

binational logic and be sampled by  $R2$  on edge ④. If the clock skew is positive, the time available for a signal to propagate from  $R1$  to  $R2$  is increased by the skew  $\delta$ . The output of the combinational logic must be valid one setup time before the rising edge of  $CLK_2$  (point ④). The constraint on the minimum clock period can then be derived as follows:

$$T + \delta \geq t_{c-q} + t_{logic} + t_{su} \quad \text{or} \quad T \geq t_{c-q} + t_{logic} + t_{su} - \delta \quad (10.3)$$

This equation suggests that clock skew actually has the potential to improve the performance of the circuit. That is, the minimum clock period required to operate the circuit reliably reduces with increasing clock skew! This is indeed correct, but unfortunately, increasing skew makes the circuit more susceptible to race conditions, which may harm the correct operation of sequential systems.

This can be illustrated by the following example: Assume again that input  $In$  is sampled on the rising edge of  $CLK_1$  at edge ① into  $R1$ . The new value at the output of  $R1$  propagates through the combinational logic and should be valid before edge ④ at  $CLK_2$ . However, if the minimum delay of the combinational logic block is *small*, the inputs to  $R2$  may change before the clock edge ②, resulting in incorrect evaluation. To avoid races, we must ensure that the minimum propagation delay through the register and logic is long enough that the inputs to  $R2$  are valid for a hold time after edge ②. The constraint can be formally stated as

$$\delta + t_{hold} < t_{(c-q, cd)} + t_{(logic, cd)} \quad (10.4)$$

or

$$\delta < t_{(c-q, cd)} + t_{(logic, cd)} - t_{hold}$$

Figure 10-7 shows the timing diagram for the case in which  $\delta < 0$ . For this case, the rising edge of  $CLK_2$  happens before the rising edge of  $CLK_1$ . On the rising edge of  $CLK_1$ , a new input is sampled by  $R1$ . The new data propagate through the combinational logic, and they are sampled by  $R2$  on the rising edge of  $CLK_2$ , which corresponds to edge ④. As Figure 10-7 and Eq. (10.3) clearly show, a negative skew adversely impacts the performance of a sequential system. However, assuming  $t_{hold} + \delta < t_{(c-q, cd)} + t_{(logic, cd)}$ , a negative skew implies that the system never fails, since edge ② happens before edge ①!

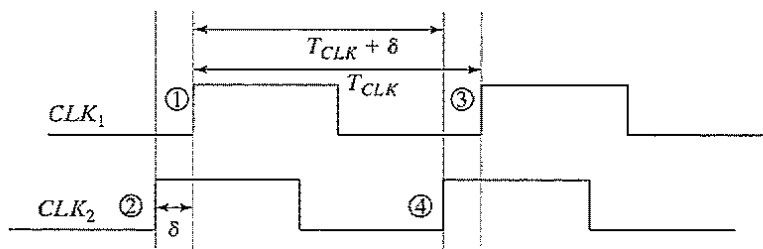


**Figure 10-7** Timing diagram for the case when  $\delta < 0$ . The rising edge of  $CLK_2$  arrives earlier than the edge of  $CLK_1$ .

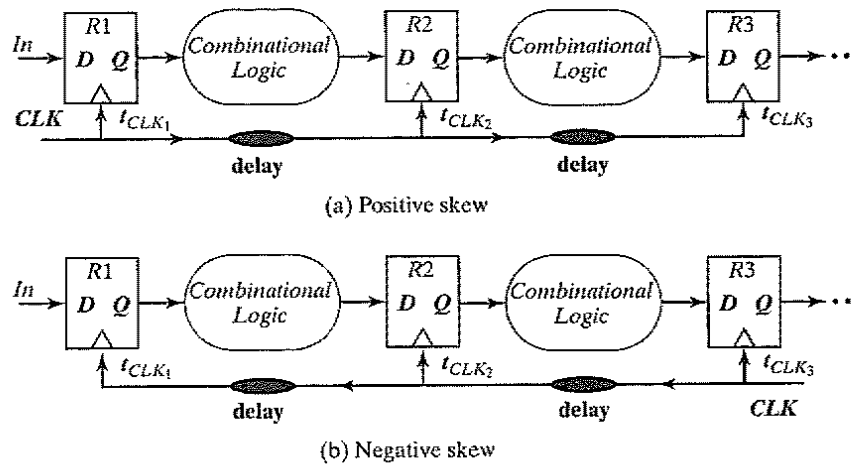


**Figure 10-8** Positive and negative clock skew scenarios.

Example scenarios for positive and negative clock skew are shown in Figure 10-8.

- $\delta > 0$ —This corresponds to a clock routed in the same direction as the flow of the data through the pipeline (Figure 10-8a). In this case, the skew has to be strictly controlled and satisfy Eq. . If the constraint is not met, the circuit malfunctions **independently of the clock period**. Reducing the clock frequency of an edge-triggered circuit does not help getting around skew problems! It is therefore necessary to satisfy the hold-time constraints at design time. On the other hand, positive skew increases the through put of the circuit as expressed by Eq. (10.3). The clock period can be shortened by  $\delta$ . The extent of this improvement is limited, as large values of  $\delta$  soon provoke violations of Eq. .
- $\delta < 0$ —When the clock is routed in the opposite direction of the data (Figure 10-8b), the skew is negative and provides significant immunity to races; if the hold time is zero or negative, races are eliminated because Eq. is unconditionally met! The skew reduces the time available for actual computation so that the clock period has to be increased by  $|\delta|$ . In summary, routing the clock in the opposite direction of the data avoids disasters, but hampers the circuit performance.

Unfortunately, since a general logic circuit can have data flowing in both directions (for example, circuits with feedback), this solution to eliminate races does not always work. Figure 10-9 shows that the skew can assume both positive and negative values, depending on the direction of the data transfer. Under these circumstances, the designer has to account for the worst case skew condition. In general, routing the clock so that only negative skew occurs is not feasible. Therefore, the design of a low-skew clock network is essential.

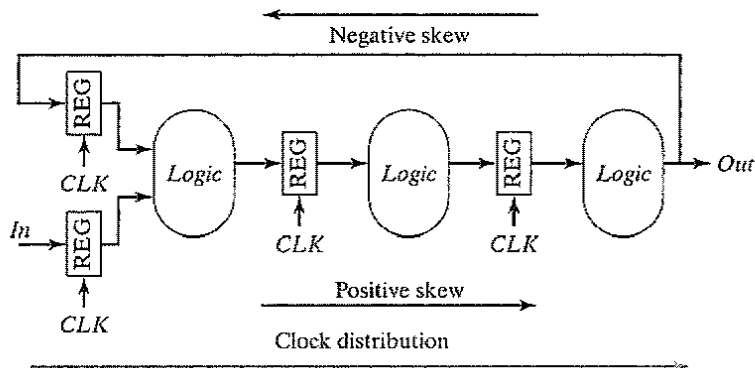


Figure 10-9 Datapath structure with feedback.

**Example 10.1 Propagation and Contamination Delay Estimation**

Consider the logic network shown in Figure 10-10. Determine the contamination and propagation delays of the network, given a worst case gate delay of  $t_{gate}$ . We also assume that the maximum and minimum delays of the gates are identical.

The contamination delay is easily found; it equals  $2t_{gate}$ , and is the delay through  $OR_1$  and  $OR_2$ . On the other hand, computation of the worst case propagation delay is not as simple. At first glance, it would appear that the worst case corresponds to path ①, and its delay is  $5t_{gate}$ . However, when analyzing the data dependencies, it becomes obvious that path ① can never be exercised. Path ① is called a *false path*. If  $A = 1$ , the critical path goes through  $OR_1$  and  $OR_2$ . If  $A = 0$  and  $B = 0$ , the critical path is through  $I_1$ ,  $OR_1$  and  $OR_2$  (corresponding to a delay of  $3t_{gate}$ ). For the case in which  $A = 0$  and  $B = 1$ , the longest path goes through  $I_1$ ,  $OR_1$ ,  $AND_3$  and  $OR_2$ . In other words, for this simple (but contrived) network, the output does not even depend on inputs  $C$  and  $D$  (that is, there is **redundancy**). Therefore, the actual propagation delay is  $4t_{gate}$ . Given the propagation and contamination delay, the minimum and maximum allowable skew can be easily computed.

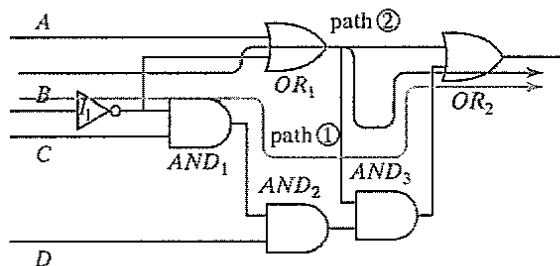


Figure 10-10 Logic network for computation of performance.

**WARNING:** The computation of the worst case propagation delay for combinational logic, due to the existence of *false paths*, cannot be obtained simply by adding the propagation delays of individual logic gates. The critical path is strongly dependent on circuit topology and data dependencies.

### Clock Jitter

*Clock jitter* refers to the temporal variation of the clock period at a given point on the chip—that is, the clock period can reduce or expand on a cycle-by-cycle basis. It is strictly a temporal uncertainty measure, and it is often specified at a given point. **Jitter** can be measured and characterized in a number of ways and is a **zero-mean random variable**. The *absolute jitter* ( $t_{jitter}$ ) refers to the worst case variation (absolute value) of a clock edge at a given location with respect to an ideally periodic reference clock edge. The *cycle-to-cycle jitter* ( $T_{jitter}$ ) typically refers to the time-varying deviations of a single clock period relative to an ideal reference clock. For a given spatial location  $i$ , it is given as  $T_{jitter}^i(n) = t_{clk,n+1}^i - t_{clk,n}^i - T_{CLK}$ , where  $t_{clk,n+1}^i$  and  $t_{clk,n}^i$  represent the arrival time of the  $n + 1^{th}$  and the  $n^{th}$  clock edges at node  $i$ , respectively, and  $T_{CLK}$  is the nominal clock period. Under the worst case conditions, the magnitude of the cycle-to-cycle jitter equals twice the absolute jitter ( $2t_{jitter}^i$ ).

Jitter directly impacts the performance of a sequential system. Figure 10-11 shows the nominal clock period, as well as the variation in period. Ideally, the clock period starts at edge ② and ends at edge ⑤, with a nominal clock period of  $T_{CLK}$ . However, the worst case scenario happens when the leading edge of the current clock period is delayed by jitter (edge ③), while jitter causes the leading edge of the next clock period to occur early (edge ④). As a result, the total time available to complete the operation is reduced by  $2t_{jitter}$  in the worst case and is given by

$$T_{CLK} - 2t_{jitter} \geq t_{c-q} + t_{logic} + t_{su} \quad \text{or} \quad T \geq t_{c-q} + t_{logic} + t_{su} + 2t_{jitter} \quad (10.5)$$

Equation (10.5) illustrates that jitter directly reduces the performance of a sequential circuit. Keeping it within strict bounds is essential if one is concerned about performance.

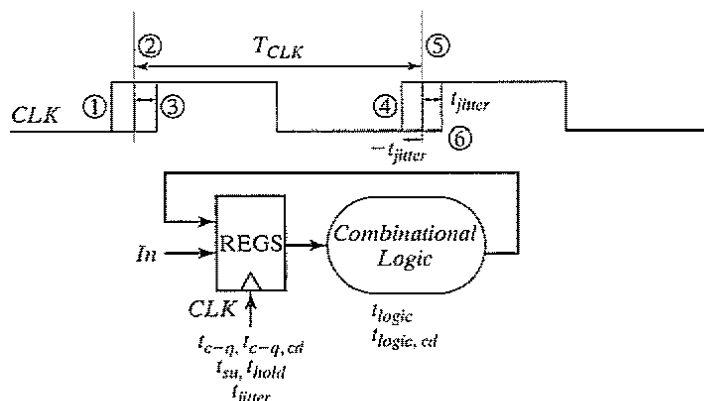


Figure 10-11 Circuit for studying the impact of jitter on performance.

### The Combined Impact of Skew and Jitter

In this section, the combined impact of skew and jitter is studied for conventional edge-triggered clocking. Consider the sequential circuit shown in Figure 10-14.

Assume that as a result of the clock distribution, there is a static skew  $\delta$  between the clock signals at the two registers (assume that  $\delta > 0$ ). Furthermore, the two clocks experience a jitter of  $t_{jitter}$ . To determine the constraint on the minimum clock period, we must look at the minimum available time to perform the required computation. The worst case occurs when the leading edge of the current clock period on  $CLK_1$  happens late (edge ③) and the leading edge of the next cycle of  $CLK_2$  happens early (edge ⑩). This results in the following constraint:

$$T_{CLK} + \delta - 2t_{jitter} \geq t_{c-q} + t_{logic} + t_{su} \quad (10.6)$$

or

$$T \geq t_{c-q} + t_{logic} + t_{su} - \delta + 2t_{jitter}$$

This equation illustrates that positive skew can provide a performance advantage. On the other hand, *jitter* always has a negative impact on the minimum clock period.<sup>1</sup>

To formulate the minimum delay constraint, consider the case in which the leading edge of the  $CLK_1$  cycle arrives early (edge ①), and the leading edges the current cycle of  $CLK_2$  arrives late (edge ⑩). The separation between edges ① and ⑩ should be smaller than the minimum delay through the network. This results in

$$\delta + t_{hold} + 2t_{jitter} < t_{(c-q, cd)} + t_{(logic, cd)} \quad (10.7)$$

or

$$\delta < t_{(c-q, cd)} + t_{(logic, cd)} - t_{hold} - 2t_{jitter}$$

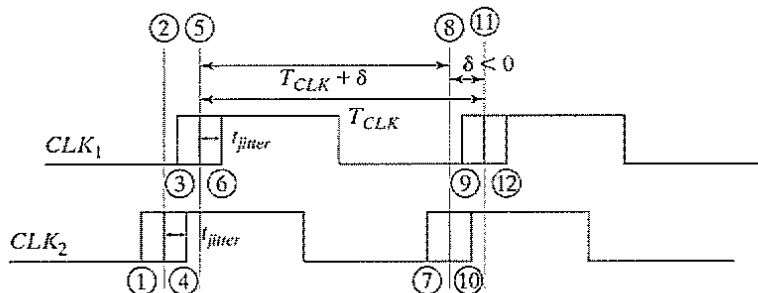


**Figure 10-12** Sequence circuit with a negative clock skew ( $\delta$ ). The skew is assumed to be larger than the *jitter*.

<sup>1</sup>This analysis is definitely for the worst case. It assumes that the jitter values at the source and the destination nodes are independent statistical variables. In reality, the clock edges involved in the hold-time analysis are derived from the same clock edge and are statistically dependent. Taking this dependence into account reduces the timing constraints substantially.

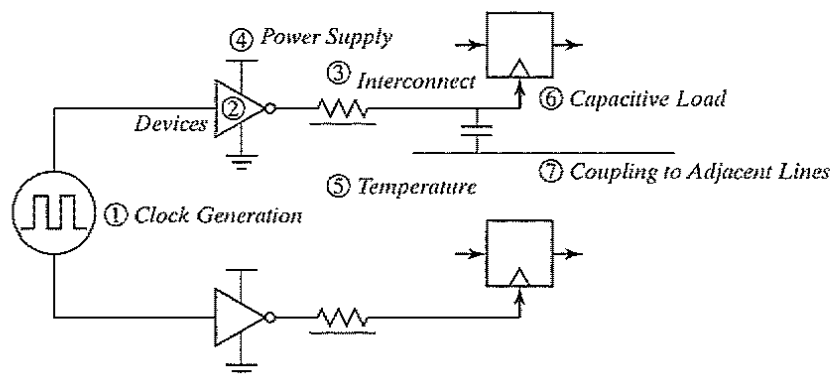


Figure 10-13 Skew and jitter sources in synchronous clock distribution.

This relation indicates that the acceptable skew is reduced by the *jitter* of the two signals.

Now consider the case in which the skew is negative ( $\delta < 0$ ), as shown in Figure 10-12. Assume that  $|\delta| > t_{jitter}$ . It can be verified that the worst case timing is exactly the same as in the previous analysis, with  $\delta$  taking a negative value. That is, negative skew reduces performance.

### 10.3.2 Sources of Skew and Jitter

A perfect *clock* is defined as a periodic signal that simultaneously triggers various memory elements on the chip. However, due to a variety of process and environmental variations, clocks are not ideal. To illustrate the sources of skew and jitter, consider a simplistic view of a typical clock generation and distribution network, as shown in Figure 10-13. A high-frequency clock is either provided from off chip or generated on chip. From a central point, the clock is distributed using multiple *matched* paths to low-level sequential elements. In this picture, two paths are shown. The clock paths include the wiring and the associated distributed buffers required to drive interconnect and loads. A key point to realize in clock distribution is that the **absolute delay through a clock distribution path is not important**; what matters is the relative arrival time at the register points at the end of each path. It is perfectly acceptable for the clock signal to take multiple cycles to get from a central distribution point to a low-level register as long as all clocks arrive at the same time at all the registers on the chip.

There are many reasons why the two parallel paths don't result in exactly the same delay. The sources of clock uncertainty can be classified in several ways. First, errors can be divided into two categories: *systematic* and *random*. *Systematic* errors are nominally identical from chip to chip and are predictable (for instance, variation in total load capacitance of each clock path). In principle, such errors can be modeled and corrected at design time, given sufficiently good models and simulators. Short of that, systematic errors can be deduced from measurements over a set of chips, and the design can be adjusted to compensate. *Random* errors are due to manufacturing variations that are difficult to model and eliminate (for instance, dopant fluctuations that result in threshold variations).

Mismatches may also be characterized as *static* or *time varying*. In practice, a continuum exists between changes that are slower than the time constant of interest and those that are faster. For example, temperature gradients on a chip vary on a millisecond time scale. A clock network

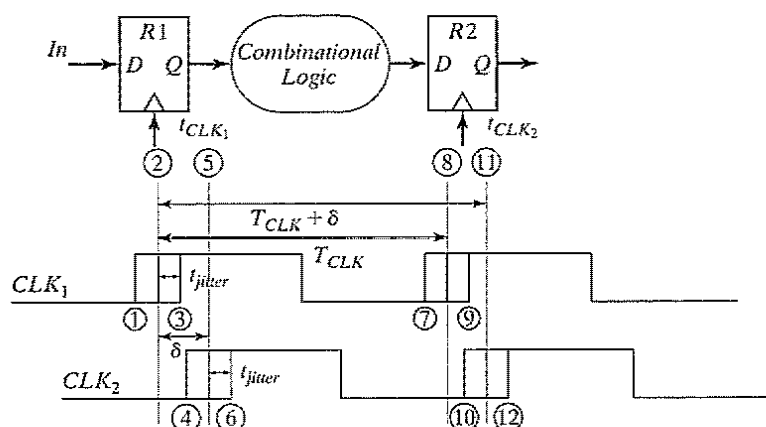


**Figure 10-14** Sequential circuit to study the impact of skew and jitter on *edge-triggered* systems. In this example, a positive skew ( $\delta$ ) is assumed.

tuned by a one-time calibration is vulnerable to the time-varying mismatch caused by the varying thermal gradients. On the other hand, thermal changes appear essentially static to a feedback network with a bandwidth of several megahertz. Another example is fielded by power-supply noise. The clock net is usually by far the largest signal net on the chip, and simultaneous transitions on the clock drivers induce noise in the power supply. This high-speed effect does not create a time-varying mismatch, because it is the same at every clock cycle and affects each rising clock edge the same way. Of course, this power-supply glitch may still cause static mismatch if it is not the same throughout the chip. The various sources of skew and jitter introduced in Figure 10-13 are described and characterized in detail in the sections that follow.

### Clock-Signal Generation (1)

The generation of the clock signal itself causes **jitter**. A typical on-chip clock generator, as described at the end of this chapter, takes a low-frequency reference clock signal and produces a high-frequency global reference for the processor. The core of such a generator is a *voltage-controlled oscillator* (VCO). This is an analog circuit, sensitive to intrinsic device noise and power-supply variations. A major problem is the coupling from the surrounding noisy digital circuitry through the substrate. This is especially a problem in modern fabrication processes that use a lightly doped epitaxy on the heavily doped substrate (to combat latch up). This causes substrate noise to travel over large distances on the chip [Maneatis00]. These noise sources cause temporal variations in the clock signal that propagate unfiltered through the clock drivers to the flip-flops, and result in *cycle-to-cycle* clock-period variations.

### Manufacturing Device Variations (2)

Distributed buffers are integral components of the clock distribution networks. They are required to drive both the register loads and the global and local interconnects. The matching of devices in the buffers along multiple clock paths is critical to minimizing timing uncertainty. Unfortunately, as a result of process variations, device parameters in the buffers vary along different paths, resulting in *static skew*. There are many sources of variations that contribute, such as oxide variations

(which affect the gain and threshold), dopant variations, and lateral dimension (width and length) variations. The doping variations can affect the depth of junction and dopant profiles and cause electrical parameters (such as device threshold and parasitic capacitances) to vary.

The orientation of polysilicon also can have some impact on the device parameters. Keeping the orientation the same across the chip for the clock drivers is therefore critical. Variation in the polysilicon critical dimension is particularly important, because it translates directly into MOS transistor channel length, impacting the drive current and switching characteristics. Spatial variation usually consists of a wafer-level (or within-wafer) variation and a die-level (or within-die) variation. At least part of this variation is systematic and therefore can be modeled and compensated for. The random variations, however, ultimately limit the matching and lower bound of the skew that can be achieved.

### Interconnect Variations (3)

Vertical and lateral dimension variations cause the interconnect capacitance and resistance to vary across a chip. Since this variation is static, it causes **skew** between different paths. One important source of interconnect variation is the *Inter-layer Dielectric (ILD)* thickness variation. In the formation of aluminum interconnect, layers of silicon dioxide are interposed between layers of patterned metallization. Oxide is deposited over a layer of patterned metal features, generally resulting in some remaining step height or surface topography. *Chemical-mechanical polishing (CMP)* is used to “planarize” the surface and remove the topography resulting from deposition and etch (as described in Chapter 3 and shown in Figure 10-15a). While at the feature scale (i.e., over an individual metal line), CMP can achieve excellent planarity, there are limitations on it over a global range. This is due primarily to variations in the polish rate, which is a function of the circuit layout density and pattern effects. Figure 10-15b shows this effect—the polish rate is higher for the lower-spatial-density region, resulting in a smaller dielectric thickness and higher capacitance.

The assessment and control of variation is of critical importance in semiconductor process development and manufacturing. Significant advances have been made to develop analytical

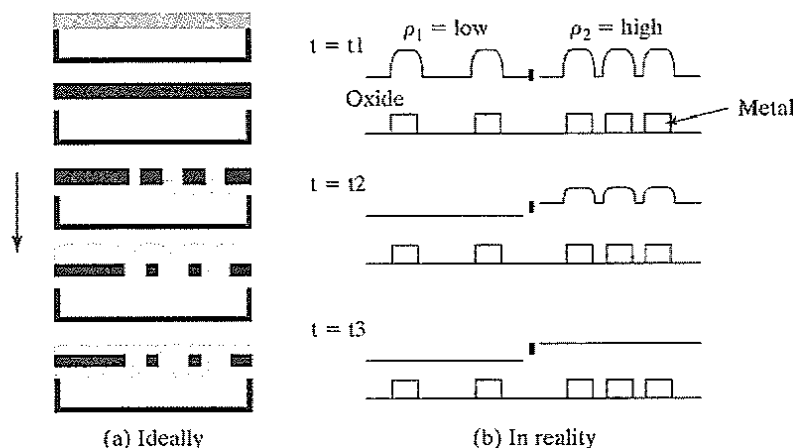


**Figure 10-15** Inter-level Dielectric (ILD) thickness variation due to density (Courtesy of Duane Boning.).

models for estimating the ILD thickness variations, based on spatial density. Since this component is often predictable from the layout, it is possible to actually correct for the systematic component at design time (e.g., by adding appropriate delays or making the density uniform by adding “dummy fills”). Figure 10-16 shows the spatial pattern density and ILD thickness for a

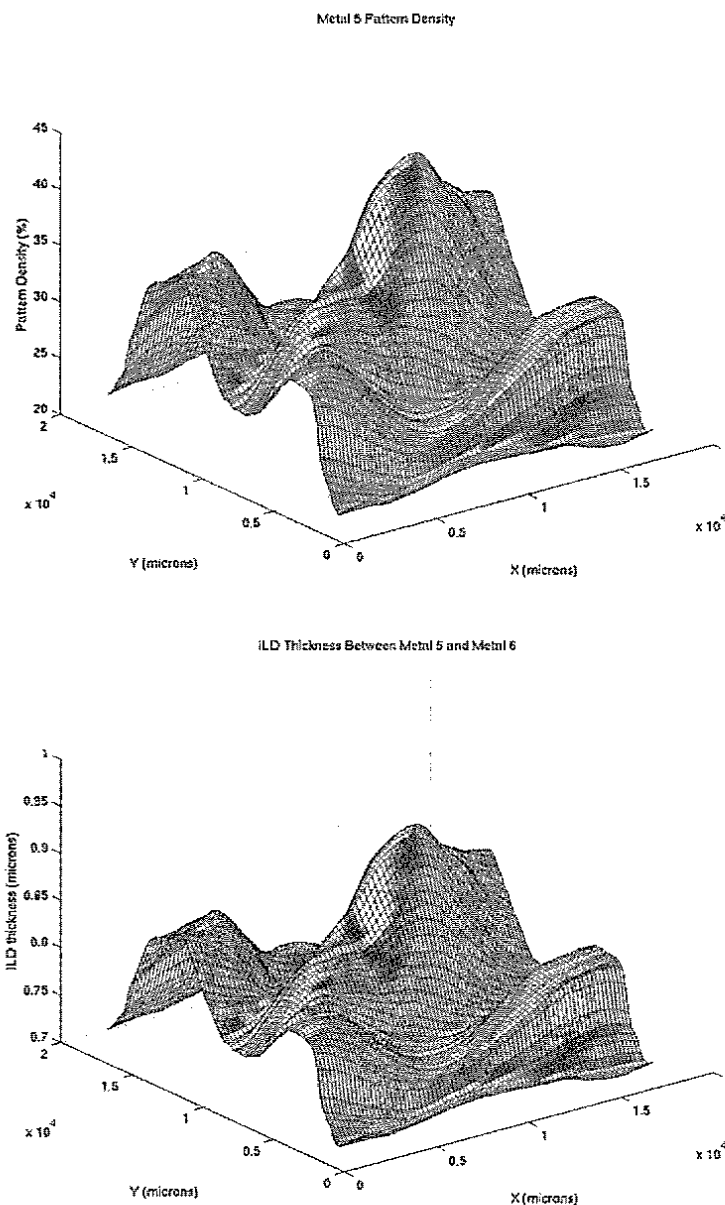


**Figure 10-16** Pattern density and ILD thickness variation for a high-performance microprocessor. (Courtesy of Duane Boning)

high-performance microprocessor. The graphs show a clear correlation between the density and the thickness of the dielectric. Hence, clock distribution networks must exploit such information in order to reduce clock skew.

Other interconnect variations include deviations in the width of the wires and line spacing, which result from photolithography and etch dependencies. At the lower levels of the metallization hierarchy, lithographic effects are more important, while etch effects that depend on width and layout are dominant at the higher levels. The width is a critical parameter because it directly impacts the resistance of the line, and the wire spacing affects the wire-to-wire capacitance. A detailed review of device and interconnect variations is presented in [Boning00]. Recent processors use copper interconnects, in which line thickness variations are also seen to be highly pattern dependent due to CMP dishing and erosion effects [Park00].

#### Environmental Variations (4 and 5)

Environmental variations probably are the most significant contributors to **skew and jitter**. The two major sources of environmental variations are *temperature* and *power supply*. Temperature gradients across the chip result from variations in power dissipation across the die. These gradients can be quite large, as shown in Figure 10-17, which displays a snapshot of the surface temperature of the DEC 21064 microprocessor. Temperature variation has become an important issue with *clock gating*, where some parts of the chip may be idle, while other parts of the chip are fully active. Clock gating has become popular in recent years as a means to minimize power dissipation in idle modules (as described in a later section). Shutting off parts of the chip leads to large temperature variations. Since the device parameters (such as threshold and mobility) depend strongly on temperature, the buffer delay for a clock distribution network can vary drastically from path to path. More importantly, this component is time varying, since the temperature changes as the logic activity of the circuit varies. Hence, it is not sufficient to simulate the clock networks at worst case corners of temperature; instead, the worst case variation in temperature must be simulated. An interesting question is whether temperature variation contributes to skew or to jitter. Clearly, the difference in temperature is time varying, but the changes are rela-

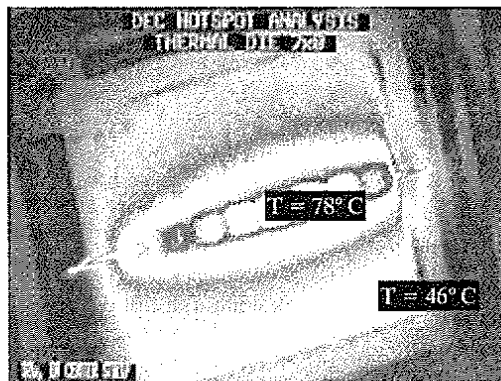


**Figure 10-17** Temperature variation (snapshot) over DEC 21064 microprocessor. The highest temperature occurs at the central clock driver [Herrick00].

tively slow (typical time constants for temperature changes are on the order of milliseconds). Therefore, it is usually considered as a skew component and the worst case conditions are used. Fortunately, by using feedback, it is possible to calibrate the temperature and to compensate for this effect.

Power-supply variations, on the other hand, are the major source of **jitter** in clock distribution networks. The delay through buffers is a very strong function of power supply, as it directly affects the drive of the transistors. As with temperature, the power-supply voltage is a strong function of the switching activity. Therefore, the buffer delay varies strongly from path to path. Power-supply variations can be classified into slow- (or static) and high-frequency variations. Static power-supply variations may result from fixed currents drawn from various modules, while high-frequency variations result from instantaneous  $IR$  drops along the power grid due to fluctuations in switching activity. Inductive effects on the power supply also are a major concern since they cause voltage fluctuations. Again, clock gating has exacerbated this problem, because the logic transitions between the idle and active states can cause major changes in current drawn from the supply. Since the power supply can change rapidly, the period of the clock signal is modulated on a cycle-by-cycle basis, resulting in jitter. The jitter on two different clock points may be correlated or uncorrelated, depending on how the power network is configured and the profile of switching patterns. Unfortunately, high-frequency power-supply changes are difficult to compensate for, even with feedback techniques. Consequently, **power-supply noise fundamentally limits the performance of clock networks**. To minimize power-supply variations, high-performance designs add decoupling capacitance around major clock drivers.

### Capacitive Coupling (6 and 7)

Changes in capacitive load also contribute to timing uncertainty. There are two major sources of capacitive-load variations: coupling between the clock lines and adjacent signal wires, and variation in gate capacitance. The clock network includes both the interconnect and the gate capacitance of latches and registers. Any coupling between the clock wire and adjacent signal results in timing uncertainty. Since the adjacent signal can transition in arbitrary directions and at arbitrary times, the exact coupling to the clock network is not fixed from cycle to cycle, causing **jitter**. Another major source of clock uncertainty is the variation in the gate capacitance contributed by the connecting sequential elements. The load capacitance is highly nonlinear and depends on the applied voltage. For many latches and registers, the clock load is a function of the current state of the latch/register (i.e., the values stored on the internal nodes of the circuit), as well as the next state. This causes the delay through the clock buffers to vary from cycle to cycle, which causes jitter.

---

### Example 10.2 Data-Dependent Clock Jitter

Consider the circuit shown in Figure 10-18, where a minimum-sized local clock buffer drives a register. (Actually, each clock buffer drives four registers, though only one is shown here.) The simulation shows *CKb*, the output of the first inverter for four possible

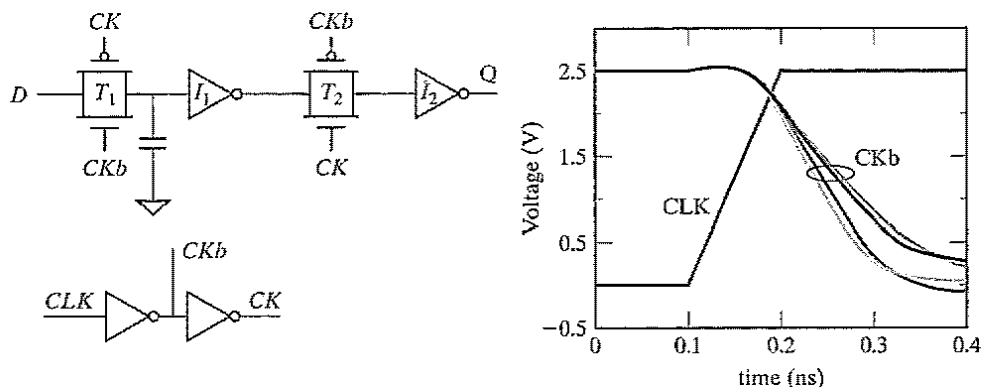


**Figure 10-18** Impact of data-dependent clock load on clock jitter for transmission-gate register.

transitions ( $0 \rightarrow 0$ ,  $0 \rightarrow 1$ ,  $1 \rightarrow 0$  and  $1 \rightarrow 1$ ). The jitter on the clock based on data-dependent capacitance is illustrated. In general, the only way to deal with this problem is to use registers that do not exhibit a large variation in load as a function of data—for example, the differential sense-amplifier register shown in Chapter 7.

### 10.3.3 Clock-Distribution Techniques

It is clear from the previous discussion that clock skew and jitter are major issues in digital circuits, and can fundamentally limit the performance of a digital system. It is therefore necessary to design a clock network that minimizes both. While designing that clock network, a close eye should be kept on the associated power dissipation. In most high-speed digital processors, a majority of the power is dissipated in the clock network. To reduce power dissipation, clock networks must support clock conditioning—that is, the ability to shut down parts of the clock network. Unfortunately, clock gating results in additional clock uncertainty (as described earlier).

In this section, an overview of basic constructs in high-performance clock distribution techniques is presented, along with a case study of clock distribution in the Alpha microprocessors. There are many degrees of freedom in the design of a clock network, including the type of material used for wires, the basic topology and hierarchy, the sizing of wires and buffers, the rise and fall times, and the partitioning of load capacitances.

#### Fabrics for Clocking

Clock networks typically include a network that is used to distribute a **global reference** to various parts of the chip, and a final stage that is responsible for **local distribution** of the clock while considering the local load variations. Most clock distribution schemes exploit the fact that the absolute delay from a central clock source to the clocking elements is irrelevant—only the relative phase between two clocking points is important. Therefore, one common approach to distributing a clock is to use balanced paths (called *trees*).

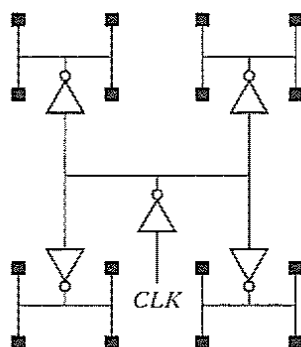


**Figure 10-19** Example of an *H*-tree clock-distribution network for 16 leaf nodes.

The most common type of clock distribution scheme is the *H-tree network*, which is illustrated in Figure 10-19 for a  $4 \times 4$  processor array. The clock is first routed to a central point on the chip. Balanced paths that include both matched interconnect and buffers then distribute the reference to the various leaf nodes. Ideally, if each path is perfectly balanced, the clock skew is zero. Although it might take multiple clock cycles for a signal to propagate from the central point to each leaf node, the arrival times are identical at every leaf node. However, in reality, process and environmental variations cause clock skew and jitter to occur.

The *H-tree* configuration is particularly useful for regular array networks in which all elements are identical and the clock can be distributed as a binary tree (for example, arrays of identical tiled processors). The concept can be generalized to a more generic setting. The more general approach, referred to as *matched RC trees*, represents a floor plan that distributes the clock signal so that the interconnections carrying the clock signals to the functional subblocks are of equal length. That is, the general approach does not rely on a regular physical structure. An example of a matched *RC* is shown in Figure 10-20. The chip is partitioned into 10 balanced load segments (tiles). The global clock driver distributes the clock to the tile drivers located at the dots in the figure. A lower level *RC*-matched tree is used to drive 580 additional drivers inside each tile. A 3D visualization of the clock delay in a tree network is shown in Figure 10-21.

An alternative clock distribution approach is the grid structure of Figure 10-22 [Bailey00]. Grids typically are used in the final stage of a clock network to distribute the clock to the clocking element loads. This approach is fundamentally different from the balanced *RC* approach. The main difference is that the delay from the final driver to each load is not matched. Rather, the absolute delay is minimized, assuming that the grid size is small. A major advantage of such a grid structure is that it allows for late design changes, since the clock is easily accessible at various points on the die. Unfortunately, the penalty is a relatively large power dissipation since the structure has a lot of “excess” interconnect. In addition to the schemes described earlier, other approaches have been devised for clock distribution. The length-matched serpentine approach is just one of them [Young97].

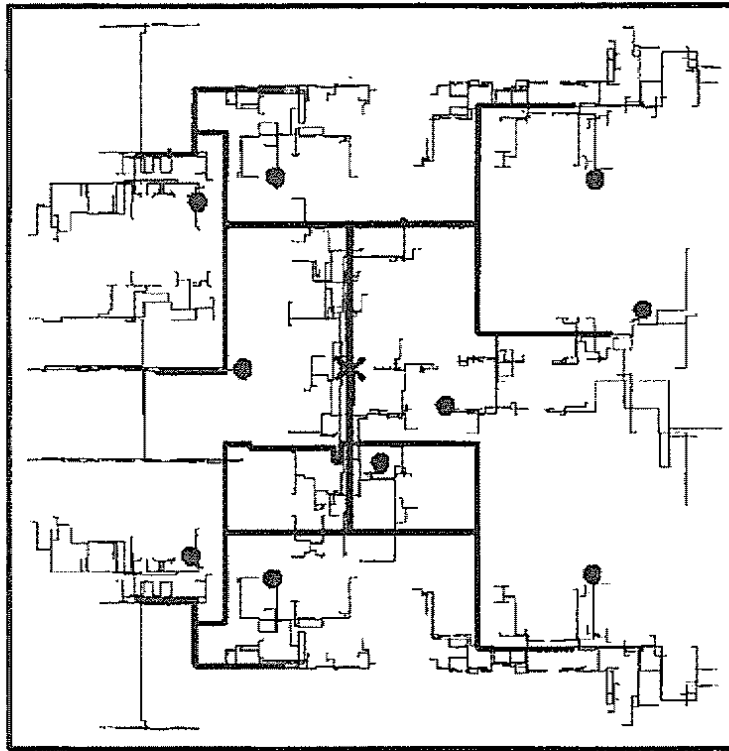


**Figure 10-20** An example RC-matched distribution for an IBM microprocessor [Restle98].

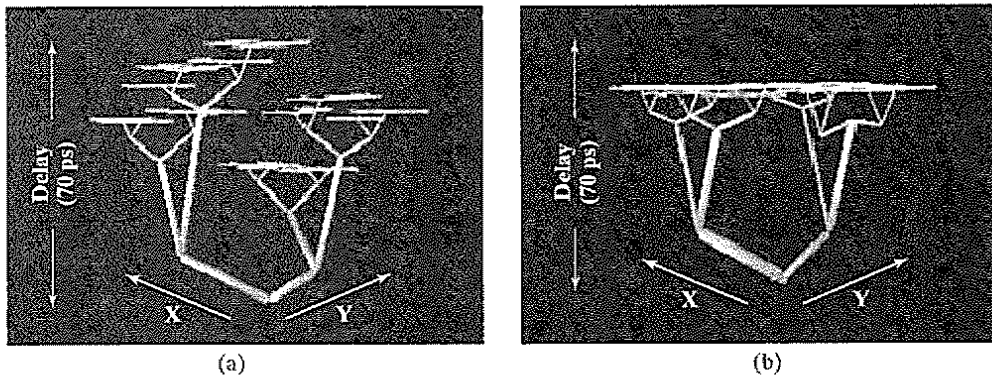


**Figure 10-21** Visualization of clock delay in a tree network driving different loads. The X- and Y-axes represent the die, while the Z-axis represents the clock delay. The width of the lines is proportional to the designed wire width. The unbalanced load creates a large skew, as is clear in (a). By careful tuning of the wire widths, the load is balanced, minimizing the skew, as shown in (b) [Restle01]. See also Colorplate 9.

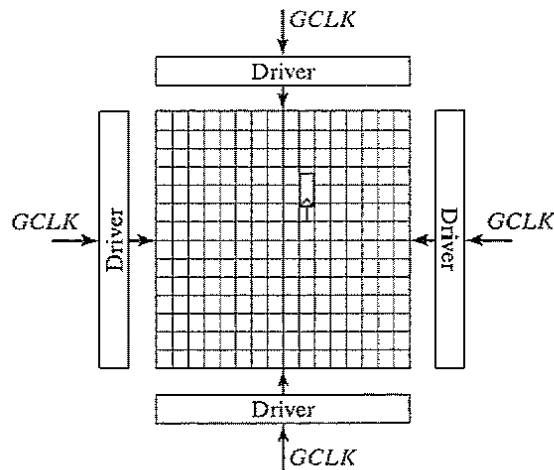


**Figure 10-22** Grid structures allow a low skew distribution and physical design flexibility at the cost of power dissipation [Bailey00].

