UNIVERSITÉ Concordia UNIVERSITY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

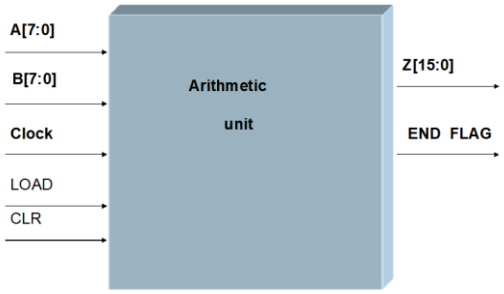
DIGITAL SYNTHESIS AND DESIGN

PROJECT COEN 6501

Term Fall 2020

DESIGN OF ARITHMETIC UNIT FOR CALCULATING

Z=1/4[A*B] +1



Submitted To

Professor Dr. OTMANE AIT MOHAMED

S. No	Student Name	Student Id
1	NANDINI PANNER SELVAM	40137073
2	RAMA KRISHNA DESINENI	40110168
3	RANJITHA ARUMUGAM	40150427
4	VARSHA SURESH	40121575

CONTENTS

ABSTRACT

In digital world, to design any system in large scale integrated circuits the parameters such as speed, area and power are main factors are considered. Researchers are trying to find a design system in order to obtain goal optimization to bring out the best performance parameters such as to operate at high speed, which can occupy minimum area, also high speed with less delay. In this project, we have included Arithmetic Logic Unit(ALU) which is an essential block of digital system performing the operation Z = ¼ [A * B] + 1. A Wallace multiplier, barrel shifter, full adder and half adder are involved in ALU. Full Adder and Half adder are used because Adders ate the core element and basic building block the design system. A and B are two 8-bit unsigned numbers and Z is 16-bit number. The two numbers are multiplied and are left shifted by two bits using shifter then 1 is added to the output. Multiplexers are used in full adder as it consumes less area. The project is simulated in Modelsim and implemented using structural modelling in VHDL. RTL Schematics are derived from RTL synthesis. XILINX ISE is used to get area and timing report.

ACKNOLWEDGEMENT

We express our sincere thanks to our Prof. Otmane Ait Mohamed for his guidance and encouragement to complete the project successfully. We are really grateful because this course has given us an opportunity to learn and design the digital devices with VHDL in depth. We also take this opportunity to thank our teaching assistant Saif Najim Eddin for his lecture and to make us hands on tool. Last but not the least we would like to thank our Concordia University for providing huge support with labs available online even in this pandemic situation.

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

The main aim of the project is to design an arithmetic logic unit which perform operation of Z = ¼ [A * B] + 1. The arithmetic unit consists of Wallace multiplier where two operands which are 8-bit are multiplied and the output is 16- bit number is passed to the barrel shifter. The barrel shifter shifts the output by two bits and given to the adders. The transition of LOAD signal from HIGH TO LOW the values of A and B are latched to register RA and RB. The CLEAR signal is used to reset registers to zero. The final output is obtained at the port Z which is 16-bit register. The LOAD signal starts calculation and ends with END_FLAG signal.

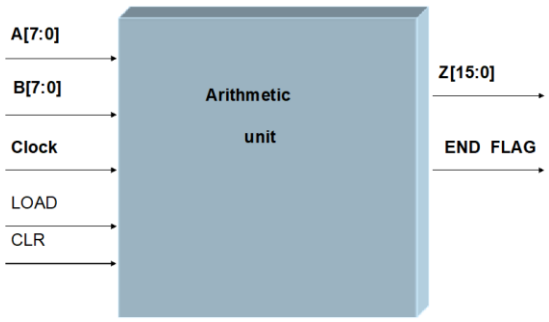


Figure 1. Arithmetic Unit

The Wallace Tree Multiplier is implemented in this design to have less delay and suitable for more for the more than 16-bit operands. This multiplier can be used in array multiplier and booth multiplier. In Wallace multiplier implementation, two operands are multiplied to reduce the partial product into two row matrix with help of adders such as full adder, half adder and carry save adder. The each bit of every column of the partial products are added together in parallel where no carry is propagated. Then again matrix is reduced by next set to obtain two row matrix model.

2. DESIGN OF FUNDAMENTAL GATES

2.1 OR GATE

OR gate is a two-input gate which performs disjunction of inputs. The output is high when one or more inputs is high. The OR gate implementation in VHDL is " output <= input_1 or input_2". The truth table is shown below. Test bench is implement and RTL schematic design is synthesized which is show below.

Input_1	Input_B	Output
0	0	0
0	1	1
1	0	1
1	1	1

Table 1: OR Gate truth table

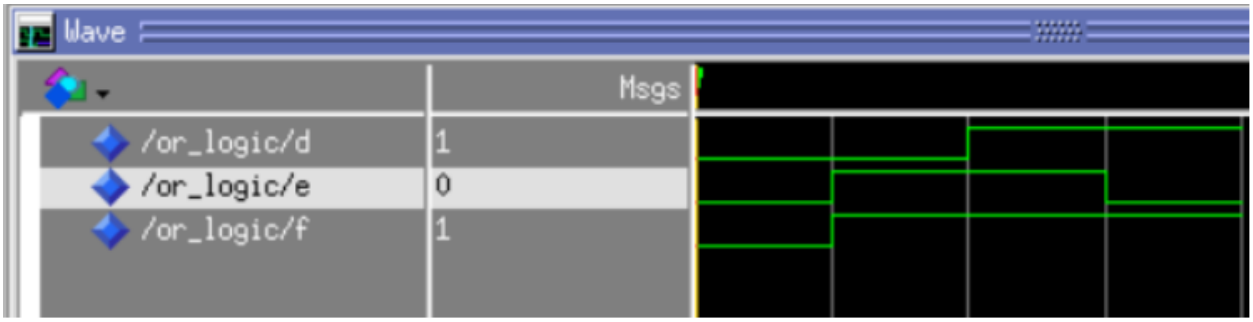


Figure 1: Simulation Result of OR gate

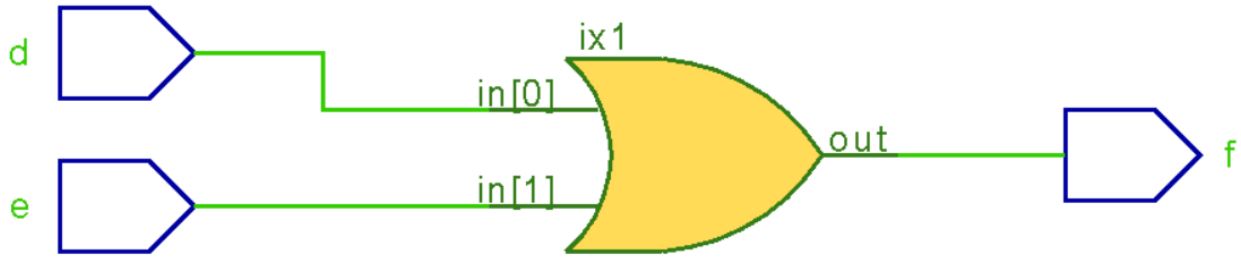


Figure 2: RTL Schematic Diagram for OR gate

2.2 AND GATE

AND gate is two-input gate which performs conjunction of inputs. The output is high when both the inputs are high. The AND gate implementation in VHDL is "output <= input_1 AND input_2". The truth table is shown below. Test bench is implement and RTL schematic design is synthesized which is show below.

Input_1	Input_2	Output
0	0	0
0	1	0
1	0	0
1	1	1

Table 2: Truth Table of AND gate

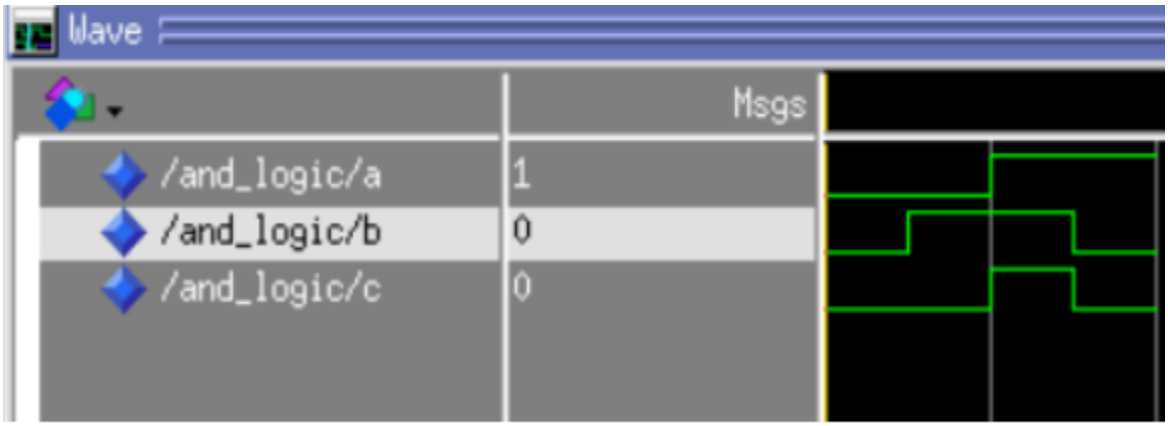


Figure 3: Simulation result of AND Gate

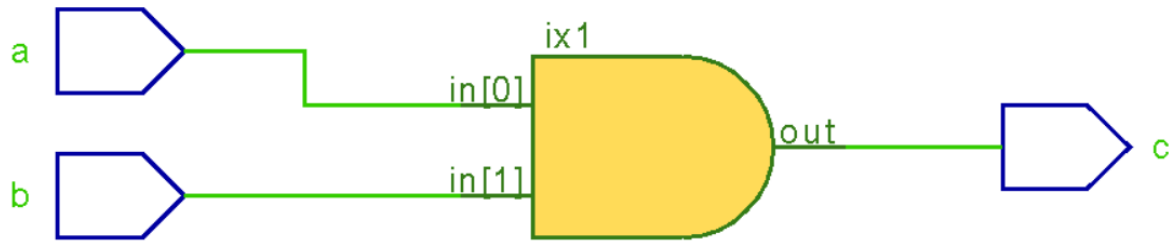


Figure 4: RTL Schematic Diagram of AND gate

2.3 NOT GATE

NOT gate is the simplest gate which performs negation of the input and it is called Inverter. It has single input and single output. It is implemented in VHDL "output <= **not** input". The truth table is shown below. Test bench is implement and RTL schematic design is synthesized which is show below.

