Datapath Subsystems 11

11.1 Introduction

Chip functions generally can be divided into the following categories:

- Datapath operators
- Memory elements
- Control structures
- Special-purpose cells
  - I/O
  - Power distribution
  - Clock generation and distribution
  - Analog and RF

CMOS system design consists of partitioning the system into subsystems of the types listed above. Many options exist that make trade-offs between speed, density, programmability, ease of design, and other variables. This chapter addresses design options for common datapath operators. The next chapter addresses arrays, especially those used for memory. Control structures are most commonly coded in a hardware description language and synthesized. Special-purpose subsystems are considered in Chapter 13.

As introduced in Chapter 1, datapath operators benefit from the structured design principles of hierarchy, regularity, modularity, and locality. They may use N identical circuits to process N-bit data. Related data operators are placed physically adjacent to each other to reduce wire length and delay. Generally, data is arranged to flow in one direction, while control signals are introduced in a direction orthogonal to the dataflow.

Common datapath operators considered in this chapter include adders, one/zero detectors, comparators, counters, Boolean logic units, error-correcting code blocks, shifters, and multipliers.

11.2 Addition/Subtraction

> *"Multitudes of contrivances were designed, and almost endless drawings made, for the purpose of economizing the time and simplifying the mechanism of carriage."*
>
> —Charles Babbage, on Difference Engine No. 1, 1864 [Morrison61]

Addition forms the basis for many processing operations, from ALUs to address generation to multiplication to filtering. As a result, adder circuits that add two binary numbers are of great interest to digital system designers. An extensive, almost endless, assortment of adder architectures serve different speed/power/area requirements. This section begins with half adders and full adders for single-bit addition. It then considers a plethora of carry-propagate adders (CPAs) for the addition of multibit words. Finally, related structures such as subtracters and multiple-input adders are discussed.

11.2.1 Single-Bit Addition

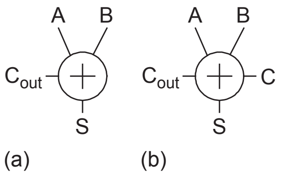


FIGURE 11.1
Half and full adders

The *half adder* of Figure 11.1(a) adds two single-bit inputs, A and B. The result is 0, 1, or 2, so two bits are required to represent the value; they are called the sum S and carry-out C_{out}. The carry-out is equivalent to a carry-in to the next more significant column of a multibit adder, so it can be described as having double the *weight* of the other bits. If multiple adders are to be cascaded, each must be able to receive the carry-in. Such a *full adder* as shown in Figure 11.1(b) has a third input called C or C_{in}.

The truth tables for the half adder and full adder are given in Tables 11.1 and 11.2. For a full adder, it is sometimes useful to define *Generate* (G), *Propagate* (P), and *Kill* (K) signals. The adder generates a carry when C_{out} is true independent of C_{in}, so $G = A \cdot B$. The adder kills a carry when C_{out} is false independent of C_{in}, so $K = \overline{A} \cdot \overline{B} = \overline{A + B}$. The adder propagates a carry; i.e., it produces a carry-out if and only if it receives a carry-in, when exactly one input is true: $P = A \oplus B$.

TABLE 11.1 Truth table for half adder

A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

TABLE 11.2 Truth table for full adder

A	B	C	G	P	K	C_{out}	S
0	0	0	0	0	1	0	0
		1				0	1
0	1	0	0	1	0	0	1
		1				1	0
1	0	0	0	1	0	0	1
		1				1	0
1	1	0	1	0	0	1	0
		1				1	1

From the truth table, the half adder logic is

$$\begin{aligned} S &= A \oplus B \\ C_{out} &= A \cdot B \end{aligned} \tag{11.1}$$

and the full adder logic is

$$\begin{aligned} S &= A\bar{B}\bar{C} + \bar{A}B\bar{C} + \bar{A}\bar{B}C + ABC \\ &= (A \oplus B) \oplus C = P \oplus C \\ C_{\text{out}} &= AB + AC + BC \\ &= AB + C(A + B) \\ &= \overline{\bar{A}\bar{B} + \bar{C}(\bar{A} + \bar{B})} \\ &= \text{MAJ}(A,\ B,\ C) \end{aligned} \tag{11.2}$$

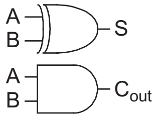


FIGURE 11.2 Half adder design

The most straightforward approach to designing an adder is with logic gates. Figure 11.2 shows a half adder. Figure 11.3 shows a full adder at the gate (a) and transistor (b) levels. The carry gate is also called a *majority* gate because it produces a 1 if at least two of the three inputs are 1. Full adders are used most often, so they will receive the attention of the remainder of this section.

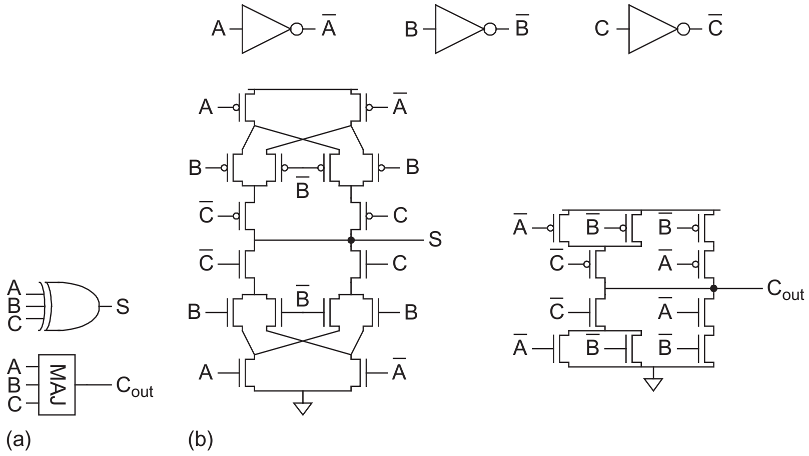


FIGURE 11.3 Full adder design

The full adder of Figure 11.3(b) employs 32 transistors (6 for the inverters, 10 for the majority gate, and 16 for the 3-input XOR). A more compact design is based on the observation that S can be factored to reuse the C_{out} term as follows:

$$S = ABC + (A + B + C)\bar{C}_{\text{out}} \tag{11.3}$$

Such a design is shown at the gate (a) and transistor (b) levels in Figure 11.4 and uses only 28 transistors. Note that the pMOS network is identical to the nMOS network rather than being the conduction complement, so the topology is called a *mirror adder*. This simplification reduces the number of series transistors and makes the layout more uniform. It is possible because the addition function is *symmetric*; i.e., the function of complemented inputs is the complement of the function.

The mirror adder has a greater delay to compute S than C_{out}. In carry-ripple adders (Section 11.2.2.1), the critical path goes from C to C_{out} through many full adders, so the

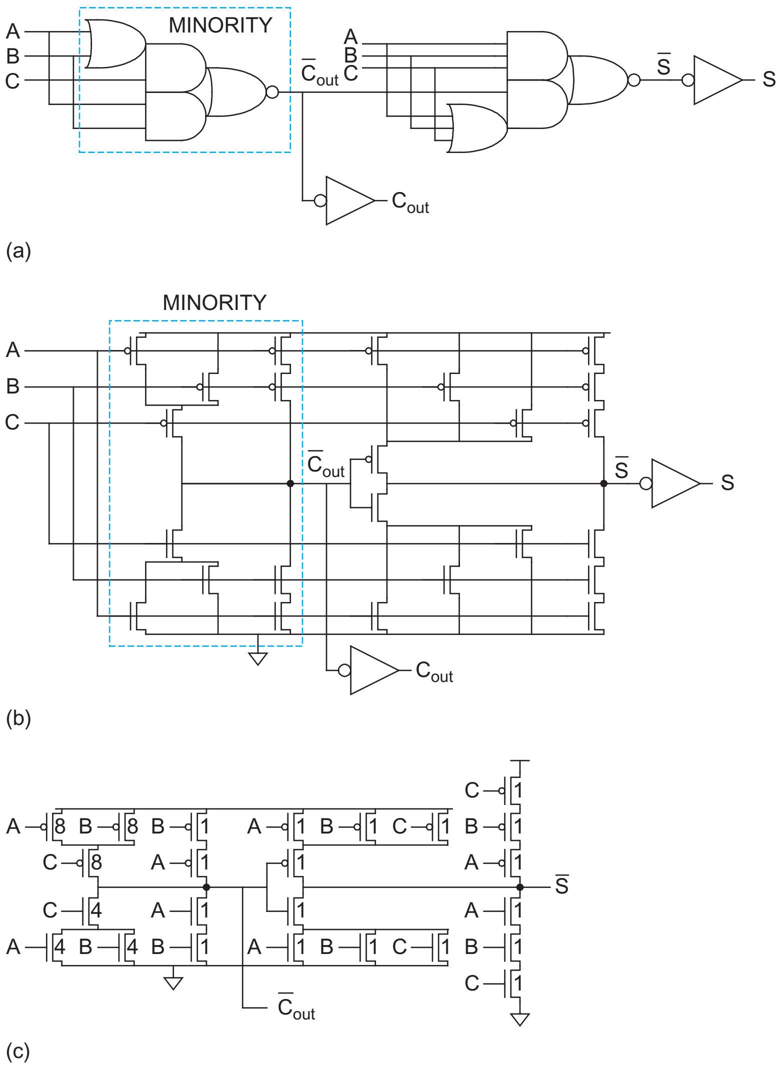


FIGURE 11.4 Full adder for carry-ripple operation

extra delay computing S is unimportant. Figure 11.4(c) shows the adder with transistor sizes optimized to favor the critical path using a number of techniques:

- Feed the carry-in signal (C) to the inner inputs so the internal capacitance is already discharged.
- Make all transistors in the sum logic whose gate signals are connected to the carry-in and carry logic minimum size (1 unit, e.g., 4 λ). This minimizes the branching effort on the critical path. Keep routing on this signal as short as possible to reduce interconnect capacitance.
- Determine widths of series transistors by logical effort and simulation. Build an asymmetric gate that reduces the logical effort from C to C_{out} at the expense of effort to S.

- Use relatively large transistors on the critical path so that stray wiring capacitance is a small fraction of the overall capacitance.
- Remove the output inverters and alternate positive and negative logic to reduce delay and transistor count to 24 (see Section 11.2.2.1).

Figure 11.5 shows two layouts of the adder (see also the inside front cover). The choice of the aspect ratio depends on the application. In a standard-cell environment, the layout of Figure 11.5(a) might be appropriate when a single row of nMOS and pMOS transistors is used. The routing for the *A*, *B*, and *C* inputs is shown inside the cell, although it could be placed outside the cell because external routing tracks have to be assigned to these signals anyway. Figure 11.5(b) shows a layout that might be appropriate for a dense datapath (if horizontal polysilicon is legal). Here, the transistors are rotated and all of the wiring is completed in polysilicon and metal1. This allows metal2 bus lines to pass over the cell horizontally. Moreover, the widths of the transistors can increase

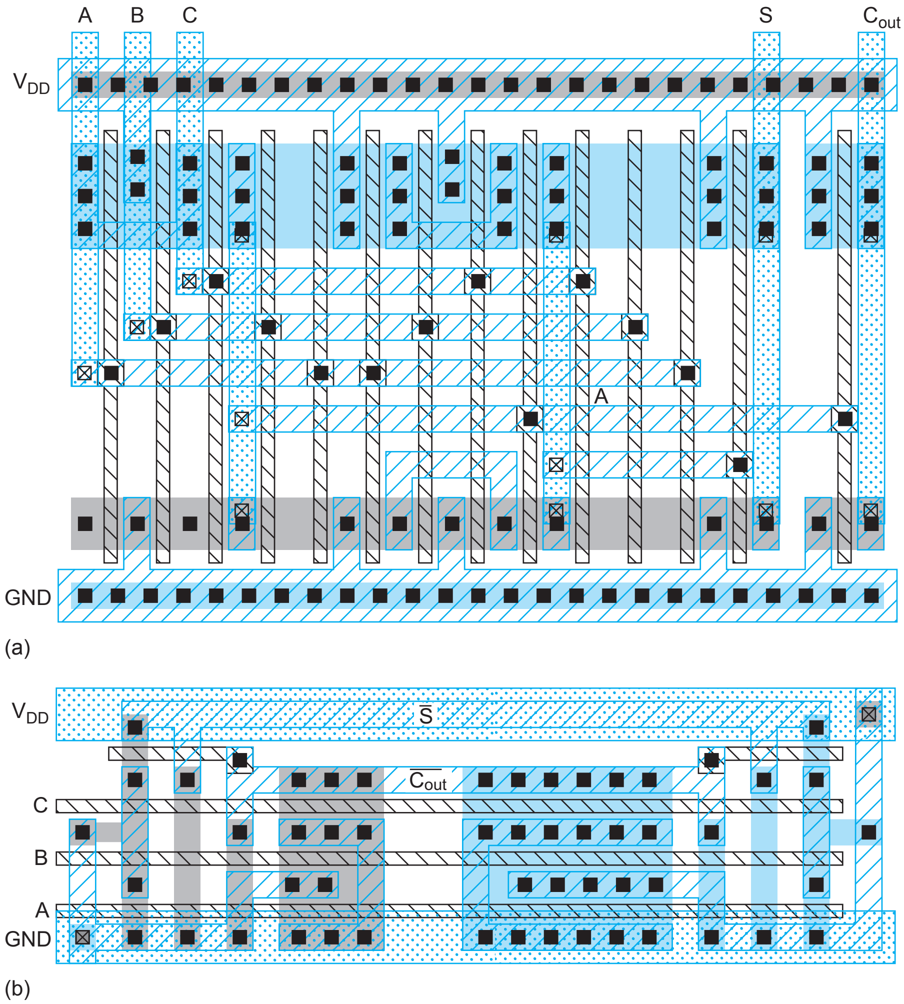


FIGURE 11.5 Full adder layouts. Color version on inside front cover.

without impacting the bit-pitch (height) of the datapath. In this case, the widths are selected to reduce the C_{in} to C_{out} delay that is on the critical path of a carry-ripple adder.

A rather different full adder design uses transmission gates to form multiplexers and XORs. Figure 11.6(a) shows the transistor-level schematic using 24 transistors and providing buffered outputs of the proper polarity with equal delay. The design can be understood by parsing the transmission gate structures into multiplexers and an "invertible inverter" XOR structure (see Section 11.7.4), as drawn in Figure 11.6(b).[1] Note that the multiplexer choosing S is configured to compute $P \oplus C$, as given in EQ (11.2).

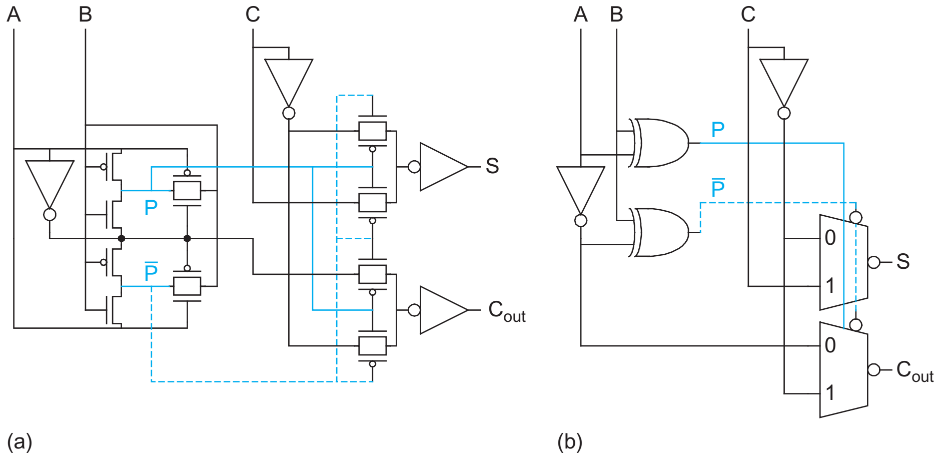


FIGURE 11.6 Transmission gate full adder

Figure 11.7 shows a complementary pass-transistor logic (CPL) approach. In comparison to a poorly optimized 40-transistor static CMOS full adder, [Yano90] finds CPL is twice as fast, 30% lower in power, and slightly smaller. On the other hand, in comparison to a careful implementation of the mirror adder, [Zimmermann97] finds the CPL delay slightly better, the power comparable, and the area much larger.

Dynamic full adders are widely used in fast multipliers when power is not a concern. As the sum logic inherently requires true and complementary versions of the inputs, dual-rail domino is necessary. Figure 11.8 shows such an adder using footless dual-rail domino XOR/XNOR and MAJORITY/MINORTY gates [Heikes94]. The delays to the two outputs are reasonably well balanced, which is important for multipliers where both paths are critical. It shares transistors in the sum gate to reduce transistor count and takes advantage of the symmetric property to provide identical layouts for the two carry gates.

Static CMOS full adders typically have a delay of 2–3 FO4 inverters, while domino adders have a delay of about 1.5.

11.2.2 Carry-Propagate Addition

N-bit adders take inputs $\{A_N, \ldots, A_1\}$, $\{B_N, \ldots, B_1\}$, and carry-in C_{in}, and compute the sum $\{S_N, \ldots, S_1\}$ and the carry-out of the most significant bit C_{out}, as shown in Figure 11.9.

[1] Some switch-level simulators, notably IRSIM, are confused by this XOR structure and may not simulate it correctly.

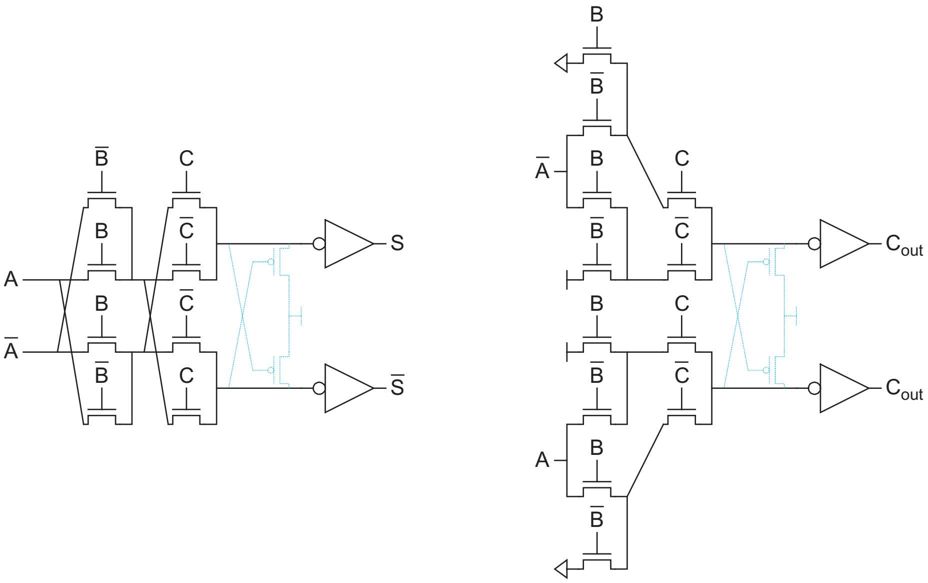


FIGURE 11.7 CPL full adder

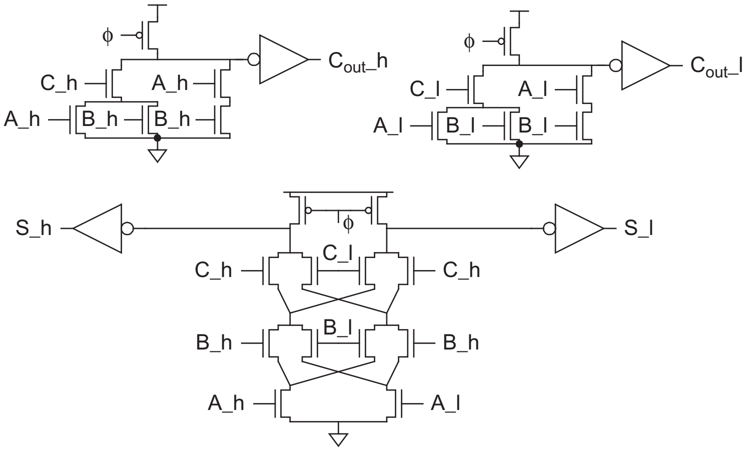


FIGURE 11.8 Dual-rail domino full

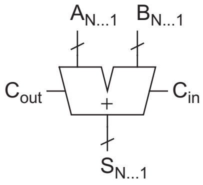


FIGURE 11.9 Carry-propagate adder

(Ordinarily, this text calls the least significant bit A_0 rather than A_1. However, for adders, the notation developed on subsequent pages is more graceful if column 0 is reserved to handle the carry.) They are called *carry-propagate adders* (CPAs) because the carry into each bit can influence the carry into all subsequent bits. For example, Figure 11.10 shows the addition $1111_2 + 0000_2 + 0/1$, in which each of the sum and carry bits is influenced by C_{in}. The simplest design is the carry-ripple adder in which the carry-out of one bit is simply connected as the carry-in to the next. Faster adders look ahead to predict the carry-out of a multibit group. This is usually done by computing group PG signals to indicate

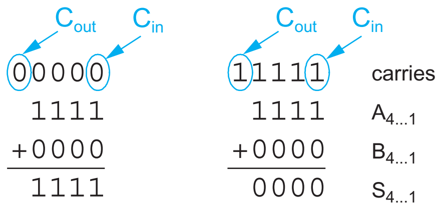


FIGURE 11.10 Example of carry propagation

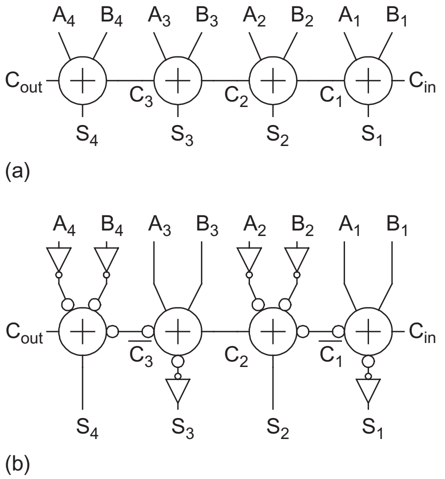


FIGURE 11.11 4-bit carry-ripple adder

whether the multibit group will propagate a carry-in or will generate a carry-out. Long adders use multiple levels of lookahead structures for even more speed.

11.2.2.1 Carry-Ripple Adder

An N-bit adder can be constructed by cascading N full adders, as shown in Figure 11.11(a) for $N = 4$. This is called a *carry-ripple adder* (or *ripple-carry adder*). The carry-out of bit i, C_i, is the carry-in to bit $i + 1$. This carry is said to have twice the *weight* of the sum S_i. The delay of the adder is set by the time for the carries to ripple through the N stages, so the $t_{C \to Cout}$ delay should be minimized.

This delay can be reduced by omitting the inverters on the outputs, as was done in Figure 11.4(c). Because addition is a *self-dual function* (i.e., the function of complementary inputs is the complement of the function), an inverting full adder receiving complementary inputs produces true outputs. Figure 11.11(b) shows a carry-ripple adder built from inverting full adders. Every other stage operates on complementary data. The delay inverting the adder inputs or sum outputs is off the critical ripple-carry path.

11.2.2.2 Carry Generation and Propagation

This section introduces notation commonly used in describing faster adders. Recall that the P (*propagate*) and G (*generate*) signals were defined in Section 11.2.1. We can generalize these signals to describe whether a group spanning bits $i \ldots j$, inclusive, generate a carry or propagate a carry. A group of bits generates a carry if its carry-out is true independent of the carry-in; it propagates a carry if its carry-out is true when there is a carry-in. These signals can be defined recursively for $i \geq k > j$ as

$$\begin{aligned} G_{i:j} &= G_{i:k} + P_{i:k} \cdot G_{k-1:j} \\ P_{i:j} &= P_{i:k} \cdot P_{k-1:j} \end{aligned} \qquad \textbf{(11.4)}$$

with the base case

$$\begin{aligned} G_{i:i} &\equiv G_i = A_i \cdot B_i \\ P_{i:i} &\equiv P_i = A_i \oplus B_i \end{aligned} \qquad \textbf{(11.5)}$$

In other words, a group generates a carry if the upper (more significant) or the lower portion generates and the upper portion propagates that carry. The group propagates a carry if both the upper and lower portions propagate the carry.[2]

The carry-in must be treated specially. Let us define $C_0 = C_{\text{in}}$ and $C_N = C_{\text{out}}$. Then we can define generate and propagate signals for bit 0 as

$$\begin{aligned} G_{0:0} &= C_{\text{in}} \\ P_{0:0} &= 0 \end{aligned} \qquad \textbf{(11.6)}$$

[2] Alternatively, many adders use $\overline{K}_i = A_i + B_i$ in place of P_i because OR is faster than XOR. The group logic uses the same gates: $G_{i:j} = G_{i:k} + \overline{K}_{i:k} \cdot G_{k-1:j}$ and $\overline{K}_{i:j} = \overline{K}_{i:k} \cdot \overline{K}_{k-1:j}$. However, $P_i = A_i \oplus B_i$ is still required in EQ (11.7) to compute the final sum. It is sometimes renamed X_i or T_i to avoid ambiguity.

Observe that the carry into bit i is the carry-out of bit i–1 and is $C_{i-1} = G_{i-1:0}$. This is an important relationship; *group generate* signals and *carries* will be used synonymously in the subsequent sections. We can thus compute the sum for bit i using EQ (11.2) as

$$S_i = P_i \oplus G_{i-1:0} \tag{11.7}$$

Hence, addition can be reduced to a three-step process:

1. Computing bitwise generate and propagate signals using EQs (11.5) and (11.6)
2. Combining PG signals to determine group generates $G_{i-1:0}$ for all $N \geq i \geq 1$ using EQ (11.4)
3. Calculating the sums using EQ (11.7)

These steps are illustrated in Figure 11.12. The first and third steps are routine, so most of the attention in the remainder of this section is devoted to alternatives for the group PG logic with different trade-offs between speed, area, and complexity. Some of the hardware can be shared in the bitwise PG logic, as shown in Figure 11.13.

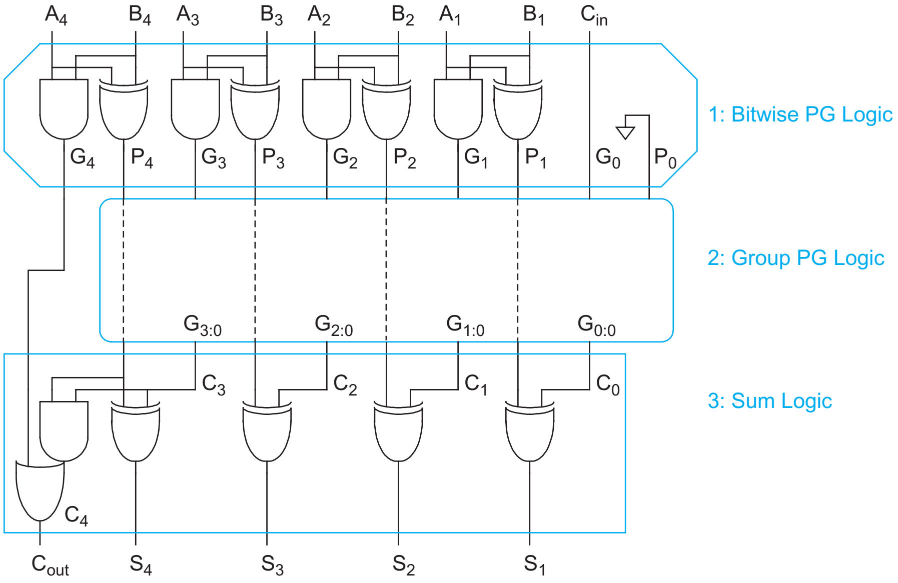


FIGURE 11.12 Addition with generate and propagate logic

Many notations are used in the literature to describe the group PG logic. In general, PG logic is an example of a *prefix* computation [Leighton92]. It accepts inputs $\{P_{N:N}, \ldots, P_{0:0}\}$ and $\{G_{N:N}, \ldots, G_{0:0}\}$ and computes the *prefixes* $\{G_{N:0}, \ldots, G_{0:0}\}$ using the relationship given in EQ (11.4). This relationship is given many names in the literature including the *delta operator*, *fundamental carry operator*, and *prefix operator*. Many other problems such as priority encoding can be posed as prefix computations and all the techniques used to build fast group PG logic will apply, as we will explore in Section 11.10.

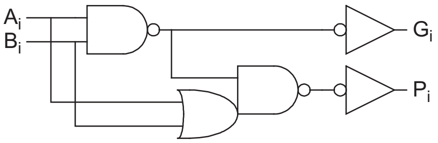


FIGURE 11.13 Shared bitwise PG logic

EQ (11.4) defines *valency*-2 (also called *radix*-2) group PG logic because it combines pairs of smaller groups. It is also possible to define higher-valency group logic to use fewer stages of more complex gates [Beaumont-Smith99], as shown in EQ (11.8) and later in Figure 11.16(c). For example, in valency-4 group logic, a group propagates the carry if all four portions propagate. A group generates a carry if the upper portion generates, the second portion generates and the upper propagates, the third generates and the upper two propagate, or the lower generates and the upper three propagate.

$$\left.\begin{aligned} G_{i:j} &= G_{i:k} + P_{i:k}\cdot G_{k-1:l} + P_{i:k}\cdot P_{k-1:l}\cdot G_{l-1:m} + P_{i:k}\cdot P_{k-1:l}\cdot P_{l-1:m}\cdot G_{m-1:j} \\ &= G_{i:k} + P_{i:k}\left(G_{k-1:l} + P_{k-1:l}\left(G_{l-1:m} + P_{l-1:m}G_{m-1:j}\right)\right) \\ P_{i:j} &= P_{i:k}\cdot P_{k-1:l}\cdot P_{l-1:m}\cdot P_{m-1:j} \end{aligned}\right\} \left(i \geq k > l > m > j\right) \qquad \textbf{(11.8)}$$

Logical Effort teaches us that the best stage effort is about 4. Therefore, it is not necessarily better to build fewer stages of higher-valency gates; simulations or calculations should be done to compare the alternatives for a given process technology and circuit family.

