11 Tree and Array Multipliers

"All my discoveries were simply improvements in notation"
GOTTFRIED WILHELM VON LEIBNIZ

Tree, or fully parallel, multipliers constitute limiting cases of high-radix multipliers (radix-2^k). With a high-performance carry-save adder (CSA) tree followed by a fast adder, logarithmic time multiplication becomes possible. The resulting multipliers are expensive but justifiable for applications in which multiplication speed is critical. One-sided CSA trees lead to much slower, but highly regular, structures known as array multipliers that offer higher pipelined throughput than tree multipliers and significantly lower chip area at the same time. Chapter topics include:

11.1 FULL-TREE MULTIPLIERS

In their simplest forms, parallel or full-tree multipliers can be viewed as extreme cases of the design in Fig. 10.12, where all the k multiples of the multiplicand are produced at once and a k-input carry-save adder (CSA) tree is used to reduce them to two operands for the final addition. Because all the multiples are combined in one pass, the tree does not require feedback links, making pipelining quite feasible.

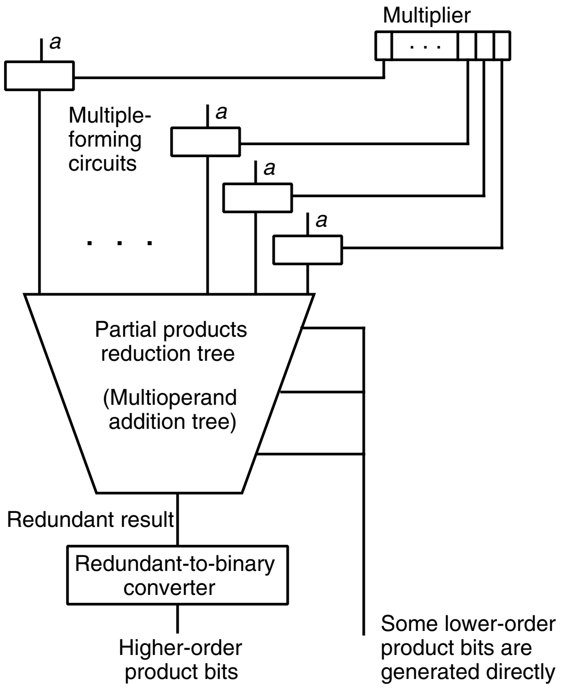

Multiplier
. . .
a
Multiple-forming circuits
a
a
a
. . .
Partial products reduction tree
(Multioperand addition tree)
Redundant result
Redundant-to-binary converter
Higher-order product bits
Some lower-order product bits are generated directly

Figure 11.1 General structure of a full-tree multiplier.

Figure 11.1 shows the general structure of a full-tree multiplier. Various multiples of the multiplicand a, corresponding to binary or high-radix digits of the multiplier x or its recoded version, are formed at the top. The multiple-forming circuits may be a collection of AND gates (binary multiplier), radix-4 Booth's multiple generators (recoded multiplier), and so on. These multiples are added in a combinational partial products reduction tree, which produces their sum in redundant form. Finally, the redundant result is converted to standard binary output at the bottom.

Many types of tree multipliers have been built or proposed. These are distinguished by the designs of the following three elements in Fig. 11.1:

Multiple-forming circuits

Partial-products reduction tree

Redundant-to-binary converter

In the remainder of this section, we focus on tree multiplier variations involving unsigned binary multiples and CSA reduction trees. With the redundant result in carry-save form, the final converter is simply a fast adder. Deviations from the foregoing multiple generation and reduction schemes are discussed in Section 11.2. Signed tree multipliers are covered in Section 11.3.

From our discussion of sequential multiplication in Chapters 9 and 10, we know how the partial-products can be formed and how, through the use of high-radix methods, the number of partial products can be reduced. The trade-offs mentioned for high-radix

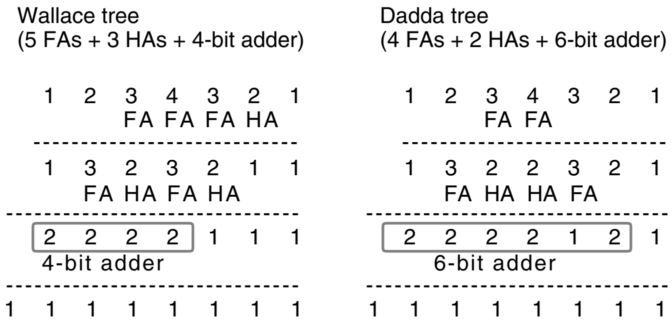


Figure 11.2 Two different binary 4 $\times$ 4 tree multipliers.

multipliers exist here as well: more complex multiple-forming circuits can lead to simplification in the reduction tree. Again, we cannot say in general which combination will lead to greater cost-effectiveness because the exact nature of the trade-off is design- and technology-dependent.

Recall Wallace's and Dadda's strategies for constructing CSA trees discussed in Section 8.3. These give rise to Wallace and Dadda tree multipliers, respectively. Essentially, Wallace's strategy for building CSA trees is to combine the partial-product bits at the earliest opportunity. With Dadda's method, combining takes place as late as possible, while keeping the critical path length of the CSA tree at a minimum. Wallace's method leads to the fastest possible design, and Dadda's strategy usually leads to a simpler CSA tree and a wider carry-propagate adder (CPA).

As a simple example, we derive Wallace and Dadda tree multipliers for 4 $\times$ 4 multiplication. Figure 11.2 shows the design process and results in tabular form, where the integers indicate the number of dots remaining in the various columns. Each design begins with 16 AND gates forming the $x_i a_j$ terms or dots, $0 \le i, j \le 3$. The resulting 16 dots are spread across seven columns in the pattern 1, 2, 3, 4, 3, 2, 1. The Wallace tree design requires 3 full adders (FAs) and 1 half-adder (HA) in the first level, then 2 FAs and 2 HAs in the second level, and a 4-bit CPA at the end. With the Dadda tree design, our first goal is to reduce the height of the partial products dot matrix from 4 to 3, thus necessitating 2 FAs in the first level. These are followed by 2 FAs and 2 HAs in the second level (reducing the height from 3 to 2) and a 6-bit CPA at the end.

Intermediate approaches between those of Wallace and Dadda yield various designs that offer speed-cost trade-offs. For example, it may be that neither the Wallace tree nor the Dadda tree leads to a convenient width for the fast adder. In such cases, a hybrid approach may yield the best results.

Note that the results introduced for carry-save multioperand addition in Chapter 8 apply to the design of partial products reduction trees with virtually no change. The only modifications required stem from the relative shifting of the operands to be added. For example, in Fig. 8.12, we see that in adding seven right-aligned k-bit operands, the CSAs are all k bits wide. In a seven-operand CSA tree of a 7 $\times$ 7 tree multiplier, the input operands appear with shifts of 0 to 6 bits, leading to the input configuration shown at the top of Fig. 11.3. We see that the shifted inputs necessitate somewhat wider blocks at

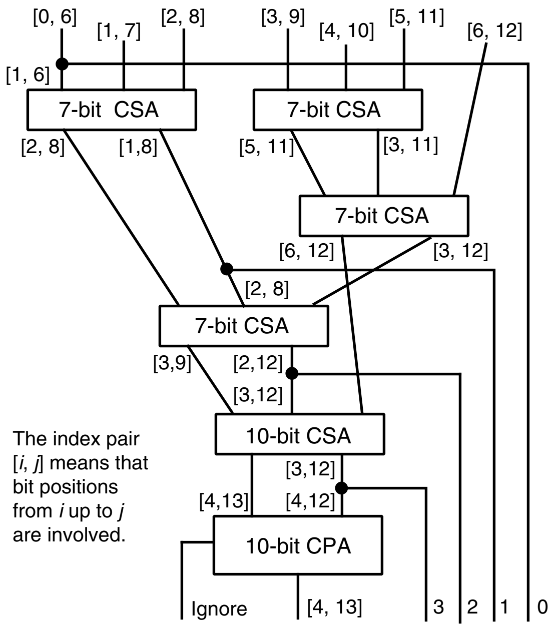


Figure 11.3 Possible CSA tree for a 7×7 tree multiplier.

the bottom of the tree. It is instructive for the reader to compare Fig. 11.3 and Fig. 8.12, noting all the differences.

