# ISTANBUL TECHNICAL UNIVERSITY

# COMPUTER ENGINEERING DEPARTMENT

## BLG 242E

## DIGITAL CIRCUITS LABORATORY EXPERIMENT REPORT

| | | |
|---|---|---|
| **EXPERIMENT NO** | : | 7 |
| **EXPERIMENT DATE** | : | 07.05.2021 |
| **LAB SESSION** | : | FRIDAY - 10.30 |
| **GROUP NO** | : | G17 |

## GROUP MEMBERS:


150180024 : ŞULE BEYZA KARADAĞ

150190710 : SENİHA SERRA BOZKURT

150190024 : AHMET FURKAN KAVRAZ


**SPRING 2021**

# Contents

# 1 INTRODUCTION [10 points]

In the following experiment, we are going to implement data buses and basic memory by using three-state buffers in Verilog. We will learn:

- How to analyze a given circuit with data bus,
- How to implement a Register Line module,
- How to implement a Register File module using Register Line module and a DECODER,
- How to implement an ALU Design and using table given to us, as operations,
- How to implement an Instruction Decoder,
- How to implement a Program Counter,
- And by using all the implementations above, how to implement Mini Computer.

# 2 MATERIALS AND METHODS [40 points]

## 2.1 Experiment

### 2.1.1 Modules we Used:

```
/*          part1          */
module not_gate(
    input wire a,
    output wire not_a
    );
    assign not_a = ~a;
endmodule
module and_gate_4input(
    input wire a,
    input wire b,
    input wire c,
    input wire d,
    output wire o
    );
    assign o = a && b && c && d;
endmodule
```

Figure 1: NOT gate and AND gate module

### 2.1.2 Part 1

In this part, we implemented:

- 4:16 decoder module with enable input. If enable signal is logical low, all the outputs of decoder will be logical low.
- 16-bit register line module. This module takes lineselect, clock, reset, and 16-bit dataIn as input and gives 16-bit stored data as output.
  - At the falling edge of the reset signal data will be cleared.
  - At the rising edge of the clock signal, the module stores the data which is given as input when the lineselect is high.

```
module decoder_4input(
    input wire [3:0] s,
    output wire [15:0] O
    );
    wire not_s0, not_s1, not_s2, not_s3;

    not_gate not0(s[0], not_s0);
    not_gate not1(s[1], not_s1);
    not_gate not2(s[2], not_s2);
    not_gate not3(s[3], not_s3);

    and_gate_4input and0(not_s3, not_s2, not_s1, not_s0, O[0]); // 0000
    and_gate_4input and1(not_s3, not_s2, not_s1, s[0]  , O[1]); // 0001
    and_gate_4input and2(not_s3, not_s2, s[1]  , not_s0, O[2]); // 0010
    and_gate_4input and3(not_s3, not_s2, s[1]  , s[0]  , O[3]); // 0011
    and_gate_4input and4(not_s3, s[2]  , not_s1, not_s0, O[4]); // 0100
    and_gate_4input and5(not_s3, s[2]  , not_s1, s[0]  , O[5]); // 0101
    and_gate_4input and6(not_s3, s[2]  , s[1]  , not_s0, O[6]); // 0110
    and_gate_4input and7(not_s3, s[2]  , s[1]  , s[0]  , O[7]); // 0111
    and_gate_4input and10(s3, not_s2, not_s1, not_s0, O[0]); // 1000
    and_gate_4input and11(s3, not_s2, not_s1, s[0]  , O[1]); // 1001
    and_gate_4input and12(s3, not_s2, s[1]  , not_s0, O[2]); // 1010
    and_gate_4input and13(s3, not_s2, s[1]  , s[0]  , O[3]); // 1011
    and_gate_4input and14(s3, s[2]  , not_s1, not_s0, O[4]); // 1100
    and_gate_4input and15(s3, s[2]  , not_s1, s[0]  , O[5]); // 1101
    and_gate_4input and16(s3, s[2]  , s[1]  , not_s0, O[6]); // 1110
    and_gate_4input and17(s3, s[2]  , s[1]  , s[0]  , O[7]); // 1111
endmodule
```

Figure 2: 4:16 decoder module with enable input. If enable signal is logical low, all the outputs of decoder will be logical low.

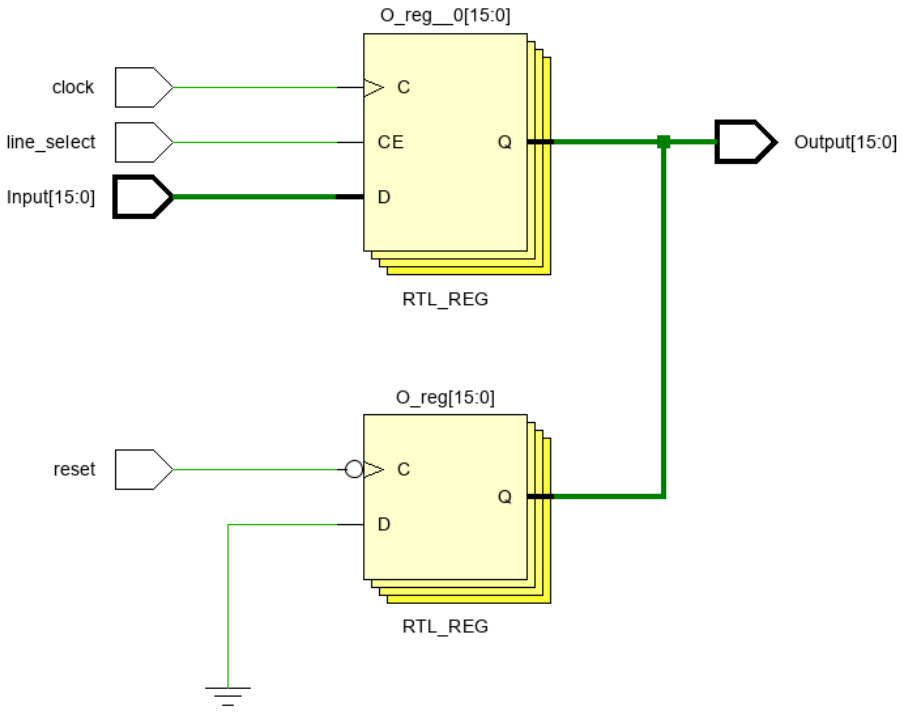



Figure 3: 16-bit register line module that takes lineselect, clock, reset, and 16-bit dataIn as input and gives 16-bit stored data as output

```
module register_line_module(
    input wire [15:0] Input,
    input wire line_select,
    input wire clock,
    input wire reset,
    output wire [15:0] Output
    );
    reg [15:0] O;

    initial begin O = 16'dZ; end
    always @(posedge clock) begin
      if (line_select) begin
          O = Input;
      end
    end

    always @(negedge reset) begin
        O = 16'dZ;
    end

    assign Output = O;

endmodule
```

Figure 4: 16-bit register line module

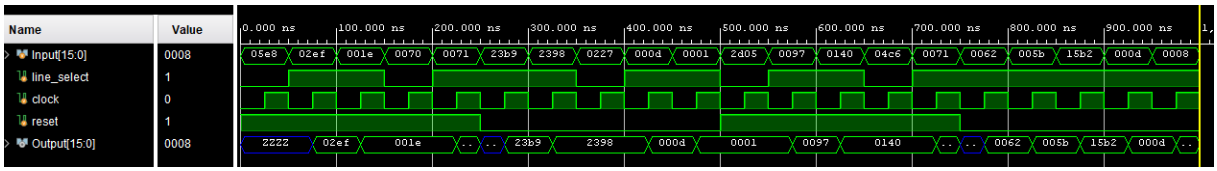



Figure 5: Simulation result of Part 1

### 2.1.3 Part 2

In this part we implemented a 16 line 16-bit register file module (32 byte) using 4:16 decoder module we implemented in **Part - 1** above and 16-bit register line module. This module takes 4-bit selA, 4-bit selB, 4-bit selWrite, 16-bit dataIn, reset, writeEnable, and clock as input and gives 16-bit dataA and 16-bit dataB. The operations we implemented of the register file module are below:

- At the falling edge of the reset, all lines of the register file will be cleared.
- selWrite input selects the line to be updated. The selected line should store the dataIn input at the rising edge of the clock when the writeEnable is high.
- Register outputs are exported via dataA and dataB. selA selects the line of the register file for dataA output. Similarly, selB selects the line of the registerfile for dataB output. These two operations do not require the clock signal.