It is essential to consider clock distribution in the earlier phases of the design of a complex circuit, since it might influence the shape and form of the chip's floor plan. It is tempting for a designer to ignore the clock network in the early phases of a project and consider it only at the end of the design cycle, when most of the chip layout is already frozen. This results in unwieldy clock networks and multiple timing constraints that hamper the performance and operation of the final circuit. With careful planning, a designer can avoid many of these problems, and clock distribution becomes a manageable operation.

#### Clock Distribution Case Study—The Digital Alpha Microprocessors

In this section, the clock distribution strategies for three generations of the Alpha microprocessor are discussed in detail. These processors have always been at the cutting edge of the technology and therefore represent an interesting perspective on the evolution of clock distribution [Herrick00].

**The Alpha 21064 Processor**—The first-generation Alpha microprocessor (21064 or EV4) from Digital Equipment Corporation used a single global clock driver [Dobberpuhl92]. The distribution of clock load capacitance among various functional blocks is shown in Figure 10-23. The total clock load equals 3.25 nF! The processor uses a single-phase clock methodology, and the 200-MHz clock is fed to a binary tree with five levels of buffering. The inputs to the clock drivers are shorted to smooth out the asymmetry in the incoming signals. The final output stage, residing in the middle of the chip, drives the clock net. The clock driver and the associated pre-drivers account for 40% of the effective switched capacitance (12.5 nF), resulting in significant power dissipation. The overall width of the clock driver was on the order of 35 cm in a 0.75- $\mu$ m technology. A detailed clock skew simulation with process variations indicates that a clock uncertainty of less than 200 ps (< 10%) was achieved.

**The Alpha 21164 Processor**—The second-generation Alpha microprocessor (EV5) operates at a clock frequency of 300 MHz while using 9.3 million transistors on a 16.5 mm.  $\times$  18.1 mm die in a 0.5- $\mu$ m CMOS technology [Bowhill95]. A single-phase clocking methodology was selected, and the design made extensive use of dynamic logic, resulting in a substantial clock load of 3.75 nF. The clock distribution system consumes 20 W, which is 40% of the total dissipation of the processor.

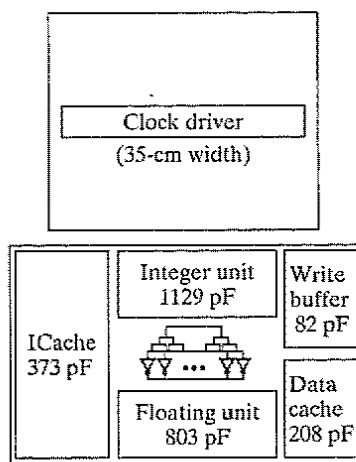
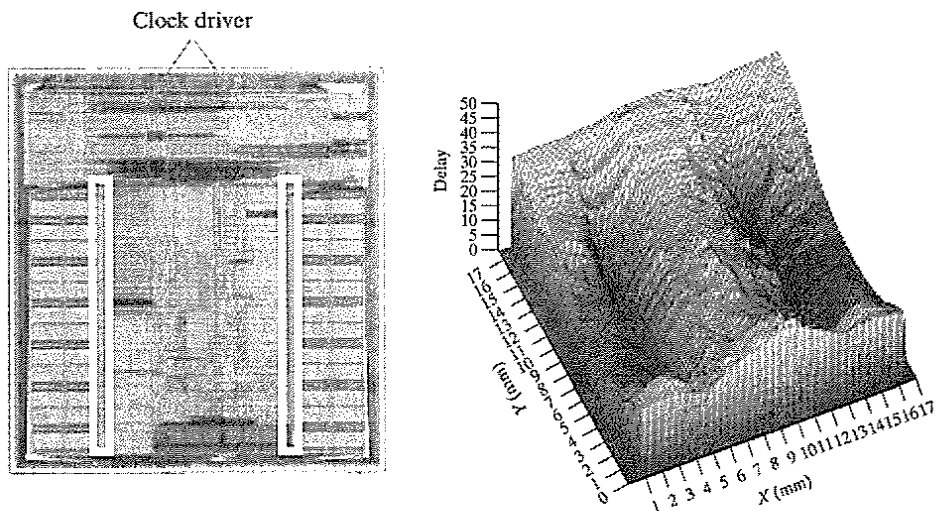


**Figure 10-23** Distribution of clock load capacitance for the 21064 Alpha processor.

The incoming clock signal is first routed through a single six-stage buffer placed at the center of the chip. The resulting signal is distributed in metal-3 to the left and right banks of final clock drivers, positioned between the secondary cache memory and the outside edge of the execution unit (see Figure 10-24a). The produced clock signal is driven onto a grid of metal-3 and metal-4 wires. The equivalent transistor width of the final driver inverter equals 58 cm! To ensure the integrity of the clock grid across the chip, the grid was extracted from the layout, and the resulting RC-network was simulated. A



(a) Chip microphotograph, showing positioning of clock drivers.

(b) Clock skew simulation.

**Figure 10-24** Clock distribution and skew in a 300-MHz microprocessor. (Courtesy of Digital Equipment Corporation.)

three-dimensional representation of the simulation results is plotted in Figure 10-24b. As evident from the plot, the skew is zero at the output of the left and right drivers. The maximum value of the absolute skew is smaller than 90 ps. The critical instruction and execution units all see the clock within 65 ps.

Clock skew and race problems were addressed using a “mix-and-match” approach. The clock skew problems were eliminated by either routing the clock in the opposite direction of the data (at a small cost in terms of performance) or by ensuring that the data could not overtake the clock. A standardized library of level-sensitive transmission-gate latches was used for the complete chip. To avoid race-through conditions, the following design guidelines were used:

- Careful sizing of the local clock buffers were carefully sized so that their skew was minimal.
- At least one gate had to be inserted between connecting latches. This gate, which can be part of the logic function or just a simple inverter, ensures was a minimum contamination delay. Special design verification tools were developed to guarantee that this rule was obeyed over the complete chip.

To improve the interlayer dielectric uniformity, filler polygons were inserted between widely spaced lines, as shown in Figure 10-25. Though this may increase the capacitance to nearby signal lines, the improved uniformity results in lower variation and clock uncertainty. The dummy fills are automatically inserted, and tied to one of the power rails ( $V_{DD}$  or GND). Dummy insertion is required today to equalize the etch-away of CMP (Chapter 3). This technique is used in many of today’s processes for controlling the clock skew.

This example demonstrates that managing clock skew and clock distribution for large, high-performance synchronous designs is a feasible task. However, making such a circuit work in a reliable way requires careful planning and intensive analysis.

**The Alpha 21264 Processor**—A hierarchical clocking scheme is used in the 600-Mhz Alpha 21264 (EV6) processor (in 0.35- $\mu\text{m}$  CMOS), shown in Figure 10-26 [Bailey98]. The choice of a hierarchical clocking scheme for this processor is a major departure from the preceding processors, which did not have

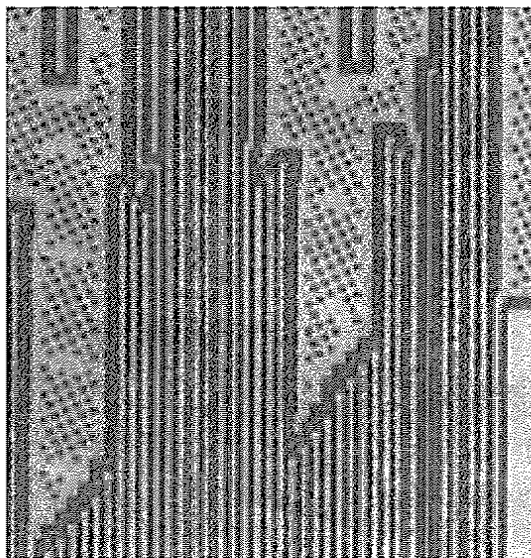


**Figure 10-25** Dummy fills reduce the ILD variation and improve clock skew.

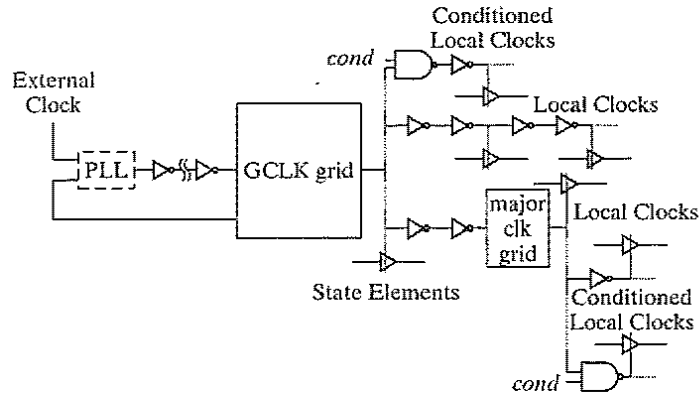


**Figure 10-26** Clock hierarchy for the Alpha 21264 Processor.

a hierarchy of clocks beyond the global clock grid. Using a hierarchical clocking approach enables trade-offs between power and skew management. Power is reduced, because the clocking networks for individual blocks can be gated. As seen in previous-generation microprocessors, the clock power contributes to a large fraction of overall power consumption. Also, the flexibility of having local clocks provides the designers with more freedom with respect to circuit styles at the module level. The drawback of using a hierarchical clock network is that skew reduction becomes more difficult. Clocks to various local registers may go through very different paths, which may contribute to the skew. However, by using timing verification tools, the skew can be managed by tweaking of the clock drivers.

The clock hierarchy consists of a *global clock grid*, called *GCLK*, that covers the entire die. State elements and clocking points exist from zero to height levels past *GCLK*. The on-chip generated clock is routed to the center of the die and distributed using tree structures to 16 distributed clock drivers (see Figure 10-27). The global clock distribution network utilizes a windowpane configuration, which achieves low skew by dividing the clock into four regions, which reduces the distance from the drivers to the loads. Each grid pane is driven from four sides, reducing the dependence on process variations. This also helps the power supply and thermal problems, as the drivers are distributed through the chip.

The use of a gridded clock has the advantage of reducing the clock skew, while providing universal availability of the clock signals. The drawback is the increased capacitance of *GCLK*, compared with a tree-distribution approach. In addition to the *GCLK*, at the next level of clock hierarchy, there is a major clock grid. The major clocks were introduced to reduce power: They have localized loads, and they can be sized appropriately to meet the skew and edge requirements for the local loading conditions at timing-critical units.

The lowest level in this hierarchy is formed by the *local clocks*, which are generated as needed from any other clock. Typically, they can be customized to meet local timing constraints. The local clocks provide great flexibility in the design of the logic blocks, but at the same time make it significantly more difficult to manage skew. Furthermore, the local clocks are more susceptible to coupling from data lines, because they are not shielded like the global gridded clocks. As a result, the local clock distribution is highly dependent on its local interconnection and thus has to be designed very carefully.

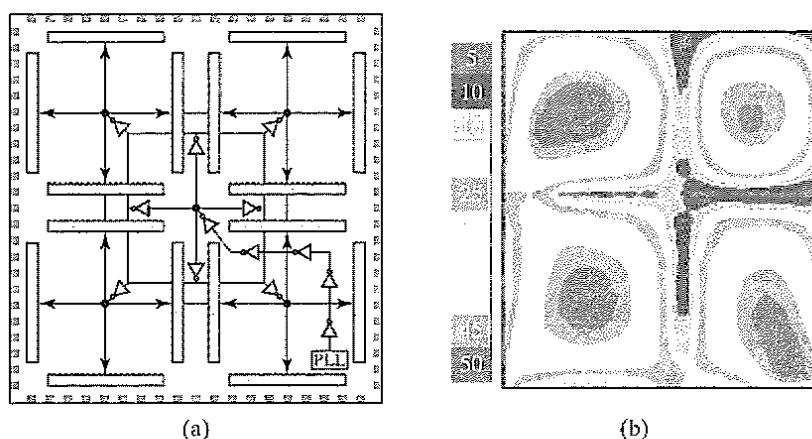


**Figure 10-27** Global clock distribution network in a windowpane structure. Pane structure (a); clock skew distribution over die (b).

#### Design Techniques—Dealing with Clock Skew and Jitter

From the preceding discussions, some useful guidelines for reducing clock skew and jitter can be derived:

1. To minimize skew, *balance clock paths* from a central distribution source to individual clocking elements, using *H-tree* structures or more generally routed matched-tree structures. When using routed clock trees, the effective clock load of each path that includes wiring as well as transistor loads must be equalized.
2. The use of *local clock grids* (instead of routed trees) can reduce skew at the cost of increased capacitive load and power dissipation.
3. If data-dependent clock load variations cause significant jitter, *differential registers* that have a data-independent clock load should be used. The use of gated clocks to save power results in a data-dependent clock load and increased jitter. In clock networks where the fixed load is large (e.g., in clock grids), the data-dependent variation might not be significant.
4. If data flow in one direction, route the *data and the clock in opposite directions*. This eliminates races at the cost of performance.
5. Avoid data-dependent noise by *shielding clock wires* from adjacent signal wires. By placing power lines ( $V_{DD}$  or GND) next to the clock wires, coupling from neighboring signal nets can be minimized or avoided.
6. Variations in interconnect capacitance due to interlayer dielectric thickness variation can be greatly reduced through the use of *dummy fills*. Dummy fills are very common and reduce skew by increasing uniformity. Systematic variations should be modeled and compensated for.
7. Variation in chip temperature across the die causes variations in clock buffer delay. The use of *feedback circuits based on delay-locked loops*, discussed later in this chapter, can compensate for temperature variations.
8. Power-supply variation is a significant component of jitter, as it impacts the cycle-to-cycle delay through clock buffers. High-frequency power-supply variation can be reduced by adding *on-chip decoupling capacitors*. Unfortunately, decoupling capacitors require a significant amount of area, and efficient packaging solutions must be leveraged in order to reduce chip area.

### 10.3.4 Latch-Based Clocking\*

The use of registers in a sequential circuit translates into a robust, reliable design methodology. Yet there are significant performance advantages to be made through the use of a latch-based design style, in which combinational logic is separated by transparent latches. In an edge-triggered system, the worst case logic path between two registers determines the minimum clock period for the entire system. If a logic block finishes before the end of the clock period, it has to sit idle until the next clock edge. The use of a latch-based methodology enables more flexible timing by allowing one stage to pass slack or to borrow time from other stages. The added flexibility also increases overall performance. Note that the latch-based methodology can be thought of as adding logic between latches of master-slave flip-flops.

Consider the latch-based system of Figure 10-28, which uses a two-phase clocking scheme. Assume that the clocks are ideal and that the two clocks are inverted versions of each other (for the sake of simplicity). In this configuration, a stable input is available to the combinational logic block A ( $CLB\_A$ ) on the falling edge of  $CLK_1$  (at edge ②). It has a maximum time equal to the  $T_{CLK}/2$  to evaluate—that is, the entire low phase of  $CLK_1$ . On the falling edge of  $CLK_2$  (at edge ③), the output  $CLB\_A$  is latched, and the computation of  $CLB\_B$  is launched.  $CLB\_B$  computes on the low phase of  $CLK_2$ , and the output is available on the falling edge of  $CLK_1$  (at edge ④). From a timing perspective, this scenario appears to be equivalent to an edge-triggered system, where  $CLB\_A$  and  $CLB\_B$  are cascaded between two edge-triggered registers (see Figure 10-29). In both cases, it appears that the time available to perform the combination of  $CLB\_A$  and  $CLB\_B$  is  $T_{CLK}$ .

However, there is an important performance-related difference. In a latch-based system, it is possible for a logic block to utilize time that is left over from the previous logic block. This phenomenon, which results from the transparency of a latch during its on time, is referred to as *slack borrowing* [Bernstein98]. It requires no explicit design changes, as the passing of slack

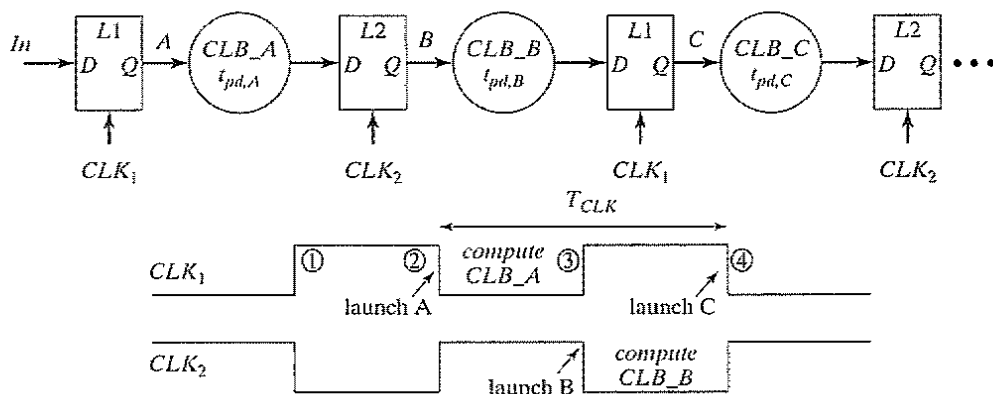


**Figure 10-28** Latch-based design in which transparent latches are separated by combinational logic.

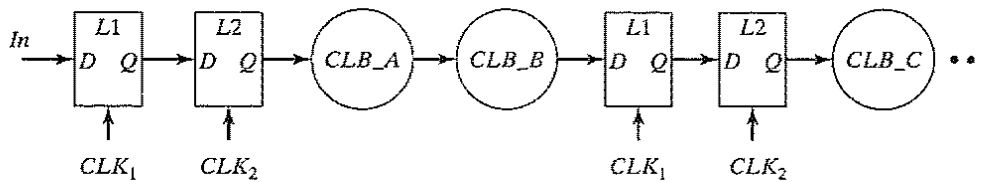


**Figure 10-29** Edge-triggered pipeline (back-to-back latches for edge-triggered registers) of the logic in Figure 10-28.

from one block to the next is automatic. The key advantage of slack borrowing is that it allows logic between cycle boundaries to use more than one clock cycle while satisfying the overall cycle time constraint. Stated in another way, if the sequential system works at a particular clock rate, and the total logic delay for a complete cycle is larger than the clock period, then unused time, or *slack*, has been implicitly borrowed from preceding stages. Formally stated, slack passing has taken place if  $T_{CLK} < t_{pd,A} + t_{pd,B}$ , and the logic functions correctly (for simplicity, the delay associated with latches are ignored). This implies that the clock rate can be higher than the worst case critical path!

As mentioned earlier, slack passing results from the level-sensitive nature of the latches. In Figure 10-28, the input to *CLB\_A* must be valid by the falling edge of  $CLK_1$  (edge ②). What happens if the combinational logic block of the previous stage finishes early and has a valid input data for *CLB\_A* before edge ②? Since the latch is transparent during the entire high phase of the clock, the input data for *CLB\_A* are valid as soon as the previous stage has finished computing. This implies that the maximum time available for *CLB\_A* is its phase time (i.e., the low phase of  $CLK_1$ ) plus any time left from the previous computation.

Consider the latch-based system of Figure 10-30. In this example, signal *a* (input to *CLB\_A*) is valid well before edge ②. This implies that the previous block did not use its entire allotment, producing slack time as denoted by the shaded area. By construction, *CLB\_A* can start computing as soon as signal *a* becomes valid. It uses this slack time to finish well before its allocated time (edge ③). Since  $L_2$  is a transparent latch, valid on the high phase of  $CLK_2$ , *CLB\_B* starts to compute using the slack provided by *CLB\_A*. Again, *CLB\_B* completes before its allocated time (edge ④), and it passes a small amount of slack to the next cycle. As this picture indicates, the total cycle delay—that is, the sum of the delay for *CLB\_A* and *CLB\_B*—is larger than the clock period. Since the pipeline behaves correctly, slack passing has taken place and a higher throughput has been achieved.

An important question related to slack passing involves to the maximum possible slack that can be passed across cycle boundaries. In Figure 10-30, it is easy to see that the earliest time that *CLB\_A* can start computing is ①. This happens if the previous logic block did not use any of its allocated time ( $CLK_1$  high phase) or if it finished by using slack from previous stages. Therefore, the maximum time that can be borrowed from the previous stage is half of a cycle or  $T_{CLK}/2$ . Similarly, *CLB\_B* must finish its operation by edge ④. This implies that

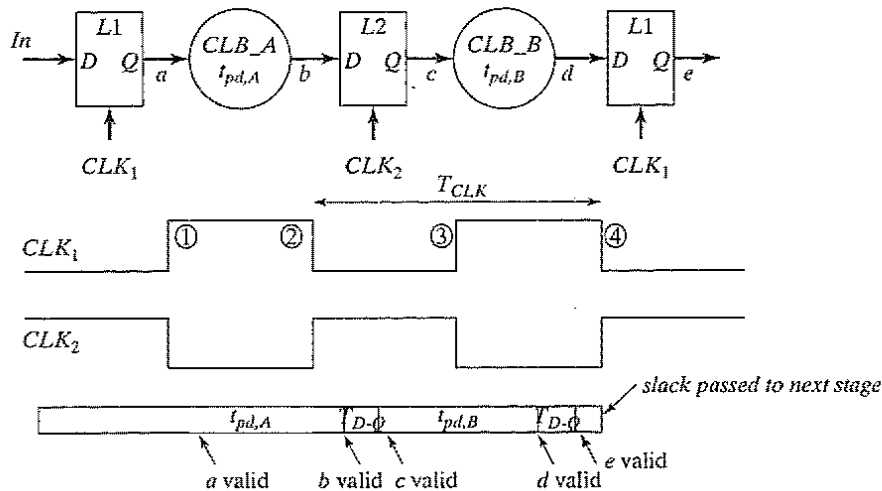


Figure 10-30 Example of slack borrowing.

the maximum logic cycle delay is equal to  $1.5 \times T_{CLK}$ . However, note that for an  $n$ -stage pipeline, the overall logic delay cannot exceed the time available of  $n \times T_{CLK}$ .

### Example 10.3 Slack-Passing Example

First, consider the negative edge-triggered pipeline of Figure 10-31. Assume that the primary input  $In$  is valid slightly after the rising edge of the clock. We can derive that the minimum clock period required is 125 ns. The latency is two clock cycles. (Actually, the output is valid 2.5 cycles after the input settles.) Note that for the first pipeline stage, one-half cycle is wasted, as the input data are available only to  $CL_1$  after the falling edge of the clock. This time can be exploited in a latch-based system.

Figure 10-32 shows a latch-based version of the same circuit. As the timing indicates, exactly the same timing can be achieved with a clock period of 100 ns. This is enabled by slack borrowing between the logical partitions.

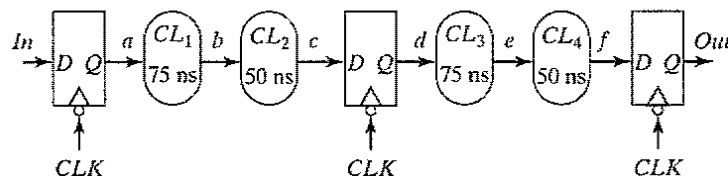


Figure 10-31 Conventional edge-triggered pipeline.

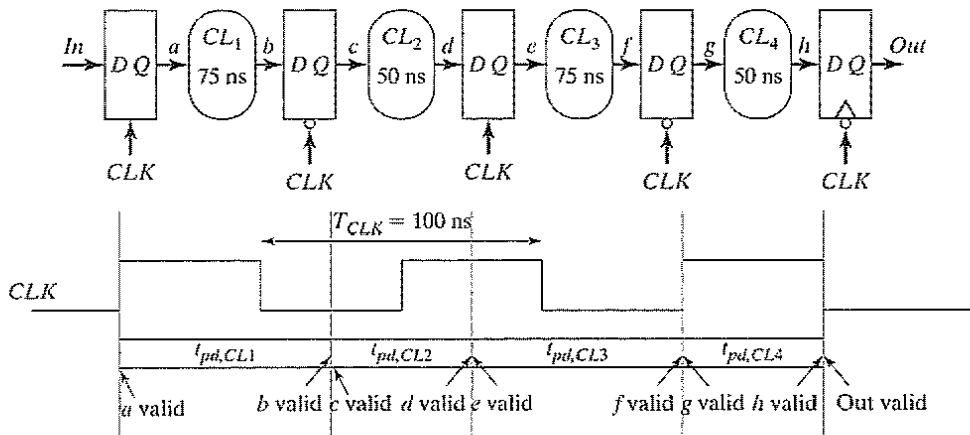


Figure 10-32 Latch-based pipeline.

If slack passing or borrowing intrigues you, we refer you to [Bernstein98], which presents an excellent quantitative analysis of this attractive, yet challenging, approach.

## 10.4 Self-Timed Circuit Design\*

### 10.4.1 Self-Timed Logic—An Asynchronous Technique

The synchronous design approach advocated in the previous sections assumes that all circuit events are orchestrated by a central clock. Those clocks have a dual function:

- They ensure that the *physical timing constraints* are met. The next clock cycle can start only when all logic transitions have settled and the system has come to a steady state. This ensures that only legal logical values are applied in the next round of computation. In short, clocks account for the worst case delays of logic gates, sequential logic elements, and the wiring.
- Clock events serve as a *logical ordering mechanism* for the global system events. A clock provides a time base that determines what will happen and when. On every clock transition, a number of operations are initiated that change the state of the sequential network.

Consider the pipelined datapath of Figure 10-33. In this circuit, the data transitions through logic stages under the command of the clock. The important point is that the clock period is chosen to be larger than the worst case delay of each pipeline stage, or  $T > \max(t_{pd1}, t_{pd2}, t_{pd3}) + t_{pd,reg}$ . This will ensure satisfaction of the physical constraint. At each clock transition, a new set of inputs is sampled, and computation is started again. The throughput of the system—which is equivalent to the number of data samples processed per second—is equivalent to the clock rate. The time at which to sample a new input and the availability of an output depend upon the *logical ordering* of the system events and clearly are orchestrated by the clock in this example.

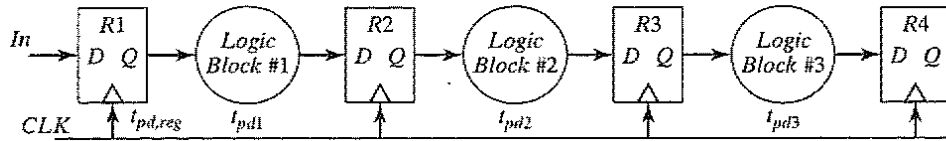


Figure 10-33 Pipelined, synchronous datapath.

The synchronous design methodology has some clear advantages. It presents a structured, deterministic approach to the problem of choreographing the myriad of events that take place in digital designs. The approach taken is to equalize the delays of all operations by making them as bad as the worst of the set. The approach is robust and easy to adhere to, which explains its enormous popularity; however, it does have several pitfalls:

- It assumes that all clock events or timing references happen simultaneously over the complete circuit. This is not the case in reality, because of effects such as clock skew and jitter.
- As all the clocks in a circuit transition at the same time, significant current flows over a very short period of time (due to the large capacitance load). This causes significant noise problems due to package inductance and power-supply grid resistance.
- The linking of physical and logical constraints has some obvious effects on the performance. For instance, the throughput rate of the pipelined system of Figure 10-33 is directly linked to the worst case delay of the slowest element in the pipeline. On average, the delay of each pipeline stage is smaller. The same pipeline could support an average throughput rate that is substantially higher than the synchronous one. For example, the propagation delay of a 16-bit adder is highly data dependent: Adding two 4-bit numbers requires a much shorter time compared with adding two 16-bit numbers.

One way to avoid these problems is to opt for an *asynchronous* design approach and to eliminate all the clocks. Designing a purely asynchronous circuit is a nontrivial and potentially hazardous task. Ensuring a correct circuit operation that avoids all potential race conditions under any operation condition and input sequence requires a careful timing analysis of the network. In fact, the logical ordering of the events is dictated by the structure of the transistor network and the relative delays of the signals. Enforcing timing constraints by manipulating the logic structure and the lengths of the signal paths requires an extensive use of CAD tools and is recommended only when strictly necessary.

A more reliable and robust technique is the self-timed approach, which presents a local solution to the timing problem [Seitz80]. Figure 10-34 uses a pipelined datapath to illustrate how this can be accomplished. It is assumed that each combinational function has a means of indicating that it has completed a computation for a particular piece of data. The computation of a logic block is initiated by asserting a *Start* signal. The combinational logic block computes on the input data, and in a data-dependent fashion (taking the physical constraints into account), generates a *Done* flag once the computation is finished. In addition, the operators must signal each other that either they are ready to receive a next input word or they have a legal data word at

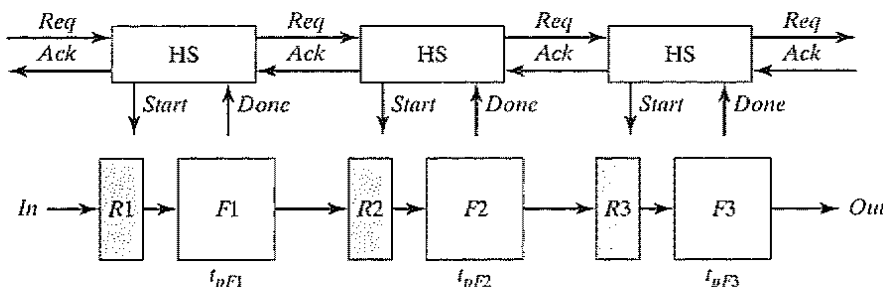


Figure 10-34 Self-timed, pipelined datapath.

their outputs that is ready for consumption. This signaling ensures the logical ordering of the events and can be achieved with the aid of extra *Ack(nowledge)* and *Req(uest)* signals. In the case of the pipelined datapath, the scenario could proceed as follows:

1. An input word arrives, and a *Req(uest)* to the block *F1* is raised. If *F1* is inactive at the time, it transfers the data and acknowledges this fact to the input buffer, which can go ahead and fetch the next word.
2. *F1* is enabled by raising the *Start* signal. After a certain amount of time, which is dependent upon the data values and operating conditions, the *Done* signal goes high, indicating the completion of the computation.
3. A *Req(uest)* is issued to the *F2* module. If this function is free, an *Ack(nowledge)* is raised, the output value is transferred, and *F1* can go ahead with its next computation.

The self-timed approach effectively separates the physical and logical ordering functions implied in circuit timing. The completion signal *Done* ensures that the physical timing constraints are met and that the circuit is in steady state before accepting a new input. The logical ordering of the operations is ensured by the acknowledge–request scheme, often called a *handshaking protocol*. Both interested parties synchronize with each other by mutual agreement (or, if you want, by “shaking hands”). The ordering protocol described previously and implemented in the module HS is only one of many that are possible. The choice of protocol is important, since it has a profound effect on the circuit performance and robustness.

When compared with the synchronous approach, self-timed circuits display some alluring properties:

- In contrast to the global centralized approach of the synchronous methodology, timing signals are generated *locally*, which avoids all problems and overheads associated with distributing high-speed clocks.
- Separating the physical and logical ordering mechanisms results in a potential increase in performance. In synchronous systems, the period of the clock has to be stretched to accommodate the slowest path over all possible input sequences. In self-timed systems, a

completed data word does not have to wait for the arrival of the next clock edge in order to proceed to the subsequent processing stages. Since circuit delays often are dependent on the actual data value, a self-timed circuit proceeds at the *average speed* of the hardware, in contrast to the *worst case* model of synchronous logic. For a ripple-carry adder, the average length of carry propagation is  $O(\log(N))$ . This is a fact that can be exploited in self-timed circuits, while in a synchronous methodology, a worst case performance that varies linearly with the number of bits ( $O(N)$ ) must be assumed.

- The automatic shutdown of blocks that are not in use can result in power savings. In addition, the power consumption overhead of generating and distributing high-speed clocks can be partially avoided. As discussed earlier, this overhead can be substantial. The use of gated clocks in synchronous design yields similar results.
- Self-timed circuits are, by nature, robust regarding variations in manufacturing and operating conditions such as temperature. Synchronous systems are limited by their performance at the extremes of the operating conditions. The performance of a self-timed system is determined by the actual operating conditions.

Unfortunately, these general properties are not without cost—they come at the expense of a substantial circuit-level overhead, which is caused by the need to generate completion signals and the need for handshaking logic that acts as a local traffic agent to order the circuit events (see block HS in Figure 10-34). Both of these topics are treated in more detail in the subsequent sections.

### 10.4.2 Completion-Signal Generation

A necessary component of self-timed logic is the circuitry to indicate when a particular piece of circuitry has completed its operation for the current piece of data. There are two common and reliable ways to generate the completion signal.

#### Dual-Rail Coding

One common approach to completion-signal generation is the use of *dual-rail coding*. It actually requires the introduction of redundancy in the data representation in order to signal that a particular bit is in either a transition or a steady-state mode. Consider the redundant data model presented in Table 10-1. Two bits ( $B0$  and  $B1$ ) are used to represent a single data bit  $B$ . For the data

**Table 10-1** Redundant signal representation to include transition state.

| $B$                      | $B0$ | $B1$ |
|--------------------------|------|------|
| in transition (or reset) | 0    | 0    |
| 0                        | 0    | 1    |
| 1                        | 1    | 0    |
| illegal                  | 1    | 1    |

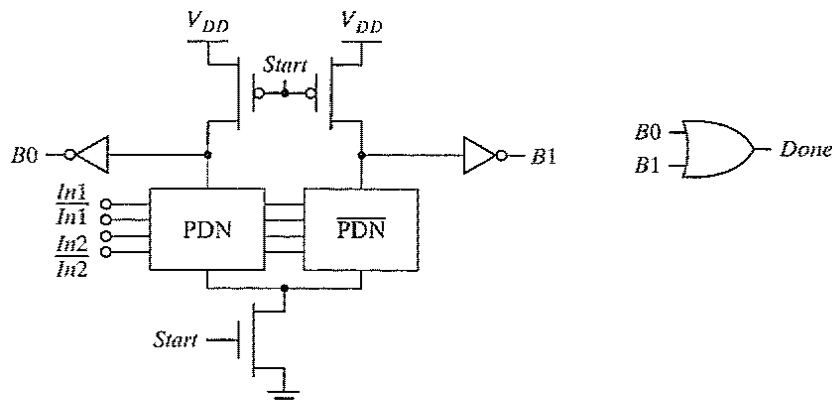


Figure 10-35 Generation of a completion signal in DCVSL.

to be valid or the computation to be completed, the circuit must be in a legal 0 ( $B0 = 0, B1 = 1$ ) or 1 ( $B0 = 1, B1 = 0$ ) state. The ( $B0 = 0, B1 = 0$ ) condition signals that the data are nonvalid and the circuit is in either a reset or a transition mode. The ( $B0 = 1, B1 = 1$ ) state is illegal and should never occur in an actual circuit.

A circuit that actually implements such a redundant representation is shown in Figure 10-35, which is a dynamic version of the DCVSL logic style where the clock is replaced by the *Start* signal [Heller84]. DCVSL uses a redundant data representation by nature of its differential dual-rail structure. When the *Start* signal is low, the circuit is precharged by the PMOS transistors, and the output ( $B0, B1$ ) goes in the *Reset-Transition* state (0, 0). When the *Start* signal goes high, signaling the initiation of a computation, the NMOS pull-down network evaluates, and one of the precharged nodes is lowered. Either  $B0$  or  $B1$ —but never both—goes high, which raises *Done* and signals the completion of the computation.

DCVSL is more expensive in terms of area than a nonredundant circuit due to its differential nature. The completion generation is performed in series with the logic evaluation, and its delay adds directly to the total delay of the logic block. The completion signals of all the individual bits must be combined to generate the completion for an  $N$ -bit data word. Completion generation thus comes at the expense of both area and speed. The benefits of the dynamic timing generation often justify this overhead.

Redundant signal representations other than the one presented in Table 10-1 can also be envisioned. One essential element is the presence of a *transition state* denoting that the circuit is in evaluation mode and the output data are not valid.

#### Example 10.4 Self-Timed Adder Circuit

An efficient implementation of a self-timed adder circuit is shown in Figure 10-36 [Abnous93]. A Manchester-carry scheme is used to boost the circuit performance. The proposed approach is based on the observation that the circuit delay of an adder is dominated by the carry-propagation path. It is therefore sufficient to use the differential

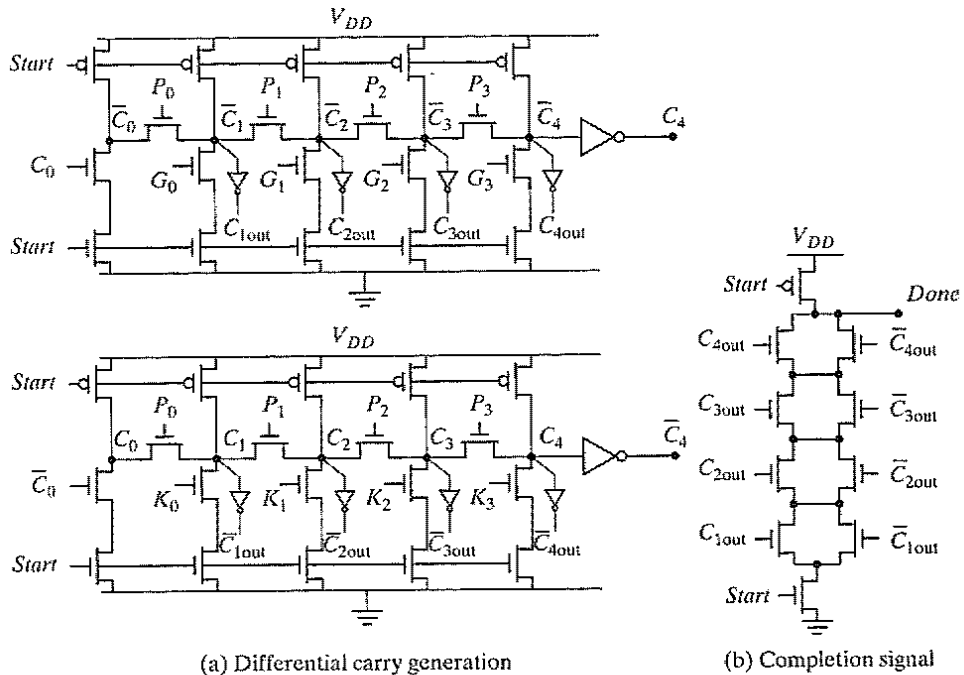


**Figure 10-36** Manchester-carry scheme with differential signal representation.

signaling in the carry path only (Figure 10-36a). The completion signal is efficiently derived by combining the carry signals of the different stages (Figure 10-36b). This safely assumes that the sum generation, which depends upon the arrival of the carry signal, is faster than the completion generation. The benefit of this approach is that the completion generation starts earlier and proceeds in parallel with sum generation, which reduces the critical timing path. All other signals, such as  $P$ (ropagate),  $G$ (enerate),  $K$ (ill), and  $S$ (um), do not require completion generation and can be implemented in single-ended logic. As shown in the circuit schematics, the differential carry paths are virtually identical. The only difference is that the  $G$ (enerate) signal is replaced by a  $K$ (ill). A simple logic analysis demonstrates that this indeed results in an inverted carry signal and, hence, a differential signaling.

### Replica Delay

While the dual-rail coding just described allows tracking of the signal statistics, it comes at the cost of power dissipation. Every single gate must transition for every new input vector, regardless of the value of the data vector. A way to reduce the overhead of completion detection is to use a *critical-path replica* configured as a delay element, as shown in Figure 10-37. To start a computation, the *Start* signal is raised and the computation of the logic network is initiated. At the same time, the start signal is fed into the replica delay line, which tracks the critical path of

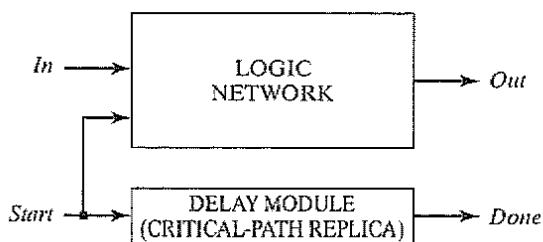


Figure 10-37 Completion-signal generation using delay module.

the logic network. It is important that the replica is structured such that no glitching transitions occur. When the output of the delay line makes a transition, it indicates that the logic is complete as the delay line mimics the critical path. In general, it is important to add extra padding in the delay line in order to compensate for possible random process variations.

The advantage of this approach is that the logic can be implemented using a standard non-redundant circuit style, such as complementary CMOS. Also, if multiple logic units are computing in parallel, it is possible to amortize the overhead of the delay line over multiple blocks. Note that this approach generates the completion signal after a time equal to the worst case delay through the network. As a result, it does not exploit the statistical properties of the incoming data. However, it can track the local effects of process variations and environmental variations (e.g., temperature or power-supply variations). This approach is widely used to generate the internal timing of semiconductor memories where self-timing is a commonly used technique.