11.2.2.3 PG Carry-Ripple Addition

The critical path of the carry-ripple adder passes from carry-in to carry-out along the carry chain majority gates. As the P and G signals will have already stabilized by the time the carry arrives, we can use them to simplify the majority function into an AND-OR gate:[3]

$$\begin{aligned} C_i &= A_iB_i + \left(A_i + B_i\right)C_{i-1} \\ &= A_iB_i + \left(A_i \oplus B_i\right)C_{i-1} \\ &= G_i + P_iC_{i-1} \end{aligned} \qquad \textbf{(11.9)}$$

Because $C_i = G_{i:0}$, carry-ripple addition can now be viewed as the extreme case of group PG logic in which a 1-bit group is combined with an i-bit group to form an (i+1)-bit group

$$G_{i:0} = G_i + P_i \cdot G_{i-1:0} \qquad \textbf{(11.10)}$$

In this extreme, the group propagate signals are never used and need not be computed. Figure 11.14 shows a 4-bit carry-ripple adder. The critical carry path now proceeds through a chain of AND-OR gates rather than a chain of majority gates. Figure 11.15 illustrates the group PG logic for a 16-bit carry-ripple adder, where the AND-OR gates in the group PG network are represented with gray cells.

Diagrams like these will be used to compare a variety of adder architectures in subsequent sections. The diagrams use black cells, gray cells, and white buffers defined in Figure 11.16(a) for valency-2 cells. Black cells contain the group generate and propagate logic (an AND-OR gate and an AND gate) defined in EQ (11.4). Gray cells containing only the group generate logic are used at the final cell position in each column because only the group generate signal is required to compute the sums. Buffers can be used to minimize the load on critical paths. Each line represents a *bundle* of the group generate and propagate signals (propagate signals are omitted after gray cells). The bitwise PG and

[3] Whenever positive logic such as AND-OR is described, you can also use an AOI gate and alternate positive and negative polarity stages as was done in Figure 11.11(b) to save area and delay.

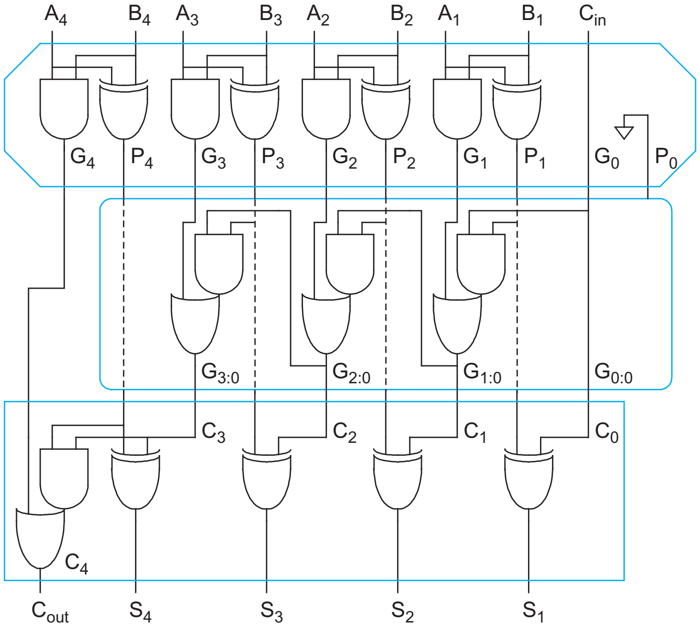


FIGURE 11.14 4-bit carry-ripple adder using PG logic

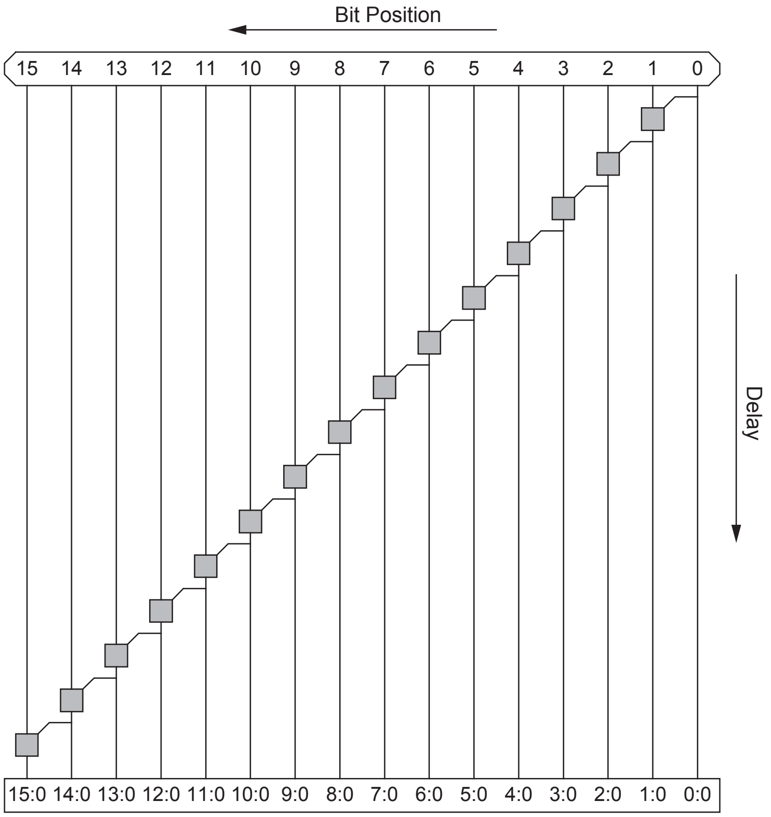


FIGURE 11.15 Carry-ripple adder group PG network

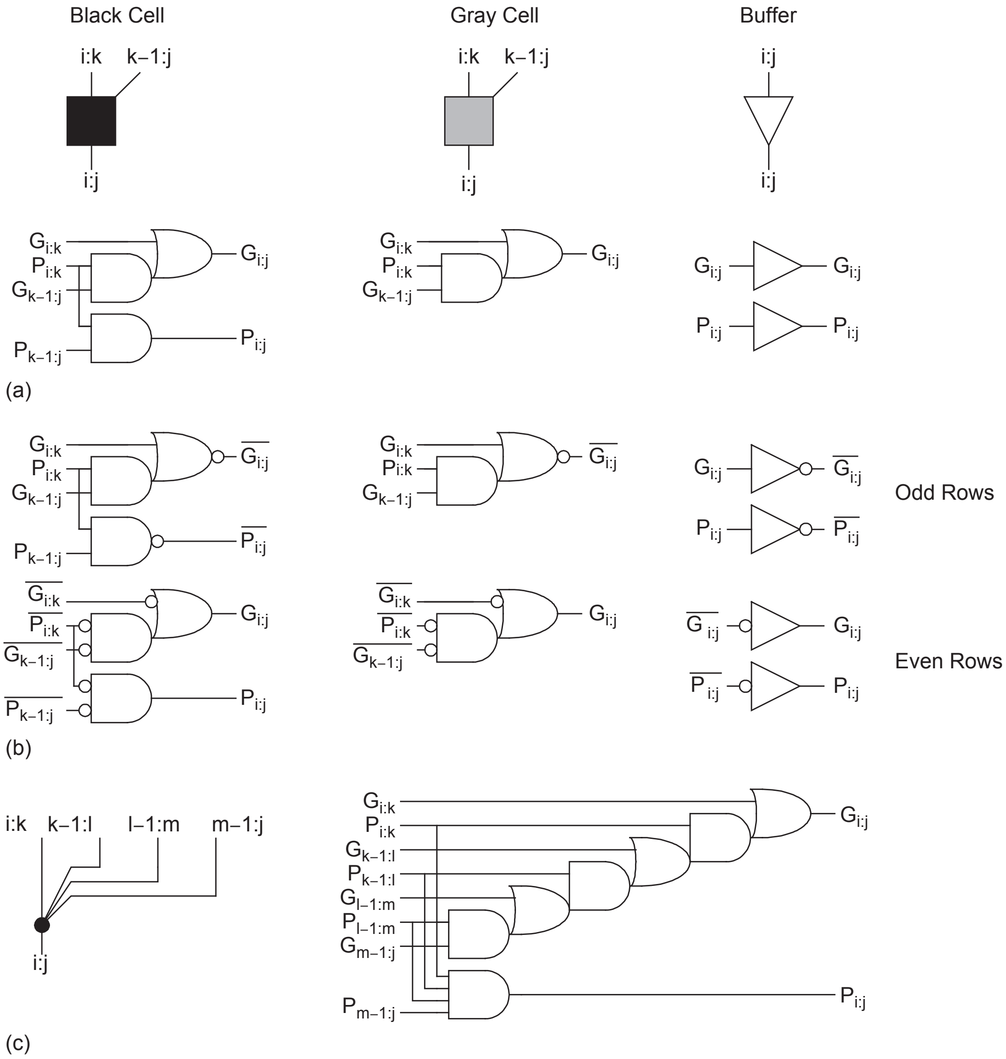


FIGURE 11.16 Group PG cells

sum XORs are abstracted away in the top and bottom boxes and it is assumed that an AND-OR gate operates in parallel with the sum XORs to compute the carry-out:

$$C_{\text{out}} = G_{N:0} = G_N + P_N G_{N-1:0} \tag{11.11}$$

The cells are arranged along the vertical axis according to the time at which they operate [Guyot97]. From Figure 11.15 it can be seen that the carry-ripple adder critical path delay is

$$t_{\text{ripple}} = t_{pg} + (N-1)\,t_{AO} + t_{\text{xor}} \tag{11.12}$$

where t_{pg} is the delay of the 1-bit propagate/generate gates, t_{AO} is the delay of the AND-OR gate in the gray cell, and t_{xor} is the delay of the final sum XOR. Such a delay estimate is only qualitative because it does not account for fanout or sizing.

Often, using noninverting gates leads to more stages of logic than are necessary. Figure 11.16(b) shows how to alternate two types of inverting stages on alternate rows of the group PG network to remove extraneous inverters. For best performance, $G_{k-1:j}$ should drive the inner transistor in the series stack. You can also reduce the number of stages by using higher-valency cells, as shown in Figure 11.16(c) for a valency-4 black cell.

11.2.2.4 Manchester Carry Chain Adder *This section is available in the online Web Enhanced chapter at* `www.cmosvlsi.com`.

11.2.2.5 Carry-Skip Adder The critical path of CPAs considered so far involves a gate or transistor for each bit of the adder, which can be slow for large adders. The *carry-skip* (also called *carry-bypass*) adder, first proposed by Charles Babbage in the nineteenth century and used for many years in mechanical calculators, shortens the critical path by computing the group propagate signals for each carry chain and using this to skip over long carry ripples [Morgan59, Lehman61]. Figure 11.17 shows a carry skip adder built from 4-bit groups. The rectangles compute the bitwise propagate and generate signals (as in Figure 11.15), and also contain a 4-input AND gate for the propagate signal of the 4-bit group. The skip multiplexer selects the group carry-in if the group propagate is true or the ripple adder carry-out otherwise.

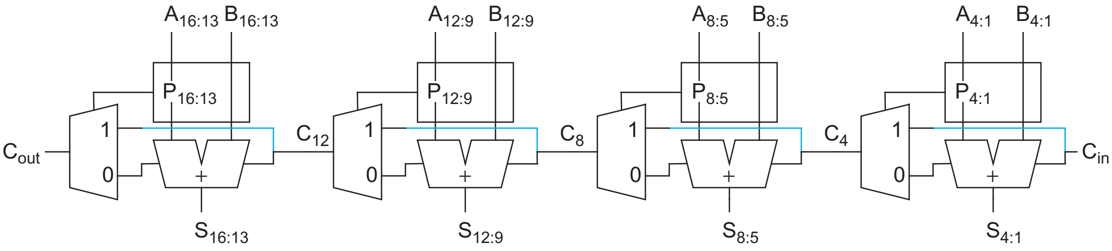


FIGURE 11.17 Carry-skip adder

The critical path through Figure 11.17 begins with generating a carry from bit 1, and then propagating it through the remainder of the adder. The carry must ripple through the next three bits, but then may skip across the next two 4-bit blocks. Finally, it must ripple through the final 4-bit block to produce the sums. This is illustrated in Figure 11.18. The 4-bit ripple chains at the top of the diagram determine if each group generates a carry. The carry skip chain in the middle of the diagram skips across 4-bit blocks. Finally, the 4-bit ripple chains with the blue lines represent *the same adders* that can produce a carry-out when a carry-in is bypassed to them. Note that the final AND-OR and column 16 are not strictly necessary because C_{out} can be computed in parallel with the sum XORs using EQ (11.11).

The critical path of the adder from Figures 11.17 and 11.18 involves the initial PG logic producing a carry out of bit 1, three AND-OR gates rippling it to bit 4, three multiplexers bypassing it to C_{12}, 3 AND-OR gates rippling through bit 15, and a final XOR to produce S_{16}. The multiplexer is an AND22-OR function, so it is slightly slower than the AND-OR function. In general, an N-bit carry-skip adder using k n-bit groups ($N = n \times k$) has a delay of

$$t_{\text{skip}} = t_{pg} + 2(n-1)t_{AO} + (k-1)t_{\text{mux}} + t_{\text{xor}} \tag{11.13}$$

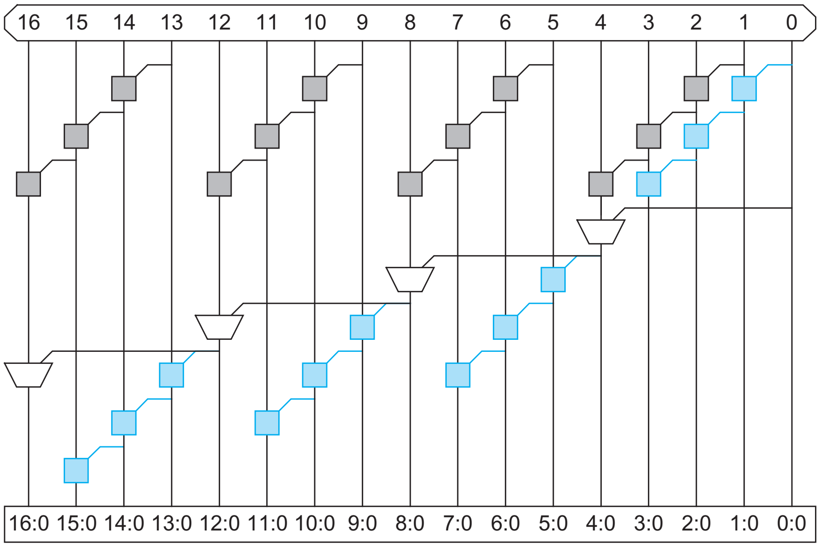


FIGURE 11.18 Carry-skip adder PG network

This critical path depends on the length of the first and last group and the number of groups. In the more significant bits of the network, the ripple results are available early. Thus, the critical path could be shortened by using shorter groups at the beginning and end and longer groups in the middle. Figure 11.19 shows such a PG network using groups of length [2, 3, 4, 4, 3], as opposed to [4, 4, 4, 4], which saves two levels of logic in a 16-bit adder.

The hardware cost of a carry-skip adder is equal to that of a simple carry-ripple adder plus k multiplexers and k n-input AND gates. It is attractive when ripple-carry adders are too slow, but the hardware cost must still be kept low. For long adders, you could use a multilevel skip approach to skip across the skips. A great deal of research has gone into choosing the best group size and number of levels [Majerski67, Oklobdzija85, Guyot87, Chan90, Kantabutra91], although now, parallel prefix adders are generally used for long adders instead.

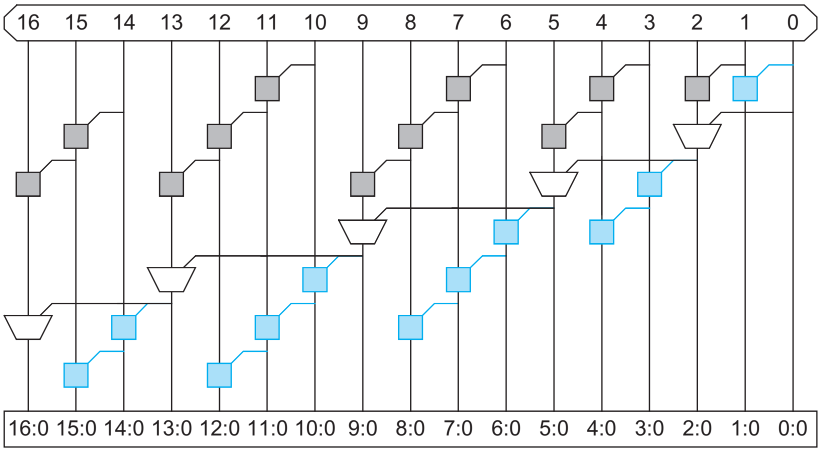


FIGURE 11.19 Variable group size carry-skip adder PG network

It might be tempting to replace each skip multiplexer in Figures 11.17 and 11.18 with an AND-OR gate combining the carry-out of the *n*-bit adder or the group carry-in and group propagate. Indeed, this works for domino-carry skip adders in which the carry out is precharged each cycle; it also works for carry-lookahead adders and carry-select adders covered in the subsequent section. However, it introduces a sneaky long critical path into an ordinary carry-skip adder. Imagine summing 111…111 + 000…000 + C_{in}. All of the group propagate signals are true. If $C_{\text{in}} = 1$, every 4-bit block will produce a carry-out. When C_{in} falls, the falling carry signal must ripple through all *N* bits because of the path through the carry out of each *n*-bit adder. Domino-carry skip adders avoid this path because all of the carries are forced low during precharge, so they can use AND-OR gates.

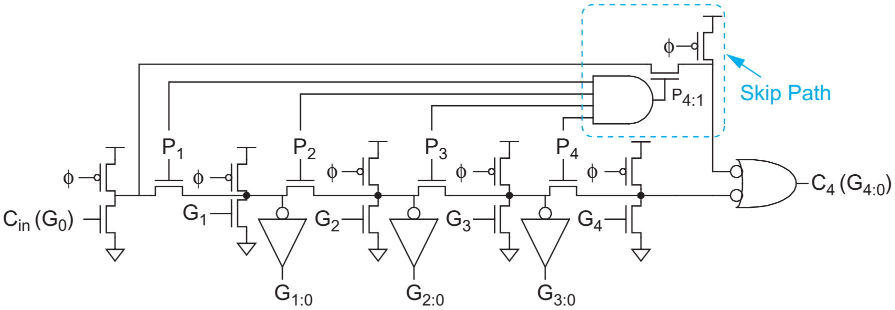


FIGURE 11.20 Carry-skip adder Manchester stage

Figure 11.20 shows how a Manchester carry chain from Section 11.2.2.4 can be modified to perform carry skip [Chan90]. A valency-5 chain is used to skip across groups of 4 bits at a time.

11.2.2.6 Carry-Lookahead Adder The *carry-lookahead adder* (CLA) [Weinberger58] is similar to the carry-skip adder, but computes group generate signals as well as group propagate signals to avoid waiting for a ripple to determine if the first group generates a carry. Such an adder is shown in Figure 11.21 and its PG network is shown in Figure 11.22 using valency-4 black cells to compute 4-bit group PG signals.

In general, a CLA using *k* groups of *n* bits each has a delay of

$$t_{\text{cla}} = t_{pg} + t_{pg(n)} + \left[\left(n-1\right)+\left(k-1\right)\right] t_{AO} + t_{\text{xor}} \tag{11.14}$$

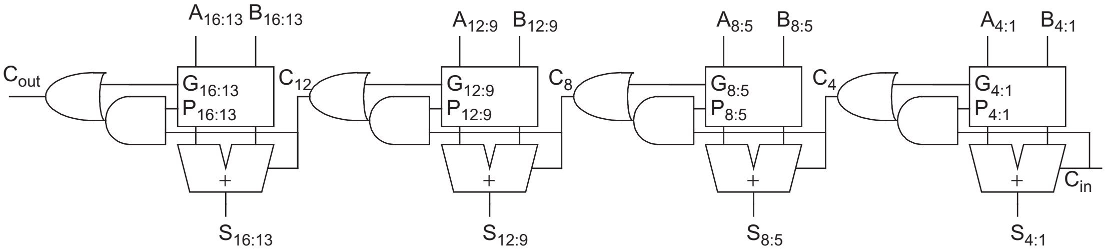


FIGURE 11.21 Carry-lookahead adder

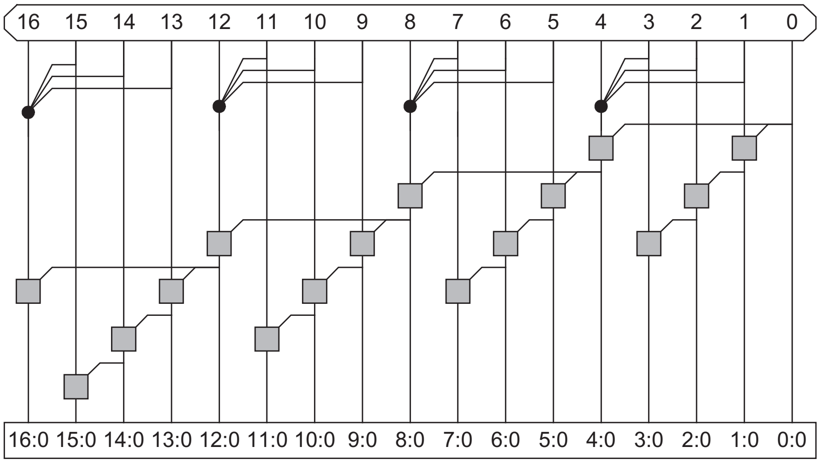


FIGURE 11.22 Carry-lookahead adder group PG network

where $t_{pg(n)}$ is the delay of the AND-OR-AND-OR-…-AND-OR gate computing the valency-n generate signal. This is no better than the variable-length carry-skip adder in Figure 11.19 and requires the extra n-bit generate gate, so the simple CLA is seldom a good design choice. However, it forms the basis for understanding faster adders presented in the subsequent sections.

CLAs often use higher-valency cells to reduce the delay of the n-bit additions by computing the carries in parallel. Figure 11.23 shows such a CLA in which the 4-bit adders are built using Manchester carry chains or multiple static gates operating in parallel.

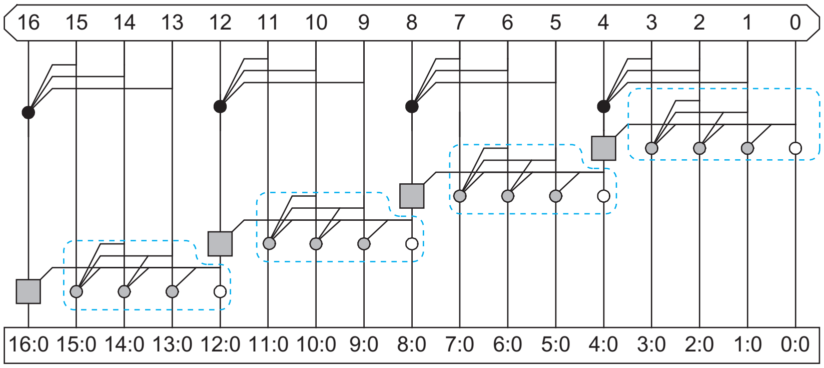


FIGURE 11.23 Improved CLA group PG network

11.2.2.7 Carry-Select, Carry-Increment, and Conditional-Sum Adders The critical path of the carry-skip and carry-lookahead adders involves calculating the carry into each n-bit group, and then calculating the sums for each bit within the group based on the carry-in. A standard logic design technique to accelerate the critical path is to precompute the outputs for both possible inputs, and then use a multiplexer to select between the two output choices. The *carry-select adder* [Bedrij62] shown in Figure 11.24 does this with a pair of

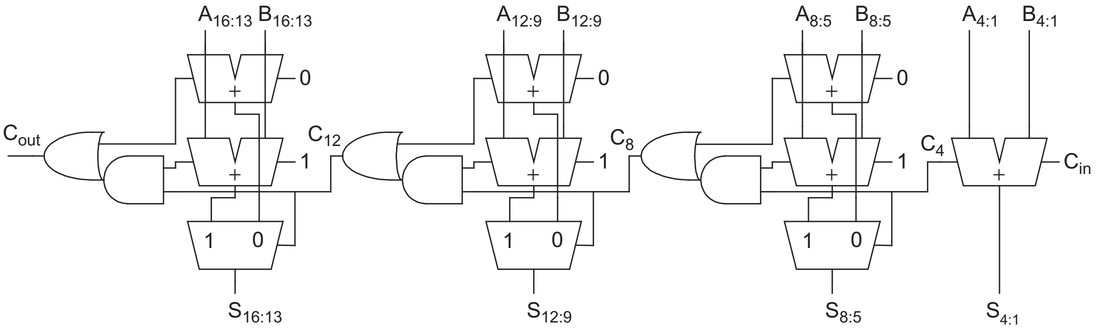


FIGURE 11.24 Carry-select adder

n-bit adders in each group. One adder calculates the sums assuming a carry-in of 0 while the other calculates the sums assuming a carry-in of 1. The actual carry triggers a multiplexer that chooses the appropriate sum. The critical path delay is

$$t_{\text{select}} = t_{pg} + \left[n + (k-2)\right] t_{AO} + t_{\text{mux}} \tag{11.15}$$

The two *n*-bit adders are redundant in that both contain the initial PG logic and final sum XOR. [Tyagi93] reduces the size by factoring out the common logic and simplifying the multiplexer to a gray cell, as shown in Figure 11.25. This is sometimes called a *carry-increment* adder [Zimmermann96]. It uses a short ripple chain of black cells to compute the PG signals for bits within a group. The bits spanned by each group are annotated on the diagram. When the carry-out from the previous group becomes available, the final gray cells in each column determine the carry-out, which is true if the group generates a carry or if the group propagates a carry and the previous group generated a carry. The carry-increment adder has about twice as many cells in the PG network as a carry-ripple adder. The critical path delay is about the same as that of a carry-select adder because a mux and XOR are comparable, but the area is smaller.

$$t_{\text{increment}} = t_{pg} + \left[(n-1) + (k-1)\right] t_{AO} + t_{\text{xor}} \tag{11.16}$$

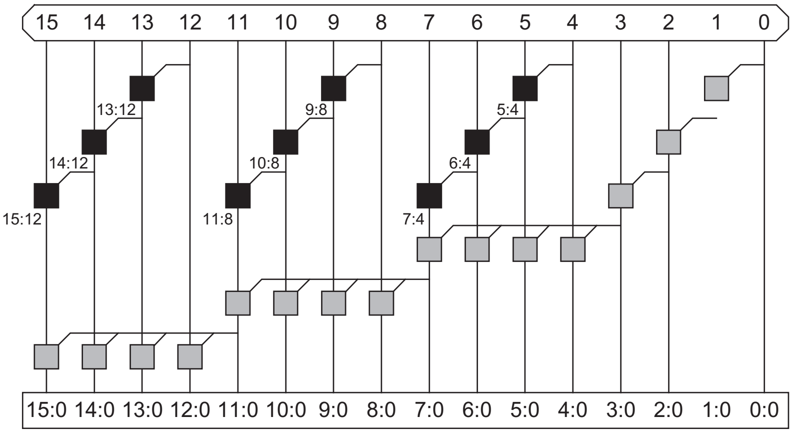


FIGURE 11.25 Carry-increment adder PG network

Of course, Manchester carry chains or higher-valency cells can be used to speed the ripple operation to produce the first group generate signal. In that case, the ripple delay is replaced by a group PG gate delay and the critical path becomes

$$t_{\text{increment}} = t_{pg} + t_{pg(n)} + \left[k-1\right] t_{AO} + t_{\text{xor}} \tag{11.17}$$

As with the carry-skip adder, the carry chains for the more significant bits complete early. Again, we can use variable-length groups to take advantage of the extra time, as shown in Figure 11.26(a). With such a variable group size, the delay reduces to

$$t_{\text{increment}} \approx t_{pg} + \sqrt{2N}\, t_{AO} + t_{\text{xor}} \tag{11.18}$$

The delay equations do not account for the fanout that each stage must drive. The fanouts in a variable-length group can become large enough to require buffering between stages. Figure 11.26(b) shows how buffers can be inserted to reduce the branching effort while not impeding the critical lookahead path; this is a useful technique in many other applications.

