Project Report on

Design and Implementation of Arithmetic Logic Unit for Z= ¼ (A*B) +1

Department of Electrical and Computer

Engineering Concordia University

Submitted By:

Zain Bhinder ([redacted])

Syed Muhammad Hasan ([redacted])

ABSTRACT

In the contemporary era, Arithmetic and Logical Unit (ALU) plays a very vital role in the field of digital computing and is employed widely for variety of applications from solving simple arithmetic operations to computing algorithms in complex circuitry. Computer architecture consists of ALU acting as a major component of central processing unit (CPU) to perform arithmetic and logic operations on the provided operands. In this project, an ALU is devised capable to solve Z= ¼ (A*B) +1 equation using Hardware Description language (VHDL). Different methodologies have been compared to select the most efficient and effective multiplier, shifter and increment component. TOP module has been implemented with and without FSM. For the extra features, the Architecture design has been further expanded for the 16 bits operands. Furthermore, the design is modified to accommodate both signed and unsigned operands. Lastly multiply accumulate unit has been implemented to perform multiplication and accumulate the result in the next result. Xilinx environment is used to develop the VHDL code of whole algorithm and multiple test benches have been developed to test the functionality of Code under different signal conditions.

CHAPTER 1: Introduction and Background

Over the last few decades, there has been significant advancement in the field of digital computing and processing. With the advent of transistor, the vacuum tubes in early computers have been replaced and the computing shifts from slow and costly mainframes to very fast and smaller processors empowered through chip technology and integrated circuits. All the digital computing, including highly efficient and high-performance VLSI relies on Arithmetic Unit (ALU). Different arithmetic units are used which help in overcoming implementation issues in complex systems and finding the right compromise between area, power and delay.

ALU is a digital circuit used to perform arithmetical and logical operations on binary numbers and employed in the form of combinational digital circuit. In the field of computer science, it is regarded as the fundamental memory block of arithmetic and logic circuits for different controlling, computing and processing circuits such as CPUs, FPUs and graphic processing units.

It is regarded as the most critical component of CPU and most of CPU operations are performed by one or more ALU in which the data is loaded through input registers. The data reside in small available CPU storage termed as register and the control decides the operation to be performed on the input. After data operation, ALU stores the obtained result in the output register and can generate different flags to show the nature of the obtained such as the zero, carryout or overflow. These flags are very important as they allow computational machines to carry out more complex tasks such as conditional branching.

For this project, we have designed an ALU which can load the values from the input register and perform different operations such as multiplication, division and increment operation to solve the described equation.

Addition is the fundamental block of arithmetic circuit and can be used for the implementation of other blocks such as multiplication, subtraction and division. For example, multiplication operation can be carried out through addition circuit using repeated addition and subtraction can be performed taking two's complement of addition. For division, subtraction can be performed repetitively.

For multiplication of two 8-bit numbers, there is a choice of different multiplication methodology in which most prominent are 8-bit array multiplier and booth multiplier. We have chosen 8-bit multiplier for this project because it has easy layout and small size due to its regular structure. Its implementation is easier and provides easy design for pipeline architecture.

For the designing of register D flip flops are implemented and the ALU control unit consist of different functions such as clock, load clear and end flag in which load clear will clear out all the values in the register and end flag will be raised after the result is generated. The architecture of ALU is designed in structural manner in which various design blocks are connected and the top module is implemented through FSM. Furthermore, multiple test benches are developed to test the functionality of system.

CHAPTER 2: Design and Implementation of Adder

In electronics, an adder is a digital circuit used to perform the addition of two numbers. They are an essential part of ALU unit as they are basic building blocks for the development of other design components. We have implemented half and full adder for this project which is discussed below in detail.

- **Half adder**

Half adder is a combinational logic circuit designed through using AND and XOR gates. It consists of two input terminals A and B which are added to obtain a carry and sum as output. The sum is obtained from the output of XOR gate and the carry through AND gate. Half adder can only add current input numbers and does not support the previous inputs. They are used for the implementation of Array multiplier, Incrementor and multiply accumulator unit. The gate diagram of half adder can be observed in the figure below:

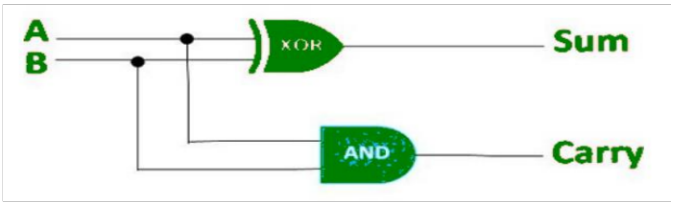


Figure 1: Half Adder Detailed Implementation

- **Full adder**

The full adder circuit comprised of two AND gates, two XOR gates and one OR gate. Unlike half adder, full adder can easily carry the current inputs as well as the output resulting from previous addition. They are used to add three inputs which are inputs A and B and CIN resulting from previous addition. The output generated will be COUT and SUM. SUM is generated through XOR of three inputs and COUT through AND and OR gates of inputs. They are used in the implementation of array multipliers. The block diagram can be observed in the figure below.

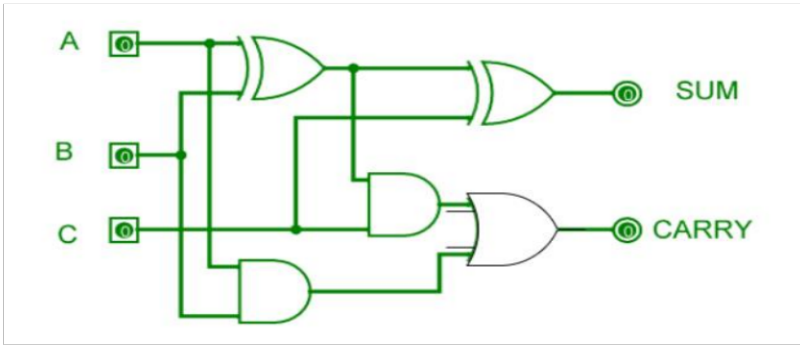


Figure 2: Full Adder Detailed Implementation

CHAPTER 3: Input & Output Register Implementation

Registers are fastest memory block of CPU which are used to temporarily hold the data. They are considered as the backbone of memory system and are frequently used in the system requiring pipelining. They can be implemented through using D flip flop (DFF) or latch. For this project, registers are implemented through flip flop which are simple if/else condition and it make designing much simpler and timing violation easier to handle. Since we have performed the expansion of design to 16 bits, so register used in that design method will be discussed.

