Analysis of Booth's Multiplier Algorithm vs Array Multiplier Algorithm and their FPGA Implementation

by

Anantha Gunturu

Submitted in partial fulfillment of the requirements for the Degree of

Master of Science

in the

Electrical Engineering

Program

YOUNGSTOWN STATE UNIVERSITY

December 2019

Analysis of Booth's Multiplier Algorithm vs Array Multiplier Algorithm and their FPGA Implementation

Anantha Gunturu

I hereby release this thesis to the public. I understand that this thesis will be made available from the OhioLINK ETD Center and the Maag Library Circulation Desk for public access. I also authorize the University or other individuals to make copies of this thesis as needed for scholarly research.

Signature:

Anantha Gunturu, Student Date

Approvals:

Frank X Li, Thesis Advisor Date

Edward Burden, Committee Member Date

Eric MacDonald, Committee Member Date

Dr. Salvatore A. Sanders, Dean of Graduate Studies Date

ABSTRACT

The purpose of this study is to understand the Booth's Multiplier algorithm for a 32-bit input and compare its performance with an Array Multiplier algorithm for a 32-bit input. The analysis involves implementing the developed VHDL design on an FPGA to understand and compare the performance of these multiplier algorithms. Efficient algorithms for signal processing are critical to very large-scale future applications such as video processing and four-dimensional medical imaging. Similarly, efficient algorithms are important for embedded and power-limited applications since, by reducing the number of computations, power consumption can be reduced considerably.

A brief review of the multiplication process, implementation, and various multiplier algorithms have been included in this document to discuss and ease the understanding and objective of this study. Altera Prime Lite Quartus II version 18.1 was used for simulation of the models. DE10 Standard FPGA development board by Terasic Technologies was used for the hardware implementation of these VHDL models.

After comparing the implementations of both 32-bit Array and Booth multiplier on a Cyclone V FPGA, a conclusion was made that the Booth multiplier has 56 Logic Elements versus 1,719 Logic Elements. Both the multipliers have shown comparable calculation performances.

ACKNOWLEDGMENT

I would like to express my sincere gratitude and thanks to my advisor, Dr. Frank Li, for his consideration, support and guidance throughout this thesis without which this study would not have been possible. I thank Dr. Eric MacDonald and Professor Edward Burden for participating in this thesis committee and providing their valuable feedback. I thank the rest of the faculty of the Electrical and Computer Engineering department for their teachings and support as I pursued this milestone in my career. I would also like to thank my friends and family for their love, support, and confidence in me.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

1.1 Organization

This thesis is organized into 5 chapters. This chapter discusses the motivation and purpose of this thesis. Chapter 2 provides background information on binary multipliers and their applications and discusses recent significant research in this field. Chapter 3 discusses the methods of analysis used during this thesis. Chapter 4 discusses the simulation results obtained from the Altera Quartus II software version 18.1. Chapter 5 concludes the analysis from implementing the subject multiplier algorithms on a DE10 Standard FPGA hardware.

1.2 Motivation

Efficient algorithms for signal processing are critical to very large-scale future applications such as video processing and four-dimensional medical imaging. Similarly, efficient algorithms are important for embedded and power-limited applications since, by reducing the number of computations, power consumption can be reduced considerably. Multiplication is a crucial operation in several Digital Signal Processing (DSP) applications involving convolution, Fast Fourier Transform (FFT) and in the Arithmetic and Logic Unit (ALU) of microprocessors. Several Very Large-Scale Integration (VLSI) design criteria such as the area, power dissipation, speed and cost are dependent on the performance of the multipliers that execute the multiplication operation. Understanding the performance aspects of various multipliers would ultimately help in designing efficient algorithms that execute these multiplication operations.

1.3 Purpose and Objective

The purpose of this thesis is to understand and study the Radix-2 Booth's Multiplier algorithm for a 32-bit input and compare its performance with that of an Array Multiplier algorithm for a 32-bit input. The study involves implementing the developed VHDL design on a DE10 Standard FPGA to understand and compare the performance of these multiplier algorithms. Altera Prime Lite Quartus II version 18.1 is chosen for simulation of the models. DE10 Standard FPGA development board by Terasic Technologies will be used for the hardware implementation of these VHDL models.

End of Chapter 1

CHAPTER 2
LITERATURE REVIEW

2.1 Background

The traditional method of multiplication regardless of the number system involves calculating partial products, shifting them to the left and then adding them together. The primary difficulty with this process is to determine the partial products, as that involves multiplying a long number (multiplicand) by one digit (of the multiplier) at a time. This impacts the speed of execution and thereby the overall performance of the system/block.

```
    12125
x   13134
-------------
      48500     // this is 12125 x 4
     36375      // this is 12125 x 3, shifted 1 position to left
    12125       // this is 12125 x 1, shifted 2 positions to left
   36375        // this is 12125 x 3, shifted 3 positions to left
  12125         // this is 12125 x 1, shifted 4 positions to left
---------------
159249750       // this is the result of 12125 x 13134 operation upon addition of all
                   partial products.
```

It is to be noted that computing the partial products could also involve the addition of the carry when applicable, to the next partial product in the process of multiplication. The standard decimal system multiplication process applies to binary system as well, although it is simpler than the decimal system as there is no table of basic multiplications to remember.

```
      1110      // this is 14 in the binary system
    x 1011      // this is 11 in the binary system
----------------
      1110      // this is 1110 x 1
     1110       // this is 1110 x 1, shifted 1 position to the left
    0000        // this is 1110 x 0, shifted 2 positions to the left
+ 1110          // this is 1110 x 1, shifted 3 positions to the left
-----------------
 010011010      // this is 154 in binary system
```

Other difficulties with the traditional multiplication style are that it handles sign of the number with a separate rule. While digital processing units include the sign of the input numbers within the number itself using the 2's complement technique. This complicates the process and often requires adjustments to the processor to accept and handle such inputs.

2.2 Multipliers

A binary multiplier is an electronic circuit built using binary adders. The multiplication operation is executed using a sequence of shifting, accumulating and adding the partial products as explained in section 2.1.

For an n-bit multiplier and m-bit multiplicand, the resultant product is n + m bits. The generation of n partial products requires n*m two input AND gates. The product is a result of n+m bits. May require at least n adders.

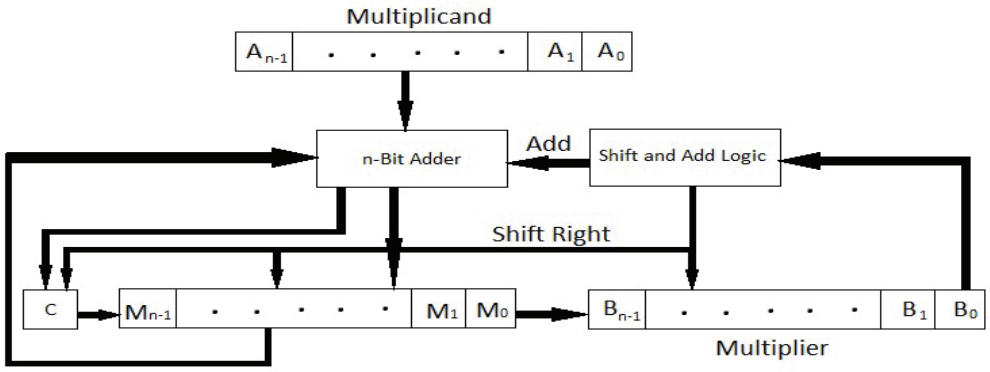


Figure 1. Two n-bit Multiplier

Below is the formal algorithm of a parallel multiplier.

1. Initialize C = 0, M = 0, A = multiplicand, B = multiplier, Count = n.
2. At each step, examine M0.
   If M0 = 1, then add A and B to put the sum in M and set the carry bit C.
3. Right shift the register pair (B, M), with C -->M_{n-1} and M_0 --> B_{n-1}.
4. Decrement the count. If count = 0, stop. If not, go to step 2.

With the understanding we gained from the details of the multiplication process, let's now try to design a 2-bit multiplier. A multiplier with inputs as 2-bits long result in a 4-bit long product. Below are the circuit and truth table representation of a 2x2 bit multiplier.

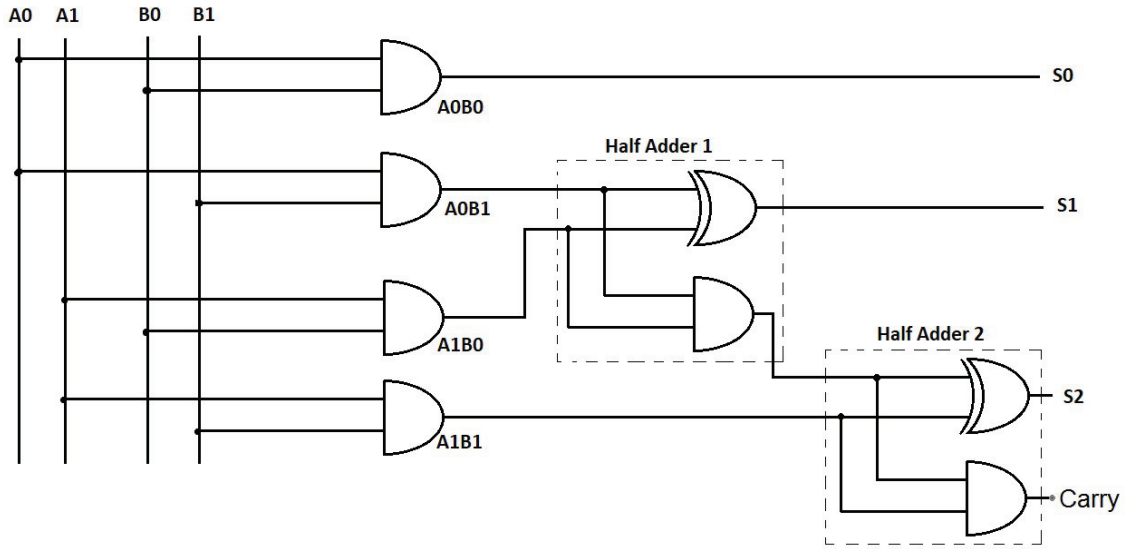


Figure 2. 2x2 bit Multiplier Implementation

A1	A0	B1	B0	A0B0	A0B1	A1B0	HA1 Carry (A0B1 + A1B0)	A1B1	S3 (HA1 Carry + A1B1)	S2	S1	S0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0
0	1	0	1	1	0	0	0	0	0	0	0	1
0	1	1	0	0	1	0	0	0	0	0	1	0
0	1	1	1	1	1	0	0	0	0	0	1	1
1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0	0	0	0	1	0
1	0	1	0	0	0	0	0	1	0	1	0	0
1	0	1	1	0	0	1	0	1	0	1	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	1	0	1	0	0	0	0	1	1
1	1	1	0	0	1	0	0	1	0	1	1	0
1	1	1	1	1	1	1	1	1	1	0	0	1

Table 1. 2x2 bit Multiplier Truth Table

2.3 Multiplier Algorithms

This section introduces some of the multiplier algorithms popularly used in various signal and image processing applications.

