University of Balamand

Embedded Project

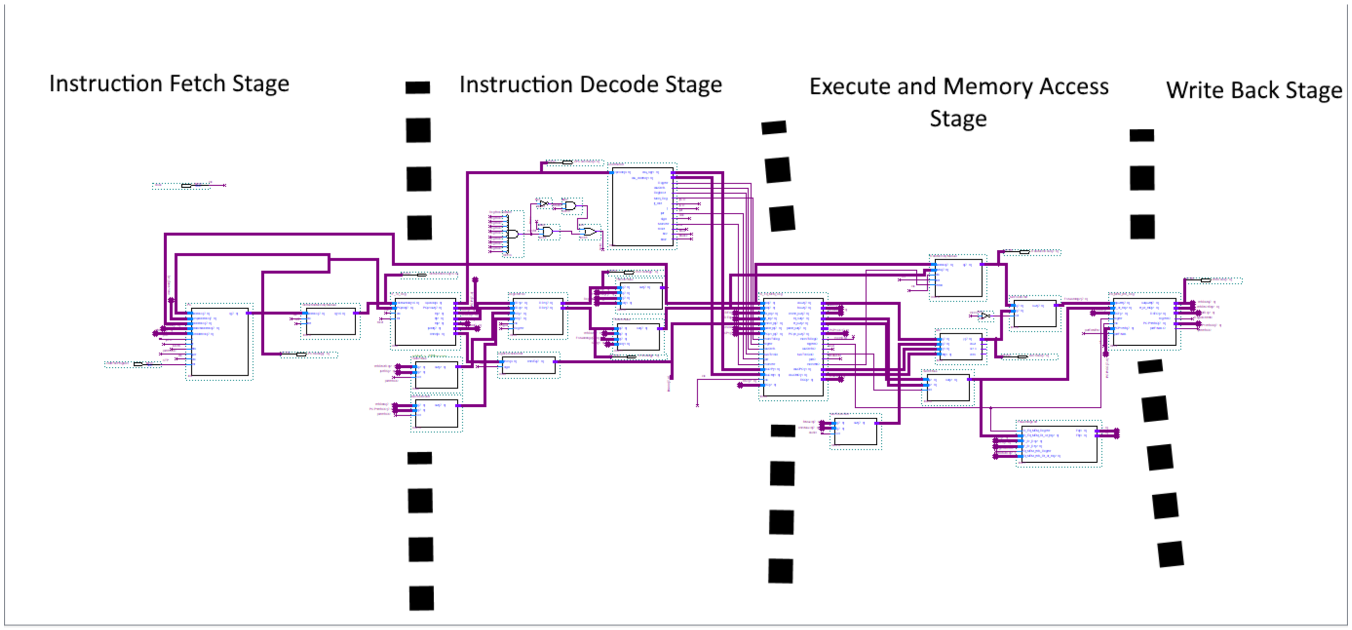
By: Michel Sfeir & Rodolph Khlat

Presented to

Dr. Rafic Ayoubi

The final report for the CPEN 313: Computer Embedded Systems

Faculty of Engineering



This is the highest-level schematic file, it connects all the components we built together. It represents a 4-stage Pipelined Processor based on a reduced MIPS instruction set. It includes a forwarding unit, control unit, and can do instructions such as add, sub, addi, andi, lw, sw, jump…

<u>Stages Division:</u>

1-Instruction Fetch

2-Instruction Decode

3-Execute/Memory Access

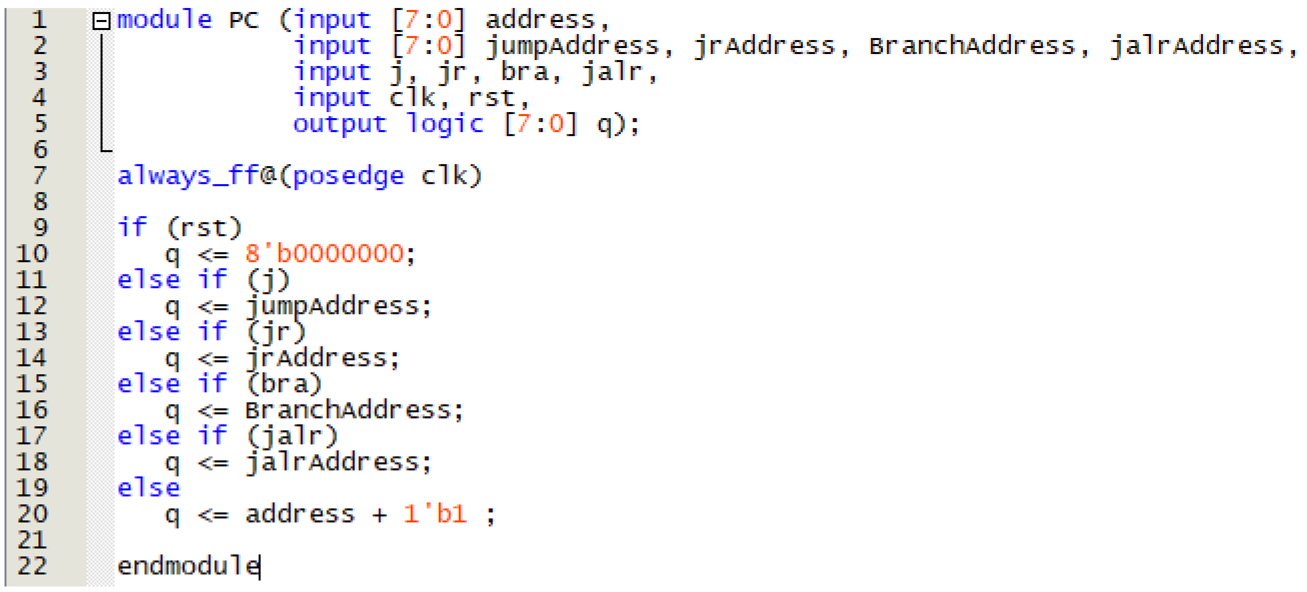
4-Write Back

<u>Part 1:</u>

The components in this part are: The Program Counter, and the Instruction Memory

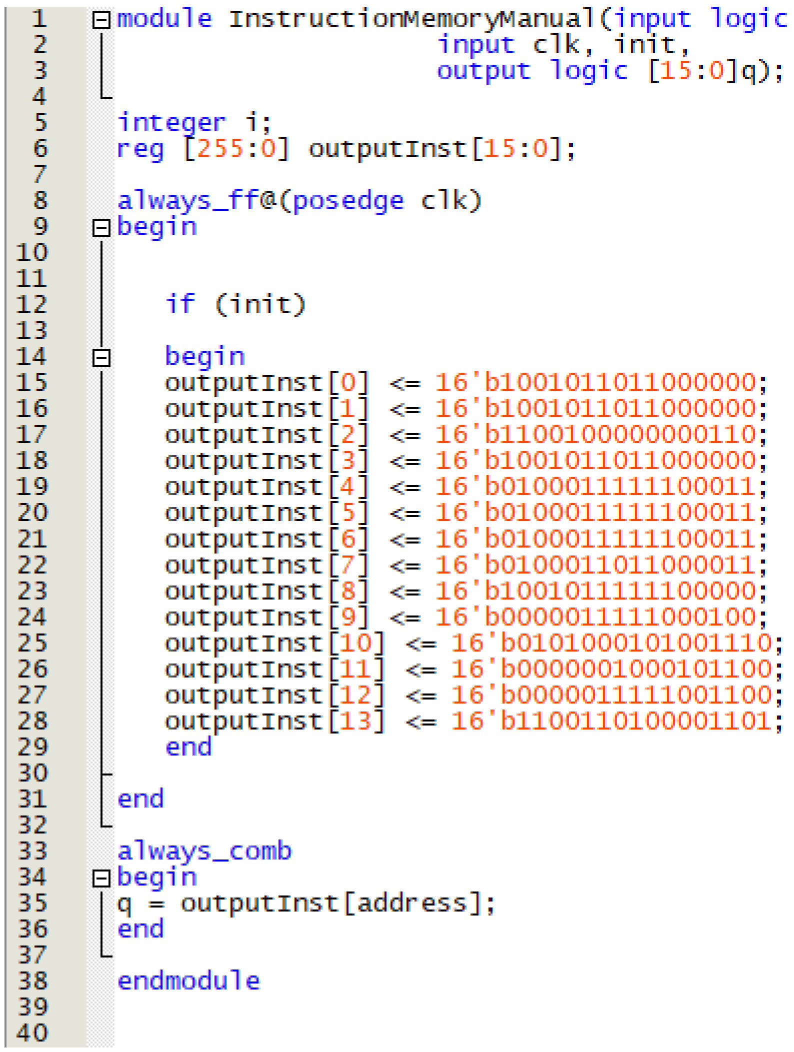
<u>Program Counter:</u>

The Program Counter normally counts up by 1 every cycle (instruction memory is instruction addressable). However, it can also do jump or branch (bez or bnez) or jalr or jr.



