size dependent. h is also a unit less quantity, because it is defined as the ratio of output capacitance to input capacitance.
3. Dependence on logic type is expressed through g . This accounts for the series/parallel configuration of transistors in a particular gate.
4. The effect of parasitic delays due to the load presented by the driver transistors themselves is expressed through p . p is size independent. It is also a dimensionless quantity as this delay is also expressed in units of τ .

7.3.4 Logical Effort for Common CMOS Gates

The logical effort of an inverter is, by definition, 1. The logical effort of other logic functions depends on their circuit topology. As we had seen in Fig.7.2 the logical effort of a two input NAND gate is $(2 + \gamma)/(1 + \gamma)$ while that of a 2 input NOR gate is $(1 + 2\gamma)/(1 + \gamma)$.

n input NAND and NOR gates are shown in the figure below. The n input NAND gate

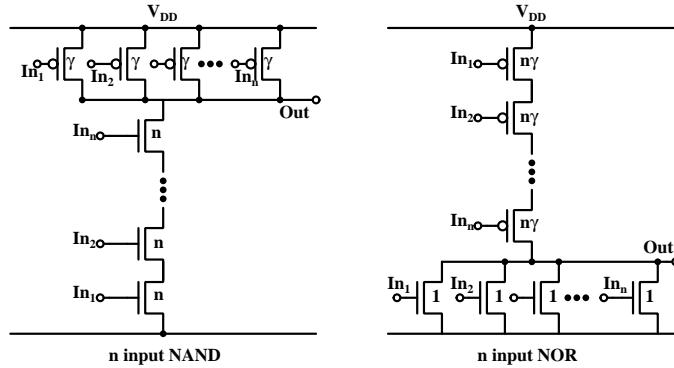


Figure 7.3: n input NAND and NOR gates

has n NMOS transistors in series and n PMOS transistors in parallel. Therefore each NMOS should have a width which is n times that of the minimum inverter. Each PMOS will have the same width as that of the minimum inverter – which is γ times the minimum size (to account for lower hole mobility). Taking the capacitance of a minimum sized transistor as the unit of capacitance, each input of the n input NAND gate loads its driver with $(n + \gamma)$ units of capacitance. The minimum inverter loads its driver by $(1 + \gamma)$ units. So the logical effort of an N input NAND gate is $(n + \gamma)/(1 + \gamma)$. This reduces to $(2 + \gamma)/(1 + \gamma)$ for a 2 input NAND, as expected.

We can approximate the parasitic delay by considering only the self capacitance which is directly connected to the output. In case of an inverter, this is proportional to $(1 + \gamma)$. For the n input NAND gate, the capacitance contributed by the n MOS transistors at the output node is n, while the capacitance from the n PMOS transistors is $n\gamma$. Therefore, the parasitic delay of the n input NAND is related to the parasitic delay of the inverter by the factor $(n + n\gamma)/(1 + \gamma) = n$. Thus we have

$$p_{NAND} = np_{inv} \quad \text{for an n input NAND gate} \quad (7.15)$$

The n input NOR gate has n PMOS transistors in series, each of which should have $n\gamma$ times the minimum width. It has n NMOS transistors in parallel, each of which can have the minimum geometry. Thus the loading on the driver of each input is $(1 + n\gamma)$ units. Thus the

logical effort of the N input NOR gate is $(1 + n\gamma)/(1 + \gamma)$. This reduces to $(1 + 2\gamma)/(1 + \gamma)$ for a 2 input NOR as expected.

For the parasitic delay of the n input NOR gate, the capacitance contributed by the n nMOS transistors at the output node is n, while the capacitance from the pMOS transistor is $n\gamma$. Therefore, the parasitic delay of the n input NOR gate is related to the parasitic delay of the inverter by the factor $(n + n\gamma)/(1 + \gamma) = n$. Thus we have

$$p_{NOR} = np_{inv} \quad \text{for an n input NOR gate} \quad (7.16)$$

A 2 way multiplexer is shown in Fig. 7.4 below. It is clear that each signal input is loaded

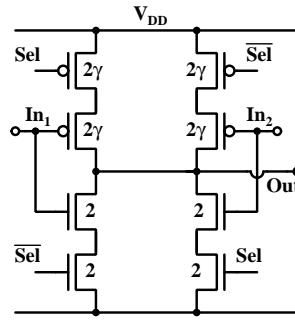


Figure 7.4: A 2 way multiplexer

with a capacitance of $(2 + 2\gamma)$ units. Therefore the logical effort for either data input in this mux is $(2 + 2\gamma)/(1 + \gamma) = 2$. The logical effort for the control inputs Sel and \bar{Sel} is also 2. The parasitic delay for the 2 way mux is $2 + 2\gamma + 2 + 2\gamma$ which is 4 times the parasitic delay of an inverter.

It is interesting to see that the logical effort will remain 2 for every data input even when we parallel n tri-stateable inverters to form an n way mux. However the parasitic delay will increase to $2np_{inv}$ if we add n units.

The table below lists the logical effort and the parasitic delays of common logic gates. Notice that the unit of time is the delay of a minimum inverter driving another minimum inverter (excluding parasitic delay). Therefore in these units, the parasitic delay of an inverter, which is the ratio the parasitic delay (in absolute units) to the inverter delay without taking parasitic delay into account, need not be 1. It is convenient to specify the parasitic delay of other gates as multiple of the parasitic delay of an inverter p_{inv} .

Gate	Logical Effort	Parasitic Delay
Inverter	1	p_{inv}
2 input NAND	$(2 + \gamma)/(1 + \gamma)$	$2p_{inv}$
n input NAND	$(n + \gamma)/(1 + \gamma)$	np_{inv}
2 input NOR	$(1 + 2\gamma)/(1 + \gamma)$	$2p_{inv}$
n input NOR	$(1 + n\gamma)/(1 + \gamma)$	np_{inv}
2 way mux	2	$4p_{inv}$
n way mux	2	$2np_{inv}$

Table 7.1: Logical effort and parasitic delays of common gates

7.4 Design of multi-stage logic

In multi-stage logic, we consider the logical effort g_i and electrical effort h_i of each stage. We define the *path logical effort* as the product of logical effort of all stages on a given path.

$$G = \prod_{i=1}^N g_i \quad (7.17)$$

Similarly, we define the *path electrical effort* as the product of electrical effort of all stages on a given path.

$$H = \prod_{i=1}^N h_i \quad \text{where} \quad h_i = \frac{C_{out_i}}{C_{in_i}} \quad \text{for stage } i. \quad (7.18)$$

If there is no branching, the load on a particular stage is just the input capacitance of the next stage, that is: $C_{out_i} = C_{in_{i+1}}$. When we multiply all electrical efforts along a path, all except the first and last capacitances cancel. Thus we get

$$H = \frac{C_L}{C_{in_1}} \quad (7.19)$$

Where C_L is the final load capacitance and C_{in_1} is the input capacitance of the first stage. If, however, there is branching at some stage i , $C_{out_i} \neq C_{in_{i+1}}$. This is because C_{out_i} includes not only $C_{in_{i+1}}$ but also the input capacitance of other logic gates which are not on the path under consideration.

7.4.1 Branching Effort

We introduce a new kind of effort, named *branching effort*, which corrects for the fact that $C_{out_i} \neq C_{in_{i+1}}$ at certain points in the logic chain. Suppose there is branching at the i 'th stage of the logic chain. The actual capacitive loading on the i 'th stage is $C_{onpath} + C_{offpath}$, where $C_{onpath} = C_{in_{i+1}}$ and $C_{offpath}$ is the sum of all the other input capacitances connected

to the output of the i 'th stage. It is convenient to continue to define H by Eqn. 7.19 even in the branching case. We now define a correction factor which is called the branching effort. Branching effort b_i at the output of a logic gate is defined to be:

$$b = \frac{C_{onpath} + C_{offpath}}{C_{onpath}} \quad (7.20)$$

where $C_{onpath} = C_{in_{i+1}}$, is the load capacitance of the gate being driven along the path being analyzed, while $C_{offpath}$ is the capacitance presented by the gates which are not on the path.

The output capacitance included in the definition of H is now multiplied by this correction factor at every stage.

$$b_i h_i = \frac{C_{onpath_i} + C_{offpath_i}}{C_{onpath_i}} \times \frac{C_{onpath_i}}{C_{in_i}} \quad (7.21)$$

If there is no branching at a stage, the corresponding b_i value is 1 since $C_{offpath} = 0$ and so, multiplication by b_i does not change anything. If, however, there is branching at the i 'th stage, multiplying h_i by b_i corrects the output capacitance, because

$$b_i h_i = \frac{C_{onpath_i} + C_{offpath_i}}{C_{onpath_i}} \times \frac{C_{onpath_i}}{C_{in_i}} = \frac{C_{onpath_i} + C_{offpath_i}}{C_{in_i}} \quad (7.22)$$

We define the branching effort of the whole path, denoted by B , as the product of the branching effort at each stage along the path.

$$B = \prod_{i=1}^N b_i \quad (7.23)$$

We can now define the *path effort*, F as the product of all logical efforts and branch corrected electrical efforts.

$$F = \prod_{i=1}^N g_i \prod_{i=1}^N b_i h_i = \prod_{i=1}^N g_i \prod_{i=1}^N b_i \prod_{i=1}^N h_i \quad (7.24)$$

So,

$$F = GBH \quad \text{where } G = \prod_{i=1}^N g_i, B = \prod_{i=1}^N b_i, \text{ and } H = \prod_{i=1}^N h_i \quad (7.25)$$

The advantage of using branching correction separately from electrical effort is that H retains its cancellation property for capacitance of intermediate stages and is defined only by the final load and the input capacitance.

$$H = \frac{C_L}{C_{in_1}} \quad (7.26)$$

The equation that defines the path effort looks quite similar to the definition of the stage effort in Equation 7.13, which defined the effort for a single logic gate. Notice, however, that

unlike the stage effort f , the path effort F does not define the delay of the path. The total delay is the *sum* of individual delays and not their product.

$$D = \sum d_i = \sum g_i b_i h_i + \sum p_i \quad (7.27)$$

Still, F is a useful quantity for optimisation of path delays as we shall see.

The path delay, D , is the sum of the delays of each of the N stages of logic in the path. As in the expression for delay in a single stage (Equation 7.14), we separate the contribution to path delay from effort and the delay due to parasitic capacitances.

$$D = \sum d_i = D_F + P \quad (7.28)$$

where D_F is the path effort delay, while P is the *path parasitic delay*. The path effort delay is simply $D_F = \sum g_i b_i h_i$ and the path parasitic delay is $P = \sum p_i$. In order to choose the optimum sizes of gates to minimize the total delay through the path, we need to minimize the above expression with respect to the transistor widths (and hence capacitances) of every stage. We have

$$D = \sum d_i = D_F + P = \sum g_i b_i h_i + P = \sum g_i b_i \frac{C_{i+1}}{C_i} + P \quad (7.29)$$

Notice that p_i (and hence P) are size independent and we can only minimize D_F by choosing optimum sizes of gates. Setting the derivative of D with respect to C_i to zero, and noticing that only two terms in the series expansion of D_F involve C_i , we can write

$$0 = \frac{\partial}{\partial C_i} \left(g_{i-1} b_{i-1} \frac{C_i}{C_{i-1}} + g_i b_i \frac{C_{i-1}}{C_i} \right) \quad (7.30)$$

This leads to

$$g_{i-1} b_{i-1} \frac{1}{C_{i-1}} - g_i b_i \frac{C_{i+1}}{C_i^2} = 0 \quad (7.31)$$

Which gives

$$g_{i-1} b_{i-1} \frac{C_i}{C_{i-1}} = g_i b_i \frac{C_{i+1}}{C_i} \quad \text{hence} \quad f_{i-1} = f_i \quad (7.32)$$

Thus, *the path delay is minimized when each stage in the path has the same stage effort*, f . The value of f which minimizes the path delay is denoted as \hat{f} . Since we had defined F as $F = \prod_{i=1}^N f_i$, in the optimum case, $F = \hat{f}^N$. In other words, the minimum delay is achieved when each stage effort is

$$\hat{f} = b_i g_i h_i = F^{1/N} \quad (7.33)$$

For this optimum effort, we obtain

$$\hat{D} = NF^{1/N} + P = N(GBH)^{1/N} + P \quad (7.34)$$

\hat{D} is the minimum delay possible for this path. We achieve this minimum delay by appropriate sizing of transistors in each stage of the logic path. Transistor sizes must be so chosen that the stage delay f is the same for all stages in the logic path. Equations 7.33 and 7.13 combine to require that each logic stage be designed so that

$$\hat{h}_i \equiv \frac{C_{out_i}}{C_{in_i}} = \frac{F^{1/N}}{b_i g_i} \quad (7.35)$$

We start with the last stage, where the output capacitance is known ($= C_L$). Since the output capacitance is known, the input capacitance can be calculated from Equation 7.35. This gives the scale factor for this stage from which, geometries of transistors can be computed.

Since $C_{out_i} = b_i C_{in_{i+1}}$, we can write the recursive relation

$$C_{in_i} = g_i b_i \frac{C_{in_{i+1}}}{F^{1/N}} \quad (7.36)$$

C_{in} for every stage can now be determined using the above recursive relation (Equation 7.36). From C_{in} , we can calculate the geometry of transistors for this stage.

We can equivalently start from the input side, computing the output capacitance and hence the input capacitance of the next stage. This can be continued till we reach the final output.

7.4.2 An example: 8-input AND network

When a large number of inputs must be combined, there are several options for the structure of the circuit. Figure 3 shows three possibilities for computing the AND function of eight inputs. Which one is best? Let us take $\gamma = 2$ and $p_{inv} = 0.6$ for this example. The parasitic delay of

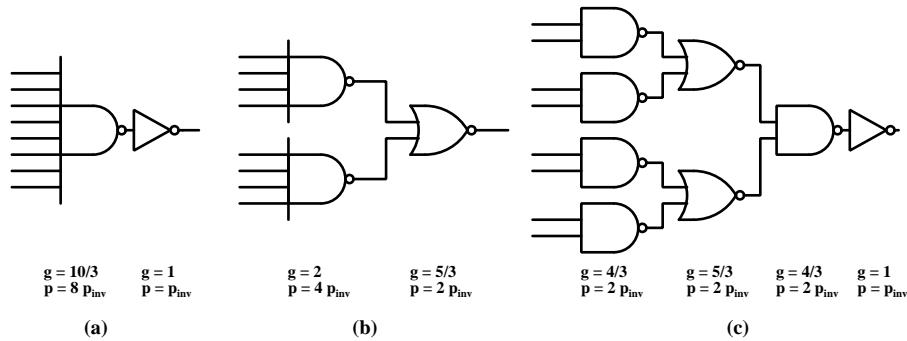


Figure 7.5: Three circuits that compute the AND function of eight inputs.

n input NANDs and NORs will then be $0.6n$. Recalling that the path logical effort, G , is the product of the logical efforts of the logic gates along the path, we find that $G = 10/3 \times 1 = 3.33$ for configuration a, $6/3 \times 5/3 = 3.33$ for configuration b, and $4/3 \times 5/3 \times 4/3 \times 1 = 2.96$ for configuration c. These figures can be used in the delay equation (7.34) to find the minimum delay that can be obtained from each circuit. The following equations also include an estimate of parasitic delays, obtained by summing the parasitic delays of each of the logic gates along the path:

$$\text{Configuration a } D = 2(3.33H)^{1/2} + 5.4 \quad (7.37)$$

$$\text{Configuration b } D = 2(3.33H)^{1/2} + 3.6 \quad (7.38)$$

$$\text{Configuration c } D = 4(2.96H)^{1/4} + 4.2 \quad (7.39)$$

It is clear from these equations that configuration b will always be better than a.

Choosing between configurations b and c depends on the electrical effort H , that must be borne by the network. When $H = 1$, delay in configuration b is 7.25, and in configuration c is 9.5. Therefore in this case, configuration b will be best. However, if $H = 12$, the delay in configuration b is 16.24, while for configuration c it is 13.97. Thus, configuration c will be best in this case. The equations show that for high electrical effort, configuration c yields the least delay because the $H^{1/4}$ factor dominates. (In the current example, configuration c is best for $H > 5.68$).

To illustrate the computation of transistor sizes to achieve least delay, consider a case where $C_{in} = 4$ units and $C_{out} = 64$ (so that $H = 16$). As discussed above, configuration c - which is a 4 stage design - is best for $H > 5.68$. So we shall choose this configuration for our example design.

- Unit of capacitance is the input capacitance of the reference inverter.
- Unit of width is the width of the n channel transistor in the reference inverter.
- So $1 + \gamma$ units of width correspond to 1 unit of capacitance.
- The input capacitance of a reference logic gate, obtained from the reference inverter by series parallel rules, is g .
- Thus, $(C_{in} = g) \Rightarrow$ scale factor = 1 over the reference logic gate.

For computation of transistor geometries, we calculate the value of C_{in} for each stage using the optimum value of stage effort \hat{f} . Then, Given C_{in} , we can determine the geometry

of individual transistors in each stage.

Unit of capacitance is the input capacitance of the reference inverter, while the unit of width is the width of the n channel transistor in the reference inverter.

So $1 + \gamma$ units of width correspond to 1 unit of capacitance. Also, the input capacitance of a reference logic gate, obtained from the reference inverter by series parallel rules, is g .

Thus, $(C_{in} = g) \Rightarrow$ scale factor = 1 over the reference logic gate. Then, for any given C_{in} , scale factor = C_{in}/g . So, if we know C_{in} , we can find the scale factor by dividing it by g , and then multiply all transistor widths in the reference logic gate by this scale factor to get individual transistor geometries. We can take the logic path from any of the inputs to the

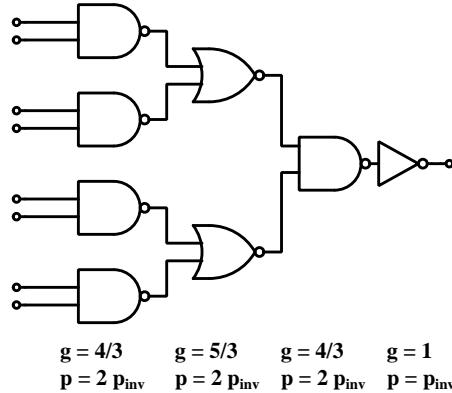


Figure 7.6: 4 stage implementation of 8 input AND

final output. On this path, we shall encounter:

1. a 2 input NAND ($g = 4/3$, $b = 1$),
2. a 2 input NOR ($g = 5/3$, $b = 1$),
3. a 2 input NAND ($g = 4/3$, $b = 1$), and
4. an inverter ($g = 1$, $b = 1$)

$$G = \frac{4}{3} \times \frac{5}{3} \times \frac{4}{3} \times 1 = 80/27 = 2.963 \quad B = 1, \quad H = \frac{64}{4} = 16$$

$$F = GBH = 47.4074, \quad \text{So} \quad \hat{f} = 47.4074^{1/4} = 2.624$$

In this computation, We begin from the load end. (We can also start from the first stage and go towards the load for the recursive calculation).

Last stage: Inverter

The last stage is an inverter, with $g = 1, b = 1$.

$$\hat{f} = 2.624 = gbh = 1 \times 1 \times h. \quad \text{So} \quad h = 2.624$$

$$h = \frac{C_{out}}{C_{in}} = \frac{64}{C_{in}} = 2.624$$

$$\text{So} \quad C_{in} = \frac{64}{2.624} = 24.39$$

So the final stage inverter is scaled up by 24.39 compared to the reference inverter. Taking the n channel transistor width in the reference inverter as the unit,
n-channel transistor width = 24.39
p-channel transistor width = $\gamma \times 24.39 = 48.78$.

Third stage: 2 input NAND

The stage driving the inverter is a 2 input NAND.

So $g = 4/3, b = 1, C_{out} = 24.39$.

$$\hat{f} = 2.624 = gbh = \frac{4}{3} \times 1 \times h$$

$$\text{So} \quad h = \frac{2.624 \times 3}{4} = 1.968$$

$$h = \frac{C_{out}}{C_{in}} = \frac{24.39}{C_{in}} = 1.968$$

$$\text{Therefore} \quad C_{in} = \frac{24.39}{1.968} = 12.3936$$

Scale factor for this stage is $12.3936/g = (12.3936 \times 3)/4 = 9.2952$.

The reference 2 input NAND gate has n channel transistor width = 2, and p channel transistor width = 2.

Therefore the transistor geometries in this stage will be $2 \times 9.2952 = 18.59$ for both n and p channel transistors.

Second stage: 2 input NOR

The stage driving the 2 input NAND is a 2 input NOR.

So $g = 5/3, b = 1, C_{out} = 12.3936$.

$$\hat{f} = 2.624 = gbh = \frac{5}{3} \times 1 \times h. \quad \text{So} \quad h = 1.5744$$

$$h = \frac{C_{out}}{C_{in}} = \frac{12.3936}{C_{in}} = 1.5744$$

Therefore $C_{in} = \frac{12.3936}{1.5744} = 7.872$

Scale factor for this stage is $7.872/g = (7.872 \times 3)/5 = 4.7232$.

The reference 2 input NOR gate has n channel transistor width = 1, and p channel transistor width = 4.

n-channel transistor width = 4.72,

p channel transistor width = $4 \times 4.7232 = 18.89$

First stage: 2 input NAND

Finally, we come to the first stage which is a 2 input NAND.

So $g = 4/3, b = 1, C_{out} = 7.872$.

$$\hat{f} = 2.624 = gbh = \frac{4}{3} \times 1 \times h. \quad \text{So} \quad h = 1.9680$$

$$h = \frac{C_{out}}{C_{in}} = \frac{7.872}{C_{in}} = 1.968$$

Therefore $C_{in} = \frac{7.872}{1.968} = 4$

This agrees with our specification that the input capacitance of the first stage should be equivalent to 4 inverters.

The scale factor will be $4/g = 4/(4/3) = 3$. In the reference NAND, all transistors have a width = 2. so in the first stage, all transistors will have a width = $3 \times 2 = 6$.

Design of 8 input AND: Summary of results

The following table gives the geometries of transistors in all stages:

Stage	I	II	III	IV
Logic type	2in NAND	2in NOR	2in NAND	Inverter
g	$4/3$	$5/3$	$4/3$	1
C_{in}	4	7.87	12.39	24.39
Scale Factor	3	4.7232	9.2952	24.39
n width	6	4.72	18.59	24.39
p width	6	18.89	18.59	48.78
Parasitic Delay	1.2	1.2	1.2	0.6

The total delay of the 4 stage implementation is:

$$4\hat{f} + \sum P_i = 4 \times 2.624 + 4.2 = 10.496 + 4.2 = 14.7$$

7.5 Optimizing the path length

Equation 7.33 requires that the number of stages in the logic path be known for optimizing the total delay. However, this may not be the optimum path length and we can sometimes get a faster circuit by buffering intermediate outputs in this logic path.

We assume that there is a logic path containing n_1 stages and we are free to add n_2 inverters to this path, if that results in a lower overall delay. We now consider the optimization of this logic path containing $N = n_1 + n_2$ stages. We shall assume that there is no requirement for n_2 to be even. (This implies that an inverted output is equally acceptable or else, the logic path and inputs can be suitably altered to produce the desired output). The optimization problem is to find the scale factors for each of the $N = n_1 + n_2$ stages, such that the delay is minimum.

The path effort $F = GBH$ for the n_1 stages of logic is known. This is because the logical effort of each of the logic gates is known to us, so that G may be evaluated. Branching, if any, in the logic chain is also known, so B can be evaluated. Finally, H depends only on the load capacitance and input capacitance, which is the starting specification for the optimization.

Addition of n_2 inverters does not change the value of G , since $g = 1$ for each of the n_2 inverters. Similarly, the inverters do not introduce any branching, so B remains the same. Finally, H is defined by the final load and the input capacitance of the first logic element, which is not changed by the inserted inverters. Therefore $F = GBH$ for the $N = n_1 + n_2$ stages (including n_2 inverters) is the same as the F for n_1 logic stages.

Additional inverters in the logic chain permit sharing the effort over a larger number of stages, which can reduce the total delay. We first find the optimum value of N . If $N > n_1$, we shall have the opportunity of reducing the delay by adding inverters.

The total delay in the N stages is the sum of delays of n_1 logic stages and n_2 inverters. For optimum delay, the stage effort should be equal for all the N stages. Therefore, the stage effort of logic stages as well as the inverters is the same and is $= F^{1/N}$. So the total delay is:

$$\hat{D} = NF^{1/N} + \sum_{i=1}^{n_1} p_i + (N - n_1)p_{inv} \quad (7.40)$$

The first term in Equation 7.40 above is the effort delay of N stages. The sum of parasitic delays of n_1 logic gates gives us the second term. Finally, the parasitic delay of $n_2 = N - n_1$ inverters gives the third term.

We define the optimum stage effort $\rho \equiv F^{1/N}$. Then

$$\hat{D} = N(\rho + p_{inv}) + \sum_{i=1}^{n_1} p_i - n_1 p_{inv} \quad (7.41)$$

Since $\rho \equiv F^{1/N}$, $N = \ln F / \ln \rho$. Therefore,

$$\hat{D} = \frac{\ln F}{\ln \rho}(\rho + p_{inv}) + \sum_{i=1}^{n_1} p_i - n_1 p_{inv} \quad (7.42)$$

It is to be noticed that F , n_1 , p_i and p_{inv} are given constants. We now optimize the total delay with respect to ρ by setting the derivative of \hat{D} with respect to ρ to zero. Differentiating by parts, we get

$$\frac{\partial \hat{D}}{\partial \rho} = 0 = -\frac{\ln F}{(\ln \rho)^2} \cdot \frac{1}{\rho} \cdot (\rho + p_{inv}) + \frac{\ln F}{\ln \rho}(1) \quad (7.43)$$

Therefore,

$$\frac{\ln F}{\ln \rho} = \frac{\ln F}{(\ln \rho)^2} \cdot \frac{1}{\rho} \cdot (\rho + p_{inv}) \quad (7.44)$$

and so,

$$1 = \frac{1}{\rho \ln \rho}(\rho + p_{inv}) \quad (7.45)$$

Which gives

$$\rho + p_{inv} = \rho \ln \rho \quad (7.46)$$

Which can be written as

$$p_{inv} + \rho(1 - \ln \rho) = 0 \quad (7.47)$$

It is interesting to note that this condition is independent of F and the value of ρ is uniquely defined by p_{inv} .

Equation 7.47 cannot be solved in closed form and either iterative solutions or graphical solutions have to be used to determine ρ from p_{inv} . In the special case when $p_{inv} = 0$, we have

$$\rho(1 - \ln \rho) = 0 \quad \text{so } \ln \rho = 1 \quad \text{which gives } \rho = e$$

This corresponds to the case of tapered inverter that we had solved before.

For non-zero values of p_{inv} , Equation 7.47 can be solved iteratively using the Newton Raphson technique. We can write Equation 7.47 as

$$f(\rho) \equiv \rho - \rho \ln \rho + p_{inv} = 0 \quad (7.48)$$

$$\text{Then } f'(\rho) = 1 - \ln \rho - \rho \frac{1}{\rho} = -\ln \rho \quad (7.49)$$

If we have a guess solution g for this equation, the next improved guess according to Newton Raphson technique is:

$$g_{next} = g - \frac{f(g)}{f'(g)} = g + \frac{g - g \ln g + p_{inv}}{\ln g} = \frac{g + p_{inv}}{\ln g} \quad (7.50)$$

We know that for $p_{inv} = 0$, the value of ρ is e . A good guess value to start iterations will be $\rho = 3$. Let us illustrate the iterative method by taking $p_{inv} = 1$. The successive values obtained from Equation 7.50, starting with $\rho = 3$ are: 3.0, 3.6410, 3.5914, 3.5911, 3.5911, ... The value converges to 4 decimal places within 3 to 4 iterations.

Similarly, for $p_{inv} = 0.6$, the successive solutions starting from 3 are: 3.2769, 3.2664, 3.2664
...

We can also solve Equation 7.46 graphically by plotting $\rho + p_{inv}$ and $\rho \ln \rho$ as functions of ρ . The value of ρ at which these two intersect is the solution of the equation.

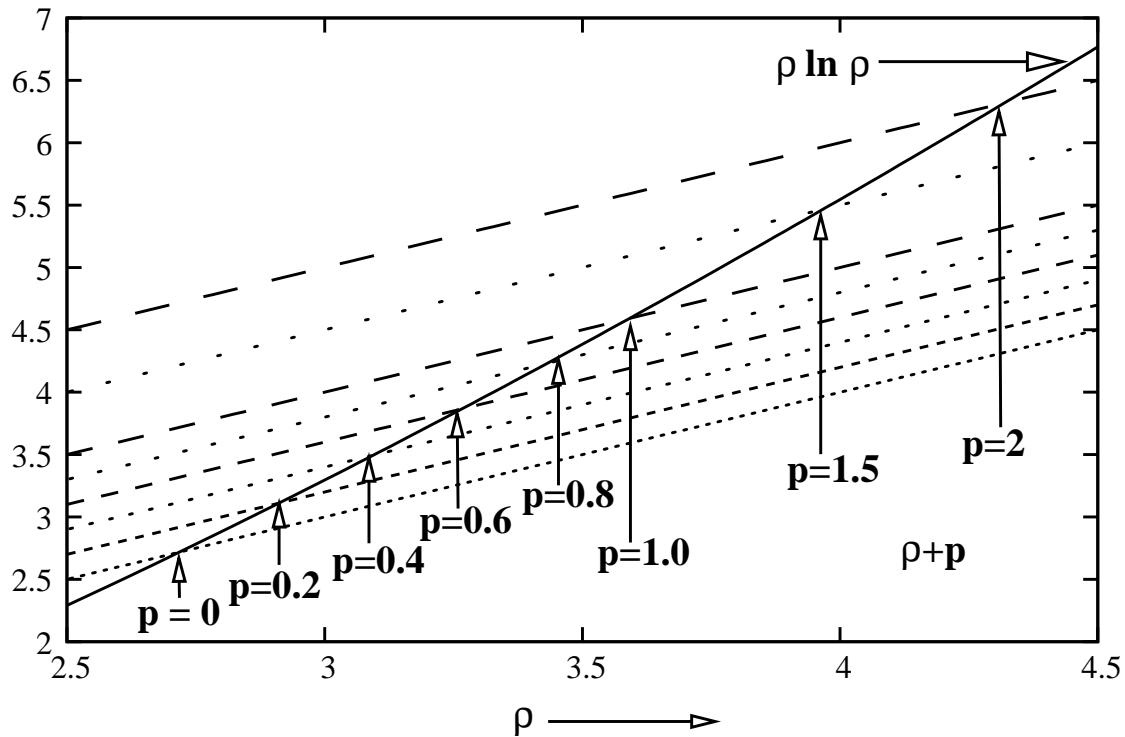


Figure 7.7: Graphical solution of the equation $\rho + p_{inv} = \rho \ln \rho$.

Either way, one can evaluate ρ if the parasitic delay of an inverter is given. Table 7.2 below gives the values of ρ and the corresponding stage delay for several values of p_{inv} .

Once ρ is known, we can evaluate the optimum number of logic stages for a given path effort by $N = \ln F / \ln \rho$. This value will be fractional in general and needs to be zapped to the nearest integer. After this rounding off, the stage effort has to be re-calculated as $f = F(1/N)$. If $N > n_1$ (where n_1 is the number of logic stages which are necessary for implementing the desired logic function), we can insert $N - n_1$ inverter stages to optimize the

p_{inv}	ρ	$\ln \rho$	$d = \rho + p_{inv}$
0	2.718 (e)	1.000	2.718
0.2	2.912	1.069	3.11
0.4	3.093	1.129	3.49
0.6	3.266	1.184	3.87
0.8	3.432	1.233	4.23
1.0	3.591	1.278	4.59
1.5	3.967	1.378	5.47
2.0	4.319	1.463	6.32

Table 7.2: ρ as a function of p_{inv} . The table also gives the optimum delay.

overall delay.

Thus, \hat{f} is the optimum stage effort when the number of logic stages is fixed and known. In this case the stage effort is F^{1/n_1} and the total delay is $n_1 F^{1/n_1} + \sum_{i=1}^{n_1} p_i$ where n_1 is the known number of logic stages.

If we have the freedom of inserting a number of inverters in the logic path to optimize delay, we follow the following procedure:

1. From p_{inv} , find the ideal stage effort ρ by solving $p_{inv} + \rho(1 - \ln \rho) = 0$. This can be done through iterative solutions or graphically as described earlier.
2. Once ρ is known, find the number of stages as $\ln F / \ln \rho$. The nearest integer to the value so found is the optimum number of stages N .
3. If $N > n_1$, insert $(N - n_1)$ inverters anywhere in the logic path. If $N \leq n_1$, take $N = n_1$.
4. The stage effort is now adjusted to $f = F^{1/N}$.
5. Given this value of f , we can start from the last stage and work backwards as earlier to calculate all transistor geometries.
6. For the last stage the output capacitance is known ($=C_L$). The input capacitance can be calculated from Equation 7.35.

$$C_{in_N} = b_N g_N \frac{C_L}{f}$$

This gives the scale factor for this stage from which, geometries of transistors in the last stage can be computed.

7. For each preceding stage, we use the recursive relation 7.36

$$C_{in_i} = g_i \frac{b_i C_{in_{i+1}}}{f}$$

8. From C_{in_i} , we can calculate the scale factor, and hence the geometry of all transistors for this stage.

The optimum number of stages will depend on $F = GBH$. Because N must be an integer, a range of values of F will require the same number of stages. Table 7.5 gives the optimum number of stages for ranges of F values. This table assumes the value of p_{inv} to be 1.0. It is

Path effort F	Optimum No. of stages \hat{N}	Min. Delay \hat{D}	range of values of Stage effort f
0-5.83	1	1.0-6.8	0-5.8
5.83-22.3	2	6.8-11.4	2.4-4.7
22.3-82.2	3	11.4-16.0	2.8-4.4
82.2-300	4	16.0-20.7	3.0-4.2
300-1090	5	20.7-25.3	3.1-4.1
1090-3920	6	25.3-29.8	3.2-4.0
3920-14200	7	29.8-34.4	3.3-4.0
14200-51000	8	34.4-39.0	3.3-3.9
51000-184000	9	39.0-43.6	3.3-3.9
184000-661000	10	43.6-48.2	3.4-3.8

Table 7.3: Optimum number of stages for different F values. This table assumes $p_{inv} = 1$, so $\rho = 3.59$.

interesting to ask how much the delay for a properly optimized circuit is changed by using the wrong number of stages. The answer, as shown in Table 7.4, is that delay is quite insensitive

N/\hat{N}	D/\hat{D}	N/\hat{N}	D/\hat{D}
0.25	7.42	1.4	1.06
0.5	1.46	2.0	1.24
0.7	1.09	3.0	1.62
1/0	1/00	4.0	2.01

Table 7.4: The relative delay of a network, D/\hat{D} , as a function of the relative error in the number of stages used, N/\hat{N} . Assumes $p_{inv} = 0.6$.

to the number of stages, provided the deviation from optimum is not too large. As the table shows, doubling the number of stages from optimum increases the delay only 24%. Using half as many stages as the optimum increases the delay by 46%. Thus one need not slavishly stick to exactly the correct number of stages. It is slightly better to err in the direction of using too many stages than too few. A stage or two more or less in a design with many stages will make little difference, provided proper transistor sizes are used. Only when very few stages are required does a change of one or two stages make a large difference.

Adders

Dinesh Sharma

EE Department
IIT Bombay, Mumbai

October 27, 2020

- 1 Half and Full Adders
- 2 Ripple Carry adder
- 3 Carry Look Ahead
 - Manchester Carry Chain
- 4 Carry Bypass Adder
- 5 Carry Select Adder
 - Stacking Carry Select Adders
- 6 Tree Adders
 - Brent Kung adder
- 7 Serial Adders

Half Adder

The truth table for addition of two bits is:

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

$$\text{sum} = A \cdot \overline{B} + B \cdot \overline{A}$$

$$\text{carry} = A \cdot B$$

- What do we do with the carry?
- Obviously, it must be added to more significant bits.
- So we need an adder with *three* inputs.

Full Adder

Truth Table for the addition of three bits is:

A	B	Cin	Sum	Cout
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Which leads to the following Karnaugh maps:

		AB	00	01	11	10
		Cin	0	1	0	1
A	B	0	0	1	0	1
		1	1	0	1	0

SUM

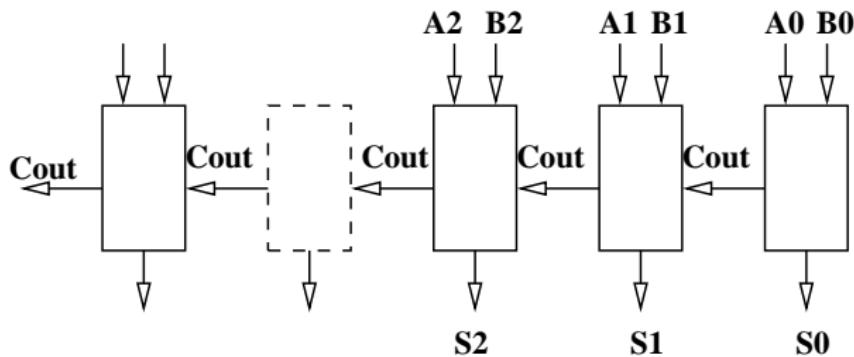
		AB	00	01	11	10
		Cin	0	0	1	0
A	B	0	0	0	1	0
		1	0	1	1	1

CARRY

$$\text{sum} = \overline{A} \cdot \overline{B} \cdot C_{in} + \overline{A} \cdot B \cdot \overline{C_{in}} + A \cdot B \cdot C_{in} + A \cdot \overline{B} \cdot \overline{C_{in}}$$

$$C_{out} = A \cdot B + B \cdot C_{in} + C_{in} \cdot A = A \cdot B + C_{in} \cdot (A + B)$$

Ripple Carry adder



- Carry out of one bit becomes Carry in of the next.
- This architecture is therefore called ripple carry adder.
- The critical delay path of the adder is the carry rippling from one bit to the next.

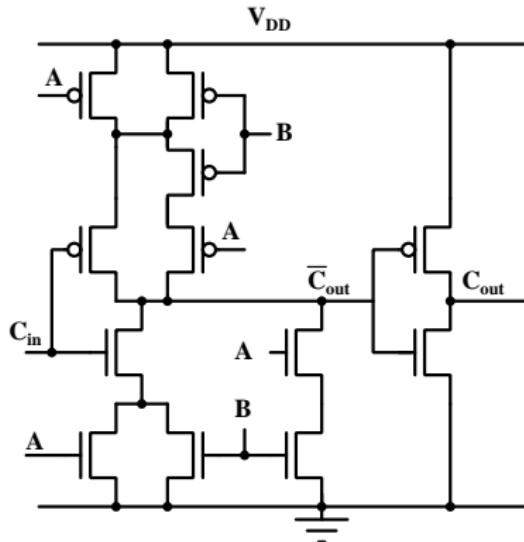
Sum derived from carry

- Because carry is on the critical path, Carry-out must be generated as quickly as possible.
- We need not optimize the delay of generating sum.
- We can in fact generate sum from Carry out.