2.3.1 Sequential Multiplier

This multiplier employs a sequential circuit using a single n-bit adder to compute the product of two binary numbers, X and Y of n-bit and m-bit length respectively. This sequential circuit processes the partial products one at a time and repeats the process m times. In each step few partial products will be generated, then added to an accumulated partial sum and the resulting partial sum will be shifted to align the accumulated sum with a partial product of next steps. Therefore, each step of a sequential multiplication consists of three operations, i.e. generating partial products, adding the generated partial products to the accumulated partial sum and shifting the partial sum.

2.3.2 Combinational Multiplier

These are used to perform multiplication of two unsigned or signed binary numbers. Given two n-bit inputs X and Y, it is possible to express the 2n-bit product in terms of a combinational function P = X.Y. Such multipliers use the technique of partial product accumulation. Each bit of the multiplier is multiplied against the multiplicand, the product is associated according to the position of the bit within the multiplier, and the resulting products are then added to form the result. If the multiplier bit is a 1, the product is a shifted copy of the multiplicand; if the multiplier bit is a 0, the partial product is 0.

2.3.3 Array Multiplier

This algorithm is very similar to the traditional multiplication process based on the add-and-shift technique followed in any number system. It employs an array of full adders and half adders for the computation of the product. The process involves multiplying bit by bit of the multiplier with the entire multiplicand input. Such individual multiplications result in multiple partial products obtained by sequential shifting and eventually adding

all the partial products to obtain the result of the multiplication. Refer to section 2.1 for an illustration.

The below figure shows the multiplication process through the generation of the partial products and their sum that becomes the result of the multiplication. The example considered below is a 4x4 input that results in an 8-bit product. p0 to p7 indicates the product as a result of the sum of appropriate partial products represented as $a_n b_n$ where n=0 to 7.

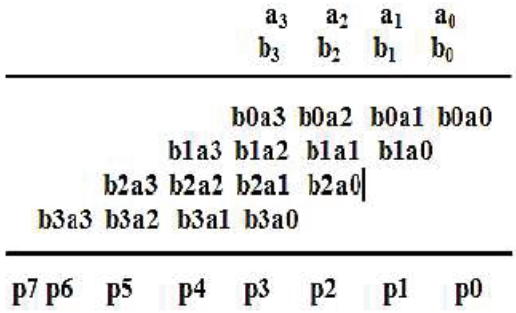


Figure 3. 4x4 Array Multiplier Methodology

The below figure shows the implementation of the above discussed 4x4 array multiplier using a combination of half adders and full adders. These adders execute the sum of partial products to form the result of the multiplication.

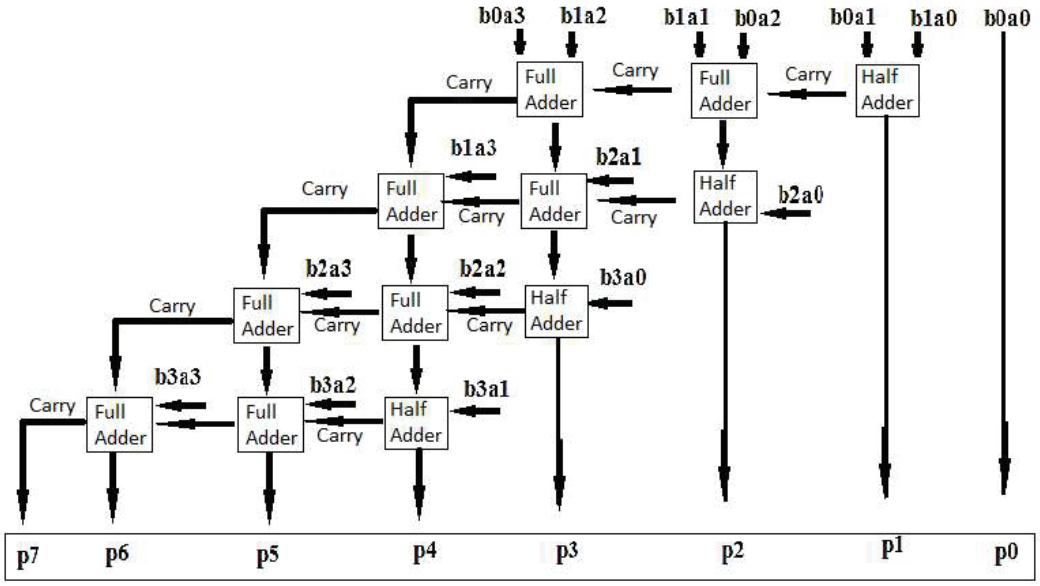


Figure 4. 4x4 Array Multiplier Implementation

2.3.4 Booth's Multiplier Algorithm

This algorithm is a very powerful and efficient algorithm to compute the multiplication of two signed binary numbers in two's complement notation. This algorithm examines adjacent pairs of bits in the 'N'-bit multiplier Q, in signed two's complement representation, including an implicit bit below the least significant bit, $N-1 = 0$. Where these two bits are equal, the product accumulator P is left unchanged. With i=0 to N-1, where $Q_i = 0$ and $Q_{i-1} = 1$, the multiplicand times 2^i is added to P; and where $Q_i = 1$ and $Q_{i-1} = 0$, the multiplicand times 2^i is subtracted from P. The final value of P is the signed product. The order of the steps is not determined in this case. Typically, it proceeds from LSB to MSB, starting at i = 0; the multiplication by 2^i is then typically replaced by incremental shifting of the P accumulator to the right between steps; low bits can be shifted out, and subsequent additions and subtractions can then be done just on the highest N bits of P. Below is a flowchart representation of this algorithm.

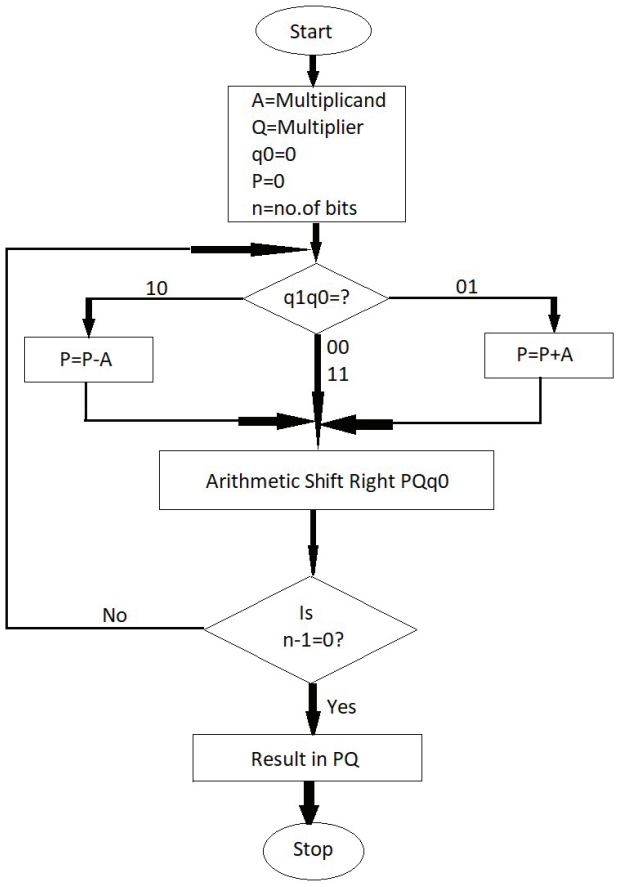


Figure 5. Radix-2 Booth's Algorithm Flowchart

The below figure shows the architecture of Radix-2 Booth's Algorithm implementation.

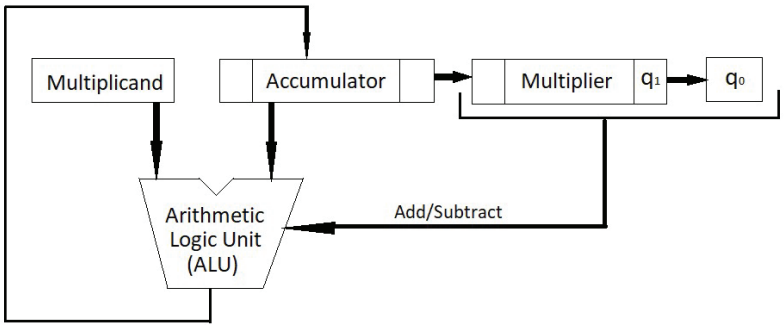


Figure 6. Booth's Algorithm Architecture

Let us understand the working of Booth's algorithm using an example. Consider that the multiplicand A= -7 and multiplier Q = +3. The working of this algorithm can be represented in the form of a tracing table showing the status at each phase of computation. In the current case, input A is a negative number and requires its 2's complement equivalent for further computation.

