8 Multioperand Addition

"If A equals success, then the formula is A = X + Y + Z. X is work. Y is play. Z is keep your mouth shut."
ALBERT EINSTEIN

In Chapters 6 and 7, we covered several speedup methods for adding two operands. Our primary motivation in dealing with multioperand addition in this chapter is that both multiplication and inner-product computation reduce to adding a set of numbers, namely, the partial products or the component products. The main idea used is that of *deferred carry assimilation* made possible by redundant representation of the intermediate results.

8.1 USING TWO-OPERAND ADDERS

Multioperand addition is implicit in both multiplication and computation of vector inner products (Fig. 8.1). In multiplying a multiplicand a by a k-digit multiplier x, the k partial products $x_i a$ must be formed and then added. For inner-product computation, the component product terms $p^{(j)} = x^{(j)}y^{(j)}$ obtained by multiplying the corresponding elements of the two operand vectors x and y, need to be added. Computing averages (e.g., in the design of a mean filter) is another application that requires multioperand addition.

We will assume that the n operands are unsigned integers of the same width k and are aligned at the least-significant end, as in the right side of Fig. 8.1. Extension of the

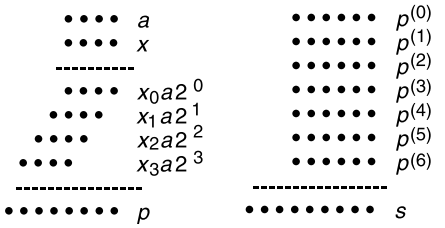

Figure 8.1 Multioperand addition problems for multiplication or inner-product computation shown in dot notation.

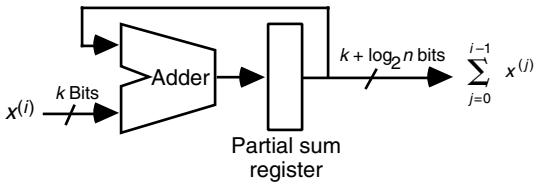

Figure 8.2 Serial implementation of multioperand addition with a single two-operand adder.

methods to signed operands are discussed in Section 8.5. Application to multiplication is the subject of Part III.

Figure 8.2 depicts a serial solution to the multioperand addition problem using a single two-operand adder. The binary operands $x^{(i)}, i = 0, 1, \cdots, n-1$, are applied, one per clock cycle, to one input of the adder, with the other input fed back from a partial sum register. Since the final sum can be as large as $n(2^k - 1)$, the partial sum register must be $\log_2(n2^k - n + 1) \approx k + \log_2 n$ bits wide.

Assuming the use of a logarithmic-time fast adder, the total latency of the scheme of Fig. 8.2 for adding n operands of width k is

$$T_{\text{serial-multi-add}} = \text{O}(n \log(k + \log n))$$

Since $k + \log n$ is no less than $\max(k, \log n)$ and no greater than $\max(2k, 2 \log n)$, we have $\log(k + \log n) = \text{O}(\log k + \log \log n)$ and

$$T_{\text{serial-multi-add}} = \text{O}(n \log k + n \log \log n)$$

Therefore, the addition time grows superlinearly with n when k is fixed and logarithmically with k for a given n.

One can pipeline this serial solution to get somewhat better performance. Figure 8.3 shows that if the adder is implemented as a four-stage pipeline, then three adders can be used to achieve the maximum possible throughput of one operand per clock cycle. Note that the presence of latches is assumed after each of the four adder stages and that a delay block simply represents a null operation followed by latches. The operation of the circuit in Fig. 8.3 is best understood if we trace the partially computed results from left to right. At the clock cycle when the ith input value $x^{(i)}$ arrives from the left and the sum of input values up to $x^{(i-12)}$ is output at the right, adder A is supplied with the two values $x^{(i)}$ and $x^{(i-1)}$. The partial results stored at the end of adder A's four stages correspond to the computations $x^{(i-1)} + x^{(i-2)}, x^{(i-2)} + x^{(i-3)}, x^{(i-3)} + x^{(i-4)}$, and $x^{(i-4)} + x^{(i-5)}$, with the latter final result used to label the output of adder A. Other labels attached to

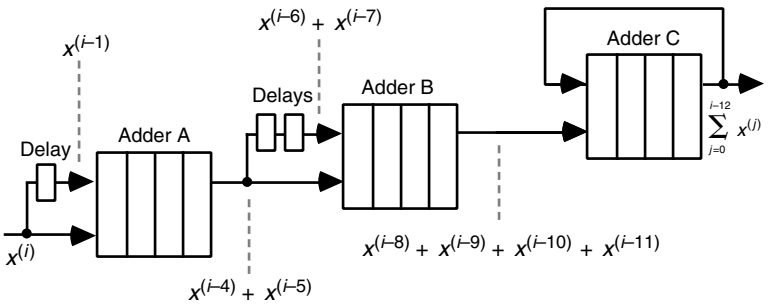

Figure 8.3 Serial multioperand addition when each adder is a four-stage pipeline.

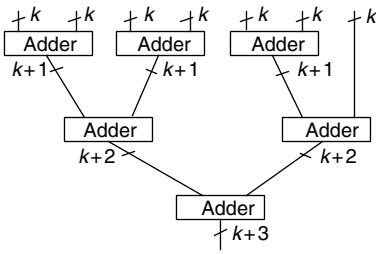

Figure 8.4 Adding seven numbers in a binary tree of adders.

the lines in Fig. 8.3 should allow the reader to continue this process, culminating in the determination of partial/final results of adder C. Even though the clock cycle is now shorter because of pipelining, the latency from the first input to the last output remains asymptotically the same with h-stage pipelining for any fixed h.

Note that the schemes shown in Figs. 8.2 and 8.3 work for any prefix computation involving a binary operator $\otimes$, provided the adder is replaced by a hardware unit corresponding to the binary operator $\otimes$. For example, similar designs can be used to find the product of n numbers or the largest value among them.

For higher speed, a tree of two-operand adders might be used, as in Fig. 8.4. Such a binary tree of two-operand adders needs $n - 1$ adders and is thus quite costly if built of fast adders. Strange as it may seem, the use of simple and slow ripple-carry (or even bit-serial) adders may be the best choice in this design. If we use fast logarithmic-time adders, the latency will be

$$\begin{aligned} T_{\text{tree-fast-multi-add}} &= \text{O}(\log\ k + \log(k+1) + \cdots + \log(k + \lceil \log_2 n \rceil - 1)) \\ &= \text{O}(\log\ n\ \log\ k + \log\ n \log\ \log\ n) \end{aligned}$$

The preceding equality can be proven by considering the two cases of $\log_2 n < k$ and $\log_2 n > k$ and bounding the right-hand side in each case. Supplying the needed details of the proof is left as an exercise. If we use ripple-carry adders in the tree of Fig. 8.4, the delay becomes

$$T_{\text{tree-ripple-multi-add}} = \text{O}(k + \log\ n)$$

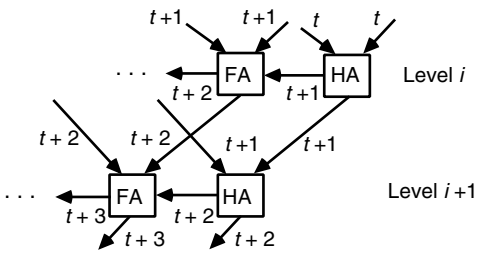

Figure 8.5 Ripple-carry adders at levels i and $i + 1$ in the tree of adders used for multioperand addition.

which can be less than the delay with fast adders for large n. Comparing the costs of this and the preceding schemes for different ranges of values for the parameters k and n is left as an exercise.

