Chapter 4

Multiplication

In this chapter, the hardware implementations of parallel multipliers are described. The basis for all of these implementations is the add-and-shift algorithm, which is similar to the way one multiplies using pencil and paper. For example, in multiplying two numbers, each bit of the multiplier requires a corresponding add-and-shift operation.

Multiplicand	110	(6)
Multiplier	$\times$101	$\times$ (5)
	110	(6×2^0)
Partial products	000	(0×2^1)
	110	(6×2^2)
Final product	11110	(30)

Figure 4.1a illustrates the concept for multiplication of two 8-bit operands, and Figure 4.1b introduces a convenient dot representation of the same multiplication. In this chapter, we will describe the three major categories of parallel multiplier implementation:

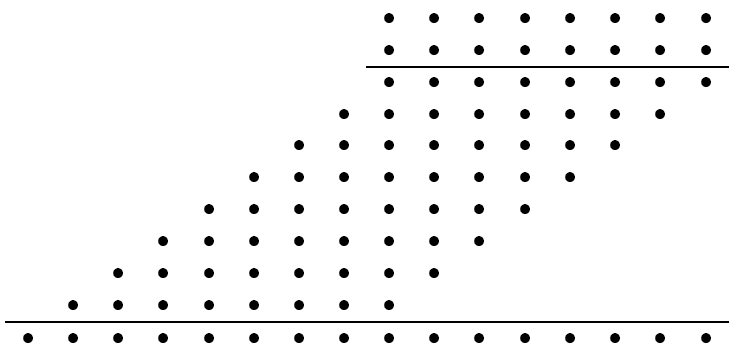
- Simultaneous generation of partial products and simultaneous reduction.
- Simultaneous generation of partial products and iterative reduction.
- Iterative arrays of cells.

4.1 Simultaneous Matrix Generation and Reduction

This scheme is made of two distinct steps. In the first step, the partial products are generated simultaneously, and in the second step, the resultant matrix is reduced to the final product. Since the algorithms for each step are mostly independent of each other, we will describe them separately.

$$
\begin{array}{cccccccccccccccc}
 & & & & & & & X7 & X_6 & X_5 & X_4 & X_3 & X_2 & X_1 & X_0 & \leftarrow \text{MULTIPLICAND} \\
 & & & & & & & Y7 & Y_6 & Y_5 & Y_4 & Y_3 & Y_2 & Y_1 & Y_0 & \leftarrow \text{MULTIPLIER} \\
\hline
 & & & & & & & A7 & A_6 & A_5 & A_4 & A_3 & A_2 & A_1 & A_0 & \leftarrow \text{A PARTIAL PRODUCT} \\
 & & & & & & B7 & B_6 & B_5 & B_4 & B_3 & B_2 & B_1 & B_0 & & \\
 & & & & & C7 & C_6 & C_5 & C_4 & C_3 & C_2 & C_1 & C_0 & & & \\
 & & & & D7 & D_6 & D_5 & D_4 & D_3 & D_2 & D_1 & D_0 & & & & \\
 & & & E7 & E_6 & E_5 & E_4 & E_3 & E_2 & E_1 & E_0 & & & & & \\
 & & F7 & F_6 & F_5 & F_4 & F_3 & F_2 & F_1 & F_0 & & & & & & \\
 & G7 & G_6 & G_5 & G_4 & G_3 & G_2 & G_1 & G_0 & & & & & & & \\
H7 & H_6 & H_5 & H_4 & H_3 & H_2 & H_1 & H_0 & & & & & & & & \\
\hline
S15 & S_14 & S_13 & S_12 & S_11 & S_10 & S_9 & S_8 & S_7 & S_6 & S_5 & S_4 & S_3 & S_2 & S_1 & S_0 & \leftarrow \text{FINAL PRODUCT}
\end{array}
$$

(a)



(b)

Figure 4.1: (a) Multiplying two 8-bit operands results in eight partial products which are added to form a 16-bit final product. (b) Dot representation of the same multiplication.

4.1.1 Partial Products Generation: Booth's Algorithm

The simplest way to generate partial products is to use AND gates as 1×1 multipliers. For example, in Figure 4.1a:

$$A_0 = Y_0X_0, A_1 = Y_0X_1, B_0 = Y_1X_0,$$

and so on. In this manner, an n-bit multiplier generates n partial products. However, it is possible to use encoding techniques that reduce the number of partial products. The modified Booth's algorithm is such an encoding technique, which reduces the number of partial products by half.

The original Booth's algorithm [2] allows the multiplication operation to skip over any contiguous string of all 1's and all 0's in the multiplier, rather than form a partial product for each bit. Skipping a string of 0's is straightforward, but in skipping over a string of 1's the following property is put to use: a string of 1's can be evaluated by subtracting the weight of the rightmost 1 from the modulus. A string of n 1's is the same as 1 followed by n 0's less 1. For example, the value of the binary string 11100 computes to $2^5 - 2^2 = 28$ (i.e., $100,000 - 100$).

A modified version of Booth's algorithm is more commonly used. The difference between the

Bit			Operation	
2^1	2^0	2^{-1}		
Y_{i+1}	Y_i	Y_{i-1}		
0	0	0	Add zero (no string)	$+0$
0	0	1	Add multiplicand (end of string)	$+X$
0	1	0	Add multiplicand (a string)	$+X$
0	1	1	Add twice the multiplicand (end of string)	$+2X$
1	0	0	Subtract twice the multiplicand (beginning of string)	$-2X$
1	0	1	Subtract the multiplicand ($-2X$ and $+X$)	$-$X
1	1	0	Subtract the multiplicand (beginning of string)	$-X$
1	1	1	Subtract zero (center of string)	-0

Table 4.1: Encoding 2 multiplier bits by inspecting 3 bits, in the modified Booth's algorithm.

Booth's and the modified Booth's algorithm is as follows: the latter always generates $n/2$ independent partial products, whereas the former generates a varying (at most $n/2+1$) number of partial products, depending on the bit pattern of the multiplier. Of course, parallel hardware implementation lends itself only to the fixed independent number of partial products. The modified multiplier encoding scheme encodes 2-bit groups and produces five partial products from an 8-bit (unsigned numbers) multiplier, the fifth partial product being a consequence of using two's complement representation of the partial products. (Only four partial products are generated if only two's complement input representation is used, as the most significant input bit represents the sign.)

Each multiplier is divided into substrings of 3 bits, with adjacent groups sharing a common bit. Booth's algorithm can be used with either unsigned or two's complement numbers (the most significant bit of which has a weight of -2^n), and requires that the multiplier be padded with a 0 to the right to form four complete groups of 3 bits each. To work with unsigned numbers, the n-bit multiplier must also be padded with one or two zeros in the multipliers to the left. Table 4.1, from Anderson [1], is the encoding table of the eight permutations of the three multiplier bits.

In using Table 4.1, the multiplier is partitioned into 3-bit groups with one bit shared between groups. If this shared bit is a 1, subtraction is indicated, since we prepare for a string of 1's. Consider the case of unsigned (i.e., positive) numbers; let X represent the multiplicand (all bits) and $Y = Y_{n-1}, Y_{n-2}, \ldots Y_1, Y_0$ an integer multiplier—the binary point following Y_0. (The placement of the point is arbitrary, but all actions are taken with respect to it.) The lowest order action is derived from multiplier bits $Y_1Y_0.0$—the LSB has been padded with a zero. Only four actions are possible: $Y_1Y_0.0$ may be either 00.0, 01.0, 10.0, or 11.0. The first two cases are straightforward; for 00.0, the partial product is 0; for 01.0, the partial product is $+X$. The other two cases are perceived as the beginning of a string of 1's. Thus, we subtract $2X$ (i.e., add $-2X$) for the case 10.0, and subtract X for the case 11.0. Higher order actions must recognize that this subtraction has occurred. The next higher action is found from multiplier bits $Y_3Y_2Y_1$ (remember, Y_1 is the shared bit). Its action on the multiplicand has 4 times the significance of $Y_1Y_0.0$. Thus, it uses the table as $Y_3Y_2Y_1$, but resulting actions are shifted by 2 (multiplied by 4). Thus, suppose the multiplier was 0010.0; the first action (10.0) would detect the start of a string of 1's and subtract $2X$, while the second action (00.1) would detect the end of a string of 1's and add X. But the second action has a scale or significance point 2 bits higher than the first action (4 times more significant). Thus, $4 \times X - 2X = 2X$, the value of the multiplier, (0010.0). This may seem to the reader to be a lot of work to simply find $2X$, and, indeed, in

A_9	A_9	A_9	A_9	A_9	A_9	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
B_9	B_9	B_9	B_9	B_9	B_8	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0		Y_1
C_9	C_9	C_9	C_8	C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0		Y_3		
D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0		Y_5				
E_7	E_6	E_5	E_4	E_3	E_2	E_1	E_0		Y_7						
S_{15}	S_{14}	S_{13}	S_{12}	S_{11}	S_{10}	S_9	S_8	S_7	S_6	S_5	S_4	S_3	S_2	S_1	S_0

Figure 4.2: Generation of five partial products in 8×8 multiplication, using modified Booth's algorithm (only four partial products are generated if the representation is restricted to two's complement).

this case two actions were required rather than one. By inspection of the table, however, only one action (addition or subtraction) is required for each *two* multiplier bits. Thus, use of the algorithm insures that for an n-bit multiplier only $n/2$ actions will be required for any multiplier bit pattern.

For the highest order action with an unsigned multiplier, the action must be derived with a leading or padded zero. For an odd number of multiplier bits, the last action will be defined by $0Y_{n-1}.Y_{n-2}$. For an even number of multiplier bits, $\frac{n}{2}+1$ actions are required, the last action being defined by $00.Y_{n-1}$.

Multipliers in two's complement form may be used directly in the algorithm. In this case, the highest order action is determined by $Y_{n-1}Y_{n-2}Y_{n-3}$ (no padding) for an even number of multiplier bits, and $Y_{n-1}Y_{n-1}.Y_{n-2}$, a sign extended group, for odd sized multipliers; e.g., suppose $Y = -1$. In two's complement, $Y = 1111\ldots11$. The lowest order action (11.0) is $-X$; all other actions (11.1) are -0, producing the desired result $(-X)$.

