ELEC-5200

Computer Architecture

FROM: Jacob Howard

TO: Dr. Harris & Tucker Johnston

DUE DATE: 11/24/21

CPU Design – Part 5

Introduction / Data Memory Design

This is the final part of our CPU Design. The goal was to add memory to our design and verify full functionality. Since I had previously added instruction memory in part 4, all I needed to add was data memory. My data memory has 32 total spaces, each capable of holding one 16-bit word. Since the design is RAM, the memory is capable of having locations overwritten with new data, and storing data in registers.

The design runs on a clock and takes the ALU output and the output of the Register File to determine what and where to write data into the memory. The design can be seen below labeled.

```
`timescale 1ns / 1ps

module DataMemory(
input clk,
input writeEn, //write enable
input  [15:0] ALU, //read/write address (ALU output)
input  [15:0] r2, //data input (R2 from regFile)
output [15:0] dataMemOut  //data output
);
reg    [15:0] ram [31:0];

      always @(posedge clk) begin
        if (writeEn)
          ram[ALU] = r2;
      end
      assign dataMemOut = ram[ALU];
    endmodule
```

Data Memory Code

Testing

Now that everything has been completed, we can go on to write a test program to test all instructions. We were required to do this in Part 4, but with the addition to Data Memory, we can test our load word and store word instructions.

Below, you can find a chart for the format of all instructions. This chart might be useful in understanding how instructions are formatted. Also, shown below is a list of binary/assembly instructions that will be used to test our CPU design.

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

Instruction Format

```
//initial testing
0001 001 000 001 001 //addi x1, x1, 1
0 01 000 000 000 011 //sw x1, x0, 0 (stores x1 into location 0 at memory)
0000 000 000 010 010 //lw x2, x0, 0 (stores MEM[0] into x2)

//arithmetic instructions
f4 r2    r1    f3   rd    op
0 010 001 000 001 000 //add x1, x2, x1 (expected 1+1 = 2)
0 001 001 001 000 000 //sub x1, x1, x0 (expected 2-2 = 0)
0 001 001 010 001 000 //and x1, x1, x1 (expected 2&2 = 2)
0 010 000 011 001 000 //or x0, x2, x1 (expected 1|0 = 1)
0 000 001 101 011 000 //sl x1, x3 (expected <<1 = 2)
0 000 011 110 001 000 //sr x3, x1 (expected >>2 = 1)

//immediate instructions
imm  r1   f3  rd  op
0001 001 001 001 001 //andi x1, x1, 1 (expected 1&1 = 1)
0001 000 010 001 001 // ori x1, x1, 1 (expected 1|1 = 1)

//branch instructions
im3 r2 r1 fun3 imm2 op
0 001 001 000 001 100 //beq x1, x1, 1 (expected branch 1&1)
0 000 001 001 001 100 //bgt x0, x1, 1 (expected 0<1, so no branch)

//Jump
 imm            rd  op
000000001 100 101 //jal x4, 0 (expected jump 1 and put pc into x4)

//halt
xxxxxxxxxxxxx 111 //halt (expected pc halts)
```