#### Example 10.5 An Alternate Completion Detection Circuit Using Current Sensing

Ideally, logic should be implemented using nonredundant CMOS (e.g., static CMOS), and the completion signal circuitry should track data dependencies. Figure 10-38 shows an approach that attempts to realize this principle [Dean94]. A current sensor is inserted in series with the combinational logic, and it monitors the current flowing through the logic. The current sensor outputs a low value when no current flows through the logic (i.e., the logic is idle) and a high value when the combinational logic is switching. This signal

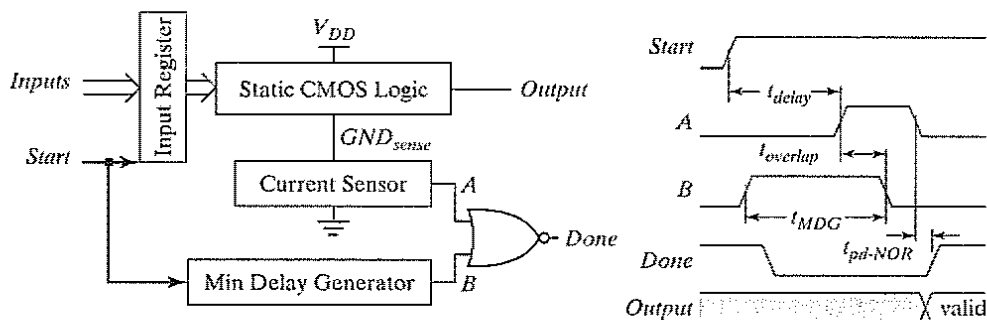


Figure 10-38 Completion-signal generation using current sensing.

effectively determines when the logic has completed its cycle. Note that this approach tracks data-dependent computation times—if only the lower order bits of an adder switch, current will stop flowing once the lower order bits switch to the final value. If the input data vector does not provoke any change in the logic from one cycle to the next, no current is drawn from the supply for static CMOS logic. In this case, a delay element that tracks the minimum delay through the logic and current sensor is used for signal completion. The outputs of the current sensor and minimum delay element are then combined.

This approach is interesting, but requires careful analog design. Ensuring reliability while keeping the overhead circuitry small is the main challenge. These concerns have kept the applicability of the approach very limited, despite its obvious potential.

### 10.4.3 Self-Timed Signaling

Besides the generation of the completion signals, a self-timed approach also requires a handshaking protocol to logically order the circuit events in order to avoid races and hazards. The functionality of the signaling (or handshaking) logic is illustrated by the example of Figure 10-39, which shows a *sender module* transmitting data to a *receiver* [Sutherland89]. The sender places the data value on the data bus ① and produces an event on the *Req* control signal by changing the polarity of the signal ②. In some cases, the request event is a rising transition; at other times, it is a falling one—the protocol described here does not distinguish between them. Upon receiving the request, the receiver accepts the data when possible and produces an event on the *Ack* signal to indicate that the data have been accepted ③. If the receiver is busy or its input buffer is full, no *Ack* event is generated, and the transmitter is stalled until the receiver becomes available by, for instance, freeing space in the input buffer. Once the *Ack* event is produced, the transmitter goes ahead and produces the next data word ④. The four events—*data change*, *request*, *data acceptance*, and *acknowledge*—proceed in a cyclic order. Successive cycles may take different amounts of time, depending on the time it takes to produce or consume the data.

This is called the *two-phase protocol*, since only two phases of operation can be distinguished for each data transmission—the active cycle of the sender and the active cycle of the receiver. Both phases are terminated by certain events. The *Req* event terminates the active cycle of the sender, while the receiver's cycle is completed by the *Ack* event. The sender is free to change the data during its active cycle. Once the *Req* event is generated, it has to keep the data constant as long as the receiver is active. The receiver can only accept data during its active cycle.

The correct operation of the sender–receiver system requires a strict ordering of the signaling events, as indicated by the arrows in Figure 10-39. Imposing this order is the task of the handshaking logic, which, in a sense, performs logic manipulations on events. An essential component of virtually any handshaking module is the *Muller C-element*. This gate, whose schematic symbol and truth table are given in Figure 10-40, performs an AND-operation on events. The output of the C-element is a copy of its inputs when both inputs are identical. When the inputs differ, the output retains its previous value. Phrased in a different way, events must occur

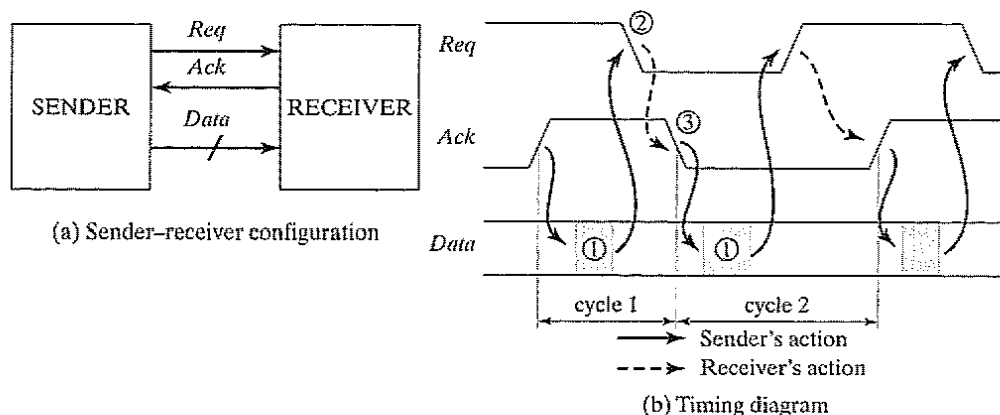


Figure 10-39 Two-phase handshaking protocol.

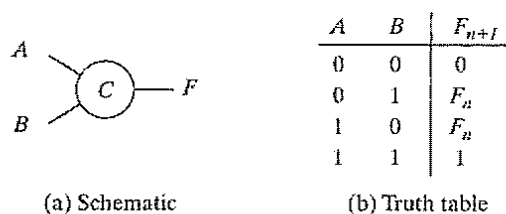


Figure 10-40 Muller C-element.

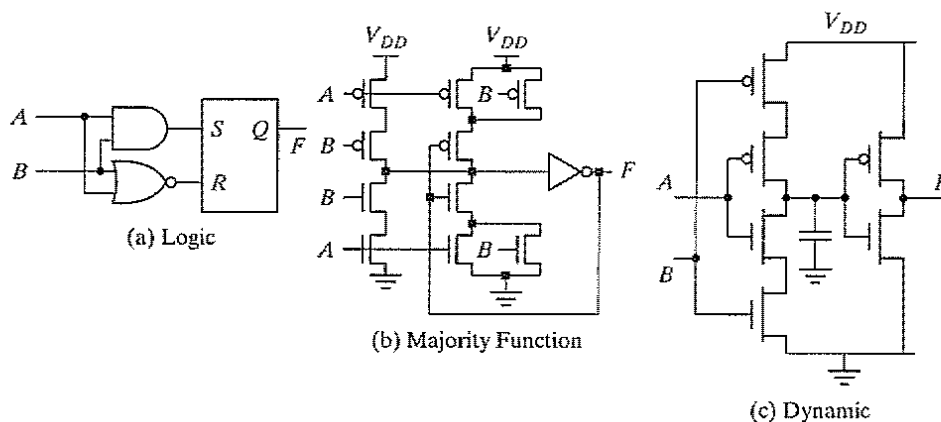


Figure 10-41 Implementations of a Muller C-element.

at both inputs of a Muller C-element for its output to change state and to create an output event. As long as this does not happen, the output remains unchanged and no output event is generated. The implementation of a C-element is centered around a latch, which should not be a surprise, given the similarities in their truth tables. Figure 10-41 presents a static and a dynamic circuit realization of the function.

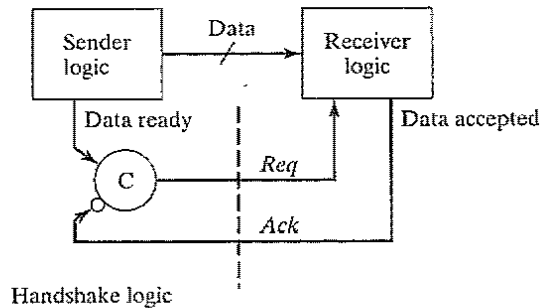


**Figure 10-42** A Muller C-element implements a two-phase handshake protocol. The circle at the lower input of the Muller C-element stands for inversion.

Figure 10-42 shows how to use this component to enforce the two-phase handshaking protocol for the example of the sender–receiver. Assume that *Req*, *Ack*, and *Data Ready* are initially 0. When the sender wants to transmit the next word, the *Data Ready* signal is set to 1, which triggers the C-element, because both its inputs are at 1. *Req* goes high—this is commonly denoted as  $Req\uparrow$ . The sender now resides in the wait mode, and control is passed to the receiver. The C-element is blocked, and no new data are sent to the data bus (*Req* stays high) as long as the transmitted data are not processed by the receiver. Once this happens, the *Data accepted* signal is raised. This can be the result of many different actions, possibly involving other C-elements communicating with subsequent blocks. An  $Ack\uparrow$  ensues, which unblocks the C-element and passes the control back to the sender. A  $Data\ ready\downarrow$  event, which might already have happened before  $Ack\uparrow$ , produces a  $Req\downarrow$ , and the cycle is repeated.

### Problem 10.1 Two-Phase Self-Timed FIFO

Figure 10-43 shows a two-phase, self-timed implementation of a FIFO (first-in first-out) buffer with three registers. Assuming that the registers accept a data word on both positive- and negative-going transitions of the *En* signals, and that the *Done* signal is simply a delayed version of *En*, examine the operation of the FIFO by plotting the timing behavior of all signals of interest. How can you observe that the FIFO is completely empty? Full? (Hint: Determine the necessary conditions on the *Ack* and *Req* signals.)

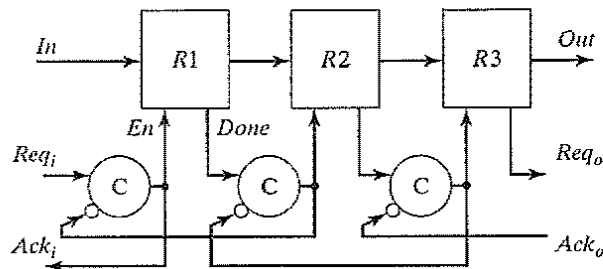


**Figure 10-43** Three-stage, self-timed FIFO, using a two-phase signaling protocol.

The two-phase protocol has the advantage of being simple and fast. However, this protocol requires the detection of transitions that may occur in either direction. Most logic devices in the MOS technology tend to be sensitive to levels or to transitions in one particular direction. Event-triggered logic, as required in the two-phase protocol, requires extra logic as well as state information in the registers and the computational elements. Since the transition direction is important, initializing all the Muller C-elements in the appropriate state is essential. If this is not done, the circuit might become deadlocked, which means that all elements are permanently blocked and nothing will ever happen. A detailed study on how to implement event-triggered logic can be found in [Sutherland89], the defining text on micropipelines.

The only alternative is to adopt a different signaling approach, such as *four-phase signaling*, or *return-to-zero (RTZ)*. This class of signaling requires that all controlling signals be brought back to their initial values before the next cycle can be initiated. Once again, this is illustrated with the example of the sender-receiver. The four-phase protocol for this example is shown in Figure 10-44.

The protocol presented is initially the same as the two-phase one. Both the *Req* and the *Ack* are initially in the zero state, however. Once a new data word is put on the bus ①, the *Req* is raised (*Req*↑ or ②), and control is passed to the receiver. When ready, the receiver accepts the data and raises *Ack* (*Ack*↑ or ③). So far, nothing new. The protocol proceeds by bringing both *Req* (*Req*↓ or ④) and *Ack* (*Ack*↓ or ⑤) back to their initial state in sequence. Only when that state is reached is the sender allowed to put new data on the bus ①. This protocol is called *four-phase signalling* because four distinct time zones can be recognized per cycle: two for the sender; two for the receiver. The first two phases are identical to the two-phase protocol, while the last two are devoted to resetting of the state. An implementation of the protocol, based on Muller C-elements, is shown in Figure 10-45. It is interesting to note that the four-phase protocol requires two C-elements in series (since four states must be represented). The *Data ready* and *Data accepted* signals must be pulses instead of single transitions.

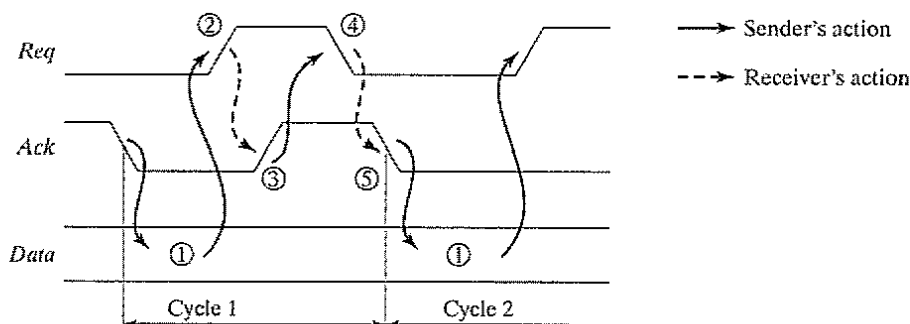


Figure 10-44 Four-phase handshaking protocol.

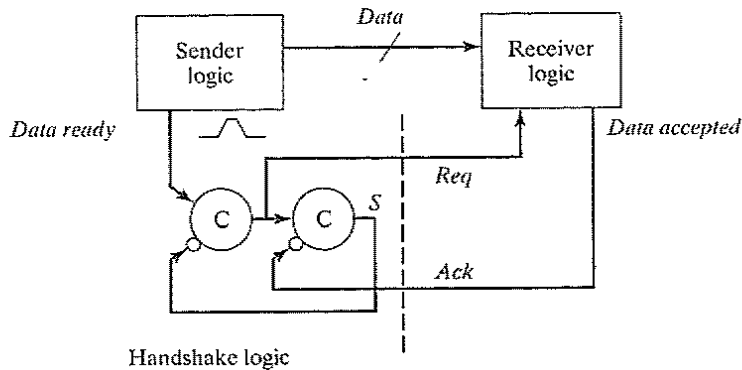


**Figure 10-45** Implementation of four-phase handshake protocol using Muller C-elements.

### Problem 10.2 Four-Phase Protocol

Derive the timing diagram for the signals shown in Figure 10-45. Assume that the *Data Ready* signal is a pulse and that the *Data Accepted* signal is a delayed version of *Req*.

The four-phase protocol has the disadvantage of being more complex and slower, since two events on *Req* and *Ack* are needed per transmission. On the other hand, it has the advantage of being robust. The logic in the sender and receiver modules does not have to deal with transitions, which can go either way; it has to consider only rising (or falling) transition events or signal levels. This is readily accomplished with traditional logic circuits. For this reason, four-phase handshakes are the preferred implementation approach for most of the current self-timed circuits. The two-phase protocol is mainly selected when the sender and receiver are far apart and the delays on the control wires (*Ack* and *Req*) are substantial.

### Example 10.6 The Pipelined Datapath—Revisited

We have introduced both the signaling conventions and the concepts of the completion-signal generation. Now it is time to bring them all together. We do this with the example of the pipelined datapath, which was presented earlier. A view of the self-timed datapath, including the timing control, is offered in Figure 10-46. The logic functions *F1* and *F2* are implemented using dual-rail, differential logic.

To understand the operation of this circuit, assume that all *Req* and *Ack* signals, including the internal ones, are set to 0, which means there is no activity in the data path. All *Start* signals are low so that all logic circuits are in precharge condition. An input request ( $Req_i \uparrow$ ) triggers the first C-element. The enable signal *En* of *R1* is raised, effectively latching the input data into the register, assuming a positive edge-triggered or a level-sensitive implementation.  $Ack_i \uparrow$  acknowledges the acceptance of the data. The second C-element is triggered as well, since  $Ack_{int}$  is low. This raises the *Start* signal and starts the evaluation of *F1*. At its completion, the output data are placed on the bus, and a

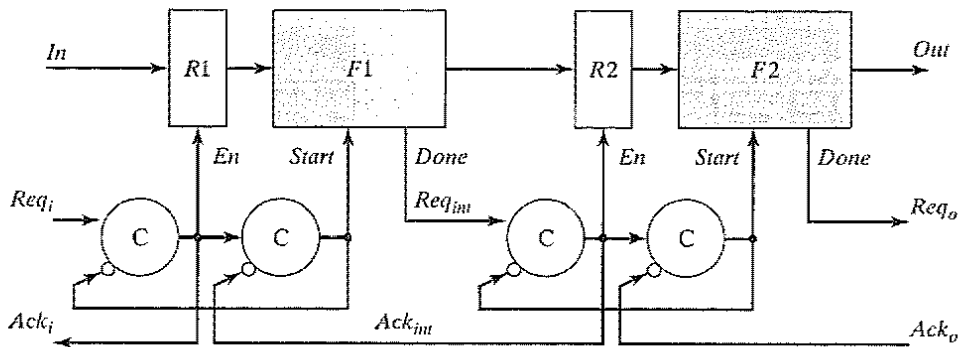


Figure 10-46 Self-timed pipelined datapath—complete composition.

request is initiated to the second stage ( $Req_{in} \uparrow$ ), which acknowledges its acceptance by raising  $Ack_{in}$ .

At this point, stage 1 is still blocked for further computations. However, the input buffer can respond to the  $Ack_i \uparrow$  event by resetting  $Req_i$  to its zero state ( $Req_i \downarrow$ ). In turn, this lowers  $En$  and  $Ack_i$ . Upon receipt of  $Ack_{in} \uparrow$ ,  $Start$  goes low, the precharge phase starts, and  $F1$  is ready for new data. Note that this sequence corresponds to the four-phase handshake mechanism described earlier. The dependencies among the events are presented in a more pictorial fashion in the *state transition diagram (STG)* shown in Figure 10-47. These STGs can become very complex. Computer tools are often used to derive STGs that ensure proper operation and optimize the performance.

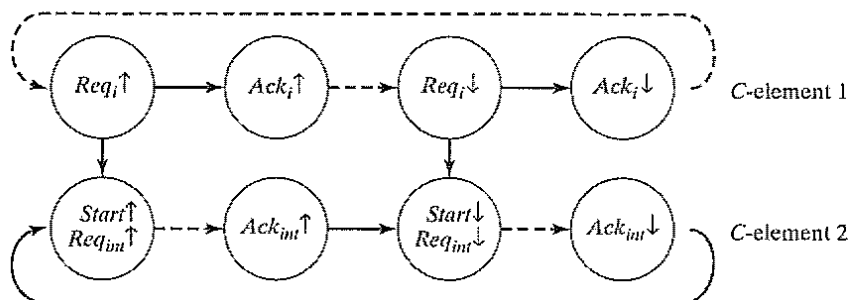


Figure 10-47 State transition diagram for pipeline stage 1. The nodes represent the signaling events, while the arrows express dependencies. Arrows in dashed lines express actions in either the preceding or the following stage.

#### 10.4.4 Practical Examples of Self-Timed Logic

Self-timed circuits can provide a significant performance advantage. Unfortunately, the overhead circuitry precludes a widespread application in general-purpose digital computing. Nevertheless, the key concepts of self-timed circuits are being exploited in a number of practical applications. We present a few examples that illustrate the use of self-timed concepts for either power savings or performance enhancement.

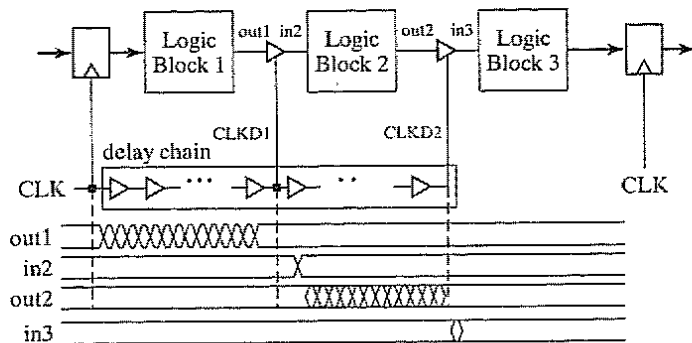


**Figure 10-48** Application of self-timing for glitch reduction.

### Glitch Reduction Using Self-Timing

A major source of unnecessary switched capacitance in large datapaths, such as bit-sliced adders and multipliers, is due to spurious transitions caused by *glitch propagation*. Imbalances in a logic network cause inputs of a logic gate or block to arrive at different times, resulting in glitching transitions. In large combinational blocks such as multipliers, transitions happen in waves as the primary input changes ripple through the logic. Enabling a logic block only when all the inputs have settled helps reduce or eliminate the glitching transitions. One approach for doing so is the use of a self-timed gating approach, which partitions each computational logic block into smaller blocks and distinct phases. Tristate buffers are inserted between each of these phases in order to prevent glitches from propagating through the datapath, as shown in Figure 10-48. Assuming an arbitrary logic network, it is fair to assume that the outputs of logic block 1 will not be synchronized. When the tristate buffers at the output of logic block 1 are enabled, the computation of logic block 2 is allowed to proceed. To reduce glitching transitions, the tristate buffer should be enabled only when it is ensured that the outputs of logic block 1 are stable and valid. The control of the tristate buffer can be performed through the use of a self-timed enable signal, generated by passing the system clock through a delay chain that models the critical path of the processor. The delay chain is then tapped at the points corresponding to the locations of the buffers, and the resulting signals are distributed throughout the chip. This technique succeeds in substantially reducing the switching capacitance of combinational logic blocks such as multipliers, even when taking into account the overhead of the delay chain, gating signal distribution, and buffers [Goodman98].

### Post-Charge Logic

An interesting form of self-timed logic is self-resetting CMOS. This structure uses a control structure different from the conventional self-timed pipeline. Instead of waiting for all the logic stages to complete their operation before transitioning to the reset stage, the idea is to precharge a logic block as soon as it completes its operation. Since the precharge happens after the operation instead of before evaluation, it is often termed *postcharge logic*. A block diagram of post-charge logic is shown in Figure 10-49 [Williams00]. As can be seen from this block diagram, the

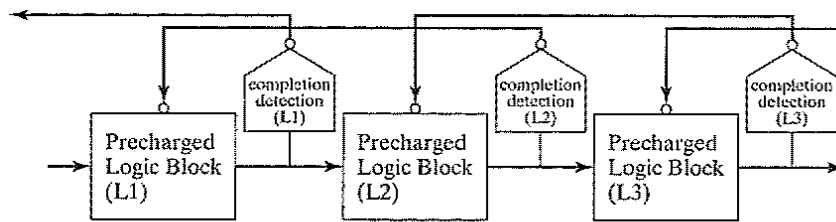


Figure 10-49 Self-resetting logic.

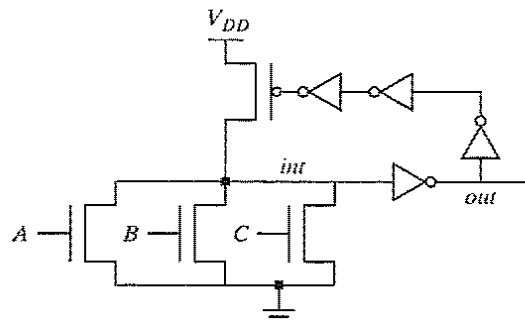


Figure 10-50 Self-resetting three-input OR.

precharging of L1 happens when its successor stage has completed and does not need its input anymore. It is possible to precharge a block based on the completion of its own output, but care must be taken to ensure that the stage that follows has properly evaluated and the input has become obsolete.

It should be noted that, unlike with other logic styles, the signals are represented as pulses and are valid only for a limited duration. The pulse width must be shorter than the reset delay or else there is a contention between the precharge device and the evaluate switches. While this logic style offers potential speed advantages, special care must be taken to ensure correct timing. Also, circuitry that converts level signals to pulses, and vice versa is required. An example of self-resetting logic is shown in Figure 10-50 [Bernstein98]. Assume that all inputs are low and that *int* is initially precharged. If *A* goes high, *int* will fall, causing *out* to go high. This causes the gate to precharge. When the PMOS precharge device is active, the inputs must be in a reset state in order to avoid contention.

### Clock-Delayed Domino

One interesting application of self-timed circuits that uses the delay-matching concept is *clock-delayed (CD) domino* [Yee96][Bernstein00]. This is a style of dynamic logic in which there is no global clock signal. Instead, the clock for one stage is derived from that for the previous stage. A simple example of a CD domino stage is shown in Figure 10-51. The two inverter delays plus the transmission gate along the clock path emulate the worst-case delay through the dynamic logic gate. The transmission gate is always turned on, and the delay of the clock path is adjusted

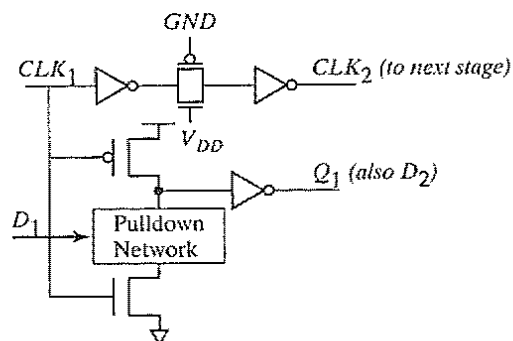


**Figure 10-51** Clock-delayed domino logic: a self-clocked logic style.

through the sizing of the devices. Clock-delayed domino was used in IBM's 1-GHz microprocessor, and it is used widely in high-speed domino logic. Clock-delayed domino can provide both inverting and noninverting functionality. This alleviates a major limitation of conventional domino techniques, which are capable only of noninverting logic. The inverter after the pull-down network is not essential because the clock arrives at the next stage only after the current stage has evaluated. The clock-evaluation edge thus arrives only after the inputs are stable. On the other hand, adding the inverter allows for the elimination of the foot switch (see Chapter 6).

## 10.5 Synchronizers and Arbiters\*

### 10.5.1 Synchronizers—Concept and Implementation

Even though a complete system may be designed in a synchronous fashion, it must still communicate with the outside world, which is generally asynchronous. An asynchronous input can change value at any time related to the clock edges of the synchronous system, as is illustrated in Figure 10-52.

Consider a typical personal computer. All operations within the system are strictly orchestrated by a central clock that provides a time reference. This reference determines what happens within the computer system at any point in time. This synchronous computer has to communicate with a human through the mouse or the keyboard, even though the human does not have knowledge of this time reference and may decide to press a keyboard key at any point in time. The way a synchronous system deals with such an asynchronous signal is to sample or poll it at regular intervals and to check its value. If the sampling rate is high enough, no transitions will be missed—this is known as the *Nyquist criterion* in the communication community. However, it might happen that the signal is polled in the middle of a transition. The resulting value is neither low nor high, but undefined. At that point, it is not clear whether the key was pressed or not. Feeding the undefined signal into the computer could be the source of all kinds of trouble, especially when it is routed to different functions or gates that might interpret it differently. For instance, one function might decide that the key is pushed and start a certain action, while another function might lean the other way and issue a competing command. This results in a conflict and thus a

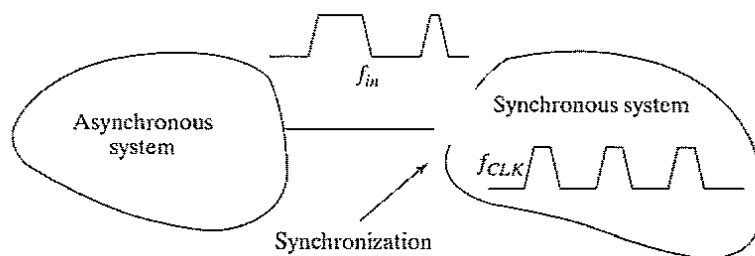


Figure 10-52 Asynchronous-synchronous interface.

potential crash. Therefore, the undefined state must be resolved in one way or another before it is interpreted further. It does not really matter what decision is made, as long as a unique result is available. For instance, it is either decided that the key is not yet pressed, a situation that will be corrected in the next poll of the keyboard, or it is concluded that the key is already pressed.

Thus, an asynchronous signal must be resolved to be either in the high or low state before it is fed into the synchronous environment. A circuit that implements such a decision-making function is called a *synchronizer*. Now comes the bad news: **Building a perfect synchronizer that always delivers a legal answer is impossible** [Chaney73] [Glasser85]! A synchronizer needs some time to come to a decision, and in certain cases this time might be arbitrarily long. An asynchronous-synchronous interface is thus always prone to errors, known as *synchronization failures*. The designer's task is to ensure that the probability of such a failure is small enough that it is not likely to disturb the normal system behavior. Typically, this probability can be reduced in an exponential fashion by waiting longer before making a decision. This is not too troublesome in the keyboard example, but in general, waiting affects system performance and should be avoided as much as possible.

To illustrate why waiting helps reduce the failure rate of a synchronizer, consider a synchronizer as shown in Figure 10-53. This circuit is a latch that is transparent during the low phase of the clock and samples the input on the rising edge of the clock  $CLK$ . However, since the sampled signal is not synchronized to the clock signal, there is a finite probability that the setup time or hold time of the latch is violated. (The probability is a strong function of the transition frequencies of the input and the clock.) As a result, once the clock goes high, there is a chance that the output of the latch resides somewhere in the undefined transition zone. The sampled signal eventually evolves into a legal 0 or 1, even in the latter case, as the latch has only two stable states.

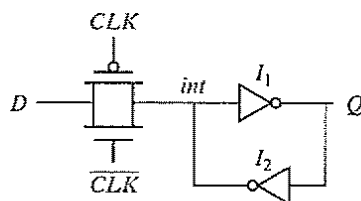


Figure 10-53 A simple synchronizer.

**Example 10.7 Synchronizer Trajectories**

Figure 10-54 shows the simulated trajectories of the output nodes of a cross-coupled static CMOS inverter pair for an initial state close to the metastable point. The inverters are composed of minimum-size devices.

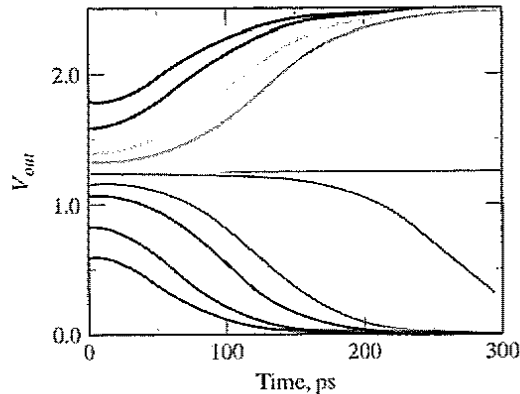


**Figure 10-54** Simulated trajectory for a simple synchronizer.

If the input is sampled such that the cross-coupled inverter pair starts at the metastable point, the voltage will remain at the metastable state forever in the absence of noise. If the data are sampled with a small offset (positive or negative), the differential voltage will evolve in an exponential way, with a time constant that is dependent on the strength of the transistors as well as the parasitic capacitances. The time it takes to reach the acceptable signal zones depends upon the initial distance of the sampled signal from the metastable point.

In order to determine the required waiting period, let us build a mathematical model of the behavior of the bistable element and use the results to determine the probability of synchronization failure as a function of the waiting period [Veendrick80].

The transient behavior around the metastable point of the cross-coupled inverter pair is accurately modeled by a system with a single dominant pole. Under this assumption, the transient response of the bistable element after the sampling clock has been turned off can be modeled as

$$v(t) = V_{MS} + (v(0) - V_{MS})e^{-t/\tau} \quad (10.8)$$

where  $V_{MS}$  is the metastable voltage of the latch,  $v(0)$  is the initial voltage after the sampling clock is turned off, and  $\tau$  is the time constant of the system. Simulations indicate that this first-order model is quite precise.

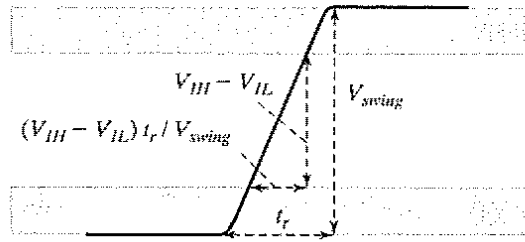


Figure 10-55 Linear approximation of signal slope.

The model can now be used to compute the range of values for  $v(0)$  that causes an error or a voltage in the undefined range, after a waiting period  $T$ . A signal is called **undefined** if its value is situated between  $V_{IH}$  and  $V_{IL}$ :

$$V_{MS} - (V_{MS} - V_{IL})e^{-T/\tau} \leq v(0) \leq V_{MS} + (V_{IH} - V_{MS})e^{-T/\tau} \quad (10.9)$$

Equation (10.9) conveys an important message: **The range of input voltages that cause a synchronization error decreases exponentially with the waiting period  $T$ .** Increasing the waiting period from  $2\tau$  to  $4\tau$  decreases the interval and the chances of an error by a factor of 7.4.

Some information about the asynchronous signal is required in order to compute the probability of an error. Assume that  $V_{in}$  is a periodical waveform with an average period  $T_{signal}$  between transitions and with identical rise and fall times  $t_r$ . Assume also that the slopes of the waveform in the undefined region can be approximated by a linear function, as shown in Figure 10-55. Using this model, we can estimate the probability  $P_{init}$  that  $v(0)$ , the value of  $V_{in}$  at the sampling time, resides in the undefined region:

$$P_{init} = \frac{\left(\frac{V_{IH} - V_{IL}}{V_{swing}}\right)t_r}{T_{signal}} \quad (10.10)$$

The chances for a synchronization error to occur depend upon the frequency of the synchronizing clock,  $\phi$ . The greater the number of sampling events, the higher the chance of running into an error. This means that the average number of synchronization errors per second,  $N_{sync}(0)$ , equals Eq. (10.11) if no synchronizer is used. Therefore, we can write

$$N_{sync}(0) = \frac{P_{init}}{T_\phi} \quad (10.11)$$

where  $T_\phi$  is the sampling period.

From Eq. (10.9), we learned that waiting a period  $T$  before observing the output reduces exponentially the probability that the signal is still undefined:

$$N_{sync}(T) = \frac{P_{init}e^{-T/\tau}}{T_\phi} = \frac{(V_{IH} - V_{IL})e^{-T/\tau}}{V_{swing}} \frac{t_r}{T_{signal}T_\phi} \quad (10.12)$$

The robustness of an asynchronous–synchronous interface is thus determined by the following parameters: signal switching rate and rise time, sampling frequency, and waiting time  $T$ .

### Example 10.8 Synchronizers and Mean Time to Failure

Consider the following design example:  $T_\phi = 5$  ns, which corresponds to a 200-MHz clock;  $T = T_\phi = 5$  ns;  $T_{\text{signal}} = 50$  ns;  $t_r = 0.5$  ns; and  $\tau = 150$  ps. From the VTC of a typical CMOS inverter, it can be derived that  $V_{IH} - V_{IL}$  approximately equals 0.5 V for a voltage swing of 2.5 V. Evaluation of Eq. (10.12) yields an error probability of  $1.38 \times 10^{-9}$  errors/s. The inverse of  $N_{\text{sync}}$  is called the *mean time to failure*, or the MTF, and equals  $7 \times 10^8$  s, or 23 years. If no synchronizer were used, the MTF would have been only 2.5  $\mu$ s!

### Design Consideration

When designing a synchronous–asynchronous interface, we must keep in mind the following observations:

- The acceptable failure rate of a system depends upon many economic and social factors and is a strong function of the system's application area.
- The exponential relation in Eq. (10.12) makes the failure rate extremely sensitive to the value of  $\tau$ . Defining a precise value of  $\tau$  is not easy in the first place, because  $\tau$  varies from chip to chip and is a function of temperature as well. The probability of an error occurring can thus fluctuate over large ranges even for the same design. A worst case design scenario is definitely advocated here. If the worst case failure rate exceeds a certain criterion, it can be reduced by increasing the value of  $T$ . A problem occurs when  $T$  exceeds the sampling period  $T_\phi$ . This can be avoided by cascading (or pipelining) a number of synchronizers, as shown in Figure 10-56. Each of those synchronizers has a waiting period equal to  $T_\phi$ . Notice that this arrangement requires the  $\phi$  pulse to be short enough to avoid race conditions. The global waiting period equals the sum of the  $T$ s of all the individual synchronizers. The increase in MTF comes at the expense of an increased latency.

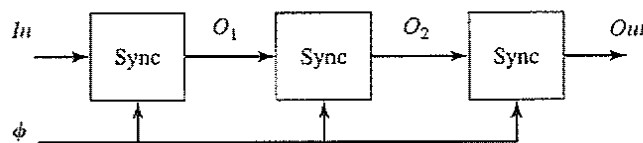


Figure 10-56 Cascading (edge-triggered) synchronizers reduces the mean time to failure.

- Synchronization errors are very hard to trace, due to their random nature. Making the mean time to failure very large does not preclude errors. The **number of synchronizers in a system should therefore be severely restricted**. A maximum of one or two per system is advocated. ■

### 10.5.2 Arbiters

Finally, a sibling of the synchronizer called the *arbiter*, interlock element, or mutual-exclusion circuit, should be mentioned. An arbiter is an element that decides which of two events has occurred first. For example, such components allow multiple processors to access a single

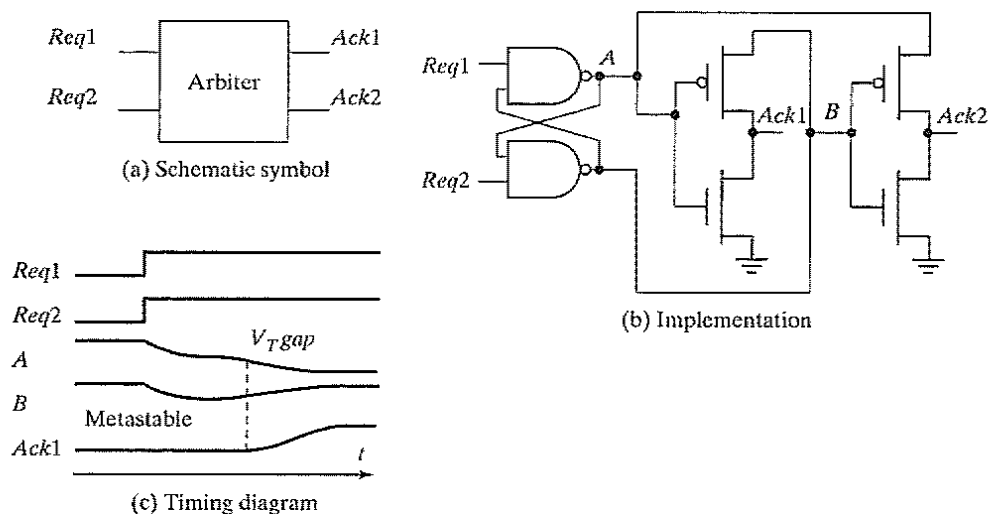


**Figure 10-57** Mutual-exclusion element (or arbiter).

resource, such as a large shared memory. A synchronizer is actually a special case of an arbiter, since it determines whether a signal transition happened before or after a clock event. A synchronizer is thus an arbiter with one of its inputs tied to the clock.

An example of a mutual-exclusion circuit is shown in Figure 10-57. It operates on two input-request signals that operate on a four-phase signaling protocol; that is, the *Req(uest)* signal has to go back to the reset state before a new *Req(uest)* can be issued. The output consists of two *Ack(nowledge)* signals that should be mutually exclusive. While *Requests* may occur concurrently, only one of the *Acknowledges* is allowed to go high. The operation is most easily visualized starting with both inputs low—neither device issuing a request—nodes *A* and *B* high, and both *Acknowledges* low. An event on one of the inputs (e.g.,  $Req1 \uparrow$ ) causes the flip-flop to switch, node *A* to go low, and  $Ack1 \uparrow$ . Concurrent events on both inputs force the flip-flop into the metastable state, and the signals *A* and *B* might be undefined for a while. The cross-coupled output structure keeps the output values low until one of the NAND outputs differs from the other by more than a threshold value  $V_T$ . This approach eliminates glitches at the output.

## 10.6 Clock Synthesis and Synchronization Using a Phase-Locked Loop\*

There are numerous digital applications that require the on-chip generation of a periodic signal. Synchronous circuits need a global periodic clock reference to orchestrate the events. Current microprocessors and high-performance digital circuits require clock frequencies in the gigahertz range. Crystal oscillators, by contrast, generate only accurate, low-jitter clocks over a frequency range from 10's of MHz to approximately 200 MHz. To generate the higher frequency required by digital circuits, a *phase-locked loop* (PLL) structure typically is used. A PLL takes an external low-frequency reference crystal frequency signal and multiplies its frequency by a rational number  $N$  (see the left side of Figure 10-58).

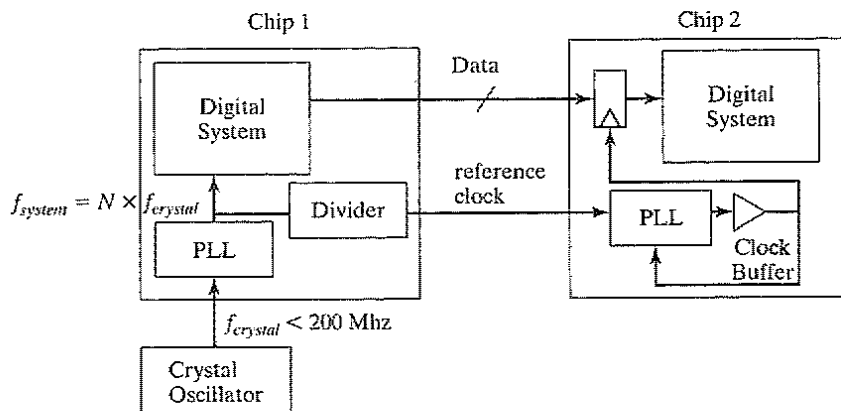


Figure 10-58 Applications of phase locked loops (PLL).