In implementing the actions, the $2X$ term is simply a 1-bit left shift of X. Thus, multiplicands must be arranged to be gated directly with respect to a scale point or shifted one bit. Subtraction is implemented by gating the complement of X (i.e., the 1's complement) and then adding a 1 with respect to the scale point. In implementing this, the Y_{i+1} can be used as a subtractor indicator that will be added to the LSB of the partial product. If bit $Y_{i+1} = 0$, no subtraction is called for, and adding 0 changes nothing. On the other hand, if bit $Y_{i+1} = 1$, then subtraction is called for and the proper two's complement is performed by adding 1 to the LSB. Of course, in the two's complement, the sign bit must be extended to the full width of the final result, as shown by the repetitive terms in Figure 4.2.

In Figure 4.2, if X and Y are 8-bit unsigned numbers, then A_8–A_0 are determined by the $Y_1Y_0.0$ action on X. Since $2X$ is a possible action, A_8 may be affected and A_9 is the sign and extension. If a $\pm X$ action is determined, then A_8 is the sign and $A_8 = A_9$. The fifth action is determined by $00.Y_7$—always a positive, unshifted action ($+X$ or $+0$). If X and Y are 8-bit two's complement numbers, then $A_8 = A_9$ are the sign and extension. Also, no fifth action $(E_7 - E_0)$ is required.

The A_9, B_9, C_9, D_9 terms appear to significantly increase the hardware required for partial product addition. There are 16 such terms. While the full addition of these terms results in the correct product formations, $S_{15} - S_0$, simpler implementations are possible. By recognizing the $A_9, B_9, \ldots$ terms are sign identifiers and generating the sign logic separately, the additional summing hardware for such terms can be eliminated.

The additional gate delay in implementing the modified Booth's algorithm consists of four gates: two gates for decoding the 3-bit multiplier and two gates for selecting X or $2X$.

2^2	2^1	2^0	2^{-1}	
Y_{i+2}	Y_{i+1}	Y_i	Y_{i-1}	Operation
0	0	0	0	$+0$
0	0	0	1	$+X$
0	0	1	0	$+X$
0	0	1	1	$+2X$
0	1	0	0	$+2X$
0	1	0	1	$+3X$
0	1	1	0	$+3X$
0	1	1	1	$+4X$
1	0	0	0	$-4X$
1	0	0	1	$-3X$
1	0	1	0	$-3X$
1	0	1	1	$-2X$
1	1	0	0	$-2X$
1	1	0	1	$-X$
1	1	1	0	$-X$
1	1	1	1	-0

Table 4.2: Extension of the modified Booth's algorithm to 3-bit multiplier group encoding. This requires generation of $\pm 3X$, which is not as trivial as the operations in the 2-bit multiplier encoding. (Compare with Table 4.1.)

An extension of the modified Booth's algorithm involves an encoding of three bits at a time while examining four multiplier bits. This scheme would generate only $n/3$ partial products. However, the encoding truth table in Table 4.2 requires the generation of a three timesmultiplicand term, which is not as trivial as generating twice the multiplicand. Thus, most hardware implementations use only the 2-bit encoding.

Example 4.1

Suppose the multiplicand (X) is to be multiplied by an unsigned Y:

$$0011101011$$

That is, decimal 235.

We use modified Booth's algorithm (Table 4.1) and assume that this multiplier is a binary integer with point indicated by (.). Now the multiplier must be decomposed into overlapping 3-bit segments and actions determined for each segment. Note that the first segment has an implied "0" to the right of the binary point. Thus, we can label each segment as follows:

$$\begin{array}{cccccccccccc} & & \overline{(5)} & & & \overline{(3)} & & & \overline{(1)} & & & \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & . & 0 \\ & & & \underline{(4)} & & & \underline{(2)} & & & & & \end{array}$$

While segment (1) is referenced to the original binary point, segment (2) is *four times more significant*. Thus, any segment (2) action on X (the multiplicand) must be scaled by a factor of four. Similarly, segment (3) is four times more significant than 2, and 16 times more significant than 1.

Now, by using the table and scaling as appropriate, we get the following actions:

Segment number	Bits	Action	Scale factor	Result
(1)	110	$-X$	1	$-X$
(2)	101	$-X$	4	$-4X$
(3)	101	$-X$	16	$-16X$
(4)	111	0	64	0
(5)	001	$+X$	256	$+256X$
		Total action		$235X$

Note that the table of actions can be simplified for the first segment (Y_{i-1} always 0) and the last segment (depending on whether there is an even or odd number of bits in the multiplier).

Also note that the actions specified in the table are independent of one another. Thus, the five result actions in the example may be summed simultaneously using the carry-save addition techniques discussed in the last chapter. $\Diamond$

4.1.2 Using ROMs to Generate Partial Products

Another way to generate partial products is to use ROMs. For example, the 8×8 multiplication of Figure 4.1 can be implemented using four 256×8 ROMs, where each ROM performs 4×4 multiplication, as shown in Figure 4.3.

In Figure 4.3, each 4-bit value of each element of the pair (Y_A, X_A) (Y_B, X_A) (Y_A, X_B) and (Y_B, X_B) is concatenated to form an 8-bit address into the 256 entry ROM table. The entry contains the corresponding 8-bit product. Thus, four tables are required to simultaneously form the products: $Y_A \cdot X_A$, $Y_B \cdot X_A$, $Y_A \cdot X_B$, and $Y_B \cdot X_B$. Note that the $Y_A \cdot X_A$ and the $Y_B \cdot X_B$ terms have disjoint significance; thus, only three terms must be added to form the product. The number of rearranged partial products that must be summed is referred to as the matrix height—the number of initial inputs to the CSA tree.

A generalization of the ROM scheme is shown in Figure 4.4 [35] for various multiplier arrays of up to 64×64. In the latter case, 256 partial products are generated. But upon rearranging, the maximum column height of the matrix is 31. Table 4.3 summarizes the partial products matrix.

These partial products can be viewed as adjacent columns of height h. Now we are ready to discuss the implementations of column reductions.

4.1.3 Partial Products Reduction

As mentioned in the last chapter, the common approach in all of the summand reduction techniques is to reduce the n partial products to two partial products. A carry look ahead is then used to add these two products. One of the first reduction implementations was the Wallace tree [36], where carry save adders are used to reduce 3 bits of column height to 2 bits (Figure 4.5). In general, the number of the required carry save adder levels (L) in a Wallace tree to reduce

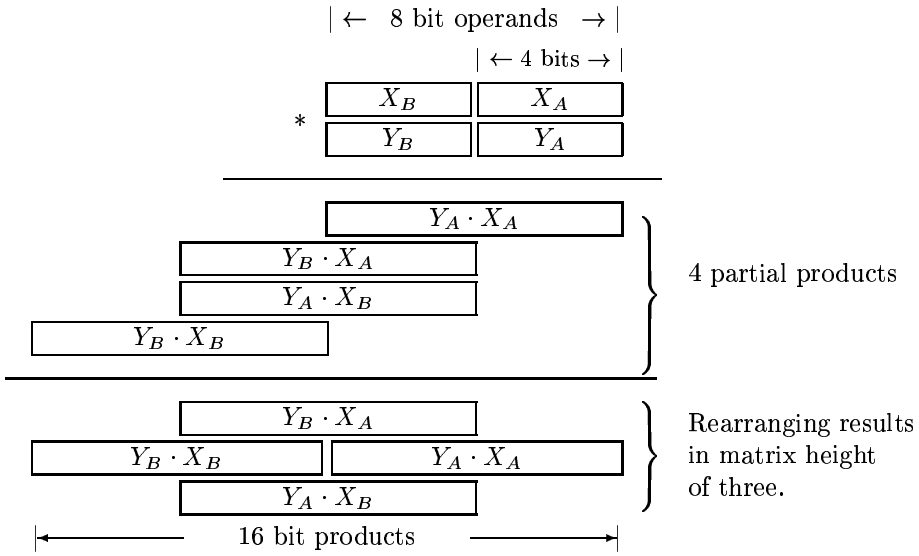


Figure 4.3: Implementation of 8×8 multiplication using four 256×8 ROMs, where each ROM performs 4×4 multiplication.

Table 4.3: Summary of maximum height of the partial products matrix for the various partial generation schemes where n is the multiple size.

Scheme	General Formula	MAX Height of the Matrix Number of Bits 8	16	24	32	40	48	56	64
1×1 multiplier (AND gate)	n	8	16	24	32	40	48	56	64
4×4 multiplier (ROM)	$(n/2) - 1$	3	7	11	15	19	23	27	31
8×8 multiplier (ROM)	$(n/4) - 1$	1	3	4	7	9	11	13	15
Modified Booth's algorithm	$(n/2)$	4	8	12	16	20	24	28	32

height h to 2 is:

$$L \doteq \left\lceil \log_{3/2}\left(\frac{h}{2}\right)\right\rceil = \left\lceil \log_{1.5}\left(\frac{h}{2}\right)\right\rceil,$$

where h is the number of operands (actions) to be summed and L is the number of CSA stages of delay required to produce the pair of column operands. For 8×8 multiplication using 1×1 multiplier generation, $h = 8$ and four levels of carry-save-adders are required, as illustrated in Figure 4.6. Following we show the number of levels versus various column heights:

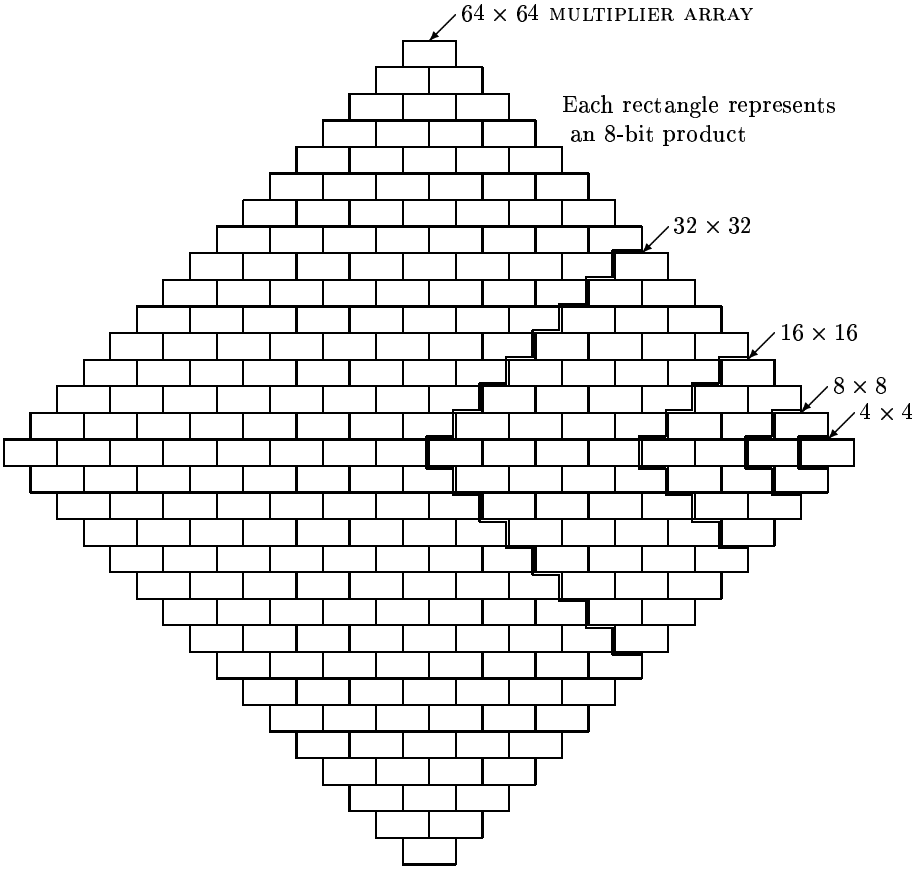


Figure 4.4: Using ROMs for various multiplier arrays for up to 64×64 multiplication. Each ROM is a 4×4 multiplier with 8-bit product. Each rectangle represents the 8-bit partial product ($h = 31$).

Column Height (h)	Number of Levels (L)
3	1
4	2
$4 < n \leq 6$	3
$6 < n \leq 9$	4
$9 < n \leq 13$	5
$13 < n \leq 19$	6
$19 < n \leq 28$	7
$28 < n \leq 42$	8
$42 < n \leq 63$	9

Dadda [9] coined the term "parallel (n, m) counter." This counter is a combinatorial network with m outputs and $n (\leq 2^m - 1)$ inputs. The m outputs represent a binary number encoding the number of ones present at the inputs. The carry save adder in the preceding Wallace tree is a (3, 2) counter.

This class of counters has been extended in an excellent article [29] that shows the Wallace tree

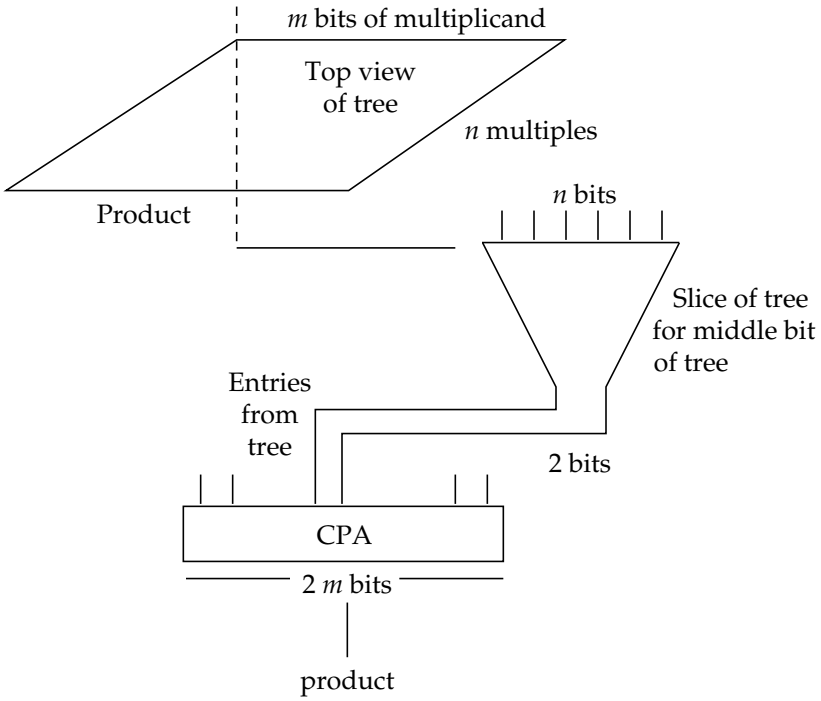


Figure 4.5: Wallace tree.

and the Dadda scheme to be special cases. The generalized counters take several successively weighted input columns and produce their weighted sum. Counters of this type are denoted as:

$$(C_{\text{R-1}},\ C_{\text{R-2}},\ \ldots,\ C_0,\ d)$$

counters, where R is the number of input columns, C_i is the number of input bits in the column of weight 2^i, and d is the number of bits in the output word. The suggested implementation for such counters is a ROM. For example, a (5, 5, 4) counter can be programmed in $1K \times 4$ ROM, where the ten address lines are treated as two adjacent columns of 5 bits each. Note that the maximum sum of the two columns is 15, which requires exactly 4 bits for its binary representation. Figures 4.7 and 4.8 illustrate the ROM implementation of the (5, 5, 4) counter and show several generalized counters. The use of the (5, 5, 4) counter to reduce the partial products in a 12×12 multiplication is shown in Figure 4.9, where the partial products are generated by 4×4 multipliers.

Parallel compressors, which are a subclass of parallel counters, have been introduced by Gajski [11]. These compressors are distinctively characterized by the set of inputs and outputs that serves as an interconnection between different packages in a one-dimensional array of compressors. These compressors are said to be more efficient than parallel counters. For more details on this interesting approach, the reader is referred to Gajski's article [11].

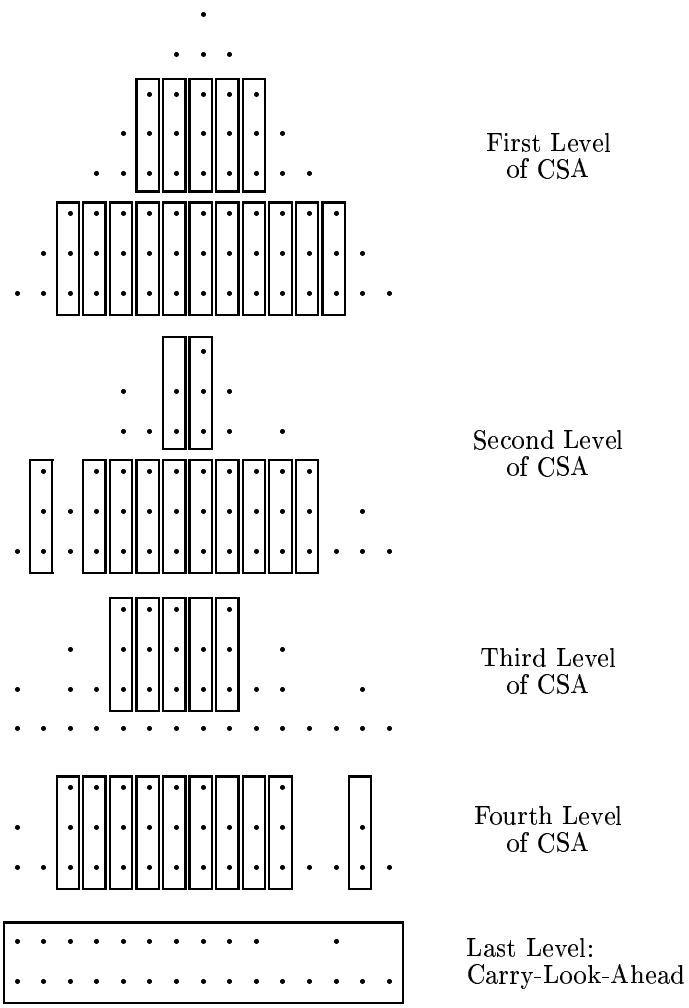
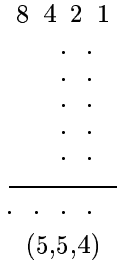


Figure 4.6: Wallace tree reduction of 8×8 multiplication, using carry save adders (CSA).

(a) Adding two columns, each 5 bits in height, gives the maximum result of 15 which is representable by 4 bits (the ROM outputs or counter).



(b) Four (5,5,4)s are used to reduce the five operands (each 8 bits wide) to two operands, which can be added using carry look ahead. Note that, regardless of the operand width, five operands are always reduced to no more than two operands.

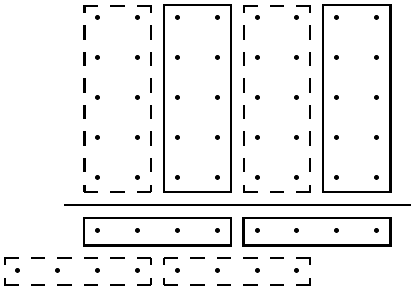


Figure 4.7: The $(5, 5, 4)$ reduces the five input operands to one operand.

4.2 Iteration and Partial Products Reduction

4.2.1 A Tale of Three Trees

The Wallace tree can be coupled with iterative techniques to provide cost effective implementations. A basic requirement for such implementation is a good latch to store intermediate results. In this case "good" means that it does not add additional delay to the computation. As we shall see in more detail later, the Earle latch (Figure 4.10), accomplishes this by simply providing a feed–back path from the output of an existing canonic circuit pair to an additional input.

Thus, an existing path delay is not increased (but fan–in requirements are increased by one). In a given logic path, an existing "and–or" pair is replaced by the latch which also performs the "and–or" function.

Use of this latch can significantly reduce implementation costs if appropriate care is taken in design (see Chapter 6).

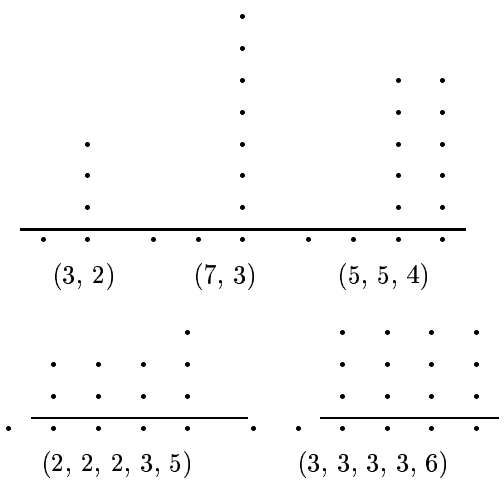


Figure 4.8: Some generalized counters from Stenzel [29].