$A= (-7)_{10} = (1001)_2$ while $(-A)= (0111)_2$

n	Accumulator, P	Multiplier, Q $q_4q_3q_2q_1$	q_0	Action
4	0000	0011	0	Initialization, Value of q_1q_0=10, P=P-A
	0111	0011	0	Arithmetic shift right PQq0
3	0011	1001	1	Value of q1q0=11, Arithmetic shift right PQq_0
2	0001	1100	1	Value of q1q0=01, P=P+A
	1010	1100	1	Arithmetic shift right PQq0
1	1101	0110	0	Value of q1q0=00, Arithmetic shift right PQq0
0	1110	1011	0	Value of n-1=0, process complete. Result in the PQ

Table 2. Radix-2 Booth's Algorithm Grouping Table

At the stage when n-1=0, the result in the PQ = 11101011. Note that this is a negative number and requires its 2's complement equivalent for the resulting product in base-10. Booth's algorithm preserves the sign of the result. With the signed bit as 1 in the value of PQ, the result shall be represented with a negative notation. 2's complement of PQ = $(00010101)_2 = (-21)_{10}$

2.3.5 Significant Improvements

a. Booth's Multiplication Algorithm

There have been significant improvements to the Booth's Multiplication algorithm such that the number of bits grouped would increase thereby reducing the number of computation stages. These strategies have proven to greatly improve the performance of the multipliers and eventually improve the efficiency of signal processing applications. Table 3 lists the bit grouping and the corresponding operation in Radix-4, Radix-8, and Radix-16 type Booth's Multiplication algorithm. A similar strategy has also been followed in developing Radix-32, Radix-128, Radix-256 and even radix-4096 type multipliers whose further research and implementation have been proposed for optimal application design.

b. Array Multiplier

Although the Array Multipliers are not the top preference for signal processing applications, there have been ongoing research and proposals to improve the efficiency of these multipliers. Use of compressors has been proposed to greatly reduce the number of half and full adders and there by reducing the power consumption. 4:2 compressors are now considered basic components in the design of parallel multipliers. It is called compressor, since it compresses four partial products into two. Study on making the Array Multipliers be applicable for signed inputs is also under proposal.

Radix 4		Radix 8		Radix 16	
Code	Operation	Code	Operation	Code	Operation
000	0	0000	0	00000	0
001	1 * Multiplicand	0001	1 * Multiplicand	00001	1 * Multiplicand
010	1 * Multiplicand	0010	1 * Multiplicand	00010	1 * Multiplicand
011	2 * Multiplicand	0011	2 * Multiplicand	00011	2 * Multiplicand
100	-2 * Multiplicand	0100	2 * Multiplicand	00100	2 * Multiplicand
101	-1 * Multiplicand	0101	3 * Multiplicand	00101	3 * Multiplicand
110	-1 * Multiplicand	0110	3 * Multiplicand	00110	3 * Multiplicand
111	0	0111	4 * Multiplicand	00111	4 * Multiplicand
		1000	-4 * Multiplicand	01000	4 * Multiplicand
		1001	-3 * Multiplicand	01001	5 * Multiplicand
		1010	-3 * Multiplicand	01010	5 * Multiplicand
		1011	-2 * Multiplicand	01011	6 * Multiplicand
		1100	-2 * Multiplicand	01100	6 * Multiplicand
		1101	-1 * Multiplicand	1101	7 * Multiplicand
		1110	-1 * Multiplicand	01110	7 * Multiplicand
		1111	0	01111	8 * Multiplicand
				10000	-8 * Multiplicand
				10001	-7 * Multiplicand
				10010	-7 * Multiplicand
				10011	-6 * Multiplicand
				10100	-6 * Multiplicand
				10101	-5 * Multiplicand
				10110	-5 * Multiplicand
				10111	-4 * Multiplicand
				11000	-4 * Multiplicand
				11001	-3 * Multiplicand
				11010	-3 * Multiplicand
				11011	-2 * Multiplicand
				11100	-2 * Multiplicand
				11101	-1 * Multiplicand
				11110	-1 * Multiplicand
				11111	0

Table 3. Bit Grouping in Radix 4, 8 and 16 Type Booth's Multiplication Algorithm

End of Chapter 2

CHAPTER 3
DESIGN AND SIMULATION

This report emphasizes studying the Radix-2 Booth's Multiplier and its comparison with the Array Multiplier. As a part of this study, VHDL models have been built to simulate and analyze the performance of these multipliers. Altera Prime Lite Quartus II version 18.1 was used for simulations.

3.1 Booth's Multiplier Design and Simulation

Following is the VHDL design for Radix-2 Booth's Multiplier.

```
library IEEE;
USE IEEE.std_logic_1164.ALL;
use IEEE.std_logic_unsigned.all;

entity Boothsmult is
  port ( clk, st: in std_logic;
          Mplier, Mcand : in std_logic_vector (31 downto 0);
          Done : out std_logic;
          Product : out std_logic_vector (62 downto 0)      );
end BoothsMult;

Architecture BoothsMult_arch of Boothsmult is

  signal state : integer range 0 to 2;
  signal Counter: integer range 0 to 31;
  signal ACC, RegB, Addout, Addout_Co: std_logic_vector (32 downto 0) :=
"000000000000000000000000000000000";
  signal RegC, Compout : std_logic_vector (31 downto 0) :=
"00000000000000000000000000000000";
  signal Co : std_logic := '0';
  alias B0: std_logic is RegB(0);
  alias B1: std_logic is RegB(1);

begin
  Product <= Acc(30 downto 0) & RegB (32 downto 1);
  Co <= B1 and not B0;   -- B1B0 = 10, add 2's complement of Compouot to ACC
  Compout <= not RegC when Co = '1' else RegC;
                    --  std_logic_vector'(0 => Co)   is 00000000Co
  Addout <= Acc + (Compout(31) & Compout) + std_logic_vector'(0 => Co);

Process(clk)
   begin
```

```
    if clk'event and clk = '1' then
      case state is
        when 0 => if St = '1' then state <= 1;  -- load operation
                            Done <= '0';
          ACC <= (others => '0');
          RegB <= Mplier & '0';
          RegC <= Mcand;
          else state <= 0 ;
          end if;
        when 1 => if (B1 xor B0) = '1' then -- shift operation
                    ACC <= Addout; state <=2;
                  else
                    ACC <= ACC(32) & ACC(32 downto 1);
                    RegB <= Acc(0) & RegB(32 downto 1);
                    if Counter /= 31 then
                      Counter <= Counter +1; state <= 1;
                    else
                      Counter <= 0; state <= 0;Done <= '1';
                    end if;
                  end if;
        when 2 =>   if Counter /= 31 then
                      Counter <= Counter +1; state <= 1;
                    else
                    Counter <= 0; state <= 0; Done <= '1';
                    end if;
                   ACC <= ACC(32) & ACC(32 downto 1);
                   RegB <= ACC(0) & RegB(32 downto 1);
                  end case;
                end if;
              end process;

end BoothsMult_arch;
```

Below is the Altera ModelSim simulation result.

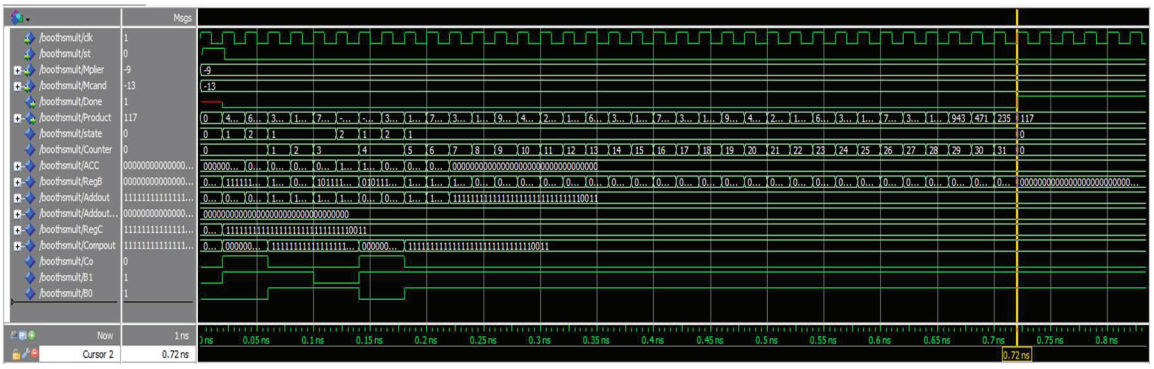


Figure 7. Radix-2 Booth's Multiplier ModelSim Simulation

3.2 Array Multiplier Design and Simulation

Following is the VHDL design for the Array Multiplier.

```
--  This is a 32-bit array multiplier for unsigned binary numbers
--Library Declaration
library IEEE;
use IEEE.std_logic_1164.all;

entity ArrayMult32 is
  port(X, Y: in std_logic_vector(31 downto 0);  --32-bit inputs
       P: out std_logic_vector(63 downto 0));  --64-bit output
end ArrayMult32;

architecture Behavioral of ArrayMult32 is
  type Matrix32 is array (0 to 31, 0 to 31) of std_logic;
  type Matrix32natural is array (1 to 32, 1 to 32) of std_logic;
  signal XY:  Matrix32; --define XY as 32x32 matrix
  signal C, S : Matrix32natural; -- define C,S as 32x32 matrix with indexes 1 to 32

--Full Adder Component Declaration
component FullAdder10ns
  port(X, Y, Cin: in std_logic;
       Cout, Sum: out std_logic);
end component;

--Half Adder Component Declaration
component HalfAdder10ns
  port(X, Y: in std_logic;
       Cout, Sum: out std_logic);
end component;

begin
  --Generate AND gates and signals
  ANDgen1:  for j in 0 to 31 generate  --For X input
    ANDgen2:  for k in 0 to 31 generate  --For Y input
      XY(j,k) <= X(j) and Y(k);          --And each X and Y input bit, store in XY matrix
    End generate;
  End generate;

P(0) <= XY(0,0); --- first bit of product

----Row 1 special case, 30 full adders with half adder on each end
  FA_loopR1 :   for  col in 2 to 31 generate  --Instantiates 30 copies of Full Adder for
Row 1
      FA_R1_col : FullAdder10ns port map (XY(0,col), XY(1,col - 1), C(1,col),
C(1,col+1), S(2,col-1));
```

```
  End generate;

  HA_R1_C32: HalfAdder10ns port map (XY(1,31), C(1,32), S(2,32), S(2,31));  --Half
Adder at Row 1, Column 32
  HA_R1C1: HalfAdder10ns port map (XY(0,1), XY(1,0), C(1,2), P(1));  --Half Adder at
Row 1, Column 1
---- End Row 1

-----Rows 2 to 30
  FA_loopR2_30 : for row in 2 to 30 generate  --Instantiate Rows 2 thru 30
    FA_loopR2 :   for  col in 2 to 31 generate  --Instantiates 30 copies of Full Adder each
      FA_row_col : FullAdder10ns port map (S(row,col), XY(row,col-1), C(row,col),
C(row,col+1), S(row+1,col-1));
    End generate;
  End generate;

  FA_loopC32 : for  row in 2 to 30 generate  --Instantiates 29 copies of Full Adder for
Column 32
      FA_row_C32 : FullAdder10ns port map (S(row,32), XY(row,31), C(row,32),
S(row+1,32), S(row+1,31));
  End generate;
 ----end full adders 2 to 30

  --Half Adders (n half adders, 32 total)30 generated here, 2 added in row 1 elsewhere
  HA_loopC1 : for  row in 2 to 31 generate  --Instantiates 30 copies of Half Adder for
Column 1
      HA_row_C1 : HalfAdder10ns port map (S(row,1), XY(row,0), C(row,2), P(row));
  End generate;
  --end half adders column 1

---row 31 product outputs
  FA_loopR31: for col in 2 to 31 generate
          FA_31_col : FullAdder10ns port map(S(31,col),XY(31,col-
1),C(31,col),C(31,col+1),P(col+30));
  end generate;
  FA_R31_C32 : Fulladder10ns port map (S(31,32),XY(31,31),C(31,32),P(63),P(62));---
last full adder generates two product outputs
----end row 31

end Behavioral;
```

Following is the declaration of a Half Adder included as a code block in the Array Multiplier Quartus II project.