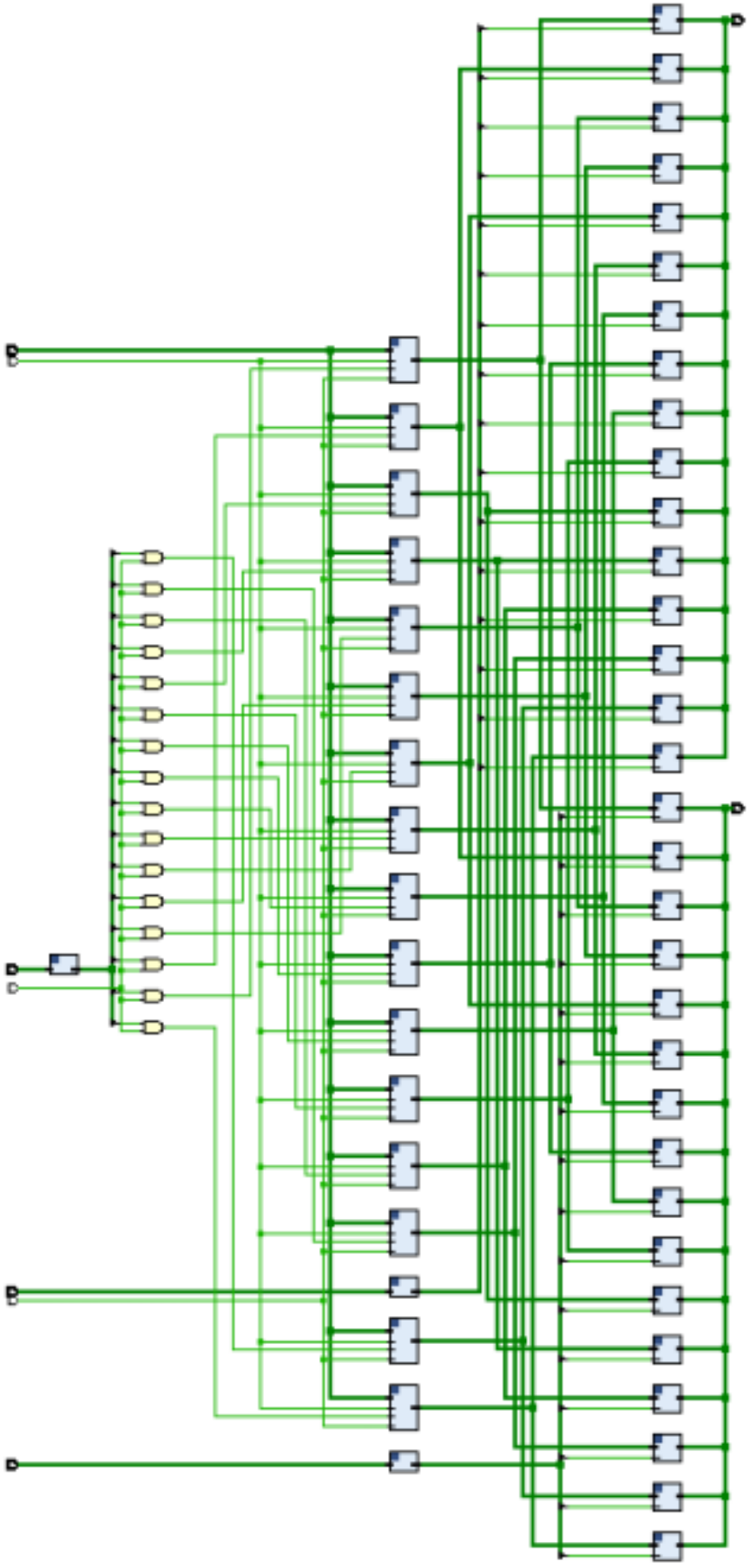


Figure 6: Circuit of Register File Design

```
/*         Part2         */
module buffer(
    input wire [15:0] Input,
    input wire enable,
    output wire [15:0] Output
    );
   assign Output = enable ? Input : 16'bZ;

endmodule

module  register_file_module(
    input wire [3:0] selA,
    input wire [3:0] selB,
    input wire [3:0] selWrite,
    input wire [15:0] dataIn,
    input wire reset,
    input wire writeEnable,
    input wire clock,
    output wire [15:0] dataA,
    output wire [15:0] dataB
);
    wire [15:0] writeLines, AselectLines, BselectLines;

    decoder_4input dec0(selWrite, writeLines);
    decoder_4input dec1(selA, AselectLines);
    decoder_4input dec2(selB, BselectLines);

    wire [15:0] out0, out1, out2, out3, out4, out5, out6, out7;
    wire [15:0] out8, out9, out10, out11, out12, out13, out14, out15;


decoder_4input dec0(selWrite, writeLines);
decoder_4input dec1(selA, AselectLines);
decoder_4input dec2(selB, BselectLines);

wire [15:0] out0, out1, out2, out3, out4, out5, out6, out7;
wire [15:0] out8, out9, out10, out11, out12, out13, out14, out15;
register_line_module reg0(dataIn, writeLines[0] && writeEnable, clock, reset, out0);
register_line_module reg1(dataIn, writeLines[1] && writeEnable, clock, reset, out1);
register_line_module reg2(dataIn, writeLines[2] && writeEnable, clock, reset, out2);
register_line_module reg3(dataIn, writeLines[3] && writeEnable, clock, reset, out3);
register_line_module reg4(dataIn, writeLines[4] && writeEnable, clock, reset, out4);
register_line_module reg5(dataIn, writeLines[5] && writeEnable, clock, reset, out5);
register_line_module reg6(dataIn, writeLines[6] && writeEnable, clock, reset, out6);
register_line_module reg7(dataIn, writeLines[7] && writeEnable, clock, reset, out7);
register_line_module reg8(dataIn, writeLines[8] && writeEnable, clock, reset, out8);
register_line_module reg9(dataIn, writeLines[9] && writeEnable, clock, reset, out9);
register_line_module regg10(dataIn, writeLines[10] && writeEnable, clock, reset, out10);
register_line_module regg11(dataIn, writeLines[11] && writeEnable, clock, reset, out11);
register_line_module regg12(dataIn, writeLines[12] && writeEnable, clock, reset, out12);
register_line_module regg13(dataIn, writeLines[13] && writeEnable, clock, reset, out13);
register_line_module regg14(dataIn, writeLines[14] && writeEnable, clock, reset, out14);
register_line_module regg15(dataIn, writeLines[15] && writeEnable, clock, reset, out15);
```

Figure 7: Code of Part 2 - continued

```
    buffer b0(out0, AselectLines[0], dataA);
    buffer b1(out1, AselectLines[1], dataA);
    buffer b2(out2, AselectLines[2], dataA);
    buffer b3(out3, AselectLines[3], dataA);
    buffer b4(out4, AselectLines[4], dataA);
    buffer b5(out5, AselectLines[5], dataA);
    buffer b6(out6, AselectLines[6], dataA);
    buffer b7(out7, AselectLines[7], dataA);
    buffer b8(out8, AselectLines[8], dataA);
    buffer b9(out9, AselectLines[9], dataA);
    buffer b10(out10, AselectLines[10], dataA);
    buffer b11(out11, AselectLines[11], dataA);
    buffer b12(out12, AselectLines[12], dataA);
    buffer b13(out13, AselectLines[13], dataA);
    buffer b14(out14, AselectLines[14], dataA);
    buffer b15(out15, AselectLines[15], dataA);

    buffer b16(out0, BselectLines[0], dataB);
    buffer b17(out1, BselectLines[1], dataB);
    buffer b18(out2, BselectLines[2], dataB);
    buffer b19(out3, BselectLines[3], dataB);
    buffer b20(out4, BselectLines[4], dataB);
    buffer b21(out5, BselectLines[5], dataB);
    buffer b22(out6, BselectLines[6], dataB);
    buffer b23(out7, BselectLines[7], dataB);
    buffer b24(out8, BselectLines[8], dataB);
    buffer b25(out9, BselectLines[9], dataB);
    buffer b26(out10, BselectLines[10], dataB);
    buffer b27(out11, BselectLines[11], dataB);
    buffer b28(out12, BselectLines[12], dataB);
    buffer b29(out13, BselectLines[13], dataB);
    buffer b30(out14, BselectLines[14], dataB);
    buffer b31(out15, BselectLines[15], dataB);

endmodule
```

Figure 8: Code of Part 2 - continued

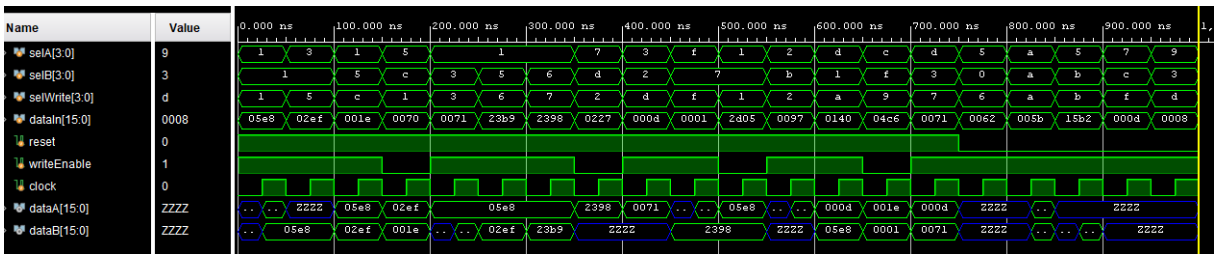



Figure 9: Simulation result of Part 2

### 2.1.4 Part 3

In this part we designed Arithmetic Logic Units (ALU) that does some operations in the Table 1.

Table 1: ALU Instructions

| OPCODE | Operation Name | Operation | Update Flag |
|---|---|---|---|
| 0 | AND | dst = srcA & srcB | No |
| 1 | OR | dst = srcA \| srcB | No |
| 2 | Addition | dst = srcA + srcB | Yes |
| 3 | Subtraction | dst = srcA - srcB | Yes |
| 4 | XOR | dst = srcA ^ srcB | No |
| 5 | Logical Shift Right | dst = srcA >> srcB | No |
| 6 | Logical Shift Left | dst = srcA << srcB | No |
| 7 | LOAD | dst = srcB | No |

- In addition to these operations, the ALU produces zero flag. Meaning that, if the result of the operation is zero, a special flip-flop (called zeroFlag) will be set to logical high.
- The inputs of the ALU will be called srcA, srcB. The output will be called dst.
- The ALU will decide its operations with the help of 3-bit Op input.
- Also, the module has clock and reset inputs and zeroFlag output.
  - At the falling edge of the reset zeroFlag will be cleared.
  - At the rising edge of the clock zeroFlag will be updated according to output of the ALU when the operation updates the flags.

For example, AND operation does not update the flag.