Consider the operation of an n–bit multiplier on an m–bit multiplicand; that is, we reduce n partial products, each of m bits, to two partial products, then propagate the carries to form a single (product) result. Trees of size L actually have some bits of their partial product tree for which L CSA stages are required. Note that for both low–order and high–order product bits the tree size is less.

Now in the case of the simple Wallace tree the time required for multiplication is:

$$\tau \leq L \cdot 2 + \text{CPA}(2m \text{ bits}),$$

where τ is in unit gate delays. Each CSA stage has 2 serial gates (an AND–OR in both sum and carry). The CPA $(2m)$ term represents the number of unit gate delays for a carry look ahead structure with operand size $2m$ bits. Actually since the tree height at the less significant bits is smaller than the middle bits, entry into the CPA from these positions arrives early. Thus, the CPA term is somewhat conservative.

The full Wallace tree is "expensive" and, perhaps more important, is topologically difficult to implement. That is, large trees are difficult to map onto planes (printed wire boards) since each CSA communicates with its own slice, transmits carries to the higher order slice, and receives carries from a lower order. This "solid" topology creates both I/0 pin difficulty (if the implementation "spills" over a single board) and wire length (routing) problems.

Iterating on smaller trees has been proposed [AND 67] as a solution. Instead of entering n multiples of the multiplicand we use an n/I tree and perform I iterations on this smaller tree.

Consider three types of tree iterations:

1. Simple iteration: in this case, the n/I multiples of the m–bit multiplicand are reduced and added to form a single partial product. This is latched and fed back into the top of the tree for assimilation on the next iteration.

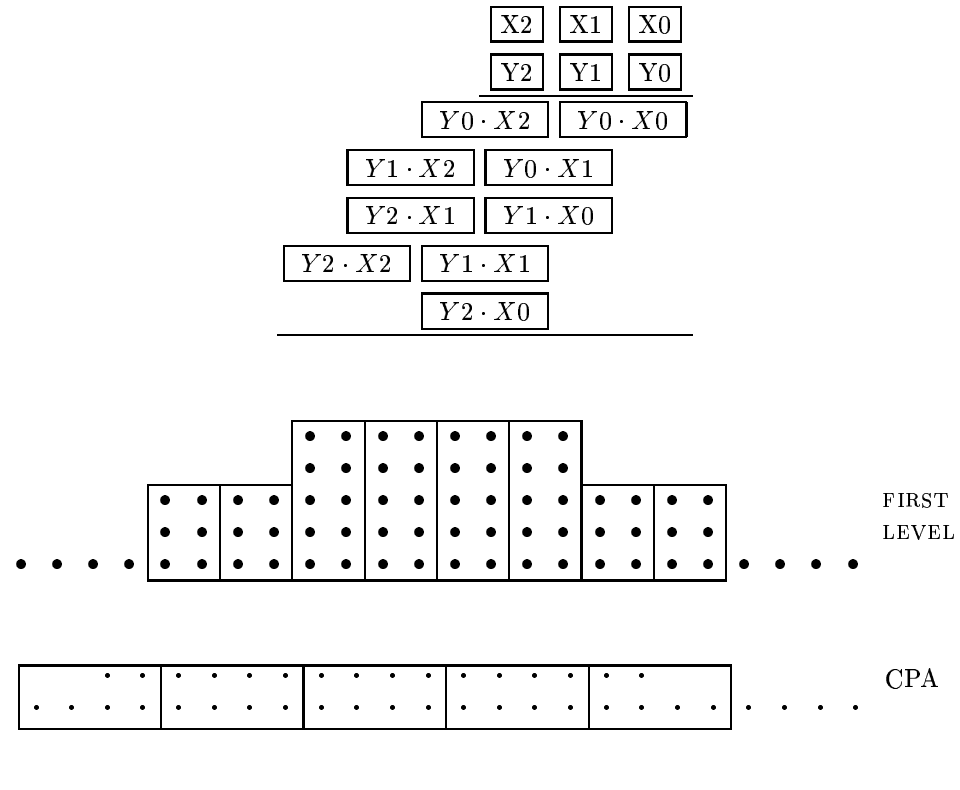


Figure 4.9: 12×12 bit partial reduction using $(5, 5, 4)$ counters. The $X0$, $Y0$, $X1$, etc., terms each represent four bits of the argument. Thus, the product $X0Y0$ is an 8-bit summand.
(a) Partial products are generated by 4×4 multipliers [29].
(b) Eight $(5, 5, 4)$s are used to compress the column height from five to two.
(c) A CPA adder is used to add the last two operands. The output of each counter in (b) produces a 4-bit result: 2 bits of the same significance, and 2 of higher. These are combined in (c).

The multiplication time is now the number of iterations, I, times the sum of the delay in the CSA tree height (two delays per CSA) and the CPA delay. The CSA tree is approximately $\log_{3/2}$ of the number of inputs $\lceil \frac{n}{\mathrm{I}} + 1 \rceil$ divided by 2, since the tree is reduced to *two* outputs, not *one*. The maximum size of the operands entering the CPA on any iteration is $m + \lceil \frac{n}{\mathrm{I}} + 1 \rceil$;

$$\tau \approx \mathrm{I}\left(2\left\lceil \log_{3/2}\left\lceil \frac{n}{\mathrm{I}} + 1 \right\rceil / 2 \right\rceil + \mathrm{CPA}\left(m + \left\lceil \frac{n}{\mathrm{I}} + 1 \right\rceil\right)\right)$$

unit gate delays.

2. Iterate on tree only: the above scheme can be improved by avoiding the use of the CPA until the partial products are reduced to two. This requires that the (shifted) two partial results be fed back into the tree *before* entering the CPA, (Figure 4.12). The "shifting" is required since the new $\frac{n}{\mathrm{I}}$ input multiples are at least (could be more, depending on

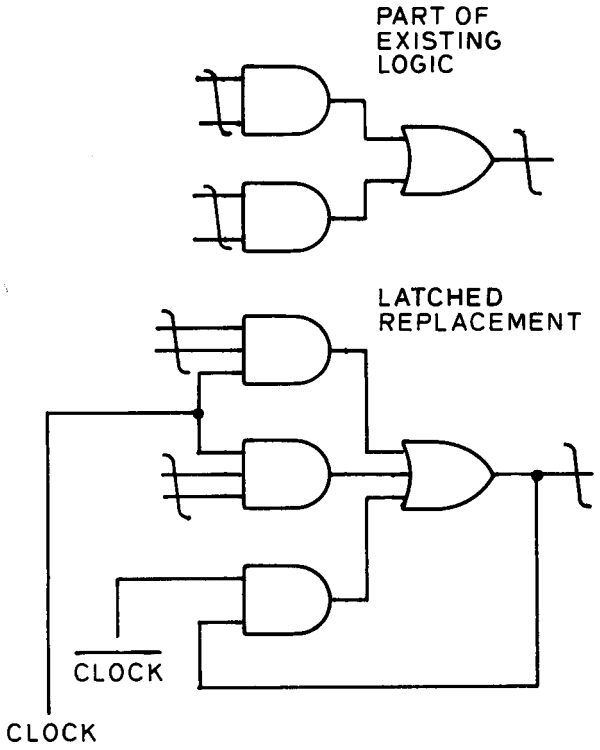


Figure 4.10: Earle latch.

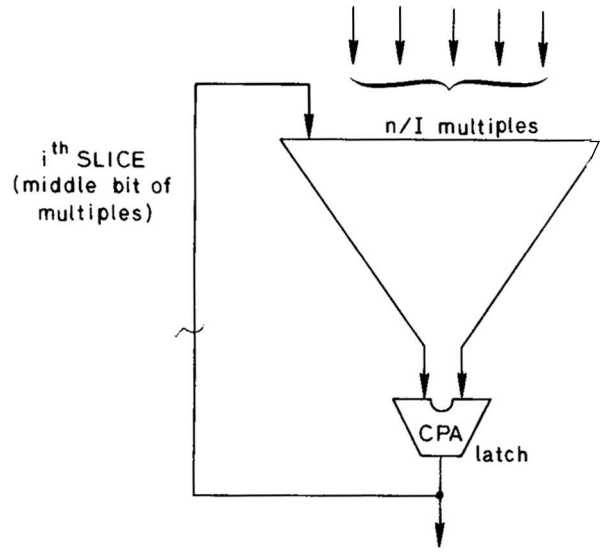


Figure 4.11: Slice of a simple iteration tree showing one product bit.

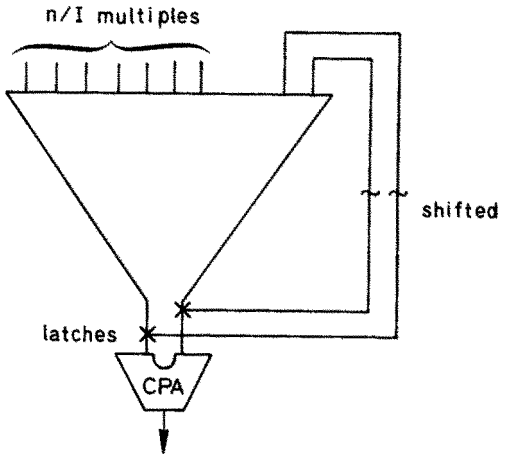


Figure 4.12: Slice of tree iteration showing one product bit.

multiplier encoding) $\frac{n}{\mathrm{I}}$ bits more significant than the previous iteration. Thus, each pair of reduced results is returned to the top of the tree and shifted to the correct significance. Therefore, we require only one CPA and I iterations on the CSA tree.

