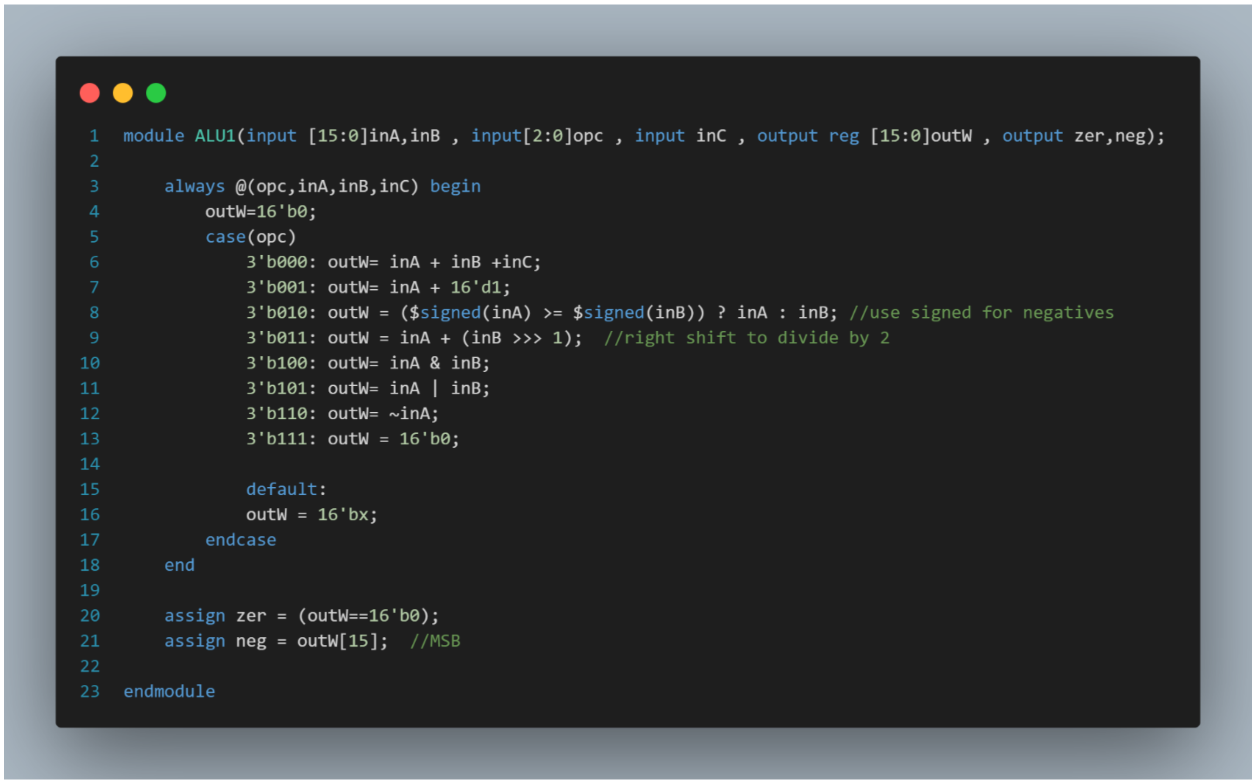
Problem 1

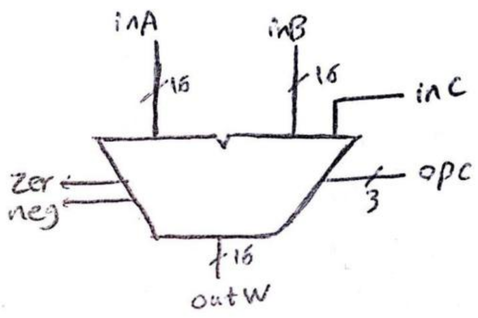
Pre_synthesize:



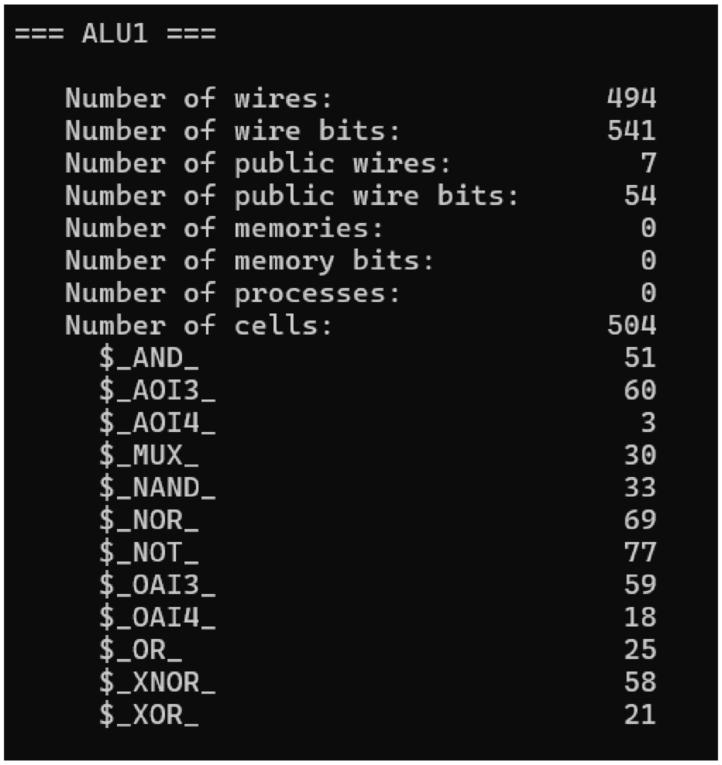
```
module ALU1(input [15:0]inA,inB , input[2:0]opc , input inC , output reg [15:0]outW , output zer,neg);

    always @(opc,inA,inB,inC) begin
        outW=16'b0;
        case(opc)
            3'b000: outW= inA + inB +inC;
            3'b001: outW= inA + 16'd1;
            3'b010: outW = ($signed(inA) >= $signed(inB)) ? inA : inB; //use signed for negatives
            3'b011: outW = inA + (inB >>> 1);  //right shift to divide by 2
            3'b100: outW= inA & inB;
            3'b101: outW= inA | inB;
            3'b110: outW= ~inA;
            3'b111: outW = 16'b0;

            default:
            outW = 16'bx;
        endcase
    end

    assign zer = (outW==16'b0);
    assign neg = outW[15];  //MSB

endmodule
```

By synthesizing the code with Yosys we will get post_synthesized code (which include the library gates). Post_synthesizesd file is uploaded in the Zip file.

Problem 1 diagram:

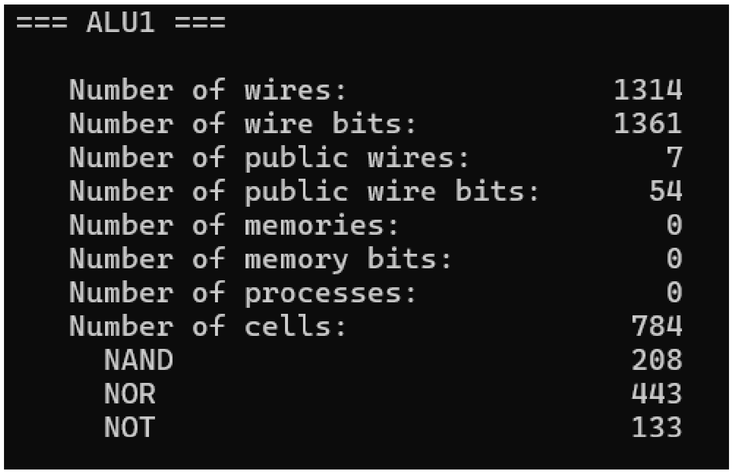


Gates used for ALU1.v



```
=== ALU1 ===

   Number of wires:                494
   Number of wire bits:            541
   Number of public wires:           7
   Number of public wire bits:      54
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                504
     $_AND_                         51
     $_AOI3_                        60
     $_AOI4_                         3
     $_MUX_                         30
     $_NAND_                        33
     $_NOR_                         69
     $_NOT_                         77
     $_OAI3_                        59
     $_OAI4_                        18
     $_OR_                          25
     $_XNOR_                        58
     $_XOR_                         21
```