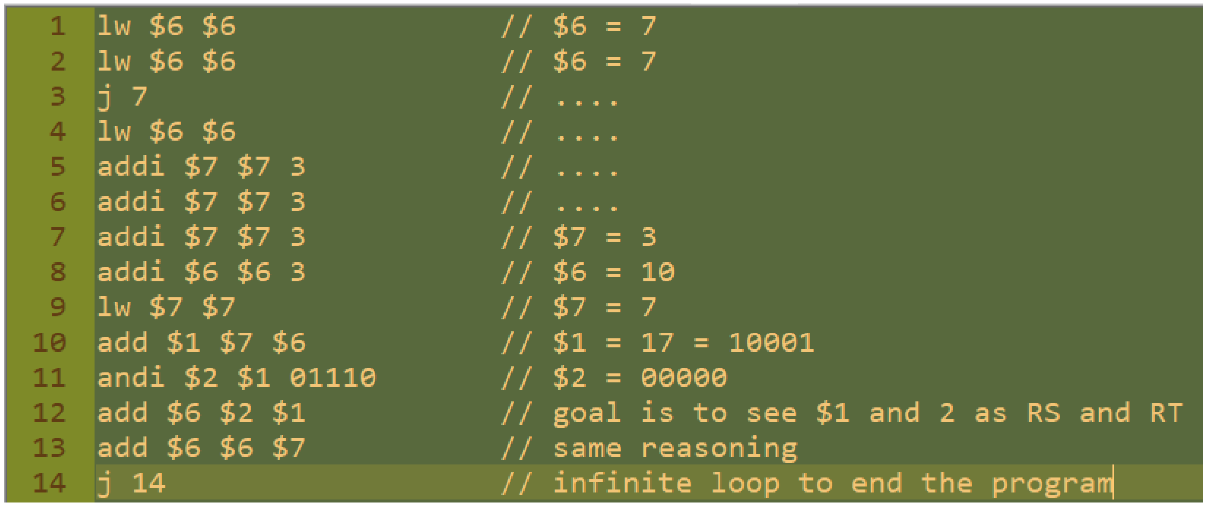
```
module PC (input [7:0] address,
           input [7:0] jumpAddress, jrAddress, BranchAddress, jalrAddress,
           input j, jr, bra, jalr,
           input clk, rst,
           output logic [7:0] q);

always_ff@(posedge clk)

if (rst)
   q <= 8'b0000000;
else if (j)
   q <= jumpAddress;
else if (jr)
   q <= jrAddress;
else if (bra)
   q <= BranchAddress;
else if (jalr)
   q <= jalrAddress;
else
   q <= address + 1'b1 ;

endmodule
```

As for the **instruction memory**, we have also designed our own version of it. All it needs is one input, which is the instruction address which it gets from the program counter, and it outputs the 16-bit instruction like so:



```
module InstructionMemoryManual(input logic [7:0]address,
                               input clk, init,
                               output logic [15:0]q);

integer i;
reg [255:0] outputInst[15:0];

always_ff@(posedge clk)
begin


   if (init)

   begin
   outputInst[0]  <= 16'b1001011011000000;
   outputInst[1]  <= 16'b1001011011000000;
   outputInst[2]  <= 16'b1100100000000110;
   outputInst[3]  <= 16'b1001011011000000;
   outputInst[4]  <= 16'b0100011111100011;
   outputInst[5]  <= 16'b0100011111100011;
   outputInst[6]  <= 16'b0100011111100011;
   outputInst[7]  <= 16'b0100011011000011;
   outputInst[8]  <= 16'b1001011111100000;
   outputInst[9]  <= 16'b0000011111000100;
   outputInst[10]  <= 16'b0101000101001110;
   outputInst[11]  <= 16'b0000001000101100;
   outputInst[12]  <= 16'b0000011111001100;
   outputInst[13]  <= 16'b1100110100001101;
   end

end

always_comb
begin
q = outputInst[address];
end

endmodule
```

As we can see, the instruction memory includes the set of instructions which constitute the program itself. We can set it to be whatever we want. In this case, we have chosen a simple program that demonstrates most of the capabilities of our processor. The instructions above in assembly language translate to:

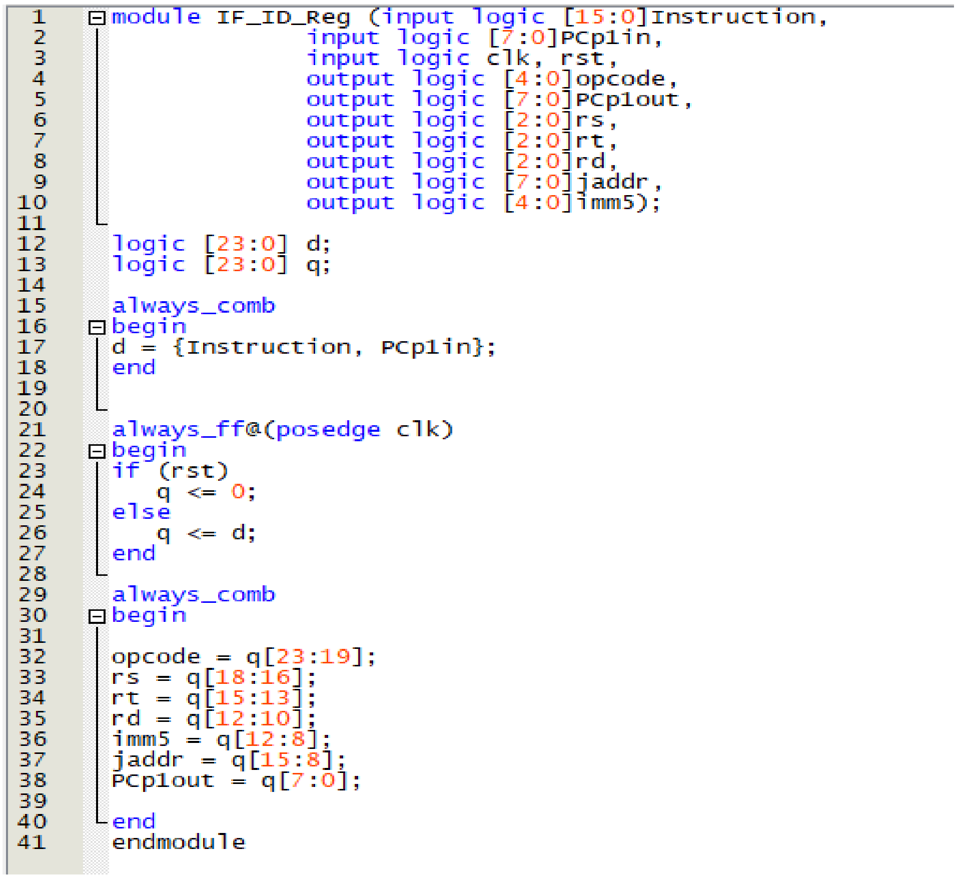


```
lw $6 $6                // $6 = 7
lw $6 $6                // $6 = 7
j 7                     // ....
lw $6 $6                // ....
addi $7 $7 3            // ....
addi $7 $7 3            // ....
addi $7 $7 3            // $7 = 3
addi $6 $6 3            // $6 = 10
lw $7 $7                // $7 = 7
add $1 $7 $6            // $1 = 17 = 10001
andi $2 $1 01110        // $2 = 00000
add $6 $2 $1            // goal is to see $1 and 2 as RS and RT
add $6 $6 $7            // same reasoning
j 14                    // infinite loop to end the program
```

This program demonstrates the use of the jump instruction in 2 ways, use of the ALU to do different operations, access to the data memory, and the use of forwarding.

Intermediary Part: IF/ID Pipeline Register

This register takes the instruction and breaks it up into its constituent parts so that it is easier to use in the next stages. It also carries over PC+1 value for use in jalr.