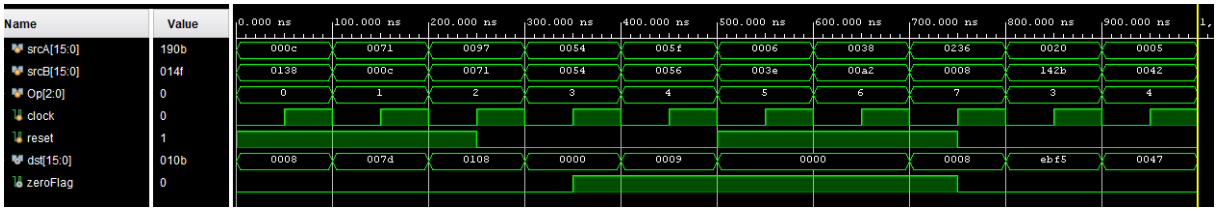



Figure 10: ALU module simulation results

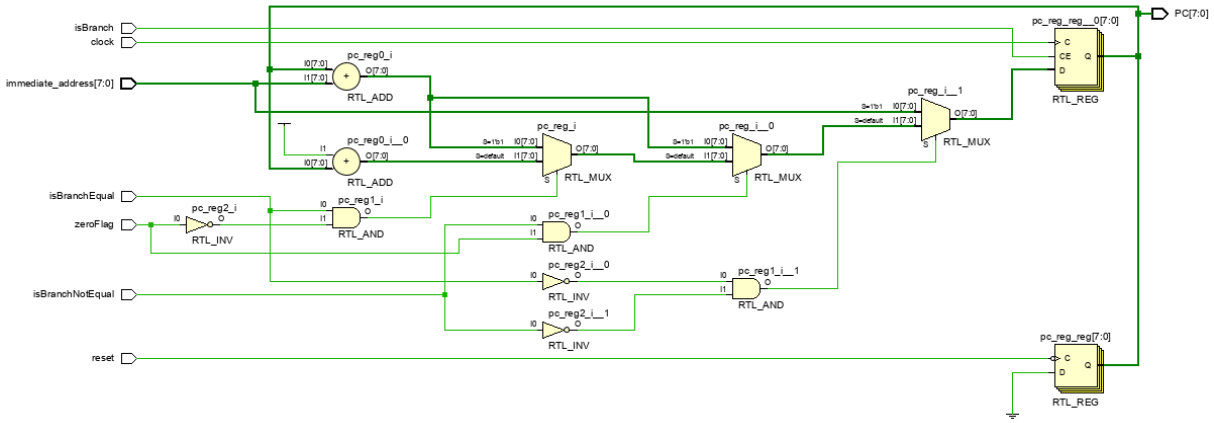



Figure 11: ALU circuit

```
/*      Part3      */
module ALU(
    input wire [15:0] srcA,
    input wire [15:0] srcB,
    output wire [15:0] dst,
    input wire [2:0] Op,
    input wire clock,
    input wire reset,
    output wire zeroFlag
    );
    reg [15:0] ALU_Result;
    reg zero, update;

    initial begin zero = 0; end
    always @(*)
    begin
      update = 0;
      case(Op)
        3'b000: // AND
           ALU_Result = srcA & srcB;
        3'b001: // OR
           ALU_Result = srcA | srcB;
        3'b010: // ADD
           ALU_Result = srcA + srcB ;
        3'b011: // SUB
           ALU_Result = srcA - srcB ;
        3'b100: // XOR
           ALU_Result = srcA^srcB;
        3'b101: // LSR
           ALU_Result = srcA >> srcB;
        3'b110: // LSL
           ALU_Result = srcA<<srcB;
        3'b111: // LD
           ALU_Result = srcB;
```

(a) ALU module codes

```
        3'b110: // LSL
           ALU_Result = srcA<<srcB;
        3'b111: // LD
           ALU_Result = srcB;

        endcase
        case(Op)
        3'b010: // ADD
           update = 1;
        3'b011: // SUB
           update = 1;
        endcase
    end

    always @(posedge clock)
    begin
      if(ALU_Result == 16'd0 && update)
        zero = 1;
    end
    always @(negedge reset)
    begin
        zero = 0;
    end

    assign zeroFlag = zero;
    assign dst = ALU_Result;

endmodule
```

(b) ALU module codes – continued

### 2.1.5 Part 4

In this part, we created the control signals of our digital system according to instruction input. Table 2 shows the instruction set of the digital system.

Table 2: Digital System Instruction Set

| OPCODE | Operation Name | Operation | Type | Update Flag |
|---|---|---|---|---|
| 0 | AND | dst = srcA & srcB | Register | No |
| 1 | OR | dst = srcA \| srcB | Register | No |
| 2 | Addition | dst = srcA + srcB | Register | Yes |
| 3 | Subtraction | dst = srcA - srcB | Register | Yes |
| 4 | XOR | dst = srcA ^ srcB | Register | No |
| 5 | Logical Shift Right | dst = srcA >>srcB | Register | No |
| 6 | Logical Shift Left | dst = srcA <<srcB | Register | No |
| 7 | LOAD | dst = Immediate | Load | No |
| 8 | ANDI | dst = srcA & Immediate | Immediate | No |
| 9 | ORI | dst = srcA \| Immediate | Immediate | No |
| 10 | ADD_Immediate | dst = srcA + Immediate | Immediate | Yes |
| 11 | Compare | Compare srcA, srcB | Register | Yes |
| 12 | NOOP | - | - | No |
| 13 | B | PC = Immediate | Branch | No |
| 14 | BNE | PC = PC + Immediate if not equal | Branch | No |
| 15 | BEQ | PC = PC + Immediate if equal | Branch | No |

According to instruction set there are 4 type of instruction in this system. Table 3, 4, 5, and 6 shows the instructions bits separation for register, immediate, load, and branch respectively.

Table 3: Register Type Instructions

| Opcode | dst | srcA | srcB |
|---|---|---|---|
| 4-bit | 4-bit | 4-bit | 4-bit |

Table 4: Immediate Type Instructions

| Opcode | dst | srcA | Immediate |
|---|---|---|---|
| 4-bit | 4-bit | 4-bit | 4-bit |

Inputs and outputs ports information are given below.

- **instruction (15-bit Input):** comes from the ROM and takes the current instruction as input.
- **opcode (4-bit Output):** contains the operation information of the instruction.
- **selWrite (4-bit Output):** contains the destination register address for writing operation.

Table 5: Load Type Instructions

| Opcode | dst | Immediate |
|---|---|---|
| 4-bit | 4-bit | 8-bit |

Table 6: Branch Type Instructions

| Opcode | Unused | Immediate |
|---|---|---|
| 4-bit | 4-bit | 8-bit |

- **selA (4-bit Output):** contains the first operand address information of the instruction in the registerFile.
- **selB (4-bit Output):** contains the second operand address information of the instruction in the registerFile.
- **fourBitImmediate (16-bit Output):** contains the 4-bit immediate value information for immediate instructions. Zero extension will be used to extend 4-bit to 16-bit.
- **eightBitImmediate (16-bit Output):** contains the 8-bit immediate value information for load and branch instructions. Zero extension will be used to extend 4-bit to 16-bit.
- **writeEnable (1-bit Output):** If the instruction needs to write the results to register file, it must be 1. Otherwise, it will be 0.
- **isLoad (1-bit Output):** The flag will 1 if the instruction is the load instruction.
- **isImmediate (1-bit Output):** The flag will 1 if the instruction is the immediate instruction.
- **isBranch (1-bit Output):** The flag will 1 if the instruction is the branch instruction (Opcode 13).