For this project, two registers RA and RB are used to hold two 16 bits input values of the operands on which the operation needs to be performed to produce a 32-bit output value. The entity of input registers has 4 input pins which are data_in, clear, clock, and load and 1 output i.e., data_out. Similarly, the output register entity has 3 input pins namely data_in, clear, clock and 2 output which data_out and end_flag.

The VHDL coding for register implementation can be discussed now. For input registers, firstly the clear signal will be checked and if it is 1 then all the values in register will be cleared. The input values are only latched in to input registers when the clear signal is 0 and clock signal appears with the rising clock edge event with load value equals to 1. The generated result after multiplication, division and increment will be of 32 bits. Therefore 32-bit output register Z is designed to store the output value. Similarly, the output register will check clear and clock to store output value. After generating the result, the End flag signal will be raised high. This logic can observe in the code below:

```
entity reg is
Port ( data_in  : IN std_logic_vector (7 downto 0);
       clr1  : IN std_logic;
       clk1 : IN std_logic;
       --load1 : IN std_logic;
       data_out  : OUT std_logic_vector (7 downto 0)
     );
end reg;
```

```
architecture Behavioral of reg is
begin
process(clr1,clk1)
begin
   if clr1='1' then
     data_out <= (OTHERS=>'0');
   elsif (clk1'event and clk1 = '1') then
     data_out<=data_in;

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

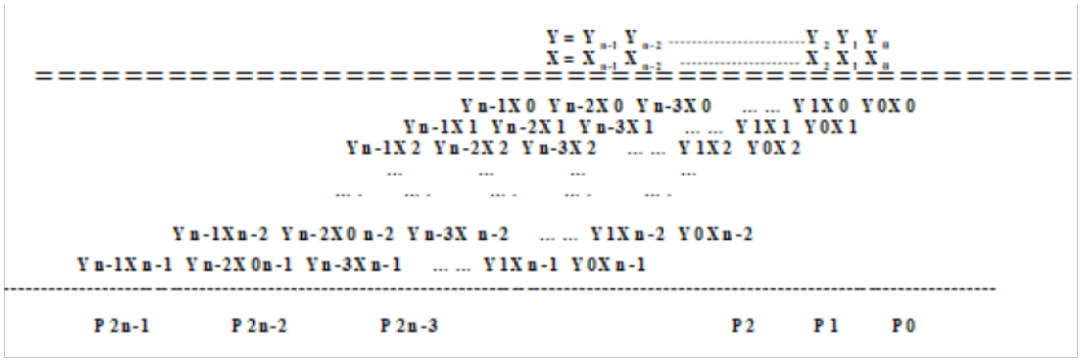
Figure 3: Entity and architecture of Registers

CHAPTER 4: Multiplier Design and Implementation

In this contemporary era, multipliers have been considered as one of the essential components of digital signal processing and many other applications. As the technology advances, scientist and researchers are focusing on developing new methodologies to design such a multiplier that has lower power consumption, high processing speed and have lesser area so that a compact VLSI implementation can be achieved.

Among these methodologies, 'add and shift' is considered as one of the simplest and most frequently used multiplication method. In this method, partial products are obtained which are then added through various methods to get the final multiplication result.

The figure below shows the procedure of multiplication for N bit multiplier and multiplicand:



In this method, the partial products of X and Y (i.e. X0Y0, Y1X0 etc.) are obtained through AND gates. The number of partial products depends upon the number of bits of multiplier and multiplicands and is equal to (N*M).

- **Array Multiplier:**
  Array multiplier is a multiplication method that is based on shift and add methodology. In our project, we have used this technique because of its simplicity and ease of design for pipeline architecture. Like shift and add method, multiplier is first multiplied by each bit of multiplicand using AND gates to obtain (N*M) partial products. These products are then added using full adders and half adders to obtain the result. The figure below illustrates basic block diagram of 16*16 array multiplier:

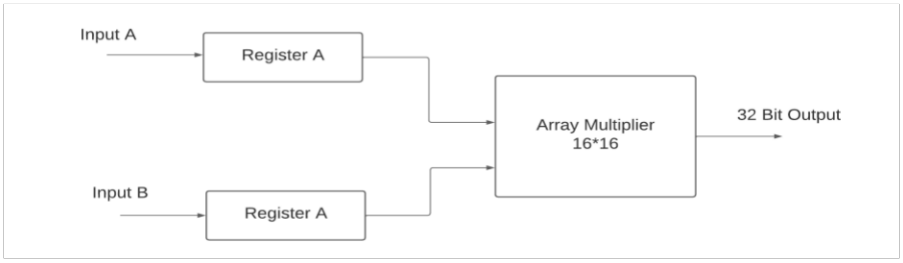


Figure 4: Basic Block Diagram of 16-bit Multiplication

Chapter 5: Design of Division Unit

In contrast with multiplication, the division process is more complex as it requires complex logic circuits. To perform simple division, we can use subtraction method. But in case, when the division is to be performed by a number which is either 2 or power of 2, then we can perform division by simply right shifting the binary number 'n' times depending upon the power of 2.

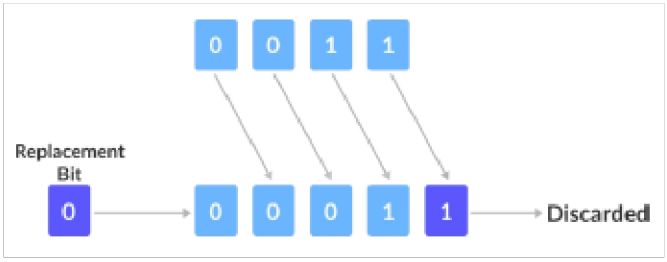


Figure 5: Simple Right Shift Method

Since in our project, we have to perform the division by 4 which is actually 2 to the power 2. So, we just need to right shift the output of the multiplier two times to divide by 4. Thus, to obtain the division result, we concatenated two 0s with the output of multiplier and discarded the last two least significant bits which are (1st and 0th bits). The code doing this functionality can be seen from following figure:

```
architecture arch of shifter is
begin
  shift_out<="00"&shift_in(15 downto 2);
end arch;
```

Figure 6: Xilinx Code for Right Shift Block

Chapter 6: Incremental Unit Implementation

The main purpose of incremental unit in our project is to perform the increment operation in a binary number by 1 bit. This means that if we provide an N-bit number at the input of this incremental block, the result obtained at the output will be (N+1).

In our project, to perform this incremental functionality in 16-bit multiplication, we declared a signal which is instantiated during the declaration by "00000000000000000000000000000001" that is equivalent to decimal 1.

Since the output of shifter which is coming at the input of our incremental block has to be incremented by 1 bit. Thus, 32 half adders are used for this purpose. The LSB of declared signal and the LSB of input are added through the first half adder. The carry generated is then propagated to the next half adder

which is then added with the 2nd bit of the incrementer input and this process is repeated for all bits. In the end, the sum of each half adder is loaded in the output bits of the incremental unit.

Chapter 7: Top Module Implementation

In the top-level module, inputs and outputs for the ALU are defined, all of the components of the ALU are called as components and their port mapping is done in the sequential order. Top level module is implemented by two methods which are given below:

1. Without FSM
2. With FSM

Each of these implementations is given below:

- **Top-Level Module without FSM**

When the top module is implemented without FSM then in that case, when the register receives the load signal as '1', it will latch the data and all of the calculations will be performed in the ALU unit to get the desired result. After generating the output, it will raise the end flag signal.

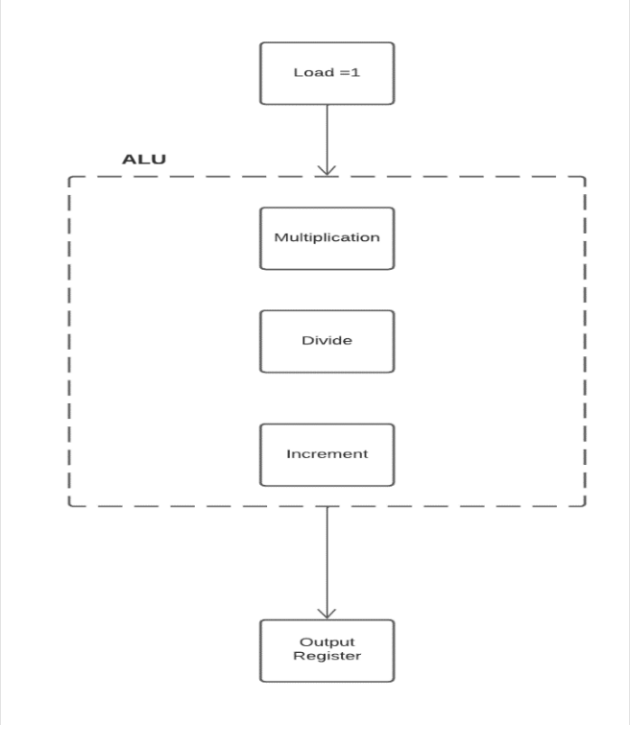


Fig 7: Flowchart for Implementation without FSM

- **Top Level Module with FSM**

When the top-level module is implemented with finite state machine, the process is executed upon reception of clock signal. Each stage is performed in a separate clock cycle. The top module implemented for this case includes a total of 7 states. Each of these 7 states is triggered when the machine receives the positive edge of clock cycle. When the machine is in Idle state then if it receives a load signal '1' then it will jump to next state which is load_val state otherwise it will remain at the Idle state. Once it is transitioned to load_val state then it will go through every state one by one with positive edge of clock cycle to generate output and raises end flag. Once it reaches, raise_flag state, then it will check the load signal. If the load signal is '1' then it will again move to load_val state otherwise it will go the idle state. To achieve better performance, back-to-back operation is implemented in the top module so that it can go to the load_val state with first going to the idle state. The state diagram for the top module implanted in this project is given below:

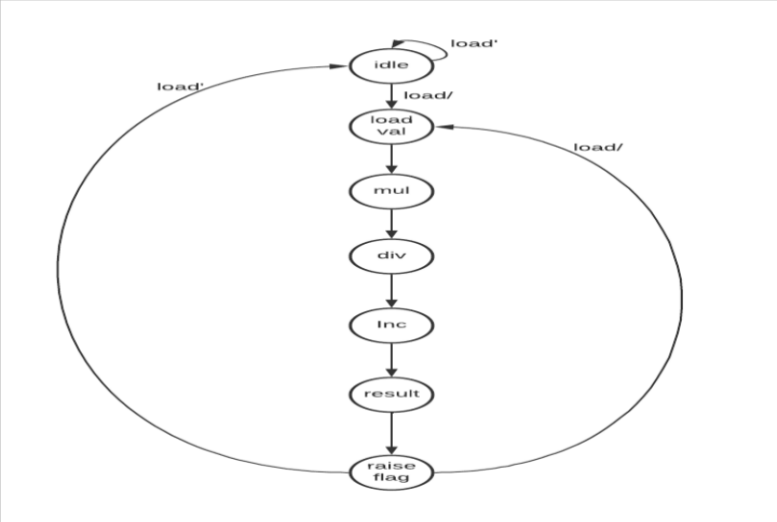


Fig:8 State diagram for implementation with FSM

HDL synthesis of the top module showcases the 7 states of the FSM mentioned above. Reset state and power up state is mentioned as idle state in the synthesis report which is highlighted by below given figure.

States	7	
Transitions	8	
Inputs	1	
Outputs	7	
Clock	clk	(rising_edge)
Reset	clr	(positive)
Reset type	asynchronous	
Reset State	idle	
Power Up State	idle	
Encoding	automatic	
Implementation	LUT	

Below picture indicates the states of the FSM and their respective encoding.

```
Analyzing FSM  for best encoding.
Optimizing FSM  on signal  with one-hot encoding.
-----------------------
 State       | Encoding
-----------------------
 idle        | 0000001
 load_val    | 0000010
 mul         | 0000100
 div         | 0001000
 incr        | 0010000
 result      | 0100000
 raise_flag  | 1000000
-----------------------
```

The RTL schematic of the top module implemented with FSM is given below:

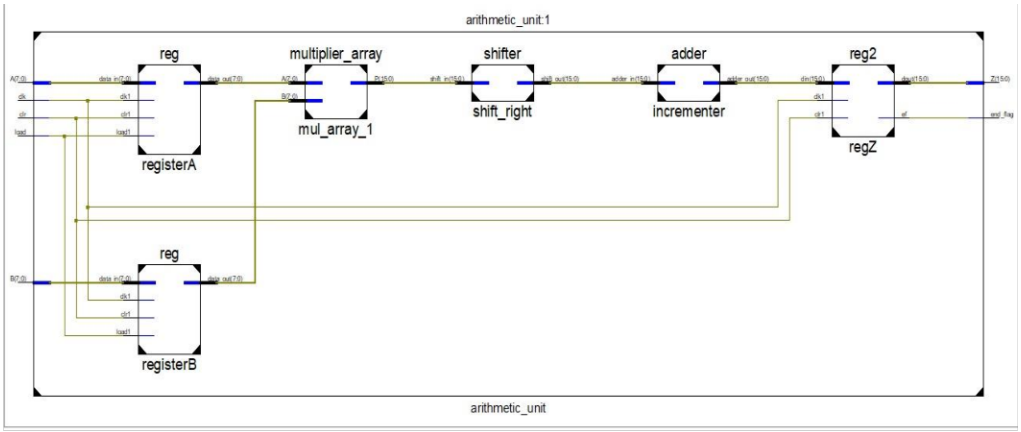


Fig:9 RTL Schematic of ALU

