PROJECT

EEE412: VLSI Design Laboratory (Sec-01)

Project title : Design of 4 bit ALU

Group - 08

Name	ID
Sk Tahmed Salim Rafid	19121028
Asef Jamil Ajwad	19121040
Pronoyan Saha	19121131

Declaration: We certify that this project is entirely our own work except where we have fully documented references to the works of others, and that the material contained in this project has not been submitted previously for assessment in any formal course of study.

Objective:

The objective of this project is to design a 4-bit ALU capable of performing 8 different arithmetic and logical operations using Verilog HDL and verify the design using timing diagrams.

Introduction:

In digital electronics, ALU stands for arithmetic logic unit. It is a digital circuit that performs arithmetic and bitwise logical operations on integer binary numbers. In this project we will design a 4-bit ALU capable of performing 8 different arithmetic and logical operations using verilog code. In our design our ALU will have a 12 bit input signal. After operating on them the ALU will output a 4 bit result. Additionally, there will be three bits, one bit output which will be high if the result overflows denoted as carry, one bit output which will represent the sign of the answer denoted as sign and finally one bit representing whether our ALU's answer is zero or not denoted as zero. In the design specification it is specified that the 12 bit input signal's (OPCODE) first four bit (MSB) will denote the type of operation the ALU system will perform. OPCODE's least significant four bits located from index 3 to 0 of OPCODE is denoted as register 'A'. OPCODE's bit 7 to 4, these four bits will denote the register 'B'. These mapping can be visualised from the table below:

	OPCODE											
	Code				**B**				**A**			
Index	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	x	x	x	x	B3	B2	B1	B0	A3	A2	A1	A0

We are group 8. Our design must include the following three arithmetic operations-addition, subtraction and multiplication. We also had to include the following five logical operations-nand,or,xor,not and nor. The code and their corresponding operation is shown in the table below:

Arithmetic operations		**Logical operations**	
Code (xxxx)	**Operation**	**Code (xxxx)**	**Operation**
1000	Addition	1010	NAND
0100	Subtraction	1110	OR
1001	Multiplication	0111	XOR
xxxx	None	0001	NOT
xxxx	None	1101	NOR

Software used:

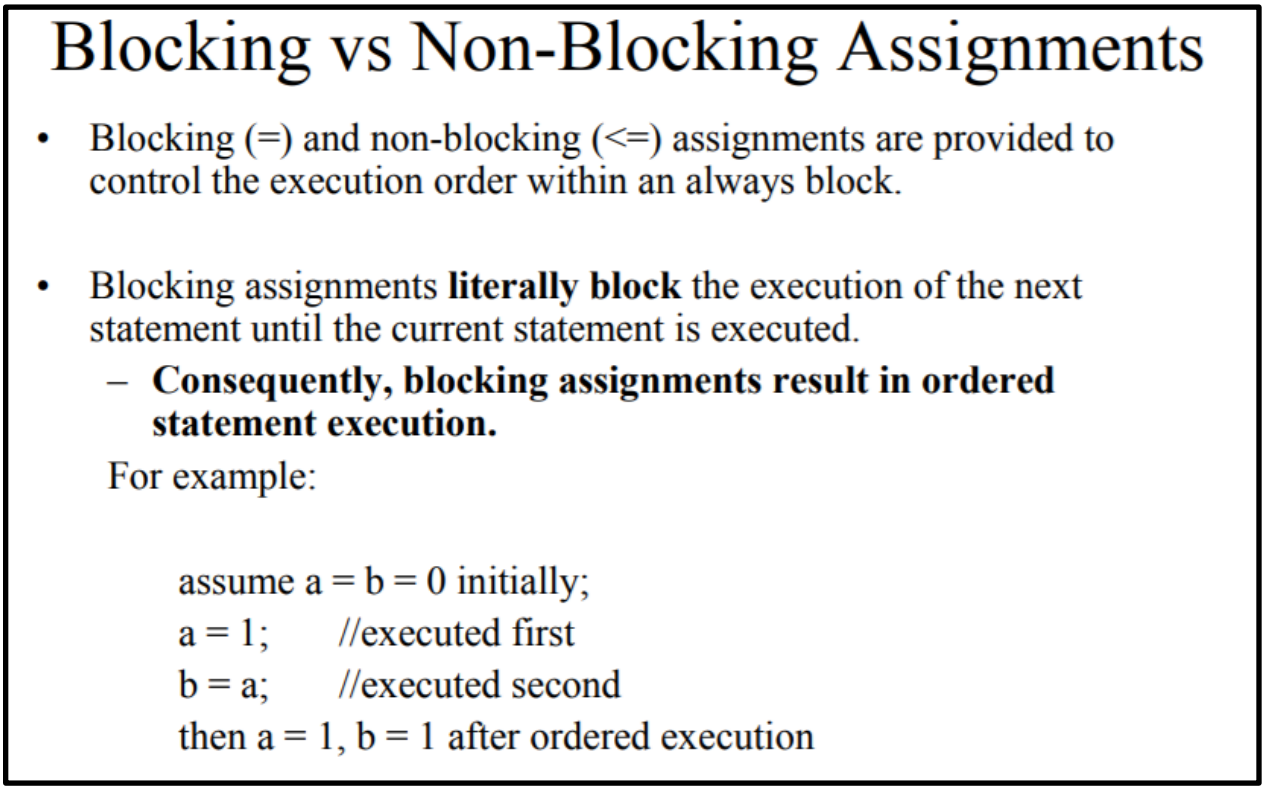
In this project we used Verilog HDL code to carry out the tasks. We have used 'Altera Quartus II' to write verilog code and simulate the system.