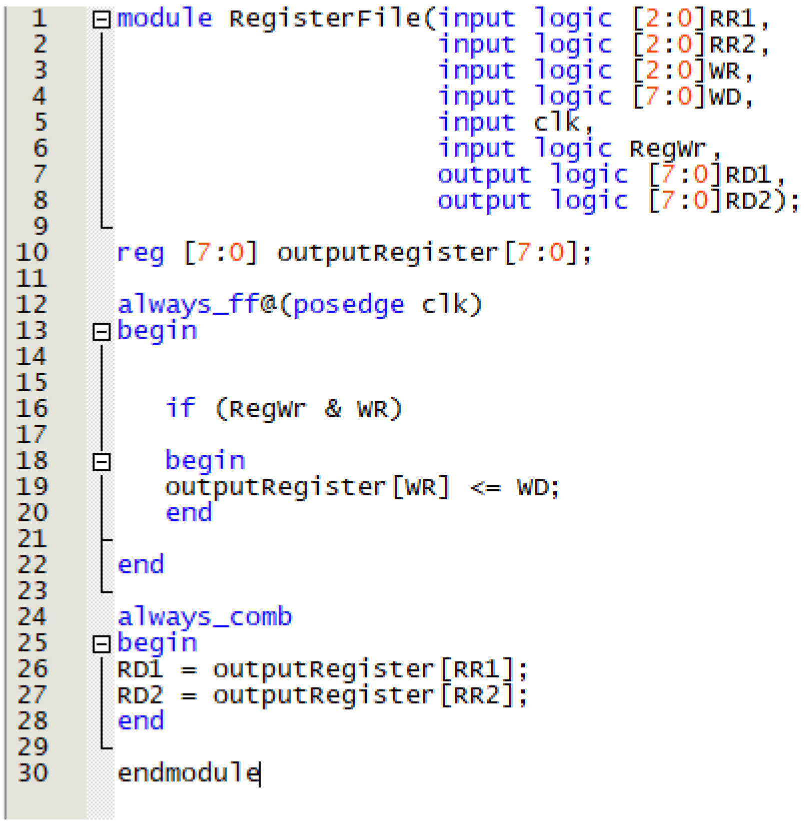


```
module IF_ID_Reg (input logic [15:0]Instruction,
                  input logic [7:0]PCp1in,
                  input logic clk, rst,
                  output logic [4:0]opcode,
                  output logic [7:0]PCp1out,
                  output logic [2:0]rs,
                  output logic [2:0]rt,
                  output logic [2:0]rd,
                  output logic [7:0]jaddr,
                  output logic [4:0]imm5);

logic [23:0] d;
logic [23:0] q;

always_comb
begin
d = {Instruction, PCp1in};
end


always_ff@(posedge clk)
begin
if (rst)
    q <= 0;
else
    q <= d;
end

always_comb
begin

opcode = q[23:19];
rs = q[18:16];
rt = q[15:13];
rd = q[12:10];
imm5 = q[12:8];
jaddr = q[15:8];
PCp1out = q[7:0];

end
endmodule
```

Part 2:

The components in this part are: The Register File, the sign extension unit, the branch detection unit, the control unit, and several multiplexers. The design of the two-to-one and three-to-one muxes is assumed to be trivial and so won't be discussed here.

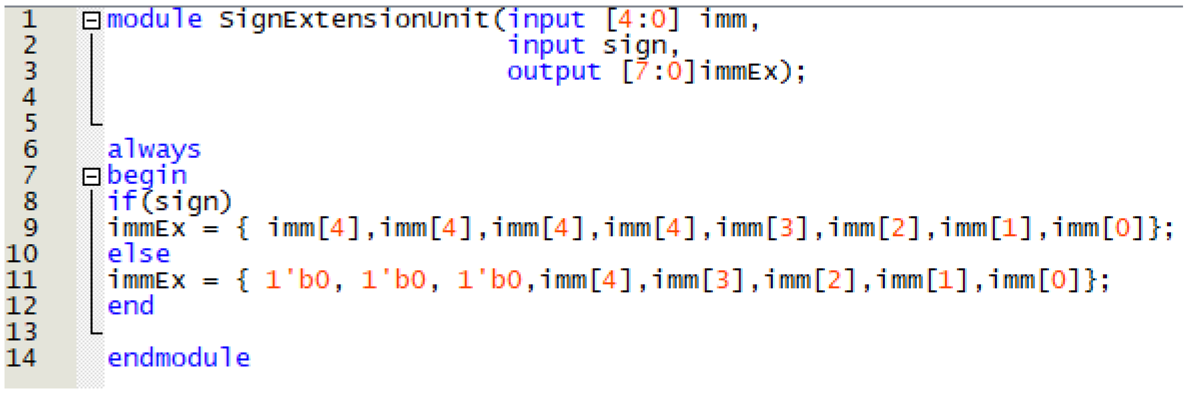
Register File:



```
module RegisterFile(input logic [2:0]RR1,
                    input logic [2:0]RR2,
                    input logic [2:0]WR,
                    input logic [7:0]WD,
                    input clk,
                    input logic RegWr,
                    output logic [7:0]RD1,
                    output logic [7:0]RD2);

reg [7:0] outputRegister[7:0];

always_ff@(posedge clk)
begin


    if (RegWr & WR)

    begin
    outputRegister[WR] <= WD;
    end

end

always_comb
begin
RD1 = outputRegister[RR1];
RD2 = outputRegister[RR2];
end

endmodule
```

It consists of 8, 8-bit registers. It has two read ports and one write port. Before writing it asserts that WR != 0, so that Register 0 is never written to. Having a register whose value is always 0 can be useful in software.

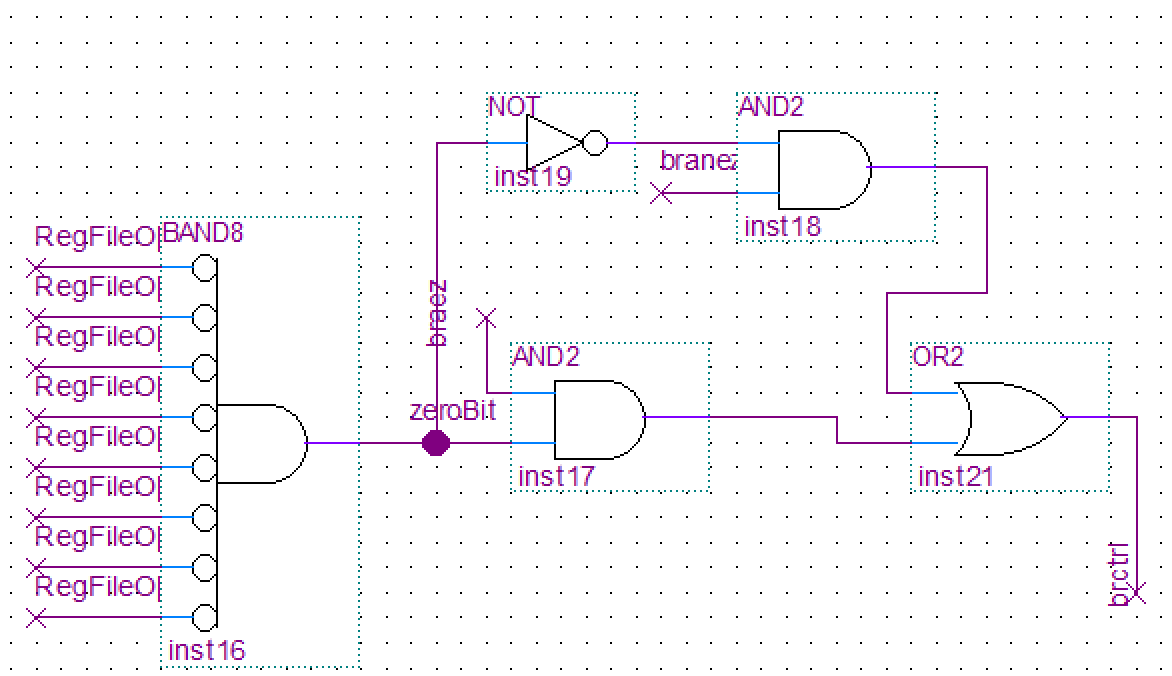
Sign Extension Unit:



```
module SignExtensionUnit(input [4:0] imm,
                         input sign,
                         output [7:0]immEx);


always
begin
if(sign)
immEx = { imm[4],imm[4],imm[4],imm[4],imm[3],imm[2],imm[1],imm[0]};
else
immEx = { 1'b0, 1'b0, 1'b0,imm[4],imm[3],imm[2],imm[1],imm[0]};
end

endmodule
```

Simply takes the 5-bit immediate given in I-format and extends it to an 8-bit immediate so that it fits with the rest of the datapath.

