VHDL BASED MODELING AND DESIGN OF PARAMETERIZABLE MULTIPLIERS FOR TESTABILITY

A thesis submitted to the
Department of Electrical and Electronic Engineering
BUET, Dhaka
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering (Electrical and Electronic)

SHAIKH ARIF SHAMS

ROLL NO: 9406214P
SESSION: 1993-94-95

AUGUST 1998

The thesis titled "***VHDL based modeling and design of parameterizable multipliers for testability***" submitted by **Shaikh Arif Shams**, Roll No. 9406214P to the Department of Electrical and Electronic Engineering, BUET has been accepted as satisfactory for partial fulfillment of the requirements for the degree of Master of Science in Engineering (Electrical and Electronic).

DECLARATION

I hereby declare that this work has been done by me and it has not been submitted elsewhere for the award of any other degree or diploma.

Countersigned

ACKNOWLEDGEMENT

It is the author's pleasure to acknowledge his heartiest gratitude and profound obligation to his Supervisor, Dr. Syed Mahfuzul Aziz, Professor, Department of Electrical and Electronic Engineering, Bangladesh University of Engineering and Technology for his excellent supervision, continuous guidance and valuable suggestion throughout the progress of the work.

The author is indebted to Mr. Nazmul Ula, Ph. D., Assistant Professor, Electrical Engineering & Computer Science, Loyola Marymount University for allowing the author to simulate the VHDL code of his design on the *ModelTek* simulator in Mr. Ula's laptop. The author is also grateful to him for his constructive criticisms and valuable suggestions.

The author wishes to thanks Dr. Iftekhar Ahmed, Lecturer, University telecom Malaysia for his help with VHDL based design tools.

ABSTRACT

Full custom design of VLSI circuits is very time consuming and costly. Such a design for a target process cannot be reused for fabrication even in a scaled down version of the same process. This makes the approach less attractive, since the complete chip has to be redesigned for the process. As a result, language based design approach has gained tremendous popularity because of the versatility and portability of such designs. Sophisticated CAD tools are being developed to automate the design procedure of complex integrated circuits.

This thesis presents the VHDL (VHSIC Hardware Description Language) based design of a parallel multiplier of variable operand wordlengths. The multipliers are very easily testable with only 19 vectors irrespective of the operand size. All the single stuck-at faults in the multiplier can be tested with these vectors. The VHDL code for the proposed multiplier can be incorporated into logic synthesis tools for the automatic generation of multiplier macrocells within a few minutes.

CONTENTS

List of Figures

List of Tables

List of Abbreviations

CPLD	Complex Programmable Logic Device
DCVS	Differential Cascode Voltage Switch
FA	Full-Adder
FPGA	Field Programmable Gate Array
LSB	Least Significant Bit
MBE	Modified Booth Encoder
MCA	Manchester Carry Adder
MSB	Most Significant Bit
SC	Selector-Complementer
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration

Chapter 1

Introduction

1.1 Aims

With the continuing advancements of VLSI technologies and marked shrinkage of process features, the need to develop process independent chip design tools is growing [1], [2]. The use of a hardware description language (HDL) for integrated circuit design eliminates the need to worry about process design rules at the design stage [2], [3]. This reduces the design complexity and time required in completing chip designs. This is very important since vendors need to market their products in the shortest possible time in order to capture a major share of the IC market and also to remain competitive. The fact that such high level designs can be implemented on a variety of target processes reduce the design cost as well. The aim of this thesis is to design easily testable parameterizable multipliers. VHSIC Hardware Description Language (VHDL) will be used for the design since it includes some very useful features for hardware design which are not available in other languages [2]-[4]. The use of VHDL will make the design portable and reusable. This chapter presents brief review of the literature on popular multiplication algorithms and their testable implementations. The review also includes a discussion of the multiplier compilers found in the available literature.

1.2 Literature Review

Multipliers are often one of the key elements in single chip digital information processors [5]-[8]. Various algorithms have been developed for multiplication of binary numbers [9]-[17]. Some of them perform unsigned multiplication while others perform two's complement multiplication. The speed of multiplication varies from one algorithm to another. While speed is one of the criteria for selection of multiplication algorithm, the power consumption and regularity of structure are two other very important criteria for VLSI implementation. Especially, for automatic synthesis, the regularity of structure is the most critical factor. The modified Booth algorithm [9], [10] for two's complement multiplication essentially reduces the number of partial products by a factor of two compared to the straightforward carry-save array multiplier [11]. Multiplication speed is almost doubled. Besides, there is no need for precomplementing the multiplier or postcomplementing the product. The multiplier structure is regular, therefore suitable for VLSI implementation.

With the advancement of integrated circuit technology, the implementation of large array multipliers on a single chip has become possible. However, due to the increasing complexity of VLSI circuits it is becoming more and more difficult and costly to test them [18], [19]. As a result, it is a common practice among circuit designers these days to give due consideration to testability at the early stages of design. Extra hardware and/or inputs are added to the original circuits to make them easily testable thereby reducing testing time and cost. The testability of parallel array multipliers have been investigated by several researchers. A number of testable multiplier architectures have been proposed by them [20]-[24]. In [20], C-testable designs of carry-save array multiplier and Baugh-Wooley's two's complement array multiplier are presented. Two designs of easily testable gate-level and DCVS logic multipliers have been proposed in [21]. These designs are based on the straightforward carry-save array multiplication scheme and have been shown to be testable with a constant number of test vectors irrespective of the array size. Such designs are referred to as "C-testable". Gate-level C-testable multipliers based on the modified Booth algorithm have been presented in [22] and [23]. A C-testable DCVS design using this algorithm has also been presented in [24].

In order to find out an optimal area and speed of a processor chip, the different modules within the chip have to be tried out for various architectures. Since it is time consuming to verify many possible layouts for each module, one approach is to use software packages called *module generators* or *silicon compilers* to provide fast and efficient design of parameterized modules. The multiplier compilers presented in [25]-[26] generate parameterizable layouts for MOS technology. The technology independence of the compiler presented in [27] is limited by the requirement that the leaf cells have to be recharacterized in the new technology. The aim of this thesis is to present the design of totally process independent VLSI array multipliers of variable size (parameterizable) using VHDL.

The proposed multiplier is based on the modified Booth algorithm. There is some specific reasons for this particular choice. First of all it reduces the number of partial products to almost half compared to straightforward carry-save array multiplier. Besides, the multiplier has a regular structure which is an extremely important criterion in the selection of schemes for VLSI design. The multipliers are made C-testable [28], i.e., they can be tested for all single stuck-at faults with a constant number of test vectors irrespective of the size of the operands. Although stuck-at fault models cannot adequately model transistor stuck-on and stuck-open faults [29]-[30], it is possible to derive equivalent stuck-at test sets for logic gates to cover transistor stuck-on and stuck-open faults [31]. Since the number of test vectors is constant for any multiplier size, the test generation for the proposed designs are considerably small. In this thesis, VHDL is chosen for designing the testable, parameterizable multipliers because of its unique features for hardware design [2]-[4]. The use of VHDL will make the designs portable and reusable.

1.3 Organization of the Thesis

Chapter 2 presents the parallel multiplication scheme using a straightforward array of carry-save adders. It also introduces the Booth algorithm for multiplication of signed binary numbers. Bit-pair recoding technique and modified Booth multiplier is also presented in this chapter. Chapter 3 analyzes the testability of the multiplier architecture based on modified Booth algorithm and presents the design of a C-testable multiplier. The VHDL model of the parameterizable and easily testable multiplier is presented in chapter 4. Finally, chapter 5 concludes the thesis with some recommendations for further research.

Chapter 2

Multiplier Algorithms and Architecture

2.1 Introduction

In this chapter, multiplication of two fixed point binary operands will be discussed. Some most common parallel multiplication schemes such as the straightforward carry-save array multiplication, Booth algorithm and method of bit-pair recoding or modified booth algorithm will be considered. An architecture based on the modified Booth algorithm for multiplication of two signed numbers will also be presented in this chapter.

2.2 Straightforward Carry-Save Array Multiplication

Multiplication can be defined as repeated addition. The number to be added is the multiplicand, the number of times it is added is the multiplier, and the result is the product. Each step of addition generates a partial product and when the operands are integer the product is twice the length of the operands in order to preserve the information content. Binary multiplication is equivalent to the logical AND operation. Thus the evaluation of partial products consists of the logical ANDing of the multiplicand and the relevant multiplier bit. Each column of partial products must then be added, and if necessary, any carry value passed to the next column. A parallel multiplier [11] is based on the observation that all partial products in the multiplication process may be independently computed in parallel. The partial product terms are called summands. If the multiplicand and the multiplier have m and n bits respectively then there will be m x n summands, which are produced by a set of mn AND gates. In a straightforward carry-save array multiplier the summands are collected through a cascaded array of carry-save adders. At the bottom of the array, an adder is used to convert the "carry save form" to the required form of output. The depth of the array and the carry propagation characteristics of the adder fix the multiplication time.

A 4 x 4 bit straightforward carry save array multiplier with the partial products enumerated [11] is shown in Fig. 2.1. The basic cell that may be used to construct this parallel multiplier is also shown in this figure. The multiplicand term x_I is propagated vertically, while the multiplier term y_I is propagated horizontally. Incoming partial product bits enter at the top and the incoming CARRY IN bits enter at the top right of the cell. The bit-wise AND operation is performed in the cell, and the SUM is passed to the next cell at the lower right. The CARRY OUT is passed to the bottom of the cell.

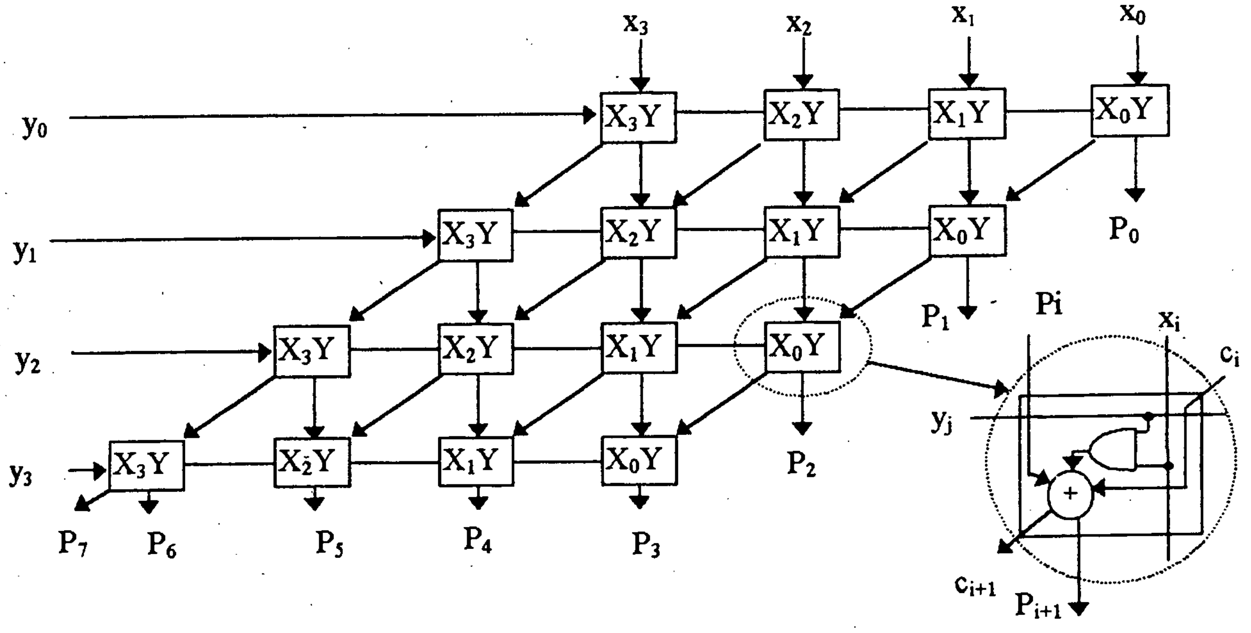


Fig. 2.1 A parallel multiplier array using carry save adders

2.3 Booth Algorithm

Booth Algorithm is a powerful direct tool for signed-number multiplication [10]. In the standard add-shift method, each non zero bit of the multiplier causes one addition of the multiple of multiplicand to the partial product. The execution time of multiplication instruction is determined mainly by the number of additions to be performed. So, the execution time can be reduced if we can reduce the number of additions. This is achieved by a method of bit-scanning which reduces the number of multiplicand multiples. This technique uses recoding of the multiplier based on the string property. The process is often referred to as "skipping over 0s" and can be generalized to shift of variable lengths if string of 0s can be detected. The greater the number of 0s in the multiplier the faster the operation. Consider a string of k consecutive 1s in the multiplier as shown below.