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

```
--Half Adder Entity Description
entity HalfAdder10ns is
  port(X, Y: in std_logic;
       Cout, Sum: out std_logic);
end HalfAdder10ns;

architecture eq2 of HalfAdder10ns is
begin
Cout <= X and Y after 10 ns;
Sum <= X xor Y after 10 ns;
end eq2;
```

Following is the declaration of a Full Adder included as a code block in the Array Multiplier Quartus II project.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity FullAdder10ns is
 Port (X, Y, Cin: in std_logic;
       Cout, Sum: out std_logic);
end FullAdder10ns;

architecture gate_level of FullAdder10ns is

begin

 Sum <= X XOR Y XOR Cin after 10 ns;
 Cout <= (X AND Y) OR (Cin AND X) OR (Cin AND Y) after 10 ns;

end gate_level;
```

Following are the results from the behavioral simulation of the above VHDL design and respective test bench;

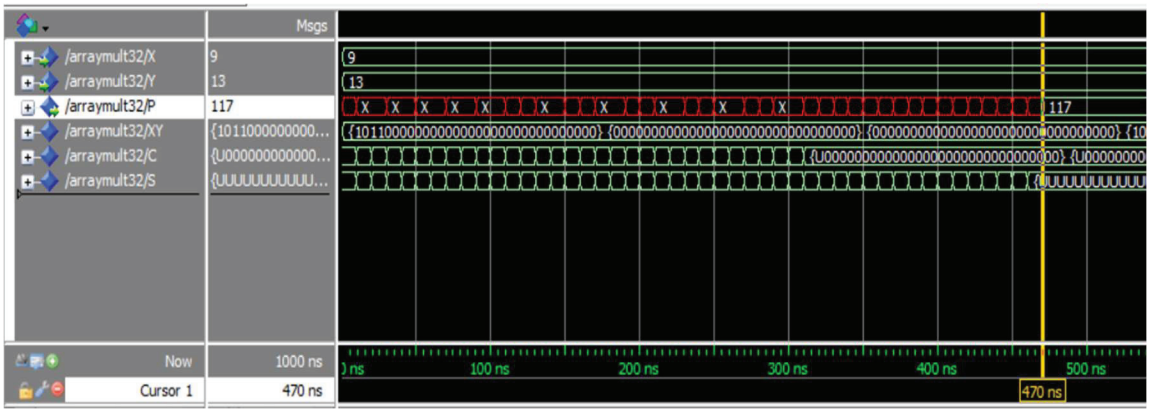


Figure 8. Array Multiplier Modelsim simulation for unsigned inputs

It was also observed that when the delay in the adders has been omitted, the array multiplier results were computed with an insignificant delay.

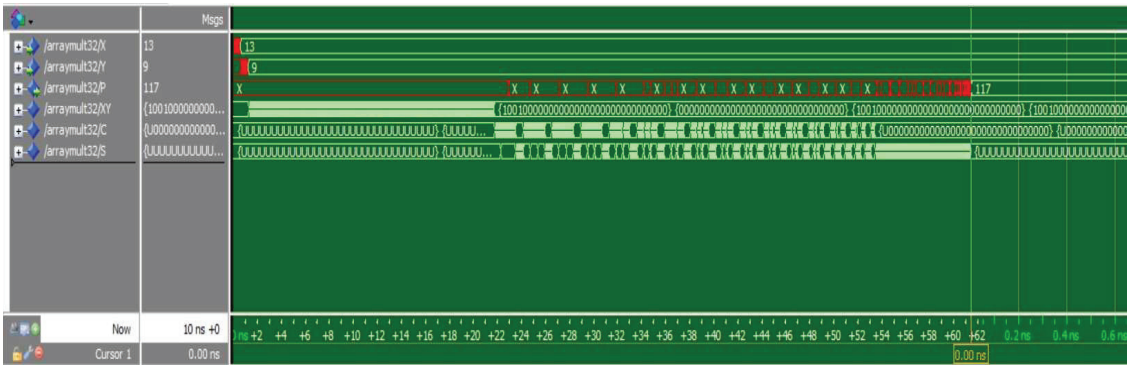


Figure 9. Array Multiplier Modelsim simulation for unsigned inputs – no adder delay

The original Array Multiplier is intended for unsigned inputs only. When a signed input is involved, it is observed that the algorithm still considers it as an unsigned input (usually a large decimal) and results in a product accordingly.

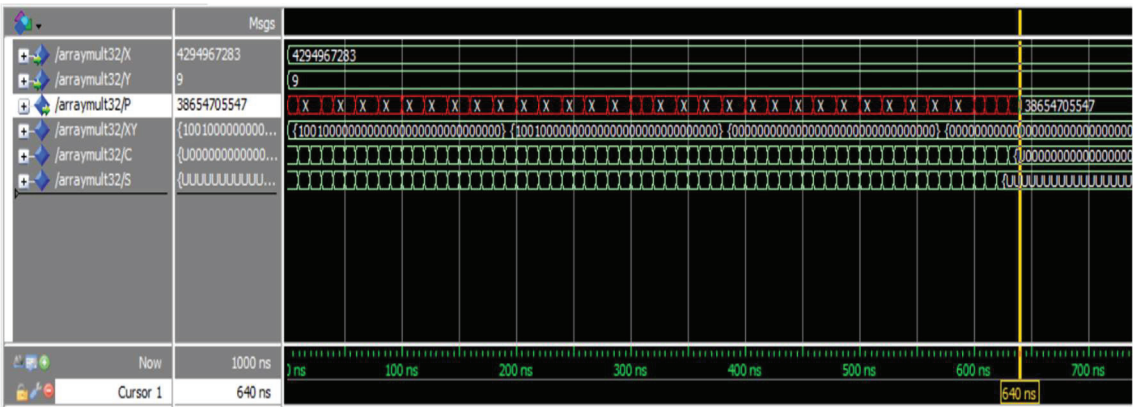


Figure 10. Array Multiplier Modelsim simulation for signed inputs

End of Chapter 3

CHAPTER 4
DESIGN AND IMPLEMENTATION

This report emphasizes on studying the performance of Radix-2 Booth's Multiplier and its comparison with the Array Multiplier. As a part of this study, the VHDL models of these multipliers were implemented on the FPGA hardware. Altera Prime Lite Quartus II version 18.1 was used for simulation and implementation of the models. DE10 Standard FPGA development board by Terasic Technologies was used for the hardware implementation of these VHDL models. The FPGA has been configured with these design modules using the Joint Test Action Group (JTAG) mode. JTAG is an industry-standard method for testing the hardware implementation of integrated designs and the interconnects on printed circuit boards (PCBs) that are implemented at the integrated circuit (IC).

4.1 About DE10 Standard FPGA

The DE10-Standard Development board includes the Intel Cyclone® V System-on-Chip (SoC) FPGA and an ARM Cortex 9 based Hard Processor Systems (HPS, processor built into the silicon as opposed to a "Soft" CPU (like NIOS) where the FPGA is configured to implement a CPU). This FPGA is recommended for exploring image processing on FPGAs by providing power for engineering development and prototyping. It is well suited for researchers looking for a low cost, entry level platform without compromising on resources available for the design and configuration.

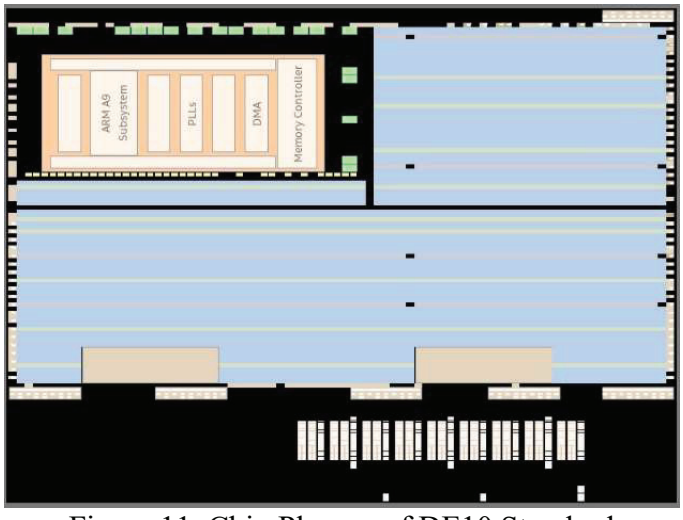


Figure 11. Chip Planner of DE10 Standard

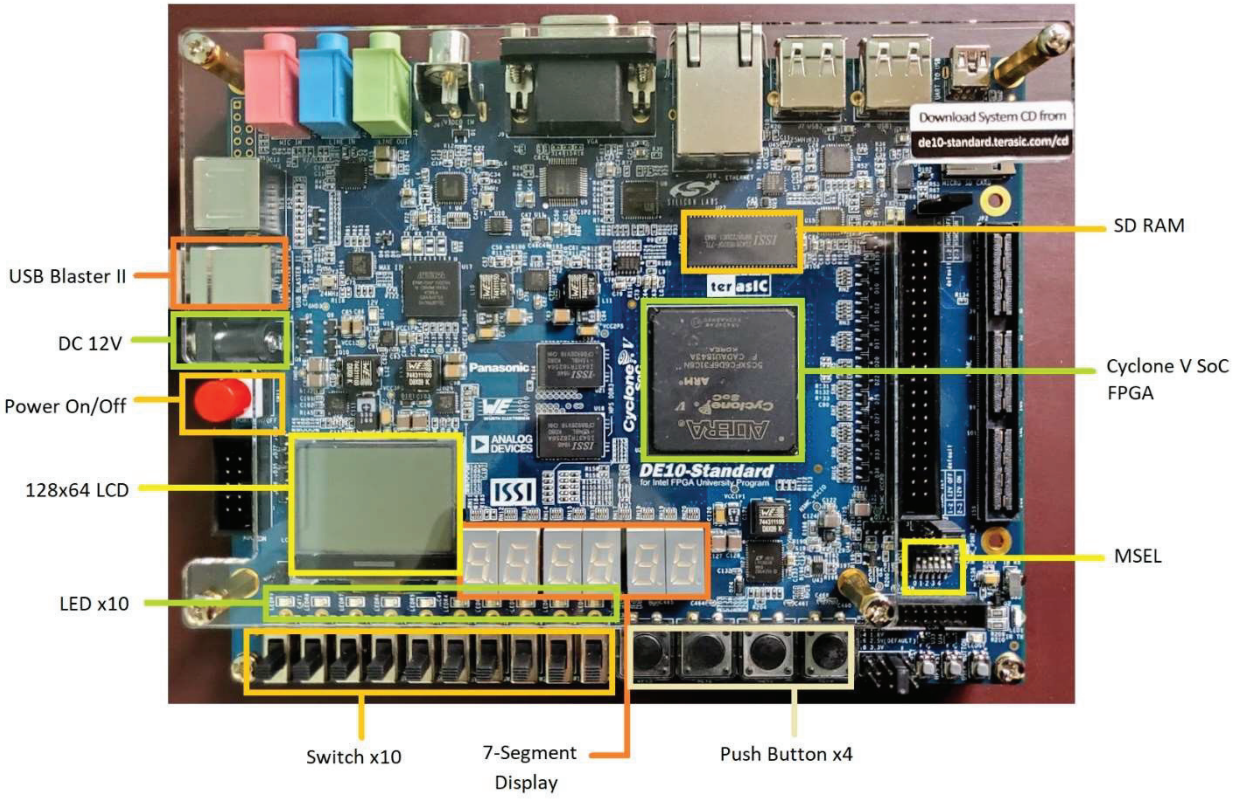


Figure 12. Terasic DE10 Standard FPGA Board

Resources	Characteristics
Logic Elements	110k
ALM	41910
Register	166036
Memory (Kb) M10K	5570
Memory (Kb) MLAB	621
Variable Precision DSP block	112
18x18 multiplier	224
FPGA PLL	6
HPS PLL	3
3 Gbps Transceiver	9
FPGA GPIO	288
HPS I/O	181
LVDS Transmitter	72
LVDS Receiver	72
PCIe Hard IP Block	2
FPGA Hard Memory Controller	1
HPS Hard Memory Controller	1
ARM Cortex-9 MPCore Processor	Dual-Core

Table 4. Intel Cyclone V SE 5CSXFC6D6F31C6N Specifications

4.2. FPGA Design Flow

The standard FPGA design flow begins with the creation of the digital circuit design using schematics or a hardware description language (HDL) such as Verilog or VHDL. This digital circuit design flow then proceeds through compilation, simulation, programming and implementation on the FPGA hardware.

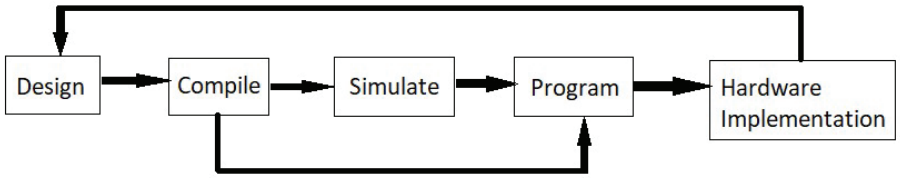


Figure 13. FPGA Design Flow

4.3 Booth's Multiplier Design and Implementation

Additional code blocks to enable the implementation of Radix-2 Booth's Multiplier algorithm are added to the initial design. These blocks include a top-level implementation code, a binary to BCD conversion code and a BCD to Hexadecimal 7 segment display code. Below are the VHDL codes for these blocks.

4.3.a Implementation:

```
library IEEE;
USE IEEE.std_logic_1164.ALL;
use IEEE.std_logic_unsigned.all;

entity implementation is
  port ( Clock_50 : in std_logic;
                        key : in std_logic; -- to enable st signal
                        SliderSwitch : in std_logic_vector (7 downto 0); -- to input the
multiplicand and the multiplier. assignment from sw[0] to sw[7].
                        seg71, seg72, seg73: out  STD_LOGIC_VECTOR (6 downto 0);
                        LEDR: out  STD_LOGIC );  -- to indicate done signal
end implementation;

Architecture implement_arch of implementation is

-- signals for the booth's multiplier
signal Mplier_32 : std_logic_vector (31 downto 0);
signal Mcand_32 : std_logic_vector (31 downto 0);
signal product : std_logic_vector (62 downto 0);
signal Done : std_logic;
```

```
-- signals for the 7 segment display
signal prod: std_logic_vector (7 downto 0):= "00000000";
signal bcd_1, bcd_2, bcd_3 : std_logic_vector (3 downto 0);

component Boothsmult is
  port ( clk, st: in std_logic;
          Mplier, Mcand : in std_logic_vector (31 downto 0);
          Done : out std_logic;
          Product : out std_logic_vector (62 downto 0)   );
end component;

component hex_seg7 is
   Port (product : in  STD_LOGIC_VECTOR (3 downto 0);
         seg7 : out  STD_LOGIC_VECTOR (0 to 6)      );
end component;

component binary_bcd is
  Port ( binary   : in  std_logic_vector (7 downto 0);
     hundreds : out std_logic_vector (3 downto 0);
     tenths     : out std_logic_vector (3 downto 0);
     unit     : out std_logic_vector (3 downto 0)   );
end component ;

begin

Mplier_32 <= "11111111111111111111111111111111" & SliderSwitch(3 downto 0);
Mcand_32 <= "11111111111111111111111111111111" & SliderSwitch(7 downto 4); -- for
demonstrating implementation of a negative mcand combination
--Mcand_32 <= "00000000000000000000000000000000" & SliderSwitch(7 downto 4); -- for
demonstrating implementation of a positive mcand combination

-- booth's multiplier implementation
Booth_mult: Boothsmult port map(Clock_50, key, Mplier_32, Mcand_32, Done,
product);

-- conversion of binary product to bcd for 7 segment display
prod <= product(7 downto 0);
binary_bcd1: binary_bcd port map(prod,bcd_1, bcd_2, bcd_3);

-- hex to 7 segment display
hex_seg71 : hex_seg7 port map (bcd_3, seg71);
hex_seg72 : hex_seg7 port map (bcd_2, seg72);
hex_seg73 : hex_seg7 port map (bcd_1, seg73);

end implement_arch;
```

4.3.b Binary to BCD conversion

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity binary_bcd is
  Port ( binary   : in  std_logic_vector (7 downto 0);
    hundreds : out std_logic_vector (3 downto 0);
    tenths     : out std_logic_vector (3 downto 0);
    unit     : out std_logic_vector (3 downto 0)   );
end binary_bcd;

architecture Behavioral of binary_bcd is
begin
bin_bcd : process (binary)
variable shift : unsigned(19 downto 0) := "00000000000000000000"; -- variable register
for storing bits
 -- Alias for parts of variable shift register
    alias num is shift(7 downto 0);
    alias unity is shift(11 downto 8);
    alias tenth is shift(15 downto 12);
    alias hundred is shift(19 downto 16);

begin
    num := unsigned(binary);
    unity := X"0";
    tenth := X"0";
    hundred := X"0";

-- Loop eight times. if the numerical value of the alias is greater than 5, then per shift and
add algorithm, alias is incremented by 3
-- and then the contents of the shift register are shifted to the left by 1 place.

for i in 1 to num'Length loop

if unity >= 5 then
unity := unity + 3;
end if;

if tenth >= 5 then
tenth := tenth + 3;
end if;

if hundred >= 5 then
hundred := hundred + 3;
end if;
```

```
-- contents of the shift register are shifted to the left by 1 place
        shift := shift_left(shift, 1);
      end loop;
 -- load contents of alias to the output registers
      hundreds <= std_logic_vector(hundred);
      tenths     <= std_logic_vector(tenth);
      unit     <= std_logic_vector(unity);
   end process;
end Behavioral;
```

4.3.c Hexadecimal to 7 Segment Display

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity hex_seg7 is
    Port (product : in  STD_LOGIC_VECTOR (3 downto 0);
          seg7 : out  STD_LOGIC_VECTOR (0 to 6)  );
end hex_seg7;

