MARMARA UNIVERSITY – FACULTY OF ENGINEERING

LABORATORY ASSIGNMENT #1 REPORT

Student Name, Surname: Hasan Şenyurt – Yusuf Akbulut

Student ID: 150120531 - 150118023

Department: Computer Engineering

TABLE OF CONTENTS

- ALU DESIGN

- COMPONENTS FOR OPERATIONS
  - 8-bit Sixteen to One Multiplexer
  - 8-bit Adder
  - 8-bit Subtractor
  - 8-bit And Operator
  - 8-bit Or Operator
  - 8-bit Move Operator
  - 8-bit Shift Left Operator
  - 8-bit Shift Right Operator
  - 8-bit Incrementor
  - 8-bit Decrementor

- FLAGS
  - Carry Flag
  - Zero Flag
  - Negative Flag
  - Overflow Flag

ALU DESIGN

ALU has 9 operations. Names and operation codes of operations are listed below:

- ADDA : 0000
- SUBZ : 0001
- ANDA : 0010
- ORA : 0011
- MOVA : 0100
- SHFTLA : 0101
- SHFTRA : 0110
- INCR : 0111
- DECR : 1000

8-bit 16x1 multiplexer will be used to choose one of these operations. ALU takes 3 inputs which are 8-bit value A, 8-bit value B and 4-bit opcode value. ALU gives 5 outputs which are 8-bit result of the chosen operation, carry flag bit, zero flag bit, negative flag bit and overflow flag bit.

ADD, SUB, INCR and DECR operations send flag results to 2 16x1 multiplexer. One of the multiplexers is for carry flag, another multiplexer is for overflow flag. Flag results is put to multiplexer according to its operation code. Carry flag result of ADD is put to 0000 input of 16x1 multiplexer, carry flag result of INCR is put to 0111 input of 16x1 multiplexer.

The empty inputs are set to 0 in all multiplexers. They will not be used in this ALU. Negative and zero flag is set after choosing operation in the 8-bit 16x1 multiplexers. Result of the operation is checked for zero flag and negative flag.