Input	Output
0	1
1	0

Table 3: Truth table of NOT gate

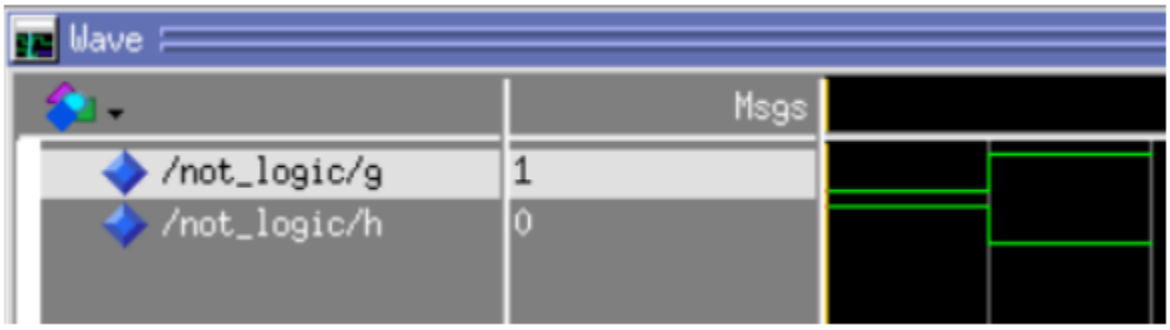


Figure 5: Simulation result of NOT gate

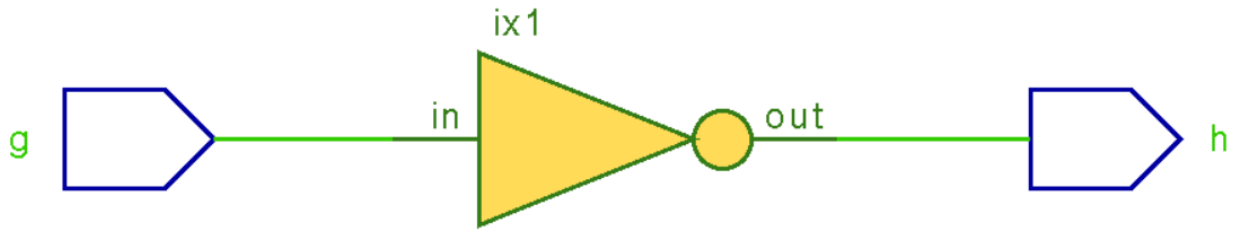


Figure 6: RTL Schematic Diagram of NOT gate

2.4 NAND GATE WITH TWO INPUTS

The two-input NAND gate is the compliment AND gate. The AND gate output is inverted using NOT gate. The NAND gate is used in D FF. The truth table of NAND gate is shown.

Input_1	Input_2	Output
0	0	1
0	1	1
1	0	1
1	1	0

Table 4: NAND gate Truth table

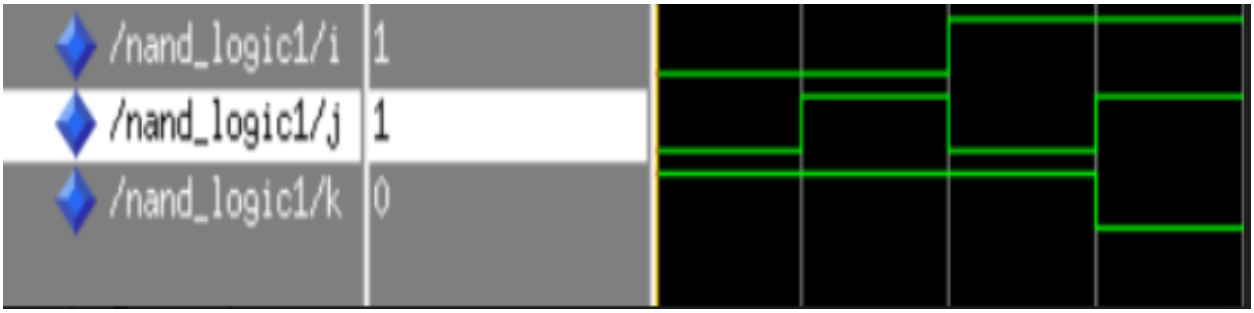


Figure 7: Simulation result of NAND gate with two inputs

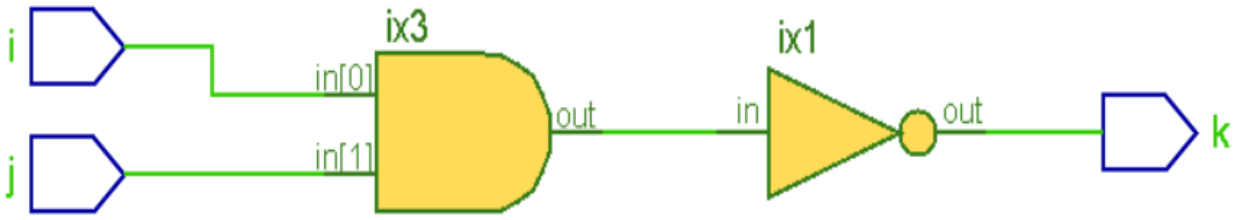


Figure 8: RTL Schematic Diagram of NAND gate with two inputs

2.5 NAND GATE WITH 3 INPUTS

The three-inputs NAND gate is implement in DFF and truth table is shown below.

Input_1	Input_2	Input_3	Output
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Table 5: 3 Input NAND Gate Truth Table

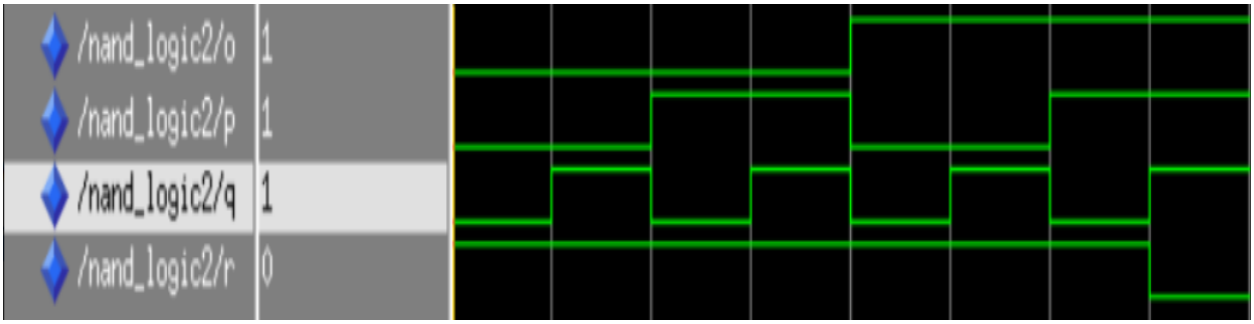


Figure 9: Simulation result of NAND gate with three inputs

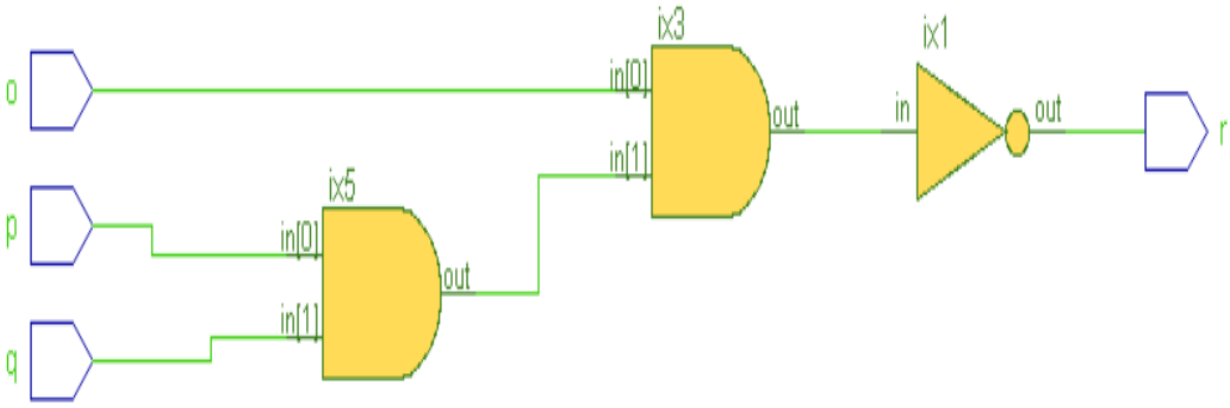


Figure 8: RTL Schematic Diagram of NAND gate with two inputs

2.6 XOR GATE

The two-input XOR gate is called as exclusive OR. This digital gate gives output high when the inputs are odd. The XOR gate is implemented in VHDL as "output <= input_1 xor input_2". The truth table is shown below. Test bench is implement and RTL schematic design is synthesized which is show below.

Input_1	Input_2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Table 6: XOR Gate Truth Table

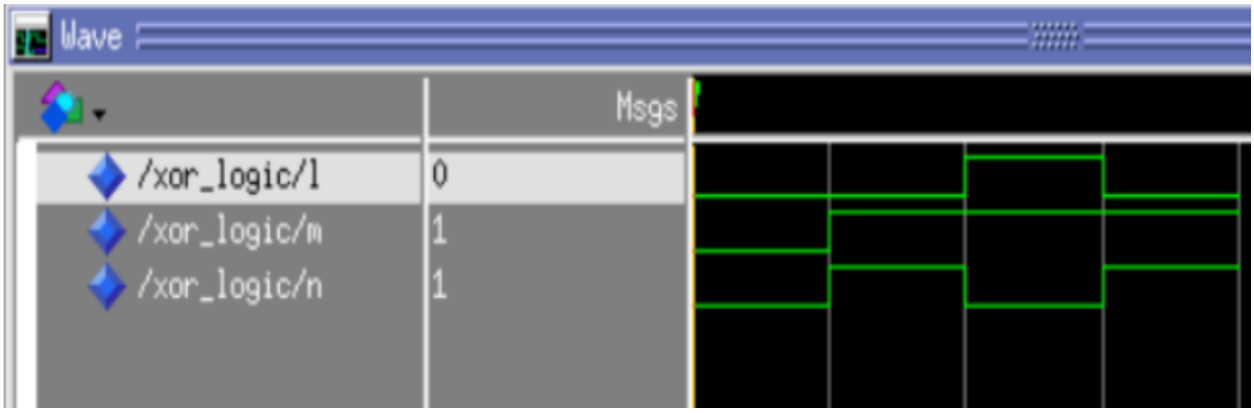


Figure 11: Simulation result of XOR gate

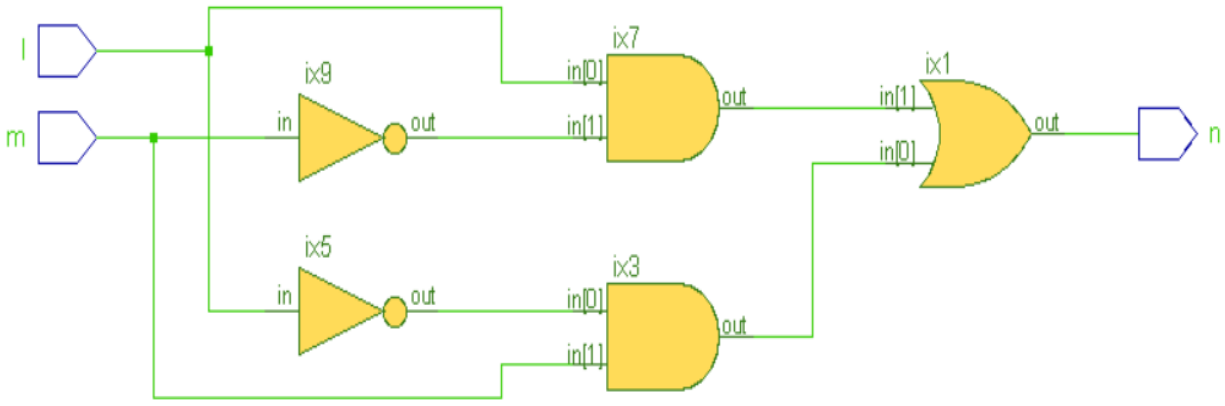


Figure 12: RTL Schematic Diagram of XOR gate

3. FUNCTIONAL BLOCKS

The basic gates which have been discussed above can be deduced from functional blocks. In a structural way, these units are built. Such functional blocks allow the addition of bits to be stored and executed.