architecture Behavioral of hex_seg7 is
begin
process (product)
BEGIN
    case product is
       when "0000"=> seg7 <="1000000";  -- '0'
       when "0001"=> seg7 <="1111001";  -- '1'
       when "0010"=> seg7 <="0100100";  -- '2'
       when "0011"=> seg7 <="0110000";  -- '3'
       when "0100"=> seg7 <="0011001";  -- '4'
       when "0101"=> seg7 <="0010010";  -- '5'
       when "0110"=> seg7 <="0000010";  -- '6'
       when "0111"=> seg7 <="1111000";  -- '7'
       when "1000"=> seg7 <="0000000";  -- '8'
       when "1001"=> seg7 <="0011000";  -- '9'
       when "1010"=> seg7 <="0001000";  -- 'A'
       when "1011"=> seg7 <="0000011";  -- 'b'
       when "1100"=> seg7 <="1000110";  -- 'C'
       when "1101"=> seg7 <="0100001";  -- 'd'
       when "1110"=> seg7 <="0000110";  -- 'E'
       when "1111"=> seg7 <="0001110";  -- 'F'
                     when others =>  NULL;
end case;
end process;
end Behavioral;
```

Upon initial compiling of the top-level module and the code blocks, the pin assignment is completed per the specifications listed in the DE10 Standard User Manual version March 20, 2018.

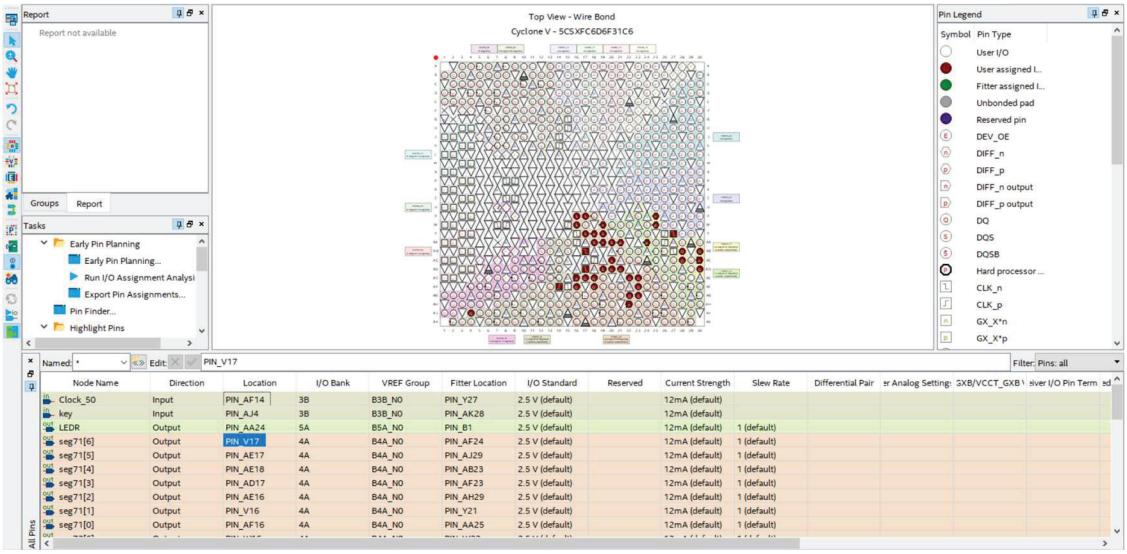


Figure 14. Booth's Multiplier Pin Assignment

The project is finally compiled to verify the pin assignment and then the FPGA hardware is configured in the JTAG mode using the Quartus II Programmer interface.

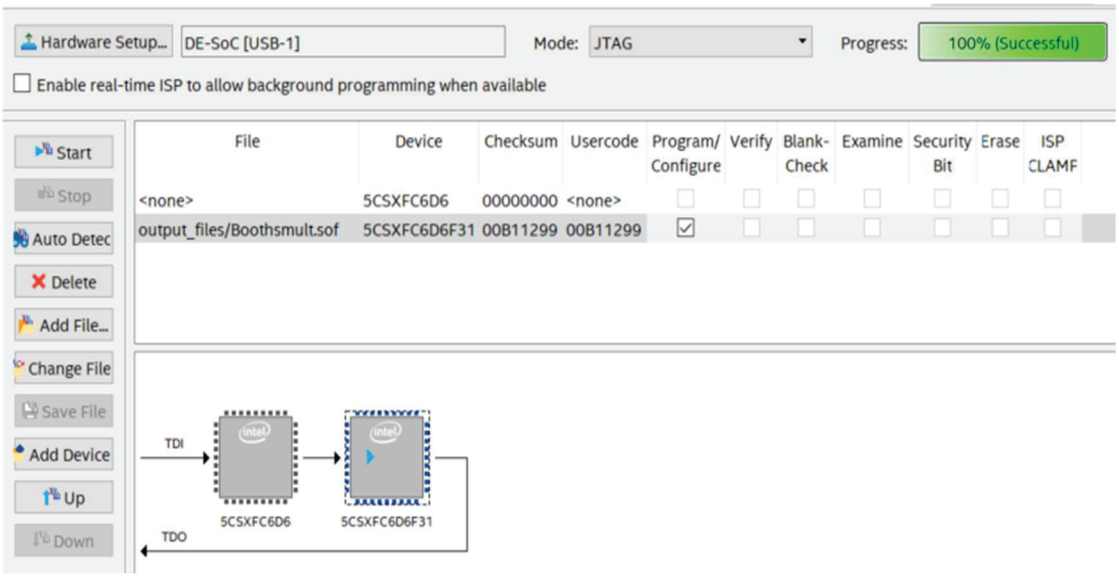
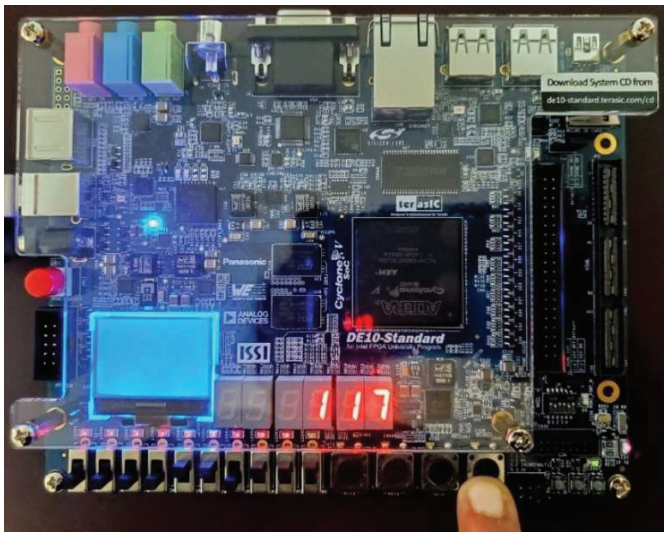


Figure 15. Quartus II Programmer interface for Booth's Multiplier

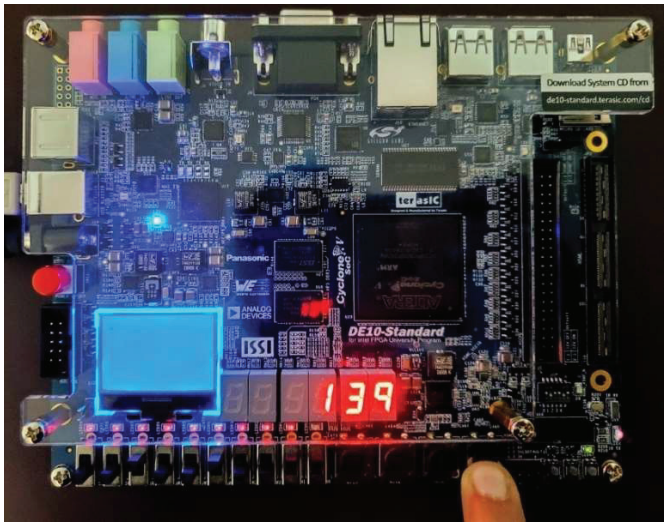
We discussed earlier that Booth's Multiplier is capable of processing unsigned as well as signed inputs. To demonstrate this characteristic, a negative multiplier and multiplicand combination has been input to the algorithm and the FPGA was configured accordingly.

In the first case, multiplier = $(-13)_{10}$ = $(11111111111111111111111111110011)_2$
Multiplicand = $(-9)_{10}$ = $(11111111111111111111111111110111)_2$
Result = $(117)_{10}$

In the second case, multiplier = $(-13)_{10}$ = $(11111111111111111111111111110011)_2$
Multiplicand = $(9)_{10}$ = $(00000000000000000000000000001001)_2$
Signed 2's complement Result = $(-117)_{10}$
Unsigned Result = $(139)_{10}$



Case 1



Case 2

Figure 16. Booth's Multiplier Algorithm Hardware Implementation

4.4 Array Multiplier Design and Implementation

Additional code blocks to enable implementation of the Array Multiplier algorithm are added to the initial design. These blocks include a top-level implementation code, a binary to BCD conversion code and a BCD to Hexadecimal 7 segment display code. Below are the VHDL codes for these blocks.

4.4.a Implementation:

```
library IEEE;
USE IEEE.std_logic_1164.ALL;
use IEEE.std_logic_unsigned.all;

entity implementation is
  port ( SliderSwitch : in std_logic_vector (7 downto 0); -- to input the multiplicand and
the multiplier. assignment from sw[0] to sw[7].
                              seg71, seg72, seg73: out  STD_LOGIC_VECTOR (6 downto 0)
);
end implementation;

Architecture implement_arch of implementation is

-- signals for the array multiplier
signal Mplier_32 : std_logic_vector (31 downto 0);
signal Mcand_32 : std_logic_vector (31 downto 0);
signal product : std_logic_vector (63 downto 0);

-- signals for the 7 segment display
signal prod: std_logic_vector (7 downto 0):= "00000000";
signal bcd_1, bcd_2, bcd_3 : std_logic_vector (3 downto 0);

component ArrayMult32 is
  port(X, Y: in std_logic_vector(31 downto 0);  --32-bit inputs
       P: out std_logic_vector(63 downto 0));  --64-bit output
end component;

component hex_7seg is
    Port (product : in  STD_LOGIC_VECTOR (3 downto 0);
          seg7 : out  STD_LOGIC_VECTOR (0 to 6)      );
end component;

component binary_bcd is
   Port ( binary   : in  std_logic_vector (7 downto 0);
      hundreds : out std_logic_vector (3 downto 0);
      tenths     : out std_logic_vector (3 downto 0);
```

```
    unit     : out std_logic_vector (3 downto 0)   );
end component ;

begin
Mplier_32 <= "00000000000000000000000000000000" & SliderSwitch(3 downto 0);
Mcand_32 <= "00000000000000000000000000000000" & SliderSwitch(7 downto 4);

-- Array Multiplier Implementation
ArrayMult: ArrayMult32 port map (Mplier_32,Mcand_32,product);

-- conversion of binary product to bcd for 7 segment display

prod <= product(7 downto 0);
binary_bcd1: binary_bcd port map(prod,bcd_1, bcd_2, bcd_3);

