Assignment-1 Report

Adithya Sunil Edakkadan **2019102005**

Table of contents

Overview

The main objective of the assignment is to build an ALU with the following functionalities:

- ADD
- SUB
- AND
- XOR

All operations are 32 bits.

All input and output should be signed and use 2's complement for the subtraction.

The ALU unit takes as input the control signal, and two 32-bit inputs, and returns the 32-bit output corresponding to the control signal chosen.

I have decided to start with the XOR module

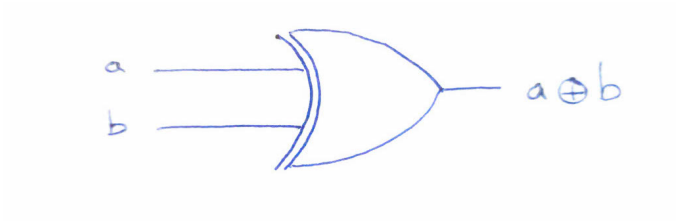
XOR

Logic, Circuit and Module design

The truth table for a XOR logic gate is as follows

a	b	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

The circuit for a XOR logic gate is as follows



Module design

Verilog has built-in gate gate primitives for all the basic gates including XOR, AND, OR, NOT, NAND, NOR and XNOR.

We are required to design a 32 bit XOR module. We have 2 possible approaches to do this. The first would be directly making a module that has 32 XOR gate primitives to operate on each bit. This module is not very modular. The second approach would be making a module that performs XOR on single bit inputs and using the single bit module repeatedly in another module for each of the 32 bits. This is much more modular as we can adjust the number of bits to our convenience without having to modify the initial module that actually performs the operation. We simply have to add or remove instatntiations of the single bit module.

I have chosen the modular approach as it makes it easier to reuse the code and hence is certainly better in the long run. In this approach there is still a choice between instatiating the single bit module 32 times one after the other or using a generate block to multiply the instance. I have made use of the generate block as it serves the purpose well while making the code shorter and cleaner.

Short note on generate block:

The generate block in verilog allows us to multiply module instances or perform conditional instatiation. It is a very convenient way of repeating the same operation or module instatiation multiple times.

Modules

The XOR gate primitive in verilog follows the format

```
xor g1(ans,a,b);
```

where g1 is the name of the gate primitive instance, ans is the output and a and b are the inputs.

Note: More inputs can be added to the xor gate by simple adding more input variables.

The module I have designed for the single bit xor operation is as follows

```
`timescale 1ns / 1ps

module xor1x1(
  input a,
  input b,
  output ans
  );

  xor g1(ans,a,b);

endmodule
```

The module I have designed for the 32 bit operation that uses the single bit operation is as follows

```
`timescale 1ns / 1ps

module xor32x1(
  input signed [31:0]a,
  input signed [31:0]b,
  output signed [31:0]ans
  );

  genvar i;

  generate for(i=0; i<32; i=i+1)
  begin
    xor1x1 g1(a[i],b[i],ans[i]);
  end
  endgenerate
endmodule
```

Testbench

To test the implemented module I have used the following test bench

```
`timescale 1ns / 1ps

module Xor_test;
  reg signed [31:0]a;
  reg signed [31:0]b;

  wire signed [31:0]ans;

  xor32x1 uut(
    .a(a),
    .b(b),
    .ans(ans)
  );

  initial begin
    $dumpfile("Xor_test.vcd");
    $dumpvars(0,Xor_test);
    a = 32'b0;
    b = 32'b0;

    #100;

    #20 a=32'b1011;b=32'b0100;
    #20 a=32'b1011;b=32'b1100;
    #20 a=-32'b1011;b=32'b1100;
    #20 a=32'b1001;b=32'b1001;
    #20 a=-32'd2;b=32'd13;
    #20 a=-32'd2;b=-32'd13;
    #20 a=32'b1001;b=32'b1001;
  end

  initial
    $monitor("a=%b b=%b ans=%b\n",a,b,ans);
endmodule
```

Results

On executing

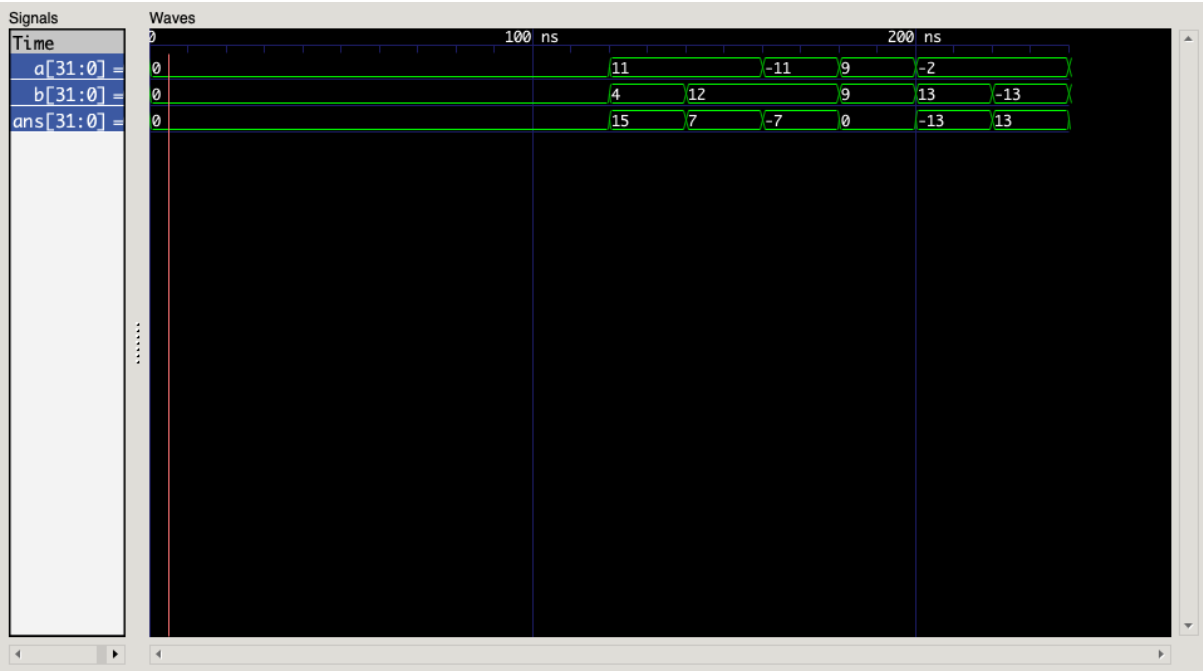
```
iverilog -o xor xor_test.v xor32x1.v xor1x1.v
vvp xor
```