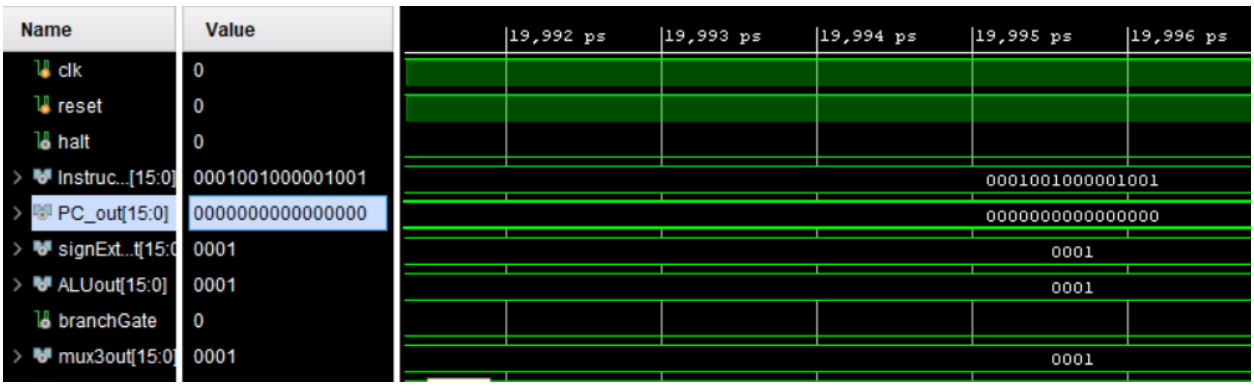
Test Instructions

Simulations

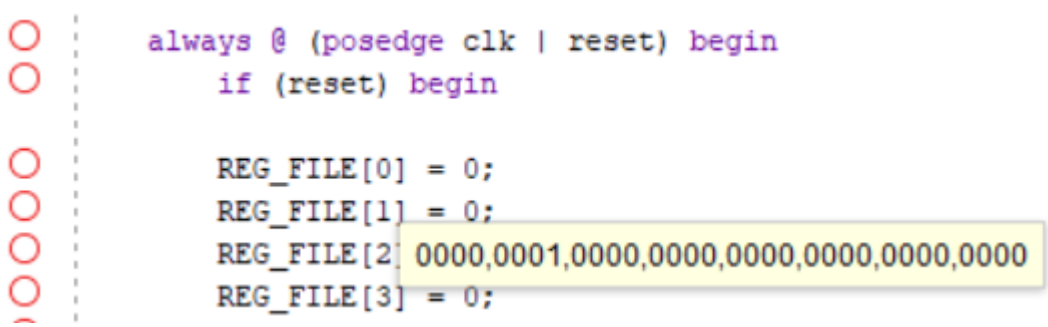
Now I will test these instructions in simulation to verify functionality. All simulation screenshots will be labeled with what each instruction was and what the expected outcome is. After all instructions have been tested, I will test branches that should not branch. I will leave a halt instruction right after the branches, so if I do not branch over, due to a branch not taken scenario, the whole process will stop. If the branch is taken, I will do a simple add instruction. This will verify that my branches work properly.

Initial Testing

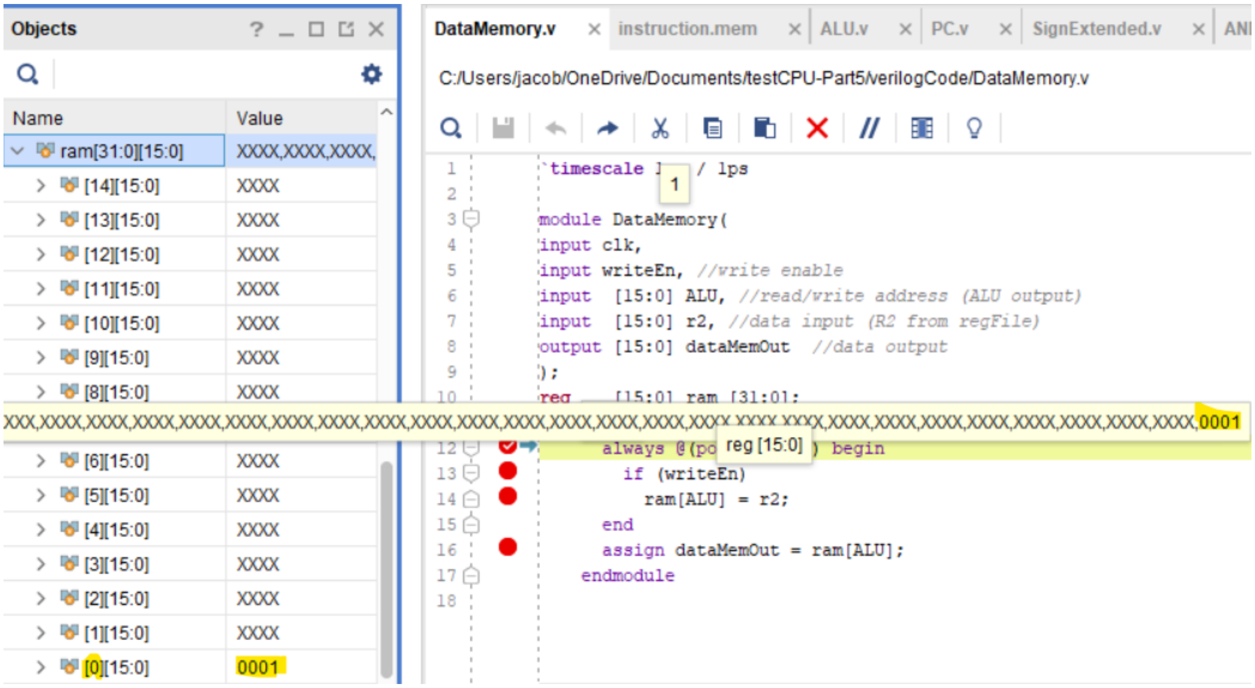
In the initial testing, we want to make sure our new memory module works correctly. So we will use an add immediate instruction to store 1 into x1 and store the value into our data memory. Once we have achieved that, we will load the value of 1 from our memory we stored into a new register to test functionality.