In wide adders, we can recursively apply multiple levels of carry-select or carry-increment. For example, a 64-bit carry-select adder can be built from four 16-bit carry-

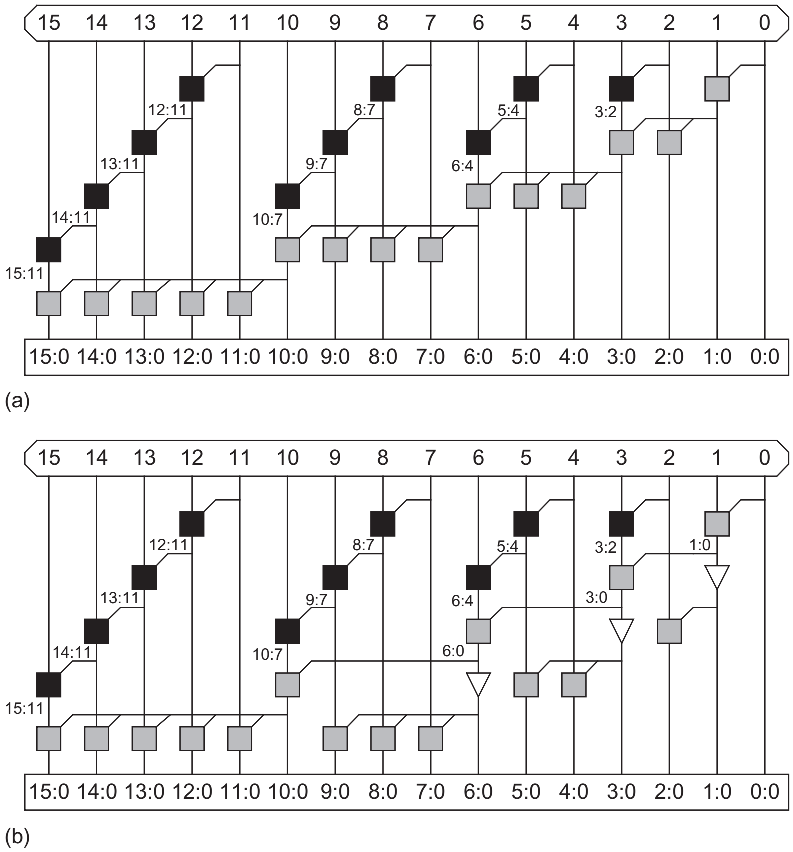
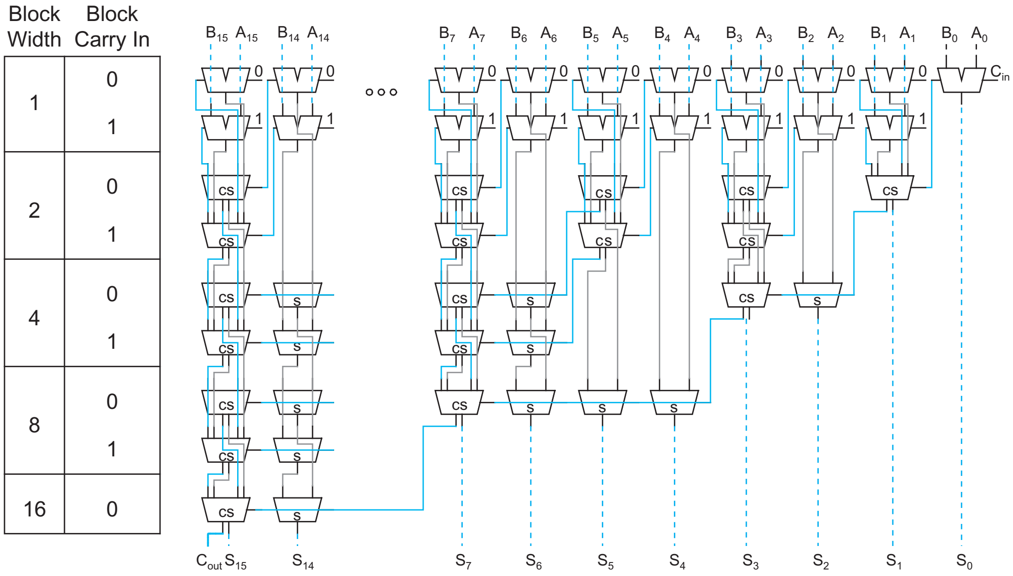


FIGURE 11.26 Variable-length carry-increment adder

select adders, each of which selects the carry-in to the next 16-bit group. Taking this to the limit, we obtain the *conditional-sum* adder [Sklansky60] that performs carry-select starting with groups of 1 bit and recursively doubling to $N/2$ bits. Figure 11.27 shows a 16-bit conditional-sum adder. In the first two rows, full adders compute the sum and carry-out for each bit assuming carries-in of 0 and 1, respectively. In the next two rows, multiplexer pairs select the sum and carry-out of the upper bit of each block of two, again assuming carries-in of 0 and 1. In the next two rows, multiplexers select the sum and carry-out of the upper two bits of each block of four, and so forth.



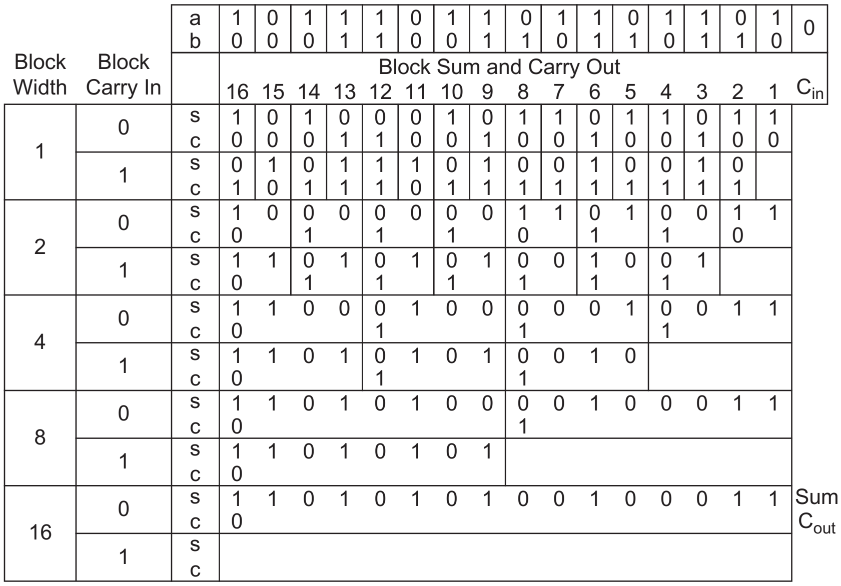
Block Width	Block Carry In
1	0
	1
2	0
	1
4	0
	1
8	0
	1
16	0

FIGURE 11.27 Conditional-sum adder

Figure 11.28 shows the operation of a conditional-sum adder in action for $N = 16$ with $C_{in} = 0$. In the block width 1 row, a pair of full adders compute the sum and carry-out for each column. One adder operates assuming the carry-in to that column is 0, while the other assumes it is 1. In the block width 2 row, the adder selects the sum for the upper half of each block (the even-numbered columns) based on the carry-out of the lower half. It also computes the carry-out of the pair of bits. Again, this is done twice, for both possibilities of carry-in to the block. In the block width 4 row, the adder again selects the sum for the upper half based on the carry-out of the lower half and finds the carry-out of the entire block. This process is repeated in subsequent rows until the 16-bit sum and the final carry-out are selected.

The conditional-sum adder involves nearly $2N$ full adders and $2N\log_2 N$ multiplexers. As with carry-select, the conditional-sum adder can be improved by factoring out the sum XORs and using AND-OR gates in place of multiplexers. This leads us to the Sklansky tree adder discussed in the next section.

11.2.2.8 Tree Adders For wide adders (roughly, $N > 16$ bits), the delay of carry-lookahead (or carry-skip or carry-select) adders becomes dominated by the delay of passing the carry through the lookahead stages. This delay can be reduced by looking ahead across the lookahead blocks [Weinberger58]. In general, you can construct a multilevel tree of look-ahead



		a	1	0	1	1	1	0	1	1	0	1	1	0	1	1	0	1	0
		b	0	0	0	1	1	0	0	1	1	0	1	1	0	1	1	0	
Block Width	Block Carry In		Block Sum and Carry Out																
			16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	C_{in}
1	0	s	1	0	1	0	0	0	1	0	1	1	0	1	1	0	1	1	
		c	0	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0	
	1	s	0	1	0	1	1	1	0	1	0	0	1	0	0	1	0		
		c	1	0	1	1	1	0	1	1	1	1	1	1	1	1	1		
2	0	s	1	0	0	0	0	0	0	0	1	1	0	1	0	0	1	1	
		c	0		1		1		1		0		1		1		0		
	1	s	1	1	0	1	0	1	0	1	0	0	1	0	0	1			
		c	0		1		1		1		1		1		1				
4	0	s	1	1	0	0	0	1	0	0	0	0	0	1	0	0	1	1	
		c	0				1				1				1				
	1	s	1	1	0	1	0	1	0	1	0	0	1	0					
		c	0				1				1								
8	0	s	1	1	0	1	0	1	0	0	0	0	1	0	0	0	1	1	
		c	0								1								
	1	s	1	1	0	1	0	1	0	1									
		c	0																
16	0	s	1	1	0	1	0	1	0	1	0	0	1	0	0	0	1	1	Sum
		c	0																C_{out}
	1	s																	
		c																	

FIGURE 11.28 Conditional-sum addition example

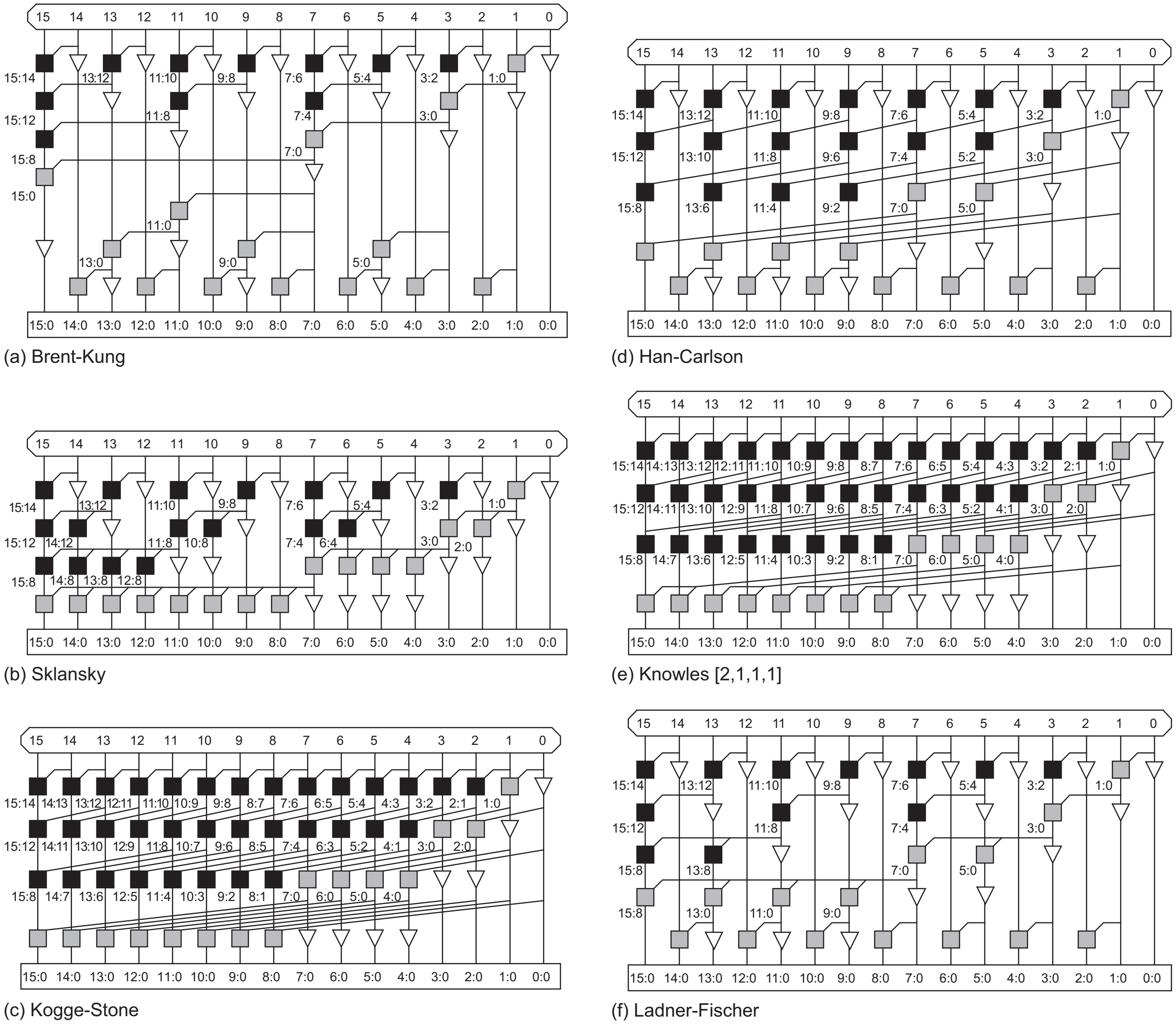
structures to achieve delay that grows with log N. Such adders are variously referred to as *tree* adders, *logarithmic* adders, *multilevel-lookahead* adders, *parallel-prefix* adders, or simply *lookahead* adders. The last name appears occasionally in the literature, but is not recommended because it does not distinguish whether multiple levels of lookahead are used.

There are many ways to build the lookahead tree that offer trade-offs among the number of stages of logic, the number of logic gates, the maximum fanout on each gate, and the amount of wiring between stages. Three fundamental trees are the Brent-Kung, Sklansky, and Kogge-Stone architectures. We begin by examining each in the valency-2 case that combines pairs of groups at each stage.

The *Brent-Kung* tree [Brent82] (Figure 11.29(a)) computes prefixes for 2-bit groups. These are used to find prefixes for 4-bit groups, which in turn are used to find prefixes for 8-bit groups, and so forth. The prefixes then fan back down to compute the carries-in to each bit. The tree requires $2\log_2 N - 1$ stages. The fanout is limited to 2 at each stage. The diagram shows buffers used to minimize the fanout and loading on the gates, but in practice, the buffers are generally omitted.

The *Sklansky* or *divide-and-conquer* tree [Sklansky60] (Figure 11.29(b)) reduces the delay to $\log_2 N$ stages by computing intermediate prefixes along with the large group prefixes. This comes at the expense of fanouts that double at each level: The gates fanout to [8, 4, 2, 1] other columns. These high fanouts cause poor performance on wide adders unless the high fanout gates are appropriately sized or the critical signals are buffered before being used for the intermediate prefixes. Transistor sizing can cut into the regularity of the layout because multiple sizes of each cell are required, although the larger gates can spread into adjacent columns. Note that the recursive doubling in the Sklansky tree is analogous to the conditional-sum adder of Figure 11.27. With appropriate buffering, the fanouts can be reduced to [8, 1, 1, 1], as explored in Exercise 11.7.

The *Kogge-Stone* tree [Kogge73] (Figure 11.29(c)) achieves both $\log_2 N$ stages and fanout of 2 at each stage. This comes at the cost of many long wires that must be routed between stages. The tree also contains more PG cells; while this may not impact the area if


(a) Brent-Kung

(d) Han-Carlson

(b) Sklansky

(e) Knowles [2,1,1,1]

(c) Kogge-Stone

(f) Ladner-Fischer

FIGURE 11.29 Tree adder PG networks

the adder layout is on a regular grid, it will increase power consumption. Despite these costs, the Kogge-Stone tree is widely used in high-performance 32-bit and 64-bit adders.

In summary, a Sklansky or Kogge-Stone tree adder reduces the critical path to

$$t_{\text{tree}} \approx t_{pg} + \left\lceil \log_2 N \right\rceil t_{AO} + t_{\text{xor}} \tag{11.19}$$

An ideal tree adder would have $\log_2 N$ levels of logic, fanout never exceeding 2, and no more than 1 wiring track ($G_{i:j}$ and $P_{i:j}$ bundle) between each row. The basic tree architectures represent cases that approach the ideal, but each differ in one respect. Brent-Kung has too many logic levels. Sklansky has too much fanout. And Kogge-Stone has too many wires. Between these three extremes, the Han-Carlson, Ladner-Fischer, and Knowles trees fill out the design space with different compromises between number of stages, fanout, and wire count.

The *Han-Carlson* trees [Han87] are a family of networks between Kogge-Stone and Brent-Kung. Figure 11.29(d) shows such a tree that performs Kogge-Stone on the odd-numbered bits, and then uses one more stage to ripple into the even positions.

The *Knowles* trees [Knowles01] are a family of networks between Kogge-Stone and Sklansky. All of these trees have $\log_2 N$ stages, but differ in the fanout and number of wires. If we say that 16-bit Kogge-Stone and Sklansky adders drive fanouts of [1, 1, 1, 1] and [8, 4, 2, 1] other columns, respectively, the Knowles networks lie between these extremes. For example, Figure 11.29(e) shows a [2, 1, 1, 1] Knowles tree that halves the number of wires in the final track at the expense of doubling the load on those wires.

The *Ladner-Fischer* trees [Ladner80] are a family of networks between Sklansky and Brent-Kung. Figure 11.29(f) is similar to Sklansky, but computes prefixes for the odd-numbered bits and again uses one more stage to ripple into the even positions. Cells at high-fanout nodes must still be sized or ganged appropriately to achieve good speed. Note that some authors use Ladner-Fischer synonymously with Sklansky.

An advantage of the Brent-Kung network and those related to it (Han-Carlson and the Ladner-Fischer network with the extra row) is that for any given row, there is never more than one cell in each pair of columns. These networks have low gate count. Moreover, their layout may be only half as wide, reducing the length of the horizontal wires spanning the adder. This reduces the wire capacitance, which may be a major component of delay in 64-bit and larger adders [Huang00].

Figure 11.30 shows a 3-dimensional taxonomy of the tree adders [Harris03]. If we let $L = \log_2 N$, we can describe each tree with three integers (l, f, t) in the range $[0, L - 1]$. The integers specify the following:

- Logic Levels: $L + l$
- Fanout: $2^f + 1$
- Wiring Tracks: 2^t

The tree adders lie on the plane $l + f + t = L - 1$. 16-bit Brent-Kung, Sklansky, and Kogge-Stone represent vertices of the cube (3, 0, 0), (0, 3, 0) and (0, 0, 3), respectively. Han-Carlson, Ladner-Fischer, and Knowles lie along the diagonals.

11.2.2.9 Higher-Valency Tree Adders Any of the trees described so far can combine more than two groups at each stage [Beaumont-Smith01]. The number of groups combined in each gate is called the *valency* or *radix* of the cell. For example, Figure 11.31 shows 27-bit valency-3 Brent-Kung, Sklansky, Kogge-Stone, and Han-Carlson trees. The rounded boxes mark valency-3 carry chains (that could be constructed using a Manchester carry chain, multiple-output domino gate, or several discrete gates). The trapezoids mark carry-increment operations. The higher-valency designs use fewer stages of logic, but each stage has greater delay. This tends to be a poor trade-off in static CMOS circuits because the stage efforts become much larger than 4, but is good in domino because the logical efforts are much smaller so fewer stages are necessary.

Nodes with large fanouts or long wires can use buffers. The prefix trees can also be internally pipelined for extremely high-throughput operation. Some higher-valency designs combine the initial PG stage with the first level of PG merge. For example, the

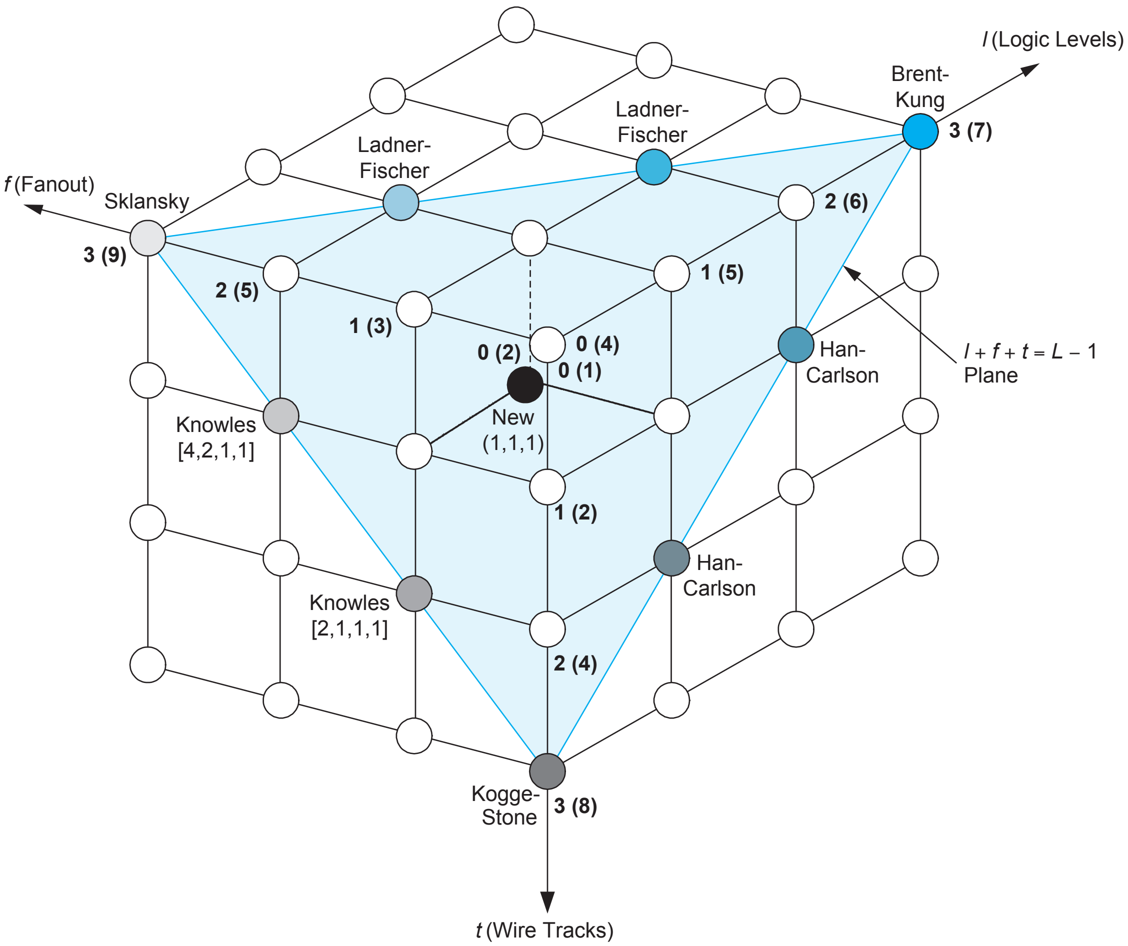


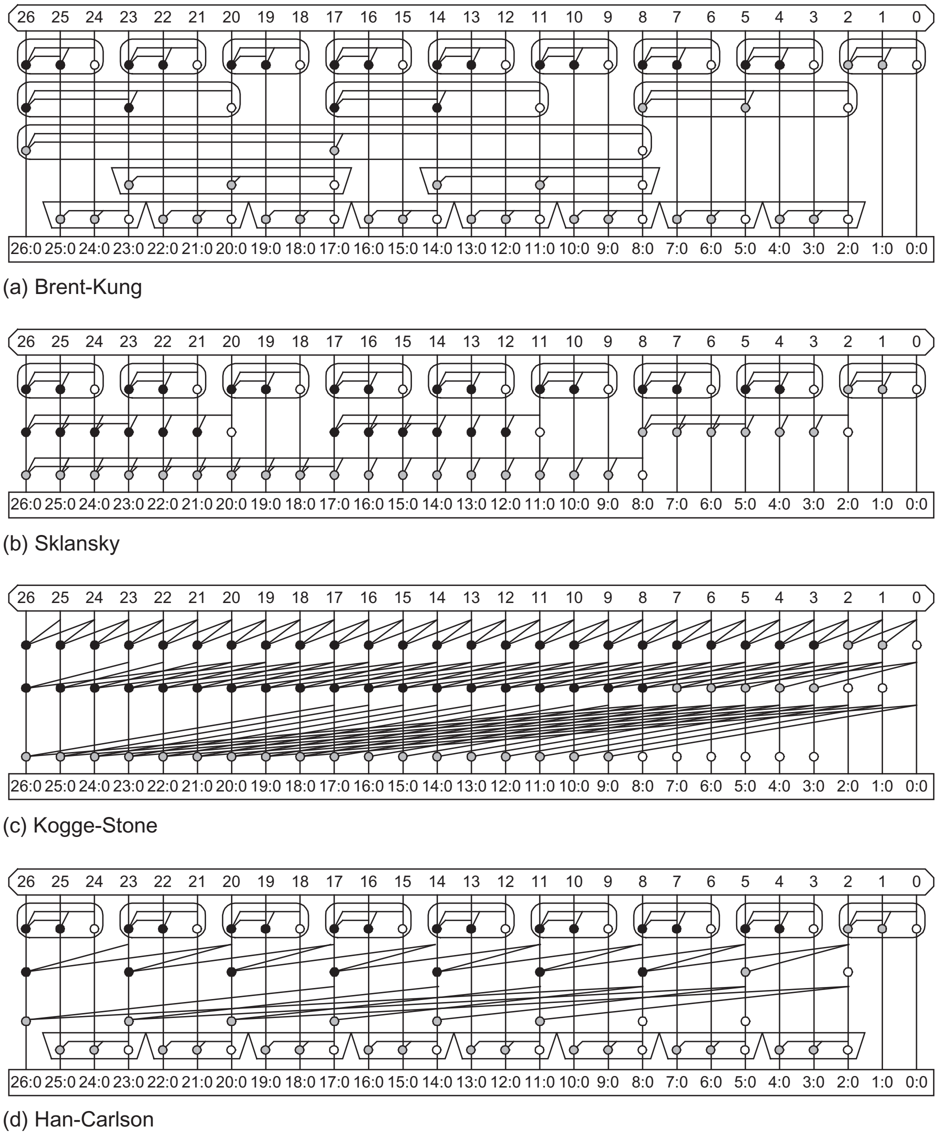
FIGURE 11.30 Taxonomy of prefix networks

Ling adder described in Section 11.2.2.11 computes generate and propagate for up to 4-bit groups from the primary inputs in a single stage.

Higher valency (v) adders can still be described in a 3-dimensional taxonomy with $L = \log_v N$ and $l + f + t = L - 1$. There are $L + l$ logic levels, a maximum fanout of $(v-1)v^f + 1$, and $(v-1)v^t$ wiring tracks at the worst level.

11.2.2.10 Sparse Tree Adders Building a prefix tree to compute carries in to every bit is expensive in terms of power. An alternative is to only compute carries into short groups (e.g., $s = 2, 4, 8$, or 16 bits). Meanwhile, pairs of s-bit adders precompute the sums assuming both carries-in of 0 and 1 to each group. A multiplexer selects the correct sum for each group based on the carries from the prefix tree. The group length can be balanced such that the carry-in and precomputed sums become available at about the same time. Such a hybrid between a prefix adder and a carry select adder is called a *sparse tree*. s is the *sparseness* of the tree.

The *spanning-tree adder* [Lynch92] is a sparse tree adder based on a higher-valency Brent-Kung tree of Figure 11.31(a). Figure 11.32 shows a simple valency-3 version that



(a) Brent-Kung

(b) Sklansky

(c) Kogge-Stone

(d) Han-Carlson

FIGURE 11.31 Higher-valency tree adders

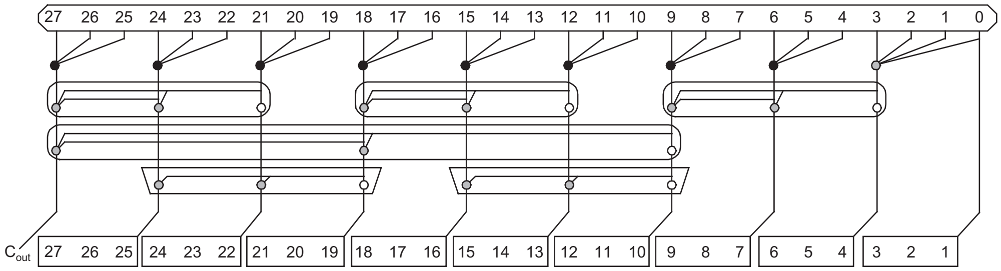


FIGURE 11.32 Valency-3 Brent-Kung sparse tree adder with $s = 3$

precomputes sums for $s = 3$-bit groups and saves one logic level by selecting the output based on the carries into each group. The carry-out (C_{out}) is explicitly shown. Note that the least significant group requires a valency-4 gray cell to compute $G_{3:0}$, the carry-in to the second select block.

[Lynch92] describes a 56-bit spanning-tree design from the AMD AM29050 floating-point unit using valency-4 stages and 8-bit carry select groups. [Kantabutra93] and [Blackburn96] describe optimizing the spanning-tree adder by using variable-length carry-select stages and appropriately selecting transistor sizes.

A carry-select box spanning bits $i \ldots j$ is shown in Figure 11.33(a). It uses short carry-ripple adders to precompute the sums assuming carry-in of 0 and 1 to the group, and then selects between them with a multiplexer, as shown in Figure 11.33(b). The adders can be simplified somewhat because the carry-ins are constant, as shown in Figure 11.33(c) for a 4-bit group.