we get the following terminal output

```
a=00000000000000000000000000000000 b=00000000000000000000000000000000 ans=0000000
00000000000000000000000000

a=00000000000000000000000000001011 b=00000000000000000000000000000100 ans=0000000
00000000000000000000000001111

a=00000000000000000000000000001011 b=00000000000000000000000000001100 ans=0000000
00000000000000000000000000111

a=11111111111111111111111111110101 b=00000000000000000000000000001100 ans=1111111
11111111111111111111111001

a=00000000000000000000000000001001 b=00000000000000000000000000001001 ans=0000000
00000000000000000000000000

a=11111111111111111111111111111110 b=00000000000000000000000000001101 ans=1111111
11111111111111111111110011

a=11111111111111111111111111111110 b=11111111111111111111111111110011 ans=0000000
00000000000000000000001101

a=00000000000000000000000000001001 b=00000000000000000000000000001001 ans=0000000
00000000000000000000000000
```

We can see that all results are correct for their respective input values of a and b.

The gtkwave output waveform is as follows



The test inputs and their respective outputs can be summarized as follows (in decimal)

a	b	$a \oplus b$
0	0	0
11	4	15
11	12	7
-11	12	-7
9	9	0
-2	13	-13
-2	-13	13

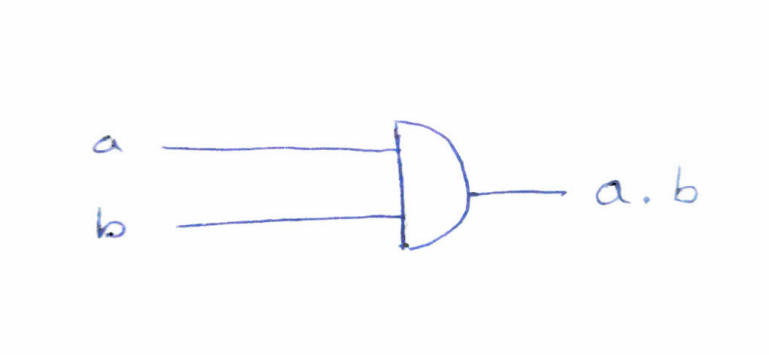
AND

Logic, Circuit and Module design

The truth table for an AND logic gate is as follows

a	b	$a.b$
0	0	0
0	1	0
1	0	0
1	1	1

The circuit for a AND logic gate is as follows



Module design

The design procedure is very similar to the above mentioned 32 bit XOR module design procedure. We make use of the AND gate primitive in verilog to perform the single bit operation. I have made use of the same modular approach where I created a module for the single bit operation and then used a generate block in the 32 bit module to instatiate the single bit module 32 times for the 32 bits of the input.

Modules

The AND gate primitive in verilog follows the format

```
and g1(ans,a,b);
```

where g1 is the name of the gate primitive instance, ans is the output and a and b are the inputs.

Note: More inputs can be added to the and gate by simple adding more input variables.

The module I have designed for the single bit and operation is as follows

```
`timescale 1ns / 1ps

module and1x1(
  input a,
  input b,
  output ans
  );

  and g1(ans,a,b);

endmodule
```

The module I have designed for the 32 bit operation that uses the single bit operation is as follows

```
`timescale 1ns / 1ps

module and32x1(
  input signed [31:0]a,
  input signed [31:0]b,
  output signed [31:0]ans
  );

  genvar i;

  generate for(i=0; i<32; i=i+1)
  begin
    and1x1 g1(a[i],b[i],ans[i]);
  end
  endgenerate
endmodule
```

Testbench

To test the implemented modules I have used the following test bench

```
`timescale 1ns / 1ps

module And_test;
  reg signed [31:0]a;
  reg signed [31:0]b;

  wire signed [31:0]ans;

  and32x1 uut(
    .a(a),
    .b(b),
    .ans(ans)
  );

  initial begin
    $dumpfile("And_test.vcd");
    $dumpvars(0,And_test);
    a = 32'b0;
    b = 32'b0;

    #100;
```

```
    #20 a=32'b1011;b=32'b0100;
    #20 a=32'b1011;b=32'b1100;
    #20 a=-32'b1011;b=32'b1100;
    #20 a=32'b1001;b=32'b1001;
    #20 a=-32'd2;b=32'd13;
    #20 a=-32'd2;b=-32'd13;
    #20 a=32'b1001;b=32'b1001;
  end

  initial
    $monitor("a=%b b=%b ans=%b\n",a,b,ans);
endmodule
```

