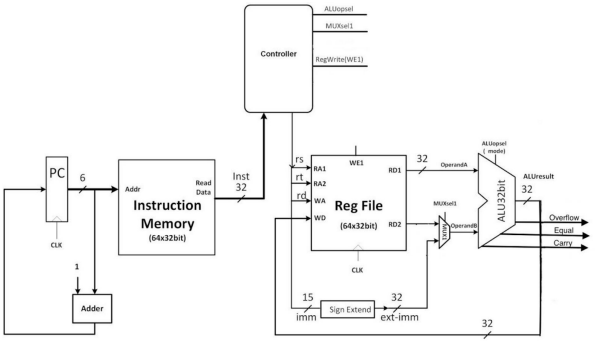
Group ID: 24
Changchuan Shen 83371717
Sir Gerad Antes 46045961

Module List:

- CS151_Processor (CS151-Processor.v) (6)
  - IM : insMem (insMem.v)
  - CTRL : Controller (Controller.v)
  - RF : RegFile64x32 (RegFile.v)
  - SE : sign_extend (sign_extend.v)
  - MUX1 : Mux2_1 (Mux2_1.v)
  - ALU : ALU32bit (ALU32bit.v) (8)
    - ADD : adder (adder.v)
    - SUB : sub (sub.v)
    - AND : andder (and.v)
    - OR : CS151_Or (CS151_Or.v)
    - NOT : CS151_not (CS151_not.v)
    - XOR : CS151_XOR (CS151_XOR.v)
    - SLL : CS151_SHIFT (CS151_SHIFT.v)
    - MOV : CS151_MOV (CS151_MOV.v)
- CS151_NOP (CS151_NOP.v)

Processor:

The CS151_Processor.v set all the modules according to the following design




```
insMem IM(.addr(addr),.RD(instruction));
Controller CTRL(.ins(instruction),.ALUopsel(ALUopsel),.MUXsel1(MUXsel1),.rs(rs),.rt(rt),.rd(rd),.imm(imm),.WE1(WE1));
RegFile64x32 RF(.reset(reset),.clk(clk),.WE1(WE1),.RA1(rs),.RA2(rt),.WA(rd),.WD(ALUresult),.RD1(OA),.RD2(RD2));
sign_extend SE(.a(imm),.out(ext_imm));
Mux2_1 MUX1(.cntrl(MUXsel1),.a(ext_imm),.b(RD2),.out(OB));
ALU32bit ALU(.OA(OA),.OB(OB),.ALUopsel(ALUopsel),.MUXsel1(MUXsel1),.ALUresult(ALUresult),.Overflow(overflow),.Equal(equal),.Carry(carry));
```

Test Waveform based on the following chart saved as a hardcoded 64x32bitRegFile(insMem.v, see below):