3.1 HALF ADDER

Half Adder is the basic and simplest functional block in digital design. Half adder perform the addition of two numbers. The outputs are sum and carry. Sum is the logical XOR operation and carry is logical AND operation. The truth table is shown below.

SUM <= H_A □ H_B;

CARRY <= H_A + H_B ;

H_A	H_B	H_SUM	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 7: Half Adder truth table

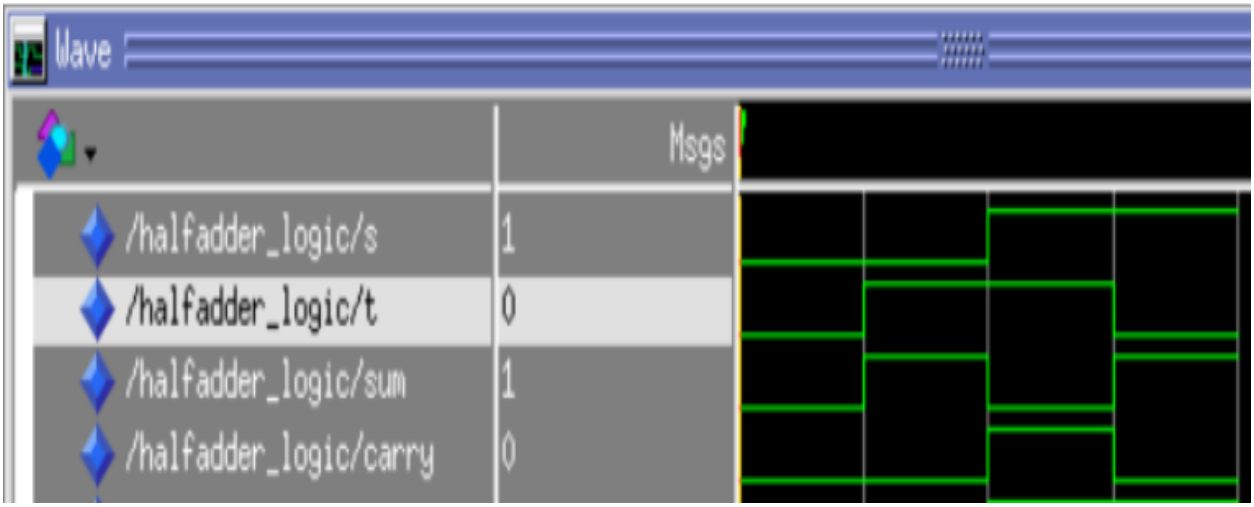


Figure 13: Simulation results of Half Adder

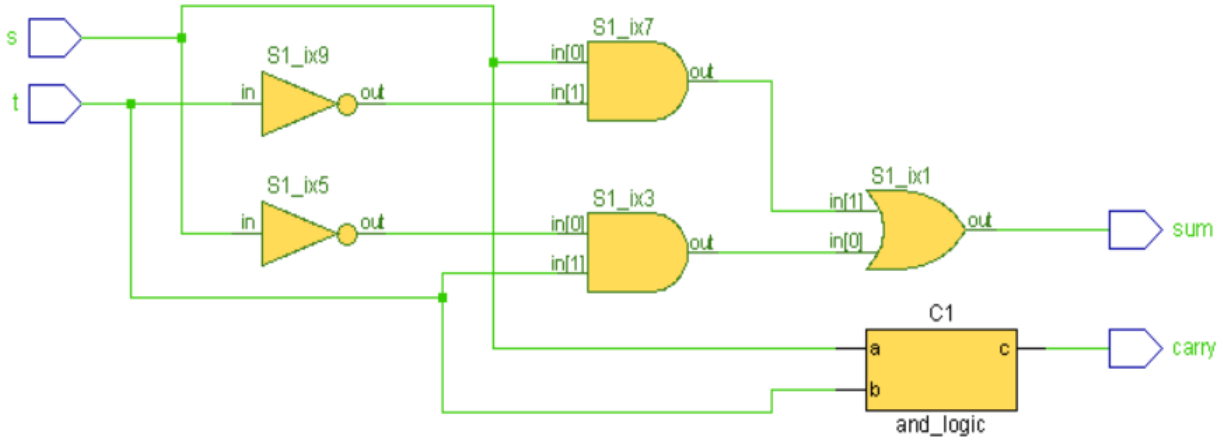


Figure 14: RTL Schematic Diagram of Half adder

3.2. FULL ADDER

Full adder is a logical circuit in which three one-bit binary numbers are added and result a sum and carry value. Full adder is used in many applications and the high speed or low power output can be obtained by combining with other full adders. But cost increases due to increase in the number of gates.

F_A	F_B	F_Cin	F_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

Table 8: Full Adder truth table

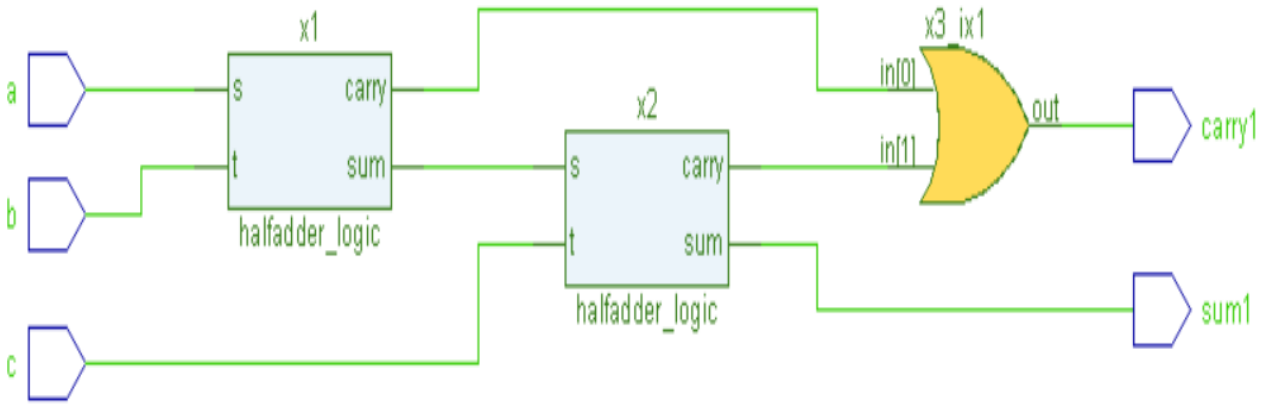


Figure 15: RTL Schematic diagram of Full adder

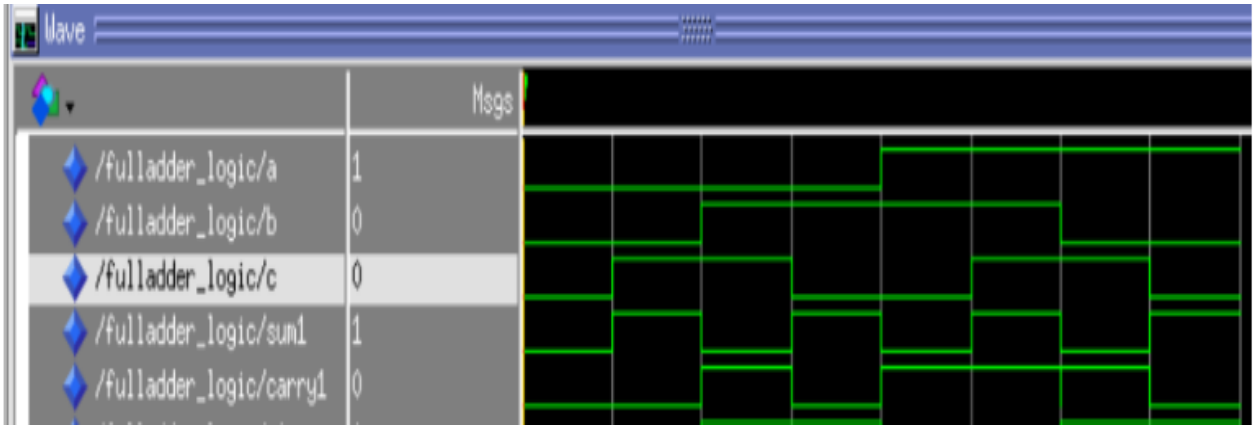


Figure 16: Simulation results of Full Adder

3.3. MULTIPLEXER

Multiplexer transmit large information over a small channel. Multiplexer is also referred as MUX which select single output from N number of inputs. In this project, we used 2:1 Multiplexer. 2:1 refers to two inputs and one output, i.e. select single output among the two inputs.

M_A	M_B	M_Sel	M_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

Table 9: 2:1 Multiplexer truth table

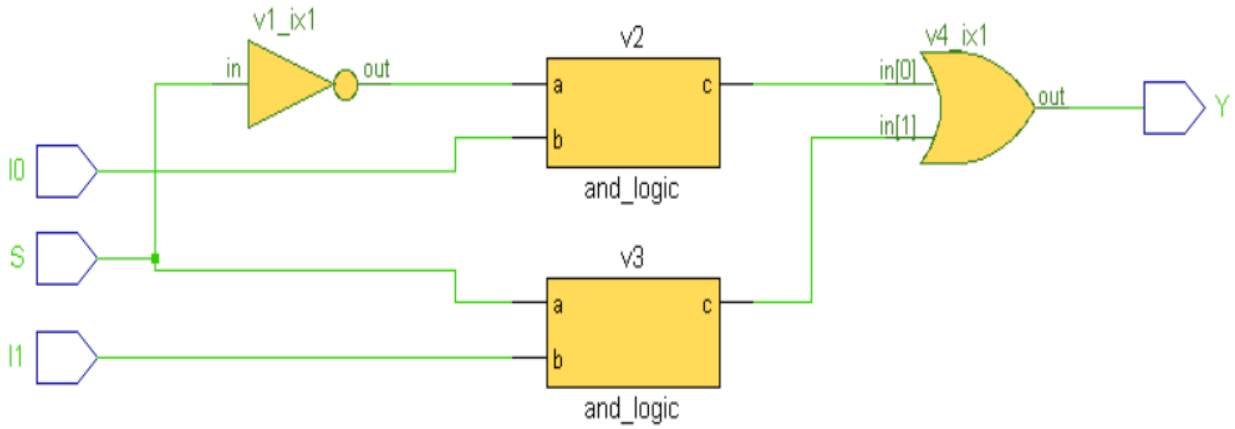


Figure 17: RTL Schematic diagram of 2:1 Multiplexer

Figure 18: Simulation results of 2:1 Multiplexer

3.4. D FLIP-FLOP

D flip flop is used for data processor and memory storage. Here, it is build using NAND gate. The main application of D flip flop is to produce delay as a buffer in timing circuit for sampling data at specific intervals. Registers of n size is built with n number of D flip flops storing each bit in the word or block of data.