Figure 8.5 shows why the delay with ripple-carry adders is $O(k + \log n)$. There are $\lceil \log_2 n \rceil$ levels in the tree. An adder in the $(i + 1)$th level need not wait for full carry propagation in level i to occur, but rather can start its addition one full-adder (FA) delay after level i. In other words, carry propagation in each level lags 1 time unit behind the preceding level. Thus, we need to allow constant time for all but the last adder level, which needs $O(k + \log n)$ time.

Can we do better than the $O(k + \log n)$ delay offered by the tree of ripple-carry adders of Fig. 8.5? The absolute minimum time is $O(\log(kn)) = O(\log k + \log n)$, where kn is the total number of input bits to be processed by the multioperand adder, which is ultimately composed of constant–fan-in logic gates. This minimum is achievable with carry-save adders (CSAs).

8.2 CARRY-SAVE ADDERS

We can view a row of binary FAs as a mechanism to reduce three numbers to two numbers rather than as one to reduce two numbers to their sum. Figure 8.6 shows the relationship of a ripple-carry adder for the latter reduction and a CSA for the former (see also Fig. 3.5).

Figure 8.7 presents, in dot notation, the relationship shown in Fig. 8.6. To specify more precisely how the various dots are related or obtained, we agree to enclose any three dots that form the inputs to a FA in a dashed box and to connect the sum and carry outputs of an FA by a diagonal line (Fig. 8.8). Occasionally, only two dots are combined to form a sum bit and a carry bit. Then the two dots are enclosed in a dashed box and the use of a half-adder (HA) is signified by a cross line on the diagonal line connecting its outputs (Fig. 8.8).

Dot notation suggests another way to view the function of a CSA: as converter of a radix-2 number with the digit set [0, 3] (3 bits in one position) to one with the digit set [0, 2] (2 bits in one position).

A CSA tree (Fig. 8.9) can reduce n binary numbers to two numbers having the same sum in $O(\log n)$ levels. If a fast logarithmic-time carry-propagate adder (CPA) is then used to add the two resulting numbers, we have the following results for the cost and

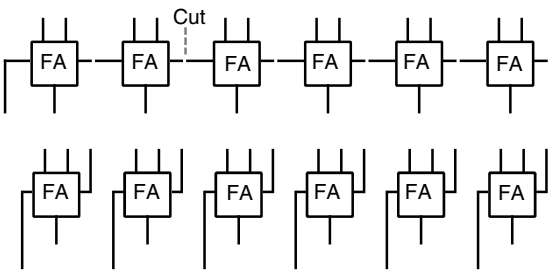

Figure 8.6 A ripple-carry adder turns into a carry-save adder if the carries are saved (stored) rather than propagated.

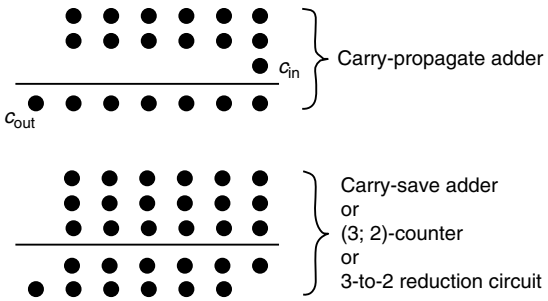

Figure 8.7 The CPA and CSA functions in dot notation.

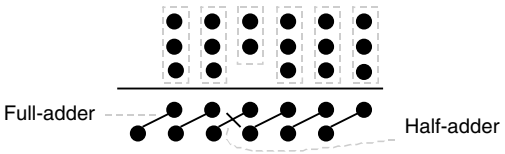

Figure 8.8 Specifying FA and HA blocks, with their inputs and outputs, in dot notation.

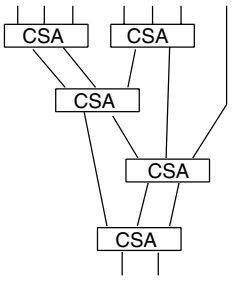

Figure 8.9 Tree of CSAs reducing seven numbers to two.

delay of n-operand addition:

$$C_{\text{carry-save-multi-add}} = (n-2)C_{\text{CSA}} + C_{\text{CPA}}$$
$$T_{\text{carry-save-multi-add}} = \text{O}(\text{tree height} + T_{\text{CPA}}) = \text{O}(\log\ n + \log\ k)$$

The needed CSAs are of various widths, but generally the widths are close to k bits; the CPA is of width at most $k + \log_2 n$.

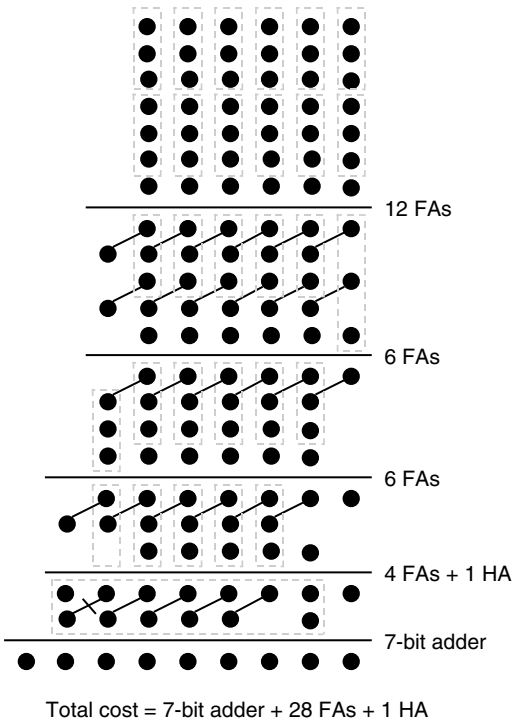

Figure 8.10 Addition of seven 6-bit numbers in dot notation.

8	7	6	5	4	3	2	1	0	Bit position
			7	7	7	7	7	7	$6 \times 2 = 12$ FAs
		2	5	5	5	5	5	3	6 FAs
		3	4	4	4	4	4	1	6 FAs
	1	2	3	3	3	3	2	1	4 FAs + 1 HA
	2	2	2	2	2	1	2	1	7-bit adder
			Carry-propagate adder						
1	1	1	1	1	1	1	1	1	

Figure 8.11 Representing a seven-operand addition in tabular form.

