Chapter

4

Low-Voltage Low-Power Multipliers

4.1 Introduction

Multiplication is an important fundamental function in arithmetic operations. In fact, multiplication-based operations such as Multiply and Accumulate (MAC) and inner product are among some of the frequently used computation-intensive arithmetic functions currently implemented in many Digital Signal-Processing (DSP) applications (such as convolution, Fast Fourier Transform (FFT), filtering, and others). They usually contribute significantly to the time delay and take up a great deal of silicon area in the DSP system. Since multiplication dominates the execution time of most DSP algorithms, using a high-speed multiplier is very desirable. Currently, multiplication time is still the dominant factor in determining the instruction cycle time of a DSP chip.

With an ever-increasing quest for greater computing power on battery-operated mobile devices, design emphasis has shifted from optimizing conventional delay time and area size to minimizing power dissipation while still maintaining high performance. Compared with previous designs, a low-power design allows portable devices to operate longer with the same amount of battery charge. The subsequent sections present an overview of the multiplication operation and of different types of parallel multipliers.

4.2 Overview of Multiplication

Multiplication can be considered as a series of repeated additions. The number to be added is called the multiplicand, the number of times it is added is called the multiplier, and the result obtained is called

the product. The basic operations involved in multiplication include generating and accumulating or adding the partial products. Consequently, to speed up the entire multiplication process, these two major steps must be optimized. The two main categories of binary arithmetic multiplication involve computing unsigned numbers and computing signed numbers.

4.2.1 Unsigned multiplication

Real-time computer applications require fast multiplication. By utilizing AND gates and full adders, multiplication can be implemented on the processor much in the same way as it is done by hand: multiply each digit of the multiplier by the multiplicand, thereby generating partial products and then sum up the respective partial products in order to generate the final result. Assume that X and Y are two n-bit unsigned numbers, where X is the multiplicand and Y is the multiplier. They can be expressed as follows [1]:

$$X = \sum_{i=0}^{n-1} X_i 2^i \tag{4.1}$$

$$Y = \sum_{j=0}^{n-1} Y_j 2^j \tag{4.2}$$

The product of X and Y is P and it can be written in the following form:

$$P = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} X_i Y_j 2^{(i+j)} \tag{4.3}$$

To illustrate further, the multiplicand X and multiplier Y can be represented as follows:

X	Multiplicand	$x_{n-1}x_{n-2}x_{n-3}\cdots x_1x_0$
Y	Multiplier	$y_{n-1}y_{n-2}y_{n-3}\cdots y_1y_0$
P	Product (X *Y)	$p_{2n-1}p_{2n-2}p_{2n-3}\cdots p_1p_0$

Each of the partial product terms $P_n = X_i Y_j$ is called the summand. Table 4.1 shows the process of multiplying two unsigned Binary-Coded Decimals (BCDs) using the paper-and-pencil method [1, 2]. Each partial product then gets stored in the Arithmetic Logic Unit register, where it occupies memory space until the final partial product is obtained. All the partial products then get added up to generate the final product. The partial storing of each partial product and the subsequent addition process involved make this method terribly inefficient.

TABLE 4.1 General Multiplication Algorithm

				X_3	X_2	X_1	X_0	$=X$ (multiplicand)
				Y_3	Y_2	Y_1	Y_0	$=Y$ (multiplier)
				X_3Y_0	X_2Y_0	X_1Y_0	X_0Y_0	Partial Product 0
			X_3Y_1	X_2Y_1	X_1Y_1	X_0Y_1		Partial Product 1
		X_3Y_2	X_2Y_2	X_1Y_2	X_0Y_2			Partial Product 2
+	X_3Y_3	X_2Y_3	X_1Y_3	X_0Y_3				Partial Product 3
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	$P = X * Y$

4.2.2 Shift/add multiplication algorithms

Sequential or 1-bit-at-a-time multiplication can be realized by initializing the cumulative partial product to "0" and successively adding to it the properly shifted terms of the multiplicand. Since each successive number to be added to the cumulative partial product is shifted by one bit with respect to the preceding one, there is a simpler approach, which is to shift the cumulative partial product by one bit in order to align its bits with those of the next partial product. Two versions of this algorithm can be devised, depending on whether the partial product terms in Table 4.1 are processed from top to bottom or from bottom to top.

When multiplying with right shifts, the partial product terms accumulate from top to bottom [3].

$$p^{(j+1)} = (P^{(j)} + Y_j X 2^n)2^{-1} \quad \text{with} \quad p^{(0)} = 0,\ p^{(n)} = p \qquad (4.4)$$

$$|-\text{add}-|$$

$$|-\text{shift right}-|$$

Because the right shifts cause the first partial product to be multiplied by 2^{-n}, the multiplicand should be premultiplied by 2^n in order to offset the effect of the right shifts. The premultiplication simply aligns the multiplicand with the upper half of the $2n$-bit cumulative partial product in the addition steps. On the other hand, multiplication with left shifts adds the partial product terms from bottom to top.

$$p^{(j+1)} = 2P^{(j)} + Y_{n-j-1}X \quad \text{with} \quad p^{(0)} = 0,\ p^{(n)} = p \qquad (4.5)$$

$$\downarrow$$

shift left

$$|-\text{add}-|$$

4.2.3 Multiplication of signed numbers

The above procedure for multiplication works well for unsigned integers or unsigned fixed-point numbers. Generally, for the multiplication of signed numbers, the negative number is first converted to its 2's complement representation, thus making sure that all the partial products are positive.

Consider the multiplication of an n-bit X and $-Y$. Converting the negative number of Y to its 2's complement format results in $(2^n - Y)$. The product in the 2's complement context, based on direct multiplication, is

$$P' = X(-Y) = X(2^n - Y) = 2^n X - XY \tag{4.6}$$

This product differs from the expected result

$$P = -XY = 2^{2n} - XY \tag{4.7}$$

$(2^{2n} - XY)$ is the 2's complement representation of $-XY$. Therefore, P' deviates from the actual result P by

$$P - P' = 2^{2n} - 2^n X = 2^n(2^n - X) \tag{4.8}$$

where $2^n(2^n - X)$ is the correction factor to be added. This is done by taking the 2's complement of X and then shifting it to the left by n positions. This corrective factor is then added to the precomputed product of P'.

If both the multiplicand and multiplier are negative, their 2's complements will be multiplied. Consider that an n-bit $-X$ is to be multiplied with an n-bit $-Y$. The product P' is given by

$$P' = (2^n - X)(2^n - Y) = 2^{2n} - 2^n X - 2^n Y + XY \tag{4.9}$$

However, the expected result is

$$P = XY \tag{4.10}$$

The difference is

$$P - P' = -2^{2n} + 2^n X + 2^n Y \tag{4.11}$$

The term 2^{2n} denotes a carry-out bit from the Most Significant Bit (MSB) that can be ignored. To get the correct result, correction factors for both multiplier and multiplicand should be added.

4.3 Types of Multiplier Architectures

The multiplier architecture can be generally classified into the following categories: serial, parallel, and serial-parallel.

4.3.1 Serial multipliers

The serial multiplier uses a successive addition algorithm. They are simple in structure because both the operands are entered in a serial manner. Therefore, the physical circuit requires less hardware and a minimum amount of chip area. However, the speed performance of the serial multiplier is poor due to the operands being entered sequentially.

4.3.2 Parallel multipliers

Three important criteria to be considered in the design of multipliers are the chip area, speed of computation and power dissipation. Most advanced digital systems incorporate a parallel multiplication unit to carry out high-speed mathematical operations. A microprocessor requires multipliers in its arithmetic logic unit and a digital signal processing system requires multipliers to implement algorithms such as convolution and filtering.

Today, high-speed parallel multipliers with much larger areas and higher complexity are used extensively in Reduced Instruction Set Computers (RISC), Digital Signal Processing (DSP), and graphics accelerators. Some examples of the parallel multiplier are the array multipliers such as the Braun multiplier and Baugh-Wooley multiplier, as well as the tree multipliers like the Wallace multiplier. Array multipliers have a more regular layout, although tree multipliers are generally faster. The major drawback of these multipliers is the relatively larger chip area consumption. It presents high-speed performance, but it is expensive in terms of silicon area and power consumption. This is because for parallel multipliers both operands are input to the multiplier in a parallel manner. As a result, the circuitry occupies a much larger area and is more complex as compared to serial multipliers.

4.3.3 Serial-parallel multipliers

The serial-parallel multiplier serves as a good trade-off between the time-consuming serial multiplier and the area-consuming parallel multipliers. These multipliers are used when there is a demand for both high speed and small area. In a device using the serial-parallel multiplier, one operand is entered serially and the other is stored in parallel with a fixed number of bits. The resultant enhancement in the processing speed and the chip area will become more significant when a large number of independent operations are performed.

