Group ID – 65

CODING PROJECT – 1

Date Of Submission – 1 November, 2016

Member #1 : Sandeep Pal (15114063)

Member #2 : Gautam Choudhary (15114027)

B.Tech. (Sophomores)

Comp. Sci. & Engg.

IIT Roorkee

Submitted to – Prof. Sudip Roy

Course : CSN-221: Computer Architecture and Microprocessors (Autumn 2016-2017)

Department of Computer Science and Engineering

Indian Institute of Technology Roorkee

# Problem Statement :

Write the Verilog code (.v) of an 8-bit ALU that can execute following four different operations:

a. Addition

b. Multiplication

c. AND

d. XOR

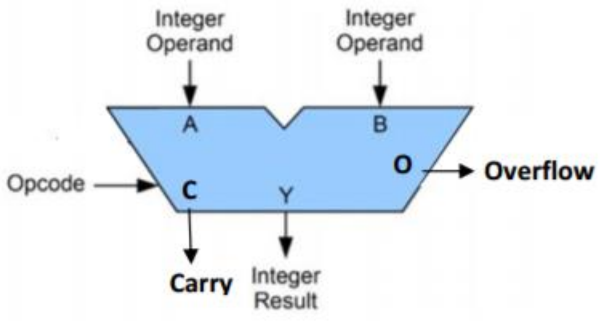
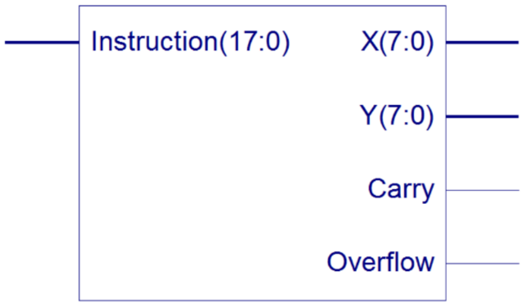


Fig.1: Arithmetic and Logic Unit (ALU)

Project Description (8Bit-ALU)

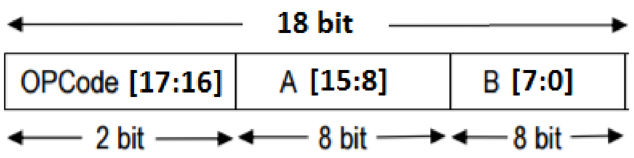
The project implements 8-Bit ALU with functionalities of ADDITION, MULTIPLICATION, AND & XOR.



Top level module is – **alu8bit.v** which implements sub modules as given:

- *addition_module_8bit.v*
    - *ripple_carry_adder.v*
- *multiplication_module_8bit.v*
- *and_module_8bit.v*
- *xor_module_8bit.v*
- *register_file_8bit.v*