Addi x1, x1, 1 (mux3 outputs 1 into x1 as expected)



Showing x1 has a value of 1 after addi

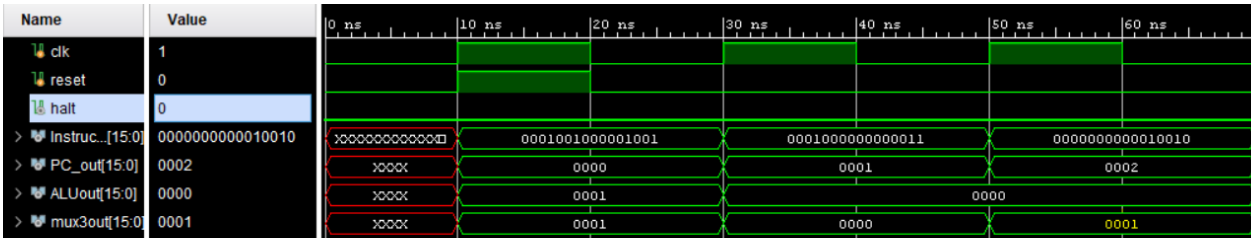


Name	Value
ram[31:0][15:0]	XXXX,XXXX,XXXX,
[14][15:0]	XXXX
[13][15:0]	XXXX
[12][15:0]	XXXX
[11][15:0]	XXXX
[10][15:0]	XXXX
[9][15:0]	XXXX
[8][15:0]	XXXX
[6][15:0]	XXXX
[5][15:0]	XXXX
[4][15:0]	XXXX
[3][15:0]	XXXX
[2][15:0]	XXXX
[1][15:0]	XXXX
[0][15:0]	0001

C:/Users/jacob/OneDrive/Documents/testCPU-Part5/verilogCode/DataMemory.v

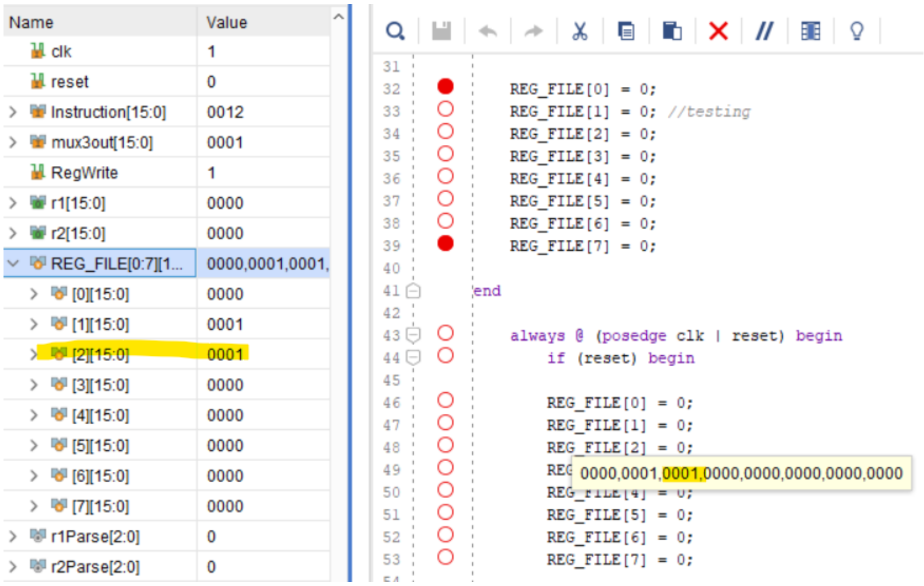
```
`timescale 1  1 / 1ps

module DataMemory(
input clk,
input writeEn, //write enable
input  [15:0] ALU, //read/write address (ALU output)
input  [15:0] r2, //data input (R2 from regFile)
output [15:0] dataMemOut  //data output
);
reg    [15:0] ram [31:0];
XXX,XXXX,XXXX,XXXX,XXXX,XXXX,XXXX,XXXX,XXXX,XXXX,XXXX,XXXX,XXXX,XXXX,XXXX,XXXX,XXXX,XXXX,XXXX,XXXX,XXXX,XXXX,XXXX,XXXX,XXXX,XXXX,0001
    always @(po reg [15:0] ) begin
      if (writeEn)
        ram[ALU] = r2;
    end
    assign dataMemOut = ram[ALU];
  endmodule
```