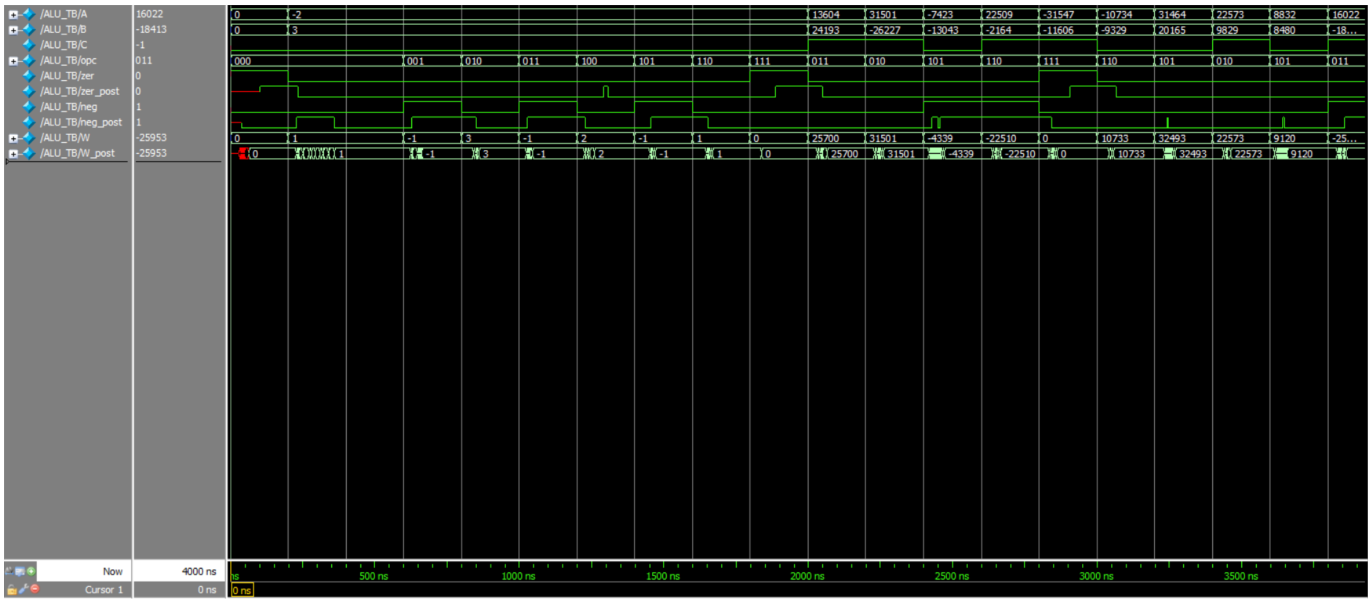
Gates used for ALU1_synth.v (library gates)



```
=== ALU1 ===

   Number of wires:               1314
   Number of wire bits:           1361
   Number of public wires:           7
   Number of public wire bits:      54
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                784
     NAND                          208
     NOR                           443
     NOT                           133
```

Gate count is higher in the synthesized version because the library only includes limited primitive cells.

