Concordia University
Engineering and Computer Science

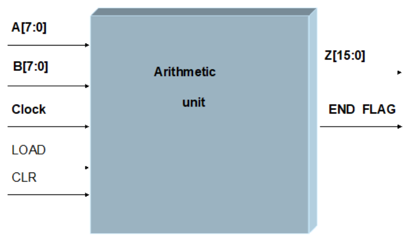
Fall 2019

COEN 6501 : Digital Design and Synthesis

Dr. Asim J. Al-Khalili

Project Report

Arithmetic Unit



<u>*Team Members*</u>

Mohammed Abu Fares (40137642)

Rayan Alzanbaqi (40091507)

Table of Content

List of Figures

Abstract

Arithmetic units are essential components that are able to perform calculations and execute arithmetic processes that play vital role in digital signal processing. Basic operations such as addition, subtraction, multiplication, and division are the most common needed tasks to be completed by an arithmetic unit. The Arithmetic logic unit (ALU) is very fundamental parameter in the central processing unit (CPU) of a computer; therefore, having a well-designed arithmetic unit contributes drastically in the performance in the digital signal processing applications. In this project, we have designed a unit that is able to achieve an arithmetic calculation using Hardware Description Language VHDL code operated by Modelsim software. Also Xilinx Integrated system Environment was used to obtain different essential parameters and reports of the designed circuit in order to have a source for analyzing the executed methodology. The design approach is discussed step by step and analyzed in this paper along with the final results.

Acknowledgment

We are glad that we had the chance to be instructed by Dr. Asim J. Al-Khalili to learn Digital Design and Synthesis. His experience for such an important field widened our knowledge in digital signal processing and made us more qualified to the work field. Also we are thankful for the facilities provided by Concordia University in order to provide the best and ideal environments to work on the project. We would like to thank our teacher assistant, Mr. Rokibul Hasan, who provided us with all required information for accessing the software easily.

1. Introduction

There are various possible approaches that can be used in order to perform a design for an Arithmetic Logic Unit (ALU). Our objective is to achieve the optimal possible approach to accomplish a well-designed balanced system of arithmetic operation in terms of speed, area, and power consumption. The design was mainly based on a structural design instead of a behavioral, which means any required step or approach is performed using circuits and connections. The required arithmetic unit that we designed in our project is capable in computing the following equation:

$$Z = \frac{1}{4}[AxB] + 1$$

Based also on the requirements of the project, the A and B operands are to be unsigned of 8-bits length which will result in unsigned 16-bits output Z. Also, both input operands A and B should be kept in an 8-bit register, which is a storage component for inputs that can be controlled through what represented by a control unit to latch and clear whenever needed. In addition to the input operands, the final result should be also kept in a final 16-bits register controlled to operate under certain conditions. This list below summarizes the requirements of the project.

<u>Essential Requirements:</u>

- The operands A and B are latched into register RA and RB when LOAD signal transit from high to low.
- The 16-bit product shall be loaded into the 16-bit Z port.
- The design should be structural.
- The CLEAR signal will clear all registers to ‘0’.
- The unit performs the arithmetic operation until END_FLAG becomes high.
- The Test Bench.

The control units are presented in our project in order to show in different tasks for the system. The units which are presented in our design are LOAD, CLOCK, and CLEAR. Each one of them contributes highly in the performance of the system and makes operation more organized. For example, a LOAD signal transition from high to low will latch operands into register A and B. In addition to that the clock has a vital role in the operation of the D Flip Flop for storing bits, which is a topic that will be discussed in the upcoming sections. Moreover, the clear signal organizes the resetting of the stored bits in the registers and makes them ready for receiving new sequence of bits. Control units also play a role in performing another requirement in the project, which is the END_FLAG operation. This operation is required to show the end of arithmetic operation by showing a high a signal. The list below summarizes the essential parameters and their function that will be shown in the project. All in all, also the delay, area, and power final reports are obtained and can be found in Appendices B.

Fundamental Parameters:

- A0-A7: Unsigned 8 bit Operand A
- B0-B7: Unsigned 8 bit Operand B
- Z0-Z15: Unsigned 16 bit output Z
- CLR: Clears selected registers to '0'
- LOAD: Loads Operand into internal registers
- CLK: Clock input
- END_FLAG: Indicates end of operation

2. The Overall Design

It is essential to consider a structural design at this stage as mentioned earlier since it is possible to execute a step using features from VHDL such as process. And each stage should be designed carefully in order to meet the requirement of the project.

The first step of the design is to analyze the given function, and it is clear that in order to perform the arithmetic operation in the function shown earlier, there will be a need for a multiplier, shifter, and adder. The multiplier to perform the multiplication of operand A and B, shifter to perform the multiplication by ¼, and the adder for the addition of decimal '1'.

Firstly, there are different multiplication common methods such as Serial Multiplier, Parallel Multiplier, Shift & Add Multiplier, Booth Multiplication, and Array multiplication. Each type has different performance and used based on design requirement. In addition to the multiplication, shifting can be done in various methods on VHDL; we have used the barrel shifter to perform the shifting since we believe it is a complete structural design to perform the shifting. Furthermore, besides the multiplication and shifting, there are different common types of adding that can be used such Ripple Carry Adder, Carry look-ahead adder, Carry skip adder, Manchester Chain adder, and Carry select adders. Also each type has different performance and used based on design requirement.

In our project, we have used Array Multiplier to perform the multiplication of operands A and B. Array multiplier is suitable for the multiplication of two unsigned numbers and has the advantage of optimizing either the delay or area in terms of design and distribution of the Full Adder/Half Adders that are used in this type of multiplier. Therefore, there are various methodologies that can manipulate the design and performance of the array multiplier (A1 & Shinde2, 2016). The carry select adder is used to perform the addition, and the main feature of the carry select adder is that it performs a high speed operation that can compensate any delay in the system.

The procedure of design to perform the arithmetic operation is to firstly latch the operands A and B into registers. The data in the registers will be taken into the array multiplier block to perform the multiplication. After that, the result of multiplication will propagate to the barrel shifter to perform the shifting of the multiplication result. Lastly, the shifted result will be taken into the carry select adder block, which will give the final result to be stored in a final result register that gives the final output. Figure 1 below summarizes the procedure in the system.

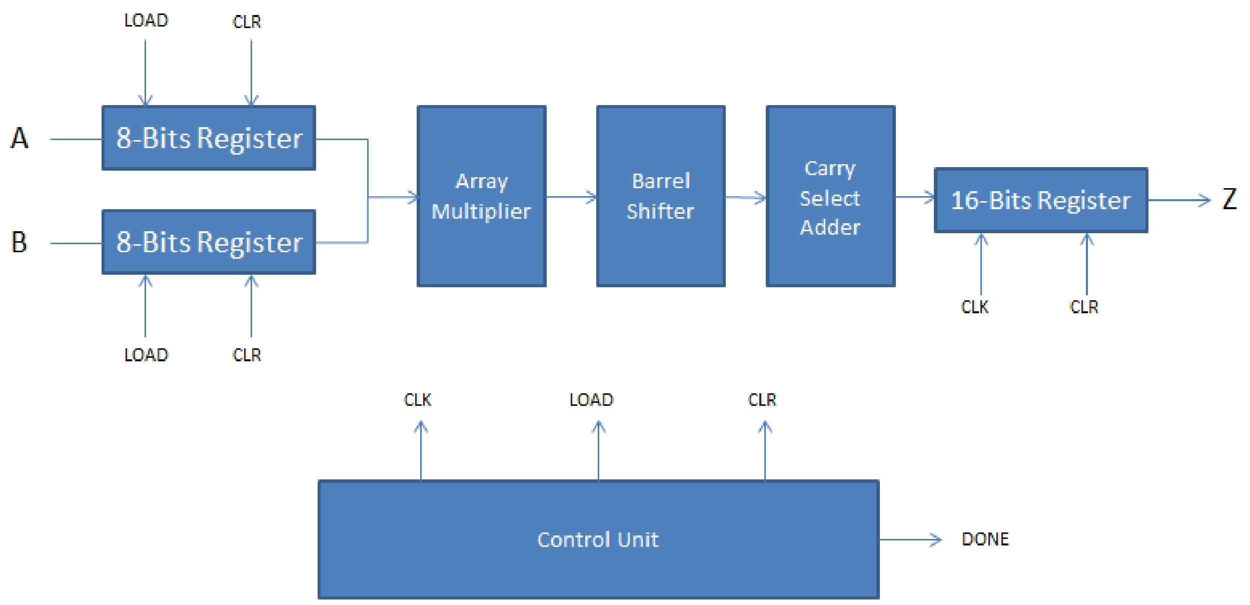


Figure 1: Block Diagram of system

All in all, the final block connections that perform the arithmetic operation are based on essential blocks, which are the multiplier, shifter, and the adder as discussed. These essential blocks are built using fundamental components that perform operations in DSP such as full adders and half adders. And these fundamental components are built using basic structures, which are the gates. A further discussion and simulation will be shown in the coming sections to explain the structure.

3. Fundamental Gates

Gates are the most basic structure in the design. And in this section, the gates used in the design will be illustrated and explained briefly.

3.1 And_1 Gate

And_1 gate is used to perform the function of the AND of two input pins. AND gate will always have a result of zero unless when both inputs of the gate are with logic 1. Table 1 below illustrates the truth table for the function of AND gate, and figure 2 represents the result of simulating and_1 gate.

A	*B*	*Output*
0	*0*	*0*
0	*1*	*0*
1	*0*	*0*
1	*1*	*1*

Table 1: Truth Table for And Gate

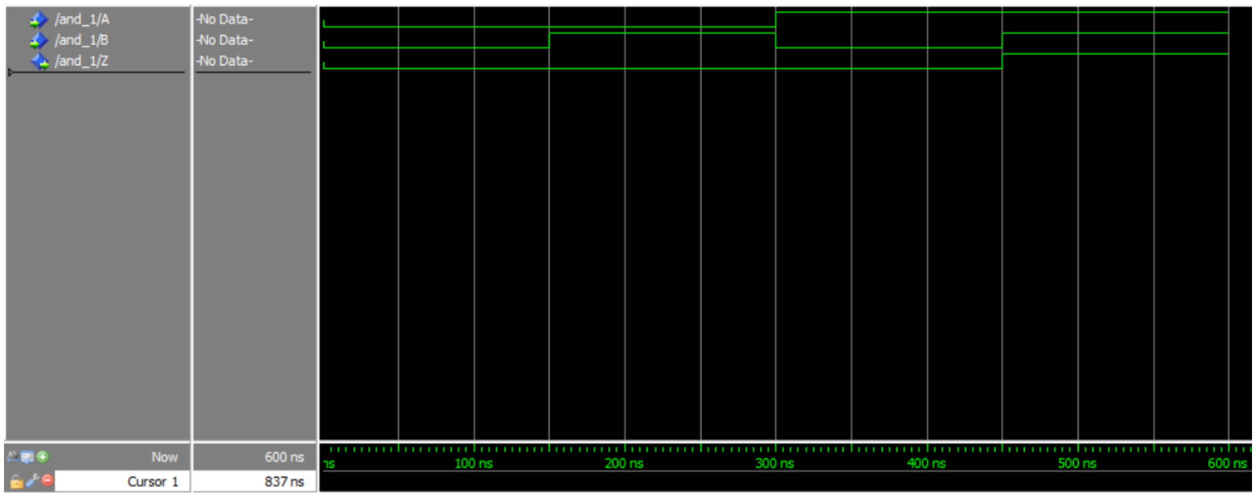


Figure 2: And gate simulation

3.2 Or_1 Gate

Or_1 gate is used to perform the function of the OR of two input pins. In this gate, as long as one of the inputs has logic 1, then the output will be 1. Table 2 below illustrates the truth table for the function of OR gate, and figure 3 represents the result of simulating or_1 gate.

A	*B*	*Output*
0	*0*	*0*
0	*1*	*1*
1	*0*	*1*
1	*1*	*1*

Table2: Truth table for OR gate

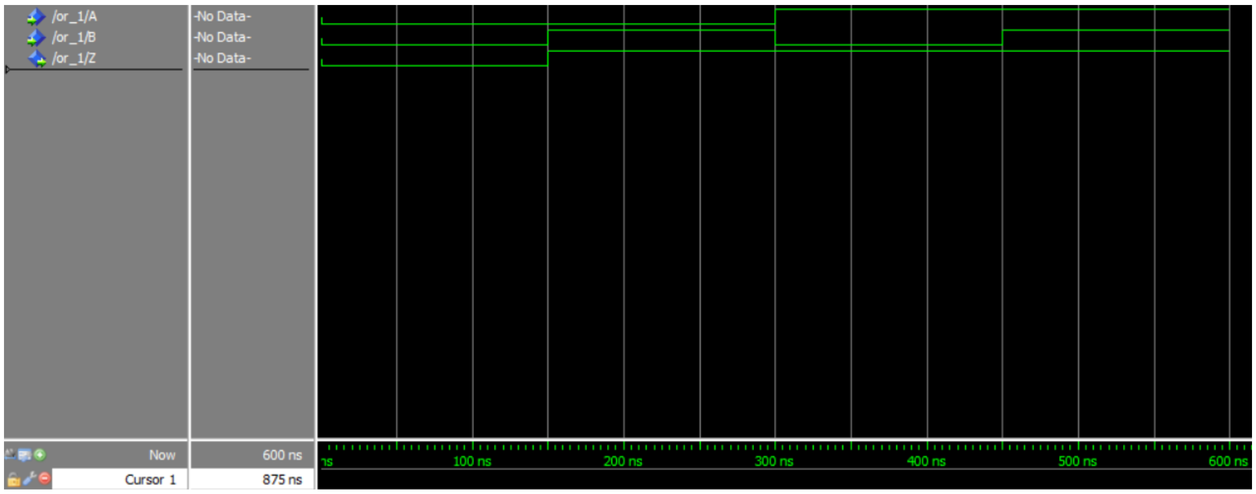


Figure 3: Or Gate Simulation

3.2 NOT_1 Gate

NOT_1 gate is used to perform the function of the NOT of one input pin. NOT gate performs as an inverter of the input signal in which it makes the output pin with opposite logic of the input pin. Table 3 below illustrates the Truth Table for the function of NOT gate, and figure 4 represents the result of simulating NOT_1 gate.

Input	*Output*
0	*1*
1	*0*

Table 3: Truth Table for NOT gate

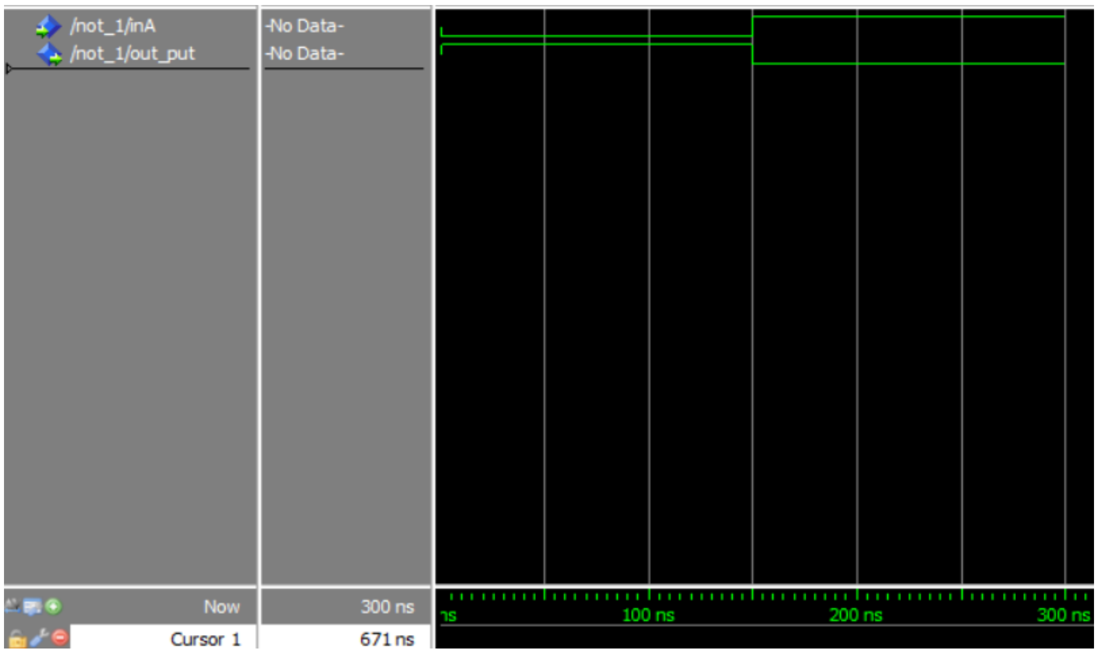


Figure 4: NOT Gate Simulation

3.3 XOR_1 Gate

XOR_1 gate is used to perform the function of the XOR of two input pins. In XOR, as long as the two input pins have different logic, then the output will be with logic 1. Also, output is logic 0 when both inputs have same logic signal. Table 4 below illustrates the truth table for the function of XOR gate, and figure 5 represents the result of simulating XOR_1 gate.

A	*B*	*Output*
0	*0*	*0*
0	*1*	*1*
1	*0*	*1*
1	*1*	*0*

Table 4: Truth table for XOR gate

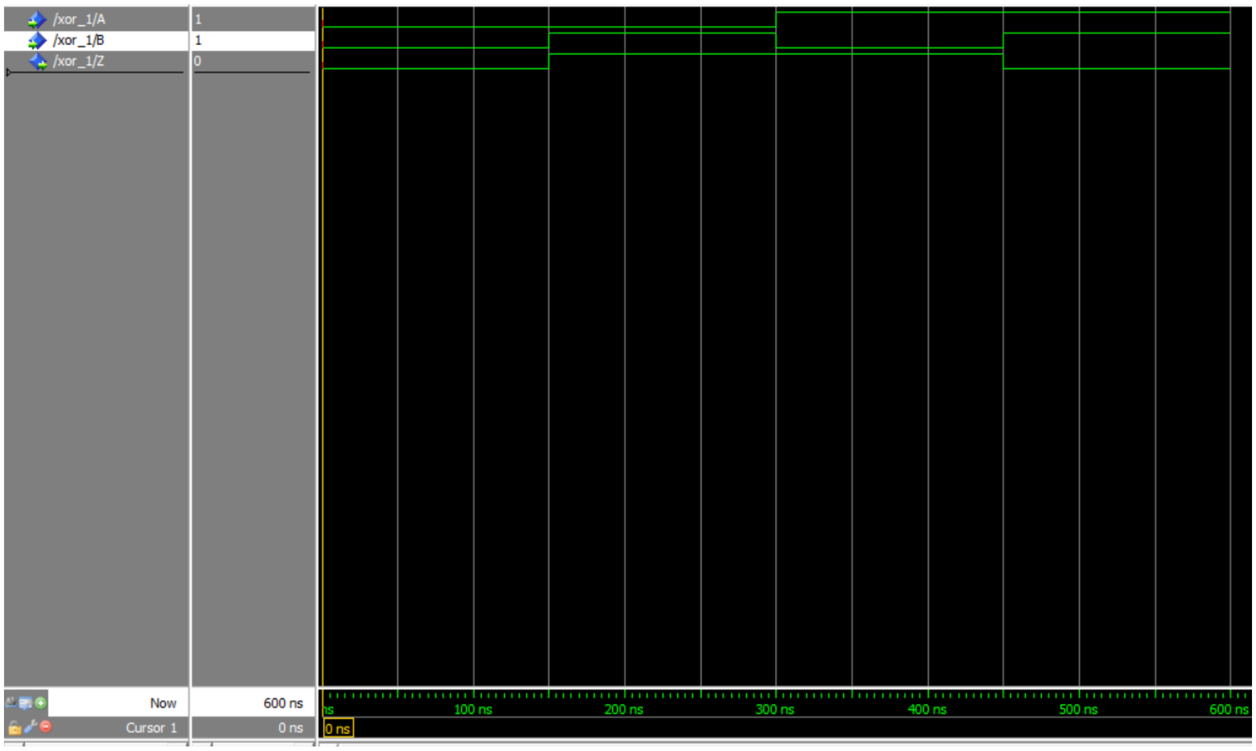


Figure 5: XOR Gate Simulation

3.4 NAND_1 Gate

NAND _1 gate is used to perform the function of the NAND of two input pins. NAND will always give logic ‘1’ unless when all inputs are with logic ‘1’ that will have a result of ‘0’. The table 5 below illustrates the truth table for the function of NAND gate, and figure 6 represents the result of simulating NAND_1 gate.

A	*B*	*Output*
0	*0*	*1*
0	*1*	*1*
1	*0*	*1*
1	*1*	*0*

Table 5: Truth Table of NAND

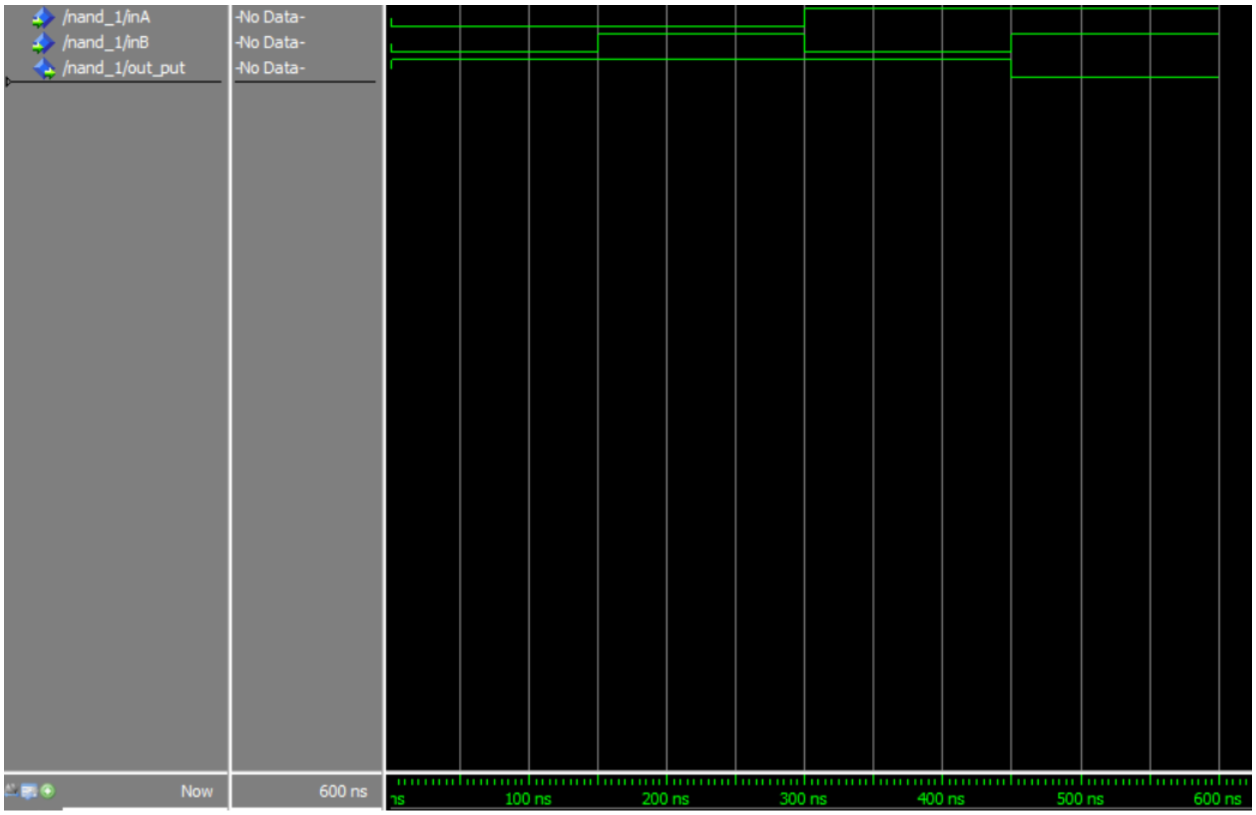


Figure 6: NAND Gate Simulation

3.5 NAND_2 Gate

This gate is another gate which is needed, and it is similar to the previous described NAND. However, this gate is built to have three inputs instead of two only. Table 6 below illustrates the Truth Table for the function of NAND gate, and figure 7 represents the result of simulating NAND_1 gate.

