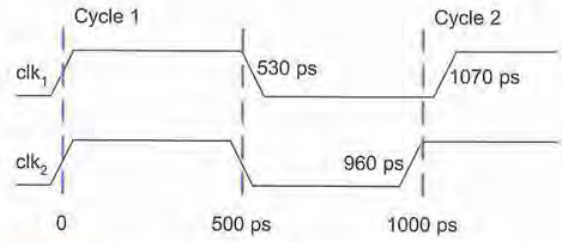


FIG 12.27 Skewed clock waveforms

For a path between two flip-flops, the hold time constraint depends on the skew between the same rising edges of both physical clocks. The setup time constraint depends on the skew between the rising edge of one physical clock and the subsequent rising edge of the other. We will see that clock distribution networks tend to introduce more skew from one cycle to the next so setup and hold time constraints can budget different amounts of skew.

The actual clock skew between two clocked elements varies with time and is different from one chip to another. Moreover, it is unknowable at design time. From the engineering perspective, a more useful parameter is the *clock skew budget*. The clock skew budget should be larger than the actual skew encountered on any long or short path on any working chip, yet no larger than necessary lest the chip be overdesigned.

While in principle designers could tabulate clock skew budgets between physical clocks at every pair of clocked elements on the chip, the table would be unreasonably large and unwieldy. Instead, they group physical clocks into *clock domains* and use a single skew budget to describe the entire domain. For example, you could define two latches to be in a local clock domain if their physical distance is no more than 500 μm. Then you could just define local and global skews, with the local skew being smaller than the global skew. If the clock period is long compared to the maximum skew, you can define only a single global skew budget and pessimistically assume all clocked elements might see this worst-case skew.

Clock skew sources can be classified as *systematic*, *random*, *drift*, and *jitter*. Figure 12.28(a) illustrates these sources in a simple clock distribution network. The global clock is distributed along wires to two gaters. One wire is 3 mm, while the other is 3.1 mm. The gaters are nominally identical, but one drives a lumped load of 1.3 pF while the other

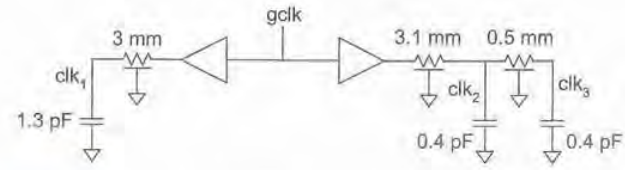


FIG 12.28 Simple clock distribution network

drives a load of 0.8 pF distributed along a 0.5 mm wire. The *systematic* clock skew is the portion that exists even under nominal conditions; this component can be predicted by simulation. By adjusting the size of one of the gaters, the systematic skew between clk_1 and clk_2 could be driven to zero. However, some systematic skew will always exist between clk_2 and clk_3 because of the flight time along the wire after the gater.

The *random* component of skew is caused by manufacturing variations that could affect the wire width, thickness, or spacing and the transistor channel length, threshold voltage, or oxide thickness. These cause unpredictable changes in resistance, capacitance, and transistor current, introducing additional skew. In principle, the actual random skew could be measured during chip test or on startup, and adjustable delay elements could be calibrated to compensate for the random skew.

Drift is caused by time-dependent environmental variations that occur relatively slowly. For example, after the chip turns on, it will heat up. The temperature affects gate and wire delay differently. Also, a temperature gradient across the chip leads to skew. Drift can also be nulled out with adjustable delay elements. Unlike random skew, compensating for drift must take place periodically rather than just once at startup. The frequency of calibration depends on the thermal time constant of the chip.

Jitter is caused by high-frequency environmental variation, particularly power supply noise. This noise leads to delay variation in the clock buffers and gaters in both time and space. Jitter is particularly insidious because it occurs too rapidly for compensation circuits to be able to counter it.

Some engineers do not report jitter as part of the skew. In such a case, they must include both jitter and skew in the setup and hold time budgets.

12.5.2 Clock System Architecture

Figure 12.29 shows an overview of a typical clock subsystem. The chip receives an external clock signal through the I/O pads. The clock generation unit may include a *phase-locked loop* (PLL) or *delay-locked loop* (DLL) to adjust the frequency or phase of the global clock, as shall be discussed in Section 12.5.3. This global clock is then distributed across the chip to points near all of the clocked elements. The clock distribution network must be carefully designed to minimize clock skew. Local clock gaters receive this global clock and drive the physical clock signals along short wires to small groups of clocked elements.

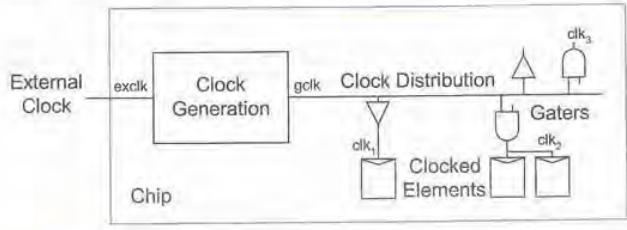


FIG 12.29 Clock subsystem

12.5.3 Global Clock Generation

The global clock generator receives an external clock signal and produces the global clock that will be distributed across the die. In this simplest case, the clock generator is simply a buffer to drive the large capacitance of the clock distribution network. However, the input pad, buffering, distribution network, and gaters have significant delay that leads to a large skew between the external clock and the physical clocks received at the clocked elements. Moreover, this delay varies with processing and environment. Because of this skew, clocked elements on the chip are no longer synchronized with the external I/O signals. Guaranteeing setup and hold times becomes problematic, particularly at high frequencies where the skew exceeds half of the clock period. More sophisticated clock generators use *phase-locked loops* (PLLs) to compensate for this delay. Moreover, phase-locked loops can perform frequency multiplication to provide an on-chip clock at a higher frequency than the external clock.

Figure 12.30 shows a typical synchronous chip interface using a phase-locked loop to compensate for on-chip clock delays [Chandrakasan01]. In this example, Chip 1 sends a clock and D_{out} to Chip 2 and receives D_{in} back from Chip 2. The data should be synchronized to the clock so each chip can sample it on the positive clock edge. However, the internal clock on Chip 2 is delayed through the clock distribution network. The PLL adjusts the internal clock in Chip 2 to correct for this delay and keep the clock synchronized with the data.

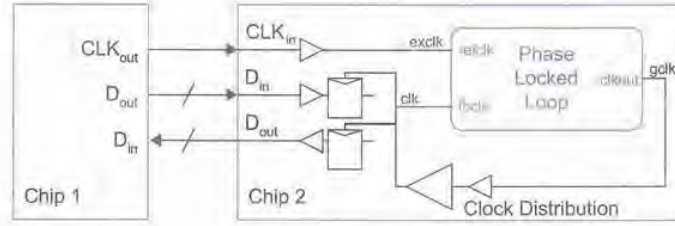


FIG 12.30 Synchronous chip interface with PLL

A phase-locked loop receives a *reference clock* and a *feedback clock* and produces an *output clock*. The phase and frequency of the output clock is automatically adjusted until the feedback clock is exactly aligned with the reference clock. In our application, the reference clock comes from Chip 1. The feedback clock is tapped off one of the physical clock wires that also drives the registers. The output clock is the *gclk* signal sent into the clock distribution network. Thus, the PLL adjusts *gclk* until the clock *clk* driving the data registers is aligned with the external clock *exclk*. This eliminates the systematic skew caused by the clock distribution delay.

Figure 12.31(a) shows a block diagram of a typical PLL. The heart of the PLL is a *voltage-controlled oscillator* (VCO). The control voltage is adjusted until the oscillator pro-

duces an output clock of the proper phase at the same frequency as the reference clock. This control is performed with a charge pump and RC filter. A phase detector determines whether the feedback clock leads or lags the reference clock. The charge pump consists of a pair of current sources enabled by the up and down signals to adjust the voltage on V_{ctrl} until the feedback clock becomes aligned with the reference.

Figure 12.31(b) shows a PLL that performs frequency multiplication. It uses a divide-by-N counter on the *fbclk* to generate a clock aligned in phase but at N times the frequency of the reference clock. For example, the Pentium 4 uses a 100 MHz external system clock that can be multiplied up to a 3 GHz core clock. With another divide-by-M counter on the *refclk* terminal, the PLL can produce any rational N/M multiple of the input frequency.

A PLL is a feedback system and requires careful analysis to ensure stability over all process and environmental corners. The RC loop filter behaves as a second-order system, although stray capacitance on V_{ctrl} often leads to third-order effects. Noise on V_{ctrl} causes the VCO to change frequency, which leads to a phase error that increases over time; this problem is called *phase-error accumulation* and appears as jitter in the clock skew budget. Therefore, the loop filter should have high enough bandwidth to rapidly correct for noise. Phase-error accumulation impacts the timing budget for one chip communicating with another. However, only cycle-cycle jitter is important for on-chip paths between flip-flops. RC loop filters are difficult to build because of uncertainties in the passive component values caused by process variation. Thus, many PLLs use an active filter.

Figure 12.32 shows a circuit-level implementation of a simple PLL. The phase detector is a *phase-frequency detector* consisting of a pair of flip-flops with asynchronous reset that pro-

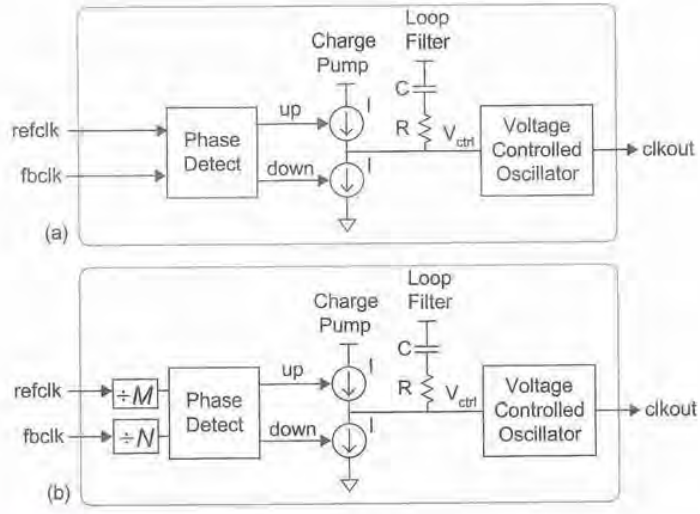


FIG 12.31 Phase-locked loop block diagram

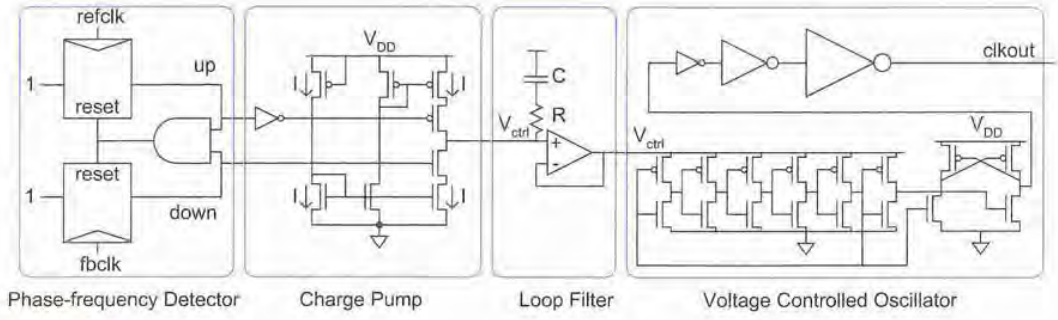


FIG 12.32 Simple PLL implementation

duces pulses to drive the frequency up or down depending on which clock arrived first. The charge pump uses a pair of current sources switched on by the *up* and *down* signals. Current sources are discussed further in Section 12.6.4. V_{ctrl} is buffered with an amplifier such as that from Figure 12.63 hooked up as a unity-gain follower. The VCO consists of an ordinary ring oscillator running off V_{ctrl} rather than V_{DD} so that its period increases as V_{ctrl} decreases. The number of stages determines the range over which the frequency can be adjusted. A level converter and some buffers amplify the signal to drive *clkout* across the chip.

Power supply and substrate noise are the primary sources of PLL jitter, so the PLL should use a filtered power supply and generous guard rings to limit jitter. [Maneatis96, Maneatis03] describe self-biased delay elements that minimize sensitivity to supply noise and process variation. [Ingino01] describes a PLL with a filtered power supply operating up to 4 GHz with a ±25 ps peak-to-peak jitter.

A delay-locked loop (DLL) is a variant of a PLL that uses a voltage-controlled delay line rather than a voltage-controlled oscillator, as shown in Figure 12.33. It can be viewed as a control system that adjusts phase rather than frequency on *clkout*. DLLs are not capable of frequency multiplication. However, noise on V_{ctrl} creates only a steady phase error, so DLLs avoid phase-error accumulation. DLLs use a first-order loop filter, so ensuring stability is also easier.

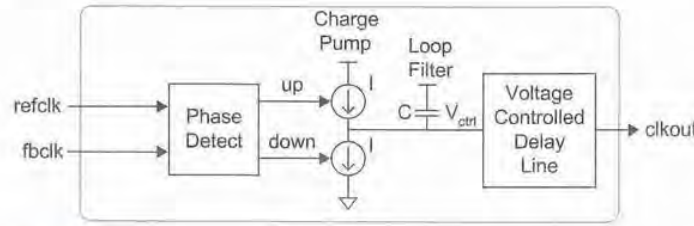


FIG 12.33 Delay-locked loop block diagram

PLLs and DLLs are notoriously difficult to build correctly. They require expertise in both feedback control systems and analog design. They must be carefully designed to acquire lock successfully and operate correctly with low jitter across process and environmental variations. A number of texts including [Chandrakasan01, Baker98, Dally98, Best03] offer introductions to loop design. For many applications, loops are best obtained as predesigned cells from a third party that specializes in loop design and offers a guarantee. True Circuits is a well-regarded PLL/DLL supplier.

12.5.4 Global Clock Distribution

The global clock must be distributed across the chip in a way that reaches all of the clocked elements at nearly the same time. In antiquated processes with slow transistors and fast wires, the clock wire had negligible delay and any convenient routing plan could be used to distribute the clock. In modern processes, the RC delay of the resistive clock wire driving its own capacitance and the clock load capacitance tends to be close to 1 ns for a well-designed distribution network covering a 15 mm square die. If the clock were routed randomly, this would lead to a clock skew of about 1 ns between physical clocks near and far from the clock generator. This could be several times the cycle time of the system. Thus, the clock distribution system must be carefully designed to equalize the *flight time* between the clock generator and the clocked receivers. Global clock distribution networks can be classified as *grids*, *H-trees*, *spines*, *ad-hoc*, or *hybrid* [Restle98].

Random skew, drift, and jitter from the clock distribution network are proportional to the delay through the network because they are caused by process or environmental variations in the distribution elements. Therefore, the designer should try to keep this distribution delay low. Unfortunately, as chips are getting larger, wires are getting slower, and clock loads are increasing, the distribution delay tends to go up even as cycle times are going down. In the past, systematic clock skew was the dominant component. Now, good clock distribution networks achieve low systematic skews, but the random, drift, and jitter components are becoming an increasing fraction of the cycle time.

12.5.4.1 Grids A clock grid is a mesh of horizontal and vertical wires driven from the middle or edges. The mesh is fine enough to deliver the clock to points nearby every clocked element. The resistance is low between any two nearby points in the mesh so the skew is also low between nearby clocked elements. This reduces the chance of hold-time problems because such problems tend to occur between nearby elements where the propagation delay between elements is also small. Grids also compensate for much of the random skew because shorting the clock together makes variations in delays irrelevant. They can be routed early in the design without detailed knowledge of latch placement. However, grids do have significant systematic skew between the points closest to the drivers and the points furthest away. They also consume a large amount of metal resources and hence have a high switching capacitance and power consumption. Section 12.8 traces the evolution of clock grids in the Alpha series of microprocessors.

12.5.4.2 H-Trees An H-tree is a fractal structure built by drawing an H shape, then recursively drawing H shapes on each of the vertices, as shown in Figure 12.34. With enough recursions, the H-tree can distribute a clock from the center to within an arbitrarily short distance of every point on the chip while maintaining exactly equal wire lengths. Buffers are added as necessary to serve as repeaters. If the clock loads were uniformly distributed around the chip, the H-tree would have zero systematic skew. Moreover, the trees tend to use less wire and thus have lower capacitance than grids [Restle98].

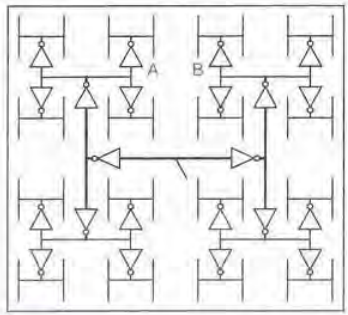


FIG 12.34 H-tree

In practice, the H-tree still shows some skew because the clock loads are not uniform, loading some leaves of the tree more than others. Moreover, the tree often must be routed around obstructions such as memory arrays. The leaves of the H do not reach every point on the chip, so some short physical clock wires are required after the local clock gater. Nevertheless, with careful tapering of the wires and sizing of the clock gaters, H-trees can deliver nearly zero systematic skew. A drawback of H-trees is that they may have high random skew, drift, and jitter between two nearby points that are leaves of different legs of the tree. For example, the points *A* and *B* in Figure 12.34 might experience large skews. As the points are close, this is a particular problem for hold times.

Figure 12.35 shows a modified H-tree used on the Itanium 2. The primary clock driver in the center of the chip sends a differential output to four differential repeaters on the leaves of the H. These repeaters drive a somewhat irregular pattern of wiring to second-level clock buffers (SLCBs) serving units all across the chip. The wiring and SLCB placement is determined by the nonuniform clock loads and obstructions on the chip. A custom clock router automatically generated the tree based on the actual clock loads so that the tree could be easily rerouted when loads change late in the design process. The SLCBs drive local clock gaters, producing the multitude of clock waveforms used on the microprocessor. Some of these waveforms were shown in Section 7.9.2.

Figure 12.36 shows the differential driver used as a primary clock buffer and repeater on the Itanium 2 [Anderson02]. The input stage is a differential amplifier sensitive to the point where the differential inputs cross over. The repeater pulses either p_1 or n_1 and p_2 or n_2 to switch the internal nodes y and $\overline{y}$. The small tristate keeper prevents these nodes from floating after the pulse terminates. The SLCB uses the same structure, but produces only a single-ended output. It also provides a current-starved adjustable delay line to compensate for systematic skew and to help locate critical paths during debug. The repeater provides a high drive capability with a low input capacitance. Thus, few stages of clock buffering are needed in the network. With so few repeaters, the area overhead of providing a filtered power supply is modest. Although the repeaters are relatively slow, their jitter is controlled with supply filtering.

12.5.4.3 Spines Figure 12.37 shows a clock distribution scheme using a pair of spines. As with the grid, the clock buffers are located in a few rows across the chip. However, instead of driving a single clock grid across the entire die, the spines drive length-matched serpentine wires to each small group of clocked elements. If the loads are uniform, the

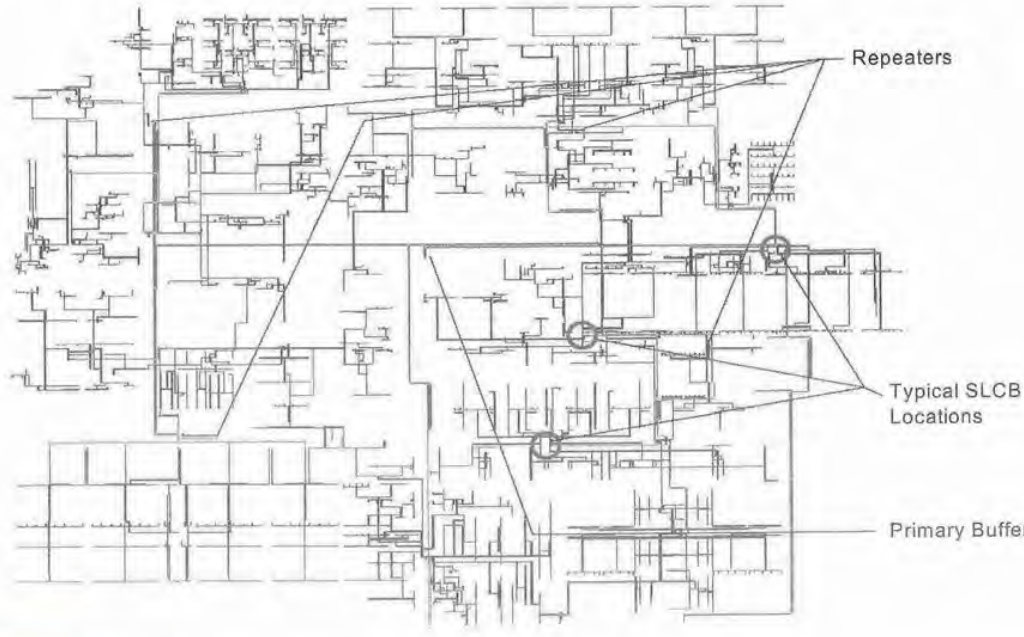


FIG 12.35 Itanium 2 modified H-tree

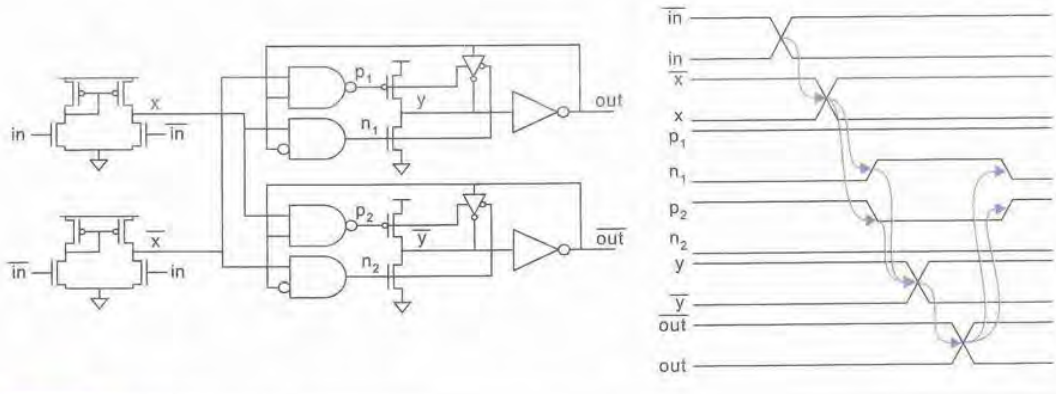


FIG 12.36 Itaniuim 2 repeater

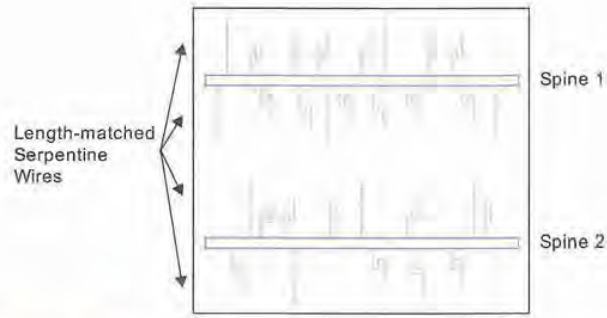


FIG 12.37 Clock spines with serpentine routing

spine avoids the systematic skew of the grid by matching the length of the clock wires. Each serpentine is driven individually so gaters can be used to save power by not switching certain wires. The serpentine is also easy to design and each load can be tuned individually. However, a system with many clocked elements may require a large number of serpentine routes, leading to high area and capacitance for the clock network. Like trees, spines also may have large local skews between nearby elements driven by different serpentines.

The Pentium II and III use a pair of clock spines [Geannopoulos98]. The Pentium 4 adds a third clock spine to reduce the length of the final clock wires [Kurd01]. Figure 12.38(a) shows the global clock buffers distributing the clock to the three spines on the Pentium 4 with zero systematic skew while Figure 12.38(b) shows a photograph of the chip annotated with the clock spine locations. The spines drive 47 independent clock domains, each of which can be gated individually. The clock domain gaters also contain adjustable delay buffers used to null out systematic and random skew and even to force deliberate clock delay to improve performance.

12.5.4.4 Ad-hoc Many ASICs running at relatively low frequencies (100's of MHz) still get away with an ad-hoc clock distribution network in which the clock is routed haphazardly with some attempt to equalize wire lengths or add buffers to equalize delay. Such ad-hoc networks can have reasonably low systematic skews because the buffer sizes can be adjusted until the nominal delays are nearly equal. However, they are subject to severe random skew when process variations affect wire and gate delays differently. This is the level that most commonly available tools support. Most design teams using ad-hoc clock networks also lack the resources to do a careful analysis of random skew, jitter, and drift. Therefore, they should be conservative in defining a skew budget and must be careful about hold time violations.

12.5.4.5 Hybrid A hybrid combination of the H-tree and grid offers lower skew than either an H-tree or grid alone. In the hybrid approach, an H-tree is used to distribute the clock to a large number of points across the die. A grid shorts these points together. Com-

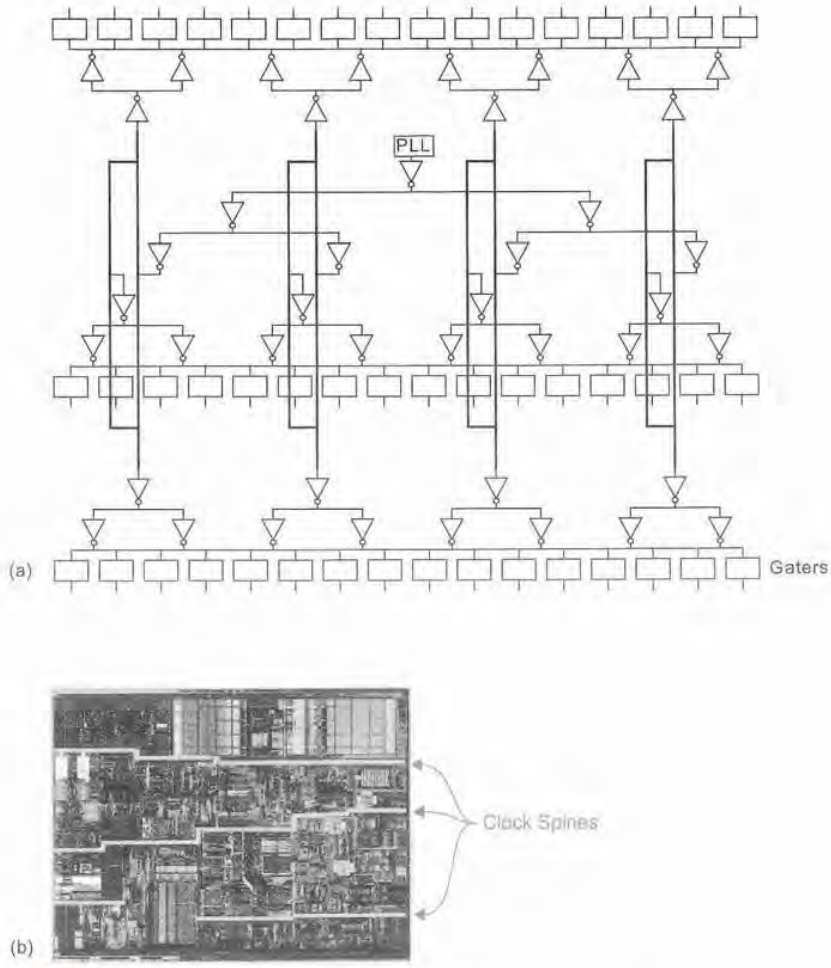


FIG 12.38 Pentium 4 clock spines. (b) © 2001 IEEE.

pared to a simple grid, the hybrid approach has lower systematic skew because the grid is driven from many points instead of just the middle or edge. Compared to an H-tree, the hybrid approach is less susceptible to skew from nonuniform load distributions. The grid also reduces local skew and brings the clock near every location where it is needed. Finally, the hybrid approach is regular, making layout of well-controlled transmission line structures easier.

IBM has used such a hybrid distribution network on a variety of microprocessors including the Power4, PowerPC, and S/390 [Restle01]. A primary buffered H-tree drives

16–64 sector buffers arranged across the chip. Each sector buffer drives a smaller tree network. Each tree can be tuned to accommodate nonuniform load capacitance by adjusting the wire widths. Together, the tunable trees drive the global clock grid at up to 1024 points. IBM uses a specialized tool to perform the tuning.

12.5.4.6 Layout Issues High-speed clock distribution networks require careful layout to minimize skew. The two guiding principles are that the network should be as uniform and as fast as possible. In a uniform network, chip-wide process or environmental variations should affect all clock paths identically. In a fast network, localized variations that cause a fractional difference between two clock path delays lead only to modest amounts of skew. For example, voltage noise that causes a 10% delay variation between two paths through an H-tree will lead to 80 ps of jitter if the tree delay is 800 ps, but 160 ps of jitter if the tree delay is 1600 ps.

Building a fast clock network requires low-resistance global clock wires with proper repeater insertion. The thick, top-level metal layer is well-suited to clock distribution. The wide wires should be shielded on both sides with V_{DD} or GND lines to prevent capacitive coupling between the clock and signal lines. The clock can even be shielded on a lower metal layer to form a microstrip waveguide [Anderson02].

Wide, low-resistance wires also have significant inductive effects, including faster than expected edge rates and overshoot. The fast edges are desirable, but overshoot should be minimized to prevent overvoltage damage. High-performance clock networks must be extracted with a field solver and modeled as transmission lines [Huang03]. Uniformity is again important: Even if the RC delays appear to be matched in a nonuniform layout, the *RLC* delays can be significantly different. As discussed in Section 4.5.5, wide wires should be split into multiple narrower traces interdigitated with V_{DD}/GND wires that provide a low-inductance current return path and minimize skin effect.

12.5.5 Local Clock Gaters

Local clock gaters receive the global clock and produce the physical clocks required by the clocked elements. The output of the gaters typically run a short distance (< 1 mm) to the clocked elements. Clock gaters are often used to stop or *gate* the clock to unused blocks of logic to save power. As discussed in Chapter 7, they can produce a variety of modified clock waveforms including pulsed clocks, delayed clocks, stretched clocks, nonoverlapping clocks, and double-frequency pulsed clocks. When used to modify the clock edges, they are sometimes called *clock choppers* or *clock stretchers*. Figure 12.39 shows a variety of clock gaters.

Most systems require a large number of clock gaters, so it is impractical to filter the power supply at every one. Variations in clock gater delay caused by voltage noise, cross-die process variation, and nonuniform temperature distribution cause skew between clocks produced by different gaters. The best way to limit this skew is to make the gater delay as short as possible. Variations in the input threshold of the clocked elements also causes skew. The best way to limit this skew is to produce crisp rise/fall times at the clock gaters. The final stage should have a fanout of no more than about 4.

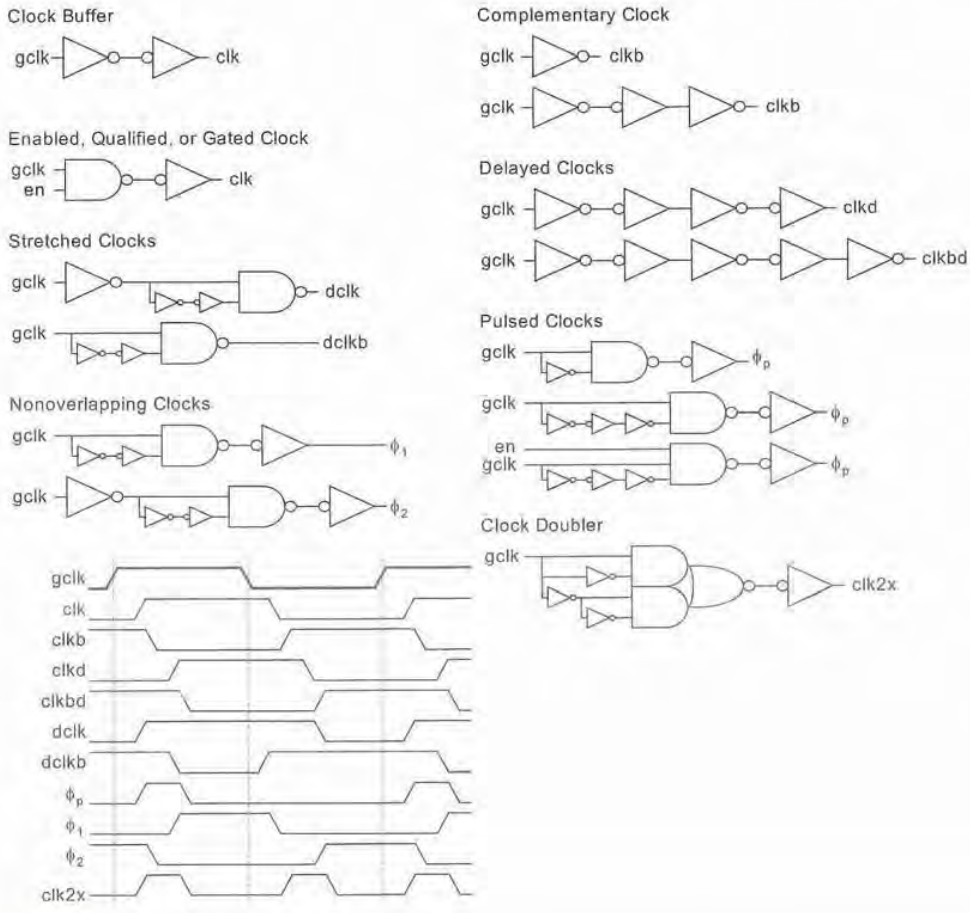


FIG 12.39 Clock gaters

Clock gaters may introduce some systematic delay between phases. For example, if *clkb* is produced with three inverters while *clk* is produced with only two, *clkb* may be delayed slightly from *clk*. The designer can either choose to carefully size the inverters such that the net delay is equal or accept that the delays are unequal and simply roll the systematic difference into timing analysis.

Figure 12.40 shows a circuit in which the delay of two inverters is matched against the delay of three when driving a fanout of *F*. The

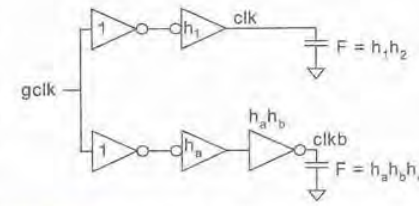


FIG 12.40 2- and 3-inverter paths

inverters are annotated with their size. The two inverters have electrical efforts of h_1 and h_2, respectively, while the three inverters have electrical efforts of h_a, h_b, and h_c. The electrical efforts should be chosen so that the delays of the chains are equal:

$$D = h_1 + h_2 + 2p_{\text{inv}} = h_a + h_b + h_c + 3p_{\text{inv}} \tag{12.5}$$

Even if the inverters have equal rise and fall delays in the TT corner, they will have unequal delays in the FS or SF corner. This can lead to skew between *clk* and *clkb* in these corners. However, if the delay of the second inverter in each chain is equal ($h_2 = h_b$), the two gaters will have equal delay in all process corners [Shoji86].

We can solve for the best electrical efforts that satisfy this constraint while giving least delay through the path. Recall that a path has least delay when its stage efforts are equal. Thus, choose $h_a = h_c = h^*$. This implies $h_1 = h^{*2}$. The delay of the first inverter in the *clk* path must equal the sum of the delays of the first and third inverter in the *clkb* path:

$$h^{*2} + p_{\text{inv}} = 2h^* + 2p_{\text{inv}} \tag{12.6}$$

This gives a quadratic equation that can be solved for h*:

$$h^* = 1 + \sqrt{1 + p_{\text{inv}}} \tag{12.7}$$

For $p_{\text{inv}} = 1$, this implies the best stage efforts are:

$$\begin{aligned} h_1 &= 5.8 \quad h_2 = \frac{F}{5.8} \\ h_a &= 2.4 \quad h_b = \frac{F}{5.8} \quad h_c = 2.4 \end{aligned} \tag{12.8}$$

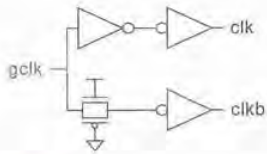


FIG 12.41 2- and 1-inverter paths

In this case, the rise/fall times of the different stages may be rather different, so the logical effort delay model is not especially accurate. These efforts make a good starting point, but further tuning should be done with a circuit simulator. The same approach can be used when the gater uses a NAND gate in place of one of the inverters.

Another approach is to try to match the delay of two inverters against one inverter and a transmission gate, as shown in Figure 12.41. This matching will not be perfect across all process corners. However, the gater may have less overall delay and hence produce less jitter from power supply noise.

12.5.6 Clock Skew Budgets

Developing an appropriate clock skew budget for design is a tricky process. The designer has a number of choices, including ignoring clock skew, budgeting worst-case clock skew everywhere, or budgeting different amounts of clock skew between different clock domains. Ultimately, the designer's objective is to build a system that achieves perfor-

mance targets and has no hold time failures while consuming as little area, power, and design effort as possible. The performance target can be a fixed number set by a standard or can simply be "as fast as possible."

It is possible to ignore clock skew if you are conservative about hold times and simply want the system to run as fast as possible. You must take reasonable care in the clock distribution network so that the skew between back-to-back flip-flops is unlikely to be too large. Many ASIC and FPGA flip-flops are designed with long contamination delays so they can tolerate significant skew before violating hold times. Build the system to run as fast as possible. When it is manufactured, clock skew will cause it to run slower than expected. The advantage of this methodology is that designers can be more productive because they do not need to think about clock skew. A disadvantage is that it uses slow flip-flops. Another drawback is that some paths really will have more skew than others. If all paths are designed to have equal delay, the paths with more skew will limit performance, while the other paths will be overdesigned and will consume more area and power than necessary. Moreover, if skew-tolerant circuit techniques are used in some places but not others, the non-tolerant circuits will tend to form the critical paths.

A related approach is to estimate the worst-case clock skew and budget it everywhere. In systems using only flip-flops, this can be done by designing to a shorter clock period. For example, if an ASIC must meet a 4 ns clock period and is predicted to have 500 ps of skew, it can be designed to meet a 3.5 ns clock period with no skew. This method requires work on the part of the clock designer to predict the clock skew, but still protects most of the designers from worrying about skew.

As cycle times get shorter than about 25 FO4 inverter delays, budgeting worst-case skew everywhere makes design impossible. Instead, multiple skew budgets must be developed that reflect smaller amounts of skew between elements in a local clock domain. This method entails more thought on the part of designers to take advantage of locality and requires a static timing analyzer that applies the appropriate skew. A good timing analyzer also properly handles skew-tolerant techniques such as transparent latches and domino gates with overlapping clocks [Harris99]. Be careful—some commercial timing analyzers such as PrimeTime do not handle these circuits well.

12.5.6.1 Clock Skew Sources As discussed earlier, clock skew comes from many sources. The output of the phase-locked loop has some jitter because of noise in the PLL and jitter in the external clock source. The clock distribution network introduces more skew from variations in the buffers and wire. The buffers may have different delays because of differences in V_{DD} and temperature, as well as random variations in their channel length and threshold voltages. The wire length and loading between buffers may not be perfectly matched. Each gater drives a physical clock along a wire, so clocked elements at different ends of the wire will see different RC delays. As mentioned in Section 2.3.2, the effective gate capacitance of the clocked loads depends on the switching activity of the source and drain. For some clocked elements, this causes significant data dependence in the clocked capacitance and the local wire delay.

For hold time checks, we are concerned with the skew between two consecutive clocked elements at a particular moment in time. For setup time checks, we are concerned

with the skew between elements from one cycle to the next. Jitter in the clock distribution network can affect the instantaneous clock period, so setup time skew budgets must include the cycle-to-cycle jitter of the entire clock distribution system even for elements in the same local clock domain. Hence, we can define separate clock skew budgets for setup time and hold time analyses.

The sources can be categorized as systematic, random, drift, and jitter. Recall that systematic skews can be modeled as extra delay and taken out of the skew budget if you are willing to do the modeling. Good clock distribution networks have close to zero systematic skew. Systematic and random skews can also be eliminated by calibrating delay lines, as will be discussed in Section 12.5.7. Drift occurs slowly enough that it can be eliminated by periodic recalibration of the delay lines. Ultimately, jitter is the most serious source of skew because it changes too rapidly to predict and counteract.

12.5.6.2 Statistical Clock Skew Budgeting The most conservative approach to estimating clock skew is to find the worst-case value of each skew source and sum these values. A real chip is unlikely to simultaneously see all of these worst cases, so such a sum is pessimistic and makes design of high-speed chips nearly impossible.

Most skew sources do not have Gaussian distributions, so taking the root sum square of the sources is inappropriate. A better approach is to perform a Monte Carlo simulation of the different skew sources to find the likely distribution of skews. The skew budget is selected at some point in this distribution. For hold times, the skew must be budgeted conservatively because the chip will not work if a hold time is violated. For example, the hold time skew budget can be selected so that 95–99% of chips will have no hold time violations.

If the goal is to build a chip that operates as fast as possible, any fixed amount of skew that affects all paths equally is irrelevant to the designer because there is nothing to do about it from the point of view of meeting setup times. However, if different paths experience different amounts of skew, a path that sees less skew can contain more logic than a path that sees a larger skew. Moreover, a path using skew-tolerant sequencing elements can contain more logic than a path between flip-flops. Hence, it is useful to predict the median skew seen in various clock domains for the purposes of setup time analysis.

As the systematic clock skew tends to be low, most clock skew sources occur from random process variations and noise. However, critical paths also experience random process variation and noise, so some will be slower than simulation predicts while others will be faster. If the chip is tuned until many critical paths have nearly the same cycle time in simulation, it is likely that a few paths will be slower than expected in the fabricated part and will limit the chip speed. It is improbable that the paths with worst-case variations in data delay are also those affected by the worst clock skew. Hence, a Monte Carlo simulation considering both variations in delay of the data paths and clock network will predict a smaller and more realistic clock skew budget [Harris01b].

Overall, choosing the appropriate clock skew budget is an ongoing source of research and debate among designers. In practice, many design teams seem to perform some calculations, and then fudge the numbers until the clock skew budget is about 10% of the cycle time. This strategy has historically led to functional chips most of the time, but becomes

more risky as cycle times decrease. Measured clock skew numbers reported in publications are notoriously optimistic; for example, [Mule02] finds an average reported skew of 3.2% of the cycle time in recent microprocessors. Part of the reason is that measuring the worst case skew is difficult. Measurements tend to be made at only a few clocked elements for a small number of clock cycles, while the chip must be designed to operate correctly for the largest skew seen anywhere on the chip anytime during its ~10^{17} cycle life span.

12.5.6.3 Case Study: Itanium 2 Skew Budget As illustrated in Section 12.5.4, the 1.0 GHz Itaniuim 2 uses an H-tree for clock distribution. The distribution network uses four levels of buffering between the PLL and the clocked elements. These buffers are called the primary driver (PD), repeater, second-level clock buffer (SLCB), and gater. The clocks are distributed on wide, shielded upper-level metal.

Table 12.2 lists the sources of variation impacting clock skew. Two adjacent pulsed latches sharing the same gater will see much less skew than two latches on opposite corners of the chip that have only the primary driver in common. Hence, we can define a hierarchy of four clock domains characterized by whether two elements share the same gater, SLCB, repeater, or only the PD. Setup times are concerned with cycle-to-cycle jitter, while hold times are not. In summary, we will define eight distinct skew budgets for these different scenarios [Harris01b]. The sources affecting each of these budgets are indicated in the eight columns.

The dominant components in the skew budget are the PLL jitter, the voltage, temperature, channel length, and threshold variations at each buffer, and the systematic mismatches in wire length and loading. Wire capacitance and resistance variations were considered negligible in the manufacturing process. Note that certain variations such as channel length of the primary driver have no effect on clock skew because they affect all physical clocks equally.

The buffer delays are 150 ps for the primary driver and repeater, 280 ps for the SLCB, and 180 ps for a simple gater, excluding wire RC flight times. The impact of the skew sources on each of these buffer delays includes:

- **Voltage**: The power network is designed to have less than ±100 mV noise on a 1.2 V supply. This full noise can be seen between any two points on the chip or from one cycle to the next at any given point; it exhibits little temporal or spatial locality. The voltage variation leads to a 13% delay change/100 mV.
- **Temperature**: The full-chip power simulation gives a temperature map shown in Figure 6.57 that predicts a variation of 20° C across the core (excluding the caches). This causes a 1.5% delay variation.
- **Channel length**: Transistors can experience systematic variations of up to ±12.5 nm from their nominal drawn L_e of 180 nm. This leads to a ±10% delay variation.
- **Threshold voltage:** The standard deviation in threshold voltages is 16.8 mV for small nMOS transistors (< 12.5 μm wide), 14.6 mV for small pMOS, 7.9 mV for wide nMOS, and 6.5 mV for wide pMOS. Monte Carlo simulations of the clock buffers show that this leads to a distribution of delays with a standard deviation of 2%.

Table 12.2 Clock skew components

	Component	Type	Same Clock Edge (hold)				Cycle-to-cycle (setup)			
			PD	Repeater	SLCB	Gater	PD	Repeater	SLCB	Gater
PLL	PLL Jitter	Jitter					x	x	x	x
PD	Voltage	Jitter					x	x	x	x
	Temperature	Drift								
	L_e	Rand								
	V_t	Rand								
	Wire	Syst	x				x			
Repeater	Voltage	Jitter	x				x	x	x	x
	Temperature	Drift	x				x			
	L_e	Rand	x				x			
	V_t	Rand	x				x			
	Loading	Syst	x				x			
	Wire	Syst	x	x			x	x		
SLCB	Voltage	Jitter	x	x			x	x	x	x
	Temperature	Drift	x	x			x	x		
	L_e	Rand	x	x			x	x		
	V_t	Rand	x	x			x	x		
	Loading	Syst	x	x			x	x		
	Wire	Syst	x	x	x		x	x	x	
Gater	Voltage	Jitter	x	x	x		x	x	x	x
	Temperature	Drift	x	x	x		x	x	x	
	L_e	Rand	x	x	x		x	x	x	
	V_t	Rand	x	x	x		x	x	x	
	Loading	Syst	x	x	x		x	x	x	
	Wire RC	Syst	x	x	x	x	x	x	x	x

- **Wire and loading**: The worst case differences in wire length and load capacitance were found through simulation of the routed H-tree.

Table 12.3 summarizes the magnitude of each of these sources of delay variation. Some sources are described by the half-range of a uniform distribution. Others are described by a standard deviation of a Gaussian distribution.

The skew budget was selected by performing a Monte Carlo simulation. $N = 400$ chips were simulated. For each chip, the systematic, random, and drift sources of skews were assigned a random value from their distribution. Jitter sources were assigned their worst case because they vary rapidly in time and the chip must work on all cycles. The

Table 12.3 Magnitude of skew sources (ps)

	Component	Half-range	Standard Deviation
PLL	PLL Jitter	7.5	
PD	Voltage	19.5	
	Temperature	n/a	
	L_e	n/a	
	V_t	n/a	
	Wire	1	
Repeater	Voltage	19.5	
	Temperature	1.1	
	L_e	15	
	V_t		5
	Loading	5	
	Wire	1.5	
SLCB	Voltage	36.4	
	Temperature	2.1	
	L_e	28	
	V_t		5.6
	Loading	10	
	Wire	4	
Gater	Voltage	23.4	
	Temperature	1.4	
	L_e	18	
	V_t		3.6
	Loading	7.5	
	Wire RC	10	

assignments were done for every clock buffer on the chip and the skew budget for that chip for each domain was defined as the maximum skew between any two elements in that domain. This gives another distribution of skews in each domain across the N chips. The setup time skew was taken from the median of the distribution to represent a "typical" chip. The hold time skew was selected at the 95th percentile in the distribution so that nearly all chips would have no hold time problems. Table 12.4 lists the skew budgets. Jitter represents more than 200 ps of the setup time skew budgets, indicating that power supply noise on the PLL and clock buffers is a major problem for the H-tree. A more careful investigation into the actual voltage noise seen at the buffers and a better effort to filter the supply noise at the buffers could offer a substantial improvement in this jitter. Hold time skew budgets are smaller because they suffer less from jitter.

Table 12.4 Clock skew budgets (ps)

Same Clock Edge (hold)				Cycle-to-cycle (setup)			
PD	Repeater	SLCB	Gater	PD	Repeater	SLCB	Gater
280	229	106	20	312	302	267	232

A more realistic budget considers that variation takes place in the logic paths as well as the clock distribution network. Thus, some critical paths will be longer than predicted while others will be shorter. This variation is called *data skew*. The total skew impacting a path is the sum of its clock and data skew. The cycle-limiting path is the one that has the worst total skew, not necessarily the one with just the worst clock skew. Another Monte Carlo simulation incorporating data skew gives total skew budgets shown in Table 12.5. The total skew is slightly higher than the clock skew alone in most cases. However, the difference between the skew in the PD and gater delays is substantially smaller. When the design goal is to maximize frequency, the difference between global and local setup time skew is most important because it indicates how much less logic can go in a global path than in a local path if all paths should be equally critical. Also note that the hold time skew in the PD domain decreased slightly because there were relatively few hold time paths crossing this domain and it was uncommon that one with bad data skew also experienced bad clock skew.

Table 12.5 Total skew budgets (ps)

Same Clock Edge (hold)				Cycle-to-cycle (setup)			
PD	Repeater	SLCB	Gater	PD	Repeater	SLCB	Gater
275	226	117	44	322	313	309	285

12.5.7 Adaptive Deskewing

Just as a PLL or a DLL can compensate for the overall clock distribution delay, additional adjustable delay buffers can compensate for mismatches in clock distribution delay along various paths. For example, the Pentium II and 4 use such buffers at the leaves of the clock spine to eliminate systematic and random variations in the clock distribution network. Figure 12.42 shows an example of a digitally adjustable delay line with eight levels of adjustment. The select signals use a *thermometer code*[1] to produce a monotonically decreasing propagation delay as more pass transistors are turned on.

In the Pentium II, a phase comparator checks the arrival times of the physical clocks and adjusts the digitally controlled delay lines to make all clocks arrive simultaneously. The loop bandwidth is low enough to ignore jitter, but high enough to compensate for tem-

[1]In an N-bit thermometer code, a number $n \in [0,N]$ is represented with n 1's in the least significant positions. For example, the number 3 is represented in an 8-bit thermometer code as 00000111.

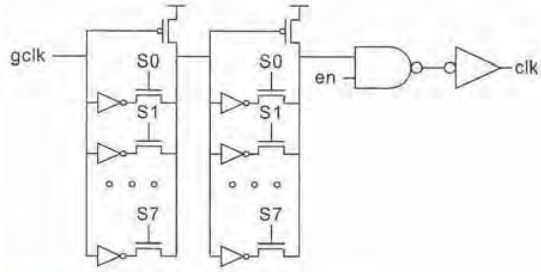


FIG 12.42 Digitally adjustable delay line

perature drift. This technique is sometimes called *adaptive deskewing* [Geannopoulos98]. In the Pentium 4, the delay line is adjusted using a scan chain through the boundary scan test access port. 46 phase comparators measure the phase of the clock gaters. Their results can be shifted out through the TAP. The delay lines can be adjusted to reduce systematic and random skew to ±8 ps, as compared to approximately 64 ps before adjustment. The delay lines can also deliberately delay certain clocks to improve performance or assist with debug [Kurd01]. The Itanium series of microprocessors uses similar deskew techniques [Tam00, Anderson02, Stinson03, Tam04]. In the 1.5 GHz Itanium 2, deskew takes place during manufacturing test; on-chip fuses are blown to eliminate the systematic and random skew without needing calibration upon reset or during normal operation.

A drawback of adaptive deskewing is that the buffers introduce extra delay. Voltage noise on the buffers appears as jitter. Unless all of the deskew buffers use well-filtered power supplies, the extra jitter from the deskew buffers can overwhelm the improvement in systematic and random skew.

12.5.8 Clocking Alternatives

A number of radical alternatives to standard clocking have been proposed. [Mule02] offers a survey of many of these methods.

Clock distribution consumes significant amounts of power and introduces much of the clock skew. On-chip metal wires have high resistance, contributing to the problems. Alternatively, the clock could be distributed on the printed circuit board or in the package using low-resistance transmission lines, and then brought onto the chip in many local regions. The clock could also be distributed optically to photodetectors around the chip or broadcast wirelessly using microwaves. Each of these methods suffers from the difficulty of testing the chip before it is packaged. If the package is expensive and the yield is not extremely high, testing chips after packaging adds significantly to the average part cost. Optical or wireless clocking also requires the development of on-chip photodetectors or antennae.

Another strategy is to distribute the clock on-chip, but use a novel architecture. Many oscillators can be distributed across the chip and operated in phase so clock distribution becomes localized [Gutnik00]. Rotary traveling-wave oscillator arrays exploit the inherent natural frequency, set by the clock network inductance and capacitance [Wood01]. If pumped at this frequency, it can oscillate by transferring the energy back and forth between magnetic and electric forms, reducing the power consumption.

Asynchronous systems eliminate the clock entirely. Proponents argue that no clocks means no clock skew and no clock power consumption. Advocates of synchronous systems point out that asynchronous systems still must distribute control signals to all the sequential elements and that variation in this delay appears as sequencing overhead in just the same way as clock skew. Moreover, the control signals dissipate power. [Sparsø01] provides a good tutorial introduction to some of the advantages of, challenges in, and techniques for asynchronous design. Synchronous designers have borrowed many techniques such as self-timed memories, self-resetting domino, and source-synchronous clocking that might once have been considered asynchronous. The debate has raged for decades, but nearly all commercial systems are still synchronous and will likely remain so indefinitely.

12.6 Analog Circuits

Although the emphasis of this book has been on digital circuits, system-on-chip designers commonly need to use some analog or radio-frequency (RF) circuitry to interface with the real world. This section offers a brief introduction to the subject of analog circuit design for digital designers. [Baker02, Gray01, Johns96] offer excellent coverage of CMOS analog circuit design while [Lee04, Razavi98] pioneered RF circuits in CMOS. The combination of analog and digital circuitry is naturally called *mixed-signal* design.