Results

On executing

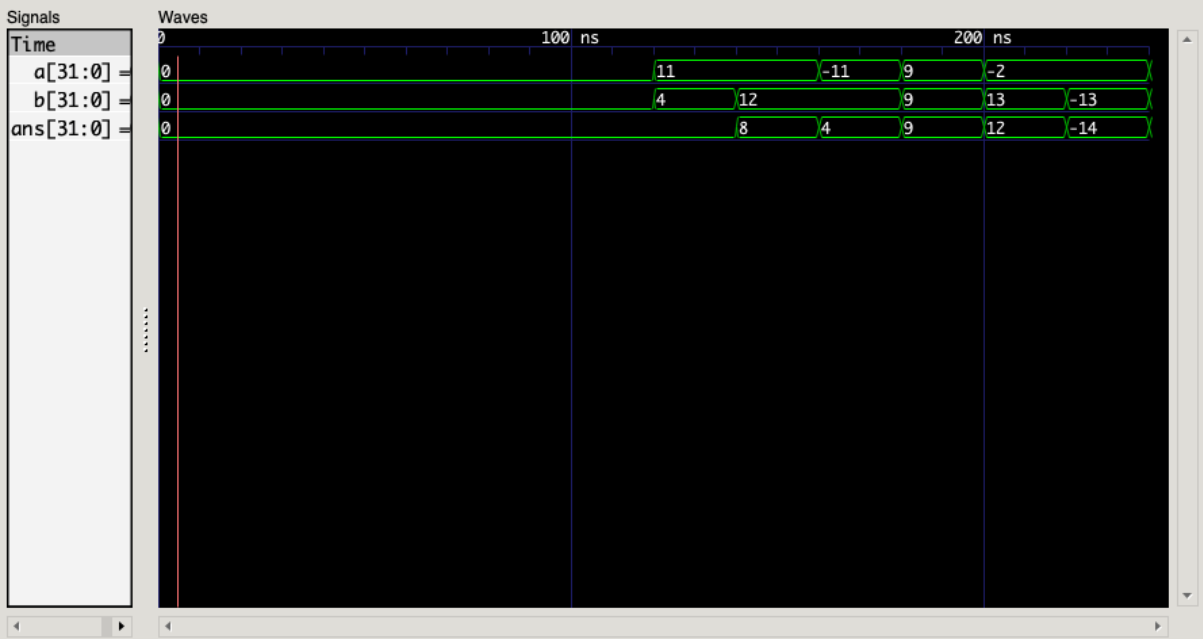
```
iverilog -o and and_test.v and32x1.v and1x1.v
vvp and
```

we get the following terminal output

```
a=00000000000000000000000000000000 b=00000000000000000000000000000000 ans=0000000
00000000000000000000000000

a=00000000000000000000000000001011 b=00000000000000000000000000000100 ans=0000000
00000000000000000000000000

a=00000000000000000000000000001011 b=00000000000000000000000000001100 ans=0000000
00000000000000000000001000

a=11111111111111111111111111110101 b=00000000000000000000000000001100 ans=0000000
00000000000000000000000100

a=00000000000000000000000000001001 b=00000000000000000000000000001001 ans=0000000
00000000000000000000001001

a=11111111111111111111111111111110 b=00000000000000000000000000001101 ans=0000000
00000000000000000000001100

a=11111111111111111111111111111110 b=11111111111111111111111111110011 ans=1111111
11111111111111111111110010

a=00000000000000000000000000001001 b=00000000000000000000000000001001 ans=0000000
00000000000000000000001001
```

We can see that all results are correct for their respective input values of a and b.

The gtkwave output waveform is as follows



The test inputs and their respective outputs can be summarized as follows (in decimal)

a	b	$a \cdot b$
0	0	0
11	4	0
11	12	8
-11	12	4
9	9	9
-2	13	12
-2	-13	-14

ADD

Logic, Circuit and Module design

We know that the truth table for an ADD function on a single bit should be as follows

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

From this we can conclude that we need to make full adders in order to perform the required operation.

From previous knowledge we know that the truth table for a full adder is of the form

