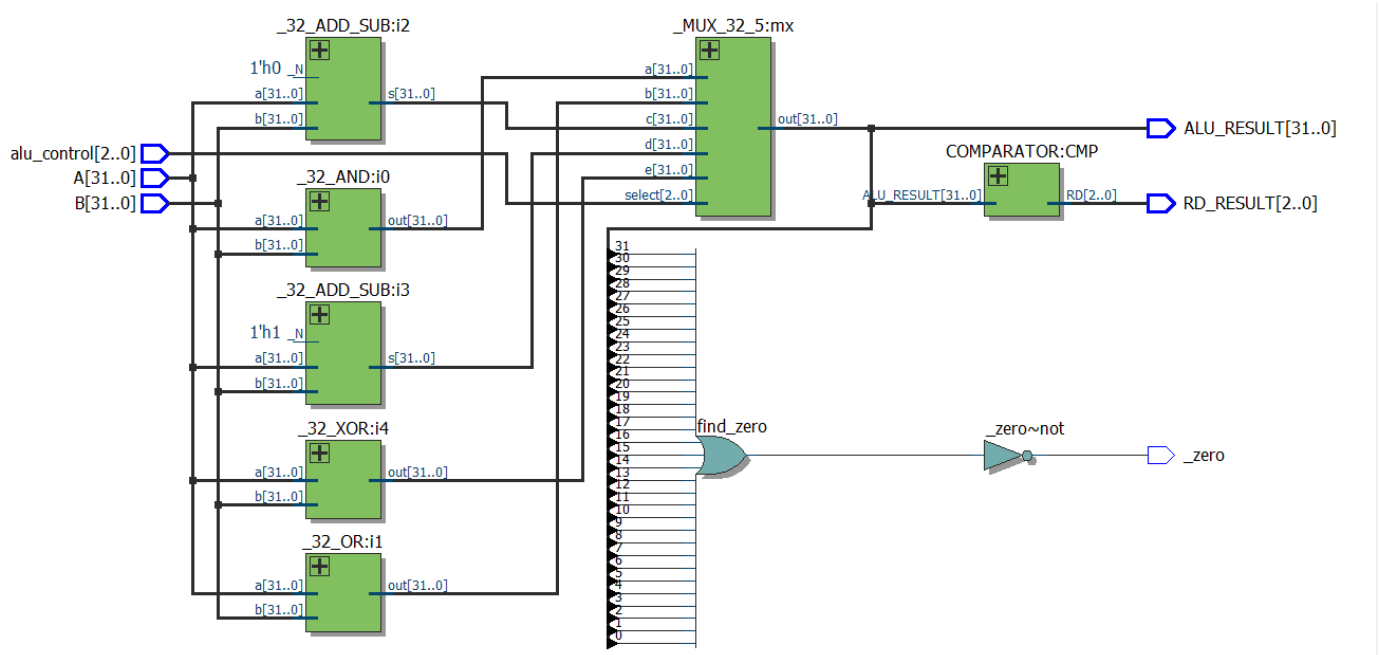
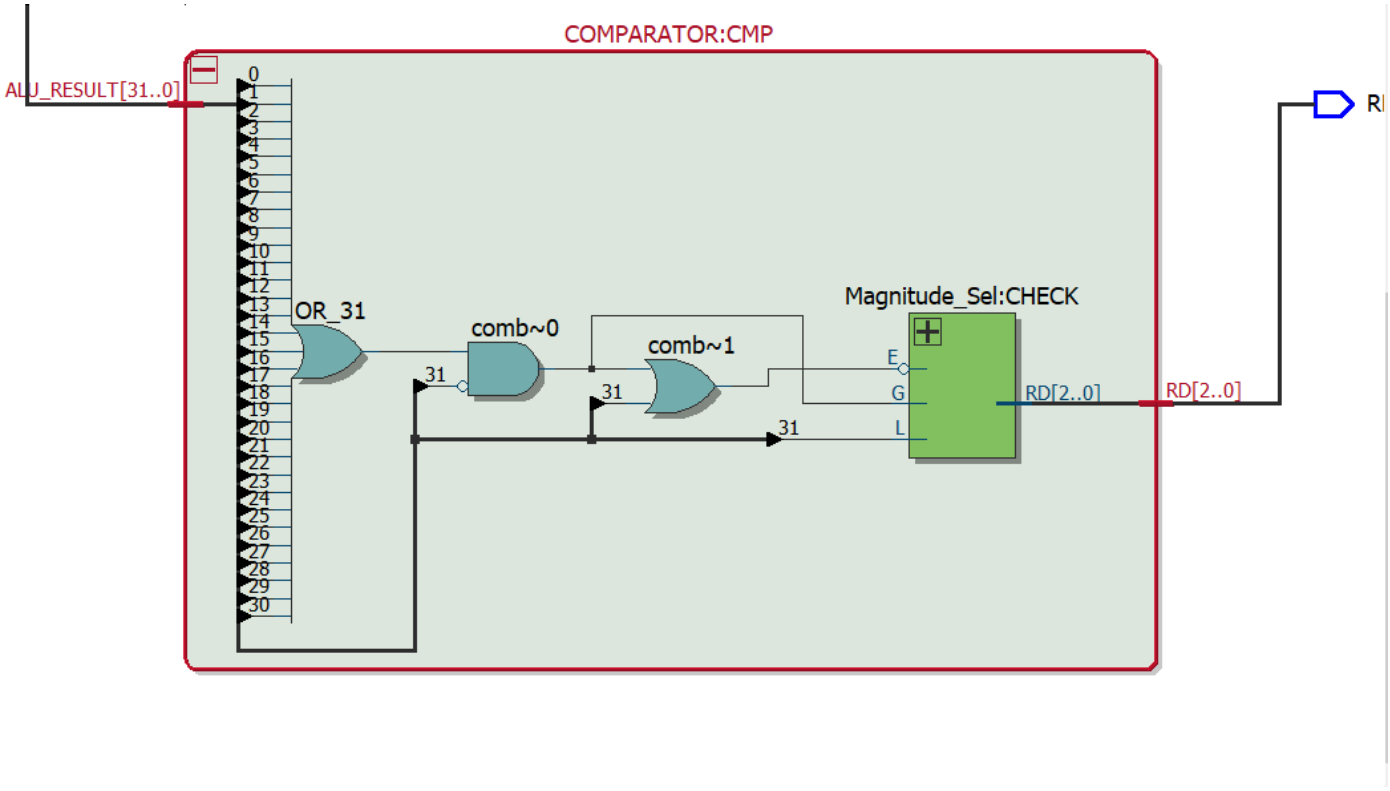
ALU_32 MODULE TEST RESULT (for ALU_RESULT as $RS[op]$RT):

```
# ==> ORN   ::  A = 11111111111111111111111111111111, B = 00000000000000000000000000000001, SEL=000, RESULT=00000000000000000000000000000001
# ==> ADDN  ::  A = 11111111111111111111111111111111, B = 00000000000000000000000000000001, SEL=001, RESULT=11111111111111111111111111111111
# ==> SUBN  ::  A = 01000000000111010110000000000001, B = 01111100000000000000011100000100, SEL=010, RESULT=10111100000111010110011100000101
# ==> XORN  ::  A = 11111111111111111111111111111111, B = 00000000000000000000000000000000, SEL=011, RESULT=11111111111111111111111111111111
```