```
type states is (idle,load_val,mul,div,incr,result,raise_flag);
signal state_reg, state_next :states;

 signal ra,rb: std_logic_vector(15 downto 0);
 signal s1,s1_out: std_logic_vector(31 downto 0);

 signal s2,s3,s2_out,s3_out: std_logic_vector(31 downto 0);
signal ra_out,rb_out: std_logic_vector(15 downto 0);
signal z_out :std_logic_vector(31 downto 0);
signal mula,mulb :std_logic_vector(15 downto 0);
signal diva :std_logic_vector(31 downto 0);
signal incra :std_logic_vector(31 downto 0);
signal rc,rc_out : std_logic_vector(31 downto 0);
 begin
 registerA : reg port map (ra, clr, clk,  ra_out);
 registerB : reg port map (rb, clr, clk,  rb_out);
 mul_array_1 : multiplier_array port map(mula, mulb, s1);
 shift_right : shifter port map(diva, s2);
 incrementer : adder port map(incra,s3);
 regZ : reg2 port map(rc,clr,clk,rc_out);
```

Fig:10 Port Mapping of functional blocks

Above figure indicates the declaration of the states of the finite state machine, signal declaration and the port mapping of the top module of the finite state machine.

```
process (clk, clr)
begin
if (clr = '1') then
state_reg <= idle;
elsif (clk'event and clk = '1') then
state_reg <= state_next;
end if;
end process;

process (state_reg, load)
begin
case state_reg is
when idle =>
   if load = '1' then
      state_next <= load_val;
   else
      state_next <= idle;
   end if;
when load_val =>
      state_next <= mul;
when mul =>
      state_next <= div;
when div =>
   state_next <= incr;

when incr =>
   state_next <= result;
when result =>
   state_next <= raise_flag;
when raise_flag =>
   if (load = '1') then
      state_next <= load_val;
   else
   state_next <= idle;
   end if;
end case;
end process;
```

Above is the code for the state_reg logic of the finite state machine. state_reg and load are added in the sensitivity list. When the process starts from idle state then it will check whether the load is '1'. If it is '1' then state_next would be load_val else the state_next would remain the Idle. When the state is Load_val then the next state would be mul, when state is mul then state-next would be div, when state is div then next_state would be incr, when incr, state_next would be result, when result then next_state would raise_flag. When in state raise_flag, it will check whether load is '1', if so state_next would be load_val else Idle.

```
process (state_next)
begin
   mula <= (others => '0');
   mulb <= (others => '0');
   diva <= (others => '0');
   incra <= (others => '0');
   end_flag <= '0';
case state_next is
when idle =>
   ra <= ra_out;
   rb <= rb_out;
   rc <= (others => '0');

when load_val =>
   ra <= A;
   rb <= B;
   rc <= (others => '0');

when mul =>
   ra <= ra_out;
   rb <= rb_out;
   rc <= rc_out;

   mula <= ra_out;
   mulb <= rb_out;
when div =>
   ra <= ra_out;
   rb <= rb_out;
   rc <= rc_out;

   diva <= s1;
when incr =>
   ra <= ra_out;
   rb <= rb_out;
   rc <= rc_out;

   incra <= s2;
when result =>
      ra <= ra_out;
      rb <= rb_out;

      rc <= s3;
when raise_flag =>
      end_flag <= '1';