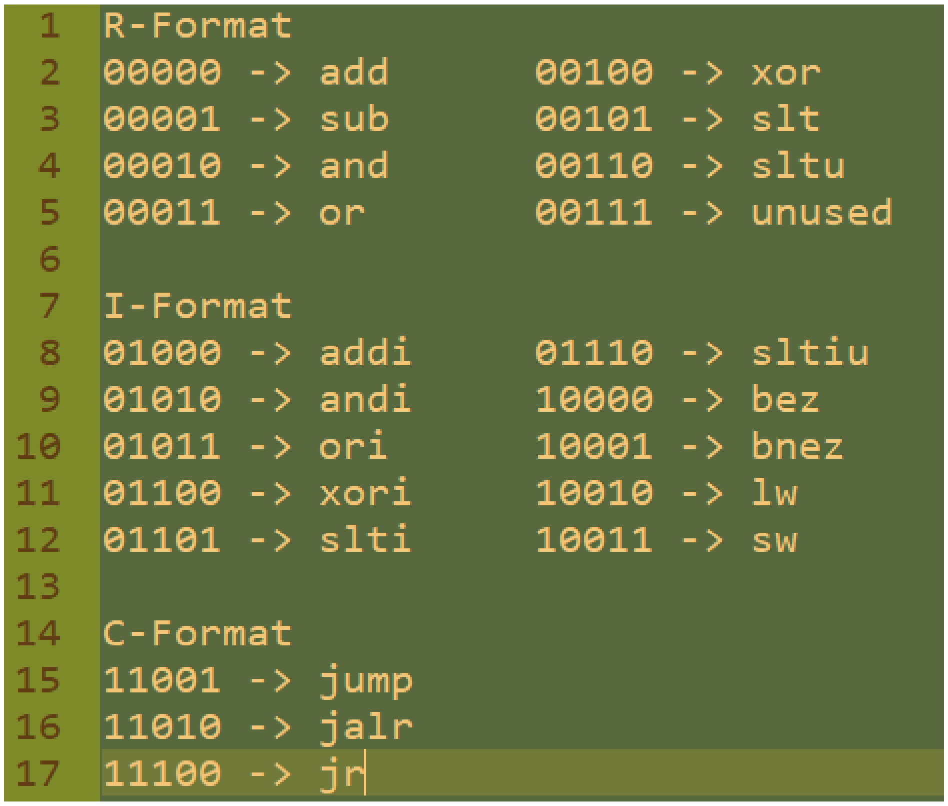
Branch Detection:



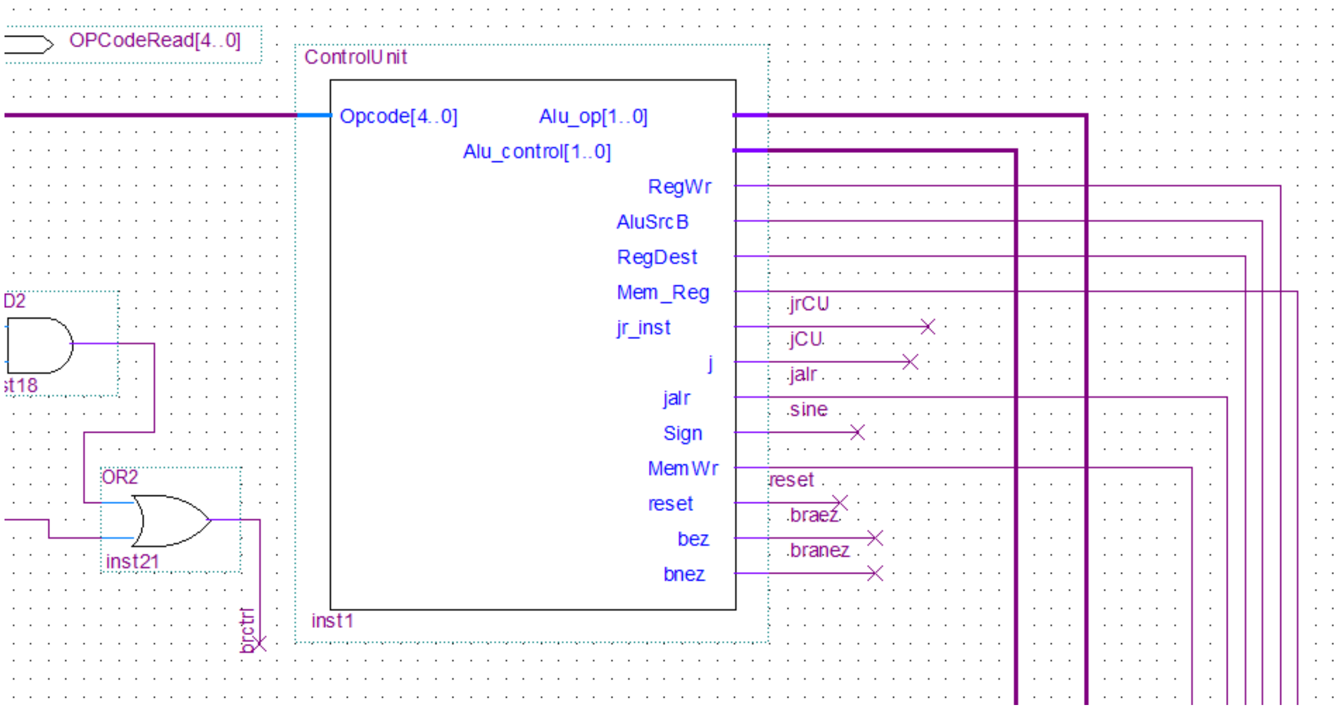
Outputs a 1 if and only if: The register being read in RS is 0 and the control unit has asserted BEZ or the register being read is not 0 and the control unit has asserted BNEZ.

Control Unit:

The control unit matches opcodes to instructions and generates control signals accordingly. Importantly, these are the opcodes it detects:

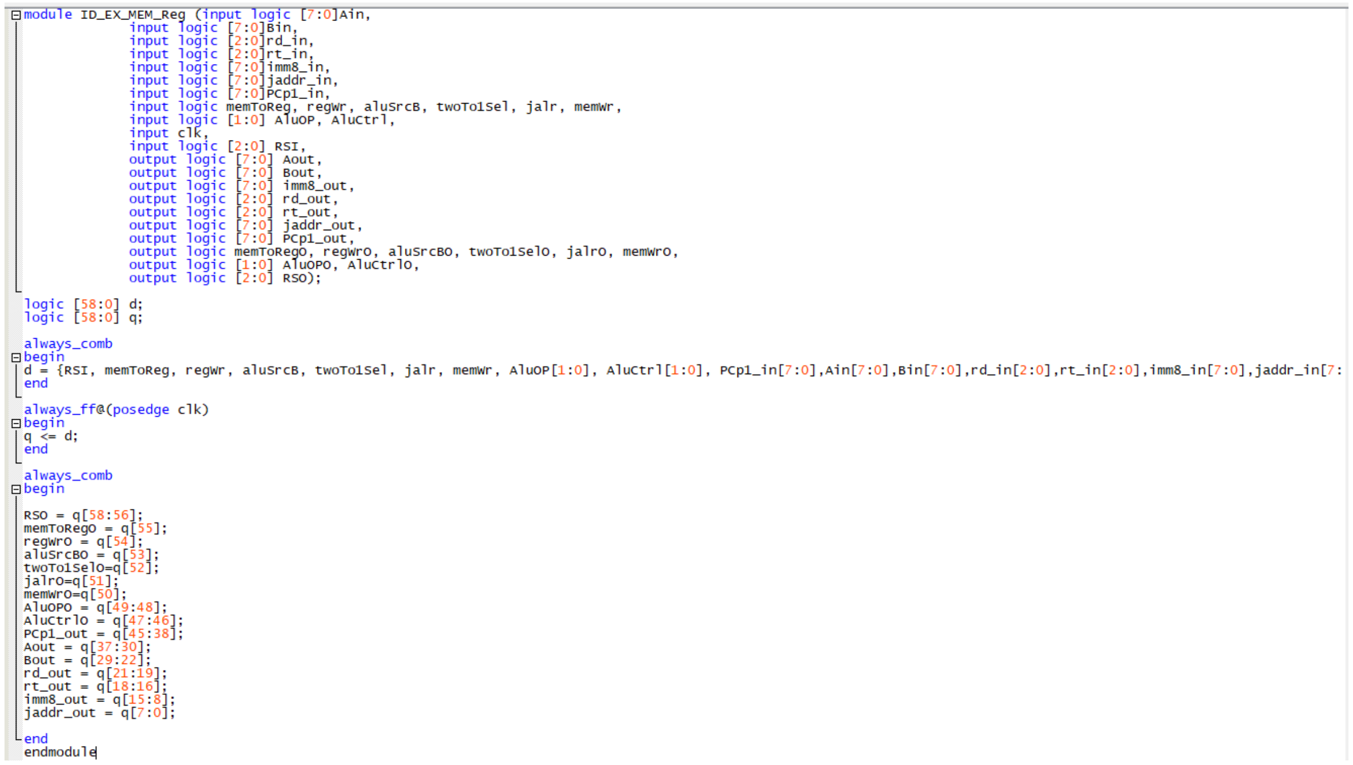


```
R-Format
00000 -> add        00100 -> xor
00001 -> sub        00101 -> slt
00010 -> and        00110 -> sltu
00011 -> or         00111 -> unused

I-Format
01000 -> addi       01110 -> sltiu
01010 -> andi       10000 -> bez
01011 -> ori        10001 -> bnez
01100 -> xori       10010 -> lw
01101 -> slti       10011 -> sw

C-Format
11001 -> jump
11010 -> jalr
11100 -> jr
```



