Assignment –1 Report

By Rohan Gupta – 2020112022, Anjali Singh - 2020102004

Overview

We are required to build an ALU which performs the logic functions of AND, XOR and arithmetic operations of addition and subtraction of two 64-bit numbers, let's say 'a' and 'b' in Verilog. For this, we have created different modules for the logic gates. Even though, Verilog has built-in primitives for AND and XOR, we decided to write down their functionalities manually.

Here, the ALU unit takes the input as control signals – 0 for Add, 1 for Subtract, 2 for AND and 3 for XOR.

AND

The truth table for AND logic is as follows: -

a	**b**	**a.b**
0	0	0
0	1	0
1	0	0
1	1	1

Approach

We are required to make a 64-bit AND logic. For this, we created a 1-bit AND logic module by manually putting the conditions, according to the truth table and then using the file to make 64-bit AND logic. This can be done by generating a for loop and a new variable 'i' which is used to assign the 64 bits to 'a' and 'b' and on further using the 1-bit AND logic created.

Modules

And 1 - Bit

```
`timescale 1ns/1ps


module and_1bit(a, b, out);

input a, b ;
output reg out;

always @(a or b) begin

        if (a == 1'b1) begin
                if(b == 1'b1)
                        out = 1'b1;

                else
                        out = 1'b0;
        end
        else
                out = 1'b0;
end

endmodule
```

We can use these 1 - bit Modules to And 64 bits using generate block.

AND 64 - bit

```
`timescale  1ns / 1ps

module and_64bit(
        input signed [63:0]a,
        input signed [63:0]b,
        output signed [63:0] out
);

genvar i;

generate for(i = 0; i < 64; i = i+1)
begin
        and_1bit g1(a[i], b[i], out[i]);
end

endgenerate
```

```
endmodule
```

Testbench

For AND 1 - Bit Module

```
`timescale 1ns/1ps


module and_1bittb;

        reg a, b;
        wire out;




        and_1bit dut(.a(a), .b(b), .out(out));

        initial begin
                $dumpfile("and_1bit.vcd");
                $dumpvars(0, and_1bittb);

                a=1'b0;
                b=1'b0;
        repeat(4) begin
                #10 a = ~a;
                #20 b = ~b;
        end

        end




  initial
                $monitor("a=%b b=%b out=%b \n",a,b,out);

endmodule
```

For AND 64 - bit Module

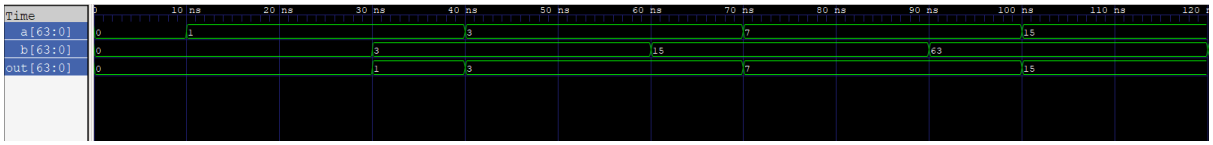
```
`timescale 1ns/1ps
```

```
`include "and_1bit.v"

module and_64bittb;

        reg signed [63:0]a;
        reg signed [63:0]b;

        wire signed [63:0]out;

        and_64bit dut(.a(a),.b(b),.out(out));

        initial begin
                $dumpfile("and_64bit.vcd");
                $dumpvars(0,and_64bittb);

                a=64'b0;
                b=64'b0;

        repeat(4) begin
                #10 begin
                        a = ~a;
                        a = a<<1;
                        a = ~a;
                end
                #20 begin
                        b = ~b;
                        b = b<<2;
                        b = ~b;
                end
        end

        end

  initial
                $monitor("a=%b b=%b\nout=%b \n\n",a,b,out);

endmodule
```