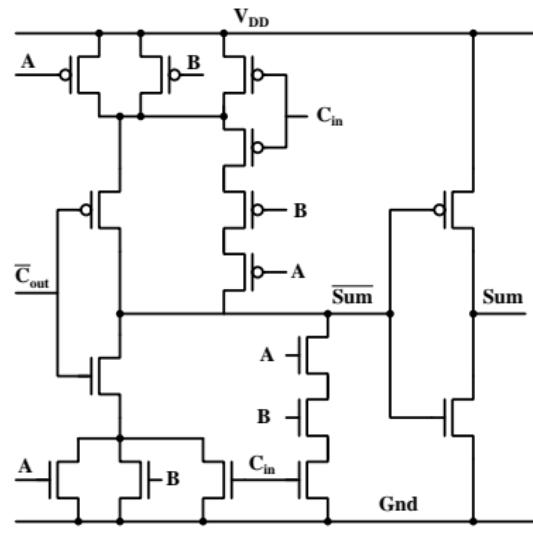
$$\begin{aligned}
 \overline{C_{out}} &= \overline{A \cdot B + C_{in} \cdot (A + B)} \\
 &= (\overline{A} + \overline{B}) \cdot (\overline{C_{in}} + \overline{A} \cdot \overline{B}) \\
 &= \overline{A} \cdot \overline{C_{in}} + \overline{B} \cdot \overline{C_{in}} + \overline{A} \cdot \overline{B} \\
 \overline{C_{out}} \cdot (A + B + C_{in}) &= A \cdot \overline{B} \cdot \overline{C_{in}} + \overline{A} \cdot B \cdot \overline{C_{in}} + \overline{A} \cdot \overline{B} \cdot C_{in}
 \end{aligned}$$

$$\begin{aligned}
 \text{sum} &= A \cdot \overline{B} \cdot \overline{C_{in}} + \overline{A} \cdot B \cdot \overline{C_{in}} + \overline{A} \cdot \overline{B} \cdot C_{in} + A \cdot B \cdot C_{in} \\
 &= \overline{C_{out}} \cdot (A + B + C_{in}) + A \cdot B \cdot C_{in}
 \end{aligned}$$

CMOS Implementation



$$C_{\text{out}} = A \cdot B + C_{\text{in}} \cdot (A + B)$$



$$\text{Sum} = \overline{C}_{\text{out}} \cdot (A + B + C_{\text{in}}) + A \cdot B \cdot C_{\text{in}}$$

Complementation Property

Both Sum and Carry show an interesting symmetry:

$$\begin{aligned}
 \text{sum} &= \overline{A} \cdot \overline{B} \cdot C_{in} + \overline{A} \cdot B \cdot \overline{C_{in}} + A \cdot \overline{B} \cdot \overline{C_{in}} + A \cdot B \cdot C_{in} \\
 \overline{\text{sum}} &= (A + B + \overline{C_{in}}) \cdot (A + \overline{B} + C_{in}) \cdot (\overline{A} + B + C_{in}) \cdot (\overline{A} + \overline{B} + \overline{C_{in}}) \\
 &= (A + A \cdot \overline{B} + A \cdot C_{in} + A \cdot B + B \cdot C_{in} + \overline{C_{in}} \cdot A + \overline{C_{in}} \cdot \overline{B}) \cdot \\
 &\quad (\overline{A} + \overline{A} \cdot \overline{B} + \overline{A} \cdot \overline{C_{in}} + \overline{A} \cdot B + B \cdot \overline{C_{in}} + C_{in} \cdot \overline{A} + C_{in} \cdot \overline{B}) \\
 &= (A + B \cdot C_{in} + \overline{B} \cdot \overline{C_{in}}) \cdot (\overline{A} + B \cdot \overline{C_{in}} + \overline{B} \cdot C_{in}) \\
 &= A \cdot B \cdot \overline{C_{in}} + A \cdot \overline{B} \cdot C_{in} + \overline{A} \cdot B \cdot C_{in} + \overline{A} \cdot \overline{B} \cdot \overline{C_{in}}
 \end{aligned}$$

Thus

$$\begin{aligned}
 \text{sum} &= \overline{A} \cdot \overline{B} \cdot C_{in} + \overline{A} \cdot B \cdot \overline{C_{in}} + A \cdot \overline{B} \cdot \overline{C_{in}} + A \cdot B \cdot C_{in} \\
 \overline{\text{sum}} &= A \cdot B \cdot \overline{C_{in}} + A \cdot \overline{B} \cdot C_{in} + \overline{A} \cdot B \cdot C_{in} + \overline{A} \cdot \overline{B} \cdot \overline{C_{in}}
 \end{aligned}$$

This shows that the **same hardware** that produces sum from A , B and C_{in} , will produce $\overline{\text{sum}}$ if the inputs are changed to \overline{A} , \overline{B} and $\overline{C_{in}}$

Complementation Property

Carry also has the same complementation property.

$$C_{out} = A \cdot B + C_{in} \cdot (A + B)$$

$$\begin{aligned}\text{Hence, } \overline{C_{out}} &= \overline{A \cdot B + C_{in} \cdot (A + B)} = (\overline{A} + \overline{B}) \cdot (\overline{C_{in}} + \overline{A} \cdot \overline{B}) \\ &= \overline{A} \cdot \overline{C_{in}} + \overline{B} \cdot \overline{C_{in}} + \overline{A} \cdot \overline{B}\end{aligned}$$

$$\begin{array}{lll}\text{Thus } & C_{out} &= A \cdot B + C_{in} \cdot (A + B) \\ \text{while } & \overline{C_{out}} &= \overline{A} \cdot \overline{B} + \overline{C_{in}} \cdot (\overline{A} + \overline{B})\end{array}$$

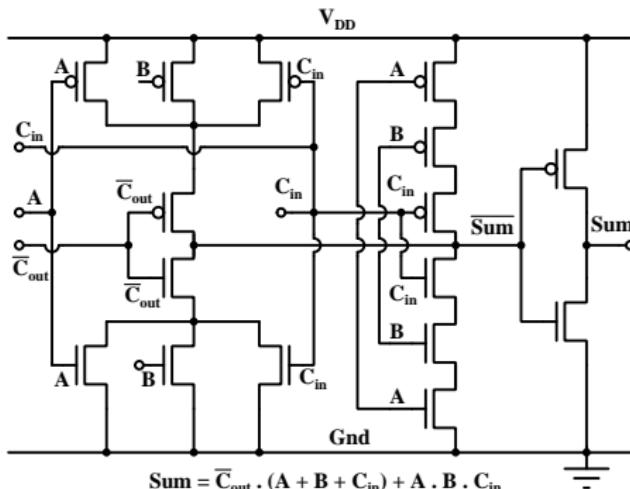
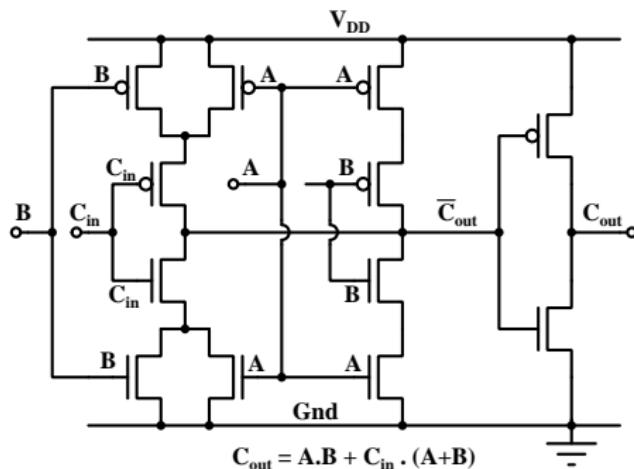
So **the same hardware** which produces C_{out} from A, B and C_{in} , will produce $\overline{C_{out}}$ from $\overline{A}, \overline{B}$ and $\overline{C_{in}}$.

Making use of the symmetry property

- In CMOS implementation, we interchange series and parallel configurations for the n and p channel transistors.
- This is to ensure that the pull up and pull down circuits are complementary.
- However, for sum and carry functions, we see that these functions are their own complements.
- Therefore, for implementing sum and carry, we can use the **same** configuration for n and p channel transistors.
- We use this to reduce the number of series connected transistors in pull up/pull down networks.

Mirror gates for Adders

- By making use of symmetry property of sum and carry, it is possible to simplify the implementations.



These are called mirror gates because the n and p transistors have the **same** series parallel combination.

This is highly unusual.

Speeding up the Ripple Carry Adder

- The worst case delay of the ripple carry adder is linear in number of bits to be added.
- To reduce the delay per stage, we can eliminate the inverter from the carry output.
- All even bit adders accept a , b and C_{in} as inputs. The mirror gate without inverter gives C_{out} as the output.
- All odd bit adders accept \bar{A} , \bar{B} and \bar{C}_{in} as inputs and thus produce C_{out} as output.
- Outputs of all bits are now compatible with inputs of the next stage.

Speeding up the Ripple Carry Adder

- Extra inverters are required to produce \overline{A} , \overline{B} and at the outputs to produce the proper result. However, these are not on the critical path, and do not add to the worst case delay.
- Extreme care needs to be taken in layout to ensure that the loading on the tree gate producing carry output is as small as possible.

Terms Independent of Carry

- Carry propagation is the critical path for a multi-bit adder.
- To speed up the adder, we would like an architecture where logic terms are classified as those dependent on carry and those which do not depend on carry.
- To speed up the adder, we would like to pre-compute all terms which do not depend on carry.
- Now when the carry arrives, we quickly compute the output carry and pass it on to the next stage.

Carry Independent Terms

We would like to analyze what information can be pre-computed from A_i and B_i , which will help us in generating C_{out} quickly from C_{in} .

- When $A_i = 0$ and $B_i = 0$, C_{out} is 0, independent of C_{in} . We define this condition as 'Kill'. $K = \overline{A} \cdot \overline{B}$
- Similarly, when $A_i = 1$ and $B_i = 1$, C_{out} is 1, independent of C_{in} . We define this condition as 'Generate': $G = A \cdot B$.
- Only when $A_i = 0$ and $B_i = 1$ or when $A_i = 1$ and $B_i = 0$, we need to wait for C_{in} to compute C_{out} . In both these cases, $C_{out} = C_{in}$.
- We call this condition as 'Propagate', and define $P = A \cdot \overline{B} + \overline{A} \cdot B$.

Using Carry Independent Terms

We define $K = \overline{A} \cdot \overline{B}$, $G = A \cdot B$ and $P = A \oplus B$

Exactly one of K, G or P is true at any time.

When $K = 1$, C_{out} is 0, independent of C_{in} .

When $G = 1$, C_{out} is 1, independent of C_{in} .

When $P = 1$, $C_{out} = C_{in}$.

P needs to be computed using an xor gate, which can be slow. However, the only difference between xor and or logic is when both inputs are 1, i.e. $G = 1$.

If we can ensure that G forces C_{out} to 1 irrespective of P, we can use the simpler ‘or’ logic to compute P.

Carry Look Ahead

C_{in} for bit $i+1$ is the C_{out} of bit i .

So we can write $C_{i+1} = G_i + P_i \cdot C_i$

Notice that the Kill signal is not required.

If $G_i = 0$, $C_{i+1} = A \oplus B = A + B$ when $G = A \cdot B = 0$

If $G_i = 1$, $C_{i+1} = 1$, and the value of P_i does not matter anyway.

So we can use $P = A + B$ instead of $P = A \oplus B$.

Now, we have the sequence:

$$C_{i+1} = G_i + P_i \cdot C_i = G_i + P_i \cdot G_{i-1} + P_i \cdot P_{i-1} \cdot C_{i-1} = \dots$$

and so on, till we reach C_0 .

Since all G_i , P_i and C_0 can be computed in parallel on arrival of the inputs, we can compute all sum and carry terms independently if we do not mind the added complexity.

Carry Look Ahead

$$C_{i+1} = G_i + P_i \cdot C_i = G_i + P_i \cdot G_{i-1} + P_i \cdot P_{i-1} \cdot C_{i-1} = \dots$$

Unfortunately, static implementation of these gates has almost as much delay as the ripple carry implementation.

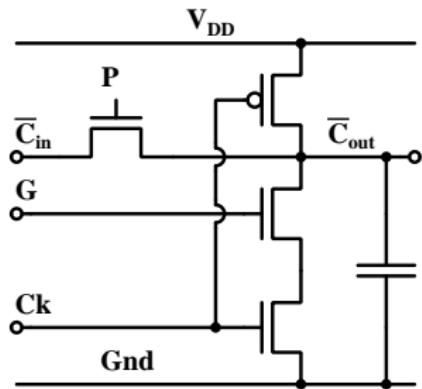
Therefore, the static implementation of computation of sum and carry terms as a logic expression depending on all A_i, B_i and C_0 is rarely used.

We can use these expressions for blocks of a small number of bits (say 4) and then propagate carry over these blocks.

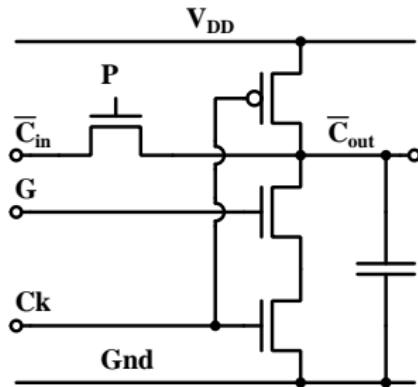
Manchester Carry Chain

Static implementation of look ahead carry is not really fast if we try to look ahead by a large number of bits, because the logic becomes very complex.

A dynamic implementation is useful and is widely used. It is known as the Manchester Carry Chain.



Manchester Carry Chain



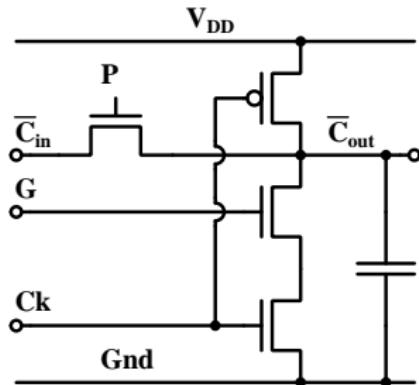
When the clock is low, the output is unconditionally charged by the pMOS.

When the clock goes high, the output will be pulled low if $G = 1$ or if $P = 1$ and $\bar{C}_{in} = 0$.

In all other cases, the output will remain high. Thus this circuit implements the required logic.

This circuit can be concatenated for all bits and since P and G are ready before \bar{C}_{in} arrives, the carry quickly ripples through from bit to bit.

Manchester Carry Chain as Carry Look Ahead



Notice that the nMOS logic can be interpreted as:

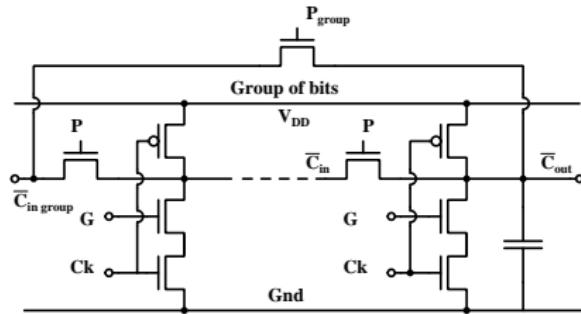
$$\overline{P.Cin + G}$$

where C_{in} itself has been recursively generated by similar logic.

- As in the static case, there is a limit to the number of bits which can be so connected.
- If $P = 1$ for many successive bits, the discharge path is through series connected pass transistors of all these gates. The discharge time for this critical path has an n^2 dependence.

Carry Bypass Adder

- The worst case for addition occurs when $P = 1$ for all bits and carry has to ripple through all bits.
- In carry bypass adder, we form groups of bits and if $P = 1$ for all members of a group, we pass on the carry input to this group directly to the input of the next group, without having to ripple through each bit.
- This improves the worst case delay of the adder.

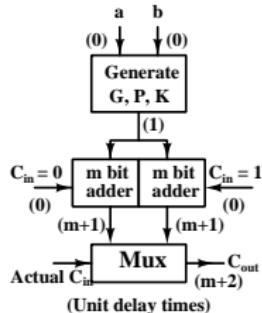


Carry Select Adder

An m bit carry select adder can be constructed as follows:

- We first compute the generate/propagate/kill signals for each bit (in parallel) from the input bits. Assuming unit gate delay model, this takes one unit of time.
- We use two m bit carry bypass adders. One of the adders assumes the carry input C_{in} to be 0, while the other assumes C_{in} to be 1. The two adders work in parallel and each takes m units of time.
- We now use a multiplexer controlled by the actual C_{in} to select the correct C_{out} . This takes one unit of time.
- The C_{out} of one such m bit adder will be used as the select input of the multiplexer of the next.
- The sum output of each bit is derived from P and C_{out} signals for the corresponding bit and appear one unit of time after C_{out} is available.

Carry Select Adder



The two m bit sub-adders assume the carry to be 0 or 1 respectively.

Times of availability of various signals are noted in parentheses in the diagram.

- The two alternatives for the carry output are ready at $(m+1)$ units of time.
- If the actual C_{in} is available at n units of time, the output will be available at $(m+2)$ or $(n+1)$, whichever is later.
- In case of 4 bit adders, this is at 6 units of time or at C_{in} arrival + 1, whichever is later.

Stacking in Carry Select adders

- The sub-adders in carry select adder can use any architecture.
- They could be Manchester carry chains, carry bypass or ripple carry adders.
- Obviously, these sub adders should not be very long, otherwise, their outputs will be ready after a long time and we shall lose the advantage of carry bypass additions.
- Then, how do we make long adders using carry select?

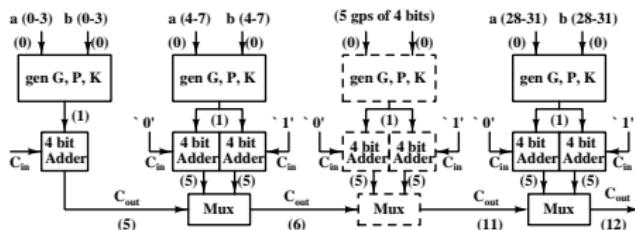
This is done by stacking several smaller carry select adders.

Linear Stacking

- We could stack several identical carry select adders.
- There is no need for carry select in the first stage, as C_{in} for this stage is available simultaneously with A_i and B_i .
- Every subsequent stage will have two sub-adders, one assuming $C_{in} = 0$, the other assuming $C_{in} = 1$.
- The correct output will be selected by the actual C_{in} when it arrives.
- Thus, after the first stage, each group of m bit adders will add only one unit of delay.
- This is much faster. However, the delay is still linear in number of bits.

Linear stacking: Example

A 32-bit adder made by cascading 8 4-bit carry select adders.



The sum generation will take another unit of time, so the overall results will be available in 13 units of time.

Bits	cy in	alt cy.s	cy out
0-3	0	-	5
4-7	5	5	6
8-11	6	5	7
12-15	7	5	8
16-19	8	5	9
20-23	9	5	10
24-27	10	5	11
28-31	11	5	12

Square-root Stacking

- Can we speed up the adder if we don't use the same no. of bits in every stage?
- In linear stacking, since all adders are identical, they are ready with their alternative outputs at the same time.
- But the carry arrives later and later at each successive group of carry select adders.
- We could have used this extra time to add up more bits in the later stages, and still be ready with the alternative results before carry arrives!
- Since the carry arrives one unit of time later at each successive group, each successive group could be longer by one bit.

Square-root Stacking

- We can do more bits of addition in the same time, if each successive stage is 1 bit longer than the previous one.
- Thus, the number of bits which can be added is given by

$$m = n_0 + n_0 + (n_0 + 1) + (n_0 + 2) + \dots = n_0 + \frac{s(n_0 + n_0 + s - 1)}{2}$$

where s is the number of stages following the first one without carry select.

- The total delay will be $n_0 + 1$ for the first stage. Each subsequent stage takes just 1 unit of time since the candidates for selection are available just in time. Thus the time taken is just $n_0 + s + 1$ units. When $s \gg n_0$, we have $m \approx s^2/2$, while the time taken is nearly s .
- Thus the time taken to add m bits is $\approx \sqrt{2m}$

Square-root Stacking: Example

For a 32 bit adder, we could use a distribution like: 4,4,5,6,7,6.

Bits	carry in	carry alternatives	carry out
0-3	0	-	5
4-7	5	5	6
8-12	6	6	7
13-18	7	7	8
19-25	8	8	9
26-31	9	7	10

Our sum will be ready at 11 - which is faster. This gain will be much higher for wider additions.

Tree Adders

- Tree adders use the idea of carry look ahead addition.
- However, these do not try to implement the complex logic expressions which result from looking ahead all the way.
- Instead, these build up the logic in a tree like structure, where each node performs simple logic operations on the results of the previous node.
- Because of the tree structure used in this, the delay is of the order of $\log n$ for an n bit adder.

Carry Look Ahead

For carry look ahead, we had defined

$$K = \overline{A} \cdot \overline{B}, \quad G = A \cdot B \quad \text{and} \quad P = A \oplus B.$$

P , G and K can be computed without waiting for C_{in} .

when $K = 1$ $C_{out} = 0$ irrespective of C_{in} .

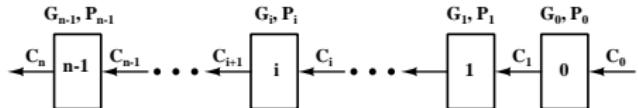
when $G = 1$ $C_{out} = 1$ irrespective of C_{in} .

When $P = 1$ $C_{out} = C_{in}$

This is the only case when we must wait for C_{in}
in order to compute C_{out}

First order P and G values

The least significant bit is indexed as 0 and the most significant bit as $n - 1$.



i 'th bit accepts C_i as input carry and produces C_{i+1} as output carry.

Output of the i 'th stage is: $C_{i+1} = G_i + P_i \cdot C_i$

C_i itself is the output of cell no. ($i-1$). Hence,

$$\begin{aligned} C_{i+1} &= G_i + P_i \cdot (G_{i-1} + P_{i-1} \cdot C_{i-1}) \\ &= (G_i + P_i \cdot G_{i-1}) + P_i \cdot P_{i-1} \cdot C_{i-1} \end{aligned}$$

Second order P and G

$$C_{i+1} = (G_i + P_i \cdot G_{i-1}) + P_i \cdot P_{i-1} \cdot C_{i-1}$$

$G_i + P_i \cdot G_{i-1}$ and $P_i \cdot P_{i-1}$ are independent of input carry.

Labeling the single bit G and P values for the i 'th cell as G_i^1 and P_i^1 , we can define

$$G_{i,i-1}^2 \equiv G_i^1 + P_i^1 \cdot G_{i-1}^1 \quad \text{and} \quad P_{i,i-1}^2 \equiv P_i^1 \cdot P_{i-1}^1$$

This permits us to write

$$C_{i+1} = G_{i,i-1}^2 + P_{i,i-1}^2 \cdot C_{i-1}$$

which evaluates C_{i+1} directly from C_{i-1} .

Second order P and G

We have

$$C_{i+1} = G_{i,i-1}^2 + P_{i,i-1}^2 \cdot C_{i-1}$$

Thus we can compute C_{i+1} from C_{i-1} directly using the same logic as before, except that we use $G_{i,i-1}^2$ and $P_{i,i-1}^2$ instead of G_i^1 and P_i^1 .

Effectively, we have created a cell which is equivalent to two original adder cells and the carry can be passed in groups of 2 bits across the adder.

Notice that the logic needed to compute $G_{i,i-1}^2$ from G_i^1 and P_i^1 is the same as that used to compute the output carry from input carry using G and P values of any order.

Higher order P and G

Continuing the same process, we can combine two second order G and P values to compute third order G and P , which will permit computation of C_{i+1} directly from C_{i-3} .

$$\begin{aligned} G_{i,i-3}^2 &\equiv G_{i,i-1}^2 + P_{i,i-1}^2 \cdot G_{i-2,i-3}^2 \quad \text{and} \quad P_{i,i-3}^3 \equiv P_{i,i-1}^2 \cdot P_{i-2,i-3}^2 \\ C_{i+1} &= G_{i,i-3}^3 + P_{i,i-3}^3 \cdot C_{i-3} \end{aligned}$$

All G and P values are independent of input carry and can be computed in constant time from A_i and B_i .

Carry can now be passed over groups of 4 bits using $G_{i,i-3}^3$ and $P_{i,i-3}^3$.

Higher order P and G

- The group size over which the carry can be computed directly *multiples* by two each time we use a higher order for G and P values.
- On the other hand, the time to compute the required higher order G and P values *increments* by one gate delay.
(logic $A + B \cdot C$ for G and $A \cdot B$ for P).
- This results in the ultimate time to generate the final carry being logarithmic in the number of bits being added.

Logarithmic Adders

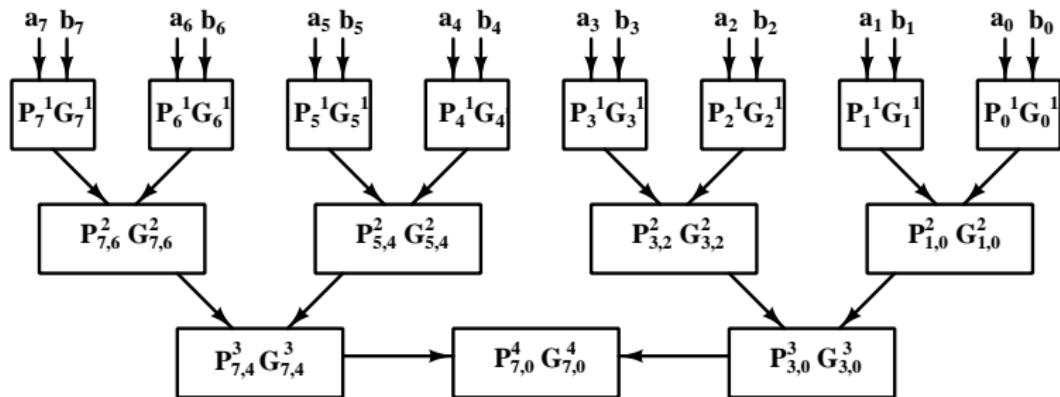
- Once the highest order P and G values have been generated, the final carry can be computed in one step from the input carry.
- In principle, we do not need the internal carries at each bit for the final result.
- However, addition is not complete unless all the sum bits have also been generated. The sum bits are given by:
$$S_i = A_i \oplus B_i \oplus C_i = P_i \oplus C_i$$
- If we want to use this relation, we do need the internal bit-wise carries.

Logarithmic Adders

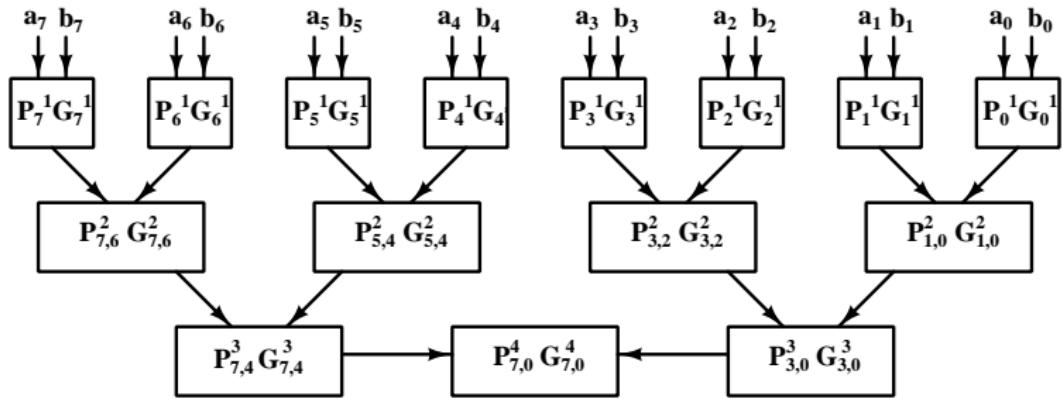
- In logarithmic adders, internal bit-wise carry may be available *after* the final carry.
- The critical path now is not the generation of the final carry, but that of bit-wise sums.
- Different architectures have been described in literature for the order of computation of G , P , C_{out} and Sum bits.
- All of these compute the final result in times which are logarithmic functions of the number of bits.
- For wide adders, these can be much faster than other architectures.

Brent Kung adder

- The Brent Kung tree adder is a logarithmic adder of low complexity.
- Values of P and G are computed in a tree fashion.
- The figure below shows the generation of P and G values for an 8 bit adder.



Brent Kung adder



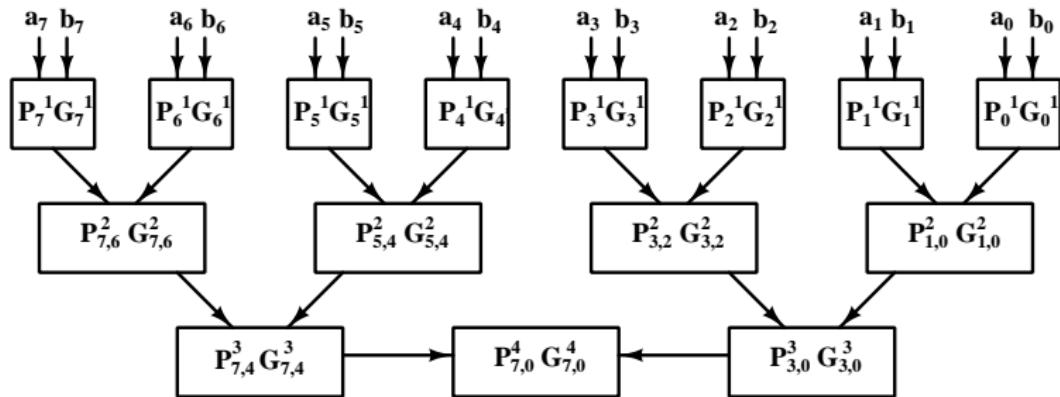
we first calculate P_i^1, G_i^1 , with $i = 0 \dots 7$.

$$G_i = A_i \cdot B_i, \quad P_i = A_i \oplus B_i$$

Next, using these values, we can generate $P_{2i+1,2i}^2, G_{2i+1,2i}^2$ with $i = 0 \dots 3$.

$$G_{2i+1,2i}^2 = G_{2i+1}^1 + P_{2i+1}^1 \cdot G_{2i}^1, \quad P_{2i+1,2i}^2 = P_{2i+1}^1 \cdot P_{2i}^1$$

Brent Kung adder



In the next step, we use second order P, G values to generate $P_{4i+3,4i}^3, G_{4i+3,4i}^3$ with $i = 0, 1$.

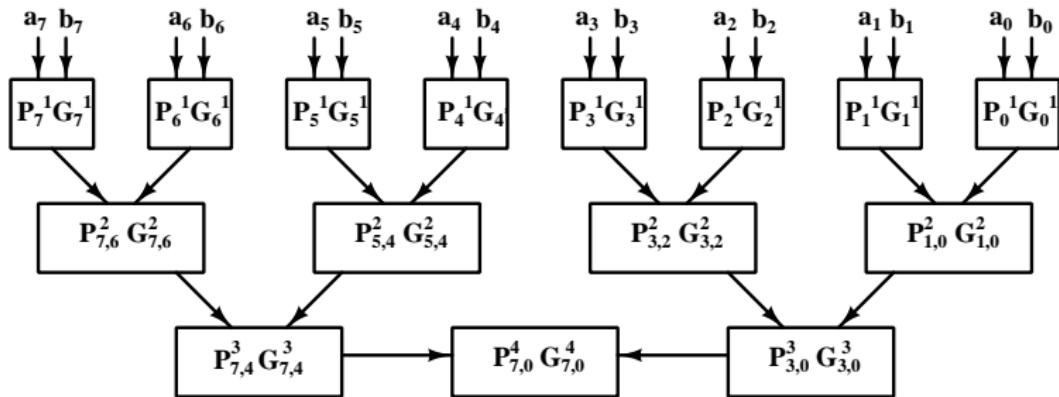
$$G_{7,4}^3 = G_{7,6}^2 + P_{7,6}^2 \cdot G_{5,4}^2,$$

$$P_{7,4}^3 = P_{7,6}^2 \cdot P_{5,4}^2$$

$$G_{3,0}^3 = G_{3,2}^2 + P_{3,2}^2 \cdot G_{1,0}^2,$$

$$P_{3,0}^3 = P_{3,2}^2 \cdot P_{1,0}^2$$

Brent Kung adder



Finally, using $G_{4i+3,4i}^3$ and $P_{4i+3,4i}^3$ (with $i = 0, 1$) we can compute $P_{7,0}^4, G_{7,0}^4$.

$$G_{7,0}^4 = G_{7,4}^3 + P_{7,4}^3 \cdot G_{3,0}^3$$

$$P_{7,0}^4 = P_{7,4}^3 \cdot P_{3,0}^3$$

Brent Kung adder

Once P and G terms of various orders are known, we can compute the values of carry outputs which depend on these and the input carry C_0 , which is available at $t = 0$.

$$C_1 = G_0^1 + P_0^1 \cdot C_0, \quad C_2 = G_{1,0}^2 + P_{1,0}^2 \cdot C_0$$

$$C_4 = G_{3,0}^3 + P_{3,0}^3 \cdot C_0, \quad C_8 = G_{7,0}^4 + P_{7,0}^4 \cdot C_0$$

When these carry values are valid, the other carry values which depend on these can be generated.

Brent Kung adder

Once C_1, C_2, C_4 and C_8 have been generated, we can produce internal carries which depend on these.

$$C_3 = G_2^1 + P_2^1 \cdot C_2, \quad C_5 = G_4^1 + P_4^1 \cdot C_4 \quad C_6 = G_{5,4}^2 + P_{5,4}^2 \cdot C_4,$$

Finally, C_7 can be generated from C_6 .

$$C_7 = G_6^1 + P_6^1 \cdot C_6$$

With all carry values generated, the corresponding sum values can be calculated using the relation $\text{Sum}_i = P_i^1 \oplus C_i$.

Serial Adders

Up to now, we have been concerned with making fast adders, even at the cost of increased complexity and power.

In many applications, speed is not as important as low power consumption and low cost.

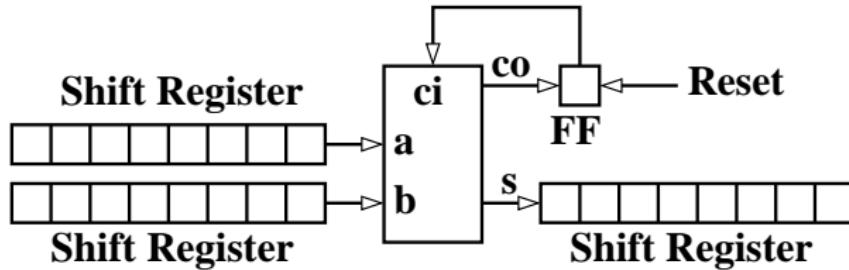
Serial adders are an attractive option in such cases.

A single full adder is used.

If numbers to be added are available in parallel form, these can be serialized using shift registers.

Serial Adders

- A single full adder adds the incoming bits. Bits to be added are fed to it serially, LSB first.
- The sum bit goes to the output while carry is stored in a flip-flop.
- Carry then gets added to the more significant bits which arrive next.
- Output can be converted to parallel form if needed, using another shift register.



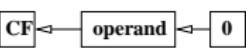
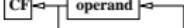
Shift and Rotate Operations

Dinesh Sharma

EE Department
IIT Bombay, Mumbai

October 27, 2020

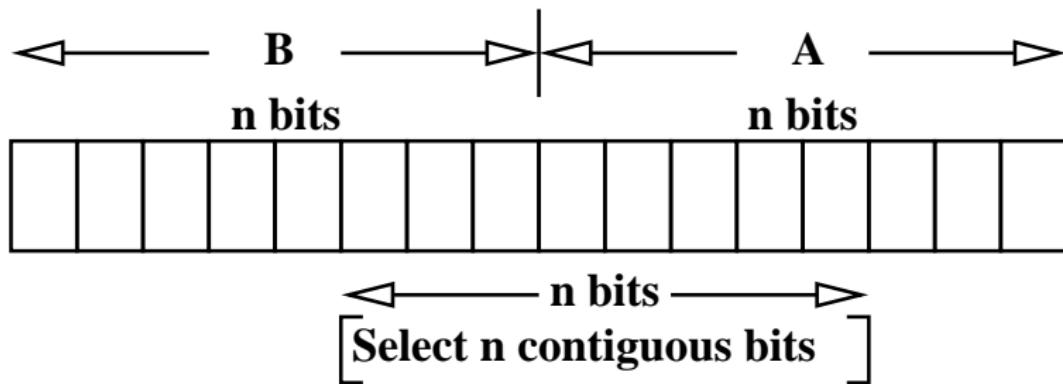
Shift and Rotate Operations

SHL	op, count	
SAL	op, count	Same as SHL
SHR	op, count	
SAR	op, count	
ROL	op, count	
ROR	op, count	
RCL	op, count	
RCR	op, count	

- This can be implemented by a bi-directional shift register.
- We have to add circuits to choose the value and point of entry of new bits.
- This can be very slow for a large number of shifts.
- Ideally, we would like a shifter which produces the result in a single clock cycle.

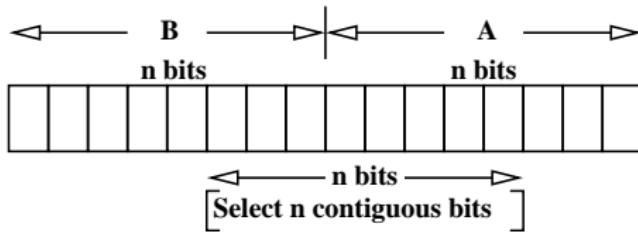
Shift/Rotate as Select Operations

- No “computation” is being carried out during shift/rotate. So why should we spend a large number of clock cycles for it?
- We can view Shift/Rotate as a selection operation.



We just have to choose B and A appropriately to implement a particular shift or rotate operation.

Shift/Rotate as Select Operations



- For all rotate operations, $B=A=\text{data}$
- For Shift Left, $B=\text{data}$, $A=0$.
- For Logical Shift Right, $B=0$, $A=\text{data}$.
- For Arithmetic Shift Right, $B=\text{replicated MSB of data}$, $A=\text{data}$.

Of course we do not actually copy data bits to A/B.

Each output bit is produced by a mux which picks out the correct input data bit.

Barrel Shifters

- Shifters which produce outputs as select operations are called barrel shifters.
- The name comes from viewing the inputs as well as outputs as a circular arrangement of bits.
- The shifter then connects the input circle to the output circle like the sections of a barrel.
- A brute force implementation will require n multiplexers of n bits each, where the control inputs for each multiplexer are generated from the amount and type of shift/rotate.
- This is quite complex and puts a heavy load on data bits.

Logarithmic Barrel Shifters

- The brute force barrel shifter places a heavy load on input data lines because each input bit is a candidate for each output position.
- The control logic is complex because the amount of shift is variable.
- The loading on data lines and control logic complexity can be reduced if we break up the shift process into parts.
- We can carry out shifts in different stages, each stage corresponding to a single bit of the binary representation of the **shift amount**.
- Thus a shift by 6 (binary: 110) will be carried out by first doing a 4 bit shift and then a 2 bit shift.

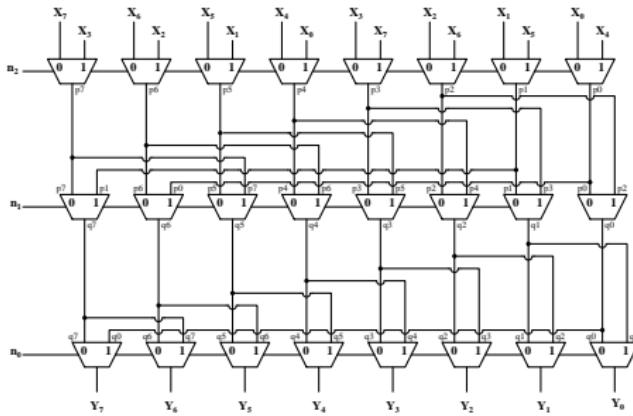
Logarithmic Barrel Shifters

- We need n bits to represent a maximum shift amount of $2^n - 1$ places.
- So the number of bits to express the shift amount (and hence the number of shift stages required) is logarithmic in the maximum shift desired.
- That is why such shifters are called Logarithmic Barrel Shifters.
- We can optionally buffer the outputs after each stage.

Logarithmic Barrel Shifter Stages

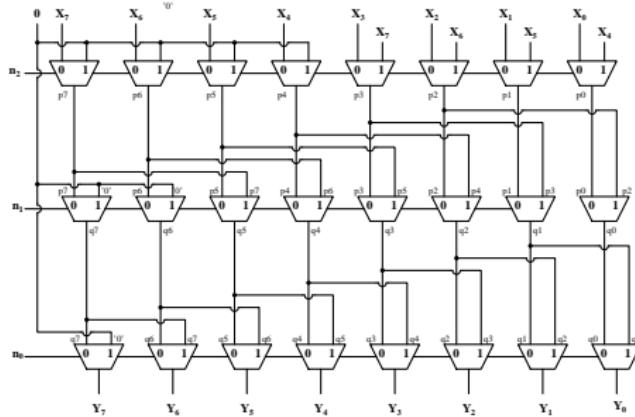
- Bit i of the **shift amount** represents
 - no shift (if it is 0)
 - a constant shift by 2^i places (if it is 1).
- If the shift amount is fixed, we do not need any electronics. The output can just be wired from the input bits.
- Using a 2 way mux controlled by bit i of **shift amount**, we can choose either the unshifted operand bit or the operand bit 2^i places away from it.
- This can be done for all bits of the operand in parallel.
- This constitutes one stage of the logarithmic shifter.
- The output can then be shifted again in the next stage, controlled by the next significant bit.

Right Rotate for an 8 bit Operand



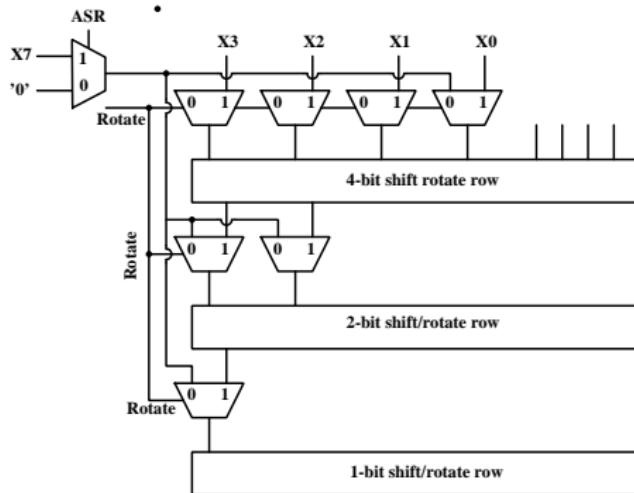
- Each input bit drives just two muxes, each with just 2 inputs.
- At each stage, the muxes select either the unshifted bit or a bit 2^n places from it.
- 3 stages are required for 0 to 7 bits of shift.