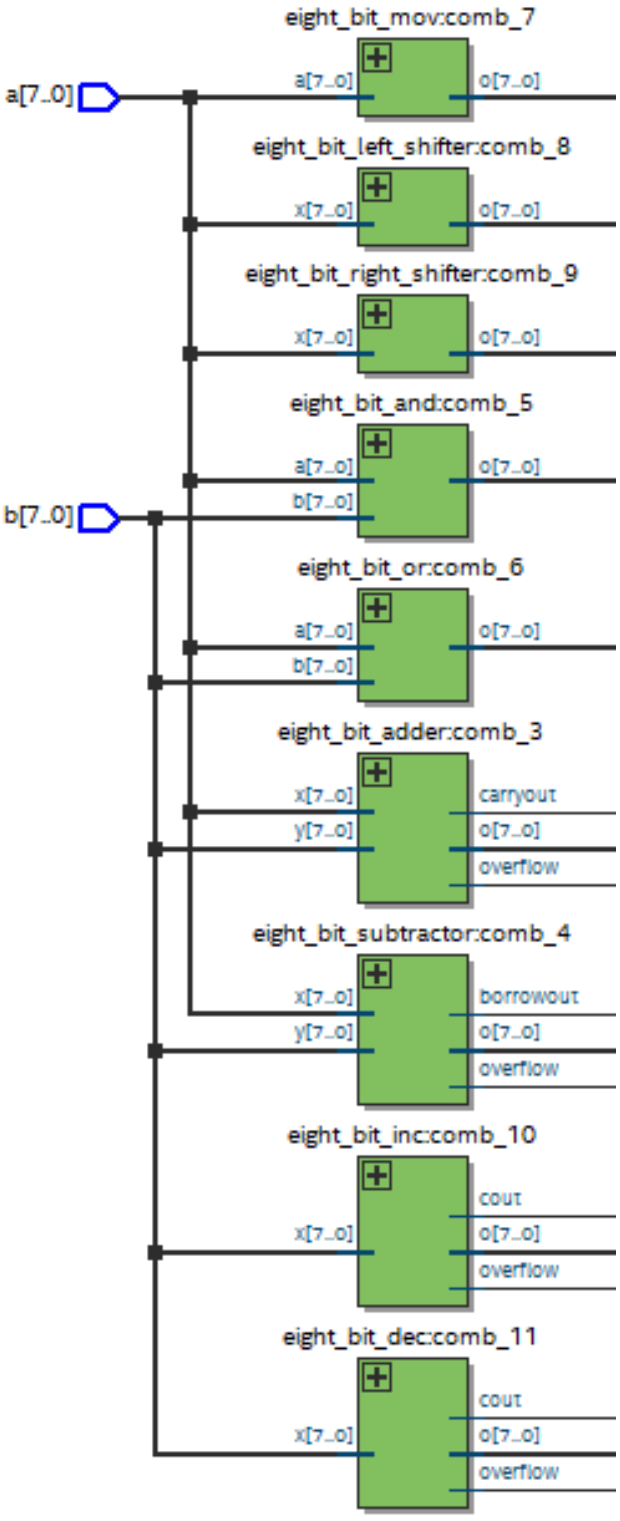

Figure-1: 9 operation blocks

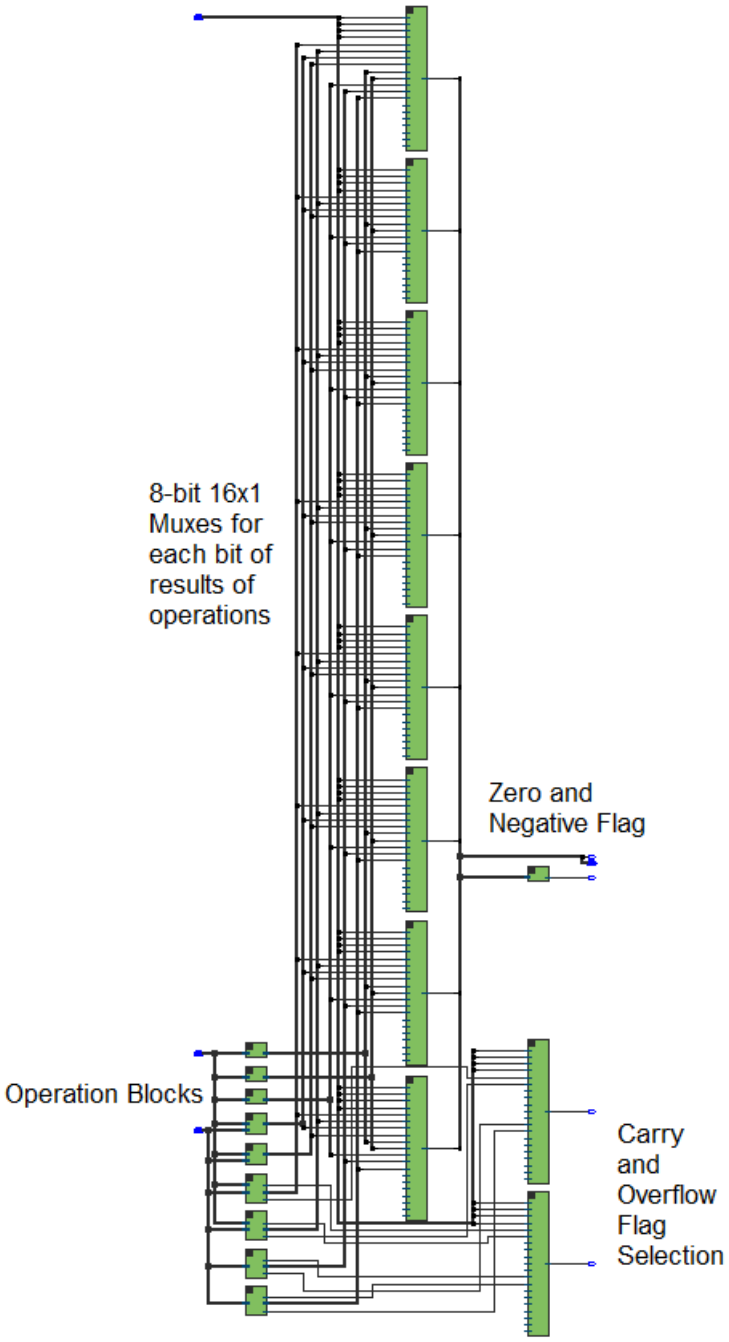

Figure-2: General design of ALU

```
module eight_bit_alu(

    input [3:0] opcode,
    input [7:0] a,
    input [7:0] b,

    output [7:0] o,
    output c,
    output z,
    output n,
    output ov
);
   //Carry and overflow flags
   wire c_add,c_sub,c_inc,c_dec,ov_add,ov_sub,ov_inc,ov_dec;

   wire [7:0]addA;
   wire [7:0]subA;
   wire [7:0]andA;
   wire [7:0]orA;
   wire [7:0]movA;
   wire [7:0]shftlA;
   wire [7:0]shftrA;
   wire [7:0]incr;
   wire [7:0]decr;

   //OPERATIONS
   eight_bit_adder(a,b,addA,c_add,ov_add);
   eight_bit_subtractor(a,b,subA,c_sub,ov_sub);
   eight_bit_and(a,b,andA);
   eight_bit_or(a,b,orA);
   eight_bit_mov(a,movA);
   eight_bit_left_shifter(a,shftlA);
   eight_bit_right_shifter(a,shftrA);
   eight_bit_inc(b,incr,c_inc,ov_inc);
   eight_bit_dec(b,decr,c_dec,ov_dec);


   //MUXES FOR ALL OPERATIONS
   sixteen_to_one_multiplexer MUX0(opcode[3],opcode[2],opcode[1],opcode[0],0,0,0,0,0,0,0,0,
   decr[0],incr[0],shftrA[0],shftlA[0],movA[0],orA[0],andA[0],subA[0],addA[0],o[0]);
   sixteen_to_one_multiplexer MUX1(opcode[3],opcode[2],opcode[1],opcode[0],0,0,0,0,0,0,0,0,
   decr[1],incr[1],shftrA[1],shftlA[1],movA[1],orA[1],andA[1],subA[1],addA[1],o[1]);
   sixteen_to_one_multiplexer MUX2(opcode[3],opcode[2],opcode[1],opcode[0],0,0,0,0,0,0,0,0,
   decr[2],incr[2],shftrA[2],shftlA[2],movA[2],orA[2],andA[2],subA[2],addA[2],o[2]);
   sixteen_to_one_multiplexer MUX3(opcode[3],opcode[2],opcode[1],opcode[0],0,0,0,0,0,0,0,0,
   decr[3],incr[3],shftrA[3],shftlA[3],movA[3],orA[3],andA[3],subA[3],addA[3],o[3]);
   sixteen_to_one_multiplexer MUX4(opcode[3],opcode[2],opcode[1],opcode[0],0,0,0,0,0,0,0,0,
   decr[4],incr[4],shftrA[4],shftlA[4],movA[4],orA[4],andA[4],subA[4],addA[4],o[4]);
   sixteen_to_one_multiplexer MUX5(opcode[3],opcode[2],opcode[1],opcode[0],0,0,0,0,0,0,0,0,
   decr[5],incr[5],shftrA[5],shftlA[5],movA[5],orA[5],andA[5],subA[5],addA[5],o[5]);
   sixteen_to_one_multiplexer MUX6(opcode[3],opcode[2],opcode[1],opcode[0],0,0,0,0,0,0,0,0,
   decr[6],incr[6],shftrA[6],shftlA[6],movA[6],orA[6],andA[6],subA[6],addA[6],o[6]);
   sixteen_to_one_multiplexer MUX7(opcode[3],opcode[2],opcode[1],opcode[0],0,0,0,0,0,0,0,0,
   decr[7],incr[7],shftrA[7],shftlA[7],movA[7],orA[7],andA[7],subA[7],addA[7],o[7]);

   //Carry flag muxes
   sixteen_to_one_multiplexer MUX8(opcode[3],opcode[2],opcode[1],opcode[0],0,0,0,0,0,0,0,0,
   c_dec,c_inc,0,0,0,0,c_sub,c_add,c);

   //Overflow flag muxes
   sixteen_to_one_multiplexer MUX9(opcode[3],opcode[2],opcode[1],opcode[0],1'b0,0,0,0,0,0,0,0,
   ov_dec,ov_inc,0,0,0,0,ov_sub,ov_add,ov);

   //Zero and Negative Flags
   zero_flag(o,z);
   xnor(n,o[7],1);

endmodule
```

Figure-3: Verilog code of ALU

COMPONENTS FOR OPERATIONS

8-bit Sixteen to One Multiplexer

8-bit 16x1 multiplexer is used to generate result according to operation code. The number of operations of 8-bit ALU is nine. So, 9 pins will be used, other 7 will not be used. 1-bit Sixteen to One Multiplexer can be built by using 2x1, 4x1 and 8x1 multiplexers. The structure of 16x1 multiplexer is below:

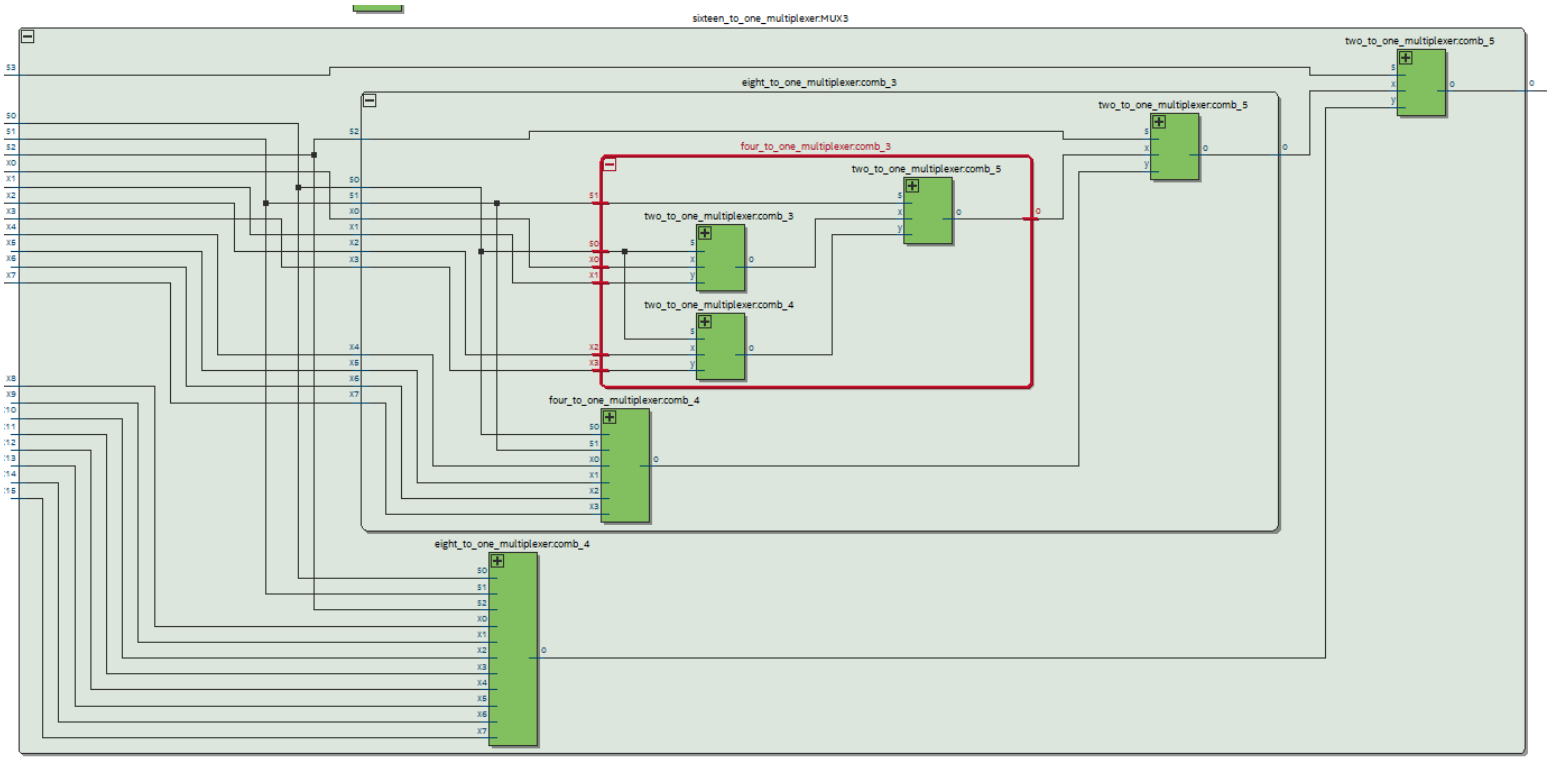

Figure-1: Inside of 16x1 Multiplexer

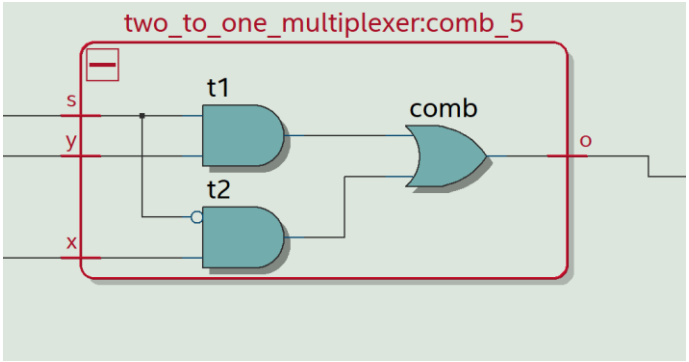

Figure-2: Inside of 2x1 Multiplexer

4x1 multiplexer is built with 3 2x1 multiplexers. 8x1 multiplexer is built with 2 4x1 multiplexers and 2x1 multiplexer. 16x1 multiplexer is built with 2 8x1 multiplexers and 2x1 multiplexer. Main idea of this structure is choosing one of the results with 2x1 every time at the end of the operation (Figure-1).