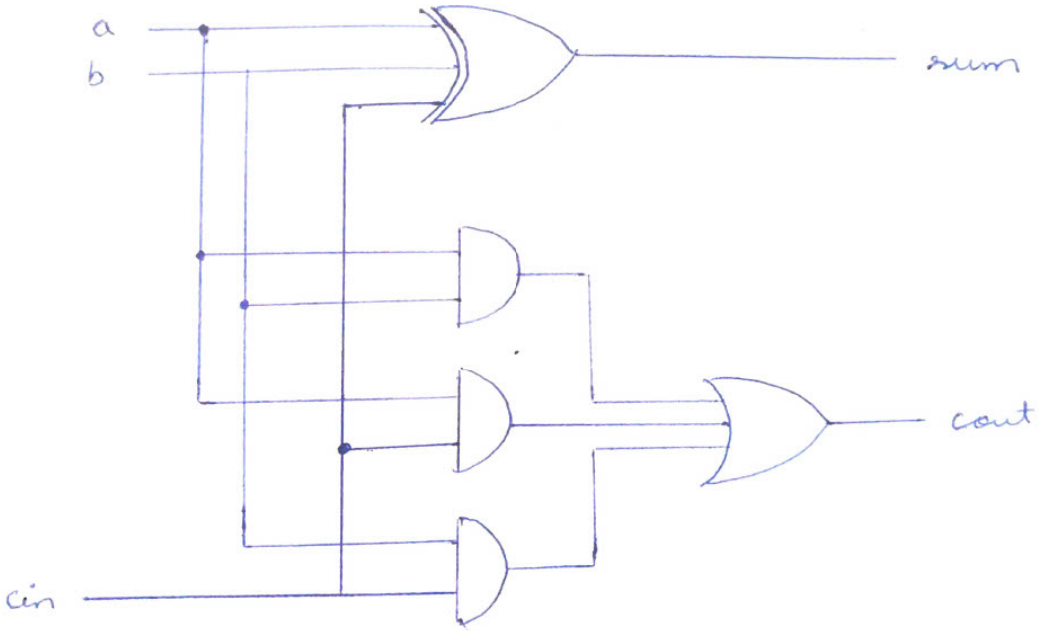
a	b	carryin	sum	$carryout$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

From the truth table we can derive the expressions for sum and carryout to be

$$sum[i] = a[i] \oplus b[i] \oplus carry[i]$$

$$carry[i+1] = a[i] \cdot b[i] + b[i] \cdot c[i] + a[i] \cdot c[i]$$

The full adder circuit is as follows



Module design

The first step in making the ADD module is the single bit full adder. We can construct the full adder as per the circuit shown above. Then we can use the single bit full adder to make the 32 bit ADD module which will use the single bit full adder on each bit of the inputs.

Note: Since we are using singed inputs we need to watch for overflow of the data values into the sign bit.

Modules

The XOR,AND and OR gate primitive in verilog follows the format

```
<gate> g1(ans,a,b);
```

where <gate> can be XOR, AND or OR, g1 is the name of the gate primitive instance, ans is the output and a and b are the inputs.

Note: More inputs can be added to the and gate by simple adding more input variables.

The module I have designed for the single bit full adder is as follows

```
`timescale 1ns / 1ps

module add1x1(
  input a,
  input b,
  input cin,
  output sum,
  output co);

  xor g1(sum,a,b,cin);
  and g2(k,a,b);
  and g3(l,a,cin);
  and g4(m,b,cin);
  or g5(co,k,l,m);

endmodule
```

The module I have designed for the 32 bit add operation that uses the single bit full adder is as follows

```
`timescale 1ns / 1ps

module add32x1(
  input signed [31:0]a,
  input signed [31:0]b,
  output signed [31:0]sum,
  output overflow);

  wire [32:0]c;
  assign c[0]=1'b0;

  genvar i;

  generate for(i=0; i<32; i=i+1)
  begin
    add1x1 g1(a[i],b[i],c[i],sum[i],c[i+1]);
  end
  endgenerate

  xor g2(overflow,c[31],c[32]);

endmodule
```

The overflow bit will be xor of the carry in and carry out of the sign bit addition because overflow ocuurs when:

1. Sum of 2 numbers with sign bits off yields a result with sign bit on
2. Sum of 2 numbers with sign bits on yields a result with sign bit off

Testbenches

To test the implemented single bit full adder I have used the following test bench

```
`timescale 1ns / 1ps

module add_test;
  reg a;
  reg b;
  reg cin;
  wire signed sum;
  wire carryout;

  add1x1 uut(
    .a(a),
    .b(b),
    .cin(cin),
    .sum(sum),
    .co(co)
  );

  initial begin
    $dumpfile("add_test.vcd");
    $dumpvars(0,add_test);
    a = 1'b0;
    b = 1'b0;

    #100;

    #20 a=1'd1;b=1'd0;cin=1'd0;
    #20 a=1'd1;b=1'd1;cin=1'd0;
    #20 a=1'd1;b=1'd0;cin=1'd1;
    #20 a=1'd0;b=1'd0;cin=1'd0;
  end

  initial begin
    $monitor("a=%b b=%b sum=%b carryout=%d\n",a,b,sum,co);
  end
endmodule
```

This testbench was used to test and veify the functioning of the full adder.