$$\begin{array}{l} \ldots\ldots, i+k, i+k-1, i+k-2, \ldots\ldots, i, i-1, \ldots\ldots \\ \ldots\ldots, \ 0\ , \underbrace{1\ , \ 1\ , \ \ , 1}_{\text{k consecutive 1s}}, \ 0\ , \ldots\ldots \end{array}$$

by using the following property of binary strings

$$2^{i+k} - 2^{i} = 2^{i+k-1} + 2^{i+k-2} + \ldots\ldots + 2^{i+1} + 2^{i} \qquad (2.1)$$

The consecutive 1s can be replaced by the following string

$$\begin{array}{l} \ldots\ldots, i+k+1, i+k, i+k-1, \ldots\ldots, i+1, \ i, i-1, \ldots\ldots \\ \ldots\ldots, \ 0\ , \underset{\substack{\uparrow \\ \text{Addition}}}{1}\ , \underbrace{0\ , \ldots\ldots, \ 0}_{\text{k-1 consecutive 0s}}\ , \underset{\substack{\uparrow \\ \text{Subtraction}}}{-1}, \ 0\ , \ldots\ldots \end{array}$$

Now consider a multiplication example in which a positive multiplier has a single block of 1s with at least one 0 at each end, for example 0 0 1 1 1 0 (14). The number of addition can be reduced by observing that a multiplier in this form can be regarded as the difference of two numbers as follows:

```
   0 1 0 0 0 0  (16)
-) 0 0 0 0 1 0  ( 2)
   -----------  -----
   0 0 1 1 1 0  (14)
```

This was shown in Eq. 2.1 and indicates that the product can be generated by one addition (addition of 2^4) and one subtraction (subtraction of 2^1). In the standard notation, the multiplier can be written as

$$0 \quad 0 \quad +1 \quad +1 \quad +1 \quad 0$$

and the recoded multiplier can be written as

$$0 \quad +1 \quad 0 \quad 0 \quad 0 \quad -1 \quad 0$$

Note that the -1 times the left-shifted multiplicand occurs at 0 to 1 boundaries and +1 times the left-shifted multiplicand occurs at 1 to 0 boundaries as the multiplier is scanned from right to left. The transformation that takes

0 1 1 1 1 ....1 1 1 0   into   +1 0 0 0 0 ....0 -1 0

is often referred as the technique of skipping over 1s. The reasoning is that in cases in which the multiplier has its 1s grouped into a few blocks, only a few versions of the multiplicand need to be added to generate the product hence, the multiplication process becomes much faster. It can also be shown that the Booth recoded multiplier algorithm works equally well for negative multiplier.

2.4 Modified Booth Algorithm

Modified Booth Algorithm is a multiplication speedup technique that guarantees that an n-bit multiplier will generate at most n/2 partial products [9], [10]. It can multiply two two's complement numbers directly and gives the product also in two's complement form. This represents a multiplication speed increase of almost a factor of 2 over the standard add-shift method.

This new technique is derived from the Booth technique. Recall from the previous discussion of a positive multiplier of 0 0 1 1 1 0 (+14). The number of addition can be reduced by observing that the multiplier in this form can be regarded as the difference of two numbers as shown below.

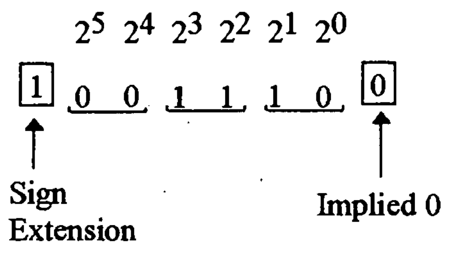
```
                $2^5$ $2^4$ $2^3$ $2^2$ $2^1$ $2^0$
                 0  1  0  0  0  0    (16)
              -) 0  0  0  0  1  0    ( 2)
                 ----------------    -----
Multiplier →     0  0  1  1  1  0    (14)
```

This indicate that the number 0 0 1 1 1 0 (14) has the same value as

$$2^4 - 2^1 = 16 - 2 = 14$$

This is true for any number of contiguous 1s, including the case in which there is a single 1 with 0s on either side. The entire concept of bit-pair recoding revolves around this method of regarding a string of 1s as the difference of two numbers.

Now returning to the multiplier being discussed and scanning it from right to left, bit by bit. In going from 0 (2^0) to 1 (2^1), we saw previously that this resulted in subtracting the value of the 1 in that position, in this case - 2^1. Scanning from 1 (2^1) to 1 (2^2) resulted in no change, that is, neither addition nor subtraction. The same is true in scanning from 1 (2^2) to 1 (2^3). However, in going from 1 (2^3) to 0 (2^4), we saw that this resulted in an addition of 2^4. There is no change in scanning from 0 (2^4) to 0 (2^5). The results of scanning this multiplier are as follows: 2^1 was subtracted and 2^4 was added. The same results can be obtained by looking at pairs of bits in the multiplier in conjunction with the bit that is to the right of the bit pair being considered, as shown below.



That is, bit pair 2^1, 2^0 is examined with an implied 0 to the right of the low-order bit; bit pair 2^3, 2^2 is examined with bit 2^1, bit pair 2^5, 2^4 is examined with bit 2^3. Scanning the bit pairs from right to left and using the rightmost bit of each pair as the column reference for the partial product placement (it is the center bit of the three bits being examined), we obtain the following multiplier bit-pair recoding scheme shown in table 2.1. It should be noted that there are a total of eight possible versions of the multiplicand.

Table 2.1. Multiplier bit-pair recoding scheme

Multiplier bit-pair		Multiplier bit on the right	Multiplicand multiples to be added	Explanation
i+1	i	i-1		
0	0	0	0 × multiplicand	No string
0	0	1	+ 1 × multiplicand	End of string
0	1	0	+ 1 × multiplicand	Single 1 (+2 -1)
0	1	1	+ 2 × multiplicand	End of string
1	0	0	- 2 × multiplicand	Beginning of string
1	0	1	- 1 × multiplicand	End/beginning of string
1	1	0	-1 × multiplicand	Beginning of string
1	1	1	0 × multiplicand	Strings of 1s

Fig. 2.2 gives an example of the bit-pair recoding multiplication technique using two 5 bit operands represented in two's complement form.

Multiplicand X = 0 0 1 1 0 (+ 6)

Multiplier Y = $\underbrace{[1]\ 1}_{-1x}\ \underbrace{0\ 0}_{+1x}\ \underbrace{1\ 0}_{-2x}\,[0]$ (- 14)

1 1 1 1 1 1 0 1 0 0

0 0 0 0 0 1 1 0

1 1 1 0 1 0

Product P = 1$\leftarrow$ 1 1 1 0 1 0 1 1 0 0 (- 84)

Fig. 2.2 Multiplication example using bit-pair recoding

2.5 Removal of Sign-bit Extension Circuitry

The modified Booth algorithm for multiplying two binary numbers basically consists of two steps. First to obtain the partial product from the proper version of the multiplicand and second to add these partial products in an appropriate array of full adders considering that summation in an array has to be done with sign bit extension, because it is a signed multiplication. However, if explicit sign extension scheme is observed large amount of circuitry is required merely to accommodate the sign-extension of the partial products. The redundancy of the sign-bit extension can be eliminated by a simple method, i.e., reducing the number of variable inputs to the array, thus reducing the number of full adders involved. Several approaches for removing the sign-extension circuitry from Booth multiplier have been proposed by previous researchers [32], [33].

Let us consider the multiplication of two 8-bit binary numbers using modified Booth algorithm. Since this algorithm scans three bits of the multiplier at a time and retires two of them to generate a partial product, the total number of partial products generated for the 8-bit multiplier is four. If a, b, c, d represents these partial products, then the addition of these partial product is illustrated in Fig. 2.3. Each partial product is shifted two bit positions to the left with respect to the preceding one in accordance with the modified Booth algorithm.

a8	*a8*	*a8*	*a8*	*a8*	*a8*	*a8*	*a8*	*a7*	*a6*	*a5*	*a4*	*a3*	*a2*	*a1*	*a0*
b8	*b8*	*b8*	*b8*	*b8*	*b8*	*b7*	*b6*	*b5*	*b4*	*b3*	*b2*	*b1*	*b0*		
c8	*c8*	*c8*	*c8*	*c7*	*c6*	*c5*	*c4*	*c3*	*c2*	*c1*	*c0*				
d8	*d8*	*d7*	*d6*	*d5*	*d4*	*d3*	*d2*	*d1*	*d0*						
P15	*P14*	*P13*	*P12*	*P11*	*P10*	*P9*	*P8*	*P7*	*P6*	*P5*	*P4*	*P3*	*P2*	*P1*	*P0*

Fig 2.3 Sign extended partial product array

Here a_8, b_8, c_8, d_8 are the sign bits. It is seen that the direct implementation of explicit sign extended array will be an uneconomical choice.

Let us assume for simplicity that the arithmetic weight of the p_8 column is 2^0, i.e., 1. Thus the p_{15} column represents a weight of 2^7. Then the sum of the sign and sign extended bits can be written as

$$\begin{aligned}\text{Sum} &= a_8(2^7+2^6+2^5+2^4+2^3+2^2+2^1+2^0)+b_8(2^7+2^6+2^5+2^4+2^3\\&\quad+2^2)+c_8(2^7+2^6+2^5+2^4)+d_8(2^7+2^6)\\&= a_8(2^8-2^0)+b_8(2^8-2^2)+c_8(2^8-2^4)+d_8(2^8-2^6)\end{aligned}$$

Since p_{15} is the most significant bit of the product output, module 2^8 addition can be used to sum the sign bits. Thus, the sum of the sign bits can be written as

$$\text{Sum} = -a_8(2^0)-b_8(2^2)-c_8(2^4)-d_8(2^6)$$

which expressed as a binary number is

$$\text{Sum} = -(0\ d_8\ 0\ c_8\ 0\ b_8\ 0\ a_8) \qquad \textbf{(2.2)}$$

The two's complement of the word $(0\ d_8\ 0\ c_8\ 0\ b_8\ 0\ a_8)$ is

$$-(0\ d_8\ 0\ c_8\ 0\ b_8\ 0\ a_8) = (1\ \bar{d}_8\ 1\ \bar{c}_8\ 1\ \bar{b}_8\ 1\ \bar{a}_8)+1 \qquad \textbf{(2.3)}$$

When the recoding scheme of Eq. 2.3 is used, the sign extended Booth partial product array appears like the one shown in Fig. 2.4.

$$\begin{array}{cccccccccccccccc}
 & & & & & & & 1 & & & & & & & & \\
 & & & & & & 1 & \overline{a_8} & a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 \\
 & & & & 1 & \overline{b_8} & b_7 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 & & \\
 & & 1 & \overline{c_8} & c_7 & c_6 & c_5 & c_4 & c_3 & c_2 & c_1 & c_0 & & & & \\
1 & \overline{d_8} & d_7 & d_6 & d_5 & d_4 & d_3 & d_2 & d_1 & d_0 & & & & & & \\
\hline
p_{15} & p_{14} & p_{13} & p_{12} & p_{11} & p_{10} & p_9 & p_8 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}$$

Fig 2.4 Recoded sign extended partial product array

Hence it is seen that elimination of the sign-extension circuitry in a modified Booth algorithm multipliers can be achieved by inverting the MSB of each partial product and adding a logic '1' at every higher significance (including the MSBs). This procedure is equivalent to recoding the MSBs of the partial products as a two's complement number and adding a logic '1' to the most significant full adder in each row of the main array.