In 2x1 multiplexer (Figure-2), there are two AND gates with x, y and s as inputs. s input is for selection. s input and not s input goes into and gates to select one of the inputs x and y.

The Verilog code for 1-bit 16x1 multiplexer is below (Figure-3):

```
module sixteen_to_one_multiplexer(
    input s3,s2,s1,s0,
    input x15,x14,x13,x12,x11,x10,x9,x8,x7,x6,x5,x4,x3,x2,x1,x0,
    output o
);
   wire t1,t2;

  eight_to_one_multiplexer(s2,s1,s0,x7,x6,x5,x4,x3,x2,x1,x0,t1);
  eight_to_one_multiplexer(s2,s1,s0,x15,x14,x13,x12,x11,x10,x9,x8,t2);
  two_to_one_multiplexer(s3, t1, t2, o);

endmodule

module eight_to_one_multiplexer(
    input s2,s1,s0,
    input x7,x6,x5,x4,x3,x2,x1,x0,
    output o

);
   wire t1,t2;

  four_to_one_multiplexer(s1,s0,x3,x2,x1,x0,t1);
  four_to_one_multiplexer(s1,s0,x7,x6,x5,x4, t2);
  two_to_one_multiplexer(s2, t1, t2, o);

endmodule

module four_to_one_multiplexer( //Four to one
 input s1,s0,
 input x3,x2,x1,x0,
 output o
);

 wire t1, t2;
 two_to_one_multiplexer(s0, x0, x1, t1);
 two_to_one_multiplexer(s0, x2, x3, t2);
 two_to_one_multiplexer(s1, t1, t2, o);
endmodule


module two_to_one_multiplexer(
 input s,
 input x,
 input y,
 output o
);
 wire t1, t2;
 and(t1, s, y);
 and(t2, ~s, x);
 or(o, t1, t2);
endmodule
```

Figure-3: Verilog Code of 16x1 Multiplexer

8-bit Adder

8-bit adder is used to add two 8-bit values and generates three outputs, which are the result of the sum, carryout bit (flag) and overflow bit (flag). 8-bit adder is built with 8 1-bit adder.

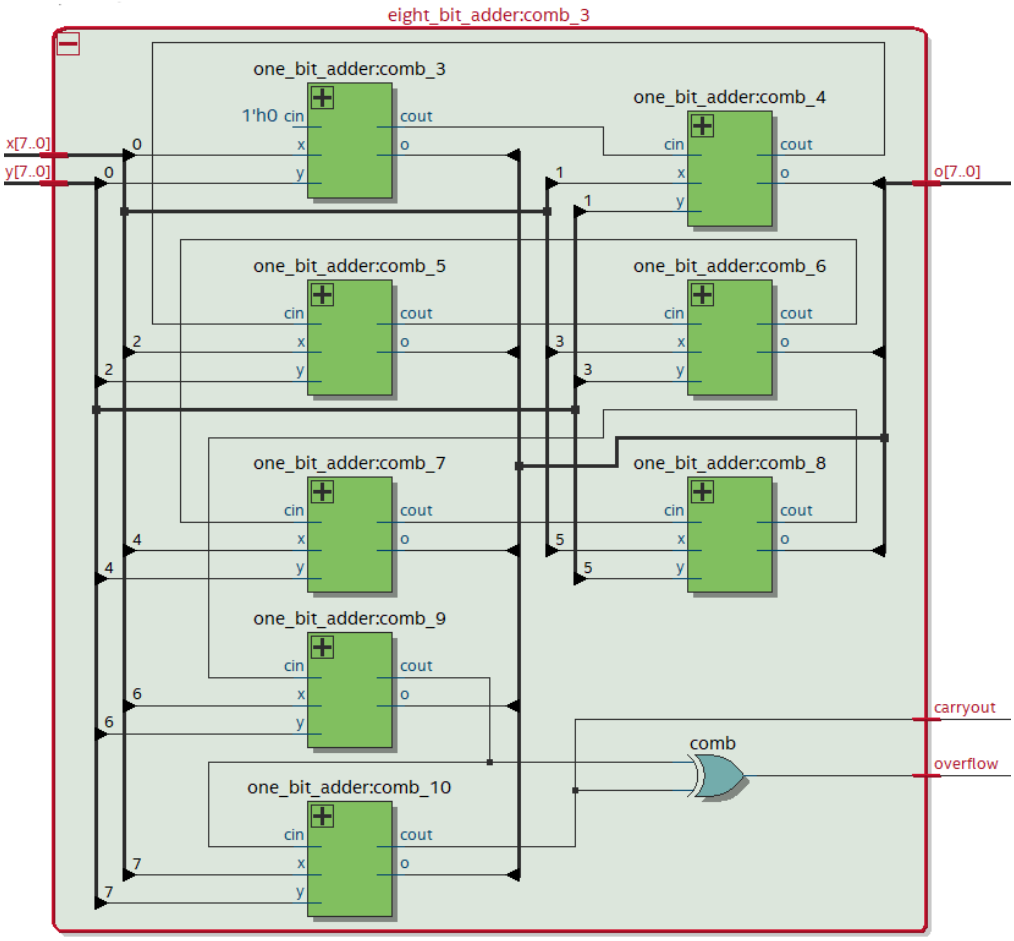

Figure-1: Inside of 8-bit adder.

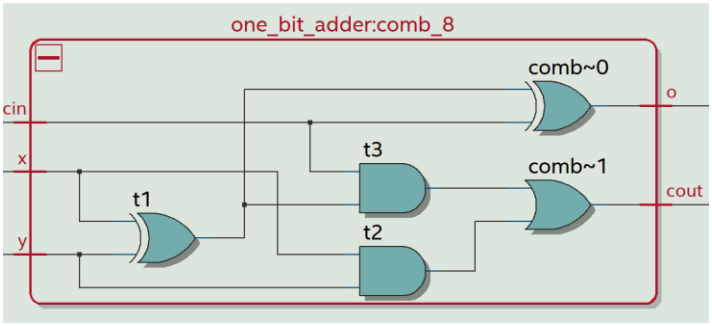

Figure-2: Inside of 1-bit adder.

1-bit adder takes 3 inputs which are carry in bit, x and y. It adds two input bits and gives result and carry out bit as outputs. The 1-bit adder after previous one takes carry out of previous one as carry in input. This structure is used for 8 times to build 8-bit adder.

Overflow flag is controlled by taking carry in and carry out bits of most significant bit (eighth bit) of the result to XOR gate. If overflow occurs, then overflow bit will be 1.

Carry flag is controlled by taking the value of carry out bit. If carry out bit is 1, then carry flag will be set to 1.

The Verilog code of 8-bit adder is below (Figure-3):

```
module eight_bit_adder(
    input [7:0] x,
    input [7:0] y,
    output [7:0] o,
    output carryout,
    output overflow
);
    wire cout1, cout2, cout3, cout4, cout5, cout6, cout7;
    one_bit_adder(x[0], y[0], 1'b0, o[0], cout1);
    one_bit_adder(x[1], y[1], cout1, o[1], cout2);
    one_bit_adder(x[2], y[2], cout2, o[2], cout3);
    one_bit_adder(x[3], y[3], cout3, o[3], cout4);
    one_bit_adder(x[4], y[4], cout4, o[4], cout5);
    one_bit_adder(x[5], y[5], cout5, o[5], cout6);
    one_bit_adder(x[6], y[6], cout6, o[6], cout7);
    one_bit_adder(x[7], y[7], cout7, o[7], carryout);

    xor(overflow,cout7,carryout);


endmodule

module one_bit_adder(
    input x,
    input y,
    input cin,
    output o,
    output cout
);
    wire t1, t2, t3;
    xor(t1, x, y);
    xor(o, t1, cin);
    and(t2, x, y);
    and(t3, t1, cin);
    or(cout, t2, t3);
endmodule
```

Figure-3: Verilog code of 8-bit adder.