To test the implemented 32 bit add module I have used the following test bench

```
`timescale 1ns / 1ps

module add_test;
  reg signed [31:0]a;
  reg signed [31:0]b;

  wire signed [31:0]sum;
  wire overflow;

  add32x1 uut(
    .a(a),
    .b(b),
    .sum(sum),
    .overflow(overflow)
  );

  initial begin
    $dumpfile("add_test.vcd");
    $dumpvars(0,add_test);
    a = 32'b0;
    b = 32'b0;

    #100;

    #20 a=32'd2147483647;b=32'd1;
    #20 a=-32'd2147483648;b=-32'd1;
    #20 a=32'd23;b=-32'd0;
    #20 a=32'b1001;b=32'b1001;
    #20 a=32'b1001;b=-32'b1001;
    #20 a=-32'd2;b=32'd13;
    #20 a=-32'd2;b=-32'd13;
    #20 a=32'd2;b=-32'd13;
    #20 a=32'd0;b=32'd0;
  end

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

Results

On executing

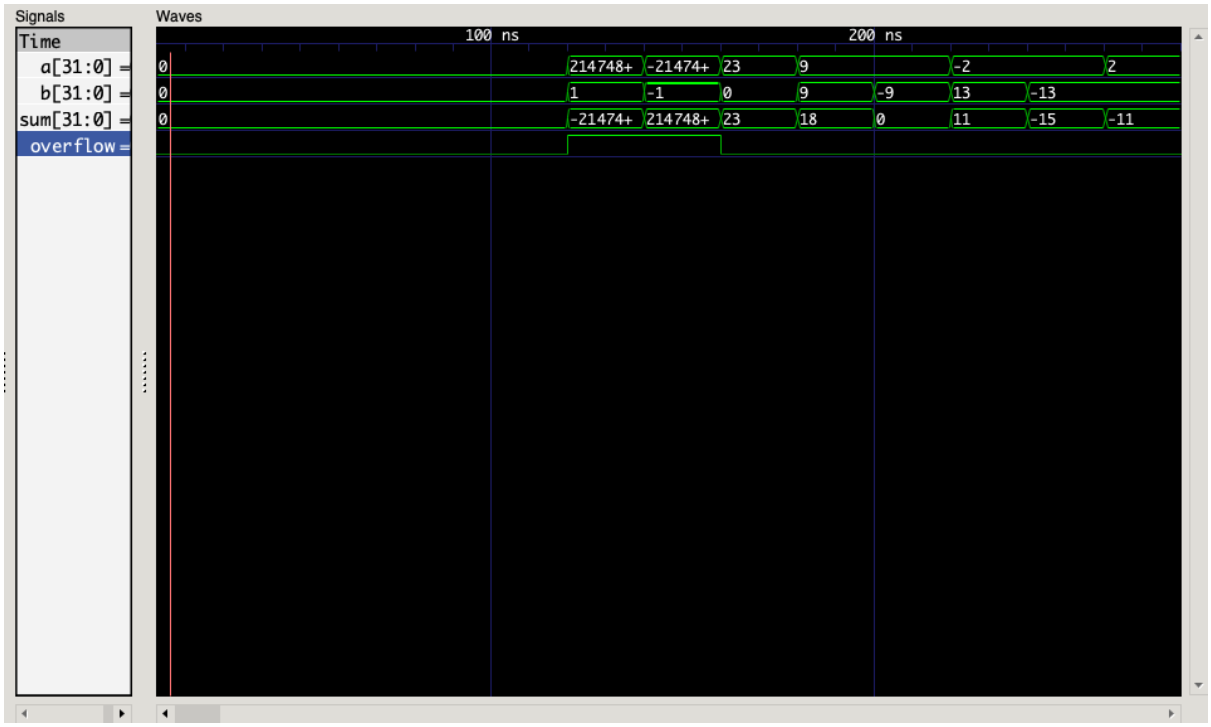
```
iverilog -o add add_test.v add1x1.v add32x1.v
vvp add
```

we get the following terminal output

```
a=          0 b=          0 sum=          0 overflow=0
a= 2147483647 b=          1 sum=-2147483648 overflow=1
a=-2147483648 b=         -1 sum= 2147483647 overflow=1
a=         23 b=          0 sum=         23 overflow=0
a=          9 b=          9 sum=         18 overflow=0
a=          9 b=         -9 sum=          0 overflow=0
a=         -2 b=         13 sum=         11 overflow=0
a=         -2 b=        -13 sum=        -15 overflow=0
a=          2 b=        -13 sum=        -11 overflow=0
a=          0 b=          0 sum=          0 overflow=0
```

We can see that all result values are correct for the respective input values of a and b. We can also note that the overflow bit is working correctly as it is detecting both negative and positive overflows and not giving any false alarms.

The gtkwave output waveform is as follows



The inputs and their respective outputs can be summarized as follows (in decimal)

a	b	$a+b$	overflow
0	0	0	0
2147483647	1	-2147483648	1
-2147483648	-1	2147483647	1
23	0	23	0
9	9	18	0
9	-9	0	0
-2	13	11	0
-2	-13	-15	0
2	-13	-11	0

SUB

Logic, Circuit and Module design

The truth table for the SUB operation will be as follows

a	b	$a-b$	$borrow$
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

We can see that rather than implementing a subtractor circuit it will be easier to implement this in the form of an adder since we already have the adder modules ready.

$$y = a - b$$
$$\implies y = a + (-b)$$

To convert this subtraction into addition we need to take the 2s complement of b and then add it to a.

We know that the 2s complement is calculated by inverting all the bits and adding 1. In order to do this we need a 32 bit inverter circuit.

Module design

We have to take the 2s complement of b in order to be able to simply add them. The 32 bit inverter can be constructed in a way similar to the rest of our 32 bit gate modules. We make use of the NOT gate primitive in verilog to perform the single bit inversion. Following the moduar approach I used a generate block in the 32 bit module to instatiate the single bit module 32 times for the 32 bits of the input. This gives us a 32 bit inverter. To find 2s complement we simple add 1 to the result from the inverter. Then we can add a and the 2s complement of b which will give us the result of the subtraction of b from a.

Modules

The XOR,AND and OR gate primitive in verilog follows the format

```
<gate> g1(ans,a,b);
```

where <gate> can be XOR, AND or OR, g1 is the name of the gate primitive instance, ans is the output and a and b are the inputs.

Note: More inputs can be added to the and gate by simple adding more input variables.

The NOT gate primitive in verilog follows the format

```
not g1(ans,b);
```

where g1 is the name of the gate primitive instance, ans is the output and b is the input.

The module I have designed for the single bit inverter is as follows