8 bit Logical Shift Right



- If we need a shift instead of a rotate, we feed a 0 instead of the corresponding bit.
- This has to be done for 4 muxes in the first stage, 2 in the second stage and 1 in the last stage.

Combining Rotate and Shift

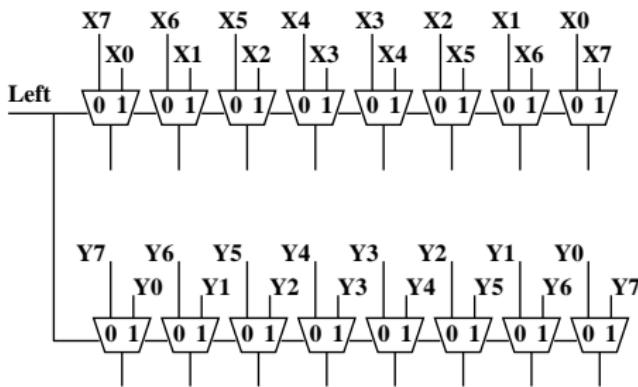


- We can combine the circuits for rotate and shift functions by putting muxes where different inputs need to be presented for the two functions.
- We can include the Arithmetic Shift function by choosing between 0 or X_7 as the bit to be inserted.

Rotate and Shift by Masking

- We can also combine the rotate and shift functions by masking.
- We use the rotate function, which does not lose any information.
- Now we can mask n bits at the left to 0 if a right shift operation was desired instead.
- In case of an arithmetic shift, n bits on the left have to be set to the same value as X7.
- Shift/Rotate Left case is similar, except that the Logical and Arithmetic shifts are the same.

Combining Left and Right Shift/Rotate



- We can use the same hardware for left and right shift/rotate operations.
- This can be done by adding rows of muxes at the input and output which reverse the order of bits.

Combining Left and Right Shift/Rotate

- We can also make use of the fact that a left rotate by n places is the same as a right rotate by $2^n - n$ places.
- $2^n - n$ is just the 2's complement of n .
- By presenting the 2's complement of n at the mux controls, we can convert a right rotate to a left rotate.
- This can be followed by a mask operation, if a shift operation was required, rather than a rotate.

Multiplier Circuits

Dinesh Sharma

EE Department
IIT Bombay, Mumbai

November 2, 2020

1 Shift and Add Multipliers

2 Array Multipliers

3 Speeding up Multipliers

- Booth Encoding
- Adding Partial Products
- Wallace Multipliers
- Dadda Multipliers

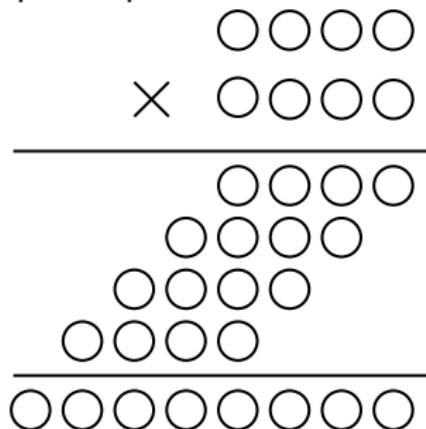
4 Multiply and Accumulate circuits

5 Serial Multipliers

- Bit Serial Multipliers
- Row Serial multipliers

Shift and Add Multipliers

An obvious way for implementing multipliers is to replicate the paper and pencil procedure in hardware.



- Initialize the product to 0, extend multiplicand to left by n bits filled with 0s.
- If the least significant bit of the multiplier is 1, add the multiplicand to product, else do nothing.
- Shift the multiplier right by one bit.
- Shift the multiplicand left by one bit.
- Repeat for n bits

Shift and Add Multipliers

- Each term being added to form the product is called a partial product.
- The name “partial product” is also used for individual bits of the terms being added - so beware!
- The paper-pencil procedure requires $n-1$ additions to a $2n$ bit accumulator.
- This uses a single adder, but takes long to complete the multiplication. A 32×32 multiplication will require 31 addition steps to a 64 bit accumulator.
- Multiplication can be made faster by using multiple adders and adding terms in a tree structure.

Array Multiplier

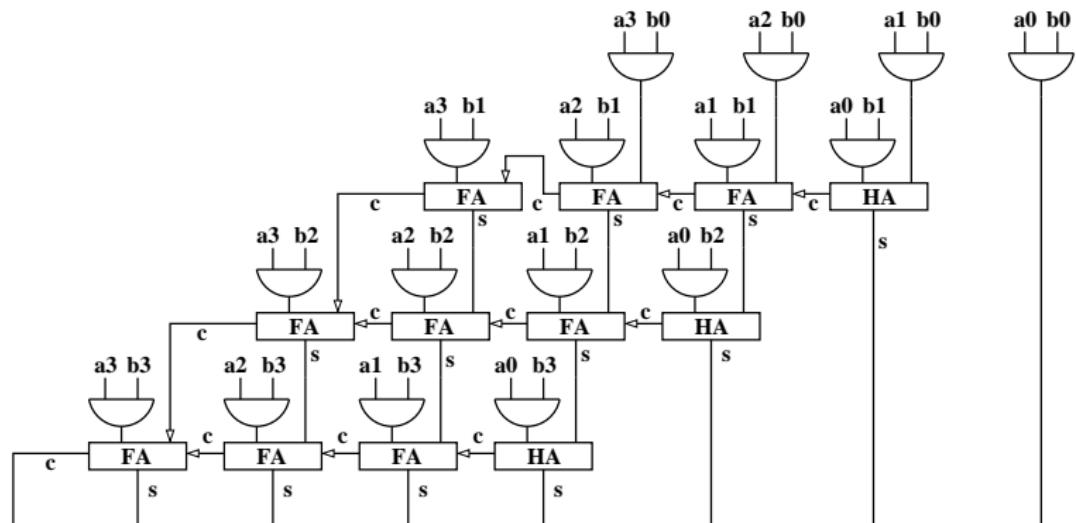
Suppose we want to multiply two n-bit numbers A and B, where

$$A = \sum_{i=0}^{n-1} 2^i a_i \quad B = \sum_{j=0}^{n-1} 2^j b_j$$

- We can regard all bits of the partial products as an array, whose (i,j) th element is $a_i \cdot b_j$. Notice that each element is just the AND of a_i and b_j .
- **All** elements of the array are available in parallel, within one gate delay of arrival of A and B.
- We can now use an array of full adders to produce the result. One input of each adder is the sum from the previous row, the other is the AND of appropriate a_i and b_j .
- This architecture is called an array multiplier.

Array Multiplier

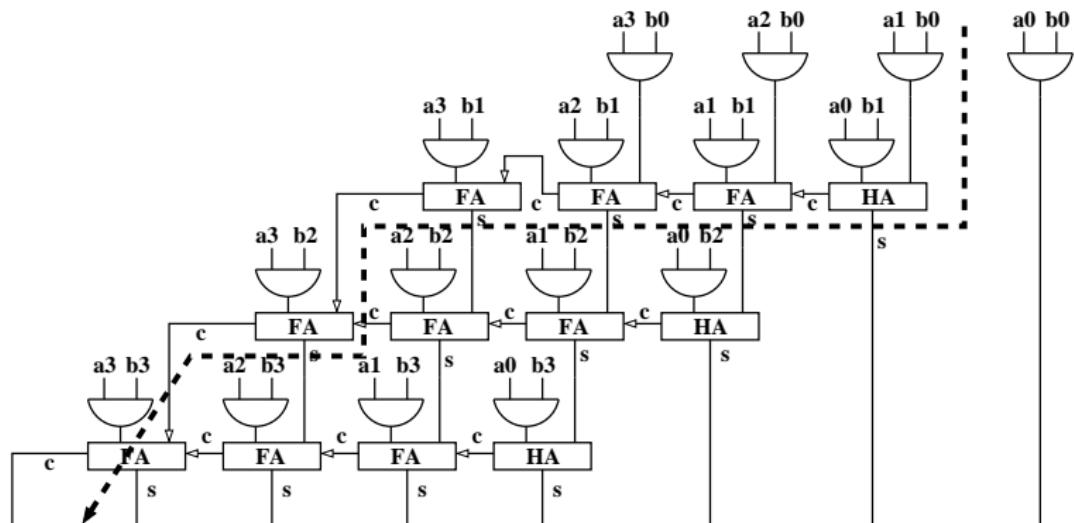
A 4X4 array multiplier is shown below.



Half adders can be used at the right end.

Critical Path through Array Multiplier

The critical path through a 4X4 array multiplier is shown below.



The critical path involves carry as well as sum outputs!

Speeding up Multipliers

The array multiplier has a regular layout with relatively short connections. However, it is still rather slow.

How can we speed up a multiplier?

There are two possibilities:

- Somehow reduce the number of partial products to be added. For example, could we multiply 2 bits at a time rather than 1?
- Since we have to add more than two terms at a time, use an adder architecture which is optimized for this.

Booth Encoding

Booth Encoding reduces the number of partial products by multiplying 2 bits at a time.

- Let the multiplicand be A and the multiplier B.
- Rather than multiplying A with successive bits of B, we can multiply it with two bits of B at a time.
- Depending on the two bits being 00, 01, 10 or 11, the partial product will be 0, A, 2A or 3A.
- 0 and A can be produced trivially.
- 2A can be produced easily by a left shift of A.
- Generating 3A presents a problem!

Booth Encoding

Multiplying the Multiplicand A by 2 bits of the multiplier at a time requires the generation of 0, A, 2A or 3A as partial products. Generating 0, A or 2A is easy.

- 3A cannot be generated directly. However, 3A can be expressed as $4A - A$.
- The task of adding $4A$ is passed on to the next group of 2 bits of the multiplier.
- Since the place value of the next group of 2 bits is 4 times the current one, adding $4A$ to the product is equivalent to adding 1 to the next group of 2 bits of the multiplier.
- $-A$ can be generated from A , using an adder/subtractor rather than an adder for accumulating the sum of partial products.

Modified Booth Encoding

To simplify the logic for deciding whether an additional $4A$ should be added on behalf of the less significant 2 bits in the multiplier, we express $2A$ also as $4A - 2A$.

- Since we anyway have an adder-subtractor, this requires no additional resources.
- The modified logic is:
 - for 00, do nothing.
 - For 01, add A .
 - for 10, subtract $2A$, ask the next group to add $4A$.
 - for 11, subtract A , ask the next group to add $4A$.
- Now the next group can just look at the more significant bit of the previous group and add 1 to the multiplier if it is '1'.

Modified Booth Encoding

- The partial product generator looks at the current 2 bits and the MSB of the previous group of 2 bits to decide its action.
- Thus, we scan the multiplier 3 bits at a time, with one bit overlapping.
- For the first group of 2 bits, we assume a 0 to the right of it.
- After handling the previous group, the multiplicand is shifted left by 2 positions. Thus, it has already been multiplied by 4.
- Therefore, adding 4 A on behalf of the previous group is equivalent to adding 1 to the multiplier corresponding to the current group.

Modified Booth Encoding

The following table summarizes the effective multiplier for generating the partial product.

Current 2-bits	Multiplier for these	Previous MSBit	Pending Increment	Total Multiplier
00	0	0	0	0
01	+1	0	0	+1
10	-2	0	0	-2
11	-1	0	0	-1
<hr/>				
00	0	1	+1	+1
01	+1	1	+1	+2
10	-2	1	+1	-1
11	-1	1	+1	0

Notice that a 111 in the 3 bit group being scanned requires no work at all.

Modified Booth Encoding

3-bits	Multiplier
000	0
001	+1
010	+1
011	+2
100	-2
101	-1
110	-1
111	0

What happens if there is a string of '1's in the multiplier?

- There will be a -1 in the beginning, because the group begins with 110.
- Similarly, there will be a +1 at the end, because it will end with 011.
- However, for the length of continuous '1's, nothing needs to be done (add zeros).

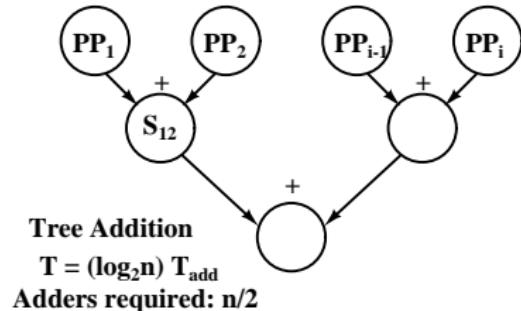
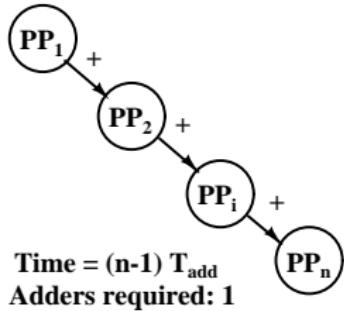
Thus Booth encoding reduces the number of partial products to half (multiplying 2 bits at a time).

It makes addition in columns of partial products fast because carry propagation during addition will be reduced.

Adding Partial Products

Multipliers can be speeded up by using special adder architectures which are optimized for adding more than two numbers.

- One option is to use tree adders rather than an accumulator.
- Several additions proceed in parallel, since all partial products are generated together.



Carry Save Adders

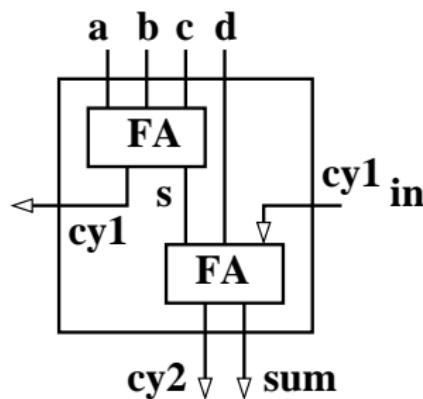
- Ordinary adders are large and complex. Also, these are slow due to rippling of carry.
- Let us consider an adder which presents its output not as one word - but two. The actual result is the sum of these.
- Obviously, an adder of this type is of no use for adding just two numbers! But it can be useful in a multiplier where we are adding multiple terms.
- For each bit column, the sum goes into one output word, while carry outs go into the other (without being added to the next more significant column).
- Now there is no rippling of carry and the output is available in constant time.
- We need a conventional adder in the end to add these two words.
- This type of adder is called a “Carry Save Adder” or CSA.

Carry Save Adders

- A Carry Save Adder (whose output is two words which must be added to produce the result) is of no use for adding just two words!
- However, we can construct a useful CSA for adding 4 bits in the same column.
- We make use of the fact that all partial product bits are available in constant time after the application of inputs.
- Since there are multiple bits to be added, we can feed three of them to a full adder.
- The sum and carry output of this adder is then available in constant time.

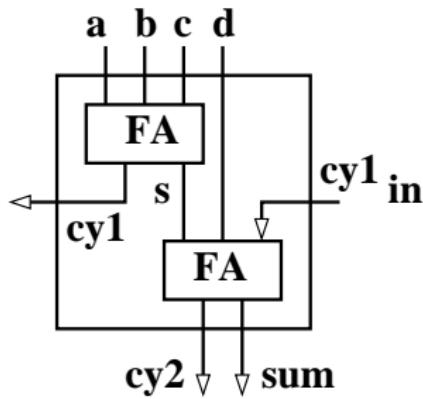
Carry Save Adders

The 4 input 2 output CSA uses two full adders as shown below:



- The first Full adder uses 3 bits of partial products of the same weight (bits in the same column).
- These are available in parallel in constant time.
- The sum output of first FA goes to the second FA.
- The carry output (**cy1**) of the first FS goes as intermediate input to the CSA used in the column to the left of this one.

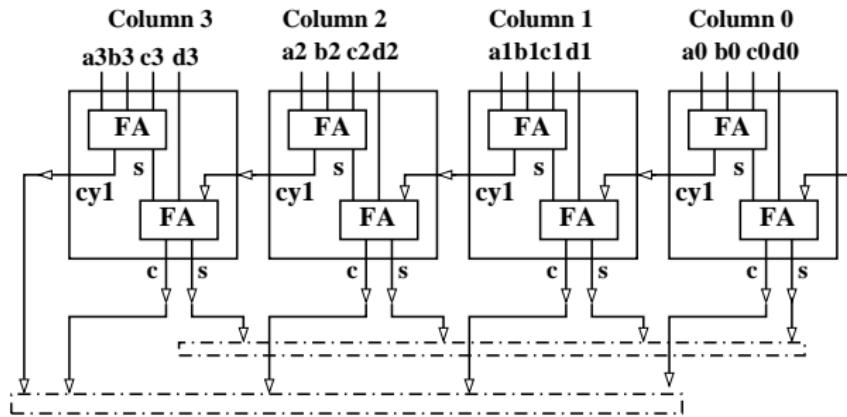
Carry Save Adders



- The second FA accepts one additional bit from the partial product column, the sum output of the first Full Adder FA1 and cy1 output coming from the CSA to its right.
- All inputs to this Full Adder are also available in constant time.
- Notice that even though cy1 goes from one column to the next significant column, **it does not ripple all the way horizontally**.
- It goes to FA2 of the more significant column whose output is not required by the next column.

Tiling Carry Save Adders

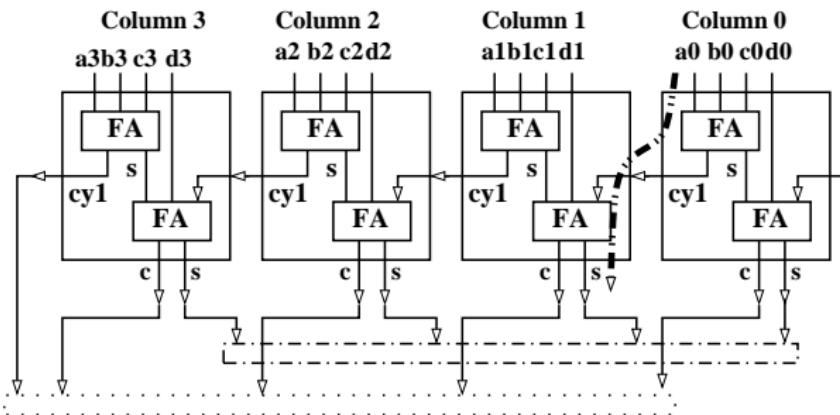
The figure below shows how we can add 4 columns of 4 bits each. Rows are labeled as a,b,c and d. Columns are 0,1,2 and 3.



- Outputs are collected in two separate registers (shown in dotted lines).
- These must be added using a conventional adder.

Critical Path of Carry Save Adders

Critical path of a 4x4 Carry Save Adder is shown below.



One can see that the critical path has been broken up.

Addition of 4 words of 32 bits each will also have a critical path of the same length.

Wallace Multipliers

- Multipliers do not have the same number of bits in every column.
- In 1964, Wallace proposed a method for a carry save like reduction scheme which is valid for columns of variable length.
- Wallace multiplier uses adders which take multiple inputs of the same weight and produce sum outputs of the same weight and carry outputs with higher weights.
- These are combined in stages to reduce the number of terms at each weight to 2 or less.
- These two terms are then added by a conventional adder to produce the final result.

Wallace Multipliers

Wallace multipliers act in three stages:

- ① Generate all bits of the partial products in parallel.
- ② Collect all partial products bits with the same place value in bunches of wires and reduce these in several layers of adders till each weight has no more than two wires.
- ③ For all bit positions which have two wires, take one wire at corresponding place values to form one number, and the other wire to form another number.

Add these two numbers using a fast adder of appropriate size.

Reduction Stage of Wallace Multipliers

- We assume that Full adders and Half adders will be used.
- A full adder takes 3 inputs and produces one output of the same weight (sum) and another of higher weight (carry). This is called a (3,2) adder. It reduces the number of wires at its own weight by 2 and adds one wire at the higher weight.
- A half adder takes 2 inputs and produces one output of the same weight (sum) and another of higher weight (carry). This is a (2,2) adder. It reduces the number of wires at its own weight by 1 and adds one wire at the higher weight.
- The reduction algorithm is general and can be used with any adders of type (n,m) . For example, a carry save adder is of type $(4,2)$.

Reduction Stage of Wallace Multipliers

- Each reduction stage looks at the number of wires for each weight and if any weight has more than 2 wires, it adds a layer of adders.
- When the numbers of wires for each weight have been reduced to 2 or less, we form one number with one of the wires at corresponding place values and another with the other wire (if present).
- These two numbers are added using a fast adder of appropriate size to generate the final product.

Reduction Stage of Wallace Multipliers

- Partial products are grouped in multiples of three rows.
- Rows which are additional over multiples of three are just passed on to the next stage.
- Wires are now reduced for each group of 3 rows.
- For any group of 3 rows, if we find 3 wires for any weight, we place a Full Adder, which generates 1 wire of the same weight and 1 wire with the next higher weight.
- If there are two wires left, we place a half adder to reduce these.
- If only one wire is left, it is carried through to the next layer.

Reduction Stage of Wallace Multipliers

- The reduction procedure is carried out for all weights. starting from the least significant weights.
- At the end of each layer, we count wires for each weight again, and if none has more than 2 wires, we proceed to the final addition stage.
- If any weight has 3 or more wires, we add another layer, and repeat this procedure till the number of wires for all weights is reduced to 2 or less.
- Now we compose one number from one of the left over wires at corresponding weights and another from the remaining wires.
- Finally, we use a conventional fast adder of appropriate size to add the two numbers.

Wallace Multiplier Example

Consider a multiplier for 4X4 bits. Partial products are generated in parallel and we have the following wires:

Bit	Terms	Wires
0	a_0b_0	1
1	a_0b_1, a_1b_0	2
2	a_0b_2, a_1b_1, a_2b_0	3
3	$a_0b_3, a_1b_2, a_2b_1, a_3b_0$	4
4	a_1b_3, a_2b_2, a_3b_1	3
5	a_2b_3, a_3b_2	2
6	a_3b_3	1

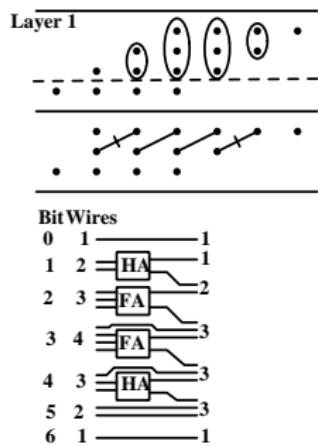
4X4 Wallace Multiplier: First Reduction

The multiplier has 4 rows of partial products, which are divided in groups of 3 and 1.

The bottom row is just passed on to the next stage.

For wires within the top 3 rows:

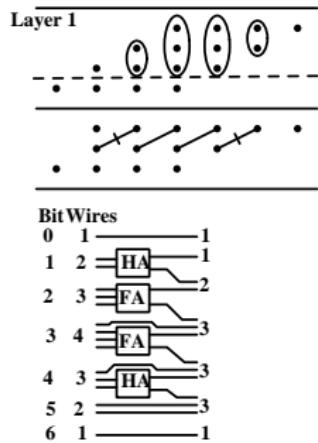
- Bit 0 has a single wire: passed through.
- Bit 1 has 2 wires: fed to a half adder.
- Bits 2 and 3 have 3 wires: fed to full adders.
- Bit 4 has 2 wires (in the group of 3): fed to a half adder.
- Bit 5 has 1 wire: passed through.



4X4 Wallace Multiplier: Outputs after First Reduction

After first reduction

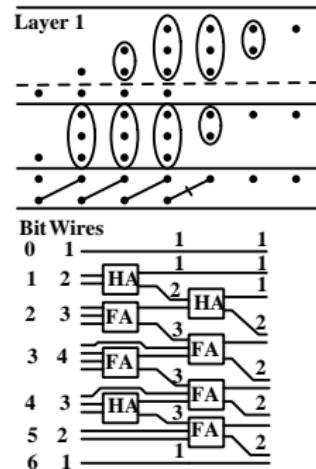
- Bits 0, 1 and 6 have a single wire.
- Bit 2 has 2 wires: carry of the half adder at bit 1 and the sum of full adder at bit 2.
- Bits 3 and 4 have 3 wires: carry of the full adder at lower weight, the sum wire from their full/half adder and a passed through wire.
- Bit 5 has 3 wires: carry of bit 4 plus 2 fed through wires.



4X4 Wallace Multiplier: Second Reduction

Since Bits 3, 4 and 5 have 3 wires each, we need another reduction layer. This will be the last reduction layer.

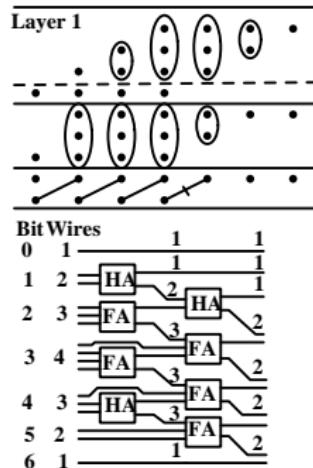
- Bits 0 and 1 have single wires. These are fed through.
- Bit 2 has 2 wires: these are fed to a half adder.
- Bits 3, 4 and 5 have 3 wires: These are fed to full adders.



4X4 Wallace Multiplier: Outputs from Second Reduction

Bits 0, 1, and 2 have single wires which carry the final result.

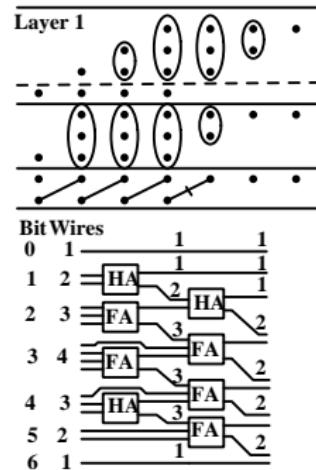
- Bit 3 has 2 wires: carry of the half adder at bit 2 and sum of the full adder at bit 3.
- Bit 4 has 2 wires: carry of the full adder at bit 3, and sum of the full adder at bit 4.
- Bit 5 has 2 wires, carry of bit 4 and sum of bit 5.
- Bit 6 has 2 wires, carry of bit 5 and 1 fed through wire.



4X4 Wallace Multiplier: Final Addition

After the second layer, no bit has more than 2 wires. Single wires at bits 0, 1 and 2 are fed through to the output.

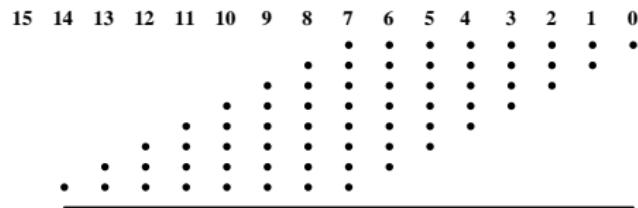
- A fast conventional adder is used to add the 2 bits each at Bits 3, 4, 5 and 6.
- Notice that we do not need a full width fast adder.
- This is because the half adders at low weights have already rippled the carry while the rest of weights were being reduced.
- This makes the final adder smaller and faster.



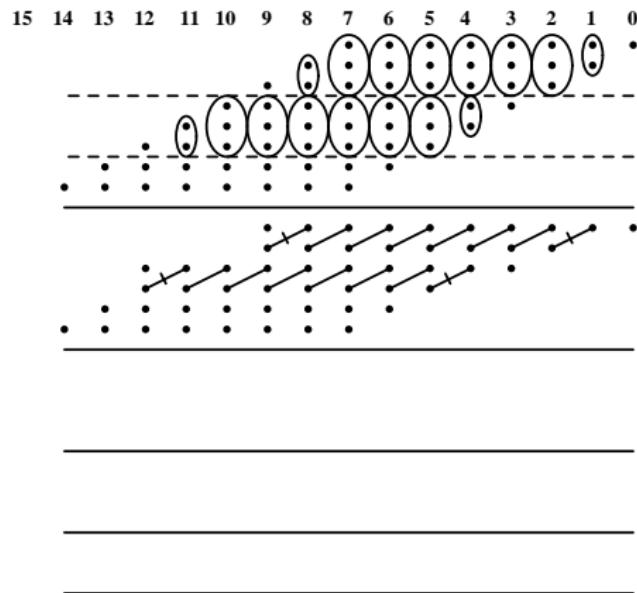
Redundant MSB in large Wallace Multipliers

- The reduction scheme as described above sometimes produces a redundant most significant bit.
- The result is still correct and the redundant bit will always be zero.
- To see this effect, let us apply the above scheme to an 8x8 multiplier.

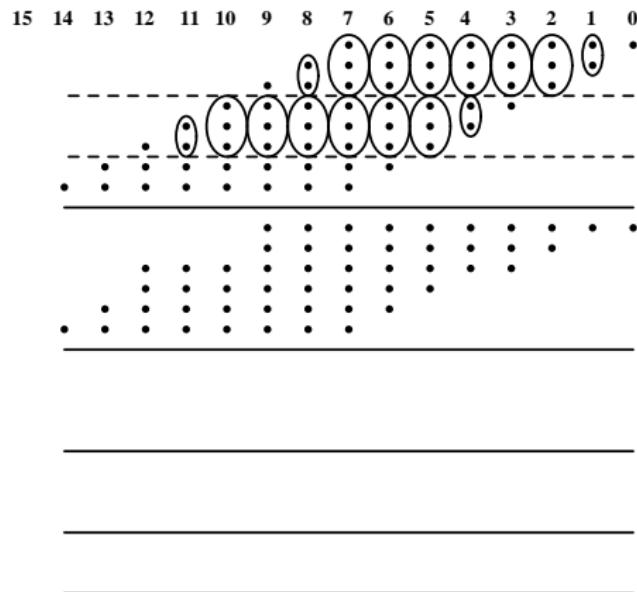
Redundant MSB in large Wallace Multipliers



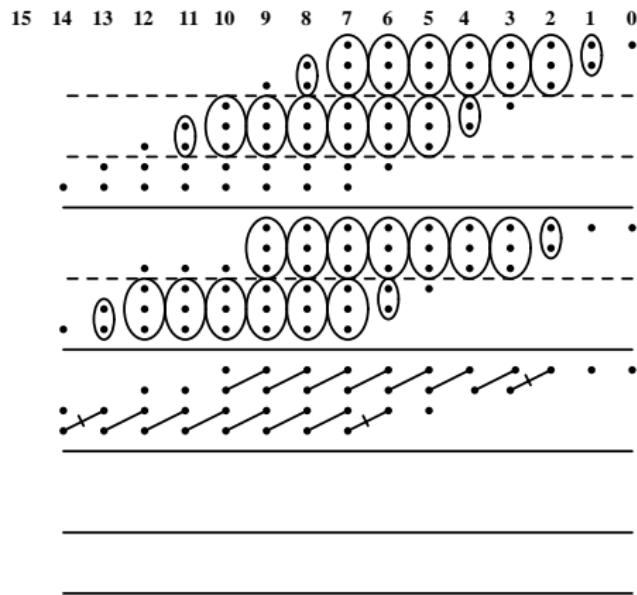
Redundant MSB in large Wallace Multipliers



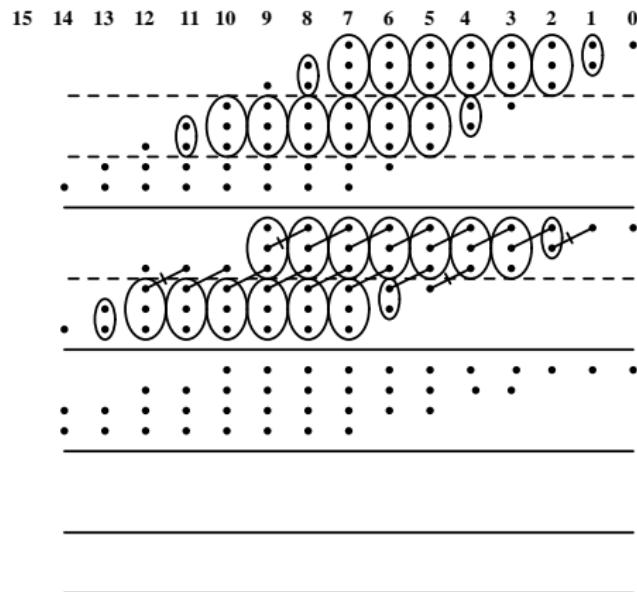
Redundant MSB in large Wallace Multipliers



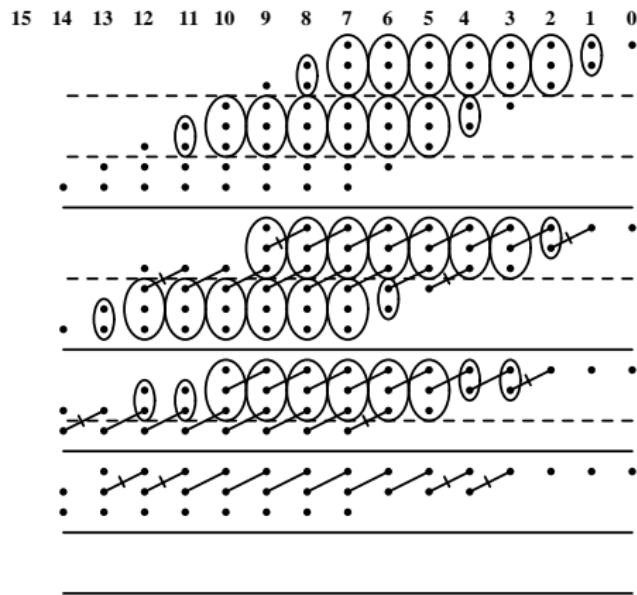
Redundant MSB in large Wallace Multipliers



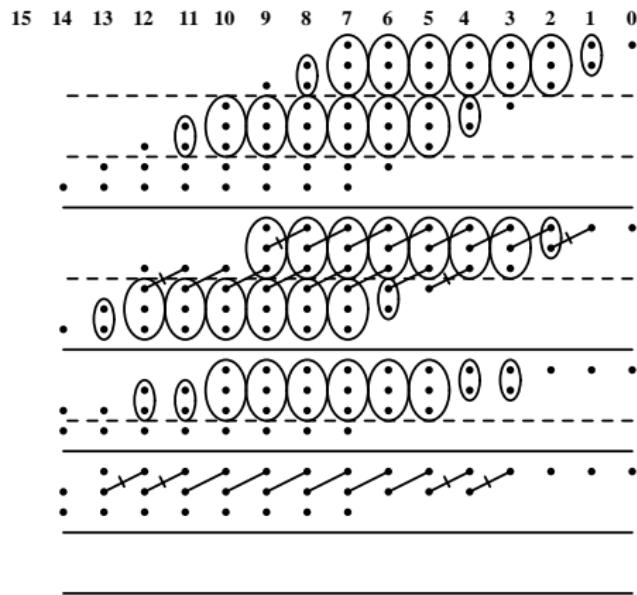
Redundant MSB in large Wallace Multipliers



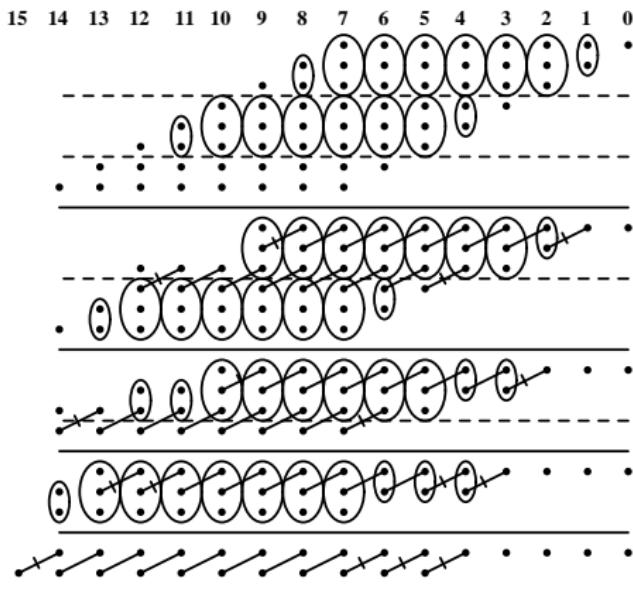
Redundant MSB in large Wallace Multipliers



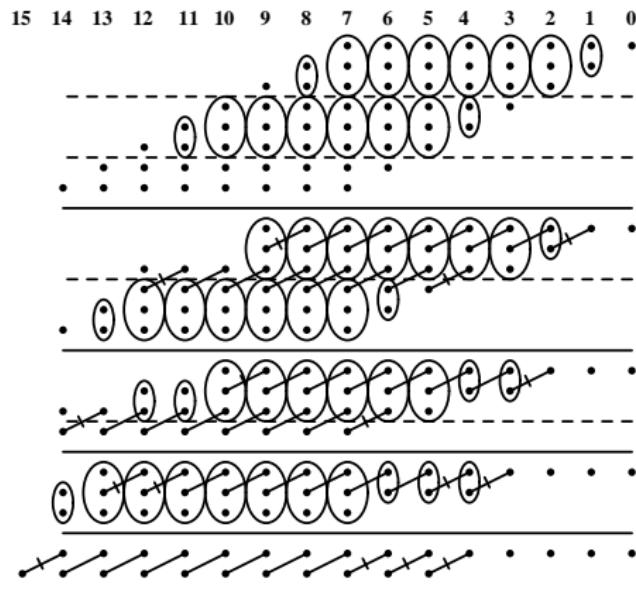
Redundant MSB in large Wallace Multipliers



Redundant MSB in large Wallace Multipliers



Redundant MSB in large Wallace Multipliers



As one can see, there are two bits at bit-14, which can produce a carry during the final addition.

When this carry is added to the bit at bit-15, it could produce another carry which will go to bit-16.

This would be an extra bit.
(Bits 0 to 16 will be 17 bits).

In practice, this does not occur, as multiplying 8 bit operands will generate at the most a 16 bit result.

Avoiding the Redundant MSB in Wallace Multipliers

One can avoid the redundant bit by modifying the reduction scheme.

- We treat all wires in a column as equivalent.
No groups of 3 rows).
- Make bunches of 3 wires and send each to a full adder.
- Now we can be left with 0, 1 or 2 wires.
- There is nothing to do for 0 wires left.
- If one wire is left, it is passed through to next layer.
- If two wires are left, we have a more complex decision.
- We need to define the capacity of a reduction layer to describe the policy for reduction of 2 wires.

Wire capacity of a reduction layer

- We define the capacity of a layer as the maximum number of wires it can accommodate. How can we determine it?
- We know that the final reduction layer should have no more than 2 wires. Now we can work **backwards** from the final layer to the first.
- Let d_j represent the maximum number of wires for any weight in layer j , where $j = 1$ for the final adder. (Thus $d_1 = 2$).
- The maximum number of wires which can be handled in layer $j+1$ (from the end) is the integral part of $(3/2)d_j$.

Wire capacity of a reduction layer

- The maximum number of wires for any weight in layer $j+1$ (from the end) is the integral part of $(3/2)d_j$.
- $j = 1$ for the final adder. Thus $d_1 = 2$.
- We go up in j , till we reach a number which is just greater than or equal to the largest bunch of wires in any weight.
- The number of reduction layers required is this $j_{final} - 1$.
- Capacities of layers starting from last layer and moving towards the top are 2, 3, 4, 6, 9, 13, 19

Reduction of two wires

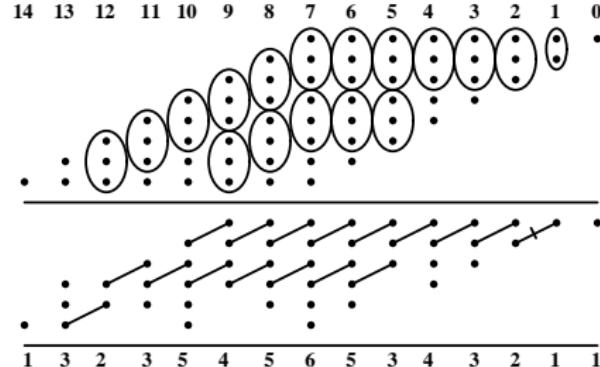
Now we can define the policy for reduction of 2 left over wires after deploying the maximum number of full adders.

- If all columns at the right have a single wire, we reduce the two wires using a half adder. (This helps in reducing the width of final adder).
- If there is a column to the right with more than one wire, we pass through the two wires to the next layer if it can accommodate these. (That is, the total number of wires do not exceed the capacity of that layer).
- If passing through the two wires would exceed the capacity of next layer, we reduce these with a half adder.

Wallace Reduction without redundant MSB

Max wires in this layer: 8, Capacity of next layer = 6.

