GEBZE TECHNICAL UNIVERSİTY

COMPUTER ENGINEERING

COMPUTER ORGANIZATION

CSE 331 – 2019 FALL

HW2 REPORT

SILA BENGİSU YÜKSEKLİ

1801042877

➢ *Explanation of Homework and Codes*

In this homework, I implemented a 32-BIT ALU. This implementation is made by using structural verilog. This 32-BIT ALU can provide some operations which are OR, AND, ADD, SUB and SET-ON-LESS-THAN. To support these operations, 4x1 MUX Module, 1-BIT ALU and Gates are used . Using of XOR gate is forbidden. That's way I implemented my own xor gate module. I also implemented my own 4x1 MUX and 1-BIT ALU modules. The codes I have written are as follows:

```
module XOR_GATE(out,input1,input2);
        input input1,input2;
        output out;
        wire input1_inverse, input2_inverse, tmp_out1, tmp_out2;
        not inputInverse1 (input1_inverse,input1);
        not inputInverse2 (input2_inverse,input2);
        and firstOut (tmp_out1,input1,input2_inverse);
        and secondOut (tmp_out2,input2,input1_inverse);
        or finalOut (out,tmp_out1,tmp_out2);
endmodule
```

This code provides using of xor gate. Formula for xor gate equals to input1'.input2 + input1.input2'. So I used two AND, two NOT gates and one OR gate.

```
module MUX4to1(Y,S0,S1,D0,D1,D2,D3);
        input S0,S1,D0,D1,D2,D3;
        output Y;
        wire S0_inverse, S1_inverse, tmp_and0, tmp_and1, tmp_and2, tmp_and3;
        not tmpInverse1 (S0_inverse,S0);
        not tmpInverse2 (S1_inverse,S1);
        and tmpOut0 (tmp_and0,S0_inverse,S1_inverse,D0);
        and tmpOut1 (tmp_and1,S0,S1_inverse,D1);
        and tmpOut2 (tmp_and2,S0_inverse,S1,D2);
        and tmpOut3 (tmp_and3,S0,S1,D3);
        or  finalOut (Y,tmp_and0,tmp_and1,tmp_and2,tmp_and3);
endmodule
```

This code provides using of 4x1 MUX. Output of the 4x1 mux equals to D0.S1'.S0' + D1.S1'.S0 + D2.S1.S0' + D3.S1.S0. So I used four AND, two NOT gates and one OR gate.

```
module _1BIT_ALU (R,Co,signBit,A,B,AlUop0,AlUop1,AlUop2,Ci,Less);
        input A,B,AlUop0,AlUop1,AlUop2,Ci,Less;
        output R,Co,signBit;
        wire tmp_xorOut, tmp_first_or_gate, tmp_first_and_gate, tmp_first_inverse,
```

```
tmp_second_and_gate;
        wire tmp_third_and_gate, tmp_second_or_gate, tmp_second_inverse;
        XOR_GATE xorGate (tmp_xorOut,B,AlUop2);
        or firstOrGate (tmp_first_or_gate,tmp_xorOut,A);
        and firstAndGate (tmp_first_and_gate,A,tmp_xorOut);
        not firstInverse (tmp_first_inverse,tmp_first_and_gate);
        and secondAndGate (tmp_second_and_gate,tmp_first_inverse,tmp_first_or_gate);
        and thirdAndGate (tmp_third_and_gate,tmp_second_and_gate,Ci);
        or secondOrGate (tmp_second_or_gate,tmp_second_and_gate,Ci);
        not secondInverse (tmp_second_inverse,tmp_third_and_gate);
        or Co_Out (Co,tmp_third_and_gate,tmp_first_and_gate);
        and fourhAndGate (signBit,tmp_second_inverse,tmp_second_or_gate);
        MUX4to1 selection (R,AlUop0,AlUop1,tmp_first_and_gate,signBit,tmp_first_or_gate,Less);
endmodule
```