12.6.1 MOS Small-signal Model

Although the MOS transistor is a nonlinear device, it can be approximated as linear for small changes around a bias point. This is useful for understanding the behavior of amplifiers and other analog circuits. The currents and voltages can be written as

$$\begin{aligned} V_{gs} &= V_{GS} + v_{gs} \\ V_{ds} &= V_{DS} + v_{ds} \\ I_{ds} &= I_{DS} + i_{ds} \end{aligned} \tag{12.9}$$

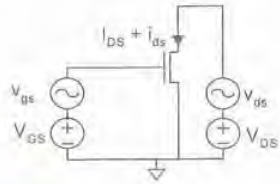


FIG 12.43 Small-signal variations around bias point

where the gate source voltage V_{gs} is expressed as a *bias-point* voltage V_{GS} plus a *small-signal* offset v_{gs}, as shown in Figure 12.43. The drain current also is expressed as a bias-point current I_{DS} plus a small-signal offset i_{ds} proportional to v_{gs} and v_{ds}.

MOS transistors are typically used in their saturation region in analog circuits. By expanding the saturation current model of EQ (2.8) around the operat-

ing point, we find the dependence of output current on small changes in input voltage.

$$\begin{aligned} I_{ds} &= \frac{\beta}{2}\left(V_{gs} - V_t\right)^2 \\ &= \frac{\beta}{2}\left(V_{GS} + v_{gs} - V_t\right)^2 \\ &= \frac{\beta}{2}\left[\left(V_{GS} - V_t\right)^2 + 2\left(V_{GS} - V_t\right)v_{gs} + v_{gs}^2\right] \\ &= \underbrace{\frac{\beta}{2}\left(V_{GS} - V_t\right)^2}_{I_{DS}} + \underbrace{\beta\left(V_{GS} - V_t\right)v_{gs} + O\left(v_{gs}^2\right)}_{i_{ds}} \end{aligned} \tag{12.10}$$

If v_{gs} is small enough, the $O(v_{gs}^2)$ term is negligible.

In general, we can find the sensitivity of current to small changes in voltage by taking the first-order Taylor series around the operating point:

$$I_{ds} \approx I_{DS} + \underbrace{\left.\frac{\partial I_{ds}}{\partial V_{gs}}\right|_{V_{gs}=V_{GS}} v_{gs} + \left.\frac{\partial I_{ds}}{\partial V_{ds}}\right|_{V_{ds}=V_{DS}} v_{ds}}_{i_{ds}} \tag{12.11}$$

We commonly write the sensitivities as

$$i_{ds} = g_m v_{gs} + g_{ds} v_{ds} \tag{12.12}$$

where

$$g_m = \left.\frac{\partial I_{ds}}{\partial V_{gs}}\right|_{V_{gs}=V_{GS}} = \beta\left(V_{GS} - V_t\right) \tag{12.13}$$

and because the saturation current is ideally independent of V_{ds},

$$g_{ds} = \left.\frac{\partial I_{ds}}{\partial V_{ds}}\right|_{V_{ds}=V_{DS}} = 0 \tag{12.14}$$

This Taylor series approach gives the same results as the direct expansion in EQ (12.10).

In a real MOSFET, the output current does increase with V_{ds} because of channel length modulation. Considering EQ (2.29),

$$g_{ds} = \left.\frac{\partial I_{ds}}{\partial V_{ds}}\right|_{V_{ds}=V_{DS}} = \lambda\beta\frac{\left(V_{GS} - V_t\right)^2}{2} = \lambda I_{DS} \tag{12.15}$$

g_m is called the *transconductance* because it reflects the dependence of the drain current on the gate voltage. g_{ds} is the *output conductance*, reflecting the dependence of the drain current on the drain voltage. Here λ is the channel length modulation coefficient, not the unit of distance. Recall that λ is inversely dependent on channel length. Current sources and high-gain analog amplifiers require low output conductance and thus often use longer than minimum transistors. Often it is convenient to think about the reciprocal, *output resistance.*

$$r_0 = r_{ds} = \frac{1}{g_{ds}} = \frac{1}{\lambda I_{DS}} \tag{12.16}$$

Figure 12.44 shows the I-V characteristics of an nMOS transistor annotated with the bias-point and small-signal parameters. The transistor is biased in saturation at $V_{GS} = 0.9$, $V_{DS} = 1.0$. The transconductance and output conductance are the slope of the I_{ds} curve with respect to small changes in V_{gs} and V_{ds} around the operating point.

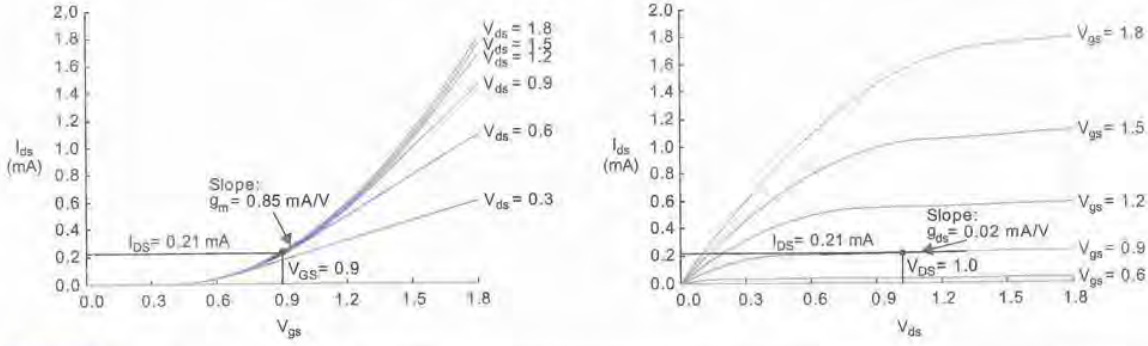


FIG 12.44 Bias-point and small-signal behavior

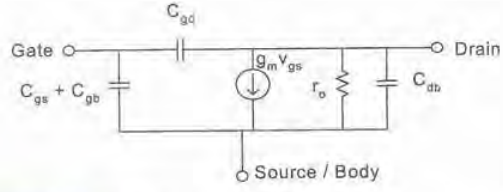


FIG 12.45 Small-signal model for an MOS transistor

Assuming the body is at the same potential as the source, we can model the transistor with the small-signal equivalent circuit of Figure 12.45 to relate small-signal voltages and currents. The current source reflects the dependence of i_{ds} on v_{gs} and the resistor reflects the dependence of i_{ds} on v_{ds}. The capacitors can be considered for high-frequency operation or ignored for low-frequency operation. The results of the small-signal model are added to the results of the bias point to find the overall behavior of the circuit.

12.6.2 Common Source Amplifier

Figure 12.46 shows a common source amplifier with a resistive load. The amplifier is biased at V_{GS} = 0.9 V, setting an output bias point V_{OUT}. If we increase V_{gs} by some small v_{in}, the transistor will turn on harder, pulling the output down by some small v_{out}. The gain of the amplifier is thus $A = v_{out} / v_{in}$.

Figure 12.47 shows the simulated DC transfer characteristics of the common source amplifier. The bias point is indicated on the graph. The slope of the transfer characteristic around the operating point is the gain. The transistor is a nonlinear device so its gain varies with the operating point. However, the slope and gain are relatively constant for modest values of v_{in}.

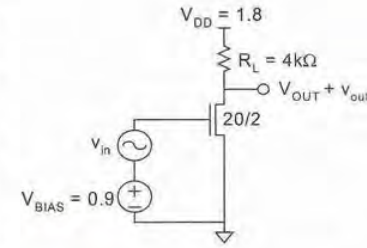


FIG 12.46 Common source amplifier

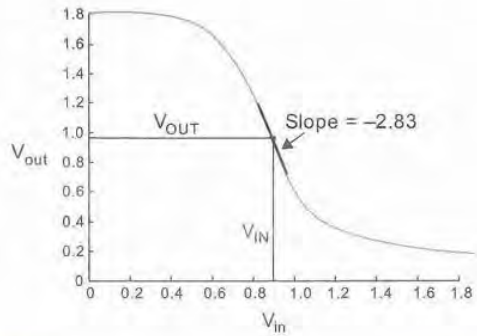


FIG 12.47 DC transfer characteristics of common source amplifier

Example

Calculate the bias point V_{OUT} and small-signal low-frequency gain if V_t = 0.4, β = 1550 μA/V² and the nMOS output impedance is infinite.

Solution: At the bias point of V_{GS} = 0.9, $I_{DS} = 1550 \cdot 10^{-6} \cdot (0.9 - 0.4)^2/2 = 193\ \mu A$[2] and thus the output voltage is $V_{OUT} = 1.8 - I_{DS}R_L = 1.03$ V. Figure 12.48 shows a small-signal equivalent circuit around this bias point. The model omits the capacitors because they act as open circuits at low frequency and also leaves out r_o because it is infinite. The load resistor ties to a small-signal ground because V_{DD} is constant. According to EQ (12.13), $g_m = 1550 \cdot 10^{-6} \cdot (0.9 - 0.4) = 0.77$ mA/V. Using Kirchhoff's current law at the output node shows $v_{out} = -g_m R_L v_{in}$, or the gain $A = v_{out}/v_{in} = -g_m R_L = -3.1$.

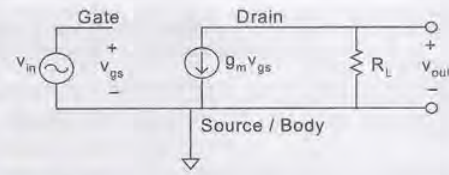


FIG 12.48 Small-signal model of common source amplifier

[2]Note that this differs slightly from Figure 12.44 because channel length modulation is neglected.

Example

Repeat the previous example if the transistor has a channel length modulation coefficient of $\lambda = 0.1\ V^{-1}$.

Solution: The bias point changes only slightly on account of channel length modulation. Assuming from the previous example that $V_{DS} \sim 1.0$, we find $I_{DS} = 1550 \cdot 10^{-6} \cdot (0.9 - 0.4)^2 \cdot (1 + 1.0 \cdot 0.1)/2 = 212\ \mu A$. The output voltage is now $V_{OUT} = 1.8 - I_{DS}R_L = 0.95$, justifying our assumption. The output resistance is $r_o = 1/(0.1 \cdot 212\ \mu A) = 47\ k\Omega$. Figure 12.49 shows a small-signal equivalent circuit around this bias point including r_o. Now the gain depends on the parallel combination of the output and load resistances, $A = -g_m(R_L \,||\, r_o) = -2.83$, where the parallel combination of two resistors is

$$R_1 \,||\, R_2 \equiv \frac{R_1 R_2}{R_1 + R_2} \tag{12.17}$$

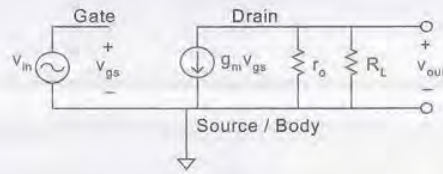


FIG 12.49 Small-signal model of common source amplifier including output resistance

The pull-up resistor can be built as a pMOS transistor of width P operating in its linear region, as shown in Figure 12.50. The gain now depends on the effective resistance of the pMOS transistor. This is simply a pseudo-nMOS inverter. Its DC transfer characteristics were discussed in Section 2.5.4.

12.6.3 The CMOS Inverter as an Amplifier

We can increase the gain of the common source amplifier by using an active load that turns ON when the nMOS transistor turns OFF and OFF when the nMOS turns ON. This can be done by building the load from a pMOS transistor connected to the input, as shown in Figure 12.51. Our high-gain amplifier is simply a CMOS inverter. Hence, the CMOS inverter is an analog amplifier operated under saturating conditions. It can also be viewed as an nMOS common-source amplifier driving a pMOS common-source amplifier. Near the input threshold voltage, the CMOS inverter acts as an inverting linear amplifier with a characteristic of

$$v_{out} = -Av_{in} \tag{12.18}$$

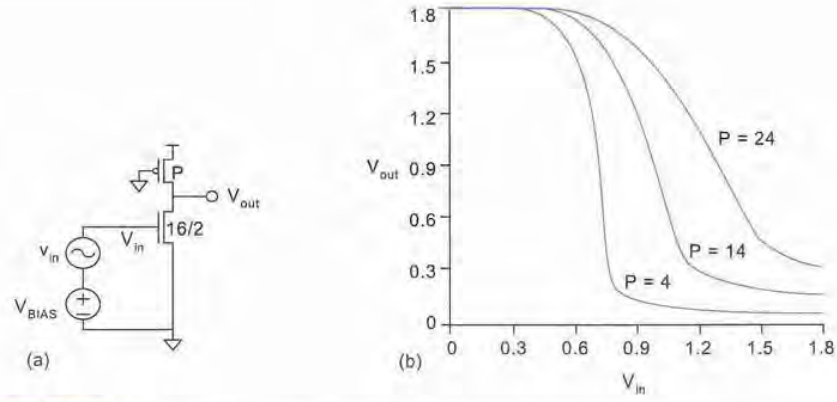


FIG 12.50 Pseudo-nMOS inverter viewed as common source amplifier with pMOS load

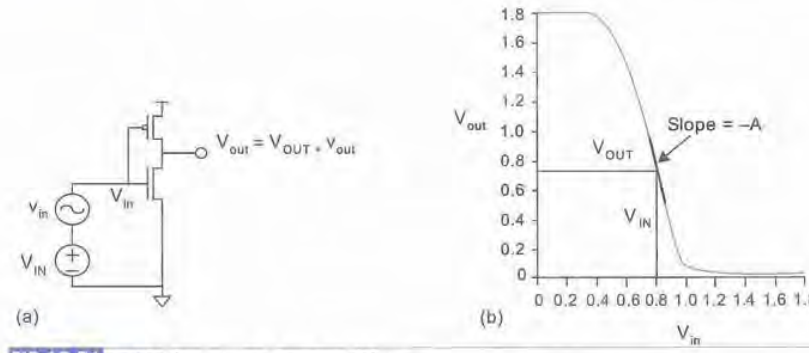


FIG 12.51 CMOS inverter as an amplifier

where A is the amplifier gain.

We can further examine this region with a circuit simulator by using the circuit shown in Figure 12.52 with a high-value resistor (10 MΩ) between input and output to *DC bias* the inverter at V_{inv}. The input is *AC coupled* using a capacitor. The gain of this amplifier is estimated using the small-signal transistor model from Figure 12.45 to construct an equivalent circuit (Figure 12.53) valid for small-signal swings around the linear operating point of the amplifier. The gain is approximately given by

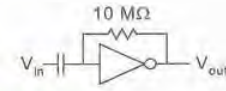


FIG 12.52 AC-coupled CMOS inverter

$$\begin{aligned} A &= g_{m-\text{total}} r_{o-\text{effective}} \\ &= \left(g_{mn} + g_{mp}\right)\left(r_{on} \mid r_{op} \mid 10\text{M}\Omega\right) \\ &\approx g_m r_o \ (\text{if } g_{mn} = g_{mp} \text{ and } r_{on} = r_{op} << 10\text{M}\Omega) \end{aligned} \tag{12.19}$$

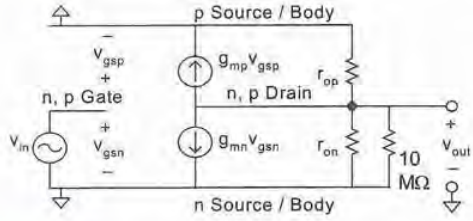


FIG 12.53 Small-signal model of amplifier

The gain depends on the channel length modulation. We can estimate it as

$$g_m = \beta\left(V_{gs} - V_t\right)$$
$$r_o = \frac{1}{\lambda I_{ds}} = \frac{1}{\lambda\left(\frac{\beta}{2}\right)\left(V_{gs} - V_t\right)^2} \tag{12.20}$$
$$A \approx g_m r_o\big|_{V_g = \frac{V_{DD}}{2}} = \frac{4}{\lambda\left(V_{DD} - 2V_t\right)}$$

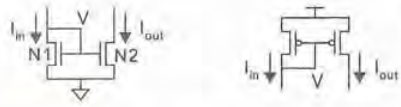


FIG 12.54 Current mirrors

12.6.4 Current Mirrors

The *current mirrors* in Figure 12.54 replicate the input current at the output. The voltage V adjusts itself to the correct level to sink or source I_{in} through $N1$. This voltage also controls the gate of $N2$. If both transistors operate in saturation where I_{ds} depends only on the gate voltage, not the drain voltage, then $I_{out} = I_{in}$. Such an ideal current source has infinite output impedance because the current is independent of the output voltage.

Real devices suffer from channel length modulation and have a finite output resistance. This output resistance makes I_{out} vary somewhat with the drain voltage on $N2$. Good current mirrors use long-channel transistors to achieve high output resistance and nearly constant I_{out}. The output impedance at the operating point is:

$$R_{out} = \frac{v_{out}}{i_{out}} \tag{12.21}$$

This can be found by applying a test voltage v_{out} to the small signal model and computing the current i_{out}. Figure 12.55 shows a small-signal model of the current mirror with the test source. Observe that the small-signal gate voltage v is pulled to ground through the parallel combination of g_{m1} and r_{o1}, indicating that the gate voltage will not stray from the bias point. Applying Kirchhoff's current law (KCL) on the drain of $N2$ shows

$$i_{out} - \frac{v_{out}}{r_{o2}} = 0 \tag{12.22}$$

so the output impedance is $R_{out} = r_{o2}$.

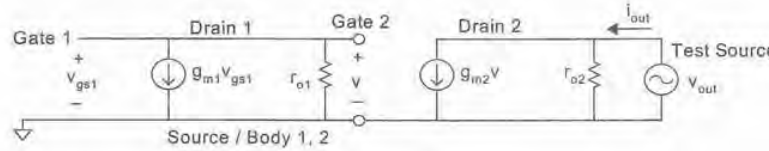


FIG 12.55 Small-signal model for current mirror output impedance

Example

Figure 12.56(a) shows a simple current source constructed from a resistor and a current mirror. It can be modeled as an ideal current source in parallel with a finite output resistance, as shown in Figure 12.56(b). Find I_{out}, the current source output impedance R_{out}, and the range of V_{out} over which the current source operates correctly. Again assume $V_t = 0.4$, $\beta = 1550\ \mu A/V^2$, and $\lambda = 0.1\ V^{-1}$.

Solution: The effect of channel length modulation on bias point is small, so we will neglect it in our analytical solution. We find the bias point by solving the nonlinear equation for input current $I_1 = 103\ \mu A$ at $V_1 = 0.765$.

$$I_1 = \frac{\beta}{2}(V_1 - V_t)^2 = \frac{\beta}{2}(1.8 - I_1 R_1 - V_t)^2 \qquad (12.23)$$

At the bias point, $I_{out} = I_1$ and $R_{out} = r_o = 1/(0.1 \cdot I_1) = 97\ k\Omega$. The current source operates correctly as long as the output transistor remains in saturation. This is true as long as $V_{out} > V_1 - V_t = 0.365$ V. A good current source should have high—ideally infinite—output impedance.

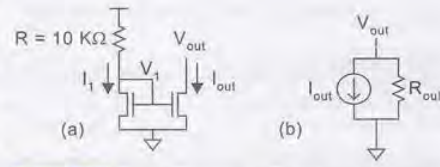


FIG 12.56 Simple current source

The output impedance can be raised by using a *cascoded* current mirror shown in Figure 12.57. In this design, V_1 and V_2 adjust to whatever they must be to sink I_{in} through $N1$ and $N3$. As $I_{out} \approx I_{in}$, the gate-to-source voltages for $N3$ and $N4$ must be nearly equal. Hence $V_3 \approx V_1$. Thus, $N1$ and $N2$ have nearly the same drain voltage as well as gate voltage, making their currents nearly equal even if the transistors have significant output conductance. By KCL, $N4$ must have the same current as $N2$, so I_{out} is very weakly sensitive to V_{out}. In other words, the cascoded current mirror has high output impedance.

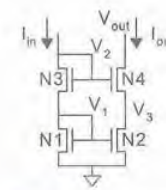


FIG 12.57 Cascoded current mirror

Figure 12.58 shows a small-signal model for the cascoded current mirror[3]. The output impedance increases to (see Exercise 12.5)

$$R_{out} = r_{o2} r_{o4} \left[\frac{1}{r_{o2}} + \frac{1}{r_{o4}} + g_{m4} \right] \approx (g_{m4} r_{o2}) r_{o4} \qquad (12.24)$$

[3]Note that we ignore the body effect in this and subsequent examples. The body effect leads to another transconductance element g_{mb} between drain and source dependent on V_{bi}.

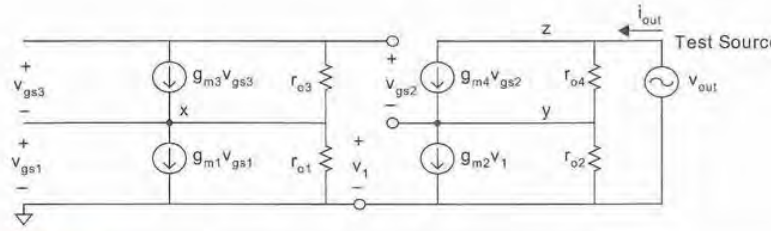


FIG 12.58 Small-signal model for cascode output impedance

Example

Find the output impedance of a cascoded current mirror using the same parameters and bias current as in the previous example.

Solution: Each transistor operates at the same current I_{out} = 103 μA and has the same output resistance r_o = 97 kΩ. At this current, g_m = 1550 • 10^{-6} • (0.765 − 0.4) = 0.57 mA/V. Thus, the cascoded current mirror has an output impedance of R_{out} = (0.57 • 10^{-3})(97 • 10^3)^2 = 5.3 MΩ, more than 50 times that of an ordinary current mirror.

Current mirrors can use multiple output transistors to create multiple copies of an input current. The mirror can also multiply a current by N by using an output transistor N times as wide as the input, as shown in Figure 12.59. Better yet, the output can be driven by N identical transistors in parallel because identical transistors match more closely.

FIG 12.59 Current mirrors with multiple outputs and with current gain

12.6.5 Differential Pairs

A differential pair steers current to two outputs. In Figure 12.60, the current I_{ref} is divided between the two outputs depending on the difference between the two input voltages. If the input voltages are equal, the output currents are equal. If one input is substantially higher than the other, it draws all of the current. *Common mode* noise that affects both inputs equally causes no change in the output current. Hence, differential pairs are widely

used because they are insensitive to many noise sources. For this reason, differential pairs are used in sense amplifiers on RAM bitline circuitry.

Differential pairs are easiest to analyze by finding the input voltage difference from the output current difference rather than vice versa. In analog circuits, the transistors normally operate in saturation. We define $V_{go} = V_{gs} - V_t$ as the *gate overdrive* of a transistor. An ideal transistor would deliver saturation current proportional to the square of the gate overdrive. According to the α-power law model, we can estimate the current of a real transistor with velocity saturation as

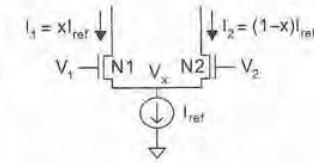


FIG 12.60 Differential pair

$$I_{ds} = kV_{go}^{\alpha} \tag{12.25}$$

for some $1 < \alpha < 2$.

When $V_1 = V_2$, the amplifier is in its balanced condition and $I_1 = I_2 = I_{ref}/2$. Both transistors have some gate overdrive V_{go}. We use this to express k in terms of I_{ref} and V_{go}, so we will be able to eliminate it from subsequent equations.

$$\frac{I_{ref}}{2} = kV_{go}^{\alpha} \Rightarrow k = \frac{I_{ref}}{2V_{go}^{\alpha}} \tag{12.26}$$

We would like to know how much differential voltage ΔV is required between V_1 and V_2 so that $N1$ draws xI_{ref} and $N2$ draws $(1-x)I_{ref}$. If we increase the gate overdrive of $N1$ by some fraction δ_1, its current becomes

$$xI_{ref} = k\left[V_{go}(1+\delta_1)\right]^{\alpha} \tag{12.27}$$

Substituting EQ (12.26), we can solve for the δ_1 required to obtain x

$$\begin{aligned} xI_{ref} &= \frac{I_{ref}}{2V_{go}^{\alpha}}\left[V_{go}(1+\delta_1)\right]^{\alpha} \\ \delta_1 &= (2x)^{\frac{1}{\alpha}} - 1 \end{aligned} \tag{12.28}$$

Similarly, the gate overdrive of $N2$ must go down by some fraction δ_2.

$$\delta_2 = 1 - \left(2(1-x)\right)^{\frac{1}{\alpha}} \tag{12.29}$$

Combining EQ (12.28) and (12.29) shows that the fractional currents will be x and $1-x$ when a differential voltage ΔV is applied.

$$\Delta V = \left[\delta_1 + \delta_2\right]V_{go} = \left[(2x)^{\frac{1}{\alpha}} - \left(2(1-x)\right)^{\frac{1}{\alpha}}\right]V_{go} \tag{12.30}$$

This relationship is plotted in Figure 12.61 for several values of α. Values of α closer to two give higher gain. In any case, if one voltage is up and the other down by an overdrive, the second transistor will be completely OFF. Hence, a voltage differential of $2V_{go}$ is always enough for one transistor to hog all the current.

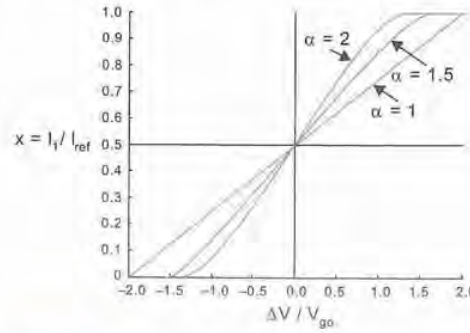


FIG 12.61 Differential pair transfer characteristics

Taking the derivative of EQ (12.30) around $x = 1/2$ and manipulating gives the small-signal transconductance of the differential pair (see Exercise 12.11).

$$\left.\frac{\partial I_1}{\partial \Delta V}\right|_{\Delta V=0} = \frac{\alpha}{4}\frac{I_{ref}}{V_{go}} = \frac{g_m}{2} \tag{12.31}$$

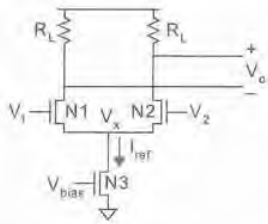


FIG 12.62 Differential amplifier

Figure 12.62 shows a differential amplifier. The amplifier consists of a differential pair (*N*1 and *N*2) driving a resistive load. The differential pair uses a transistor *N*3 with an adjustable bias voltage as the current source; this voltage could be set by a current mirror elsewhere. The high impedance load is often built from a pMOS current mirror because it is much more compact than large passive resistors.

The output impedance is $R_{out} = (R_L \,||\, r_o)$. Because the output is taken differentially, the voltage gain is twice that of each half.

$$A = \frac{V_o}{\Delta V} = 2\frac{\partial I_1}{\partial \Delta V} R_{out} = g_m\left(R_L \,||\, r_o\right) \tag{12.32}$$

An ideal differential amplifier is sensitive only to the voltage difference between the two inputs, not the average (*common mode*) input voltage. A real amplifier works well only while all the transistors remain in saturation. The gain drops off when one of the transistors enters the linear region. This means the headroom available is diminishing as V_{DD} shrinks.

12.6.6 Simple CMOS Operational Amplifier

Figure 12.63 shows a simple CMOS operational amplifier. It consists of a differential amplifier followed by a common source amplifier to achieve high gain. The differential amplifier uses a pMOS current mirror as a load to get high impedance r_{op2} in a compact area. The output is taken single-ended to a pMOS common source amplifier $P3$ loaded by nMOS current mirror $N5$. $N3$ and R set the bias voltage and current for the op-amp.

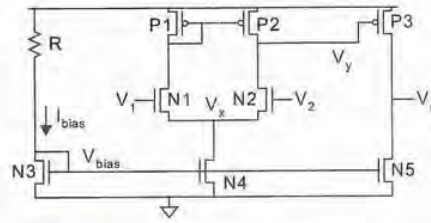


FIG 12.63 Two-stage CMOS operational amplifier

The small-signal gain of the differential stage is computed from EQ(12.32).

$$A_{\text{diff}} = \frac{v_y}{v_1 - v_2} = g_{mn2}\left(r_{on2} \,||\, r_{op2}\right) \tag{12.33}$$

The small-signal gain of the common source amplifier is:

$$A_{\text{commonsource}} = \frac{v_o}{v_y} = -g_{mp3}\left(r_{op3} \,||\, r_{on5}\right) \tag{12.34}$$

Hence, the overall gain of the amplifier is:

$$A = \frac{v_o}{v_2 - v_1} = g_{mn2}g_{mp3}\left(r_{on2} \,||\, r_{op2}\right)\left(r_{op3} \,||\, r_{on5}\right) \tag{12.35}$$

This operational amplifier works well as a comparator. It senses a small difference between the two inputs and drives V_o high or low depending on which input is higher. It is only suited to driving capacitive loads; it does not deliver enough current to drive resistive loads well. It also has a limited common-mode input range to keep all of the transistors in saturation. If the circuit is used in a feedback application, a compensation capacitor may need to be connected between V_y and V_o to ensure stability. [Baker98] and [Gray01] describe op-amp design and frequency response in much more detail.

12.6.7 Digital-to-analog and Analog-to-digital Converter Basics

Digital-to-analog converters (DACs) are relatively easy to design if the target resolution and speed are moderate. Speed, linearity, power dissipation, size, and ease of design are of importance when selecting a DAC architecture. Analog-to-digital converters (ADCs) are more of a challenge, but converters with low speed and precision can be implemented quite easily. The descriptions given in the following two sections can form the start of investigations if implementation is envisaged.

DACs and ADCs are sampled data systems. As such, some more circuits are normally required apart from the basic converter. Figure 12.64 illustrates a DAC in use in a system.

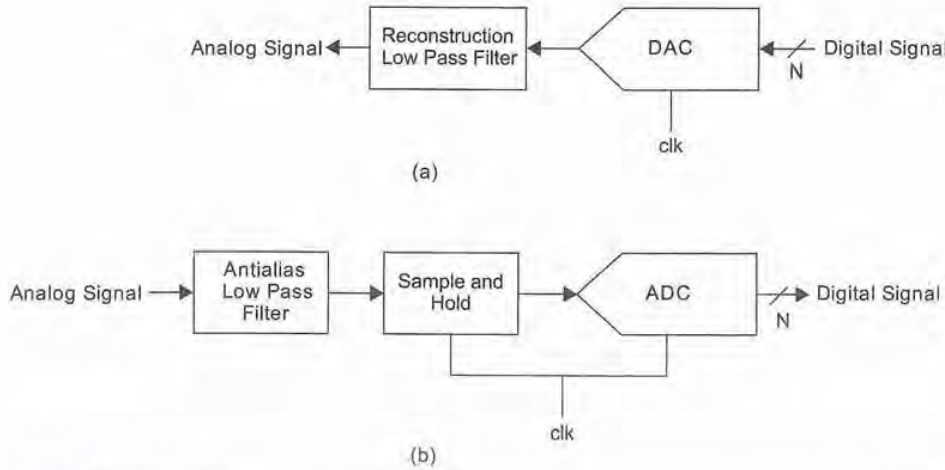


FIG 12.64 DAC and ADC in system

The DAC is followed by a reconstruction filter that converts the quantized DAC output to a smoothed analog value. The complete ADC circuit is similar, but converts from analog to digital. It also has a low-pass filter on the input and usually a sample and hold circuit to hold the input while the ADC does a conversion. The purpose of these blocks will become more apparent as the next two sections unfold.

Before discussing DACs and ADCs in detail, a few DAC metrics will be explained [Hoeschele94]. The parameters are also applicable to ADCs, except rather than a specification on a digital-to-analog transformation, the ADC parameters refer to the deviation from an ideal quantized analog-to-digital transformation.

12.6.7.1 Resolution and Full-scale Range The first parameter of interest in a DAC (or ADC) is the *resolution*. This specifies how many individual quantized steps the DAC possesses. For instance, an N-bit DAC has 2^N individual steps. Thus, an 8-bit DAC/ADC has 256 steps. The *full-scale range* (FSR) is the maximum output voltage (or current) of the DAC/ADC. The resolution (R) of the DAC/ADC is given by $R = FSR/2^N$. So for a 1 V FSR 8-bit DAC/ADC, the resolution is roughly 4 mV while a 10-bit DAC/ADC would have a resolution of approximately 1 mV. The FSR is related to the voltage supply. In the case of a DAC or ADC, the FSR can not be greater than V_{DD} and common implementations have FSRs of the order of V_{DD} minus one or two V_t drops. As processes scale and V_{DD} decreases, analog performance becomes more difficult to achieve because the resolution for a given size converter decreases in relation to any noise that might be present.

12.6.7.2 Linearity A primary concern with a DAC/ADC is the linearity or accuracy of transformation from a digital code to a quantized analog value. Linearity of a DAC/ADC

is determined by component linearity, where the components in CMOS are transistors, resistors, and capacitors. It is also affected by the introduction of unwanted signals that are classified as noise. Static (DC) linearity is normally calculated using measures called the *Integral Nonlinearity* (INL) and *Differential Nonlinearity* (DNL).

- DNL: The departure from one LSB in transitioning from one digital code to the next. For instance, in Figure 12.65(a) the DNL highlighted is approximately 0.5 least significant bits (LSB) for the 3-bit DAC shown.
- INL: The maximum deviation from a straight line drawn between the endpoints of the DAC (ADC) output (input) characteristic. Again, the INL illustrated is around 0.5 LSB.
- Offset: The difference between (nominally) zero and the actual value when the digital code for zero is applied. This is a fraction of an LSB in the figure.

A good DAC is monotonic; i.e., for each increase (or decrease) in digital code input the DAC, the analog output increases (or decreases) in value. Correspondingly, for each increase in analog input, an ADC would expect to see an increase in the digital code. A non-monotonic DAC step is shown in Figure 12.65(b), where the analog output decreases to an increase in the applied digital code. Depending on the application, this may be undesirable.

Typical INL and DNL plots are shown in Figure 12.66. These are usually measured at DC. In the plots, the INL is +0.2/−0.25 LSBs and the DNL is ±0.11 LSBs.

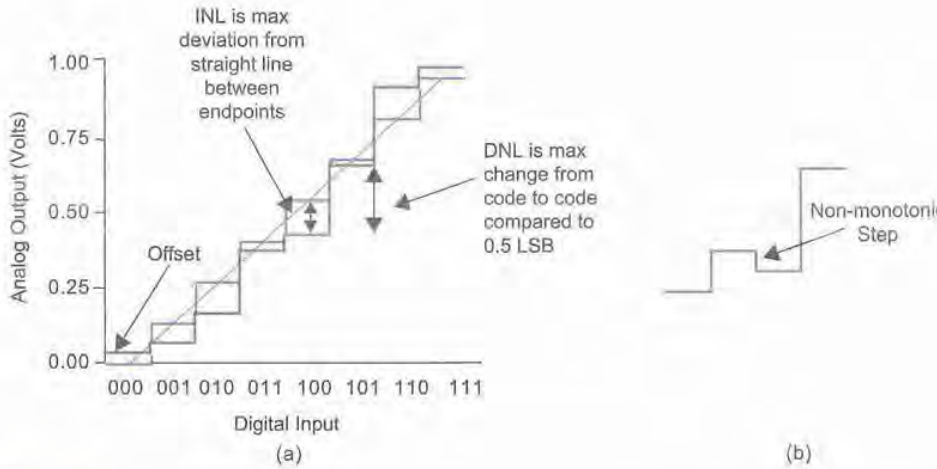
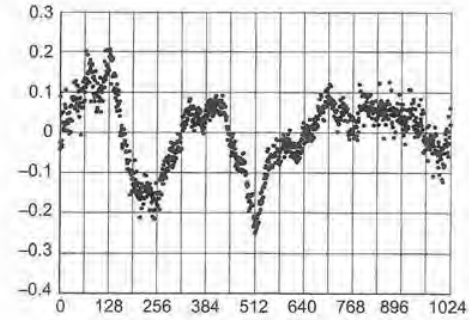
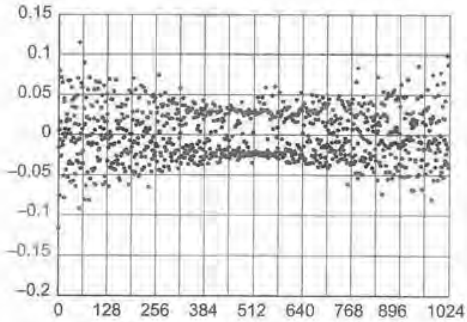


FIG 12.65 DAC linearity measures



(a) DAC INL of analog output versus digital input code



(b) DAC DNL of analog output versus digital input code

FIG 12.66 Typical DAC INL and DNL plots

12.6.7.3 Noise and Distortion Measures With a (digital) sine wave applied to the DAC, the ratio of the desired signal energy to harmonics and noise is called the *Signal-to-Noise Ratio* (SNR). In practice, to measure this in a DAC, the DAC is fed a digital sine wave from a numerically controlled oscillator. The DAC is low-pass filtered, as shown in Figure 12.64, and then fed to a spectrum analyzer. The undesired signals are then apparent.

In comparison, an ADC measures this by applying a high-quality sine wave to the ADC and doing an FFT on the stored digital samples for a number of cycles of the input.

In addition, dynamic measures such as the *Total Harmonic Distortion* (THD) and *Spurious Free Dynamic Range* (SFDR) are also of interest. The THD measures all unwanted harmonic content and expresses this as a ratio of desired-to-undesired outputs. For instance, in a DAC, for a desired frequency of f_{signal}, there may be significant second harmonic ($2f_{signal}$) and third harmonic ($3f_{signal}$) noise caused by device non-linearity. These unwanted harmonics distort the required signal and degrade the THD.

SNR and THD are classical measures of analog linearity, which originally were applied to totally analog systems. A more popular measure with DACs and ADCs that takes into consideration digital artifacts is the *spurious free dynamic range* (SFDR). The SFDR is a measure over a specified frequency range of the desired output versus unwanted signals. These include noise from sources such as switches and noise injected from other sources on chip. SNR, THD, and SFDR are measured in dB (decibels).

The *Intermodulation Distortion* (IMD) is measured by outputting two simultaneous sine waves and measuring the amplitude of spurious products. This is a measure of the extent of unwanted multiplication products that may not show up with a simple single frequency (tone) test. ADCs measure this by applying two analog sine waves to the ADC and analyzing the digital output.

Figure 12.67 shows a typical DAC spectrum. The harmonics and SFDR are shown. The "fuzz" is typical of sampled data systems.

The noise and distortion values can be transformed into a single value called the *Effective Number of Bits* (ENOB), which is defined by:

$$\text{ENOB} = (\text{SNR} - 1.76) / 6.02$$

This rolls all performance numbers into a single value, which can be used to compare DAC implementations. The effective SNR can also be evaluated from an ENOB number.

$$\text{SNR} = \text{ENOB} \cdot 6.02 + 1.76 \text{ dB}$$

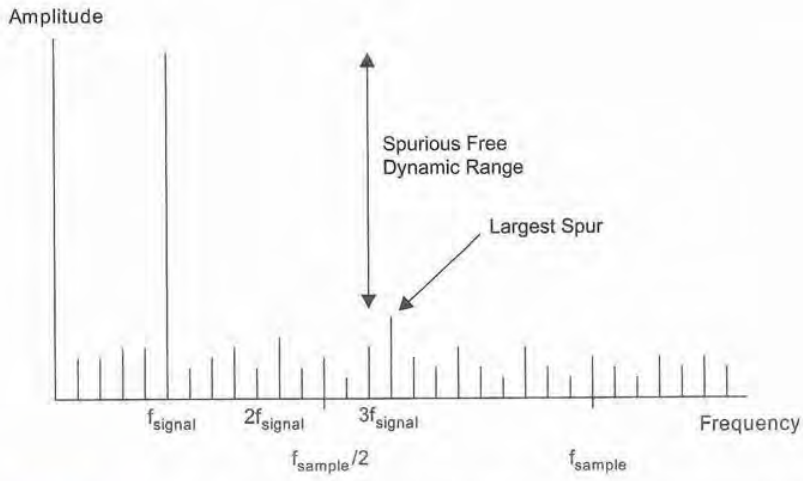


FIG 12.67 Typical DAC frequency plot showing harmonics and noise

Example

An 8-bit converter has a maximum SNR of 49.92 dB. If an implementation has an ENOB value of 7.1 bits, what is the SNR (in dB)?

Solution: SNR = 7.1 • 6.02 + 1.76 = 44.5 dB

Example

If the required SFDR is 47 dB, how many ENOBs are required?

Solution: ENOB = (47 – 1.76)/6.02 = 7.51 bits

As mentioned previously, DACs and ADCs are sampled data systems. Two major artifacts are present in DACs and ADCs. The first effect is due to sampling theory. This results in the replication of copies of the baseband signals at multiples of the clock frequency, as is shown in Figure 12.68(a). This requires that a low-pass filter follow a DAC to eliminate the unwanted signals. This is commonly called a *reconstruction filter*. In the figure, the Nyquist frequency ($f_{sample}/2$) is noted. This is the maximum signal frequency that can be generated by a DAC clocked at f_{sample}.

The second artifact results from the frequency response of a pulse, which is a $\sin(x)/x$ function as shown in Figure 12.68(b). This results in a roll-off with frequency of the

desired (and undesired) signals. This is also illustrated in Figure 12.68(a). This may have to be compensated for by the digital circuits driving the DAC if a flat response is required. Similarly, an ADC responds to signals at multiples of the sampling frequency. Thus, an *antialiasing filter* has to be provided so that the ADC does not see these signals. In addition, compensation may be needed for the $\sin(x)/x$ response.

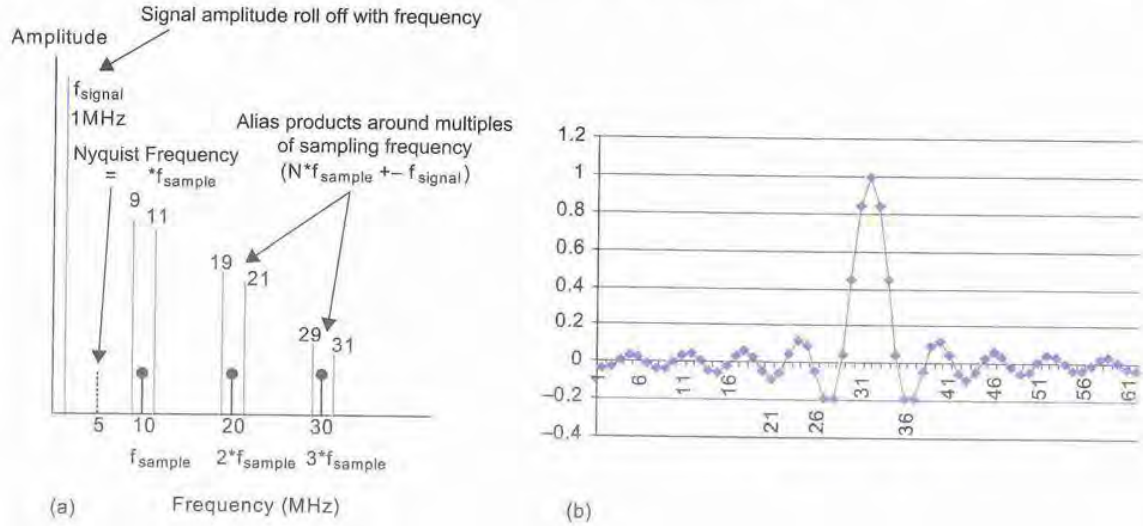


FIG 12.68 Frequency response of a DAC

Example

For a sample frequency of 10 Mhz, what are the alias products for a 1 MHz signal?

Solution: Products appear at the signal frequency offset from multiples of the sample frequency. So the first is at $f_{sample} - f_{signal}$ = 9 MHz and the next at $f_{sample} + f_{signal}$ = 11 MHz. The next two are offset from the second harmonic, i.e., 19 MHz and 21 Mhz, and so on. These are illustrated in Figure 12.68(a).

12.6.8 Digital-to-analog Converters

This section will present a selection of DACs roughly in order of increasing implementation complexity. All DACs presented here are within the capability of a careful CMOS circuit designer. DACs are simpler than ADCs.

A DAC can be implemented in software if a processor or DSP is available on the chip already, as shown in Figure 12.69. The processor provides a stream of 1's and 0's, which, when integrated (low-pass filtered), provides the required analog output. The RC time

constant of the low-pass filter is designed to cut out the harmonics present in the output at multiples of the sampling frequency. Typically, a simple DAC such as this is highly *oversampled*; i.e., the clock frequency of the serial bit stream is many times the eventual analog output frequency. For the simple first-order filter shown, this oversampling ratio might be in the range of 256–1024.

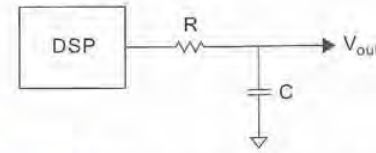


FIG 12.69 A DSP or processor-controlled DAC

If no processor is available, a converter called the *Pulse Width Modulated* (PWM) DAC, shown in Figure 12.70, can be implemented. This DAC employs a digital counter and a comparator. The counter cycles through a set of 2^N values and the comparator is set high when the count is less than the digital input B. This results in a waveform with a varying duty cycle. If B is small, the waveform will spend the majority of the time low. Conversely, if B is large, the output is high for most of the counter cycle. The linearity of this and the previous converter can be quite high, as it is limited by the linearity of the on-chip R and C in the filter, which can be reasonably good.

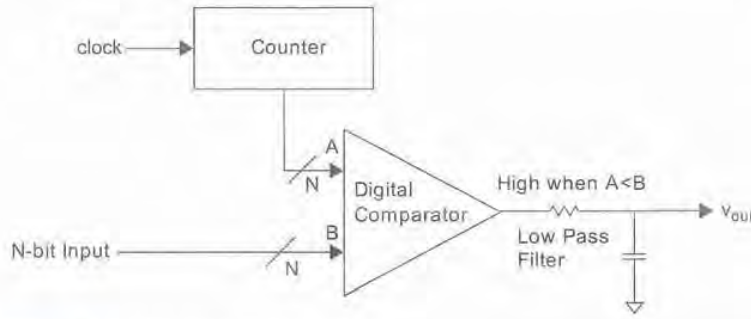


FIG 12.70 A pulse width modulated DAC

The *resistor string DAC* is perhaps the most straightforward Nyquist-rate DAC to design. A Nyquist-rate DAC is one in which the analog output values change at the DAC update rate and the usable analog frequencies extend to half the Nyquist (clock) rate. Two versions of this DAC are shown in Figure 12.71. A reference *voltage ladder*, consisting of 2^N resistors for an N-bit DAC, is connected from the supply to ground. A 2-bit converter using four resistors is shown in Figure 12.71(a). CMOS switches are used to switch the appropriate reference voltage to the output. A decoder switch can be used as shown, in which case the switch depth is N. Alternatively, each switch can be individually decoded as shown in Figure 12.71(b), in which case, the switch depth is one. While simple, this DAC is slower than other designs due to the RC time constant through the ladder and switches. In addition, the load resistance has to be high compared to the resistor string. Typically, it is useful as a reference DAC driving a CMOS buffer, op-amp, or comparator. It is suitable for moderate speed applications. Resistors can be constructed using polysilicon. The DAC is inherently monotonic due to the resistor string.

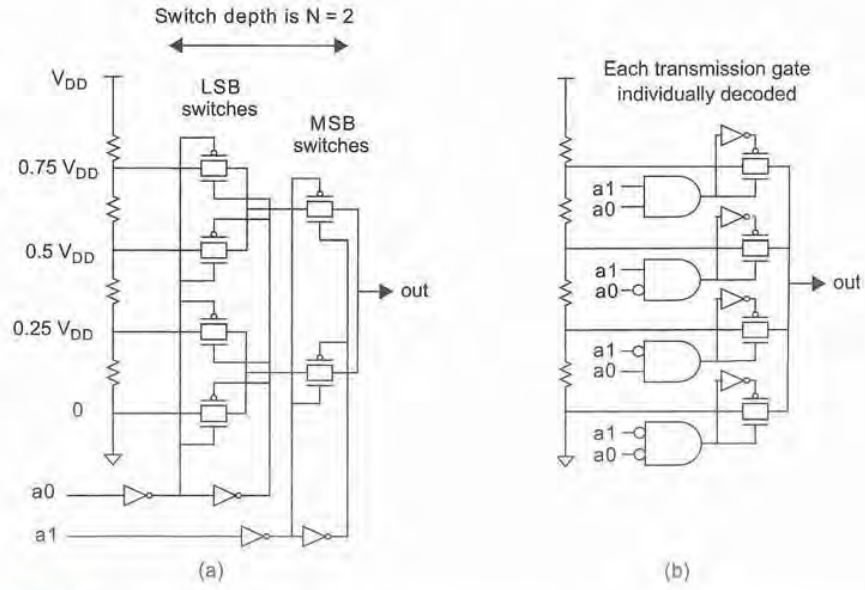


FIG 12.71 Resistor string DACs

The *R-2R DAC* uses an array of resistors of value R and $2R$, as the name suggests. A 4-bit version of this converter is shown in Figure 12.72. The DAC employs resistors, CMOS switches, and a buffer amplifier. The number of resistors required is $2N + 1$. The buffer has to be designed carefully to allow the input to be linearly amplified.

The fastest DACs in CMOS are those built from current sources. The basic principle of a 1-bit current DAC is shown in Figure 12.73(a). A digitally controlled current is switched into a resistor to create a voltage swing proportional to the product of the resistance and current. A simple current source ($P1$) is shown in Figure 12.73(b) along with a differential switch ($P2$, $P3$). The switch toggles the current from one leg to the other of a differential resistor network ($R1$, $R2$) under the control of complementary signals S and SN. The current source $P1$ is set with V_{bias}. The differential voltage ($V_{out} - V_{outn}$) is the DAC output. Figure 12.73(c) shows an improved cascode current source ($P1$, $P2$) with better linearity. Although pMOS devices have been shown to allow a GND-referenced system, nMOS devices can also be used for a V_{DD}-referenced system.

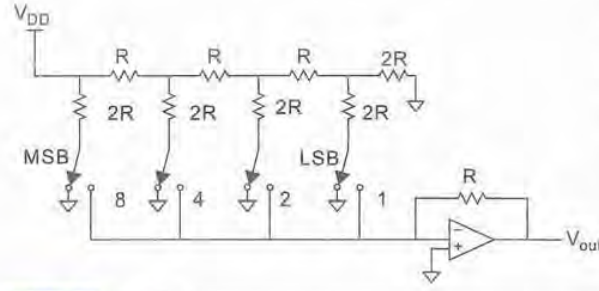


FIG 12.72 R-2R DAC

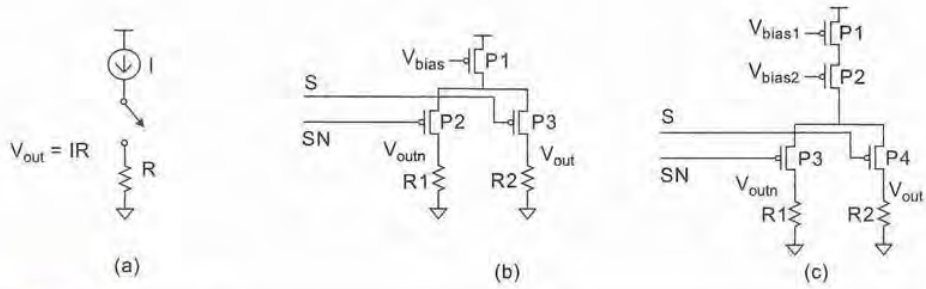


FIG 12.73 A current DAC

There are two basic methods of building a *current DAC*. The first is shown in Figure 12.74(a). Here, *N* weighted current sources are used to build an *N*-bit DAC. Each current source is built by sizing the current source in line with the bit weighting. For a 4-bit converter, a 1X, 2X, 4X, and 8X current source would be required. These could be built from

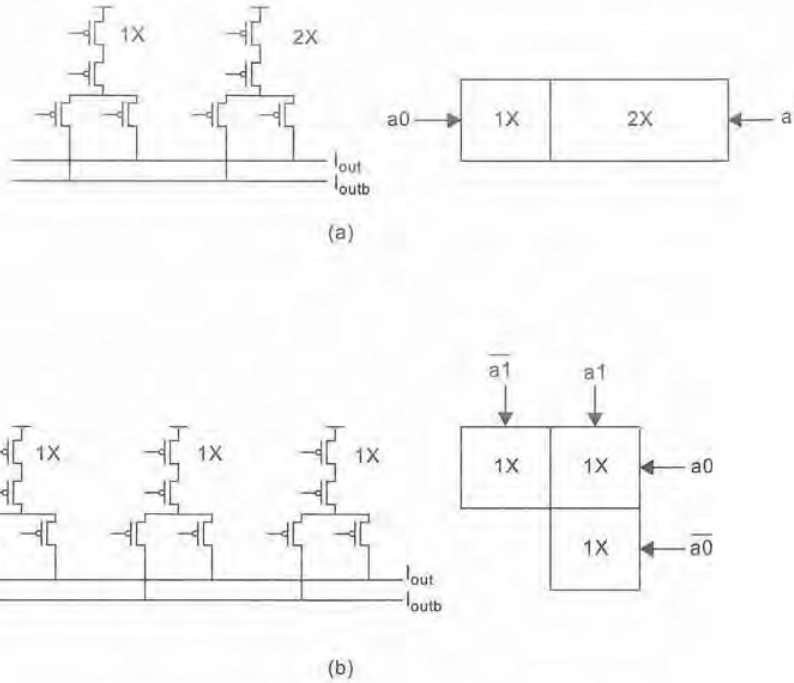


FIG 12.74 Current DAC architectures

1X, 2X, 4X, and 8X the basic current source using current mirroring and appropriately sizing the transistors. This style of current DAC is suitable for small DACs or DACs where speed is paramount. Problems can surface when designers try to match the scaled current sources to each other.