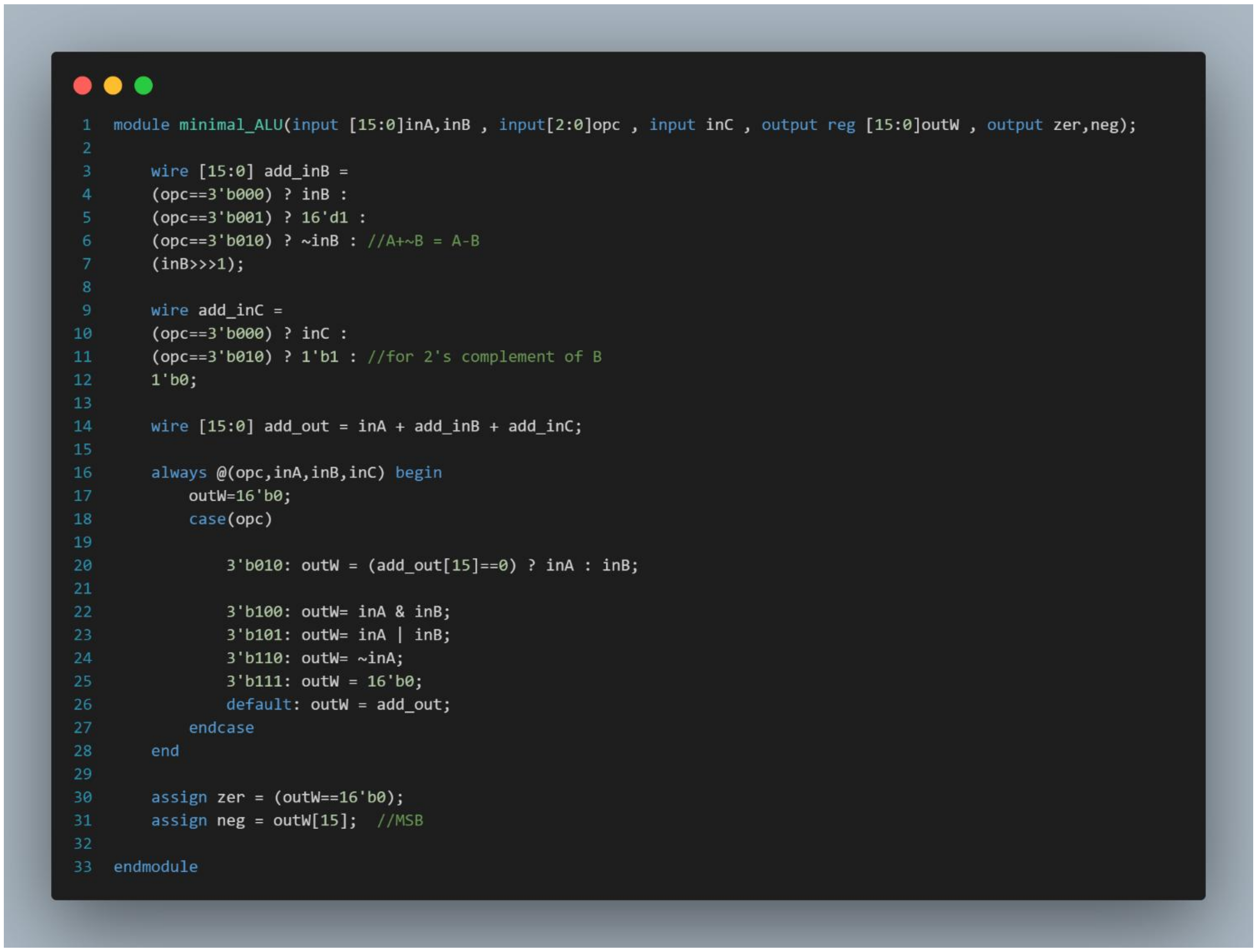
Here is the waveform of the test bench that I put at the end of report, to compare part a and c (W and W_post)



As we can see here the outputs match exactly. The only difference is delay, because library cells have propagation delays. Behavioral simulation is faster and has no delay but gate-level simulation is slower and includes realistic propagation delay.