Opcode of add operation is 0000 in binary. The simulation of add operation is below:

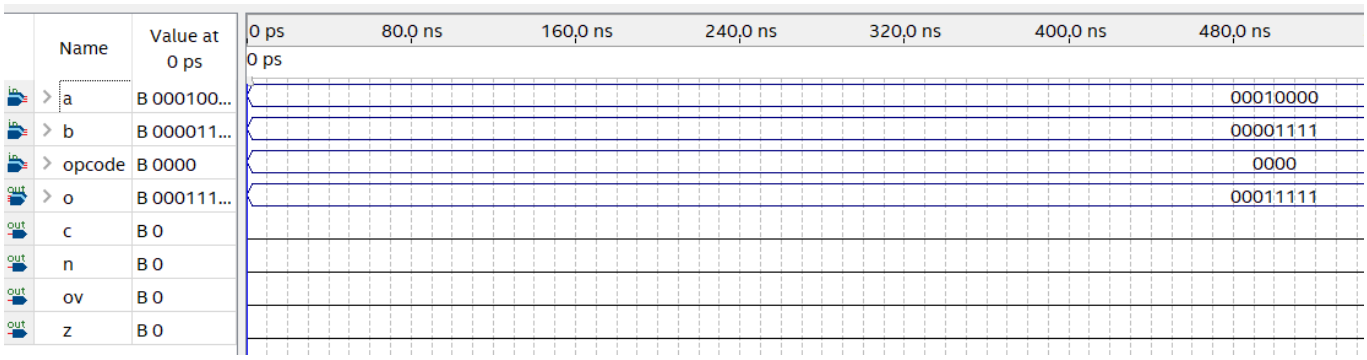

Figure-4: Simulation of 8-bit adder.

8-bit Subtractor

8-bit subtractor is used to subtract one 8-bit value from another 8-bit value and generates three outputs, which are the result of the subtraction, carryout bit (flag) and overflow bit (flag). 8-bit subtractor is built with 2 8-bit adder. One of the inputs of the 8-bit adder will be two's complement of 8-bit value. First value and the value which is two's complement will be added. This addition is same as subtraction.

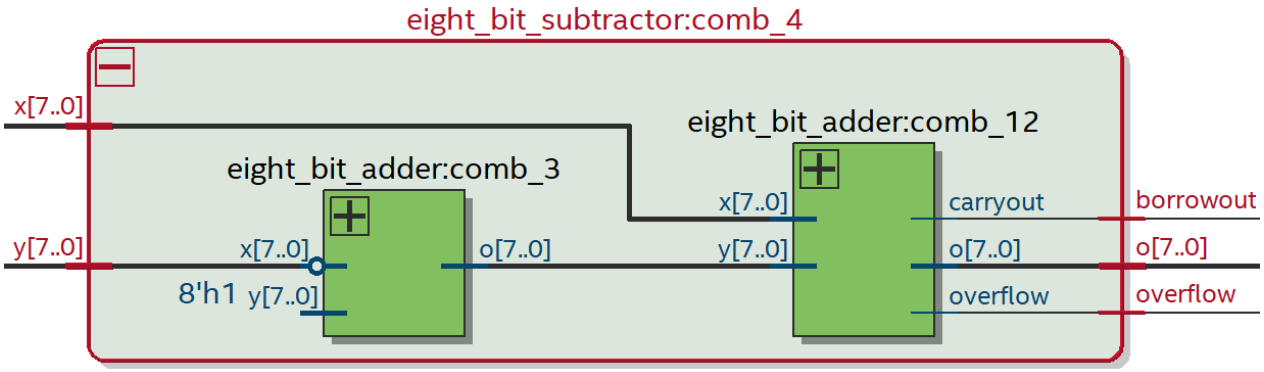

Figure-1: Inside of 8-bit Subtractor

First 8-bit adder adds negation of second input and one to generate two's complement. Second 8-bit adder adds first input and two's complement of second input to get subtraction result. Overflow and carry flags are controlled by 8-bit adder as same of add operation.

The Verilog code of 8-bit subtractor is below (Figure-2):

```
module eight_bit_subtractor(
    input [7:0] x,
    input [7:0] y,
    output [7:0] o,
    output borrowout,
    output overflow
);

    wire [7:0]t1,bout,ov;

    eight_bit_adder(~y,1,t1,bout,ov);
    eight_bit_adder(x,t1,o,borrowout,overflow);

endmodule
```

Figure-2: Verilog code of 8-bit Subtractor

Opcode of 8-bit subtractor is 0001 in binary. The simulation of sub operation is below:

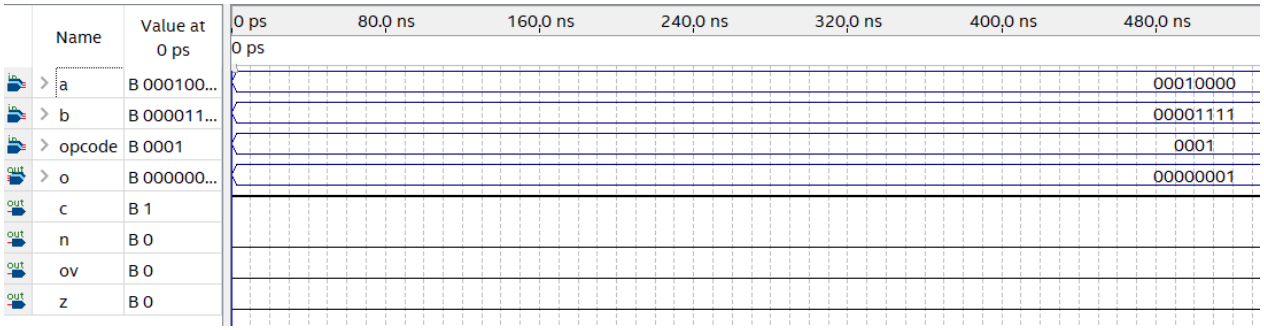

Figure-3: Simulation of sub operation

8-bit And Operator

8-bit and operator is used to make bitwise and operations for 8-bit values. It takes two inputs and gives result as output. 8 AND gates are used for every bit of two inputs.

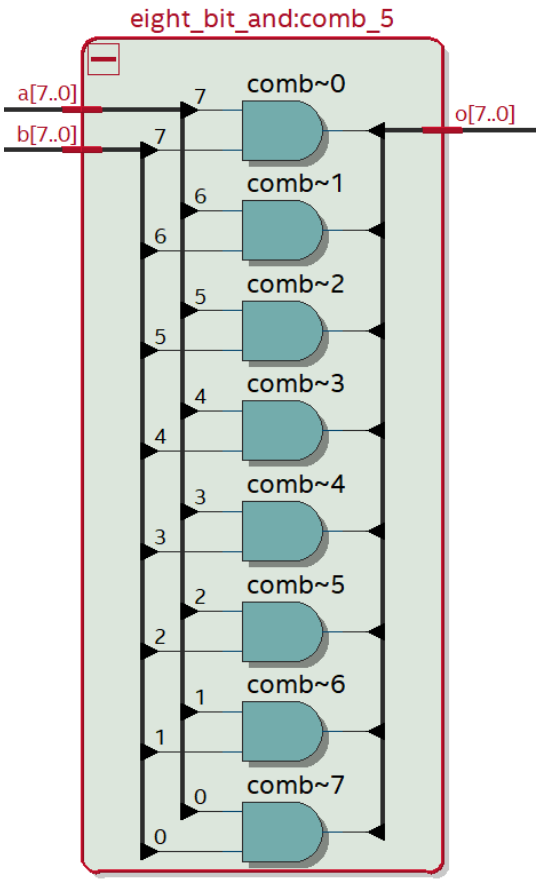

Figure-1: Inside of 8-bit and operator

The Verilog code of 8-bit and operator is below:

```
module eight_bit_and(

    input [7:0] a,
    input [7:0] b,
    output [7:0] o

);
    and(o[7],a[7],b[7]);
    and(o[6],a[6],b[6]);
    and(o[5],a[5],b[5]);
    and(o[4],a[4],b[4]);
    and(o[3],a[3],b[3]);
    and(o[2],a[2],b[2]);
    and(o[1],a[1],b[1]);
    and(o[0],a[0],b[0]);


endmodule
```

Figure-2: Verilog code of 8-bit and operator

Opcode of and operation is 0010 in binary. The simulation of and operation is below:

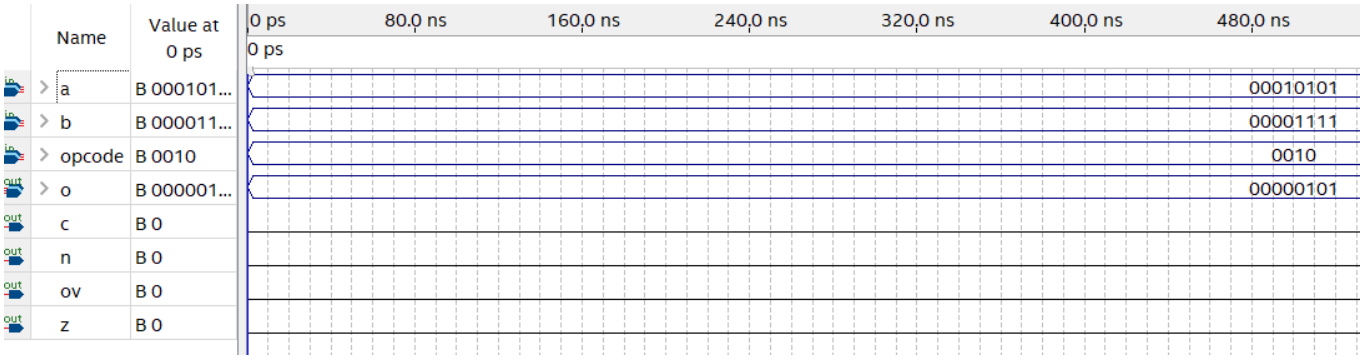

Figure-3: Simulation of and operation

8-bit Or Operator

8-bit and operator is used to make bitwise or operations for 8-bit values. It takes two inputs and gives result as output. 8 OR gates are used for every bit of two inputs.

