Concordia University
Engineering and Computer Science

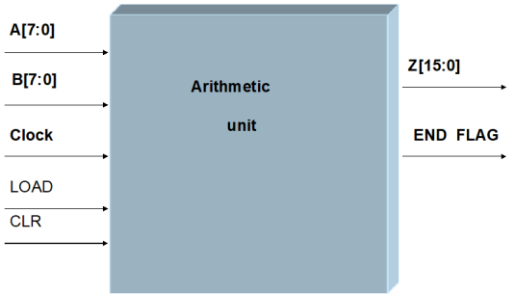
Fall 2021

COEN 6501: Digital Design and Synthesis

Dr. Marwan Ammar

Project Report

Arithmetic Unit Implementation which is capable of calculating **Z = ¼ [A²*B] + 1**



Team Members

Abdul Aziz Midthuru (**40163297**)

Abhishek Madhu **(40193107)**

Ragul Nivash Rangasamy Sekar **(40169564**)

Shivendra Arulalan **(40197455)**

Thenmozhie Rajan (**40192527**)

ABSTRACT

An arithmetic unit which performs the operation Z = ¼ [A^2 * B] + 1 is implemented in this project. A multiplier, a shift register and a ripple carry adder are used to design this arithmetic unit. A and B are two 8-bit unsigned numbers and Z is 42 bit and 16 bit. Two method of implementation is done. One is implemented with Wallace tree method that include RCA, half adder and full adder and other is implemented with Wallace tree method with CSA, full adder, multiplexer where pipelining and expansion of the method for 16-bit operand.

First the two 8-bit numbers are multiplied with ¼ using shifting methodology then 1 is added to the output of shift register. This project is implemented by using basic logic gates, 4:1 and 2:1 multiplexers, registers, D-FlipFlop. When multiplexers are used the area of the unit is lesser than the area of the unit which is implemented using logic gates. This project has been implemented using structural modelling in VHDL. The program is simulated using ModelSim. RTL schematics are obtained from RTL synthesis. The area and timing report is obtained from XILINX ISE.

Contents

Table Of Figures

Tables

ACKNOWLEDGEMENT

On the actual start of this report, we would like to expand our true commitment towards every one of the personages who have helped us in this undertaking. Without their dynamic direction, help, collaboration and consolation, we would have not gained ground in the undertaking. We take this opportunity to express our true gratitude to **Dr. Marwan Ammar**, who instructed this course Digital Design and Synthesis and offered us a chance to figure out how to design the digital device, understanding the design algorithms and VHDL inside and out. His direction all through the course time, his lecture slides and the materials he gave in the site helped us a great deal to finish this project effectively. We are consistently appreciative to Concordia University and its labs without which we couldn't do our task report. We additionally wish to stretch out our respects to our teaching assistant **Vivek Bansal** for assisting us with getting hands on tools utilized and for his interminable consolation and consistent help to us towards the finishing of the project.

1.INTRODUCTION

The objective of this project is to design an arithmetic unit which is capable of calculating Z = ¼ [A2 * B] + 1. The arithmetic unit consists of two multipliers, where the first multiplier has two input 8-bit numbers and generates the result of 16-bit number which is an input to the second multiplier which is then multiplied with an 8-bit number that generates the result of 24-bit number. The resultant 24-bit output number is shifted by two bits and then the shifted output is added by one. At the point when the LOAD signal changes from HIGH to LOW the input values A and B are stacked into the registers RA and RB. The final output shows up at the port Z as displayed in the figure. The CLEAR sign is utilized to reset all registers to zero. The clock signal is positive edge set off signal. The calculation begins with LOAD signal and finishes with END FLAG signal.

Multiplication is an important fundamental function in arithmetic logic operation. Computational performance of a DSP system is limited by its multiplication performance and since, multiplication dominates the execution time of most DSP algorithms; therefore high-speed multiplier is much desired. Currently, multiplication time is still the dominant factor in determining the instruction cycle time of a DSP chip. With an ever-increasing quest for greater computing power on battery-operated mobile devices, design emphasis has shifted from optimizing conventional delay time area size to minimizing power dissipation while still maintaining the high performance.

2.Project description

The goal of this project is to design an arithmetic unit capable of calculating **Z = ¼ [A² * B] + 1**. The unit will receive the operands A, and B 8-bit un***signed*** numbers. A 1 to 0 transition at the LOAD pin will latch the operands into internal register RA and RB. The unit outputs the results in 16 bit register RZ output port. Each calculation starts with a LOAD signal and ends with an END_FLAG signal.

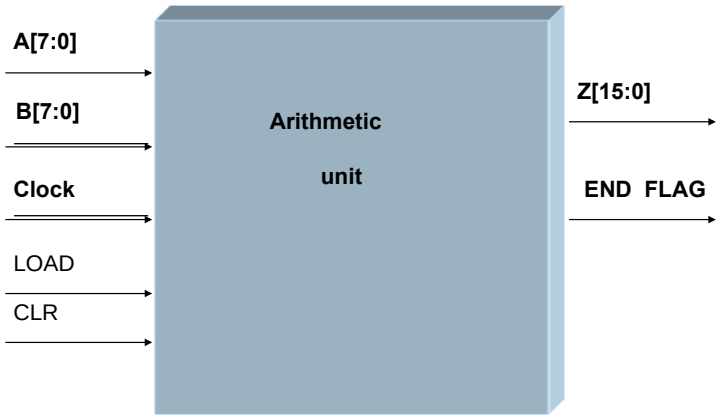


Figure 1 Black Box

2.1 Requirements:

Table 2 is a summary of the design requirements. There are certain behaviours on the inputs that were not specified in the COEN 6501 project description and would allow for different interpretation on these signals.

Table 1 Design Requirements

Requirement number	Description	Comment
R1	The choice of type of multiplier, is left to the student	
R2	The design shall be structural.	

R3	The operands A and B are latched into register RA and RB when LOAD signal transit from high to low.	
R4	The CLEAR signal will clear all registers to '0'.	
R5	The 16-bit product shall be loaded into the 16-bit Z port.	
R6	The unit performs the arithmetic operation until END_FLAG becomes high.	

Signal Specifications

A0-A7: Unsigned 8bit Operand A

B0-B 7: Unsigned 8bit Operand B

Z0-Z15: Signed Output

CLR: Clears selected registers

LOAD: Loads Operand into internal registers

CLK: Clock input

END_FLAG: Indicates end of operation

2.2. Detailed Block and Proposed Algorithm:

The steps followed to implement the arithmetic unit as follows,

1. Input the 8-bit unsigned number
2. Use registers to load 8-bit value until the load becomes high
3. The 8-bit unsigned number is now fed to multiplier to find the multiplication of two unsigned numbers. Initially find all the partial products of the multiplication. Later, add all the partial products to find the final result. The 16-bit output is again multiplied with 8-bit number. Two different ways are implemented which has explained in detail in further sessions.
4. After squaring, the result is then fed to adder which performs addition of a result with '1'.
5. The resulted 16-bit is now stored in 16-bit register which will release the output when the load is high. End Flag is set once the operation ends.

3. DESIGN AND ANALYSIS OF COMPONENTS

We use the basic logic gates which are AND, OR, NOT and XOR gates. Here 0 is called "false" and 1 is called "true", the gate acts in the same way as the logical operators. The following illustration and table show the circuit symbol and logic combinations logic gates.
List of components used are:

- Basic Logic Gates.
- Half Adders.
- Full Adders.
- Ripple Carry Adders.
- Carry Save Adder.

3.1 Basic Logic Gates:

3.1.1 AND

The AND gate is a basic digital logic gate that implements logical conjunction (∧) from mathematical logic. A HIGH output (1) results only if all the inputs to the AND gate are HIGH (1). If none or not all inputs to the AND gate are HIGH, LOW output results. The function can be extended to any number of inputs. The AND gate can be implemented in VHDL using the statement "output<= input1 and input2". The truth table is presented as below. It behaves according to the truth table. The and gate is tested with a test bench based on the truth table and the simulation is show below.

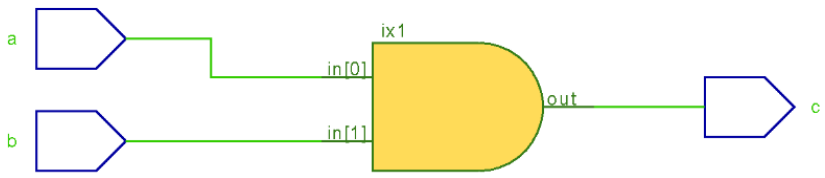


Figure 2 RTL Schematic for AND gate

Truth table of AND Gate:

A	B	C= A.B
0	0	0
0	1	0
1	0	0

1	1	1

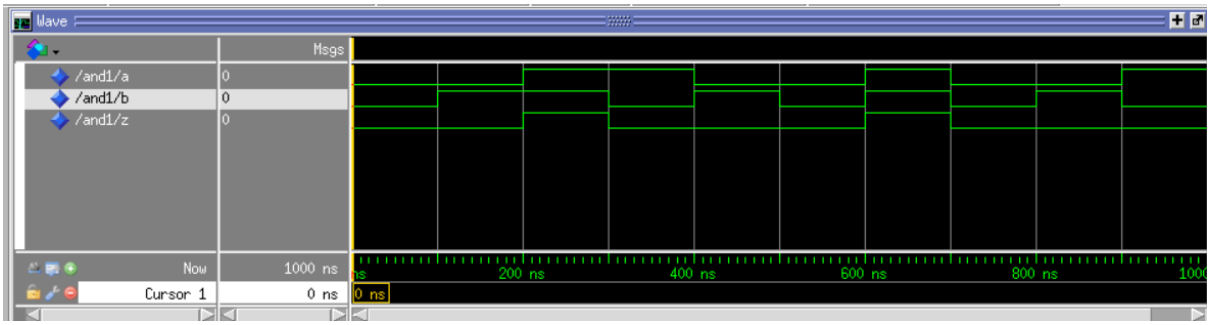


Figure 3 Simulation of AND gate

3.1.2 OR

The OR gate is a digital logic gate that implements logical disjunction (∨) from mathematical logic. A HIGH output (1) results if one or both the inputs to the gate are HIGH (1). If neither input is high, a LOW output (0) results. In another sense, the function of OR effectively finds the maximum between two binary digits, just as the complementary AND function finds the minimum. The OR gate can be implemented in VHDL using the statement "output<= input1 or input2". It behaves according to the truth table presented. The OR gate is tested with a test bench based on the truth table and the simulation is shown below.

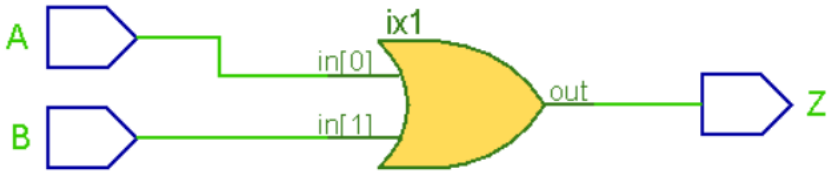


Figure 4 RTL Schematic for OR gate

Truth table of OR gate

A	B	C= A or B
0	0	0
0	1	1

1	0	1
1	1	1

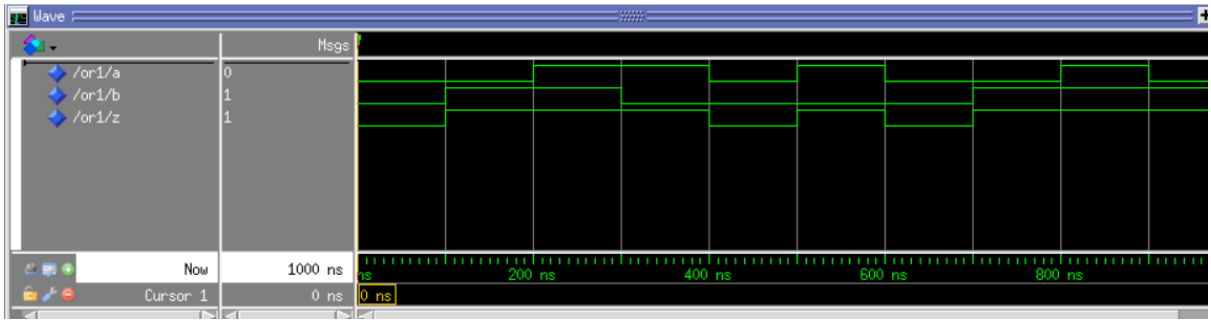


Figure 5 Simulation Result of OR Gate

3.1.3 XOR

XOR is an abbreviation for "Exclusively-OR". The simplest XOR gate is a two-input digital circuit that outputs a logical "1" if the two input values differ, i.e., its output is a logical "1" if either of its inputs are 1, but not at the same time (exclusively). The XOR gate can be implemented in VHDL using the statement "output<= input1 **xor** input2". The XOR gate is tested with a test bench based on the truth table and the simulation is shown below.

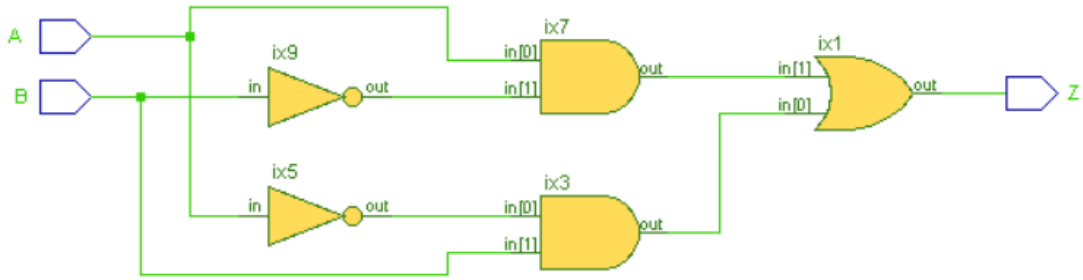


Figure 6 RTL Schematic for XOR gate

Table 2 XOR Truth Table

A	B	C
0	0	0
0	1	1
1	0	1

1	1	0

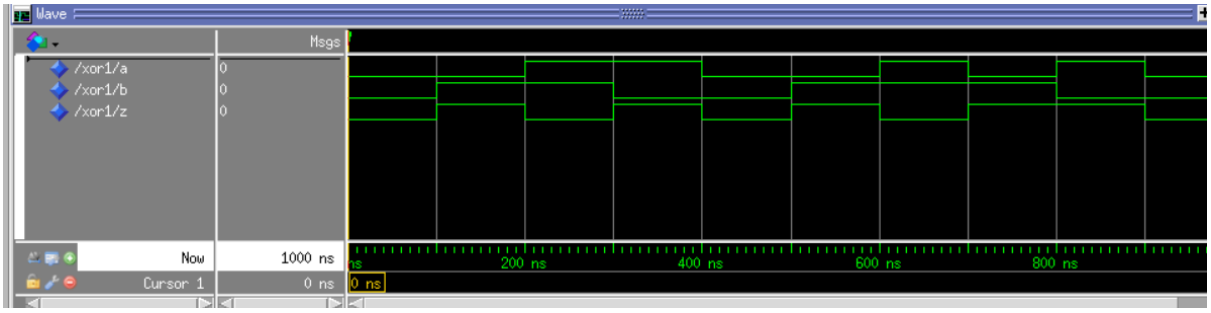


Figure 7 Simulation Result of XOR gate

3.1.4 NOT

In digital logic, NOT gate is a logic gate which implements logical negation. In mathematical logic it is equivalent to the logical negation operator (¬). It is also called an inverter. The truth table is shown for this gate below. The NOT gate is tested with a test bench based on the truth table and the simulation is show below.

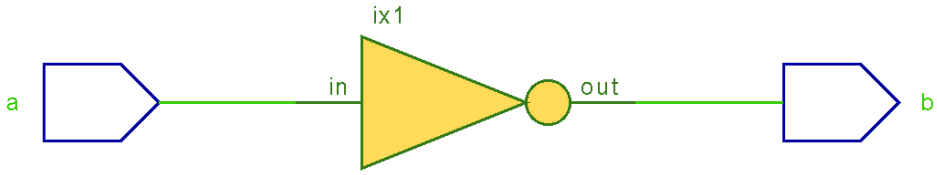


Figure 8 RTL Schematic for NOT gate

Table 3 Not Table

A	B
0	1
1	0

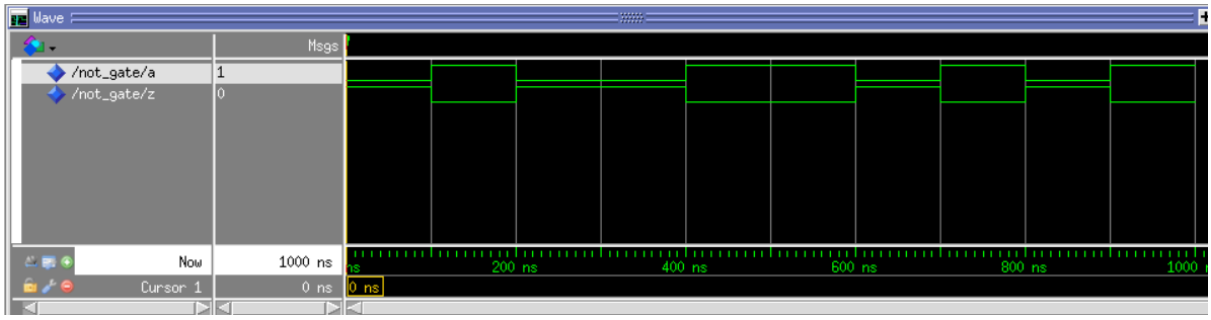


Figure 9 Simulation of NOT gate

4. Design of Basic Functional Blocks

The functional blocks perform the basic functions using the basic gates. In this section we will discuss all the basic blocks used in our project including their internal circuit, truth table and the simulation result. These functional blocks are designed in order to perform the functions like adding bits, storing bits, etc.

4.1 HALF_ADDER

The half adder is used in performing the logical addition for two inputs and this inputs generate two outputs which are sum and carry. SUM is the output in logical XOR operation of inputs A and B. The output carry is logical AND of the inputs A and B.

