ELEC-5200

Computer Architecture

FROM: Jacob Howard

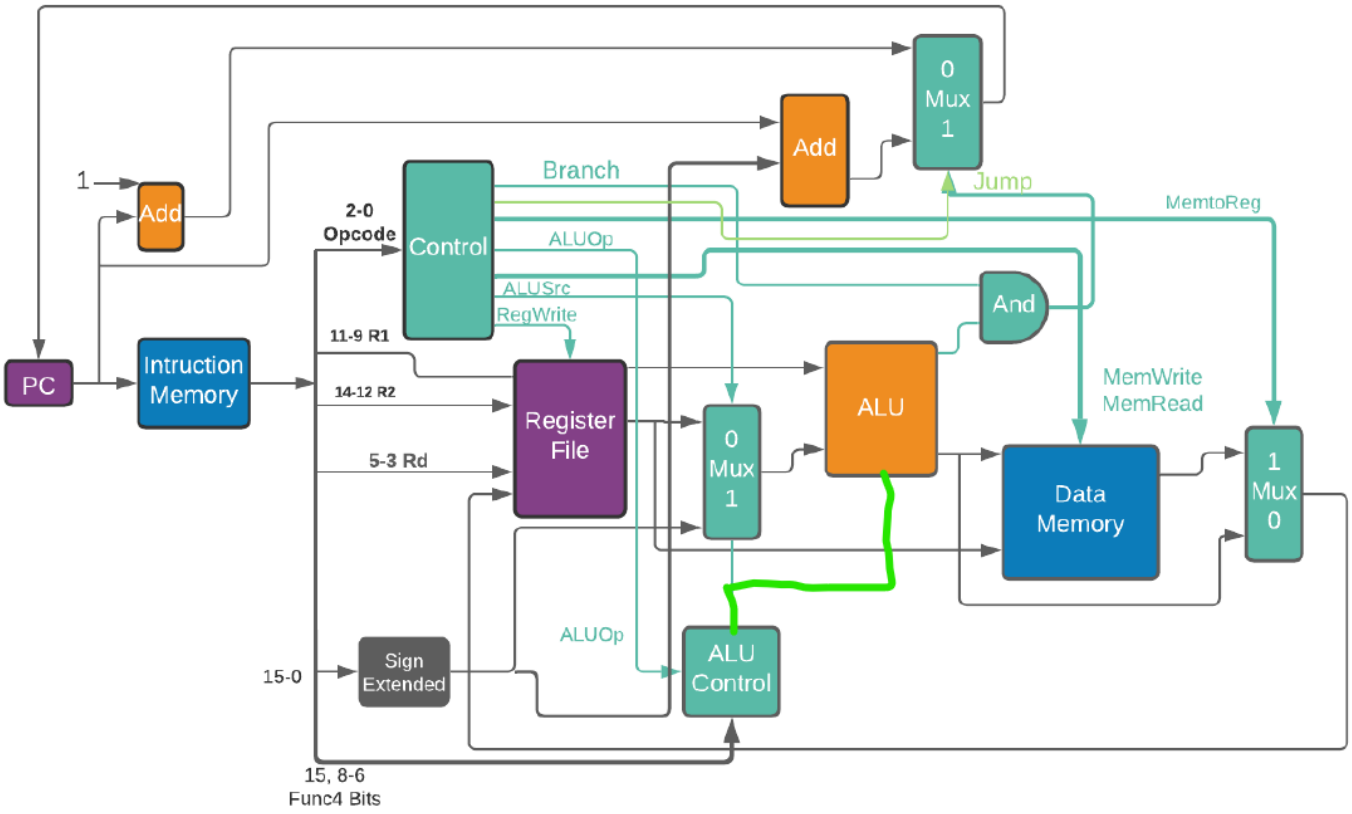
TO: Dr. Harris & Tucker Johnston

DUE DATE: 10/27/21

CPU Design – Part 3

Datapath

This is an overview of the Datapath design I created in Part 2. I noticed one flaw in the design where the ALU Control was connected to the RegisterFile/SignExtended MUX instead of to the ALU. The correction is drawn in highlighted pen to show the flaw and correction. I will officially fix this problem on the datapath design. Use this datapath as a reference to why designs were made in the Verilog Design.

