CMOS LOGIC

A CMOS transistor (or device) has four terminals: gate , source , drain , and a fourth terminal that we shall ignore until the next section. A CMOS transistor is a switch. The switch must be conducting or on to allow current to flow between the source and drain terminals (using open and closed for switches is confusing-for the same reason we say a tap is on and not that it is closed). The transistor source and drain terminals are equivalent as far as digital signals are concerned-we do not worry about labeling an electrical switch with two terminals.

- V_{AB} is the potential difference, or voltage, between nodes A and B in a circuit; V_{AB} is positive if node A is more positive than node B.
- Italics denote variables; constants are set in roman (upright) type. Uppercase letters denote DC, large-signal, or steady-state voltages.
- For TTL the positive power supply is called VCC (V_{CC} or V_{CC}). The ’C’ denotes that the supply is connected indirectly to the collectors of the npn bipolar transistors (a bipolar transistor has a collector, base, and emitter-corresponding roughly to the drain, gate, and source of an MOS transistor).
- Following the example of TTL we used VDD (V_{DD} or V_{DD}) to denote the positive supply in an NMOS chip where the devices are all n -channel transistors and the drains of these devices are connected indirectly to the positive supply. The supply nomenclature for NMOS chips has stuck for CMOS.
- VDD is the name of the power supply node or net; V_{DD} represents the value (uppercase since V_{DD} is a DC quantity). Since V_{DD} is a variable, it is italic (words and multiletter abbreviations use roman-thus it is V_{DD} , but V_{drain}).
- Logic designers often call the CMOS negative supply VSS or VSS even if it is actually ground or GND. I shall use VSS for the node and V_{SS} for the value.
- CMOS uses positive logic -VDD is logic ’1’ and VSS is logic ’0’.

We turn a transistor on or off using the gate terminal. There are two kinds of CMOS transistors: n -channel transistors and p -channel transistors. An n -channel transistor requires a logic ’1’ (from now on I’ll just say a ’1’) on the gate to make the switch conducting (to turn the transistor on). A p -channel transistor requires a logic ’0’ (again from now on, I’ll just say a ’0’) on the gate to make the switch nonconducting (to turn the transistor off). The p -channel transistor symbol has a bubble on its gate to remind us that the gate has to be a ’0’ to turn the transistor on . All this is shown in Figure 2.1(a) and (b).

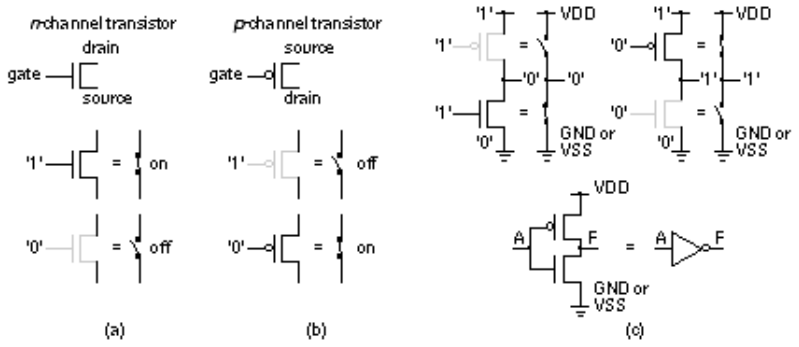


FIGURE 2.1 CMOS transistors as switches. (a) An n -channel transistor. (b) A p -channel transistor. (c) A CMOS inverter and its symbol (an equilateral triangle and a circle).

If we connect an n -channel transistor in series with a p -channel transistor, as shown in Figure 2.1(c), we form an inverter . With four transistors we can form a two-input NAND gate (Figure 2.2a). We can also make a two-input NOR gate (Figure 2.2b). Logic designers normally use the terms NAND gate and logic gate (or just gate), but I shall try to use the terms NAND cell and logic cell rather than NAND gate or logic gate in this chapter to avoid any possible confusion with the gate terminal of a transistor.

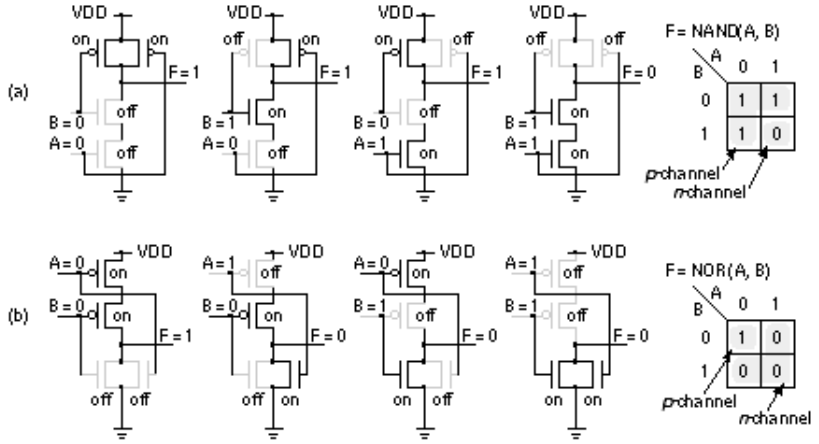


FIGURE 2.2 CMOS logic. (a) A two-input NAND logic cell. (b) A two-input NOR logic cell. The n -channel and p -channel transistor switches implement the ’1’s and ’0’s of a Karnaugh map.

2.1 CMOS Transistors

2.2 The CMOS Process

2.3 CMOS Design Rules

2.4 Combinational Logic Cells

2.5 Sequential Logic Cells

2.6 Datapath Logic Cells

2.7 I/O Cells

2.8 Cell Compilers

2.9 Summary

2.10 Problems

2.11 Bibliography

2.12 References

2.1 CMOS Transistors

Figure 2.3 illustrates how electrons and holes abandon their dopant atoms leaving a depletion region around a transistor's source and drain. The region between source and drain is normally nonconducting. To make an n -channel transistor conducting, we must apply a positive voltage V_{GS} (the gate voltage with respect to the source) that is greater than the n -channel transistor threshold voltage , V_{tn} (a typical value is 0.5 V and, as far as we are presently concerned, is a constant). This establishes a thin (ª 50 Å) conducting channel of electrons under the gate. MOS transistors can carry a very small current (the subthreshold current -a few microamperes or less) with $V_{GS} < V_{tn}$, but we shall ignore this. A transistor can be conducting ($V_{GS} > V_{tn}$) without any current flowing. To make current flow in an n -channel transistor we must also apply a positive voltage, V_{DS} , to the drain with respect to the source. Figure 2.3 shows these connections and the connection to the fourth terminal of an MOS transistor-the bulk (well , tub , or substrate) terminal. For an n -channel transistor we must connect the bulk to the most negative potential, GND or VSS, to reverse bias the bulk-to-drain and bulk-to-source pn -diodes. The arrow in the four-terminal n -channel transistor symbol in Figure 2.3 reflects the polarity of these pn -diodes.

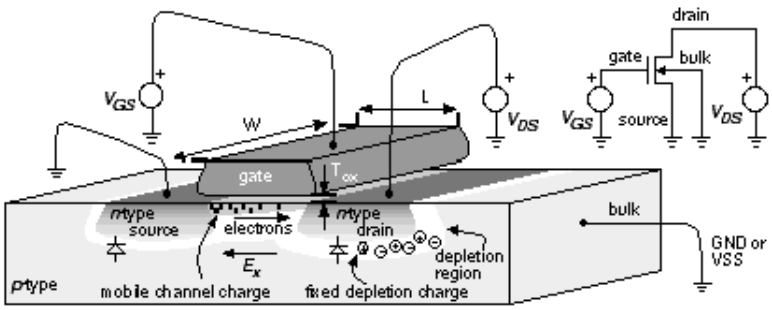


FIGURE 2.3 An n -channel MOS transistor. The gate-oxide thickness, T_{OX} , is approximately 100 angstroms (0.01 m m). A typical transistor length, L = 2 l . The bulk may be either the substrate or a well. The diodes represent pn -junctions that must be reverse-biased.

The current flowing in the transistor is

current (amperes) = charge (coulombs) per unit time (second). (2.1)

We can express the current in terms of the total charge in the channel, Q (imagine taking a picture and counting the number of electrons in the channel at that instant). If t_f (for time of flight -sometimes called the transit time) is the time that it takes an electron to cross between source and drain, the drain-to-source current, I_{DSn} , is

$$I_{DSn} = Q / t_f \cdot \quad (2.2)$$

We need to find Q and t_f . The velocity of the electrons v (a vector) is given by the equation that forms the basis of Ohm's law:

$$v = - m_n E , \quad (2.3)$$

where m_n is the electron mobility (m_p is the hole mobility) and E is the electric field (with units Vm^{-1}).

Typical carrier mobility values are m_n = 500-1000 $cm^2 V^{-1} s^{-1}$ and m_p = 100-400 $cm^2 V^{-1} s^{-1}$. Equation 2.3 is a vector equation, but we shall ignore the vertical electric field and concentrate on the horizontal electric field, E_x , that moves the electrons between source and drain. The horizontal component of the electric field is $E_x = - V_{DS} / L$, directed from the drain to the source, where L is the channel length (see Figure 2.3). The electrons travel a distance L with horizontal velocity $v_x = - m_n E_x$, so that

$$t_f = \frac{L}{} = \frac{L^2}{} \quad . \quad (2.4)$$

$v_x \quad m_n V_{DS}$

Next we find the channel charge, Q . The channel and the gate form the plates of a capacitor, separated by an insulator-the gate oxide. We know that the charge on a linear capacitor, C, is Q = C V . Our lower plate, the channel, is not a linear conductor. Charge only appears on the lower plate when the voltage between the gate and the channel, V_{GC} , exceeds the n -channel threshold voltage. For our nonlinear capacitor we need to modify the equation for a linear capacitor to the following:

$$Q = C (V_{GC} - V_{tn}) \cdot \quad (2.5)$$

The lower plate of our capacitor is resistive and conducting current, so that the potential in the channel, V_{GC} , varies. In fact, $V_{GC} = V_{GS}$ at the source and $V_{GC} = V_{GS} - V_{DS}$ at the drain. What we really should do is find an expression for the channel charge as a function of channel voltage and sum (integrate) the charge all the way across the channel, from x = 0 (at the source) to x = L (at the drain). Instead we shall assume that the channel voltage, V_{GC} (x), is a linear function of distance from the source and take the average value of the charge, which is thus

$$Q = C [(V_{GS} - V_{tn}) - 0.5 V_{DS}] \cdot \quad (2.6)$$

The gate capacitance, C , is given by the formula for a parallel-plate capacitor with length L , width W , and plate separation equal to the gate-oxide thickness, T_{ox} . Thus the gate capacitance is

$$C = \frac{WL e_{ox}}{T_{ox}} = WLC_{ox} \quad , (2.7)$$

where e_{ox} is the gate-oxide dielectric permittivity. For silicon dioxide, $Si0_2$, e_{ox} ª 3.45 ¥ 10^{-11} Fm^{-1} , so that, for a typical gate-oxide thickness of 100 Å (1 Å = 1 angstrom = 0.1 nm), the gate capacitance per unit area, C_{ox} ª 3 f F m m^{-2} .

Now we can express the channel charge in terms of the transistor parameters,

$$Q = WL C_{ox} [(V_{GS} - V_{tn}) - 0.5 V_{DS}] \cdot \quad (2.8)$$

Finally, the drain-source current is

$$\begin{aligned} I_{DSn} &= Q/ t_f \\ &= (W/L) m_n C_{ox} [(V_{GS} - V_{tn}) - 0.5 V_{DS}] V_{DS} \\ &= (W/L)k'_n [(V_{GS} - V_{tn}) - 0.5 V_{DS}] V_{DS} \cdot \qquad (2.9) \end{aligned}$$

The constant k'_n is the process transconductance parameter (or intrinsic transconductance):

$$k'_n = m_n C_{ox} . \quad (2.10)$$

We also define b_n , the transistor gain factor (or just gain factor) as

$$b_n = k'_n (W/L) . \quad (2.11)$$

The factor W/L (transistor width divided by length) is the transistor shape factor .

Equation 2.9 describes the linear region (or triode region) of operation. This equation is valid until $V_{DS} = V_{GS} - V_{tn}$ and then predicts that I_{DS} decreases with increasing V_{DS} , which does not make physical sense. At $V_{DS} = V_{GS} - V_{tn} = V_{DS(sat)}$ (the saturation voltage) there is no longer enough voltage between the gate and the drain end of the channel to support any channel charge. Clearly a small amount of charge remains or the current would go to zero, but with very little free charge the channel resistance in a small region close to the drain increases rapidly and any further increase in V_{DS} is dropped over this region. Thus for $V_{DS} > V_{GS} - V_{tn}$ (the saturation region , or pentode region, of operation) the drain current IDS remains approximately constant at the saturation current , $I_{DSn(sat)}$, where

$$I_{DSn(sat)} = (b_n/2)(V_{GS} - V_{tn})^2 \; ; \quad V_{GS} > V_{tn} . \quad (2.12)$$

Figure 2.4 shows the n -channel transistor I_{DS} -V_{DS} characteristics for a generic 0.5 m m CMOS process that we shall call G5 . We can fit Eq. 2.12 to the long-channel transistor characteristics (W = 60 m m, L = 6 m m) in Figure 2.4(a). If $I_{DSn(sat)}$ = 2.5 mA (with V_{DS} = 3.0 V, V_{GS} = 3.0 V, V_{tn} = 0.65 V, T_{ox} =100 Å), the intrinsic transconductance is

$$k'_n = \frac{2(L/W)\, I_{DSn(sat)}}{(V_{GS} - V_{tn})^2} \quad (2.13)$$

$$= \frac{2\,(6/60)\,(2.5 \times 10^{-3})}{(3.0 - 0.65)^2}$$

$$= 9.05 \times 10^{-5}\ AV^{-2}$$

or approximately 90 m AV $^{-2}$. This value of k'_n , calculated in the saturation region, will be different (typically lower by a factor of 2 or more) from the value of k'_n measured in the linear region. We assumed the mobility, m_n , and the threshold voltage, V_{tn} , are constants-neither of which is true, as we shall see in Section 2.1.2.

For the p -channel transistor in the G5 process, $I_{DSp\,(sat)}$ = -850 m A (V_{DS} = -3.0 V, V_{GS} = -3.0 V, V_{tp} = -0.85 V, W = 60 m m, L = 6 m m). Then

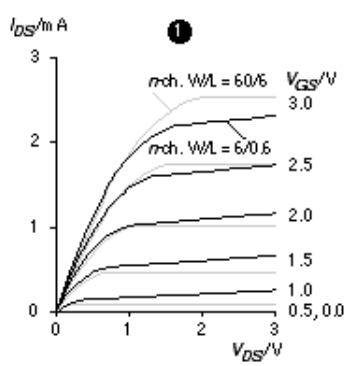
$$k'_p = \frac{2\,(L/W)\,(-I_{DSp\,(sat)})}{(V_{GS} - V_{tp})^2} \qquad (2.14)$$

$$= \frac{2\,(6/60)\,(850 \times 10^{-6})}{(-3.0 - (-0.85))^2}$$

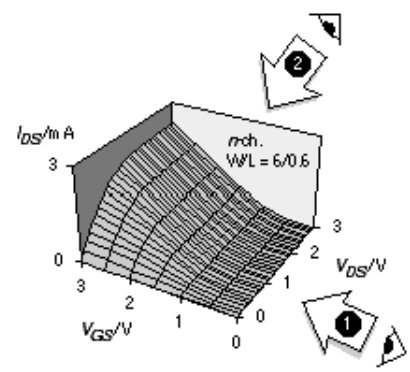
$$= 3.68 \times 10^{-5}\ \mathrm{AV}^{-2}$$

The next section explains the signs in Eq. 2.14.

(a)



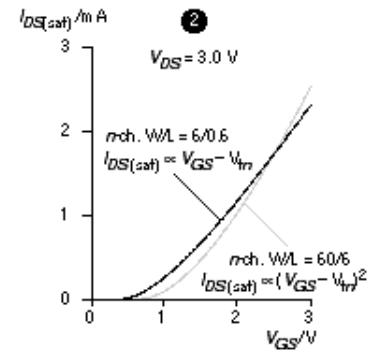
(b)



(c)

FIGURE 2.4 MOS n -channel transistor characteristics for a generic

FIGURE 2.4 MOS n -channel transistor characteristics for a generic 0.5 m m process (G5). (a) A short-channel transistor, with W = 6 m m and L = 0.6 m m (drawn) and a long-channel transistor (W = 60 m m, L = 6 m m) (b) The 6/0.6 characteristics represented as a surface. (c) A long-channel transistor obeys a square-law characteristic between I_{DS} and V_{GS} in the saturation region (V_{DS} = 3 V). A short-channel transistor shows a more linear characteristic due to velocity saturation. Normally, all of the transistors used on an ASIC have short channels.



2.1.1 P-Channel Transistors

The source and drain of CMOS transistors look identical; we have to know which way the current is flowing to distinguish them. The source of an n -channel transistor is lower in potential than the drain and vice versa for a p -channel transistor. In an n -channel transistor the threshold voltage, V_{tn} , is normally positive, and the terminal voltages V_{DS} and V_{GS} are also usually positive. In a p -channel transistor V_{tp} is normally negative and we have a choice: We can write everything in terms of the magnitudes of the voltages and currents or we can use negative signs in a consistent fashion.

Here are the equations for a p -channel transistor using negative signs:

$$I_{DSp} = -k'_p (W/L) [(V_{GS} - V_{tp}) - 0.5 V_{DS}] V_{DS} \; ; \quad V_{DS} > V_{GS} - V_{tp} \quad (2.15)$$

$$I_{DSp\,(sat)} = - b_p /2 (V_{GS} - V_{tp})^2 \; ; \quad V_{DS} < V_{GS} - V_{tp} .$$

In these two equations V_{tp} is negative, and the terminal voltages V_{DS} and V_{GS} are also normally negative (and -3 V < -2 V, for example). The current I_{DSp} is then negative, corresponding to conventional current flowing from source to drain of a p -channel transistor (and hence the negative sign for $I_{DSp\,(sat)}$ in Eq. 2.14).

2.1.2 Velocity Saturation

For a deep submicron transistor, Eq. 2.12 may overestimate the drain-source current by a factor of 2 or more. There are three reasons for this error. First, the threshold voltage is not constant. Second, the actual length of the channel (the electrical or effective length, often written as L_{eff}) is less than the drawn (mask) length. The third reason is that Eq. 2.3 is not valid for high electric fields. The electrons cannot move any faster than about $v_{max\,n} = 10^5$ ms^{-1} when the electric field is above 10^6 Vm^{-1} (reached when 1 V is dropped across 1 m m); the electrons become velocity saturated . In this case $t_f = L_{eff} / v_{max\,n}$, the drain-source saturation current is independent of the transistor length, and Eq. 2.12 becomes

$$I_{DSn\,(sat)} = W v_{max\,n} C_{ox} (V_{GS} - V_{t\,n}) ; \quad V_{DS} > V_{DS\,(sat)} \text{ (velocity saturated).} \quad (2.16)$$

We can see this behavior for the short-channel transistor characteristics in Figure 2.4(a) and (c).

Transistor current is often specified per micron of gate width because of the form of Eq. 2.16. As an example, suppose $I_{DSn\,(sat)} / W = 300$ m A m m $^{-1}$ for the n -channel transistors in our G5 process (with $V_{DS} = 3.0$ V, $V_{GS} = 3.0$ V, $V_{t\,n} = 0.65$ V, $L_{eff} = 0.5$ m m and $T_{ox} = 100$ Å). Then E_x ª (3 - 0.65) V / 0.5 m m ª 5 V m m $^{-1}$,

$$v_{max\,n} = \frac{I_{DSn\,(sat)} / W}{C_{ox} (V_{GS} - V_{t\,n})} \quad (2.17)$$

$$= \frac{(300 ¥ 10^{-6}) (1 ¥ 10^{6})}{(3.45 ¥ 10^{-3}) (3 - 0.65)}$$

$$= 37{,}000 \text{ ms}^{-1}$$

and t_f ª 0.5 m m/37,000 ms $^{-1}$ ª 13 ps.

