COMPUTER ARITHMETIC ALGORITHMS

ISRAEL KOREN

University of Massachusetts, Amherst

PRENTICE HALL, Englewood Cliffs, New Jersey 07632

5 FAST ADDITION

5.1 TWO-OPERAND ADDERS

The addition of two operands is the most frequent operation in almost any arithmetic unit. A two-operand adder is not only used when performing additions and subtractions but is also often employed when executing more complex operations like multiplication and division. Consequently, a fast two-operand adder is essential.

The most straightforward implementation of a parallel adder for two operands $x_{n-1}, x_{n-2} \cdots, x_0$ and $y_{n-1}, y_{n-2} \cdots, y_0$ is through the use of n basic units called *full adders*. A full adder (FA) is a logical circuit that accepts two operand bits, say x_i and y_i, and an incoming carry bit, denoted by c_i, and then produces the corresponding sum bit, denoted by s_i, and an outgoing carry bit, denoted by c_{i+1}. As this notation suggests, the outgoing carry c_{i+1} is also the incoming carry for the subsequent FA, which has x_{i+1} and y_{i+1} as input bits. The FA is a combinatorial digital circuit implementing the binary addition of three bits through the following Boolean equations:

$$s_i = x_i \oplus y_i \oplus c_i \tag{5.1}$$

where $\oplus$ is the exclusive-or operation, and

$$c_{i+1} = x_i \cdot y_i + c_i \cdot (x_i + y_i) \tag{5.2}$$

where $x_i \cdot y_i$ is the AND operation, $x_i \wedge y_i$, and $x_i + y_i$ is the OR operation, $x_i \vee y_i$.

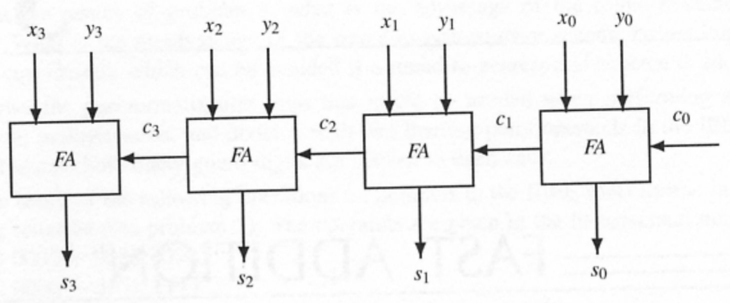

Figure 5.1 A 4-bit ripple-carry adder.

A parallel adder consisting of FAs for $n = 4$ is depicted in Figure 5.1. In a parallel arithmetic unit, all $2n$ input bits (x_i and y_i) are usually available to the adder at the same time. However, the carries have to propagate from the FA in position 0 (the position of the FA whose inputs are x_0 and y_0) to position i in order for the FA in that position to produce the correct sum and carry-out bits. In other words, we need to wait until the carries *ripple* through all n FAs before we can claim that the sum outputs are correct and may be used in further calculations. Because of this, the parallel adder shown in Figure 5.1 is called a *ripple-carry adder.* Note that the FA in position i, being a combinatorial circuit, will see an incoming carry $c_i = 0$ at the beginning of the operation, and will accordingly produce a sum bit s_i. The incoming carry c_i may change later on, resulting in a corresponding change in s_i. Thus, a ripple effect can be observed at the sum outputs of the adder as well, continuing until the carry propagation is complete. Also, notice that in an add operation, the incoming carry in position 0, c_0, is always zero and, as a result, one may replace the FA in this position by a simpler unit that is capable of adding only two bits. Such a circuit is called a half adder (HA) and its Boolean equations can be obtained from equations (5.1) and (5.2) by setting c_i equal to 0. Still, an FA is frequently used to enable us to add a 1 in the least-significant position (*ulp*). This is needed to implement a subtract operation in the two's complement method. Here, the subtrahend is complemented and then added to the minuend. This is accomplished by taking the one's complement of the subtrahend and adding a *forced* carry to the FA in position 0 by setting $c_0 = 1$.