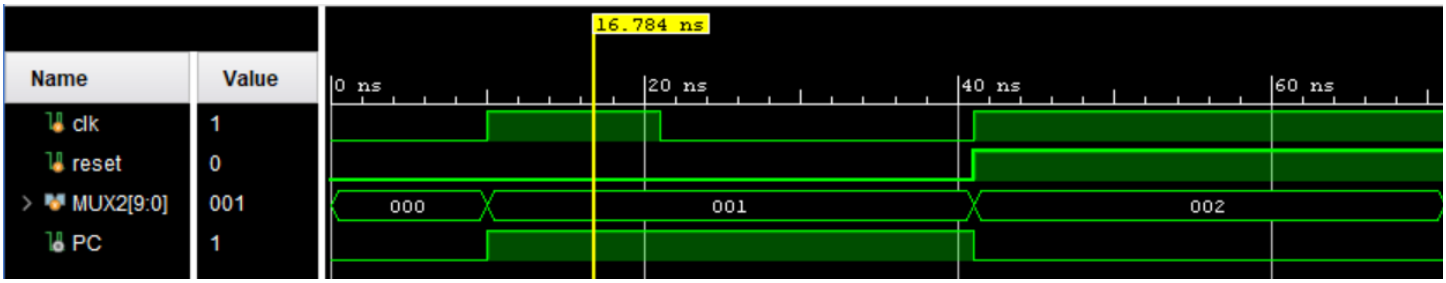


Verilog Design

I will go over my design for each major component in my datapath. All full Verilog code will be be shown at the bottom of this report, while I will show parts of my design here to describe design decisions and any simulations that may be useful.

PC

The PC was the first major component I design in Verilog and is the component that points to which instruction will be called next from the instruction memory. The PC is the only thing affected by clock and reset inputs. A simulation of the PC counting up 1 based off of the MUX output and being reset back to zero is shown below. The code can be found at the bottom of the report.

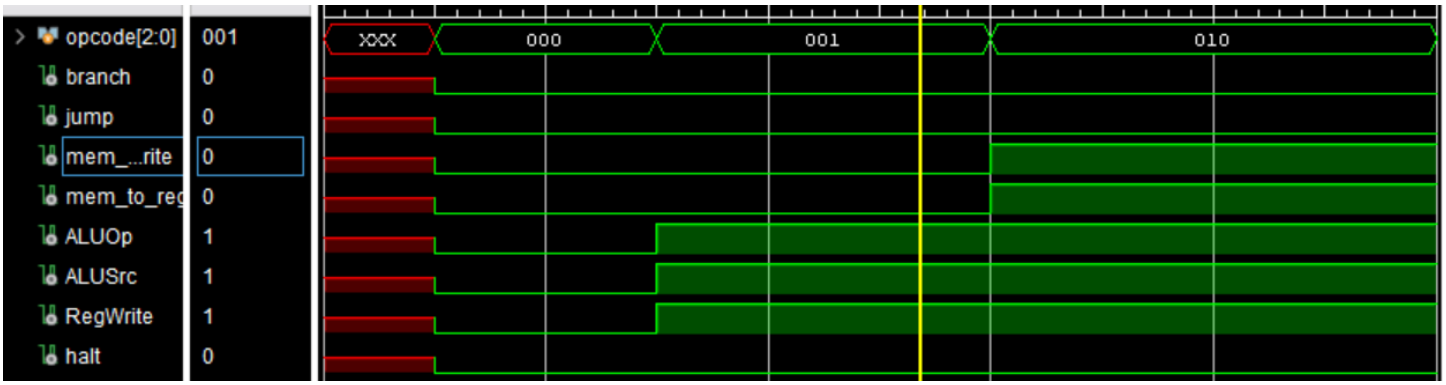


Intruction Memory

Next I implemented the instruction memory component. This component will store all instructions in a ROM file and read lines based on where the PC tells it to. It will output a 16-bit word instruction for all other components to use. The code is shown at the bottom of the report and is labbeled.

Control Unit

Next I created the control unit, which decodes the opcode. When decoded, different signals are sent out based on the type of instruction. The control unit determins whether the instruction is R-Type, I-Type, S-Type, SB-Type, or UJ-Type and sends signals to other components based on the decoded type. We can see in the simulation below that based on the opcode, the correct signals are sent out to the other components. The Control Unit code can be seen at the bottom of this report labeled.