Problem 2

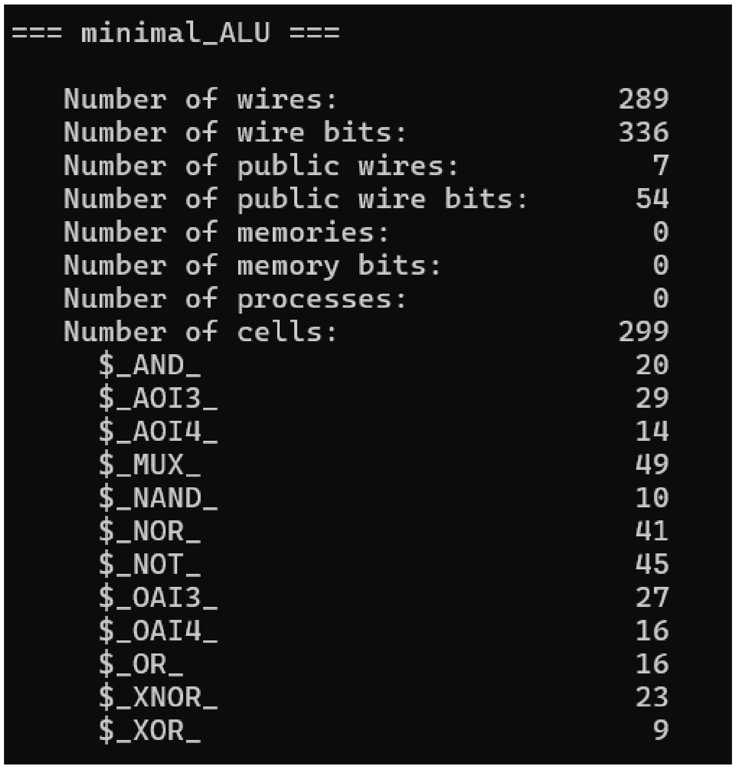
Pre_synthesizes:



```verilog
module minimal_ALU(input [15:0]inA,inB , input[2:0]opc , input inC , output reg [15:0]outW , output zer,neg);

    wire [15:0] add_inB =
    (opc==3'b000) ? inB :
    (opc==3'b001) ? 16'd1 :
    (opc==3'b010) ? ~inB : //A+~B = A-B
    (inB>>>1);

    wire add_inC =
    (opc==3'b000) ? inC :
    (opc==3'b010) ? 1'b1 : //for 2's complement of B
    1'b0;

    wire [15:0] add_out = inA + add_inB + add_inC;

    always @(opc,inA,inB,inC) begin
        outW=16'b0;
        case(opc)

            3'b010: outW = (add_out[15]==0) ? inA : inB;

            3'b100: outW= inA & inB;
            3'b101: outW= inA | inB;
            3'b110: outW= ~inA;
            3'b111: outW = 16'b0;
            default: outW = add_out;
        endcase
    end

    assign zer = (outW==16'b0);
    assign neg = outW[15];  //MSB

endmodule
```

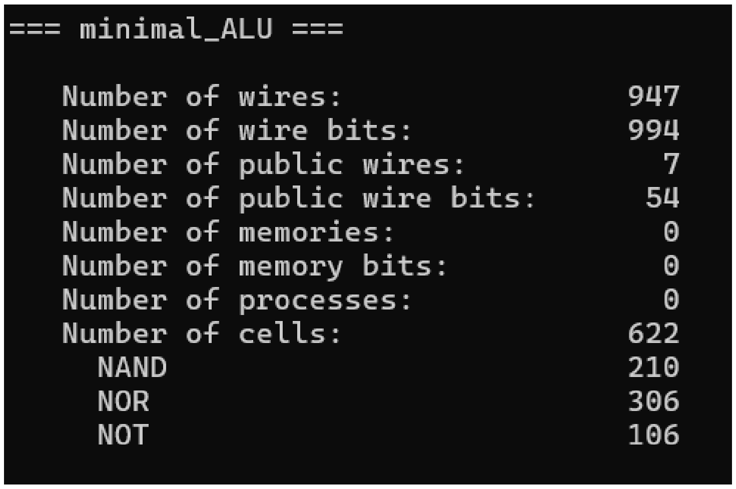
In this version I share one adder for first four opcodes. For each of these opcodes, I only change the inputs to the adder (add_inB and add_inC), so the ALU can perform different operations with the same hardware. add_inB, add_inC, and add_out choose the correct inputs. This reduces the number of gates. Therefore, we must have fewer amount of gates in this case rather than the first problem.