A	*B*	*C*	*Output*
0	*0*	*0*	*1*
0	*0*	*1*	*1*
0	*1*	*0*	*1*
0	*1*	*1*	*1*
1	*0*	*0*	*1*
1	*0*	*1*	*1*
1	*1*	*0*	*1*
1	*1*	*1*	*0*

Table 6: Truth Table of NAND with 3-inputs

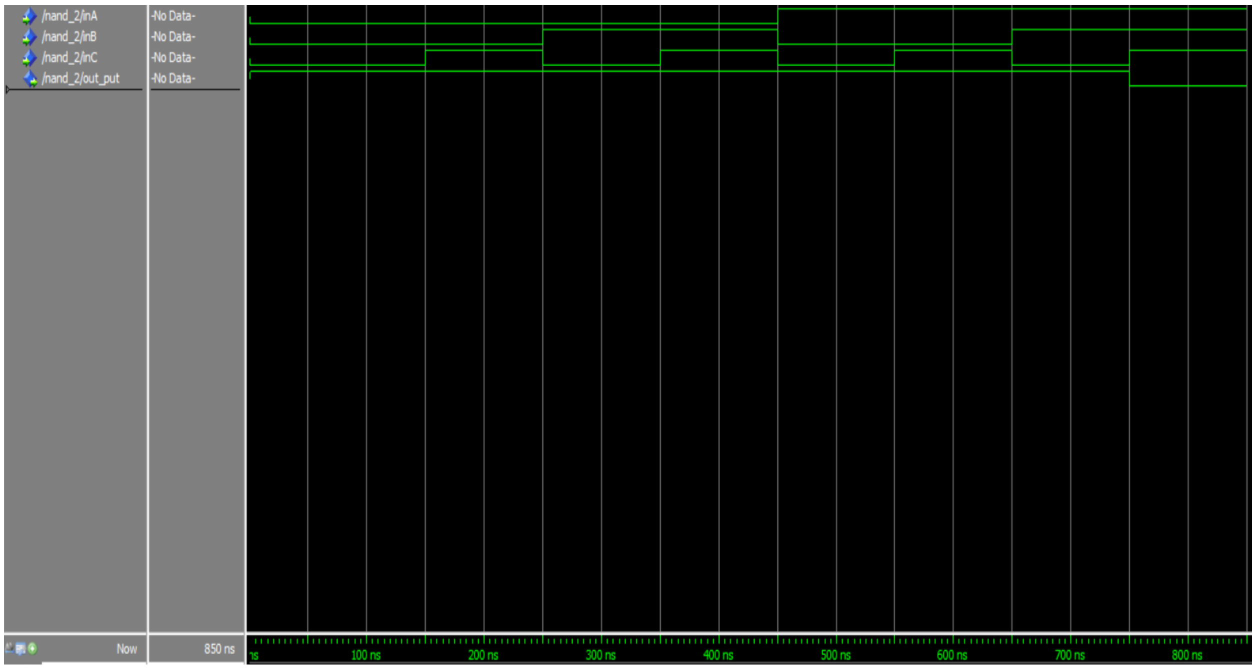


Figure 7: NAND Gate Simulation with 3-Inputs

4. Building Required Components

There are different basic components required in the design, and each component consists of group of gates represented previously made with different arrangement and connections in a way to provide a structural design. These components can perform more complex operations that are fundamental such as multiplication, storing, shifting, and addition of bits. The creation of these essential operations will be the key in creating more complex functions that are required for the final design output. The method for connection of component can be found in Appendix A.

4.1 Half Adder

Half adder is one of the most essential components in the design. Like in any normal arithmetic operation, the function of this component is to add two bits and give two outputs, which are the sum and the carry. Table 7 below illustrates the truth table of the half adder.

A	B	Sum	$Carry$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 7: Truth table of Half Adder

The function of the output for Half Adder can be represented also in the following Boolean formula:

$$Sum = A + B$$

$$Carry = A * B$$

In addition to that, figure 8 below illustrates the result of simulating the Half Adder and figure 9 shows the circuit.

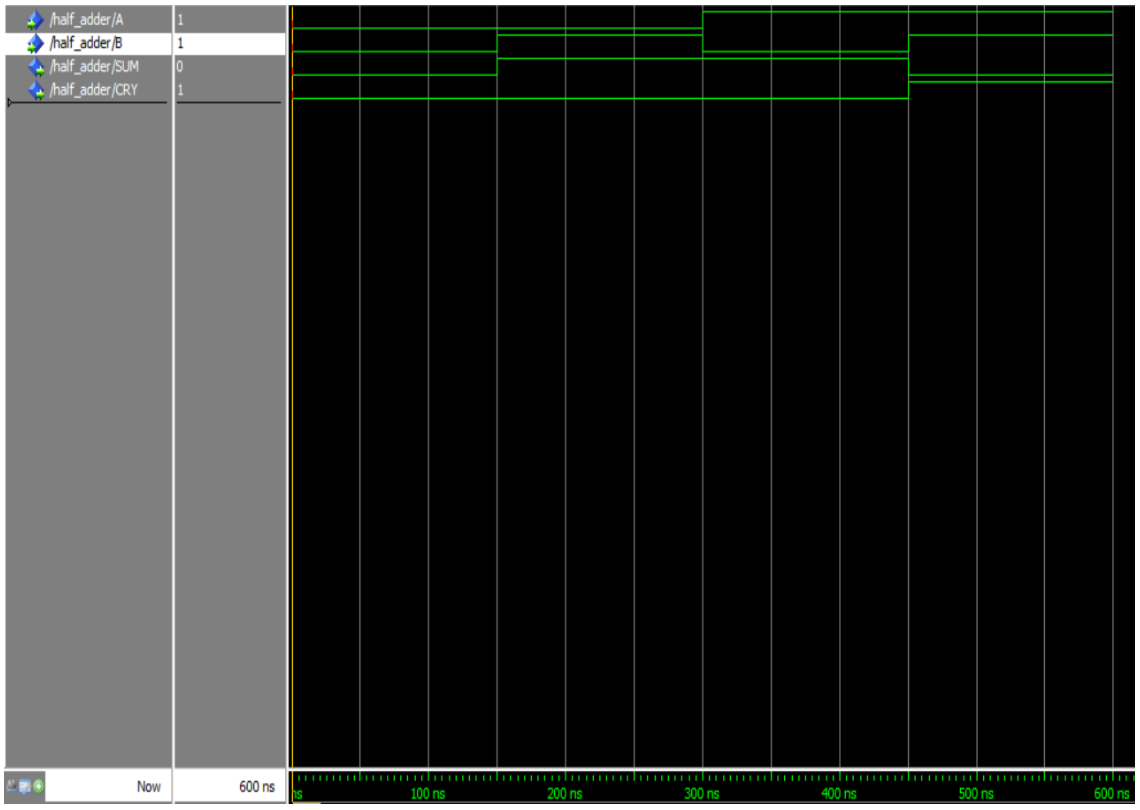


Figure 8: Half Adder Simulation

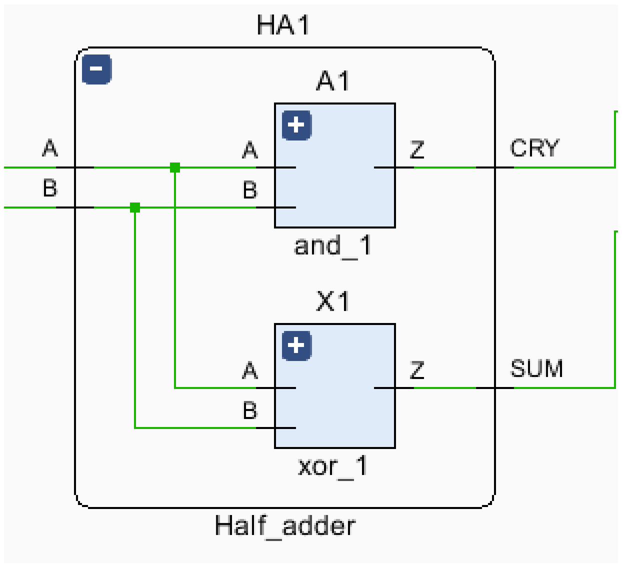


Figure 9: Half Adder Circuit

4.2 Full Adder

Full Adder is another fundamental component in the design. This component is created by cascading two half adders, and it performs the addition of three-bits instead of only two-bits as compared to the single half adder, which are the two-bit input and the carry coming from first half adder to the one after it. Also, as in the single half adder, it also has only two pin final outputs which are the sum and carry out. The full adder has more functional ability as compared to the half adder; however, the delay here is higher, which is one of the factors that should be considered in the design. Table 8 below illustrates the truth table of the Full Adder.

A	*B*	*Carry in*	*Sum*	*Carry out*
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 8: Truth Table for Full Adder

The functions of the output for full adder can be represented also in the following Boolean formula:

$$\mathbf{Sum} = (\mathbf{A} \oplus \mathbf{B}) \oplus \mathbf{Carry\ in}$$

$$\mathrm{Carry}_{\mathrm{out}} = \mathrm{A} * \mathrm{B} + \mathrm{C}(\mathrm{A} \oplus \mathrm{B})$$

In addition to that, figure 10 below illustrates the result of simulating the full adder, and figure 11 shows the circuit.

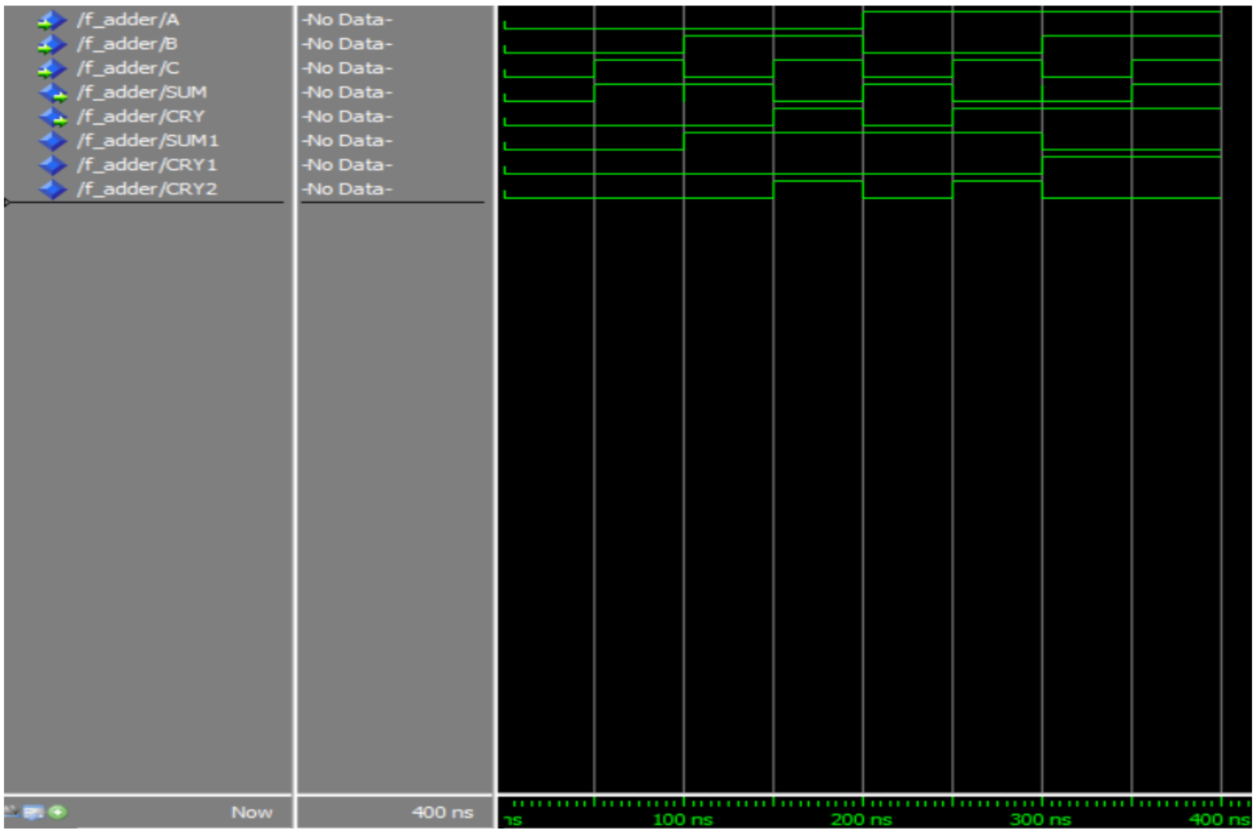


Figure 10: Full Adder Simulation

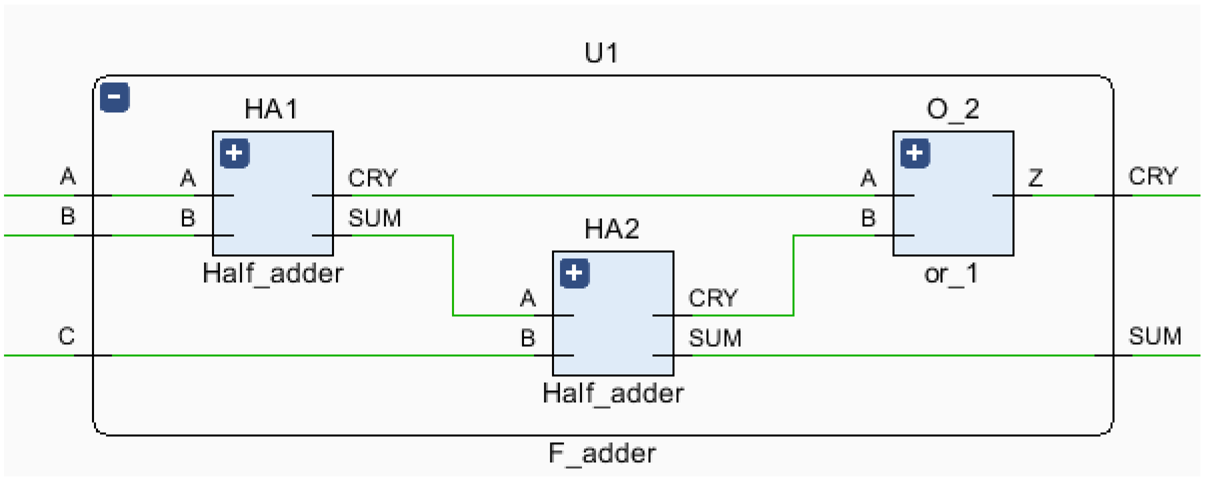


Figure 11: Full Adder Circuit

4.3 Two-to-One Multiplexer

This component main objective is to select one out of two inputs to be in the output side using another input pin for selecting. It is composed of two AND gates, one NOT gate, and one OR gate. Table 9 below represents the truth table of 2-1 multiplexer.

A	*B*	*SELECT*	*OUTPUT*
0	X	*0*	0
1	X	*0*	1
X	*0*	*1*	*0*
X	*1*	*1*	1

Table 9: Truth Table for 2-1 Multiplexer

The functions of the output for 2-1 multiplexer can be represented also in the following Boolean formula as below:

$$OUTPUT = (A * \overline{SELECT}) + (B * \overline{SELECT})$$

In addition to that, figure 12 below illustrates the result of simulating the 2-1 multiplexer, and figure 13 shows the circuit.

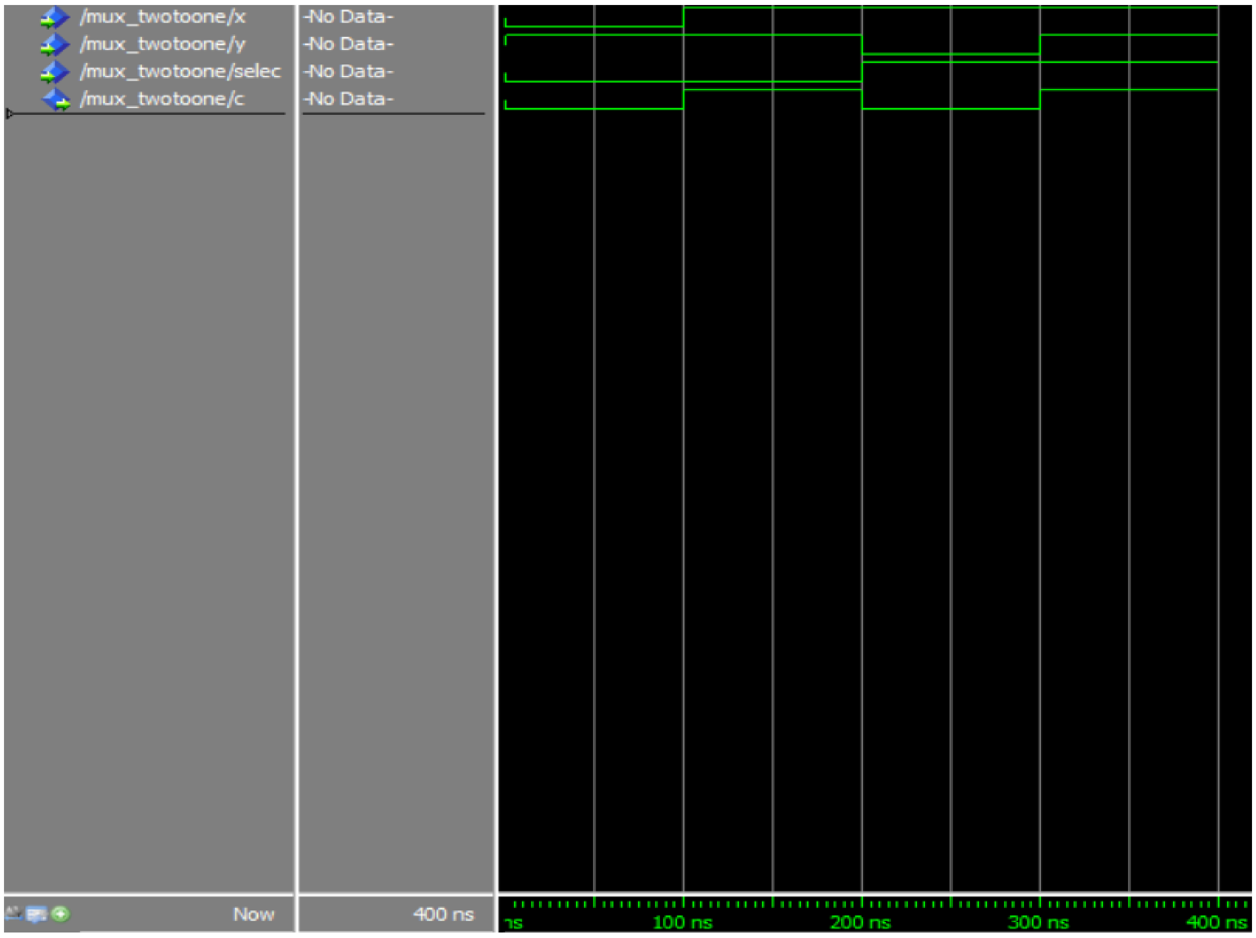


Figure 12: 2-1 Multiplexer Simulation

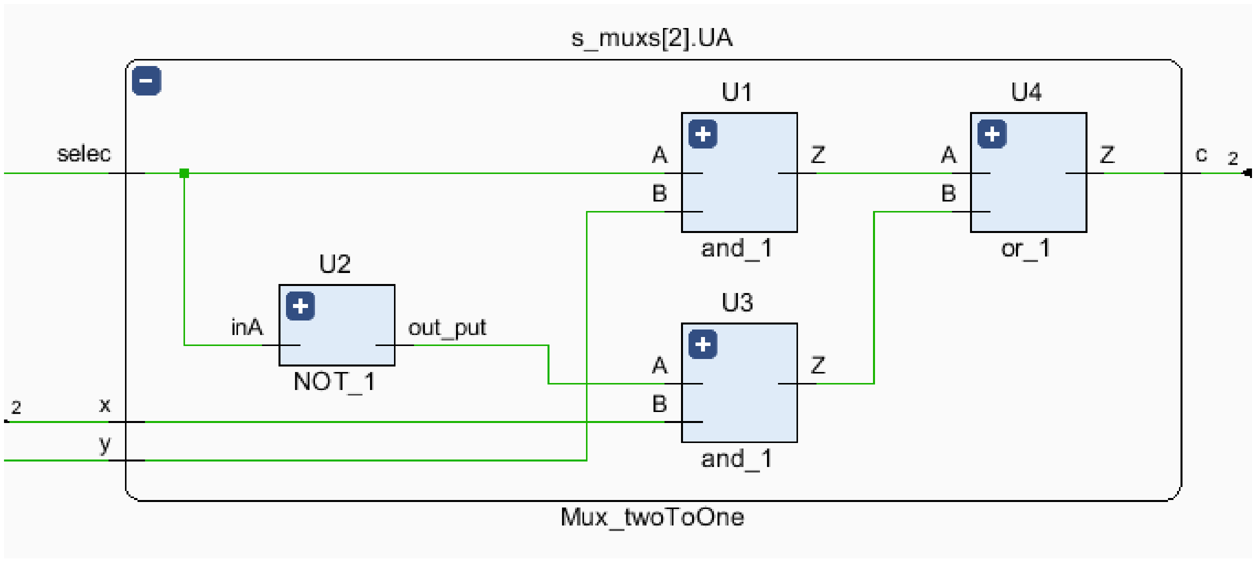


Figure 13: 2-1 Multiplexer Circuit

4.4 D-Flip-Flop

Storing bits is one of the requirements of our project. And D-Flip-Flop is essential for storing since it has the ability of storing one bit and can be the basic structure for a sequential circuit that operate based on clock. A register of n-bit needs n-D-Flip-Flop. The requirements of the arithmetic unit that any register created from the D-Flip-Flop must have a clear signal that is required for clearing the register. The structure of the D-Flip-Flop block created operates on a rising edge of the clock and active low-reset. Table 10 below explains the operation of the D-Flip-Flop, figure 14 and figure 15 shows the simulation and circuit respectively for it.