end case;
end process;
 Z <= std_logic_vector(rc_out);
```

Above is the VHDL code for state_next of the FSM. Initially, mula, mulb, diva, incra, end_flag is initialized with '0'. When the state is IDLE then output is set as '0'. When the state is load_val then it will load the value into register A and B. When the state is mul then it will do the multiplication. When the state is div then it will do Division, when incr then it will increment by 1 and finally when the state is result then it will display the result and raise the end_flag signal.

- **Simulation for 16-Bit Operands Unsigned**

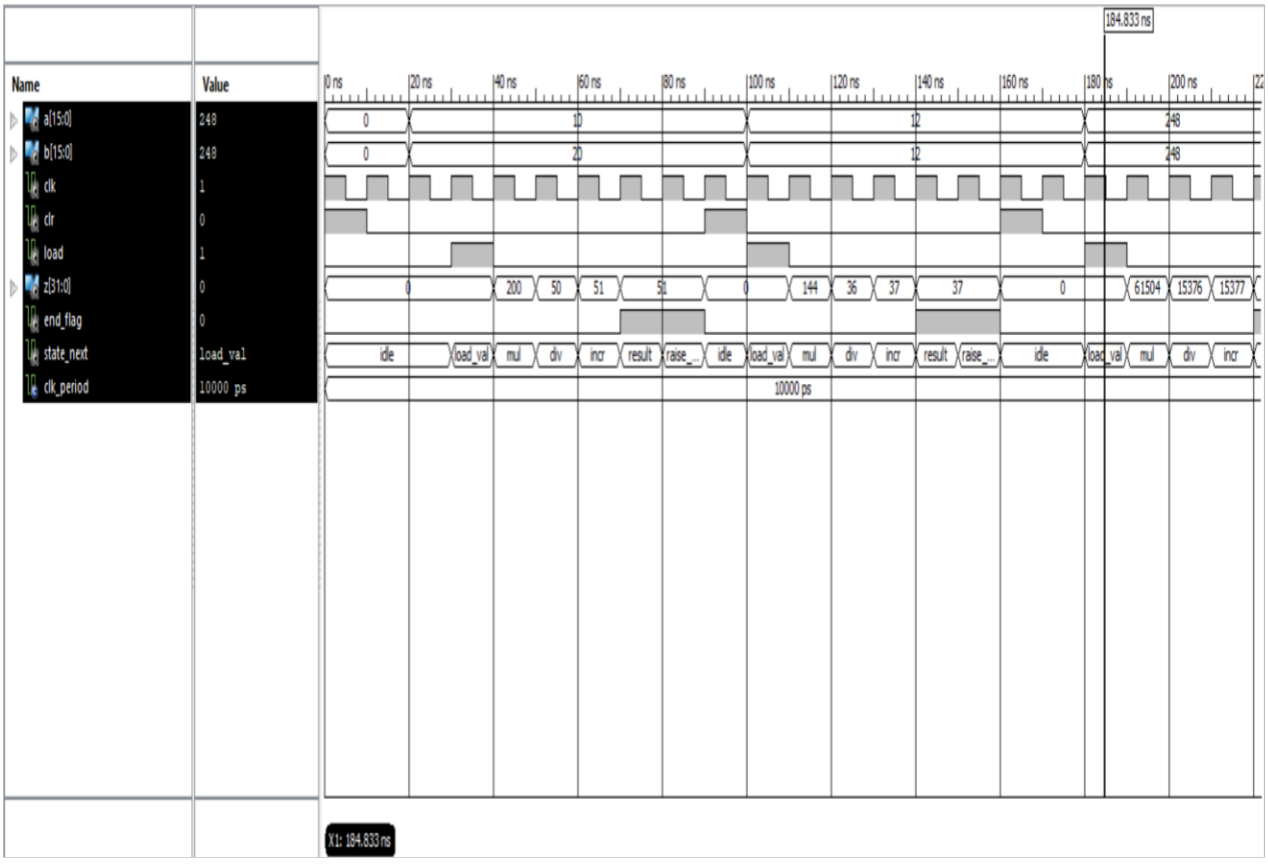


Fig:11 Timing diagram for 16-bit Unsigned operation

Chapter 9: 8-Bit Signed Implementation

Signed multiplication is an important functionality of any arithmetic unit. In our project, we have also implemented the signed number multiplication. For this purpose, we have formulated an algorithm and on the basis of that algorithm, calculation of the given input numbers is performed. This algorithm can be seen from the following figure:

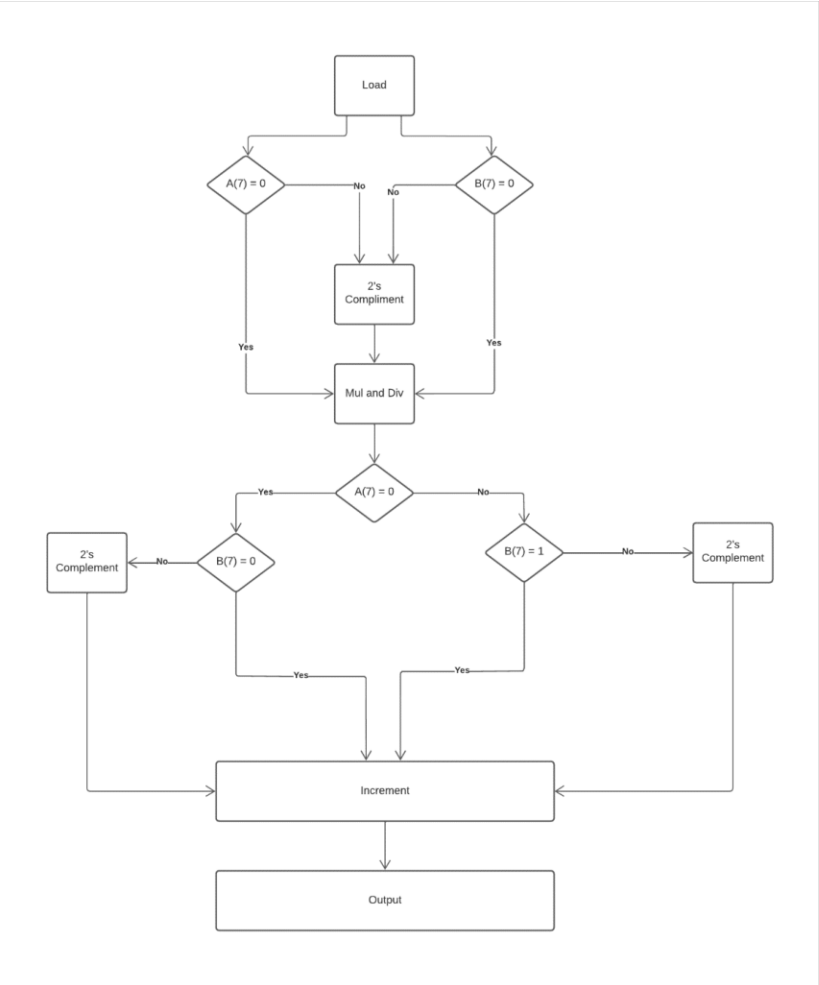


Figure 12: Algorithm for Signed Number Calculation

As we know from normal math concept, the easiest way to do the signed multiplication is to take the magnitude of the multiplicand and multiplier, and then use the original sign to determine the sign of the result. If both signs are positive or negative the result will be positive but if any one of them was negative then output will be negative.

We have used the same concept in our signed number calculation. The signed inputs A and B are first stored in the registers A and B upon receiving the positive edge of the clock and load='1' signal. In

multiplication unit, the MSBs of both ‘A’ and ‘B’ are checked. If any of them is ‘1’ which implies that the number is negative, we took the 2s complement of that number so that it becomes positive.

To take 2s complement, all the bits are inverted using the NOT gate and then it is incremented by 1. This can be seen from a section of code in following figure:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity multiplier_array is
 Port ( A ,B : IN std_logic_vector(7 DOWNTO 0);
        P : OUT std_logic_vector(15 DOWNTO 0) );
end multiplier_array;

architecture Behavioral of multiplier_array is
signal C1,C2,C3,C4,C5,C6,C7: std_logic_vector(7 DOWNTO 0);
signal S1,S2,S3,S4,S5,S6,S7: std_logic_vector(7 downto 0);
signal AB1,AB2,AB3,AB4,AB5,AB6,AB7,AB8:std_logic_vector(7 downto 0);
signal c,d: std_logic_vector(7 DOWNTO 0);
signal x,y: std_logic_vector(7 downto 0):="00000000";
signal x1,y1: std_logic_vector(7 downto 0):="00000000";
signal out2,out1: std_logic_vector(15 downto 0);


component full_adder port(AF,BF,CI : IN std_logic;
                          SF,CF :OUT std_logic);
 END component;
 COMPONENT half_adder port (AH,BH :IN std_logic;
                            SH, CH :out std_logic);
                            end component  ;
begin
process(a,b,x,y,x1,y1)
begin
```

```
if A(7)= '1' then
 X(7) <= (not A(7)) ;
  X(6) <=(not A(6)) ;
  X(5) <=(not A(5)) ;
 X(4) <= (not A(4))  ;
 X(3) <=(not A(3))  ;
 X(2) <=(not A(2))  ;
  X(1) <=(not A(1));
 X(0) <=(not A(0));

   X1 <= std_logic_vector(unsigned(X)+1);

 c<=  x1;
else
 c<= A;

end if ;

 if B(7)= '1' then
 Y(7) <= (not B(7)) ;
  Y(6) <=(not B(6)) ;
  Y(5) <=(not B(5)) ;
 Y(4) <= (not B(4))  ;
 Y(3) <=(not B(3))  ;
 Y(2) <=(not B(2))  ;
  Y(1) <=(not B(1));
 Y(0) <=(not B(0));

   Y1 <= std_logic_vector(unsigned(Y)+1);

 D<=  Y1;
else
 D<= b;

end if ;

end process;
```