The value for $v_{max\,n}$ is lower than the 10^5 ms $^{-1}$ we expected because the carrier velocity is also lowered by mobility degradation due the vertical electric field-which we have ignored. This vertical field forces the carriers to keep "bumping" in to the interface between the silicon and the gate oxide, slowing them down.

2.1.3 SPICE Models

The simulation program SPICE (which stands for Simulation Program with Integrated Circuit Emphasis) is often used to characterize logic cells. Table 2.1 shows a typical set of model parameters for our G5 process. The SPICE parameter KP (given in m AV $^{-2}$) corresponds to k'_n (and k'_p). SPICE parameters VT0 and TOX correspond to $V_{t\,n}$ (and $V_{t\,p}$), and T_{ox} . SPICE parameter U0 (given in cm ^{2} V $^{-1}$ s $^{-1}$) corresponds to the ideal bulk mobility values, m_n (and m_p). Many of the other parameters model velocity saturation and mobility degradation (and thus the effective value of k'_n and k'_p).

TABLE 2.1 SPICE parameters for a generic 0.5 m m process, G5 (0.6 m m drawn gate length). The n-channel transistor characteristics are shown in Figure 2.4.

.MODEL CMOSN NMOS LEVEL=3 PHI=0.7 TOX=10E-09 XJ=0.2U TPG=1 VTO=0.65

DELTA=0.7
+ LD=5E-08 KP=2E-04 UO=550 THETA=0.27 RSH=2 GAMMA=0.6 NSUB=1.4E+17 NFS=6E+11
+ VMAX=2E+05 ETA=3.7E-02 KAPPA=2.9E-02 CGDO=3.0E-10 CGSO=3.0E-10 CGBO=4.0E-10
+ CJ=5.6E-04 MJ=0.56 CJSW=5E-11 MJSW=0.52 PB=1
.MODEL CMOSP PMOS LEVEL=3 PHI=0.7 TOX=10E-09 XJ=0.2U TPG=-1 VTO=-0.92
DELTA=0.29
+ LD=3.5E-08 KP=4.9E-05 UO=135 THETA=0.18 RSH=2 GAMMA=0.47 NSUB=8.5E+16
NFS=6.5E+11
+ VMAX=2.5E+05 ETA=2.45E-02 KAPPA=7.96 CGDO=2.4E-10 CGSO=2.4E-10 CGBO=3.8E-10
+ CJ=9.3E-04 MJ=0.47 CJSW=2.9E-10 MJSW=0.505 PB=1

2.1.4 Logic Levels

Figure 2.5 shows how to use transistors as logic switches. The bulk connection for the n -channel transistor in Figure 2.5(a-b) is a p -well. The bulk connection for the p -channel transistor is an n -well. The remaining connections show what happens when we try and pass a logic signal between the drain and source terminals.

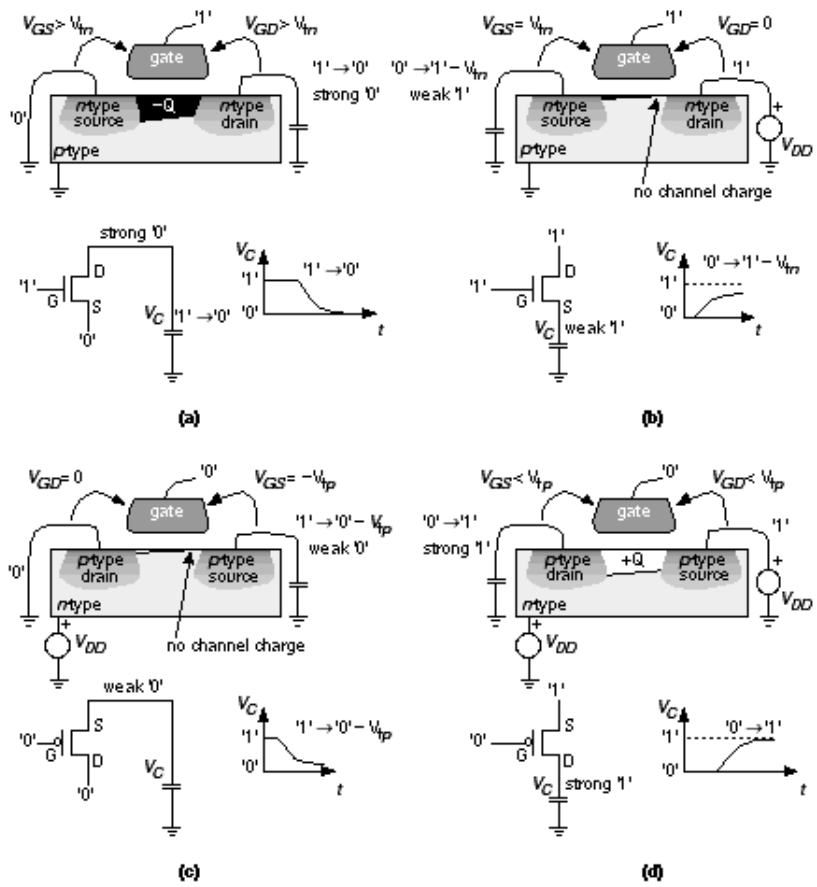


FIGURE 2.5 CMOS logic levels. (a) A strong ’0’. (b) A weak ’1’. (c) A weak ’0’. (d) A strong ’1’. (V_{tn} is positive and V_{tp} is negative.) The depth of the channels is greatly exaggerated.

In Figure 2.5(a) we apply a logic ’1’ (or VDD -I shall use these interchangeably) to the gate and a logic ’0’ (V_{SS}) to the source (we know it is the source since electrons must flow from this point, since V_{SS} is the lowest voltage on the chip). The application of these voltages makes the n -channel transistor conduct current, and electrons flow from source to drain.

Suppose the drain is initially at logic ’1’; then the n -channel transistor will begin to discharge any capacitance that is connected to its drain (due to another logic cell, for example). This will continue until the drain terminal reaches a logic ’0’, and at that time V_{GD} and V_{GS} are both equal to V_{DD} , a full logic ’1’. The transistor is strongly conducting now (with a large channel charge, Q , but there is no current flowing since V_{DS} = 0 V). The transistor will strongly object to attempts to change its drain terminal from a logic ’0’. We say that the logic level at the drain is a strong ’0’.

In Figure 2.5(b) we apply a logic ’1’ to the drain (it must now be the drain since electrons have to flow toward a logic ’1’). The situation is now quite different-the transistor is still on but V_{GS} is decreasing as the source voltage approaches its final value. In fact, the source terminal never gets to a logic ’1’-the source will stop increasing in voltage when V_{GS} reaches V_{tn} . At this point the transistor is very nearly off and the source voltage creeps slowly up to $V_{DD} - V_{tn}$. Because the transistor is very nearly off, it would be easy for a logic cell connected to the source to change the potential there, since there is so little channel charge. The logic level at the source is a weak ’1’. Figure 2.5(c-d) show the state of affairs for a p -channel transistor is the exact reverse or complement of the n -channel transistor situation.

In summary, we have the following logic levels:

- An n -channel transistor provides a strong ’0’, but a weak ’1’.
- A p -channel transistor provides a strong ’1’, but a weak ’0’.

Sometimes we refer to the weak versions of ’0’ and ’1’ as degraded logic levels . In CMOS technology we can use both types of transistor together to produce strong ’0’ logic levels as well as strong ’1’ logic levels.

2.2 The CMOS Process

Figure 2.6 outlines the steps to create an integrated circuit. The starting material is silicon, Si, refined from quartzite (with less than 1 impurity in 10^{10} silicon atoms). We draw a single-crystal silicon boule (or ingot) from a crucible containing a melt at approximately 1500 °C (the melting point of silicon at 1 atm. pressure is 1414 °C). This method is known as Czochralski growth. Acceptor (p -type) or donor (n -type) dopants may be introduced into the melt to alter the type of silicon grown.

The boule is sawn to form thin circular wafers (6, 8, or 12 inches in diameter, and typically 600 m m thick), and a flat is ground (the primary flat), perpendicular to the <110> crystal axis-as a "this edge down" indication. The boule is drawn so that the wafer surface is either in the (111) or (100) crystal planes. A smaller secondary flat indicates the wafer crystalline orientation and doping type. A typical submicron CMOS processes uses p -type (100) wafers with a resistivity of approximately 10 W cm-this type of wafer has two flats, 90° apart. Wafers are made by chemical companies and sold to the IC manufacturers. A blank 8-inch wafer costs about $100.

To begin IC fabrication we place a batch of wafers (a wafer lot) on a boat and grow a layer (typically a few thousand angstroms) of silicon dioxide , SiO_2 , using a furnace. Silicon is used in the semiconductor industry not so much for the properties of silicon, but because of the physical, chemical, and electrical properties of its native oxide, SiO_2 . An IC fabrication process contains a series of masking steps (that in turn contain other steps) to create the layers that define the transistors and metal interconnect.

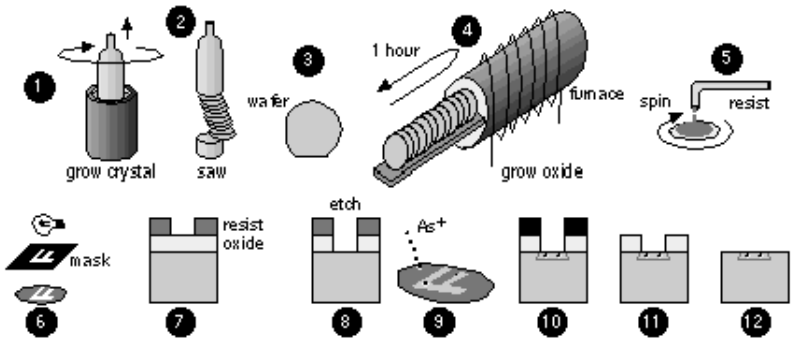


FIGURE 2.6 IC fabrication. Grow crystalline silicon (1); make a wafer (2-3); grow a silicon dioxide (oxide) layer in a furnace (4); apply liquid photoresist (resist) (5); mask exposure (6); a cross-section through a wafer showing the developed resist (7); etch the oxide layer (8); ion implantation (9-10); strip the resist (11); strip the oxide (12). Steps similar to 4-12 are repeated for each layer (typically 12-20 times for a CMOS process).

Each masking step starts by spinning a thin layer (approximately 1 m m) of liquid photoresist (resist) onto each wafer. The wafers are baked at about 100 °C to remove the solvent and harden the resist before being exposed to ultraviolet (UV) light (typically less than 200 nm wavelength) through a mask . The UV light alters the structure of the resist, allowing it to be removed by developing. The exposed oxide may then be etched (removed). Dry plasma etching etches in the vertical direction much faster than it does horizontally (an anisotropic etch). Wet etch techniques are usually isotropic . The resist functions as a mask during the etch step and transfers the desired pattern to the oxide layer.

Dopant ions are then introduced into the exposed silicon areas. Figure 2.6 illustrates the use of ion implantation . An ion implanter is a cross between a TV and a mass spectrometer and fires dopant ions into the silicon wafer. Ions can only penetrate materials to a depth (the range , normally a few microns) that depends on the closely controlled implant energy (measured in keV-usually between 10 and 100 keV; an electron volt, 1 eV, is 1.6 ¥ 10^{-19} J). By using layers of resist, oxide, and polysilicon we can prevent dopant ions from reaching the silicon surface and thus block the silicon from receiving an implant . We control the doping level by counting the number of ions we implant (by integrating the ion-beam current). The implant dose is measured in atoms/cm^2 (typical doses are from 10^{13} to 10^{15} cm^{-2}). As an alternative to ion implantation we may instead strip the resist and introduce dopants by diffusion from a gaseous source in a furnace.

Once we have completed the transistor diffusion layers we can deposit layers of other materials. Layers

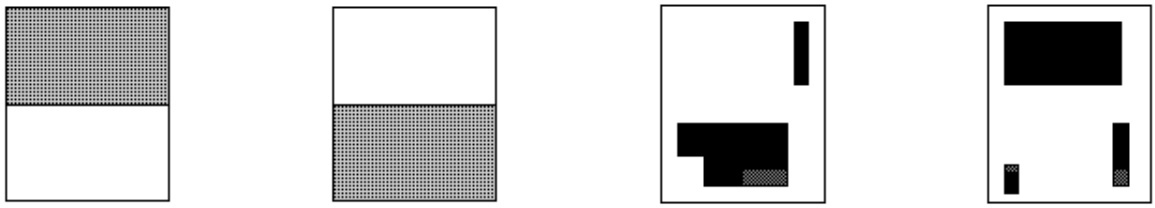
of polycrystalline silicon (polysilicon or poly), SiO_2 , and silicon nitride (Si_3N_4), for example, may be deposited using chemical vapor deposition (CVD). Metal layers can be deposited using sputtering . All these layers are patterned using masks and similar photolithography steps to those shown in Figure 2.6.

TABLE 2.2 CMOS process layers.

Mask/layer name	Derivation from drawn layers	Alternative names for mask/layer	MOSIS mask label
n -well	= nwell 1	bulk, substrate, tub, n -tub, moat	CWN
p -well	= pwell [1]	bulk, substrate, tub, p -tub, moat	CWP
active	= pdiff + ndiff	thin oxide, thinox, island, gate oxide	CAA
polysilicon	= poly	poly, gate	CPG
n -diffusion implant 2	= grow (ndiff)	ndiff, n -select, nplus, n+	CSN
p -diffusion implant [2]	= grow (pdiff)	pdiff, p -select, pplus, p+	CSP
contact	= contact	contact cut, poly contact, diffusion contact	CCP and CCA 3
metal1	= m1	first-level metal	CMF
metal2	= m2	second-level metal	CMS
via2	= via2	metal2/metal3 via, m2/m3 via	CVS
metal3	= m3	third-level metal	CMT
glass	= glass	passivation, overglass, pad	COG

Table 2.2 shows the mask layers (and their relation to the drawn layers) for a submicron, silicon-gate, three-level metal, self-aligned, CMOS process . A process in which the effective gate length is less than 1 m m is referred to as a submicron process . Gate lengths below 0.35 m m are considered in the deep-submicron regime.

Figure 2.7 shows the layers that we draw to define the masks for the logic cell of Figure 1.3. Potential confusion arises because we like to keep layout simple but maintain a "what you see is what you get" (WYSIWYG) approach. This means that the drawn layers do not correspond directly to the masks in all cases.



(a) nwell (b) pwell (c) ndiff (d) pdiff

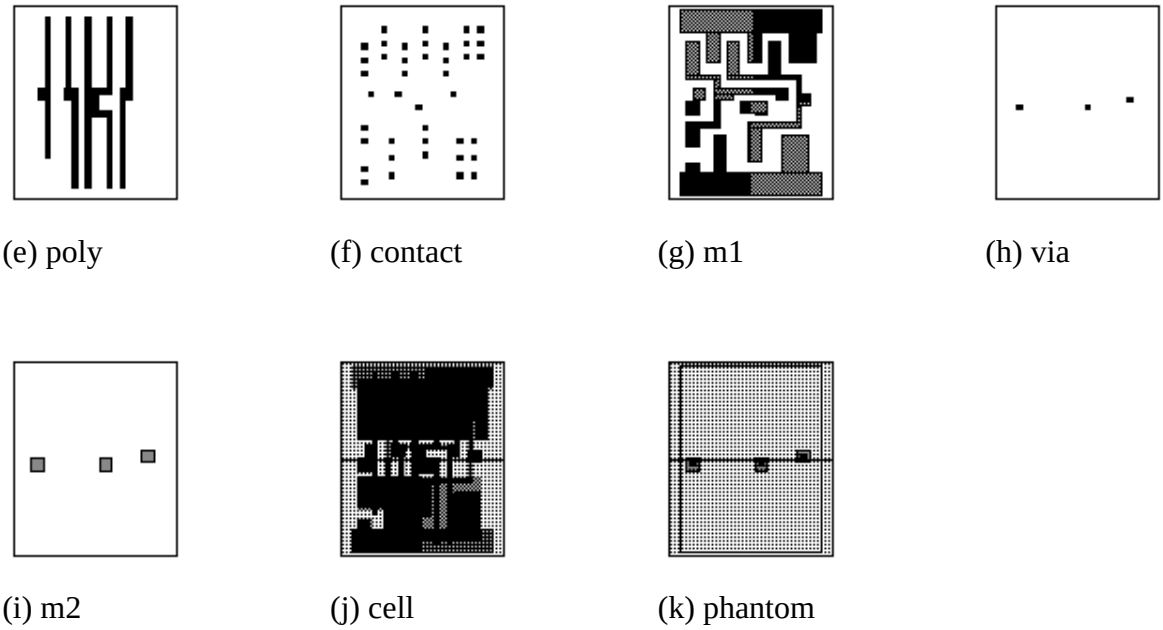


FIGURE 2.7 The standard cell shown in Figure 1.3. (a)-(i) The drawn layers that define the masks. The active mask is the union of the ndiff and pdiff drawn layers. The n -diffusion implant and p -diffusion implant masks are bloated versions of the ndiff and pdiff drawn layers. (j) The complete cell layout. (k) The phantom cell layout. Often an ASIC vendor hides the details of the internal cell construction. The phantom cell is used for layout by the customer and then "instantiated" by the ASIC vendor after layout is complete. This layout uses grayscale stipple patterns to distinguish between layers.

We can construct wells in a CMOS process in several ways. In an n-well process , the substrate is p -type (the wafer itself) and we use an n -well mask to build the n -well. We do not need a p -well mask because there are no p -wells in an n -well process-the n -channel transistors all sit in the substrate (the wafer)-but we often draw the p -well layer as though it existed. In a p-well process we use a p -well mask to make the p -wells and the n -wells are the substrate. In a twin-tub (or twin-well) process, we create individual wells for both types of transistors, and neither well is the substrate (which may be either n -type or p -type). There are even triple-well processes used to achieve even more control over the transistor performance. Whatever process that we use we must connect all the n -wells to the most positive potential on the chip, normally VDD, and all the p -wells to VSS; otherwise we may forward bias the bulk to source/drain pn -junctions. The bulk connections for CMOS transistors are not usually drawn in digital circuit schematics, but these substrate contacts (well contacts or tub ties) are very important. After we make the well(s), we grow a layer (approximately 1500 Å) of Si_3N_4 over the wafer. The active mask (CAA) leaves this nitride layer only in the active areas that will later become transistors or substrate contacts. Thus

CAA (mask) = ndiff (drawn) ? pdiff (drawn) , (2.18)

the ? symbol represents OR (union) of the two drawn layers, ndiff and pdiff. Everything outside the active areas is known as the field region, or just field .

Next we implant the substrate to prevent unwanted transistors from forming in the field region-this is the field implant or channel-stop implant . The nitride over the active areas acts as an implant mask and we may use another field-implant mask at this step also. Following this we grow a thick (approximately 5000 Å) layer of SiO_2 , the field oxide (FOX). The FOX will not grow over the nitride areas. When we strip the nitride we are left with FOX in the areas we do not want to dope the silicon. Following this we deposit, dope, mask, and etch the poly gate material, CPG (mask) = poly (drawn). Next we create the doped regions that form the sources, drains, and substrate contacts using ion implantation. The poly gate functions like masking tape in these steps. One implant (using phosphorous or arsenic ions) forms the n -type source/drain for the n -channel transistors and n -type substrate contacts (CSN). A second implant (using boron ions) forms the p -type source-drain for the p -channel transistors and p -type substrate contacts (CSP). These implants are masked as follows

CSN (mask) = grow (ndiff (drawn)), (2.19)

CSP (mask) = grow (pdiff (drawn)), (2.20)

where "grow" means that we expand or bloat the drawn ndiff and drawn pdiff layers slightly (usually by a few l).