Results:

```
a=0000000000000000000000000000000000000000000000000000000000000000 b=0000000000000000000000000000000000000000000000000000000000000000
out=0000000000000000000000000000000000000000000000000000000000000000


a=0000000000000000000000000000000000000000000000000000000000000001 b=0000000000000000000000000000000000000000000000000000000000000000
out=0000000000000000000000000000000000000000000000000000000000000000


a=0000000000000000000000000000000000000000000000000000000000000001 b=0000000000000000000000000000000000000000000000000000000000000011
out=0000000000000000000000000000000000000000000000000000000000000001


a=0000000000000000000000000000000000000000000000000000000000000011 b=0000000000000000000000000000000000000000000000000000000000000011
out=0000000000000000000000000000000000000000000000000000000000000011


a=0000000000000000000000000000000000000000000000000000000000000011 b=0000000000000000000000000000000000000000000000000000000000001111
out=0000000000000000000000000000000000000000000000000000000000000011


a=0000000000000000000000000000000000000000000000000000000000000111 b=0000000000000000000000000000000000000000000000000000000000001111
out=0000000000000000000000000000000000000000000000000000000000000111


a=0000000000000000000000000000000000000000000000000000000000000111 b=0000000000000000000000000000000000000000000000000000000000111111
out=0000000000000000000000000000000000000000000000000000000000000111


a=0000000000000000000000000000000000000000000000000000000000001111 b=0000000000000000000000000000000000000000000000000000000000111111
out=0000000000000000000000000000000000000000000000000000000000001111


a=0000000000000000000000000000000000000000000000000000000000001111 b=0000000000000000000000000000000000000000000000000000000011111111
out=0000000000000000000000000000000000000000000000000000000000001111
```

Gtkwave Output:



Addition (ADD)

The truth table for ADD logic is as follows: -

- For 1-Bit

a	b	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

- For Full adder

a	b	Carry in	Sum	Carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1

1	1	1	1	1

Approach

Here,

- sum[i] = a[i] XOR b[i] XOR carryin[i] and
- Carryout = carry[i+1] = a[i].b[i] + b[i].c[i] + c[i].a[i]

The above functions are made using the modules made earlier. For 1-Bit ADD block, we have used AND 1-Bit and XOR 1-Bit and then using them directly to generate the functions for single bits above. For this, we have included all the required files in the module. Then, for 64-Bit Add module, we have used the 1-Bit Add module using the generate block, as a loop.

Modules

For Add 1-Bit:

```
`timescale 1ns / 1ps

    `include "../And/and_1bit.v"
    `include "../Or/or_1bit.v"
    `include "../Xor/xor_1bit.v"

module add_1bit(
  input a,
  input b,
  input cin,
  output sum,
  output co);

  xor_1bit g1(a, b, d);
  xor_1bit g2(cin, d, sum);

  and_1bit g3(a, b, k);
  and_1bit g4(a, cin, l);
  and_1bit g5(b, cin, m);

  or_1bit g6(m, k, q);
  or_1bit g7(q, l, co);

endmodule
```

For Add 64-Bit:

```
`timescale 1ns/1ps
module add_64bit(a, b, out, overflow);

        input signed [63:0]a ;
        input signed [63:0]b ;
        output signed [63:0]out ;
        output overflow;

        wire [64:0]carry;

        assign carry[0] = 1'b0;

        genvar i;

        generate for(i = 0; i < 64; i = i+1)
        begin
                add_1bit g1(a[i], b[i], carry[i], out[i], carry[i+1]);
        end
        endgenerate

        xor g2(overflow, carry[64], carry[63]);

endmodule
```

Testbench

```
`timescale 1ns/1ps
`include "add_1bit.v"

module add_64bittb;

  reg signed [63:0]a;
  reg signed [63:0]b;

  wire signed [63:0]out;
  wire  overflow;
  add_64bit dut(.a(a), .b(b), .out(out), .overflow(overflow));

  initial begin
    $dumpfile("add_64bit.vcd");
    $dumpvars(0, add_64bittb);

    a=64'd11;
    b=64'd4;

repeat(2) begin
```

```
    #2 begin
      a = -a;
    end
    #2 begin
      b = -b;
    end
  end


  repeat(4) begin

 #2 begin
      a = ~a;
      a = a<<2;
      a = ~a;
    end

    #2 begin
      b = ~b;
      b = b<<4;
      b = ~b;
    end

  end

    #2 a =
64'b0111111111111111111111111111111111111111111111111111111111111111;
    #2 b =
64'b0000000000000000000000000000000000000000000000000000000000000001;

  end

  initial
    $monitor("a=%d b=%d out=%d overflow=%b\n",a, b, out, overflow);

endmodule
```