This code provides using of 1-BIT ALU. I used four AND, two NOT and three OR Gates and one XOR gate in this design. I also used my MUX design. I send select bits and other four inputs to the mux to make a selection.

```
module _32BIT_ALU (R,C31,V,Set,A,B,C0,ALUop0,ALUop1);
        input [31:0] A,B;
        input C0,ALUop0,ALUop1;
        output [31:0] R;
        output C31,V,Set;
        wire [30:0] C;
        wire [31:0] sign;

_1BIT_ALU ALU0(R[0],C[0],sign[0],A[0],B[0],ALUop0,ALUop1,C0,C0,Set),
ALU1(R[1],C[1],sign[1],A[1],B[1],ALUop0,ALUop1,C0,C[0],1'b0),
ALU2(R[2],C[2],sign[2],A[2],B[2],ALUop0,ALUop1,C0,C[1],1'b0),
ALU3(R[3],C[3],sign[3],A[3],B[3],ALUop0,ALUop1,C0,C[2],1'b0),
 ALU4(R[4],C[4],sign[4],A[4],B[4],ALUop0,ALUop1,C0,C[3],1'b0),
ALU5(R[5],C[5],sign[5],A[5],B[5],ALUop0,ALUop1,C0,C[4],1'b0),
ALU6(R[6],C[6],sign[6],A[6],B[6],ALUop0,ALUop1,C0,C[5],1'b0),
ALU7(R[7],C[7],sign[7],A[7],B[7],ALUop0,ALUop1,C0,C[6],1'b0),
 ALU8(R[8],C[8],sign[7],A[8],B[8],ALUop0,ALUop1,C0,C[7],1'b0),
ALU9(R[9],C[9],sign[9],A[9],B[9],ALUop0,ALUop1,C0,C[8],1'b0),
ALU10(R[10],C[10],sign[10],A[10],B[10],ALUop0,ALUop1,C0,C[9],1'b0),
ALU11(R[11],C[11],sign[11],A[11],B[11],ALUop0,ALUop1,C0,C[10],1'b0),
ALU12(R[12],C[12],sign[12],A[12],B[12],ALUop0,ALUop1,C0,C[11],1'b0),
ALU13(R[13],C[13],sign[13],A[13],B[13],ALUop0,ALUop1,C0,C[12],1'b0),
ALU14(R[14],C[14],sign[14],A[14],B[14],ALUop0,ALUop1,C0,C[13],1'b0),
ALU15(R[15],C[15],sign[15],A[15],B[15],ALUop0,ALUop1,C0,C[14],1'b0),
                                        ALU16(R[16],C[16],sign[16],A[16],B[16],ALUop0,ALUop1,C0,C[15],1'b0),
ALU17(R[17],C[17],sign[17],A[17],B[17],ALUop0,ALUop1,C0,C[16],1'b0),
ALU18(R[18],C[18],sign[18],A[18],B[18],ALUop0,ALUop1,C0,C[17],1'b0),
ALU19(R[19],C[19],sign[19],A[19],B[19],ALUop0,ALUop1,C0,C[18],1'b0),
```

```
ALU20(R[20],C[20],sign[20],A[20],B[20],ALUop0,ALUop1,C0,C[19],1'b0),
ALU21(R[21],C[21],sign[21],A[21],B[21],ALUop0,ALUop1,C0,C[20],1'b0),
ALU22(R[22],C[22],sign[22],A[22],B[22],ALUop0,ALUop1,C0,C[21],1'b0),
ALU23(R[23],C[23],sign[23],A[23],B[23],ALUop0,ALUop1,C0,C[22],1'b0),
ALU24(R[24],C[24],sign[24],A[24],B[24],ALUop0,ALUop1,C0,C[23],1'b0),
ALU25(R[25],C[25],sign[25],A[25],B[25],ALUop0,ALUop1,C0,C[24],1'b0),
ALU26(R[26],C[26],sign[26],A[26],B[26],ALUop0,ALUop1,C0,C[25],1'b0),
ALU27(R[27],C[27],sign[27],A[27],B[27],ALUop0,ALUop1,C0,C[26],1'b0),
ALU28(R[28],C[28],sign[28],A[28],B[28],ALUop0,ALUop1,C0,C[27],1'b0),
ALU29(R[29],C[29],sign[29],A[29],B[29],ALUop0,ALUop1,C0,C[28],1'b0),
ALU30(R[30],C[30],sign[30],A[30],B[30],ALUop0,ALUop1,C0,C[29],1'b0),
ALU31(R[31],C31,sign[31],A[31],B[31],ALUop0,ALUop1,C0,C[30],1'b0);
        XOR_GATE  overflow (V,C31,C[30]);
        XOR_GATE  signFlag (Set,V,sign[31]);
Endmodule
```