D_in	Clk	D_Q	D_Q'
x	low	0	1
0	high	0	1
1	high	1	0

Table 10: D Flip flop truth table

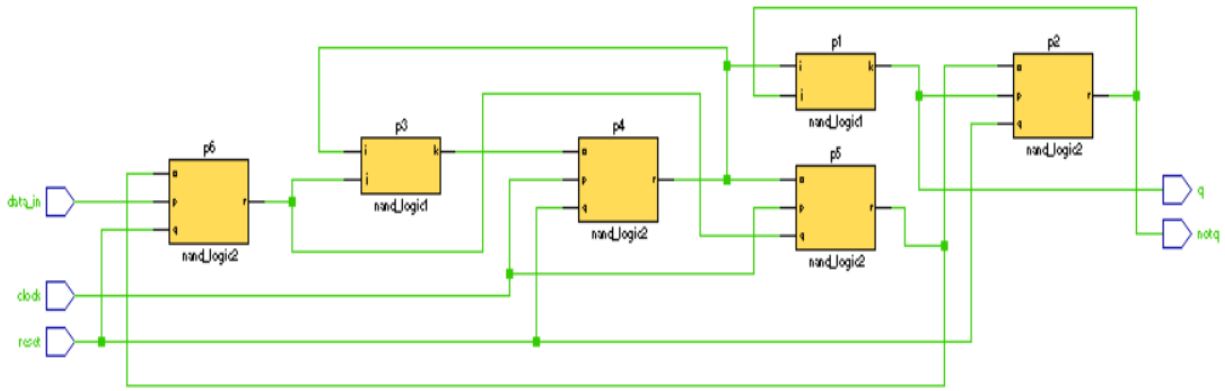


Figure 19: RTL schematic diagram of D Flip flop

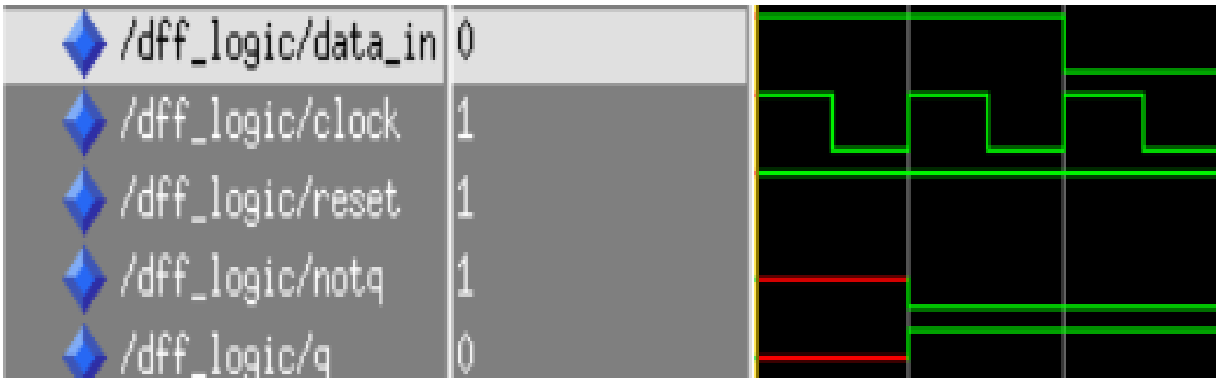


Figure 20: Simulation results of D Flip Flop

4. Design Of Higher-Level Functional Blocks

The higher level functional blocks such as Registers, 8-bit registers, 16-bit registers, Incrementor, Barrel shifter, Wallace multiplier is discussed in this section.

REGISTERS

Registers are central processing unit part and are measured with the number of bits they hold, example: 8-bit register, 16-bit register, 32-bit register, 64-bit register etc. It can also operates by breaking down the storage memory into smaller units such as 32-bit into four 8-bit and operated at same time. In this project, 8-bit register and 16-bit registers are used.

4.1. 8-bit Register

8-bit register has 8 inputs and 8 outputs and D flip flop is attached inside the register.

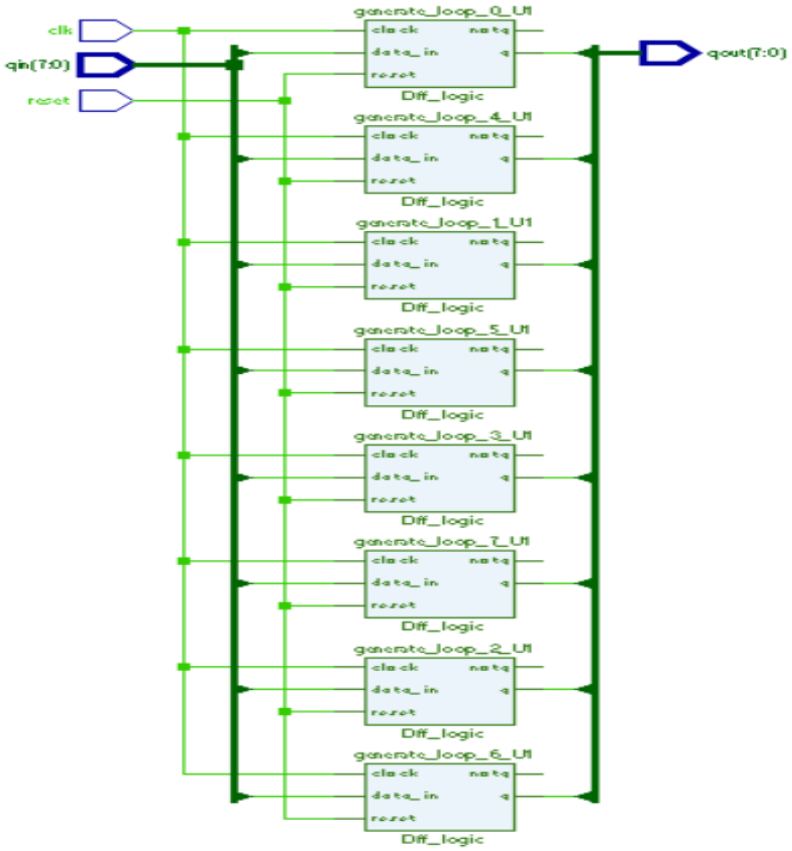


Figure 21. RTL Schematic diagram of 8-bit register

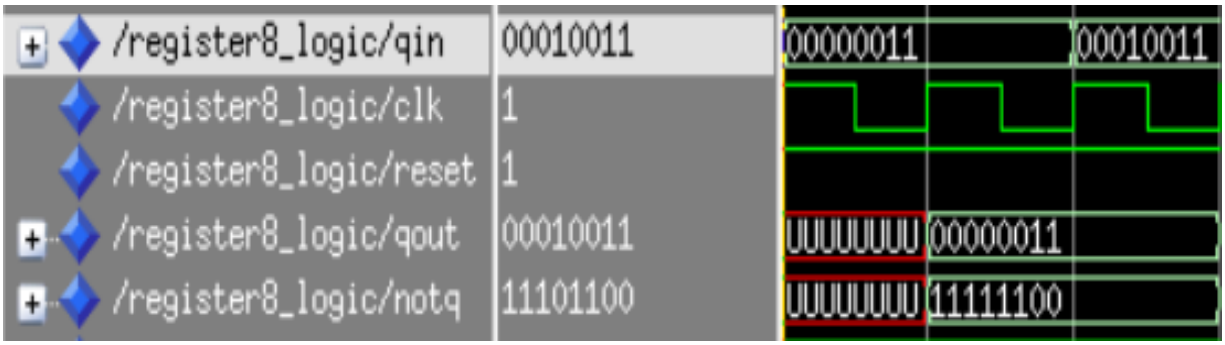


Figure 22. Simulation results of 8-bit register

4.2. 16-bit Register

16-bit register stores the array multiplier result. The array multiplier multiplies two 8-bit operands and results 16-bits to the register.

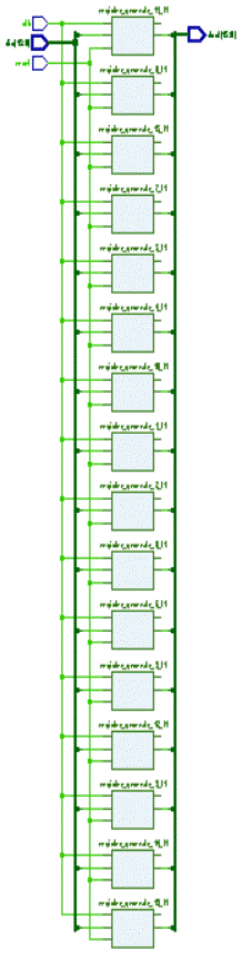


Figure 23. RTL schematic diagram of 16-bit register

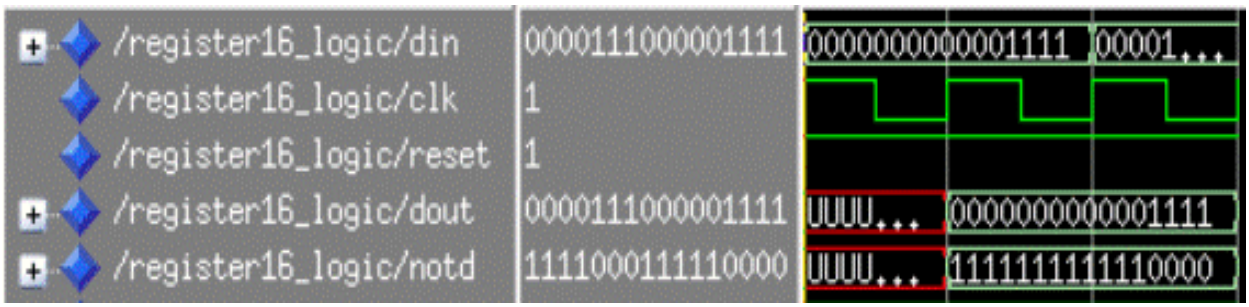


Figure 24. Simulation results of 16-bit register

4.3. INCREMENTOR

Incrementor add one to the input value. It is built with series of half adder and with carry in and carry out. The RTL schematic diagram and simulation results are in the following.

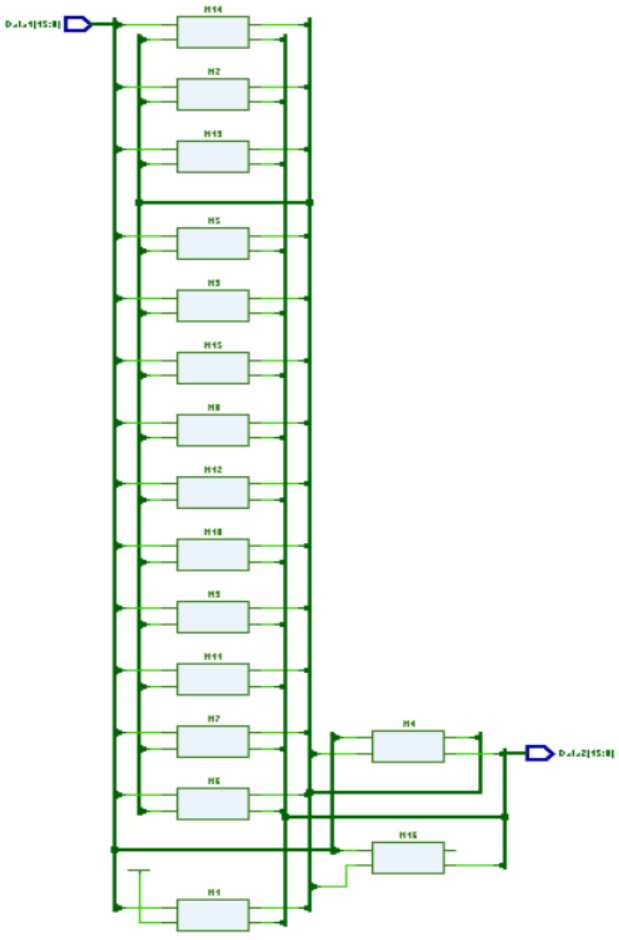


Figure 25. RTL Schematic diagram of Incrementor

Figure 26. Simulation results of Incrementor

4.4. BARREL SHIFTER

A barrel shifter shifts the operands with the control signal value. It is a combinational circuit where shifting to the left defines multiplication by two and shifting to the right defines to division by two. By considering binary values, shifting mechanism works perfectly. For decimal values, this works fine only for the multiples of 4 values. For a value not multiple of 4, shifting results only the decimal part not the fractional part.