Another and equally important function of a PLL is to synchronize communications between chips. As shown in Figure 10-58, a reference clock is sent along with the parallel data being communicated. (In this example only the transmit path from chip 1 to chip 2 is shown.) Since chip-to-chip communication most often occurs at a lower rate than the on-chip clock rate, the reference clock is divided, but kept in phase with the system clock. In chip 2, the reference clock is used to synchronize all the input flip-flop, which can present a significant clock load in the case of wide data busses. Unfortunately, implementing clock buffers to deal with this problem introduces skew between the data and the sample clock. A PLL aligns (i.e., de-skews) the output of the clock buffer with respect to the data. In addition, the PLL can multiply the frequency of the incoming reference clock, allowing the core of the second chip to operate at a higher frequency than the input reference clock.

### 10.6.1 Basic Concept

A set of two or more periodic signals of the same frequency is well defined if we know one of them and its *phase* with respect to the other signals, as shown in Figure 10-59, where  $\phi_1$  and  $\phi_2$  represent the phases of the two signals. The *relative phase* is the difference between the two.

A PLL is a complex, nonlinear feedback circuit. Its basic operation is best understood with the aid of Figure 10-60 [Jeong87]. The *voltage-controlled oscillator* (VCO) takes an analog control input and generates a clock signal of the desired frequency. In general, there is a nonlinear relationship between the control voltage ( $v_{cont}$ ) and the output frequency. To synthesize a system clock of a particular frequency, it is necessary to set the control voltage to the appropriate value. This is a function of the rest of the blocks and the feedback loop in the PLL. The feedback loop is critical to the tracking process and environmental variations. The feedback also allows for frequency multiplication.

The reference clock is, in general, generated off chip from an accurate crystal reference. It is compared with a divided version of the system clock (i.e., the local clock), using a phase detector, which determines the phase difference between the signals and produces an

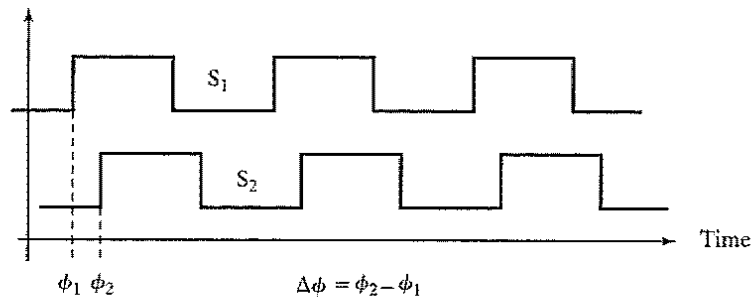


Figure 10-59 Relative and absolute phase of two periodic signals.

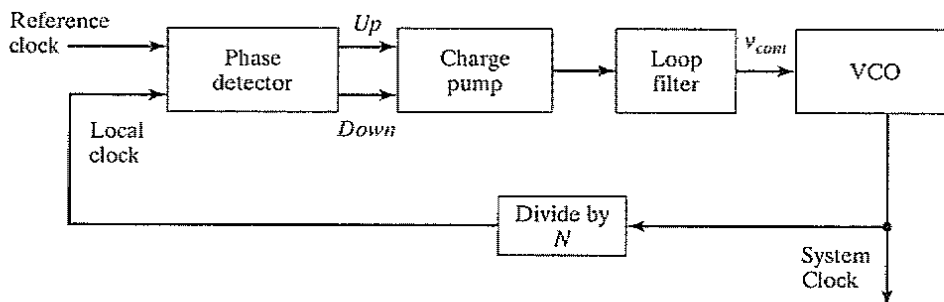


Figure 10-60 Functional composition of a phase-locked loop (PLL).

*Up* or *Down* signal when the local clock lags or leads the reference signal. It detects which of the two input signals arrives earlier and produces the appropriate output signal. The *Up* and *Down* signals are fed into a charge pump, which translates the digital encoded control information into an analog voltage [Gardner80]. An *Up* signal increases the value of the control voltage and speeds up the VCO, which causes the local signal to catch up with the reference clock. A *Down* signal, on the other hand, slows down the oscillator and eliminates the phase lead of the local clock.

Passing the output of the charge pump directly into the VCO creates a jittery clock signal. The edge of the local clock jumps back and forth instantaneously and oscillates around the targeted position. As discussed earlier, clock jitter is highly undesirable, since it reduces the time available for logic computation, and therefore should be kept within a given percentage of the clock period. This is partially accomplished by the introduction of the *loop filter*. This low-pass filter removes the high-frequency components from the VCO control voltage and smooths out its response, which results in a reduction of the jitter. Note that the PLL structure is a feedback structure. The addition of extra phase shifts, as introduced by a high-order filter, may result in instability. Important properties of a PLL are its *lock range*, the range of input frequencies over which the loop can maintain functionality; the *lock time*, the time it takes for the PLL to lock onto a given input signal; and the jitter. When in lock, the system clock is  $N$  times the reference clock frequency.

A PLL is an analog circuit and is inherently sensitive to noise and interference. This is especially true for the loop filter and VCO, for which induced noise has a direct effect on the resulting clock jitter. A major source of interference is the noise coupling through the supply rails and the substrate. This is a particular concern in digital environments, where noise is introduced by a variety of sources. Analog circuits with a high supply rejection, such as differential VCOs, are therefore desirable [Kim90]. In summary, integrating a highly sensitive component into a hostile digital environment is nontrivial and requires expert analog design. More detailed descriptions of various components of a PLL are given in the following subsections.

### 10.6.2 Building Blocks of a PLL

#### Voltage Controlled Oscillator (VCO)

A VCO generates a periodic signal with a frequency that is a linear function of the input control voltage  $v_{cont}$ . In other words, the VCO frequency can be expressed as

$$\omega = \omega_0 + K_{vco} \cdot v_{cont} \quad (10.13)$$

Since the phase is the time integral of the frequency, the output phase of the VCO block is given by

$$\phi_{out} = \omega_0 t + K_{vco} \cdot \int_{-\infty}^t v_{cont} dt \quad (10.14)$$

where  $K_{vco}$  is the gain of the VCO given in rad/s/V,  $\omega_0$  is a fixed frequency offset, and  $v_{cont}$  controls a frequency centered around  $\omega_0$ . The output signal has the form

$$x(t) = A \cdot \cos\left(\omega_0 t + K_{vco} \cdot \int_{-\infty}^t v_{cont} dt\right) \quad (10.15)$$

Various implementation strategies for VCOs were discussed in Chapter 7. Differential ring oscillators are the method of choice at present.

#### Phase Detectors

The phase detector determines the relative phase difference between two incoming signals and outputs a signal that is proportional to this phase difference. One of the inputs to the phase detector is a reference clock that typically is generated off chip, while the other clock input is a divided version of the VCO. Two basic types of phase detectors are commonly used—the *XOR gate* and the *phase frequency detector* (PFD).

**XOR Phase Detector** An XOR gate, the simplest of the two detectors, is useful as a phase detector because of the following observation: the relative phase difference between the inputs is reflected by the time over which the two inputs are different. Figure 10-61 shows the XOR-function of two waveforms. Feeding this output in a low-pass filter results into a dc-signal that is proportional to the phase difference, as shown in Figure 10-61(c).

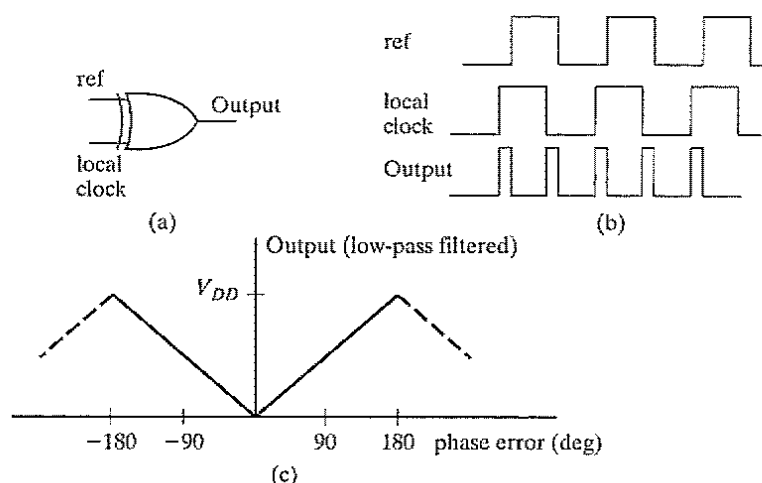


**Figure 10-61** (a) The XOR as a phase detector; (b) its output before low-pass filtering; and (c) transfer characteristics.

For this detector, a deviation in a positive or negative direction from the perfect in-phase condition (i.e., a phase error of zero) produces the same change in duty factor, resulting in the same average voltage. Thus the linear phase range is only 180 degrees. In steady state, a PLL locks into a quadrature phase relationship ( $\frac{1}{4}$  cycle offset) between the two inputs.

**Phase Frequency Detector** The phase frequency detector (PFD) is the most commonly used form of phase detector, and it solves several of the shortcomings of the detector discussed previously [Dally98]. As the name implies, the output of the PFD is dependent on both the phase and frequency difference between the applied signals. Accordingly, it cannot lock to an incorrect multiple of the frequency. The PFD takes two clock inputs and produces two outputs, *UP* and *DN*, as shown in Figure 10-62.

The PFD is a state machine with three states. Assume that both *UP* and *DN* outputs are initially low. When input *A* leads *B*, the *UP* output is asserted on the rising edge of input *A*. The *UP* signal remains in this state until a low-to-high transition occurs on input *B*. At that time, the *DN* output is asserted, causing both flip-flops to reset through the asynchronous reset signal. Notice that there is a small pulse on the *DN* output, whose duration is equal to the delay through the AND gate and register *Reset-to-Q* delay. The pulse width of the *UP* pulse is equal to the phase error between the two signals. The roles are reversed for the case in which input *A* lags *B*. A pulse proportional to the phase error is generated on the *DN* output. If the loop is in lock, short pulses will be generated on the *UP* and *DN* outputs.

The circuit also acts as a frequency detector, providing a measure of the frequency error (see Figure 10-63). For the case in which *A* is at a higher frequency than *B*, the PFD generates a lot more *UP* pulses—with the average pulse proportional to the frequency difference—while the number of *DN* pulses are close to zero, on average. Exactly the opposite is true for the case in

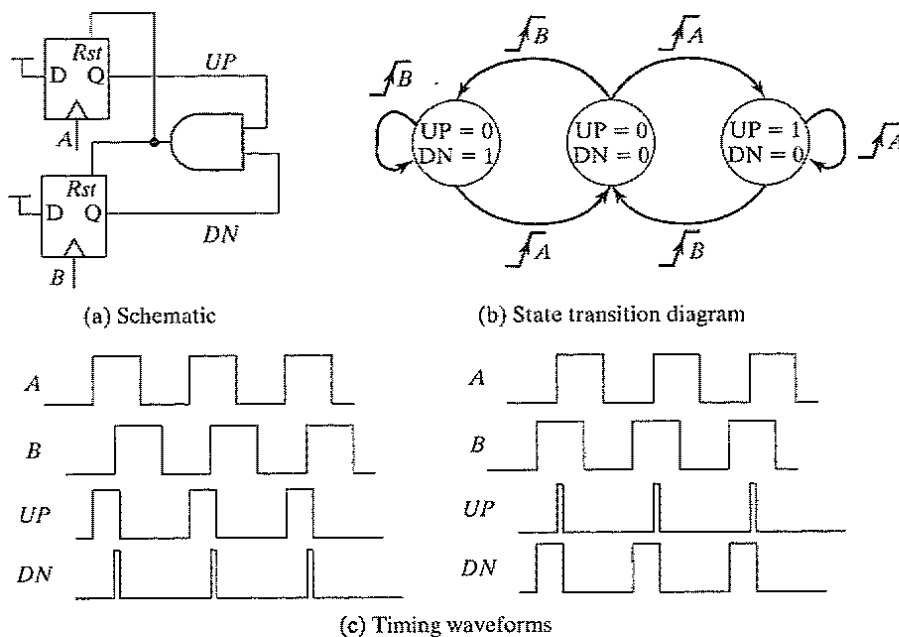


**Figure 10-62** Phase frequency detector.

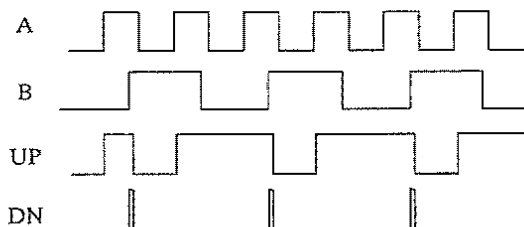


**Figure 10-63** Timing of the PFD measuring frequency error.

which  $B$  has a frequency larger than that of  $A$ —many more pulses are generated on the  $DN$  output than on the  $UP$  output.

The phase characteristics of the phase detector are shown in Figure 10-64. Notice that the linear range has been expanded to  $4\pi$ .

### Charge Pump

The  $UP$  and  $DN$  pulses must be converted into an analog voltage that controls the VCO. One possible implementation is shown in Figure 10-65. A pulse on the  $UP$  signal adds an amount of charge to the capacitor proportional to the width of the  $UP$  pulse, while a pulse on the  $DN$  signal removes charge proportional to the  $DN$  pulse width. If the width of the  $UP$  pulse is larger than that of the  $DN$  pulse, there is a net increase in the control voltage. This effectively increases the frequency of the VCO.

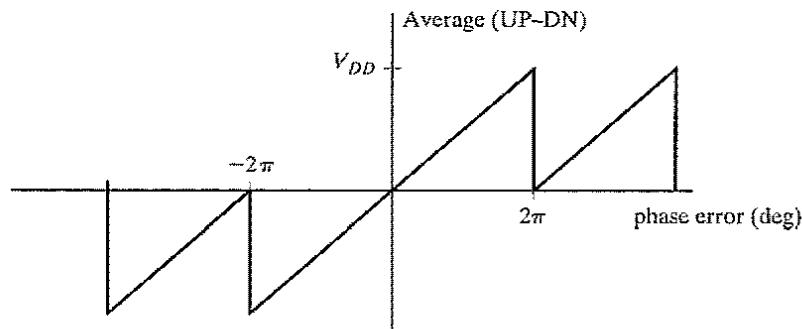


Figure 10-64 PFD phase characteristics.

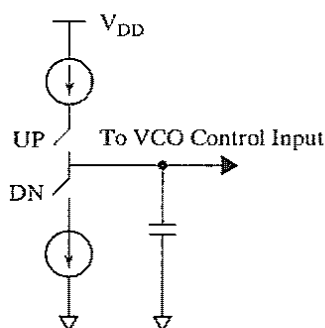


Figure 10-65 Charge pump.

### Example 10.9 Transient Behavior of Phase-Locked Loop

The settling time of a PLL is the time it takes to reach steady-state behavior. The length of the startup transient is strongly dependent on the bandwidth of the loop filter. Figure 10-66 shows a SPICE level simulation of a PLL implemented in a 0.25- $\mu\text{m}$  CMOS. An ideal VCO is used to speed up the simulation. In this example, a reference frequency of 100 MHz is chosen, and the PLL multiplies this frequency by 8 to 800 MHz. The Figure 10-66a illustrates the transient response and the settling process of the control voltage input of the VCO. Once the control voltage reaches its final value, the output frequency (the clock of the digital system) settles to steady state. The simulation on the right shows the reference frequency, the output of the divider, and the output frequency in steady state. Figure 10-66b shows the PLL in lock at  $f_{out} = 8 \times f_{ref}$ , while Figure 10-66c shows the waveforms during the locking process when the output is not in phase with the input, and the frequencies are not related yet by the divide ratio.

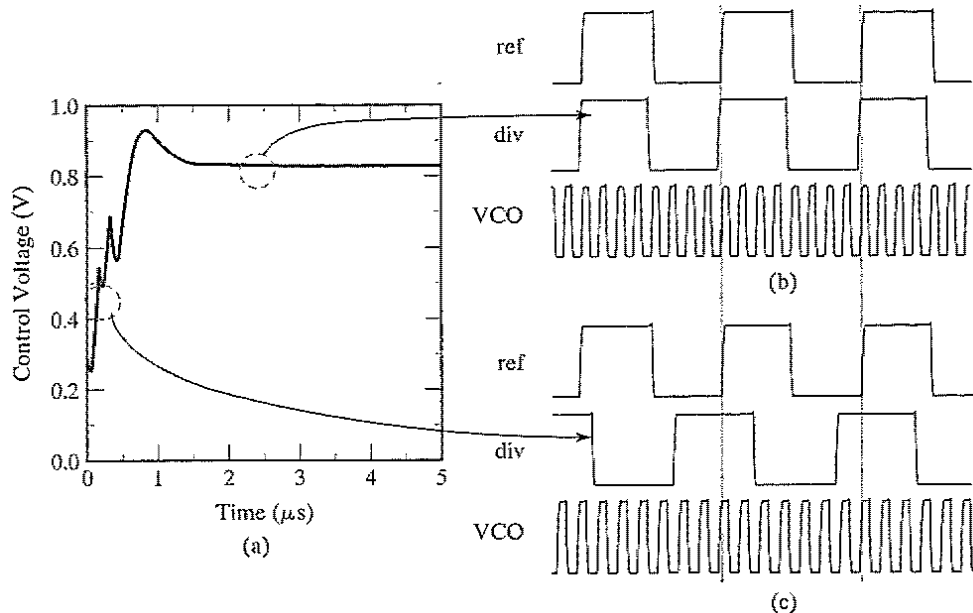


**Figure 10-66** SPICE simulation of a PLL. The control voltage of the VCO is shown along with the waveforms before lock and after lock.

### Summary

In a short span of time, phase-locked loops have become an essential component of any high-performance digital design. Their design requires considerable skill, integrating analog circuitry into a hostile digital environment. Yet experience has demonstrated that this combination is perfectly feasible, and it leads to new and better solutions.

## 10.7 Future Directions and Perspectives

This section highlights some of the trends in timing optimization for combining high performance and low power dissipation.

### 10.7.1 Distributed Clocking Using DLLs

A recent trend in high-performance clocking is the use of *delay locked loop* (DLL), a variation of the PLL structure. The schematic of a DLL is shown in Figure 10-67a [Maneatis00]. The key component of a DLL is a *voltage-controlled delay line* (VCDL). It consists of a cascade of adjustable delay elements (for instance, a current-starved inverter). The idea is to **delay the output clock such that it lines up perfectly with the reference**. Unlike the case of a VCO, there is no clock generator. The reference frequency is fed into the input of the VCDL. Similar to the case of a PLL structure, a phase detector compares the reference frequency with the output of the delay line ( $F_O$ ) and generates an *UP-DN* error signal. Note that only a phase and not a phase-frequency detection, is required. When in lock, there is no error between the two clocks. The

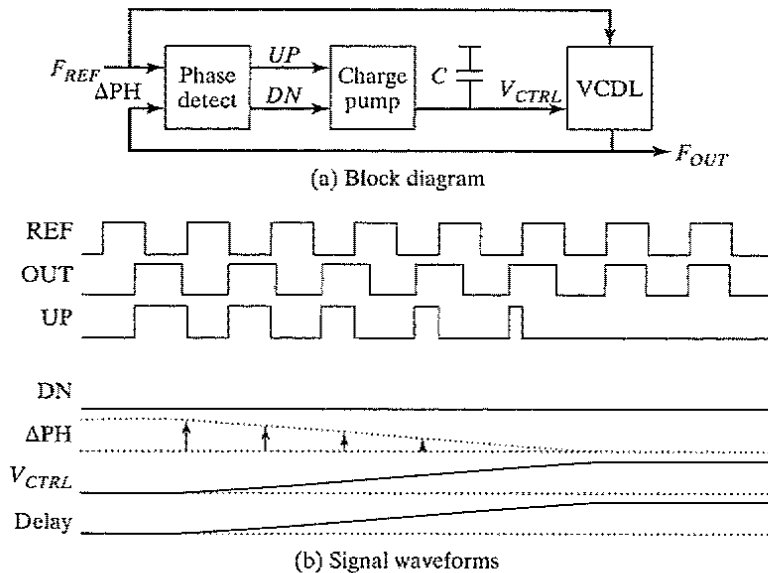


Figure 10-67 Delay-locked loop [Maneatis00].

function of the feedback is to adjust the delay through the VCDL such that the rising edge of the input reference clock ( $f_{REF}$ ) and the output clock ( $f_O$ ) are aligned.

A qualitative sketch of the signals in the DLL is shown in Figure 10-67b and c. Initially, the DLL is out of lock. Since the first edge of the output arrives before the reference edge, an  $UP$  pulse of width equal to the error between the two signals occurs. The role of the charge pump is to generate a charge packet proportional to the error, increasing the VCDL control voltage. This causes the edge of the output signal to be delayed in the next cycle. (This implementation of the VCDL assumes that a larger voltage results in larger delay.) After many cycles, the phase error is corrected, and the two signals are in lock. Note that a DLL does not alter the frequency of the input reference, but rather adjusts its phase.

Figure 10-68 shows the utilization of a DLL structure in a clock distribution network. The chip is partitioned into many small regions (or tiles). A global clock is distributed to each tile in a low-skew manner; this could be done through the package or by using low-skew, on-chip routing schemes. For the purpose of simplicity, the figure shows a two-tile chip, but this is easily extended to many regions. Inside each tile, the global clock is buffered before driving the digital load. In front of each buffer is a VCDL. The goal of the clock network is to deliver a signal to the digital circuit with close-to-zero skew and jitter. As we observed earlier, the static and dynamic variations of the buffers cause the phase error between the buffered clocks to be nonzero and time varying. The feedback inside each tile adjusts the control voltage of VCDL such that the buffered output is locked in phase to the global input clock. The feedback loop compensates for both static process variations and for slow dynamic variations (e.g., temperature). Such configurations have become common in high-performance digital microprocessors, as well as in media

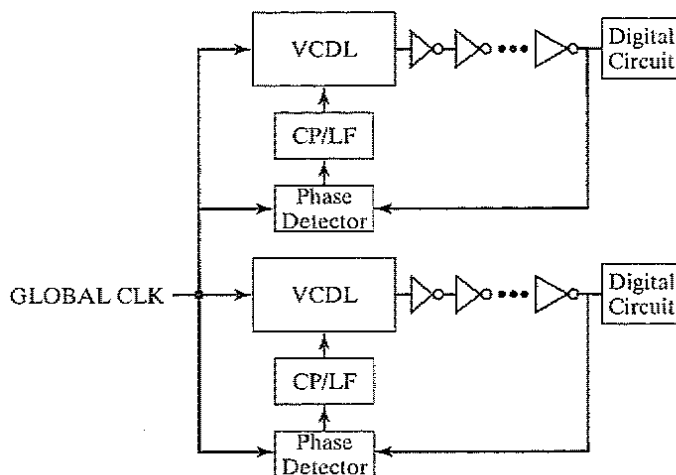


Figure 10-68 DLL approach to clock distribution.

processors. The preceding approach of clock distribution using multiple DLLs can be extended to the use of multiple distributed PLLs on a chip [Gutnik00]. Distributed PLLs offer potential performance advantages, but careful system analysis is required for stable operation.

### 10.7.2 Optical Clock Distribution

By now, the reader should have become aware of the fact that future high-performance multi-GHz systems face some fundamental synchronization problems. The performance of a digital design is fundamentally limited by process and environmental variations. Even with aggressive active clock management schemes, such as the use of DLLs and PLLs, the variations in power supply and clock load result in unacceptable clock uncertainty. Researchers are furiously searching for innovative solutions. An alternative approach that has received a lot of attention is the use of optics for systemwide synchronization. An excellent review of the rationale and trade-offs in optical interconnects versus electrical interconnects is given in [Miller00].

The potential advantages of optical technology for clock distribution are that the delay of optical signals is not sensitive to temperature and that the clock edges do not degrade over long distances (i.e., tens of meters). In fact, it is possible to deliver an optical clock signal with, at most, 10–100 ps of uncertainty over tens of meters. Optical clocks can be distributed on-chip via waveguides or through free space. Figure 10-69 shows the plot of an optical clock architecture, using waveguides. The off-chip optical source is brought to the chip, distributed through waveguides, and converted through receiver circuitry to a local electrical clock distribution network. The chip is divided into small sections (or tiles), and the global clock is distributed from the photon source through waveguides with splitters and bends to each of the sections. Notice that an *H*-tree is used in distributing the optical clock. At the end of the waveguide is a photodetector, which can be implemented using silicon or germanium. Upon reaching the detector in each section, the optical pulses that represent the global clock are converted into current pulses.

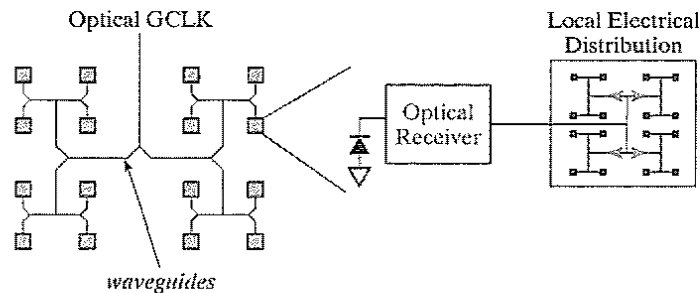


**Figure 10-69** Architecture for optical clock distribution (courtesy of Lionel Kimerling).

These have a very small magnitude (tens of  $\mu\text{A}$ ), and are fed into an amplifier that amplifies the signal into voltage signals appropriate for digital processing. The electrical clock is then distributed to the local load using conventional techniques [Kimerling00].

The optical approach has the advantage that the skew of the global clock to the photodetectors is virtually zero. There are some variations in the arrival time of the optical signal, but they are minimal. For instance, variations at the waveguide bends may cause the energy losses to differ from path to path. Optics has the additional advantage that many of the difficulties associated with electromagnetic wave propagation—such as cross talk and inductive coupling—are avoided.

On the other hand, the challenge lies in the design of the optical receiver. To amplify and transform the small current pulses into reasonable voltage signals requires multiple amplifier stages. These are susceptible to process and environmental variations, and thus also to skew. The problem becomes very similar to the conventional electrical approach. The advantage, however, is that the problem is confined to these receivers, which makes it more tractable.

Optical clocking may have an important future in high-performance systems. The challenges of dealing with process variations in the opto-electronic circuitry must be addressed first for this to become a reality.

### 10.7.3 Synchronous versus Asynchronous Design

The self-timed approach offers a potential solution to the growing clock-distribution problem. It translates the global clock signal into a number of local synchronization problems. Independence from physical timing constraints is achieved with the aid of completion signals. Handshaking logic is needed to ensure the logical ordering of the circuit events and to avoid race conditions. This requires adherence to a certain protocol, which normally consists of either two or four phases.

Despite all its advantages, self-timing has only been used in a number of isolated cases. Examples of self-timed circuits can be found in signal processing [Jacobs90], fast arithmetic units (such as self-timed dividers) [Williams87], simple microprocessors [Martin89] and memory (static RAM, FIFOs). In general, synchronous logic is both faster and simpler, since the overhead of completion-signal generation and handshaking logic is avoided. The design of a

foolproof network of handshaking units that is robust with respect to races, live lock, and dead lock is nontrivial and requires the availability of dedicated design-automation tools.

On the other hand, distributing a clock at high speed becomes exceedingly difficult. Skew management requires extensive modeling and analysis, as well as careful design. It will not be easy to extend this methodology into the next generation of designs. With the increasing timing uncertainty in synchronous circuits, self-timing is bound to become more attractive in the years to come [Sutherland02]. This observation is already reflected in the fact that the routing network for the latest generation of massively parallel supercomputers is completely implemented using self-timing [Seitz92]. For self-timing to become a mainstream design technique, however (if it ever will), further innovations in circuit and signaling techniques and design methodologies are needed.

Other alternative timing approaches might emerge as well. Possible candidates are fully asynchronous designs or islands of synchronous units connected by an asynchronous network. The latter method, called the *globally-asynchronous locally-synchronous approach*, is quite attractive. It avoids the pitfalls of self-timed design at the local circuit level, while eliminating the need for strict phase synchronization between blocks that are quite distant from each other on the die. In fact, the pure phase-synchronicity requirement between large modules in a system-on-a-chip is most often too large a constraint. Mesochronous or plesiochronous communication would work just as well. It is our conjecture that these styles of synchronization will become much more common in the coming decade.

## 10.8 Summary

This chapter has explored the timing of sequential digital circuits:

- An in-depth analysis of the synchronous digital circuits and clocking approaches was presented. Clock *skew* and *jitter* substantially impact the functionality and performance of a system. Important parameters are the clocking scheme used and the nature of the clock generation and distribution network.
- Alternative timing approaches, such as *self-timed design*, are becoming attractive for dealing with clock distribution problems. Self-timed design uses completion signals and handshaking logic to isolate physical timing constraints from event ordering.
- The connection of synchronous and asynchronous components introduces the risk of synchronization failure. The introduction of *synchronizers* helps to reduce that risk, but can never eliminate it.
- *Phase-locked loops* are becoming an important element of the digital designer's toolbox. They are used to generate high-speed clock signals on a chip. The analog nature of the PLL makes its design a real challenge.
- Important trend in clock distribution include the use of *delay-locked loops* to actively adjust delays on a chip.
- The key message of this chapter is that synchronization and timing are among the most intriguing challenges facing the digital designer of the next decade.

## 10.9 To Probe Further

While system timing is an important topic, a comprehensive reference work is not available in this area. One of the best discussions so far is the chapter by Chuck Seitz in [Mead80, Chapter 7]. Other in-depth overviews are given in [Bakoglu90], [Johnson93, Chapter 11], [Bernstein98, Chapter 7], and [Chandrakasan00, Chapters 12 and 13]. A collection of papers on clock distribution networks is presented in [Friedman95]. Numerous other publications are available on this topic in the leading journals, some of which are mentioned in the following list of references.

### References

- [Abnous93] A. Abnous and A. Behzad, "A High-Performance Self-Timed CMOS Adder," in *EE241 Final Class Project Reports*, by J. Rabaey, Univ. of California–Berkeley, May 1993.
- [Bailey98] D. Bailey and B. Benschneider, "Clocking Design and Analysis for a 600-MHz Alpha Microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 33, pp. 1627–1633, November 1998.
- [Bailey00] D. Bailey, "Clock Distribution," in [Chandrakasan00].
- [Bakoglu90] H. Bakoglu, *Circuits, Interconnections and Packaging for VLSI*, Addison-Wesley, pp. 338–393, 1980.
- [Boning00] D. Boning, S. Nassif, "Models of Process Variations in Device and Interconnect," in [Chandrakasan00].
- [Bowhill95] W. Bowhill et al., "A 300 MHz Quad-Issue CMOS RISC Microprocessor," *Technical Digest of the 1995 International Solid State Circuits Conference (ISSCC)*, San Francisco, pp. 182–183, February 1995.
- [Bernstein98] K. Bernstein et al., *High Speed CMOS Design Styles*, Kluwer Academic Publishers, 1998.
- [Bernstein00] K. Bernstein, "Basic Logic Families," in [Chandrakasan00].
- [Chandrakasan00] A. Chandrakasan, W. Bowhill, and F. Fox, *Design of High-Performance Microprocessor Circuits*, IEEE Press, 2000.
- [Chaney73] T. Chaney and F. Rosenberger, "Anomalous Behavior of Synchronizer and Arbiter Circuits," *IEEE Trans. on Computers*, vol. C-22, pp. 421–422, April 1973.
- [Dally98] W. Dally and J. Poulton, *Digital Systems Engineering*, Cambridge University Press, 1998.
- [Dean94] M. Dean, D. Dill, and M. Horowitz, "Self-Timed Logic Using Current-Sensing Completion Detection," *Journal of VLSI Signal Processing*, pp. 7–16, February 1994.
- [Dopperpuhl92] D. Dopperpuhl et al., "A 200 MHz 64-b Dual Issue CMOS Microprocessor," *IEEE Journal on Solid State Circuits*, vol. 27, no. 11, pp. 1555–1567, November 1992.
- [Friedman95] E. Friedman, ed., *Clock Distribution Networks in VLSI Circuits and Systems*, IEEE Press, 1995.
- [Gardner80] F. Gardner, "Charge-Pump Phase-Locked Loops," *IEEE Trans. on Communications*, vol. COM-28, pp. 1849–58, November 1980.
- [Glasser85] L. Glasser and D. Dopperpuhl, *The Design and Analysis of VLSI Circuits*, Addison-Wesley, pp. 360–365, 1985.
- [Goodman98] J. Goodman, A. P. Dancy, A. P. Chandrakasan, "An Energy/Security Scalable Encryption Processor Using an Embedded Variable Voltage DC/DC Converter," *IEEE Journal of Solid State Circuits*, pp. 1799–1809, 1998.
- [Gutnik00] Gutnik, V., A. P. Chandrakasan, "Active GHz Clock Network Using Distributed PLLs," *IEEE Journal of Solid State Circuits*, pp. 1553–1560, November 2000.
- [Heller84] L. Heller et al., "Cascade Voltage Switch Logic: A Differential CMOS Logic Family," *IEEE International Solid State Conference Digest*, San Francisco, pp. 16–17, February 1984.
- [Herrick00] B. Herrick, "Design Challenges in Multi-GHz Microprocessors," *Proceedings ASPDAC 2000*, Yokohama, January 2000.
- [Jacobs90] G. Jacobs and R. Brodersen, "A Fully Asynchronous Digital Signal Processor," *IEEE Journal on Solid State Circuits*, vol. 25, no. 6, pp. 1526–1537, December 1990.
- [Jeong87] D. Jeong et al., "Design of PLL-Based Clock Generation Circuits," *IEEE Journal on Solid State Circuits*, vol. SC-22, no. 2, pp. 255–261, April 1987.

- [Johnson93] H. Johnson and M. Graham, *High-Speed Digital Design—A Handbook of Black Magic*, Prentice Hall, 1993.
- [Kim90] B. Kim, D. Helman, and P. Gray, "A 30 MHz Hybrid Analog/Digital Clock Recovery Circuit in 2  $\mu$ m CMOS," *IEEE Journal on Solid State Circuits*, vol. SC-25, no. 6, pp. 1385–1394, December 1990.
- [Kimerling00] L. Kimerling, "Photons to the Rescue: Microelectronics becomes Microphotonics," *Interface*, vol. 9, no. 2, 2000.
- [Martin89] A. Martin et al., "The First Asynchronous Microprocessor: Test Results," *Computer Architecture News*, vol. 17, no. 4, pp. 95–110, June 1989.
- [Maneatis00] J. Maneatis, "Design of High-Speed CMOS PLLs and DLLs," in [Chandrakasan00].
- [Mead80] C. Mead and L. Conway, *Introduction to VLSI Design*, Addison-Wesley, 1980.
- [Messerschmitt90] D. Messerschmitt, "Synchronization in Digital Systems," *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 8, pp. 1404–1419, October 1990.
- [Miller00] D. Miller, "Rationale and Challenges for Optical Interconnects to Electronic Chips," *Proceedings of the IEEE*, pp. 728–749, June 2000.
- [Park00] T. Park, T. Tugbawa, and D. Boning, "An Overview of Methods for Characterization of Pattern Dependencies in Copper CMP," *Chemical Mechanical Polish for ULSI Multilevel Interconnection Conference (CMP-MIC 2000)*, pp. 196–205, March 2000.
- [Restle98] P. Restle, K. Jenkins, A. Deutsch, P. Cook, "Measurement and Modeling of On-Chip Transmission Line Effects in a 400 MHz Microprocessor," *IEEE Journal of Solid State Circuits*, pp. 662–665, April 1998.
- [Restle01] P. Restle, "Technical Visualizations in VLSI Design," *Proceedings 38th Design Automation Conference*, Las Vegas, June 2001.
- [Seitz80] C. Seitz, "System Timing," in [Mead80], pp. 218–262, 1980.
- [Seitz92] C. Seitz, "Mosaic C: An Experimental Fine-Grain Multicomputer," in *Future Tendencies in Computer Science, Control and Applied Mathematics*, Proceedings International Conference on the 25th Anniversary of INRIA, Springer-Verlag, pp. 69–85, 1992.
- [Sutherland89] I. Sutherland, "Micropipelines," *Communications of the ACM*, pp. 720–738, June 1989.
- [Sutherland02] I. Sutherland and J. Ebergen, "Computers Without Clocks," *Scientific American*, vol. 287, no. 2, pp. 62–69, August 2002.
- [Veendrick80] H. Veendrick, "The Behavior of Flip Flops Used as Synchronizers and Prediction of Their Failure Rates," *IEEE Journal of Solid State Circuits*, vol. SC-15, no. 2, pp. 169–176, April 1980.
- [Williams87] T. Williams et al., "A Self-Timed Chip for Division," in *Proceedings of Advanced Research in VLSI 1987*, Stanford Conference, pp. 75–96, March 1987.
- [Williams00] T. Williams, "Self-Timed Pipelines," in [Chandrakasan00].
- [Yee96] G. Yee et al., "Clock-Delayed Domino for Adder and Combinational Logic Design," *Proceedings IEEE International Conference on Computer Design (ICCD)*, pp. 332–337, 1996.
- [Young97] I. Young, M. Mar, and B. Brushan, "A 0.35  $\mu$ m CMOS 3-880 MHz PLL N/2 Clock Multiplier and Distribution Network with Low Jitter for Microprocessors," *IEEE International Solid State Conference Digest*, pp. 330–331, 1997.

## Exercises and Design Problem

Visit the book's web site <http://bwrc.eecs.berkeley.edu/1cBook> for challenging exercises and design problems.



## Design Verification

*Simulation versus Verification*

*Electrical and Timing Verification*

*Formal Verification*

Up to this point, we have relied heavily on simulation as the preferred method for ensuring the correctness of a design and for extracting critical parameters such as speed and power dissipation. Simulation is, however, only one of the techniques the designer has in his toolbox to accomplish those objectives. In general, it is useful to differentiate between *simulation* and *verification*.

In the *simulation* approach, the value of a design parameter such as noise margin, propagation delay, or dissipated energy is determined by applying a set of excitation vectors to the circuit model of choice and extracting parameters from the obtained signal waveforms. While this approach is very flexible, it has the disadvantage that the results depend strongly upon the choice of the excitations. For example, a charge-redistribution condition in a dynamic logic gate is not detected by a simulation if the exact sequence of input patterns that causes the charge sharing is not applied. Similarly, the delay of an adder varies widely depending on the input signal. Identification of the worst-case delay requires a careful choice of the excitation vector so that the complete carry path is exercised. In other words, the designer must have a good understanding of the intrinsics of the circuit module and its operation. Failing to do so produces meaningless results.

*Verification*, on the other hand, attempts to extract the system parameters directly from the circuit description. For instance, the critical path of an adder can be recognized from an inspection of the circuit diagram or a model of it. This approach has the advantage that the result is independent of the choice of excitation vectors and is supposedly foolproof. On the other hand, it relies on a number of implicit assumptions regarding design techniques and methodologies. For the example of the adder, determination of the propagation delay requires an understanding of the logic operation of the composing circuitry—for example, dynamic or static logic—and a definition of the term *propagation delay*. As a second example, it is necessary to identify the register elements first before the maximum clock speed of a synchronous circuit can be determined. The resulting tools are therefore restricted in scope and handle only a limited class of circuit styles, such as single-phase synchronous design. In this insert, we analyze a number of the popular verification techniques.

### Electrical Verification

Given the transistor schematics of a digital design, it is possible to verify whether a number of basic rules are satisfied. Some examples of typical rules help illustrate this concept, and many others can be derived from previous chapters:

- The number of inversions between two C<sup>2</sup>MOS gates should be even.
- In a pseudo-NMOS gate, a well-defined ratio between the PMOS pull-up and NMOS pull-down devices is necessary to guarantee a sufficient noise-margin low  $NM_L$ .
- To ensure that rise and fall times of the signal waveforms stay within limits, minimum bounds can be set on the sizes of the driver transistors as a function of the fan-out.
- The maximum amount of charge sharing in a dynamic design should be such that the noise-margin high is not violated.

Pure common sense can help define a large set of rules to which a design should always adhere. Applying them requires an in-depth understanding of the circuit structure. An electrical verifier, therefore, starts with the identification of well-known substructures in the overall circuit schematic. Typical templates are simple logic gates, pass transistors, and registers. The verifier traverses the resulting network on a rule-per-rule basis. Since electrical rules tend to be specific to a particular design style, they should be able to be easily modified. For example, *rule-based expert systems* allow for an easy updating of the rule base [DeMan85]. The individual rules can be complex and even invoke a circuit simulator for a small subsection of the network to verify whether a given condition is met. In summary, electrical verification is a helpful tool, and it can dramatically reduce the risk of malfunction.

### Timing Verification

As circuits become more complex, it is increasingly difficult to define exactly which paths through the network are critical with respect to timing. One solution is to run extensive SPICE simulations that may take a long time to finish. Even then, this does not guarantee that the iden-

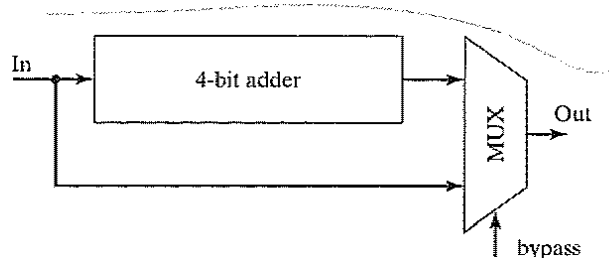


**Figure G-1** Example of false path in timing verification.

tified critical path is the worst case, since the delay path is a function of the applied signal patterns. A timing verifier traverses the electrical network and rank-orders the various paths, based on delay. This delay can be determined in a number of ways. One approach is to build an *RC*-model of the network and compute bounds on the delay of the resulting passive network. To obtain more accurate results, many timing verifiers first extract the details of the longest path(s), based on the *RC*-model, and perform circuit simulation on the reduced circuit in order to obtain a better estimation. Examples of early timing verifiers are the Crystal [Ousterhout83] and TV [Jouppi84] systems.