Example 5.1

Consider the following two operands for the adder in Figure 5.1: $x_3, x_2, x_1, x_0 = 1111$ and $y_3, y_2, y_1, y_0 = 0001$. Δ_{FA} denotes the operation time (delay) of an FA, assuming that the delays associated with generating the sum output and the carry-out are equal. This may be the case if, for example, both circuits use a two-level gate implementation. The following diagram shows the sum and carry signals as a function of the time, T, measured in Δ_{FA} units:

$T = 0$		1111
	+	0001
$T = \Delta_{FA}$	Carry	0001
	Sum	1110
$T = 2\Delta_{FA}$	Carry	0011
	Sum	1100
$T = 3\Delta_{FA}$	Carry	0111
	Sum	1000
$T = 4\Delta_{FA}$	Carry	1111
	Sum	0000

This is the longest carry propagation chain that can occur when adding two 4-bit numbers. In synchronous arithmetic units, the time allowed for the adder's operation must be the worst-case delay, which is, in the general case, $n \cdot \Delta_{FA}$. This means that the adder is assumed to produce the correct sum after this fixed delay, regardless of the actual carry propagation time, which might be very short, as in 0101+0010.

Consider now the subtract operation 0101 − 0010, which is performed by adding the two's complement of the subtrahend to the minuend. The two's complement is formed by taking the one's complement of 0010, which is 1101, and setting the forced carry, c_0, to 1 yielding 0011. □

It is clear that the long carry propagation chains must be dealt with in order to speed up the addition. Two main approaches can be envisioned: One is to reduce the carry propagation time; the other is to detect the completion of the carry propagation and avoid wasting time while waiting for the fixed delay (of $n \cdot \Delta_{FA}$ for ripple-carry adders) unless absolutely necessary. Clearly, the second approach leads to a variable addition time, which may be inconvenient in a synchronous design. We therefore concentrate on the first approach and study several schemes for accelerating carry propagation. The technique for detection of carry completion is left to the reader as an exercise.

5.2 CARRY-LOOK-AHEAD ADDERS

The most commonly used scheme for accelerating carry propagation is the *carry-look-ahead* scheme. The main idea behind carry-look-ahead addition is an attempt to generate all incoming carries in parallel (for all the $n - 1$ high order FAs) and avoid the need to wait until the correct carry propagates from the stage (FA) of the adder where it has been generated. This can be accomplished in principle, since the carries generated and the way they propagate depend only on the digits of the original numbers $x_{n-1}x_{n-2} \cdots x_0$ and $y_{n-1}y_{n-2} \cdots y_0$. These digits are available simultaneously to all stages of the adder and, consequently, each stage can have all the information it needs in order to calculate the correct value of the incoming carry and compute the sum bit accordingly. This, however, would require a large number of inputs to each stage of the adder, rendering it impractical.

One may reduce the number of inputs at each stage by extracting from the input digits the information needed to determine whether new carries will be generated and whether they will be propagated. To this end, we will study in detail the generation and propagation of carries.

There are stages in the adder in which a carry-out is generated regardless of the incoming carry, and as a result, no additional information on previous input digits is required. These are the stages for which $x_i = y_i = 1$. There are other stages that are only capable of propagating the incoming carry; i.e., $x_i y_i = 10$, or $x_i y_i = 01$. Only a stage in which $x_i = y_i = 0$ cannot propagate a carry. To assimilate the information regarding the generation and propagation of carries, we define the following logic functions using the AND and OR operations. Let $G_i = x_i \cdot y_i$ denote the *generated carry* and let $P_i = x_i + y_i$ denote the *propagated carry*. As a result, the Boolean expression (5.2) for the carry-out can be rewritten as

$$c_{i+1} = x_i y_i + c_i(x_i + y_i) = G_i + c_i \cdot P_i$$

Substituting $c_i = G_{i-1} + c_{i-1}P_{i-1}$ in the above expression yields

$$c_{i+1} = G_i + G_{i-1}P_i + c_{i-1}P_{i-1}P_i$$

Further substitutions result in

$$\begin{aligned} c_{i+1} &= G_i + G_{i-1}P_i + G_{i-2}P_{i-1}P_i + c_{i-2}P_{i-2}P_{i-1}P_i = \cdots \\ &= G_i + G_{i-1}P_i + G_{i-2}P_{i-1}P_i + \cdots + c_0P_0P_1 \cdots P_i \end{aligned} \tag{5.3}$$