A tidy breakdown of different types of digital multipliers is portrayed in Fig. 4.1. Contemporary digital signal processing algorithms for image processing and telecommunication applications are increasingly dependent on both matrix arithmetic and vector-like arithmetic. In addition, given the exponentially rising processor performance requirement and

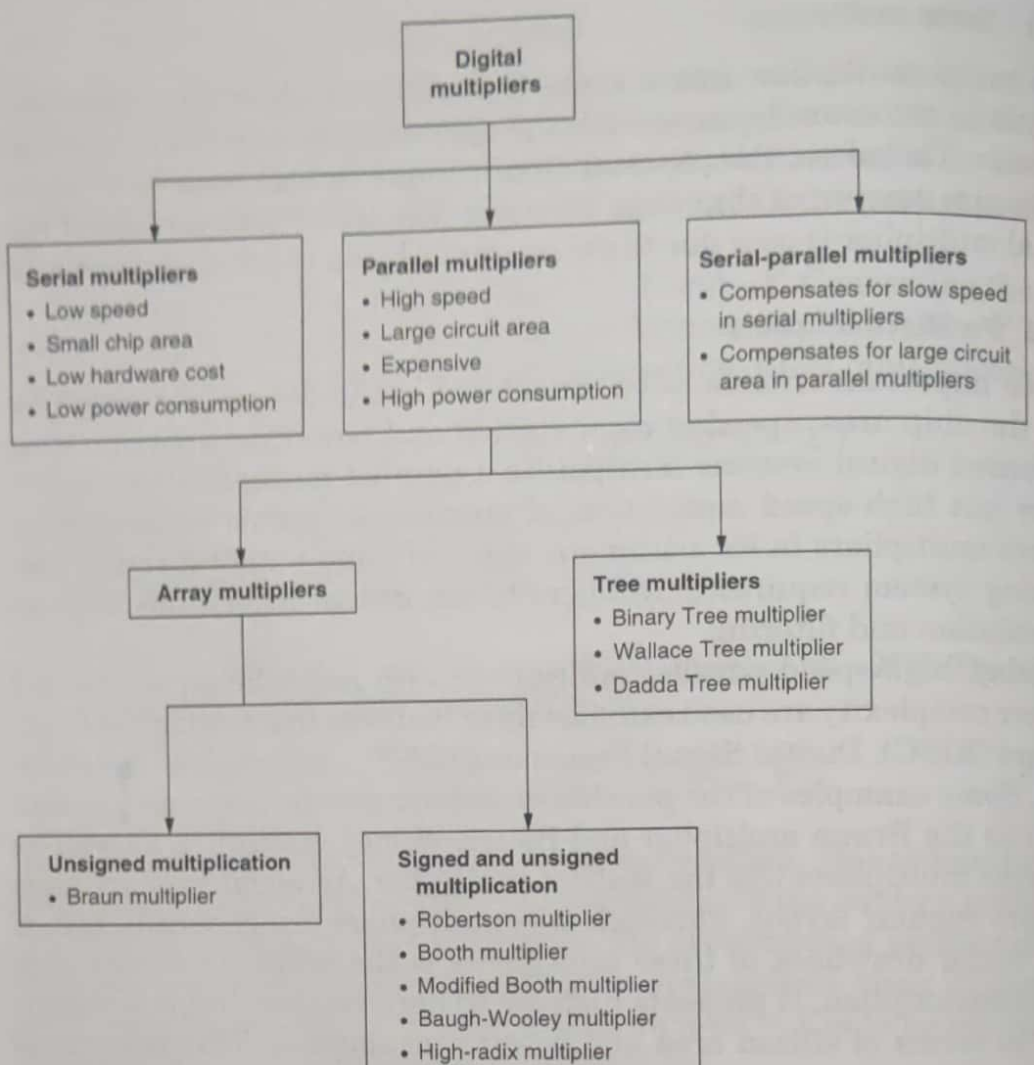


Figure 4.1 Types of digital multipliers.

the arithmetic-intensive nature of many applications, such as speech and image processing, waveform shaping, infinite impulse response digital filtering, channel equalization, networking, multimedia and computer vision, high-speed multipliers will continue to be in high demand. Therefore, in the following sections, only different types of parallel multipliers will be discussed in detail.

4.4 Braun Multiplier

Braun Edward Louis first proposed the Braun multiplier in 1963 [4]. It is a simple parallel multiplier that is commonly known as the Carry Save Array Multiplier. This multiplier is restricted to performing multiplication of two unsigned numbers. It consists of an array of AND gates and adders arranged in an iterative structure that does not require logic registers. This is also known as the non-additive multiplier since it does not add an additional operand to the result of the multiplication.

4.4.1 Architecture of Braun multiplier

An $n \times n$-bit Braun multiplier requires $n(n-1)$ adders and n^2 AND gates [1, 5]. One efficient implementation of the Braun multiplier is the regular layout of the adder array as shown in Fig. 4.2. The internal structure of the full adder used in the Braun multiplier is depicted in Fig. 4.3 [6]. This makes Braun multipliers ideal for Very Large Scale Integration (VLSI) and Application Specific Integrated Circuit (ASIC) realization.

Each of the $X_i Y_j$ product bits is generated in parallel with the AND gates [1]. Each partial product can be added to the previous sum of partial products by using a row of adders. The carry-out signals are shifted one bit to the left and are then added to the sums of the first adder and the new partial product. The shifting of the carry-out bits to the left is done by a Carry Save Adder (CSA). As the carry bits are passed diagonally downward to the next adder stage, there is no horizontal carry propagation for the first four rows. Instead, the respective carry bit is "saved" for the subsequent adder stage. Ripple Carry Adders (RCA) are used at the final stage of the array to output the final result.

4.4.2 Performance of Braun multiplier

The Braun multiplier performs well for unsigned operands that are less than 16 bits, in terms of speed, power and area. Besides, it has a simple and regular structure as compared to other multiplier schemes. However, the number of components required in building the Braun multiplier increases quadratically with the number of bits. This makes the Braun multiplier inefficient and so it is rarely employed while handling large operands. Another pitfall of the Braun multiplier is its potential susceptibility to glitching problems at the last stage of the full adders due to the exploitation of the Ripple Carry Adders (RCA).

4.4.3 Speed consideration

The delay of the Braun multiplier is dependent on the delay of the full adder cell and also on the final adder in the last row. In the multiplier array, a full adder with balanced carry and sum delays is desirable because the sum and carry signals are both in the critical path. The speed and power of the full adder are very important for large arrays.

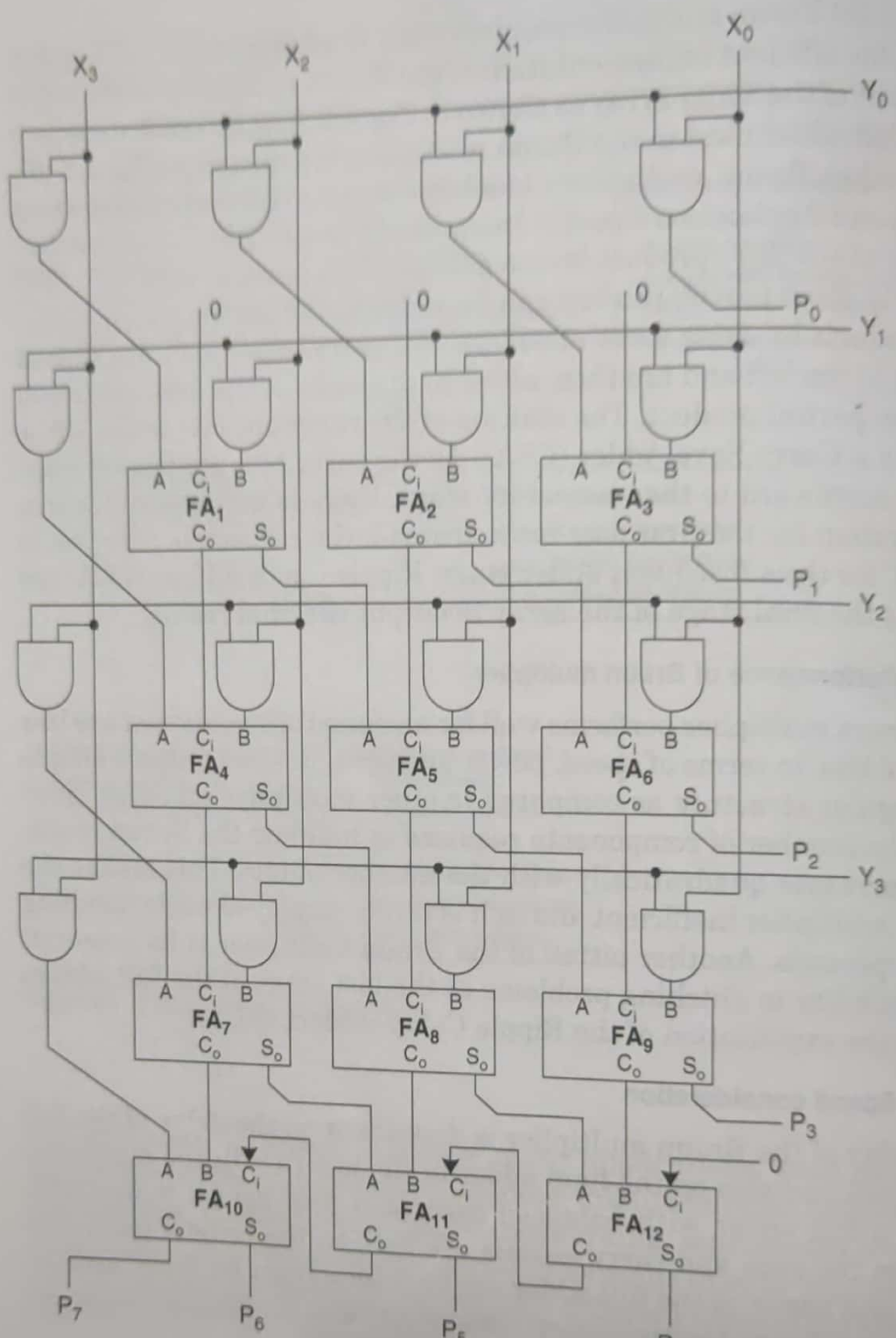
The worst-case multiplication time of a Braun multiplier can be expressed as [5]

$$t_{\text{Braun}} = (n-1)t_{\text{carry-save}} + t_{\text{AND}} + (n-1)t_{\text{Ripple-Carry}} \tag{4.12}$$

where $t_{\text{carry-save}}$ = time required to generate Carry-out (C_{out}) or Sum (S_{out}) at the output after the inputs are supplied to a CSA

$t_{\text{Ripple-Carry}}$ = time taken for the Carry-out (C_{out}) or Sum (S_{out}) to be generated at the output after the inputs are supplied to a RCA

t_{AND} = delay of an AND gate



X: 4-bit multiplicand
Y: 4-bit multiplier
P: 8-bit product of X and Y
$P_n = X_i Y_j$ is a product bit

Figure 4.2 Schematic diagram of a 4 × 4-bit Braun multiplier.