Version used: **Quartus II Version 8.1 build 163 10/28/2008 SJ Web Edition**

Working procedure:

Verilog HDL code can be written in two ways, structural and behavioral. First, we had to choose one of the approaches among these two. As we didn't have the requirement to generate a layout from the verilog code so we decided to write it in the behavioral style. Another influence behind our decision is that the behavioral representation of verilog is easier than structural. In our code we took an input of 12 bits and deconstructed the input according to our OPCODE table attached above. Then we declared a temporary register of 5 bits to store the result of the operations along

the carry bit. Using an always block we monitored the input and in the block we executed the steps sequentially. Here we wanted to execute the lines sequentially as we developed the logic in that way, so we used '=' instead of '<='. The main difference between them is '=' is a blocking statement meaning the next line won't execute until the current line is executed, whereas '<=' is a non-blocking statement, so multiple non-blocking statements will execute parallely. After execution of the right side of the statement, the left side's assignment is done at once. '=' working principle can be seen from the image below.



So, we used blocking statements. Then using a case block we checked the OPCODE's most significant four bits and according to those four bits we did various operations in our code. After the operation we set the temporary result's last four bit as output and then we checked if the result was zero or not. The carry is set by default as zero and in the addition operation we set the carry according to the output and the same goes for the sign bit except the sign is only set in subtraction operation. Finally, using the Quartus II's waveform generator we checked our system for various inputs and observed the resultant timing diagrams.

Verilog code:

```
module ALU_4bit(OUT,Carry,Sign,Zero,In);
output reg[3:0]OUT;
output reg Carry,Sign,Zero;
reg [4:0]result;
input [11:0]In;

reg [3:0]A,B,Code;

always @(In)
      begin
            Carry = 0;
            Sign = 0;
            A = In[3:0];
            B = In[7:4];
            Code = In[11:8];
            case (Code)
                  4'b1010: result = ~(A & B);
                  4'b1000:
                        begin
                        result = A + B;
                        Carry = result[4];
                        end
                  4'b1110: result = A | B;
                  4'b0100:
                        begin
                        if (B > A)
                              begin
                              result = B - A;
                              Sign = 1;
                              end
                        else
                              result = A - B;
                        end
                  4'b0111: result = A ^ B;
                  4'b0001: result = ~A;
                  4'b1001: result = A[1:0] * B[1:0];
                  4'b1101: result  = ~(A | B);
```

```
                default:
                      begin
                      Carry = 0;
                      Sign = 0;
                      result = 0;
                      end
            endcase
            OUT = result[3:0];
            if (result == 0)
                  Zero = 1;
            else
                  Zero = 0;
      end
endmodule
```

Timing diagram:

In our project we are provided with various OPCODE for various operations. Simulation for various opcodes are carried out and the resultant timing diagrams are attached below.

Arithmetic operations

OPCODE (Addition) : 1000 1001 1010

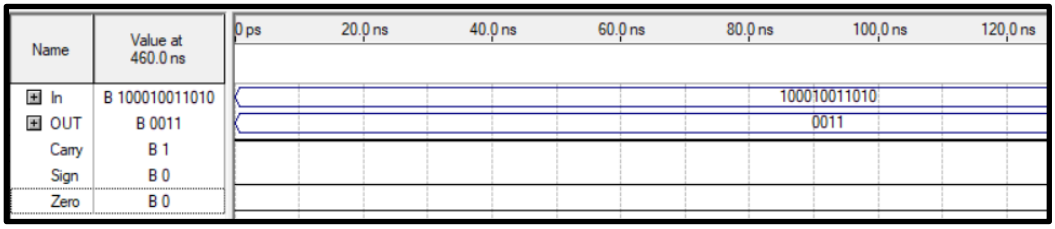


Figure1: OPCODE (1000 1001 1010). OUT= 0011, Carry=1, Sign=0 and Zero=0.

OPCODE (Subtraction) : 0010 1001 1010

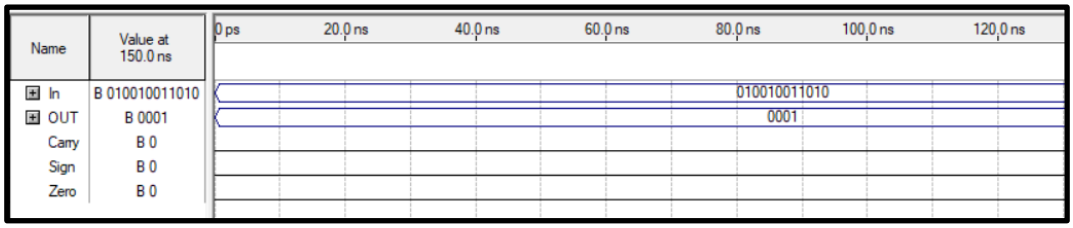


Figure2: OPCODE (0100 1001 1010). OUT= 0001, Carry=0, Sign=0 and Zero=0.

OPCODE (Subtraction and negative resultant) : 0010 1010 1001

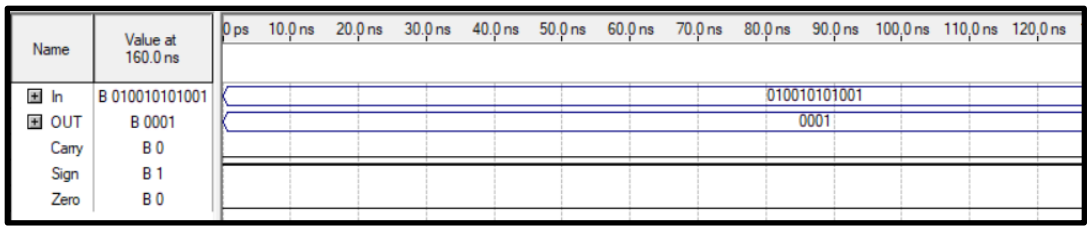


Figure3: OPCODE (0100 1010 1001). OUT= 0001, Carry=0, Sign=1 and Zero=0.