Sw x1, x0, 0 (storing x1 into MEM[0] works as expected)



Name	Value
clk	1
reset	0
halt	0
Instruc...[15:0]	0000000000010010
PC_out[15:0]	0002
ALUout[15:0]	0000
mux3out[15:0]	0001

Lw x2, x0, 0 (we see mux3's outvalue being MEM[0]s value which will be stored in x2)



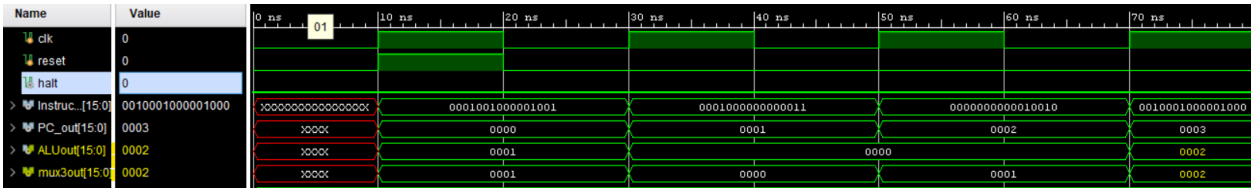
Name	Value
clk	1
reset	0
Instruction[15:0]	0012
mux3out[15:0]	0001
RegWrite	1
r1[15:0]	0000
r2[15:0]	0000
REG_FILE[0:7][1...	0000,0001,0001,
[0][15:0]	0000
[1][15:0]	0001
[2][15:0]	0001
[3][15:0]	0000
[4][15:0]	0000
[5][15:0]	0000
[6][15:0]	0000
[7][15:0]	0000
r1Parse[2:0]	0
r2Parse[2:0]	0

```

    REG_FILE[0] = 0;
    REG_FILE[1] = 0; //testing
    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 @ (posedge clk | reset) begin
        if (reset) begin

            REG_FILE[0] = 0;
            REG_FILE[1] = 0;
            REG_FILE[2] = 0;
            REG 0000,0001,0001,0000,0000,0000,0000,0000
            REG_FILE[4] = 0;
            REG_FILE[5] = 0;
            REG_FILE[6] = 0;
            REG_FILE[7] = 0;
```

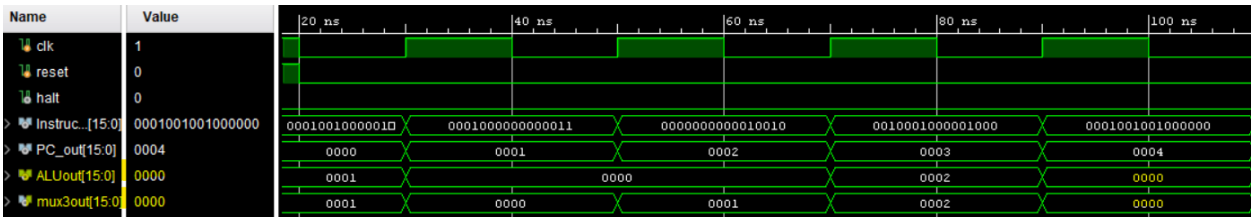
Lw stores MEM[0] into x2 as expected

Full Program Testing

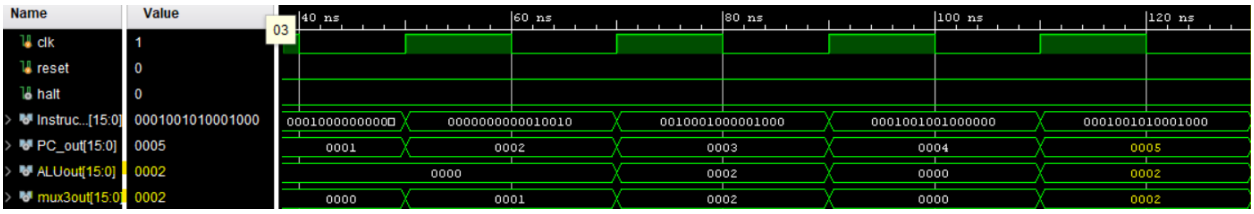
Now that we have successfully verified that our new memory module works and we have completed a full CPU design, we will test all other instructions to make sure our CPU works properly. We will also test branch not taken, as asked by the professor. All other instructions have been listed above, but will also be labeled under screenshots.