Figure 13: 2s Complement Code in Multiplier Block

After that, the multiplication and division are performed similarly as discussed in above sections for unsigned number. The output of the division unit is then fed to the two’s complement converter unit where we again checked the MSB of inputs A and B. If the MSB of both the inputs are 1 or 0 then we know that the final answer will be positive. Thus, the output was directly fed to incremental unit where it is incremented by 1 using 16 half adders (as discussed above in incremental unit). However, if MSB of any one input is 1, then we need to take the 2s complement of the result coming from division unit to convert that result back to its expected sign. This process can be seen from following code snip:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
 use IEEE.std_logic_signed.all;
entity converter is
  port(
    data_in: IN std_logic_vector(15 downto 0);
    A,B: IN std_logic_vector(7 downto 0);
    data_out: OUT std_logic_vector(15 downto 0)
      );
end converter;
architecture conv_arch of converter is
signal out_con :std_logic_vector(15 downto 0);
BEGIN
  Process(a,b,data_in,out_con)
  begin
  if ((A(7)= '0') and (B(7)= '0')) then
   data_out <= data_in;
  elsif ((A(7)= '1') and (B(7)= '1')) then
      data_out <= data_in;
  elsif ((A(7)= '1') or (B(7)= '1'))  then
OUT_CON(15) <= (not DATA_IN(15)) ;
  OUT_CON(14) <=(not DATA_IN(14)) ;
  OUT_CON(13) <=(not DATA_IN(13)) ;
 OUT_CON(12) <= (not DATA_IN(12))  ;
 OUT_CON(11) <=(not DATA_IN(11))  ;
 OUT_CON(10) <=(not DATA_IN(10))  ;
  OUT_CON(9) <=(not DATA_IN(9));
 OUT_CON(8) <=(not DATA_IN(8));
 OUT_CON(7) <= (not DATA_IN(7)) ;
  OUT_CON(6) <=(not DATA_IN(6)) ;
  OUT_CON(5) <=(not DATA_IN(5)) ;
 OUT_CON(4) <= (not DATA_IN(4))  ;
 OUT_CON(3) <=(not DATA_IN(3))  ;
 OUT_CON(2) <=(not DATA_IN(2))  ;
  OUT_CON(1) <=(not DATA_IN(1));
 OUT_CON(0) <=(not DATA_IN(0));