Figure 12.74(b) shows an alternative architecture which uses $2^N - 1$ identical current sources. This is called a *fully segmented DAC*. The linearity can be quite good if the matching of the current sources can be maintained. A 2-bit converter is shown in schematic form and a possible floorplan is shown.

Figure 12.75 shows a full implementation of a 4-bit current DAC. To the basic cascode current source and switch, a decode OAI gate is added in each current cell. An array of fifteen current cells is used. To the left of the current array, a row decoder, and at the bottom a column decoder, have been added. These decode the two LSBs and two MSBs respectively, and drive row and column lines into the array. Depending on which lines are activated, a number of current sources are turned on in proportion to the presented digital value. Each current source is biased by the common V_{bias1} and V_{bias2} voltages. These can be generated using a *replica bias* generator (using a replica of the unit current source) as shown (at the top left) and these in turn can be buffered by operational amplifiers. Some design hints have been included in the figure. Examples of this style of DAC can be found in [Bugeja00, Tiilikainen01].

12.6.9 Analog-to-digital Converters

Like DACs, Analog-to-digital converters (ADCs) are rated by precision, speed of conversion, power dissipation, chip size, and ease of design. ADCs also feature Nyquist converters and oversampled converters. Achieving state of the art ADC performance is an exacting science and the descriptions here about designing an ADC are not presented in great detail. However, moderate ADC design is quite within the capability of the CMOS designer who has an interest in analog circuits. A good strategy is to gain experience by "having a go" in the academic environment by designing and following an ADC design through fabrication and measurement. With the accuracy of simulation and layout extraction tools available for CMOS processes, many ADC (and DAC) architectures can be simulated in extreme detail, with the simulation results being close to fabricated chip results. Thus, experimentation can be completed "virtually."

An ADC is primarily categorized by the speed of conversion and number of bits. In common with DACs, the effective number of bits (ENOBs) tell the real story of a particular ADC. In common with DACs, other measures of interest are the offset, INL, DNL, SNR, and SFDR.

Figure 12.76 shows a *dual slope ADC*, which uses CMOS switches, an integrator and comparator, and digital logic to implement a relatively high-precision converter. In operation, the integrator is first reset by closing SWc. Next, the (negative) input voltage is integrated for a known time by closing switch SWa and opening SWc. SWa is opened and SWb closed with a (positive) reference voltage connected. The time at which the output crosses zero is measured and the input voltage can be calculated from:

$$V_{in}/|V_{ref}| = t_{measure}/t_{integrate}$$

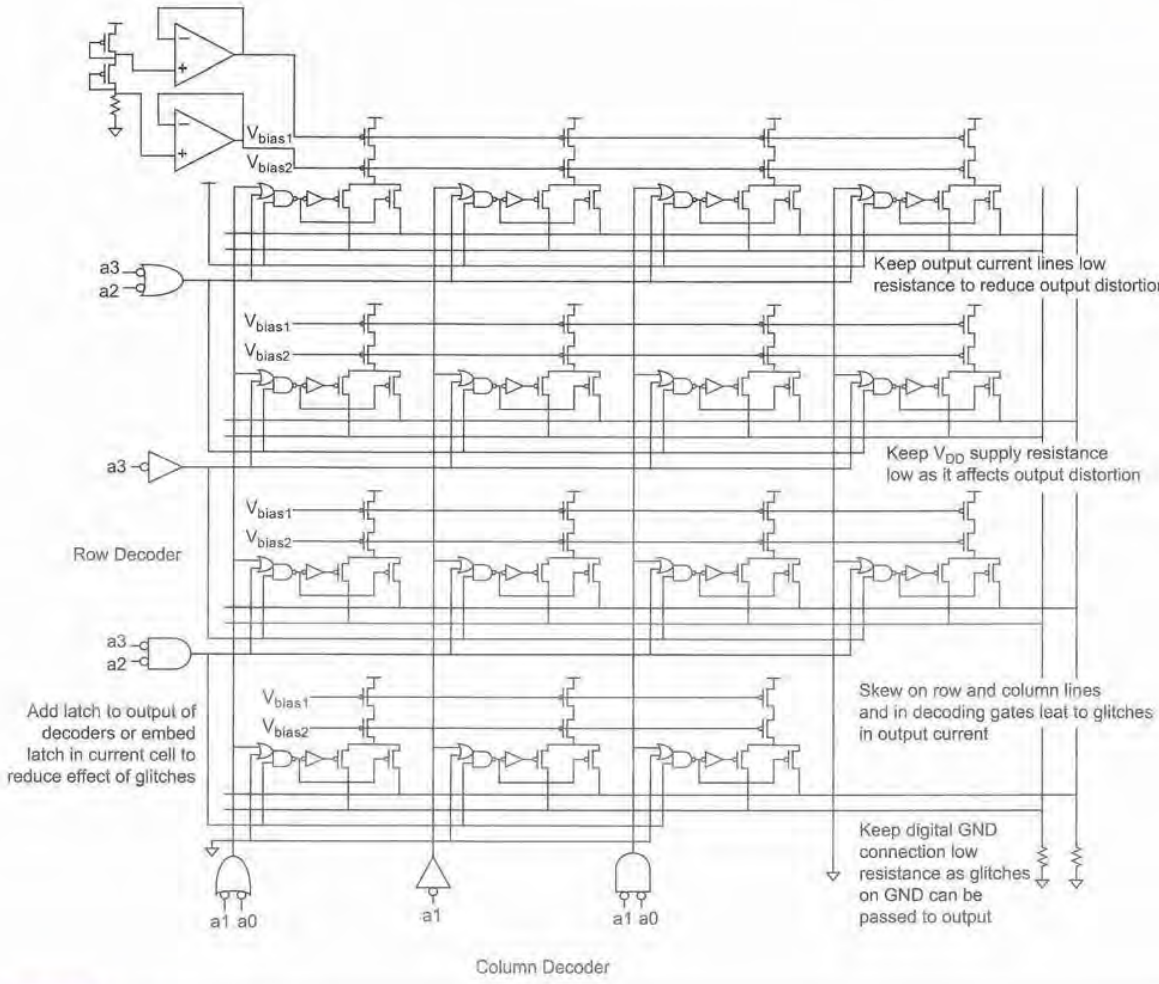


FIG 12.75 A 4-bit current DAC

This converter is relatively slow due to the integration times, but can achieve good precision with simple analog components. CMOS inverters can be used for the amplifiers, as shown later in this section. An example of a dual slope ADC can be found in [Rodgers89].

Figure 12.77 shows an ADC that compares the input to a value set by a DAC. The easiest algorithm for conversion is to ramp the DAC from zero to full scale and observe when the comparator switches. This would take 2^N clock cycles for an N-bit converter. A quicker method is to use a successive approximation algorithm. The following pseudo-

code demonstrates the successive approximation algorithm. This algorithm completes in $N + 1$ cycles and is preferred for this reason (i.e., 9 cycles versus 256 for an 8-bit ADC). The normal input range is zero to $V_{fullscale} \cdot (2^{N-1})/(2^N)$.

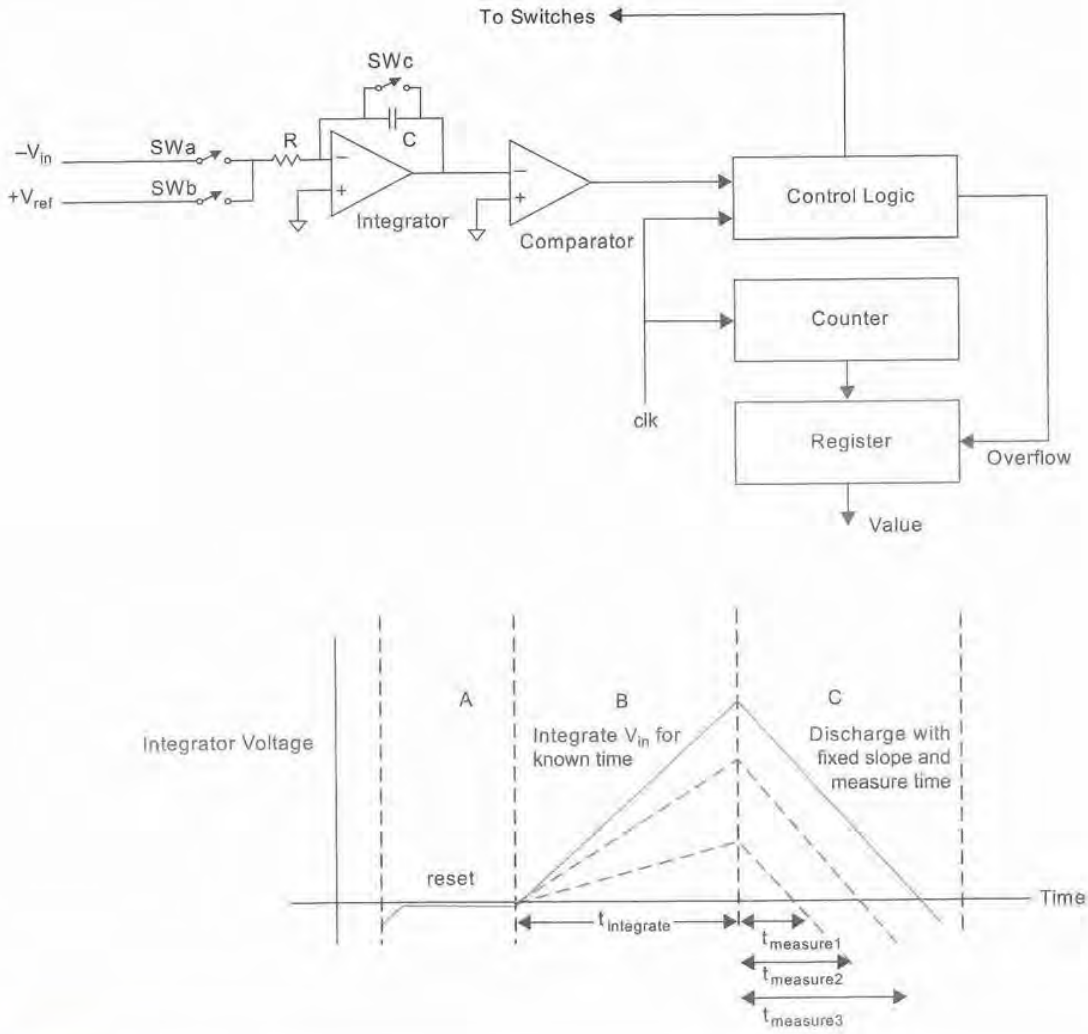


FIG 12.76 Dual slope ADC

```
Initialize:  Vdac = Vfullscale/2
             Vstep = Vfullscale/4
             Vout = 0

   loop repeat N
   write(Vdac) ;;output DAC value
   flag = read(comparator) ;; flag = Vin>Vdac
   if (flag)
      {
         Vdac = Vdac + Vstep ;; DAC value is high so reduce
         Vout = Vout + 1 ;;note contribution to output
      }
      else
      {
         Vdac = Vdac – Vstep ;; DAC is low so increase
      }
      Vout = Vout << 1 ;; multiply output by two
      Vstep = Vstep >> 1 ;; divide step by two
   repeat
```

Typical operation of the *successive approximation ADC* is shown in Figure 12.78 for a 3-bit converter with a 1 V full scale and 0.28 V applied to the input. In time slot 0, V_{dac} is set to half scale and the comparator sampled. As V_{dac} is greater than V_{in}, V_{step} of 0.25 V is subtracted from V_{dac}. In cycle 2, the process is repeated and 0.125 V is added to V_{dac}. Finally, 0.0625 V is subtracted for a final value 0.3125 V. The next cycle successively approximates 0.6875 V for a 0.72 V input. V_{out} lags the input by a conversion cycle of four clock cycles.

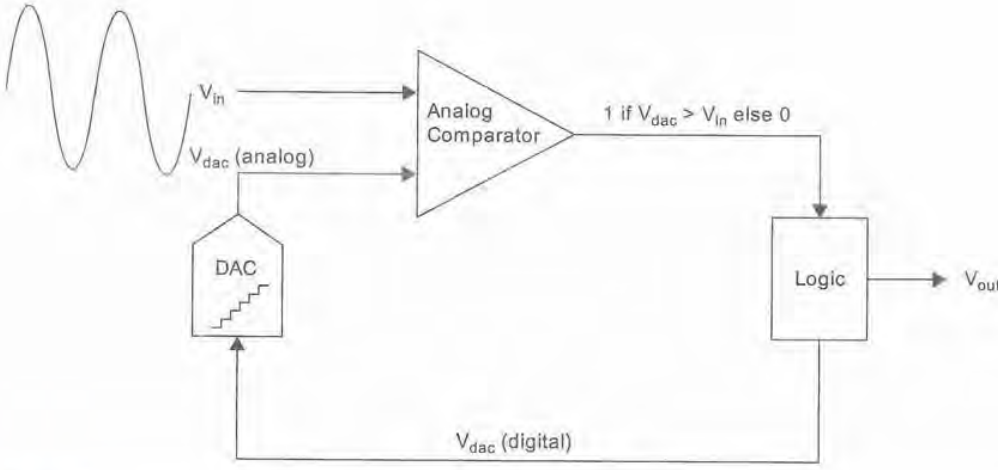


FIG 12.77 Successive approximation ADC

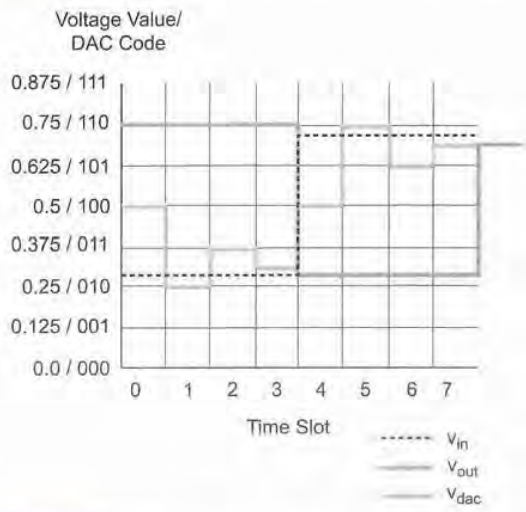


FIG 12.78 Successive approximation ADC operation

Representative successive approximation ADCs can be found in [Sauerbrey03, Promizter01, Mortezapour00]. One of the original commercial implementations can be found in [Timko80].

The *flash* converter is shown in Figure 12.79. This is the fastest converter architecture. The design consists of 2^N parallel comparators, each of which is fed with the input and a monotonically increasing reference voltage. This is most easily generated by using a string of resistors that is grounded at one end and fed at the other end with a reference voltage. When the analog input rises above the reference value fed to a particular comparator, the comparator turns on. Thus, a thermometer-like sequence of 1's will appear at the output of the comparators, rising and falling in sympathy with the analog input. The whole system is clocked at some frequency determined by the fastest time that the signal can be sampled and the comparator switched. The *thermometer code* is then fed to digital logic to extract a binary number corresponding to the analog input.

While much design effort can go into the comparator and associated sampling circuitry, a simple but still useful cir-

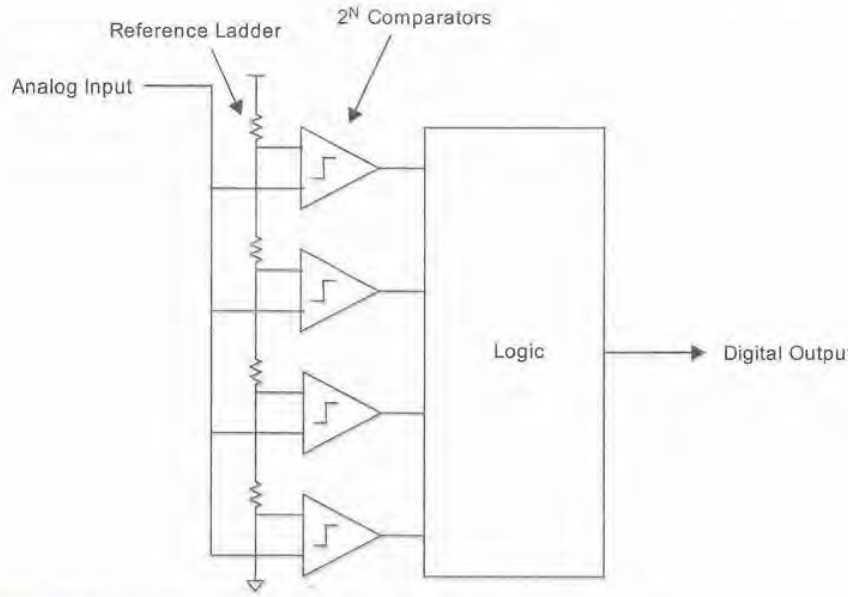


FIG 12.79 Flash ADC architecture

cuit uses one or two inverters [Dingwall79, Dingwall85]. The basic circuit is shown in Figure 12.80(a). The circuit consists of three CMOS switches, a capacitor, and a CMOS inverter, connected as shown. The operation is explained in Figure 12.80(b). In the reset cycle, the inverter is biased by connecting its input to its output. This settles the output at the inverter threshold voltage, as defined in Section 2.5.1. One end of the sampling capacitor is connected to the input of the inverter. In addition, the analog input is connected to the other end of the capacitor. In this configuration, the capacitor assumes a charge $Q = C \cdot (V_a - V_{cm})$, where V_{cm} is the inverter threshold voltage and V_a is the analog input voltage. In the second phase (Figure 12.80(c)), the reference voltage V_{ref} is connected to the capacitor and the charge is maintained. As the inverter is in high gain mode, the difference in analog input and reference voltage is amplified by the inverter. If the gain is high enough, the inverter drives the output voltage to the rails.

If a single stage does not have enough gain, another capacitively coupled gain stage can be added, as shown in Figure 12.81 [Dingwall85]. This figure shows the complete circuit as well as a latch to store the comparator output.

The design decisions are fairly simple with this design. The capacitor size has to be selected. The size of the capacitor in relation to the input size of the inverter determines the overall gain of a single stage. A value of roughly 10x the input inverter gate capacitance is a start. The inverter also has to be sized. Normally, there is no advantage to making the inverter much larger than minimum-sized because its size determines the sampling capacitor and the overall input capacitance of the converter. However, with sub-micron transistors, it is advantageous to lengthen the gate from minimum to flatten the I-V characteristic and reduce g_{ds}. The nMOS and pMOS transistors should be sized to place V_{cm} in the center of V_{ref}, which is commonly V_{DD}. The disadvantages of such a simple circuit compared to differential circuits are that the comparator is sensitive to common mode noise on the input, ground, and V_{DD}. Using V_{DD} as a reference also implies that this supply should be filtered. However, as supply voltages are reduced, the opportunity to build more sophisticated circuits is lessened and careful layout and design can yield good results for this comparator. This is especially true with an insulated SOI substrate or triple-well processes.

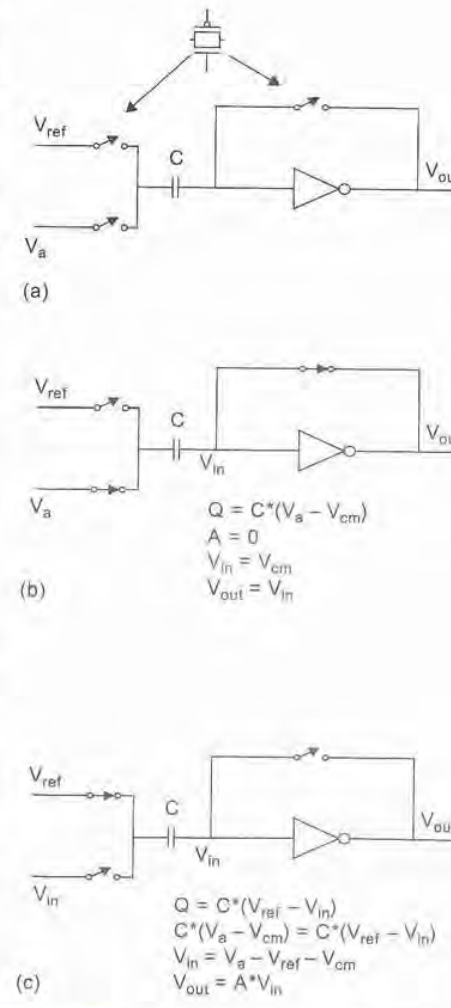


FIG 12.80 Simple CMOS inverter-based comparator

The output of the comparators is a thermometer code and the next stage has to decode this to a binary value. A "bubble gate," as shown in Figure 12.82, is used to determine the highest one in the thermometer code. While a simple 2-input gate can be used, the gate shown detects a sequence of 110 in the output. This prevents a spurious one (and zero bubble) from falsely triggering the decoder.

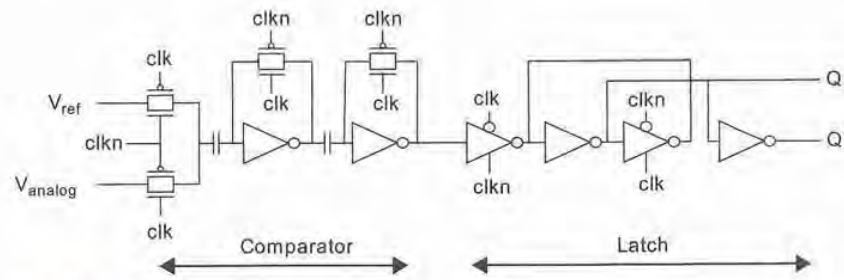


FIG 12.81 Improved gain comparator circuit with output latch

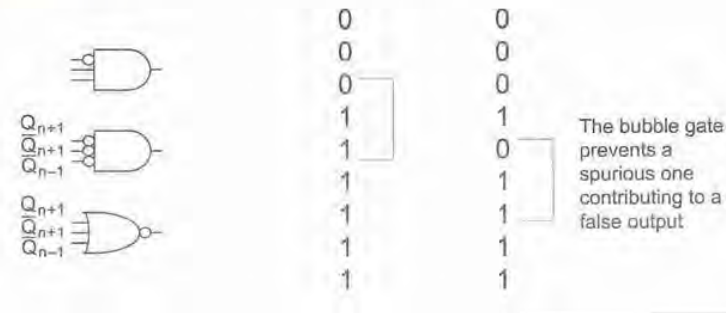


FIG 12.82 Thermometer bubble gate

The output of the bubble gate is then fed to a 1-hot-to-binary encoder. The first example shown in Figure 12.83(a) uses pseudo-nMOS NOR gates to convert to a binary number. The second encoder, shown in Figure 12.83(b), uses multiplexers. Another method is to count the number of 1's in the thermometer code with a cascade of adders.

This completes the design of the flash converter. Although it is the fastest converter, the large number of comparators places a significant load on any circuit driving the ADC. This usually limits the flash converter to fewer than 8 bits. Representative flash converters can be found in [Jiang03, Uyttenhove03, Donovan02, Scholtens02]. Some reduction in the number of comparators can be achieved using interpolation techniques and a technique called *folding*. Representative converters can be found in [Nauta95, Li03].

The *pipeline ADC* essentially trades the high speed and low latency of the flash converter for longer latency and a slightly more complicated (and hence lower speed) design. However, the power dissipation can be much lower than for a flash converter of the same speed. The design is outlined in Figure 12.84(a). The ADC is composed of (ideally) N identical stages for an N-bit converter. Each stage contributes a bit to the overall result. A single stage is shown is Figure 12.84(b). The input V_{in} is presented to an ADC that subtracts or adds a reference voltage (V_{ref}) from the input. The difference is then amplified by

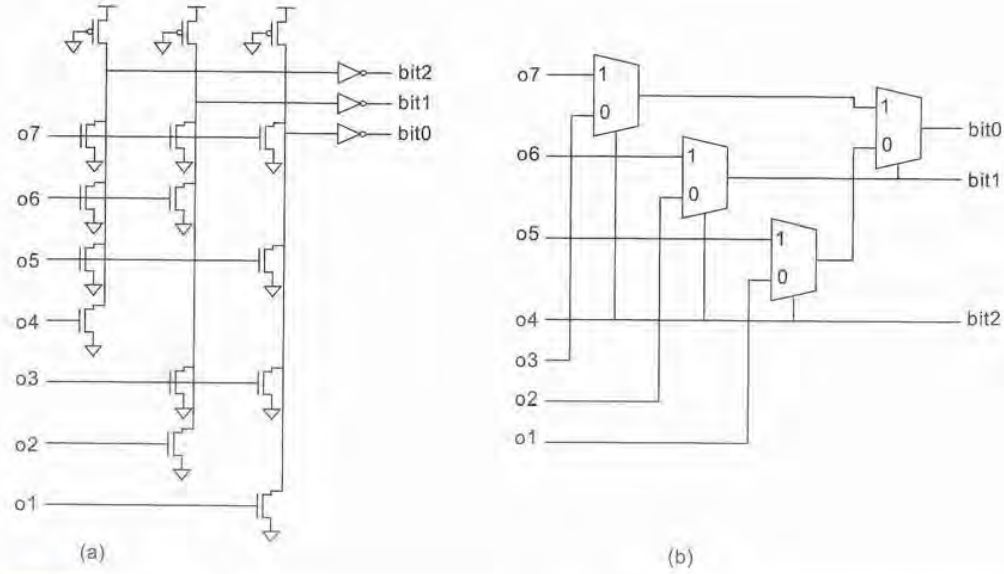


FIG 12.83 Flash ADC encoder

a factor of two and the process repeated. (This amounts to a distributed successive approximation converter.)

While all of these operations may seem difficult to achieve, a typical stage is shown in Figure 12.85 using CMOS switches, capacitors, and a gain stage.

The gain stage in the pipeline converter has to have enough gain to accurately subtract and perform the multiplication (assuming matching capacitors). The *folded cascode* stage shown in Figure 12.86 is a popular gain stage because it combines good gain into a single stage. This illustrates that the circuit complexity can be moderate. However, the design of the amplifier is beyond the scope of this text and the reader is referred to the literature for further details [Gray01].

Representative pipeline converters and descriptions of their operation can be found in [Lin91, Cho95, Jamal02, Chang03, Poulton03].

One of the most ubiquitous ADC architectures in use today is the *sigma-delta* architecture shown in Figure 12.87. This converter, which was developed in the late 1970s, is ideal for processes where digital circuits are easier to implement than analog circuits [Candy76]. The converter works by subtracting a delayed digitized sample of the analog input from itself. This is passed through a filter, $F(s)$, and then sent to a comparator. In the first-order example shown, this is then applied to a 1-bit DAC, which completes the loop. The digital output of the comparator is passed through a digital filter and the digital code for the analog input retained. Sigma-delta converters are oversampled converters. That is, they operate at a multiple of the required signal frequency. Oversampling ratios of

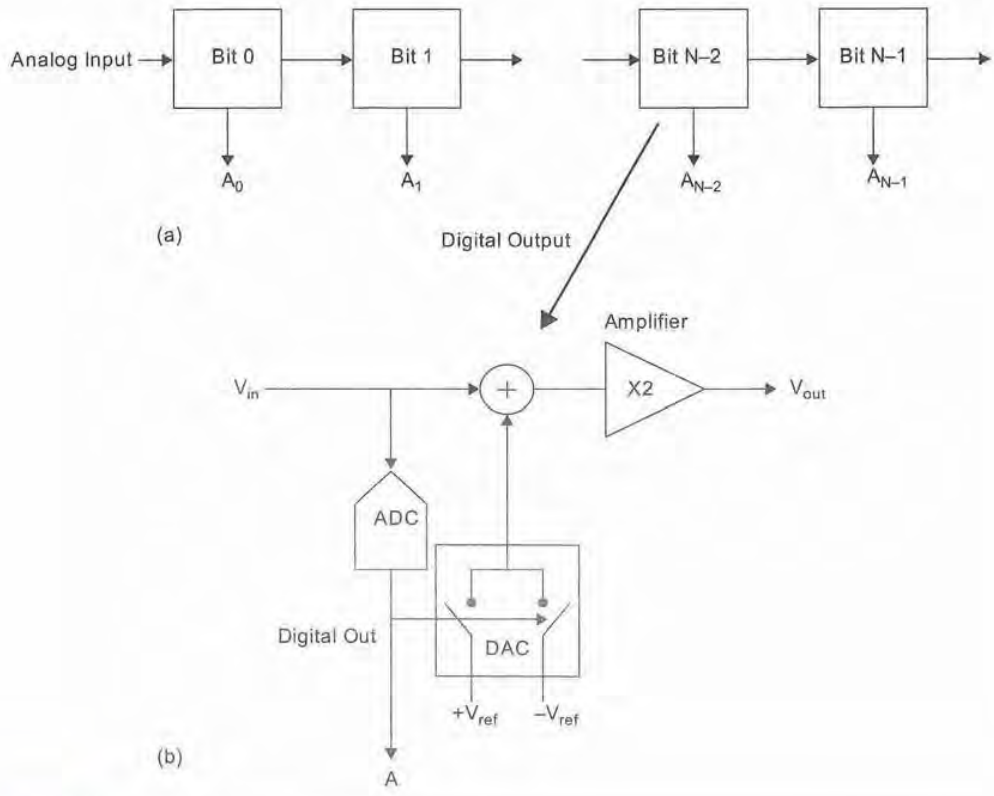


FIG 12.84 Pipeline ADC block diagram

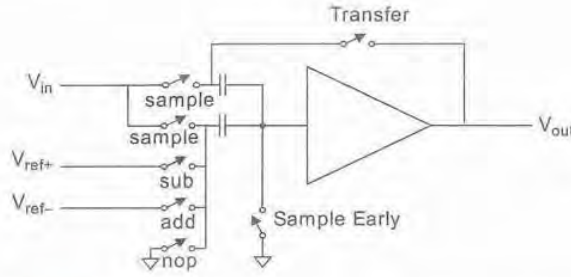


FIG 12.85 Pipeline ADC single stage

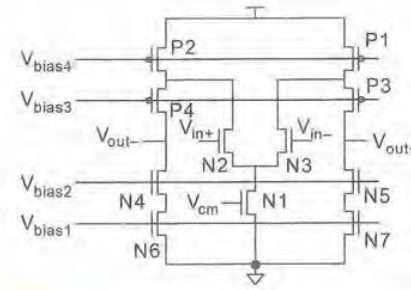


FIG 12.86 CMOS folded cascode op-amp

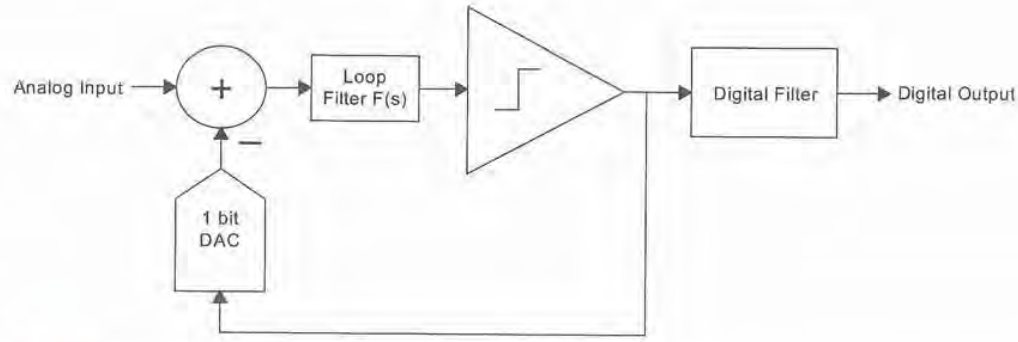


Fig 12.87 Sigma-delta ADC

between 40 to over 1000 are commonly used. As the oversampling ratio increases, so does the ADC precision (of course, at the expense of speed). Sigma-delta ADCs are commonly used in CD players at audio frequencies with resolutions up to 20 bits, but they have also been used at radio frequencies (10 MHz) for moderate precision (8–10 bits). They are popular CMOS converters because the analog components are limited to a few blocks, which can be designed carefully to obtain high performance. Representative examples can be found in [Kappes03, Sauerbrey02, Gupta03].

Apart from the basic ADCs presented, many more architectures have been invented. A single flash converter can be modified to be used twice at half the resolution and half the speed. This is the basis for a two-step flash converter. Finally, any converter can be used in parallel with delayed clocks to form an arbitrarily high-speed converter by interleaving the parallel converters. A CMOS converter implemented from 80 8-bit 250 MHz current-mode pipeline converters demonstrated operation at 20 GHz [Poulton03].

12.6.10 Radio Frequency (RF) Circuits

RF circuits are generally low in device count but high in design effort. While we will illustrate some RF circuits in this section, it must be understood that device sizing and component selection should be done in conjunction with a specialist RF design text [Lee98, Razavi98, Leung02]. To repeat, the point of showing these circuits is to encourage digital designers to explore these allied CMOS design areas.

As an introduction, a typical radio transceiver is shown in Figure 12.88. The receiver is what is called a *direct conversion* or *homodyne* architecture. It takes a 5 GHz carrier and downconverts this to a baseband signal by multiplying with a 5 GHz on-chip oscillator. The resulting signal is filtered by a 10 MHz low-pass filter and then amplified by a variable-gain amplifier. This signal is then fed to a 40 MHz ADC. While this is a simplified receiver, it includes most of the key modules required for RF applications at any frequency, namely, amplifiers (fixed-gain and variable-gain), oscillators, mixers, and filters.

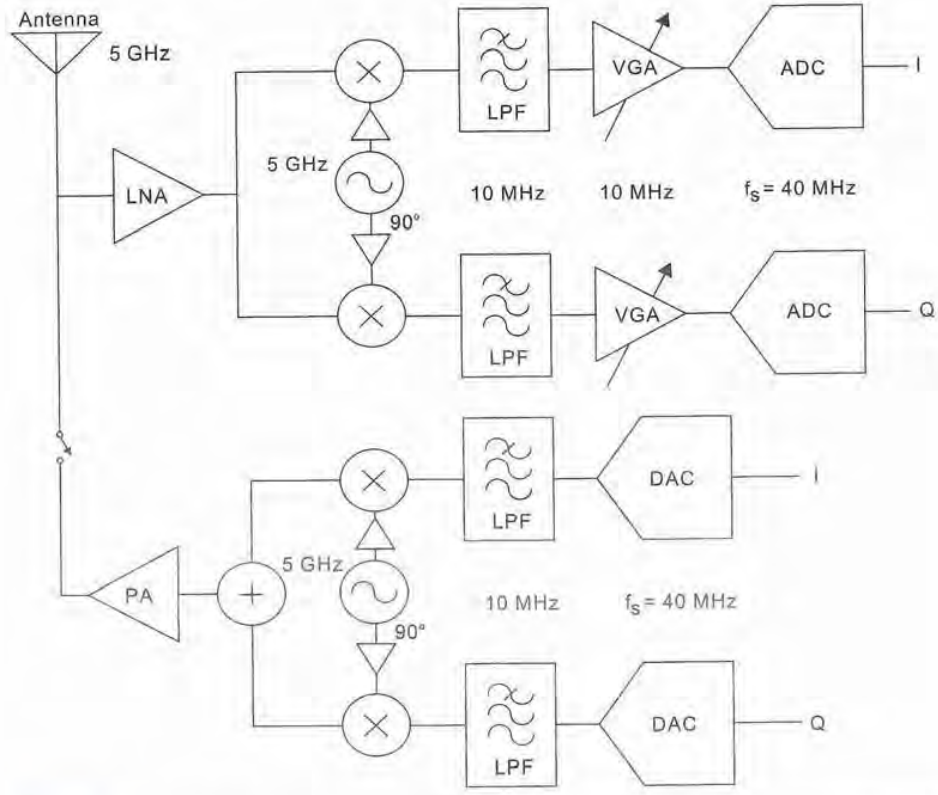


FIG 12.88 Typical CMOS radio transceiver

The transmitter path is roughly the reverse of the receiver. It starts with digital IQ values, which are fed to a DAC. The output is filtered by a low-pass reconstruction filter and then upconverted to 5 GHz. The output of the upmixer is amplified and fed to an antenna via an external transmit/receive switch. The transmitter is simpler than the receiver because of the larger signal amplitudes [vanZeijl02].

The rest of this section will introduce representative examples of each type of circuit introduced in this architecture.

A variety of CMOS RF amplifiers are shown in Figure 12.89 [Lee98]. Figure 12.89(a) shows a simple resistively loaded common source gain stage. It is biased by resistor $R1$ connected to V_{bias}. These circuits can give bandwidths in the GHz regions for submicron processes and should not be overlooked despite their simplicity. The stage is inherently wide-band. Figure 12.89(b) shows a tuned amplifier. Inductor $L1$ resonates with device and load capacitance to provide gain at a particular frequency. The stage is biased similarly to the resistively loaded example. Figure 12.89(c) shows the corresponding dif-

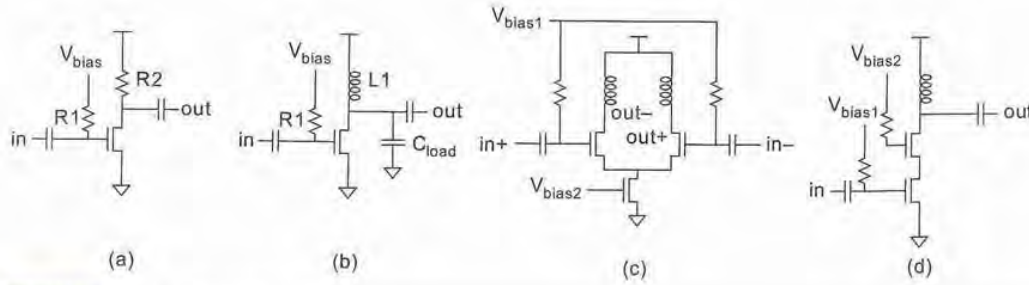


FIG 12.89 CMOS RF amplifiers

ferential stage, while Figure 12.89(d) shows a cascode gain stage. This is normally the starting point for a low-noise amplifier (LNA) that might be used at the front-end of a receiver.

Figure 12.90 shows the cascode RF amplifier with some added inductors. The source and gate inductances are used to tune out the input capacitance and present a resistive (50 Ω usually) load at the input. This stage can also be implemented differentially, which illustrates that while the initial circuit might be simple, extracting top performance from a simple gain stage requires sophisticated design.

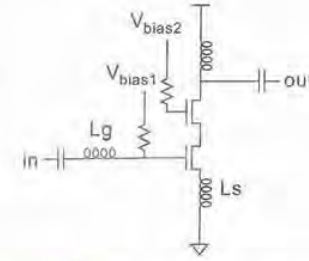


FIG 12.90 Cascode RF amplifier

Figure 12.91 shows an LC *voltage-controlled oscillator* (VCO) capable of operation well into the microwave regions (> 10 GHz) with modern processes in conjunction with the circuitry required to implement a phase locked loop. The inductors (*L1*, *L2*) resonate with the stray capacitance at the drain of *M3* and *M4* to oscillate at the required frequency. The cross-coupled nMOS transistors (*M3*,

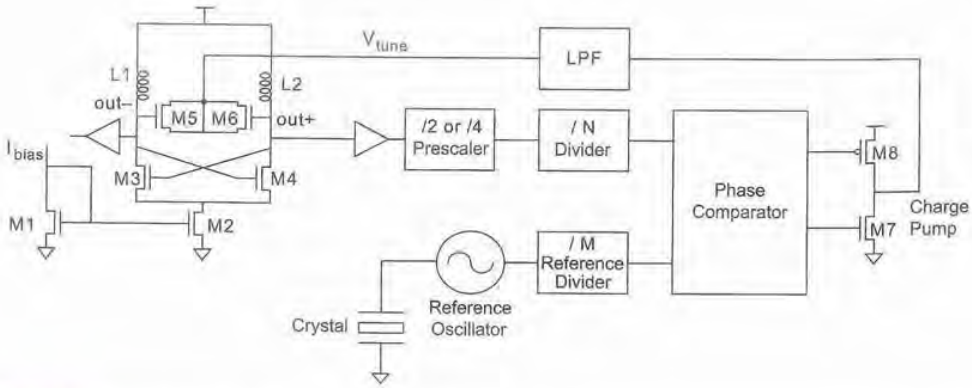


FIG 12.91 LC oscillator in a phased locked loop

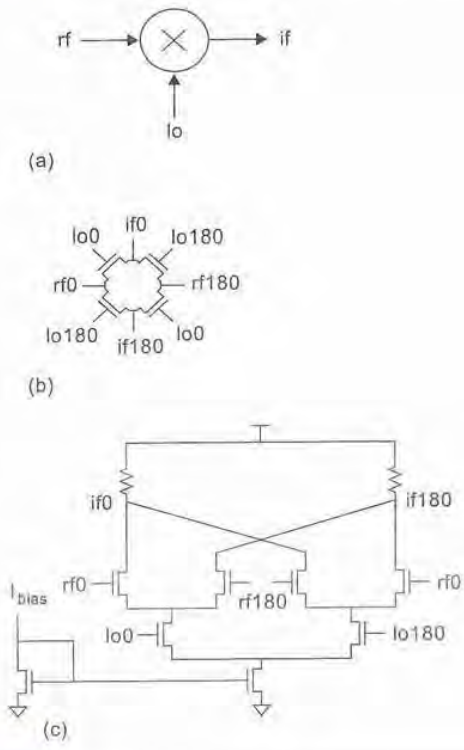


FIG 12.92 CMOS mixers

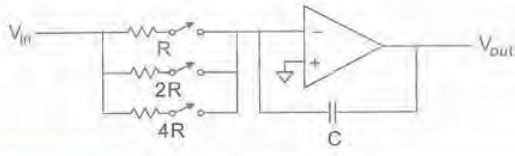


FIG 12.93 Low-pass filter

*M*4) provide the gain for the oscillator. Transistors *M*5 and *M*6 are used as *varactors* or voltage-variable capacitors to tune the oscillator frequency. *M*1/*M*2 forms a current mirror to bias the oscillator. The output of the oscillator is buffered and fed to a special *prescaler* that divides by two or four and presents a lower-frequency digital waveform to a conventional CMOS logic divider. A crystal oscillator is used as a reference by a phase comparator. The phase comparator feeds a charge pump, which, in conjunction with a low-pass filter, produces an increasing or decreasing analog voltage V_{tune} that is fed back to the *LC* oscillator. When configured properly with the correct ratios in the VCO divider and the reference divider, the feedback loop stabilizes the *LC* oscillator and prevents it from drifting with voltage or temperature.

Parameters of interest in an oscillator include the frequency of operation, power dissipation, and phase noise. The latter often determines the usefulness of a given oscillator in a particular system. Phase noise in turn is determined mainly by the gain of the transistors at the frequency of interest and the circuit *Q* of the inductors. A lot of effort goes into creating high *Q* inductors to achieve low-phase noise oscillators.

A *mixer* is an analog multiplier that converts one frequency to another. Figure 12.92(a) shows a symbol for a mixer and a typical application where a high radio frequency (RF) signal (say 5 GHz) is converted to a lower intermediate frequency (IF) signal (say 1 GHz) by mixing with a local oscillator (say 4 GHz). The mixer produces the sum and difference of the RF and LO (i.e., 1 GHz and 9 GHz). This is simply the result of multiplying two sine waves together. The unwanted product (9 GHz) is eliminated by filtering at the output of the mixer. The simplest CMOS mixer is the quad FET switch shown in Figure 12.92(b). Signals with their corresponding phases are shown. Correct bias has to be applied to each port. Figure 12.92(c) shows an active mixer based on a *Gilbert cell* [Zhou03, Melly01]. This has higher gain and lower noise than the ring mixer, but also has lower dynamic range.

Low-pass filters are circuits of use in an RF environment. Figure 12.93 shows a simple low-pass active filter that can be tuned by altering resistor combinations using CMOS switches. This form of continuous time filter can be increased in order, but the same type

of tuning can be employed. Alternatively, capacitors can be switched and fixed resistors used.

The CMOS switch idea can also be used to build variable-gain amplifiers as shown in Figure 12.94. This design employs binary weighted capacitors.

As mentioned previously, decreasing V_{DD} presents challenges for linear circuits, especially amplifiers. Figure 12.95 shows a high-speed op-amp with feed-forward compensation that only uses pMOS loaded differential stages, which work well at low supply voltages [Harrison03]. This is suitable for the filter and active gain control (AGC) applications mentioned previously. As with other circuits included in this section, the circuit is included here to illustrate the point that high-frequency amplifiers are not necessarily complex in circuit terms.

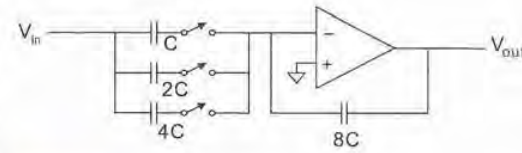


FIG 12.94 Gain controlled amplifier

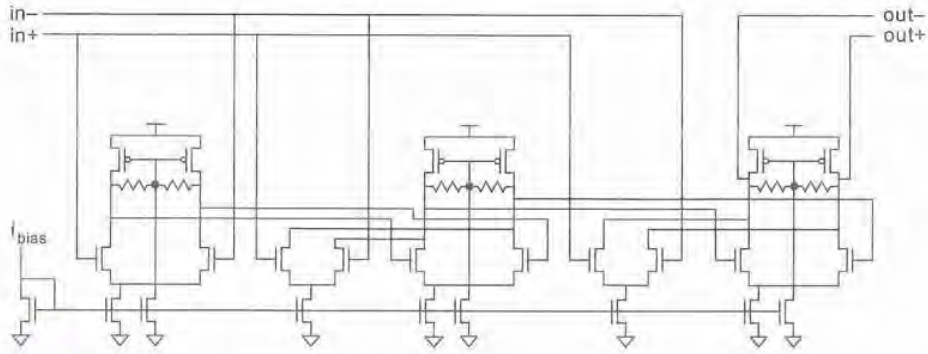


FIG 12.95 A high-speed CMOS op-amp

12.6.11 Analog Summary

This section was included to provide a brief introduction to the type of circuits that a designer might encounter in a mixed-signal analog or RF system-on-chip design. In such a short space, we can hardly do justice to this expansive area. However, we hope that the principles and circuits provided here will provide a springboard to understanding these circuits in more depth by consulting specialist texts and experimenting with circuits through simulation.

12.7 Pitfalls and Fallacies

Neglecting package parasitics

The resistance, capacitance, and inductance of the package have enormous impact on the power and I/O signal integrity of high-speed digital chips. They must be incorporated into modeling.

Using an inadequate power grid

A power grid should use generous amounts of the top two metal layers running in orthogonal directions. A mesh that mostly runs in only one direction is subject to excessive IR drops when many gates on a single wire switch simultaneously. It can also lead to serious inductive problems because of the huge current loops. The power grid should use many narrow wires interdigitated with the signals to provide a low S:R ratio rather than a few wide wires forming large current return loops. The grid should also avoid slots and other discontinuities that might lead to large current loops and high inductance.

Goofing your PLL/DLL

Phase-locked loops are notoriously difficult to design correctly. If poorly designed, they can oscillate at the wrong frequency, fail to acquire lock, or have excessive jitter. Careful circuit design is necessary to ensure they work across process variation and reject power supply noise. If the PLL does not work, testing the rest of the chip can be difficult or impossible. Most successful companies either have an in-house team that specializes in PLLs or they license their loops from a reputable third party.

Top Six Ways to Fool the Masses about Clock Skew

1) **Calculate clock skew without using process variation data**
   Random skew depends entirely on the mismatch of transistors (especially L_e) and wires on a chip. This mismatch varies with distance and layout technique. The process corners model worst-case variation from chip to chip, which can be far greater than between two nearby transistors; this results in unacceptably conservative skew budgets. But reliable data for on-chip variation can be hard to obtain, especially for small ASIC design teams and universities. Unless this data is used, clock skew budgeting is largely a matter of guesswork.

2) **Claim "zero skew"**
   Many papers state that a system has *zero skew* when the writers really mean that it has zero systematic skew. These systems may have significant random skew as well as drift and jitter. The term *zero skew* is deceptive and is best avoided.

3) **Report only systematic skew**
   Many papers also report only the systematic skew. In a well-balanced clock distribution network, systematic skew is often smaller than random skew and jitter.

4) **Ignore jitter**
   Jitter depends on time and space and is difficult to model or estimate. Unsophisticated clocking strategies sometimes ignore jitter. This results in unrealistic skew budgets. In particular, active deskew buffers increase clock distribution delay. Voltage noise on the buffers appears as jitter. Unless the supplies are unusually quiet, the buffers can increase jitter more than they decrease systematic or random skew.

5) **Report measured skew at only two elements over a brief period of time in a quiet environment**
   Measuring skew on a chip is also difficult. Some papers measure clock interarrival times at only two or a few points on the chip for a brief period of time and report those as the skew. As a chip has many clocked elements, you are unlikely to find the worst-case skew by measuring just a few points. Moreover, measurements over a brief time interval are unlikely to capture worst-case jitter. The chip

should be exercised through a variety of modes that cause large fluctuations in supply current to cause maximum power supply noise and clock jitter.

6) **Don't report the skew budget used during design** Designers often choose rather conservative clock skew budgets during design because they must ensure the design will operate correctly. Reporting a "measured" skew rather than a skew budget will give a smaller number.

12.8 Historical Perspective

Clock distribution is a persistent challenge in VLSI design. The clock often accounts for 30–50% of the chip's dynamic power. As clock periods have decreased, the clock distribution network requires more elaborate design to deliver clocks with skews below 10% of the cycle time. The DEC Alpha series of microprocessors led the clock frequency race through much of the 1990s. In this section, we will trace the evolution of their clock grids. Figure 12.96(a) shows the grids for the Alpha 21064, 21164, and 21264 microprocessors, while Figure 12.96(b) plots the systematic clocks skew across the die [Gronowski98].

In the 200 MHz Alpha 21064 [Dobberpuhl92], the clock grid drove TSPC latches directly without any local clock gaters. The final clock load is 3.5 nF, driven by a 35 cm (!) wide inverter arranged along the center of the chip. Such a wide inverter is built from many smaller inverters ganged in parallel. A binary tree layout is used to distribute the external clock to all of the inverters simultaneously. The clock skew is zero near the center and increases toward the edges. The clock driver generates so much heat that it raises the chip temperature by 30° C nearby. Checking for hold time violations is simply a matter of counting enough gates between latches.

In the 300–433 MHz Alpha 21164 [Bowhill95, Gronowski96], the clock grid directly drove conventional two-phase dynamic transmission gate latches. Two banks of clock buffers located midway between the center and the edges of the die drive the 3.75 nF clock load. The banked drivers also lead to a more even temperature distribution across the die.

The 600 MHz Alpha 21264 [Gieseke97, Gronowski98, Bailey98] used a clock grid that in turn drove an assortment of local clocks through gaters. The grid is divided into four "window panes," each of which is driven from its four edges. Each driver along the edge of a pane uses many inverters in parallel. The clock is distributed to the inverters in a fashion that equalizes the RC delay by tapering the wires appropriately. The systematic skew is close to zero along the edges of the panes and remains relatively small near the center because the wires are short. Some gaters perform clock gating to reduce power consumption. Others generate delayed clocks to fix critical paths. Because the system now has multiple logical clocks with different arrival times, checking for hold time violations requires more careful timing analysis.

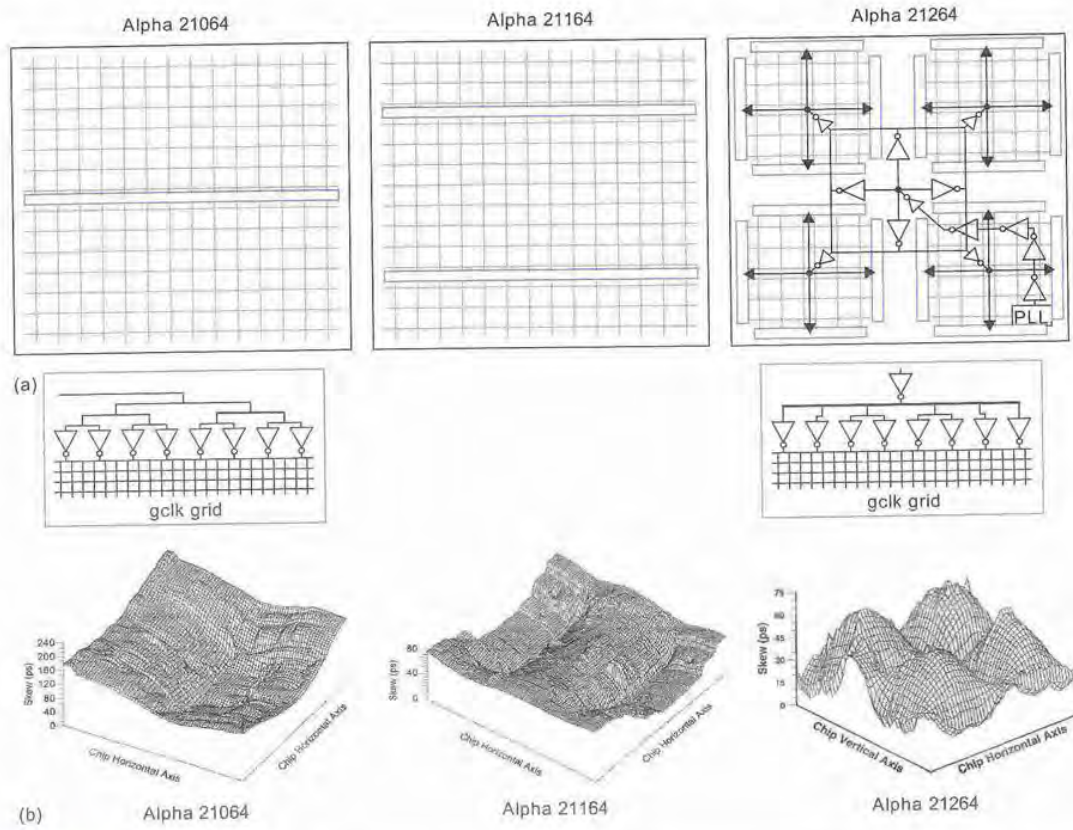


FIG 12.96 Alpha clock grids and systematic skew. (b) © 1998 IEEE.