One problem that hampered many earlier systems is that they identified *false paths*—that is, critical paths that can never be exercised during normal circuit operation. For example, a false path exists in the carry-bypass adder discussed in Chapter 11 (also shown in simplified form in Figure G-1). From a simple analysis of only the circuit topology, one would surmise that the critical path of the circuit passes through the adder and multiplexer modules as illustrated by the arrow. A closer look at the circuit operation reveals that such a path is not feasible. All individual adder bits must be in the propagate mode for *In* to propagate through the complete adder. However, the *bypass* signal is asserted under these conditions, and the bottom path through the multiplexer is selected instead. The actual critical path is thus shorter than what would be predicted from the first-order analysis. Detecting false timing paths is not easy, since it requires an understanding of the logic functionality of the network. Newer timing verifiers are remarkably successful in accomplishing this and have become one of the more important design aids for the high-performance circuit designer [Devadas91].

### Example G.1 Example of Timing Verification

The output of a static timing verifier, the PathMill tool from Synopsys [PathMill], is shown in Figure G-2. The input to the verification process is a transistor netlist, but gate- and block-level models can be included as well. The analysis considers capacitive and resistive parasitics that are obtained from the transistor schematic or layout extraction.

The output of the timing analysis is an ordered list of critical timing paths. For the example of Figure G-2, the longest path extends from node *B* (falling edge) to node *Y* (falling edge) over nodes *S1*, *S2*, and *X3*. The predicted delay is 1.14 ns. Other paths close to

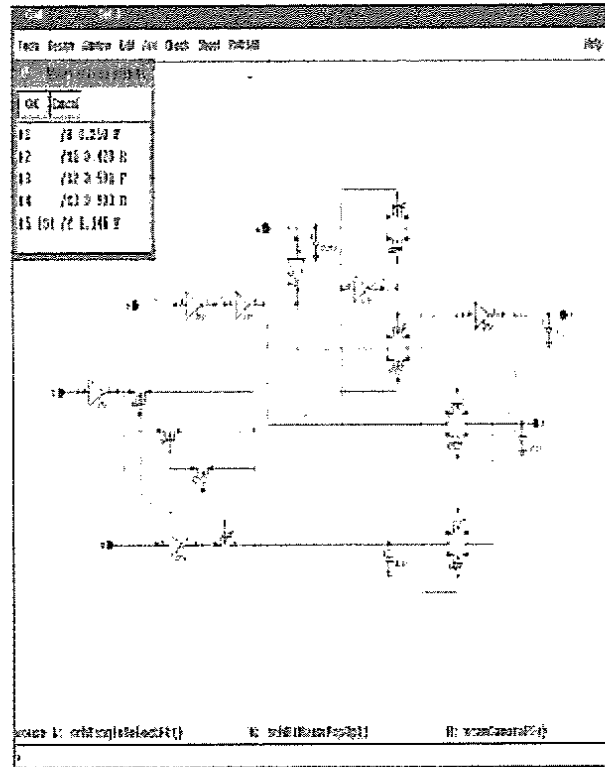


**Figure G-2** Example of static timing verifier response as generated by the PathMill tool (Courtesy of Synopsys, Inc.). The results are displayed using the Cadence DFII tool. The critical timing path runs from node *B* to node *Y* via nodes *S1*, *S2*, and *X3*.

the critical one are in order of decreasing delay:  $B(F) \rightarrow Y(R)$  (1.11 ns),  $B(R) \rightarrow Y(F)$  (1.04 ns),  $C(R) \rightarrow Z(R)$  (1.04 ns), and  $D(R) \rightarrow Y(R)$  (1.00 ns). Observe that the circuit contains multiple chained pass transistors that make the analysis complicated and increase the chances for false paths to occur. An example of the latter is the path  $A(F) \rightarrow X1 \rightarrow X3 \rightarrow Z(R)$  for  $B = 0$ . For a low value on  $B$ , node  $Z$  can only make a falling transition.

### Functional (or Formal) Verification

Each component (transistor, gate, or functional block) of a circuit can be described behaviorally as a function of its inputs and internal state. By combining these component descriptions, an overall circuit model can be generated that symbolically describes the behavior of the complete circuit. Formal verification compares this derived behavior with the designer's initial specification. Although not identical, the two descriptions need to be mathematically equivalent for the circuit to be correct.

Formal verification is the designer's ultimate dream of what design automation should be able to accomplish—*proof that the circuit will work as specified*. Unfortunately, no general and widely accepted verifier has yet been realized. The complexity of the problem is illustrated by the following argument: One way to prove that two circuit descriptions are identical is to compare the outputs while enumerating all possible input patterns and sequences thereof. This is an intractable approach with a computation time exponential in the number of inputs and states.

This does not mean that formal verification is a fantasy, however. Techniques have been proposed and successfully implemented for certain classes of circuits. For instance, assuming that a circuit is synchronous helps minimize the search space. Proving the equivalence between state machines has been one of the main research targets and has led to some remarkable advances (e.g., [Coudert90]). In general, formal verification techniques fall into two major classes:

- *Equivalence Proving*—Design often follows a refinement approach. An initial high-level description is refined into a more detailed one in a step-by-step fashion. Equivalence-proving tools establish that the obtained result is functionally equivalent to the original description. As an example, such a tool could establish that the transistor schematic of an N-input static CMOS NAND gate truly implements the intended NAND function.
- *Property Proving*—A designer often wants to know if her creation possesses some well-defined properties. A possible requirement could be, "My circuit should never experience a race condition." A property-proving tool analyzes whether this goal is met; if it is not met, the tool also analyzes the circumstances.

One of the prime necessities for formal verification to work is that the design is described in a manner that is unambiguous and that has a clearly defined meaning (or semantics). When these conditions are met, formal-verification techniques have yielded some remarkable results and are very effective. For instance, finite-state machines (FSMs) represent an area where formal verification has made some major advances. The secret to the success is a well-defined mathematical description model.

### Summary

Although not yet in the mainstream of the design-automation process, functional verification might become one of the important assets in the designer's toolbox. For this to happen, important progress must be made in the fields of design specification and interpretation.

### References

- [Coudert90] O. Coudert and J. Madre, *Proceedings ICCAD 1990*, pp. 126–129, November 1990.
- [DeMan85] H. De Man, I. Bolsens, E. Vandenmeersch, and J. Van Cleynenbreugel, "DIALOG: An Expert Debugging System for MOS VLSI Design," *IEEE Trans. on CAD*, vol. 4, no. 3, pp. 301–311, July 1985.
- [Devadas91] S. Devadas, K. Keutzer, and S. Malik, "Delay Computations in Combinational Logic Circuits: Theory and Algorithms," *Proc. ICCAD-91*, pp. 176–179, Santa Clara, 1991.

- [Jouppi84] N. Jouppi, *Timing Verification and Performance Improvement of MOS VLSI Designs*, Ph. D. diss., Stanford University, 1984.
- [Ousterhout83] J. Ousterhout, "Crystal: A Timing Analyzer for nMOS VLSI Circuits," *Proc. 3rd Caltech Conf. on VLSI* (Bryant ed.), Computer Science Press, pp. 57–69, March 1983.
- [PathMill] PathMill: Transistor-Level Static Timing Analysis, [http://www.synopsys.com/products/analysis/pathmill\\_ds.html](http://www.synopsys.com/products/analysis/pathmill_ds.html), Synopsys, Inc.

# Designing Arithmetic Building Blocks

*Designing adders, multipliers, and shifters  
for performance, area, or power*

*Logic and system optimizations for datapath modules*

*Power-delay trade-offs in datapaths*

- 11.1 Introduction
- 11.2 Datapaths in Digital Processor Architectures
- 11.3 The Adder
  - 11.3.1 The Binary Adder: Definitions
  - 11.3.2 The Full Adder: Circuit Design Considerations
  - 11.3.3 The Binary Adder: Logic Design Considerations
- 11.4 The Multiplier
  - 11.4.1 The Multiplier: Definitions
  - 11.4.2 Partial-Product Generation
  - 11.4.3 Partial-Product Accumulation
  - 11.4.4 Final Addition
- 11.5 The Shifter
  - 11.5.1 Barrel Shifter
  - 11.5.2 Logarithmic Shifter
- 11.6 Other Arithmetic Operators
- 11.7 Power and Speed Trade-Offs in Datapath Structures\*
  - 11.7.1 Design-Time Power-Reduction Techniques
  - 11.7.2 Run-Time Power Management
  - 11.7.3 Reducing the Power in Standby (or Sleep) Mode

## 11.8 Perspective: Design as a Trade-off

## 11.9 Summary

## 11.10 To Probe Further

### 11.1 Introduction

After the in-depth study of the design and optimization of the basic digital gates, it is time to test our acquired skills on a somewhat larger scale and put them in a more system-oriented perspective.

We will apply the techniques of the previous chapters to design a number of circuits often used in the datapaths of microprocessors and signal processors. More specifically, we discuss the design of a representative set of modules such as adders, multipliers, and shifters. The speed and power of these elements often dominates the overall system performance. Hence, a careful design optimization is required. It rapidly becomes obvious that the design task is not straightforward. For each module, multiple equivalent logic and circuit topologies exist, each of which has its own positives and negatives in terms of area, speed, or power.

Although far from complete, the analysis presented helps focus on the essential trade-offs that must be made in the course of the digital design process. You will see that optimization at only one design level—for instance, through transistor sizing only—leads to inferior designs. A global picture is therefore of crucial importance. Digital designers focus their attention on the gates, circuits, or transistors that have the largest impact on their goal function. The noncritical parts of the circuit can be developed routinely. We also may develop first-order performance models that foster understanding of the fundamental behavior of a module. The discussion also clarifies which computer aids can help to simplify and automate this phase of the design process.

Before analyzing the design of the arithmetic modules, a short discussion of the role of the datapath in the digital-processor picture is appropriate. This not only highlights the specific design requirements for the datapath, but also puts the rest of this book in perspective. Other processor modules—the input/output, controller, and memory modules—have different requirements and were discussed in Chapter 8. After an analysis of the area-power-delay trade-offs in the design of adders, multipliers, and shifters, we will use the same structures to illustrate some of the power-minimization approaches introduced in Chapter 6. The chapter concludes with a short perspective on datapath design and its trade-offs.

### 11.2 Datapaths in Digital Processor Architectures

We introduced the concept of a digital processor in Chapter 8. Its components consist of the datapath, memory, control, and input/output blocks. The datapath is the core of the processor—this is where all computations are performed. The other blocks in the processor are support units that either store the results produced by the datapath or help to determine what will happen in the next cycle. A typical datapath consists of an interconnection of basic combinational functions, such as arithmetic operators (addition, multiplication, comparison, and shift) or logic (AND, OR, and XOR). The design of the arithmetic operators is the topic of this chapter. The

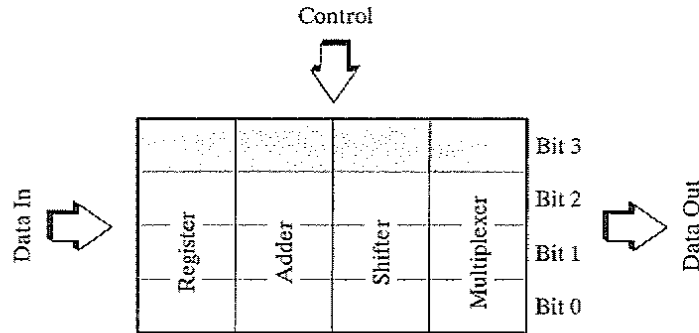


Figure 11-1 Bit-sliced datapath organization.

intended application sets constraints on the datapath design. In some cases, such as in personal computers, processing speed is everything. In most other applications, there is a maximum amount of power that is allowed to be dissipated, or there is maximum energy available for computation while maintaining the desired throughput.

Datapaths often are arranged in a *bit-sliced* organization, as shown in Figure 11-1. Instead of operating on single-bit digital signals, the data in a processor are arranged in a *word*-based fashion. Typical microprocessor datapaths are 32 or 64 bits wide, while the dedicated signal processing datapaths, such as those in DSL modems, magnetic disk drives, or compact-disc players are of arbitrary width, typically 5 to 24 bits. For instance, a 32-bit processor operates on data words that are 32 bits wide. This is reflected in the organization of the datapath. Since the same operation frequently has to be performed on each bit of the data word, the datapath consists of 32 bit slices, each operating on a single bit—hence the term *bit sliced*. Bit slices are either identical or resemble a similar structure for all bits. The datapath designer can concentrate on the design of a single slice that is repeated 32 times.

### 11.3 The Adder

Addition is the most commonly used arithmetic operation. It often is the speed-limiting element as well. Therefore, careful optimization of the adder is of the utmost importance. This optimization can proceed either at the logic or circuit level. Typical logic-level optimizations try to rearrange the Boolean equations so that a faster or smaller circuit is obtained. An example of such a logic optimization is the *carry lookahead adder* discussed later in the chapter. Circuit optimizations, on the other hand, manipulate transistor sizes and circuit topology to optimize the speed. Before considering both optimization processes, we provide a short summary of the basic definitions of an adder circuit (as defined in any book on logic design [e.g., Katz94]).

#### 11.3.1 The Binary Adder: Definitions

Table 11.1 shows the truth table of a binary full adder.  $A$  and  $B$  are the adder inputs,  $C_i$  is the carry input,  $S$  is the sum output, and  $C_o$  is the carry output. The Boolean expressions for  $S$  and  $C_o$  are given in Eq. (11.1).

Table 11-1 Truth table for full adder.

| $A$ | $B$ | $C_i$ | $S$ | $C_o$ | Carry Status       |
|-----|-----|-------|-----|-------|--------------------|
| 0   | 0   | 0     | 0   | 0     | delete             |
| 0   | 0   | 1     | 1   | 0     | delete             |
| 0   | 1   | 0     | 1   | 0     | propagate          |
| 0   | 1   | 1     | 0   | 1     | propagate          |
| 1   | 0   | 0     | 1   | 0     | propagate          |
| 1   | 0   | 1     | 0   | 1     | propagate          |
| 1   | 1   | 0     | 0   | 1     | generate/propagate |
| 1   | 1   | 1     | 1   | 1     | generate/propagate |

$$\begin{aligned}
 S &= A \oplus B \oplus C_i \\
 &= A\bar{B}\bar{C}_i + \bar{A}B\bar{C}_i + \bar{A}\bar{B}C_i + ABC_i \\
 C_o &= AB + BC_i + AC_i
 \end{aligned}
 \tag{11.1}$$

It is often useful from an implementation perspective to define  $S$  and  $C_o$  as functions of some intermediate signals  $G$  (generate),  $D$  (delete), and  $P$  (propagate).<sup>1</sup>  $G = 1$  ( $D = 1$ ) ensures that a carry bit will be *generated* (*deleted*) at  $C_o$  independent of  $C_i$ , while  $P = 1$  guarantees that an incoming carry will propagate to  $C_o$ . Expressions for these signals can be derived from inspection of the truth table:

$$\begin{aligned}
 G &= AB \\
 D &= \bar{A}\bar{B} \\
 P &= A \oplus B
 \end{aligned}
 \tag{11.2}$$

We can rewrite  $S$  and  $C_o$  as functions of  $P$  and  $G$  (or  $D$ ):

$$\begin{aligned}
 C_o(G, P) &= G + PC_i \\
 S(G, P) &= P \oplus C_i
 \end{aligned}
 \tag{11.3}$$

Notice that  $G$  and  $P$  are only functions of  $A$  and  $B$  and are not dependent upon  $C_i$ . In a similar way, we can also derive expressions for  $S(D, P)$  and  $C_o(D, P)$ .

<sup>1</sup>Note that the propagate signal sometimes is also defined as the OR function of the inputs  $A$  and  $B$ —this condition guarantees that the input carry propagates to the output when  $A = B = 1$ , too. We will provide appropriate warning whenever this definition is used.

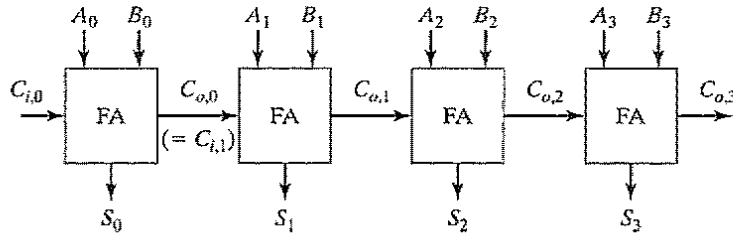


Figure 11-2 Four-bit ripple-carry adder: topology.

An  $N$ -bit adder can be constructed by cascading  $N$  full-adder (FA) circuits in series, connecting  $C_{o,k-1}$  to  $C_{i,k}$  for  $k = 1$  to  $N-1$ , and the first carry-in  $C_{i,0}$  to 0 (Figure 11-2). This configuration is called a *ripple-carry adder*, since the carry bit “ripples” from one stage to the other. The delay through the circuit depends upon the number of logic stages that must be traversed and is a function of the applied input signals. For some input signals, no rippling effect occurs at all, while for others, the carry has to ripple all the way from the *least significant bit (lsb)* to the *most significant bit (msb)*. The propagation delay of such a structure (also called the *critical path*) is defined as the *worst case delay over all possible input patterns*.

In the case of the ripple-carry adder, the worst case delay happens when a carry generated at the least significant bit position propagates all the way to the most significant bit position. This carry is finally consumed in the last stage to produce the sum. The delay is then proportional to the number of bits in the input words  $N$  and is approximated by

$$t_{adder} \approx (N-1)t_{carry} + t_{sum} \quad (11.4)$$

where  $t_{carry}$  and  $t_{sum}$  equal the propagation delays from  $C_i$  to  $C_o$  and  $S$ , respectively.<sup>2</sup>

### Example 11.1 Propagation Delay of Ripple-Carry Adder

Derive the values of  $A_k$  and  $B_k$  ( $k = 0 \dots N-1$ ) so that the worst case delay is obtained for the ripple-carry adder.

The worst case condition requires that a carry be generated at the *lsb* position. Since the input carry of the first full adder  $C_{i,0}$  is always 0, this both  $A_0$  and  $B_0$  must equal 1. All the other stages must be in propagate mode. Hence, either  $A_i$  or  $B_i$  must be high. Finally, we would like to physically measure the delay of a transition on the *msb* sum bit. Assuming an initial value of 0 for  $S_{N-1}$ , we must arrange a  $0 \rightarrow 1$  transition. This is achieved by setting both  $A_{N-1}$  and  $B_{N-1}$  to 0 (or 1), which yields a high sum bit given the incoming carry of 1.

For example, the following values for  $A$  and  $B$  trigger the worst case delay for an 8-bit addition:

$$A: 0000001; B: 0111111$$

<sup>2</sup>Equation (11.4) assumes that both the delay from the input signals  $A_0$  (or  $B_0$ ) to  $C_{o,0}$  for the *lsb*, and the  $C_i$ -to- $C_o$  delay for all other bits equal to  $t_{carry}$ .

To set-up the worst case delay transition, all the inputs can be kept constant with  $A_0$  undergoing a  $0 \rightarrow 1$  transition.

The left-most bit represents the *msb* in this binary representation. Observe that this is only one of the many worst case patterns. This case exercises the  $0 \rightarrow 1$  delay of the final sum. Derive several other cases that exercise the  $0 \rightarrow 1$  and  $1 \rightarrow 0$  transitions.

Two important conclusions can be drawn from Eq. (11.4):

- The propagation delay of the ripple-carry adder is *linearly* proportional to  $N$ . This property becomes increasingly important when designing adders for the wide data paths ( $N = 16 \dots 128$ ) that are desirable in current and future computers.
- When designing the full-adder cell for a fast ripple-carry adder, it is far more important to optimize  $t_{carry}$  than  $t_{sum}$  since the latter has only a minor influence on the total value of  $t_{adder}$ .

Before starting an in-depth discussion on the circuit design of full-adder cells, the following additional logic property of the full adder is worth mentioning:

**Inverting all inputs to a full adder results in inverted values for all outputs.**

This property, also called the *inverting property*, is expressed in the pair of equations

$$\begin{aligned}\bar{S}(A, B, C_i) &= S(\bar{A}, \bar{B}, \bar{C}_i) \\ \bar{C}_o(A, B, C_i) &= C_o(\bar{A}, \bar{B}, \bar{C}_i)\end{aligned}\tag{11.5}$$

and will be extremely useful when optimizing the speed of the ripple-carry adder. It states that the circuits of Figure 11-3 are identical.

### 11.3.2 The Full Adder: Circuit Design Considerations

#### Static Adder Circuit

One way to implement the full-adder circuit is to take the logic equations of Eq. (11.1) and translate them directly into complementary CMOS circuitry. Some logic manipulations can help to reduce the transistor count. For instance, it is advantageous to share some logic between the

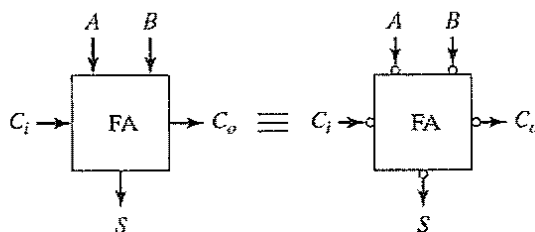


**Figure 11-3** Inverting property of the full adder. The circles in the schematics represent inverters.

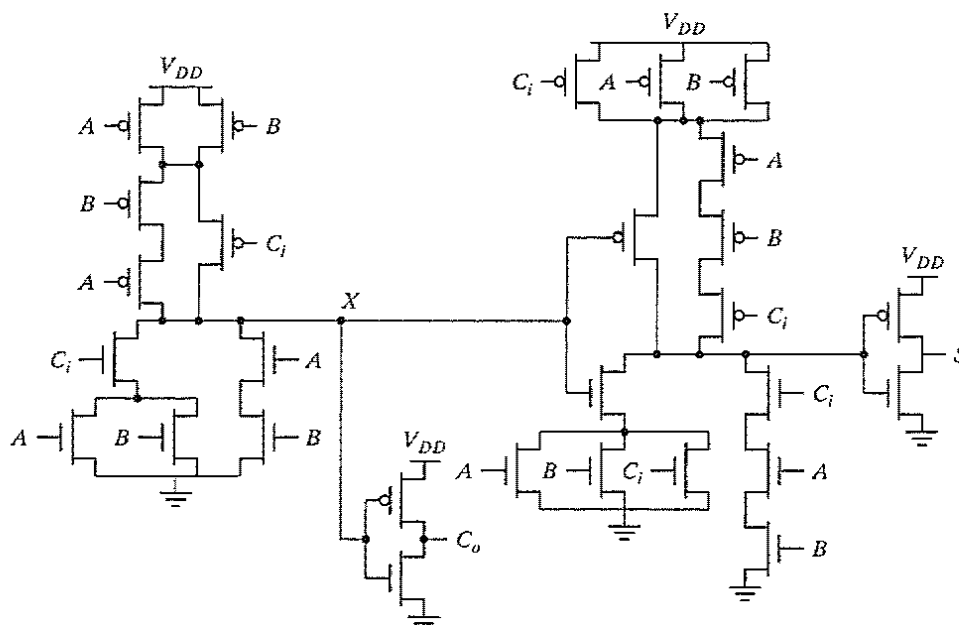


Figure 11-4 Complementary static CMOS implementation of full adder.

sum- and carry-generation subcircuits, as long as this does not slow down the carry generation, which is the most critical part, as stated previously. The following is an example of such a reorganized equation set:

$$C_o = AB + BC_i + AC_i$$

and

$$S = ABC_i + \overline{C_o}(A + B + C_i) \quad (11.6)$$

The equivalence with the original equations is easily verified. The corresponding adder design, using complementary static CMOS, is shown in Figure 11-4 and requires 28 transistors. In addition to consuming a large area, this circuit is slow:

- Tall PMOS transistor stacks are present in both *carry*- and *sum*-generation circuits.
- The intrinsic load capacitance of the  $C_o$  signal is large and consists of two diffusion and six gate capacitances, plus the wiring capacitance.
- The signal propagates through two inverting stages in the carry-generation circuit. As mentioned earlier, minimizing the carry-path delay is the prime goal of the designer of high-speed adder circuits. Given the small load (fan-out) at the output of the carry chain, having two logic stages is too high a number, and leads to extra delay.
- The sum generation requires one extra logic stage, but that is not that important, since a factor appears only once in the propagation delay of the ripple-carry adder of Eq. (11.4).

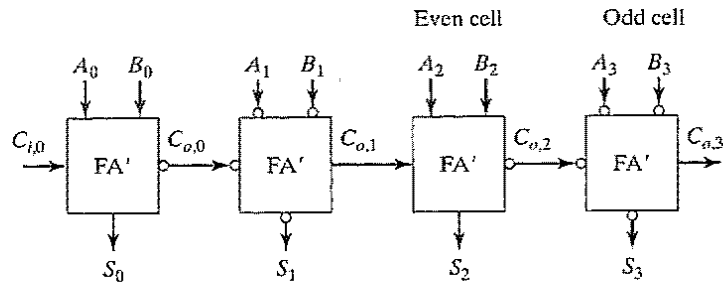


**Figure 11-5** Inverter elimination in carry path. FA' stands for a full adder without the inverter in the carry path.

Although slow, the circuit includes some smart design tricks. Notice that the first gate of the carry-generation circuit is designed with the  $C_i$  signal on the smaller PMOS stack, lowering its logical effort to 2. Also, the NMOS and PMOS transistors connected to  $C_i$  are placed as close as possible to the output of the gate. This is a direct application of a circuit-optimization technique discussed in Section 4.2—transistors on the critical path should be placed as close as possible to the output of the gate. For instance, in stage  $k$  of the adder, signals  $A_k$  and  $B_k$  are available and stable long before  $C_{i,k}$  ( $= C_{o,k-1}$ ) arrives after rippling through the previous stages. In this way, the capacitances of the internal nodes in the transistor chain are precharged or discharged in advance. On arrival of  $C_{i,k}$ , only the capacitance of node  $X$  has to be (dis)charged. Putting the  $C_{i,k}$  transistors closer to  $V_{DD}$  and GND would require not only the (dis)charging of the capacitance of node  $X$ , but also of the internal capacitances.

The speed of this circuit can now be improved gradually by using some of the adder properties discussed in the previous section. First, the number of inverting stages in the carry path can be reduced by exploiting the inverting property—inverting all the inputs of a full-adder cell also inverts all the outputs. This rule allows us to eliminate an inverter in a carry chain, as demonstrated in Figure 11-5.

### Mirror Adder Design

An improved adder circuit, also called the *mirror adder*, is shown in Figure 11-6 [Weste93]. Its operation is based on Eq. (11.3). The carry-generation circuitry is worth analyzing. First, the carry-inverting gate is eliminated, as suggested in the previous section. Secondly, the PDN and PUN networks of the gate are not dual. Instead, they form a clever implementation of the propagate/generate/delete function—when either  $D$  or  $G$  is high,  $\overline{C_o}$  is set to  $V_{DD}$  or GND, respectively. When the conditions for a *Propagate* are valid (or  $P$  is 1),<sup>3</sup> the incoming carry is propagated (in inverted format) to  $\overline{C_o}$ . This results in a considerable reduction in both area and delay. The analysis of the sum circuitry is left to the reader. The following observations are worth considering:

- This full-adder cell requires only 24 transistors.

<sup>3</sup>The  $P = A + B$  definition of the *propagate* signal is used here.

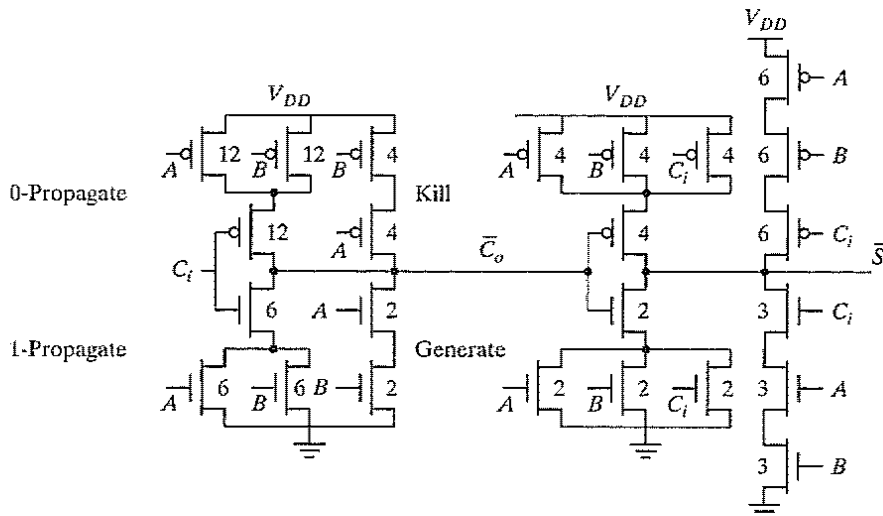


Figure 11-6 Mirror adder—circuit schematics.

- The NMOS and PMOS chains are completely symmetrical, which still yields correct operation due to *self-duality* of both the sum and carry functions. As a result, a maximum of two series transistors can be found in the carry-generation circuitry.
- The transistors connected to  $C_i$  are placed closest to the output of the gate.
- Only the transistors in the carry stage have to be optimized for speed. All transistors in the sum stage can be of minimum size. When laying out the cell, the most critical issue is the minimization of the capacitance at node  $\bar{C}_o$ . Shared diffusions reduce the stack node capacitances.
- In the adder cell of Figure 11-4, the inverter can be sized independently to drive the  $C_i$  input of the adder stage that follows. If the carry circuit in Figure 11-6 is symmetrically sized, each of its inputs has a logical effort of 2. This means that the optimal fan-out, sized for minimum delay, should be  $(4/2) = 2$ . However, the output of this stage drives two internal gate capacitances and six gate capacitances in the connecting adder cell. A clever solution to keep the transistor sizes the same in each stage is to increase the size of the carry stage to about three to four times the size of the sum stage. This maintains the optimal fan-out of 2. The resulting transistor sizes are annotated on Figure 11-6, where a PMOS/NMOS ratio of 2 is assumed.

### Transmission-Gate-Based Adder

A full adder can be designed to use multiplexers and XORs. While this is impractical in a complementary CMOS implementation, it becomes attractive when the multiplexers and XORs are implemented as transmission gates. A full-adder implementation based on this approach is shown in Figure 11-7 and uses 24 transistors. It is based on the *propagate-generate* model, introduced in Eq. (11.3). The propagate signal, which is the XOR of inputs  $A$  and  $B$ , is used to

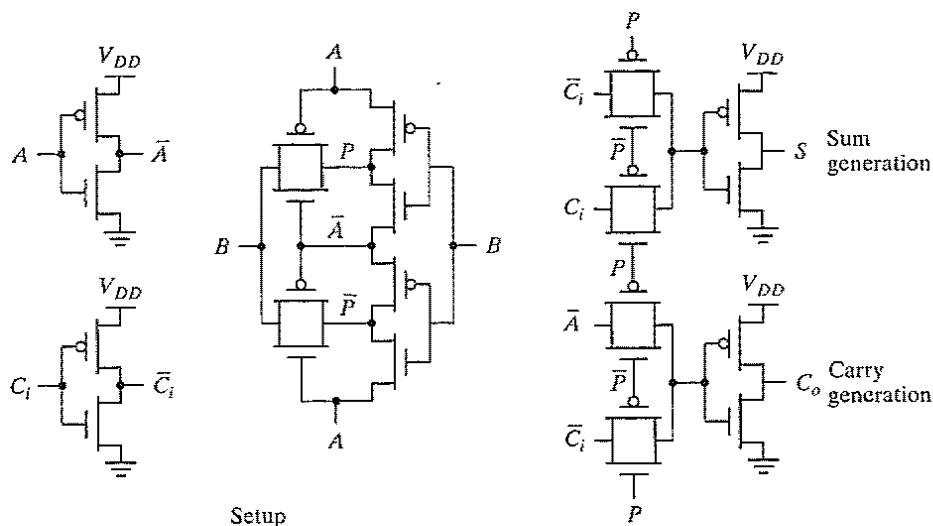


**Figure 11-7** Transmission-gate-based full-adder cell with sum and carry delays of similar value (after [Weste93]).

select the true or complementary value of the input carry as the new sum output. Based on the propagate signal, the output carry is either set to the input carry, or either one of inputs  $A$  or  $B$ . One of interesting features of such an adder is that it has similar delays for both sum and carry outputs.

### Manchester Carry-Chain Adder

The carry-propagation circuitry in Figure 11-7 can be simplified by adding generate and delete signals, as shown in Figure 11-8a. The propagate path is unchanged, and it passes  $C_i$  to the  $C_o$  output if the propagate signal ( $A_i \oplus B_i$ ) is true. If the propagate condition is not satisfied, the output is either pulled low by the  $D_i$  signal or pulled up by  $\bar{G}_i$ . The dynamic implementation (Figure 11-8b), makes even further simplification possible. Since the transitions in a dynamic circuit are monotonic, the transmission gates can be replaced by NMOS-only pass transistors. Precharging the output eliminates the need for the kill signal (for the case in which the carry chain propagates the complementary values of the carry signals).

A *Manchester carry-chain adder* uses a cascade of pass transistors to implement the carry chain [Kilburn60]. An example, based on the dynamic circuit version introduced in Figure 11-8, is shown in Figure 11-9. During the precharge phase ( $\phi = 0$ ), all intermediate nodes of the pass-transistor carry chain are precharged to  $V_{DD}$ . During evaluation, the  $A_k$  node is discharged when there is an incoming carry and the propagate signal  $P_k$  is high, or when the generate signal for stage  $k$  ( $G_k$ ) is high.

Figure 11-10 shows an example layout of the Manchester carry chain in stick-diagram format. The datapath layout consists of three rows of cells organized in bit-sliced style: The top row

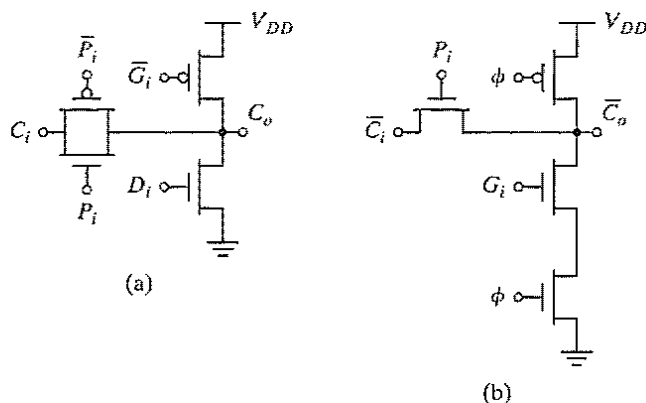


**Figure 11-8** Manchester carry gates. (a) Static, using propagate, generate, and kill, (b) dynamic implementation, using only propagate and generate signals.

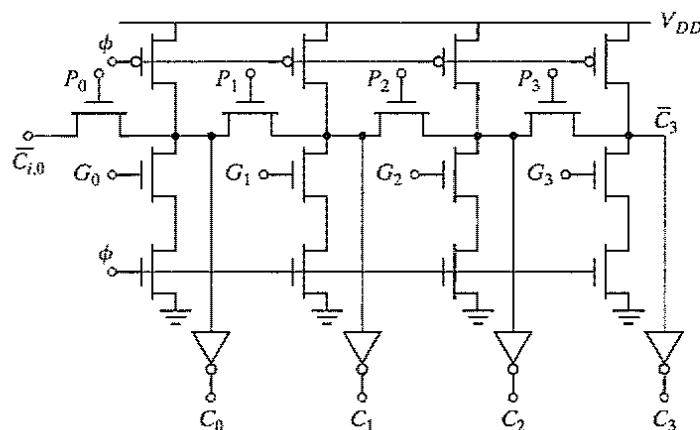


**Figure 11-9** Manchester carry-chain adder in dynamic logic (four-bit section).

of cells computes the propagate and generate signals, the middle row propagates the carry from left to right, and the bottom row generates the final sums.

The worst case delay of the carry chain of the adder in Figure 11-9 is modeled by the linearized  $RC$  network of Figure 11-11. As derived in Chapter 4, the propagation delay of such a network equals

$$t_p = 0.69 \sum_{i=1}^N C_i \left( \sum_{j=1}^i R_j \right) = 0.69 \frac{N(N+1)}{2} RC \quad (11.7)$$

when all  $C_i = C$  and  $R_j = R$ .

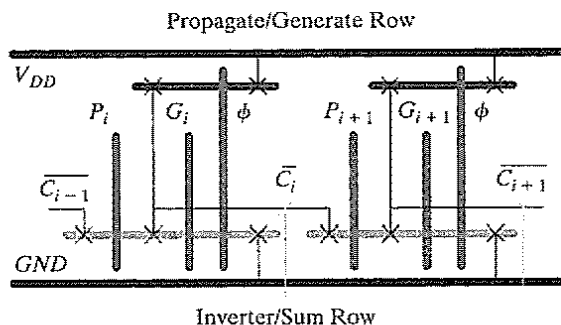


Figure 11-10 Stick diagram of two bits of a Manchester carry chain.

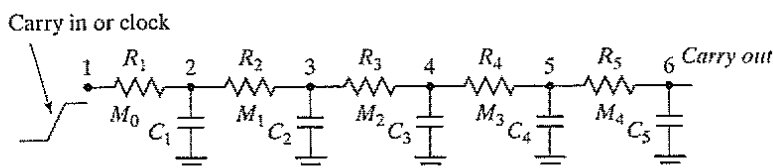


Figure 11-11 Equivalent network to determine propagation delay of a carry chain.

### Example 11.2 Sizing of Manchester Carry Chain

The capacitance per node on the carry chain equals four diffusion capacitances, one inverter input capacitance, and the wiring capacitance proportional to the size of the cell. The inverter and the PMOS precharging transistor can be kept at unit size. Together with the wire capacitance, the fixed capacitance can be estimated as 15 fF (for our technology). If a unit-sized transistor with width  $W_0$  has a resistance of 10 k $\Omega$  and a diffusion capacitance of 2 fF, then the  $RC$  time constant for a chain of transistors of width  $W$  is

$$RC = \left( 6 \text{ fF} \cdot \frac{W}{W_0} + 15 \text{ fF} \right) \cdot 10 \text{ k}\Omega \cdot \frac{W_0}{W}$$

Increasing the transistor width reduces this time constant, but it also loads the gates in the previous stage. Therefore, the transistor size is limited by the input loading capacitance.

Unfortunately, the distributed  $RC$ -nature of the carry chain results in a propagation delay that is quadratic in the number of bits  $N$ . To avoid this, it is necessary to insert signal-buffering inverters. The optimum number of stages per buffer depends on the equivalent resistance of the inverter and the resistance and capacitance of the pass transistors, as was discussed in Chapter 9. In our technology, and in most other practical cases, this number is between 3 and 4. Adding the inverter makes the overall propagation delay a linear function of  $N$ , as is the case with ripple-carry adders.

### 11.3.3 The Binary Adder: Logic Design Considerations

The ripple-carry adder is only practical for the implementation of additions with a relatively small word length. Most desktop computers use word lengths of 32 bits, while servers require 64; very fast computers, such as mainframes, supercomputers, or multimedia processors (e.g., the Sony PlayStation2) [Suzuoki99], require word lengths of up to 128 bits. The linear dependence of the adder speed on the number of bits makes the usage of ripple adders rather impractical. Logic optimizations are therefore necessary, resulting in adders with  $t_p < O(N)$ . We briefly discuss a number of those in the sections that follow. We concentrate on the circuit design implications, since most of the presented structures are well known from the traditional logic design literature.

#### The Carry-Bypass Adder

Consider the four-bit adder block of Figure 11-12a. Suppose that the values of  $A_k$  and  $B_k$  ( $k = 0 \dots 3$ ) are such that all propagate signals  $P_k$  ( $k = 0 \dots 3$ ) are high. An incoming carry  $C_{i,0} = 1$  propagates under those conditions through the complete adder chain and causes an outgoing carry  $C_{o,3} = 1$ . In other words,

$$\begin{aligned} \text{if } (P_0 P_1 P_2 P_3 = 1) \text{ then } C_{o,3} &= C_{i,0} \\ \text{else either DELETE or GENERATE occurred} \end{aligned} \quad (11.8)$$

This information can be used to speed up the operation of the adder, as shown in Figure 11-12b. When  $BP = P_0 P_1 P_2 P_3 = 1$ , the incoming carry is forwarded immediately to the next block through the bypass transistor  $M_b$ —hence the name *carry-bypass adder* or *carry-skip adder* [Lehman62]. If this is not the case, the carry is obtained by way of the normal route.