```
`timescale 1ns / 1ps

module not1x1(
  input a,
  output ans
  );

  not g1(ans,a);

endmodule
```

The module I have designed for the 32 bit inverter that uses the single bit inverter module is as follows

```
`timescale 1ns / 1ps

module not32x1(
  input signed [31:0]a,
  output signed [31:0]ans
```

```
  );

  genvar i;

  generate for(i=0; i<32; i=i+1)
  begin
    not1x1 g1(a[i],ans[i]);
  end
  endgenerate

endmodule
```

And using this 32 bit inverter module and the 32 bit adder module I have designed the subtractor module as follows

```
`timescale 1ns / 1ps

module sub32x1(
  input signed [31:0]a,
  input signed [31:0]b,
  output signed [31:0]ans,
  output overflow);

  wire [31:0]nb;
  not32x1 g1(b,nb);

  wire [31:0]l;
  assign l=32'b1;

  wire [31:0]bcomp;
  add32x1 g2(nb,l,bcomp,c);

  add32x1 g3(a,bcomp,ans,overflow);

endmodule
```

Testbench

To test the inverter modules I have used the following test bench

```
`timescale 1ns / 1ps

module not_test;
  reg signed [31:0]a;
  reg signed [31:0]b;

  wire signed [31:0]ans;

  not32x1 uut(
```

```
    .a(a),
    .ans(ans)
  );

  initial begin
    $dumpfile("not_test.vcd");
    $dumpvars(0,not_test);
    a = 32'b0;

    #100;

    #20 a=32'b1011;
    #20 a=32'b1011;
    #20 a=-32'b1011;
    #20 a=32'b1001;
    #20 a=-32'd2;
    #20 a=-32'd2;
    #20 a=32'b1001;
  end

  initial begin
    $monitor("a=%b b=%b ans=%b\n",a,b,ans);
  end
endmodule
```

This testbench was used to verify the functioning of the 32 bit inverter module.

To test the subtractor module I have used the following test bench

```
`timescale 1ns / 1ps

module sub_test;
  reg signed [31:0]a;
  reg signed [31:0]b;
  wire signed [31:0]ans;
  wire overflow;

  sub32x1 uut(
    .a(a),
    .b(b),
    .ans(ans),
    .overflow(overflow)
  );

  initial begin
    $dumpfile("sub_test.vcd");
    $dumpvars(0,sub_test);
    a = 32'b0;
    b = 32'b0;

    #100;
```

```
    #20 a=32'd2147483647;b=-32'd1;
    #20 a=-32'd2147483648;b=32'd1;
    #20 a=32'd23;b=-32'd0;
    #20 a=32'b1001;b=32'b1001;
    #20 a=32'b1001;b=-32'b1001;
    #20 a=-32'd2;b=32'd13;
    #20 a=-32'd2;b=-32'd13;
    #20 a=32'd2;b=-32'd13;
    #20 a=32'd0;b=32'd0;
  end

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

Results

On executing

```
iverilog -o sub sub_test.v sub32x1.v not/not1x1.v not/not32x1.v ../Add/add32x1.v
../Add/add1x1.vvvp
vvp sub
```

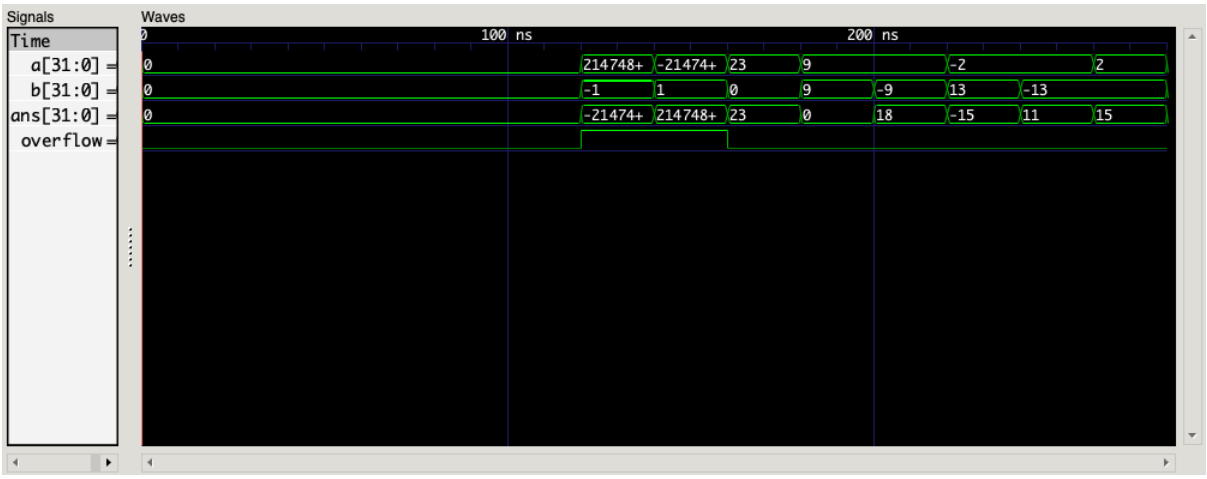
we get the following terminal output

```
a=          0 b=          0 ans=          0 overflow=0

a= 2147483647 b=         -1 ans=-2147483648 overflow=1

a=-2147483648 b=          1 ans= 2147483647 overflow=1

a=         23 b=          0 ans=         23 overflow=0

a=          9 b=          9 ans=          0 overflow=0

a=          9 b=         -9 ans=         18 overflow=0

a=         -2 b=         13 ans=        -15 overflow=0

a=         -2 b=        -13 ans=         11 overflow=0

a=          2 b=        -13 ans=         15 overflow=0

a=          0 b=          0 ans=          0 overflow=0
```

We can see that all result values are correct for the respective input values of a and b. We can also note that the overflow bit is working correctly as it is

detecting both negative and positive overflows and not giving any false alarms.