During implantation the dopant ions are blocked by the resist pattern defined by the CSN and CSP masks. The CSN mask thus prevents the n -type regions being implanted with p -type dopants (and vice versa for the CSP mask). As we shall see, the CSN and CSP masks are not intended to define the edges of the n -type and p -type regions. Instead these two masks function more like newspaper that prevents paint from spraying everywhere. The dopant ions are also blocked from reaching the silicon surface by the poly gates and this aligns the edge of the source and drain regions to the edges of the gates (we call this a self-aligned process). In addition, the implants are blocked by the FOX and this defines the outside edges of the source, drain, and substrate contact regions.

The only areas of the silicon surface that are doped n -type are

n -diffusion (silicon) = (CAA (mask) ? CSN (mask)) ? (ÿ CPG (mask)) ; (2.21)

where the ? symbol represents AND (the intersection of two layers); and the ÿ symbol represents NOT.

Similarly, the only regions that are doped p -type are

p -diffusion (silicon) = (CAA (mask) ? CSP (mask)) ? (ÿ CPG (mask)) . (2.22)

If the CSN and CSP masks do not overlap, it is possible to save a mask by using one implant mask (CSN or CSP) for the other type (CSP or CSN). We can do this by using a positive resist (the pattern of resist remaining after developing is the same as the dark areas on the mask) for one implant step and a negative resist (vice versa) for the other step. However, because of the poor resolution of negative resist and because of difficulties in generating the implant masks automatically from the drawn diffusions (especially when opposite diffusion types are drawn close to each other or touching), it is now common to draw both implant masks as well as the two diffusion layers.

It is important to remember that, even though poly is above diffusion, the polysilicon is deposited first

and acts like masking tape. It is rather like airbrushing a stripe-you use masking tape and spray everywhere without worrying about making straight lines. The edges of the pattern will align to the edge of the tape. Here the analogy ends because the poly is left in place. Thus,

n -diffusion (silicon) = (ndiff (drawn)) ? (ÿ poly (drawn)) and (2.23)

p -diffusion (silicon) = (pdiff (drawn)) ? (ÿ poly (drawn)) . (2.24)

In the ASIC industry the names nplus, n +, and n -diffusion (as well as the p -type equivalents) are used in various ways. These names may refer to either the drawn diffusion layer (that we call ndiff), the mask (CSN), or the doped region on the silicon (the intersection of the active and implant mask that we call n -diffusion)-very confusing.

The source and drain are often formed from two separate implants. The first is a light implant close to the edge of the gate, the second a heavier implant that forms the rest of the source or drain region. The separate diffusions reduce the electric field near the drain end of the channel. Tailoring the device characteristics in this fashion is known as drain engineering and a process including these steps is referred to as an LDD process , for lightly doped drain ; the first light implant is known as an LDD diffusion or LDD implant.

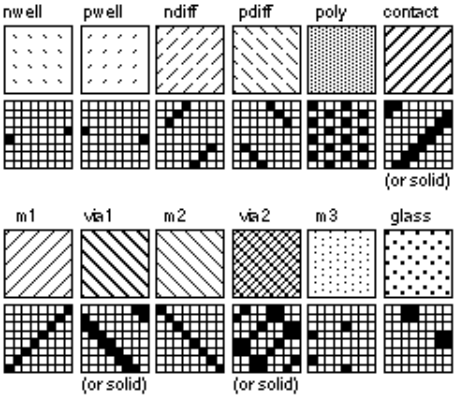


FIGURE 2.8 Drawn layers and an example set of black-and-white stipple patterns for a CMOS process. On top are the patterns as they appear in layout. Underneath are the magnified 8-by-8 pixel patterns. If we are trying to simplify layout we may use solid black or white for contact and vias. If we have contacts and vias placed on top of one another we may use stipple patterns or other means to help distinguish between them. Each stipple pattern is transparent, so that black shows through from underneath when layers are superimposed. There are no standards for these patterns.

Figure 2.8 shows a stipple-pattern matrix for a CMOS process. When we draw layout you can see through the layers-all the stipple patterns are OR’ed together. Figure 2.9 shows the transistor layers as they appear in layout (drawn using the patterns from Figure 2.8) and as they appear on the silicon. Figure 2.10 shows the same thing for the interconnect layers.

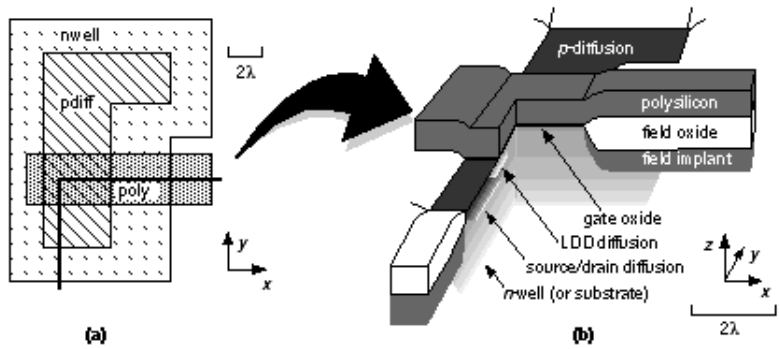
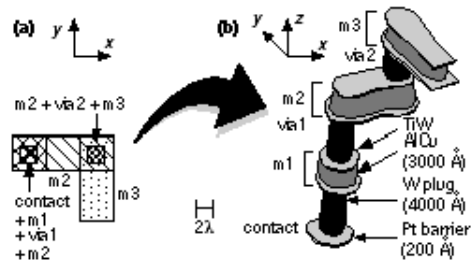


FIGURE 2.9 The transistor layers. (a) A p -channel transistor as drawn in layout. (b) The corresponding silicon cross section (the heavy lines in part a show the cuts). This is how a p -channel transistor would look just after completing the source and drain implant steps.

FIGURE 2.10 The interconnect layers. (a) Metal layers as drawn in layout. (b) The corresponding structure (as it might appear in a scanning-electron micrograph). The insulating layers between the metal layers are not shown. Contact is made to the underlying silicon through a platinum barrier layer. Each via consists of a tungsten plug. Each metal layer consists of a titanium-tungsten and aluminum-copper sandwich. Most deep submicron CMOS processes use metal structures similar to this. The scale, rounding, and irregularity of the features are realistic.



2.2.1 Sheet Resistance

Tables 2.3 and 2.4 show the sheet resistance for each conducting layer (in decreasing order of resistance) for two different generations of CMOS process.

TABLE 2.3 Sheet resistance (1 m m CMOS).

Layer	Sheet resistance	Units
n -well	1.15 ± 0.25	k W / square
poly	3.5 ± 2.0	W / square
n -diffusion	75 ± 20	W / square
p -diffusion	140 ± 40	W / square
m1/2	70± 6	m W / square
m3	30± 3	m W / square

TABLE 2.4 Sheet resistance (0.35 m m CMOS).

Layer	Sheet resistance	Units
n -well	1 ± 0.4	k W / square
poly	10 ± 4.0	W / square
n -diffusion	3.5 ± 2.0	W / square
p -diffusion	2.5 ± 1.5	W / square
m1/2/3	60 ± 6	m W / square
metal4	30 ± 3	m W / square

The diffusion layers, n -diffusion and p -diffusion, both have a high resistivity-typically from 1-100 W /square. We measure resistance in W / square (ohms per square) because for a fixed thickness of material it does not matter what the size of a square is-the resistance is the same. Thus the resistance of a

rectangular shape of a sheet of material may be calculated from the number of squares it contains times the sheet resistance in W / square. We can use diffusion for very short connections inside a logic cell, but not for interconnect between logic cells. Poly has the next highest resistance to diffusion. Most submicron CMOS processes use a silicide material (a metallic compound of silicon) that has much lower resistivity (at several W /square) than the poly or diffusion layers alone. Examples are tantalum silicide, TaSi; tungsten silicide, WSi; or titanium silicide, TiSi. The stoichiometry of these deposited silicides varies. For example, for tungsten silicide W:Si ª 1:2.6.

There are two types of silicide process. In a silicide process only the gate is silicided. This reduces the poly sheet resistance, but not that of the source-drain. In a self-aligned silicide (salicide) process, both the gate and the source-drain regions are silicided. In some processes silicide can be used to connect adjacent poly and diffusion (we call this feature LI , white metal, local interconnect, metal0, or m0). LI is useful to reduce the area of ASIC RAM cells, for example.

Interconnect uses metal layers with resistivities of tens of m W /square, several orders of magnitude less than the other layers. There are usually several layers of metal in a CMOS ASIC process, each separated by an insulating layer. The metal layer above the poly gate layer is the first-level metal (m1 or metal1), the next is the second-level metal (m2 or metal2), and so on. We can make connections from m1 to diffusion using diffusion contacts or to the poly using polysilicon contacts .

After we etch the contact holes a thin barrier metal (typically platinum) is deposited over the silicon and poly. Next we form contact plugs (via plugs for connections between metal layers) to reduce contact resistance and the likelihood of breaks in the contacts. Tungsten is commonly used for these plugs. Following this we form the metal layers as sandwiches. The middle of the sandwich is a layer (usually from 3000 Å to 10,000 Å) of aluminum and copper. The top and bottom layers are normally titanium-tungsten (TiW, pronounced "tie-tungsten"). Submicron processes use chemical-mechanical polishing (CMP) to smooth the wafers flat before each metal deposition step to help with step coverage.

An insulating glass, often sputtered quartz (SiO_2), though other materials are also used, is deposited between metal layers to help create a smooth surface for the deposition of the metal. Design rules may refer to this insulator as an intermetal oxide (IMO) whether they are in fact oxides or not, or interlevel dielectric (ILD). The IMO may be a spin-on polymer; boron-doped phosphosilicate glass (BPSG); Si_3N_4 ; or sandwiches of these materials (oxynitrides, for example).

We make the connections between m1 and m2 using metal vias , cuts , or just vias . We cannot connect m2 directly to diffusion or poly; instead we must make these connections through m1 using a via. Most processes allow contacts and vias to be placed directly above each other without restriction, arrangements known as stacked vias and stacked contacts . We call a process with m1 and m2 a two-level metal (2LM) technology. A 3LM process includes a third-level metal layer (m3 or metal3), and some processes include more metal layers. In this case a connection between m1 and m2 will use an m1/m2 via, or via1 ; a connection between m2 and m3 will use an m2/m3 via, or via2 , and so on.

The minimum spacing of interconnects, the metal pitch , may increase with successive metal layers. The minimum metal pitch is the minimum spacing between the centers of adjacent interconnects and is equal to the minimum metal width plus the minimum metal spacing.

Aluminum interconnect tends to break when carrying a high current density. Collisions between high-energy electrons and atoms move the metal atoms over a long period of time in a process known as electromigration . Copper is added to the aluminum to help reduce the problem. The other solution is to reduce the current density by using wider than minimum-width metal lines.

Tables 2.5 and 2.6 show maximum specified contact resistance and via resistance for two generations of CMOS processes. Notice that a m1 contact in either process is equal in resistance to several hundred squares of metal.

TABLE 2.5 Contact resistance (1 m m CMOS).

Contact/via type	Resistance (maximum)
m2/m3 via (via2)	5 W
m1/m2 via (via1)	2 W
m1/ p -diffusion contact	20 W
m1/ n -diffusion contact	20 W
m1/poly contact	20 W

TABLE 2.6 Contact resistance (0.35 m m CMOS).

Contact/via type	Resistance (maximum)
m2/m3 via (via2)	6 W
m1/m2 via (via1)	6 W
m1/ p -diffusion contact	20 W
m1/ n -diffusion contact	20 W
m1/poly contact	20 W

1. If only one well layer is drawn, the other mask may be derived from the drawn layer. For example, p -well (mask) = not (nwell (drawn)). A single-well process requires only one well mask.

2. The implant masks may be derived or drawn.

3. Largely for historical reasons the contacts to poly and contacts to active have different layer names. In the past this allowed a different sizing or process bias to be applied to each contact type when the mask was made.

2.3 CMOS Design Rules

Figure 2.11 defines the design rules for a CMOS process using pictures. Arrows between objects denote a minimum spacing, and arrows showing the size of an object denote a minimum width. Rule 3.1, for example, is the minimum width of poly (2 l). Each of the rule numbers may have different values for different manufacturers-there are no standards for design rules. Tables 2.7-2.9 show the MOSIS scalable CMOS rules. Table 2.7 shows the layer rules for the process front end , which is the front end of the line (as in production line) or FEOL . Table 2.8 shows the rules for the process back end (BEOL), the metal interconnect, and Table 2.9 shows the rules for the pad layer and glass layer.

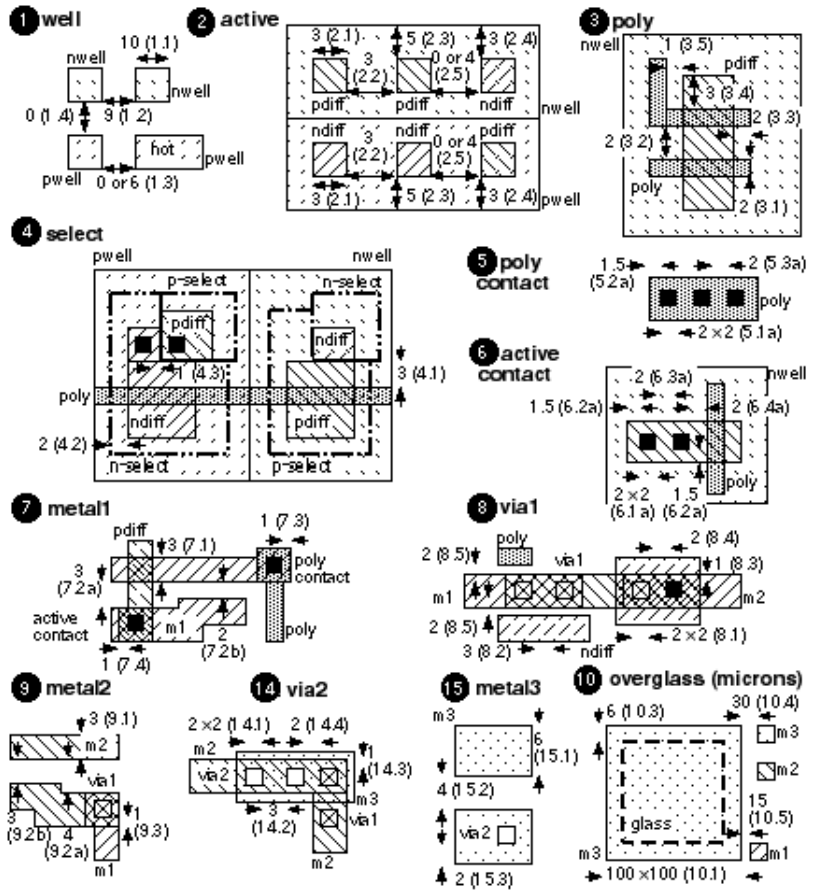


FIGURE 2.11 The MOSIS scalable CMOS design rules (rev. 7). Dimensions are in l . Rule numbers are in parentheses (missing rule sets 11-13 are extensions to this basic process).

TABLE 2.7 MOSIS scalable CMOS rules version 7-the process front end.

Layer	Rule	Explanation	Value / l
well (CWN, CWP)	1.1	minimum width	10
	1.2	minimum space (different potential, a hot well)	9
	1.3	minimum space (same potential)	0 or 6
	1.4	minimum space (different well type)	0
active (CAA)	2.1/2.2	minimum width/space	3
	2.3	source/drain active to well edge space	5
	2.4	substrate/well contact active to well edge space	3
	2.5	minimum space between active (different implant type)	0 or 4
poly (CPG)	3.1/3.2	minimum width/space	2
	3.3	minimum gate extension of active	2

	3.4	minimum active extension of poly	3
	3.5	minimum field poly to active space	1
select (CSN, CSP)	4.1	minimum select spacing to channel of transistor 1	3
	4.2	minimum select overlap of active	2
	4.3	minimum select overlap of contact	1
	4.4	minimum select width and spacing 2	2
poly contact (CCP)	5.1.a	exact contact size	2 ¥ 2
	5.2.a	minimum poly overlap	1.5
	5.3.a	minimum contact spacing	2
active contact (CCA)	6.1.a	exact contact size	2 ¥ 2
	6.2.a	minimum active overlap	1.5
	6.3.a	minimum contact spacing	2
	6.4.a	minimum space to gate of transistor	2

TABLE 2.8 MOSIS scalable CMOS rules version 7-the process back end.

Layer	Rule	Explanation	Value / l
metal1 (CMF)	7.1	minimum width	3
	7.2.a	minimum space	3
	7.2.b	minimum space (for minimum-width wires only)	2
	7.3	minimum overlap of poly contact	1
	7.4	minimum overlap of active contact	1
via1 (CVA)	8.1	exact size	2 ¥ 2
	8.2	minimum via spacing	3
	8.3	minimum overlap by metal1	1
	8.4	minimum spacing to contact	2
	8.5	minimum spacing to poly or active edge	2
metal2 (CMS)	9.1	minimum width	3
	9.2.a	minimum space	4
	9.2.b	minimum space (for minimum-width wires only)	3
	9.3	minimum overlap of via1	1
via2 (CVS)	14.1	exact size	2 ¥ 2
	14.2	minimum space	3
	14.3	minimum overlap by metal2	1
	14.4	minimum spacing to via1	2
metal3 (CMT)	15.1	minimum width	6

	15.2	minimum space	4
	15.3	minimum overlap of via2	2

TABLE 2.9 MOSIS scalable CMOS rules version 7-the pads and overglass (passivation).

Layer	Rule	Explanation	Value
glass (COG)	10.1	minimum bonding-pad width	100 m m ¥ 100 m m
	10.2	minimum probe-pad width	75 m m ¥ 75 m m
	10.3	pad overlap of glass opening	6 m m
	10.4	minimum pad spacing to unrelated metal2 (or metal3)	30 m m
	10.5	minimum pad spacing to unrelated metal1, poly, or active	15 m m

The rules in Table 2.7 and Table 2.8 are given as multiples of l . If we use lambda-based rules we can move between successive process generations just by changing the value of l . For example, we can scale 0.5 m m layouts (l = 0.25 m m) by a factor of 0.175 / 0.25 for a 0.35 m m process (l = 0.175 m m)-at least in theory. You may get an inkling of the practical problems from the fact that the values for pad dimensions and spacing in Table 2.9 are given in microns and not in l . This is because bonding to the pads is an operation that does not scale well. Often companies have two sets of design rules: one in l (with fractional l rules) and the other in microns. Ideally we would like to express all of the design rules in integer multiples of l . This was true for revisions 4-6, but not revision 7 of the MOSIS rules. In revision 7 rules 5.2a/6.2a are noninteger. The original Mead-Conway NMOS rules include a noninteger 1.5 l rule for the implant layer.

---

1. To ensure source and drain width.

2. Different select types may touch but not overlap.

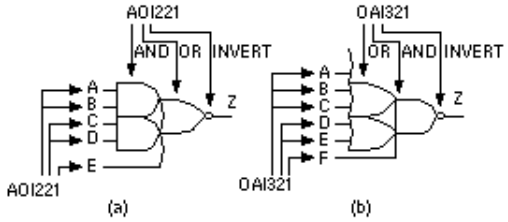
2.4 Combinational Logic Cells

The AND-OR-INVERT (AOI) and the OR-AND-INVERT (OAI) logic cells are particularly efficient in CMOS. Figure 2.12 shows an AOI221 and an OAI321 logic cell (the logic symbols in Figure 2.12 are not standards, but are widely used). All indices (the indices are the numbers after AOI or OAI) in the logic cell name greater than 1 correspond to the inputs to the first "level" or stage-the AND gate(s) in an AOI cell, for example. An index of ’1’ corresponds to a direct input to the second-stage cell. We write indices in descending order; so it is AOI221 and not AOI122 (but both are equivalent cells), and AOI32 not AOI23. If we have more than one direct input to the second stage we repeat the ’1’; thus an AOI211 cell performs the function Z = (A.B + C + D)’. A three-input NAND cell is an OAI111, but calling it that would be very confusing. These rules are not standard, but form a convention that we shall adopt and one that is widely used in the ASIC industry.