There is no compelling reason to keep all the bits of the input or intermediate operands together and feed them to multibit CSAs, thus necessitating the use of many HAs that simply rearrange the dots without contributing to their reduction. Doing the reduction with 1-bit FAs and HAs, as in Fig. 11.2, leads to lower complexity and perhaps even greater speed. Deriving the Wallace and Dadda tree multipliers to perform the same function as the circuit of Fig. 11.3 is left as an exercise.

One point is quite clear from Fig. 11.3 or its Wallace tree and Dadda tree equivalents: a logarithmic depth reduction tree based on CSAs has an irregular structure that makes its design and layout quite difficult. Additionally, connections and signal paths of varying lengths lead to logic hazards and signal skew that have implications for both performance and power consumption. In very large-scale integration (VLSI) design, we strive to build circuits from iterated or recursive structures that lend themselves to efficient automatic synthesis and layout. Alternative reduction trees that are more suitable for VLSI implementation are discussed next.

11.2 ALTERNATIVE REDUCTION TREES

Recall from our discussion in Section 8.4 that a (7; 2)-counter slice can be designed that takes 7 bits in the same column i as inputs and produces 1 bit in each of the columns i and $i + 1$ as outputs. Such a slice, when suitably replicated, can perform the function of

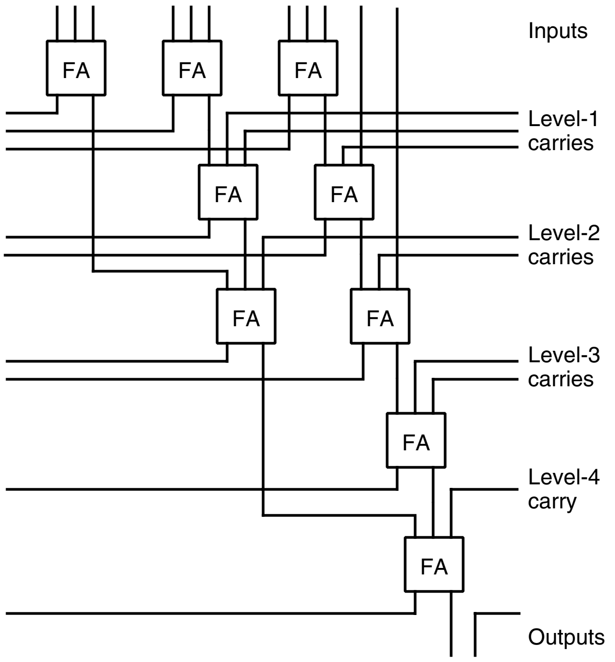


Figure 11.4 A slice of a balanced-delay tree for 11 inputs.

the reduction tree part of Fig. 11.3. Of course, not all columns in Fig. 11.3 have seven inputs. The preceding iterative circuit can then be left intact and supplied with dummy 0 inputs in the interest of regularity, or it can be pruned by removing the redundant parts in each slice. Such optimizations are well within the capabilities of automated design tools.

Based on Table 8.1, an (11; 2)-counter has at least five FA levels. Figure 11.4 shows a particular five-level arrangement of FAs for performing 11-to-2 reduction with the property that all outputs are produced after the same number of FA delays. Observe how all carries produced in level i enter FAs in level $i + 1$. The FAs of Fig. 11.4 can be laid out to occupy a narrow vertical slice that can then be replicated to form an 11-input reduction tree of desired width. Such balanced-delay trees are quite suitable for VLSI implementation of parallel multipliers.

The circuit of Fig. 11.4 is composed of three columns containing one, three, and five FAs, going from left to right. It is now easy to see that the number of inputs can be expanded from 11 to 18 by simply appending to the right of the circuit an additional column of seven FAs. The top FA in the added column will accommodate three new inputs, while each of the others, except for the lowermost two, can accept one new input; these latter FAs must also accommodate a sum coming from above and a carry coming from the right. Note that the FAs in the various columns are more or less independent in that adjacent columns are linked by just one wire. This property makes it possible to lay out the circuit in a narrow slice without having to devote a lot of space to the interconnections.

Instead of building partial products reduction trees from CSAs, or (3; 2)-counters, one can use a module that reduces four numbers to two as the basic building block. Then,

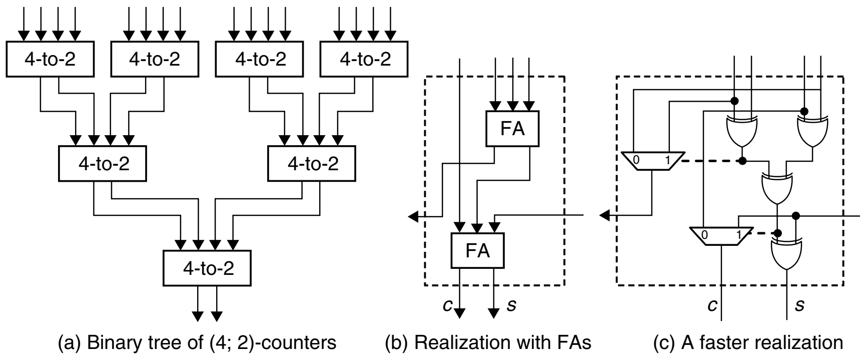


Figure 11.5 Tree multiplier with a more regular structure based on 4-to-2 reduction modules.

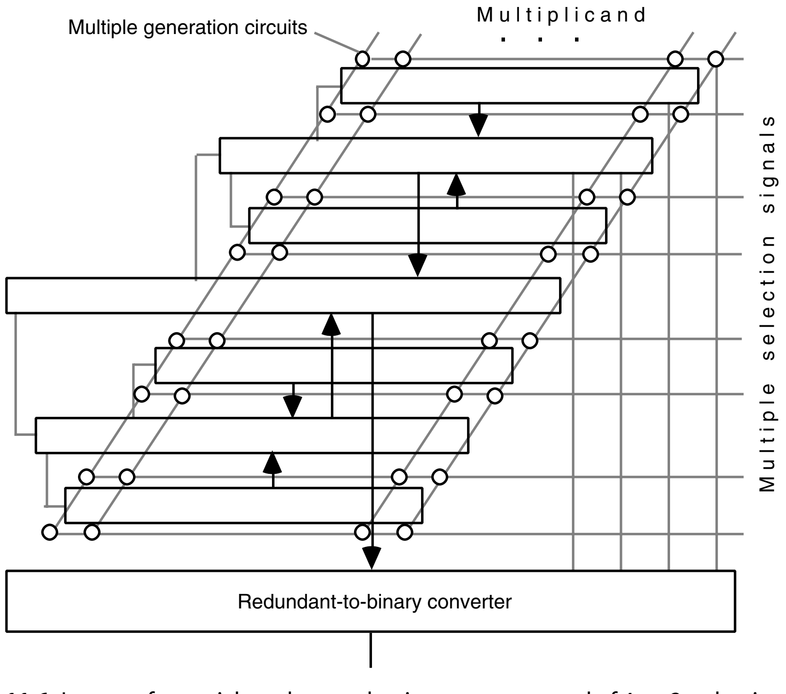

Multiple generation circuits
Multiplicand
. . .
Multiple selection signals
Redundant-to-binary converter

Figure 11.6 Layout of a partial products reduction tree composed of 4-to-2 reduction modules. Each solid arrow represents two numbers.

partial products reduction trees can be structured as binary trees that possess a recursive structure making them more regular and easier to lay out (Fig. 11.5a). Figure 11.6 shows a possible way of laying out the seven-module tree of Fig. 11.5a. Note that adding a level to the tree of Fig. 11.6 involves duplicating the tree and inserting a 4-to-2 reduction module between them.

In Fig. 11.6, the first, third, fifth, and seventh rectangular boxes correspond to top-level blocks of Fig. 11.5a. These blocks receive four multiples of the multiplicand (two from above and two from below) and reduce them to a pair of numbers for the second and sixth blocks. Each of the latter blocks in turn supplies two numbers to the fourth block, which feeds the redundant-to-binary converter.