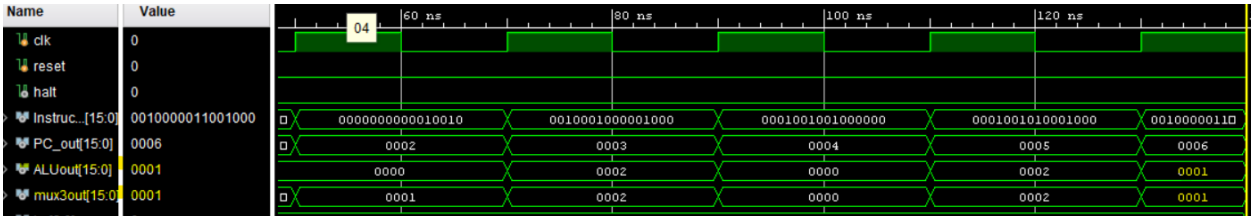
Add x1, x2, x1 (adding 1+1 and storing in x1 gives us 2 as expected in ALU output and MUX3)



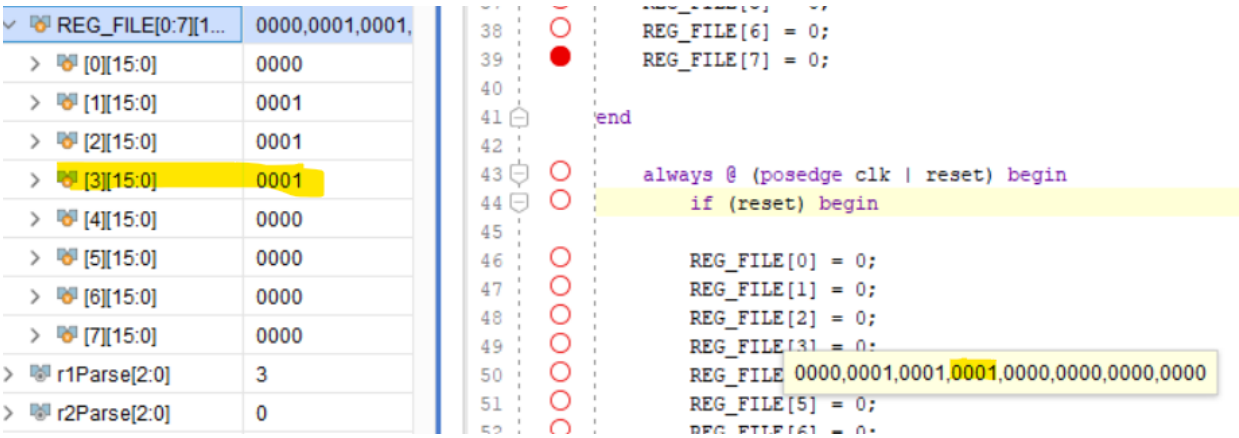
Sub x1, x1, x0 (2-2 eq 0, shown in ALUout and MUX3out, as expected and value is stored in x0)



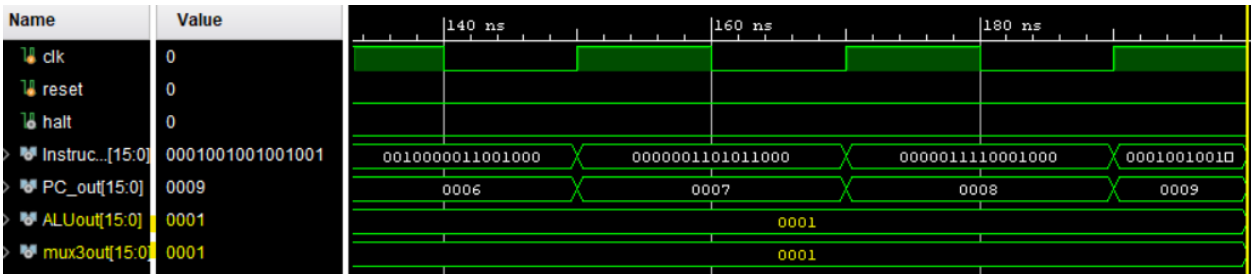
And x1, x1, x1 (AND'ing 2 and 2 gives us 2 in ALU output and MUX3 output which will be stored in x1)