INPUT : 18 bit instruction



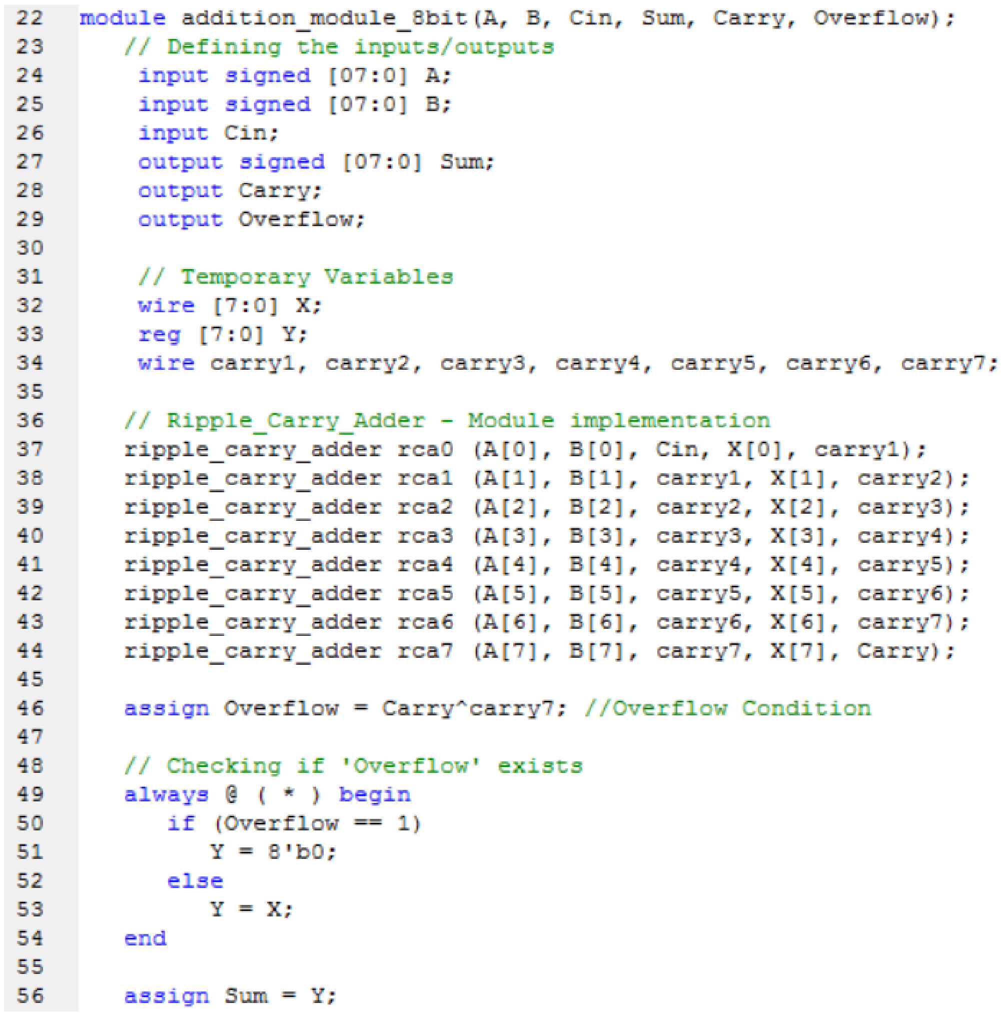
Input Instruction Format

OUTPUT :

- *X [8 bit]*
- *Y [8 bit]*
- *Carry*
- *Overflow*

Note : REGISTER FILE (4registers x 8bit) has also been implemented in the project.

ADDITION MODULE (addition_module-8bit.v)

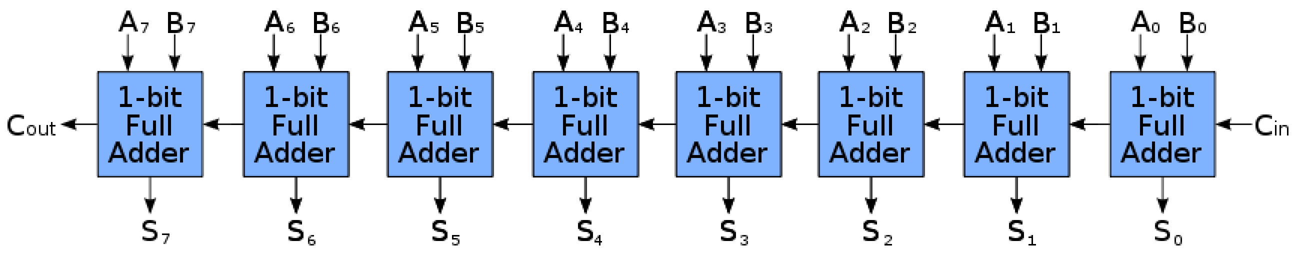


```
module addition_module_8bit(A, B, Cin, Sum, Carry, Overflow);
   // Defining the inputs/outputs
    input signed [07:0] A;
    input signed [07:0] B;
    input Cin;
    output signed [07:0] Sum;
    output Carry;
    output Overflow;

    // Temporary Variables
    wire [7:0] X;
    reg [7:0] Y;
    wire carry1, carry2, carry3, carry4, carry5, carry6, carry7;

   // Ripple_Carry_Adder - Module implementation
   ripple_carry_adder rca0 (A[0], B[0], Cin, X[0], carry1);
   ripple_carry_adder rca1 (A[1], B[1], carry1, X[1], carry2);
   ripple_carry_adder rca2 (A[2], B[2], carry2, X[2], carry3);
   ripple_carry_adder rca3 (A[3], B[3], carry3, X[3], carry4);
   ripple_carry_adder rca4 (A[4], B[4], carry4, X[4], carry5);
   ripple_carry_adder rca5 (A[5], B[5], carry5, X[5], carry6);
   ripple_carry_adder rca6 (A[6], B[6], carry6, X[6], carry7);
   ripple_carry_adder rca7 (A[7], B[7], carry7, X[7], Carry);

   assign Overflow = Carry^carry7; //Overflow Condition

   // Checking if 'Overflow' exists
   always @ ( * ) begin
      if (Overflow == 1)
         Y = 8'b0;
      else
         Y = X;
   end

   assign Sum = Y;
```