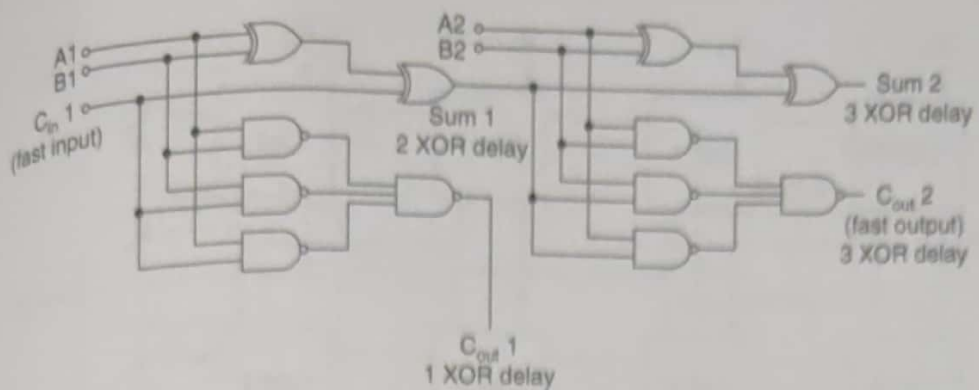


Figure 4.3 Two optimally interconnected full adders.

4.4.4 Enhanced Braun multiplier

Replacing the full adders FA1, FA2, FA3, and FA12 in Fig. 4.2 with half adders can enhance the performance of the Braun multiplier. Each replacement will result in a savings of three logic gates. Nonetheless, even though the performance is improved when replacing full adders with half adders, the regularity of the structure gets disturbed if modified in such a manner.

Optimizing the interconnections between the adders, so that the delay throughout each adder's path is approximately the same, can enhance the performance of the Braun multiplier. The connections are done according to the principle that the long delay path originating from the previous adder should be connected to the short delay path of the next one. Figure 4.3 shows a modified Braun multiplier obtained by optimally interconnecting two full adders with fast input and fast output [7]. The long delay path coming from the previous adder can be connected to either input A2 or input B2. This interconnection can be applied to the partial product array using full adders. This is because all the partial product bits in the same bit position are logically the same and therefore interchangeable.

4.5 Baugh-Wooley Multiplier

The Baugh-Wooley multiplier is an enhanced version of the Braun multiplier. It is designed to cater to multiplication of both signed and unsigned operands, which are represented in the 2's complement number system [8]. The partial products are adjusted so that the negative signs are moved to the last steps, which in turn maximize the regularity of the multiplication array.

4.5.1 Architecture of Baugh-Wooley multiplier

The architecture of the Baugh-Wooley multiplier is also based on the carry-save algorithm. It inherits the regular and repeating structure

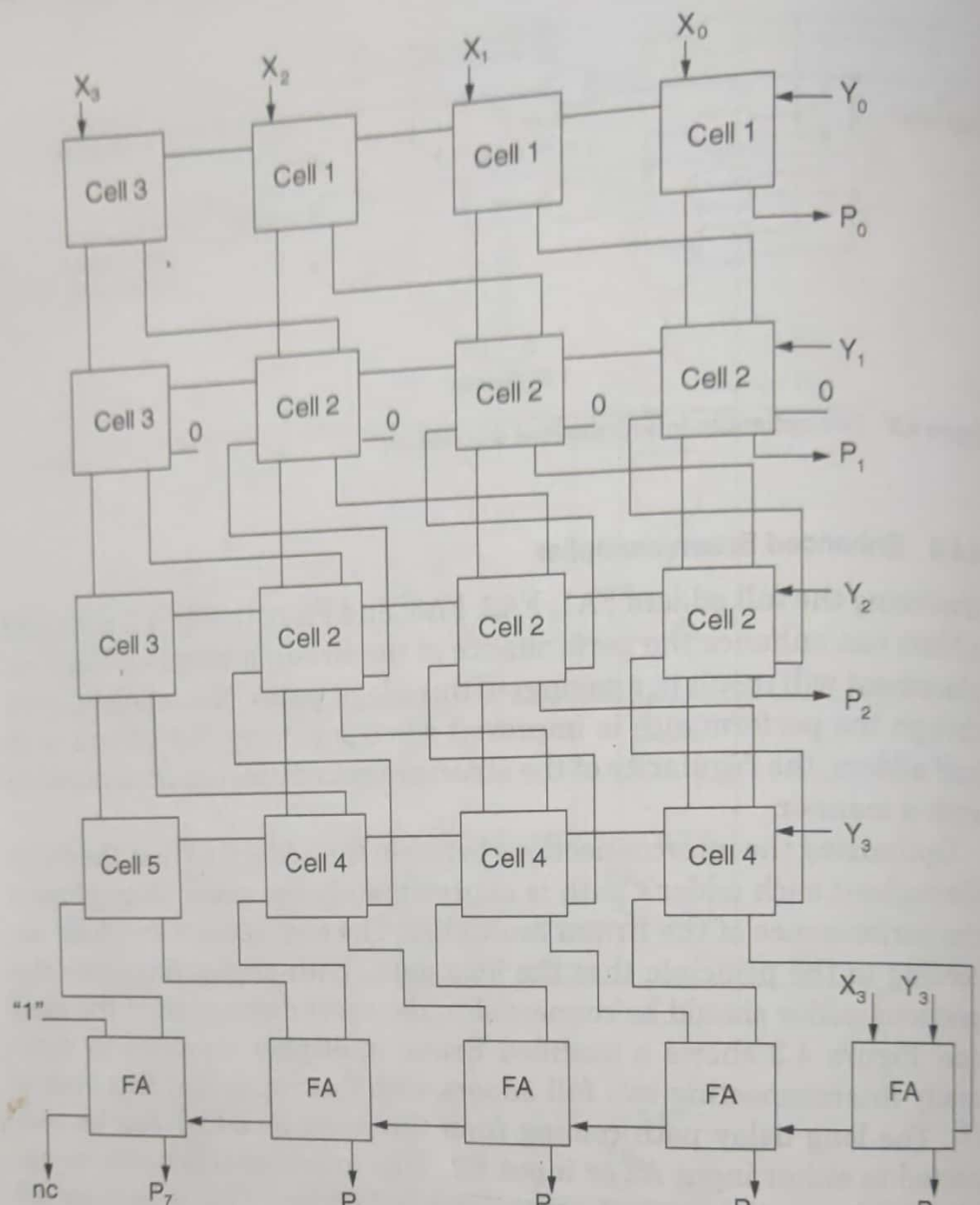


Figure 4.4 Schematic circuit of a 4 × 4-bit Baugh-Wooley multiplier.

of the array multiplier. The structure of a 4 × 4-bit 2's complement multiplier is shown in Fig. 4.4, with the cell number representing the type of basic cell [1, 9]. The macroscopic views of these basic cells are shown in Fig. 4.5.

4.5.2 Algorithm of Baugh-Wooley multiplier

The Baugh-Wooley multiplier operates on signed operands with 2's complement representation to make sure that the signs of all the partial products are positive. To reiterate, the numerical values of the 2's complement numbers, X and Y, can be obtained by using the following

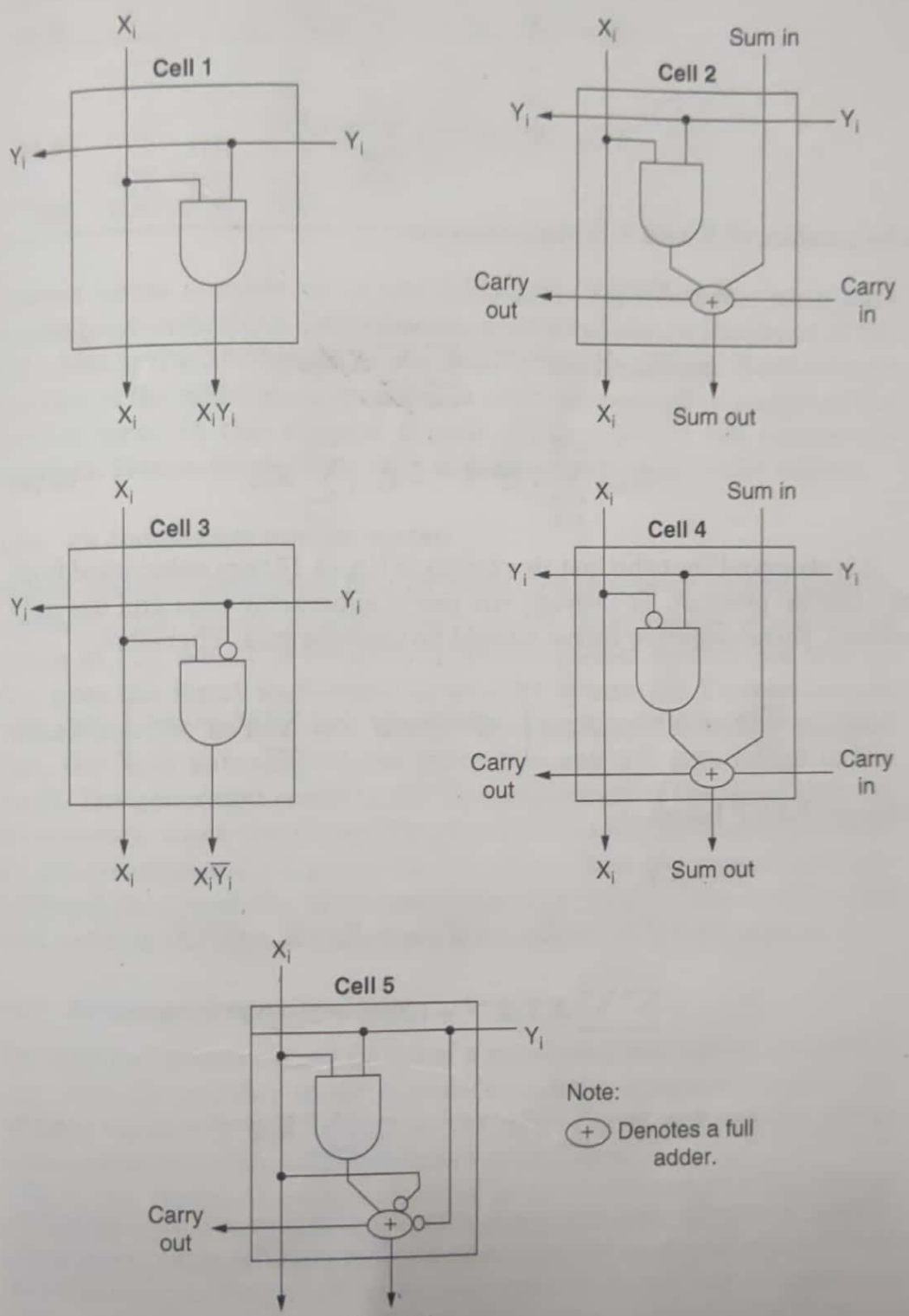


Figure 4.5 The five basic building blocks of the Baugh-Wooley multiplier.