Gates used for minimal_ALU.v



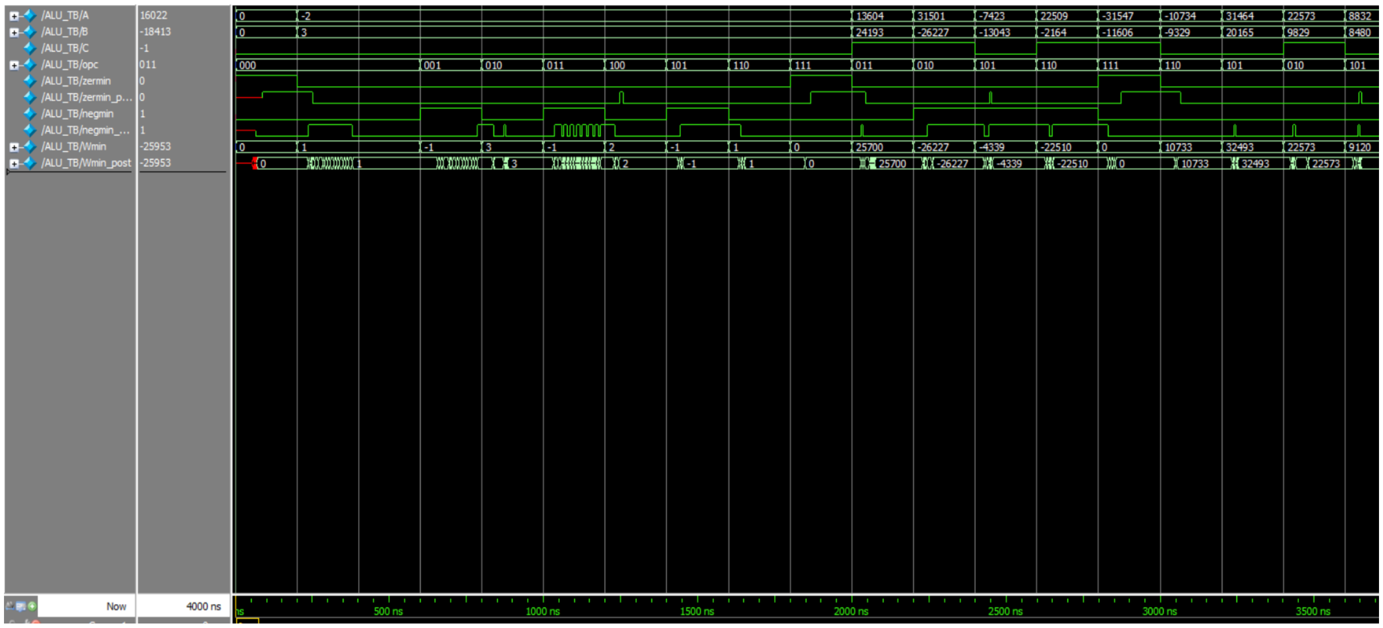
```
=== minimal_ALU ===

   Number of wires:                289
   Number of wire bits:            336
   Number of public wires:           7
   Number of public wire bits:      54
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                299
     $_AND_                         20
     $_AOI3_                        29
     $_AOI4_                        14
     $_MUX_                         49
     $_NAND_                        10
     $_NOR_                         41
     $_NOT_                         45
     $_OAI3_                        27
     $_OAI4_                        16
     $_OR_                          16
     $_XNOR_                        23
     $_XOR_                          9
```