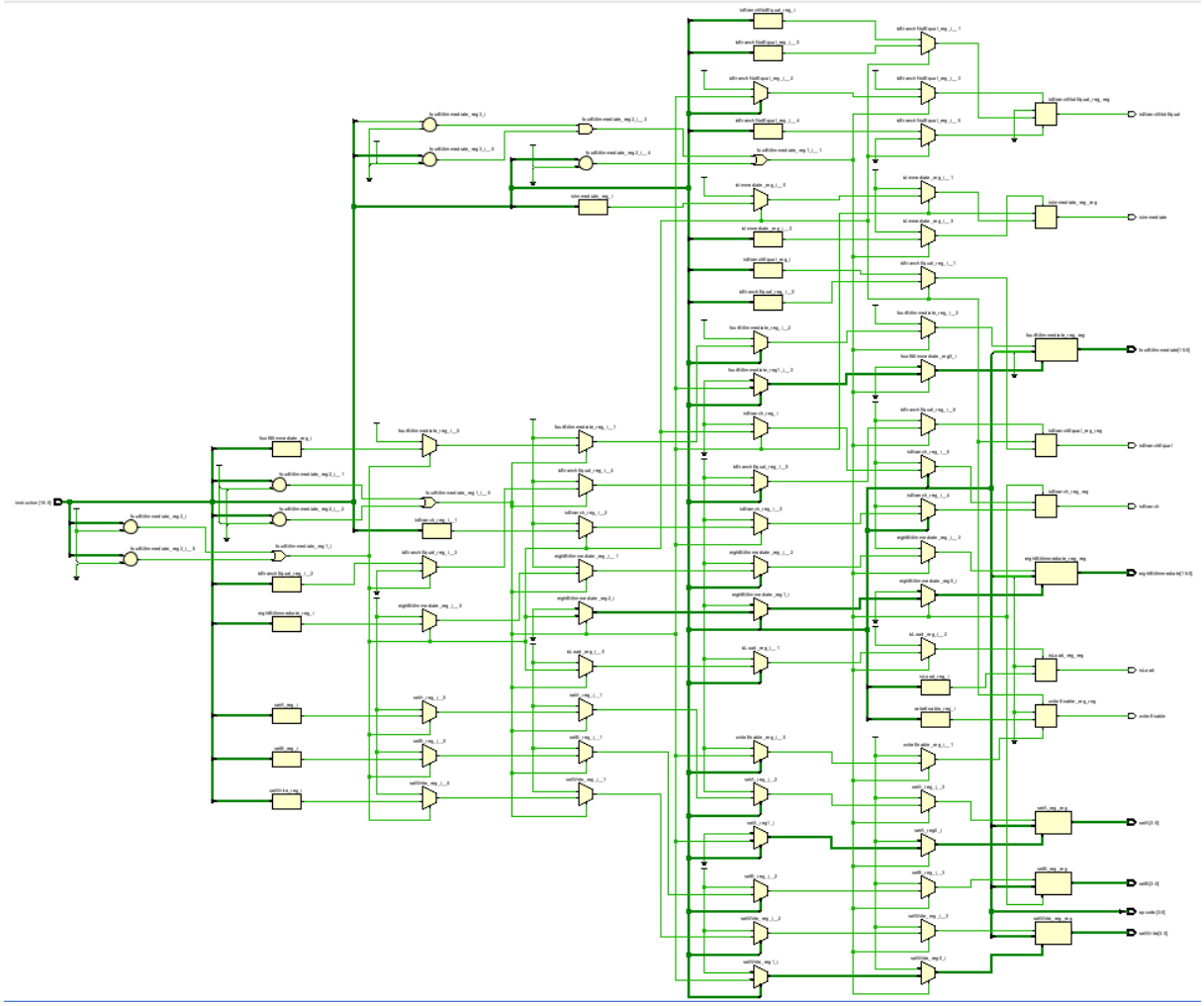


Figure 13: Instruction Decoder circuit

```
/*       Part4       */
module instruction_decoder(
    input wire [15:0] instruction,
    output wire [3:0] opcode,
    output wire [3:0] selWrite,
    output wire [3:0] selA,
    output wire [3:0] selB,
    output wire [15:0] fourBitImmediate,
    output wire [15:0] eightBitImmediate,
    output wire writeEnable,
    output wire isLoad,
    output wire isImmediate,
    output wire isBranch,
    output wire isBranchNotEqual,
    output wire isBranchEqual
    );
    reg [3:0] opcode_reg, selA_reg, selB_reg, selWrite_reg;
    reg [15:0] fourBitImmediate_reg, eightBitImmediate_reg;
    reg writeEnable_reg, isLoad_reg, isImmediate_reg;
    reg isBranch_reg, isBranchNotEqual_reg, isBranchEqual_reg;


    assign opcode = opcode_reg;
    assign selWrite = selWrite_reg;
    assign selA = selA_reg;
    assign selB = selB_reg;
    assign fourBitImmediate = fourBitImmediate_reg;
    assign eightBitImmediate = eightBitImmediate_reg;
    assign writeEnable = writeEnable_reg;
    assign isLoad = isLoad_reg;
    assign isImmediate = isImmediate_reg;
    assign isBranch = isBranch_reg;
    assign isBranchNotEqual = isBranchNotEqual_reg;
    assign isBranchEqual = isBranchEqual_reg;


    always @(*) begin
        opcode_reg = instruction[15:12];
```

Figure 14: Instruction Decoder

```
    always @(*) begin
        opcode_reg = instruction[15:12];

        // register
        if ((opcode_reg[3] == 0 & opcode_reg != 4'b0111) || opcode_reg == 4'b1011) begin
            fourBitImmediate_reg = 16'bZ;
            eightBitImmediate_reg = 16'bZ;
            isLoad_reg = 0;
            isImmediate_reg = 0;
            isBranch_reg = 0;
            isBranchNotEqual_reg = 0;
            isBranchEqual_reg = 0;

            selB_reg = instruction[3:0];
            selA_reg = instruction[7:4];
            selWrite_reg = instruction[11:8];
            writeEnable_reg = 1;
        end

        //load
        else if (opcode_reg == 4'b0111) begin
            selA_reg = 4'dZ;
            selB_reg = 4'dZ;
            fourBitImmediate_reg = 16'bZ;
            isLoad_reg = 0;
            isImmediate_reg = 0;
            isBranchNotEqual_reg = 0;
            isBranchEqual_reg = 0;

            isBranch_reg = 1;
            eightBitImmediate_reg = 16'd0;
            eightBitImmediate_reg[7:0] = instruction[7:0];
            selWrite_reg = instruction[11:8];
            writeEnable_reg = 1;

        end

        //Immediate
        else if (opcode_reg[3:1] == 3'b100 || opcode_reg == 4'b1010) begin
            selB_reg = 4'dZ;
            eightBitImmediate_reg = 16'bZ;
```

Figure 15: Instruction Decoder – continued

```
//Immediate
else if (opcode_reg[3:1] == 3'b100 || opcode_reg == 4'b1010) begin
    selB_reg = 4'dZ;
    eightBitImmediate_reg = 16'bZ;
    isLoad_reg = 0;
    isBranch_reg = 0;
    isBranchNotEqual_reg = 0;
    isBranchEqual_reg = 0;

    isImmediate_reg = 1;
    fourBitImmediate_reg = 16'd0;
    fourBitImmediate_reg[3:0] = instruction[3:0];
    selA_reg = instruction[7:4];
    selWrite_reg = instruction[11:8];
    writeEnable_reg = 1;

end

//Branch
else if (opcode_reg[3:1] == 3'b111 || opcode_reg == 4'b1101) begin
    selA_reg = 4'dZ;
    selB_reg = 4'dZ;
    selWrite_reg = 4'dZ;
    fourBitImmediate_reg = 16'bZ;
    writeEnable_reg = 0;
    isLoad_reg = 0;
    isImmediate_reg = 0;

    isBranch_reg = 1;
    eightBitImmediate_reg = 16'd0;
    eightBitImmediate_reg[7:0] = instruction[7:0];
    if (opcode_reg == 4'd14) begin
        isBranchNotEqual_reg = 1;
        isBranchEqual_reg = 0;
    end
    else if (opcode_reg == 4'd15) begin
        isBranchEqual_reg = 1;
        isBranchNotEqual_reg = 0;
    end
end

//NOOP
else if (opcode_reg == 4'd12) begin
    selA_reg = 4'dZ;
```

Figure 16: Instruction Decoder – continued

```
                isBranchNotEqual_reg = 1;
                isBranchEqual_reg = 0;
            end
            else if (opcode_reg == 4'd15) begin
                isBranchEqual_reg = 1;
                isBranchNotEqual_reg = 0;
            end
        end

        //NOOP
        else if (opcode_reg == 4'd12) begin
            selA_reg = 4'dZ;
            selB_reg = 4'dZ;
            selWrite_reg = 4'dZ;
            fourBitImmediate_reg = 16'bZ;
            writeEnable_reg = 0;
            isLoad_reg = 0;
            isImmediate_reg = 0;

            isBranch_reg = 0;
            eightBitImmediate_reg = 16'dZ;
            isBranchNotEqual_reg = 0;
            isBranchEqual_reg = 0;
        end
    end
endmodule
```

Figure 17: Instruction Decoder – continued

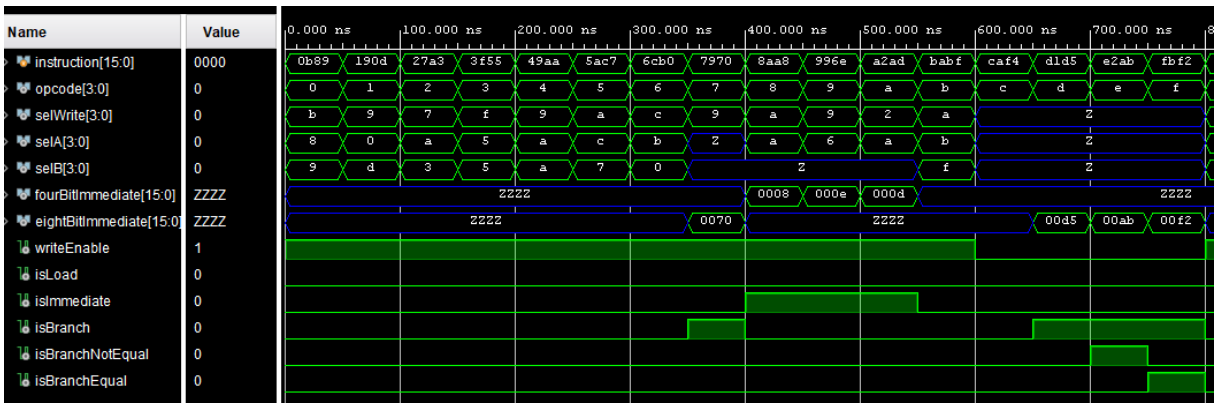



Figure 18: Instruction Decoder

### 2.1.6 Part 5

In this part, program counter module develops the program counter module for the digital design. The output of the program counter module (PC) stores the current program address of the system.

- The inputs of the module are reset, clock, isBranch, isBranchNotEqual, isBranchEqual, and immediateAddress (8-bit). The instruction fetched from the program memory using this information.
- PC will be cleared at the falling edge of the reset signal. If the operation is branch (Opcode 13), PC value will be the immediateAddress (8-bit input) at the rising edge of the clock.
- PC value will be the immediateAddress (8-bit input) at the rising edge of the clock.
- PC value will be current PC value + immediateAddress at the rising edge of the clock if the operation is branch not equal and flag shows the result is not equal, or the operation is branch equal and flag shows the result is equal.
- You can access the result information using zeroFlag.
- Otherwise, PC value will be next address of the memory at the rising edge of the clock.

