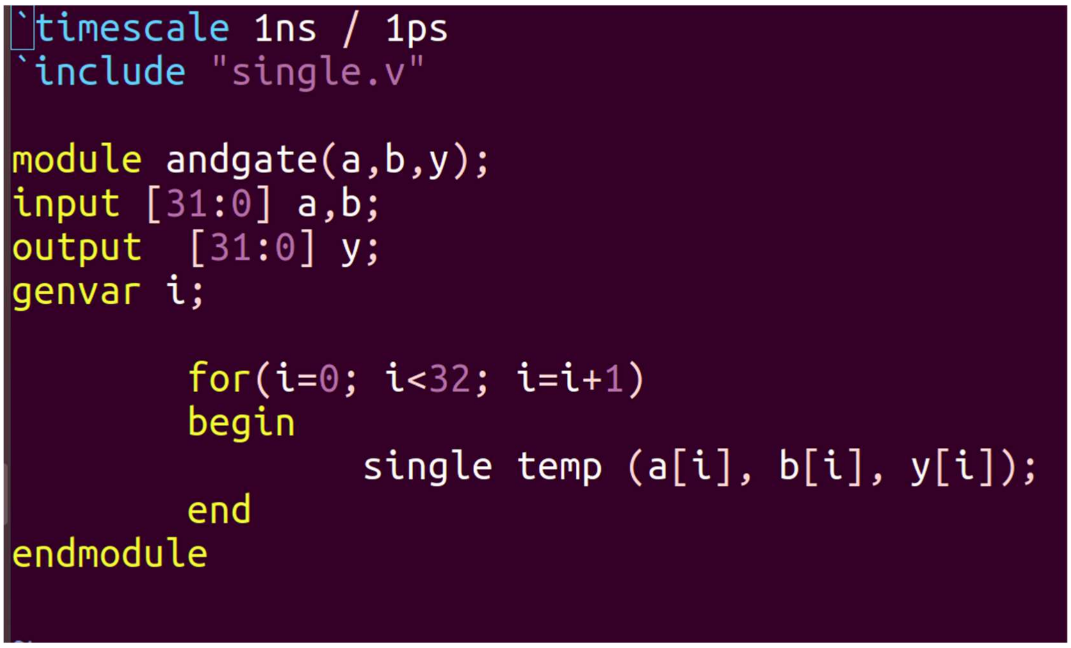
# IPA Assignment-1

## AND32:

We are given two 32-bit numbers a and b. We are expected to do bitwise AND operation and produce the output y.

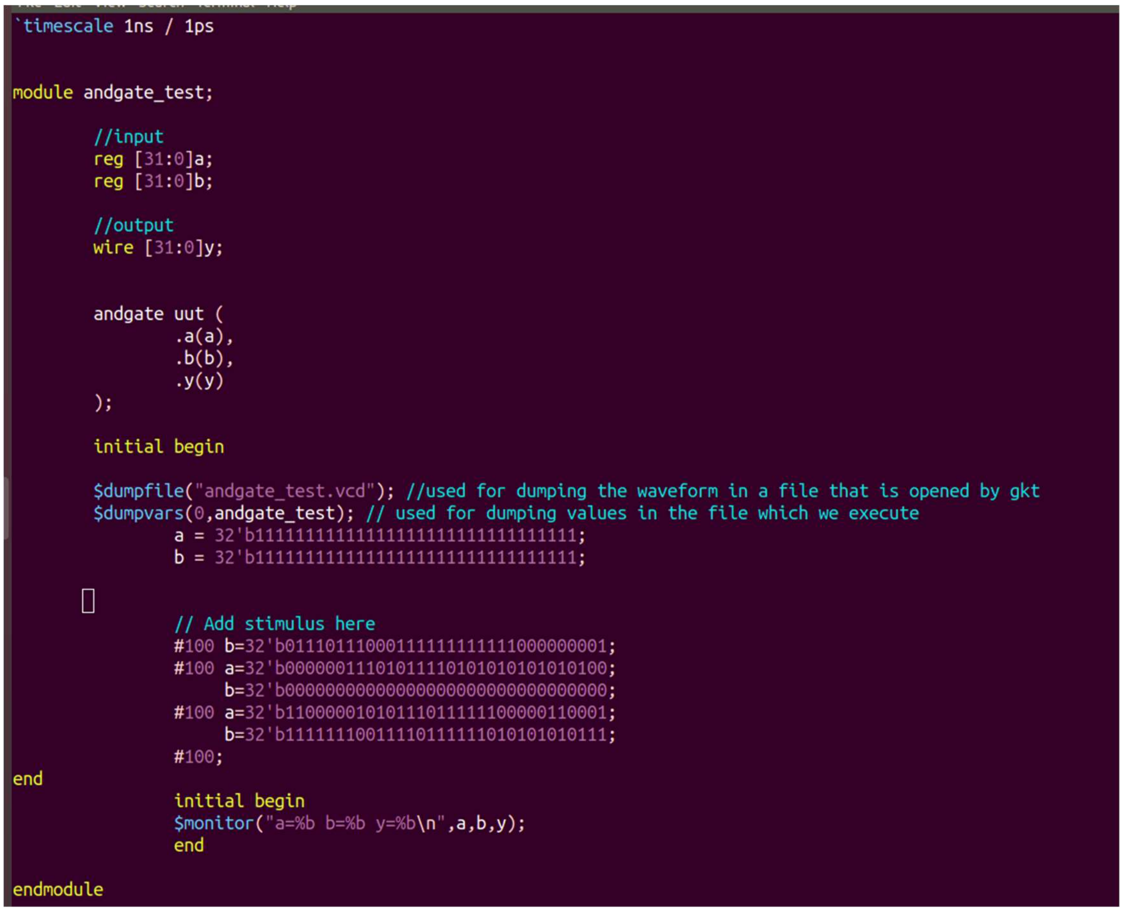
## Code:



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

module andgate(a,b,y);
input [31:0] a,b;
output  [31:0] y;
genvar i;

        for(i=0; i<32; i=i+1)
        begin
                single temp (a[i], b[i], y[i]);
        end
endmodule
```

- In the code, we initially define out timescale to be 1ns/1ps. The time delays are measure in nanosecond and the precision is 1picosecond.
- We included single.v which contains a module which does AND operation between two bits.
- We defined genvar i; which is used to define the loop variable.
- We are asked to do the bitwise AND of 32 bits. So we run a for loop where we call the module single (it performs a[i]&b[i] to give y[i]).
- After running the loop, we finally get y in which each bit is the AND of a and b.

## Test bench:

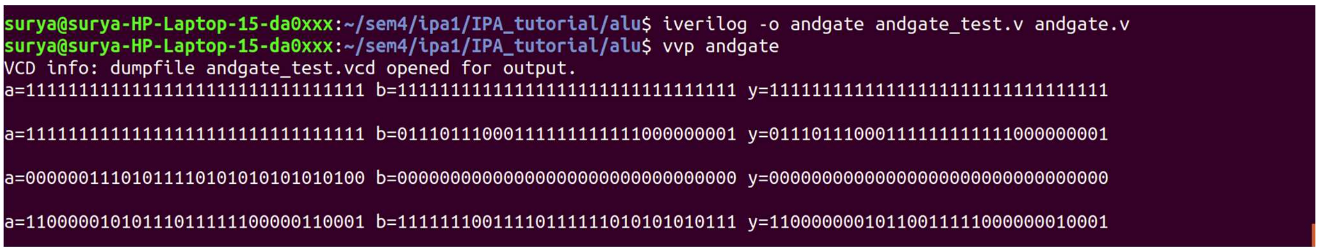


```
`timescale 1ns / 1ps


module andgate_test;

        //input
        reg [31:0]a;
        reg [31:0]b;

        //output
        wire [31:0]y;


        andgate uut (
                .a(a),
                .b(b),
                .y(y)
        );

        initial begin

        $dumpfile("andgate_test.vcd"); //used for dumping the waveform in a file that is opened by gkt
        $dumpvars(0,andgate_test); // used for dumping values in the file which we execute
                a = 32'b11111111111111111111111111111111;
                b = 32'b11111111111111111111111111111111;


                // Add stimulus here
                #100 b=32'b01110111000111111111111000000001;
                #100 a=32'b00000011101011110101010101010100;
                     b=32'b00000000000000000000000000000000;
                #100 a=32'b11000001010111011111100000110001;
                     b=32'b11111110011110111111010101010111;
                #100;
end
                initial begin
                $monitor("a=%b b=%b y=%b\n",a,b,y);
                end