This code provides using of 32-BIT ALU. In this design I used my 1-BIT ALU design. C0 equals to ALUop2. Set is connected to the input of set of first 1-bit ALU. Input of Less of others is set as zero. This design will be done by combining 32 piece of 1-bit ALU. At the end, I calculated overflow by using this formula: V = Ci+1 xor Ci. Set output depends on overflow. In the last line, I calculated set output. This output indicates whether the input1 is less than other input. So I used two XOR gates additionally. 1-BIT ALU has seventeen gates,(so 32x17 + 2 xor Gates), 32-BIT ALU gate has 544 gates.

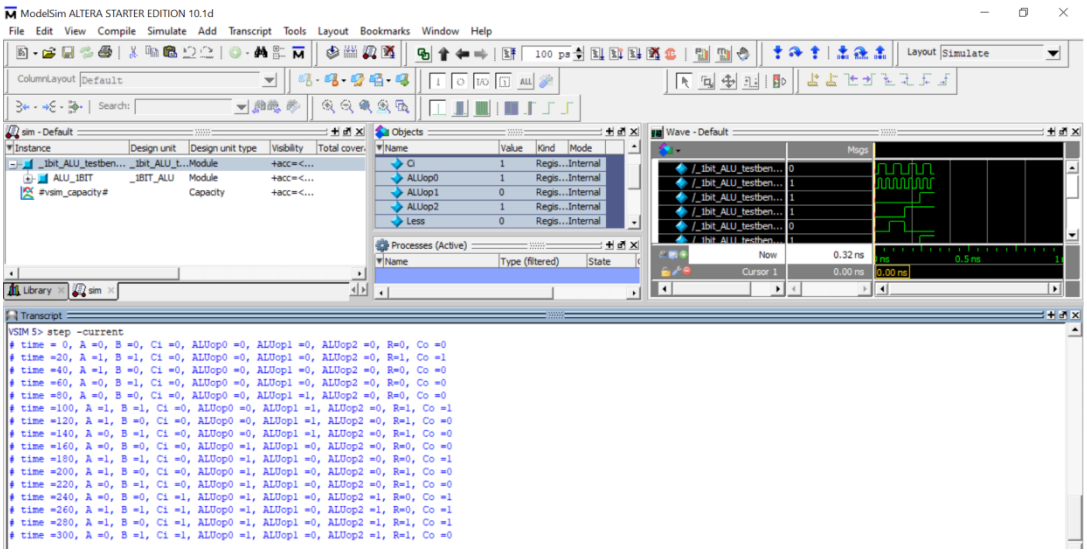
➢ *TEST CASES*

I wrote two test benches. After I have implemented 1-BIT ALU module, I wrote a test bench to check whether it works right. The tests I have written, are as follows:

```
module _1bit_ALU_testbench();

  reg A,B,Ci,ALUop0,ALUop1,ALUop2,Less;
  wire R,Co,V,signBit,Set;
_1BIT_ALU ALU_1BIT (R,Co,signBit,A,B,ALUop0,ALUop1,ALUop2,Ci,Less);

initial begin

        A = 1'b0;  B = 1'b0; ALUop0 = 1'b0; ALUop1 = 1'b0; ALUop2 = 1'b0; Ci = 1'b0; Less = 1'b0;
        #`DELAY;
        A = 1'b1;  B = 1'b1; ALUop0 = 1'b0; ALUop1 = 1'b0; ALUop2 = 1'b0; Ci = 1'b0; Less = 1'b
        #`DELAY;
        A = 1'b1;  B = 1'b0; Ci = 1'b0; ALUop0 = 1'b0; ALUop1 = 1'b0; ALUop2 = 1'b0; Less = 1'b0;
        #`DELAY;
        A = 1'b0;  B = 1'b1; Ci = 1'b0; ALUop0 = 1'b0; ALUop1 = 1'b0; ALUop2 = 1'b0; Less = 1'b0;
        #`DELAY;

        A = 1'b0;  B = 1'b0; ALUop0 = 1'b0; ALUop1 = 1'b1; ALUop2 = 1'b0; Ci = 1'b0; Less = 1'b0;