An example for adding seven 6-bit numbers is shown in Fig. 8.10. A more compact tabular representation of the same process is depicted in Fig. 8.11, where the entries represent the number of dots remaining in the respective columns or bit positions. We begin on the first row with seven dots in each of bit positions 0–5; these dots represent the seven 6-bit inputs. Two FAs are used in each 7-dot column, with each FA converting three dots in its column to one dot in that column and one dot in the next higher column. This leads to the distribution of dots shown on the second row of Fig. 8.11. Next, one FA is used in each of the bit positions 0–5 containing three dots or more, and so on, until no column contains more than two dots (see below for details). At this point, a CPA is

used to reduce the resulting two numbers to the final 9-bit sum represented by a single dot in each of the bit positions 0–8.

In deriving the entries of a row from those of the preceding one, we begin with column 0 and proceed to the leftmost column. In each column, we cast out multiples of 3 and for each group of three that we cast out, we include 1 bit in the same column and 1 bit in the next column to the left. Columns at the right that have already been reduced to 1 need no further reduction. The rightmost column with a 2 can be either reduced using an HA or left intact, postponing its reduction to the final CPA. The former strategy tends to make the width of the final CPA smaller, while the latter strategy minimizes the number of FAs and HAs at the expense of a wider CPA. In the example of Fig. 8.10, and its tabular form in Fig. 8.11, we could have reduced the width of the final CPA from 7 bits to 6 bits by applying an extra HA to the two dots remaining in bit position 1.

Figure 8.12 depicts a block diagram for the carry-save addition of seven k-bit numbers. By tagging each line in the diagram with the bit positions it carries, we see that even though the partial sums do grow in magnitude as more numbers are combined, the widths of the CSAs stay pretty much constant throughout the tree. Note that the lowermost CSA in Fig. 8.12 could have been made only $k - 1$ bits wide by letting the two lines in bit position 1 pass through. The CPA would then have become $k + 1$ bits wide.

Carry-save addition can be implemented serially using a single CSA, as depicted in Fig. 8.13. This is the preferred method when the operands arrive serially or must be read out from memory one by one. Note, however, that in this case both the CSA and final CPA will have to be wider.

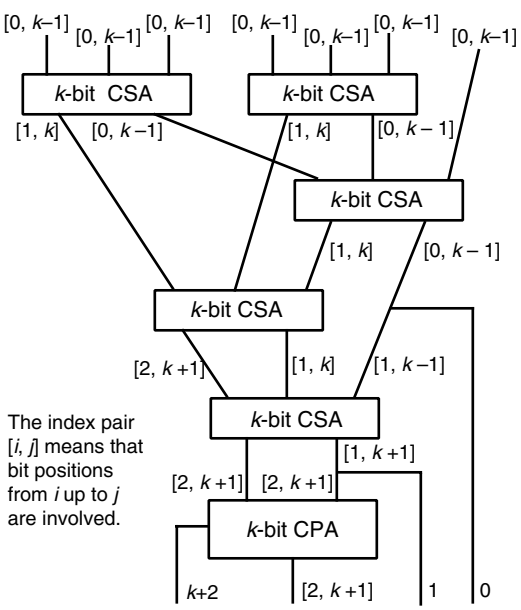

Figure 8.12 Adding seven k-bit numbers and the CSA/CPA widths required.

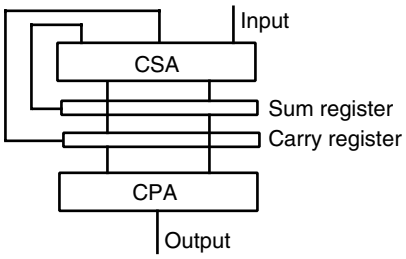

Figure 8.13 Serial carry-save addition by means of a single CSA.

8.3 WALLACE AND DADDA TREES

The CSA tree of Fig. 8.12, which reduces seven k-bit operands to two $(k+2)$-bit operands having the same sum, is known as a seven-input Wallace tree. More generally, an n-input Wallace tree reduces its k-bit inputs to two $(k + \log_2 n - 1)$-bit outputs. Since each CSA reduces the number of operands by a factor of 1.5, the smallest height $h(n)$ of an n-input Wallace tree satisfies the following recurrence:

$$h(n) = 1 + h(\lceil 2n/3 \rceil)$$

Applying this recurrence provides an exact value for the height of an n-input Wallace tree. If we ignore the ceiling operator in the preceding equation and write it as $h(n) = 1 + h(2n/3)$, we obtain a lower bound for the height, $h(n) \geq \log_{1.5}(n/2)$, where equality occurs only for $n = 2, 3$. Another way to look at the preceding relationship between the number of inputs and the tree height is to find the maximum number of inputs $n(h)$ that can be reduced to two outputs by an h-level tree. The recurrence for $n(h)$ is

$$n(h) = \lfloor 3n(h-1)/2 \rfloor$$

Again ignoring the floor operator, we obtain the upper bound $n(h) \leq 2(3/2)^h$. The lower bound $n(h) > 2(3/2)^{h-1}$ is also easily established. The exact value of $n(h)$ for $0 \leq h \leq 20$ is given in Table 8.1.

In Wallace trees, we reduce the number of operands at the earliest opportunity (see the example in Fig. 8.10). In other words, if there are m dots in a column, we immediately apply $\lfloor m/3 \rfloor$ FAs to that column. This tends to minimize the overall delay by making the final CPA as short as possible.

However, the delay of a fast adder is usually not a smoothly increasing function of the word width. A carry-lookahead adder, for example, may have essentially the same delay for word widths of 17–32 bits. In Dadda trees, we reduce the number of operands to the next lower value of $n(h)$ in Table 8.1 using the fewest FAs and HAs possible. The justification is that seven, eight, or nine operands, say, require four CSA levels; thus there is no point in reducing the number of operands below the next lower $n(h)$ value in the table, since this would not lead to a faster tree.