expressions [1]:

$$X = -X_{n-1}2^{n-1} + \sum_{i=0}^{n-2} X_i 2^i \tag{4.13}$$

$$Y = -Y_{n-1}2^{n-1} + \sum_{j=0}^{n-2} Y_j 2^j \tag{4.14}$$

The product of X and Y is expressed as

$$\begin{aligned} P &= XY \\ &= X_{n-1}Y_{n-1}2^{2n-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{n-2} X_i Y_j 2^{i+j} \\ &\quad -X_{n-1}\sum_{j=0}^{n-2} Y_j 2^{n+j-1} - Y_{n-1}\sum_{i=0}^{n-2} X_i 2^{n+i-1} \end{aligned} \tag{4.15}$$

It is observed that the last two terms of Eq. (4.15) are subtracted from the partial product. To prevent the use of subtractor cells and use only adders, these negative terms should be transformed. Therefore,

$$-X_{n-1}\sum_{j=0}^{n-2} Y_j 2^{n+j-1} = X_{n-1}\left(-2^{2n-2} + 2^{n-1} + \sum_{j=0}^{n-2} \overline{Y}_j 2^{n+j+1}\right) \tag{4.16}$$

The product P becomes

$$\begin{aligned} P &= XY \\ &= -2^{2n-1} + (\overline{X}_{n-1} + \overline{Y}_{n-1} + X_{n-1}Y_{n-1})2^{2n-2} \\ &\quad + \sum_{i=0}^{n-2}\sum_{j=0}^{n-2} X_i Y_j 2^{i+j} + (X_{n-1} + Y_{n-1})2^{n-1} \\ &\quad + X_{n-1}\sum_{j=0}^{n-2} \overline{Y}_j 2^{n+j-1} + Y_{n-1}\sum_{i=0}^{n-2} \overline{X}_i 2^{n+i-1} \end{aligned} \tag{4.17}$$

Using a step-by-step approach, this 2's complement multiplication algorithm can be converted into an equivalent parallel array expression, as adopted by the Baugh-Wooley multiplier. Each partial product bit is the result of an AND gate of a multiplier. The signs of all partial product bits are positive. The computation steps of a 4 × 4-bit array multiplier are shown in Table 4.2.

The multiplication result of a 4 × 4-bit multiplier is an 8-bit output, hence there are eight vertical columns shown in Table 4.2. Each of these

TABLE 4.2 Illustration of the Computational Process of a 4 × 4-bit Multiplier

7	6	5	4	3	2	1	0
				x_3	x_2	x_1	x_0
				y_3	y_2	y_1	y_0
				$\overline{x_3y_0}$	x_2y_0	x_1y_0	x_0y_0
			$\overline{x_3y_1}$	x_2y_1	x_1y_1	x_0y_1	
		$\overline{x_3y_2}$	x_2y_2	x_1y_2	x_0y_2		
	$\overline{x_3y_3}$	$\overline{x_2y_3}$	$\overline{x_1y_3}$	$\overline{x_0y_3}$			

product terms is made up of one AND gate. The variables with bars denote prior inversions. Inverters are connected before the input of the full adder or the AND gates as required by the algorithm. Each column represents the addition in accordance with the respective weight of the product term. In this scheme, a total of $n(n-1)+3$ full adders are required. Hence, for the case of $n = 4$, the array requires 15 adders.

4.5.3 2's complement number system

Signed multiplicands must first be converted into their 2's complement representation before multiplication. A 2's complement generator is shown in Fig. 4.6. It is basically a combinational circuit that will either pass the input unchanged or convert it into the 2's complement form. When the control line Comp-Sig (Complementary Signal) goes high, the XOR gates invert the input bits and a 1 gets added to the result. The generated result is the 2's complement of the input bits. On the contrary, when the Comp-Sig goes low, the multiplicand inputs do not get inverted and a 0 gets added to them. Once the signed multiplicands get processed, the Most Significant Bit (MSB) of the result would then indicate the sign of the result (1 for negative, 0 for positive).

4.5.4 Performance consideration

The area and power consumption of a number of multiplier structures vary with the number of bit operands and the layout strategies. Increasing regularity and locality at the silicon level reduces the power consumption in a standard-cell-based design flow.

Since the Baugh-Wooley multiplier is an evolvement of the Braun multiplier, its performance can also be improved by using the earlier-mentioned optimized interconnections, as shown in Fig. 4.7 [6].

4.6 Booth Multiplier

Area-efficient and fast multipliers are the essential blocks for high-performance computing. Therefore, multipliers should be small enough so that a larger number of them may be integrated on a single chip.

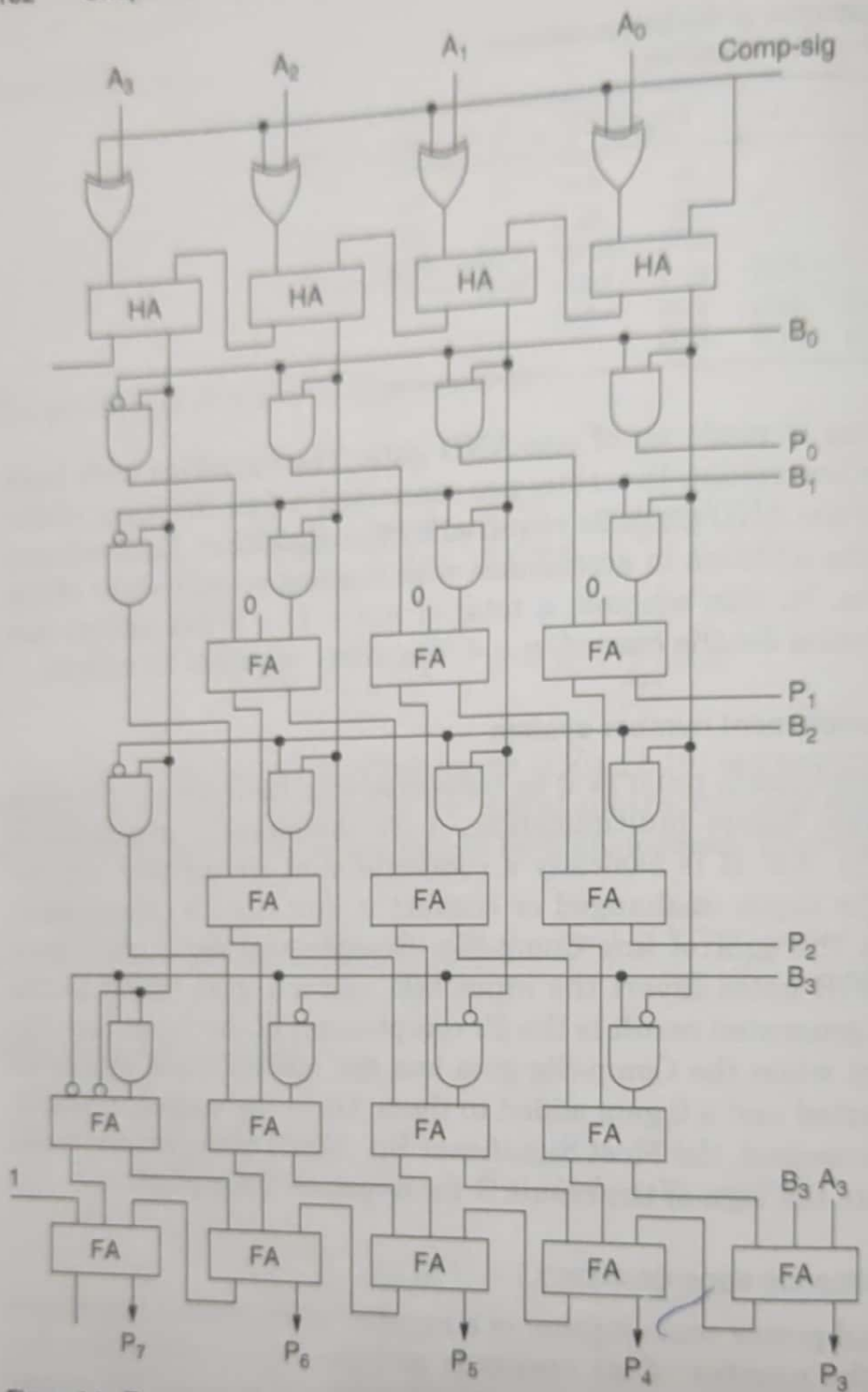


Figure 4.6 Baugh-Wooley multiplier with 2's complement generator.