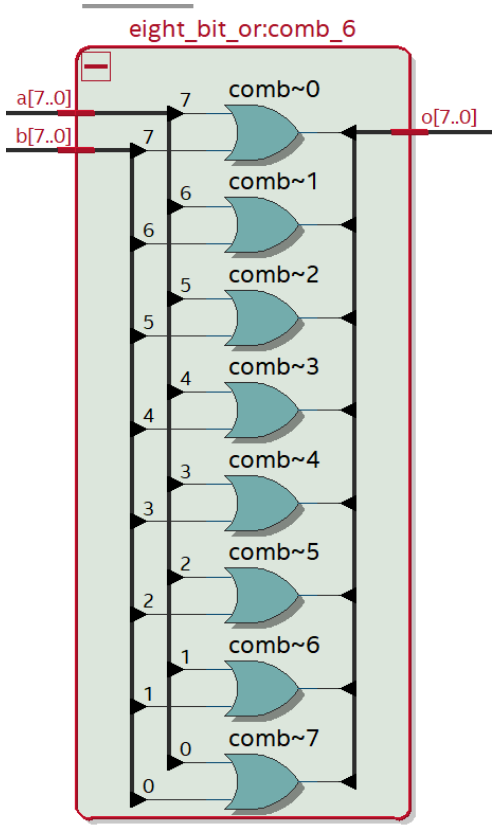

Figure-1: Inside of 8-bit Or Operator

The Verilog code of 8-bit or operator is below:

```
module eight_bit_or(

    input [7:0] a,
    input [7:0] b,
    output [7:0] o

);
    or(o[7],a[7],b[7]);
    or(o[6],a[6],b[6]);
    or(o[5],a[5],b[5]);
    or(o[4],a[4],b[4]);
    or(o[3],a[3],b[3]);
    or(o[2],a[2],b[2]);
    or(o[1],a[1],b[1]);
    or(o[0],a[0],b[0]);


endmodule
```

Figure-2: Verilog code of 8-bit Or Operation

Opcode of 8-bit or operation is 0011. The simulation of or operation is below:

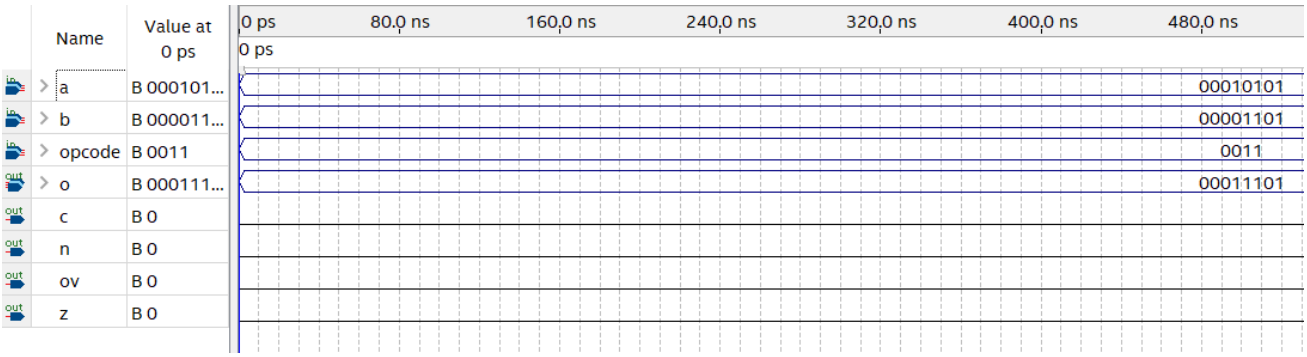

Figure-3: Simulation of Or Operation

8-bit Move Operator

8-bit move operator is used to move input to result without any changes. It takes one input (input a, same with first input of ALU) and gives result as output. 8 AND gates are used to make bitwise and operations for same bits of input.

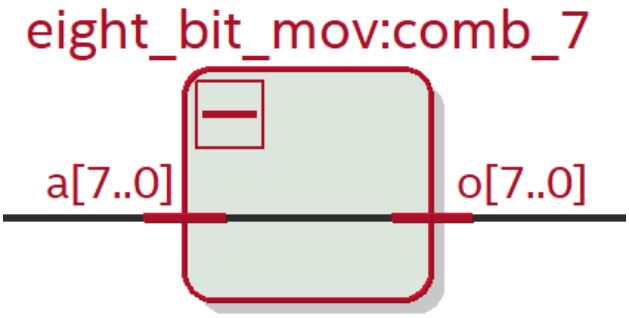

Figure-1: Inside of 8-bit Move operator

The Verilog code of 8-bit or operator is below:

```
module eight_bit_mov(

    input [7:0] a,
    output [7:0] o

);

    and(o[7],a[7],a[7]);
    and(o[6],a[6],a[6]);
    and(o[5],a[5],a[5]);
    and(o[4],a[4],a[4]);
    and(o[3],a[3],a[3]);
    and(o[2],a[2],a[2]);
    and(o[1],a[1],a[1]);
    and(o[0],a[0],a[0]);

endmodule
```

Figure-2: Verilog code of 8-bit move operator

Opcode of move operation is 0100. The simulation of move operation is below:

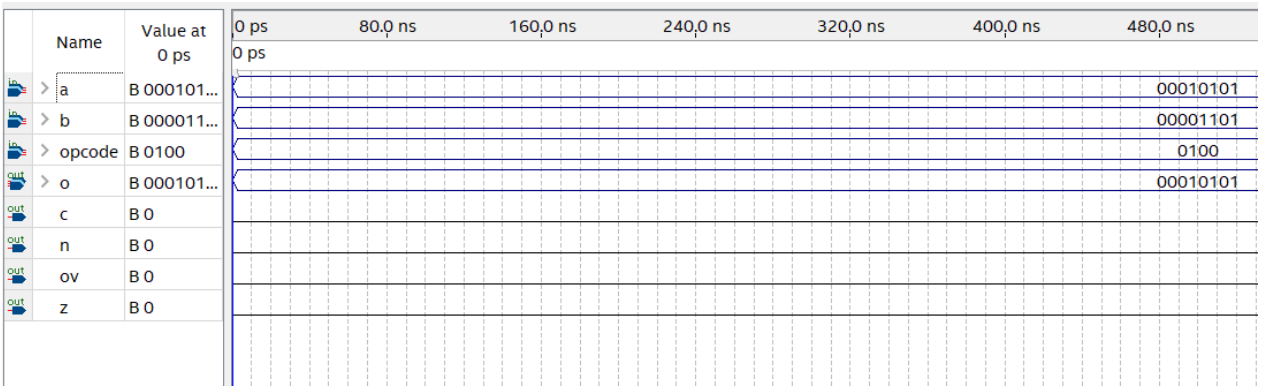

Figure-3: Simulation of 8-bit move operation

8-bit Shift Left Operator

In this operation, each bit is shifted one unit to the left. The leftmost bit is gone and replaced with a rightmost 0. We use an additional module called one_bit_shifter to perform this operation. What is done in this module is to take back the given input as it was with or gate. In 8-bit left shifter, using this module, each bit is shifted one unit to the left. Finally, the right most bit is set to zero.