In the Alpha 21064 and 21164, the global clock distribution network must have low resistance to directly drive such a large load capacitance. This in turn requires an enormous amount of metal wiring dedicated to the clock grid. The wire also has a high capacitance, so the clock grid consumes a large amount of power. For example, the 21164 clock distribution system consumes 20 W, or 40% of total chip power. In the 21264, the local clock gaters have electrical effort that reduces the capacitance seen directly on the global clock. This is beneficial because it reduces the metal usage and capacitance of the global clock grid while simultaneously reducing global clock skew. However, the local gaters introduce additional skew from random variation, drift, and jitter.

Summary

This chapter has surveyed package, power distribution, I/O, clock, and analog subsystem design. While each topic is a book in itself and a specialty design area, the short fat VLSI designer must understand enough about each area to optimize the system as a whole.

Packages connect the chip to the board or module, protect the chip, and are the first link in removing heat. They should offer plenty of connections, low thermal resistance, and low parasitics, while still being inexpensive to manufacture and test. Flip-chip packaging using solder bumps distributed across the die has become popular because of the large number of connections and low inductance.

The power distribution network consists of elements on the chip, package, and board. It must deliver a stable voltage across the chip under fluctuating current demands. Noise is caused by both average and peak current requirements. Multiple bypass capacitors offer low impedance to help filter high-frequency IR and L di/dt noise, but the DC supply resistance must be low enough to deliver the average current. V_{DD} and GND lines should be interdigitated in both directions with signal wires to provide small current return loops and low inductance. The supply wires must also have enough cross-sectional area to avoid electromigration problems. These requirements imply large amounts of metal and bypass capacitance, yet cost constraints dictate no more chip area than necessary.

I/O signals include inputs, outputs, bidirectional signals, and analog signals. The I/O pads must deliver adequate bandwidth to large off-chip capacitances at voltage levels compatible with other chips. They must also protect the core circuitry against overvoltage and electrostatic discharge.

A clocking subsystem includes clock generation, distribution, and gater elements. The clock generator can use a PLL to align the on-chip clock to an external reference for synchronous communication and to perform frequency multiplication. The clock distribution network should send the global clock to all clocked elements with low skew, yet not consume excessive power or area. The gaters perform local clock stopping or can produce multiple phases from the single global clock.

Basic analog building blocks include common source amplifiers, current mirrors, and differential amplifiers. From these, we can construct operational amplifiers, D/A converters, and A/D converters.

Exercises

12.1 A ceramic PGA package with a good heat sink and fan has a thermal resistance to the ambient of 10° C/W. The thermal resistance from the die to the package is 2° C/W. If the package is in a chassis that will never exceed 50° C and the maximum acceptable die temperature is 110° C, how much power can the chip dissipate?

12.2 Explain how an electrostatic discharge event could cause latchup on a CMOS chip.

12.3 Comment on the advantages and disadvantages of H-trees and clock grids. How does the hybrid tree/grid improve on a standard grid?

12.4 Calculate the bias-point and small-signal low-frequency gain of the common source amplifier from Figure 12.46 if $V_t = 0.7, \beta = 240\ \mu A/V^2$, and the nMOS output impedance is infinite. Let $V_{BIAS} = 3\ V$, $V_{DD} = 15\ V$, and $R_L = 10\ k\Omega$.

12.5 Prove EQ (12.24).

12.6 Calculate the output impedance of the Wilson current mirror in Figure 12.97.

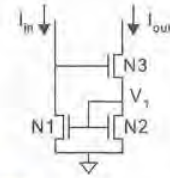


FIG 12.97 Wilson current mirror

12.7 Design a current source that sinks 200 μA, using the process parameters from the example in Section 12.6.4. What is the minimum drain voltage over which your current source operates?

12.8 Find the output impedance of your current source from Exercise 12.7. By what fraction does the current change as the output node changes by 1 V?

12.9 Simulate the operational amplifier of Figure 12.63. Using minimum-size transistors and a 10 kΩ resistor, what is the gain?

12.10 What changes would you make to the amplifier from Exercise 12.9 to increase the gain? What are the tradeoffs involved? What gain can you achieve using reasonable changes?

12.11 Prove EQ (12.31).

12.12 Use SPICE to find the transconductance and output resistance of a minimum-size transistor in your process biased at $V_{gs} = V_{ds} = V_{DD}/2$. What is the $g_m r_o$ product?

12.13 Repeat Exercise 12.12 for a transistor with 2x minimum channel length. How does the product change?

12.14 In Section 12.6.8 on resistor string DACs, it was mentioned that a similar DAC can be implemented with capacitors. Design the architecture of a 4-bit capacitor DAC.

12.15 A bias generator is required to generate 16 steps from 0 to 100 μA to bias an amplifier. Design a CMOS DAC to do this, assuming the presence of a 50 μA reference current.

12.16 Differential circuits provide good noise immunity and have the advantage of dual rail inputs and outputs. Pseudo-differential circuits based on CMOS inverter amplifiers can be implemented by using two signal paths that process each signal, but do not provide for the same level of noise immunity. Design a single stage of a pipeline ADC that uses this style of differential circuit.

12.17 To reduce clock and decoder skew in a current-mode DAC, a latch is often included in the current cell. Design the circuit for such a cell, demonstrating where the latch would be placed. If this is a slave latch, where would the master latch be located?

Appendix

Verilog

A

A.1 Introduction

This appendix gives a quick introduction to the Verilog Hardware Description Language (HDL). There are many texts on Verilog ([Smith00, Thomas02, Ciletti99] and others) that provide a more in-depth treatment. The IEEE standard itself is quite readable as well as authoritative [IEEE1364-01]. Many books treat Verilog as a programming language, which is not the best way of viewing it. Verilog is better understood as a shorthand for describing digital hardware. It is best to begin your design process by planning, on paper or in your mind, the hardware you want. (For example, the MIPS processor consists of an FSM controller and a datapath built from registers, adders, multiplexers, etc.) Then, write Verilog that implies that hardware to a synthesis tool. A common error among beginners is to write a program without thinking about the hardware that is implied. If you don't know what hardware you are implying, you are almost certain to get something that you didn't want. Sometimes this means extra latches appearing in your circuit in places you didn't expect. Other times, it means that the circuit is much slower than required or it takes far more gates than it would have if it were more carefully described.

The Verilog language was developed by Gateway Design Automation as a proprietary language for logic simulation in 1984. Gateway was acquired by Cadence in 1989 and Verilog was made an open standard in 1990 under the control of Open Verilog International. The language, with some revisions, became an IEEE standard in 1995 and was updated in 2001. This appendix is consistent with the 2001 standard.

As mentioned in Section 1.8.4, there are two general styles of description: *behavioral* and *structural*. Structural Verilog describes how a module is composed of simpler modules or basic primitives such as gates or transistors. Behavioral Verilog describes how the outputs are computed as functions of the inputs. There are two general types of statements used in behavioral Verilog. *Continuous assignment* statements necessarily imply combinational logic[1] because the output on the left side is a function of the inputs on the right side. *Always* blocks can imply combinational logic or sequential logic, depending on how they are used. It is good practice to partition your design into combinational and sequential components and then write Verilog in such a way that you get what you want. If you

[1]Recall that the outputs of *combinational logic* depend only on the present inputs, while outputs of *sequential logic* depend on both past and present inputs. In other words, combinational logic is memoryless, while sequential logic has memory or *state*.

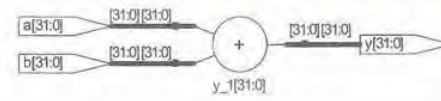
don't know whether a block of logic is combinational or sequential, you are likely to get the wrong thing. A particularly common mistake is to use `always` blocks to model combinational logic, but to accidentally imply latches or flip-flops.

This appendix focuses on a subset of Verilog sufficient to synthesize any hardware function. The language contains many other commands that are beyond the scope of this tutorial.

A.2 Behavioral Modeling with Continuous Assignments

A 32-bit adder is a complex design at the schematic level of representation. It can be constructed from 32 full adder cells, each of which in turn requires about six 2-input gates. Verilog provides a much more compact description. In each of the examples in this appendix, Synplify Pro was used to synthesize the Verilog into hardware. The Verilog code is shown adjacent to the schematic it implies.

```
module adder(input  [31:0] a,
             input  [31:0] b,
             output [31:0] y);

   assign y = a + b;
endmodule
```



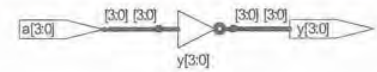
A Verilog module is like a cell in a schematic. It begins with a description of the inputs and outputs, which in this case are 32-bit busses.

During simulation, an `assign` statement causes the left side (`y`) to be updated any time the right side (`a/b`) changes. This necessarily implies combinational logic; the output on the left side is a function of the current inputs given on the right side. A 32-bit adder is a good example of combinational logic.

A.2.1 Bitwise Operators

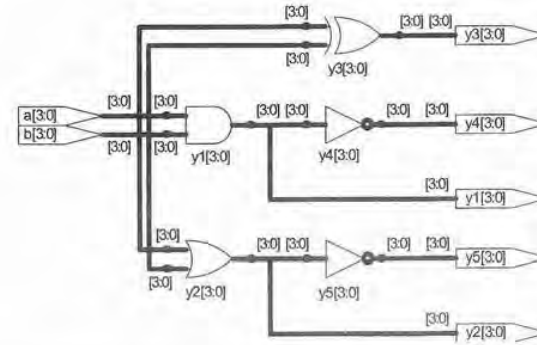
Verilog has a number of *bitwise* operators that act on busses. For example, the following module describes four inverters.

```
module inv(input  [3:0] a,
           output [3:0] y);

   assign y = ~a;
endmodule
```



Similar bitwise operations are available for the other basic logic functions:

```
module gates(input [3:0]  a, b,
             output [3:0] y1, y2, y3, y4, y5);

   /* Five different two-input logic
      gates acting on 4 bit busses */
   assign y1 = a & b;     // AND
   assign y2 = a | b;     // OR
   assign y3 = a ^ b;     // XOR
   assign y4 = ~(a & b); // NAND
   assign y5 = ~(a | b); // NOR
endmodule
```



A.2.2 Comments and White Space

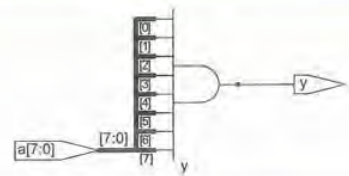
The previous examples showed two styles of comments, just like those used in C or Java. Comments beginning with `/*` continue, possibly across multiple lines, to the next `*/`. Comments beginning with `//` continue to the end of the line. It is important to properly comment complex logic so that six months from now, you can understand what you did or so that some poor slob assigned to fix your buggy code will be able to figure it out. ☺

Verilog is not picky about the use of white space. Nevertheless, proper indenting and spacing is helpful to make nontrivial designs readable. Verilog is case-sensitive. Be consistent in your use of capitalization and underscores in signal and module names. Be sparing with the use of underscores because they can increase the risk of carpal tunnel syndrome.

A.2.3 Reduction Operators

Reduction operators imply a multiple-input gate acting on a single bus. For example, the following module describes an 8-input AND gate with inputs `a[0]`, `a[1]`, `a[2]`, ..., `a[7]`.

```
module and8(input  [7:0] a,
            output       y);

   assign y = &a;
endmodule
```

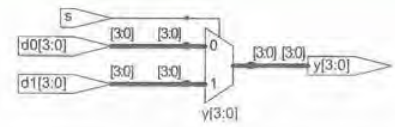


As one would expect, `|`, `^`, `~&`, and `~|` reduction operators are available for OR, XOR, NAND, and NOR as well. Recall that a multi-bit XOR performs parity, returning true if an odd number of inputs are true.

A.2.4 Other Operators

The conditional operator `?:` works like the same operator in C or Java and is useful for describing multiplexers. It is called a *ternary operator* because it takes three inputs. If the first input is nonzero, the result is the expression in the second input. Otherwise, the result is the expression in the third input.

```
module mux2(input  [3:0] d0, d1,
            input        s,
            output [3:0] y);

  assign y = s ? d1 : d0;
endmodule
```



A number of arithmetic functions are supported, including `+`, `-`, `*`, `<`, `>`, `<=`, `>=`, `==`, `!=`, `<<`, `>>`, `<<<`, `>>>`, `/`, and `%`. Recall from other languages that % is the modulo operator: `a%b` equals the remainder of `a` when divided by `b`. These operations imply a vast amount of hardware. `==` and `!=` (equality/inequality) on N-bit inputs require N 2-input XNORs to determine equality of each bit and an N-input AND or NAND to combine all the bits, as shown in Figure 10.52. Addition, subtraction, and comparison all require an adder, which is expensive in hardware. Variable logical left and right shifts `<<` and `>>` and arithmetic left and right shifts `<<<` and `>>>` imply a barrel shifter. Multipliers are even more costly. Do not use these statements without contemplating the number of gates you are generating. Moreover, the implementations are not necessarily efficient for your problem.

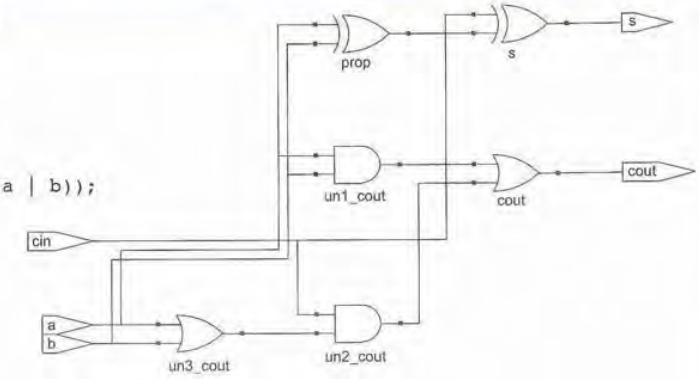
Some synthesis tools ship with optimized libraries for special functions like adders and multipliers. For example, the Synopsys DesignWare libraries produce reasonably good multipliers. If you do not have a license for the libraries, you'll probably be disappointed with the speed and gate count of a multiplier your synthesis tool produces from when it sees `*`. Many synthesis tools choke on `/` and `%` because these are nontrivial functions to implement in combinational logic.

A.3 Basic Constructs

A.3.1 Internal Signals

Often it is convenient to break a complex calculation into intermediate variables. For example, in a full adder, we sometimes define the propagate signal as the XOR of the two inputs A and B. The sum from the adder is the XOR of the propagate signal and the carry-in. We can declare the propagate signal using a `wire` statement, in much the same way we use local variables in a programming language.

```
module fulladder(input  a, b, cin,
                 output s, cout);

   wire prop;

   assign prop = a ^ b;
   assign s = prop ^ cin;
   assign cout = (a & b) | (cin & (a | b));
endmodule
```



Technically, it is not necessary to declare single-bit wires. However, it is necessary to declare multi-bit busses and is good practice to declare all signals. Some Verilog simulation and synthesis tools give errors that are difficult to decipher when a wire is not declared.

A.3.2 Precedence

Notice that we fully parenthesized the `cout` computation. We could take advantage of operator precedence to use fewer parentheses:

```
assign cout = a&b | cin&(a|b);
```

The operator precedence from highest to lowest is much as you would expect in other languages, as shown in Table A.1. AND has precedence over OR.

Table A.1 Operator Precedence

Symbol	Meaning	Precedence
~	NOT	Highest
*, /, %	MUL, DIV, MODULO	
+, -	PLUS, MINUS	
<<, >>,	Logical Left/Right Shift	
<<<, >>>	Arithmetic Left/Right Shift	
<, <=, >, >=	Relative Comparison	
==, !=	Equality Comparison	
&, ~&	AND, NAND	
^, ~^	XOR, XNOR	
|, ~|	OR, NOR	
?:	Conditional	Lowest

A.3.3 Constants

Constants can be specified in binary, octal, decimal, or hexadecimal. Table A.2 gives examples.

Table A.2 Constants

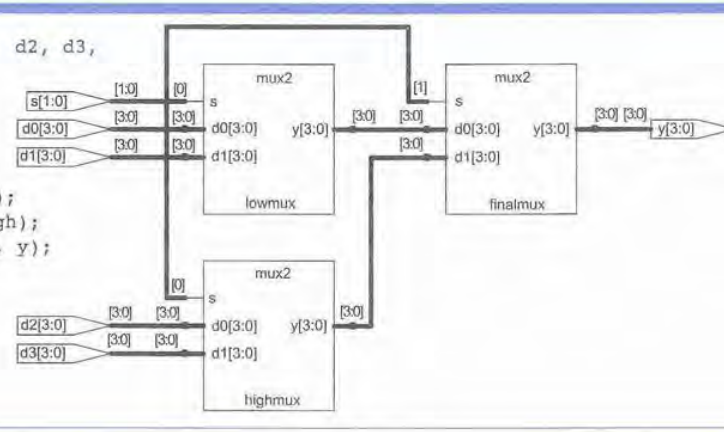
Number	# Bits	Base	Decimal Equivalent	Stored
`3'b101`	3	Binary	5	101
`'b11`	unsized	Binary	3	000000..00011
`8'b11`	8	Binary	3	00000011
`8'b1010_1011`	8	Binary	171	10101011
`3'd6`	3	Decimal	6	110
`6'o42`	6	Octal	34	100010
`8'hAB`	8	Hexadecimal	171	10101011
`42`	unsized	Decimal	42	0000...00101010

It is good practice to specify the length of the number in bits, even though the second row shows that this is not strictly necessary. If you don't specify the length, one day you may be surprised when Verilog assumes the constant has additional leading 0's that you didn't intend. Underscores in numbers are ignored and can be helpful in breaking long numbers into more readable chunks. If the base is omitted, the number is assumed to be decimal.

A.3.4 Hierarchy

Nontrivial designs are developed in a hierarchical form, in which complex modules are composed of submodules. For example, a 4-input multiplexer can be constructed from three 2-input multiplexers:

```
module mux4(input  [3:0] d0, d1,  d2, d3,
            input  [1:0] s,
            output [3:0] y);

   wire [3:0] low, high;

   mux2 lowmux(d0, d1, s[0], low);
   mux2 highmux(d2, d3, s[0], high);
   mux2 finalmux(low, high, s[1], y);
endmodule
```

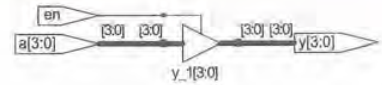


This is an example of the structural coding style because the mux is built from simpler modules. It is good practice to avoid (or at least minimize) mixing structural and behavioral descriptions within a single module. Generally, simple modules are described behaviorally and larger modules are composed structurally from these building blocks.

A.3.5 Tristates

It is possible to leave a bus floating rather than drive it to 0 or 1. This floating value is called `'z` in Verilog. For example, a tristate buffer produces a floating output when the enable is false.

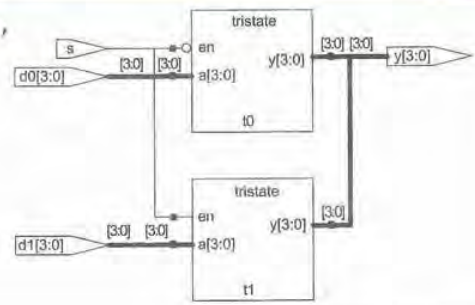
```
module tristate(input  [3:0] a,
                input        en,
                output [3:0] y);

  assign y = en ? a : 4'bz;
endmodule
```



Floating inputs to gates causes undefined outputs, displayed as `'x` in Verilog. At start-up, state nodes such as the internal node of flip-flops are also usually initialized to `'x`, as we will see later.

We could define a multiplexer using two tristates so that the output is continuously driven by exactly one tristate. This guarantees that there are no floating nodes.

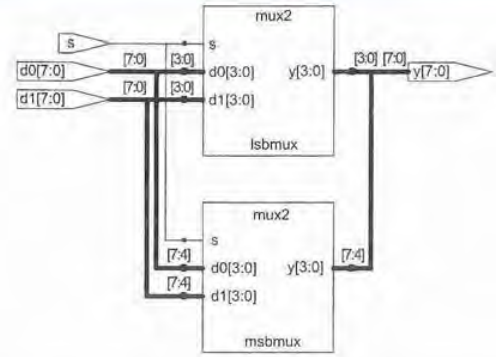
```
module mux2(input  [3:0] d0, d1,
            input        s,
            output [3:0] y);

  tristate t0(d0, ~s, y);
  tristate t1(d1, s, y);
endmodule
```



A.3.6 Bit Swizzling

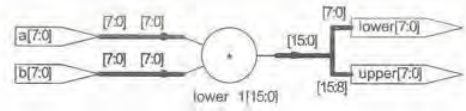
Often it is necessary to operate on parts of a bus or to concatenate (join together) signals to construct busses. In the `mux4` example, the least significant bit `s[0]` of a 2-bit select signal was used for the low and high muxes and the most significant bit `s[1]` was used for the final mux. Use ranges to select subsets of a bus. For example, an 8-bit wide 2-input mux can be constructed from two 4-bit wide 2-input muxes.

```
module mux2_8(input  [7:0] d0, d1,
              input        s,
              output [7:0] y);

   mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
   mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
endmodule
```



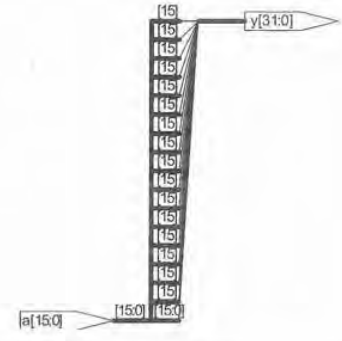
The {} notation is used to concatenate busses. For example, the following 8 × 8 multiplier produces a 16-bit result, which is placed on the upper and lower 8-bit result busses.

```
module mul(input [7:0] a, b,
           output [7:0] upper, lower);

   assign {upper, lower} = a*b;
endmodule
```



A 16-bit 2's complement number is sign-extended to 32 bits by copying the most significant bit to each of the upper 16 positions. The Verilog syntax concatenates 16 copies of `a[15]` to the 16-bit `a[15:0]` bus. Some synthesis tools produce a warning that `a` is a "feedthrough net." This means that the input "feeds through" to the output. `y[15:0]` should have the same value as `a[15:0]`, so we did intend a feedthrough. If you get a feedthrough net warning where you did not intend one, check for a mistake in your Verilog.

```
module signextend(input  [15:0] a,
                  output [31:0] y);

   assign y = {{16{a[15]}}, a[15:0]};
endmodule
```



The next statement generates a silly combination of two busses. Don't confuse the 3-bit binary constant `3'b101` with bus `b`. Note that it was important to specify the length of 3 bits in the constant; otherwise many additional 0's might have appeared in the middle of `y`.

```
assign y = {a[2:1], {3{b[0]}}, a[0], 3'b101, b[1:3]};
```

This produces

```
y = a[2] a[1] b[0] b[0] b[0] a[0] 1 0 1 b[1] b[2] b[3]
```

A.3.7 Delays

The delay of a statement can be specified in arbitrary units. For example, the following code defines an inverter with a 42-unit propagation delay. Delays have no impact on synthesis, but can be helpful while debugging simulation waveforms because they make cause and effect more apparent.

```
assign #42 y = ~a;
```

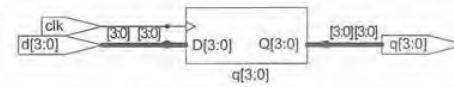
A.4 Behavioral Modeling with `always` Blocks

`Assign` statements are reevaluated every time any term on the right side changes. Therefore, they must describe combinational logic. `always` blocks are reevaluated only when signals in the header (called a *sensitivity list*) change. Depending on the form, `always` blocks can imply either sequential or combinational circuits.

A.4.1 Registers

Verilog refers to edge-triggered flip-flops as *registers*. Registers are described with an `always @(posedge clk)` statement:

```
module flop(input            clk,
            input      [3:0] d,
            output reg [3:0] q);

  always @(posedge clk)
    q <= d;
endmodule
```



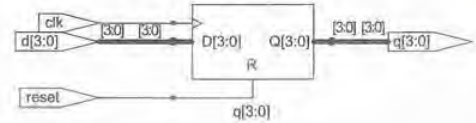
The body of the `always` statement is only evaluated on the rising (positive) edge of the clock. At this time, the output `q` is copied from the input `d`. The `<=` is called a *nonblocking assignment* and is pronounced *"gets,"* as in *"q gets d."* Think of it as a regular equals sign

for now; we'll return to the more subtle points in Section A.4.6. Notice that <= is used instead of `assign` inside the `always` block.

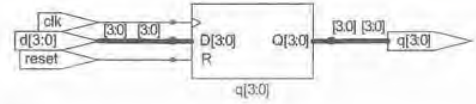
All the signals on the left side of assignments in `always` blocks must be declared as `reg`. This is a confusing point for new Verilog users. In this circuit, `q` is also the output. Declaring a signal as `reg` does not mean the signal is actually a register! All it means is that it appears on the left side in an `always` block. We will see examples of combinational signals later that are declared `reg`, but that have no flip-flops.

At startup, the `q` output is initialized to `'x`. Generally, it is good practice to use resettable registers so that on power-up you can put your system in a known state. The reset can be either asynchronous or synchronous, as discussed in Section 7.3.4. Asynchronous resets occur immediately. Synchronous resets only change the output on the rising edge of the clock.

```
module flopr(input            clk,
             input            reset,
             input      [3:0] d,
             output reg [3:0] q);

   // asynchronous reset
   always @(posedge clk, posedge reset)
      if (reset) q <= 4'b0;
      else       q <= d;
endmodule
```



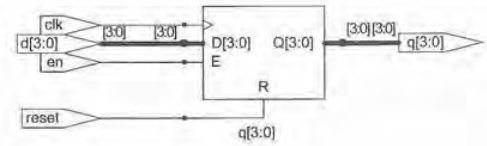

```
module flopr(input            clk,
             input            reset,
             input      [3:0] d,
             output reg [3:0] q);

   // synchronous reset
   always @(posedge clk)
      if (reset) q <= 4'b0;
      else       q <= d;
endmodule
```



Note that the asynchronously resettable flop evaluates the always block when either `clk` or `reset` rise so that it immediately responds to `reset`. The synchronously reset flop is not sensitized to `reset` in the `@` list, so it waits for the next clock edge before clearing the output.

You can also consider registers with enables that only respond to the clock when the enable is true. The following register with enable and asynchronous reset retains its old value if both `reset` and `en` are false.

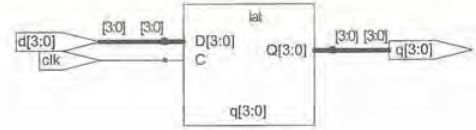
```
module flopren(input            clk,
               input            reset,
               input            en,
               input      [3:0] d,
               output reg [3:0] q);

   // asynchronous reset
   always @(posedge clk, posedge reset)
      if (reset) q <= 4'b0;
      else if (en) q <= d;
endmodule
```



A.4.2 Latches

`Always` blocks can be used to model transparent latches, also known as D latches When the clock is high, the latch is *transparent* and the data input flows to the output. When the clock is low, the latch goes *opaque* and the output remains constant.

```
module latch(input            clk,
             input      [3:0] d,
             output reg [3:0] q);

   always @(clk, d)
      if (clk) q <= d;
endmodule
```



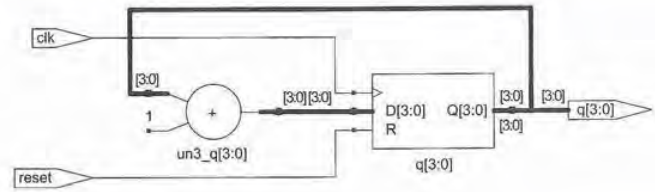
The latch evaluates the `always` block any time either `clk` or `d` change. If `clk` is high, the output gets the input. Notice that even though `q` is a latch node, not a register node, it is still declared as `reg` because it is on the left side of a `<=` in an `always` block. Some synthesis tools are primarily intended to target edge-triggered flip-flops and will produce warnings when generating latches.

A.4.3 Counters

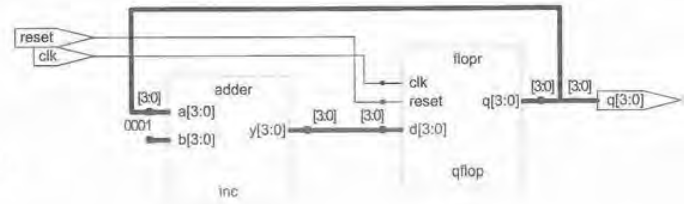
Consider two ways of describing a 4-bit counter with asynchronous reset. The first scheme (behavioral) implies a sequential circuit containing both the 4-bit register and an adder. The second scheme (structural) explicitly declares modules for the register and adder.

Either scheme is good for a simple circuit such as a counter. As you develop more complex finite state machines, it is a good idea to separate the next state logic from the registers in your Verilog code. Verilog does not protect you from yourself here and there are many simple errors that lead to circuits unlike those you intended, as will be explored in Section A.9.

```
module counter(input            clk,
               input            reset,
               output reg [3:0] q);

   // counter using always block

   always @(posedge clk)
      if (reset) q <= 4'b0;
      else       q <= q+1;
endmodule
```



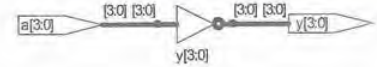

```
module counter(input        clk,
               input        reset,
               output [3:0] q);

   wire [3:0] nextq;

   // counter using module calls

   flopr qflop(clk, reset, nextq, q);
   adder inc(q, 4'b0001, nextq);
   // assumes a 4-bit adder
endmodule
```



A.4.4 Combinational Logic

`Always` blocks imply sequential logic when some of the inputs do not appear in the `@` stimulus list or might not cause the output to change. For example, in the flop module, `d` is not in the `@` list, so the flop does not immediately respond to changes of `d`. In the latch, `d` is in the `@` list, but changes in `d` are ignored unless `clk` is high. `Always` blocks can also be used to imply combinational logic if they are written in such a way that the output is reevaluated every time there are changes in any of the inputs. The following code shows how to define a bank of inverters with an `always` block. Note that `y` must be declared as `reg` because it appears on the left side of a `<=` or `=` sign in an `always` block. Nevertheless, `y` is the output of combinational logic, not a register.

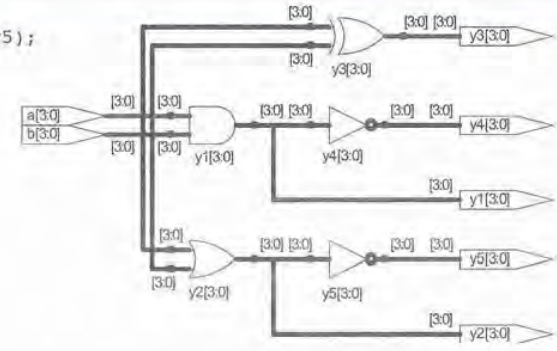
```
module inv(input      [3:0] a,
           output reg [3:0] y);

   always @(*)
      y <= ~a;
endmodule
```



`always @(*)` evaluates the statements inside the `always` block whenever any of the signals on the right side of <= or = change inside the `always` block. Thus `@(*)` is a safe way to model combinational logic. In this particular example, `@(a)` would also have sufficed.

Similarly, the next example defines five banks of different kinds of gates. In this case, an `@(a, b)` would have been equivalent to `@(*)`. However, `@(*)` is better because it avoids common mistakes of missing signals in the stimulus list. Also notice that the `begin` / `end` construct is necessary because multiple commands appear in the `always` block. This is analogous to { } block structure in C or Java. The `begin` / `end` was not needed in the `flopr` example because an `if` / `else` command counts as a single statement.

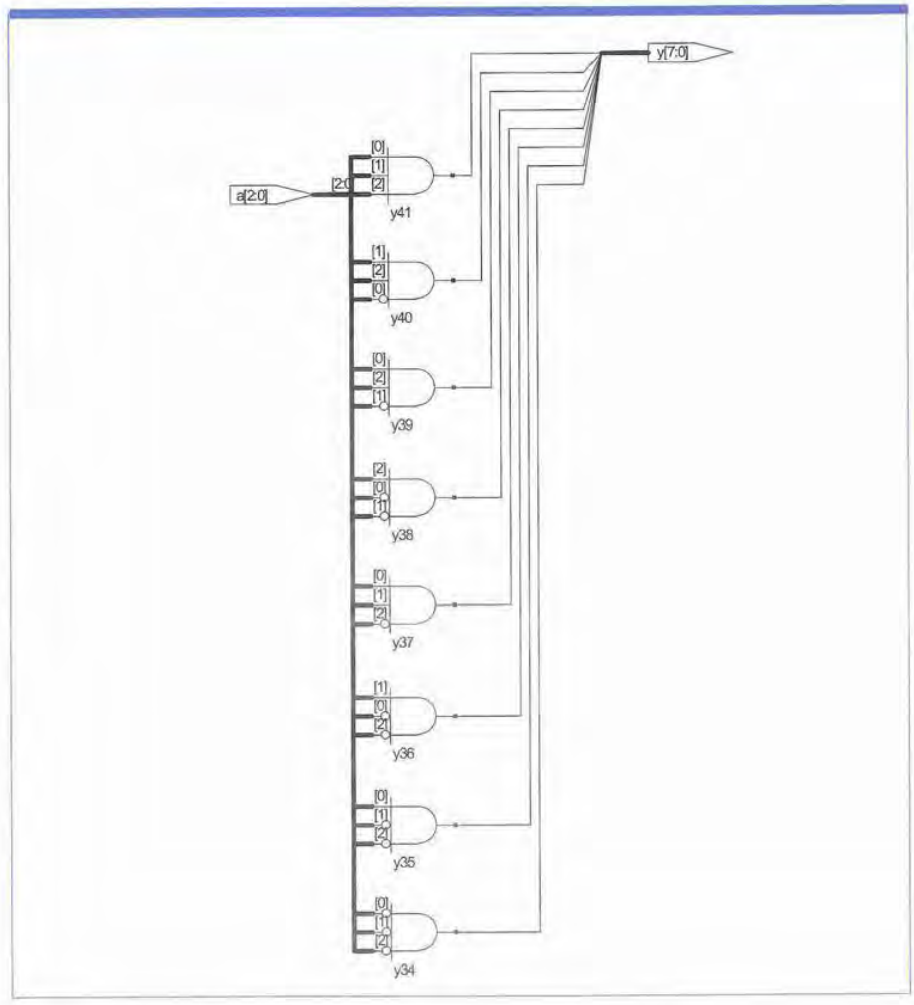
```
module gates(input      [3:0] a, b,
             output reg [3:0] y1, y2, y3, y4, y5);

   always @(*)
      begin
         y1 <= a & b; // AND
         y2 <= a | b; // OR
         y3 <= a ^ b; // XOR
         y4 <= ~(a & b); // NAND
         y5 <= ~(a | b); // NOR
      end
endmodule
```



These two examples are poor applications of `always` blocks for modeling combinational logic because they require more lines than the equivalent approach with `assign` statements. Moreover, they pose the risk of inadvertently implying sequential logic (see Section A.9.2). A better application of the `always` block is a decoder, which takes advantage of the `case` statement that can only appear inside an `always` block.

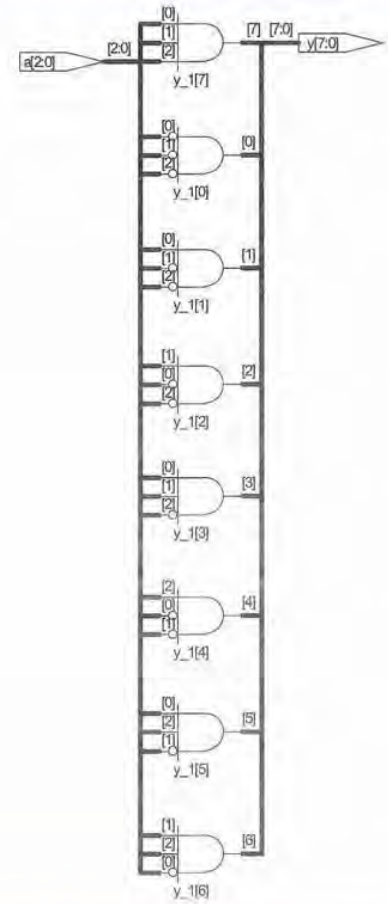
```
module decoder_always(input      [2:0] a,
                      output reg [7:0] y);

   // a 3:8 decoder
   always @(*)
      case (a)
         3'b000:  y <= 8'b00000001;
         3'b001:  y <= 8'b00000010;
         3'b010:  y <= 8'b00000100;
         3'b011:  y <= 8'b00001000;
         3'b100:  y <= 8'b00010000;
         3'b101:  y <= 8'b00100000;
         3'b110:  y <= 8'b01000000;
         3'b111:  y <= 8'b10000000;
      endcase
endmodule
```

continued



Using the `case` statement is probably clearer than using Boolean equations in an `assign` statement:

```
module decoder_assign(input  [2:0] a,
                      output [7:0] y);

  assign y[0] = ~a[0] & ~a[1] & ~a[2];
  assign y[1] =  a[0] & ~a[1] & ~a[2];
  assign y[2] = ~a[0] &  a[1] & ~a[2];
  assign y[3] =  a[0] &  a[1] & ~a[2];
  assign y[4] = ~a[0] & ~a[1] &  a[2];
  assign y[5] =  a[0] & ~a[1] &  a[2];
  assign y[6] = ~a[0] &  a[1] &  a[2];
  assign y[7] =  a[0] &  a[1] &  a[2];
endmodule
```



An even better example is the logic for a 7-segment display decoder from [Ciletti99]. The 7-segment display is shown in Figure A.1. The decoder takes a 4-bit number and displays its decimal value on the segments. For example, the number 0111 = 7 should turn on segments `a`, `b`, and `c`. The equivalent logic with `assign` statements describing the detailed logic for each bit would be tedious. This more abstract approach is faster to write, clearer to read, and can be automatically synthesized down to an efficient logic implementation. This example also illustrates the use of parameters to define constants to make the code more readable. The `case` statement has a `default` to display a blank output when the input is outside the range of decimal digits.

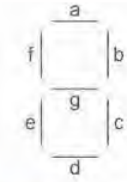
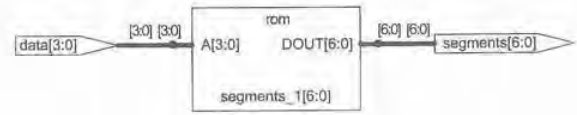


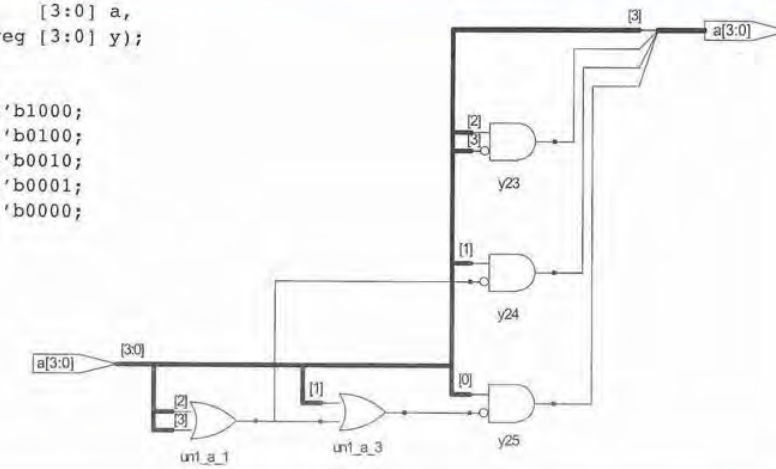
FIG A.1 7-segment display mapping

```
module sevenseg(input      [3:0] data,
                output reg [6:0] segments);

   // Segment  #          abc_defg
   parameter   BLANK = 7'b000_0000;
   parameter   ZERO  = 7'b111_1110;
   parameter   ONE   = 7'b011_0000;
   parameter   TWO   = 7'b110_1101;
   parameter   THREE = 7'b111_1001;
   parameter   FOUR  = 7'b011_0011;
   parameter   FIVE  = 7'b101_1011;
   parameter   SIX   = 7'b101_1111;
   parameter   SEVEN = 7'b111_0000;
   parameter   EIGHT = 7'b111_1111;
   parameter   NINE  = 7'b111_1011;

   always @(*)
      case (data)
         0: segments <= ZERO;
         1: segments <= ONE;
         2: segments <= TWO;
         3: segments <= THREE;
         4: segments <= FOUR;
         5: segments <= FIVE;
         6: segments <= SIX;
         7: segments <= SEVEN;
         8: segments <= EIGHT;
         9: segments <= NINE;
         default: segments <= BLANK;
      endcase
endmodule
```

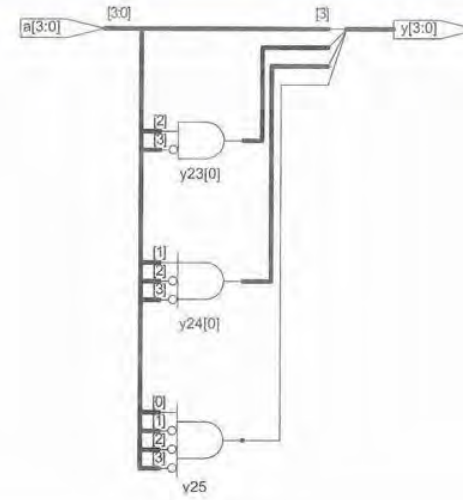


Finally, compare three descriptions of a priority encoder that sets one output true corresponding to the most significant input that is true. The `if` statement can appear in `always` blocks and makes the logic quite readable. The `casez` statement also appears in `always` blocks and allows don't care's in the case logic, indicated with the `?` symbol. The `assign` statements synthesize to the same results, but are arguably less clear to read. Note that `a[3]` is another example of a feedthrough net because `y[3] = a[3]`. Of these three styles, the `if/else` approach is recommended for describing priority encoders because it is the easiest for most engineers to recognize. `Case` statements are best reserved for functions specified by truth tables and `casez` statements should be used for functions specified by truth tables with don't cares. In the first two descriptions, `y` must be declared as a `reg` because it is assigned inside an `always` block.

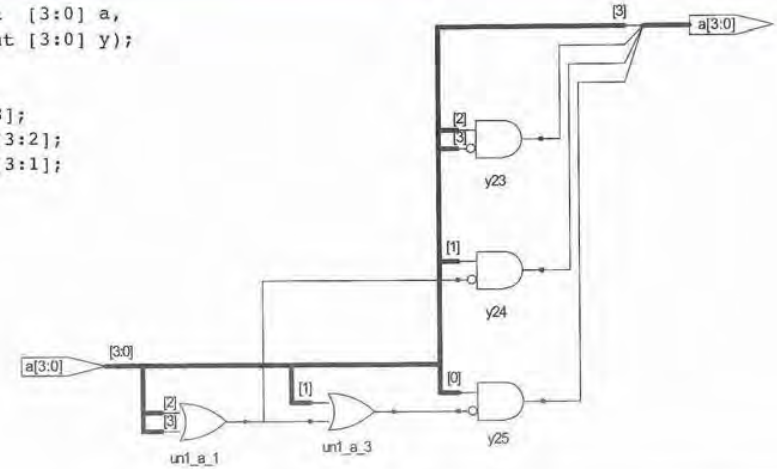
```
module priority_if(input      [3:0] a,
                   output reg [3:0] y);

   always @(*)
      if      (a[3]) y <= 4'b1000;
      else if (a[2]) y <= 4'b0100;
      else if (a[1]) y <= 4'b0010;
      else if (a[0]) y <= 4'b0001;
      else           y <= 4'b0000;
endmodule
```




```
module priority_casez(input      [3:0] a,
                      output reg [3:0] y);

   always @(*)
      casez(a)
         4'b1???: y <= 4'b1000;
         4'b01??: y <= 4'b0100;
         4'b001?: y <= 4'b0010;
         4'b0001: y <= 4'b0001;
         default: y <= 4'b0000;
      endcase
endmodule
```



```
module priority_assign(input  [3:0] a,
                       output [3:0] y);

   assign y[3] = a[3];
   assign y[2] = a[2] & ~a[3];
   assign y[1] = a[1] & ~|a[3:2];
   assign y[0] = a[0] & ~|a[3:1];
endmodule
```

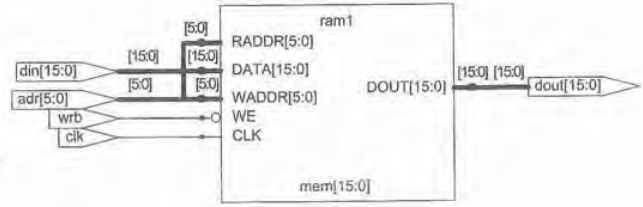


It is easy to accidentally imply sequential logic with `always` blocks when combinational logic is intended. The resulting bugs can be difficult to track down. Therefore, to imply combinational logic, it is safer to use `assign` statements than `always` blocks. Nevertheless, the convenience of constructs such as `if` or `case` that must appear in `always` blocks justifies the modeling style as long as you thoroughly understand what you are doing.

A.4.5 Memories

Verilog has an array construct used to describe memories. The following module describes a 64-word × 16-bit synchronous RAM that is written on the positive edge of the clock when `wrb` is low. The internal signal `mem` is declared as `reg` rather than `wire` because it is assigned inside an `always` block.

```
module ram(input         clk,
           input  [5:0]  addr,
           input         wrb,
           input  [15:0] din,
           output [15:0] dout);

   reg  [15:0] mem[63:0]; // the memory

   always @(posedge clk)
      if (~wrb) mem[addr] <= din;

   assign dout = mem[addr];
endmodule
```



Synthesis tools are often restricted to generating gates from a library and produce poor memory arrays. A specialized memory generator is commonly used instead.

A.4.6 Blocking and Nonblocking Assignment

Verilog supports two types of assignments inside an `always` block. *Blocking assignments* use the = statement. *Nonblocking assignments* use the <= statement. Do not confuse either type with the `assign` statement, which cannot appear inside `always` blocks at all.

A group of blocking assignments inside a `begin/end` block are evaluated sequentially, just as you would expect in a standard programming language. A group of nonblocking assignments are evaluated in parallel; all of the statements are evaluated before any of the left sides are updated. This is what you would expect in hardware because real logic gates all operate independently rather than waiting for the completion of other gates.

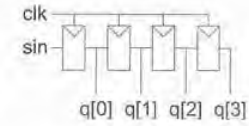
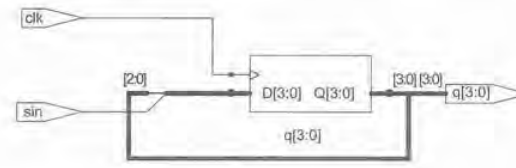


FIG A.2 Intended shift register

For example, consider two attempts to describe a shift register. On each clock edge, the data at `sin` should be shifted into the first flop, as shown in Figure A.2. The first flop shifts to the second flop. The data in the second flop shifts to the third flop, and so on until the last element drops off the end.

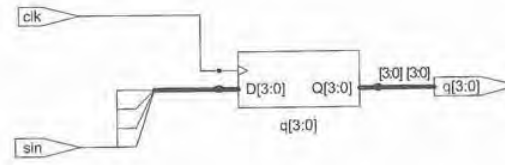
```
module shiftreg(input            clk,
                input            sin,
                output reg [3:0] q);

  // This is a correct implementation
  // using nonblocking assignment

  always @(posedge clk)
    begin
      q[0] <= sin; // nonblocking <=
      q[1] <= q[0];
      q[2] <= q[1];
      q[3] <= q[2];
      // even better to write
      // q <= {q[2:0], sin};
    end
endmodule
```



The nonblocking assignments mean that all of the values on the right sides are assigned simultaneously. Therefore, `q[1]` will get the original value of `q[0]`, not the value of `sin` that gets loaded into `q[0]`. This is what we would expect from real hardware. Of course, all of this could be written on one line for brevity.

Blocking assignments are more familiar from traditional programming languages, but they inaccurately model hardware. Consider the same module using blocking assignments. When `clk` rises, the Verilog says that `q[0]` should be copied from `sin`. Then `q[1]` should be copied from the new value of `q[0]` and so forth. All four registers immediately get the `sin` value.

```
module shiftreg(input            clk,
                input            sin,
                output reg [3:0] q);

   // This is a bad implementation
   // using blocking assignment

   always @(posedge clk)
      begin
         q[0] = sin; // blocking =
         q[1] = q[0];
         q[2] = q[1];
         q[3] = q[2];
      end
endmodule
```



The moral of this illustration is to use nonblocking assignments in `always` blocks when modeling sequential logic. Using sufficient ingenuity, such as reversing the orders of the four commands, you could make blocking assignments work correctly, but they offer no advantages and harbor great risks.

Finally, note that each `always` block implies a separate block of logic. Therefore, a given `reg` can be assigned in only one `always` block. Otherwise, two pieces of hardware with shorted outputs will be implied.

A.5 Finite State Machines

There are two styles of finite state machines. In *Mealy machines* (Figure A.3(a)), the output is a function of the current state and inputs. In *Moore machines* (Figure A.3(b)), the output is a function of only the current state.

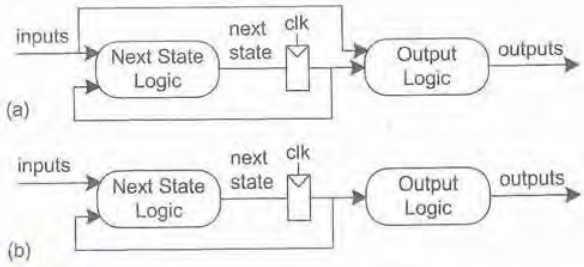


FIG A.3 Moore and Mealy machines

FSMs are modeled in Verilog with an `always` block defining the state registers and combinational logic defining the next state and output logic.

Let us first consider a simple finite state machine with one output and no inputs, a divide-by-3 counter. The output should be asserted every three clock cycles. The state transition diagram for a Moore machine is shown in Figure A.4. The output value is labeled in each state because the output is only a function of the state.

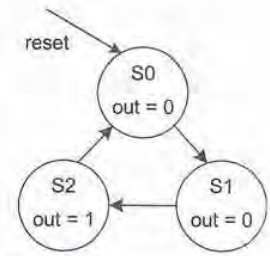
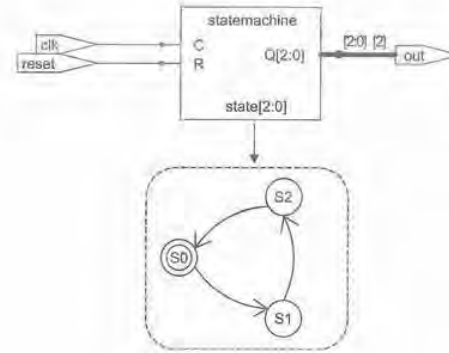


FIG A.4 Divide-by-3 counter state transition diagram

```
module divideby3FSM(input  clk,
                    input  reset,
                    output out);

   reg  [1:0] state, nextstate;

   parameter S0 = 2'b00;
   parameter S1 = 2'b01;
   parameter S2 = 2'b10;

   // State Register
   always @(posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;

   // Next State Logic
   always @(*)
      case (state)
         S0: nextstate <= S1;
         S1: nextstate <= S2;
         S2: nextstate <= S0;
         default: nextstate <= S0;
      endcase

   // Output Logic
   assign out = (state == S2);
endmodule
```



The FSM model is divided into three portions: the *state register*, *next state logic*, and *output logic*. The state register logic describes an asynchronously resettable register that resets to an initial state and otherwise advances to the computed next state. Defining states with parameters allows the easy modification of state encodings and makes the code easier to read. The next state logic computes the next state as a function of the current state and

inputs; in this example, there are no inputs. A `case` statement in an `always @(state or inputs)` block is a convenient way to define the next state. It is important to have a `default` if not all cases are enumerated; otherwise `nextstate` would not be assigned in the undefined cases. This implies that `nextstate` should keep its old value, which would require the existence of latches. Finally, the output logic may be a function of the current `state` alone in a Moore machine or of the current `state` and inputs in a Mealy machine. Depending on the complexity of the design, `assign` statements, `if` statements, or `case` statements may be most readable and efficient.

The next example shows a finite state machine with an input `a` and two outputs. Output `x` is true when the input is the same now as it was last cycle. Output `y` is true when the input is the same now as it was for the past two cycles. This is a Mealy machine because the output depends on the current inputs as well as the state. The outputs are labeled on each transition after the input. The state transition diagram is shown in Figure A.5.