Results:

```
a=                  11 b=                   4 out=                  15 overflow=0
a=                 -11 b=                   4 out=                  -7 overflow=0
a=                 -11 b=                  -4 out=                 -15 overflow=0
a=                  11 b=                  -4 out=                   7 overflow=0
a=                  11 b=                   4 out=                  15 overflow=0
a=                  47 b=                   4 out=                  51 overflow=0
a=                  47 b=                  79 out=                 126 overflow=0
a=                 191 b=                  79 out=                 270 overflow=0
a=                 191 b=                1279 out=                1470 overflow=0
a=                 767 b=                1279 out=                2046 overflow=0
a=                 767 b=               20479 out=               21246 overflow=0
a=                3071 b=               20479 out=               23550 overflow=0
a=                3071 b=              327679 out=              330750 overflow=0
a= 9223372036854775807 b=              327679 out=-9223372036854448130 overflow=1
a= 9223372036854775807 b=                   1 out=-9223372036854775808 overflow=1
```

Gtkwave Output:

```
a=                  11 b=                   4 out=                  15 overflow=0
a=                 -11 b=                   4 out=                  -7 overflow=0
a=                 -11 b=                  -4 out=                 -15 overflow=0
a=                  11 b=                  -4 out=                   7 overflow=0
a=                  11 b=                   4 out=                  15 overflow=0
a=                  47 b=                   4 out=                  51 overflow=0
a=                  47 b=                  79 out=                 126 overflow=0
a=                 191 b=                  79 out=                 270 overflow=0
a=                 191 b=                1279 out=                1470 overflow=0
a=                 767 b=                1279 out=                2046 overflow=0
a=                 767 b=               20479 out=               21246 overflow=0
a=                3071 b=               20479 out=               23550 overflow=0
a=                3071 b=              327679 out=              330750 overflow=0
a= 9223372036854775807 b=              327679 out=-9223372036854448130 overflow=1
a= 9223372036854775807 b=                   1 out=-9223372036854775808 overflow=1
```

XOR

The truth table for XOR logic is as follows: -

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

Approach

Here, we are required to design a 64-bit XOR module. For making a XOR logic gate for two 64-bit numbers, we created a 1-bit XOR module, where we manually wrote the outputs of every input of ‘a’ and ‘b’. Therein, we used this module in making the 64-bit XOR module by generating a for loop for all the 64 bits.

Modules

Xor 1-bit:

```
`timescale 1ns/1ps

module xor_1bit(a, b, out);

input a ,b ;
output reg out;

always @(a or b ) begin

        if (a == 1'b1) begin
                if(b == 1'b0)
                        out = 1'b1;

                else
                        out = 1'b0;
        end
        else
                if(b == 1'b1)
                        out = 1'b1;

                else
                        out = 1'b0;
end
```

```
endmodule
```

Xor 64-bit:

```
`timescale  1ns/1ps


        input signed [63:0]a,
        input signed [63:0]b,
        output signed [63:0] out
);


        xor_1bit g1(a[i], b[i], out[i]);
end
```

Testbench:

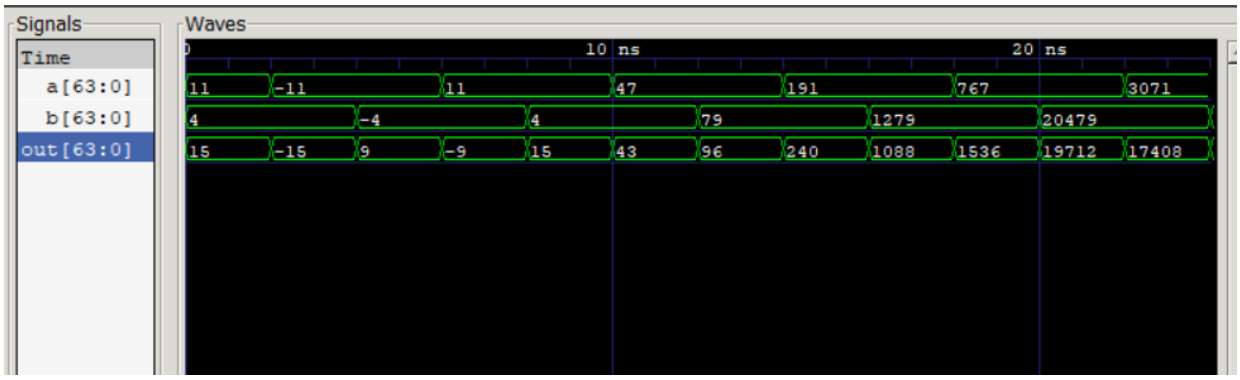
```
`timescale 1ns/1ps
`include "xor_1bit.v"

module xor_64bittb;

        reg signed [63:0]a;
        reg signed [63:0]b;

        wire signed [63:0]out;

        xor_64bit dut(.a(a),.b(b),.out(out));

        initial begin
                $dumpfile("xor_64bit.vcd");
                $dumpvars(0, xor_64bittb);


                a=64'd11;
                b=64'd4;

                repeat(2) begin
                        #2 begin
                                a = -a;
                        end
                        #2 begin
```

```
                                b = -b;
                        end
                        end


                repeat(4) begin

                        #2 begin
                                a = ~a;
                                a = a<<2;
                                a = ~a;
                                end

                        #2 begin
                                b = ~b;
                                b = b<<4;
                                b = ~b;
                        end

                end


                end

  initial
                $monitor("a=%b b=%b out=%b \n", a, b, out);

endmodule
```

Results:

```
a=00000000000000000000000000000000000000000000000000000000000001011 b=00000000000000000000000000000000000000000000000000000000000000100 out=00000000000000000000000000000000
00000000000000000000000000001111

a=11111111111111111111111111111111111111111111111111111111110101 b=00000000000000000000000000000000000000000000000000000000000000100 out=11111111111111111111111111111111
11111111111111111111111111110001

a=11111111111111111111111111111111111111111111111111111111110101 b=11111111111111111111111111111111111111111111111111111111111111100 out=00000000000000000000000000000000
00000000000000000000000000001001

a=00000000000000000000000000000000000000000000000000000000000001011 b=11111111111111111111111111111111111111111111111111111111111111100 out=11111111111111111111111111111111
11111111111111111111111110111

a=00000000000000000000000000000000000000000000000000000000000001011 b=00000000000000000000000000000000000000000000000000000000000000100 out=00000000000000000000000000000000
00000000000000000000000000001111

a=00000000000000000000000000000000000000000000000000000000000101111 b=00000000000000000000000000000000000000000000000000000000000000100 out=00000000000000000000000000000000
00000000000000000000000000101011

a=00000000000000000000000000000000000000000000000000000000000101111 b=00000000000000000000000000000000000000000000000000000000001001111 out=00000000000000000000000000000000
00000000000000000000000001100000

a=00000000000000000000000000000000000000000000000000000000010111111 b=00000000000000000000000000000000000000000000000000000000001001111 out=00000000000000000000000000000000
00000000000000000000000011110000

a=00000000000000000000000000000000000000000000000000000000010111111 b=00000000000000000000000000000000000000000000000000000010011111111 out=00000000000000000000000000000000
00000000000000000000010001000000

a=00000000000000000000000000000000000000000000000000000001011111111 b=00000000000000000000000000000000000000000000000000000010011111111 out=00000000000000000000000000000000
00000000000000000000011000000000

a=00000000000000000000000000000000000000000000000000000001011111111 b=00000000000000000000000000000000000000000000000000000100111111111111 out=00000000000000000000000000000000
00000000000000000010011010000000

a=00000000000000000000000000000000000000000000000000000101111111111 b=00000000000000000000000000000000000000000000000000000100111111111111 out=00000000000000000000000000000000
00000000000000000100010000000000

a=00000000000000000000000000000000000000000000000000000101111111111 b=00000000000000000000000000000000000000000000000000000100111111111111111 out=00000000000000000000000000000000
00000000000000010011110100000000
```

Gtkwave Output:



Subtract (SUB)

The truth table for subtract is as follows: -

a	b	a-b	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Approach:

Here, we are supposed to make a 64-bit subtractor which performs subtraction, b from a (a – b). For this let us consider the expression, a + (-b). This includes addition of two numbers, ‘a’ and ‘-b.’ Therefore, the full adder module, that we have made earlier can be used to determine this operation. For negating ‘b,’ we are going to use the method of 2’s complement, where all 64 bits of ‘b’ are inverted using the NOT logic and then a single bit of 1 is added. This is how we get negation ‘b.’ Now, using the full adder module we created, we will simply add ‘a’ and ‘-b.’