If the 4-to-2 reduction modules are internally composed of two CSA levels, as suggested in Fig. 11.5b, then there may be more CSA levels in the binary tree structure than in Wallace or Dadda trees. However, regularity of interconnections, and the resulting efficient layout, can more than compensate for the added logic delays due to the greater circuit depth. Direct realization of 4-to-2 reduction modules from their input/output specifications can lead to more compact and/or faster circuits. The realization depicted in Fig. 11.5c, for example, has a latency of three XOR gate levels, compared with four XOR gate levels that would result from the design of Fig. 11.5b.

Note that a 4-to-2 reduction circuit for binary operands can be viewed as a generalized signed-digit adder for radix-2 numbers with the digit set [0, 2], where the digits are encoded in the following 2-bit code:

$$\text{Zero: } (0,0) \qquad \text{One: } (0,1) \text{ or } (1,0) \qquad \text{Two: } (1,1)$$

A variant of this binary tree reduction scheme is based on binary signed-digit (BSD), rather than carry-save, representation of the partial products [Taka85]. These partial products are combined by a tree of BSD adders to obtain the final product in BSD form. The standard binary result is then obtained via a BSD-to-binary converter, which is essentially a fast subtractor for subtracting the negative component of the BSD number from its positive part. One benefit of BSD partial products is that negative multiples resulting from the sign bit in 2's-complement numbers can be easily accommodated (see Section 11.3). Some inefficiency results from the extra bit used to accommodate the digit signs going to waste for most of the multiples that are positive.

Carry-save and BSD numbers are not the only ones that allow fast reduction via limited-carry addition. Several other digit sets are possible that offer certain advantages depending on technological capabilities and constraints [Parh96]. For example, radix-2 partial products using the digit set [0, 3] lend themselves to an efficient parallel-carries addition process (Fig. 3.11c), while also accommodating three, rather than one or two, multiples of a binary multiplicand. Interestingly, the final conversion from the redundant digit set [0, 3] to [0, 1] is not any harder than conversion from [0, 2] to [0, 1].

Clearly, any method used for building the CSA tree can be combined with radix-2^b Booth's recoding to reduce the tree size. However, for modern VLSI technology, the use of Booth recoding in tree multipliers has been questioned [Vill93]; it seems that the additional CSAs needed for reducing k, rather than k/b, numbers could be less complex than the Booth recoding logic when wiring and the overhead due to irregularity and nonuniformity are taken into account.

11.3 TREE MULTIPLIERS FOR SIGNED NUMBERS

When one is multiplying 2's-complement numbers directly, each of the partial products to be added is a signed number. Thus, for the CSA tree to yield the correct sum of its inputs, each partial product must be sign-extended to the width of the final product. Recall our

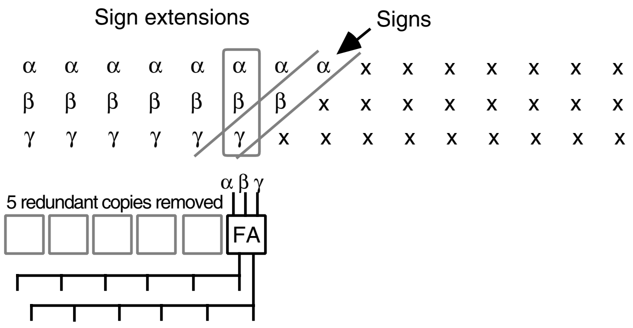


Figure 11.7 Sharing of FAs to reduce the CSA width in a signed tree multiplier.

discussion of signed multioperand addition in Section 8.5, where the 2's-complement operands were assumed to be aligned at their least-significant bits. In particular, refer to Fig. 8.19 for two possible methods based on sign extension (with hardware sharing) and transforming negative bits into positive bits.

Considerations for adding 2's-complement partial products are similar, the only difference being the shifts. Figure 11.7 depicts an example with three sign-extended partial products. We see that here too a single FA can produce the results needed in several different columns. If this procedure is applied to all rows in the partial products bit matrix, the resulting structure will be somewhat more complex than the one assuming unsigned operands. Note that because of the shifts, there are fewer repetitions in Fig. 11.7 than in Fig. 8.19, thus making the expansion in width to accommodate the signs slightly larger.

Another approach, due to Baugh and Wooley [Baug73], is even more efficient and is thus often preferred, in its original or modified form, for 2's-complement multiplication. To understand this method, we begin with unsigned multiplication in Fig. 11.8a and note that the negative weight of the sign bit in 2's-complement representation must be taken into account to obtain the correct product (Fig. 11.8b). To avoid having to deal with negatively weighted bits in the partial products matrix, Baugh and Wooley suggest that we modify the bits in the way shown in Fig. 11.8c, adding five entries to the bit matrix in the process.

Baugh and Wooley's strategy increases the maximum column height by 2, thus potentially leading to greater delay through the CSA tree. For example, in the 5×5 multiplication depicted in Fig. 11.8c, maximum column height is increased from 5 to 7, leading to an extra CSA level. In this particular example, however, the extra delay can be avoided by removing the x_4 entry from column 4 and placing two x_4 entries in column 3, which has only four entries. This reduces the maximum height to 6, which can still be handled by a three-level CSA tree.

To prove the correctness of the Baugh–Wooley scheme, let us focus on the entry $a_4\bar{x}_0$ in Fig. 11.8c. Given that the sign bit in 2's-complement numbers has a negative weight, this entry should have been $-a_4x_0$. We note that

$$-a_4x_0 = a_4(1 - x_0) - a_4 = a_4\bar{x}_0 - a_4$$

Figure 11.8 Baugh–Wooley 2's-complement multiplication.

$$\begin{array}{cccccccccc}
 & & & & & a_4 & a_3 & a_2 & a_1 & a_0 \\
 & & & & & x_4 & x_3 & x_2 & x_1 & x_0 \\
\cline{6-10}
 & & & & & a_4x_0 & a_3x_0 & a_2x_0 & a_1x_0 & a_0x_0 \\
 & & & & a_4x_1 & a_3x_1 & a_2x_1 & a_1x_1 & a_0x_1 & \\
 & & & a_4x_2 & a_3x_2 & a_2x_2 & a_1x_2 & a_0x_2 & & \\
 & & a_4x_3 & a_3x_3 & a_2x_3 & a_1x_3 & a_0x_3 & & & \\
 & a_4x_4 & a_3x_4 & a_2x_4 & a_1x_4 & a_0x_4 & & & & \\
\hline
p_9 & p_8 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}$$

(a) Unsigned multiplication

$$\begin{array}{cccccccccc}
 & & & & & -a_4x_0 & a_3x_0 & a_2x_0 & a_1x_0 & a_0x_0 \\
 & & & & -a_4x_1 & a_3x_1 & a_2x_1 & a_1x_1 & a_0x_1 & \\
 & & & -a_4x_2 & a_3x_2 & a_2x_2 & a_1x_2 & a_0x_2 & & \\
 & & -a_4x_3 & a_3x_3 & a_2x_3 & a_1x_3 & a_0x_3 & & & \\
 & a_4x_4 & -a_3x_4 & -a_2x_4 & -a_1x_4 & -a_0x_4 & & & & \\
\hline
p_9 & p_8 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}$$

(b) 2's-complement bit-matrix

$$\begin{array}{cccccccccc}
 & & & & & a_4\bar{x}_0 & a_3x_0 & a_2x_0 & a_1x_0 & a_0x_0 \\
 & & & & a_4\bar{x}_1 & a_3x_1 & a_2x_1 & a_1x_1 & a_0x_1 & \\
 & & & a_4\bar{x}_2 & a_3x_2 & a_2x_2 & a_1x_2 & a_0x_2 & & \\
 & & a_4\bar{x}_3 & a_3x_3 & a_2x_3 & a_1x_3 & a_0x_3 & & & \\
 & a_4x_4 & \bar{a}_3x_4 & \bar{a}_2x_4 & \bar{a}_1x_4 & \bar{a}_0x_4 & & & & \\
 & \bar{a}_4 & & & & a_4 & & & & \\
1 & \bar{x}_4 & & & & x_4 & & & & \\
\hline
p_9 & p_8 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}$$