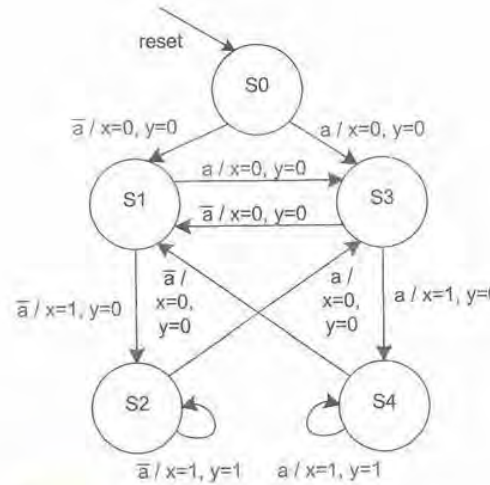
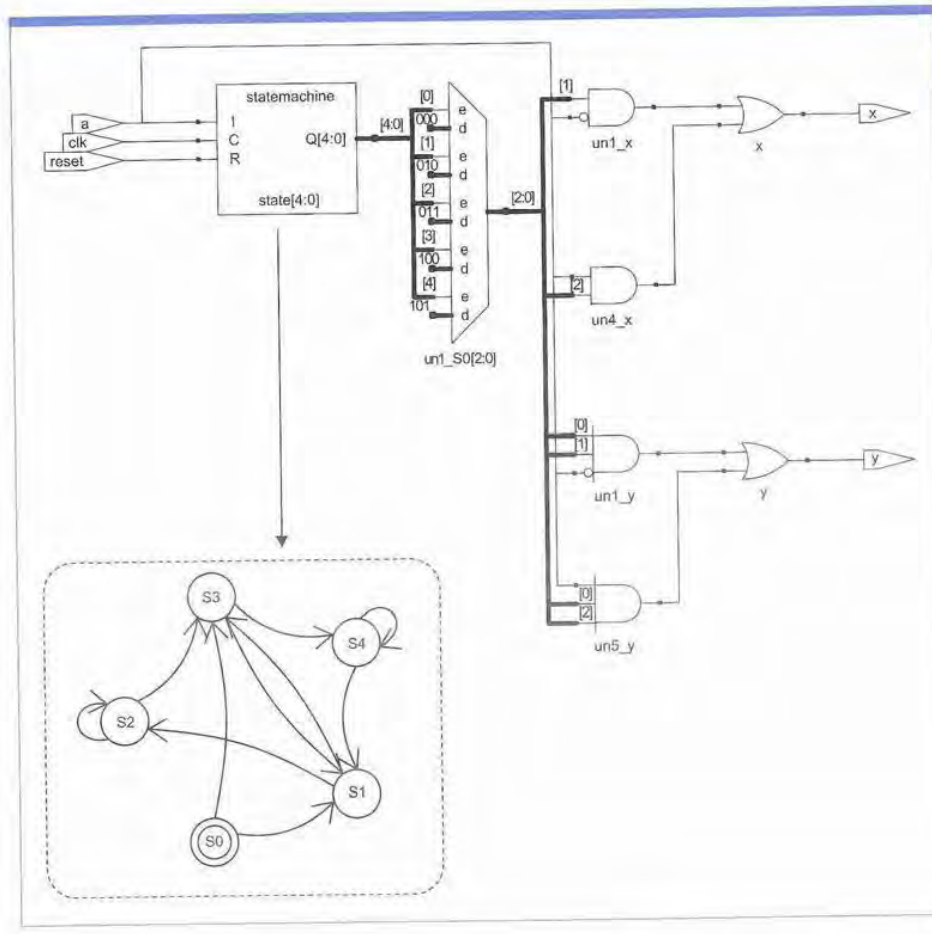


FIG A.5 History FSM state transition diagram

```
module historyFSM(input  clk,
                  input  reset,
                  input  a,
                  output x, y);

   reg  [2:0] state, nextstate;

   parameter S0 = 3'b000;
   parameter S1 = 3'b010;
   parameter S2 = 3'b011;
   parameter S3 = 3'b100;
   parameter S4 = 3'b101;

   // State Register
   always @(posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;

   // Next State Logic
   always @(*)
      case (state)
         S0: if (a) nextstate <= S3;
             else   nextstate <= S1;
         S1: if (a) nextstate <= S3;
             else   nextstate <= S2;
         S2: if (a) nextstate <= S3;
             else   nextstate <= S2;
         S3: if (a) nextstate <= S4;
             else   nextstate <= S1;
         S4: if (a) nextstate <= S4;
             else   nextstate <= S1;
         default: nextstate <= S0;
      endcase

   // Output Logic
   assign x = (state[1] & ~a) |
              (state[2] & a);
   assign y = (state[1] & state[0] & ~a) |
              (state[2] & state[0] & a);
endmodule
```

continued



The output logic equations depend on the specific state encoding and were worked out by hand. A more general approach is independent of the encodings and requires less thinking, but more code:

```
// Output Logic
always @(state or a)
   case (state)
      S0:  begin
              x <= 0; y <= 0;
           end
      S1:  if (a) begin
              x <= 0; y <= 0;
           end else begin
              x <= 1; y <= 0;
           end
      S2:  if (a) begin
              x <= 0; y <= 0;
           end else begin
              x <= 1; y <= 1;
           end
      S3:  if (a) begin
              x <= 1; y <= 0;
           end else begin
              x <= 0; y <= 0;
           end
      S4:  if (a) begin
              x <= 1; y <= 1;
           end else begin
              x <= 0; y <= 0;
           end
   endcase
```

You may be tempted so simplify the `case` statement. For example, case `S4` could be reduced to:

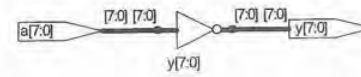
```
// bad simplification of S4
S4:   if (a) begin
         y <= 1;
      end else begin
         x <= 0; y <= 0;
      end
```

The designer reasons that to get to state `S4`, we must have passed through state `S3` with `a` high, setting `x` high. Therefore, the assignment of `x` is optimized out of `S4` when `a` is high. This is incorrect reasoning. The modified approach implies sequential logic. Specifically, a latch is implied that holds the old value of `x` when `x` is not assigned. The latch holds its output under a peculiar set of circumstances; `a` and the state must be used to compute the latch clock signal. This is undoubtedly not what you wanted, but was easy to inadvertently imply. The moral of this example is that if any signal gets assigned in any branch of an `if` or `case` statement, it must be assigned in all branches lest a latch be implied.

A.6 Parameterized Modules

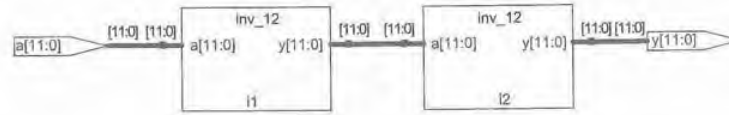
So far, all of our modules have had fixed-width inputs and outputs. Thus, we have needed separate modules for 4- and 8-bit wide 2-input multiplexers. Verilog permits variable bit widths using *parameterized modules*. For example, we can declare a parameterized bank of inverters with a default of 8 gates as:

```
module inv
   #(parameter width = 8)
    (input  [width-1:0] a,
     output [width-1:0] y);

   assign y = ~a;
endmodule
```



We can adjust the parameter when we instantiate the block. For example, we can build a bank of 12 buffers using a pair of banks of 12 inverters.

```
module buffer
   #(parameter numbits = 12)
    (input  [numbits-1:0] a,
     output [numbits-1:0] y);

   wire [numbits-1:0] x;

   inv #(numbits) i1(a, x);
   inv #(numbits) i2(x, y);
endmodule
```



A.7 Structural Primitives

When coding at the structural level, primitives exist for basic logic gates and transistors. Examples for a full adder carry circuit (majority gate) were given in Section 1.8.4.

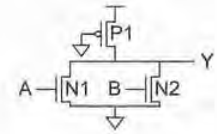
Gate primitives include `not`, `and`, `or`, `xor`, `nand`, `nor`, and `xnor`. The output is declared first; multiple inputs may follow. For example, a 4-input AND gate may be given as

```
and g1(y, a, b, c, d);
```

Transistor primitives include `tranif1`, `tranif0`, `rtranif1`, and `rtranif0`. `tranif1` is an nMOS transistor that turns ON when the gate is '1' while `tranif0` is a pMOS transistor. The `rtranif` primitives are resistive transistors, i.e., weak transistors that can be overcome by a stronger driver. For example, a pseudo-nMOS NOR gate with a weak pull-up

is modeled with three transistors. Most synthesis tools map only onto gates, not transistors, so these transistor primitives are only for simulation.

```
module nor2(input  a, b,
            output y);

   tranif1  n1(y, gnd, a);
   tranif1  n2(y, gnd, b);
   rtranif0 p1(y, vdd, gnd);
endmodule
```



The `tranif` devices are bi-directional; that is, the source and drain are symmetric. Verilog also supports unidirectional `nmos` and `pmos` primitives that only allow a signal to flow from the input terminal to the output terminal. Real transistors are inherently bi-directional, so unidirectional models can result in simulation not catching bugs that would exist in real hardware. Therefore, `tranif` primitives are preferred for simulation.

A.8 Test Benches

Verilog models are tested through simulation. For small designs, it may be practical to manually apply inputs to a simulator and visually check for the correct outputs. For larger designs, this procedure is usually automated with a *test bench*.

The following code shows an adder and its associated test bench. The test bench uses nonsynthesizable system calls to read a file, apply the *test vectors* to the *device under test* (DUT), check the results, and report any discrepancies. The `initial` statement defines a block that is executed only on startup of simulation. In it, the `$readmemh` reads a file (in hexadecimal form) into an array in memory. The `testvector.tv` example file shows four test vectors, each consisting of two 32-bit inputs and an expected 32-bit output. As the array is much larger than the number of test vectors, the remaining entries are filled with x's. The next `always` block defines a clock that repeats forever, being low for 50 units of time, then high for 50. On each positive edge of the clock, the next test vector is applied to the inputs and the expected output is saved. The actual output is sampled on the negative edge of the clock to permit some time for it to settle. It is not uncommon for bad logic to generate `z` and `x` values. The `!=` and `==` comparison operations return `x` if either argument has `z` or `x`, so they are not reliable. The `!==` and `===` commands check for an exact match, including `z` or `x` values. If there is a mismatch, the `$display` command is used to print the discrepancy. Each time `vectornum` is incremented, the test bench checks to see if the test is complete. It terminates testing with the `$finish` command after 100 vectors have been applied or the test vector consists of x's; for our `testvector.tv` file, it will end after the four vectors are applied.

```
module adder(a, b, y);
    input      [31:0]     a, b;
    output     [31:0]     y;

    assign y = a + b;
endmodule

module testbench();
    reg     [31:0] testvectors[1000:0];
    reg            clk;
    reg     [10:0] vectornum, errors;
    reg     [31:0] a, b, expectedy;
    wire    [31:0] y;

    // instantiate device under test
    adder  dut(a, b, y);

    // read the test vector file and initialize test
    initial
        begin
            $readmemh("testvectors.tv", testvectors);
            vectornum = 0; errors = 0;
        end

    // generate a clock to sequence tests
    always
        begin
            clk = 0; #50; clk = 1; #50;
        end

    // on each clock step, apply next test
    always @(posedge clk)
        begin
            a = testvectors[vectornum*3];
            b = testvectors[vectornum*3 + 1];
            expectedy = testvectors[vectornum*3 + 2];
        end

    // then check for correct results
    always @(negedge clk)
        begin
            vectornum = vectornum + 1;
            if (y !== expectedy) begin
                $display("Inputs were %h, %h", a, b);
                $display("Expected %h but actual %h", expectedy, y);
                errors = errors + 1;
            end
        end
```

```
    // halt at the end of file
    always @(vectornum)
        begin
            if (vectornum == 100 || testvectors[vectornum*3] === 32'bx)
                begin
                    $display("Completed %d tests with %d errors. ",
                                vectornum, errors);
                    $finish;
                end
        end
endmodule

testvectors.tv file:
00000000
00000000
00000000

00000001
00000000
00000001

ffffffff
00000003
00000002

12345678
12345678
2468acf0
```

A.9 Pitfalls

This section includes a set of style guidelines and examples of a number of bad circuits produced by common Verilog coding errors. The examples include the warnings given by Synopsys Design Compiler and Synplify Pro synthesis tools.

A.9.1 Verilog Style Guidelines

1. Use only nonblocking assignments inside `always` blocks.
2. Define your combinational logic using `assign` statements when practical. Only use `always` blocks to define combinational logic if constructs like `if` or `case` make your logic much clearer or more compact.
3. When modeling combinational logic with an `always` block, if a signal is assigned in any branch of an `if` or `case` statement, it must be assigned in all branches.

4. Include `default` cases in your `case` statements.
5. Partition your design into leaf cells and non-leaf cells. Leaf cells contain behavioral code (`assign` statements or `always` blocks), but do not instantiate other cells. Non-leaf cells contain structural code (i.e., they instantiate other cells, but contain no logic). Minor exceptions to this guideline can be made to keep the code readable.
6. Use parameters to define state names and constants.
7. Properly indent your code, as shown in the examples in this guide.
8. Use comments liberally.
9. Use meaningful signal names. Use `a, b, c, ...` for generic logic gate inputs. Use `x, y, z` for generic combinational outputs and `q` for a generic state element output. Use descriptive names for nongeneric cells. Do not use `foo`, `bar`, or `baz`!
10. Be consistent in your use of capitalization and underscores.
11. Do not ignore synthesis warnings unless you understand what they mean.

The clock, registers, and latches are common places to introduce bugs. Many FPGA and ASIC design flows use a conservative methodology of supplying a single clock to edge-triggered, asynchronously resettable registers with clock enables. While this will sacrifice performance and cost extra area, it reduces both the number of errors a designer can make and the expense of debugging these errors. The following guidelines are common to such a style.

1. Use only positive edge-triggered registers. Avoid `@(negedge clk)`, SR latches, and transparent latches.
2. Be certain not to inadvertently imply latches. Check synthesis reports for warnings such as

   (Synopsys) *Warning: Latch inferred in design '...' read with 'hdlin_check_no_latch'.*
   (Synplicity) *@W: ...Latch generated from always block for signal ...*
3. Provide an asynchronous reset to all of your registers with a common signal name.
4. Provide a common clock to all of your registers whenever possible. Avoid gated clocks, which may lead to extra clock skew and hold time failures. Use a clock enable instead.
5. If you get any "Bus Conflict" messages or `x`'s in your simulation, be sure to find their cause and fix the problem.

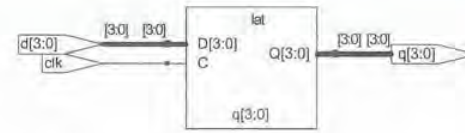
A.9.2 Incorrect Stimulus List

The following circuit was intended to be a transparent latch, but the `d` input was omitted from the stimulus list. When synthesized, it still produces a transparent latch, but with a warning. If the Verilog is simulated, `q` will change on the rising edge of `clk` but not on a change in `d` while `clk` is stable high. Thus, the circuit will simulate as if it were a flip-flop.

Inconsistencies between simulation and synthesis are a common reason for chips to fail, so correcting these warnings is vital.

(Synopsys) *Warning: Variable 'd' is being read*
in routine notquitealatch line 5 in file 'notquitealatch.v',
but does not occur in the timing control of the block which begins there. (HDL-180)
(Synplicity) *@W: notquitealatch.v(5): Incomplete sensitivity list - assuming completeness*
@W:" notquitealatch.v":5:12:5:15
@W: notquitealatch.v(6): Referenced variable d is not in sensitivity list
@W:"c: notquitealatch.v":6:20:6:21

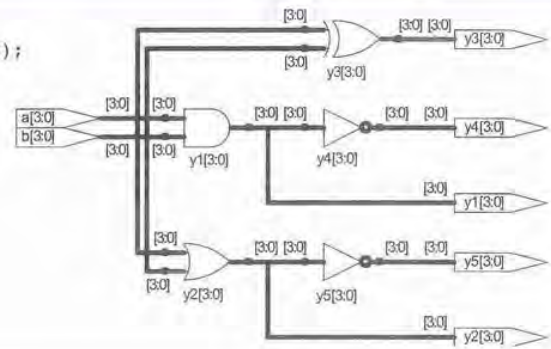
```
module notquitealatch(input             clk,
                      input      [3:0] d,
                      output reg [3:0] q);

   always @(clk) // left out 'or d'
      if (clk) q <= d;
endmodule
```



Similarly, the `b` input in the following combinational logic was omitted from the stimulus list of the `always` block. Synthesis tools do generate the intended gates, but give another warning.

(Synopsys) *Warning: Variable 'b' is being read*
in routine gates line 4 in file 'gates.v',
but does not occur in the timing control of the block which begins
there. (HDL-180)
(Synplicity) *@W: gates_bad.v(4): Incomplete sensitivity list - assuming completeness*
@W:" gates_bad.v":4:12:4:13
@W: gates_bad.v(6): Referenced variable b is not in sensitivity list
@W:"c: gates_bad.v":6:19:6:20

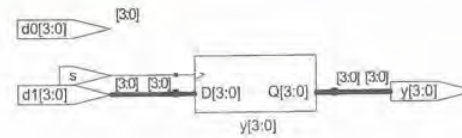
```
module gates(input      [3:0] a, b,
             output reg [3:0] y1, y2, y3, y4, y5);

   always @(a) // missing 'or b'
      begin
         y1 <= a & b; // AND
         y2 <= a | b; // OR
         y3 <= a ^ b; // XOR
         y4 <= ~(a & b); // NAND
         y5 <= ~(a | b); // NOR
      end
endmodule
```



The next example is supposed to model a multiplexer, but the author incorrectly wrote `@(posedge s)` rather than `@(s)`. Because `s` is by definition high immediately after its positive edge, this circuit actually behaves as a flip-flop that captures `d1` on the rising edge of `clk` and ignores `d0`. Some tools will synthesize the flip-flop, while others will just produce an error message.

(Synopysis) *Error: clock variable s is being used as data. (HDL-175)*
(Synplicity) *@W: badmux.v(1): Input d0 is unused @W:" badmux.v":1:31:1:33*

```
module badmux(input      [3:0] d0, d1,
              input            s,
              output reg [3:0] y);

   always @(posedge s)
      if (s) y <= d1;
      else   y <= d0;

endmodule
```



A.9.3 Missing `begin/end` Block

In the following example, two variables are supposed to be assigned in the `always` block. The `begin/end` block is missing. This is a syntax error.

(Synopsys) *Error: syntax error at or near token '[' (File: notquiteatwobitflop.v Line: 7) (VE-0)*
Error: Can't read 'verilog' file 'J:/Classes/E155/Fall2000/synopsys/flop2.v'. (UID-59)
(Synplicity) *@E: notquiteatwobitflop.v(7): Expecting : @E:" notquiteatwobitflop.v":7:9:7:9*

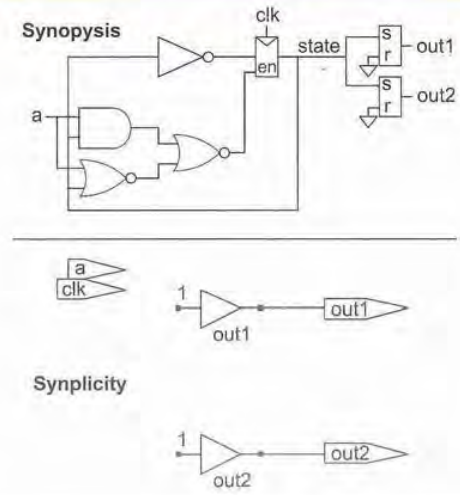
```
module notquiteatwobitflop(input           clk,
                           input      [1:0] d,
                           output reg [1:0] q);

   always @(posedge clk)
      q[1] = d[1];
      q[0] = d[0];
endmodule
```

A.9.4 Undefined Outputs

In the next example of a finite state machine, the user intended `out1` to be high when the state is 0 and `out2` to be high when the state is 1. However, the code never sets the outputs low. Synopsys produces a circuit with an SR latch and a transparent latch that can set the

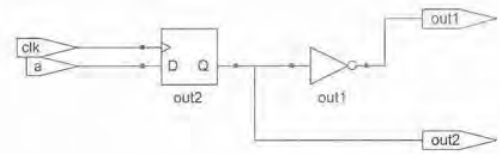
output high, but never resets the output low. Synplicity produces a circuit in which both outputs are hardwired to 1.

```
module FSMbad(input      clk,
              input      a,
              output reg out1, out2);

   reg  state;

   always @(posedge clk)
      if (state == 0) begin
         if (a) state <= 1;
      end else begin
         if (-a) state <= 0;
      end

   always @(*)
      if (state == 0) out1 <= 1;
      else            out2 <= 1;
      // neglect to set out1/out2 to 0
endmodule
```



A corrected version of the code produces the desired state machine.

```
module FSMgood(input      clk,
               input      a,
               output reg out1, out2);

   reg  state;

   always @(posedge clk)
      if (state == 0) begin
         if (a) state <= 1;
      end else begin
         if (-a) state <= 0;
      end

   always @(*)
      if (state == 0) begin
         out1 <= 1;
         out2 <= 0;
      end else begin
         out2 <= 1;
         out1 <= 0;
      end
endmodule
```



A.9.5 Incomplete Specification of Cases

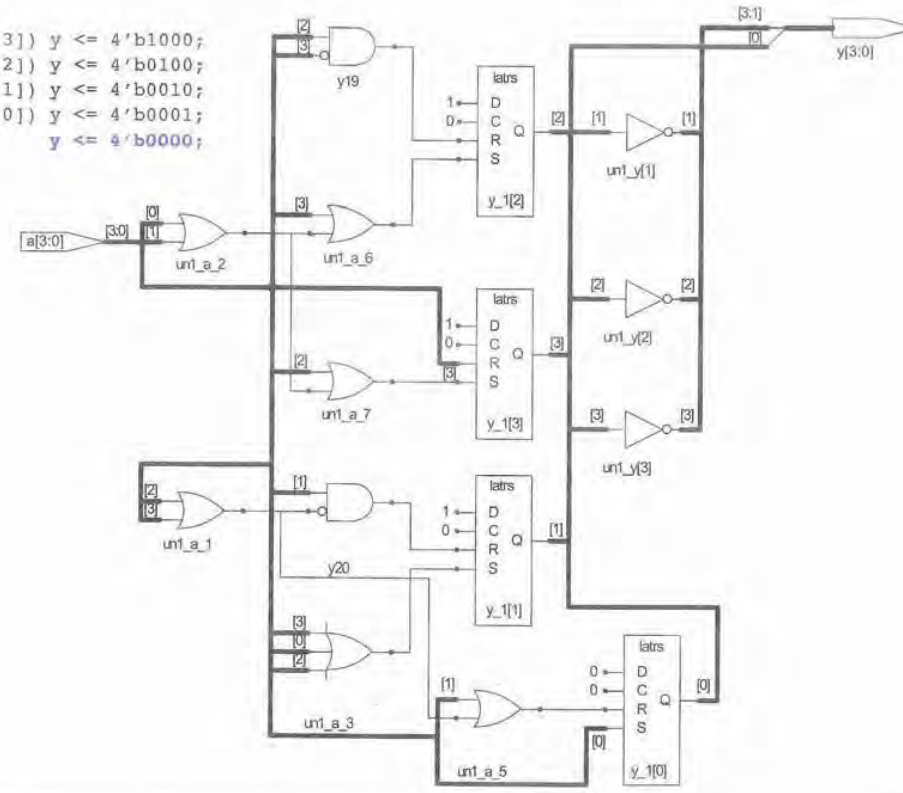
The next examples show an incomplete specification of input possibilities. The priority encoder fails to check for the possibility of no true inputs. It therefore incorrectly implies latches to hold the previous output when all four inputs are false. Synplicity produces a bizarre circuit with SR latches.

The synthesis tool will warn that a latch or memory device is implied. The astute designer will detect the problem by knowing that a priority encoder should be a combinational circuit and therefore have no memory devices.

(Synopsys) *Inferred memory devices in process in routine priority_always line 4 in file 'priority_always_bad.v'.*

(Synplicity) *@W: priority_always_bad.v(5): Latch generated from always block for signal y[3:0], probably caused by a missing assignment in an if or case stmt @W:"priority_always_bad.v":5:6:5:8*

```
module priority_always(input      [3:0] a,
                       output reg [3:0] y);

   always @(*)
      if      (a[3]) y <= 4'b1000;
      else if (a[2]) y <= 4'b0100;
      else if (a[1]) y <= 4'b0010;
      else if (a[0]) y <= 4'b0001;
      // else        y <= 4'b0000;
endmodule
```



The next example of a 7-segment display decoder shows the same type of problem in a case statement.

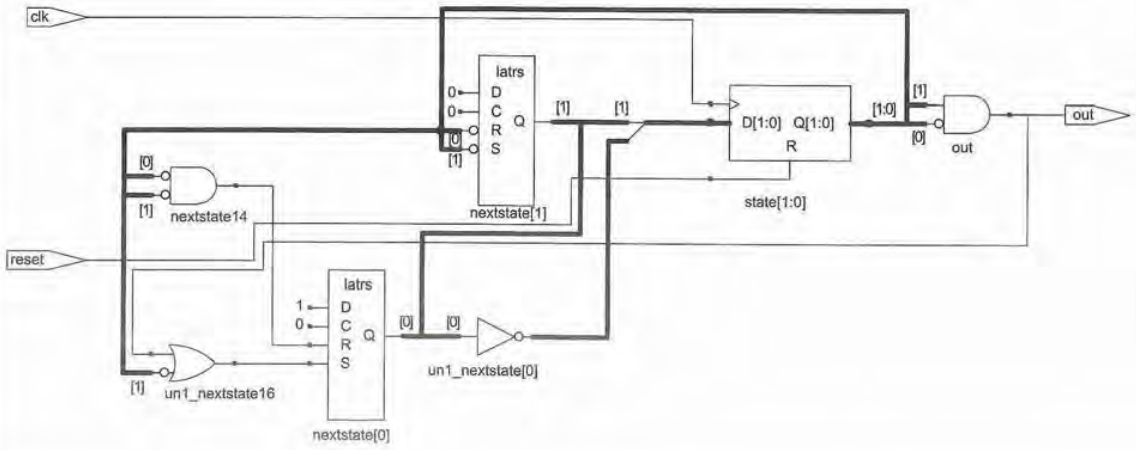
```
module seven_seg_display_decoder(input      [3:0] data,
                                 output reg [6:0] segments);

   always @(*)
      case (data)
         0: segments <= 7'b000_0000; // ZERO
         1: segments <= 7'b111_1110; // ONE
         2: segments <= 7'b011_0000; // TWO
         3: segments <= 7'b110_1101; // THREE
         4: segments <= 7'b011_0011; // FOUR
         5: segments <= 7'b101_1011; // FIVE
         6: segments <= 7'b101_1111; // SIX
         7: segments <= 7'b111_0000; // SEVEN
         8: segments <= 7'b111_1111; // EIGHT
         9: segments <= 7'b111_1011; // NINE
         // default: segments <= 7'b000_0000;
      endcase
endmodule
```

Similarly, it is a common mistake to forget the `default` in the next state or output logic of an FSM.

```
module divideby3FSM(input  clk,
                    input  reset,
                    output out);

   reg  [1:0] state, nextstate;

   parameter S0 = 2'b00;
   parameter S1 = 2'b01;
   parameter S2 = 2'b10;

   // State Register
   always @(posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;

   // Next State Logic
   always @(state)
      case (state)
         S0: nextstate <= S1;
         S1: nextstate <= S2;
         S2: nextstate <= S0;
         //default: nextstate <= S0;
      endcase
```

continued

```
    // Output Logic
    assign out = (state == S2);
endmodule
```

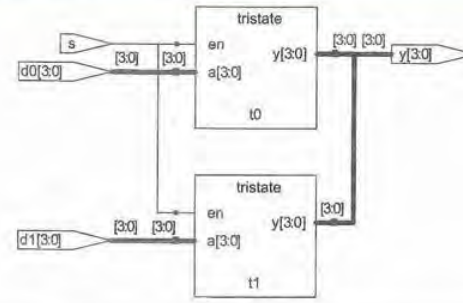


A.9.6 Shorted Outputs

Bad code can sometimes lead to shorted outputs of gates. For example, the tristate drivers in the following multiplexer should have mutually exclusive enable signals, but instead are both active simultaneously and produce a conflict when `d0` and `d1` are not equal.

Synthesis may not report any errors. However, during simulation, you will observe `x`'s rather than 0's or 1's when the bus is simultaneously being driven high and low. You may also get a "Bus Conflict" warning message.

```
module mux2(input  [3:0] d0, d1,
            input        s,
            output [3:0] y);

   tristate t0(d0, s, y); // wanted -s
   tristate t1(d1, s, y);
endmodule
```



Another cause of shorted outputs is when a `reg` is assigned in two different `always` blocks. For example, the following code tries to model a register with asynchronous reset and asynchronous set. The first `always` block models the reset and ordinary operation. The second `always` block attempts to incorporate the asynchronous set. Synthesis infers a separate piece of hardware for each `always` block, with a shorted output and may report an error.

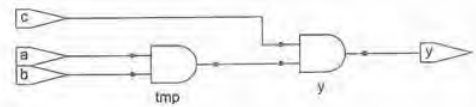
(Synopsys) *Error: the net '/ver1/q' has more than one driver*
(Synplicity) *@E: floprs_bad.v(5): Only one always block may assign a given variable q[3:0]*
@E:"floprs_bad.v":5:33:5:34

```
module floprsen(input            clk,
                input            reset,
                input            set,
                input      [3:0] d,
                output reg [3:0] q);

   // bad asynchronous set and reset

   always @(posedge clk, posedge reset)
      if (reset) q <= 0;
      else q <= d;

   always @(set)
      if (set) q <= 1;
endmodule
```

A.9.7 Incorrect Use of Nonblocking Assignments

Section A.4.6 recommended using nonblocking assignments in all `always` blocks. This can occasionally get you in trouble, as shown in the following 3-input AND gate.

```
module and3bad(a, b, c, y);
   input            a, b, c;
   output           y;

   reg              y;
   reg              tmp;

   always @(a, b, c)
   begin
      tmp <= a & b;
      y <= tmp & c;
   end
endmodule
```



In this example, suppose that initially, a = 0 and b = c = 1. Therefore, tmp initially is 0. When a rises, the always block is triggered. tmp is given the value 1, but y is in parallel given the value 0 because the new value of tmp has not yet been written. The code can synthesize correctly, but simulate incorrectly without warning. The problem can be avoided by using always @(*) to describe combinational logic.

A.10 Example: MIPS Processor

To illustrate a nontrivial Verilog design, this section lists the Verilog code and test bench for the MIPS processor subset discussed in Chapter 1. The example handles only the LB, SB, ADD, SUB, AND, OR, SLT, BEQ, and J instructions. It uses an 8-bit datapath and only eight registers. Because the instruction is 32 bits wide, it is loaded in four successive fetch cycles across an 8-bit path to external memory.

The test bench initializes a 512-byte memory with instructions and data from a text file. The code exercises each of the instructions. The mipstest.asm assembly language file and memfile.dat text file are shown below. The test bench runs until it observes a memory write. If the value 7 is written to address 5, the code probably executed correctly. If all goes well, the test bench should take 100 cycles (1000 ns) to run.

```
# mipstest.asm
# 9/16/03 David Harris David_Harris@hmc.edu
#
# Test MIPS instructions.  Assumes memory was
# initialized as:
# word 16: 3 - be careful of endianness
# word 17: 5
# word 18: 12

main:     #Assembly Code          effectMachine Code
          lb $2, 68($0)           # initialize $2 = 580020044
          lb $7, 64($0)           # initialize $7 = 380070040
          lb $3, 69($7)           # initialize $3 = 1280e30045
          or $4, $7, $2           # $4 <= 3 or 5 = 700e22025
          and $5, $3, $4          # $5 <= 12 and 7 = 400642824
          add $5, $5, $4          # $5 <= 4 + 7 = 1100a42820
          beq $5, $7, end         # shouldn't be taken10a70008
          slt $6, $3, $4          # $6 <= 12 < 7 = 00064302a
          beq $6, $0, around      # should be taken10c00001
          lb $5, 0($0)            # shouldn't happen80050000
around:   slt $6, $7, $2          # $6 <= 3 < 5 = 100e2302a
          add $7, $6, $5          # $7 <= 1 + 11 = 1200c53820
          sub $7, $7, $2          # $7 <= 12 - 5 = 700e23822
          j end                   # should be taken0800000f
          lb $7, 0($0)            # shouldn't happen80070000
end:      sb $7, 0($2)            # write adr 5 <= 7a0470000
```

```
memfile.dat
80020044
80070040
80e30045
00e22025
00642824
00a42820
10a70008
0064302a
10c00001
80050000
00e2302a
00c53820
00e23822
0800000f
80070000
a0470000
03000000
05000000
0c000000
```

```
//-------------------------------
// mips.v
// Max Yi (byyi@hmc.edu) and David_Harris@hmc.edu 12/9/03
// Model of subset of MIPS processor described in Ch 1
//-------------------------------

// top level design for testing
module top #(parameter WIDTH = 8, REGBITS = 3)();

   reg                 clk;
   reg                 reset;
   wire                memread, memwrite;
   wire    [WIDTH-1:0] adr, writedata;
   wire    [WIDTH-1:0] memdata;

   // instantiate devices to be tested
   mips #(WIDTH,REGBITS) dut(clk, reset, memdata, memread, memwrite, adr, writedata);

   // external memory for code and data
   exmemory #(WIDTH) exmem(clk, memwrite, adr, writedata, memdata);

   // initialize test
   initial
      begin
         reset <= 1; # 22; reset <= 0;
      end

   // generate clock to sequence tests
   always
      begin
         clk <= 1; # 5; clk <= 0; # 5;
      end

   always@(negedge clk)
      begin
         if(memwrite)
            if(adr == 5 & writedata == 7)
               $display("Simulation completely successful");
            else $display("Simulation failed");
      end
endmodule
```

```
// external memory accessed by MIPS
module exmemory #(parameter WIDTH = 8)
                 (input                  clk,
                  input                  memwrite,
                  input      [WIDTH-1:0] adr, writedata,
                  output reg [WIDTH-1:0] memdata);

   reg  [31:0] RAM [(1<<WIDTH-2)-1:0];
   wire [31:0] word;

   initial
      begin
         $readmemh("memfile.dat",RAM);
      end

   // read and write bytes from 32-bit word
   always @(posedge clk)
      if(memwrite)
         case (adr[1:0])
            2'b00: RAM[adr>>2][7:0] <= writedata;
            2'b01: RAM[adr>>2][15:8] <= writedata;
            2'b10: RAM[adr>>2][23:16] <= writedata;
            2'b11: RAM[adr>>2][31:24] <= writedata;
         endcase

   assign word = RAM[adr>>2];
   always @(*)
      case (adr[1:0])
         2'b00: memdata <= word[31:24];
         2'b01: memdata <= word[23:16];
         2'b10: memdata <= word[15:8];
         2'b11: memdata <= word[7:0];
      endcase
endmodule

// simplified MIPS processor
module mips #(parameter WIDTH = 8, REGBITS = 3)
            (input                  clk, reset,
             input  [WIDTH-1:0] memdata,
             output             memread, memwrite,
             output [WIDTH-1:0] adr, writedata);

   wire [31:0] instr;
   wire        zero, alusrca, memtoreg, iord, pcen, regwrite, regdst;
   wire [1:0]  aluop,pcsource,alusrcb;
   wire [3:0]  irwrite;
   wire [2:0]  alucont;

   controller  cont(clk, reset, instr[31:26], zero, memread, memwrite,
                    alusrca, memtoreg, iord, pcen, regwrite, regdst,
                    pcsource, alusrcb, aluop, irwrite);
   alucontrol  ac(aluop, instr[5:0], alucont);
   datapath    #(WIDTH, REGBITS)
               dp(clk, reset, memdata, alusrca, memtoreg, iord, pcen,
                  regwrite, regdst, pcsource, alusrcb, irwrite, alucont,
                  zero, instr, adr, writedata);
endmodule

module controller(input clk, reset,
                  input      [5:0] op,
                  input            zero,
                  output reg       memread, memwrite, alusrca, memtoreg, iord,
                  output           pcen,
                  output reg       regwrite, regdst,
                  output reg [1:0] pcsource, alusrcb, aluop,
                  output reg [3:0] irwrite);
```

```
parameter   FETCH1  = 4'b0001;
parameter   FETCH2  = 4'b0010;
parameter   FETCH3  = 4'b0011;
parameter   FETCH4  = 4'b0100;
parameter   DECODE  = 4'b0101;
parameter   MEMADR  = 4'b0110;
parameter   LBRD    = 4'b0111;
parameter   LBWR    = 4'b1000;
parameter   SBWR    = 4'b1001;
parameter   RTYPEEX = 4'b1010;
parameter   RTYPEWR = 4'b1011;
parameter   BEQEX   = 4'b1100;
parameter   JEX     = 4'b1101;

parameter   LB      = 6'b100000;
parameter   SB      = 6'b101000;
parameter   RTYPE   = 6'b0;
parameter   BEQ     = 6'b000100;
parameter   J       = 6'b000010;

reg [3:0] state, nextstate;
reg       pcwrite, pcwritecond;

// state register
always @(posedge clk)
   if(reset) state <= FETCH1;
   else state <= nextstate;

// next state logic
always @(*)
   begin
      case(state)
         FETCH1:  nextstate <= FETCH2;
         FETCH2:  nextstate <= FETCH3;
         FETCH3:  nextstate <= FETCH4;
         FETCH4:  nextstate <= DECODE;
         DECODE:  case(op)
                     LB:      nextstate <= MEMADR;
                     SB:      nextstate <= MEMADR;
                     RTYPE:   nextstate <= RTYPEEX;
                     BEQ:     nextstate <= BEQEX;
                     J:       nextstate <= JEX;
                     default: nextstate <= FETCH1; // should never happen
                  endcase
         MEMADR:  case(op)
                     LB:      nextstate <= LBRD;
                     SB:      nextstate <= SBWR;
                     default: nextstate <= FETCH1; // should never happen
                  endcase
         LBRD:    nextstate <= LBWR;
         LBWR:    nextstate <= FETCH1;
         SBWR:    nextstate <= FETCH1;
         RTYPEEX: nextstate <= RTYPEWR;
         RTYPEWR: nextstate <= FETCH1;
         BEQEX:   nextstate <= FETCH1;
         JEX:     nextstate <= FETCH1;
         default: nextstate <= FETCH1; // should never happen
      endcase
   end

always @(*)
   begin
   // set all outputs to zero, then conditionally assert just the appropriate ones
         irwrite <= 4'b0000;
         pcwrite <= 0; pcwritecond <= 0;
         regwrite <= 0; regdst <= 0;
         memread <= 0; memwrite <= 0;
         alusrca <= 0; alusrcb <= 2'b00; aluop <= 2'b00;
         pcsource <= 2'b00;
         iord <= 0; memtoreg <= 0;
```

```
case(state)
   FETCH1:
      begin
         memread <= 1;
         irwrite <= 4'b1000;
         alusrcb <= 2'b01;
         pcwrite <= 1;
      end
   FETCH2:
      begin
         memread <= 1;
         irwrite <= 4'b0100;
         alusrcb <= 2'b01;
         pcwrite <= 1;
      end
   FETCH3:
      begin
         memread <= 1;
         irwrite <= 4'b0010;
         alusrcb <= 2'b01;
         pcwrite <= 1;
      end
   FETCH4:
      begin
         memread <= 1;
         irwrite <= 4'b0001;
         alusrcb <= 2'b01;
         pcwrite <= 1;
      end
   DECODE: alusrcb <= 2'b11;
   MEMADR:
      begin
         alusrca <= 1;
         alusrcb <= 2'b10;
      end
   LBRD:
      begin
         memread <= 1;
         iord    <= 1;
      end
   LBWR:
      begin
         regwrite <= 1;
         memtoreg <= 1;
      end
   SBWR:
      begin
         memwrite <= 1;
         iord     <= 1;
      end
   RTYPEEX:
      begin
         alusrca <= 1;
         aluop   <= 2'b10;
      end
   RTYPEWR:
      begin
         regdst   <= 1;
         regwrite <= 1;
      end
   BEQEX:
      begin
         alusrca     <= 1;
         aluop       <= 2'b01;
         pcwritecond <= 1;
         pcsource    <= 2'b01;
      end
   JEX:
      begin
         pcwrite  <= 1;
         pcsource <= 2'b10;
      end
```

```
        endcase
    end
  assign pcen = pcwrite | (pcwritecond & zero); // program counter enable
endmodule

module alucontrol(input      [1:0] aluop,
                  input      [5:0] funct,
                  output reg [2:0] alucont);

  always @(*)
    case(aluop)
      2'b00: alucont <= 3'b010;  // add for lb/sb/addi
      2'b01: alucont <= 3'b110;  // sub (for beq)
      default: case(funct)       // R-Type instructions
                 6'b100000: alucont <= 3'b010; // add (for add)
                 6'b100010: alucont <= 3'b110; // subtract (for sub)
                 6'b100100: alucont <= 3'b000; // logical and (for and)
                 6'b100101: alucont <= 3'b001; // logical or (for or)
                 6'b101010: alucont <= 3'b111; // set on less (for slt)
                 default:   alucont <= 3'b101; // should never happen
               endcase
    endcase
endmodule

module datapath #(parameter WIDTH = 8, REGBITS = 3)
                 (input                  clk, reset,
                  input  [WIDTH-1:0] memdata,
                  input                  alusrca, memtoreg, iord, pcen, regwrite, regdst,
                  input  [1:0]           pcsource, alusrcb,
                  input  [3:0]           irwrite,
                  input  [2:0]           alucont,
                  output                 zero,
                  output [31:0]          instr,
                  output [WIDTH-1:0] adr, writedata);

  // the size of the parameters must be changed to match the WIDTH parameter
  parameter CONST_ZERO = 8'b0;
  parameter CONST_ONE =  8'b1;

  wire [REGBITS-1:0] ra1, ra2, wa;
  wire [WIDTH-1:0]   pc, nextpc, md, rd1, rd2, wd, a, src1, src2, aluresult,
                     aluout, constx4;

  // shift left constant field by 2
  assign constx4 = {instr[WIDTH-3:0],2'b00};

  // register file address fields
  assign ra1 = instr[REGBITS+20:21];
  assign ra2 = instr[REGBITS+15:16];
  mux2       #(REGBITS) regmux(instr[REGBITS+15:16], instr[REGBITS+10:11], regdst, wa);

  // independent of bit width, load instruction into four 8-bit registers over four cycles
  flopen     #(8)       ir0(clk, irwrite[0], memdata[7:0], instr[7:0]);
  flopen     #(8)       ir1(clk, irwrite[1], memdata[7:0], instr[15:8]);
  flopen     #(8)       ir2(clk, irwrite[2], memdata[7:0], instr[23:16]);
  flopen     #(8)       ir3(clk, irwrite[3], memdata[7:0], instr[31:24]);

  // datapath
  flopenr    #(WIDTH)  pcreg(clk, reset, pcen, nextpc, pc);
  flop       #(WIDTH)  mdr(clk, memdata, md);
  flop       #(WIDTH)  areg(clk, rd1, a);
  flop       #(WIDTH)  wrd(clk, rd2, writedata);
  flop       #(WIDTH)  res(clk, aluresult, aluout);
  mux2       #(WIDTH)  adrmux(pc, aluout, iord, adr);
  mux2       #(WIDTH)  src1mux(pc, a, alusrca, src1);
  mux4       #(WIDTH)  src2mux(writedata, CONST_ONE, instr[WIDTH-1:0],
                               constx4, alusrcb, src2);
  mux4       #(WIDTH)  pcmux(aluresult, aluout, constx4, CONST_ZERO, pcsource, nextpc);
  mux2       #(WIDTH)  wdmux(aluout, md, memtoreg, wd);
  regfile    #(WIDTH,REGBITS) rf(clk, regwrite, ra1, ra2, wa, wd, rd1, rd2);
```

```
   alu        #(WIDTH) alunit(src1, src2, alucont, aluresult);
   zerodetect #(WIDTH) zd(aluresult, zero);
endmodule

module alu #(parameter WIDTH = 8)
            (input      [WIDTH-1:0] a, b,
             input      [2:0]       alucont,
             output reg [WIDTH-1:0] result);

   wire     [WIDTH-1:0] b2, sum, slt;

   assign b2 = alucont[2] ? ~b:b;
   assign sum = a + b2 + alucont[2];
   // slt should be 1 if most significant bit of sum is 1
   assign slt = sum[WIDTH-1];

   always@(*)
      case(alucont[1:0])
         2'b00: result <= a & b;
         2'b01: result <= a | b;
         2'b10: result <= sum;
         2'b11: result <= slt;
      endcase
endmodule

module regfile #(parameter WIDTH = 8, REGBITS = 3)
                (input                 clk,
                 input                 regwrite,
                 input  [REGBITS-1:0] ra1, ra2, wa,
                 input  [WIDTH-1:0]   wd,
                 output [WIDTH-1:0]   rd1, rd2);

   reg  [WIDTH-1:0] RAM [(1<<REGBITS)-1:0];

   // three ported register file
   // read two ports combinationally
   // write third port on rising edge of clock
   // register 0 hardwired to 0
   always @(posedge clk)
      if (regwrite) RAM[wa] <= wd;

   assign rd1 = ra1 ? RAM[ra1] : 0;
   assign rd2 = ra2 ? RAM[ra2] : 0;
endmodule

module zerodetect #(parameter WIDTH = 8)
                   (input [WIDTH-1:0] a,
                    output            y);

   assign y = (a==0);
endmodule

module flop #(parameter WIDTH = 8)
             (input                 clk,
              input      [WIDTH-1:0] d,
              output reg [WIDTH-1:0] q);

   always @(posedge clk)
      q <= d;
endmodule

module flopen #(parameter WIDTH = 8)
               (input                 clk, en,
                input      [WIDTH-1:0] d,
                output reg [WIDTH-1:0] q);

   always @(posedge clk)
      if (en) q <= d;
endmodule
```

```
module flopenr #(parameter WIDTH = 8)
                (input                  clk, reset, en,
                 input      [WIDTH-1:0] d,
                 output reg [WIDTH-1:0] q);

   always @(posedge clk)
      if      (reset) q <= 0;
      else if (en)    q <= d;
endmodule

module mux2 #(parameter WIDTH = 8)
             (input  [WIDTH-1:0] d0, d1,
              input              s,
              output [WIDTH-1:0] y);

   assign y = s ? d1 : d0;
endmodule

module mux4 #(parameter WIDTH = 8)
             (input      [WIDTH-1:0] d0, d1, d2, d3,
              input      [1:0]       s,
              output reg [WIDTH-1:0] y);

   always @(*)
      case(s)
         2'b00: y <= d0;
         2'b01: y <= d1;
         2'b10: y <= d2;
         2'b11: y <= d3;
      endcase
endmodule
```

Appendix

VHDL B

B.1 Introduction

This appendix gives a quick introduction to VHDL. VHDL is an acronym for the VHSIC Hardware Description Language. VHSIC is in turn an acronym for the US Department of Defense Very High Speed Integrated Circuits program. [Ashenden01] offers a definitive and readable treatment of the language.

VHDL was originally developed in 1981 by the Department of Defense as a language to describe the structure and function of hardware. The IEEE standardized it in 1987, and a revised version was adopted in 1993, updated in 2000, and updated again in 2002 [IEEE1076-02]. The language was first envisioned for documentation, but quickly was adopted for simulation and synthesis. Compared to Verilog, VHDL is more verbose and cumbersome, as you might expect of a language developed by committee. However, VHDL has some features that are convenient for large team design projects, and government contractors and telecommunications companies use it extensively. Religious wars have raged over which HDL is superior, but both are used too widely for CAD vendors not to support them.

As mentioned in Section 1.8.4, there are two general styles of description: *behavioral* and *structural*. Structural VHDL describes how a module is composed of simpler modules or basic primitives such as gates or transistors. Behavioral VHDL describes how the outputs are computed as functions of the inputs. There are two general types of statements used in behavioral VHDL. *Concurrent signal assignments* imply combinational logic. *Processes* can imply combinational logic or sequential logic, depending how they are used. It is good practice to partition your design into combinational and sequential components and then write VHDL in such a way that you get what you want. If you don't know whether a block of logic is combinational or sequential, you are likely to get the wrong thing.

This appendix focuses on a synthesizable subset of the VHDL language. The language also contains extensive capabilities for system modeling that are beyond the scope of this tutorial.

B.2 Behavioral Modeling with Concurrent Signal Assignments

A 32-bit adder is a complex design at the schematic level of representation. It can be constructed from 32 full adder cells, each of which in turn requires about six 2-input gates. VHDL provides a more compact description:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity adder is
    port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
         y:    out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture synth of adder is
begin
    y <= a + b;
end;
```

This example has three parts: the *library use clauses*, the *entity declaration*, and the *architecture body*. The library part will be discussed in Section B.3.7. The entity defines the inputs and outputs of the adder *block*, which in this case are 32-bit busses. The architecture describes what the block does. In this case, the output `y` is computed as the sum of `a` and `b`. `y <= a + b` is called a *concurrent signal assignment* statement because multiple statements could happen in parallel just as multiple logic gates can operate concurrently.

B.2.1 Bitwise Operators

VHDL has a number of *bitwise* operators that act on busses. For example, the following module describes four inverters.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity inv is
    port(a: in  STD_LOGIC_VECTOR(3 downto 0);
         y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of inv is
begin
    y <= not a;
end;
```

Similar bitwise operations are available for the other basic logic functions:

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity gates is
    port(a, b:               in  STD_LOGIC_VECTOR(3 downto 0);
         y1, y2, y3, y4, y5: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of gates is
begin
    -- Five different two-input logic gates acting on 4 bit busses
    y1 <= a and b;
    y2 <= a or b;
    y3 <= a xor b;
    y4 <= a nand b;
    y5 <= a nor b;
end;
```

B.2.2 Comments and White Space

The previous examples showed a comment. Comments begin with -- and continue to the end of the line. Comments spanning multiple lines use -- at the beginning of each line.

VHDL is not picky about the use of white space. Nevertheless, proper indenting and spacing is helpful to make nontrivial designs readable. VHDL is not case-sensitive, but a consistent use of upper and lower case also makes the code readable and more portable to other tools that are case-sensitive.

B.2.3 Other Operators

VHDL defines a number of operators available in concurrent assignment statements including:

Multiplying Operators:	`*, /, mod, rem`
Adding Operators:	`+, -`
Relational Operators:	`=, /=, <, <=, >, >=`
Logical Operators:	`not, and, or, nand, nor, xor, xnor`

= and /= (equality / inequality) on N-bit inputs require N 2-input XNORs to determine equality of each bit and an N-input AND or NAND to combine all the bits. Addition, subtraction, and comparison all require an adder, which is expensive in hardware. Multipliers are even more costly. Do not use these statements without contemplating the number of gates you are generating. Moreover, the implementations are not always particularly efficient for your problem. Division, `mod` (modulo), and `rem` (remainder) are only supported when the right operand is a power of two.

Some synthesis tools ship with optimized libraries for special functions like adders and multipliers. For example, the Synopsys DesignWare libraries produce reasonably good multipliers. If you do not have a license for the libraries, you'll probably be disappointed with the speed and gate count of a multiplier your synthesis tool produces from when it sees `*`.

B.2.4 Conditional Signal Assignment Statements

Conditional signal assignments perform different operations depending on certain conditions. For example, a 2:1 multiplexer can use conditional signal assignment to select one of two 4-bit inputs.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
    port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
         s:      in  STD_LOGIC;
         y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of mux2 is
begin
    y <= d0 when s = '0' else d1;
end;
```

A 4:1 multiplexer can select one of four inputs using multiple `else` clauses in the conditional signal assignment.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux4 is
    port(d0, d1, d2, d3: in  STD_LOGIC_VECTOR(3 downto 0);
         s:              in  STD_LOGIC_VECTOR(1 downto 0);
         y:              out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of mux4 is
begin
    y <= d0 when s = "00" else
         d1 when s = "01" else
         d2 when s = "10" else
         d3;
end;
```

B.2.5 Selected Signal Assignment Statements

Selected signal assignment statements provide a shorthand when selecting from one of several possibilities. They are analogous to using a `case` statement in place of multiple `if/`

else statements, as will be described in Section B.4.4. The 4:1 multiplexer could have been written with a selected signal assignment as:

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux4 is
    port(d0, d1, d2, d3: in  STD_LOGIC_VECTOR(3 downto 0);
         s:              in  STD_LOGIC_VECTOR(1 downto 0);
         y:              out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth2 of mux4 is
begin
    with s select y <=
        d0 when "00",
        d1 when "01",
        d2 when "10",
        d3 when others;
end;
```

B.3 Basic Constructs

B.3.1 Blocks, Entities, and Architectures

A cell or module in VHDL is called a *block* and is the basic unit of design. It has inputs and outputs specified in an *entity declaration* using the *port* statement. An *architecture body* defines what the block does. VHDL separates the architecture body from the entity declaration to allow multiple bodies for a single block. The syntax of the architecture body is

```
architecture <name> of <block> is
begin
   -- body goes here
end;
```

For example, we have seen body for the adder block with an architecture named synth. It uses a concurrent signal assignment in a behavioral description of the block. The name of the architecture carries no special meaning to VHDL, but helps the human reader understand that the code is intended to be synthesizable (rather than just for simulation). As the design is refined, you can imagine creating a structural description such as 32 full adders connected in a ripple-carry fashion. This architecture might be named struct. While VHDL permits selecting among multiple architectures for a single entity, the syntax is complicated and the simplest approach is to only provide one architecture for each entity.

A side effect of separating architectures from entities is that simple blocks involve substantially more typing in VHDL than in Verilog.

B.3.2 Internal Signals

Often it is convenient to break a complex calculation into intermediate variables. For example, in a full adder, we sometimes define the propagate signal as the XOR of the two inputs a and b. The sum from the adder is the XOR of the propagate signal and the carry-in. We can name the propagate signal using a `signal` statement, in much the same way we use local variables in a programming language.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fulladder is
    port(a, b, cin:  in STD_LOGIC;
         s, cout:    out STD_LOGIC);
end;

architecture synth of fulladder is
    signal prop: STD_LOGIC;
begin
    prop <= a xor b;
    s <= prop xor cin;
    cout <= (a and b) or (cin and (a or b));
end;
```

B.3.3 Precedence

Notice that we fully parenthesized the `cout` computation. This is necessary in VHDL because the logical operators all have equal precedence. If we had written

```
cout = a and b or cin and (a or b);
```

it could have been interpreted as either

```
cout = ((a and b) or cin) and (a or b));
out = (a and b) or (cin and (a or b)));
```

These are definitely not the same function, so VHDL demands parentheses.

Table B.1 lists operator precedence. All the operators on a particular row have equal precedence and have higher precedence than the row below. Thus, multiplication takes place before addition, as you would expect in most programming languages or in mathematics. On the other hand, AND does not take place before OR, unlike what you would expect in a conventional Boolean equation. Parentheses are used to group expressions when necessary.

Table B.1 Operator Precedence

not	Highest
*, /, mod, rem	
+, -, &	
=, /=, <, <=, >, >=	
=, = =, !=	
and, or, nand, nor, xor	Lowest

B.3.4 Hierarchy

Nontrivial designs are developed in a hierarchical form, in which complex blocks are composed of simpler blocks. For example, a 4-input multiplexer can be constructed from three 2-input multiplexers that were defined earlier.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux4 is
    port(d0, d1, d2, d3: in  STD_LOGIC_VECTOR(3 downto 0);
         s:              in  STD_LOGIC_VECTOR(1 downto 0);
         y:              out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture struct of mux4 is
    component mux2 port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
                        s:      in  STD_LOGIC;
                        y:      out STD_LOGIC_VECTOR(3 downto 0));
    end component;
    signal low, high: STD_LOGIC_VECTOR(3 downto 0);
begin
    lowmux:   mux2 port map(d0, d1, s(0), low);
    highmux:  mux2 port map(d2, d3, s(0), high);
    finalmux: mux2 port map(low, high, s(1), y);
end;
```

This is an example of the structural coding style because the mux4 is built from simpler blocks. It is good practice to avoid (or at least minimize) mixing structural and behavioral descriptions within a single module. Generally, simple modules are described behaviorally and larger modules are composed structurally from these building blocks.