endmodule
```

- Writing the test which is responsible for testing the module that we have written is fairly simple.
- We define what the input and outputs are initially. Later, we instantiate the unit under test.
- Then we dump the waveforms into a file so that we can open that file in gtkwave later.
- We also use dumpvars to see that all the values of the instantiated variables are dumped into a file which we later use for executing.
- Then we actually give the values for a and b, then change them with appropriate delays. Later we print all of them in the monitor.

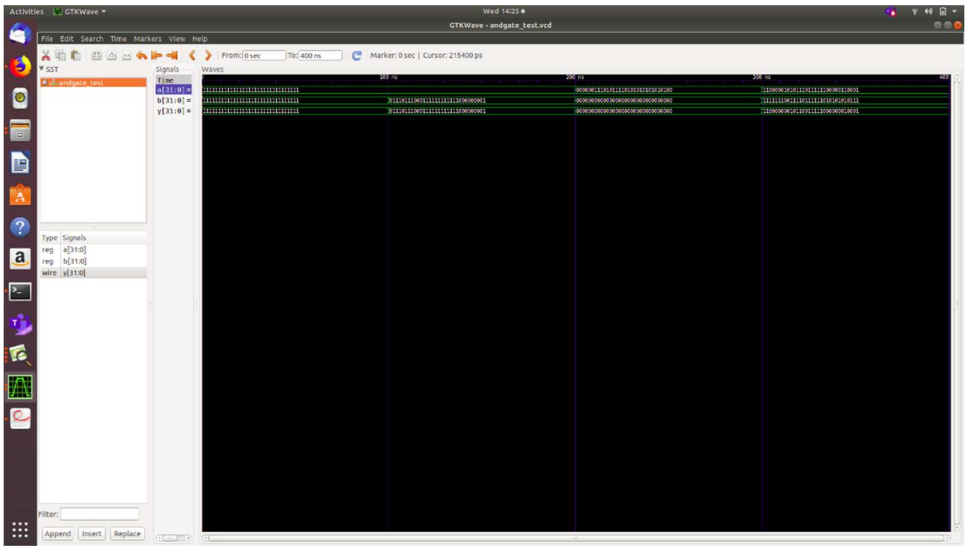
## Results:

Terminal output:

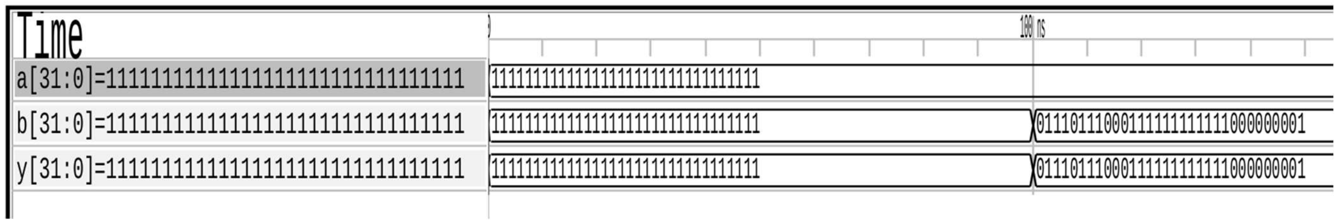


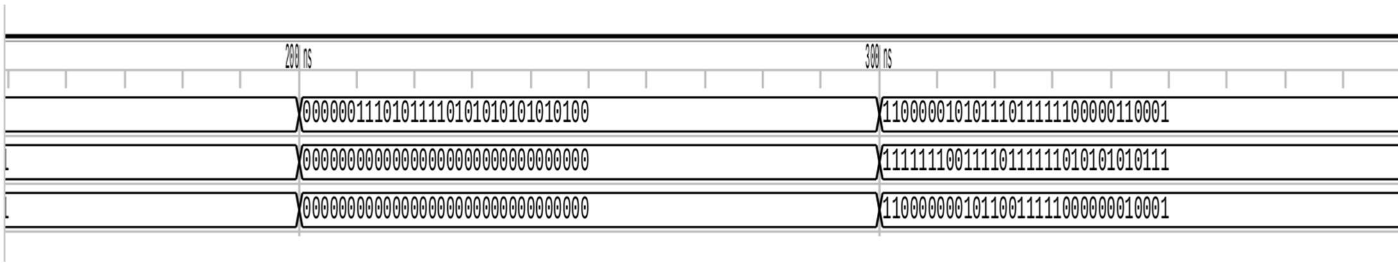
```
surya@surya-HP-Laptop-15-da0xxx:~/sem4/ipa1/IPA_tutorial/alu$ iverilog -o andgate andgate_test.v andgate.v
surya@surya-HP-Laptop-15-da0xxx:~/sem4/ipa1/IPA_tutorial/alu$ vvp andgate
VCD info: dumpfile andgate_test.vcd opened for output.
a=11111111111111111111111111111111 b=11111111111111111111111111111111 y=11111111111111111111111111111111

a=11111111111111111111111111111111 b=01110111000111111111111000000001 y=01110111000111111111111000000001

a=00000011101011110101010101010100 b=00000000000000000000000000000000 y=00000000000000000000000000000000

a=11000001010111011111100000110001 b=11111110011110111111010101010111 y=11000000010110011111000000010001
```

GTKwave output:



Due to lack of visibility in the above file, a pdf version saved is shown below.

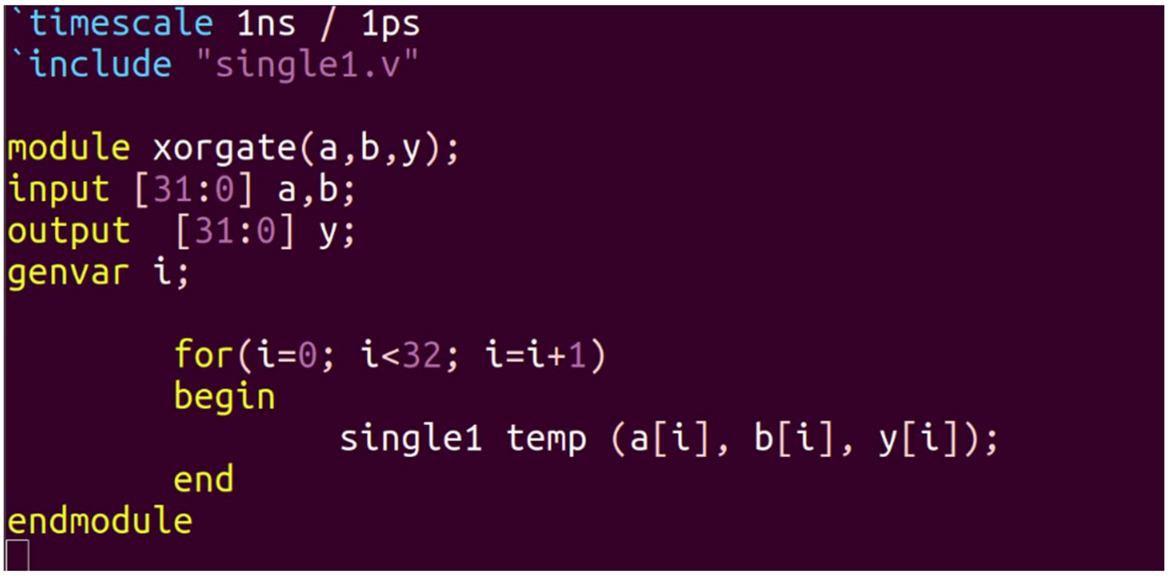




## XOR32:

We are given two 32-bit numbers a and b. We are expected to do bitwise XOR operation and produce the output y.

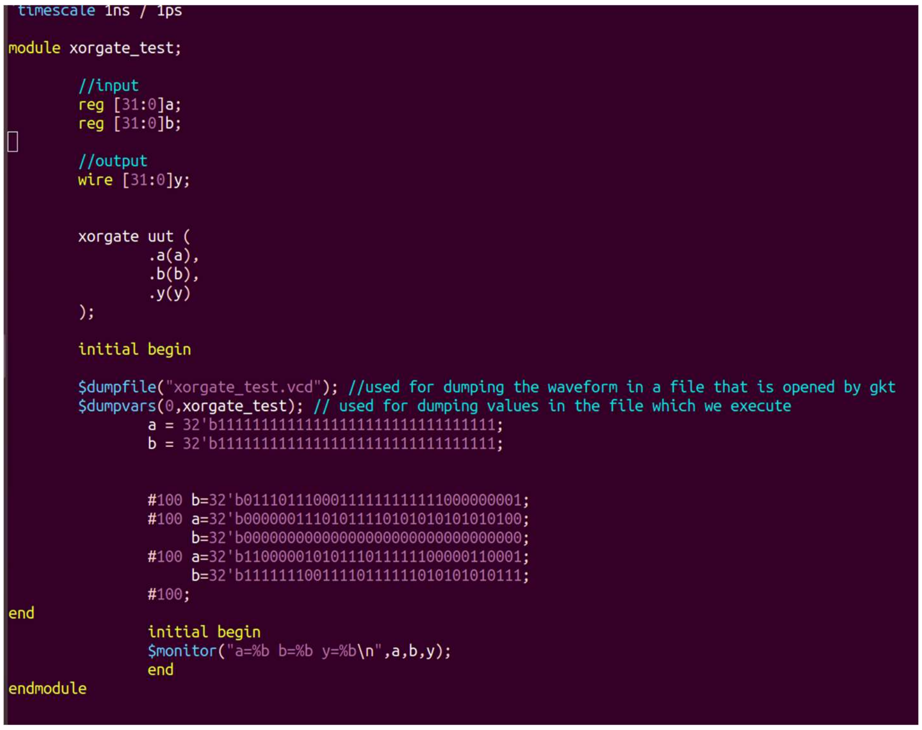
## Code:



```
`timescale 1ns / 1ps
`include "single1.v"

module xorgate(a,b,y);
input [31:0] a,b;
output  [31:0] y;
genvar i;

        for(i=0; i<32; i=i+1)
        begin
                single1 temp (a[i], b[i], y[i]);
        end
endmodule
```

- In the code, we initially define out timescale to be 1ns/1ps. The time delays are measure in nanosecond and the precision is upto 1picosecond.
- We included single1.v which contains a module which does XOR operation between two bits.
- We defined genvar i; which is used to define the loop variable.
- We are asked to do the bitwise XOR of 32 bits. So we run a for loop where we call the module single1(it performs a[i]^b[i] to give y[i]).
- After running the loop, we finally get y in which each bit is the XOR of a and b.

## Test bench:

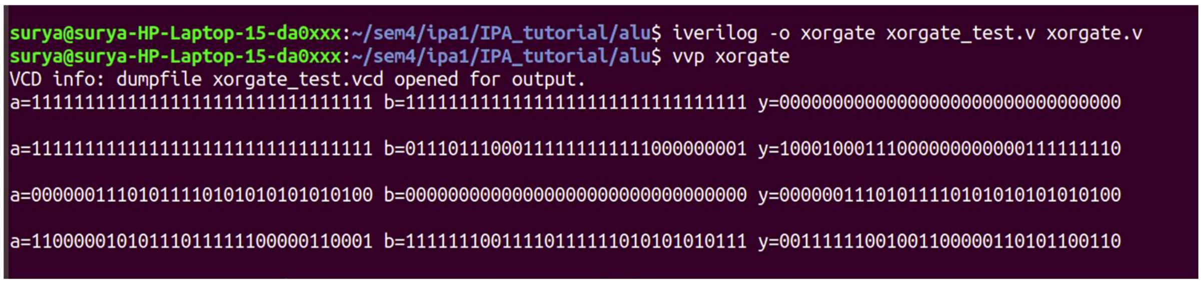