This type of expression allows us to calculate all the carries in *parallel* from the original digits $x_{n-1}x_{n-2}\cdots x_0$ and $y_{n-1}y_{n-2}\cdots y_0$ and the forced carry c_0. For example, for a 4-bit adder, the carries are

$$\begin{aligned} c_1 &= G_0 + c_0P_0 \\ c_2 &= G_1 + G_0P_1 + c_0P_0P_1 \\ c_3 &= G_2 + G_1P_2 + G_0P_1P_2 + c_0P_0P_1P_2 \\ c_4 &= G_3 + G_2P_3 + G_1P_2P_3 + G_0P_1P_2P_3 + c_0P_0P_1P_2P_3 \end{aligned} \tag{5.4}$$

If this is done for all stages of the adder, then for each stage a Δ_G delay is required to generate all P_i and G_i, where Δ_G is the delay of a single gate. A delay of $2\Delta_G$ is then needed to generate all c_i (assuming a two-level gate implementation) and another $2\Delta_G$ to generate the sum digits, s_i, in parallel (again, assuming a two-level gate implementation). Hence, a total of $5\Delta_G$ time units is needed, regardless of n, the number of bits in each operand. However, for a large value of n, say, $n = 32$, an extremely large number of gates is needed and, more importantly, gates with a very large fan-in are required (fan-in is the number of gate inputs, and is equal to $n + 1$ in this case). Therefore, we must reduce the span of the look-ahead at the expense of speed. We may divide the n stages into groups and have a separate carry-look-ahead in each group.

The groups can then be interconnected by the ripple-carry method. Dividing the adder into equal-sized groups has the additional benefit of modularity, requiring the detailed design of only a single integrated circuit. A group size of 4 is commonly used, and ICs capable of adding two sequences, each consisting of four digits with carry-look-ahead, are available. Size 4 was selected because it is a common factor of most word sizes, and also because of technology-dependent constraints (e.g., the available number of input/output pins).

For n bits and groups of size 4, there are $n/4$ groups. To propagate a carry through a group once the P_i's, G_i's, and c_0 are available, we need $2\Delta_G$ time units. Thus, $1\Delta_G$ is needed to generate all P_i and G_i, $(n/4)\cdot 2\Delta_G$ are needed to propagate the carry through all bits, and an additional delay of $2\Delta_G$ is needed to generate the sum outputs, for a total of $(2\frac{n}{4}+3)\Delta_G = (\frac{n}{2}+3)\Delta_G$. This is almost a fourfold reduction in delay compared to the $2n\Delta_G$ delay of a ripple-carry adder.

We may further speed up the addition by providing a carry-look-ahead over groups in addition to the internal look-ahead within the group. We define a *group-generated carry,* G^*, and a *group-propagated carry,* P^*, for a group of size 4 as follows: $G^* = 1$ if a carry-out (of the group) is generated internally and $P^* = 1$ if a carry-in (to the group) is propagated internally to produce a carry-out (of the group). The Boolean equations for these carries are

$$\begin{aligned} G^* &= G_3 + G_2P_3 + G_1P_2P_3 + G_0P_1P_2P_3 \\ P^* &= P_0P_1P_2P_3 \end{aligned} \tag{5.5}$$

The group-generated and group-propagated carries for several groups can now be used to generate group carry-ins in a manner similar to single-bit carry-ins in equation (5.4). A combinatorial circuit implementing these equations is available as a separate and standard IC. This IC is called a *carry-look-ahead generator,* and its use is illustrated in the following example.