The time for multiplication is now:

$$\tau \approx I\left(2\left\lceil \log_{3/2}\left\lceil \frac{n}{\mathrm{I}}+2\right\rceil /2\right\rceil\right)+\mathrm{CPA}\,(2m)$$

3. Iterate on lowest level of tree: In this case the partial products are assimilated by returning them to the lowest level of the tree. When they are assimilated to two partial products again a single CPA is used, (Figure 4.13). Thus:

   As the Earle latch requires (approximately) a minimum of four gate delays for a pipeline stage, returning the CSA output one level back into the tree provides an optimum implementation (Chapter 6 provides more detailed discussion). The tree height is increased; but only the initial set of multiples sees the delay of the total tree. Subsequent sets are introduced at intervals of four gate delays, (Figure 4.13). Thus, the time for multiplication is now:

$$\tau \approx 2\left(\left\lceil \log_{\frac{3}{2}}\left\lceil \frac{n}{\mathrm{I}}\right\rceil /2\right\rceil\right)+\mathrm{I}\cdot 4+\mathrm{CPA}(2m)$$

Note that while the cost of the tree has been significantly reduced, only the $\mathrm{I}\cdot 4$ term differs

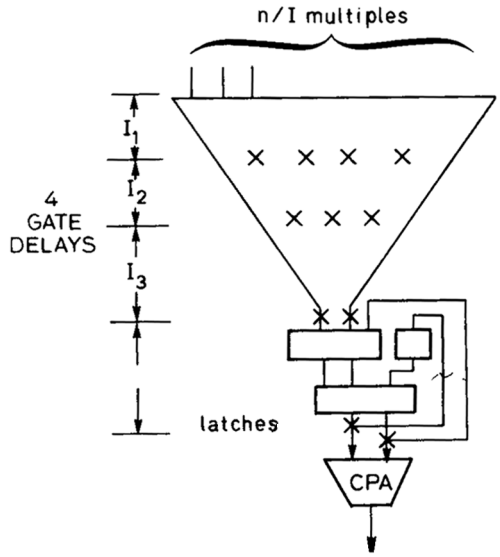


Figure 4.13: Slice of low level tree iteration.

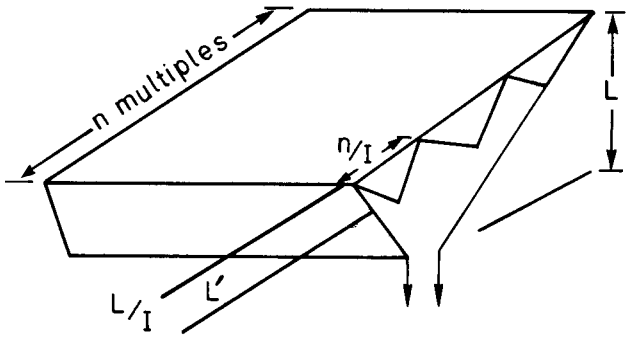


Figure 4.14: Iteration.

from the full Wallace tree time. So long as I is chosen so that this term does not dominate the total time, attractive implementations are possible using a tree of size L′ instead of L levels.

In all of these approaches, we are reducing the height of the tree by inputing significantly fewer terms, about $\frac{n}{\mathrm{I}}$ instead of n. These fewer input terms mean a much smaller tree—fewer components (cost) and quicker generation of a partial result, but now I iterations are needed for a complete result instead of a single pass.

4.3 Iterative Array of Cells

The matrix generation/reduction scheme, (Wallace tree), is the fastest way to perform parallel multiplication; however, it also requires the most hardware. The iterative array of cells requires less hardware but it is slower. The hardware required in the iterative approach can be calculated from the following formula:

$$\text{number of building blocks} = \left\lceil \frac{\mathrm{N} \times \mathrm{M}}{n \times m} \right\rceil$$

where N, M are the number of bits in the final multiplication, and n, m are the number of bits in each building block. For example, nine 4×4 multipliers are required to perform 12×12 multiplication in the iterative approach (since $(12 \times 12)/(4 \times 4) = 9$). By contrast, using the matrix generation technique to do 12×12 multiplication requires 13 adders in addition to the nine 4×4 multipliers (see Figure 4.9). In general, the iterative array of cells is more attractive for shorter operand lengths since their delay increases linearly with operand length, whereas the delay of the matrix–generation approach increases with the log of the operand length.

The simplest way to construct an iterative array of cells is to use 1–bit cells, which are simply full adders. Figure 4.15 depicts the construction of a 5×5 unsigned multiplication from such cells.

In the above equation, we define the arithmetic value (weight) of a logical zero state as μ and a logical one state as ν; then $P(\mu, \nu)$ is a variable with states μ and ν. Thus, a conventional

carry-save adder with unsigned inputs performs $X_0+Y_0+C_0 = C_1+S_0$, or $(0,1)+(0,1)+(0,1) = (0,2)+(0,1)$.

In two's complement, the most significant bit of the arguments is the sign bit, thus for 5×5 bit two's complement multiplication of $Y_4Y_3Y_2Y_1Y_0$ by $X_4X_3X_2X_1X_0$, Y_4 and X_4 indicate the sign. Thus, the terms

$$X_0Y_4, X_1Y_4, \ldots, X_3Y_4$$

and

$$X_4Y_3, X_44Y_2, X_4Y_1, X_4Y_0$$

have range $(0,-1)$.

$$X_0Y_4 + 0 + X_1Y_3 \text{ is } (0,-1) + 0 + (0,1) =$$

This can be implemented by type II′: $(0,1) + (0,-1) + (0,-1) = (0,-2) + (0,1)$

The negative carry out and the negative X_iY_4 requires type II′ circuits along the right diagonal, until negative input X_4Y_3 combines with X_3Y_4 defining a type I′ situation wherein all inputs are zero or negative.

Now the type II cell has equations:

$$C_1 = A_0B_0C_0 + \bar{A}_0B_0C_0 + \bar{A}\bar{B}_0C_0 + \bar{A}B_0\bar{C}_0$$

$$S_0 = A_0B_0C_0 + \bar{A}_0\bar{B}_0C_0 + \bar{A}_0B_0\bar{C}_0 + A_0\bar{B}_0\bar{C}_0$$

(recall that $S_0 = (0,-1)$

Figure 4.15 can be generalized by creating a 2 bit adder cell (Figure 4.18) akin to the 1-bit cell of Figure 4.16. For unsigned numbers, an array of circuits of the type called L101:

$$A_1 + B_1 + A_0 + B_0 + C_0 = C_2 + S_1 + S_0$$

$$(0,2) + (0,2) + (0,1) + (0,1) + (0,1) = (0,4) + (0,2) + (0,1)$$

is required.

In (Figure 4.15), if we let
$A_1 = X_0Y_2$
$B_1 = X_1Y_1$
$A_0 = X_0Y_1$
$B_0 = X_1Y_0$
$C_0 = 0$
we capture the upper rightmost 2-bit positions with one 2-bit cell (Figure 4-16b). Continuing for the 5×5 unsigned multiplication we need one half the number of cells (plus 1) as shown in Figure 4.15. We can again extend this to two's complement arithmetic.

To perform signed (two's complement) multiplication the scheme is slightly more complex, as described in [PEZ 70]. Pezaris illustrates his scheme by building 5×5 multipliers from two types of circuits, using the following 1–bit adder cell (Figure 4.16.):

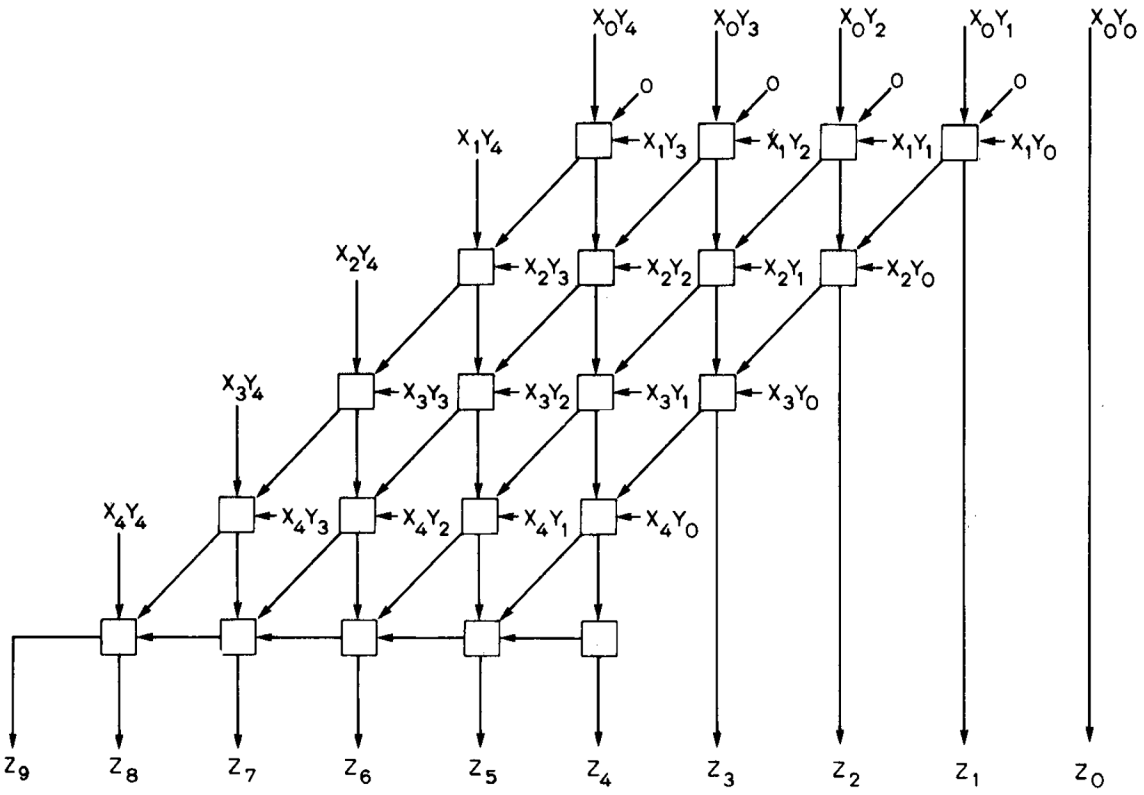


Figure 4.15: 5×5 unsigned multiplication.

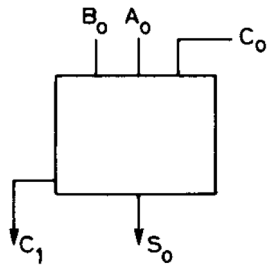


Figure 4.16: 1–bit adder cell.