There are many ways to represent the logical operator, AND. I shall use the middle dot and write A · B (rather than AB, A.B, or A ? B); occasionally I may use AND(A, B). Similarly I shall write A + B as well as OR(A, B). I shall use an apostrophe like this, A’, to denote the complement of A rather than A since sometimes it is difficult or inappropriate to use an overbar (vinculum) or diacritical mark

(macron). It is possible to misinterpret AB’ as A B rather than AB (but the former alternative would be A · B’ in my convention). I shall be careful in these situations.

FIGURE 2.12 Naming and numbering complex CMOS combinational cells. (a) An AND-OR-INVERT cell, an AOI221. (b) An OR-AND-INVERT cell, an OAI321. Numbering is always in descending order.



We can express the function of the AOI221 cell in Figure 2.12(a) as

Z = (A · B + C · D + E)’ . (2.25)

We can also write this equation unambiguously as Z = OAI221(A, B, C, D, E), just as we might write X = NAND (I, J, K) to describe the logic function X = (I · J · K)’.

This notation is useful because, for example, if we write OAI321(P, Q, R, S, T, U) we immediately know that U (the sixth input) is the (only) direct input connected to the second stage. Sometimes we need to refer to particular inputs without listing them all. We can adopt another convention that letters of the input names change with the index position. Now we can refer to input B2 of an AOI321 cell, for example, and know which input we are talking about without writing

Z = AOI321(A1, A2, A3, B1, B2, C) . (2.26)

Table 2.10 shows the AOI family of logic cells with three indices (with branches in the family for AOI, OAI, AO, and OA cells). There are 5 types and 14 separate members of each branch of this family. There are thus 4 ¥ 14 = 56 cells of the type X abc where X = {OAI, AOI, OA, AO} and each of the indexes a , b , and c can range from 1 to 3. We form the AND-OR (AO) and OR-AND (OA) cells by adding an inverter to the output of an AOI or OAI cell.

TABLE 2.10 The AOI family of cells with three index numbers or less.

Cell type 1	Cells	Number of unique cells
Xa1	X21, X31	2
Xa11	X211, X311	2
Xab	X22, X33, X32	3
Xab1	X221, X331, X321	3
Xabc	X222, X333, X332, X322	4
Total		14

2.4.1 Pushing Bubbles

The AOI and OAI logic cells can be built using a single stage in CMOS using series-parallel networks of transistors called stacks. Figure 2.13 illustrates the procedure to build the n -channel and p -channel stacks, using the AOI221 cell as an example.

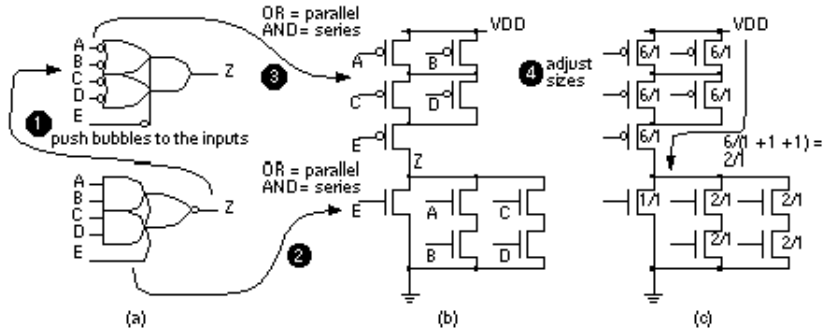


FIGURE 2.13 Constructing a CMOS logic cell-an AOI221. (a) First build the dual icon by using de Morgan's theorem to "push" inversion bubbles to the inputs. (b) Next build the n -channel and p -channel stacks from series and parallel combinations of transistors. (c) Adjust transistor sizes so that the n- channel and p -channel stacks have equal strengths.

Here are the steps to construct any single-stage combinational CMOS logic cell:

1. Draw a schematic icon with an inversion (bubble) on the last cell (the bubble-out schematic). Use de Morgan's theorems -"A NAND is an OR with inverted inputs and a NOR is an AND with inverted inputs"-to push the output bubble back to the inputs (this the dual icon or bubble-in schematic).
2. Form the n -channel stack working from the inputs on the bubble-out schematic: OR translates to a parallel connection, AND translates to a series connection. If you have a bubble at an input, you need an inverter.
3. Form the p -channel stack using the bubble-in schematic (ignore the inversions at the inputs-the bubbles on the gate terminals of the p -channel transistors take care of these). If you do not have a bubble at the input gate terminals, you need an inverter (these will be the same input gate terminals that had bubbles in the bubble-out schematic).

The two stacks are network duals (they can be derived from each other by swapping series connections for parallel, and parallel for series connections). The n -channel stack implements the strong '0's of the function and the p -channel stack provides the strong '1's. The final step is to adjust the drive strength of the logic cell by sizing the transistors.

2.4.2 Drive Strength

Normally we ratio the sizes of the n -channel and p -channel transistors in an inverter so that both types of transistors have the same resistance, or drive strength . That is, we make $b_n = b_p$. At low dopant

concentrations and low electric fields m_n is about twice m_p. To compensate we make the shape factor, W/L, of the p -channel transistor in an inverter about twice that of the n -channel transistor (we say the logic has a ratio of 2). Since the transistor lengths are normally equal to the minimum poly width for both types of transistors, the ratio of the transistor widths is also equal to 2. With the high dopant concentrations and high electric fields in submicron transistors the difference in mobilities is less-typically between 1 and 1.5.

Logic cells in a library have a range of drive strengths. We normally call the minimum-size inverter a 1X inverter. The drive strength of a logic cell is often used as a suffix; thus a 1X inverter has a cell name such as INVX1 or INVD1. An inverter with transistors that are twice the size will be an INVX2. Drive strengths are normally scaled in a geometric ratio, so we have 1X, 2X, 4X, and (sometimes) 8X or even higher, drive-strength cells. We can size a logic cell using these basic rules:

- Any string of transistors connected between a power supply and the output in a cell with 1X drive should have the same resistance as the n -channel transistor in a 1X inverter.
- A transistor with shape factor W_1/L_1 has a resistance proportional to L_1/W_1 (so the larger W_1 is, the smaller the resistance).
- Two transistors in parallel with shape factors W_1/L_1 and W_2/L_2 are equivalent to a single transistor $(W_1/L_1 + W_2/L_2)/1$. For example, a 2/1 in parallel with a 3/1 is a 5/1.
- Two transistors, with shape factors W_1/L_2 and W_2/L_2, in series are equivalent to a single $1/(L_1/W_1 + L_2/W_2)$ transistor.

For example, a transistor with shape factor 3/1 (we shall call this "a 3/1") in series with another 3/1 is equivalent to a 1/((1/3) + (1/3)) or a 3/2. We can use the following method to calculate equivalent transistor sizes:

- To add transistors in parallel, make all the lengths 1 and add the widths.
- To add transistors in series, make all the widths 1 and add the lengths.

We have to be careful to keep W and L reasonable. For example, a 3/1 in series with a 2/1 is equivalent to a 1/((1/3) + (1/2)) or 1/0.83. Since we cannot make a device 2 l wide and 1.66 l long, a 1/0.83 is more naturally written as 3/2.5. We like to keep both W and L as integer multiples of 0.5 (equivalent to making W and L integer multiples of l), but W and L must be greater than 1.

In Figure 2.13(c) the transistors in the AOI221 cell are sized so that any string through the p -channel stack has a drive strength equivalent to a 2/1 p -channel transistor (we choose the worst case, if more than one transistor in parallel is conducting then the drive strength will be higher). The n -channel stack is sized so that it has a drive strength of a 1/1 n -channel transistor. The ratio in this library is thus 2.

If we were to use four drive strengths for each of the AOI family of cells shown in Table 2.10, we would have a total of 224 combinational library cells-just for the AOI family. The synthesis tools can handle this number of cells, but we may not be able to design this many cells in a reasonable amount of time. Section 3.3, "Logical Effort," will help us choose the most logically efficient cells.

2.4.3 Transmission Gates

Figure 2.14(a) and (b) shows a CMOS transmission gate (TG , TX gate, pass gate, coupler). We connect a p -channel transistor (to transmit a strong ’1’) in parallel with an n -channel transistor (to transmit a strong ’0’).

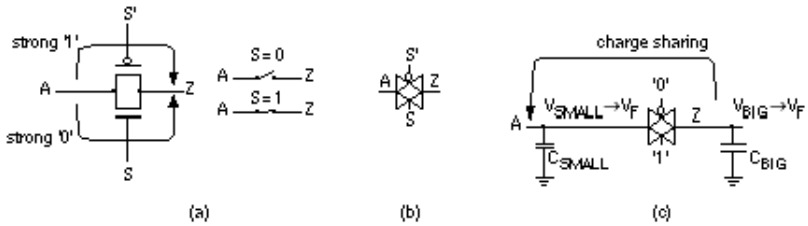


FIGURE 2.14 CMOS transmission gate (TG). (a) An n- channel and p -channel transistor in parallel form a TG. (b) A common symbol for a TG. (c) The charge-sharing problem.

We can express the function of a TG as

Z = TG(A, S) , (2.27)

but this is ambiguous-if we write TG(X, Y), how do we know if X is connected to the gates or sources/drains of the TG? We shall always define TG(X, Y) when we use it. It is tempting to write TG(A, S) = A · S, but what is the value of Z when S =’0’ in Figure 2.14(a), since Z is then left floating? A TG is a switch, not an AND logic cell.

There is a potential problem if we use a TG as a switch connecting a node Z that has a large capacitance, C_{BIG} , to an input node A that has only a small capacitance C_{SMALL} (see Figure 2.14c). If the initial voltage at A is V_{SMALL} and the initial voltage at Z is V_{BIG} , when we close the TG (by setting S = ’1’) the final voltage on both nodes A and Z is

$$V_F = \frac{C_{BIG} V_{BIG} + C_{SMALL} V_{SMALL}}{C_{BIG} + C_{SMALL}} \quad . \ (2.28)$$

Imagine we want to drive a ’0’ onto node Z from node A. Suppose C_{BIG} = 0.2 pF (about 10 standard loads in a 0.5 m m process) and C_{SMALL} = 0.02 pF, V_{BIG} = 0 V and V_{SMALL} = 5 V; then

$$V_F = \frac{(0.2 ¥ 10^{-12}) (0) + (0.02 ¥ 10^{-12}) (5)}{(0.2 ¥ 10^{-12}) + (0.02 ¥ 10^{-12})} = 0.45 \text{ V} \ . \ (2.29)$$

This is not what we want at all, the "big" capacitor has forced node A to a voltage close to a ’0’. This type of problem is known as charge sharing . We should make sure that either (1) node A is strong enough to overcome the big capacitor, or (2) insulate node A from node Z by including a buffer (an

inverter, for example) between node A and node Z. We must not use charge to drive another logic cell-only a logic cell can drive a logic cell.

If we omit one of the transistors in a TG (usually the p -channel transistor) we have a pass transistor . There is a branch of full-custom VLSI design that uses pass-transistor logic. Much of this is based on relay-based logic, since a single transistor switch looks like a relay contact. There are many problems associated with pass-transistor logic related to charge sharing, reduced noise margins, and the difficulty of predicting delays. Though pass transistors may appear in an ASIC cell inside a library, they are not used by ASIC designers.

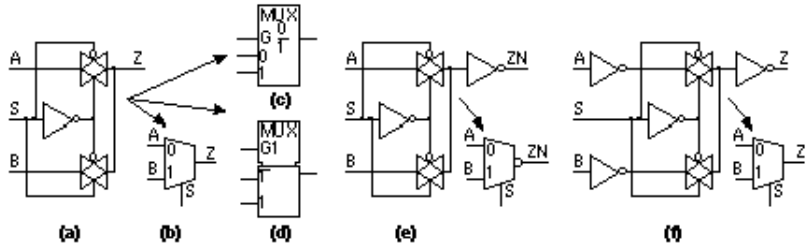


FIGURE 2.15 The CMOS multiplexer (MUX). (a) A noninverting 2:1 MUX using transmission gates without buffering. (b) A symbol for a MUX (note how the inputs are labeled). (c) An IEEE standard symbol for a MUX. (d) A nonstandard, but very common, IEEE symbol for a MUX. (e) An inverting MUX with output buffer. (f) A noninverting buffered MUX.

We can use two TGs to form a multiplexer (or multiplexor-people use both orthographies) as shown in Figure 2.15(a). We often shorten multiplexer to MUX . The MUX function for two data inputs, A and B, with a select signal S, is

$$Z = TG(A, S') + TG(B, S) . \quad (2.30)$$

We can write this as $Z = A \cdot S' + B \cdot S$, since node Z is always connected to one or other of the inputs (and we assume both are driven). This is a two-input MUX (2-to-1 MUX or 2:1 MUX). Unfortunately, we can also write the MUX function as $Z = A \cdot S + B \cdot S'$, so it is difficult to write the MUX function unambiguously as Z = MUX(X, Y, Z). For example, is the select input X, Y, or Z? We shall define the function MUX(X, Y, Z) each time we use it. We must also be careful to label a MUX if we use the symbol shown in Figure 2.15(b). Symbols for a MUX are shown in Figure 2.15(b-d). In the IEEE notation ’G’ specifies an AND dependency. Thus, in Figure 2.15(c), G = ’1’ selects the input labeled ’1’. Figure 2.15(d) uses the common control block symbol (the notched rectangle). Here, G1 = ’1’ selects the input ’1’, and G1 = ’0’ selects the input ’ 1 ’. Strictly this form of IEEE symbol should be used only for elements with more than one section controlled by common signals, but the symbol of Figure 2.15(d) is used often for a 2:1 MUX.

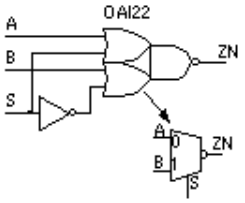
The MUX shown in Figure 2.15(a) works, but there is a potential charge-sharing problem if we cascade MUXes (connect them in series). Instead most ASIC libraries use MUX cells built with a more conservative approach. We could buffer the output using an inverter (Figure 2.15e), but then the MUX becomes inverting. To build a safe, noninverting MUX we can buffer the inputs and output (Figure 2.15f)-requiring 12 transistors, or 3 gate equivalents (only the gate equivalent counts are shown from now on).

Figure 2.16 shows how to use an OAI22 logic cell (and an inverter) to implement an inverting MUX. The implementation in equation form (2.5 gates) is

$$
\begin{aligned}
ZN &= A' \cdot S' + B' \cdot S \\
&= [(A' \cdot S')' \cdot (B' \cdot S)']' \\
&= [\,(A + S) \cdot (B + S')]' \\
&= OAI22[A, S, B, NOT(S)]\ . \quad (2.31)
\end{aligned}
$$

(both A' and NOT(A) represent an inverter, depending on which representation is most convenient-they are equivalent). I often use an equation to describe a cell implementation.

FIGURE 2.16 An inverting 2:1 MUX based on an OAI22 cell.



The following factors will determine which MUX implementation is best:

1. Do we want to minimize the delay between the select input and the output or between the data inputs and the output?
2. Do we want an inverting or noninverting MUX?
3. Do we object to having any logic cell inputs tied directly to the source/drain diffusions of a transmission gate? (Some companies forbid such transmission-gate inputs -since some simulation tools cannot handle them.)
4. Do we object to any logic cell outputs being tied to the source/drain of a transmission gate? (Some companies will not allow this because of the dangers of charge sharing.)
5. What drive strength do we require (and is size or speed more important)?

A minimum-size TG is a little slower than a minimum-size inverter, so there is not much difference between the implementations shown in Figure 2.15 and Figure 2.16, but the difference can become important for 4:1 and larger MUXes.

2.4.4 Exclusive-OR Cell

The two-input exclusive-OR (XOR , EXOR, not-equivalence, ring-OR) function is

$$A1 \ ?\ A2 = XOR(A1, A2) = A1 \cdot A2' + A1' \cdot A2\ . \quad (2.32)$$

We are now using multiletter symbols, but there should be no doubt that A1' means anything other than NOT(A1). We can implement a two-input XOR using a MUX and an inverter as follows (2 gates):

$$XOR(A1, A2) = MUX[NOT(A1), A1, A2]\ , \quad (2.33)$$

where

$$MUX(A, B, S) = A \cdot S + B \cdot S' \quad (2.34)$$

This implementation only buffers one input and does not buffer the MUX output. We can use inverter buffers (3.5 gates total) or an inverting MUX so that the XOR cell does not have any external connections to source/drain diffusions as follows (3 gates total):

$$XOR(A1, A2) = NOT[MUX(NOT[NOT(A1)], NOT(A1), A2)] \quad (2.35)$$

We can also implement a two-input XOR using an AOI21 (and a NOR cell), since

$$\begin{aligned} XOR(A1, A2) &= A1 \cdot A2' + A1' \cdot A2 \\ &= [\,(A1 \cdot A2) + (A1 + A2)'\,]' \\ &= AOI21[A1, A2, NOR(A1, A2)], \quad (2.36) \end{aligned}$$

(2.5 gates). Similarly we can implement an exclusive-NOR (XNOR, equivalence) logic cell using an inverting MUX (and two inverters, total 3.5 gates) or an OAI21 logic cell (and a NAND cell, total 2.5 gates) as follows (using the MUX function of Eq. 2.34):

$$\begin{aligned} XNOR(A1, A2) &= A1 \cdot A2 + NOT(A1) \cdot NOT(A2 \\ &= NOT[NOT[MUX(A1, NOT\ (A1), A2]] \\ &= OAI21[A1, A2, NAND(A1, A2)] . \quad (2.37) \end{aligned}$$

---

1. Xabc: X = {AOI, AO, OAI, OA}; a, b, c = {2, 3}; { } means "choose one."

2.5 Sequential Logic Cells

There are two main approaches to clocking in VLSI design: multiphase clocks or a single clock and synchronous design . The second approach has the following key advantages: (1) it allows automated design, (2) it is safe, and (3) it permits vendor signoff (a guarantee that the ASIC will work as simulated). These advantages of synchronous design (especially the last one) usually outweigh every other consideration in the choice of a clocking scheme. The vast majority of ASICs use a rigid synchronous design style.

2.5.1 Latch

Figure 2.17(a) shows a sequential logic cell-a latch . The internal clock signals, CLKN (N for negative) and CLKP (P for positive), are generated from the system clock, CLK, by two inverters (I4 and I5) that are part of every latch cell-it is usually too dangerous to have these signals supplied externally, even though it would save space.

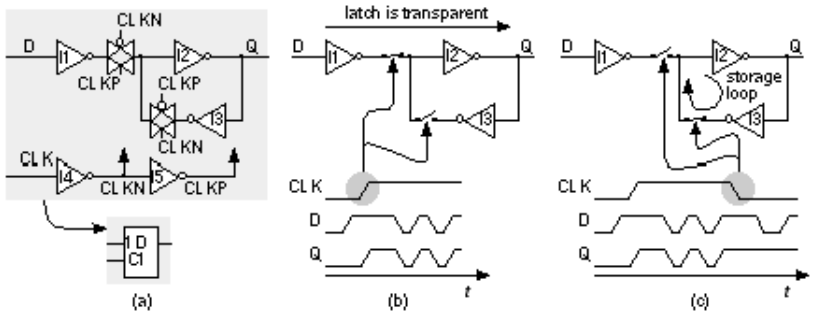


FIGURE 2.17 CMOS latch. (a) A positive-enable latch using transmission gates without output buffering, the enable (clock) signal is buffered inside the latch. (b) A positive-enable latch is transparent while the enable is high. (c) The latch stores the last value at D when the enable goes low.