Gates used for minimalALU_synth.v



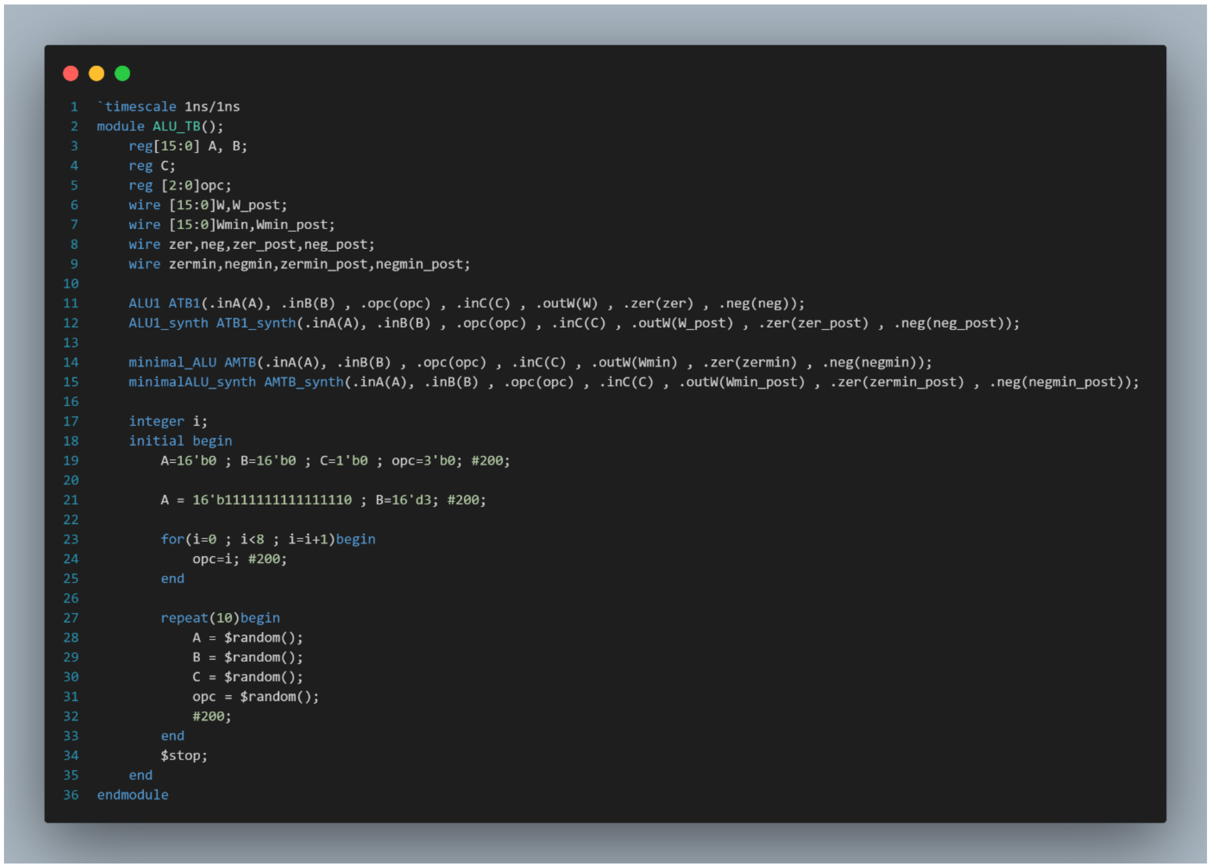
```
=== minimal_ALU ===

   Number of wires:                947
   Number of wire bits:            994
   Number of public wires:           7
   Number of public wire bits:      54
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                622
     NAND                          210
     NOR                           306
     NOT                           106
```

Obviously the post_synthesized one has more gates because we have limited available gates in our library.