Conventional array multipliers, like the Braun multiplier and the Baugh-Wooley multiplier, achieve comparatively good performance, but they require large areas of silicon, unlike the add-shift algorithms, which require less hardware and exhibit poorer performance [10]. The Booth multiplier makes use of the Booth encoding algorithm in order to reduce the number of partial products by considering two bits of the multiplier at a time, thereby achieving a speed advantage over other

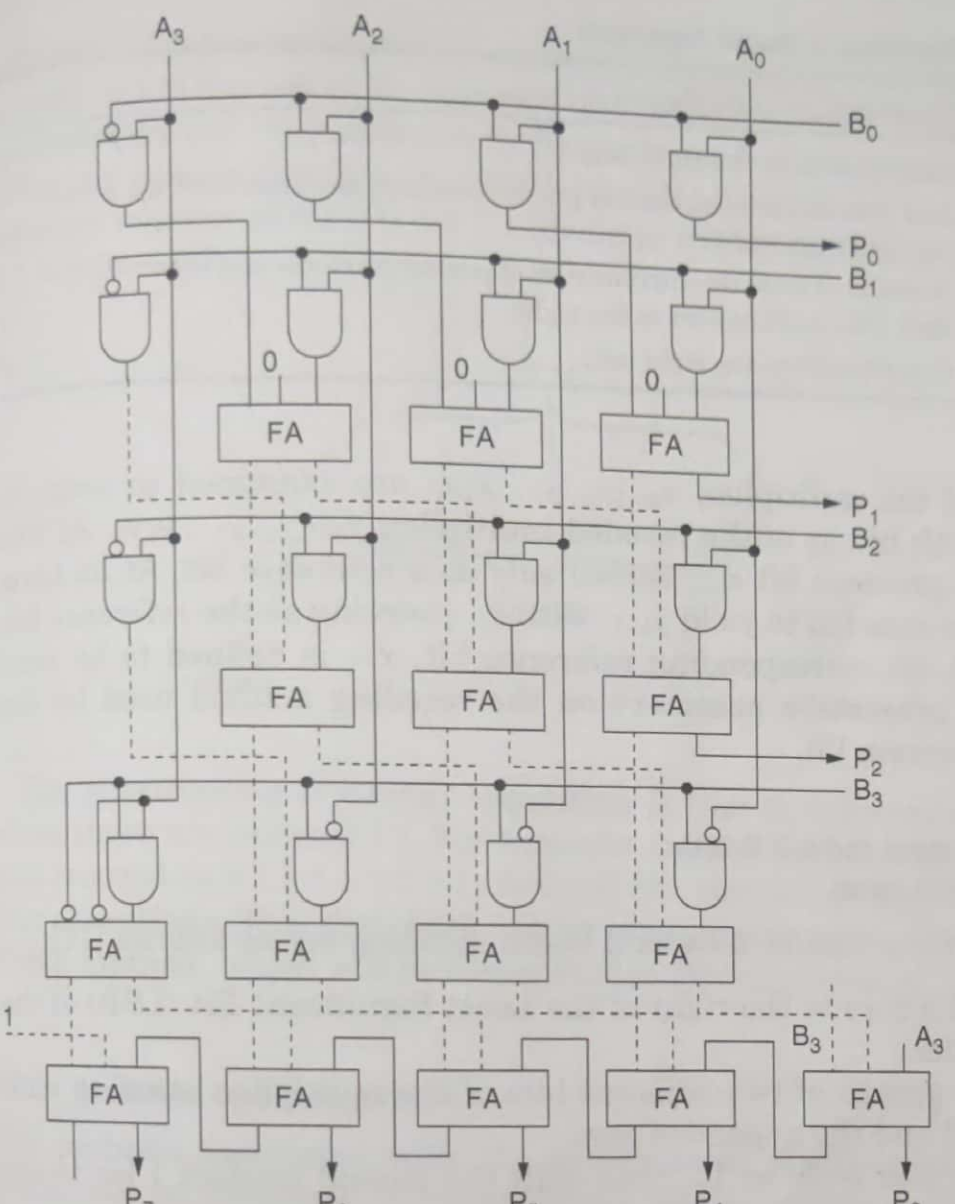


Figure 4.7 Modified Baugh-Wooley multiplier with optimized interconnection (dotted lines).

multiplier architectures. This algorithm is valid for both signed and unsigned operands.

4.6.1 Booth's algorithm

In 1951, A.D. Booth proposed the Booth algorithm (also known as the radix-2 algorithm) for multiplication that accepts numbers in 2's complement form, based on radix-2 computation. It can handle signed binary multiplication by using 2's complement representation [11]. This increases the complexity of how the signs of the operands get stored in auxiliary circuits.

In order to start the process, an imaginary 0 is appended to the right of the multiplier. Subsequently, the current bit x_i and the previous

TABLE 4.3 Recoding in Booth Algorithm

x_i	x_{i-1}	Operation on multiplicand A.	y_i
0	0	One bit shift to the right only.	0
0	1	Add A to the existing sum of partial products and then shift the result one bit to the right.	1
1	0	Subtract A from the current sum of partial products and then shift the result one bit to the right.	$\bar{1}$
1	1	One bit shift to the right only.	0

bit x_{i-1} of the multiplier, $x_{n-1}x_{n-2}\ldots x_1x_0$ are examined in order to yield the ith bit, y_i of the recoded multiplier, $y_{n-1}y_{n-2}\ldots y_1y_0$. At this point, the previous bit x_{i-1} serves only as a reference bit. At its turn, x_{i-1} will be recoded to yield y_{i-1}, with x_{i-2} serving as the reference bit. For $i = 0$, its corresponding reference bit, x_{-1} is defined to be zero. Table 4.3 presents a summary on the recoding method used by the Booth's theorem [3].

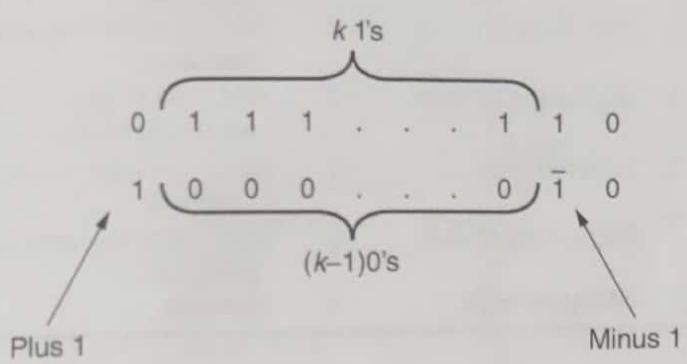
4.6.2 Standard radix-2 Booth multiplication rules

The rules for a standard radix-2 Booth recoding are as follows [11]:

1. Append a zero to the right of the Least Significant Bit (LSB) of the multiplier.
2. Inspect groups of two adjacent bits of the multiplier, starting with the LSB and the appended zero.
   - If the pair is 00 or 11, then shift the partial product 1 bit to the right.
   - If the pair is 01, then add the multiplicand to the partial product and shift the partial product 1 bit to the right.
   - If the pair is 10, subtract the multiplicand from the partial product and shift the new partial product to the right by 1 bit.
3. Proceed with overlapping pairs of bits such that the MSB of a pair becomes the LSB of the next pair. In this manner, 1 bit of the multiplier number is eliminated in each pass through the algorithm.
4. When the last pair of bits is examined, the partial product is updated following the rules except that no shift is performed.

The multiplication procedure uses recoding of the 2's complement multiplier with the underlying fact that a k–long sequence of 1's is equivalent to a $(k-1)$-long sequence of 0's. This replacement of a string of 1's by 0's helps reduce the partial products. Any arbitrary binary number can be recoded in this fashion, so that each group of 1 or more

adjacent 1's gets substituted by a minus 1 (denoted by $\bar{1}$) at its low end, a positive 1 to the left of its high end and intervening 0's. This recoding methodology need not necessarily be performed in any predetermined order and can even be done in parallel for all bit positions. For example, a binary number of 0 1 0 0 0 0 1 1 (decimal 67) is recoded to 1 $\bar{1}$ 0 0 0 1 0 $\bar{1}$ (decimal 67).



The shortcoming of Booth's algorithm is that it becomes inefficient when there are isolated 1's. For example, 0 0 1 0 1 0 1 0 1 (decimal 85) gets recoded as 0 1 $\bar{1}$ 1 $\bar{1}$ 1 $\bar{1}$ 1 $\bar{1}$ (decimal 85), requiring eight, instead of four operations. This drawback can be overcome by using the modified Booth method, which will be illustrated in the next section.

4.6.3 Modified Booth algorithm

The problem associated with the conventional Booth encoding algorithm can be overcome by processing 3 bits at a time during recoding. Recoding the multiplier in a higher radix is a powerful way to speed up the standard Booth multiplication algorithm. Therefore, since in each cycle a greater number of bits can be inspected and eliminated, the total number of cycles required to obtain the product gets reduced [7].

The number, n, of the bits inspected in radix, r, is given by [7]

$$n = 1 + \log_2 r \tag{4.18}$$

For example, in each cycle of the radix-4 algorithm, 3-bits get inspected and two get eliminated. The modified Booth's algorithm also commences by appending a zero to the right of the LSB. The multiplier bits x_i and x_{i-1} get recoded into y_i and y_{i-1} with x_{i-2} acting as a reference bit. Similarly, in another step, x_{i-2} and x_{i-3} get recoded into y_{i-2} and y_{i-3} while x_{i-4} serves as the reference bit. Thus, each multiplier gets divided into substrings of 3 bits with adjacent groups sharing a common bit. The term x_{-1} with a zero value is added on the right side of the

TABLE 4.4 Modified Booth Recoding

x_i	x_{i-1}	x_{i-2}	y_i	y_{i-1}	Comments	Recoded digit	Operation on multiplicand A before shifting 2 bits to the right
0	0	0	0	0	String of zeros	0	Nothing
0	0	0	0	1	End of ones	+1	Add A to the existing sum of partial products
0	0	1	0	1	Isolated one	+1	Add A to the existing sum of partial products
0	1	0	0	1	End of ones	+2	Add $2A$ to the existing sum of partial products
0	1	1	1	0	Beginning of ones	−2	Add $-2A$ to the existing sum of partial products
1	0	0	$\bar{1}$	0	Isolated zero	−1	Add $-A$ to the existing sum of partial products
1	0	1	0	$\bar{1}$	Beginning of ones	−1	Add $-A$ to the existing sum of partial products
1	1	0	0	$\bar{1}$	String of ones	0	Nothing
1	1	1	0	0			

multiplier X. The rightmost 3-bit group becomes $x_1x_0(x_{-1})$, followed by $x_3x_2(x_1)$, $x_5x_4(x_3)$, and so on.