(c) Baugh–Wooley method's bit-matrix

$$\begin{array}{cccccccccc}
 & & & & & \overline{a_4x_0} & a_3x_0 & a_2x_0 & a_1x_0 & a_0x_0 \\
 & & & & \overline{a_4x_1} & a_3x_1 & a_2x_1 & a_1x_1 & a_0x_1 & \\
 & & & \overline{a_4x_2} & a_3x_2 & a_2x_2 & a_1x_1 & a_0x_2 & & \\
 & & \overline{a_4x_3} & a_3x_3 & a_2x_3 & a_1x_3 & a_0x_3 & & & \\
 & a_4x_4 & \overline{a_3x_4} & \overline{a_2x_4} & \overline{a_1x_4} & \overline{a_0x_4} & & & & \\
1 & & & & 1 & & & & & \\
\hline
p_9 & p_8 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}$$

(d) Modified Baugh–Wooley method

Hence, we can replace $-a_4x_0$ with the two entries $a_4\bar{x}_0$ and $-a_4$. If instead of $-a_4$ we use an entry a_4, the column sum increases by $2a_4$. To compensate for this, we must insert $-a_4$ in the next higher column. The same argument can be repeated for $a_4\bar{x}_1, a_4\bar{x}_2$, and $a_4\bar{x}_3$. Each column, other than the first, gets an a_4 and a $-a_4$, which cancel each other out. The p_8 column gets a $-a_4$ entry, which can be replaced with $\bar{a}_4 - 1$. The same argument can be repeated for the $\bar{a}_ix_4$ entries, leading to the insertion of x_4 in the p_4 column and $\bar{x}_4 - 1$ in the p_8 column. The two -1s thus produced in the eighth column are equivalent to a -1 entry in the p_9 column, which can in turn be replaced with a 1 and a borrow into the nonexistent (and inconsequential) tenth column.

Another way to justify the Baugh–Wooley method is to transfer all negatively weighted a_4x_i terms, $0 \le i \le 3$, to the bottom row, thus leading to two negative numbers (the preceding number and the one formed by the a_ix_4 bits, $0 \le i \le 3$) in the last two rows. Now, the two numbers x_4a and a_4x must be subtracted from the sum of all the positive elements. Instead of subtracting $x_4 \times a$, we add x_4 times the 2's complement of a, which consists of 1's complement of a plus x_4 (similarly for a_4x). The reader should be able to supply the other details.

A modified form of the Baugh–Wooley method, (Fig. 11.8d) is preferable because it does not lead to an increase in the maximum column height. Justifying this modified form is left as an exercise.

11.4 PARTIAL-TREE AND TRUNCATED MULTIPLIERS

If the cost of a full-tree multiplier is unacceptably high for a particular application, then a variety of mixed serial-parallel designs can be considered. Let h be a number smaller than k. One idea is to perform the k-operand addition needed for $k \times k$ multiplication via $\lceil k/h \rceil$ passes through a smaller CSA tree. Figure 11.9 shows the resulting design that includes an $(h + 2)$-input CSA tree for adding the cumulative partial product (in stored-carry form) and h new operands, feeding back the resulting sum and carry to be combined with the next batch of h operands.

Since the next batch of h operands will be shifted by h bits with respect to the current batch, h bits of the derived sum and $h - 1$ bits of the carry can be relaxed after each pass. These are combined using an h-bit adder to yield h bits of the final product, with the carry-out kept in a flip-flop to be combined with the next inputs. Alternatively, these relaxed bits can be kept in carry-save form by simply shifting them to the right in their respective registers and postponing the conversion to standard binary format to the very end. This is why parts of Fig. 11.9 are rendered in light gray. The latter approach might be followed if a fast double-width adder is already available in the arithmetic/logic unit for other reasons.

Note that the design depicted in Fig. 11.9 corresponds to radix-2^h multiplication. Thus, our discussions in Sections 10.3 and 10.4 are relevant here as well. In fact, the difference between high-radix and partial-tree multipliers is quantitative rather than qualitative (see Fig. 10.13). When h is relatively small, say up to 8 bits, we tend to view the multiplier of Fig. 11.9 as a high-radix multiplier. On the other hand, when h is a significant fraction of k, say $k/2$ or $k/4$, then we view the design as a partial-tree

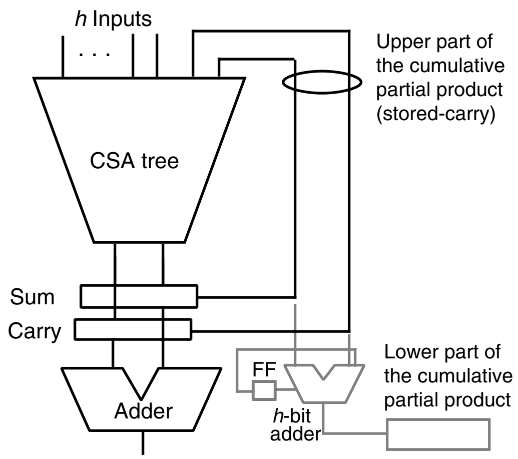

h Inputs
. . .
Upper part of the cumulative partial product (stored-carry)
CSA tree
Sum
Carry
FF
Lower part of the cumulative partial product
Adder
h-bit adder

Figure 11.9 General structure of a partial-tree multiplier.

multiplier. In Section 11.6, we will see that a pipelined variant of the design in Fig. 11.9 can be considerably faster when h is large.

Figure 11.9 has been drawn with the assumption of radix-2 multiplication. If radix-2^b Booth's recoding is applied first to produce one multiple for every b bits of the multiplier, then b times fewer passes are needed and bh bits can be relaxed after each pass. In this case, the small adder in Fig. 11.9 will be bh bits wide.

Thus far, our multipliers were all designed to produce double-width, or full-precision, products. In many applications, a single-width product might be sufficient. Consider, for example, k-bit fractional operands a and x, whose exact product has $2k$ bits. A k-bit fractional result can be obtained by truncating or rounding the double-width result to k bits. However, this might be viewed as wasteful, given that all the bits on the right half of the partial products bit-matrix of Fig. 11.10, to the right of the vertical dashed line, have only a slight impact on the final result. Why not simply drop all those bits to save on the AND gates that produce them and the CSAs that combine and reduce them? Let us see what would happen if we do decide to drop the said bits in the 8×8 multiplication depicted in Fig. 11.10. In the worst case, when all the dropped bits are 1s, we would lose a value equal to $8/2 + 7/4 + 6/8 + 5/16 + 4/32 + 3/64 + 2/128 + 1/256 \approx 7.004\,ulp$, where *ulp* is the weight or worth of the least-significant bit of each operand. If this maximum error of $-7\,ulp$ is tolerable, then the multiplier can be greatly simplified. However, we can do substantially better, will little additional cost.

One way to reduce the error of our truncated multiplier is to keep the first column of dots to the right of the vertical dashed line in Fig. 11.10, dropping only the dots in columns indexed -10 to -16. This modification will improve the error bound computed above by $8/2 = 4\,ulp$ in the partial products accumulation phase, but introduces a possible error of $ulp/2$ when the extra product bit p_{-9} is dropped to form a k-bit final product. Thus, the maximum error is reduced from $7\,ulp$ to $3.5\,ulp$, at the expense of more circuitry to generate and process the eight previously ignored dots. Another possibility is to drop columns -9 and beyond as before, but introduce a compensating 1 term in column -6. The error now ranges from about $-3\,ulp$, when all the dropped bits are 1s, to $4\,ulp$, when

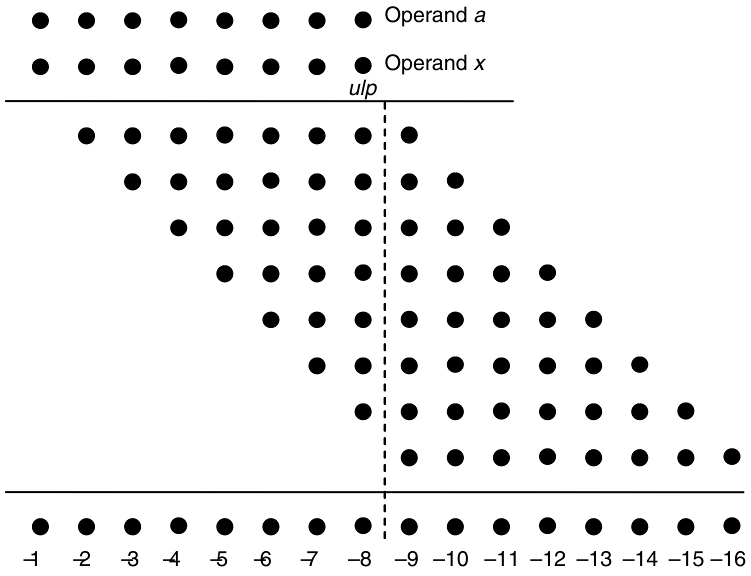


Figure 11.10 The idea of a truncated multiplier with 8-bit fractional operands.

all the dropped bits are 0s. The latter error is comparable in magnitude to that of the preceding method, but it is achieved at a much lower cost. This *constant compensation* method can be further refined to produce better results. Finally, we can resort to *variable compensation*, exemplified by the insertion of two dots with values a_{-1} and x_{-1} (leading bits of the two operands) in column -7. The idea here is to provide greater compensation for the value of the dropped bits when they are more likely to be 1s. Error analysis for this approach is left as an exercise.