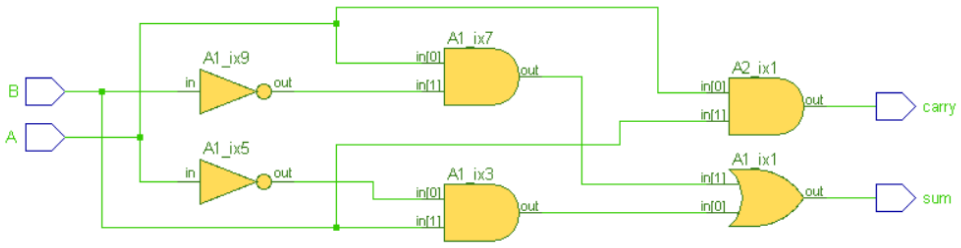


Figure 10 RTL Schematic for Half Adder

The truth table of half adder is as follows:

Table 4 Half Adder Truth Table

A	B	SUM	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

SUM= A ⊕ B

CARRY= A . B

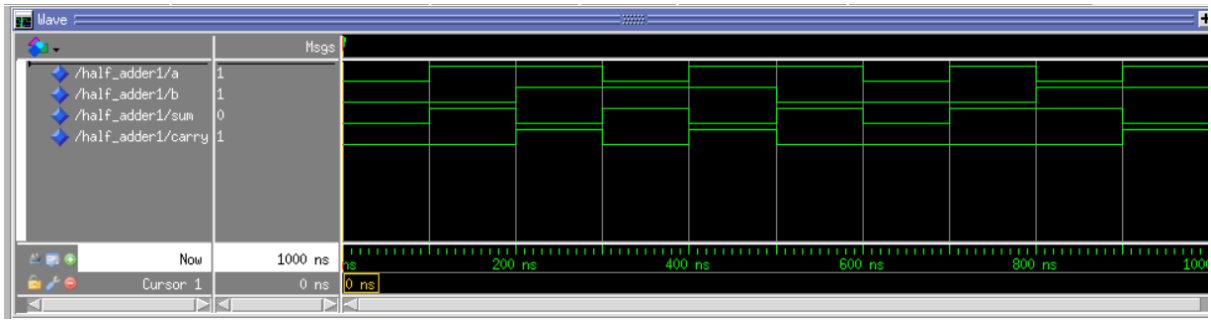


Figure 11 Simulation Result of Half Adder

4.2 FULL ADDER

The full adder performs the logical addition of two inputs along with a carry in and two outputs are generated - sum and carry. In this project, Full adder is constructed from two half adders by connecting the A and B to the input of one half adder and then connecting the sum of the two to an input of the second adder as shown in the RTL schematic. The sum and carry can be calculated from

$S1 = D \oplus E \oplus F$

$C1 = (D \cdot E) + (F \cdot (D \oplus E))$

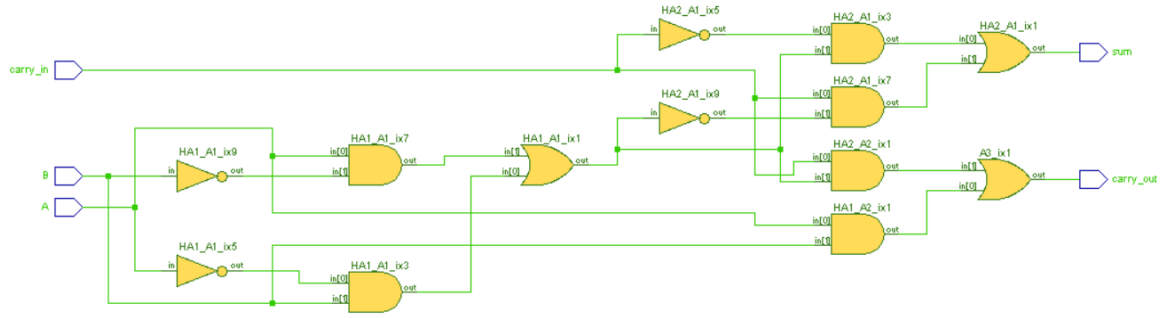


Figure 102 RTL schematic for Full Adder

The truth table for full adder is as follows:

Table 5 Full Adder Truth Table

A	B	carry_in	sum	carry_out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0

0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

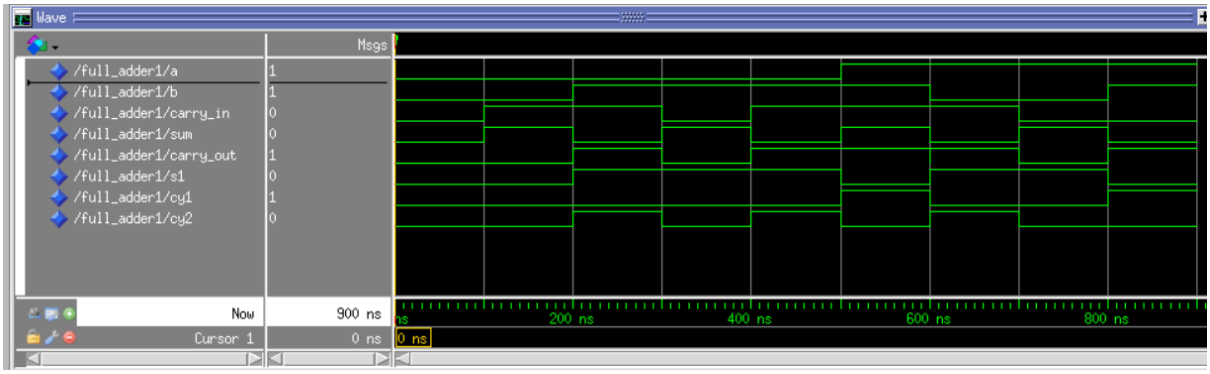
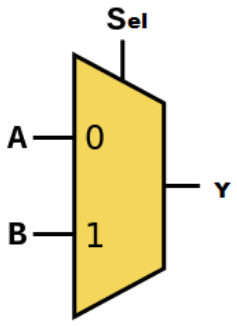


Figure 13 Simulation result of Full Adder

4.3 2:1 MULTIPLEXER

The term "multiplexer" is also known as "Mux," and it refers to a device that selects only one output from a set of N inputs. Multiplexing is a technique for sending a large number of data units across a short range. We utilized the basic 2:1 mux in this project, which consists of a single select line that selects one of the two inputs and outputs it. Block 3 shows the fundamental block diagram for a 2:1 mux.



Block 1: 2:1 Mux

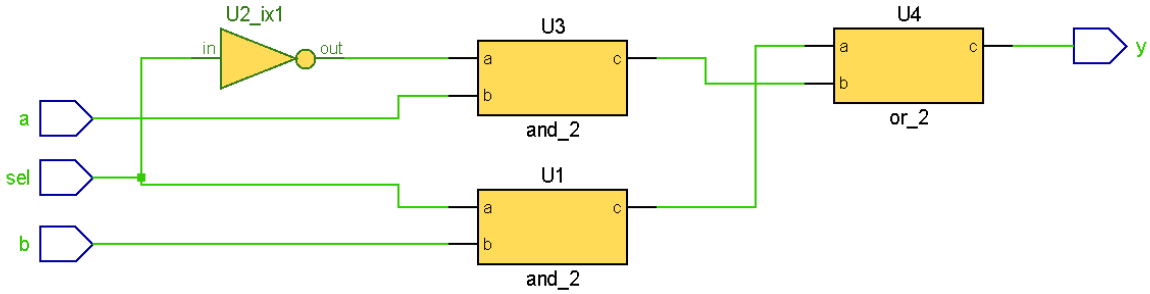


Figure 14 RTL Schematic for 2:1 Mux

The truth table for 2:1 mux with a select line is as follows:

Table 6 Truth Table for 2:1 Mux

A	B	Sel	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

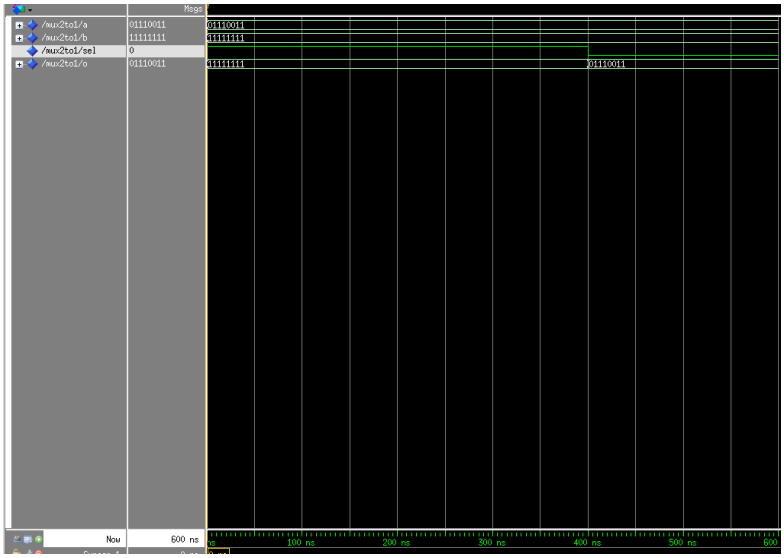
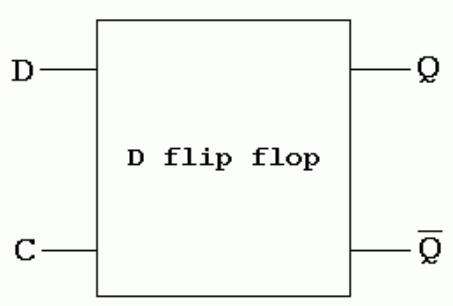


Figure 15 Simulation Result for 2:1 Mux

4.4 D FLOP-FLOP

D Flip Flops are used as a part of memory storage elements and also as a data processor. It can build either by NAND gate or by NOR gate. One of the important applications of D flip flop is to produce delay in timing circuit, as a buffer for sampling data at specific intervals. A and B are the two input numbers which have to be loaded into two 8-bit registers RA and RB. To store the number, am 8 bit D flipflop is used as the register. This RA and RB are connected to the array multiplier where both the numbers are multiplied. Registers of size n can be built from n number of D flip flops to store each bit in the word or block of data. The basic block diagram of D flip flop is as follows,



Block 2: D Flip Flop

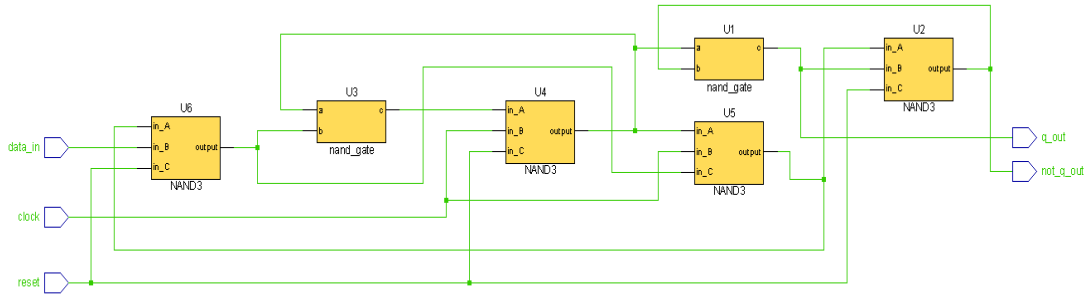


Figure 16 RTL schematic for D Flip Flop

Whenever the clock signal is low, the input is not going to affect the output state, so the clock must be high for the inputs to get active. The truth table for D flip flop in the presence of clock is as follows

Table 7 D Flip Flop Truth Table

D	C	Q	Q’
x	low	0	1
0	high	0	1
1	high	1	0

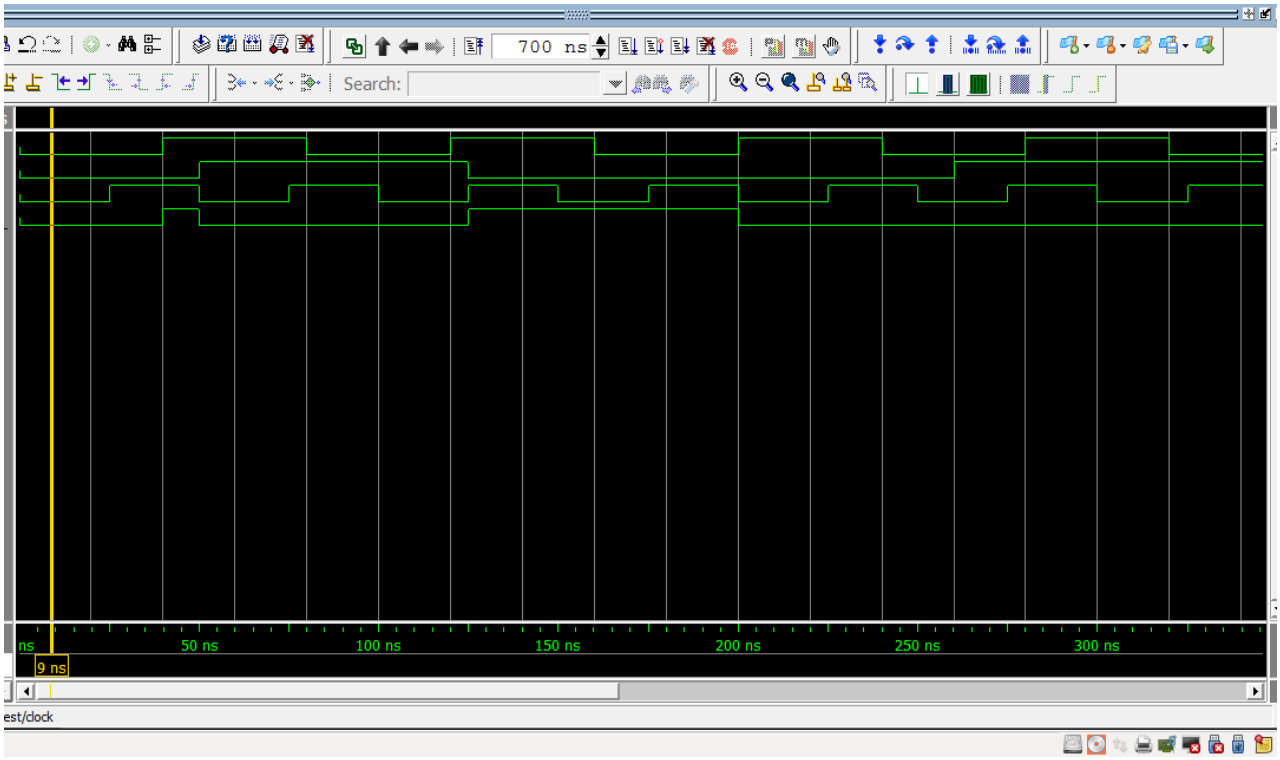


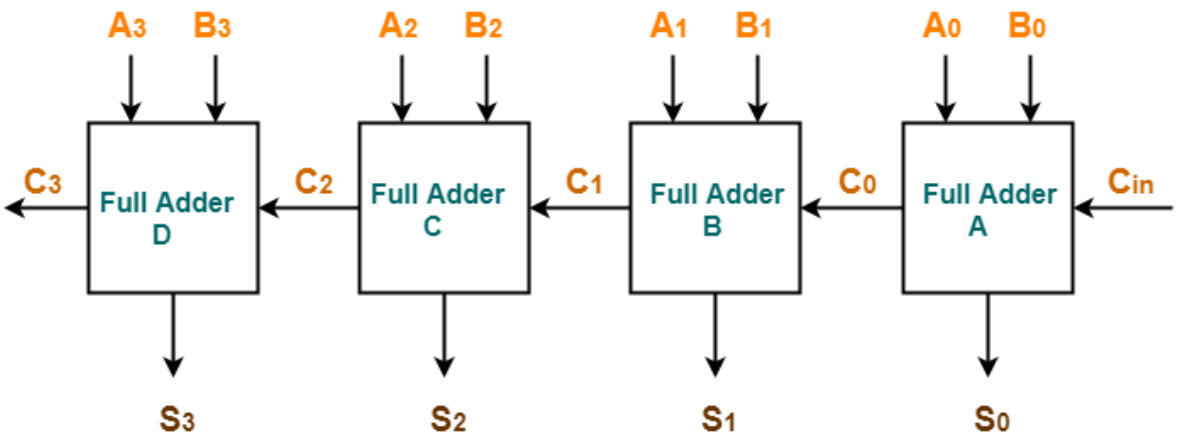
Figure 17 Simulation Result for D flip flop

5. Design of Higher-Level Functional Blocks

In this section we will discuss about the higher-level functional blocks such as Ripple Carry Adder, Carry Save Adder which is built by the basic functional block such as Full Adder. The carry-out of each full adder is the carry in of next full adder. And also, the devices such as 8-bit, 16-bit and 24-bit registers are used for storing bits are discussed here.

5.1 RIPPLE CARRY ADDER

Multiple full adder circuits can be cascaded in parallel to add an N-bit number to make an N-bit parallel adder. The ripple carry adder is a logic circuit in which one full adder's carry-out is a carry-in for the subsequent full adder. The ripple carry adder gets its name from the fact that a carry bit is rippled into the next stage. It's also evident that a half adder can only replace the first full adder. The ripple carry adder's layout is simple, allow for faster design time. However, because each full adder must wait for the carry bit to be calculated from the previous full adder, the ripple carry adder is relatively slow when compared to other adders. However, it provides efficient area and has less power dissipation. The gate delay may be simply computed by looking at the overall adder circuit. To allow binary vector strings of larger sizes, a number of full adders can be added to the ripple carry adder, or ripple carry adders of different sizes can be cascaded. In other words, the ripple carry adder's final output is valid only after the the joint propagation delays of all full adder circuits inside it. The 24-bit ripple carry adder's fundamental block diagram is given below.



Block 3: 4-bit Ripple Carry Adder

As per the project specification, we require 24-bit Ripple carry adder. Though the ripple carry adder gives the more delay but it acquires less area and consumes less power when compared to all the other adders. RTL schematic for 24-bit RCA is below,

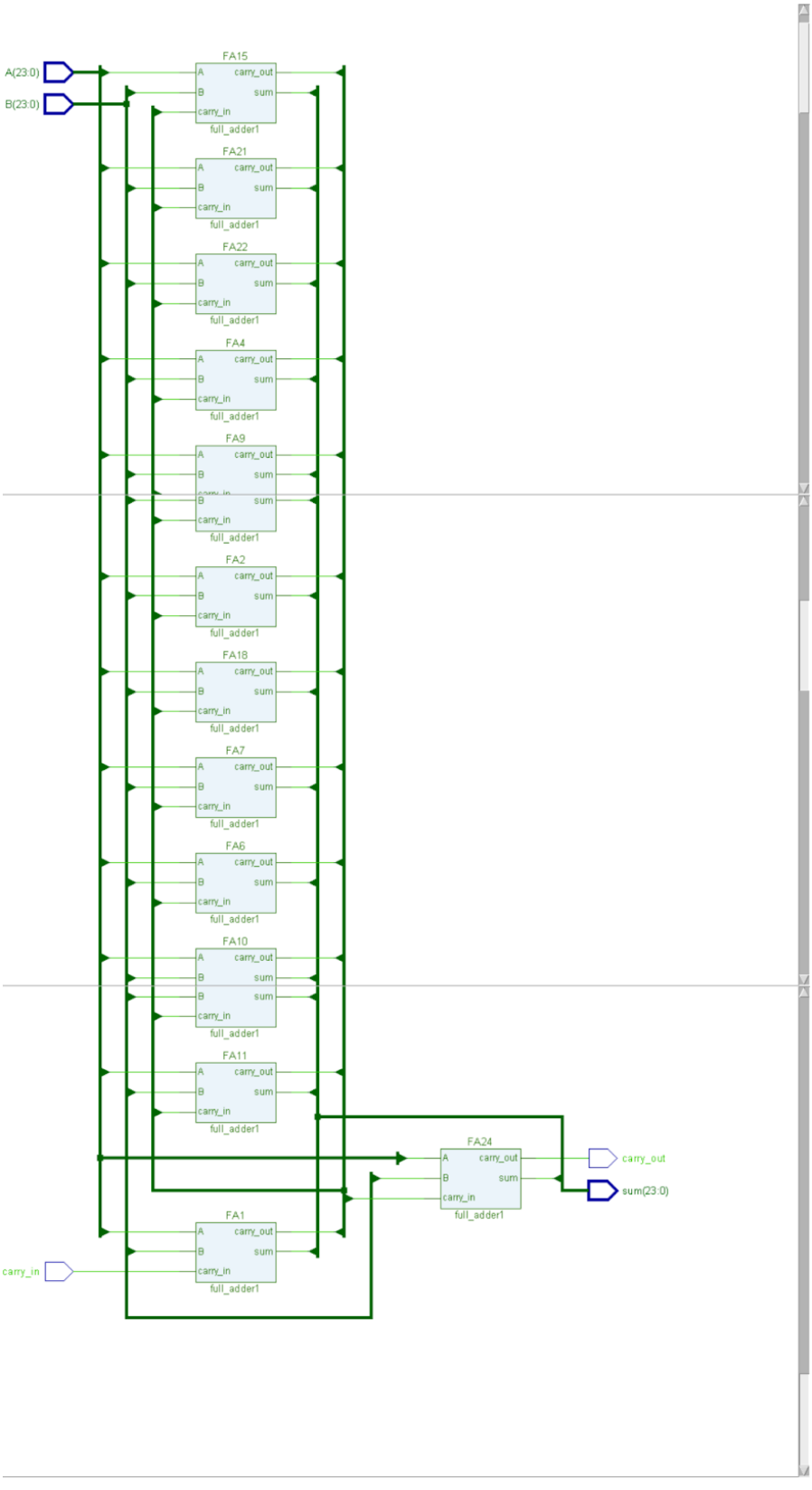


Figure 18 RTL schematic for 24-bit Ripple Carry Adder

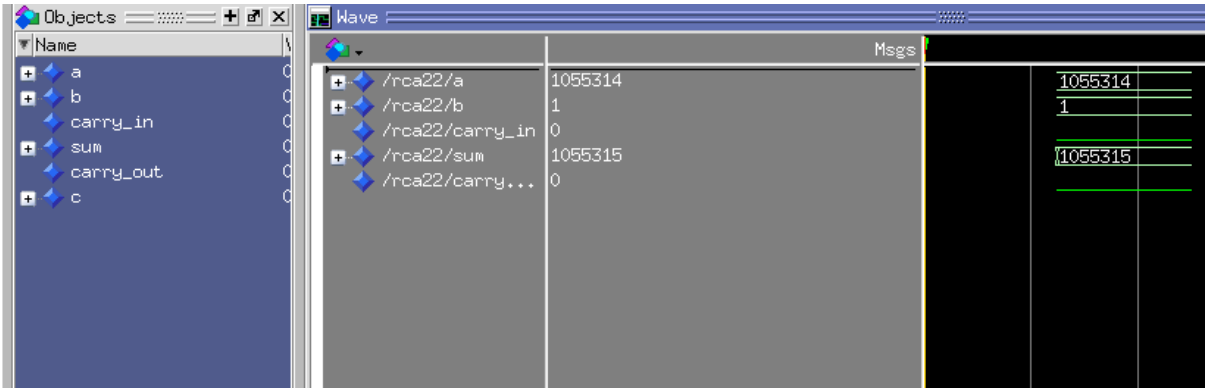


Figure 19 Simulation result for 24-bit Adder

6.Multiplier

Multipliers play an important role in today's digital signal processing and various other applications. With advances in technology, many researchers have tried and are trying to design multipliers which offer either of the following design targets – high speed, low power consumption, regularity of layout and hence less area or even combination of them in one multiplier thus making them suitable for various high speed, low power and compact VLSI implementation.