As the comments are stated in the code, this module produces the addition of two 8bit operands A & B.

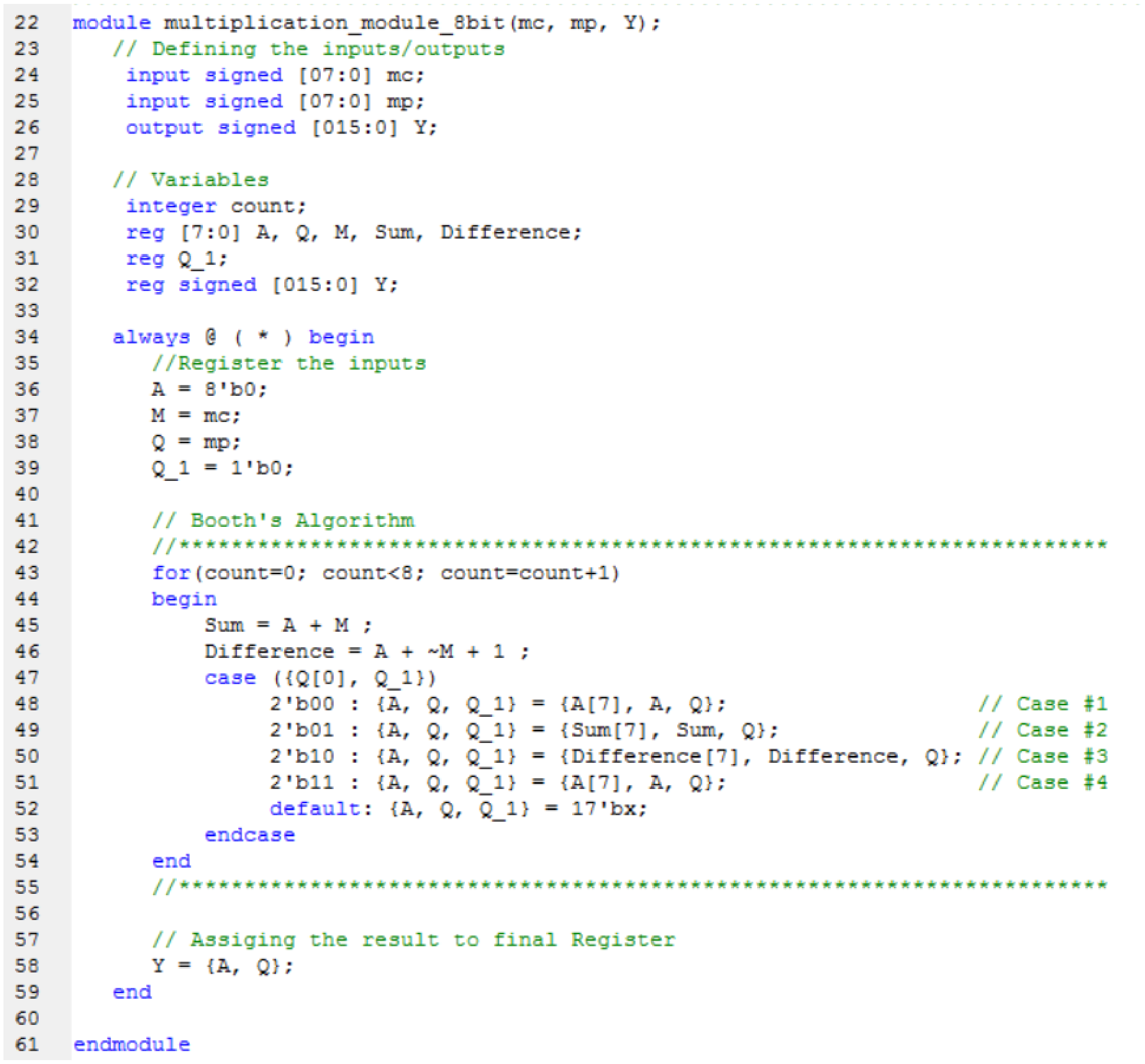
- **Carry** and **Overflow** flags have also been realized in this 8bit - FULL ADDER module.
- *Carry* bit is considered in case of UNSIGNED numbers.
- *Overflow* bit is considered for SIGNED numbers.
- Also, if the *overflow* bit is '1', the SUM is "**zero**".