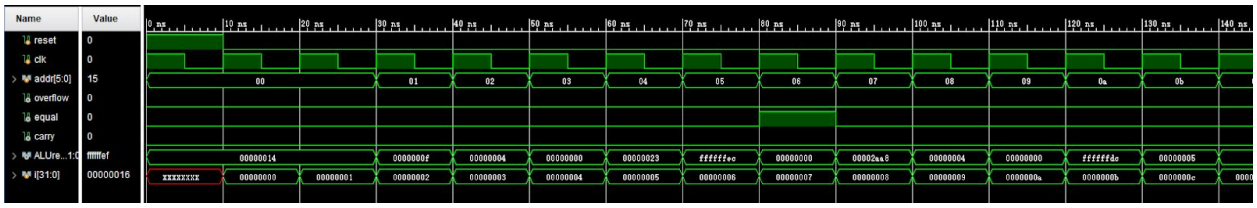
FUNCTION	RD	RS	RT / IMM	Operation	Result (hex)	note
ADDI	R1	R0	0x0014	R1 = R0 + 0x0014	0x00000014	
ADDI	R2	R0	0x000F	R2 = R0 + 0x000F	0x0000000F	
ADDI	R3	R0	0x0004	R3 = R0 + 0x0004	0x00000004	
NOP						
ADD	R4	R2	R1	R4 = R2 + R1	0x00000023	
SUBI	R5	R0	0x0014	R5 = R0 - 0x0014	0xFFFFFFEC	
SUB	R6	R1	R1	R6 = R1 - R1	0x00000000	assert equal flag
ANDI	R8	R5	0x2AAA	R8 = R5 & 0x2AAA	0x00002AA8	
AND	R2	R2	R1	R2 = 4	0x00000004	
NOP						
NOT	R9	R4		R9 = ~(R4)	0xFFFFFFDC	
ORI	R9	R2	0x0001	R9 = R2 | 0x0001	0x00000005	
SLLI	R9		0x0002	R9 = R9 << 2	0x00000014	
MOV	R2	R9		R2 = R9	0x00000014	
NOT	R6	R0		R6 = ~(R0)	0xFFFFFFFF	
ADDI	R1	R0	0x0002	R1 = R0 + 2	0x00000002	
SLLI	R1		0x001E	R1 = R1 << 30	0x80000000	
NOP						
ADD	R7	R6	R1	R7 = R6 + R1	0x7FFFFFFF	assert overflow and carry
ADD	R6	R6	R6	R6 = R6 + R6	0xFFFFFFFE	assert carry
NOP						
OR	R4	R4	R5	R4 = R4 | R5	0xFFFFFFEF	
XOR	R4	R8	R2	R4 = R8 ^ R2	0x00002ABC	
XORI	R4	R4	0x2AAA	R4 = R4 ^ 0x2AAA	0x00000016	
MOVI	R10		0x0006	R10 = 0x0006	0x00000006	
MOVI	R2		0x0004	R2 = 0x0004	0x00000004	
SLL	R10	R2		R10 = R10 << R2	0x00000060	

```
module insMem(
    input [5:0] addr,
    output [31:0] RD
    );

    reg [31:0] RF[63:0];

    assign RD = RF[addr];
    initial begin
        RF[0]  = {1'b1,6'd0,6'd1,4'b0001,15'h0014}; //0x14
        RF[1]  = {1'b1,6'd0,6'd2,4'b0001,15'h000F}; //0xF
        RF[2]  = {1'b1,6'd0,6'd3,4'b0001,15'h0004}; //0x4
        RF[3]  = 32'b0;
        RF[4]  = {1'b0,6'd2,6'd4,4'b0001,6'd1,9'h0}; //0x23
        RF[5]  = {1'b1,6'd0,6'd5,4'b0010,15'h0014}; //0xFFFFFFEC
        RF[6]  = {1'b0,6'd1,6'd6,4'b0010,6'd1,9'h0};//0x0 equal
        RF[7]  = {1'b1,6'd5,6'd8,4'b0101,15'h2AAA}; //0x2AA8
        RF[8]  = {1'b0,6'd2,6'd2,4'b0101,6'd1,9'h0};//0x4
        RF[9]  = 32'b0;
        RF[10] = {1'b0,6'd4,6'd9,4'b0111,15'b0};    //0xFFFFFFDC
        RF[11] = {1'b1,6'd2,6'd9,4'b0110,15'h0001}; //0x5

        RF[12] = {1'b1,6'd0,6'd9,4'b1001,15'h0002}; //0x14
        RF[13] = {1'b0,6'd9,6'd2,4'b1011,15'h0};  // 0x14
        RF[14] = {1'b0,6'd0,6'd8,4'b0111,15'h0}; //0xFFFFFFFF
        RF[15] = {1'b1,6'd0,6'd1,4'b0001,15'h0002}; //0x2
        RF[16] = {1'b1,6'd0,6'd1,4'b1001,15'h001E}; //0x80000000
        RF[17] = 32'b0;
        RF[18] = {1'b0,6'd8,6'd7,4'b0001,6'd1,9'h0}; //0x7FFFFFFF overflow carry
        RF[19] = {1'b0,6'd8,6'd8,4'b0001,6'd8,9'h0}; //0xFFFFFFFE carry
        RF[20] = 32'b0;
        RF[21] = {1'b0,6'd4,6'd4,4'b0110,6'd5,9'h0}; //0xFFFFFFEF
        RF[22] = {1'b0,6'd8,6'd4,4'b1000,6'd2,9'h0}; //0x2ABC
        RF[23] = {1'b1,6'd4,6'd4,4'b1000,15'h2AAA}; //0x16
        RF[24] = {1'b1,6'd0,6'd10,4'b1011,15'h0006}; //0x6
        RF[25] = {1'b1,6'd0,6'd2,4'b1011,15'h0004}; //0x4
        RF[26] = {1'b0,6'd2,6'd10,4'b1001,15'h0}; //0x60
    end
endmodule
```

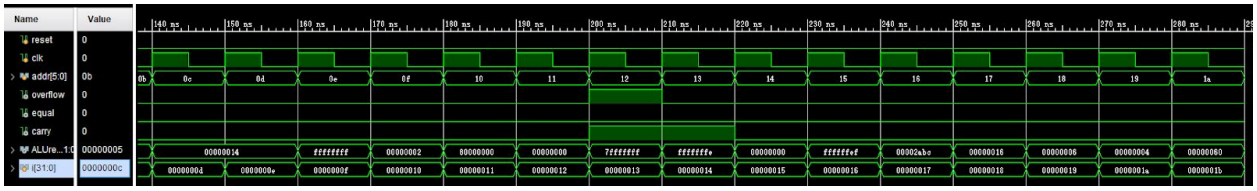
Testbench file name: Processor_tb.v

Result:

0-140ns:



140-280ns:



ALU32bit:

```
module ALU32bit(
    input MUXsel1,
    input [31:0] OA,OB,
    input [3:0] ALUopsel,
    output [31:0] ALUresult,
    output Overflow,
    output Equal,
    output Carry
    );
    wire [31:0] f0000;//NOP
    wire [31:0] f0001;//ADD
    wire oa,ca;
    wire [31:0] f0010;//SUB
    wire es,os,cs;
    wire [31:0] f0101;//AND
    wire [31:0] f0110;//OR
    wire [31:0] f0111;//NOT
    wire [31:0] f1000;//XOR
    wire ex;
    wire [31:0] f1001;//SLL
    wire osl;
    wire [31:0] f1011;//MOV

    reg [31:0] result,MOVA;
    reg o,e,c;
    adder ADD(.A(OA),.B(OB),.Sum(f0001),.Carry(ca),.Overflow(oa));
    sub SUB(.A(OA),.B(OB),.Sum(f0010),.Equal(es),.Carry(cs),.Overflow(os));
    andder AND(.a(OA),.b(OB),.andder(f0101));
    CS151_Or OR(.a(OA),.b(OB),.out(f0110));
    CS151_not NOT(.a(OA),.out(f0111));
    CS151_XOR XOR(.a(OA),.b(OB),.out(f1000),.equal(ex));
    CS151_SHIFT SLL(.a(OA),.b(OB),.out(f1001),.overflow(osl));
    CS151_MOV MOV(.a(MOVA),.out(f1011));
```