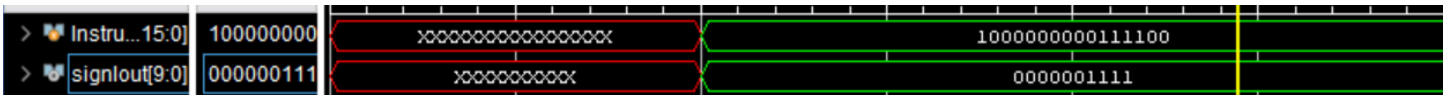


Sign Extended

For the Sign Extended block, I made some engineering decisions that should help my immediate values be correct regardless of type. I chose to decode the type in the sign extended block and based off the type, I would parse the immediate values accordingly and send the parsed immediate value to the output. Shown below is a snippet of code for parsing an SB-Type Instruction and sending the correct immediate vale to the ALU/Adder2. From the table shown below, you can see why it is important to parse the values correctly based on each type. In our simulation, we can see that the sign extended block correctly parses sb-type instructions. The simulation is shown below.

```
sbType: begin //SB-Type instuctions
        parseSB[3] = InstructionMemory[15];
        parseSB[2:0] = InstructionMemory[5:3];
        signIout[3:0] = parseSB[3:0]; //output
```

Type	15th bit	14-12	11-9	8-6	5-3	2-0 bits
R-Type	Funct4[3]	R2	R1	Func4[2:0]	Rd	opcode
I-Tpye	Imm[3:0]		R1	Func4[2:0]	rd	opcode
S-Type	Imm[3]	R2	R1	Func4[2:0]	Imm[2:0]	opcode
SB-Type	Imm[3]	R2	R1	Func4[2:0]	Imm[2:0]	opcode
UJ-Type	Imm[9:0]				rd	opcode



Register File

Next, I designed the register file. I designed the Register File to have 8 registeres, as that is all that my 16-bit word instructions will allow. I made inputs for reading address 1&2 and writing data correctly to the register file. I will show my code directly below as it is not too long and will explain how I designed all 8 registers.

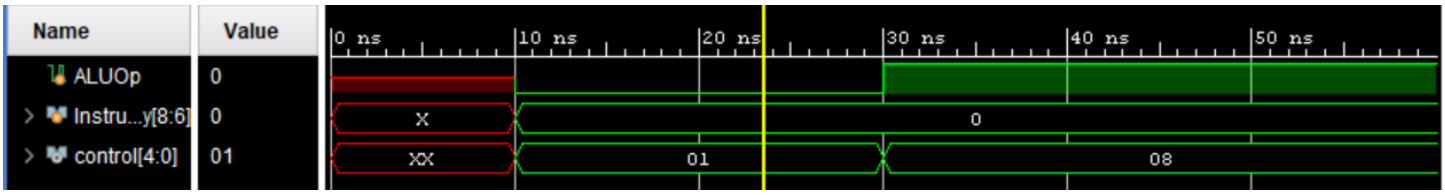
```
//inputs/outputs above this
reg [15:0] REG_FILE [0:7];

assign RegReadData1 = REG_FILE[RegReadAddr1];
assign RegReadData2 = REG_FILE[RegReadAddr2];

initial begin
   REG_FILE[0] = 0;
   REG_FILE[1] = 0;
   REG_FILE[2] = 0;
   REG_FILE[3] = 0;
   REG_FILE[4] = 0;
   REG_FILE[5] = 0;
   REG_FILE[6] = 0;
   REG_FILE[7] = 0;
```

```
end

    always @ (*) begin
        if (RegWrite) //if RegWrite is enabled
            REG_FILE[RegWriteAddr] = RegWriteData;
           REG_FILE[0] = 0;
    end

endmodule
```

ALU Control

Before I talk about the ALU, I need to descus my design for the ALU Control Unit and how it works. I designed my ALU Control to decode my Function Bits based off the type. Since my function bits are only 4-bits long and other types share the same function bits, I had to first decode the type of instruction. Once the type was determined, I could then decode the exact instruction to be performed. The way I did this was have my Opcode Control Unit send a number (0-7) to the ALU Control to determine the type. For example, an ALUOp signal of zero would be arithmetic type, while a signal of 1 would be immediate arithmetic, and so on. Once the type was determined, I looked at the function bits to decode the exact instruction to send to the ALU. The ALU will get a signal of 1-20 based on the exact instruction it should perform. 1 would we an ADD instruction where as 20 would be the JALR instruction.

We can see from the simulation below, when testing for an ADD instruction, we get an output of 1, and when testing for an ADDI instruction we get an output of 8 as expected. This is how the ALU will know what operation to perform. The code can be seen at the bottom of the report, labbeled.