RTL VIEW OF ALU WITH COMPARATOR:

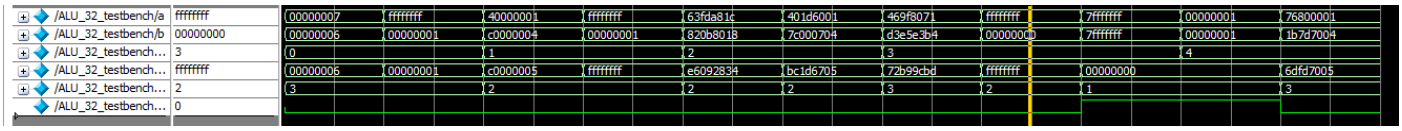


RTL VIEW OF COMPARATOR:

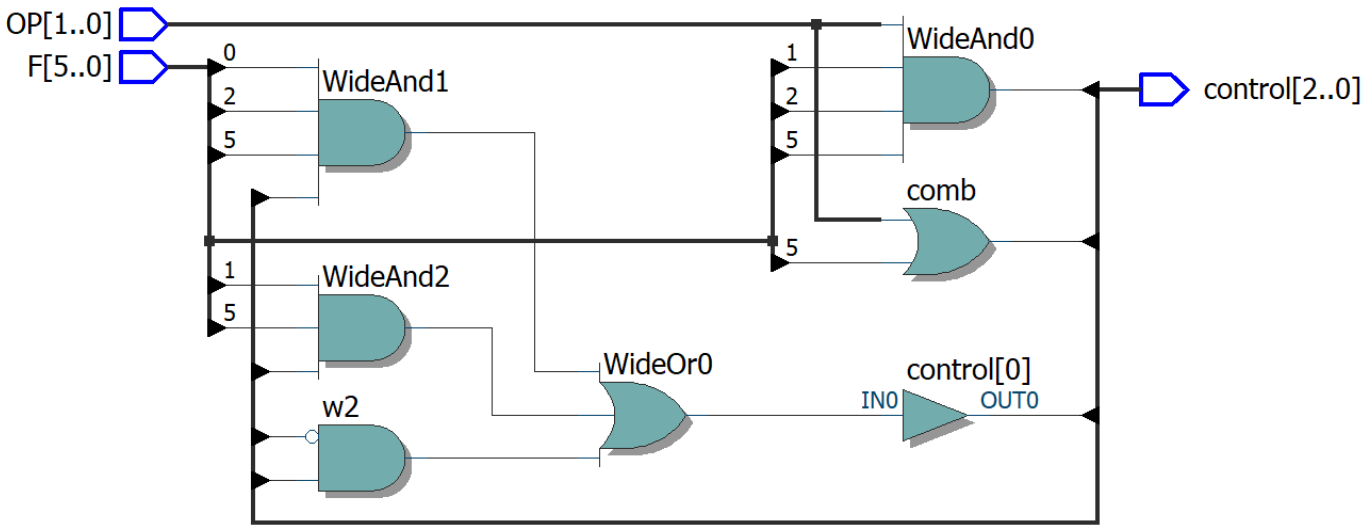


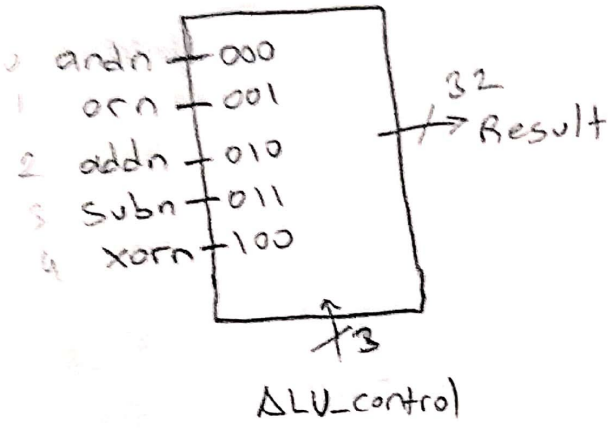
ALU_32 MODULE TEST RESULT (RD_RESULT AFTER COMPARATOR):

```
# ==> ANDN  ::  A = 00000000000000000000000000000111, B = 00000000000000000000000000000110, ALU_CONTROL=000, RESULT=00000000000000000000000000000110, RD=  3, Zero: 0
# ==> ANDN  ::  A = 11111111111111111111111111111111, B = 00000000000000000000000000000001, ALU_CONTROL=000, RESULT=00000000000000000000000000000001, RD=  3, Zero: 0
# ==> ORN   ::  A = 01000000000000000000000000000001, B = 11000000000000000000000000000100, ALU_CONTROL=001, RESULT=11000000000000000000000000000101, RD=  2, Zero: 0
# ==> ORN   ::  A = 11111111111111111111111111111111, B = 00000000000000000000000000000001, ALU_CONTROL=001, RESULT=11111111111111111111111111111111, RD=  2, Zero: 0
# ==> ADDN  ::  A = 01100011111111011010100000011100, B = 10000010000010111000000000011000, ALU_CONTROL=010, RESULT=11100110000010010010100000110100, RD=  2, Zero: 0
# ==> ADDN  ::  A = 01000000000111010110000000000001, B = 01111100000000000000011100000100, ALU_CONTROL=010, RESULT=10111100000111010110011100000101, RD=  2, Zero: 0
# ==> SUBN  ::  A = 01000110100111111000000001110001, B = 11010011111001011110001110110100, ALU_CONTROL=011, RESULT=01110010101110011001110010111101, RD=  3, Zero: 0
# ==> SUBN  ::  A = 11111111111111111111111111111111, B = 00000000000000000000000000000000, ALU_CONTROL=011, RESULT=11111111111111111111111111111111, RD=  2, Zero: 0
# ==> SUBN  ::  A = 01111111111111111111111111111111, B = 01111111111111111111111111111111, ALU_CONTROL=011, RESULT=00000000000000000000000000000000, RD=  1, Zero: 1
# ==> XORN  ::  A = 00000000000000000000000000000001, B = 00000000000000000000000000000001, ALU_CONTROL=100, RESULT=00000000000000000000000000000000, RD=  1, Zero: 1
# ==> XORN  ::  A = 01110110100000000000000000000001, B = 00011011011110101110000000000100, ALU_CONTROL=100, RESULT=01101101111111010111000000000101, RD=  3, Zero: 0
```