// connect all the wires with submodules

```
    assign ALUresult = result;
    assign Overflow = o;
    assign Equal = e;
    assign Carry = c;
    always @(*) begin
        case(ALUopsel)
        4'b0000:{result,e,o,c} = {32'b0,1'b0,1'b0,1'b0};
        4'b0001:{result,e,o,c} = {f0001,1'b0,oa,ca};
        4'b0010:{result,e,o,c} = {f0010,es,os,cs};
        4'b0101:{result,e,o,c} = {f0101,1'b0,1'b0,1'b0};
        4'b0110:{result,e,o,c} = {f0110,1'b0,1'b0,1'b0};
        4'b0111:{result,e,o,c} = {f0111,1'b0,1'b0,1'b0};
        4'b1000:{result,e,o,c} = {f1000,ex,1'b0,1'b0};
        4'b1001:{result,e,o,c} = {f1001,1'b0,osl,1'b0};
        4'b1011:begin {result,e,o,c} = {f1011,1'b0,1'b0,1'b0};MOVA = MUXsel1 ? OB:OA; end
        endcase
    end

endmodule
```

// setup outputs according to selectline

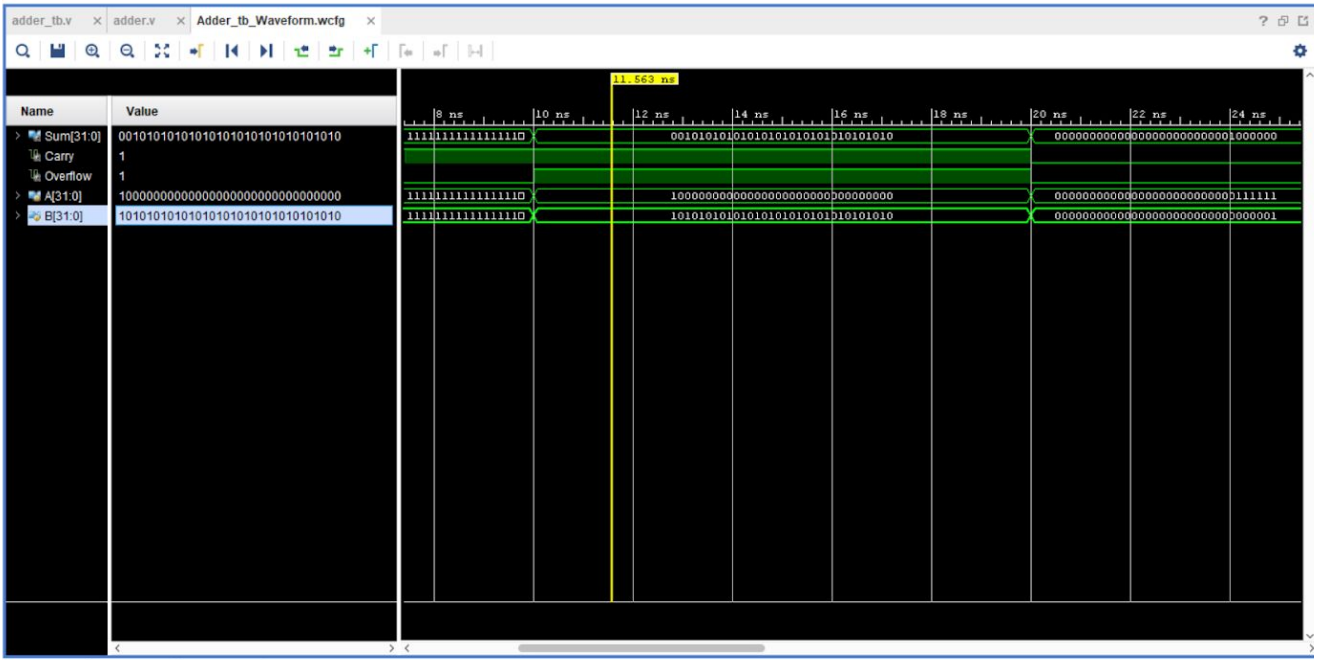
Modules(includes the submodules inside ALU32bit):

Adder:

The adder has two inputs [31:0]A, and [31:0]B. Three output the [31:0]Sum (which is the result of the addition of inputs A and B), the Carry flag(if the sum is too big) and the Overflow flag(if there if overflow) The code below is for the adder. First there are the inputs and outputs. Next is a wire [32:0] OverflowCheck which is one bit bigger than the sum to check for overflow. Next is the integer OFC to store if there is overflow. Next is Sum = A + B which is what the adder does. Next is OverflowCheck = A+B which will be used to check if there is a carry. Then there is a

check for the Carry Flag, if the Overflowcheck is the same as the sum then Carry flag is high if not then low. If theres is overflow then the signs of the two inputs will be the same. If the are then if the input sign and the sum sign are different there is overflow. Then the overflow and carry flasgs are assigned.

```
module adder(
    input [31:0]A,
    input [31:0]B,
    output [31:0]Sum,
    output [1:0]Carry,
    output [1:0]Overflow
    );
wire [32:0] OverflowCheck; // will be used to compare to the sum for carry check and  is one bit more
integer OFC; // used to store if there is overflow
assign Sum = A+B; // what the adder does
assign OverflowCheck = A+B; // is the sum but one more bit
integer C; // used to store if there is carry
always@(*) begin
    if (OverflowCheck !== Sum ) // if overflowcheck and sum aren't equal that means there's a carry
        C = 1;
    else
        C = 0;
    end
always@(*) begin
        if (A[31] == B[31])begin //for overflow if same sign that might be overflow
            if (A[31] != Sum[31] ) // is signs have fliped that means overflow
                OFC = 1;
            else
                OFC = 0;
            end
        else
            OFC = 0;
        end
assign Overflow = OFC; // assign overflow
assign Carry = C; // assign carry
endmodule
```