```
 timescale 1ns / 1ps

module xorgate_test;

        //input
        reg [31:0]a;
        reg [31:0]b;


        //output
        wire [31:0]y;


        xorgate uut (
                .a(a),
                .b(b),
                .y(y)
        );

        initial begin

        $dumpfile("xorgate_test.vcd"); //used for dumping the waveform in a file that is opened by gkt
        $dumpvars(0,xorgate_test); // used for dumping values in the file which we execute
                a = 32'b11111111111111111111111111111111;
                b = 32'b11111111111111111111111111111111;


                #100 b=32'b01110111000111111111111000000001;
                #100 a=32'b00000011101011110101010101010100;
                     b=32'b00000000000000000000000000000000;
                #100 a=32'b11000001010111011111100000110001;
                     b=32'b11111110011110111111010101010111;
                #100;
end
                initial begin
                $monitor("a=%b b=%b y=%b\n",a,b,y);
                end
endmodule
```

- Writing the  tesbench which is responsible for testing the module that we have written is fairly simple.
- We define what the input and outputs are initially. Later, we instantiate the unit under test.
- Then we dump the waveforms into a file so that we can open that file in gtkwave later.
- We also use dumpvars to see that all the values of the instantiated variables are dumped into a file which we later use for executing.
- Then we actually give the values for a and b, then change them with appropriate delays. Later we print all of them in the monitor.

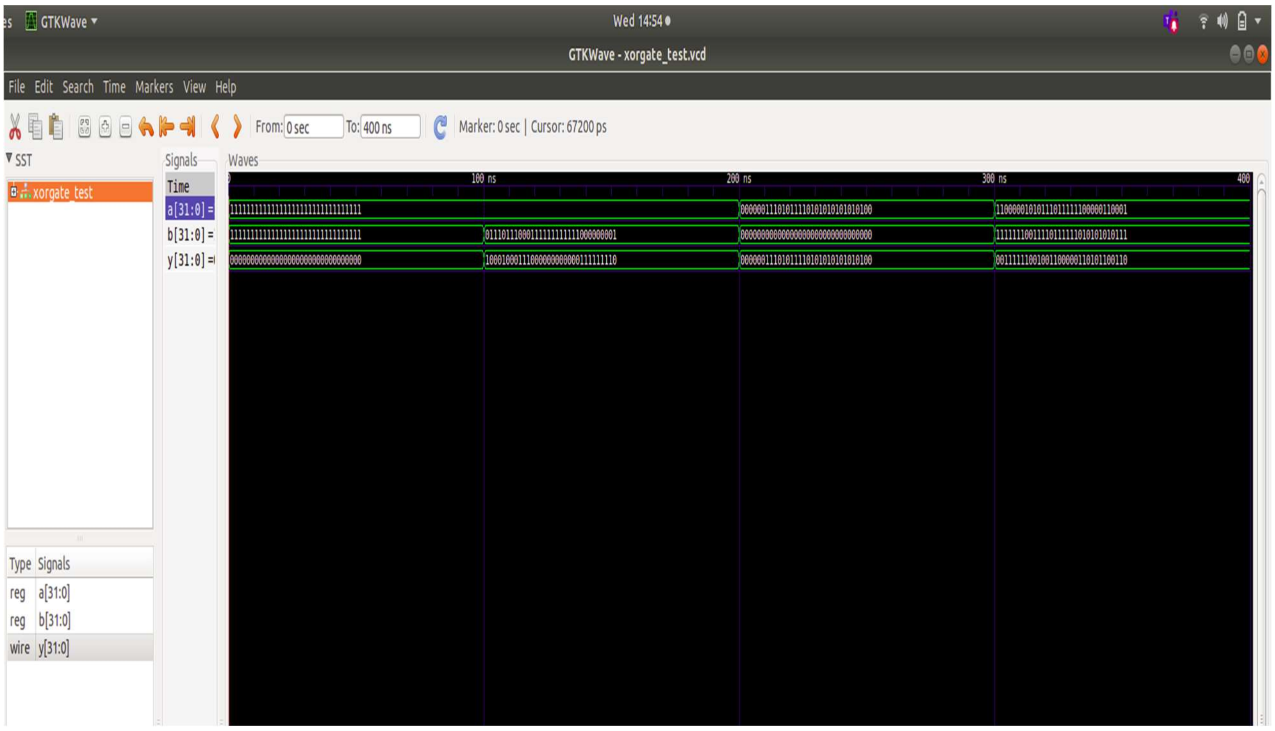
## Results:

Terminal output:



```
surya@surya-HP-Laptop-15-da0xxx:~/sem4/ipa1/IPA_tutorial/alu$ iverilog -o xorgate xorgate_test.v xorgate.v
surya@surya-HP-Laptop-15-da0xxx:~/sem4/ipa1/IPA_tutorial/alu$ vvp xorgate
VCD info: dumpfile xorgate_test.vcd opened for output.
a=11111111111111111111111111111111 b=11111111111111111111111111111111 y=00000000000000000000000000000000

a=11111111111111111111111111111111 b=01110111000111111111111000000001 y=10001000111000000000000111111110

a=00000011101011110101010101010100 b=00000000000000000000000000000000 y=00000011101011110101010101010100

a=11000001010111011111100000110001 b=11111110011110111111010101010111 y=00111111001001100000110101100110
```

GTKwave output:

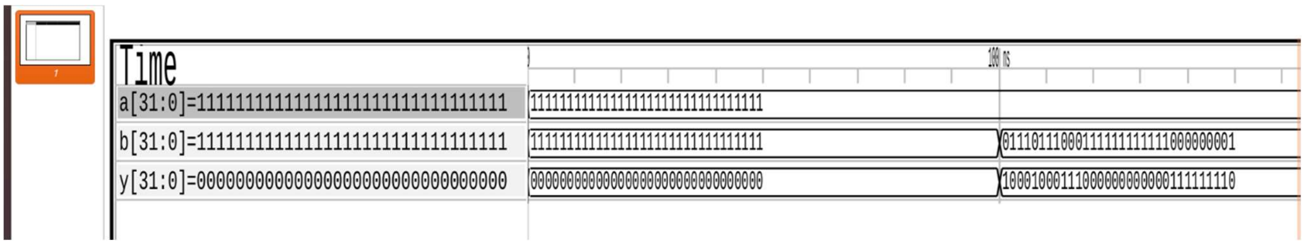


GTKWave - xorgate_test.vcd

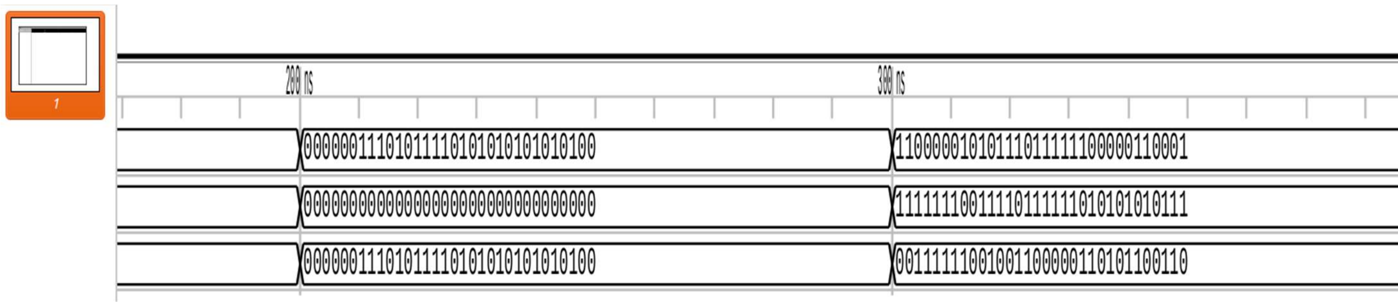
From: 0 sec To: 400 ns Marker: 0 sec | Cursor: 67200 ps

```
SST
xorgate_test

Type  Signals
reg   a[31:0]
reg   b[31:0]
wire  y[31:0]

Signals    Waves
Time       0                                 100 ns                            200 ns                            300 ns                            400
a[31:0] =  11111111111111111111111111111111                                    00000011101011110101010101010100  11000001010111011111100000110001
b[31:0] =  11111111111111111111111111111111  01110111000111111111111000000001  00000000000000000000000000000000  11111110011110111111010101010111
y[31:0] =  00000000000000000000000000000000  10001000111000000000000111111110  00000011101011110101010101010100  00111111001001100000110101100110
```

Due to lack of visibility in the above file, a pdf version saved is shown below.