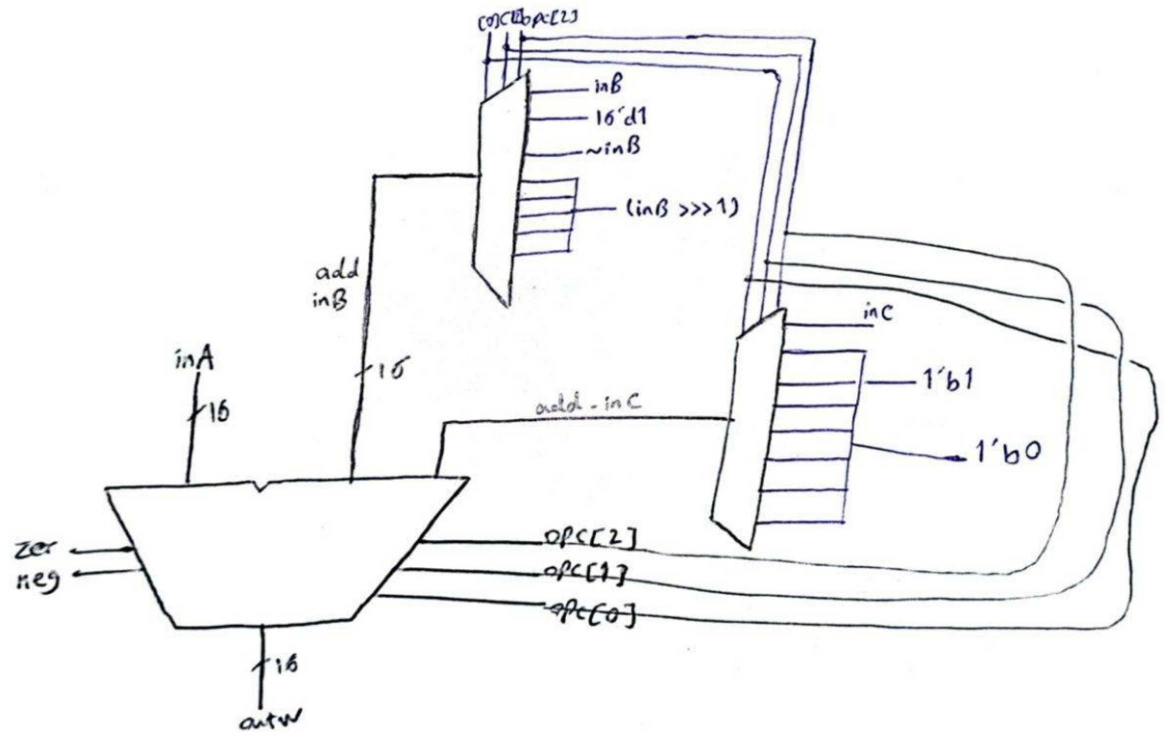
Just like the first problem the post_synthesizes outputs have delay.

The test bench of 2 problems:



```
`timescale 1ns/1ns
module ALU_TB();
    reg[15:0] A, B;
    reg C;
    reg [2:0]opc;
    wire [15:0]W,W_post;
    wire [15:0]Wmin,Wmin_post;
    wire zer,neg,zer_post,neg_post;
    wire zermin,negmin,zermin_post,negmin_post;

    ALU1 ATB1(.inA(A), .inB(B) , .opc(opc) , .inC(C) , .outW(W) , .zer(zer) , .neg(neg));
    ALU1_synth ATB1_synth(.inA(A), .inB(B) , .opc(opc) , .inC(C) , .outW(W_post) , .zer(zer_post) , .neg(neg_post));

    minimal_ALU AMTB(.inA(A), .inB(B) , .opc(opc) , .inC(C) , .outW(Wmin) , .zer(zermin) , .neg(negmin));
    minimalALU_synth AMTB_synth(.inA(A), .inB(B) , .opc(opc) , .inC(C) , .outW(Wmin_post) , .zer(zermin_post) , .neg(negmin_post));

    integer i;
    initial begin
        A=16'b0 ; B=16'b0 ; C=1'b0 ; opc=3'b0; #200;

        A = 16'b1111111111111110 ; B=16'd3; #200;

        for(i=0 ; i<8 ; i=i+1)begin
            opc=i; #200;
        end

        repeat(10)begin
            A = $random();
            B = $random();
            C = $random();
            opc = $random();
            #200;
        end
        $stop;
    end
endmodule
```

Problem 2 diagram:



for 010 opcode the most significant bit of add_out must be connected to a comparator and the output of it should connects to a selector(to choose between A and B)

Problem 3:

In both designs, the outputs are similar but their structural code has delay because the synthesized netlists use real library cells with non zero propagation time. The behavioral ALU synthesizes have more hardware because each opcode is implemented separately, while the structural one has less hardware because it just uses a single adder for first opcodes.

Shared data path for the second problem makes its critical path longer than the first one. Therefore, the hardware_oriented design is smaller but slower than behavioral ALU.