11.5 ARRAY MULTIPLIERS

Consider a full-tree multiplier (Fig. 11.1) in which the reduction tree is one-sided and the final adder has a ripple-carry design, as depicted in Fig. 11.11. Such a tree multiplier, which is composed of the slowest possible CSA tree and the slowest possible CPA, is known as an array multiplier.

But why would anyone be interested in such a slow multiplier? The answer is that an array multiplier is very regular in its structure and uses only short wires that go from one FA to horizontally, vertically, or diagonally adjacent FAs. Thus, it has a very simple and efficient layout in VLSI. Furthermore, it can be easily and efficiently pipelined by inserting latches after every CSA or after every few rows (the last row must be handled differently, as discussed in Section 11.6, because its latency is much larger than the others).

The free input of the topmost CSA in the array multiplier of Fig. 11.11 can be used to realize a multiply-add module yielding $p = ax + y$. This is useful in a variety of applications involving convolution or inner-product computation. When only the

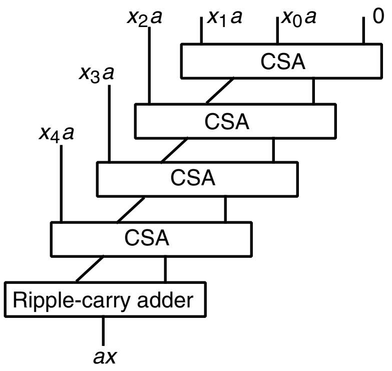
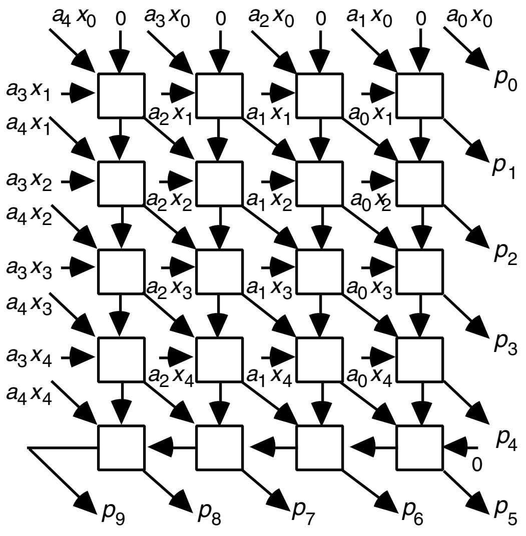


Figure 11.11 A basic array multiplier uses a one-sided CSA tree and a ripple-carry adder.



a_4x_0 0 a_3x_0 0 a_2x_0 0 a_1x_0 0 a_0x_0
a_3x_1 a_4x_1 a_2x_1 a_1x_1 a_0x_1
a_3x_2 a_4x_2 a_2x_2 a_1x_2 a_0x_2
a_3x_3 a_4x_3 a_2x_3 a_1x_3 a_0x_3
a_3x_4 a_4x_4 a_2x_4 a_1x_4 a_0x_4
0
p_0 p_1 p_2 p_3 p_4
p_9 p_8 p_7 p_6 p_5

Figure 11.12 Detailed design of a 5 × 5 array multiplier using FA blocks.

computation of ax is desired, the topmost CSA in the array multiplier of Fig. 11.11 can be removed, with x_0a and x_1a input to the second CSA directly.

Figure 11.12 shows the design of a 5 × 5 array multiplier in terms of FA cells and two-input AND gates. The sum outputs are connected diagonally, while the carry outputs are linked vertically, except in the last row, where they are chained from right to left. The design in Fig. 11.12 assumes unsigned numbers, but it can be easily converted to a 2's-complement array multiplier using the Baugh–Wooley method. This involves adding a FA at the right end of the ripple-carry adder, to take in the a_4 and x_4 terms, and a couple of FAs at the lower left edge to accommodate the $\bar{a}_4, \bar{x}_4$, and 1 terms of Fig. 11.8C (see Fig. 11.13). Most of the connections between FA blocks in Fig. 11.13 have been removed to avoid clutter. The modified diagonal connections in Fig. 11.13 will be described shortly.

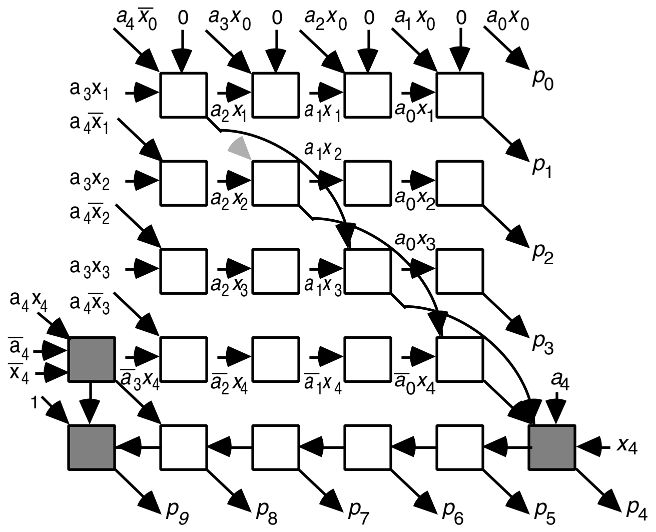


Figure 11.13 Modifications in a 5 × 5 array multiplier to deal with 2's-complement inputs using the Baugh–Wooley method (inclusion of the three shaded FA blocks) or to shorten the critical path (the curved links).

In view of the simplicity of an array multiplier for 2's-complement numbers based on the Baugh–Wooley method, we no longer use techniques proposed by Pezaris [Peza71] and others that required in some of the array positions variants of an FA cell capable of accommodating some negatively weighted input bits and producing one or both outputs with negative weight(s).

If we build a cell containing an FA and an AND gate to internally form the term a_jx_i, the unsigned array multiplier of Fig. 11.12 turns into Fig. 11.14. Here, the x_i and a_j bits are broadcast to rows and columns of cells, with the row-i, column-j cell, forming the term a_jx_i and using it as an input to its FA. If desired, one can make the design less complex by replacing the cells in the first row, or the first two rows, by AND gates.

The critical path through a $k \times k$ array multiplier, when the sum generation logic of an FA block has a longer delay than the carry-generation circuit, goes through the main (top left to bottom right) diagonal in Fig. 11.13 and proceeds horizontally in the last row to the p_9 output. The overall delay of the array multiplier can thus be reduced by rearranging the FA inputs such that some of the sum signals skip rows (they go from row i to row $i + h$ for some $h > 1$). Figure 11.13 shows the modified connections on the main diagonal for $h = 2$. The lower right cell now has one too many inputs, but we can redirect one of them to the second cell on the main diagonal, which now has one free input. Note, however, that such skipping of levels makes for a less regular layout, which also requires longer wires, and hence may not be a worthwhile modification in practice.