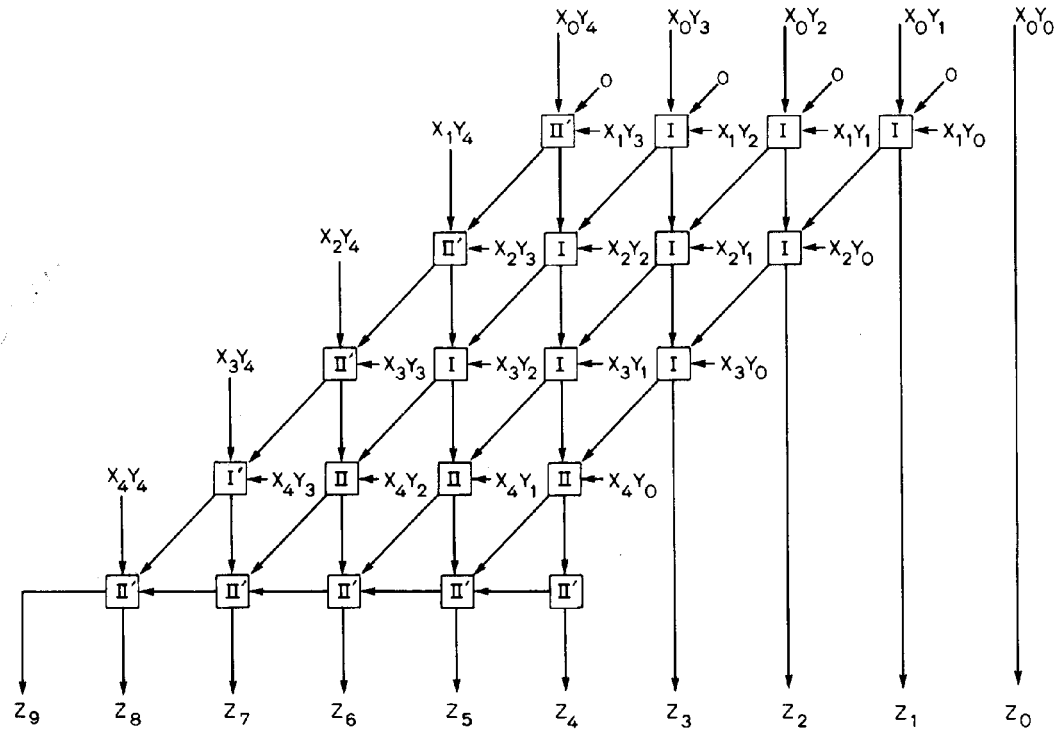


Figure 4.17: 5×5 two's complement multiplication [PEZ 70].

	A_0	B_0	C_0	C_1	S_0
TYPE	I:$(0,1)$	$+(0,1)$	$+(0,1)$	$=(0,2)$	$+(0,1)$
TYPE	II:$(0,-1)$	$+(0,1)$	$+(0,1)$	$=(0,2)$	$+(0,-1)$
TYPE	II′:$(0,1)$	$+(0,-1)$	$+(0,-1)$	$=(0,-2)$	$+(0,1)$
TYPE	I′:$(0,-1)$	$+(0,-1)$	$+(0,-1)$	$=(0,-2)$	$+(0,-1)$

Type I is the conventional carry save adder, and it is the only type used in Figure 4.15 for the unsigned multiplication. Types I and I′ correspond to identical truth tables (because if $x+y+z = u+v$, then $-x-y-z = -u-v$) and, therefore, to identical circuits. Similarly, types II and II′ correspond to identical circuits. Figure 4.17 shows the entire 5×5 multiplication.

Pezaris extends the 1–bit adder cell to a 2–bit adder cell, as shown below:

Implementation of this method with 2–bit adders requires *three* types of circuits, (the L101, L102, L103). The arithmetic operations performed by these three types are given below:

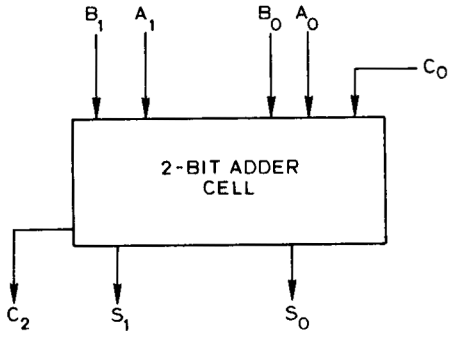


Figure 4.18: 2–bit adder cell.

	A_1	B_1	A_0	B_0	C_0	C_2	S_1	S_0
L101 :	$(0,2)+$	$(0,2)$	$+(0,1)+$	$(0,1)$	$+(0,1)=$	$(0,4)+$	$(0,2)$	$+(0,1)$
L102 :	$(0,2)+$	$(0,-2)+$	$(0,1)+$	$(0,1)$	$+(0,1)=$	$(0,4)+$	$(0,-2)+$	$(0,1)$
L103 :	$(0,2)+$	$(0,-2)+$	$(0,1)+$	$(0,-1)+$	$(0,1)=$	$(0,4)+$	$(0,-2)+$	$(0,1)$

Iterative multipliers perform the operation

$$S = X \cdot Y + K,$$

where K is a constant to be added to the product (whereas the matrix generation schemes perform only $S = X \cdot Y$). The device uses the (3–bit) modified Booth's algorithm to halve the number f partial products generated. Figure 4.19 shows the block diagram of the iterative cell. The X_{-1} input is needed in expanding horizontally since the Booth encoder may call for 2X, which is implemented by a left shift. The Y_{-1} is used as the overlap bit during multiplier encoding. Note that outputs S_4 and S_5 are needed only on the most significant portion of each partial product (these 2 bits are used for sign correction). Figure 4.20 shows the implementation of a 12×12 two's complement multiplier. This scheme can be extended to larger cells. For example, in Figure 4.20, the dotted line encloses an 8×8 iterative cell.

4.4 Detailed Design of Large Multipliers

4.4.1 Design Details of a 64×64 Multiplier

In this section, we describe the design of a 64×64 multiplier using the technique of simultaneous generation of partial products. The design uses standard design macros. Four types of macros are needed to implement the three steps of a parallel multiplier.

1. Simultaneous generation of partial products—using an 8×8 multiplier macro.
2. Reduction of the partial products to two operands—(5,5,4) counter.
3. Adding the two operands—adders with carry look ahead.

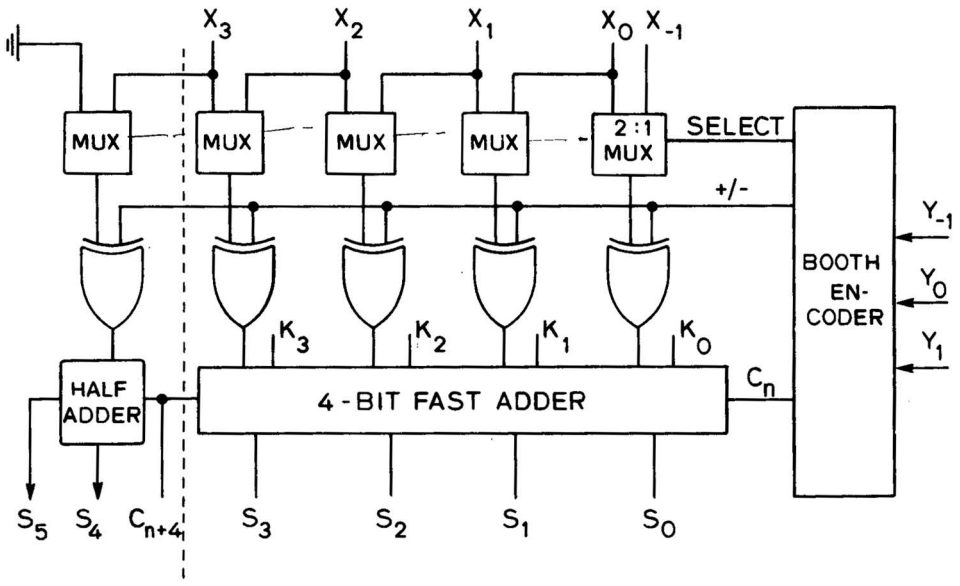


Figure 4.19: Block diagram of 2×4 iterative multiplier.

Figure 4.21 depicts the generation of the partial products in a 64×64 multiplication, using 8×8 multipliers. Each of the two 64-bit operands is made of 8 bytes (byte = 8 bits), which are numbered 0, 1, 2, ... 7 from the least to the most significant byte. Thus, 64-bit multiplication involves multiplying each byte of the multiplicand (X) by all 8 bytes of the multiplier (Y). For example, in Figure 4.22, the eight rectangles marked with a dot are those generated from multiplying byte 0 of Y by each of the 8 bytes of X. Note that the product "01" is shifted 8 bits with respect to product "00," as is "02" with respect to "01," and so on. These 8-bit shifts are due to the shifted position of each byte within the 64-bit operand. Also, note that for each $X_i \cdot Y_j (i \neq j)$ byte multiplication, there is a corresponding product $X_j Y_i$ with the same weight. Thus, product "01" corresponds to product "10" and product "12" corresponds to product "21." As before, for $N \times M$ multiplication, the number of $n \times m$ multipliers required is:

$$\frac{N \times M}{n \times m,}$$

and in our specific case:

$$\text{Number of multipliers } = \frac{64 \times 64}{8 \times 8} = 64 \text{ multipliers.}$$

The next step is to reduce the partial products to two operands. As shown in Figure 4.21, a $(5, 5, 4)$ can reduce two columns, each 5 bits high, to two operands. The matrix of the partial products in Figure 4.22 can be viewed as eight groups of 16 columns each. The height of the eight groups is equal to 1, 3, 5, 7, 9, 11, 13 and 15. Figure 4.23 illustrates with a dot representation the use of (5,5,4)s to reduce the various column heights to 2 bits high. We now can compute the total number of counters required to reduce the partial product matrix of a 64×64 multiplication to two operands:

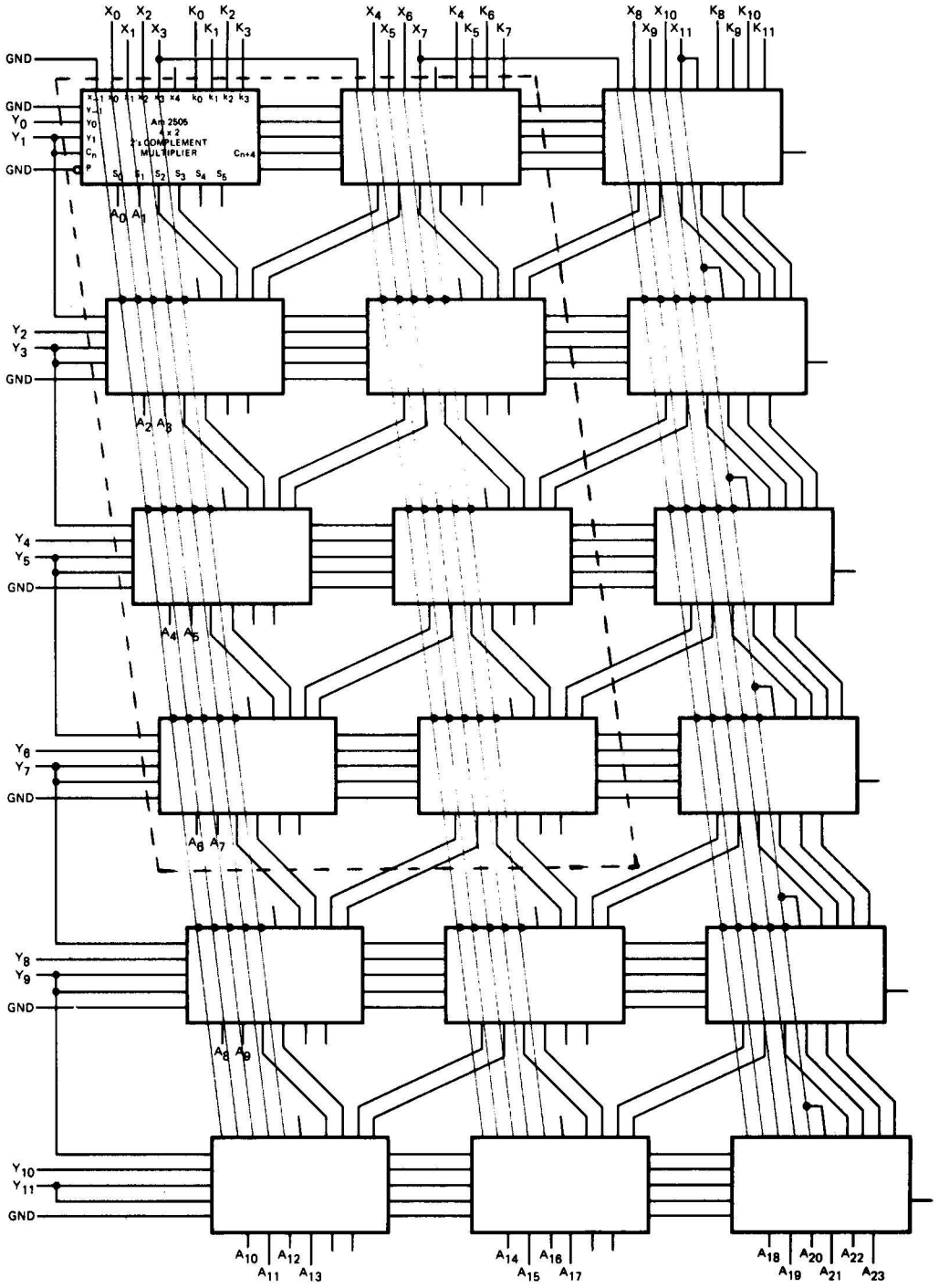


Figure 4.20: 12×12 two's complement multiplication $A = X \cdot Y + K$. Adapted from [GHE 71].

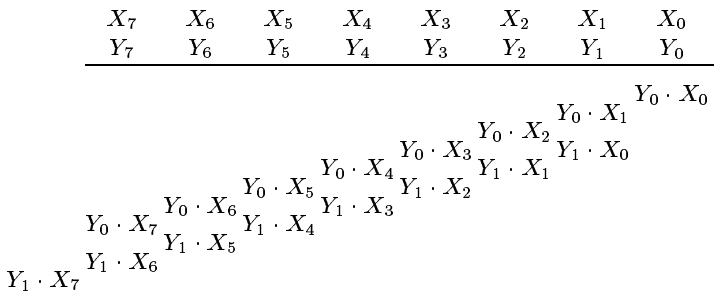


Figure 4.21: A 64×64 multiplier using 8×8 multipliers. Only 16 of the 64 partial products are shown. Each 8×8 multiplier produces a 16–bit result.

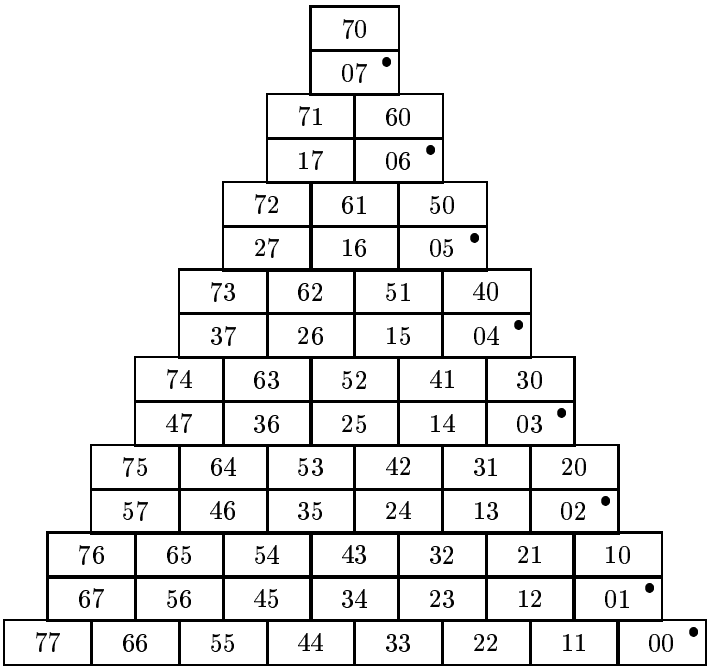


Figure 4.22: Partial products generation of 64×64 multiplication using 8×8 multipliers. Each rectangle represents the 16-bit product of each 8×8 multiplier. These partial products are later reduced to two operands, which are then added by a CPA adder. Each box entry above corresponds to a partial product index pair; e.g., the 70 corresponds to the term $Y7 \cdot X0$.

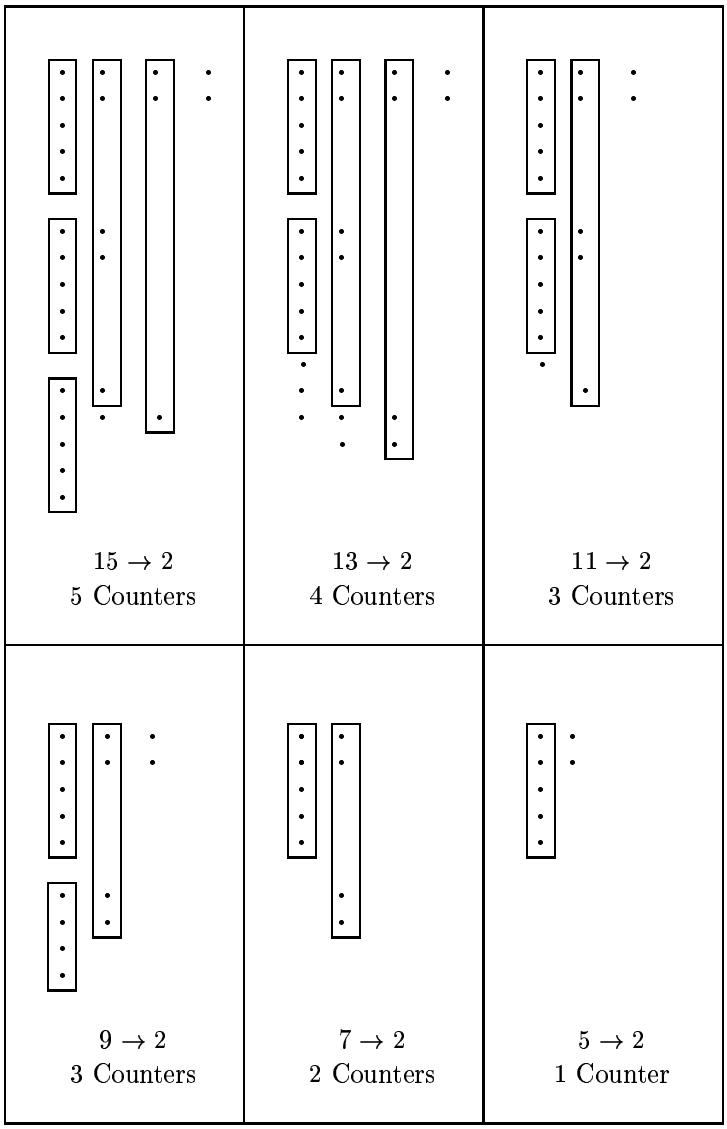


Figure 4.23: Using (5,5,4)s to reduce various column heights to 2 bits high. The 15 → 2 shows the summation as required in Figure 4.21 (height 15). Each other height required in Figure 4.22 is shown in 5,5,4 implementation.

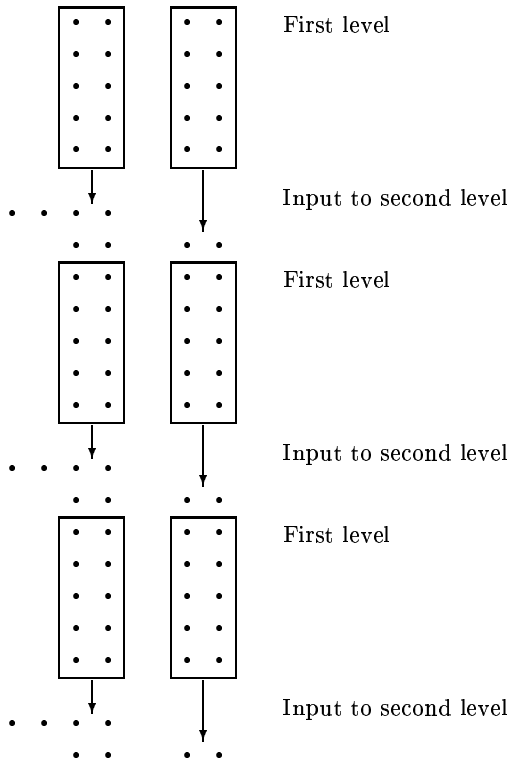


Figure 4.24: Using (5,5,4) counters to implement the reduction of the partial products of height 15, showing only the first level and its output.