```
Time                                                                    100 ns
a[31:0]=11111111111111111111111111111111  11111111111111111111111111111111
b[31:0]=11111111111111111111111111111111  11111111111111111111111111111111  01110111000111111111111000000001
y[31:0]=00000000000000000000000000000000  00000000000000000000000000000000  10001000111000000000000111111110
```

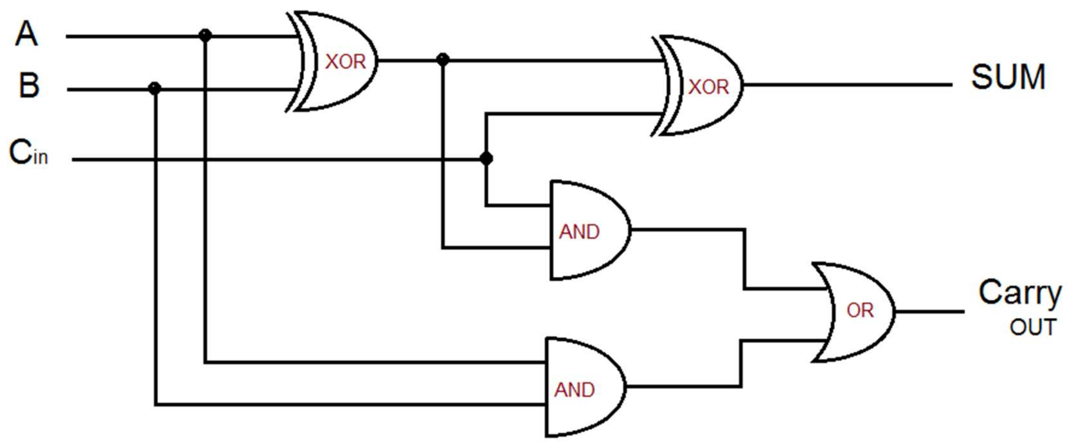


```
200 ns                              300 ns
00000011101011110101010101010100    11000001010111011111100000110001
00000000000000000000000000000000    11111110011110111111010101010111
00000011101011110101010101010100    00111111001001100000110101100110
```

## ADD32:

We are given two 32-bit signed numbers a and b. We are expected to add the numbers and produce the output y.

## Theory behind the adder:



The above figure depicts a full adder. The basic idea is that when we add 3 bits, we get a sum and a carry bit.

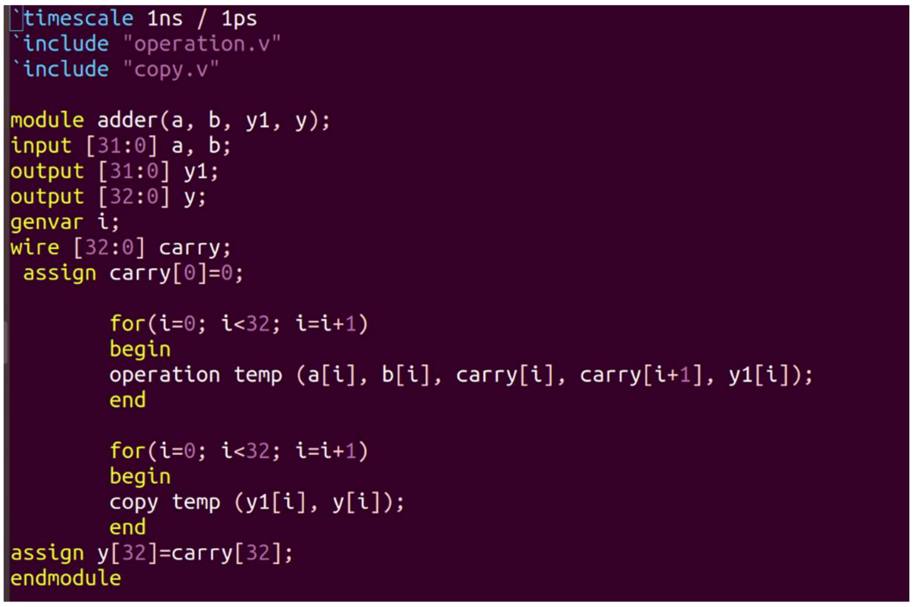
We have two 32 bit numbers

A31 A30..................A2 A1 A0

+ B31 B30.................B2 B1 B0

- Now, initially, we have two bits A0, B0. Take Cin=0(C0=0). Then we get sum and carry. The sum represents Y0 whereas the carry C1 is taken as Cin for the second bit addition where we are expected to add A1, B1, C1. Now, we get the sum Y1 and carry C2 which is provided as Cin for the next operation.
- We repeat this for all the bits to get Y31 in the last step along with C32. This process of building an adder sees carry out generated from carry in. So this gives the adder the name ripple carry adder.
- The last carry C32 is the 33rd bit. It gives us information about overflow. This is a very basic and intuitive method of solving the question. One might use carry look ahead and other methods for even more effective ways.

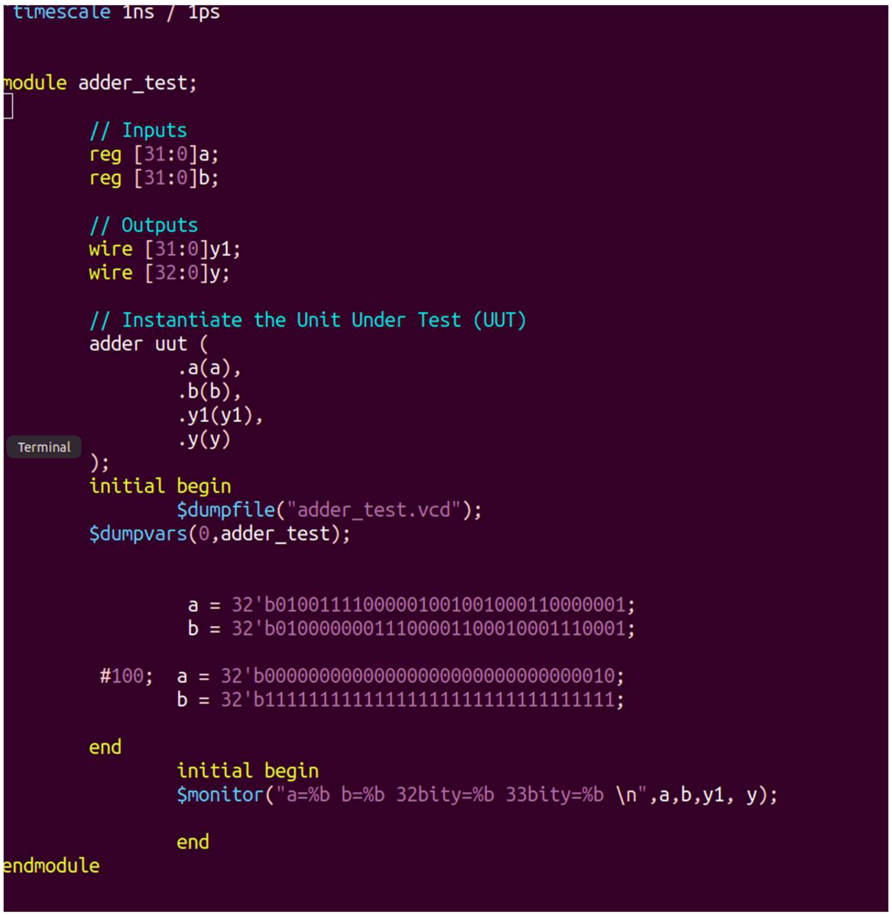
## Code:



```
`timescale 1ns / 1ps
`include "operation.v"
`include "copy.v"

module adder(a, b, y1, y);
input [31:0] a, b;
output [31:0] y1;
output [32:0] y;
genvar i;
wire [32:0] carry;
 assign carry[0]=0;

        for(i=0; i<32; i=i+1)
        begin
        operation temp (a[i], b[i], carry[i], carry[i+1], y1[i]);
        end

        for(i=0; i<32; i=i+1)
        begin
        copy temp (y1[i], y[i]);
        end
assign y[32]=carry[32];
endmodule
```

- We have defined the timescale and include the necessary modules. Here, the adder module contains inputs a and b. The outputs are y1 and y. We need two outputs due to the fact that there might be an overflow resulting in a 33rd bit.
- Y1 is the output corresponding to the 32 bit answer whereas y corresponds to the actual 33 bit output possible. The first bit(most significant bit) in y indicates an overflow by showing 1 and it shows 0 when there is no overflow.
- The carry array is of 33 bit length and we take the initial carry bit carry[0]=0. Now, we pass the parameters a[i], b[i], carry[i] as inputs in a module operation where we get outputs carry[i+1] and y1[i];
- In operation module we take
- **y1[i]=a[i]^b[i]^c[i]**
- **carry[i+1]= ((a[i]^b[i])&carry[i])|a[i]&b[i]**
- Now, we get y1[i] which is the 32bit answer neglecting the last carry bit.
- To include that bit as well, we write a loop where we assign y[i]=y1[i] and take the last bit y[32]=carry[32].

## Test bench:

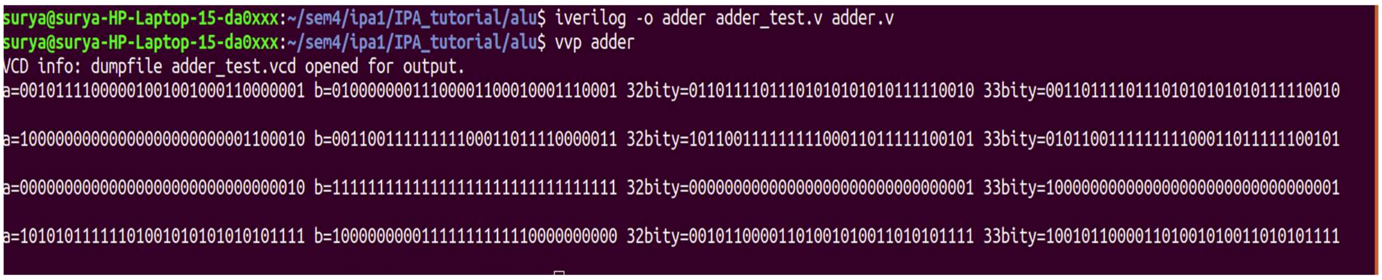


```
timescale 1ns / 1ps


module adder_test;

        // Inputs
        reg [31:0]a;
        reg [31:0]b;

        // Outputs
        wire [31:0]y1;
        wire [32:0]y;

        // Instantiate the Unit Under Test (UUT)
        adder uut (
                .a(a),
                .b(b),
                .y1(y1),
                .y(y)
        );
        initial begin
                $dumpfile("adder_test.vcd");
        $dumpvars(0,adder_test);


                 a = 32'b01001111000000100100100011000001;
                 b = 32'b01000000001110000110001000111000l;