Let us redo the example of Fig. 8.10 by means of Dadda's strategy. Figure 8.14 shows the result. We start with seven rows of dots, so our first task is to reduce the number of

Table 8.1 The maximum number $n(h)$ of inputs for an h-level CSA tree

h	$n(h)$	h	$n(h)$	h	$n(h)$
0	2	7	28	14	474
1	3	8	42	15	711
2	4	9	63	16	1066
3	6	10	94	17	1599
4	9	11	141	18	2398
5	13	12	211	19	3597
6	19	13	316	20	5395

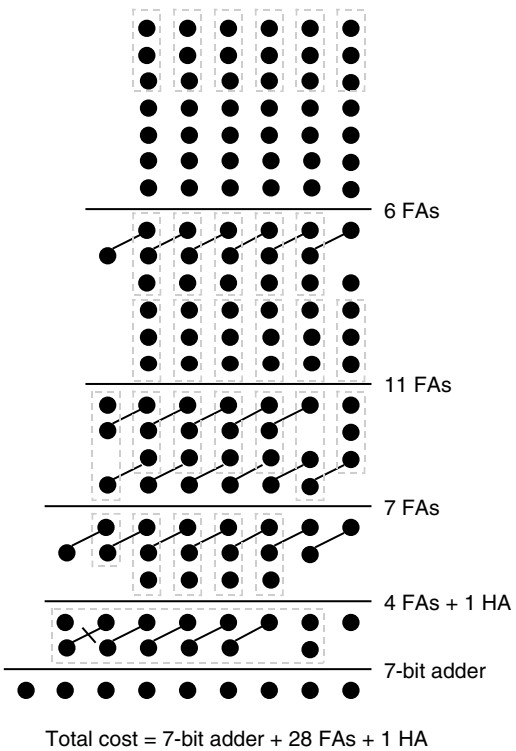

Figure 8.14 Using Dadda's strategy to add seven 6-bit numbers

rows to the next lower $n(h)$ value (i.e., 6). This can be done by using 6 FAs; next, we aim for four rows, leading to the use of 11 FAs, and so on. In this particular example, the Wallace and Dadda approaches result in the same number of FAs and HAs and the same width for the CPA. Again, the CPA width could have been reduced to 6 bits by using an extra HA in bit position 1.

Since a CPA has a carry-in signal that can be used to accommodate one of the dots, it is sometimes possible to reduce the complexity of the CSA tree by leaving three dots

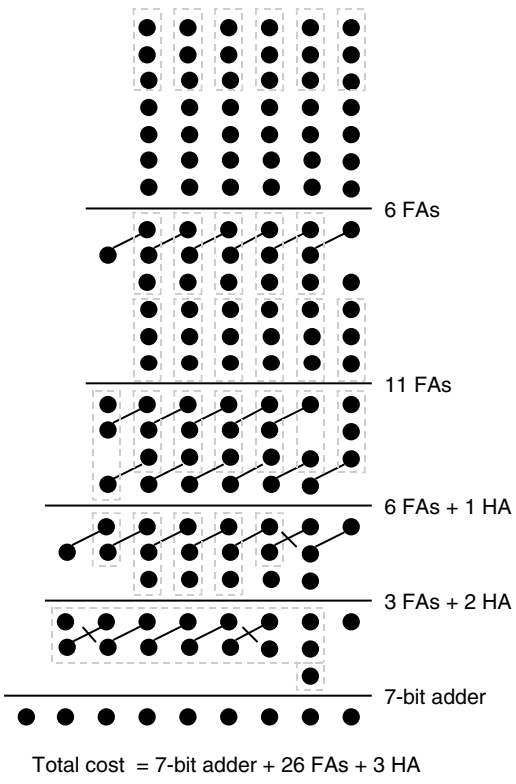

Figure 8.15 Adding seven 6-bit numbers by taking advantage of the final adder's carry-in.

in the least-significant position of the adder. Figure 8.15 shows the same example as in Figs. 8.10 and 8.14, but with two FAs replaced with HAs, leaving an extra dot in each of the bit positions 1 and 2.

8.4 PARALLEL COUNTERS AND COMPRESSORS

A 1-bit FA is sometimes referred to as a (3; 2)-counter, meaning that it counts the number of 1s among its 3 input bits and represents the result as a 2-bit number. This can be easily generalized: an $(n; \lceil \log_2(n+1) \rceil)$-counter has n inputs and produces a $\lceil \log_2(n+1) \rceil$-bit binary output representing the number of 1s among its n inputs. Such a circuit is also known as an n-input parallel counter.

A 10-input parallel counter, or a (10; 4)-counter, is depicted in Fig. 8.16 in terms of both dot notation and circuit diagram with FAs and HAs. A row of such (10; 4)-counters, one per bit position, can reduce a set of 10 binary numbers to 4 binary numbers. The dot notation representation of this reduction is similar to that of (3; 2)-counters, except that each diagonal line connecting the outputs of a (10; 4)-counter will go through four dots. A (7; 3)-counter can be similarly designed.

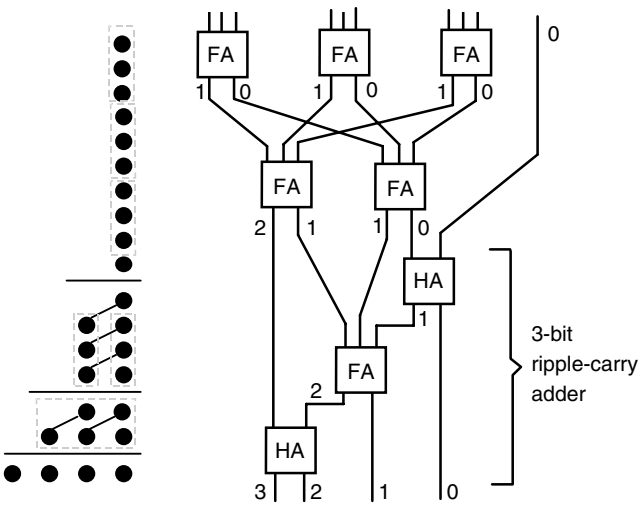

Figure 8.16 A 10-input parallel counter also known as a (10; 4)-counter.

Even though a circuit that counts the number of 1s among n inputs is known as a parallel counter, we note that this does not constitute a true generalization of the notion of a sequential counter. A sequential counter receives 1 bit (the count signal) and adds it to a stored count. A parallel counter, then, could have been defined as a circuit that receives n count signals and adds them to a stored count, thus in effect incrementing the count by the sum of the input count signals. Such a circuit has been called an "accumulative parallel counter" [Parh95]. An accumulative parallel counter can be built from a parallel incrementer (a combinational circuit receiving a number and producing the sum of the input number and n count signals at the output) along with a storage register.

Both parallel and accumulative parallel counters can be extended by considering signed count signals. These would constitute generalizations of sequential up/down counters [Parh89]. Accumulative and up/down parallel counters have been applied to the design of efficient Hamming weight comparators, circuits that are used to decide whether the number of 1s in a given bit-vector is greater than or equal to a threshold, or to determine which of two bit-vectors contains more 1s [Parh09].

A parallel counter reduces a number of dots in the same bit position into dots in different positions (one in each). This idea can be easily generalized to circuits that receive "dot patterns" (not necessarily in a single column) and convert them to other dot patterns (not necessarily one in each column). If the output dot pattern has fewer dots than the input dot pattern, compression takes place; repeated use of such circuits can eventually lead to the reduction of n numbers to a small set of numbers (ideally two).

A generalized parallel counter (parallel compressor) is characterized by the number of dots in each input column and in each output column. We do not consider such circuits in their full generality but limit ourselves to those that output a single dot in each column. Thus, the output side of such parallel compressors is again characterized by a single integer representing the number of columns spanned by the output. The input side is characterized by a sequence of integers corresponding to the number of inputs in various columns.