OPCODE (Multiplication) : 1001 1001 1010

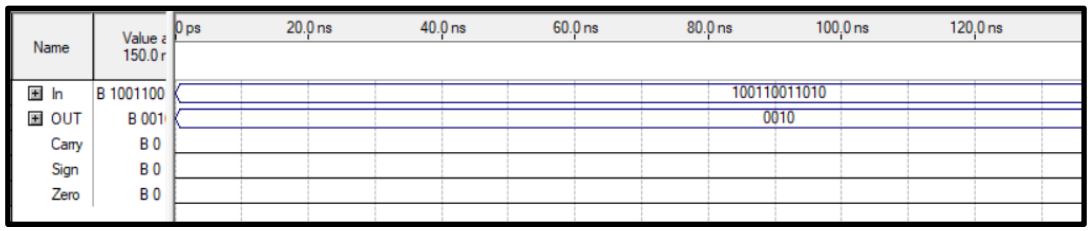


Figure4: OPCODE (1001 1001 1010). OUT= 0010, Carry=0, Sign=0 and Zero=0.

Logical operations

OPCODE (NAND) : 1010 1001 1010

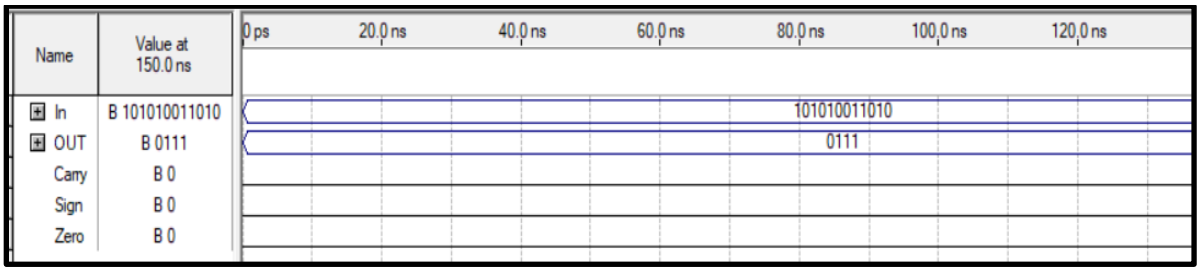


Figure5: OPCODE (1010 1001 1010). OUT= 0111, Carry=0, Sign=0 and Zero=0.

OPCODE (OR) : 1110 1001 1010

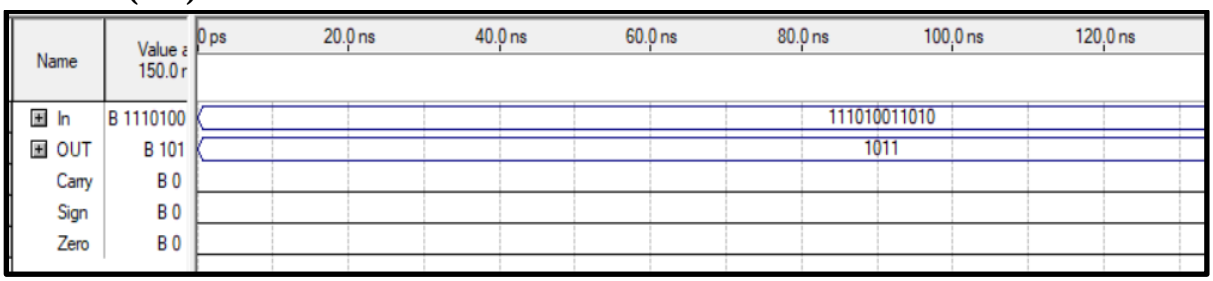


Figure6: OPCODE (1110 1001 1010). OUT= 1011, Carry=0, Sign=0 and Zero=0.

OPCODE (XOR) : 0111 1001 1010

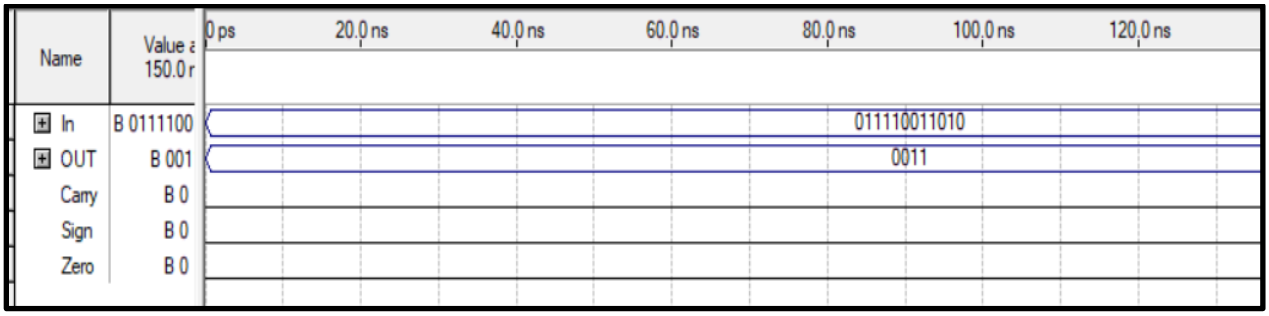


Figure7: OPCODE (0111 1001 1010). OUT= 0011, Carry=0, Sign=0 and Zero=0.