         #100;  a = 32'b00000000000000000000000000000010;
                b = 32'b11111111111111111111111111111111;

        end
                initial begin
                $monitor("a=%b b=%b 32bity=%b 33bity=%b \n",a,b,y1, y);

                end
endmodule
```

Writing the test bench is similar to that of the previous cases. There is nothing new done for adder.

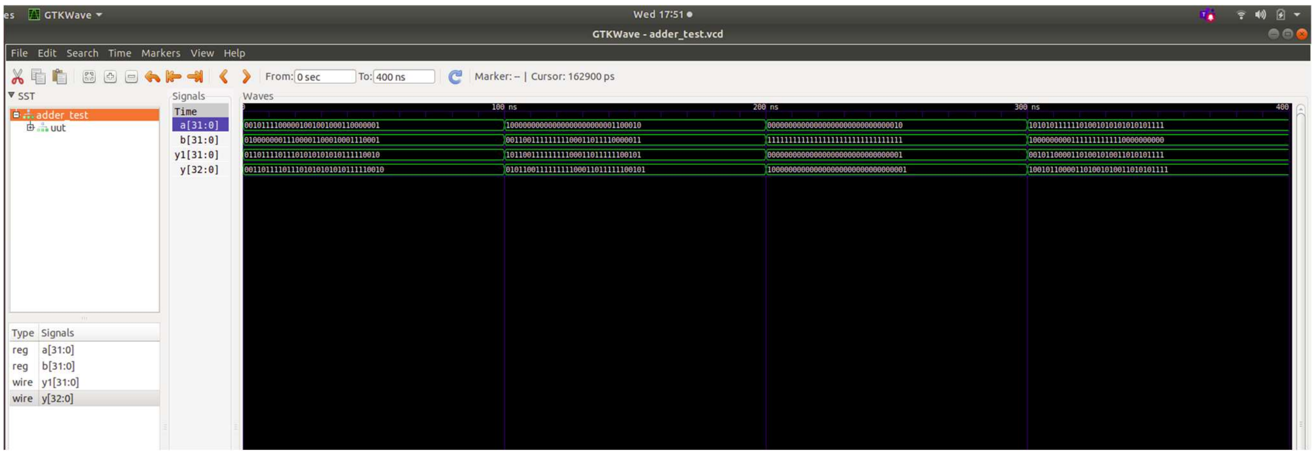
## Results:

Terminal output:



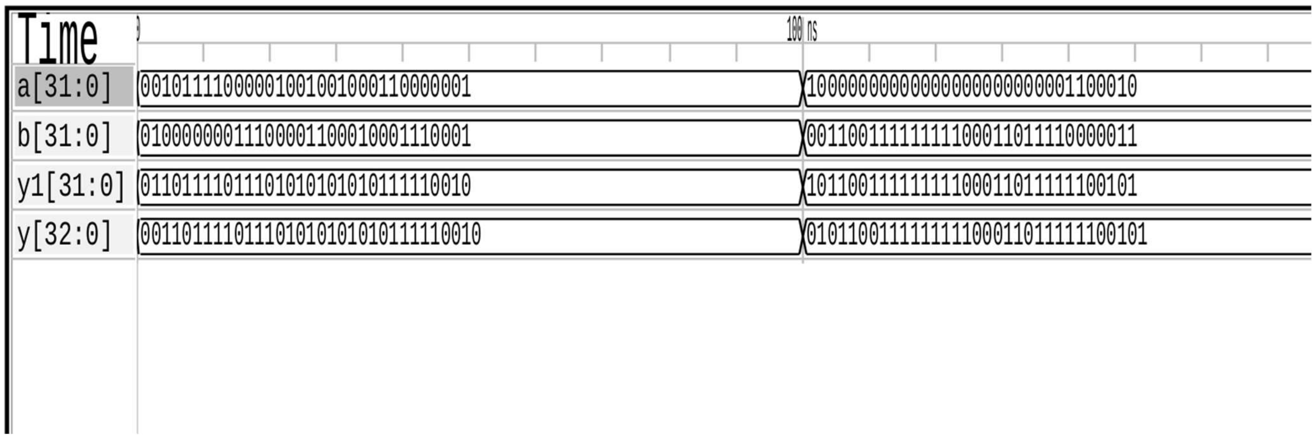
```
surya@surya-HP-Laptop-15-da0xxx:~/sem4/ipa1/IPA_tutorial/alu$ iverilog -o adder adder_test.v adder.v
surya@surya-HP-Laptop-15-da0xxx:~/sem4/ipa1/IPA_tutorial/alu$ vvp adder
VCD info: dumpfile adder_test.vcd opened for output.
a=00101111000001001001000110000001 b=01000000001110000110001000111000l 32bity=01101111011101010101010111110010 33bity=001101111011101010101010111110010

a=10000000000000000000000001100010 b=00110011111111100011011110000011 32bity=10110011111111100011011111100101 33bity=010110011111111100011011111100101

a=00000000000000000000000000000010 b=11111111111111111111111111111111 32bity=00000000000000000000000000000001 33bity=100000000000000000000000000000001

a=10101011111101001010101010101111 b=10000000001111111111110000000000 32bity=00101100001101001010011010101111 33bity=100101100001101001010011010101111
```

GTKwave output:

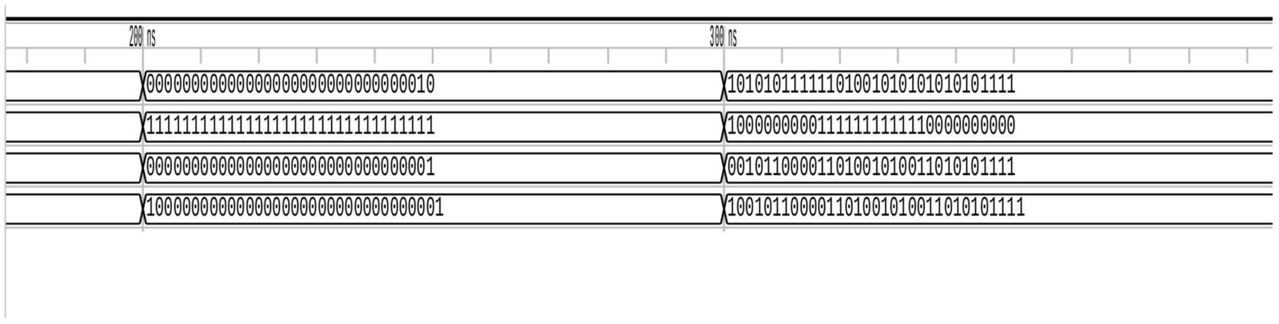


```
GTKWave - adder_test.vcd
File  Edit  Search  Time  Markers  View  Help
From: 0 sec  To: 400 ns    Marker: -- | Cursor: 162900 ps

SST: adder_test
       uut

Type  Signals
reg   a[31:0]
reg   b[31:0]
wire  y1[31:0]
wire  y[32:0]

Signals   Waves
Time      0                                  100 ns                              200 ns                              300 ns                              400
a[31:0]   00101111000001001001000110000001   10000000000000000000000001100010   00000000000000000000000000000010   10101011111101001010101010101111
b[31:0]   01000000011100001100010001110001   00110011111111100011011110000011   11111111111111111111111111111111   10000000001111111111110000000000
y1[31:0]  01101111011101010101010111110010   10110011111111100011011111100101   00000000000000000000000000000001   00101100001101001010011010101111
y[32:0]   001101111011101010101010111110010  010110011111111100011011111100101  100000000000000000000000000000001  100101100001101001010011010101111
```

Due to lack of visibility in the above file, a pdf version saved is shown below.



```
Time      0                                    100 ns
a[31:0]   00101111000001001001000110000001     10000000000000000000000001100010
b[31:0]   01000000011100001100010001110001     00110011111111100011011110000011
y1[31:0]  01101111011101010101010111110010     10110011111111100011011111100101
y[32:0]   001101111011101010101010111110010    010110011111111100011011111100101
```