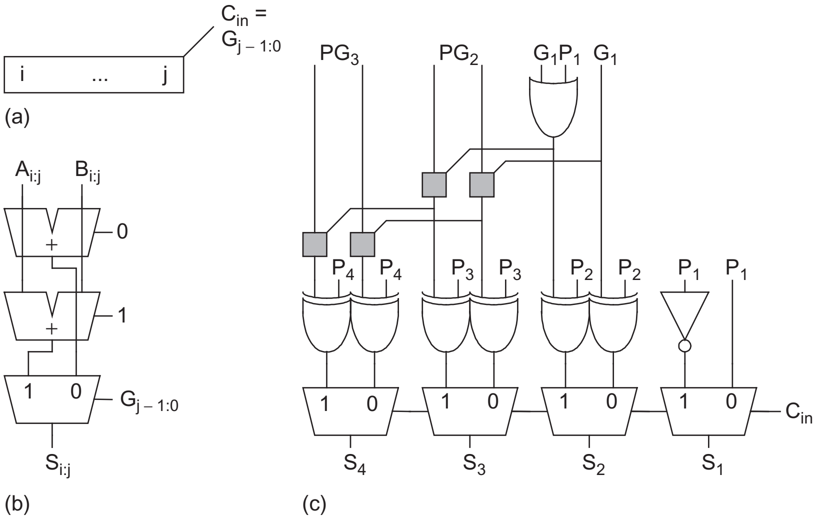


FIGURE 11.33 Carry-select implementation

[Mathew03] describes a 32-bit *sparse-tree adder* using a valency-2 tree similar to Sklansky to compute only the carries into each 4-bit group, as shown in Figure 11.34. This reduces the gate count and power consumption in the tree. The tree can also be viewed as a (2, 2, 0) Ladner-Fischer tree with the final two tree levels and XOR replaced by the select multiplexer. The adder assumes the carry-in is 0 and does not produce a carry-out, saving one input to the least-significant gray box and eliminating the prefix logic in the four most significant columns.

These sparse tree approaches are widely used in high-performance 32–64-bit higher-valency adders because they offer the small number of logic levels of higher-valency trees while reducing the gate count and power consumption in the tree. Figure 11.35 shows a 27-bit valency-3 Kogge-Stone design with carry-select on 3-bit groups. Observe how the number of gates in the tree is reduced threefold. Moreover, because the number of wires is also reduced, the extra area can be used for shielding to reduce path delay. This design can also be viewed as the Han-Carlson adder of Figure 11.31(d) with the last logic level replaced by a carry-select multiplexer.

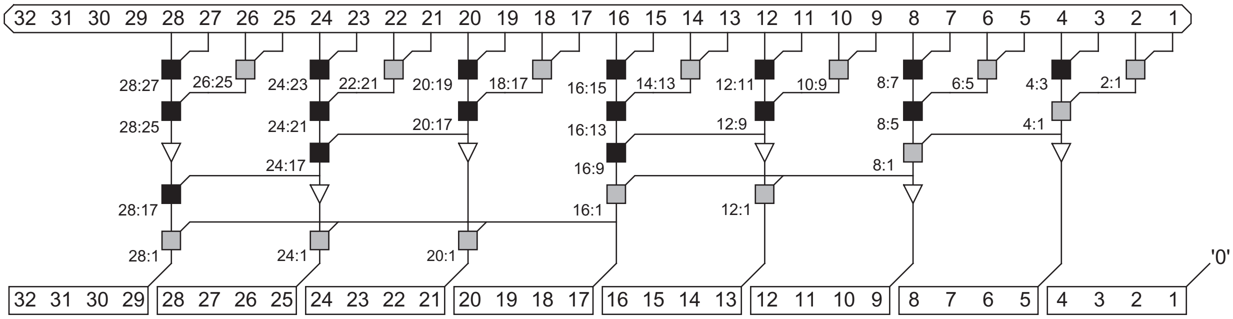


FIGURE 11.34 Intel valency-2 Sklansky sparse tree adder with $s = 4$

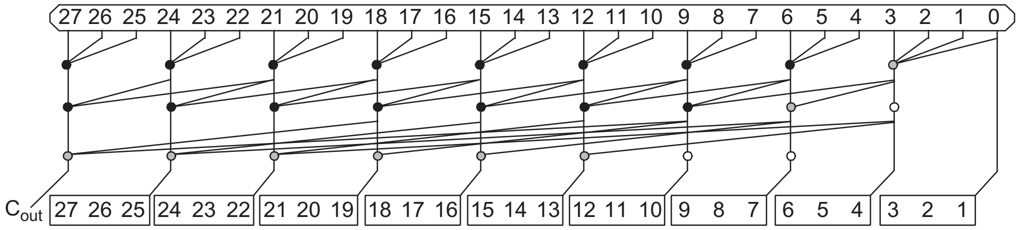


FIGURE 11.35 Valency-3 Kogge-Stone sparse tree adder with $s = 3$

Sparse trees reduce the costly part of the prefix tree. For Kogge-Stone architectures, they reduce the number of wires required by a factor of s. For Sklansky architectures, they reduce the fanout by s. For Brent-Kung architectures, they eliminate the last $\log_v s$ logic levels. In effect, they can move an adder toward the origin in the (l, f, t) design space. These benefits come at the cost of a fanout of s to the final select multiplexer, and of area and power to precompute the sums.

11.2.2.11 Ling Adders

Ling discovered a technique to remove one series transistor from the critical group generate path through an adder at the expense of another XOR gate in the sum precomputation [Ling81, Doran88, Bewick94]. The technique depends on using $\overline{K}$ in place of P in the prefix network, and on the observation that $G_i\overline{K_i} = (A_iB_i)(A_i + B_i) = G_i$.

Define a *pseudogenerate* (sometimes called *pseudo-carry*) signal $H_{i:j} = G_i + G_{i-1:j}$. This is simpler than $G_{i:j} = G_i + P_iG_{i-1:j}$. $G_{i:j}$ can be obtained later from $H_{i:j}$ with an AND operation when it is needed:

$$\overline{K}_i H_{i:j} = \overline{K}_i G_i + \overline{K}_i G_{i-1:j} = G_i + \overline{K}_i G_{i-1:j} = G_{i:j} \tag{11.20}$$

The advantage of pseudogenerate signals over regular generate is that the first row in the prefix network is easier to compute.

Also define a *pseudopropagate* signal I that is simply a shifted version of propagate: $I_{i:j} = \overline{K}_{i-1:j-1}$. Group pseudogenerate and pseudopropagate signals are combined using the same black or gray cells as ordinary group generate and propagate signals, as you may show in Exercise 11.11.

$$\begin{aligned} H_{i:j} &= H_{i:k} + I_{i:k}H_{k-1:j} \\ I_{i:j} &= I_{i:k}I_{k-1:j} \end{aligned} \tag{11.21}$$

The true group generate signals are formed from the pseudogenerates using EQ (11.20). These signals can be used to compute the sums with the usual XOR: $S_i = P_i \oplus G_{i-1:0} = P_i \oplus (\overline{K}_{i-1} H_{i-1:0})$. To avoid introducing an AND gate back onto the critical path, we expand S_i in terms of $H_{i-1:0}$

$$S_i = H_{i-1:0}\left[P_i \oplus \overline{K}_{i-1}\right] + \overline{H}_{i-1:0}\left[P_i\right] \qquad \textbf{(11.22)}$$

Thus, sum selection can be performed with a multiplexer choosing either $P_i \oplus \overline{K}_{i-1}$ or P_i based on $H_{i-1:0}$.

The Ling adder technique can be used with any form of adder that uses black and gray cells in a prefix network. It works with any valency and for both domino and static designs. The initial PG stage and the first levels of the prefix network are replaced by a cell that computes the group H and I signals directly. The middle of the prefix network is identical to an ordinary prefix adder but operates on H and I instead of G and P. The sum-selection logic uses the multiplexer from EQ (11.22) rather than an XOR. In sparse trees, the sum out of s-bit blocks is selected directly based on the H signals.

For a valency-v adder, the Ling technique converts a generate gate with v series nMOS transistors and v series pMOS transistors to a pseudogenerate gate with $v - 1$ series nMOS but still v series pMOS. For example, in valency 2, the AOI gate becomes a NOR2 gate. This is not particularly helpful for static logic, but is beneficial for domino implementations because the series pMOS are eliminated and the nMOS stacks are shortened.

Another advantage of the Ling technique is that it allows the first level pseudogenerate and pseudopropagate signals to be computed directly from the A_i and B_i inputs rather than based on G_i and K_i gates. For example, Figure 11.36 compares static gates that compute $G_{2:1}$ and $H_{2:1}$ directly from $A_{2:1}$ and $B_{2:1}$. The H gate has one fewer series transistor and much less parasitic capacitance. $H_{3:1}$ can also be computed directly from $A_{3:1}$ and $B_{3:1}$ using the complex static CMOS gate shown in Figure 11.37(a) [Quach92]. Similarly, Figure 11.37(b) shows a compound domino gate that directly computes $H_{4:1}$ from A and B using only four series transistors rather than the five required for $G_{4:1}$ [Naffziger96, Naffziger98].

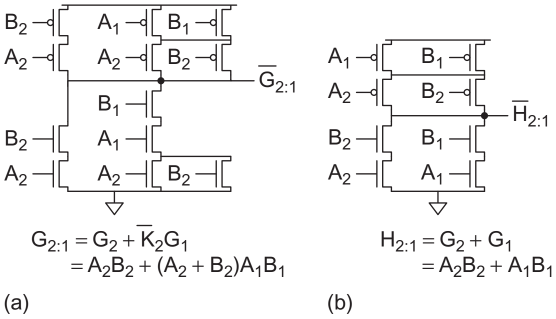


FIGURE 11.36 2-bit generate and pseudogenerate gates using primary inputs

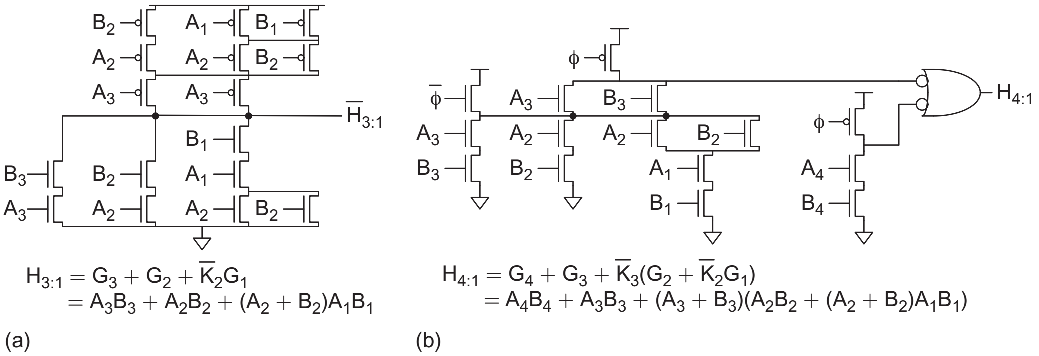

B2
A1
B1
A2
A2
B2
A3
A3
H3:1
B1
B3
B2
A1
A3
A2
A2
B2
H3:1 = G3 + G2 + K2G1
= A3B3 + A2B2 + (A2 + B2)A1B1
(a)
ϕ
ϕ
A3
B3
H4:1
A3
A2
A2
B2
ϕ
B3
B2
A1
A4
B1
B4
H4:1 = G4 + G3 + K3(G2 + K2G1)
= A4B4 + A3B3 + (A3 + B3)(A2B2 + (A2 + B2)A1B1)
(b)

FIGURE 11.37 3-bit and 4-bit pseudogenerate gates using primary inputs

[Jackson04] proposed applying the Ling method recursively to factor out the $\overline{K}$ signal elsewhere in the adder tree. [Burgess09] showed that this recursive Ling technique opens up a new design space containing faster and smaller adders.

11.2.2.12 An Aside on Domino Implementation Issues *This section is available in the online Web Enhanced chapter at* `www.cmosvlsi.com`.

11.2.2.13 Summary Having examined so many adders, you probably want to know which adder should be used in which application. Table 11.3 compares the various adder architectures that have been illustrated with valency-2 prefix networks. The category "logic levels" gives the number of AND-OR gates in the critical path, excluding the initial PG logic and final XOR. Of course, the delay depends on the fanout and wire loads as well as the number of logic levels. The category "cells" refers to the approximate number of gray and black cells in the network. Carry-lookahead is not shown because it uses higher-valency cells. Carry-select is also not shown because it is larger than carry-increment for the same performance.

In general, carry-ripple adders should be used when they meet timing constraints because they use the least energy and are easy to build. When faster adders are required, carry-increment and carry-skip architectures work well for 8–16 bit lengths. Hybrids combining these techniques are also popular. At word lengths of 32 and especially 64 bits, tree adders are distinctly faster.

TABLE 11.3 Comparison of adder architectures

Architecture	Classification	Logic Levels	Max Fanout	Tracks	Cells
Carry-Ripple		$N - 1$	1	1	N
Carry-Skip $(n = 4)$		$N/4 + 5$	2	1	$1.25N$
Carry-Increment $(n = 4)$		$N/4 + 2$	4	1	$2N$
Carry-Increment (variable group)		$\sqrt{2N}$	$\sqrt{2N}$	1	$2N$
Brent-Kung	$(L–1, 0, 0)$	$2\log_2 N - 1$	2	1	$2N$
Sklansky	$(0, L–1, 0)$	$\log_2 N$	$N/2 + 1$	1	$0.5\, N\log_2 N$
Kogge-Stone	$(0, 0, L - 1)$	$\log_2 N$	2	$N/2$	$N\log_2 N$
Han-Carlson	$(1, 0, L - 2)$	$\log_2 N + 1$	2	$N/4$	$0.5\, N\log_2 N$
Ladner Fischer $(l = 1)$	$(1, L - 2, 0)$	$\log_2 N + 1$	$N/4 + 1$	1	$0.25\, N\log_2 N$
Knowles [2,1,…,1]	$(0, 1, L - 2)$	$\log_2 N$	3	$N/4$	$N\log_2 N$

There is still debate about the best tree adder designs; the choice is influenced by power and delay constraints, by domino vs. static and custom vs. synthesis choices, and by the specific manufacturing process. Moreover, careful optimization of a particular architecture is more important than the choice of tree architecture.

When power is no concern, the fastest adders use domino or compound domino circuits [Naffziger96, Park00, Mathew03, Mathew05, Oklobdzija05, Zlatanovici09, Wijeratne07]. Several authors find that the Kogge-Stone architecture gives the lowest possible delay [Silberman98, Park00, Oklobdzija05, Zlatanovici09]. However, the large number of long wires consume significant energy and require large drivers for speed. Other architectures such as Sklansky [Mathew03] or Han-Carlson [Vangal02] offer better energy efficiency because they have fewer long wires. Valency-4 dynamic gates followed by inverters tend to give a slight speed advantage [Naffziger96, Park00, Zlatanovici09, Harris04, Oklobdzija05], but compound domino implementations using valency-2 dynamic gates followed by valency-2 HI-skew static gates are also used [Mathew03]. Sparse trees save energy in domino adders with little effect on performance [Naffziger96, Mathew03, Zlatanovici09]. The Ling optimization is not used universally, but several studies have found it to be beneficial [Quach92, Naffziger96, Zlatanovici09, Grad04]. The UltraSparc III used a dual-rail domino Kogge-Stone adder [Heald00]. The Itanium 2 and Hewlett Packard PA-RISC lines of 64-bit microprocessors used a dual-rail domino sparse tree Ling adder [Naffziger96, Fetzer02]. The 65 nm Pentium 4 uses a compound domino radix-2 Sklansky sparse tree [Wijeratne07]. A good 64-bit domino adder takes 7–9 FO4 delays and has an area of 4–12 Mλ^2 [Naffziger96, Zlatanovici09, Mathew05].

Power-constrained designs use static adders, which consume one third to one tenth the energy of dynamic adders and have a delay of about 13 FO4 [Oklobdzija05, Harris03, Zlatanovici09]. For example, the CELL processor floating point unit uses a valency-2 static Kogge-Stone adder [Oh06].

[Patil07] presents a comprehensive study of energy-delay design space for adders. The paper concludes that the Sklansky architecture is most energy efficient for any delay requirement because it avoids the large number of power-hungry wires in Kogge-Stone and the excessive number of logic levels in Brent-Kung. The high-fanout gates in the Sklansky tree are upsized to maintain a reasonable logical effort. Static adders are most efficient using valency-2 cells, which provide a stage effort of about 4. Domino adders are most efficient alternating valency-4 dynamic gates with static inverters. The sum precomputation logic in a static sparse tree adder costs more energy than it saves from the prefix network. In a domino adder, a sparseness of 2 does save energy because the sum precomputation can be performed with static gates. Figure 11.38 shows some results, finding that static adders are most energy-efficient for slow adders, while domino become better at high speed requirements and dual-rail domino Ling adders are preferable only for the very fastest and most energy-hungry adders. The very fast delays are achieved using a higher V_{DD} and lower V_t. [Zlatanovici09] explores the energy-delay space for 64-bit domino adders and came to the contradictory conclusion that Kogge-Stone is superior. Again, alternating valency-4 dynamic gates with static inverters and using a sparseness of 2 gave the best results, as shown in Figure 11.39. Other reasonable adders are almost as good in the energy-delay space, so there is not a compelling reason to choose one topology over another and the debate about the "best" adder will doubtlessly rage into the future.

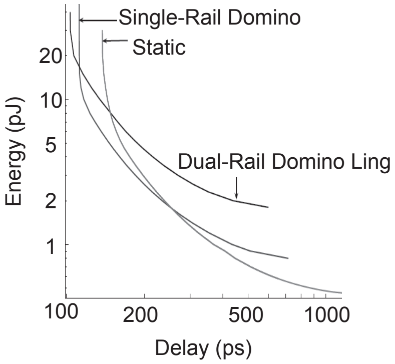


FIGURE 11.38 Energy-delay trade-offs for 90 nm 32-bit Sklansky static, domino, and dual-rail domino adders. FO4 inverter delay in this process at 1.0 V and nominal V_t is 31 ps. (© IEEE 2007.)

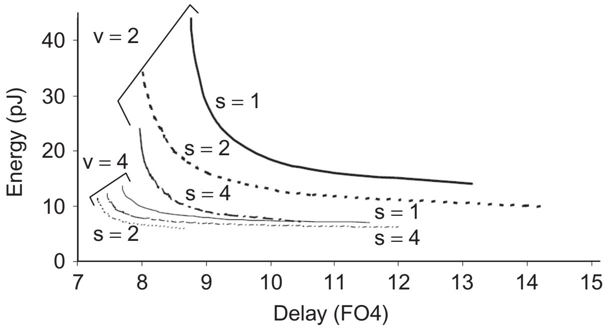


FIGURE 11.39 Energy-delay trade-offs for 90 nm 64-bit domino Kogge-Stone Ling adders as a function of valency (v) and sparseness (s). (© IEEE 2009.)

Good logic synthesis tools automatically map the "+" operator onto an appropriate adder to meet timing constraints while minimizing area. For example, the Synopsys DesignWare libraries contain carry-ripple adders, carry-select adders, carry-lookahead adders, and a variety of prefix adders. Figure 11.40 shows the results of synthesizing

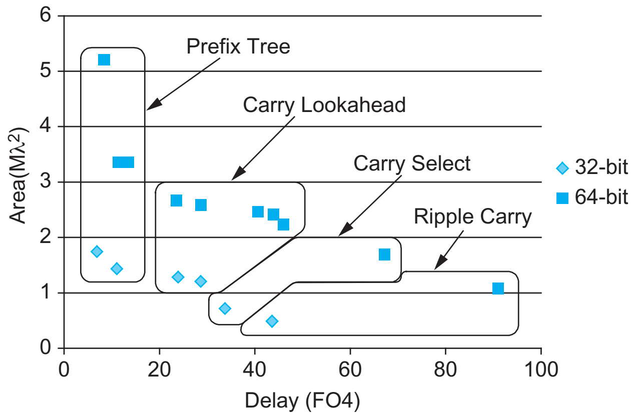


FIGURE 11.40 Area vs. delay of synthesized adders

32-bit and 64-bit adders under different timing constraints. As the latency decreases, synthesis selects more elaborate adders with greater area. The results are for a 0.18 μm commercial cell library with an FO4 inverter delay of 89 ps in the TTTT corner and the area includes estimated interconnect as well as gates. The fastest designs use tree adders and achieve implausibly fast prelayout delays of 7.0 and 8.5 FO4 for 32-bit and 64-bit adders, respectively, by creating nonuniform designs with side loads carefully buffered off the critical path. The carry-select adders achieve an interesting area/delay trade-off by using carry-ripple for the lower three-fourths of the bits and carry-select only on the upper fourth. The results will be somewhat slower when wire parasitics are included.

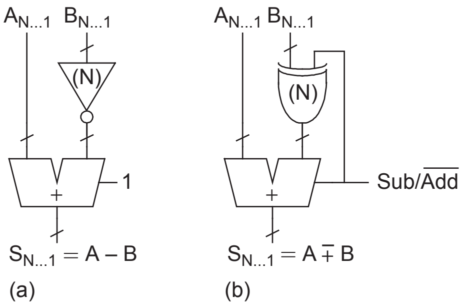


FIGURE 11.41 Subtracters

11.2.3 Subtraction

An N-bit subtracter uses the two's complement relationship

$$A - B = A + \overline{B} + 1 \tag{11.23}$$

This involves inverting one operand to an N-bit CPA and adding 1 via the carry input, as shown in Figure 11.41(a). An adder/subtracter uses XOR gates to conditionally invert B, as shown in Figure 11.41(b). In prefix adders, the XOR gates on the B inputs are sometimes merged into the bitwise PG circuitry.

11.2.4 Multiple-Input Addition

The most obvious method of adding k N-bit words is with $k - 1$ cascaded CPAs as illustrated in Figure 11.42(a) for 0001 + 0111 + 1101 + 0010. This approach consumes a large amount of hardware and is slow. A better technique is to note that a full adder sums three inputs of unit weight and produces a sum output of unit weight and a carry output of double weight. If N full adders are used in parallel, they can accept three N-bit input words $X_{N...1}$, $Y_{N...1}$, and $Z_{N...1}$, and produce two N-bit output words $S_{N...1}$ and $C_{N...1}$, satisfying $X + Y + Z = S + 2C$, as shown in Figure 11.42(b). The results correspond to the sums and carries-out of each adder. This is called *carry-save redundant format* because the carry outputs are preserved rather than propagated along the adder. The full adders in this application are sometimes called *[3:2] carry-save adder* (CSA) because they accept three inputs and produce two outputs in carry-save form. When the carry word C is shifted left by one position (because it has double weight) and added to the sum word S with an ordinary CPA, the result is $X + Y + Z$. Alternatively, a fourth input word can be added to the carry-save redundant result with another row of CSAs, again resulting in a carry-save redundant result. Such carry-save addition of four numbers is illustrated in Figure 11.42(c), where the underscores in the carry outputs serve as reminders that the carries must be shifted left one column on account of their greater weight.

The critical path through a [3:2] adder is for the sum computation, which involves one 3-input XOR, or two levels of XOR2. This is much faster than a CPA. In general, k

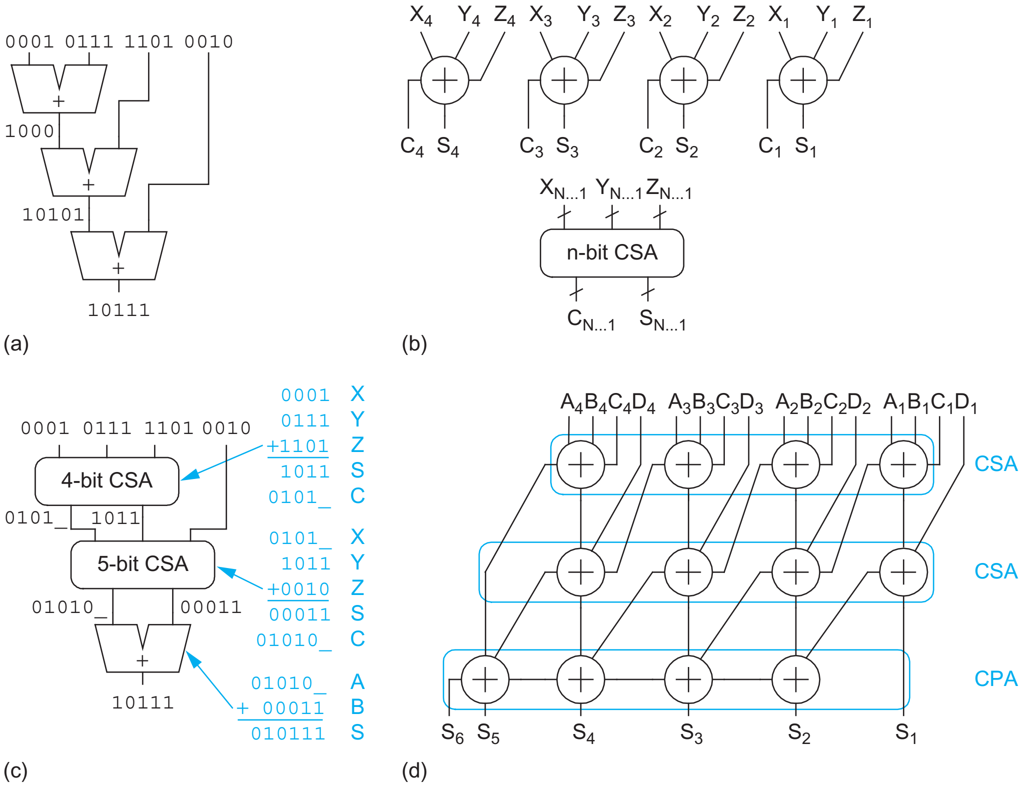


FIGURE 11.42 Multiple-input adders

numbers can be summed with $k - 2$ [3:2] CSAs and only one CPA. This approach will be exploited in Section 11.9 to add many partial products in a multiplier rapidly. The technique dates back to von Neumann's early computer [Burks46].

When one of the inputs to a CSA is a constant, the hardware can be reduced further. If a bit of the input is 0, the CSA column reduces to a half-adder. If the bit is 1, the CSA column simplifies to $S = \overline{A \oplus B}$ and $C = A + B$.

11.2.5 Flagged Prefix Adders

Sometimes it is necessary to compute either $A + B$, and then, depending on a late-arriving control signal, adding 1. Some applications include calculating $A + B \bmod 2^n{-}1$ for cryptography and Reed-Solomon coding, computing the absolute difference $|A - B|$, doing addition/subtraction of sign-magnitude numbers, and performing rounding in certain floating-point adders [Beaumont-Smith99]. A straightforward approach is to build two adders, provide a carry to one, and select between the results. [Burgess02] describes a clever alternative called a *flagged prefix adder* that uses much less hardware.

A flagged prefix adder receives A, B, and a control signal, *inc*, and computes $A + B +$ *inc*. Recall that an ordinary adder computes the prefixes $G_{i-1:0}$ as the carries into each column i, then computes the sum $S_i = P_i \oplus G_{i-1:0}$. In this situation, there is no C_{in} and hence column 0 is omitted; $G_{i-1:1}$ is used instead. The goal of the flagged prefix adder is to adjust these carries when *inc* is asserted. A flagged prefix adder instead uses

$G'_{i-1:1} = G_{i-1:1} + P_{i-1:1} \cdot inc$. Thus, if *inc* is true, it generates a carry into all of the low order bits whose group propagate signals are TRUE. The modified prefixes, $G'_{i-1:1}$, are called *flags*. The sums are computed in the same way with an XOR gate: $S_i = P_i \oplus G'_{i-1:1}$.

To produce these flags, the flagged prefix adder uses one more row of gray cells. This requires that the former bottom row of gray cells be converted to black cells to produce the group propagate signals. Figure 11.43 shows a flagged prefix Kogge-Stone adder. The new row, shown in blue, is appended to perform the late increment. Column 0 is eliminated because there is no C_{in}, but column 16 is provided because applications of flagged adders will need the generate and propagate signals spanning the entire *n* bits.

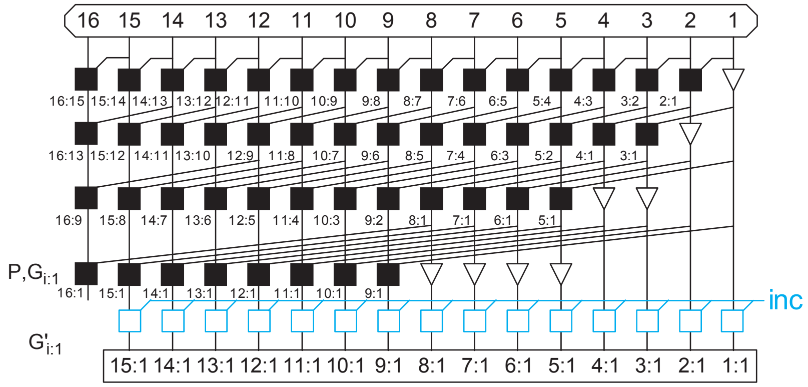


FIGURE 11.43 Flagged prefix Kogge-Stone adder

11.2.5.1 Modulo $2^n - 1$ Addition To compute $A + B \bmod 2^n - 1$ for unsigned operands, an adder should first compute $A + B$. If the sum is greater than or equal to $2^n - 1$, the result should be incremented and truncated back to *n* bits. $G_{n:1}$ is TRUE if the adder will overflow; i.e., the result is greater than $2^n - 1$. $P_{n:1}$ is TRUE if all columns propagate, which only occurs when the sum equals $2^n - 1$. Hence, modular addition can done with a flagged prefix adder using $inc = G_{n:1} + P_{n:1}$.

Compared to ordinary addition, modular addition requires one more row of black cells, an OR gate to compute *inc*, and a buffer to drive *inc* across all *n* bits.

11.2.5.2 Absolute Difference $|A - B|$ is called the *absolute difference* and is commonly used in applications such as video compression. The most straightforward approach is to compute both $A - B$ and $B - A$, then select the positive result. A more efficient technique is to compute $A + \overline{B}$ and look at the sign, indicated by $\overline{G}_{n:1}$. If the result is negative, it should be inverted to obtain $B - A$. If the result is positive, it should be incremented to obtain $A - B$.

All of these operations can be performed using a flagged prefix adder enhanced to invert the result conditionally. Modify the sum logic to calculate $S_i = (P_i \oplus inv) \oplus G'_{i-1:1}$. Choose $inv = \overline{G}_{n:1}$ and $inc = G_{n:1}$.