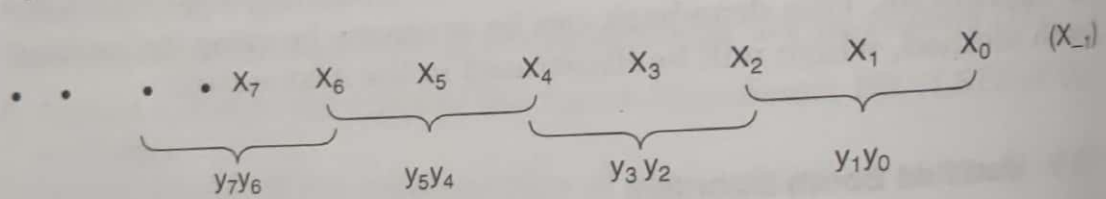


Table 4.4 illustrates a set of rules for mechanizing the radix-4 modified Booth algorithm for odd values of i, such as 1, 3, and 5 [3]. An easier way to identify the required operation is by adhering to the formula $x_{i-1}+x_{i-2}-2x_i$. It can be observed that the radix-4 modified Booth algorithm deals with the isolated 1 or 0 more efficiently than Booth's original method. As an example, for the cases of the isolated 1 and the isolated 0, X gets recoded into 01 and $0\bar{1}$, respectively. Simplification takes place and only a single operation is required.

Booth's recoding method does not propagate the carry to subsequent stages. Each of the recoded digits can be obtained independently. Figure 4.8 shows the basic block diagram of a modified Booth multiplier [1]. A colon is used to represent concatenation of bits. It consists of the Booth encoder and the sign extension bits, the multiplier array, which comprises the partial product's generators and 1-bit adders, and the final stage adder, which executes the $2n$-bit addition. Hence, for each of the distinct 3-bit segments of the multiplier, X, the Booth encoder generates the five signals: 0, $+1A$, $+2A$, $-1A$, and $-2A$.

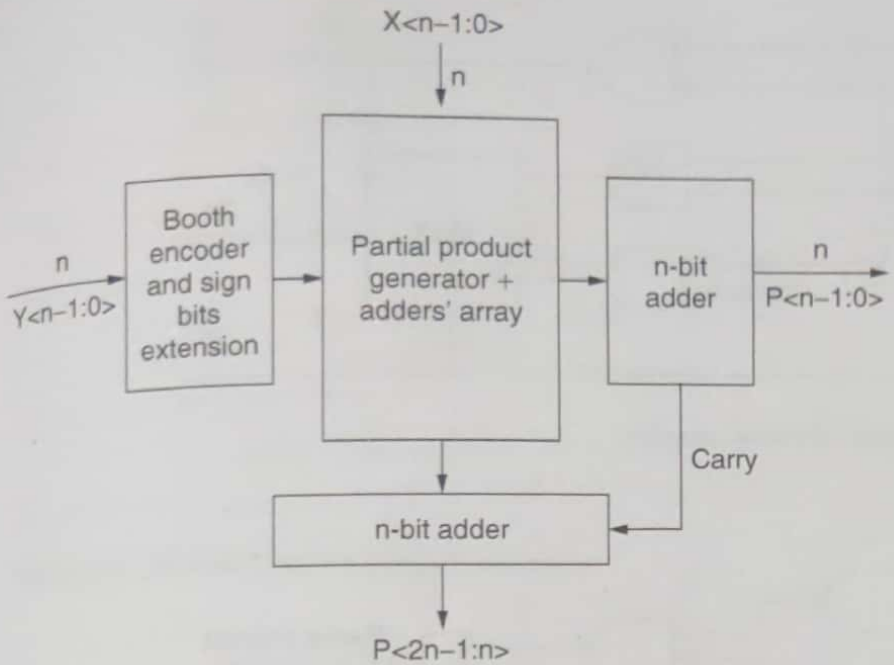


Figure 4.8 Block diagram of an $n \times n$-bit modified Booth multiplier.

4.6.4 Booth encoder

The Booth encoder implements Booth encoding of the three multiplier bits and also handles the sign extension logic. Each encoder is dedicated to one partial product in the array. Since there is a circuit for each of the five possible generated partial product signals, one and only one signal is high during the steady state operation. The carry propagation circuits are independent of the partial product circuits and they do not share any inputs.

In the conventional modified Booth algorithm encoder, three signals X_j, $2X_j$, and M_j, are generated from three adjacent bits, b_{j-1}, b_j, and b_{j+1} for selecting a partial product that is 0, $+A$, $-A$, $+2A$, or $-2A$. In this case, A is a multiplicand of n-bit width. The X_j and $2X_j$ signals show whether the partial product is doubled and an active M_j means that the negative partial product should be used. The structure of the Booth encoder is shown in Fig. 4.9 and the partial product generator for this implementation is depicted in Fig. 4.10 [7].

Direct implementation of $A * B$ and $-A * B$ is not possible using the circuits shown in Figs. 4.9 and 4.10. Nevertheless, it can be done using the sign-select Booth encoder, which uses an extra control signal "Sgn," to directly implement either $A * B$ or $-A * B$. An extra output signal PL_j is also provided to represent the selection of the positive partial product. The sign-select Booth encoder is given in Fig. 4.11 [7].

In the conventional encoder, only M_j, which represents the negative partial product, can be generated. With the sign-select Booth encoder,

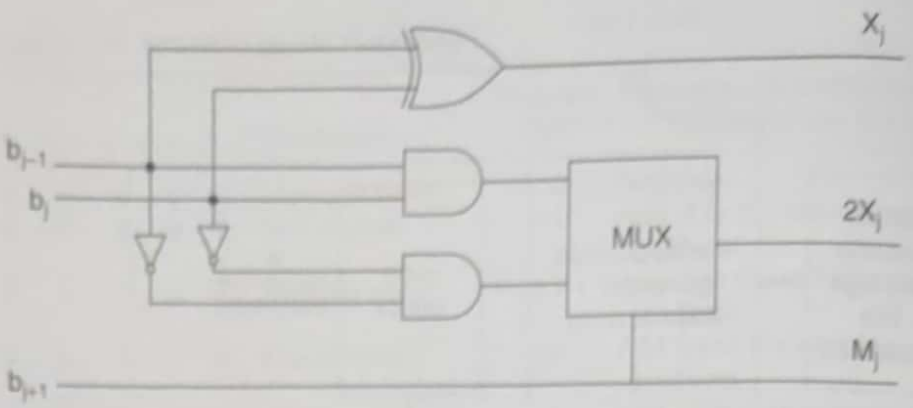


Figure 4.9 Structure of Booth encoder.

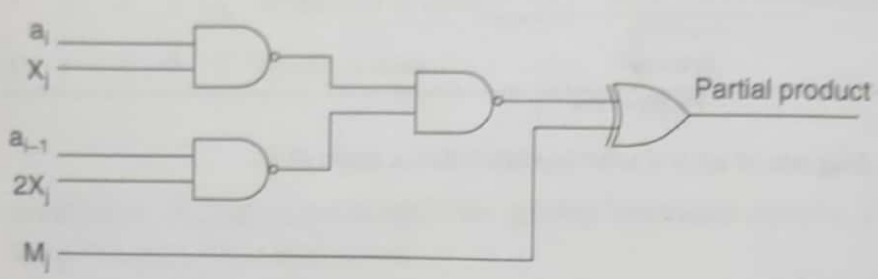


Figure 4.10 Conventional partial product generator.

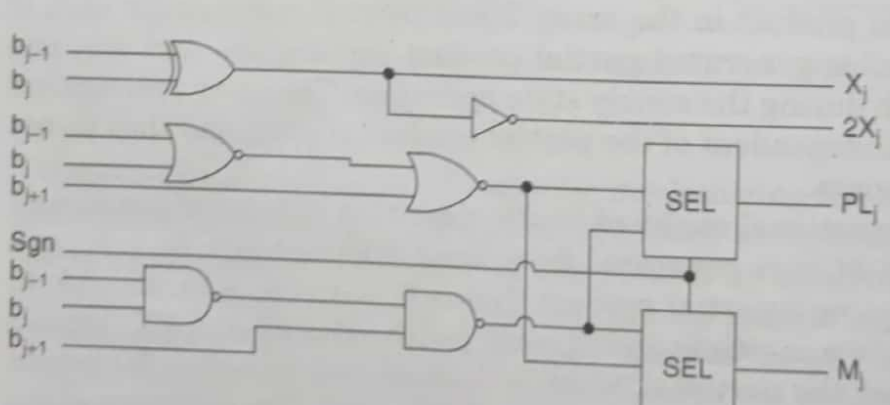


Figure 4.11 Sign-select Booth encoder.

signals for both negative and positive partial products can be generated. PL_j and M_j signals become active when the partial product is positive or negative, respectively. Using this Booth encoder, the partial product generator can also be simplified, as shown in Fig. 4.12. The simplicity of this circuit leads to smaller area usage and low-power consumption.

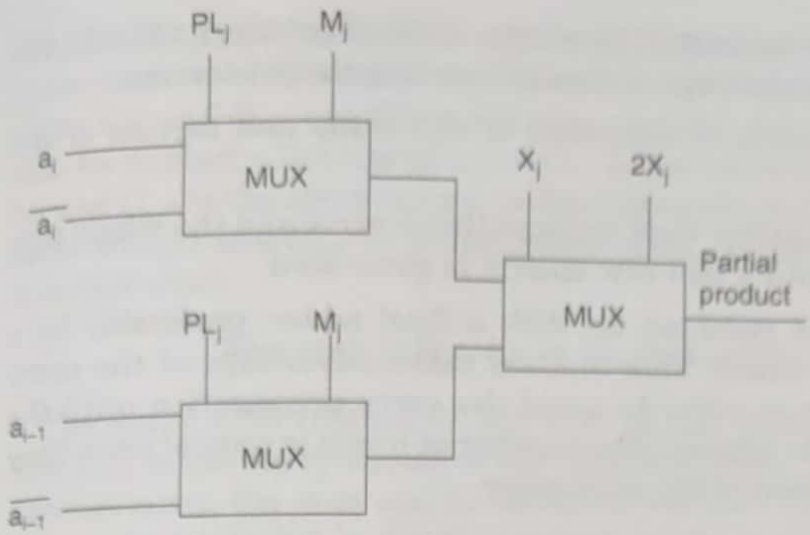


Figure 4.12 Modified partial product generator.