Reset	*D_in*	*CLK*	*Q*	$\bar{Q}$
0	*X*	*X*	*0*	*1*
1	*0*	*Rise*	*0*	*1*
1	*1*	*Rise*	*1*	*0*

Table 10: Truth Table of D-Flip-Flop

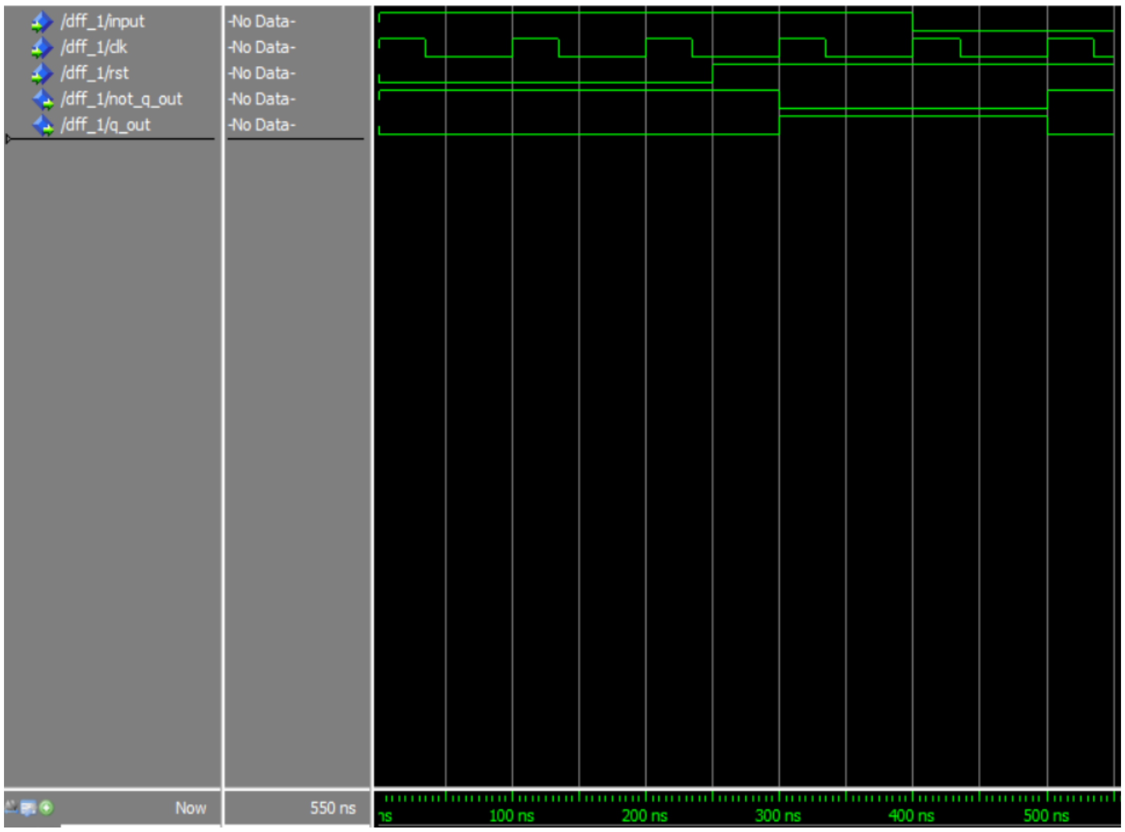


Figure 14: Simulation of D-Flip Flop

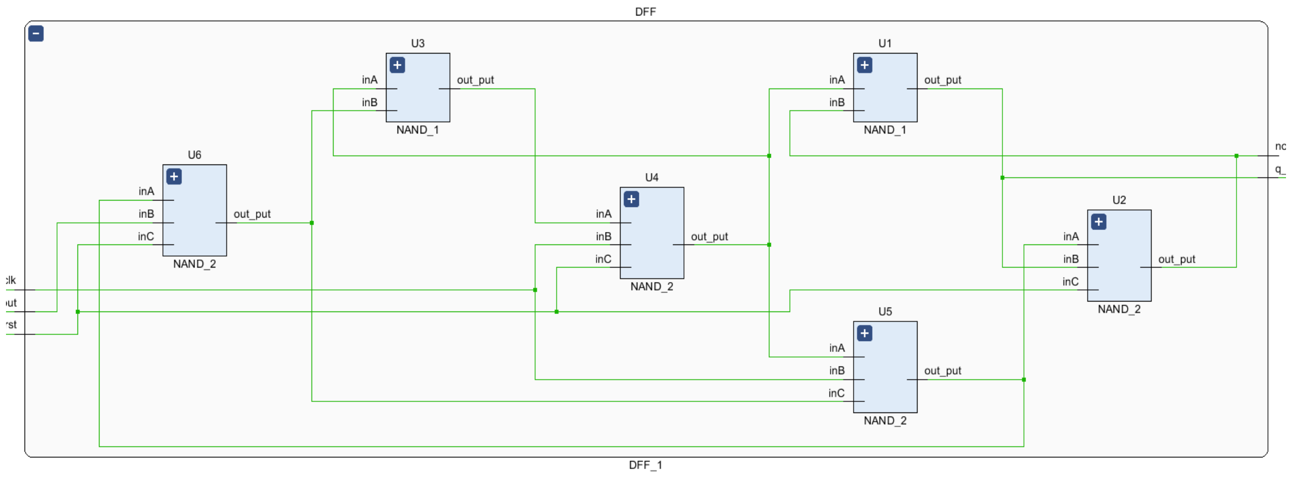


Figure 15: D-Flip Flop Circuit

4.5 Registers

Registers are one of the most essential components in DSP. It comprises of D-Flip-Flop that has the state of storing information. And in our project we have used registers to store input and output bits. In our system we used two 8-bits register to hold the data for of input (RA and RB), and there is a 16-bits registers (RZ) to hold the final result.

An n-bit register has n-bit input and output. In order to meet the requirement of our project, register is connected to a load signal that is required in the operation control; it should latch data into the internal register when there is a negative edge of a transition from 1 to 0 of the load signal. Therefore, in order to execute this step, an inverter is used to invert the clock that will allow the operation of negative edge load signal. In addition to the load signal, a clear or reset signal is also needed to clear the register from data. Figure 16 below shows an example of 8-bit register using D-Flip-Flop, figure 17 and figure 18 shows circuit and simulation test for 8-bit register respectively.

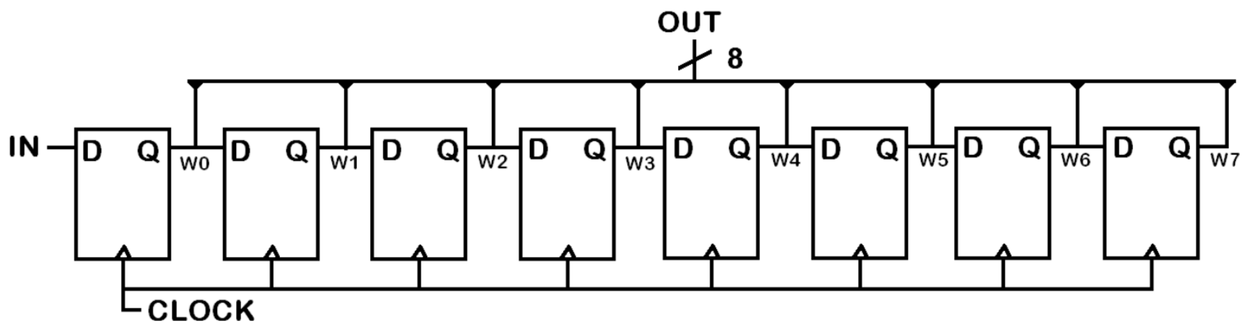


Figure 16: 8-Bits D-Flip Flop Register

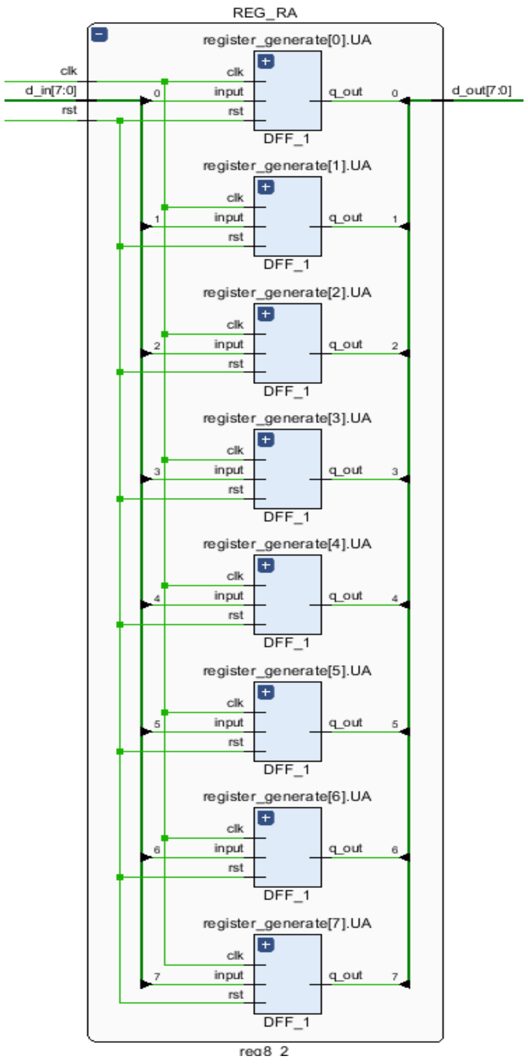


Figure 17:8-bits Register Circuit

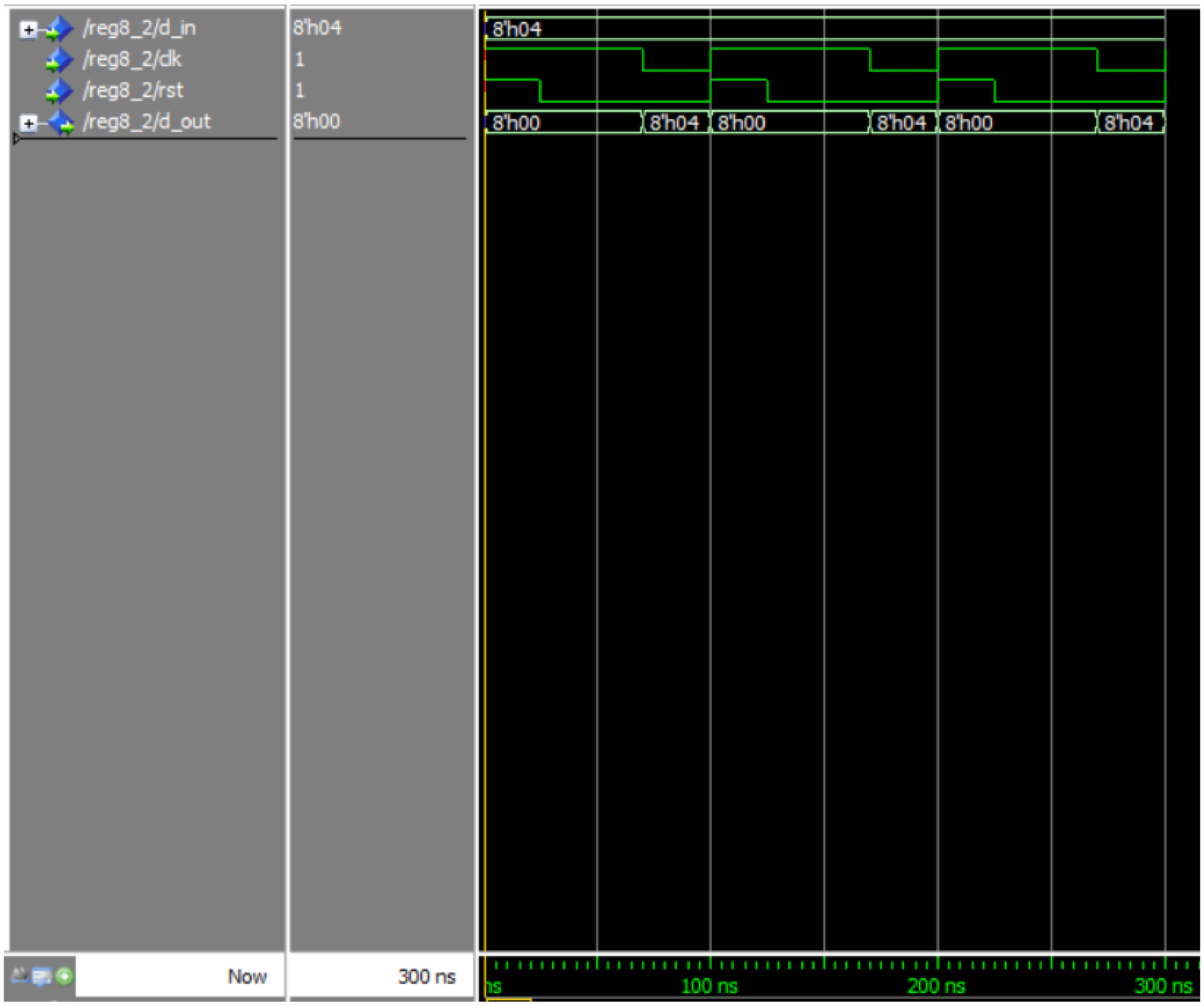


Figure 18: Simulation for 8-bits Register using D Flip Flop

5. Building the Multiplier

Array multiplier is used in the design in order to perform the first step of the arithmetic operation in order to multiply the two 8-bits inputs. This multiplier consists of three blocks which are the multiplier and multiplicand, partial products, and the sum. The block diagram below represents blocks of Array Multiplier.

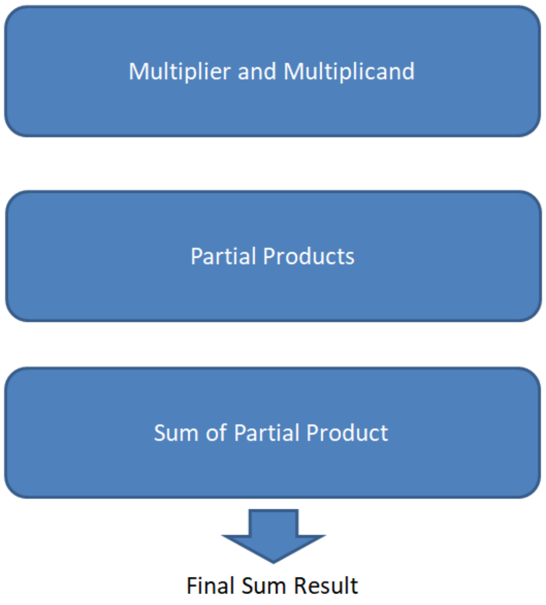


Figure 19: Block Diagram of Array Multiplier

5.1 Partial Product

The partial product is created by multiplying each bit of the multiplier by the multiplicand using AND gate. Multiplying n bit multiplier by m bit multiplicand will result in mxn partial products. And same like arithmetic multiplication operation, the partial product of each line N will be shifted by one place N+1 after each step. Figure 20 below represents the partial product generation of multiplying different 8-bits by 8-bits.

Multiplicand								A7	A6	A5	A4	A3	A2	A1	A0	
Multiplier							x	B7	B6	B5	B4	B3	B2	B1	B0	
								A7B0	A6B0	A5B0	A4B0	A3B0	A2B0	A1B0	A0B0	
							A7B1	A6B1	A5B1	A4B1	A3B1	A2B1	A1B1	A0B1		
						A7B3	A6B3	A5B3	A4B2	A3B2	A2B2	A1B2	A0B2			
					A7B3	A6B3	A5B3	A4B3	A3B3	A2B3	A1B3	A0B3				Partial Products
				A7B4	A6B4	A5B4	A4B4	A3B4	A2B4	A1B4	A0B4					
			A7B5	A6B5	A5B5	A4B5	A3B5	A2B5	A1B5	A0B5						
		A7B6	A6B6	A5B6	A4B6	A3B6	A2B6	A1B6	A0B6							
	A7B7	A6B7	A5B7	A4B7	A3B7	A2B7	A1B7	A0B7								

Figure 20: Partial Product Arrangement

5.2 Sum of Partial Product

After the creation of partial product, the sum of them should be executed in order to get the final product result. The sum of partial product can be done using half adders and full adders. In our design we have used 56 full adders to perform the addition of the partial product result. Figure 21 below represents the sum of partial product distribution connection of 56 Full Adders. Also, figure 22 is showing the circuit with connection for addition.

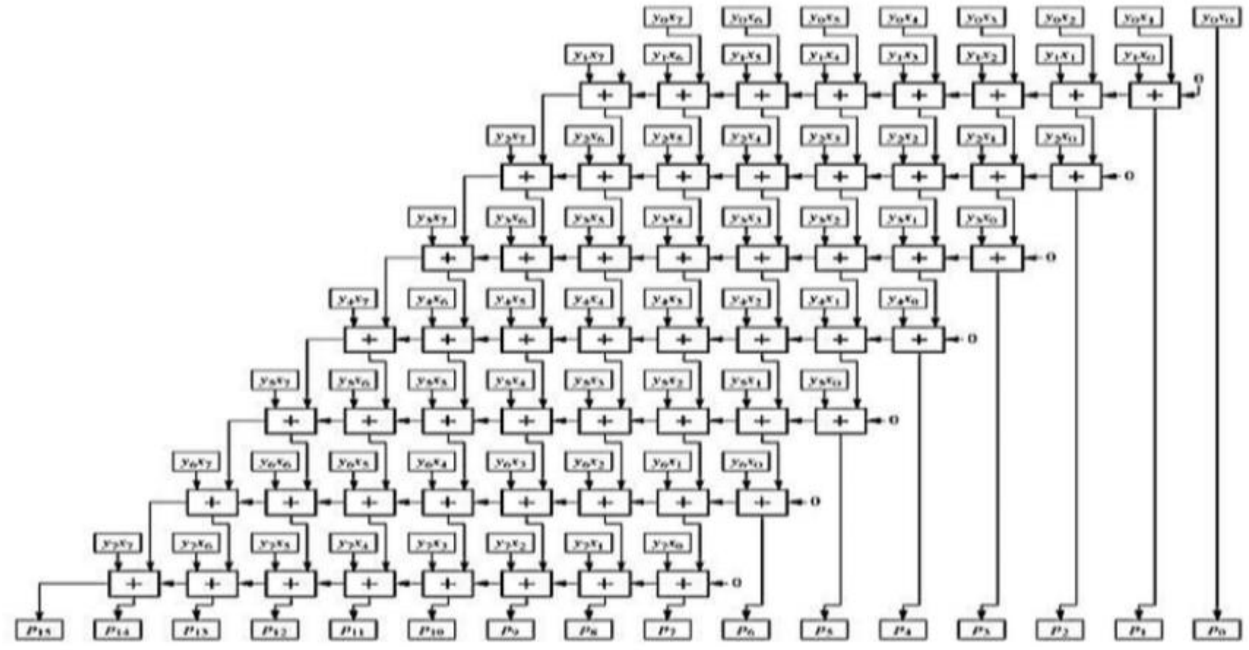


Figure 21: Sum of Partial Product with Full Adders

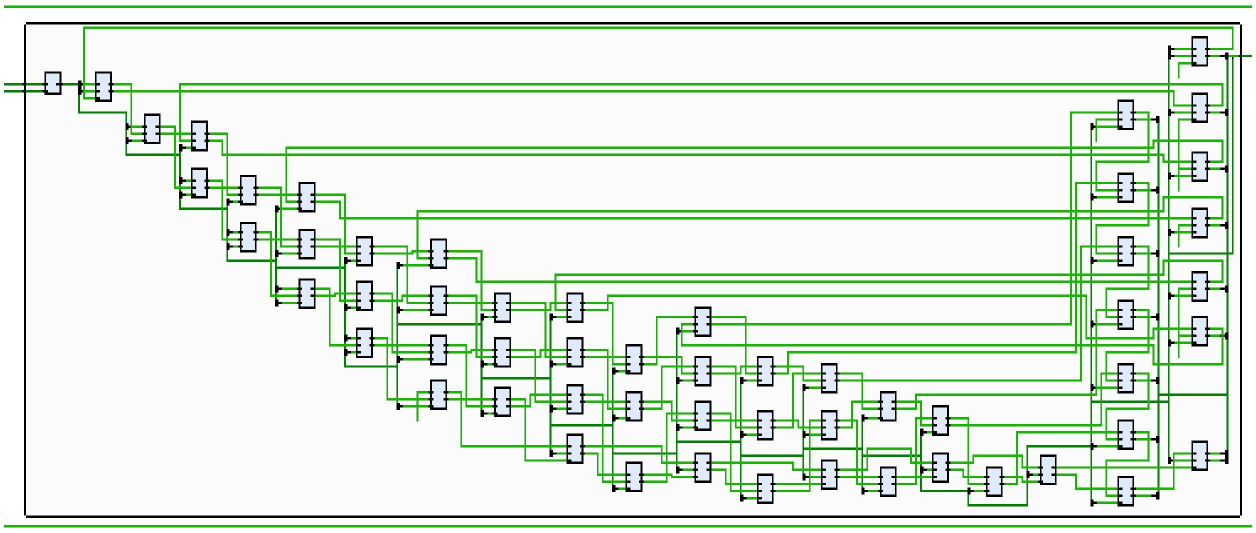


Figure 22: Array Multiplier Circuit

6. Building the Shifter

Shifter is needed in our system in order to perform the second step of the arithmetic operation, which is the multiplication by ¼. In order to perform this operation, the multiplication result from previous step should be shifted by two bits to the right. Therefore, the last two shifted bits will represent an after fixed point bits, and the other bits will be representing the result that is before the fixed point. In our design, we have used Barrel Shifter in order to perform a structural shifting by two bits to the right.