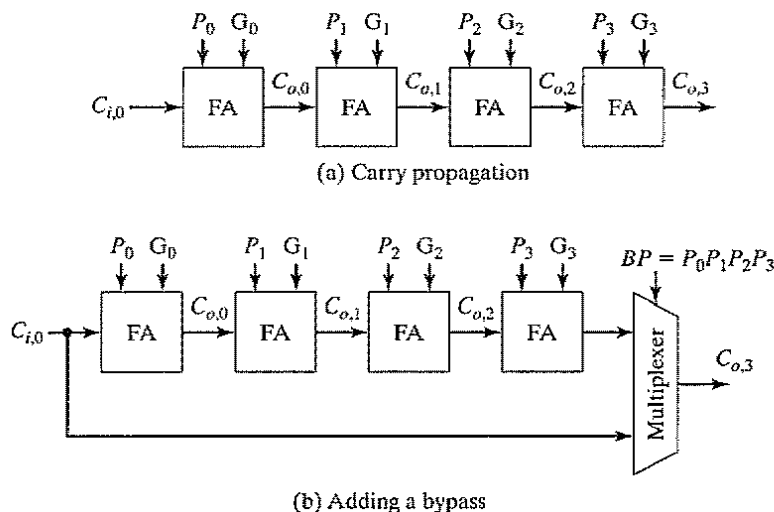


Figure 11-12 Carry-bypass structure—basic concept.

**Example 11.3 Carry Bypass in Manchester Carry-Chain Adder**

Figure 11-13 shows the possible carry-propagation paths when the full-adder circuit is implemented in Manchester-carry style. This picture demonstrates how the bypass speeds up the addition: The carry propagates either through the bypass path, or a carry is generated somewhere in the chain. In both cases, the delay is smaller than the normal ripple configuration. The area overhead incurred by adding the bypass path is small and typically ranges between 10 and 20%. However, adding the bypass path breaks the regular bit-slice structure (as was present in Figure 11-10).

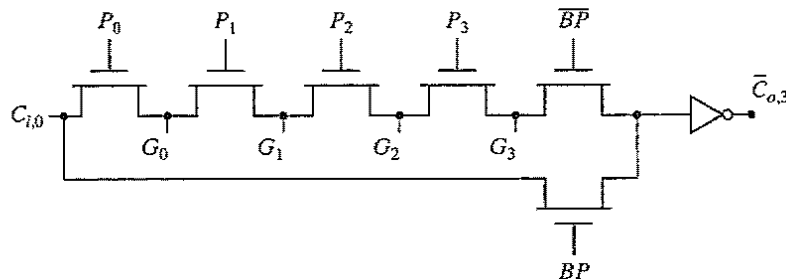


**Figure 11-13** Manchester carry-chain implementation of bypass adder.

Let us now compute the delay of an  $N$ -bit adder. At first, we assume that the total adder is divided in  $(N/M)$  equal-length bypass stages, each of which contains  $M$  bits. An approximating expression for the total propagation time can be derived from Figure 11-14a and is given in Eq. (11.9). Namely,

$$t_p = t_{\text{setup}} + M t_{\text{carry}} + \left(\frac{N}{M} - 1\right) t_{\text{bypass}} + (M - 1) t_{\text{carry}} + t_{\text{sum}} \quad (11.9)$$

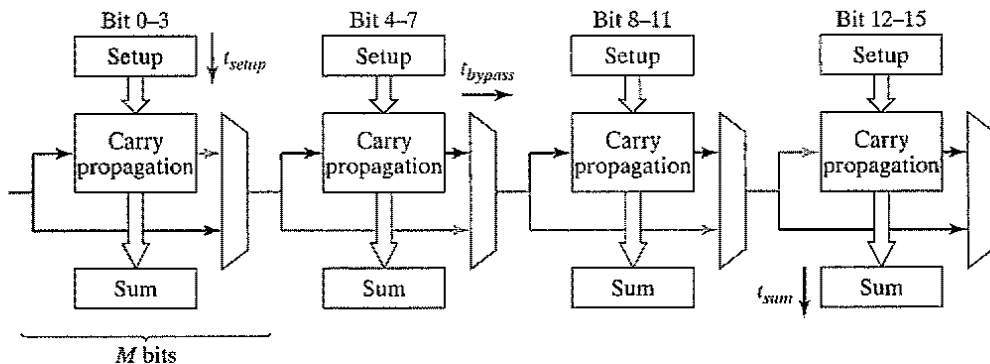


**Figure 11-14** ( $N = 16$ ) carry-bypass adder: composition. The worst case delay path is shaded in gray.

with the composing parameters defined as follows:

- $t_{\text{setup}}$ : the fixed overhead time to create the generate and propagate signals.
- $t_{\text{carry}}$ : the propagation delay through a single bit. The worst case carry-propagation delay through a single stage of  $M$  bits is approximately  $M$  times larger.
- $t_{\text{bypass}}$ : the propagation delay through the bypass multiplexer of a single stage.
- $t_{\text{sum}}$ : the time to generate the sum of the final stage.

The critical path is shaded in gray on the block diagram of Figure 11-14. From Eq. (11.9), it follows that  $t_p$  is still linear in the number of bits  $N$ , since in the worst case, the carry is generated at the first bit position, ripples through the first block, skips around  $(N/M - 2)$  bypass stages, and is consumed at the last bit position without generating an output carry. The optimal number of bits per skip block is determined by technological parameters such as the extra delay of the bypass-selecting multiplexer, the buffering requirements in the carry chain, and the ratio of the delay through the ripple and the bypass paths.

Although still linear, the slope of the delay function increases in a more gradual fashion than for the ripple-carry adder, as pictured in Figure 11-15. This difference is substantial for larger adders. Notice that the ripple adder is actually faster for small values of  $N$ , for which the overhead of the extra bypass multiplexer makes the bypass structure not interesting. The cross-over point depends upon technology considerations and is normally situated between four and eight bits.

#### Problem 11.1 Delay of Carry-Skip Adder

Determine an input pattern that triggers the worst case delay in a 16-bit ( $4 \times 4$ ) carry-bypass adder. Assuming that  $t_{\text{carry}} = t_{\text{setup}} = t_{\text{skip}} = t_{\text{sum}} = 1$ , determine the delay and compare it with that of a normal ripple adder.

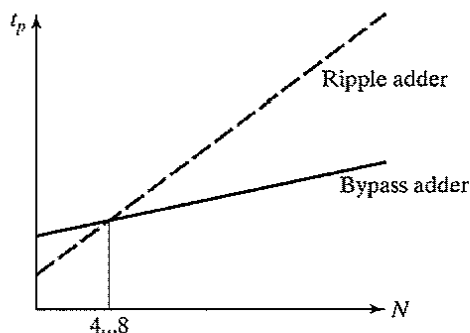


Figure 11-15 Propagation delay of ripple-carry versus carry-bypass adder.

### The Linear Carry-Select Adder

In a ripple-carry adder, every full-adder cell has to wait for the incoming carry before an outgoing carry can be generated. One way to get around this linear dependency is to anticipate both possible values of the carry input and evaluate the result for both possibilities in advance. Once the real value of the incoming carry is known, the correct result is easily selected with a simple multiplexer stage. An implementation of this idea, appropriately called the *carry-select adder* [Bedrij62], is demonstrated in Figure 11-16. Consider the block of adders, which is adding bits  $k$  to  $k + 3$ . Instead of waiting on the arrival of the output carry of bit  $k - 1$ , both the 0 and 1 possibilities are analyzed. From a circuit point of view, this means that two carry paths are implemented. When  $C_{o,k-1}$  finally settles, either the result of the 0 or the 1 path is selected by the multiplexer, which can be performed with a minimal delay. As is evident from Figure 11-16, the hardware overhead of the carry-select adder is restricted to an additional carry path and a multiplexer, and equals about 30% with respect to a ripple-carry structure.

A full carry-select adder is now constructed by chaining a number of equal-length adder stages, as in the carry-bypass approach (see Figure 11-17). The critical path is shaded in gray. From inspection of the circuit, we can derive a first-order model of the worst case propagation delay of the module, written as

$$t_{add} = t_{setup} + Mt_{carry} + \left(\frac{N}{M}\right)t_{mux} + t_{sum} \quad (11.10)$$

where  $t_{setup}$ ,  $t_{sum}$ , and  $t_{mux}$  are fixed delays and  $N$  and  $M$  represent the total number of bits, and the number of bits per stage, respectively.  $t_{carry}$  is the delay of the carry through a single full-adder cell. The carry delay through a single block is proportional to the length of that stage or equals  $Mt_{carry}$ .

The propagation delay of the adder is, again, linearly proportional to  $N$  (Eq. (11.10)). The reason for this linear behavior is that the *block-select* signal that selects between the 0 and 1 solutions still has to ripple through all stages in the worst case.

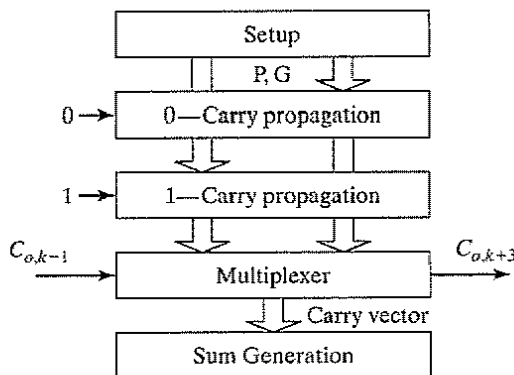


Figure 11-16 Four-bit carry-select module—topology.

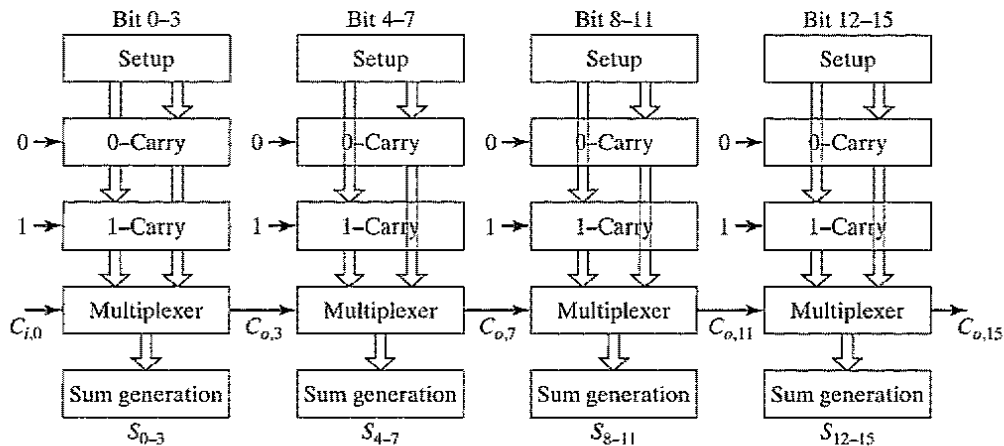


Figure 11-17 Sixteen-bit, linear carry-select adder. The critical path is shaded in gray.

### Problem 11.2 Linear Carry-Select Delay

Determine the delay of a 16-bit linear carry-select adder by using unit delays for all cells. Compare the result with that of Problem 11.1. Compare various block configurations as well.

### The Square-Root Carry-Select Adder

The next structure illustrates how an alert designer can make a major impact. To optimize a design, it is essential to locate the critical timing path first. Consider the case of a 16-bit linear carry-select adder. To simplify the discussion, assume that the full-adder and multiplexer cells have identical propagation delays equal to a normalized value of 1. The worst case arrival times of the signals at the different network nodes with respect to the time the input is applied are marked and annotated on Figure 11-18a. This analysis demonstrates that the critical path of the adder ripples through the multiplexer networks of the subsequent stages.

One striking opportunity is readily apparent. Consider the multiplexer gate in the last adder stage. The inputs to this multiplexer are the two carry chains of the block and the block-multiplexer signal from the previous stage. A major mismatch between the arrival times of the signals can be observed. The results of the carry chains are stable long before the multiplexer signal arrives. It makes sense to equalize the delay through both paths. This can be achieved by progressively adding more bits to the subsequent stages in the adder, requiring more time for the generation of the carry signals. For example, the first stage can add 2 bits, the second contains 3, the third has 4, and so forth, as demonstrated in Figure 11-18b. The annotated arrival times show that this adder topology is faster than the linear organization, even though an extra stage is needed. In fact, the same propagation delay is also valid for a 20-bit adder. Observe that the discrepancy in arrival times at the multiplexer nodes has been eliminated.

In effect, the simple trick of making the adder stages progressively longer results in an adder structure with sublinear delay characteristics. This is illustrated by the following analysis:

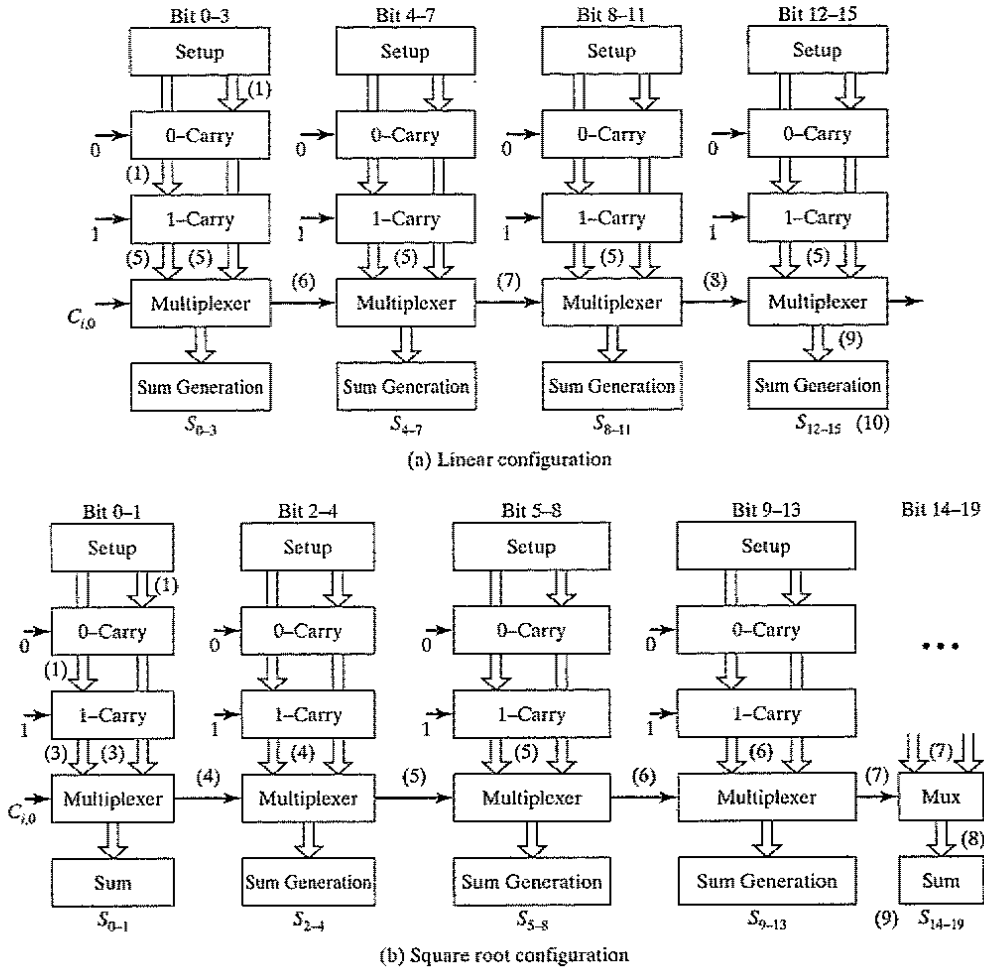


**Figure 11-18** Worst case signal arrival times in carry-select adders. The signal arrival times are marked in parentheses.

Assume that an  $N$ -bit adder contains  $P$  stages, and the first stage adds  $M$  bits. An additional bit is added to each subsequent stage. The following relation then holds:

$$\begin{aligned}
 N &= M + (M + 1) + (M + 2) + (M + 3) + \dots + (M + P - 1) \\
 &= MP + \frac{P(P - 1)}{2} = \frac{P^2}{2} + P\left(M - \frac{1}{2}\right)
 \end{aligned}
 \quad (11.11)$$

If  $M \ll N$  (e.g.,  $M = 2$ , and  $N = 64$ ), the first term dominates, and Eq. (11.11) can be simplified to

$$N \approx \frac{P^2}{2}
 \quad (11.12)$$

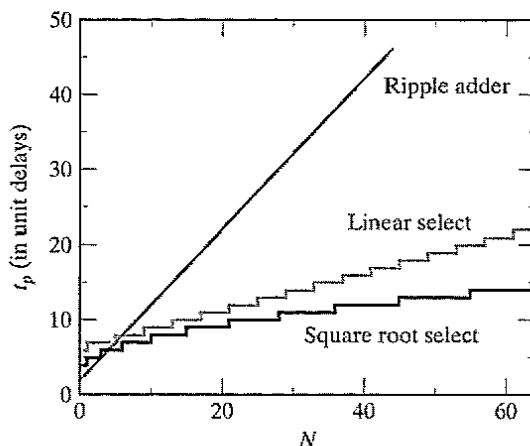


**Figure 11-19** Propagation delay of square-root carry-select adder versus linear ripple and select adders. The unit delay model is used to model the cell delays.

or

$$P \approx \sqrt{2N} \quad (11.13)$$

Equation (11.13) can be used to express  $t_{add}$  as a function of  $N$  by rewriting Eq. (11.10):

$$t_{add} = t_{setup} + Mt_{carry} + (\sqrt{2N})t_{mux} + t_{sum}. \quad (11.14)$$

The delay is proportional to  $\sqrt{N}$  for large adders ( $N \gg M$ ), or  $t_{add} = O(\sqrt{N})$ . This square-root relation has a major impact, which is illustrated in Figure 11-19, where the delays of both the linear and square-root select adders are plotted as a function of  $N$ . It can be observed that for large values of  $N$ ,  $t_{add}$  becomes almost a constant.

### Problem 11.3 Unequal Bypass Groups in Carry-Bypass Adder

A careful reader might be interested in applying the previous technique to carry-bypass adders. We saw earlier that their delay is a linear function of a number of bits. Can they be modified to achieve better than linear delay by using variable group sizes?

It does make sense to make the consecutive groups gradually larger. However, the technique used in carry-select adders does not directly apply to this case, and a progressive increase in stage sizes eventually increases the delay. Consider a carry-bypass adder in which the last stage is the largest: The carry signal that propagates through that stage and gets consumed at the *msb* position (with no chance of bypassing it) is on the critical path for the sum generation. Increasing the size of the last group does not help the problem.

Based on this discussion and assuming constant delays for carry and bypass gates, sketch the profile of the carry bypass network that achieves a delay that is better than linear.

### The Carry-Lookahead Adder\*

**The Monolithic Lookahead Adder** When designing even faster adders, it is essential to get around the rippling effect of the carry that is still present in one form or another in both the carry-bypass and carry-select adders. The *carry-lookahead* principle offers a possible way to do so [Weinberger56, MacSorley61]. As stated before, the following relation holds for each bit position in an  $N$ -bit adder:

$$C_{o,k} = f(A_k, B_k, C_{o,k-1}) = G_k + P_k C_{o,k-1} \quad (11.15)$$

The dependency between  $C_{o,k}$  and  $C_{o,k-1}$  can be eliminated by expanding  $C_{o,k-1}$ :

$$C_{o,k} = G_k + P_k(G_{k-1} + P_{k-1}C_{o,k-2}) \quad (11.16)$$

In a fully expanded form,

$$C_{o,k} = G_k + P_k(G_{k-1} + P_{k-1}(\dots + P_1(G_0 + P_0 C_{i,0}))) \quad (11.17)$$

with  $C_{i,0}$  typically equal to 0.

This expanded relationship can be used to implement an  $N$ -bit adder. For every bit, the carry and sum outputs are independent of the previous bits. The ripple effect has thus been effectively eliminated, and the addition time should be independent of the number of bits. A block diagram of the overall composition of a carry-lookahead adder is shown in Figure 11-20.

Such a high-level model contains some hidden dependencies. When we study the detailed schematics of the adder, it becomes obvious that the constant addition time is wishful thinking and that the real delay is at least increasing linearly with the number of bits. This is illustrated in Figure 11-21, where a possible circuit implementation of Eq. (11.17) is shown for  $N = 4$ . Note that the circuit exploits the self-duality and the recursivity of the carry-lookahead equation to build a mirror structure, similar in style to the single-bit full adder of Figure 11-6.<sup>4</sup> The large fan-in of the circuit makes it prohibitively slow for larger values of  $N$ . Implementing it with simpler gates requires multiple logic levels. In both cases, the propagation delay increases. Further-

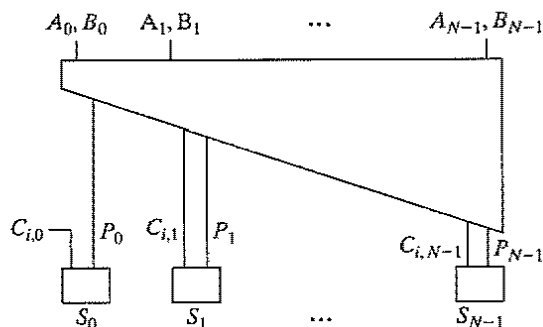


Figure 11-20 Conceptual diagram of a carry-lookahead adder.

<sup>4</sup>Similar to the mirror adder, this circuit requires that the *Propagate* signal be defined as  $P = A + B$ .

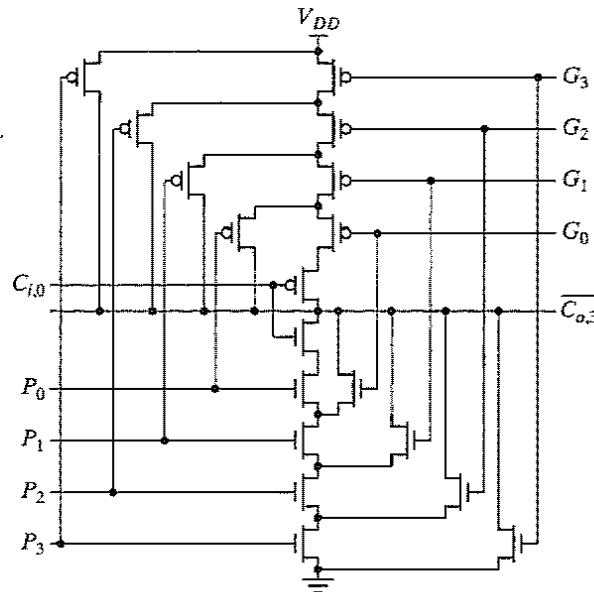


**Figure 11-21** Schematic diagram of mirror implementation of four-bit lookahead adder (from [Weste85]).

more, the fan-out on some of the signals tends to grow excessively, slowing down the adder even more. For instance, the signals  $G_0$  and  $P_0$  appear in the expression for every one of the subsequent bits. Hence, the capacitance on these lines is substantial. Finally, the area of the implementation grows progressively with  $N$ . Therefore, the lookahead structure suggested by Eq. (11.16) is only useful for small values of  $N$  ( $\leq 4$ ).

**The Logarithmic Lookahead Adder—Concept** For a carry-lookahead group of  $N$  bits, the transistor implementation has  $N + 1$  parallel branches with up to  $N + 1$  transistors in the stack. Since wide gates and large stacks display poor performance, the carry-lookahead computation has to be limited to up to two or four bits in practice. In order to build very fast adders, it is necessary to organize carry propagation and generation into recursive trees. A more effective implementation is obtained by hierarchically decomposing the carry propagation into subgroups of  $N$  bits:

$$\begin{aligned}
 C_{o,0} &= G_0 + P_0 C_{i,0} \\
 C_{o,1} &= G_1 + P_1 G_0 + P_1 P_0 C_{i,0} = (G_1 + P_1 G_0) + (P_1 P_0) C_{i,0} = G_{1:0} + P_{1:0} C_{i,0} \\
 C_{o,2} &= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{i,0} = G_2 + P_2 C_{o,1} \\
 C_{o,3} &= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_{i,0} \\
 &= (G_3 + P_3 G_2) + (P_3 P_2) C_{o,1} = G_{3:2} + P_{3:2} C_{o,1}
 \end{aligned} \tag{11.18}$$

In Eq. (11.18), the carry-propagation process is decomposed into subgroups of two bits.  $G_{i:j}$  and  $P_{i:j}$  denote the generate and propagate functions, respectively, for a group of bits (from bit positions  $i$  to  $j$ ). Therefore, we call them *block generate* and *propagate* signals.  $G_{i:j}$  equals 1 if the group generates a carry, independent of the incoming carry. The block propagate  $P_{i:j}$  is true if an incoming carry propagates through the complete group. This condition is equivalent to the carry bypass, discussed earlier. For example,  $G_{3:2}$  is equal to 1 when a carry either is generated at bit position 3 or is generated at position 2 and propagated through position 3, or  $G_{3:2} = G_3 + P_3G_2$ .  $P_{3:2}$  is true when an incoming carry propagates through both bit positions, or  $P_{3:2} = P_3P_2$ .

Note that the format of the new expression for the carry is equivalent to the original one, except that the generate and propagate signals are replaced with block generate and propagate signals. The notation  $G_{i:j}$  and  $P_{i:j}$  generalizes the original carry equations, since  $G_i = G_{i:i}$  and  $P_i = P_{i:i}$ . Another generalization is possible by treating the generate and propagate functions as a pair  $(G_{i:j}, P_{i:j})$ , rather than considering them as separate functions. A new Boolean operator, called the *dot* operator ( $\cdot$ ), can be introduced. This operator on the pairs and allows for the combination and manipulation of blocks of bits:

$$(G, P) \cdot (G', P') = (G + PG', PP') \quad (11.19)$$

Using this operator we can now decompose  $(G_{3:2}, P_{3:2}) = (G_3, P_3) \cdot (G_2, P_2)$ . The dot operator obeys the associative property, but it is not commutative.

#### Example 11.4 Ripple-Carry Adder Expressed by Using the Dot Operator

With the dot operator, a four-bit ripple carry adder can be re-written as

$$(C_{a,3}, 0) = [(G_3, P_3) \cdot (G_2, P_2) \cdot (G_1, P_1) \cdot (G_0, P_0)] \cdot (C_{i,0}, 0)$$

The associative property allows us to rewrite this function and express  $C_{a,3}$  as a function of 2 group carries:

$$\begin{aligned} (G_{3:0}, P_{3:0}) &= [(G_3, P_3) \cdot (G_2, P_2)] \cdot [(G_1, P_1) \cdot (G_0, P_0)] \\ &= (G_{3:2}, P_{3:2}) \cdot (G_{1:0}, P_{1:0}) \end{aligned}$$

By exploiting the associative property of the dot operator, a tree can be constructed that effectively computes the carries at all  $2^i - 1$  positions (that is, 1, 3, 7, 15, etc.) for  $i = 1 \dots \log_2(N)$ . The crucial advantage is that the computation of the carry at position  $2^i - 1$  takes only  $\log_2(N)$  steps. In other words, the output carry of an  $N$ -bit adder can be computed in  $\log_2(N)$  time. This is a major improvement over the previously described adders. For example, for an adder of 64 bits, the propagation delay of a linear adder is proportional to 64. For a square-root select adder, it is reduced to 8, while, for a logarithmic adder, the proportionality constant is 6. This is illustrated in Figure 11-22, which shows the block diagram of a 16-bit logarithmic adder. The carry at position 15 is computed by combining the results of blocks (0:7) and (8:15). Each of these, in turn, is

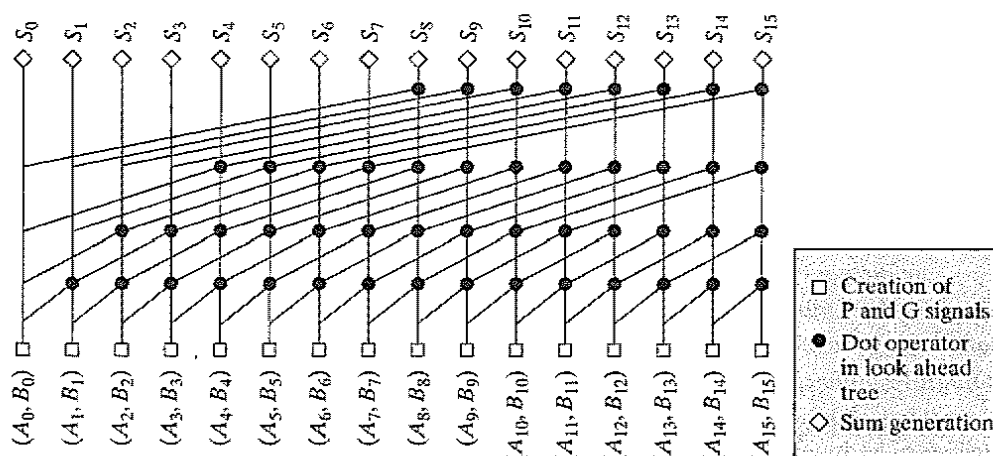


Figure 11-22: Schematic diagram for Kogge–Stone 16-bit lookahead logarithmic adder.

composed hierarchically. For instance, (0:7) is the composition of (0:3) and (4:7), while (0:3) consists of (0:1) and (2:3), etc.

Computing the carries at just the  $2^i - 1$  positions is obviously not sufficient. It is necessary to derive the carry signals at the intermediate positions as well. One way to accomplish this is by replicating the tree at every bit position, as illustrated in Figure 11-22 for  $N = 16$ . For instance, the carry at position 6 is computed by combining the results of blocks (6:3) and (2:0). This complete structure, which frequently is referred to as a *Kogge–Stone tree* [Kogge73], is a member of the *radix-2* class of trees. Radix-2 means that the tree is binary: It combines two carry words at a time at each level of hierarchy. The total adder requires 49 complex logic gates each to implement the dot operator. In addition, 16 logic modules are needed for the generation of the propagate and generate signals at the first level ( $P_i$  and  $G_i$ ), as well as 16 sum-generation gates.

#### Design Example—Implementing a Lookahead Adder in Dynamic Logic

The combination of carry-lookahead (CLA) techniques and dynamic logic seems to be ideal when very high performance is the ultimate goal. It is therefore useful to walk through the complete design of a dynamic CLA.

The first module generates the propagate and generate signal, as shown in Figure 11-23. The addition of a separate inverter to drive the keeper represents a small twist. This approach is beneficial in gates that drive a sizable fan-out. By decoupling its driver from the fan-out it allows for a quick disengagement of the keeper after the transition starts. The inverter that is driving successive logic gates, on the other hand, is optimized to drive a fan-out of two (for  $G$  outputs) or three (for  $P$  outputs) NMOS pull-down networks.

Each of the black dots in Figure 11-22 represents two gates that compute the block-level propagate and generate signals, as shown in Figure 11-24. Since these gates are not located at the beginning of the pipeline, the evaluation transistor (also called the *foot switch*) is optional, as discussed in Chapter 6. This approach is commonly used in dynamic datapaths. During the precharge phase, all the outputs of the domino gates are guaranteed to be low, turning off any discharge path in the succeeding domino stages. Elimination of the foot switch in any stage other than the first lowers the logical effort of the gates and speeds up

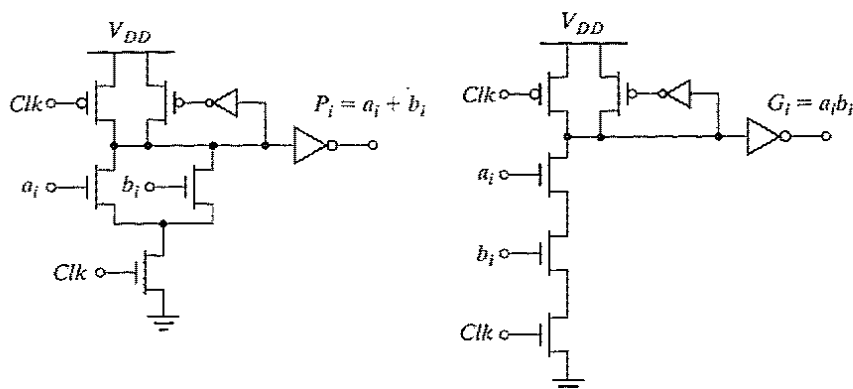


**Figure 11-23** Dynamic Implementation of propagate and generate signals.

the evaluation. For example, the propagate gate in Figure 11-24 has a logical effort of two-thirds instead of unity (assuming that the inverter is symmetrical). However, there is a drawback to this method. During the precharge phase, a short-circuit current exists in all gates without a foot switch until their inputs get precharged. To avoid the short-circuit current, the clock at each stage  $Clk_k$  is delayed from the previous stage. This approach is called *clock-delayed domino* as was introduced in Chapter 10. Note that the clock is delayed by the same amount for all bit-slices per stage, thus simplifying the implementation.

By putting together seven stages of logic in a bit-sliced fashion— $P$ - $G$  generation, followed by six dot operators—a 64-bit adder can be constructed. The only logic stage that is missing to complete the dynamic adder design is the final sum generation. The sum generation requires an XOR function, which is not easily built in domino logic. Static XOR gates could be used, but these produce nonmonotonic transitions and thus cannot be used to drive other domino gates. This might not be a problem per se, since the sum generation typically is the final stage of the addition operation. However, the latch that follows the sum generation cannot be transparent, because this could cause a violation of the transition rule for the succeeding domino gates.

One way of implementing the sum in domino logic is through *sum selection*, in which both care for the sum are computed as  $S_i^0 = a_i \oplus b_i$  and  $S_i^1 = a_i \odot b_i$ . The dynamic gate of Figure 11-25 is then used to select one of these possibilities, based on the incoming carry.

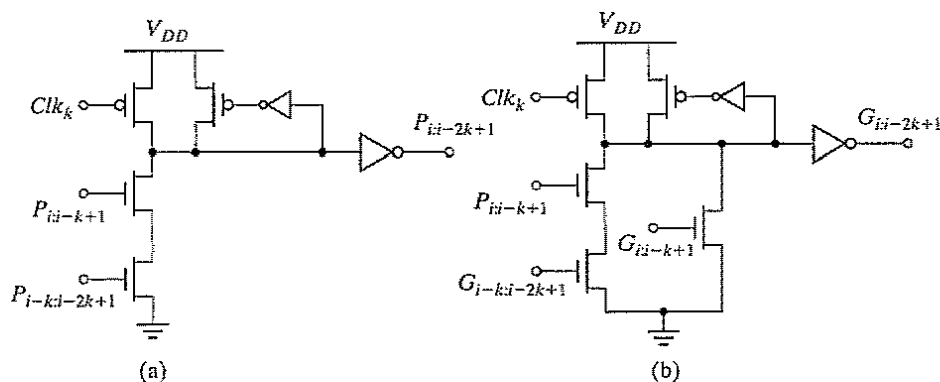


**Figure 11-24** Implementation of the dot operator in dynamic logic: (a) propagate and (b) generate logic at stage  $k$  and bit position  $i$  (see Figure 11-22).

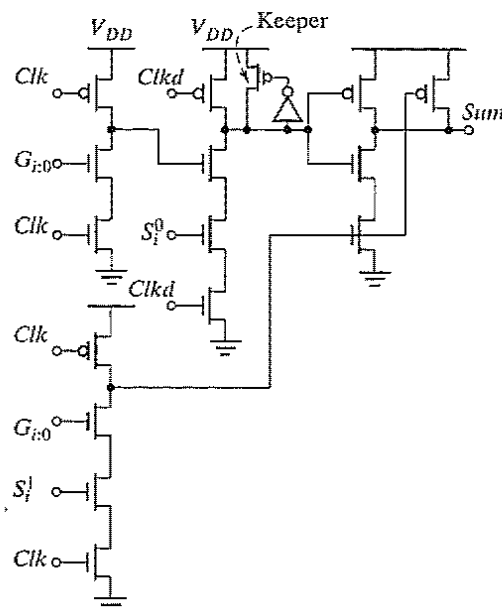


Figure 11-25 Sum select in dynamic logic.

The implementation of the multiplexer gate requires three logic levels, because no complementary carry is available in domino logic. As usual, keepers should be placed at all dynamic nodes, but the one shown in the figure is absolutely critical. The first two dynamic stages (on top) violate the dynamic logic design rules: two dynamic gates are cascaded without introducing an inverter. A glitch might happen at the output of the second gate if both gates are evaluated with the same clock. Delaying the clock of the second gate (*Clkd*) helps to address this issue, although the delayed clock presents a *hard-timing edge*—all the inputs to the second gate must have finished their transition before the clock raises.

It should be emphasized that when a designer chooses to use these design techniques, circuit robustness cannot be compromised. All clocks are subject to random skew, and the delayed clock must have enough margin to absorb the worst case skew. If *Clkd* arrives early, the adder will malfunction. However, proper sizing of the keeper may add a kind of safety net to the design. It can suppress the glitch at the output of the second stage, in case the clock arrives early. The keeper must be sized large enough to minimize this glitch, but not so large that it would compromise performance.

Another option is to implement the adder in differential domino logic, where both signal polarities are available, and the inversion problem is avoided. However, the overall design would be quite power hungry. ■

#### Problem 11.4 Static Adder Tree Design

Design a 16-bit carry-lookahead adder tree in static complementary CMOS. Design the lookahead tree by using inverted logic gates, avoiding the addition of inverters. Highlight the critical path of this adder. What are the logical efforts of gates along the critical path? How would you size it for minimal delay?

**Logarithmic Lookahead Adder—Alternatives** Designers of fast adders sometimes revert to other styles of tree structures as they trade off for area, power, or performance. We briefly discuss the Brent–Kung adder and the radix-4 adder, two of the more common alternative structures.

The Kogge–Stone tree of Figure 11-22 has some interesting properties. First, its interconnect structure is regular, which makes implementation quite easy. Furthermore, the fan-out throughout the tree is fairly constant, especially on the critical paths. The task of sizing the transistors for optimal performance is therefore simplified. At the same time, however, the replication of the carry trees to generate the intermediate carries comes at a large cost in terms of both area and power. Designers sometimes trade off some delay for area and power by choosing less complex trees. A simpler tree structure computes only the carries to the powers-of-two bit positions [Brent82], as illustrated in Figure 11-26 for  $N = 16$ .

The forward binary tree realizes the carry signals only at positions  $2^N - 1$ :

$$\begin{aligned}
 (C_{0,0}, 0) &= (G_0, P_0) \cdot (C_{i,0}, 0) \\
 (C_{0,1}, 0) &= [(G_1, P_1) \cdot (G_0, P_0)] \cdot (C_{i,0}, 0) = (G_{1:0}, P_{1:0}) \cdot (C_{i,0}, 0) \\
 (C_{0,3}, 0) &= [(G_{3:2}, P_{3:2}) \cdot (G_{1:0}, P_{1:0})] \cdot (C_{i,0}, 0) = (G_{3:0}, P_{3:0}) \cdot (C_{i,0}, 0) \\
 (C_{0,7}, 0) &= [(G_{7:4}, P_{7:4}) \cdot (G_{3:0}, P_{3:0})] \cdot (C_{i,0}, 0) = (G_{7:0}, P_{7:0}) \cdot (C_{i,0}, 0) \\
 &\dots
 \end{aligned} \tag{11.20}$$

The forward binary-tree structure is not sufficient to generate the complete set of carry bits. An *inverse binary tree* is needed to realize the other carry bits (shown in gray lines in Figure 11-26). This structure combines intermediate results to produce the remaining carry bits. It is left for the reader to verify that this structure produces the correct expressions for all carry

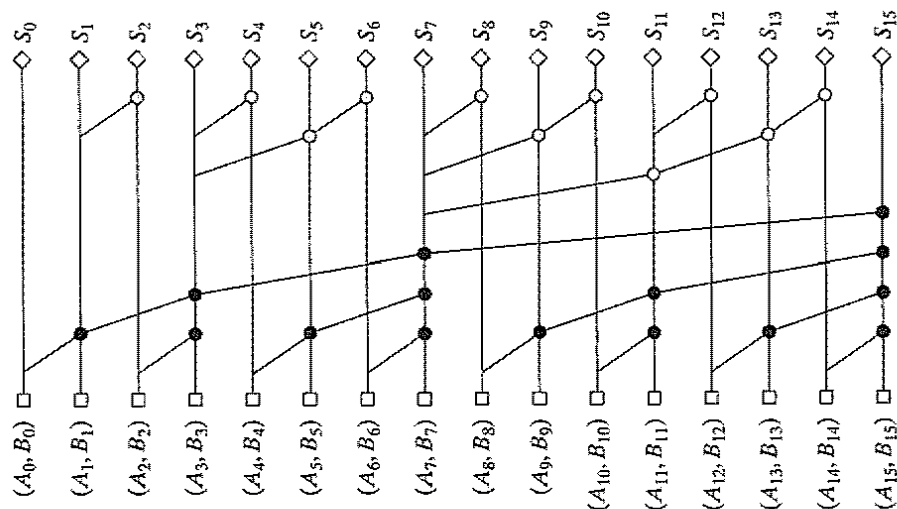


Figure 11-26 A 16-bit Brent–Kung tree. The reverse binary tree is colored gray.