2.6 An Architecture Based on Modified Booth Algorithm

Fig. 2.5 represents an 8 by 8 bit multiplier architecture based on the modified Booth algorithm for multiplication of two binary numbers that are in two's complement form [24]. Elimination of the sign extension circuitry is achieved by the procedure described above. The modified Booth encoder (MBE) block in each row operates on three multiplier bits to generate the control signals CM, K_1 and K_2 according to the modified Booth recoding scheme as shown in Table 2.2. In this recoding scheme five possible partial products can be formed: 0, +X, -X, +2X, -2X where X denotes the multiplicand. The selector complementers (SC) in Fig. 2.5 consist of multiplexers which operate on the multiplicand bits to generate 0, X or 2X as partial products depending on the control signals K_1, K_2 and complementers (2-input EX-OR gates) which generate one's complements of these partial products only when CM signal is high. Moreover, these one's complemented partial products are converted to their two's complement form by addition of a logic '1' to their LSBs. The addition of the partial products are accomplished by an array of carry save full adders (FA). The Manchester carry adders (MCA) on the right-hand side and the bottom of the Fig. 2.5 operates on the results coming out of the main array (the array containing SCs and FAs) to generate the final product output.

Table 2.2: Modified Booth recoding table

MBE inputs			MBE outputs			Partial Product	SC output
Y_{i+1}	Y_i	Y_{i-1}	K_1	K_2	CM	Generated	Z
0	0	0	0	0	0	0	0
0	0	1	1	0	0	+X	X_i
0	1	0	1	0	0	+X	X_i
0	1	1	0	1	0	+2X	X_{i-1}
1	0	0	0	1	1	-2X	$\bar{X}_{i-1}$
1	0	1	1	0	1	-X	$\bar{X}_i$
1	1	0	1	0	1	-X	$\bar{X}_i$
1	1	1	0	0	0	0	0

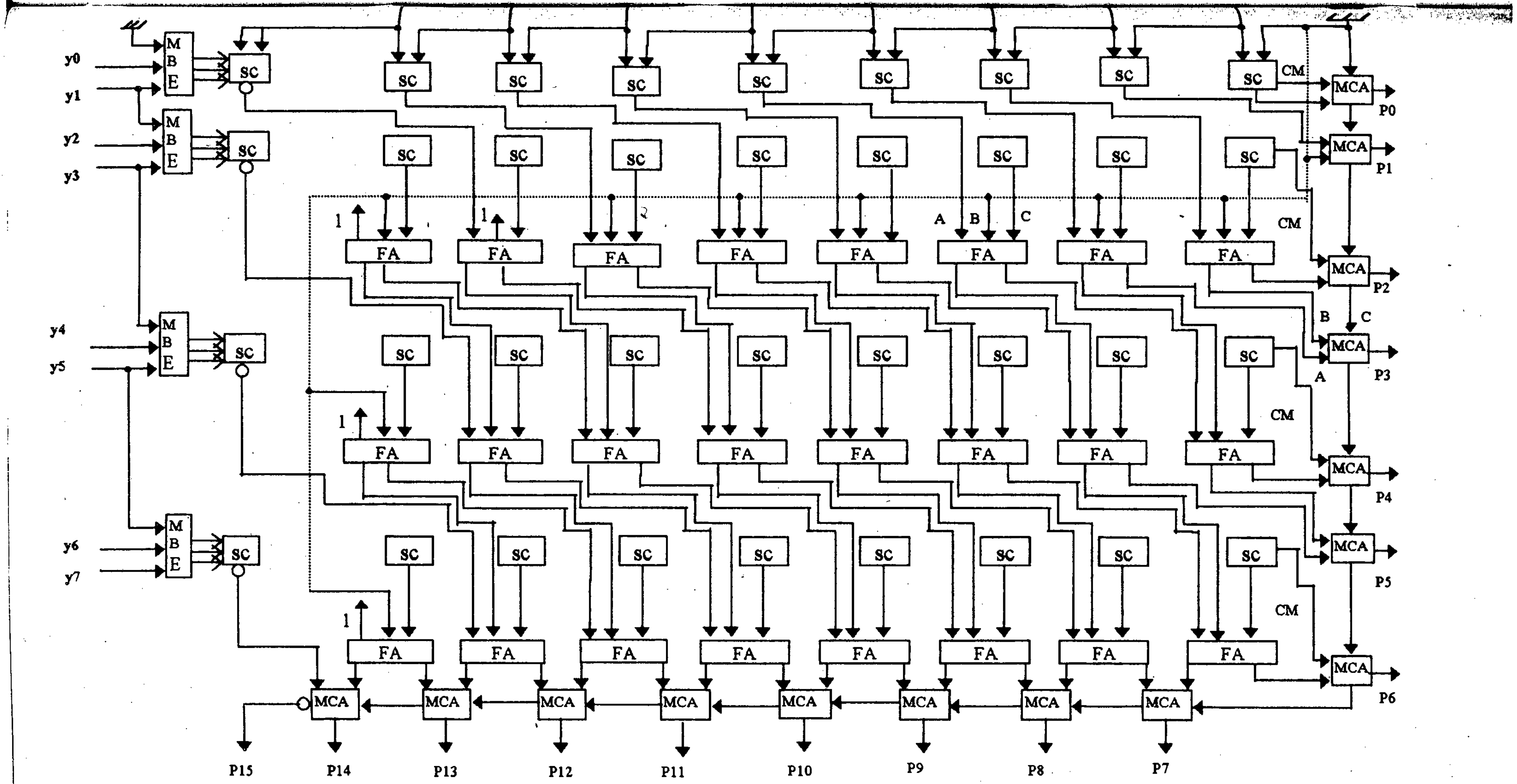


Fig 2.5 : An 8 by 8 bit modified Booth multiplier arrray

(Horizontal Controls and vertical multiplicand routings are omitted for clarity)

Chapter 3

Testability of the Multiplier

3.1 Introduction

With the increasing complexity of VLSI circuits, it is difficult to test them unless due consideration to testability is given at the early stages of design. The high device-to-pin ratio severely limits the controllability of internal signal lines in VLSI chip [19]. Also, there exists a large number of faults of various types, many of which cannot be modelled by the traditional stuck-at fault model. Test pattern generation and verification procedures are becoming very costly or even computationally infeasible to implement [18]. However, VLSI circuits like array multipliers having regular iterative structure have been shown to be easily testable by slight modification of the conventional design [20]. The multiplier architecture presented in Chapter 2 will be modified in this chapter in order to convert it to an easily testable one.

3.2 Testing Approach

The objective of the testing approach adopted in this research is to exhaustively test the full-adders (FAs), Manchester carry adders (MCAs) and modified Booth encoders (MBEs). Such a test set will be applicable to any arbitrary logic implementation of these cells. The fault model used in this research assumes:

a) at most one basic cell in an array multiplier is faulty at a time;

b) the fault is a permanent fault (i.e. the fault permanently changes the circuit's logic characteristics);

c) the fault may alter the cell's output functions in any arbitrary way, as long as the faulty cell remains combinational circuit.

It is necessary to modify the design of the modified Booth encoders with a significant increase in complexity and gate count in order to generate exhaustive test set for the selector-complementers (SCs). However, Takach and Jha [21] have shown that hardware overhead reduction is possible for array multipliers if a fault model based on single (stuck-at) faults is used instead of the single cell fault model. They have also shown that a set of test vectors which detect all single stuck-at faults in a gate level carry-save multiplier can be readily adopted to detect all detectable single stuck-at, transistor stuck-on and stuck-open faults in a DCVS implementation of the multiplier. Therefore, the selector-complementers will be tested for single stuck-at faults only. Moreover, although MBEs are eventually exhaustively tested, this testing does not guarantee the fault propagation to the primary outputs of the array. Due to this, equivalent gate level circuit for MBE will be tested for single stuck-at faults.

3.3 Modification of the Architecture for Testability

The main challenge in testing array multipliers is the difficulty of controlling the inputs of internal adder cells from the primary inputs, namely the multiplier (Y) and multiplicand (X) inputs. In fact, some patterns cannot be applied to some adders cells. To overcome this problem, extra inputs and sometimes extra hardware is added to enhance controllability and observability of the internal signal lines in VLSI circuits.

A testable architecture for an 6×8 bit multiplier is shown in Fig. 3.1. Comparing to its non testable version, this architecture has 4 extra controllable inputs e_1, e_2, e_3, e_4,

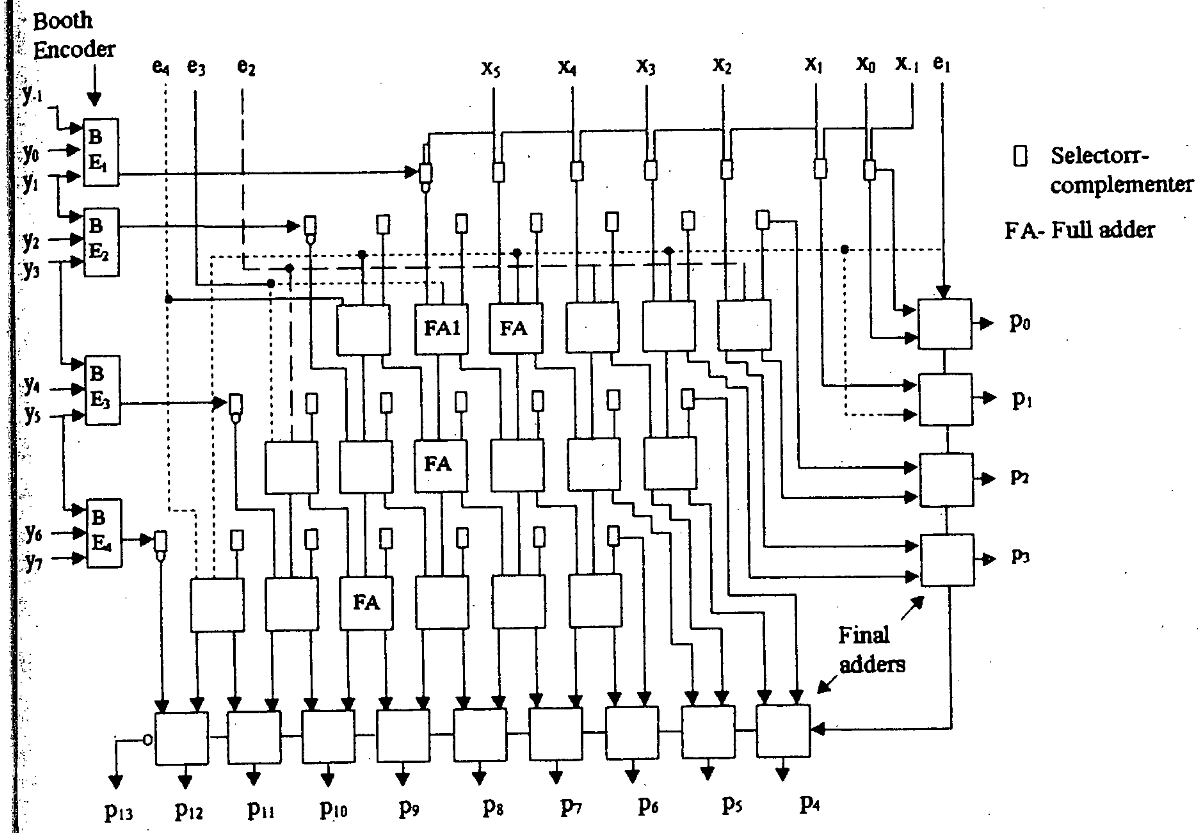


Fig. 3.1 Architecture of the multiplier with recoded sign bits

x_{-1} and y_{-1} to enhance the controllability of various cells. For normal multiplication operation these extra inputs will have the following logic values: $e_1 = 0$, $e_2 = 0$, $e_3 = 1$, $e_4 = 1$, $x_{-1} = 0$ and $y_{-1} = 0$.