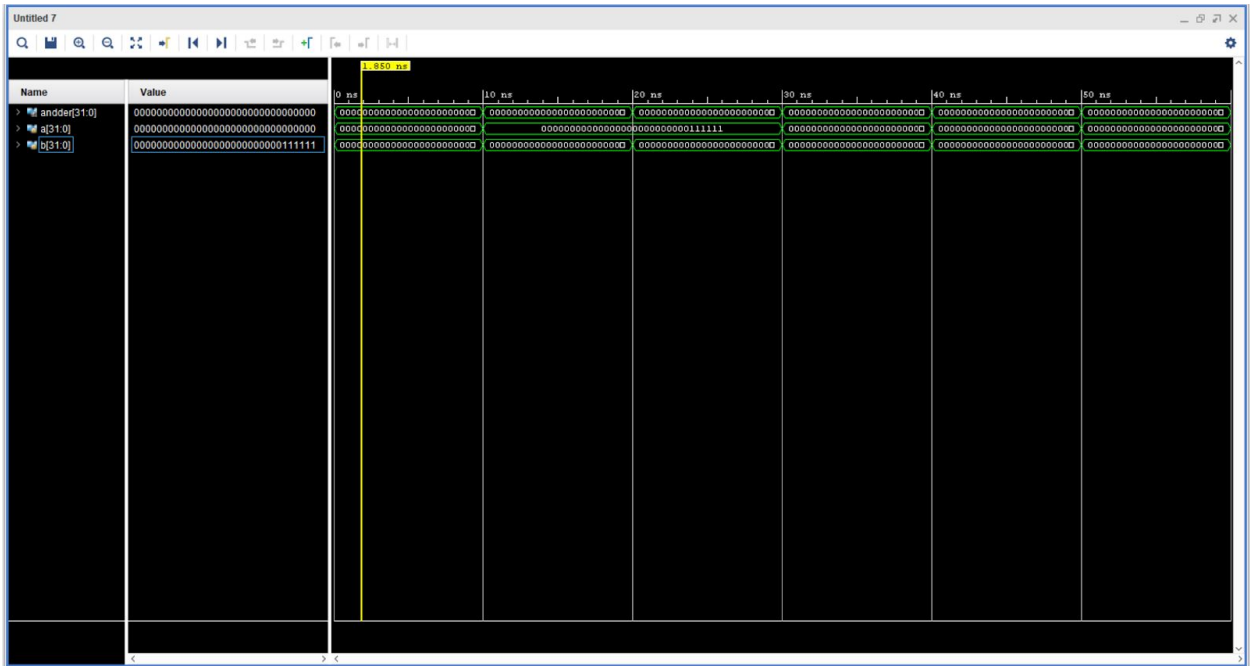


This waveform so that the adder is properly working. During the addition of A= 32'b11111111111111111111111111111111; B=32'b1; we got the result of 11111111111111111111111111111110 which is the correct result and the Carry flag is high because of there being a carry.

During the addition of A= 32'b11111111111111111111111111111111; B=32'b11111111111111111111111111111111; we got the result of 00101010101010101010101010101010 which is the correct result and the Carry flag and Overflow flag being high.
The other two were throwins for a non carry and overflow flags.
10 A= 32'b10000000000000000000000000000000;
B=32'b10101010101010101010101010101010;
#10 A= 32'b111111; B=32'b000001;

AND:
There are two inputs input [31:0]a, input [31:0]b, and one output output [31:0]andder(which is the result of the two inputs being anded. The code below is just assigning the and(which is when assigned a high bit if both bits are high and if not is low.) of a and b to the output(andder).
Code

```
module andder(
    input [31:0]a,
    input [31:0]b,
    output [31:0]andder
    );
assign andder = a&b; // the and of a and b being assigned to the output.
Endmodule
```



During the AND of a= 32'b000000; b=32'b111111; the result was 0 which is the correct result. This testing the result of no bits being the same.

During the AND of a= 32'b111111; b=32'b000001; the result was 1 which is the correct result. This was testing the result of 1 bit being the same.

During the AND of a= 32'b111111; b=32'b111111; the result was 111111 which is the correct result. This was testing the result of all bits being the same.

There rest were random throwins.

