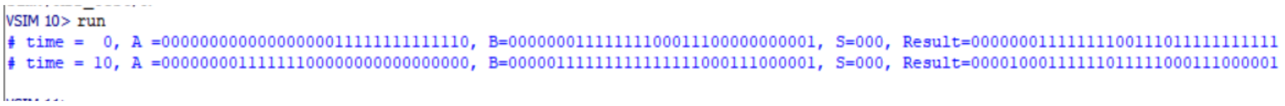
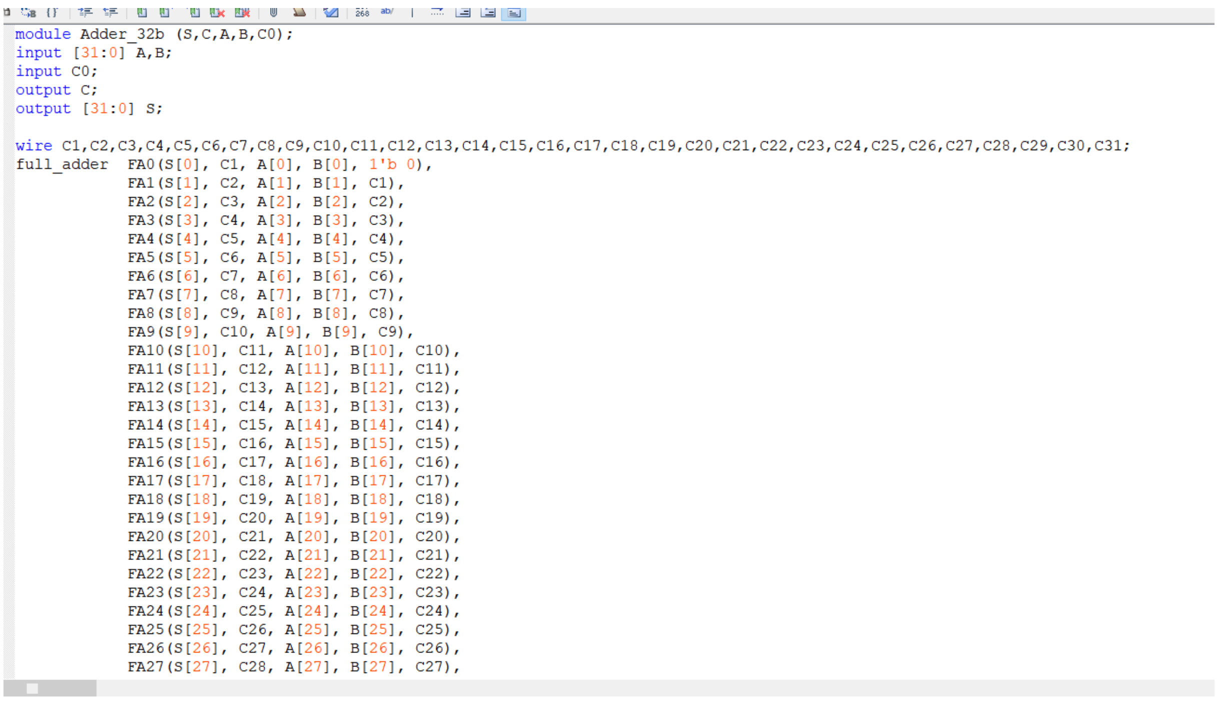
ADD operation:

ADD_test

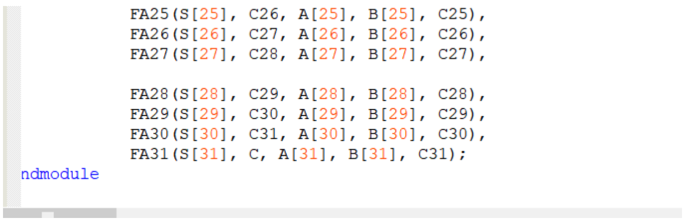


```
VSIM 10> run
# time =  0, A =0000000000000000001111111111110, B=00000001111111100011100000000001, S=000, Result=00000001111111100111011111111111
# time = 10, A =00000000111111100000000000000000, B=00000111111111111111000111000001, S=000, Result=00001000111111011111000111000001
```