Compared to ordinary addition, absolute difference requires a bank of inverters to obtain $\overline{B}$, one more row of black cells, buffers to drive *inv* and *inc* across all *n* bits, and a row of XORs to invert the result conditionally. Note that $(P_i \oplus inv)$ can be precomputed so this does not affect the critical path.

11.2.5.3 Sign-Magnitude Arithmetic Addition of sign-magnitude numbers involves examining the signs of the operands. If the signs agree, the magnitudes are added and the sign is unchanged. If the signs differ, the absolute difference of the magnitudes must be computed. This can be done using the flagged carry adder described in the previous section. The sign of the result is sign(A) $\oplus$ $G_{n:1}$.

Subtraction is identical except that the sign of B is first flipped.

11.3 One/Zero Detectors

Detecting all ones or zeros on wide N-bit words requires large fan-in AND or NOR gates. Recall that by DeMorgan's law, AND, OR, NAND, and NOR are fundamentally the same operation except for possible inversions of the inputs and/or outputs. You can build a tree of AND gates, as shown in Figure 11.44(a). Here, alternate NAND and NOR gates have been used. The path has log N stages. In general, the minimum logical effort is achieved with a tree alternating NAND gates and inverters and the path logical effort is

$$G_{\text{and}}\left(N\right)=\left(\frac{4}{3}\right)^{\log_2 N} = N^{\log_2 \frac{4}{3}} = N^{0.415} \tag{11.24}$$

A rough estimate of the path delay driving a path electrical effort of H using static CMOS gates is

$$D \approx \left(\log_4 F\right) t_{FO4} = \left(\log_4 H + 0.415 \log_4 N\right) t_{FO4} \tag{11.25}$$

where t_{FO4} is the fanout-of-4 inverter delay.

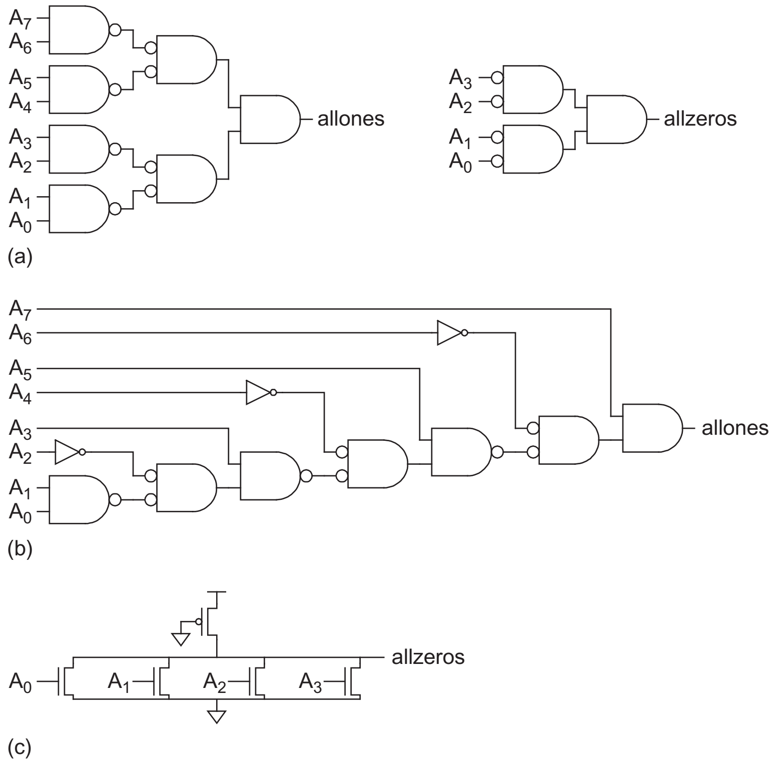


FIGURE 11.44 One/zero detectors

If the word being checked has a natural skew in the arrival time of the bits (such as at the output of a ripple adder), the designer might consider an asymmetric design that favors the late-arriving inputs, as shown in Figure 11.44(b). Here, the delay from the latest bit A_7 is a single gate.

Another fast detector uses a pseudo-nMOS or dynamic NOR structure to perform the “wired-OR,” as shown in Figure 11.44(c). This works well for words up to about 16 bits; for larger words, the gates can be split into 8–16-bit chunks to reduce the parasitic delay and avoid problems with subthreshold leakage.

11.4 Comparators

11.4.1 Magnitude Comparator

A *magnitude comparator* determines the larger of two binary numbers. To compare two unsigned numbers A and B, compute $B - A = B + \overline{A} + 1$. If there is a carry-out, $A \leq B$; otherwise, $A > B$. A zero detector indicates that the numbers are equal. Figure 11.45 shows a 4-bit unsigned comparator built from a carry-ripple adder and two’s complementer. The relative magnitude is determined from the carry-out (C) and zero (Z) signals according to Table 11.4. For wider inputs, any of the faster adder architectures can be used.

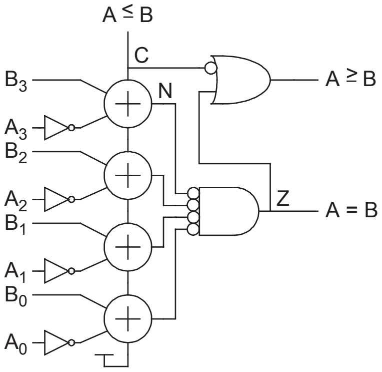


FIGURE 11.45
Unsigned magnitude comparator

Comparing signed two’s complement numbers is slightly more complicated because of the possibility of overflow when subtracting two numbers with different signs. Instead of simply examining the carry-out, we must determine if the result is negative (N, indicated by the most significant bit of the result) and if it overflows the range of possible signed numbers. The overflow signal V is true if the inputs had different signs (most significant bits) and the output sign is different from the sign of B. The actual sign of the difference $B - A$ is $S = N \oplus V$ because overflow flips the sign. If this corrected sign is negative ($S = 1$), we know $A > B$. Again, the other relations can be derived from the corrected sign and the Z signal.

TABLE 11.4 Magnitude comparison

Relation	Unsigned Comparison	Signed Comparison
$A = B$	Z	Z
$A \neq B$	$\overline{Z}$	$\overline{Z}$
$A < B$	$C \cdot \overline{Z}$	$\overline{S} \cdot \overline{Z}$
$A > B$	C	S
$A \leq B$	C	$\overline{S}$
$A \geq B$	$\overline{C} + Z$	$S + Z$

11.4.2 Equality Comparator

An *equality comparator* determines if ($A = B$). This can be done more simply and rapidly with XNOR gates and a ones detector, as shown in Figure 11.46.

11.4.3 $K = A + B$ Comparator

OPTIONAL

Sometimes it is necessary to determine if ($A + B = K$). For example, the sum-addressed memory [Heald98] described in Section 12.2.2.4 contains a decoder that must match against the sum of two numbers, such as a register base address and an immediate offset. Remarkably, this comparison can be done faster than computing $A + B$ because no carry propagation is necessary. The key is that if you know A and B, you also know what the carry into each bit must be if $K = A + B$ [Cortadella92]. Therefore, you only need to check adjacent pairs of bits to verify that the previous bit produces the carry required by the current bit, and then use a ones detector to check that the condition is true for all N pairs. Specifically, if $K = A + B$, Table 11.5 lists what the carry-in c_{i-1} must have been for this to be true and what the carry-out c_i will be for each bit position i.

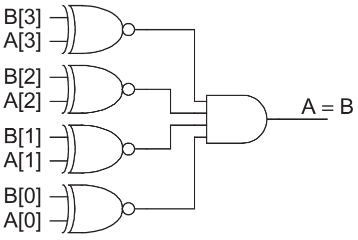


FIGURE 11.46 Equality comparator

TABLE 11.5 Required and generated carries if $K = A + B$

A_i	B_i	K_i	c_{i-1} (required)	c_i (produced)
0	0	0	0	0
0	0	1	1	0
0	1	0	1	1
0	1	1	0	0
1	0	0	1	1
1	0	1	0	0
1	1	0	0	1
1	1	1	1	1

From this table, you can see that the required c_{i-1} for bit i is

$$c_{i-1} = A_i \oplus B_i \oplus K_i \tag{11.26}$$

and the c_{i-1} produced by bit $i - 1$ is

$$c_{i-1} = \left(A_{i-1} \oplus B_{i-1}\right)\overline{K}_{i-1} + A_{i-1} \cdot B_{i-1} \tag{11.27}$$

Figure 11.47 shows one bitslice of a circuit to perform this operation. The XNOR gate is used to make sure that the required carry matches the produced carry at each bit position; then the AND gate checks that the condition is satisfied for all bits.

11.5 Counters

Two commonly used types of counters are *binary counter*s and *linear-feedback shift registers*. An N-bit binary counter sequences through 2^N outputs in binary order. Simple designs have a minimum cycle time that increases with N, but faster designs operate in constant time. An N-bit linear-feedback shift register sequences through up to $2^N - 1$ outputs in pseudo-random order. It has a short minimum cycle time independent of N, so it is useful for extremely fast counters as well as pseudo-random number generation.

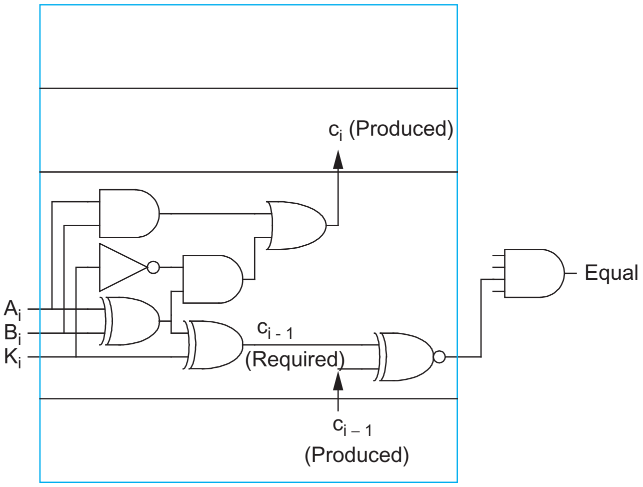


FIGURE 11.47 $A + B = K$ comparator

Some of the common features of counters include the following:

- *Resettable*: counter value is reset to 0 when *RESET* is asserted (essential for testing)
- *Loadable*: counter value is loaded with *N*-bit value when *LOAD* is asserted
- *Enabled*: counter counts only on clock cycles when *EN* is asserted
- *Reversible*: counter increments or decrements based on $\overline{UP}/DOWN$ input
- *Terminal Count: TC* output asserted when counter overflows (when counting up) or underflows (when counting down)

In general, divide-by-*M* counters ($M < 2^N$) can be built using an ordinary *N*-bit counter and circuitry to reset the counter upon reaching *M*. *M* can be a programmable input if an equality comparator is used. Alternatively, a loadable counter can be used to restart at $N - M$ whenever *TC* indicates that the counter overflowed.

11.5.1 Binary Counters

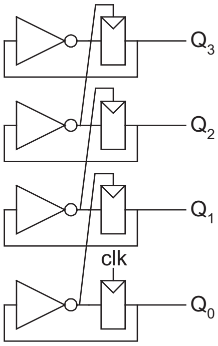


FIGURE 11.48 Asynchronous ripple-carry counter

The simplest binary counter is the *asynchronous ripple-carry counter*, as shown in Figure 11.48. It is composed of *N* registers connected in toggle configuration, where the falling transition of each register clocks the subsequent register. Therefore, the delay can be quite long. It has no reset signal, making it difficult to test. In general, asynchronous circuits introduce a whole assortment of problems, so the ripple-carry counter is shown mainly for historical interest and is not recommended for commercial designs.

A general *synchronous up/down counter* is shown in Figure 11.49(a). It uses a resettable register and full adder for each bit position. The cycle time is limited by the ripple-carry delay. While a faster adder could be used, the next section describes a better way to build fast counters. If only an *up counter* (also called an *incrementer*) is required, the full adder degenerates into a half adder, as shown in Figure 11.49(b). Including an input multiplexer allows the counter to load an initialization value. A clock enable is also often provided to each register for conditional counting. The terminal count (*TC*) output indicates that the counter has overflowed or underflowed. Figure 11.50 shows a fully featured resettable loadable enabled synchronous up/down counter.

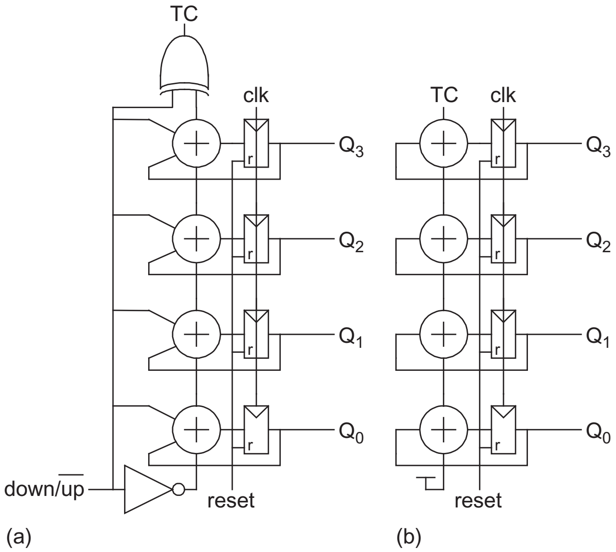


FIGURE 11.49 Synchronous counters

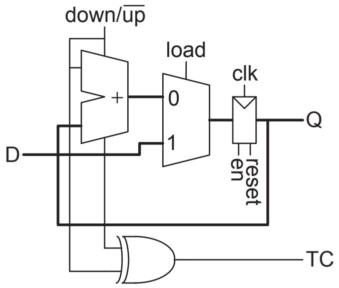


FIGURE 11.50 Synchronous up/down counter with reset, load, and enable

11.5.2 Fast Binary Counters

The speed of the counter in Figure 11.49 is limited by the adder. This can be overcome by dividing the counter into two or more segments [Ercegovac89]. For example, a 32-bit counter could be constructed from a 4-bit *prescalar* counter and a 28-bit counter, as shown in Figure 11.51. The *TC* output of the prescalar enables counting on the more significant segment. Now, the cycle time is limited only by the prescalar speed because the 28-bit adder has 2^4 cycles to produce a result. By using more segments, a counter of arbitrary length can run at the speed of a 1- or 2-bit counter.

Prescaling does not suffice for up/down counters because the more significant segment may have only a single cycle to respond when the counter changes direction. To solve this, a *shadow register* can be used on the more significant segments to hold the previous value that should be used when the direction changes [Stan98]. Figure 11.52 shows the more significant segment for a fast up/down counter. On reset (not shown in the figure), the *dir* register is set to 0, *Q* to 0, and *shadow* to −1. When $\overline{UP}/DOWN$ changes, *swap* is

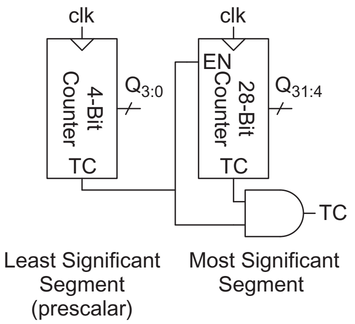


FIGURE 11.51 Fast binary counter

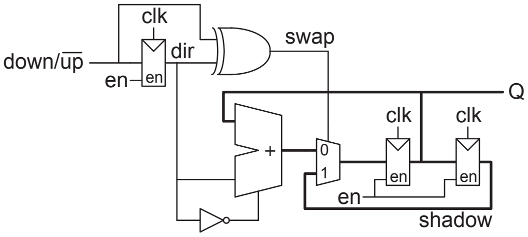


FIGURE 11.52 Fast binary up/down counter (most significant segment)

asserted for a cycle to load the new *count* from the *shadow* register rather than the adder (which may not have had enough time to ripple carries).

11.5.3 Ring and Johnson Counters

A *ring counter* consists of an M-bit shift register with the output fed back to the input, as shown in Figure 11.53(a). On reset, the first bit is initialized to 1 and the others are initialized to 0. TC toggles once every M cycles. Ring counters are a convenient way to build extremely fast prescalars because there is no logic between flip-flops, but they become costly for larger M.

A *Johnson* or *Mobius counter* is similar to a ring counter, but inverts the output before it is fed back to the input, as shown in Figure 11.53(b). The flip-flops are reset to all zeros and count through $2M$ states before repeating. Table 11.6 shows the sequence for a 3-bit Johnson counter.

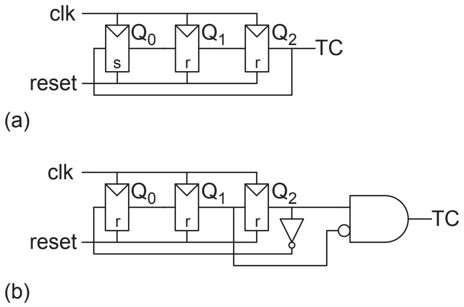


FIGURE 11.53 3-bit ring and Johnson counters

TABLE 11.6 Johnson counter sequence

Cycle	Q_0	Q_1	Q_2	TC
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0
4	0	1	1	0
5	0	0	1	1
6	0	0	0	0
Repeats forever				

11.5.4 Linerar-Feedback Shift Registers

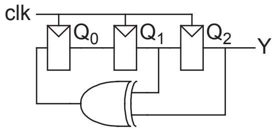


FIGURE 11.54 3-bit LFSR

A linear-feedback shift register (LFSR) consists of N registers configured as a shift register. The input to the shift register comes from the XOR of particular bits of the register, as shown in Figure 11.54 for a 3-bit LFSR. On reset, the registers must be initialized to a nonzero value (e.g., all 1s). The pattern of outputs for the LFSR is shown in Table 11.7.

TABLE 11.7 LFSR sequence

Cycle	Q_0	Q_1	Q_2 / Y
0	1	1	1
1	0	1	1
2	0	0	1
3	1	0	0
4	0	1	0
5	1	0	1
6	1	1	0
7	1	1	1
Repeats forever			

This LFSR is an example of a *maximal-length* shift register because its output sequences through all $2^n - 1$ combinations (excluding all 0s). The inputs fed to the XOR are called the *tap sequence* and are often specified with a *characteristic polynomial*. For example, this 3-bit LFSR has the characteristic polynomial $1 + x^2 + x^3$ because the taps come after the second and third registers.

The output *Y* follows the 7-bit sequence [1110010]. This is an example of a *pseudo-random bit sequence* (PRBS). LFSRs are used for high-speed counters and pseudo-random number generators. The pseudo-random sequences are handy for built-in self-test and bit-error-rate testing in communications links. They are also used in many spread-spectrum communications systems such as GPS and CDMA where their correlation properties make other users look like uncorrelated noise.

Table 11.8 lists characteristic polynomials for some commonly used maximal-length LFSRs. For certain lengths, *N*, more than two taps may be required. For many values of *N*, there are multiple polynomials resulting in different maximal-length LFSRs. Observe that the cycle time is set by the register and a small number of XOR delays. [Golomb81] offers the definitive treatment on linear-feedback shift registers.

TABLE 11.8 Characteristic polynomials

N	Polynomial
3	$1 + x^2 + x^3$
4	$1 + x^3 + x^4$
5	$1 + x^3 + x^5$
6	$1 + x^5 + x^6$
7	$1 + x^6 + x^7$
8	$1 + x^1 + x^6 + x^7 + x^8$
9	$1 + x^5 + x^9$
15	$1 + x^{14} + x^{15}$
16	$1 + x^4 + x^{13} + x^{15} + x^{16}$
23	$1 + x^{18} + x^{23}$
24	$1 + x^{17} + x^{22} + x^{23} + x^{24}$
31	$1 + x^{28} + x^{31}$
32	$1 + x^{10} + x^{30} + x^{31} + x^{32}$

Example 11.1

Sketch an 8-bit linear-feedback shift register. How long is the pseudo-random bit sequence that it produces?

SOLUTION: Figure 11.55 shows an 8-bit LFSR using the four taps after the 1st, 6th, 7th, and 8th bits, as given in Table 11.7. It produces a sequence of $2^8 - 1 = 255$ bits before repeating.

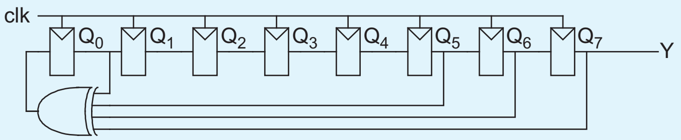


FIGURE 11.55 8-bit LFSR

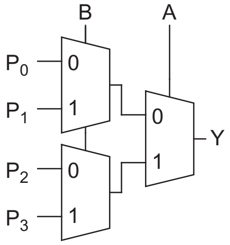


FIGURE 11.56
Boolean logical unit

11.6 Boolean Logical Operations

Boolean logical operations are easily accomplished using a multiplexer-based circuit, as shown in Figure 11.56. Table 11.9 shows how the inputs are assigned to perform different logical functions. By providing different P values, the unit can perform other operations such as XNOR(A, B) or NOT(A). An *Arithmetic Logic Unit* (ALU) requires both arithmetic (add, subtract) and Boolean logical operations.

TABLE 11.9 Functions implemented by Boolean unit

Operation	P_0	P_1	P_2	P_3
AND(A, B)	0	0	0	1
OR(A, B)	0	1	1	1
XOR(A, B)	0	1	1	0
NAND(A, B)	1	1	1	0
NOR(A, B)	1	0	0	0

11.7 Coding

Error-detecting and error-correcting codes are used to increase system reliability. Memory arrays are particularly susceptible to soft errors caused by alpha particles or cosmic rays flipping a bit. Such errors can be detected or even corrected by adding a few extra *check bits* to each word in the array. Codes are also used to reduce the bit error rate in communication links.

The simplest form of error-detecting code is *parity*, which detects single-bit errors. More elaborate *error-correcting codes* (ECC) are capable of single-error correcting and double-error detecting (SEC-DED). Gray codes are another useful alternative to the standard binary codes. All of the codes are heavily based on the XOR function, so we will examine a variety of CMOS XOR designs.

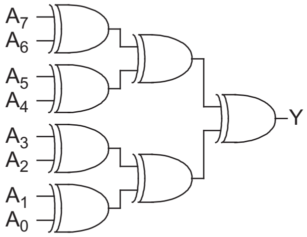


FIGURE 11.57
8-bit parity generator

11.7.1 Parity

A parity bit can be added to an N-bit word to indicate whether the number of 1s in the word is even or odd. In *even parity*, the extra bit is the XOR of the other N bits, which ensures the (N + 1)-bit coded word has an even number of 1s:

$$A_n = \text{PARITY} = A_0 \oplus A_1 \oplus A_2 \oplus \ldots \oplus A_{n-1} \qquad \textbf{(11.28)}$$

Figure 11.57 shows a conventional implementation. Multi-input XOR gates can also be used.

11.7.2 Error-Correcting Codes

The *Hamming distance* [Hamming50] between a pair of binary numbers is the number of bits that differ between the two numbers. A single-bit error transforms a data word into another word separated by a Hamming distance of 1. Error-correcting codes add check bits to the data word so that the minimum Hamming distance between valid words increases. Parity is an example of a code with a single check bit and a Hamming distance of 2 between valid words, so that single-bit errors lead to invalid words and hence are detectable. If more check bits are added so that the minimum distance between valid words is 3, a single-bit error can be corrected because there will be only one valid word within a distance of 1. If the minimum distance between valid words is 4, a single-bit error can be corrected and an error corrupting two bits can be detected (but not corrected). If the probability of bit errors is low and uncorrelated from one bit to another, such single error-correcting, double error-detecting (SEC-DED) codes greatly reduce the overall error rate of the system. Larger Hamming distances improve the error rate further at the expense of more check bits.

In general, you can construct a distance-3 Hamming code of length up to $2^c - 1$ with c check bits and $N = 2^c - c - 1$ data bits using a simple procedure [Wakerly00]. If the bits are numbered from 1 to $2^c - 1$, each bit in a position that is a power of 2 serves as a check bit. The value of the check bit is chosen to obtain even parity for all bits with a 1 in the same position as the check bit, as illustrated in Figure 11.58(a) for a 7-bit code with 4 data bits and 3 check bits. The bits are traditionally reorganized into contiguous data and check bits, as shown in Figure 11.58(b). The structure is called a *parity-check matrix* and each check bit can be computed as the XOR of the highlighted data bits:

$$
\begin{aligned}
C_0 &= D_3 \oplus D_1 \oplus D_0 \\
C_1 &= D_3 \oplus D_2 \oplus D_0 \\
C_2 &= D_3 \oplus D_2 \oplus D_1
\end{aligned}
\tag{11.29}
$$

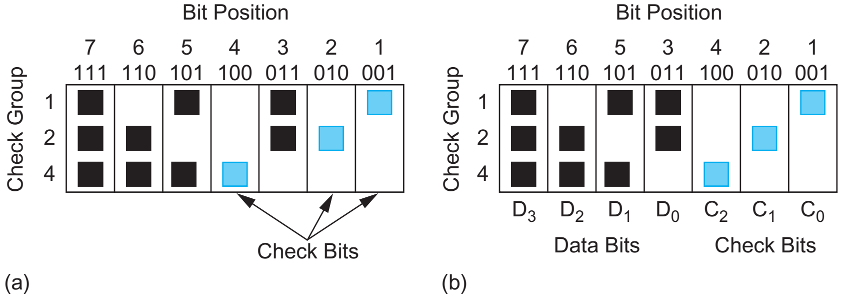


FIGURE 11.58 Parity-check matrix

The error-correcting decoder examines the check bits. If they all have even parity, the word is considered to be correct. If one or more groups have odd parity, an error has occurred. The pattern of check bits that have the wrong parity is called the *syndrome* and corresponds to the bit position that is incorrect. The decoder must flip this bit to recover the correct result.

Example 11.2

Suppose the data value 1001 were to be transmitted using a distance-3 Hamming code. What are the check bits? If the data bits were garbled into 1101 during transmission, explain what the syndrome would be and how the data would be corrected.

SOLUTION: According to EQ (11.29), the check bits should be 100, corresponding to a transmitted word of 1001100. The received word is 1101100. The syndrome is 110, i.e., odd parity on check bits C_2 and C_1, which indicates an error in bit position 110 = 6. This position is flipped to produce a corrected word of 1001100 and the check bits are discarded, leaving the proper data value of 1001.

A SEC-DED distance-4 Hamming code can be constructed from a distance-3 code by adding one more parity bit for the entire word. If there is a single-bit error, parity will fail and the check bits will indicate how to correct the data. If there is a double-bit error, the check bits will indicate an error, but parity will pass, indicating a detectable but uncorrectable double-bit error.

The parity check matrix determines the number of XORs required in the encoding and decoding logic. A SEC-DED Hamming code for a 64-bit data word has 8 check bits. It requires 296 XOR gates. The parity logic for the entire word has 72 inputs. The Hsiao SEC-DED achieves the same function with the same number of data and check bits but is ingeniously designed to minimize the cost, using only 216 XOR gates and parity logic with a maximum of 27 inputs. [Hsiao70] shows parity-check matrices for 16, 32, and 64-bit data words with 6, 7, and 8 check bits.

As the data length and allowable decoder complexity increase, other codes become efficient. These include Reed-Solomon, BCH, and Turbo codes. [Lin83, Sweeney02, Sklar01, Fujiwara06] and many other texts provide extensive information on a variety of error-correcting codes.

11.7.3 Gray Codes

The *Gray codes*, named for Frank Gray, who patented their use on shaft encoders [Gray53], have a useful property that consecutive numbers differ in only one bit position. While there are many possible Gray codes, one of the simplest is the *binary-reflected Gray code* that is generated by starting with all bits 0 and successively flipping the right-most bit that produces a new string. Table 11.10 compares 3-bit binary and binary-reflected Gray codes. Finite state machines that typically move through consecutive states can save power by Gray-coding the states to reduce the number of transitions. When a counter value must be synchronized across clock domains, it can be Gray-coded so that the synchronizer is certain to receive either the current or previous value because only one bit changes each cycle.

TABLE 11.10 3-bit Gray code

Number	Binary	Gray Code
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

Converting between N-bit binary B and binary-reflected Gray code G representations is remarkably simple.

$$\begin{array}{lll} \text{Binary} \rightarrow \text{Gray} & \text{Gray} \rightarrow \text{Binary} & \\ G_{N-1} = B_{N-1} & B_{N-1} = G_{N-1} & \\ G_i = B_{i+1} \oplus B_i & B_i = B_{i+1} \oplus G_i & N-1 > i \geq 0 \end{array} \quad \textbf{(11.30)}$$

11.7.4 XOR/XNOR Circuit Forms

One of the chronic difficulties in CMOS circuit design is to construct a fast, compact, low-power XOR or XNOR gate. Figure 11.59 shows a number of common static single-rail 2-input XOR designs; XNOR designs are similar. Figure 11.59(a) and Figure 11.59(b) show gate-level implementations; the first is cute, but the second is slightly more efficient. Figure 11.59(c) shows a complementary CMOS gate. Figure 11.59(d) improves the gate by optimizing out two contacts and is a commonly used standard cell design. Figure 11.59(e) shows a transmission gate design. Figure 11.59(f) is the 6-transistor "invertible inverter" design. When *A* is 0, the transmission gate turns on and *B* is passed to the output. When *A* is 1, the *A* input powers a pair of transistors that invert *B*. It is compact, but nonrestoring. Some switch-level simulators such as IRSIM cannot handle this unconventional design. Figure 11.59(g) [Wang94] is a compact and fast 4-transistor pass-gate design, but does not swing rail to rail.