-- hex to 7 segment display

hex_seg71 : hex_7seg port map (bcd_3, seg71);
hex_seg72 : hex_7seg port map (bcd_2, seg72);
hex_seg73 : hex_7seg port map (bcd_1, seg73);

end implement_arch;
```

4.4.b Binary to BCD conversion

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity binary_bcd is
  Port ( binary   : in  std_logic_vector (7 downto 0);
    hundreds : out std_logic_vector (3 downto 0);
    tenths     : out std_logic_vector (3 downto 0);
    unit     : out std_logic_vector (3 downto 0)   );
end binary_bcd;

architecture Behavioral of binary_bcd is
 begin
bin_bcd : process (binary)
-- variable register for storing bits
variable shiftreg : unsigned(19 downto 0) := "00000000000000000000";
-- Alias for parts of variable shift register
    alias num is shiftreg (7 downto 0);
    alias unity is shiftreg (11 downto 8);
    alias tenth is shiftreg (15 downto 12);
    alias hundred is shiftreg (19 downto 16);
```

```
begin
     num := unsigned(binary);
     unity := X"0";
     tenth := X"0";
     hundred := X"0";

-- Loop eight times. if the numerical value of the alias is greater than 5, then per shift and
add algorithm, alias is incremented by 3
-- and then the contents of the shift register are shifted to the left by 1 place.

for i in 1 to num'Length loop
if unity >= 5 then
   unity := unity + 3;
end if;
if tenth >= 5 then
   tenth := tenth + 3;
end if;
if hundred >= 5 then
  hundred := hundred + 3;
end if;
-- contents of the shift register are shifted to the left by 1 place
     shiftreg := shift_left(shiftreg, 1);
     end loop;
-- load contents of alias to the output registers
     hundreds <= std_logic_vector(hundred);
     tenths    <= std_logic_vector(tenth);
     unit    <= std_logic_vector(unity);
end process;
end Behavioral;
```

4.4.c Hexadecimal to 7 Segment Display

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity hex_7seg is
   Port (product : in  STD_LOGIC_VECTOR (3 downto 0);
         seg7 : out  STD_LOGIC_VECTOR (0 to 6)        );
end hex_7seg;

architecture Behavioral of hex_7seg is
begin

--'a' corresponds to MSB of seg7 and 'g' corresponds to LSB of seg7.
process (product)
BEGIN
   case product is
```