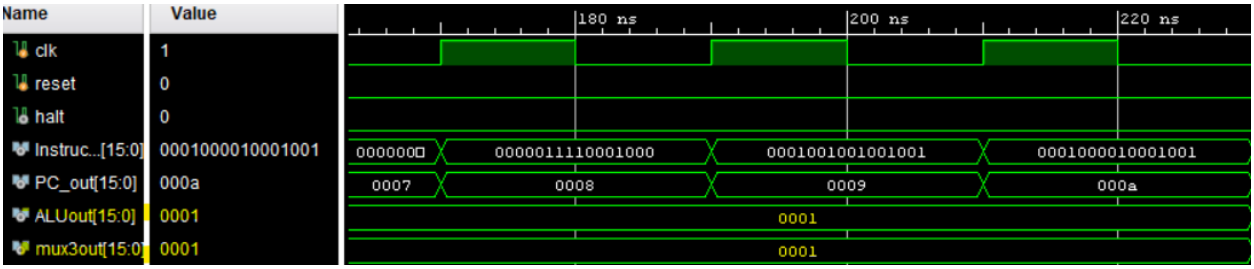
Or x2, x0, x1 (OR'ing 1 and 0 gives us 1 in ALU output as expected and is stored in x1)



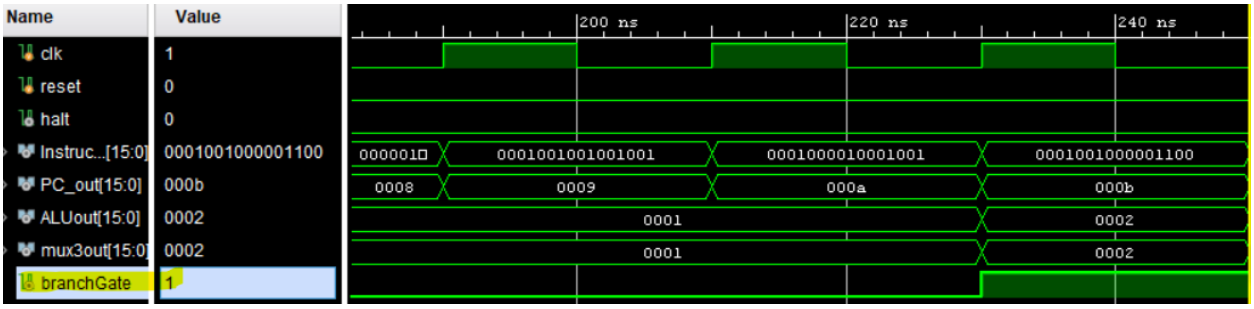
When performing both shifts, we can see we store the value of 1 in x3 as desired



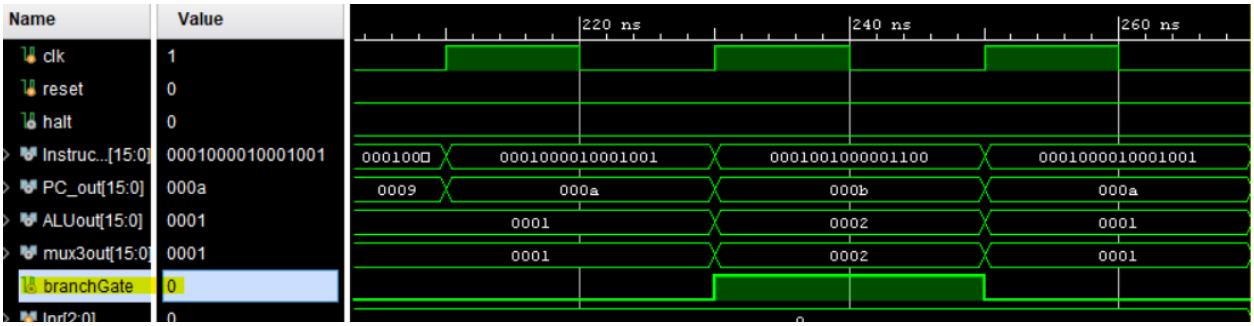
Andi x1, x1, 1 (AND'ing x1 and 1 together and get 1 as output as expected)