3.4 Testing the Individual Cells

In this section, the patterns required for testing the various individual cells of the multiplier for single stuck-at faults are derived.

3.4.1 Testing of MBEs for Single Stuck-at Fault

The logic diagram of the modified Booth encoder used in the multiplier is shown in Fig. 3.2.

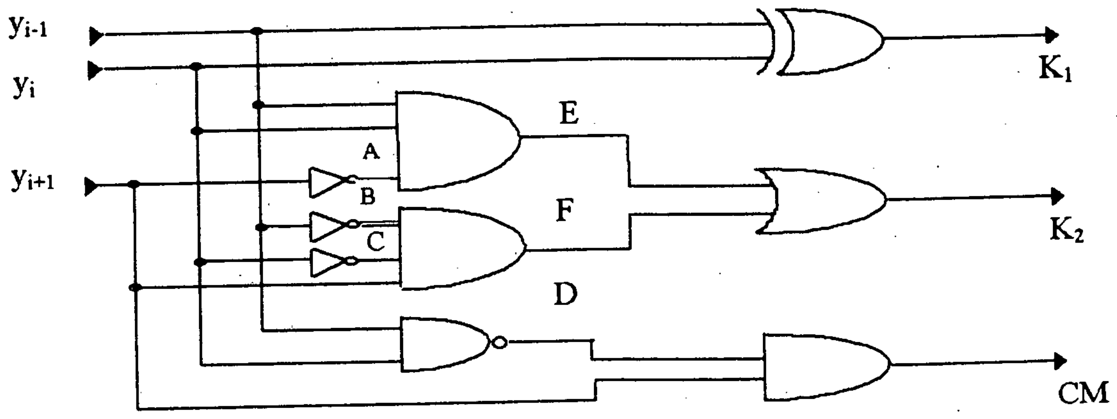


Figure: 3.2 Gate level design of the Modified Booth Encoder (MBE)

The circuit has twelve nodes and so twenty four possible stuck-at faults. For the three primary inputs there will be eight possible test vectors which will be identified as t_0 to t_7, where the suffix is the decimal equivalent of the binary numbers ($y_{i-1}y_iy_{i+1}$). The fault coverage is conveniently displayed in the fault-matrix shown in Table 3.1. The tick against each test indicates the fault covered by that test.

Table 3.1 Fault matrix for the MBE logic circuit

Test	y_{i-1}	y_{i-1}	y_i	y_i	y_{i+1}	y_{i+1}	A	A	B	B	C	C	D	D	E	E	F	F	K_1	K_1	K_2	K_2	CM	CM
	/0	/1	/0	/1	/1	/1	/0	/1	/0	/1	/0	/1	/0	/1	/0	/1	/0	/1	/0	/1	/0	/1	/0	/1
t_0		√		√		√										√		√		√		√		√
t_1		√		√	√				√		√		√				√			√	√		√	
t_2		√	√			√										√		√	√			√		√
t_3		√	√		√							√	√			√		√	√			√	√	
t_4	√			√		√				√						√		√	√					√
t_5	√			√	√					√			√			√		√	√			√	√	
t_6	√		√				√								√					√	√			√
t_7	√		√					√						√		√		√		√		√		√

From the above fault matrix it is seen that the test vectors t_1, t_3, t_6 and t_7 are the essential tests. These four essential tests cover all the faults except $y_{I+1}/1$ and B/1. A single test that covers both of these faults is t_4. Hence a set of test patterns for the inputs $(y_{I-1}y_Iy_{I+1})$ of the MBE of Fig. 3.2 that detect any single stuck-at fault in the MBE is {001, 011, 100, 110, 111}.

3.4.2 Testing of the SCs for Single Stuck-at Fault

The logic diagram of the selector –complementer block is shown in Fig. 3.3. It has a total of five inputs. However, for testing of single stuck-at fault we will derive fault matrix for only the selector part. This is because the complementer part is nothing but an EX-OR gate whose one input is the complement signal CM and the other is output of selector circuit Z_i. Since output of an EX-OR gate inverts due to inversion of any one of its inputs so if we can test only the selector part for single stuck-at fault we may declare that this fault will propagate to the SC output due to that fault propagation property of EX-OR gate. This criterion will also reduce the number of input test vectors of SC blocks from twenty five to sixteen. These will be identified as t_0 to t_{15}, where suffix is the decimal equivalent of the binary number ($K_1x_iK_2x_{I-1}$). Table 3.2 shows the fault matrix.

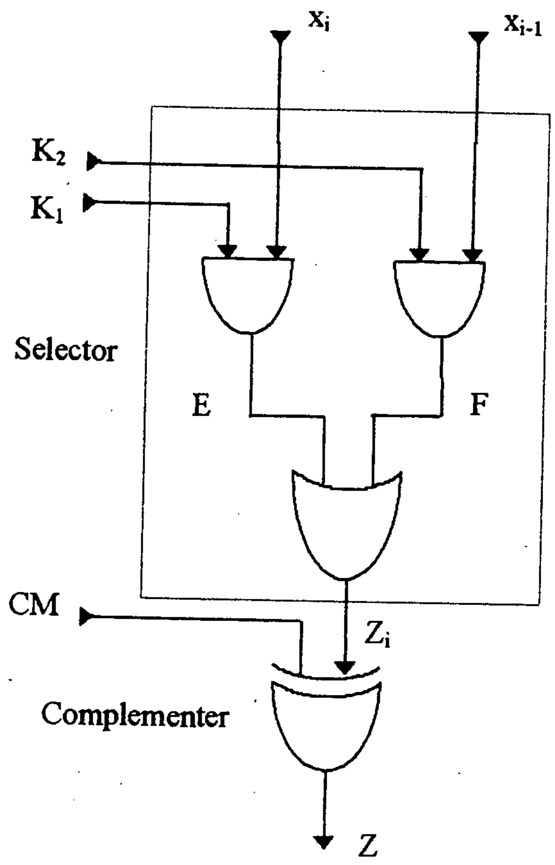


Fig. 3.3 Gate Level Design of the Selector-Complementer Block

Table 3.2 Fault Matrix for the Selector Block

Test	$K_1/0$	$K_1/1$	$K_2/0$	$K_2/1$	$x_i/0$	$x_i/1$	$x_{i-1}/0$	$x_{i-1}/1$	E/0	E/1	F/0	F/1	$Z_i/0$	$Z_i/1$
t_0										√		√		√
t_1				√						√		√		√
t_2								√		√		√		√
t_3			√				√			√	√		√	
t_4		√								√		√		√
t_5		√		√						√		√		√
t_6		√						√		√	√	√		√
t_7		√	√				√			√			√	
t_8						√				√		√		√
t_9				√		√				√		√		√
t_{10}						√		√		√		√		√
t_{11}			√			√	√			√	√		√	
t_{12}	√				√				√			√	√	
t_{13}	√			√	√				√			√	√	
t_{14}	√				√			√	√			√	√	
t_{15}	√		√		√		√		√		√		√	

Identifying the indistinguishable faults and dominant faults in the fault matrix of Table 3.2, it is found that the test vectors needed to test any single stuck-at fault are t_2, t_7, t_8 and t_{13}. So the set of test pattern for the inputs ($K_1x_iK_2x_{I-1}$) of the selector circuit to detect any single stuck-at fault is {0010, 0111, 1000, 1101}.

3.5 Testing the Multiplier

In this section, a set of test vectors for testing the multiplier will be derived. The vectors will cover the exhaustive testing of the FAs, MCAs, MBEs as well as the stuck-at faults in the selector-complementers.

3.5.1 Test Vectors

A set of test vectors for detecting all single stuck-at faults in a larger version of the multiplier of Fig. 3.1 is shown in Table 3.3. An 8-bit multiplicand X and a 8-bit multiplier Y are shown with their LSBs to the right most position. The underlined bits have to be replicated for generating the test vectors for multipliers with larger operand wordlenths.

3.5.2 Exhaustive Testing of the Full-Adders

The first twelve test vectors t_1-t_{12} of Table 3.3 set up the patterns required for exhaustive testing of all the full-adders as explained in the following steps:

1) The test vector t_1 applies 000 to most of the full-adders. However, the full adders affected by the inverted sign bits of the partial products receive 100. Test vector t_2 applies 000 to these full-adders.

2) Application of pattern 111 to all the full-adders is accomplished with the vectors t_3 and t_4.

3) The vector t_5 applies 100 to all the full-adders except the one labeled 'FA1' in the second row of Fig. 3.1 which receives the pattern 010. t_6 applies 100 to FA1.

4) The vector t_7 applies 011 to all the full-adders except the one labeled 'FA1' which receives the pattern 101. t_8 applies 011 to FA1.

Table 3.3 A set of test vectors for an 8 ×8 bit multiplier

Vectors	X	x_{-1}	Y	y_{-1}	$e_4e_3e_2e_1$
t_1	0000 000$\underline{0}$	0	$\underline{0}$000 0000	0	0000
t_2	1000 000$\underline{0}$	0	$\underline{01}$01 0101	0	0000
t_3	1111 111$\underline{1}$	1	$\underline{01}$01 0101	0	1111
t_4	1000 000$\underline{0}$	0	$\underline{10}$10 1010	1	1111
t_5	0000 000$\underline{0}$	0	$\underline{1}$111 1111	0	1100
t_6	0000 000$\underline{0}$	0	$\underline{0011}$ 0011	1	0011
t_7	1111 111$\underline{1}$	1	$\underline{01}$01 0011	0	0011
t_8	1111 111$\underline{1}$	1	$\underline{0100}$ 0100	1	1100
t_9	0101 01$\underline{01}$	0	$\underline{1100}$ 1100	1	0110
t_{10}	010101$\underline{01}$	0	$\underline{0011}$ 0011	0	1001
t_{11}	1111 111$\underline{1}$	1	$\underline{0011}$ 0011	0	0011
t_{12}	1111 111$\underline{1}$	1	$\underline{1100}$ 1100	1	1100
t_{13}	0000 000$\underline{0}$	0	$\underline{10}$10 1010	1	1100
t_{14}	1111 111$\underline{1}$	1	$\underline{10}$10 1010	1	0000
t_{15}	111 1111$\underline{1}$	0	$\underline{1001}$ 1001	1	1111
t_{16}	111 1111$\underline{1}$	0	$\underline{0110}$ 0110	0	0000
t_{17}	0000 000$\underline{0}$	0	$\underline{0110}$ 0110	0	1111
t_{18}	1111 111$\underline{1}$	1	$\underline{1001}$ 1001	1	1001
t_{19}	1111 111$\underline{1}$	1	$\underline{1}$111 1111	1	0000

* The bits to be replicated for larger multipliers are underlined

5) t_9 applies 001 and 110 to alternate full-adders, t_{10} applies 110 and 001 to alternate full-adders in each row.

6) t_6 applies 010 to the full-adders in the even rows except FA1. It was seen in step 3 that FA1 gets 010 by t_5. Application of 010 to the full-adders in the odd rows is accomplished with the test vector t_{11}.

7) t_8 applies 101 to the full-adders in the even rows except FA1. It was seen in step 4 that FA1 gets 101 by t_7. Application of 101 to the full-adders in the odd rows is accomplished with the test vector t_{12}.

Table 3.4 shows the results of exhaustive testing of the full-adders.