Example 5.2

For $n = 16$ there are four groups, with outputs G_0^*, G_1^*, G_2^*, G_3^* and P_0^*, P_1^*, P_2^*, P_3^*. These serve as inputs to a carry-look-ahead generator, whose outputs are denoted by c_4, c_8, and c_{12}, satisfying

$$\begin{aligned} c_4 &= G_0^* + c_0P_0^* \\ c_8 &= G_1^* + G_0^*P_1^* + c_0P_0^*P_1^* \\ c_{12} &= G_2^* + G_1^*P_2^* + G_0^*P_1^*P_2^* + c_0P_0^*P_1^*P_2^* \end{aligned} \tag{5.6}$$

A 16-bit adder with four groups, each with internal carry-look-ahead and an additional carry-look-ahead generator, are depicted in Figure 5.2. The operation of this adder consists of the following four steps:

1. All groups generate in parallel bit-carry-generate, G_i, and bit-carry-propagate, P_i.

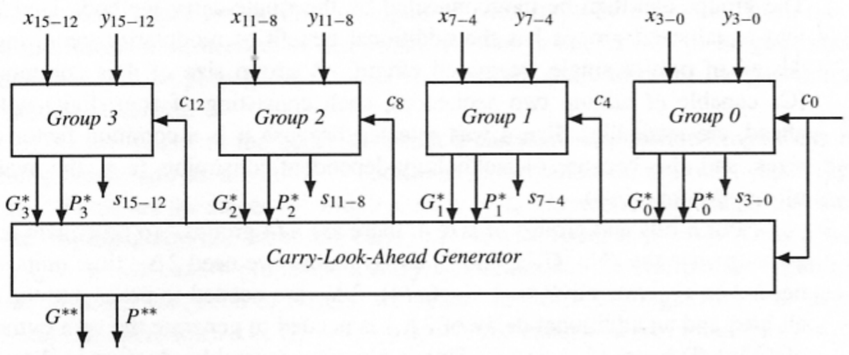

Figure 5.2 A 16-bit two-level carry-look-ahead adder. (The notation x_{3-0} means x_3, x_2, x_1, x_0.)

2. All groups generate in parallel group-carry-generate, G_i^*, and group-carry-propagate, P_i^*.
3. The carry-look-ahead generator produces the carries c_4, c_8, and c_{12} into the groups.
4. The groups calculate their individual sum bits (in parallel) with internal carry-look-ahead. In other words, they first generate the internal carries according to equation (5.4) and then the sum bits.

The minimum time delay associated with the steps (1)–(4) (assuming a minimum number of gate levels in all circuits) is $1\Delta_G$ for step 1, $2\Delta_G$ for step 2, $2\Delta_G$ for step 3, and $4\Delta_G$ for step 4. Thus, the total addition time is $9\Delta_G$ instead of $11\Delta_G$, which is the addition time if the external carry-look-ahead generator is not used and the carry ripples among the groups. These calculations yield only theoretical estimates for the addition time. In practice, one has to use the typical delays associated with the particular integrated circuits employed in order to calculate the addition time more accurately (see any integrated circuit databook). □

As shown in Figure 5.2, the carry-look-ahead generator produces two additional outputs, G^{**} and P^{**}, whose Boolean equations are similar to those in equation (5.5). These new outputs are called *section-carry generate* and *section-carry propagate*, respectively, where a section, in this case, is a set of four groups and consists of 16 bits. As before, the number of groups in a section is commonly set at four because of implementation-related considerations, and not because of any limitation of the underlying algorithm.

If the number of bits to be added is larger than 16, say, 64, we may use either four circuits, each similar to the one shown in Figure 5.2, with a ripple-carry between adjacent sections, or use another level of carry-look-ahead, and achieve a faster execution of addition. This is exactly the same circuit as above, accepting the four pairs of section-carry-generate and section-carry-propagate, and producing the carries c_{16}, c_{32}, and c_{48}.

As the number of bits, n, increases, more levels of carry-look-ahead generators can be added in order to speed up the addition. The required number of levels (for maximum speed up) approaches $\log_b n$, where b is the *blocking factor*; i.e., the number of bits in a group, the number of groups in a section, and so on. The blocking factor is 4 in the conventional implementation depicted in Figure 5.2. The overall addition time of a carry-look-ahead adder is therefore proportional to $\log_b n$.