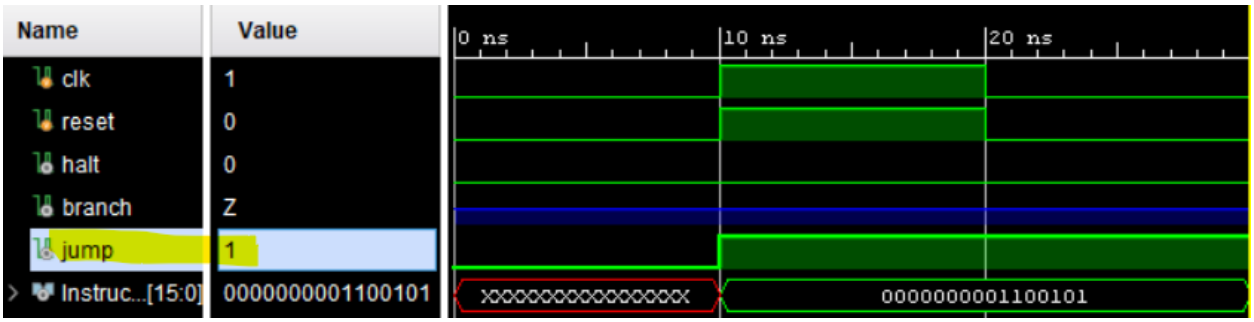
Ori x1, x1, x1 (OR'ing 1 and 1 and get 1 as expected)



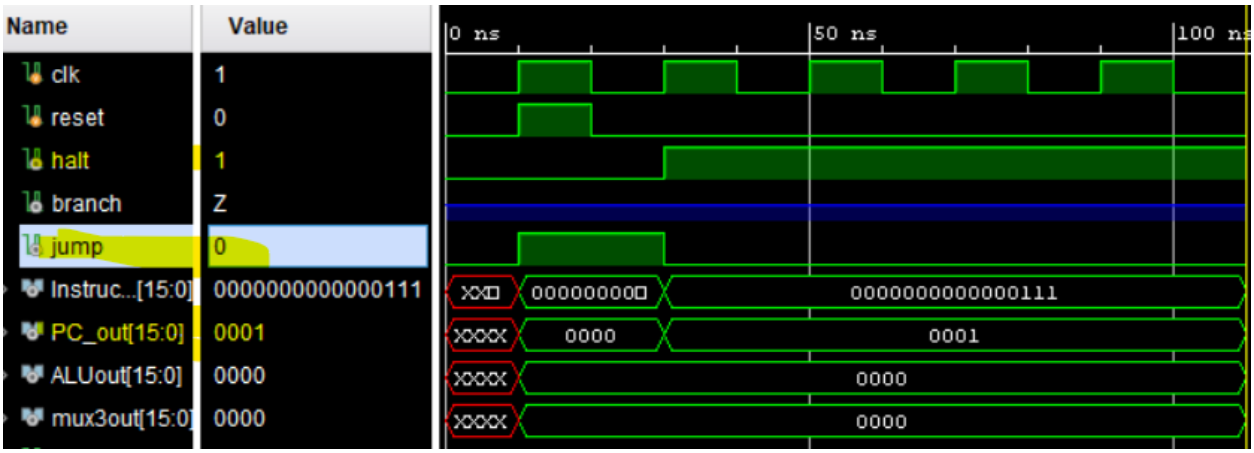
Beq x1, x1, 1 (as we can see, the branch gate sends a signal when requirements are met, and allows a branch. In this case, we only branch once from PC)



Bge x0, x1, 1 (as expected, we do not branch (branchGate = 0) as 0 is less than 1)



Jal x4, 1 (jump works as expected, jumping 1, as seen by the screenshot)



Halt (we can see after jumping to halt instruction, the clock continues to pulse, but the pc does not continue to count. The CPU is halted from any further instructions)

Conclusion

As we can see, all instructions work as expected. We also tested a branch not taken scenario to prove that we will not branch if values do not allow for a branch. With this part of the CPU Project done, we have successfully designed a working 16-bit CPU.