6.1 Creating Barrel Shifter

In order to create a Barrel Shifter, rows of 2-1 Multiplexers are needed. And in our design for shifting 16-bits by two-bits to the right, we have used three rows of Multiplexers each of 16-mux. Based on the select signal for the multiplexer, it will propagate the needed signals from one row to the next one in the required path for shifting. But since in our case the number of shifts is fixed, then a constant select signal is used for the multiplexers. Figure 23 below represents the block diagram connection of the barrel shifter.

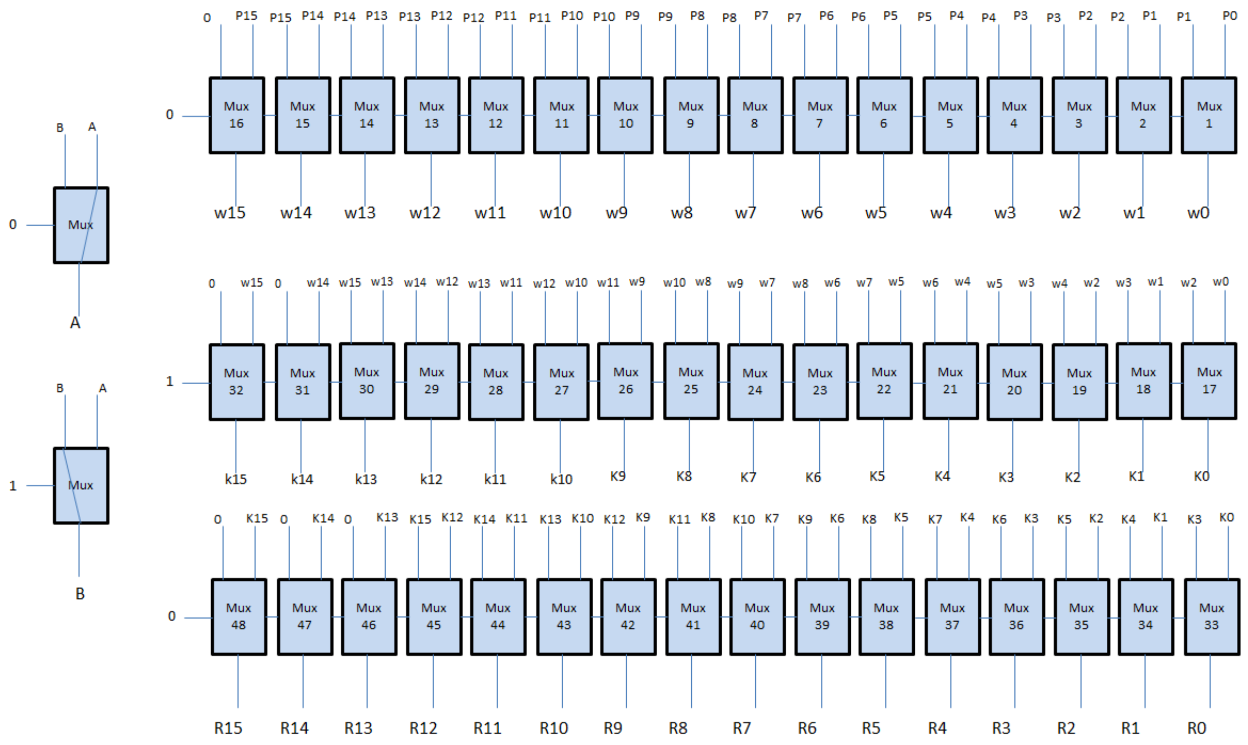
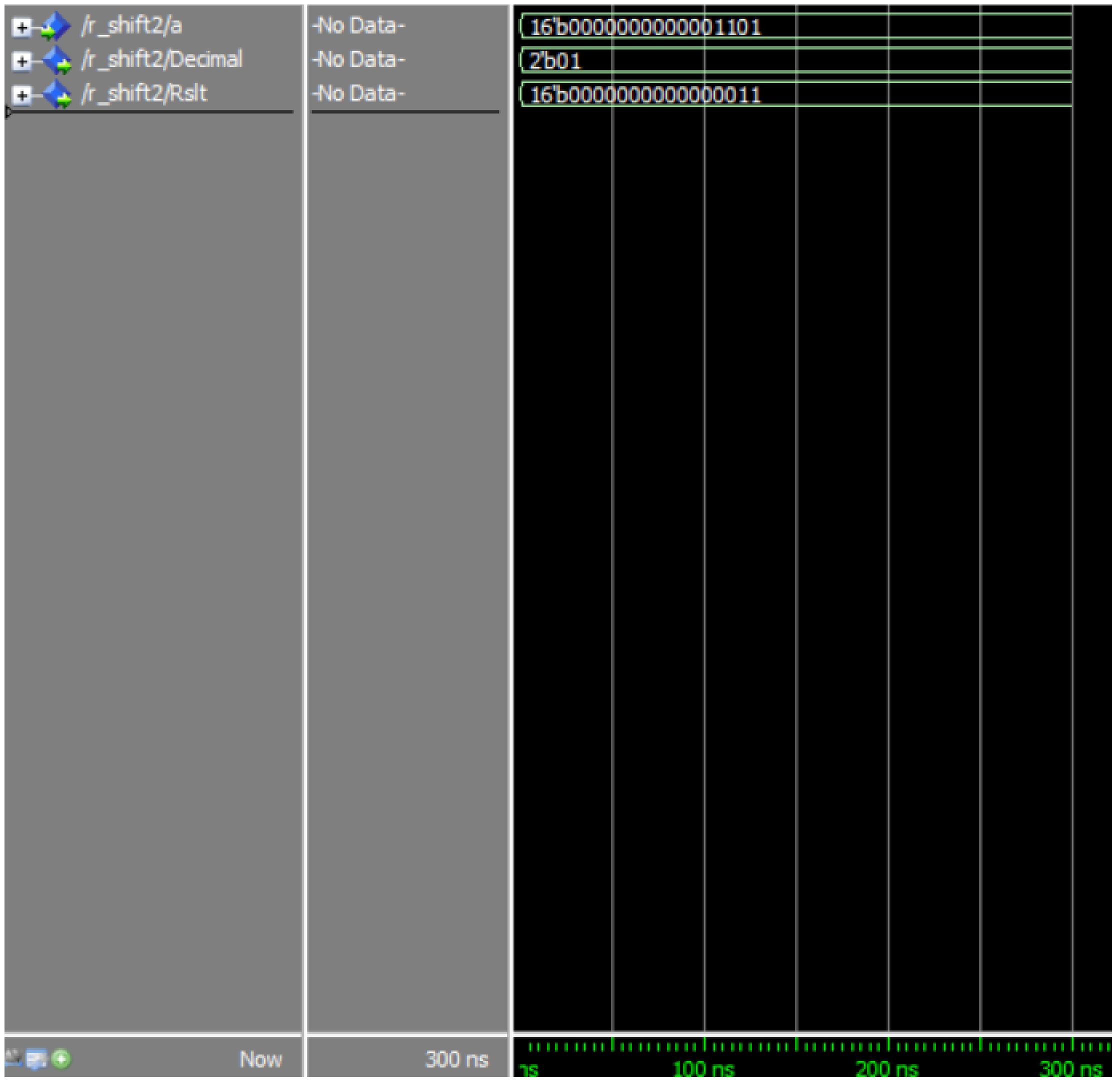


Figure 23: Barrel Shifter Blocks

In addition to that, figure 24 below shows a test example simulation for shifting; the decimal output represents the first shifted 2-bits of what after a fixed point, and figure 25 shows the circuit.



```
/r_shift2/a        -No Data-   16'b0000000000001101
/r_shift2/Decimal  -No Data-   2'b01
/r_shift2/Rslt     -No Data-   16'b0000000000000011
```

Figure 24: Test Simulation for Barrel Shifter

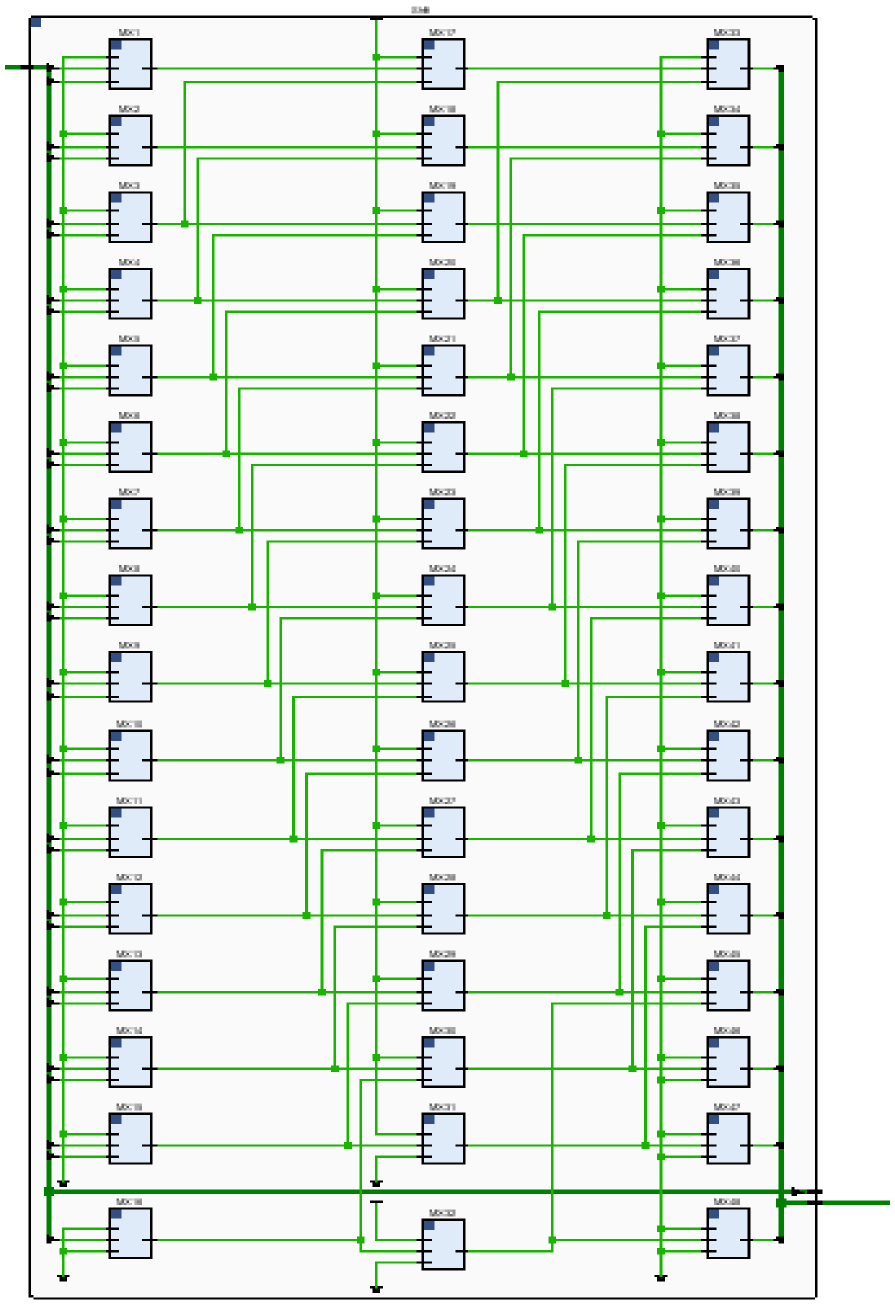


Figure 25: Barrel Shifter Circuit

7. Building the Adder

Carry Select Adder (CSA) is the type of adder used to design the addition of the shifted result with 1. In order to build the final Carry Select Adder, firstly a Ripple Carry Adder should be created.

7.1 Creating Ripple Carry Adder Block

Ripple Carry Adder (RCA) is the essential structure in the Carry Select Adder. For this system, a 4-bit carry ripple adder is created. For a ripple carry adder of n-bits, it will need n Full Adders that are cascaded in order to perform the addition of n bits. Figure 26 below represents 4-bit ripple carry adder. And as the figure shows, for every single Full Adder's carry$_{out}$ will be the carry$_{in}$ of the next one to finally obtain the final sum.

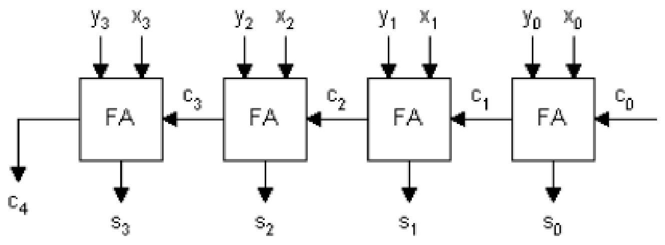


Figure 26: 4-bit RCA

A test simulation is shown in figure 27 below for 4-bit Ripple Carry Adder, and figure 28 shows the circuit.

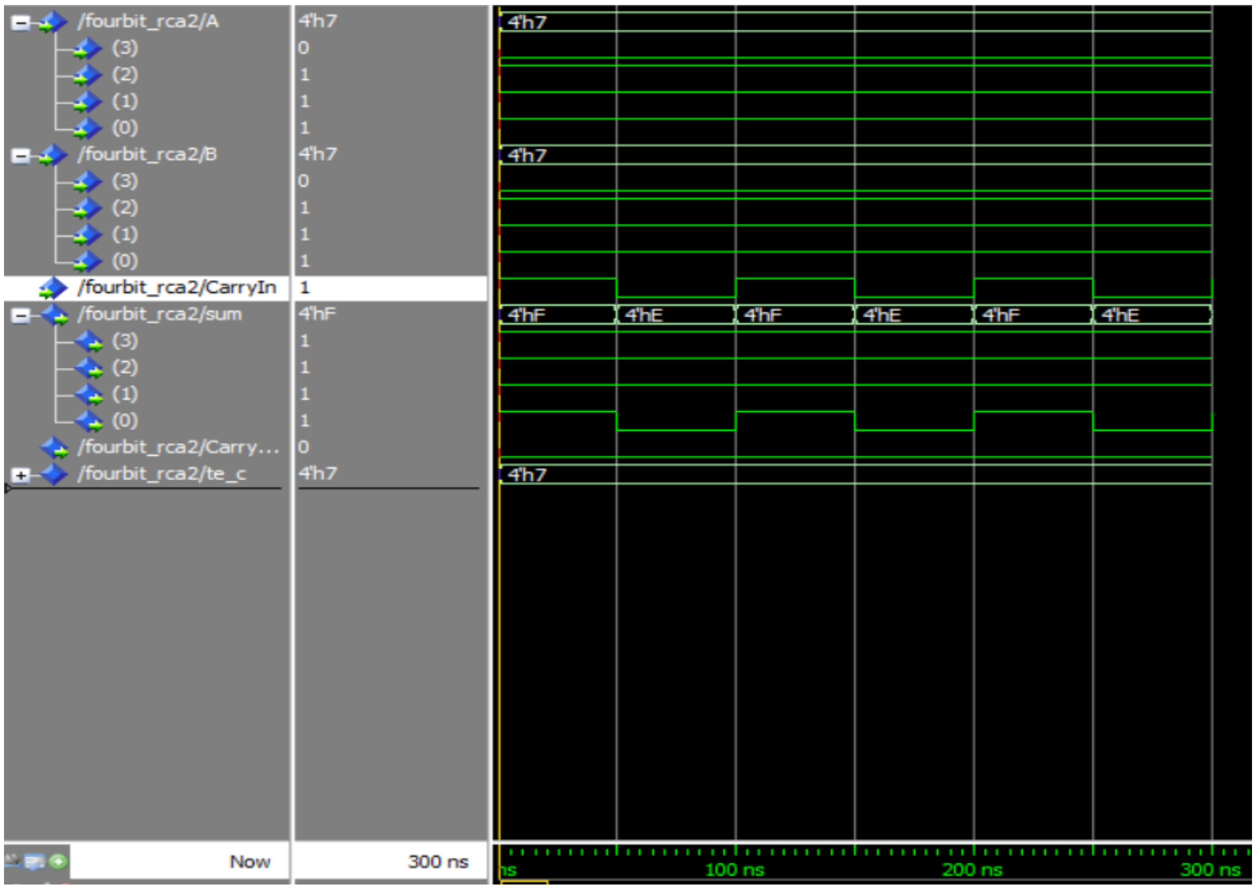


Figure 27: Test Simulation of 4-bit RCA

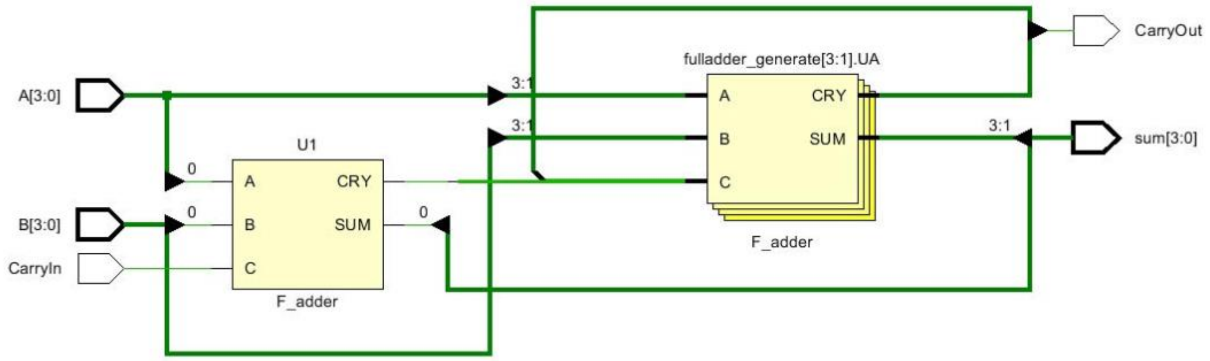


Figure 28: 4-bit RCA circuit

On the other hand, in our desired arithmetic of operation is to add the shifted bits with 1. Adding the first bit of value 1 will be done in the final creation of the Carry Select Adders; however, the remaining sequence will be always of logic ‘0’ that will be added with each bit of the shifted result created from previous step, more details will be shown in upcoming sections regarding the addition of '0'. Figure 29 and figure 30 below show the test simulation and circuit of RCA considering one input of zero value.