This maximum height occurs when partial products 70 ($Y7 \cdot X0$) through 44 (or 33) are to be summed (Figure 4.22). For example, the top 5,5,4 counter above would have inputs from 70, 07, 71, 17, 61, the middle counter inputs from 16, 62, 26, 52, 25, and the lowest from 53, 35, 43, 34, and 44. The three counters above provide three 4-bit outputs (2 bits of the same significance, 2 of higher). Thus, six outputs must be summed in the second level: three from the three shown counters, and three from the three lower order counters.

Number of Columns	Height of Columns	Number of Counters per Two Columns	Number of Counters for all Columns of Same Height
16	15	5	$8 \times 5 = 40$
16	13	4	$8 \times 4 = 32$
16	11	3	$8 \times 3 = 24$
16	9	3	$8 \times 3 = 24$
16	7	2	$8 \times 2 = 16$
16	5	1	$8 \times 1 = 8$
16	3	1	$8 \times 1 = 8$
16	1	—	—
Total number of counters			152

The last step in the 64×64 multiplication is to add the two operands using 4-bit adders and carry look ahead units, as described earlier. Since the double length product is 128 bits long, a 128-bit carry look ahead (CLA) adder is used.

4.4.2 Design Details of a 56×56 Single Length Multiplier

The last section described a 64×64 multiplier with double length precision. In this section, we illustrate the implementation of single length multiplication and the hardware reduction associated with it, as contrasted with double length multiplication. We select 56×56 multiplication because 56 bits is a commonly used mantissa length in long floating point formats.

A single length multiplier is used in fractional multiplication, where the precision of the product is equal to that of each of the operands. For example, in implementing the mantissa multiplier in a hardware floating point processor, input operands and the output product have the range and the precision of:

$$0.5 \leq \text{mantissa range} \leq 1 - 2^{-n},$$

$$\text{mantissa precision} = 2^{-n},$$

where n is the number of bits in each operand.

Figure 4.25 shows the partial products generated by using 8×8 multipliers. The double length product is made of 49 multipliers, since

$$\text{Number of multipliers } = \frac{56 \times 56}{8 \times 8} = 49.$$

However, for single precision, a substantial saving in the number of multipliers can be realized by "chopping" away all the multipliers to the right of the 64 MSBs (shaded area); but we need to make sure that this chopping does not affect the precision of the 56 MSBs. Assume a worst case contribution from the "chopped" area; that is, all ones. The MSB of the discarded part has the weight of 2^{-65}, and the column height at this bit position is 11. Thus, the first few terms of the "chopped" area are:

$$\begin{aligned} \text{Max error} &= 11 * (2^{-65} + 2^{-66} + 2^{-67} + \ldots) \\ \text{But:} &\quad 11 < 2^4 \\ \text{Therefore:} &\quad \text{Max error} < (2^{-61} + 2^{-62} + 2^{-63} + \ldots) \end{aligned}$$

From the last equation, it is obvious that "chopping" right of the 64 MSBs will give us 60 correct MSBs, which is more than enough to handle the required 56 bits plus the 2 guard bits.

Now we can compute the hardware savings of single length multiplication. From Figure 4.25, we count 15 fully shaded multipliers. We cannot remove the half shaded multipliers, since their most significant half is needed for the 58-bit precision. Thus, a total of 34 instead of 49 multipliers is used for the partial product generation.

Using the technique outlined previously, the number of counters needed for column reduction is easily computed for double and single length.

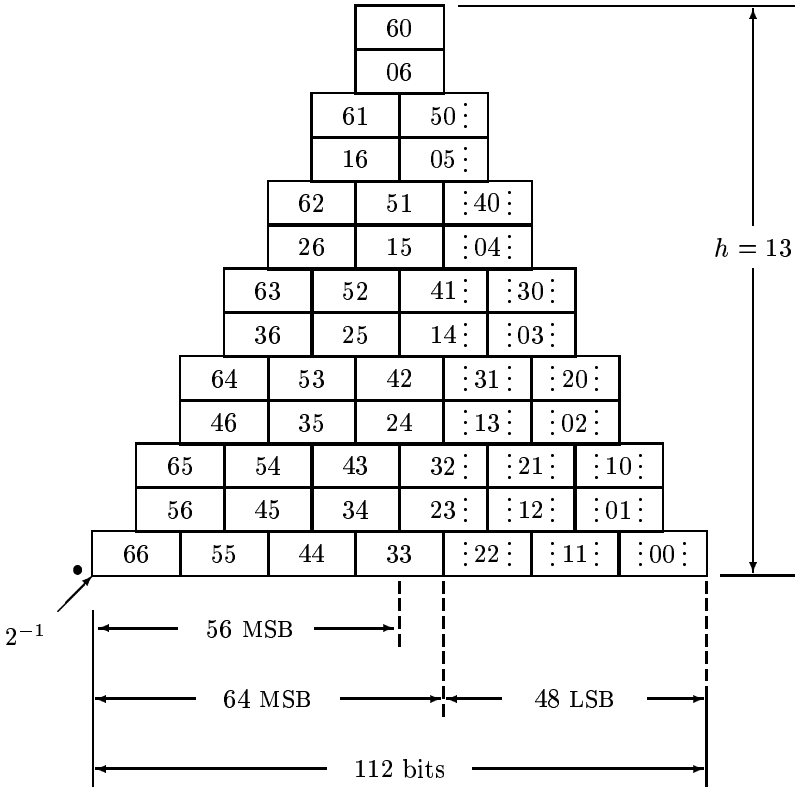


Figure 4.25: Partial products generation in a 56×56 multiplication, using 8×8 multipliers. The shaded multipliers can be removed when a single length product (56 MSBs) is required.

Number of Columns		Height of Columns	Number of Counters per Two Columns	Number of Counters for All Columns of Same Height	
Double Length	Single Length			Double Length	Single Length
16	16	13	4	32	32
16	8	11	3	24	12
16	8	9	3	24	12
16	8	7	2	16	8
16	8	5	1	8	4
16	8	3	1	8	4
16	8	1	—	—	—
Total Number of Counters				112	72

Finally, adding the resulting two operands in the double length case (112 bits) requires a carry look ahead (CLA) adder. For the single length case, the addition of the two 64 bit operands is accomplished by a 64-bit CLA adder. There is a 34% hardware savings using the single length multiplication. However, there is no speed improvement to a first approximation.

4.5 Exercises

1. Design a modified Booth's encoder for a 4-bit multiplier.
2. Find the delay of the encoder in problem 1.
3. Construct an action table for modified Booth's algorithm (2-bit multiplier encoded) for sign and magnitude numbers (be careful about the sign bit).
4. Design a modified Booth's encoder for sign and magnitude numbers (4-bit multiplier encoded).
5. Construct the middle bit section of the CSA tree for 48×48 multiplication for:
   (a) Simple iteration.
   (b) Iterate on tree.
   (c) Iterate on lowest level of tree.
6. Compute the number of CSA's required for:
   (a) 1-bit section.
   (b) Total tree.
7. Derive an improved delay formula (taking into account the skewed significance of the partial products) for:
   (a) Full Wallace tree.
   (b) Simple iteration.
   (c) Iterate on tree.
   (d) Iterate on lowest level of tree.
8. Suppose 256×8 ROMs are used to implement 12 bit $\times$ 12 bit multiplication. Find the partial products, the rearranged product matrix, and the delay required to form a 24 bit product.
9. If (5, 5, 4) counters are used for partial product reduction, compute the number of counter levels required for column heights of 8, 16, 32, and 64 bits. How many such counters are required for each height?
10. Suppose (5, 5, 4) counters are being used to implement an iterative multiplication with a column height of 32 bits. Iteration on the tree only is to be used. Compare the tree delay and number of counters required for one, two, and four iterations.
11. Refer to Figure 4.2. Develop the logic on the A_9, B_9, C_9, and D_9 terms to eliminate the required summing of these terms to form the required product. Assume X and Y are 8-bit unsigned integers.
12. Refer to Table 4.1. Show that this table is valid for the two's complement form of the multiplier.

13. Implement a $(5, 5, 4)$ counter using only CSAs. Use a minimum number of levels and then a minimum number of CSAs.

    (a) Show in dot representation each CSA in each level.

    (b) Your implementation has how many gate delays?

14. We wish to build a 16×16 bit multiplier. Simple AND gates are used to generate the partial products ($t_{\max} = 20$ ps, $t_{\min} = 10$ ps).

    (a) Determine the height of the partial product tree.

    (b) How many AND gates are required?

    (c) How many levels of (5,5,4) counters are required? Estimate the total number of counters.

    (d) If the counters have an access time of $\mathrm{P}_{\max} = 30$ ps, $\mathrm{P}_{\min} = 22$ ps, determine the latency of the multiplier (before CPA).

    (e) An alternative design for the multiplier would use iteration of the partial product tree. If only one level of (5,5,4) counters is used, how many iterations are required?

15. In the above problem how many gates are needed for PP generation and how many ROM's for PP reduction?

16. Implement a $(7, 3)$ counter using only CSAs. Use a minimum number of levels and then a minimum number of CSAs.

    (a) Show in dot representation each CSA in each level.

    (b) Repeat for a (5,5,4) counter.

17. We want to perform 64×64 bit multiplicand using 8×8 PP generators. What is the PP tree height?

    Suppose the tree reduction is to be done by (7,3) counters. Assume a three input CPA is available. Show the counter reduction levels for the max column height (h). How many (7,3) counter levels are required?

    Suppose we now wish to iterate on the tree using one level of (7,3) counters.

    Suppose PP generation takes $P_{\max} = 400$ ps, (7,3) counter takes $P_{\max} = 200$ ps, and three input CPA takes $P_{\max} = 300$ ns.

    How would you arrange to assimilate (reduce) the PP's and find the product? Show the number of PP reduced for each iteration.

    The total time for multiplication is how many nsec?

18. Design a circuit to compute the sum of the number of ones in a 128^{b} word, using only $64^{\mathrm{K}} \times 8^{\mathrm{b}}$ ROM's (no CPA's). You should minimize the number of levels, then the number of parts, then the number of types of ROM configurations (contents). Use dot notation.

19. A method to perform a DRC multiply (1's complement) has been proposed. It uses an unsigned 2's complement multiplier which takes two n-bit operands and produces a $2n$-bit result, and a CLA adder which adds the high n-bits to the low n-bits of the multiplier

output. Don't worry about checking for overflow. Assume the DRC result will fit in n bits.

Either prove that this will always produce the correct DRC result, or give an example where it will fail.