Multiplication Algorithm:

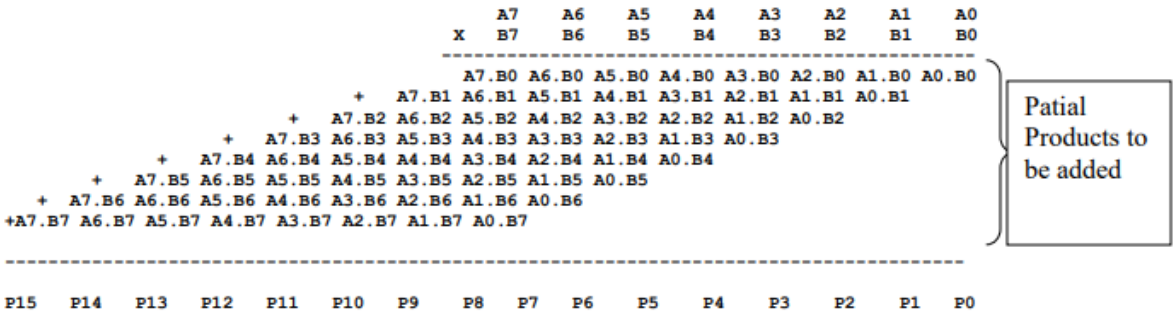
The multiplication algorithm for an N bit multiplicand by N bit multiplier is shown below:

Y= Yn-1 Yn-2 .....Y2 Y1 Y0 Multiplicand

X= Xn-1 Xn-2 ..... X2 X1 X0 Multiplier 2

AND gates are used to generate the Partial Products, PP, If the multiplicand is N-bits and the Multiplier is M-bits then there is N* M partial product. The way that the partial products are generated or summed up is the difference between the different architectures of various multipliers. Multiplication of binary numbers can be decomposed into additions.

Consider the multiplication of two 8-bit numbers A and B to generate the 16 bit product P.



The equation for the addition is: $$P(m+n) = A(m)B(n) = \sum_{i=0}^{m-1}\sum_{j=0}^{n-1} a_i b_j 2^{i+j} \quad .$$

Why we chose Wallace tree multiplier?

	Array Multiplier	Modified Booth Multiplier	Wallace Tree Multiplier	Modified Booth & Wallace Tree Multiplier
Area – Total CLB's (#)	1165	1292	1659	1239
Maximum Delay (ns)	187.87ns	139.41ns	101.14ns	101.43ns
Power Consumption at highest speed (mW)	16.6506mW (at 188ns)	23.136mW (at 140ns)	30.95mW (at 101.14ns)	30.862mW (at 101.43ns)
Delay •Power Product (DP) (ns mW)	3128.15	3225.39	3130.28	3130.33
Area • Power Product (AP) (# mW)	19.397 x 10^3	29.891 x 10^3	51.346 x 10^3	38.238 x 10^3
Area • Delay Product (AD) (# ns)	218.868 x 10^3	180.118 x 10^3	167.791 x 10^3	125.671 x 10^3
Area • Delay^2 Product(AD^2) (# ns^2)	41.119 x 10^6	25.110 x 10^6	16.970 x 10^6	**12.747 x 10^6**

From the above table it is clear that Wallace tree multiplier has minimum delay.

Wallace Tree Multiplier:

A Wallace tree is an efficient hardware implementation of a digital circuit that multiplies two integers by following three steps i.e.,

Multiply each bit of one of the arguments, by each bit of the other, yielding n^2 results. Depending on position of the multiplied bits, the wires carry different weights.

Reduce the number of partial products to two by layers of full and half adders.Group the wires in two numbers, and add them with a conventional adder.The benefit of the Wallace tree is that there are only O(logn) reduction layers, and each layer has O(1) propagation delay. As making the partial products is O(1) and the final addition is O(logn), the multiplication is only O(logn), not much slower than addition (however, much more expensive in the gate count). Naively adding partial products with regular adders would require $O(\log^2 n)$ time.

Multiplication of 16-bit number with 8-bit number,

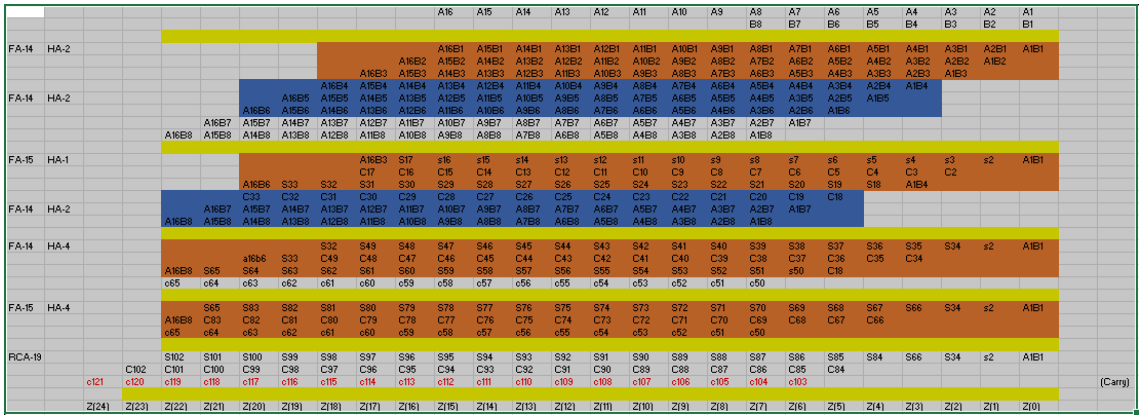
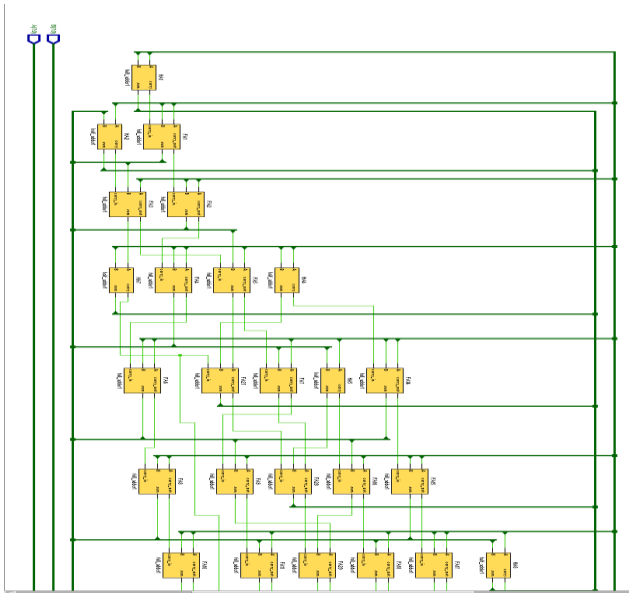


Figure 20 24-bit Wallace Tree Multiplier Reduction



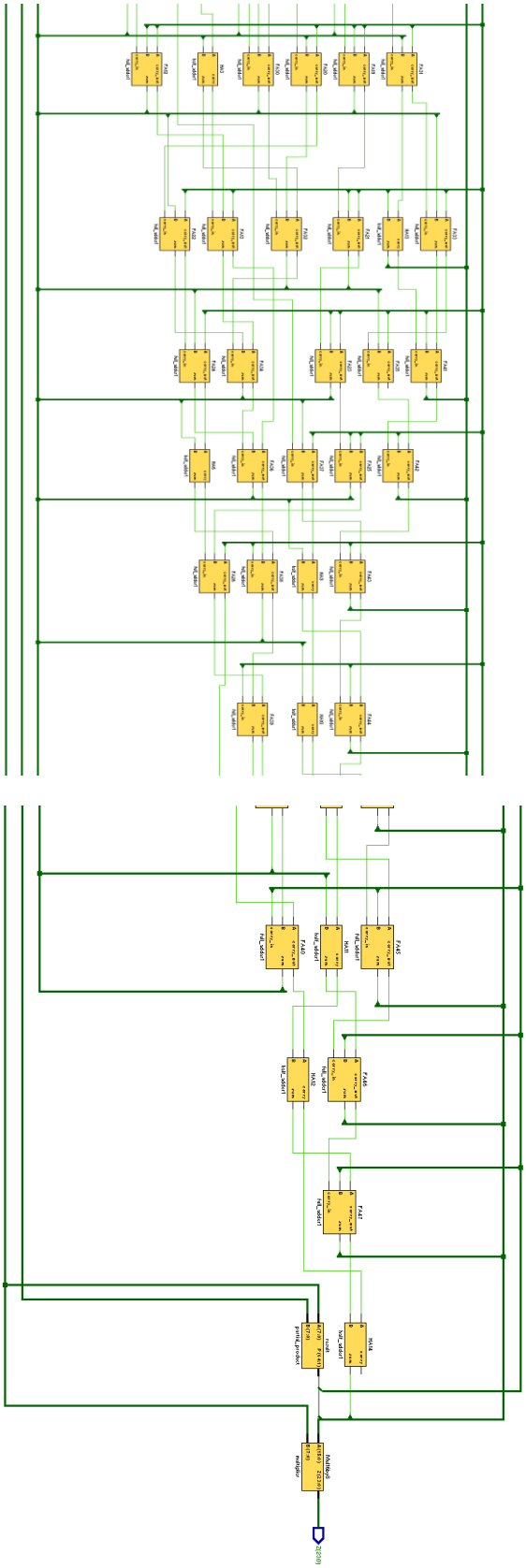


Figure 20 24-bit Wallace Tree Multiplier RTL schematic diagram

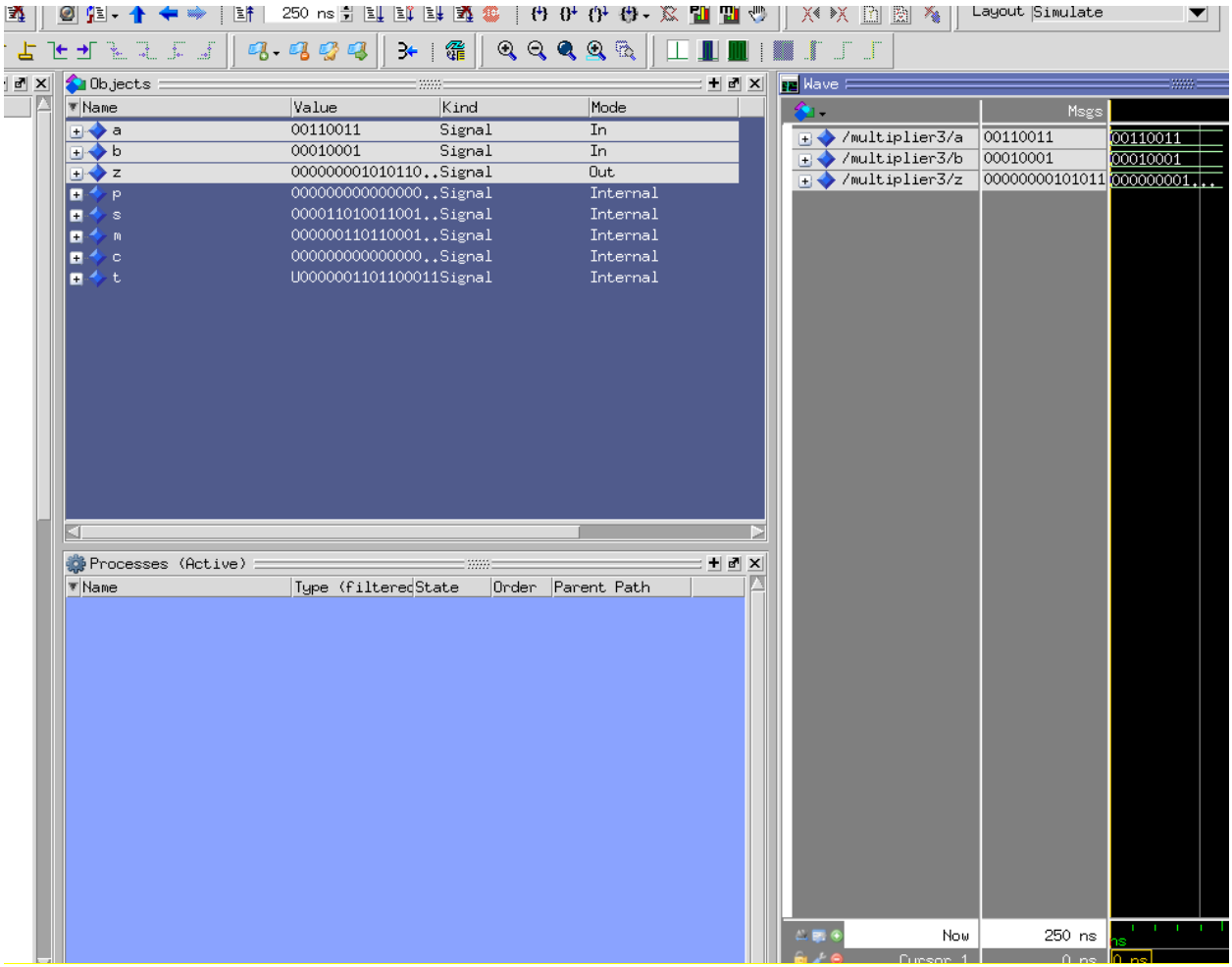


Figure 21 Simulation result for (A^2 *B) for Wallace Tree Multiplier

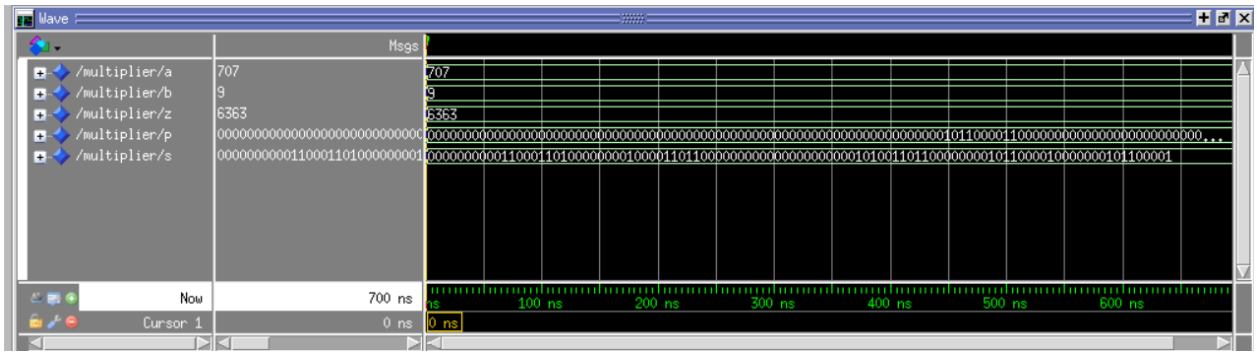


Figure 22 Simulation result for (A*B) for Wallace Tree Multiplier

7. IMPLEMENTATION OF ARITHMETIC UNIT

7.1 Use of Registers to load value:

During raising edge of the clock, when Clear is not enabled and Load is set to 0, Load the unsigned 8-bit number into the 8-bit register.

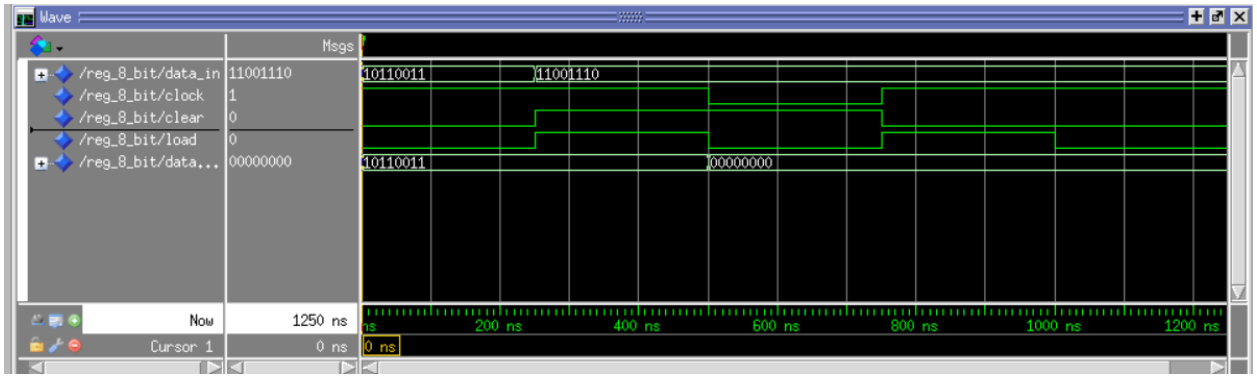


Figure 2311 .Simulation result of 8-bit register.

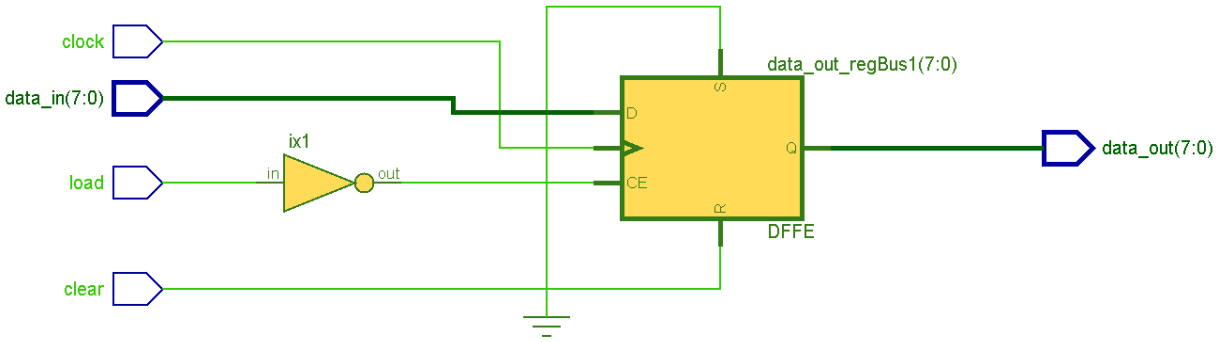


Figure 24 RTL Schematic for 8-bit register.

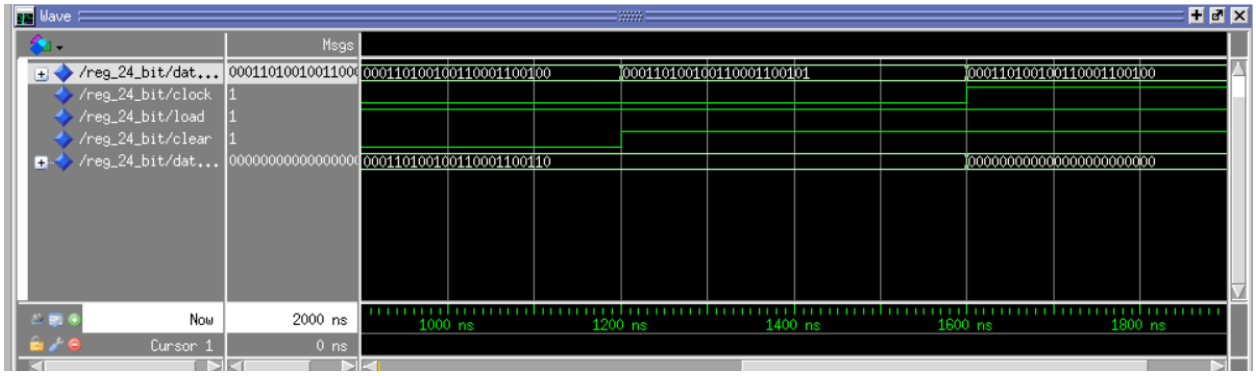


Figure 25 Simulation result of 24-bit register.