32 bit adder: hard codded for 32 bit numbers. It does operation bit-by-bit



```
module Adder_32b (S,C,A,B,C0);
input [31:0] A,B;
input C0;
output C;
output [31:0] S;

wire C1,C2,C3,C4,C5,C6,C7,C8,C9,C10,C11,C12,C13,C14,C15,C16,C17,C18,C19,C20,C21,C22,C23,C24,C25,C26,C27,C28,C29,C30,C31;
full_adder  FA0(S[0], C1, A[0], B[0], 1'b 0),
            FA1(S[1], C2, A[1], B[1], C1),
            FA2(S[2], C3, A[2], B[2], C2),
            FA3(S[3], C4, A[3], B[3], C3),
            FA4(S[4], C5, A[4], B[4], C4),
            FA5(S[5], C6, A[5], B[5], C5),
            FA6(S[6], C7, A[6], B[6], C6),
            FA7(S[7], C8, A[7], B[7], C7),
            FA8(S[8], C9, A[8], B[8], C8),
            FA9(S[9], C10, A[9], B[9], C9),
            FA10(S[10], C11, A[10], B[10], C10),
            FA11(S[11], C12, A[11], B[11], C11),
            FA12(S[12], C13, A[12], B[12], C12),
            FA13(S[13], C14, A[13], B[13], C13),
            FA14(S[14], C15, A[14], B[14], C14),
            FA15(S[15], C16, A[15], B[15], C15),
            FA16(S[16], C17, A[16], B[16], C16),
            FA17(S[17], C18, A[17], B[17], C17),
            FA18(S[18], C19, A[18], B[18], C18),
            FA19(S[19], C20, A[19], B[19], C19),
            FA20(S[20], C21, A[20], B[20], C20),
            FA21(S[21], C22, A[21], B[21], C21),
            FA22(S[22], C23, A[22], B[22], C22),
            FA23(S[23], C24, A[23], B[23], C23),
            FA24(S[24], C25, A[24], B[24], C24),
            FA25(S[25], C26, A[25], B[25], C25),
            FA26(S[26], C27, A[26], B[26], C26),
            FA27(S[27], C28, A[27], B[27], C27),
```

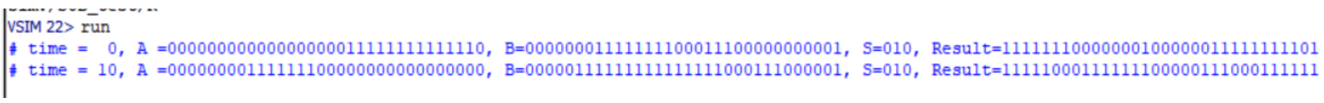


```
            FA25(S[25], C26, A[25], B[25], C25),
            FA26(S[26], C27, A[26], B[26], C26),
            FA27(S[27], C28, A[27], B[27], C27),

            FA28(S[28], C29, A[28], B[28], C28),
            FA29(S[29], C30, A[29], B[29], C29),
            FA30(S[30], C31, A[30], B[30], C30),
            FA31(S[31], C, A[31], B[31], C31);
ndmodule
```

Full_adder is a different custom module that achieve full adder operation. Also it uses half_adder inside.

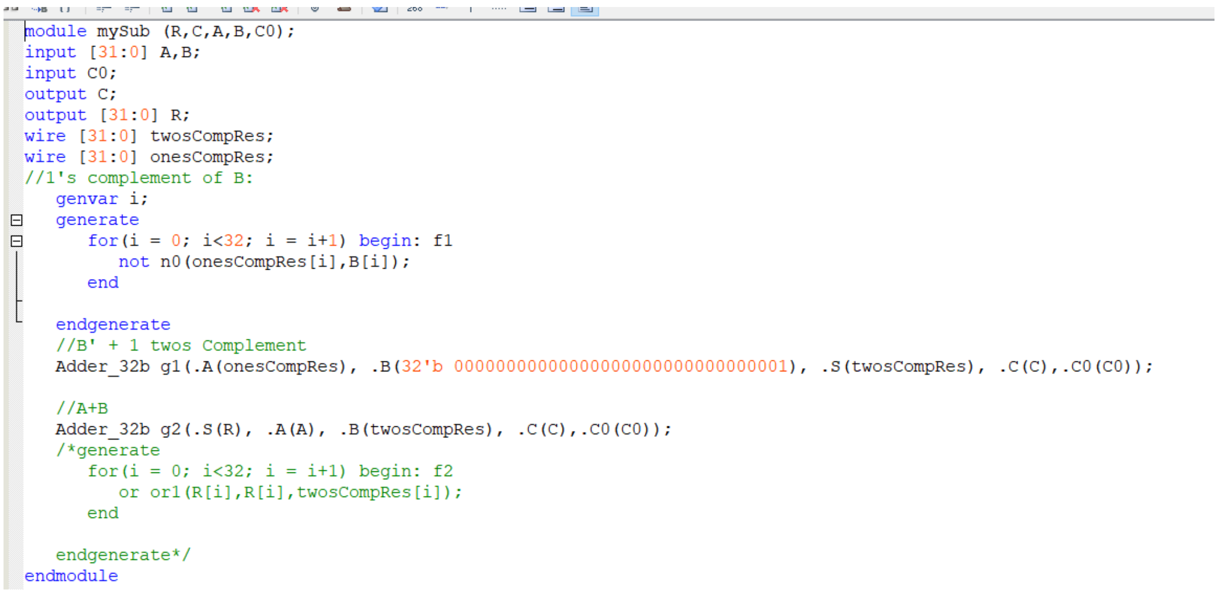
SUB operation:

SUB_test



```
VSIM 22> run
# time =  0, A =0000000000000000001111111111110, B=00000001111111100011100000000001, S=010, Result=11111110000000100000011111111101
# time = 10, A =00000000111111100000000000000000, B=00000111111111111111000111000001, S=010, Result=11111000111111100000111000111111
```

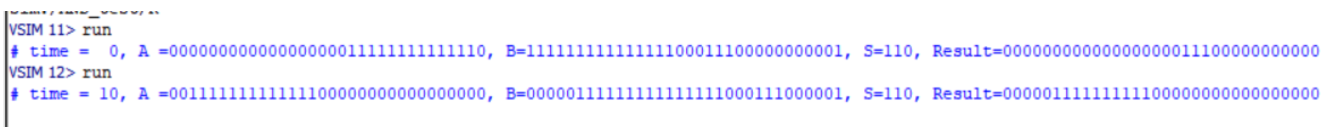
First, does one’s complement after that to reach two’s complement, adds 1 to number. After these 2 operation it’s is a simple adding operation



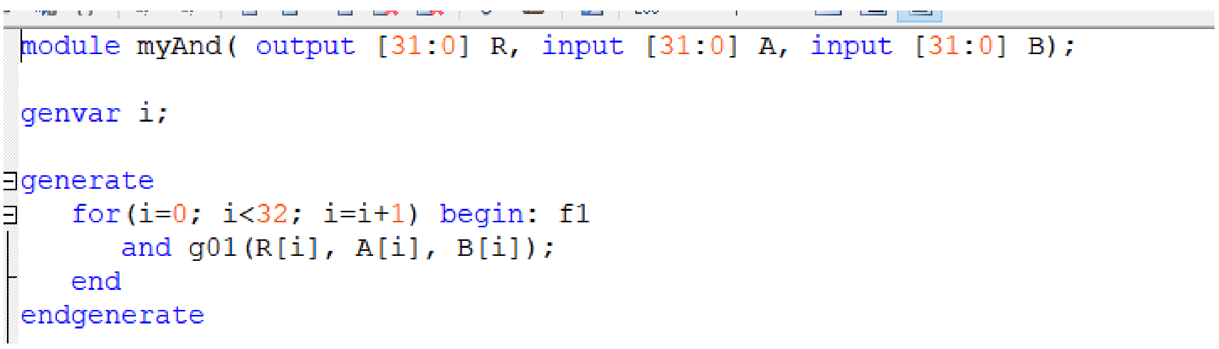
```
module mySub (R,C,A,B,C0);
input [31:0] A,B;
input C0;
output C;
output [31:0] R;
wire [31:0] twosCompRes;
wire [31:0] onesCompRes;
//1's complement of B:
   genvar i;
   generate
      for(i = 0; i<32; i = i+1) begin: f1
         not n0(onesCompRes[i],B[i]);
      end

   endgenerate
   //B' + 1 twos Complement
   Adder_32b g1(.A(onesCompRes), .B(32'b 00000000000000000000000000000001), .S(twosCompRes), .C(C),.C0(C0));

   //A+B
   Adder_32b g2(.S(R), .A(A), .B(twosCompRes), .C(C),.C0(C0));
   /*generate
      for(i = 0; i<32; i = i+1) begin: f2
         or or1(R[i],R[i],twosCompRes[i]);
      end

   endgenerate*/
endmodule
```

AND operation:

AND_test.v



```
VSIM 11> run
# time =  0, A =00000000000000000011111111111110, B=11111111111111100011100000000001, S=110, Result=00000000000000000011100000000000
VSIM 12> run
# time = 10, A =00111111111111100000000000000000, B=00000111111111111111100011000001, S=110, Result=00000111111111110000000000000000
```