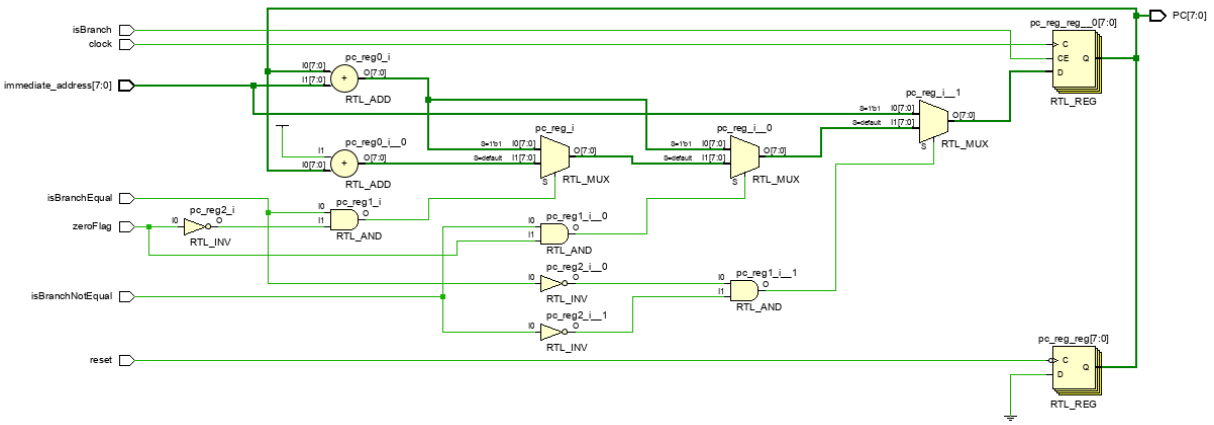



Figure 19: Program counter circuit

```
/*   Part5    */
module program_counter(
    input wire reset,
    input wire clock,
    input wire zeroFlag,
    input wire isBranch,
    input wire isBranchNotEqual,
    input wire isBranchEqual,
    input wire [7:0] immediate_address,
    output wire [7:0] PC
);
    reg [7:0] pc_reg;

    assign PC = pc_reg;

    always @(posedge clock) begin
        if (isBranch) begin
            if (!isBranchEqual && !isBranchNotEqual) begin
                pc_reg = immediate_address;
            end

            else if (isBranchNotEqual && zeroFlag) begin
                pc_reg = pc_reg + immediate_address;
            end

            else if (isBranchEqual && !zeroFlag) begin
                pc_reg = pc_reg + immediate_address;
            end
            else begin
                pc_reg = pc_reg +8'd1;
            end
       end
    end
    always @(negedge reset) begin
        pc_reg = 8'd0;
    end
endmodule
```

Figure 20: Part 5 module code of Program Counter

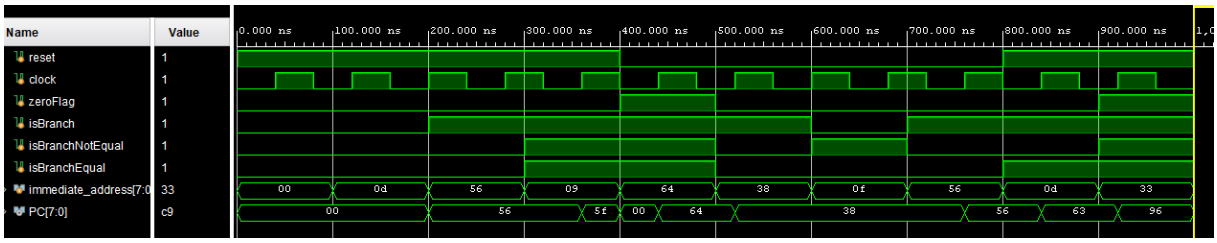



Figure 21: Part 5 module simulation results of Program Counter

### 2.1.7 Part 6

In this part, we implemented the mini computer using the modules we implemented in the previous parts. The module only get clock and reset signals. Clock and reset signal are connected to other modules which needs this signals. We give instruction, dataA, dataB, dst, PC, input value of ALU srcB for debugging and test of the system. We also used the ROM module, developed for us. We directly added this module to our design files. We also included the memory data to our project, the guideline have been added to homework files for this operation. The input connection information of the modules are given below.

- Program Memory
  - PC ← PC output of the Program Counter
- Instruction Decoder
  - instruction ← instruction output of the Program Memory
- Register File
  - selA ← selA output of the Instruction Decoder
  - selB← selB output of the Instruction Decoder
  - selWrite ← selWrite output of the Instruction Decoder
  - dataIn ← dst output of the ALU
  - writeEnable ← writeEnable output of the Instruction Decoder
- ALU
  - srcA ← dataA output of the Register File
  - srcB ← It can be dataB, 4-bit immediate or 8-bit immediate according to instruction type. You must select it for each type of instruction.
  - Op ← Opcode output of the Instruction Decoder

- Program Counter
  - zeroFlag v zeroFlag output of the ALU
  - isBranch ← isBranch output of the Instruction Decoder
  - isBranchNotEqual ← isBranchNotEqual output of the Instruction Decoder
  - isBranchEqual ← isBranchEqual output of the Instruction Decoder
  - immediateAddress ← One of the immediate value output of the Instruction Decoder.

```
module minicomputer(
    input wire clock,
    input wire reset,
    output wire [15:0] instruction,
    output wire [15:0] dataA,
    output wire [15:0] dataB,
    output wire [15:0] dst,
    output wire [7:0] pc,
    output wire [15:0] srcB
);

    wire [3:0] opcode, selWrite, selA, selB;
    wire [15:0] fourBitImmediate, eightBitImmediate;
    wire writeEnable, isLoad, isImmediate, isBranch, isBranchNotEqual, isBranchEqual, zeroFlag;

    ProgramMemory f0(pc, instruction);

    instruction_decoder f1(instruction, opcode, selWrite, selA, selB, fourBitImmediate,
        eightBitImmediate, writeEnable, isLoad, isImmediate, isBranch, isBranchNotEqual, isBranchEqual);

    register_file_module f2(selA, selB, selWrite, dst, reset, writeEnable, clock, dataA, dataB);

    assign srcB = (isLoad | isBranch) ? eightBitImmediate: (isImmediate)? fourBitImmediate:((opcode < 4'd7) | (opcode == 4'd11))? dataB:4'dZ;

    ALU f3(dataA, srcB, dst, opcode[2:0], clock, reset, zeroFlag);

    program_counter f4(reset, clock, zeroFlag, isBranch, isBranchNotEqual, isBranchEqual, eightBitImmediate[7:0], pc);
    program_counter f5(reset, clock, zeroFlag, isBranch, isBranchNotEqual, isBranchEqual, fourBitImmediate[7:0], pc);

endmodule
```

Figure 22: Mini Computer module

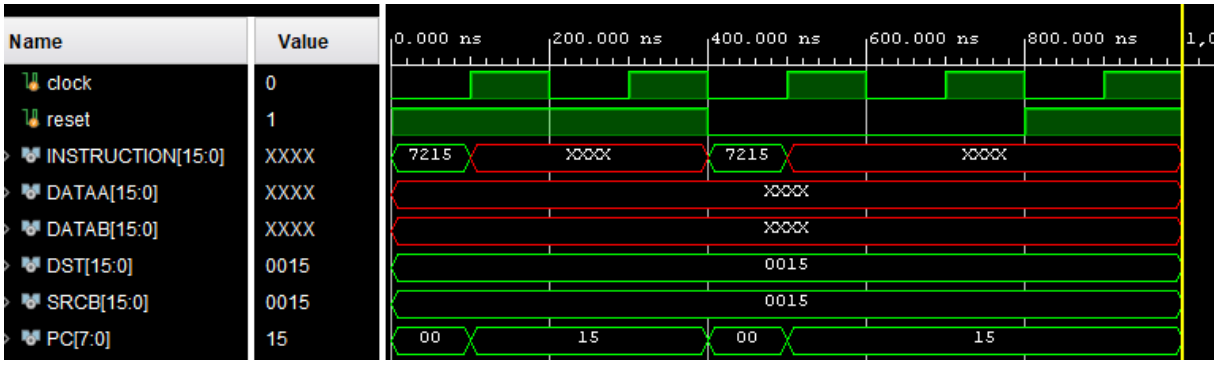



Figure 23: Mini Computer module simulation results

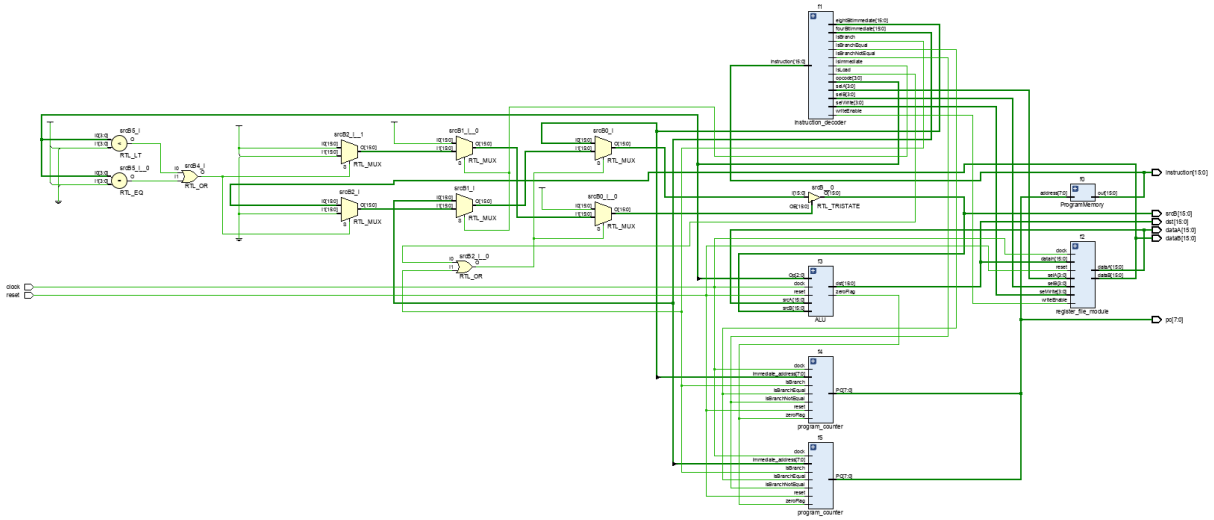


Figure 24: Mini Computer module circuit design

# 3 RESULTS [15 points]

You can see the simulation codes we used for Experiment parts and their simulation results here. To carry out each different implementation of the experiment, we arranged inputs of each simulation according to part it belongs. The simulation results we obtained after completing each step of the experiments resulted IN THE WAY we expected. This concludes the fact that our experiment is coherent for each individual part, except for the part-6.