   data_out<= std_logic_vector(signed(out_con)+1);
   end if;
   end process;
end conv_arch;
```

Figure 14: Two's Complement Converter Component Code

After completing the creation of additional block for proper working, the Top model was slightly modified, and the RTL schematic was created. This RTL diagram can be seen from following figure:

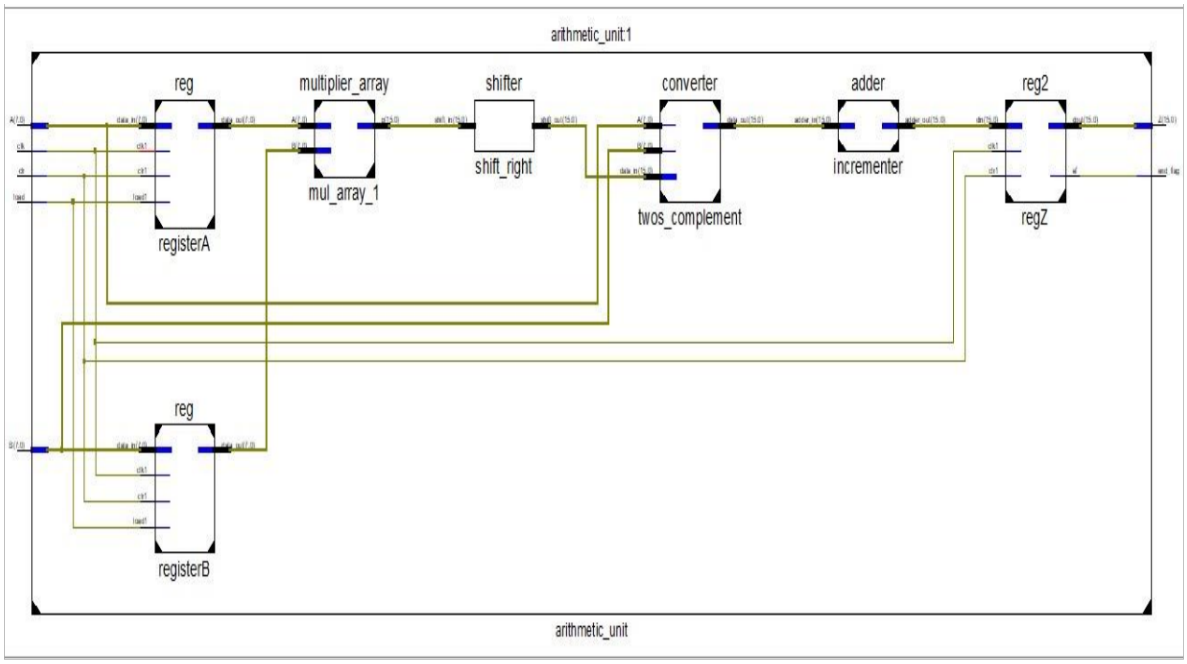


Figure 15: RTL Schematic for Signed Number Operation

Finally, a test bench was created to test the system for different inputs. In the test bench, we provided 3 different combinations of input to test the functionality. Firstly, one input is kept negative and other is kept positive i.e. a= -8 and b= +8. The output we received is -15 as expected. In second case, we provided

both negative number and output came as positive satisfying the equation. Finally both positive values of input were tested and we got expected results.

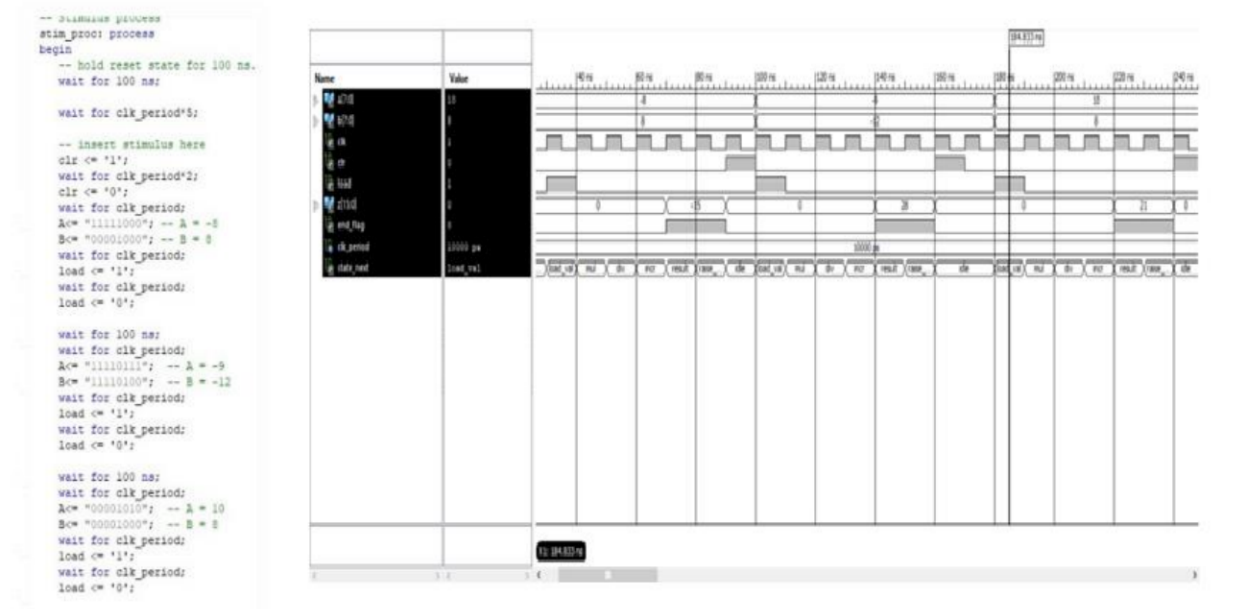


Figure 16: Test Bench and Timing Diagram

Chapter 10: Implementation of Multiply Accumulator Unit

In digital signal processing, where it is required to perform convolution, co-relation and most importantly fast Fourier transform, the multiply accumulator (MAC) unit is a necessary component as it performs the multiplication operation and accumulate the result which is then added to the next result. This accumulation of result can be calculation till n^{th} term and that is why it is a helpful block in performing operation like Fourier Transform, convolution etc. where we take the summation of given equation till n^{th} term.

To implement this MAC unit, we first devised a flow diagram. This diagram can be seen from figure X. The initial process is same as normal operation i.e. the input numbers are first stored in the register A and B upon receiving the load signal which are then forwarded to the multiplier block, followed by division and incremental unit. The change starts after the incremental unit. The output of incremental block is sent to accumulator adder where this new value is being added with the previous/old result stored in accumulator register. The output of this accumulator adder is then sent to output register Z and same value is stored in accumulator register via signal from register Z so that it can be added with the next value of incremental unit.

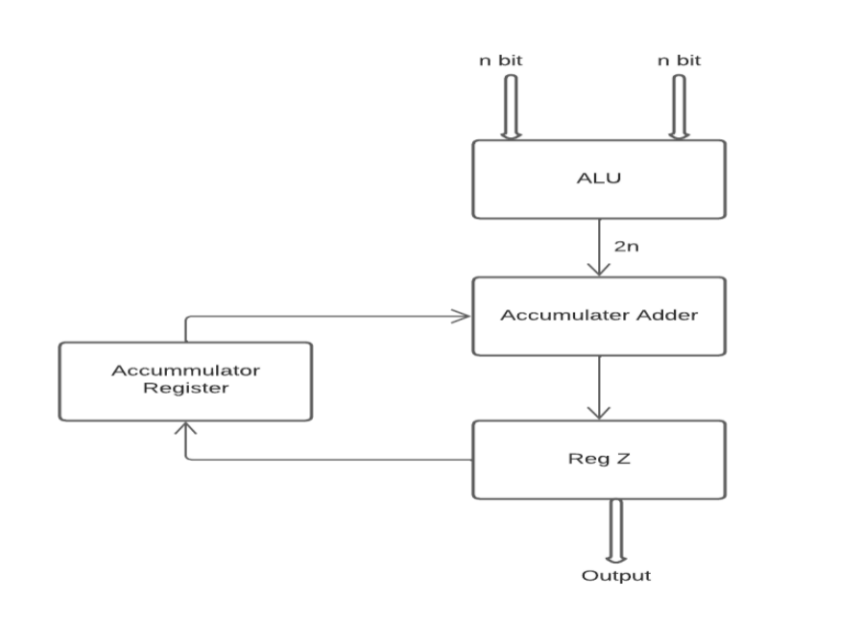


Figure 17: Process Flow Diagram of MAC

In this project, we have implemented this MAC functionality for two 8-bit input data. Now let us discuss how these two additional components used in MAC unit have been created through VHDL code. Firstly, for accumulator register, we have used the flip-flop methodology that we have used to create the input and output registers. This 16-bit accumulator register has clear, clock and load signal. When there is a clear signal then the value in accumulator register will be reset to zero. When there is positive edge of the clock, the accumulator register will take the value from reg Z and store it, so that it can be added with the next output result coming from incremental unit.