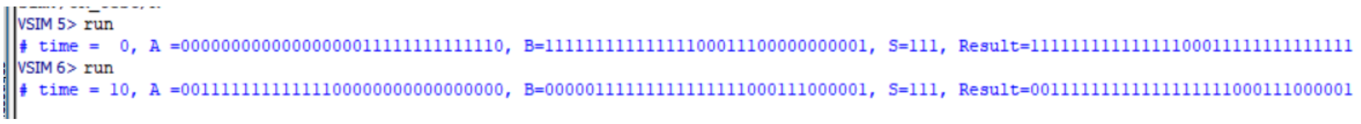


```
module myAnd( output [31:0] R, input [31:0] A, input [31:0] B);

genvar i;

generate
    for(i=0; i<32; i=i+1) begin: f1
        and g01(R[i], A[i], B[i]);
    end
endgenerate
```

And operation takes 32 bit numbers and performs bit-by-bit

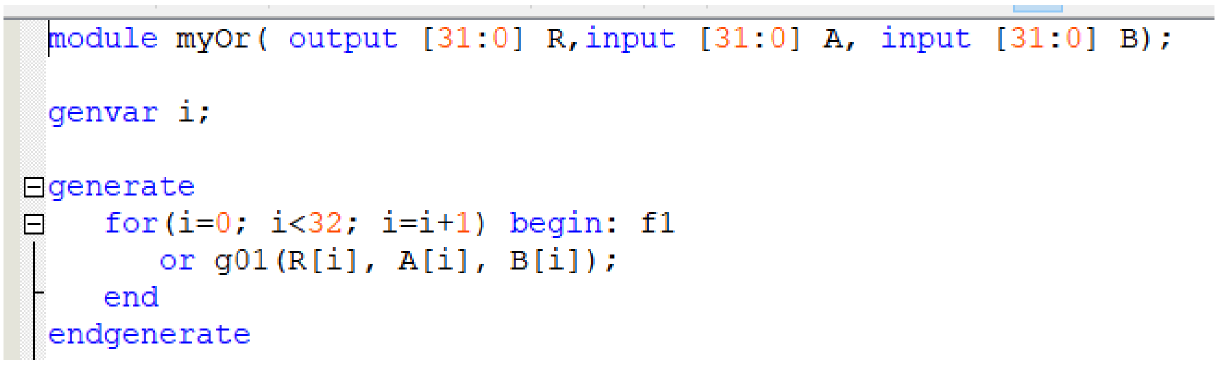
OR operation:

OR_test.v



```
VSIM 5> run
# time =  0, A =00000000000000000011111111111110, B=11111111111111100011100000000001, S=111, Result=11111111111111100011111111111111
VSIM 6> run
# time = 10, A =00111111111111110000000000000000, B=00000111111111111111000111000001, S=111, Result=00111111111111111111000111000001
```

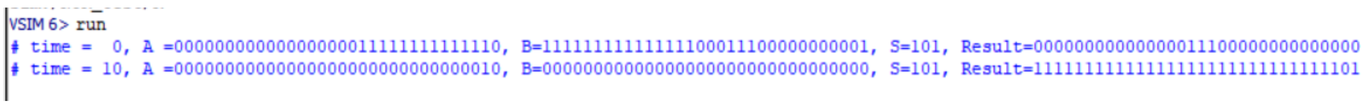
Same with and operation.



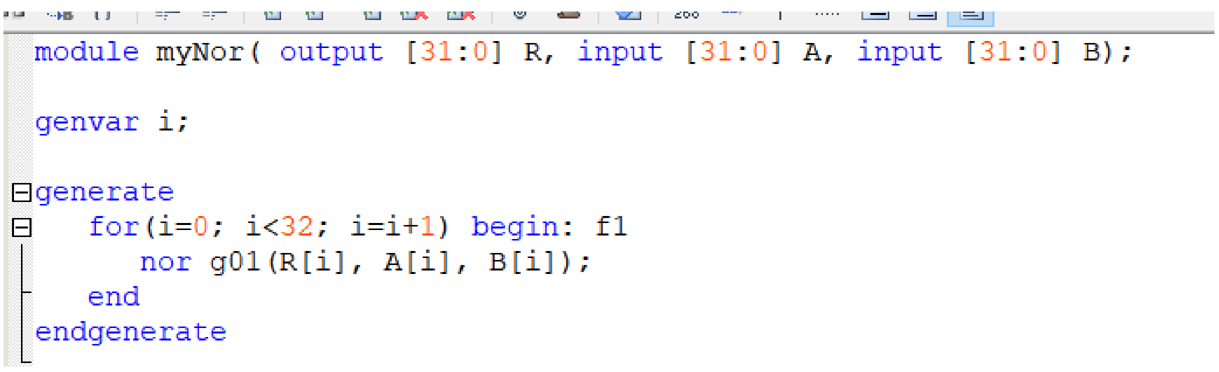
```
module myOr( output [31:0] R,input [31:0] A, input [31:0] B);

genvar i;

generate
    for(i=0; i<32; i=i+1) begin: f1
        or g01(R[i], A[i], B[i]);
    end
endgenerate
```