OPCODE (NOT) : 0001 1001 1010

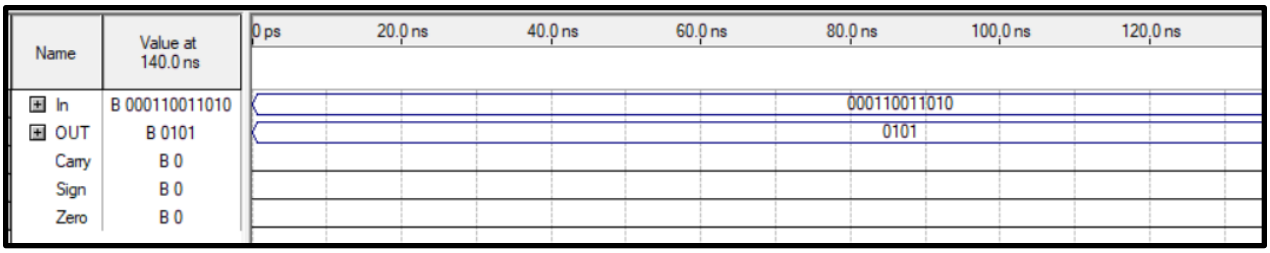


Figure8: OPCODE (0001 1001 1010). OUT= 0101, Carry=0, Sign=0 and Zero=0

OPCODE (NOR) : 1101 1001 1010

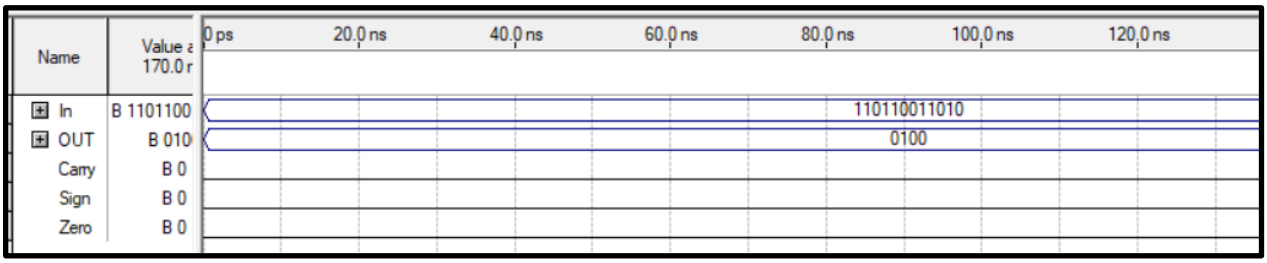


Figure9: OPCODE (1101 1001 1010). OUT= 0100, Carry=0, Sign=0 and Zero=0

Extra conditions

OPCODE (Subtraction result zero) : 0010 1001 1001

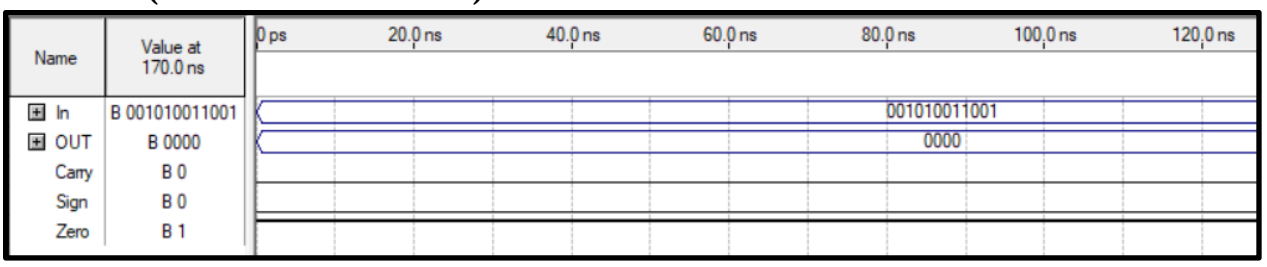


Figure10: OPCODE (0010 1001 1001). OUT= 0000, Carry=0, Sign=0 and Zero=1

OPCODE (Not defined) : 1111 1001 1001:

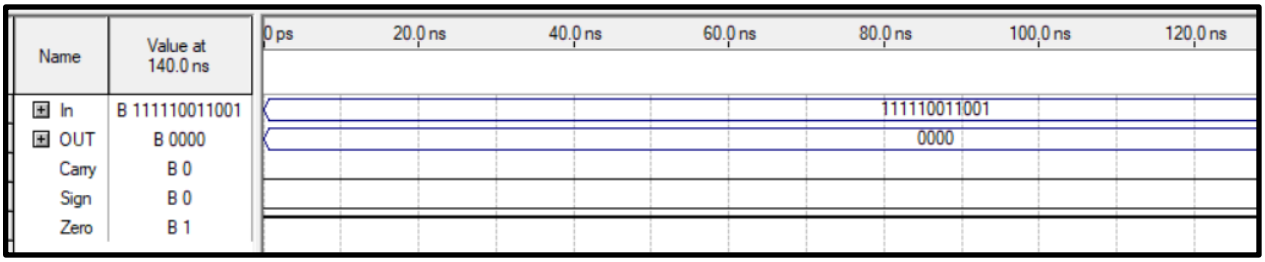


Figure10: OPCODE (1111 1001 1001). OUT= 0000, Carry=0, Sign=0 and Zero=1

Result and observations:

Explanation of code:

Initially we named the module as ALU_4bit. Then we have written the outputs (OUT, Carry, Sign, Zero) followed by the input (In) in the parenthesis. In the main code we have declared OUT as a 4 bit register. Carry, Sign and Zero is declared as one bit output. We have addition operations in our design. In addition, bit overflow might occur, so we declared a 5 bit register which will temporarily store the result. According to our design requirement we will give a 12 bit input so we declared In as a 12 bit input. In our design input signal's most significant four bits denote the type of operation and input's bit 7 to 4 is denoted as register B and bit 3 to 0 is denoted as register A. So, in our code we declared 3 four bit registers (Code, A, B). All these explained code can be seen from below.

```
module ALU_4bit(OUT,Carry,Sign,Zero,In);
output reg[3:0]OUT;
output reg Carry,Sign,Zero;
reg [4:0]result;
input [11:0]In;
reg [3:0]A,B,Code;
```

Then we used 'always @(In)' to monitor the input. In the code initially we set Sign, Carry as low. To deconstruct the 'In' signal in register A, B and Code we used array slicing of verilog. Then we used a 'case(Code)' block which operates various operations according to the value of Code.