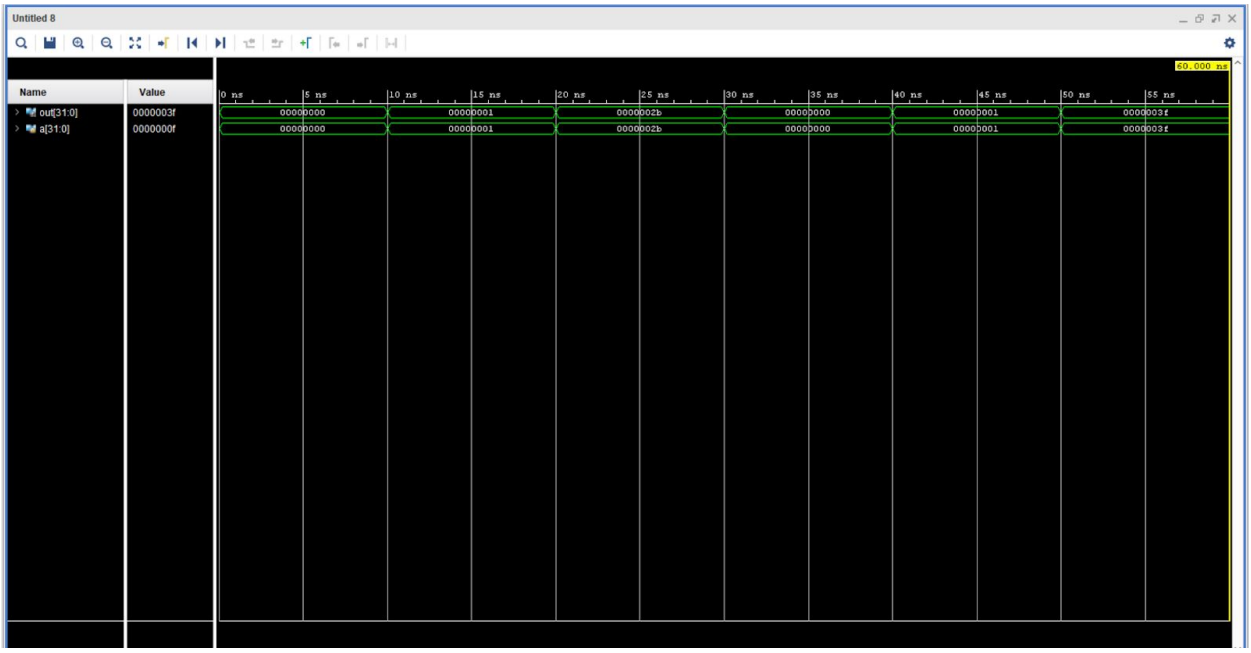
```
#10 a= 32'H0; b=32'H0014;
#10 a= 32'H1; b=32'H000F;
#10 a= 32'b111111; b=32'H0004;
#10 a= 32'HF; b=32'HF;
```

MOV:

The mov has one input input [31:0]a and one output output [31:0]out. The MOV module just put the input value in the output value.

Code

```
module CS151_MOV(
   input [31:0]a,
   output [31:0]out
   );
assign out = a; // put the input value to the output value
Endmodule
```




```
    a= 32'b000000;
    #10 a= 32'b000001;
    #10 a= 32'b101011;
    #10 a= 32'H0;
    #10 a= 32'H1;
    #10 a= 32'b111111;
    #10 a= 32'HF;
```

All input values after the MOV module are the output values which is how it works.

NOP:

For the NOP module it does nothing the is no code and there is nothing to test. It does nothing.

Code

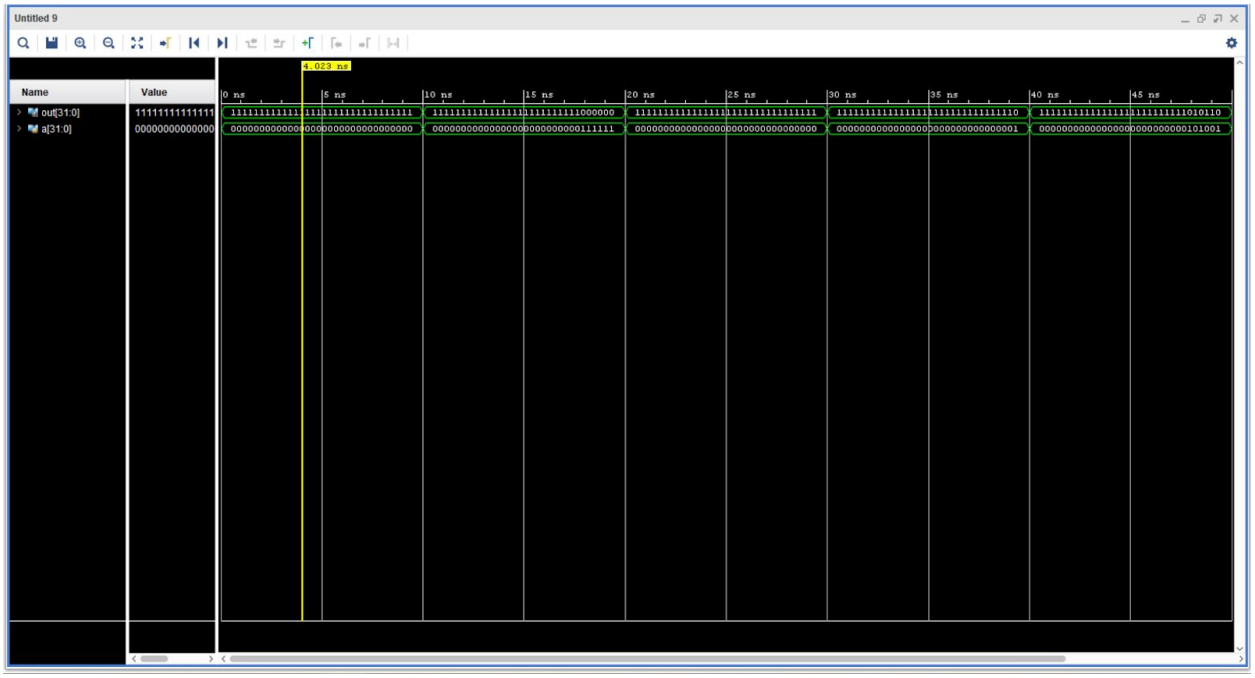
```
module CS151_NOP(

  );
endmodule
```

NOT:

For the NOT module there is one input input [31:0]a, and one output output [31:0]out. The out is assigned the inverted bits of the input a.

```
module CS151_not(
  input [31:0]a,
  output [31:0]out
  );
assign out = ~a; // output is assigned the inverted bits of input
```

endmodule




```
        a= 32'b000000;
    #10 a= 32'b111111;
    #10 a= 32'H0;
    #10 a= 32'H1;
    #10 a= 32'b101001;
    #10 a= 32'b111000;
```

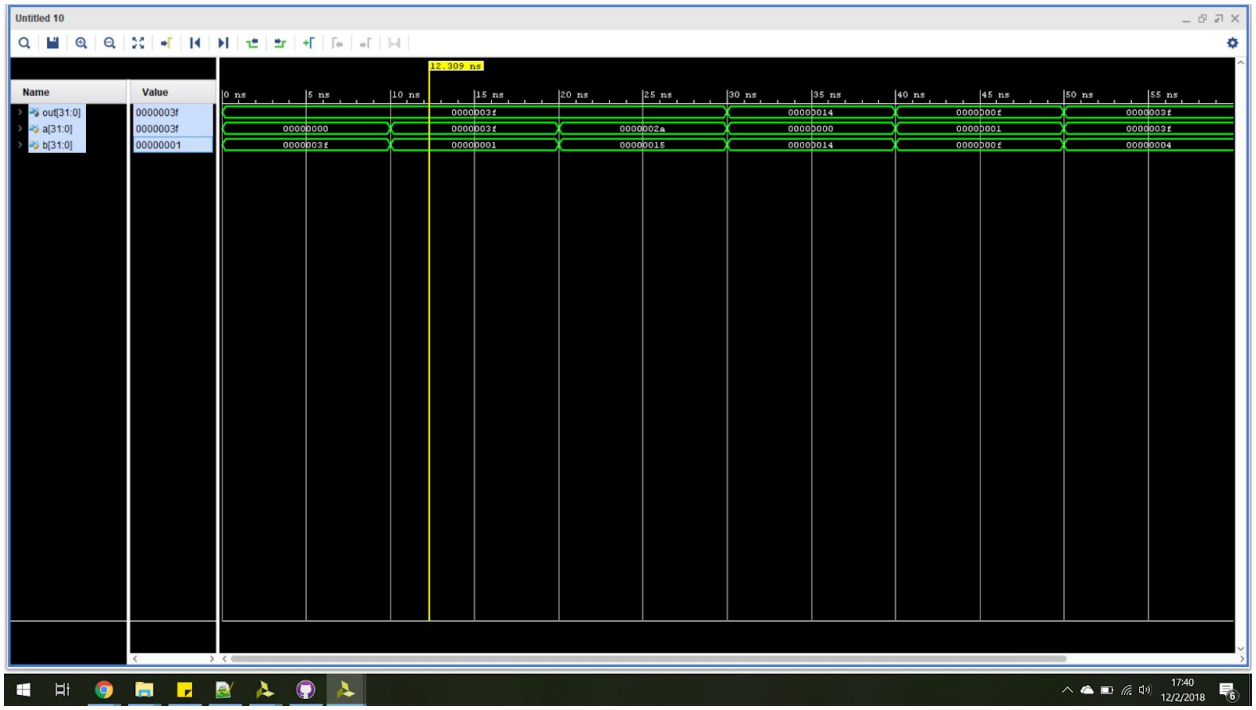
These output values are shown above to be the inverse of the input bits

OR:

The OR module has two inputs input [31:0]a, input [31:0]b, and one output output [31:0]out. The module will take compare the bits that high in each position and return that result to the output.

Code

```
module CS151_Or(
   input [31:0]a,
   input [31:0]b,
   output [31:0]out
   );
assign out = a|b; //does the Or opertion
endmodule
```



During the OR operation inputs a= 32'b000000; b=32'b111111; resulted in 111111 which is the correct result(we wanted to test all zeros and a couple of 1 bits)

During the OR operation inputs #10 a= 32'b111111; b=32'b000001; returned 1 which is the correct result (we want to test 1 bit being ored)

During the OR operation inputs #10 a= 32'b101010; b=32'b010101; resulted in 0 which is correct (we want to test values that would result in 0)

```
#10 a= 32'H0; b=32'H0014;
#10 a= 32'H1; b=32'H000F;
#10 a= 32'b111111; b=32'H0004;
#10 a= 32'HF; b=32'HF;
```

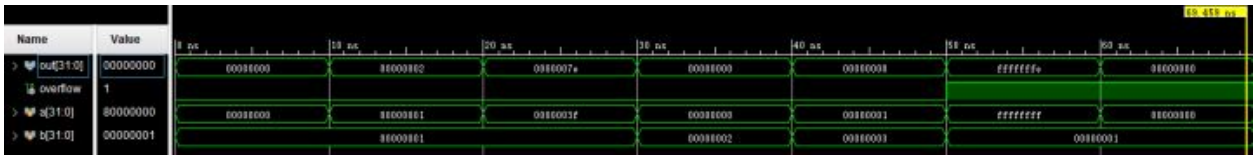
SHIFT:

The shift operator has two inputs input [31:0]a, input [31:0]b, and two outputs output [31:0]out (The result of the shift), and output overflow(the overflow flag showing overflow when high). The operator assignes output of the shift of a by b. Then checks for overflow because there is overflow when output is smaller than the input because of the left most 1 bit being shifted off.