Intermediary Part 2: ID/EXMEM Pipeline Register

A register to pass on all the control signals that weren't consumed in the previous stage, as well as RS/RT/RD and Jump Address and PC+1 for JALR



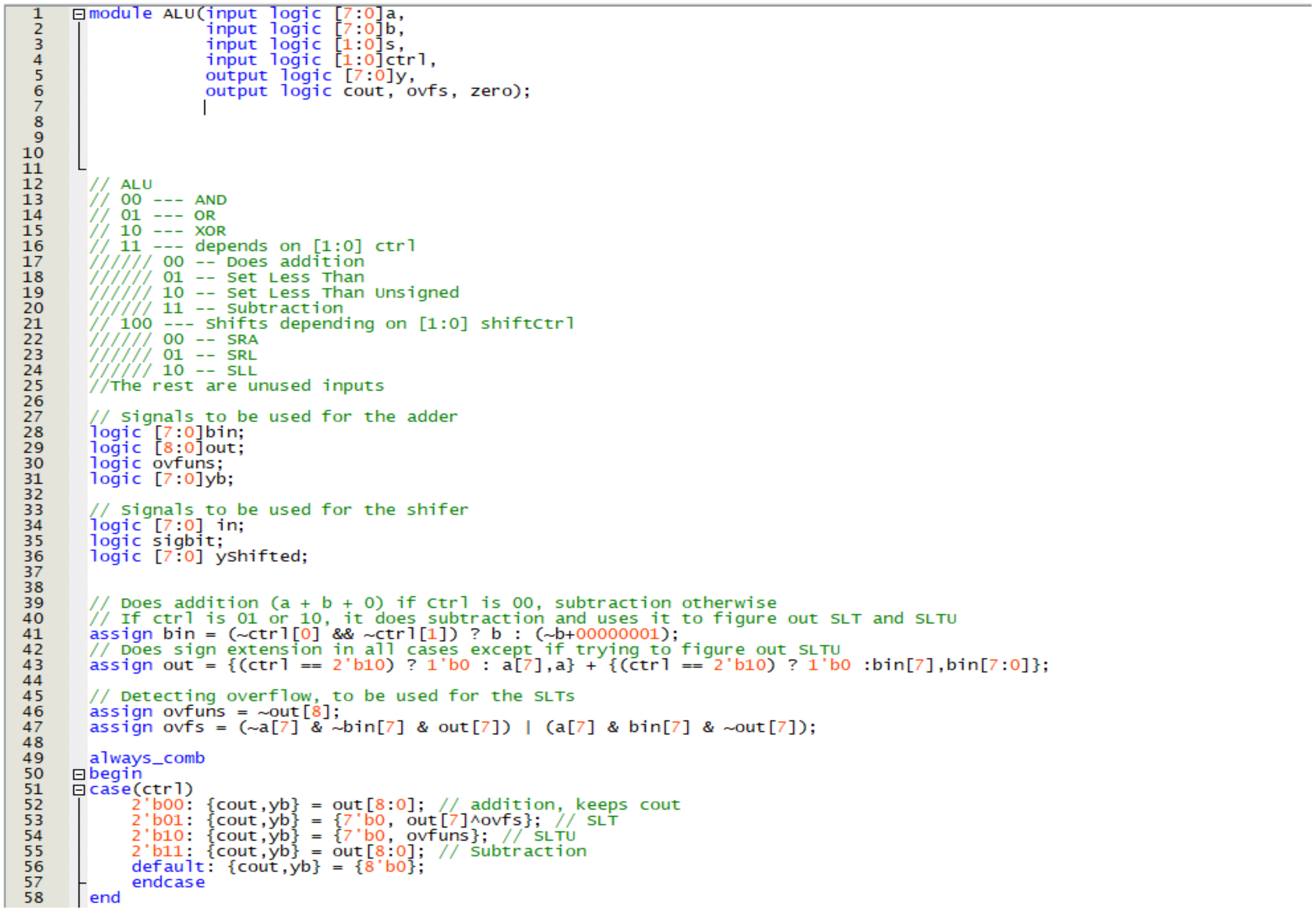
```
module ID_EX_MEM_Reg (input logic [7:0]Ain,
              input logic [7:0]Bin,
              input logic [2:0]rd_in,
              input logic [2:0]rt_in,
              input logic [7:0]imm8_in,
              input logic [7:0]jaddr_in,
              input logic [7:0]PCp1_in,
              input logic memToReg, regWr, aluSrcB, twoTo1Sel, jalr, memWr,
              input logic [1:0] AluOP, AluCtrl,
              input clk,
              input logic [2:0] RSI,
              output logic [7:0] Aout,
              output logic [7:0] Bout,
              output logic [7:0] imm8_out,
              output logic [2:0] rd_out,
              output logic [2:0] rt_out,
              output logic [7:0] jaddr_out,
              output logic [7:0] PCp1_out,
              output logic memToRegO, regWrO, aluSrcBO, twoTo1SelO, jalrO, memWrO,
              output logic [1:0] AluOPO, AluCtrlO,
              output logic [2:0] RSO);

logic [58:0] d;
logic [58:0] q;

always_comb
begin
d = {RSI, memToReg, regWr, aluSrcB, twoTo1Sel, jalr, memWr, AluOP[1:0], AluCtrl[1:0], PCp1_in[7:0],Ain[7:0],Bin[7:0],rd_in[2:0],rt_in[2:0],imm8_in[7:0],jaddr_in[7:
end

always_ff@(posedge clk)
begin
q <= d;
end

always_comb
begin

RSO = q[58:56];
memToRegO = q[55];
regWrO = q[54];
aluSrcBO = q[53];
twoTo1SelO=q[52];
jalrO=q[51];
memWrO=q[50];
AluOPO = q[49:48];
AluCtrlO = q[47:46];
PCp1_out = q[45:38];
Aout = q[37:30];
Bout = q[29:22];
rd_out = q[21:19];
rt_out = q[18:16];
imm8_out = q[15:8];
jaddr_out = q[7:0];

end
endmodule
```

Part 3:

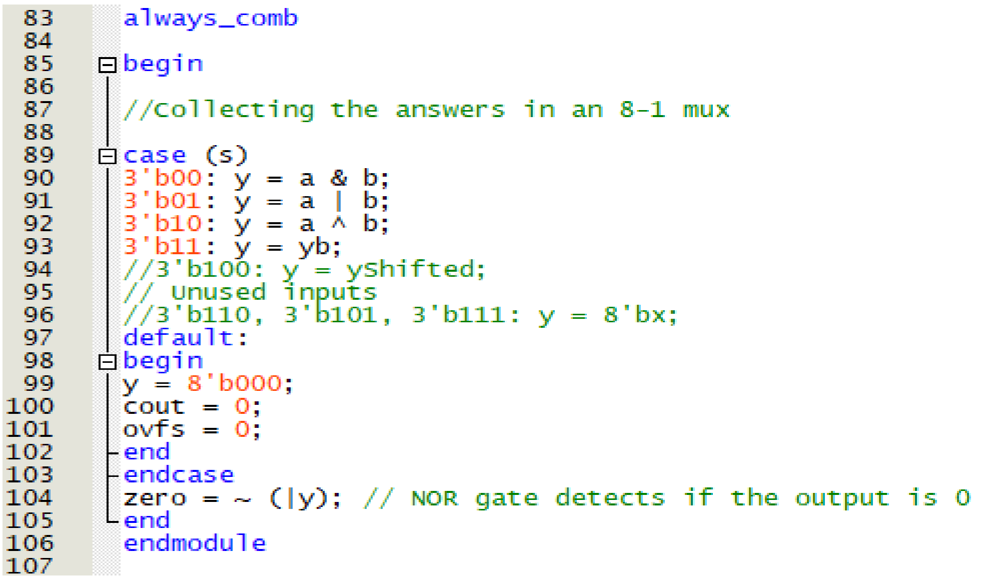
The important components for this part are the ALU and the Data Memory (we also used a couple of multiplexers)

ALU:

We used the ALU designed in Logic 1, repurposed so that it does not have to do shifting operations.

