Concordia University
Engineering and Computer Science

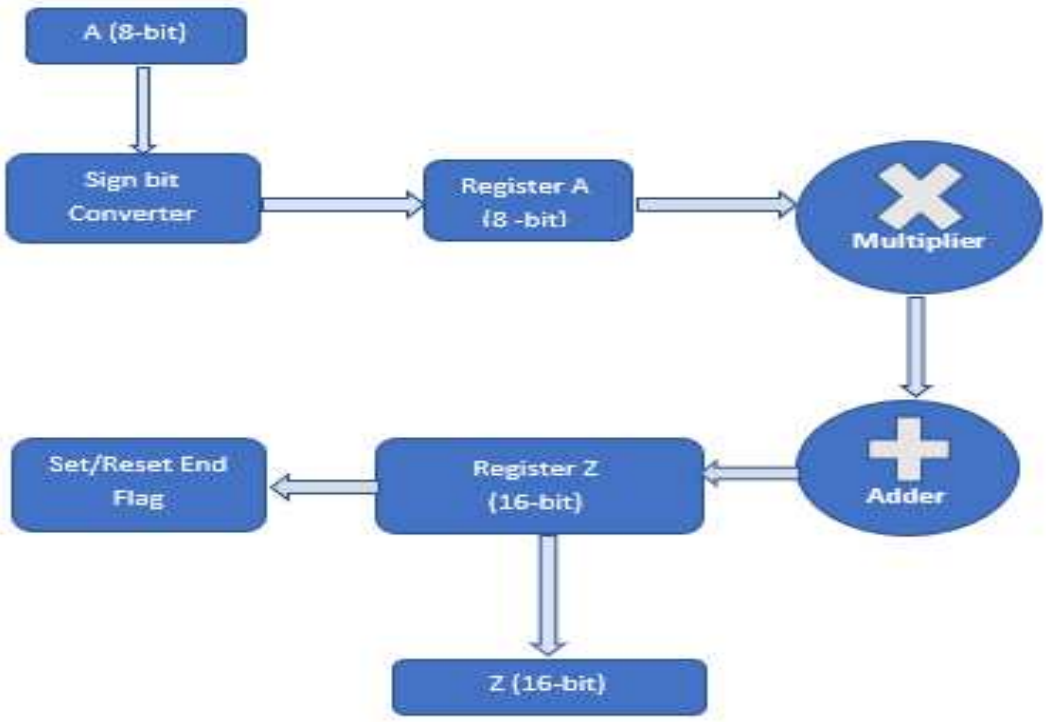
Fall 2018

COEN 6501: Digital Design and Synthesis

Dr. Asim J. Al-Khalili

Project Report

Arithmetic Unit Implementation which is capable of calculating $A^2 - 1$



Team Members

Amulya Prabakhar (**40089026**)

Sibi Ravichandran (**40076654**)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

In this Project, we have implemented an arithmetic unit which performs $\mathbf{A^2 - 1}$ operation, which takes 8-bit signed input and results in 16-bit output. The design methodology used is by converting signed number to unsigned one and feeding it to multiplier to carry out squaring. We have proposed two different methodologies for the squarer unit. Later, pass the result through adder to perform minus one operation.

We have used the structural design for constructing VHDL code and Test Bench. ModelSim is the tool used for compilation, simulation and for RTL. Also we have used XILINX ISE to find area and timing report through which we can determine the performance of the unit.

ACKNOWLEDGEMENT

On the very outset of this report, we would like to extend our sincere obligation towards allthe personages who have helped us in this endeavour. Without their active guidance, help, cooperation and encouragement, we would have not made headway in the project. We take this opportunity to express our sincere thanks to **Dr. Asim J. Al-Khalili**, who instructed this course on Digital Design and Synthesis and gave us an opportunity to learn to design the digital device, understanding the design algorithms and VHDL in depth. His guidance throughout the course time, his lecture slides and the materials he provided in the website helped us a lot to complete this project successfully. We are always thankful to Concordia University and its labs without which we were not able to do our project report. We also wish to extend our regards to **Anish Goel** for helping us to get hands on tools used and for his ceaseless encouragement andcontinual support to us towards the completion of the project.

1. INTRODUCTION

The main aim of the project is to implement an arithmetic unit which can calculate **$A^2 - 1$.** The input for the system being an operand A, which is 8-bit signed number. The unit needs to be designed in such a way that a 1 to 0 transition at load should latch the operand into Internal register RA. The 16bit result produced from the unit outputs from register RZ output port.

The low power and high speed VLSI can be implemented with different logic style. The three important considerations for VLSI design are power, area and delay. There are many proposed logics (or) low power dissipation and high speed and each logic style has its own advantages in terms of speed and power.

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. BLOCK DIAGRAM AND DESCRIPTION

2.1. UNIT BLOCK

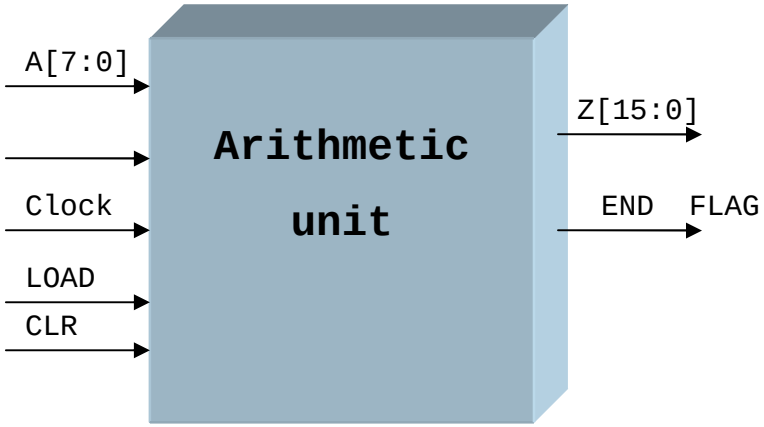


Figure 1. The Unit black box

Requirement number	Description
R1	The design shall be structural.
R2	The operand A is latched into register RA when LOAD signal transit from high to low.
R3	The CLEAR signal will clear all registers to '0'.
R4	The 16-bit multiplication product shall be loaded into the 16-bit Z port.
R5	The unit performs the arithmetic operation until END_FLAG becomes high.
R6	The Test Bench, & Stimulator can be constructed using "Algorithmic" modelling method.

Table 1. Design Requirements

Signal Specifications

Signal Name	Signal Description
A0-A7	Signed 8-bit Operand A
Z0-Z15	Signed Output
LR	Clears selected registers
LOAD	Loads Operand into internal register
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 signed number
2. Use signed bit converter to convert sign number to unsigned number
3. Use registers to load 8-bit value until the load becomes high
4. The 8-bit unsigned number is now fed to multiplier to find the square of a number. Initially find all the partial products of the multiplication. Later, add all the partial products to find the final result.Two different ways are used for addition of partial products, which has been explained in detail in section 4.3.3.
5. After squaring, the result is then fed to adder which performs addition of a result with '1'.
6. 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.

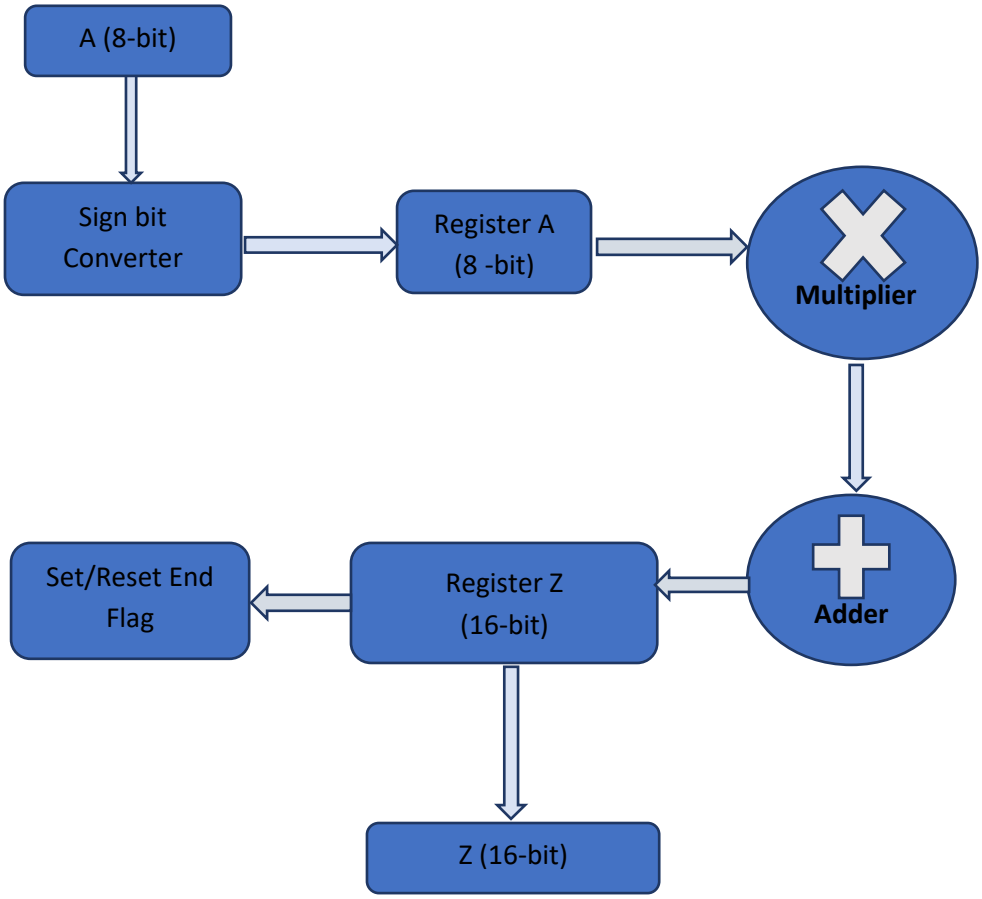


Figure 2. Detailed Block Diagram of the Unit

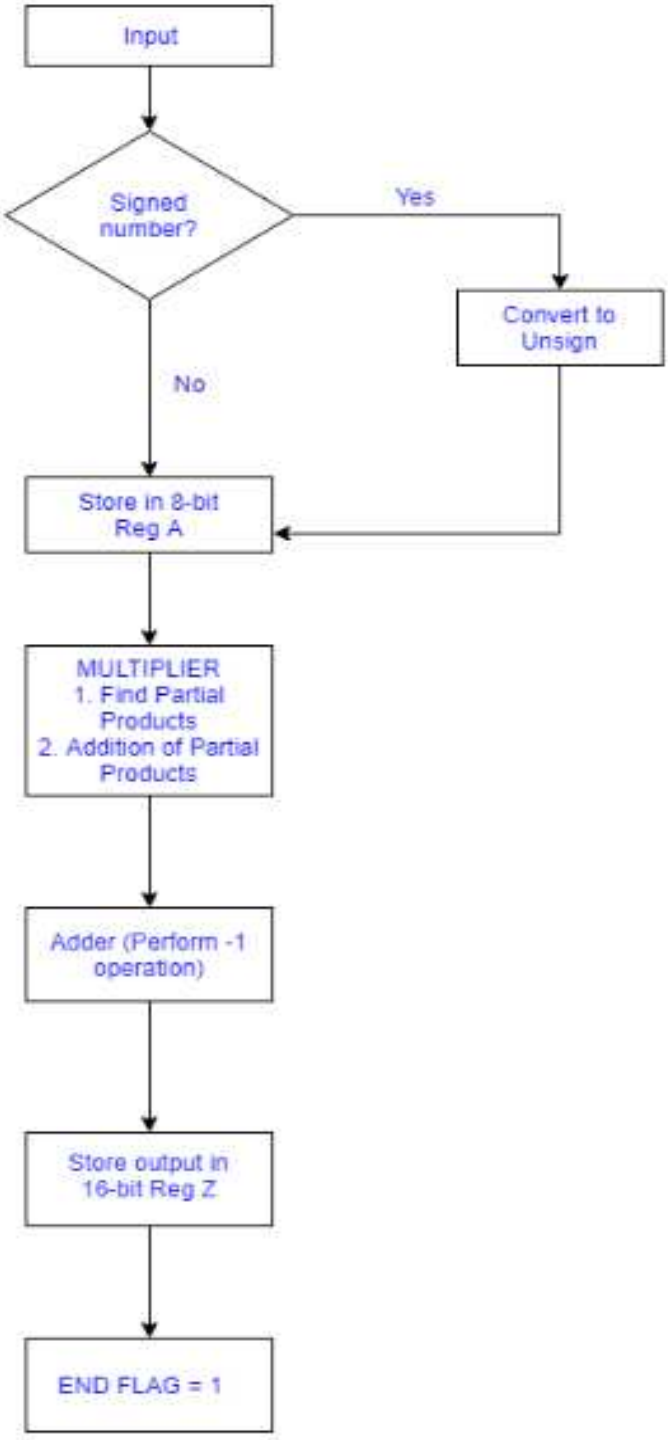


Figure 3. Flow Chart of the Unit

3. DESIGN AND ANALYSIS OF COMPONENTS

List of components used,

3.1. Basic Logic Gates.
3.2. Half Adders.
3.3. Full Adders.
3.4. Ripple Carry Adders.
3.5. Registers.

3.1. Basic Logic Gates:

3.1.1. AND Gate:

The **AND gate** performs logical conjunction.

A HIGH output is obtained only when all the inputs are HIGH.

Refer to section A1 and A2 for VHDL code and test bench.

Figure 4. Logic Symbol of AND Gate

Table 2. Truth table of AND Gate

INPUT		OUTPUT
A	B	C = A AND B
0	0	0
0	1	0
1	0	0
1	1	1

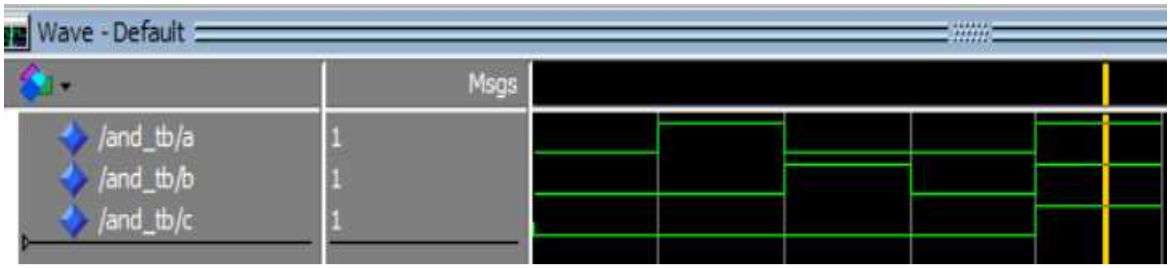


Figure 5. Simulation Result of AND gate

3.1.2. OR Gate:

The **OR gate** performs logical disjunction.

A HIGH output is obtained only when one or all the inputs are HIGH.

Refer to section A3 and A4 for VHDL code and test bench.

Figure 6. Logic Symbol of OR Gate

INPUT		OUTPUT
A	B	C= A OR B
0	0	0
0	1	1
1	0	1
1	1	1

Table 3. Truth table of OR Gate

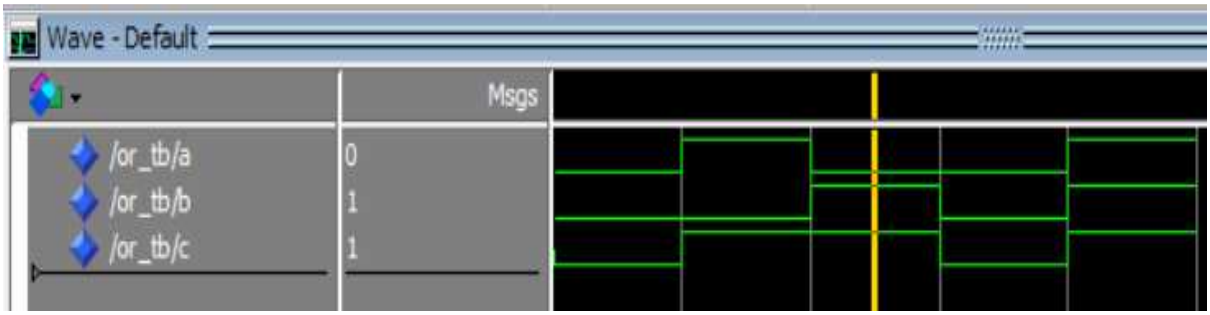


Figure 7. Simulation Result of OR gate

3.1.3. NOT Gate:

The **NOT gate** performs logical negation, A HIGH input results in LOW output and vice versa.

Refer to section A5 and A6 for VHDL code and test bench.

INPUT	OUTPUT
A	C= NOT A
0	1
1	0

Table 4. Truth table of NOT Gate

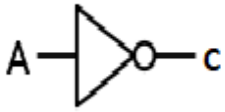


Figure 8. Logic Symbol of NOT Gate

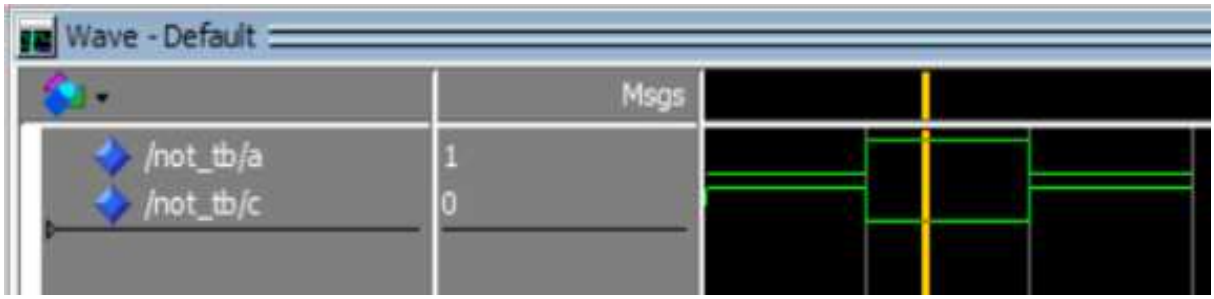


Figure 9. Simulation Result of NOT gate

3.1.4. NAND Gate:

The **NAND gate** performs logical inversion of AND gate.

A LOW output is obtained only when all the inputs are HIGH.

Refer to section A7 and A8 for VHDL code and test bench.

INPUT		OUTPUT
A	B	C= A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

Table 5. Truth table of NAND Gate

Figure 10. Logic Symbol of NAND Gate

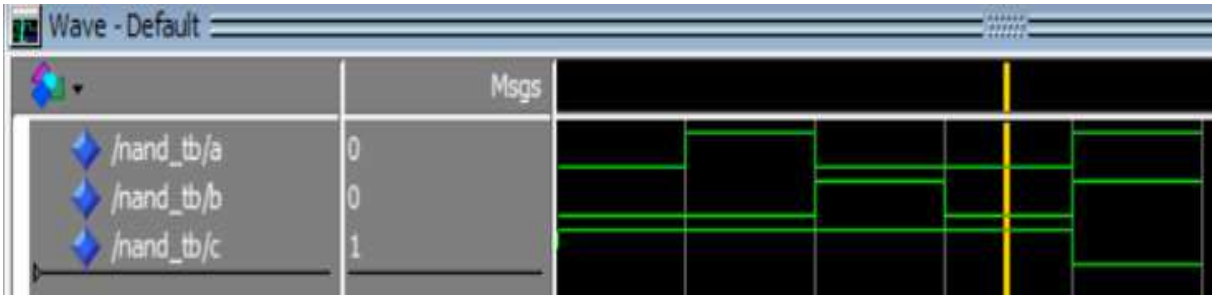


Figure 11. Simulation Result of NAND gate

3.1.5. NOR Gate:

The **NOR gate** performs logical inversion of OR gate.

A LOW output is obtained only when one or all the inputs are HIGH.

Refer to section A9 and A10 for VHDL code and test bench.

INPUT		OUTPUT
A	B	C= A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

Table 6. Truth table of NOR Gate

Figure 12. Logic Symbol of NOR Gate

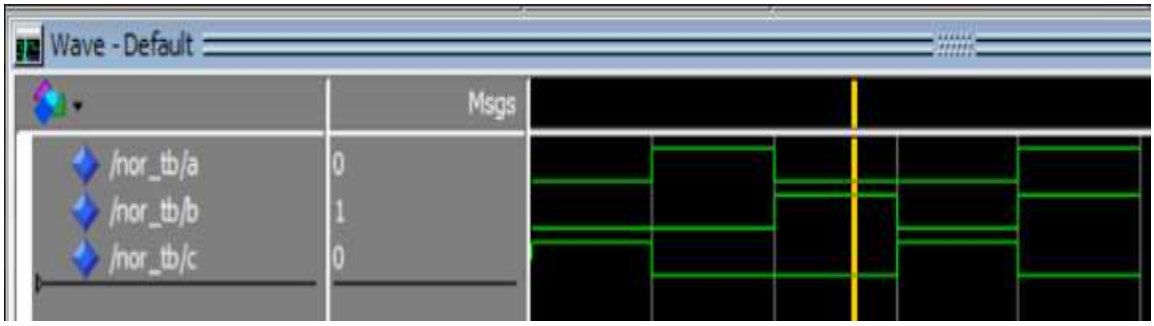


Figure 13. Simulation Result of NOR gate.

3.1.6. XOR Gate:

The **XOR gate** performs addition modulo 2.

A HIGH output is obtained only when the inputs are not same.

Refer to section A11 and A12 for VHDL code and test bench.

INPUT		OUTPUT
A	B	C= A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Table 7. Truth table of XOR Gate

Figure 14. Logic Symbol of XOR Gate

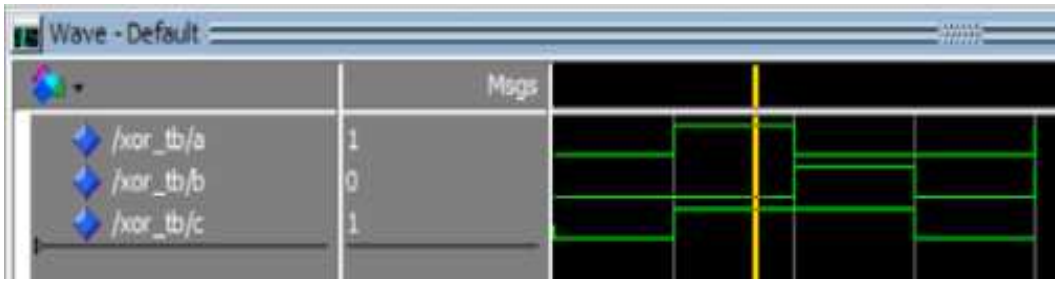


Figure 15. Simulation Result of XOR gate.

3.1.7. XNOR Gate:

The **XNOR gate** performs logical complement of XOR gate.

A HIGH output is obtained only when the inputs are same.

Refer to section A13 and A14 for VHDL code and test bench.

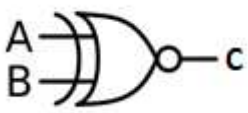


Figure 16. Logic Symbol of XNOR Gate

INPUT		OUTPUT
A	B	C= A XNOR B
0	0	1
0	1	0
1	0	0
1	1	1

Table 8. Truth table of XNOR Gate

Figure 17. Simulation Result of XNOR gate.

3.2. Half Adder:

The **half adder** adds two single binary digits *A* and *B*. It has two outputs, sum (*sum*) and carry (*carry*). The carry signal represents an overflow into the next digit of a multi-digit addition. The simplest half-adder design as per figure, incorporates an XOR gate for sum and an AND gate for carry. The Boolean logic for the sum (in this case sum) will be A′B + AB′ whereas for the carry (carry) will be AB. With the addition of an OR gate to combine their carry outputs, two half adders can be combined to make a full adder. The half adder adds two input bits and generates a carry and sum, which are the two outputs of a half adder.

Refer to section A15 and A16 for VHDL code and test bench.

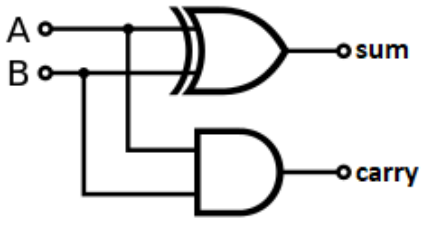


Figure 18. Logic Symbol of Half Adder.

Table 9.

INPUT		OUTPUT	
A	B	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Table 9. Truth table of Half Adder.

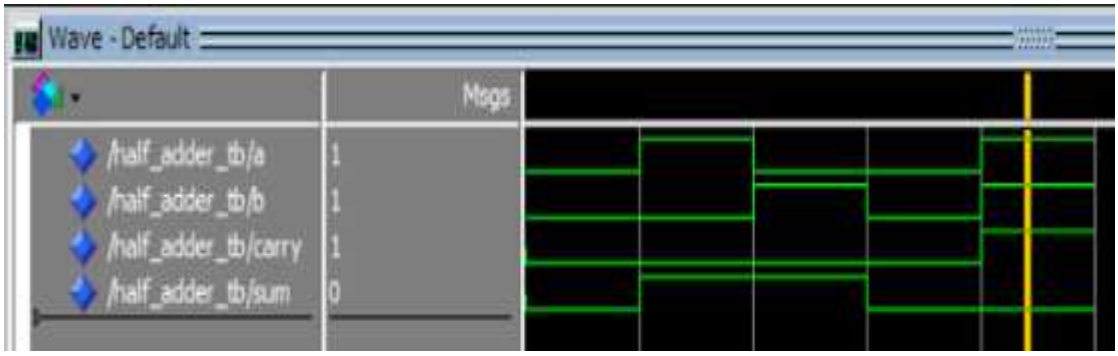


Figure 19. Simulation Result of Half Adder.