The gtkwave output waveform is as follows



The inputs and their respective outputs can be summarized as follows (in decimal)

a	b	$a-b$	overflow
0	0	0	0
2147483647	1	-2147483648	1
-2147483648	-1	2147483647	1
23	0	23	0
9	9	0	0
9	-9	18	0
-2	13	-15	0
-2	-13	11	0
2	-13	15	0

ALU

Logic, Circuit and Module design

The final ALU wrapper should be able to call each of the modules implemented above based on the control input. The ALU unit should take as input the control signal and two 32-bit inputs and return the 32-bit output corresponding to the control input chosen.

The control inputs and their respective functionalities are as follows

Control Input		Functionality
2'b00	0	ADD
2'b01	1	SUB
2'b10	2	AND
2'b11	3	XOR

Module design

The ALU wrapper simply consists of switching statements or case statements to select the operation output to be returned as the ALU output based on the control input.

Module

The module I have designed for the ALU wrapper is as follows

```
`timescale 1ns / 1ps

`include "../Add/add32x1.v"
`include "../Add/add1x1.v"
`include "../Sub/sub32x1.v"
`include "../Sub/not/not1x1.v"
`include "../Sub/not/not32x1.v"
`include "../Xor/xor32x1.v"
`include "../Xor/xor1x1.v"
`include "../And/and32x1.v"
`include "../And/and1x1.v"

module alu(
  input [1:0]control,
  input signed [31:0]a,
```

```
  input signed [31:0]b,
  output signed [31:0]ans,
  output overflow
  );

  wire signed [31:0]ans1;
  wire signed [31:0]ans2;
  wire signed [31:0]ans3;
  wire signed [31:0]ans4;
  reg signed [31:0]ansfinal;
  reg overflowfinal;

  add32x1 g1(a,b,ans1,overflow1);
  sub32x1 g2(a,b,ans2,overflow2);
  and32x1 g3(a,b,ans3);
  xor32x1 g4(a,b,ans4);
  always @(*)
  begin
    case(control)
      2'b00:begin
          ansfinal=ans1;
          overflowfinal=overflow1;
        end
      2'b01:begin
          ansfinal=ans2;
          overflowfinal=overflow2;
        end
      2'b10:begin
          ansfinal=ans3;
          overflowfinal=1'b0;
        end
      2'b11:begin
          ansfinal=ans4;
          overflowfinal=1'b0;
        end
    endcase
  end

  assign ans= ansfinal;
  assign overflow= overflowfinal;
endmodule
```

Testbench

To test the ALU I have used the following test bench

```
`timescale 1ns / 1ps

module Alu_test;
  reg [1:0]control;
  reg signed [31:0]a;
  reg signed [31:0]b;

  wire signed [31:0]ans;
```

```
  wire overflow;

  alu uut(
    .control(control),
    .a(a),
    .b(b),
    .ans(ans),
    .overflow(overflow)
  );

  initial begin
    $dumpfile("Alu_test.vcd");
    $dumpvars(0,Alu_test);
    control=2'b00;
    a = 32'b0;
    b = 32'b0;

    #100;

    #20 control=2'b00;a=32'b1011;b=32'b0100;
    #20 control=2'b01;a=32'b1011;b=32'b0100;
    #20 control=2'b10;a=32'b1011;b=32'b0100;
    #20 control=2'b11;a=32'b1011;b=32'b0100;
    #20 control=2'b00;a=-32'b1011;b=32'b0100;
    #20 control=2'b01;a=-32'b1011;b=32'b0100;
    #20 control=2'b10;a=-32'b1011;b=32'b0100;
    #20 control=2'b11;a=-32'b1011;b=32'b0100;
    #20 control=2'b00;a=32'b1011;b=-32'b0100;
    #20 control=2'b01;a=32'b1011;b=-32'b0100;
    #20 control=2'b10;a=32'b1011;b=-32'b0100;
    #20 control=2'b11;a=32'b1011;b=-32'b0100;
    #20 control=2'b00;a=-32'b1011;b=-32'b0100;
    #20 control=2'b01;a=-32'b1011;b=-32'b0100;
    #20 control=2'b10;a=-32'b1011;b=-32'b0100;
    #20 control=2'b11;a=-32'b1011;b=-32'b0100;
    #20 control=2'b00;a=32'd2147483647;b=32'd1;
    #20 control=2'b01;a=32'd2147483647;b=-32'd1;
    #20 control=2'b00;a=32'b0;b=32'b0;
  end

  initial
    $monitor("control=%b a=%b b=%b ans=%b overflow=%b\n",control,a,b,ans,overflow
);
endmodule
```