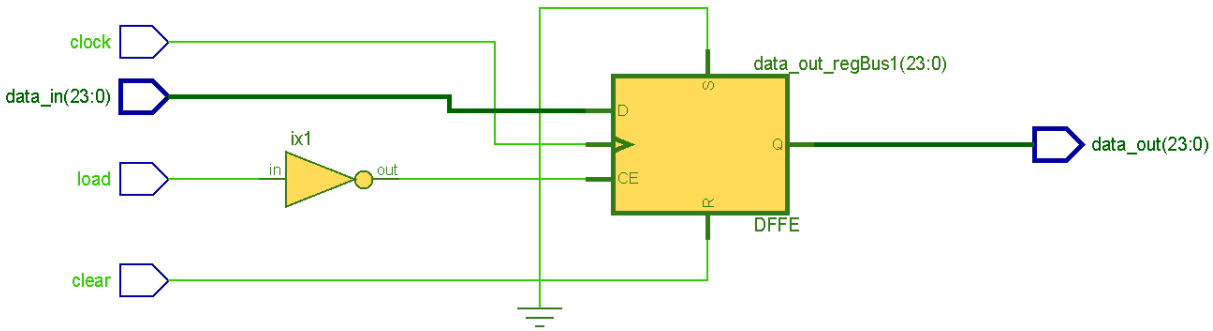


Figure 26 RTL Schematic for 24-bit register.

7.2 Multiplier Implementation:

After getting the 8-bit input, multiplication is done to get the output of the 8-bit input.

7.2.1 Algorithm:

The multiplication algorithm for a 16- bit multiplicand by 8-bit multiplier is shown in Figure 34:

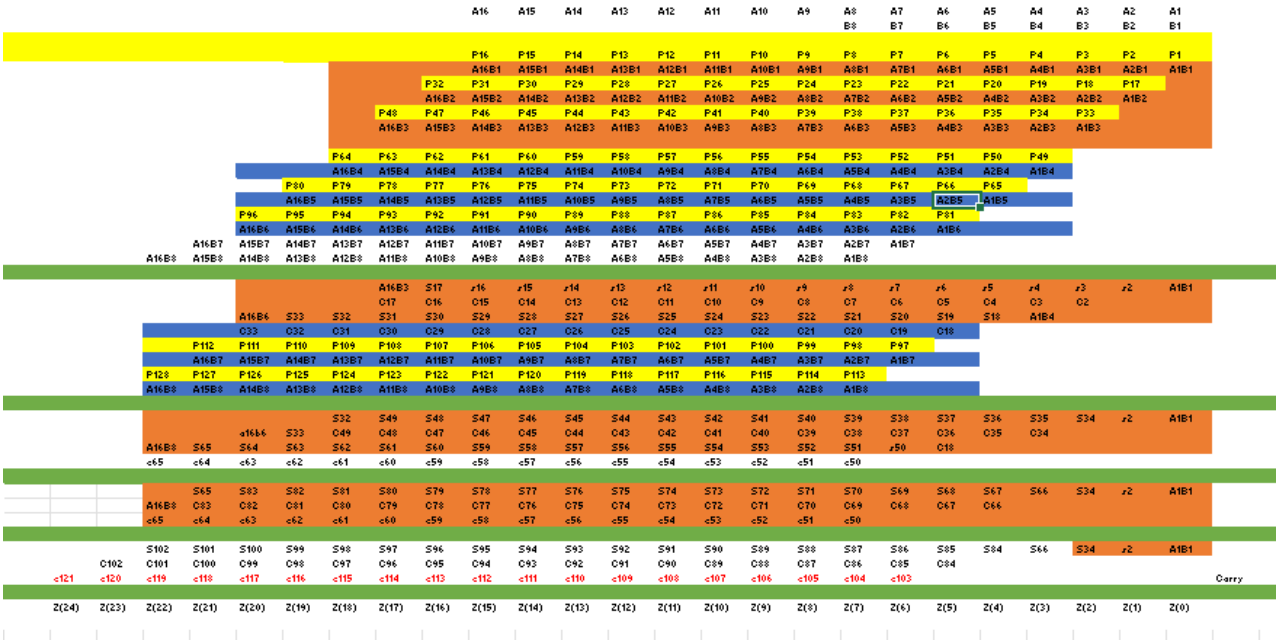


Figure 27 Algorithm used for multiplication

Since the arithmetic unit has performed a multiplier operation, the multiplier and multiplicand are same in this case. The input to the circuit is an 16-bit unsigned positive number.

Input: A= $a_{16}a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}a_9a_8a_7a_6a_5a_4a_3a_2a_1$

Input: B= $b_7b_6b_5b_4b_3b_2b_1$

Once the multiplier gets the 8-bit input, the next step is to determine the partial products. The ***number of partial products is given by m*n*** where m – the number of bits is multiplicand and n- the number of bits in multiplier. In this case the ***number of partial products is 128*** since the number of bits in multiplier and multiplicand is eight. The Partial Product is referred by the short form P. In this case P ranges from P1 to P64 where P1 = a_0a_0 , P2= a_0a_1 and so on.

Two input AND Gates are employed to determine the values of Partial Products. The operation of the AND Gate has been explained in the previous sections.

.

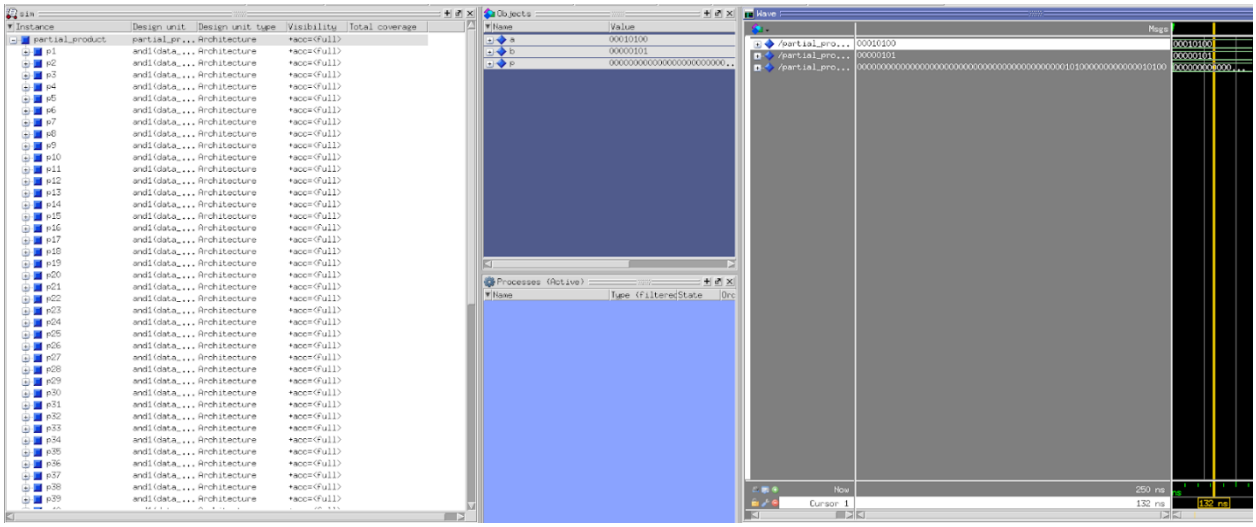


Figure 28 Simulation Result of (8X8) Partial Products Generation

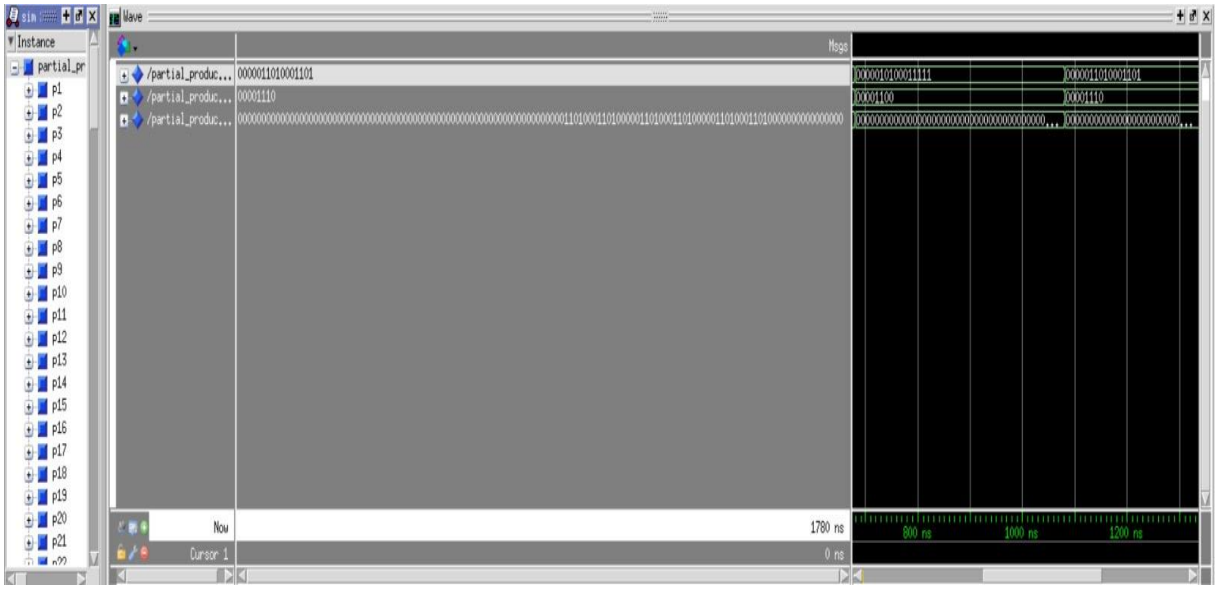


Figure 29 Simulation Result of (16X8) Partial Products Generation

The 16-bit output is then multiplied with 8-bit value. (i.e) 16X8 multiplication takes place.

7.3 Addition of Partial Products, Shifting, Adding and storing:

Once all the partial products have been determined we have to perform the addition on the partial products and the addition of the partial products is done using half and full adder finally stored in 24 bit register.

A shift register is a type of digital circuit using a cascade of flip-flops where the output of one flip-flop is connected to the input of the next. They share a single clock signal, which causes the data stored in the system to shift from one location to the next. After the multiplication to obtain (A^2*B) the 24-bit number has to be shifted right to left by two bits .The shifter shifts the bit by two bits from left to right. When the 24-bit number is shifted the last two bits from the left move into decimal places and the two zeros are filled in the place of the first two bits from the right. As the designed output register is 24 bit the two decimal bits are represented separately. So the output of this calculation is 24 bit number. The first two bits from the right will always be zeros.

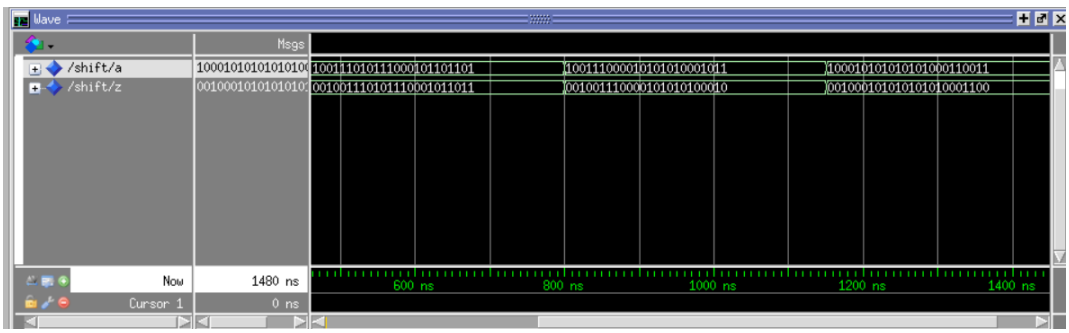


Figure 30 Simulation Result of Shift Registers.

7.4 Adding 1:

Final 24-bit output is then added by 1. This block is made up of RCA. The schematic diagram and simulation result is shown below,

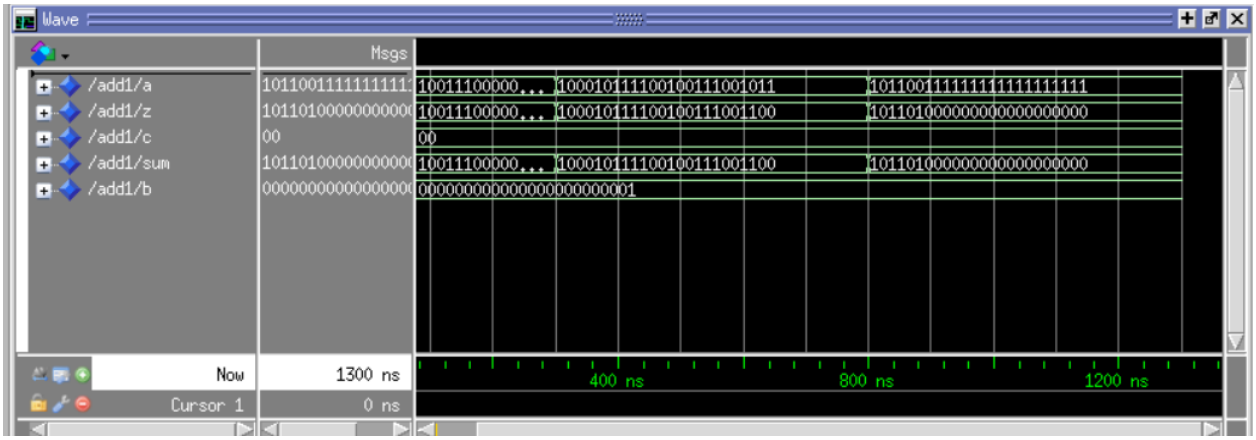


Figure 31 Simulation Result of Adding1

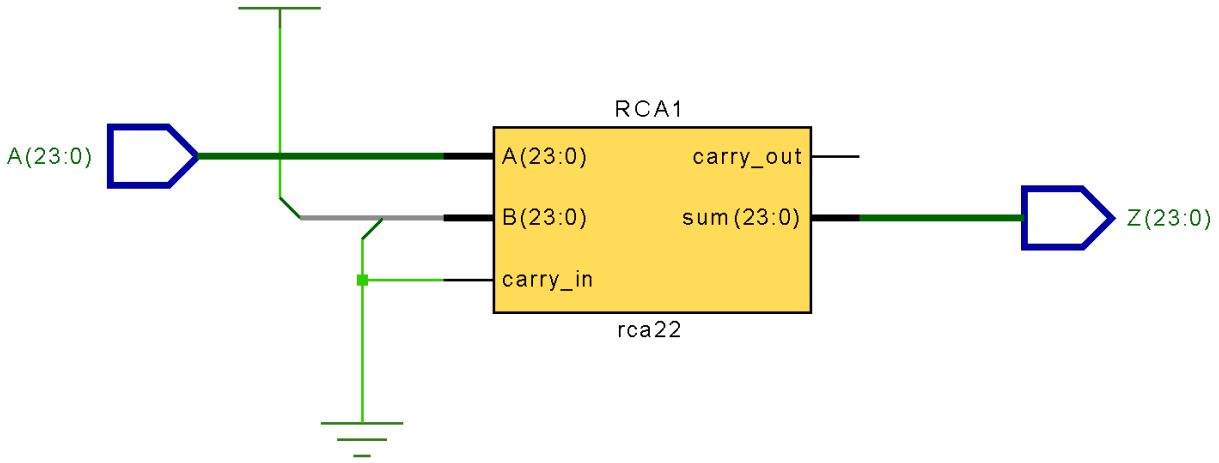


Figure 32 RTL Schematic diagram for Adding1

8. EXPERIMENT RESULT:

The final simulation results of the above proposed three different implementations are shown in the figure attached below,

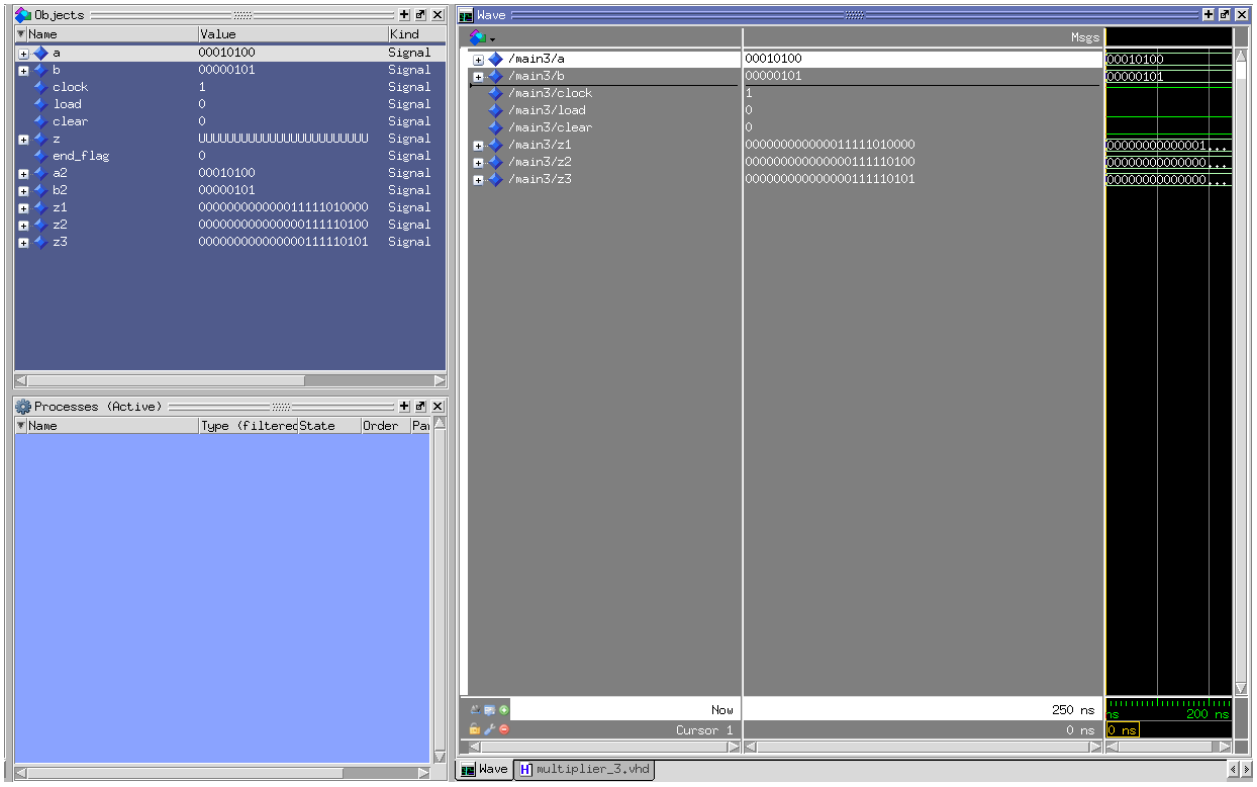


Figure 33 Simulation Result of Z = ¼ [A^2 * B] + 1 using Wallace tree multiplier.