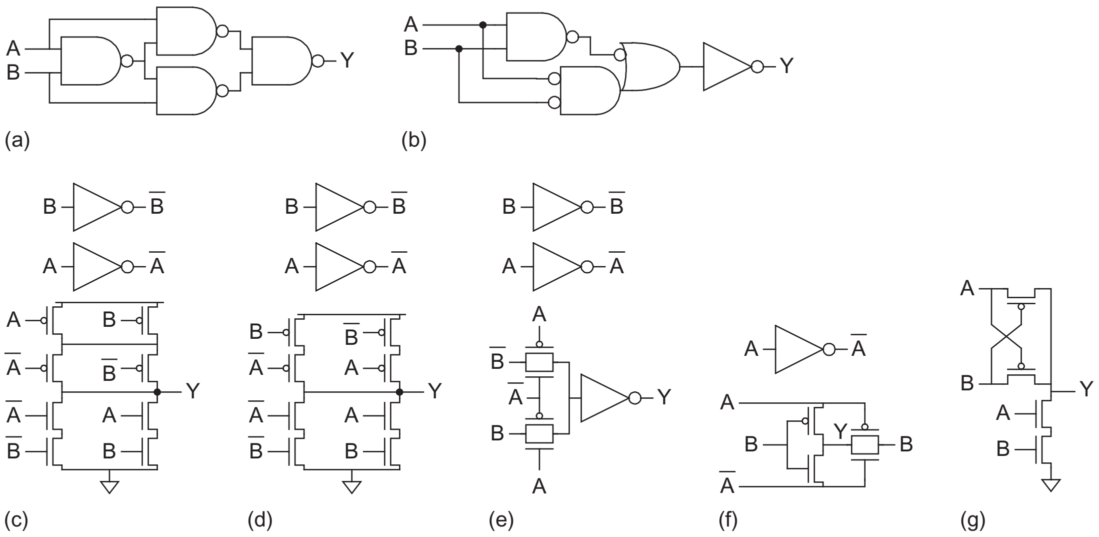


FIGURE 11.59 Static 2-input XOR designs

XOR gates with 3 or 4 inputs can be more compact, although not necessarily faster than a cascade of 2-input gates. Figure 11.60(a) is a 4-input static CMOS XOR [Griffin83] and Figure 11.60(b) is a 4-input CPL XOR/XNOR, while Figure 9.20(c) showed a 4-input CVSL XOR/XNOR. Observe that the true and complementary trees share most of the transistors. As mentioned in Chapter 9, CPL does not perform well at low voltage.

Dynamic XORs pose a problem because both true and complementary inputs are required, violating the monotonicity rule. The common solutions mentioned in Section 11.2.2.11 are to either push the XOR to the end of a chain of domino logic and build it

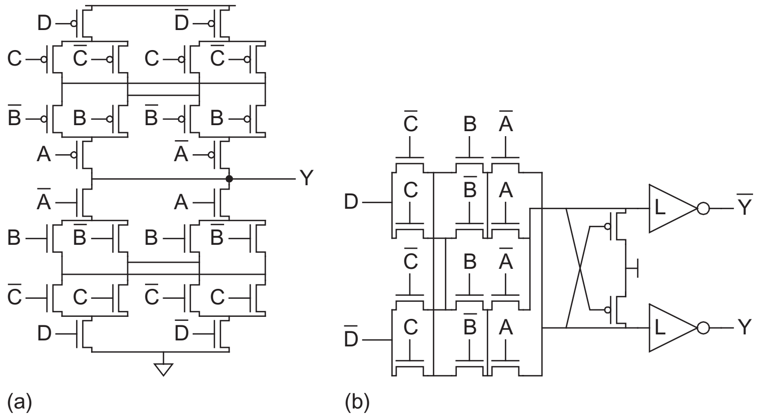


FIGURE 11.60 4-input XOR designs

with static CMOS or to construct a dual-rail domino structure. A dual-rail domino 2-input XOR was shown in Figure 9.30(c).

11.8 Shifters

Shifts can either be performed by a constant or variable amount. Constant shifts are trivial in hardware, requiring only wires. They are also an efficient way to perform multiplication or division by powers of two. A variable shifter takes an N-bit input, A, a shift amount, k, and control signals indicating the shift type and direction. It produces an N-bit output, Y. There are three common types of variable shifts, each of which can be to the left or right:

- *Rotate*: Rotate numbers in a circle such that empty spots are filled with bits shifted off the other end
  - Example: 1011 ROR 1 = 1101; 1011 ROL 1 = 0111
- *Logical shift*: Shift the number to the left or right and fills empty spots with zeros.
  - Example: 1011 LSR 1 = 0101; 1011 LSL 1 = 0110
- *Arithmetic shift*: Same as logical shifter, but on right shifts fills the most significant bits with copies of the sign bit (to properly sign, extend two's complement numbers when using right shift by k for division by 2^k).
  - Example: 1011 ASR 1 = 1101; 1011 ASL 1 = 0110

Conceptually, rotation involves an array of N N-input multiplexers to select each of the outputs from each of the possible input positions. This is called an *array shifter*. The array shifter requires a decoder to produce the 1-of-N-hot shift amount. In practice, multiplexers with more than 4–8 inputs have excessive parasitic capacitance, so they are faster to construct from $\log_v N$ levels of v-input multiplexers. This is called a *logarithmic shifter*. For example, in a radix-2 logarithmic shifter, the first level shifts by $N/2$, the second by $N/4$, and so forth until the final level shifts by 1. In a logarithmic shifter, no decoder is necessary. The CMOS transmission gate multiplexer of Figure 9.47 is especially well-suited to logarithmic shifters because the hefty wire capacitance is driven directly by an inverter rather than through a pair of series transistors. 4:1 or 8:1 transmission gate multiplexers reduce the number of levels by a factor of 2 or 3 at the expense of more wiring and fanout. Pairs or triplets of the shift amount are decoded to drive one-hot mux selects at each level. [Tharakan92] describes a domino logarithmic shifter using 3:1 multiplexers to reduce the number of logic levels.

A left rotate by k bits is equivalent to a right rotate by $N - k$ bits. Computing $N - k$ requires a subtracter in the critical path. Taking advantage of two's complement arithmetic and the fact that rotation is cyclic modulo N, $N - k = N + \bar{k} + 1 = \bar{k} + 1$. Thus, the left rotate can be performed by preshifting right by 1, then doing a right rotate by the complemented shift amount.

Logical and arithmetic shifts are similar to rotates, but must replace bits at one end or the other with a *kill value* (either 0 or the sign bit). The two major shifter architectures are funnel shifters and barrel shifters. In a *funnel shifter*, the kill values are incorporated at the beginning, while in a *barrel shifter*, the kill values are chosen at the end. Each of these architectures is described below. Both barrel and funnel shifters can use array or logarithmic implementations. [Huntzicker08] examines the energy-delay trade-offs in static shifters. For general-purpose shifting, both architectures are comparable in energy and delay. Given typical parasitics capacitances, the theory of Logical Effort shows that logarithmic structure using 4:1 multiplexers is most efficient. If only shift operations (but not rotates) are required, the funnel architecture is simpler, while if only rotates (but not shifts) are required, the barrel is simpler.

11.8.1 Funnel Shifter

The *funnel shifter* creates a $2N - 1$-bit input word Z from A and/or the kill values, then selects an N-bit field from this input word, as shown in Figure 11.61. It gets its name from the way the wide word funnels down to a narrower one. Table 11.11 shows how Z is formed for each type of shift. Z incorporates the 1-bit preshift for left shifts.

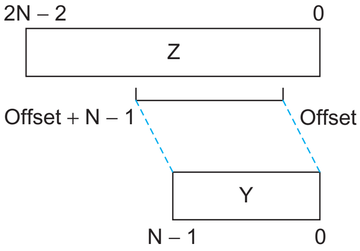


FIGURE 11.61 Funnel shifter function

TABLE 11.11 Funnel shifter source generator

Shift Type	$Z_{2N-2:N}$	Z_{N-1}	$Z_{N-2:0}$	Offset
Logical Right	$A_{N-2:0}$	A_{N-1}	$A_{N-2:0}$	k
Arithmetic Right	0	A_{N-1}	$A_{N-2:0}$	k
Rotate Right	sign	A_{N-1}	$A_{N-2:0}$	k
Logical/Arithmetic Left	$A_{N-1:1}$	A_0	$A_{N-1:1}$	$\bar{k}$
Rotate Left	$A_{N-1:1}$	A_0	0	$\bar{k}$

The simplest funnel shifter design consists of an array of N N-input multiplexers accepting 1-of-N-hot select signals (one multiplexer for each output bit). Such an array shifter is shown in Figure 11.62 using nMOS pass transistors for a 4-bit shifter. The shift amount is conditionally inverted and decoded into select signals that are fed vertically across the array. The outputs are taken horizontally. Each row of transistors attached to an output forms one of the multiplexers. The $2N - 1$ inputs run diagonally to the appropriate mux inputs. Figure 11.63 shows a stick diagram for one of the N^2 transistors in the array. nMOS pass transistors suffer a threshold drop, but the problem can be solved by precharging the outputs (done in the Alpha 21164 [Gronowski96]) or by using full CMOS transmission gates.

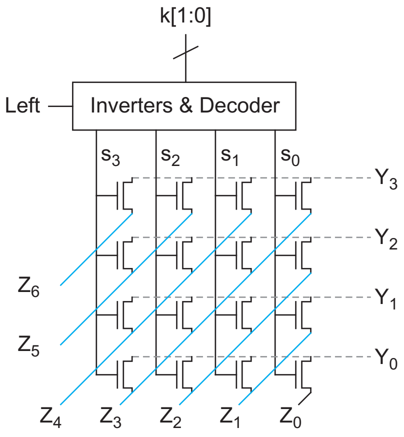


FIGURE 11.62 4-bit array funnel shifter

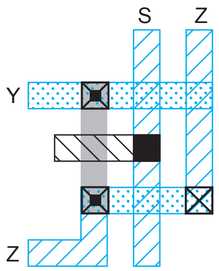


FIGURE 11.63 Array funnel shifter cell stick diagram

The array shifter works well for small shifters in transistor-level designs, but has high parasitic capacitance in larger shifters, leading to excessive delay and energy. Moreover, array shifters are not amenable to standard cell designs. Figure 11.64 shows a 4-bit logarithmic shifter based on multiple levels of 2:1 multiplexers (which, of course, can be transmission gates) [Lim72]. The XOR gates on the control inputs conditionally invert the shift amount for left shifts.

Figure 11.65 shows a 32-bit funnel shifter using a 4:1 multiplexer followed by an 8:1 multiplexer [Huntzicker08]. The source generator selects the 63-bit *Z*. The first stage performs a coarse shift right by 0, 8, 16, or 24 bits. The second stage performs a fine shift right by 0–7 bits. The mux decode block conditionally inverts *k* for left shifts, computes the 1-hot selects, and buffers them to drive the wide multiplexers.

Conceptually, the source generator consists of a 2*N* − 1-bit 5:1 multiplexer controlled by the shift type and direction. Figure 11.66 shows how the source generator logic can be simplified. The horizontal control lines need to be buffered to drive the high fanout and they are on the critical path. Even if they are available early, the sign bit is still critical. If only certain types of shifts or rotates are supported, the logic can be optimized down further.

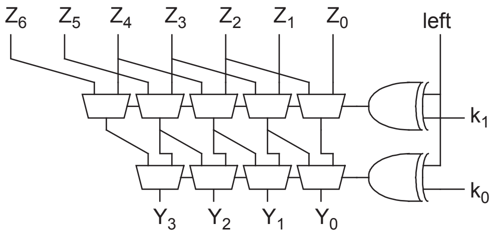


FIGURE 11.64 4-bit logarithmic funnel shifter

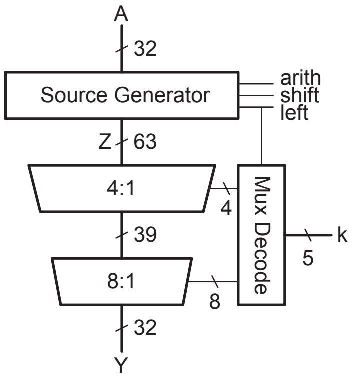


FIGURE 11.65
32-bit logarithmic funnel shifter

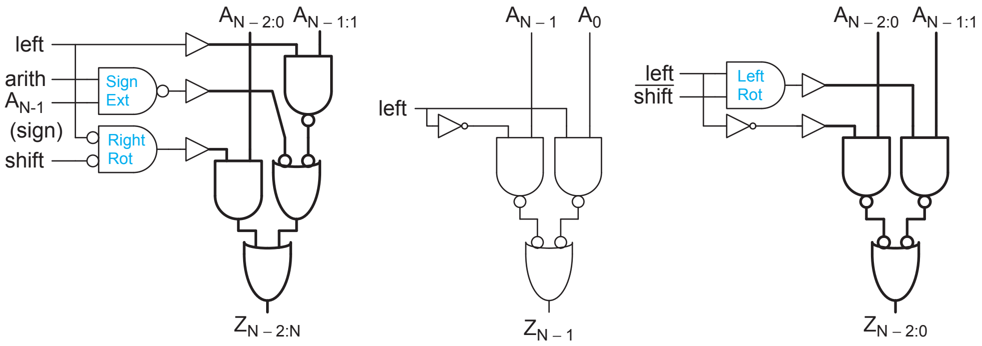


FIGURE 11.66 Optimized source generator logic

The funnel shifter presents a layout problem because the source generator and early stages of multiplexers are wider than the rest of the datapath. Figure 11.67 shows a floorplan in which the source generator is folded to fit the datapath. Such folding also reduces wire lengths, saving energy. Depending on the layout constraints, the extra seven most significant bits of the first-level multiplexer may be folded into another row or incorporated into the zipper.

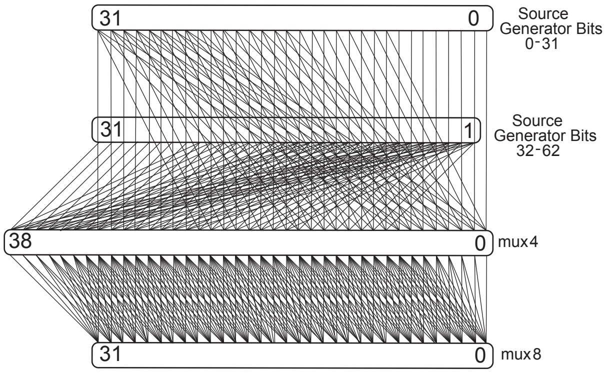


FIGURE 11.67 Funnel shifter floorplans

11.8.2 Barrel Shifter

A *barrel shifter* performs a right rotate operation [Davis69]. As mentioned earlier, it handles left rotations using the complementary shift amount. Barrel shifters can also perform shifts when suitable masking hardware is included. Barrel shifters come in array and logarithmic forms; we focus on logarithmic barrel shifters because they are better suited for large shifts.

Figure 11.68(a) shows a simple 4-bit barrel shifter that performs right rotations. Notice how, unlike funnel shifters, barrel shifters contain long wrap-around wires. In a large shifter, it is beneficial to upsize or buffer the drivers for these wires. Figure 11.68(b) shows an enhanced version that can rotate left by prerotating right by 1, then rotating right by $\bar{k}$. Performing logical or arithmetic shifts on a barrel shifter requires a way to mask out the bits that are rotated off the end of the shifter, as shown in Figure 11.68(c).

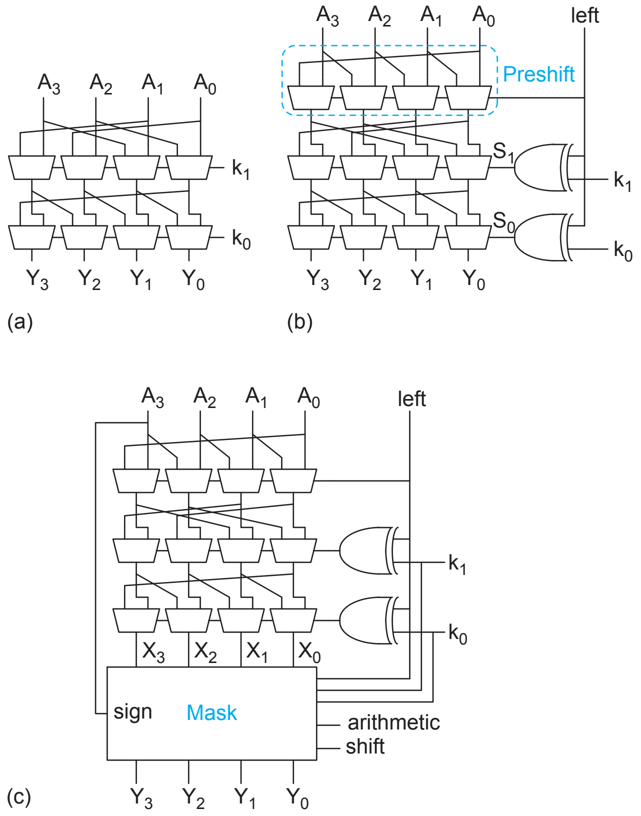


FIGURE 11.68 Barrel shifters: (a) rotate right, (b) rotate left or right, (c) rotates and shifts

Figure 11.69 shows a 32-bit barrel shifter using a 5:1 multiplexer and an 8:1 multiplexer. The first stage rotates right by 0, 1, 2, 3, or 4 bits to handle a prerotate of 1 bit and a fine rotate of up to 3 bits combined into one stage. The second stage rotates right by 0, 4, 8, 12, 16, 20, 24, or 28 bits. The critical path starts with decoding the shift amount for the first stage. If the shift amount is available early, the delay from A to Y improves substantially.

While the rotation is taking place, the masking unit generates an N-bit mask with ones where the kill value should be inserted for right shifts. For a right shift by m, the m most significant bits are ones. This is called a thermometer code and the logic to compute it is described in Section 11.10. When the rotation result X is complete, the masking unit replaces the masked bits with the kill value. For left shifts, the mask is reversed. Figure 11.70 shows masking logic. If only certain shifts are supported, the unit can be simplified, and if only rotates are supported, the masking unit can be eliminated, saving substantial hardware, power, and delay.

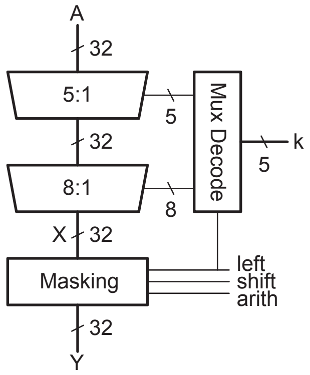


FIGURE 11.69 32-bit logarithmic barrel shifter

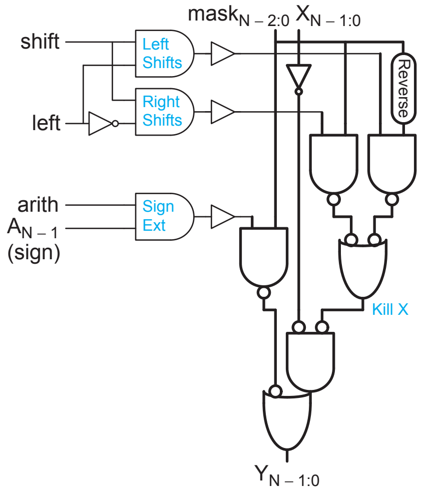


FIGURE 11.70 Barrel shifter masking logic

11.8.3 Alternative Shift Functions

Other flavors of shifts, including shuffles, bit-reversals, interchanges, extraction, and deposit, are sometimes required, especially for cryptographic and multimedia applications [Hilewitz04, Hilewitz07]. These are also built from appropriate combinations of multiplexers.

11.9 Multiplication

Multiplication is less common than addition, but is still essential for microprocessors, digital signal processors, and graphics engines. The most basic form of multiplication consists of forming the product of two unsigned (positive) binary numbers. This can be accomplished through the traditional technique taught in primary school, simplified to base 2. For example, the multiplication of two positive 6-bit binary integers, 25_{10} and 39_{10}, proceeds as shown in Figure 11.71.

```
      011001 : 25₁₀       multiplicand
    × 100111 : 39₁₀       multiplier
      011001          ┐
     011001           │
    011001            │   partial
   000000             │   products
  000000              │
+011001               ┘
001111001111 : 975₁₀      product
```

FIGURE 11.71 Multiplication example

$M \times N$-bit multiplication $P = Y \times X$ can be viewed as forming N partial products of M bits each, and then summing the appropriately shifted partial products to produce an $M + N$-bit result P. Binary multiplication is equivalent to a logical AND operation. Therefore, generating partial products consists of the logical ANDing of the appropriate bits of the multiplier and multiplicand. Each column of partial products must then be added and, if necessary, any carry values passed to the next column. We denote the multiplicand as $Y = \{y_{M-1}, y_{M-2}, \ldots, y_1, y_0\}$ and the multiplier as $X = \{x_{N-1}, x_{N-2}, \ldots, x_1, x_0\}$. For unsigned multiplication, the product is given in EQ (11.31). Figure 11.72 illustrates the generation, shifting, and summing of partial products in a 6 × 6-bit multiplier.

$$P = \left(\sum_{j=0}^{M-1} y_j 2^j\right)\left(\sum_{i=0}^{N-1} x_i 2^i\right) = \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} x_i y_j 2^{i+j} \qquad \textbf{(11.31)}$$

						y_5	y_4	y_3	y_2	y_1	y_0	Multiplicand
						x_5	x_4	x_3	x_2	x_1	x_0	Multiplier
						x_0y_5	x_0y_4	x_0y_3	x_0y_2	x_0y_1	x_0y_0	Partial Products
					x_1y_5	x_1y_4	x_1y_3	x_1y_2	x_1y_1	x_1y_0		
				x_2y_5	x_2y_4	x_2y_3	x_2y_2	x_2y_1	x_2y_0			
			x_3y_5	x_3y_4	x_3y_3	x_3y_2	x_3y_1	x_3y_0				
		x_4y_5	x_4y_4	x_4y_3	x_4y_2	x_4y_1	x_4y_0					
	x_5y_5	x_5y_4	x_5y_3	x_5y_2	x_5y_1	x_5y_0						
p_{11}	p_{10}	p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0	Product

FIGURE 11.72 Partial products

Large multiplications can be more conveniently illustrated using *dot diagrams*. Figure 11.73 shows a dot diagram for a simple 16 × 16 multiplier. Each dot represents a placeholder for a single bit that can be a 0 or 1. The partial products are represented by a horizontal boxed row of dots, shifted according to their weight. The multiplier bits used to generate the partial products are shown on the right.

There are a number of techniques that can be used to perform multiplication. In general, the choice is based upon factors such as latency, throughput, energy, area, and design complexity. An obvious approach is to use an $M + 1$-bit carry-propagate adder (CPA) to add the first two partial products, then another CPA to add the third partial product to the running sum, and so forth. Such an approach requires $N - 1$ CPAs and is slow, even if a fast CPA is employed. More efficient parallel approaches use some sort of array or tree of full adders to sum the partial products. We begin with a simple array for unsigned multipliers, and then modify the array to handle signed two's complement numbers using the Baugh-Wooley algorithm. The number of partial products to sum can be reduced using Booth encoding and the number of logic levels required to perform the summation can be reduced with Wallace trees. Unfortunately, Wallace trees are complex to lay out and have long, irregular wires, so hybrid array/tree structures may be more attractive. For completeness, we consider a serial multiplier architecture. This was once popular when gates were relatively expensive, but is now rarely necessary.

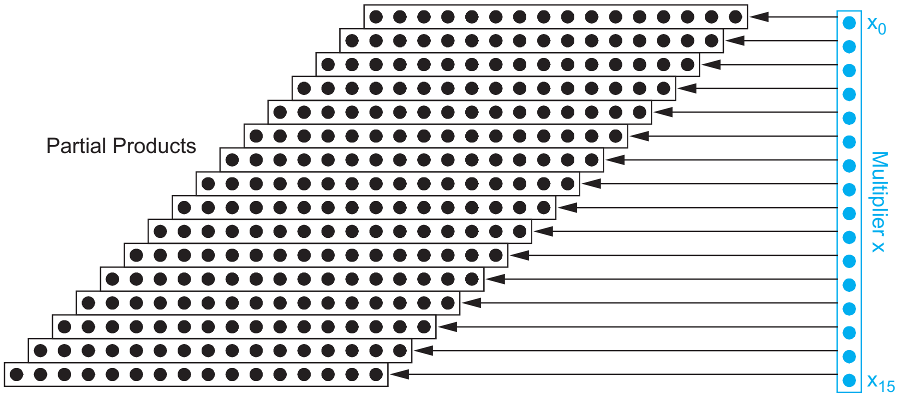


FIGURE 11.73 Dot diagram

11.9.1 Unsigned Array Multiplication

Fast multipliers use carry-save adders (CSAs, see Section 11.2.4) to sum the partial products. A CSA typically has a delay of 1.5–2 FO4 inverters independent of the width of the partial product, while a carry-propagate adder (CPA) tends to have a delay of 4–15+ FO4 inverters depending on the width, architecture, and circuit family. Figure 11.74 shows a 4 × 4 array multiplier for unsigned numbers using an array of CSAs. Each cell contains a 2-input AND gate that forms a partial product and a full adder (CSA) to add the partial product into the running sum. The first row converts the first partial product into carry-save redundant form. Each later row uses the CSA to add the corresponding partial product to the carry-save redundant result of the previous row and generate a carry-save redundant result. The least significant N output bits are available as sum outputs directly from CSAs. The most significant output bits arrive in carry-save redundant form and require an M-bit carry-propagate adder to convert into regular binary form. In Figure 11.74, the CPA is implemented as a carry-ripple adder. The array is regular in structure and uses a single type of cell, so it is easy to design and lay out. Assuming the carry output is faster than the sum output in a CSA, the critical path through the array is marked on the figure with a dashed line. The adder can easily be pipelined with the placement of registers between rows. In practice, circuits are assigned rectangular blocks in the floorplan so the parallelogram shape wastes space. Figure 11.75 shows the same adder squashed to fit a rectangular block.

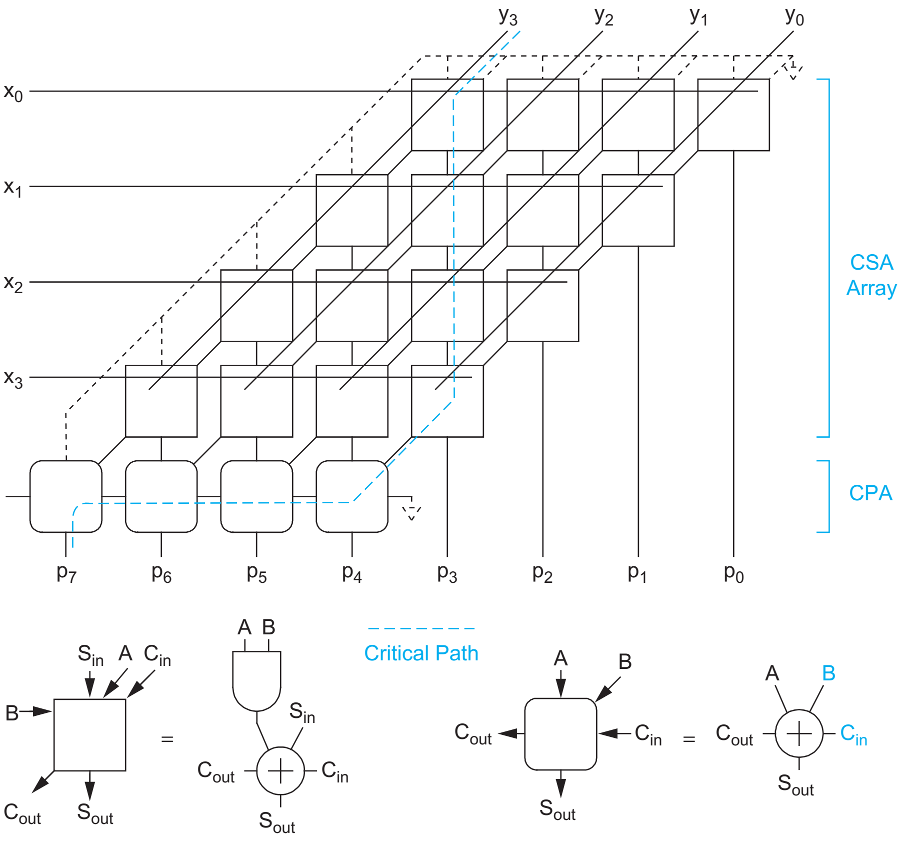


FIGURE 11.74 Array multiplier

A key element of the design is a compact CSA. This not only benefits area but also helps performance because it leads to short wires with low wire capacitance. An ideal CSA design has approximately equal sum and carry delays because the greater of these two delays limits performance. The mirror adder from Figure 11.4 is commonly used for its compact layout even though the sum delay exceeds the carry delay. The sum output can be connected to the faster carry input to partially compensate [Sutherland99, Hsu06a].

Note that the first row of CSAs adds the first partial product to a pair of 0s. This leads to a regular structure, but is inefficient. At a slight cost to regularity, the first row of CSAs can be used to add the first three partial products together. This reduces the number of rows by two and correspondingly reduces the adder propagation delay. Yet another way to improve the multiplier array performance is to replace the bottom row with a faster CPA such as a lookahead or tree adder. In summary, the critical path of an array multiplier involves N–2 CSAs and a CPA.