To emphasize the difference between a latch and flip-flop, sometimes people refer to the clock input of a latch as an enable . This makes sense when we look at Figure 2.17(b), which shows the operation of a latch. When the clock input is high, the latch is transparent -changes at the D input appear at the output Q (quite different from a flip-flop as we shall see). When the enable (clock) goes low (Figure 2.17c), inverters I2 and I3 are connected together, forming a storage loop that holds the last value on D until the enable goes high again. The storage loop will hold its state as long as power is on; we call this a static latch. A sequential logic cell is different from a combinational cell because it has this feature of storage or memory.

Notice that the output Q is unbuffered and connected directly to the output of I2 (and the input of I3), which is a storage node. In an ASIC library we are conservative and add an inverter to buffer the output, isolate the sensitive storage node, and thus invert the sense of Q. If we want both Q and QN we have to add two inverters to the circuit of Figure 2.17(a). This means that a latch requires seven inverters and two TGs (4.5 gates).

The latch of Figure 2.17(a) is a positive-enable D latch, active-high D latch, or transparent-high D latch (sometimes people also call this a D-type latch). A negative-enable (active-low) D latch can be built by inverting all the clock polarities in Figure 2.17(a) (swap CLKN for CLKP and vice-versa).

2.5.2 Flip-Flop

Figure 2.18(a) shows a flip-flop constructed from two D latches: a master latch (the first one) and a slave latch . This flip-flop contains a total of nine inverters and four TGs, or 6.5 gates. In this flip-flop design the storage node S is buffered and the clock-to-Q delay will be one inverter delay less than the clock-to-QN delay.

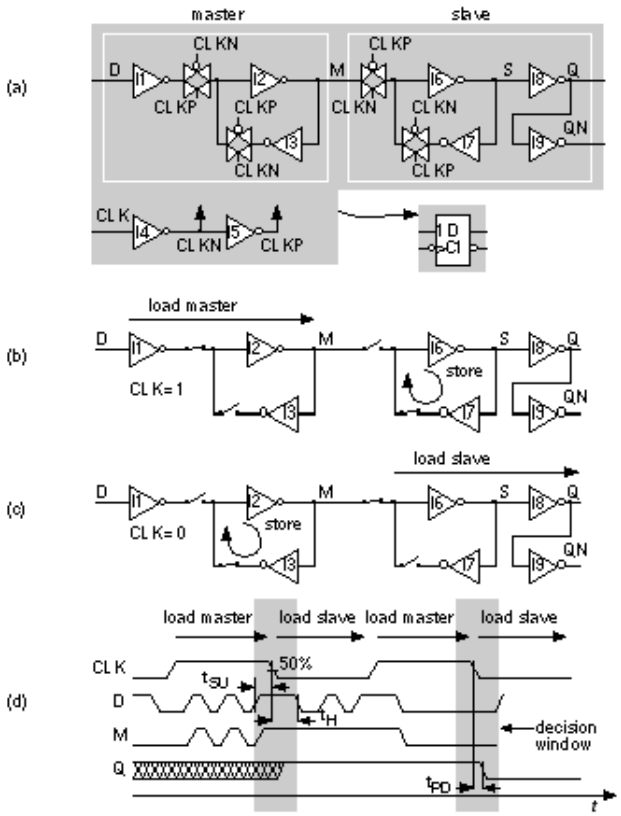


FIGURE 2.18 CMOS flip-flop. (a) This negative-edge-triggered flip-flop consists of two latches: master and slave. (b) While the clock is high, the master latch is loaded. (c) As the clock goes low, the slave latch loads the value of the master latch. (d) Waveforms illustrating the definition of the flip-flop setup time t_{SU}, hold time t_H, and propagation delay from clock to Q, t_{PD}.

In Figure 2.18(b) the clock input is high, the master latch is transparent, and node M (for master) will follow the D input. Meanwhile the slave latch is disconnected from the master latch and is storing whatever the previous value of Q was. As the clock goes low (the negative edge) the slave latch is enabled and will update its state (and the output Q) to the value of node M at the negative edge of the clock. The slave latch will then keep this value of M at the output Q, despite any changes at the D input while the clock is low (Figure 2.18c). When the clock goes high again, the slave latch will store the captured value of M (and we are back where we started our explanation).

The combination of the master and slave latches acts to capture or sample the D input at the negative clock edge, the active clock edge . This type of flip-flop is a negative-edge-triggered flip-flop and its behavior is quite different from a latch. The behavior is shown on the IEEE symbol by using a triangular "notch" to denote an edge-sensitive input. A bubble shows the input is sensitive to the negative edge. To build a positive-edge-triggered flip-flop we invert the polarity of all the clocks-as we did for a latch.

The waveforms in Figure 2.18(d) show the operation of the flip-flop as we have described it, and illustrate the definition of setup time (t_{SU}), hold time (t_H), and clock-to-Q propagation delay (t_{PD}). We must keep the data stable (a fixed logic ’1’ or ’0’) for a time t_{SU} prior to the active clock edge, and stable for a time t after the active clock edge (during the decision window shown).

stable for a time t_H after the active clock edge (during the decision window shown).

In Figure 2.18(d) times are measured from the points at which the waveforms cross 50 percent of V_{DD}. We say the trip point is 50 percent or 0.5. Common choices are 0.5 or 0.65/0.35 (a signal has to reach 0.65 V_{DD} to be a '1', and reach 0.35 V_{DD} to be a '0'), or 0.1/0.9 (there is no standard way to write a trip point). Some vendors use different trip points for the input and output waveforms (especially in I/O cells).

The flip-flop in Figure 2.18(a) is a D flip-flop and is by far the most widely used type of flip-flop in ASIC design. There are other types of flip-flops-J-K, T (toggle), and S-R flip-flops-that are provided in some ASIC cell libraries mainly for compatibility with TTL design. Some people use the term register to mean an array (more than one) of flip-flops or latches (on a data bus, for example), but some people use register to mean a single flip-flop or a latch. This is confusing since flip-flops and latches are quite different in their behavior. When I am talking about logic cells, I use the term register to mean more than one flip-flop.

To add an asynchronous set (Q to '1') or asynchronous reset (Q to '0') to the flip-flop of Figure 2.18(a), we replace one inverter in both the master and slave latches with two-input NAND cells. Thus, for an active-low set, we replace I2 and I7 with two-input NAND cells, and, for an active-low reset, we replace I3 and I6. For both set and reset we replace all four inverters: I2, I3, I6, and I7. Some TTL flip-flops have dominant reset or dominant set , but this is difficult (and dangerous) to do in ASIC design. An input that forces Q to '1' is sometimes also called preset . The IEEE logic symbols use 'P' to denote an input with a presetting action. An input that forces Q to '0' is often also called clear . The IEEE symbols use 'R' to denote an input with a resetting action.

2.5.3 Clocked Inverter

Figure 2.19 shows how we can derive the structure of a clocked inverter from the series combination of an inverter and a TG. The arrows in Figure 2.19(b) represent the flow of current when the inverter is charging (I_R) or discharging (I_F) a load capacitance through the TG. We can break the connection between the inverter cells and use the circuit of Figure 2.19(c) without substantially affecting the operation of the circuit. The symbol for the clocked inverter shown in Figure 2.19(d) is common, but by no means a standard.

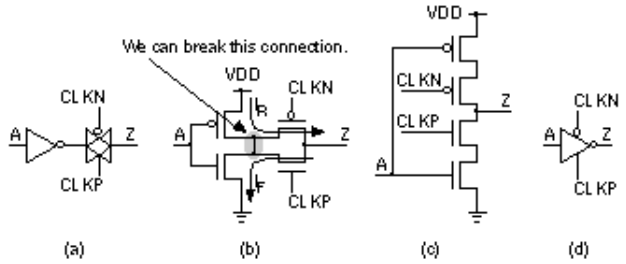


FIGURE 2.19 Clocked inverter. (a) An inverter plus transmission gate (TG). (b) The current flow in

the inverter and TG allows us to break the connection between the transistors in the inverter.
(c) Breaking the connection forms a clocked inverter. (d) A common symbol.

We can use the clocked inverter to replace the inverter-TG pairs in latches and flip-flops. For example, we can replace one or both of the inverters I1 and I3 (together with the TGs that follow them) in Figure 2.17(a) by clocked inverters. There is not much to choose between the different implementations in this case, except that layout may be easier for the clocked inverter versions (since there is one less connection to make).

More interesting is the flip-flop design: We can only replace inverters I1, I3, and I7 (and the TGs that follow them) in Figure 2.18(a) by clocked inverters. We cannot replace inverter I6 because it is not directly connected to a TG. We can replace the TG attached to node M with a clocked inverter, and this will invert the sense of the output Q, which thus becomes QN. Now the clock-to-Q delay will be slower than clock-to-QN, since Q (which was QN) now comes one inverter later than QN.

If we wish to build a flip-flop with a fast clock-to-QN delay it may be better to build it using clocked inverters and use inverters with TGs for a flip-flop with a fast clock-to-Q delay. In fact, since we do not always use both Q and QN outputs of a flip-flop, some libraries include Q only or QN only flip-flops that are slightly smaller than those with both polarity outputs. It is slightly easier to layout clocked inverters than an inverter plus a TG, so flip-flops in commercial libraries include a mixture of clocked-inverter and TG implementations.

2.6 Datapath Logic Cells

Suppose we wish to build an n -bit adder (that adds two n -bit numbers) and to exploit the regularity of this function in the layout. We can do so using a datapath structure.

The following two functions, SUM and COUT, implement the sum and carry out for a full adder (FA) with two data inputs (A, B) and a carry in, CIN:

$$SUM = A ? B ? CIN = SUM(A, B, CIN) = PARITY(A, B, CIN) , \quad (2.38)$$

$$COUT = A \cdot B + A \cdot CIN + B \cdot CIN = MAJ(A, B, CIN). \quad (2.39)$$

The sum uses the parity function (’1’ if there are an odd numbers of ’1’s in the inputs). The carry out, COUT, uses the 2-of-3 majority function (’1’ if the majority of the inputs are ’1’). We can combine these two functions in a single FA logic cell, ADD(A[i], B[i], CIN, S[i], COUT), shown in Figure 2.20(a), where

$$S[i] = SUM (A[i], B[i], CIN) , \quad (2.40)$$

$$COUT = MAJ (A[i], B[i], CIN) . \quad (2.41)$$

Now we can build a 4-bit ripple-carry adder (RCA) by connecting four of these ADD cells together as shown in Figure 2.20(b). The i th ADD cell is arranged with the following: two bus inputs A[i], B[i];

one bus output S[i]; an input, CIN, that is the carry in from stage (i - 1) below and is also passed up to the cell above as an output; and an output, COUT, that is the carry out to stage (i + 1) above. In the 4-bit adder shown in Figure 2.20(b) we connect the carry input, CIN[0], to VSS and use COUT[3] and COUT[2] to indicate arithmetic overflow (in Section 2.6.1 we shall see why we may need both signals). Notice that we build the ADD cell so that COUT[2] is available at the top of the datapath when we need it.

Figure 2.20(c) shows a layout of the ADD cell. The A inputs, B inputs, and S outputs all use m1 interconnect running in the horizontal direction-we call these data signals. Other signals can enter or exit from the top or bottom and run vertically across the datapath in m2-we call these control signals. We can also use m1 for control and m2 for data, but we normally do not mix these approaches in the same structure. Control signals are typically clocks and other signals common to elements. For example, in Figure 2.20(c) the carry signals, CIN and COUT, run vertically in m2 between cells. To build a 4-bit adder we stack four ADD cells creating the array structure shown in Figure 2.20(d). In this case the A and B data bus inputs enter from the left and bus S, the sum, exits at the right, but we can connect A, B, and S to either side if we want.

The layout of buswide logic that operates on data signals in this fashion is called a datapath . The module ADD is a datapath cell or datapath element . Just as we do for standard cells we make all the datapath cells in a library the same height so we can abut other datapath cells on either side of the adder to create a more complex datapath. When people talk about a datapath they always assume that it is oriented so that increasing the size in bits makes the datapath grow in height, upwards in the vertical direction, and adding different datapath elements to increase the function makes the datapath grow in width, in the horizontal direction-but we can rotate and position a completed datapath in any direction we want on a chip.

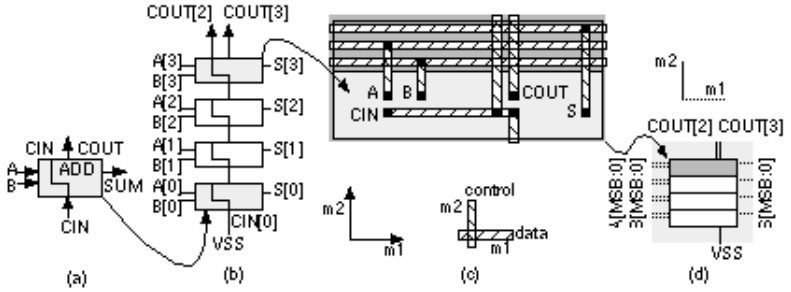


FIGURE 2.20 A datapath adder. (a) A full-adder (FA) cell with inputs (A and B), a carry in, CIN, sum output, S, and carry out, COUT. (b) A 4-bit adder. (c) The layout, using two-level metal, with data in m1 and control in m2. In this example the wiring is completed outside the cell; it is also possible to design the datapath cells to contain the wiring. Using three levels of metal, it is possible to wire over the top of the datapath cells. (d) The datapath layout.

What is the difference between using a datapath, standard cells, or gate arrays? Cells are placed together in rows on a CBIC or an MGA, but there is no generally no regularity to the arrangement of the cells within the rows-we let software arrange the cells and complete the interconnect. Datapath layout automatically takes care of most of the interconnect between the cells with the following advantages:

- Regular layout produces predictable and equal delay for each bit.
- Interconnect between cells can be built into each cell.

There are some disadvantages of using a datapath:

- The overhead (buffering and routing the control signals, for example) can make a narrow (small number of bits) datapath larger and slower than a standard-cell (or even gate-array) implementation.
- Datapath cells have to be predesigned (otherwise we are using full-custom design) for use in a wide range of datapath sizes. Datapath cell design can be harder than designing gate-array macros or standard cells.
- Software to assemble a datapath is more complex and not as widely used as software for assembling standard cells or gate arrays.

There are some newer standard-cell and gate-array tools that can take advantage of regularity in a design and position cells carefully. The problem is in finding the regularity if it is not specified. Using a datapath is one way to specify regularity to ASIC design tools.

2.6.1 Datapath Elements

Figure 2.21 shows some typical datapath symbols for an adder (people rarely use the IEEE standards in ASIC datapath libraries). I use heavy lines (they are 1.5 point wide) with a stroke to denote a data bus (that flows in the horizontal direction in a datapath), and regular lines (0.5 point) to denote the control signals (that flow vertically in a datapath). At the risk of adding confusion where there is none, this stroke to indicate a data bus has nothing to do with mixed-logic conventions. For a bus, A[31:0] denotes a 32-bit bus with A[31] as the leftmost or most-significant bit or MSB , and A[0] as the least-significant bit or LSB . Sometimes we shall use A[MSB] or A[LSB] to refer to these bits. Notice that if we have an n -bit bus and LSB = 0, then MSB = n - 1. Also, for example, A[4] is the fifth bit on the bus (from the LSB). We use a ’ S ’ or ’ADD’ inside the symbol to denote an adder instead of ’+’, so we can attach ’-’ or ’+/-’ to the inputs for a subtracter or adder/subtracter.

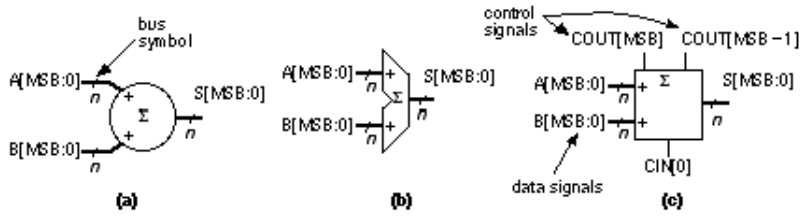


FIGURE 2.21 Symbols for a datapath adder. (a) A data bus is shown by a heavy line (1.5 point) and a bus symbol. If the bus is n -bits wide then MSB = n - 1. (b) An alternative symbol for an adder. (c) Control signals are shown as lightweight (0.5 point) lines.

Some schematic datapath symbols include only data signals and omit the control signals-but we must not forget them. In Figure 2.21, for example, we may need to explicitly tie CIN[0] to VSS and use

COUT[MSB] and COUT[MSB - 1] to detect overflow. Why might we need both of these control signals? Table 2.11 shows the process of simple arithmetic for the different binary number representations, including unsigned, signed magnitude, ones' complement, and two's complement.

TABLE 2.11 Binary arithmetic.

Operation	Binary Number Representation			
	Unsigned	Signed magnitude	Ones' complement	Two's complement
	no change	**if** positive **then** MSB = 0 **else** MSB = 1	**if** negative **then** flip bits	**if** negative **then** {flip bits; add 1}
3 =	0011	0011	0011	0011
-3 =	NA	1011	1100	1101
zero =	0000	0000 or 1000	1111 or 0000	0000
max. positive =	1111 = 15	0111 = 7	0111 = 7	0111 = 7
max. negative =	0000= 0	1111 = -7	1000 = -7	1000 = -8
addition =				
S = A + B = addend + augend	S = A + B	**if** SG(A) = SG(B) **then** S = A + B **else** { **if** B < A **then** S = A - B **else** S = B - A}	S = A + B + COUT[MSB] COUT is carry out	S = A + B
SG(A) = sign of A				
addition result:				
OV = overflow,	OR = COUT[MSB]	**if** SG(A) = SG(B) **then** OV = COUT[MSB] **else** OV = 0 (impossible)	OV =	OV =
OR = out of range	COUT is carry out		XOR(COUT[MSB], COUT[MSB-1])	XOR(COUT[MSB], COUT[MSB - 1])
SG(S) = sign of S	NA	**if** SG(A) = SG(B) **then** SG(S) = SG(A) **else** { **if** B < A	NA	NA

S = A + B	NA	**then** SG(S) = SG(A) **else** SG(S) = SG(B)}	NA	NA
subtraction = D = A - B = minuend - subtrahend	D = A - B	SG(B) = NOT(SG(B)); D = A + B	Z = -B (negate); D = A + Z	Z = -B (negate); D = A + Z
subtraction result : OV = overflow, OR = out of range	OR = BOUT[MSB] BOUT is borrow out	as in addition	as in addition	as in addition
negation : Z = -A (negate)	NA	Z = A; SG(Z) = NOT(SG(A))	Z = NOT(A)	Z = NOT(A) + 1

2.6.2 Adders

We can view addition in terms of generate , G[i], and propagate , P[i], signals.

method 1	method 2	
G[i] = A[i] · B[i]	G[i] = A[i] · B[i]	(2.42)
P[i] = A[i] ? B[i	P[i] = A[i] + B[i]	(2.43)
C[i] = G[i] + P[i] · C[i -1]	C[i] = G[i] + P[i] · C[i -1]	(2.44)
S[i] = P[i] ? C[i -1]	S[i] = A[i] ? B[i] ? C[i -1]	(2.45)

where C[i] is the carry-out signal from stage i , equal to the carry in of stage (i + 1). Thus, C[i] = COUT[i] = CIN[i + 1]. We need to be careful because C[0] might represent either the carry in or the carry out of the LSB stage. For an adder we set the carry in to the first stage (stage zero), C[-1] or CIN[0], to ’0’. Some people use delete (D) or kill (K) in various ways for the complements of G[i] and P[i], but unfortunately others use C for COUT and D for CIN-so I avoid using any of these. Do not confuse the two different methods (both of which are used) in Eqs. 2.42-2.45 when forming the sum, since the propagate signal, P[i] , is different for each method.