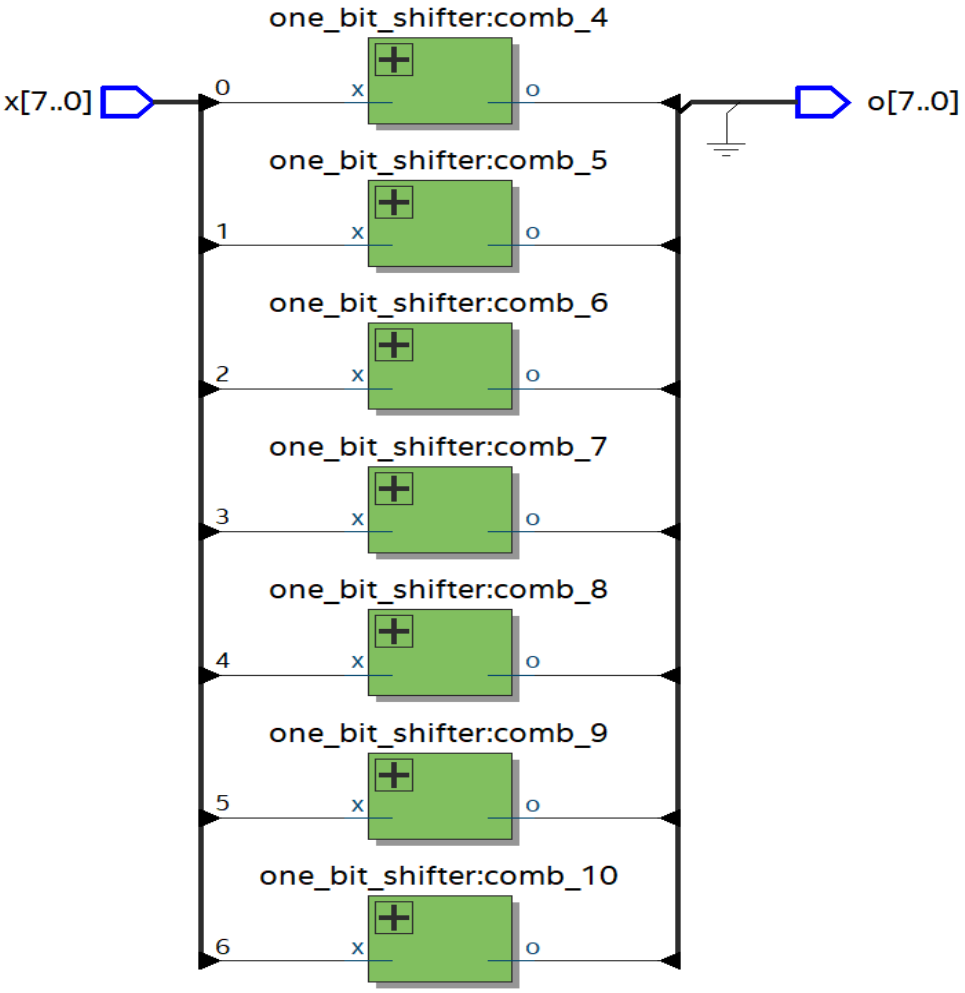

Figure-1: Inside of 8-bit shift left operator

```
1  module eight_bit_left_shifter(
2      input[7:0] x,
3      output[7:0] o
4  );
5      or(o[0], 1'b0, 1'b0);
6      one_bit_shifter(x[0], o[1]);
7      one_bit_shifter(x[1], o[2]);
8      one_bit_shifter(x[2], o[3]);
9      one_bit_shifter(x[3], o[4]);
10     one_bit_shifter(x[4], o[5]);
11     one_bit_shifter(x[5], o[6]);
12     one_bit_shifter(x[6], o[7]);
13 endmodule
14
15
16 module one_bit_shifter(
17     input x,
18     output o
19 );
20     or(o,x,1'b0);
21 endmodule
```

Figure-2: Verilog code for the 8-bit shift left operator

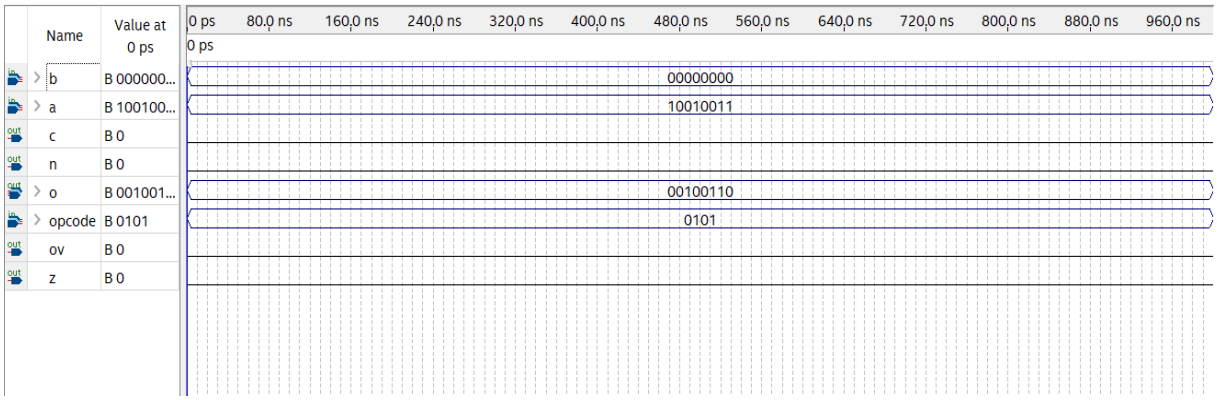

Figure-3: Simulation for the 8-bit shift left operator

8-bit Shift Right Operator

The method used in shift left operation is used. It's just that the bits are shifted right instead of left. The right most bit is eliminated and the leftmost bit is replaced by 0.

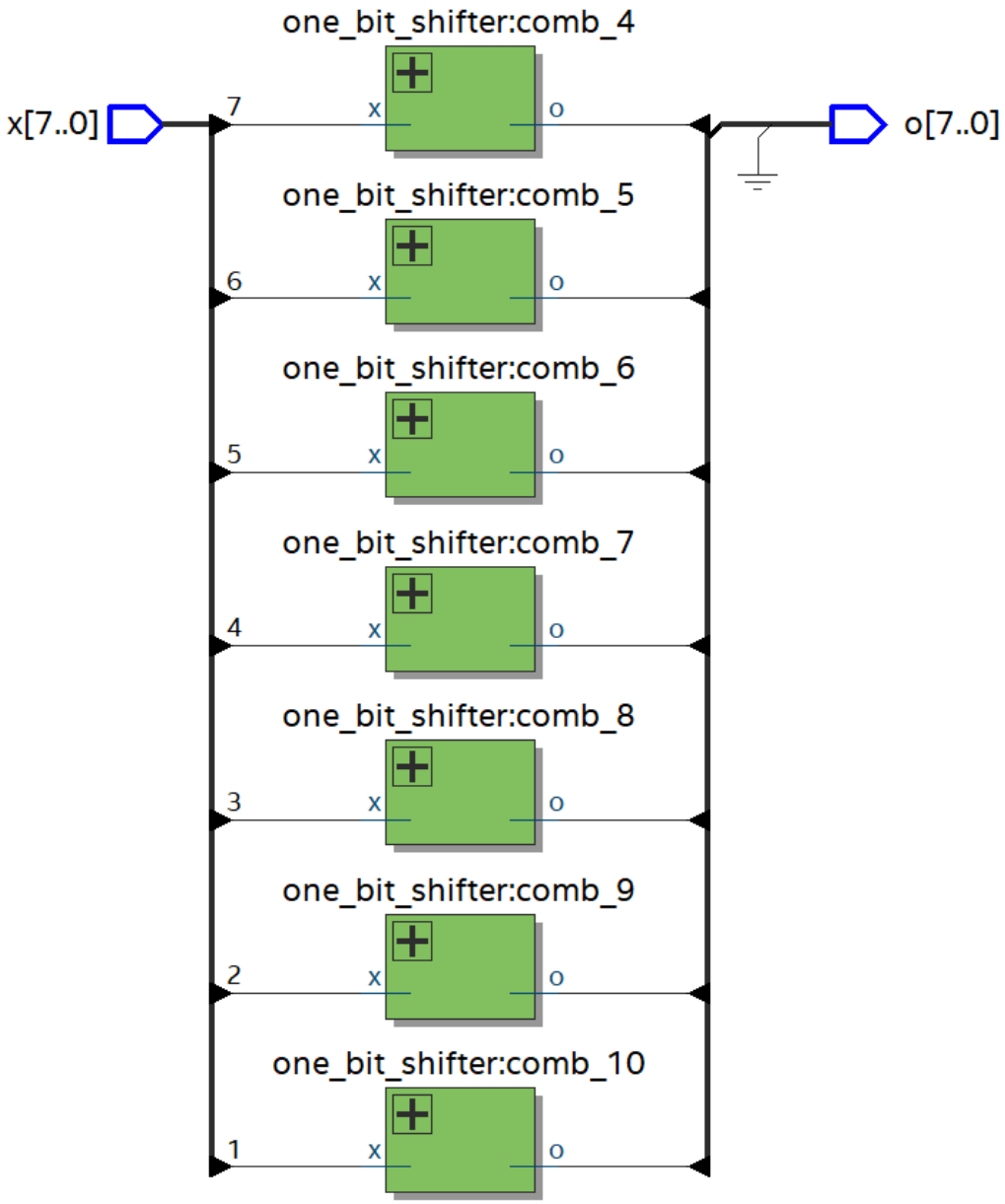

Figure-1: Inside of 8-bit shift right operator

```
1  module eight_bit_right_shifter(
2      input[7:0] x,
3      output[7:0] o
4  );
5      or(o[7], 1'b0, 1'b0);
6      one_bit_shifter(x[7], o[6]);
7      one_bit_shifter(x[6], o[5]);
8      one_bit_shifter(x[5], o[4]);
9      one_bit_shifter(x[4], o[3]);
10     one_bit_shifter(x[3], o[2]);
11     one_bit_shifter(x[2], o[1]);
12     one_bit_shifter(x[1], o[0]);
13 endmodule
14
15
16 module one_bit_shifter(
17     input x,
18     output o
19 );
20     or(o,x,1'b0);
21 endmodule
```