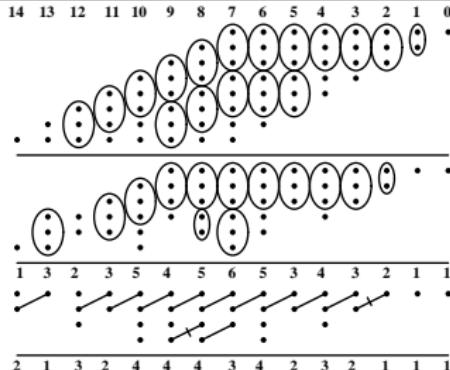
Bit No.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
In Wires	0	1	2	3	4	5	6	7	8	7	6	5	4	3	2	1
FA	0	0	0	1	1	1	2	2	2	2	2	1	1	1	0	0
Remaining	0	1	2	0	1	2	0	1	2	1	0	2	1	0	2	1
HA	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
PT	0	1	2	0	1	2	0	1	2	1	0	2	1	0	0	1
Sums	0	0	0	1	1	1	2	2	2	2	2	1	1	1	1	0
Carries to Higher bits	0	0	0	1	1	1	2	2	2	2	2	1	1	1	1	0
Output Wires	0	1	3	2	3	5	4	5	6	5	3	4	3	2	1	1



Wallace Reduction without redundant MSB

Second reduction layer: Capacity of next layer = 4.

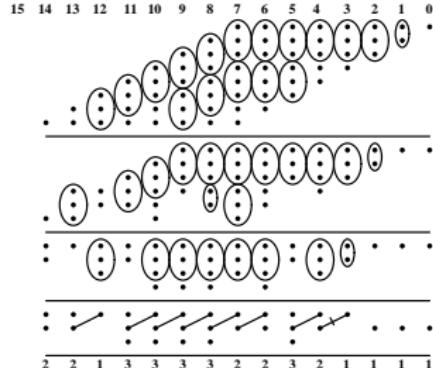
Bit No.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
In Wires	0	1	3	2	3	5	4	5	6	5	3	4	3	2	1	1
FA	0	0	1	0	1	1	1	1	2	1	1	1	1	0	0	0
Remaining	0	1	0	2	0	2	1	2	0	2	0	1	0	2	1	1
HA	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0
PT	0	1	0	2	0	2	1	0	0	2	0	1	0	0	1	1
Sums	0	0	1	0	1	1	1	2	2	1	1	1	1	1	0	0
Carries to Higher bits	0	0	1	0	1	1	1	2	2	1	1	1	1	1	0	0
Output Wires	0	2	1	3	2	4	4	4	3	4	2	3	2	1	1	1



Wallace Reduction without redundant MSB

Third reduction layer: Capacity of next layer = 3.

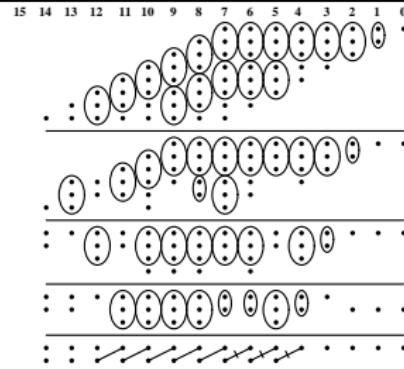
Bit No.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
In Wires	0	2	1	3	2	4	4	4	3	4	2	3	2	1	1	1
FA	0	0	0	1	0	1	1	1	1	1	0	1	0	0	0	0
Remaining	0	2	1	0	2	1	1	1	0	1	2	0	2	1	1	1
HA	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
PT	0	2	1	0	2	1	1	1	0	1	2	0	0	1	1	1
Sums	0	0	0	1	0	1	1	1	1	1	0	1	1	0	0	0
Carries to Higher bits	0	0	0	1	0	1	1	1	1	1	0	1	1	0	0	0
Output Wires	0	2	2	1	3	3	3	3	2	2	3	2	1	1	1	1



Wallace Reduction without redundant MSB

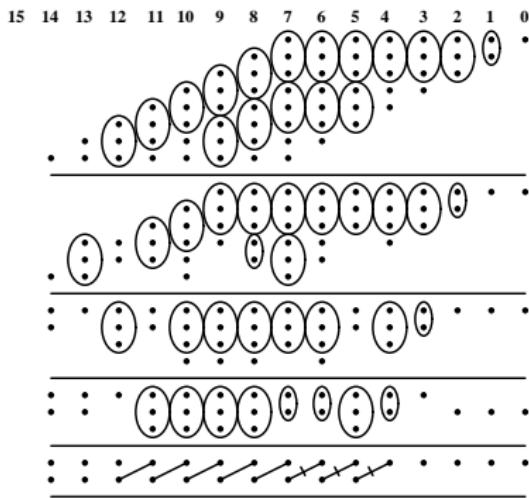
Final reduction layer: Capacity of next layer = 2.

Bit No.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
In Wires	0	2	2	1	3	3	3	3	2	2	3	2	1	1	1	1
FA	0	0	0	0	1	1	1	1	0	0	1	0	0	0	0	0
Remaining	0	2	2	1	0	0	0	0	2	2	0	2	1	1	1	1
HA	0	0	0	0	0	0	0	0	1	1	0	1	0	0	0	0
PT	0	2	2	1	0	0	0	0	0	0	0	0	1	1	1	1
Sums	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
Carries to Higher bits	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
Output Wires	0	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1



Thus we have reached two wires without generating a bit at b15. There are two wires at b14 and if these produce a carry, it will go to b15 and there is no redundant b15.

Wallace Reduction without redundant MSB



We have reduced the number of wires at all bit positions to ≤ 2 without generating a bit at b15.

There are two wires at b14 and if these produce a carry, it will go to b15 and there is no redundant b16.

Dadda Multipliers

Dadda multipliers are very similar to Wallace multipliers and use the same 3 stages:

- ① Generate all bits of the partial products in parallel.
- ② Collect all partial products bits with the same place value in bunches of wires and reduce these in several layers of adders till each weight has no more than two wires.
- ③ For all bit positions which have two wires, take one wire at corresponding place values to form one number, and the other wire to form another number.

Add these two numbers using a fast adder of appropriate size.

The difference is in the reduction stage.

Dadda Multipliers

- Wallace multipliers reduce as soon as possible, while Dadda multipliers reduce as late as possible.
- Dadda multipliers plan on reducing the final number of wires for any weight to 2 with as few and as small adders as possible.
- We determine the number of layers required first, beginning from the **last** layer, where no more than 2 wires should be left.
- The number of layers in Dadda multipliers is the same as in Wallace multipliers.

Dadda Multipliers: Number of layers

- We work back from the final adder to earlier layers till we find that we can manage all wires generated by the partial product generator.
- We know that the final adder can take no more than 2 wires for each weight.
- Let d_j represent the maximum number of wires for any weight in layer j , where $j = 1$ for the final adder. (Thus $d_1 = 2$).
The maximum number of wires which can be handled in layer $j+1$ (from the end) is the integral part of $\frac{3}{2}d_j$.
- We go up in j , till we reach a number which is just greater than or equal to the largest bunch of wires in any weight.
- The number of reduction layers required is this $j_{final} - 1$.

Wire Reduction in Dadda Multipliers

- At each layer we know the maximum number of wires which should be left for the next layer.
- For each weight, we place the **least number** of **smallest** adders, such that the wires going out to the next layer do not exceed the maximum number of wires it can handle.
- At each weight, we must consider all the sum and pass through wires at this weight, as well as the wires which will be transferred through carry of the less significant weights, to the next layer.
- That is why we must begin with the lowest weight and go towards higher weights in each layer.

Dadda Multiplier: Example

Take the example of 4-bit by 4-bit multiplication multiplying $a_3a_2a_1a_0$ by $b_3b_2b_1b_0$. As before, partial products are generated in parallel and we have the following wires:

Weight	Terms	Wires
1	a_0b_0	1
2	a_0b_1, a_1b_0	2
4	a_0b_2, a_1b_1, a_2b_0	3
8	$a_0b_3, a_1b_2, a_2b_1, a_3b_0$	4
16	a_1b_3, a_2b_2, a_3b_1	3
32	a_2b_3, a_3b_2	2
64	a_3b_3	1

4x4 Dadda Multiplier: Number of Reduction Layers

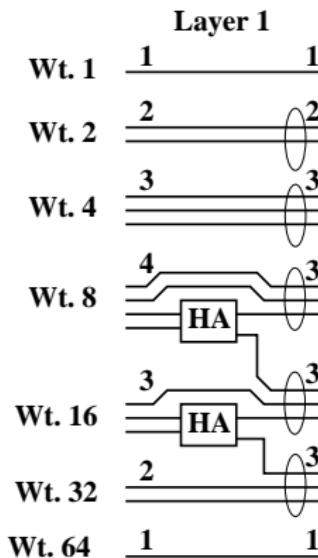
- Maximum no. of wires for any weight in this example is 4.
- $d_1 = 2, d_2 = 3, d_3 = 4$. So we need 2 layers of reduction.
- The first reduction layer should reduce the number of wires at any weight to a maximum of 3.
- The second layer will then reduce these to a maximum of 2 wires.
- At each reduction layer, we scan from less significant weights to more significant ones, keeping track of additional carry wires which will be transferred *at the output* from lower weights to higher ones.

4x4 Dadda Multiplier: First Reduction Layer

Weight	Wires
1	1
2	2
4	3
8	4
16	3
32	2
64	1

- Weights 1, 2 and 4 have 3 or less wires. These are passed through.
- Weight 8 has 4 wires. No carry is anticipated from lower weights. A half adder is used to reduce the output wires to 3. (Half Adder Sum + 2 wires passed through).
- Weight 16 has 3 wires, but we anticipate a carry from the adder at weight 8. So we should reduce by 1 to keep the total **output** wires to 3. So this column is also reduced using a half adder.

4X4 Dadda Multiplier: After First Reduction



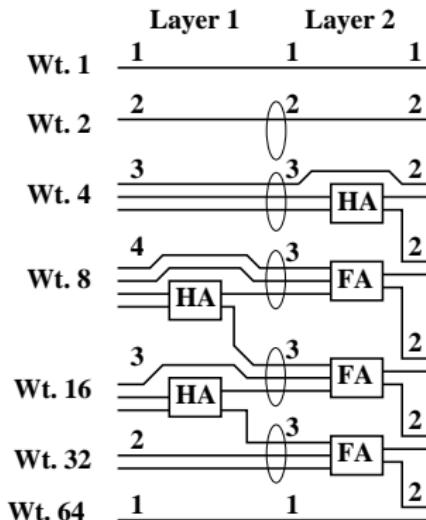
- Wt.1 has the single wire which was fed through.
- Wt.2 has 2 fed through wires.
- Wt.4 has 3 wires: all passed through.
- Wt.8 has 3 wires: sum of the half adder at wt.4, and 2 passed through.
- Wt.16 has 3 wires: carry of wt. 8, sum of half adder at 16 and 1 passed through.
- Wt.32 has 2 wires: carry of wt. 16 and 2 passed through.
- Wt.64 has 1 fed through wire.

4X4 Dadda Multiplier: Second Reduction

In the second layer, we should leave no more than 2 wires at any weight, as this is the last stage.

- As before, we anticipate the number of carry wires transferred from the lower weight when planning reduction using half or full adders.
- In Dadda multipliers, we use minimum hardware during reduction. So the smallest adder which will reduce the output wires to 2 will be used.
- At the lowest weights, if the number of wires is less than or equal to 2, we just pass these through.
- So the single wire at Wt. 1, and the 2 wires at Wt. 2 are just fed through.

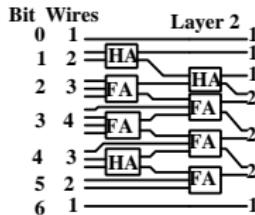
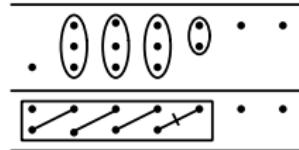
4X4 Dadda Multiplier: Second Reduction



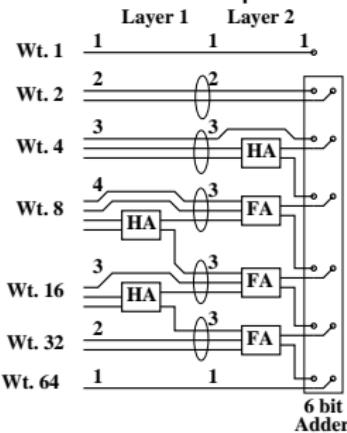
- 3 wires at Wt. 4 are reduced to 2 by a half adder: No carry in expected.
- Wt. 8 has 3 input wires. Carry will arrive from Wt. 4. Reduced using a Full adder.
- Wt. 16 has 3 input wires. Carry will arrive from Wt. 8. Reduced using a full adder.
- Wt. 32 has 3 input wires. Carry will arrive from Wt. 16. Reduced using a full adder.
- Wt. 64 has 1 input wire. Carry will arrive from Wt. 32, making it 2 output wires, which will be fed through.

4X4 Multipliers: Final Addition

Wallace Multiplier



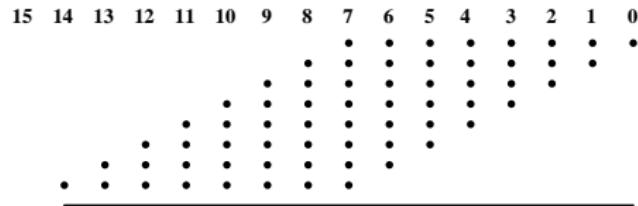
Dadda Multiplier



Notice that we have used only 3 Full Adders and 3 Half Adders during reduction, whereas Wallace multiplier requires 5 Full Adders and 3 Half Adders.

We require a 6-bit final adder for Dadda multiplier, whereas Wallace multiplier needs only a 4 bit final adder.

Dadda 8X8 Multiplier: Dot diagrams



capacity: 6

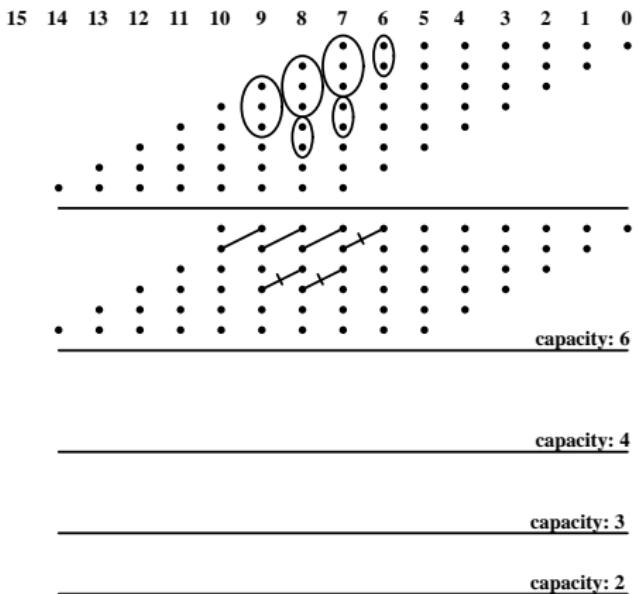
capacity: 4

capacity: 3

capacity: 2

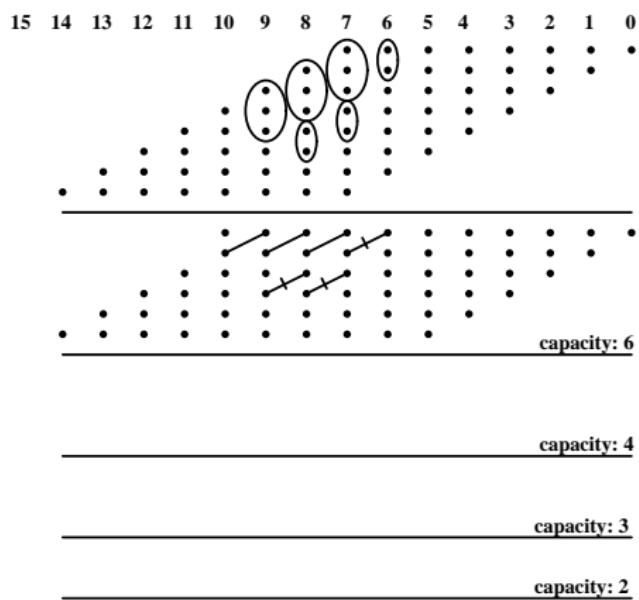
Dadda 8X8 Multiplier: First reduction

- Capacity of next layer is 6. Since bits 0-5 have six or less wires, these are just passed through.
- bit 6 has 7 wires. To reduce to six, we place a half adder. (This gives a sum wire at bit 6 and a carry wire at bit 7).
- These outputs are shown by a dot each at bits 6 and 7, joined by a crossed line.
- Remaining 5 bits are passed through.



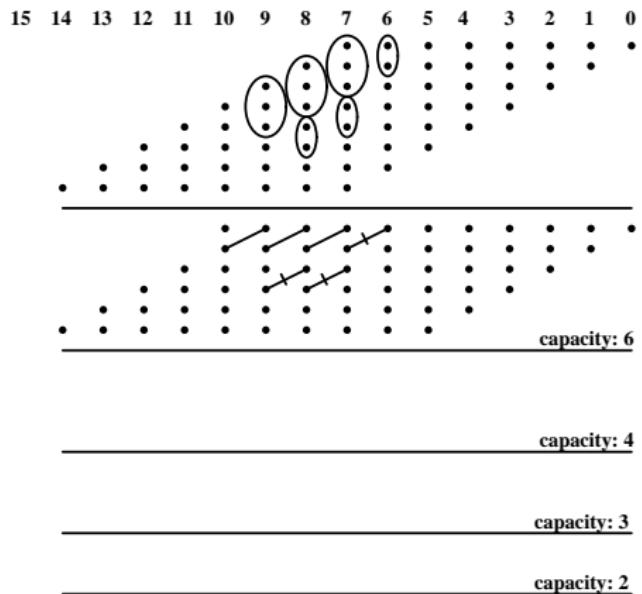
Dadda 8X8 Multiplier: First reduction

- Bit 7 has 8 wires. Of the six output places, one is already occupied by the carry of half adder at bit 6. So we should produce only 5 outputs at this bit – a reduction by 3.
- This can be done through a full and a half adder. Outputs of the full adder are shown as a dot at bit 7 (sum) and another at bit 8 (carry) joined by a line.
- Outputs of the half adder are also shown by two dots joined by a crossed line as before.



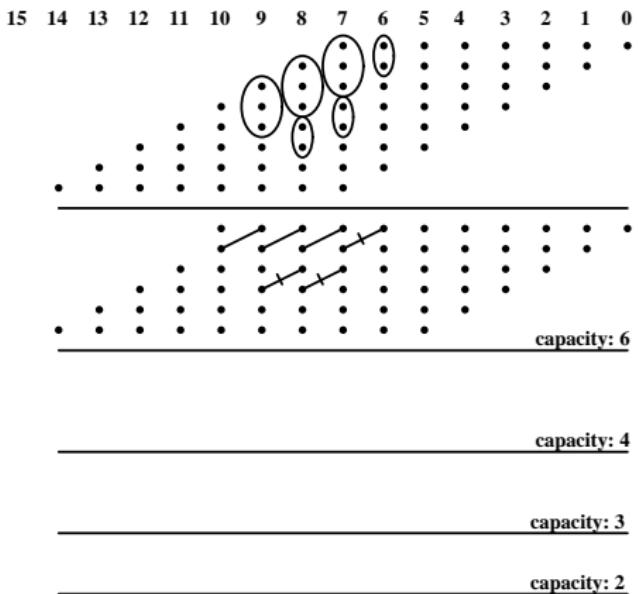
Dadda 8X8 Multiplier: First reduction

- Bit 8 has 7 wires. Of the 6 output places, 2 are occupied by carries of full and half adder.
- So we should produce only 4 outputs at this bit – again a reduction by 3.
- This can be done through a full and a half adder as before.
- Full and half adder take up 5 wires. The remaining 2 are passed through.



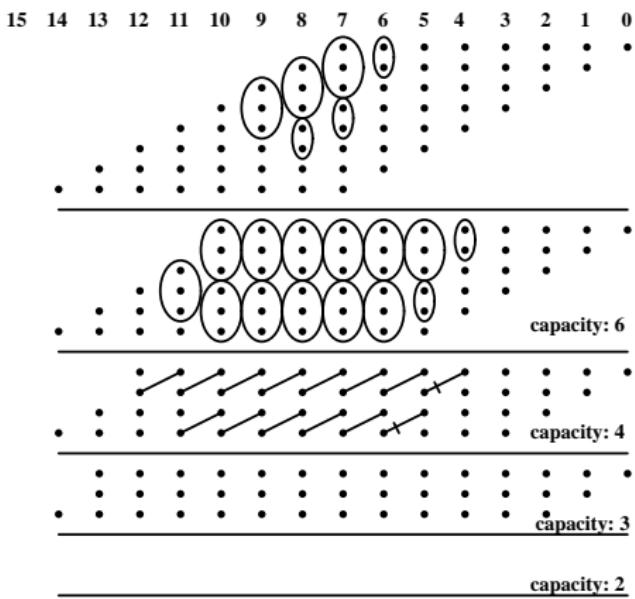
Dadda 8X8 Multiplier: First reduction

- Bit 9 has 6 wires, which should be reduced to 4 (since two places are taken up by carries of full and half adders).
- This can be done by a full adder whose outputs are shown by dots at bit 9 and 10 joined by a line.
- The remaining 3 wires are passed through.
- Wires of all the higher bits can be passed through without exceeding the limit of 6 outputs.



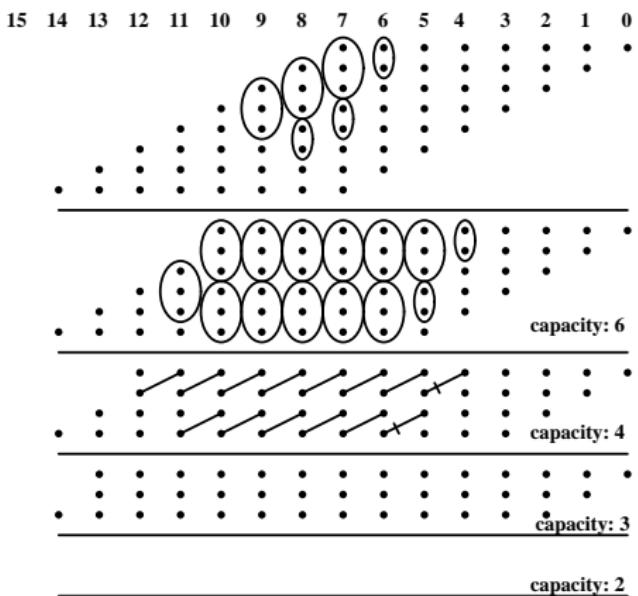
Dadda 8X8 Multiplier: Second reduction

- The output capacity of next layer is 4.
- Wires of bits 0-3 can just be passed through.
- For all bit position, we reduce the output places available by the incoming carries of previous bit.
- We place minimal number of full and half adders to reduce the total output wires to 4.
- Each full adder (FA) reduces wires by 2, half adder (HA) reduces by 1.



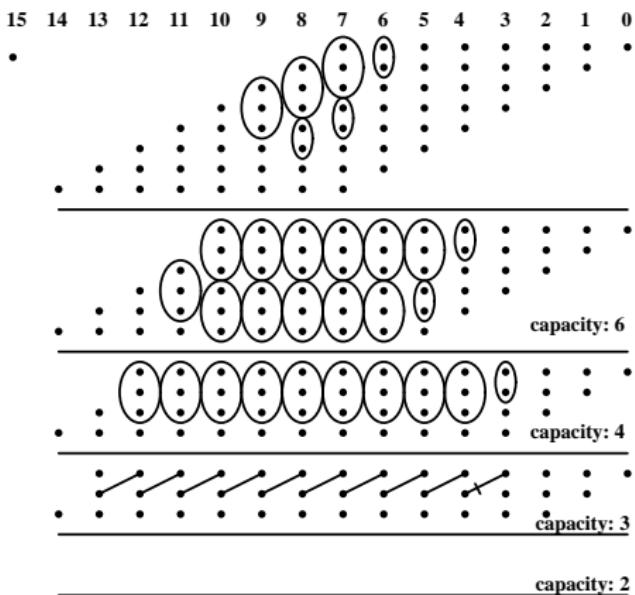
Dadda 8X8 Multiplier: Second reduction

- Bit 4 has 5 wires. Reduced to 4 by HA.
- Bit 5 has 6 wires, reduced to (4-1) by FA+HA.
- Bit 6 has 6 wires, reduced to (4-2) by 2 FAs.
- This is repeated till bit 10.
- Bit 11 has 4 wires, reduced to (4-2) by a FA.
- All other wires can be passed through.



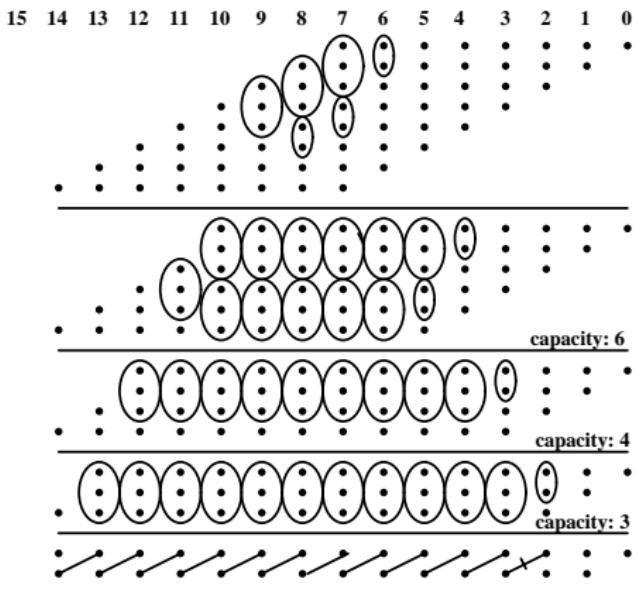
Dadda 8X8 Multiplier: Third reduction

- The reduction procedure can be repeated at each layer.
- If 2 wires (or multiples of 2) are to be reduced, we place FAs till 1 or 0 wires are left.
- If 1 wire remains, we place a Half adder.
- This layer requires a half adder at bit 4, FA + HA at bit 5, 2 FAs at bits 6-10 and a full adder at bit 11.
- Rest of the wires are just passed through.



Dadda 8X8 Multiplier: Final reduction

- Capacity of the final layer is 2.
- We continue with the same procedure, placing a half adder at bit 3 and full adders at bits 4-12.
- The remaining wires are passed through. Now we can make two words of 14 bit width and add these using a fast adder to get the final product.
- Notice there is no extra bit!



Comparison of Wallace and Dadda Multipliers

- Wallace and Dadda multipliers need the same number of layers.
- Dadda multiplier minimizes the number of adders, so it has the potential for lower complexity and power.
- Dadda multiplier uses more pass throughs and smaller adders which have lower delay. Thus it can also minimize the critical delay for reaching the final 2 wire stage.
- However, The final addition in Dadda multiplier needs wider carry propagating adders, which can slow it down.
- A careful evaluation inclusive of wiring and parasitic delays has to be made to determine which is the faster adder for a given configuration and process.

Multiply and Accumulate circuits

- A common task during data processing is the evaluation of quantities like $\sum c_i X_i$.
- This can be made easier if we have a dedicated hardware circuit which can compute $A \times B + C$. Here the size of the operand C is the same as that of the *product* $A \times B$.
- This circuit is the multiply and accumulate or MAC circuit.
- The MAC circuit is not much more complex compared to a multiplier. This is because during multiplication we are anyway adding multiple bits in every column. The accumulator just provides an additional wire at each bit position.
- This circuit is much faster than separate multiplication and addition because the latter requires two steps of addition with rippling carry while the MAC requires only one.

8x8 MAC with 16 bit Accumulator

Consider for example a MAC circuit which multiplies two 8 bit operands and adds the product to a 16 bit accumulator.

The number of wires from partial sums of the multiplier is:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Wires	0	1	2	3	4	5	6	7	8	7	6	5	4	3	2	1

If we include a wire at each position from the accumulator, we get the wire count as:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Wire	1	2	3	4	5	6	7	8	9	8	7	6	5	4	3	2

8x8 MAC, Wallace reduction

We can now proceed to reduce these wires as a Wallace tree.

Stage 1: Max. wires:9, capacity of next stage = 6

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
In	1	2	3	4	5	6	7	8	9	8	7	6	5	4	3	2
FA	0	0	1	1	1	2	2	2	3	2	2	2	1	1	1	0
HA	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1
PT	1	2	0	1	2	0	1	0	0	2	1	0	2	1	0	0
Out	1	3	2	3	5	4	6	6	5	6	5	3	4	3	2	1

Wallace reduction: Stage 2

Stage 2: capacity of next stage = 4

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
In	1	3	2	3	5	4	6	6	5	6	5	3	4	3	2	1
FA	0	1	0	1	1	1	2	2	1	2	1	1	1	1	0	0
HA	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0
PT	1	0	2	0	2	1	0	0	0	0	2	0	1	0	0	1
Out	2	1	3	2	4	4	4	4	4	3	4	2	3	2	1	1

Wallace reduction: Stage 3

Stage 3: capacity of next stage = 3

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
In	2	1	3	2	4	4	4	4	4	3	4	2	3	2	1	1
FA	0	0	1	0	1	1	1	1	1	1	1	0	1	0	0	0
HA	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
PT	1	1	0	2	1	1	1	1	1	0	1	2	0	0	1	1
Out	1	2	1	3	3	3	3	3	3	2	2	3	2	1	1	1

Wallace reduction: Stage 4

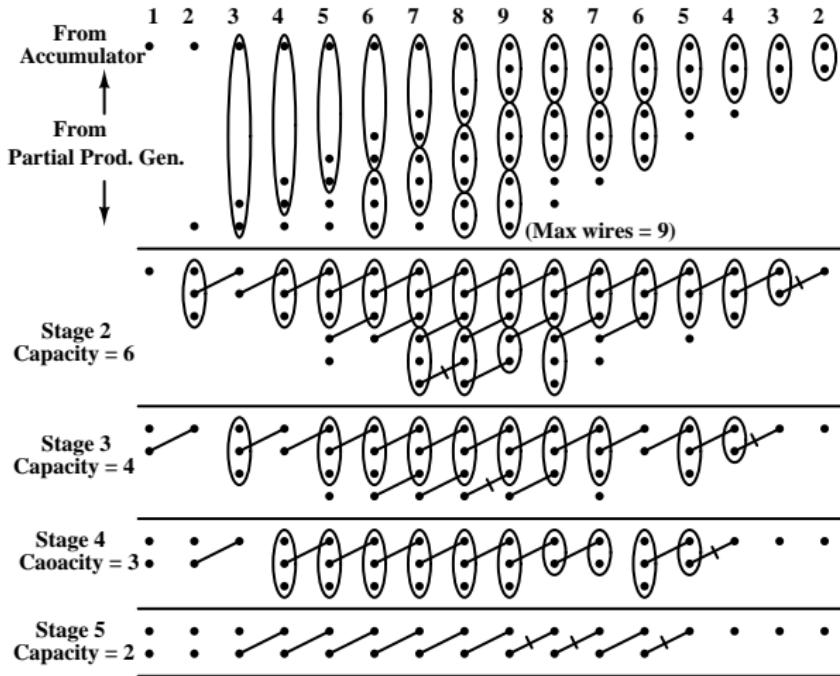
Stage 4: capacity of next stage = 2

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
In	1	2	1	3	3	3	3	3	3	2	2	3	2	1	1	1
FA	0	0	0	1	1	1	1	1	1	0	0	1	0	0	0	0
HA	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	0
PT	1	2	1	0	0	0	0	0	0	0	0	0	0	1	1	1
Out	2	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1

Now the maximum wires at any position is 2. We compose two words using one wire each and add these using a fast conventional adder to get the final result.

Wallace Tree for MAC

The dot diagram for this scheme is shown below:



8x8 MAC

- Complexity of the MAC circuit is not much higher than a Wallace tree 8x8 multiplier.
- This is so for any wire reduction scheme. We could have used the Dadda scheme and the complexity would not be much more than the plain 8x8 Dadda multiplier.
- The result is produced much faster than separate multiplication and addition.
- This is because a traditional adder is used only once in the MAC. Otherwise, the multiplier will use it once and then the addition to the accumulator will again involve a traditional addition.

Serial Multipliers

Often, we need multipliers which have very low complexity or very low power consumption and speed is not very important.

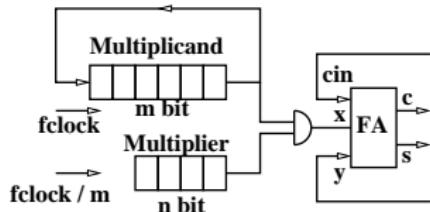
Serial multipliers are a good option in such cases.

Low complexity multipliers can be bit serial or row serial.

Bit serial multipliers require $m \times n$ clocks for completing an $m \times n$ multiplication.

Row serial multipliers require only n steps, but we require m full adders rather than just one.

Bit Serial Multipliers: Partial Product Generation

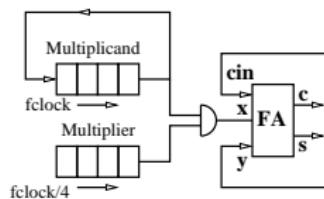


- Each bit of the multiplier needs to be ANDed with each bit of the multiplicand.
- This requires that all multiplicand bits be presented one after the other, every time a new bit from the multiplier is taken up.
- This can be managed by using a re-circulating shift register for the multiplicand, which is clocked at a rate which is m times faster than the clock to the multiplier shift register.
- The inputs y and Cin to the full adder have to be appropriately selected and timed to generate the correct product.

Bit Serial Multipliers: Partial Product Generation

Consider a 4×4 bit serial multiplier.

The x input to the Full Adder appears in the following order:



ck	x	ck	x	ck	x	ck	x
0	a0b0	4	a0b1	8	a0b2	12	a0b3
1	a1b0	5	a1b1	9	a1b2	13	a1b3
2	a2b0	6	a2b1	10	a2b2	14	a2b3
3	a3b0	7	a3b1	11	a3b2	15	a3b3

Bit Serial Multipliers: Partial Product addition

The arrival time of partial product bits is:

ck	x	ck	x	ck	x	ck	x
0	a0b0	4	a0b1	8	a0b2	12	a0b3
1	a1b0	5	a1b1	9	a1b2	13	a1b3
2	a2b0	6	a2b1	10	a2b2	14	a2b3
3	a3b0	7	a3b1	11	a3b2	15	a3b3

We need additions as follows:

$$\begin{array}{r}
 & a3 & a2 & a1 & a0 \\
 \times & b3 & b2 & b1 & b0 \\
 \hline
 & a3b0 & a2b0 & a1b0 & a0b0 \\
 & a3b1 & a2b1 & a1b1 & a0b1 \\
 a3b2 & a2b2 & a1b2 & a0b2 \\
 a3b3 & a2b3 & a1b3 & a0b3
 \end{array}$$

Bit Serial Multipliers: Partial Product addition

Let us put the arrival time of terms in parentheses next to each term.

×	a3 b3	a2 b2	a1 b1	a0 b0
	a3b0(3) a2b1(6) a1b2(9) a0b3(12)	a2b0(2) a1b1(5) a0b2(8)	a1b0(1) a0b1(4)	a0b0(0)
a3b3(15) a3b2(11) a2b3(14)	a3b1(7) a2b2(10) a1b3(13)			

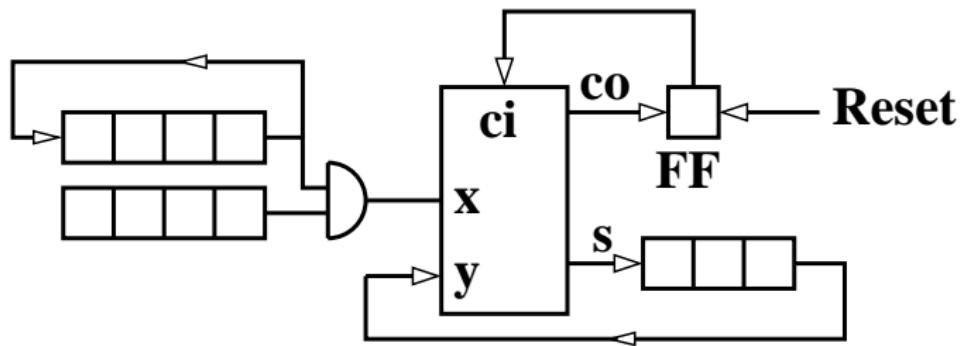
It is clear that for all additions, the earlier terms have to wait for 3 clock cycles before the later terms arrive.

We can manage this by putting a 3 bit shift register at the sum output and presenting the delayed output at the 'y' input of the full adder.

The carry output can be added immediately in the next clock, since it should go to the next column to its left.

Bit Serial Multipliers: Partial Product addition

A 3 clock delay for sum and a 1 clock delay for carry leads to the following circuit.

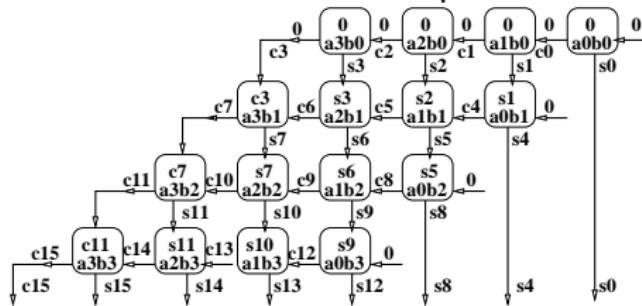


Unfortunately, it does not work!

We have to take care of a few exceptions at row ends.

Bit Serial Multiplier: Exceptions

Let us look at all the exceptions in detail.



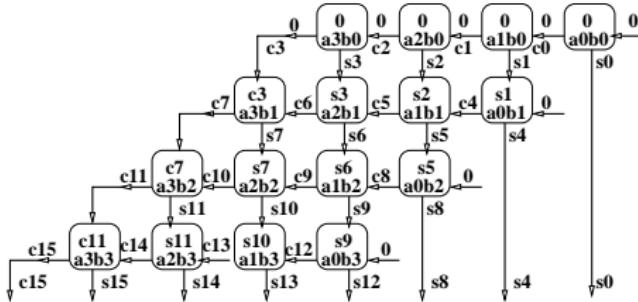
In the adjoining figure, sum and carry terms are indexed by the clock interval in which these were generated.

- At clocks 0, 4, 8 and 12, carry input should be forced to 0.
- At clocks 7, 11 and 15, the adder y input should receive carry terms (c3, c7 and c11) instead of sum terms (s4, s8 and s12).
- At these clocks, the sum terms should be taken out as result bits.

Bit Serial Multiplier: Exceptions

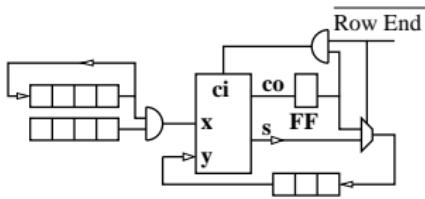
At clocks 0, 4, 8 and 12:

- Carry input should be forced to 0.
- The carry FF output (which is a 1 clock delayed version of cout) should be inserted in the 3-bit shift register.
Thus C3 (which is always 0), C7 and C11 will emerge at clocks 7, 11 and 15 respectively.
- The sum terms should be taken out as result bits.



Bit Serial Multiplier: Implementation

With exception handling at the end of rows, the serial multiplier will work.



- Carry input is forced to 0 at row ends.
- The mux normally inserts the sum into the shift register. However, at row ends, it inserts the delayed carry output.

- The sum terms at row ends can be taken out as the low bits of the product.
- One can add another shift register at the output to collect these.
- The 2 more significant bits of the shift register and the last sum and carry provide the high bits of the product at the end.

Row Serial Multipliers

- We need not reduce the complexity all the way down to a single adder for serial multipliers.
- We could have a row of n adders performing additions in parallel.
- Taking the example of 4×4 multiplication, we are trying to perform the following operations:

$$\begin{array}{r} \text{a3 a2 a1 a0} \\ \times \text{ b3 b2 b1 b0} \\ \hline \text{a3b0 a2b0 a1b0 a0b0} \\ \text{a3b1 a2b1 a1b1 a0b1} \\ \text{a3b2 a2b2 a1b2 a0b2} \\ \text{a3b3 a2b3 a1b3 a0b3} \\ \hline \end{array}$$

- We would like to perform 4 additions per clock interval.