About part-1, see **Part 1** section above. In this part, we implemented

- 4:16 decoder module with enable input. If enable signal is logical low, all the outputs of decoder will be logical low.
- 16-bit register line module. This module takes lineselect, clock, reset, and 16-bit dataIn as input and gives 16-bit stored data as output. For simulation codes and results, see figures below:

```
/* PART1  -- Simulation of Register Line */
reg [15:0] Input;
reg line_select, clock, reset;
wire [15:0] Output;

register_line_module uut( Input, line_select, clock, reset, Output);

initial begin
    clock = 1; reset = 0;
    Input = 16'd1512;   line_select = 0;   #50;
    Input = 16'd751;    line_select = 1;   #50;
    Input = 16'd30;     line_select = 1;   #50;
    Input = 16'd112;    line_select = 0;   #50;
    Input = 16'd113;    line_select = 1;   #50;
    Input = 16'd9145;   line_select = 1;   #50;
    Input = 16'd9112;   line_select = 1;   #50;
    Input = 16'd551;    line_select = 0;   #50;
    Input = 16'd13;     line_select = 1;   #50;
    Input = 16'd1;      line_select = 1;   #50;
    Input = 16'd11525;  line_select = 0;   #50;
    Input = 16'd151;    line_select = 1;   #50;
    Input = 16'd320;    line_select = 1;   #50;
    Input = 16'd1222;   line_select = 0;   #50;
    Input = 16'd113;    line_select = 1;   #50;
    Input = 16'd98;     line_select = 1;   #50;
    Input = 16'd91;     line_select = 1;   #50;
    Input = 16'd5554;   line_select = 1;   #50;
    Input = 16'd13;     line_select = 1;   #50;
    Input = 16'd8;      line_select = 1;   #50;
end
always begin
    clock = ~clock; #25;
end
always begin
    reset = ~reset; #250;
end
```

Figure 25: Simulation code of Part 1

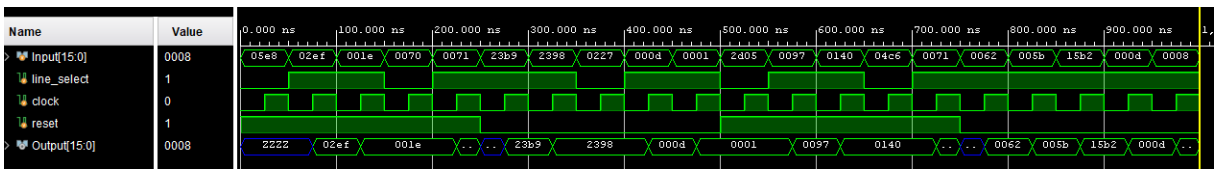



Figure 26: Simulation result of the circuit given in Part 1

About part-2, see **Part 2** section above. In this part, we implemented a 16 line 16-bit register file module (32 byte) using 4:16 decoder module and 16-bit register line module. For simulation codes and results, see figures below:

```
/* PART2 -- Simulation of Register File Module  */

reg [3:0] selA, selB, selWrite;
reg [15:0] dataIn;
reg reset, writeEnable, clock;
wire [15:0] dataA;
wire [15:0] dataB;

register_file_module uut(selA, selB, selWrite, dataIn, reset, writeEnable, clock, dataA, dataB);
initial begin
    clock = 1; reset = 0;
    dataIn = 16'd1512;    selA = 4'd1;   selB = 4'd1;   selWrite = 4'd1;   writeEnable = 1;   #50;
    dataIn = 16'd751;     selA = 4'd3;   selB = 4'd1;   selWrite = 4'd5;   writeEnable = 1;   #50;
    dataIn = 16'd30;      selA = 4'd1;   selB = 4'd5;   selWrite = 4'd12;  writeEnable = 1;   #50;
    dataIn = 16'd112;     selA = 4'd5;   selB = 4'd12;  selWrite = 4'd1;   writeEnable = 0;   #50;
    dataIn = 16'd113;     selA = 4'd1;   selB = 4'd3;   selWrite = 4'd3;   writeEnable = 1;   #50;
    dataIn = 16'd9145;    selA = 4'd1;   selB = 4'd5;   selWrite = 4'd6;   writeEnable = 1;   #50;
    dataIn = 16'd9112;    selA = 4'd1;   selB = 4'd6;   selWrite = 4'd7;   writeEnable = 1;   #50;
    dataIn = 16'd551;     selA = 4'd7;   selB = 4'd13;  selWrite = 4'd2;   writeEnable = 0;   #50;
    dataIn = 16'd13;      selA = 4'd3;   selB = 4'd2;   selWrite = 4'd13;  writeEnable = 1;   #50;
    dataIn = 16'd1;       selA = 4'd15;  selB = 4'd7;   selWrite = 4'd15;  writeEnable = 1;   #50;
    dataIn = 16'd11525;   selA = 4'd1;   selB = 4'd7;   selWrite = 4'd1;   writeEnable = 0;   #50;
    dataIn = 16'd151;     selA = 4'd2;   selB = 4'd11;  selWrite = 4'd2;   writeEnable = 1;   #50;
    dataIn = 16'd320;     selA = 4'd13;  selB = 4'd1;   selWrite = 4'd10;  writeEnable = 1;   #50;
    dataIn = 16'd1222;    selA = 4'd12;  selB = 4'd15;  selWrite = 4'd9;   writeEnable = 0;   #50;
    dataIn = 16'd113;     selA = 4'd13;  selB = 4'd3;   selWrite = 4'd7;   writeEnable = 1;   #50;
    dataIn = 16'd98;      selA = 4'd5;   selB = 4'd0;   selWrite = 4'd6;   writeEnable = 1;   #50;
    dataIn = 16'd91;      selA = 4'd10;  selB = 4'd10;  selWrite = 4'd10;  writeEnable = 1;   #50;
    dataIn = 16'd5554;    selA = 4'd5;   selB = 4'd11;  selWrite = 4'd11;  writeEnable = 1;   #50;
    dataIn = 16'd13;      selA = 4'd7;   selB = 4'd12;  selWrite = 4'd15;  writeEnable = 1;   #50;
    dataIn = 16'd8;       selA = 4'd9;   selB = 4'd3;   selWrite = 4'd13;  writeEnable = 1;   #50;
end
always begin
    clock = ~clock; #25;
end
always begin
    reset = ~reset; #750;
end
```

Figure 27: Simulation code of Part 2

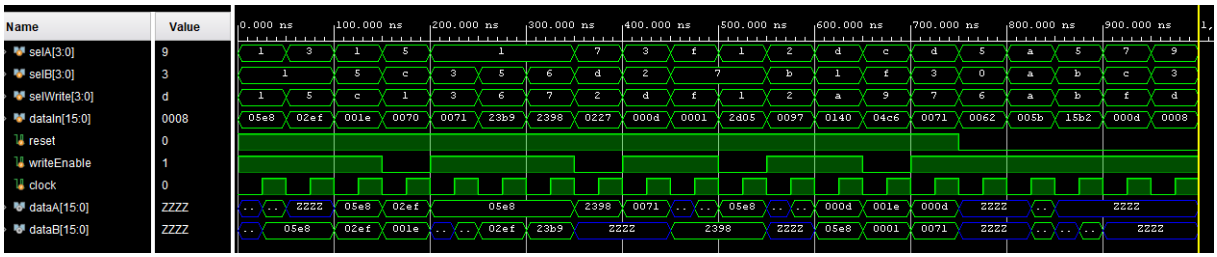



Figure 28: Simulation result of Part 2

About part-3, see **Part 3** section above. In this part, we implemented an Arithmetic Logic Units (ALU) that does some operations in the given table. In addition to these operations, we have other operations, see **Part 3** section above. For simulation codes and results, see figures below:

```
/* PART3 -- Simulation of ALU */
reg [15:0] srcA, srcB;
reg [2:0] Op;
reg clock, reset;
wire [15:0] dst;
wire zeroFlag;

ALU uut(srcA, srcB, dst, Op, clock, reset, zeroFlag);

initial begin
    clock = 1; reset = 0;
    srcA = 16'd12;    srcB = 16'd312;   Op = 3'd0;   #100;
    srcA = 16'd113;   srcB = 16'd12;    Op = 3'd1;   #100;
    srcA = 16'd151;   srcB = 16'd113;   Op = 3'd2;   #100;
    srcA = 16'd84;    srcB = 16'd84;    Op = 3'd3;   #100;
    srcA = 16'd95;    srcB = 16'd86;    Op = 3'd4;   #100;
    srcA = 16'd6;     srcB = 16'd62;    Op = 3'd5;   #100;
    srcA = 16'd56;    srcB = 16'd162;   Op = 3'd6;   #100;
    srcA = 16'd566;   srcB = 16'd8;     Op = 3'd7;   #100;
    srcA = 16'd32;    srcB = 16'd5163;  Op = 3'd3;   #100;
    srcA = 16'd5;     srcB = 16'd66;    Op = 3'd4;   #100;
    srcA = 16'd6411;  srcB = 16'd335;   Op = 3'd0;   #100;
end
always begin
    clock = ~clock;  #50;
end
always begin
    reset = ~reset;  #250;
end
```