Results

On executing

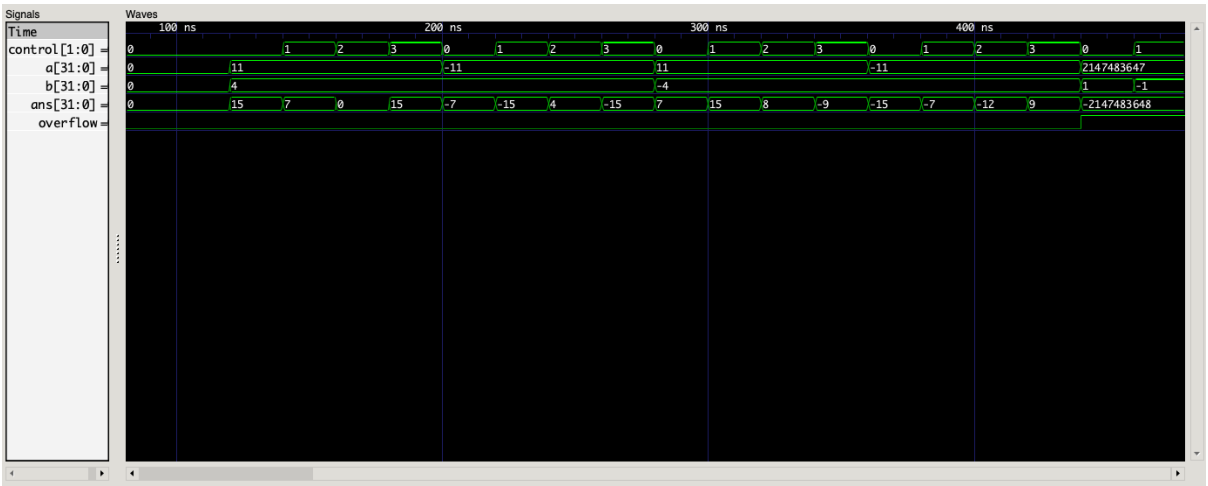
```
iverilog -o alu alu_test.v alu.v
vvp alu
```

we get the following terminal output

```
control=00 a=00000000000000000000000000000000 b=00000000000000000000000000000000
 ans=00000000000000000000000000000000 overflow=0

control=00 a=00000000000000000000000000001011 b=00000000000000000000000000000100
 ans=00000000000000000000000000001111 overflow=0

control=01 a=00000000000000000000000000001011 b=00000000000000000000000000000100
 ans=00000000000000000000000000000111 overflow=0

control=10 a=00000000000000000000000000001011 b=00000000000000000000000000000100
 ans=00000000000000000000000000000000 overflow=0

control=11 a=00000000000000000000000000001011 b=00000000000000000000000000000100
 ans=00000000000000000000000000001111 overflow=0

control=00 a=11111111111111111111111111110101 b=00000000000000000000000000000100
 ans=11111111111111111111111111111001 overflow=0

control=01 a=11111111111111111111111111110101 b=00000000000000000000000000000100
 ans=11111111111111111111111111110001 overflow=0

control=10 a=11111111111111111111111111110101 b=00000000000000000000000000000100
 ans=00000000000000000000000000000100 overflow=0

control=11 a=11111111111111111111111111110101 b=00000000000000000000000000000100
 ans=11111111111111111111111111110001 overflow=0

control=00 a=00000000000000000000000000001011 b=11111111111111111111111111111100
 ans=00000000000000000000000000000111 overflow=0

control=01 a=00000000000000000000000000001011 b=11111111111111111111111111111100
 ans=00000000000000000000000000001111 overflow=0

control=10 a=00000000000000000000000000001011 b=11111111111111111111111111111100
 ans=00000000000000000000000000001000 overflow=0

control=11 a=00000000000000000000000000001011 b=11111111111111111111111111111100
```

```
 ans=11111111111111111111111111110111 overflow=0

control=00 a=11111111111111111111111111110101 b=11111111111111111111111111111100
 ans=11111111111111111111111111110001 overflow=0

control=01 a=11111111111111111111111111110101 b=11111111111111111111111111111100
 ans=11111111111111111111111111111001 overflow=0

control=10 a=11111111111111111111111111110101 b=11111111111111111111111111111100
 ans=11111111111111111111111111110100 overflow=0

control=11 a=11111111111111111111111111110101 b=11111111111111111111111111111100
 ans=00000000000000000000000000001001 overflow=0

control=00 a=01111111111111111111111111111111 b=00000000000000000000000000000001
 ans=10000000000000000000000000000000 overflow=1

control=01 a=01111111111111111111111111111111 b=11111111111111111111111111111111
 ans=10000000000000000000000000000000 overflow=1

control=00 a=00000000000000000000000000000000 b=00000000000000000000000000000000
 ans=00000000000000000000000000000000 overflow=0
```

The gtkwave output waveform is as follows



The inputs and their respective outputs can be summarized as follows (in decimal)

Control	a	b	$a+b$	overflow
0	0	0	0	0
0	11	4	15	0
1	11	4	7	0
2	11	4	0	0
3	11	4	15	0
0	-11	4	-7	0
1	-11	4	-15	0
2	-11	4	4	0
3	-11	4	-15	0
0	11	-4	7	0
1	11	-4	15	0
2	11	-4	8	0
3	11	-4	-9	0
0	-11	-4	-15	0
1	-11	-4	-7	0
2	-11	-4	-12	0
3	-11	-4	9	0
0	2147483647	1	-2147483648	1
1	2147483647	-1	-2147483648	1

We can see that all result values are correct for the respective input values of a and b. We can also note that the overflow bit is working correctly and not giving any false alarms.

Hence, I have successfully created an ALU with the required functionalities and verified the working of the ALU as well as the individual modules for each functionality.