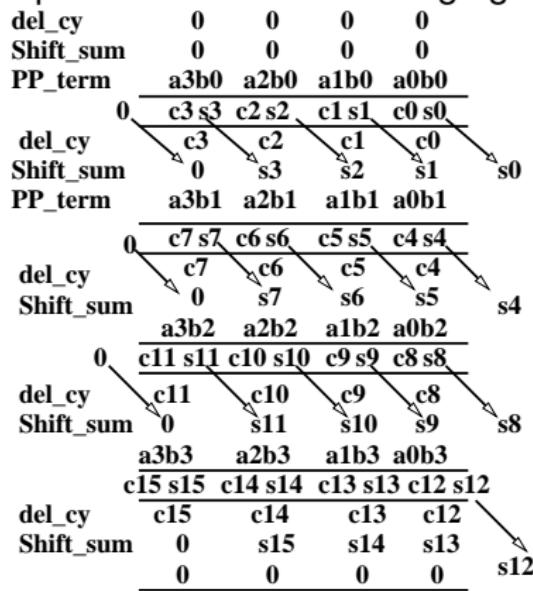
Row Serial Multipliers

- We can use n full adders arranged in n columns.
- The carry of previous addition should remain at the same column.
- The sum from the previous addition *in the left column* is brought to this column by a shift operation.
- This sum has the same weight as the carry generated during the previous clock in this column.
- These two are added to the partial product bit for this column.

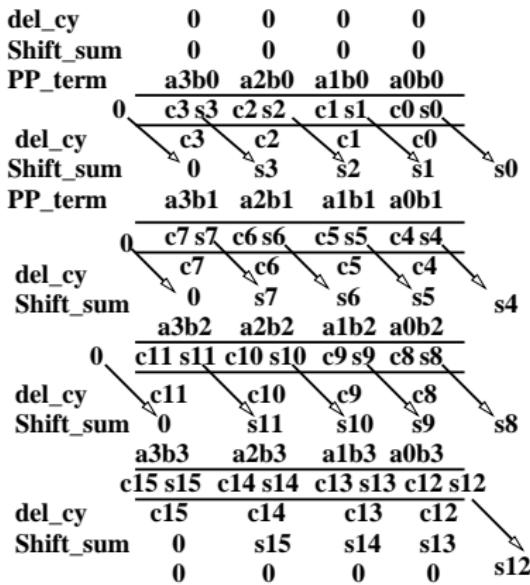
Row Serial Multipliers

The addition process using 4 adders is represented in the following figure.

$$\begin{array}{r}
 \begin{array}{cccc} a_3 & a_2 & a_1 & a_0 \end{array} \\
 \times \begin{array}{cccc} b_3 & b_2 & b_1 & b_0 \end{array} \\
 \hline
 \begin{array}{cccc} a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\ a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 \\ a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 \\ a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 \end{array}
 \end{array}$$



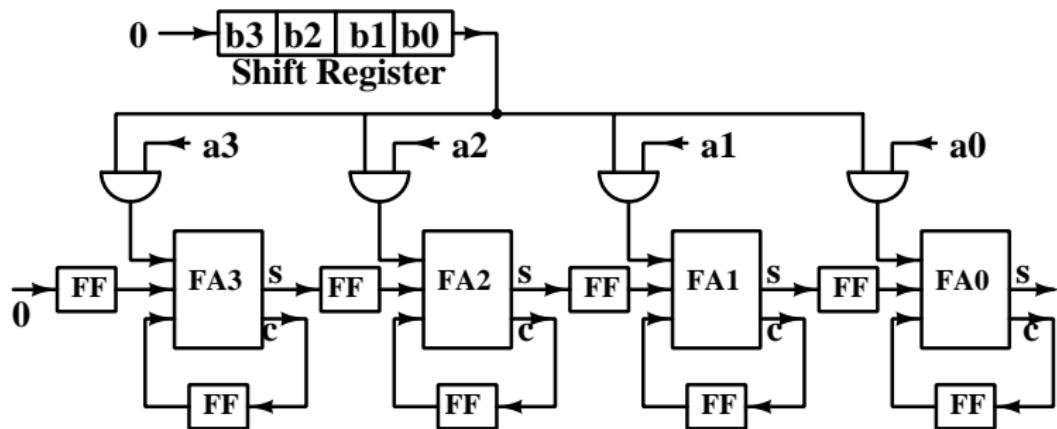
Row Serial Multipliers



- Notice that the same 'a' term is used in a given adder.
- The 'b' term has to be shifted right every time to generate the right partial product bit.
- Sums have to be shifted right to be added to the carry of the previous addition in the same column.
- 4 additional clock cycles will be required to ripple the carry in the last addition. During these, the partial product bits will be 0.

Row Serial Multipliers

This scheme can be implemented as follows:



Arithmetic Circuits

Dinesh Sharma
EE Department, IIT Bombay

November 11, 2020

Contents

1	Adders	4
1.1	Half Adder	4
1.2	Full Adder	4
1.3	Ripple Carry adder	5
1.4	Carry Look Ahead	8
1.4.1	Manchester Carry Chain	9
1.5	Carry Bypass Adder	10
1.6	Carry Select Adder	11
1.6.1	Stacking Carry Select Adders	12
1.7	Tree Adders	14
1.7.1	Brent Kung adder	16
1.7.2	Other logarithmic adders	17
1.8	Serial Adders	17
2	Shift and Rotate Operations	19
2.1	Shift and Rotate Operations	19
2.2	Barrel Shifters	20
2.2.1	Shift/Rotate as Select Operations	20
2.3	Barrel Shifters	20
2.3.1	Logarithmic Barrel Shifters	21
2.3.2	Right Rotate for an 8 bit Operand	21
2.3.3	8 bit Logical Shift Right	22
2.3.4	Combining Rotate and Shift Operations	23
2.3.5	Rotate and Shift by Masking	23
2.3.6	Bidirectional Shift and Rotate Operations	23
3	Multipliers	25
3.1	Shift and Add Multipliers	25
3.2	Array Multipliers	26
3.2.1	Critical Path through an Array Multiplier	27
3.3	Speeding up Multipliers	27

3.4	Booth Encoding	27
3.5	Modified Booth Encoding	28
3.6	Efficient Addition of Partial Products	29
3.6.1	Carry Save Adders	29
3.6.2	Critical Path of Carry Save Adders	31
3.7	Wallace Multipliers	31
3.7.1	Reduction Stage of Wallace Multipliers	32
3.7.2	Wallace Multiplier Example	34
3.7.3	Redundant MSB in large Wallace Multipliers	36
3.7.4	Avoiding the Redundant MSB in Wallace Multipliers	39
3.7.5	Wallace 8x8 Reduction without redundant MSB	39
3.7.6	Dadda Multipliers	44
3.8	Multiply and Accumulate circuits	48
3.9	Serial Multipliers	51
3.9.1	Bit Serial Multipliers	51
3.9.2	Bit Serial Multiplier: Implementation	53
3.9.3	Row Serial multipliers	54

List of Figures

1.1	Karnaugh map for full adder	5
1.2	A ripple carry adder	5
1.3	CMOS implementation for sum and carry	6
1.4	Mirror gate implementation of sum and carry	7
1.5	Manchester carry chain stage	10
1.6	Carry By-pass adder	10
1.7	Carry select adder	11
1.8	Linear stacking of carry select adders	12
1.9	Generation of P and G values in Brent Kung adder	16
2.1	Right rotate for an 8 bit operand	22
2.2	Logical Right Shift for an 8 bit operand	22
2.3	Combined Right Rotate/Shift for an 8 bit operand	23
2.4	Bit reversal for bidirectional Shift and Rotate Operations	24
3.1	Shift and add multiplier	25
3.2	Array multiplier	26
3.3	Critical path in an array multiplier	27
3.4	Tree additions in multipliers	30
3.5	Carry save adder	30
3.6	Tiling Carry Save Adders	31
3.7	Critical path of a carry save adder	32
3.8	First reduction stage of a 4x4 Wallace multiplier	34
3.9	Second reduction stage of a 4x4 Wallace multiplier	35
3.10	Partial product generation for bit-serial multiplication	51
3.11	4x4 bit serial multiplier	53
3.12	4x4 row serial multiplication	55
3.13	4x4 row serial multiplier	55

Chapter 1

Adders

1.1 Half Adder

The truth table for addition of two bits is:

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

From this, we can see that

$$\text{sum} = A \cdot \bar{B} + B \cdot \bar{A} \quad (1.1)$$

$$\text{carry} = A \cdot B \quad (1.2)$$

What do we do with the carry? Obviously, it must be added to more significant bits. Therefore, for subsequent stages, we need an adder with *three* inputs: A, B and Carry in. This kind of adder is called a full adder.

1.2 Full Adder

Truth Table for the addition of three bits is:

A	B	Cin	Sum	Cout
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

This leads to the Karnaugh map shown in fig. 1.1:

AB		00	01	11	10	
Cin	0	0	1	0	1	SUM
0	0	0	1	0	1	
1	1	0	1	1	0	

AB		00	01	11	10	
Cin	0	0	0	1	0	CARRY
0	0	0	1	1	0	
1	0	1	1	1	1	

Figure 1.1: Karnaugh map for full adder

$$\text{So } \text{sum} = \overline{A} \cdot \overline{B} \cdot C_{in} + \overline{A} \cdot B \cdot \overline{C_{in}} + A \cdot B \cdot C_{in} + A \cdot \overline{B} \cdot \overline{C_{in}}$$

$$C_{out} = A \cdot B + B \cdot C_{in} + C_{in} \cdot A = A \cdot B + C_{in} \cdot (A + B)$$

1.3 Ripple Carry adder

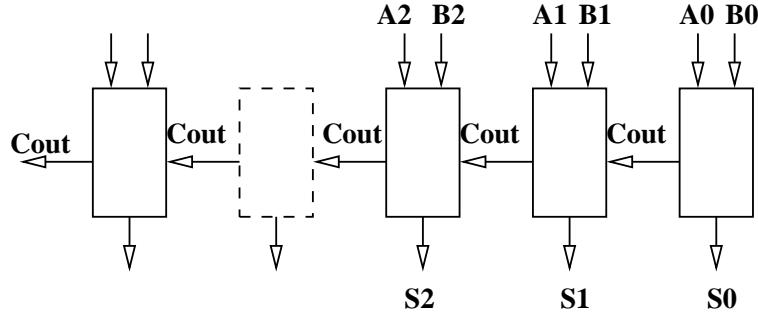


Figure 1.2: A ripple carry adder

- Carry out of one bit becomes Carry in of the next.
- This architecture is therefore called ripple carry adder.
- The critical delay path of the adder is the carry rippling from one bit to the next.

Because carry is on the critical path, Carry-out must be generated as quickly as possible. We need not optimize the delay of generating sum. We can in fact generate sum from

Carry out.

$$\overline{C_{out}} = \overline{A \cdot B + C_{in} \cdot (A + B)} \quad (1.3)$$

$$= (\overline{A} + \overline{B}) \cdot (\overline{C_{in}} + \overline{A} \cdot \overline{B}) \quad (1.4)$$

$$= \overline{A} \cdot \overline{C_{in}} + \overline{B} \cdot \overline{C_{in}} + \overline{A} \cdot \overline{B} \quad (1.5)$$

$$\overline{C_{out}} \cdot (A + B + C) = A \cdot \overline{B} \cdot \overline{C_{in}} + \overline{A} \cdot B \cdot \overline{C_{in}} + \overline{A} \cdot \overline{B} \cdot C_{in} \quad (1.6)$$

$$\text{sum} = A \cdot \overline{B} \cdot \overline{C_{in}} + \overline{A} \cdot B \cdot \overline{C_{in}} + \overline{A} \cdot \overline{B} \cdot C_{in} + A \cdot B \cdot C_{in} \quad (1.7)$$

$$= \overline{C_{out}} \cdot (A + B + C_{in}) + A \cdot B \cdot C_{in} \quad (1.8)$$

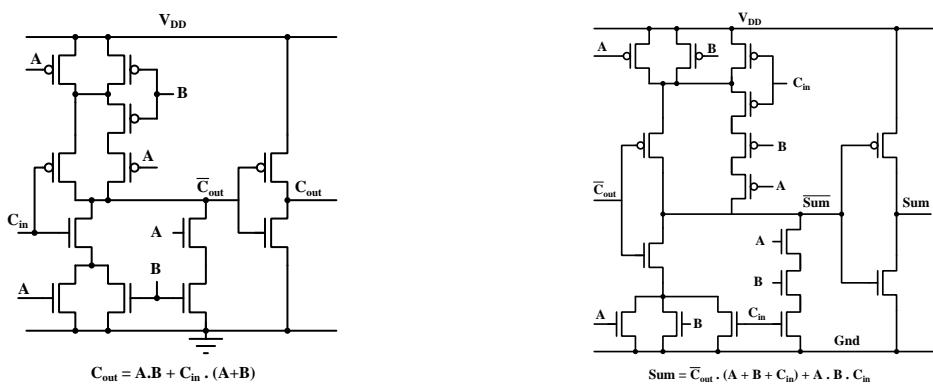


Figure 1.3: CMOS implementation for sum and carry

Both Sum and Carry show an interesting symmetry:

$$\text{sum} = \overline{A} \cdot \overline{B} \cdot C_{in} + \overline{A} \cdot B \cdot \overline{C_{in}} + A \cdot \overline{B} \cdot \overline{C_{in}} + A \cdot B \cdot C_{in} \quad (1.9)$$

$$\overline{\text{sum}} = (A + B + \overline{C_{in}}) \cdot (A + \overline{B} + C_{in}) \cdot (\overline{A} + B + C_{in}) \cdot (\overline{A} + \overline{B} + \overline{C_{in}}) \quad (1.10)$$

$$= (A + A \cdot \overline{B} + A \cdot C_{in} + A \cdot B + B \cdot C_{in} + \overline{C_{in}} \cdot A + \overline{C_{in}} \cdot \overline{B}) \cdot \quad (1.11)$$

$$(\overline{A} + \overline{A} \cdot \overline{B} + \overline{A} \cdot \overline{C_{in}} + \overline{A} \cdot B + B \cdot \overline{C_{in}} + C_{in} \cdot \overline{A} + C_{in} \cdot \overline{B}) \quad (1.12)$$

$$= (A + B \cdot C_{in} + \overline{B} \cdot \overline{C_{in}}) \cdot (\overline{A} + B \cdot \overline{C_{in}} + \overline{B} \cdot C_{in}) \quad (1.13)$$

$$= A \cdot B \cdot \overline{C_{in}} + A \cdot \overline{B} \cdot C_{in} + \overline{A} \cdot B \cdot C_{in} + \overline{A} \cdot \overline{B} \cdot \overline{C_{in}} \quad (1.14)$$

This shows that the **same hardware** that produces sum from A , B and C_{in} , will produce $\overline{\text{sum}}$ if the inputs are changed to \overline{A} , \overline{B} and $\overline{C_{in}}$

$$C_{out} = A \cdot B + C_{in} \cdot (A + B) \quad (1.15)$$

$$\overline{C_{out}} = \overline{A \cdot B + C_{in} \cdot (A + B)} \quad (1.16)$$

$$= (\overline{A} + \overline{B}) \cdot (\overline{C_{in}} + \overline{A} \cdot \overline{B}) \quad (1.17)$$

$$= \overline{A} \cdot \overline{C_{in}} + \overline{B} \cdot \overline{C_{in}} + \overline{A} \cdot \overline{B} \quad (1.18)$$

Thus the carry function also has the same property:

The same hardware which produces C_{out} from A, B and C_{in} , will produce \overline{C}_{out} from $\overline{A}, \overline{B}$ and \overline{C}_{in} .

- In CMOS implementation, we interchange series and parallel configurations for the n and p channel transistors.
- This is to ensure that the pull up and pull down circuits are complementary.
- However, for sum and carry functions, we see that these functions are their own complements.
- Therefore, for implementing sum and carry, we can use the **same** configuration for n and p channel transistors.
- We use this to reduce the number of series connected transistors in pull up/pull down networks.

By making use of symmetry property of sum and carry, it is possible to simplify the implementations. These are called mirror gates because the n and p transistors have the **same**

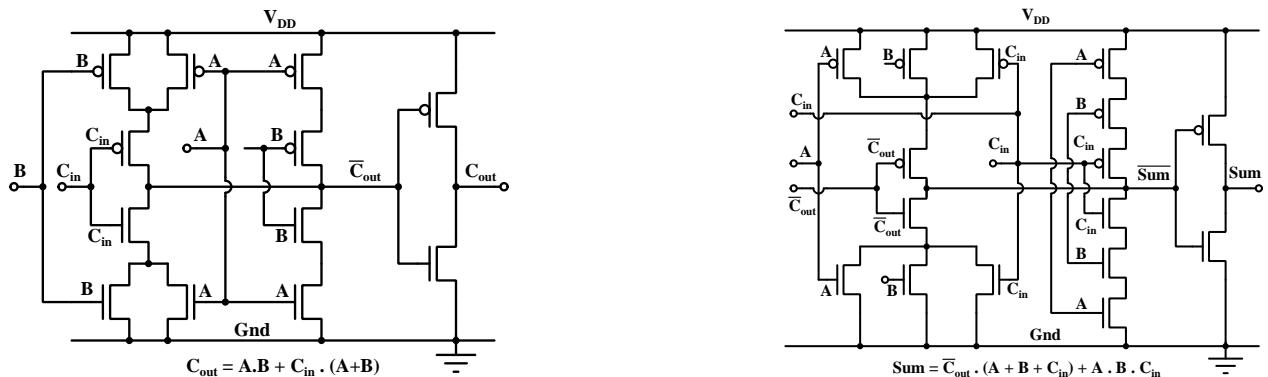


Figure 1.4: Mirror gate implementation of sum and carry

series parallel combination. This is highly unusual.

The worst case delay of the ripple carry adder is linear in number of bits to be added. To reduce the delay per stage, we can eliminate the inverter from the carry output. All even bit adders accept a, b and C_{in} as inputs. The mirror gate gives \overline{C}_{out} as the output. All odd bit adders accept $\overline{A}, \overline{B}$ and \overline{C}_{in} as inputs and thus produce C_{out} as output. Outputs of all bits

are now compatible with inputs of the next stage.

Extra inverters are required to produce $\overline{A}, \overline{B}$ and at the outputs to produce the proper result. However, these are not on the critical path, and do not add to the worst case delay. Extreme care needs to be taken in layout to ensure that the loading on the tree gate producing carry output is as small as possible.

1.4 Carry Look Ahead

Carry propagation is the critical path for a multi-bit adder. To speed up the adder, we would like an architecture where logic terms are classified as those dependent on carry and those which do not depend on carry. Then we can pre-compute all terms which do not depend on carry. When carry arrives, we can quickly compute the output carry and pass it on to the next stage.

Let us analyze what information can be pre-computed from A_i and B_i , which will help us in generating C_{out} quickly from C_{in} .

- When $A_i = 0$ and $B_i = 0$, C_{out} is 0, independent of C_{in} . We define this condition as ‘Kill’. $K = \overline{A} \cdot \overline{B}$
- Similarly, when $A_i = 1$ and $B_i = 1$, C_{out} is 1, independent of C_{in} . We define this condition as ‘Generate’: $G = A \cdot B$.
- Only when $A_i = 0$ and $B_i = 1$ or when $A_i = 1$ and $B_i = 0$, we need to wait for C_{in} to compute C_{out} . In both these cases, $C_{out} = C_{in}$.
- We call this condition as ‘Propagate’, and define $P = A \cdot \overline{B} + \overline{A} \cdot B$.

We define $K = \overline{A} \cdot \overline{B}$, $G = A \cdot B$ and $P = A \oplus B$
Exactly one of K, G or P is true at any time.

When $K = 1$, C_{out} is 0, independent of C_{in} .
When $G = 1$, C_{out} is 1, independent of C_{in} .
When $P = 1$, $C_{out} = C_{in}$.

P is computed using an xor gate, which can be slow. However, the only difference between xor and or logic is when both inputs are 1, i.e. $G = 1$.

If we can ensure that G forces C_{out} to 1 irrespective of P, we can use the simpler ‘or’ logic to compute P.

C_{in} for bit i+1 is the C_{out} of bit i.

So we can write $C_{i+1} = G_i + P_i \cdot C_i$

Notice that the Kill signal is not required.

If $G_i = 0$, $C_{i+1} = A \oplus B = A + B$ when $G = A \cdot B = 0$

If $G_i = 1$, $C_{i+1} = 1$, and the value of P_i does not matter anyway.

So we can use $P = A + B$ instead of $P = A \oplus B$.

Now, we have the sequence:

$$C_{i+1} = G_i + P_i \cdot C_i = G_i + P_i \cdot G_{i-1} + P_i \cdot P_{i-1} \cdot C_{i-1} = \dots$$

and so on, till we reach C_0 .

Since all G_i , P_i and C_0 can be computed in parallel on arrival of the inputs, we can compute all sum and carry terms independently if we do not mind the added complexity.

$$C_{i+1} = G_i + P_i \cdot C_i = G_i + P_i \cdot G_{i-1} + P_i \cdot P_{i-1} \cdot C_{i-1} = \dots$$

Unfortunately, static implementation of these gates has almost as much delay as the ripple carry implementation.

Therefore, the static implementation of computation of sum and carry terms as a logic expression depending on all A_i , B_i and C_0 is rarely used.

However, a dynamic implementation is still useful, and is known as the Manchester Carry Chain.

1.4.1 Manchester Carry Chain

When the clock is low, the output is unconditionally charged by the pMOS.

When the clock goes high, the output will be pulled low if $G = 1$ or if $P = 1$ and $\overline{C_{in}} = 0$.

In all other cases, the output will remain high. Thus this circuit implements the required logic.

This circuit can be concatenated for all bits and since P and G are ready before $\overline{C_{in}}$ arrives, the carry quickly ripples through from bit to bit.

Notice that the nMOS logic can be interpreted as:

$$\overline{P \cdot C_{in} + G}$$

where C_{in} itself has been recursively generated by similar logic.

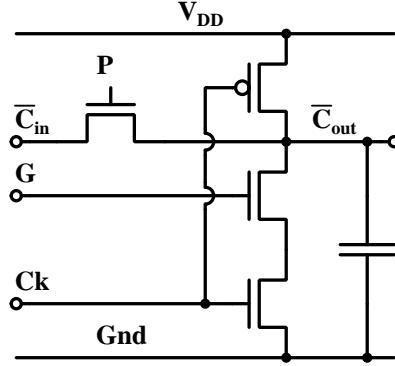


Figure 1.5: Manchester carry chain stage

As in the static case, there is a limit to the number of bits which can be so connected. If $P = 1$ for many successive bits, the discharge path is through series connected pass transistors of all these gates. The discharge time for this critical path has an n^2 dependence.

1.5 Carry Bypass Adder

The worst case for addition occurs when $P = 1$ for all bits and carry has to ripple through all the bits. In carry bypass adder, we form groups of bits and if $P = 1$ for all members of a group, we pass on the carry input to this group directly to the input of the next group, without having to ripple through each bit.

This improves the worst case delay of the adder.

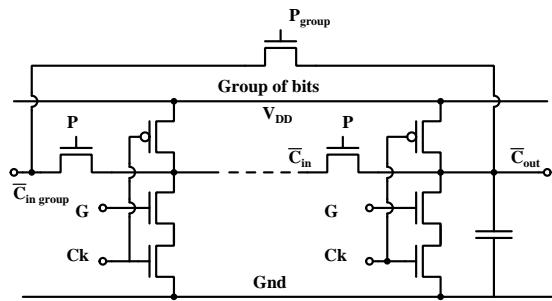


Figure 1.6: Carry By-pass adder

1.6 Carry Select Adder

An m bit carry select adder can be constructed as follows:

- We first compute the generate/propagate/kill signals for each bit (in parallel) from the input bits. Assuming unit gate delay model, this takes one unit of time.
- We use two m bit carry bypass adders. One of the adders assumes the carry input C_{in} to be 0, while the other assumes C_{in} to be 1. The two adders work in parallel and each takes m units of time.
- We now use a multiplexer controlled by the actual C_{in} to select the correct C_{out} . This takes one unit of time.
- The C_{out} of one such m bit adder will be used as the select input of the multiplexer of the next.
- The sum output of each bit is derived from P and C_{out} signals for the corresponding bit and appear one unit of time after C_{out} is available.

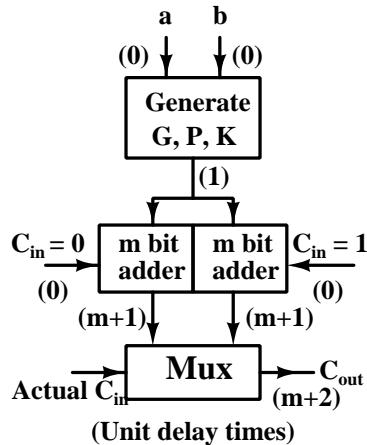


Figure 1.7: Carry select adder

Times of availability of various signals are noted in parentheses in the diagram.

- The two alternatives for the carry output are ready at $(m+1)$ units of time.
- If the actual C_{in} is available at n units of time, the output will be available at $(m+2)$ or $(n+1)$, whichever is later.
- In case of 4 bit adders, this is at 6 units of time or at C_{in} arrival + 1, whichever is later.

1.6.1 Stacking Carry Select Adders

The sub-adders in carry select adder can use any architecture. They could be Manchester carry chains, carry bypass or ripple carry adders. Obviously, these sub adders should not be very long, otherwise, their outputs will be ready after a long time and we shall lose the advantage of carry select additions. Then, how do we make long adders using carry select?

This is done by stacking several smaller carry select adders.

We can stack several identical carry select adders. There is no need for carry select in the first stage, as C_{in} for this stage is available simultaneously with A_i and B_i .

Every subsequent stage will have two sub-adders, one assuming $C_{in} = 0$, the other assuming $C_{in} = 1$. The correct output will be selected by the actual C_{in} when it arrives.

Thus, after the first stage, each group of m bit adders will add only one unit of delay. This is much faster. However, the delay is still linear in number of bits.

A 32-bit adder made by cascading 8 4-bit carry select adders. Notice that the first group

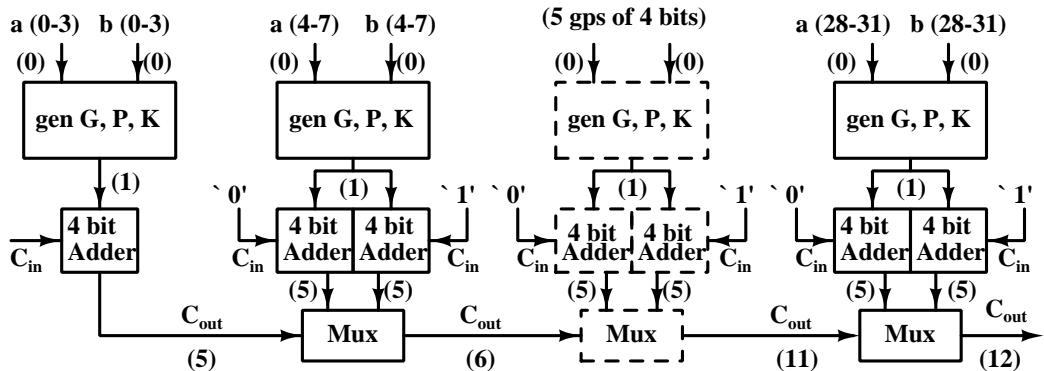


Figure 1.8: Linear stacking of carry select adders

does not need two adders and a mux, since the input carry is known to this group.

Bits	cy in	alt cy.s	cy out
0-3	0	-	5
4-7	6	5	6
8-11	7	5	7
12-15	8	5	8
16-19	9	5	9
20-23	10	5	10
24-27	11	5	11
28-31	12	5	12

The sum generation will take another unit of time, so the overall results will be available in 13 units of time.

Can we speed up the adder if we don't use the same no. of bits in every stage? In linear stacking, since all adders are identical, they are ready with their alternative outputs at the same time. But the carry arrives later and later at each successive group of carry select adders. We could have used this extra time to add up more bits in the later stages, and still be ready with the alternative results before carry arrives! Since the carry arrives one unit of time later at each successive group, each successive group could be longer by one bit.

We can do more bits of addition in the same time, if each successive stage is 1 bit longer than the previous one. Since the first stage does not add a mux delay, the first two stages will use the same number of bits. Thus, the number of bits which can be added is given by

$$m = n_0 + n_0 + (n_0 + 1) + (n_0 + 2) + \dots = n_0 + \frac{s(n_0 + n_0 + s - 1)}{2}$$

where s is the number of stages following the first one without carry select.

The total delay will be $n_0 + 1$ for the first stage. Each subsequent stage takes just 1 unit of time since the candidates for selection are available just in time. Thus the time taken is just $n_0 + s + 1$ units. When $s \gg n_0$, we have $m \approx s^2/2$, while the time taken is nearly s .

Thus the time taken to add m bits is $\approx \sqrt{2m}$

For a 32 bit adder, we could use a distribution like: 4,4,5,6,7,6.

Bits	carry in	carry alternatives	carry out
0-3	0	4	5
4-7	5	5	6
8-12	6	6	7
13-18	7	7	8
19-25	8	8	9
26-31	9	8	10

Our sum will be ready at 11 - which is much faster. This gain will be even higher for wider additions.

In square root tiling, the size of the sub-adders can become quite large for the later adders. Each of these can therefore be constructed as independent adders with any architecture. In particular, we might construct the sub-adders themselves as tiled carry select adders. Since these are faster, we can accommodate more bits in each of these stages. Thus the bits processed increase faster than square of the number of stages. In the asymptotic limit, the time for addition can be reduced to be logarithmic in the number of bits, rather than just square root.

However, it is possible to complete the addition in logarithmic time using tree adders, with much less complexity. These are the adders used in most modern systems using wide adders.

1.7 Tree Adders

Tree adders use the idea of carry look ahead addition. However, these do not try to implement the complex logic expressions which result from looking ahead. Instead, these build up the logic in a tree like structure, where each node performs simple logic operations.

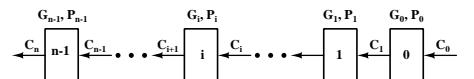
Recall that we had defined

$$K = \overline{A} \cdot \overline{B}, \quad G = A \cdot B \quad \text{and} \quad P = A \oplus B \quad (1.19)$$

When $K = 1$, $C_{out} = 0$, while if $G = 1$, $C_{out} = 1$, irrespective of C_{in} .

When $P = 1$, $C_{out} = C_{in}$ and this is the only case when we must wait for C_{in} to compute C_{out}

Consider an n bit adder with the least significant bit indexed as 0 and the most significant bit as $n - 1$. The i 'th bit accepts C_i as input carry and produces C_{i+1} as output carry.



We can express the output of the i 'th stage as:

$$C_{i+1} = G_i + P_i \cdot C_i \quad (1.20)$$

Substituting for C_i , which is the output of cell no. (i-1),

$$\begin{aligned} C_{i+1} &= G_i + P_i \cdot (G_{i-1} + P_{i-1} \cdot C_{i-1}) \\ &= G_i + P_i \cdot G_{i-1} + P_i \cdot P_{i-1} \cdot C_{i-1} \end{aligned}$$

If we try to compute the output carry for each bit by extending this logic to a function of $G(i)$, $P(i)$ and C_0 , the expressions become very complex and their implementation will be slow. However, we can divide this work in a tree structure and that eventually leads to faster adders. Recall that

$$C_{i+1} = G_i + P_i \cdot G_{i-1} + P_i \cdot P_{i-1} \cdot C_{i-1}$$

We can divide the expression for the output carry in parts which are independent of input carry and parts which are dependent on it. Let us call the original single bit G and P values for the i 'th cell as G_i^1 and P_i^1 . Then we can define

$$G_{i,i-1}^2 \equiv G_i^1 + P_i^1 \cdot G_{i-1}^1 \quad \text{and} \quad P_{i,i-1}^2 \equiv P_i^1 \cdot P_{i-1}^1 \quad (1.21)$$

This permits us to write

$$C_{i+1} = G_{i,i-1}^2 + P_{i,i-1}^2 \cdot C_{i-1} \quad (1.22)$$

Thus we can compute C_{i+1} from C_{i-1} directly using the same logic as before, except that we use $G_{i,i-1}^2$ and $P_{i,i-1}^2$ instead of G_i^1 and P_i^1 .

Notice that the logic needed to compute $G_{i,i-1}^2$ from G_i^1 and P_i^1 is the same as that used to compute the output carry from input carry using G and P values of any order. Effectively, we have created a cell which is equivalent to two original adder cells and the carry has to be passed in groups of 2 bits across the adder. Of course, we need two stages to compute the second order G and P values, but these values are independent of carry and can be computed in parallel for all bit pairs.

Continuing the same process, we can combine two second order G and P values to compute third order G and P , which will permit computation of C_{i+1} directly from C_{i-3} .

$$G_{i,i-3}^2 \equiv G_{i,i-1}^2 + P_{i,i-1}^2 \cdot G_{i-2,i-3}^2 \quad \text{and} \quad P_{i,i-3}^3 \equiv P_{i,i-1}^2 \cdot P_{i-2,i-3}^2 \quad (1.23)$$

$$C_{i+1} = G_{i,i-3}^3 + P_{i,i-3}^3 \cdot C_{i-3} \quad (1.24)$$

Thus carry can now be passed over groups of 4 bits. Notice that the group size over which the carry can be computed directly *multiplies* by two each time we use a higher order for G and P values, while the time to compute the required higher order G and P values *increments* by one gate delay for logic $A + B \cdot C$ (for G) or $A \cdot B$ (for P). This is what results in ultimate

time to generate the final carry being logarithmic in the number of bits being added.

Since the generation of final carry has been speeded up substantially, we have to re-examine our assumption that carry propagation is the critical step in adder design. Addition is not complete unless all the sum bits and the terminal carry have been generated. In principle, we do not need the internal carries at each bit for the final result. The sum values at each bit are:

$$S_i = A_i \oplus B_i \oplus C_i = P_i \oplus C_i \quad (1.25)$$

For generating sums using this relation, we do need the internal carries at each bit. These can be computed using relations of the type described in equations 1.20, 1.22 and 1.24 etc. Different architectures have been described in literature for the order of computation of G , P , C_{out} and Sum bits. All of these compute the final sum in times which are logarithmic functions of the number of bits. For wide adders, these can be much faster than other architectures.

1.7.1 Brent Kung adder

To illustrate the operation of tree adders, we use the architecture described by Brent and Kung for a tree adder. Many other tree adders have been described in the literature and we use this one as an example. In this adder, we compute the P and G values in a tree fashion, as shown in fig.1.9 for an 8 bit adder.

From the values of a_i, b_i , we first calculate P_i^1, G_i^1 , with $i = 0 \dots 7$. Next, using these values,

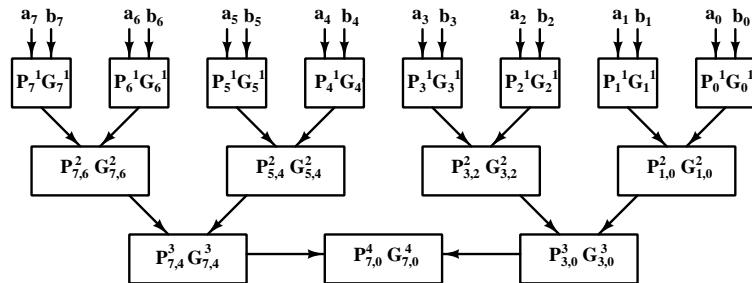


Figure 1.9: Generation of P and G values in Brent Kung adder

we can generate $P_{2i+1,2i}^2, G_{2i+1,2i}^2$ with $i = 0 \dots 3$. In the next step, we use these values to generate $P_{4i+3,4i}^3, G_{4i+3,4i}^3$ with $i = 0, 1$. Finally, using these values, we can compute $P_{7,0}^4, G_{7,0}^4$.

As soon as values of P and G terms of various orders are known, we can compute the values of carry outputs which depend on these and the input carry.

$$C_1 = G_0^1 + P_0^1 \cdot C_0, \quad C_2 = G_{1,0}^2 + P_{1,0}^2 \cdot C_0$$

$$C_4 = G_{3,0}^3 + P_{3,0}^3 \cdot C_0, \quad C_8 = G_{7,0}^4 + P_{7,0}^4 \cdot C_0$$

When these carry values are valid, the other carry values which depend on these can be generated.

$$C_3 = G_2^1 + P_2^1 \cdot C_2, \quad C_5 = G_4^1 + P_4^1 \cdot C_4 \quad C_6 = G_{5,4}^2 + P_{5,4}^2 \cdot C_4,$$

Finally, C_7 can be generated from C_6 .

$$C_7 = G_6^1 + P_6^1 \cdot C_6$$

With all carry values generated, the corresponding sum values can be calculated using the relation $\text{Sum}_i = P_i^1 \oplus C_i$.

1.7.2 Other logarithmic adders

Brent Kung adder has low complexity and fanout from each stage. Other tree adders, (such as Kogge Stone adder) can be faster, following a similar scheme but with higher fanout and wiring congestion.

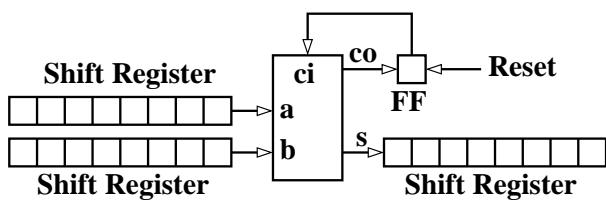
Rather than using radix 2 for generating P, G values, we can use radix 4. In this scheme, we use more complex logic for combining 4 P, G values every time to generate the next level P, G values. Thus, we can generate $P_{3,0}$ and $G_{3,0}$ directly from A_i, B_i . This reduces the depth of the tree, but at the same time, logic for each combining stage is more complex.

The other choice in logarithmic adders is for sparsity. The only use for generating bit-wise internal carries is for computing the sum bits. However, we may choose to generate only the even bit carries and compute sums for even as well odd bits using only these using more complex logic expressions.

1.8 Serial Adders

Up to now, we have been concerned with making fast adders, even at the cost of increased complexity and power. In many applications, speed is not as important as low power consumption and low cost.

Serial adders are an attractive option in such cases. A single full adder is used. If numbers to be added are available in parallel form, these can be serialized using shift registers.



A single full adder adds the incoming bits from operands A and B. Bits to be added are fed to it serially, LSB first.

- The sum bit goes to the output while carry is stored in a flip-flop.
- Carry then gets added to the next more significant bits which arrive in the next clock cycle.
- Output can be converted to parallel form if needed, using another shift register.

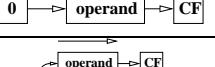
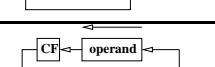
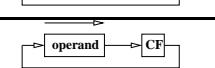
Chapter 2

Shift and Rotate Operations

We often need to shift and rotate operands in order to perform various operations such as serialization/de-serialization of data, multiplication etc.

2.1 Shift and Rotate Operations

The following operations are often required for processing of data:

SHL	op, count	
SAL	op, count	Same as SHL
SHR	op, count	
SAR	op, count	
ROL	op, count	
ROR	op, count	
RCL	op, count	
RCR	op, count	

These operations can be implemented by bi-directional shift registers with some control logic which provides circuits to choose the value and point of entry of new bits. However, this implementation can be very slow for a large number of shifts.

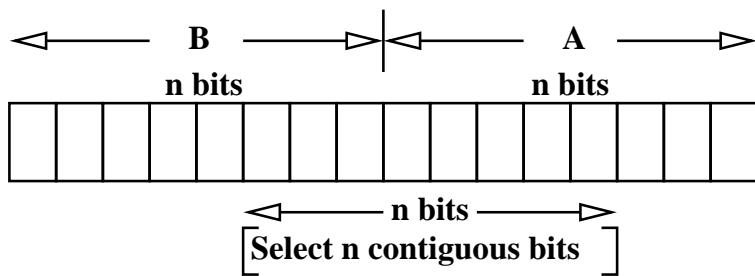
2.2 Barrel Shifters

Shift and rotate operations involve no computation. So why should we spend a large number of clock cycles for performing these operations? Ideally, we would like a shifter which produces the result in a single clock cycle.

2.2.1 Shift/Rotate as Select Operations

We can view Shift/Rotate operations as selection operations. In that case, the output can be produced directly by selecting the correct bits to appear in the desired position at the output.

Imagine two words A and B positioned as shown in the figure below.



We just have to choose B and A appropriately and select the correct range of n contiguous bits to implement various shift or rotate operations.

For all rotate operations, we choose $B=A=\text{data}$. For Shift Left, we choose $B=\text{data}$ and $A=0$. For Logical Shift Right we choose $B=0$ and $A=\text{data}$. For Arithmetic Shift Right, we make $B = \text{replicated MSB of data}$, and set $A = \text{data}$.

Of course we do not actually copy data bits to A/B. Each output bit is produced by a mux which picks out the correct input data bit.