NOR operation:

NOR_test.v



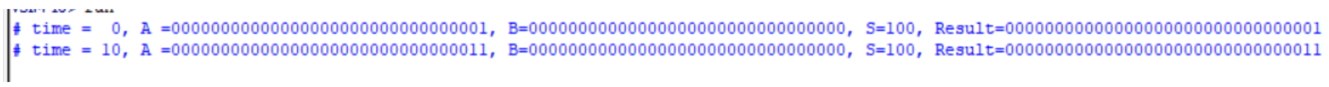
```
VSIM 6> run
# time =  0, A =00000000000000000011111111111110, B=11111111111111100011100000000001, S=101, Result=00000000000000011100000000000000
# time = 10, A =00000000000000000000000000000010, B=00000000000000000000000000000000, S=101, Result=11111111111111111111111111111101
```



```
module myNor( output [31:0] R, input [31:0] A, input [31:0] B);

genvar i;

generate
    for(i=0; i<32; i=i+1) begin: f1
        nor g01(R[i], A[i], B[i]);
    end
endgenerate
```

Slt operation:

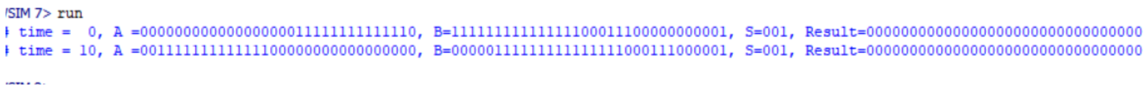
Slt_test.v



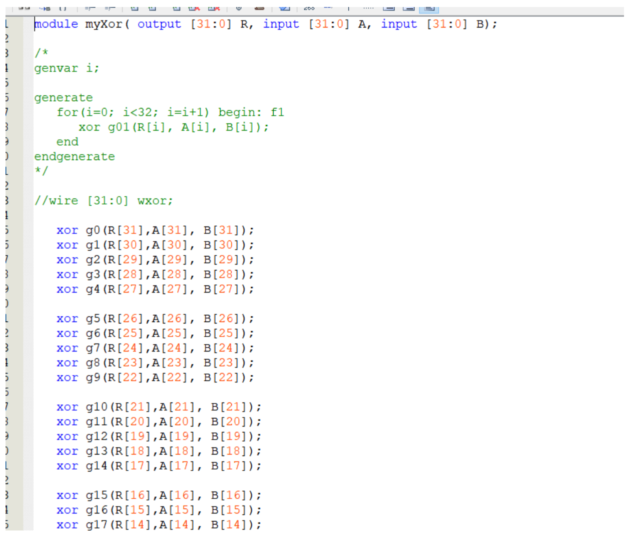
```
# time =  0, A =00000000000000000000000000000001, B=00000000000000000000000000000000, S=100, Result=00000000000000000000000000000001
# time = 10, A =00000000000000000000000000000011, B=00000000000000000000000000000000, S=100, Result=00000000000000000000000000000011
```

Xor operation:

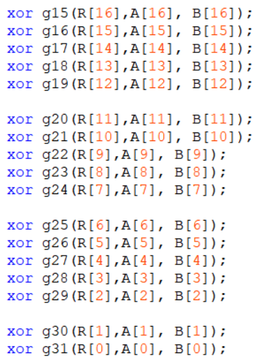
Doesn’t work hocam :[



```
/SIM 7> run
# time =  0, A =00000000000000000011111111111110, B=11111111111111100011100000000001, S=001, Result=00000000000000000000000000000000
# time = 10, A =00111111111111100000000000000000, B=00000111111111111111000111000001, S=001, Result=00000000000000000000000000000000
```