➢ This 8bit FULL ADDER implements eight 1bit Ripple Carry Adders.



NOTE : This module works properly even with 'negative' numbers.

MULTIPLICATION MODULE (multiplication_module-8bit.v)



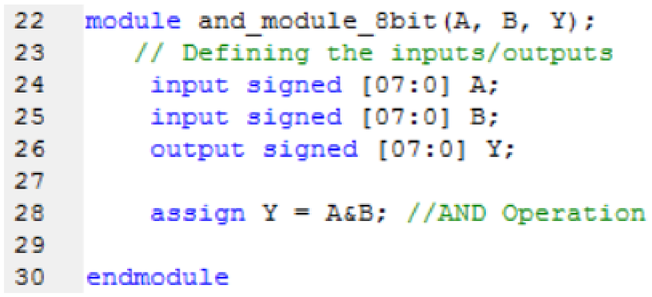
```verilog
module multiplication_module_8bit(mc, mp, Y);
   // Defining the inputs/outputs
    input signed [07:0] mc;
    input signed [07:0] mp;
    output signed [015:0] Y;

   // Variables
    integer count;
    reg [7:0] A, Q, M, Sum, Difference;
    reg Q_1;
    reg signed [015:0] Y;

   always @ ( * ) begin
      //Register the inputs
      A = 8'b0;
      M = mc;
      Q = mp;
      Q_1 = 1'b0;

      // Booth's Algorithm
      //**********************************************************************
      for(count=0; count<8; count=count+1)
      begin
          Sum = A + M ;
          Difference = A + ~M + 1 ;
          case ({Q[0], Q_1})
               2'b00 : {A, Q, Q_1} = {A[7], A, Q};                // Case #1
               2'b01 : {A, Q, Q_1} = {Sum[7], Sum, Q};            // Case #2
               2'b10 : {A, Q, Q_1} = {Difference[7], Difference, Q}; // Case #3
               2'b11 : {A, Q, Q_1} = {A[7], A, Q};                // Case #4
               default: {A, Q, Q_1} = 17'bx;
          endcase
      end
      //**********************************************************************

      // Assiging the result to final Register
      Y = {A, Q};
   end

endmodule
```

This module implements **BOOTH's ALGORITHM** for multiplication.

- It takes input as *Multiplier*(mc [8bit]) and *Multiplicand*(mp [8bit]) and gives output in the 16bit register-pair {A, Q}
- Iterations have been achieved by a '**for**' loop.

NOTE : This module works properly even with 'negative' numbers.