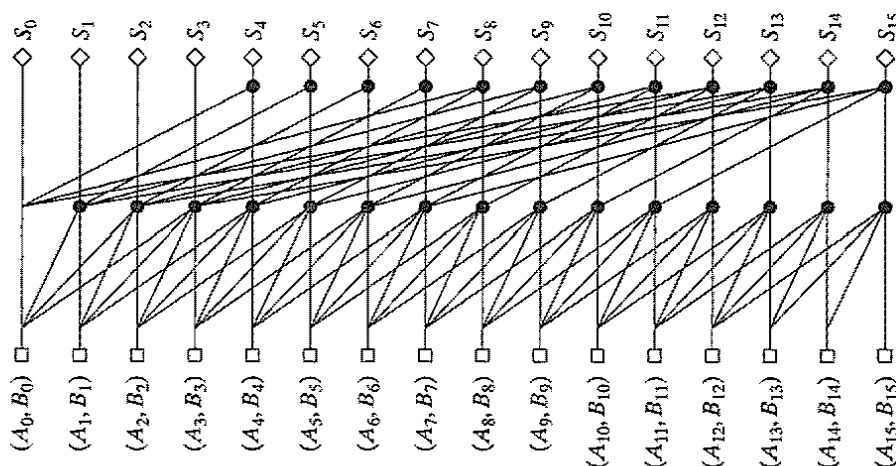


Figure 11-27 Radix-4 Kogge-Stone tree for 16-bit operands.

bits. The resulting structure, commonly called the *Brent-Kung adder*, uses 27 dot gates, or almost half of the 49 needed for a full radix-2 tree, and it needs fewer wires as well. The wiring structure is less regular, however, and fan-out varies from gate to gate, making performance optimization more difficult. Especially the fan-out of the middle node ( $C_{0,7}$ ), which equals one sum and five dot operations for this example, is of major concern. This observation makes the Brent-Kung adder rather unsuited for very large adders ( $> 32$  bits).

An option to reduce the depth of the tree is to combine four signals at a time at each level of the hierarchy. The resulting tree is now of class *radix-4*, because it uses building blocks of order 4 as shown in Figure 11-27. A 16-bit addition needs only two stages of carry logic. Be aware that each gate is more complex and that having less logic stages may not always result in faster operation (as we saw in Chapter 5).

#### Problem 11.5 Radix-4 Dot Operator in Dynamic Logic

Design the radix-4 dot operators in dynamic logic. Use the radix-2 circuits as reference. How is the sum select implemented for radix-4? Using the method of logical effort, compare the delays of radix-2 and radix-4 designs for 16-bit adders. Which one is faster?

On average, a lookahead adder is several times larger than a ripple adder, but has dramatic speed advantages for larger operands. The logarithmic behavior makes it preferable over bypass or select adders for larger values of  $N$ . The exact value of the cross point depends heavily on technology and circuit design factors.

The discussion of adders is by no means complete. Due to its impact on the performance of computational structures, the design of fast adder circuits has been the subject of many publications. It is even possible to construct adder structures with a propagation delay that is *independent of the number of bits*. Examples of those are the carry-save structures and the redundant

binary arithmetic structures [Swartzlander90]. These adders require number-encoding and -decoding steps, whose delay is a function of  $N$ . Therefore, they are only interesting when embedded in larger structures such as multipliers or high-speed signal processors.

## 11.4 The Multiplier

Multiplications are expensive and slow operations. The performance of many computational problems often is dominated by the speed at which a multiplication operation can be executed. This observation has, for instance, prompted the integration of complete multiplication units in state-of-the-art digital signal processors and microprocessors.

Multipliers are, in effect, complex adder arrays. Therefore, the majority of the topics discussed in the preceding section are of value in this context as well. The analysis of the multiplier gives us some further insight into how to optimize the performance (or the area) of complex circuit topologies. After a short discussion of the multiply operation, we discuss the basic array multiplier. We also discuss different approaches to partial product generation, accumulation and their final summation.

### 11.4.1 The Multiplier: Definitions

Consider two *unsigned* binary numbers  $X$  and  $Y$  that are  $M$  and  $N$  bits wide, respectively. To introduce the multiplication operation, it is useful to express  $X$  and  $Y$  in the binary representation

$$X = \sum_{i=0}^{M-1} X_i 2^i \quad Y = \sum_{j=0}^{N-1} Y_j 2^j \quad (11.21)$$

with  $X_i, Y_j \in \{0, 1\}$ . The multiplication operation is then defined as follows:

$$\begin{aligned} Z = X \times Y &= \sum_{k=0}^{M+N-1} Z_k 2^k \\ &= \left( \sum_{i=0}^{M-1} X_i 2^i \right) \left( \sum_{j=0}^{N-1} Y_j 2^j \right) = \sum_{i=0}^{M-1} \left( \sum_{j=0}^{N-1} X_i Y_j 2^{i+j} \right) \end{aligned} \quad (11.22)$$

The simplest way to perform a multiplication is to use a single two-input adder. For inputs that are  $M$  and  $N$  bits wide, the multiplication takes  $M$  cycles, using an  $N$ -bit adder. This *shift-and-add* algorithm for multiplication adds together  $M$  *partial products*. Each partial product is generated by multiplying the multiplicand with a bit of the multiplier—which, essentially, is an AND operation—and by shifting the result on the basis of the multiplier bit's position.

A faster way to implement multiplication is to resort to an approach similar to manually computing a multiplication. All the partial products are generated at the same time and organized in an array. A multioperand addition is applied to compute the final product. The approach is illustrated in Figure 11-28. This set of operations can be mapped directly into hardware. The

|   |                   |                  |
|---|-------------------|------------------|
|   | 1 0 1 0 1 0       | Multiplicand     |
| × | 1 0 1 1           | Multiplier       |
|   | 1 0 1 0 1 0       | Partial products |
|   | 1 0 1 0 1 0       |                  |
|   | 0 0 0 0 0 0       |                  |
|   | 1 0 1 0 1 0       |                  |
| + | 1 0 1 0 1 0       |                  |
|   | 1 1 1 0 0 1 1 1 0 | Result           |

Figure 11-28 Binary multiplication—an example.

resulting structure is called an *array multiplier* and combines the following three functions: *partial-product generation*, *partial-product accumulation*, and *final addition*.

#### Problem 11.6 Multiplication by a Constant

It is often necessary to implement multiplication of an operand by a constant. Describe how you would proceed with this task.

#### 11.4.2 Partial-Product Generation

Partial products result from the logical AND of multiplicand  $X$  with a multiplier bit  $Y_i$  (see Figure 11-29). Each row in the partial-product array is either a copy of the multiplicand or a row of zeroes. Careful optimization of the partial-product generation can lead to some substantial delay and area reductions. Note that in most cases the partial-product array has many zero rows that have no impact on the result and thus represent a waste of effort when added. In the case of a multiplier consisting of all ones, all the partial products exist, while in the case of all zeros, there is none. This observation allows us to reduce the number of generated partial products by half.

Assume, for example, an eight-bit multiplier of the form 01111110, which produces six nonzero partial-product rows. One can substantially reduce the number of nonzero rows by recoding this number ( $2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2$ ) into a different format. The reader can verify that the form  $100000\bar{1}0$ , with  $\bar{1}$  a shorthand notation for  $-1$ , represents the same number. Using

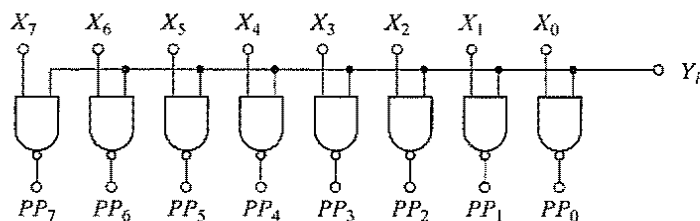


Figure 11-29 Partial-product generation logic.

this format, we have to add only two partial products, but the final adder has to be able to perform subtraction as well. This type of transformation is called *Booth's recoding* [Booth51], and it reduces the number of partial products to, at most, one half. It ensures that for every two consecutive bits, at most one bit will be 1 or -1. Reducing the number of partial products is equivalent to reducing the number of additions, which leads to a speedup as well as an area reduction. Formally, this transformation is equivalent to formatting the multiplier word into a base-4 scheme, instead of the usual binary format:

$$Y = \sum_{j=0}^{(N-1)/2} Y_j 4^j \text{ with } (Y_j \in \{-2, -1, 0, 1, 2\}) \quad (11.23)$$

Note that 1010...10 represents the worst case multiplier input because it generates the most partial products (one half). While the multiplication with  $\{0, 1\}$  is equivalent to an AND operation, multiplying with  $\{-2, -1, 0, 1, 2\}$  requires a combination of inversion and shift logic. The encoding can be performed on the fly and requires some simple logic gates.

Having a variable-size partial-product array is not practical for multiplier design, and a *modified Booth's recoding* is most often used instead [MacSorley61]. The multiplier is partitioned into three-bit groups that overlap by one bit. Each group of three is recoded, as shown in Table 11-2, and forms one partial product. The resulting number of partial products equals half of the multiplier width. The input bits to the recoding process are the two current bits, combined with the upper bit from the next group, moving from *msb* to *lsb*.

**Table 11-2** Modified Booth's recoding.

| Partial Product Selection Table |                    |
|---------------------------------|--------------------|
| Multiplier Bits                 | Recoded Bits       |
| 000                             | 0                  |
| 001                             | + Multiplicand     |
| 010                             | + Multiplicand     |
| 011                             | + 2 × Multiplicand |
| 100                             | - 2 × Multiplicand |
| 101                             | - Multiplicand     |
| 110                             | - Multiplicand     |
| 111                             | 0                  |

In simple terms, the modified Booth's recoding essentially examines the multiplier for strings of ones from *msb* to *lsb* and replaces them with a leading 1, and a  $-1$  at the end of the string. For example, 011 is understood as the beginning of a string of ones and is therefore replaced by a leading 1 (or 100), while 110 is seen as the end of a string and is replaced by a  $-1$  at the least significant position (or 0 $\bar{1}$ 0).

#### Example 11.5 Modified Booth's Recoding

Consider the eight-bit binary number 01111110 shown earlier. This can be divided into four overlapping groups of three bits, going from *msb* to *lsb*: 00 (1), 11 (1), 11 (1), 10 (0). Recoding by using Table 11-2 yields: 10 (2 $\times$ ), 00 (0 $\times$ ), 00 (0 $\times$ ),  $\bar{1}$ 0 ( $-2\times$ ), or, in combined format, 1000000 $\bar{1}$ 0. This is equivalent to the result we obtained before.

#### Problem 11.7 Booth's Recoder

Design the combinational logic that implements a modified Booth's recoding for a parallel multiplier, using Table 11-2. Compare its implementation in complementary and pass-transistor CMOS.

#### 11.4.3 Partial-Product Accumulation

After the partial products are generated, they must be summed. This accumulation is essentially a multioperand addition. A straightforward way to accumulate partial products is by using a number of adders that will form an array—hence, the name, *array multiplier*. A more sophisticated procedure performs the addition in a tree format.

#### The Array Multiplier

The composition of an array multiplier is shown in Figure 11-30. There is a one-to-one topological correspondence between this hardware structure and the manual multiplication shown in Figure 11-28. The generation of  $N$  partial products requires  $N \times M$  two-bit AND gates (in the style of Figure 11-29).<sup>5</sup> Most of the area of the multiplier is devoted to the adding of the  $N$  partial products, which requires  $N - 1$   $M$ -bit adders. The shifting of the partial products for their proper alignment is performed by simple routing and does not require any logic. The overall structure can easily be compacted into a rectangle, resulting in a very efficient layout.

Due to the array organization, determining the propagation delay of this circuit is not straightforward. Consider the implementation of Figure 11-30. The partial sum adders are implemented as ripple-carry structures. Performance optimization requires that the critical timing path be identified first. This turns out to be nontrivial. In fact, a large number of paths of almost identical length can be identified. Two of those are highlighted in Figure 11-31. A closer

<sup>5</sup>This particular implementation does not employ Booth's recoding. Adding recoding does not substantially change the implementation. The number of adders is reduced by half, and the partial-product generation is slightly more complex.

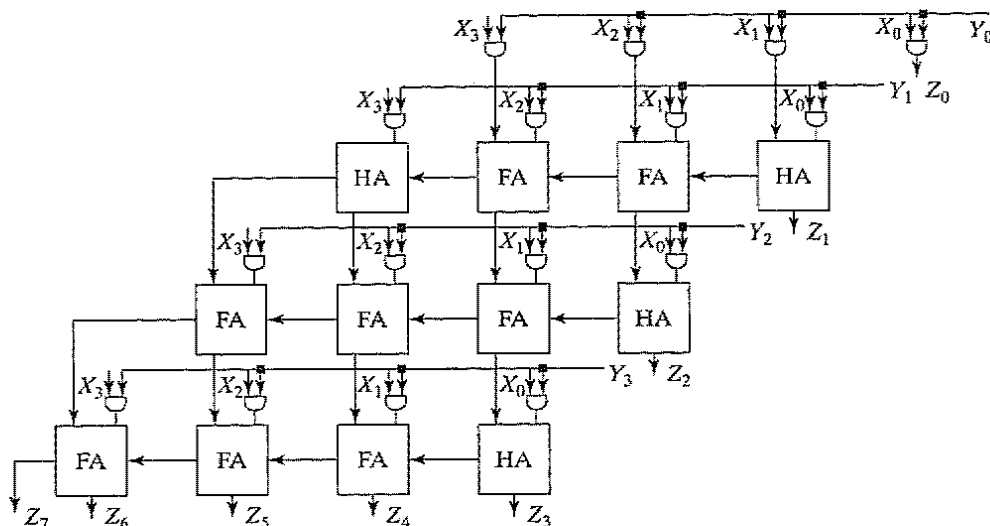


**Figure 11-30** A  $4 \times 4$  bit-array multiplier for unsigned numbers—composition. HA stands for a half adder, or an adder cell with only two inputs. The hardware for the generation and addition of one partial product is shaded in gray.

look at those critical paths yields an approximate expression for the propagation delay (derived here for critical path 2). We write this as

$$t_{mult} \approx [(M-1) + (N-2)]t_{carry} + (N-1)t_{sum} + t_{and} \quad (11.24)$$

where  $t_{carry}$  is the propagation delay between input and output carry,  $t_{sum}$  is the delay between the input carry and sum bit of the full adder, and  $t_{and}$  is the delay of the AND gate.

Since all critical paths have the same length, speeding up just one of them—for instance, by replacing one adder by a faster one such as a carry-select adder—does not make much sense from a design standpoint. All critical paths have to be attacked at the same time. From Eq. (11.24), it can be deduced that the minimization of  $t_{mult}$  requires the minimization of both  $t_{carry}$

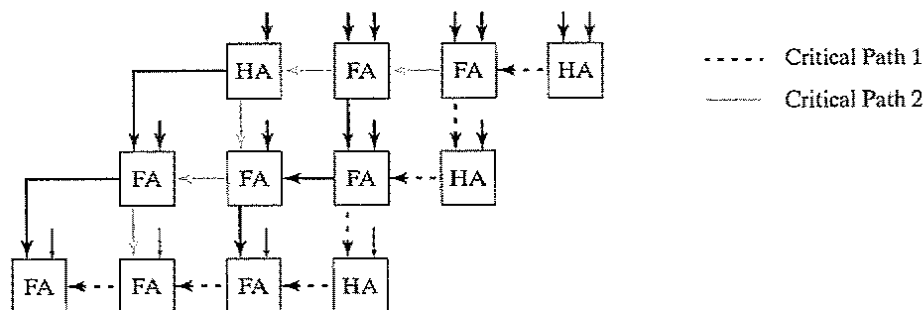


**Figure 11-31** Ripple-carry based  $4 \times 4$  multiplier (simplified diagram). Two of the possible critical paths are highlighted.

and  $t_{sum}$ . In this case, it could be beneficial for  $t_{carry}$  to equal  $t_{sum}$ . This contrasts with the requirements for adder cells discussed before, where a minimal  $t_{carry}$  was of prime importance. An example of a full-adder circuit with comparable  $t_{sum}$  and  $t_{carry}$  delays was shown in Figure 11-7.

### Problem 11.8 Signed-Binary Multiplier

The multiplier presented in Figure 11-30 only handles unsigned numbers. Adjust the structure so that two's-complement numbers are also accepted.

### Carry-Save Multiplier

Due to the large number of almost identical critical paths, increasing the performance of the structure of Figure 11-31 through transistor sizing yields marginal benefits. A more efficient realization can be obtained by noticing that the multiplication result does not change when the output carry bits are passed diagonally downwards instead of only to the right, as shown in Figure 11-32. We include an extra adder called a *vector-merging* adder to generate the final result. The resulting multiplier is called a *carry-save multiplier* [Wallace64], because the carry bits are not immediately added, but rather are “saved” for the next adder stage. In the final stage, carries and sums are merged in a fast carry-propagate (e.g., carry-lookahead) adder stage. While this structure has a slightly increased area cost (one extra adder), it has the advantage that its worst case critical path is shorter and uniquely defined, as highlighted in Figure 11-32 and is expressed as

$$t_{mult} = t_{and} + (N - 1)t_{carry} + t_{merge} \quad (11.25)$$

still assuming that  $t_{add} = t_{carry}$ .

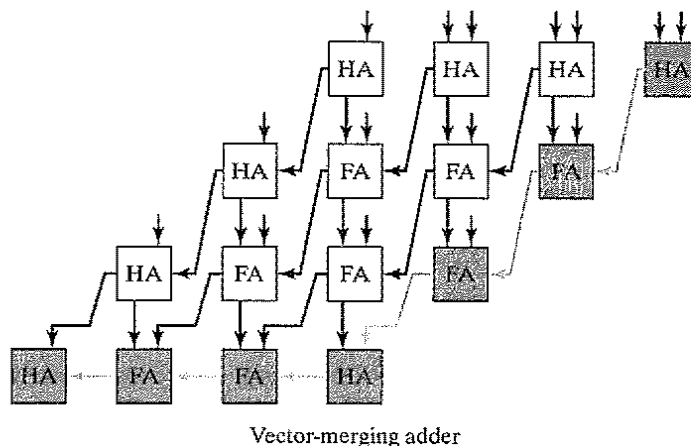


**Figure 11-32** A  $4 \times 4$  carry-save multiplier. The critical path is highlighted in gray.

### Example 11.6 Carry-Save Multiplier

When mapping the carry-save multiplier of Figure 11-32 onto silicon, one has to take into account some other topological considerations. To ease the integration of the multiplier into the rest of the chip, it is advisable to make the outline of the module approximately rectangular. A floor plan for the carry-save multiplier that achieves this goal is shown in Figure 11-33. Observe the regularity of the topology. This makes the generation of the structure amenable to automation.

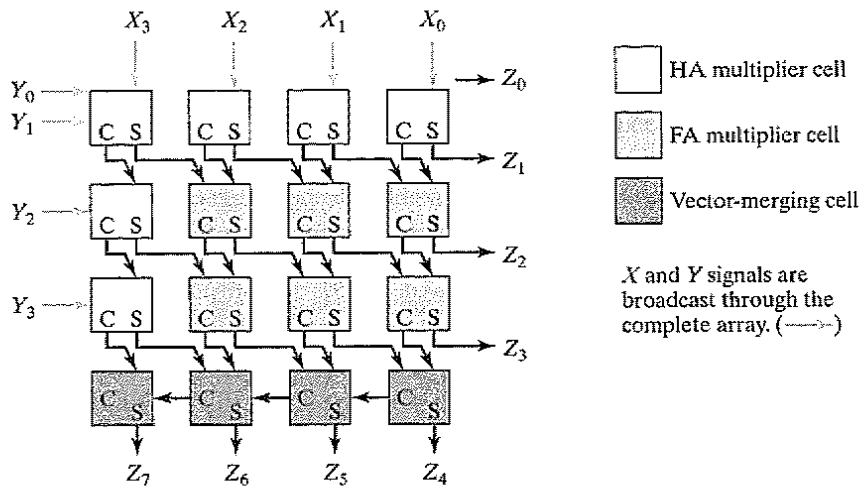


**Figure 11-33** Rectangular floorplan of carry-save multiplier. Different cells are differentiated by shades of gray.  $X$  and  $Y$  signals are AND'ed before being added. The leftmost column of cells is redundant and can be eliminated.

### The Tree Multiplier

The partial-sum adders can also be rearranged in a treelike fashion, reducing both the critical path and the number of adder cells needed. Consider the simple example of four partial products each of which is four bits wide, as shown in Figure 11-34a. The number of full adders needed for this operation can be reduced by observing that only column 3 in the array has to add four bits. All other columns are somewhat less complex. This is illustrated in Figure 11-34b, where the original matrix of partial products is reorganized into a tree shape to visually illustrate its varying depth. The challenge is to realize the complete matrix with a minimum depth and a minimum number of adder elements. The first type of operator that can be used to cover the array is a full adder, which takes three inputs and produces two outputs: the sum, located in the same column and the carry, located in the next one. For this reason, the FA is called a *3-2 compressor*. It is denoted by a circle covering three bits. The other operator is the half-adder, which takes two input bits in a column and produces two outputs. The HA is denoted by a circle covering two bits.

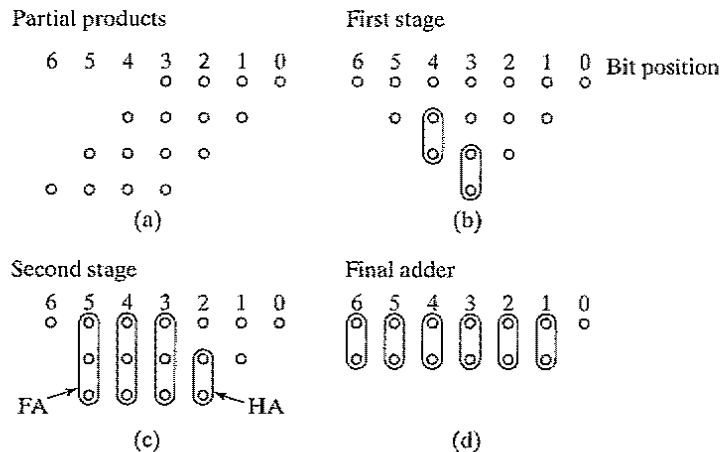


**Figure 11-34** Transforming a partial-product tree (a) into a Wallace tree (b,c,d), using an iterative covering process. The example shown is for a four-bit operand.

To arrive at the minimal implementation, we iteratively cover the tree with FAs and HAs, starting from its densest part. In a first step, we introduce HAs in columns 4 and 3 (Figure 11-34b). The reduced tree is shown in Figure 11-34c. A second round of reductions creates a tree of depth 2 (Figure 11-34d). Only three FAs and three HAs are used for the reduction process, compared with six FAs and six HAs in the carry-save multiplier of Figure 11-32! The final stage consists of a simple two-input adder, for which any type of adder can be used (as discussed in the next section, “Final Addition”).

The presented structure is called the *Wallace tree multiplier* [Wallace64], and its implementation is shown in Figure 11-35. The tree multiplier realizes substantial hardware savings for larger multipliers. The propagation delay is reduced as well. In fact, it can be shown that the propagation delay through the tree is equal to  $O(\log_{3/2}(N))$ . While substantially faster than the carry-save structure for large multiplier word lengths, the Wallace multiplier has the disadvantage of being very irregular, which complicates the task of coming up with an efficient layout. This irregularity is visible even in the four-bit implementation of Figure 11-35.

There are numerous other ways to accumulate the partial-product tree. A number of compression circuits has been proposed in the literature, a detailed discussion of which is beyond the scope of this book. They are all based on the concept that when full adders are used as 3:2 compressors, the number of partial products is reduced by two-thirds per multiplier stage. One can even go a step further and devise a 4–2 (or higher order) compressor, such as in [Weinberger81, Santoro89]. In fact, many of today’s high-performance multipliers do just that.

#### 11.4.4 Final Addition

The final step for completing the multiplication is to combine the result in the final adder. Performance of this “vector-merging” operation is of key importance. The choice of the adder style depends on the structure of the accumulation array. A carry-lookahead adder is the preferable

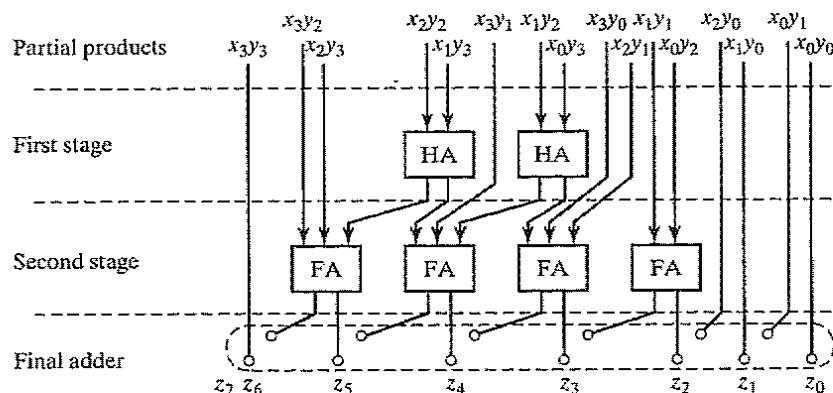


**Figure 11-35** Wallace tree for four-bit multiplier.

option if all input bits to the adder arrive at the same time, as it yields the smallest possible delay. This is the case if a pipeline stage is placed right before the final addition. Pipelining is a technique frequently used in high-performance multipliers. In nonpipelined multipliers, the arrival-time profile of the inputs to the final adder is quite uneven due to the varying logic depths of the multiplier tree. Under these circumstances, other adder topologies, such as carry select, often yield performance numbers similar to lookahead at a substantially reduced hardware cost [Oklobdzija01].

### 11.4.5 Multiplier Summary

All the presented techniques can be combined to yield multipliers with extremely high performance. For instance, a  $54 \times 54$  multiplier can achieve a propagation delay of only 4.4 ns in a 0.25- $\mu\text{m}$  CMOS technology by combining Booth encoding with a Wallace tree by using 4–2 compression in pass-transistor implementation and by using a mixed carry-select, carry-lookahead topology for the final adder [Ohkubo95]. More information on these multipliers (and others) can be found in the references [e.g., Swartzlander90, Oklobdzija01].

## 11.5 The Shifter

The shift operation is another essential arithmetic operation that requires adequate hardware support. It is used extensively in floating-point units, scalars, and multiplications by constant numbers. The latter can be implemented as a combination of add and shift operations. Shifting a data word left or right over a constant amount is a trivial hardware operation and is implemented by the appropriate signal wiring. A programmable shifter, on the other hand, is more complex and requires active circuitry. In essence, such a shifter is nothing less than an intricate multiplexer circuit. A simple one-bit left–right shifter is shown in Figure 11-36. Depending on the control signals, the input word is either shifted left or right, or else it remains unchanged. Multi-bit shifters can be built by cascading a number of these units. This approach rapidly becomes complex, unwieldy, and ultimately too slow for larger shift values. Therefore, a more structured

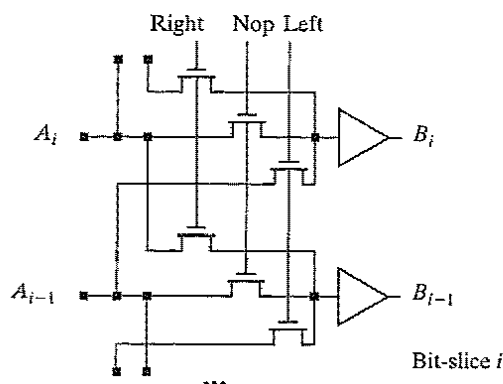


**Figure 11-36** One-bit (left-right) programmable shifter. The data passes undisturbed under the nop condition.

approach is advisable. Next, we discuss two commonly used shift structures, the *barrel shifter* and the *logarithmic shifter*.

### 11.5.1 Barrel Shifter

The structure of a barrel shifter is shown in Figure 11-37. It consists of an array of transistors, in which the number of rows equals the word length of the data, and the number of columns equals the maximum shift width. In this case, both are set equal to four. The control wires are routed diagonally through the array. A major advantage of this shifter is that the signal has to pass through at most one transmission gate. In other words, the propagation delay is theoretically constant and independent of the shift value or shifter size. This is not true in reality, however, because the capacitance at the input of the buffers rises linearly with the maximum shift width.

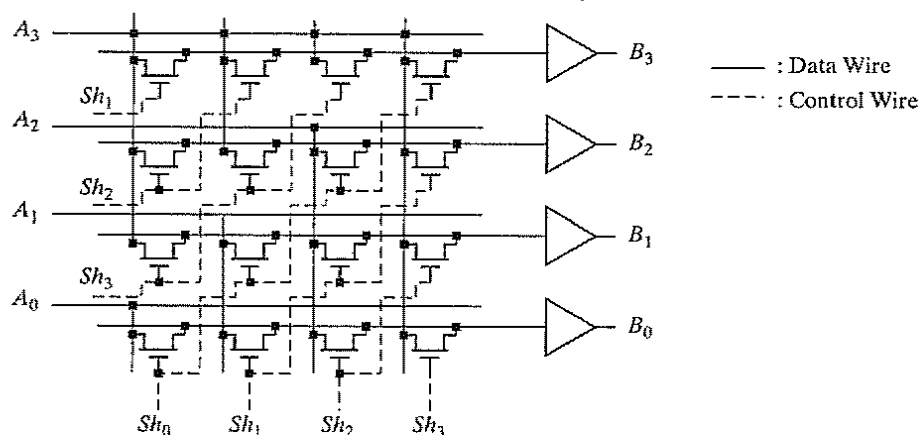


**Figure 11-37** Barrel shifter with a programmable shift width from zero to three bits to the right. The structure supports automatic repetition of the sign bit ( $A_3$ ), also called *sign-bit extension*.

An important property of this circuit is that the layout size is not dominated by the active transistors as in the case of all other arithmetic circuits, but by the number of wires running through the cell. More specifically, the size of the cell is bounded by the pitch of the metal wires!

Another important consideration when selecting a shifter is the format in which the shift value must be presented. From the schematic diagram of Figure 11-37, we see that the barrel shifter needs a control wire for every shift bit. For example, a four-bit shifter needs four control signals. To shift over three bits, the signals  $Sh_3:Sh_0$  take on the value 1000. Only one of the signals is high. In a processor, the required shift value normally comes in an encoded binary format, which is substantially more compact. For instance, the encoded control word needs only two control signals and is represented as 11 for a shift over three bits. To translate the latter representation into the former (with only one bit high), an extra module called a *decoder* is required. (Decoders are treated in detail in Chapter 12.)

---

**Problem 11.9 Two's Complement Shifter**

Explain why the shifter shown in Figure 11-37 implements a two's complement shift.

---

### 11.5.2 Logarithmic Shifter

While the barrel shifter implements the whole shifter as a single array of pass transistors, the logarithmic shifter uses a staged approach. The total shift value is decomposed into shifts over powers of two. A shifter with a maximum shift width of  $M$  consists of a  $\log_2 M$  stages, where the  $i$ th stage either shifts over  $2^i$  or passes the data unchanged. An example of a shifter with a maximum shift value of seven bits is shown in Figure 11-38. For instance, to shift over five bits, the first stage is set to shift mode, the second to pass mode, and the last stage again to shift. Notice that the control word for this shifter is already encoded, and no separate decoder is required.

The speed of the logarithmic shifter depends on the shift width in a logarithmic way, since an  $M$ -bit shifter requires  $\log_2 M$  stages. Furthermore, the series connection of pass transistors slows the shifter down for larger shift values. A careful introduction of intermediate buffers is therefore necessary, as discussed in Chapter 6.

In general, we conclude that a barrel shifter is appropriate for smaller shifters. For larger shift values, the logarithmic shifter becomes more effective, in terms of both area and speed. Furthermore, the logarithmic shifter is easily parameterized, allowing for automatic generation. The most important concept of this section is that the exploitation of regularity in an arithmetic operator can lead to dense and high-speed circuit implementations.

## 11.6 Other Arithmetic Operators

In the previous sections, we only discussed a subset of the large number of arithmetic circuits required in the design of microprocessors and signal processors. Besides adders, multipliers, and shifters, others operators such as comparators, dividers, counters, and goniometric operators (sine, cosine, tangent) are often needed. A full analysis of these circuits is beyond the scope of

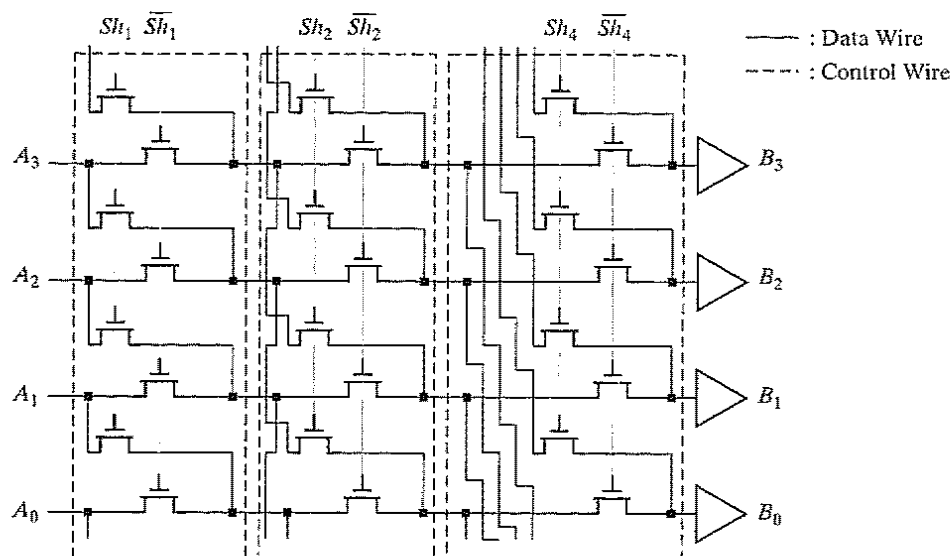


**Figure 11-38** Logarithmic shifter with maximal shift width of seven bits to the right. (Only the four least significant bits are shown.)

this book. We refer the interested reader to some of the excellent references available on the topic (e.g., [Swartzlander90], [Oklobdzija01]).

The reader should be aware that most of the design ideas introduced in this chapter apply to these other operators as well. For instance, comparators can be devised with a linear, square root, and logarithmic dependence on the number of bits. In fact, some operators are simple derivatives of the adder or multiplier structures presented earlier. For example, Figure 11-39 shows how a two's complement subtractor can be realized by combining a two's complement adder with an extra inversion stage, or how a subtractor can be used to implement  $A \geq B$ .

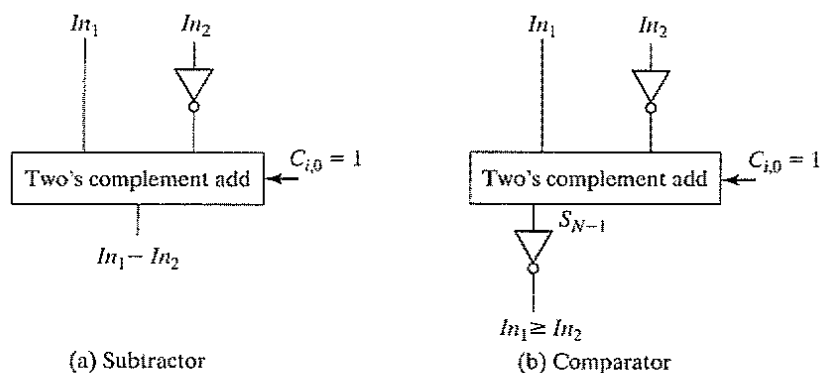


**Figure 11-39** Arithmetic structures derived from a full adder.

**Problem 11.10 Comparator**

Derive a logic diagram for a comparator that implements the following logic functions:  $\geq$ ,  $=$ , and  $\leq$ .

**Case Study—Design of an Arithmetic-Logic Unit (ALU)**

The core of any microprocessor or microcontroller is the arithmetic-logic unit (or ALU). The ALU combines addition and subtraction with other operations, such as shifting and bitwise logic operations (AND, OR, and XOR).

An ALU taken from a 64-bit high-end microprocessor is shown in Figure 11-40 [Mathew01.] It consists of two levels of wide multiplexers, a 64-bit adder, a logic unit, an operator-merging multiplexer and the write-back bus driver. The first pair of 9:1 multiplexers selects the from nine different input sources, three of which are from register files and caches. The other six are bypass paths that come directly from the six ALUs—the microprocessor in question can issue six integer instructions in one cycle. One of the bypass loops actually feeds the output of the ALU back to its own input, creating a single cycle loop that defines the critical path. The second 5:1 multiplexer on the  $A$  input performs a partial shifting of the operand, while the 2:1 multiplexer on the  $B$  input simply inverts the operand to implement subtraction. The adder executes two's complement addition or subtraction on the operands  $A$  and  $B$ . Two's complement subtraction is performed by inverting all the bits of the operand  $B$  (using the 2:1 MUX) and by setting the carry-in to one. The sum-selection block not only implements the sum selection, but also merges the outputs of arithmetic and logic units. Finally, a strong buffer drives the loop-back bus, which presents a large load to the ALU. For example, for a six-issue Intel/HP Itanium processor [Fetzer02], the bus has to connect to all six units, translating into more than 2 mm of wire length in a 0.18- $\mu\text{m}$  technology. When added to the load from the register file, it presents over 0.5 pF of capacitive load.

An ALU represents a typical example of a bit-sliced design. Each slice in each block is pitch matched, which minimizes the vertical routing. A floor plan of the 64-bit ALU is shown in Figure 11-41. Intercell routing is done horizontally in metal-3, except for the long horizontal wires between adder stages 1 and 2 and 2 and 3, which are laid out in a combination of metal-3 and metal-4. For example, if the adder is implemented as a radix-4 CLA, the longest wire after the second PG stage crosses 48 cells.

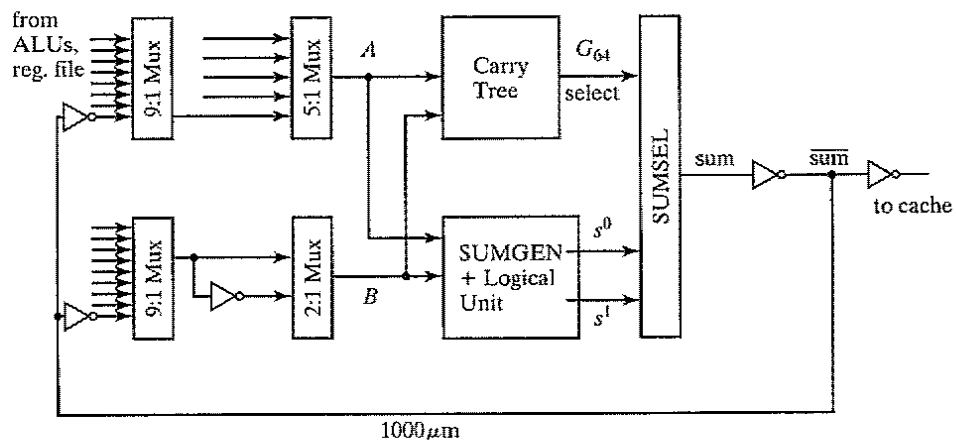


**Figure 11-40** A 64-bit arithmetic-logic unit.

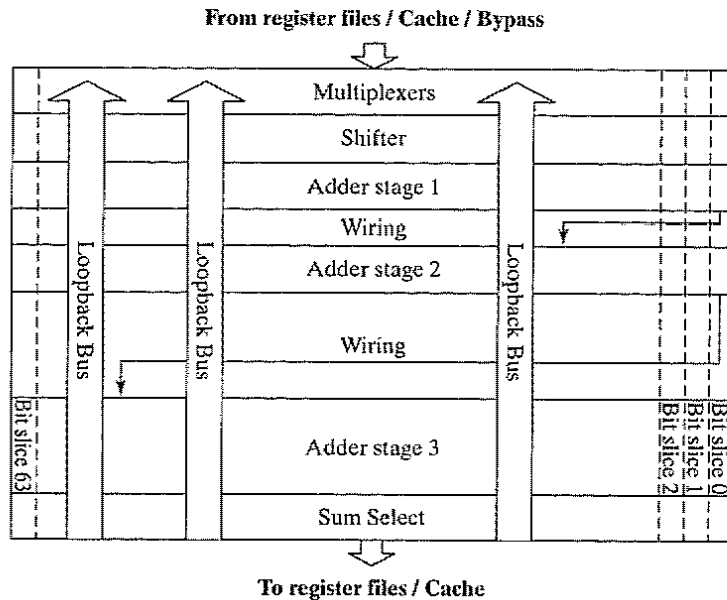


Figure 11-41 Floor plan of a 64-bit ALU.

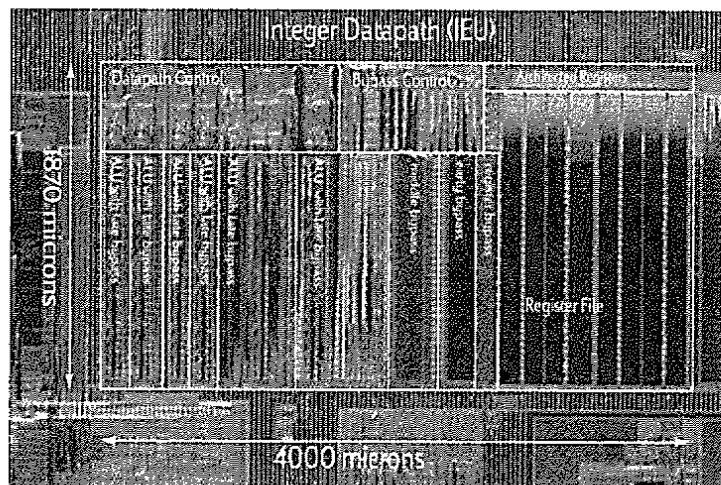


Figure 11-42 Microphotograph of the Itanium processor, showing bit-sliced design of six ALUs together with bypasses, register file, and control logic. (Courtesy of Intel.)