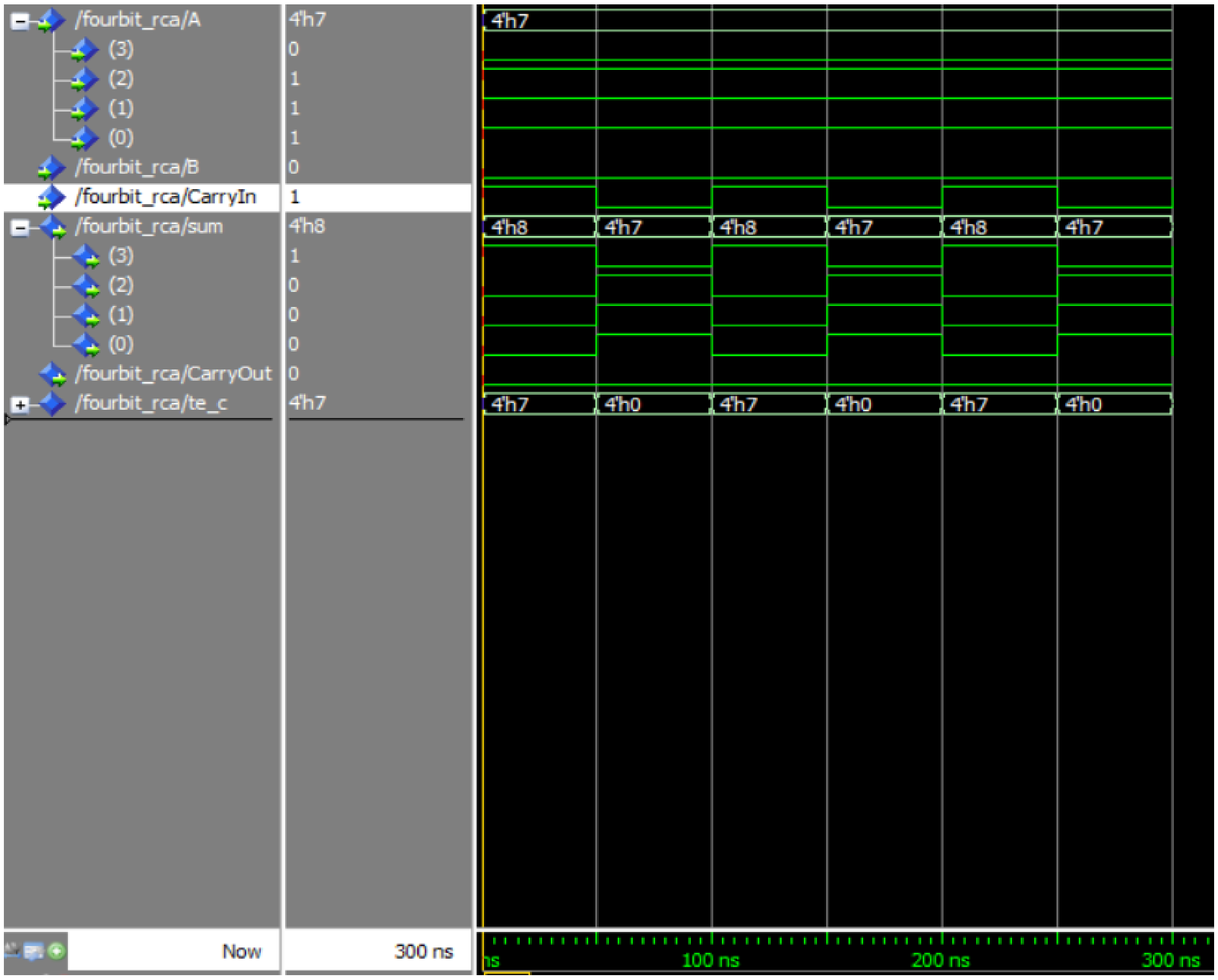


Figure 29: 4-bit RCA with one input as zero

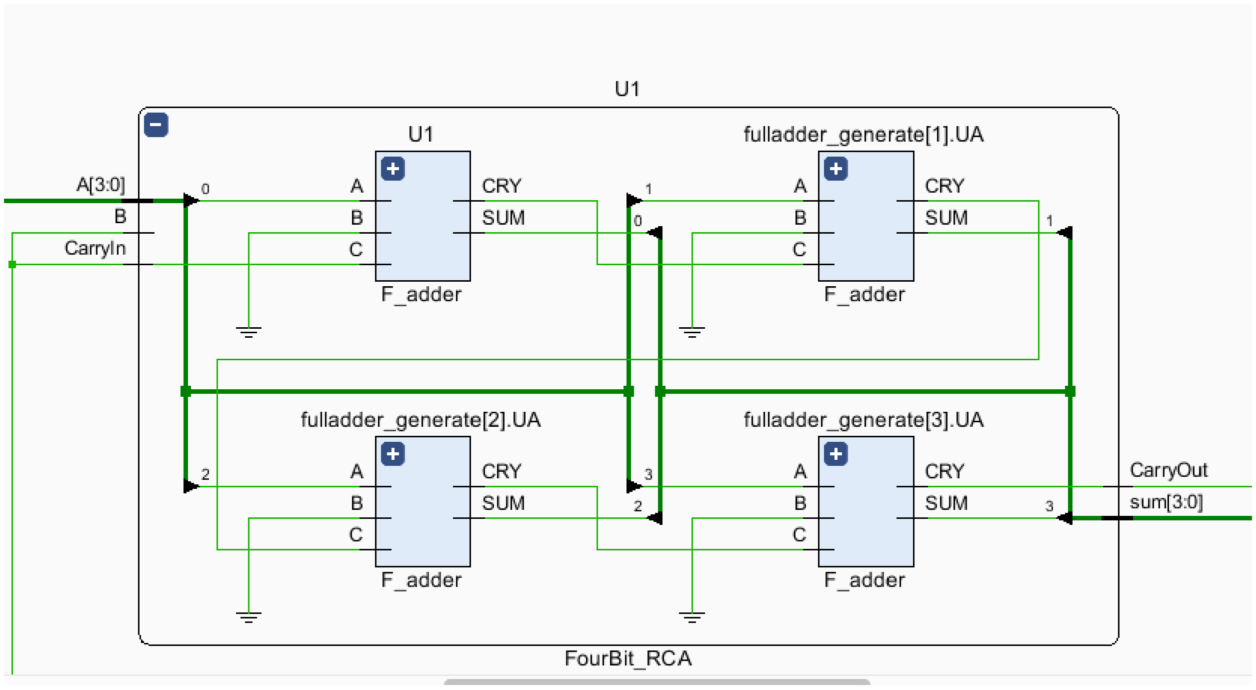


Figure 30: 4-bit RCA with one input as zero

7.2 Creating Carry Select Adder Block

Carry select adder (CSA) is composed of several consecutive stages of ripple carry adder that adds bits using cascaded full adders. At every stage of ripple carry adder in CSA, there will be two pairs of ripple carry adders that calculate the sum and carry; one pair is calculating them based on a carry$_{in}$ of logic ‘0’ while the other with logic ‘1’. Once the carry propagates from one stage to the next stage, depending on that value of propagated carry, MUX is used to choose the correct calculation to create the required output. In other words, this strategy will help in avoiding the wait of the carry$_{in}$ to calculate the sum since in this case the sum have already been calculated and just need to be chosen carry reaches the choosing stage. Figure 31 below illustrates a 4-bit CSA, figure 32 and figure 33 represent a test simulation for 4-bit CSA and the circuit respectively.

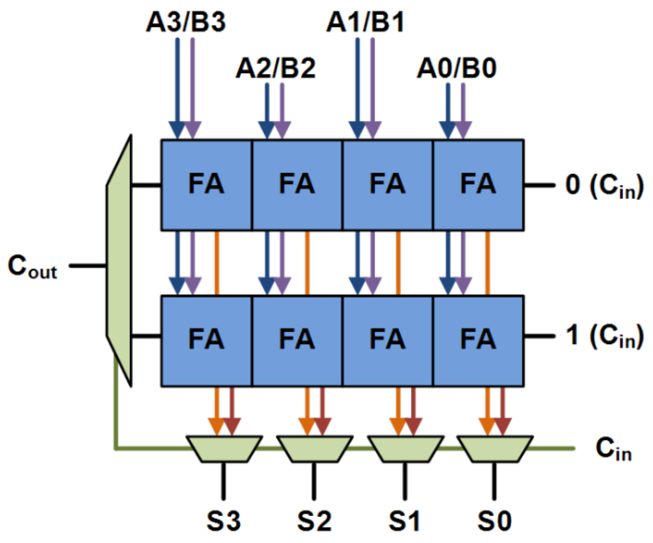
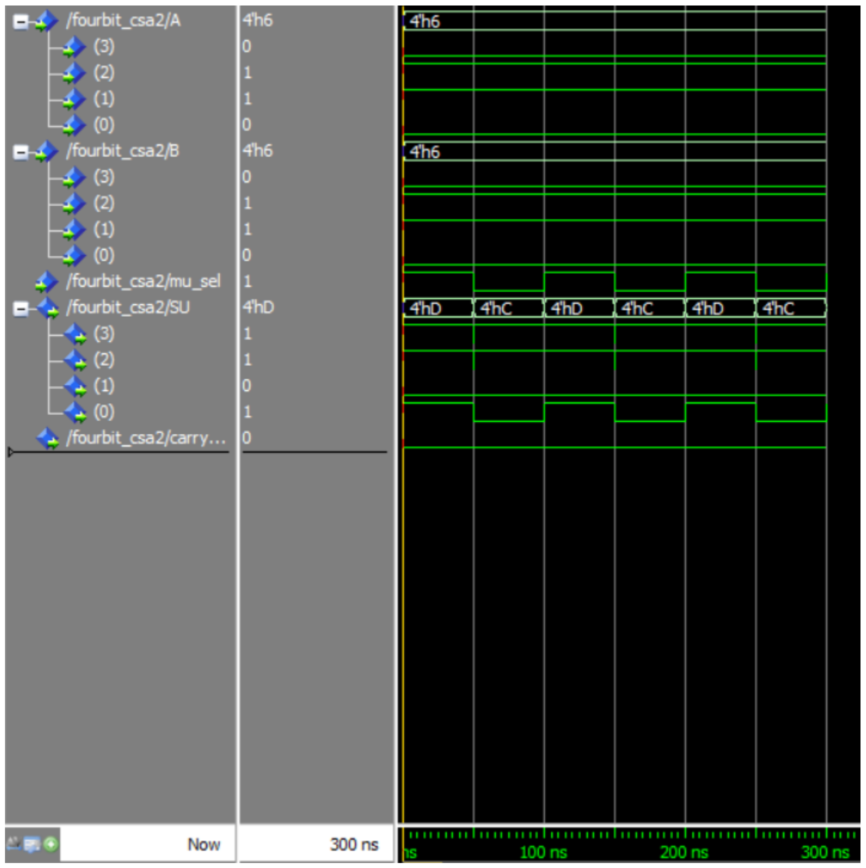


Figure 31: 4-bit CSA



```
/fourbit_csa2/A        4'h6
  (3)                  0
  (2)                  1
  (1)                  1
  (0)                  0
/fourbit_csa2/B        4'h6
  (3)                  0
  (2)                  1
  (1)                  1
  (0)                  0
/fourbit_csa2/mu_sel   1
/fourbit_csa2/SU       4'hD
  (3)                  1
  (2)                  1
  (1)                  0
  (0)                  1
/fourbit_csa2/carry... 0
Now                    300 ns
```

Figure 32: 4-bit CSA test simulation

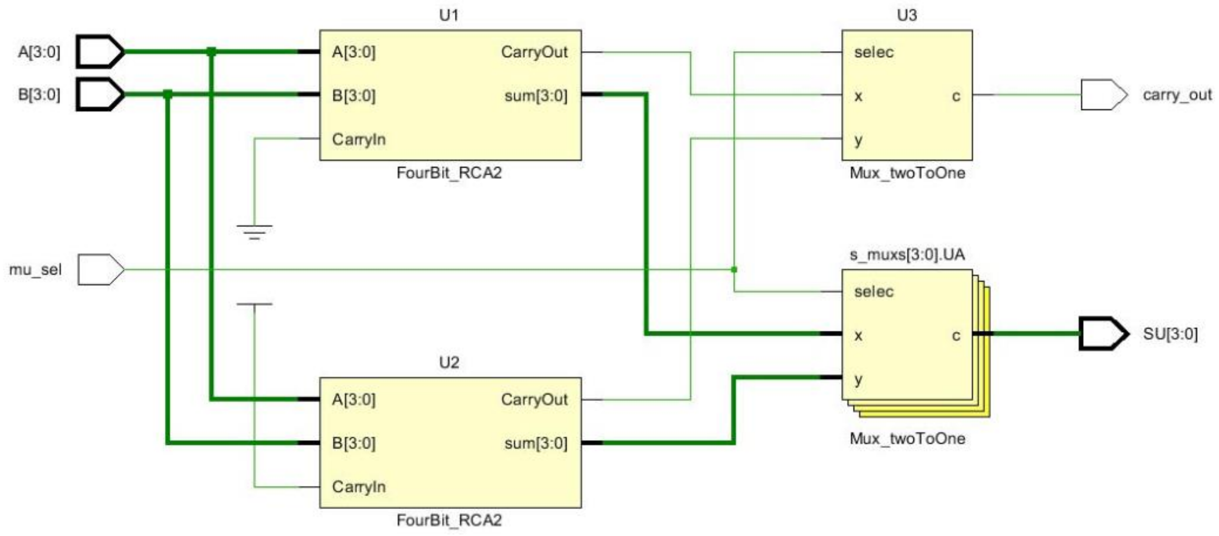


Figure 33: 4-bit CSA Circuit

Moreover, the same scenario for the 4-bit CRA goes for CSA block, in order to execute the arithmetic operation of the addition correctly, also always logic ‘0’ should be added with each bit of the shifted result created from previous step. Figure 34 and figure 35 below illustrate a simulation test for 4-bit CSA with one input of zero and the composed circuit respectively.

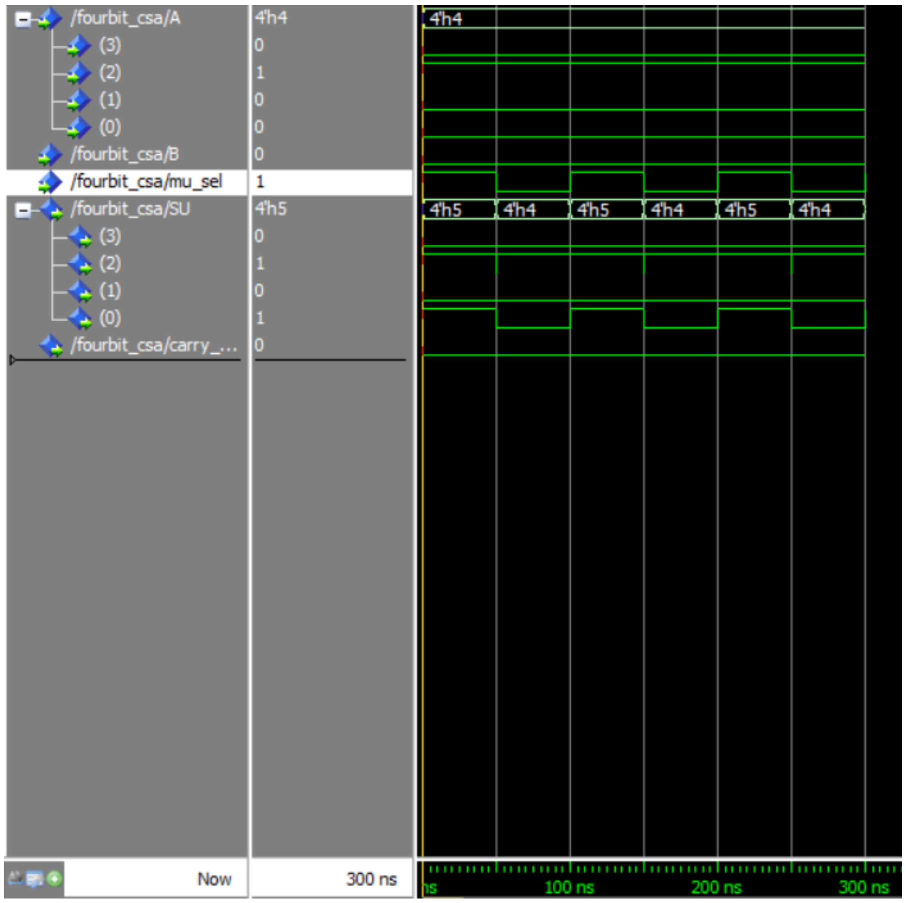


Figure 34: 4-bit CSA with one input as zero

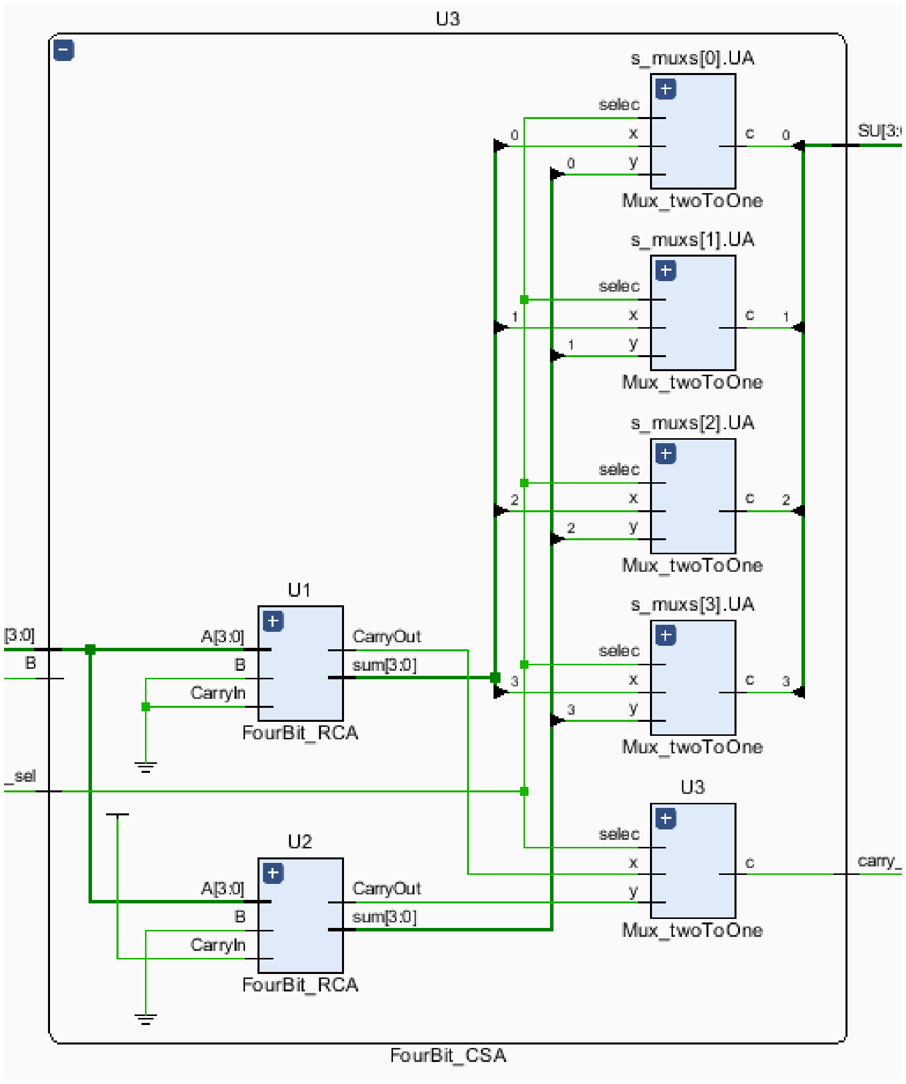


Figure 35: 4-bit CSA Circuit with one input as zero

Furthermore, the Carry Select Adder can be executed in different arrangement and sequence. And choosing the ideal arrangement is very crucial in order to avoid the delay as much as possible. The best way for connection is to consider an equal number of divisions for the stages of CSA in order to minimize the delay. As an example, the delay of an 8-bit CSA is calculated using the formula below:

$$t_{CSA} = 4 * t_{FA} + t_{mux}$$

In our design, based on equation above, we have made different distribution possibilities for further analysis to calculate the delay of 16-bit CSA, and since it can be assumed that the delay of the multiplexer is less than that of the full adder, then an assumption was made that the delay of MUX to be half of that for full adder in order to get the following possibilities:

1- $4x4x4x4$: $t_{CSA} = 4t_{FA} + 1.5t_{FA} = 5.5t_{FA}$
2- $4x3x3x3x3$: $t_{CSA} = 4t_{FA} + 2t_{FA} = 6t_{FA}$
3- $6x6x6$: $t_{CSA} = 6t_{FA} + 1t_{FA} = 7t_{FA}$
4- $8x8$: $t_{CSA} = 8t_{FA} + 0.5t_{FA} = 8.5t_{FA}$

Therefore, the optimal arrangement considered is to take the arrangement with the lowest delay which is by considering the first case.

7.3 Creating Final Carry Select Adder

After deciding the distribution methodology, then the final connection of the carry select adder will comprise of 4-bit ripple carry adder followed by three 4-bit CSA. However, for the first 4-bit ripple carry adder, we designed for sake of simplicity the first adder of it to be half adder instead of full adder since there is no carry$_{in}$ at this stage. Moreover, first half adder will be adding the first bit from shifted result with bit ‘1’ which is required by the arithmetic operation, and the same case then applies when RCA and CSA blocks were built, the remaining inputs will all be having one zero. Figure 36 below shows the design connection for the full CSA where bit ‘R’ represents the bits of shifted result.

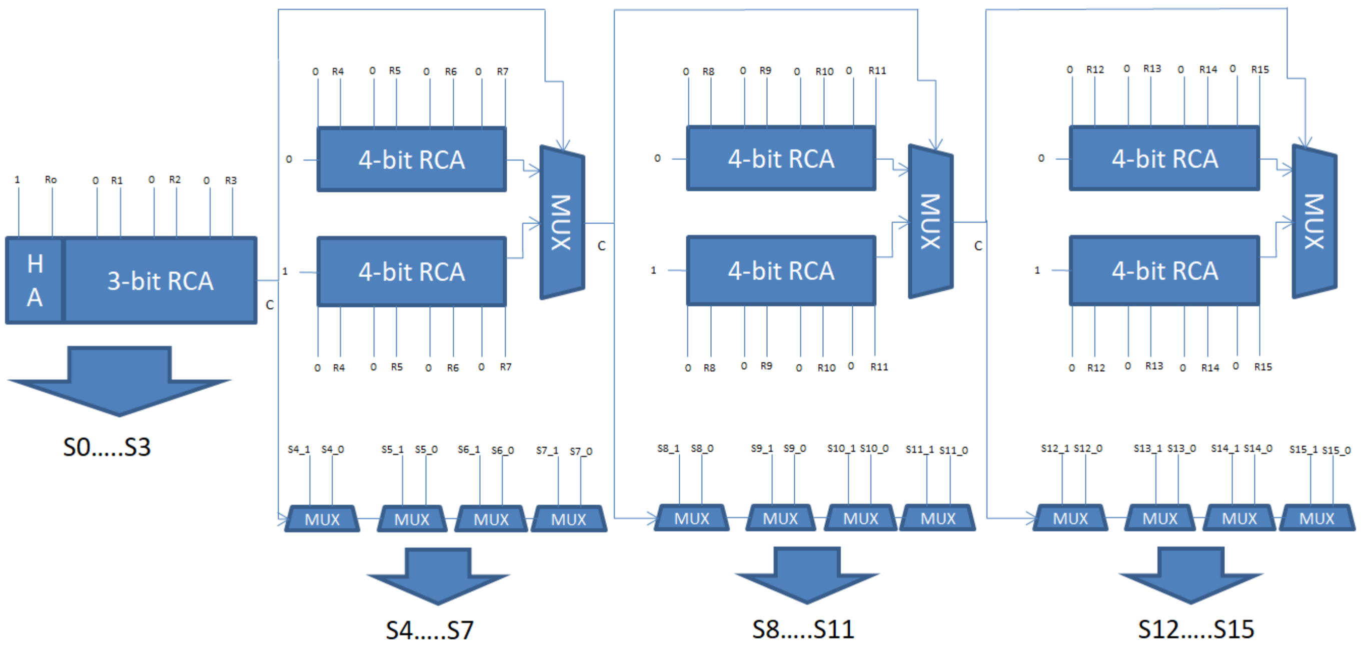


Figure 36: 16-bit CSA Blocks

Figure 37 below shows the test simulation for random number addition with ‘1’ of CSA, and figure 38 shows the circuit. The simulated file code in appendix A show more details about the connection.