The architecture must first declare the mux2 ports using the *component declaration* statement. This allows VHDL tools to make sure that the component you wish to use has the same ports as the component that was declared somewhere else in an entity statement, preventing errors caused by changing the entity but not the use. However, it makes VHDL code rather cumbersome.

B.3.5 Bit Swizzling

Often it is necessary to work on parts of a bus or to concatenate (join together) signals to construct busses. The mux4 example showed using the least significant bit `s(0)` of a 2-bit select signal for the low and high muxes and the most significant bit `s(1)` for the final mux. Use ranges to select subsets of a bus. For example, an 8-bit wide 2-input mux can be constructed from two 4-bit wide 2-input muxes:

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2_8 is
    port(d0, d1: in  STD_LOGIC_VECTOR(7 downto 0);
         s:      in  STD_LOGIC;
         y:      out STD_LOGIC_VECTOR(7 downto 0));
end;
```

```
architecture struct of mux2_8 is
    component mux2 port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
                        s:      in  STD_LOGIC;
                        y:      out STD_LOGIC_VECTOR(3 downto 0));
    end component;
begin

    lsbmux:  mux2 port map(d0(3 downto 0), d1(3 downto 0),
                           s, y(3 downto 0));
    msbhmux: mux2 port map(d0(7 downto 4), d1(7 downto 4),
                           s, y(7 downto 4));
end;
```

The & operator is used to concatenate busses. For example, the following code multiplies a number by four by shifting it left two positions.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity shift2 is
    port(a: in  STD_LOGIC_VECTOR(7 downto 0);
         y: out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of shift2 is
begin
    y <= a(5 downto 0) & "00";
end;
```

Variable left and right shifts can also be done with the SHL and SHR functions:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity shifter is
    port(a:   in  STD_LOGIC_VECTOR(7 downto 0);
         amt: in  STD_LOGIC_VECTOR(2 downto 0);
         y:   out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of shifter is
begin
    y <= SHL(a, amt);
end;
```

B.3.6 Types

VHDL uses a strict data typing system that can be clumsy at times. In logic design, our signals are typically binary digits or binary words. In addition to '0' and '1,' it is convenient to have values such as 'z' (high impedance, representing a tristate buffer with a floating output as described in Section B.3.8), 'u' (uninitialized, representing a flip-flop at startup

before reset where the value might be high or low), and 'x' (unknown, representing a node with contention that is driven high by one gate and low by another or the output of a gate whose inputs are 'z')[1]. The STD_LOGIC and STD_LOGIC_VECTOR types represent single- and multiple-bit binary values. Table B.2 shows a truth table for an AND gate using STD_LOGIC types. Notice that the operators are optimistic in that they can sometimes determine the result despite some inputs being unknown. For example '0' and 'z' returns '0' because the output of an AND gate is always '0' if either input is '0.'

Table B.2 AND gate truth table

		A				
		0	1	z	x	u
B	0	0	0	0	0	0
	1	0	1	x	x	u
	z	0	x	x	x	u
	x	0	x	x	x	u
	u	0	u	u	u	u

Despite its fundamental importance, the STD_LOGIC type is not built into VHDL. Instead, it is part of the IEEE.STD_LOGIC_1164 library defined by the IEEE 1164 standard [IEEE1164-93]. Thus, every file must include the library statements we have seen in previous examples. VHDL does have built-in BIT and BIT_VECTOR types that define only '0' and '1,' but we will avoid these types in this tutorial because STD_LOGIC is better suited for logic simulation with tristates, uninitialized nodes, and contention.

In simulation, it may be necessary to define what to do when an input is 'x' or 'u.' For example, the select signal to a multiplexer might be an illegal or undefined value. The mux4 examples in Sections B.2.4 and B.2.5 default to selecting d3 in such an event.

VHDL also has a *Boolean* type with two values: true and false. Boolean values are created by comparisons (like s = '0') and used in conditional statements such as if or when. Boolean and STD_LOGIC values are not interchangeable despite the temptation to think true means '1' and false means '0.' Thus, the following statements are illegal:

```
y <= d1 when s else d0;
q <= (state = S2);
```

Instead we must write

```
y <= d1 when s = '1' else d0;
q <= '1' when state = S2 else '0';
```

[1]Technically, STD_LOGIC also has values of W, L, H, and—for weak unknown, low, and high values and for don't cares, but we will not need them in this tutorial.

While we will not declare any signals to be Boolean, they are automatically implied by comparisons and used by conditional statements.

Similarly, VHDL has an INTEGER type representing a 2's complement integer of at least 32 bits (spanning values -2^{31} ... $2^{31}-1$). Integer values are used as array indices. For example, in the statement

```
lowmux:   mux2 port map(d0, d1, s(0), low);
```

0 is an integer serving as an index to choose one bit of the s signal. We cannot directly index an array with a STD_LOGIC or STD_LOGIC_VECTOR signal. Instead, we must convert the signal to an integer. This is demonstrated in the following 8:1 multiplexer that selects one bit from a vector using a 3-bit index. The conv_integer function is defined in STD_LOGIC_UNSIGNED and performs the conversion from STD_LOGIC_VECTOR to integer for positive (unsigned) values.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity mux8 is
   port(d: in  STD_LOGIC_VECTOR(7 downto 0);
        s: in  STD_LOGIC_VECTOR(2 downto 0);
        y: out STD_LOGIC);
end;

architecture synth of mux8 is
begin
    y <= d(conv_integer(s));
end;
```

VHDL also permits enumeration types. For example, the divide-by-3 finite state machine described in Section B.5 uses three states. We can give the states names using the enumeration type rather than referring to them by binary values.

```
type statetype is (S0, S1, S2);
signal state, nextstate: statetype;
```

B.3.7 Library and Use Clauses

VHDL uses a number of standard libraries that extend the built-in capabilities of the language. As we have seen, the IEEE.STD_LOGIC_1164 library defines the essential STD_LOGIC and STD_LOGIC_VECTOR types. Unfortunately, it does not define basic operations such as addition, comparison, shifts, or conversion to integer for STD_LOGIC_VECTOR data.

Synopsys has developed another freely available library to perform these functions for unsigned numbers. Many other CAD vendors have adopted it for compatibility. The library is typically called STD_LOGIC_UNSIGNED. An equivalent library called STD_LOGIC_SIGNED handles STD_LOGIC_VECTORS representing signed 2's complement

numbers. Right-shifts fill the most significant bit with 0's for unsigned numbers and sign-extend for signed numbers.

The syntax to load a library and use all of its functions is:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
```

B.3.8 Tristates

It is possible to leave a bus floating rather than drive it to '0' or '1.' This floating value is called 'z' in VHDL. For example, a tristate buffer produces a floating output when the enable is '0.'

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity tristate is
    port(a:  in  STD_LOGIC_VECTOR(3 downto 0);
         en: in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of tristate is
begin
    y <= "ZZZZ" when en = '0' else a;
end;
```

Multiple tristates driving different nonfloating values onto a bus causes contention, displayed as 'x' for STD_LOGIC signals. Floating inputs to gates also cause undefined outputs, displayed as 'x.' Thus, a bus driven by multiple tristates should have one and only one driver active at any given time.

We could define a multiplexer using two tristates so that the output is always driven by exactly one tristate. This guarantees that there are no floating nodes.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
    port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
         s:      in  STD_LOGIC;
         y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture struct of mux2 is
    component tristate port(a:  in  STD_LOGIC_VECTOR(3 downto 0);
                            en: in  STD_LOGIC;
                            y:  out STD_LOGIC_VECTOR(3 downto 0));
    end component;
    signal sbar: STD_LOGIC;
begin
    sbar <= not s;
    t0: tristate port map(d0, sbar, y);
    t1: tristate port map(d1, s, y);
end;
```

B.3.9 Delays

The delay of a statement can be specified in arbitrary units. For example, the following code defines an inverter with a 100 ps propagation delay. Delays have no impact on synthesis, but can be helpful while debugging simulation waveforms because they make cause and effect more apparent.

```
y <= not a after 100 ps;
```

B.4 Behavioral Modeling with Process Statements

Concurrent signal assignments are reevaluated every time any term on the right side changes. Therefore, they must describe combinational logic. `Process` statements are reevaluated only when signals in the header (called a *sensitivity list*) change. Depending on the form, `process` statements can imply either sequential or combinational circuits.

B.4.1 Flip-flops

Positive edge-triggered flip-flops are described with a process controlled by `clk`:

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flop is
    port(clk: in  STD_LOGIC;
         d:   in  STD_LOGIC_VECTOR(3 downto 0);
         q:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of flop is
begin
    process(clk) begin
       if clk'event and clk = '1' then -- or use "if RISING_EDGE(clk) then"
           q <= d;
        end if;
    end process;
end;
```

VHDL has several idioms for specifying positive edge-triggered flip-flops. In this example, the flip-flop copies `d` to `q` when an event takes place on `clk` and `clk` has a value of '1.' An event is a change in a signal value, i.e., a rising or falling transition. The `RISING_EDGE(clk)` syntax is also acceptable.

At startup, the `q` output is initialized to 'u' by the simulator before the first clock edge arrives. Generally, it is good practice to use resettable registers so that on power-up you

can put your system in a known state. The reset can be either asynchronous or synchronous. Asynchronous resets occur immediately. Synchronous resets only change the output on the rising edge of the clock.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flopr is
    port(clk, reset: in  STD_LOGIC;
         d:          in  STD_LOGIC_VECTOR(3 downto 0);
         q:          out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synchronous of flopr is
begin
    process(clk) begin
        if clk'event and clk = '1' then
            if reset = '1' then
                q <= "0000";
            else q <= d;
            end if;
        end if;
    end process;
end;

architecture asynchronous of flopr is
begin
    process(clk, reset) begin
        if reset = '1' then
            q <= "0000";
        elsif clk'event and clk = '1' then
            q <= d;
        end if;
    end process;
end;
```

Note that the asynchronously resettable flop evaluates the `process` statement when either `clk` or `reset` change so that it immediately responds to `reset`. The synchronously reset flop is not sensitized to reset, so it waits for the next clock edge before clearing the output.

You can also consider registers with enables that only respond to the clock when the enable is true. The following register retains its old value if both `reset` and `en` are false.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flopenr is
    port(clk, reset, en: in  STD_LOGIC;
         d:              in  STD_LOGIC_VECTOR(3 downto 0);
         q:              out STD_LOGIC_VECTOR(3 downto 0));
end;
```

```
architecture asynchronous of flopenr is -- asynchronous reset
begin
    process(clk, reset) begin
        if reset = '1' then
            q <= "0000";
        elsif clk'event and clk = '1' then
            if en = '1' then
                q <= d;
            end if;
        end if;
    end process;
end;
```

B.4.2 Latches

Transparent latches are also modeled with process statements. When the clock is high, the latch is *transparent* and the data input flows to the output. When the clock is low, the latch goes *opaque* and the output remains constant.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity latch is
    port(clk: in  STD_LOGIC;
         d:   in  STD_LOGIC_VECTOR(3 downto 0);
         q:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of latch is
begin
    process(clk, d) begin
        if clk = '1' then q <= d;
        end if;
    end process;
end;
```

The latch evaluates the `process` statement any time either `clk` or `d` change. If `clk` is high, the output gets the input.

B.4.3 Counters

Consider two ways of describing a 4-bit counter with asynchronous reset. The first scheme (behavioral) implies a sequential circuit containing both the 4-bit register and an adder. The second scheme (structural) explicitly declares modules for the register and adder. Either scheme is good for a simple circuit such as a counter. As you develop more complex finite state machines, it is a good idea to separate the next state logic from the registers in your code.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity counter is
    port(clk, reset: in     STD_LOGIC;
         q:          buffer STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of counter is
begin
    process(clk) begin
        if clk'event and clk = '1' then
            if reset = '1' then q <= "0000";
            else q <= q + "0001";
            end if;
        end if;
    end process;
end;

architecture struct of counter is
    component flopr port(clk, reset: in  STD_LOGIC;
                         d:           in  STD_LOGIC_VECTOR(3 downto 0);
                         q:           out STD_LOGIC_VECTOR(3 downto 0));
    end component;
    component adder port(a, b:  in  STD_LOGIC_VECTOR(3 downto 0);
                         y:     out STD_LOGIC_VECTOR(3 downto 0));
    end component;
    signal nextq: STD_LOGIC_VECTOR(3 downto 0);
begin
    qflop: flopr port map(clk, reset, nextq, q);
    inc:   adder_4 port map(q, "0001", nextq);
end;
```

Note that `q` is defined as `buffer` instead of `out`. `Buffer` ports are required when a signal is used as both input and output within a block, as in the case of `q <= q + "0001"`. Also, `adder_4` is identical to the adder block defined earlier except that it is 4 bits wide.

B.4.4 Combinational Logic

`Process` statements imply sequential logic when some of the inputs do not appear in the `@` stimulus list or might not cause the output to change. For example, in the flop module, `d` is not in the `@` list, so the flop does not immediately respond to changes of `d`. In the latch, `d` is in the `@` list, but changes in `d` are ignored unless `clk` is high so the latch is also sequential. Processes can also be used to imply combinational logic if they are written in such a way that the process is reevaluated every time there are changes in any of the inputs. The following architecture shows how to define a bank of inverters with a `process`. Only the architecture is listed; the rest of the code is the same as the other inverter example. The `begin` and `end process` statements are required even though the process only contains one assignment.

```
architecture proc of inv is
begin
    process(a) begin
        y <= not a;
    end process;
end;
```

Similarly, the next example defines five banks of different kinds of gates.

```
architecture proc of gates is
begin
    process(a, b) begin
        y1 <= a and b;
        y2 <= a or b;
        y3 <= a xor b;
        y4 <= a nand b;
        y5 <= a nor b;
end process;
end;
```

Processes can also contain `if` statements. The following example describes a priority encoder that determines the most significant input bit that is asserted.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity priority is
    port(a: in  STD_LOGIC_VECTOR(3 downto 0);
         y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of priority is
begin
    process(a) begin
        if    a(3) = '1' then y <= "1000";
        elsif a(2) = '1' then y <= "0100";
        elsif a(1) = '1' then y <= "0010";
        elsif a(0) = '1' then y <= "0001";
        else                  y <= "0000";
        end if;
    end process;
end;
```

Processes can also contain `case` statements. For example, a 3:8 decoder could be written using a `case` statement. This is easier to read than a description of the same decoder using Boolean equations.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity decoder is
    port(a: in  STD_LOGIC_VECTOR(2 downto 0);
         y: out STD_LOGIC_VECTOR(7 downto 0));
end;
```

```
architecture proc_case of decoder is
begin
    process(a) begin
        case a is
           when "000" =>  y <= "00000001";
           when "001" =>  y <= "00000010";
           when "010" =>  y <= "00000100";
           when "011" =>  y <= "00001000";
           when "100" =>  y <= "00010000";
           when "101" =>  y <= "00100000";
           when "110" =>  y <= "01000000";
           when others => y <= "10000000";
        end case;
    end process;
end;

architecture boolean of decoder is
begin
    y(0) <= not a(2) and not a(1) and not a(0);
    y(1) <= not a(2) and not a(1) and a(0);
    y(2) <= not a(2) and a(1) and not a(0);
    y(3) <= not a(2) and a(1) and a(0);
    y(4) <= a(2) and not a(1) and not a(0);
    y(5) <= a(2) and not a(1) and a(0);
    y(6) <= a(2) and a(1) and not a(0);
    y(7) <= a(2) and a(1) and a(0);
end;
```

Another even better application of `case` is the logic for a 7-segment display decoder. The 7-segment display is shown in Figure B.1. The decoder takes a 4-bit number and displays its decimal value on the segments. For example, the number 0111 = 7 should turn on segments `a`, `b`, and `c`. The equivalent logic with concurrent signal assignments describing the detailed logic for each bit would be tedious. This more abstract approach is faster to write, clearer to read, and can be automatically synthesized down to an efficient logic implementation.

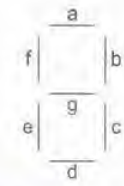


FIG B.1 7-segment display mapping

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity seven_seg_decoder is
    port(data:     in  STD_LOGIC_VECTOR(3 downto 0);
         segments: out STD_LOGIC_VECTOR(6 downto 0));
end;

architecture synth of seven_seg_decoder is
--                                    Segment #'s: abcdefg
    constant BLANK: STD_LOGIC_VECTOR(6 downto 0) := "0000000";
    constant ZERO:  STD_LOGIC_VECTOR(6 downto 0) := "1111110";
    constant ONE:   STD_LOGIC_VECTOR(6 downto 0) := "0110000";
    constant TWO:   STD_LOGIC_VECTOR(6 downto 0) := "1101101";
    constant THREE: STD_LOGIC_VECTOR(6 downto 0) := "1111001";
```

```
    constant FOUR:  STD_LOGIC_VECTOR(6 downto 0) := "0110011";
    constant FIVE:  STD_LOGIC_VECTOR(6 downto 0) := "1011011";
    constant SIX:   STD_LOGIC_VECTOR(6 downto 0) := "1011111";
    constant SEVEN: STD_LOGIC_VECTOR(6 downto 0) := "1110000";
    constant EIGHT: STD_LOGIC_VECTOR(6 downto 0) := "1111111";
    constant NINE:  STD_LOGIC_VECTOR(6 downto 0) := "1111011";
begin
    process(data) begin
        case data is
            when "0000" => segments <= ZERO;
            when "0001" => segments <= ONE;
            when "0010" => segments <= TWO;
            when "0011" => segments <= THREE;
            when "0100" => segments <= FOUR;
            when "0101" => segments <= FIVE;
            when "0110" => segments <= SIX;
            when "0111" => segments <= SEVEN;
            when "1000" => segments <= EIGHT;
            when "1001" => segments <= NINE;
            when others => segments <= BLANK;
        end case;
    end process;
end;
```

This example shows the use of *constant declarations* to make the code more readable. The `case` statement has an `others` clause to display a blank output when the input is outside the range of decimal digits.

Overall, concurrent, conditional, and selected signal assignment statements can mimic the effects of processes with sequential assignments, `if` statements, and `case` statements. Moreover, processes can inadvertently imply sequential logic if the output is not assigned a value for all inputs. One reasonable style of VHDL coding is to use processes only for flip-flops and latches.

B.4.5 Memories

VHDL has an array construct used to describe memories. The following module describes a 64-word × 16-bit RAM that is written when `wrb` is low and otherwise read. Synthesis tools are often restricted to generating gates from a library and produce poor memory arrays. A specialized memory generator is commonly used instead.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity memory is
    port(addr: in  STD_LOGIC_VECTOR(5 downto 0);
         wrb:  in  STD_LOGIC;
         din:  in  STD_LOGIC_VECTOR(15 downto 0);
         dout: out STD_LOGIC_VECTOR(15 downto 0));
end;
```

```
architecture synth of memory is
    type ramtype is array (63 downto 0) of STD_LOGIC_VECTOR(15 downto 0);
    signal mem: ramtype;
begin
    process(addr, wrb, din) begin
        if wrb = '0' then mem(conv_integer(addr)) <= din;
        else dout <= mem(conv_integer(addr));
        end if;
    end process;
end;
```

Note that the memory contents are stored in the `signal` *mem*. It is declared to be of `type` *ramtype*, an array of 64 16-bit words. Arrays are indexed with an integer, but `addr` is a `STD_LOGIC_VECTOR`. Hence, the `conv_integer` function is used to convert between data types. This does not produce any hardware, but keeps the VHDL type system happy.

B.5 Finite State Machines

There are two styles of finite state machines. In *Mealy machines* (Figure B.2(a)), the output is a function of the current state and inputs. In *Moore machines* (Figure B.2(b)), the output is a function of only the current state. FSMs are modeled in VHDL with `process` defining the *state registers* and combinational logic defining the *next state* and *output logic*.

Let us first consider a simple finite state machine with one output and no inputs, a divide-by-3 counter. The output should be asserted every three clock cycles. The state transition diagram for a Moore machine is shown in Figure B.3. The output value is labeled in each state because the output is only a function of the state.

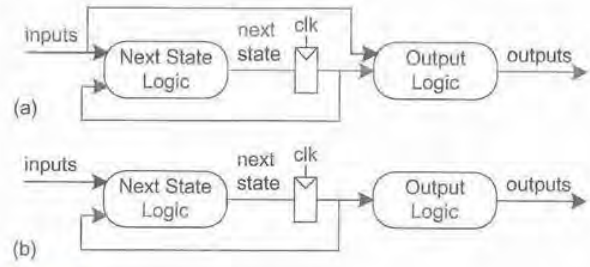


FIG B.2 Moore and Mealy machines

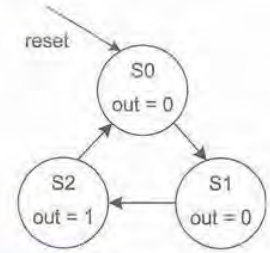


FIG B.3 Divide-by-3 counter state transition diagram

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity divideby3FSM is
    port(clk, reset: in  STD_LOGIC;
         q:          out STD_LOGIC);
end;

architecture synth of divideby3FSM is
    type statetype is (S0, S1, S2);
    signal state, nextstate: statetype;
begin
    -- state register
    process(clk, reset) begin
```

```
        if reset = '1' then state <= S0;
        elsif clk'event and clk = '1' then state <= nextstate;
        end if;
    end process;

    -- next state logic
    nextstate <= S1 when state = S0 else
                 S2 when state = S1 else
                 S0;

    -- output logic
    q <= '1' when state = S2 else '0';
end;
```

Defining states with a type rather than a binary encoding makes the code easier to read. It also allows the synthesis tool to search for an encoding that is fastest or uses the least area. For example, the synthesis tool might choose `S0` = 00, `S1` = 01, and `S2` = 10 or might choose `S0` = 000, `S1` = 010, `S2` = 100.

The FSM model is divided into three portions: the state register, next state logic, and output logic. The state register logic describes an asynchronously resettable register that resets to an initial state and otherwise advances to the computed next state. The next state logic computes `nextstate` as a function of `state` and the inputs; in this example, there are no inputs. The final `else` is essential to make sure `nextstate` gets a value even when `state` has an illegal value; otherwise, latches might be implied. Finally, the output logic may be a function of the current state alone in a Moore machine or of the current state and inputs in a Mealy machine.

It would be tempting to write the output logic as

```
q <= (state = S2);
```

Recall that `(state = S2)` returns a `boolean` result, either `true` or `false`. `q` is of type `STD_LOGIC`. VHDL is picky about types, so the `when/else` clause is needed to convert Boolean to '1' or '0' logic values.

The next example shows a finite state machine with an input `a` and two outputs. Output `x` is true when the input is the same now as it was last cycle. Output `y` is true when the input is the same now as it was for the past two cycles. This is a Mealy machine because the output depends on the current inputs as well as the state. The outputs are labeled on each transition after the input. The state transition diagram is shown in Figure B.4.

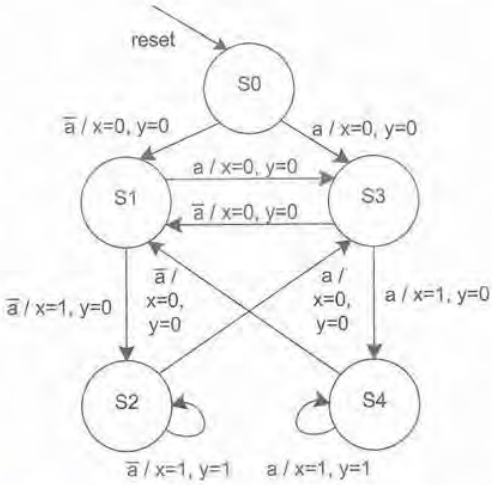


FIG B.4 History FSM state transition diagram

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity historyFSM is
    port(clk, reset: in  STD_LOGIC;
```

```
        a:          in  STD_LOGIC;
        x, y:       out STD_LOGIC);
end;

architecture synth of historyFSM is
    type statetype is (S0, S1, S2, S3, S4);
    signal state, nextstate: statetype;
begin
    -- state register
    process(clk, reset) begin
        if reset = '1' then state <= S0;
        elsif clk'event and clk = '1' then state <= nextstate;
        end if;
    end process;

    -- next state logic
    process(state, a) begin
        case state is
            when S0 =>  if a = '1' then nextstate <= S3;
                        else            nextstate <= S1;
                        end if;
            when S1 =>  if a = '1' then nextstate <= S3;
                        else            nextstate <= S2;
                        end if;
            when S2 =>  if a = '1' then nextstate <= S3;
                        else            nextstate <= S2;
                        end if;
            when S3 =>  if a = '1' then nextstate <= S4;
                        else            nextstate <= S1;
                        end if;
            when S4 =>  if a = '1' then nextstate <= S4;
                        else            nextstate <= S1;
                        end if;
            when others =>              nextstate <= S0;
        end case;
    end process;

    -- output logic
    x <= '1' when ((state = S1 or state = s2) and a = '1') or
                  ((state = S3 or state = S4) and a = '1')
             else '1';
    y <= '1' when (state = S2 and a = '1') or (state = S4 and a = '1')
             else '1';
end;
```

B.6 Parameterized Blocks

So far, all of our blocks have had fixed-width inputs and outputs. Thus, we have needed separate entities for 4- and 8-bit wide 2-input multiplexers. VHDL can automatically

produce hardware with parameterized bit width. For example, we can declare a parameterized bank of inverters with a default of 8 gates as:

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity inv is
    generic(width: integer := 8);
    port(a: in  STD_LOGIC_VECTOR(width-1 downto 0);
         y: out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture synth of inv is
begin
    y <= not a;
end;
```

We can adjust the parameter when we instantiate the block. For example, we can build a bank of 12 buffers using a pair of banks of 12 inverters.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity buf is
    generic(numbits: integer := 12);
    port(a: in  STD_LOGIC_VECTOR(numbits-1 downto 0);
         y: out STD_LOGIC_VECTOR(numbits-1 downto 0));
end;

architecture synth of buf is
    component inv generic(width: integer);
                  port(a: in  STD_LOGIC_VECTOR(width-1 downto 0);
                       y: out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    signal x: STD_LOGIC_VECTOR(numbits-1 downto 0);
begin
    i1: inv generic map(numbits) port map(a, x);
    i2: inv generic map(numbits) port map(x, y);
end;
```

VHDL also has the ability to generate an adjustable number of gates connected in arbitrary ways. The following code produces an N-input AND gate as a cascade of $N-1$ 2-input AND gates (N = `width` = 8). It uses the `generate` command in conjunction with `for` and `if` statements to conditionally produce an array of gates. `Generate` can easily produce large numbers of gates, so use it with care to be sure you intend to imply all the hardware.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity andN is
    generic(width: integer := 8);
    port(a: in  STD_LOGIC_VECTOR(width-1 downto 0);
         y: out STD_LOGIC);
end;
```

```
architecture synth of andN is
    signal i: integer;
    signal x: STD_LOGIC_VECTOR(width-1 downto 1);
begin
    AllBits: for i in width-1 downto 1 generate
        LowBit: if i = 1 generate
            A1: x(1) <= a(0) and a(1);
          end generate;
          OtherBits: if i /= 1 generate
            Ai: x(i) <= a(i) and x(i-1);
          end generate;
    end generate;
    y <= x(width-1);
end;
```

In a similar example, `generate` can instantiate an array of other blocks. For example, an N-bit ripple-carry adder can be built from N full adders with the carry signals chained together. Of course, using the + symbol to imply an adder is less effort and gives the synthesis tool more freedom to trade off area and delay as the constraints demand.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity rippleadder is
    generic(width: integer := 8);
    port(a, b: in  STD_LOGIC_VECTOR(width-1 downto 0);
         s:    out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture synth of rippleadder is
    component fulladder port(a, b, cin:  in  STD_LOGIC;
                             s, cout:    out STD_LOGIC);
    end component;
    signal i: integer;
    signal c: STD_LOGIC_VECTOR(width-1 downto 0);
begin
    AllBits: for i in width-1 downto 0 generate
        LowBit: if i = 0 generate
            FA0: fulladder port map(a(0), b(0), '0', s(0), c(0));
        end generate;
        OtherBits: if i /= 0 generate
            FAi: fulladder port map(a(i), b(i), c(i-1), s(i), c(i));
        end generate;
    end generate;
end;
```

B.7 Example: MIPS Processor

To illustrate a nontrivial VHDL design, this section lists the VHDL code and test bench for the MIPS processor subset discussed in Chapter 1. The example handles only the LB, SB, ADD, SUB, AND, OR, SLT, BEQ, and J instructions. It uses an 8-bit datapath and only eight

registers. As the instruction is 32 bits wide, it is loaded in four successive fetch cycles across an 8-bit path to external memory.

The test bench initializes a 512-byte memory with instructions and data from a text file. The code exercises each of the instructions. The `mipstest.asm` assembly language file and `memfile.dat` text file are shown in Section A.10. The test bench runs until it observes a memory write. If the value 7 is written to address 5, the code executed correctly. If all goes well, the test bench should take 100 cycles (1000 ns) to run.

The code uses the `conv_std_logic_vector` function that converts an integer into a `STD_LOGIC_VECTOR` of the specified width. This is helpful to create constants of parameterized widths. The function is in the `IEEE.STD_LOGIC_ARITH` library. It also uses the `**` exponentiation operator to determine the number of register file entries given the number of bits of register address.

```
-------------------------------------------------------------
-- mips.vhd
-- David_Harris@hmc.edu 9/9/03
-- Model of subset of MIPS processor described in Ch 1
-------------------------------------------------------------

-------------------------------------------------------------
-- Entity Declarations
-------------------------------------------------------------

library IEEE; use IEEE.STD_LOGIC_1164.all; use IEEE.STD_LOGIC_UNSIGNED.all;
entity top is -- top-level design for testing
    generic(width:   integer := 8;      -- default 8-bit datapath
            regbits: integer := 3);     -- and 3 bit register addresses (8 regs)
end;

library IEEE; use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.STD_LOGIC_UNSIGNED.all;  use IEEE.STD_LOGIC_ARITH.all;
entity memory is -- external memory accessed by MIPS
    generic(width: integer);
    port(clk, memwrite:  in STD_LOGIC;
         adr, writedata: in STD_LOGIC_VECTOR(width-1 downto 0);
         memdata:        out STD_LOGIC_VECTOR(width-1 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mips is -- simplified MIPS processor
    generic(width:   integer := 8;      -- default 8-bit datapath
            regbits: integer := 3);     -- and 3 bit register addresses (8 regs)
    port(clk, reset:        in  STD_LOGIC;
         memdata:           in  STD_LOGIC_VECTOR(width-1 downto 0);
         memread, memwrite: out STD_LOGIC;
         adr, writedata:    out STD_LOGIC_VECTOR(width-1 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity controller is -- control FSM
    port(clk, reset:                        in  STD_LOGIC;
         op:                                in  STD_LOGIC_VECTOR(5 downto 0);
         zero:                              in  STD_LOGIC;
         memread, memwrite, alusrca, memtoreg,
         iord, pcen, regwrite, regdst:      out STD_LOGIC;
         pcsource, alusrcb, aluop:          out STD_LOGIC_VECTOR(1 downto 0);
         irwrite:                           out STD_LOGIC_VECTOR(3 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity alucontrol is -- ALU control decoder
    port(aluop:   in  STD_LOGIC_VECTOR(1 downto 0);
         funct:   in  STD_LOGIC_VECTOR(5 downto 0);
```

```
        alucont: out STD_LOGIC_VECTOR(2 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all; use IEEE.STD_LOGIC_ARITH.all;
entity datapath is  -- MIPS datapath
    generic(width, regbits: integer);
    port(clk, reset:        in  STD_LOGIC;
         memdata:           in  STD_LOGIC_VECTOR(width-1 downto 0);
         alusrca, memtoreg, iord, pcen,
         regwrite, regdst:  in  STD_LOGIC;
         pcsource, alusrcb: in  STD_LOGIC_VECTOR(1 downto 0);
         irwrite:           in  STD_LOGIC_VECTOR(3 downto 0);
         alucont:           in  STD_LOGIC_VECTOR(2 downto 0);
         zero:              out STD_LOGIC;
         instr:             out STD_LOGIC_VECTOR(31 downto 0);
         adr, writedata:    out STD_LOGIC_VECTOR(width-1 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all; use IEEE.STD_LOGIC_UNSIGNED.all;
entity alu is -- Arithmetic/Logic unit with add/sub, AND, OR, set less than
    generic(width: integer);
    port(a, b:    in  STD_LOGIC_VECTOR(width-1 downto 0);
         alucont: in  STD_LOGIC_VECTOR(2 downto 0);
         result:  out STD_LOGIC_VECTOR(width-1 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all; use IEEE.STD_LOGIC_ARITH.all;
entity regfile is -- three-port register file of 2**regbits words x width bits
    generic(width, regbits: integer);
    port(clk:          in  STD_LOGIC;
         write:        in  STD_LOGIC;
         ra1, ra2, wa: in  STD_LOGIC_VECTOR(regbits-1 downto 0);
         wd:           in  STD_LOGIC_VECTOR(width-1 downto 0);
         rd1, rd2:     out STD_LOGIC_VECTOR(width-1 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity zerodetect is -- true if all input bits are zero
    generic(width: integer);
    port(a: in  STD_LOGIC_VECTOR(width-1 downto 0);
         y: out STD_LOGIC);
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity flop is -- flip-flop
    generic(width: integer);
    port(clk: in  STD_LOGIC;
         d:   in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:   out STD_LOGIC_VECTOR(width-1 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity flopen is -- flip-flop with enable
    generic(width: integer);
    port(clk, en: in  STD_LOGIC;
         d:       in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:       out STD_LOGIC_VECTOR(width-1 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all; use IEEE.STD_LOGIC_ARITH.all;
entity flopenr is -- flip-flop with enable and synchronous reset
    generic(width: integer);
    port(clk, reset, en: in  STD_LOGIC;
         d:              in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:              out STD_LOGIC_VECTOR(width-1 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux2 is -- two-input multiplexer
    generic(width: integer);
    port(d0, d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
         s:      in  STD_LOGIC;
         y:      out STD_LOGIC_VECTOR(width-1 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux4 is  -- four-input multiplexer
    generic(width: integer);
```

```
    port(d0, d1, d2, d3: in  STD_LOGIC_VECTOR(width-1 downto 0);
         s:              in  STD_LOGIC_VECTOR(1 downto 0);
         y:              out STD_LOGIC_VECTOR(width-1 downto 0));
end;

------------------------------------------------------------
-- Architecture Definitions
------------------------------------------------------------
architecture test of top is
    component mips generic(width:   integer := 8;   -- default 8-bit datapath
                           regbits: integer := 3);  -- and 3 bit register addresses (8 regs)
        port(clk, reset:        in  STD_LOGIC;
             memdata:           in  STD_LOGIC_VECTOR(width-1 downto 0);
             memread, memwrite: out STD_LOGIC;
             adr, writedata:    out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component memory generic(width: integer);
        port(clk, memwrite:  in  STD_LOGIC;
             adr, writedata: in  STD_LOGIC_VECTOR(width-1 downto 0);
             memdata:        out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    signal clk, reset, memread, memwrite: STD_LOGIC;
    signal memdata, adr, writedata: STD_LOGIC_VECTOR(width-1 downto 0);
begin
    -- mips being tested
    dut: mips generic map(width, regbits)
             port map(clk, reset, memdata, memread, memwrite, adr, writedata);
    -- external memory for code and data
    exmem: memory generic map(width)
                  port map(clk, memwrite, adr, writedata, memdata);

    -- Generate clock with 10 ns period
    process begin
        clk <= '1';
        wait for 5 ns;
        clk <= '0';
        wait for 5 ns;
    end process;

    -- Generate reset for first two clock cycles
    process begin
        reset <= '1';
        wait for 22 ns;
        reset <= '0';
        wait;
    end process;

    -- check that 7 gets written to address 5 at end of program
    process (clk) begin
        if (clk'event and clk = '0' and memwrite = '1') then
            if (conv_integer(adr) = 5 and conv_integer(writedata) = 7) then
                report "Simulation completed successfully";
            else report "Simulation failed.";
            end if;
        end if;
    end process;
end;

architecture synth of memory is
begin
    process is
        file mem_file: text open read_mode is "memfile.dat";
        variable L: line;
        variable ch: character;
        variable index, result: integer;
        type ramtype is array (511 downto 0) of STD_LOGIC_VECTOR(7 downto 0);
        variable mem: ramtype;
    begin
        -- initialize memory from file
        for i in 0 to 511 loop -- set all contents low
            mem(conv_integer(i)) := "00000000";
        end loop;
        index := 0;
        while not endfile(mem_file) loop
            readline(mem_file, L);
            for j in 0 to 3 loop
                result := 0;
                    for i in 1 to 2 loop
                        read(L, ch);
                        if '0' <= ch and ch <= '9' then
                            result := result*16 + character'pos(ch)-character'pos('0');
                        elsif 'a' <= ch and ch <= 'f' then
```

```
                        result := result*16 + character'pos(ch)-character'pos('a')+10;
                    else report "Format error on line " & integer'image(index)
                        severity error;
                    end if;
                end loop;
                mem(index*4+j) := conv_std_logic_vector(result, width);
            end loop;
            index := index + 1;
        end loop;
        -- read or write memory
        loop
            if clk'event and clk = '1' then
                if (memwrite = '1') then mem(conv_integer(adr)) := writedata;
                end if;
            end if;
            memdata <= mem(conv_integer(adr));
            wait on clk, adr;
        end loop;
    end process;
end;

architecture struct of mips is
    component controller
        port(clk, reset:                    in  STD_LOGIC;
             op:                            in  STD_LOGIC_VECTOR(5 downto 0);
             zero:                          in  STD_LOGIC;
             memread, memwrite, alusrca, memtoreg,
             iord, pcen, regwrite, regdst: out STD_LOGIC;
             pcsource, alusrcb, aluop:     out STD_LOGIC_VECTOR(1 downto 0);
             irwrite:                       out STD_LOGIC_VECTOR(3 downto 0));
    end component;
    component alucontrol
        port(aluop:      in  STD_LOGIC_VECTOR(1 downto 0);
             funct:      in  STD_LOGIC_VECTOR(5 downto 0);
             alucont:    out STD_LOGIC_VECTOR(2 downto 0));
    end component;
    component datapath generic(width, regbits: integer);
        port(clk, reset:         in  STD_LOGIC;
             memdata:            in  STD_LOGIC_VECTOR(width-1 downto 0);
             alusrca, memtoreg, iord, pcen,
             regwrite, regdst:   in  STD_LOGIC;
             pcsource, alusrcb: in  STD_LOGIC_VECTOR(1 downto 0);
             irwrite:            in  STD_LOGIC_VECTOR(3 downto 0);
             alucont:            in  STD_LOGIC_VECTOR(2 downto 0);
             zero:               out STD_LOGIC;
             instr:              out STD_LOGIC_VECTOR(31 downto 0);
             adr, writedata:     out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    signal instr: STD_LOGIC_VECTOR(31 downto 0);
    signal zero, alusrca, memtoreg, iord, pcen, regwrite, regdst: STD_LOGIC;
    signal aluop, pcsource, alusrcb: STD_LOGIC_VECTOR(1 downto 0);
    signal irwrite: STD_LOGIC_VECTOR(3 downto 0);
    signal alucont: STD_LOGIC_VECTOR(2 downto 0);
begin
    cont: controller port map(clk, reset, instr(31 downto 26), zero,
                              memread, memwrite, alusrca, memtoreg,
                              iord, pcen, regwrite, regdst,
                              pcsource, alusrcb, aluop, irwrite);
    ac: alucontrol port map(aluop, instr(5 downto 0), alucont);
    dp: datapath generic map(width, regbits)
                 port map(clk, reset, memdata, alusrca, memtoreg,
                          iord, pcen, regwrite, regdst,
                          pcsource, alusrcb, irwrite,
                          alucont, zero, instr, adr, writedata);
end;

architecture synth of controller is
    type statetype is (FETCH1, FETCH2, FETCH3, FETCH4, DECODE, MEMADR,
                       LBRD, LBWR, SBWR, RTYPEEX, RTYPEWR, BEQEX, JEX);
        constant LB:    STD_LOGIC_VECTOR(5 downto 0) := "100000";
        constant SB:    STD_LOGIC_VECTOR(5 downto 0) := "101000";
        constant RTYPE: STD_LOGIC_VECTOR(5 downto 0) := "000000";
        constant BEQ:   STD_LOGIC_VECTOR(5 downto 0) := "000100";
        constant J:     STD_LOGIC_VECTOR(5 downto 0) := "000010";
        signal state, nextstate:    statetype;
        signal pcwrite, pcwritecond: STD_LOGIC;
begin
    process (clk) begin -- state register
        if clk'event and clk = '1' then
            if reset = '1' then state <= FETCH1;
```

```
        else state <= nextstate;
        end if;
    end if;
end process;

process (state, op) begin -- next state logic
    case state is
        when FETCH1 => nextstate <= FETCH2;
        when FETCH2 => nextstate <= FETCH3;
        when FETCH3 => nextstate <= FETCH4;
        when FETCH4 => nextstate <= DECODE;
        when DECODE => case op is
                           when LB | SB => nextstate <= MEMADR;
                           when RTYPE => nextstate <= RTYPEEX;
                           when BEQ => nextstate <= BEQEX;
                           when J => nextstate <= JEX;
                           when others => nextstate <= FETCH1; -- should never happen
                       end case;
        when MEMADR => case op is
                           when LB => nextstate <= LBRD;
                           when SB => nextstate <= SBWR;
                           when others => nextstate <= FETCH1; -- should never happen
                       end case;
        when LBRD => nextstate <= LBWR;
        when LBWR => nextstate <= FETCH1;
        when SBWR => nextstate <= FETCH1;
        when RTYPEEX => nextstate <= RTYPEWR;
        when RTYPEWR => nextstate <= FETCH1;
        when BEQEX => nextstate <= FETCH1;
        when JEX => nextstate <= FETCH1;
        when others => nextstate <= FETCH1; -- should never happen
    end case;
end process;

process (state) begin
    -- set all outputs to zero, then conditionally assert just the appropriate ones
    irwrite <= "0000";
    pcwrite <= '0'; pcwritecond <= '0';
    regwrite <= '0';  regdst <= '0';
    memread <= '0'; memwrite <= '0';
    alusrca <= '0'; alusrcb <= "00"; aluop <= "00";
    pcsource <= "00";
    iord <= '0'; memtoreg <= '0';

    case state is
        when FETCH1 => memread <= '1';
                       irwrite <= "1000";
                       alusrcb <= "01";
                       pcwrite <= '1';
        when FETCH2 => memread <= '1';
                       irwrite <= "0100";
                       alusrcb <= "01";
                       pcwrite <= '1';
        when FETCH3 => memread <= '1';
                       irwrite <= "0010";
                       alusrcb <= "01";
                       pcwrite <= '1';
        when FETCH4 => memread <= '1';
                       irwrite <= "0001";
                       alusrcb <= "01";
                       pcwrite <= '1';
        when DECODE => alusrcb <= "11";
        when MEMADR => alusrca <= '1';
                       alusrcb <= "10";
        when LBRD =>   memread <= '1';
                       iord <= '1';
        when LBWR =>   regwrite <= '1';
                       memtoreg <= '1';
        when SBWR =>   memwrite <= '1';
                       iord <= '1';
        when RTYPEEX => alusrca <= '1';
                        aluop <= "10";
        when RTYPEWR => regdst <= '1';
                        regwrite <= '1';
        when BEQEX =>  alusrca <= '1';
                       aluop <= "01";
                       pcwritecond <= '1';
                       pcsource <= "01";
        when JEX =>    pcwrite <= '1';
                       pcsource <= "10";
```

```
        end case;
    end process;

    pcen <= pcwrite or (pcwritecond and zero); -- program counter enable
end;

architecture synth of alucontrol is
begin
    process(aluop, funct) begin
        case aluop is
            when "00" => alucont <= "010"; -- add (for lb/sb/addi)
            when "01" => alucont <= "110"; -- sub (for beq)
            when others => case funct is          -- R-type instructions
                               when "100000" => alucont <= "010"; -- add (for add)
                               when "100010" => alucont <= "110"; -- subtract (for sub)
                               when "100100" => alucont <= "000"; -- logical and (for and)
                               when "100101" => alucont <= "001"; -- logical or (for or)
                               when "101010" => alucont <= "111"; -- set on less (for slt)
                               when others   => alucont <= "---"; -- should never happen
                           end case;
        end case;
    end process;
end;

architecture struct of datapath is
    component alu generic(width: integer);
        port(a, b:    in  STD_LOGIC_VECTOR(width-1 downto 0);
             alucont: in  STD_LOGIC_VECTOR(2 downto 0);
             result:  out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component regfile generic(width, regbits: integer);
        port(clk:          in  STD_LOGIC;
             write:        in  STD_LOGIC;
             ra1, ra2, wa: in  STD_LOGIC_VECTOR(regbits-1 downto 0);
             wd:           in  STD_LOGIC_VECTOR(width-1 downto 0);
             rd1, rd2:     out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component zerodetect generic(width: integer);
        port(a: in  STD_LOGIC_VECTOR(width-1 downto 0);
             y: out STD_LOGIC);
    end component;
    component flop generic(width: integer);
        port(clk: in  STD_LOGIC;
             d:   in  STD_LOGIC_VECTOR(width-1 downto 0);
             q:   out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component flopen generic(width: integer);
        port(clk, en: in  STD_LOGIC;
             d:       in  STD_LOGIC_VECTOR(width-1 downto 0);
             q:       out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component flopenr generic(width: integer);
        port(clk, reset, en: in  STD_LOGIC;
             d:              in  STD_LOGIC_VECTOR(width-1 downto 0);
             q:              out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component mux2 generic(width: integer);
        port(d0, d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
             s:      in  STD_LOGIC;
             y:      out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component mux4 generic(width: integer);
        port(d0, d1, d2, d3: in  STD_LOGIC_VECTOR(width-1 downto 0);
             s:              in  STD_LOGIC_VECTOR(1 downto 0);
             y:              out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    constant CONST_ONE:  STD_LOGIC_VECTOR(width-1 downto 0) := conv_std_logic_vector(1, width);
    constant CONST_ZERO: STD_LOGIC_VECTOR(width-1 downto 0) := conv_std_logic_vector(0, width);
    signal ra1, ra2, wa: STD_LOGIC_VECTOR(regbits-1 downto 0);
    signal pc, nextpc, md, rd1, rd2, wd, a,
           src1, src2, aluresult, aluout, dp_writedata, constx4: STD_LOGIC_VECTOR(width-1 downto 0);
    signal dp_instr: STD_LOGIC_VECTOR(31 downto 0);

begin
    -- shift left constant field by 2
    constx4 <= dp_instr(width-3 downto 0) & "00";

    -- register file address fields
    ra1 <= dp_instr(regbits+20 downto 21);
    ra2 <= dp_instr(regbits+15 downto 16);
```

```
    regmux:  mux2 generic map(regbits) port map(dp_instr(regbits+15 downto 16),
                                                dp_instr(regbits+10 downto 11), regdst, wa);

    -- independent of bit width, load dp_instruction into four 8-bit registers over four cycles
    ir0:     flopen generic map(8) port map(clk, irwrite(0), memdata(7 downto 0), dp_instr(7 downto 0));
    ir1:     flopen generic map(8) port map(clk, irwrite(1), memdata(7 downto 0), dp_instr(15 downto 8));
    ir2:     flopen generic map(8) port map(clk, irwrite(2), memdata(7 downto 0), dp_instr(23 downto 16));
    ir3:     flopen generic map(8) port map(clk, irwrite(3), memdata(7 downto 0), dp_instr(31 downto 24));
    -- datapath
    pcreg:   flopenr generic map(width) port map(clk, reset, pcen, nextpc, pc);
    mdr:     flop generic map(width) port map(clk, memdata, md);
    areg:    flop generic map(width) port map(clk, rd1, a);
    wrd:     flop generic map(width) port map(clk, rd2, dp_writedata);
    res:     flop generic map(width) port map(clk, aluresult, aluout);
    adrmux:  mux2 generic map(width) port map(pc, aluout, iord, adr);
    src1mux: mux2 generic map(width) port map(pc, a, alusrca, src1);
    src2mux: mux4 generic map(width) port map(dp_writedata, CONST_ONE,
                                              dp_instr(width-1 downto 0), constx4, alusrcb, src2);
    pcmux:   mux4 generic map(width) port map(aluresult, aluout, constx4, CONST_ZERO, pcsource, nextpc);
    wdmux:   mux2 generic map(width) port map(aluout, md, memtoreg, wd);
    rf:      regfile generic map(width, regbits) port map(clk, regwrite, ra1, ra2, wa, wd, rd1, rd2);
    alunit:  alu generic map(width) port map(src1, src2, alucont, aluresult);
    zd:      zerodetect generic map(width) port map(aluresult, zero);

    -- drive outputs
    instr <= dp_instr; writedata <= dp_writedata;
end;

architecture synth of alu is
    signal b2, sum, slt: STD_LOGIC_VECTOR(width-1 downto 0);
begin
    b2 <= not b when alucont(2) = '1' else b;
    sum <= a + b2 + alucont(2);
    -- slt should be 1 if most significant bit of sum is 1
    slt <= conv_std_logic_vector(1, width) when sum(width-1) = '1'
           else conv_std_logic_vector(0, width);
    with alucont(1 downto 0) select result <=
        a and b when "00",
        a or b  when "01",
        sum     when "10",
        slt     when others;
end;

architecture synth of regfile is
    type ramtype is array (2**regbits-1 downto 0) of STD_LOGIC_VECTOR(width-1 downto 0);
        signal mem: ramtype;
begin
    -- three-ported register file
    -- read two ports combinationally
    -- write third port on rising edge of clock
    process(clk) begin
        if clk'event and clk = '1' then
            if write = '1' then mem(conv_integer(we)) <= wd;
            end if;
        end if;
    end process;
    process(ra1, ra2) begin
        if (conv_integer(ra1) = 0) then rd1 <= conv_std_logic_vector(0, width); -- register 0 holds 0
        else rd1 <= mem(conv_integer(ra1));
        end if;
        if (conv_integer(ra2) = 0) then rd2 <= conv_std_logic_vector(0, width);
        else rd2 <= mem(conv_integer(ra2));
        end if;
    end process;
end;

architecture synth of zerodetect is
    signal i: integer;
    signal x: STD_LOGIC_VECTOR(width-1 downto 1);
begin -- N-bit AND of inverted inputs
    AllBits: for i in width-1 downto 1 generate
        LowBit: if i = 1 generate
                    A1: x(1) <= not a(0) and not a(1);
                end generate;
                OtherBits: if i /= 1 generate
                    Ai: x(i) <= not a(i) and x(i-1);
                end generate;
        end generate;
    y <= x(width-1);
end;
```

```
architecture synth of flop is
begin
    process(clk) begin
        if clk'event and clk = '1' then -- or use "if RISING_EDGE(clk) then"
            q <= d;
        end if;
    end process;
end;

architecture synth of flopen is
begin
    process(clk) begin
        if clk'event and clk = '1' then
            if en = '1' then q <= d;
            end if;
        end if;
    end process;
end;

architecture synchronous of flopenr is
begin
    process(clk) begin
        if clk'event and clk = '1' then
            if reset = '1' then
                q <= CONV_STD_LOGIC_VECTOR(0, width); -- produce a vector of all zeros
            elsif en = '1' then q <= d;
            end if;
        end if;
    end process;
end;

architecture synth of mux2 is
begin
    y <= d0 when s = '0' else d1;
end;

architecture synth of mux4 is
begin
    y <= d0 when s = "00" else
         d1 when s = "01" else
         d2 when s = "10" else
         d3;
end;
```

References

The IEEE Journal of Solid-State Circuits is abbreviated as *JSSC* because it is cited heavily. Most of the references in IEEE publications since 1988 can be obtained from `ieeexplore.ieee.org`.

[Acken83] J. Acken, "Testing for bridging faults (shorts) in CMOS circuits," *Proc. Design Automation Conf.*, 1983, pp. 717-718.

[Acosta95] A. Acosta, M. Valencia, A. Barriga, M. Bellido, and J. Huertas, "SODS: A new CMOS differential-type structure," *JSSC*, vol. 30, no. 7, July 1995, pp. 835-838.

[Afghahi90] M. Afghahi and C. Svensson, "A unified single-phase clocking scheme for VLSI systems," *JSSC*, vol. 25, no. 1, Feb. 1990, pp. 225-233.

[Allam00] M. Allam, M. Anis, and M. Elmasry, "High-speed dynamic logic styles for scaled-down CMOS and MTCMOS technologies," *Proc. Intl. Symp. Low Power Electronics and Design*, 2000, pp. 155-160.