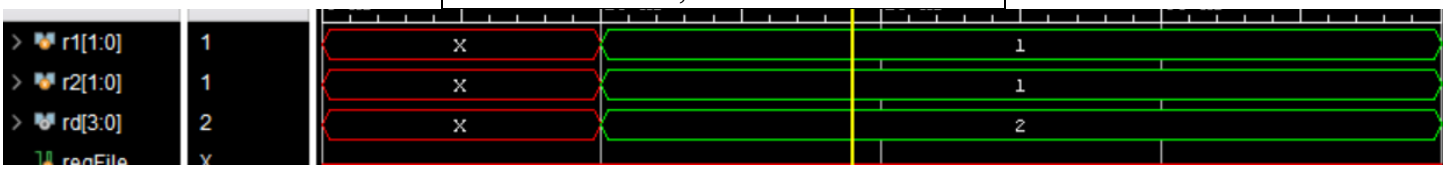


ALU

Lastly, I designed the ALU. The ALU takes the output from the ALU control to determine the instruction to do. Once the instruction is determined wit will take register values and perform operations. A code snippet of instruction values are shown below and the entire code van be seen at the bottom of the report. When testing the ADDi instruction in simulation (making sure the ALUControl signal is 1 for ADD), we see that r1+r2 is performed and stored into rd. The simulation can be seen below.

```
//Arithmatic
        ADD = 1,
        SUB = 2,
        AND = 3,
        OR = 4,
        XOR = 5,
        SL = 6,
```

```
SR = 7,


//Immediate Arithmatic
ADDI = 8,
ANDI = 9,
ORI = 10,
XORI = 11,
SLI = 12,
SRI = 13,
//load/store instrutions
LOAD = 14,
STORE = 15,
//Branch Instructions
BEQ = 16,
BGT = 17,
BGE = 18,
BLT = 19,
//JALR Instruction
JALR = 20;
```



All Other Componants

The Data Memory was not required to be completed in this part, so it is not finished for this design. The PC+1 adder and PC+SignExtended adder are not shown along with the MUX's and AND Gate as these are not "Main" components. They are designed in Verilog, but are not required to show in this report and their functionality is easy to understand based off looking at the DataPath.

All Verilog Code

PC

```
module PC(
input clk, //the clock
input reset, //resets PC
input [9:0] MUX2, //the input from MUX2 will change the count on PC
output reg PC  //the output of pc will tell instuction memory where to point
);

begin
always @(clk | reset)

if (reset == 1) begin //if reset is active, reset pc to 0
PC = 0; //resets PC to 0
end

else begin  //if reset is not active, then PC will equal MUX2
PC = MUX2; //mux2 is the PC mux. It will either equad PC+1 or PC +branch/jump location
end
end
endmodule
```

Intruction Memory

```
module IntructionMemory(
input [15:0] Addr,
input [15:0] data
);

reg [15:0] ROM [0:65535];

initial $readmemb("intruction_mem.mem", ROM);

assign data = ROM[Addr];

endmodule
```

Control Unit

```
module ControlUnit(
input [2:0] opcode,
output reg branch,
output reg jump,
output reg mem_read_write,
output reg mem_to_reg,
output reg ALUOp,
```

```
output reg ALUSrc,
output reg RegWrite,
output reg halt
);
//Opcode of all instructions
parameter ALU_A = 3'b000,
        ALU_I = 3'b001,
        LOAD = 3'b010,
        STORE = 3'b011,
        BRANCH = 3'b100,
        JAL = 3'b101,
        JALR = 3'b110,
        HALT = 3'b111;

always @ (opcode) //opcode change
begin
branch = 0;
jump = 0;
mem_read_write = 0; //do nothing
ALUOp = 0;
ALUSrc = 0;
RegWrite = 0;
halt = 0;

case (opcode)

   ALU_A: begin //Arithmatic Intructions
   ALUOp = 0; //operation 0, Arithmatic instructions
   ALUSrc = 0;
   RegWrite = 0; //sets zero on mux1 (register/sign muz)
   mem_to_reg = 0;
   end

   ALU_I: begin //Immediate Instructions
   ALUOp = 1; //operation 1, immidiate arithmatic
   ALUSrc = 1; //gets immidiate sign extended from mux
   RegWrite = 1;
   mem_to_reg = 0;
   end

   LOAD: begin //Load Word Instruction
   ALUOp = 3; //operation 3, load word
   ALUSrc = 1; //for immidiates
   mem_read_write = 1; //read memory
   RegWrite = 1;
   mem_to_reg = 1; //memory
   end
```

```
    STORE: begin //Store Word Instruction
    ALUOp = 5;  //operation 5, store word
    ALUSrc = 1;
    mem_read_write = 2; //write memory (2 because we want zero to not read nor write to memory)
    end

    BRANCH: begin //branch instructions
    branch = 1;
    ALUOp = 6; //operation 6, branch
    ALUSrc = 1;
    end

    JAL: begin //Jump and Link Intruction
    jump = 1;
    RegWrite = 1;
    mem_read_write = 2;
    end

    JALR: //Jump and Link Register Intruction
    begin
    ALUOp = 7; //operation 7, jump and link register (regular jump does not require operation)
    ALUSrc = 1; //Immediate
    RegWrite = 1;
    mem_read_write = 2;
    end

    HALT: begin
    halt = 1; //halts everything :)
    end

endcase

end

endmodule
```