3.3. Full Adder:

A **full adder** adds binary numbers and accounts for values carried in as well as out. A one-bit full-adder adds three one-bit numbers, often written as *A*, *B*, and *carry_in*; *A* and *B* are the operands, and *carry_in* is a bit carried in from the previous less-significant stage.The full adder is usually a component in a cascade of adders, which add 8, 16, 32, etc. bit binary numbers. The circuit produces a two-bit output. Output carry and sum typically represented by the signals *carry_out* and *sum*, where the sum equals ***2carry_out + sum.***

A full adder can be implemented in many different ways such as with a custom transistor-level circuit or composed of other gates. One example implementation is with (refer Figure 20)

$$sum = A \oplus B \oplus carry\_in.$$

$$carry\_out = (A \cdot B) + (carry\_in \cdot (A \oplus B)).$$

A full adder can also be constructed from two half adders as per by connecting *A* and *B* to the input of one half adder, then taking its sum-output *sum* as one of the inputs to the second half adder and *carry_in* as its other input, and finally the carry outputs from the two half-adders are connected to an OR gate. (Refer Figure 21) The sum-output from the second half adder is the final sum output (*sum*) of the full adder and the output from the OR gate is the final carry output (carry_out). The critical path of a full adder runs through both XOR gates and ends at the sum bit *s*. Assumed that an XOR gate takes 1 delays to complete, the delay imposed by the critical path of a full adder is equal to

$$T_{\mathrm{FA}} = 2 \cdot T_{\mathrm{XOR}} = 2D.$$

The critical path of a carry runs through one XOR gate in adder and through 2 gates (AND and OR) in carry-block and therefore, if AND or OR gates take 1 delay to complete, has a delay of

$$T_{\mathrm{c}} = T_{\mathrm{XOR}} + T_{\mathrm{AND}} + T_{\mathrm{OR}} = D + D + D = 3D.$$

Refer to section A17 and A18 for VHDL code and test bench.

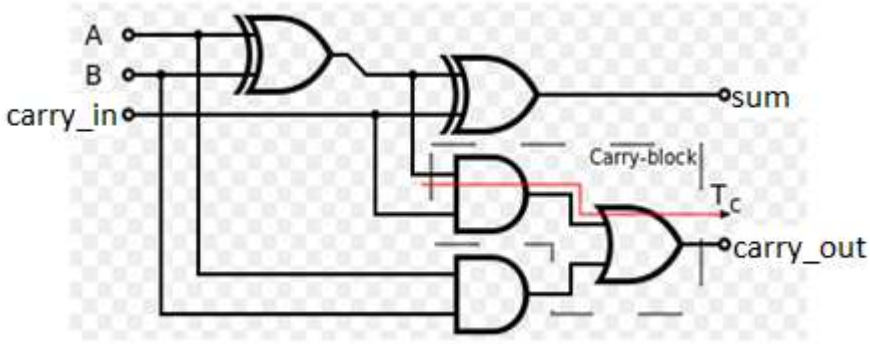


Figure 20. Logical diagram of Full Adder.

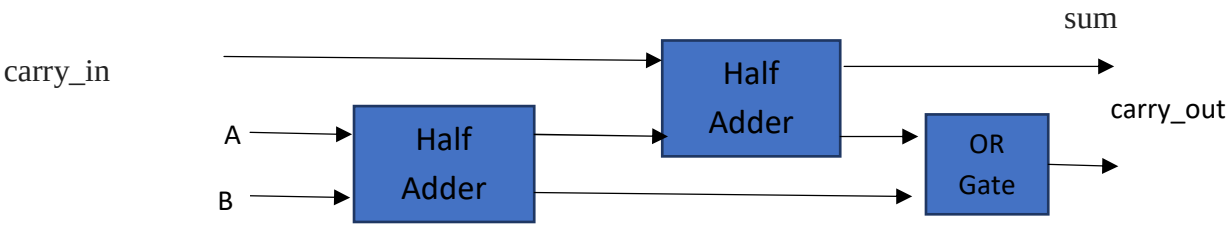


Figure 21. Full Adder using 2 Half Adders.

INPUT			OUTPUT	
A	B	carry_in	carry_out	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 10. Truth table of Full Adder.

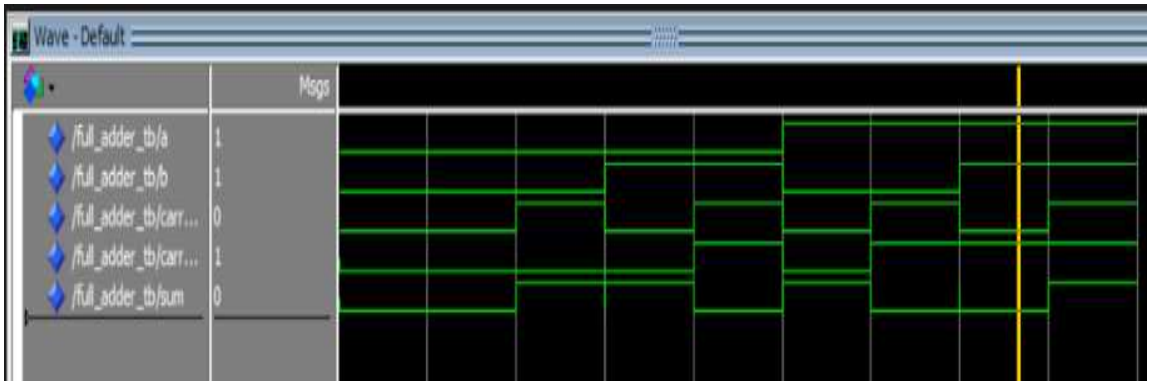


Figure 22. Simulation Result of Full Adder.

3.4. Ripple Carry Adders:

A ripple carry adder is a digital circuit that produces the arithmetic sum of two binary numbers. It can be constructed with full adders connected in cascade, with the carry output from each full adder connected to the carry input of the next full adder in the chain. Figure 23 shows the interconnection of four full adder (FA) circuits to provide a 4-bit ripple carry adder.

In the ripple carry adder, the output is known after the carry generated by the previous stage is produced. Thus, the sum of the most significant bit is only available after the carry signal has rippled through the adder from the least significant stage to the most significant stage. As a result, the final sum and carry bits will be valid after a considerable delay.

A 8-bit ripple carry adder can be constructed using two 4-bit ripple carry adder as shown in the Figure 24, Refer to section A19 to A22 for VHDL code and Test Bench.

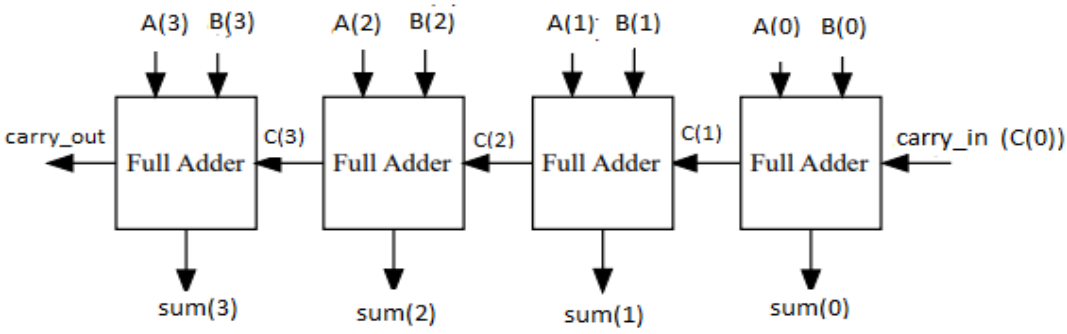


Figure 23. Block Diagram of 4-bit Ripple Carry Adder.

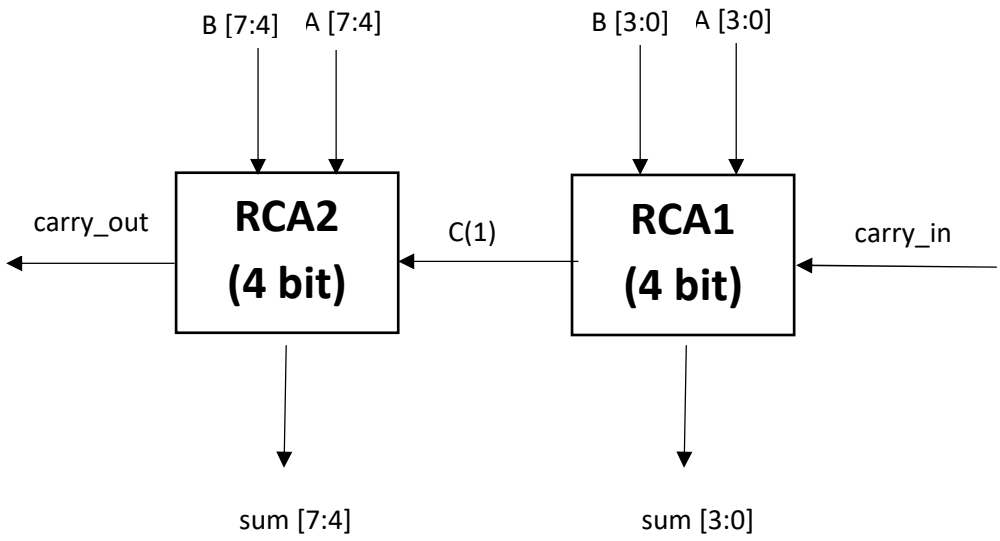


Figure 24. Block Diagram of 8-bit Ripple Carry Adder

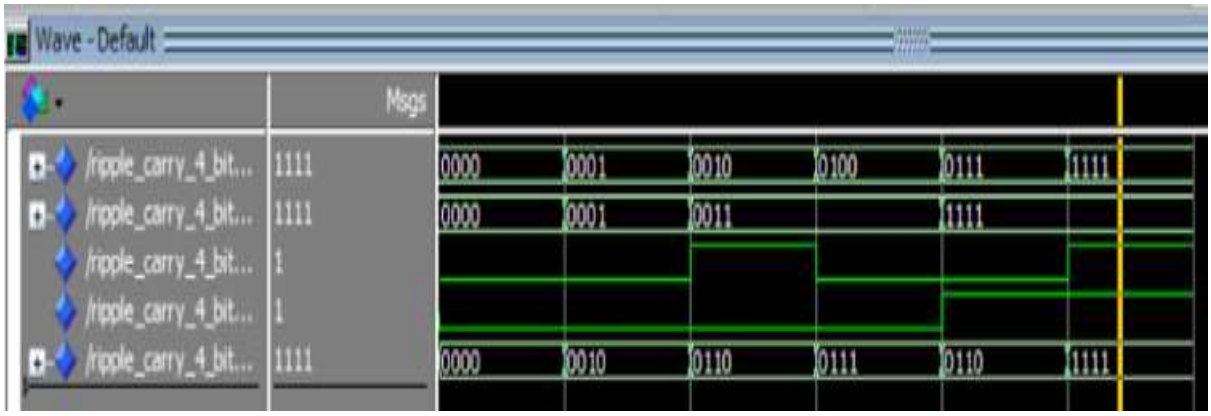


Figure 25. Simulation Result of 4-bit Ripple Carry Adder.

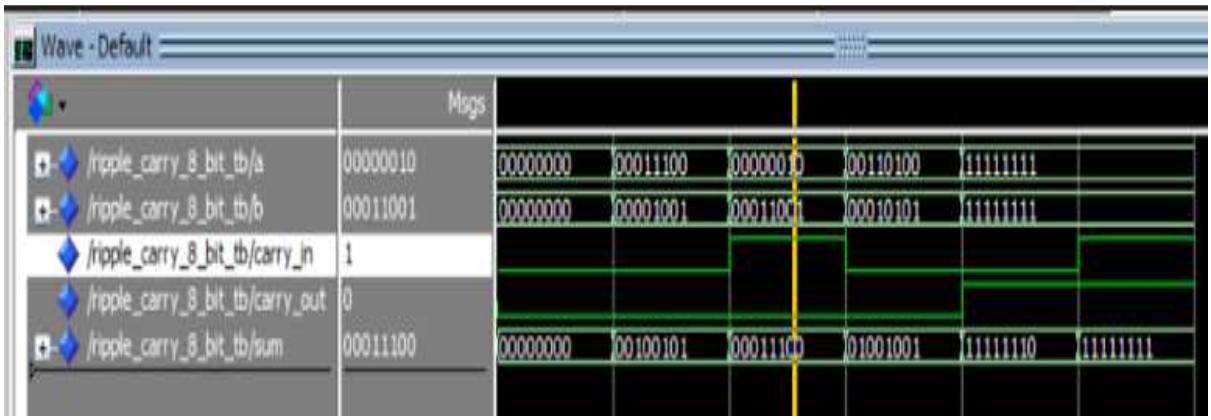


Figure 26. Simulation Result of 8-bit Ripple Carry Adder.

3.5. Registers:

Parallel in Parallel out (PIPO) shift registers are the type of storage devices in which both data loading as well as data retrieval processes occur in parallel mode. Figure 27 shows a PIPO register capable of storing n-bit input data word (Data in). Here each flip-flop stores an individual bit of the data in appearing as its input (FF_1 stores B_1 appearing at D_1; FF_2 stores B_2 appearing at D_2 … FF_n stores B_n appearing at D_n) at the instant of first clock pulse. Further, at the same instant, the bit stored in each individual flip-flop also appears at their respective output pins ($Q_1 = D_1$; $Q_2 = D_2$ … $Q_n = B_n$). This indicates that both data storage as well as data recovery occur at a single (and at the same) clock pulse in PIPO registers.

Here if $SH/\overline{LD}$ line goes low, A_2 AND gates of all the combinational circuits become active while A_1 gates become inactive. Thus the bits of the input data word (Data in) appearing as inputs to the gates A_2 are passed on as the OR gate outputs which are further loaded/stored into respective flip-flops at the appearance of first leading edge of the clock (except the bit B_1 which gets directly stored into FF_1 at the first clock tick). This indicates that all the bits of the input data word are stored into the register components at the same clock tick. At the same time, these bits also appear at the output pins of the respective flip-flops thus yielding parallel-output data word at the same clock tick.Refer to section A23 to A26for VHDL code and test bench.

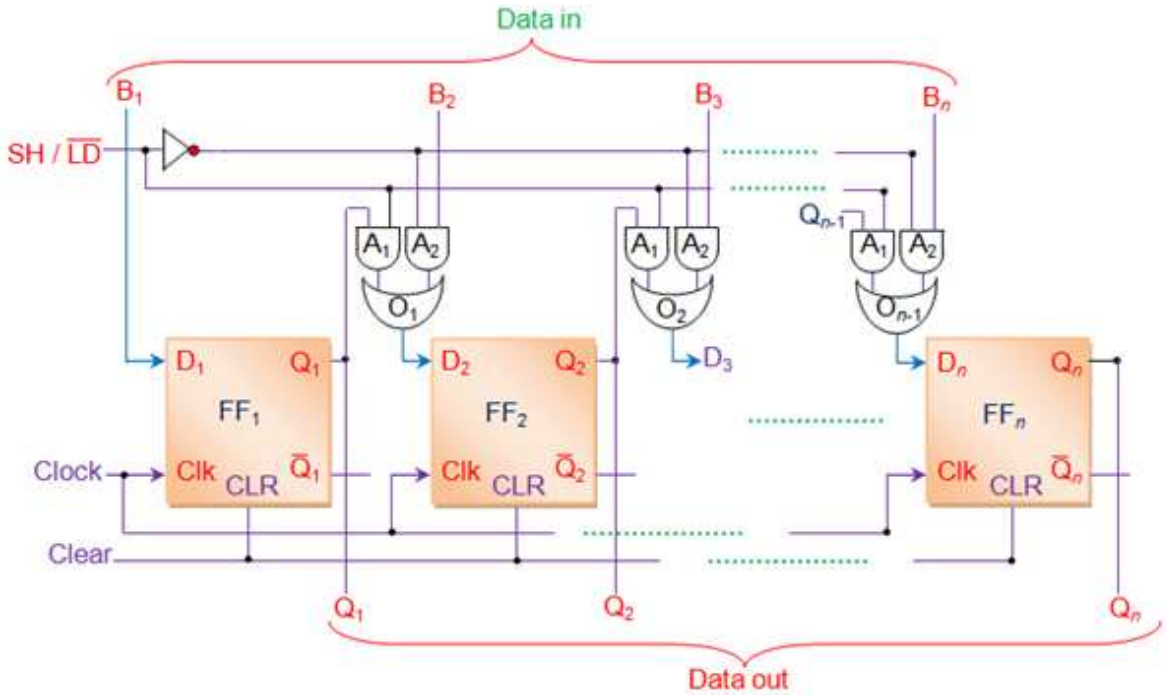


Figure 27. Block Diagram of Parallel In Parallel Out Registers.

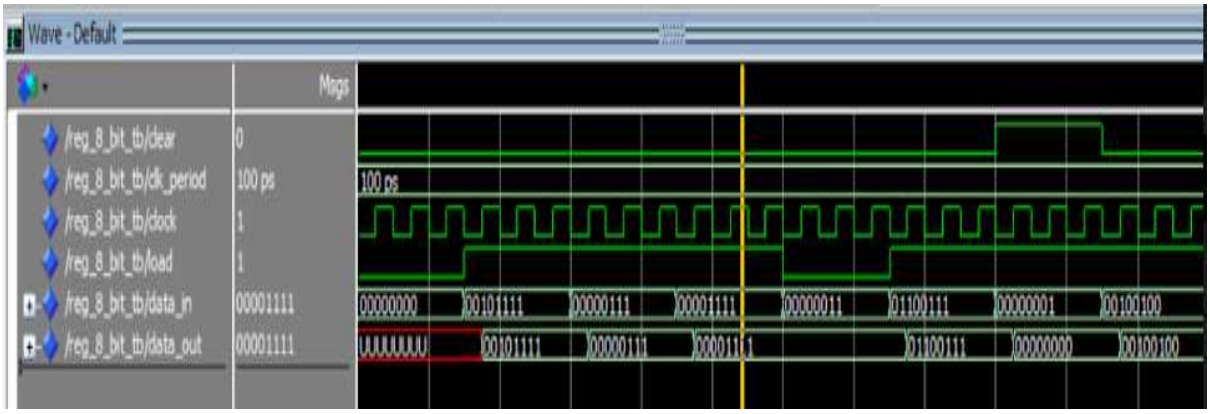


Figure 28. Simulation Result of 8-bit Parallel in Parallel out Registers.

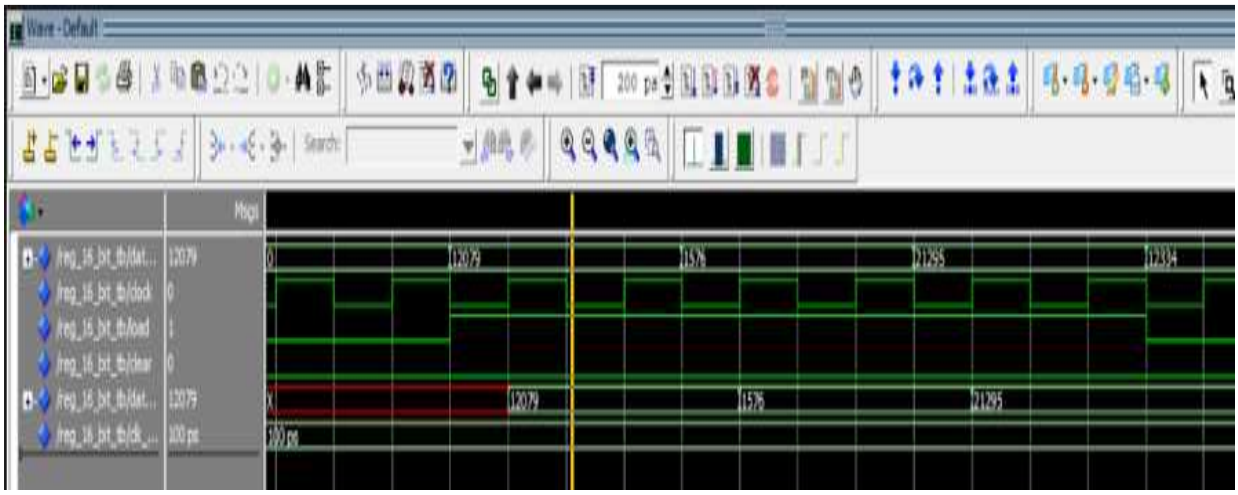


Figure 29. Simulation Result of 16-bit Parallel in Parallel out Registers.

4. IMPLEMENTATION OF ARITHMETIC UNIT.

4.1. Conversion of Signed bit to unsigned bit:

This component is used to convert an 8-bit signed number to an 8-bit unsigned number. The component will receive the input. Once it has received the input, the component will check whether the MSB (input) is equal to 1 or 0. If the Most Significant Bit is equal to one then the input is a negative number else it is a positive number. If the input is positive, the input is directly passed as output. If the input is negative, the component will use the Not gates and take one's complement of the input and then use a 8-bit Ripple carry adder to add '1' to the one's complement obtained from the previous step. This 2's complement number is given as output of the component. Refer to A27 and A28 for VHDL code and Test Bench.

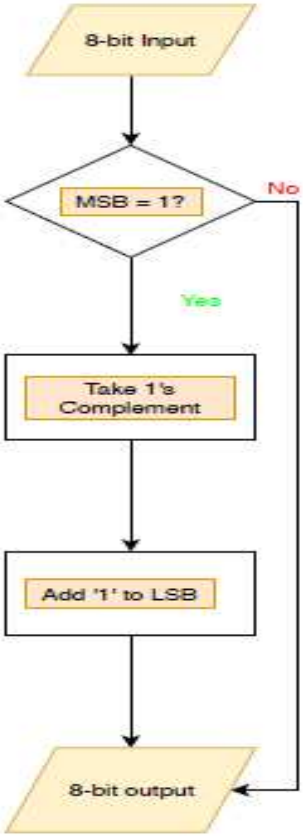


Figure 30. Flow chart of Signed to Unsigned bit Conversion.

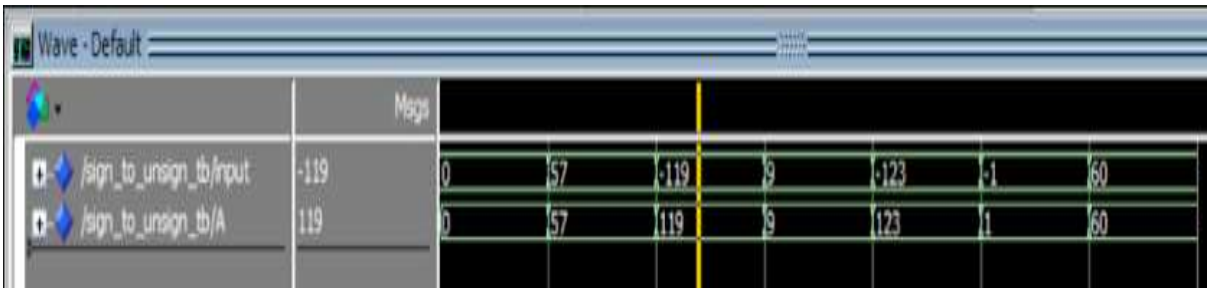


Figure 31. Simulation result of Signed to Unsigned bit Conversion.

4.2. Use of Registers to load value:

During raising edge of the clock, when Clear is not enabled and Load is set to 1, Load the unsigned 8-bit number into the 8-bit register.

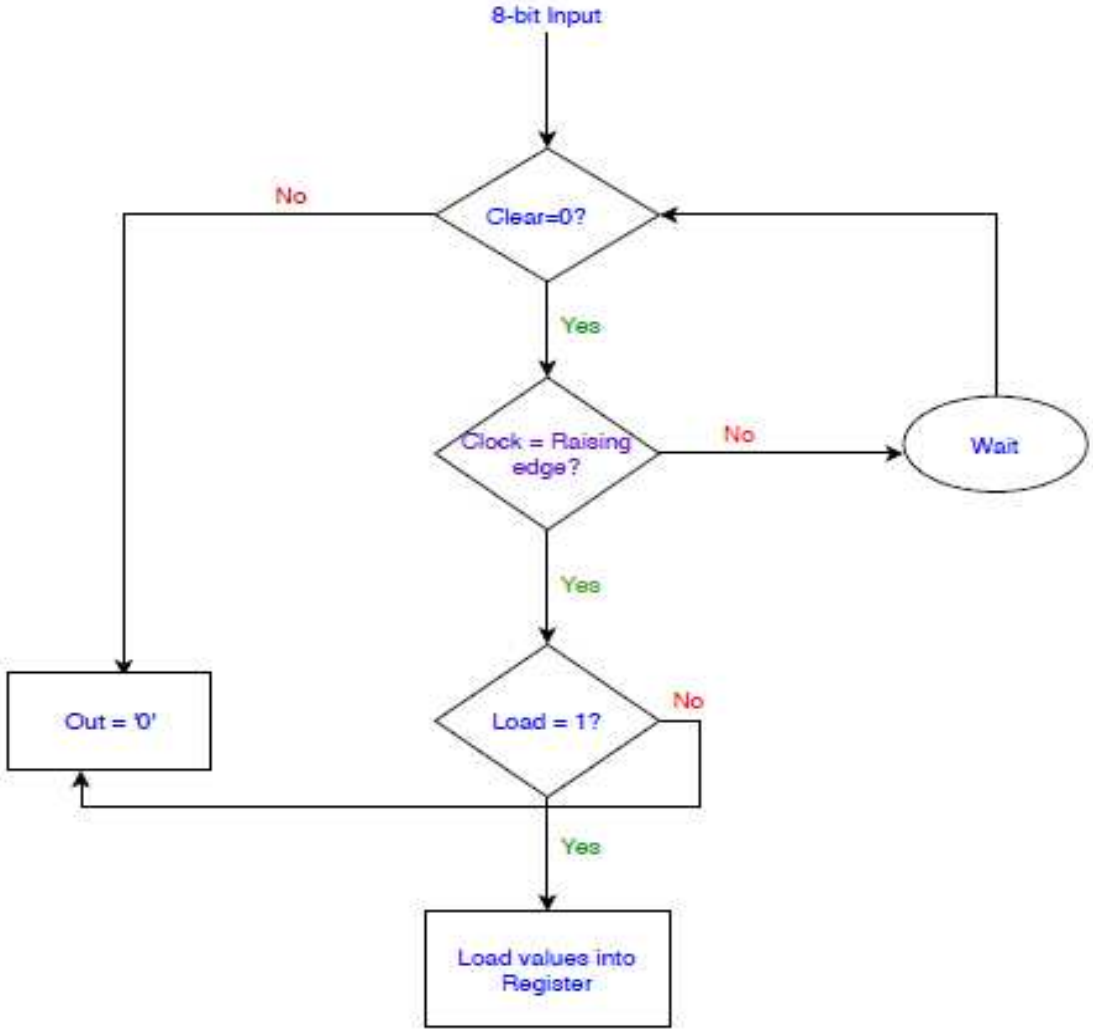


Figure 32. Algorithm used to store value to the register.