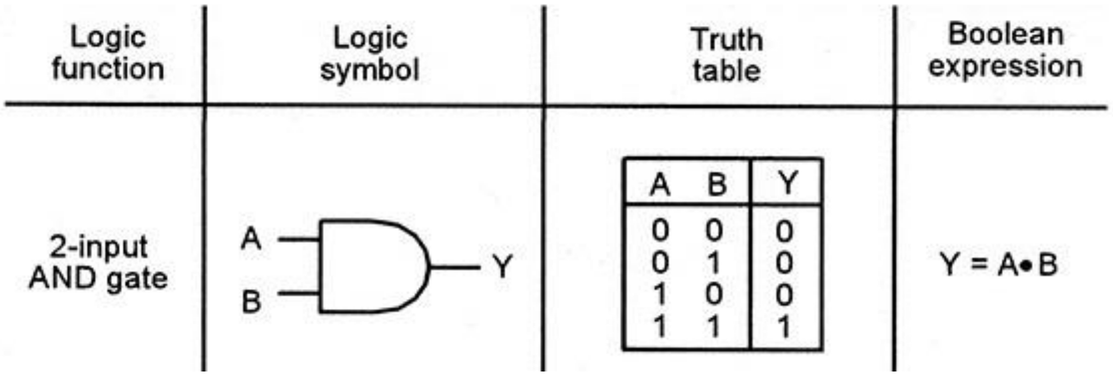
AND GATE MODULE (and_module-8bit.v)

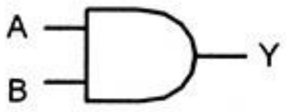


```
module and_module_8bit(A, B, Y);
   // Defining the inputs/outputs
    input signed [07:0] A;
    input signed [07:0] B;
    output signed [07:0] Y;

    assign Y = A&B; //AND Operation

endmodule
```

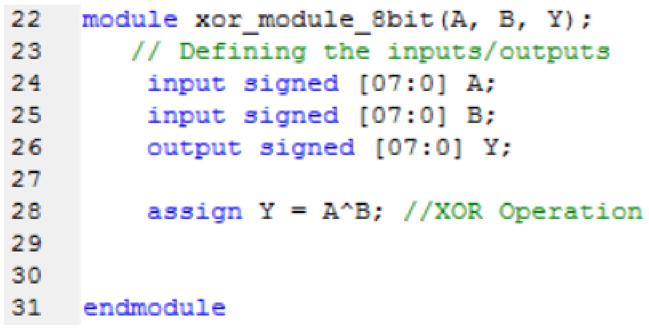
This is a simple **AND** gate module.

- It takes two **8bit** operands as Input and generates the ‘*bitwise*’ **AND** operation result of the two.



Logic function	Logic symbol	Truth table	Boolean expression
2-input AND gate		A B | Y 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1	Y = A•B

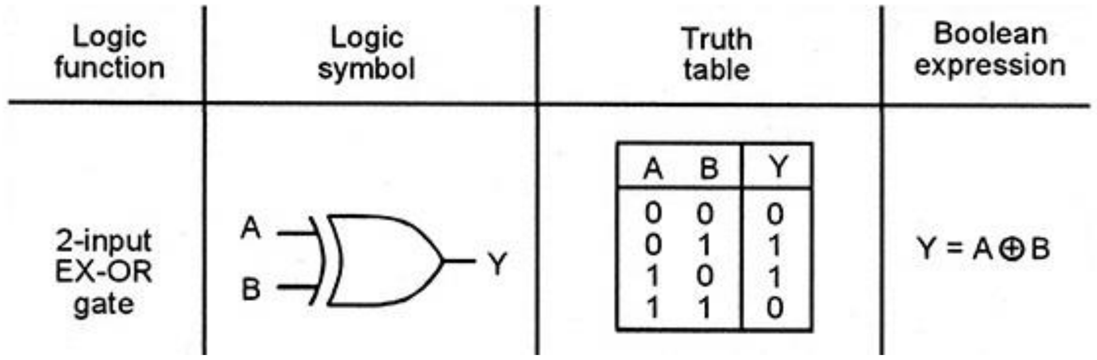
XOR GATE MODULE (xor_module-8bit.v)

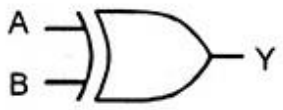


```verilog
module xor_module_8bit(A, B, Y);
   // Defining the inputs/outputs
    input signed [07:0] A;
    input signed [07:0] B;
    output signed [07:0] Y;

    assign Y = A^B; //XOR Operation


endmodule
```

This is a simple **XOR** gate module.