```
          200 ns                                300 ns
          00000000000000000000000000000010      10101011111101001010101010101111
          11111111111111111111111111111111      10000000001111111111110000000000
          00000000000000000000000000000001      00101100001101001010011010101111
          100000000000000000000000000000001     100101100001101001010011010101111
```

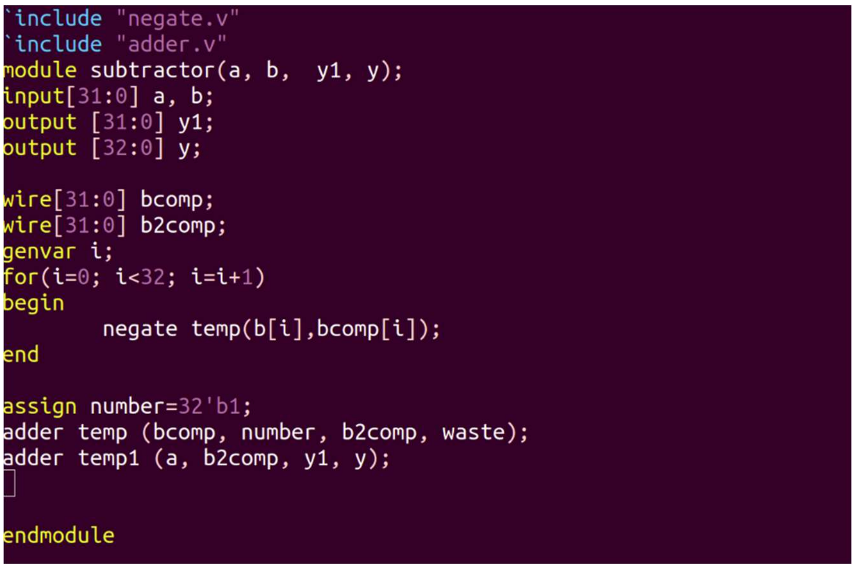
## Subtract32:

We are given two 32-bit signed numbers a and b. We are expected to subtract b from a (i.e. a-b) and produce the output y.

## Theory behind the subtractor:

- We are expected to implement a subtractor. a-b=a+(-b). So now, if we are  able to find out the value of –b, we can add a and –b to get the output.
- We know that to find –b, the method is by taking 2’s complement.
- To find 2’s complement of a number b, we just have to flip the bits of it. This means that for each of the 32 bits, if we have 1, we make it 0 and if we have 0, we make it 1.
- Now to this flipped number say ~b, we need to add 1. This means that we pass ~b, 1 as input to the adder and get –b.
- Now, since we have a, -b we pass them in the adder to get y1(32 bit answer) and y(33 bit answer). The extra bit is to accommodate an overflow incase any.

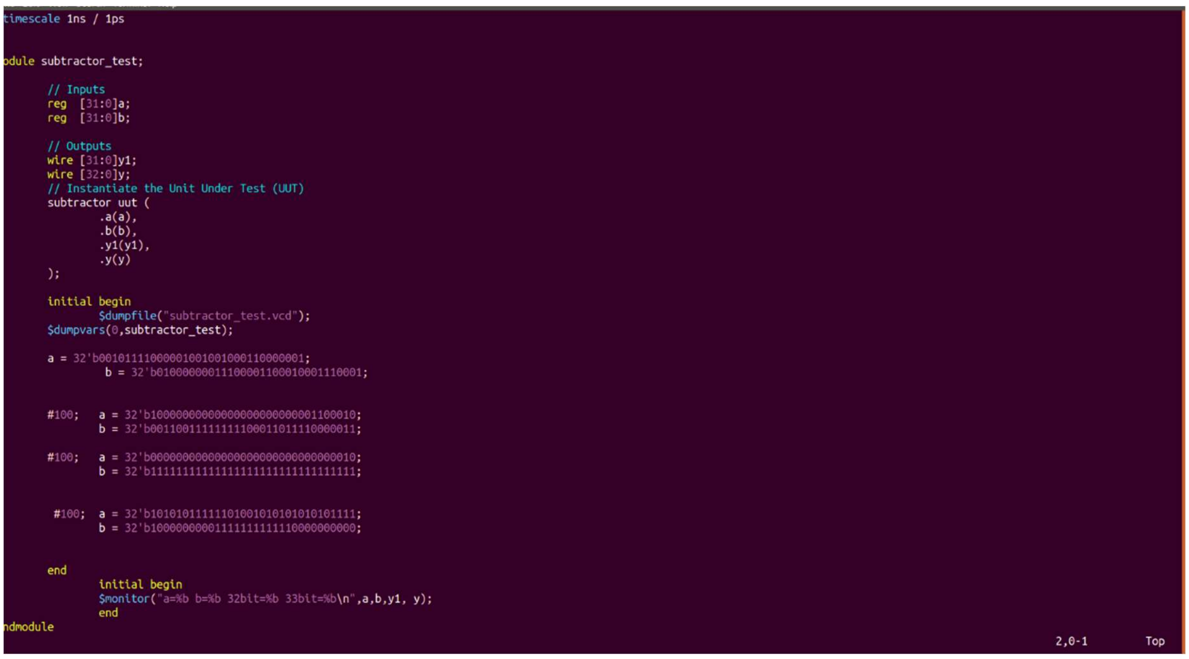
## Code:



```
`include "negate.v"
`include "adder.v"
module subtractor(a, b,  y1, y);
input[31:0] a, b;
output [31:0] y1;
output [32:0] y;

wire[31:0] bcomp;
wire[31:0] b2comp;
genvar i;
for(i=0; i<32; i=i+1)
begin
        negate temp(b[i],bcomp[i]);
end

assign number=32'b1;
adder temp (bcomp, number, b2comp, waste);
adder temp1 (a, b2comp, y1, y);


endmodule
```

- We include the necessary modules in the beginning and then define the inputs and outputs. We define temporary arrays, bcomp and b2comp.
- We write a for loop that takes b[i] as input and returns the output as ~b[i].
- All these values are stored in the vector bcomp. Then we pass bcomp and 1 into the adder(which adds the two numbers ) to get b2comp.
- Now, we simply pass a and b2comp into the adder to get y1(32 bit ) output and y(33 bit output).

## Test bench:

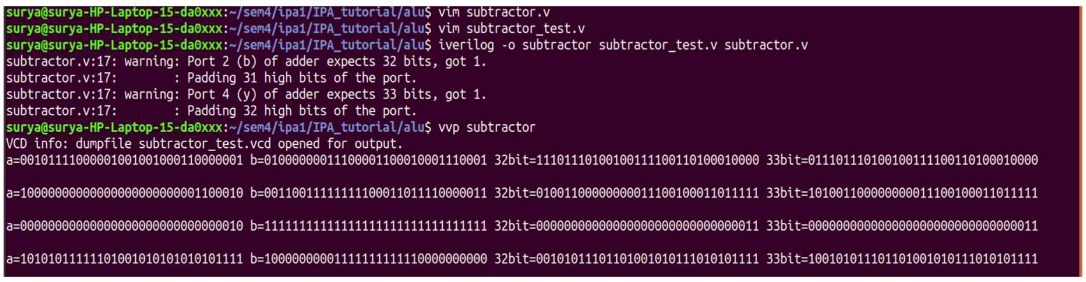


```
timescale 1ns / 1ps


odule subtractor_test;

      // Inputs
      reg  [31:0]a;
      reg  [31:0]b;

      // Outputs
      wire [31:0]y1;
      wire [32:0]y;
      // Instantiate the Unit Under Test (UUT)
      subtractor uut (
              .a(a),
              .b(b),
              .y1(y1),
              .y(y)
      );

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

      a = 32'b00101111000001001001000110000001;
               b = 32'b01000000001110000110001000111000 1;


      #100;   a = 32'b10000000000000000000000001100010;
              b = 32'b00110011111111100011011110000011;

      #100;   a = 32'b00000000000000000000000000000010;
              b = 32'b11111111111111111111111111111111;


       #100;  a = 32'b10101011111101001010101010101111;
              b = 32'b10000000001111111111110000000000;


      end
              initial begin
              $monitor("a=%b b=%b 32bit=%b 33bit=%b\n",a,b,y1, y);
              end
ndmodule
                                                                                2,0-1         Top
```

The test bench follows the same proecdure. Just defining the inputs, outputs, instantiating, dumping in necessary files and displaying input and ouput on monitor.

## Results:

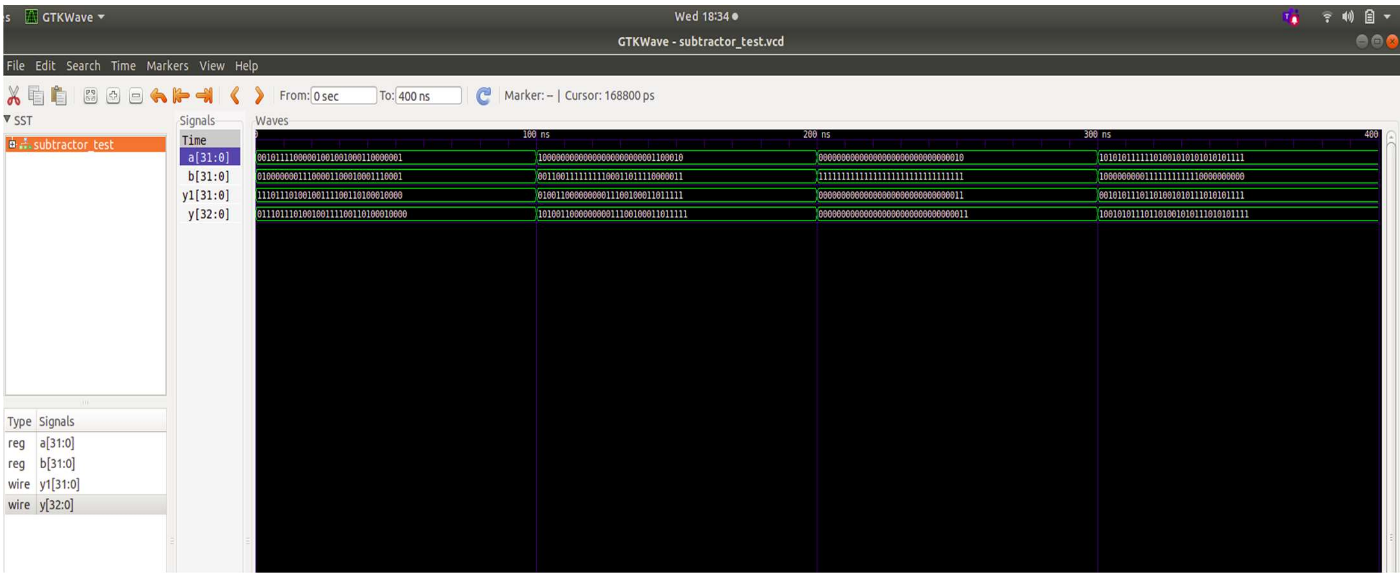
Terminal output:



```
surya@surya-HP-Laptop-15-da0xxx:~/sem4/ipa1/IPA_tutorial/alu$ vim subtractor.v
surya@surya-HP-Laptop-15-da0xxx:~/sem4/ipa1/IPA_tutorial/alu$ vim subtractor_test.v
surya@surya-HP-Laptop-15-da0xxx:~/sem4/ipa1/IPA_tutorial/alu$ iverilog -o subtractor subtractor_test.v subtractor.v
subtractor.v:17: warning: Port 2 (b) of adder expects 32 bits, got 1.
subtractor.v:17:        : Padding 31 high bits of the port.
subtractor.v:17: warning: Port 4 (y) of adder expects 33 bits, got 1.
subtractor.v:17:        : Padding 32 high bits of the port.
surya@surya-HP-Laptop-15-da0xxx:~/sem4/ipa1/IPA_tutorial/alu$ vvp subtractor
VCD info: dumpfile subtractor_test.vcd opened for output.
a=00101111000001001001000110000001 b=01000000001110000110001000111000 1 32bit=11101110100100111100110100010000 33bit=011101110100100111100110100010000

a=10000000000000000000000001100010 b=00110011111111100011011110000011 32bit=01001100000000011100100011011111 33bit=101001100000000011100100011011111

a=00000000000000000000000000000010 b=11111111111111111111111111111111 32bit=00000000000000000000000000000011 33bit=000000000000000000000000000000011

a=10101011111101001010101010101111 b=10000000001111111111110000000000 32bit=00101011101101001010111010101111 33bit=100101011101101001010111010101111
```

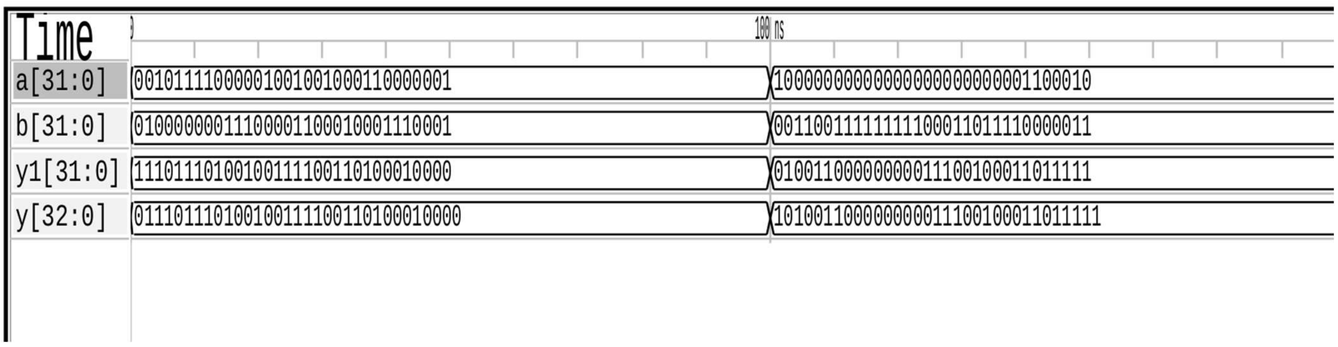
Note that the warnings are due to the fact that we are adding 32b’1(which is representation of 1). So the remaining bits are 0 padded which is expected already.

GKTwave output:

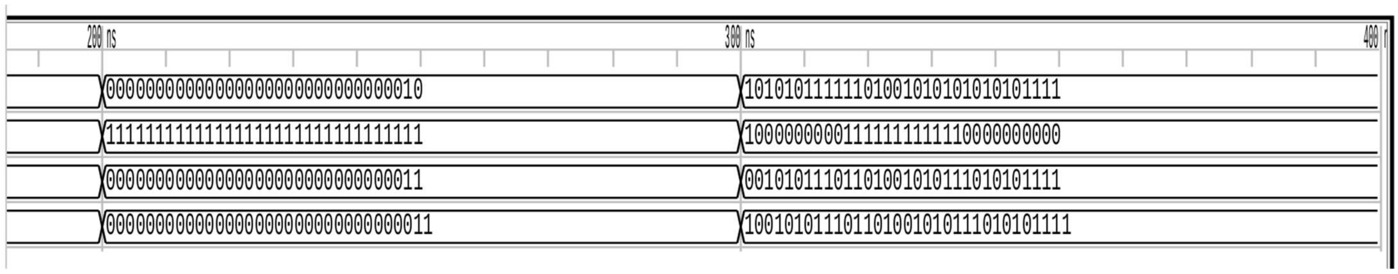


```
GTKWave - subtractor_test.vcd
From: 0 sec   To: 400 ns   Marker: -- | Cursor: 168800 ps

Type  Signals
reg   a[31:0]
reg   b[31:0]
wire  y1[31:0]
wire  y[32:0]

Time      0                                  100 ns                              200 ns                              300 ns
a[31:0]   00101111000001001001000110000001   10000000000000000000000001100010    00000000000000000000000000000010    10101011111101001010101010101111
b[31:0]   01000000011100001100010001110001   00110011111111100011011110000011    11111111111111111111111111111111    10000000001111111111100000000000
y1[31:0]  11101110100100111100110100010000   01001100000000011100100011011111    00000000000000000000000000000011    00101011101101001010111010101111
y[32:0]   011101110100100111100110100010000  101001100000000011100100011011111   000000000000000000000000000000011   100101011101101001010111010101111
```

Due to lack of visibility in the above file, a pdf version saved is shown below.



```
Time       0                                    100 ns
a[31:0]    00101111000001001001000110000001     10000000000000000000000001100010
b[31:0]    01000000011100001100010001110001     00110011111111100011011110000011
y1[31:0]   11101110100100111100110100010000     01001100000000011100100011011111
y[32:0]    011101110100100111100110100010000    101001100000000011100100011011111
```



```
200 ns                                  300 ns                                 400 ns
00000000000000000000000000000010        10101011111101001010101010101111
11111111111111111111111111111111        10000000001111111111100000000000
00000000000000000000000000000011        00101011101101001010111010101111
000000000000000000000000000000011       100101011101101001010111010101111
```

## ALU32:

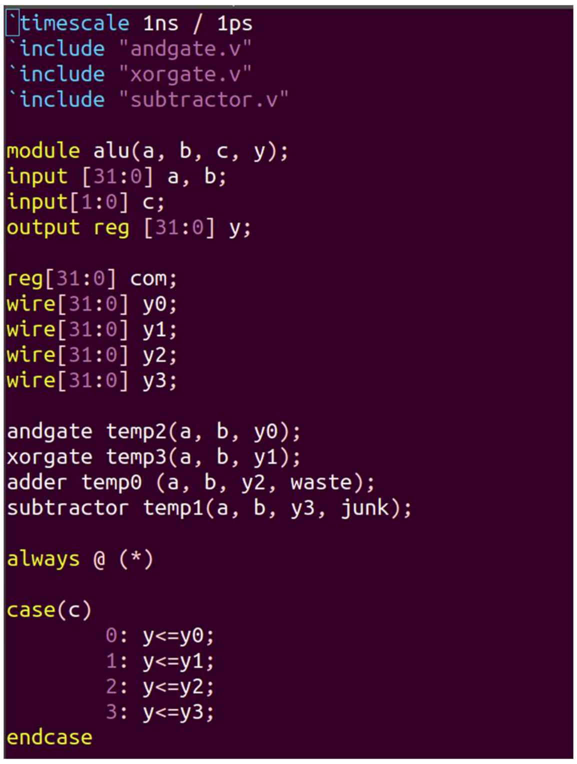
Now, our final goal is to combine the all the above question using a select line c. If, c=0 perform AND between a and b

c=1 perform XOR between a and b

c=2 perform addition between a and b

C=3 perform subtraction between a and b

## Code:



```
`timescale 1ns / 1ps
`include "andgate.v"
`include "xorgate.v"
`include "subtractor.v"

module alu(a, b, c, y);
input [31:0] a, b;
input[1:0] c;
output reg [31:0] y;

reg[31:0] com;
wire[31:0] y0;
wire[31:0] y1;
wire[31:0] y2;
wire[31:0] y3;

andgate temp2(a, b, y0);
xorgate temp3(a, b, y1);
adder temp0 (a, b, y2, waste);
subtractor temp1(a, b, y3, junk);

always @ (*)

case(c)
        0: y<=y0;
        1: y<=y1;
        2: y<=y2;
        3: y<=y3;
endcase
```

The code is self explanatory. We include the necessary modules. We mention what the inputs and outputs are. Then we take 4 wires which are used to store the computed values of the operations from 0 to 4. Then we write a case statement where depending upon the value of c, we assign y to one of yi.

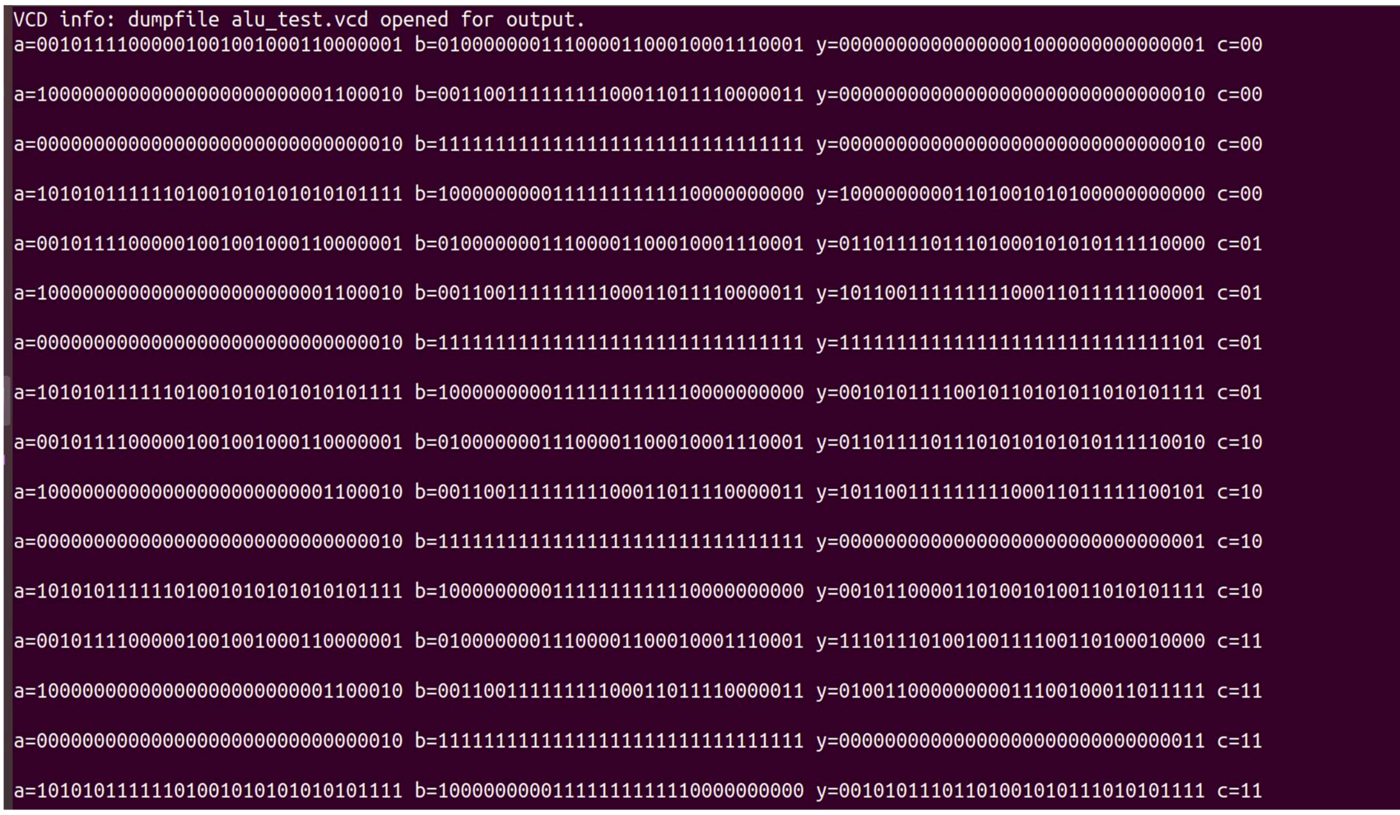
## Test bench:

Writing the test bench is slightly longer here. The overall procedure is the same but the values which we take and want to check make the testbench long. The values which we take are