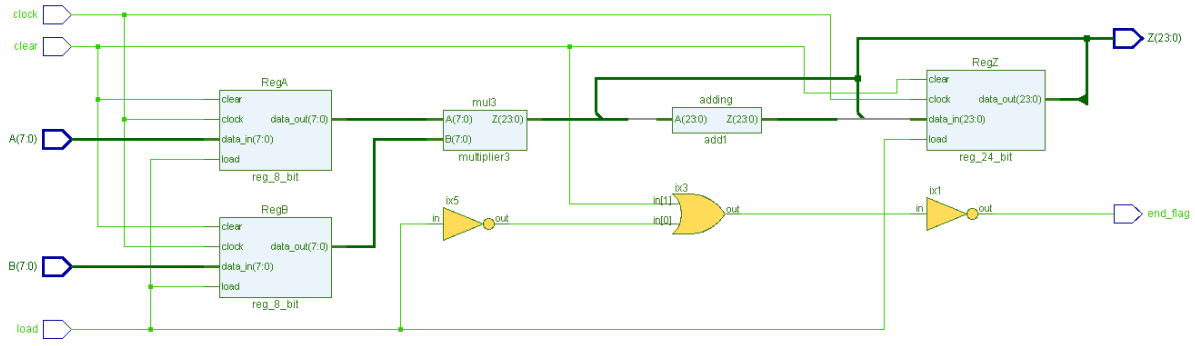


Figure 3412 Final RTL schematic using Wallace Tree Multiplier

9. CONCLUSION

The main aim of this project is to design an arithmetic unit that performs the operation **Z = ¼ [A^2 * B] + 1** where A,B is a unsigned 8-bit number. This report gives the detailed step by step information on how we implemented the arithmetic unit and the algorithm we followed to implement the design. This report also provides the test benches of different components to test the functionality of the design.

While performing the addition of partial products we have employed three different approaches. One by using normal multiplication, other by Array multiplier and another by Wallace tree Multiplier. We used tools like precision and Xilinx ISE to get the area and delay of the three different designs. The report on area and delay of the three different approaches are given in the Appendix III. On comparing the results we came to the conclusion with respect to area and delay, the use of Wallace tree multiplier is performing with great speed compared to array and normal multiplier. Finally we have used ripple carry adder for final addition in order to get the best output

Through this report we conclude that the arithmetic unit to perform **Z = ¼ [A^2 * B] + 1** was successfully implemented using three different approaches and both the approaches were successfully tested with the help of test bench.

Worksheet

'✓' REPRESENTS SIGNIFICANT CONTRIBUTION

M1: Method one, M2: Method two.

S.NO	VHDL CODES	Shivendra Arulalan	Abhishek Madhu	Abdul Aziz	Thenmozhie Rajan	Ragul Nivash
1	Basic Logic Gates		☐		☐ M1	☐ M1
2	Half Adder	M2 ☐		M2 ☐	M1 ☐	☐ M1
3	Full Adder	M2 ☐		M2 ☐	☐ M1	☐ M1
4	2_1 Mux	M2 ☐		M2 ☐		
5	D Flip-Flop	M2 ☐		M2 ☐		
6	Carry Select Adder	M2 ☐		M2 ☐		
7	Ripple Carry Adder				☐ M1	☐ M1
8	8-Bit Register		☐		☐ M1	☐ M1
9	16-Bit Register				M1 ☐	M1 ☐
10	24- Bit register				M1 ☐	M1 ☐
11	Wallace Multiplier	M2 ☐		M2 ☐	M1 ☐	M1 ☐

12	Positive Edge trigger	M2		M2		
12	¼ Shifter	M2				
13	Combinational Logic of Multiplier, Adder and Shifter	M2		M2	M1	M1
14	24 Bit adder	M2		M2	M1	M1
15	Timing Report					
S. No	REPORT CHAPTER	Shivendra Arulalan	Abhishek Madhu	Abdul Aziz	Thenmozhie Rajan	Ragul Nivash
1.	Abstract					
2.	Introduction					
3.	Basic Logic Gates					
4.	Basic Functional Blocks					
5.	Arithmetic Unit					
6.	Area Report					
7.	Timing Report					
8.	Project Description					

9.	Design and analysis of components					
10.	Design of Mux					
11.	Design of Registers					
12.	Simulation					
13.	Conclusion					

APPENDICES

Appendix I –

A.VHDL Codes:

1. Logic Gate

1.1AND

```
-- Declare library files:
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
-- Entity Declaration:
entity and1 is
port(A,B:in std_logic;Z:out std_logic);
end and1;
-- Architecture Implementation:
```

```
architecture data_flow of and1 is
begin
Z<= A and B;
end data_flow;
-- end of and_gate.vhd
```

1.2 OR

```
-- Declare library files:
library ieee;
use ieee.std_logic_1164.all;
-- Entity Declaration:
entity or1 is
port(A,B:in std_logic;Z:out std_logic);
end or1;
architecture data_flow of or1 is
begin
Z<= A or B;
end data_flow;
```

1.3 XOR

```
-- Declare library files:
library ieee;
use ieee.std_logic_1164.all;
-- Entity Declaration:
entity xor1 is
port(A,B:in std_logic;Z:out std_logic);
end xor1;
architecture data_flow of xor1 is
begin
Z<= ((not A) and B) or (A and (not B));
```

```
end data_flow;
```

1.4 NOT

```
library ieee;
use ieee.std_logic_1164.all;

entity not_gate is
port(A: in std_logic; Z: out std_logic);
end not_gate;

architecture data_flow of not_gate is
begin
Z<= not A;
end data_flow;
```

2. Basic Functional Blocks

2.1 HALF ADDER

```
library ieee;

use ieee.std_logic_1164.all;


entity half_adder1 is

port( A, B : in std_logic;

sum, carry : out std_logic);

end half_adder1;

architecture structural of half_adder1 is

component xor1

port(A,B:in std_logic;Z:out std_logic);

end component;

component and1

port(A,B:in std_logic;Z:out std_logic);

end component;

begin

A1:xor1 port map(A,B,sum);
```

```
A2:and1 port map(A,B,carry);
end structural;
```

2.2 FULL ADDER

```
library ieee;
use ieee.std_logic_1164.all;

entity full_adder1 is
port( A, B, carry_in : in std_logic;
                sum, carry_out : out std_logic);
end full_adder1;

architecture structural of full_adder1 is
component half_adder1
port( A, B : in std_logic;
                sum, carry : out std_logic);
end component;
component or1
port(A,B:in std_logic;Z:out std_logic);
end component;
signal s1,cy1,cy2:std_logic;
begin
HA1:half_adder1 port map(A,B,s1,cy1);
HA2:half_adder1 port map(carry_in,s1,sum,cy2);
A3:or1 port map(cy1,cy2,carry_out);
end structural;
```

3. Higher Level Functional Blocks

3.1 24-BIT RIPPLE CARRY ADDER

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

-- Entity Declaration:
entity rca22 is
        port( A, B  : in std_logic_vector (23 downto 0);
                carry_in : in std_logic;
                sum     : out std_logic_vector (23 downto 0);
                carry_out : out std_logic);
end rca22;

-- Architecture Implementation:
architecture dataflow of rca22 is

-- Component Declaration of full_adder:
component full_adder1
        port( A, B, carry_in : in std_logic;
                sum, carry_out : out std_logic);
end component;

-- Signal declaration of internal variables:
signal C: std_logic_vector (24 downto 0);

begin
```

```
-- Assign the value of Carry in to C(0):
C(0) <= carry_in;

FA1: full_adder1 port map (A(0), B(0), C(0), sum(0), C(1));
FA2: full_adder1 port map (A(1), B(1), C(1), sum(1), C(2));
FA3: full_adder1 port map (A(2), B(2), C(2), sum(2), C(3));
FA4: full_adder1 port map (A(3), B(3), C(3), sum(3), C(4));
FA5: full_adder1 port map (A(4), B(4), C(4), sum(4), C(5));
FA6: full_adder1 port map (A(5), B(5), C(5), sum(5), C(6));
FA7: full_adder1 port map (A(6), B(6), C(6), sum(6), C(7));
FA8: full_adder1 port map (A(7), B(7), C(7), sum(7), C(8));
FA9: full_adder1 port map (A(8), B(8), C(8), sum(8), C(9));
FA10: full_adder1 port map (A(9), B(9), C(9), sum(9), C(10));
FA11: full_adder1 port map (A(10), B(10), C(10), sum(10), C(11));
FA12: full_adder1 port map (A(11), B(11), C(11), sum(11), C(12));
FA13: full_adder1 port map (A(12), B(12), C(12), sum(12), C(13));
FA14: full_adder1 port map (A(13), B(13), C(13), sum(13), C(14));
FA15: full_adder1 port map (A(14), B(14), C(14), sum(14), C(15));
FA16: full_adder1 port map (A(15), B(15), C(15), sum(15), C(16));

FA17: full_adder1 port map (A(16), B(16), C(16), sum(16), C(17));
FA18: full_adder1 port map (A(17), B(17), C(17), sum(17), C(18));
FA19: full_adder1 port map (A(18), B(18), C(18), sum(18), C(19));
FA20: full_adder1 port map (A(19), B(19), C(19), sum(19), C(20));
FA21: full_adder1 port map (A(20), B(20), C(20), sum(20), C(21));
FA22: full_adder1 port map (A(21), B(21), C(21), sum(21), C(22));
FA23: full_adder1 port map (A(22), B(22), C(22), sum(22), C(23));
FA24: full_adder1 port map (A(23), B(23), C(23), sum(23), C(24));
```

```
-- Assign the value of C(24) to Carry out:
carry_out <= C(24);


end dataflow;
```

3.2 8-BIT REGISTER

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


entity reg_8_bit is
Port ( data_in : in STD_LOGIC_VECTOR (7 downto 0);
       clock,clear,load : in STD_LOGIC;
       data_out : out STD_LOGIC_VECTOR (7 downto 0));
end reg_8_bit;



architecture Behavioral of reg_8_bit is
begin
  process (clock)
  begin
   if clear = '1' then
        data_out <="00000000";
    -- check for positive edge of clock
   elsif clock'event and clock = '1' then


      -- if the load signal is high then load the value of input to registers:
      if load ='0' then
        data_out <=data_in;
      end if;
```

```
    end if;
  end process;
end;
```

3.3 24-BIT REGISTER

```
library IEEE;
use ieee.std_logic_1164.all;
entity reg_24_bit is
        port( data_in : in std_logic_vector (23 downto 0);
                clock, load, clear: in std_logic;
                data_out : out std_logic_vector (23 downto 0));
end reg_24_bit;
-- Architecture implementation:
architecture regC of reg_24_bit is
begin
  process (clock)
  begin
    if clear = '1' then
       data_out <="000000000000000000000000";
    -- check for positive edge of clock
    elsif clock'event and clock = '1' then
      -- if the clear signal is high then reset the register:

      -- if the load signal is high then load the value of input to registers:
      if load ='0' then
        data_out <=data_in;
      end if;
    end if;
  end process;
```

```
end regC;
```

4. Multiplier:

4.1 (16*8) Wallace Tree Multiplier

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

-- Entity Declaration:
entity multiplier is
        port( A : in std_logic_vector (15 downto 0);
                B: in std_logic_vector(7 downto 0);
                 Z: out std_logic_vector (23 downto 0));
end multiplier;

-- Architecture Implementation:
architecture mul of multiplier is

-- Component Declaration of partial product:
Component partial_product24
        port( A  : in std_logic_vector (15 downto 0);
                B:in std_logic_vector(7 downto 0);
                  P: out std_logic_vector (128 downto 1));
end Component;

-- Component Declaration of half adder:
Component half_adder1
        port( A, B : in std_logic;
```

```
                sum, carry : out std_logic);
end Component;

-- Component Declaration of full_adder:
Component full_adder1
        port( A, B, carry_in : in std_logic;
                sum, carry_out : out std_logic);
end Component;

-- Signal declarations:
signal P: std_logic_vector (128 downto 1);
signal S: std_logic_vector (102 downto 2);
signal C: std_logic_vector (121 downto 2);

begin

-- Finding the partial products of the 8 bit number:
result: partial_product24 port map (A (15 downto 0),B (7 downto 0), P (128 downto 1));

-- Performing addition on partial products using half adders and full adders:

-- First STAGE:
Z(0)<=P(1);
HA1: half_adder1 port map (P(2), P(17), S(2), C(2));
FA1: full_adder1 port map (P(3), P(18),P(33), S(3), C(3));
FA2: full_adder1 port map (P(4), P(19),P(34), S(4), C(4));
FA3: full_adder1 port map (P(5), P(20),P(35), S(5), C(5));
FA4: full_adder1 port map (P(6), P(21),P(36), S(6), C(6));
```

FA5: full_adder1 port map (P(7), P(22),P(37), S(7), C(7));

FA6: full_adder1 port map (P(8), P(23),P(38), S(8), C(8));

FA7: full_adder1 port map (P(9), P(24),P(39), S(9), C(9));

FA8: full_adder1 port map (P(10), P(25),P(40), S(10), C(10));

FA9: full_adder1 port map (P(11), P(26),P(41), S(11), C(11));

FA10: full_adder1 port map (P(12), P(27),P(42), S(12), C(12));

FA11: full_adder1 port map (P(13), P(28),P(43), S(13), C(13));

FA12: full_adder1 port map (P(14), P(29),P(44), S(14), C(14));

FA13: full_adder1 port map (P(15), P(30),P(45), S(15), C(15));

FA14: full_adder1 port map (P(16), P(31),P(46), S(16), C(16));

HA2: half_adder1 port map (P(32), P(47), S(17), C(17));

HA3: half_adder1 port map (P(50), P(65), S(18), C(18));

FA15: full_adder1 port map (P(51), P(66),P(81), S(19), C(19));

FA16: full_adder1 port map (P(52), P(67),P(82), S(20), C(20));

FA17: full_adder1 port map (P(53), P(68),P(83), S(21), C(21));

FA18: full_adder1 port map (P(54), P(69),P(84), S(22), C(22));

FA19: full_adder1 port map (P(55), P(70),P(85), S(23), C(23));

FA20: full_adder1 port map (P(56), P(71),P(86), S(24), C(24));

FA21: full_adder1 port map (P(57), P(72),P(87), S(25), C(25));

FA22: full_adder1 port map (P(58), P(73),P(88), S(26), C(26));

FA23: full_adder1 port map (P(59), P(74),P(89), S(27), C(27));

FA24: full_adder1 port map (P(60), P(75),P(90), S(28), C(28));

FA25: full_adder1 port map (P(61), P(76),P(91), S(29), C(29));

FA26: full_adder1 port map (P(62), P(77),P(92), S(30), C(30));

FA27: full_adder1 port map (P(63), P(78),P(93), S(31), C(31));

FA28: full_adder1 port map (P(64), P(79),P(94), S(32), C(32));

HA4: half_adder1 port map (P(80), P(95), S(33), C(3));

--SECOND STAGE:

Z(1)<=S(2);

HA5: half_adder1 port map (S(3), C(2), S(34), C(34));

FA29: full_adder1 port map (S(4), C(3),P(49), S(35), C(35));

FA30: full_adder1 port map (S(5), C(4),S(18), S(36), C(36));

FA31: full_adder1 port map (S(6), C(5),S(19), S(37), C(37));

FA32: full_adder1 port map (S(7), C(6),S(20), S(38), C(38));

FA33: full_adder1 port map (S(8), C(7),S(21), S(39), C(39));

FA34: full_adder1 port map (S(9), C(8),S(22), S(40), C(40));

FA35: full_adder1 port map (S(10), C(9),S(23), S(41), C(41));

FA36: full_adder1 port map (S(11), C(10),S(24), S(42), C(42));

FA37: full_adder1 port map (S(12), C(11),S(25), S(43), C(43));

FA38: full_adder1 port map (S(13), C(12),S(26), S(44), C(44));

FA39: full_adder1 port map (S(14), C(13),S(27), S(45), C(45));

FA40: full_adder1 port map (S(15), C(14),S(28), S(46), C(46));

FA41: full_adder1 port map (S(16), C(15),S(29), S(47), C(47));

FA42: full_adder1 port map (S(17), C(16),S(30), S(48), C(48));

FA43: full_adder1 port map (P(48), C(17),S(31), S(49), C(49));

HA6: half_adder1 port map (C(19), P(97), S(50), C(50));

FA44: full_adder1 port map (C(20), P(98),P(113), S(51), C(51));

FA45: full_adder1 port map (C(21), P(99),P(114), S(52), C(52));

FA46: full_adder1 port map (C(22), P(100),P(115), S(53), C(53));

FA47: full_adder1 port map (C(23), P(101),P(116), S(54), C(54));

FA48: full_adder1 port map (C(24), P(102),P(117), S(55), C(55));

FA49: full_adder1 port map (C(25), P(103),P(118), S(56), C(56));

FA50: full_adder1 port map (C(26), P(104),P(119), S(57), C(57));

FA51: full_adder1 port map (C(27), P(105),P(120), S(58), C(58));

FA52: full_adder1 port map (C(28), P(106),P(121), S(59), C(59));

FA53: full_adder1 port map (C(29), P(107),P(122), S(60), C(60));

FA54: full_adder1 port map (C(30), P(108),P(123), S(61), C(61));

FA55: full_adder1 port map (C(31), P(109),P(124), S(62), C(62));

FA56: full_adder1 port map (C(32), P(110),P(125), S(63), C(63));

FA57: full_adder1 port map (P(96), P(111),P(126), S(64), C(64));

HA7: half_adder1 port map (P(112), P(127), S(65), C(65));

--THIRD STAGE:

Z(2)<=S(34);

HA8: half_adder1 port map (S(35), C(34), S(66), C(66));

HA9: half_adder1 port map (S(36), C(35), S(67), C(67));

FA58: full_adder1 port map (S(37), C(36),C(18), S(68), C(68));

FA59: full_adder1 port map (S(38), C(37),S(50), S(69), C(69));

FA60: full_adder1 port map (S(39), C(38),S(51), S(70), C(70));

FA61: full_adder1 port map (S(40), C(39),S(52), S(71), C(71));

FA62: full_adder1 port map (S(41), C(40),S(53), S(72), C(72));

FA63: full_adder1 port map (S(42), C(41),S(54), S(73), C(73));

FA64: full_adder1 port map (S(43), C(42),S(55), S(74), C(74));

FA65: full_adder1 port map (S(44), C(43),S(56), S(75), C(75));

FA66: full_adder1 port map (S(45), C(44),S(57), S(76), C(76));

FA67: full_adder1 port map (S(46), C(45),S(58), S(77), C(77));

FA68: full_adder1 port map (S(47), C(46),S(59), S(78), C(78));

```
FA69: full_adder1 port map (S(48), C(47),S(60), S(79), C(79));
FA70: full_adder1 port map (S(49), C(48),S(61), S(80), C(80));
FA71: full_adder1 port map (S(32), C(49),S(62), S(81), C(81));
HA10: half_adder1 port map (S(33), S(63), S(82), C(82));
HA11: half_adder1 port map (P(96), S(64), S(83), C(83));

--FOURTH STAGE:
Z(3)<=S(66);
HA12: half_adder1 port map (S(67), C(66), S(84), C(84));
HA13: half_adder1 port map (S(68), C(67), S(85), C(85));
HA14: half_adder1 port map (S(69), C(68), S(86), C(86));
FA72: full_adder1 port map (S(70), C(69),C(50), S(87), C(87));
FA74: full_adder1 port map (S(71), C(70),C(51), S(88), C(88));
FA75: full_adder1 port map (S(72), C(71),C(52), S(89), C(89));
FA76: full_adder1 port map (S(73), C(72),C(53), S(90), C(90));
FA77: full_adder1 port map (S(74), C(73),C(54), S(91), C(91));
FA78: full_adder1 port map (S(75), C(74),C(55), S(92), C(92));
FA79: full_adder1 port map (S(76), C(75),C(56), S(93), C(93));
FA80: full_adder1 port map (S(77), C(76),C(57), S(94), C(94));
FA81: full_adder1 port map (S(78), C(77),C(58), S(95), C(95));
FA82: full_adder1 port map (S(79), C(78),C(59), S(96), C(96));
FA83: full_adder1 port map (S(80), C(79),C(60), S(97), C(97));
FA84: full_adder1 port map (S(81), C(80),C(61), S(98), C(98));
FA85: full_adder1 port map (S(82), C(81),C(62), S(99), C(99));
FA86: full_adder1 port map (S(83), C(82),C(63), S(100), C(100));
FA87: full_adder1 port map (S(65), C(83),C(64), S(101), C(101));
HA15: half_adder1 port map (P(128), C(65), S(102), C(102));
```