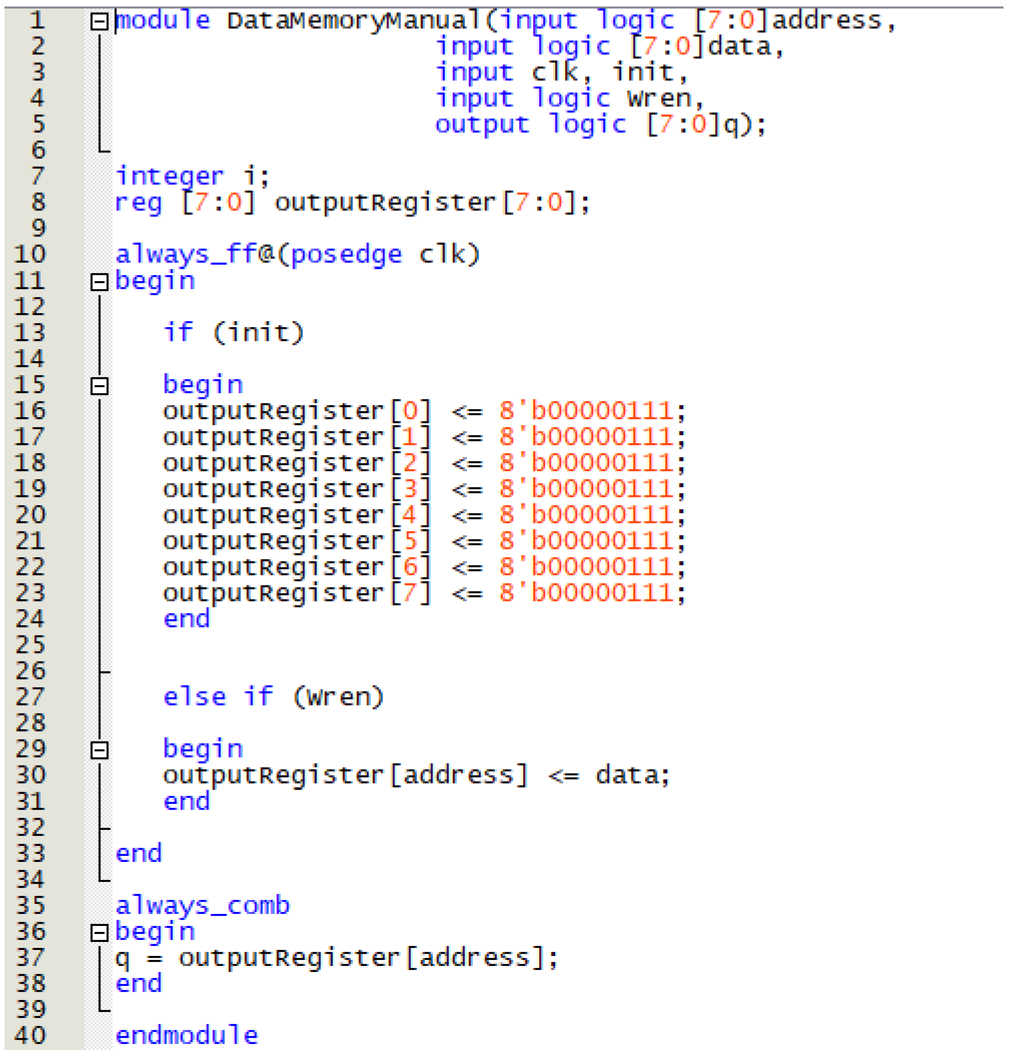


```
module ALU(input logic [7:0]a,
               input logic [7:0]b,
               input logic [1:0]s,
               input logic [1:0]ctrl,
               output logic [7:0]y,
               output logic cout, ovfs, zero);




// ALU
// 00 --- AND
// 01 --- OR
// 10 --- XOR
// 11 --- depends on [1:0] ctrl
////// 00 -- Does addition
////// 01 -- Set Less Than
////// 10 -- Set Less Than Unsigned
////// 11 -- Subtraction
// 100 --- Shifts depending on [1:0] shiftCtrl
////// 00 -- SRA
////// 01 -- SRL
////// 10 -- SLL
//The rest are unused inputs

// Signals to be used for the adder
logic [7:0]bin;
logic [8:0]out;
logic ovfuns;
logic [7:0]yb;

// Signals to be used for the shifer
logic [7:0] in;
logic sigbit;
logic [7:0] yShifted;


// Does addition (a + b + 0) if Ctrl is 00, subtraction otherwise
// If ctrl is 01 or 10, it does subtraction and uses it to figure out SLT and SLTU
assign bin = (~ctrl[0] && ~ctrl[1]) ? b : (~b+00000001);
// Does sign extension in all cases except if trying to figure out SLTU
assign out = {(ctrl == 2'b10) ? 1'b0 : a[7],a} + {(ctrl == 2'b10) ? 1'b0 :bin[7],bin[7:0]};

// Detecting overflow, to be used for the SLTs
assign ovfuns = ~out[8];
assign ovfs = (~a[7] & ~bin[7] & out[7]) | (a[7] & bin[7] & ~out[7]);

always_comb
begin
case(ctrl)
      2'b00: {cout,yb} = out[8:0]; // addition, keeps cout
      2'b01: {cout,yb} = {7'b0, out[7]^ovfs}; // SLT
      2'b10: {cout,yb} = {7'b0, ovfuns}; // SLTU
      2'b11: {cout,yb} = out[8:0]; // Subtraction
      default: {cout,yb} = {8'b0};
      endcase
end
```



```
always_comb

begin

//Collecting the answers in an 8-1 mux

case (s)
3'b00: y = a & b;
3'b01: y = a | b;
3'b10: y = a ^ b;
3'b11: y = yb;
//3'b100: y = yShifted;
// Unused inputs
//3'b110, 3'b101, 3'b111: y = 8'bx;
default:
begin
y = 8'b000;
cout = 0;
ovfs = 0;
end
endcase
zero = ~ (|y); // NOR gate detects if the output is 0
end
endmodule
```

Data Memory:

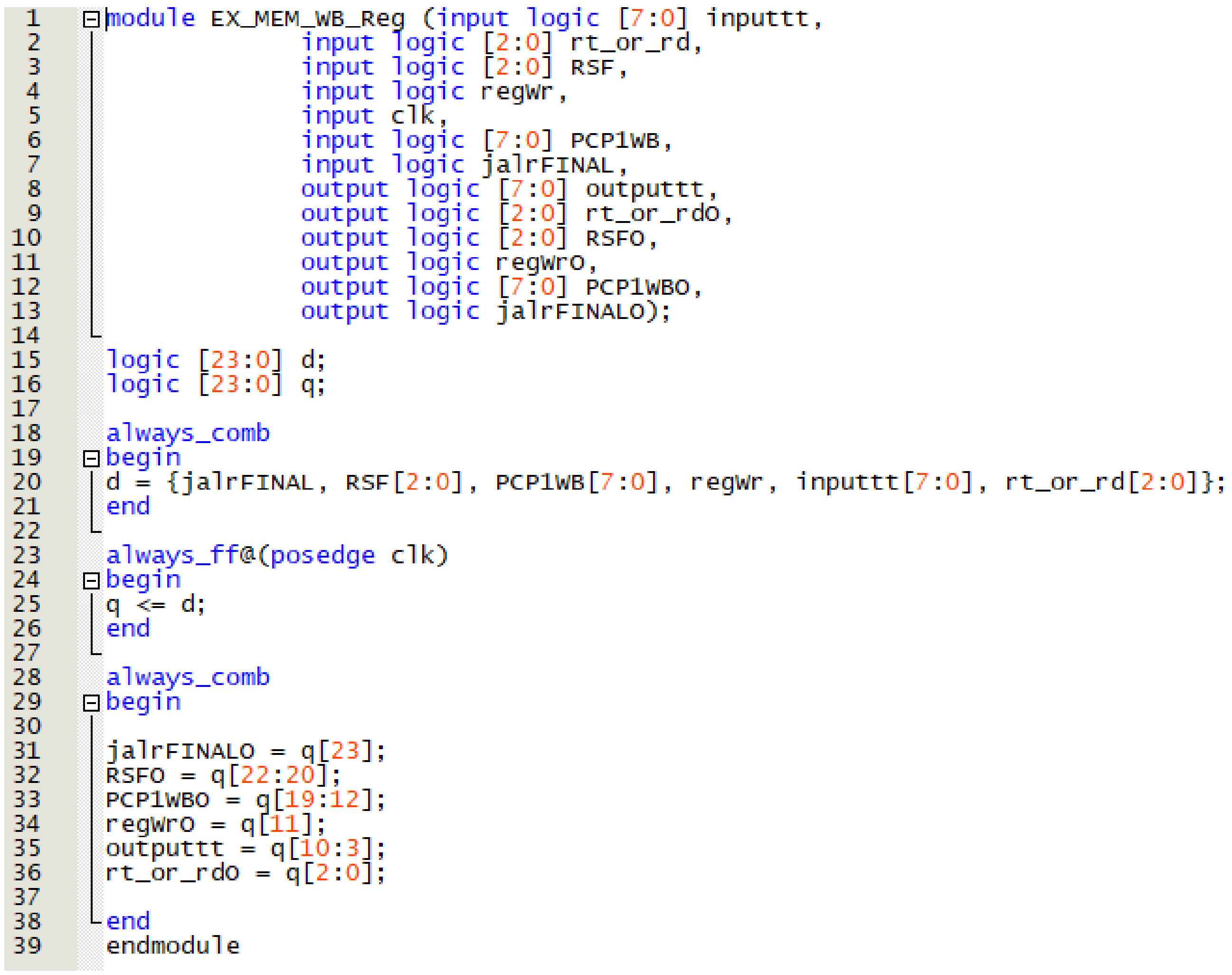


```
module DataMemoryManual(input logic [7:0]address,
                        input logic [7:0]data,
                        input clk, init,
                        input logic wren,
                        output logic [7:0]q);

integer i;
reg [7:0] outputRegister[7:0];

always_ff@(posedge clk)
begin

   if (init)

   begin
   outputRegister[0] <= 8'b00000111;
   outputRegister[1] <= 8'b00000111;
   outputRegister[2] <= 8'b00000111;
   outputRegister[3] <= 8'b00000111;
   outputRegister[4] <= 8'b00000111;
   outputRegister[5] <= 8'b00000111;
   outputRegister[6] <= 8'b00000111;
   outputRegister[7] <= 8'b00000111;
   end


   else if (wren)

   begin
   outputRegister[address] <= data;
   end

end

always_comb
begin
q = outputRegister[address];
end

endmodule
```