- ➢ It takes two **8bit** operands as Input and generates the ‘*bitwise*’ **XOR** operation result of the two.



Logic function	Logic symbol	Truth table	Boolean expression
2-input EX-OR gate	 A B Y	A B | Y 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0	$Y = A \oplus B$

# Testbench (alu8bit_tb.v)

The project is successfully ‘***synthesized***’ to test the **8bit ALU**.

Testbench file is created with the following ‘Input Test Cases’ :

- Test Case #1 :
  - Instruction – 18’b 00   0010 0001   0000 0111
    - *OpCode* – 00 (Addition)
    - A – 33
    - B – 7
  - OUTPUT
    - X – 0
    - Y – 40
    - Carry – 0
    - Overflow – 0

- Test Case #2 :
  - Instruction – 18’b 00   1101 0111   0000 1111
    - *OpCode* – 00 (Addition)
    - A – (-41)
    - B – 15
  - OUTPUT
    - X – 0
    - Y – (-26)
    - Carry – 0
    - Overflow – 0

- Test Case #3 :
  - Instruction – 18’b 00   0111 0111   0011 1001
    - *OpCode* – 00 (Addition)
    - A – 119
    - B – 57
  - OUTPUT
    - X – 0
    - Y – 0
    - Carry – 0
    - Overflow – 1

- Test Case #4 :
  - Instruction – 18’b 01   0000 0011   0001 1101
    - *OpCode* – 01 (Multiplication)