4.7 Wallace Tree Multiplier

Booth's algorithm effectively reduces the number of partial products by half. However, for large-operand multipliers such as 32-bit and above, the partial products are longer than 16 bits and are considered unacceptably large. In this case, the performance of the modified Booth algorithm is degraded. The Wallace tree multiplication algorithm, however, can reduce the number of partial products by employing multiple input compressors capable of accumulating several partial products concurrently [12].

In 1964, C.S. Wallace proposed the Wallace Tree multiplier, which can handle the multiplication process for large operands. This is achieved by minimizing the number of partial product bits in a fast and efficient way by means of a CSA tree constructed from 1-bit full adders.

This algorithm uses pseudo adders to add three inputs and then produces two outputs whose sum is equivalent to the sum of the three inputs. The main advantage of using these pseudo adders comes from having the ability to avoid carry propagation, which then increases the speed of multiplication. Using several pseudo adders concurrently can further enhance multiplication speed. The Wallace tree approach can reduce the number of partial products, in parallel, with a resulting overall delay proportional to $\log_{3/2} n$, for n number of rows.

The main disadvantage of the Wallace tree algorithm is that the architecture exhibits some irregularities in the layout because it has a relatively complicated interconnection scheme. In general, its multiplication process can be summarized as follows:

1. After generating the partial products, a set of counters reduces the partial product matrix but it does not propagate the carries.
2. The resulting matrix is composed of the sums and carries of the counters.
3. Another set of counters then reduces this matrix and the whole process continues until a two-row matrix is generated.
4. The two rows get summed up with a final adder, preferably by a carry propagate adder. This method takes advantage of the carry save architecture in order to avoid the carry propagation until the final adder. In this scheme, the number of levels is crucial since they determine the speed of the multiplier.

The conventional Wallace tree algorithm reduces the propagation stages by incorporating 3:2 compressors [12].

4.7.1 4:2 compressors

In 1981, A. Weinberger of IBM originated the idea of 4:2 compressors. He proposed to reduce the number of propagation stages in tree multipliers by replacing the 3:2 compressors of the conventional Wallace tree algorithm with 4:2 compressors. The advantage of tree multipliers is that their speed increases logarithmically in proportion to the operand's length, as opposed to the case of iterative arrays, for which the speed increases linearly in proportion to the size of the operands. The 4:2 compressor is able to yield a much more regular structure than the 3:2 counter because it can reduce four inputs of the same weight to two.

The 4:2 compressor and its equivalent building blocks are depicted in Figs. 4.13(a) and 4.13(b), respectively [1]. Upon receiving four numbers

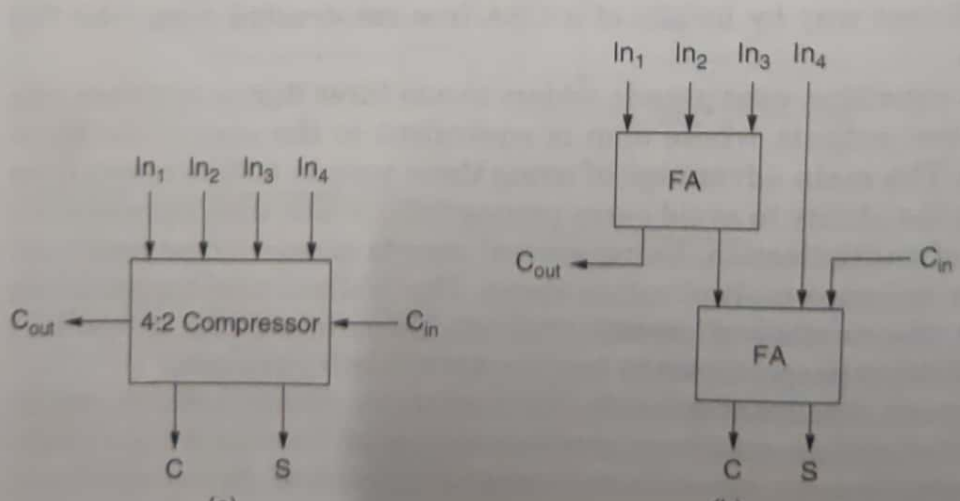


Figure 4.13 4:2 compressor: (*a*) schematic diagram; (*b*) equivalent circuit.

(In_1, In_2, In_3, and In_4) and a carry-in (C_{in}), it compresses them and then generates two numbers (S and C) and a carry-out (C_{out}). Interestingly, when C_{in} and C_{out} are taken into account, the 4:2 compressor could be also be dubbed a 5:3 compressor. C_{in} is the carry input from the lower bit and C_{out} is the carry output to the higher bit, so C_{in} and C_{out} have the same weight and C_{out} is independent of C_{in}. Hence, carry propagation does not occur.

$$\text{The sum is } S = In_1 \oplus In_2 \oplus In_3 \oplus In_4 \oplus C_{in} \tag{4.19}$$

The carry C is related to the sum of the first adder, In_4 and C_{in}. The C_{out} is the carry signal of the addition of In_1, In_2 and In_3. By this arrangement, the sum can be obtained via four XOR gate delays ($S = [[(In_1 \oplus In_2) \oplus In_3] \oplus In_4] \oplus C_{in}$), which is identical to the result in the Wallace tree structure using 2-layer carry save adders. Therefore, it can be rearranged to be

$$S = [(In_1 \oplus In_2) \oplus (In_4 \oplus C_{in})] \oplus C_{in} \tag{4.20}$$

With this arrangement, three XOR gate delays are involved. In this example the speed advantage can therefore be obtained.

The 4:2 compressors, using the optimized interconnections, as shown in Fig. 4.13, seriously affect how cells interconnect horizontally. These compressors, therefore, lead to a minimum propagation of the carry signal in the multiplier. Indeed, the 4:2 compressor is not even a counter because the two output bits, C and S, do not represent all the five possible input combinations. C_{out} is independent of C_{in}, so only limited carry propagation occurs. The advantage of using the 4:2 compressors becomes apparent in the much simpler and more regular wiring of the multiplier tree. Another noteworthy advantage is the notion of the horizontal and vertical signal path. The existence of a horizontal path has led to the idea of moving the critical path of the multiplier tree away from the center and toward the most significant end where the depth of the column of partial product bits is smaller than in the middle.

The advantage of proper interconnection of the fast inputs and the fast outputs has been illustrated in Fig. 4.3. The objective of proper interconnections between the fast inputs and fast outputs is to minimize the critical path of the 4:2 compressors. The delay through the 4:2 compressor is equivalent to three XOR gate delays, regardless of the path. If this is used to reduce the partial product bits in a 24 × 24-bit multiplier, the delay can be reduced by four XOR gate delays instead of using the regular 4:2 compressors. Furthermore, use of the 4:2 compressor permits the reduction of the vertical critical path while the path involving the carry propagation (horizontal path) remains unchanged. However, horizontal propagation is fast and it is limited to one bit per level.