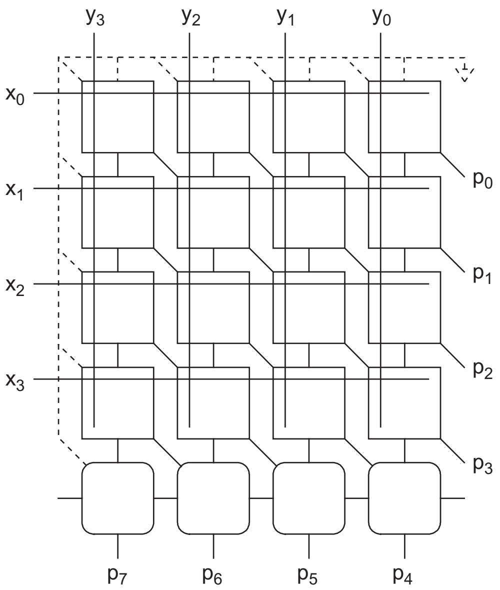


FIGURE 11.75 Rectangular array multiplier

11.9.2 Two's Complement Array Multiplication

Multiplication of two's complement numbers at first might seem more difficult because some partial products are negative and must be subtracted. Recall that the most significant bit of a two's complement number has a negative weight. Hence, the product is

$$\begin{aligned} P &= \left(-y_{M-1} 2^{M-1} + \sum_{j=0}^{M-2} y_j 2^j \right) \left(-x_{n-1} 2^{N-1} + \sum_{i=0}^{N-2} x_i 2^i \right) \\ &= \sum_{i=0}^{N-2} \sum_{j=0}^{M-2} x_i y_j 2^{i+j} + x_{N-1} y_{M-1} 2^{M+N-2} - \left(\sum_{i=0}^{N-2} x_i y_{M-1} 2^{i+M-1} + \sum_{j=0}^{M-2} x_{N-1} y_j 2^{j+N-1} \right) \end{aligned} \tag{11.32}$$

In EQ (11.32), two of the partial products have negative weight and thus should be subtracted rather than added. The *Baugh-Wooley* [Baugh73] multiplier algorithm handles subtraction by taking the two's complement of the terms to be subtracted (i.e., inverting the bits and adding one). Figure 11.76 shows the partial products that must be summed. The upper parallelogram represents the unsigned multiplication of all but the most significant bits of the inputs. The next row is a single bit corresponding to the product of the most significant bits. The next two pairs of rows are the inversions of the terms to be subtracted. Each term has implicit leading and trailing zeros, which are inverted to leading and trailing ones. Extra ones must be added in the least significant column when taking the two's complement.

The multiplier delay depends on the number of partial product rows to be summed. The *modified Baugh-Wooley multiplier* [Hatamian86] reduces this number of partial products by precomputing the sums of the constant ones and pushing some of the terms upward into extra columns. Figure 11.77 shows such an arrangement. The parallelogram-shaped array can again be squashed into a rectangle, as shown in Figure 11.78, giving a design almost identical to the unsigned multiplier of Figure 11.75. The AND gates are replaced by NAND gates in the hatched cells and 1s are added in place of 0s at two of the

							y_5	y_4	y_3	y_2	y_1	y_0
							x_5	x_4	x_3	x_2	x_1	x_0
$\sum_{i=0}^{N-2}\sum_{j=0}^{M-2} x_i y_j 2^{i+j}$								x_0y_4	x_0y_3	x_0y_2	x_0y_1	x_0y_0
							x_1y_4	x_1y_3	x_1y_2	x_1y_1	x_1y_0	
						x_2y_4	x_2y_3	x_2y_2	x_2y_1	x_2y_0		
					x_3y_4	x_3y_3	x_3y_2	x_3y_1	x_3y_0			
				x_4y_4	x_4y_3	x_4y_2	x_4y_1	x_4y_0				
$x_{N-1} y_{M-1} 2^{M+N-2}$		x_5y_5										
$-\sum_{i=0}^{N-2} x_i y_{M-1} 2^{i+M-1}$	1	1	$\overline{x_4y_5}$	$\overline{x_3y_5}$	$\overline{x_2y_5}$	$\overline{x_1y_5}$	$\overline{x_0y_5}$	1	1	1	1	1
												1
$-\sum_{j=0}^{M-2} x_{N-1} y_j 2^{j+N-1}$	1	1	$\overline{x_5y_4}$	$\overline{x_5y_3}$	$\overline{x_5y_2}$	$\overline{x_5y_1}$	$\overline{x_5y_0}$	1	1	1	1	1
												1
	p_{11}	p_{10}	p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

FIGURE 11.76 Partial products for two's complement multiplier

						y_5	y_4	y_3	y_2	y_1	y_0
						x_5	x_4	x_3	x_2	x_1	x_0
					1	$\overline{x_5y_0}$	x_0y_4	x_0y_3	x_0y_2	x_0y_1	x_0y_0
					$\overline{x_5y_1}$	x_1y_4	x_1y_3	x_1y_2	x_1y_1	x_1y_0	
				$\overline{x_5y_2}$	x_2y_4	x_2y_3	x_2y_2	x_2y_1	x_2y_0		
			$\overline{x_5y_3}$	x_3y_4	x_3y_3	x_3y_2	x_3y_1	x_3y_0			
		$\overline{x_5y_4}$	x_4y_4	x_4y_3	x_4y_2	x_4y_1	x_4y_0				
1	x_5y_5	$\overline{x_4y_5}$	$\overline{x_3y_5}$	$\overline{x_2y_5}$	$\overline{x_1y_5}$	$\overline{x_0y_5}$					
p_{11}	p_{10}	p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	

FIGURE 11.77 Simplified partial products for two's complement multiplier

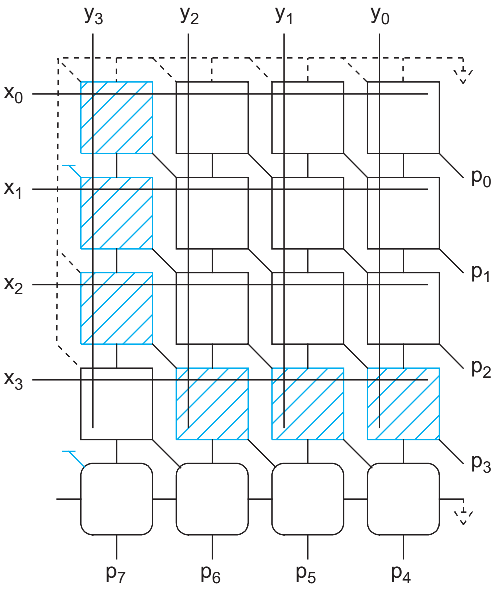


FIGURE 11.78 Modified Baugh-Wooley two's complement multiplier

unused inputs. The signed and unsigned arrays are so similar that a single array can be used for both purposes if XOR gates are used to conditionally invert some of the terms depending on the mode.

11.9.3 Booth Encoding

The array multipliers in the previous sections compute the partial products in a radix-2 manner; i.e., by observing one bit of the multiplier at a time. Radix 2^r multipliers produce

N/r partial products, each of which depend on r bits of the multiplier. Fewer partial products leads to a smaller and faster CSA array. For example, a radix-4 multiplier produces $N/2$ partial products. Each partial product is 0, Y, $2Y$, or $3Y$, depending on a pair of bits of X. Computing $2Y$ is a simple shift, but $3Y$ is a *hard multiple* requiring a slow carry-propagate addition of $Y + 2Y$ before partial product generation begins.

Booth encoding was originally proposed to accelerate serial multiplication [Booth51]. *Modified Booth encoding* [MacSorley61] allows higher radix parallel operation without generating the hard $3Y$ multiple by instead using negative partial products. Observe that $3Y = 4Y - Y$ and $2Y = 4Y - 2Y$. However, $4Y$ in a radix-4 multiplier array is equivalent to Y in the next row of the array that carries four times the weight. Hence, partial products are chosen by considering a pair of bits along with the most significant bit from the previous pair. If the most significant bit from the previous pair is true, Y must be added to the current partial product. If the most significant bit of the current pair is true, the current partial product is selected to be negative and the next partial product is incremented.

Table 11.12 shows how the partial products are selected, based on bits of the multiplier. Negative partial products are generated by taking the two's complement of the multiplicand (possibly left-shifted by one column for $-2Y$). An unsigned radix-4 Booth-encoded multiplier requires $\lceil (N+1)/2 \rceil$ partial products rather than N. Each partial product is $M + 1$ bits to accommodate the $2Y$ and $-2Y$ multiples. Even though X and Y are unsigned, the partial products can be negative and must be sign extended properly. The Booth selects will be discussed further after an example.

TABLE 11.12 Radix-4 modified Booth encoding values

Inputs			Partial Product	Booth Selects		
x_{2i+1}	x_{2i}	x_{2i-1}	PP_i	SINGLE$_i$	DOUBLE$_i$	NEG$_i$
0	0	0	0	0	0	0
0	0	1	Y	1	0	0
0	1	0	Y	1	0	0
0	1	1	$2Y$	0	1	0
1	0	0	$-2Y$	0	1	1
1	0	1	$-Y$	1	0	1
1	1	0	$-Y$	1	0	1
1	1	1	-0 $(= 0)$	0	0	1

Example 11.3

Repeat the multiplication of $P = Y \times X = 011001_2 \times 100111_2$ from Figure 11.71, applying Booth encoding to reduce the number of partial products.

SOLUTION: Figure 11.79 shows the multiplication. X is written vertically and the bits are used to select the four partial products. Each partial product is shifted two columns left of the previous one because it has four times the weight. The upper bits are sign-extended with 1s for negative partial products and 0s for positive partial products. The partial products are added to obtain the result.

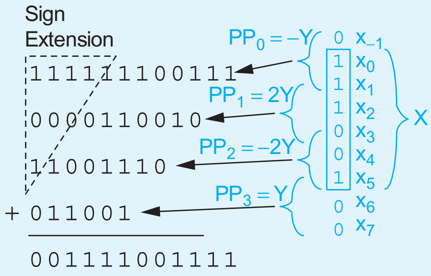


FIGURE 11.79 Booth-encoded example

In a typical radix-4 Booth-encoded multiplier design, each group of 3 bits (a pair, along with the most significant bit of the previous pair) is encoded into several select lines ($SINGLE_i$, $DOUBLE_i$, and NEG_i, given in the rightmost columns of Table 11.12) and driven across the partial product row as shown in Figure 11.80. The multiplier *Y* is distributed to all the rows. The select lines control Booth selectors that choose the appropriate multiple of *Y* for each partial product. The Booth selectors substitute for the AND gates of a simple array multiplier to determine the *i*th partial product. Figure 11.80 shows a conventional Booth encoder and selector design [Goto92]. *Y* is zero-extended to $M + 1$ bits. Depending on $SINGLE_i$ and $DOUBLE_i$, the A22OI gate selects either 0, *Y*, or 2*Y*. Negative partial products should be two's-complemented (i.e., invert and add 1). If NEG_i is asserted, the partial product is inverted. The extra 1 can be added in the least significant column of the next row to avoid needing a CPA.

Even in an unsigned multiplier, negative partial products must be sign-extended to be summed correctly. Figure 11.81 shows a 16-bit radix-4 Booth partial product array for an unsigned multiplier using the dot diagram notation. Each dot in the Booth-encoded mul-

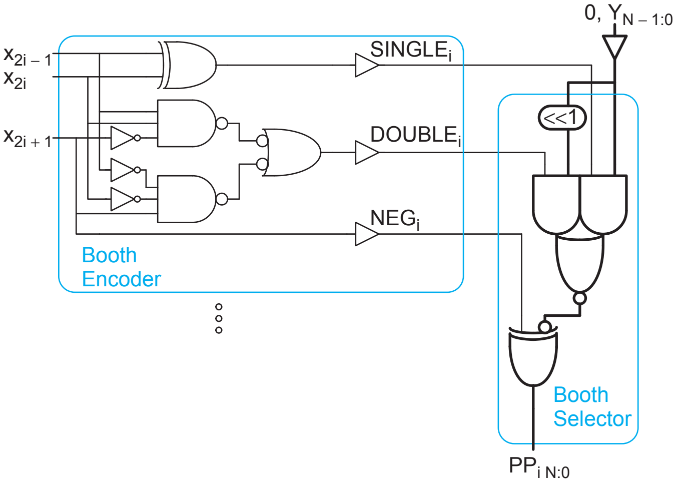


FIGURE 11.80 Radix-4 Booth encoder and selector

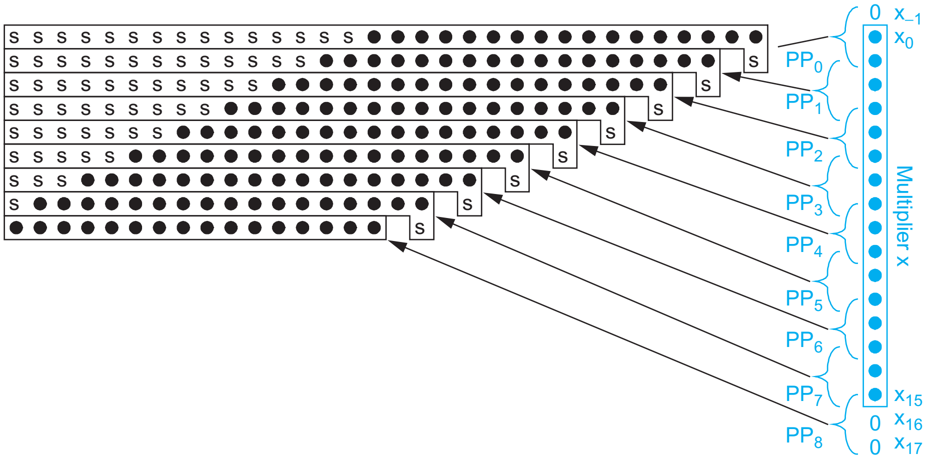


FIGURE 11.81 Radix-4 Booth-encoded partial products with sign extension

tiplier is produced by a Booth selector rather than a simple AND gate. Partial products 0–7 are 17 bits. Each partial product *i* is sign extended with $s_i = NEG_i = x_{2i+1}$, which is 1 for negative multiples (those in the bottom half of Table 11.12) or 0 for positive multiples. Observe how an extra 1 is added to the least significant bit in the next row to form the 2's complement of negative multiples. Inverting the implicit leading zeros generates leading ones on negative multiples. The extra terms increase the size of the multiplier. PP_8 is required in case PP_7 is negative; this partial product is always 0 or *Y* because x_{16} and x_{17} are 0. Hence, partial product 8 is only 16 bits.

Observe that the sign extension bits are all either 1s or 0s. If a single 1 is added to the least significant position in a string of 1s, the result is a string of 0s plus a carry-out the top bit that may be discarded. Therefore, the large number of *s* bits in each partial product can be replaced by an equal number of constant 1s plus the inverse of *s* added to the least significant position, as shown in Figure 11.82(a). These constants mostly can be optimized out of the array by precomputing their sum. The simplified result is shown in Figure 11.82(b). As usual, it can be squashed to fit a rectangular floorplan.

The critical path of the multiplier involves the Booth decoder, the select line drivers, the Booth selector, approximately *N*/2 CSAs, and a final CPA. Each partial product fills about *M* + 5 columns. 54 × 54-bit radix-4 Booth multipliers for IEEE double-precision floating-point units are typically 20–50% smaller (and arguably up to 20% faster) than nonencoded counterparts, so the technique is widely used. The multiplier requires *M* × *N*/2 Booth selectors.

Because the selectors account for a substantial portion of the area and only a small fraction of the critical path, they should be optimized for size over speed. For example,

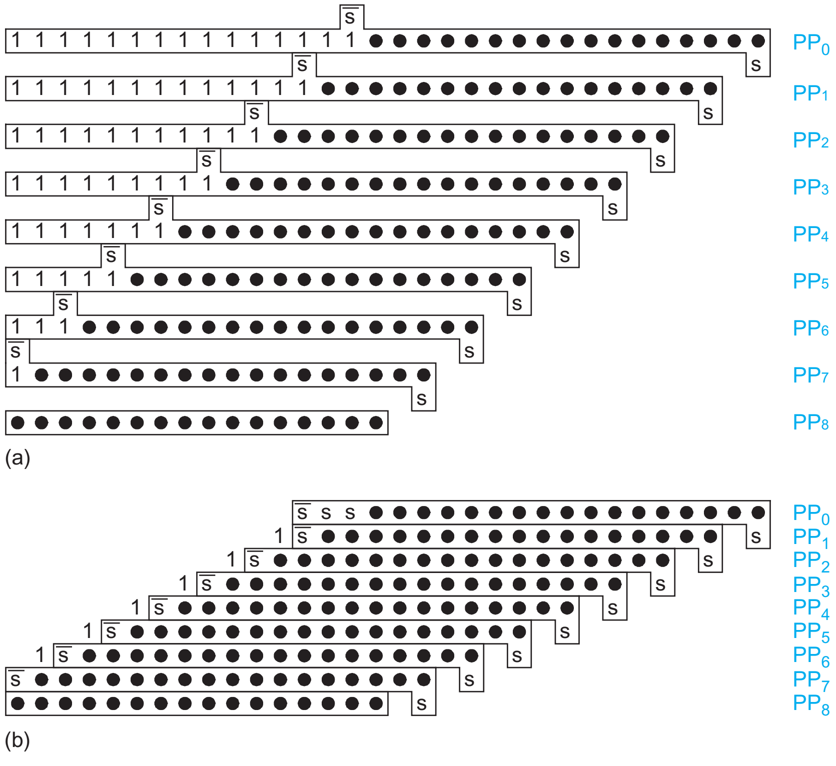


FIGURE 11.82 Radix-4 Booth-encoded partial products with simplified sign extension

[Goto97] describes a *sign select* Booth encoder and selector that uses only 10 transistors per selector bit at the expense of a more complex encoder. [Hsu06a] presents a *one-hot* Booth encoder and selector that chooses one of the six possible partial products using a transmission gate multiplexer. Exercise 11.18 explores yet another encoding.

11.9.3.1 Booth Encoding Signed Multipliers Signed two's complement multiplication is similar, but the multiplicand may have been negative so sign extension must be done based on the sign bit of the partial product, PP_{iM} [Bewick94]. Figure 11.83 shows such an array, where the sign extension bit is $e_i = PP_{iM}$. Also notice that PP_8, which was either Y or 0 for unsigned multiplication, is always 0 and can be omitted for signed multiplication because the multiplier x is sign-extended such that $x_{17} = x_{16} = x_{15}$. The same Booth selector and encoder can be employed (see Figure 11.80), but Y should be sign-extended rather than zero-extended to $M + 1$ bits.

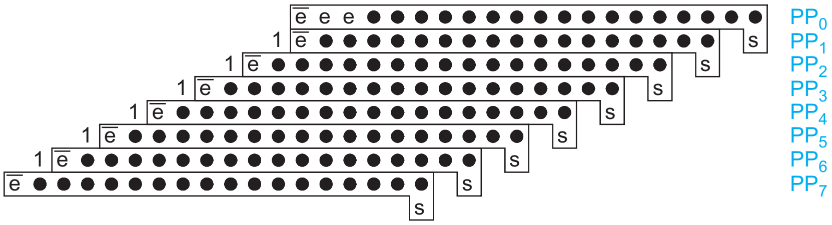


FIGURE 11.83 Radix-4 Booth-encoded partial products for signed multiplication

11.9.3.2 Higher Radix Booth Encoding Large multipliers can use Booth encoding of higher radix. For example, ordinary radix-8 multiplication reduces the number of partial products by a factor of 3, but requires hard multiples of $3Y$, $5Y$, and $7Y$. Radix-8 Booth-encoding only requires the hard $3Y$ multiple, as shown in Table 11.13. Although this requires a CPA before partial product generation, it can be justified by the reduction in array size and delay. Higher-radix Booth encoding is possible, but generating the other hard multiples appears not to be worthwhile for multipliers of fewer than 64 bits. Similar techniques apply to sign-extending higher-radix multipliers.

TABLE 11.13 Radix-8 modified Booth encoding values

x_{i+2}	x_{i+1}	x_i	x_{i-1}	Partial Product
0	0	0	0	0
0	0	0	1	Y
0	0	1	0	Y
0	0	1	1	$2Y$
0	1	0	0	$2Y$
0	1	0	1	$3Y$
0	1	1	0	$3Y$
0	1	1	1	$4Y$
1	0	0	0	$-4Y$
1	0	0	1	$-3Y$
1	0	1	0	$-3Y$

continues

TABLE 11.13 Radix-8 modified Booth encoding values (continued)

1	0	1	1	$-2Y$
1	1	0	0	$-2Y$
1	1	0	1	$-Y$
1	1	1	0	$-Y$
1	1	1	1	-0

11.9.4 Column Addition

The critical path in a multiplier involves summing the dots in each column. Observe that a CSA is effectively a "ones counter" that adds the number of 1s on the *A*, *B*, and *C* inputs and encodes them on the sum and carry outputs, as summarized in Table 11.14. A CSA is therefore also known as a *(3,2) counter* because it converts three inputs into a count encoded in two outputs [Dadda65]. The carry-out is passed to the next more significant column, while a corresponding carry-in is received from the previous column. This is called a *horizontal path* because it crosses columns. For simplicity, a carry is represented as being passed directly down the column. Figure 11.84 shows a dot diagram of an array multiplier column that sums *N* partial products sequentially using *N*–2 CSAs. For example, the 16 × 16 Booth-encoded multiplier from Figure 11.82(b) sums nine partial products with seven levels of CSAs. The output is produced in carry-save redundant form suitable for the final CPA.

TABLE 11.14 An adder as a ones counter

A	*B*	*C*	Carry	Sum	Number of 1s
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	1	1
0	1	1	1	0	2
1	0	0	0	1	1
1	0	1	1	0	2
1	1	0	1	0	2
1	1	1	1	1	3

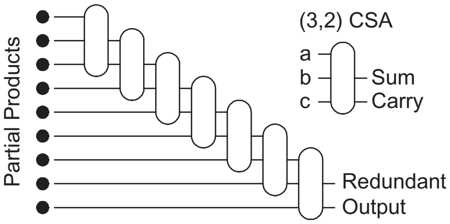


FIGURE 11.84 Dot diagram for array multiplier

The column addition is slow because only one CSA is active at a time. Another way to speed the column addition is to sum partial products in parallel rather than sequentially. Figure 11.85 shows a *Wallace tree* using this approach [Wallace64]. The Wallace tree requires

$$\left\lceil \log_{3/2}\left({}^{N}\!/_{2}\right)\right\rceil$$

levels of (3,2) counters to reduce *N* inputs down to two carry-save redundant form outputs.

Even though the CSAs in the Wallace tree are shown in two dimensions, they are logically packed into a single column of the multiplier. This leads to long and irregular wires along the column to connect the CSAs. The wire capacitance increases the delay and energy of multiplier, and the wires can be difficult to lay out.

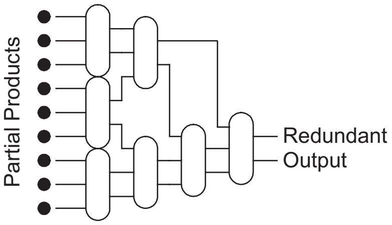


FIGURE 11.85 Dot diagram for Wallace tree multiplier

11.9.4.1 [4:2] Compressor Trees

[4:2] compressors can be used in a binary tree to produce a more regular layout, as shown in Figure 11.86 [Weinberger81, Santoro89]. A [4:2] compressor takes four inputs of equal weight and produces two outputs. It can be constructed from two (3,2) counters as shown in Figure 11.87. Along the way, it generates an intermediate carry, t_i, into the next column and accepts a carry, t_{i-1}, from the previous column, so it may more aptly be called a *(5,3) counter*. This horizontal path does not impact the delay because the output of the top CSA in one column is the input of the bottom CSA in the next column. The [4:2] CSA symbol emphasizes only the primary inputs and outputs to emphasize the main function of reducing four inputs to two outputs. Only

$$\left\lceil \log_2 \left({}^{N}\!/_{2} \right) \right\rceil$$

levels of [4:2] compressors are required, although each has greater delay than a CSA. The regular layout and routing also make the binary tree attractive.

To see the benefits of a [4:2] compressor, we introduce the notion of fast and slow inputs and outputs. Figure 11.88 shows a simple gate-level CSA design. The longest path through the CSA involves two levels of XOR2 to compute the sum. *X* is called a *fast input*, while *Y* and *Z* are *slow inputs* because they pass through a second level of XOR. *C* is the *fast output* because it involves a single gate delay, while *S* is the *slow output* because it involves two gate delays. A [4:2] compressor might be expected to use four levels of XOR2s. Figure 11.89 shows various [4:2] compressor designs that reduce the critical path to only 3 XOR2s. In Figure 11.89(a), the slow output of the first CSA is connected to the fast input of the second. In Figure 11.89(b), the [4:2] compressor has been munged into a single cell,

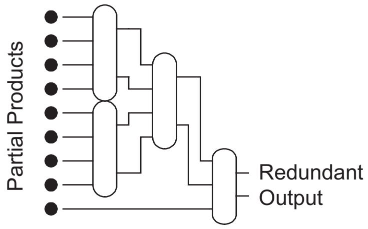


FIGURE 11.86 Dot diagram for [4:2] tree multiplier

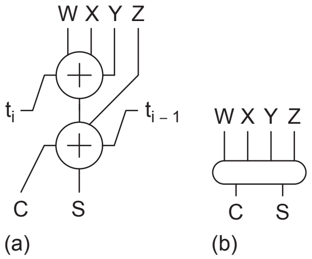


FIGURE 11.87 [4:2] compressor (a) implementation with two CSAs (b) symbol

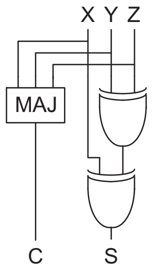


FIGURE 11.88 Gate-level carry-save adder

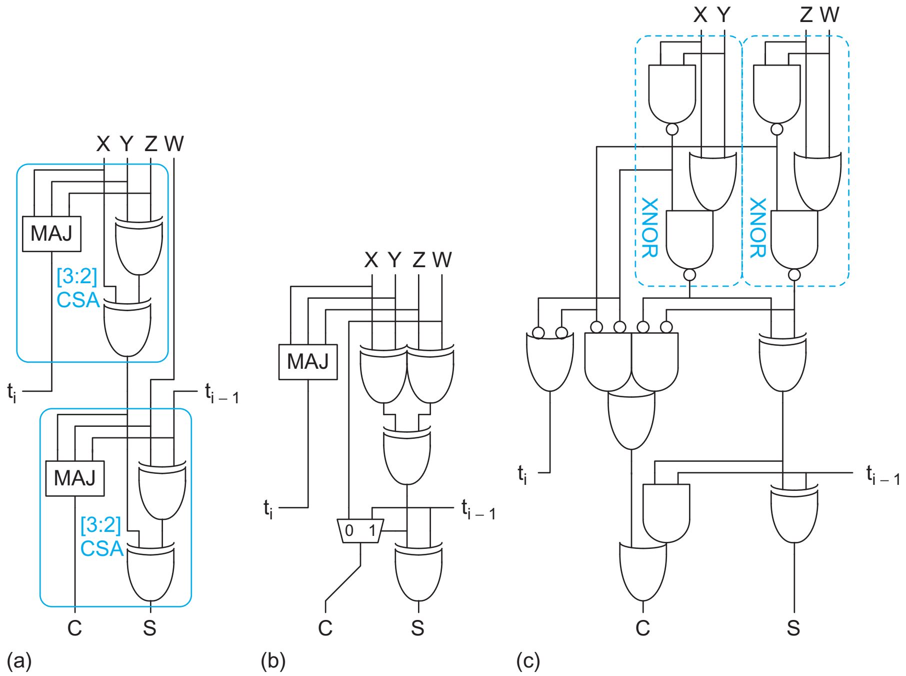


FIGURE 11.89 [4:2] compressors

allowing a majority gate to be replaced with a multiplexer. In Figure 11.89(c), the initial XORs have been replaced with 2-level XNOR circuits that allow some sharing of subfunctions, reducing the transistor count [Goto92].

Figure 11.90 shows a transmission gate implementation of a [4:2] compressor from [Goto97]. It uses only 48 transistors, allowing for a smaller multiplier array with shorter wires. Note that it uses three distinct XNOR circuit forms and two transmission gate multiplexers.

Figure 11.91 compares floorplans of the 16 × 16 Booth-encoded array multiplier from Figure 11.84, the Wallace tree from Figure 11.85, and the [4:2] tree from Figure 11.86. Each row represents a horizontal slice of the multiplier containing a Booth selector or a CSA. Vertical busses connect CSAs. The Wallace tree has the most irregular and lengthy wiring. In practice, the parallelogram may be squashed into a rectangular form to make better use of the space. [Itoh01n] and [Huang05] describes floorplanning issues in tree multipliers.

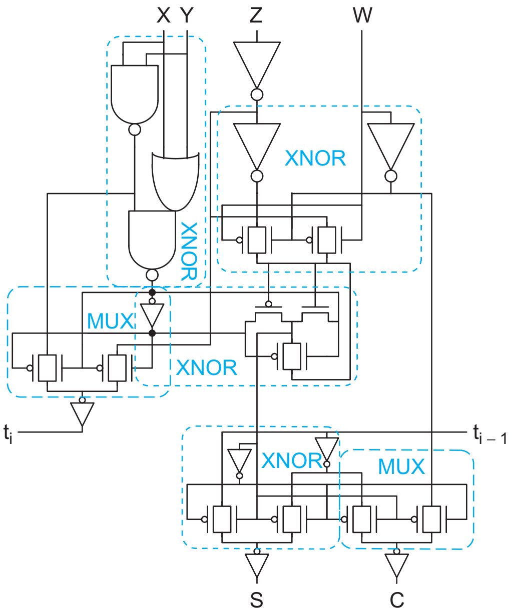


FIGURE 11.90 Transmission gate [4:2] compressor

11.9.4.2 Three-Dimensional Method The notion of connecting slow outputs to fast inputs generalizes to compressors with more than four inputs. By examining the entire partial product array at once, one can construct trees for each column that sum all of the partial products in the shortest possible time. This approach is called the *three-dimensional method* (TDM) because it considers the arrival time as a third dimension along with rows and columns [Oklobdzija96, Stelling98].