Code:

```
module CS151_SHIFT(
    input [31:0]a,
    input [31:0]b,
    output [31:0]out,
    output overflow

    );
```

```
assign out = a<<b; // doing shift of a by b
integer OFC; // show overflow
always@(*) begin
        if (out < a  ) // overflow will happen when the left most high bit is shifted off thus when the output is smaller than the input you have overflow.
           OFC = 1;
        else
           OFC = 0;
        end
assign overflow = OFC; // assign overflow flag
Endmodule
```



During the << operation a= 32'b000000; b=1; resulted in 0 (we wanted to we one that didi nothing)

During the << operation #10 a= 32'b000001; b=1; resulted in 10 (we wanted to see a shift of one value)

#10 a= 32'b111111; b=1;

#10 a= 32'H0; b=2;

#10 a= 32'H1; b=3;

During the << operation

#10 a= 32'b11111111111111111111111111111111; b=1;

#10 a= 32'b10000000000000000000000000000000; b=1;

(we wanted to see a shift that resulted in overflow)

#10 a= 32'HFFFFFFFF; b=1;

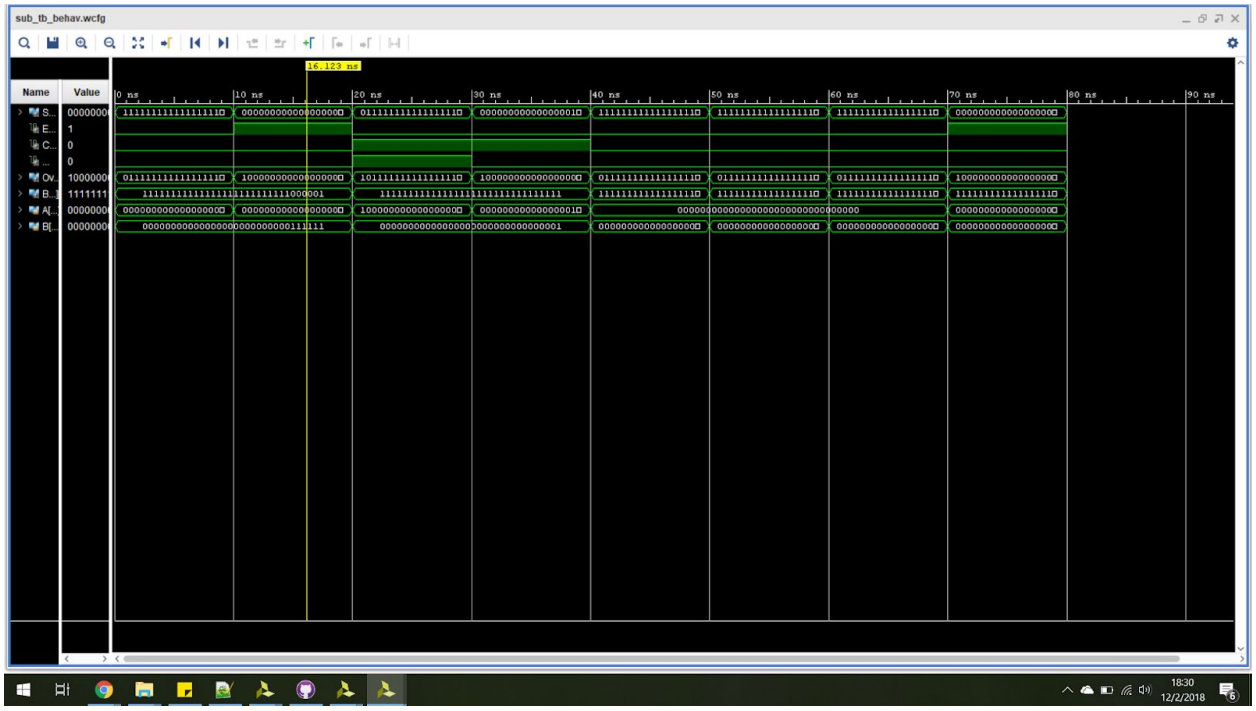
SUB:

The sub operator has two inputs input [31:0]A, input [31:0]B, and six outputs output [31:0]Sum, output [1:0]Equal, output [1:0]Carry output [1:0]Overflow, output [32:0]OverflowCheck, // this was to see the subtraction after twos comp ,output [31:0] Beta // the twos comp number. The code shows the substraction of the two inputs. The equal falg is raised if the output is equal to 0. The carry flag is raised if the output if of the extedned sum(OverflowCheck) first bit is high. Then the overflow flag is high if the input signs are opposite then the sum and first input sign are opposites.

```
module sub(
  input [31:0]A,
   input [31:0]B,
   output [31:0]Sum,
```

```
   output [1:0]Equal,
   output [1:0]Carry,
   output [1:0]Overflow,
   output [32:0]OverflowCheck, // this was to see the subtraction after twos comp
   output [31:0] Beta // this was to see the twos comp
   );
   wire [32:0] OverflowCheck; //check for carry with a number bigger than the sum
   wire [31:0] ov; //is the sum after twos comp
   wire[31:0] ba; // forgot to coment out
   integer OFC; // integer to check the if overflow
   integer s; // is the sum
   integer C; // the integer for carry
   assign Sum = A-B; // is the subtracting
   assign Beta = ~B + 1'b1; // twos comp
   assign OverflowCheck = A+Beta; // twos comp then add with [32:0]
   assign ov = A+Beta; //twos comp then add with [31:0]
 always@(*) begin
          if (Sum == 32'b00000000000000000000000000000000 ) // if after sub is 0 then equal
             s = 1;
          else
             s = 0;
          end
  always@(*) begin
       if (OverflowCheck[32] == 1'b1 ) // if first number is high on on the extended sum then carry
high
          C = 1;
       else
          C = 0;
       if (A==B)
          C=0;
       end
   always@(*) begin
       if (A[31] != B[31])begin //  overflow for subtraction needs to be opposite signs. Before twos
comp
             if (A[31] != ov[31] )    // if input sign is opposite sign from the sum then overflow
                OFC = 1;
             else
                OFC = 0;
             end
       else begin
          OFC = 0;
             end
             end
```

```
    assign Overflow = OFC;
    assign Carry = C;
    assign Equal = s;

endmodule
```