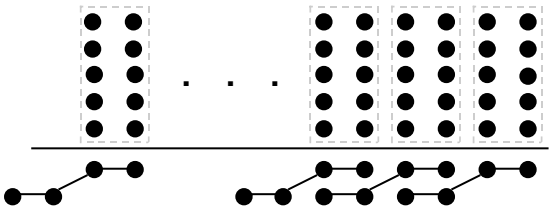

Figure 8.17 Dot notation for a (5, 5; 4)-counter and the use of such counters for reducing five numbers to two numbers.

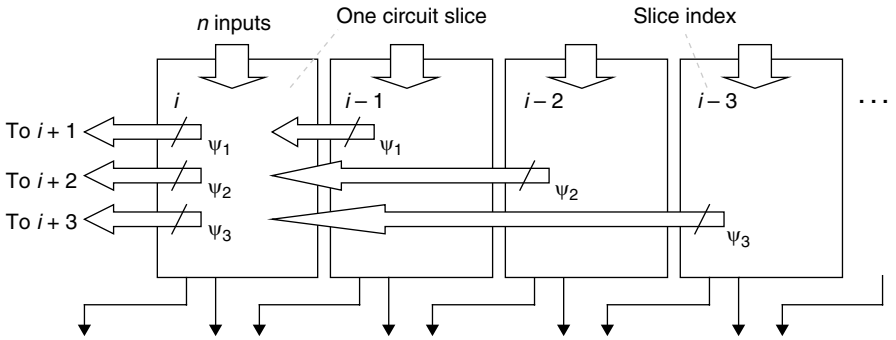

Figure 8.18 Schematic diagram of an $(n; 2)$-counter built of identical circuit slices.

For example, a (4, 4; 4)-counter receives 4 bits in each of two adjacent columns and produces a 4-bit number representing the sum of the four 2-bit numbers received. Similarly, a (5, 5; 4)-counter, depicted in Fig. 8.17, reduces five 2-bit numbers to a 4-bit number. The numbers of input dots in various columns do not have to be the same. For example, a (4, 6; 4)-counter receives 6 bits of weight 1 and 4 bits of weight 2 and delivers their weighted sum in the form of a 4-bit binary number. For a counter of this type to be feasible, the sum of the output weights must equal or exceed the sum of its input weights. In other words, if there are n_j dots in each of h input columns, $0 \leq j \leq h-1$, the associated generalized parallel counter, denoted as $(n_{h-1}, \ldots, n_1, n_0; k)$-counter, is feasible only if $\Sigma(n_j 2^j) \leq 2^k - 1$.

Generalized parallel counters are quite powerful. For example, a 4-bit binary FA is really a (2, 2, 2, 3; 5)-counter.

Since our goal in multioperand carry-save addition is to reduce n numbers to two numbers, we sometimes talk of $(n; 2)$-counters, even though, with our preceding definition, this does not make sense for $n > 3$. By an $(n; 2)$-counter, $n > 3$, we usually mean a slice of a circuit that helps us reduce n numbers to two numbers when suitably replicated. Slice i of the circuit receives n input bits in position i, plus transfer or "carry" bits from one or more positions to the right ($i-1, i-2$, etc.), and produces output bits in the two positions i and $i+1$ plus transfer bits into one or more higher positions ($i+1, i+2$, etc.).

Figure 8.18 shows the block diagram of an $(n; 2)$-counter, composed of k identical circuit slices with horizontal interconnections among them. Each slice combines n input bits with a number of carries coming from slices to its right, producing 2 output bits along with carries that are sent to its left. If ψ_j denotes the number of transfer bits from

slice i to slice $i + j$, the fundamental inequality to be satisfied for this scheme to work is

$$n + \psi_1 + \psi_2 + \psi_3 + \cdots \leq 3 + 2\psi_1 + 4\psi_2 + 8\psi_3 + \cdots$$

where 3 represents the maximum value of the 2 output bits. For example, a (7; 2)-counter can be built by allowing $\psi_1 = 1$ transfer bit from position i to position $i + 1$ and $\psi_2 = 1$ transfer bit into position $i + 2$. For maximum speed, the circuit slice must be designed in such a way that transfer signals are introduced as close to the circuit's outputs as possible, to prevent the transfers from rippling through many stages. Design of a (7; 2)-counter using these principles is left as an exercise.

For $n = 4$, a (4; 2)-counter can be synthesized with $\psi_1 = 1$, that is, with 1 carry bit between adjacent slices. An efficient circuit realization for such a counter will be presented in Section 11.2, in connection with reduction circuits for parallel multipliers organized as binary trees (see Fig. 11.5).

8.5 ADDING MULTIPLE SIGNED NUMBERS

When the operands to be added are 2's-complement numbers, they must be sign-extended to the width of the final result if multiple-operand addition is to yield their correct sum. The example in Fig. 8.19 shows extension of the sign bits x_{k-1}, y_{k-1}, and z_{k-1} across five extra positions.

It appears, therefore, that sign extension may dramatically increase the complexity of the CSA tree used for n-operand addition when n is large. However, since the sign extension bits are identical, a single FA can do the job of several FAs that would be receiving identical inputs if used. With this hardware-sharing scheme, the CSA widths are only marginally increased. For the three operands in Fig. 8.19a, a single (3; 2)-counter can be used in lieu of six that would be receiving the same input bits x_{k-1}, y_{k-1}, and z_{k-1}.

It is possible to avoid sign extension by taking advantage of the negative-weight interpretation of the sign bit in 2's-complement representation. A negative sign bit $-x_{k-1}$

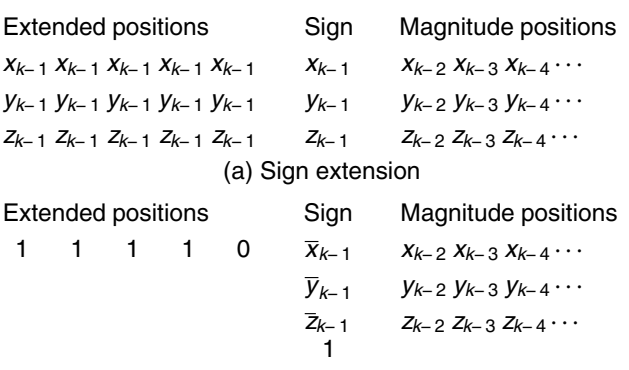

Figure 8.19 Adding three 2's-complement numbers via two different methods.

can be replaced by $1 - x_{k-1} = \bar{x}_{k-1}$ (the complement of x_{k-1}), with the extra 1 canceled by inserting a -1 in that same column. Multiple -1s in a given column can be paired, with each pair replaced by a -1 in the next higher column. Finally, a solitary -1 in a given column is replaced by 1 in that column and -1 in the next higher column. Eventually, all the -1s disappear off the left end and at most a single extra 1 is left in some of the columns.