Similar to the Instruction Memory, it has only one read port and one write port and uses the same address for both since we will never need to read and write from memory at the same time. At program start it is initialized with whatever values we define here. In this case for the purposes of demonstration I have filled the first 8 registers with the value of 7.

Intermediary Part 3: EXMEM/WB Pipeline Register

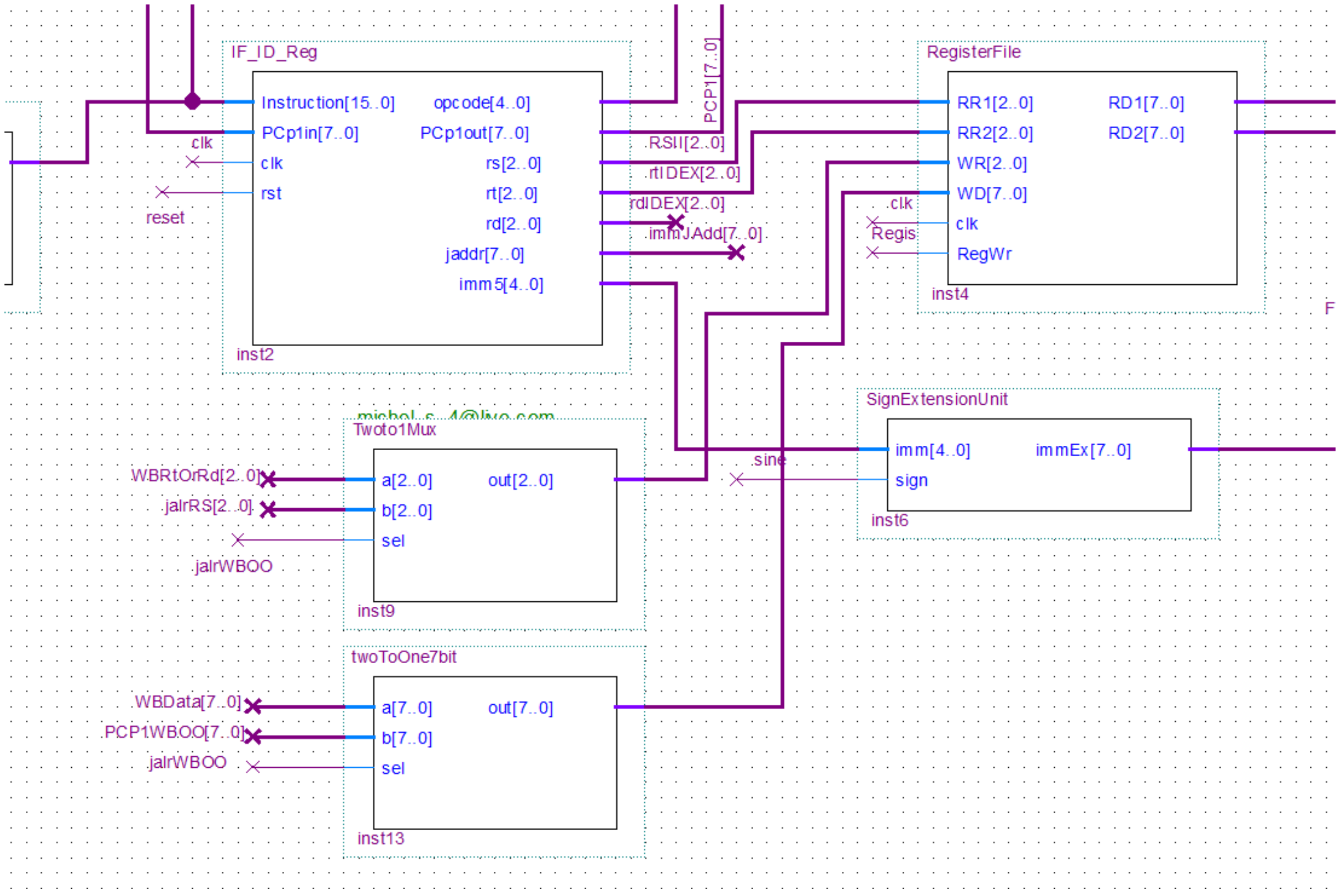
Very few signals still need to be passed on to the last stage. In particular, we need the value to be written back in the register file, its address, and WriteEnable signal to determine whether we want to write anything at all. If Jalr is asserted, we may also need to write PC + 1 in RS so we carry those 3 signals also



```
module EX_MEM_WB_Reg (input logic [7:0] inputtt,
                      input logic [2:0] rt_or_rd,
                      input logic [2:0] RSF,
                      input logic regWr,
                      input clk,
                      input logic [7:0] PCP1WB,
                      input logic jalrFINAL,
                      output logic [7:0] outputtt,
                      output logic [2:0] rt_or_rdO,
                      output logic [2:0] RSFO,
                      output logic regWrO,
                      output logic [7:0] PCP1WBO,
                      output logic jalrFINALO);

logic [23:0] d;
logic [23:0] q;

always_comb
begin
d = {jalrFINAL, RSF[2:0], PCP1WB[7:0], regWr, inputtt[7:0], rt_or_rd[2:0]};
end

always_ff@(posedge clk)
begin
q <= d;
end

always_comb
begin

jalrFINALO = q[23];
RSFO = q[22:20];
PCP1WBO = q[19:12];
regWrO = q[11];
outputtt = q[10:3];
rt_or_rdO = q[2:0];

end
endmodule
```

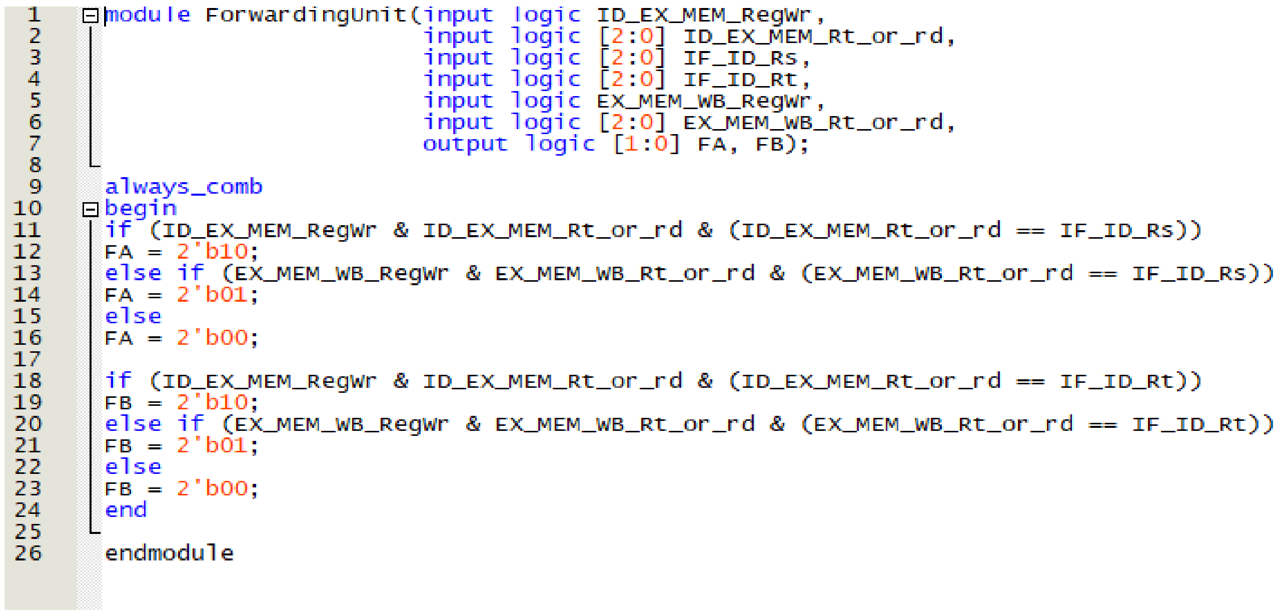
Part 4:

There is no specific hardware for this part. We simply need a couple of multiplexers at the input of the register file to determine what, if anything, we are going to write in it.