5.6 CARRY-SAVE ADDERS

When three or more operands are to be added simultaneously (e.g., in multiplication) using two-operand adders, the time-consuming carry-propagation must be repeated several times. If the number of operands is k, then carries have to propagate $(k-1)$ times. Several techniques for multiple operand addition that attempt to lower the carry-propagation penalty have been proposed and implemented. The technique that is most commonly used is *carry-save addition.* In carry-save addition, we let the carry propagate only in the last step, while in all the other steps we generate a partial sum and a sequence of carries separately. Thus, a carry-save adder (CSA) accepts three n-bit operands and generates two n-bit results, an n-bit partial sum, and an n-bit carry. A second CSA accepts these two bit-sequences and another input operand, and generates a new partial sum and carry. A CSA is therefore, capable of reducing the number of operands to be added from 3 to 2, without any carry propagation.

A carry-save adder may be implemented in several different ways. In the simplest implementation, the basic element of the carry-save adder is a full adder with three inputs, x, y, and z, whose arithmetic operation can be described by

$$x + y + z = 2c + s \tag{5.10}$$

where s and c are the sum and carry outputs, respectively. Their values are

$$s = (x + y + z) \; mod \; 2 \qquad \text{and} \qquad c = \frac{(x + y + z) - s}{2} \tag{5.11}$$

The outputs are the weighted binary representation of the number of 1's in the inputs. We therefore call the FA a (3, 2) counter, as shown in Figure 5.8. An n-bit CSA consists of n (3,2) counters operating in parallel with no carry links interconnecting them.

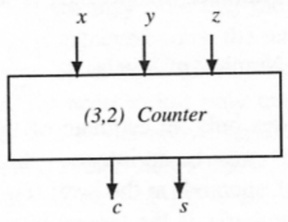

Figure 5.8 A (3,2) counter.

A carry-save adder for four 4-bit operands X, Y, Z, and W, is shown in Figure 5.9. The upper two levels are 4-bit CSAs, while the third level is a 4-bit carry-propagating adder (CPA). The latter is a ripple-carry adder, but may be replaced by a carry-look-ahead adder or any other fast CPA. One should note that partial sum bits and carry bits are interconnected to guarantee that only bits having the same weight are added by any (3,2) counter.

In order to add the k operands $X_1, X_2, \cdots, X_k$ we need $(k-2)$ CSA units and one CPA. If the CSAs are arranged in a cascade, as in Figure 5.9, then the time to add the k operands is

$$(k-2) \cdot T_{CSA} + T_{CPA}$$

where T_{CPA} is the operation time of a CPA and T_{CSA} is the operation time of a CSA, which equals the delay of a full adder, Δ_{FA}. The latter is at least $2 \cdot \Delta_G$, where Δ_G is

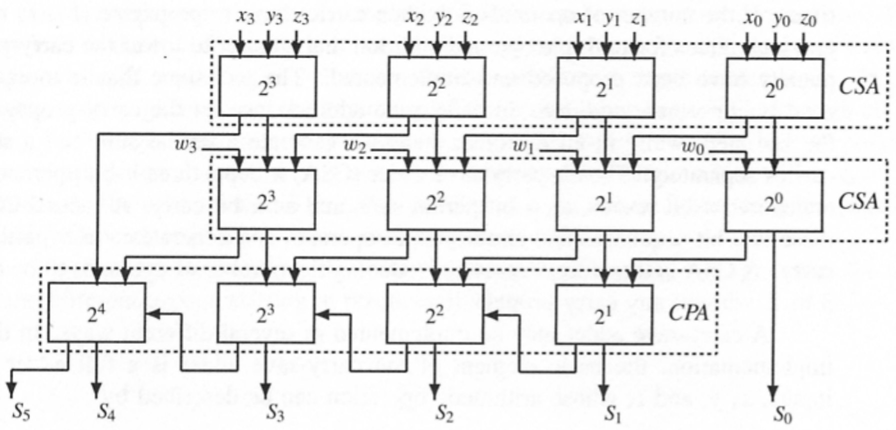

Figure 5.9 A carry-save adder for four operands.

the delay of a single gate. Note that the final result may reach a length of $n + \lceil log_2 k \rceil$ bits, since the sum of k operands, of size n bits each, can be as large as $(2^n - 1)k$.