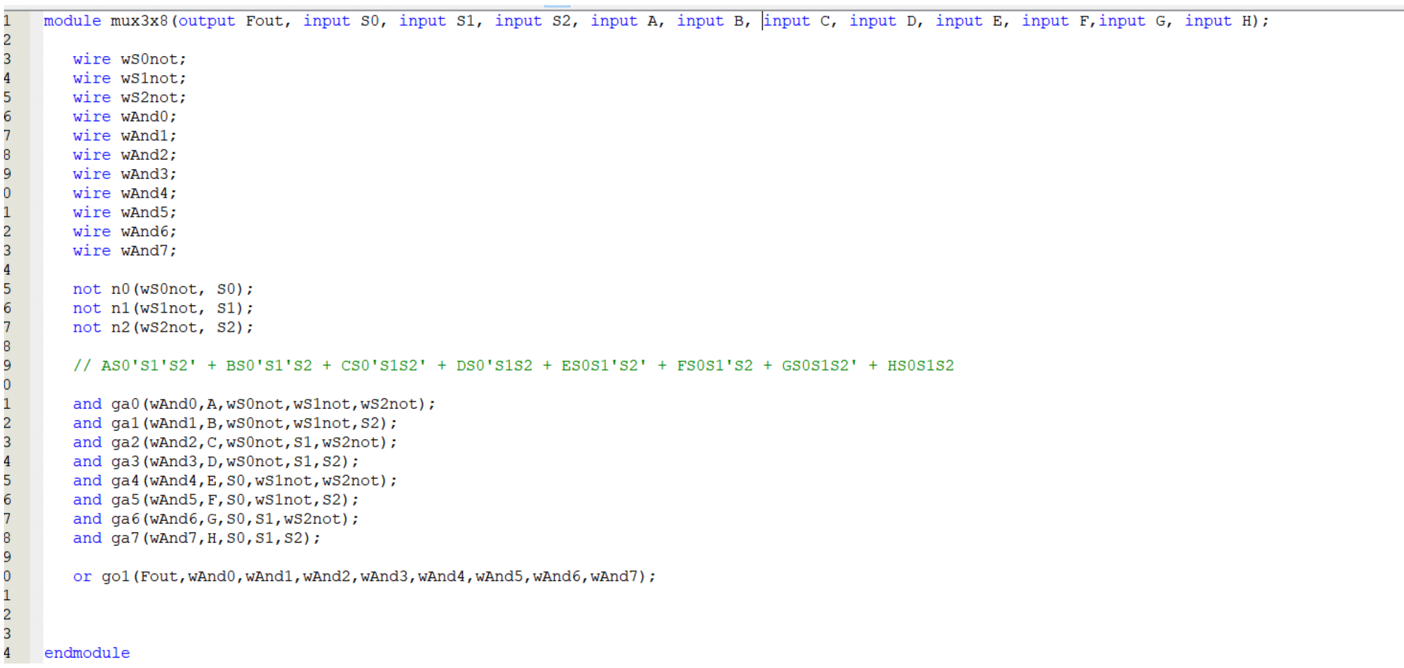


```verilog
module myXor( output [31:0] R, input [31:0] A, input [31:0] B);

/*
genvar i;

generate
   for(i=0; i<32; i=i+1) begin: f1
      xor g01(R[i], A[i], B[i]);
   end
endgenerate
*/

//wire [31:0] wxor;

   xor g0(R[31],A[31], B[31]);
   xor g1(R[30],A[30], B[30]);
   xor g2(R[29],A[29], B[29]);
   xor g3(R[28],A[28], B[28]);
   xor g4(R[27],A[27], B[27]);

   xor g5(R[26],A[26], B[26]);
   xor g6(R[25],A[25], B[25]);
   xor g7(R[24],A[24], B[24]);
   xor g8(R[23],A[23], B[23]);
   xor g9(R[22],A[22], B[22]);

   xor g10(R[21],A[21], B[21]);
   xor g11(R[20],A[20], B[20]);
   xor g12(R[19],A[19], B[19]);
   xor g13(R[18],A[18], B[18]);
   xor g14(R[17],A[17], B[17]);

   xor g15(R[16],A[16], B[16]);
   xor g16(R[15],A[15], B[15]);
   xor g17(R[14],A[14], B[14]);
```