Figure-2: Verilog code of 8-bit shift left operator

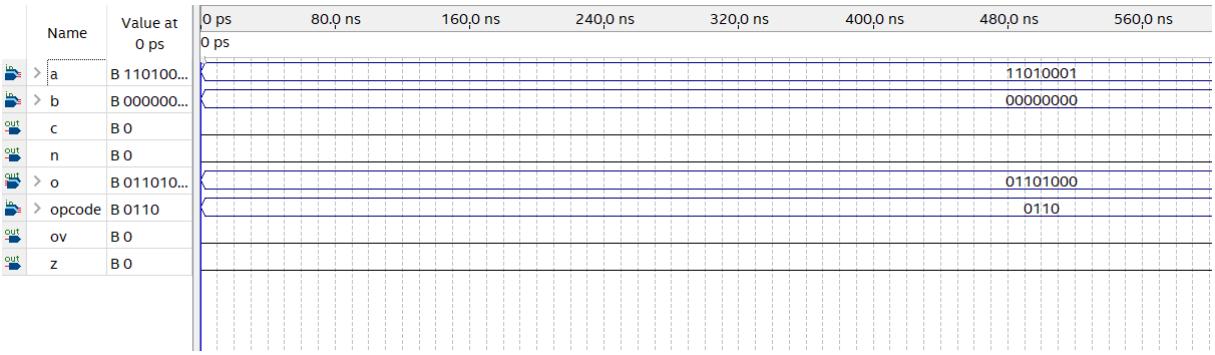

Figure-3: Simulation of 8-bit shift left operator

8-bit Incrementer

An 8-bit adder is used for this operation. The only difference is 1 is given for the second input parameter.

```
module eight_bit_inc(

    input [7:0] x,
    output [7:0] o,
    output cout,
    output overflow

);

    eight_bit_adder(x,1,o,cout,overflow);

endmodule
```

Figure-1: Inside of 8-bit incrementer

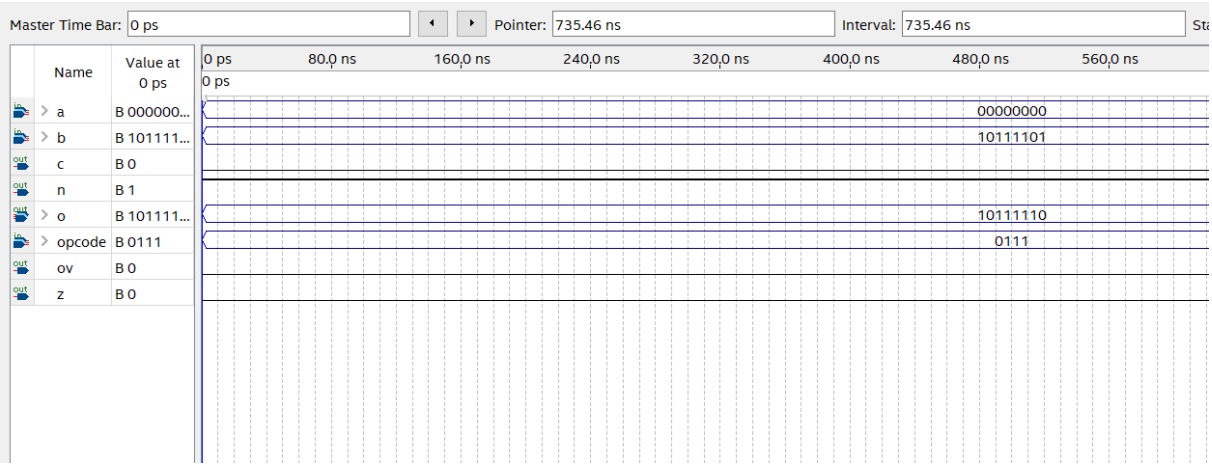

Figure-2: Simulation of 8-bit incrementer

8-bit Decrementer

An 8-bit subtractor is used for this operation. The only difference is 1 is given for the second input parameter.

```
1  module eight_bit_dec(
2
3      input [7:0] x,
4      output [7:0] o,
5      output cout,
6      output overflow
7  );
8
9      eight_bit_subtractor(x,1,o,cout,overflow);
10
11 endmodule
```

Figure-1: Inside of 8-bit decrementer

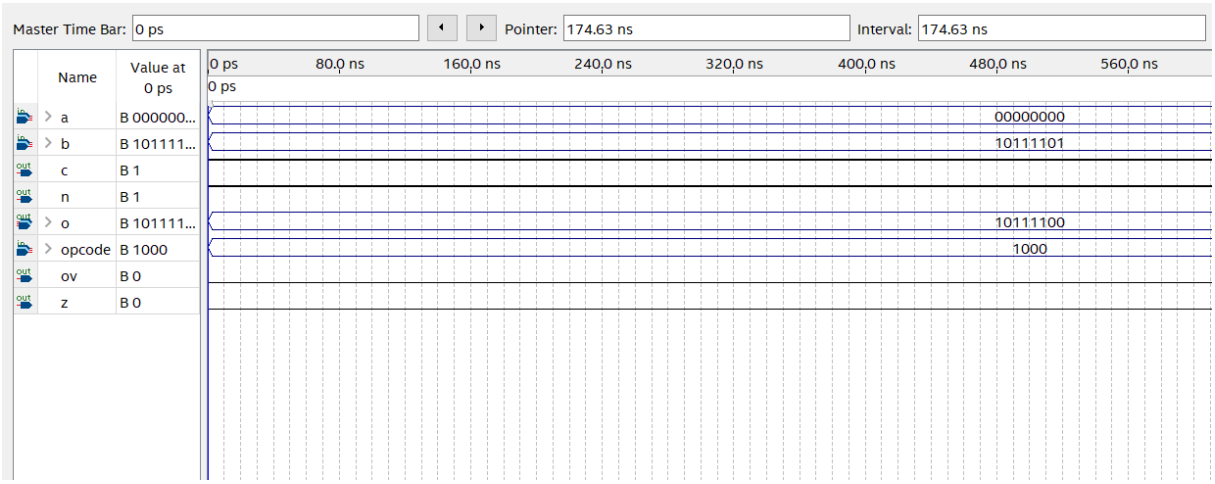

Figure-2: Inside of 8-bit decrementer

FLAGS

Carry Flag

Carry flag is controlled in 8-bit adder. So, when 8-bit adder is used inside of 8-bit subtractor, 8-bit incrementor and 8-bit decrementor, carry flag is set automatically by 8-bit adder.

Carry flag is controlled by taking the value of carry out bit. If carry out bit is 1, then carry flag will be set to 1.

Add, sub, inc and dec operations effect carry flag. The simulation of carry flag with sub operation is below:

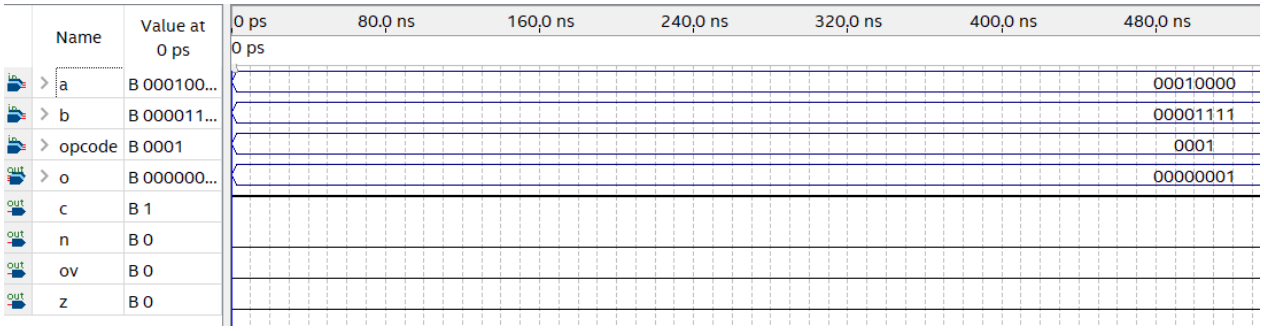

Figure-1: Simulation of carry flag

Zero Flag

Zero flag is controlled with its own module. In module, there is one OR gate which takes every bit of the input. After that, not gate gives 1 as output if all the bits are 0. 6 OR gates and 1 NOR gate are used to build this structure in Verilog code.