Figure 2.22(a) shows a conventional RCA. The delay of an n -bit RCA is proportional to n and is limited by the propagation of the carry signal through all of the stages. We can reduce delay by using pairs of

"go-faster" bubbles to change AND and OR gates to fast two-input NAND gates as shown in Figure 2.22(a). Alternatively, we can write the equations for the carry signal in two different ways:

either $C[i] = A[i] \cdot B[i] + P[i] \cdot C[i-1]$ (2.46)

or $C[i] = (A[i] + B[i]) \cdot (P[i]' + C[i-1])$, (2.47)

where $P[i]' = NOT(P[i])$. Equations 2.46 and 2.47 allow us to build the carry chain from two-input NAND gates, one per cell, using different logic in even and odd stages (Figure 2.22b):

even stages	odd stages	
$C1[i]' = P[i] \cdot C3[i-1] \cdot C4[i-1]$	$C3[i]' = P[i] \cdot C1[i-1] \cdot C2[i-1]$	(2.48)
$C2[i] = A[i] + B[i]$	$C4[i]' = A[i] \cdot B[i]$	(2.49)
$C[i] = C1[i] \cdot C2[i]$	$C[i] = C3[i]' + C4[i]'$	(2.50)

(the carry inputs to stage zero are $C3[-1] = C4[-1] = '0'$). We can use the RCA of Figure 2.22(b) in a datapath, with standard cells, or on a gate array.

Instead of propagating the carries through each stage of an RCA, Figure 2.23 shows a different approach. A carry-save adder (CSA) cell CSA(A1[i], A2[i], A3[i], CIN, S1[i], S2[i], COUT) has three outputs:

$S1[i] = CIN$, (2.51)

$S2[i] = A1[i] ? A2[i] ? A3[i] = PARITY(A1[i], A2[i], A3[i])$, (2.52)

$COUT = A1[i] \cdot A2[i] + [(A1[i] + A2[i]) \cdot A3[i]] = MAJ(A1[i], A2[i], A3[i])$. (2.53)

The inputs, A1, A2, and A3; and outputs, S1 and S2, are buses. The input, CIN, is the carry from stage (i - 1). The carry in, CIN, is connected directly to the output bus S1-indicated by the schematic symbol (Figure 2.23a). We connect CIN[0] to VSS. The output, COUT, is the carry out to stage (i + 1).

A 4-bit CSA is shown in Figure 2.23(b). The arithmetic overflow signal for ones' complement or two's complement arithmetic, OV, is XOR(COUT[MSB], COUT[MSB - 1]) as shown in Figure 2.23(c). In a CSA the carries are "saved" at each stage and shifted left onto the bus S1. There is thus no carry propagation and the delay of a CSA is constant. At the output of a CSA we still need to add the S1 bus (all the saved carries) and the S2 bus (all the sums) to get an n -bit result using a final stage that is not shown in Figure 2.23(c). We might regard the n -bit sum as being encoded in the two buses, S1 and S2, in the form of the parity and majority functions.

We can use a CSA to add multiple inputs-as an example, an adder with four 4-bit inputs is shown in Figure 2.23(d). The last stage sums two input buses using a carry-propagate adder (CPA). We have used an RCA as the CPA in Figure 2.23(d) and (e), but we can use any type of adder. Notice in Figure 2.23(e) how the two CSA cells and the RCA cell abut together horizontally to form a bit slice (or slice) and then the slices are stacked vertically to form the datapath.

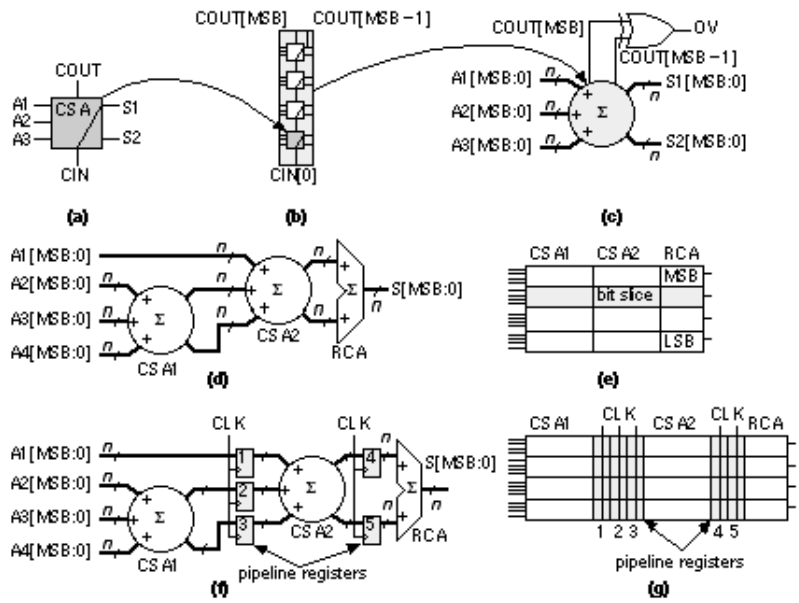


FIGURE 2.22 The carry-save adder (CSA). (a) A CSA cell. (b) A 4-bit CSA. (c) Symbol for a CSA. (d) A four-input CSA. (e) The datapath for a four-input, 4-bit adder using CSAs with a ripple-carry adder (RCA) as the final stage. (f) A pipelined adder. (g) The datapath for the pipelined version showing the pipeline registers as well as the clock control lines that use m2.

We can register the CSA stages by adding vectors of flip-flops as shown in Figure 2.23(f). This reduces the adder delay to that of the slowest adder stage, usually the CPA. By using registers between stages of combinational logic we use pipelining to increase the speed and pay a price of increased area (for the registers) and introduce latency . It takes a few clock cycles (the latency, equal to n clock cycles for an n -stage pipeline) to fill the pipeline, but once it is filled, the answers emerge every clock cycle. Ferris wheels work much the same way. When the fair opens it takes a while (latency) to fill the wheel, but once it is full the people can get on and off every few seconds. (We can also pipeline the RCA of Figure 2.20. We add i registers on the A and B inputs before ADD[i] and add (n - i) registers after the output S[i], with a single register before each C[i].)

The problem with an RCA is that every stage has to wait to make its carry decision, C[i], until the previous stage has calculated C[i - 1]. If we examine the propagate signals we can bypass this critical path. Thus, for example, to bypass the carries for bits 4-7 (stages 5-8) of an adder we can compute BYPASS = P[4].P[5].P[6].P[7] and then use a MUX as follows:

$$C[7] = (G[7] + P[7] \cdot C[6]) \cdot BYPASS' + C[3] \cdot BYPASS \quad (2.54)$$

Adders based on this principle are called carry-bypass adders (CBA) [Sato et al., 1992]. Large, custom adders employ Manchester-carry chains to compute the carries and the bypass operation using TGs or just pass transistors [Weste and Eshraghian, 1993, pp. 530-531]. These types of carry chains may be part of a predesigned ASIC adder cell, but are not used by ASIC designers.

Instead of checking the propagate signals we can check the inputs. For example we can compute SKIP = (A[i - 1] ? B[i - 1]) + (A[i] ? B[i]) and then use a 2:1 MUX to select C[i]. Thus,

$$CSKIP[i] = (G[i] + P[i] \cdot C[i-1]) \cdot SKIP' + C[i-2] \cdot SKIP \quad (2.55)$$

This is a carry-skip adder [Keutzer, Malik, and Saldanha, 1991; Lehman, 1961]. Carry-bypass and carry-skip adders may include redundant logic (since the carry is computed in two different ways-we just take the first signal to arrive). We must be careful that the redundant logic is not optimized away during logic synthesis.

If we evaluate Eq. 2.44 recursively for i = 1, we get the following:

$$\begin{aligned} C[1] &= G[1] + P[1] \cdot C[0] \\ &= G[1] + P[1] \cdot (G[0] + P[1] \cdot C[-1]) \\ &= G[1] + P[1] \cdot G[0] . \qquad (2.56) \end{aligned}$$

This result means that we can "look ahead" by two stages and calculate the carry into the third stage (bit 2), which is C[1], using only the first-stage inputs (to calculate G[0]) and the second-stage inputs. This is a carry-lookahead adder (CLA) [MacSorley, 1961]. If we continue expanding Eq. 2.44, we find:

$$C[2] = G[2] + P[2] \cdot G[1] + P[2] \cdot P[1] \cdot G[0] ,$$

$$C[3] = G[3] + P[2] \cdot G[2] + P[2] \cdot P[1] \cdot G[1] + P[3] \cdot P[2] \cdot P[1] \cdot G[0] . \quad (2.57)$$

As we look ahead further these equations become more complex, take longer to calculate, and the logic becomes less regular when implemented using cells with a limited number of inputs. Datapath layout must fit in a bit slice, so the physical and logical structure of each bit must be similar. In a standard cell or gate array we are not so concerned about a regular physical structure, but a regular logical structure simplifies design. The Brent-Kung adder reduces the delay and increases the regularity of the carry-lookahead scheme [Brent and Kung, 1982]. Figure 2.24(a) shows a regular 4-bit CLA, using the carry-lookahead generator cell (CLG) shown in Figure 2.24(b).

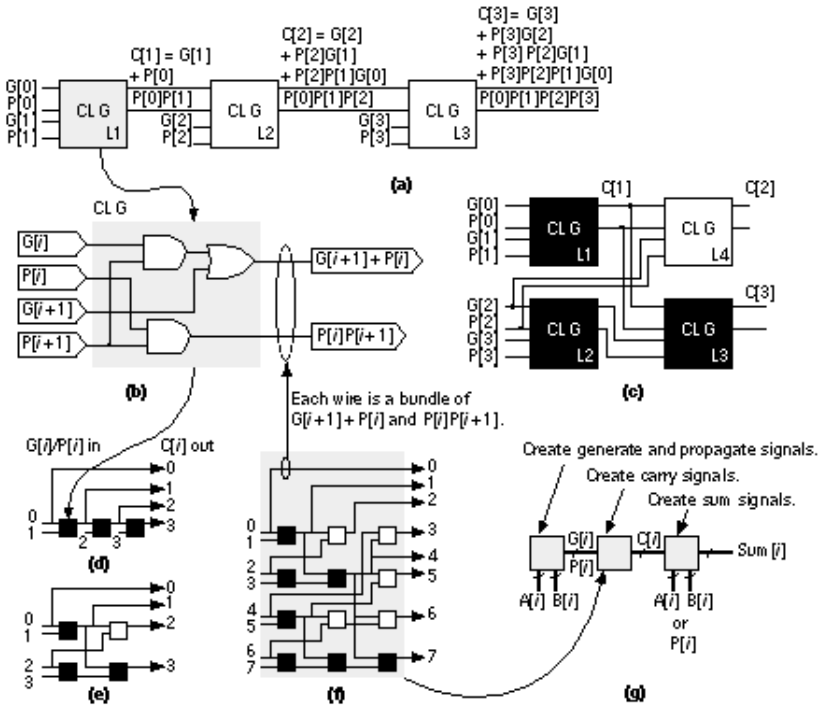


FIGURE 2.23 The Brent-Kung carry-lookahead adder (CLA). (a) Carry generation in a 4-bit CLA. (b) A cell to generate the lookahead terms, C[0]-C[3]. (c) Cells L1, L2, and L3 are rearranged into a tree that has less delay. Cell L4 is added to calculate C[2] that is lost in the translation. (d) and (e) Simplified representations of parts a and c. (f) The lookahead logic for an 8-bit adder. The inputs, 0-7, are the propagate and carry terms formed from the inputs to the adder. (g) An 8-bit Brent-Kung CLA. The outputs of the lookahead logic are the carry bits that (together with the inputs) form the sum. One advantage of this adder is that delays from the inputs to the outputs are more nearly equal than in other adders. This tends to reduce the number of unwanted and unnecessary switching events and thus reduces power dissipation.

In a carry-select adder we duplicate two small adders (usually 4-bit or 8-bit adders-often CLAs) for the cases CIN = ’0’ and CIN = ’1’ and then use a MUX to select the case that we need-wasteful, but fast [Bedrij, 1962]. A carry-select adder is often used as the fast adder in a datapath library because its layout is regular.

We can use the carry-select, carry-bypass, and carry-skip architectures to split a 12-bit adder, for example, into three blocks. The delay of the adder is then partly dependent on the delays of the MUX between each block. Suppose the delay due to 1-bit in an adder block (we shall call this a bit delay) is approximately equal to the MUX delay. In this case may be faster to make the blocks 3, 4, and 5-bits long instead of being equal in size. Now the delays into the final MUX are equal-3 bit-delays plus 2 MUX delays for the carry signal from bits 0-6 and 5 bit-delays for the carry from bits 7-11. Adjusting the block size reduces the delay of large adders (more than 16 bits).

We can extend the idea behind a carry-select adder as follows. Suppose we have an n -bit adder that generates two sums: One sum assumes a carry-in condition of ’0’, the other sum assumes a carry-in condition of ’1’. We can split this n -bit adder into an i -bit adder for the i LSBs and an (n - i)-bit adder

for the n - i MSBs. Both of the smaller adders generate two conditional sums as well as true and complement carry signals. The two (true and complement) carry signals from the LSB adder are used to select between the two (n - i + 1)-bit conditional sums from the MSB adder using 2(n - i + 1) two-input MUXes. This is a conditional-sum adder (also often abbreviated to CSA) [Sklansky, 1960]. We can recursively apply this technique. For example, we can split a 16-bit adder using i = 8 and n = 8; then we can split one or both 8-bit adders again-and so on.

Figure 2.25 shows the simplest form of an n -bit conditional-sum adder that uses n single-bit conditional adders, H (each with four outputs: two conditional sums, true carry, and complement carry), together with a tree of 2:1 MUXes (Qi_j). The conditional-sum adder is usually the fastest of all the adders we have discussed (it is the fastest when logic cell delay increases with the number of inputs-this is true for all ASICs except FPGAs).

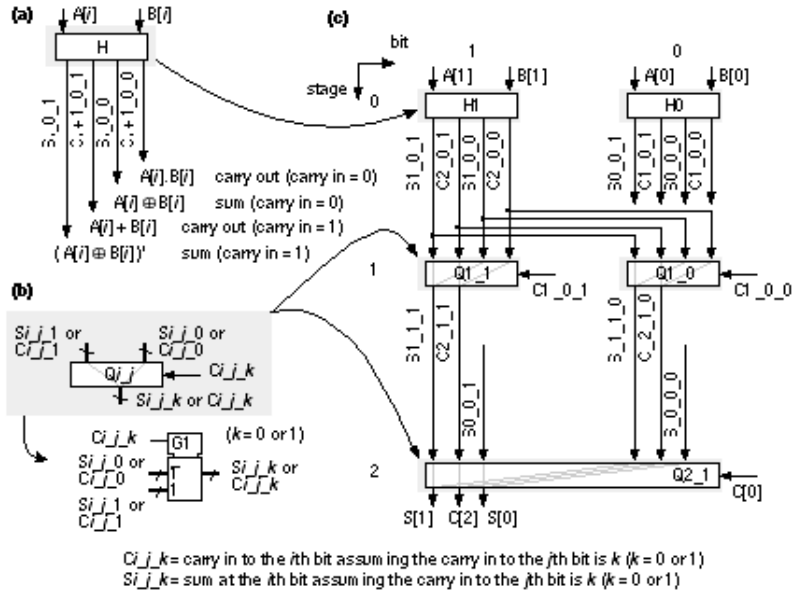


FIGURE 2.24 The conditional-sum adder. (a) A 1-bit conditional adder that calculates the sum and carry out assuming the carry in is either ’1’ or ’0’. (b) The multiplexer that selects between sums and carries. (c) A 4-bit conditional-sum adder with carry input, C[0].

2.6.3 A Simple Example

How do we make and use datapath elements? What does a design look like? We may use predesigned cells from a library or build the elements ourselves from logic cells using a schematic or a design language. Table 2.12 shows an 8-bit conditional-sum adder intended for an FPGA. This Verilog implementation uses the same structure as Figure 2.25, but the equations are collapsed to use four or five variables. A basic logic cell in certain Xilinx FPGAs, for example, can implement two equations of the same four variables or one equation with five variables. The equations shown in Table 2.12 requires three levels of FPGA logic cells (so, for example, if each FPGA logic cell has a 5 ns delay, the 8-bit conditional-sum adder delay is 15 ns).

TABLE 2.12 An 8-bit conditional-sum adder (the notation is described in Figure 2.25).

```
module m8bitCSum (C0, a, b, s, C8); // Verilog conditional-sum adder for an FPGA

input [7:0] C0, a, b; output [7:0] s; output C8;

wire A7,A6,A5,A4,A3,A2,A1,A0,B7,B6,B5,B4,B3,B2,B1,B0,S8,S7,S6,S5,S4,S3,S2,S1,S0;

wire C0, C2, C4_2_0, C4_2_1, S5_4_0, S5_4_1, C6, C6_4_0, C6_4_1, C8;

assign {A7,A6,A5,A4,A3,A2,A1,A0} = a; assign {B7,B6,B5,B4,B3,B2,B1,B0} = b;

assign s = { S7,S6,S5,S4,S3,S2,S1,S0 };

assign S0 = A0^B0^C0 ; // start of level 1: & = AND, ^ = XOR, | = OR, ! = NOT

assign S1 = A1^B1^(A0&B0|(A0|B0)&C0) ;

assign C2 = A1&B1|(A1|B1)&(A0&B0|(A0|B0)&C0) ;

assign C4_2_0 = A3&B3|(A3|B3)&(A2&B2) ; assign C4_2_1 = A3&B3|(A3|B3)&(A2|B2) ;

assign S5_4_0 = A5^B5^(A4&B4) ; assign S5_4_1 = A5^B5^(A4|B4) ;

assign C6_4_0 = A5&B5|(A5|B5)&(A4&B4) ; assign C6_4_1 = A5&B5|(A5|B5)&(A4|B4) ;

assign S2 = A2^B2^C2 ; // start of level 2

assign S3 = A3^B3^(A2&B2|(A2|B2)&C2) ;

assign S4 = A4^B4^(C4_2_0|C4_2_1&C2) ;

assign S5 = S5_4_0& !(C4_2_0|C4_2_1&C2)|S5_4_1&(C4_2_0|C4_2_1&C2) ;

assign C6 = C6_4_0|C6_4_1&(C4_2_0|C4_2_1&C2) ;

assign S6 = A6^B6^C6 ; // start of level 3

assign S7 = A7^B7^(A6&B6|(A6|B6)&C6) ;

assign C8 = A7&B7|(A7|B7s)&(A6&B6|(A6|B6)&C6) ;

endmodule
```

Figure 2.26 shows the normalized delay and area figures for a set of predesigned datapath adders. The data in Figure 2.26 is from a series of ASIC datapath cell libraries (Compass Passport) that may be synthesized together with test vectors and simulation models. We can combine the different adder

techniques, but the adders then lose regularity and become less suited to a datapath implementation.

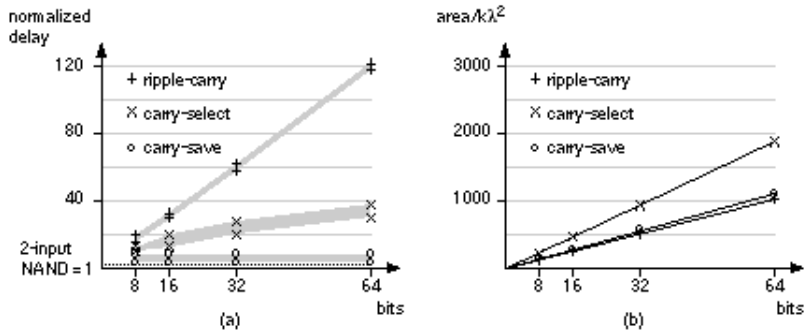


FIGURE 2.25 Datapath adders. This data is from a series of submicron datapath libraries. (a) Delay normalized to a two-input NAND logic cell delay (approximately equal to 250 ps in a 0.5 m m process). For example, a 64-bit ripple-carry adder (RCA) has a delay of approximately 30 ns in a 0.5 m m process. The spread in delay is due to variation in delays between different inputs and outputs. An n -bit RCA has a delay proportional to n . The delay of an n -bit carry-select adder is approximately proportional to $\log_2 n$. The carry-save adder delay is constant (but requires a carry-propagate adder to complete an addition). (b) In a datapath library the area of all adders are proportional to the bit size.