```
        when "0000"=> seg7 <="1000000";  -- '0'
        when "0001"=> seg7 <="1111001";  -- '1'
        when "0010"=> seg7 <="0100100";  -- '2'
        when "0011"=> seg7 <="0110000";  -- '3'
        when "0100"=> seg7 <="0011001";  -- '4'
        when "0101"=> seg7 <="0010010";  -- '5'
        when "0110"=> seg7 <="0000010";  -- '6'
        when "0111"=> seg7 <="1111000";  -- '7'
        when "1000"=> seg7 <="0000000";  -- '8'
        when "1001"=> seg7 <="0011000";  -- '9'
        when "1010"=> seg7 <="0001000";  -- 'A'
        when "1011"=> seg7 <="0000011";  -- 'b'
        when "1100"=> seg7 <="1000110";  -- 'C'
        when "1101"=> seg7 <="0100001";  -- 'd'
        when "1110"=> seg7 <="0000110";  -- 'E'
        when "1111"=> seg7 <="0001110";  -- 'F'
                    when others =>  NULL;
    end case;
end process;
end Behavioral;
```

Upon initial compiling of the top-level module and the code blocks, the pin assignment is completed with reference to the DE10 Standard User Manual version March20, 2018.

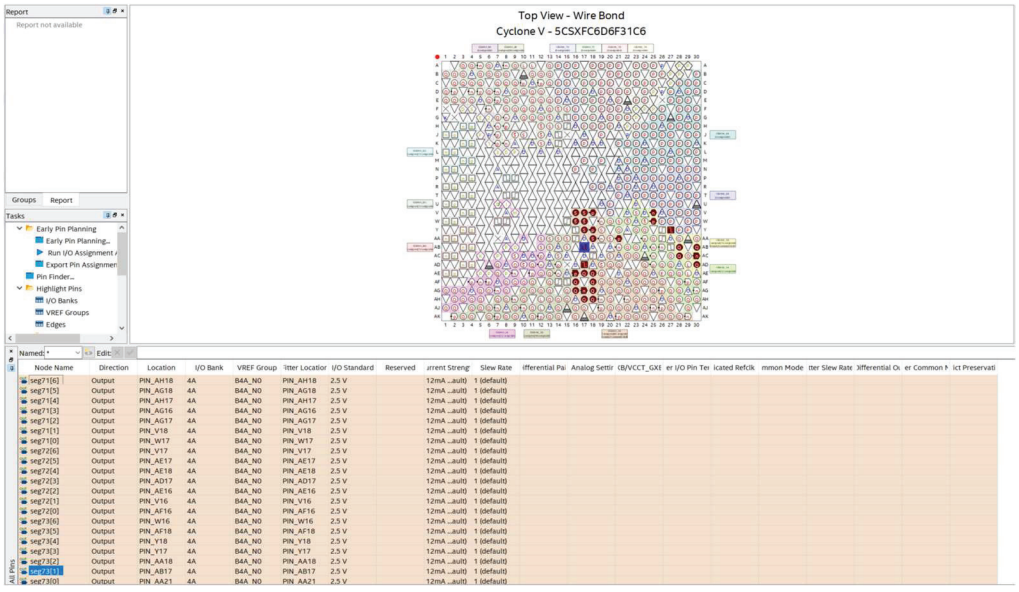


Figure 17. Booth's Multiplier Pin Assignment

The project is then final compiled for the pin assignment and then the FPGA hardware is configured in the JTAG mode using the Quartus II Programmer interface.

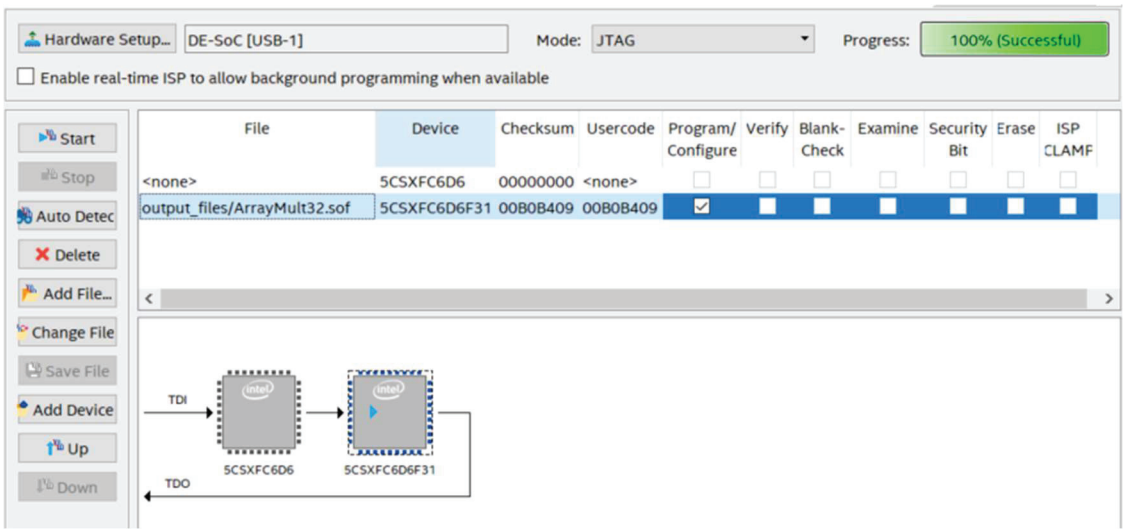


Figure 18. Quartus II Programmer interface for Array Multiplier

We discussed earlier that the Array Multiplier algorithm is capable of processing only unsigned inputs and reviewed the simulation results when a signed as well as an unsigned input combination is used in the algorithm. The below case shows the implementation of two positive inputs.

Multiplier = $(+13)_{10}$ = $(00000000000000000000000000001101)_2$
Multiplicand = $(+5)_{10}$ = $(00000000000000000000000000000101)_2$
Result = $(65)_{10}$

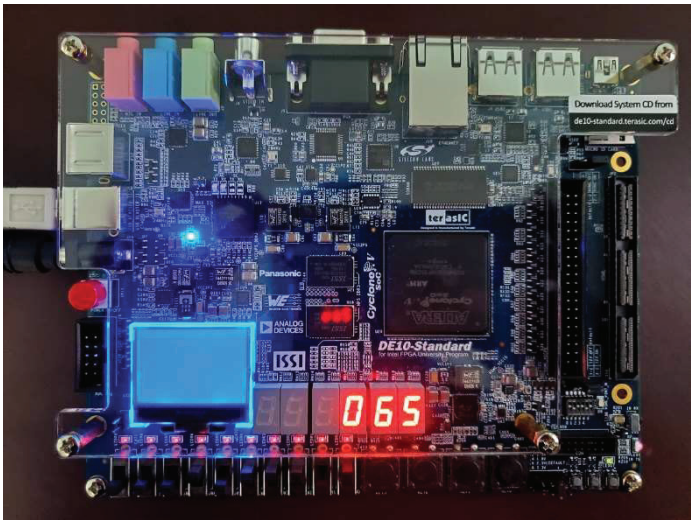


Figure 19. Array Multiplier Algorithm Hardware Implementation

End of Chapter 4

CHAPTER 5
CONCLUSION

The primary objective of this thesis has been to understand the functioning of binary multipliers and design their VHDL models to analyze and compare the performance of individual multipliers. The role of these multipliers has been realized to be crucial when considering the grand scheme of their application. Several digital signal processing applications are based on the multiplication process. Hence the efficiency of these signal processing applications greatly relies on the performance of these multiplication algorithms. The goal in designing such critical blocks will be to ensure their minimal ultimate space utilization on the FPGA.

The subject multipliers of this report were the Booth's Multiplication Algorithm and the Array Multiplication Algorithm. The VHDL models were built considering a 32x32 bit input to these multipliers. Altera Prime Lite Quartus II version 18.1 was used for simulation and implementation of the models. DE10 Standard FPGA development board by Terasic Technologies was used for the hardware implementation of these VHDL models.

Logic utilization is calculated by estimating how many half-ALMs are needed to fit a design and is a good representation of how full a device is. The logic utilization for the Booth's Multiplication algorithm has been realized to be 3% of the total logic utilization of the Array Multiplication algorithm. Combinational ALUT usage is the actual number of completely or partially used half-ALMs in the design after logic analysis and synthesis. The Booth's Multiplier needed only 2% of the total ALUTs needed for an Array Multiplier. It was also observed that as the range of the inputs increases, the complexity of implementing an Array multiplier increases as a result of an increase in the number of levels of adders needed to accomplish the product result. See Figure 22 and 23 for a representation on the implementation and design space required by Booth's and Array multipliers respectively that was realized during this study. Therefore, the radix-2 type Booth's Multiplication algorithm considered for this study proved to be more efficient than the Array multiplier.

	Resource BoothsMult	Usage
1	Estimate of Logic utilization (ALMs needed)	56
2		
3	⌄ Combinational ALUT usage for logic	51
1	-- 7 input functions	0
2	-- 6 input functions	4
3	-- 5 input functions	4
4	-- 4 input functions	34
5	-- <=3 input functions	9
4		
5	Dedicated logic registers	106
6		
7	I/O pins	130
8		
9	Total DSP Blocks	0
10		
11	Maximum fan-out node	clk~input
12	Maximum fan-out	106
13	Total fan-out	898
14	Average fan-out	2.15

	Resource ArrayMult	Usage
1	Estimate of Logic utilization (ALMs needed)	1719
2		
3	⌄ Combinational ALUT usage for logic	2590
1	-- 7 input functions	0
2	-- 6 input functions	848
3	-- 5 input functions	360
4	-- 4 input functions	190
5	-- <=3 input functions	1192
4		
5	Dedicated logic registers	0
6		
7	I/O pins	128
8		
9	Total DSP Blocks	0
10		
11	Maximum fan-out node	Y[0]~input
12	Maximum fan-out	83
13	Total fan-out	10436
14	Average fan-out	3.67

Figure 20. Resource Usage Summary – Booth's vs Array Multiplier Algorithm

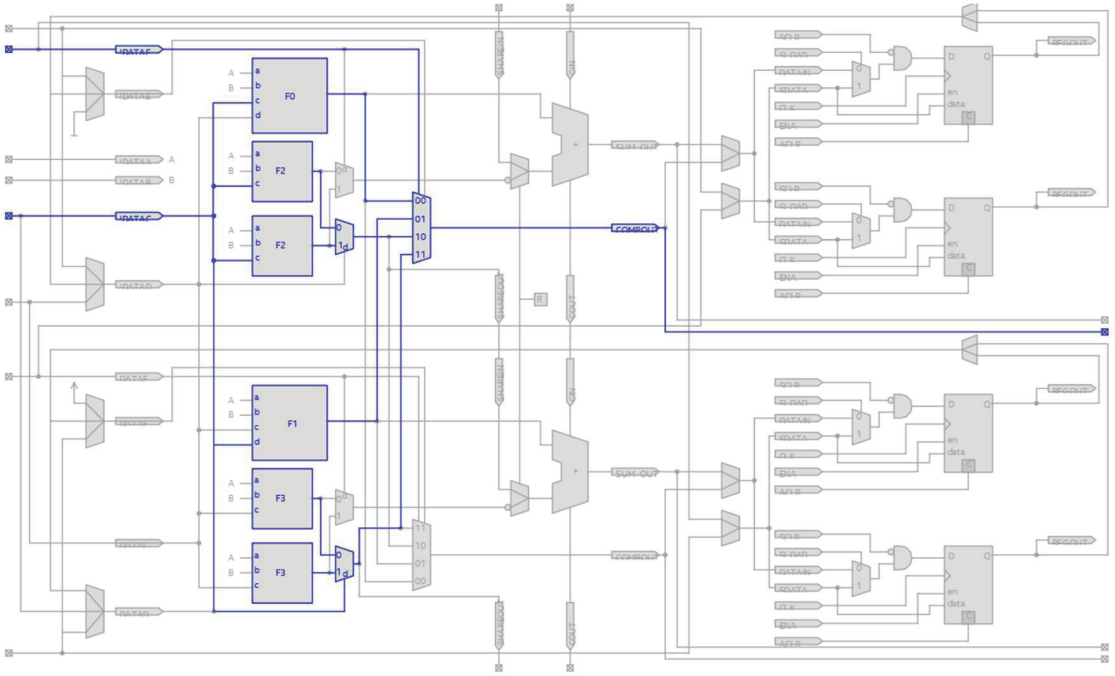


Figure 21. Chip Planner - Array Multiplier Algorithm

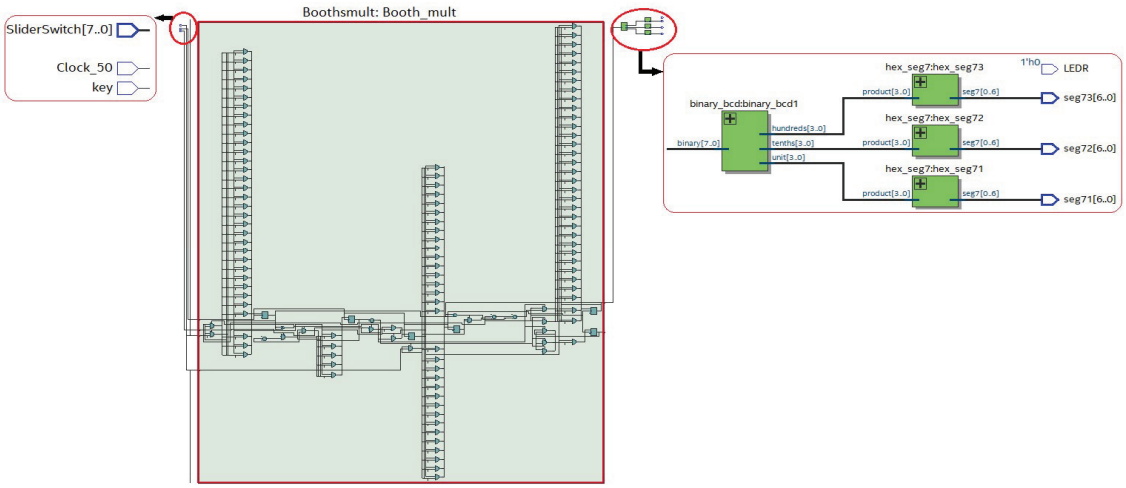


Figure 22. RTL View – Booth's Multiplier Implementation

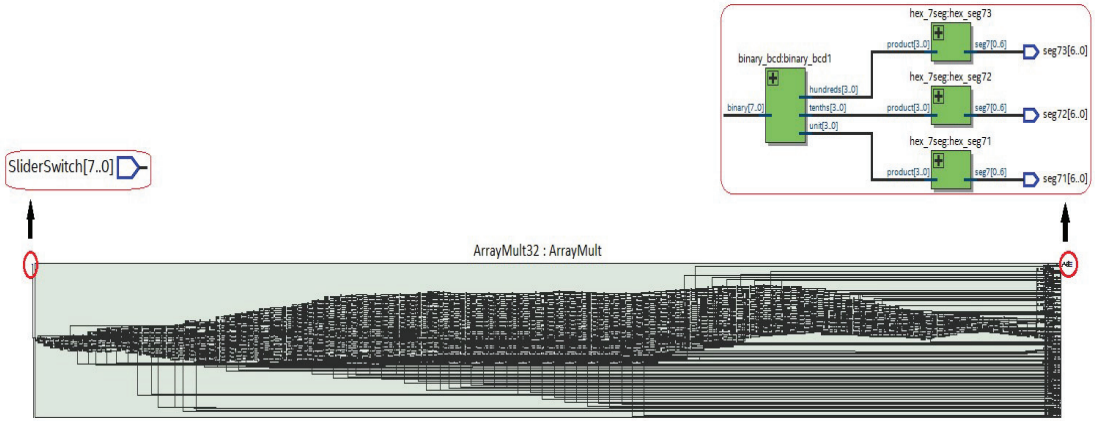


Figure 23. RTL View – Array Multiplier Implementation

Based on the understanding on the performance of Radix-4, Radix-8 and Radix-16 type Booth's Multipliers, it can be assumed that the efficiency of signal processing applications would be greatly improved as a result of lesser logic utilization. It is proposed that further study and implementation of higher radix order Booth's multipliers would benefit the efficiency of their applications.

Considering the outcome of this study and the assumptions made with the understanding from this study, it can be noted that the modified Booth's Algorithm holds the future of designing the signal processing applications with a promise of increased efficiency, lesser power consumption and lesser space utilization.

End of Chapter 5

BIBLIOGRAPHY

1. Quartus II Handbook, Edition 2014.12.15. Retrieved from: https://courses.cs.washington.edu/courses/cse467/15wi/docs/Quartus_II_Handbook.pdf
2. DE10-Standard User Manual, by Terasic Technologies, Edition March 20, 2018. Retrieved from: https://www.intel.com/content/dam/altera-www/global/en_US/portal/dsn/42/doc-us-dsnbk-42-5505271707235-de10-standard-user-manual-sm.pdf
3. DE10-Standard My First FPGA, by Terasic Technologies, Edition February 15, 2017. Retrieved from: https://rocketboards.org/foswiki/pub/Documentation/DE10Standard/DE10-Standard_My_First_Fpga.pdf
4. DE10-Standad Getting Started Guide, Edition April 20, 2017. Retrieved from: https://rocketboards.org/foswiki/pub/Documentation/DE10Standard/DE10-Standard_Getting_Started_Guide.pdf
5. Intel® Quartus® Prime Pro Edition User Guide, UG-20140 | Edition 2019.09.30. Retrieved from: https://www.intel.com/content/www/us/en/programmable/documentation/spj1513986956763.html
6. QUARTUS PRIME INTRODUCTION USING VHDL DESIGNS, Edition March 2018.

7. Bewick, Gary & Flynn, Michael. (1970). Fast Multiplication: Algorithms And Implementation. Retrieved from: https://www.researchgate.net/publication/2575879_Fast_Multiplication_Algorithms_And_Implementation
8. Laxman S, Darshan Prabhu R, Mahesh S Shetty, Mrs. Manjula BM, Dr. Chirag Sharma "FPGA Implementation of Different Multiplier Architectures" ISSN 2250-2459, Volume 2, Issue 6, June 2012)
9. Arvind Chakrapani, S.ShanmugaPriya, P.Thenmozhi, R.Vishnu priya, N.Yashika "Simulation Analysis of Binary Multipliers used in the MAC Unit of Digital Signal Processors", International Journal of Pure and Applied Mathematics, ISSN: 1311-8080 (printed version); ISSN: 1314-3395 (on-line version)
10. Snehal R Deshmukh, Dinkar L Bhombe, "Performance Comparison of Different Multipliers using Booth Algorithm", International Journal of Engineering Research & Technology(IJERT) Vol. 3 Issue 2, February – 2014

End of Document