According to design specification when Code is 1010 we need to perform NAND operation. This is done by doing bit wise AND (&) and then finally inverting (~) the output. Then we stored the output in a temporary result register. When Code is 1000 we need to perform addition operation. The result is stored in the result register. The MSB is then set as output of the Carry. When Code is 1110 we need to do bitwise OR and it is done using '|' operator. When Code is 0100 we need to perform subtraction. To do subtraction we checked whether B is greater than A or not. If B is larger than A then we set the Sign as 1 and subtract A from B and store the result in the result register. Otherwise we keep the Sign as default 0 and subtract B from A. When the Code is 0111 we perform bit wise XOR by (^) operator and store the result in the result register. When Code is 0001 we perform NOT operation using (~) and store the result in the result register. When Code is 1001 we performed multiplication of the last two bits of A and B. To do these we used array slicing and used (*) operator. When Code 1101 we had to perform NOR operation which is done by using bit wise or and then inverting it. Then the result is stored in the result register. Finally, we used default in case block which will handle any undefined Code. In that case result, Sign and Carry is set as 0. The explained code can be seen from below.

```
always @(In)
     begin
          Carry = 0;
          Sign = 0;
          A = In[3:0];
          B = In[7:4];
          Code = In[11:8];
          case (Code)
               4'b1010: result = ~(A & B);
               4'b1000:
                    begin
                    result = A + B;
                    Carry = result[4];
                    end
               4'b1110: result = A | B;
               4'b0100:
```

```
                    begin
                    if (B > A)
                         begin
                         result = B - A;
                         Sign = 1;
                         end
                    else
                         result = A - B;
                    end
               4'b0111: result = A ^ B;
               4'b0001: result = ~A;
               4'b1001: result = A[1:0] * B[1:0];
               4'b1101: result  = ~(A | B);
               default:
                    begin
                    Carry = 0;
                    Sign = 0;
                    result = 0;
                    end
          endcase
```

The temporary result is stored in the result register. But the output should be four bits. So, we set the OUT as result register's least significant four bits. Then we check if the result is zero or not. If the result register is 0 then we set the Zero as 1, otherwise 0. These codes are provided below.

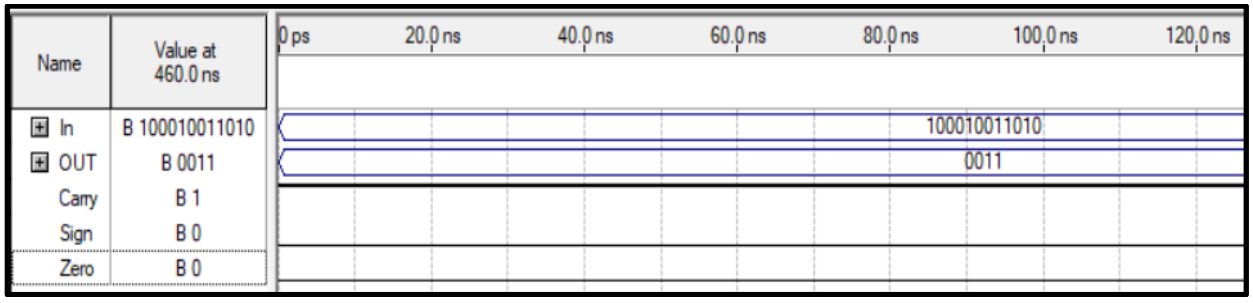
```
OUT = result[3:0];
          if (result == 0)
               Zero = 1;
          else
               Zero = 0;
```

The total explained code is already attached in the "Verilog code" section.

Validating timing diagram:

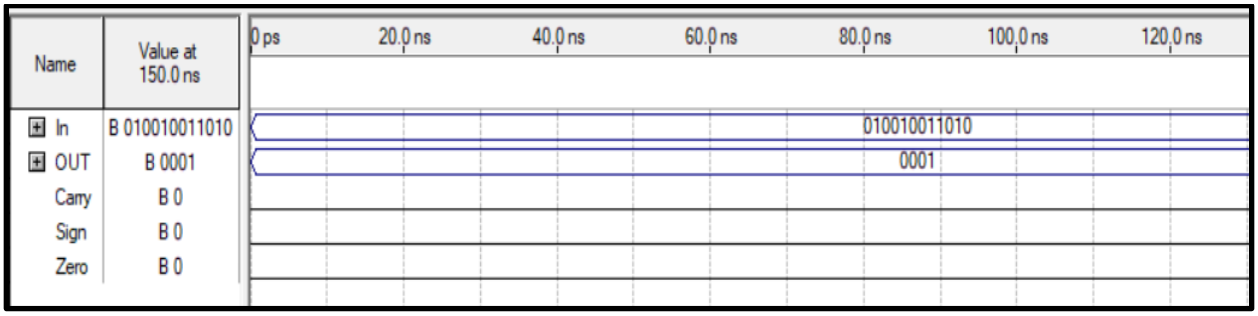
Here we will reference all the provided timing diagrams from the 'Timing diagram' section.

Addition:

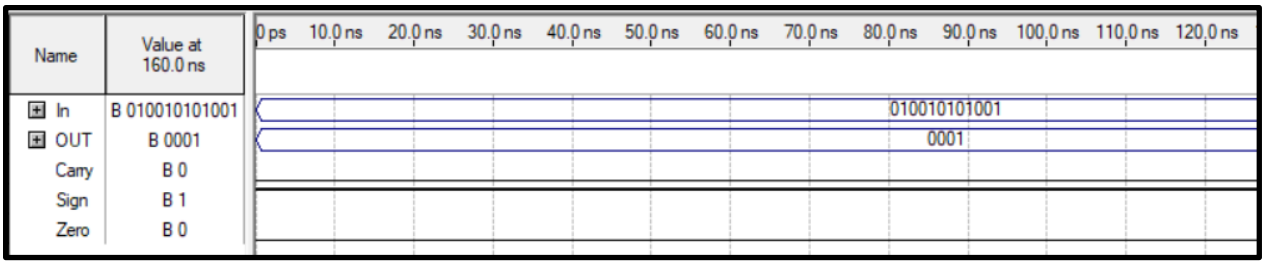


In this timing diagram the OPCODE is '**1000 1001 1010'.** Here Code=1000, A=1010 and B=1001. So we have to perform addition and the result of the addition in binary should be (1010+1001)=10011. As the output is four bits so in the timing diagram we can see OUT as 0011. The MSB is carry, thus we are seeing Carry as 1 in the timing diagram. As the result is positive so Sign is set as 0 and as the result is not 0 so Zero is also set as 0.