```
--FIFTH STAGE:
Z(4)<=S(84);
HA16: half_adder1 port map (S(85), C(84), Z(5), C(103));
FA88: full_adder1 port map (S(86), C(85),C(103), Z(6), C(104));
FA89: full_adder1 port map (S(87), C(86),C(104), Z(7), C(105));
FA90: full_adder1 port map (S(88), C(87),C(105), Z(8), C(106));
FA91: full_adder1 port map (S(89), C(88),C(106), Z(9), C(107));
FA92: full_adder1 port map (S(90), C(89),C(107), Z(10), C(108));
FA93: full_adder1 port map (S(91), C(90),C(108), Z(11), C(109));
FA94: full_adder1 port map (S(92), C(91),C(109), Z(12), C(110));
FA95: full_adder1 port map (S(93), C(92),C(110), Z(13), C(111));
FA96: full_adder1 port map (S(94), C(93),C(111), Z(14), C(112));
FA97: full_adder1 port map (S(95), C(94),C(112), Z(15), C(113));
FA98: full_adder1 port map (S(96), C(95),C(113), Z(16), C(114));
FA99: full_adder1 port map (S(97), C(96),C(114), Z(17), C(115));
FA100: full_adder1 port map (S(98), C(97),C(115), Z(18), C(116));
FA101: full_adder1 port map (S(99), C(98),C(116), Z(19), C(117));
FA102: full_adder1 port map (S(100), C(99),C(117), Z(20), C(118));
FA103: full_adder1 port map (S(101), C(100),C(118), Z(21), C(119));
FA104: full_adder1 port map (S(102), C(101),C(119), Z(22), C(120));
HA17: half_adder1 port map (C(102), C(120), Z(23), C(121));
end mul;
```

(8*8) Multiplier with 24-bit output:

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

-- Entity Declaration:
entity multiplier3 is
```

```
        port( A,B : in std_logic_vector (7 downto 0);
                Z: out std_logic_vector (23 downto 0));
end multiplier3;

-- Architecture Implementation:
architecture mul3 of multiplier3 is

-- Component Declaration of partial product:
Component partial_product
        port( A,B  : in std_logic_vector (7 downto 0);
                P: out std_logic_vector (64 downto 1));
end Component;

-- component declaration for 16 by 8 bit multiplication
Component multiplier
        port( A : in std_logic_vector (15 downto 0);
                B: in std_logic_vector(7 downto 0);
                Z: out std_logic_vector (23 downto 0));
end Component;

-- Component Declaration of half adder:
Component half_adder1
        port( A, B : in std_logic;
                sum, carry : out std_logic);
end Component;

-- Component Declaration of full_adder:
Component full_adder1
```

```
        port( A, B, carry_in : in std_logic;
                    sum, carry_out : out std_logic);
end Component;

-- Signal declarations:
signal P: std_logic_vector (64 downto 1);
signal S: std_logic_vector (21 downto 0);
signal M: std_logic_vector (37 downto 0);
signal C: std_logic_vector (60 downto 0);
signal T: std_logic_vector (16 downto 0);

begin

-- Finding the partial products of the 8 bit number:
result: partial_product port map (A (7 downto 0),B (7 downto 0), P (64 downto 1));

-- Performing addition on partial products using half adders and full adders:

-- First Phase:
M(0)<=P(1);
HA1: half_adder1 port map (P(2), P(9), M(1), C(0));
FA1: full_adder1 port map (P(3), P(10),C(0), S(0), C(1));
HA2: half_adder1 port map (P(17), S(0), M(2), C(2));
FA2: full_adder1 port map (P(4), P(11),C(1), S(1), C(3));
FA3: full_adder1 port map (P(18), S(1),C(2), M(3), C(4));
FA4: full_adder1 port map (P(5), P(12),C(3), S(2), C(5));
FA5: full_adder1 port map (P(19), S(2),C(4), M(4), C(6));
FA6: full_adder1 port map (P(6), P(13),C(5), S(3), C(7));
```

FA7: full_adder1 port map (P(20), S(3),C(6), M(5), C(8));

FA8: full_adder1 port map (P(7), P(14),C(7), S(4), C(9));

FA9: full_adder1 port map (P(21), S(4),C(8), M(6), C(10));

FA10: full_adder1 port map (P(8), P(15),C(9), S(5), C(11));

FA11: full_adder1 port map (P(22), S(5),C(10), M(7), C(12));

FA12: full_adder1 port map (P(16), P(23),C(11), S(6), C(13));

HA3: half_adder1 port map (C(12), S(6), M(8), C(14));

FA13: full_adder1 port map (P(27), C(13),C(14), M(9), C(15));

M(10)<=C(15);

--Second Phase:

M(11)<=P(25);

HA4: half_adder1 port map (P(26), P(33), M(12), C(16));

FA14: full_adder1 port map (P(27), P(34),C(16), S(7), C(17));

HA5: half_adder1 port map (P(41), S(7), M(13), C(18));

FA15: full_adder1 port map (P(28), P(35),C(17), S(8), C(19));

FA16: full_adder1 port map (P(42), S(8),C(18), M(14), C(20));

FA17: full_adder1 port map (P(29), P(36),C(19), S(9), C(21));

FA18: full_adder1 port map (P(43), S(9),C(20), M(15), C(22));

FA19: full_adder1 port map (P(30), P(37),C(21), S(10), C(23));

FA20: full_adder1 port map (P(44), S(10),C(22), M(16), C(24));

FA21: full_adder1 port map (P(31), P(38),C(23), S(11), C(25));

FA22: full_adder1 port map (P(45), S(11),C(24), M(17), C(26));

FA23: full_adder1 port map (P(32), P(39),C(25), S(12), C(27));

FA24: full_adder1 port map (P(46), S(12),C(26), M(18), C(28));

FA25: full_adder1 port map (P(40), P(47),C(27), S(13), C(29));

HA6: half_adder1 port map (C(28), S(13), M(19), C(30));

FA26: full_adder1 port map (P(48), C(29),C(30), M(20), C(31));

```
M(21)<=C(31);



--third phase:


M(22)<=M(0);
M(23)<=M(1);
M(24)<=M(2);
HA7: half_adder1 port map (M(3), M(11), M(25), C(32));
FA27: full_adder1 port map (M(4), M(12),C(32), M(26), C(33));
FA28: full_adder1 port map (M(5), M(13),C(33), M(27), C(34));
FA29: full_adder1 port map (M(6), M(14),C(34), S(14), C(35));
HA8 : half_adder1 port map (P(49),S(14), M(28), C(36));
FA30: full_adder1 port map (M(7), M(15),C(35), S(15), C(37));
FA31: full_adder1 port map (P(50), S(15),C(36), M(29), C(38));
FA32: full_adder1 port map (M(8), M(16),C(37), S(16), C(39));
FA33: full_adder1 port map (P(51), S(16),C(38), M(30), C(40));
FA34: full_adder1 port map (M(9), M(17),C(39), S(17), C(41));
FA35: full_adder1 port map (P(52), S(17),C(40), M(31), C(42));
FA36: full_adder1 port map (M(10), M(18),C(41), S(18), C(43));
FA37: full_adder1 port map (P(53), S(18),C(32), M(32), C(44));
FA38: full_adder1 port map (M(19), C(43),P(54), S(19), C(45));
HA9 : half_adder1 port map (C(44),S(19), M(33), C(46));
FA39: full_adder1 port map (M(20), C(45),P(55), S(20), C(47));
HA10 : half_adder1 port map (C(46),S(20), M(34), C(48));
FA40: full_adder1 port map (M(21), C(47),P(56), S(21), C(49));
HA11 : half_adder1 port map (C(48),S(21), M(35), C(50));
HA12 : half_adder1 port map (C(49),C(50), M(36), C(51));
```

```
M(37)<=C(51);


--Fourth Phase:
T(0)<=M(22);
T(1)<=M(23);
T(2)<=M(24);
T(3)<=M(25);
T(4)<=M(26);
T(5)<=M(27);
T(6)<=M(28);
HA13 : half_adder1 port map (M(29),P(57), T(7), C(52));
FA41: full_adder1 port map (M(30), P(58),C(52), T(8), C(53));
FA42: full_adder1 port map (M(31), P(59),C(53), T(9), C(54));
FA43: full_adder1 port map (M(32), P(60),C(54), T(10), C(55));
FA44: full_adder1 port map (M(33), P(61),C(55), T(11), C(56));
FA45: full_adder1 port map (M(34), P(62),C(56), T(12), C(57));
FA46: full_adder1 port map (M(35), P(63),C(57), T(13), C(58));
FA47: full_adder1 port map (M(36), P(64),C(58), T(14), C(59));
HA14 : half_adder1 port map (M(37),C(59), T(15), C(60));


Mul16by8: multiplier port map( T(15 downto 0), A ( 7 downto 0), Z( 23 downto 0));
end mul3;
```

5.¼ Shifter:

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


entity shift is
port ( A : in std_logic_vector (23 downto 0);
```

```
    Z: out std_logic_vector (23 downto 0));
end shift;

architecture shift1 of shift is

--swapping of variables
begin

Z(0) <= A(2);
Z(1) <= A(3);
Z(2) <= A(4);
Z(3) <= A(5);
Z(4) <= A(6);
Z(5) <= A(7);
Z(6) <= A(8);
Z(7) <= A(9);
Z(8) <= A(10);
Z(9) <= A(11);
Z(10) <= A(12);
Z(11) <= A(13);
Z(12) <= A(14);
Z(13) <= A(15);
Z(14) <= A(16);
Z(15) <= A(17);
Z(16) <= A(18);
Z(17) <= A(19);
Z(18) <= A(20);
Z(19) <= A(21);
```

```
Z(20) <= A(22);
Z(21) <= A(23);
Z(22) <= '0';
Z(23) <= '0';
end ;
```

6.Adding “1”:

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

-- Entity Declaration:
entity add1 is
        port( A  : in std_logic_vector (23 downto 0);
         Z : out std_logic_vector (23 downto 0));
end add1;
architecture dataflow of add1 is
component rca22
  port( A, B      : in std_logic_vector (23 downto 0);
                    carry_in        : in std_logic;
                    sum             : out std_logic_vector (23 downto 0);
                    carry_out : out std_logic);
end component;

--signal declarations:
signal C: std_logic_vector (1 downto 0);
signal sum: std_logic_vector (23 downto 0);
signal B: std_logic_vector (23 downto 0);
begin
B <= "000000000000000000000001";
```

```
C(0)<='0';
RCA1: rca22 port map(A(23 downto 0),B(23 downto 0),C(0),sum(23 downto 0),C(1));
Z(0)<=sum(0);
Z(1)<=sum(1);
Z(2)<=sum(2);
Z(3)<=sum(3);
Z(4)<=sum(4);
Z(5)<=sum(5);
Z(6)<=sum(6);
Z(7)<=sum(7);
Z(8)<=sum(8);
Z(9)<=sum(9);
Z(10)<=sum(10);
Z(11)<=sum(11);
Z(12)<=sum(12);
Z(13)<=sum(13);
Z(14)<=sum(14);
Z(15)<=sum(15);

Z(16)<=sum(16);
Z(17)<=sum(17);
Z(18)<=sum(18);
Z(19)<=sum(19);
Z(20)<=sum(20);
Z(21)<=sum(21);
Z(22)<=sum(22);
Z(23)<=sum(23);
end dataflow;
```

7.Combinational Logic of Multiplier, Adder and Shifter:

7.1 Final Result of Wallace Tree Multiplier (1/4 (A^2*B)+1):

```
library IEEE;
use ieee.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity main3 is
        port( A ,B : in std_logic_vector (7 downto 0);
            clock: in std_logic;
            load: in std_logic;
            clear: in std_logic;
             Z     : out std_logic_vector (23 downto 0);
            end_flag : out std_logic);
end main3;

architecture operation of main3 is

-- Component Declaration of 8-bit register:
Component reg_8_bit
        port( data_in : in std_logic_vector (7 downto 0);
                clock, load, clear: in std_logic;
                data_out : out std_logic_vector (7 downto 0));
end Component;

-- Component Declaration of 16-bit register:
Component reg_16_bit
        port( data_in : in std_logic_vector (15 downto 0);
```

```
                clock, load, clear: in std_logic;
                data_out : out std_logic_vector (15 downto 0));
end Component;

-- Component Declaration of 24-bit register:
Component reg_24_bit
        port( data_in : in std_logic_vector (23 downto 0);
                clock, load, clear: in std_logic;
                data_out : out std_logic_vector (23 downto 0));
end Component;

-- Component Declaration of multiplier:
Component multiplier3
        port( A ,B      : in std_logic_vector (7 downto 0);
                Z       : out std_logic_vector (23 downto 0));
end Component;

-- Component Declaration of shifter:
Component shift
        port( A: in std_logic_vector (23 downto 0);
                Z: out std_logic_vector (23 downto 0));
end Component;

-- Component Declaration of addition:
component add1 is
        port( A  : in std_logic_vector (23 downto 0);
                Z       : out std_logic_vector (23 downto 0));
end component;
```

```
-- Signal Declarations:

-- Signals for 8-bit Register:
signal A2 : std_logic_vector (7 downto 0);
signal B2 : std_logic_vector (7 downto 0);

-- Signals for multiplier:
signal Z1 : std_logic_vector(23 downto 0);

-- Signals for shifting:
signal Z2 : std_logic_vector(23 downto 0);

-- Signals for adding:
signal Z3 : std_logic_vector(23 downto 0);

begin

-- Store the 8-bit number into a 8-bit register:
RegA: reg_8_bit port map ( A(7 downto 0), clock, load, clear, A2(7 downto 0));
RegB: reg_8_bit port map ( B(7 downto 0), clock, load, clear, B2(7 downto 0));

-- Perform the operation:
mul3: multiplier3 port map (A2(7 downto 0),B2(7 downto 0), Z1(23 downto 0));


--Shifting operation:
shift1: shift port map( Z1(23 downto 0),Z2(23 downto 0));
```

--Addition operation:

adding: add1 port map (Z2(23 downto 0),Z3(23 downto 0));

-- Store the 24-bit number into a 24-bit register:

RegZ: reg_24_bit port map (Z3(23 downto 0), clock, load, clear, Z(23 downto 0));

end_flag <= '0' when (load = '0' or clear='1') else '1';

end operation;

8. Logic of Multiplier, Adder and Shifter with pipelining and expansion of method for 16-bit operand:

8.1 MUX (2:1)

use ieee.std_logic_1164.all;

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity proj_mux2to1 is
  generic(width : integer := 8);
  port(
    a   : in  std_logic_vector(width-1 downto 0);
    b   : in  std_logic_vector(width-1 downto 0);
    sel : in  std_logic;
    o   : out std_logic_vector(width-1 downto 0));
  end proj_mux2to1;
```

```
architecture mux of proj_mux2to1 is
begin
  process(a,b,sel)
    begin
      if sel='0' then
        o <= a;
      else
        o <= b;
      end if;
    end process;
end mux;
```

8.2 D-Flip Flop

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity proj_dff is
  generic(
        delay : time := 1 ns);
  port(
    d    : in  std_logic;
    en   : in  std_logic;
    clk  : in  std_logic;
    q    : out std_logic
    );
end proj_dff;

architecture proj_dff_arch of proj_dff is
```

```
begin
   process (clk) is
   begin
      if rising_edge(clk) then
         if (en ='1') then
            q <= d;
           end if;
         end if;
   end process;
end architecture proj_dff_arch;
```

8.3 D-Flip Flop Vector

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity proj_dff_v is
  generic(width : integer := 1;
          delay : time := 1 ns);
  port(
    d    : in  std_logic_vector(width-1 downto 0);
    en   : in  std_logic;
    clk  : in  std_logic;
    q    : out std_logic_vector(width-1 downto 0));
  end proj_dff_v;

  architecture proj_dff_v_arch of proj_dff_v is
  begin
    process (clk) is
```

```
    begin
      if rising_edge(clk) then
        if (en ='1') then
          q <= d;
        end if;
      end if;
    end process;
  end architecture proj_dff_v_arch;
```

8.4 Edge detector

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity proj_edge is
  generic(del   : time := 2 ns;
          pos   : integer := 1);
   port (
     sig      :in std_logic;
     edg      :out std_logic
   );
end proj_edge;

architecture proj_edge_arch of proj_edge is
   signal sig_d, sig_n, temp  :std_logic;
  begin
sig_d <= sig after del;
sig_n <= not(sig);
```

```
temp <= sig_d xor sig_n;
process(sig)
 begin
 if(pos = 1) then
    edg <= (sig and not(sig_d));
  else
    edg <= (not(sig) and sig_d);
 end if;
end process;
end proj_edge_arch;
```

8.5 Carry Save Adder

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_arith.all;


entity proj_csa is
 generic(width : integer := 6);
 Port (
   x : in std_logic_vector (width - 1 downto 0);
   y : in  std_logic_vector (width - 1 downto 0);
   z : in  std_logic_vector (width - 1 downto 0);
   s : out std_logic_vector (width - 1 downto 0);
   c : out std_logic_vector (width - 1 downto 0)
    );
end proj_csa;
architecture proj_csa_arch of proj_csa is
 begin
```

```
  gen_csa : for i in (width - 1) downto 0 generate


      s(i)<= ((x(i) xor y(i)) xor z(i));
      c(i) <= (x(i) and y(i)) or (x(i) and z(i)) or (y(i) and z(i));
end generate gen_csa;


  end proj_csa_arch;
```

8.6 Full Adder that works for all width

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity proj_fa is
  generic(width : natural :=16);

  port (
    x     : in std_logic_vector(width-1 downto 0);
    y     : in std_logic_vector(width-1 downto 0);
    cin   : in std_logic;
    s     : out std_logic_vector(width-1 downto 0);
    cout  : out std_logic
    );
end proj_fa ;

architecture proj_fa_arch of proj_fa is
  signal c : std_logic_vector(width - 1 downto 0);
begin
gen_fa :
```

```
for i in (width - 1) downto 0 generate
  lower_bit : if (i = 0) generate
              s(i)    <= x(i) xor y(i) xor cin;
              c(i)    <= (x(i) and y(i)) or ((x(i) xor y(i)) and cin);
        end generate lower_bit;
  upper_bit : if (i > 0) generate
              s(i)    <= x(i) xor y(i) xor c(i-1);
                c(i) <= (x(i) and y(i)) or ((x(i) xor y(i)) and c(i-1));
    end generate upper_bit;
end generate gen_fa;
end proj_fa_arch;
```

8.7 Multiplier (8*8)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_arith.all;



entity proj_mult8 is
  port (
    a,b : in std_logic_vector(7 downto 0);
    z   : out std_logic_vector(15 downto 0)
  );
end proj_mult8;

architecture proj_mult8_arch of proj_mult8 is

component proj_csa
```

```
  generic(width : integer := 6);
port  (
x : in std_logic_vector (width - 1 downto 0);
    y : in  std_logic_vector (width - 1 downto 0);
    z : in  std_logic_vector (width - 1 downto 0);
    s : out std_logic_vector (width - 1 downto 0);
    c : out std_logic_vector (width - 1 downto 0)
     );
  end component;

  component proj_fa
    generic(width : natural :=16);
  port (
  x     : in std_logic_vector(width-1 downto 0);
     y     : in std_logic_vector(width-1 downto 0);
     cin   : in std_logic;
     s     : out std_logic_vector(width-1 downto 0);
     cout  : out std_logic
     );
  end component;


type array1 is array (7 downto 0) of std_logic_vector(7 downto 0);
  signal ab : array1;
 signal s1, c1, s3, c3, s6, c6 : std_logic_vector(11 downto 0);
  signal s2, c2, s4, c4, s5, c5 : std_logic_vector(5 downto 0);
  signal x, y : std_logic_vector(15 downto 0);
```

```
--Temp Concate signals

 signal temp_x1,temp_y1,temp_z1,temp_x3,temp_z3,temp_y6,temp_z6        : std_logic_vector(11 downto 0);

 signal temp_z2,temp_y4,temp_z4,temp_y5                                : std_logic_vector(5 downto 0);

 begin

 gen_mult : for i in 7 downto 0 generate

   ab(i) <= a(7 downto 0) and (7 downto 0 => b(i));

 end generate gen_mult;