Table 3.4 Exhaustive testing of the FAs

Pattern applied to FAs	Test vector required
000	t_1, t_2
111	t_3, t_4
100	t_5, t_6
011	t_7, t_8
001	t_9, t_{10}
110	t_9, t_{10}
010	t_5, t_6, t_{11}
101	t_7, t_8, t_{12}

Now let us consider how the effect(s) of a fault in a full-adder is transmitted to the primary outputs (observable outputs) of the multiplier. The sum output of a full-adder is the parity (EX-OR) of the three input bits. Therefore, if one of these three inputs is inverted due to appearance of a faulty signal then the sum output of the full-adder is also inverted. The carry output may or may not be inverted depending on the logic values of the other two inputs. This is also true for manchester carry adder (MCA), because it realizes the same logic function as a full-adder. Since all the full-adders are exhaustively tested, the effect of a fault in a full-adder is transmitted to it's output(s). The two outputs of each full-adder of Fig. 3.1 are connected to the primary outputs of the multiplier through two different chains of three input EX-OR gates (of FAs and final MCAs). Hence the effect of a fault in a full-adder is transmitted to the observable output(s).

3.5.3 Exhaustive Testing of the Manchester Carry Adders

All the manchester carry adders are exhaustively tested using a subset of test vectors from Table 3.3. The combinations of test vectors that apply various patterns to all the manchester carry adders are listed in Table 3.5. The effect of a fault in any MCA is transmitted to it's sum output which is a primary output of the multiplier.

Table 3.5 Exhaustive testing of the MCAs

Pattern applied to MCAs	Test vector required
000	t_1, t_2
111	t_3, t_4
101	t_3, t_5, t_7
011	t_7, t_{13}
010	t_6, t_8, t_{14}
100	t_6, t_8
001	t_9, t_{10}, t_{11},t_{12}, t_{15}, t_{16}
110	t_9, t_{10}, t_{11},t_{12}, t_{15}, t_{16}

3.5.4 Testing of the Modified Booth Encoders

The modified Booth encoders are tested in two ways. First they are exhaustively tested However unlike the FAs and MCAs, exhaustive testing of the MBEs does not necessarily guarantee the transmission of the effect of a faults in an MBE to the primary outputs of the multiplier. That is why the MBE block is also tested for single stuck-at faults.

3.5.4.1 Exhaustive Testing

The modified Booth Encoders are exhaustively tested by the vectors shown in Table 3.6.

Table 3.6 Exhaustive testing of the MBEs

Pattern applied to MBEs	Test vector required
000	t_1
010	t_2
101	t_4
111	t_5, t_6
011	t_{11}, t_{12}
100	t_{11}, t_{12}
001	t_{15}, t_{16}
110	t_{15}, t_{16}

3.5.4.2 Testing for Single Stuck-at Fault

Unlike the full-adders and manchester carry adders, exhaustive testing of the MBEs does not necessarily guarantee the transmission of the effect of a fault in an MBE to the primary output of the multiplier. Fault propagation depends on the type of fault and its effect on the output of the MBE. Also, note from Figures 3.1 and 3.2 that the output of an MBE in any row, namely CM, K_1, and K_2 are inputs to the selector-complementers (SC) in that row (fan-out nodes). In the rest of this sub-section, the test vectors which propagates any single stuck-at fault in an MBE to the primary outputs of the multiplier will be derived.

It was shown in Section 3.4.1 that a set of test patterns for the inputs $(y_{i-1}y_iy_{i+1})$ of the MBE of Fig. 3.2 that detect any single stuck-at fault in the MBE is {001, 011, 100, 110, 111}. Note from Fig. 3.2 that every input to the MBE has a fan-out of three. This means that the effect of a fault at one of the input nodes might propagate to more than one output of the faulty MBE. Because of the fan-outs at the outputs of the MBE these faulty signals from MBE might propagate through two different paths, i.e. the SC and the chains of carry-save adders, and then reconverge at the final adders (MCAs). Also, note from Fig. 3.1 that the adjacent MBEs share one multiplier bit. Therefore, a fault on one of the shared multiplier bits might affect both the MBE sharing that bit resulting in transmission of the fault through both the MBEs and subsequent reconvergence in the array of FAs and MCAs. It can be verified that because of these reasons some path sensitive patterns (mentioned above) applied to the MBEs for detecting some single stuck-at faults result in negative reconvergence [18] of the fault effects unless these patterns are accompanied by application of appropriate patterns to the multiplicand. It was extensively verified in this research that the test vectors t_{11}, t_{12}, t_{15}, t_{16} and t_{19} apply all the necessary patterns to all the MBEs along with appropriate multiplicand patterns so that any single stuck-at fault in the MBE is propagated to the primary outputs of the multiplier.

3.5.5 Test Vectors for SCs

As mentioned in Section 3.4.2, a set of test patterns for the inputs ($K_1x_iK_2x_{i-1}$) of the selector block of Fig. 3.3 that detect any single stuck-at fault in the selector is {0010, 0111, 1000, 1101}. The selector in any row have two common input nodes, namely K_1, and K_2 (fan-out nodes). It is verified that the vectors t_2, t_3, t_5, t_{17} and t_{18} sensitize the single stuck-at faults at these nodes and propagate them to the output of the selectors. Each of this selector output is passed though a complementer (an EX-OR gate) whose other input is the MBE output CM (complement signal). The effect of a single stuck-at fault at one of the multiplicand bits is propagated through the outputs of the SCs in that column to one of the inputs of the full-adders in that column. Now the sum output or both the sum and carry outputs of a full-adder are inverted due to a faulty input signal. Each output of every full-adder is connected to the primary outputs of the multiplier through a chain of 3-input EX-OR circuits (of FAs and final MCAs). Thus, these faults are essentially propagated to the primary outputs of the multiplier.

3.6 Calculation of Overhead

3.6.1 Hardware Overhead

The testable design of the multiplier presented earlier does not require any extra logic compared to the original design presented in Chapter 2. However, the testable version requires four extra inputs which increases the number of input pins of the multiplier chip. For a large multiplier, e.g., 32×32 bit multiplier the penalty in terms of extra pins will not be as severe as for a multiplier of small operand wordlengths, e.g., 8×8 bit. The lines carrying the above extra signals will increase the silicon area of the testable multiplier chip compared to the non-testable design.

3.6.2 Delay Overhead

Compared to the non-testable design, the testable multiplier will not have an extra logic gate delay. However, there will be some additional delay due to the extra wiring capacitances associated with various nodes connected to the extra inputs.

3.7 Summary

The testable design of the multiplier presented in this chapter requires only test vectors to test all single stuck-at faults. Also, the full-adders, Manchester carry adders and the modified Booth encoders are exhaustively tested. The numbers of test vectors for any larger multiplier will still remain the same, i.e., 19. Therefore, this testable multiplier can be said to be C-testable [28].

Chapter 4

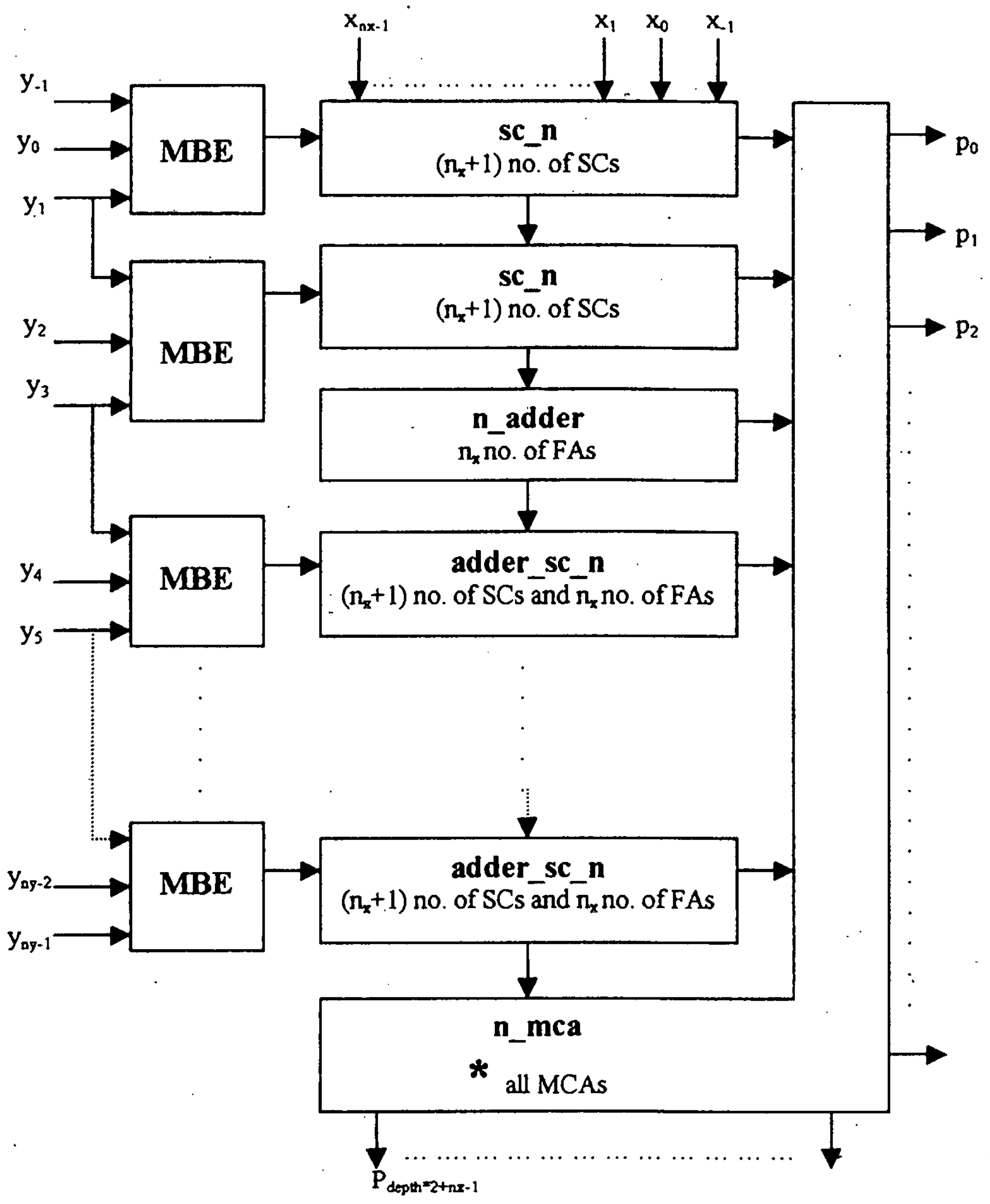
VHDL Modeling of the Multiplier

4.1 Introduction

Design of VLSI circuits entirely at custom level is very time consuming and costly. Such a design for a target process cannot be reused for fabrication even in a scaled down version of the same process. So, to get versatility and portability, high level languages are being used to design process independent VLSI circuits. One of the recent methodology is the introduction of VHDL (VHSIC Hardware Description Language). It not only allows the design of process independent VLSI circuits, but also the automation of the complete design process. In this chapter, the design of a parameterizable and easily testable multiplier is presented using VHDL. Because of the use of VHDL, it will be possible to generate multiplier layouts for various target processes and also program FPGAs (Field Programmable Gate Arrays), CPLDs (Complex Programmable Logic Devices) from various vendors. The capabilities of VHDL to generate arbitrary n-bit modules is exploited in this chapter in designing the parameterizable multiplier.