Since almost half the latency of an array multiplier is due to the cells in the last row, it is interesting to speculate about whether we can do the final addition faster. Obviously, it is possible to replace the last row of cells with a fast adder, but this would adversely

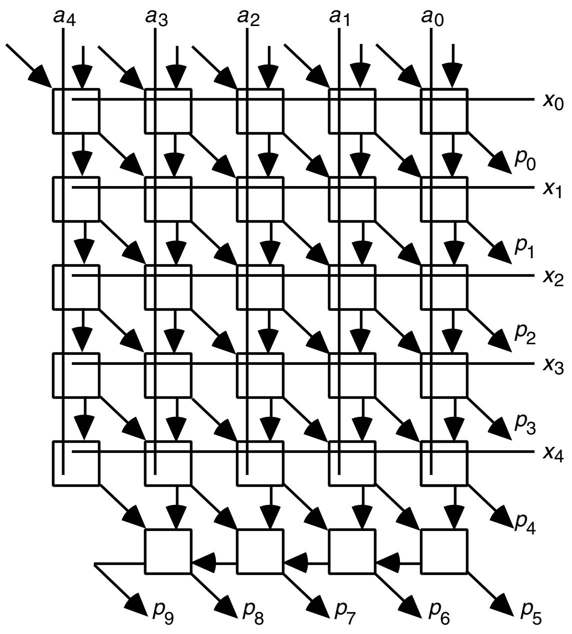


Figure 11.14 Design of a 5 × 5 array multiplier with two additive inputs and FA blocks that include AND gates.

affect the regularity of the design. Besides, even a fast adder is still much slower than the other rows, making pipelining more difficult.

To see how the ripple-carry portion of an array multiplier can be eliminated, let us arrange the k^2 terms $a_j x_i$ in a triangle, with bits distributed in $2k - 1$ columns according to the pattern

$$1 \quad 2 \quad 3 \qquad \cdots \qquad k-1 \qquad k \qquad k-1 \qquad \cdots \qquad 3 \quad 2 \quad 1$$

The least-significant bit of the product is output directly, and the other bits are reduced gradually by rows of FAs and HAs (rectangular boxes in Fig. 11.15). Let us focus on the ith level and assume that the first $i - 1$ levels have already yielded two versions of the final product bits past the B_i boundary, one assuming that the next carry-save addition will produce a carry across B_i and another assuming no carry (Fig. 11.16).

At the ith level, the shaded block in Fig. 11.15 produces two versions of its sum and carry, conditional upon a future carry or no carry across B_{i+1}. The conditional sum bits from the shaded block are simply appended to the i bits coming from above. So, two versions of the upper $i + 1$ bits of the product are obtained, conditional upon the future carry across the B_{i+1} boundary. The process is then repeated in the lower levels, with each level extending the length of the conditional portion by 1 bit and the lowermost multiplexer (mux) providing the last k bits of the end product in nonredundant form.

The conceptual design of Fig. 11.15 can be translated to an actual multiplier circuit after certain optimizations to remove redundant elements [Cimi96], [Erce90].

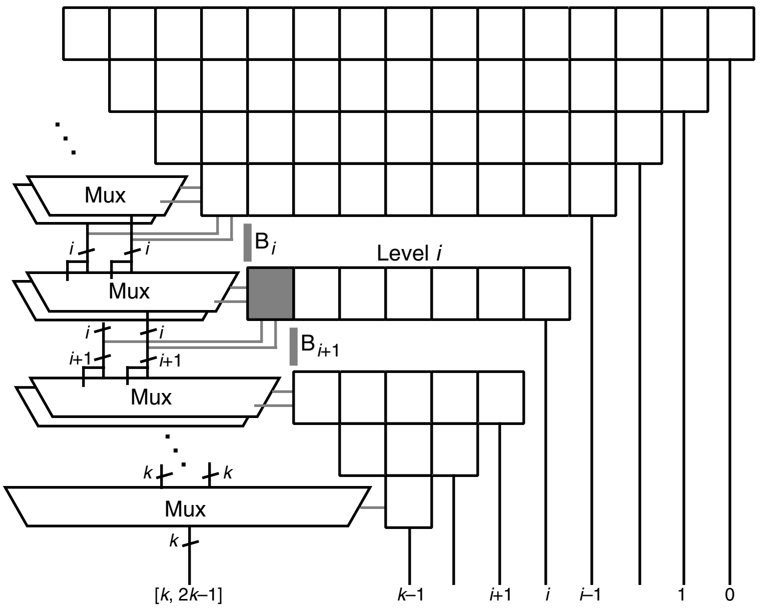


Figure 11.15 Conceptual view of a modified array multiplier that does not need a final CPA.

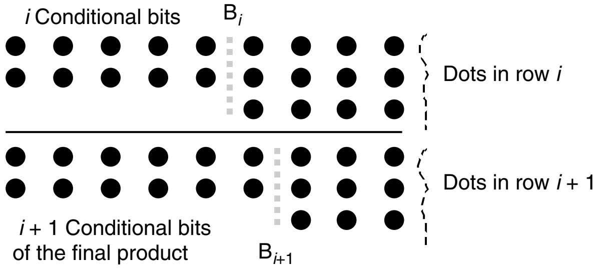


Figure 11.16 Carry-save addition, performed in level i, extends the conditionally computed bits of the final product.

11.6 PIPELINED TREE AND ARRAY MULTIPLIERS

A full-tree multiplier can be easily pipelined. The partial products reduction tree of a full-tree multiplier is a combinational circuit that can be sliced into pipeline stages. A new set of inputs cannot be applied to the partial-tree multiplier of Fig. 11.9, however, until the sum and carry for the preceding set have been latched. Given that for large h, the depth of the tree can be significant, the rate of the application of inputs to the tree, and thus the speed of the multiplier, is limited.

Now, if instead of feeding back the tree outputs to its inputs, we feed them back into the middle of the $(h + 2)$-input tree, as shown in Fig. 11.17, the pipeline rate will be dictated by the delay through only two CSA levels rather than by the depth of the entire tree. This leads to much faster multiplication.

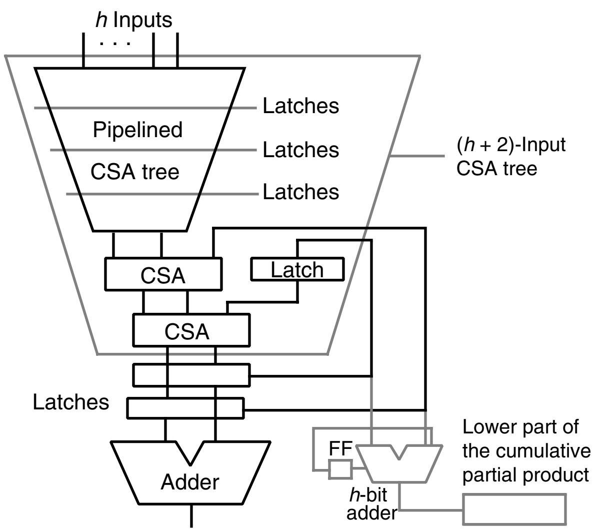
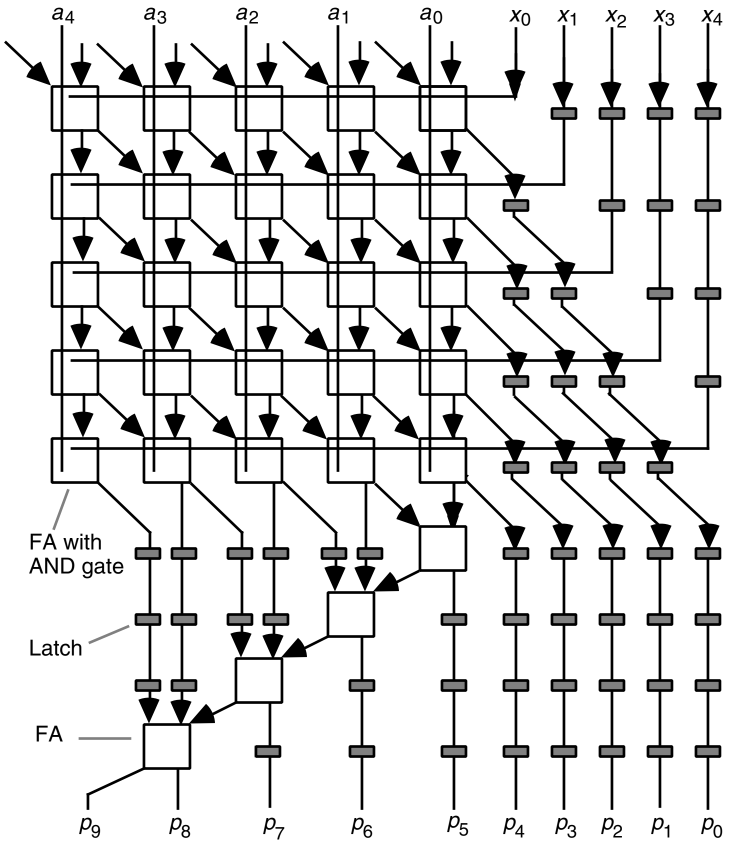


Figure 11.17 Efficiently pipelined partial-tree multiplier.



a_4 a_3 a_2 a_1 a_0 x_0 x_1 x_2 x_3 x_4
FA with
AND gate
Latch
FA
p_9 p_8 p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0

Figure 11.18 Pipelined 5 × 5 array multiplier using latched FA blocks. The small shaded rectangles are latches.