[Allam01] M. Allam and M. Elmasry, "Dynamic current mode logic (DyCML): a new low-power high-performance logic style," *JSSC*, vol. 36, no. 3, March 2001, pp. 550-558.

[Alvandpour02] A. Alvandpour, R. Krishnamurthy, K. Soumyanath, and S. Borkar, "A sub-130-nm conditional keeper technique," *JSSC*, vol. 37, no. 5, May 2002, pp. 633-638.

[Amrutur98] B. Amrutur and M. Horowitz, "A replica technique for wordline and sense control in low-power SRAM's," *JSSC*, vol. 33, no. 8, Aug. 1998, pp. 1208-1219.

[Amrutur00] B. Amrutur and M. Horowitz, "Speed and power scaling of SRAM's," *JSSC*, vol. 35, no. 2, Feb. 2000, pp. 175-185.

[Amrutur01] B. Amrutur and M. Horowitz, "Fast low-power decoders for RAMs," *JSSC*, vol. 36, no. 10, Oct. 2001, pp. 1506-1515.

[Anastasakis02] D. Anastaskasis, R. Damiano, H. Ma, and T. Stanion, "A practical and efficient method for compare-point matching," *Proc. Design Automation Conf.*, June 2002, pp. 305-310.

[Anderson02] F. Anderson, J. Wells, and E. Berta, "The core clock system on the next generation Itanium microprocessor," *Proc. IEEE Intl. Solid-State Circuits Conf.*, Feb. 2002, pp. 146-147, 453.

[Ando80] H. Ando, "Testing VLSI with random access scan," *Digest of Papers COMPCON 80*, Feb. 1980, pp. 50-52.

[Artisan02] Artisan Components, *TSMC 0.18μm Process 1.8-Volt SAGE-X Standard Cell Library Databook*, Release 4.0, Feb. 2002.

[Ashenden01] P. Ashenden, *The Designer's Guide to VHDL*, 2nd ed., San Francisco, CA: Morgan Kaufmann, 2001.

[Ayers03] D. Ayers, "VLSI Power Delivery," EE371 Lecture Notes, Stanford University, April 29, 2003.

[Baghini02] M. Baghini and M. Desai, "Impact of technology scaling on metastability performance of CMOS synchronizing latches," *Proc. Intl. Conf. VLSI Design*, 2002, pp. 317-322.

[Bailey98] D. Bailey and B. Benschneider, "Clocking design and analysis for a 600-MHz Alpha microprocessor," *JSSC*, vol. 33, no. 11, Nov. 1998, pp. 1627-1633.

[Baker97] K. Baker and J. van Beers, "Shmoo plotting: the black art of IC testing," *IEEE Design and Test of Computers*, vol. 14, no. 3, July-Sept. 1997, pp. 90-97.

[Baker98] R. Jacob Baker, H. Li, and D. Boyce, *CMOS Circuit Design, Layout, and Simulation*, New York: Wiley-Interscience, 1998.

[Baker02] J. Baker, *CMOS Mixed-Signal Circuit Design*, Piscataway, NJ: IEEE Press, 2002.

[Bakoglu90] H. Bakoglu, *Circuits, Interconnections, and Packaging for VLSI*, Reading, MA: Addison-Wesley, 1990.

[Balamurugan01] G. Balamurugan and N. Shanbhag, "The twin-transistor noise-tolerant dynamic circuit technique," *JSSC*, vol. 36, no. 2, Feb. 2001, pp. 273-280.

[Barke88] E. Barke, "Line-to-ground capacitance calculation for VLSI: a comparison," *IEEE Trans. Computer-Aided Design*, vol. 7, no. 2, Feb. 1988, pp. 295-298.

[Baugh73] C. Baugh and B. Wooley, "A two's complement parallel array multiplication algorithm," *IEEE Trans. Computers*, vol. C-22, no. 12, Dec. 1973, pp. 1045-1047.

[Beaumont-Smith99] A. Beaumont-Smith, N. Burgess, S. Lefrere, and C. Lim, "Reduced latency IEEE floating-point adder architectures," *Proc. IEEE Symp. Computer Arithmetic*, April 1999, pp. 35-42.

[Beaumont-Smith01] A. Beaumont-Smith and C. Lim, "Parallel prefix adder design," *Proc. IEEE Symp. Computer Arithmetic*, 2001, pp. 218-225.

[Bedrij62] O. Bedrij, "Carry-select adder," *IRE Trans. Electronic Computers*, vol. 11, June 1962, pp. 340-346.

[Bernstein99] K. Bernstein, K. Carrig, C. Durham, P. Hansen, D. Hogenmiller, E. Nowak, and N. Roher, *High Speed CMOS Design Styles*, Boston: Kluwer Academic Publishers, 1999.

[Bernstein00] K. Bernstein and N. Rohrer, *SOI Circuit Design Concepts*, Boston: Kluwer Academic Publishers, 2000.

[Best03] R. Best, *Phase-Locked Loops: Design, Simulation, and Applications*, 5th ed., McGraw-Hill, 2003.

[Bewick94] G. Bewick, Fast Multiplication: Algorithms and Implementation, Ph.D. Thesis, Stanford University, CSL-TR-94-617, 1994.

[Black69] J. Black, "Electromigration—A brief survey and some recent results," *IEEE Trans. Electron Devices*, vol. ED-16, no. 4, April 1969, pp. 338-347.

[Blackburn96] J. Blackburn, L. Arndt, and E. Swartzlander, "Optimization of spanning tree carry lookahead adders," *Proc. 30th Asilomar Conf. Signals, Systems, and Computers*, vol. 1, 1996, pp. 177-181.

[Booth51] A. Booth, "A signed binary multiplication technique," *Quarterly J. Mechanics and Applied Mathematics*, vol. IV, part 2, June 1951, pp. 236-240.

[Borkar03] S. Borkar, T. Kamik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture," *Proc. Design Automation Conf.*, 2003, pp. 338-342.

[Bouldin03] D. Bouldin, A. Miller, and C. Tan, "Teaching custom integrated circuit design and verification." *Proc. Microelectronics Systems Education Conf.*, 2003, pp. 48-49.

[Bowhill95] W. Bowhill et al., "Circuit implementation of a 300-MHz 64-bit second-generation CMOS Alpha CPU," *Digital Technical Journal*, vol. 7, no. 1, 1995, pp. 100-115.

[Bowman99] K. Bowman, B. Austin, J. Eble, X. Tang, and J. Meindl, "A physical alpha-power law MOSFET model," *JSSC*, vol. 34, no. 10, Oct. 1999, pp. 1410-1414.

[Brent82] R. Brent and H. Kung, "A regular layout for parallel adders," *IEEE Trans. Computers*, vol. C-31, no. 3, March 1982, pp. 260-264.

[Brooks95] F. Brooks, *The Mythical Man-Month*, Boston: Addison-Wesley, 1995.

[Brown03] A. Brown, "Fast films," *IEEE Spectrum*, vol. 40, no. 2, Feb. 2003, pp. 36-40.

[Bugeja00] A. Bugeja and B. Song, "A self-trimming 14-b 100MSample/s CMOS DAC," *JSSC*, vol. 35, no. 12, Dec. 2000, pp. 1841-1852.

[Burks46] A. Burks, H. Goldstine, and J. von Neumann, *Preliminary discussion of the logical design of an electronic computing instrument, part 1*, vol. 1, Inst. Advanced Study, Princeton, NJ, 1946.

[Burleson98] W. Burleson, M. Ciesielski, F. Klass, and W. Liu, "Wave-pipelining: a tutorial and research survey," *IEEE Trans. VLSI*, vol. 6, no. 3, Sept. 1998, pp. 464-474.

[Calma84] Calma Corporation, GDS II Stream Format, July 1984.

[Candy76] J. Candy, W. Ninke, and B. Wooley, "A per-channel A/D converter having 15-segment μ-255 companding," *IEEE Transactions on Communications*, vol. 24, no. 1, Jan. 1976, pp. 33-42.

[Carr72] W. Carr and J. Mize, *MOS/LSI Design and Application*, New York: McGraw-Hill, 1972.

[Celik02] M. Celik, L. Pileggi, and A. Odabasioglu, *IC Interconnect Analysis*, Boston: Kluwer Academic Publishers, 2002.

[Chan90] P. Chan and M. Schlag, "Analysis and design of CMOS Manchester adders with variable carry-skip," *IEEE Trans. Computers*, vol. 39, no. 8, Aug. 1990, pp. 983-992.

[Chandrakasan01] A. Chandrakasan, W. Bowhill, and F. Fox, ed., *Design of High-Performance Microprocessor Circuits*, Piscataway, NJ: IEEE Press, 2001.

[Chaney73] T. Chaney and C. Molnar, "Anomalous behavior of synchronizer and arbiter circuits," *IEEE Trans. Computers*, vol. C-22, April 1973, pp. 421-422.

[Chaney83] T. Chaney, "Measured flip-flop responses to marginal triggering," *IEEE Trans. Computers*, vol. C-32, no. 12, Dec. 1983, pp. 1207-1209.

[Chang03] D. Chang and U. Moon, "A 1.4-V 25-MS/s pipelined ADC using opamp-reset switching technique," *JSSC*, vol. 38, no. 8, Aug. 2003, pp. 1401-1404.

[Chao89] H. Chao and C. Johnston, "Behavior analysis of CMOS D flip-flops," *JSSC*, vol. 24, no. 5, Oct. 1989, pp. 1454-1458.

[Chappell91] T. Chappell, B. Chappell, S. Schuster, J. Allan, S. Klepner, R. Joshi, and R. Franch, "A 2-ns cycle, 3.8-ns access 512-kb CMOS ECL SRAM with a fully pipelined architecture," *JSSC*, vol. 26, no. 11, Nov. 1991, pp. 1577-1585.

[Cheng99] Y. Cheng and C. Hu, *MOSFET Modeling & BSIM3 User's Guide*, Boston: Kluwer Academic Publishers, 1999.

[Cheng00y] Y. Cheng, C. Tsai, C. Teng, and S. Kang, *Electrothermal Analysis of VLSI Systems*, Boston: Kluwer Academic Publishers, 2000.

[Chern92] J. Chern, J. Huang, L. Arledge, P. Li, and P. Yang," Multilevel metal capacitance models for CAD design synthesis systems," *IEEE Electron Device Letters*, vol. 13, no. 1, Jan. 1992, pp. 32-34.

[Childs84] R. Childs, J. Crawford, D. House, and R. Noyce, "A processor family for personal computers," *Proc. IEEE*, vol. 72, no. 3, March 1984, pp. 363-376.

[Chillarige03] Y. Chillarige, S. Dubey, S. Sompur, and B. Wong, "A 399ps arithmetic logic unit (ALU) implemented using Propagate (P), Generate (G), and Kill (K) signals in push-pull style for a next generation UltraSparc microprocessor," *Proc. IEEE Custom Integrated Circuits Conf.*, 2003.

[Chinnery02] D. Chinnery and K. Keutzer, *Closing the Gap Between ASIC and Custom: Tools and techniques for high-performance ASIC design*, Boston: Kluwer Academic Publishers, 2002.

[Cho95] T. Cho and P. Gray, "A 10 b, 20 Msample/s, 35mW pipeline A/D converter," *JSSC*, vol. 30, no. 3, March 1995, pp. 166-172.

[Choi97] J. Choi, L. Jang, S. Jung and J. Choi, "Structured design of a 288-tap FIR filter by optimized partial product tree compression," *JSSC*, vol. 32, no. 3, March 1997, pp. 468-476.

[Choudhury97] M. Choudhury and J. Miller, "A 300 MHz CMOS microprocessor with multi-media technology," *Proc. IEEE Intl. Solid-State Circuits Conf.*, 1997, pp. 170-171.

[Chu86] K. Chu and D. Pulfrey, "Design procedures for diffential cascode voltage switch circuits," *JSSC*, vol. SC-21, no. 6, Dec. 1986, pp. 1082-1087.

[Chu87] K. Chu and D. Pulfrey, "A comparison of CMOS circuit techniques: differential cascode voltage switch logic versus conventional logic," *JSSC*, vol. SC-22, no. 4, Aug. 1987, pp. 528-532.

[Ciletti99] M. Ciletti, *Modeling, Synthesis, and Rapid Prototyping with the VERILOG HDL*, Upper Saddle River, NJ: Prentice Hall, 1999.

[Clark02] L. Clark, S. Demmons, N. Deutscher, and F. Ricci, "Standby power management for a 0.18μm microprocessor," *Proc. Intl. Symp. Low Power Electronics and Design*, Aug. 2002, pp. 7-12.

[Cobbold66] R. Cobbold, "Temperature effects on M.O.S. transistors," *Electronics Letters*, vol. 2, no. 6, June 1966, pp. 190-192.

[Cobbold70] R. Cobbold, *Theory and Application of Field Transistors*, New York: Wiley Interscience, 1970.

[Collins01] P. Collins, M. Arnold, and P. Avouris, "Engineering carbon nanotubes and nanotube circuits using electrical breakdown," *Science*, vol. 292, 27 April 2001, pp. 706-709.

[Colwell95] R. Colwell and R. Steck, "A 0.6 μm BiCMOS processor with dynamic execution," *Proc. IEEE Solid-State Circuits Conf.*, 1995, pp. 176-177.

[Cortadella92] J. Cortadella and J. Llaberia, "Evaluation of A+B=K conditions without carry propagation," *IEEE Trans. Computers*, vol. 41, no. 11, Nov. 1992, pp. 1484-1487.

[Covino97] J. Covino, "Dynamic CMOS circuit with noise immunity," US Patent 5,650,733, 1997.

[Crews03] M. Crews and Y. Yuenyongsgool, "Practical design for transferring signals between clock domains," *EDN Magazine*, Feb. 20, 2003, pp. 65-71.

[Curran02] B. Curran et al., "IBM eServer z900 high-frequency microprocessor technology, circuits, and design methodology," *IBM J. Research and Development*, vol. 46, no. 4/5, July/Sept. 2002, pp. 631-644.

[Dabral98] S. Dabral and T. Maloney, *Basic ESD and I/O Design*, New York: John Wiley & Sons, 1998.

[Dadda65] L. Dadda, "Some schemes for parallel multipliers," *Alta Frequenza*, vol. 34, no. 5, May 1965, pp. 349-356.

[Dally98] W. Dally and J. Poulton, *Digital Systems Engineering*, Cambridge, UK: Cambridge University Press, 1998.

[Davari99] B. Davari, "CMOS technology: present and future," *Symp. VLSI Circuits Digest Tech. Papers*, 1999, pp. 5-10.

[Dekker90] R. Dekker, F. Beenker, and L. Thijssen, "A realistic fault model and test algorithms for static random access memories," *IEEE Trans. Computer-Aided Design*, vol. 9, no. 6, June 1990, pp. 567-572.

[Deleganes02] D. Deleganes, J. Douglas, B. Kommandur, and M. Patyra, "Designing a 3GHz, 130nm, Intel Pentium 4 processor," *Symp. VLSI Circuits Digest Tech. Papers*, 2002, pp. 130-133.

[Delgado-Frias00] J. Delgado-Frias and J. Nyathi, "A high-performance encoder with priority lookahead," *IEEE Trans. Circuits and Systems I*, vol. 47, no. 9, Sept. 2000, pp. 1390-1393.

[Dennard68] R. Dennard, "Field-effect transistor memory," US Patent 3,387,286, 1968.

[Dennard74] R. Dennard et al., "Design of ion-implanted MOSFET's with very small physical dimensions," *JSSC*, vol. SC-9, no. 5, Oct. 1974, pp. 256-268.

[Dhanesha95] H. Dhanesha, K. Falakshahi, and M. Horowitz, "Array-of-arrays architecture for parallel floating point multiplication," *Proc. Conf. Advanced Research in VLSI*, 1995, pp. 150-157.

[Dike99] C. Dike and E. Burton, "Miller and noise effects in a synchronizing flip-flop," *JSSC*, vol. 34, no. 6, June 1999, pp. 849-855.

[Dingwall79] A. Dingwall, "Monolithic expandable 6 bit 20 MHz CMOS/SOS A/D converter," *JSSC*, vol. SC-14, no. 6, Dec. 1979, pp. 926-932.

[Dingwall85] A. Dingwall and V. Zazzu, "An 8-MHz CMOS subranging 8-bit A/D converter," *JSSC*, vol. SC-20, no. 6, Dec. 1985, pp. 1138-1143.

[Dobbalaere95] I. Dobbalaere, M. Horowitz, and A. El Gamal, "Regenerative feedback repeaters for programmable interconnect," *JSSC*, vol. 30, no. 11, Nov. 1995, pp. 1246-1253.

[Dobberpuhl92] D. Dobberpuhl et al., "A 200-MHz 64-b dual-issue CMOS microprocessor," *JSSC*, vol. 27, no. 11, Nov. 1992, pp. 1555-1867.

[Dobson95] J. Dobson and G. Blair, "Fast two's complement VLSI adder design," *Electronics Letters*, vol. 31, no. 20, Sept. 1995, pp. 1721-1722.

[Donnay03] S. Donnay and G. Gielen, eds., *Substrate Noise Coupling in Mixed-Signal ASICs*, Boston: Kluwer Academic Publishers, 2003.

[Donovan02] C. Donovan and M. Flynn, "A "digital" 6-bit ADC in 0.25μm CMOS," *JSSC*, vol. 37, no. 3, March 2002, pp. 432-437.

[Doyle91] B. Doyle, B. Fishbein, and K. Mistry, "NBTI-enhanced hot carrier damage in p-channel MOSFETs," *Proc. Intl. Electron Devices Meeting*, 1991, pp. 529-532A.

[Draper97] D. Draper et al., "Circuit techniques in a 266-MHz MMX-enabled processor," *JSSC*, vol. 32, no. 11, Nov. 1997, pp. 1650-1664.

[D'Souza96] G. D'Souza, "Dyanmic logic circuit with reduced charge leakage," US Patent 5,483,181, 1996.

[Edwards93] B. Edwards, A. Corry, N. Weste and C. Greenberg, "A single-chip video ghost canceller," *JSSC*, vol. 28, no. 3, March 1993, pp. 379-383..

[Eichelberger78] E. Eichelberger and T. Williams, "A logic design structure for LSI testability," *J. Design Automation and Fault Tolerant Computing*, vol. 2, no. 2, May 1978, pp. 165-178.

[Elmore48] W. Elmore, "The transient response of damped linear networks with particular regard to wideband amplifiers," *J. Applied Physics*, vol. 19, no. 1, Jan. 1948, pp. 55-63.

[Ercegovac04] M. Ercegovac and T. Lang, *Digital Arithmetic*, San Francisco: Morgan Kaufmann, 2004.

[Estreich82] D. Estreich and R. Dutton, "Modeling latch-up in CMOS integrated circuits," *IEEE Trans. Computer-Aided Design*, vol. CAD-1, no. 4, Oct. 1982, pp. 157-162.

[Faggin96] F. Faggin, M. Hoff, S. Mazor, and M. Shima, "The history of the 4004," *IEEE Micro*, vol. 16, no. 6, Dec. 1996, pp. 10-20.

[Fahim02] A. Fahim and M. Elmasry, "Low-power high-performance arithmetic circuits and architectures," *JSSC*, vol. 37, no. 1, Jan. 2002, pp. 90-94.

[Fetzer02] E. Fetzer, M. Gibson, A. Klein, N. Calick, C. Zhu, E. Busta, and B. Mohammad, "A fully bypassed six-issue integer datapath and register file on the Itanium-2 microprocessor," *JSSC*, vol. 37, no. 11, Nov. 2002, pp. 1433-1440.

[Flannagan85] S. Flannagan, "Synchronization reliability in CMOS technology," *JSSC*, vol. SC-20, no. 4, Aug. 1985, pp. 880-882.

[Flynn01] M. Flynn and S. Oberman, *Advanced Computer Arithmetic Design*, New York: John Wiley & Sons, 2001.

[Foty96] D. Foty, *MOSFET Modeling with SPICE: Principles and Practices*, Upper Saddle River, NJ: Prentice Hall, 1996.

[Friedman84] V. Friedman and S. Liu, "Dynamic logic CMOS circuits," *JSSC*, vol. SC-19, no. 2, April 1984, pp. 263-266.

[Frohman69] D. Frohman-Bentchkowsky and A. Grove, "Conductance of MOS transistors in saturation," *IEEE Trans. Electron Devices*, vol. ED-16, no. 1, Jan. 1969, pp. 108-113.

[Frowerk77] R. Frowerk, "Signature Analysis: A New Digital Field Service Method," *Hewlett Packard Journal*, May 1977, pp. 2-8.

[Gajski83] D. Gajski and R. Kuhn, "New VLSI tools," *Computer*, vol. 16, no. 12, Dec. 1983, pp. 11-14.

[Galiay80] J. Galiay, Y. Crouzet, and M. Verginiault, "Physical versus logical fault models MOS LSI circuits: impact on their testability," *IEEE Trans. Computers*, vol. C-29, no. 6, June 1980, pp. 527-531.

[Gauthier02] C. Gauthier and B. Amick, "Inductance: Implications and solutions for high-speed digital circuits: the chip electrical interface," *Proc. IEEE Intl. Solid-State Circuits Conf.*, vol.2, 2002, pp. 565-565.

[Geannopoulos98] G. Geannopoulos and X. Dai, "An adaptive digital deskewing circuit for clock distribution networks," *Proc. IEEE Intl. Solid-State Circuits Conf.*, 1998, pp. 400-401.

[Gelsinger01] P. Gelsinger, "Microprocessors for the new millennium: challenges, opportunities, and new frontiers," *Proc. IEEE Intl. Solid-State Circuits Conf.*, 2001, pp. 22-25.

[George96] S. George, A. Ott, and J. Klaus, "Surface chemistry for atomic layer growth," *J. Phys. Chem.*, vol. 100, 1996, pp. 13121-13131.

[Gerosa94] G. Gerosa et al., "A 2.2 W, 80 MHz superscalar RISC microprocessor," *JSSC*, vol. 29, no. 12, Dec. 1994, pp. 1440-1452.

[Gielis91] G. Gielis, R. van de Plassche, and J. van Valburg, "A 540-MHz 10-b polar-to-cartesian converter," *JSSC*, vol. 26, no. 11, Nov. 1991, pp. 1645-1650.

[Gieseke97] B. Gieseke et al., "A 600-MHz superscalar RISC microprocessor with out-of-order execution," *Proc. IEEE Intl. Solid-State Circuits Conf.*, 1997, pp. 176-177, 451.

[Glasser85] L. Glasser and D. Dobberpuhl, *The Design and Analysis of VLSI Circuits*, Reading, MA: Addison Wesley, 1985.

[Golden99] M. Golden et al., "A seventh-generation x86 microprocessor," *JSSC*, vol. 34, no. 11, Nov. 1999, pp. 1466-1477.

[Golomb81] S. Golomb, *Shift Register Sequences*, Revised Edition, Laguna Hills, CA: Aegean Park Press, 1981.

[Gonclaves83] N. Gonclaves and H. DeMan, "NORA: a racefree dynamic CMOS technique for pipelined logic structures," *JSSC*, vol. SC-18, no. 3, June 1983, pp. 261-266.

[Gonzalez96] R. Gonzalez and M. Horowitz, "Energy dissipation in general purpose microprocessors," *JSSC*, vol. 31, no. 9, Sept. 1996, pp. 1277-1284.

[Gray53] F. Gray, "Pulse code communications," US Patent 2,632,058, 1953.

[Gray01] P. Gray, P. Hurst, S. Lewis, and R. Meyer, *Analysis and Design of Analog Integrated Circuits*, 4th ed., New York: John Wiley & Sons, 2001.

[Grayver98] E. Grayver and B. Daneshrad, "Direct digital frequency synthesis using a modified CORDIC," *Proc. IEEE Intl. Symp. Circuits and Systems*, May 1998, pp. 241-244.

[Greenhill02] D. Greenhill, Design for Reliability Tutorial, *Proc. IEEE Intl. Solid-State Circuits Conf.*, 2002.

[Griffin83] W. Griffin and J. Hiltebeitel, "CMOS 4-way XOR circuit," *IBM Technical Disclosure Bulletin*, vol. 25, no. 11B, April 1983, pp. 6066-6067.

[Gronowski96] P. Gronowski et al., "A 433-MHz 64-b quad-issue RISC microprocessor," *JSSC*, vol. 31, no. 11, Nov. 1996, pp. 1687-1696.

[Gronowski98] P. Gronowski, W. Bowhill, R. Preston, M. Gowan, and R. Allmon, "High-performance microprocessor design," *JSSC*, vol. 33, no. 5, May 1998, pp. 676-686.

[Grosspietsch92] K. Grosspietsch, "Associative processors and memories: a survey," *IEEE Micro*, vol. 12, no. 3, June 1992, pp. 12-19.

[Grotjohn86] T. Grotjohn and B. Hoefflinger, "Sample-set differntial logic (SSDL) for complex high-speed VLSI," *JSSC*, vol. SC-21, no. 2, April 1986, pp. 367-369.

[Gupta03] S. Gupta and V. Fong, "A 64-MHz clock-rate sigma-delta ADC with 88-dB SNDR and −105-dB IM3 distortion at a 1.5-MHz signal frequency," *JSSC*, vol. 37, no. 12, Dec. 2002, pp. 1653-1661.

[Gutierrez01] E. Gutierrez, J. Deen, and C. Claeys (eds.), *Low Temperature Electronics: Physics, Devices, Circuits, and Applications*, New York: Academic Press, 2001.

[Gutnik00] V. Gutnik and A. Chandrakasan, "Active GHz clock network using distributed PLLs," *JSSC*, vol. 35, no. 11, Nov. 2000, pp. 1553-1560.

[Guyot87] A. Guyot, B. Hochet, and J. Muller, "A way to build efficient carry-skip adders," *IEEE Trans. Computers*, vol. 36, no. 10, Oct. 1987, pp. 1144-1152.

[Guyot97] A. Guyot and S. Abou-Samra, "Modeling power consumption in arithmetic operators," *Microelectronic Engineering*, vol. 39, 1997, pp. 245-253.

[Hamming50] R. Hamming, "Error detecting and error correcting codes," *Bell Systems Technical Journal*, vol. 29, pp. 147-160.

[Hamzaoglu02] F. Hamzaoglu and M. Stan, "Circuit-level techniques to control gate leakage for sub-100nm CMOS," *Proc. Intl. Symp. Low Power Electronics and Design*, 2002, pp. 60-63.

[Han87] T. Han and D. Carlson, "Fast area-efficient VLSI adders," *Proc. IEEE Symp. Computer Arithmetic*, 1987, pp. 49-56.

[Harame01a] D. Harame and B. Meyerson, "The early history of IBM's SiGe mixed signal technology," *IEEE Transactions on Electron Devices*, vol. 48, no. 11, Nov. 2001, pp. 2555-2567.

[Harame01b] D. Harame et al., "Current status and future trends of SiGe BiCMOS technology," *IEEE Transactions on Electron Devices*, vol. 48, no. 11, Nov. 2001, pp. 2575-2594.

[Haring96] R. Haring et al., "Self-resetting logic register and incrementer," *Symp. VLSI Circuits Digest Tech. Papers*, 1996, pp. 18-19.

[Harrer02] H. Harrer et al., "First and second-level packaging for the IBM eServer z900," *IBM J. Research and Development*, vol. 46, no. 4/5, July/Sept. 2002, pp. 397-420.

[Harris97] D. Harris and M. Horowitz, "Skew-tolerant domino circuits," *JSSC*, vol. 32, no. 11, Nov. 1997, pp. 1702-1711.

[Harris99] D. Harris, M. Horowitz, and D. Liu, "Timing analysis including clock skew," *IEEE Trans. Computer-Aided Design*, vol. 18, no. 11, Nov. 1999, pp. 1608-1618.

[Harris01a] D. Harris, *Skew-Tolerant Circuit Design*, San Francisco, CA: Morgan Kaufmann, 2001.

[Harris01b] D. Harris and S. Naffziger, "Statistical clock skew modeling with data delay variations," *IEEE Trans. VLSI*, vol. 9, no. 6, Dec. 2001, pp. 888-898.

[Harris03] D. Harris, "A taxonomy of prefix networks," *Proc. 37th Asilomar Conf. Signals, Systems, and Computers*, 2003, pp. 2213-2217.

[Harrison03] J. Harrison and N. Weste, "A 500 MHz CMOS anti-alias filter using feed-forward op-amps with local common-mode feedback," *Proc. IEEE Intl. Solid-State Circuits Conf.*, Feb. 2003, pp. 132-133.

[Hashemian92] R. Hashemian and C. Chen, "A new parallel technique for design of decrement/increment and two's complement circuits," *Proc. IEEE Midwest Symp. Circuits and Systems*, vol. 2, 1992, pp. 887-890.

[Hashimoto02] T. Hashimoto et al., "Integration of a 0.13-μm CMOS and a high performance self-aligned SiGe HBT featuring low base resistance," *Proc. Intl. Electron Devices Meeting*, Dec. 2002, pp. 779-782.

[Hatamian86] M. Hatamian and G. Cash, "A 70-MHz 8-bit x 8-bit parallel pipelined multiplier in 2.5-μm CMOS," *JSSC*, vol. 21, no. 4, Aug. 1986, pp. 505-513.

[Haykin00] S. Haykin, *Digital Communications*, New York: John Wiley & Sons, 2000.

[Hazucha00] P. Hazucha, C. Svensson, and S. Wender, "Cosmic-ray soft error rate characterization of a standard 0.6-μm CMOS process," *JSSC*, vol. 35, no. 10, Oct. 2000, pp. 1422-1429.

[Heald93] R. Heald and J. Holst, "A 6-ns cycle 256 kb cache memory and memory management unit," *JSSC*, vol. 28, no. 11, Nov. 1993, pp. 1078-1083.

[Heald98] R. Heald et al., "64-Kbyte sum-addressed-memory cache with 1.6-ns cycle and 2.6-ns latency," *JSSC*, vol. 33, no. 11, Nov. 1998, pp. 1682-1689.

[Heald00] R. Heald et al., "A third-generation SPARC v9 64-b microprocessor," *JSSC*, vol. 35, no. 11, Nov. 2000, pp. 1526-1538.

[Hedenstierna87] N. Hedenstierna and K. Jeppson, "CMOS circuit speed and buffer optimization," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, no. 2, March 1987, pp. 270-281.

[Heikes94] C. Heikes, "A 4.5mm^2 multiplier array for a 200MFLOP pipelined coprocessor," *Proc. IEEE Intl. Solid-State Circuits Conf.*, 1994, pp. 290-291.

[Heller84] L. Heller, W. Griffin, J. Davis and N. Thoma, "Cascode voltage switch logic: a differential CMOS logic family," *Proc. IEEE Intl. Solid-State Circuits Conf.*, 1984, pp. 16-17.

[Hennessy90] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, San Mateo, CA: Morgan Kaufmann Publishers, Inc. 1990.

[Hess94] C. Hess and L. Weiland, "Drop in process control checkerboard test structure for efficient online process characterization and defect problem debugging," *Proc. IEEE Int. Conf. Microelectronic Test Structures*, vol. 7, March, 1994, pp. 152-159.

[Hidaka89] H. Hidaka, K. Fujishima, Y. Matsuda, M. Asakura, and T. Yoshihara, "Twisted bit-line architectures for multi-megabit DRAM's," *JSSC*, vol. 24, no. 1, Feb. 1989, pp. 21-27.

[Hill68] C. Hill, "Noise margin and noise immunity in logic circuits," *Microelectronics*, vol. 1, April 1968, pp. 16-21.

[Hinton01] G. Hinton et al., "A 0.18-μm CMOS IA-32 processor with a 4-GHz integer execution unit," *JSSC*, vol. 36, no. 11, Nov. 2001, pp. 1617-1627.

[Hisamoto98] D. Hisamoto et al., "A folded-channel MOSFET for deep-sub-tenth micron era," *Tech. Digest Intl. Electron Devices Meeting*, San Francisco, Dec. 1998, pp. 1032-1034..

[Ho98] R. Ho, B. Amrutur, K. Mai, B. Wilburn, T. Mori, and M. Horowitz, "Application of on-chip samplers for test and measurement of integrated circuits," *Symp. VLSI Circuits Digest Tech. Papers*, 1998, pp. 138-139.

[Ho01] R. Ho, K. Mai, and M. Horowitz, "The future of wires," *Proc. IEEE*, vol. 89, no. 4, April 2001, pp. 490-504.

[Ho03a] R. Ho, K. Mai, and M. Horowitz, "Efficient on-chip global interconnects," *Symp. VLSI Circuits Digest Tech. Papers*, 2003, pp. 271-274.

[Ho03b] R. Ho, K. Mai, and M. Horowitz, "Managing wire scaling: a circuit perspective," *Proc. IEEE Interconnect Technology Conf.*, 2003, pp. 177-179.

[Hoeneisen72] B. Hoeneisen and C. Mead, "Fundamental limitations in Microelectronics-I. MOS technology," *Solid-State Electronics*, vol. 15, 1972, pp. 819-829.

[Hoeschele94] D. Hoeschele, *Analog-to-Digital and Digital-to-Analog Conversion Techniques*, New York: Wiley-Interscience, 1994.

[Hoppe90] B. Hoppe, G. Neuendorf, D. Schmitt-Landsiedel, and W. Specks, "Optimization of high-speed CMOS logic circuits with analytical models for signal delay, chip area, and dynamic power dissipation," *IEEE Trans. Computer-Aided Design*, vol. 9, no. 3, March 1990, pp. 236-247.

[Horowitz83] M. Horowitz and R. Dutton, "Resistance extraction from mask layout data," *IEEE Trans. Computer-Aided Design*, vol. CAD-2, no. 3, July 1983, pp. 145-150.

[Horowitz87] M. Horowitz et al., "MIPS-X: a 20-MIPS peak, 32-bit microprocessor with on-chip cache," *JSSC*, vol. SC-22, no. 5, Oct. 1987, pp. 790-799.

[Horowitz02] M. Horowitz, EE371 Course Notes, Stanford University, Spring 2002, www.stanford.edu/class/ee371

[Horstmann89] J. Horstmann, H. Eichel, and R. Coates, "Metastability behavior of CMOS ASIC flip-flops in theory and test," *JSSC*, vol. 24, no. 1, Feb. 1989, pp. 146-157.

[Hrishikesh02] M. Hrishikesh et al., "The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays," *Proc. Intl. Symp. Computer Architecture*, 2002, pp. 14-24.

[Hsu91] W. Hsu, B. Sheu, and S. Gowda, "Design of reliable VLSI circuits using simulation techniques," *JSSC*, vol. 26, no. 3, March 1991, pp. 452-457.

[Hsu92] W. Hsu, B. Sheu, S. Gowda and C. Hwang, "Advanced integrated-circuit reliability simulation including dynamic stress effects," *JSSC*, vol. 27, no. 3, March 1992, pp. 247-257.

[Hu90] Y. Hu and S. Chen, "GM_Plan: A Gate Matrix Layout Algorithm Based on Artificial Intelligence Planning Techniques," *IEEE Trans. Computer-Aided Design*, vol. 9, no. 8, Aug. 1990, pp. 836-845.

[Hu92] C. Hu, "IC reliability simulation," *JSSC*, vol. 27, no. 3, March 1992, pp. 241-246.

[Hu95] C. Hu, K. Rodbell, T. Sullivan, K. Lee, and D. Bouldin, "Electromigration and stress-induced voiding in fine Al and Al-alloy thin-film lines," *IBM J. Research and Development*, vol. 39, no. 4, July 1995, pp. 465-497.

[Huang00] Z. Huang and M. Ercegovac, "Effect of wire delay on the design of prefix adders in deep-submicron technology," *Proc. 34th Asilomar Conf. Signals, Systems, and Computers*, vol. 2, 2000, pp. 1713-1717.

[Huang02] C. Huang, J. Wang and Y. Huang, "Design of high-performance CMOS priority encoders and incrementer/decrementers using multilevel lookahead and multilevel folding techniques," *JSSC*, vol. 37, no. 1, Jan. 2002, pp. 63-76.

[Huang03] X. Huang et al., "Loop-based interconnect modeling and optimization approach for multigigahertz clock network design," *JSSC*, vol. 38, no. 3, March 2003, pp. 457-463.

[Huh98] Y. Huh, Y. Sung, and S. Kang, "A study of hot-carrier-induced mismatch drift: a reliability issue for VLSI circuits," *JSSC*, vol. 33, no. 6, June 1998, pp. 921-927.

[Huitema03] E. Huitema et al., "Plastic transistors in active-matrix displays," *Proc. IEEE Intl. Solid-State Circuits Conf.*, Feb. 2003, pp. 380-381.

[Hwang89] I. Hwang and A. Fisher, "Ultrafast compact 32-bit CMOS adders in multiple-output domino logic," *JSSC*, vol. 24, no. 2, April 1989, pp. 358-369.

[Hwang99a] W. Hwang, R. Joshi, and W. Henkels, "A 500-MHz, 32-Word x 64-bit, eight-port self-resetting CMOS register file," *JSSC*, vol. 34, no. 1, Jan. 1999, pp. 56-67.

[Hwang99b] W. Hwang, G. Gristede, P. Sanda, S. Wang, and D. Heidel, "Implementation of a self-resetting CMOS 64-bit parallel adder with enhanced testability," *JSSC*, vol. 34, no. 8, Aug. 1999, pp. 1108-1117.

[Hwang02] D. Hwang, F. Dengwei, A. Willson, Jr, "A 400-MHz processor for the efficient conversion of rectangular to polar coordinates for digital communications applications," *Symp. VLSI Circuits Digest Tech. Papers*, June 2002, pp. 248-51.

[IEEE1076-02] *IEEE Standard 1076-2002, VHDL Language Reference Manual.*

[IEEE1149.1-01] *IEEE Standard 1149.1-2001, Test Access Port and Boundary-Scan Architecture.*

[IEEE1164-93] *IEEE Standard 1164-1993, Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164).*

[IEEE1364-01] *IEEE Standard 1364-2001, Verilog Hardware Description Language.*

[Ingino01] J. Ingino and V. von Kaenel, "A 4-GHz clock system for a high-performance system-on-a-chip design," *JSSC*, vol. 36, no. 11, Nov. 2001, pp. 1693-1698.

[Intel03] Intel Corporation, *Microprocessor Quick Reference Guide*, http://www.intel.com/pressroom/kits/quickreffam.htm, 2003.

[Ismail99] Y. Ismail, E. Friedman, and J. Neves, "Figures of merit to characterize the importance of on-chip interconnect," *IEEE Trans. VLSI*, vol. 7, no. 4, Dec. 1999, pp. 442-449.

[Itoh01k] K. Itoh, *VLSI Memory Chip Design*, Berlin: Springer-Verlag, 2001.

[Itoh01n] N. Itoh et al., "A 600-MHz 54 x 54-bit multiplier with rectangular-styled Wallace tree," *JSSC*, vol. 36, no. 2, Feb. 2001, pp. 249-257.

[Jamal02] S. Jamal et al., "A 10-b 120-Msample/s time-interleaved analog-to-digital converter with digital background calibration," *JSSC*, vol. 37, no. 12, Dec. 2002, pp. 1618-1627.

[Jayasumana91] A. Jayasumana, Y. Malaiya, and R. Rajsuman," Design of CMOS circuits for stuck-open fault testability," *JSSC*, vol. 26, no. 1, Jan. 1991, pp. 58-61.

[Jiang03] X. Jiang, Z. Wang, and F. Chang, "A 2 GS/s 6 b ADC in 0.18 μm CMOS," *Proc. IEEE Intl. Solid-State Circuits Conf.*, Feb. 2003, pp.322-323.

[Ji-ren87] Y. Ji-ren, I. Karlsson, and C. Svensson, "A true single-phase-clock dynamic CMOS circuit technique," *JSSC*, vol. SC-22, no. 5, Oct. 1987, pp. 899-901.

[Johns96] D. Johns and K. Martin, *Analog Integrated Circuit Design*, New York: John Wiley & Sons, 1996.

[Johnson88] M. Johnson, "A symmetric CMOS NOR gate for high-speed applications," *JSSC*, vol. SC-23, no. 5, Oct. 1988, pp. 1233-1236.

[Johnson91] B. Johnson, T. Quarles, A. Newton, D. Pederson, A. Sangiovanni-Vincentelli, *SPICE3 Version 3e User's Manual*, UC Berkeley, April 1991.

[Johnson93] H. Johnson and M. Graham, *High-Speed Digital Design: A Handbook of Black Magic*, Upper Saddle River, NJ: Prentice Hall, 1993.

[Josephson02] D. Josephson, "The manic depression of microprocessor debug," *Proc. Intl. Test Conf.*, 2002, pp. 657-663.

[Juhnke95] T. Juhnke and H. Klar, "Calculation of the soft error rate of submicron CMOS logic circuits," *JSSC*, vol. 30, no. 7, July 1995, pp. 830-834.

[Jung01] S. Jung, S. Yoo, K. Kim, and S. Kang, "Skew-tolerant high-speed (STHS) domino logic," *Proc. IEEE Intl. Symp. Circuits and Systems*, 2001, pp. 154-157.

[Kamon94] M. Kamon, J. Tsuk, and J. White, "FASTHENRY: a multipole-accelerated 3-D inductance extraction program," *IEEE Trans. Microwave Theory and Techniques*, vol. 42, no. 9, Sept. 1994, pp. 1750-1758.

[Kang03] S. Kang and Y. Leblebici, *CMOS Digital Integrated Circuits*, 3rd ed., Boston: McGraw Hill, 2003.

[Kantabutra91] V. Kantabutra, "Designing optimum carry-skip adders," *Proc. IEEE Symp. Computer Arithmetic*, 1991, pp. 146-153.

[Kantabutra93] V. Kantabutra, "A recursive carry-lookahead / carry-select hybrid adder," *IEEE Trans. Computers*, vol. 42, no. 12, Dec. 1993, pp. 1495-1499.

[Kao01] J. Kao and A. Chandrasakan, "MTCMOS sequential circuits," *Proc. 27th European Solid-State Circuits Conf.*, 2001, pp. 332-335.

[Kappes03] M. Kappes, "A 2.2-mW CMOS bandpass continuous-time delta-sigma ADC with 68 dB of dynamic range and 1-MHz bandwidth for wireless applications," *JSSC*, vol. 38, no. 7, July 2003, pp. 1098-1104.

[Karnik01] T. Karnik, B. Bloechel, K. Soumyanath, V. De, and S. Borkar, "Scaling trends of cosmic rays induced soft errors in static latches beyond 0.18m," *Symp. VLSI Circuits Digest Tech. Papers*, 2001, pp. 61-62.

[Keeth01] B. Keeth and J. Baker, *DRAM Circuit Design: A Tutorial*, Piscataway, NJ: IEEE Press, 2001.

[Keshavarzi01] A. Keshavarzi et al., "Effectivness of reverse body bias for leakage control in scaled dual Vt CMOS ICs," *Proc. Intl. Symp. Low Power Electronics and Design*, 2001, pp. 207-212.

[Keyes70] R. Keyes, E. Harris, and K. Konnerth, "The role of low temperatures in the operation of logic circuitry," *Proc. IEEE*, vol. 58, no. 12, Dec. 1970, pp. 1914-1932.

[Kielkowski95] R. Kielkowski, SPICE: *Practical Device Modeling*, Boston: McGraw-Hill, 1995.

[Kilburn59] T. Kilburn, D. Edwards, and D. Aspinall, "Parallel addition in a digital computer - a new fast carry," *Proc. IEE*, vol. 106B, 1959, pp. 460-464.

[Kim02] Y. Kim et al., "50 nm gate length logic technology with 9-layer Cu interconnects for 90 nm node SoC applications," *Proc. Intl. Electron Devices Meeting*, 2002, p. 69.

[Kio01] S. Kio, L. McMurchie, and C. Sechen, "Application of output prediction logic to differential CMOS," *Proc. IEEE Computer Society Workshop on VLSI*, 2001, pp. 57-65.

[Klass99] F. Klass et al., "A new family of semidynamic and dynamic flip-flops with embedded logic for high-performance processors," *JSSC*, vol. 34, no. 5, May 1999, pp. 712-716.

[Klaus98] J. Klaus, A. Ott, A. Dillon, and S. George, "Atomic layer controlled growth of Si_3N_4 films using sequential surface reactions," *Surf. Sci.*, vol. 418, 1998, pp. L14-L19.

[Knebel98] D. Knebel et al., "Diagnosis and characterization of timing-related defects by time-dependent light emission," *IEEE Intl. Test Conf.*, 1998, pp. 733-739.

[Knowles01] S. Knowles, "A family of adders," *Proc. IEEE Symp. Computer Arithmetic*, 2001, pp. 277-284.

[Koenemann79] B. Koenemann, J. Mucha, and G. Zwiehoff, "Built-in logic block observation techniques," *Proc. Intl. Test Conf.*, Oct. 1979, pp. 37-41.

[Kogge73] P. Kogge and H. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Trans. Computers*, vol. C-22, no. 8, Aug. 1973, pp. 786-793.

[Kong01w] W. Kong, R. Venkatraman, R. Castagnetti, F. Duan, and S. Ramesh, "High-density and high-performance 6T-SRAM for system-on-chip in 130 nm CMOS technology," *Tech. Digest Symp. VLSI Technology*, 2001, pp. 105-106.

[Koren02] I. Koren, *Computer Arithmetic Algorithms*, 2nd ed., Natick, Mass.: A.K. Peters, 2002.

[Kovacs98] G. Kovacs, *Micromachined Transducers Sourcebook*, Boston: McGraw-Hill, 1998.

[Kozu96] S. Kozu et al., "A 100 MHz 0.4W RISC processor with 200 MHz multiply-adder, using pulse-register technique," *Proc. IEEE Intl. Solid-State Circuits Conf.*, 1996, pp. 140-141.

[Krambeck82] R. Krambeck, C. Lee, and H. Law, "High speed compact circuits with CMOS," *JSSC*, vol. SC-17, no. 3, June 1982, pp. 614-619.

[Kumar94] R. Kumar, "ACMOS: an adaptive CMOS high performance logic," *Electronics Letters*, vol. 30, no. 6, March 1994, pp. 483-484.

[Kumar01] R. Kumar, "Interconnect and noise immunity design for the Pentium 4 processor," *Intel Technology Journal*, vol. 5, no. 1, Q1 2001, pp. 1-12.

[Kuo01] J. Kuo and S. Lin, *Low-Voltage SOI CMOS VLSI Devices and Circuits*, New York: Wiley Interscience, 2001.

[Kurd01] N. Kurd, J. Barkatullah, R. Dizon, T. Fletcher, and P. Madland, "A multigigahertz clocking scheme for the Pentium 4 microprocessor," *JSSC*, vol. 36, no. 11, Nov. 2001, pp. 1647-1653.

[Kuroda96] T. Kuroda et al., "A 0.9-V, 150-MHz, 10-mW, 4 mm^2, 2-D discrete cosine transform core processor with variable threshold-voltage (VT) scheme," *JSSC*, vol. 31, no. 11, Nov. 1996, pp. 1770-1779.

[Ladner80] R. Ladner and M. Fischer, "Parallel prefix computation," *J. ACM*, vol. 27, no. 4, Oct. 1980, pp. 831-838.

[Lai97] F. Lai and W. Hwang, "Design and implementation of differential cascode voltage switch with pass-gate (DCVSPG) logic for high-performance digital systems," *JSSC*, vol. 32, no. 4, April 1997, pp. 563-573.

[Lakshmanan01] A. Lakshmanan and R. Sridhar, "Input controlled refresh for noise tolerant dynamic circuits," *Proc. 14th IEEE Intl. ASIC/SOC Conf.*, 2001, pp. 129-133.

[Larsson94] P. Larsson and C. Svensson, "Impact of clock slope on true single phase clocked (TSPC) CMOS circuits," *JSSC*, vol. 29, no. 6, June 1994, pp. 723-726.

[Larsson97] P. Larsson, "Parasitic resistance in an MOS transistor used as on-chip decoupling capactance," *JSSC*, vol. 32, no. 4, April 1997, pp. 574-576.

[Lasserre99] F. Lasserre et al., "Laser beam backside probing of CMOS integrated circuits," *Microelectronics and Reliability*, June 1999, vol. 39, no. 6, pp. 957-961.

[Leblebici96] Y. Leblebici, "Design considerations for CMOS digital circuits with improved hot-carrier reliability," *JSSC*, vol. 31, no. 7, July 1996, pp. 1014-1024.

[Lee02] S. Lee and H. Yoo, "Race logic architecture (RALA): a novel logic concept using the race scheme of input variables," *JSSC*, vol. 37, no. 2, Feb. 2002, pp. 191-201.

[Lee04] T. Lee, *The Design of CMOS Radio-Frequency Integrated Circuits*, 2nd ed., Cambridge: Cambridge University Press, 2004.

[Lee86] C. Lee and E. Szeto, "Zipper CMOS," *IEEE Circuits and Systems Magazine*, May 1986, pp. 10-16.

[Lee92] K. Lee and M. Breuer, "Design and test rules for CMOS circuits to facilitate IDDQ testing of bridging faults," *IEEE Trans. On CAD of Integrated circuits*, vol. 11, no. 5, May 1992, pp. 659-670.

[Lee98] M. Lee, "A multilevel parasitic interconnect capacitance modeling and extraction for reliable VLSI on-chip clock delay evaluation," *JSSC*, vol. 33, no. 4, April 1998, pp. 657-661.

[Lehman61] M. Lehman and N. Burla, "Skip technique for high-speed carry-propagation in binary arithmetic units," *IRE Trans. Electronic Computers*, vol. 10, Dec. 1961, pp. 691-698.

[Leighton92] F. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays; Trees; Hypercubes*, San Francisco: Morgan Kaufmann, 1992.

[Leung02] B. Leung, *VLSI for Wireless Communication*, Upper Saddle River, NJ: Prentice Hall, 2002.

[Lev95] L. Lev et al., "A 64-b microprocessor with multimedia support," *JSSC*, vol. 30, no. 11, Nov. 1995, pp. 1227-1238.

[Li03] Y. Li and E. Sanchez-Sinencio, "A wide input bandwidth 7-bit 300-Msample/s folding and current-mode interpolating ADC," *JSSC*, vol. 38, no. 8, Aug. 2003, pp.1405-1410.

[Liew90] B. Liew, N. Cheung, and C. Hu, "Projecting interconnect electromigration lifetime for arbitrary current waveforms," *IEEE Trans. Electron Devices*, vol. 37, no. 5, May 1990, pp. 1343-1351.

[Lim72] R. Lim, "A barrel switch design," *Computer Design*, Aug. 1972, pp. 76-78.

[Lin83] S. Lin and D. Costello, *Error Control Coding: Fundamentals and Applications*, Upper Saddle River, NJ: Prentice Hall, 1983.

[Lin91] Y. Lin, B. Kim and P. Gray, "A 13-b 2.5-MHz self-calibrated pipelined A/D converter in 3-μm CMOS," *JSSC*, vol. 26, no. 4, April 1991, pp. 628-636.

[Ling81] H. Ling, "High-speed binary adder," *IBM J. Research and Development*, vol. 25, no. 3, May 1981, pp. 156-166.

[Liu01] X. Liu, C. Lee, C. Zhou, and J. Han, "Carbon nanotube field-effect inverters," *Appl. Phys. Letters*, vol. 79, no. 20, Nov. 2001, pp. 3329-3331

[Lohstroh79] J. Lohstroh, "Static and dynamic noise margins of logic circuits," *JSSC*, vol. SC-14, no. 3, June 1979, pp. 591-598.

[Lohstroh83] J. Lohstroh, E. Seevinck, and J. de Groot, "Worst-case static noise margin criteria for logic circuits and their mathematical equivalence," *JSSC*, vol. SC-18, no. 6, Dec. 1983, pp. 803-807.

[Lovett98] S. Lovett, M. Welten, A. Mathewson, and B. Mason, "Optimizing MOS transistor mismatch," *JSSC*, vol. 33, no. 1, Jan. 1998, pp. 147-150.

[Lu88b] S. Lu, "Implementation of iterative networks with CMOS differential logic," *JSSC*, vol. 23, no. 4, Aug. 1988, pp. 1013-1017.

[Lu91] S. Lu and M. Ercegovac, "Evaluation of two-summand adders implemented in ECDL CMOS differential logic," *JSSC*, vol. 26, no. 8, Aug. 1991, pp. 1152-1160.

[Lu93] F. Lu, H. Samueli, J.Yuan, C. Svensson, "A 700 MHz 24-b pipelined accumulator in 1.2-μm CMOS for application as a numerically controlled oscillator," *JSSC*, vol. 28, no. 8, Aug 1993, pp. 878-886.

[Lu93b] F. Lu and H. Samueli, "A 200-MHz CMOS pipelined multiplier-accumulator using a quasi-domino dynamic full-adder cell design," *JSSC*, vol. 28, no. 2, Feb. 1993, pp. 123-132.

[Lynch91] T. Lynch and E. Swartzlander, "The redundant cell adder," *Proc. IEEE Symp. Computer Arithmetic*, 1991, pp. 165-170.

[Lynch92] T. Lynch and E. Swartzlander, "A spanning tree carry lookahead adder," *IEEE Trans. Computers*, vol. 41, no. 8, Aug. 1992, pp. 931-939.

[Lyon87] R. Lyon and R. Schediwy, "CMOS static memory with a new four-transistor memory cell," *Proc. Advanced Research in VLSI*, March 1987, pp. 111-132

[Ma98] T. Ma, "Making silicon nitride film a viable gate dielectric," *IEEE Trans. Electron Devices*, vol. 45, no. 3, March 1998, pp. 680-690.

[MacSorley61] O. MacSorley, "High-speed arithmetic in binary computers," *Proc. IRE*, vol. 49, part 1, Jan. 1961, pp. 67-91.

[Mahalingam85] M. Mahalingam, "Thermal management in semiconductor device packages," *Proc. IEEE Custom Integrated Circuits Conf.*, 1985, pp. 46-49.