Modules

Not 1-bit:

```
`timescale 1ns / 1ps

module not_1bit (
    input a,
    output no
);
reg no;

always @(a) begin

    if (a == 1'b0)
    begin
        no = 1'b1;
    end

    else if (a == 1'b1)
    begin
        no = 1'b0;
    end
end

endmodule
```

Not 64-bit:

```
`timescale 1ns / 1ps

module not_64bit (
    input signed [63:0]a,
    output signed [63:0]out
);
```

```
genvar i;

generate for(i = 0; i < 64; i = i + 1)
begin
    not_1bit g1(a[i], out[i]);
end

endgenerate

endmodule
```

Sub 64-bit:

```
`timescale 1ns / 1ps


module sub_64bit (a, b, out, overflow);

input signed [63:0]a;
input signed [63:0]b;
output signed [63:0]out;
output overflow;

wire [63:0]no; // no = not output
not_64bit g1(b, no); // inverting bits of input b

wire [63:0]finv; // for final inversion
assign finv = 64'b1;

wire [63:0]add_finv;
add_64bit g2(finv, no, add_finv, fadd);

add_64bit g3(a, add_finv, out, overflow);

endmodule
```

Testbench

```
`timescale 1ns/1ps


`include "add_64bit.v"
`include "add_1bit.v"
`include "not_1bit.v"
`include "not_64bit.v"

module sub_64bittb;
```

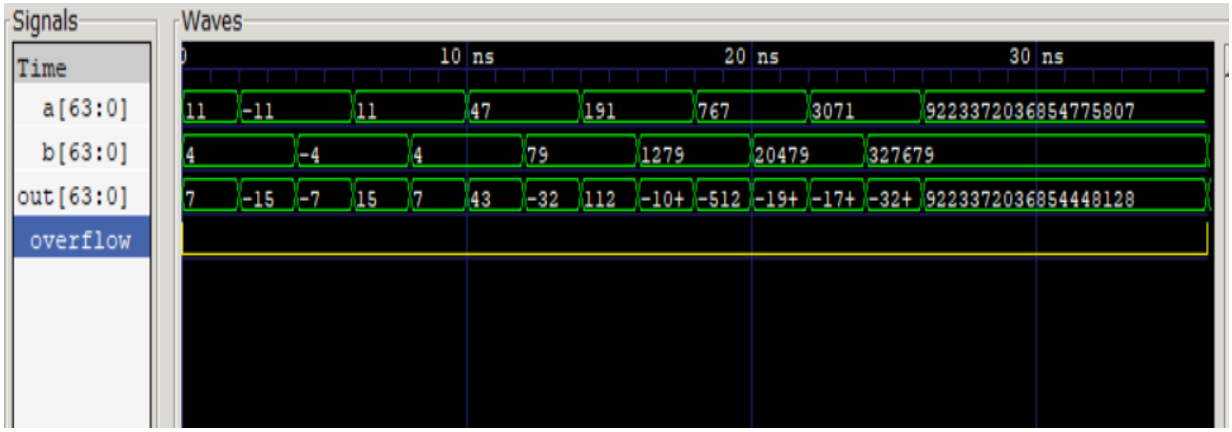
```
        reg signed [63:0]a;
        reg signed [63:0]b;

        wire signed [63:0]out;
        wire  overflow;
        sub_64bit dut(.a(a), .b(b), .out(out), .overflow(overflow));

        initial begin
                $dumpfile("sub_64bit.vcd");
                $dumpvars(0, sub_64bittb);

                a=64'd11;
                b=64'd4;

                repeat(2) begin
                        #2 begin
                                a = -a;
                        end
                        #2 begin
                                b = -b;
                        end
                        end


                repeat(4) begin

                        #2 begin
                                a = ~a;
                                a = a<<2;
                                a = ~a;
                                end

                        #2 begin
                                b = ~b;
                                b = b<<4;
                                b = ~b;
                        end

                end
                        #2 a =
64'b0111111111111111111111111111111111111111111111111111111111111111;
                        #2 b = -64'd1;

                end

  initial
```

```
                $monitor("a=%d b=%d out=%d overflow=%b\n", a, b, out,
overflow);

endmodule
```