2.3 Barrel Shifters

Shifters which produce outputs as select operations are called barrel shifters. The name comes from viewing the inputs as well as outputs as a circular arrangement of bits. The shifter then connects the input circle to the output circle like the sections of a barrel.

A brute force implementation of such a shifter will require n multiplexers of n bits each, where the control inputs for each multiplexer are generated from the amount and type of shift/rotate that is desired. This requires quite a large number of complex n way multiplexers and puts a heavy load on data bits. Therefore rather than trying to complete the entire

operation in one go, we use logarithmic barrel shifters, which are not so complex and take of the order of $\log_2 n$ operations to produce the result

2.3.1 Logarithmic Barrel Shifters

The control logic of a one step barrel shifter is complex because the amount of shift is variable. The loading on data lines and control logic complexity can be reduced if we break up the shift/rotate process into parts. We can carry out shifts in different stages, each stage corresponding to a single bit of the binary representation of the **shift amount**. Now the i 'th stage needs to produce either no shift (if the corresponding control bit is '0') or a shift by a constant amount which is 2^i . Thus each stage requires only a two way mux for each bit. For example, a shift by 6 (binary: 110) will be carried out by first doing a 4 bit shift and then a 2 bit shift.

Since we need n bits to represent a maximum shift amount of $2^n - 1$ places, the number of bits to express the shift amount (and hence the number of shift stages required) is logarithmic in the maximum shift desired.

That is why such shifters are called Logarithmic Barrel Shifters. We can optionally buffer the outputs after each stage.

Bit i of the **shift amount** represents no shift (if it is 0) and a constant shift by 2^i places (if it is 1). If the amount of shift is fixed, the required bit can just be wired from the input bits.

The 2 way mux is controlled by bit i of the **shift amount**, Using this, we can choose either the unshifted operand bit or the operand bit 2^i places away from it.

This can be done for all bits of the operand in parallel. This constitutes one stage of the logarithmic shifter. The output can then be shifted again in the next stage, controlled by the next significant bit.

2.3.2 Right Rotate for an 8 bit Operand

For an 8 bit operand, the amount of rotation will be between 0 to 7 places, which can be represented by 3 bits. So we need a 3 stage implementation.

To perform a right rotate operation, we use the scheme shown in Figure 2.1.

Each input bit of the count drives eight 2-way muxes, one for each bit of the operand. At each stage, the muxes select either the unshifted bit or a bit 2^n places from it. A total of 3 stages are required for 0 to 7 bits of shift.

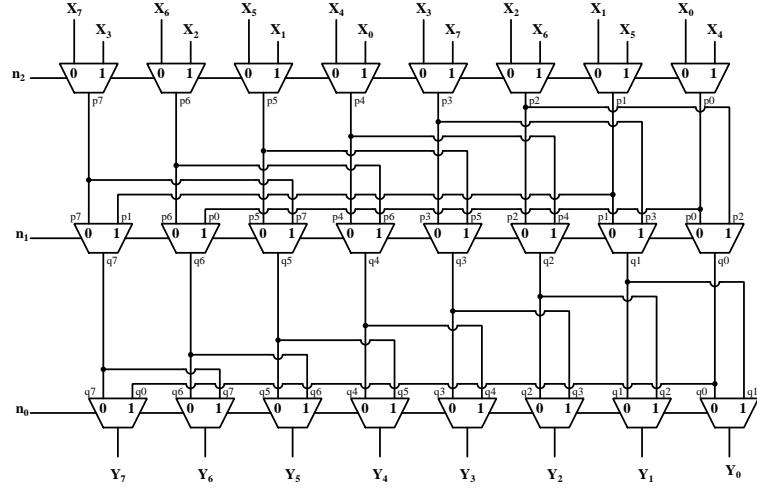


Figure 2.1: Right rotate for an 8 bit operand

2.3.3 8 bit Logical Shift Right

If we need a shift instead of a rotate, we feed a 0 instead of the corresponding data bit. We

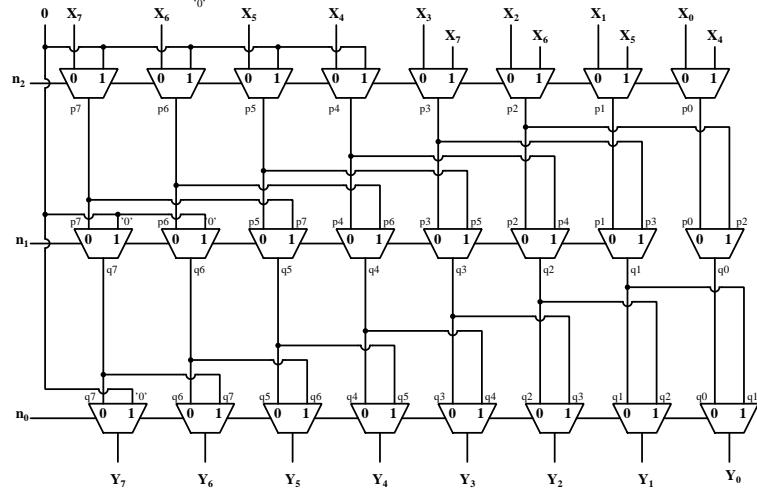


Figure 2.2: Logical Right Shift for an 8 bit operand

need to input 0's instead of data bits to the first 4 muxes in the first stage, to the first 2 muxes in the second stage and to 1 mux in the last stage.

2.3.4 Combining Rotate and Shift Operations

We can combine the circuits for rotate and shift functions by putting muxes where different inputs need to be presented for the two functions. Further, we can include the Arithmetic

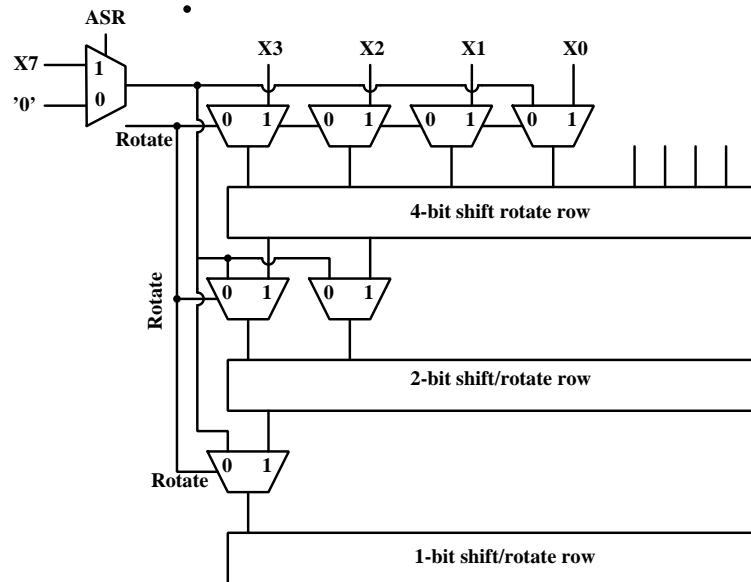


Figure 2.3: Combined Right Rotate/Shift for an 8 bit operand

Shift function by choosing between 0 or X7 as the bit to be inserted from the right.

2.3.5 Rotate and Shift by Masking

We can also combine the rotate and shift functions by masking. We use the rotate function which does not lose any information, as the primary circuit. Now we can mask n bits at the left to 0 if a right shift operation was desired instead. In case of an arithmetic shift, n bits on the left have to be set to the same value as X7.

Shift/Rotate Left case is similar, except that the Logical and Arithmetic shifts are not different operations.

2.3.6 Bidirectional Shift and Rotate Operations

It is possible to use the same hardware for left and right shift/rotate operations. This can be done by adding rows of muxes at the input and output which reverse the order of bits.

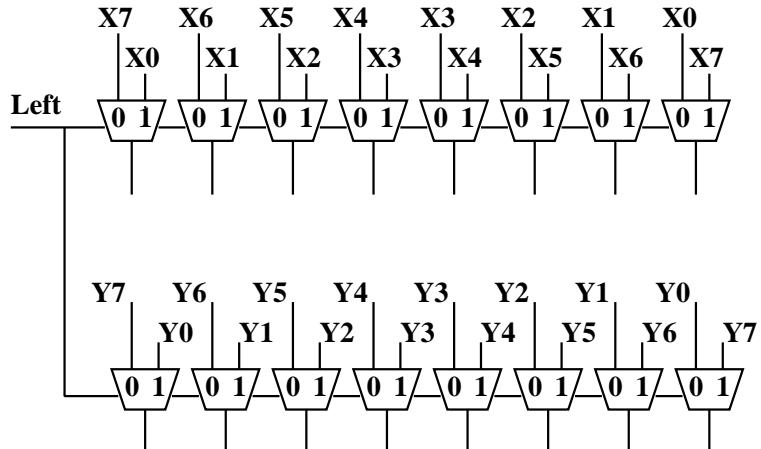


Figure 2.4: Bit reversal for bidirectional Shift and Rotate Operations

We can also make use of the fact that a left rotate by n places is the same as a right rotate by $2^n - n$ places. Now $2^n - n$ is just the 2's complement of n . By presenting the 2's complement of n at the mux controls, we can convert a right rotate to a left rotate. This can be followed by a mask operation, if a shift operation was required, rather than a rotate.

Chapter 3

Multipliers

3.1 Shift and Add Multipliers

An obvious way for implementing multipliers is to replicate the paper and pencil procedure in hardware.

$$\begin{array}{r} \text{○○○○} \\ \times \text{○○○○} \\ \hline \text{○○○○} \\ \text{○○○○} \\ \text{○○○○} \\ \text{○○○○} \\ \hline \text{○○○○○○○○} \end{array}$$

Figure 3.1: Shift and add multiplier

- Initialize the product to 0, extend multiplicand to left by n bits filled with 0s.
- If the least significant bit of the multiplier is 1, add the multiplicand to product, else do nothing.
- Shift the multiplier right by one bit.
- Shift the multiplicand left by one bit.
- Repeat for n bits

Each term being added to form the product is called a partial product. The name “partial product” is also used for individual bits of the terms being added - so beware!

The paper-pencil procedure requires $n-1$ additions to a $2n$ bit accumulator. This uses a single adder, but takes long to complete the multiplication. A 32×32 multiplication will require 31 addition steps to a 64 bit accumulator.

Multiplication can be made faster by using multiple adders and adding terms in a tree structure.

3.2 Array Multipliers

Suppose we want to multiply two n -bit numbers A and B , where

$$A = \sum_{i=0}^n 2^i a_i \quad B = \sum_{j=0}^n 2^j b_j$$

We can regard all bits of the partial products as an array, whose (i,j) th element is $a_i \cdot b_j$. Notice that each element is just the AND of a_i and b_j . All elements of the array are available in parallel, within one gate delay of arrival of A and B . We can now use an array of full adders to produce the result. One input of each adder is the sum from the previous row, the other is the AND of appropriate a_i and b_j . This architecture is called an array multiplier.

A 4×4 array multiplier is shown in fig.3.2. Half adders can be used at the right end.

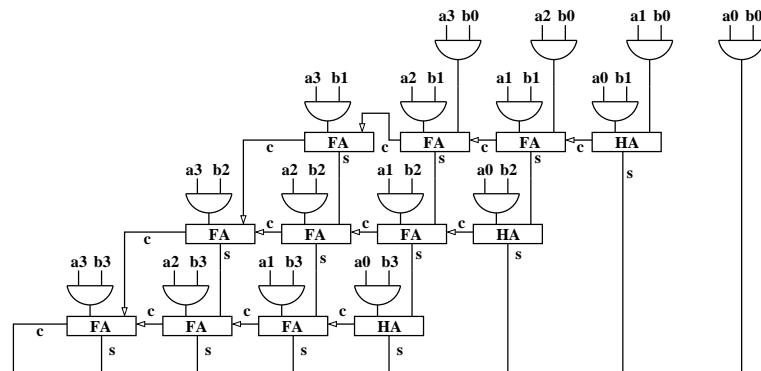


Figure 3.2: Array multiplier

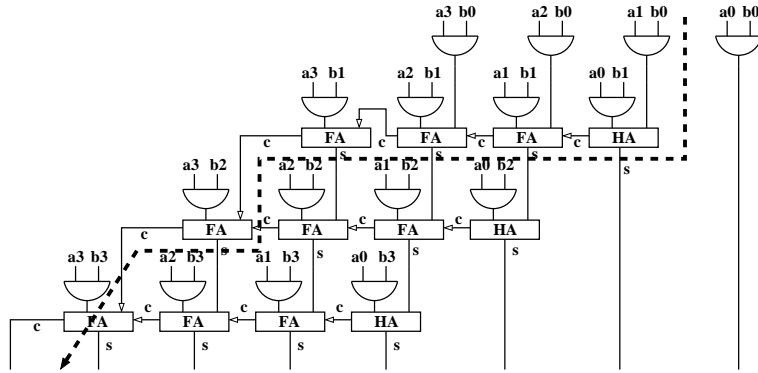


Figure 3.3: Critical path in an array multiplier

3.2.1 Critical Path through an Array Multiplier

The critical path through a 4X4 array multiplier is shown in fig.3.3 The critical path involves carry as well as sum outputs!

3.3 Speeding up Multipliers

The array multiplier has a regular layout with relatively short connections. However, it is still rather slow. How can we speed up a multiplier?

There are two possibilities:

- Somehow reduce the number of partial products to be added. For example, could we multiply 2 bits at a time rather than 1?
- Since we have to add more than two terms at a time, use an adder architecture which is optimized for this.

3.4 Booth Encoding

Booth Encoding reduces the number of partial products by multiplying 2 bits at a time.

Let the multiplicand be A and the multiplier B. Rather than multiplying A with successive bits of B, we can multiply it with two bits of B at a time. Depending on the two bits being 00, 01, 10 or 11, the partial product will be 0, A, 2A or 3A.

- 0 and A can be produced trivially.

- $2A$ can be produced easily by a left shift of A .
- Generating $3A$ presents a problem!

However, $3A$ can be expressed as $4A - A$. The task of adding $4A$ is passed on to the next group of 2 bits of the multiplier. Since the place value of the next group of 2 bits is 4 times the current one, adding $4A$ to the product is equivalent to adding 1 to the next group of 2 bits of the multiplier. $-A$ can be generated from A , using an adder/subtractor rather than an adder for accumulating the sum of partial products.

3.5 Modified Booth Encoding

To simplify the logic for deciding whether an additional $4A$ should be added on behalf of the less significant 2 bits in the multiplier, we express $2A$ also as $4A - 2A$.

Since we anyway have an adder-subtractor, this requires no additional resources. The modified logic is:

- for 00, do nothing.
- for 01, add A .
- for 10, subtract $2A$, ask the next group to add $4A$.
- for 11, subtract A , ask the next group to add $4A$.

Now the next group can just look at the more significant bit of the previous group and add 1 to the multiplier if it is ‘1’.

The partial product generator looks at the current 2 bits and the MSB of the previous group of 2 bits to decide its action. Thus, we scan the multiplier 3 bits at a time, with one bit overlapping. For the first group of 2 bits, we assume a 0 to the right of it. After handling the previous group, the multiplicand is shifted left by 2 positions. Thus, it has already been multiplied by 4. Therefore, adding $4A$ on behalf of the previous group is equivalent to adding 1 to the multiplier corresponding to the current group.

Table 3.1 summarizes the effective multiplier for generating the partial product. The partial product generator looks at the current 2 bits and the MSB of the previous group of 2 bits to decide its action. Thus, we scan the multiplier 3 bits at a time, with one bit overlapping. For the first group of 2 bits, we assume a 0 to the right of it. After handling the previous group, the multiplicand is shifted left by 2 positions. Thus, it has already been multiplied by 4. Therefore, adding $4A$ on behalf of the previous group is equivalent to adding 1 to the multiplier corresponding to the current group. Notice that a 111 in the 3 bit group being scanned requires no work at all.

What happens if there is a string of ‘1’s in the multiplier? Consider multiplication by $111 \dots 111$. As described earlier, we should add implied zeros to the right and left of this multiplier.

Current 2-bits	Multiplier for these	Previous MSBit	Pending Increment	Total Multiplier
00	0	0	0	0
01	+1	0	0	+1
10	-2	0	0	-2
11	-1	0	0	-1
00	0	1	+1	+1
01	+1	1	+1	+2
10	-2	1	+1	-1
11	-1	1	+1	0

Table 3.1: Effective multipliers for Modified Booth Algorithm

- Because the group begins with 110, there will be a -1 in the beginning.
- The group ends with 011. So there will be a +2 at the end,
- However, for the length of continuous '1's, nothing needs to be done (add zeros).

Curr. 2 bits	Prev. MSB	Multi- plier
00	0	0
00	1	+1
01	0	+1
01	1	+2
10	0	-2
10	1	-1
11	0	-1
11	1	0

Thus Booth encoding reduces the number of partial products to about half (multiplying 2 bits at a time). It also makes addition in columns of partial products fast because carry propagation during addition will be reduced.

3.6 Efficient Addition of Partial Products

Multipliers can be speeded up by using special adder architectures which are optimized for adding more than two numbers. One option is to use tree adders rather than an accumulator. Several additions proceed in parallel, since all partial products are generated together. Fig.3.4 shows the use of tree adders in a multiplier.

3.6.1 Carry Save Adders

Ordinary adders are large and complex. Also, these are slow due to rippling of carry. Let us consider an adder which presents its output not as one word - but two. The actual result is



Figure 3.4: Tree additions in multipliers

the sum of these.

Obviously, this kind of adder is of no use for adding just two words! But it can be useful in a multiplier where we are adding multiple terms. For each bit column, the sum goes into one output word, while carry outs go into the other (without being added to the next more significant column). Now there is no rippling of carry and the output is available in constant time.

We do need a conventional adder in the end to add these two words. This type of adder, which reduces the product to two words which must be added using a conventional adder is called a “Carry Save Adder” or CSA. For example, we can construct a useful CSA for adding

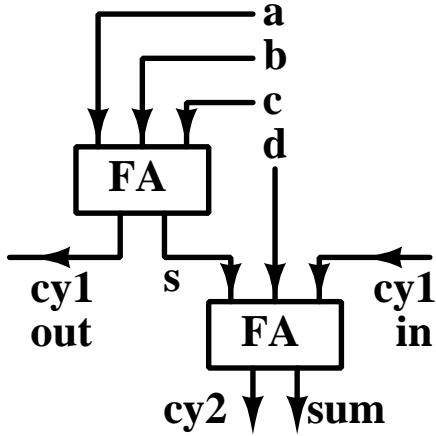


Figure 3.5: Carry save adder

4 bits in the same column. The 4 input 2 output CSA uses two full adders as shown in fig.3.5: We make use of the fact that all partial product bits are available in constant time after the application of inputs. Since there are 4 bits to be added, we feed three of them to a full adder.

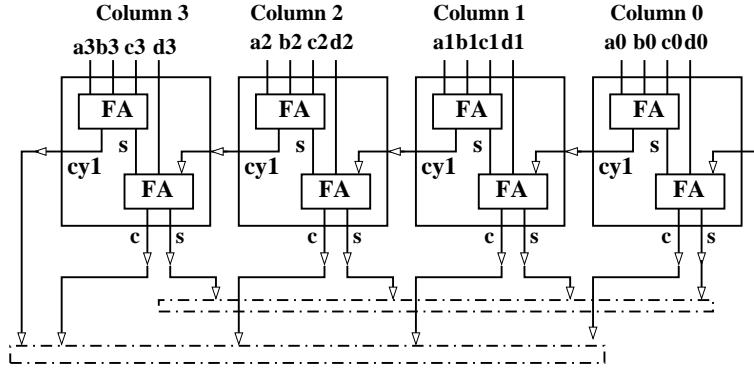


Figure 3.6: Tiling Carry Save Adders

The sum and carry output of this adder is then available in constant time.

The sum output of first FA goes to the second FA. The carry output (cy1) of the first FS goes as intermediate input to the CSA used in the column to the left of this one.

The second FA accepts one left over bit from the partial product column, the sum output of the first Full Adder FA1 and cy1 output coming from the CSA to its right. All inputs to this Full Adder are also available in constant time. Notice that even though cy1 goes from one column to the next significant column, **it does not ripple all the way** horizontally. It goes to FA2 of the more significant column whose output is not required by the next column.

3.6.2 Critical Path of Carry Save Adders

Figure 3.6 shows how we can add 4 columns of 4 bits each.

Rows are labeled as a,b,c and d while columns are indexed as 0,1,2 and 3. Outputs are collected in two separate registers (shown in dotted lines). These must be added using a conventional adder. Critical path of a 4x4 Carry Save Adder is shown in fig.3.7. One can see that the critical path has been broken up. Addition of 4 words of 32 bits each will have a critical path of the same length as that for 4 words of 4 bits each.

3.7 Wallace Multipliers

Multipliers do not have the same number of bits in every column. In 1964, Wallace proposed a method for a carry save adder like reduction scheme which is valid for columns of variable size. Wallace multiplier assumes adders which take multiple inputs of the same weight and produce sum outputs of the same weight and carry outputs with higher weights. These are combined in stages to reduce the number of terms at each weight to 2 or less. These two

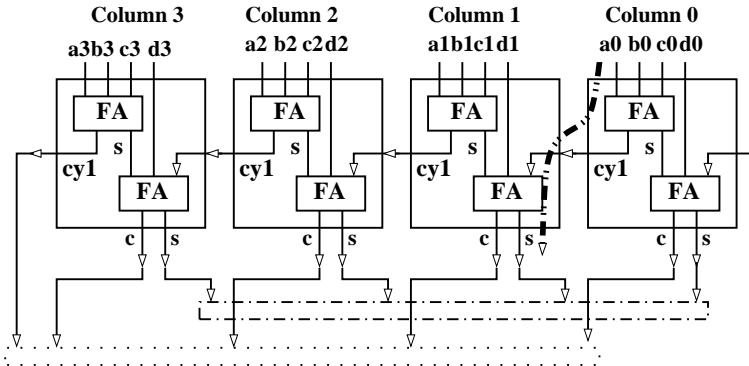


Figure 3.7: Critical path of a carry save adder

terms are then added by a conventional adder to produce the final result.

Wallace multipliers act in three stages:

1. Generate all bits of the partial products in parallel.
2. Collect all partial products bits with the same place value in bunches of wires and reduce these in several layers of adders till each weight has no more than two wires.
3. For all bit positions which have two wires, take one wire at corresponding place values to form one number, and the other wire to form another number.

Add these two numbers using a fast adder of appropriate size.

We assume that Full adders and Half adders will be used. A full adder takes 3 inputs and produces one output of the same weight (sum) and another of higher weight (carry). This is called a (3,2) adder. It reduces the number of wires at its own weight by 2 and adds one wire at the higher weight.

A half adder takes 2 inputs and produces one output of the same weight (sum) and another of higher weight (carry). This is a (2,2) adder. It reduces the number of wires at its own weight by 1 and adds one wire at the higher weight.

3.7.1 Reduction Stage of Wallace Multipliers

The reduction algorithm is general and can be used with any adders of type (n,m). For example, a carry save adder is of type (4,2).

- Each reduction stage looks at the number of wires for each weight and if any weight has more than 2 wires, it adds a layer of adders.
- When the numbers of wires for each weight have been reduced to 2 or less, we form one number with one of the wires at corresponding place values and another with the other wire (if present).
- These two numbers are added using a fast adder of appropriate size to generate the final product.

The original paper by Wallace did not give a reduction algorithm in detail. Subsequently, the Wallace reduction algorithm has been interpreted in many ways. We first describe the algorithm as interpreted by Dadda and other authors in many papers on multipliers. This, however, produces redundant most significant bits for multipliers of some sizes. For example, an 8x8 bit multiplier following this scheme will produce 17 bits. Of course, the 17th bit will always come out to be '0', because the product can be no wider than 16 bits. Since this scheme is widely described in literature, we shall first give the details of this scheme. Subsequently, we give a Wallace wire reduction scheme which does not lead to a redundant bit.

Wire Reduction Scheme for Wallace Multipliers

If any weight contains more than two wires, we use a reduction algorithm to reduce the number of wires. We divide the total number of rows to be added in groups of 3 rows. Any rows which are additional to these groups are passed on to the next stage as they are.

Next we take groups of 3 rows one at a time. If there are 3 wires in any column of this group, we place a full adder. The sum output of this adder goes to the same column in the next stage. The carry output of the adder goes to the column with next higher weight. If a column has two wires, it is reduced with a half adder. Again the sum output goes to the same column in the next stage and the carry wire joins the column with higher weight. If a column has a single wire, it is passed through to the next stage.

This is repeated for all groups of 3 rows.

When all groups of three rows have been reduced this way, we count wires at all weights and continue this procedure till no weight has more than two wires.

At the end, many contiguous bits at the least significant end will have a single wire. These are carried through to the result. For bits which have two wires, one is allocated to one word and the other to another word, and these two words are added using a fast adder.

3.7.2 Wallace Multiplier Example

Consider a multiplier for 4X4 bits. Partial products are generated in parallel and the number of wires at each weight are shown in the table on the right.

We shall use this example to introduce the “dot convention” used for representing wire reduction in many multipliers.

Bit	Terms	Wires
0	a0b0	1
1	a0b1, a1b0	2
2	a0b2, a1b1, a2b0	3
3	a0b3, a1b2, a2b1, a3b0	4
4	a1b3, a2b2, a3b1	3
5	a2b3, a3b2	2
6	a3b3	1

4X4 Wallace Multiplier: First Reduction

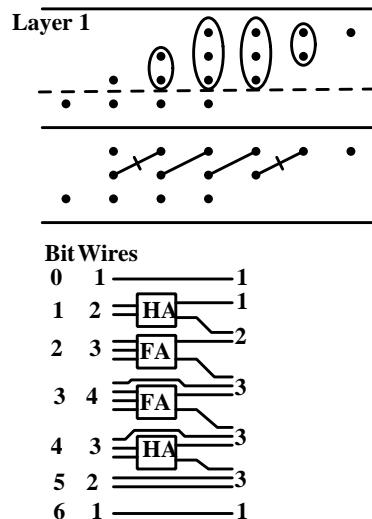


Figure 3.8: First reduction stage of a 4x4 Wallace multiplier

Figure 3.8 shows a 4x4 Wallace multiplier. This multiplier has 4 rows of partial products. The upper three rows are grouped together and can have 1, 2 or 3 wires. the bottom row is just passed through to the next stage.

Each dot in the upper diagram represents a bit at its respective weight. When we place a full adder to reduce a column, a dot representing the sum is placed at the same weight in the next stage. A dot representing carry is placed at the next higher weight. These two dots are joined by a line to show that these two are the results from a full adder. Similarly, when we use a half adder, we place a dot in the same column for the sum and a dot in the next higher weight column for the carry. These are joined with a crossed line to show that these are from a half adder. Bits which are passed through to the next stage are represented by dots not

connected to any line.

After grouping the original 4 rows of partial products in groups of 3 and 1:

Bit 0 has a single wire: which is passed through.

Bit 1 has 2 wires: which are fed to a half adder.

Bits 2 and 3 have 3 wires each, which are fed to full adders.

Bit 4 has 2 wires (in the group of 3), which are fed to a half adder.

Bit 5 has 1 wire, which is passed through.

4X4 Wallace Multiplier: Second Reduction

After first reduction, there are three rows of wires (at bits 3, 4 and 5). So we need another reduction stage. The three rows form a group and there are no passed through rows in this stage. Bits 0 and 1 have single input wires, which are passed through.

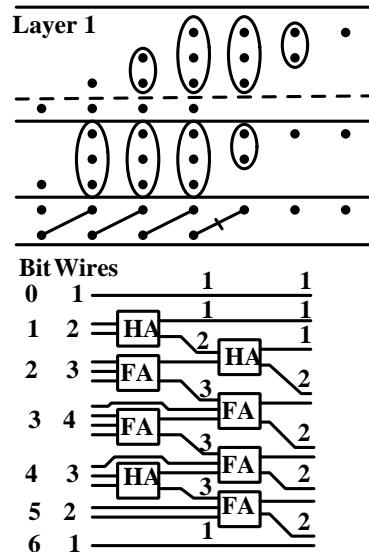


Figure 3.9: Second reduction stage of a 4x4 Wallace multiplier

Bit 2 has 2 wires. These are reduced using a half adder. The sum goes as the only output wire at this bit, while the carry goes to bit 3.

Bits 3, 4 and 5 have 3 input wires each. These are reduced using full adders. Thus these bits have two wires at their outputs – one from the sum of their full adder and the other from the carry produced by the adder at the lower weight.

Bit 6 has one input wire. This is joined by the carry of the adder at bit 5 to produce 2 output wires.

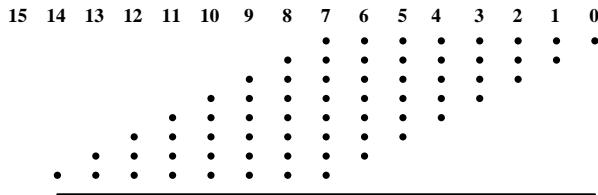
Final Addition

After the second layer, no weight has more than 2 wires. Single wires at bits 0, 1 and 2 are fed through to the output.

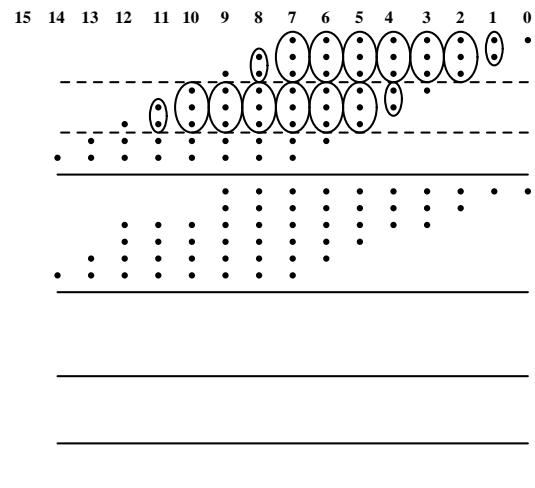
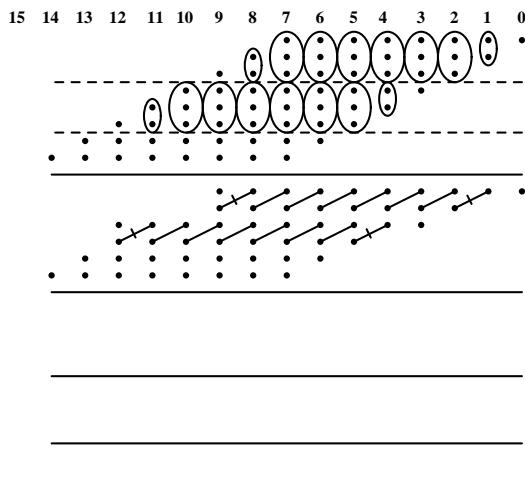
- A fast conventional adder is used to add the 2 bits each at bits 3, 4, 5 and 6.
- Notice that we do not need a full width fast adder. This is because the half adders at low weights have already rippled the carry while the rest of weights were being reduced.
- This makes the final adder smaller and faster.

3.7.3 Redundant MSB in large Wallace Multipliers

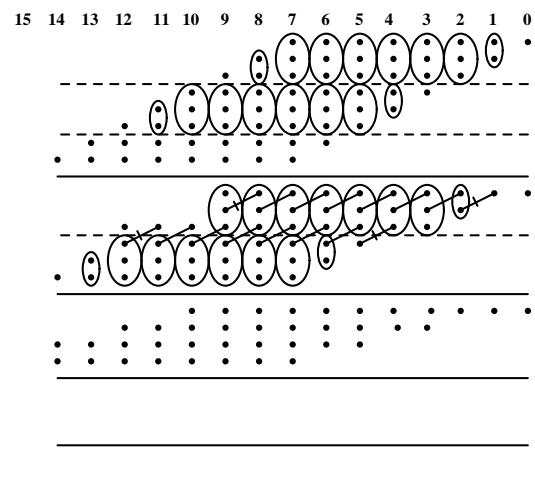
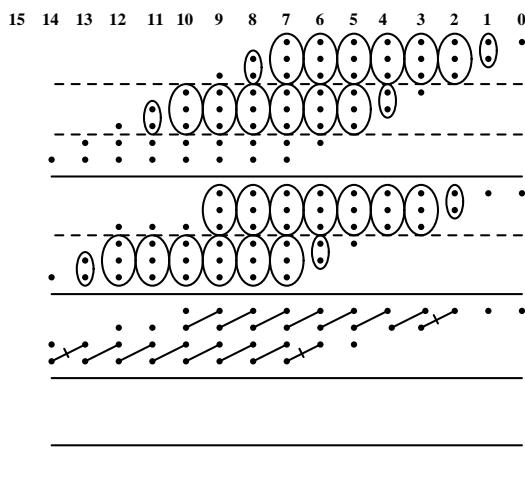
The reduction scheme as described above sometimes produces a redundant most significant bit. The result is still correct and the redundant bit will always be zero. To see this effect, let us apply the above scheme to an 8x8 multiplier.



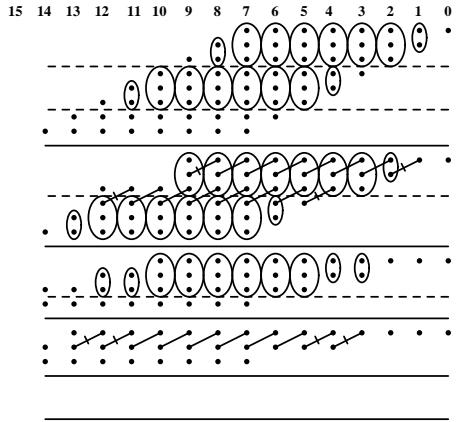
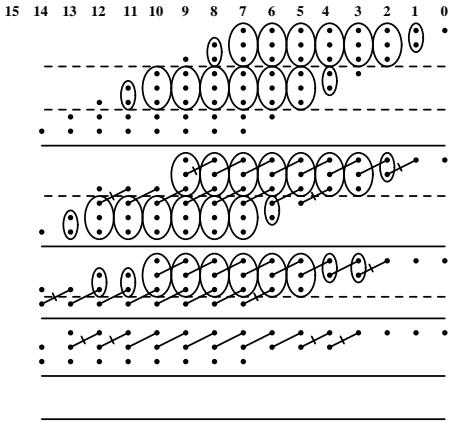
The eight rows of partial products are divided into groups 3, 3 and 2. The last two rows are just passed through to the next stage.



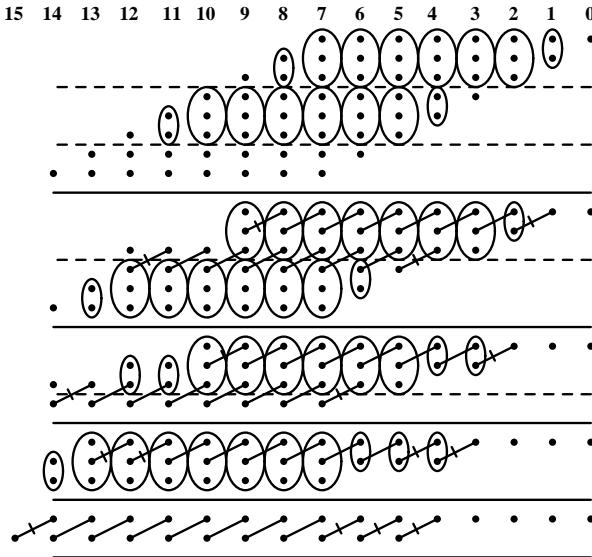
Taking each group of 3 rows, we place a full adder wherever we find three dots and a half adder where there are two dots. Single dots are passed through. This reduces the output rows to 6.



The six rows are divided into two groups of three each and we place full and half adders as shown in the figure on the left above. This reduces the number of rows to four.



The four rows output from the previous stage are grouped as 3 and 1 as shown in the figure on the left. The fourth row is just passed through, while the group of 3 rows is reduced by full and half adders. we are now left with 3 rows of output wires which need to be reduced to two by the next stage.



The 3 rows of input wires to the last stage form a single group and are reduced by full and half adders to 2 rows as shown in the figure above.

As one can see, there are two bits at bit-14, which can produce a carry during the final addition. When this carry is added to the bit at bit-15, it could produce another carry which will go to bit-16. This would be an extra bit. (Bits 0 to 16 will be 17 bits). In practice, 17 bits will not be produced, as multiplying 8 bit operands should generate at the most a 16 bit result.

3.7.4 Avoiding the Redundant MSB in Wallace Multipliers

One can avoid the redundant bit by modifying the reduction scheme:

We treat all wires in a column as equivalent. (No groups of 3 rows).

As long as there are 3 or more wires, make bunches of 3 wires and send each to a full adder. Now we can be left with 0, 1 or 2 wires. There is nothing to do for 0 wires left. If one wire is left, it is passed through to next layer.

When two wires are left, we have a more complex decision to take. For this, we first need to define the capacity of a reduction layer.

Wire capacity of a reduction layer

We define the capacity of a layer as the maximum number of wires it can accommodate. How can we determine it?

We know that the final reduction layer should have no more than 2 wires. Now we can work **backwards** from the final layer to the first. Let d_j represent the maximum number of wires for any weight in layer j , where $j = 1$ for the final adder. (Thus $d_1 = 2$). The maximum number of wires which can be handled in layer $j+1$ (from the end) is the integral part of $(3/2)d_j$ with $j = 1$ for the final adder. Thus $d_1 = 2$. We go up in j , till we reach a number which is just greater than or equal to the largest bunch of wires in any weight. The number of reduction layers required is this $j_{final} - 1$. Capacities of layers starting from last layer and moving towards the top are 2, 3, 4, 6, 9, 13, 19

Reduction of two wires

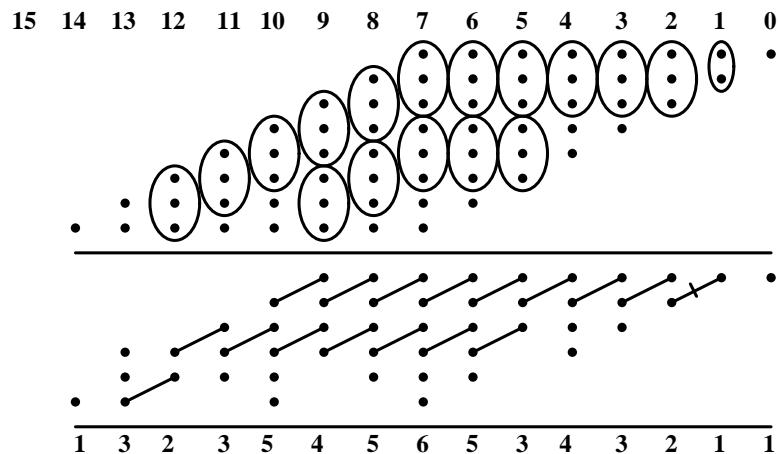
Now we can define the policy for reduction of 2 left over wires after deploying the maximum number of full adders.

- If all columns at the right have a single wire, we reduce the two wires using a half adder. (This helps in reducing the width of final adder).
- If there is a column to the right with more than one wire, we pass through the two wires to the next layer if it can accommodate these. (That is, the total number of wires do not exceed the capacity of that layer).
- If passing through the two wires would exceed the capacity of next layer, we reduce these with a half adder.

3.7.5 Wallace 8x8 Reduction without redundant MSB

We start with a maximum of 8 wires in any column. Capacity of the next layer is 6.

Bit No.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
In Wires	0	1	2	3	4	5	6	7	8	7	6	5	4	3	2	1
FA	0	0	0	1	1	1	2	2	2	2	2	1	1	1	0	0
Remaining	0	1	2	0	1	2	0	1	2	1	0	2	1	0	2	1
HA	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
PT	0	1	2	0	1	2	0	1	2	1	0	2	1	0	0	1
Sums	0	0	0	1	1	1	2	2	2	2	2	1	1	1	1	0
Carries to Higher bits	0	0	0	1	1	1	2	2	2	2	2	1	1	1	1	0
Output Wires	0	1	3	2	3	5	4	5	6	5	3	4	3	2	1	1

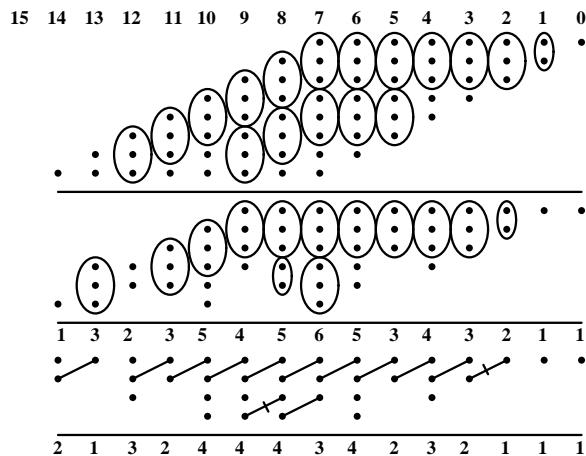


Two wires need to be handled at bits 1, 4, 7, 10 and 13. Since there is a single wire at bit 0, the two wires at bit 1 are reduced using a half adder. In all other cases, passing through the two wires will not make the number of output wires greater than 6. So we pass those through.