Figure 11.18 shows one way to pipeline an array multiplier. Inputs are applied from above and the product emerges from below after 9 clock cycles ($2k - 1$ in general). All FA blocks used are assumed to have output latches for both sum and carry. Note how the x_i inputs needed for the various rows of the array multiplier are delayed through the insertion of latches in their paths and how the 4-bit ripple-carry adder at the bottom row of Fig. 11.14 has been pipelined in Fig. 11.18.

PROBLEMS

11.1 Unsigned full-tree multipliers

Consider the design of a 7×7 unsigned full-tree multiplier as depicted in Fig. 11.3.

a. Compare Figs. 11.3 and 8.12, discussing all the differences.
b. Design the required partial products reduction tree using Wallace's method.
c. Design the required partial products reduction tree using Dadda's method.
d. Compare the designs of parts a, b, and c with respect to speed and cost.

11.2 Unsigned full-tree multipliers

Consider the design of an 8×8 unsigned full-tree multiplier.

a. Draw a diagram similar to Fig. 11.3 to determine the number and widths of the carry-save adders required.
b. Repeat part a, this time using 4-to-2 reduction circuits built of two CSAs.
c. Design the required partial products reduction tree using Wallace's method.
d. Design the required partial products reduction tree using Dadda's method.
e. Produce one design with its final adder width between those in parts c and d.
f. Compare the designs of parts a–e with respect to speed and cost.

11.3 Balanced-delay trees

Find the relationship between the number n of inputs and circuit depth d of a balanced-delay tree (Fig. 11.4) and show that the depth grows as $\sqrt{n}$.

11.4 Variations in full-tree multipliers

Tabulate the number of full-adder levels in a tree that reduces k multiples of the multiplicand to 2, for $4 \leq k \leq 1024$, using:

a. Carry-save adders as the basic elements.
b. Elements, internally built from two CSA levels, that reduce four operands to two.
c. Same elements as in part b, except that in the first level of the tree only, the use of CSAs is allowed (this is helpful, e.g., for $k = 24$).
d. Discuss the implications of the results of parts a–c in the design of full-tree multipliers.

11.5 Tree multiplier with Booth's recoding

We need a 12 × 12 signed-magnitude binary multiplier. Design the required 11 × 11 unsigned multiplication circuit by first generating a recoded version of the multiplier having six radix-4 digits in $[-2, 2]$ and then adding the six partial products represented in 2's-complement form by a minimal network of FAs. *Hint:* 81 FAs should do.

11.6 Modified Baugh–Wooley method

Prove that the modified Baugh–Wooley method for multiplying 2's-complement numbers, shown in Fig. 11.8d, is correct.

11.7 Signed full-tree multipliers

Consider the design of an 8 × 8 full-tree multiplier for 2's-complement inputs.

a. Draw a diagram similar to Fig. 11.3 to determine the number and widths of the carry-save adders required if the operands are to be sign-extended (Fig. 11.7).
b. Design the 8 × 8 multiplier using the Baugh–Wooley method.
c. Design the 8 × 8 multiplier using the modified Baugh–Wooley method.
d. Compare the designs of parts a–c with respect to speed and cost.

11.8 Partial-tree multipliers

In Fig. 11.9, the tree has been drawn with no intermediate output corresponding to the lower-order bits of the sum of its $h+2$ inputs. If h is large, a few low-order bits of the sum will likely become available before the final sum and carry results. How does this affect the h-bit adder delineated by gray lines?

11.9 Pezaris array multiplier

Consider a 5 × 5 array multiplier, similar to that in Fig. 11.12 but with 2's-complement inputs, and view the AND terms a_4x_i and a_jx_4 as being negatively weighted. Consider also two modified forms of a FA cell: FA$'$ has one negatively weighted input, producing a negatively weighted sum and a positively weighted carry, while FA$''$ has two negatively weighted inputs, producing a negative carry and a positive sum. Design a 5 × 5 Pezaris array multiplier using FA, FA$'$, and FA$''$ cells as needed, making sure that any negatively weighted output is properly connected to a negatively weighted input (use small "bubbles" to mark negatively weighted inputs and outputs on the various blocks). Note that FA$'''$, with all three inputs and two outputs carrying negative weights, is the same as FA. Note also that the output must have only 1 negatively weighted bit at the sign position.

11.10 2's-complement array multipliers

Consider the design of a 5 × 5 2's-complement array multiplier. Assume that an FA block has latencies of T_c and T_s ($T_c < T_s < 2T_c$) for its carry and sum outputs.

a. Find the overall latency for the 5 × 5 array multiplier with the Baugh–Wooley method (Fig. 11.13, regular design without row skipping).
b. Repeat part a with the modified Baugh–Wooley method.
c. Compare the designs in parts a and b and discuss.
d. Generalize the preceding results and comparison to the case of $k \times k$ array multipliers.

11.11 Array multipliers

Design array multipliers for the following number representations.

a. Binary signed-digit numbers using the digit set $[-1, 1]$ in radix 2.
b. 1's-complement numbers.

11.12 Multiply-add modules

Consider the design of a module that performs the computation $p = ax + y + z$, where a and y are k-bit unsigned integers and x and z are l-bit unsigned integers.

a. Show that p is representable with $k + l$ bits.
b. Design a tree multiplier to compute p for $k = 8$ and $l = 4$ based on a Wallace tree and a CPA.
c. Repeat part b using a Dadda tree.
d. Show that an 8 × 4 array multiplier can be readily modified to compute p.

11.13 Pipelined array multipliers

Consider the 5 × 5 pipelined array multiplier in Fig. 11.18.

a. Show how the four lowermost FAs and the latches immediately above them can be replaced by a number of latched HAs. *Hint:* Some HAs will have to be added in the leftmost column, corresponding to p_9, which currently contains no element.
b. Compare the design in part a with the original design in Fig. 11.18.
c. Redesign the pipelined multiplier in Fig. 11.18 so that the combinational delay in each pipeline stage is equal to two FA delays (ignore the difference in delays between the sum and carry outputs).
d. Repeat part c for the array multiplier derived in part a.
e. Compare the array multiplier designs of parts c and d with respect to throughput and throughput/cost. State your assumptions clearly.

11.14 Effectiveness of Booth's recoding

As mentioned at the end of Section 11.2, the effectiveness of Booth recoding in tree multipliers has been questioned [Vill93]. Booth's recoding essentially reduces the number of partial products by a factor of 2. A (4, 2) reduction circuit built, for example, from two CSAs offers the same reduction. Show through a simple approximate analysis of the delay and cost of a $k \times k$ unsigned multiplier based on Booth's recoding and (4, 2) initial reduction that Booth's recoding has the edge in terms of gate count but that it may lose on other grounds. Assume, for simplicity, that k is even.

11.15 VLSI implementation of tree multipliers

Wallace and Dadda trees tend to be quite irregular and thus ill-suited to compact VLSI implementation. Study the bit-slice implementation method for tree multipliers suggested in [Mou92] and apply it to the design of a 12×12 multiplier.

11.16 Faster array multipliers

Present the complete design of an 8×8 array multiplier built without a final CPA (Fig. 11.15). Compare the resulting design with a simple 8×8 array multiplier with respect to speed, cost, and cost-effectiveness.