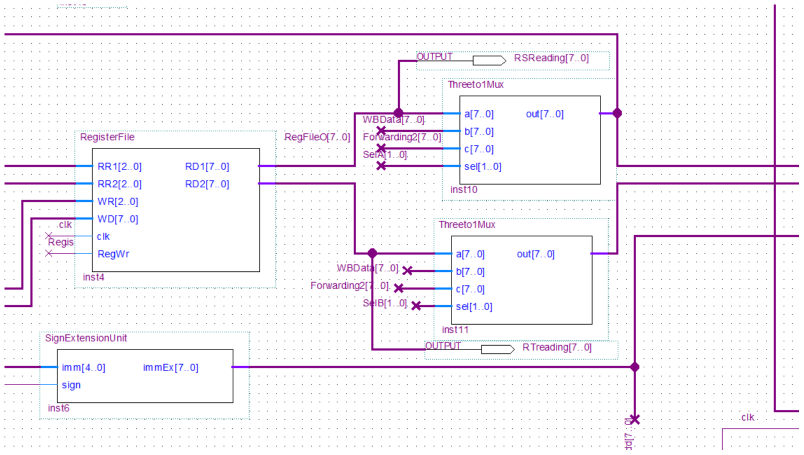
Forwarding Unit:

In some cases our processor may need to forward data if it is needed in instructions faster than it can be written to the Register File itself. This forwarding unit takes inputs from different stages to determine whether this is necessary and outputs the signals that make forwarding happen



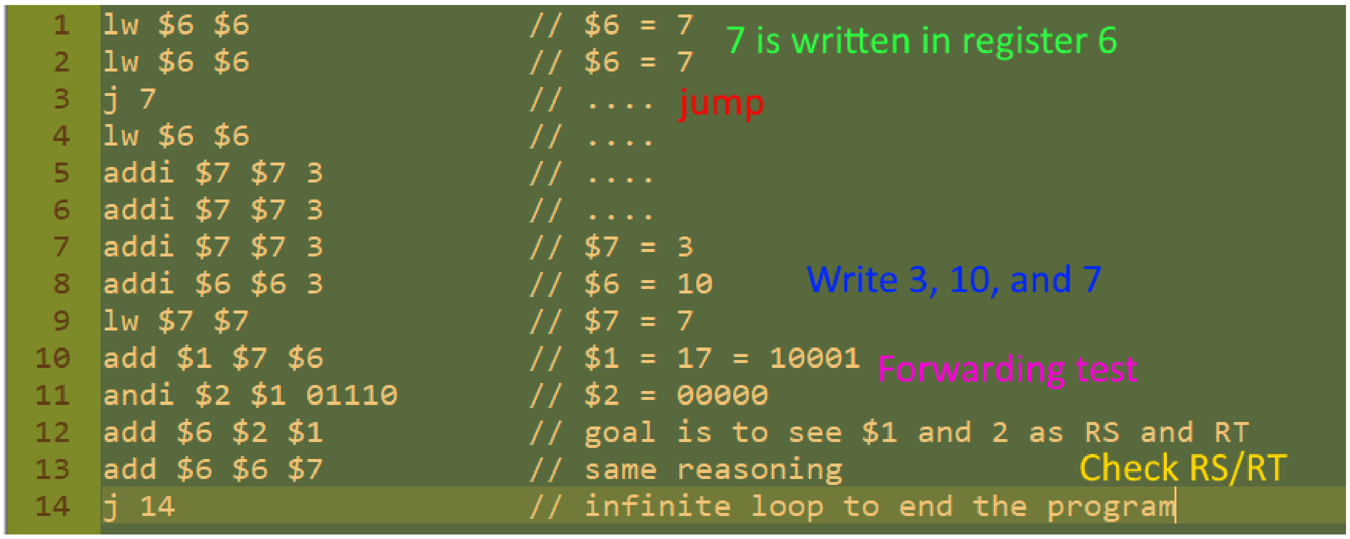
```
module ForwardingUnit(input logic ID_EX_MEM_RegWr,
                      input logic [2:0] ID_EX_MEM_Rt_or_rd,
                      input logic [2:0] IF_ID_Rs,
                      input logic [2:0] IF_ID_Rt,
                      input logic EX_MEM_WB_RegWr,
                      input logic [2:0] EX_MEM_WB_Rt_or_rd,
                      output logic [1:0] FA, FB);

always_comb
begin
if (ID_EX_MEM_RegWr & ID_EX_MEM_Rt_or_rd & (ID_EX_MEM_Rt_or_rd == IF_ID_Rs))
FA = 2'b10;
else if (EX_MEM_WB_RegWr & EX_MEM_WB_Rt_or_rd & (EX_MEM_WB_Rt_or_rd == IF_ID_Rs))
FA = 2'b01;
else
FA = 2'b00;

if (ID_EX_MEM_RegWr & ID_EX_MEM_Rt_or_rd & (ID_EX_MEM_Rt_or_rd == IF_ID_Rt))
FB = 2'b10;
else if (EX_MEM_WB_RegWr & EX_MEM_WB_Rt_or_rd & (EX_MEM_WB_Rt_or_rd == IF_ID_Rt))
FB = 2'b01;
else
FB = 2'b00;
end

endmodule
```

The forwarding unit outputs are used in this part of the program:



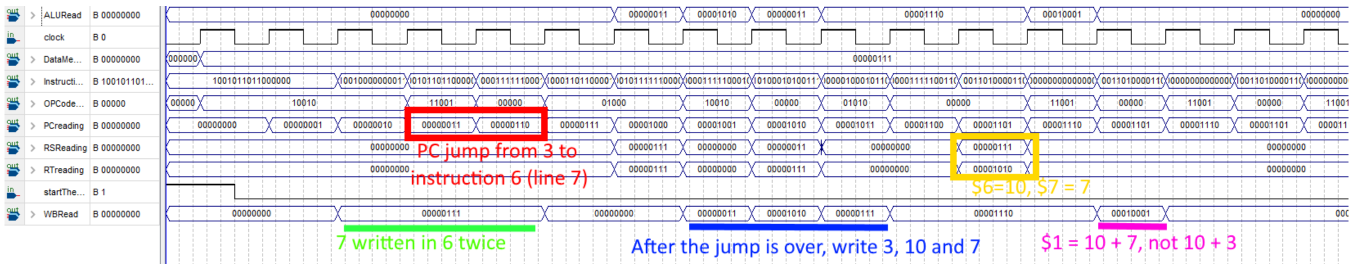
Finally, the simulation shows all of this working:

Remember the predictions from the Instruction Memory reading:



```
1  lw $6 $6             // $6 = 7
2  lw $6 $6             // $6 = 7
3  j 7                  // ....
4  lw $6 $6             // ....
5  addi $7 $7 3         // ....
6  addi $7 $7 3         // ....
7  addi $7 $7 3         // $7 = 3
8  addi $6 $6 3         // $6 = 10
9  lw $7 $7             // $7 = 7
10 add $1 $7 $6         // $1 = 17 = 10001
11 andi $2 $1 01110     // $2 = 00000
12 add $6 $2 $1         // goal is to see $1 and 2 as RS and RT
13 add $6 $6 $7         // same reasoning
14 j 14                 // infinite loop to end the program
```

We can see all of these in the simulation if we know where to look, thanks to all the inserted outputs:


ALURead B 00000000
clock B 0
DataMe... B 00000000
Instructi... B 100101101...
OPCode... B 00000
PCreading B 00000000
RSReading B 00000000
RTreading B 00000000
startThe... B 1
WBRead B 00000000
PC jump from 3 to instruction 6 (line 7)
$6=10, $7 = 7
7 written in 6 twice
After the jump is over, write 3, 10 and 7
$1 = 10 + 7, not 10 + 3