RTL VIEW OF ALU CONTROL:



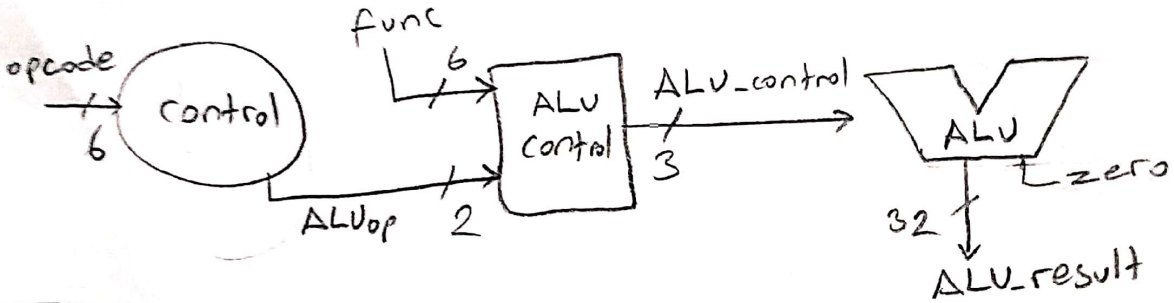


ALU_control:

[2]:

[1] =

[0] =



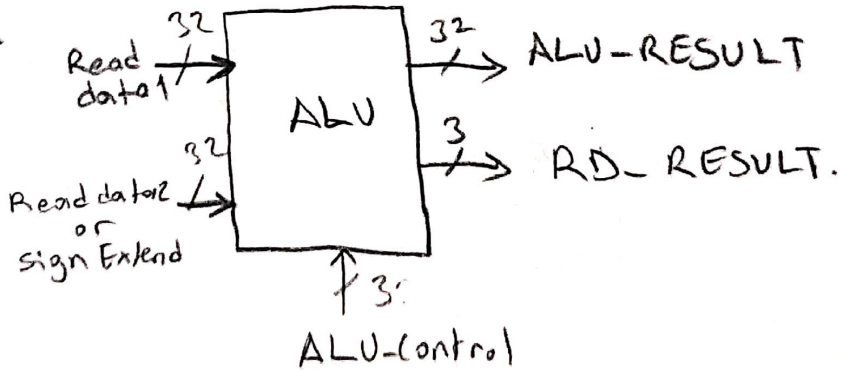
Comparator

rs+rt = ALU_Result

```
if(rs+rt == 0) $rd <= 1
else if(rs+rt <0) $rd <= 2
    else $rd <= 3
```

- ALU module has a RD_RESULT output.