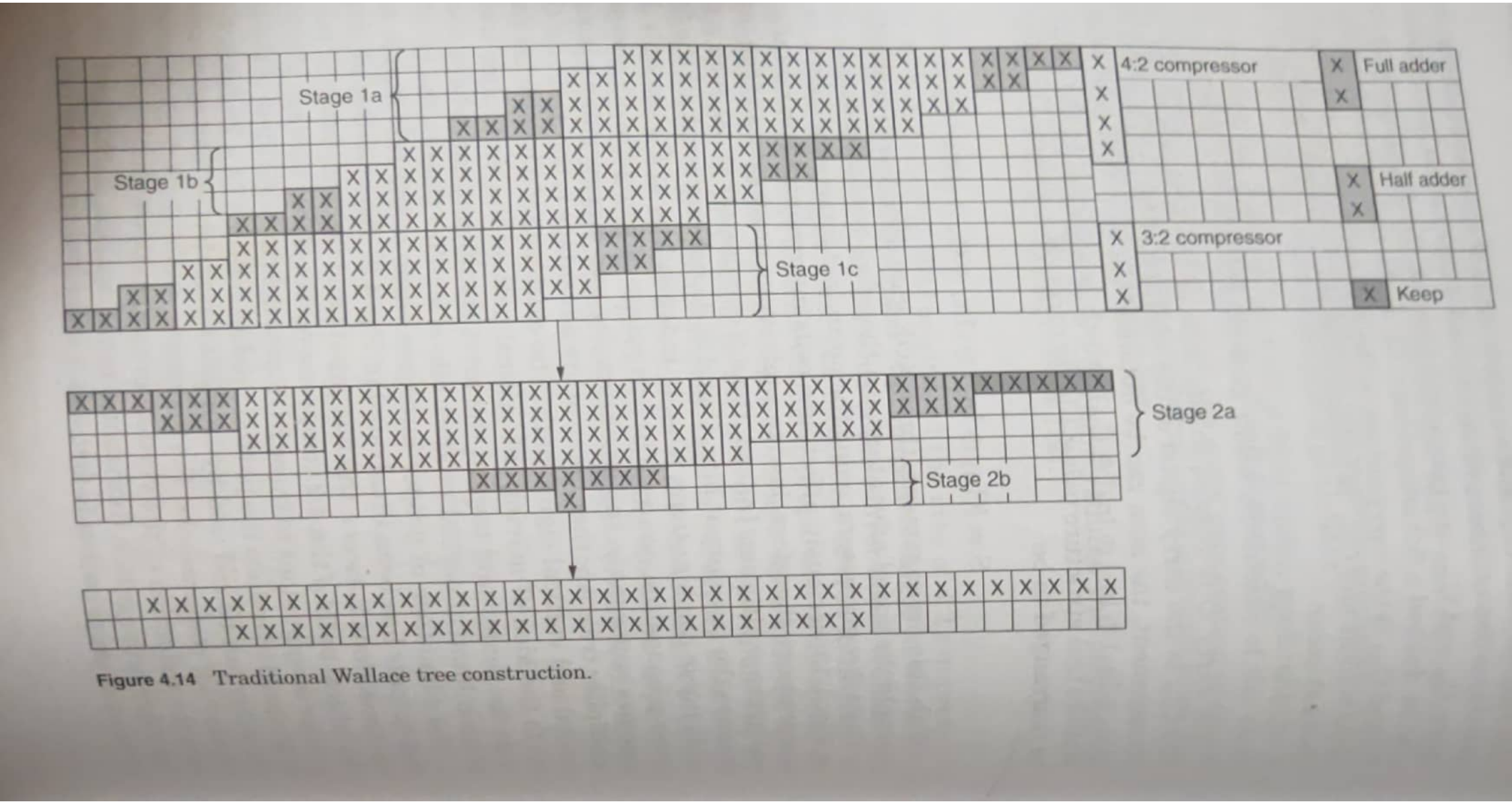


Figure 4.14 Traditional Wallace tree construction.

The speed for a particular algorithm implementation is generally calculated in terms of XOR delays [13]. This is so because the critical path in the multiplier consists of a path through a series of XOR gates that are independent of the algorithm used. Furthermore, the speed comparison is independent of the technology, which is characterized by its ability to realize a fast XOR function.

The idea of 4:2 compressors can be extended to 6:2 and 9:2 compressors. These new compressors can be built with the help of the (3, 2) counters and 4:2 compressors. For example, the 6:2 compressor can be built with one 4:2 compressor and two (3, 2) counters, whereas the 9:2 compressor can be built with one 6:2 compressor and three (3, 2) counters [13].

4.7.2 Wallace tree construction

The Wallace tree can be constructed in many ways. The following method considers all the bits in each set of four rows all at a time and compresses them all in the manner shown in Fig. 4.14 [14]. For example, to compress a 12-partial product tree, the Wallace tree uses 4:2 compressors, 3:2 compressors, full adders and half adders. Each 4:2 compressor takes in four bits from the same position, j, 1 bit from the previous position, $j-1$, (which is the carry-out of the compressor in the previous position) and outputs one sum bit in the position, j, and two carry-outs to the next position, $j+1$.

The superior performance of the Wallace tree comes at a heavy price: non-regularity and wasted area. In the conventional tree, the partial products in the tree get added in one direction, from top to bottom, with the number of compressors or adders and wires increasing as the adding stage keeps moving forward. Other than this, the compression process differs at each position because of the inconsistency in the number of partial products to be compressed. Because of its non-regularity, it becomes almost impossible to lay out a Wallace tree multiplier in a rectangular shape without wasting a significant amount of chip area. In a conventional Wallace tree, every partial product is added in one direction from top to bottom. Therefore the number of compressors or full adders increases as the adding stage moves downward, and the layout becomes wider in the bottom than at the top.

Figure 4.15 shows the increasing width of the adding stages. Each horizontal line corresponds to one stage of compressors in the Wallace tree. As illustrated in Fig. 4.16, when the tree is laid out in a rectangular shape, 38 percent of the overall rectangular area gets wasted. The wasted area is called the dead area.

The following approach can be adopted to solve the above-mentioned problem. The partial products are divided into two groups around the

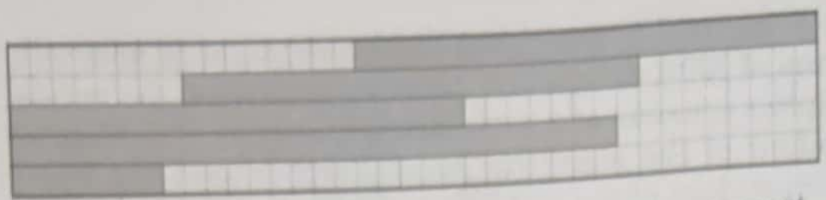


Figure 4.15 Layout of Wallace tree multiplier before arrangement.

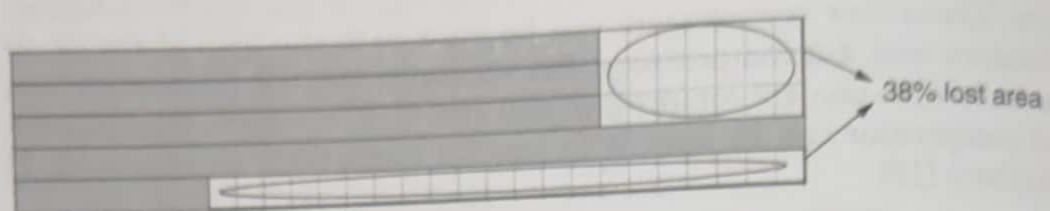


Figure 4.16 Layout of conventional Wallace tree multiplier after arrangement.

middle of the tree, such that the partial products in one group get added from the top all the way to the bottom, whereas the partial products in the other group get added from the bottom all the way to the top. This procedure can be explained as follows: First, the partial product bits are divided into two separate trees called the left tree and the right tree. The right tree consists of the least significant bits and the left tree consists of the most significant bits. The right tree then gets compressed in a Wallace tree fashion in an upward direction, whereas the left tree gets compressed in a downward direction. Figure 4.17 explains this procedure. In this way, area wastage can be minimized.

Now, one of the two sub-trees adds up in a downward direction while the other sub-tree adds up in an upward direction. As a result, the area

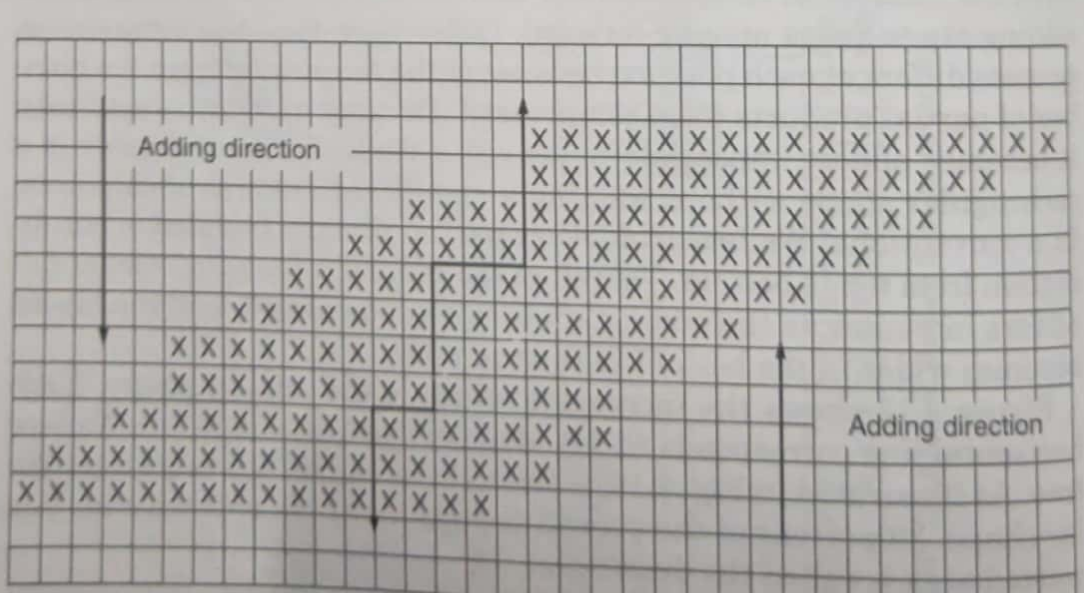


Figure 4.17 Splitting a Wallace tree into two trees.

(a) Regular Wallace tree (b) Modified Wallace tree

Figure 4.18 Sketch of layouts of the conventional and modified Wallace trees.

can be reduced greatly. The layouts of the conventional and modified Wallace trees are shown in Fig. 4.18.

4.8 Conclusions

This chapter gives a glimpse of the multiplication techniques, types of multiplier architectures and a wide-ranging review of the different multiplication algorithms. A myriad of eminent architectures of multipliers, including Braun multiplier, Baugh-Wooley multiplier, Booth's multiplier and Wallace tree multiplier have been covered.

Braun multiplier is useful for applications where the area is of major concern and the importance of high-speed is downplayed. This multiplier offers the advantage of high regularity in its structure. However, its drawback is that it is only meant for unsigned multiplication. In addition, its hardware requirement is greater than what the other multiplier algorithms require when handling operands that are greater than 16 bits.

The main motivation behind the realization of the Baugh-Wooley multiplier is to multiply negative numbers based on the 2's complement number representation. The Baugh-Wooley multiplier resembles the latter in layout regularity and enhancement methods. Like the Braun multiplier, it is very efficient in handling multiplicands of less than 16 bits but it becomes less efficient as the operands get larger.

Ever since Booth's algorithm got introduced, it has always been used and developed because it renders the advantage of reducing the number of partial products. This algorithm is useful for applications where the hardware cost is of major concern. Modified Booth algorithm can reduce the delay and implementation cost by a factor of two.

The Wallace tree multiplier is a fast multiplier with irregular routing. Wallace tree multiplier is widely used for high-speed applications. Since the wiring of Wallace tree is considerably complex, any solution to solve the dead area problem may add to a significant wiring overhead, which may cause an increase in the design time.

Adopting hybrid architectures such a combination of a modified Booth recoding technique and a Wallace tree structure can further enhance the multiplier's speed. The modified Booth recoding technique is capable of reducing the number of partial products to half while the Wallace