        #`DELAY;
        A = 1'b1;  B = 1'b1; ALUop0 = 1'b0; ALUop1 = 1'b1; ALUop2 = 1'b0; Ci = 1'b0; Less = 1'b0;
        #`DELAY;
```

```
A = 1'b1;  B = 1'b0; Ci = 1'b0; ALUop0 = 1'b0; ALUop1 = 1'b1; ALUop2 = 1'b0; Less = 1'b0;
#`DELAY;
A = 1'b0;  B = 1'b1; Ci = 1'b0; ALUop0 = 1'b0; ALUop1 = 1'b1; ALUop2 = 1'b0; Less = 1'b0;
#`DELAY;

A = 1'b0;  B = 1'b0; ALUop0 = 1'b1; ALUop1 = 1'b0; ALUop2 = 1'b0; Ci = 1'b0; Less = 1'b0;
#`DELAY;
A = 1'b1;  B = 1'b1; ALUop0 = 1'b1; ALUop1 = 1'b0; ALUop2 = 1'b0; Ci = 1'b0; Less = 1'b0;
#`DELAY;
A = 1'b1;  B = 1'b0; Ci = 1'b0; ALUop0 = 1'b1; ALUop1 = 1'b0; ALUop2 = 1'b0; Less = 1'b0;
#`DELAY;
A = 1'b0;  B = 1'b1; Ci = 1'b0; ALUop0 = 1'b1; ALUop1 = 1'b0; ALUop2 = 1'b0; Less = 1'b0;
#`DELAY;

A = 1'b0;  B = 1'b0; ALUop0 = 1'b1; ALUop1 = 1'b0; ALUop2 = 1'b1; Ci = 1'b1; Less = 1'b0;
#`DELAY;
A = 1'b1;  B = 1'b1; ALUop0 = 1'b1; ALUop1 = 1'b0; ALUop2 = 1'b1; Ci = 1'b1; Less = 1'b0;
#`DELAY;
A = 1'b1;  B = 1'b0; Ci = 1'b1; ALUop0 = 1'b1; ALUop1 = 1'b0; ALUop2 = 1'b1; Less = 1'b0;
#`DELAY;
A = 1'b0;  B = 1'b1; Ci = 1'b1; ALUop0 = 1'b1; ALUop1 = 1'b0; ALUop2 = 1'b1; Less = 1'b0;
#`DELAY;

end

initial
begin
        $monitor("time =%2d, A =%1b, B =%1b, Ci =%1b, ALUop0 =%1b, ALUop1 =%1b, ALUop2
=%1b, R=%1b, Co =%1b",$time, A, B, Ci, ALUop0, ALUop1, ALUop2, R, Co);
end

endmodule
```

Result of this case is shown in the below:



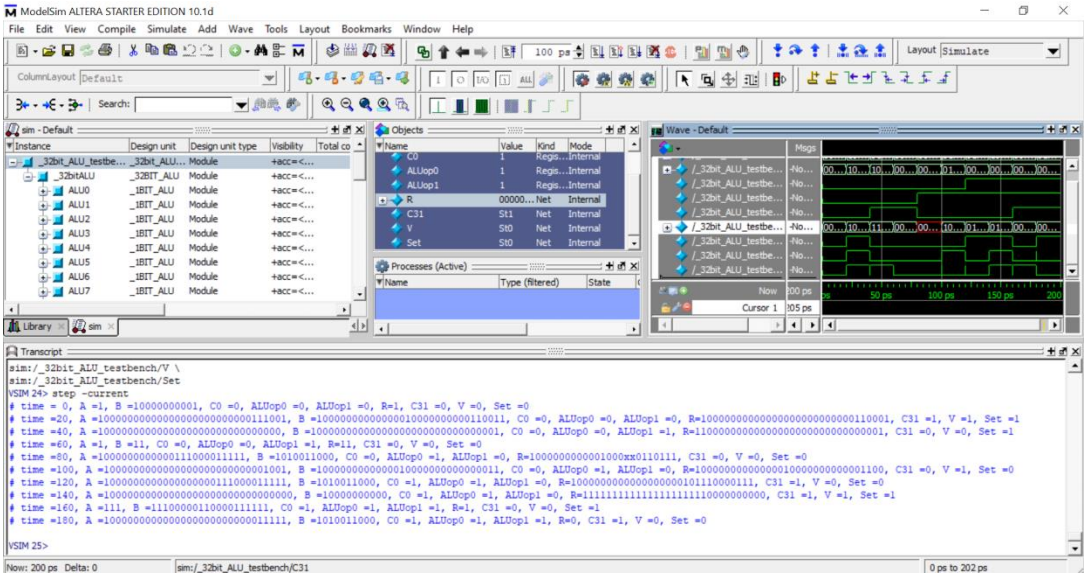
This testbench tests and, or, add and sub operations for 1-bit ALU.Less input is ignored for this situation. Because output of set is observed in _32BIT_ALU module.