Figure 29: Simulation code Part 3

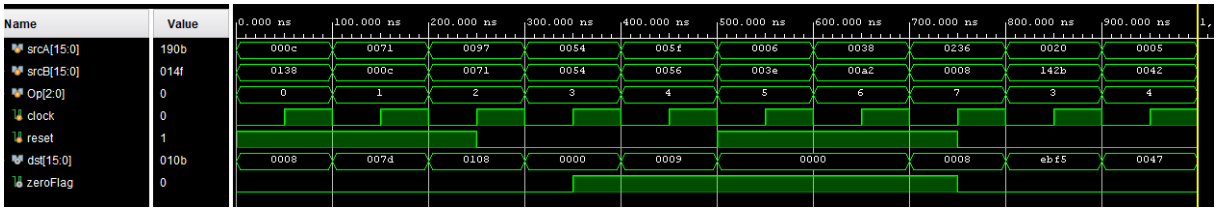



Figure 30: Simulation result of Part 3

About part-4, see **Part 4** section above. In this part, we implemented an 8 byte memory module using 8-bit memory line module. 8 byte memory module should take 8-bit data as input and give 8-bit data as output. Also, the module should take 3-bit address, chip select, reset, read enable, write enable, and clock inputs for required operations. For simulation codes and results, see figures below:

```
/* PART4 -- Simulation of Instruction Decoder */
reg [15:0] instruction;
wire [3:0] opcode, selWrite, selA, selB;
wire [15:0] fourBitImmediate, eightBitImmediate;
wire writeEnable, isLoad, isImmediate, isBranch;
wire isBranchNotEqual,isBranchEqual;

instruction_decoder uut(instruction, opcode, selWrite,
    selA, selB,
    fourBitImmediate, eightBitImmediate,
    writeEnable, isLoad, isImmediate, isBranch,
    isBranchNotEqual, isBranchEqual);

initial begin
    instruction = 16'b0000101110001001; #50;
    instruction = 16'b0001100100001101; #50;
    instruction = 16'b0010011110100011; #50;
    instruction = 16'b0011111101010101; #50;
    instruction = 16'b0100100110101010; #50;
    instruction = 16'b0101101011000111; #50;
    instruction = 16'b0110110010110000; #50;
    instruction = 16'b0111100101110000; #50;

    instruction = 16'b1000101010101000; #50;
    instruction = 16'b1001100101101110; #50;
    instruction = 16'b1010001010101101; #50;
    instruction = 16'b1011101010111111; #50;
    instruction = 16'b1100101011110100; #50;
    instruction = 16'b1101000111010101; #50;
    instruction = 16'b1110001010101011; #50;
    instruction = 16'b1111101111110010; #50;
    instruction = 16'b0000000000000000; #50;
end
```

Figure 31: Simulation code Part 4

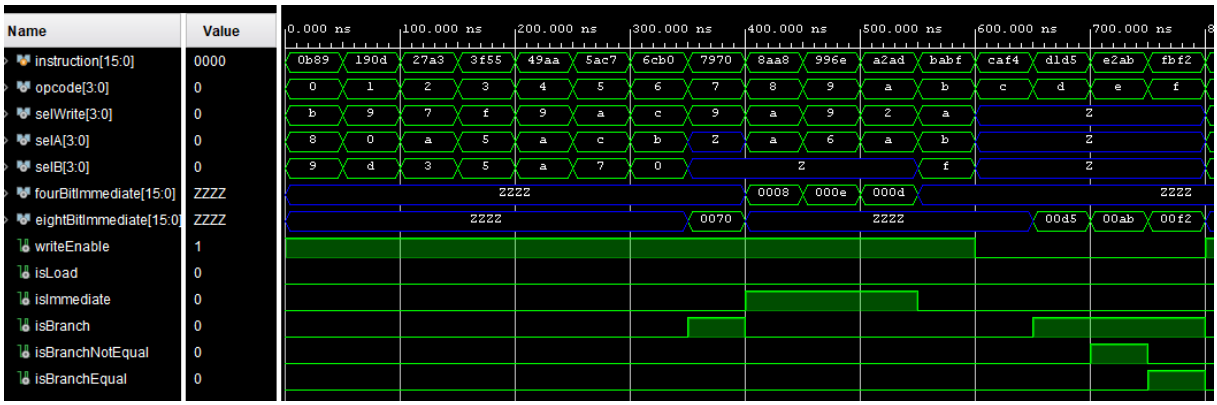



Figure 32: Simulation result of Part 4

About part-5, see **Part 5** section above. In this part, we implemented an instruction decoder: see **Part 5** section above. For simulation codes and results, see figures below:

```
/* PART5 -- Simulation of Program Counter  */
reg reset, clock;
reg zeroFlag, isBranch, isBranchNotEqual, isBranchEqual;
reg [7:0] immediate_address;
wire [7:0] PC;

program_counter uut(reset, clock, zeroFlag, isBranch, isBranchNotEqual, isBranchEqual, immediate_address, PC);

initial begin
    clock = 1; reset = 0;
    zeroFlag = 0; isBranch = 0; isBranchNotEqual = 0; isBranchEqual = 0; immediate_address = 8'd0;   #100;
    zeroFlag = 0; isBranch = 0; isBranchNotEqual = 0; isBranchEqual = 0; immediate_address = 8'd13;  #100;
    zeroFlag = 0; isBranch = 1; isBranchNotEqual = 0; isBranchEqual = 0; immediate_address = 8'd86;  #100;
    zeroFlag = 0; isBranch = 1; isBranchNotEqual = 1; isBranchEqual = 1; immediate_address = 8'd9;   #100;
    zeroFlag = 1; isBranch = 1; isBranchNotEqual = 1; isBranchEqual = 1; immediate_address = 8'd100; #100;
    zeroFlag = 0; isBranch = 1; isBranchNotEqual = 0; isBranchEqual = 0; immediate_address = 8'd56;  #100;
    zeroFlag = 0; isBranch = 0; isBranchNotEqual = 1; isBranchEqual = 0; immediate_address = 8'd15;  #100;
    zeroFlag = 0; isBranch = 1; isBranchNotEqual = 0; isBranchEqual = 0; immediate_address = 8'd86;  #100;
    zeroFlag = 0; isBranch = 1; isBranchNotEqual = 0; isBranchEqual = 1; immediate_address = 8'd13;  #100;
    zeroFlag = 1; isBranch = 1; isBranchNotEqual = 1; isBranchEqual = 1; immediate_address = 8'd51;  #100;

end
always begin
clock = ~clock; #40;
end
always begin
reset = ~reset; #400;
end
```

Figure 33: Simulation code of Part 5

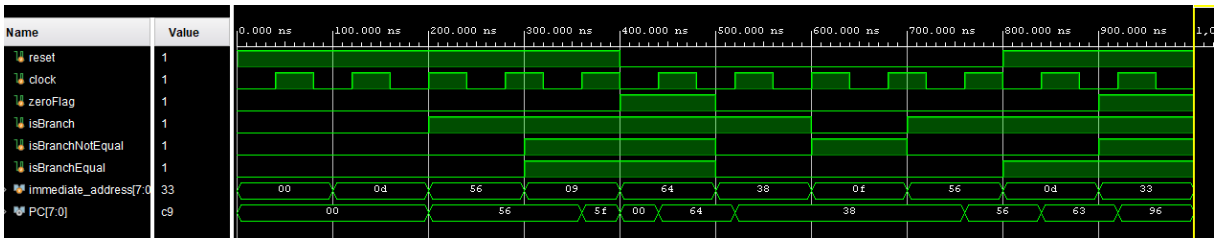



Figure 34: Simulation result of Part 5

About part-6, see **Part 6** section above. In this part, we tried to implement a mini computer using the modules we implemented in the previous parts. For simulation codes and results, see figures below:

```
/*    Part6    */

reg clock, reset;
wire [15:0] INSTRUCTION, DATAA, DATAB, DST, SRCB;
wire [7:0] PC;

minicomputer uut( clock, reset, INSTRUCTION, DATAA, DATAB, DST, PC, SRCB);

initial begin
    clock = 1; reset = 0;
end
always begin
    clock = ~clock; #100;
end
always begin
    reset = ~reset; #400;
end
```

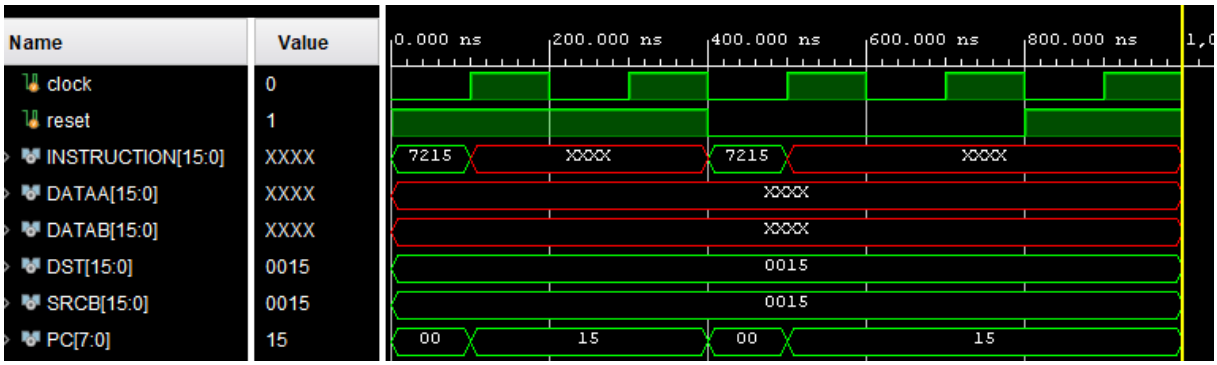



Figure 35: Simulation code and result of Part 6

P.S.: We failed.

# 4 DISCUSSION [25 points]

**First**, we implemented 4:16 decoder with enable input and 16-bit register line module. Register line module stores 16-bit data and according to our input values it can change its value. It gives 16-bit data, which it is storing, as output. Also, we use clock signal for updating data and reset signal for clearing data.