11.17 Pipelined partial-tree multipliers

a. Would it be cost-effective to implement an 8×8 unsigned multiplier using the pipelined design of Fig. 11.17 with $h = 4$?

b. With reference to the VLSI complexity discussions in Section 10.6, show that the multiplication time in a pipelined partial-tree multiplier is $\mathrm{O}(k/h + \log k)$.

11.18 Pipelined tree multipliers

Stating all your assumptions clearly and taking k/h to be an integer, compare the times needed to add k operands by means of the circuits in Figs. 11.9 and 11.17. Provide concrete numbers for the special case of $k = 24$, with $h = 4$ and $h = 6$.

11.19 Signed tree multipliers

a. Prove that the three terms $a_{k-1}x_{k-1}, \overline{a}_{k-1}$, and $\overline{x}_{k-1}$ in the next to leftmost column of Fig. 11.8c can be replaced by the two terms 1 and $\overline{a}_{k-1}\overline{x}_{k-1}$ in the same column [Blan74].

b. Prove that the four entries in the leftmost two columns of Fig. 11.8c can be replaced by a single term $a_{k-1} \vee x_{k-1}$ in each of the two columns [Blan74].

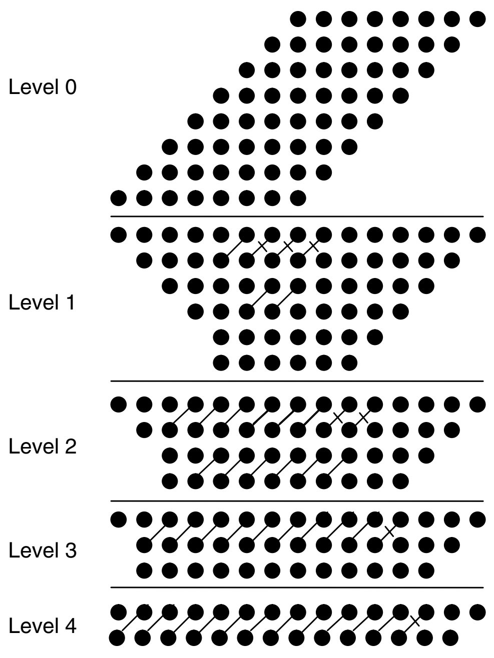
c. Show how the (modified) Baugh–Wooley method works for a $k \times m$ multiplication, $k > m$. As an example, construct the equivalent of Fig. 11.8 in the case of 5×3 multiplication.

d. Formulate the modifications of parts a and b for the more general case given in part c.

11.20 Mystery diagram

a. Explain what the following diagram signifies.

b. In the hardware unit described by the diagram of part a, what types of components are used and how many of each?



11.21 Wallace-tree multipliers

a. Design a (4; 3)-counter with latency comparable to a (3; 2)-counter built of two HAs and an OR gate.

b. Show that a single (4; 3)-counter of the type designed in part a can be used, along with conventional (3; 2)-counters, to build a 5×5 multiplier that is faster than one using a pure Wallace tree.

c. Repeat part b for a 14×14 multiplier. *Hint:* Handle the last five rows in the triangular partial products bit-matrix as in part b.

d. Derive general synthesis guidelines for optimized Wallace trees that include a single (4; 3)-counter of the type designed in part a [Robi98].

11.22 Unsigned/2's-complement tree multiplier

Most processors allow both unsigned and signed numbers as operands in their integer arithmetic units. Discuss the design of a $k \times k$ tree multiplier that can act as an unsigned multiplier (for $t = 0$) or 2's-complement multiplier (for $t = 1$), where t is a control signal.

11.23 Overflow detection in multipliers

Our discussion of truncated multipliers in Section 11.4 assumed the generation of a single-width product with fractional input operands. If a single-width product is desired with integer inputs, the possibility of overflow must be taken into account. Study the problem of overflow detection in such integer multipliers and prepare a two-page report outlining the design techniques and the performance penalty, if any [Gok06].

11.24 Alternate design for array multipliers

In the one-sided tree of Fig. 11.11, we can replace each CSA with a ripple-carry adder that forwards a single value (the sum of its two inputs) to the next row.

a. Draw a diagram, similar to Fig. 11.12, to show the new design for 5×5 multiplication.
b. Compare the design of part a with that shown in Fig. 11.12 with respect to cost, delay, and cost-effectiveness.
c. Generalize the discussion of part b to $k \times k$ array multipliers of the two designs.

REFERENCES AND FURTHER READINGS

[Baug73] Baugh, C. R., and B. A. Wooley, "A Two's Complement Parallel Array Multiplication Algorithm," *IEEE Trans. Computers*, Vol. 22, pp. 1045–1047, 1973.

[Bewi94] Bewick, G. W., "Fast Multiplication: Algorithms and Implementation," PhD dissertation, Stanford University, 1994.

[Blan74] Blankenship, P. E., "Comments on 'A Two's Complement Parallel Array Multiplication Algorithm,'" *IEEE Trans. Computers*, Vol. 23, p. 1327, 1974.

[Cimi96] Ciminiera, L., and P. Montuschi, "Carry-Save Multiplication Schemes Without Final Addition," *IEEE Trans. Computers*, Vol. 45, No. 9, pp. 1050–1055, 1996.

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

[Erce90] Ercegovac, M. D., and T. Lang, "Fast Multiplication Without Carry-Propagate Addition," *IEEE Trans. Computers*, Vol. 39, No. 11, pp. 1385–1390, 1990.

[Gok06] Gok, M., M. J. Schulte, and M. G. Arnold, "Integer Multipliers with Overflow Detection," *IEEE Trans. Computers*, Vol. 55, No. 8, pp. 1062–1066, 2006.

[Mou92] Mou, Z.-J., and F. Jutand, "'Overturned-Stairs' Adder Trees and Multiplier Design," *IEEE Trans. Computers*, Vol. 41, No. 8, pp. 940–948, 1992.

[Parh96] Parhami, B., "Comments on 'High-Speed Area-Efficient Multiplier Design Using Multiple-Valued Current Mode Circuits,'" *IEEE Trans. Computers*, Vol. 45, No. 5, pp. 637–638, 1996.

[Peza71] Pezaris, S. D., "A 40-ns 17-Bit by 17-Bit Array Multiplier," *IEEE Trans. Computers*, Vol. 20, pp. 442–447, 1971.

[Robi98] Robinson, M. E., and E. Swartzlander Jr., "A Reduction Scheme to Optimize the Wallace Multiplier," *Proc. Int'l Conf. Computer Design*, pp. 122–127, 1998.

[Schu93] Schulte, M. J., and E. E. Swartzlander, Jr., "Truncated Multiplication with Correction Constant," in *VLSI Signal Processing VI*, pp. 388–396, 1993.

[Swar99] Swartzlander, E. E., “Truncated Multiplication with Approximate Rounding,” *Proc. 33rd Asilomar Conf. Signals Systems and Computers*, pp. 1480–1483, 1999.

[Taka85] Takagi, N., H. Yasuura, and S. Yajima, “High-Speed VLSI Multiplication Algorithm with a Redundant Binary Addition Tree,” *IEEE Trans. Computers*, Vol. 34, No. 9, pp. 789–796, 1985.

[Town03] Townsend, W. J., E. E. Swartzlander, and J. A. Abraham, “A Comparison of Dadda and Wallace Multiplier Delays,” *Proc. SPIE Conf. Advanced Signal Processing: Algorithms, Architectures, and Implementations*, pp. 552–560, 2003.

[Vill93] Villager, D., and V. G. Oklobdzija, “Analysis of Booth Encoding Efficiency in Parallel Multipliers Using Compressors for Reduction of Partial Products,” *Proc. Asilomar Conf. Signals, Systems, and Computers*, pp. 781–784, 1993.

[Vuil83] Vuillemin, J., “A Very Fast Multiplication Algorithm for VLSI Implementation,” *Integration: The VLSI Journal*, Vol. 1, pp. 39–52, 1983.

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

[Zura86] Zuras, D., and W. H. McAllister, “Balanced Delay Trees and Combinatorial Division in VLSI,” *IEEE J. Solid-State Circuits*, Vol. 21, pp. 814–819, 1986.