new ALU →



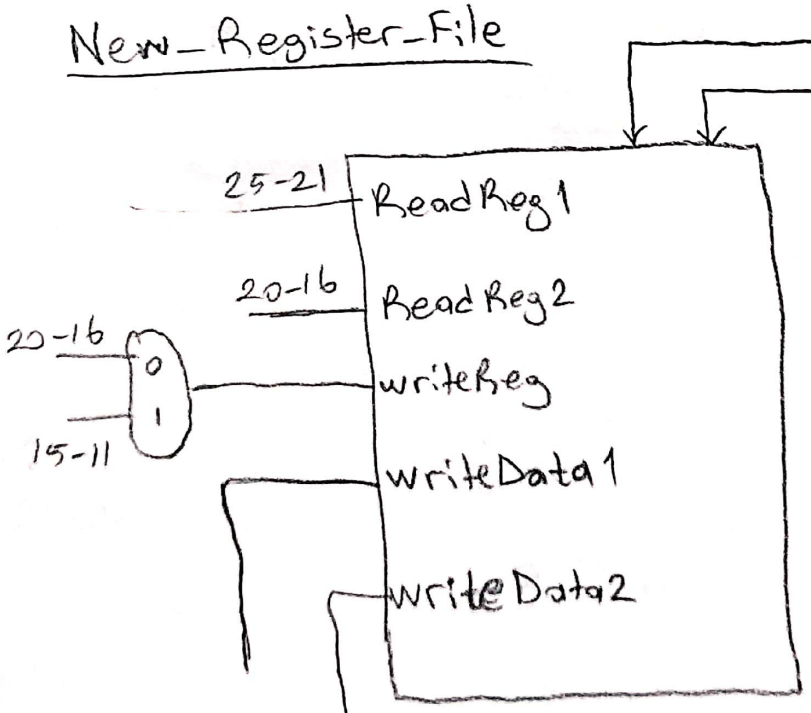
lw	00	X	010
sw	00	X	010
beq	01	X	011
andn	10	100100	000
orn	10	100101	001
addn	10	100000	010
subn	10	100010	011
xorn	10	100110	100
	ALU op [1-0]	function [5-0]	ALU control

$Control[2] = ALUop1 \cdot F5 \cdot F2 \cdot F1$

$Control[1] = ALUop1 + F5$

$Control[0] = \overline{ALUop1} \cdot ALUop0 + ALUop1 \cdot F5 \cdot F2 \cdot F0 + ALUop1 \cdot F5 \cdot F1$

New_Register_File

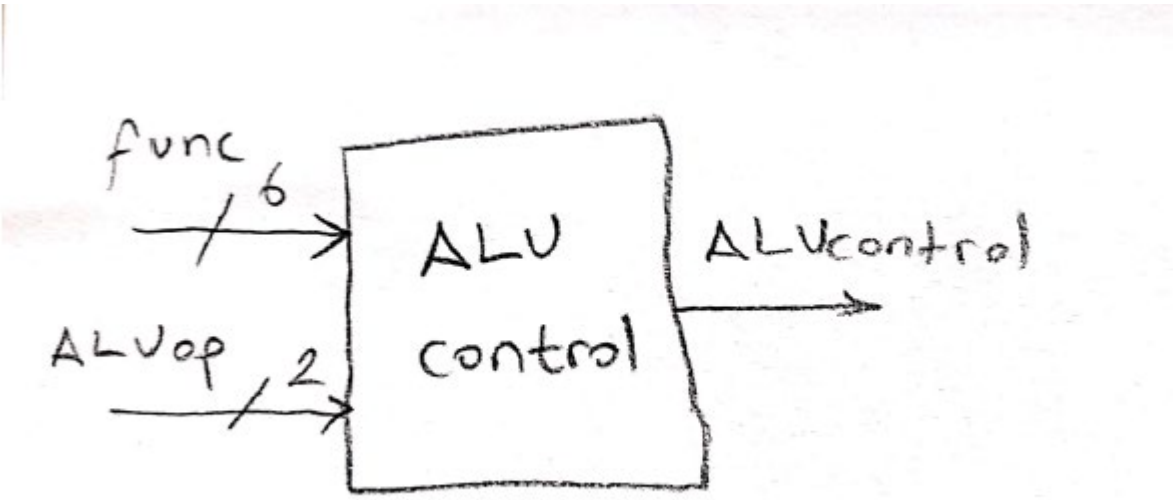


They will be written at the same clock time (RS, RD)