-- Temporary signals

--temp_x1  <= (ab(6)(7 to 7) & ab(5)(7 to 7) & ab(4)(7 to 7) & ab(5)(5 to 5) & ab(6)(3 to 3) & ab(7)(1 to 1) & ab(0)(7 downto 2));

temp_x1  <= (ab(6)(7 downto 7) & ab(5)(7 downto 7) & ab(4)(7 downto 7) & ab(5)(5 downto 5) & ab(6)(3 downto 3) & ab(7)(1 downto 1) & ab(0)(7 downto 2));

temp_y1  <= (ab(7)(6 downto 6) & ab(6)(6 downto 6) & ab(5)(6 downto 6) & ab(6)(4 downto 4)) & ab(7)(2 downto 2) & ab(1)(7 downto 1);

temp_z1  <= ('0' & ab(7)(5 downto 5) & ab(6)(5 downto 5) & ab(7)(3 downto 3) & ab(2)(7 downto 0));

temp_z2  <= ('0' & ab(5)(4 downto 0));

temp_x3  <= (ab(7)(7 downto 7) & s1(11 downto 1));

temp_z3  <= (''000'' & ab(7)(4 downto 4) & s2(5 downto 0) & ab(3)(1 downto 0));

temp_y4  <= (''000'' & ab(6)(2 downto 0));

temp_z4  <= (''0000'' & ab(7)(0 downto 0) & '0');

temp_y5  <= ('0' & s4(5 downto 1));

temp_y6  <= (c3(10 downto 9) & s5(5 downto 0) & c3(2 downto 0) & '0');

temp_z6  <= ('0' & c5(5 downto 0) & '0' & s4(0) & '0' & ab(4)(0 downto 0) & '0');


-- Reduction 1

 proj_csa_1 : proj_csa generic map (12)  port map  (x => temp_x1, y => temp_y1, z => temp_z1,s => s1,c => c1);
```

```
proj_csa_2 : proj_csa generic map (6)  port map  (x => ab(3)(7 downto 2),
                                                  y => ab(4)(6 downto 1),
                                                  z => temp_z2,
                                                  s => (s2),
                                                  c => (c2));
-- Reduction 2
proj_csa_3 : proj_csa generic map (12)  port map  ( x => temp_x3,
                                                    y => c1(11 downto 0),
                                                    z => temp_z3,
                                                    s => (s3),
                                                    c => (c3));

proj_csa_4 : proj_csa generic map (6)  port map   ( x => c2(5 downto 0),
                                                    y => temp_y4,
                                                    z => temp_z4,
                                                    s => s4,
                                                    c => c4);
-- Reduction 3
proj_csa_5 : proj_csa generic map (6)  port map    ( x => c3(8 downto 3),
                                                     y => temp_y5,
                                                     z => c4(5 downto 0),
                                                     s => s5,
                                                     c => c5);

proj_csa_6 : proj_csa generic map (12)  port map    ( x => s3(11 downto 0),
                                                      y => temp_y6,
                                                      z => temp_z6,
                                                      s => s6,
```

```
                                        c => c6);
-- Reduction 4 Final


                                        x <= (c3(11) & s6(11 downto 0) & s1(0) & ab(0)(1 downto 1) & ab(0)(0
downto 0));
                                        y <= (c6(11 downto 0) & "00" & ab(1)(0 downto 0) & '0');
proj_fa_1 : proj_fa generic map (16)  port map        ( x => x, y => y, cin => '0', s => z );


end proj_mult8_arch;
```

9 Final Result of Wallace Tree Multiplier extra features(1/4 (A^2*B)+1): (Refer Appendix III for Test Bench)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity proj_toplevel is
generic(width : integer := 8);
port (
  a     : in std_logic_vector(width - 1 downto 0);
  b     : in std_logic_vector(width - 1 downto 0);
  clk   : in std_logic;
  load  : in std_logic;
  clr   : in std_logic;
  endF  : out std_logic;
  z     : out std_logic_vector(width*2 - 1 downto 0)
```

```
  );
end proj_toplevel;

architecture proj_toplevel_arch of proj_toplevel is

--Edge
component proj_edge
  generic(del   : time := 2 ns;
          pos   : integer := 1);
   port (
      sig      :in std_logic;
      edg      :out std_logic
   );
 end component;

--Dff_v
component proj_dff_v
  generic(width : integer := 1;
          delay : time := 1 ns);
  port(
    d    : in  std_logic_vector(width-1 downto 0);
    en   : in  std_logic;
    clk  : in  std_logic;
    q    : out std_logic_vector(width-1 downto 0));

  end component;
  --Dff
  component proj_dff
```

```
    generic(
          delay : time := 1 ns);
    port(
      d    : in  std_logic;
      en   : in  std_logic;
      clk  : in  std_logic;
      q    : out std_logic
      );
    end component;
--Mult8
component proj_mult8
  port (
   a,b : in std_logic_vector(7 downto 0);
   z   : out std_logic_vector(15 downto 0)
 );
 end component;

--Fa
component proj_fa
  generic(width : natural :=16);

  port (
    x     : in std_logic_vector(width-1 downto 0);
    y     : in std_logic_vector(width-1 downto 0);
    cin   : in std_logic;
    s     : out std_logic_vector(width-1 downto 0);
    cout  : out std_logic
    );
```

```
  end component;


--Mux
component proj_mux2to1
  generic(width : integer := 8);
  port(
    a   : in  std_logic_vector(width-1 downto 0);
    b   : in  std_logic_vector(width-1 downto 0);
    sel : in  std_logic;
    o   : out std_logic_vector(width-1 downto 0));
  end component;


 -- signal declaration
signal a_str, b_str, a_reg, b_reg, f1_b                                  : std_logic_vector(width - 1 downto 0);
signal c_pos, c_neg, ld_pos,  in_ld, ld_neg                              : std_logic;
--signal ld_neg : std_logic_vector(0 downto 0);
signal e_flag, c_flag, f1_e_flag, f1_c_flag, f2_e_flag, f2_c_flag, f3_e_flag, f3_c_flag : std_logic;
signal f0_l_en, f0_c_en, f1_en, f2_en, f3_en                             : std_logic;
signal a_sq, f1_a_sq                                                     : std_logic_vector(width*2 - 1 downto 0);
signal f1_ab_lsb, f1_ab_msb, f2_ab_lsb, f2_ab_msb                        : std_logic_vector(width*2 - 1
downto 0 );
signal f2_ab_msb_c, f2_ab_lsb_c                                          : std_logic_vector(width*2 + 8 - 1
downto 0);
signal f2_ab_mult                                                        : std_logic_vector(width*3 - 1 downto 0);
signal f2_ab_drop, f3_ab_drop                                            : std_logic_vector(width*3 - 3 downto
0);
signal f3_ab_incr, f3_z_out, f3_ab_drop_a                                : std_logic_vector(width*3 - 3
downto 0);
```

```
begin
  a_str <= (width - 1 downto 0 =>  (not clr)) and a;
  b_str <= (width - 1 downto 0 =>  (not clr)) and b;

f0_pedge_ld : proj_edge generic map (pos => 1) port map (sig => load,
                                        edg => ld_pos);
f0_nedge_ld : proj_edge generic map(pos => 0 ) port map (
                        sig => load,
                        edg => ld_neg);
f0_pedge_c : proj_edge generic map(pos => 1)port map (
                        sig => clr,
                        edg => c_pos);
f0_nedge_c : proj_edge  generic map(pos => 0) port map (
                        sig => clr,
                        edg => c_neg);
  in_ld <= ld_neg or  clr;

f0_dff_ain : proj_dff_v generic map(width => width)  port map(d => a_str,
                                                clk => in_ld,
                                                en => '1',
                                                q => a_reg);

f0_dff_bin : proj_dff_v generic map(width => width)  port map(d => b_str,
                                                clk => in_ld,
                                                en => '1',
                                                q => b_reg);

f0_l_en <= (clk and ld_pos) or ld_neg;
```

```
f0_ld_ain : proj_dff  port map(d => ld_neg,
                                clk => f0_l_en,
                                en => '1',
                                q => e_flag);

--f0_c_en <= (clk and c_neg) or c_pos;
f0_c_en <= c_pos;
f0_dff_clr : proj_dff  port map(d => clr,
                                clk => f0_c_en,
                                en  => '1',
                                q   => c_flag);


-- Equation for Z = 1/4 [A *A * B] +1

-- A*A

f0_mult8_sq : proj_mult8 port map(a => a_reg, b => a_reg, z => a_sq);
f1_en <= e_flag or c_flag;

-- Flop stage 1

f1_sq : proj_dff_v generic map(width => width*2) port map(d => a_sq, clk => clk, en => f1_en, q =>
f1_a_sq);
f1_bin : proj_dff_v generic map(width => width) port map(d => b_reg, clk => clk, en => f1_en, q => f1_b);
f1_e : proj_dff  port map(d => e_flag, clk => clk, en => '1', q => f1_e_flag);
f1_clr : proj_dff port map(d => c_flag, clk => clk, en => '1', q => f1_c_flag);
```

```
-- *B - partial mult

f1_mult8_bl : proj_mult8  port map(a => f1_a_sq(7 downto 0), b => f1_b, z => f1_ab_lsb);

f1_mult8_bm : proj_mult8  port map(a => f1_a_sq(15 downto 8), b => f1_b, z => f1_ab_msb);


 --flop stage 2

f2_en <= (f1_e_flag or f1_c_flag);


f2_lsb : proj_dff_v generic map(width => width*2) port map(d => f1_ab_lsb, clk => clk, en => f2_en, q =>
f2_ab_lsb);


f2_msb : proj_dff_v generic map(width => width*2) port map(d => f1_ab_msb, clk => clk, en => f2_en, q
=> f2_ab_msb);


f2_e : proj_dff port map(d => f1_e_flag, clk => clk, en => '1', q => f2_e_flag);


f2_clr : proj_dff port map(d => f1_c_flag, clk => clk, en => '1', q => f2_c_flag);


-- * B - result

f2_ab_msb_c <= f2_ab_msb & "00000000";

f2_ab_lsb_c <= "00000000" & f2_ab_lsb;

f2_fa_mult : proj_fa  generic map(width => width*3) port map(x => f2_ab_msb_c, y => f2_ab_lsb_c, cin
=> '0', s => f2_ab_mult);


--1/4 [A * A * B]


 f2_ab_drop <= f2_ab_mult(width*3 - 1 downto 2);


 --Flop stage 2
```

```
 f3_en <= (f2_e_flag or f2_c_flag);


 f3_drp : proj_dff_v generic map(width => width*3 - 2) port map (d => f2_ab_drop, clk => clk, en =>
f3_en, q => f3_ab_drop);


 f3_e : proj_dff port map(d => f2_e_flag,  clk => clk, en => '1',  q => f3_e_flag);


 f3_clr : proj_dff   port map(d => f2_c_flag,  clk => clk, en => '1',  q => f3_c_flag);


-- 1/4 [A * A * B] + 1

 --f3_ab_drop_a <= (width*3 - 3 => '0') & '1';

 f3_ab_drop_a <= (0 => '1', others => '0');

 f3_incr : proj_fa generic map(width => width*3 - 2) port map ( x => f3_ab_drop, y => f3_ab_drop_a, cin
=> '0', s => f3_ab_incr);


 --Output


 f3_mux : proj_mux2to1 generic map(width => width*3 - 2) port map(a => (others => '0'), b => f3_ab_incr,
sel => f3_c_flag, o => f3_z_out);


z <= f3_z_out(width*2-1 downto 0);

endF <= f3_e_flag;


 end proj_toplevel_arch;
```

9.1 Multiplier Implementation for 16-bit operand:

```
library ieee;

use ieee.std_logic_1164.all;

use ieee.numeric_std.all;

use ieee.std_logic_arith.all;
```

```
use ieee.std_logic_unsigned.all;


entity proj_toplevel is
generic(width : integer := 16;
        widthm : integer := 8 );
port (
  a     : in std_logic_vector(width - 1 downto 0);
  b     : in std_logic_vector(width - 1 downto 0);
  clk   : in std_logic;
  load  : in std_logic;
  clr   : in std_logic;
  endF  : out std_logic;
  z     : out std_logic_vector(width*3 - 3 downto 0)
  );
end proj_toplevel;


architecture proj_toplevel_arch of proj_toplevel is


--Edge
component proj_edge
  generic(del   : time := 2 ns;
          pos   : integer := 1);
   port (
      sig      :in std_logic;
      edg      :out std_logic
   );
 end component;
```

```
--Dff_v
component proj_dff_v
  generic(width : integer := 1;
          delay : time := 1 ns);
  port(
    d    : in  std_logic_vector(width-1 downto 0);
    en   : in  std_logic;
    clk  : in  std_logic;
    q    : out std_logic_vector(width-1 downto 0));

  end component;
  --Dff
  component proj_dff
    generic(
           delay : time := 1 ns);
    port(
      d    : in  std_logic;
      en   : in  std_logic;
      clk  : in  std_logic;
      q    : out std_logic
      );
    end component;
--Mult8
component proj_mult8
  port (
   a,b : in std_logic_vector(7 downto 0);
   z   : out std_logic_vector(15 downto 0)
 );
```

```
 end component;


--Fa
component proj_fa
  generic(width : natural :=16);


  port (
    x     : in std_logic_vector(width-1 downto 0);
    y     : in std_logic_vector(width-1 downto 0);
    cin   : in std_logic;
    s     : out std_logic_vector(width-1 downto 0);
    cout  : out std_logic
    );
  end component;


--Mux
component proj_mux2to1
  generic(width : integer := 8);
  port(
    a   : in  std_logic_vector(width-1 downto 0);
    b   : in  std_logic_vector(width-1 downto 0);
    sel : in  std_logic;
    o   : out std_logic_vector(width-1 downto 0));
  end component;


  -- signal declaration
signal a_str, b_str, a_reg, b_reg, f1_b                    : std_logic_vector(width - 1 downto 0);
signal c_pos, c_neg, ld_pos,  in_ld, ld_neg                : std_logic;
```

```
signal e_flag, c_flag, f1_e_flag, f1_c_flag, f2_e_flag, f2_c_flag, f3_e_flag, f3_c_flag : std_logic;

signal f0_l_en, f0_c_en, f1_en, f2_en, f3_en                                    : std_logic;

signal a_sq_0, a_sq_1, a_sq_2, a_sq_3                                           : std_logic_vector(widthm*2 - 1
downto 0);

signal a_sq_0_c, a_sq_1_c, a_sq_2_c, a_sq_3_c                                   : std_logic_vector(width*2 - 1
downto 0);

signal a_sq_lsb, a_sq_msb, a_sq, f1_a_sq                                        : std_logic_vector(width*2 - 1
downto 0);

signal f1_ab_lsb_0, f1_ab_lsb_1, f1_ab_lsb_2, f1_ab_lsb_3                       : std_logic_vector(widthm*2
- 1 downto 0 );

signal f1_ab_msb_0, f1_ab_msb_1, f1_ab_msb_2, f1_ab_msb_3                       :
std_logic_vector(widthm*2 - 1 downto 0 );

signal f1_ab_lsb_0_c, f1_ab_lsb_1_c, f1_ab_lsb_2_c, f1_ab_lsb_3_c               :
std_logic_vector(width*2 + widthm - 1 downto 0 );

signal f1_ab_msb_0_c, f1_ab_msb_1_c, f1_ab_msb_2_c, f1_ab_msb_3_c               :
std_logic_vector(width*2 + widthm - 1 downto 0 );

signal f1_ab_lsb_01, f1_ab_lsb_23, f1_ab_msb_01, f1_ab_msb_23                   :
std_logic_vector(width*2 + widthm - 1 downto 0 );

signal f1_ab_lsb, f1_ab_msb, f2_ab_lsb, f2_ab_msb                               : std_logic_vector(width*2 +
widthm - 1 downto 0 );

signal f2_ab_msb_c, f2_ab_lsb_c                                                 : std_logic_vector(width*2 + widthm
+ 8 - 1 downto 0);

signal f2_ab_mult                                                               : std_logic_vector(width*3 - 1 downto 0);

signal f2_ab_drop, f3_ab_drop                                                   : std_logic_vector(width*3 - 3 downto
0);

signal f3_ab_incr, f3_z_out, f3_ab_drop_a                                       : std_logic_vector(width*3 - 3
downto 0);


begin

  a_str <= (width - 1 downto 0 =>  (not clr)) and a;

  b_str <= (width - 1 downto 0 =>  (not clr)) and b;


f0_pedge_ld : proj_edge generic map (pos => 1) port map (sig => load, edg => ld_pos);
```

```
f0_nedge_ld : proj_edge generic map (pos => 0) port map (sig => load, edg => ld_neg);

f0_pedge_c  : proj_edge generic map (pos => 1) port map (sig => clr,  edg => c_pos);

f0_nedge_c  : proj_edge generic map (pos => 0) port map (sig => clr,  edg => c_neg);


in_ld <= ld_neg or  clr;


f0_dff_ain : proj_dff_v generic map(width => width)  port map(d => a_str, clk => in_ld, en => '1', q =>
a_reg);

f0_dff_bin : proj_dff_v generic map(width => width)  port map(d => b_str, clk => in_ld, en => '1', q =>
b_reg);


f0_l_en <= (clk and ld_pos) or ld_neg;


f0_ld_ain : proj_dff  port map(d => ld_neg, clk => f0_l_en, en => '1', q => e_flag);


f0_c_en <= c_pos;


f0_dff_clr : proj_dff  port map(d => clr, clk => f0_c_en, en  => '1', q   => c_flag);


-- Equation for Z = 1/4 [A *A * B] +1


-- A*A


f0_mult8_sq_0 : proj_mult8 port map(a => a_reg( 7 downto 0 ), b => a_reg( 7 downto 0 ), z => a_sq_0);

f0_mult8_sq_1 : proj_mult8 port map(a => a_reg(15 downto 8 ), b => a_reg( 7 downto 0 ), z => a_sq_1);

f0_mult8_sq_2 : proj_mult8 port map(a => a_reg( 7 downto 0 ), b => a_reg(15 downto 8 ), z => a_sq_2);

f0_mult8_sq_3 : proj_mult8 port map(a => a_reg(15 downto 8 ), b => a_reg(15 downto 8 ), z => a_sq_3);


a_sq_0_c <= x"0000" & a_sq_0;
```

```
a_sq_1_c <= x"00"   & a_sq_1 & x"00";

a_sq_2_c <= x"00"   & a_sq_2 & x"00";

a_sq_3_c <=          a_sq_3 & x"0000";


f0_sq_1 : proj_fa  generic map(width => width*2) port map(x => a_sq_1_c, y => a_sq_0_c, cin => '0', s =>
a_sq_lsb);

f0_sq_2 : proj_fa  generic map(width => width*2) port map(x => a_sq_3_c, y => a_sq_2_c, cin => '0', s =>
a_sq_msb);

f0_sq_3 : proj_fa  generic map(width => width*2) port map(x => a_sq_lsb, y => a_sq_msb, cin => '0', s =>
a_sq);


f1_en <= e_flag or c_flag;


-- Flop stage 1


f1_sq : proj_dff_v generic map(width => width*2) port map(d => a_sq, clk => clk, en => f1_en, q =>
f1_a_sq);

f1_bin : proj_dff_v generic map(width => width) port map(d => b_reg, clk => clk, en => f1_en, q => f1_b);

f1_e : proj_dff  port map(d => e_flag, clk => clk, en => '1', q => f1_e_flag);

f1_clr : proj_dff port map(d => c_flag, clk => clk, en => '1', q => f1_c_flag);


-- *B - partial mult


f1_mult8_bl0 : proj_mult8 port map(a => f1_a_sq( 7 downto 0 ), b => f1_b( 7 downto 0 ), z =>
f1_ab_lsb_0);

f1_mult8_bl1 : proj_mult8 port map(a => f1_a_sq(15 downto 8 ), b => f1_b( 7 downto 0 ), z =>
f1_ab_lsb_1);

f1_mult8_bl2 : proj_mult8 port map(a => f1_a_sq(23 downto 16), b => f1_b( 7 downto 0 ), z =>
f1_ab_lsb_2);

f1_mult8_bl3 : proj_mult8 port map(a => f1_a_sq(31 downto 24), b => f1_b( 7 downto 0 ), z =>
f1_ab_lsb_3);
```

```
f1_mult8_bm0 : proj_mult8 port map(a => f1_a_sq( 7 downto 0 ), b => f1_b(15 downto 8 ), z =>
f1_ab_msb_0);

f1_mult8_bm1 : proj_mult8 port map(a => f1_a_sq(15 downto 8 ), b => f1_b(15 downto 8 ), z =>
f1_ab_msb_1);

f1_mult8_bm2 : proj_mult8 port map(a => f1_a_sq(23 downto 16), b => f1_b(15 downto 8 ), z =>
f1_ab_msb_2);

f1_mult8_bm3 : proj_mult8 port map(a => f1_a_sq(31 downto 24), b => f1_b(15 downto 8 ), z =>
f1_ab_msb_3);


f1_ab_lsb_0_c <= x"000000" & f1_ab_lsb_0;

f1_ab_lsb_1_c <= x"0000"   & f1_ab_lsb_1 & x"00";

f1_ab_lsb_2_c <= x"00"     & f1_ab_lsb_2 & x"0000";

f1_ab_lsb_3_c <=           f1_ab_lsb_3 & x"000000";

f1_ab_msb_0_c <= x"000000" & f1_ab_msb_0;

f1_ab_msb_1_c <= x"0000"   & f1_ab_msb_1 & x"00";

f1_ab_msb_2_c <= x"00"     & f1_ab_msb_2 & x"0000";

f1_ab_msb_3_c <=           f1_ab_msb_3 & x"000000";


f1_madd_l01 : proj_fa  generic map(width => width*2 + widthm) port map(x => f1_ab_lsb_1_c, y =>
f1_ab_lsb_0_c, cin => '0', s => f1_ab_lsb_01);

f1_madd_l23 : proj_fa  generic map(width => width*2 + widthm) port map(x => f1_ab_lsb_3_c, y =>
f1_ab_lsb_2_c, cin => '0', s => f1_ab_lsb_23);

f1_madd_lsb : proj_fa  generic map(width => width*2 + widthm) port map(x => f1_ab_lsb_23,  y =>
f1_ab_lsb_01,  cin => '0', s => f1_ab_lsb);

f1_madd_m01 : proj_fa  generic map(width => width*2 + widthm) port map(x => f1_ab_msb_1_c, y =>
f1_ab_msb_0_c, cin => '0', s => f1_ab_msb_01);

f1_madd_m23 : proj_fa  generic map(width => width*2 + widthm) port map(x => f1_ab_msb_3_c, y =>
f1_ab_msb_2_c, cin => '0', s => f1_ab_msb_23);

f1_madd_msb : proj_fa  generic map(width => width*2 + widthm) port map(x => f1_ab_msb_23,  y =>
f1_ab_msb_01,  cin => '0', s => f1_ab_msb);


 --flop stage 2
```

```
f2_en <= (f1_e_flag or f1_c_flag);

f2_lsb : proj_dff_v generic map(width => width*2 + widthm) port map(d => f1_ab_lsb, clk => clk, en => f2_en, q => f2_ab_lsb);

f2_msb : proj_dff_v generic map(width => width*2 + widthm) port map(d => f1_ab_msb, clk => clk, en => f2_en, q => f2_ab_msb);

f2_e : proj_dff port map(d => f1_e_flag, clk => clk, en => '1', q => f2_e_flag);

f2_clr : proj_dff port map(d => f1_c_flag, clk => clk, en => '1', q => f2_c_flag);

-- * B - result

f2_ab_msb_c <= f2_ab_msb & "00000000";

f2_ab_lsb_c <= "00000000" & f2_ab_lsb;

f2_fa_mult : proj_fa  generic map(width => width*3) port map(x => f2_ab_msb_c, y => f2_ab_lsb_c, cin => '0', s => f2_ab_mult);

--1/4 [A * A * B]

 f2_ab_drop <= f2_ab_mult(width*3 - 1 downto 2);

 --Flop stage 2

 f3_en <= (f2_e_flag or f2_c_flag);

 f3_drp : proj_dff_v generic map(width => width*3 - 2) port map (d => f2_ab_drop, clk => clk, en => f3_en, q => f3_ab_drop);
```

```
f3_e : proj_dff port map(d => f2_e_flag,  clk => clk, en => '1',  q => f3_e_flag);

f3_clr : proj_dff   port map(d => f2_c_flag,  clk => clk, en => '1',  q => f3_c_flag);

-- 1/4 [A * A * B] + 1

f3_ab_drop_a <= (0 => '1', others => '0');

f3_incr : proj_fa generic map(width => width*3 - 2) port map ( x => f3_ab_drop, y => f3_ab_drop_a, cin => '0', s => f3_ab_incr);

--Output

f3_mux : proj_mux2to1 generic map(width => width*3 - 2) port map(a => (others => '0'), b => f3_ab_incr, sel => f3_c_flag, o => f3_z_out);

z <= f3_z_out;

endF <= f3_e_flag;

end proj_toplevel_arch;
```