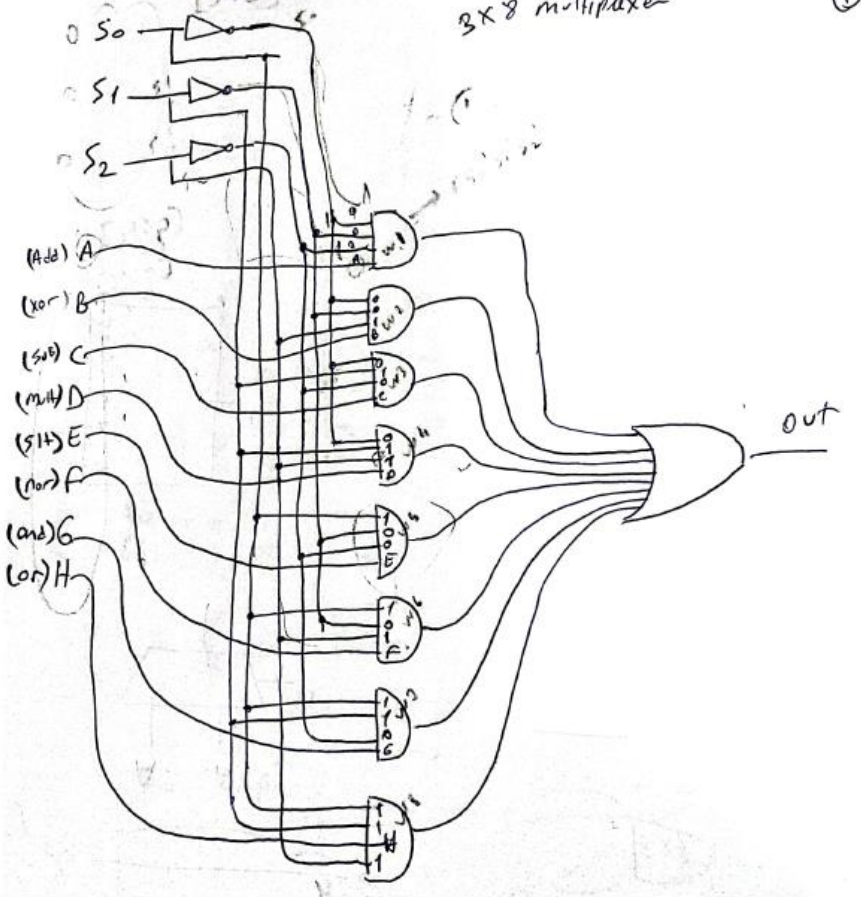
```verilog
xor g15(R[16],A[16], B[16]);
xor g16(R[15],A[15], B[15]);
xor g17(R[14],A[14], B[14]);
xor g18(R[13],A[13], B[13]);
xor g19(R[12],A[12], B[12]);

xor g20(R[11],A[11], B[11]);
xor g21(R[10],A[10], B[10]);
xor g22(R[9],A[9], B[9]);
xor g23(R[8],A[8], B[8]);
xor g24(R[7],A[7], B[7]);

xor g25(R[6],A[6], B[6]);
xor g26(R[5],A[5], B[5]);
xor g27(R[4],A[4], B[4]);
xor g28(R[3],A[3], B[3]);
xor g29(R[2],A[2], B[2]);

xor g30(R[1],A[1], B[1]);
xor g31(R[0],A[0], B[0]);
```

Multiplexer:



```
module mux3x8(output Fout, input S0, input S1, input S2, input A, input B, input C, input D, input E, input F,input G, input H);

   wire wS0not;
   wire wS1not;
   wire wS2not;
   wire wAnd0;
   wire wAnd1;
   wire wAnd2;
   wire wAnd3;
   wire wAnd4;
   wire wAnd5;
   wire wAnd6;
   wire wAnd7;

   not n0(wS0not, S0);
   not n1(wS1not, S1);
   not n2(wS2not, S2);

   // AS0'S1'S2' + BS0'S1'S2 + CS0'S1S2' + DS0'S1S2 + ES0S1'S2' + FS0S1'S2 + GS0S1S2' + HS0S1S2

   and ga0(wAnd0,A,wS0not,wS1not,wS2not);
   and ga1(wAnd1,B,wS0not,wS1not,S2);
   and ga2(wAnd2,C,wS0not,S1,wS2not);
   and ga3(wAnd3,D,wS0not,S1,S2);
   and ga4(wAnd4,E,S0,wS1not,wS2not);
   and ga5(wAnd5,F,S0,wS1not,S2);
   and ga6(wAnd6,G,S0,S1,wS2not);
   and ga7(wAnd7,H,S0,S1,S2);

   or go1(Fout,wAnd0,wAnd1,wAnd2,wAnd3,wAnd4,wAnd5,wAnd6,wAnd7);



endmodule
```