Figure 11.92 shows an example of a 16 × 16 multiplier. The parallelogram at the top shows the dot diagram from Figure 11.82(b) containing nine partial product rows obtained through Booth encoding. The partial products in each of the 32 columns must be summed to produce the 32-bit result. As we have seen, this is done with a compressor to produce a pair of outputs, followed by a final CPA.

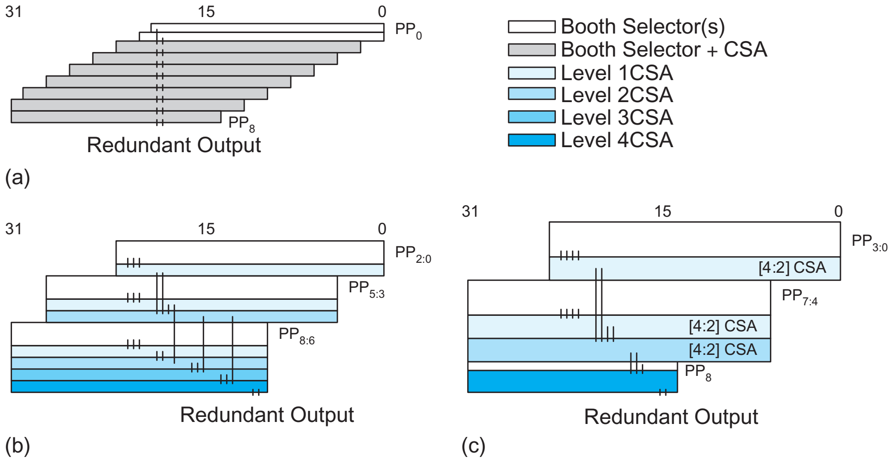


FIGURE 11.91 16 × 16 Booth-encoded multiplier floorplans: (a) array, (b) Wallace tree, (c) [4:2] tree

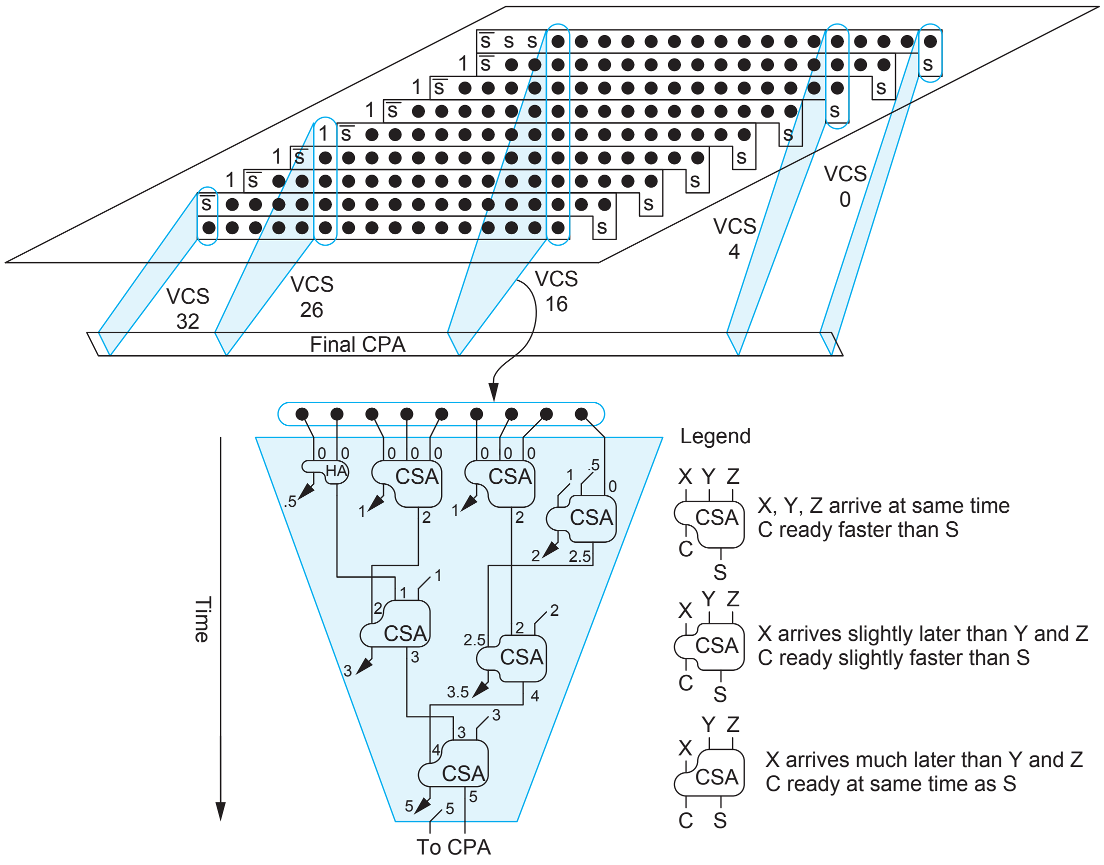

VCS
0
VCS
4
VCS
16
VCS
26
VCS
32
Final CPA
Time
HA
CSA
4:2
To CPA
Legend
X Y Z
C
S
X, Y, Z arrive at same time
C ready faster than S
X arrives slightly later than Y and Z
C ready slightly faster than S
X arrives much later than Y and Z
C ready at same time as S

FIGURE 11.92 Vertical compressor slices in a TDM multiplier

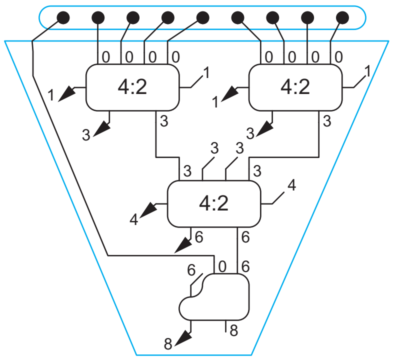


FIGURE 11.93 Vertical compressor slice using [4:2] compressors

In the three-dimensional method, each column is summed with a *vertical compressor slice* (VCS) made of CSAs. In Figure 11.92, VCS 16 adds nine partial products. In this diagram, the horizontal carries between compressor slices are shown explicitly.

Each wire is labeled with its arrival time. All partial product inputs arrive at time 0. The diagram assumes that an XOR2 and a majority gate each have unit delay. Thus, a path through a CSA from any input to *C* or from *X* to *S* takes one unit delay, and that a path from *Y* or *Z* to *S* takes two unit delays. A half adder is assumed to have half the delay. Horizontal carries are represented by diagonal lines coming from behind the slice or pointing out of the slice. VCS 16 receives five horizontal carries in from VCS 15 and produces six horizontal carries out to VCS 17. The final carry out is also shifted by one column before driving the CPA. The inputs to the CSAs are arranged based on their arrival times to minimize the delay of the multiplier. Note how the CSA shape is drawn to emphasize the asymmetric delays. Also, note that VCS 16 is not the slowest; some of the subsequent slices have one unit more delay because the horizontal carries arrive later. [Oklobdzija96] describes an algorithm for choosing the fastest arrangement of CSAs in each VCS given arbitrary CSA delays. In comparison, Figure 11.93 shows the same VCS 16 using [4:2] CSAs; more XOR levels are required but the wiring is more regular.

Table 11.15 lists the number of XOR levels on the critical path for various numbers of partial products. [4:2] trees offer a substantial improvement over Wallace trees in logic levels as well as wiring complexity. TDM generally saves one level of XOR over [4:2] trees, or more for very large multiplies. This savings comes at the cost of irregular wiring, so [4:2] trees and variants thereof remain popular.

TABLE 11.15 Comparison of XOR levels in multiplier trees

# Partial Products	Wallace Tree	4:2 Tree	TDM
8	8	6	5
9	8	8	6
16	12	9	8
24	14	11	10
32	16	12	11
64	20	15	14

11.9.4.3 Hybrid Multiplication Arrays offer regular layout, but many levels of CSAs. Trees offer fewer levels of CSAs, but less regular layout and some long wires. A number of hybrids have been proposed that offer trade-offs between these two extremes. These include *odd/even arrays* [Hennessy90], *arrays of arrays* [Dhanesha95], *balanced delay trees* [Zuras86], *overturned-staircase trees* [Mou90], and *upper/lower left-to-right leapfrog* (ULLRF) trees [Huang05]. They can achieve nearly as few levels of logic as the Wallace tree while offering more regular (and faster) wiring. None have caught on as distinctly better than [4:2] trees.

11.9.5 Final Addition

The output of the partial product array or tree is an $M + N$-bit number in carry-save redundant form. A CPA performs the final addition to convert the result back to nonredundant form.

The inputs to the CPA have nonuniform arrival times. As Figure 11.91 illustrated, the partial products form a parallelogram, with the middle columns having more partial products than the left or right columns. Hence, the middle columns arrive at the CPA later than the others. This can be exploited to simplify the CPA [Zimmermann96, Oklobdzija96]. Figure 11.94 shows an example of a 32-bit prefix network that takes advantage of nonuniform arrival times out of a 16 × 16-bit multiplier. The initial and final stages to compute bitwise PG signals and the sums are not shown. The path from the latest middle inputs to the output involves only four levels of cells. The total number of cells

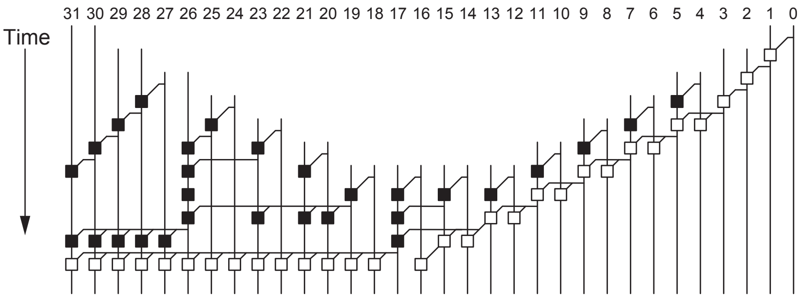


FIGURE 11.94 CPA prefix network with nonuniform input arrival times

and the energy consumption is much less than that of a conventional Kogge-Stone or Sklansky CPA.

OPTIONAL

11.9.6 Fused Multiply-Add

Many algorithms, particularly in digital signal processing, require computing $P = X \times Y + Z$. While this can be done with a multiplier and adder, it is much faster to use a *fused multiply-add* unit, which is simply an ordinary multiplier modified to accept another input Z that is summed just like the other partial products [Montoye90]. The extra partial product increases the delay of an array multiplier by just one extra CSA.

WEB ENHANCED

11.9.7 Serial Multiplication

This section is available in the online Web Enhanced chapter at `www.cmosvlsi.com`.

11.9.8 Summary

The three steps of multiplication are partial product generation, partial product reduction, and carry propagate addition. A simple $M \times N$ multiplier generates N partial products using AND gates. For multipliers of 16 or more bits, radix-4 Booth encoding is typically used to cut the number of partial products in two, saving substantial area and power. Some implementations find Booth encoding is faster, while others find it has little speed benefit. The partial products are then reduced to a pair of numbers in carry-save redundant form using an array or tree of CSAs. Trees have fewer levels of logic, but longer and less regular wiring; nevertheless most large multipliers use trees or hybrid structures. Pass transistor Booth selectors and CSAs were popular in the 1990s, but the trend is toward static CMOS as supply voltage scales. Finally, a CPA converts the result to nonredundant form. The CPA can be simplified based on the nonuniform arrival times of the bits.

Table 11.16 compares reported implementations of 54 × 54-bit multipliers for double-precision floating point arithmetic. All of the implementations use radix-4 Booth encoding.

TABLE 11.16 54 × 54-bit multipliers

Design	Process (μm)	PP Reduction	Circuits	Area (mm × mm)	Area (Mλ^2)	Transistors	Latency (ns)	Power (mW)
[Mori91]	0.5	4:2 tree	Pass Transistor XOR	3.6 × 3.5	200	82k	10	870
[Goto92]	0.8	4:2 tree	Static	3.4 × 3.9	80	83k	13	875
[Heikes94]	0.8	array	Dual-Rail Domino	2.1 × 2.2	28		20 (2-stage pipeline)	
[Ohkubo95]	0.25	4:2 tree	Pass Transistors	3.7 × 3.4	805	100k	4.4	
[Goto97]	0.25	4:2 tree	Pass Transistors	1.0 × 1.3	84	61k	4.1	
[Itoh01]	0.18	4:2 tree	Static	1 × 1	100		3.2 (2-stage pipeline)	
[Belluomini05]	90 nm	3:2 and 4:2 tree	LSDL	0.4 × 0.3	61			1800 @ 8 GHz
[Kuang05]	90 nm	3:2 and 4:2 tree	Pass Transistor and Domino	0.5 × 0.4	94			426 @ 4 GHz

11.10 Parallel-Prefix Computations

OPTIONAL

Many datapath operations involve calculating a set of outputs from a set of inputs in which each output bit depends on all the previous input bits. Addition of two N-bit inputs $A_N \ldots A_1$ and $B_N \ldots B_1$ to produce a sum output $Y_N \ldots Y_1$ is a classic example; each output Y_i depends on a carry-in c_{i-1} from the previous bit, which in turn depends on a carry-in c_{i-2} from the bit before that, and so forth. At first, this dependency chain might seem to suggest that the delay must involve about N stages of logic, as in a carry-ripple adder. However, we have seen that by looking ahead across progressively larger blocks, we can construct adders that involve only log N stages. Section 11.2.2.2 introduced the notion of addition as a prefix computation that involves a bitwise precomputation, a tree of group logic to form the prefixes, and a final output stage, shown in Figure 11.12. In this section, we will extend the same techniques to other prefix computations with associative group logic functions.

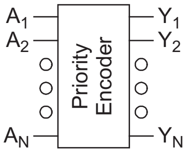


FIGURE 11.95 Priority encoder

Let us begin with the *priority encoder* shown in Figure 11.95. A common application of a priority encoder circuit is to arbitrate among N units that are all requesting access to a shared resource. Each unit i sends a bit A_i indicating a request and receives a bit Y_i indicating that it was granted access; access should only be granted to a single unit with highest priority. If the least significant bit of the input corresponds to the highest priority, the logic can be expressed as follows:

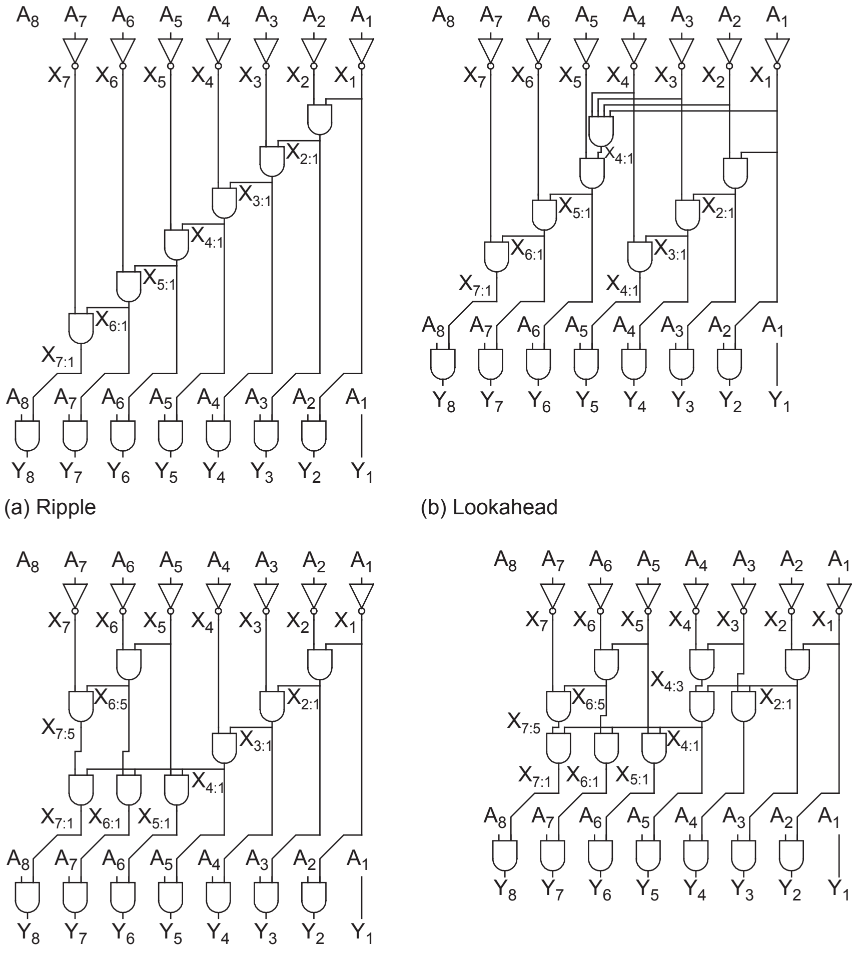
$$\begin{aligned} Y_1 &= A_1 \\ Y_2 &= A_2 \cdot \overline{A_1} \\ Y_3 &= A_3 \cdot \overline{A_2} \cdot \overline{A_1} \\ &\ldots \\ Y_N &= A_N \cdot \overline{A_{N-1}} \cdot \ldots \cdot \overline{A_1} \end{aligned} \qquad \textbf{(11.33)}$$

We can express priority encoding as a prefix operation by defining a prefix $X_{i:j}$ indicating that none of the inputs $A_i \ldots A_j$ are asserted. Then, priority encoding can be defined with bitwise precomputation, group logic, and output logic with $i \geq k > j$:

$$\begin{aligned} X_{i:i} &= \overline{A_i} && \text{bitwise precomputation} \\ X_{i:j} &= X_{i:k} \cdot X_{k-1:j} && \text{group logic} \\ Y_i &= A_i \cdot X_{i-1:1} && \text{output logic} \end{aligned} \qquad \textbf{(11.34)}$$

Any of the group networks (e.g., ripple, skip, lookahead, select, increment, tree) discussed in the addition section can be used to build the group logic to calculate the $X_{i:0}$ prefixes. Short priority encoders use the ripple structure. Medium-length encoders may use a skip, lookahead, select, or increment structure. Long encoders use prefix trees to obtain log N delay. Figure 11.96 shows four 8-bit priority encoders illustrating the different group logic. Each design uses an initial row of inverters for the $X_{i:i}$ precomputation and a final row of AND gates for the Y_i output logic. In between, ripple, lookahead, increment, and Sklansky networks form the prefixes with various trade-offs between gate count and delay. Compare these trees to Figure 11.15, Figure 11.22, Figure 11.25, and Figure 11.29(b), respectively. [Wang00, Delgado-Frias00, Huang02] describe a variety of priority encoder implementations.

An *incrementer* can be constructed in a similar way. Adding 1 to an input word consists of finding the least significant 0 in the word and inverting all the bits up to this point. The X prefix plays the role of the propagate signal in an adder. Again, any of the prefix networks can be used with varying area-speed trade-offs.



(a) Ripple

(b) Lookahead

(c) Increment

(d) Sklansky

FIGURE 11.96 Priority encoder trees

$$\begin{array}{ll} X_{i:i} = A_i & \text{bitwise precomputation} \\ X_{i:j} = X_{i:k} \cdot X_{k-1:j} & \text{group logic} \\ Y_i = A_i \oplus X_{i-1:1} & \text{output logic} \end{array} \qquad \textbf{(11.35)}$$

Decrementers and *two's complement* circuits are also similar [Hashemian92]. The decrementer finds the least significant 1 and inverts all the bits up to this point. The two's complement circuit negates a signed number by inverting all the bits above the least significant 1.

A binary-to-thermometer decoder is another application of a prefix computation. The input B is a k-bit representation of the number M. The output Y is a 2^k-bit number with the M most significant bits set to 1, as given in Table 11.17. A simple approach is to use an ordinary k:2^k decoder to produce a one-hot 2^k-bit word A. Then, the following prefix computation can be applied:

$$\begin{aligned} X_{i:i} &= A_{N-i} && \text{bitwise precomputation} \\ X_{i:j} &= X_{i:k} + X_{k-1:j} && \text{group logic} \\ Y_i &= X_{i:0} && \text{output logic} \end{aligned} \tag{11.36}$$

TABLE 11.17 Binary to thermometer decoder

B	*Y*
000	00000000
001	10000000
010	11000000
011	11100000
100	11110000
101	11111000
110	11111100
111	11111110

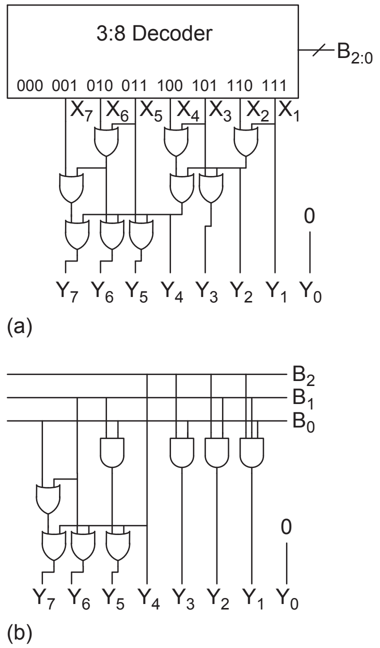

3:8 Decoder
$B_{2:0}$
000 001 010 011 100 101 110 111
X_7 X_6 X_5 X_4 X_3 X_2 X_1
0
Y_7 Y_6 Y_5 Y_4 Y_3 Y_2 Y_1 Y_0
(a)
B_2
B_1
B_0
0
Y_7 Y_6 Y_5 Y_4 Y_3 Y_2 Y_1 Y_0
(b)

FIGURE 11.97
Binary-to-thermometer decoders

Figure 11.97(a) shows an 8-bit binary-to-thermometer decoder using a Sklansky tree. The 3:8 decoder contains eight 3-input AND gates operating on true and complementary versions of the input. However, the logic can be significantly simplified by eliminating the complemented AND inputs, as shown in Figure 11.97(b)

In a slightly more complicated example, consider a modified priority encoder that finds the first two 1s in a string of binary numbers. This might be useful in a cache with two write ports that needs to find the first two free words in the cache. We will use two prefixes: X and W. Again, $X_{i:j}$ indicates that none of the inputs $A_i \ldots A_j$ are asserted. $W_{i:j}$ indicates exactly one of the inputs $A_i \ldots A_j$ is asserted. We will produce two 1-hot outputs, Y and Z, indicating the first two 1s.

$$\begin{aligned} X_{i:i} &= \overline{A_i} && \\ W_{i:i} &= A_i && \text{bitwise precomputation} \\ X_{i:j} &= X_{i:k} \cdot X_{k-1:j} && \\ W_{i:j} &= W_{i:k} \cdot X_{k-1:j} + X_{i:k} \cdot W_{k-1:j} && \text{group logic} \\ Y_i &= A_i \cdot X_{i-1:1} && \\ Z_i &= A_i \cdot W_{i-1:1} && \text{output logic} \end{aligned} \tag{11.37}$$

11.11 Pitfalls and Fallacies

Equating logic levels and delay

Comparing a novel design with the best existing design is difficult. Some engineers cut corners by merely comparing logic levels. Unfortunately, delay depends strongly on the logical effort of each stage, the fanout it must drive, and the wiring capacitance. For example, [Srinivas92] claims that a novel adder is 20–28% faster than the fastest known binary lookahead adder, but does not present simulation results. Moreover, it reports some of the speed advantages to three or four significant figures. On closer examination [Dobson95], the adder proves to just be a hybrid tree/carry-select design with some unnecessary precomputation.

Designing circuits with threshold drops

In modern processes, single-pass transistors that pull an output to $V_{DD} - V_t$ are generally unacceptable because the threshold drop (amplified by the body effect) results in an output with too little noise margin. Moreover, when they drive the gate terminals of a subsequent stage, the stage turns partially ON and consumes static power. Many 10-transistor full-adder cells have been proposed that suffer from such a threshold drop problem.

Reinventing adders

There is an enormous body of literature on adders with various trade-offs among speed, area, and power consumption. The design space has been explored fairly well and many designers (one of the authors included) have spent quite a bit of time developing a "new" adder, only to find that it is only a minor variation on an existing theme. Similarly, a number of recent publications on priority encoders reinvent prefix network techniques that have already been explored in the context of addition.

Summary

This chapter has presented a range of datapath subsystems. How one goes about designing and implementing a given CMOS chip is largely affected by the availability of tools, the schedule, the complexity of the system, and the final cost goals of the chip. In general, the simplest and least expensive (in terms of time and money) approach that meets the target goals should be chosen. For many systems, this means that synthesis and place & route is good enough. Modern synthesis tools draw on a good library of adders and multipliers with various area/speed trade-offs that are sufficient to cover a wide range of applications. For systems with the most stringent requirements on performance or density, custom design at the schematic level still provides an advantage. Domino parallel-prefix trees provide the fastest adders when the high power consumption can be tolerated. Domino CSAs are also used in fast multipliers. However, in multiplier design, the wiring capacitance is paramount and a multiplier with compact cells and short wires can be fast as well as small and low in power.

Exercises

11.1 Design a fast 8-bit adder. The inputs may drive no more than 30 λ of transistor width each and the output must drive a 20/10 λ inverter. Simulate the adder and determine its delay.

11.2 When adding two unsigned numbers, a carry-out of the final stage indicates an overflow. When adding two signed numbers in two's complement format, overflow detection is slightly more complex. Develop a Boolean equation for overflow as a function of the most significant bits of the two inputs and the output.

11.3 Repeat Exercise 11.2 for a signed add/subtract unit like that shown in Figure 11.41(b). Your overflow output should be a function of the subsignal and the most significant bits of the two inputs and the output.

11.4 Develop equations for the logical effort and parasitic delay with respect to the C_0 input of an n-stage Manchester carry chain computing $C_1 \ldots C_n$. Consider all of the internal diffusion capacitances when deriving the parasitic delay. Use the transistor widths shown in Figure 11.98 and assume the P_i and G_i transistors of each stage share a single diffusion contact.

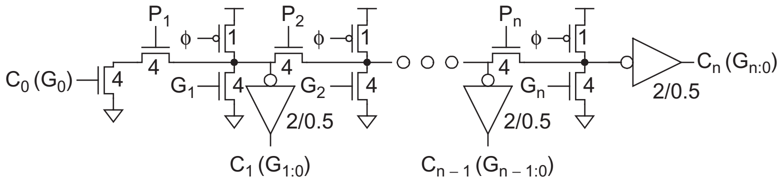


FIGURE 11.98 Manchester carry chain

11.5 Using the results of Exercise 11.4, what Manchester carry chain length gives the least delay for a long adder?

11.6 The carry increment adder in Figure 11.26(b) with variable block size requires five stages of valency-2 group PG cells for 16-bit addition. How many stages are required for 32-bit addition? For 64-bit addition?

11.7 Sketch the PG network for a modified 16-bit Sklansky adder with fanout of [8, 1, 1, 1] rather than [8, 4, 2, 1]. Use buffers to prevent the less-significant bits from loading the critical path.

11.8 Figure 11.29 shows PG networks for various 16-bit adders and Figure 11.30 illustrates how these networks can be classified as the intersection of the $l + f + t = 3$ plane with the face of a cube. The plane also intersects one point inside the cube at $(l, f, t) = (1, 1, 1)$ [Harris03]. Sketch the PG network for this 16-bit adder.

11.9 Sketch a diagram of the group PG tree for a 32-bit Ladner-Fischer adder.

11.10 Write a Boolean expression for C_{out} in the circuit shown in Figure 11.6(b). Simplify the equation to prove that the pass-transistor circuits do indeed compute the majority function.

11.11 Prove EQ (11.21).

11.12 Sketch a design for a comparator computing $A - B = k$.

11.13 Show how the layout of the parity generator of Figure 11.57 can be designed as a linear column of XOR gates with a tree-routing channel.

11.14 Design an ECC decoder for distance-3 Hamming codes with $c = 3$. Your circuit should accept a 7-bit received word and produce a 4-bit corrected data word. Sketch a gate-level implementation.

11.15 How many check bits are required for a distance-3 Hamming code for 8-bit data words? Sketch a parity-check matrix and write the equations to compute each of the check bits.

11.16 Find the 4-bit binary-reflected Gray code values for the numbers 0–15.

11.17 Design a Gray-coded counter in which only one bit changes on each cycle.

11.18 Table 11.12 and Figure 11.80 illustrated radix-4 Booth encoding using *SINGLE*, *DOUBLE*, and *NEG*. An alternative encoding is to use *POS*, *NEG*, and *DOUBLE*. *POS* is true for the multiples *Y* and 2*Y*. *NEG* is true for the multiples –*Y* and –2*Y*. *DOUBLE* is true for the multiples 2*Y* and –2*Y*. Design a Booth encoder and selector using this encoding.

11.19 Adapt the priority encoder logic of EQ (11.37) to produce three 1-hot outputs corresponding to the first three 1s in an input string.

11.20 Sketch a 16-bit priority encoder using a Kogge-Stone prefix network.

11.21 Use Logical Effort to estimate the delay of the priority encoder from Exercise 11.20. Assume the path electrical effort is 1.

11.22 Write equations for a prefix computation that determines the second location in which the pattern 10 appears in an *N*-bit input string. For example, 010010 should return 010000.

11.23 [Jackson04] proposes an extension of the Ling adder formulation to simplify cells later in the prefix network. Design a 16-bit adder using this technique and compare it to a conventional 16-bit Ling adder.