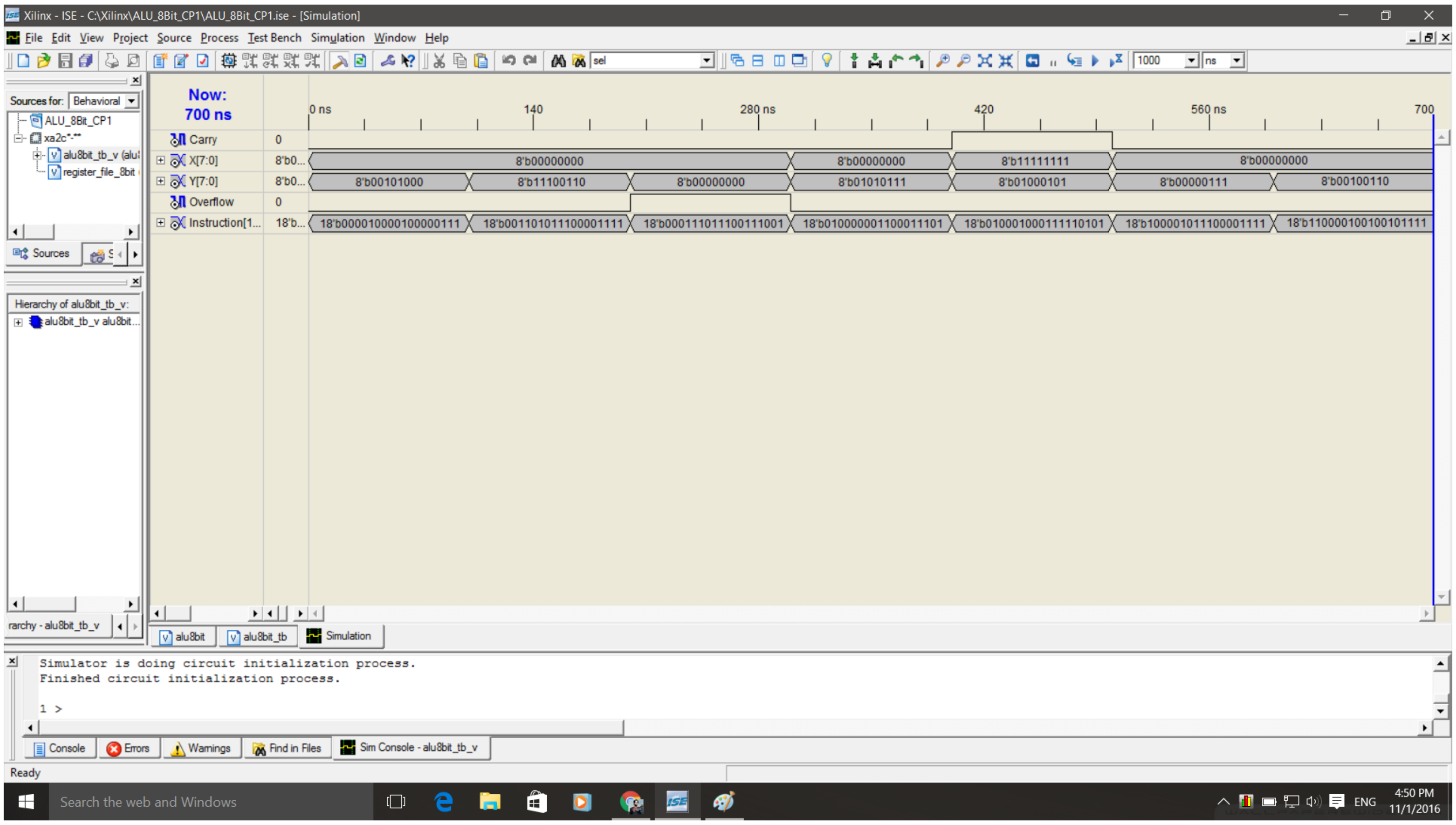
        - A – 3
        - B – 29
    - OUTPUT
        - X – 0
        - Y – 87
        - Carry – 0
        - Overflow – 0

- Test Case #5 :
    - **Instruction** – 18'b 01   0001 0001   1111 0101
        - *OpCode* – 01 (Multiplication)
        - A – 17
        - B – (-11)
    - OUTPUT
        - X – 69
        - Y – 255
        - Carry – 1
        - Overflow – 0

- Test Case #6 :
    - **Instruction** – 18'b 10   0001 0111   0000 1111
        - *OpCode* – 10 (AND)
        - A – 23
        - B – 15
    - OUTPUT
        - X – 0
        - Y – 7
        - Carry – 0
        - Overflow – 0

- Test Case #7 :
    - **Instruction** – 18'b 11   0000 1001   0010 1111
        - *OpCode* – 11 (XOR)
        - A – 9
        - B – 47
    - OUTPUT
        - X – 0
        - Y – 38
        - Carry – 0
        - Overflow – 0

All the outputs match with the expected values. Hence, ALU works properly.



```
Xilinx - ISE - C:\Xilinx\ALU_8Bit_CP1\ALU_8Bit_CP1.ise - [Simulation]

Now: 700 ns        0 ns        140        280 ns        420        560 ns        700

Carry            0
X[7:0]           8'b0...   8'b00000000 | 8'b00000000 | 8'b11111111 | 8'b00000000
Y[7:0]           8'b0...   8'b00101000 | 8'b11100110 | 8'b00000000 | 8'b01010111 | 8'b01000101 | 8'b00000111 | 8'b00100110
Overflow         0
Instruction[1... 18'b...   18'b000010000100000111 | 18'b001101011100001111 | 18'b000111011100111001 | 18'b010000001100011101 | 18'b010001000111110101 | 18'b100001011100001111 | 18'b110000100100101111
```

```
Simulator is doing circuit initialization process.
Finished circuit initialization process.

1 >
```

Timing diagram of the 8bit ALU