Figure 8.19b shows how this method is applied when adding three 2's-complement numbers. The three sign bits are complemented and three -1s are inserted in the sign position. These three -1s are then replaced by a 1 in the sign position and two -1s in the next higher position (k). These two -1s are then removed and, instead, a single -1 is inserted in position $k + 1$. The latter -1 is in turn replaced by a 1 in position $k + 1$ and a -1 in position $k + 2$, and so on. The -1 that moves out from the leftmost position is immaterial in view of $(k + 5)$-bit 2's-complement arithmetic being performed modulo 2^{k+5}.

8.6 MODULAR MULTIOPERAND ADDERS

For the same reasons offered for modular two-operand addition in Section 7.6, on occasion we need to add n numbers modulo a given constant m. An obvious approach would be to perform the required computation in two stages: (1) Forming the proper sum of the input operands, using any of the multioperand adder designs described thus far, and (2) reducing the sum modulo m. In many cases, however, we can obtain more efficient designs by merging (interlacing) the addition and modular reduction operations.

As in the case of two-operand addition, the three special moduli $2^k, 2^k - 1$, and $2^k + 1$ are easier to deal with. For $m = 2^k$, we simply drop any bit that is produced in column k. This simplification is depicted in Fig. 8.20a. Thus, for example, no CSA in Fig. 8.12 needs to extend past position $k - 1$ in this case. For $m = 2^k - 1$, a bit generated in position k is reinserted into position 0, as shown in Fig. 8.20b. Given the empty slot available in position 0, this "end-around carry" does not lead to an increase in latency. In the case of $m = 2^k + 1$, assuming nonzero operands with diminished-1 encoding, the arguments presented in Example 7.3 suggest that an inverted end-around carry (Fig. 8.20c) allows the conversion of three diminished-1 inputs to two diminished-1 outputs.

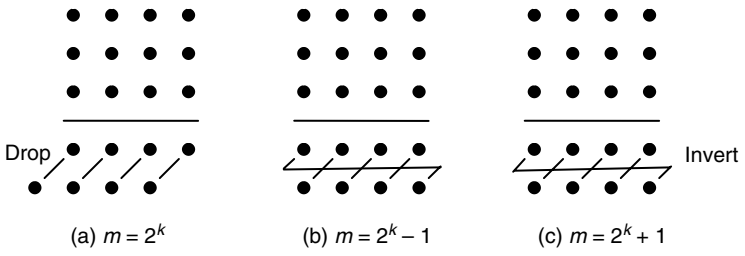

Figure 8.20 Modular carry-save addition with special moduli.

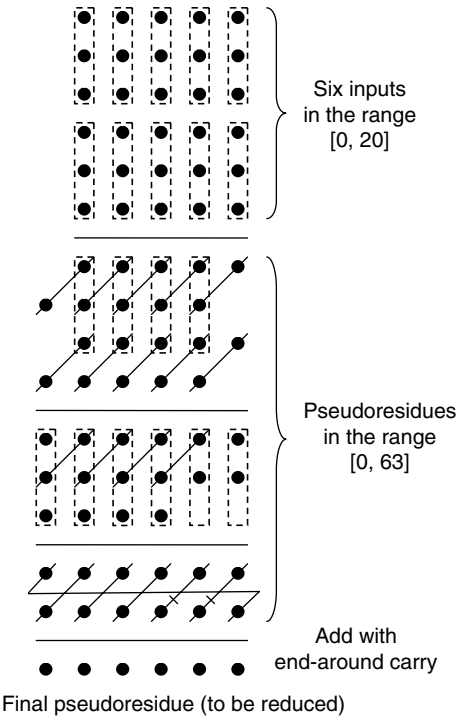

Figure 8.21 Modulo-21 reduction of 6 numbers taking advantage of the fact that $64 = 1 \bmod 21$ and using 6-bit pseudoresidues.

For a general modulus m, we need multioperand addition schemes that are more elaborate than (inverted) end-around carry. Many techniques have been developed for specific values of m. For example, if m is such that $2^h = 1 \bmod m$ for a fairly small value of h, one can perform tree reduction with h-bit pseudoresidues (see Section 4.5) and end-around carry [Pies94]. To apply this method to mod-21 addition of a set of n input integers in the range [0, 20], we can use any tree reduction scheme, while keeping all intermediate values in the range [0, 63]. Bits generated in column 6 are then fed back to column 0 in the same manner as the end-around carry used for modulo-63 reduction, given that $64 = 1 \bmod 21$. Once all operands have been combined into two 6-bit values, the latter are added with end-around carry and the final 6-bit sum is reduced modulo 21. Figure 8.21 depicts an example with $n = 6$.

PROBLEMS

8.1 Pipelined multioperand addition

a. Present a design similar to Fig. 8.3 for adding a set of n input numbers, with a throughput of one input per clock cycle, if each adder block is a two-stage pipeline.

b. Repeat part a for a pipelined adder with eight stages.

c. Discuss methods for using the pipelined multioperand addition scheme of Fig. 8.3 when the number of pipeline stages in an adder block is not a power of 2. Apply your method to the case of an adder with five pipeline stages.

8.2 Multioperand addition with two-operand adders

Consider all the methods discussed in Section 8.1 for adding n unsigned integers of width k using two-operand adders.

a. Using reasonable assumptions, derive exact, as opposed to asymptotic, expressions for the delay and cost of each method.
b. On a two-dimensional coordinate system, with the axes representing n and k, identify the regions where each method is best in terms of speed.
c. Repeat part b, this time using delay $\times$ cost as the figure of merit for comparison.

8.3 Comparing multioperand addition schemes

Consider the problem of adding n unsigned integers of width k.

a. Identify two methods whose delays are $O(\log\ k + n)$ and $O(k + \log\ n)$.
b. On a two-dimensional coordinate system, with logarithmic scales for both n and k, identify the regions in which one design is faster than the other. Describe your assumptions about implementations.
c. Repeat part b, this time comparing cost-effectiveness rather than just speed.

8.4 Building blocks for multioperand addition

A CSA reduces three binary numbers to two binary numbers. It costs c units and performs its function with a time delay d. An "alternative reduction adder" (ARA) reduces five binary numbers to two binary numbers. It costs $3c$ units and has a delay of $2d$.

a. Which of the two elements, CSA or ARA, is more cost-effective for designing a tree that reduces 32 operands to 2 operands if used as the only building block? Ignore the widths of the CSA and ARA blocks and focus only on their numbers.
b. Propose an efficient design for 32-to-2 reduction if both CSA and ARA building blocks are allowed.

8.5 CSA trees

Consider the problem of adding eight 8-bit unsigned binary numbers.

a. Using tabular representation, show the design of a Wallace tree for reducing the eight operands to two operands.
b. Repeat part a for a Dadda tree.
c. Compare the two designs with respect to speed and cost.