Sign Extended Unit

```
module SignExtended(
input [15:0] InstructionMemory,
output reg [9:0] signIout //sign extended immediates output. Max value should not be over 10-bits
);
//parsing data for immidiate values
reg [3:0] parseI;
reg [3:0] parseSB;
reg [9:0] parseUJ;
reg [3:0] parseLW;
reg [3:0] parseSW;
```

```
//opcode
reg [2:0] opcode;


//parameters used to know how to parse immediate output
parameter iType = 3'b001,
        sbType = 3'b100,
        ujType = 3'b101,
        lw = 3'b010,
        sw = 3'b011;

always @ (InstructionMemory)
begin
//clears everything to zero
signIout = 0; //setting output to zero
opcode = 0;
parseI = 0;
parseSB = 0;
parseUJ = 0;
parseLW = 0;
parseSW = 0;
//setting opcode
opcode[2:0] = InstructionMemory[2:0]; //assigning opcode as first 3-bits from memory instructions
case (opcode)

   iType: begin //I-Type instructions
       parseI[3:0] = InstructionMemory[15:12];
       signIout[3:0] = parseI[3:0]; //sets output to the parsed immidiates
   end

   sbType: begin //SB-Type instuctions
      parseSB[3] = InstructionMemory[15];
      parseSB[2:0] = InstructionMemory[5:3];
      signIout[3:0] = parseSB[3:0];
   end

   ujType: begin //UJ-Type instruction
      parseUJ[9:0] = InstructionMemory[15:6];
      signIout[9:0] = parseUJ[9:0];
   end

   lw: begin //load word is i-type parsing
      parseLW[3:0] = InstructionMemory[15:12];
      signIout[3:0] = parseLW[3:0];
   end

   sw: begin //store word is s-type parsing and same as sb-type parsing
      parseSW[3] = InstructionMemory[15];
```

```
        parseSW[2:0] = InstructionMemory[5:3];
        signIout[3:0] = parseSW[3:0];
    end
endcase

end
endmodule
```