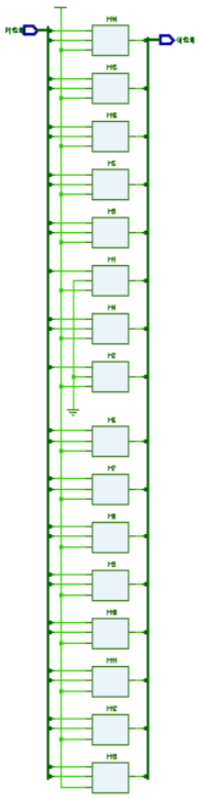


Figure 27. RTL Schematic diagram of Barrel Shifter

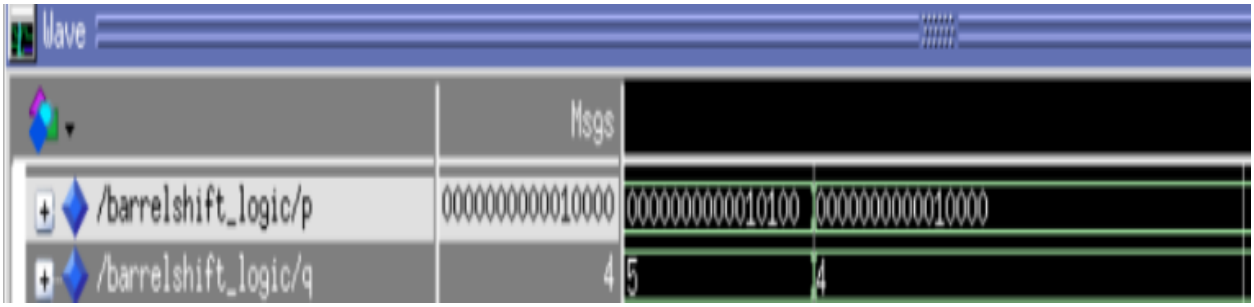


Figure 28. Simulation results of barrel shifter

4.5. WALLACE MULTIPLIER UNIT

Basically, the multiplier unit performs add and shift algorithm. There are many multiplier methods, and we chose Wallace multiplier in this project as this has less delay. Also, this is more suitable for the operands more than 16 bits in length which increase the speed since the addition of partial product is less. Wallace multiplier is used in conjunction with array multiplier and booth multiplier.

In this design, number of partial products produced during multiplication process is decreased and less speed is achieved with reduce number of sequential adding stages. In this architecture, multiplication of two operands are performed with half adder, full adder and carry save adder which reduce partial product matrix into two-row matrix and are these rows are added using fast carry propagate adder.

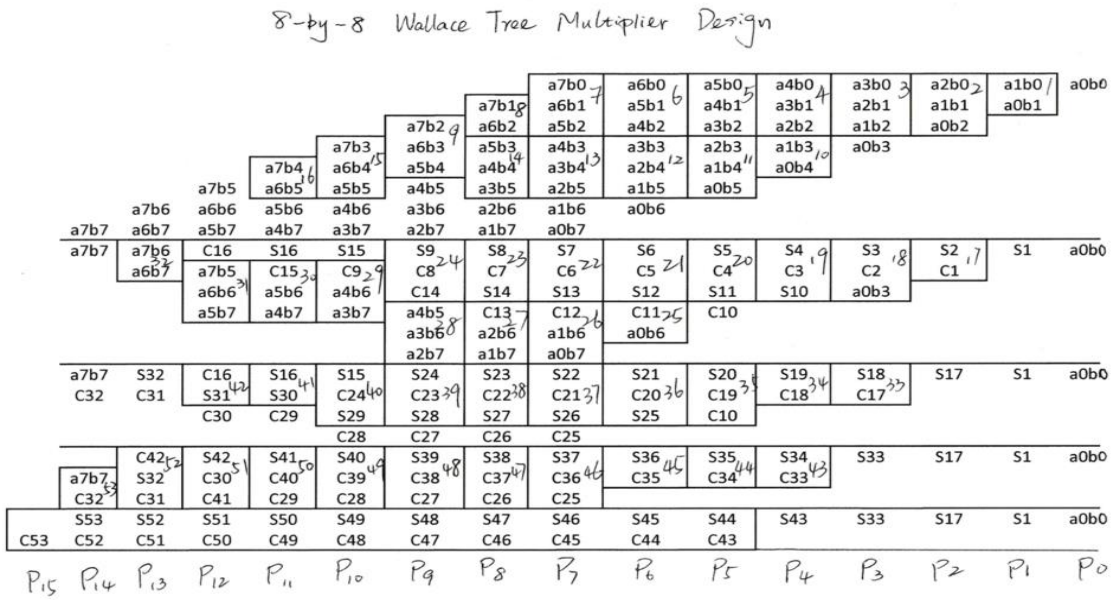


Figure 29 8-Bit Wallace Tree Multiplier

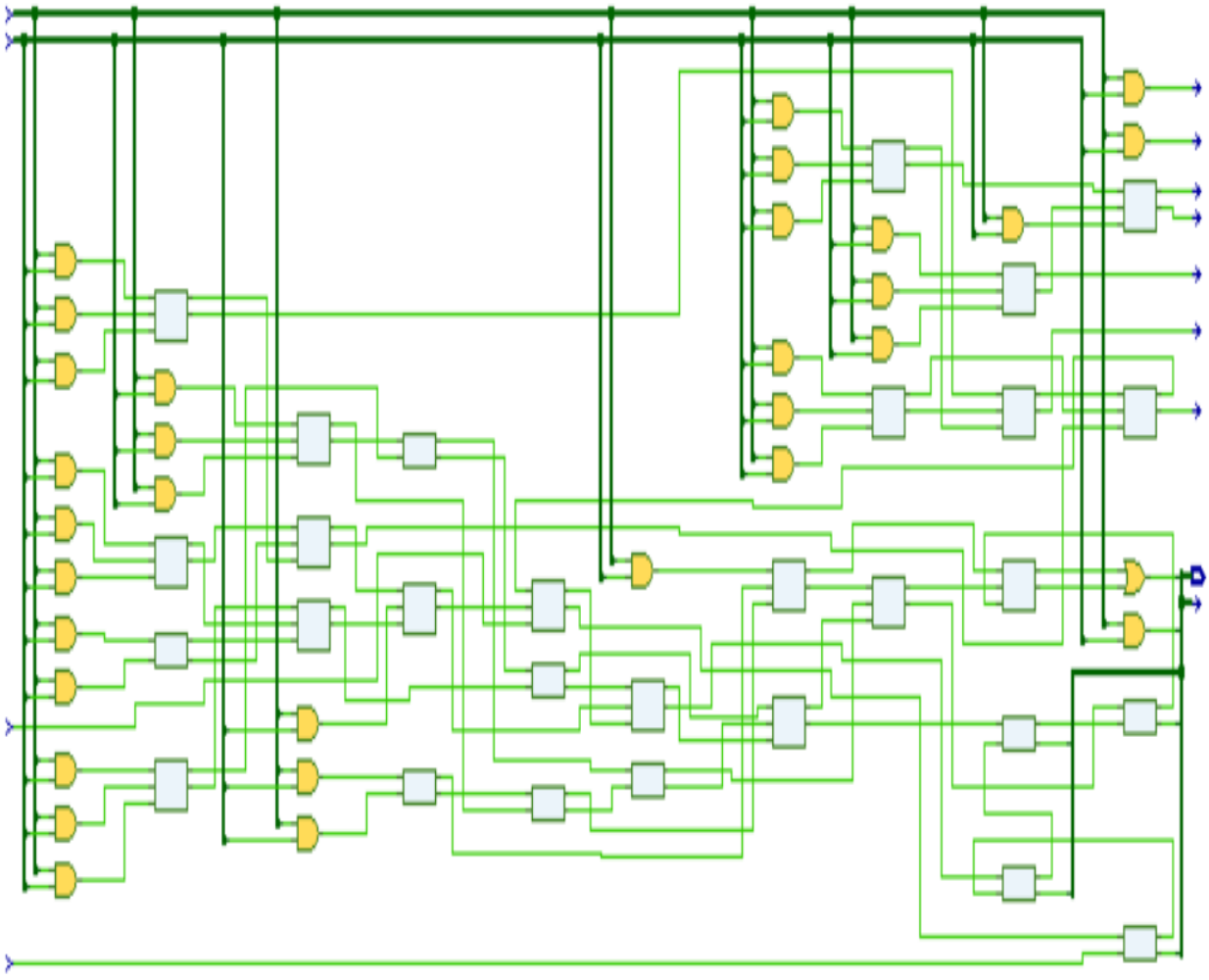


Figure 30. RTL Schematic diagram of multiplier

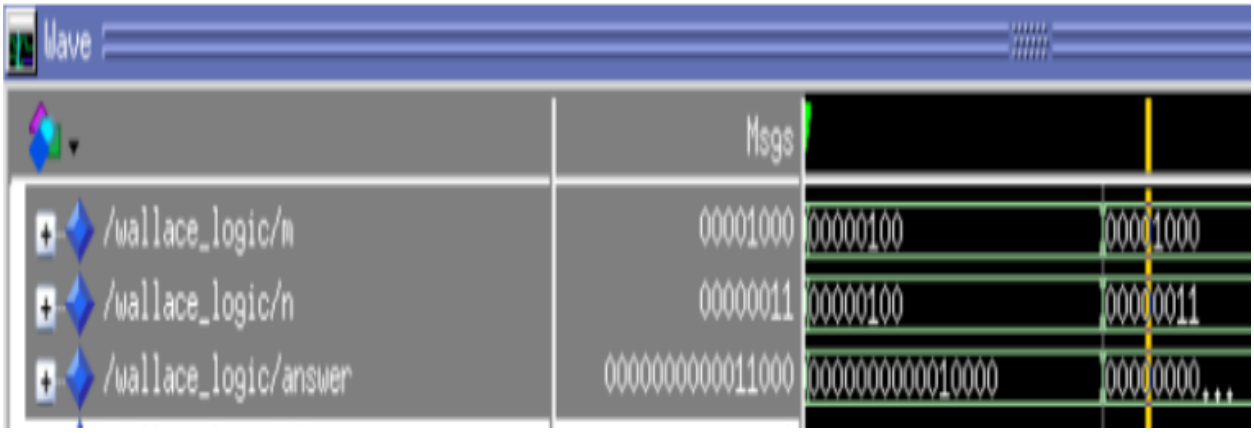


Figure 31. Simulation results of multiplier

5. IMPLEMENTATION OF LOGIC UNIT

This section computes the ¼[A*B]+1 using 8-bit register, Wallace multiplier, Barrel shifter and incrementor. The corresponding implementation and simulation are explained in the previous sections. Here, two inputs A and B are applied to Arithmetic logic unit and it passes as in the following block diagram and 16-bit output is available at Z port. The output of 8-bit register is given to the multiplier for multiplication and its output is fed to Barrel shifter. This divide the value by four and at the end incrementor increments the value by one.