The loop-back wiring is routed in the top-level metal (e.g., metal-5). In case of a multiple-issue microprocessor, these busses span across all ALUs, as can be seen in Figure 11-42. The number of loop-back busses frequently determines the bit-pitch in multiissue ALUs. In our example, nine 64-bit-wide busses have to cross back over the ALU. To avoid needing to add extra wiring space, the bit width is set to nine

bus-wire pitches. Since these wires are long, they are designed with double width and at least double spacing to reduce their resistance and capacitance. ■

### 11.7 Power and Speed Trade-Offs in Datapath Structures\*

In the preceding discussion on adders, multipliers, and shifters, we mainly explored the trade-off between speed and area and ignored power considerations. In this section, we briefly analyze the third dimension of the design exploration space. Since most of the approaches to minimize power were already introduced in Chapter 6, the discussion that follows serves mostly as an illustration of the concepts advanced there.

Typical digital designs are either *latency* or *throughput* constrained. A latency-constrained design has to finish computation by a given deadline. Interactive communication and gaming are examples of such. Throughput-constrained designs, on the other hand, must maintain a required data throughput. A 1000BaseT Gigabit Ethernet connection has to maintain a constant throughput of 1 Gigabit/s. The architectural optimization techniques that are available to the designer differ based on the design constraints. Pipelining or parallelization, for instance, works effectively for the throughput-constrained scenario, but it may not be applicable to the latency-constrained case. For example, a throughput of 1 Gb/s over copper wires is achieved in Gigabit Ethernet by processing four 250 Mb/s streams of data in parallel.

With a fixed architecture of the datapath, speed, area, and power can be traded off through the choice of the supply voltage(s), transistor thresholds, and device sizes. This opens the door for a large variety of power minimization techniques, which are summarized in Table 11-3. They are classified as follows:

- **Enable Time**—Some design techniques are implemented (or enabled) at *design time*. Transistor widths and lengths, for instance, are fixed at the time of design. Supply voltage and transistor thresholds, on the other hand, can be either assigned statically during the design phase, or changed dynamically at *run time*. Other techniques primarily address the time that a function or module in a digital design is in idle mode (or standby). It is only logical to require that the power dissipation of a module in *sleep mode* should be absolutely minimal.

**Table 11-3** Power minimization techniques.

|         | Constant Throughput/Latency                                                    |                                         | Variable Throughput/Latency              |
|---------|--------------------------------------------------------------------------------|-----------------------------------------|------------------------------------------|
|         | Design Time                                                                    | Sleep Mode                              | Run Time                                 |
| Active  | Lower $V_{DD}$ , Multi- $V_{DD}$ ,<br>Transistor Sizing<br>Logic Optimizations | Clock gating                            | Dynamic voltage scaling                  |
| Leakage | Multi- $V_{th}$ +<br>Active Techniques                                         | Sleep transistors,<br>Variable $V_{th}$ | Variable $V_{th}$ +<br>Active Techniques |

- **Targeted Dissipation Source**—Another classification of the power-management techniques concerns the source of power dissipation they address: active (dynamic) power or leakage (static) power. Lowering the supply voltage, for example, is a very attractive technique: It not only reduces the energy consumed per transition in a quadratic way—albeit at the expense of performance—but also reduces the leakage current. On the other hand, increasing the threshold voltage mainly impacts the leakage component.

The sleep mode of operation deserves some special attention. If a digital block still receives a clock while in idle mode, its clock distribution network, together with the attached flip-flops, continues to consume energy, even while no computation is performed. Recall that, typically, one third of total energy of a digital system is consumed in clock distribution network. A common method to reduce power in idle mode is the *clock gating* technique introduced in Chapter 10. In this approach, the main clock connection to a module is turned off (or *gated*) whenever the block is idle. However, clock gating does not reduce the leakage power of the idle block. More complicated schemes to lower the standby power have to be used, such as increasing the transistor thresholds or switching off the power rails.

In the following sections, we discuss each of these design-time and run-time techniques in detail.

### 11.7.1 Design-Time Power-Reduction Techniques

#### Reducing the Supply Voltage

A reduction in supply voltage results in quadratic power savings and thus is the most attractive approach. On the negative side, the delay of CMOS gates increases inversely with supply voltage. At the datapath level, this loss of performance can be compensated for by other means, such as logical and architectural optimizations. For example, a ripple-carry adder can be replaced by a faster structure, such as a lookahead adder. The latter implementation is larger and more complex, which translates into a larger physical and switching capacitance. This is more than offset by the fact that the faster adder can run at a lower supply voltage for the same performance.

The trade-off between ripple and lookahead operates at the logical level. Similarly, and even more effectively, architectural optimizations can be employed to compensate for the effect of a reduced  $V_{DD}$ , as illustrated in Example 11.7.

---

#### Example 11.7 Minimizing the Power Consumption by Using Parallelism

To illustrate how architectural techniques can be used to compensate for reduced speed, a simple eight-bit datapath consisting of an adder and a comparator is analyzed, assuming a 2- $\mu\text{m}$  CMOS technology [Chandrakasan92]. As shown in Figure 11-43, inputs  $A$  and  $B$  are added, and the result is compared with input  $C$ . Assume that the worst case delay through the adder, comparator, and latch is approximately 25 ns at a supply voltage of 5 V. At best, the system can be clocked with a clock period of  $T = 25$  ns. When required to run at this maximum possible throughput, it is clear that the operating

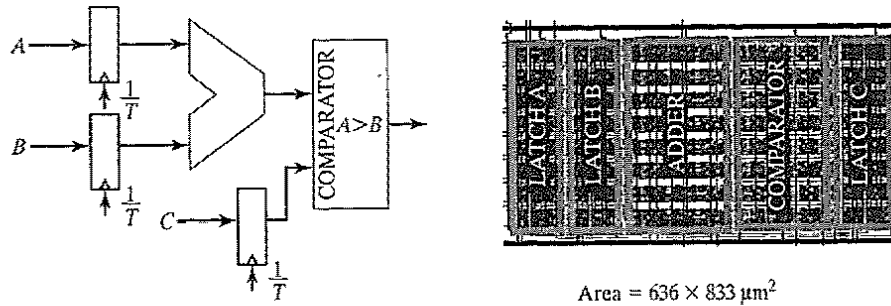


Figure 11-43 A simple datapath with corresponding layout.

voltage cannot be reduced any further because no extra delay can be tolerated. We use this topology as the reference datapath for our study and present power improvement numbers with respect to this reference. The average power consumed by the reference datapath is given by

$$P_{ref} = C_{ref} V_{ref}^2 f_{ref} \quad (11.26)$$

where  $C_{ref}$  is the total effective capacitance being switched per clock cycle. The effective capacitance can be determined by averaging the energy over a sequence of random input patterns with a uniform distribution.

One way to maintain throughput while reducing the supply voltage is to utilize a parallel architecture. As shown in Figure 11-44, two identical adder-comparator datapaths connect in parallel, allowing each unit to work at half the original rate while maintaining the original throughput. Since the speed requirements for the adder, comparator, and latch have decreased from 25 ns to 50 ns, the voltage can be dropped from 5 V to 2.9 V—the voltage that doubles the delay. While the datapath capacitance has increased by a factor of two, the operating frequency has correspondingly decreased by a factor of two. Unfortunately, there also is a slight increase in the total effective capacitance due to the extra routing and data multiplexing. This results in an increased capacitance by a factor of 2.15. The power for the parallel datapath is thus given by

$$P_{par} = C_{par} V_{par}^2 f_{par} = (2.15 C_{ref}) (0.58 V_{ref})^2 \left( \frac{f_{ref}}{2} \right) \approx 0.36 P_{ref} \quad (11.27)$$

The approach presented *trades off area for power*, as the resulting area is approximately 3.4 times larger than the original design. This technique is only applicable when the design is not area constrained. Furthermore, parallelism introduces extra routing overhead, which might cause additional dissipation. Careful optimization is needed to minimize this overhead.

Parallelism is not the only way to compensate for the loss in performance. Other architectural approaches, such as the use of pipelining, can accomplish the same goal. The most impor-

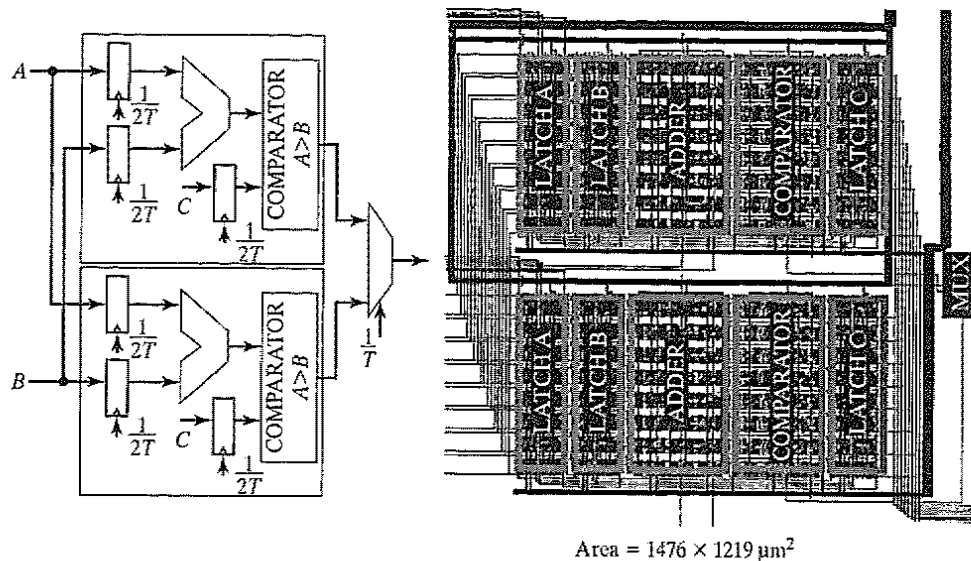


**Figure 11-44** Parallel implementation of the simple datapath.

tant message in the preceding analysis is that if power dissipation is the prime concern, dropping the supply voltage is the most effective means to achieve that goal. The subsequent loss in performance can be compensated for, if necessary, by an increase in area. Within certain bounds, this is acceptable, since area is no longer the compelling issue it used to be due to the dramatic increase in integration levels of the last decade.

#### Problem 11.11 Reducing the Supply Voltage, Using Pipelining

A pipeline stage is introduced between the adder and the comparator of the reference datapath of Figure 11-44. You may assume that this roughly halves the propagation delay of the logic, while it increases the capacitance by 15%. Obviously, an extra pipeline register is needed on input C as well. Determine how much power can be saved by this approach, given that the throughput has to remain constant compared with the reference datapath. Comment on the area overhead.

#### Using Multiple Supply Voltages

Reducing the supply voltage is fairly straightforward, but it may not be optimal. Reduced supply evenly lowers the power dissipation of all logic gates, while evenly increasing their delay. A better approach is to selectively decrease the supply voltage on some of the gates:

- those which correspond to fast paths and finish the computation early, and
- those with gates that drive large capacitances, have the largest benefit for the same delay increment.

This approach, however requires the use of more than one supply voltage. Multiple supply voltages are already employed frequently in today's ICs. A separate supply voltage is provided for the I/O for compatibility reasons, where the I/O ring is designed with transistors with thicker gate oxides to sustain higher voltages. The logic core is powered from lower voltage supplies and uses transistors with thinner oxides. This approach can be extended to lower the power dissipation of a circuit. For instance, every module could select the most appropriate voltage from two (or more) supply options. Even more extreme, multiple voltages can be assigned on a gate-by-gate basis.

**Module-Level Voltage Selection** Consider, for instance, the digital system shown in Figure 11-45, which consists of a datapath block with a critical path of 10 ns and a control block with a much shorter critical path of 4 ns, operating from the same supply voltage of 2.5 V. Also, assume that the datapath block has a fixed latency and throughput and that no architectural transformation can be applied to lower its supply. Since the control block finishes early (in other words, it has *timing slack*), its supply voltage can be lowered. A reduction to  $V_{DDL} = 1$  V increases its critical path delay to 10 ns, and lowers its power dissipation by more than five times.<sup>6</sup> We effectively exploit the discrepancy in the critical-path length of the various modules (called the *slack*) to selectively lower the power consumption of the modules with the larger slack.

When combining multiple supply voltages on a die, *level converters* are required whenever a module at the lower supply has to drive a gate at the higher voltage. If a gate supplied by  $V_{DDL}$  drives a gate at  $V_{DDH}$ , the PMOS transistor never turns off, resulting in static current and reduced output swing as illustrates in Figure 11-46. A level conversion performed at the boundaries of supply voltage domains prevents these problems. An asynchronous level converter, based on the DVSL template (Chapter 6) and similar to the low-swing signalling gate of Figure 9-32, is shown in Figure 11-46. The cross-coupled PMOS transistors perform the level conversion by using positive feedback. The delay of this level converter is quite sensitive to transistor-sizing and supply-voltage issues. The NMOS transistors operate with a reduced overdrive,  $V_{DDL} - V_{Th}$ , compared with the PMOS devices. They have to be made large to be able to overpower the positive feedback. For a low value of  $V_{DDL}$ , the delay can become very long. Due to

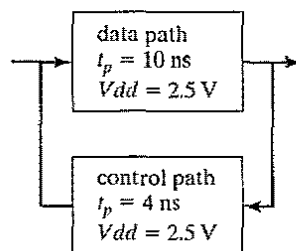
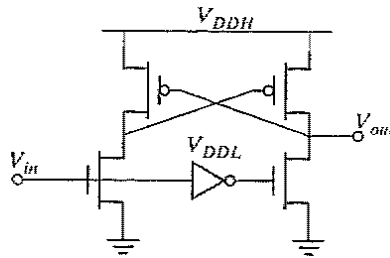


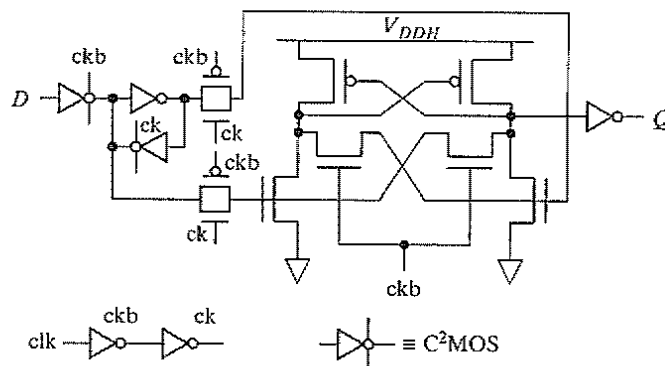
**Figure 11-45** Design with diverging critical-path lengths.

<sup>6</sup>This uses the simplifying assumption that the propagation delay is inversely proportional to the supply voltage.



the reduced overdrive, the circuit is also very sensitive to variations in the supply voltage. Note that level converters are not needed for a step-down change in voltage.

**Multiple Supplies Inside a Block** The same approach can be applied at much smaller granularity by individually setting the supply voltage for each cell inside a block [Usami95, Hamada01]. Examining the histogram of the critical-path delays for a typical digital block reveals that only a few paths are critical or near critical and that many paths have much shorter delays. The shorter paths essentially waste energy, as there is no reward for finishing early. For each of these paths, the supply voltage could be lowered to the optimum level. Minimum energy consumption would be achieved if all paths become critical. However, this is not easily achievable, as many logic gates are shared between different paths, and it is impractical to generate and



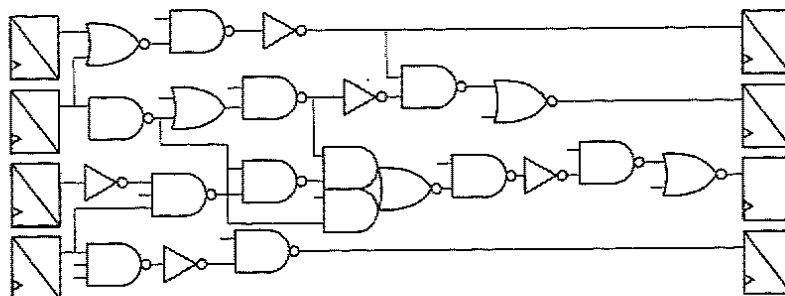


**Figure 11-48** Dual-supply design using clustered voltage scaling. Each path starts with a high supply and switches to a low supply (gray logic gates) if there is a delay slack. Level conversion is performed in the registers.

distribute a wide range of supply voltages. A more practical implementation, employing only two supplies, is shown in Figure 11-48. Using a *clustered voltage-scaling* technique, each path starts with the high supply voltage and switches to the low supply when delay slack is available. Level conversion is performed in the registers at the end of the paths, using circuits such as the one introduced in Figure 11-47. Note that the level conversion is necessary only if the logic section that follows cannot run entirely off the low supply.

The impact of this approach is illustrated in Figure 11-49, which plots the path-length distribution of a typical logic block for single and dual supplies. When a single supply is used, a large fraction of the paths is substantially shorter than critical. Adding the second supply shifts the delay distribution closer to the target delay, making many more paths critical. If the design has very few critical paths and most of paths finish early, energy savings would be large. On the other hand, if most of the paths inside a block are critical, introducing the second supply does not yield any energy savings. You may wonder if adding a third or even a fourth supply voltage may yield even greater benefits to equalize the larger delay spreads. Unfortunately, a number of studies have shown that for typical path-delay distributions, adding more supplies only yields marginal savings [Hamada01, Augsburg02].

Similarly, one may ask whether this dual-supply technique yields a larger benefit, compared with a uniform reduction the supply. In using clustered voltage scaling, the dual-supply approach is **more effective when large switched capacitances are concentrated toward the**

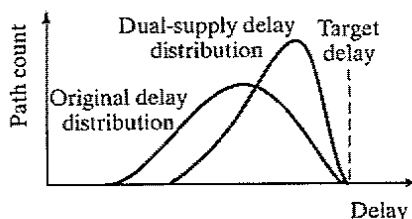


**Figure 11-49** Typical path-delay distribution before and after applying the second supply.

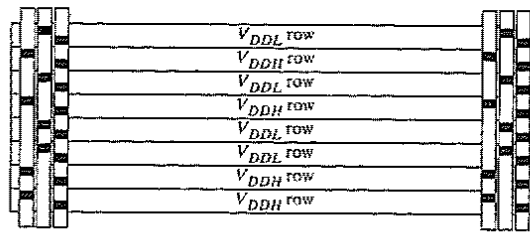


**Figure 11-50** Layout of a dual-supply logic block where all cells at different supplies are placed in different rows.

end of the block, such as in buffer chains [Stojanovic02]. This observation is in accordance with the concept of low-swing busses, introduced in Chapter 9. A bus typically presents the largest capacitance, and it is more advantageous to lower the supply voltage on its driver than that on any other logic gate. Since it has the largest capacitance, the payoff in power savings is the largest for the same increase in delay.

Distributing multiple supply voltages on a die complicates the design of the power-distribution network and tends to tax the traditional place-and-route tools. As we saw many times before, a structured approach can help to minimize the impact. One simple option is to place cells with different supplies into different rows of a standard cell layout, as shown in Figure 11-50. The second supply can be brought in from outside the chip, or it can be generated on the die, using an internal DC–DC converter. Step-down switching DC–DC converters have a conversion efficiency of well over 90% and yield only a minimal overhead. Still the  $V_{DDL}$  distribution network has to meet all necessary design requirements, such as decoupling to minimize the voltage variations and immunity to electromigration.

### Using Multiple Device Thresholds

The use of devices with multiple thresholds offers another way of trading off speed for power. Most sub-0.25- $\mu\text{m}$  CMOS technologies offer two types of  $n$ - and  $p$ -type transistors, with thresholds differing by about 100 mV. This higher threshold device features a leakage current that is about one order of magnitude lower than that of the lower threshold transistor, at the expense of a ~30% reduction in active current. Therefore, the low-threshold devices are preferably used in timing-critical paths, while the high thresholds are used anywhere else. The assignment can be done on a per-cell basis, rather than per individual transistor [Wei99, Kato00]. Note that the use of multiple thresholds does not require level converters or any other special circuits, as shown Figure 11-51. Clustering of the logic is not required either; as many gates as possible should be converted to high threshold until the timing slack is completely consumed. A careful assignment of the thresholds can reduce the leakage power by as much as 80%–90%.

While the multiple-threshold approach primarily addresses the leakage current, it yields a small reduction in active power as an additional benefit. This is primarily due to a reduced gate-channel capacitance in the off state and a small reduction in signal swing on the internal nodes of a gate ( $V_{DD} - V_{ThH}$ ). Such a reduction is partially offset by increased source and drain junction sidewall capacitances. The overall active power reduction turns out to be only about 4%.

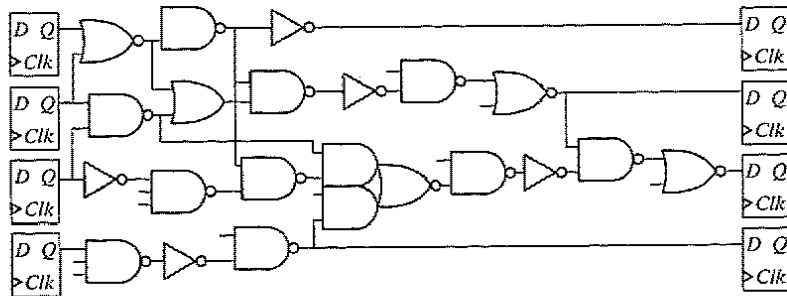


**Figure 11-51** Use of dual-threshold gates. Gray-shaded gates use low thresholds and are employed only in critical paths.

### Reducing Switching Capacitance through Transistor Sizing

The input capacitance of a complementary CMOS gate is directly proportional to its size and, therefore, also to its speed. In Chapter 6, we examined the optimal sizing of gates and found that each gate in a logic path should be sized to have an effective fan-out of approximately 4 to achieve the minimum delay for that path. An interesting question is how to size a circuit for minimum energy when the allowable delay is longer than minimal?

We know that for an inverter chain with a given load and number of stages, the minimum delay is achieved when the size of each inverter in the chain is the geometric mean of its neighbors:

$$s_i^2 = s_{i+1}s_{i-1} \quad (11.28)$$

When the minimum delay that is achieved by this sizing is below the desired delay, an optimization problem can be formulated that minimizes the switching capacitance under delay constraints. One option is to reduce the number of stages and increase the tapering factor, as suggested in Chapter 9. Even better, an analytical solution exists for this problem [Ma94], which establishes that the optimal approach is to adjust the tapering factor at each stage according to the following equation:

$$s_i^2 = \frac{s_{i+1}s_{i-1}}{1 + \mu s_{i-1}} \quad (11.29)$$

The parameter  $\mu$  is a nonnegative number that depends on the amount of timing slack available (with respect to the chain sized for the minimum delay). This solution is intuitively clear—the last stages in the inverter chain are the largest; hence, they are the prime candidates for downsizing. Since downsizing any of the inverters in the chain by a given percentage causes the same delay increment. Hence, we rather do it at the stage where we get the largest impact, which is the largest one.

The same principle applies to a logic path. When it is optimized for minimum delay, the delay of each stage is the same (while the intrinsic delays of gates may differ). Therefore,

the delay of the largest energy consuming gates should be increased first. This idea can be extrapolated to the energy–delay optimization of a general combinational logic block. However, applying it in practice is not trivial. Downsizing one path affects the delay of all paths that share logic gates with it, making it difficult to isolate and optimize one particular path. In addition, many paths in a general combinational block are reconvergent.

### Example 11.8 Energy-Delay Trade-Offs in Adder Design

Let us examine the energy-delay optimization of the 64-bit Kogge–Stone tree adder. There are many paths through an adder, and not all of these paths are balanced. A crucial question is how to identify the paths for resizing or initial sizing. One option is to select all the paths in the adder equal to the critical path. Since the paths through an adder roughly correspond to different bit slices, we allocate each gate in the adder to a bit slice. There are 64 bit slices, and a total of nine stages of logic. This partition works well for Figure 11-52a shows the resulting energy map for the minimum delay. It can be seen that the adder consumes the largest energy along the longest carry paths. Figure 11-52b shows the energy profile of the same adder, this time resized, allowing for a 10% delay increase. The energy dissipation is reduced by 54%. In contrast, a dual-supply solution saves only 27%, while a single reduced supply yields 22% savings [Stojanovic02]. In summary, **sizing is an effective power-reduction method for datapaths, where the majority of energy is consumed inside the block, rather than in driving the external load.**

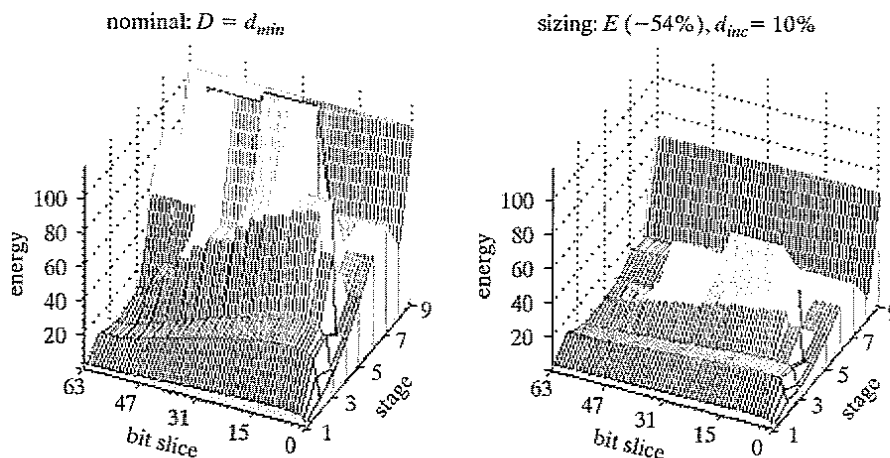


**Figure 11-52** Energy profiles of a 64-bit adder: (a) sized for minimum delay, (b) sized for 10% delay increment and minimized energy.

### Reducing Switching Capacitance through Logic and Architecture Optimizations

As the effective capacitance is the product of the physical capacitance and the switching activity, minimization of both factors is recommended. A wide variety of logic and architectural optimizations exist that reduce the activity at no expense in performance. Some of those were already introduced in Chapter 6. For the sake of brevity, we only show a small number of representative optimizations. (Refer to [Rabaey96] and [Chandrakasan98] for an in-depth overview.)

**Reducing Switching Activity by Resource Allocation** Multiplexing multiple operations on a single hardware unit has a detrimental effect on the power consumption. Besides increasing the physical capacitance, it can also increase the switching activity. This is illustrated with a simple experiment in Figure 11-53, which compares the power consumption of two counters running simultaneously. In the first case, both counters run on separate hardware, while in the second case, they are multiplexed on the same unit. Figure 11-53b plots the number of switching events as a function of the skew between the two counters. The nonmultiplexed case is always superior, except when both counters run in a completely synchronous fashion. The multiplexing tends to randomize the data signals presented to the operational unit, which results in increased switching activity. When power consumption is a concern, it is often beneficial to avoid the excessive reuse of resources. Observe that CMOS hardware units consume only negligible amounts of power when idle. Providing dedicated, specialized operators only presents an extra cost in area, while being generally beneficial in terms of speed and power.

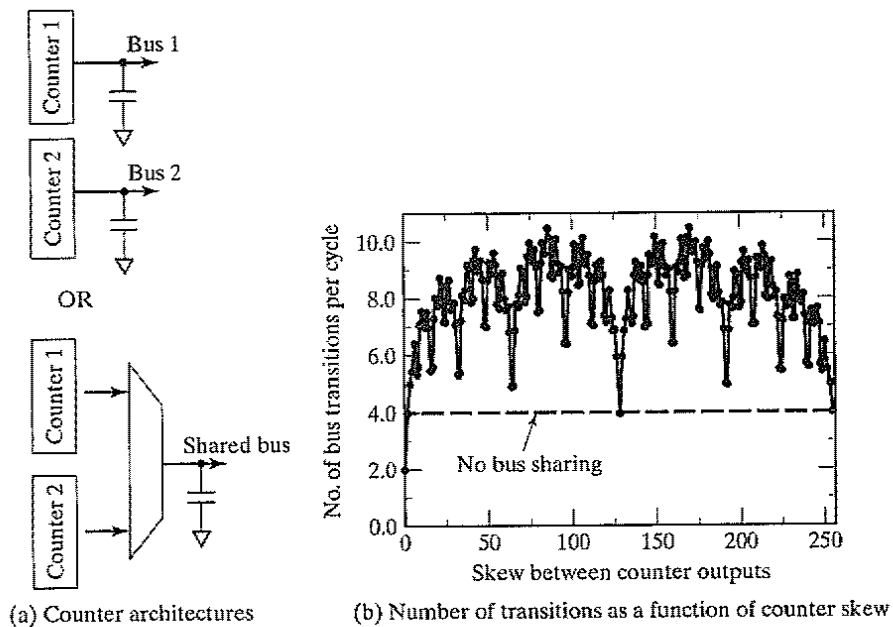


**Figure 11-53** Multiplexing increases the switching activity.

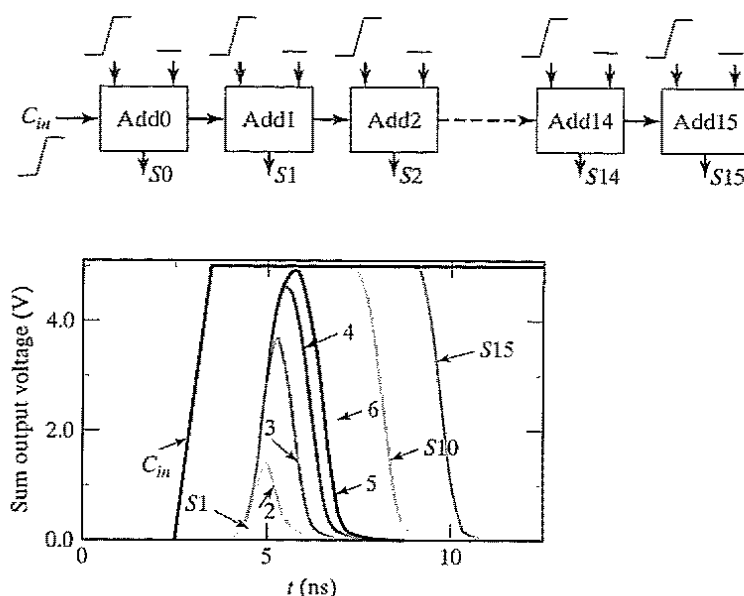


Figure 11-54 Glitching in the sum bits of a 16-bit ripple-carry adder.

**Reducing Glitching through Path Balancing** Dynamic hazards, or *glitching*, are a major contribution to the dissipation in complex structures such as adders and multipliers. The wide discrepancy in the lengths of the signal paths in some of those structures can be the cause of spurious transients. This is demonstrated in Figure 11-54, which displays the simulated response of a 16-bit ripple adder for all inputs going simultaneously from 0 to 1. A number of the sum bits are shown as a function of time. The sum signals should be zero for all bits. Unfortunately, a 1 appears briefly at all of the outputs, since the carry takes a significant amount of time to propagate from the first bit to the last. Notice how the glitch becomes more pronounced as it travels down the chain.

A dramatic reduction in glitching activity can be obtained by selecting structures with balanced signal paths. The tree lookahead adder structures (such as Kogge–Stone) and the tree multipliers have this property; therefore, they should be more attractive from a power point of view, even in the presence of a larger physical capacitance. An inspection of the lookahead structure of Figure 11-22 reveals that the timing paths to the inputs of dot operators are of a similar length, although some deviations may occur due to differences in loading and fan-out.

### 11.7.2 Run-Time Power Management

#### Dynamic Supply Voltage Scaling (DVS)

A static reduction of the supply voltage, as discussed in the preceding paragraphs, lowers the energy per operation and extends the battery life at the expense of performance. This performance penalty is often not acceptable, especially in applications that are latency constrained.

Substantial power reductions are still obtainable, based on the realization that the peak performance is not continuously required. Consider, for instance, a general-purpose processor to be used in portable applications, such as notebooks, electronic organizers, and cellular phones. The computational functions to be executed on such a processor fall into three major categories: compute-intensive tasks, low-speed functions, and idle-mode operation. Compute-intensive and short-latency tasks need the full computational throughput of the processor to achieve real-time constraints. MPEG video and audio decompression are examples of such. Low-throughput and long-latency tasks, such as text processing, data entry, and memory backups, operate under far more relaxed completion deadlines and require only a fraction of the maximum throughput of the microprocessor. There is no reward for finishing the computation early, and if a task is completed early, it can be considered a waste of energy. Finally, portable processors spend a large fraction of their time on idle, waiting for a user action or an external wake-up event. In sum, the computational throughput and latency expected from a mobile processor vary drastically over time.

Even compute-intensive operations, such as MPEG decoding, show variable computational requirements while processing a typical stream of data. For example, the number of times an MPEG decoder computes an inverse discrete cosine transform (IDCT) per video frame varies widely, depending upon the amount of motion in the video scenes. This is illustrated in Figure 11-55, which plots the distribution of the number of IDCTs/frame for a typical video sequence. The processor that is executing this algorithm experiences a different computational workload from frame to frame.

Lowering the clock frequency when executing the reduced workloads reduces the power, but does not save on energy—every operation is still executed at the high voltage level. However, if both supply voltage and frequency are lowered simultaneously, the energy is reduced. In order to maintain the required throughput for high workloads and minimize energy for low workloads, both supply and frequency must be dynamically varied according to the requirements application that is currently being executed. This technique is called *dynamic voltage scaling* (DVS). The concept is illustrated in Figure 11-56 [Burd00, Gutnik97]. It operates under

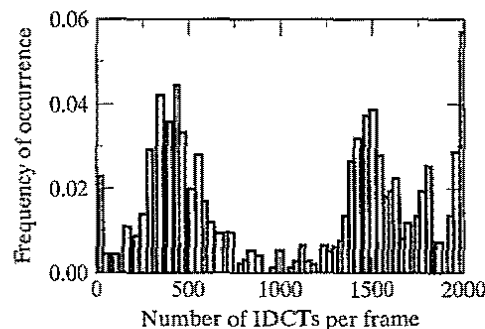


Figure 11-55 Typical IDCT histogram for MPEG decoding.

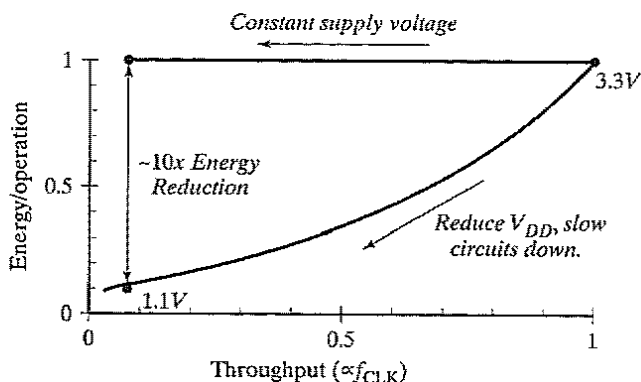


**Figure 11-56** Energy/operation versus throughput ( $1/T$ ) for constant and variable supply voltage operation.

the guideline that a function should always be operated at the lowest supply voltage that meets the timing constraints.

The DVS concept is enabled by the observation that the delay of most CMOS circuits and functions track each other well over a range of supply voltages, which is a necessity for system operation under varying supply conditions. Figure 11-57 shows the delays of a number of representative CMOS blocks (such as NAND gates, ring oscillators, register files, and SRAM), over a supply voltage range from 1 V to 4 V [Burd00]. Excellent performance tracking can be observed. Note that some circuit families, such as NMOS-only pass-transistor logic do not follow this behavior over the complete range of supplies.

A practical implementation of a dynamic-voltage scaling system now consists of the following components:

- a processor that can operate under a wide variety of supply voltages,

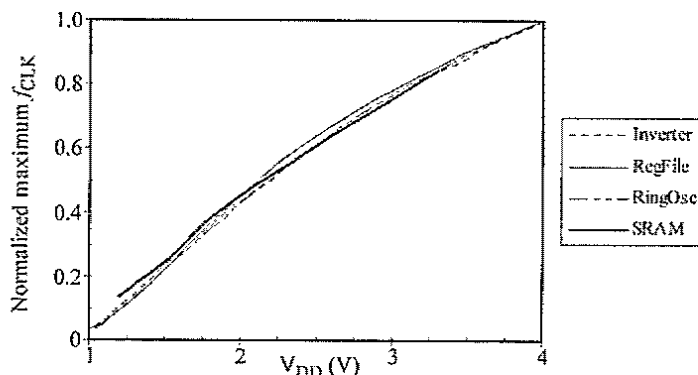


**Figure 11-57** Delay of representative CMOS functions as a function of supply voltage for a 0.6- $\mu\text{m}$  CMOS technology.

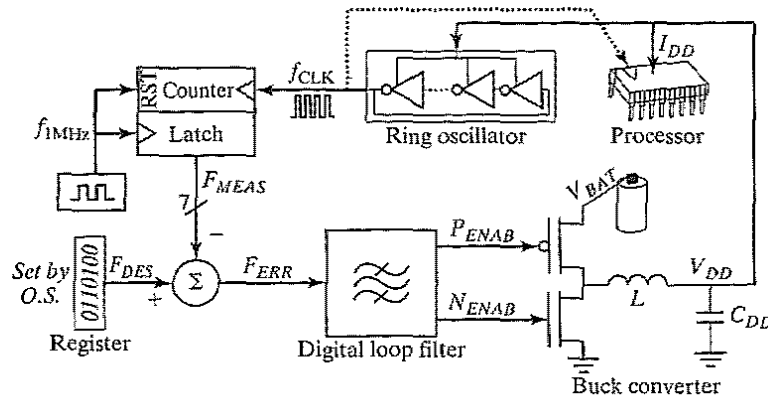


**Figure 11-58** Block diagram of a dynamic voltage-scaled system.

- a supply-regulation loop that sets the minimum voltage necessary for operation at a desired frequency, and
- an operating system that calculates the required frequencies to meet requested throughputs and task completion deadlines.

One possible implementation is shown in Figure 11-58. The core of the DVS system is a ring oscillator, whose oscillating frequency matches the microprocessor critical path. When included inside the power-supply control loop, this ring oscillator provides the translation between the supply voltage and the clock frequency. The operating system digitally sets the desired frequency ( $F_{DES}$ ). The current value of ring oscillator frequency is measured and compared with the desired frequency. The difference is used as a feedback error. By adjusting the supply voltage, the supply-voltage loop changes the ring oscillator frequency to set this error value to 0.

The task of the scheduler (or real-time operating system) is to determine dynamically the optimal frequency (or voltage) as a function of the combined computational requirements of all active tasks in the system. In the more complex case of a general-purpose processor, each task should supply a completion deadline (e.g., video frame rate) or a desired execution frequency. The voltage scheduler then estimates the number of processor cycles necessary for completing each of the tasks and computes the optimal processor frequency [Pering99]. In the case of a single task with varying performance requirements, a queue can be used to determine the computational load and to adjust the voltage accordingly. This is illustrated in Figure 11-59, where the depth of the input queue is used to set the supply voltage (and frequency) of a stream-based signal processor [Gutnik97].

### Dynamic Threshold Scaling (DTS)

In analogy to the dynamic variation of the supply voltage, it is attractive to adjust the threshold voltage of the transistors dynamically. For low-latency computation, the threshold should be lowered to its minimal value; for low speed computation, it can be increased; and in the standby

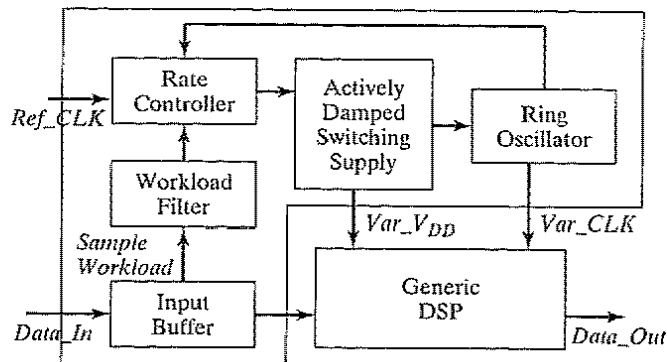


**Figure 11-59** Using a queue to determine the workload in a signal processor [Gutnik97].

mode, it should be set to the highest possible value to minimize the leakage current. Substrate bias is the control knob that allows us to vary the threshold voltages dynamically. In order to do so, we have to operate the transistors as four-terminal devices. This is only possible in a triple-well process, as independent control of all four terminals of both  $n$ - and  $p$ -type devices is required, as shown in Figure 11-60. Substrate biasing can be implemented for a complete chip, on a block-by-block or a cell-by-cell basis. Per cell granularity of substrate biasing, however, has a large layout cost.

Similar to dynamic supply-voltage scaling scheme, the variable threshold voltage scheme is based on a feedback loop, which can be set to accomplish a variety of goals:

- It can lower the leakage in the standby mode
- It can compensate for threshold variations across the chip during normal operation of the circuit
- It can throttle the throughput of the circuit to lower both the active and leakage power based on performance requirements.

Since the current flow into the substrate is much smaller than into the supply lines, DTS has a smaller circuit overhead than DVS. A feedback system, designed to control the leakage in

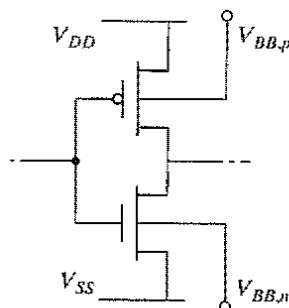


**Figure 11-60** An inverter with body terminals used for threshold control.