A better way to organize the CSAs, and reduce the operation time, is in the form of a tree commonly called a Wallace tree [18]. A six-operand Wallace tree is illustrated in Figure 5.10. The left arrows on the carry outputs of the CSAs indicate that these outputs have to be shifted to the left before being added to the sum bits, as shown in Figure 5.9. In this tree, the number of operands is reduced by a factor of 2/3 at each level. Consequently,

$$\text{Number of levels} \approx \frac{log\ (k/2)}{log\ (3/2)} \tag{5.12}$$

Equation (5.12) provides only an estimate of the number of levels, since at each level the number of operands must be an integer. Thus, if N_i is the number of operands at level i, then the number of operands at the level $(i+1)$ above can be at most $\lfloor N_i \cdot 3/2 \rfloor$ (where the floor $\lfloor x \rfloor$ of a number x is the largest integer that is smaller than or equal to x). The number of operands at the bottom level (i.e., level 0) is 2, so that the maximum number of operands at level 1 is 3 and the maximum number of operands at level 2 is $\lfloor 9/2 \rfloor = 4$. The resulting sequence of numbers is 2,3,4,6,9,13,19,28, etc. Starting with

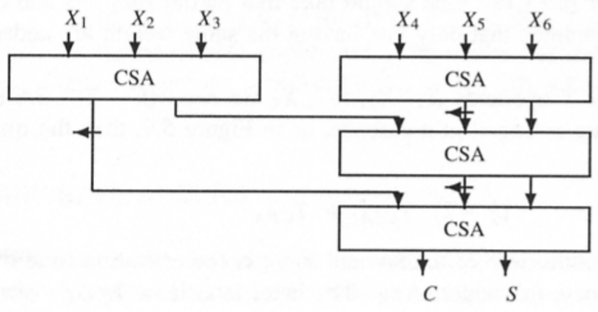

Figure 5.10 A CSA tree for six operands.

five operands, we still need three levels as we do for six operands. The entries in Table 5.1 were generated using similar arguments. This table shows the exact number of levels required for up to 63 operands.

TABLE 5.1 THE NUMBER OF LEVELS IN A CSA TREE FOR k OPERANDS.

Number of operands	Number of levels
3	1
4	2
$5 \leq k \leq 6$	3
$7 \leq k \leq 9$	4
$10 \leq k \leq 13$	5
$14 \leq k \leq 19$	6
$20 \leq k \leq 28$	7
$29 \leq k \leq 42$	8
$43 \leq k \leq 63$	9

Example 5.6

For $k = 12$, five levels are needed, resulting in a delay of $5 \cdot T_{CSA}$, instead of $10 \cdot T_{CSA}$, which is the delay for a linear cascade of 10 CSAs. □

Examining Table 5.1, we may note that the most economical implementation (in terms of number of levels) is achieved when the number of operands is an element of the series 3,4,6,9,13,19,28, $\cdots$. Thus, for a given number of operands, say, k, which is not an element of this series, we need to use only enough CSAs to reduce k to the closest (and smaller than k) element in the above series. For example, for $k = 27$, we may use 8 CSAs (with 24 inputs) rather than 9 CSAs, in the top level, so that the number of operands in the next level will be $8 \cdot 2 + 3 = 19$, which is an element of the series. The remaining part of the tree will have its operands follow the series.

The idea of using a (3,2) counter to form multi-operand adders can be extended to a (7,3) counter, whose three outputs represent the number of 1's in its seven inputs. Another example is the (15,4) counter or, in general, any (k, m) counter where k and m satisfy

$$2^m - 1 \geq k \quad \text{or} \quad m \geq \lceil log_2(k+1) \rceil$$

A (7,3) counter, for example, can be implemented using (3,2) counters as shown in Figure 5.11, where intermediate results are added according to their weight. However, this implementation requires four (3,2) counters arranged in three levels and therefore provides no speed-up compared to an implementation based on (3,2) counters. A (7,3) counter can also be implemented directly as a multilevel circuit that may have a smaller overall delay depending on the particular technology employed [10]. A different implementation of the (7,3) counter is through a ROM of size $2^7 \times 3 = 128 \times 3$ bits. The access time of this ROM is unlikely to be smaller than the delay associated with the implementation in

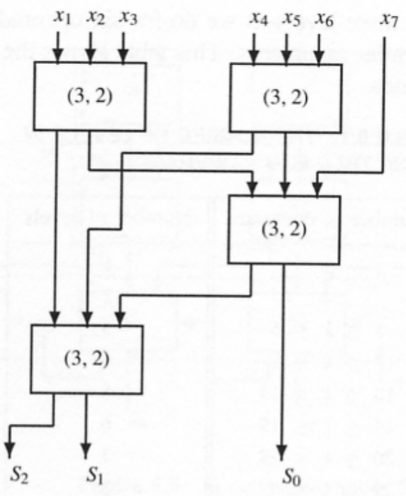

Figure 5.11 A (7,3) counter using (3,2) counters.