Other testbench I wrote tests and, or, add,sub and set-on-less-than operations for 32-bit ALU. The tests I have written, are as follows:

```
module _32bit_ALU_testbench();

reg [31:0] A,B;
reg C0,ALUop0,ALUop1;
wire [31:0] R;
wire C31,V,Set;
_32BIT_ALU _32bitALU (R,C31,V,Set,A,B,C0,ALUop0,ALUop1);

initial begin

//two different cases for testing AND operation.
A = 32'b00000000000000000000000000000001;  B = 32'b00000000000000000000010000000001; C0 = 1'b0;
ALUop0 = 1'b0; ALUop1 = 1'b0;
#`DELAY;
A = 32'b10000000000000000000000000111001;  B = 32'b10000000000000010000000000110011; C0 = 1'b0;
ALUop0 = 1'b0; ALUop1 = 1'b0;
#`DELAY;

//two different cases for testing OR operation.
A = 32'b01000000000000000000000000000000;  B = 32'b10000000000000000000000000000001; C0 = 1'b0;
ALUop0 = 1'b0; ALUop1 = 1'b1;
#`DELAY;
A = 32'b00000000000000000000000000000001;  B = 32'b00000000000000000000000000000011; C0 = 1'b0;
ALUop0 = 1'b0; ALUop1 = 1'b1;
#`DELAY;

//two different cases for testing ADD operation.
A = 32'b00000001000000000000111000011111;  B = 32'b00000000000000000000001010011000; C0 = 1'b0;
ALUop0 = 1'b1; ALUop1 = 1'b0;
#`DELAY;
A = 32'b01000000000000000000000000001001;  B = 32'b01000000000000010000000000000011; C0 = 1'b0;
ALUop0 = 1'b1; ALUop1 = 1'b0;
#`DELAY;

//two different cases for testing SUB operation.
A = 32'b01000000000000000000111000011111;  B = 32'b00000000000000000000001010011000; C0 = 1'b1;
ALUop0 = 1'b1; ALUop1 = 1'b0;
#`DELAY;
A = 32'b10000000000000000000000000000000;  B = 32'b00000000000000000000010000000000;     C0 =
1'b1; ALUop0 = 1'b1; ALUop1 = 1'b0;
#`DELAY;

//two different cases for testing SET-ON-LESS-THAN operation.
A = 32'b00000000000000000000000000000111;  B = 32'b00000000000011100000110000111111;    C0 = 1'b1;
ALUop0 = 1'b1; ALUop1 = 1'b1;
```

```
#`DELAY;
A = 32'b01000000000000000000000000011111;  B = 32'b00000000000000000000000001010011000; C0 = 1'b1;
ALUop0 = 1'b1; ALUop1 = 1'b1;
#`DELAY;

end

initial
begin
        $monitor("time =%2d, A =%1b, B =%1b, C0 =%1b, ALUop0 =%1b, ALUop1 =%1b, R=%1b, C31 =%1b, V
=%1b, Set =%1b",$time, A, B, C0, ALUop0, ALUop1, R, C31, V, Set);
end

endmodule
```

Result of this case is shown in the below:



I used ten different numbers for this testbench. Every two numbers are used for one operation. I also checked correctness of the results by changing print format to %1d. %1d format converts numbers to decimal format, so I calculated the resulton paper additionally.