Figure 33. Simulation result of 8-bit register.

4.3. Squarer Implementation:

After getting the 8-bit input, multiplication is done to get the square of the 8-bit input. In our case, since we have converted the unsigned number into signed number, a normal multiplier algorithm is implemented.

4.3.1. Algorithm:

The multiplication algorithm for a 4- bit multiplicand by 4-bit multiplier is shown in Figure 34:

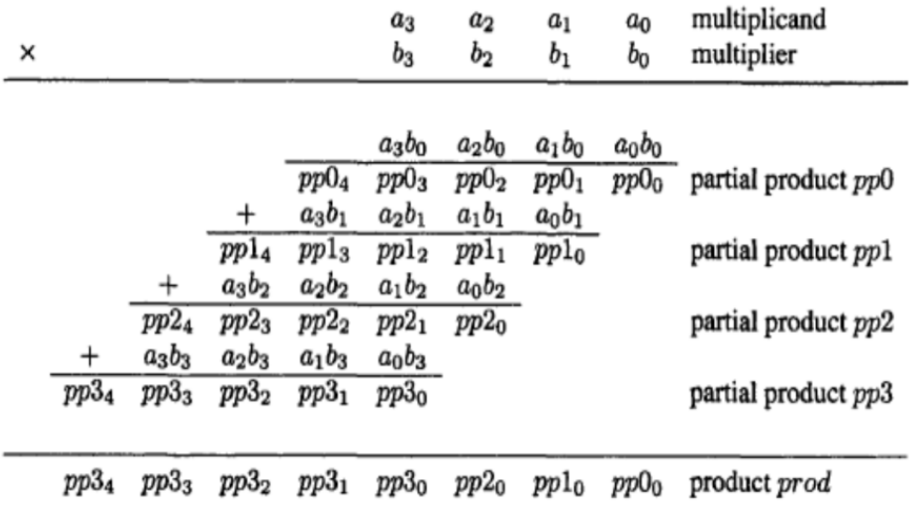


Figure 34. Algorithm used for multiplication.

Since the arithmetic unit has performed a square operation, the multiplier and multiplicand are same in this case. The input to the circuit is an 8-bit unsigned positive number. **Input: A= $a_7a_6a_5a_4a_3a_2a_1a_0$**

4.3.2. Partial Products:

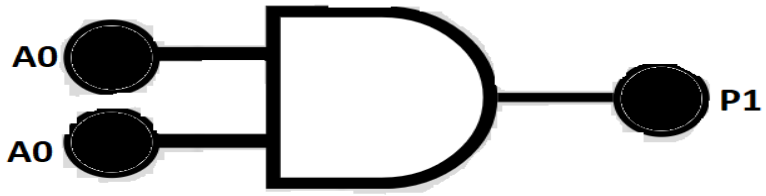
Once the multiplier gets the 8-bit input, the next step is to determine the partial products. The ***number of partial products is given by m*n*** where m – the number of bits is multiplicand and n- the number of bits in multiplier. In this case the ***number of partial products is 64*** since the number 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. Table 11 gives all the partial products, their representations and their formulas.

PP	Bits	PP	Bits	PP	Bits	PP	Bits	PP	Bits	PP	Bits	PP	Bits	PP	Bits
P1	a_0 a_0	P9	a_1 a_0	P17	a_2 a_0	P25	a_3 a_0	P33	a_4 a_0	P41	a_5 a_0	P49	a_6 a_0	P57	a_7 a_0
P2	a_0 a_1	P10	a_1 a_1	P18	a_2 a_1	P26	a_3 a_1	P34	a_4 a_1	P42	a_5 a_1	P50	a_6 a_1	P58	a_7 a_1
P3	a_0 a_2	P11	a_1 a_2	P19	a_2 a_2	P27	a_3 a_2	P35	a_4 a_2	P43	a_5 a_2	P51	a_6 a_2	P59	a_7 a_2
P4	a_0 a_3	P12	a_1 a_3	P20	a_2 a_3	P28	a_3 a_3	P36	a_4 a_3	P44	a_5 a_3	P52	a_6 a_3	P60	a_7 a_3
P5	a_0 a_4	P13	a_1 a_4	P21	a_2 a_4	P29	a_3 a_4	P37	a_4 a_4	P45	a_5 a_4	P53	a_6 a_4	P61	a_7 a_4
P6	a_0 a_5	P14	a_1 a_5	P22	a_2 a_5	P30	a_3 a_5	P38	a_4 a_5	P46	a_5 a_5	P54	a_6 a_5	P62	a_7 a_5
P7	a_0 a_6	P15	a_1 a_6	P23	a_2 a_6	P31	a_3 a_6	P39	a_4 a_6	P47	a_5 a_6	P55	a_6 a_6	P63	a_7 a_6
P8	a_0 a_7	P16	a_1 a_7	P24	a_2 a_7	P32	a_3 a_7	P40	a_4 a_7	P48	a_5 a_7	P56	a_6 a_7	P64	a_7 a_7

Table 11. Partial Products table for 8-bit Inputs.

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.



AND gate to generate Partial Product.

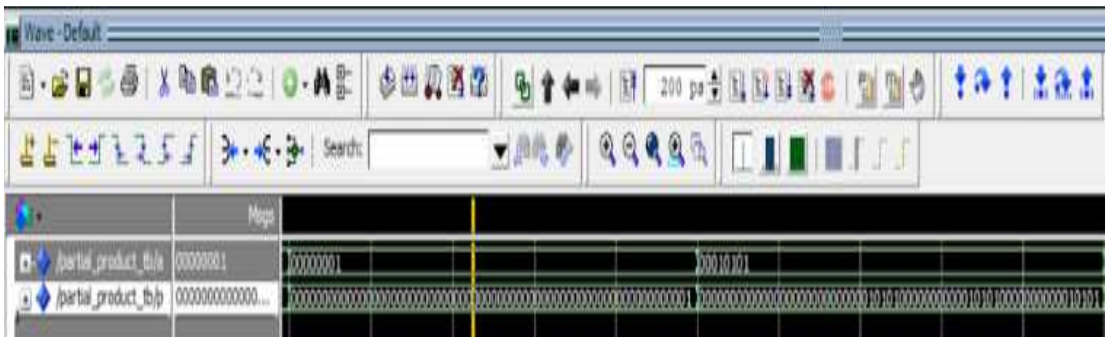


Figure 35. Simulation Result of Partial Products Generation.

4.3.3. Addition of Partial Products:

Once all the partial products have been determined we have to perform the addition on the partial products which is given in Table 12:

Table 12. Simplification of Partial Products table for 8-bit Inputs.

MULTIPLICAND								**a_7**	**a_6**	**a_5**	**a_4**	**a_3**	**a_2**	**a_1**	**a_0**
MULTIPLIER								**a_7**	**a_6**	**a_5**	**a_4**	**a_3**	**a_2**	**a_1**	**a_0**
PARTIAL PRODUCTS															
								$a_0\ a_7$	$a_0\ a_6$	$a_0\ a_5$	$a_0\ a_4$	$a_0\ a_3$	$a_0\ a_2$	$a_0\ a_1$	$a_0\ a_0$
							$a_1\ a_7$	$a_1\ a_6$	$a_1\ a_5$	$a_1\ a_4$	$a_1\ a_3$	$a_1\ a_2$	$a_1\ a_1$	$a_1\ a_0$	
						$a_2\ a_7$	$a_2\ a_6$	$a_2\ a_5$	$a_2\ a_4$	$a_2\ a_3$	$a_2\ a_2$	$a_2\ a_1$	$a_2\ a_0$		
					$a_3\ a_7$	$a_3\ a_6$	$a_3\ a_5$	$a_3\ a_4$	$a_3\ a_3$	$a_3\ a_2$	$a_3\ a_1$	$a_3\ a_0$			
				$a_4\ a_7$	$a_4\ a_6$	$a_4\ a_5$	$a_4\ a_4$	$a_4\ a_3$	$a_4\ a_2$	$a_4\ a_1$	$a_4\ a_0$				
			$a_5\ a_7$	$a_5\ a_6$	$a_5\ a_5$	$a_5\ a_4$	$a_5\ a_3$	$a_5\ a_2$	$a_5\ a_1$	$a_5\ a_0$					
		$a_6\ a_7$	$a_6\ a_6$	$a_6\ a_5$	$a_6\ a_4$	$a_6\ a_3$	$a_6\ a_2$	$a_6\ a_1$	$a_6\ a_0$						
	$a_7\ a_7$	$a_7\ a_6$	$a_7\ a_5$	$a_7\ a_4$	$a_7\ a_3$	$a_7\ a_2$	$a_7\ a_1$	$a_7\ a_0$							
AFTER SIMPLIFICATION															
	$a_7\ a_6$	$a_5\ a_7$	$a_5\ a_6$	$a_3\ a_7$	$a_2\ a_7$	$a_1\ a_7$	$a_0\ a_7$	$a_0\ a_6$	$a_0\ a_5$	$a_3\ a_1$	$a_0\ a_3$	$a_0\ a_1$	a_2a_0	0	$a_0\ a_0$
	$a_7\ a_7$		$a_4\ a_7$	$a_4\ a_6$	$a_3\ a_6$	$a_2\ a_6$	$a_1\ a_6$	$a_1\ a_5$	$a_1\ a_4$	$a_4\ a_0$	$a_1\ a_2$	$a_1\ a_1$			
			$a_6\ a_6$		$a_4\ a_5$	$a_3\ a_5$	$a_2\ a_5$	$a_2\ a_4$	$a_2\ a_3$		$a_2\ a_2$				
					$a_5\ a_5$		$a_3\ a_4$		$a_3\ a_3$						
							$a_4\ a_4$								
IN TERMS OF PARTIAL PRODUCT NAME															
	P63	P48	P47	P32	P24	P58	P8	P7	P6	P26	P25	P17	P2	0	P1
	P64		P40	P53	P52	P51	P15	P42	P13	P33	P18		P10		
			P55		P38	P30	P22	P35	P27		P19				
					P46		P36		P28						
							P37								
OUTPUT BITS															
Z_{15}	Z_{14}	Z_{13}	Z_{12}	Z_{11}	Z_{10}	Z_9	Z_8	Z_7	Z_6	Z_5	Z_4	Z_3	Z_2	Z_1	Z_0

The addition of the partial products is done by two different means. The first is done by half adders and full adders and the second is by the use of 8-bit Ripple Carry Adders.

Refer A29 to A34 sections for VHDL code and test bench.

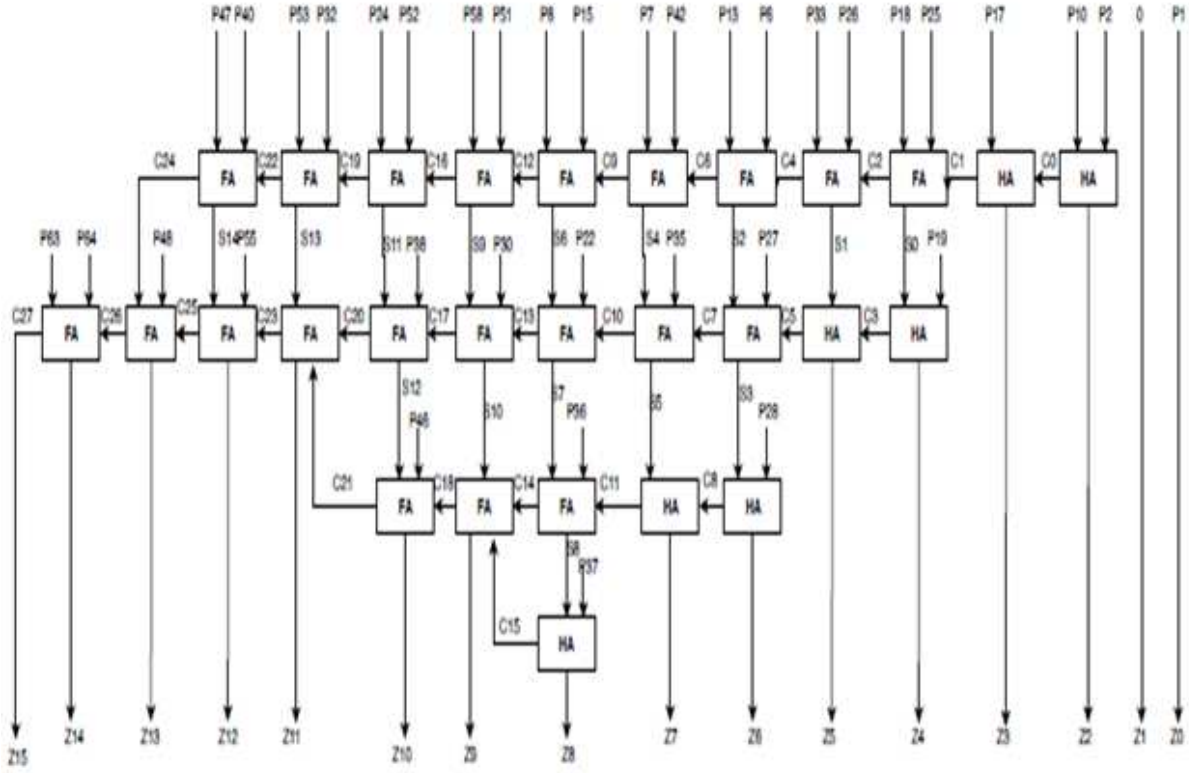


Figure 36. Addition of Partial Products using Half Adders and Full Adders.

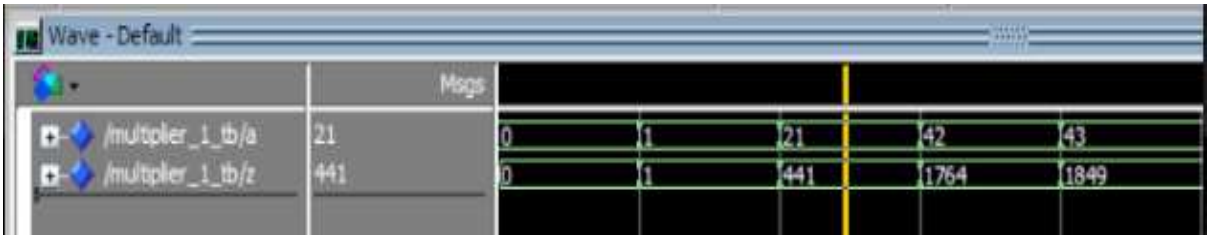


Figure 37. Simulation Results of multiplication using Half Adders and Full Adders.

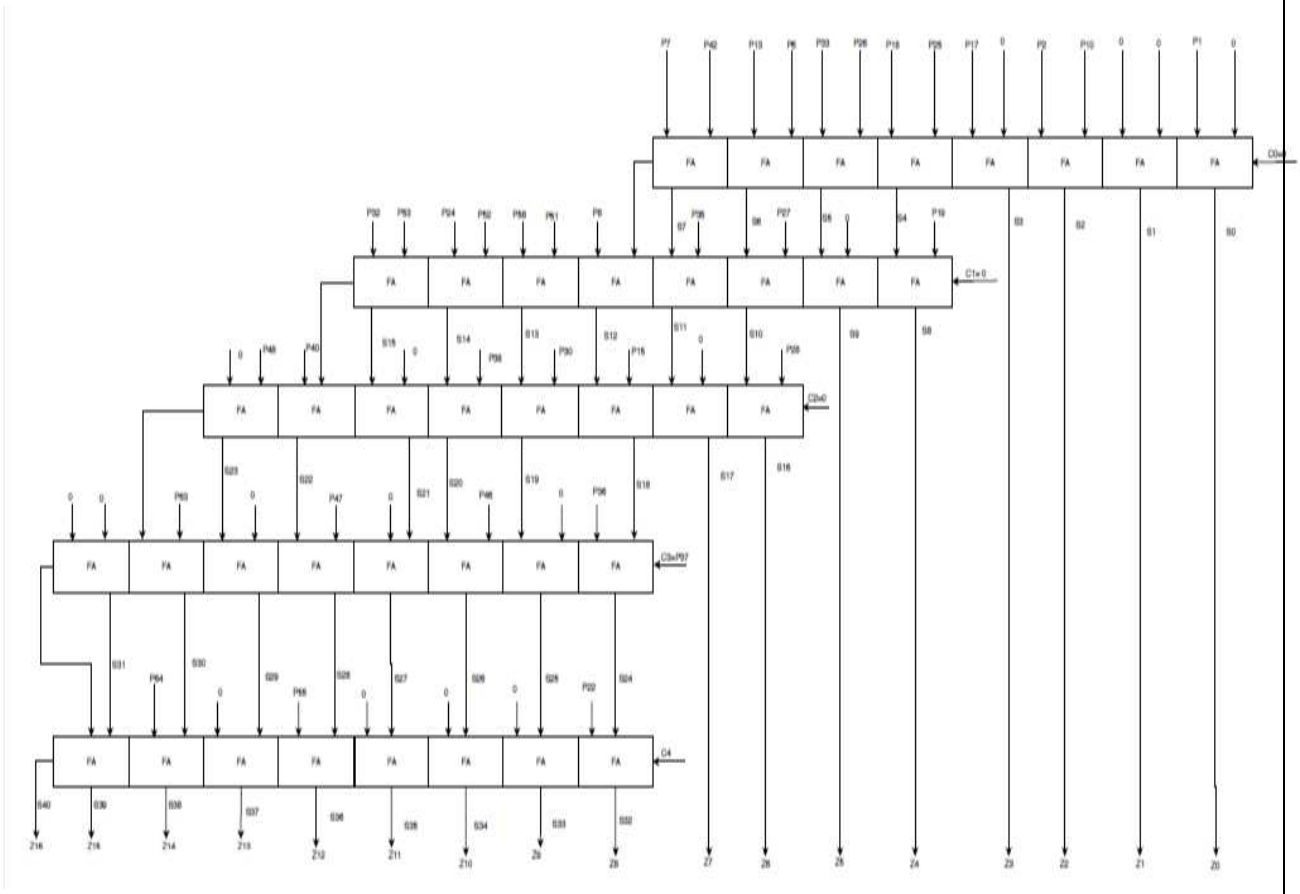


Figure 38. Addition of Partial Products using Ripple Carry Adders.

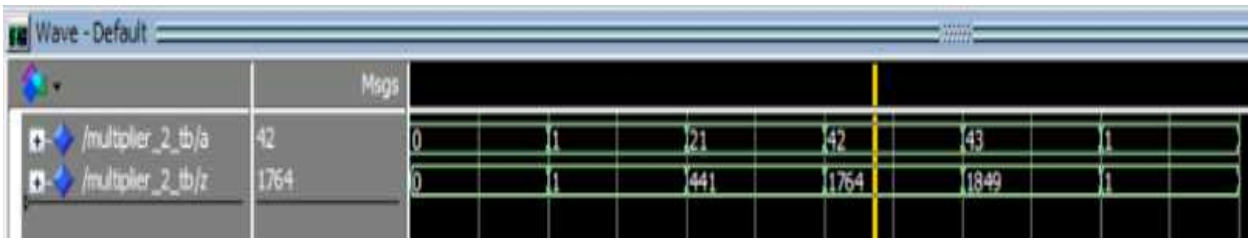


Figure 39. Simulation Results of multiplication using Ripple Carry Adders.

4.3.4. Perform -1 operation:

The output of the multiplier is then fed to ripple carry adder to perform -1 operation.
Steps followed are as follows,

1. Take 2's complement of 1 which is "1111111111111111".
2. Input multiplier output and 2's complement of 1 as inputs to Ripple carry adder.

Refer section A35 and A36 for VHDL code and test bench.

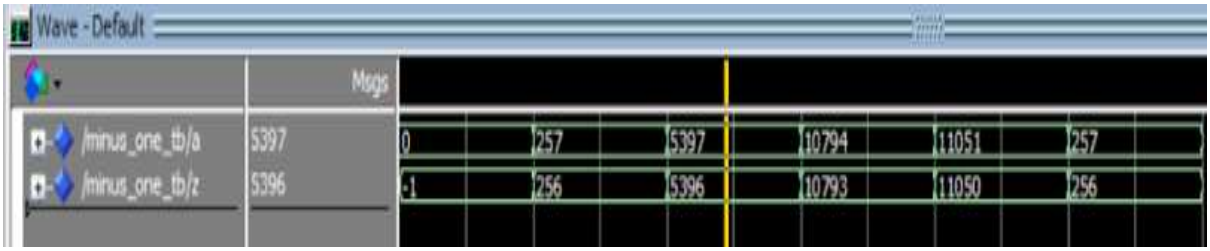


Figure 40. Simulation Result of Minus one operation.

4.4. Use of 16-bit register to store the output:

After performing $\mathbf{A^2 - 1}$, the 16-bit output is fed to the registers which introduce a delay of 1 clock cycle. Output is obtained at the raising edge of the clock when load = 1. Once the operation is completed End Flag is set to 1.

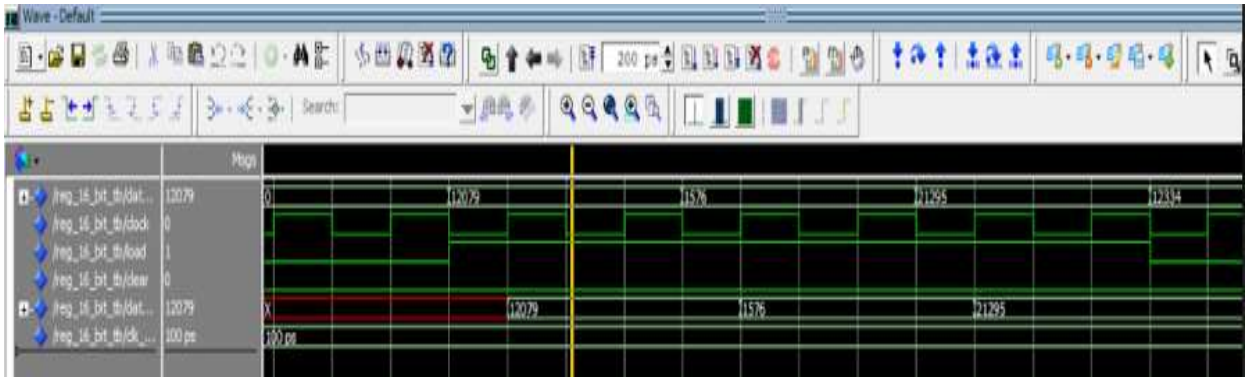


Figure 41. Simulation Result of 16-Bit Register operation.

5. EXPERIMENT RESULT

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

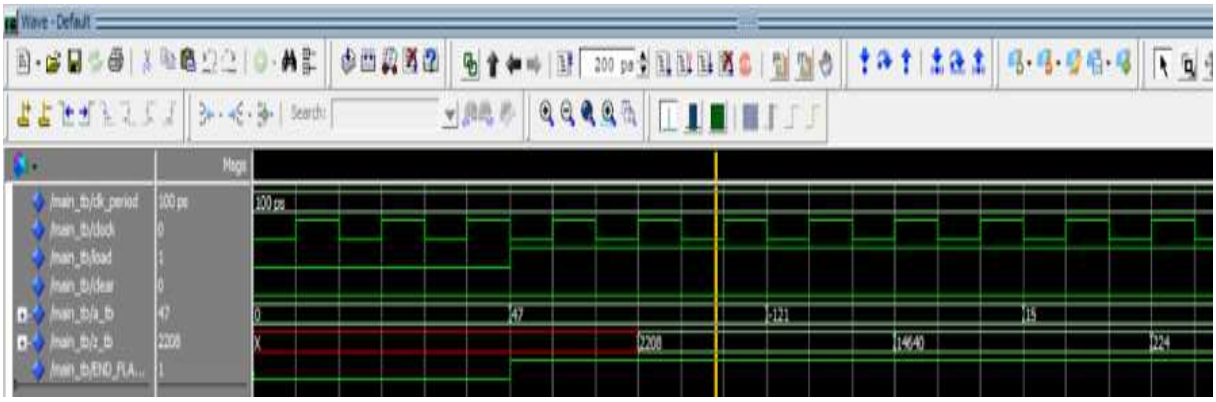


Figure 42. Simulation Result of $\mathbf{A^2 - 1}$ using Full adders and half adders.

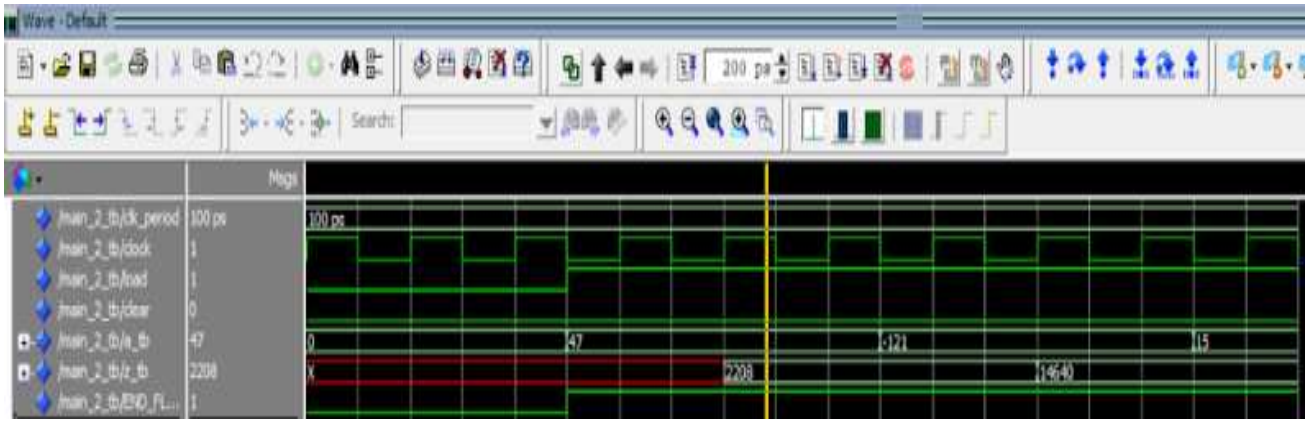


Figure 43. Simulation Result of $\mathbf{A^2 - 1}$ using Ripple Carry Adders.