4.2 Partitioning

The first step toward developing a VHDL code is to partition the design into simpler modular blocks. The multiplier architecture presented in Fig. 3.1 has a regular structure which is very convenient for high level coding. Only four different types of basic cells, viz., modified Booth encoder (MBE), selector-complementer (SC), full-adder



* no. of MCAs = depth*2 + $n_x - 1$
where depth = $n_y/2$ for even n_y
$(n_y+1)/2$ for odd n_y
n_x = no. of bits in the multiplicand
n_y = no. of bits in the multiplier
The additional test inputs (e_1, e_2, e_3, e_4) are not shown for simplicity

Fig. 4.1 Partitioning the multiplier into modular blocks

(FA) and manchester carry adder (MCA) can be identified in this multiplier. However, considering the interconnection of these cells, the whole multiplier is partitioned into a few parameterized blocks each of which is composed of a number of these basic cells. The multiplier architecture of Fig. 3.1 can be redrawn as shown in Fig. 4.1 and contains the following modular blocks:

(i) Modified Booth encoder (MBE)
(ii) sc_n
(iii) n_adder
(iv) adder_sc_n
(v) n_mca

The first one, i.e., the MBE is a single modified Booth encoder. This component has to be instantiated as many times as required depending on the number of multiplier bits n_y. The exact number of instantiation required is termed as the *depth* of the multiplier and is given by

$$\text{depth} = \begin{cases} n_y/2 \text{ for even } n_y \\ (n_y+1)/2 \text{ for odd } n_y. \end{cases} \qquad (4.1)$$

The block *sc_n* is composed of (n_x+1) number of selector-complementers, where n_x is the number of bits in the multiplicand. As shown in Fig. 4.1, this block is used in the first two rows of the multiplier array corresponding to the first two rows of selector-complementers in the architecture of Fig. 3.1. The next block in the modular array of Fig. 4.1 is the *n_adder* block corresponding to the first row of full adders of the multiplier array of Fig. 3.1. This block consists of n_x number of full adders. The block *adder_sc_n* is a combination of one row of SCs followed by one row of FAs. It consists of (n_x+1) number of selector-complementers and n_x number of full adders. This block is used as many times as required after the first row of full-adders. The exact number of *adder_sc_n* blocks depends on the depth of the multiplier and is determined from the number of bits in the multiplier (n_y) using equation no. 4.1. The next block in the modular array of Fig.

4.1 is the *n_mca* block which consists of (depth*2+n_x −1) manchester carry adders. This block in Fig. 4.1 corresponds to all the MCAs at the final stage of the multiplier array shown in Fig. 3.1.

4.3 Design Hierarchy

The VHDL code of the multiplier is at the top level of the design hierarchy. The hierarchy is shown in Fig. 4.2. In the top level VHDL code, all the modular blocks mentioned above are instantiated as many times as required depending on the operand wordlengths. For the modified Booth encoders, the code of only one MBE cell is written in VHDL. In the top level code, this block is instantiated a number of times equal to the *depth* of the multiplier. The VHDL codes of the blocks *sc_n*, *n_adder* and *adder_sc_n* are written such that they are already parameterized on the value of n_x. Two instantiations of *sc_n* and one instantiation of *n_adder* are made in the VHDL code of the multiplier. Number of instances of *adder_sc_n* in the top level code depends on n_y. The VHDL code of the block *n_mca* is written such that it is already parameterized completely on the value of both n_x and n_y. Only one instantiation of this block is made in the top level VHDL code of the multiplier.

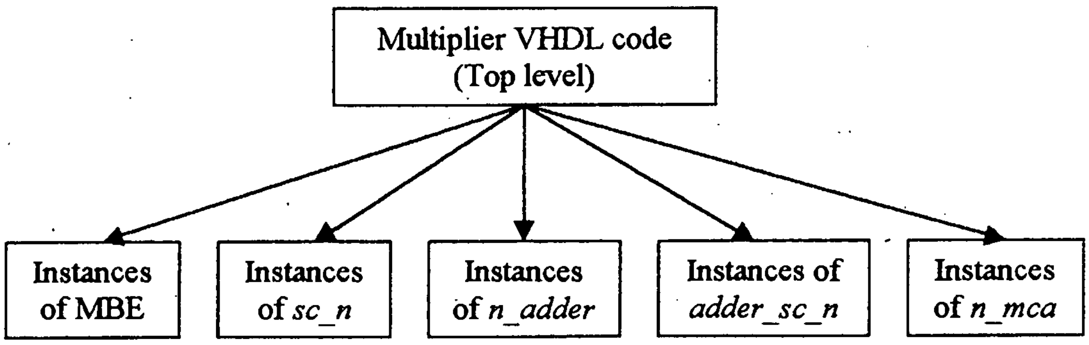


Fig. 4.2 Design Hierarchy

4.4 VHDL Model

In this section, the VHDL codes developed for all the modular blocks discussed in section 4.2 are presented.

4.4.1 The Modified Booth Encoder

As mentioned before, the number of modified booth encoders in the multiplier array depends on the number of bits (n_y) in the multiplier (Y). The VHDL code for a single modified Booth encoder is developed first. Then in the top level VHDL code of the multiplier, the required number of modified Booth encoders is specified as one of the generics *depth* and then instantiated [2], [4]. The logic functions performed by the modified Booth encoder shown in Fig. 3.2 are as follows:

$$K_1 = y_{i-1} \oplus y_I \tag{4.2}$$

$$K_2 = y_{i-1}\ y_i\ \overline{y}_{i+1} + \overline{y}_{i-1}\ \overline{y}_i\ y_{i+1} \tag{4.3}$$

$$CM = \overline{y_{i-1} y_i} \cdot y_{i+1} = \overline{y_{i-1}}\, y_{i+1} + \overline{y_i}\ y_{i+1} \tag{4.4}$$

The VHDL code of a single modified Booth encoder is given below:

```
-- Modified Booth's encoder
-- For C-testable Modified Booth's Array multiplier

library IEEE;
use IEEE.std_logic_1164.all;


entity mod_both_encoder is
  port(y_i_1, y_i, y_i_2: in std_logic;
       K1, K2, CM: out std_logic
      );
end mod_both_encoder;

architecture dataflow of mod_both_encoder is
begin
     K1 <= y_i_1 xor y_i;
```

```
      K2 <= (y_i_1 and y_i and (not y_i_2)) or
            ((not y_i_1) and (not y_i) and y_i_2);
      CM <= (y_i_2 and (not y_i)) or ((not y_i_1) and y_i_2);
end dataflow;

configuration cfg_mod_both_encoder of mod_both_encoder is
   for dataflow
   end for;
end cfg_mod_both_encoder;
```

4.4.2 The sc_n

This block consists of a number of selector-complementers depending on the number of bits (n_x) in the multiplicand (X). There are (n_x+1) selector-complementers in this block including the leftmost selector-complementer whose output is complemented. In the VHDL code of *sc_n*, the number of bits in the multiplicand is explicitly specified as one of the generics, *nx*. The VHDL code for the entire *sc_n* block is written and two instantiations of this are made in the top level VHDL code of the multiplier as per Fig. 4.1. The logic function performed by one selector-complementer shown in Fig. 3.3 is given by

$$Z_i = K_1x_i + K_2x_{i-1} \quad \textbf{(4.5)}$$

The VHDL code of the *sc_n* block is given below:

```
-- Selector-complementer
-- Generic model with N-bit size
-- For C-testable Modified Booth's Array multiplier

library IEEE;
use IEEE.std_logic_1164.all;
```

```
entity sc_n is
  generic (nx : integer := 8);
  port(X : in std_logic_vector (nx-1 downto 0);
       CM, K1, K2, X_1 : in std_logic;
       Z : out std_logic_vector (nx downto 0)
      );
end sc_n;

architecture rtl of sc_n is
begin
    process (X,CM,K1,K2,X_1)
    variable r_in : std_logic;
    variable y : std_logic;
    begin
     for I in nx downto 0 loop
        if I = 0 then
           r_in := X_1;
        else
           r_in := X(I-1);
        end if;
        y := (X(I) and K1) or (K2 and r_in);
        if I = nx then
           Z(I) <= not(CM xor y);
        else
           Z(I) <= CM xor y;
        end if;
     end loop;
    end process;
end rtl;

configuration cfg_sc_n of sc_n is
   for rtl
   end for;
end cfg_sc_n;
```

4.4.3 The n_adder

As mentioned in section 4.2, the number of full adders in the *n_adder* block is n_x, which is explicitly specified in the VHDL code as one of the generics, *nx*. One instantiation of this block is made in the top level VHDL code of the multiplier in accordance with Fig. 4.1. The logic functions performed by a single 1-bit full-adder is as follows:

$$SUM = A \oplus B \oplus C \tag{4.6}$$

$$CARRY = AB + BC + CA \tag{4.7}$$

Where, A, B and C are the three inputs to the full-adder.

The VHDL code of the block *n_adder* is shown below:

```
-- Generic N number of 1-bit Full adder
-- For C-testable Modified Booth's Array multiplier

library IEEE;
use IEEE.std_logic_1164.all;

entity n_adder is
  generic (nx : integer := 8);
  port(A, B, Cin: in std_logic_vector (nx-1 downto 0);
       Sum, Cout: out std_logic_vector (nx-1 downto 0));
end n_adder;

architecture rtl of n_adder is
begin
    process (A, B, Cin)
    begin
     Sum <= A xor B xor Cin;
     Cout <= (A and B) or (B and Cin) or (Cin and A);
    end process;
end rtl;

configuration cfg_n_adder of n_adder is
   for rtl
   end for;
end cfg_n_adder;
```

4.4.4 The adder_sc_n

In this block, the number of selector-complementers is (n_x+1) in a row and the number of full adders is n_x in the following row. One of the inputs of all the full-adders in this block comes from the outputs of the selector-complementers of the same block except the leftmost one. In the VHDL code of the *adder_sc_n* block, *nx* is specified as a generic. The VHDL code of the block *adder_sc_n* is given below:

```
-- Combined selector-complementer and adder
-- Generic model with N-bit size
-- For C-testable Modified Booth's Array multiplier

library IEEE;
use IEEE.std_logic_1164.all;

entity adder_sc_n is
  generic (nx : integer := 8);
  port(A, B, X: in std_logic_vector (nx-1 downto 0);
       CM, K1, K2, X_1: in std_logic;
       Z_not: out std_logic;
       Sum, Cout: out std_logic_vector (nx-1 downto 0));
end adder_sc_n;

architecture rtl of adder_sc_n is
  signal Z : std_logic_vector (nx-1 downto 0);
begin
    process (A,B,CM,K1,K2,X_1)
    variable r_in, y : std_logic;
    begin
     for I in 0 to nx loop
        if I = 0 then
           r_in := X_1;
        else
           r_in := X(I-1);
        end if;
        if I < nx then
           Z(I) <= (X(I) and K1) or (K2 and r_in);
        else
           y := (X(I) and K1) or (K2 and r_in);
        end if;
     end loop;
    -- The following part is for adder.

    Sum <= A xor B xor Z;
    Cout <= (A and B) or (B and Z) or (Z and A);
    Z_not <= (not y);
   end process;
end rtl;

configuration cfg_adder_sc_n of adder_sc_n is
   for rtl
   end for;
end cfg_adder_sc_n;
```

4.4.5 The n_mca

This block consists of a number of manchester carry adders, depending on the number of bits (n_x) in the multiplicand and the number of modified Booth encoders (depth) which is dependent on the number of bits (n_y) in the multiplier. The exact number of manchester carry adders in this block is (depth*2+ n_x −1). In the VHDL code of the block *n_mca*, the numbers *nx* and *depth* are explicitly specified as generics in order to generate the parameterized manchester carry adder chain. The manchester carry adders often employ some form of *fast* carry propagation scheme [11] and are implemented differently than the carry save adders (full-adders) used in the array. However, both these adders perform the same logic functions and therefore may be regarded to be the same from the point of view of logic functionality. The VHDL code of the block *n_mca* is shown below:

```
--  Carry Adder Chain
-- For C-testable Modified Booth's Array multiplier

library IEEE;
use IEEE.std_logic_1164.all;
```

```
entity n_mca is
  generic (nx : integer := 8; ny : integer := 10; depth :
integer := 5);
  port(A, B : in std_logic_vector (depth*2+nx-2 downto 0);
       e1 : in std_logic;
       P : out std_logic_vector (depth*2+nx-1 downto 0));
end n_mca;

architecture rtl of n_mca is
begin
    process (A, B, e1)
    variable Cin, Cout : std_logic;
    begin
     for I in 0 to depth*2+nx-2 loop
        if I = 0 then
           Cin := e1;
        else
           Cin := Cout;
        end if;
        P(I) <= A(I) xor B(I) xor Cin;
        Cout := (A(I) and B(I)) or (B(I) and Cin)
             or (Cin and A(I));
     end loop;
     P(depth*2+nx-1) <= (not Cout);
    end process;
end rtl;

configuration cfg_n_mca of n_mca is
   for rtl
   end for;
end cfg_n_mca;
```