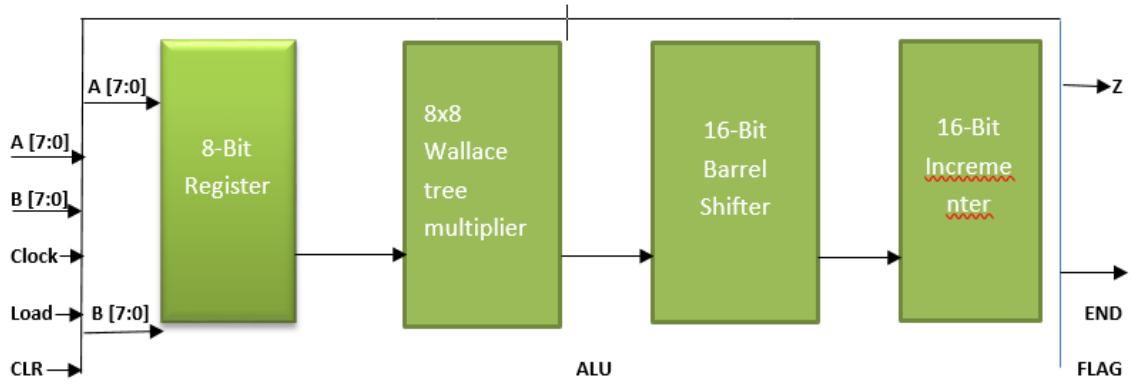


Figure 32. Block diagram of the arithmetic logic unit

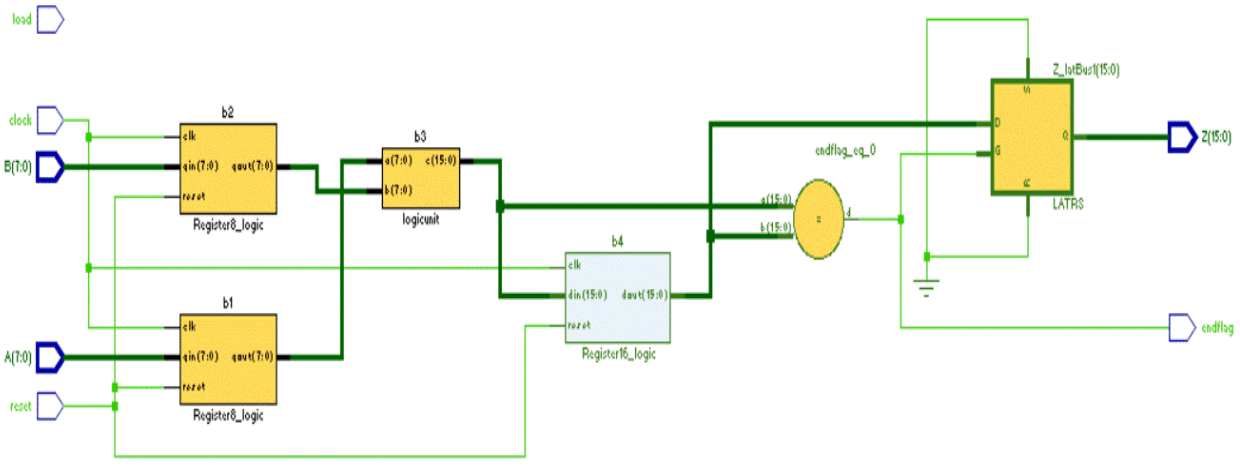


Figure 33. RTL Schematic Diagram of Arithmetic Logical Unit

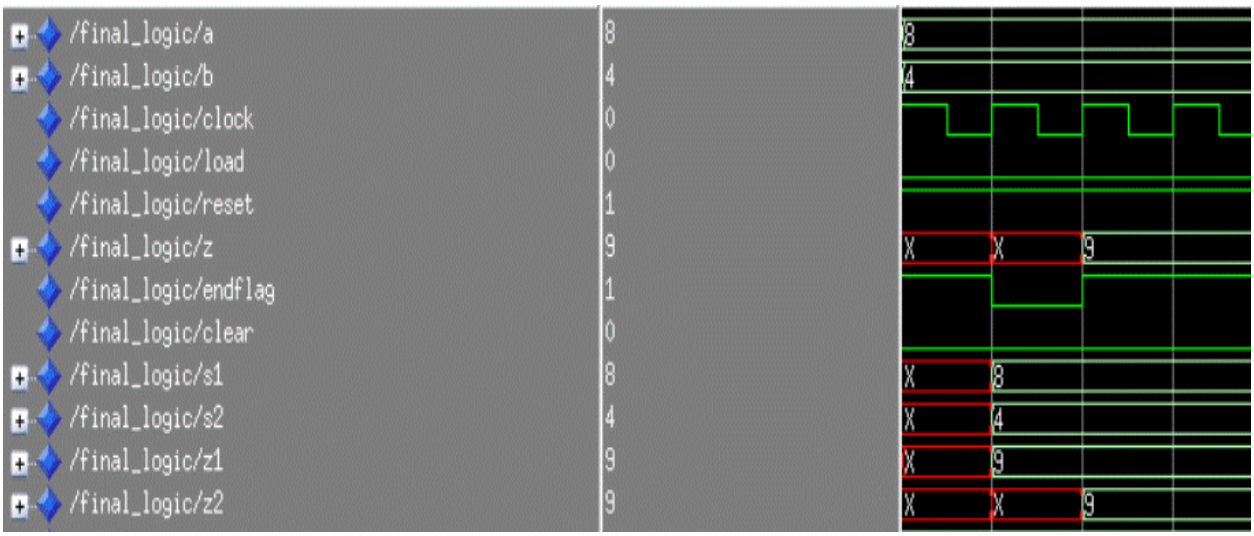


Figure 34 Simulation results of Arithmetic Unit

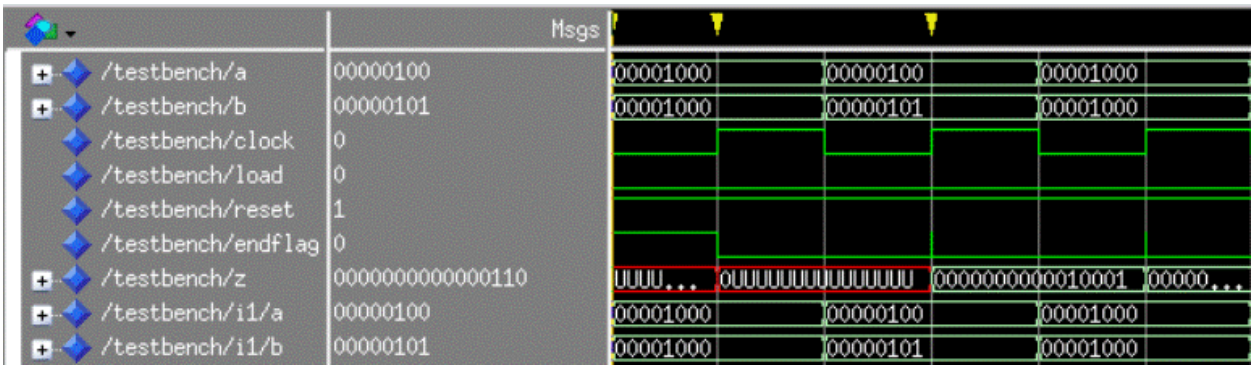


Figure 35 Testbench results of Arithmetic Unit

6. FUTURE WORK

- The synthesis design can be implemented on FPGA.
- The design and synthesis can also be executed using 16-bit operands.
- Instead of Wallace multiplier we can use Array or Booth multiplier. Also carry select adder and ripple adder can be used.
- Implementation of pipelining helps is performance improvement.

7. CONCLUSION

We have developed the Arithmetic Unit in this project and have mainly concentrated on area, power and delay reduction, and provided VHDL codes and RTL synthesis diagrams including Area and Timing Report. Using different multipliers and adders, more area and power can be reduced.

8. REFERENCES

- Lecture notes
- VHDL: Programming by Example: Edition 4, Douglas L. PerryJun. 2002.
- www.google.com
- www.youtube.com
- https://www.tutorialspoint.com/vlsi_design/vlsi_design_vhdl_introduction.htm

STRUCTURAL VHDL SOURCE CODE

AND Gate:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity and_logic is
port (a,b : in std_logic;
      c : out std_logic);
end and_logic;
architecture struct_and of and_logic is
begin
c <= a and b;
end struct_and;
```

OR Gate:

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

```
entity or_logic is
port (d,e : in std_logic;
      f: out std_logic);
end or_logic;
architecture struct_or of or_logic is
begin
f <= d or e;
end struct_or;
```

NOT Gate:

```
library ieee;
use ieee.std_logic_1164.all;
entity not_logic is
port
(g : in std_logic;
h : out std_logic);
end not_logic;
architecture struct_not of not_logic is
begin
  h <= not g;
end struct_not;
```

XOR Gate:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity xor_logic is
port (l,m : in std_logic;
n : out std_logic);
end xor_logic;
architecture struct_xor of xor_logic is
begin
n <= (not (l) and m) or (l and not(m) );
end struct_xor;
```

Two input NAND Gate:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity nand_logic1 is
```

```
port (i : in std_logic ;
 j : in std_logic ;
 k : out std_logic);
end nand_logic1;
architecture struct_nand of nand_logic1 is
begin
 k <= i NAND j;
end struct_nand;
```

Three input NAND Gate:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity nand_logic2 is
port (o, p, q : in std_logic ;
  r: out std_logic);
end nand_logic2;
architecture struct_nand1 of nand_logic2 is
begin
 r <= not(o and (p and q));
```

```
end struct_nand1;
```

Two-to-One Multiplexer:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity mux2to1_logic is
    port ( I0: in std_logic;
           I1 : in std_logic;
           S : in std_logic;
           Y : out std_logic);
end mux2to1_logic;
architecture struct_mux of mux2to1_logic is
component and_logic
port (a,b : in std_logic;
      c : out std_logic);
end component;
component not_logic
port
(g : in std_logic;
h : out std_logic);
```

```
end component;
component or_logic
port (d,e : in std_logic;
      f: out std_logic);
end component;
signal Sb,y1,y2 :std_logic;
begin
v1:not_logic port map(S,Sb);
v2:and_logic port map(Sb,I0,y1);
v3:and_logic port map(S,I1,y2);
v4:or_logic port map(y1,y2,Y);
end struct_mux;
```

Half Adder:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity halfadder_logic is
port (s,t : in std_logic;
sum, carry : out std_logic);
```

```
end halfadder_logic;
architecture struct_halfadder of halfadder_logic is
component xor_logic
port (l,m: in std_logic; n: out std_logic);
end component;
component and_logic
port (a,b: in std_logic; c: out std_logic);
end component;begin
S1: xor_logic port map (s, t, sum);
C1: and_logic  port map (s, t, carry);
  end struct_halfadder;
```

Full Adder:

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

entity fulladder_logic is
port (a,b,c: in std_logic;
      sum1, carry1 : out std_logic);
```

```
end fulladder_logic;

architecture struct_fulladder of fulladder_logic is

component halfadder_logic
port (s, t : in std_logic; sum,carry : out std_logic);
end component;
component or_logic
port (d,e: in std_logic; f: out std_logic);
end component;
signal s1, c1, c2: std_logic;
begin
x1: halfadder_logic port map (a, b, s1, c1);
x2: halfadder_logic port map (s1, c, sum1, c2);
x3: or_logic port map (c1, c2, carry1);
end struct_fulladder;
```

Barrel Shifter:

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