Results:

```
a=                  11 b=                   4 out=                   7 overflow=0
a=                 -11 b=                   4 out=                 -15 overflow=0
a=                 -11 b=                  -4 out=                  -7 overflow=0
a=                  11 b=                  -4 out=                  15 overflow=0
a=                  11 b=                   4 out=                   7 overflow=0
a=                  47 b=                   4 out=                  43 overflow=0
a=                  47 b=                  79 out=                 -32 overflow=0
a=                 191 b=                  79 out=                 112 overflow=0
a=                 191 b=                1279 out=               -1088 overflow=0
a=                 767 b=                1279 out=                -512 overflow=0
a=                 767 b=               20479 out=              -19712 overflow=0
a=                3071 b=               20479 out=              -17408 overflow=0
a=                3071 b=              327679 out=             -324608 overflow=0
a= 9223372036854775807 b=              327679 out= 9223372036854448128 overflow=0
a= 9223372036854775807 b=                  -1 out=-9223372036854775808 overflow=1
```

Gtkwave Output:



ALU

Approach

Approach ALU consists of all the functions we have made so far. The modules below consist of the included files, case statements to assign the control of each functionality with its respective control number.

`Control 0 - ADD x and y Control 1 – Subtract y from x Control 2 – AND x and y Control 3 – XOR x and y`

Modules

ALU:

```
`timescale 1ns/1ps

`include "../And/and_1bit.v"
`include "../And/and_64bit.v"
`include "../Xor/xor_1bit.v"
`include "../Xor/xor_64bit.v"
`include "../Or/or_1bit.v"

`include "../Add/add_1bit.v"
`include "../Add/add_64bit.v"

`include "../Sub/not_1bit.v"
`include "../Sub/not_64bit.v"
`include "../Sub/sub_64bit.v"


module alu (


        input[1:0]control,
        input signed [63:0]a,
        input signed [63:0]b,
        output signed[63:0]out,
        output overflow
);


        reg signed[63:0]out;
        reg overflow;

        wire signed [63:0] out1;
        wire overflow1;
        wire signed [63:0] out2;
```

```
        wire overflow2;
        wire signed [63:0] out3;
        wire signed [63:0] out4;


        and_64bit g1(a, b, out3);
        xor_64bit g2(a, b, out4);
        add_64bit g3(a, b, out1, overflow1);
        sub_64bit g4(a, b, out2, overflow2);


        always@(*)
        begin
                case(control)
                        2'b00:begin
                                out = out1;
                                overflow = overflow1;
                        end
                        2'b01:begin
                                out = out2;
                                overflow = overflow2;

                        end
                        2'b10:begin
                                out = out3;
                                overflow = 1'b0;
                        end
                        2'b11:begin
                                out = out4;
                                overflow = 1'b0;
                        end

                endcase
        end



endmodule
```