4.4.6 The Multiplier

In the top level VHDL code of the multiplier, the aforementioned blocks have been declared as components. Then several instantiations of these components have been made according to the size of the multiplier. Three generics are specified in the *entity* of multiplier. These are *nx*, for the number of bits in the multiplicand; *ny*, for the number of bits in the multiplier; *depth*, for the number of modified booth encoders needed for the desired size of the multiplier. Primary inputs and outputs of the multiplier have been declared as inputs and outputs respectively in the *entity* of the VHDL code of the multiplier. Intermediate signals between several blocks have been declared as signals in the architecture of the VHDL code of the multiplier. The VHDL code of the top level block of the multiplier is given below:

```
-- C-testable Modified Booth's Array multiplier

library IEEE;
use IEEE.std_logic_1164.all;

entity booth_multiplier is
  generic (nx : integer := 8; ny : integer := 10; depth :
integer := 5);
  port(X: in std_logic_vector (nx-1 downto 0);
       Y: in std_logic_vector (ny-1 downto 0);
       X_1, Y_1: in std_logic;
       E1, E2, E3, E4: in std_logic;
       P: out std_logic_vector (depth*2+nx-1 downto 0));
end booth_multiplier;

architecture structure of booth_multiplier is

component n_adder
  generic (nx : integer := 8);
  port(A, B, Cin: in std_logic_vector (nx-1 downto 0);
       Sum, Cout: out std_logic_vector (nx-1 downto 0));
end component;

component mod_both_encoder
  port(y_i_1, y_i, y_i_2: in std_logic;
       K1, K2, CM: out std_logic);
end component;

component n_mca
  generic (nx, ny, depth : integer);
  port(A, B : in std_logic_vector (depth*2+nx-2 downto 0);
       e1 : in std_logic;
       P : out std_logic_vector (depth*2+nx-1 downto 0));
end component;
```

```
component sc_n
  generic (nx : integer := 8);
  port(X : in std_logic_vector (nx-1 downto 0);
       CM, K1, K2, X_1 : in std_logic;
       Z : out std_logic_vector (nx downto 0));
end component;

component adder_sc_n
  generic (nx : integer := 8);
  port(A, B, X: in std_logic_vector (nx-1 downto 0);
       CM, K1, K2, X_1: in std_logic;
       Z_not: out std_logic;
       Sum, Cout: out std_logic_vector (nx-1 downto 0));
end component;
subtype a_width is std_logic_vector (nx-1 downto 0);
  type a_depth is array (integer range 0 to depth-2) of
a_width;

  subtype b_width is std_logic_vector (nx downto 0);
  type b_depth is array (integer range 0 to depth-1) of
b_width;

  signal Sum, Cout : a_depth;
  signal K1, K2, CM : std_logic_vector (depth-1 downto 0);
  signal AV, BV, Cin : a_depth;
  signal Z : b_depth;
  signal Z_not : std_logic_vector (depth-1 downto 0);
  signal AM : std_logic_vector (depth*2+nx-2 downto 0);
  signal BM : std_logic_vector (depth*2+nx-2 downto 0);
  signal EA, EB: std_logic_vector(1 downto 0);

begin

    -- Instantiate modified booth encoder
mbe0: for I in 0 to depth-2 generate
        c:if I= 0 generate
         c0: mod_both_encoder port map  (Y_1, Y(I), Y(I+1),
                              K1(I), K2(I), CM(I));
            end generate;

        m:if I > 0 generate
         c1: mod_both_encoder port map  (Y(I*2-1), Y(I*2),
Y(I*2+1), K1(I), K2(I), CM(I));
        end generate;
      end generate;
mbe1: if (ny mod 2) = 1 generate
```

```
          c2: mod_both_encoder port map  (Y(depth*2-3),
          Y(depth*2-2), Y(depth*2-2), K1(depth-1), K2(depth-1),
          CM(depth-1));
            end generate;

     mbe2: if (ny mod 2) = 0 generate
     c3: mod_both_encoder port map (Y(depth*2-3), Y(depth*2-2),
     Y(depth*2-1), K1(depth-1), K2(depth-1), CM(depth-1));
     end generate;

              -- Instantiate selector-complementer modules
         sc_0: sc_n generic map (nx)
                 port map  (X, CM(0), K1(0), K2(0), X_1, Z(0));

         sc_1: sc_n generic map (nx)
                 port map  (X, CM(1), K1(1), K2(1), X_1, Z(1));

         adder0: process (E1, E2, E3, E4, Z)
               begin
                for J in 0 to nx-1 loop
                     if (J mod 2) = 1 then
                      BV(0)(J) <= E1;
                  end if;
                  if (J mod 2) = 0 and J < nx-2 then
                      BV(0)(J) <= E2;
                  end if;
                  if J = nx-2 then
                      BV(0)(J) <= E3;
                  end if;
                  if J < nx-1 then
                      AV(0)(J) <= Z(0)(J+2);
                   end if;
                  if J = nx-1 then
                      AV(0)(J) <= E4;
                  end if;
                  Cin(0)(J) <= Z(1)(J);
                   end loop;
             end process;

                 add0: n_adder generic map (nx)
                port map (AV(0), BV(0), Cin(0), Sum(0), Cout(0));

          Z_not(0) <= Z(0)(nx);
          Z_not(1) <= Z(1)(nx);
          EA(0) <= E1;
          EA(1) <= E2;
          EB(0) <= E4;
     EB(1) <= E3;
     --adder_sc1:
     yy1: for I in 1 to depth-2 generate
                yy2:for J in 0 to nx-1 generate
                 xx2 : if J < nx-1 generate
                    BV(I)(J) <= Cout(I-1)(J+1);
                  end generate;
                  xx3: if J = nx-1 generate
                    BV(I)(J) <= EA(I mod 2);
                 end generate;
                 xx4: if J < nx-2 generate
                    AV(I)(J) <= Sum(I-1)(J+2);
                  end generate;
            xx5:        if J = nx-2 generate
                 AV(I)(J) <= Z_not(I);
                 end generate;
              xx6:  if J = nx-1 generate
                 AV(I)(J) <= EB(I mod 2);
                 end generate;
             end generate;

        add_sc1:
            adder_sc_n
            generic map (nx)
            port map  (AV(I), BV(I), X, CM(I+1), K1(I+1),
                 K2(I+1), X_1, Z_not(I+1), Sum(I), Cout(I));
            end generate;

-- Instance carry adder chain
   mca: process (CM, Z(0),Sum,Cout,E1, Z_not)
             begin
            for I in 0 to (depth-1)*2 loop
            if I = 0 then
            BM(I) <= CM(I);
            AM(I) <= Z(0)(I);
            end if;
            if I = 1 then
            BM(I) <= Z(0)(I);
            AM(I) <= E1;
            end if;
            if I > 1 and (I mod 2) = 0 then
            BM(I) <= CM(I/2);
            AM(I) <= Sum(I/2-1)(0);
            end if;
             if I > 1 and (I mod 2) = 1 then
            BM(I) <= Cout((I-1)/2-1)(0);
            AM(I) <= Sum((I-1)/2-1)(1);
            end if;
               end loop;
               for I in depth*2-1 to depth*2+nx-2 loop
            if I = depth*2+nx-2 then
            BM(I) <= Cout(depth-2)(nx-1);
            AM(I) <= Z_not(depth-1);
            else
            BM(I) <= Cout(depth-2)(I-nx+1);
             AM(I) <= Sum(depth-2)(I-nx+2);
             end if;
          end loop;
       end process;

       mc: n_mca generic map (nx, ny, depth)
           port map (AM, BM, E1, P);

end structure;
```

4.4.7 Achieving Parameterizability

In this design a very powerful feature (generic) of VHDL has been used to achieve parameterizability. The above design is valid for multiplier arrays of any operand size provided that the number of bits in the multiplicand (X) and multiplier (Y) as well as the *depth* are explicitly specified as generics in the VHDL codes of the individual blocks and in the top level multiplier. The number of MBEs is determined by the generic, *depth* which is dependent on the generic *ny* (the number of multiplier bits). The number of SCs, FAs and MCAs are determined by the generics *nx* and *depth*. So, it is evident that parameterizability is effectively achieved in this design.

4.5 Testbench and Simulation

Finally, a testbench has been written to test whether the multiplier performs the multiplication operation properly or not. The testbench is shown below:

```
-- Test bench for C-testable Modified Booth's Array
multiplier

library IEEE;
use IEEE.std_logic_1164.all;
--use work.booth_multiplier;

entity tb_booth_multiplier is
  generic (nx : integer := 8; ny : integer := 10; depth :
integer := 5);
  end tb_booth_multiplier;

architecture vector of tb_booth_multiplier is

  component booth_multiplier
  generic (nx : integer := 8; ny : integer := 10; depth :
integer := 5);
  port(
       X: in std_logic_vector (nx-1 downto 0);
       Y: in std_logic_vector (ny-1 downto 0);
       X_1, Y_1: in std_logic;
       E1, E2, E3, E4: in std_logic;
       P: out std_logic_vector (depth*2+nx-1 downto 0)
      );
  end component;

  signal X: std_logic_vector (nx-1 downto 0);
  signal Y: std_logic_vector (ny-1 downto 0);
  signal X_1, Y_1:  std_logic;
  signal  E1, E2, E3, E4:  std_logic;
  signal  P: std_logic_vector (depth*2+nx-1 downto 0);
  constant PERIOD : time := 50 ns;

Begin

     U1: booth_multiplier generic map (nx, ny, depth)
          port map (X, Y, X_1, Y_1, E1, E2, E3, E4, P);

     U2: Process
          variable Test : integer := 0;
          begin
             case Test is
               when 0 =>
                  X <= (others => '0');
                  Y <= (others => '0');
                  X_1 <= '0';
```

```
        Y_1 <= '0';
        E1 <= '0';
        E2 <= '0';
        E3 <= '0';
        E4 <= '0';
    when 1 =>
        X <= ('1',others => '0');
        Y <= "0101010101";
    when 2 =>
        X <= (others => '1');
        X_1 <= '1';
        E1 <= '1';
        E2 <= '1';
        E3 <= '1';
        E4 <= '1';
    when 3 =>
        X <= ('1',others => '0');
        Y <= "1010101010";
        X_1 <= '0';
        Y_1 <= '1';
    when 4 =>
        X <= (others => '0');
        Y <= (others => '1');
        Y_1 <= '0';
        E1 <= '0';
        E2 <= '0';
    when 5 =>
        Y <= "1100110011";
        Y_1 <= '1';
        E1 <= '1';
        E2 <= '1';
        E3 <= '0';
        E4 <= '0';
    when 6 =>
        X <= (others => '1');
        Y <= "0101010011";
        X_1 <= '1';
        Y_1 <= '0';
    when 7 =>
        Y <= "0001000100";
        Y_1 <= '1';
        E1 <= '0';
        E2 <= '0';
        E3 <= '1';
        E4 <= '1';
```

```
when 8 =>
   X <= "01010101";
   Y <= "0011001100";
   X_1 <= '0';
   E2 <= '1';
   E4 <= '0';
when 9 =>
   Y <= "1100110011";
   Y_1 <= '0';
   E1 <= '1';
   E2 <= '0';
   E3 <= '0';
   E4 <= '1';
when 10 =>
   X <= (others => '1');
   --Y <= "00110011";
   X_1 <= '1';
  -- E1 <= '1';
   E2 <= '1';
  -- E3 <= '0';
   E4 <= '0';
when 11 =>
   Y <= "0011001100";
   Y_1 <= '1';
   E1 <= '0';
   E2 <= '0';
   E3 <= '1';
   E4 <= '1';
when 12 =>
   X <= (others => '0');
   Y <= "1010101010";
   X_1 <= '0';
when 13 =>
   X <= (others => '1');
   X_1 <= '1';
   E3 <= '0';
   E4 <= '0';
when 14 =>
   Y <= "0110011001";
   X_1 <= '0';
   E1 <= '1';
   E2 <= '1';
   E3 <= '1';
   E4 <= '1';
```

```
                    when 15 =>
                        Y <= "1001100110";
                        Y_1 <= '0';
                        E1 <= '0';
                        E2 <= '0';
                        E3 <= '0';
                        E4 <= '0';
--                  when 16 =>
--                      X <= (others => '0');
--                      Y <= "01010101";
                    when 17 =>
                            X <= (others => '0');
                    --      Y <= "1001100110";
                        E1 <= '1';
                        E2 <= '1';
                        E3 <= '1';
                        E4 <= '1';
        --          when 18 =>
                        X <= (others => '1');
                        Y <= "0110011001";
                      X_1 <= '1';
                        Y_1 <= '1';
        --              E1 <= '1';
                        E2 <= '0';
                        E3 <= '0';
        --              E4 <= '1';
                    when 19 =>
                        --X <= (others => '1');
                        Y <= (others => '1');
                        E1 <= '0';
                        E4 <= '0';
                when others => Null;
                end case;
                wait for PERIOD;
                Test := Test + 1;
        end process;

end vector;
```

The above testbench has been used to simulate the functionality of the multiplier. The simulation results have shown that the VHDL based design of the multiplier is functionally correct.