There are other adders that are not used in datapaths, but are occasionally useful in ASIC design. A serial adder is smaller but slower than the parallel adders we have described [Denyer and Renshaw, 1985]. The carry-completion adder is a variable delay adder and rarely used in synchronous designs [Sklansky, 1960].

2.6.4 Multipliers

Figure 2.27 shows a symmetric 6-bit array multiplier (an n -bit multiplier multiplies two n -bit numbers; we shall use n -bit by m -bit multiplier if the lengths are different). Adders a0-f0 may be eliminated, which then eliminates adders a1-a6, leaving an asymmetric CSA array of 30 (5 ¥ 6) adders (including one half adder). An n -bit array multiplier has a delay proportional to n plus the delay of the CPA (adders b6-f6 in Figure 2.27). There are two items we can attack to improve the performance of a multiplier: the number of partial products and the addition of the partial products.

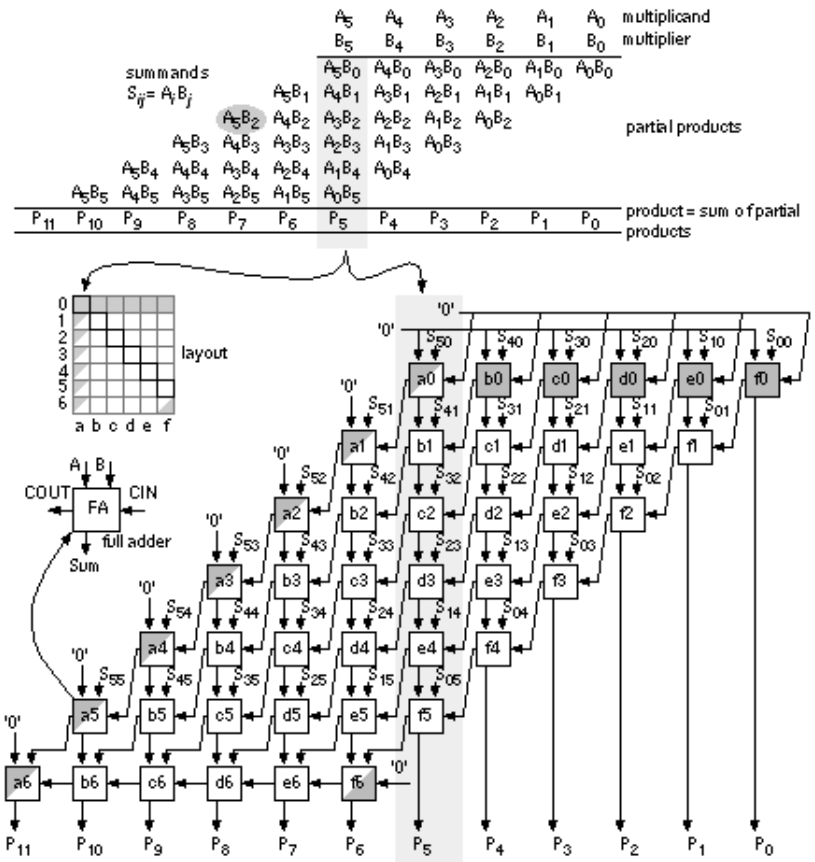


FIGURE 2.26 Multiplication. A 6-bit array multiplier using a final carry-propagate adder (full-adder cells a6-f6, a ripple-carry adder). Apart from the generation of the summands this multiplier uses the same structure as the carry-save adder of Figure 2.23(d).

Suppose we wish to multiply 15 (the multiplicand) by 19 (the multiplier) mentally. It is easier to calculate 15 ¥ 20 and subtract 15. In effect we complete the multiplication as 15 ¥ (20 - 1) and we could write this as 15 ¥ 2 1 , with the overbar representing a minus sign. Now suppose we wish to multiply an 8-bit binary number, A, by B = 00010111 (decimal 16 + 4 + 2 + 1 = 23). It is easier to multiply A by the canonical signed-digit vector (CSD vector) D = 0010 1 001 (decimal 32 - 8 + 1 = 23) since this requires only three add or subtract operations (and a subtraction is as easy as an addition). We say B has a weight of 4 and D has a weight of 3. By using D instead of B we have reduced the number of partial products by 1 (= 4 - 3).

We can recode (or encode) any binary number, B, as a CSD vector, D, as follows (canonical means there is only one CSD vector for any number):

$$D_i = B_i + C_i - 2C_{i+1}, \quad (2.58)$$

where C_{i+1} is the carry from the sum of $B_{i+1} + B_i + C_i$ (we start with $C_0 = 0$).

As another example, if B = 011 ($B_2 = 0$, $B_1 = 1$, $B_0 = 1$; decimal 3), then, using Eq. 2.58,

$$D_0 = B_0 + C_0 - 2C_1 = 1 + 0 - 2 = 1,$$
$$D_1 = B_1 + C_1 - 2C_2 = 1 + 1 - 2 = 0,$$
$$D_2 = B_2 + C_2 - 2C_3 = 0 + 1 - 0 = 1, \quad (2.59)$$

so that D = 10 1 (decimal 4 - 1 = 3). CSD vectors are useful to represent fixed coefficients in digital filters, for example.

We can recode using a radix other than 2. Suppose B is an (n + 1)-digit two's complement number,

$$B = B_0 + B_1 2 + B_2 2^2 + \ldots + B_i 2^i + \ldots + B_{n-1} 2^{n-1} - B_n 2^n . \quad (2.60)$$

We can rewrite the expression for B using the following sleight-of-hand:

$$\begin{aligned} 2B - B = B &= -B_0 + (B_0 - B_1)2 + \ldots + (B_{i-1} - B_i)2^i + \ldots + B_{n-1} 2^{n-1} - B_n 2^n \\ &= (-2B_1 + B_0)2^0 + (-2B_3 + B_2 + B_1)2^2 + \ldots \\ &\quad + (-2B_i + B_{i-1} + B_{i-2})2^{i-1} + (-2B_{i+2} + B_{i+1} + B_i)2^{i+1} + \ldots \\ &\quad + (-2B_n + B_{i-1} + B_{i-2})2^{n-1} . \end{aligned} \quad (2.61)$$

This is very useful. Consider B = 101001 (decimal 9 - 32 = -23, n = 5),

$$\begin{aligned} B &= 101001 \\ &= (-2B_1 + B_0)2^0 + (-2B_3 + B_2 + B_1)2^2 + (-2B_5 + B_4 + B_3)2^4 \\ &\quad ((-2 ¥ 0) + 1)2^0 + ((-2 ¥ 1) + 0 + 0)2^2 + ((-2 ¥ 1) + 0 + 1)2^4 . \end{aligned} \quad (2.62)$$

Equation 2.61 tells us how to encode B as a radix-4 signed digit, E = 12 1 (decimal -16 - 8 + 1 = -23). To multiply by B encoded as E we only have to perform a multiplication by 2 (a shift) and three add/subtract operations.

Using Eq. 2.61 we can encode any number by taking groups of three bits at a time and calculating

$$\begin{aligned} E_j &= -2B_i + B_{i-1} + B_{i-2}, \\ E_{j+1} &= -2B_{i+2} + B_{i+1} + B_i, \ldots, \end{aligned} \quad (2.63)$$

where each 3-bit group overlaps by one bit. We pad B with a zero, $B_n \ldots B_1 B_0 0$, to match the first term in Eq. 2.61. If B has an odd number of bits, then we extend the sign: $B_n B_n \ldots B_1 B_0 0$. For example, B = 01011 (eleven), encodes to E = 1 11 (16 - 4 - 1); and B = 101 is E = 1 1. This is called Booth encoding and reduces the number of partial products by a factor of two and thus considerably reduces the area as well as increasing the speed of our multiplier [Booth, 1951].

Next we turn our attention to improving the speed of addition in the CSA array. Figure 2.28(a) shows a section of the 6-bit array multiplier from Figure 2.27. We can collapse the chain of adders a0-f5 (5 adder delays) to the Wallace tree consisting of adders 5.1-5.4 (4 adder delays) shown in Figure 2.28(b).

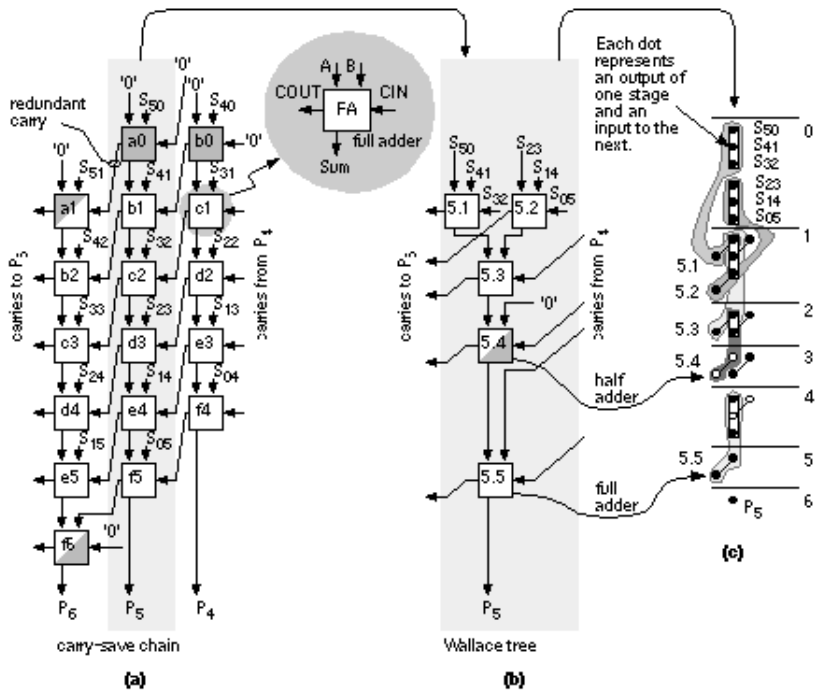


FIGURE 2.27 Tree-based multiplication. (a) The portion of Figure 2.27 that calculates the sum bit, P_5, using a chain of adders (cells a0-f5). (b) We can collapse this chain to a Wallace tree (cells 5.1-5.5). (c) The stages of multiplication.

Figure 2.28(c) pictorially represents multiplication as a sort of golf course. Each link corresponds to an adder. The holes or dots are the outputs of one stage (and the inputs of the next). At each stage we have the following three choices: (1) sum three outputs using a full adder (denoted by a box enclosing three dots); (2) sum two outputs using a half adder (a box with two dots); (3) pass the outputs directly to the next stage. The two outputs of an adder are joined by a diagonal line (full adders use black dots, half adders white dots). The object of the game is to choose (1), (2), or (3) at each stage to maximize the performance of the multiplier. In tree-based multipliers there are two ways to do this-working forward and working backward.

In a Wallace-tree multiplier we work forward from the multiplier inputs, compressing the number of signals to be added at each stage [Wallace, 1960]. We can view an FA as a 3:2 compressor or (3, 2) counter -it counts the number of ’1’s on the inputs. Thus, for example, an input of ’101’ (two ’1’s) results in an output ’10’ (2). A half adder is a (2, 2) counter . To form P_5 in Figure 2.29 we must add 6 summands (S_{05}, S_{14}, S_{23}, S_{32}, S_{41}, and S_{50}) and 4 carries from the P_4 column. We add these in stages 1-7, compressing from 6:3:2:2:3:1:1. Notice that we wait until stage 5 to add the last carry from column P_4, and this means we expand (rather than compress) the number of signals (from 2 to 3) between stages 3 and 5. The maximum delay through the CSA array of Figure 2.29 is 6 adder delays. To

this we must add the delay of the 4-bit (9 inputs) CPA (stage 7). There are 26 adders (6 half adders) plus the 4 adders in the CPA.

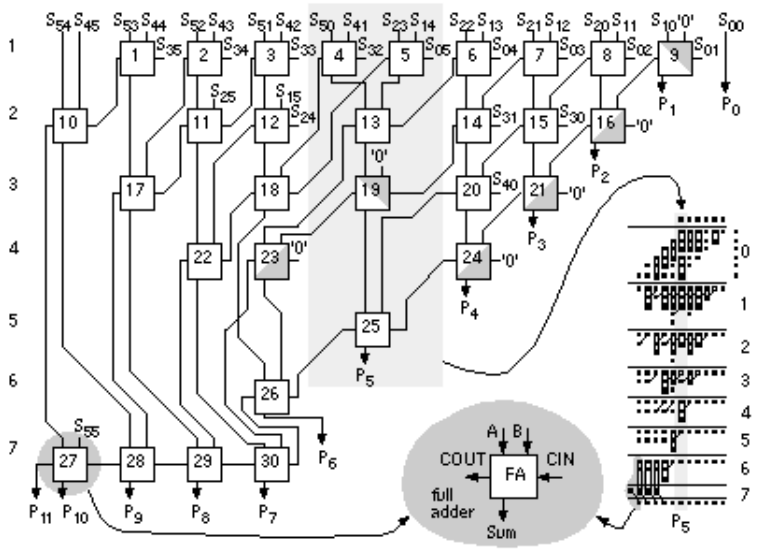


FIGURE 2.28 A 6-bit Wallace-tree multiplier. The carry-save adder (CSA) requires 26 adders (cells 1-26, six are half adders). The final carry-propagate adder (CPA) consists of 4 adder cells (27-30). The delay of the CSA is 6 adders. The delay of the CPA is 4 adders.

In a Dadda multiplier (Figure 2.30) we work backward from the final product [Dadda, 1965]. Each stage has a maximum of 2, 3, 4, 6, 9, 13, 19, . . . outputs (each successive stage is 3/2 times larger-rounded down to an integer). Thus, for example, in Figure 2.28(d) we require 3 stages (with 3 adder delays-plus the delay of a 10-bit output CPA) for a 6-bit Dadda multiplier. There are 19 adders (4 half adders) in the CSA plus the 10 adders (2 half adders) in the CPA. A Dadda multiplier is usually faster and smaller than a Wallace-tree multiplier.

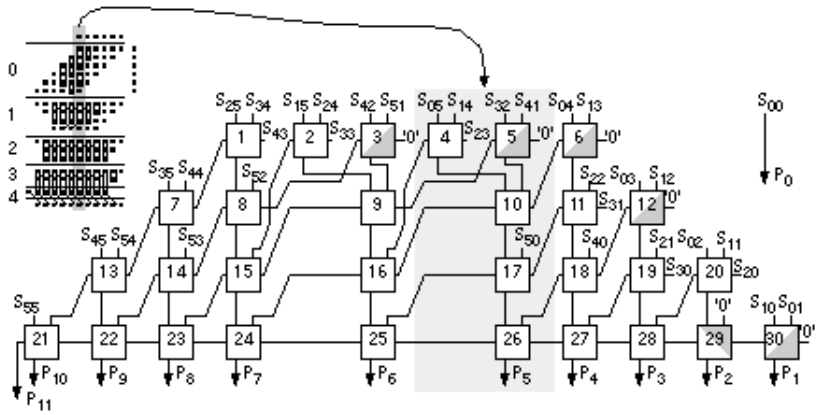


FIGURE 2.29 The 6-bit Dadda multiplier. The carry-save adder (CSA) requires 20 adders (cells 1-20, four are half adders). The carry-propagate adder (CPA, cells 21-30) is a ripple-carry adder (RCA). The CSA is smaller (20 versus 26 adders), faster (3 adder delays versus 6 adder delays), and more regular than the Wallace-tree CSA of Figure 2.29. The overall speed of this implementation is approximately

the same as the Wallace-tree multiplier of Figure 2.29; however, the speed may be increased by substituting a faster CPA.

In general, the number of stages and thus delay (in units of an FA delay-excluding the CPA) for an n -bit tree-based multiplier using (3, 2) counters is

$$\log_{1.5} n = \log_{10} n / \log_{10} 1.5 = \log_{10} n / 0.176 . \quad (2.64)$$

Figure 2.31(a) shows how the partial-product array is constructed in a conventional 4-bit multiplier. The Ferrari-Stefanelli multiplier (Figure 2.31b) "nests" multipliers-the 2-bit submultipliers reduce the number of partial products [Ferrari and Stefanelli, 1969].

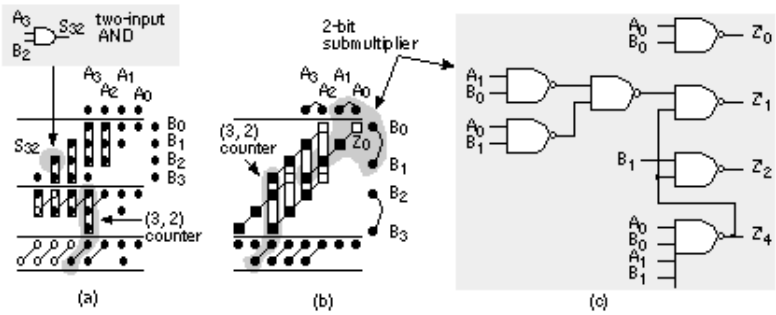


FIGURE 2.30 Ferrari-Stefanelli multiplier. (a) A conventional 4-bit array multiplier using AND gates to calculate the summands with (2, 2) and (3, 2) counters to sum the partial products. (b) A 4-bit Ferrari-Stefanelli multiplier using 2-bit submultipliers to construct the partial product array. (c) A circuit implementation for an inverting 2-bit submultiplier.

There are several issues in deciding between parallel multiplier architectures:

1. Since it is easier to fold triangles rather than trapezoids into squares, a Wallace-tree multiplier is more suited to full-custom layout, but is slightly larger, than a Dadda multiplier-both are less regular than an array multiplier. For cell-based ASICs, a Dadda multiplier is smaller than a Wallace-tree multiplier.
2. The overall multiplier speed does depend on the size and architecture of the final CPA, but this may be optimized independently of the CSA array. This means a Dadda multiplier is always at least as fast as the Wallace-tree version.
3. The low-order bits of any parallel multiplier settle first and can be added in the CPA before the remaining bits settle. This allows multiplication and the final addition to be overlapped in time.
4. Any of the parallel multiplier architectures may be pipelined. We may also use a variably pipelined approach that tailors the register locations to the size of the multiplier.
5. Using (4, 2), (5, 3), (7, 3), or (15, 4) counters increases the stage compression and permits the size of the stages to be tuned. Some ASIC cell libraries contain a (7, 3) counter-a 2-bit full-adder . A (15, 4) counter is a 3-bit full adder. There is a trade-off in using these counters between the speed and size of the logic cells and the delay as well as area of the interconnect.
6. Power dissipation is reduced by the tree-based structures. The simplified carry-save logic produces fewer signal transitions and the tree structures produce fewer glitches than a chain.
7. None of the multiplier structures we have discussed take into account the possibility of staggered

arrival times for different bits of the multiplicand or the multiplier. Optimization then requires a logic-synthesis tool.

2.6.5 Other Arithmetic Systems

There are other schemes for addition and multiplication that are useful in special circumstances. Addition of numbers using redundant binary encoding avoids carry propagation and is thus potentially very fast. Table 2.13 shows the rules for addition using an intermediate carry and sum that are added without the need for carry. For example,

binary	decimal	redundant binary	CSD vector	
1010111	87	10101001	10 1 0 1 00 1	addend
+ 1100101	101	+ 11100111	+ 01100101	augend
		01001110	= 11 00 1 100	intermediate sum
		1 1 00010 1	11000000	intermediate carry
= 10111100	= 188	1 1 1000 1 00	10 1 00 1 100	sum

TABLE 2.13 Redundant binary addition.