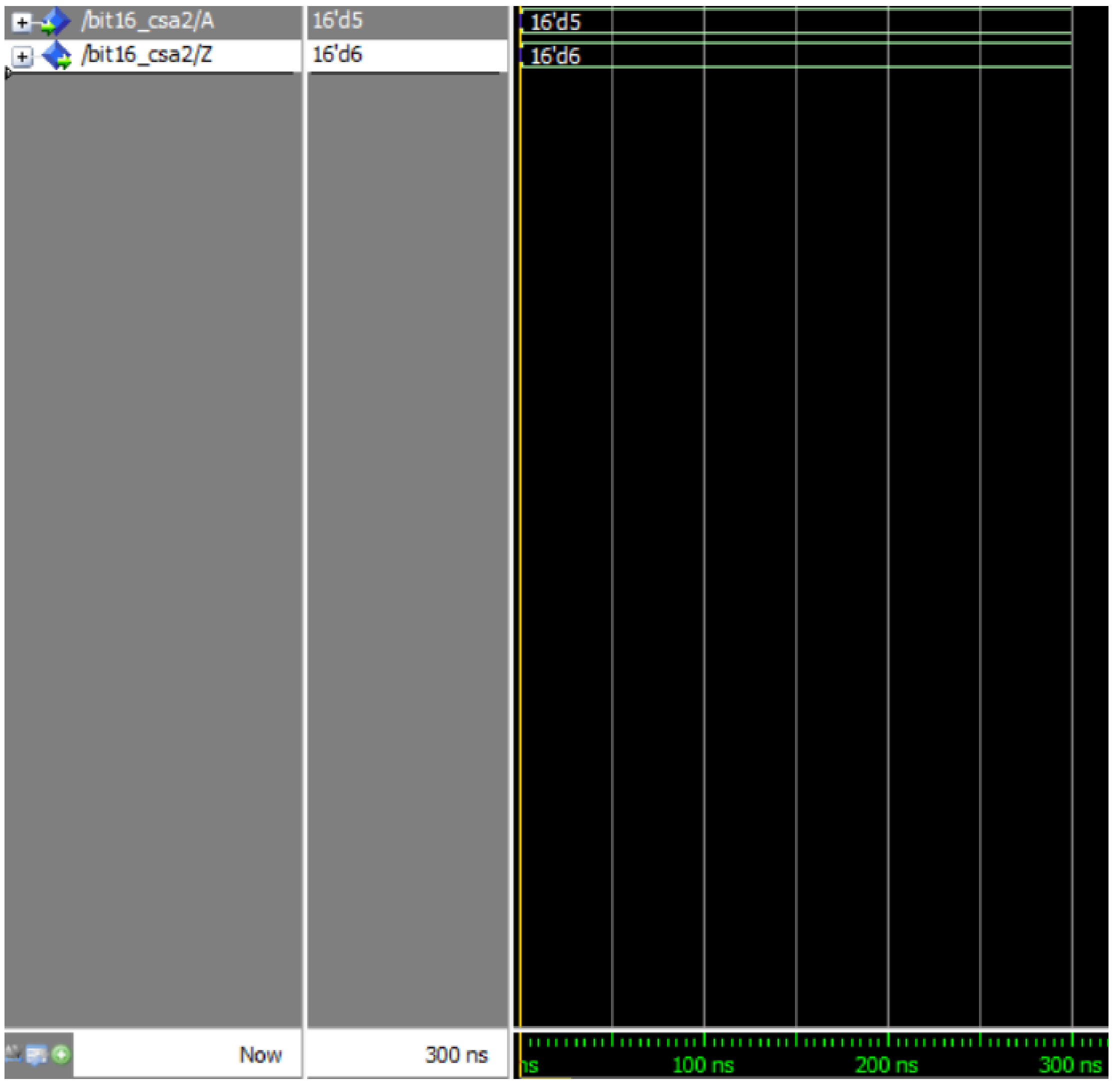


Figure 37: Test Simulation for 16-bit CSA

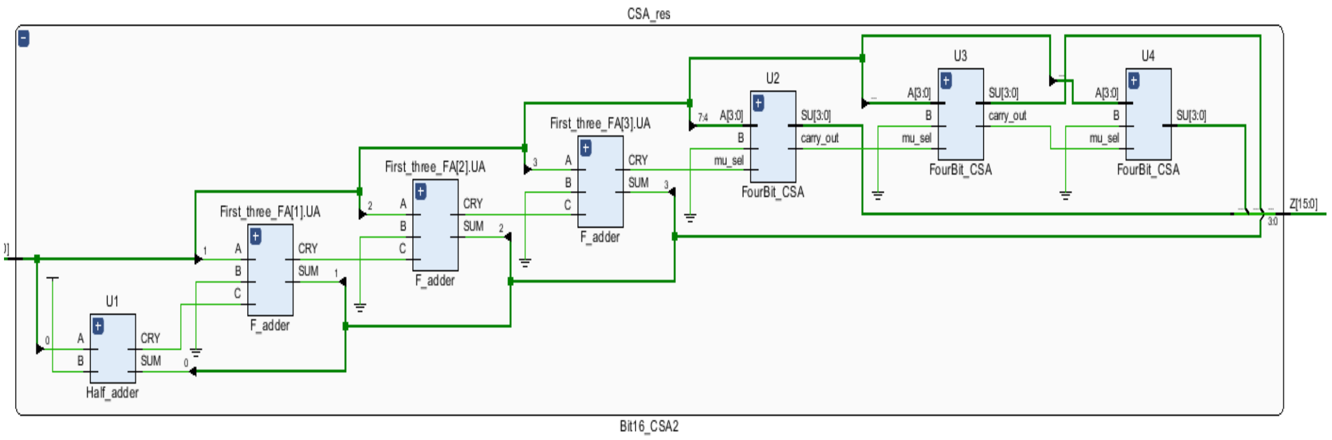


Figure 38: 16-bit CSA Circuit

8. Final Simulations

8.1 Test Bench

The test bench is a condition in VHDL that contributes in part of a complete simulation environment for the analyzed system (unit under test, UUT). A quick test bench simulation for a test sequence numbers is shown in figure 39 below, and it performs as per the requirements that will be discussed in depth in the next section.

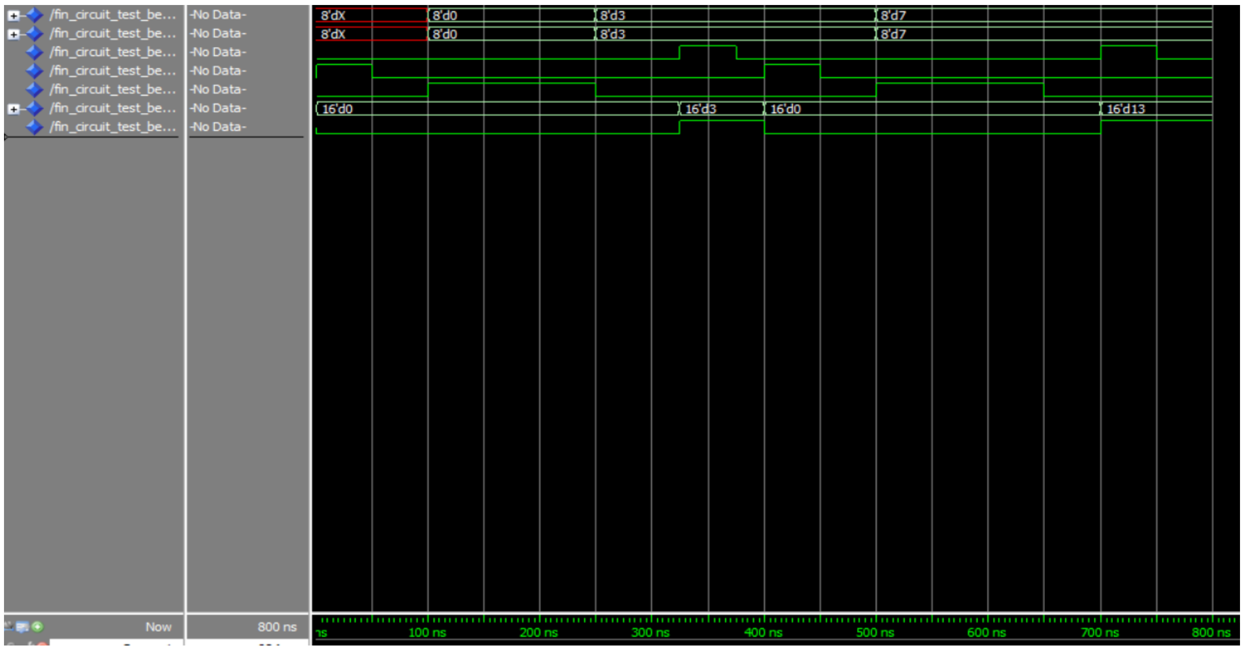


Figure 39: Test Bench for Final Circuit

8.2 Results and Discussion

After constructing the blocks required the project, the final arithmetic unit has been built, and all final requirements are obtained. There are two 8-bits operand registers (RA and RB) that will latch the operands into the register when there is a transition for LOAD signal from 1 to 0. It is also obtained that the CLR signal is resetting the registers and clearing the values. In addition to that, the END_FLAG signal that gives logic ‘1’ to show the end of the each operation by inverting the LOAD signal and feed it to the D Flip Flop. Along with the final result Z, the final simulation is including the decimal that is represented in binary to show the value of the decimal in the arithmetic operation. The list below shows the tested random decimal numbers along with their result.

- $\frac{1}{4}(50x50) + 1 = 626$
- $\frac{1}{4}(102x88) + 1 = 2245$
- $\frac{1}{4}(30x31) + 1 = 233.5$
- $\frac{1}{4}(79x84) + 1 = 1660$
- $\frac{1}{4}(126x126) + 1 = 3970$
- $\frac{1}{4}(99x101) + 1 = 2500.75$

Figures below illustrate the simulations of above list showing the parameters and the final implemented circuit. The final implemented code can be found in the appedix A.

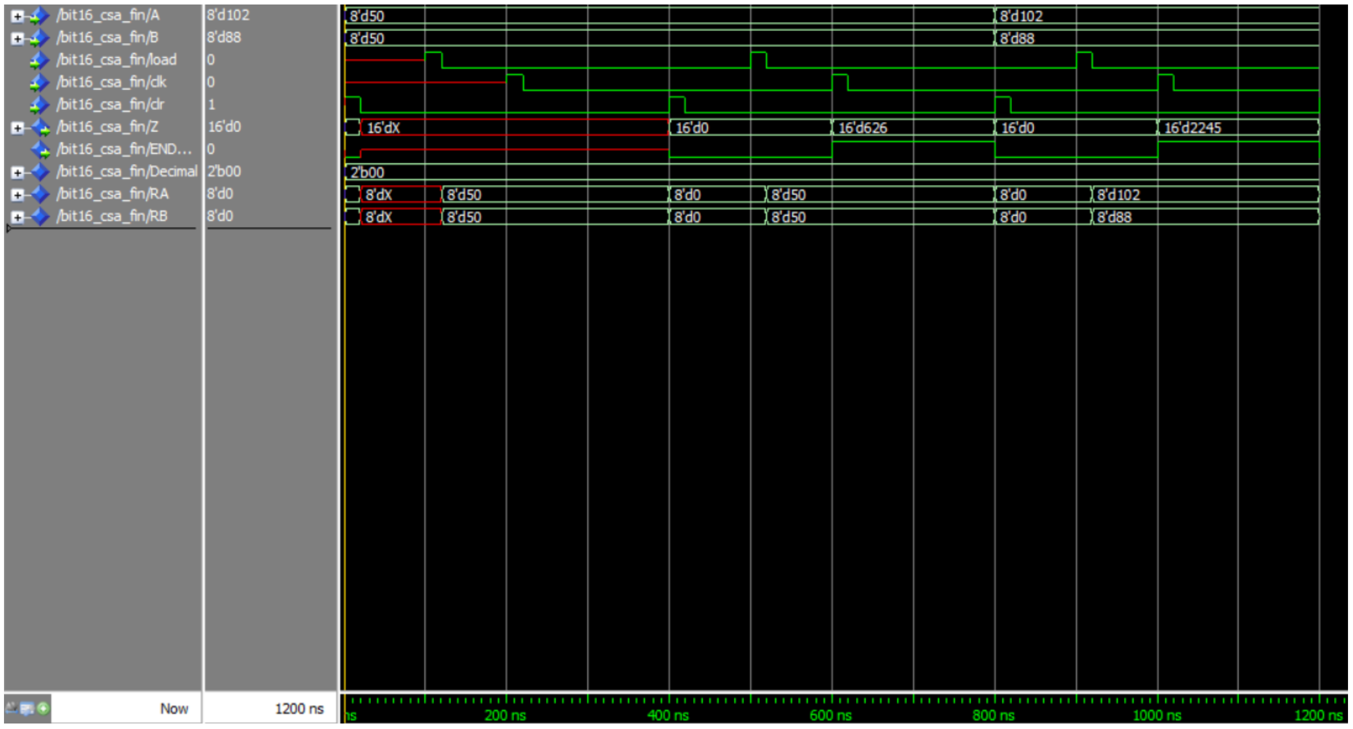


Figure 40: Test Results 1

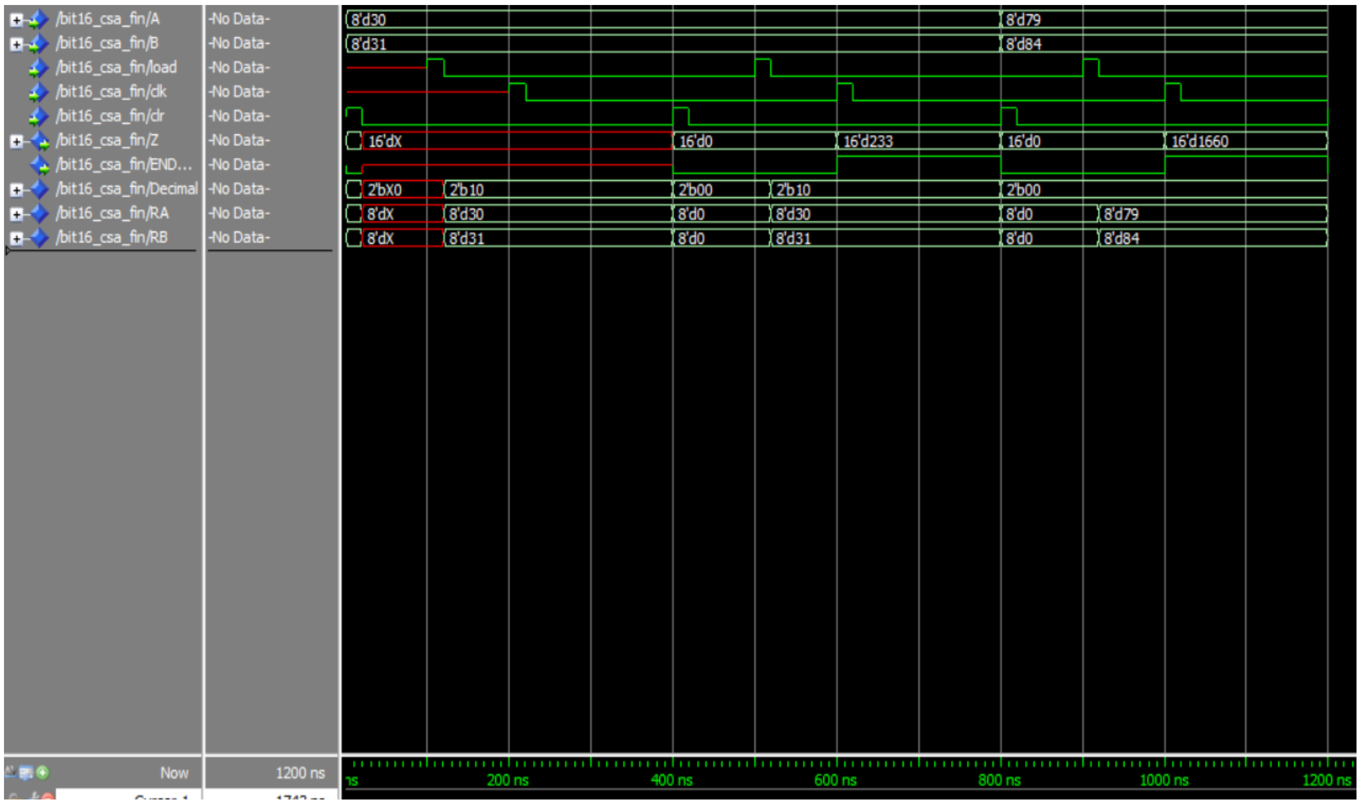


Figure 41: Test Results 2

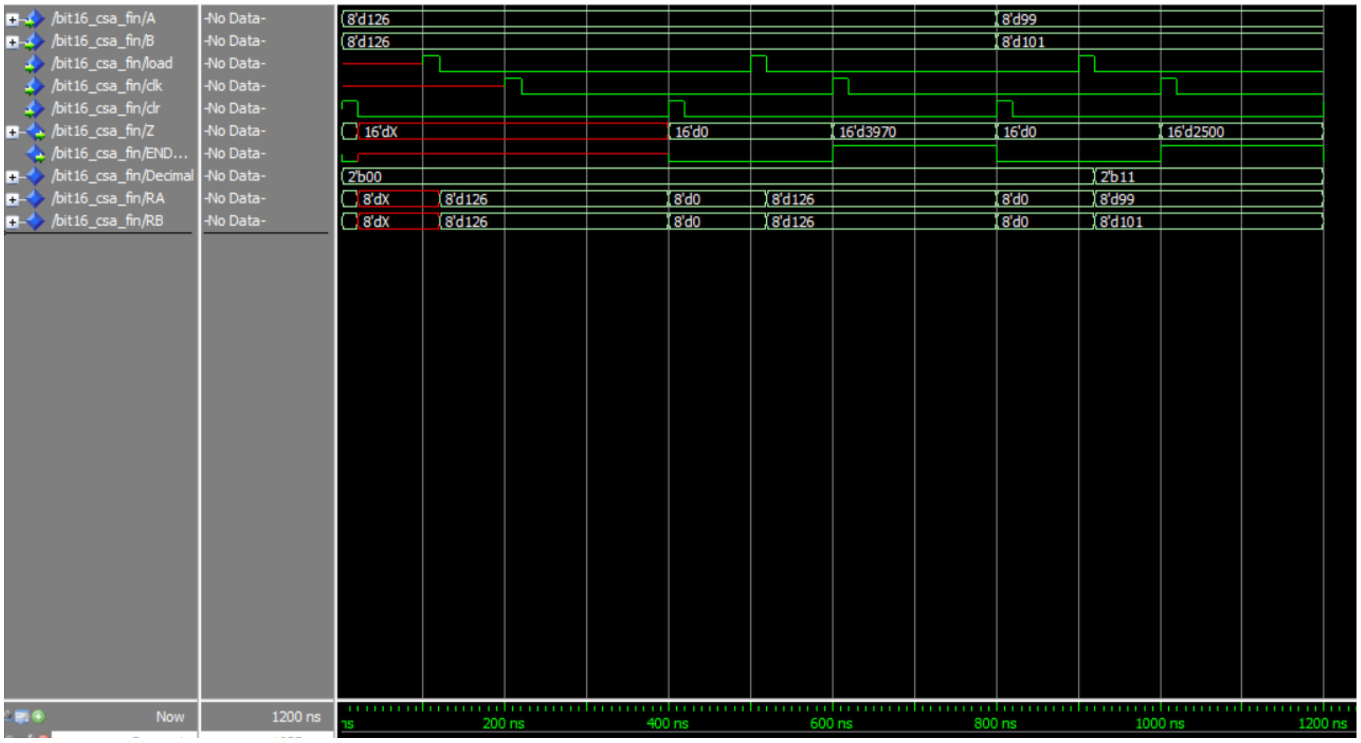


Figure 42: Test result 3

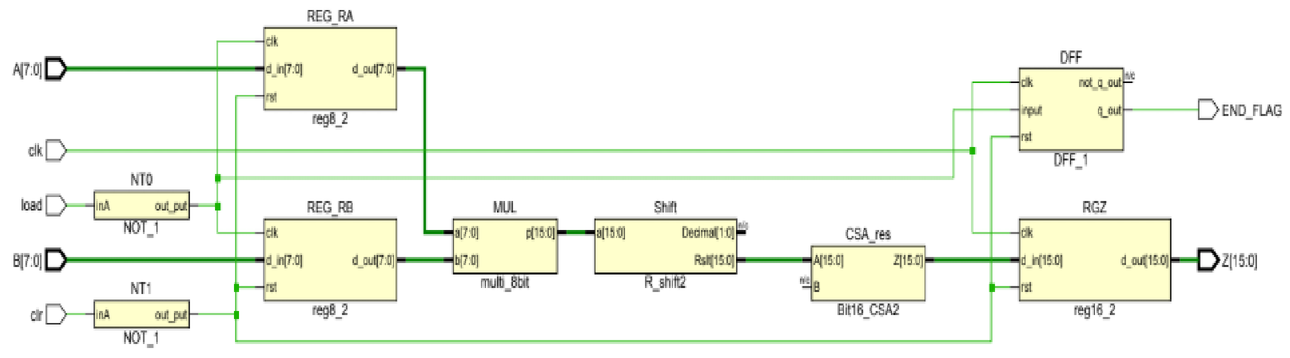


Figure 43: Final Circuit

8.3 Possible Alternative Design Approach

As mentioned earlier, there are various methodologies that can be used to build the design of arithmetic unit, and each methodology can affect the performance in a vital way. For example, a recent research and analysis of 8-bit array multiplier showed that the performance in terms of delay, area, and power can be altered by changing the number of the transistors used in the full adder. Therefore, a design can be improved for a certain aspect depending on the objective that needs to be achieved in a project (A1 & Shinde2, 2016).

9. Conclusion

All in all, the main objective of this project is to design an arithmetic unit that can multiply two multiplied unsigned 8-bits operands by ¼ and add to them one. Also, it was essential to make the design structurally and with the optimal possible scheme that provide a balanced system. Array multiplier was used to perform the multiplication, barrel shifter to perform the shifting by two bits to the right, and carry select adder the perform the addition. A register were designed to hold the two 8-bits operands, and the register must latch new operands for a transition of LOAD signal from high to low, and it must be cleared by a CLR signal. A register also was designed to hold the final result of 16-bits and latch it to the final output. Moreover, an END_FLAG signal was implemented to show the end of the arithmetic operation.