```
entity barrelshift_logic is
port
( P: in std_Logic_vector (15 downto 0);
 Q : out std_logic_vector (15 downto 0)
);
end barrelshift_logic;
architecture struct_barrel of barrelshift_logic is
  component mux2to1_logic
port ( I0: in std_logic;
          I1 : in std_logic;
          S : in std_logic;
          Y : out std_logic);
end component;
begin
M1:  mux2to1_logic port map (P(15), '0', '1', Q(15));
M2:  mux2to1_logic port map (P(14), '0', '1', Q(14));
M3:  mux2to1_logic port map (P(13), P(15), '1', Q(13));
M4:  mux2to1_logic port map (P(12), P(14), '1', Q(12));
M5:  mux2to1_logic port map (P(11), P(13), '1', Q(11));
M6:  mux2to1_logic port map (P(10), P(12), '1', Q(10));
M7:  mux2to1_logic port map (P(9), P(11), '1', Q(9));
```

```
M8:  mux2to1_logic port map (P(8), P(10), '1', Q(8));
M9:  mux2to1_logic port map (P(7), P(9), '1', Q(7));
M10: mux2to1_logic port map (P(6), P(8), '1', Q(6));
M11: mux2to1_logic port map (P(5), P(7), '1', Q(5));
M12: mux2to1_logic port map (P(4), P(6), '1', Q(4));
M13: mux2to1_logic port map (P(3), P(5), '1', Q(3));
M14: mux2to1_logic port map (P(2), P(4), '1', Q(2));
M15: mux2to1_logic port map (P(1), P(3), '1', Q(1));
M16: mux2to1_logic port map (P(0), P(2), '1', Q(0));
  end struct_barrel;
```

Incrementer:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity Incrementer_logic is
port
(
Data1: in std_logic_vector (15 downto 0);
Data2: out std_logic_vector (15 downto 0)
);
```

```
end Incrementer_logic;
architecture struct_incrementer of Incrementer_logic is
component halfadder_logic is
port (s,t : in std_logic;
sum, carry : out std_logic);
end component;
signal c: std_logic_vector (15 downto 0);
begin
N1: halfadder_logic port map (Data1(0), '1', Data2(0), c(0));
N2: halfadder_logic port map (Data1(1), c(0), Data2(1), c(1));
N3: halfadder_logic port map (Data1(2), c(1), Data2(2), c(2));
N4: halfadder_logic port map (Data1(3), c(2), Data2(3), c(3));
N5: halfadder_logic port map (Data1(4), c(3), Data2(4), c(4));
N6: halfadder_logic port map (Data1(5), c(4), Data2(5), c(5));
N7: halfadder_logic port map (Data1(6), c(5), Data2(6), c(6));
N8: halfadder_logic port map (Data1(7), c(6), Data2(7), c(7));
N9: halfadder_logic port map (Data1(8), c(7), Data2(8), c(8));
N10: halfadder_logic port map (Data1(9), c(8), Data2(9), c(9));
N11: halfadder_logic port map (Data1(10), c(9), Data2(10), c(10));
N12: halfadder_logic port map (Data1(11), c(10), Data2(11), c(11));
N13: halfadder_logic port map (Data1(12), c(11), Data2(12), c(12));
```

```
N14: halfadder_logic port map (Data1(13), c(12), Data2(13), c(13));
N15: halfadder_logic port map (Data1(14), c(13), Data2(14), c(14));
N16: halfadder_logic port map (Data1(15), c(14), Data2(15), c(15));
end struct_incrementer;
```

Wallace Multiplier:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity Wallace_logic is
  Port (M : in  std_logic_vector(7 downto 0);
        N : in  std_logic_vector (7 downto 0);
        answer : out std_logic_vector (15 downto 0));
end Wallace_logic;
architecture struct_wallace of Wallace_logic is
component fulladder_logic is
    port (a,b,c: in std_logic;
      sum1, carry1 : out std_logic);
end component;
component halfadder_logic is
```

```
    port (s,t : in std_logic;

sum, carry : out std_logic);

end component;

signal
x00,x01,x02,x03,x04,x05,x06,x07,x10,x11,x12,x13,x14,x15,x16,x17,x20,x21,x22,
x23,x24,x25,x26,x27,x30,x31,x32,x33,x34,x35,x36,x37,x40,x41,x42,x43,x44,x45,
x46,x47,x50,x51,x52,x53,x54,x55,x56,x57,x60,x61,x62,x63,x64,x65,x66,x67,x70,
x71,x72,x73,x74,x75,x76,x77:std_logic;

signal
y01,y02,y03,y04,y05,y06,y07,y08,y09,y10,y11,y12,y13,y14,y15,y16,y17,y18,y19,
y20,y21,y22,y23,y24,y25,y26,y27,y28,y29,y30,y31,y32,y33,y34,y35,y36,y37,y38,
y39,y40,y41,y42,y43,y44,y45,y46,y47,y48,y49,y50,y51,y52,y53,y54,y55,y56,y57,
y58,y59,y60,y61,y62,y63,y64,y65,y66,y67,y68:std_logic;

signal
z01,z02,z03,z04,z05,z06,z07,z08,z09,z10,z11,z12,z13,z14,z15,z16,z17,z18,z19,z2
0,z21,z22,z23,z24,z25,z26,z27,z28,z29,z30,z31,z32,z33,z34,z35,z36,z37,z38,z39,z
40,z41,z42,z43,z44,z45,z46,z47,z48,z49,z50,z51,z52,z53,z54,z55,z56,z57,z58,z59
,z60,z61,z62,z63,z64,z65,z66,z67,z68:std_logic;

begin

x00 <= M(0) and N(0);

x10 <= M(1) and N(0);

x20 <= M(2) and N(0);

x30 <= M(3) and N(0);

x40 <= M(4) and N(0);

x50 <= M(5) and N(0);

x60 <= M(6) and N(0);

x70 <= M(7) and N(0);
```

```
x01 <= M(0) and N(1);

x11 <= M(1) and N(1);

x21 <= M(2) and N(1);

x31 <= M(3) and N(1);

x41 <= M(4) and N(1);

x51 <= M(5) and N(1);

x61 <= M(6) and N(1);

x71 <= M(7) and N(1);

x02 <= M(0) and N(2);

x12 <= M(1) and N(2);

x22 <= M(2) and N(2);

x32 <= M(3) and N(2);

x42 <= M(4) and N(2);

x52 <= M(5) and N(2);

x62 <= M(6) and N(2);

x72 <= M(7) and N(2);

x03 <= M(0) and N(3);

x13 <= M(1) and N(3);

x23 <= M(2) and N(3);

x33 <= M(3) and N(3);

x43 <= M(4) and N(3);
```

x53 <= M(5) and N(3);

x63 <= M(6) and N(3);

x73 <= M(7) and N(3);

x04 <= M(0) and N(4);

x14 <= M(1) and N(4);

x24 <= M(2) and N(4);

x34 <= M(3) and N(4);

x44 <= M(4) and N(4);

x54 <= M(5) and N(4);

x64 <= M(6) and N(4);

x74 <= M(7) and N(4);

x05 <= M(0) and N(5);

x15 <= M(1) and N(5);

x25 <= M(2) and N(5);

x35 <= M(3) and N(5);

x45 <= M(4) and N(5);

x55 <= M(5) and N(5);

x65 <= M(6) and N(5);

x75 <= M(7) and N(5);

x06 <= M(0) and N(6);

x16 <= M(1) and N(6);

```
x26 <= M(2) and N(6);

x36 <= M(3) and N(6);

x46 <= M(4) and N(6);

x56 <= M(5) and N(6);

x66 <= M(6) and N(6);

x76 <= M(7) and N(6);

x07 <= M(0) and N(7);

x17 <= M(1) and N(7);

x27 <= M(2) and N(7);

x37 <= M(3) and N(7);

x47 <= M(4) and N(7);

x57 <= M(5) and N(7);

x67 <= M(6) and N(7);

x77 <= M(7) and N(7);

H1:  halfadder_logic port map(x01,x10,y01,z01);

F1:  fulladder_logic port map(x20,x02,x11,y02,z02);

F2:  fulladder_logic port map(x30,x21,x12,y03,z03);

F3:  fulladder_logic port map(x40,x31,x22,y04,z04);

H2:  halfadder_logic port map(x13,x04,y05,z05);

F4:  fulladder_logic port map(x50,x41,x32,y06,z06);

F5:  fulladder_logic port map(x23,x14,x05,y07,z07);
```

F6: fulladder_logic port map(x60,x51,x42,y08,z08);

F7: fulladder_logic port map(x33,x24,x15,y09,z09);

F8: fulladder_logic port map(x70,x61,x52,y10,z10);

F9: fulladder_logic port map(x43,x34,x25,y11,z11);

H3: halfadder_logic port map(x16,x07,y12,z12);

F10: fulladder_logic port map(x71,x62,x53,y13,z13);

F11: fulladder_logic port map(x44,x35,x26,y14,z14);

F12: fulladder_logic port map(x72,x63,x54,y15,z15);

F13: fulladder_logic port map(x45,x36,x27,y16,z16);

F14: fulladder_logic port map(x73,x64,x55,y17,z17);

H4: halfadder_logic port map(x46,x37,y18,z18);

F15: fulladder_logic port map(x74,x65,x56,y19,z19);

F16: fulladder_logic port map(x75,x66,x57,y20,z20);

H5: halfadder_logic port map(x76,x67,y21,z21);

H6: halfadder_logic port map(y02,z01,y22,z22);

F17: fulladder_logic port map(x03,z02,y03,y23,z23);

F18: fulladder_logic port map(y04,y05,z03,y24,z24);

F19: fulladder_logic port map(y06,y07,z04,y25,z25);

F20: fulladder_logic port map(y08,y09,x06,y26,z26);

H7: halfadder_logic port map(z06,z07,y27,z27);

F21: fulladder_logic port map(y10,y11,y12,y28,z28);

H8: halfadder_logic port map(z08,z09,y29,z29);

F22: fulladder_logic port map(y13,y14,x17,y30,z30);

F23: fulladder_logic port map(z10,z11,z12,y31,z31);

F24: fulladder_logic port map(y15,y16,z13,y32,z32);

F25: fulladder_logic port map(y17,y18,z15,y33,z33);

F26: fulladder_logic port map(y19,z17,z18,y34,z34);

H9: halfadder_logic port map(y20,z19,y35,z35);

H10: halfadder_logic port map(y21,z20,y36,z36);

H11: halfadder_logic port map(y23,z22,y37,z37);

H12: halfadder_logic port map(z23,y24,y38,z38);

F27: fulladder_logic port map(z24,y25,z05,y39,z39);

F28: fulladder_logic port map(z25,y26,y27,y40,z40);

F29: fulladder_logic port map(z26,z27,y28,y41,z41);

F30: fulladder_logic port map(z28,z29,y30,y42,z42);

F31: fulladder_logic port map(z30,z31,y32,y43,z43);

F32: fulladder_logic port map(z32,z16,y33,y44,z44);

F33: fulladder_logic port map(z33,x47,y34,y45,z45);

H13: halfadder_logic port map(y35,z34,y46,z46);

H14: halfadder_logic port map(z35,y36,y47,z47);

F34: fulladder_logic port map(x77,z21,z36,y48,z48);

H15: halfadder_logic port map(z37,y38,y49,z49);