Second reduction layer

At this layer, maximum wires in any column is 6. Capacity of the next layer is 4.

Bit No.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
In Wires	0	1	3	2	3	5	4	5	6	5	3	4	3	2	1	1
FA	0	0	1	0	1	1	1	1	2	1	1	1	1	0	0	0
Remaining	0	1	0	2	0	2	1	2	0	2	0	1	0	2	1	1
HA	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0
PT	0	1	0	2	0	2	1	0	0	2	0	1	0	0	1	1
Sums	0	0	1	0	1	1	1	2	2	1	1	1	1	1	0	0
Carries to Higher bits	0	0	1	0	1	1	1	2	2	1	1	1	1	1	0	0
Output Wires	0	2	1	3	2	4	4	4	3	4	2	3	2	1	1	1

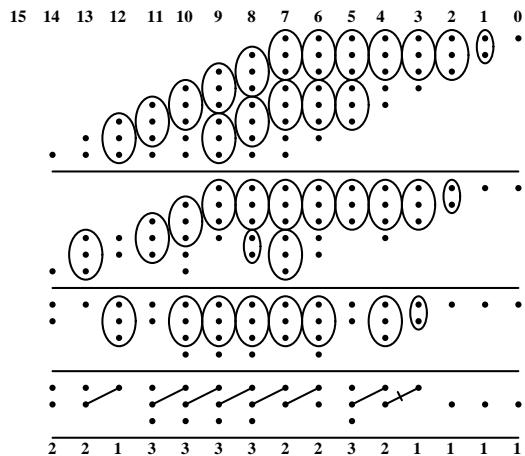


This time, we have to handle 2 wires at bits 2, 6, 8, 10 and 12. Since there is a single wire at bits 0 and 1, we reduce the two wires at bit 2 using a half adder. Two wires at bit 6 can be passed through because the number of output wires will not exceed 4. However, at bit 8, we anticipate two carry wires from bit 7 and a sum wire from the bunch of 3 wires at bit 8 itself. Passing through the 2 wires will make the number of output wires 5, which will exceed the capacity of the next layer (4). Therefore the 2 wires left at bit 8 must be reduced using a half adder. Two wires at bits 10 and 12 can be passed through because the number of output wires do not exceed 4.

Third reduction layer

Capacity of next layer is 3.

Bit No.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
In Wires	0	2	1	3	2	4	4	4	3	4	2	3	2	1	1	1
FA	0	0	0	1	0	1	1	1	1	1	0	1	0	0	0	0
Remaining	0	2	1	0	2	1	1	1	0	1	2	0	2	1	1	1
HA	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
PT	0	2	1	0	2	1	1	1	0	1	2	0	0	1	1	1
Sums	0	0	0	1	0	1	1	1	1	1	0	1	1	0	0	0
Carries to Higher bits	0	0	0	1	0	1	1	1	1	1	0	1	1	0	0	0
Output Wires	0	2	2	1	3	3	3	3	2	2	3	2	1	1	1	1

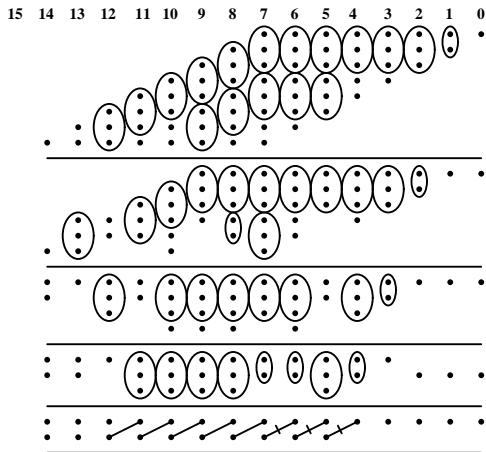


This time we have to handle two wires at bits 3, 5, 11 and 14. Since there is a single wire at bits 0, 1 and 2, we should reduce the 2 wires at bit 3 using a half adder. Passing through the 2 wires at bits 5, 11 and 14 does not exceed 3 output wires, so these are passed through.

Final reduction layer

Capacity of next layer is 2.

Bit No.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
In Wires	0	2	2	1	3	3	3	3	2	2	3	2	1	1	1	1
FA	0	0	0	0	1	1	1	1	0	0	1	0	0	0	0	0
Remaining	0	2	2	1	0	0	0	0	2	2	0	2	1	1	1	1
HA	0	0	0	0	0	0	0	0	1	1	0	1	0	0	0	0
PT	0	2	2	1	0	0	0	0	0	0	0	0	1	1	1	1
Sums	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
Carries to Higher bits	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
Output Wires	0	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1



This time there are two wires at bit positions b4, b6, b7, b13 and b14. As before, we should reduce b4 wires using a half adder because there are single wires at all less significant positions. Passing through the 2 wires at bits 6 and 7 would produce 3 output wires, exceeding the target of 2 wires. Therefore here too we place half adders. However, there is no incoming carry at bits b13 and b14 and so we can pass the 2 wires through.

Thus we have reached two wires without generating a wire at b15. There are two wires at b14 and if these produce a carry, it will go to b15. However, there is no redundant b16 now.

We have reduced the number of wires at all bit positions to ≤ 2 without generating a bit at b15.

There are two wires at b14 and if these produce a carry, it will go to b15 and there is no redundant b16.

3.7.6 Dadda Multipliers

Dadda multipliers are very similar to Wallace multipliers and use the same 3 stages:

1. Generate all bits of the partial products in parallel.
2. Collect all partial products bits with the same place value in bunches of wires and reduce these in several layers of adders till each weight has no more than two wires.
3. For all bit positions which have two wires, take one wire at corresponding place values to form one number, and the other wire to form another number.

Add these two numbers using a fast adder of appropriate size.

The difference is in the reduction stage.

Wallace multipliers reduce as soon as possible, while Dadda multipliers reduce as late as possible. Dadda multipliers plan on reducing the final number of wires for any weight to 2 with as few and as small adders as possible. We determine the number of layers required first, beginning from the **last** layer, where no more than 2 wires should be left. The number of layers in Dadda multipliers is the same as in Wallace multipliers.

We work back from the final adder to earlier layers till we find that we can manage all wires generated by the partial product generator.

We know that the final adder can take no more than 2 wires for each weight.

Let d_j represent the maximum number of wires for any weight in layer j , where $j = 1$ for the final adder. (Thus $d_1 = 2$).

The maximum number of wires which can be handled in layer $j+1$ (from the end) is the integral part of $3/2d_j$.

We go up in j , till we reach a number which is just greater than or equal to the largest bunch of wires in any weight. The number of reduction layers required is this final $j - 1$.

Wire Reduction in Dadda Multipliers

At each layer we know the maximum number of wires which should be left for the next layer.

For each weight, we place the least number of smallest adders, such that the wires going out to the next layer do not exceed the maximum number of wires it can handle.

At each weight, we must consider all the incoming wires, as well as the carry wires which will be transferred from the less significant weights in the next layer.

That is why we must begin with the lowest weight and go towards higher weights in each layer.

Dadda Multiplier: Example

Take the example of 4-bit by 4-bit multiplication multiplying $a_3a_2a_1a_0$ by $b_3b_2b_1b_0$. As before, partial products are generated in parallel and we have the following wires:

Weight	Terms	Wires
1	a_0b_0	1
2	a_0b_1, a_1b_0	2
4	a_0b_2, a_1b_1, a_2b_0	3
8	$a_0b_3, a_1b_2, a_2b_1, a_3b_0$	4
16	a_1b_3, a_2b_2, a_3b_1	3
32	a_2b_3, a_3b_2	2
64	a_3b_3	1

Number of Reduction Layers

Maximum no. of wires for any weight in this example is 4. $d_1 = 2$, $d_2 = 3$, $d_3 = 4$. So we need 2 layers of reduction.

The first reduction layer should reduce the number of wires at any weight to a maximum of 3. The second layer will then reduce these to a maximum of 2 wires.

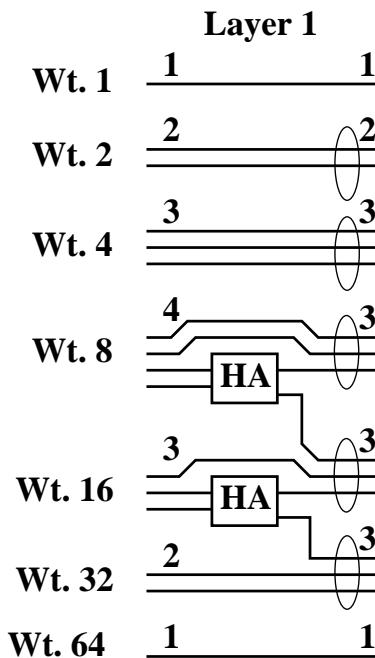
At each reduction layer, we scan from less significant weights to more significant ones, keeping track of additional carry wires which will be transferred *at the output* from lower weights to higher ones.

First Reduction Layer

Weight	Wires
1	1
2	2
4	3
8	4
16	3
32	2
64	1

- Weights 1, 2 and 4 have 3 or less wires. These are passed through.
- Weight 8 has 4 wires. No carry is anticipated from lower weights. A half adder is used to reduce the output wires to 3. (Half Adder Sum + 2 wires passed through).
- Weight 16 has 3 wires, but we anticipate a carry from the adder at weight 8. So we should reduce by 1 to keep the total **output** wires to 3. So this column is also reduced using a half adder.

After First Reduction



- Wt.1 has the single wire which was fed through.
- Wt.2 has 2 fed through wires.

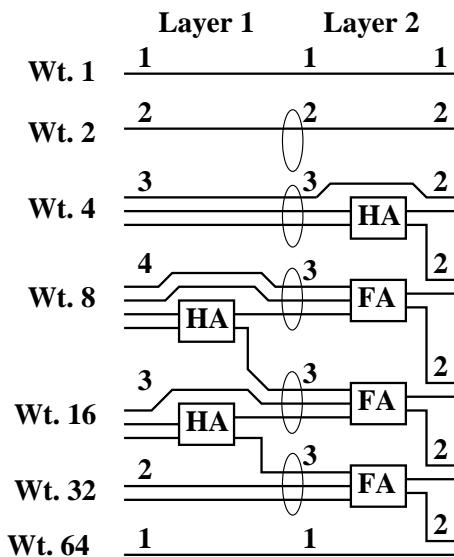
- Wt.4 has 3 wires: all passed through.
- Wt.8 has 3 wires: sum of the half adder at wt.4, and 2 passed through.
- Wt.16 has 3 wires: carry of wt. 8, sum of half adder at 16 and 1 passed through.
- Wt.32 has 3 wires: carry of wt. 16 and 2 passed through.
- Wt.64 has 1 fed through wire.

Second Reduction

In the second layer, we should leave no more than 2 wires at any weight, as this is the last stage.

- As before, we anticipate the number of carry wires transferred from the lower weight when planning reduction using half or full adders.
- In Dadda multipliers, we use minimum hardware during reduction. So the smallest adder which will reduce the output wires to 2 will be used.
- At the lowest weights, if the number of wires is less than or equal to 2, we just pass these through.
- So the single wire at Wt. 1, and the 2 wires at Wt. 2 are just fed through.

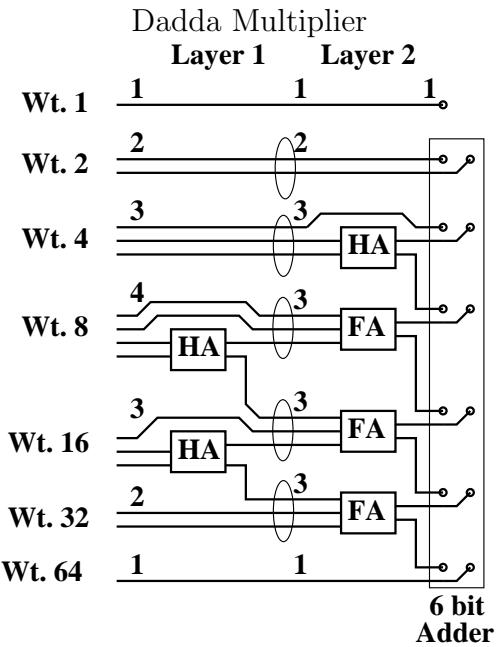
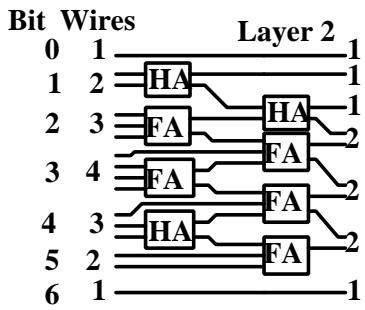
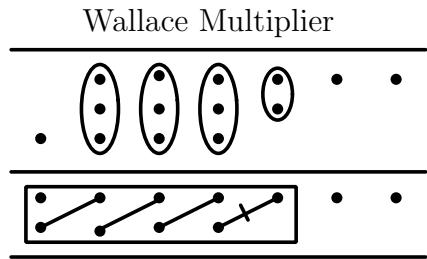
4X4 Dadda Multiplier: Second Reduction



- 3 wires at Wt. 4 are reduced to 2 by a half adder: No carry in expected.

- Wt. 8 has 3 input wires. Carry will arrive from Wt. 4. Reduced using a Full adder.
- Wt. 16 has 3 input wires. Carry will arrive from Wt. 8. Reduced using a full adder.
- Wt. 32 has 3 input wires. Carry will arrive from Wt. 16. Reduced using a full adder.
- Wt. 64 has 1 input wire. Carry will arrive from Wt. 32, making it 2 output wires, which will be fed through.

4X4 Multipliers: Final Addition



Notice that we have used only 3 Full Adders and 3 Half Adders during reduction, whereas Wallace multiplier requires 5 Full Adders and 3 Half Adders.

However, we require a 6-bit final adder for Dadda multiplier, whereas Wallace multiplier needs only a 4 bit final adder.

3.8 Multiply and Accumulate circuits

A common operation required in data processing is evaluation of quantities of the type $\sum c_i X_i$. This requires that the value of the product at each term should be added to the current value of the sum to get its new value. For implementing such calculations, it would be useful if we can have a circuit in hardware which will efficiently multiply two operands as well as add a third. Notice that this third operand will be of the size of the product. Such circuits are called Multiply and Accumulate circuits.

Multiply and Accumulate circuits are easy to implement because during multiplication, we are anyway adding multiple bits in a column. The accumulator just provides an additional bit in every column and this can be easily added along with the partial product bits in a carry-save fashion using any of the wire reduction techniques described above.

Consider for example a Multiply and Accumulate circuit which multiplies two 8 bit operands and adds a 16 bit operand to it. Let us apply the Wallace wire reduction technique to compute $A \times B + C$, where A and B are 8 bit operands, while C is a 16 bit operand. The product computation will generate wires at bit positions 0 through 14 in the usual trapezoidal shape. Thus the number of wires from partial sums of the multiplier will be:

Bit pos.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Partial Sum Wires	0	1	2	3	4	5	6	7	8	7	6	5	4	3	2	1

The operand to be added just provides one additional wire at each bit position from 0 to 15. Thus the number of wires at each bit position becomes

Bit pos.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Wire count	1	2	3	4	5	6	7	8	9	8	7	6	5	4	3	2

We can now proceed to reduce these wires as a Wallace tree till we are left with no more than 2 wires at each position. Finally, we shall add these two groups using a fast traditional adder. The tables below show the wire reduction at each stage:

Stage 1: Max. wires:9, capacity of next stage = 6

Bit pos.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Input Wires	1	2	3	4	5	6	7	8	9	8	7	6	5	4	3	2
Full Adders	0	0	1	1	1	2	2	2	3	2	2	2	1	1	1	0
Half Adders	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1
Pass Through	1	2	0	1	2	0	1	0	0	2	1	0	2	1	0	0
Output Wires	1	3	2	3	5	4	6	6	5	6	5	3	4	3	2	1

Stage 2: capacity of next stage = 4

Bit pos.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Input Wires	1	3	2	3	5	4	6	6	5	6	5	3	4	3	2	1
Full Adders	0	1	0	1	1	1	2	2	1	2	1	1	1	1	0	0
Half Adders	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0
Pass Through	1	0	2	0	2	1	0	0	0	0	2	0	1	0	0	1
Output Wires	2	1	3	2	4	4	4	4	4	3	4	2	3	2	1	1

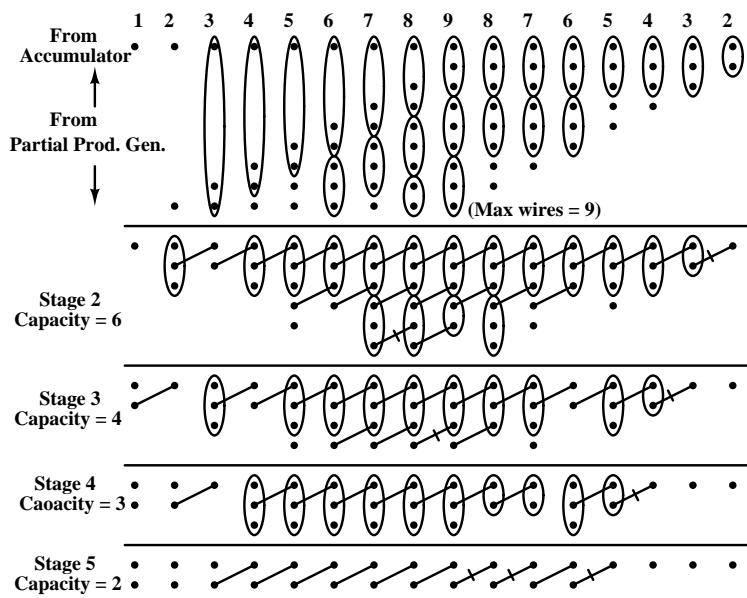
Stage 3: capacity of next stage = 3

Bit pos.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Input Wires	2	1	3	2	4	4	4	4	3	4	2	3	2	1	1	1
Full Adders	0	0	1	0	1	1	1	1	1	1	0	1	0	0	0	0
Half Adders	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
Pass Through	2	1	0	2	1	1	1	1	1	0	1	2	0	0	1	1
Output Wires	2	2	1	3	3	3	3	3	3	2	2	3	2	1	1	1

Stage 4: capacity of next stage = 2

Bit pos.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Input Wires	2	2	1	3	3	3	3	3	3	2	2	3	2	1	1	1
Full Adders	0	0	0	1	1	1	1	1	1	0	0	1	0	0	0	0
Half Adders	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	0
Pass Through	2	2	1	0	0	0	0	0	0	0	0	0	0	1	1	1
Output Wires	2	2	2	2	2	2	2	2	2	2	2	2	2	1	1	1

The dot diagram for this scheme is shown below:



Finally, we need a 12 bit fast adder to get the final result.

The complexity of this circuit is not much more than a plain 8x8 multiplier and the time taken to produce the results is much less than the time taken to multiply first and then to add. This is because the latter requires two additions in which the carry ripples while the Multiply and Accumulate circuit requires only one such addition.

3.9 Serial Multipliers

Often, we need multipliers which have very low complexity or very low power consumption and speed is not very important. Serial multipliers are a good option in such cases.

Low complexity multipliers can be bit serial or row serial. Bit serial multipliers require $m \times n$ clocks for completing an $m \times n$ multiplication. Row serial multipliers require only n steps, but we require m full adders rather than just one.

3.9.1 Bit Serial Multipliers

For serial multiplication, partial product bits need not be generated in parallel. These can be generated as and when these are required. Each bit of the multiplier needs to be ANDed

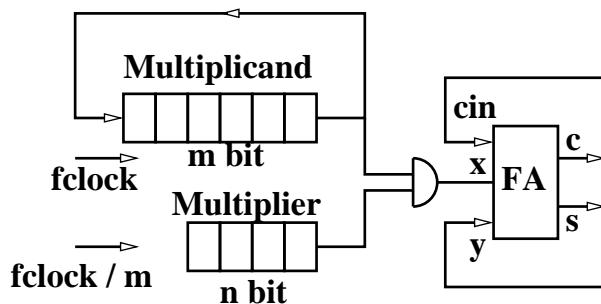
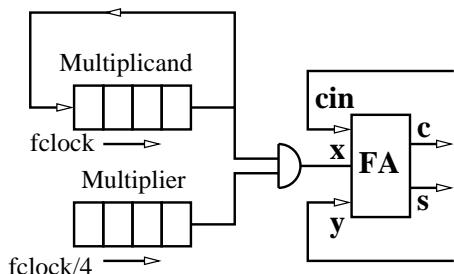


Figure 3.10: Partial product generation for bit-serial multiplication

with each bit of the multiplicand. This requires that all multiplicand bits be presented one after the other every time a new bit from the multiplier is taken up. This can be managed by using a re-circulating shift register for the multiplicand, which is clocked at a rate which is m times faster than the clock supplied to the multiplier shift register. The inputs y and Cin to the full adder have to be appropriately selected and timed to generate the correct product.

Consider a 4×4 bit serial multiplier.

The x input to the Full Adder appears in the following order:



ck	x	ck	x	ck	x	ck	x
0	a0b0	4	a0b1	8	a0b2	12	a0b3
1	a1b0	5	a1b1	9	a1b2	13	a1b3
2	a2b0	6	a2b1	10	a2b2	14	a2b3
3	a3b0	7	a3b1	11	a3b2	15	a3b3

We need additions as follows:

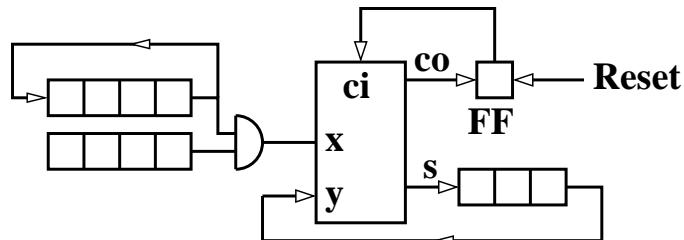
$$\begin{array}{r}
 & a_3 & a_2 & a_1 & a_0 \\
 \times & b_3 & b_2 & b_1 & b_0 \\
 \hline
 & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 \\
 a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 \\
 a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3
 \end{array}$$

Let us put the arrival time of terms in parentheses next to each term.

$$\begin{array}{r}
 & a_3 & a_2 & a_1 & a_0 \\
 \times & b_3 & b_2 & b_1 & b_0 \\
 \hline
 & a_3b_0(3) & a_2b_0(2) & a_1b_0(1) & a_0b_0(0) \\
 a_3b_1(7) & a_2b_1(6) & a_1b_1(5) & a_0b_1(4) \\
 a_3b_2(11) & a_2b_2(10) & a_1b_2(9) & a_0b_2(8) \\
 a_3b_3(15) & a_2b_3(14) & a_1b_3(13) & a_0b_3(12)
 \end{array}$$

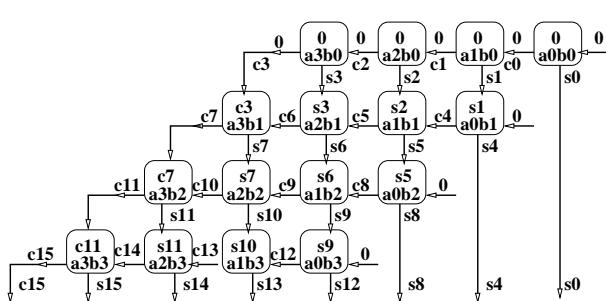
It is clear that for all additions, the earlier terms have to wait for 3 clock cycles before the later terms arrive.

We can manage this by putting a 3 bit shift register at the sum output and presenting the delayed output at the 'y' input of the full adder. The carry output can be added immediately in the next clock, since it should go to the next column to its left. A 3 clock delay for sum and a 1 clock delay for carry leads to the following circuit.



Unfortunately, it does not work as shown!

We need to take care of a few exceptions. Let us look at all the exceptions in detail. In the figure below, each box contains the y and x inputs presented to the adder. The sum and carry terms are indexed by the clock cycle in which these were generated. For example, c_7 is the carry generated in the 7th clock cycle. Notice that in the first four cycles, 0 is being added to partial products and therefore the carry is always 0.



- At clocks 0, 4, 8 and 12, carry input should be forced to 0.
- At clocks 7, 11 and 15, the adder y input should receive carry terms (c_3 , c_7 and c_{11}) instead of sum terms (s_4 , s_8 and s_{12}).
- The sum terms s_4 , s_8 and s_{12} should be taken out as result bits and not inserted in the 3 bit delay shift register.

We can take care of these exceptions by inserting the carry FF output (which is a 1 clock delayed version of cout) in the 3-bit shift register instead of the sum terms. Thus C_3 (which is always 0), C_7 and C_{11} will emerge from the shift register at clocks 7, 11 and 15 respectively and will be added to the correct partial product bits. The corresponding sum terms should be taken out as result bits.

3.9.2 Bit Serial Multiplier: Implementation

With exception handling at the end of rows, the serial multiplier will work. Notice the changes

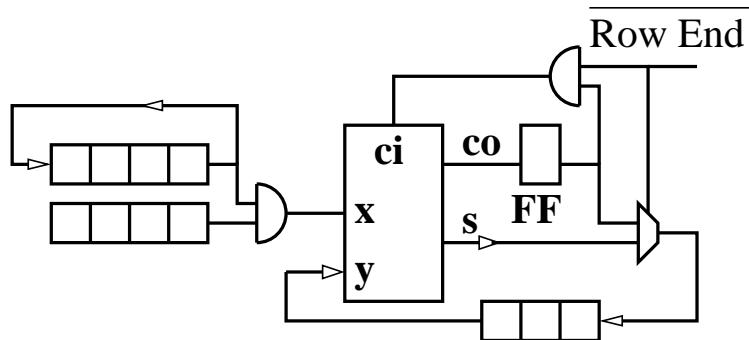


Figure 3.11: 4x4 bit serial multiplier

from the earlier suggested circuit:

- Carry input is forced to 0 at row ends.
- The mux normally inserts the sum into the shift register. However, at row ends, it inserts the delayed carry output.
- The sum terms at row ends are taken out as the low bits of the product.

- One can add another shift register at the output to collect these.
- The 2 more significant bits of the shift register and the last sum and carry provide the high bits of the product at the end.

3.9.3 Row Serial multipliers

We need not reduce the complexity all the way down to a single adder for serial multipliers. We could have a row of n adders in parallel performing n serial additions. We can use n full adders arranged in n columns. The carry output of previous addition is retained at the same column. The sum from the previous addition *in the left column* is brought to this column by a shift operation. This sum has the same weight as the carry generated during the previous clock in this column. These two are added to the partial product bit for this column.

Taking the example of 4×4 multiplication, we are trying to perform the following operations:

$$\begin{array}{r}
 \text{a3 a2 a1 a0} \\
 \times \text{b3 b2 b1 b0} \\
 \hline
 \text{a3b0 a2b0 a1b0 a0b0} \\
 \text{a3b1 a2b1 a1b1 a0b1} \\
 \text{a3b2 a2b2 a1b2 a0b2} \\
 \text{a3b3 a2b3 a1b3 a0b3}
 \end{array}$$

The addition process using 4 adders is represented in Figure 3.12. Notice that the same ‘a’ term is used for generating partial products for a given column. The ‘b’ term has to be shifted right every time to generate the right partial product bit.

Sums have to be shifted right to be added to the carry of the previous addition in the same column. 4 additional clock cycles will be required to ripple the carry in the last addition. During these, the partial product bits will be 0.

This scheme can be implemented as follows:

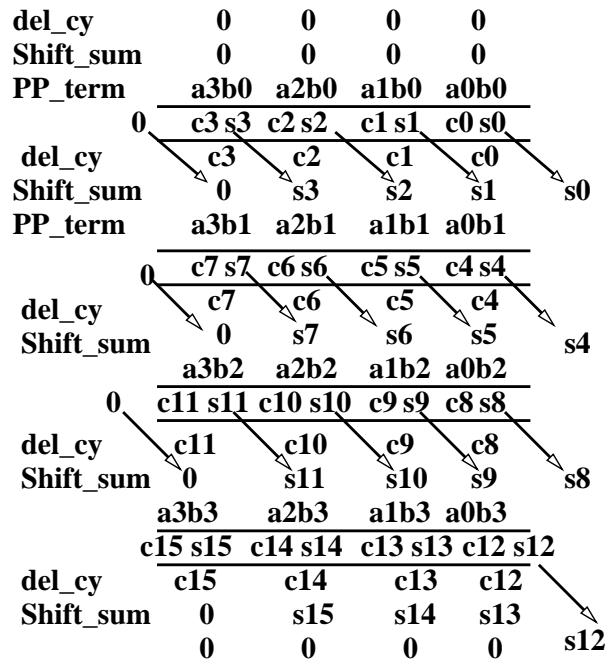


Figure 3.12: 4x4 row serial multiplication

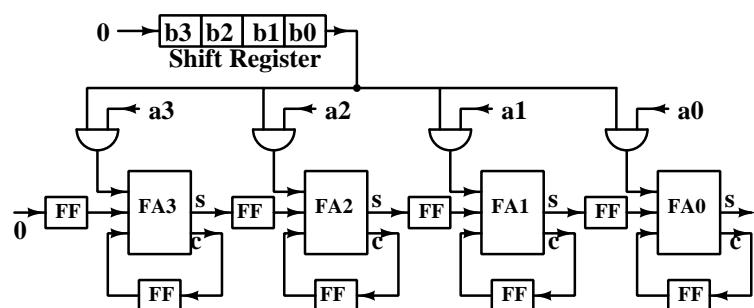


Figure 3.13: 4x4 row serial multiplier

Semiconductor Memories

- Most electronic systems require storage and retrieval of data. Systems requiring a small amount of storage can use flipflops for this purpose.
- However, Flipflops are not efficient for large storage requirements. For large storage, we need specialised arrangements of storage elements called memories.
- Depending on their characteristics, memories are classified as
 - Static Random Access Memories (SRAM),
 - Dynamic Random Access Memories (DRAM),
 - Read Only Memories (ROMs)
 - Electrically Programmable Memories (EPROMs)
 - Electrically Programmable and Erasable Memories (EEPROMs)
 - Flash Memories
- Practically all memory systems these days use semiconductor memories. (Magnetic “cores” were used earlier for storage. Some memory terminology is acquired from those times).

Semiconductor Memories

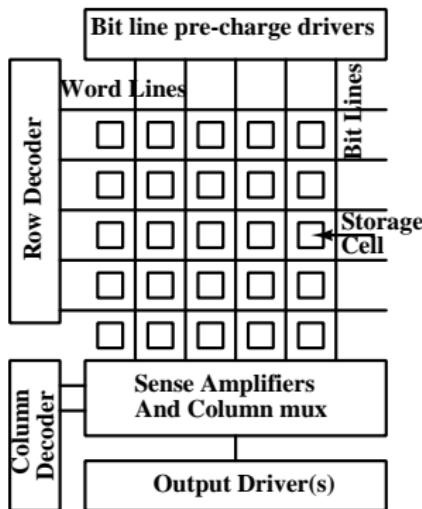
- Data stored in a memory can be accessed sequentially or in a random order.
- For Random Access, an address is provided to the memory, which then returns the stored content from that address or writes new data at this address.
- Conveying the address to the memory takes time, which may slow down memory access and affect system performance adversely.
- Many modern memory systems use a “burst mode” in which a start address is supplied, and then data is read or written from sequential addresses in a burst.
- The address is automatically adjusted in the memory by incrementing the initially supplied address with each read or write in a burst.
- While the storage mechanism varies from one type of memory to another, the internal organisation and the method for accessing the contents are very similar for all types of memories.

Semiconductor Memories

- At each unique address, the memory can store a single bit or multiple bits with a width of various sizes. Correspondingly, the memory is said to be bit oriented or byte or word oriented.
- The stored data may last only as long as power is applied to the memory. Such storage is called volatile.
- Some memories can retain their information even when power is removed. Such memories are called non-volatile.
- A memory must include circuits to decode the supplied address in order to activate a specific storage location associated with this address for read or write operation.
- A read or write operation must start only after address decoding is complete. Otherwise, a read or write may occur from a storage cell corresponding to a transient value of the changing address.
- This can result in data corruption.

Memory Organization

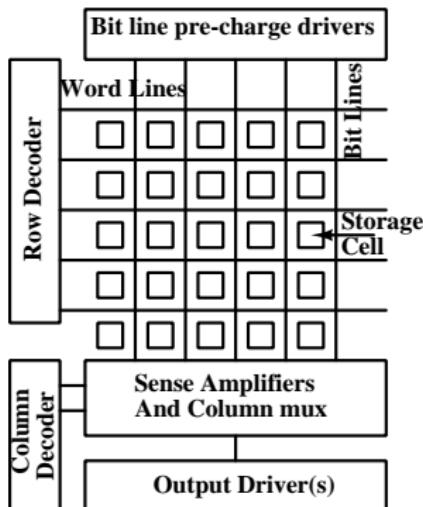
To understand the terminology used for memories, it is important to list the sub-units which are used in a typical memory. Details of their operation will vary depending on the type of memory, but the overall organisation is quite similar.



- Row/Column Address decoders provide the logic used for decoding the address in order to activate the selected memory location.
- Memory array is a two dimensional array in which the storage elements are arranged. The address decoder typically selects a row and a column to identify the memory cell being addressed.
- A sense amplifier is used in every column to amplify the small signal provided by the addressed cell and to amplify it to a rail-to-rail logic level

Memory Organization

The long bit lines in a memory would present a very heavy load for the tiny storage cells to drive. Therefore these are pre-charged to a voltage close to V_{DD} .



- An addressed storage cell, when connected to these lines will not change their voltage level if the connected node is at '1'.
- If the connected note is at '0', it will lower the voltage of the bit line by a small amount.
- The sense amplifier detects the presence or absence of this small voltage dip and amplifies it to a full scale '0' or '1'. For reliable operation, we run differential lines (bit and \bar{bit}) and connect these to Q and \bar{Q} outputs of the storage cell.
- Output drivers provide sufficient drive to take the off-chip lines quickly to a '0' or a '1'.

Memory Timing parameters

The speed of a memory subsystem is characterized by several timing parameters.

Read Access time: This is the delay between presentation of an address to a memory and the availability of data at that location.

Row and Column Address strobes Often the address is specified in two halves – known as Row Address and Column Address. Each half of the address is latched into the memory using strobe signals. These strobes are known as Row Address Strobe (RAS) and Column Address Strobe (CAS) signals.

RAS/CAS times The Read Access time is broken into two halves. RAS to CAS delay (T_{RCD}) is the time gap required between the Row and column address strobes. CAS latency (CL) is the delay between Column strobe and availability of data.

Memory Timing parameters

Memory systems pre-charge long internal lines connected to a memory array and then sense the contents of a selected location by connecting it to the pre-charged lines and

Write recovery time: This is the interval needed between a write and the pre-charge operation. Since a pre-charge operation forcibly takes all bit lines to '1'.

Memory cycle time: This is the time required to complete successive read or write operations. This determines the rate at which one can issue new read or write requests to a memory. The read and write cycle times need not be the same, but most systems specify a single cycle time, which is the longer of the two.

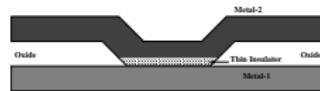
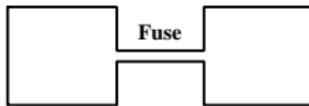
Types of storage Cells:ROM

Read only memories are those which are programmed just once with fixed data and which will be read many times but never altered.

- In Read Only Memories, we use a permanently ON or OFF switch as a memory element.
- The switch may be implemented using MOS transistors. The transistor is turned ON or OFF permanently by connecting its gate to V_{DD} or ground at the time of chip fabrication using a mask. Such memories are called Mask Programmable ROMs.
- ROMs may also be made using poly-silicon fuses or anti-fuse structures as switches. These were discussed during our discussion on Field Programmable logic. A fuse is blown or an anti-fuse shorted to program data into the One Time Programmable (OTP) ROM or PROMs.

Types of storage Cells:ROM

Fuse and anti-fuse type structures can be used to connect or disconnect devices to implement ROM celss.



- A fuse structure uses a narrow neck in an interconnect, which can be blown by passing a heavy current through it.
- A memory cell can be programmed to '0' or '1' by connecting or disconnecting a node to ground directly or through a transistor.
- An anti-fuse structure uses a structure with a thin insulator between two connecting wires.
- The insulator can be shorted by applying a high voltage across it.
- This provides the means of connecting or not connecting a node to ground programmably.

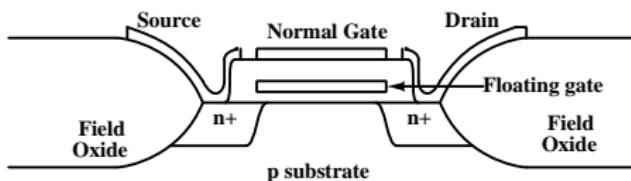
Types of storage Cells:EPROM

A variant of ROMs are Electrically Programmable ROMs or EPROMs. The data in these memories may be altered by using unusual operating conditions. The time taken to alter the contents of an EPROM is long, so the programming is carried out infrequently.

- We can implement permanently ON or OFF transistors by injecting charge into the gate region. Charge can be injected into the gate region by generating energetic carriers by avalanching and applying a field of appropriate sign between the gate and substrate.
- The injected charge into the semiconductor remains trapped in the insulating gate material, which changes the turn on voltage V_T . Thus an nMOS can be turned ON at zero gate bias by injecting holes into the gate region and making its V_T negative.
- Such memories are erased by exposure to Ultra Violet light, which releases the trapped charge from the gate oxide back into silicon.

Types of storage Cells: EEPROM

Erasure of EPROMs is through exposure to UV light, which requires a specially packaged device which will let in UV light during erasure through a quartz window. This can be avoided using electrically erasable programmable ROMs or EEPROMs.



EEPROMs use MOS structures which are programmed in a similar manner to EPROMs, but which permit erasure through application of a high field opposite to that used for injecting charge into the gate.

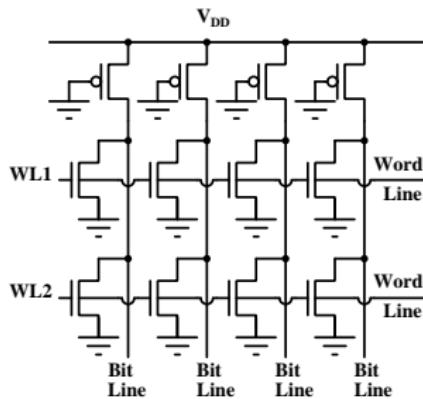
A floating gate MOS device, which uses an extra floating gate embedded in the oxide to store injected charge, can be used for this.

Types of storage Cells:EEPROM

- The process of high field injection and removal of oxide charge results in some damage to the MOS device.
- So the process of programming and erasing can be carried out for a limited number of times, after which the repeated charge injection and removal damages the device to such an extent that it cannot be programmed or erased any more.
- The number of times that a device can be programmed and erased is called the endurance of an EEPROM.

EEPROM configurations: NOR EEPROM

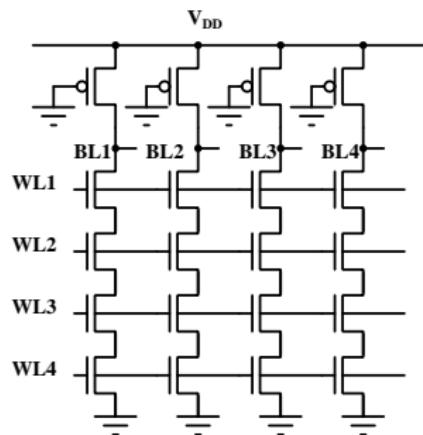
To address and read/write EEPROMs, programmable transistors can be connected in different ways.



- In practically all memory devices, there are perpendicular interconnect lines called Word lines and bit lines.
- In the NOR configuration of EEPROMs, bit lines are pulled up using permanently ON pMOS devices, as in pseudo nMOS logic.
- A programmable nMOS transistor is connected to ground at each crossing of a word line and a bit line.
- If the nMOS is ON, the bit line is pulled down to zero. If it is OFF, the bit line remains high due to the pull up transistor.