6. CONCLUSION

The main aim of this project is to design an arithmetic unit that performs the operation (A^2-1) where A is a signed 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 two different approaches. One by using Full adders and half adders and the other is by using 8-bit ripple carry adders. We used tools like precision and Xilinx ISE to get the area and delay of the two different designs. The report on area and delay of the two 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 full adders and half adders for the addition of partial products proves to be efficient than the use of ripple carry adders.

We have not employed Carry skip, Carry select or Manchester carry adders because in our case the carry out of first ripple carry adder does not act as the carry in of second ripple carry adder. Instead the carry out of the first ripple carry adder acts as the input to one of the full adder block present within the second ripple carry adder. So it won't be efficient to use carry skip, carry select or Manchester carry adders in this case.

Through this report we conclude that the arithmetic unit to perform (A^2-1) was successfully implemented using two different approaches and both the approaches were successfully tested with the help of test bench.

7. REFERENCES

1. Users.encs.concordia.ca. (2018). *Lecture Notes and Slides:*. [online] Available at: https://users.encs.concordia.ca/~asim/COEN_6501/Lecture_Notes/Lecture_Notes.htm [Accessed 19 Nov. 2018].

2. Encs.concordia.ca. (2018). [online] Available at: http://www.encs.concordia.ca/helpdesk/resource/manuals_tutorials/tutorial.pdf [Accessed 19 Nov. 2018].

3. Users.encs.concordia.ca. (2018). *http://public*. [online] Available at: https://users.encs.concordia.ca/~asim/COEN_6501/useful_links.htm [Accessed 19 Nov. 2018].

4. En.wikipedia.org. (2018). *Logic gate*. [online] Available at: https://en.wikipedia.org/wiki/Logic_gate [Accessed 19 Nov. 2018].

5. Ece.uvic.ca. (2018). [online] Available at: http://www.ece.uvic.ca/~fayez/courses/ceng465/lab_465/project1/adders.pdf [Accessed 19 Nov. 2018].

6. Irdindia.in. (2018). [online] Available at: http://www.irdindia.in/journal_ijeecs/pdf/vol2_iss8/7.pdf [Accessed 19 Nov. 2018].

7. Vaidya, S. and Dandekar, D. (2010). Delay-Power Performance Comparison of Multipliers in VLSI Circuit Design. *International journal of Computer Networks & Communications*, 2(4), pp.47-56.

8. Circuitdigest.com. (2018). *D Flip-Flop Circuit Diagram: Working & Truth Table Explained*. [online] Available at: https://circuitdigest.com/electronic-circuits/d-flip-flops [Accessed 19 Nov. 2018].

9. Electrical4u.com. (2018). *Parallel in Parallel Out (PIPO) Shift Register*. [online] Available at: https://www.electrical4u.com/parallel-in-parallel-out-pipo-shift-register/ [Accessed 19 Nov. 2018].

APPENDICES

Appendix I - VHDL Codes and Test Benches

A1. AND GATE- VHDL CODE

```
-- Project      : COEN6501
-- File Name    : and_gate.vhd
-- Author       : Amulya Prabhakar
-- Date                 : 29- October- 2018
-- Description  : The function of this component is to perform AND operation on two
binary                  inputs.

-- Declare library files:
library IEEE;
use ieee.std_logic_1164.all;

-- Entity Declaration:
entity and_gate is
port( A, B : in std_logic;
C : out std_logic);
end and_gate;

-- Architecture Implementation:
architecture andg of and_gate is
begin
C <= A and B;
end;

-- end of and_gate.vhd
```

A2. AND GATE- TEST BENCH

```
-- Project      : COEN6501
-- File Name    : and_gate_tb.vhd
-- Author       : Sibi Ravichandran
-- Date                 : 02- November- 2018
-- Description : The function of this test bench is to test the functionality of AND Gate.

-- Declare library files:
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Entity Declaration:
entity and_tb is
-- No port is required for a test bench
end and_tb;

-- Architecture Declaration
architecture rtl of and_tb is
```

```
-- Component declaration for AND Gate:
Component and_gate
port( A, B : in std_logic;
C : out std_logic);
end Component ;

--inputs
signal a: std_logic:= '0';
signal b: std_logic:= '0';
--outputs
signal c : std_logic;

begin
uut: and_gate PORT MAP(a=>A,b=>B,c=>C);

--Stimulus Process
stim_proc:process
begin
wait for 100 ps;
a<='1';
b<='0';

wait for 100 ps;
a<='0';
b<='1';

wait for 100 ps;
a<='0';
b<='0';

wait for 100 ps;
a<='1';
b<='1';

end process;
end rtl;

-- end of and_gate_tb.vhd
```

A3. OR GATE- VHDL CODE

```
-- Project      : COEN6501
-- File Name    : or_gate.vhd
-- Author       : Amulya Prabhakar
-- Date                 : 29- October- 2018
 -- Description        : The function of this component is to perform OR operation on
two binary inputs.

-- Declare library files:
```

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

-- Entity Declaration:
entity or_gate is
port( A, B : in std_logic;
C : out std_logic);
end or_gate;

-- Architecture Implementation:
architecture org of or_gate is
begin
C <= A or B;
end;

-- end of or_gate.vhd
```

A4. OR GATE- TEST BENCH

```
-- Project      : COEN6501
-- File Name    : or_gate_tb.vhd
-- Author       : Sibi Ravichandran
-- Date                 : 02- November- 2018
-- Description : The function of this test bench is to test the functionality of OR Gate.

-- Declare library files:
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Entity Declaration:
entity or_tb is
-- No port is required for a test bench
end or_tb;

-- Architecture Declaration
architecture rtl of or_tb is

-- Component declaration for OR Gate:
Component or_gate
port( A, B : in std_logic;
C : out std_logic);
end Component ;

--inputs
signal a: std_logic:= '0';
signal b: std_logic:= '0';

--outputs
signal c : std_logic;
```

```
begin
uut: or_gate PORT MAP(a=>A,b=>B,c=>C);

--Stimulus Process
stim_proc:process
begin
wait for 100 ps;
a<='1';
b<='0';

wait for 100 ps;
a<='0';
b<='1';

wait for 100 ps;
a<='0';
b<='0';

wait for 100 ps;
a<='1';
b<='1';

end process;
end rtl;

-- end of or_gate_tb.vhd
```

A5. NOT GATE- VHDL CODE

```
-- Project      : COEN6501
-- File Name    : not_gate.vhd
-- Author       : Amulya Prabhakar
-- Date                 : 29- October- 2018
-- Description  : The function of this component is to perform NOT operation on a
binary input.

-- Declare library files:
library IEEE;
use ieee.std_logic_1164.all;

-- Entity Declaration:
entity not_gate is
port( A : in std_logic;
C : out std_logic);
end not_gate;

-- Architecture Implementation:
architecture notg of not_gate is
begin
C <= not(A);
```

```
end;

-- end of not_gate.vhd
```

A6. NOT GATE- TEST BENCH

```
-- Project      : COEN6501
-- File Name    : not_gate_tb.vhd
-- Author       : Sibi Ravichandran
-- Date                 : 02- November- 2018
-- Description : The function of this test bench is to test the functionality of NOT Gate.

-- Declare library files:
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Entity Declaration:
entity not_tb is
-- No port is required for a test bench
end not_tb;

-- Architecture Declaration
architecture rtl of not_tb is

-- Component declaration for NOT Gate:
Component not_gate
port( A : in std_logic;
C : out std_logic);
end Component ;

--inputs
signal a: std_logic:= '0';

--outputs
signal c : std_logic;

begin
uut: not_gate PORT MAP(a=>A,c=>C);

--Stimulus Process
stim_proc:process
begin
wait for 100 ps;
a<='1';

wait for 100 ps;
a<='0';

end process;
end rtl;
```

```
-- end of not_gate_tb.vhd
```

A7. NAND GATE- VHDL CODE

```
-- Project      : COEN6501
-- File Name    : nand_gate.vhd
-- Author       : Amulya Prabhakar
-- Date                 : 29- October- 2018
-- Description  : The function of this component is to perform NAND operation on two
  binary inputs.

-- Declare library files:
library IEEE;
use ieee.std_logic_1164.all;

-- Entity Declaration:
entity nand_gate is
port( A, B : in std_logic;
C : out std_logic);
end nand_gate;

-- Architecture Implementation:
architecture nandg of nand_gate is
begin
C <= A nand B;
end;

-- end of nand_gate.vhd
```

A8. NAND GATE- TEST BENCH

```
-- Project      : COEN6501
-- File Name    : nand_gate_tb.vhd
-- Author       : Sibi Ravichandran
-- Date                 : 02- November- 2018
-- Description : The function of this test bench is to test the functionality of NAND
Gate.

-- Declare library files:
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Entity Declaration:
entity nand_tb is
-- No port is required for a test bench
end nand_tb;

-- Architecture Declaration
```

```
architecture rtl of nand_tb is


-- Component declaration for NAND Gate:
Component nand_gate
port( A, B : in std_logic;
C : out std_logic);
end Component ;

--inputs
signal a: std_logic:= '0';
signal b: std_logic:= '0';

--outputs
signal c : std_logic;

begin
uut: nand_gate PORT MAP(a=>A,b=>B,c=>C);

--Stimulus Process
stim_proc:process
begin
wait for 100 ps;
a<='1';
b<='0';

wait for 100 ps;
a<='0';
b<='1';

wait for 100 ps;
a<='0';
b<='0';

wait for 100 ps;
a<='1';
b<='1';

end process;
end rtl;

-- end of nand_gate_tb.vhd
```

A9. NOR GATE- VHDL CODE

```
-- Project      : COEN6501
-- File Name    : nor_gate.vhd
-- Author       : Amulya Prabhakar
-- Date                 : 29- October- 2018
```

```
-- Description : The function of this component is to perform NOR operation on two
binary inputs.

-- Declare library files:
library IEEE;
use ieee.std_logic_1164.all;

-- Entity Declaration:
entity nor_gate is
port( A, B : in std_logic;
C : out std_logic);
end nor_gate;

-- Architecture Implementation:
architecture norg of nor_gate is
begin
C <= A nor B;
end;

-- end of nor_gate.vhd
```

A10. NOR GATE- TEST BENCH

```
-- Project      : COEN6501
-- File Name    : nor_gate_tb.vhd
-- Author       : Sibi Ravichandran
-- Date                 : 02- November- 2018
-- Description : The purpose of this test bench is to test the functionality of NOR Gate.

-- Declare library files:
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Entity Declaration:
entity nor_tb is
-- No port is required for a test bench
end nor_tb;

-- Architecture Declaration
architecture rtl of nor_tb is

-- Component declaration for NOR Gate:
Component nor_gate
port( A, B : in std_logic;
C : out std_logic);
end Component ;

--inputs
signal a: std_logic:= '0';
signal b: std_logic:= '0';
```

```
--outputs
signal c : std_logic;

begin
uut: nor_gate PORT MAP(a=>A,b=>B,c=>C);

--Stimulus Process
stim_proc:process
begin
wait for 100 ps;
a<='1';
b<='0';

wait for 100 ps;
a<='0';
b<='1';

wait for 100 ps;
a<='0';
b<='0';

wait for 100 ps;
a<='1';
b<='1';

end process;
end rtl;

-- end of nor_gate_tb.vhd
```

A11. EX-OR GATE- VHDL CODE

```
-- Project      : COEN6501
-- File Name    : exor_gate.vhd
-- Author       : Amulya Prabhakar
-- Date                 : 29- October- 2018
-- Description  : The function of this component is to perform EX-OR operation on two binary inputs.

-- Declare library files:
library IEEE;
use ieee.std_logic_1164.all;

-- Entity Declaration:
entity exor_gate is
port( A, B : in std_logic;
C : out std_logic);
end exor_gate;
```

```
-- Architecture Implementation:
architecture exorg of exor_gate is
begin
C <= A xor B;
end;

-- end of exor_gate.vhd
```

A12. EX-OR GATE- TEST BENCH

```
-- Project      : COEN6501
-- File Name    : xor_gate_tb.vhd
-- Author       : Sibi Ravichandran
-- Date                 : 02- November- 2018
-- Description : The function of this test bench is to test the functionality of the XOR
Gate.

-- Declare library files:
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Entity Declaration:
entity xor_tb is
-- No port is required for a test bench
end xor_tb;

-- Architecture Declaration
architecture rtl of xor_tb is

-- Component declaration for EX-OR Gate:
Component exor_gate
port( A, B : in std_logic;
C : out std_logic);
end Component ;

--inputs
signal a: std_logic:= '0';
signal b: std_logic:= '0';

--outputs
signal c : std_logic;

begin
uut: exor_gate PORT MAP(a=>A,b=>B,c=>C);

--Stimulus Process
stim_proc:process
begin
wait for 100 ps;
a<='1';
b<='0';
```

```
wait for 100 ps;
a<='0';
b<='1';

wait for 100 ps;
a<='0';
b<='0';

wait for 100 ps;
a<='1';
b<='1';

end process;
end rtl;

-- end of xor_gate_tb.vhd
```

A13. EX-NOR GATE- VHDL CODE

```
-- Project      : COEN6501
-- File Name    : exnor_gate.vhd
-- Author       : Amulya Prabhakar
-- Date                 : 29- October- 2018
-- Description  : The function of this component is to perform EX-NOR operation on two binary inputs.

-- Declare library files:
library IEEE;
use ieee.std_logic_1164.all;

-- Entity Declaration:
entity xnor_gate is
port( A, B : in std_logic;
C : out std_logic);
end xnor_gate;

-- Architecture Implementation:
architecture xnorg of xnor_gate is
begin
C <= A xnor B;
end;

-- end of exnor_gate.vhd
```

A14. EX-NOR GATE- TEST BENCH

```
-- Project      : COEN6501
-- File Name    : exnor_gate_tb.vhd
-- Author       : Sibi Ravichandran
-- Date                 : 02- November- 2018
```

```
-- Description : The function of this test bench is to test the functionality of EX-NOR Gate.

-- Declare library files:
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Entity Declaration:
entity exnor_tb is
-- No port is required for a testbench
end exnor_tb;

-- Architecture Declaration
architecture rtl of exnor_tb is

-- Component declaration for EX-NOR Gate:
Component xnor_gate
port( A, B : in std_logic;
C : out std_logic);
end Component ;

--inputs
signal a: std_logic:= '0';
signal b: std_logic:= '0';

--outputs
signal c : std_logic;

begin
uut: xnor_gate PORT MAP(a=>A,b=>B,c=>C);

--Stimulus Process
stim_proc:process
begin
wait for 100 ps;
a<='1';
b<='0';

wait for 100 ps;
a<='0';
b<='1';

wait for 100 ps;
a<='0';
b<='0';

wait for 100 ps;
a<='1';
b<='1';
```

```
end process;
end rtl;

-- end of exnor_gate_tb.vhd
```

A15. HALF ADDER- VHDL CODE

```
-- Project      : COEN6501
-- File Name    : half_adder.vhd
-- Author       : Amulya Prabhakar
-- Date                 : 30- October- 2018
-- Description : The function of this component is to perform addition on two binary inputs and provide the sum and carry output. This component employs an EX-OR gate and an AND gate to perform this operation.

-- Declare library files:
library IEEE;
use ieee.std_logic_1164.all;

-- Entity Declaration:
entity half_adder is
        port( A, B : in std_logic;
                sum, carry : out std_logic);
end half_adder;

-- Architecture Implementation:
architecture ha of half_adder is

-- Component Declaration of XOR gate:
component exor_gate
        port( A, B : in std_logic;
                C : out std_logic);
end component;

-- Component Declaration of AND gate:
component and_gate
        port( A, B : in std_logic;
                C : out std_logic);
end component;

begin
-- SUM = A XOR B and CARRY = A AND B:
EXOR: exor_gate port map (A,B,sum);
ANDG: and_gate port map (A,B,carry);
end;

-- end of half_adder.vhd
```

A16. HALF ADDER- TEST BENCH

```
-- Project      : COEN6501
-- File Name    : half_adder_tb.vhd
-- Author       : Sibi Ravichandran
-- Date                 : 02- November- 2018
-- Description : The function of this test bench is to test the functionality of Half Adder.
-- Declare library files:
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Entity Declaration:
entity half_adder_tb is
        -- No port declarations for test bench
end half_adder_tb;

-- Architecture Implementation:
architecture rtl of half_adder_tb is

-- Component declaration for half adder:
Component half_adder
        port( A, B : in std_logic;
                sum, carry : out std_logic);
end Component;

--inputs
signal a: std_logic:= '0';
signal b: std_logic:= '0';

--outputs
signal sum : std_logic;
signal carry : std_logic;

begin
        uut: half_adder PORT MAP(a=>A,b=>B,sum=>sum,carry=>carry);

        --Stimulus Process
        stim_proc:process
        begin
                wait for 100 ps;
                a<='1';
                b<='0';

                wait for 100 ps;
                a<='0';
                b<='1';

                wait for 100 ps;
                a<='0';
                b<='0';
```

```
                wait for 100 ps;
                a<='1';
                b<='1';

        end process;
end rtl;

-- end of half_adder_tb.vhd
```

A17. FULL ADDER- VHDL CODE

```
-- Project      : COEN6501
-- File Name    : full_adder.vhd
-- Author       : Amulya Prabhakar
-- Date                 : 30- October- 2018
-- Description : The function of this component is to perform addition on three binary
inputs and provide the sum and carry output. This component employs two half adders
and one OR gate to perform this operation.

-- Declare library files:
library IEEE;
use ieee.std_logic_1164.all;

-- Entity Declaration:
entity full_adder is
        port( A, B, carry_in : in std_logic;
                sum, carry_out : out std_logic);
end full_adder;

-- Architecture Implementation:
architecture fa of full_adder is

-- Component Declaration of half_adder gate:
Component half_adder
        port( A, B : in std_logic;
                sum, carry : out std_logic);
end Component;

-- Component Declaration of OR gate:
component or_gate
        port( A, B : in std_logic;
                C : out std_logic);
end component;

-- Signal declaration of internal variables:
signal carry_propagate: std_logic;
signal carry_generate_1: std_logic;
signal carry_generate_2: std_logic;
```

```
begin
PG: half_adder port map (A,B,carry_propagate,carry_generate_1);
SC: half_adder port map (carry_propagate, carry_in, sum, carry_generate_2);
ORG: or_gate port map (carry_generate_1,carry_generate_2, carry_out);
end;
-- end of full_adder.vhd
```

A18. FULL ADDER- TEST BENCH

```
-- Project      : COEN6501
-- File Name    : full_adder_tb.vhd
-- Author       : Sibi Ravichandran
-- Date                 : 02- November- 2018
-- Description : The function of this test bench is to test the functionality of Full Adder.

-- Declare library files:
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Entity Declaration:
entity full_adder_tb is
-- No port declarations for test bench
end full_adder_tb;

-- Architecture Implementation:
architecture rtl of full_adder_tb is

-- Component declaration for full adder:
Component full_adder
port( A, B, carry_in : in std_logic;
sum, carry_out : out std_logic);
end Component;

--inputs
signal a: std_logic:= '0';
signal b: std_logic:= '0';
signal carry_in: std_logic:= '0';

--outputs
signal sum : std_logic;
signal carry_out : std_logic;

begin
uut: full_adderPORTMAP(a=>A,b=>B,carry_in=>carry_in,sum=>sum,carry_out=>carry_out);

--Stimulus Process
stim_proc:process
        begin
                wait for 100 ps;
                a<='0';
```

```
                b<='0';
                carry_in<='0';

                wait for 100 ps;
                a<='0';
                b<='0';
                carry_in<='1';

                wait for 100 ps;
                a<='0';
                b<='1';
                carry_in<='0';

                wait for 100 ps;
                a<='0';
                b<='1';
                carry_in<='1';

                wait for 100 ps;
                a<='1';
                b<='0';
                carry_in<='0';

                wait for 100 ps;
                a<='1';
                b<='0';
                carry_in<='1';

                wait for 100 ps;
                a<='1';
                b<='1';
                carry_in<='0';

                wait for 100 ps;
                a<='1';
                b<='1';
                carry_in<='1';

        end process;
end rtl;

-- end of full_adder_tb.vhd
```

A19. 4- BIT RIPPLE CARRY ADDER- VHDL CODE

```
-- Project      : COEN6501
-- File Name    : ripple_carry_4bit.vhd
-- Author       : Amulya Prabhakar
-- Date                 : 30- October- 2018
```

```
-- Description : The function of this component is to perform parallel addition on two
4-bit binary inputs. This component employs four full adders to perform the addition.

-- Declare library files:
library IEEE;
use ieee.std_logic_1164.all;

-- Entity Declaration:
entity ripple_carry_4bit is
        port( A, B  : in std_logic_vector (3 downto 0);
                carry_in        : in std_logic;
                sum             : out std_logic_vector (3 downto 0);
                carry_out : out std_logic);
end ripple_carry_4bit;

-- Architecture Implementation:
architecture rca4 of ripple_carry_4bit is

-- Component Declaration of full_adder:
component full_adder
        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 (4 downto 0);

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

FA1: full_adder port map (A(0), B(0), C(0), sum(0), C(1));
FA2: full_adder port map (A(1), B(1), C(1), sum(1), C(2));
FA3: full_adder port map (A(2), B(2), C(2), sum(2), C(3));
FA4: full_adder port map (A(3), B(3), C(3), sum(3), C(4));

-- Assign the value of C(4) to Carry out:
carry_out <= C(4);
end;

-- end of ripple_carry_4bit.vhd
```

A20. 4- BIT RIPPLE CARRY ADDER- TEST BENCH

```
-- Project      : COEN6501
-- File Name    : ripple_carry_4_bit_tb.vhd
-- Author       : Sibi Ravichandran
-- Date                 : 02- November- 2018
-- Description : The function of this test bench is to test the functionality of 4-bit Ripple
Carry Adder.
```

```
-- Declare library files:
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Entity Declaration:
entity ripple_carry_4_bit_tb is
-- No port declarations for test bench
end ripple_carry_4_bit_tb;

-- Architecture Implementation:
architecture rtl of ripple_carry_4_bit_tb is

-- Component declaration for 4-Bit Ripple Carry Adder:
Component ripple_carry_4bit
        port( A, B       : in std_logic_vector (3 downto 0);
                carry_in       : in std_logic;
                sum            : out std_logic_vector (3 downto 0);
                carry_out : out std_logic);
end Component;

--inputs
signal a: std_logic_vector(3 downto 0) := "0000";
signal b: std_logic_vector(3 downto 0) := "0000";
signal carry_in: std_logic:= '0';
--outputs
signal sum: std_logic_vector(3 downto 0);
signal carry_out : std_logic;

begin
uut:
ripple_carry_4bitPORTMAP(a=>A,b=>B,carry_in=>carry_in,sum=>sum,carry_out=>c
arry_out);

--Stimulus Process
        stim_proc:process
        begin
                wait for 100 ps;
                a<="0001";
                b<="0001";
                carry_in<='0';

                wait for 100 ps;
                a<="0010";
                b<="0011";
                carry_in<='1';

                wait for 100 ps;
                a<="0100";
                b<="0011";
```

```
                carry_in<='0';

                wait for 100 ps;
                a<="0111";
                b<="1111";
                carry_in<='0';

                wait for 100 ps;
                a<="1111";
                b<="1111";
                carry_in<='1';
        end process;
end rtl;
-- end of ripple_carry_4bit.vhd
```

A21. 8- BIT RIPPLE CARRY ADDER- VHDL CODE

```
-- Project      : COEN6501
-- File Name    : ripple_carry_8bit.vhd
-- Author       : Amulya Prabhakar
-- Date                 : 30- October- 2018
-- Description : The function of this component is to perform parallel addition on two
8-bit binary inputs. This component employs two 4-bit Ripple Carry Adders to perform
the operation.