```
F35: fulladder_logic port map(y39,z38,z49,y50,z50);
F36: fulladder_logic port map(y40,z39,z50,y51,z51);
F37: fulladder_logic port map(z40,y41,y29,y52,z52);
F38: fulladder_logic port map(z41,y31,y42,y53,z53);
F39: fulladder_logic port map(z14,z42,y43,y54,z54);
F40: fulladder_logic port map(y44,z43,z54,y55,z55);
F41: fulladder_logic port map(z44,y45,z55,y56,z56);
F42: fulladder_logic port map(y46,z45,z56,y57,z57);
F43: fulladder_logic port map(z46,y47,z57,y58,z58);
F44: fulladder_logic port map(y48,z47,z58,y59,z59);
H16: halfadder_logic port map(z51,y52,y60,z60);
F45: fulladder_logic port map(z52,y53,z60,y61,z61);
F46: fulladder_logic port map(z53,y54,z61,y62,z62);
H17: halfadder_logic port map(y55,z62,y63,z63);
H18: halfadder_logic port map(y56,z63,y64,z64);
H19: halfadder_logic port map(y57,z64,y65,z65);
H20: halfadder_logic port map(y58,z65,y66,z66);
H21: halfadder_logic port map(y59,z66,y67,z67);
F47: fulladder_logic port map(z48,z59,z67,y68,z68);
answer(0) <= x00;
answer(1) <= y01;
```

```
answer(2) <= y22;
answer(3) <= y37;
answer(4) <= y49;
answer(5) <= y50;
answer(6) <= y51;
answer(7) <= y60;
answer(8) <= y61;
answer(9) <= y62;
answer(10) <= y63;
answer(11) <= y64;
answer(12) <= y65;
answer(13) <= y66;
answer(14) <= y67;
answer(15) <= y68 or z68;
end struct_wallace;
```

Logic Unit:

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

```
entity logicunit is
port
(
a,b : in std_logic_vector(7 downto 0);
c : out std_logic_vector (15 downto 0));
end logicunit;
architecture logicunit_struct of logicunit is
component Wallace_logic is
  Port (M : in  std_logic_vector(7 downto 0);
        N : in  std_logic_vector (7 downto 0);
        answer : out std_logic_vector (15 downto 0));
end component;
component barrelshift_logic is
port
( P: in std_Logic_vector (15 downto 0);
 Q : out std_logic_vector (15 downto 0)
);
end component;
component Incrementer_logic is
port
(
```

```
Data1: in std_logic_vector (15 downto 0);
Data2: out std_logic_vector (15 downto 0)
);
end component;
signal x,y : std_logic_vector (15 downto 0);
begin
a1: Wallace_logic port map (a(7 downto 0), b(7 downto 0), x(15 downto 0));
a2: Incrementer_logic port map (y(15 downto 0), c(15 downto 0));
a3: barrelshift_logic port map (x(15 downto 0), y(15 downto 0));
end logicunit_struct;
```

D-Flipflop:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity Dff_logic is
port
(
d: in std_logic;
clk : in std_logic;
```

```
reset : in std_logic;
notq : out std_logic;
q : out std_logic);
end Dff_logic;

architecture Dff_struct of Dff_logic is
component nand_logic1 is
port (i : in std_logic ;
 j : in std_logic ;
 k : out std_logic);
end component;
component nand_logic2 is
port (o, p, q : in std_logic ;
  r: out std_logic);
end component;
signal  s,r,notQ1,Q1,t1,t2: std_logic;
begin
p1 : nand_logic1 port map (s, notq1, q1);
p2 : nand_logic2 port map (r,q1,reset, notq1);
p3 : nand_logic1 port map (s,t2, t1);
p4: nand_logic2 port map (t1, clk,reset, s);
```

```
p5 : nand_logic2 port map (s, clk,t2,r);
p6 : nand_logic2 port map (r,d,reset,t2);
q<=Q1;
notq<=notQ1;
end Dff_struct;
```

8 Bit register:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity Register8_logic is
port
(
qin: in std_logic_vector(7 downto 0);
clk,reset : in std_logic;
qout: out std_logic_vector (7 downto 0));
end Register8_logic;
architecture Register8_struct of Register8_logic is
   signal notq: std_logic_vector (7 downto 0);
    component Dff_logic is
  port
```

```
 (d: in std_logic;
 clk : in std_logic;
 reset : in std_logic;
 notq : out std_logic;
 q : out std_logic);
 end component;
 begin
 generate_loop: for i in  0 to 7 generate
 U1: Dff_logic port map (qin(i),clk,reset,notq(i),qout(i));
    end generate;
   end Register8_struct;
```

16 Bit Register:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity Register16_logic is
port
(
din: in std_logic_vector(15 downto 0);
clk,reset : in std_logic;
```

```
dout: out std_logic_vector (15 downto 0));
end Register16_logic;
architecture Register16_struct of Register16_logic is
  signal notd : std_logic_vector (15 downto 0);
component Dff_logic is
  port
(d: in std_logic;
clk : in std_logic;
reset : in std_logic;
notq : out std_logic;
q : out std_logic);
end component;
begin
 register_generate : for i in 0 to 15 generate
l1 : Dff_logic port map (din(i), clk, reset, notd(i), dout(i));
end generate;
end Register16_struct;
```

Arithmetic Logic Unit

```
library ieee;
```

```
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity final_logic is
port
(
A,B : in std_logic_vector(7 downto 0);
clock,load,reset : in std_logic;
Z : out std_logic_vector (15 downto 0);
endflag : out std_logic);
end final_logic;
architecture finallogic_struct of final_logic is
signal clear : std_logic := '0';
signal s1,s2 : std_logic_vector(7 downto 0);
signal z1,z2 : std_logic_vector(15 downto 0);
component Register8_logic is
port
(
qin: in std_logic_vector(7 downto 0);
clk,reset : in std_logic;
qout: out std_logic_vector (7 downto 0));
 end component;
```

```
component Register16_logic is
port
(
din: in std_logic_vector(15 downto 0);
clk,reset : in std_logic;
dout: out std_logic_vector (15 downto 0));
end component;
component logicunit is
port
(
a,b : in std_logic_vector(7 downto 0);
c : out std_logic_vector (15 downto 0));
end component;
begin
b1: Register8_logic port map (A(7 downto 0), clock, reset,s1(7 downto 0));
b2: Register8_logic port map (B(7 downto 0), clock, reset, s2(7 downto 0));
b3: logicunit port map (s1(7 downto 0), s2(7 downto 0), z1 (15 downto 0));
b4: Register16_logic port map (z1 (15 downto 0), clock, reset, z2(15 downto 0));
process(clock,load,reset)
 begin
  if (reset = '1') then
```

```
    if falling_edge(load) then
     clear <= '1';
end if;
else
 clear <='0';
end if;
end process;
process (z1,z2)
begin
 if (z1=z2) then
  endflag <='1';
  Z <= z2;
 else
   endflag <= '0';
end if ;
end process;
end finallogic_struct;
```

TEST BENCH

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

```
entity testbench is
end testbench;

architecture testbench_struct of testbench is

component final_logic is
port
(
A,B : in std_logic_vector(7 downto 0);
clock,load,reset : in std_logic;
Z : out std_logic_vector (15 downto 0);
endflag : out std_logic);
end component;

signal a,b : std_logic_vector(7 downto 0);
signal Clock : std_logic;
signal Load : std_logic;
signal Reset : std_logic;
signal Endflag : std_logic;
signal z : std_logic_vector(15 downto 0);

begin

  I1 : final_logic port map (a,b,Clock,Load,Reset,z,Endflag);

    process
    begin


        a<= "00001000";
        b<= "00001000";
        Clock<= '0';
        Reset <= '1';
        Load <= '0';

        wait for 100 ns;

         a<= "00001000";
         b<= "00001000";
         Clock<= '1';
```

```
 Reset <= '1';
 Load <= '0';

 wait for 100 ns;


  a<= "00000100";
  b<= "00000101";
  Clock<= '0';
  Reset <= '1';
  Load <= '0';

 wait for 100 ns;




a<= "00000100";
b<= "00000101";
Clock<= '1';
Reset <= '1';
Load <= '0';

wait for 100 ns;

a<= "10000100";
b<= "10000101";
Clock<= '0';
Reset <= '1';
Load <= '0';

wait for 100 ns;




 a<= "10000100";
 b<= "10000101";
 Clock<= '1';
 Reset <= '1';
```

```
        Load <= '0';

        wait for 100 ns;


    end process;
    end testbench_struct;
```

AREA REPORT

```
//  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_desine@grace.encs.concordia.ca #1 SMP Tue Nov 10
08:19:23 CST 2020 3.10.0-1160.6.1.el7.x86_64 x86_64

//

//  Start time Mon Dec  7 19:51:31 2020


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

Device Utilization for 2VP2fg256

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

Resource                          Used    Avail   Utilization

------------------------------------------------------------
```

IOs	36	140	25.71%
Global Buffers	0	16	0.00%
LUTs	8	2816	0.28%
CLB Slices	8	1408	0.57%
Dffs or Latches	16	3236	0.49%
Block RAMs	0	12	0.00%
Block Multipliers	0	12	0.00%
Block Multiplier Dffs	0	432	0.00%
GT_CUSTOM	0	4	0.00%

****

Library: work Cell: final_logic View: finallogic_struct

****

Cell	Library	References			Total Area
GND	xcv2p	1	x		
IBUF	xcv2p	18	x		
LD	xcv2p	16	x	1	16 Dffs or Latches
LUT4_L	xcv2p	8	x	1	8 LUTs
MUXCY	xcv2p	1	x	1	1 MUX CARRYs
MUXCY_L	xcv2p	7	x	1	7 MUX CARRYs
OBUF	xcv2p	17	x		

```
 Register16_logic    work      1 x      1      1 Register16_logic
 Register8_logic     work      2 x      1      2 Register8_logic
 VCC                 xcv2p     1 x
 logicunit           work      1 x      1      1 logicunit

 Number of ports :                          36
 Number of nets :                          135
 Number of instances :                      73
 Number of references to this view :         0

Total accumulated area :
 Number of Dffs or Latches :                16
 Number of LUTs :                            8
 Number of MUX CARRYs :                      8
 Black Box Register16_logic :                1
 Black Box Register8_logic :                 2
 Number of gates :                           8
 Black Box logicunit :                       1
 Number of accumulated instances :          73


****************************
 IO Register Mapping Report
****************************
Design: work.final_logic.finallogic_struct
```

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			
reset	Input			
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			
endflag	Output			

Total registers mapped: 0

APPENDIX

PROJECT TASK BREAKDOWN

SL.NO	Implementation & Synthesis	Nandini Panneer Selvam	Rama Krishna Desineni	Ranjitha Arumugum	Varsha Suresh
1.	Basic Gates (OR, AND, XOR, NOT)		✓	✓	
2.	NAND gates	✓			✓
3.	Half Adder	✓			
4.	Full Adder		✓		
5.	2:1 Multiplexer			✓	
6.	D-Flip Flop				✓
7.	Incrementor			✓	
8.	16-Bit Barrel Shifter	✓			
9.	8-Bit Register		✓		
10.	Wallace Multiplier Unit			✓	
11.	Arithmetic Logic Unit	✓			
12.	Test Bench For 8-bit Arithmetic Unit		✓		✓

SL.NO	Documentation	Nandini Panneer Selvam	Rama Krishna Desineni	Ranjitha Arumugum	Varsha Suresh
1.	Project description and Design Specification	✓	✓		
2.	Simulation, Synthesis and Implementation of			✓	✓

	Functional Unit Blocks				
3.	Implementation, Simulation and Synthesis of Arithmetic Logical Unit	✓			✓
4.	Conclusion		✓	✓	
5.	Coding	✓	✓	✓	✓