8.6 CSA trees

We have seen that the maximum number of operands that can be combined using an h-level tree of CSAs is $n(h) = \lfloor 3n(h-1)/2 \rfloor$.

a. Prove the inequality $n(h) \geq 2n(h-2)$.
b. Prove the inequality $n(h) \geq 3n(h-3)$.
c. Show that both bounds of parts a and b are tight by providing one example in which equality holds.
d. Prove the inequality $n(h) \geq n(h-a)\lfloor n(a)/2 \rfloor$ for $a \geq 0$. *Hint:* Think of the h-level tree as the top $h-a$ levels followed by an a-level tree and consider the lines connecting the two parts.

8.7 A three-operand addition problem

Effective 24-bit addresses in the IBM System 370 family of computers were computed by adding three unsigned values: two 24-bit numbers and a 12-bit number. Since address computation was needed for each instruction, speed was critical and using two addition steps wouldn't do, particularly for the faster computers in the family.

a. Suggest a fast addition scheme for this address computation. Your design should produce an "address invalid" signal when there is an overflow.
b. Extend your design so that it also indicates if the computed address is in the range $[0, u]$, where u is a given upper bound (an input to the circuit).

8.8 Parallel counters

Design a 255-input parallel counter using (7; 3)-counters and 4-bit binary adders as the only building blocks.

8.9 Parallel counters

Consider the synthesis of an n-input parallel counter.

a. Prove that $n - \log_2 n$ is a lower bound on the number of FAs needed.
b. Show that n FAs suffice for this task. *Hint:* Think in terms of how many FAs might be used as HAs in the worst case.
c. Prove that $\log_2 n + \log_3 n - 1$ is a lower bound on the number of FA levels required. *Hint:* First consider the problem of determining the least-significant output bit, or actually, that of reducing the weight-2^0 column to 3 bits.

8.10 Generalized parallel counters

Consider a (1, 4, 3; 4) generalized parallel counter.

a. Design the generalized parallel counter using only FA blocks.
b. Show how this generalized parallel counter can be used as a 3-bit binary adder.
c. Use three such parallel counters to reduce five 5-bit unsigned binary numbers into three 6-bit numbers.
d. Show how such counters can be used for 4-to-2 reduction.

8.11 Generalized parallel counters

a. Is a (3, 1; 3)-counter useful? Why (not)?
b. Design a (3, 3; 4)-counter using (3; 2)-counters as the only building blocks.
c. Use the counters of part b, and a 12-bit adder, to build a 6×6 unsigned multiplier.
d. Viewing a 4-bit binary adder as a (2, 2, 2, 3; 5)-counter and using dot notation, design a circuit to add five 6-bit binary numbers using only 4-bit adders as your building blocks.

8.12 Generalized parallel counters

We want to design a slice of a (7; 2)-counter as discussed in Section 8.4.

a. Present a design for slice i based on $\psi_1 = 1$ transfer bit from position $i - 1$ along with $\psi_2 = 1$ transfer bit from position $i - 2$.
b. Repeat part a with $\psi_1 = 4$ transfer bits from position $i - 1$ and $\psi_2 = 0$.
c. Compare the designs of parts a and b with respect to speed and cost.

8.13 Generalized parallel counters

We have seen that a set of $k/2$ (5, 5; 4)-counters can be used to reduce five k-bit operands to two operands. *Hint:* This is possible because the 4-bit outputs of adjacent counters overlap in 2 bits, making the height of the output dot matrix equal to 2.

a. What kind of generalized parallel counter is needed to reduce seven operands to two operands?
b. Repeat part a for reducing nine operands.
c. Repeat part a for the general case of n operands, obtaining the relevant counter parameters as functions of n.

8.14 Accumulative parallel counters

Design a 12-bit, 50-input accumulative parallel counter. The counter has a 12-bit register in which the accumulated count is kept. When the "count" signal goes high, the input count (a number between 0 and 50) is added to the stored count. Try to make your design as fast as possible. Ignore overflow (i.e., assume modulo-2^{12} operation). *Hint:* A 50-input parallel counter followed by a 12-bit adder isn't the best design.

8.15 Unsigned versus signed multioperand addition

We want to add four 4-bit binary numbers.

a. Construct the needed circuit, assuming unsigned operands.
b. Repeat part a, assuming sign-extended 2's-complement operands.
c. Repeat part a, using the negative-weight interpretation of the sign bits.
d. Compare the three designs with respect to speed and cost.

8.16 Adding multiple signed numbers

a. Present the design of a multioperand adder for computing the 9-bit sum of sixteen 6-bit, 2's-complement numbers based on the use of negatively weighted sign bits, as described at the end of Section 8.5.
b. Redo the design using straightforward sign extension.
c. Compare the designs of parts a and b with respect to speed and cost and discuss.

8.17 Ternary parallel counters

In balanced ternary representation (viz., $r = 3$ and digit set $[-1, 1]$), (4; 2)-counters can be designed [De94]. Choose a suitable encoding for the three digit values and present the complete logic design of such a (4; 2)-counter.

8.18 Generalized parallel counters

a. Show an implementation of a (5, 5; 4)-counter using (3; 2)-counters.
b. One level of (5, 5; 4) counters can be used to reduce five operands to two. What is the maximum number of operands that can be reduced to two when two levels of (5, 5; 4) counters are used?
c. Generalize the result of part b to a counter that reduces x columns of n dots to a $2x$-bit result.

8.19 CSA trees

a. Show the widths of the four CSAs required to reduce six 32-bit unsigned binary numbers to two.
b. Repeat part a, but assume that the six 32-bit unsigned numbers are the partial products of a 32×6 multiplication (i.e., they are not aligned at least-significant bits but shifted to the left by 0, 1, 2, 3, 4, and 5 bits).

8.20 Using (7; 3)- and (7; 2)-counters

a. Given a circuit corresponding to a slice of a (7; 2)-counter, with slice i receiving carries from positions $i - 2$ and $i - 1$ and producing carries for positions $i + 1$ and $i + 2$, show that it can be used as a (7; 3)-counter if desired.
b. Show how to use two copies of a (7; 2)-counter slice to synthesize a slice of an (11; 2)-counter.
c. How can one synthesize a slice of a (15; 2)-counter using (7; 2)-counter slices?

8.21 Parallel counters using sorting networks

An n-input parallel counter can be synthesized by first using an n-input bit-sorting network that arranges the n bits in ascending order and then detecting the position of the single transition from 0 to 1 in the output sequence. Study the suitability of this method for synthesizing parallel counters. For a discussion of sorting networks see [Parh99], Chapter 7. Note that for bit-sorting, the 2-sorter components needed

contain nothing but a 2-input AND gate and a 2-input OR gate. For other design ideas see [Fior99].