-- Declare library files:
library IEEE;
use ieee.std_logic_1164.all;

-- Entity Declaration:
entity ripple_carry_8bit is
        port( A, B      : in std_logic_vector (7 downto 0);
                carry_in        : in std_logic;
                sum             : out std_logic_vector (7 downto 0);
                carry_out : out std_logic);
end ripple_carry_8bit;

-- Architecture Implementation:
architecture rca8 of ripple_carry_8bit is

-- Component Declaration of 4-bit RCA:
component ripple_carry_4bit
        port( A, B      : in std_logic_vector (3 downto 0);
                carry_in        : in std_logic;
                sum             : out std_logic_vector (3 downto 0);
                carry_out : out std_logic);
end component;

-- Signal declaration of internal variables:
signal C: std_logic_vector (2 downto 0);
```

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

RCA1: ripple_carry_4bit port map (A (3 downto 0), B(3 downto 0), C(0), sum (3 downto 0), C(1));
RCA2: ripple_carry_4bit port map (A (7 downto 4), B(7 downto 4), C(1), sum (7 downto 4), C(2));

-- Assign the value of C(2) to Carry out:
carry_out <= C(2);
end;

-- end of ripple_carry_8bit.vhd
```

A22. 8- BIT RIPPLE CARRY ADDER- TEST BENCH

```
-- Project      : COEN6501
-- File Name    : ripple_carry_8_bit_tb.vhd
-- Author       : Sibi Ravichandran
-- Date                 : 02- November- 2018
-- Description  : The function of this test bench is to test the functionality of  8-bit Ripple Carry Adder.

-- Declare library files:
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Entity Declaration:
entity ripple_carry_8_bit_tb is
        -- No port declarations for test bench
end ripple_carry_8_bit_tb;

-- Architecture Implementation:
architecture rtl of ripple_carry_8_bit_tb is

-- Component declaration for 8-Bit Ripple Carry Adder:
Component ripple_carry_8bit
        port( A, B      : in std_logic_vector (7 downto 0);
                carry_in        : in std_logic;
                sum             : out std_logic_vector (7 downto 0);
                carry_out : out std_logic);
end Component;

--inputs
signal a: std_logic_vector(7 downto 0):= "00000000";
signal b: std_logic_vector(7 downto 0):= "00000000";
signal carry_in: std_logic:= '0';

--outputs
```

```
signal sum: std_logic_vector(7 downto 0);
signal carry_out : std_logic;

begin
uut:
ripple_carry_8bit
PORTMAP(a=>A,b=>B,carry_in=>carry_in,sum=>sum,carry_out=>carry_out);

--Stimulus Process
stim_proc:process
        begin
                wait for 100 ps;
                a<="00011100";
                b<="00001001";
                carry_in<='0';

                wait for 100 ps;
                a<="00000010";
                b<="00011001";
                carry_in<='1';

                wait for 100 ps;
                a<="00110100";
                b<="00010101";
                carry_in<='0';

                wait for 100 ps;
                a<="11111111";
                b<="11111111";
                carry_in<='0';

                wait for 100 ps;
                a<="11111111";
                b<="11111111";
                carry_in<='1';

        end process;
end rtl;

-- end of ripple_carry_8_bit_tb.vhd
```

A23. 8- BIT REGISTER- VHDL CODE

```
-- Project      : COEN6501
-- File Name    : reg_8_bit.vhd
-- Author       : Sibi Ravichandran
-- Date                 : 31- October- 2018
-- Description : The function of this component is to load the 8-Bit input into register
when the Clock is high and the load signal is high. It will reset the register when the
value of clear signal is high.
```

```
-- Declare library files:
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Entity Declaration:
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 implementation:
architecture Behavioral of reg_8_bit is
begin
  process (clock)
  begin
    -- Check for positive edge of clock
    if clock'event and clock = '1' then
      -- If the clear signal is high then reset the register:
      if clear = '1' then
        data_out <="00000000";
      -- If the load signal is high then load the value of input to registers:
      elsif load ='1' then
        data_out <=data_in;
      end if;
    end if;
  end process;
end;

-- end of reg_8_bit.vhd
```

A24. 8- BIT REGISTER- TEST BENCH

```
-- Project      : COEN6501
-- File Name    : reg_8_bit_tb.vhd
-- Author       : Amulya Prabhakar
-- Date                 : 03- November- 2018
-- Description : The function of this test bench is to test the functionality of 8-Bit Register.

-- Declare library files:
library IEEE;
use ieee.std_logic_1164.all;

-- Entity Declaration:
entity reg_8_bit_tb is
        -- No port declarations for test bench
end reg_8_bit_tb;
```

```
-- Architecture implementation:
architecture rtl of reg_8_bit_tb is

-- Component declaration for 8-BitRegister:
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;

--inputs
signal data_in: std_logic_vector (7 downto 0):= "00000000";
signal clock: std_logic := '0';
signal load: std_logic  := '0';
signal clear: std_logic := '0';

--outputs
signal data_out: std_logic_vector (7 downto 0);
constant clk_period : time := 100 ps;

begin
uut:
reg_8_bitPORTMAP(data_in=>data_in,clock=>clock,load=>load,clear=>clear,data_ou
t=>data_out);

  -- Process to toggle the value of the clock every 50 ps
  clk_process :process
  begin
       clock <= '0';
       wait for clk_period/2;  --for 0.5 ns signal is '0'.
       clock <= '1';
       wait for clk_period/2;  --for next 0.5 ns signal is '1'.
  end process;

  -- process for testing the register
  stim_proc:process
  begin
                wait for 200 ps;
                data_in<="00101111";
                clear<='0';
                load<='1';

                wait for 200 ps;
                data_in<="00000111";
                clear<='0';
                load<='1';

                wait for 200 ps;
                data_in<="00001111";
```

```
                clear<='0';
                load<='1';

                wait for 200 ps;
                data_in<="00000011";
                clear<='0';
                load<='0';

                wait for 200 ps;
                data_in<="01100111";
                clear<='0';
                load<='1';

                wait for 200 ps;
                data_in<="00000001";
                clear<='1';
                load<='1';

                wait for 200 ps;
                data_in<="00100100";
                clear<='0';
                load<='1';
        end process;
end rtl;
-- end of reg_8_bit_tb.vhd
```

A25. 16- BIT REGISTER- VHDL CODE

```
-- Project      : COEN6501
-- File Name    : reg_16_bit.vhd
-- Author       : Sibi Ravichandran
-- Date                 : 31- October- 2018
-- Description : The function of this component is to load the 16-Bit input into  register when the Clock is high and the load signal is high. It will reset the register when the value of clear signal is high.

-- Declare library files:
library IEEE;
use ieee.std_logic_1164.all;

-- Entity Declaration:
entity reg_16_bit is
        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 reg_16_bit;

-- Architecture implementation:
architecture reg of reg_16_bit is
```

```
begin
  process (clock)
  begin
    -- check for positive edge of clock
    if clock'event and clock = '1' then
      --if the clear signal is high then reset the register:
      if clear = '1' then
        data_out <="0000000000000000";
      --if the load signal is high then load the value of input to registers:
      elsif load ='1' then
        data_out <=data_in;
      end if;
    end if;
  end process;
end;
-- end of reg_16_bit.vhd
```

A26. 16- BIT REGISTER- TEST BENCH

```
-- Project      : COEN6501
-- File Name    : reg_16_bit_tb.vhd
-- Author       : Amulya Prabhakar
-- Date                 : 03- November- 2018
-- Description : The function of this test bench is to test the functionality of 16-Bit Register.


-- Declare library files:
library IEEE;
use ieee.std_logic_1164.all;

-- Entity Declaration:
entity reg_16_bit_tb is
        -- No port declarations for test bench
end reg_16_bit_tb;

-- Architecture implementation:
architecture rtl of reg_16_bit_tb is

-- Component declaration for 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;

--inputs
signal data_in: std_logic_vector (15 downto 0):= "0000000000000000";
signal clock: std_logic := '0';
```

```
    signal load: std_logic  := '0';
    signal clear: std_logic := '0';

    --outputs
    signal data_out: std_logic_vector (15 downto 0);
    constant clk_period : time := 100 ps;

    begin
   uut:
reg_16_bitPORTMAP(data_in=>data_in,clock=>clock,load=>load,clear=>clear,data_out
=>data_out);

     -- Process to toggle the value of the clock every 50 ps
     clk_process :process
     begin
          clock <= '0';
          wait for clk_period/2;  --for 0.5 ns signal is '0'.
          clock <= '1';
          wait for clk_period/2;  --for next 0.5 ns signal is '1'.
     end process;

     -- process for testing the register
     stim_proc:process
     begin

                    wait for 200 ps;
                    data_in<="0010111100101111";
                    clear<='0';
                    load<='1';

                    wait for 200 ps;
                    data_in<="0000011000101000";
                    clear<='0';
                    load<='1';

                    wait for 200 ps;
                    data_in<="0101001100101111";
                    clear<='0';
                    load<='1';

                    wait for 200 ps;
                    data_in<="0011000000101110";
                    clear<='0';
                    load<='0';

                    wait for 200 ps;
                    data_in<="0110011101111111";
                    clear<='0';
                    load<='1';
```

```
                wait for 200 ps;
                data_in<="0010111100000001";
                clear<='1';
                load<='1';

                wait for 200 ps;
                data_in<="0010111100111100";
                clear<='0';
                load<='1';

  end process;
end rtl;

-- end of reg_16_bit_tb.vhd
```

A27. SIGNED TO UNSIGNED- VHDL CODE

```
-- Project      : COEN6501
-- File Name    : sign_to_unsign.vhd
-- Author       : Sibi Ravichandran
-- Date                 : 31- October- 2018
-- Description :  The function of this component is to check the Most Significant Byte of
the input and decide whether it is a negative number or positive number. If it is a
negative number, then the component will convert the negative number into positive
number and pass it as output, else it will pass the input as output without any changes.

-- Declare library files:
library IEEE;
use ieee.std_logic_1164.all;

-- Entity Declaration:
entity sign_2_unsign is
        port( input     : in std_logic_vector (7 downto 0);
                A               : out std_logic_vector (7 downto 0));
end sign_2_unsign;

-- Architecture Implementation:
architecture s2us of sign_2_unsign is

-- Component Declaration of not_gate:
component not_gate
        port( A : in std_logic;
                C : out std_logic);
end component;

-- Component Declaration of ripple_carry_8bit:
Component ripple_carry_8bit
        port( A, B      : in std_logic_vector (7 downto 0);
                carry_in        : in std_logic;
```

```
                sum             : out std_logic_vector (7 downto 0);
                carry_out : out std_logic);
end Component;

-- Signal to hold temporary values:
signal complement: std_logic_vector (7 downto 0);
signal one: std_logic_vector (7 downto 0);
signal output: std_logic_vector (7 downto 0);
signal carry_in: std_logic;
signal carry_out: std_logic;

begin

-- Taking 1's complement:
complement <= NOT (input);

-- Assign the signal one with the value "00000001":
one <= "00000001";

-- Assign the signal carry_in with the value '0':
carry_in <='0';

-- Convert 1's complement number to 2's complement:
RCA8: ripple_carry_8bit port map (complement (7 downto 0),one (7 downto 0),
carry_in, output (7 downto 0), carry_out);

-- Pass the output value depending on the MSB of the input:
process (input,output)
begin
        if input(7)='0' then
                A <=input;
        elsif input(7)='1' then
          A <=output;
        end if;
end process;
end;

-- end of sign_to_unsign.vhd
```

A28. SIGNED TO UNSIGNED- TEST BENCH

```
-- Project      : COEN6501
-- File Name    : signed_to_unsigned_tb.vhd
-- Author       : Amulya Prabhakar
-- Date                 : 03- November- 2018
-- Description : The function of this test bench is to test the functionality of component
used to convert the 8-bit signed binary number to unsigned number.

-- Declare library files:
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;

-- Entity Declaration:
entity sign_to_unsign_tb is
        -- No port declarations for test bench
end sign_to_unsign_tb;

-- Architecture Implementation:
architecture rtl of sign_to_unsign_tb is

-- Component declaration for sign_to_unsign:
Component sign_2_unsign
        port( input      : in std_logic_vector (7 downto 0);
                A                : out std_logic_vector (7 downto 0));
end Component;

--inputs
signal input: std_logic_vector (7 downto 0):= "00000000";
--outputs
signal A : std_logic_vector (7 downto 0);

begin
        uut: sign_2_unsign PORT MAP(input=>input,A=>A);

        --Stimulus Process
        stim_proc:process
        begin
                wait for 100 ps;
                input<="00111001";

                wait for 100 ps;
                input<="10001001";

                wait for 100 ps;
                input<="00001001";

                wait for 100 ps;
                input<="10000101";

                wait for 100 ps;
                input<="11111111";

                wait for 100 ps;
                input<="00111100";

        end process;
end rtl;

-- end of sign_to_unsign_tb.vhd
```

A29. PARTIAL PRODUCT- VHDL CODE

```
-- Project      : COEN6501
-- File Name    : partial_product.vhd
-- Author       : Sibi Ravichandran
-- Date                 : 30- October- 2018
-- Description : The function of this component is to determine the value of 64 Partial
Products by performing AND Operation with the values of 8-bit input.

-- Declare library files:
library IEEE;
use ieee.std_logic_1164.all;

-- Entity Declaration:
entity partial_product is
        port( A         : in std_logic_vector (7 downto 0);
                P               : out std_logic_vector (64 downto 1));
end partial_product;

-- Architecture Implementation:
architecture pp of partial_product is

-- Component Declaration of and_gate:
Component and_gate
        port( A, B : in std_logic;
                C : out std_logic);
end Component;

begin

P1 : and_gate port map (A(0), A(0), P(1));
P2 : and_gate port map (A(0), A(1), P(2));
P3 : and_gate port map (A(0), A(2), P(3));
P4 : and_gate port map (A(0), A(3), P(4));
P5 : and_gate port map (A(0), A(4), P(5));
P6 : and_gate port map (A(0), A(5), P(6));
P7 : and_gate port map (A(0), A(6), P(7));
P8 : and_gate port map (A(0), A(7), P(8));
P9 : and_gate port map (A(1), A(0), P(9));
P10: and_gate port map (A(1), A(1), P(10));
P11: and_gate port map (A(1), A(2), P(11));
P12: and_gate port map (A(1), A(3), P(12));
P13: and_gate port map (A(1), A(4), P(13));
P14: and_gate port map (A(1), A(5), P(14));
P15: and_gate port map (A(1), A(6), P(15));
P16: and_gate port map (A(1), A(7), P(16));
P17: and_gate port map (A(2), A(0), P(17));
P18: and_gate port map (A(2), A(1), P(18));
P19: and_gate port map (A(2), A(2), P(19));
P20: and_gate port map (A(2), A(3), P(20));
```

```
P21: and_gate port map (A(2), A(4), P(21));
P22: and_gate port map (A(2), A(5), P(22));
P23: and_gate port map (A(2), A(6), P(23));
P24: and_gate port map (A(2), A(7), P(24));
P25: and_gate port map (A(3), A(0), P(25));
P26: and_gate port map (A(3), A(1), P(26));
P27: and_gate port map (A(3), A(2), P(27));
P28: and_gate port map (A(3), A(3), P(28));
P29: and_gate port map (A(3), A(4), P(29));
P30: and_gate port map (A(3), A(5), P(30));
P31: and_gate port map (A(3), A(6), P(31));
P32: and_gate port map (A(3), A(7), P(32));
P33: and_gate port map (A(4), A(0), P(33));
P34: and_gate port map (A(4), A(1), P(34));
P35: and_gate port map (A(4), A(2), P(35));
P36: and_gate port map (A(4), A(3), P(36));
P37: and_gate port map (A(4), A(4), P(37));
P38: and_gate port map (A(4), A(5), P(38));
P39: and_gate port map (A(4), A(6), P(39));
P40: and_gate port map (A(4), A(7), P(40));
P41: and_gate port map (A(5), A(0), P(41));
P42: and_gate port map (A(5), A(1), P(42));
P43: and_gate port map (A(5), A(2), P(43));
P44: and_gate port map (A(5), A(3), P(44));
P45: and_gate port map (A(5), A(4), P(45));
P46: and_gate port map (A(5), A(5), P(46));
P47: and_gate port map (A(5), A(6), P(47));
P48: and_gate port map (A(5), A(7), P(48));
P49: and_gate port map (A(6), A(0), P(49));
P50: and_gate port map (A(6), A(1), P(50));
P51: and_gate port map (A(6), A(2), P(51));
P52: and_gate port map (A(6), A(3), P(52));
P53: and_gate port map (A(6), A(4), P(53));
P54: and_gate port map (A(6), A(5), P(54));
P55: and_gate port map (A(6), A(6), P(55));
P56: and_gate port map (A(6), A(7), P(56));
P57: and_gate port map (A(7), A(0), P(57));
P58: and_gate port map (A(7), A(1), P(58));
P59: and_gate port map (A(7), A(2), P(59));
P60: and_gate port map (A(7), A(3), P(60));
P61: and_gate port map (A(7), A(4), P(61));
P62: and_gate port map (A(7), A(5), P(62));
P63: and_gate port map (A(7), A(6), P(63));
P64: and_gate port map (A(7), A(7), P(64));

end;

-- end of partial_product.vhd
```

A30. PARTIAL PRODUCT- TEST BENCH

```
-- Project      : COEN6501
-- File Name    : partial_product_tb.vhd
-- Author       : Amulya Prabhakar
-- Date                 : 03- November- 2018
-- Description : The function of this test bench is to test the functionality of the component which is used to determine the values of the partial products using AND Gate.

-- Declare library files:
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Entity Declaration:
entity partial_product_tb is
        -- No port declarations for test bench
end partial_product_tb;

-- Architecture Implementation:
architecture rtl of partial_product_tb is

-- Component declaration for partial product:
Component partial_product
        port( A         : in std_logic_vector (7 downto 0);
                P               : out std_logic_vector (64 downto 1));
end Component;

--inputs
signal a: std_logic_vector (7 downto 0):= "00000000";

--outputs
signal p : std_logic_vector(64 downto 1);

begin
        uut: partial_product PORT MAP(a=>A,p=>P);

        --Stimulus Process
        stim_proc:process
        begin
                wait for 100 ps;
                a<="00000001";

                wait for 100 ps;
                a<="00010101";

                wait for 100 ps;
                a<="10101010";

                wait for 100 ps;
```

```
                a<="10101011";

        end process;
end rtl;

-- end of partial_product_tb.vhd
```

A31. MULTIPLIER-1- VHDL CODE

```
-- Project      : COEN6501
-- File Name    : multiplier_1.vhd
-- Author       : Sibi Ravichandran
-- Date                 : 01- November- 2018
-- Description : The function of this component is to perform multiplication on two 8-bit unsigned (positive) binary numbers. This component employs half adders and full adders to perform the addition of Partial Products.

-- Declare library files:
library IEEE;
use ieee.std_logic_1164.all;

-- Entity Declaration:
entity multiplier_1 is
        port( A         : in std_logic_vector (7 downto 0);
              Z         : out std_logic_vector (15 downto 0));
end multiplier_1;

-- Architecture Implementation:
architecture mult of multiplier_1 is

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

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

-- Component Declaration of full_adder:
Component full_adder
        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 (14 downto 0);
signal C: std_logic_vector (27 downto 0);

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

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

-- Z0:
Z(0) <= P(1);

-- Z1:
Z(1) <= '0';

-- Z2:
HA1: half_adder port map (P(2), P(10), Z(2), C(0));

-- Z3:
HA2: half_adder port map (P(17), C(0), Z(3), C(1));

-- Z4:
FA1: full_adder port map (P(18), P(25), C(1), S(0),C(2));
HA3: half_adder port map (S(0), P(19), Z(4), C(3));

-- Z5:
FA2: full_adder port map (P(33), P(26), C(2), S(1),C(4));
HA4: half_adder port map (S(1), C(3), Z(5), C(5));

-- Z6:
FA3: full_adder port map (P(13), P(6), C(4), S(2),C(6));
FA4: full_adder port map (S(2), P(27), C(5), S(3),C(7));
HA5: half_adder port map (S(3), P(28), Z(6), C(8));

-- Z7:
FA5: full_adder port map (P(7), P(42), C(6), S(4),C(9));
FA6: full_adder port map (S(4), P(35), C(7), S(5),C(10));
HA6: half_adder port map (S(5), C(8), Z(7), C(11));

-- Z8:
FA7: full_adder port map (P(8), P(15), C(9), S(6),C(12));
FA8: full_adder port map (S(6), P(22), C(10), S(7),C(13));
FA9: full_adder port map (S(7), P(36), C(11), S(8),C(14));
HA7: half_adder port map (S(8), P(37), Z(8), C(15));

-- Z9:
FA10: full_adder port map (P(58), P(51), C(12), S(9),C(16));
FA11: full_adder port map (S(9), P(30), C(13), S(10),C(17));
FA12: full_adder port map (S(10), C(14), C(15), Z(9),C(18));
```

```
        -- Z10:
        FA13: full_adder port map (P(24), P(52), C(16), S(11),C(19));
        FA14: full_adder port map (S(11), P(38), C(17), S(12),C(20));
        FA15: full_adder port map (S(12), P(46), C(18), Z(10),C(21));

        --Z11:
        FA16: full_adder port map (P(32), P(53), C(19), S(13),C(22));
        FA17: full_adder port map (S(13), C(20), C(21), Z(11),C(23));

        --Z12:
        FA18: full_adder port map (P(40), P(47), C(22), S(14),C(24));
        FA19: full_adder port map (P(55), S(14), C(23), Z(12),C(25));

        --Z13:
        FA20: full_adder port map (P(48), C(24), C(25), Z(13),C(26));

        --Z14:
        FA21: full_adder port map (P(64), P(63), C(26), Z(14),C(27));

        --Z15:
        Z(15) <= C(27);
        end;
-- end of multiplier_1.vhd
```

A32. MULTIPLIER-1- TEST BENCH

```
        -- Project      : COEN6501
        -- File Name    : multiplier_1_tb.vhd
        -- Author       : Amulya Prabhakar
        -- Date                 : 03- November- 2018
        -- Description  : The function of this test bench is to test the functionality of multiplier_1
        which employs full adder and half adder for the addition of the partial products.

        -- Declare library files:
        library IEEE;
        use IEEE.STD_LOGIC_1164.ALL;

        -- Entity Declaration:
        entity multiplier_1_tb is
                -- No port declarations for test bench
        end multiplier_1_tb;

        -- Architecture Implementation:
        architecture rtl of multiplier_1_tb is

        -- Component declaration for multiplier_1:
        Component multiplier_1
                port( A         : in std_logic_vector (7 downto 0);
                        Z               : out std_logic_vector (15 downto 0));
```

```
end component;

--inputs
signal a: std_logic_vector (7 downto 0):= "00000000";

--outputs
signal z : std_logic_vector(15 downto 0);

begin
	uut: multiplier_1 PORT MAP(a=>A,z=>Z);

	--Stimulus Process
	stim_proc:process
	begin
		wait for 100 ps;
		a<="00000001";

		wait for 100 ps;
		a<="00010101";

		wait for 100 ps;
		a<="00101010";

		wait for 100 ps;
		a<="00101011";

	end process;
end rtl;

-- end of multiplier_1_tb.vhd
```

A33. MULTIPLIER-2- VHDL CODE

```
-- Project      : COEN6501
-- File Name    : multiplier_2.vhd
-- Author       : Sibi Ravichandran
-- Date                 : 01- November- 2018
-- Description : The function of this component is to perform multiplication on two 8-bit unsigned (positive) binary numbers. This component employs the 8-bit Ripple Carry Adders to perform the addition of Partial Products.

-- Declare library files:
library IEEE;
use ieee.std_logic_1164.all;

-- Entity Declaration:
entity multiplier_2 is
	port( A		: in std_logic_vector (7 downto 0);
		Z		: out std_logic_vector (15 downto 0));
end multiplier_2;
```

```
-- Architecture Implementation:
architecture mult of multiplier_2 is

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

-- Component Declaration of ripple_carry_8bit adder:
Component ripple_carry_8bit
        port( A, B      : in std_logic_vector (7 downto 0);
                carry_in        : in std_logic;
                sum             : out std_logic_vector (7 downto 0);
                carry_out : out std_logic);
end Component;

-- Signal declarations:
-- Carry in and carry out of RCA-8 bit:
signal C: std_logic_vector (9 downto 0);

-- inputs to RCA:
signal input_1: std_logic_vector (7 downto 0);
signal input_2: std_logic_vector (7 downto 0);
signal input_3: std_logic_vector (7 downto 0);
signal input_4: std_logic_vector (7 downto 0);
signal input_5: std_logic_vector (7 downto 0);
signal input_6: std_logic_vector (7 downto 0);
signal input_7: std_logic_vector (7 downto 0);
signal input_8: std_logic_vector (7 downto 0);
signal input_9: std_logic_vector (7 downto 0);
signal input_10: std_logic_vector (7 downto 0);

-- Outputs of RCA:
signal sum: std_logic_vector (39 downto 0);

-- Outputs of Partial Product component:
signal P: std_logic_vector (64 downto 1);

begin

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

-- Performing addition on partial products using 8-bit Ripple carry adders:

-- Setting the value of inputs to the Ripple Carry Adder (8-bit) -1:
input_1(0) <= '0';
input_1(1) <= '0';
```

```
input_1(2) <= P(10);
input_1(3) <= '0';
input_1(4) <= P(25);
input_1(5) <= P(26);
input_1(6) <= P(6);
input_1(7) <= P(42);

input_2(0) <= P(1);
input_2(1) <= '0';
input_2(2) <= P(2);
input_2(3) <= P(17);
input_2(4) <= P(18);
input_2(5) <= P(33);
input_2(6) <= P(13);
input_2(7) <= P(7);

C(0) <='0';
```

RCA1: ripple_carry_8bit port map (input_1 (7 downto 0), input_2 (7 downto 0), C(0), sum (7 downto 0), C(1));

```
-- Setting the value of inputs to the Ripple Carry Adder (8-bit) -2:
input_3(0) <= P(19);
input_3(1) <= '0';
input_3(2) <= P(27);
input_3(3) <= P(35);
input_3(4) <= C(1);
input_3(5) <= P(58);
input_3(6) <= P(24);
input_3(7) <= P(32);

input_4(0) <= sum(4);
input_4(1) <= sum(5);
input_4(2) <= sum(6);
input_4(3) <= sum(7);
input_4(4) <= P(8);
input_4(5) <= P(51);
input_4(6) <= P(52);
input_4(7) <= P(53);

C(2) <='0';
```

RCA2: ripple_carry_8bit port map (input_3 (7 downto 0), input_4 (7 downto 0), C(2), sum (15 downto 8), C(3));

```
-- Setting the value of inputs to the Ripple Carry Adder (8-bit) -3:
input_5(0) <= sum(10);
input_5(1) <= sum(11);
input_5(2) <= sum(12);
input_5(3) <= sum(13);
```

```
input_5(4) <= sum(14);
input_5(5) <= sum(15);
input_5(6) <= C(3);
input_5(7) <= P(48);

input_6(0) <= P(28);
input_6(1) <= '0';
input_6(2) <= P(15);
input_6(3) <= P(30);
input_6(4) <= P(38);
input_6(5) <= '0';
input_6(6) <= P(40);
input_6(7) <= '0';

C(4) <='0';
```

RCA3: ripple_carry_8bit port map (input_5 (7 downto 0), input_6 (7 downto 0), C(4), s
um (23 downto 16), C(5));

```
-- Setting the value of inputs to the Ripple Carry Adder (8-bit) -4:
input_7(0) <= P(36);
input_7(1) <= '0';
input_7(2) <= P(46);
input_7(3) <= '0';
input_7(4) <= P(47);
input_7(5) <= '0';
input_7(6) <= P(63);
input_7(7) <= '0';

input_8(0) <= sum(18);
input_8(1) <= sum(19);
input_8(2) <= sum(20);
input_8(3) <= sum(21);
input_8(4) <= sum(22);
input_8(5) <= sum(23);
input_8(6) <= C(5);
input_8(7) <= '0';

C(6) <=P(37);
```

RCA4: ripple_carry_8bit port map (input_7 (7 downto 0), input_8 (7 downto 0), C(6),
sum (31 downto 24), C(7));

```
-- Setting the value of inputs to the Ripple Carry Adder (8-bit) -5:
input_9(0) <=  P(22);
input_9(1) <=  '0';
input_9(2) <=  '0';
input_9(3) <=  '0';
input_9(4) <=  P(55);
input_9(5) <=  '0';
```

```
input_9(6) <=  P(64);
input_9(7) <=  C(7);

input_10(0) <= sum(24);
input_10(1) <= sum(25);
input_10(2) <= sum(26);
input_10(3) <= sum(27);
input_10(4) <= sum(28);
input_10(5) <= sum(29);
input_10(6) <= sum(30);
input_10(7) <= sum(31);

C(8) <='0';

RCA5: ripple_carry_8bit port map (input_9 (7 downto 0), input_10 (7 downto 0), C(8),
sum (39 downto 32), C(9));

-- Assign the outputs with the value of sum of RCA:

Z(0)<= sum(0);
Z(1)<= sum(1);
Z(2)<= sum(2);
Z(3)<= sum(3);
Z(4)<= sum(8);
Z(5)<= sum(9);
Z(6)<= sum(16);
Z(7)<= sum(17);
Z(8)<= sum(32);
Z(9)<= sum(33);
Z(10)<=sum(34);
Z(11)<=sum(35);
Z(12)<=sum(36);
Z(13)<=sum(37);
Z(14)<=sum(38);
Z(15)<=sum(39);

end;

-- end of multiplier_2.vhd
```

A34. MULTIPLIER-2- TEST BENCH

```
-- Project      : COEN6501
-- File Name    : multiplier_2_tb.vhd
-- Author       : Amulya Prabhakar
-- Date                 : 03- November- 2018
-- Description  : The function of this test bench is to test the functionality of multiplier_2
which employs 8-Bit Ripple carry adder for the addition of the partial products.

-- Declare library files:
```

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

-- Entity Declaration:
entity multiplier_2_tb is
        -- No port declarations for test bench
end multiplier_2_tb;

-- Architecture Implementation:
architecture rtl of multiplier_2_tb is

-- Component declaration for multiplier_2:
Component multiplier_2
        port( A         : in std_logic_vector (7 downto 0);
              Z         : out std_logic_vector (15 downto 0));
end component;

--inputs
signal a: std_logic_vector (7 downto 0):= "00000000";

--outputs
signal z : std_logic_vector(15 downto 0);

begin
        uut: multiplier_2 PORT MAP(a=>A,z=>Z);

        --Stimulus Process
        stim_proc:process
        begin
                wait for 100 ps;
                a<="00000001";

                wait for 100 ps;
                a<="00010101";

                wait for 100 ps;
                a<="00101010";

                wait for 100 ps;
                a<="00101011";

        end process;
end rtl;

-- end of multiplier_2_tb.vhd
```

A35. MINUS-ONE- VHDL CODE

```
        -- Project      : COEN6501
```

```
-- File Name   : minus_one.vhd
-- Author      : Sibi Ravichandran
-- Date        : 01- November- 2018
-- Description : The function of this component is to subtract one from the 16-Bit input given to the component. This component employs two 8-bit inputs to perform this operation.