ALU Control

```
module ALUControl( //tells ALU what operation to perform
input ALUOp,
input [8:6] InstructionMemory,
output reg [4:0] control
);

parameter //Arithmatic
        A = 0, //Arithmatic instructions
        ADD = 4'b0000,
        SUB = 4'b0001,
        AND = 4'b0010,
        OR = 4'b0011,
        XOR = 4'b0100,
        SL = 4'b0101,
        SR = 4'b0110,
        //Immidiate Arithmatic
        I = 1,
        ADDI = 4'b0000,
        ANDI = 4'b0001,
        ORI = 4'b0010,
        XORI = 4'b0011,
        SLI = 4'b0100,
        SRI = 4'b0101,
        //Load Instruction
        LOAD = 3,//only 1 instruction
        //Store Instruction
        STORE = 4,//only 1 instruction
        //Branch
        BRANCH = 6,
        BEQ = 4'b0000,
        BGT = 4'b0001,
        BGE = 4'b0010,
        BLT = 4'b0011,
        //Jump and Link Reg
        JALR  = 7; //only 1 instruction

always @ (ALUOp)
begin
```

```
control = 0; //sets control to zero at start so no instructions are performed
case (ALUOp)

   A: begin //arithmatic
      case (InstructionMemory) //Intruction decoder
         ADD: begin
         control = 1;
         end
         SUB: begin
         control = 2;
         end
         AND: begin
         control = 3;
         end
         OR: begin
         control = 4;
         end
         XOR: begin
         control = 5;
         end
         SL: begin
         control = 6;
         end
         SR: begin
         control = 7;
         end
      endcase //end instruction decoder case
   end

   I: begin //Immediate math
         case (InstructionMemory) //Intruction decoder
            ADDI: begin
            control = 8;
            end
            ANDI: begin
            control = 9;
            end
            ORI: begin
            control = 10;
            end
            XORI: begin
            control = 11;
            end
            SLI: begin
            control = 12;
            end
            SRI: begin
            control = 13;
```

```
            end
          endcase //end instruction decoder case
       end

       LOAD: begin //Load Word
            control = 14;
            end
       STORE: begin //Load Word
             control = 15;
             end

    BRANCH: begin //Branch Instructions
       case(InstructionMemory) //decoder case
          BEQ: begin
          control = 16;
          end
          BGT: begin
          control = 17;
          end
          BGE: begin
          control = 18;
          end
          BLT: begin
          control = 19;
          end
       endcase//end instruction decoder case
          end

     JALR: begin
       control = 20;
       end

endcase //endcase for ALUOperation case

end

endmodule
```

ALU

```
module ALU (
//temp input/output of reg for testing
input [1:0] r1,
input [1:0] r2,
output reg [3:0] rd,

input regFile,
input mux2, //mux for regFile/SignExtended
```

```
input ALUControl,
output reg data,
output reg branch_gate //compair output for branches
);

//wire r1;
//wire r2;
wire i = mux2; //immidiate (sign extended will handle immidiate field from opcode)
//reg rd;

parameter //Arithmatic
        ADD = 1,
        SUB = 2,
        AND = 3,
        OR = 4,
        XOR = 5,
        SL = 6,
        SR = 7,
        //Immediate Arithmatic
        ADDI = 8,
        ANDI = 9,
        ORI = 10,
        XORI = 11,
        SLI = 12,
        SRI = 13,
        //load/store instrutions
        LOAD = 14,
        STORE = 15,
        //Branch Instructions
        BEQ = 16,
        BGT = 17,
        BGE = 18,
        BLT = 19,
        //JALR Instruction
        JALR = 20;


always @(ALUControl) //ALUControl will output a number from 0-20 so the ALU will know what operations to
perfrom
begin
//setting ALU outputs to 0 at the begining
data = 0;
branch_gate = 0;

case (ALUControl)
   //Arithmatic
   ADD: begin
   rd = r1+r2;
```

```
    end
    SUB: begin
    rd = r1-r2;
    end
    AND: begin
    rd = r1&r2;
    end
    OR: begin
    rd = r1 | r2;
    end
    XOR: begin
    rd = r1^r2;
    end
    SL: begin
    rd = 1<<r1;
    end
    SR: begin
    rd = 1>>r1;
    end
    //Immediate Arithmatic
    ADDI: begin
    rd = r1+i;
    end
    ANDI: begin
    rd = r1&i;
    end
    ORI: begin
    rd = r1 | i;
    end
    XORI: begin
    rd = r1^i;
    end
    SLI: begin
    rd = 1<<i;
    end
    SRI: begin
    rd = 1>>i;
    end
    //Load/Store Word
    LOAD: begin
    rd = r1+i;
    end
    STORE: begin
    //finish later
    end
    //Branch
    BEQ: begin
       if (r1 == r2) begin
```

```
      branch_gate = 1; //allowing a branch because terms are met
      end
      else begin
      branch_gate = 0; //terms not met, not allowing branch
      end
   end
   BGT: begin
      if (r1 > r2) begin
      branch_gate = 1; //allowing a branch because terms are met
      end
      else begin
      branch_gate = 0; //terms not met, not allowing branch
      end
   end
   BGE: begin
      if (r1 >= r2) begin
      branch_gate = 1; //allowing a branch because terms are met
      end
      else begin
      branch_gate = 0; //terms not met, not allowing branch
      end
   end
   BLT: begin
      if (r1 > r2) begin
      branch_gate = 1; //allowing a branch because terms are met
      end
      else begin
      branch_gate = 0; //terms not met, not allowing branch
      end
   end
   //Jump and Link Register
   JALR: begin
   //finish
   end
endcase

end
endmodule
```