- RegWrite_Rs means new instructions. Thus, this command will write data to RS

- And also to write RS, instruction mux should be RD[15-11]

mips_register module



NOTE : **I COULD NOT MADE THE DATAPATH CONNECTIONS PROPERLY THEREFORE I SUBMIT ONLY MODULES THAT I MADE.**

I HAVE THE FOLLOWING MAIN MODULES IN MY ASSIGNMENT:

```
module _HA(s, c, a, b);
module _FA(sum, carry_out, a, b, carry_in);
module _32_AND (out, a, b);
module _32_OR (out, a, b);
module _32_NOR (out, a, b);
module _32_XOR (out, a, b);
module _32_ADD_SUB (s,a,b,_N);
`define DELAY 50
module _32_ADD_SUB_testbench();
```

```
//=============================================================
module _MUX_2_1 (out,a,b, select);
//=============================================================
module _MUX_32_4 (out, a,b,c,d, select0, select1);
//=============================================================
module _MUX_32_8 (out, a,b,c,d,e,f,g,h, select);
//=============================================================
module _MUX_32_2 (out,a,b, select);
//=============================================================
module _MUX_32_5 (out, a,b,c,d,e, select);
//=============================================================
```

```
module _PC (clock,jump_signal,beq_signal,beq_equal,extended_32);
endmodule
module Next_PC (nextPC,clock,jump_signal,branch_signal,CurrentPC);
endmodule
```

```
module mips_registers(  read_data_1, read_data_2,
                        write_data1,write_data2,
                        read_reg_1, read_reg_2,
                        write_reg, reg_write_signal,
                        reg_write_RS_signal,clk);
```

```
module ALU_32 (ALU_RESULT, A, B, alu_control, RD_RESULT, _zero);
    input [31:0] A, B;
    input [2:0] alu_control;
    output [31:0] ALU_RESULT;
    output [2:0] RD_RESULT;
    output _zero;
```

```
`define DELAY 50
module ALU_32_testbench();
reg [31:0]a, b;
reg [2:0]alu_cntrol;
wire [31:0] alu_result;
wire [2:0] RD;
wire _zero;
ALU_32 alu(alu_result, a, b, alu_cntrol, RD, _zero);
initial begin
```

```
module ALU_CONTROL (input [5:0] F, input [1:0] OP, output [2:0] control);

wire w1,w2,w3,w4;

and ( control[2], OP[1],F[5],F[2],F[1] );
or  ( control[1], OP[1],F[5] );
not (w1,control[1]);
and (w2,w1,control[0]);
and (w3,control[1],F[5],F[2],F[0] );
and (w4,control[1],F[5],F[1]);
or  ( control[0], w2, w3, w4);

endmodule
```

```
module COMPARATOR ( input [31:0] ALU_RESULT, output [2:0] RD );
endmodule



module COMPARATOR_tb;
```

```
module Magnitude_Sel ( G,E,L, RD );

endmodule

module Magnitude_Sel_TB();

endmodule
```

```
module data_mem (read_data, mem_address, write_data, memRead_signal, signal_memWrite,clock);
    output reg [31:0] read_data;
    input [31:0] mem_address,write_data;
    input memRead_signal,signal_memWrite,clock;
```

```
module instr_mem(instruction ,PC);
    input [31:0] PC;
    output reg [31:0] instruction;
    reg [31:0] instr_mem [255:0];
    //always takes the instruction with  PC
        always @(*) begin
            instruction = instr_mem[PC];
        end
endmodule
```

```
module jump_address(jump_address,instr,PC,zero);
    input[3:0] PC;
    input zero;
    input [25:0]instr;
    output[31:0]jump_address;
```

```
module zero_extend(_imm,zeroExt_imm,zero);
    input [15:0] _imm;
    input zero;
    output [31:0] zeroExt_imm;
```

```
module sign_extend_imm(_imm,signExt_imm);
    input [15:0] _imm;
    output [31:0] signExt_imm;
```

The data memory size will be 256KB whereas the instruction memory size will be 16KB. Remember that addressing for a 256KB memory only requires 18 bits instead of 32 bits in regular MIPS. Update your design accordingly.

NOTE : **I encountered some problems when working with 18 bit instead of 32 thus I made conections as 32 bit in memory modules.**