-- Declare library files:
library IEEE;
use ieee.std_logic_1164.all;

-- Entity Declaration:
entity minus_one is
        port( A         : in std_logic_vector (15 downto 0);
                Z               : out std_logic_vector (15 downto 0));
end minus_one;

-- Architecture Implementation:
architecture minus of minus_one is

-- Component Declaration of ripple_carry_8bit adder:
Component ripple_carry_8bit
        port( A, B      : in std_logic_vector (7 downto 0);
                carry_in        : in std_logic;
                sum             : out std_logic_vector (7 downto 0);
                carry_out : out std_logic);
end Component;

-- Signal Declarations:
-- inputs to RCA:
signal input_1: std_logic_vector (7 downto 0);
signal input_2: std_logic_vector (7 downto 0);
signal input_3: std_logic_vector (7 downto 0);
signal input_4: std_logic_vector (7 downto 0);

-- Outputs of the 8-bit RCA:
signal C: std_logic_vector (2 downto 0);
signal sum: std_logic_vector (15 downto 0);


begin

-- Setting the value of inputs to the Ripple Carry Adder (8-bit) -1:
input_1(0) <= A(0);
input_1(1) <= A(1);
input_1(2) <= A(2);
input_1(3) <= A(3);
input_1(4) <= A(4);
input_1(5) <= A(5);
input_1(6) <= A(6);
```

```
input_1(7) <= A(7);

input_2 <= "11111111";

-- Setting the value of inputs to the Ripple Carry Adder (8-bit) -2:
input_3(0) <= A(8);
input_3(1) <= A(9);
input_3(2) <= A(10);
input_3(3) <= A(11);
input_3(4) <= A(12);
input_3(5) <= A(13);
input_3(6) <= A(14);
input_3(7) <= A(15);

input_4 <= "11111111";

-- carry in:
C(0) <='0';

RCA1: ripple_carry_8bit port map (input_1 (7 downto 0), input_2 (7 downto 0), C(0),
sum (7 downto 0), C(1));
RCA2: ripple_carry_8bit port map (input_3 (7 downto 0), input_4 (7 downto 0), C(1),
sum (15 downto 8), C(2));

-- Assign the outputs:

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);

end;

-- end of minus_one.vhd
```

A36. MINUS-ONE- TEST BENCH

```
-- Project       : COEN6501
```

```
-- File Name   : minus_one_tb.vhd
-- Author      : Amulya Prabhakar
-- Date        : 03- November- 2018
-- Description : The function of this test bench is to test the functionality of minus one component which employs 8-Bit Ripple carry adder to subtract the 1 from the 16 bit input.

-- Declare library files:
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Entity Declaration:
entity minus_one_tb is
        -- No port declarations for test bench
end minus_one_tb;

-- Architecture Implementation:
architecture rtl of minus_one_tb is

-- Component declaration for minus_one:
Component minus_one
        port( A         : in std_logic_vector (15 downto 0);
              Z         : out std_logic_vector (15 downto 0));
end Component;

--inputs
signal a: std_logic_vector (15 downto 0):= "0000000000000000";

--outputs
signal z : std_logic_vector(15 downto 0);

begin
        uut: minus_one PORT MAP(a=>A,z=>Z);

        --Stimulus Process
        stim_proc:process
        begin
                wait for 100 ps;
                a<="0000000100000001";

                wait for 100 ps;
                a<="0001010100010101";

                wait for 100 ps;
                a<="0010101000101010";

                wait for 100 ps;
                a<="0010101100101011";

        end process;
```

```
end rtl;
-- end of minus_one_tb.vhd
```

A37. MAIN_1- VHDL CODE

```
-- Project      : COEN6501
-- File Name    : main.vhd
-- Author       : Sibi Ravichandran
-- Date         : 01- November- 2018
-- Description :The function of this component is to perform the operation (A*A)-1 where A is an 8-bit input and to store the value of output in Z. After performing the operation the value of end_flag will be set to high.

-- Declare library files:
library IEEE;
use ieee.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

-- Entity Declaration:
entity main is
        port( A         : in std_logic_vector (7 downto 0);
            clock: in std_logic;
      load: in std_logic;
      clear: in std_logic;
      Z           : out std_logic_vector (15 downto 0);
                  end_flag : out std_logic);
end main;

-- Architecture Implementation:
architecture rtl of main is

-- Component Declaration of signed to unsigned converter:
Component sign_2_unsign
        port( input     : in std_logic_vector (7 downto 0);
                A               : out std_logic_vector (7 downto 0));
end Component;

-- 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 multiplier:
Component multiplier_1
        port( A         : in std_logic_vector (7 downto 0);
                Z               : out std_logic_vector (15 downto 0));
end Component;
```

```
-- Component Declaration of minus one:
Component minus_one
        port( A         : in std_logic_vector (15 downto 0);
                Z               : out std_logic_vector (15 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;

-- Signal Declarations:
-- Unsigned 8-bit number:
signal A1 : std_logic_vector (7 downto 0);

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

-- Signals for multiplier:
signal Z1 : std_logic_vector(15 downto 0);

-- Signals for minus operation:
signal Z2 : std_logic_vector(15 downto 0);

begin

-- Convert the signed input number to unsigned 8-bit number:
S2U: sign_2_unsign port map (A (7 downto 0), A1 (7 downto 0));

-- Store the unsigned 8-bit number into a 8-bit register:
Reg8: reg_8_bit port map ( A1(7 downto 0), clock, load, clear, A2(7 downto 0));

-- Perform the square operation Z1 = A*A:
Mul1: multiplier_1 port map (A2(7 downto 0), Z1(15 downto 0));

-- Perform the minus one operation Z2= Z1-1:
Minus: minus_one port map (Z1(15 downto 0), Z2(15 downto 0));

-- Store the result in 16-bit register:
REG16: reg_16_bit port map (Z2(15 downto 0), clock, load, clear, Z(15 downto 0));

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

-- end of main.vhd
```

A38. MAIN_1- TEST BENCH

```
-- Project      : COEN6501
-- File Name    : main_tb.vhd
-- Author       : Amulya Prabhakar
-- Date                 : 03- November- 2018
-- Description : The function of this test bench is to test the functionality of main file
which implements the arithmetic unit that performs the operation (A*A)-1

-- Declare library files:
library IEEE;
use ieee.std_logic_1164.all;

-- Entity Declaration:
entity main_tb is
        -- No port declarations for test bench
end main_tb;

-- Architecture Implementation:
architecture rtl of main_tb is

-- Component Declaration of mainr:
Component main
          port( A          : in std_logic_vector (7 downto 0);
              clock: in std_logic;
      load: in std_logic;
      clear: in std_logic;
      Z          : out std_logic_vector (15 downto 0);
                  end_flag : out std_logic);
end Component;

-- Signal Declarations:
--input of main
signal a_tb: std_logic_vector (7 downto 0):= "00000000";
signal clock: std_logic:='0';
signal load: std_logic:='0';
signal clear: std_logic:='0';

-- Output of main
signal z_tb : std_logic_vector(15 downto 0);
signal END_FLAG_tb: std_logic;

constant clk_period : time := 100 ps;
begin

UUT: main port map ( A=>a_tb,clock=>clock, load=>load, clear=>clear,Z=>z_tb,end_flag
=>END_FLAG_tb);

        --Process to toggle the value of the clock every 50 ps
        clk_process :process
         begin
```

```
        clock <= '0';
        wait for clk_period/2;  --for 0.5 ns signal is '0'.
        clock <= '1';
        wait for clk_period/2;  --for next 0.5 ns signal is '1'.
    end process;

    -- process for testing the register
            stim_proc:process
            begin

                    wait for 300 ps;
                    a_tb<="00101111";
                    clear<='0';
                    load<='1';

                    wait for 300 ps;
                    a_tb<="10000111";
                    clear<='0';
                    load<='1';

                    wait for 300 ps;
                    a_tb<="00001111";
                    clear<='0';
                    load<='1';

                    wait for 300 ps;
                    a_tb<="00000011";
                    clear<='0';
                    load<='0';

                    wait for 300 ps;
                    a_tb<="01100111";
                    clear<='0';
                    load<='1';

                    wait for 300 ps;
                    a_tb<="00000001";
                    clear<='1';
                    load<='1';

                    wait for 300 ps;
                    a_tb<="00100100";
                    clear<='0';
                    load<='1';

            end process;
    end rtl;
    -- end of main_tb.vhd
```

A39. MAIN_2- VHDL CODE

```
-- Project      : COEN6501
-- File Name    : main_2.vhd
-- Author       : Sibi Ravichandran
-- Date         : 01- November- 2018
-- Description : The function of this component is to perform the operation (A*A)-1
where A is an 8-bit input and to store the value of output in Z. After performing the
operation the value of end_flag will be set to high. Here the multiplication is performed
by the RCA.

-- Declare library files:
library IEEE;
use ieee.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

-- Entity Declaration:
entity main_2 is
        port( A         : in std_logic_vector (7 downto 0);
            clock: in std_logic;
      load: in std_logic;
      clear: in std_logic;
      Z           : out std_logic_vector (15 downto 0);
                  end_flag : out std_logic);
end main_2;

-- Architecture Implementation:
architecture rtl of main_2 is

-- Component Declaration of signed to unsigned converter:
Component sign_2_unsign
        port( input     : in std_logic_vector (7 downto 0);
                  A             : out std_logic_vector (7 downto 0));
end Component;

-- 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 multiplier:
Component multiplier_2
        port( A         : in std_logic_vector (7 downto 0);
                  Z             : out std_logic_vector (15 downto 0));
end Component;

-- Component Declaration of minus one:
Component minus_one
        port( A         : in std_logic_vector (15 downto 0);
```

```
                    Z               : out std_logic_vector (15 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;

    -- Signal Declarations:
    -- Unsigned 8-bit number:
    signal A1 : std_logic_vector (7 downto 0);

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

    -- Signals for multiplier:
    signal Z1 : std_logic_vector(15 downto 0);

    -- Signals for minus operation:
    signal Z2 : std_logic_vector(15 downto 0);

    begin

    -- Convert the signed input number to unsigned 8-bit number:
    S2U: sign_2_unsign port map (A (7 downto 0), A1 (7 downto 0));

    -- Store the unsigned 8-bit number into a 8-bit register:
    Reg8: reg_8_bit port map ( A1(7 downto 0), clock, load, clear, A2(7 downto 0));

    -- Perform the square operation Z1 = A*A:
    Mul1: multiplier_2 port map (A2(7 downto 0), Z1(15 downto 0));

    -- Perform the minus one operation Z2= Z1-1:
    Minus: minus_one port map (Z1(15 downto 0), Z2(15 downto 0));

    -- Store the result in 16-bit register:
    REG16: reg_16_bit port map (Z2(15 downto 0), clock, load, clear, Z(15 downto 0));

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

    end;
-- end of main_2.vhd
```

A40. MAIN_2- TEST BENCH

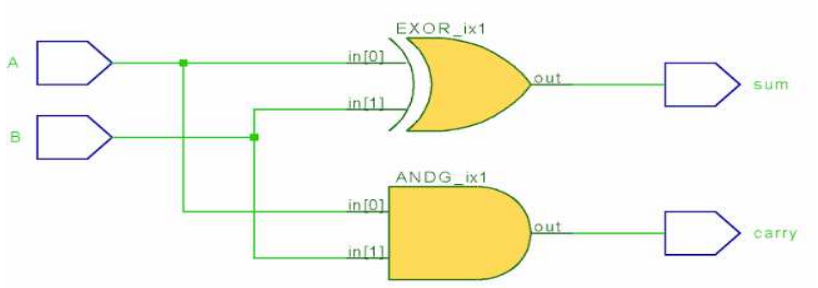
```
    -- Project      : COEN6501
    -- File Name    : main_2_tb.vhd
    -- Author       : Amulya Prabhakar
    -- Date                 : 03- November- 2018
```

```
-- Description :The function of this test bench is to test the functionality of main_2 file
which implements the arithmetic unit that performs the operation (A*A)-1

-- Declare library files:
library IEEE;
use ieee.std_logic_1164.all;

-- Entity Declaration:
entity main_2_tb is
        -- No port declarations for test bench
end main_2_tb;

-- Architecture Implementation:
architecture rtl of main_2_tb is

-- Component Declaration of main:
Component main_2
          port( A          : in std_logic_vector (7 downto 0);
              clock: in std_logic;
      load: in std_logic;
      clear: in std_logic;
      Z            : out std_logic_vector (15 downto 0);
                  end_flag : out std_logic);
end Component;

-- Signal Declarations:
--input of main
signal a_tb: std_logic_vector (7 downto 0):= "00000000";
signal clock: std_logic:='0';
signal load: std_logic:='0';
signal clear: std_logic:='0';

-- output of main
signal z_tb : std_logic_vector(15 downto 0);
signal END_FLAG_tb: std_logic;

constant clk_period : time := 100 ps;
begin

UUT:      main_2      port      map      (      A=>a_tb,clock=>clock,      load=>load,
clear=>clear,Z=>z_tb,end_flag =>END_FLAG_tb);

  -- Process to toggle the value of the clock every 50 ps
  clk_process :process
   begin
        clock <= '0';
        wait for clk_period/2;  --for 0.5 ns signal is '0'.
        clock <= '1';
        wait for clk_period/2;  --for next 0.5 ns signal is '1'.
    end process;
```

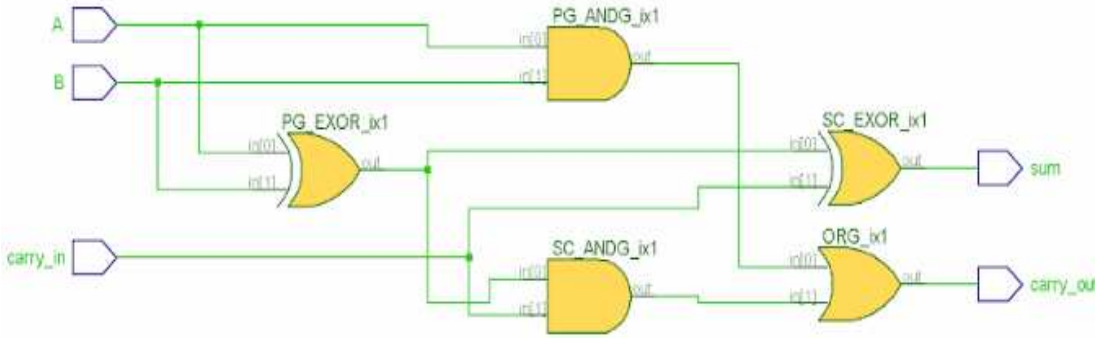
```
-- process for testing the register
        stim_proc:process
        begin

                wait for 300 ps;
                a_tb<="00101111";
                clear<='0';
                load<='1';

                wait for 300 ps;
                a_tb<="10000111";
                clear<='0';
                load<='1';

                wait for 300 ps;
                a_tb<="00001111";
                clear<='0';
                load<='1';

                wait for 300 ps;
                a_tb<="00000011";
                clear<='0';
                load<='0';

                wait for 300 ps;
                a_tb<="01100111";
                clear<='0';
                load<='1';

                wait for 300 ps;
                a_tb<="00000001";
                clear<='1';
                load<='1';

                wait for 300 ps;
                a_tb<="00100100";
                clear<='0';
                load<='1';