A[i]	B[i]	A[i - 1]	B[i - 1]	Intermediate sum	Intermediate carry
1	1	x	x	0	1
1	0	A[i - 1]=0/1 and B[i - 1]=0/1		1	0
0	1	A[i - 1]= 1 or B[i - 1]= 1		1	1
1	1	x	x	0	0
1	1	x	x	0	0
0	0	x	x	0	0
0	1	A[i - 1]=0/1 and B[i - 1]=0/1		1	1
1	0	A[i - 1]= 1 or B[i - 1]= 1		1	0
1	1	x	x	0	1

The redundant binary representation is not unique. We can represent 101 (decimal), for example, by 1100101 (binary and CSD vector) or 1 1 100111. As another example, 188 (decimal) can be represented by 10111100 (binary), 1 1 1000 1 00, 10 1 00 1 100, or 10 1 000 1 00 (CSD vector). Redundant binary addition of binary, redundant binary, or CSD vectors does not result in a unique sum, and addition of two CSD vectors does not result in a CSD vector. Each n -bit redundant binary number requires a rather wasteful 2 n -bit binary number for storage. Thus 10 1 is represented as 010010, for example (using sign magnitude). The other disadvantage of redundant binary arithmetic is the need to convert to and from binary representation.

Table 2.14 shows the (5, 3) residue number system . As an example, 11 (decimal) is represented as [1, 2] residue (5, 3) since $11R_5$ = 11 mod 5 = 1 and $11R_3$ = 11 mod 3 = 2. The size of this system is thus 3 ¥ 5 = 15. We add, subtract, or multiply residue numbers using the modulus of each bit position-without any

carry. Thus:

4	[4, 1]	12	[2, 0]	3	[3, 0]
+ 7	+ [2, 1]	- 4	- [4, 1]	¥ 4	¥ [4, 1]
= 11	= [1, 2]	= 8	= [3, 2]	= 12	= [2, 0]

TABLE 2.14 The 5, 3 residue number system.

n	residue 5	residue 3	n	residue 5	residue 3	n	residue 5	residue 3
0	0	0	5	0	2	10	0	1
1	1	1	6	1	0	11	1	2
2	2	2	7	2	1	12	2	0
3	3	0	8	3	2	13	3	1
4	4	1	9	4	0	14	4	2

The choice of moduli determines the system size and the computing complexity. The most useful choices are relative primes (such as 3 and 5). With p prime, numbers of the form 2^p and $2^p - 1$ are particularly useful ($2^p - 1$ are Mersenne's numbers) [Waser and Flynn, 1982].

2.6.6 Other Datapath Operators

Figure 2.32 shows symbols for some other datapath elements. The combinational datapath cells, NAND, NOR, and so on, and sequential datapath cells (flip-flops and latches) have standard-cell equivalents and function identically. I use a bold outline (1 point) for datapath cells instead of the regular (0.5 point) line I use for scalar symbols. We call a set of identical cells a vector of datapath elements in the same way that a bold symbol, A , represents a vector and A represents a scalar.

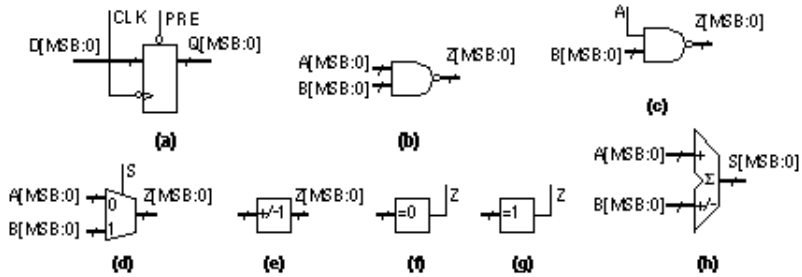


FIGURE 2.31 Symbols for datapath elements. (a) An array or vector of flip-flops (a register). (b) A two-input NAND cell with databus inputs. (c) A two-input NAND cell with a control input. (d) A buswide MUX. (e) An incrementer/decrementer. (f) An all-zeros detector. (g) An all-ones detector. (h) An adder/subtracter.

A subtracter is similar to an adder, except in a full subtracter we have a borrow-in signal, BIN; a borrow-out signal, BOUT; and a difference signal, DIFF:

$$\text{DIFF} = A \;?\; \text{NOT}(B) \;?\; \text{NOT}(\text{BIN})$$

$$\text{SUM}(A, \text{NOT}(B), \text{NOT}(\text{BIN})) \qquad (2.65)$$

$$\text{NOT}(\text{BOUT}) = A \cdot \text{NOT}(B) + A \cdot \text{NOT}(\text{BIN}) + \text{NOT}(B) \cdot \text{NOT}(\text{BIN})$$

$$\text{MAJ}(\text{NOT}(A), B, \text{NOT}(\text{BIN})) \qquad (2.66)$$

These equations are the same as those for the FA (Eqs. 2.38 and 2.39) except that the B input is inverted and the sense of the carry chain is inverted. To build a subtracter that calculates (A - B) we invert the entire B input bus and connect the BIN[0] input to VDD (not to VSS as we did for CIN[0] in an adder). As an example, to subtract B = ’0011’ from A = ’1001’ we calculate ’1001’ + ’1100’ + ’1’ = ’0110’. As with an adder, the true overflow is XOR(BOUT[MSB], BOUT[MSB - 1]).

We can build a ripple-borrow subtracter (a type of borrow-propagate subtracter), a borrow-save subtracter, and a borrow-select subtracter in the same way we built these adder architectures. An adder/subtracter has a control signal that gates the A input with an exclusive-OR cell (forming a programmable inversion) to switch between an adder or subtracter. Some adder/subtracters gate both inputs to allow us to compute (-A - B). We must be careful to connect the input to the LSB of the carry chain (CIN[0] or BIN[0]) when changing between addition (connect to VSS) and subtraction (connect to VDD).

A barrel shifter rotates or shifts an input bus by a specified amount. For example if we have an eight-input barrel shifter with input ’1111 0000’ and we specify a shift of ’0001 0000’ (3, coded by bit position) the right-shifted 8-bit output is ’0001 1110’. A barrel shifter may rotate left or right (or switch between the two under a separate control). A barrel shifter may also have an output width that is smaller than the input. To use a simple example, we may have an 8-bit input and a 4-bit output. This situation is equivalent to having a barrel shifter with two 4-bit inputs and a 4-bit output. Barrel shifters are used extensively in floating-point arithmetic to align (we call this normalize and denormalize) floating-point numbers (with sign, exponent, and mantissa).

A leading-one detector is used with a normalizing (left-shift) barrel shifter to align mantissas in floating-point numbers. The input is an n -bit bus A, the output is an n -bit bus, S, with a single ’1’ in the bit position corresponding to the most significant ’1’ in the input. Thus, for example, if the input is A = ’0000 0101’ the leading-one detector output is S = ’0000 0100’, indicating the leading one in A is in bit position 2 (bit 7 is the MSB, bit zero is the LSB). If we feed the output, S, of the leading-one detector to the shift select input of a normalizing (left-shift) barrel shifter, the shifter will normalize the input A. In our example, with an input of A = ’0000 0101’, and a left-shift of S = ’0000 0100’, the barrel shifter will shift A left by five bits and the output of the shifter is Z = ’1010 0000’. Now that Z is aligned (with the MSB equal to ’1’) we can multiply Z with another normalized number.

The output of a priority encoder is the binary-encoded position of the leading one in an input. For example, with an input A = ’0000 0101’ the leading 1 is in bit position 3 (MSB is bit position 7) so the output of a 4-bit priority encoder would be Z = ’0011’ (3). In some cell libraries the encoding is reversed so that the MSB has an output code of zero, in this case Z = ’0101’ (5). This second, reversed, encoding scheme is useful in floating-point arithmetic. If A is a mantissa and we normalize A to ’1010 0000’ we have to subtract 5 from the exponent, this exponent correction is equal to the output of the priority encoder.

An accumulator is an adder/subtracter and a register. Sometimes these are combined with a multiplier to

form a multiplier-accumulator (MAC). An incrementer adds 1 to the input bus, Z = A + 1, so we can use this function, together with a register, to negate a two's complement number for example. The implementation is Z[i] = XOR(A[i], CIN[i]), and COUT[i] = AND(A[i], CIN[i]). The carry-in control input, CIN[0], thus acts as an enable: If it is set to '0' the output is the same as the input.

The implementation of arithmetic cells is often a little more complicated than we have explained. CMOS logic is naturally inverting, so that it is faster to implement an incrementer as

Z[i (even)] = XOR(A[i], CIN[i]) and COUT[i (even)] = NAND(A[i], CIN[i]).

This inverts COUT, so that in the following stage we must invert it again. If we push an inverting bubble to the input CIN we find that:

Z[i (odd)] = XNOR(A[i], CIN[i]) and COUT[i (even)] = NOR(NOT(A[i]), CIN[i]).

In many datapath implementations all odd-bit cells operate on inverted carry signals, and thus the odd-bit and even-bit datapath elements are different. In fact, all the adder and subtracter datapath elements we have described may use this technique. Normally this is completely hidden from the designer in the datapath assembly and any output control signals are inverted, if necessary, by inserting buffers.

A decrementer subtracts 1 from the input bus, the logical implementation is Z[i] = XOR(A[i], CIN[i]) and COUT[i] = AND(NOT(A[i]), CIN[i]). The implementation may invert the odd carry signals, with CIN[0] again acting as an enable.

An incrementer/decrementer has a second control input that gates the input, inverting the input to the carry chain. This has the effect of selecting either the increment or decrement function.

Using the all-zeros detectors and all-ones detectors , remember that, for a 4-bit number, for example, zero in ones' complement arithmetic is '1111' or '0000', and that zero in signed magnitude arithmetic is '1000' or '0000'.

A register file (or scratchpad memory) is a bank of flip-flops arranged across the bus; sometimes these have the option of multiple ports (multiport register files) for read and write. Normally these register files are the densest logic and hardest to fit in a datapath. For large register files it may be more appropriate to use a multiport memory. We can add control logic to a register file to create a first-in first-out register (FIFO), or last-in first-out register (LIFO).

In Section 2.5 we saw that the standard-cell version and gate-array macro version of the sequential cells (latches and flip-flops) each contain their own clock buffers. The reason for this is that (without intelligent placement software) we do not know where a standard cell or a gate-array macro will be placed on a chip. We also have no idea of the condition of the clock signal coming into a sequential cell. The ability to place the clock buffers outside the sequential cells in a datapath gives us more flexibility and saves space. For example, we can place the clock buffers for all the clocked elements at the top of the datapath (together with the buffers for the control signals) and river route (in river routing the interconnect lines all flow in the same direction on the same layer) the connections to the clock lines. This saves space and allows us to guarantee the clock skew and timing. It may mean, however, that there is a fixed overhead associated with a datapath. For example, it might make no sense to build a 4-bit

datapath if the clock and control buffers take up twice the space of the datapath logic. Some tools allow us to design logic using a portable netlist . After we complete the design we can decide whether to implement the portable netlist in a datapath, standard cells, or even a gate array, based on area, speed, or power considerations.

2.7 I/O Cells

Figure 2.33 shows a three-state bidirectional output buffer (Tri-State ® is a registered trademark of National Semiconductor). When the output enable (OE) signal is high, the circuit functions as a noninverting buffer driving the value of DATAin onto the I/O pad. When OE is low, the output transistors or drivers , M1 and M2, are disconnected. This allows multiple drivers to be connected on a bus. It is up to the designer to make sure that a bus never has two drivers-a problem known as contention .

In order to prevent the problem opposite to contention-a bus floating to an intermediate voltage when there are no bus drivers-we can use a bus keeper or bus-hold cell (TI calls this Bus-Friendly logic). A bus keeper normally acts like two weak (low drive-strength) cross-coupled inverters that act as a latch to retain the last logic state on the bus, but the latch is weak enough that it may be driven easily to the opposite state. Even though bus keepers act like latches, and will simulate like latches, they should not be used as latches, since their drive strength is weak.

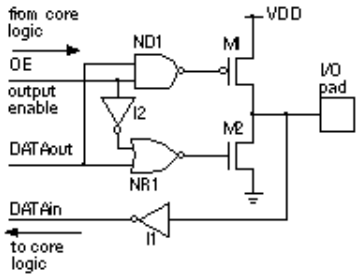
Transistors M1 and M2 in Figure 2.33 have to drive large off-chip loads. If we wish to change the voltage on a C = 200 pF load by 5 V in 5 ns (a slew rate of 1 Vns $^{-1}$) we will require a current in the output transistors of I_{DS} = C (d V /d t) = (200 ¥ 10 $^{-12}$) (5/5 ¥ 10 $^{-9}$) = 0.2 A or 200 mA.

Such large currents flowing in the output transistors must also flow in the power supply bus and can cause problems. There is always some inductance in series with the power supply, between the point at which the supply enters the ASIC package and reaches the power bus on the chip. The inductance is due to the bond wire, lead frame, and package pin. If we have a power-supply inductance of 2 nH and a current changing from zero to 1 A (32 I/O cells on a bus switching at 30 mA each) in 5 ns, we will have a voltage spike on the power supply (called power-supply bounce) of L (d I /d t) = (2 ¥ 10 $^{-9}$)(1/(5 ¥ 10 $^{-9}$)) = 0.4 V.

We do several things to alleviate this problem: We can limit the number of simultaneously switching outputs (SSOs), we can limit the number of I/O drivers that can be attached to any one VDD and GND pad, and we can design the output buffer to limit the slew rate of the output (we call these slew-rate limited I/O pads). Quiet-I/O cells also use two separate power supplies and two sets of I/O drivers: an AC supply (clean or quiet supply) with small AC drivers for the I/O circuits that start and stop the output slewing at the beginning and end of a output transition, and a DC supply (noisy or dirty supply) for the transistors that handle large currents as they slew the output.

The three-state buffer allows us to employ the same pad for input and output- bidirectional I/O . When we want to use the pad as an input, we set OE low and take the data from DATAin. Of course, it is not necessary to have all these features on every pad: We can build output-only or input-only pads.

FIGURE 2.32 A three-state bidirectional output buffer. When the output enable, OE, is ’1’ the output section is enabled and drives the I/O pad. When OE is ’0’ the output buffer is placed in a high-impedance state.



We can also use many of these output cell features for input cells that have to drive large on-chip loads (a clock pad cell, for example). Some gate arrays simply turn an output buffer around to drive a grid of interconnect that supplies a clock signal internally. With a typical interconnect capacitance of 0.2pFcm $^{-1}$, a grid of 100 cm (consisting of 10 by 10 lines running all the way across a 1 cm chip) presents a load of 20 pF to the clock buffer.

Some libraries include I/O cells that have passive pull-ups or pull-downs (resistors) instead of the transistors, M1 and M2 (the resistors are normally still constructed from transistors with long gate lengths). We can also omit one of the driver transistors, M1 or M2, to form open-drain outputs that require an external pull-up or pull-down. We can design the output driver to produce TTL output levels rather than CMOS logic levels. We may also add input hysteresis (using a Schmitt trigger) to the input buffer, I1 in Figure 2.33, to accept input data signals that contain glitches (from bouncing switch contacts, for example) or that are slow rising. The input buffer can also include a level shifter to accept TTL input levels and shift the input signal to CMOS levels.

The gate oxide in CMOS transistors is extremely thin (100 Å or less). This leaves the gate oxide of the I/O cell input transistors susceptible to breakdown from static electricity (electrostatic discharge , or ESD). ESD arises when we or machines handle the package leads (like the shock I sometimes get when I touch a doorknob after walking across the carpet at work). Sometimes this problem is called electrical overstress (EOS) since most ESD-related failures are caused not by gate-oxide breakdown, but by the thermal stress (melting) that occurs when the n -channel transistor in an output driver overheats (melts) due to the large current that can flow in the drain diffusion connected to a pad during an ESD event.

To protect the I/O cells from ESD, the input pads are normally tied to device structures that clamp the input voltage to below the gate breakdown voltage (which can be as low as 10 V with a 100 Å gate oxide). Some I/O cells use transistors with a special ESD implant that increases breakdown voltage and provides protection. I/O driver transistors can also use elongated drain structures (ladder structures) and large drain-to-gate spacing to help limit current, but in a salicide process that lowers the drain resistance this is difficult. One solution is to mask the I/O cells during the salicide step. Another solution is to use pnpn and npnp diffusion structures called silicon-controlled rectifiers (SCRs) to clamp voltages and divert current to protect the I/O circuits from ESD.

There are several ways to model the capability of an I/O cell to withstand EOS. The human-body model (HBM) represents ESD by a 100 pF capacitor discharging through a 1.5 k W resistor (this is an International Electrotechnical Committee, IEC, specification). Typical voltages generated by the human body are in the range of 2-4 kV, and we often see an I/O pad cell rated by the voltage it can withstand using the HBM. The machine model (MM) represents an ESD event generated by automated machine

handlers. Typical MM parameters use a 200 pF capacitor (typically charged to 200 V) discharged through a 25 W resistor, corresponding to a peak initial current of nearly 10 A. The charge-device model (CDM , also called device charge-discharge) represents the problem when an IC package is charged, in a shipping tube for example, and then grounded. If the maximum charge on a package is 3 nC (a typical measured figure) and the package capacitance to ground is 1.5 pF, we can simulate this event by charging a 1.5 pF capacitor to 2 kV and discharging it through a 1 W resistor.

If the diffusion structures in the I/O cells are not designed with care, it is possible to construct an SCR structure unwittingly, and instead of protecting the transistors the SCR can enter a mode where it is latched on and conducting large enough currents to destroy the chip. This failure mode is called latch-up . Latch-up can occur if the pn -diodes on a chip become forward-biased and inject minority carriers (electrons in p -type material, holes in n -type material) into the substrate. The source-substrate and drain-substrate diodes can become forward-biased due to power-supply bounce or output undershoot (the cell outputs fall below V_{SS}) or overshoot (outputs rise to greater than V_{DD}) for example. These injected minority carriers can travel fairly large distances and interact with nearby transistors causing latch-up. I/O cells normally surround the I/O transistors with guard rings (a continuous ring of n -diffusion in an n -well connected to VDD, and a ring of p -diffusion in a p

2.8 Cell Compilers

The process of hand crafting circuits and layout for a full-custom IC is a tedious, time-consuming, and error-prone task. There are two types of automated layout assembly tools, often known as a silicon compilers . The first type produces a specific kind of circuit, a RAM compiler or multiplier compiler , for example. The second type of compiler is more flexible, usually providing a programming language that assembles or tiles layout from an input command file, but this is full-custom IC design.

We can build a register file from latches or flip-flops, but, at 4.5-6.5 gates (18-26 transistors) per bit, this is an expensive way to build memory. Dynamic RAM (DRAM) can use a cell with only one transistor, storing charge on a capacitor that has to be periodically refreshed as the charge leaks away. ASIC RAM is invariably static (SRAM), so we do not need to refresh the bits. When we refer to RAM in an ASIC environment we almost always mean SRAM. Most ASIC RAMs use a six-transistor cell (four transistors to form two cross-coupled inverters that form the storage loop, and two more transistors to allow us to read from and write to the cell). RAM compilers are available that produce single-port RAM (a single shared bus for read and write) as well as dual-port RAMs , and multiport RAMs . In a multi-port RAM the compiler may or may not handle the problem of address contention (attempts to read and write to the same RAM address simultaneously). RAM can be asynchronous (the read and write cycles are triggered by control and/or address transitions asynchronous to a clock) or synchronous (using the system clock).

In addition to producing layout we also need a model compiler so that we can verify the circuit at the behavioral level, and we need a netlist from a netlist compiler so that we can simulate the circuit and verify that it works correctly at the structural level. Silicon compilers are thus complex pieces of software. We assume that a silicon compiler will produce working silicon even if every configuration has not been tested. This is still ASIC design, but now we are relying on the fact that the tool works correctly and therefore the compiled blocks are correct by construction

2.9 Summary

The most important concepts that we covered in this chapter are the following:

- The use of transistors as switches
- The difference between flip-flop and a latch
- The meaning of setup time and hold time
- Pipelines and latency
- The difference between datapath, standard-cell, and gate-array logic cells
- Strong and weak logic levels
- Pushing bubbles
- Ratio of logic
- Resistance per square of layers and their relative values in CMOS
- Design rules and l