Zero flag is controlled in exit of ALU similar to negative flag. The Verilog code of zero flag operation is below:

```
module zero_flag(

    input [7:0] x,
    output o

);


     wire [5:0] t;

     or(t[5],x[7],x[6]);
     or(t[4],x[5],x[4]);
     or(t[3],x[3],x[2]);
     or(t[2],x[1],x[0]);

     or(t[1],t[5],t[4]);
     or(t[0],t[3],t[2]);

     nor(o,t[1],t[0]);

endmodule
```

Figure-1: Verilog code of zero flag module

Gate level display of zero flag module is below:

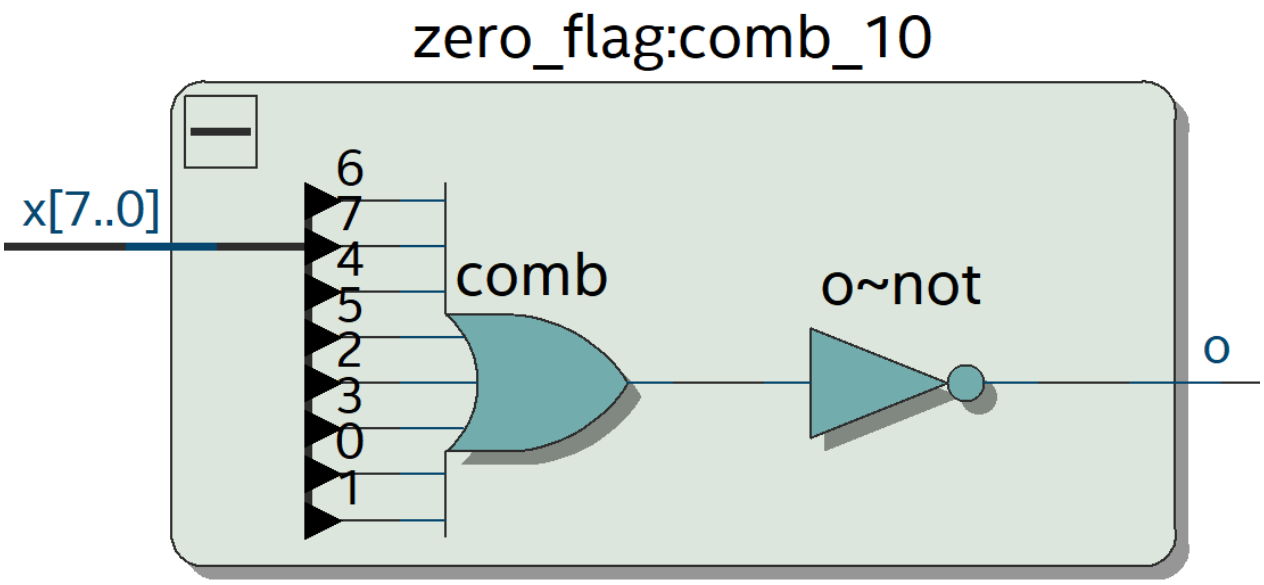

Figure-2: Inside of zero flag module

The simulation of zero flag is below:

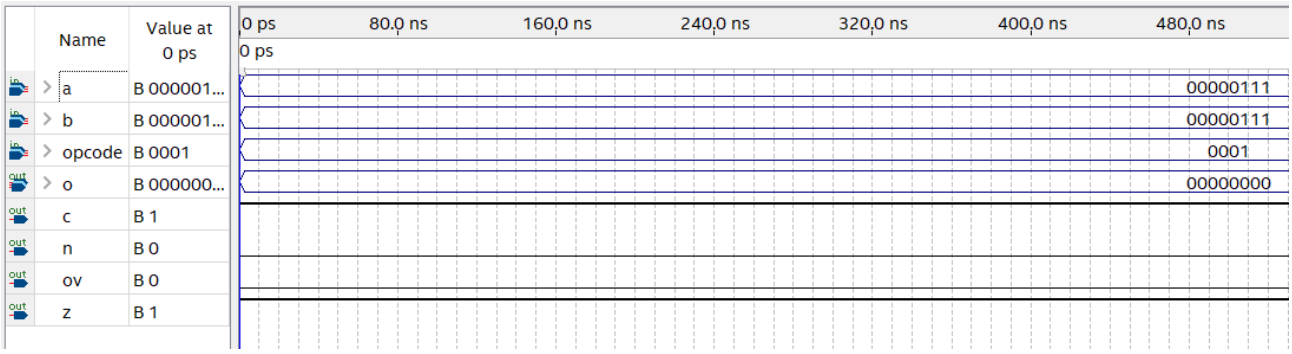

Figure-3: Simulation of zero flag

Negative Flag

Negative flag is controlled in exit of ALU. The result of selected operation will be controlled with XNOR gate. XNOR gate is used to check whether most significant bit of result is equal to 1. If it is equal to one, then it means it is a negative result. Negative flag will be set to 1.

The Verilog code of negative flag operation is below:

```
xnor(n,o[7],1);
```

Figure-1: Verilog code of negative flag

The simulation of negative flag is below:

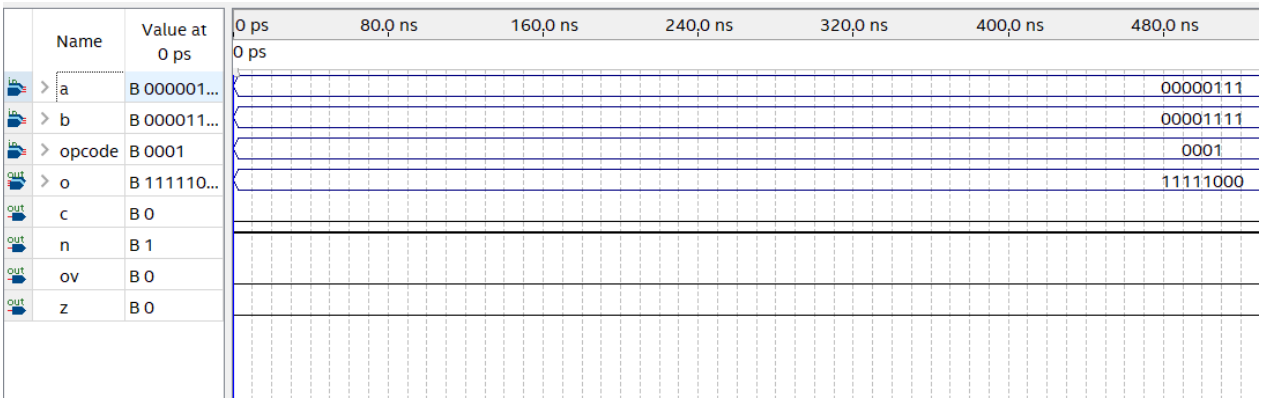

Figure-2: Simulation of zero flag

Overflow Flag

Overflow flag is controlled in 8-bit adder. So, when 8-bit adder is used inside of 8-bit subtractor, 8-bit incrementor and 8-bit decrementor, overflow flag is set automatically by 8-bit adder.

Overflow flag is controlled by taking carry in and carry out bits of most significant bit (eighth bit) of the result to XOR gate. If overflow occurs, then overflow bit will be 1. Verilog code of overflow flag is below:

```
module eight_bit_adder(
    input [7:0] x,
    input [7:0] y,
    output [7:0] o,
    output carryout,
    output overflow
);
    wire cout1, cout2, cout3, cout4, cout5, cout6, cout7;
    one_bit_adder(x[0], y[0], 1'b0, o[0], cout1);
    one_bit_adder(x[1], y[1], cout1, o[1], cout2);
    one_bit_adder(x[2], y[2], cout2, o[2], cout3);
    one_bit_adder(x[3], y[3], cout3, o[3], cout4);
    one_bit_adder(x[4], y[4], cout4, o[4], cout5);
    one_bit_adder(x[5], y[5], cout5, o[5], cout6);
    one_bit_adder(x[6], y[6], cout6, o[6], cout7);
    one_bit_adder(x[7], y[7], cout7, o[7], carryout);

    xor(overflow,cout7,carryout);
```

Figure-1: Verilog code of overflow flag

Add, sub, inc and dec operations effect overflow flag. The simulation of overflow flag with add operation is below:

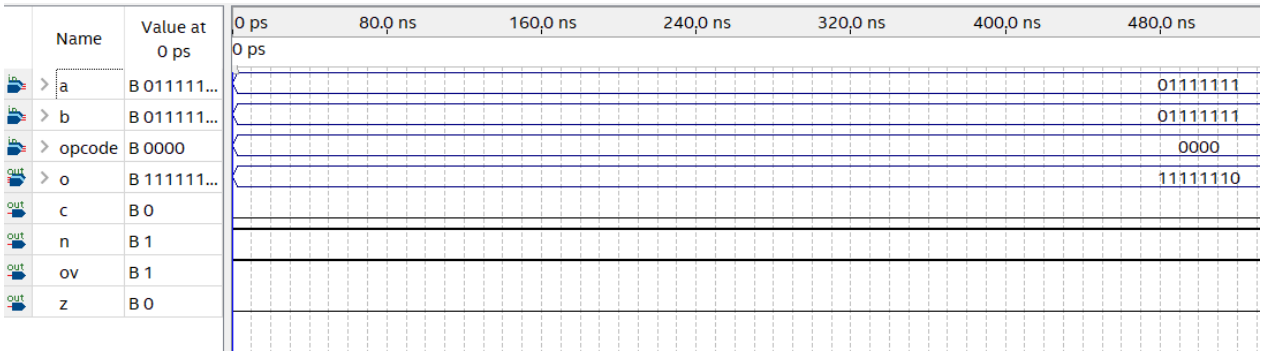

Figure-2: Simulation of overflow flag