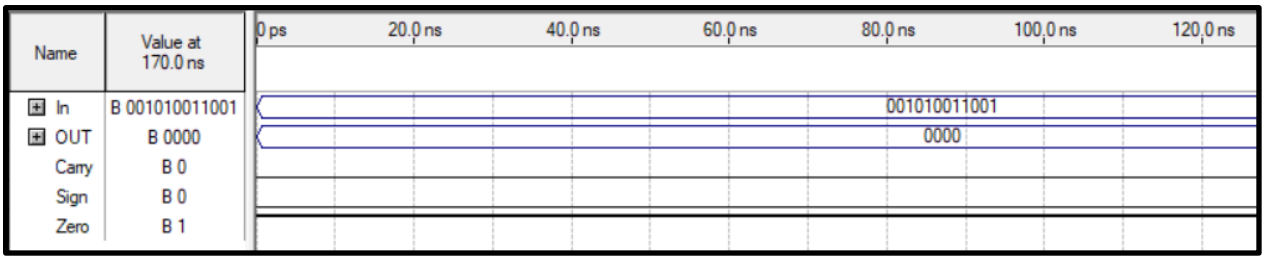
Subtraction:



In this timing diagram the OPCODE is '**0100 1001 1010'.** Here Code=0100, A=1010 and B=1001. So we have to perform subtraction and the result of the subtraction in binary should be (1010-1001)=0001. As the output is four bits so in the timing diagram we can see OUT as 0001. There is no carry, thus we are seeing Carry as 0 in the timing diagram. As the result is positive so Sign is set as 0 and as the result is not 0 so Zero is also set as 0.

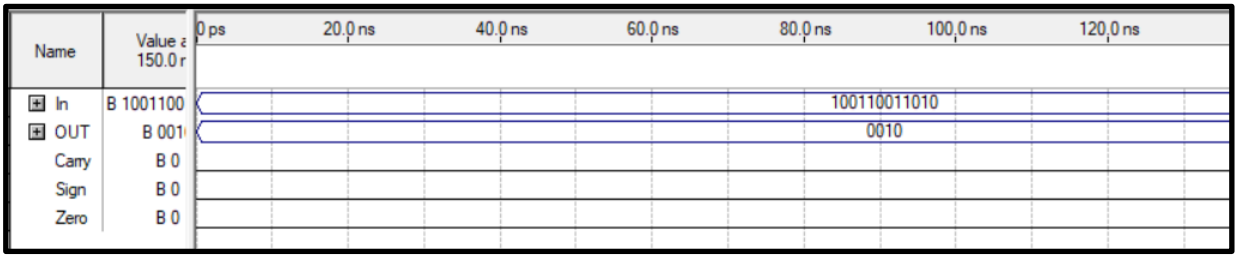


In this timing diagram the OPCODE is '**0100 1010 1001'.** Here Code=0100, A=1001 and B=1010. So we have to perform subtraction and the result of the subtraction in binary should be (1010-1001)=0001 but the result is negative. As the output is four bits so in the timing diagram we can see OUT as 0001. There is no carry, thus we are seeing Carry as 0 in the timing diagram. As the result is negative so Sign is set as 1 and as the result is not 0 so Zero is also set as 0.



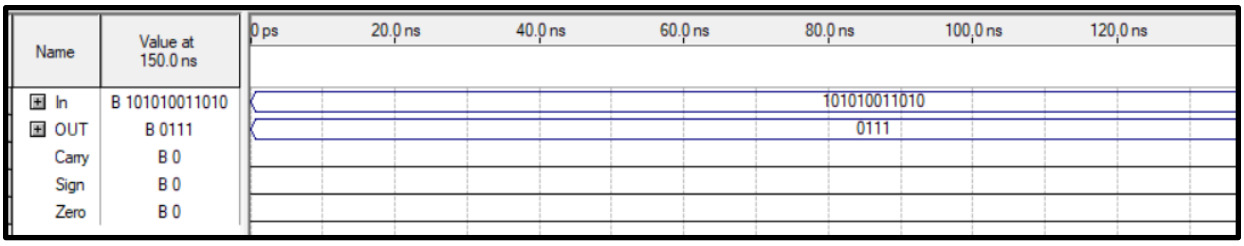
In this timing diagram the OPCODE is '**0010 1001 1001'.** Here Code=0010, A=1001 and B=1001. So we have to perform subtraction and the result of the subtraction in binary should be (1001-1001)=0000. As the output is four bits so in the timing diagram we can see OUT as 0000. There is no carry, thus we are seeing Carry as 0 in the timing diagram. As the result is not negative so Sign is set as 0 and as the result is 0 so Zero is set as 1

Multiplication:



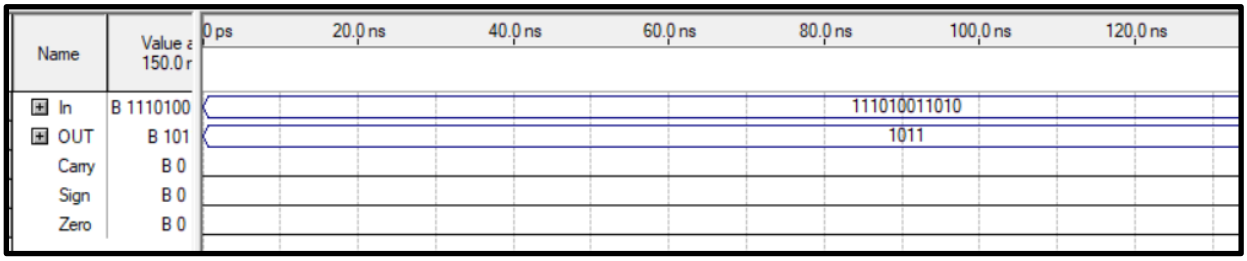
In this timing diagram the OPCODE is '**1001 1001 1010'.** Here Code=1001, A=1010 and B=1001. We have to perform multiplication of the last two bits of A and B and the result of the multiplication in binary should be (10*01)=010. As the output is four bits, in the timing diagram we can see OUT as 0010. There is no carry, thus we are seeing Carry as 0 in the timing diagram. As the result is positive so Sign is set as 0 and as the result is not 0 so Zero is also set as 0.

NAND:



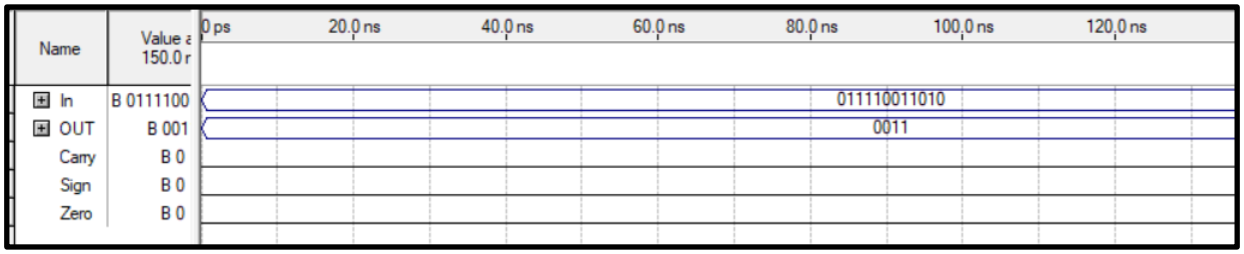
In this timing diagram the OPCODE is '**1010 1001 1010'.** Here Code=1010, A=1010 and B=1001. So, we have to perform NAND. Output of NAND is high unless both the inputs are 1. As the output is four bits so in the timing diagram we can see OUT as 0111. There is no carry, thus we are seeing Carry as 0 in the timing diagram. As the result is positive so Sign is set as 0 and as the result is not 0 so Zero is also set as 0. So, the NAND operation is successfully performed.

OR:



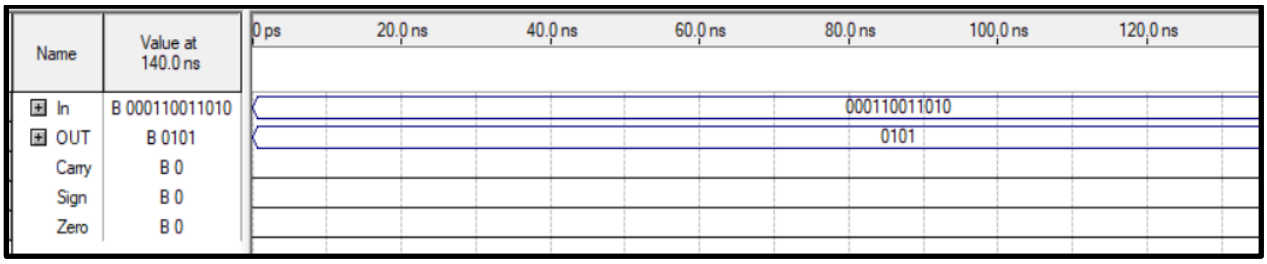
In this timing diagram the OPCODE is '**1110 1001 1010'.** Here Code=1110, A=1010 and B=1001. So, we have to perform OR. Output of OR is high if any of the input is 1. As the output is four bits so in the timing diagram we can see OUT as 1011. There is no carry, thus we are seeing Carry as 0 in the timing diagram. As the result is positive so Sign is set as 0 and as the result is not 0 so Zero is also set as 0. So, the OR operation is successfully performed.

XOR:



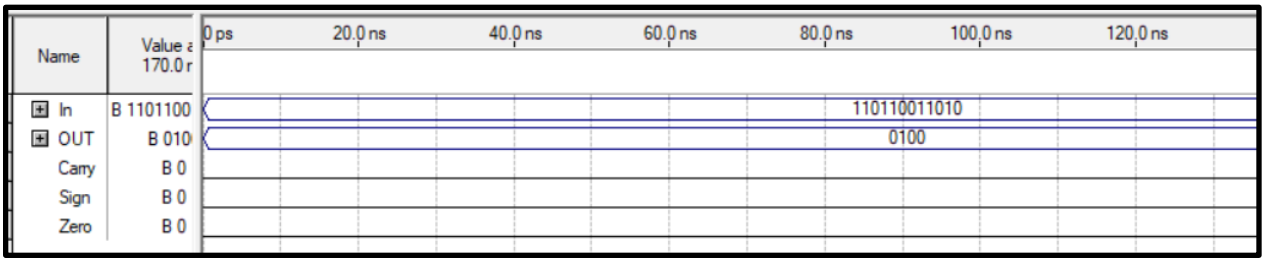
In this timing diagram the OPCODE is '**0111 1001 1010'.** Here Code=0111, A=1010 and B=1001. So, we have to perform XOR. Output of XOR is high if both the inputs are not the same. As the output is four bits so in the timing diagram we can see OUT as 0011. There is no carry, thus we are seeing Carry as 0 in the timing diagram. As the result is positive so Sign is set as 0 and as the result is not 0 so Zero is also set as 0. So, the XOR operation is successfully performed.