A= 32'b000000; B=32'b111111;
During the sub opertaion inputs #10 A= 32'b111111; B=32'b111111; were subtracted and equal 0 thus rasing the equal flag.

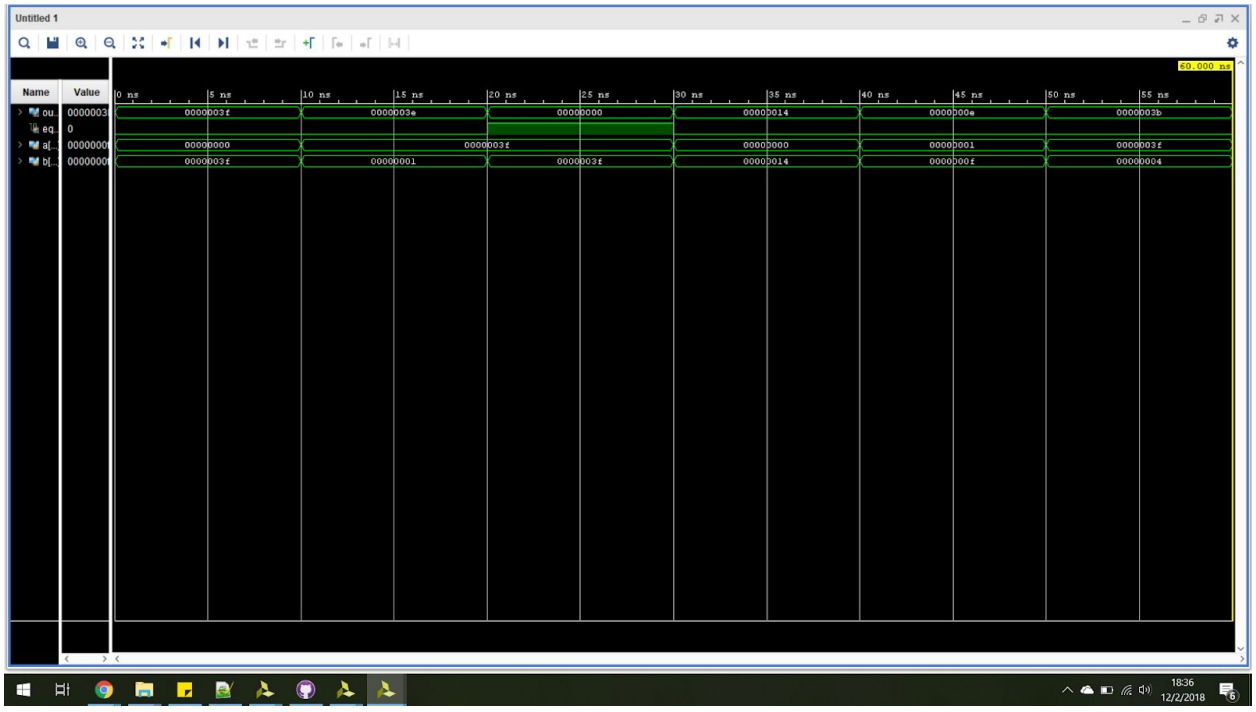
During the sub operation inputs#10 A= 32'b10000000000000000000000000000000; B=32'b01; were subtracted resulting in a correct overflow and carry.

```
    #10 A= 32'HFFFF; B=32'H1;
    #10 A= 32'H0; B=32'H0014;
    #10 A= 32'H0; B=32'H000F;
    #10 A= 32'H0; B=32'H0004;
    #10 A= 32'HF; B=32'HF;
    #10 A= 32'H0; B=32'H0;
```

XOR:
The XOR operator has the inputs input [31:0]a, input [31:0]b, and the outputs output [31:0]out(the sum), output equal(is high when the inputs are equal) The XOR operator first xors the two inputs a and b. Then checks the if the output is 0 then if so makes the equal flag high.
Code

```
module CS151_XOR(
```

```
    input [31:0]a,
    input [31:0]b,
    output [31:0]out,
    output equal
    );
assign out = a^b; // xoring
integer e;
always@(*) begin
    if (out == 32'b00000000000000000000000000000000) // check if out is 0 if so then equal flag is high.
        e = 1;
    else
        e = 0;
    end
assign equal = e; // equal being assigned.
Endmodule
```



During the xor operation inputs a= 32'b000000; b=32'b111111; resulted in 111111 which is correct(wanted to test zeros and a couple ones)

```
        #10 a= 32'b111111; b=32'b000001;
```

During the xor operation inputs #10 a= 32'b111111; b=32'b111111; resulted in 0 because in xor its high if bits are opposite. (wanted to test equal flag.)

```
        #10 a= 32'H0; b=32'H0014;
        #10 a= 32'H1; b=32'H000F;
        #10 a= 32'b111111; b=32'H0004;
        #10 a= 32'HF; b=32'HF;
```