EEPROM configurations: NAND EEPROM

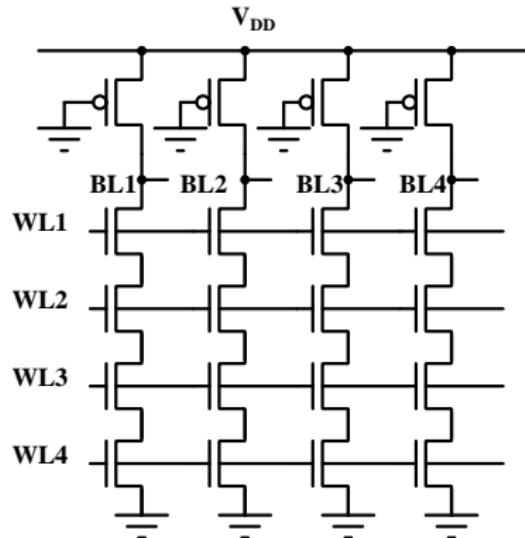
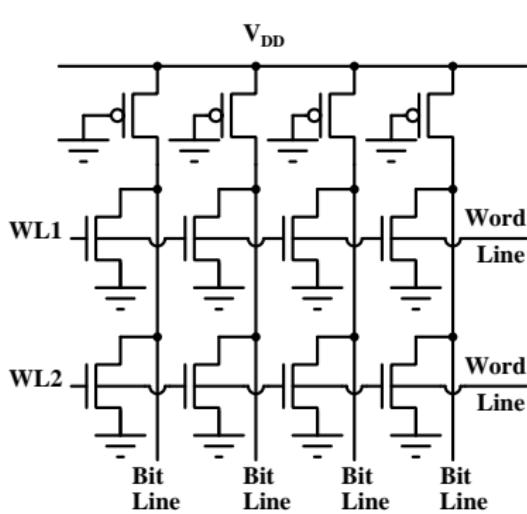
In a NAND EEPROM also, we have a pseudo nMOS like arrangement of transistors, with pMOS pull ups for each bit line.



- nMOS transistors are all connected in series in a particular column.
- All word lines except the selected one are kept at '1', while the selected world line is kept at '0'.
- If the selected transistor is ON, it pulls the bit line output lower. Otherwise it remains high.
- Unlike pseudo nMOS, we need not pull the output all the way down to logic '0'. This enables us to put a lot of nMOS transistors in series.

The selected column (bit line) is multiplexed to a sense amplifier, which brings the output to a proper level for logic '0' or '1'.

EEPROM configurations: NOR and NAND

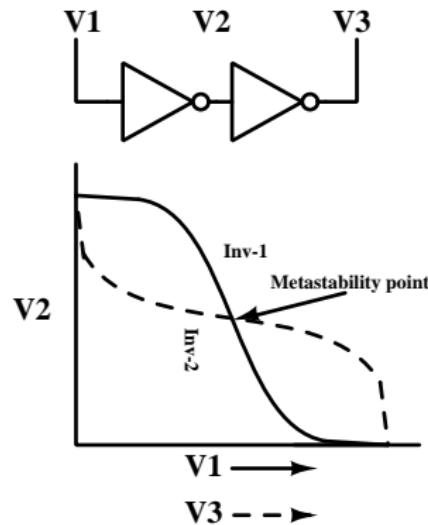
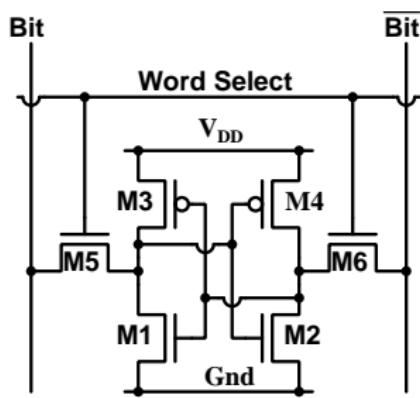


- NAND EEPROMs are much denser, because these don't need a contact window for every nMOS transistor.
- NOR is faster, because discharge is through a single transistor.
- Practically all thumb drives etc. use a NAND configuration these days.

Read/Write memories

- Read/write memories provide comparable times for read and write operation.
- These are suitable for applications where memory has to be modified frequently – such as data storage for a micro-processor based system.
- Static Read-Write memory (SRAM) uses a simple latch to store the data. The latch is implemented as two cross connected inverters and is accessed through pass transistors.
- Data in static Read Write memory will be retained as long as power is applied. This kind of storage is called volatile storage.
- Dynamic read-write memory (DRAM) stores data on a small sized capacitor. It consists of the capacitor and a single coupling transistor.
- DRAM is much denser, but needs frequent refreshing of the data, since the charge stored on the capacitor can leak away.

SRAM cell

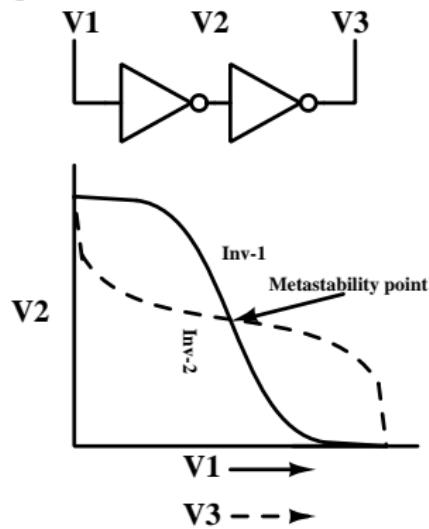


The 6 Transistor RAM cell contains cross connected inverters coupled to the bit and $\overline{\text{bit}}$ lines through pass transistors.

The pass transistors are enabled by the word select line.

SRAM cell: Butterfly diagram

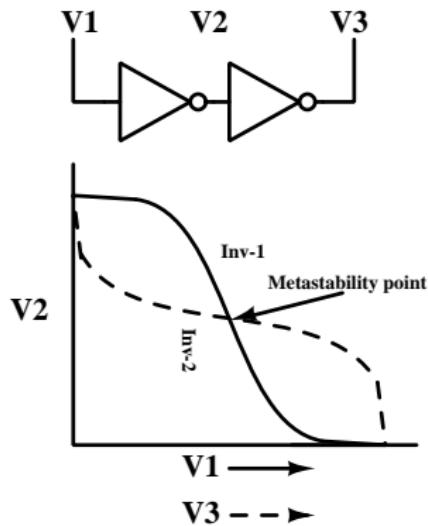
If we remove the feedback connection in the latch used in an SRAM cell, we get the two inverters as shown below.



- The solid line curve is the transfer characteristic of the first inverter (V_2 vs V_1).
- The dashed line curve is the transfer characteristic of the second inverter, with its input voltage V_2 along the Y axis and its output voltage V_3 along the X axis.
- This curve is known as a Butterfly diagram due to its shape.

In the latch, V_3 is shorted to V_1 . So the static characteristics of the latch are satisfied where the solid line and the dashed line curves cross ($V_1=V_3$).

SRAM cell: Stable and Metastable points

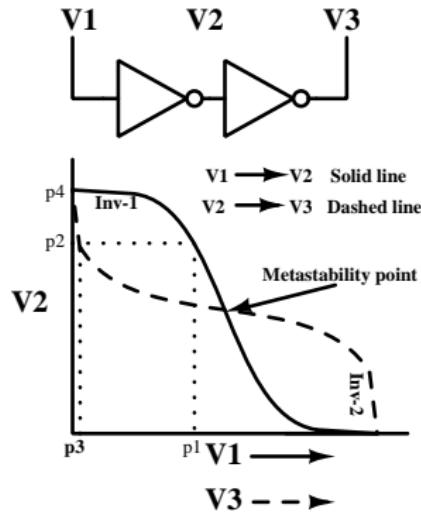


- The solid and dashed line curves cross at 3 points.
- At the left most point, $V_1 = 0$, so $V_2 = V_{DD}$ by the solid line curve.
- For $V_2 = V_{DD}$, $V_3 = 0$. by dashed line curve. So $V_1 = V_3$.
- At the right most point, $V_1 = V_{DD}$, so $V_2 = 0$ (solid line curve). For $V_2 = 0$, $V_3 = V_{DD}$ (dashed line curve). So $V_1 = V_3$ again.

$V_1 = V_3$ at the middle point as well. However, this is an unstable point as the least bit of perturbation from this point leads to one of the two stable points described above.

SRAM cell: Stable points

Suppose the latch is stable at the left crossing point and we perturb the voltage at V1.

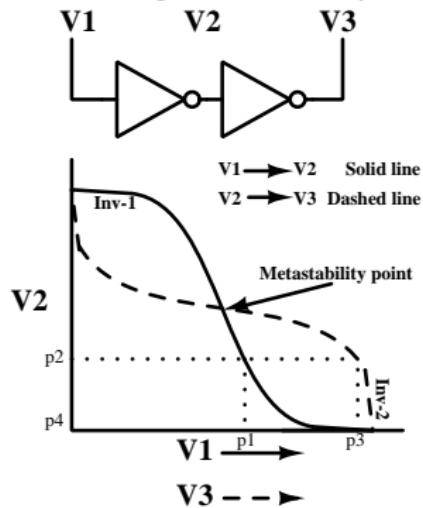


- If we perturb V1 to the point p1, the output of Inv1 will be p2 (solid line transfer curve for Inv1).
- With its input at p2, the output of Inv2 will be p3 (dashed line transfer curve for Inv2).
- But the output of Inv2 is shorted to input of Inv1 in the latch. With its input at p3, the output of Inv1 will be p4 (solid line transfer curve for Inv1).

In fact, perturbing V1 to any point to the left of the middle crossing will quickly recover to the left crossing point with $V1 = 0$ and $V2 = V_{DD}$.

SRAM cell: Stable points

What happens if the latch is stable at the right crossing point and we perturb the voltage from this point?



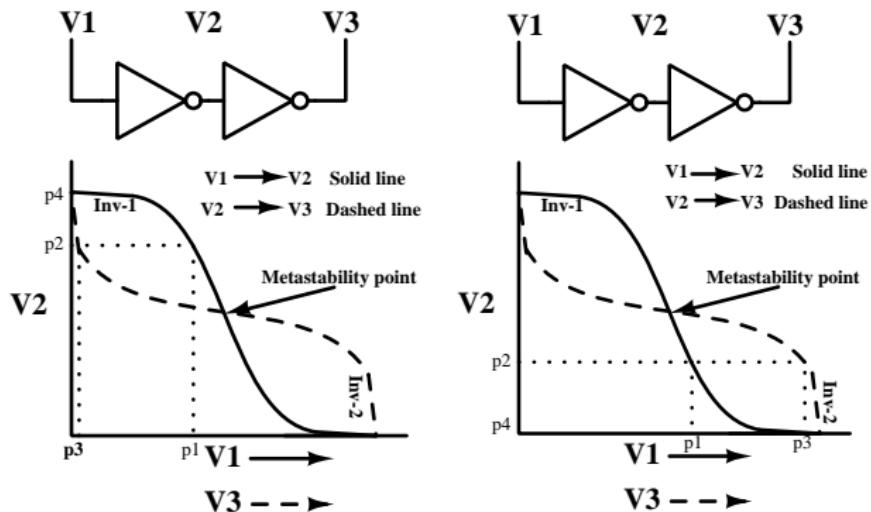
- If we perturb V1 to the point p1, the output of Inv1 will be p2 (solid line transfer curve for Inv1).
- With its input at p2, the output of Inv2 will be p3 (dashed line transfer curve for Inv2).
- But the output of Inv2 is shorted to input of Inv1 in the latch. With its input at p3, the output of Inv1 will be p4 (solid line transfer curve for Inv1).

In fact, perturbing V1 to any point to the right of the middle crossing will quickly recover to the right crossing point with $V1 = V_{DD}$ and $V2 = 0$.

SRAM cell: metastable point

What about the middle crossing point?

Network equations are satisfied here – The output voltage of Inv1, when applied to Inv2 gives back the same voltage.

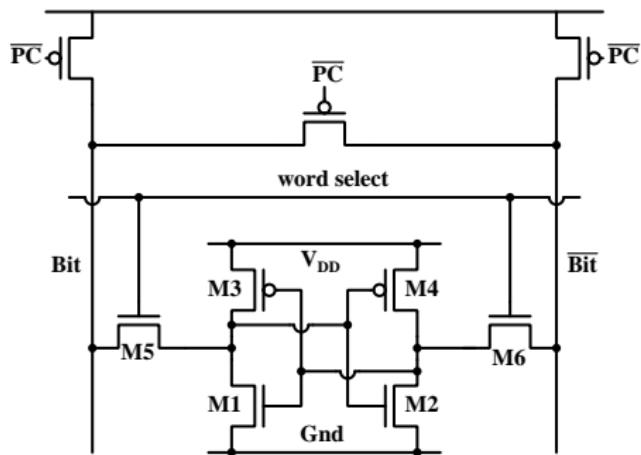


However, perturbing even by a small amount to left will make the system go to the left crossing point and even by a small amount to the right will make the system go to the right crossing point.

So this point is not stable and is called the metastable point.

SRAM cell: Read cycle

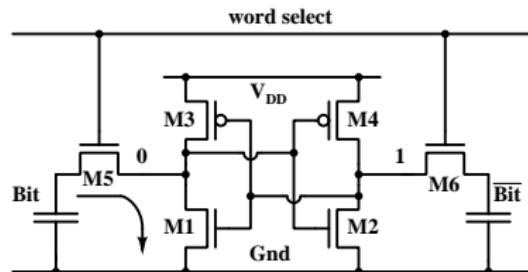
In order to read the memory, the address is placed by the processor on the address lines. The row and column addresses are decoded by the memory.



- The row decoder makes the Word Select line of the addressed row 'high'.
- On receiving the \overline{Rd} command, the memory generates a pre-charge pulse (\overline{PC}) to precharge the Bit and $\overline{\text{Bit}}$ lines to 'high'.
- The column address decoder connects the Bit and $\overline{\text{Bit}}$ lines of the selected column to the sense amplifier.

Depending on the data stored in the cell, one of its outputs of the latch will be 'high' while the other will be 'low'.

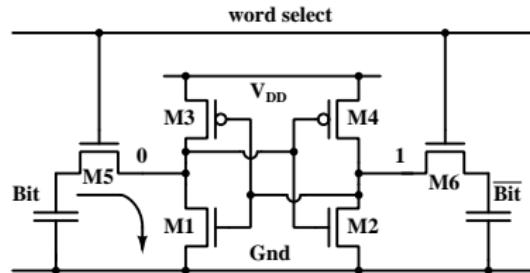
SRAM cell: Read cycle



- One of the outputs of the latch is high, while the other is low.
- Bit and $\overline{\text{Bit}}$ lines are shown here as capacitors.

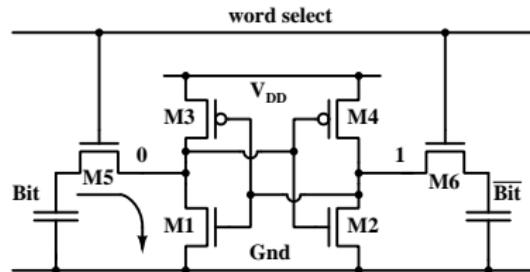
- The high node is unchanged, but the low node starts discharging the Bit or the $\overline{\text{Bit}}$ line (whichever is connected to it)
- This creates a voltage difference between the Bit and $\overline{\text{Bit}}$ lines.
- The sense amplifier amplifies the voltage difference and produces a digital '0' or '1' at its output.
- This digital data is then buffered to the output pad by a tristateable driver, which is activated only during read cycles.

SRAM cell: Read upset



- At the end of the cycle, the word select line returns to ‘low’, disconnecting the latch from Bit and Bit̄ lines.
- The low node, which would have been pulled a little high due to connection with the pre-charged Bit or the Bit̄ line is restored back to 0V by the feedback action as discussed earlier.
- The geometry of the access transistors has to be carefully determined to ensure that the low node is not pulled to too high a voltage when it is connected to the pre-charged Bit or Bit̄ line.
- If the voltage of the low node goes too high when it is connected to the pre-charged Bit or Bit̄ line, it may zap to a ‘1’ value rather than going back to ‘0’.
- This is called a read upset.

SRAM cell: Read upset



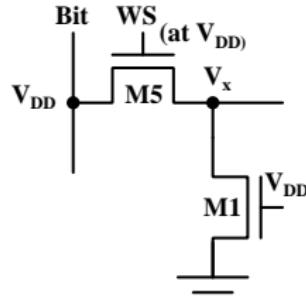
- consider the memory cell with a '0' stored in it. M3 and M2 are OFF.
- We find the condition that the voltage at the drain of M1 remains below V_{Tn} . Then M2 will remain OFF during the entire read cycle.

Since M2 remains OFF, The voltage at the gate of M1 will remain at V_{DD} for the entire cycle.

Since the drain and gate voltages of M5 are equal (and at V_{DD}), it is in saturation.

The gate of M1 is at V_{DD} while its drain is below V_{Tn} , so M2 remains in linear regime.

SRAM cell: Read upset



Equating currents through M5 and M1:

$$\frac{K_{n5}}{2} (V_{DD} - V_x - V_{Tn})^2 = K_{n1} \left((V_{DD} - V_{Tn})V_x - \frac{1}{2} V_x^2 \right)$$

We define $\beta \equiv K_{n1}/K_{n5}$. This gives:

$$(V_{DD} - V_x - V_{Tn})^2 = 2\beta V_x (V_{DD} - V_{Tn} - \frac{1}{2} V_x)$$

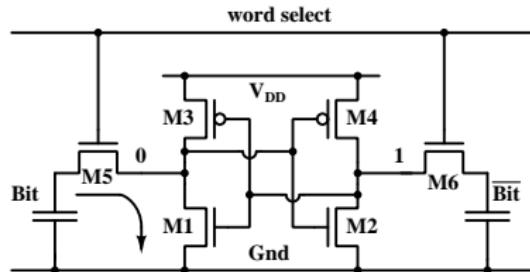
In the limiting case, $V_x = V_{Tn}$. Then:

$$(V_{DD} - 2V_{Tn})^2 = 2\beta V_{Tn} (V_{DD} - \frac{3}{2} V_{Tn}) = \beta V_{Tn} (2V_{DD} - 3V_{Tn})$$

Hence,

$$\beta = \frac{(V_{DD} - 2V_{Tn})^2}{V_{Tn}(2V_{DD} - 3V_{Tn})}$$

SRAM cell: Read upset



$$\beta = \frac{(V_{DD} - 2V_{Tn})^2}{V_{Tn}(2V_{DD} - 3V_{Tn})}$$

If we take $V_{Tn} = V_{DD}/5$, we get

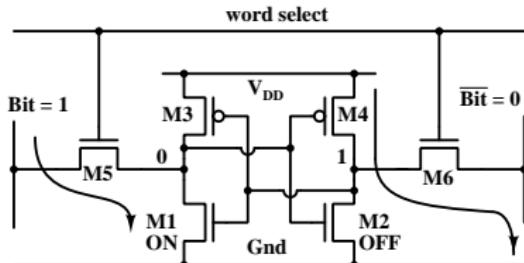
$$\beta = \frac{(5V_{Tn} - 2V_{Tn})^2}{V_{Tn}(10V_{Tn} - 3V_{Tn})} = \frac{9V_{Tn}^2}{7} V_{Tn}^2 = 9/7 = 1.2857$$

- Thus the value of β should be ≥ 1.3 to keep V_x below V_{Tn} .
- Notice that the geometry ratio of M5 and M1, as well as the maximum voltage rise at V_x is fixed by this consideration.
- This must be kept in mind while discussing the write operation.

SRAM cell: Write cycle

During the write cycle,

- The address is placed by the processor on the address lines. and the row and column address are decoded by the memory.
- When the word line goes HIGH, the access transistors are turned on for **all** cells in this row.
- When \overline{Wr} is asserted, the Bit and $\overline{\text{Bit}}$ lines are driven to Data and $\overline{\text{Data}}$ respectively **in the selected column only**.

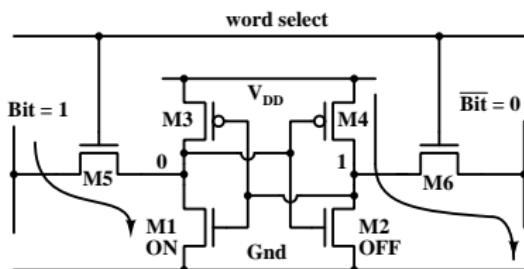


Thus, in the selected column of the selected row, the cell goes through a write cycle.

For all the remaining columns in the selected row, cells go through a read cycle.

SRAM cell: Write cycle

When the word line goes HIGH, the access transistors are turned on for all cells in this row.

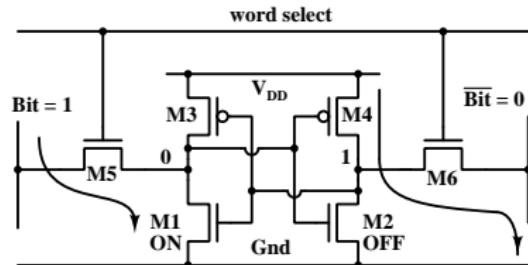


- In the selected column, the complementary values of Data and Data are copied to the latch through the pass transistors.
- Once the voltage at the node connected to the pass transistors passes the meta-stability point, the positive feed back ensures that it will go all the way to '0' or '1'.

After the write process is complete, the Word line is brought low again.

SRAM cell: Write cycle

- Considerations of Read upset fix the ratio between the pass transistor and pull down transistor width.
- The voltage rise at the node which is at '0' is limited to $\approx V_{Tn}$, with the bit line at '1'.
- Then how do we write a '1' to the cell – which requires the node to be taken above $\approx V_{DD}/2$?

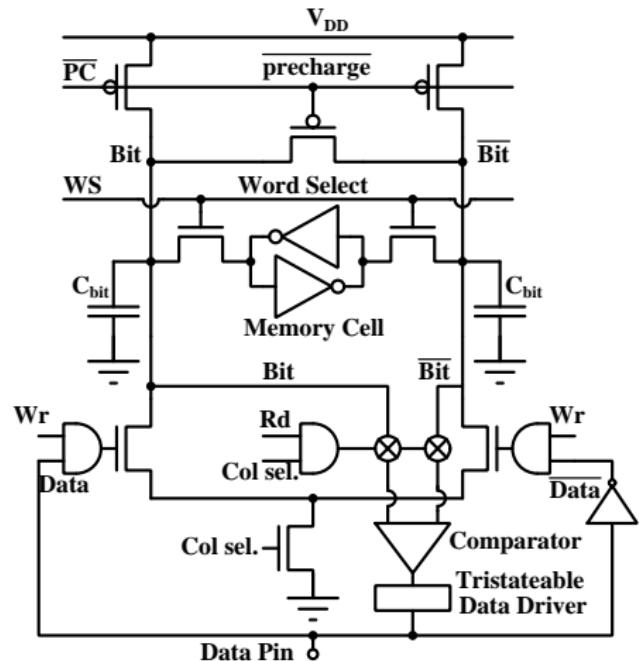


- The major work for writing a '1' has to be done at the 'high' node which must be pulled down to well below $V_{DD}/2$.
- This consideration fixes the geometry ratio of M4 and M6.

Notice that M1, M3 and M5 should have the same size as M2, M4 and M6 respectively.

SRAM cell: Full Data path

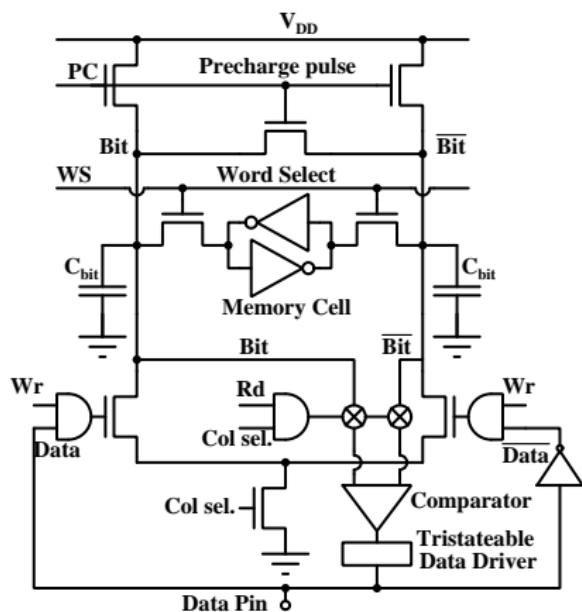
The data path during read and write cycles is shown below.



- All 3 pMOS transistors on the top turn on during pre-charge. The weak shorting transistor ensures that the voltages on Bit and $\overline{\text{Bit}}$ are equal.
- The pass gates connected to Bit and $\overline{\text{Bit}}$ lines are activated only if this is a read cycle and this column has been selected.
- When activated, these feed the sense amplifier/comparator, which then drives a tri-stateable buffer to the data pin.

SRAM cell: Alternate Bit line pull up

We can use nMOS transistors instead of pMOS to pre-charge the Bit and $\overline{\text{Bit}}$ lines. The pre-charge pulse should now be a positive pulse.

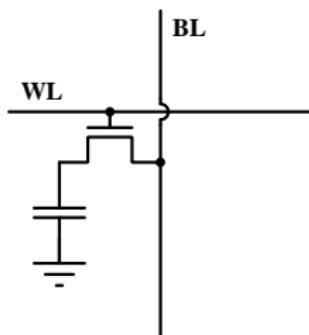


- Mobility of n channel transistors is higher and the source follower configuration has lower output impedance. So charging is much faster.
- However, the maximum voltage to which the Bit and $\overline{\text{Bit}}$ lines can be charged is now $V_{DD} - V_{Tn}$.
- Transistor geometries have to be adjusted due to lower voltage on the Bit and $\overline{\text{Bit}}$ lines.
- This lower voltage is actually an advantage for the design of sense amplifier, whose common mode range does not have to go all the way up to V_{DD} .

DRAM cell

- DRAM uses a single transistor per cell, so it is much denser than SRAM.
- The storage is on a capacitor – so it needs periodic refreshing to avoid data loss due to leakage.
- To store sufficient charge on the capacitor, special technological steps are required. So it is not process compatible with other CMOS circuits.

1T DRAM cell

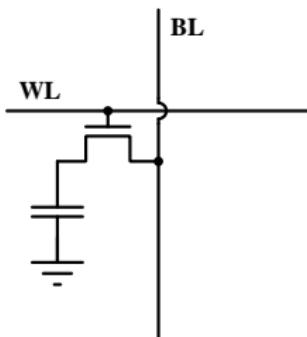


- In the read cycle, the bit line is pre-charged ‘high’.
- If the capacitor stores a ‘0’ (it is discharged), it pulls the voltage on the bit line a little lower due to charge sharing.
- If it stores a ‘1’, the voltage on the bit line remains unchanged.

Presence or absence of a voltage change is detected by the sense amplifier and driven to the output as in the case of an SRAM.

Destructive Read in a DRAM cell

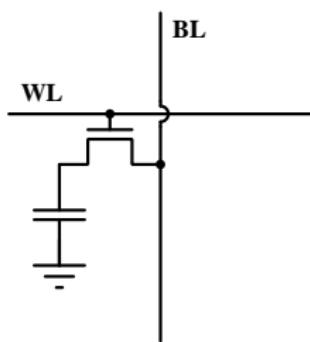
1T DRAM cell



- During a read cycle, the storage capacitor is connected to the bit line in the entire word line.
- Since the storage capacitor is much smaller than the bit line capacitor, it gets charged to '1' during charge sharing.
- All cells which had a '0' stored have their data destroyed!
- So the Read operation takes place on the entire row, and the read data needs to be written back to the entire row.
- Thus a read cycle is always followed by an internal write operation for the whole row.
- This also refreshes the data in the entire row, restoring any leaked charges.

Write Cycle in a DRAM cell

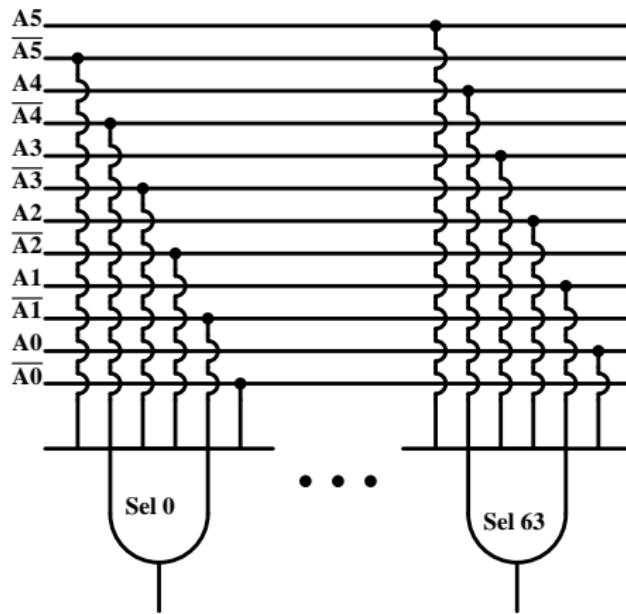
1T DRAM cell



- During a write cycle, the bit line is driven to the data value in the selected column.
- When the word select signal comes, the data is copied to the storage capacitor.
- The columns which are not selected by the current address go through a read cycle.
- This includes the refresh operation.
- So the data is refreshed for the entire word line whenever it is accessed for a read or a write operation.
- Banks of DRAM memories need a controller, which periodically performs a dummy read on every row in order to refresh the data stored in the memory.
- Modern DRAM chips have refresh circuitry on chip, which internally do the refresh operation if some word lines are not being accessed. These memories are also called SDRAMs.

Address decoding in Memories

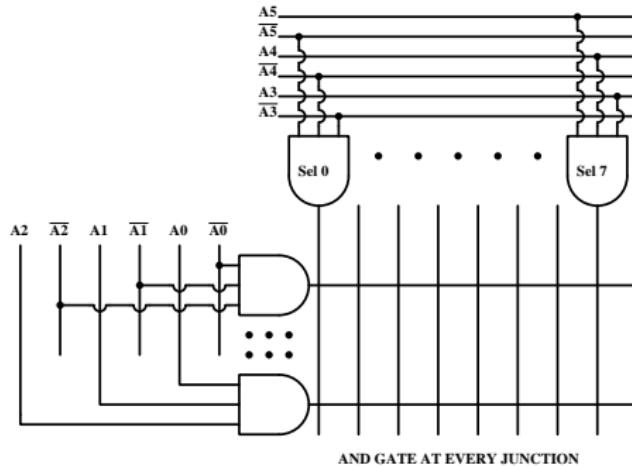
A one step decoder for 6 bit row or column address is shown below.



This kind of decoding is not practical.

Address decoding in Memories

A more practical approach decodes different groups of memory separately. We separately decode 3 bits each in this example and then combine the select outputs using AND gates.



I-O circuits

Dinesh Sharma

EE Department
IIT Bombay, Mumbai

November 19, 2020

IO circuits

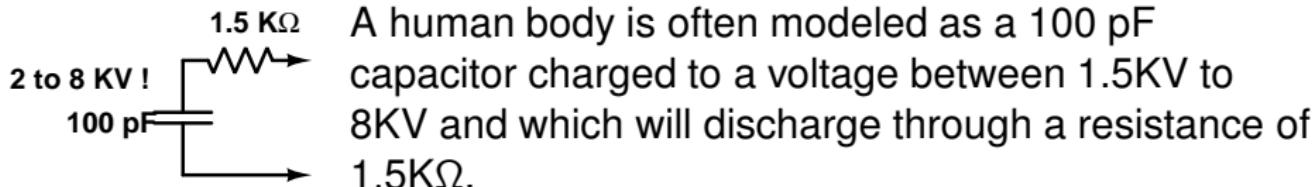
- Circuits used for I-O require special design and layout techniques.
- While devices and on-chip interconnects have been aggressively scaled over the years, connections to the external world have remained practically unscaled.
- A wire attached to the chip to connect to the external circuits cannot be made much thinner due to considerations of mechanical strength and soldering techniques.
- As a result connections to the outside circuits have to be through I-O pads which are large and not scaled with the technology.
- Wires are attached to these pads through ultrasonic or thermo-compression bonding. The core of the circuit has to be protected from the thermal and electrical stress produced by this process.

IO circuits

- The supply voltage in the core of VLSI needs to be scaled with the technology. System voltages outside the chip are changed much less frequently.
- A system may need to integrate multiple chips which work at different supply voltages.
- As a result, we may need to provide voltage translation from external signals to internal signals and *vice versa*.
- Signals coming in from outside may have slower rise and fall times with noise imposed on them. Input buffers may have to use hysteresis to avoid false transitions from '0' to '1' and back due to noise, as the input transitions through the trip voltage between these levels.

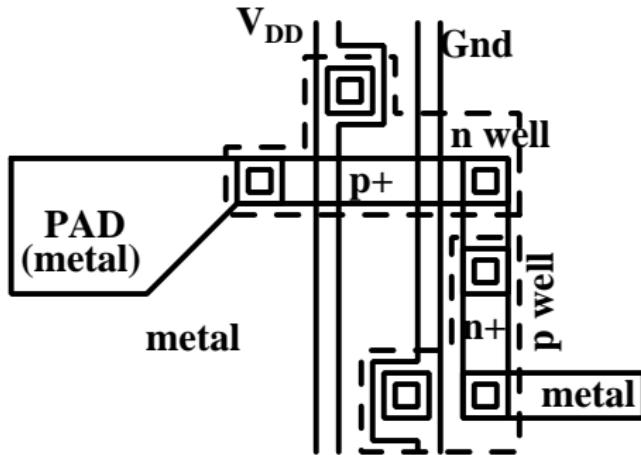
I/O circuits

- IC need to be handled by human beings before being inserted into circuits.
- Human beings may carry huge electrostatic charges.
- Since CMOS circuitry has high input impedance and very thin gate oxides, input transistors will breakdown immediately when exposed to high voltages.

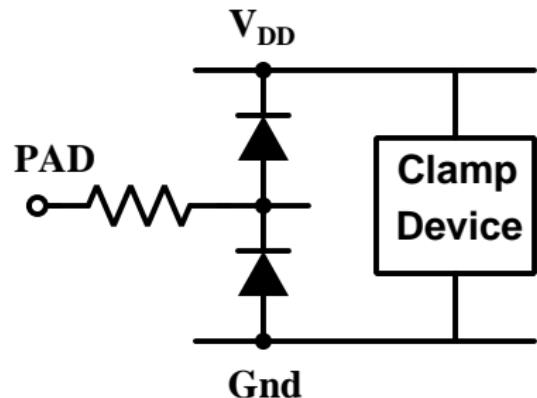


Special protection circuits have to be placed at inputs to protect delicate MOS devices from destruction due to these high voltages. This protection should work even when the chip is not powered!

Input pad protection

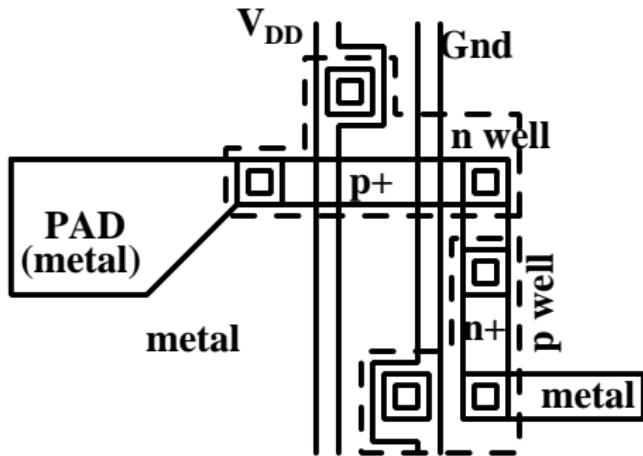


Not to scale!

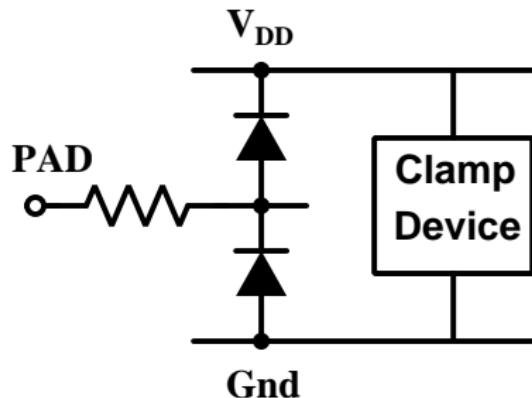


The resistors and diodes are actually the same device – a p+ diffused line in an n-well and an n+ diffused line in a p-well.

Input pad protection



Not to scale!

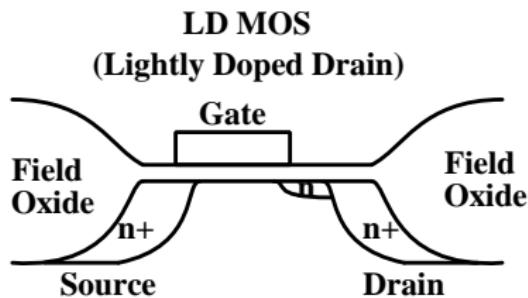


Circuits at I-O pads need to be put in separate n and p wells, with good n+ and p+ guard bands around these to prevent latchup.

Designing the clamp device

- The clamp device must *not* turn on as long as the voltage between the supply rails is within specifications.
- However, it *must* turn on if the voltage exceeds the specified supply voltage by a relatively small amount.
- The clamping action should be non-destructive.
- This puts severe limits on the turn on voltage of the clamp device with process and temperature variations and scaled down supply voltages.
- Typically, SCR devices (using the latchup structure!) or thick oxide MOS devices are used for this purpose.

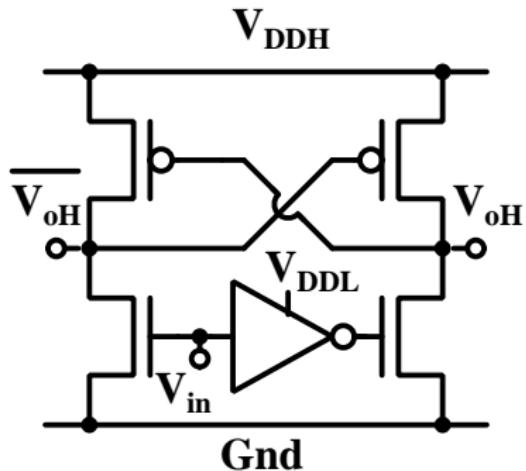
I-O transistors



Special transistors with thicker gate oxides and lightly doped extensions of the drain to stand higher than usual voltages at their gates and drains are provided in most CMOS processes these days

- These transistors are called I-O transistors or H-V transistors.
- Because of higher oxide thickness, their transconductance is lower.
- So these are not often used in the core of the VLSI.

Voltage Level Translation at input or output

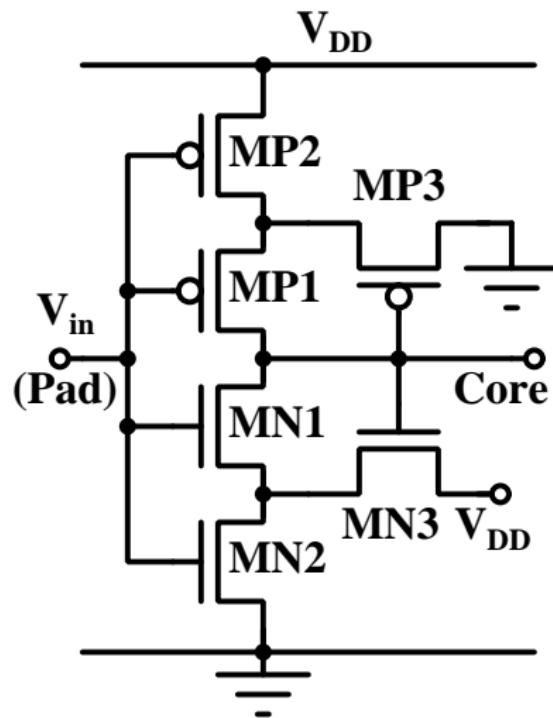


- The circuit shows a low swing to high swing translator using CVSL.
- The inverter uses low supply voltage.
- Transistors are high voltage devices.

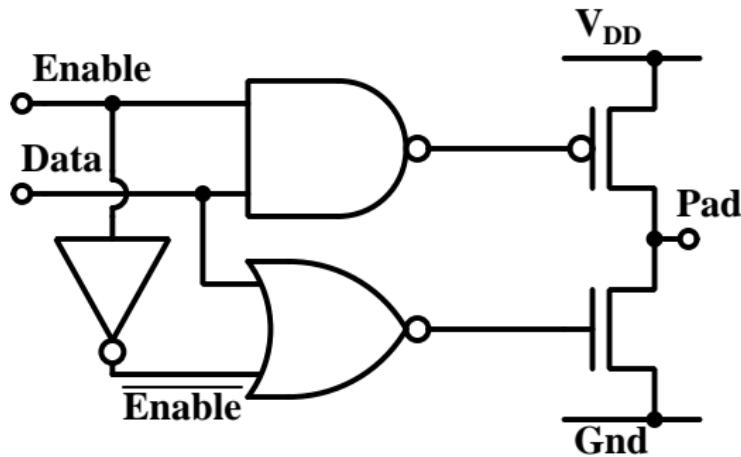
Thick oxide transistors

High swing to Low swing can be managed with a simple inverter using thick oxide devices.

Noise immunity through Hysteresis



Bi-directional Output driver



Compact Bi-directional Pad