[Maier97] C. Maier et al., "A 533-MHz BiCMOS superscalar RISC microprocessor," *JSSC*, vol. 32, no. 11, Nov. 1997, pp. 1625-1634.

[Majerski67] S. Majerski, "On determination of optimal distributions of carry skips in adders," *IEEE Trans. Electronic Computers*, vol. EC-16, no. 1, 1967, pp. 45-58.

[Maluf00] N. Maluf, *An Introduction to Microelectromechanical Systems Engineering*, Norwood, MA: ArtechHouse, 2000.

[Maneatis96] J. Maneatis, "Low-jitter process-independent DLL and PLL based on self-biased techniques," *JSSC*, vol. 31, no. 11, Nov. 1996, pp. 1723-1732.

[Maneatis03] J. Maneatis, I. McClatchie, J. Maxey, and M. Shankaradas, "Self-biased high-bandwidth low-jitter 1-to-4096 multiplier clock generator PLL," *JSSC*, vol. 38, no. 11, Nov. 2003, pp. 1795-1803.

[Mathew03] S. Mathew, M. Anders, R. Krishnamurthy, and S. Borkar, "A 4-GHz 130-nm address generation unit with 32-bit sparse-tree adder core," *JSSC*, vol. 38, no. 5, May 2003, pp. 689-695.

[Matsui94] M. Matsui et al., "A 200 MHz 13 mm^2 2-D DCT macrocell using sense-amplifier pipeline flip-flop scheme," *JSSC*, vol. 29, no. 12, Dec. 1994, pp. 1482-1490.

[May79] T. May and M. Woods, "Alpha-particle-induced soft errors in dynamic memories," *IEEE Trans. Electron Devices*, vol. ED-26, no. 1, Jan. 1979, pp. 2-9.

[McMurchie00] L. McMurchie, S. Kio, G. Yee, T. Thorp, and C. Sechen, "Output prediction logic: a high-performance CMOS design technique," *Proc. Intl. Conf. Computer Design*, 2000, pp. 247-254.

[Mead80] C. Mead and L. Conway, *Introduction to VLSI Systems*, Reading, MA: Addison-Wesley, 1980.

[Mehta99] G. Mehta, D. Harris, and D. Singh, "Pulsed Domino Latches," US Patent 5,880,608, 1999.

[Meier99] N. Meier, T. Marieb, P. Flinn, R. Gleixner, and J. Bravman, "In-situ studies of electromigration voiding in passivated copper interconnects," *AIP Conf. Proc. 491*, Fifth Intl. Workshop on Stress-Induced Phenomena in Metallization, June 1999, p. 180.

[Meijs84] N. van der Meijs, J. Fokkema, "VLSI circuit reconstruction from mask topology," *Integration, The VLSI Journal, vol. 2*, no. 2, June 1984, pp. 85-119.

[Melly01] T. Melly, A. Porret, C. Enz, and E. Vittoz, "An analysis of flicker noise rejection in low-power and low-voltage CMOS mixers," *JSSC*, vol. 36, no. 1, Jan. 2001, pp. 102-109.

[Merchant01] S. Merchant, S. Kang, M. Sanganeria, B. van Schravendijk, and T. Mountsier, "Copper interconnects for semiconductor devices," *JOM: Journal of the Minerals, Metals, and Materials Society*, vol. 53, no. 6, June 2001, pp. 43-48.

[Messerschmitt90] D. Messerschmitt, "Synchronization in digital system design," *IEEE J. Selected Areas Communications*, vol. 8, no. 8, Oct. 1990, pp. 1404-1419.

[Mezhiba03] A. Mezhiba and E. Friedman, *Power Distribution Networks in High Speed Integrated Circuits*, Boston: Kluwer Academic Publishers, 2003.

[Miller00] R. Miller et al., "The development of 157nm small field and mid-field microsteppers," *Proc. SPIE*, vol. 4000, Optical Microlithography XIII, Christopher J. Progler ed., 2000, pp. 567-578.

[Miyatake01] H. Miyatake, M. Tanaka, and Y. Mori, "A design for high-speed low-power CMOS fully parallel content-addressable memory macros," *JSSC*, vol. 36, no. 6, June 2001, pp. 956-968.

[Mizuno94] T. Mizuno, J. Okumtura, and A. Toriumi, "Experimental study of threshold voltage fluctuation due to statistical variation of chanel dopant number in MOSFET's," *IEEE Trans. Electron Devices*, vol. 41, no. 11, Nov. 1994, pp. 2216-2221.

[Moazzami90] R. Moazzami and C. Hu, "Projecting gate oxide reliability and optimizing reliability screens," *IEEE Trans. Electron Devices*, vol. 37, no. 7, July 1990, pp. 1643-1650.

[Montanaro96] J. Montanaro et al., "A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor," *JSSC*, vol. 31, no. 11, Nov. 1996, pp. 1703-1714.

[Moore65] G. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, April 1965.

[Moore03] G. Moore, "No exponential is forever: but 'forever' can be delayed!" *Proc. IEEE Intl. Solid-State Circuits Conf.*, 2003, pp. 1-19.

[Morgan59] C. Morgan and D. Jarvis, "Transistor logic using current switching routing techniques and its application to a fast carry-propagation adder, *Proc. IEE*, vol. 106B, 1959, pp. 467-468.

[Morrison61] P. Morrison and E. Morrison, eds., *Charles Babbage: On the Principles and Development of the Calculator*, New York: Dover, 1961.

[Mortezapour00] S. Mortezapour and E. Lee, "A 1-V, 8-bit successive approximation ADC in standard CMOS process," *JSSC*, vol. 35, no. 4, April 2000, pp. 642-646.

[Morton99] S. Morton, "On-chip inductance issues in multiconductor systems," *Proc. Design Automation Conf.*, 1999, pp. 921-926.

[Morton02] S. Morton, "Inductance: Implications and solutions for high-speed digital circuits," *Proc. IEEE Intl. Solid-State Circuits Conf.*, vol. 2 , Feb 2002, pp. 554-557.

[Mou90] Z. Mou and F. Jutand, "A class of close-to-optimum adder trees allowing regular and compact layout," *Proc. IEEE Intl. Conf. Computer Design*, 1990, pp. 251-254.

[Mule02] A. Mule, E. Glytsis, T. Gaylord, and J. Meindl, "Electrical and optical clock distribution networks for gigascale microprocessors," *IEEE Trans. VLSI*, vol. 10, no. 5, Oct. 2002, pp. 582-594.

[Muller03] R. Muller, T. Kamins, and M. Chan, *Device Electronics for Integrated Circuits*, 3rd ed., New York: John Wiley & Sons, 2003.

[Murabayashi96] F. Murabayashi et al., "2.5 V CMOS circuit techniques for a 200 MHz superscalar RISC processor," *JSSC*, vol. 31, no. 7, July 1996, pp. 972-980.

[Murphy80] B. Murphy, "Unified field-effect transistor theory including velocity saturation," *JSSC*, vol. SC-15, no. 3, June 1980, pp. 325-327.

[Mutoh95] S. Mutoh et al., "1-V power supply high-speed digital circuit technology with multithreshold-voltage CMOS," *JSSC*, vol. 30, no. 8, Aug. 1995, pp. 847-854.

[Nabors92] K. Nabors, S. Kim, and J. White, "Fast capacitance extraction of general three-dimensional structures," *IEEE Trans. Microwave Theory and Techniques*, vol. 40, no. 7, July 1992, pp. 1496-1506.

[Nadig77] H. Nadig, "Signature analysis—concepts, examples and guidelines," *Hewlett Packard Journal*, vol. 28, no. 9, May 1977, pp. 15-21.

[Naffziger96] S. Naffziger, "A subnanosecond 0.5μm 64b adder design," *Proc. IEEE Intl. Solid-State Circuits Conf.*, 1996, pp. 362-363.

[Naffziger98] S. Naffziger, "High speed addition using Ling's equations and dynamic CMOS logic," US Patent 5,719,803, 1998.

[Naffziger02] S. Naffziger, G. Colon-Bonet, T. Fischer, R. Riedlinger, T. Sullivan, and T. Grutkowski, "The implemetnation of the Itanium 2 microprocessor," *JSSC*, vol. 37, no. 11, Nov. 2002, pp. 1448-1460.

[Nagel75] L. Nagel, *SPICE2: a computer program to simulate semiconductor circuits*, Memo ERL-M520, Dept. of Electrical Engineering and Computer Science, University of California at Berkeley, May 9, 1975.

[Nair78] R. Nair, S. Thatte, and J. Abraham, "Efficient algorithms for testing semiconductor random-access memories," *IEEE Trans. Computers*, vol. C-27, no. 6, June 1978, pp. 572-576.

[Nalamalpu02] A. Nalamalpu, S. Srinivasan, and W. Burleson, "Boosters for driving long onchip interconnects—design issues, interconnect synthesis, and comparison with repeaters," *IEEE Trans. Computer-Aided Design*, vol. 21, no. 1, Jan. 2002, pp. 50-62.

[Nambu98] H. Nambu et al., "A 1.8-ns access, 550-MHz, 4.5-Mb CMOS SRAM," *JSSC*, vol. 33, no. 11, Nov. 1998, pp. 1650-1658.

[Narayanan96] V. Narayanan, B. Chappell, and B. Fleischer, "Static timing analysis for self-resetting circuits," *Proc. Intl. Conf. Computer-Aided Design*, 1996, pp. 119-126.

[Narendra99] S. Narendra, D. Antoniadis, and V. De, "Impact of using adaptive body bias to compensate die-to-die Vt variation on within-die Vt variation," *Proc. Intl. Symp. Low Power Electronics and Design*, 1999, pp. 229-232.

[Narendra01] S. Narendra, S. Borkar, V. De, D. Antoniadis, and A. Chandrakasan, "Scaling of stack effect and its application for leakage reduction," *Proc. Intl. Symp. Low Power Electronics and Design*, 2001, pp. 195-200.

[Narendra03] S. Narendra, A. Keshavarzi, B. Bloechel, S. Borkar, and V. De, "Forward body bias for microprocessors in 130-nm technology generation and beyond," *JSSC*, vol. 38, no. 5, May 2003, pp. 696-701.

[Nauta95] B. Nauta and A. Venes, "A 70-MS/s 110-mW 8-b CMOS folding and interpolating A/D converter," *JSSC*, vol. 30, no. 12, Dec. 1995, pp. 1302-1308.

[Ng96] P. Ng, P. Balsara, and D. Steiss, "Performance of CMOS differential circuits," *JSSC*, vol. 31, no. 6, June 1996, pp. 841-846.

[Nikoli00] B. Nikolí, V. Oklobdíja, V. Stojanovi, W. Jia, J. Chiu, and M. Leung, "Improved sense-amplifier-based flip-flop: design and measurements," *JSSC*, vol. 35, no. 6, June 2000, pp. 876-884.

[Noice83] D. Noice, *A clocking discipline for two-phase digital integrated circuits*, Stanford University Technical Report, Jan. 1983.

[Nowak02] E. Nowak, "Maintaining the benefits of CMOS scaling when scaling bogs down," *IBM J. Research and Development*, vol. 46, no. 2/3, March/May 2002, pp. 169-180

[Nowka98] K. Nowka and T. Galambos, "Circuit design techniques for a gigahertz integer microprocessor," *Proc. Intl. Conf. Computer Design*, 1998, pp. 11-16.

[Ohkubo95] N. Ohkubo et al., "A 4.4 ns CMOS 54 x 54-b multiplier using pass-transistor multiplexer," *JSSC*, vol. 30, no. 3, March 1995, pp. 251-257.

[Oklobdíja85] V. Oklobdíja and E. Barnes, "Some optimal schemes for ALU implementation in VLSI technology," Proc. IEEE Symp. Comp. Arithmetic, 1985.

[Oklobdíja86] V. Oklobdíja and R. Montoye, "Design-performance trade-offs in CMOS-domino logic," *JSSC*, vol. SC-21, no. 2, April 1986, pp. 304-309.

[Ortiz-Conde02] A. Ortiz-Conde, F. Sánchez, J. Liou, A. Cerdeira, M. Estrada, and Y. Yue, "A review of recent MOSFET threshold voltage extraction methods," *Microelectronics Reliability*, vol. 42, 2002, pp. 583-596.

[Oshawa87] T. Oshawa et al., "A 60-ns 4-Mbit CMOS DRAM with built-in self-test function," *JSSC*, vol. SC-22, no. 5, Oct. 1987, pp. 663-668.

[Paik96] W. Paik, H. Ki, and S. Kim, "Push-pull pass-transistor logic family for low voltage and low power," *Proc. 22nd European Solid-State Circuits Conf.*, 1996, pp. 116-119.

[Parameswar96] A. Parameswar, H. Hara, and T. Sakurai, "A swing restored pass-transistor logic-based multiply and accumulate circuit for multimedia applications," *JSSC*, vol. 31, no. 6, June 1996, pp. 804-809.

[Paraskevopoulos87] D. Paraskevopoulos and C. Fey, "Studies in LSI technology economics III: design schedules for application-specific integrated circuits," *JSSC*, vol. SC-22, no. 2, April 1987, pp. 223-229.

[Parhami00] B. Parhami, *Computer Arithmetic Algorithms and Hardware Designs*, New York: Oxford University Press, 2000.

[Parihar01] S. Parihar et al., "A high density 0.10 µm CMOS technology using low K dielectric and copper interconnect," *Proc. Intl. Electron Devices Meeting*, 2001, pp. 11.4.1-11.4.4.

[Park99] J. Park, J. Lee, and W. Kim, "Current sensing differential logic: a CMOS logic for high reliability and flexibility," *JSSC*, vol. 34, no. 6, June 1999, pp. 904-908.

[Parker03] K. Parker, *The Boundary-Scan Handbook*, Boston: Kluwer Academmic Publishers, 2003.

[Partovi94] H. Partovi and D. Draper, "A regenerative push-pull differential logic family," *Proc. IEEE Intl. Solid-State Circuits Conf.*, 1994, pp. 294-295.

[Partovi96] H. Partovi et al., "Flow-through latch and edge-triggered flip-flop hybrid elements," *Proc. IEEE Intl. Solid-State Circuits Conf.*, 1996, pp. 138-139.

[Pasternak87] J. Pasternak, A. Shubat, and C. Salama, "CMOS differential pass-transistor logic design," *JSSC*, vol. SC-22, no. 2, April 1987, pp. 216-222.

[Pasternak91] J. Pasternak and C. Salama, "Design of submicrometer CMOS differential pass-transistor logic circuits," *JSSC*, vol. 26, no. 9, Sept. 1991, pp. 1249-1258.

[Patterson04] D. Patterson and J. Hennessy, *Computer Organization and Design*, 3rd ed., San Francisco, CA: Morgan Kaufmann, 2004.

[Paul02] B. Paul and K. Roy. Testing cross-talk induced delay faults in static CMOS circuit through dynamic timing analysis. *Proc. Intl. Test Conf.*, Oct. 2002, pp. 384-390.

[Pelgrom89] M. Pelgrom, A. Duinmaijer, and A. Welbers, "Matching properties of MOS transistors," *JSSC*, vol. 24, no. 5, Oct. 1989, pp. 1433-1440.

[Peng02] C. Peng et al., "A 90 nm generation copper dual damascene technology with ALD TaN barrier," *Tech. Digest Intl. Electron Devices Meeting*, Dec. 2002, pp. 603-606.

[Penney72] W. Penney and L. Lau, *MOS Integrated Circuits*, New York: Van Nostrand Reinhold, 1972.

[Petegem94] W. van Petegem, B. Geeraerts, W. Sansen, and B. Graindourze, "Electrothermal simulation and design of integrated circuits," *JSSC*, vol. 29, no. 2, Feb. 1994, pp. 143-146.

[Pfennings85] L. Pfennings, W. Mol, J. Bastiens, and J. van Dijk, "Differential split-level CMOS logic for subnanosecond speeds," *JSSC*, vol. SC-20, no. 5, Oct. 1985, pp. 1050-1055.

[Pihl98] J. Pihl, "Single-ended swing restoring pass transistor cells for logic synthesis and optimization," *Proc. IEEE Intl. Symp. Circuits and Systems*, vol. 2, 1998, pp. 41-44.

[Piña02] C. Piña, "Evolution of the MOSIS VLSI educational program," *Proc. Electronic Design, Test, and Applications Workshop*, 2002, pp. 187-191.

[Poulton03] K. Poulton et al., "A 20Gs/s 8 b ADC with a 1 MB Memory in 0.18 µm CMOS," *Proc. IEEE Solid-State Circuits Conf.*, Feb. 2003, pp. 318-319.

[Pretorius86] J. Pretorius, A. Shubat, and A. Salama, "Latched domino CMOS logic," *JSSC*, vol. SC-21, no. 4, Aug. 1986, pp. 514-522.

[Price95] D. Price, "Pentium FDIV flaw—lessons learned," *IEEE Micro*, vol. 15, no. 2, April 1995, pp. 86-88.

[Prince99] B. Prince, *High Performance Memories: New Architecture DRAMs and SRAMs—Evolution and Function*, New York: John Wiley & Sons, 1999.

[Proebsting91] R. Proebsting, "Speed enhancement technique for CMOS circuits," US Patent 4,985,643, 1991.

[Promizter01] G. Promitzer, "12-bit low-power fully differential switched capacitor noncalibrating successive approximation ADC with 1MS/s," *JSSC*, vol. 36, no. 7, July 2001, pp. 1138-1143.

[Quader94] K. Quader, E. Minami, W. Huang, P. Ko, and C. Hu, "Hot-carrier-reliability design guidelines for CMOS logic circuits," *JSSC*, vol. 29, no. 3, March 1994, pp. 253-262.

[Rabaey03] J. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits*, 2nd Ed., Upper Saddle River, NJ: Prentice Hall, 2003.

[Razavi98] B. Razavi, *RF Microelectronics*, Upper Saddle River, NJ: Prentice Hall, 1998.

[Reddy02] V. Reddy et al., "Impact of negative bias temperature instability on digital circuit reliability," *Proc. 40th IEEE Intl. Reliability Physics Symp.*, 2002, pp. 248-254.

[Restle98] P. Restle and A. Deutsch, "Designing the best clock distribution network," *Symp. VLSI Circuits Digest Tech. Papers*, 1998, pp. 2-5.

[Restle01] P. Restle et al., "A clock distribution network for microprocessors," *JSSC*, vol. 36, no. 5, May 2001, pp. 792-799.

[Riordan97] M. Riordan and L. Hoddeson, *Crystal Fire: The Invention of the Transistor and the Birth of the Information Age*, New York: W. W. Norton & Co, 1998.

[Rodgers89] B. Rodgers and C. Thurber, "A Monolithic ±5½-digit BiMOS A/D converter," *JSSC*, vol. 24, no. 3, June 1989, pp. 617-626.

[Rotella02] F. Rotella, V. Blaschke, and D. Howard, "A broad-band scalable lumped-element inductor model using analytic expressions to incorporate skin effect, substrate loss, and proximity effect," *Tech. Digest Intl. Electron Devices Meeting*, Dec. 2002, pp. 471-474.

[Ruehli73] A. Ruehli and P. Brennan, "Efficient capacitance calculations for three-dimensional multiconductor systems," *IEEE Trans. Microwave Theory and Techniques*, vol. MTT-21, No. 2, Feb. 1973, pp. 76-82.

[Rusu00] S. Rusu and G. Singer, "The first IA-64 microprocessor," *JSSC*, vol. 35, no. 11, Nov. 2000, pp. 1539-1544.

[Rusu03] S. Rusu, J. Stinson, S. Tam, J. Leung, H. Muljono, and B. Cherkauer, "A 1.5-GHz 130-nm Itanium 2 processor with 6-MB on-die L3 cache," *JSSC*, vol. 38, no. 11, Nov. 2003, pp. 1887-1895.

[Ryan01] P. Ryan et al., "A single chip PHY COFDM modem for IEEE 802.11a with integrated ADCs and DACs," *Proc. IEEE Intl. Solid-State Circuits Conf.*, 2001, pp. 338-339, 463.

[Rzepka98] S. Rzepka, K. Banerjee, E. Meusel, and C. Hu, "Characterization of self-heating in advanced VLSI interconnect lines based on thermal finite element simulation," *IEEE Trans. Components, Packaging, and Manufacturing Technology* - Part A, vol. 21, no. 3, Sept. 1998, pp. 406-411.

[Sah64] C. T. Sah, "Characteristics of the Metal-Oxide-Semiconductor Transistors," *IEEE Trans. Electron Devices*, ED-11, July 1964, pp. 324-345.

[Saint02] C. Saint and J. Saint, *IC Mask Design: Essential Layout Techniques*, New York: McGraw-Hill, 2002.

[Sakurai83] T. Sakurai, "Approximation of wiring delay in MOSFET LSI," *JSSC*, vol. SC-18, no.4, Aug. 1983, pp. 418-426.

[Sakurai86] T. Sakurai, K. Nogami, M. Kakumu, and T. Iizuka, "Hot-carrier generation in submicrometer VLSI environment," *JSSC*, vol. SC-21, no. 1, Feb. 1986, pp. 187-192.

[Sakurai90] T. Sakurai and R. Newton, "Alpha-Power Law MOSFET Model and its Applications to CMOS Inverter Delay and Other Formulas," *JSSC*, vol. 25, no. 2, April 1990, pp. 584-594.

[Samavati98] H. Samavati, A. Hajimiri, A. Shahani, G. Nazzerbakht, and T. Lee, "Fractal capacitors," *JSSC*, vol. 33, no. 12, Dec. 1998, pp. 2035-2041.

[Santoro89] M. Santoro, Design and Clocking of VLSI Multipliers, Ph.D. Thesis, Stanford University, CSL-TR-89-397, 1989.

[Sauerbrey02] J. Sauerbrey, T. Tille, D. Schmitt-Landsiedel and R. Thewes, "A 0.7-V MOSFET-only switched-opamp sigma delta modulator in standard digital CMOS technology," *JSSC*, vol. 37, no. 12, Dec. 2002, pp. 1662-1669.

[Sauerbrey03] J. Sauerbrey, D. Schmitt-Landsiedel and R. Thewes, "A 0.5V 1-μW successive approximation ADC," *JSSC*, vol. 38, no. 7, July 2003, pp. 1261-1265.

[Schellenberg03] F. Schellenberg, "A little light magic," *IEEE Spectrum*, vol. 40, no. 9, Sept. 2003, pp. 34-39.

[Schmitt38] O.H. Schmitt, "A thermionic trigger," *J. Scientific Instruments*, vol. 15, Jan. 1938, pp. 24-26.

[Scholtens02] P. Scholtens and M. Vertregt, "A 6-b 1.6-Gsample/s flash ADC in 0.18 μm CMOS using averaging termination," *JSSC*, vol. 37, no. 12, Dec. 2002, pp. 1599-1609.

[Schultz90] K. Schultz, R. Francis and K. Smith, "Ganged CMOS: trading standby power for speed," *JSSC*, vol. SC-25, no. 3, June 1990, pp. 870-873.

[Schultz95] K. Schultz and P. Gulak, "Architectures for large-capacity CAMs," *Integration*, vol. 18, no. 2-3, 1995, pp. 151-171.

[Schutten03] R. Schutten, T. Fitzbatrick, "Design for verification—blueprint for productivity and product quality," Synopsys white paper, 2003.

[Seeds67] R. Seeds, "Yield and cost analysis of bipolar LSI," *Intl. Electron Device Meeting*, Oct. 1967.

[Shahidi02] G. Shahidi, "SOI technology for the GHz era," *IBM J. Research and Development*, vol. 46, no. 2/3, March/May 2002, pp. 121-131.

[Sharma00] `http://www.free-ip.com/cordic/`

[She02] M. She et al., "JVD silicon nitride as tunnel dielectric in p-channel flash memory," *IEEE Electron Device Letters*, vol. 23, no. 2, Feb. 2002, pp. 91-93.

[Shepard99] K. Shepard, V. Narayanan, and R. Rose, "Harmony: static noise analysis of deep submicron digital integrated circuits," *IEEE Trans. Computer-Aided Design*, vol. 18, no. 8, Aug. 1999, pp. 1132-1150.

[Sheu87a] B. Sheu and P. Ko, "Measurement and modeling of short-channel MOS transistor gate capacitances," *JSSC*, vol. SC-22, no. 3, June 1987, pp. 464-472.

[Sheu87b] B. Sheu, D. Scharfetter, P. Ko, and M. Jeng, "BSIM: Berkeley short-channel IGFET model for MOS transistors," *JSSC*, vol. SC-22, no. 4, Aug. 1987, pp. 558-566.

[Shichman68] H. Shichman and D. Hodges, "Modeling and simulation of insulated-gate field-effect transistor switching circuits," *JSSC*, vol. SC-3, no. 3, Sept. 1968, pp. 285-289.

[Shockley52] W. Shockley, "A unipolar 'field-effect' transistor," *Proc. IRE*, vol. 40, 1952, pp. 1365-1376.

[Shoji82] M. Shoji, "Electrical design of BELLMAC-32a microprocessor," *Proc. IEEE Intl. Conf. Circuits and Computers, Sept. 1982*, pp. 112-115.

[Shoji85] M. Shoji, "FET scaling in domino CMOS gates," *JSSC*, vol. SC-20, no. 5, Oct. 1985, pp. 1067-1071.

[Shoji86] M. Shoji, "Elimination of process-dependent clock skew in CMOS VLSI," *JSSC*, vol. SC-21, no. 5, Oct. 1986, pp. 875-880.

[SIA97] Semiconductor Industry Association, *International Technology Roadmap for Semiconductors*, 1997.

[SIA02] Semiconductor Industry Association, *International Technology Roadmap for Semiconductors*, 2002 Update, public.itrs.net.

[Silberman98] J. Silberman et al., "A 1.0-GHz single-issue 64-bit PowerPC integer microprocessor," *JSSC*, vol. 33, no. 11, Nov. 1998, pp. 1600-1608.

[Simon92] T. Simon, "A fast static CMOS NOR gate," *Proc. Advanced Research in VLSI and Parallel Systems*, 1992, pp. 180-192.

[Sklansky60] J. Sklansky "Conditional-sum addition logic," *IRE Trans. Electronic Computers*, vol. EC-9, June 1960, pp. 226-231.

[Smith00] D. Smith and P. Franzon, *Verilog Styles for Synthesis of Digital Circuits*, Upper Saddle River, NJ: Prentice Hall, 2000.

[Soeleman01] H. Soeleman, K. Roy, and B. Paul, "Robust subthreshold logic for ultra-low power operation," *IEEE Trans. VLSI*, vol. 9, no. 1, Feb. 2001, pp. 90-99.

[Solomatnikov00] A. Solomatnikov, D. Somasekhar, K. Roy, and C. Koh, "Skewed CMOS: noise-immune high-performance low-power static circuit family," *Proc. IEEE Intl. Conf. Computer Design*, 2000, pp. 241-246.

[Somasekhar96] D. Somasekhar and K. Roy, "Differential current switch logic: a low power DCVS logic family," *JSSC*, vol. 31, no. 7, July 1996, pp. 981-991.

[Somasekhar98] D. Somasekhar and K. Roy, "LVDCSL: a high fan-in, high-performance, low-voltage differential current switch logic family," *IEEE Trans. VLSI*, vol. 6, no. 4, Dec. 1998, pp. 573-577.

[Somasekhar00] D. Somasekhar, S. Choi, K. Roy, Y. Ye, and V. De, "Dynamic noise analysis in precharge-evaluate circuits," *Proc. Design Automation Conf.*, 2000, pp. 243-246.

[Song96] M. Song, G. Kang, S. Kim, and B. Kang, "Design methodology for high speed and low power digital circuits with energy economized pass-transistor logic (EEPL)," *Proc. 22nd European Solid-State Circuits Conf.*, 1996, pp. 120-123.

[Song01] S. Song et al., "On the gate oxide scaling of high performance CMOS transistors," *Proc. Intl. Electron Devices Meeting*, 2001, pp. 3.2.1-3.2.4.

[Sowiati01] T. Sowlati, V. Vathulya, and D. Leenaerts, "High density capacitance structures in submicron CMOS for low power RF applications," *Proc. Intl. Symp. Low Power Electronics and Design*, Aug. 2001, pp. 243-246.

[Sparsø01] J. Sparsø and S. Furber, eds., *Principles of Asynchronous Circuit Design: A Systems Perspective*, Boston: Kluwer Academic Publishers, 2001.

[Srinivas92] H. Srinivas and K. Parhi, "A fast VLSI adder architecture," *JSSC*, vol. 27, no. 5, May 1992, pp. 761-767.

[Stinson03] J. Stinson and S. Rusu, "A 1.5 GHz third generation Itanium processor," *Proc. Design Automation Conf.*, 2003, pp. 706-709.

[Stojanovic99] V. Stojanovic and V. Oklobdija, "Comparative analysis of master-slave latches and flip-flops for high-performance and low-power systems," *JSSC*, vol. 34, no. 4, April 1999, pp. 536-548.

[Stroud02] C. Stroud, *A Designer's Guide to Built-in Self-Test*, Boston: Kluwer Academic Publishers, 2002.

[Sun87] J. Sun, Y. Taur, R. Dennard, and S. Klepner, "Submicrometer-channel CMOS for low-temperature operation," *IEEE Trans. Electron Devices*, vol. ED-34, no. 1, Jan. 1987, pp. 19-26.

[Sutherland99] I. Sutherland, B. Sproull, and D. Harris, *Logical Effort: Designing Fast CMOS Circuits*, San Francisco, CA: Morgan Kaufmann, 1999.

[Suzuki73] Y. Suzuki, K. Odagawa and T. Abe, "Clocked CMOS calculator circuitry," *JSSC*, vol. SC-8, no. 6, Dec. 1973, pp. 462-469.

[Suzuki93] M. Suzuki, N. Ohkubo, T. Shinbo, T. Yamanaka, A. Shimizu, K. Sasaki, and Y. Nakagome, "A 1.5-ns 32-b CMOS ALU in double pass-transistor logic," *JSSC*, vol. 28, no. 11, Nov. 1993, pp. 1145-1151.

[Sweeney02] P. Sweeney, *Error Control Coding: From Theory to Practice*, New York: John Wiley & Sons, 2002.

[Sylvester98] D. Sylvester and K. Keutzer, "Getting to the bottom of deep submicron," *Proc. IEEE/ACM Intl. Conf. Computer-Aided Design*, 1998, pp. 203-211.

[Tam00] S. Tam, S. Rusu, U. Desai, R. Kim, J. Zhang, and I. Young, "Clock generation and distribution for the first IA-64 microprocessor," *JSSC*, vol. 35, no. 11, Nov. 2000, pp. 1545-1552.

[Tam04] S. Tam, R. Limaye, and U. Desai, "Clock generation and distribution for the 130-nm Itanium 2 processor with 6-MB on-die L3 cache," *JSSC*, vol. 39, no. 4, Apr. 2004.

[Tharakan92] G. Tharakan and S. Kang, "A new design of a fast barrel switch network," *JSSC*, vol. 27, no. 2, Feb. 1992, pp. 217-221.

[Thomas02] D. Thomas and P. Moorby, *The Verilog Hardware Description Language*, 5th Ed. Boston: Kluwer Academic Publishers, 2002.

[Thorp99] T. Thorp, G. Yee, and C. Sechen, "Design and synthesis of monotonic circuits," *Proc. IEEE Intl. Conf. Computer Design*, 1999, pp. 569-572.

[Tiilikainen01] M. Tiilikainen, "A 14-bit 1.8-V 20-mW 1-mm^2 CMOS DAC," *JSSC*, vol. 36, no. 7, July 2001, pp. 1144-1147.

[Timko80] M. Timko and P. Holloway, "Circuit techniques for achieving high speed-high resolution A/D conversion," *JSSC*, vol. SC-15, no. 6, Dec. 1980, pp. 1040-1051.

[Timmermann94] D. Timmermann, B. Rix, H. Hahn and B. Hosticka, "A CMOS floating-point vector-arithmetic unit," *JSSC*, vol. 29, no. 5, Sept. 1994, pp. 634-639.

[Tobias95] P. Tobias and D. Trindade, *Applied Reliability*, 2nd ed., New York: Van Nostrand Reinhold, 1995.

[Troutman86] R. Troutman, *Latchup in CMOS Technology: The Problem and its Cure*, Boston: Kluwer Academic Publishers, 1986.

[Tschanz02] J. Tschanz et al., "Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage," *JSSC*, vol. 37, no. 11, Nov. 2002, pp. 1396-1402.

[Tsividis99] Y. Tsividis, *Operation and Modeling of the MOS Transistor*, 2nd ed., Boston: McGraw-Hill, 1999.

[Tyagi93] A. Tyagi, "A reduced-area scheme for carry-select adders," *JSSC*, vol. 42, no. 10, Oct. 1993, pp. 1163-1170.

[Uehara81] T. Uehara and W. vanCleemput, "Optimal layout of CMOS functional arrays," *IEEE Trans. Computers*, vol. C-30, no. 5, May 1981, pp. 305-312.

[Unger86] S. Unger and C. Tan, "Clocking schemes for high-speed digital systems," *IEEE Trans. Computers*, vol. 35, no. 10, Oct. 1986, pp. 880-895.

[Usami94] K. Usami and M. Horowitz, "Clustered voltage scaling technique for low-power design," *Proc. Intl. Symp. Low Power Electronics*, 1994, pp. 3-8.

[Uyttenhove03] K. Uyttenhove and M. Steyaert, "A 1.8-V 6-bit 1.3 GHz flash ADC in 0.25μm CMOS," *JSSC*, vol. 38, no. 7, July 2003, pp. 1115-1122.

[Vadasz66] L. Vadasz and A. Grove, "Temperature dependence of MOS transistor characteristics below saturation," *IEEE. Trans. Electron Devices*, vol. ED-13, no. 13, 1966, pp. 863-866.

[Vadasz69] L. Vadasz, A. Grove, T. Rowe, and G. Moore, "Silicon-gate technology," *IEEE Spectrum*, vol. 6, no. 10, Oct. 1969, pp. 28-35.

[van Berkel99] C. van Berkel and C. Molnar, "Beware the three-way arbiter," *JSSC*, vol. 34, no. 6, June 1999, pp. 840-848.

[Vangal02] S. Vangal et al., "5-GHz 32-bit integer execution core in 130-nm dual-V_T CMOS," *JSSC*, vol. 37, no. 11, Nov. 2002, pp. 1421-1432.

[vanZeijl02] P. van Zeijl et al., "A Bluetooth radio in 0.18μm CMOS," *JSSC*, vol. 37, no. 12, Dec. 2002, pp. 1679-1687.

[Veendrick80] H. Veendrick, "The behavior of flip-flops used as synchronizers and prediction of their failure rate," *JSSC*, vol. SC-15, no. 2, April 1980, pp. 169-176.

[Veendrick84] H. Veendrick, "Short-circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits," *JSSC*, vol. SC-19, no. 4, Aug. 1984, pp. 468-473.

[Vittal99] A. Vittal et al., "Crosstalk in VLSI interconnections," *IEEE Trans. Computer-Aided Design*, vol. 18, no. 12, Dec. 1999, pp. 1817-1824.

[Volder59] J. Volder, "The CORDIC trigonometric computing technique," *IRE Trans. Electronic Computers*, vol. EC-8, no. 3, Sept. 1959, pp-330-334.

[Vollertsen99] R. Vollertsen, "Burn-in," *IEEE Integrated Reliability Workshop Final Report*, 1999, pp. 167-173.

[Wadell01] B. Wadell, *Transmission Line Design Handbook*, Norwood, MA: Artech House, 1991.

[Wakerly00] J. Wakerly, *Digital Design Principles and Practices*, 3rd ed., Upper Sadde River, NJ: Prentice Hall, 2000.

[Wallace64] C. Wallace, "A suggestion for a fast multiplier," *IEEE Trans. Electronic Computers*, Feb. 1964, pp. 14-17.

[Wang86] L. Wang and E. McCluskey, "Complete feedback shift register design for built-in self test," *Proc. Design Automation Conf.*, Nov. 1986, pp. 56-59.

[Wang89] J. Wang, C. Wu, and M. Tsai, "CMOS nonthreshold logic (NTL) and cascode nonthreshold logic (CNTL) for high-speed applications," *JSSC*, vol. 24, no. 3, June 1989, pp. 779-786.

[Wang93] Z. Wang, G. Jullien, W. Miller, and J. Wang, "New concepts for the design of carry-lookahead adders," *Proc. IEEE Intl. Symp. Circuits and Systems*, 1993, vol. 3, pp. 1837-1840.

[Wang94] J. Wang, S. Fang, and W. Feng, "New efficient designs for XOR and XNOR functions on the transistor level," *JSSC*, vol. 29, no. 7, July 1994, pp. 780-786.

[Wang97] Z. Wang, G. Jullien, W. Miller, J. Wang, and S. Bizzan, "Fast adders using enhanced multiple-output domino logic," *JSSC*, vol. 32, no. 2, Feb. 1997, pp. 206-214.

[Wang00j] J. Wang and C. Huang, "High-speed and low-power CMOS priority encoders," *JSSC*, vol. 35, no. 10, Oct. 2000, pp. 1511-1514.

[Wang00l] L. Wang and N. Shanbhag, "An energy-efficient noise-tolerant dynamic circuit technique," *IEEE Trans. Circuits and Systems II*, vol. 47, no. 11, Nov. 2000, pp. 1300-1306.

[Wang01] J. Wang, C. Chang, and C. Yeh, "Analysis and design of high-speed and low-power CMOS PLAs," *JSSC*, vol. 36, no. 8, Aug. 2001, pp. 1250-1262.

[Wanlass63] F. Wanlass and C. Sah, "Nanowatt logic using field effect metal-oxide semiconductor triodes," *Proc. IEEE Intl. Solid-State Circuits Conf.*, 1963, pp. 32-33.

[Webb97] C. Webb et al., "A 400-MHz S/390 microprocessor," *JSSC*, vol. 32, no. 11, Nov. 1997, pp. 1665-1675.

[Wei98] L. Wei, Z. Chen, M. Johnson, K. Roy, and V. De, "Design and optimization of low voltage high performance dual threshold CMOS circuits," *Proc. Design Automation Conf.*, 1998, pp. 489-494.

[Weinberger58] A. Weinberger and J. Smith, "A logic for high-speed addition," System Design of Digital Computer at the National Bureau of Standards: Methods for High-Speed Addition and Multiplication, National Bureau of Standards, Circular 591, Section 1, Feb. 1958, pp. 3-12.

[Weinberger81] A. Weinberger, "4-2 carry-save adder module," *IBM Technical Disclosure Bulletin*, vol. 23, no. 8, Jan. 1981, pp. 3811-3814.

[Weiss02] D. Weiss, J. Wuu, and V. Chin, "The on-chip 3-MB subarray-based third-level cache on an Itanium microprocessor," *JSSC*, vol. 37, no. 11, Nov. 2002, pp. 1523-1529.

[Weste93] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*, 2nd ed., Reading, MA: Addison-Wesley, 1993.

[Wicht01] B. Wicht, S. Paul, and D. Schmitt-Landsiedel, "Analysis and compensation of the bitline multiplexer in SRAM current sense amplifiers," *JSSC*, vol. 36, no. 11, Nov. 2001, pp. 1745-1755.

[Williams83] T. Williams and K. Parker, "Design for Testability—A Survey," *Proc. IEEE*, vol. 71, no. 1, Jan. 1983, pp. 98-112.

[Williams86] T. Williams, "Design for testability," *Proc. NATO Advanced Study Inst. Computer Design Aids for VLSI Circuits*, (P. Antognetti et al. ed.), NATO AIS Series, 1986, Martinus Nijhoff Publishers, pp. 359-416.

[Williams91] T. Williams and M. Horowitz, "A zero-overhead self-timed 160-ns 54-b CMOS divider," *JSSC*, vol. 26, no. 11, Nov. 1991, pp. 1651-1661.

[Wing82] O. Wing, "Automated gate matrix layout," *Proc. IEEE Intl. Symp. Circuits and Systems*, vol. 2, 1982, pp. 681-685.

[Wolf00] S. Wolf and R. Tauber, *Silicon Processing for the VLSI Era*, 2nd ed., Sunset Beach, CA: Lattice Press, 2000.

[Wong02] H. Wong, "Beyond the conventional transistor," *IBM Journal of Research and Development*, vol. 46, no. 2/3, March/May 2002, pp. 133-168.

[Wood01] J. Wood, T. Edwards, and S. Lipa, "Rotary traveling-wave oscillator arrays: a new clock technology," *JSSC*, vol. 36, no. 11, Nov. 2001, pp. 1654-1665.

[Wu87] C. Wu, J. Wang and M. Tsai, "The analysis and design of CMOS multidrain logic and stacked multidrain logic," *JSSC*, vol. SC-22, no. 1, Feb. 1987, pp. 47-56.

[Wu91] C. Wu and K. Cheng, "Latched CMOS differential logic (LCDL) for complex high-speed VLSI," *JSSC*, vol. 26, no. 9, Sept. 1991, pp. 1324-1328.

[Wurtz93] L. Wurtz, "An efficient scaling procedure for domino CMOS logic," *JSSC*, vol. 28, no. 9, Sept. 1993, pp. 979-982.

[Yamada95] H. Yamada, T. Hotta, T. Nishiyama, F. Murabayashi, T. Yamauchi, and H. Sawamoto, "A 13.3 ns double-precision floating-point ALU and multiplier," *Proc. Intl. Conf. Computer Design*, 1995, pp. 466-470.

[Yang98] S. Yang et al., "A high performance 180 nm generation logic technology," *Tech. Digest Intl. Electron Device Meeting*, Dec. 1998, pp. 197-200.

[Yano90] K. Yano, T. Yamanaka, T. Nishida, M. Saito, K. Shimohigashi, and A. Shimizu, "A 3.8-ns 16 x 16-b multiplier using complementary pass-transistor logic," *JSSC*, vol. 25, no. 2, April 1990, pp. 388-395.

[Yano96] K. Yano, Y. Sasaki, K. Rikino, and K. Seki, "Top-down pass-transistor logic design," *JSSC*, vol. 31, no. 6, June 1996, pp. 792-803.

[Ye98] Y. Ye, S. Borkar, and V. De, "A new technique for standby leakage reduction in high-performance circuits," *Symp. VLSI Circuits Digest Tech. Papers*, 1998, pp. 40-41.

[Ye00] Y. Ye, J. Tschanz, S. Narendra, S. Borkar, M. Stan, and V. De, "Comparative delay, noise and energy of high-performance domino adders with stack node preconditioning (SNP)," *Symp. VLSI Circuits Digest Tech. Papers*, 2000, pp. 188-191.

[Yee00] G. Yee and C. Sechen, "Clock-delayed dominio for dynamic circuit design," *IEEE Trans. VLSI*, vol. 8, no. 4, Aug. 2000, pp. 425-430.

[Yoon02] J. Yoon et al., "CMOS-compatible surface-micromachined suspended-spiral inductors for multi-GHz silicon RF Ics" *IEEE Electron Device Letters*, vol. 23, no. 10, Oct. 2002, pp. 591-593.

[Yoshimoto83] M. Yoshimoto et al., "A divided word-line structure in the static RAM and its application to a 64K full CMOS RAM," *JSSC*, vol. SC-18, no. 5, Oct. 1983, pp. 479-485.

[Young00] K. Young et al., "A 0.13 μm CMOS technology with 193 nm lithography and Cu/Low-k for high performance applications," *Proc. Intl. Electron Devices Meeting*, 2000, pp. 563-566.

[Yuan82] C. Yuan and T. Trick, "A simple formula for the estimation of the capacitance of two-dimensional interconnects in VLSI circuits," *IEEE Electron Device Letters*, vol. EDL-3, Dec. 1982, pp. 391-393.

[Yuan89] J. Yuan and C. Svensson, "High-speed CMOS circuit technique," *JSSC*, vol. 24, no. 1, Feb. 1989, pp. 62-70.

[Zhou99] X. Zhou, K. Lim, and D. Lim, "A simple and unambiguous definition of threshold voltage and its implications in deep-submicron MOS device modeling," *IEEE Trans. Electron Devices*, vol. 46, no. 4, April 1999, pp. 807-809.

[Zhou03] Y. Zhou and J. Yuan, "A 10-bit wide-band CMOS direct digital RF amplitude modulator," *JSSC*, vol. 38, no. 7, July 2003, pp. 1182-1188.

[Zhuang92] N. Zhuang and H. Wu, "A new design of the CMOS full adder," *JSSC*, vol. 27, no. 5, May 1992, pp. 840-844.

[Ziegler02] J. Ziegler, *Ion-Implantation—Science and Technology, 2002 Edition*, IIT Press, 2002.

[Ziegler96] J. Ziegler, "Terrestrial cosmic rays," *IBM J. Research and Development*, vol. 40, no. 1, Jan. 1996, pp. 19-39.

[Zimmermann96] R. Zimmermann, "Non-heuristic optimization and synthesis of parallel-prefix adders," *Proc. Intl. Workshop on Logic and Architecture Synthesis*, Dec. 1996, pp. 123-132.

[Zimmermann97a] R. Zimmermann and W. Fichtner, "Low-power logic styles: CMOS versus pass-transistor logic," *JSSC*, vol. 32, no. 7, July 1997, pp. 1079-1090.

[Zimmermann97b] R. Zimmermann, *Binary Adder Architectures for Cell-Based VLSI and their Synthesis*, ETH Dissertation 12480, Swiss Federal Institute of Technology, 1997.

[Zuras86] D. Zuras and W. McAllister, "Balanced delay trees and combinatorial division in VLSI," *JSSC*, vol. SC-21, no. 5, Oct. 1986, pp. 814-819.

Index

Abbreviated MOSIS design rules					
Layer	Rule	Description	SCMOS	SUBM	DEEP
Well	1.1	Width	10	12	12
	1.2	Spacing to well at different potential	9	18	18
	1.3	Spacing to well at same potential	6	6	6
Active (diffusion)	2.1	Width	3	3	3
	2.2	Spacing to active	3	3	3
	2.3	Source/drain surround by well	5	6	6
	2.4	Substrate/well contact surround by well	3	3	3
	2.5	Spacing to active of opposite type	4	4	4
Poly	3.1	Width	2	2	2
	3.2	Spacing to poly over field oxide	2	3	3
	3.2a	Spacing to poly over active	2	3	4
	3.3	Gate extension beyond active	2	2	2.5
	3.4	Active extension beyond poly	3	3	4
	3.5	Spacing of poly to active	1	1	1
Select	4.1	Spacing from substrate/well contact to gate	3	3	3
	4.2	Overlap of active	2	2	2
	4.3	Overlap of substrate/well contact	1	1	1.5
	4.4	Spacing to select	2	2	4
Contact (to poly or active)	5.1, 6.1	Width (exact)	2x2	2x2	2x2
	5.2b, 6.2b	Overlap by poly or active	1	1	1
	5.3, 6.3	Spacing to contact	2	3	4
	5.4, 6.4	Spacing to gate	2	2	2
	5.5b	Spacing of poly contact to other poly	4	5	5
	5.7b, 6.7b	Spacing to active/poly for multiple poly/active contacts	3	3	3
	6.8b	Spacing of active contact to poly contact	4	4	4
Metal1	7.1	Width	3	3	3
	7.2	Spacing to metal1	2	3	3
	7.3, 8.3	Overlap of contact or via	1	1	1
	7.4	Spacing to metal for lines wider than 10 λ	4	6	6
Via1–Via(N-1)	8.1, 14.1, …	Width (exact)	2x2	2x2	3x3
	8.2, 14.2, …	Spacing to via on same layer	3	3	3
	8.4	Spacing to contacts (if no stacked vias)	2	2	n/a
	8.5	Spacing of via1 to poly or active edge	2	2	n/a
	14.4	Spacing of via2 to via1 (if no stacked vias)	2	2	n/a
Metal2–Metal(N-1)	9.1, …	Width	3	3	3
	9.2, …	Spacing to same layer metal	3	3	4
	9.3, …	Overlap of via	1	1	1
	9.4, …	Spacing to metal for lines wider than 10 λ	6	6	8
Metal3 (3-layer process)	15.1	Width	6	5	n/a
	15.2	Spacing to metal3	4	3	n/a
	15.3	Overlap of via2	2	2	n/a
	15.4	Spacing to metal for lines wider than 10 λ	8	6	n/a

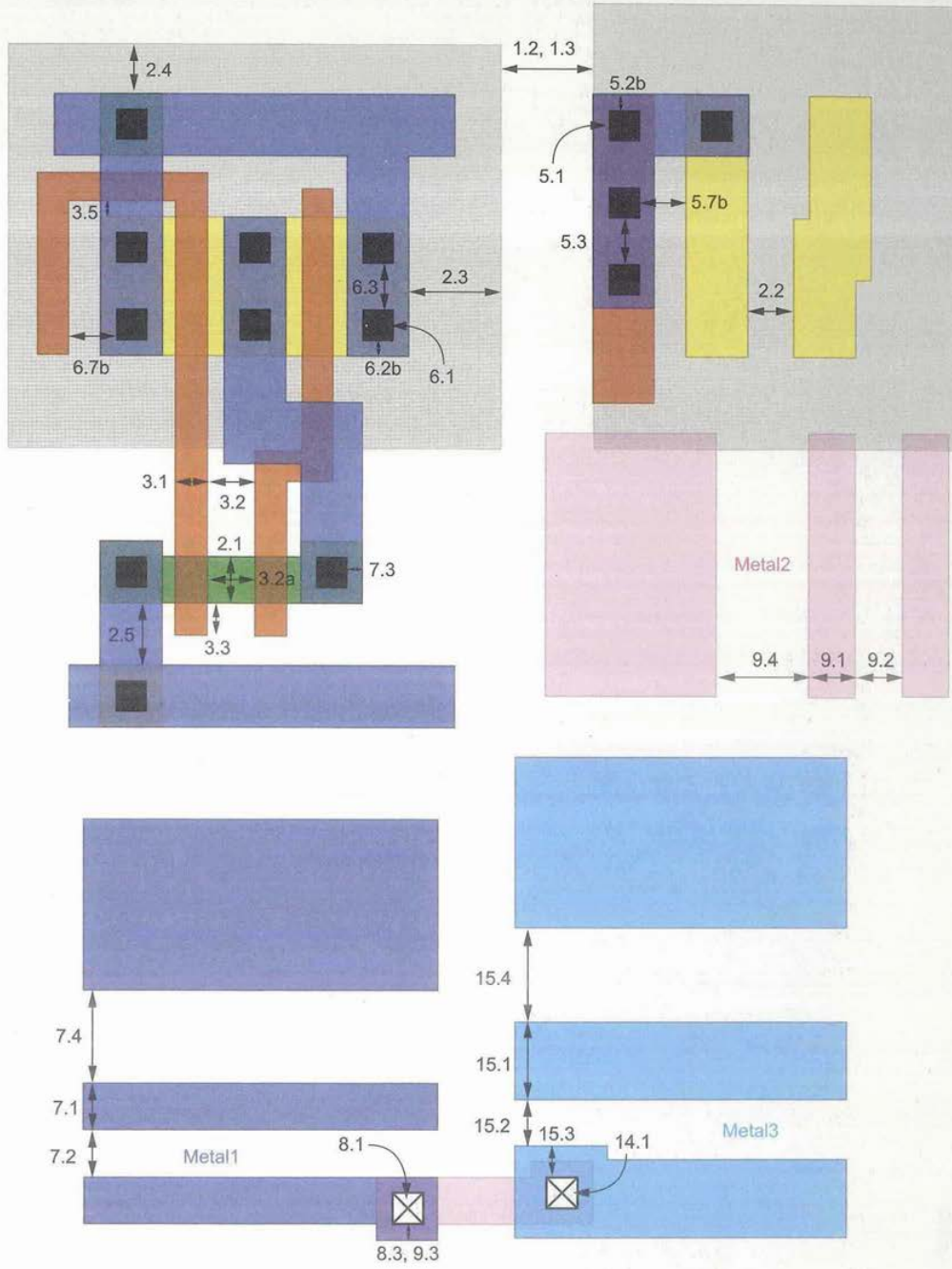


Fig 3.11 MOSIS Design Rules

CMOS VLSI

CMOS VLSI DESIGN

Third Edition

This best-selling text on CMOS VLSI design has been extensively revised to include details on modern techniques for the design of complex and high-performance CMOS systems on a chip. Covering CMOS design from the digital systems level to the circuit level, *CMOS VLSI Design*, Third Edition includes both an explanation of fundamental principles and a guide to good design practices.

Neil H.E. Weste and new coauthor David Harris draw upon their extensive industry and classroom experience to explain modern practices of chip design. Having worked at such companies as Intel, Cisco, and Bell Labs, the authors bring a real-world perspective to the topic by sharing lessons learned in their professional practices.

New to this Edition

- Comprehensive coverage of key CMOS design issues: circuits, interconnect, and clocking
- Detailed treatment of low-power techniques
- Integration of logical effort for efficient design of fast circuits
- Tutorial chapters on Verilog, VHDL, and SPICE (circuit) simulation
- Extensive exercises and worked-out examples for learning reinforcement

About the Authors

Neil H.E. Weste is Director of NHEW R&D Pty Ltd. and an Adjunct Professor at both Macquarie University and the University of Adelaide. He received his B.Sc., B.E.(Elec) and Ph.D. degrees from the University of Adelaide. His interests are wireless technologies, systems on a chip, analog, RF and digital IC design, and technology incubation.

David Harris is an Associate Professor of Engineering at Harvey Mudd College in Claremont, CA. David received his Ph.D. from Stanford University and his S.B. and M. Eng. degrees from MIT. His research interests include high-speed CMOS VLSI design, microprocessors, and computer arithmetic. He holds seven patents, is the author of two other VLSI books, and has designed chips at Sun Microsystems, Intel, Hewlett-Packard, and Evans & Sutherland.

PEARSON
Addison Wesley

Addison-Wesley Computing • Leading Authors • Quality Products
Log on to **aw-bc.com/computing** for a full list of titles.

9 780321 149015 90000
ISBN 0-321-14901-7