One set of simulation results is shown in Fig. 4.3. All numbers shown in this figure are in hexadecimal. The multiplier (Y) is fixed at 02. Results are shown for different values of multiplicand (X). For example, 82 (-126 in decimal) multiplied by 02 gives a product output of FF04 (-252 in decimal) which is the desired result. Also, 06 multiplied by 02 gives 000C (12 in decimal) which is the desired result.

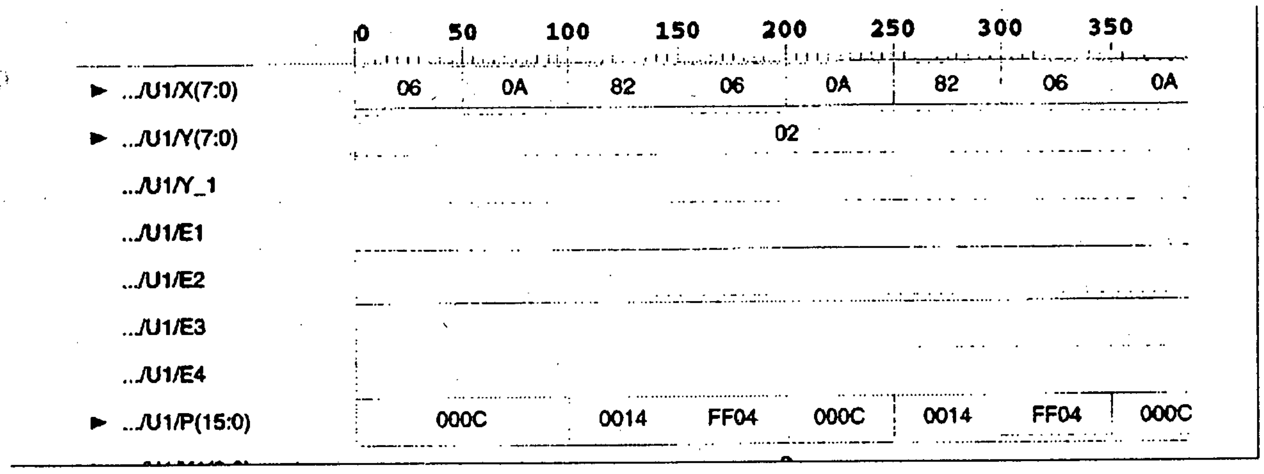


Fig. 4.3 Simulation result

Chapter 5

Conclusions and Recommendations

5.1 Conclusions

The design of an easily testable parallel array multiplier has been presented in this thesis. The multiplier can be tested for all single stuck-at faults using only 19 vectors. The number of test vectors remains constant irrespective of the operand wordlengths. This means that multiplier arrays of different sizes constructed using the proposed architecture will require the same number of vectors (19 only) for testing all single stuck-at faults. However, the wordlengths of the X and Y operands of the test vectors have to be adjusted according to the proposed pattern. Such multipliers are called C-testable.

The modified Booth algorithm has been used to design the multiplier. The regular structure of the array has reduced the complexity of testing to a great extent. Since, modified Booth algorithm generates approximately half the number of partial products compared to the straightforward carry-save array multiplication scheme[9], [10], the proposed multiplier will be almost two times faster than the carry-save one.

The above design has been coded in VHDL for the automatic synthesis of testable multipliers. The capability of VHDL to generate arbitrary n-bit modules has been exploited in designing parameterizable architectures. The VHDL code accepts the wordlengths of the operands X and Y as inputs. Depending on these wordlengths, it generates a multiplier array of the appropriate size. Due to the use of a high-level language like VHDL for the design, it can be used for synthesizing parameterizable multipliers for any target process. That is, the same VHDL code can be used to design multipliers for a variety of processes without having to make any modification in the design or the VHDL code. Thus, the VHDL based design proposed in this thesis is process independent. FPGA implementation of multipliers of various sizes are also possible using the same VHDL code.

Finally, it is expected that the proposed VHDL based design will be very useful for the quick generation of easily testable multiplier macrocells of arbitrary size for a variety of target processes.

5.2 Further Work

Future work may include the synthesis of a multiplier circuit from the VHDL code for a suitable target process. After necessary simulation and verification, the synthesized circuit may be fabricated. Finally, the fabricated circuit (chip) can be tested for evaluation of practical performance. Multipliers of various sizes can also be implemented on FPGAs using the proposed VHDL code. The practical performance of the two types of implementations may then be compared.

Another important direction of research would be to integrate the VHDL code into an Electronic Design Automation (EDA) tool. It will then form the testable multiplier compiler part of the tool. Also, VHDL codes may be developed for the automatic synthesis of various arithmetic, control and memory blocks. All these codes can then be integrated along with the proposed multiplier code for the development of a new Electronic Design Automation (EDA) tool.

References

[1] D. D. Gajski, N. D. Dutt, C. H. Wu, Y. L. Lin, "High-level synthesis, introduction to chip and system design," *Kluwer Academic Publishers*, 1991.

[2] Z. Navabi, "VHDL analysis and modeling of digital systems," *McGraw-Hill Inc.*, NY, 1993.

[3] J. R. Armstrong, "Chip level modeling with VHDL," *Prentice-Hall International Inc.*, USA, 1989.

[4] D. Pellerin, D. Taylor, "VHDL made easy," *Prentice-Hall International Inc.*, USA, 1997.

[5] K. Takeda, F. Ishino, "A single-chip 80-bit floating point processor," *IEEE J. of Solid-State Circuits,* Vol. SC-20, No.5, pp. 986-991, Oct. 1985.

[6] P. A. Lynn, W. Fuerst, "Introductory digital signal processing with computer applications," *Jhon Willey & Sons*. 1992.

[7] A. V. Oppenheim and R. W. Schafer, "Discreet-time signal processing," *Prentice-Hall of India PTY. Ltd.*, Delhi, 1994.

[8] Frank P. J. M. Welton, A. Delaruelle, "A 2-μm CMOS 10-MHz microprogrammable signal processing core with an on-chip multiport memory bank," *IEEE J. of Solid-State Circuits,* Vol. SC-20, No.3, pp. 754-760, June 1985.

[9] L. P. Rubinfield, "A proof of the modified Booth's algorithm for multiplication," *IEEE Trans. Comput.,* pp. 1014-1015, Oct. 1975.

[10] J. J. F. Cavanagh, "Digital computer arithmetic design and implementation," *McGraw-Hill Book Company,* New York, 1985.

[11] N.Weste and K. Eshraghian, "Principle of CMOS VLSI design," *Addision-Wesley Publishing Company*, Sydney, 1993.

[12] A. Pucknell and K. Eshraghian, "Basic VLSI design," *Prentice-Hall of Australia PTY. Ltd.*, Sydney, 1994.

[13] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Trans. on Electronic Comput.*, Vol. EC-13, pp. 14-17, Feb. 1964.

[14] L. Dadda, "Some schemes for parallel multipliers," *Alta Frequenza*, Vol. 34, No. 5, pp. 349-356, May 1965.

[15] D. G Crawley and G. A. J. Amaratunga, " 8 × 8 bit pipelined Dadda multiplier in CMOS," *IEE Proceedings*, Vol. 135, Pt. G, No. 6, pp. 231-240, Dec. 1988.

[16] C. R. Baugh and B. A. Wooley, "A two's complement parallel array multiplication algorithm," *IEEE Trans. Comput.*, Vol. C-22, No. 12, pp. 1045-1047, Dec. 1973.

[17] J. A. Starzyk and Z. S. R. Dandu, "Overlapped multi-bit scanning multiplier," *Proceedings of IEEE Int.Conf. on Comp. Design: VLSI in Computers*, ICCD; 85, NY. Oct. 1985, pp. 363-366.

[18] B. R. Wilkins, "Testing digital circuits: an introduction," *Van Nostrand Reinhold (UK) Co. Ltd.*, 1986.

[19] T.Williams and K. Parker, "Design for testability - a survey," *IEEE Trans. Comput.*,Vol. C-31, pp. 2-15, Jan. 1982.

[20] J. P. Shen and F. J. Ferguson, "The design of easily testable VLSI array multiplication," *IEEE Trans. Comput.*, Vol. C-33, No. 6, pp. 554-560, June 1984.

[21] A. R. Takach and N. K. Jha, "Easily testable gate-level and DCVS multipliers," *IEEE Trans. Computer-Aided Design*, Vol.10, No. 7, pp. 932-942, July 1991.

[22] S. M. Aziz, "A C-testable modified Booth's array multiplier," *8th International Conference on VLSI design*, New Delhi, India, Jan. 1995, pp. 278-282.

[23] R. Stans, "The testability of a modified Booth multiplier," *Proceedings of 1st Europian Test Conference*, 1989, pp.286-2938.

[24] W. A. J. Waller and S. M. Aziz, "A C-testable parallel multiplier using differential cascode voltage switch (DCVS) logic," *International Conference on Very Large Scale Integration: VLSI '93,* Sept. 6-10, 1993, pp. 3.4.1-10.

[25] N. F. Benschop, "Layout compilers for variable array multipliers," *Proc. Custom Integrated Circuits Conf.*, May 1983, pp. 336-339.

[26] K. C. Chu and R. Sharma, "A technology independent MOS multiplier generator," 21st Design Automation Conf., 1984, pp. 90-97.

[27] G. Venzl and R. Mitchell, "A compilable binary tree parallel multiplier designed for speed and testability," *Proc. Custom Integrated Circuits Conf.*, 1989, pp. 93-94.

[28] A. D. Friedman, "Easily testable iterative systems," *IEEE Trans. Comput.,* Vol. C-22, pp. 1061-1064, Dec. 1973.

[29] Y. K. Malaiya, "Testing stuck-on faults in CMOS integrated circuits," *Proc of International Conference on CAD,* Santa Clara, CA, Nov. 1984, pp. 248-250.

[30] S. M. Reddy and M. K. Reddy, "Testable realizations for FET stuck-open faults in CMOS combinational logic circuits," *IEEE Trans. Comput.,* Vol. C-35, No. 8, pp. 742-754, Aug. 1986.

[31] S. M. Reddy, V. D. Agarwal and S. K. Jain, "A gate level model for CMOS combinational logic circuits with applications to fault detection," 21st Design Automation Conf., 1984, pp. 504-509.

[32] M. Roorda, "Method to reduce the sign bit extension in a multiplier that uses the modified Booth algorithm," *Electronic Letters,* Vol. 22, No. 20, pp. 1061-1062, 25th Sept. 1986.

[33] N. Burgess, "Removal of sign-extension circuitry from Booth's algorithm multiplier-accumulators," *Electronic Letters,* Vol. 26, No. 17, pp. 1413-1415, 16th Aug. 1990.