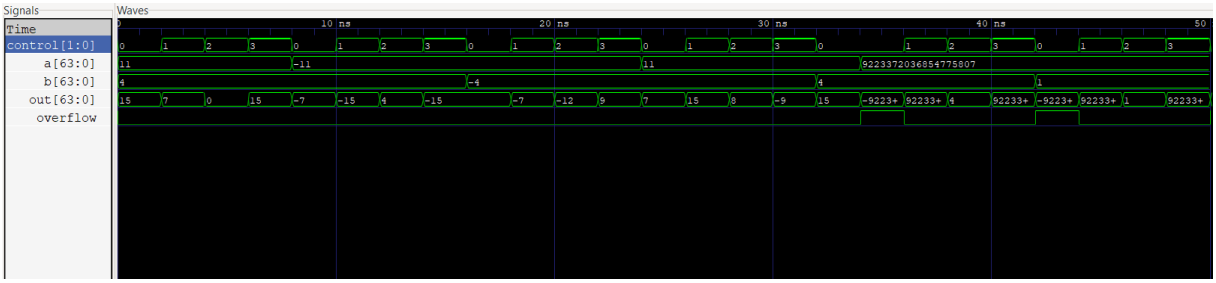
Testbench:

```
`timescale 1ns / 1ps

module alu_tb;

  reg signed [63:0]a;
  reg signed [63:0]b;
```

```
  wire signed [63:0]out;
  wire  overflow;
  reg [1:0] control;

  alu dut(.a(a), .b(b), .control(control), .out(out), .overflow(overflow));


  initial begin
    $dumpfile("alu.vcd");
    $dumpvars(0,alu_tb);


  control = 2'b00;

  a=64'b1011;
  b=64'b0100;

  repeat(2) begin
    #2 control = control + 2'b01;
    #2 control = control + 2'b01;
    #2 control = control + 2'b01;
    #2 begin
      control = control + 2'b01;
      a = -a;
    end

    #2 control = control + 2'b01;
    #2 control = control + 2'b01;
    #2 control = control + 2'b01;
    #2 begin
      control = control + 2'b01;
      b = -b;
    end
  end

  #2 a = 64'b0111111111111111111111111111111111111111111111111111111111111111;

  repeat(2) begin
    #2 control = control + 2'b01;
    #2 control = control + 2'b01;
    #2 control = control + 2'b01;


    #2 begin
      control = control + 2'b01;
      b =
64'b0000000000000000000000000000000000000000000000000000000000000001;

    end
```

```
  end


  end

  initial
    $monitor("control = %d a=%d
b=%d  out=%d  overflow=%b\n",control,a,b,out,overflow);

endmodule
```

Gtkwave Output:



Results:

```
control = 0 a=                  11 b=                   4  out=                  15  overflow=0
control = 1 a=                  11 b=                   4  out=                   7  overflow=0
control = 2 a=                  11 b=                   4  out=                   0  overflow=0
control = 3 a=                  11 b=                   4  out=                  15  overflow=0
control = 0 a=                 -11 b=                   4  out=                  -7  overflow=0
control = 1 a=                 -11 b=                   4  out=                 -15  overflow=0
control = 2 a=                 -11 b=                   4  out=                   4  overflow=0
control = 3 a=                 -11 b=                   4  out=                 -15  overflow=0
control = 0 a=                 -11 b=                  -4  out=                 -15  overflow=0
control = 1 a=                 -11 b=                  -4  out=                  -7  overflow=0
control = 2 a=                 -11 b=                  -4  out=                 -12  overflow=0
control = 3 a=                 -11 b=                  -4  out=                   9  overflow=0
control = 0 a=                  11 b=                  -4  out=                   7  overflow=0
control = 1 a=                  11 b=                  -4  out=                  15  overflow=0
control = 2 a=                  11 b=                  -4  out=                   8  overflow=0
control = 3 a=                  11 b=                  -4  out=                  -9  overflow=0
control = 0 a=                  11 b=                   4  out=                  15  overflow=0
control = 0 a= 9223372036854775807 b=                   4  out=-9223372036854775805  overflow=1
control = 1 a= 9223372036854775807 b=                   4  out= 9223372036854775803  overflow=0
control = 2 a= 9223372036854775807 b=                   4  out=                   4  overflow=0
control = 3 a= 9223372036854775807 b=                   4  out= 9223372036854775803  overflow=0
control = 0 a= 9223372036854775807 b=                   1  out=-9223372036854775808  overflow=1
```

```
control = 1 a= 9223372036854775807 b=                   1  out= 9223372036854775806  overflow=0
control = 2 a= 9223372036854775807 b=                   1  out=                   1  overflow=0
control = 3 a= 9223372036854775807 b=                   1  out= 9223372036854775806  overflow=0
control = 0 a= 9223372036854775807 b=                   1  out=-9223372036854775808  overflow=1
```