```
a = 32'b00101111000001001001000110000001;
 b = 32'b01000000011100001100010001110001;
 c=00;

#100; a = 32'b10000000000000000000000001100010;
 b = 32'b00110011111111100011011110000011;

#100; a = 32'b00000000000000000000000000000010;
 b = 32'b11111111111111111111111111111111;


#100; a = 32'b10101011111101001010101010101111;
 b = 32'b10000000001111111111110000000000;
 #100;

a = 32'b00101111000001001001000110000001;
 b = 32'b01000000011100001100010001110001;
 c=01;

#100; a = 32'b10000000000000000000000001100010;
 b = 32'b00110011111111100011011110000011;

#100; a = 32'b00000000000000000000000000000010;
 b = 32'b11111111111111111111111111111111;


#100; a = 32'b10101011111101001010101010101111;
 b = 32'b10000000001111111111110000000000;
 #100;
 a = 32'b00101111000001001001000110000001;
 b = 32'b01000000011100001100010001110001;
 c=10;

#100; a = 32'b10000000000000000000000001100010;
 b = 32'b00110011111111100011011110000011;

#100; a = 32'b00000000000000000000000000000010;
 b = 32'b11111111111111111111111111111111;


#100; a = 32'b10101011111101001010101010101111;
 b = 32'b10000000001111111111110000000000;
 #100;
 a = 32'b00101111000001001001000110000001;
 b = 32'b01000000011100001100010001110001;
 c=11;

#100; a = 32'b10000000000000000000000001100010;
 b = 32'b00110011111111100011011110000011;

#100; a = 32'b00000000000000000000000000000010;
 b = 32'b11111111111111111111111111111111;


#100; a = 32'b10101011111101001010101010101111;
 b = 32'b10000000001111111111110000000000;
 #100;
```

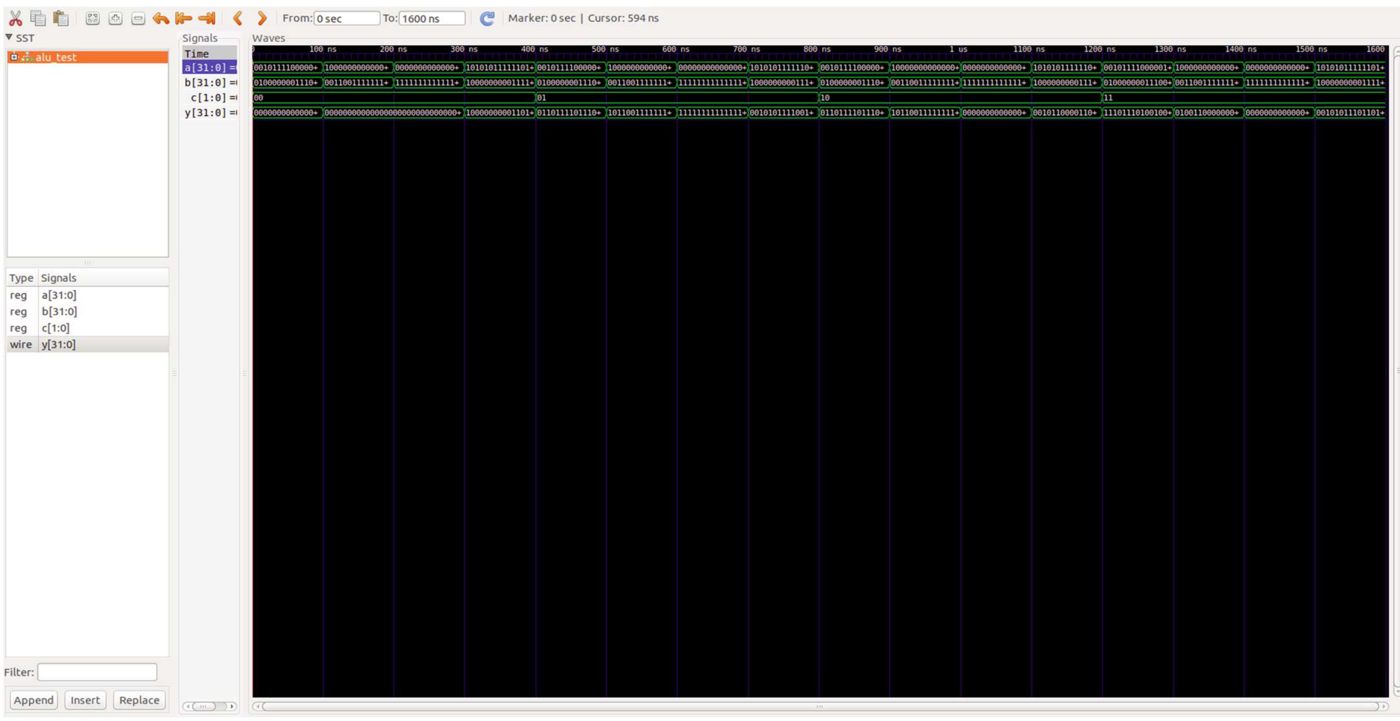
## Results:

Terminal output:



```
VCD info: dumpfile alu_test.vcd opened for output.
a=00101111000001001001000110000001 b=01000000011100001100010001110001 y=00000000000000001000000000000001 c=00

a=10000000000000000000000001100010 b=00110011111111100011011110000011 y=00000000000000000000000000000010 c=00

a=00000000000000000000000000000010 b=11111111111111111111111111111111 y=00000000000000000000000000000010 c=00

a=10101011111101001010101010101111 b=10000000001111111111110000000000 y=10000000001101001010100000000000 c=00

a=00101111000001001001000110000001 b=01000000011100001100010001110001 y=01101111011101000101010111110000 c=01

a=10000000000000000000000001100010 b=00110011111111100011011110000011 y=10110011111111100011011111100001 c=01

a=00000000000000000000000000000010 b=11111111111111111111111111111111 y=11111111111111111111111111111101 c=01

a=10101011111101001010101010101111 b=10000000001111111111110000000000 y=00101011110010110101011010101111 c=01

a=00101111000001001001000110000001 b=01000000011100001100010001110001 y=01101111011101010101010111110010 c=10

a=10000000000000000000000001100010 b=00110011111111100011011110000011 y=10110011111111100011011111100101 c=10

a=00000000000000000000000000000010 b=11111111111111111111111111111111 y=00000000000000000000000000000001 c=10

a=10101011111101001010101010101111 b=10000000001111111111110000000000 y=00101100001101001010011010101111 c=10

a=00101111000001001001000110000001 b=01000000011100001100010001110001 y=11101110100100111100110100010000 c=11

a=10000000000000000000000001100010 b=00110011111111100011011110000011 y=01001100000000011100100011011111 c=11

a=00000000000000000000000000000010 b=11111111111111111111111111111111 y=00000000000000000000000000000011 c=11

a=10101011111101001010101010101111 b=10000000001111111111110000000000 y=00101011101101001010111010101111 c=11
```

GKTwave output:



```
From: 0 sec  To: 1600 ns   Marker: 0 sec | Cursor: 594 ns

SST
  alu_test

Type  Signals
reg   a[31:0]
reg   b[31:0]
reg   c[1:0]
wire  y[31:0]

Time     0 - 100 ns - 200 ns - 300 ns - 400 ns - 500 ns - 600 ns - 700 ns - 800 ns - 900 ns - 1 us - 1100 ns - 1200 ns - 1300 ns - 1400 ns - 1500 ns - 1600
a[31:0]  0010111100000+ 1000000000000+ 0000000000000+ 1010101111101+ 0010111100000+ 1000000000000+ 00000000000000+ 1010101111110+ 0010111100000+ 10000000000000+ 0000000000000+ 1010101111110+ 0010111000001+ 1000000000000+ 0000000000000+ 1010101111101+
b[31:0]  0100000001110+ 0011001111111+ 1111111111111+ 10000000001111+ 0100000001110+ 0011001111111+ 11111111111111+ 1000000000111+ 0100000001110+ 00110011111111+ 1111111111111+ 1000000000111+ 01000000011100+ 0011001111111+ 1111111111111+ 10000000001111+
c[1:0]   00                                                         01                                                            10                                                           11
y[31:0]  0000000000000+ 00000000000000000000000000000+ 10000000001101+ 0110111101110+ 1011001111111+ 11111111111111+ 0010101111001+ 0110111101110+ 10110011111111+ 0000000000000+ 0010110000110+ 11101110100100+ 0100110000000+ 0000000000000+ 0010101101101+
```

If we make a pdf, we need to attach a lot pictures. Since already the output of the terminal is attached above, zoomed plots are not added.

The following link contains all the plots and codes:

https://iiitaphyd-my.sharepoint.com/:f:/g/personal/sri_surya_students_iiit_ac_in/Elt-NZT5zhVCsKfWZ0tI3RABQym-O9W_0YpAMarUcFKzyQ?e=qJNlzw