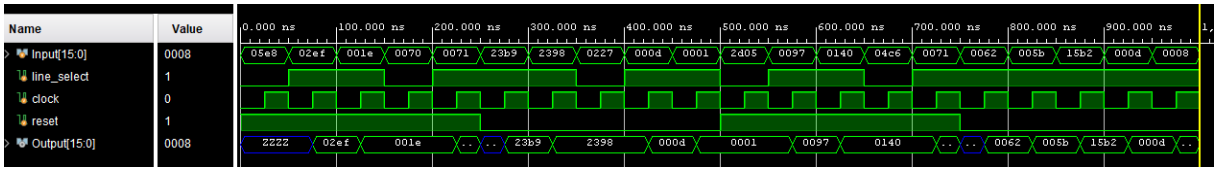



Figure 36: Simulation result of Part 1

In **second** part, we implement 16 line 16-bit register file module, which is totally 32 byte, by using decoder and register line module which we implemented in previous part. Also, we use 16-bit buffer when getting output from circuit. In this module, we can select two data with using selA and selB and get them as output and the selWrite is used for changing the data in the registers. And, reset is used for clearing all registers.

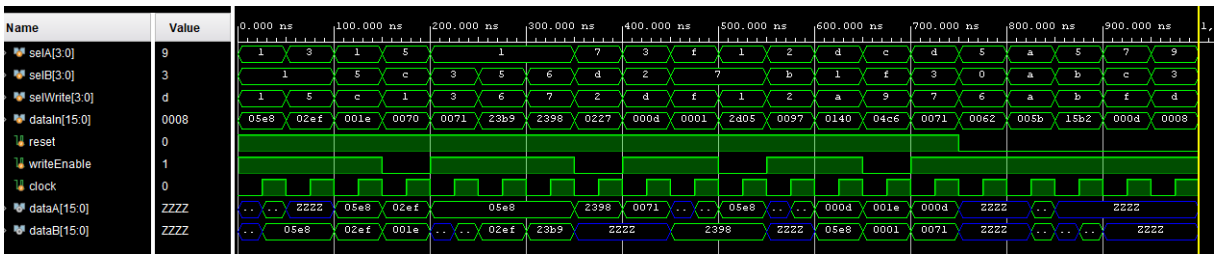
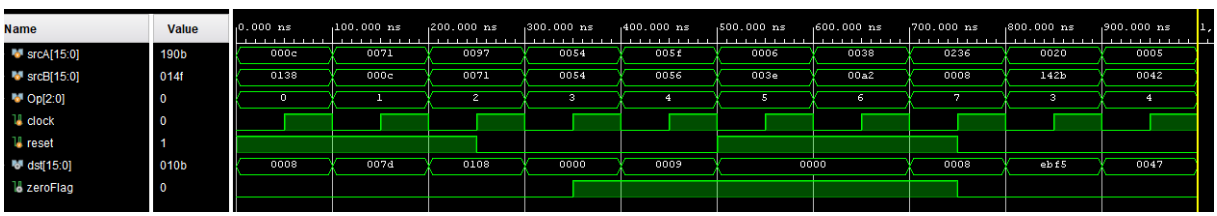



Figure 37: Simulation result of Part 2

In part **three**, the ALU, which is Aritmethic Logic Unit, is implemented according to given table about the ALU Instructions. The operations are "And", "OR", "Addition", "Subtraction", "XOR", "Logical Shift Right", "Logical Shift Left" and "Load". The operations are applied according to Opcode which is one of the input of ALU. Also in here, the reset and clock signals are used. Unlike other parts, there is a zeroFlag which is used for the result equals to zero or not and it is only updated in addition and subtraction operations. **Simulation result of Part 3:**



In part **four**, we implement Instruction Decoder which is basically decodes the instruction and creates output value according to given instruction. There are four types of Operation, they are Register, Load, Immediate and Branch. The outputs are different according to these operation types. Some outputs are not used in some operation types, so we use high impedance for them. And we created the outputs using these types and decoding instruction data.

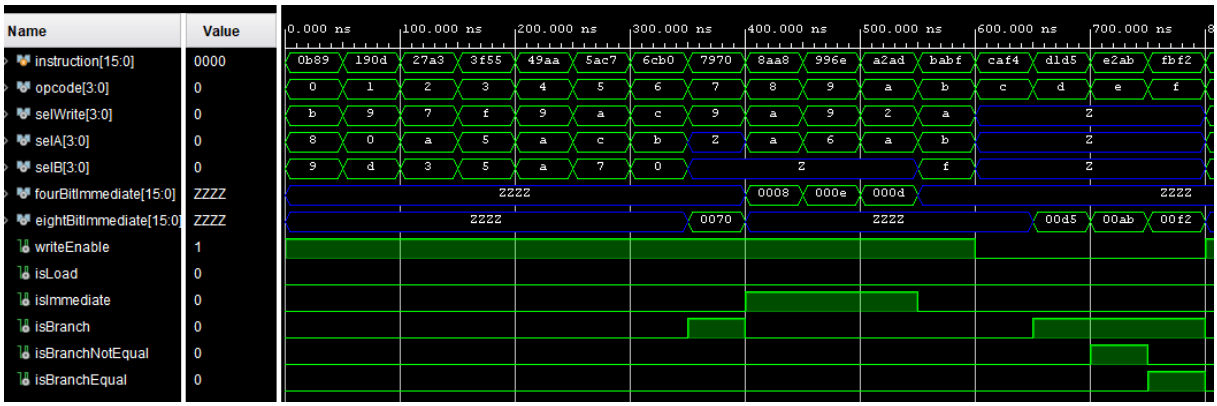



Figure 38: Simulation result of Part 4

In part **five**, we implement program counter module which is used for changing program counter module. The program counter is an address value that is used for storing the address of current program address. The operations are branching, resetting and incrementing the program counter value. This operations are provided according to the inputs isBranch, isBranchNotEqual, isBranchEqual and zeroFlag. And for synchronization the clock signal is used. Also, there is a reset signal for setting address value to zero.

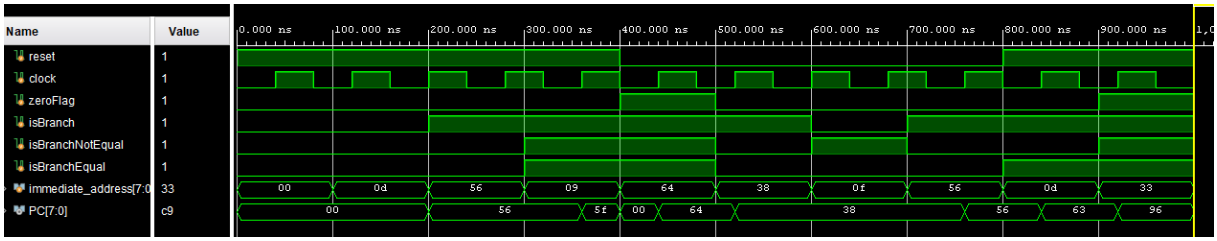



Figure 39: Simulation result of Part 5

And in **final** part, we tried to implement mini computer with using what we have implemented in previous parts. We connected all parts according to the homework document. However, we can not get an exact success in this part. There are some improper implementations. We tried to fix them but we failed.

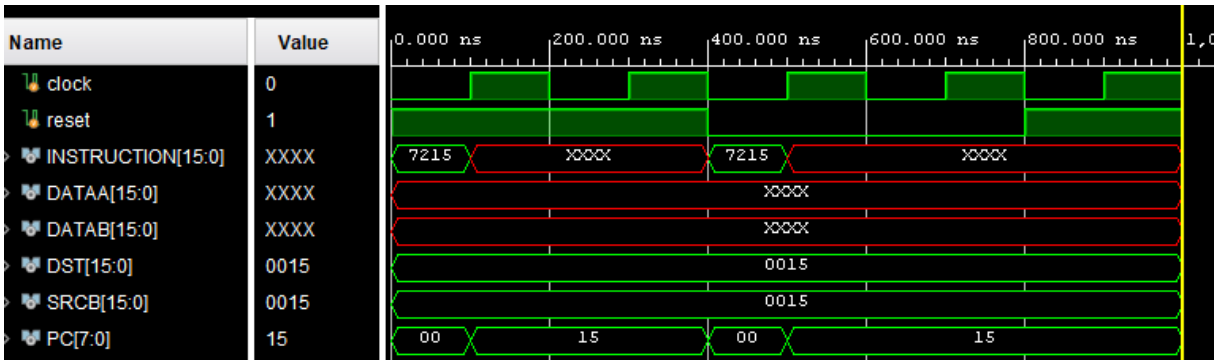



Figure 40: Simulation result of Part 6

# 5 CONCLUSION [10 points]

In Part 1, we designed 4:16 decoder module with enable input and 16-bit register line module. We have a line select, a clock and a reset input to determine what operation to take, clear or store.

Then in Part 2, we implemented a 16 line 16-bit register file module (32 byte) using modules that we designed in previous part. We also create our design according to required operations in experiment pdf file.

In Part 3, we learned how to design Arithmetic Logic Units with given operations list. In addition we used an update flag different from the other parts.

In Part 4, we designed the control signals of our digital system. We have implemented the instructions in the given tables one by one.

Then in Part 5, we add a program counter module for our design in order to store the current program address of the system.

Finally, we tried to combine all of the parts of the experiment and create the mini computer. In this part of the experiment, we used ROM module which is something new unlike other experiments. Although each part of the experiment, except Part 6, works in a proper way separately, we have not been able to combine all of those parts in Part 6. We had the most difficulty in this part.

In conclusion, we tried different methods, then couldn't manage to find the method that works properly. But the parts separately works fine, as the simulation results imply.