NOT:



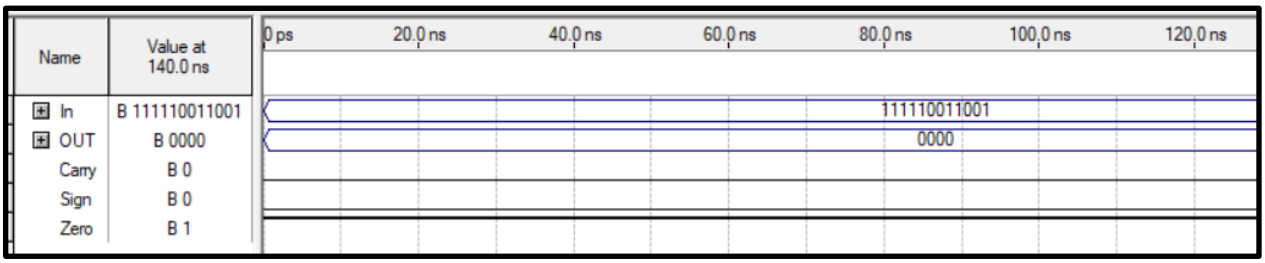
In this timing diagram the OPCODE is '**0001 1001 1010'.** Here Code=0111, A=1010. So, we have to perform NOT. As the output is four bits so in the timing diagram we can see OUT as ~A=0101. There is no carry, thus we are seeing Carry as 0 in the timing diagram. As the result is positive so Sign is set as 0 and as the result is not 0 so Zero is also set as 0. So, the NOT operation is successfully performed.

NOR:



In this timing diagram the OPCODE is '**1101 1001 1010'.** Here Code=1101, A=1010 and B=1001. So, we have to perform NOR. NOR's output is 1 only when both the inputs are 0. As the output is four bits so in the timing diagram we can see OUT as 0100. There is no carry, thus we are seeing Carry as 0 in the timing diagram. As the result is positive so Sign is set as 0 and as the result is not 0 so Zero is also set as 0. So, the NOR operation is successfully performed.

Undefined:

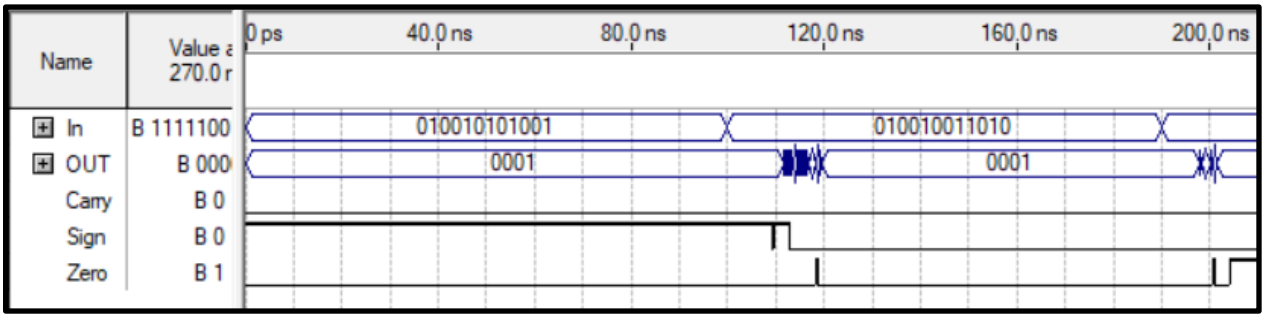


In this timing diagram the OPCODE is '**1111 1001 1010'.** Here Code=1111, A=1001 and B=1001. As no operation is defined for Code=1111, so the OUT is set as 0000. The Carry and Sign is also set as 0. As the OUT is 0000 so the Zero is set as 1.

Discussion:

In this project we designed a 4 bit ALU. Throughout the process we learned to develop the algorithm behind the specified project problem. Initially we planned to develop the system using a structural way of coding. But after a little bit of progress we found out that it is difficult to design the whole system using structural method as we need to design a few mux and many full adders, which contain a large number of logic gates. So, we went with the behavioral design of the system instead of structural. From this we understood, for simulation purposes it is better to write verilog in a behavioral way. During the development process of the system we developed the algorithm keeping in mind the ordered execution. In order to achieve this we used blocking statements here by using '='. For the ease of calculation we wrote the code such that the result is stored in a temporary variable and then set the last four bits as output. We found that for addition operation,

bit overflow might occur so a temporary variable of 5 bit is used to store the result. While trying to perform subtraction, we noticed that if the result was a negative number, an erroneous result was given. We solved the issue by checking which number is larger and subtracting the smaller number from the larger one. If the result is supposed to be negative, we set the sign as 1. Finally, to decide whether the result is zero we used an if else block and based on it set the value of zero. Further we tried to do multiple operations in the simulation. The result is shown below.



Here OPCODE is "**0100 1010 1001**" in the first 100ns. As the Code is 0100 so the system will perform subtraction. The result of the subtraction is (1001-1010)=0001 but the result will be negative, therefore the Sign is 1. From 100ns to 190ns the OPCODE is "**0100 1001 1010**", therefore subtraction will be performed and the result will be positive 1. This is what we have observed from the timing diagram. We have also observed a few glitches in the system while transitioning from one input to another. We could not solve this problem as there is a small undefined state when all the bit's result is being calculated. Apart from this, our verilog code generated desired output throughout the process for all the preset inputs.

Reference:

- **Blocking vs Non-Blocking Assignments:**

  http://www.eecs.umich.edu/courses/eecs270/270lab/270_docs/Advanced_Verilog.pdf

Peer evaluation (On a scale of 10):

Evaluated by ⇒ ⇓ Peer being evaluated	Sk Tahmed Salim Rafid	Asef Jamil Ajwad	Pronoyan Saha
Sk Tahmed Salim Rafid (19121028)	X	10	10
Asef Jamil Ajwad (19121040)	10	X	10
Pronoyan Saha (19121131)	10	10	X