        end process;
end rtl;
-- end of main_2_tb.vhd
```

Appendix II – RTL design blocks

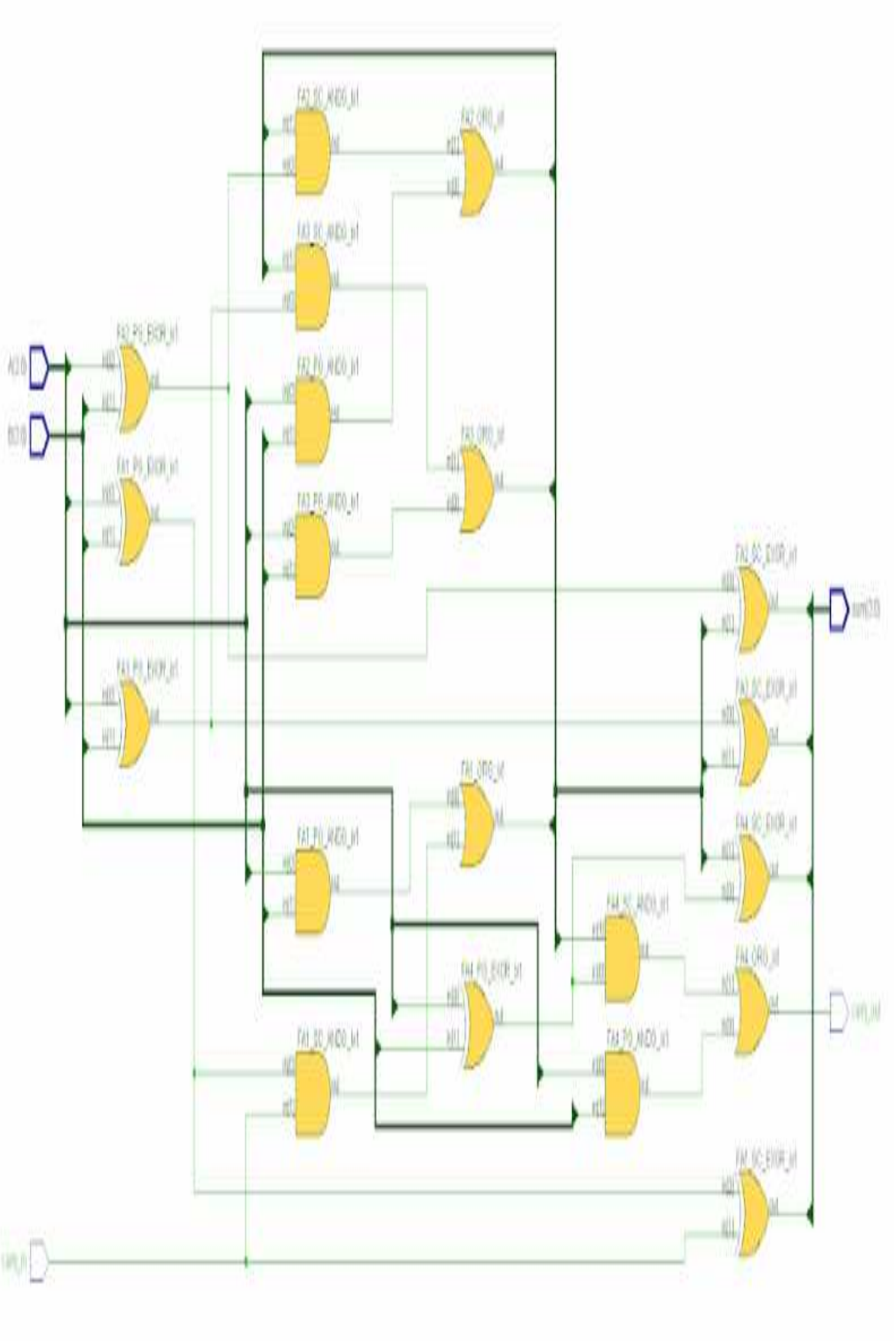
B1. Half Adder



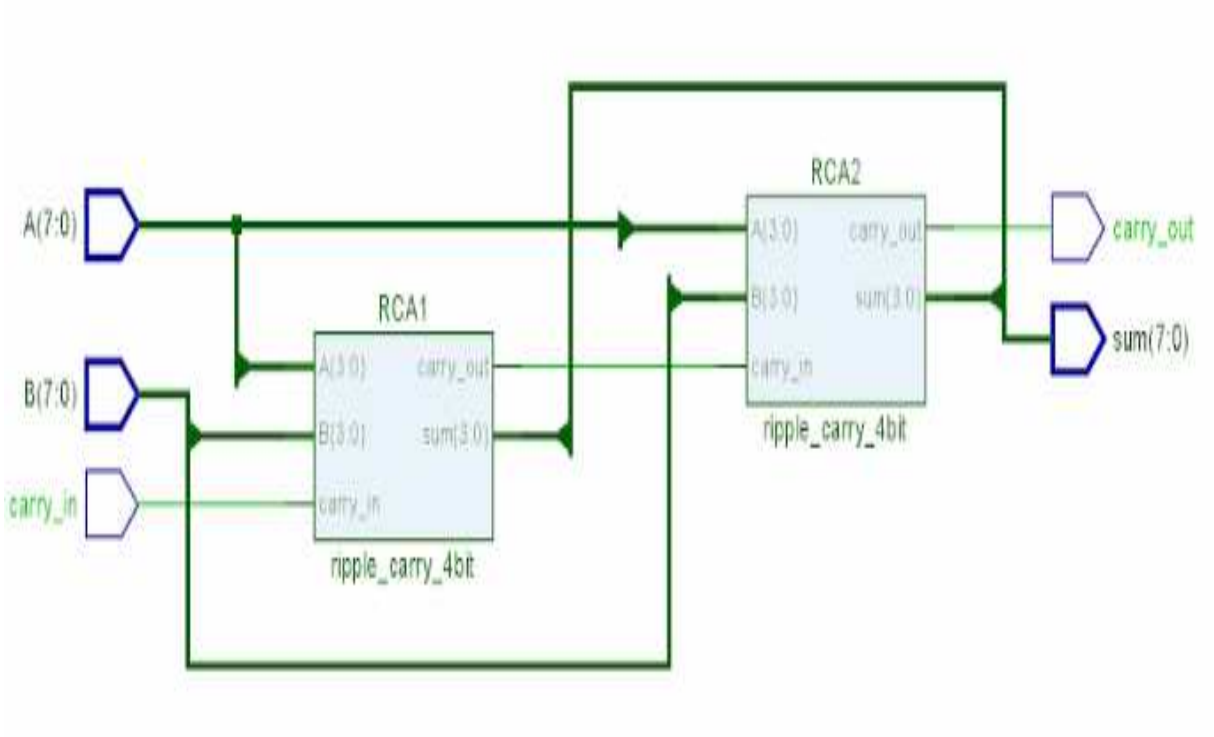
B2. Full Adder



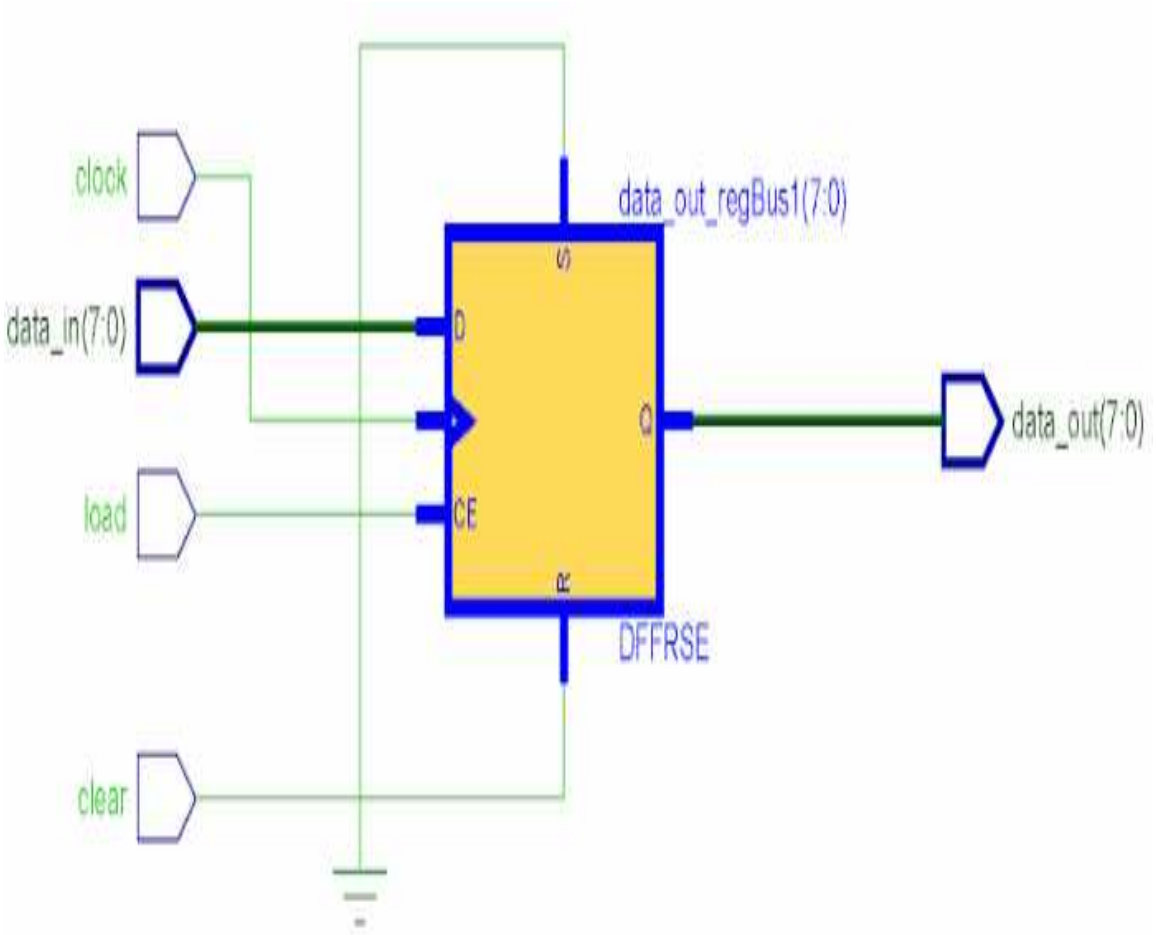
B3. 4-Bit RCA



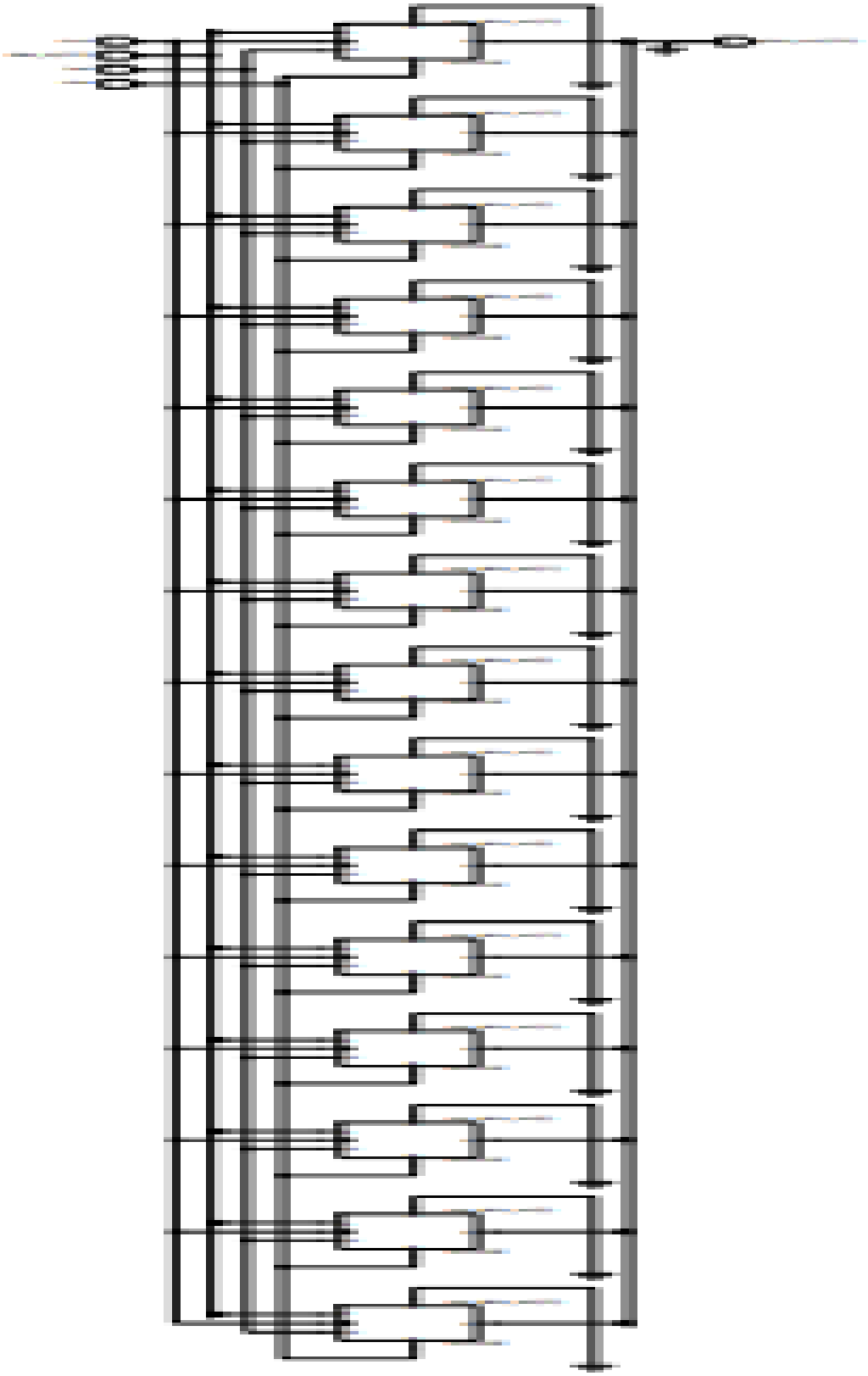
B4. 8-Bit RCA



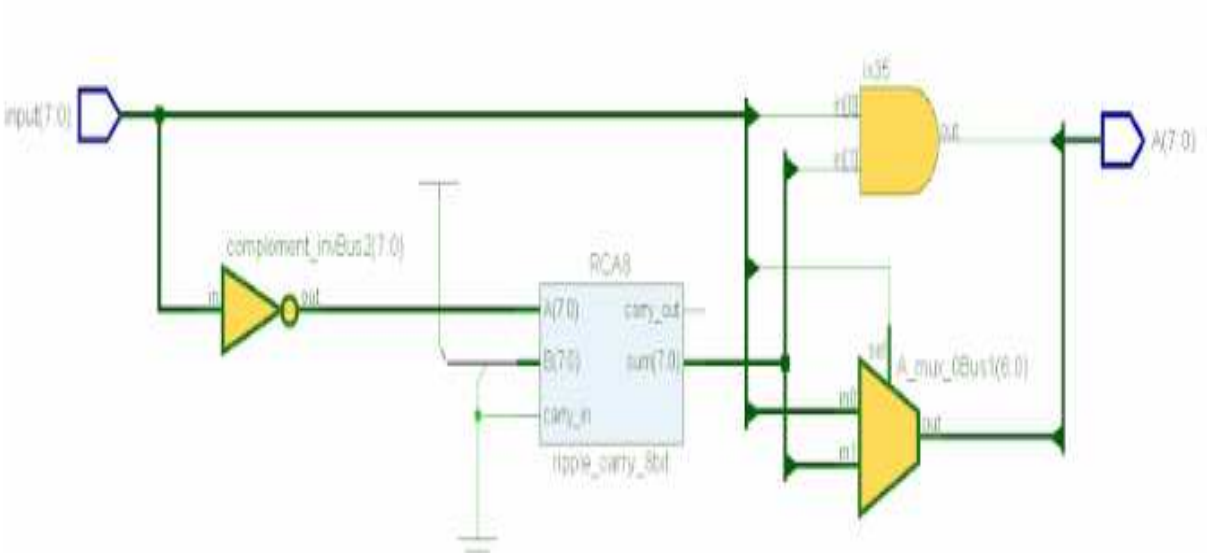
B5. 8-bit Register



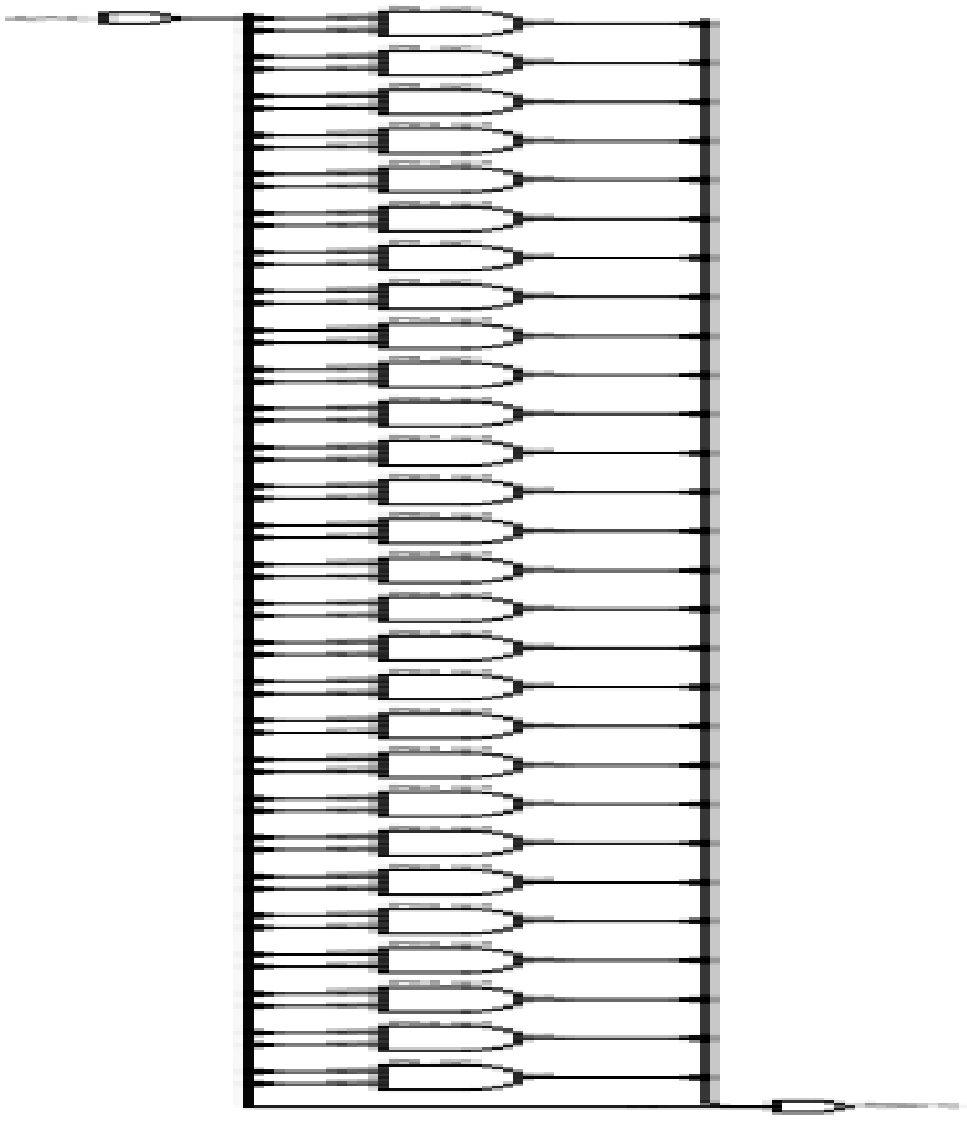
B6. 16-bit Register



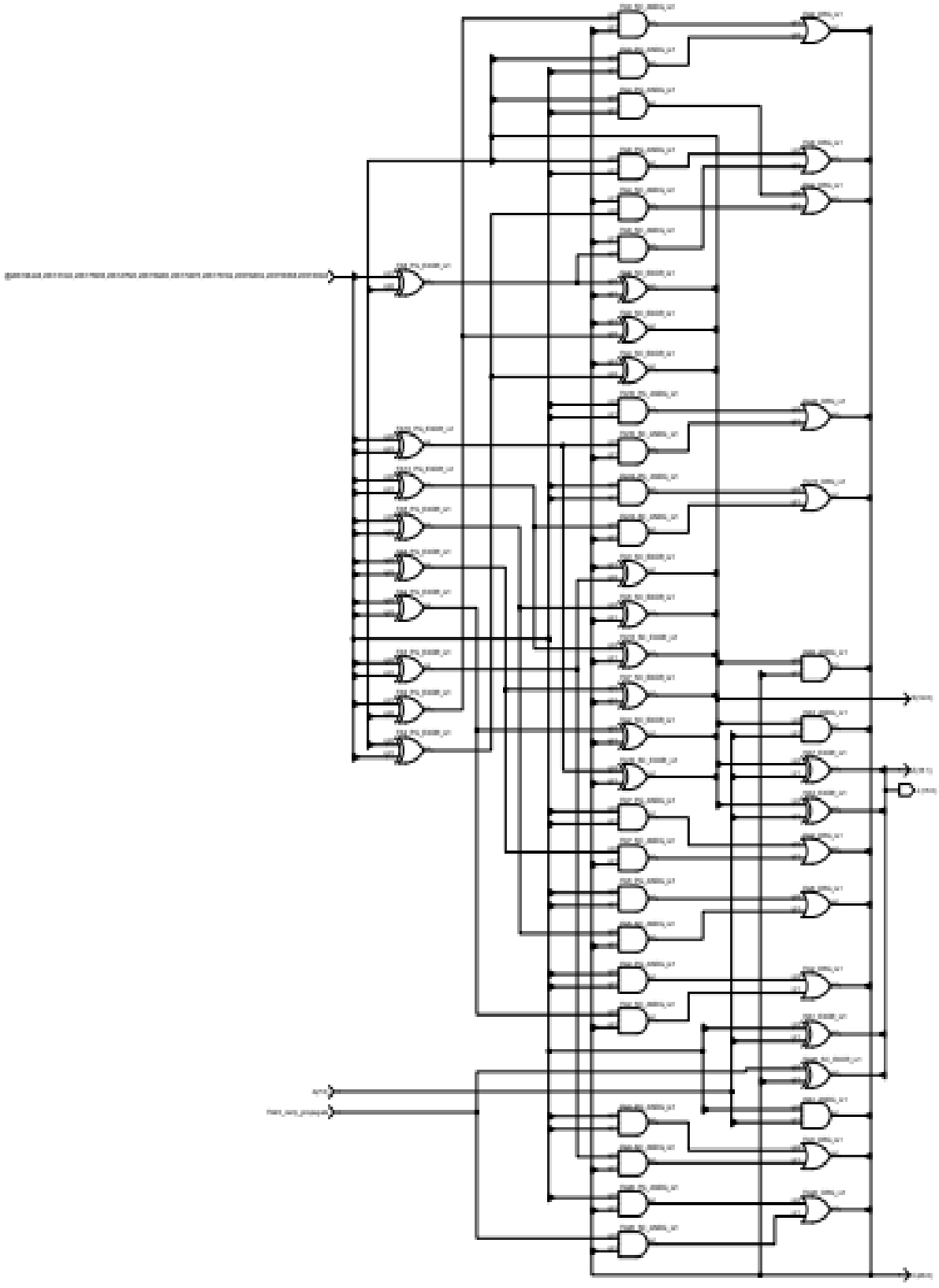
B7. Signed to Unsigned bit Converter



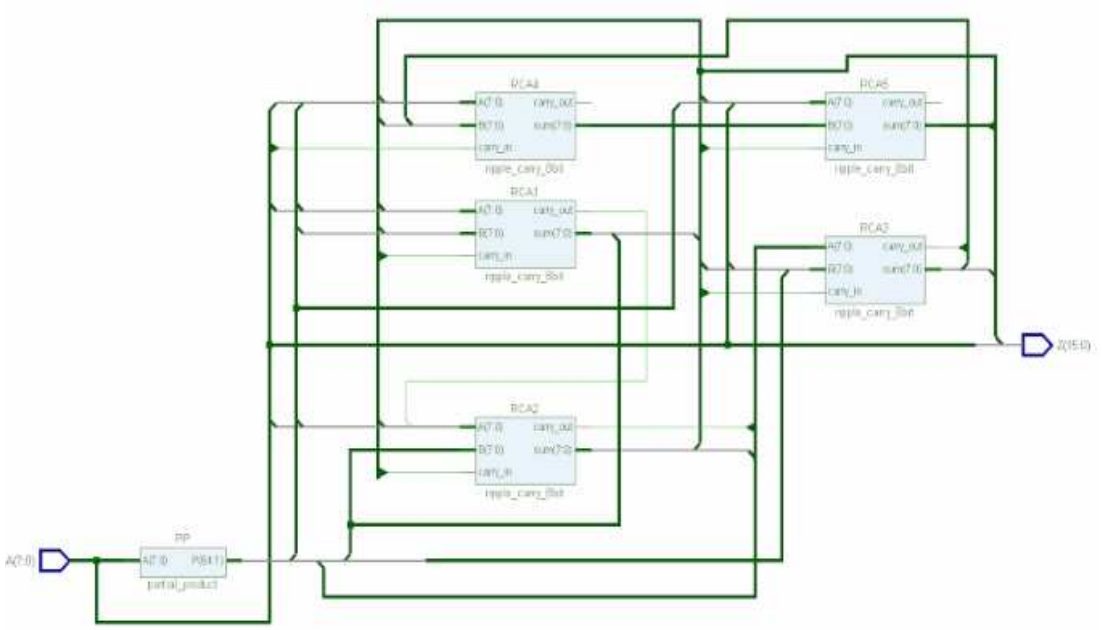
B8. Partial Products Generator



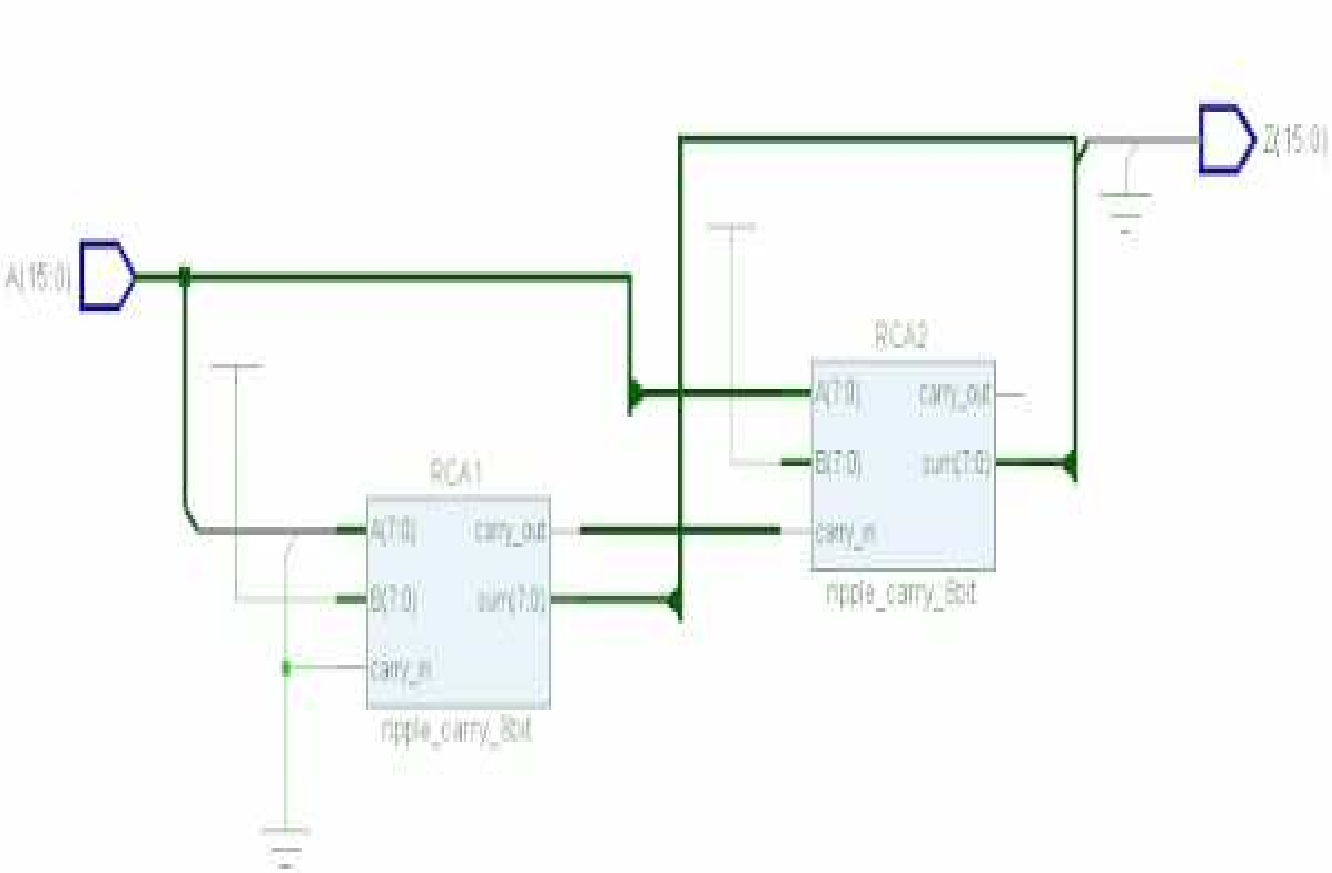
B9. Multiplier using Full Adders and Half Adder



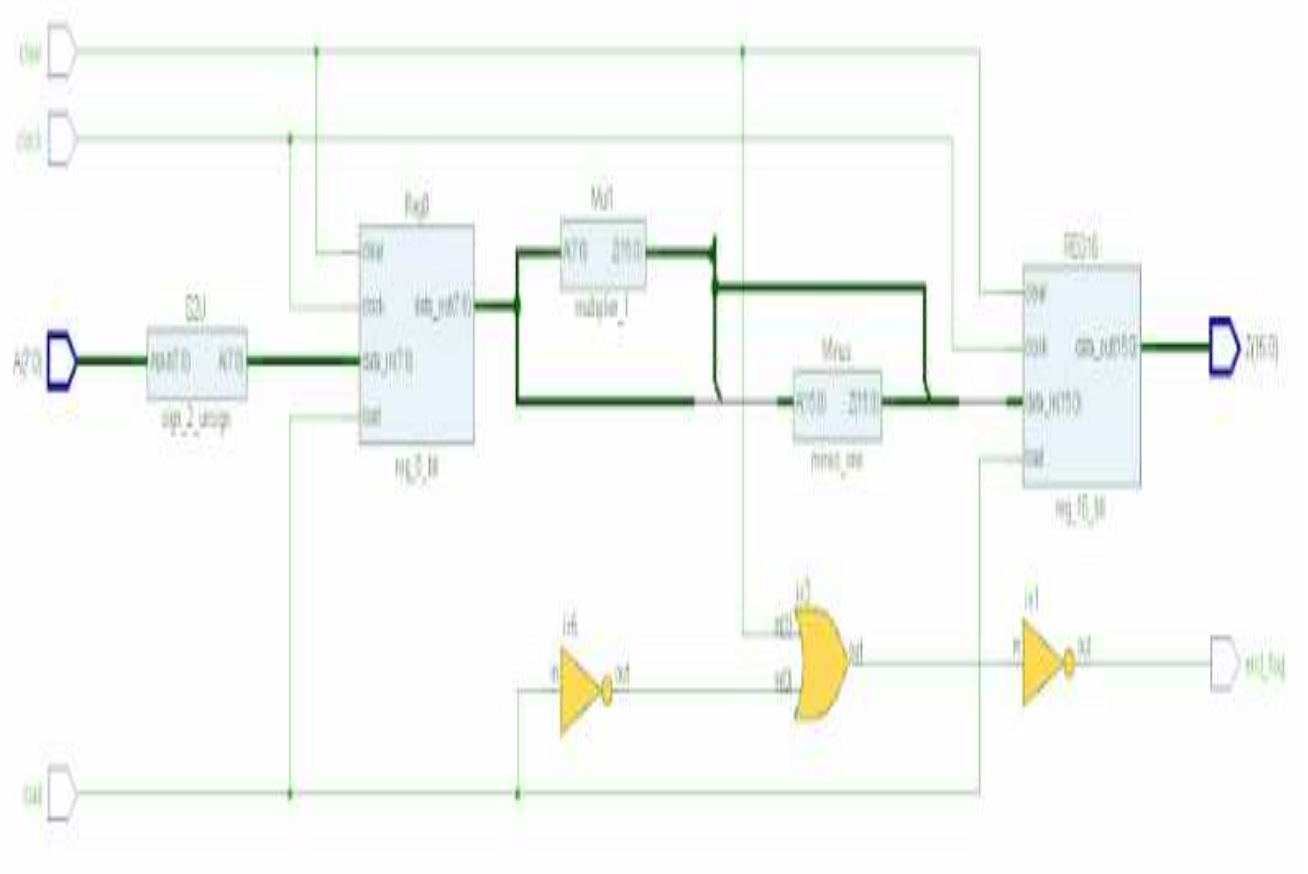
B10. Multiplier using RCA



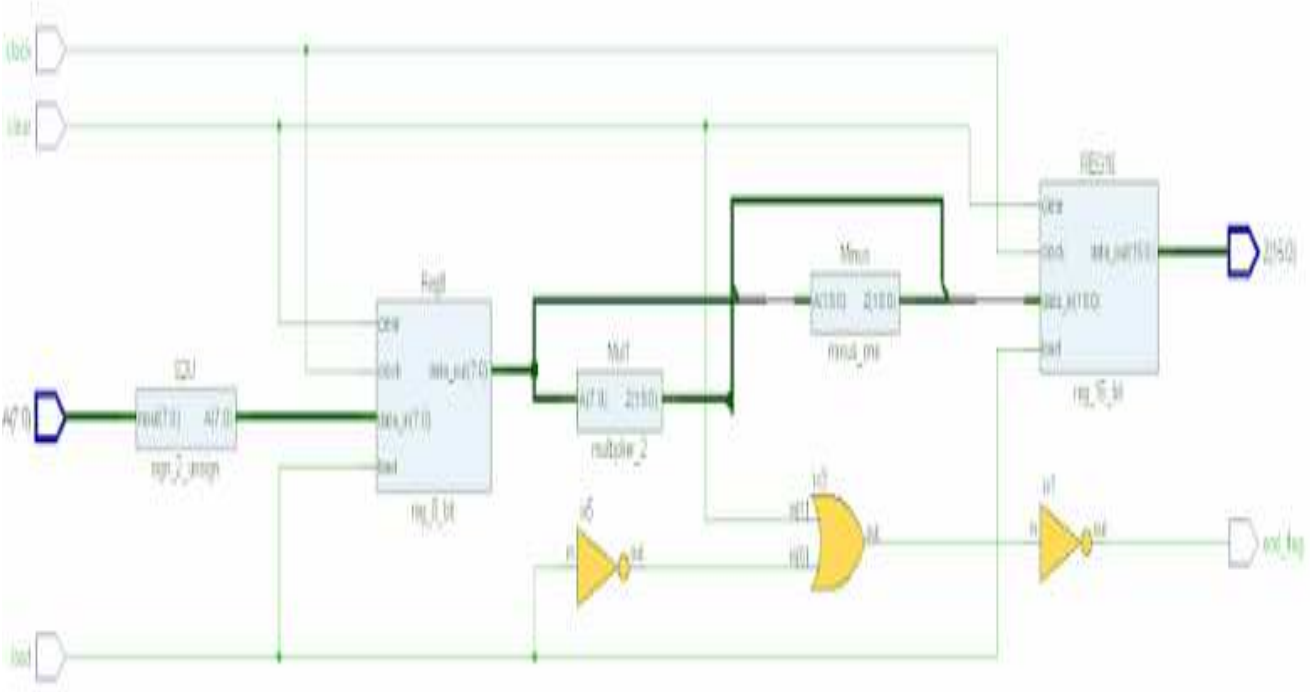
B11. Minus one operator



B12. A2 – 1 using first approach



B13. A^2 – 1 using second approach



Appendix III – Area and Timing report from Xilinx ISE

C1. Precision RTL Area Report for Multiplier_1 using Full Adders and Half Adders.

****

Device Utilization for 2VP2fg256

****

Resource	Used	Avail	Utilization
IOs	24	140	17.14%
Global Buffers	0	16	0.00%
LUTs	98	2816	3.48%
CLB Slices	49	1408	3.48%
Dffs or Latches	0	3236	0.00%
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: multiplier_1 View: mult

****

Cell	Library	References		Total Area
GND	xcv2p	1 x		
IBUF	xcv2p	8 x		
LUT2	xcv2p	26 x	1	26 LUTs
LUT3	xcv2p	12 x	1	12 LUTs
LUT4	xcv2p	60 x	1	60 LUTs
MUXF5	xcv2p	1 x	1	1 MUXF5
OBUF	xcv2p	16 x		

Number of ports :	24
Number of nets :	132
Number of instances :	124
Number of references to this view :	0

Total accumulated area :	
Number of LUTs :	98
Number of MUXF5 :	1
Number of gates :	98
Number of accumulated instances :	124