Figure 5.11. However, a speed-up may be achieved if a ROM implementation is used for a (k, m) counter with higher values of k and m.

When several (7,3) counters (in parallel) are used to add seven operands, we obtain three results, and a second level of (3,2) counters is needed to reduce these to two results (sum and carry) to be added by a CPA. A similar situation arises when (15,4) or more complex counters are used, generating more than two results and consequently requiring a second level of counters. In some cases, the additional level of counters can be combined with the first level of counters, resulting in a more convenient implementation.

In what follows, we show how the (7,3) counter can be combined with a (3,2) counter. We call the combined counter a (7,2) counter. A straightforward implementation of a (7,2) counter is shown in Figure 5.12, where the bottom right (3,2) counter is the additional (3,2) counter, while the remaining four (3,2) counters constitute the ordinary (7,3) counter that is depicted in Figure 5.11. A (7,2) counter in column i has seven primary inputs of weight 2^i and two carry inputs from column $(i-1)$ and $(i-2)$. It generates two primary outputs, denoted by $S2^i$ and $S2^{i+1}$, reflecting their weights, and two outgoing carries $C2^{i+1}$ and $C2^{i+2}$, to columns $(i+1)$ and $(i+2)$, respectively. Note that the input carries to the (7,2) counter in Figure 5.12 do not participate in the generation of the two output carries in order to avoid a slow carry-propagation chain.

All the previously described counters are single-column counters. For multi-operand addition we can generalize these single-column counters into multiple-column counters. We define a generalized parallel counter as a counter that adds l input columns and produces an m-bit output [16]. The notation we use for such a counter is

$$(k_{l-1}, k_{l-2}, \cdots, k_o, m)$$

where k_i is the number of input bits in the i–th column with weight 2^i. Clearly, a (k, m) counter is a special case of this generalized counter. The number of outputs m

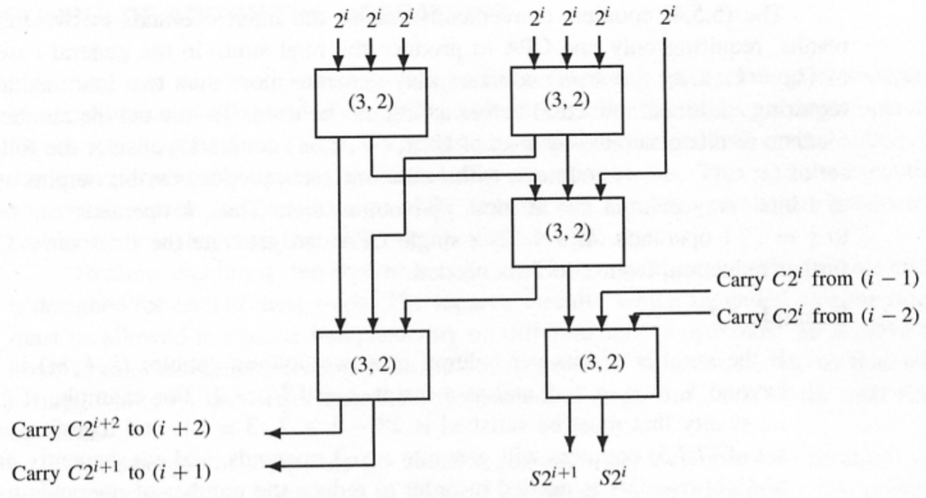
2^i 2^i 2^i 2^i 2^i 2^i 2^i
(3, 2)
(3, 2)
(3, 2)
Carry $C2^i$ from $(i-1)$
Carry $C2^i$ from $(i-2)$
(3, 2)
(3, 2)
Carry $C2^{i+2}$ to $(i+2)$
Carry $C2^{i+1}$ to $(i+1)$
$S2^{i+1}$ $S2^i$