Test bench

```
library ieee;

use ieee.std_logic_1164.all;

use ieee.numeric_std.all;

use ieee.std_logic_arith.all;

use ieee.std_logic_unsigned.all;
```

```
entity proj_final_tb is
  generic(width : integer := 16);
end proj_final_tb;

architecture testbench of proj_final_tb is

component proj_toplevel
generic(width : integer := 8);
port (
  a     : in std_logic_vector(width - 1 downto 0);
  b     : in std_logic_vector(width - 1 downto 0);
  clk   : in std_logic;
  load  : in std_logic;
  clr   : in std_logic;
  endF  : out std_logic;
  z     : out std_logic_vector(width*3 - 3 downto 0)
  );
end component;

signal clk,endF_tb,ld_tb,clr_tb : std_logic;
signal z_tb : std_logic_vector(width*3 - 3 downto 0);
signal a_tb,b_tb : std_logic_vector(width - 1 downto 0);
  constant clk_period : time := 10 ns;
begin -- Arch
    process
    begin
      clk <= '0';
      wait for clk_period/2;
```

clk <= '1';

wait for clk_period/2;

end process;

a_tb <= x"0000", x"0004" after 50 ns, x"0009" after 100 ns, x"FFFF" after 160 ns;

b_tb <= x"0000", x"0002" after 50 ns, x"0005" after 100 ns, x"FFFF" after 160 ns;

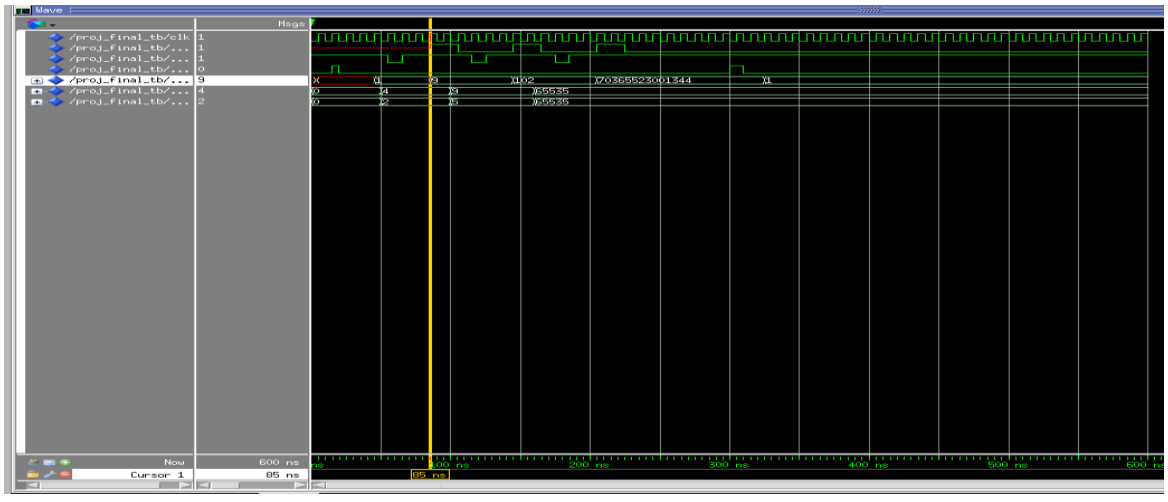
ld_tb <= '1', '0' after 55 ns, '1' after 65 ns, '0' after 115 ns, '1' after 125 ns, '0' after 175 ns, '1' after 185 ns;

clr_tb <= '0', '1' after 15 ns, '0' after 20 ns, '1' after 300 ns, '0' after 310 ns;

proj_toplevel_1 : proj_toplevel generic map (16) port map(a => a_tb, b => b_tb, clk => clk, load => ld_tb, clr => clr_tb, endF => endF_tb, z => z_tb);

end testbench;

Simulation Result:



Appendix II – Test Benches :

--Wallace Multiplier

```
Library ieee;
use ieee.std_logic_1164.all;
entity test_wallace_multiplier is
end test_wallace_multiplier;
architecture testbench of test_wallace_multiplier is
component multiplier3
port( A,B: in std_logic_vector(7 downto 0);
Z: out std_logic_vector(23 downto 0));
end component;

signal A,B: std_logic_vector(7 downto 0);
signal Z:std_logic_vector(23 downto 0);
begin
u1: multiplier3 port map(A(7 downto 0),B(7 downto 0),Z(23 downto 0));
process

begin
A<="00000000";
B<="00000000";
wait for 100 ns;
A<="00000010";
B<="00000010";
wait for 100 ns;
end process;
end testbench;

--xor
Library ieee;
```

```
use ieee.std_logic_1164.all;
entity test_xor is
end test_xor;
architecture testbench of test_xor is
signal a,b: std_logic;
signal c:std_logic;
component xor1
port(A,B: in std_logic;
Z: out std_logic);
end component;
begin
u1: xor1 port map(a,b,c);
process
begin
a<='0';
b<='0';
wait for 100 ns;
a<='0';
b<='1';
wait for 100 ns;
a<='1';
b<='0';
wait for 100 ns;
a<='1';
b<='1';
wait for 100 ns;
end process;
end testbench;
```

```
--or
Library ieee;
use ieee.std_logic_1164.all;
entity test_or is
end test_or;
architecture testbench of test_or is
signal a,b: std_logic;
signal c:std_logic;
component or1
port(A,B: in std_logic;
Z: out std_logic);
end component;
begin
u1: or1 port map(a,b,c);
process
begin
a<='0';
b<='0';
wait for 100 ns;
a<='0';
b<='1';
wait for 100 ns;
a<='1';
b<='0';
wait for 100 ns;
a<='1';
b<='1';
```

```
wait for 100 ns;
end process;
end testbench;

--not
library ieee;
use ieee.std_logic_1164.all;
entity test_not is
end test_not;
architecture structural of test_not is
component not_gate
port(A:in std_logic;
Z:out std_logic);
end component;
signal a,b: std_logic;
begin
u1: not_gate port map(a,b);
process
begin
a<='0';
wait for 100 ns;
a<='1';
wait for 100 ns;
end process;
end structural;

--full
library ieee;
```

```
use ieee.std_logic_1164.all;
entity test_fulladder is
end test_fulladder;
architecture testbench of test_fulladder is
signal d,e,f: std_logic;
signal s1,c1: std_logic;
component full_adder1 is
port(A,B,carry_in:in std_logic;
sum,carry_out: out std_logic);
end component;
begin
g1:full_adder1 port map(d,e,f,s1,c1);
process
begin
d<='0';
e<='0';
f<='0';
wait for 100 ns;
d<='0';
e<='0';
f<='1';
wait for 100 ns;
d<='0';
e<='1';
f<='0';
wait for 100 ns;
d<='0';
e<='1';
```

```
f<='1';
wait for 100 ns;
d<='1';
e<='0';
f<='0';
wait for 100 ns;
d<='1';
e<='0';
f<='1';
wait for 100 ns;
d<='1';
e<='1';
f<='0';
wait for 100 ns;
d<='1';
e<='1';
f<='1';
wait for 100 ns;
end process;
end testbench;

--and
library ieee;
use ieee.std_logic_1164.all;
entity and_gate_test is
end and_gate_test;
architecture testbench of and_gate_test is
signal a,b : std_logic ;
```

```
signal c :std_logic ;
component and1
port(A,B: in std_logic;
Z:out std_logic);
end component;
begin
h1: and1 port map(a,b,c);
process
begin
a<= '0';
b<='0';
wait for 100 ns;
a<= '0';
b<='1';
wait for 100 ns;
a<= '1';
b<='0';
wait for 100 ns;
a<= '1';
b<='1';
wait for 100 ns;
end process;
end testbench;

--8 bit
library ieee;
use ieee.std_logic_1164.all;
entity test_8bitreg is
```

```
end test_8bitreg;
architecture testbench of test_8bitreg is
signal data_in : STD_LOGIC_VECTOR (7 downto 0);
signal clock,clear,load : STD_LOGIC;
signal data_out : STD_LOGIC_VECTOR (7 downto 0);
component reg_8_bit
port(data_in :in std_logic_vector(7 downto 0);
clock,clear,load:in std_logic;
data_out:out std_logic_vector(7 downto 0));
end component;
begin
u1: reg_8_bit port map(data_in,clock,clear,load,data_out);
process
begin

data_in<="00000000";
clock<='1';
clear<='0';
load<='0';
wait for 100 ns;

data_in<="00000001";
clock<= '1';
clear<='0';
load<='0';
wait for 100 ns;

data_in<="00001111";
```

```
clock<='1';
clear<='0';
load<='0';
wait for 100 ns;
end process;
end testbench;

--half
library ieee;
use ieee.std_logic_1164.all;
entity test_halfadder is
end test_halfadder;
architecture testbench of test_halfadder is
signal ha,hb: std_logic;
signal hsum,hcarry: std_logic;
component half_adder1
port(A,B:in std_logic;
sum,carry: out std_logic);
end component;
begin
e1:half_adder1 port map (ha,hb,hsum,hcarry);
process
begin
ha<= '0';
hb<='0';
wait for 100 ns;
ha<= '0';
hb<='1';
```

```
wait for 100 ns;
ha<= '1';
hb<='0';
wait for 100 ns;
ha<= '1';
hb<='1';
wait for 100 ns;
end process;
end testbench;

--Final Code Test_Bench:
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity testbench is
end testbench;

architecture behavior of testbench is
component main3
port(A,B : in std_logic_vector(7 downto 0);
clock: in std_logic;
load: in std_logic;
clear: in std_logic;
Z: out std_logic_vector (23 downto 0);
end_flag: out std_logic);
end component;
signal clock1,clear1,load1: std_logic:= '0';
```

```
signal A1,B1: std_logic_vector (7 downto 0);
signal Z1: std_logic_vector (23 downto 0);
begin

tes : main3 port map(
A =>A1,
B =>B1,
clock=>clock1,
clear=>clear1,
load=>load1,
Z=>Z1);
clock1 <= not(clock1) after 500ns;
load1 <= '0' after 200ns, '1' after 300ns;
A1<="00000010";
B1<="00010000";
clear1 <= '1' after 100ns, '0' after 200ns;
end;
```

Appendix III – Test Benches :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity proj_final_tb is
  generic(width : integer := 8);
end proj_final_tb;
```

```
architecture testbench of proj_final_tb is


component proj_toplevel
generic(width : integer := 8);
port (
  a     : in std_logic_vector(width - 1 downto 0);
  b     : in std_logic_vector(width - 1 downto 0);
  clk   : in std_logic;
  load  : in std_logic;
  clr   : in std_logic;
  endF  : out std_logic;
  z     : out std_logic_vector(width*2 - 1 downto 0)
  );
end component;

signal clk,endF_tb,ld_tb,clr_tb : std_logic;
signal z_tb : std_logic_vector(width*2 - 1 downto 0);
signal a_tb,b_tb : std_logic_vector(width - 1 downto 0);

  constant clk_period : time := 10 ns;
begin -- Arch
    process
    begin
      clk <= '0';
      wait for clk_period/2;
      clk <= '1';
      wait for clk_period/2;
    end process;
```

a_tb <= x"00", x"04" after 50 ns, x"09" after 100 ns, x"FF" after 160 ns;

b_tb <= x"00", x"02" after 50 ns, x"05" after 100 ns, x"FF" after 160 ns;

ld_tb <= '1', '0' after 55 ns, '1' after 65 ns, '0' after 115 ns, '1' after 125 ns, '0' after 175 ns, '1' after 185 ns;

clr_tb <= '0', '1' after 15 ns, '0' after 20 ns, '1' after 200 ns, '0' after 210 ns;

proj_toplevel_1 : proj_toplevel generic map (8) port map(a => a_tb, b => b_tb, clk => clk, load => ld_tb, clr => clr_tb, endF => endF_tb, z => z_tb);

end testbench;

Appendix III – Area and Timing report from Xilinx ISE

C1. Precision RTL Area Report -Wallace Tree Multiplier:

```
//  Precision RTL Synthesis  64-bit 2016.1.0.15 (Production Release) Wed Jun  8 09:35:56 PDT 2016
//
//  Copyright (c) Mentor Graphics Corporation, 1996-2016, All Rights Reserved.
//             Portions copyright 1991-2008 Compuware Corporation
//                       UNPUBLISHED, LICENSED SOFTWARE.
//           CONFIDENTIAL AND PROPRIETARY INFORMATION WHICH IS THE
//         PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS
//
//  Running on Linux r_rangas@poise.encs.concordia.ca #1 SMP Tue Oct 12 08:40:46 CDT 2021 3.10.0-1160.45.1.el7.x86_64 x86_64
//
//  Start time Mon Dec  6 13:31:05 2021

****************************************************************
Device Utilization for 2VP30ff896
****************************************************************
Resource                        Used    Avail   Utilization
----------------------------------------------------------------
IOs                             44      556       7.91%
Global Buffers                  1       16        6.25%
LUTs                            615     27392     2.25%
CLB Slices                      308     13696     2.25%
Dffs or Latches                 39      29060     0.13%
Block RAMs                      0       136       0.00%
Block Multipliers               0       136       0.00%
Block Multiplier Dffs           0       4896      0.00%
GT_CUSTOM                       0       8         0.00%
----------------------------------------------------------------

*********************************************

Library: work    Cell: main3    View: operation

*********************************************

  Cell          Library  References     Total Area
```

```
 BUFGP          xcv2p     1 x
 FDCE           xcv2p    39 x       1     39 Dffs or Latches
 GND            xcv2p     1 x
 IBUF           xcv2p    18 x
 LUT1           xcv2p     1 x       1      1 LUTs
 LUT2           xcv2p     1 x       1      1 LUTs
 OBUF           xcv2p    25 x
 multiplier3    work      1 x     575    575 LUTs
                                   591    591 gates
 rca22          work      1 x      39     39 gates
                                    39     39 LUTs

 Number of ports :                          44
 Number of nets :                          156
 Number of instances :                      88
 Number of references to this view :         0

Total accumulated area :
 Number of Dffs or Latches :                39
 Number of LUTs :                          615
 Number of gates :                         632
 Number of accumulated instances :         700


*******************************
 IO Register Mapping Report
*******************************
Design: work.main3.operation

+-------------+-----------+----------+----------+----------+
| Port        | Direction |   INFF   |  OUTFF   |  TRIFF   |
+-------------+-----------+----------+----------+----------+
| A(7)        | Input     |          |          |          |
+-------------+-----------+----------+----------+----------+
| A(6)        | Input     |          |          |          |
```

A(5)	Input			
A(4)	Input			
A(3)	Input			
A(2)	Input			
A(1)	Input			
A(0)	Input			
B(7)	Input			
B(6)	Input			
B(5)	Input			
B(4)	Input			
B(3)	Input			
B(2)	Input			
B(1)	Input			
B(0)	Input			
clock	Input			
load	Input			
clear	Input			
Z(23)	Output			

Z(22)	Output			
Z(21)	Output			
Z(20)	Output			
Z(19)	Output			
Z(18)	Output			
Z(17)	Output			
Z(16)	Output			
Z(15)	Output			
Z(14)	Output			
Z(13)	Output			
Z(12)	Output			
Z(11)	Output			
Z(10)	Output			
Z(9)	Output			
Z(8)	Output			
Z(7)	Output			
Z(6)	Output			
Z(5)	Output			

Z(4)	Output			
Z(3)	Output			
Z(2)	Output			
Z(1)	Output			
Z(0)	Output			
end_flag	Output			

Total registers mapped: 0