```
*****************************
IO Register Mapping Report
*****************************
Design: work.multiplier_1.mult
```

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			
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			

Total registers mapped: 0

C2. Xilinx ISE Timing Report for Multiplier_1 using Full Adders and Half Adders

Release 10.1 Trace (lin64)

/nfs/sw_cmc/x86_64.EL7/tools/xilinx_10.1/ISE/bin/lin64/unwrapped/trce -ise
/nfs/home/s/s_avicha/COEN_6501/Modelsim/xilinx/main/mult_1/mult_1.ise -intstyle
ise -e 3 -s 7 -xml multiplier_1 multiplier_1.ncd -o multiplier_1.twr
multiplier_1.pcf -ucf multiplier.ucf

Design file: multiplier_1.ncd
Physical constraint file: multiplier_1.pcf
Device,package,speed: xc2vp30,ff896,-7 (PRODUCTION 1.94 2008-01-09)
Report level: error report

Environment Variable Effect
---------- ------
NONE No environment variables were set

INFO:Timing:2698 - No timing constraints found, doing default enumeration.
INFO:Timing:2752 - To get complete path coverage, use the unconstrained paths option. All paths that are not constrained will be reported in the unconstrained paths section(s) of the report.
INFO:Timing:3339 - The clock-to-out numbers in this timing report are based on a 50 Ohm transmission line loading model. For the details of this model, and for more information on accounting for different loading conditions, please see the device datasheet.

Data Sheet report:

All values displayed in nanoseconds (ns)

Pad to Pad

Source Pad	Destination Pad	Delay
A(0)	Z(0)	3.399
A(0)	Z(2)	6.246
A(0)	Z(3)	6.806
A(0)	Z(4)	6.842
A(0)	Z(5)	7.085
A(0)	Z(6)	9.24
A(0)	Z(7)	8.853
A(0)	Z(8)	11.821
A(0)	Z(9)	11.002
A(0)	Z(10)	11.111
A(0)	Z(11)	11.964
A(0)	Z(12)	13.249
A(0)	Z(13)	12.963
A(0)	Z(14)	12.659
A(0)	Z(15)	13.056
A(1)	Z(2)	6.247
A(1)	Z(3)	6.368
A(1)	Z(4)	5.993
A(1)	Z(5)	6.611
A(1)	Z(6)	8.569
A(1)	Z(7)	8.182
A(1)	Z(8)	11.15
A(1)	Z(9)	10.331
A(1)	Z(10)	10.44
A(1)	Z(11)	11.293

A(1)	Z(12)	12.578
A(1)	Z(13)	12.292
A(1)	Z(14)	11.988
A(1)	Z(15)	12.385
A(2)	Z(3)	7.89
A(2)	Z(4)	8.81
A(2)	Z(5)	8.881
A(2)	Z(6)	10.198
A(2)	Z(7)	9.902
A(2)	Z(8)	12.904
A(2)	Z(9)	12.085
A(2)	Z(10)	12.194
A(2)	Z(11)	13.047
A(2)	Z(12)	14.332
A(2)	Z(13)	14.046
A(2)	Z(14)	13.739
A(2)	Z(15)	14.139
A(3)	Z(4)	5.664
A(3)	Z(5)	6.14
A(3)	Z(6)	7.524
A(3)	Z(7)	7.228
A(3)	Z(8)	10.23
A(3)	Z(9)	9.411
A(3)	Z(10)	9.52
A(3)	Z(11)	10.373
A(3)	Z(12)	11.658
A(3)	Z(13)	11.372
A(3)	Z(14)	11.065
A(3)	Z(15)	11.465
A(4)	Z(5)	6.057
A(4)	Z(6)	7.92
A(4)	Z(7)	7.533
A(4)	Z(8)	10.501
A(4)	Z(9)	9.682
A(4)	Z(10)	9.791
A(4)	Z(11)	10.644
A(4)	Z(12)	11.929
A(4)	Z(13)	11.643
A(4)	Z(14)	11.339
A(4)	Z(15)	11.736
A(5)	Z(6)	9.944
A(5)	Z(7)	9.537
A(5)	Z(8)	12.539
A(5)	Z(9)	11.738
A(5)	Z(10)	12.076

A(5)	Z(11)	12.867
A(5)	Z(12)	14.152
A(5)	Z(13)	13.866
A(5)	Z(14)	13.854
A(5)	Z(15)	13.959
A(6)	Z(7)	7.084
A(6)	Z(8)	10.084
A(6)	Z(9)	9.265
A(6)	Z(10)	9.277
A(6)	Z(11)	10.227
A(6)	Z(12)	11.512
A(6)	Z(13)	11.226
A(6)	Z(14)	10.919
A(6)	Z(15)	11.319
A(7)	Z(8)	9.323
A(7)	Z(9)	8.504
A(7)	Z(10)	8.88
A(7)	Z(11)	9.612
A(7)	Z(12)	10.897
A(7)	Z(13)	10.611
A(7)	Z(14)	10.444
A(7)	Z(15)	10.704

Analysis completed Sun Nov 11 21:03:54 2018

Trace Settings:

Trace Settings

Peak Memory Usage: 298 MB

C3. Precision RTL Area Report for Multiplier_2 using Ripple Carry Adders.

****
Device Utilization for 2VP2fg256
****

Resource	Used	Avail	Utilization
IOs	24	140	17.14%
Global Buffers	0	16	0.00%
LUTs	78	2816	2.77%
CLB Slices	39	1408	2.77%
Dffs or Latches	0	3236	0.00%
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: multiplier_2 View: mult

****

Cell	Library	References		Total Area
GND	xcv2p	1 x		
IBUF	xcv2p	8 x		
LUT2	xcv2p	22 x	1	22 LUTs
LUT3	xcv2p	6 x	1	6 LUTs
LUT4	xcv2p	50 x	1	50 LUTs
OBUF	xcv2p	16 x		

Number of ports :	24
Number of nets :	111
Number of instances :	103
Number of references to this view :	0

Total accumulated area :	
Number of LUTs :	78
Number of gates :	78
Number of accumulated instances :	103

****
IO Register Mapping Report
****
Design: work.multiplier_2.mult

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			
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			

Total registers mapped: 0

C4. Xilinx ISE Timing Report for Multiplier_2 using Ripple Carry Adders

Release 10.1 Trace (lin64)
Copyright (c) 1995-2008 Xilinx, Inc. All rights reserved.

/nfs/sw_cmc/x86_64.EL7/tools/xilinx_10.1/ISE/bin/lin64/unwrapped/trce -ise
/nfs/home/s/s_avicha/COEN_6501/Modelsim/xilinx/main/mult_2/mult_2.ise -intstyle
ise -e 3 -s 7 -xml multiplier_2 multiplier_2.ncd -o multiplier_2.twr
multiplier_2.pcf -ucf multiplier.ucf

Design file: multiplier_2.ncd
Physical constraint file: multiplier_2.pcf
Device,package,speed: xc2vp30,ff896,-7 (PRODUCTION 1.94 2008-01-09)
Report level: error report

Environment Variable Effect

NONE No environment variables were set

INFO:Timing:2698 - No timing constraints found, doing default enumeration.
INFO:Timing:2752 - To get complete path coverage, use the unconstrained paths
option. All paths that are not constrained will be reported in the
unconstrained paths section(s) of the report.
INFO:Timing:3339 - The clock-to-out numbers in this timing report are based on
a 50 Ohm transmission line loading model. For the details of this model,
and for more information on accounting for different loading conditions,
please see the device datasheet.

Data Sheet report:

All values displayed in nanoseconds (ns)

Pad to Pad

Source Pad	Destination Pad	Delay
A(0)	Z(0)	3.582
A(0)	Z(2)	6.114

A(0)	Z(3)	6.404
A(0)	Z(4)	5.608
A(0)	Z(5)	5.753
A(0)	Z(6)	8.248
A(0)	Z(7)	8.078
A(0)	Z(8)	11.566
A(0)	Z(9)	12.867
A(0)	Z(10)	12.317
A(0)	Z(11)	13.25
A(0)	Z(12)	14.699
A(0)	Z(13)	13.547
A(0)	Z(14)	13.47
A(0)	Z(15)	13.61
A(1)	Z(2)	5.559
A(1)	Z(3)	6.511
A(1)	Z(4)	6.173
A(1)	Z(5)	6.043
A(1)	Z(6)	8.44
A(1)	Z(7)	8.27
A(1)	Z(8)	11.758
A(1)	Z(9)	13.059
A(1)	Z(10)	12.509
A(1)	Z(11)	13.442
A(1)	Z(12)	14.891
A(1)	Z(13)	13.739
A(1)	Z(14)	13.662
A(1)	Z(15)	13.802
A(2)	Z(3)	8.341
A(2)	Z(4)	8.466
A(2)	Z(5)	8.312
A(2)	Z(6)	10.179
A(2)	Z(7)	10.009
A(2)	Z(8)	13.497
A(2)	Z(9)	14.798
A(2)	Z(10)	14.248
A(2)	Z(11)	15.181
A(2)	Z(12)	16.63
A(2)	Z(13)	15.478
A(2)	Z(14)	15.401
A(2)	Z(15)	15.541
A(3)	Z(4)	5.93
A(3)	Z(5)	5.858
A(3)	Z(6)	7.563

A(3)	Z(7)	7.393
A(3)	Z(8)	10.881
A(3)	Z(9)	12.182
A(3)	Z(10)	11.632
A(3)	Z(11)	12.565
A(3)	Z(12)	14.014
A(3)	Z(13)	12.862
A(3)	Z(14)	12.785
A(3)	Z(15)	12.925
A(4)	Z(5)	5.568
A(4)	Z(6)	7.227
A(4)	Z(7)	7.057
A(4)	Z(8)	10.545
A(4)	Z(9)	11.846
A(4)	Z(10)	11.296
A(4)	Z(11)	12.229
A(4)	Z(12)	13.678
A(4)	Z(13)	12.526
A(4)	Z(14)	12.449
A(4)	Z(15)	12.589
A(5)	Z(6)	10.674
A(5)	Z(7)	10.504
A(5)	Z(8)	13.992
A(5)	Z(9)	15.293
A(5)	Z(10)	14.743
A(5)	Z(11)	15.676
A(5)	Z(12)	17.125
A(5)	Z(13)	15.973
A(5)	Z(14)	15.896
A(5)	Z(15)	16.036
A(6)	Z(7)	6.72
A(6)	Z(8)	9.862
A(6)	Z(9)	11.163
A(6)	Z(10)	10.613
A(6)	Z(11)	11.546
A(6)	Z(12)	12.995
A(6)	Z(13)	11.95
A(6)	Z(14)	11.873
A(6)	Z(15)	12.013
A(7)	Z(8)	8.852
A(7)	Z(9)	10.153
A(7)	Z(10)	9.603
A(7)	Z(11)	10.276
A(7)	Z(12)	11.725
A(7)	Z(13)	10.562

A(7)	Z(14)	10.485
A(7)	Z(15)	10.625

Analysis completed Sun Nov 11 21:09:39 2018

Trace Settings:

Trace Settings

Peak Memory Usage: 298 MB

C5. Precision RTL Area Report for $A^2 - 1$ unit using Full Adders and Half Adders.

****
Device Utilization for 2VP2fg256
****

Resource	Used	Avail	Utilization
IOs	28	140	20.00%
Global Buffers	1	16	6.25%
LUTs	113	2816	4.01%
CLB Slices	57	1408	4.05%
Dffs or Latches	23	3236	0.71%
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: main View: rtl

****

Cell	Library	References		Total Area
BUFGP	xcv2p	1 x		
FDRE	xcv2p	23 x	1	23 Dffs or Latches
IBUF	xcv2p	10 x		
LUT1	xcv2p	1 x	1	1 LUTs
LUT2	xcv2p	4 x	1	4 LUTs
LUT3	xcv2p	9 x	1	9 LUTs
LUT4	xcv2p	24 x	1	24 LUTs
OBUF	xcv2p	17 x		
multiplier_2	work	1 x	76	76 gates
			76	76 LUTs

Number of ports :	28
Number of nets :	123
Number of instances :	90
Number of references to this view :	0

Total accumulated area :	
Number of Dffs or Latches :	23
Number of LUTs :	113
Number of gates :	114
Number of accumulated instances :	165

****
IO Register Mapping Report
****
Design: work.main.rtl

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			
clock	Input			
load	Input			
clear	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			
end_flag	Output			

Total registers mapped: 0

C6. Xilinx ISE Timing Report for $A^2 - 1$ unit using Full Adders and Half Adders.

Release 10.1 Trace (lin64)
Copyright (c) 1995-2008 Xilinx, Inc. All rights reserved.

/nfs/sw_cmc/x86_64.EL7/tools/xilinx_10.1/ISE/bin/lin64/unwrapped/trce -ise
/nfs/home/s/s_avicha/COEN_6501/Modelsim/xilinx/main/main/main.ise -intstyle ise
-e 3 -s 7 -xml main main.ncd -o main.twr main.pcf -ucf main.ucf

Design file: main.ncd
Physical constraint file: main.pcf
Device,package,speed: xc2vp30,ff896,-7 (PRODUCTION 1.94 2008-01-09)
Report level: error report

Environment Variable Effect

NONE No environment variables were set

INFO:Timing:2698 - No timing constraints found, doing default enumeration.
INFO:Timing:2752 - To get complete path coverage, use the unconstrained paths
option. All paths that are not constrained will be reported in the
unconstrained paths section(s) of the report.
INFO:Timing:3339 - The clock-to-out numbers in this timing report are based on
a 50 Ohm transmission line loading model. For the details of this model,
and for more information on accounting for different loading conditions,
please see the device datasheet.

Data Sheet report:

All values displayed in nanoseconds (ns)

Setup/Hold to clock clock

	Setup to	Hold to		Clock
Source	clk (edge)	clk (edge)	Internal Clock(s)	Phase
A(0)	0.953(R)	1.787(R)	clock_int	0
A(1)	1.257(R)	1.581(R)	clock_int	0
A(2)	3.244(R)	-0.182(R)	clock_int	0
A(3)	0.491(R)	1.818(R)	clock_int	0

A(4)	0.506(R)	1.799(R)	clock_int	0
A(5)	2.414(R)	-0.104(R)	clock_int	0
A(6)	-0.482(R)	1.851(R)	clock_int	0
A(7)	0.175(R)	1.775(R)	clock_int	0
clear	-0.232(R)	2.546(R)	clock_int	0
load	0.109(R)	1.873(R)	clock_int	0

Clock clock to Pad

	clk (edge)		Clock
Destination	to PAD	Internal Clock(s)	Phase
Z(0)	7.198(R)	clock_int	0
Z(1)	7.364(R)	clock_int	0
Z(2)	7.140(R)	clock_int	0
Z(3)	7.145(R)	clock_int	0
Z(4)	7.333(R)	clock_int	0
Z(5)	7.213(R)	clock_int	0
Z(6)	7.211(R)	clock_int	0
Z(7)	7.282(R)	clock_int	0
Z(8)	8.195(R)	clock_int	0
Z(9)	7.105(R)	clock_int	0
Z(10)	7.080(R)	clock_int	0
Z(11)	7.059(R)	clock_int	0
Z(12)	8.472(R)	clock_int	0
Z(13)	7.531(R)	clock_int	0
Z(14)	7.127(R)	clock_int	0
Z(15)	7.604(R)	clock_int	0

Clock to Setup on destination clock clock

	Src:Rise	Src:Fall	Src:Rise	Src:Fall
Source Clock	Dest:Rise	Dest:Rise	Dest:Fall	Dest:Fall
clock	6.826			

Pad to Pad

Source Pad	Destination Pad	Delay
clear	end_flag	4.237
load	end_flag	3.906

Analysis completed Sun Nov 11 20:06:44 2018

Trace Settings:

Trace Settings

Peak Memory Usage: 299 MB

C7. Precision RTL Area Report for $A^2 - 1$ unit using Ripple Carry Adders.

****
Device Utilization for 2VP2fg256
****

Resource	Used	Avail	Utilization
IOs	28	140	20.00%
Global Buffers	1	16	6.25%
LUTs	125	2816	4.44%
CLB Slices	63	1408	4.47%
Dffs or Latches	23	3236	0.71%
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: main_2 View: rtl

****

Cell	Library	References		Total Area
BUFGP	xcv2p	1 x		
FDRE	xcv2p	23 x	1	23 Dffs or Lat
IBUF	xcv2p	10 x		
LUT1	xcv2p	1 x	1	1 LUTs
LUT2	xcv2p	5 x	1	5 LUTs
LUT3	xcv2p	10 x	1	10 LUTs
LUT4	xcv2p	22 x	1	22 LUTs
OBUF	xcv2p	17 x		
multiplier_2	work	1 x	88	88 gates

88 88 LUTs

Number of ports :	28
Number of nets :	120
Number of instances :	90
Number of references to this view :	0

Total accumulated area :	
Number of Dffs or Latches :	23
Number of LUTs :	125
Number of gates :	126
Number of accumulated instances :	177

****
IO Register Mapping Report
****
Design: work.main_2.rtl

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			
clock	Input			
load	Input			
clear	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			
end_flag	Output			

Total registers mapped: 0

C8. Xilinx ISE Timing Report for $A^2 - 1$ unit using Ripple Carry Adders

Release 10.1 Trace (lin64)
Copyright (c) 1995-2008 Xilinx, Inc. All rights reserved.

/nfs/sw_cmc/x86_64.EL7/tools/xilinx_10.1/ISE/bin/lin64/unwrapped/trce -ise
/nfs/home/s/s_avicha/COEN_6501/Modelsim/xilinx/main/main_2/main_2.ise -intstyle
ise -e 3 -s 7 -xml main_2 main_2.ncd -o main_2.twr main_2.pcf -ucf main.ucf

Design file: main_2.ncd
Physical constraint file: main_2.pcf
Device,package,speed: xc2vp30,ff896,-7 (PRODUCTION 1.94 2008-01-09)
Report level: error report

Environment Variable Effect
---------- ------
NONE No environment variables were set

INFO:Timing:2698 - No timing constraints found, doing default enumeration.
INFO:Timing:2752 - To get complete path coverage, use the unconstrained paths option. All paths that are not constrained will be reported in the unconstrained paths section(s) of the report.
INFO:Timing:3339 - The clock-to-out numbers in this timing report are based on a 50 Ohm transmission line loading model. For the details of this model, and for more information on accounting for different loading conditions, please see the device datasheet.

Data Sheet report:

All values displayed in nanoseconds (ns)

Setup/Hold to clock clock
Setup/Hold to clock clock

	Setup to	Hold to		Clock
Source	clk (edge)	clk (edge)	Internal Clock(s)	Phase
A(0)	0.814(R)	1.414(R)	clock_int	0
A(1)	0.892(R)	1.732(R)	clock_int	0
A(2)	2.842(R)	-0.131(R)	clock_int	0
A(3)	-0.043(R)	1.736(R)	clock_int	0
A(4)	-0.261(R)	1.696(R)	clock_int	0
A(5)	1.841(R)	0.217(R)	clock_int	0
A(6)	-0.661(R)	1.993(R)	clock_int	0

A(7)	0.244(R)	1.731(R)	clock_int	0
clear	0.058(R)	2.536(R)	clock_int	0
load	0.362(R)	2.044(R)	clock_int	0

Clock clock to Pad

Destination	clk (edge) to PAD	Internal Clock(s)	Clock Phase
Z(0)	7.388(R)	clock_int	0
Z(1)	7.357(R)	clock_int	0
Z(2)	7.357(R)	clock_int	0
Z(3)	7.134(R)	clock_int	0
Z(4)	7.331(R)	clock_int	0
Z(5)	7.036(R)	clock_int	0
Z(6)	7.442(R)	clock_int	0
Z(7)	7.191(R)	clock_int	0
Z(8)	8.212(R)	clock_int	0
Z(9)	7.413(R)	clock_int	0
Z(10)	7.309(R)	clock_int	0
Z(11)	7.265(R)	clock_int	0
Z(12)	8.151(R)	clock_int	0
Z(13)	7.253(R)	clock_int	0
Z(14)	7.249(R)	clock_int	0
Z(15)	7.211(R)	clock_int	0

Clock to Setup on destination clock clock

Source Clock	Src:Rise Dest:Rise	Src:Fall Dest:Rise	Src:Rise Dest:Fall	Src:Fall Dest:Fall
clock	8.392			

Pad to Pad

Source Pad	Destination Pad	Delay
clear	end_flag	4.442
load	end_flag	4.019

Analysis completed Sun Nov 11 20:16:57 2018

Trace Settings:

Trace Settings

Peak Memory Usage: 299 MB

Appendix IV – Work Summary Sheet

WORK	SIBI	AMULYA
Algorithm and Work Flow for Project	Y	Y
VHDL CODE		
AND GATE		Y
OR GATE		Y
NOT GATE		Y
NAND GATE		Y
NOR GATE		Y
EX-OR GATE		Y
EX-NOR GATE		Y
HALF ADDER		Y
FULL ADDER		Y
4-BIT RCA		Y
8-BIT RCA		Y
8-BIT Register	Y	
16-BIT Register	Y	
SIGNED TO UNSIGNED	Y	
PARTIAL PRODUCT	Y	
MULTIPLIER_1 (with Half Adder and Full Adder)	Y	
MULTIPLIER_2 (with RCA)	Y	
MINUS ONE	Y	
MAIN	Y	
TEST BENCH		
AND GATE	Y	
OR GATE	Y	
NOT GATE	Y	
NAND GATE	Y	
NOR GATE	Y	
EX-OR GATE	Y	

EX-NOR GATE	Y	
HALF ADDER	Y	
FULL ADDER	Y	
4-BIT RCA	Y	
8-BIT RCA	Y	
8-BIT Register		Y
16-BIT Register		Y
SIGNED TO UNSIGNED		Y
PARTIAL PRODUCT		Y
MULTIPLIER_1 (with Half Adder and Full Adder)		Y
MULTIPLIER_2 (with RCA)		Y
MINUS ONE		Y
MAIN		Y
AREA, POWER AND DELAY CALCULATION	Y	Y
REPORT	Y	Y