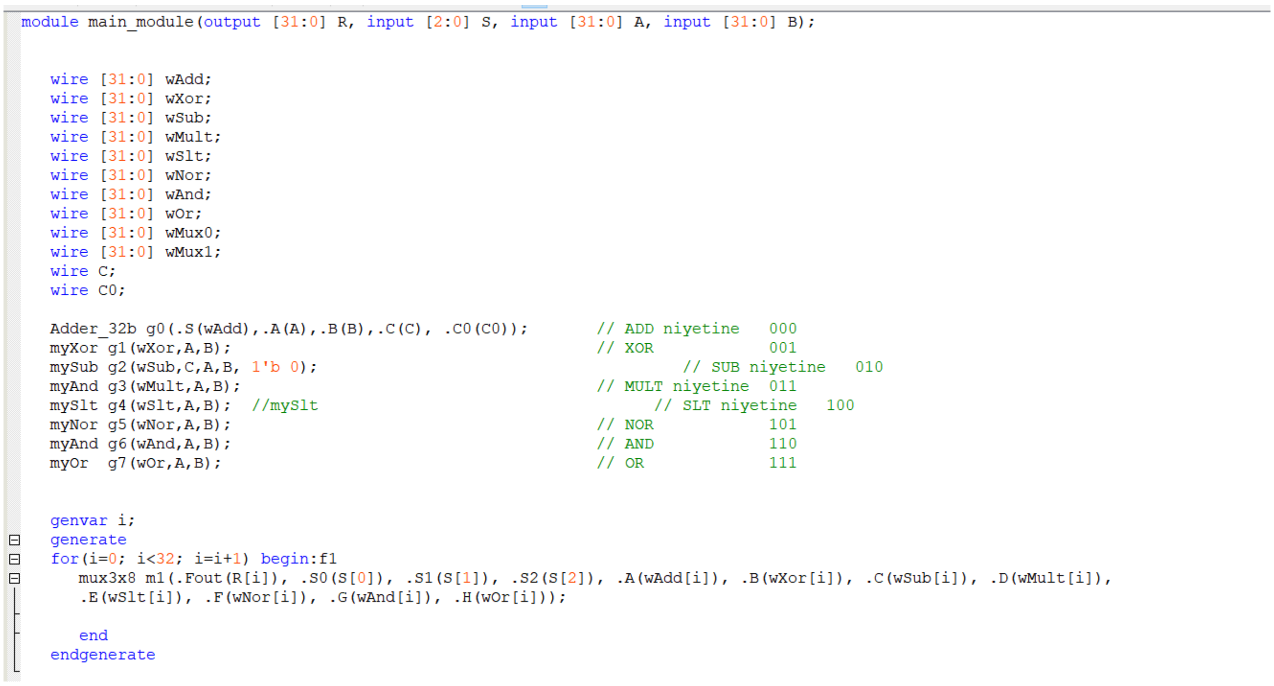
3X8 multiplexer ①



$$= A s_0' s_1' s_2' + B s_0' s_1' s_2 + C s_0' s_1 s_2' + D s_0' s_1 s_2 + E s_0 s_1' s_2'$$
$$+ F s_0 s_1' s_2 + G s_0 s_1 s_2' + H s_0 s_1 s_2$$

$\gg$ No simplification

This is the main program. First I fill the 32 bit wires and carry operation bit-by-bit inside a for loop



```
module main_module(output [31:0] R, input [2:0] S, input [31:0] A, input [31:0] B);


   wire [31:0] wAdd;
   wire [31:0] wXor;
   wire [31:0] wSub;
   wire [31:0] wMult;
   wire [31:0] wSlt;
   wire [31:0] wNor;
   wire [31:0] wAnd;
   wire [31:0] wOr;
   wire [31:0] wMux0;
   wire [31:0] wMux1;
   wire C;
   wire C0;

   Adder_32b g0(.S(wAdd),.A(A),.B(B),.C(C), .C0(C0));       // ADD niyetine   000
   myXor g1(wXor,A,B);                                      // XOR            001
   mySub g2(wSub,C,A,B, 1'b 0);                                      // SUB niyetine   010
   myAnd g3(wMult,A,B);                                     // MULT niyetine  011
   mySlt g4(wSlt,A,B);  //mySlt                                   // SLT niyetine   100
   myNor g5(wNor,A,B);                                      // NOR            101
   myAnd g6(wAnd,A,B);                                      // AND            110
   myOr  g7(wOr,A,B);                                       // OR             111


   genvar i;
   generate
   for(i=0; i<32; i=i+1) begin:f1
      mux3x8 m1(.Fout(R[i]), .S0(S[0]), .S1(S[1]), .S2(S[2]), .A(wAdd[i]), .B(wXor[i]), .C(wSub[i]), .D(wMult[i]),
      .E(wSlt[i]), .F(wNor[i]), .G(wAnd[i]), .H(wOr[i]));

      end
   endgenerate
```