References

1.A1, A. K., & Shinde2, K. D. (2016). *Performance Analysis and Implementation of Array Multiplier using various Full Adder Designs for DSP Applications: A VLSI Based Approach.* Retrieved from ResearchGate: https://www.researchgate.net/publication/308327427_Performance_Analysis_and_Implementation_of_Array_Multiplier_using_various_Full_Adder_Designs_for_DSP_Applications_A_VLSI_Based_Approach

Appendices A- VHDL Code

1. Gates

1.1 XOR_1

```
-- XOR_1 Gate
-- Purpose: To implement a two input XOR gate


library ieee;
use ieee.std_logic_1164.all;


entity xor_1 is
port (A,B: in std_logic; Z: out std_logic);
end xor_1;

architecture DataFlow of xor_1 is

begin
Z <= (not A and B) or (A and not(B) );

end DataFlow;
```

1.2 and_1

```
-- and_1 Gate
-- Purpose: To implement a two input AND gate



library ieee;
use ieee.std_logic_1164.all;

entity and_1 is

port (A,B: in std_logic; Z: out std_logic);
end and_1;
```

```
architecture DataFlow of and_1 is


begin

Z <= A and B;

end DataFlow;
```

1.3 or_1

```
-- or_1 Gate
-- Purpose: To implement a two input OR gate




library ieee;
use ieee.std_logic_1164.all;

entity or_1 is

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

end or_1;

architecture DataFlow of or_1 is

begin
Z <= A or B;

end DataFlow;
```

1.4 NAND_1

```
-- NAND_1 gate
-- Purpose: To implement two inputs NAND gate
```

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

--  Entity

entity NAND_1 is

port
(
inA, inB : in std_logic;
out_put : out std_logic
);

end NAND_1;

--  architecture

architecture DataFlow of NAND_1 is

begin

out_put <= inA nand inB;

end DataFlow;
```

1.4 NAND_2

```
-- NAND_2 gate
-- Purpose: Implement 3 inputs NAND gate


library ieee;
use ieee.std_logic_1164.all;
-- Entity

entity NAND_2 is

port
(
inA, inB, inC : in std_logic;
out_put : out std_logic
);
```

```
end NAND_2;

-- Begin

architecture DataFlow of NAND_2 is

begin

out_put <= NOT (inA and (inB and inC));

end DataFlow;
```

1.5 NOT_1

```
-- NOT_1 gate
-- Purpose: Implement the NOT for input


library ieee;
use ieee.std_logic_1164.all;
-- Entity

entity NOT_1 is

port
(
inA : in std_logic;
out_put : out std_logic
);
end NOT_1;

-- Begin

architecture DataFlow of NOT_1 is

begin

out_put <= NOT(inA);

end DataFlow;
```

2. Components

2.1 Half_adder

```
-- Half Adder
-- Purpose: Does addition of 2 bits


library ieee;
use ieee.std_logic_1164.all;

entity Half_adder is

port (A,B: in std_logic; SUM, CRY: out std_logic);

end Half_adder;


-- Body

architecture STRUCTURAL of Half_adder is

component xor_1

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

end component;

component and_1

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

end component;

begin
X1: xor_1 port map (A,B,SUM);
A1: and_1 port map (A,B,CRY);
```

```
end STRUCTURAL;
```

2.2 F_adder

```
-- Full Adder
-- Purpose: Does addition of 3 bits


library ieee;
use ieee.std_logic_1164.all;

entity F_adder is

port (A,B,C: in std_logic; SUM, CRY: out std_logic);

end F_adder;

-- Body

architecture STRUCTURAL of F_adder is

component Half_adder

port (A,B : in std_logic; SUM,CRY: out std_logic);

end component;

component or_1

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

end component;

signal SUM1, CRY1, CRY2: std_logic;

begin
HA1: Half_adder port map (A, B, SUM1, CRY1);
HA2: Half_adder port map (SUM1, C, SUM, CRY2);
O_2: Or_1 port map (CRY1, CRY2, CRY);
```

```
end STRUCTURAL;
```

2.3 DFF_1

```
-- Flip Flop
-- Purpose: Store a bit and  low reset active


library ieee;
use ieee.std_logic_1164.all;


-- Entity
entity DFF_1 is
port
(
input : in std_logic;
clk : in std_logic;
rst : in std_logic;
not_q_out : out std_logic;
q_out : out std_logic
);

end DFF_1;
-- Begin
architecture behavioral of DFF_1 is


--Components

component NAND_1
port
(
inA, inB : in std_logic;
out_put : out std_logic
);

end component;
```

```
component NAND_2
port
(
inA, inB, inC : in std_logic;
out_put : out std_logic
);

end component;

-- Signals
signal s, r, not_q, q, t1, t2 : std_logic;
begin

-- Build NANDs
U1: NAND_1 port map (s, not_q, q);
U2: NAND_2 port map (r, q, rst, not_q);
U3: NAND_1 port map (s, t2, t1);
U4: NAND_2 port map (t1, clk, rst, s);
U5: NAND_2 port map (s, clk, t2, r);
U6: NAND_2 port map (r, input, rst, t2);

--Outputs

q_out <= q;
not_q_out <= not_q;

end behavioral;
```

2.4 Mux_twoToOne

```
-- 2-1 Mux
-- Purpose: Choose one output out of two using select signal

library ieee;
use ieee.std_logic_1164.all;


-- Begin entity
```

```
entity Mux_twoToOne is

port
(
x, y, selec : in std_logic;
c : out std_logic
);
end Mux_twoToOne;

-- Architecture

architecture Behavioral of Mux_twoToOne is

-- Components


component and_1
port
(
A, B : in std_logic;
Z : out std_logic
);
end component;


component NOT_1
port
(
inA : in std_logic;
out_put : out std_logic
);
end component;

component or_1
port
(
A, B : in std_logic;
Z : out std_logic
);
end component;

-- Signals
```

```
signal te0, te1, te2 : std_logic;

begin

-- Connections

U1: and_1 port map(selec, y, te0);
U2: NOT_1 port map (selec, te1);
U3: and_1 port map (te1,x, te2);
U4: or_1 port map (te0, te2, c);

end Behavioral;
```

3. Registers

3.1 reg8_2

```
-- 8-Bit register
-- Purpose: Hold the operand A and B input bits


library ieee;
use ieee.std_logic_1164.all;


-- Entity


entity reg8_2 is
port
(
d_in : in std_logic_vector(7 downto 0);
clk, rst : in std_logic;
d_out : out std_logic_vector(7 downto 0)
);
end reg8_2;
```

```
-- Architecture

architecture behavioral of reg8_2 is

-- Signals

signal nt_q : std_logic_vector (7 downto 0);

-- Components


component DFF_1
port
(
input : in std_logic;
clk : in std_logic;
rst : in std_logic;
not_q_out : out std_logic;
q_out : out std_logic
);
end component;



component NOT_1

port
(
inA : in std_logic;
out_put : out std_logic
);
end component;

begin


-- Create register
```

```
register_generate: for i in 0 to 7 generate

--Connections

UA: DFF_1 port map (d_in(i), clk, rst, nt_q(i), d_out(i));

end generate;

end behavioral;
```

3.2 reg16_2

```
-- 16bit register
-- Purpose: Store 16 bit result


library ieee;
use ieee.std_logic_1164.all;


-- Entity


entity reg16_2 is
port
(
d_in : in std_logic_vector(15 downto 0);
clk, rst : in std_logic;
d_out : out std_logic_vector(15 downto 0)
);
end reg16_2;




-- Architecture

architecture behavioral of reg16_2 is

-- Signals
```

```
signal nt_q : std_logic_vector (15 downto 0);

-- Components


component DFF_1

port
(
input : in std_logic;
clk : in std_logic;
rst : in std_logic;
not_q_out : out std_logic;
q_out : out std_logic
);
end component;

component NOT_1

port
(
inA : in std_logic;
out_put : out std_logic
);
end component;

begin



-- Generate registers


register_generate: for i in 0 to 15 generate


UA: DFF_1 port map (d_in(i), clk, rst, nt_q(i), d_out(i));

end generate;
end behavioral;
```

4. Operations Sub-Blocks

4.1 PP_generation

```
-- Partial Product Generation
-- Purpose: Create partial products using AND gates


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;


entity PP_generation is

Port ( A : in STD_LOGIC_VECTOR (7 downto 0);
B : in STD_LOGIC_VECTOR (7 downto 0);
Z : out STD_LOGIC_VECTOR (63 downto 0));
end PP_generation;

architecture Behavioral of PP_generation is
-- component declaration


component and_1

port(a,b : in std_logic; z: out std_logic);

end component;

begin

a0: and_1 port map(a(0),b(0),z(0));a1: and_1 port map(a(1),b(0),z(1));
a2: and_1 port map(a(2),b(0),z(2));a3: and_1 port map(a(3),b(0),z(3));
a4: and_1 port map(a(4),b(0),z(4));a5: and_1 port map(a(5),b(0),z(5));
a6: and_1 port map(a(6),b(0),z(6));a7: and_1 port map(a(7),b(0),z(7));
a8: and_1 port map(a(0),b(1),z(8));a9: and_1 port map(a(1),b(1),z(9));
a10: and_1 port map(a(2),b(1),z(10));a11: and_1 port map(a(3),b(1),z(11));
```

```
a12: and_1 port map(a(4),b(1),z(12));a13: and_1 port map(a(5),b(1),z(13));
a14: and_1 port map(a(6),b(1),z(14));a15: and_1 port map(a(7),b(1),z(15));
a16: and_1 port map(a(0),b(2),z(16));a17: and_1 port map(a(1),b(2),z(17));
a18: and_1 port map(a(2),b(2),z(18));a19: and_1 port map(a(3),b(2),z(19));
a20: and_1 port map(a(4),b(2),z(20));a21: and_1 port map(a(5),b(2),z(21));
a22: and_1 port map(a(6),b(2),z(22));a23: and_1 port map(a(7),b(2),z(23));
a24: and_1 port map(a(0),b(3),z(24));a25: and_1 port map(a(1),b(3),z(25));
a26: and_1 port map(a(2),b(3),z(26));a27: and_1 port map(a(3),b(3),z(27));
a28: and_1 port map(a(4),b(3),z(28));a29: and_1 port map(a(5),b(3),z(29));
a30: and_1 port map(a(6),b(3),z(30));a31: and_1 port map(a(7),b(3),z(31));
a32: and_1 port map(a(0),b(4),z(32));a33: and_1 port map(a(1),b(4),z(33));
a34: and_1 port map(a(2),b(4),z(34));a35: and_1 port map(a(3),b(4),z(35));
a36: and_1 port map(a(4),b(4),z(36));a37: and_1 port map(a(5),b(4),z(37));
a38: and_1 port map(a(6),b(4),z(38));a39: and_1 port map(a(7),b(4),z(39));
a40: and_1 port map(a(0),b(5),z(40));a41: and_1 port map(a(1),b(5),z(41));
a42: and_1 port map(a(2),b(5),z(42));a43: and_1 port map(a(3),b(5),z(43));
a44: and_1 port map(a(4),b(5),z(44));a45: and_1 port map(a(5),b(5),z(45));
a46: and_1 port map(a(6),b(5),z(46));a47: and_1 port map(a(7),b(5),z(47));
a48: and_1 port map(a(0),b(6),z(48));a49: and_1 port map(a(1),b(6),z(49));
a50: and_1 port map(a(2),b(6),z(50));a51: and_1 port map(a(3),b(6),z(51));
a52: and_1 port map(a(4),b(6),z(52));a53: and_1 port map(a(5),b(6),z(53));
a54: and_1 port map(a(6),b(6),z(54));a55: and_1 port map(a(7),b(6),z(55));
a56: and_1 port map(a(0),b(7),z(56));a57: and_1 port map(a(1),b(7),z(57));
a58: and_1 port map(a(2),b(7),z(58));a59: and_1 port map(a(3),b(7),z(59));
a60: and_1 port map(a(4),b(7),z(60));a61: and_1 port map(a(5),b(7),z(61));
a62: and_1 port map(a(6),b(7),z(62));a63: and_1 port map(a(7),b(7),z(63));


end Behavioral;
```

4.2 FourBit_RCA

```
-- 4 bit RCA
-- Purpose: Make the addition of 4-bits using RCA method

library ieee;
use ieee.std_logic_1164.all;


-- Entity

entity FourBit_RCA is
```

```
port
(
A : in std_logic_vector (3 downto 0);
B : in std_logic ;
CarryIn : in std_logic;
sum : out std_logic_vector (3 downto 0);
CarryOut : out std_logic
);
end FourBit_RCA;

-- Architecture

architecture Behavioral of FourBit_RCA is

-- Components

component F_adder
port
(
A, B, C : in std_logic;
CRY, SUM : out std_logic
);
end component;


-- Signals


signal te_c : std_logic_vector (3 downto 0);

begin

U1: F_adder port map (A(0), '0',CarryIn, te_c(0), SUM(0));

-- Generate full adder sequence

fulladder_generate: for i in 1 to 3 generate
UA: F_adder port map (A(i), '0' , te_c(i-1), te_c(i), SUM(i));

end generate;
```

```
-- Get the output

CarryOut <= te_c(3);

end Behavioral;
```

4.3 FourBit_CSA

```
-- 4-bit CSA
-- Purpose: 4-Bit CSA used to add 4-bits using CSA method and to be used for final CSA

library ieee;
use ieee.std_logic_1164.all;

--Entity

entity FourBit_CSA is

port
(
A : in std_logic_vector (3 downto 0);
B: in std_logic;
mu_sel : in std_logic;
SU : out std_logic_vector (3 downto 0);
carry_out : out std_logic
);
end FourBit_CSA;

--Architecture


architecture Behavioral of FourBit_CSA is


--Components

component FourBit_RCA
port
(
A : in std_logic_vector (3 downto 0);
B : in std_logic;
CarryIn : in std_logic;
sum : out std_logic_vector (3 downto 0);
CarryOut : out std_logic
);
end component;

component Mux_twoToOne

port
(
x, y, selec : in std_logic;
c : out std_logic
);
end component;

--Signals

signal te_s0 : std_logic_vector (3 downto 0);
signal te_s1 : std_logic_vector (3 downto 0);
signal te_c0 : std_logic;
signal te_c1 : std_logic;

begin

-- Generate RCA result for carry possibility of 0

U1: FourBit_RCA port map (A,'0','0', te_s0, te_c0);

-- Generate RCA result for carry possibility of 1

U2: FourBit_RCA port map (A,'0', '1', te_s1, te_c1);

-- Connection of Multiplexers for sum

s_muxs: for i in 3 downto 0 generate

UA: Mux_twoToOne port map (te_s0(i), te_s1(i), mu_sel, SU(i));

end generate;

-- Connection of multiplexer for carry
```

```
U3: Mux_twoToOne port map (te_c0, te_c1, mu_sel, carry_out);

end Behavioral;
```

5. Operation Main Blocks

5.1 multi_8bit

```
-- 8-bit Array Multiplier
-- Purpose: Multiply two unsigned 8-bits

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;


entity multi_8bit is

Port ( a : in STD_LOGIC_VECTOR (7 downto 0);
b : in STD_LOGIC_VECTOR (7 downto 0);
p : out STD_LOGIC_VECTOR (15 downto 0));

end multi_8bit;

architecture Behavioral of multi_8bit is

--- Component Declaration

component PP_generation
port(a,b : in std_logic_vector(7 downto 0);
z: out std_logic_vector(63 downto 0));
end component;

component F_adder
port (A,B,C: in std_logic;
SUM, CRY: out std_logic);
end component;

--- Signal Declaration
```

```
signal z :STD_LOGIC_VECTOR (63 downto 0);
signal cf :STD_LOGIC_VECTOR (59 downto 0);
signal S :STD_LOGIC_VECTOR (55 downto 0);
begin
p1: PP_generation port map(a,b,z);
p(0) <= z(0);
fa1: F_adder port map(z(1),z(8),'0',p(1),cf(1));
fa2: F_adder port map(z(2),z(9),cf(1),s(2),cf(2));
fa3: F_adder port map(s(2),z(16),'0',p(2),cf(3));
fa4: F_adder port map(z(3),cf(2),z(10),s(4),cf(4));
fa5: F_adder port map(s(4),cf(3),z(17),s(5),cf(5));
fa6: F_adder port map(s(5),'0',z(24),p(3),cf(6));
fa7: F_adder port map(z(4),cf(4),z(11),s(7),cf(7));
fa8: F_adder port map(s(7),cf(5),z(18),s(8),cf(8));
fa9: F_adder port map(s(8),cf(6),z(25),s(9),cf(9));
fa10: F_adder port map(s(9),'0',z(32),p(4),cf(10));
fa11: F_adder port map(z(5),cf(7),z(12),s(11),cf(11));
fa12: F_adder port map(s(11),cf(8),z(19),s(12),cf(12));
fa13: F_adder port map(s(12),cf(9),z(26),s(13),cf(13));
fa14: F_adder port map(s(13),cf(10),z(33),s(14),cf(14));
fa15: F_adder port map(s(14),'0',z(40),p(5),cf(15));
fa16: F_adder port map(z(6),cf(11),z(13),s(16),cf(16));
fa17: F_adder port map(s(16),cf(12),z(20),s(17),cf(17));
fa18: F_adder port map(s(17),cf(13),z(27),s(18),cf(18));
fa19: F_adder port map(s(18),cf(14),z(34),s(19),cf(19));
fa20: F_adder port map(s(19),cf(15),z(41),s(20),cf(20));
fa21: F_adder port map(s(20),'0',z(48),p(6),cf(21));
fa22: F_adder port map(z(7),cf(16),z(14),s(22),cf(22));
fa23: F_adder port map(s(22),cf(17),z(21),s(23),cf(23));
fa24: F_adder port map(s(23),cf(18),z(28),s(24),cf(24));
fa25: F_adder port map(s(24),cf(19),z(35),s(25),cf(25));
fa26: F_adder port map(s(25),cf(20),z(42),s(26),cf(26));
fa27: F_adder port map(s(26),cf(21),z(49),s(27),cf(27));
fa28: F_adder port map(s(27),'0',z(56),p(7),cf(28));
fa29: F_adder port map('0',cf(22),z(15),s(29),cf(29));
fa30: F_adder port map(s(29),cf(23),z(22),s(30),cf(30));
fa31: F_adder port map(s(30),cf(24),z(29),s(31),cf(31));
fa32: F_adder port map(s(31),cf(25),z(36),s(32),cf(32));
fa33: F_adder port map(s(32),cf(26),z(43),s(33),cf(33));
fa34: F_adder port map(s(33),cf(27),z(50),s(34),cf(34));
fa35: F_adder port map(s(34),cf(28),z(57),p(8),cf(35));
```

```
fa36: F_adder port map(cf(29),z(23),cf(30),s(36),cf(36));
fa37: F_adder port map(s(36),cf(31),z(30),s(37),cf(37));
fa38: F_adder port map(s(37),cf(32),z(37),s(38),cf(38));
fa39: F_adder port map(s(38),cf(33),z(44),s(39),cf(39));
fa40: F_adder port map(s(39),cf(34),z(51),s(40),cf(40));
fa41: F_adder port map(s(40),cf(35),z(58),p(9),cf(41));
fa42: F_adder port map(cf(36),z(31),cf(37),s(42),cf(42));
fa43: F_adder port map(s(42),cf(38),z(38),s(43),cf(43));
fa44: F_adder port map(s(43),cf(39),z(45),s(44),cf(44));
fa45: F_adder port map(s(44),cf(40),z(52),s(45),cf(45));
fa46: F_adder port map(s(45),cf(41),z(59),p(10),cf(46));
fa47: F_adder port map(cf(42),z(39),cf(43),s(47),cf(47));
fa48: F_adder port map(s(47),cf(44),z(46),s(48),cf(48));
fa49: F_adder port map(s(48),cf(45),z(53),s(49),cf(49));
fa50: F_adder port map(s(49),cf(46),z(60),p(11),cf(50));
fa51: F_adder port map(cf(47),z(47),cf(48),s(51),cf(51));
fa52: F_adder port map(s(51),cf(49),z(54),s(52),cf(52));
fa53: F_adder port map(s(52),cf(50),z(61),p(12),cf(53));
fa54: F_adder port map(cf(51),z(55),cf(52),s(54),cf(54));
fa55: F_adder port map(s(54),cf(53),z(62),p(13),cf(55));
fa56: F_adder port map(cf(55),z(63),cf(54),p(14),p(15));

end Behavioral;
```

5.2 R_shift2

```
-- Barrel Shifter
-- Purpose: Shift multiplication result by two bits to the right


library ieee;
use ieee.std_logic_1164.all;

--Entity

entity R_shift2 is

port (
a : in STD_LOGIC_VECTOR (15 downto 0);
Decimal : out STD_LOGIC_VECTOR (1 downto 0);
Rslt : out STD_LOGIC_VECTOR (15 downto 0)
);

end R_shift2;

--Architecture

architecture Behavioral of R_shift2 is


--Components

component Mux_twoToOne
port
(
x, y, selec : in std_logic;
c : out std_logic
);
end component;

--Signals

signal w: STD_LOGIC_VECTOR (15 downto 0);
signal k: STD_LOGIC_VECTOR (15 downto 0);


begin

--Casting the Decimals

Decimal(0) <= A(0);
Decimal(1) <= A(1);

--First row of barrel shifter

MX1: Mux_twoToOne port map (A(0), A(1),'0',w(0));
MX2: Mux_twoToOne port map (A(1), A(2),'0',w(1));
MX3: Mux_twoToOne port map (A(2), A(3),'0',w(2));
MX4: Mux_twoToOne port map (A(3), A(4),'0',w(3));
MX5: Mux_twoToOne port map (A(4), A(5),'0',w(4));
MX6: Mux_twoToOne port map (A(5), A(6),'0',w(5));
MX7: Mux_twoToOne port map (A(6), A(7),'0',w(6));
```

```
MX8: Mux_twoToOne port map (A(7), A(8),'0',w(7));
MX9: Mux_twoToOne port map (A(8), A(9),'0',w(8));
MX10: Mux_twoToOne port map (A(9), A(10),'0',w(9));
MX11: Mux_twoToOne port map (A(10), A(11),'0',w(10));
MX12: Mux_twoToOne port map (A(11), A(12),'0',w(11));
MX13: Mux_twoToOne port map (A(12), A(13),'0',w(12));
MX14: Mux_twoToOne port map (A(13), A(14),'0',w(13));
MX15: Mux_twoToOne port map (A(14), A(15),'0',w(14));
MX16: Mux_twoToOne port map (A(15), '0','0',w(15));

--Second row of barrel shifter

MX17: Mux_twoToOne port map ( w(0),w(2), '1',k(0));
MX18: Mux_twoToOne port map ( w(1),w(3), '1',k(1));
MX19: Mux_twoToOne port map ( w(2),w(4), '1',k(2));
MX20: Mux_twoToOne port map ( w(3),w(5), '1',k(3));
MX21: Mux_twoToOne port map ( w(4),w(6), '1',k(4));
MX22: Mux_twoToOne port map ( w(5),w(7), '1',k(5));
MX23: Mux_twoToOne port map ( w(6),w(8), '1',k(6));
MX24: Mux_twoToOne port map ( w(7),w(9), '1',k(7));
MX25: Mux_twoToOne port map ( w(8),w(10), '1',k(8));
MX26: Mux_twoToOne port map ( w(9),w(11), '1',k(9));
MX27: Mux_twoToOne port map ( w(10),w(12), '1',k(10));
MX28: Mux_twoToOne port map ( w(11),w(13), '1',k(11));
MX29: Mux_twoToOne port map ( w(12),w(14), '1',k(12));
MX30: Mux_twoToOne port map ( w(13),w(15), '1',k(13));
MX31: Mux_twoToOne port map ( w(14),'0', '1',k(14));
MX32: Mux_twoToOne port map ( w(15),'0', '1',k(15));