8.22 Design of (4; 2)-counters

A (4; 2)-counter is in essence a (5; 2, 1)-compressor: it receives 4 bits plus a carry-in and produces 2 bits of weight 2 (one of them is carry-out) and 1 bit of weight 1.

a. Express a (7; 2)-counter in this form, using two columns of dots at the output.
b. Repeat part a with three columns of dots at the output.
c. Show that the following logic equations [Shim97] implement a (4; 2)-counter with inputs $c_{in}, x_0, x_1, x_2, x_3$, and outputs c_{out}, y_1, y_0.

$$c_{out} = (x_0 \vee x_1)(x_2 \vee x_3), \quad s = x_0 \oplus x_1 \oplus x_2 \oplus x_3,$$
$$y_0 = c_{in} \oplus s, \quad y_1 = s\, c_{in} \vee \bar{s}(x_0 x_1 \vee x_2 x_3)$$

8.23 Saturating multioperand adder

In certain applications, when the result of an arithmetic operation exceeds the maximum allowed value, it would be inappropriate to generate the result modulo a power of 2. For example, in media processing, we do not want addition of 1 to a black pixel coded as FF in hexadecimal to turn it into a white pixel 00. Discuss how multioperand addition can be performed with saturation so that whenever the final sum exceeds $2^k - 1$, the maximum value $2^k - 1$ is produced at output.

8.24 Height of *n*-input Wallace tree

a. Prove that the minimum height $h(n)$ of an n-input Wallace tree, $n \geq 3$, does not in general satisfy $h(n) = \lceil \log_{1.5}(n/2) \rceil$.
b. Does the relationship $h(n) = \lceil \log_{1.5}[n/2 + (n \bmod 3)/4] \rceil$ hold? Prove or disprove.

8.25 Tabular representation of multioperand addition

The following describes a multioperand addition process in tabular form.

1	2	3	4	5	6	7	8	7	6	5	4	3	2	1
1	2	3	4	6	6	6	6	6	6	5	4	3	2	1
1	2	4	4	4	4	4	4	4	4	4	4	3	2	1
1	3	3	3	3	3	3	3	3	3	3	3	3	2	1
2	2	2	2	2	2	2	2	2	2	2	2	2	2	1

a. Explain the process described by this table.
b. In the hardware implementation implied by the table, what types of components are used and how many of each? Be as precise as possible in specifying the components used.

8.26 Tabular representation of multioperand addition

The following describes a multioperand addition process in tabular form.

	8	8	8	8	8	8	8	8
2	6	6	6	6	6	6	6	4
4	4	4	4	4	4	4	3	2
1 3	3	3	3	3	3	3	2	1
2 2	2	2	2	2	2	2	1	1

a. Explain the process described by this table.
b. In the hardware implementation implied by the table, what types of components are used and how many of each? Be as precise as possible in specifying the components used.

8.27 Fast modular addition

Using multioperand addition methods, redesign the mod-13 adder of Fig. 4.3 to make it as fast as possible and estimate the latency of your design in gate levels. *Hint*: Precompute $x + y + 3$.

8.28 Saturating parallel counters

Study the design of parallel counters that provide an exact count when the number of 1 inputs is below a given threshold τ and saturate to τ otherwise.

8.29 Latency of a parallel counter

We can build a $(2^h - 1)$-input parallel counter recursively from two $(2^{h-1} - 1; h - 1)$-counters that feed an $(h - 1)$-bit ripple-carry adder. The smaller counters accommodate $2^h - 2$ of the inputs, with the last input inserted as carry-in of the final adder. The resulting structure will resemble Fig. 8.5, except that the HAs are replaced with FAs to allow the use of a carry-in signal into each ripple-carry adder.

a. Design a (31; 5)-counter based on this strategy.
b. Derive the latency of your design, in terms of FA levels, and compare the result with the number of levels suggested by Table 8.1.
c. What do you think is the source of the discrepancy in part b?

8.30 Modular multioperand addition

For each of the following moduli, devise an efficient method for multioperand addition using the pseudoresidue method discussed at the end of Section 8.6 [Pies94].

a. 11
b. 23
c. 35

REFERENCES AND FURTHER READINGS

[Dadd65] Dadda, L., "Some Schemes for Parallel Multipliers," *Alta Frequenza*, Vol. 34, pp. 349–356, 1965.

[Dadd76] Dadda, L., "On Parallel Digital Multipliers," *Alta Frequenza*, Vol. 45, pp. 574–580, 1976.

[De94] De, M., and B. P. Sinha, "Fast Parallel Algorithm for Ternary Multiplication Using Multivalued I^2L Technology," *IEEE Trans. Computers*, Vol. 43, No. 5, pp. 603–607, 1994.

[Didi04] Didier, L. S., and P.-Y. H. Rivaille, "A Comparative Study of Modular Adders," *Advanced Signal Processing Algorithms, Architectures, and Implementations XIV* (Proc. SPIE Conf.), 2004, pp. 13–20.

[Fior99] Fiore, P. D., "Parallel Multiplication Using Fast Sorting Networks," *IEEE Trans. Computers*, Vol. 48, No. 6, pp. 640–645, 1999.

[Fost71] Foster, C. C., and F. D. Stockton, "Counting Responders in an Associative Memory," *IEEE Trans. Computers*, Vol. 20, pp. 1580–1583, 1971.

[Kore03] Koren, I., Y. Koren, and B. G. Oomman, "Saturating Counters: Application and Design Alternatives," *Proc. 16th IEEE Symp. Computer Arithmetic*, pp. 228–235, 2003.

[Parh89] Parhami, B., "Parallel Counters for Signed Binary Signals," *Proc. 23rd Asilomar Conf. Signals, Systems, and Computers*, pp. 513–516, 1989.

[Parh95] Parhami, B., and C.-H. Yeh, "Accumulative Parallel Counters," *Proc. 29th Asilomar Conf. Signals, Systems, and Computers*, pp. 966–970, 1995.

[Parh99] Parhami, B., *Introduction to Parallel Processing: Algorithms and Architectures*, Plenum, 1999.

[Parh09] Parhami, B., "Efficient Hamming Weight Comparators for Binary Vectors Based on Accumulative and Up/Down Parallel Counters," *IEEE Trans. Circuits and Systems II*, Vol. 56, No. 2, pp. 167–171, 2009.

[Pies94] Piestrak, S. J., "Design of Residue Generators and Multioperand Modular Adders Using Carry-Save Adders," *IEEE Trans. Computers*, Vol. 43, No. 1, pp. 68–77, 1994.

[Shim97] Shim, D., and W. Kim, "The Design of 16 × 16 Wave Pipelined Multiplier Using Fan-In Equalization Technique," *Proc. Midwest Symp. Circuits & Systems*, Vol. 1, pp. 336–339, 1997.

[Swar73] Swartzlander, E. E., "Parallel Counters," *IEEE Trans. Computers*, Vol. 22, No. 11, pp. 1021–1024, 1973.

[Wall64] Wallace, C. S., "A Suggestion for a Fast Multiplier," *IEEE Trans. Electronic Computers*, Vol. 13, pp. 14–17, 1964.

[Wang96] Wang, Z., G. A. Jullien, and W. C. Carter, "An Efficient Tree Architecture for Modulo $2^n + 1$ Multiplication," *J. VLSI Signal Processing*, Vol. 14, No. 3, pp. 241–248, 1996.