Figure 5.12 A (7,2) counter with two carries, in bit position i.

must satisfy

$$2^m - 1 \geq \sum_{i=0}^{l-1} k_i 2^i \tag{5.13}$$

If all l columns have the same height k (i.e., $k_0 = k_1 = \ldots = k_{l-1} = k$), then the inequality that has to be satisfied is

$$2^m - 1 \geq \overset{5}{k} \cdot (\overset{3}{2^l} - 1) \tag{5.14}$$

A simple example of these counters is the (5,5,4) counter shown in Figure 5.13. For this counter, $k = 5$, $l = 2$ and $m = 4$, and inequality (5.14) turns into an equality, implying that all 16 combinations of the output bits are useful. (5,5,4) counters can be used to reduce five operands (of any length) to two results that can then be added with a CPA. The length of operands will determine the number of (5,5,4) counters in parallel.

A reasonable way of implementing generalized counters is by using ROMs. For example, the (5,5,4) counter shown in Figure 5.13 can be realized with a $2^{(5+5)} \times 4$ ROM (i.e., 1024×4).

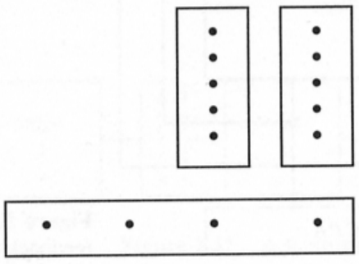

Figure 5.13 A (5,5,4) counter. The dots represent input or output bits.

The (5,5,4) counters conveniently reduce the input operands to two intermediate results, requiring only one CPA to produce the final sum. In the general case, a string of $(k_0 = k, ..., k_{l-1} = k, m)$ counters may generate more than two intermediate results, requiring additional reduction before a CPA can be used. To find out the number of intermediate results generated by a set of $(k, k, \cdots, k, m)$ counters, consider the following. A set of $(k, k, \cdots, k, m)$ counters, with l columns each, produces m-bit outputs at intervals of l bits. Any column has at most $\lceil \frac{m}{l} \rceil$ output bits. Thus, k operands can be reduced to $s = \lceil \frac{m}{l} \rceil$ operands. If $s = 2$, a single CPA can generate the final sum. Otherwise, further reduction, from s to 2, is needed.

Example 5.7

If the number of bits per column in a two-column counter (k, k, m) is increased beyond 5, then $m \geq 5$ and as a result, $s = \lceil \frac{m}{2} \rceil > 2$. For example, if $k = 7$, the inequality that must be satisfied is $2^m - 1 \geq 7 \cdot 3 = 21$, and therefore $m = 5$. A set of (7,7,5) counters will generate $s = 3$ operands, and consequently another set of (3,2) counters is needed in order to reduce the number of operands to 2. □

The hardware complexity of a carry-save adder for a large number of operands might be prohibitive, independent of the particular type of parallel counters employed. One way to reduce the hardware complexity is to design a smaller carry-save tree and use it iteratively. The n operands are divided into $\lceil n/j \rceil$ groups of j operands each, and a tree for $j + 2$ operands with two feedback paths and a CPA is designed, as shown in Figure 5.14. The two feedback paths make it necessary to complete the first pass through the CSA tree before the second set of j operands is applied. This slows down the execution of the multiple-operand addition, since pipelining is not possible. In the next section we discuss pipelining in general and describe ways to modify the tree structure in Figure 5.14 to support pipelining.

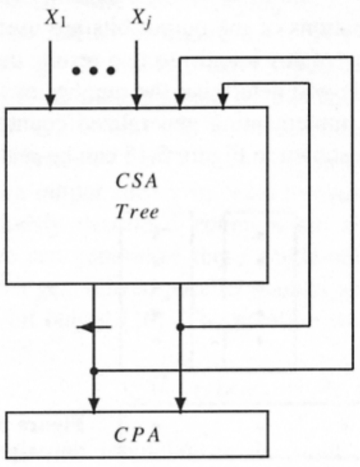

Figure 5.14 A CSA tree with two feedback paths and j new operands.