To design accumulator adder, as we know that we have to add the value stored in accumulator register and the output of incremental unit, we are required to use combination of half and full adders. To add the first 2 bits, since we only have two inputs, we have used a half adder. The sum is stored in the 0^{th} bit of result and carry is propagated for the next addition. For the next addition, we have 2 input bits and one carry propagated from previous addition. Thus, we have used full adder to perform this addition and similarly the sum is stored in the 1^{st} bit of result and carry is propagated for next addition. This process is repeated until all the bits are added.

After creating separate components, these components were port map as previously and slight changes were done in the top module to get appropriate input and output signals at each blocks. The RTL schematic of whole system with MAC can be seen from following figure:

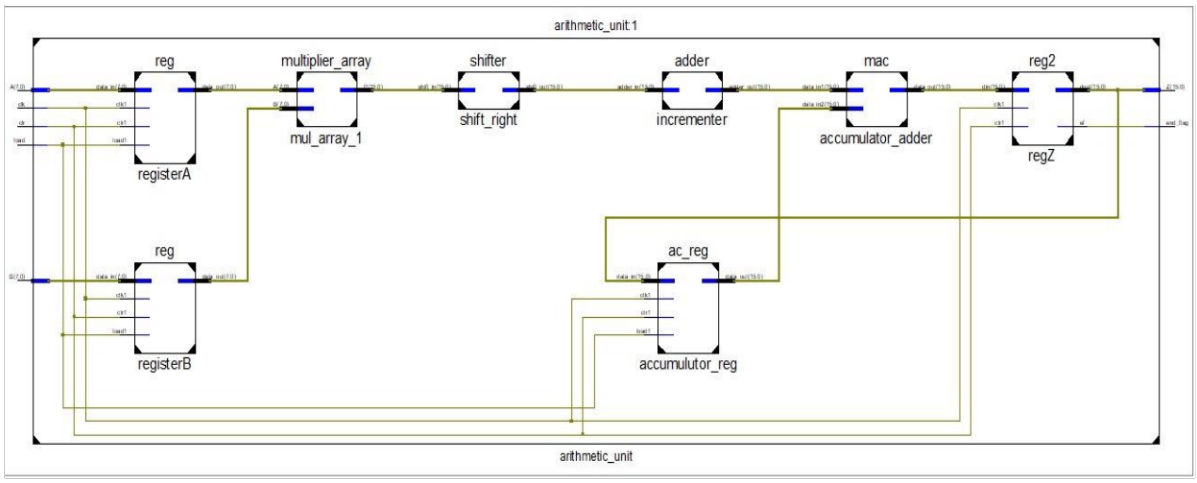


Figure 18: RTL Schematic of MAC

Finally to verify the functionality without finite state machine technique, test bench was createdwith clock period of 10ns. Initially clear signal was set to 0 to verify that the output is 0. Then at 40 ns load signal was set high, thus the value '953'(which is the solution of given equation) is loaded in the register Z and accumulator register. Upon receiving the next load signal the values in the input registers will be changed to the new values and thus new output will be calculated from equation. Here, previous result i.e. '953' will be added into this new value and thus we are getting output '2241'. This working can be verified from following figure:

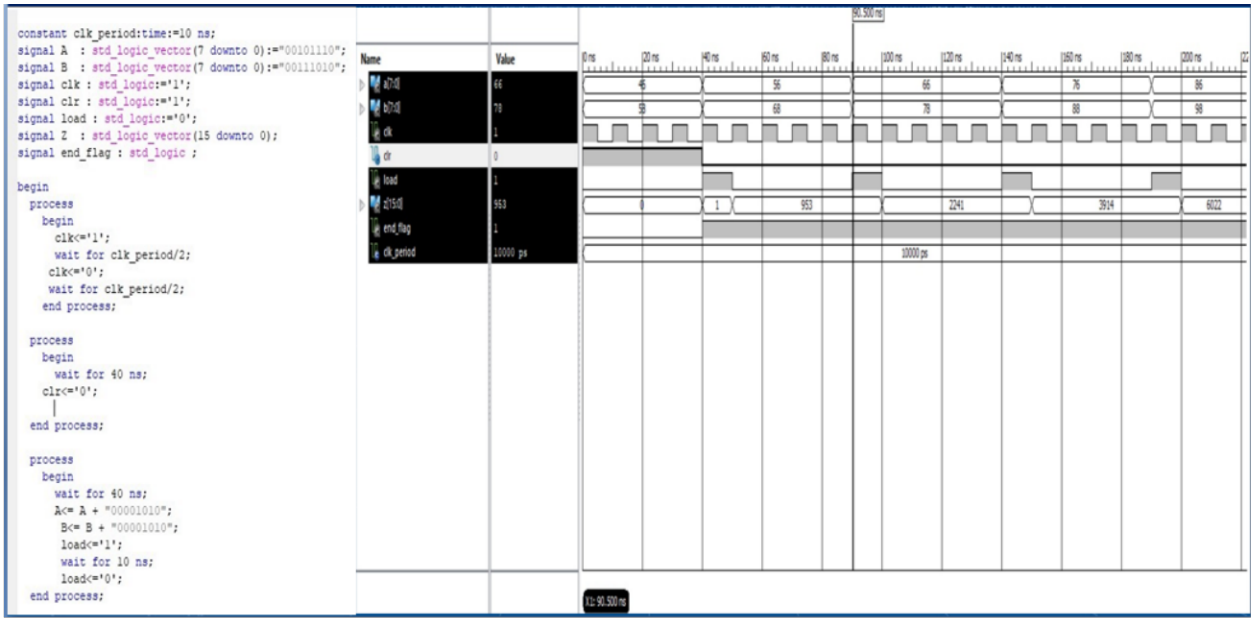


Figure 19: Test Bench and Timing Diagram

Chapter 11: Conclusion

This work presented the design and implementation of Arithmetic unit capable to solve Z= ¼ (A*B) +1 equation. For this purpose, two inputs and one output register are implemented through flip flop to hold the input values and output result. Both input and output registers will reset the value to 0 through clear signal. Values are loaded in input register through load =1 and output register will raise the end flag after generating result. Then to perform the desired operations, array multiplier, right shift and incrementor units are developed to perform multiplication, division and increment operations respectively. The top module is implemented with and without FSM. Furthermore, the design is expanded to facilitate 16-bit input operand and is capable to handle both signed and unsigned values. In addition, MLA unit has been implemented to perform multiplication and accumulate the result in the next result. Multiple test benches are developed to verify the functionality of code for each developed unit.