--Third row of barrel shifter

MX33: Mux_twoToOne port map (k(0),k(3),'0',Rslt(0));
MX34: Mux_twoToOne port map (k(1),k(4),'0',Rslt(1));
MX35: Mux_twoToOne port map (k(2),k(5),'0',Rslt(2));
MX36: Mux_twoToOne port map (k(3),k(6),'0',Rslt(3));
MX37: Mux_twoToOne port map (k(4),k(7),'0',Rslt(4));
MX38: Mux_twoToOne port map (k(5),k(8),'0',Rslt(5));
MX39: Mux_twoToOne port map (k(6),k(9),'0',Rslt(6));
MX40: Mux_twoToOne port map (k(7),k(10),'0',Rslt(7));
MX41: Mux_twoToOne port map (k(8),k(11),'0',Rslt(8));
MX42: Mux_twoToOne port map (k(9),k(12),'0',Rslt(9));
MX43: Mux_twoToOne port map (k(10),k(13),'0',Rslt(10));
```

```
MX44: Mux_twoToOne port map (k(11),k(14),'0',Rslt(11));
MX45: Mux_twoToOne port map (k(12),k(15),'0',Rslt(12));
MX46: Mux_twoToOne port map (k(13),'0','0',Rslt(13));
MX47: Mux_twoToOne port map (k(14),'0','0',Rslt(14));
MX48: Mux_twoToOne port map (k(15),'0','0',Rslt(15));

end Behavioral;
```

5.3 Bit16_CSA2

```
-- 16-CSA Block
-- Purpose: Perform the addition of 1 for the shifted result

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

-- Entity
entity Bit16_CSA2 is

port
(
A : in std_logic_vector (15 downto 0);
B : in std_logic;
Z : out std_logic_vector (15 downto 0)
);
end Bit16_CSA2;


-- Architecture

architecture Behavioral of Bit16_CSA2 is

-- Components

component FourBit_CSA
port
(
A : in std_logic_vector (3 downto 0);
B: in std_logic ;
mu_sel : in std_logic;
SU : out std_logic_vector (3 downto 0);
carry_out : out std_logic
);
end component;

component Half_adder
port
(
A : in std_logic;
B : in std_logic;
CRY : out std_logic;
SUM : out std_logic
);
end component;

component F_adder
port
(
A, B, C : in std_logic;
CRY, SUM : out std_logic
);
end component;

-- Signals

signal te_c : std_logic_vector (4 downto 0);
signal te_mxsel : std_logic_vector (1 downto 0);


begin


-- First half adder for sum of first bit


U1: Half_adder port map (A(0),'1', te_c(0), z(0));

-- Create full adders for remaining three bits addition

First_three_FA: for i in 1 to 3 generate
UA: F_adder port map (A(i),'0', te_c(i-1), te_c(i),z(i));

end generate;
```

```
-- First 4-bits CSA block

U2: FourBit_CSA port map (A (7 downto 4), '0', te_c(3), z(7 downto 4), te_mxsel(0));

-- Second 4-bits CSA block

U3: FourBit_CSA port map (A (11 downto 8), '0', te_mxsel(0), z(11 downto 8), te_mxsel(1));

-- Third 4-bits CSA block

U4: FourBit_CSA port map (A (15 downto 12), '0', te_mxsel(1), z(15 downto 12));

end Behavioral;
```

6. Final Circuit

Final_circuit

```
-- Final Circuit
-- Purpose: Perform the arithmetic operation

library ieee;
use ieee.std_logic_1164.all;



-- Entity

entity Final_circuit is

port
(
A : in std_logic_vector (7 downto 0);
B : in std_logic_vector (7 downto 0);
load: in std_logic ;
clk: in std_logic;
clr: in std_logic;
Z : out std_logic_vector (15 downto 0);
END_FLAG :out std_logic

);
end Final_circuit;


-- Architecture

architecture Behavioral of Final_circuit is

-- Components

component R_shift2

port ( a : in STD_LOGIC_VECTOR (15 downto 0);
Decimal : out STD_LOGIC_VECTOR (1 downto 0);
Rslt : out STD_LOGIC_VECTOR (15 downto 0)
);

end component;


component reg16_2
port
(
d_in : in std_logic_vector(15 downto 0);
clk, rst : in std_logic;
d_out : out std_logic_vector(15 downto 0)
);
end component;


component reg8_2
port
(
d_in : in std_logic_vector(7 downto 0);
clk, rst : in std_logic;
d_out : out std_logic_vector(7 downto 0)
);
end component;
```

```
component Half_adder
port
(
A : in std_logic;
B : in std_logic;
CRY : out std_logic;
SUM : out std_logic
);
end component;

component F_adder
port
(
A, B, C : in std_logic;
CRY, SUM : out std_logic
);
end component;

component multi_8bit
Port ( a : in STD_LOGIC_VECTOR (7 downto 0);
b : in STD_LOGIC_VECTOR (7 downto 0);
p : out STD_LOGIC_VECTOR (15 downto 0));
end component;

component DFF_1
port
(
input : in std_logic;
clk : in std_logic;
rst : in std_logic;
not_q_out : out std_logic;
q_out : out std_logic
);
end component;

component NOT_1 is

port
(
inA : in std_logic;
out_put : out std_logic
);
```

```
end component;

component Bit16_CSA2 is

port
(
A : in std_logic_vector (15 downto 0);
B : in std_logic;
Z : out std_logic_vector (15 downto 0)

);
end component;

-- Signals

signal te_c : std_logic_vector (4 downto 0);
signal te_mxsel : std_logic_vector (1 downto 0);
signal Decimal : std_logic_vector (1 downto 0);
signal Rslt : std_logic_vector (15 downto 0);
signal inv_load: std_logic;
signal inv_clr: std_logic;
signal vld_d, n_vld_d : std_logic;
signal RA : std_logic_vector (7 downto 0);
signal RB : std_logic_vector (7 downto 0);
signal RZ : std_logic_vector (15 downto 0);
signal BT : std_logic;
signal multi_result: std_logic_vector (15 downto 0);
signal final_result: std_logic_vector (15 downto 0);

--Begin

begin

-- Create invert signals

NT0: NOT_1 port map (load, inv_load);
NT1: NOT_1 port map (clr, inv_clr);

--create operand register A and B

REG_RA: reg8_2 port map (A, inv_load, inv_clr, RA);
REG_RB: reg8_2 port map (B, inv_load, inv_clr, RB);
```

```
--create multiplication result

MUL: multi_8bit port map (RA,RB, multi_result);


--create  Shift result

Shift: R_shift2 port map (multi_result,Decimal,Rslt);

--create CSA result

CSA_res: Bit16_CSA2 port map (Rslt,BT, RZ);


-- create register for final result

RGZ: reg16_2 port map (RZ, clk, inv_clr, final_result);

--create output Z from RZ


z <= final_result;

--create DFF for storing

DFF: DFF_1 port map (inv_load,clk, inv_clr, n_vld_d, vld_d);

--cast the END_FLAG

END_FLAG <=  vld_d;


end Behavioral;
```

7. Test Bench

Fin_Circuit_test_bench

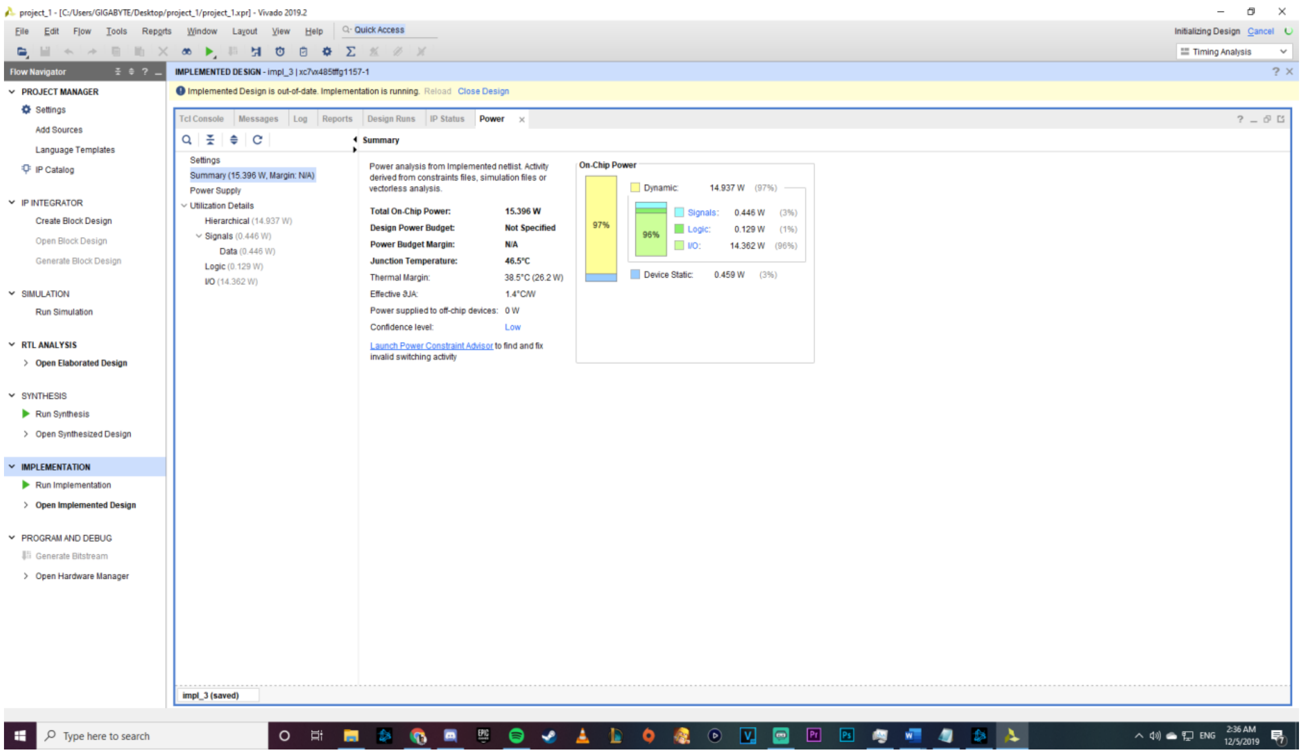
```
--Test Bench
```

```
--Purpose: Test Bench for Final circuit

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use ieee.std_logic_unsigned.all;


ENTITY Fin_Circuit_test_bench IS
END Fin_Circuit_test_bench;

ARCHITECTURE behavior OF Fin_Circuit_test_bench IS

COMPONENT final_circuit

PORT(
A : IN std_logic_vector(7 downto 0);
B : IN std_logic_vector(7 downto 0);
Z : OUT std_logic_vector(15 downto 0);
clk : IN std_logic;
clr : IN std_logic;
load: in std_logic;
END_FLAG : OUT std_logic
);

END COMPONENT;
--Inputs
signal A : std_logic_vector(7 downto 0);
signal B : std_logic_vector(7 downto 0);
signal clk : std_logic := '0';
signal clr : std_logic := '0';
signal load: std_logic := '0';
--Outputs
signal Z : std_logic_vector(15 downto 0);
signal END_FLAG : std_logic;

BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: final_circuit PORT MAP (
A => A,
B => B,
Z => Z,
clk => clk,
clr => clr,
load => load,
END_FLAG => END_FLAG
);

clk_process :process
  begin
  clk <= '0';
  wait for 325ns;
  clk <= '1';
  wait for 50ns;
  end process;

  stim_proc: process
  begin

  clr <= '1';
  wait for 50 ns;

  clr <= '0';
wait for 50ns;

load <= '1';
 A <= "00000000";
B <="00000000";
 wait for 150ns;

  load <= '0';
A <= "00000011";
B <="00000011";
 wait for 150ns;

clr<='1';
wait for 50ns;

clr <='0';
wait for 50ns;

load <='1';
A <="00000111";
B <="00000111";
wait for 150ns;
```

```
load <='0';
wait for 50ns;


  wait;

  end process;

END;
```

Appendices B- Reports

1. Power Report



IMPLEMENTED DESIGN - impl_3 | xc7vx485tffg1157-1

Implemented Design is out-of-date. Implementation is running. Reload Close Design

Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**15.396 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**46.5°C**
Thermal Margin:	38.5°C (26.2 W)
Effective ϑJA:	1.4°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

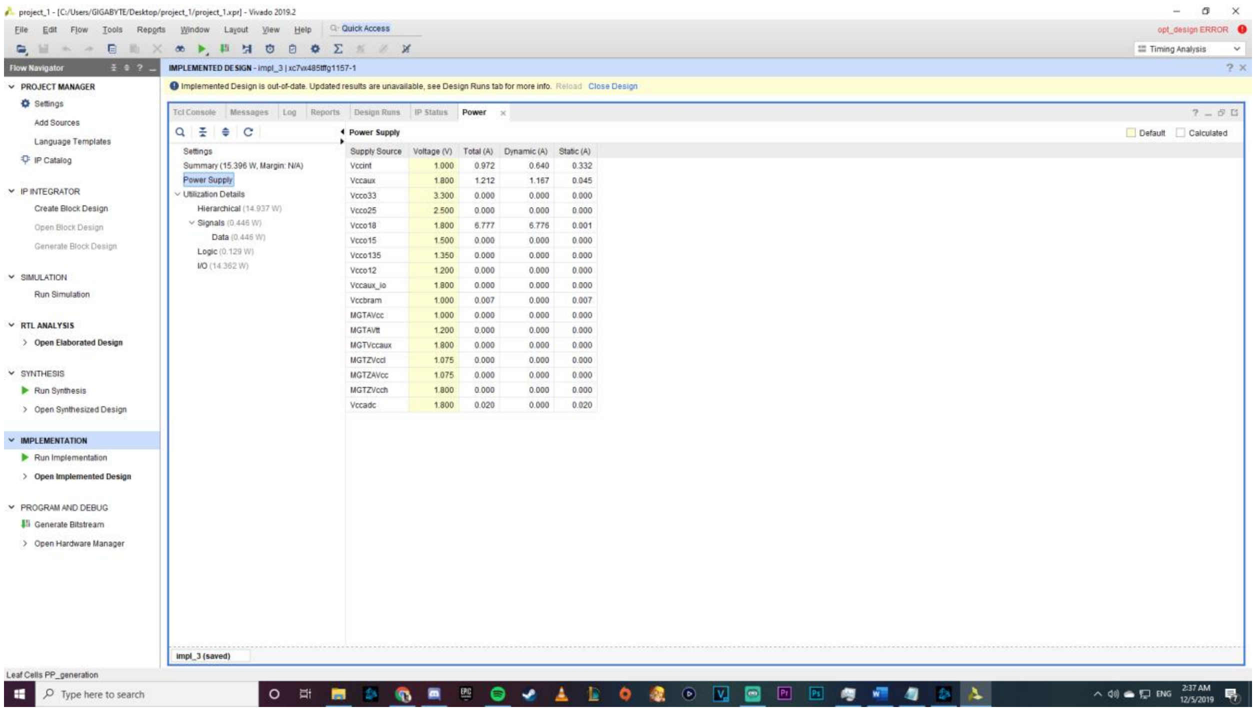
Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	14.937 W	(97%)
Signals:	0.446 W	(3%)
Logic:	0.129 W	(1%)
I/O:	14.362 W	(96%)
Device Static:	0.459 W	(3%)

Settings
Summary (15.396 W, Margin: N/A)
Power Supply
Utilization Details
Hierarchical (14.937 W)
Signals (0.446 W)
Data (0.446 W)
Logic (0.129 W)
I/O (14.362 W)

2. Power Supply



IMPLEMENTED DESIGN - impl_3 | xc7vx485tffg1157-1

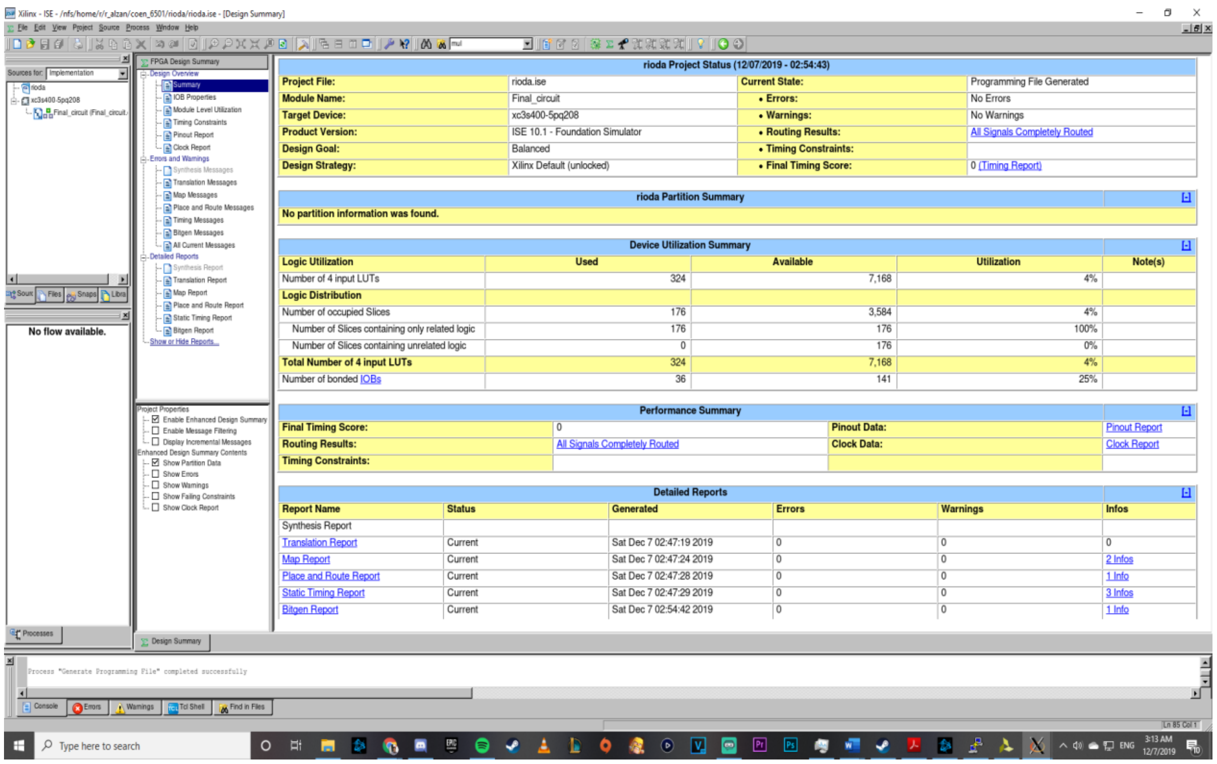
Implemented Design is out-of-date. Updated results are unavailable, see Design Runs tab for more info.

Settings
Summary (15.396 W, Margin: N/A)
Power Supply
Utilization Details
Hierarchical (14.937 W)
Signals (0.446 W)
Data (0.446 W)
Logic (0.129 W)
I/O (14.362 W)

Power Supply

Supply Source	Voltage (V)	Total (A)	Dynamic (A)	Static (A)
Vccint	1.000	0.972	0.640	0.332
Vccaux	1.800	1.212	1.167	0.045
Vcco33	3.300	0.000	0.000	0.000
Vcco25	2.500	0.000	0.000	0.000
Vcco18	1.800	6.777	6.776	0.001
Vcco15	1.500	0.000	0.000	0.000
Vcco135	1.350	0.000	0.000	0.000
Vcco12	1.200	0.000	0.000	0.000
Vccaux_io	1.800	0.000	0.000	0.000
Vccbram	1.000	0.007	0.000	0.007
MGTAVcc	1.000	0.000	0.000	0.000
MGTAVtt	1.200	0.000	0.000	0.000
MGTVccaux	1.800	0.000	0.000	0.000
MGTZVccl	1.075	0.000	0.000	0.000
MGTZAVcc	1.075	0.000	0.000	0.000
MGTZVcch	1.800	0.000	0.000	0.000
Vccadc	1.800	0.020	0.000	0.020

3. Circuit Area



rioda Project Status (12/07/2019 - 02:54:43)

Project File:	rioda.ise	**Current State:**	Programming File Generated
Module Name:	Final_circuit	• **Errors:**	No Errors
Target Device:	xc3s400-5pq208	• **Warnings:**	No Warnings
Product Version:	ISE 10.1 - Foundation Simulator	• **Routing Results:**	All Signals Completely Routed
Design Goal:	Balanced	• **Timing Constraints:**	
Design Strategy:	Xilinx Default (unlocked)	• **Final Timing Score:**	0 (Timing Report)

rioda Partition Summary

No partition information was found.

Device Utilization Summary

Logic Utilization	Used	Available	Utilization	Note(s)
Number of 4 input LUTs	324	7,168	4%	
Logic Distribution				
Number of occupied Slices	176	3,584	4%	
Number of Slices containing only related logic	176	176	100%	
Number of Slices containing unrelated logic	0	176	0%	
Total Number of 4 input LUTs	324	7,168	4%	
Number of bonded IOBs	36	141	25%	

Performance Summary

Final Timing Score:	0	**Pinout Data:**	Pinout Report
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report
Timing Constraints:			

Detailed Reports

Report Name	Status	Generated	Errors	Warnings	Infos
Synthesis Report					
Translation Report	Current	Sat Dec 7 02:47:19 2019	0	0	0
Map Report	Current	Sat Dec 7 02:47:24 2019	0	0	2 Infos
Place and Route Report	Current	Sat Dec 7 02:47:28 2019	0	0	1 Info
Static Timing Report	Current	Sat Dec 7 02:47:29 2019	0	0	3 Infos
Bitgen Report	Current	Sat Dec 7 02:54:42 2019	0	0	1 Info

```
Process "Generate Programming File" completed successfully
```

Appendix C – Beak Down of Work

Break Down of Work

'✓' REPRESENTS SIGNIFICANT CONTRIBUTION

Sr. No.	VHDL Code Work	*Mohammed*	*Rayan*
1	Gates	✓	✓
2	Half Adder		✓
3	Full Adder		✓
4	D Flip-Flop	✓	
5	Mux 2:1	✓	
6	8 bit register		✓
7	16 bit register	✓	
8	Partial Product for Array Multiplier		✓
9	4-bit RCA	✓	
10	4-bit CSA	✓	
11	8bit array multiplier	✓	✓
12	Barrel Shifter	✓	
13	16bit CSA	✓	
14	Final Circuit	✓	
15	Test Bench	✓	✓
16	Xilinx/ Vivado reports		✓

Sr. No.	Report Work – By Chapters	*Mohammed*	*Rayan*
1	Abstract		✓
2	Introduction	✓	
3	The overall design	✓	
4	Fundamental gates		✓
5	Building required components		✓
6	Building the multiplier	✓	
7	Building the shifter	✓	
8	Building the adder	✓	
9	Final simulations	✓	
10	Conclusion		✓
11	Final report preparation	✓	