ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 242E
DIGITAL CIRCUITS LABORATORY EXPERIMENT REPORT

EXPERIMENT NO	:	7
EXPERIMENT DATE	:	07.05.2021
LAB SESSION	:	FRIDAY - 14.00
GROUP NO	:	G14

GROUP MEMBERS:

150180112	:	ÖMER MALİK KALEMBAŞI
150190014	:	FEYZA ÖZEN
150190108	:	EKİN TAŞYÜREK

SPRING 2021

Contents

1 INTRODUCTION [10 points]

In this experiment, we implemented a single cycle CPU with Verilog. In the 1st part we designed a register line and decoder. In the other parts we designed a register file, an ALU, an instruction decoder, and a program counter. At the end of the experiment, we combined them to implement a mini computer.

2 MATERIALS AND METHODS [40 points]

2.1 PART 1

In this part, we implemented a 4:16 decoder with enable input and then a 16-bit register line.

A decoder has n select inputs and gives 2^n outputs. Select inputs decide which output is going to be 1. When the selected output is 1, others are 0. If there is an enable input and it is 1, the decoder works. If the enable input is 0, all the outputs are 0. Our decoder module has 2 inputs that are 4-bit select input and 1 bit enable input. Its output is 16-bit output.

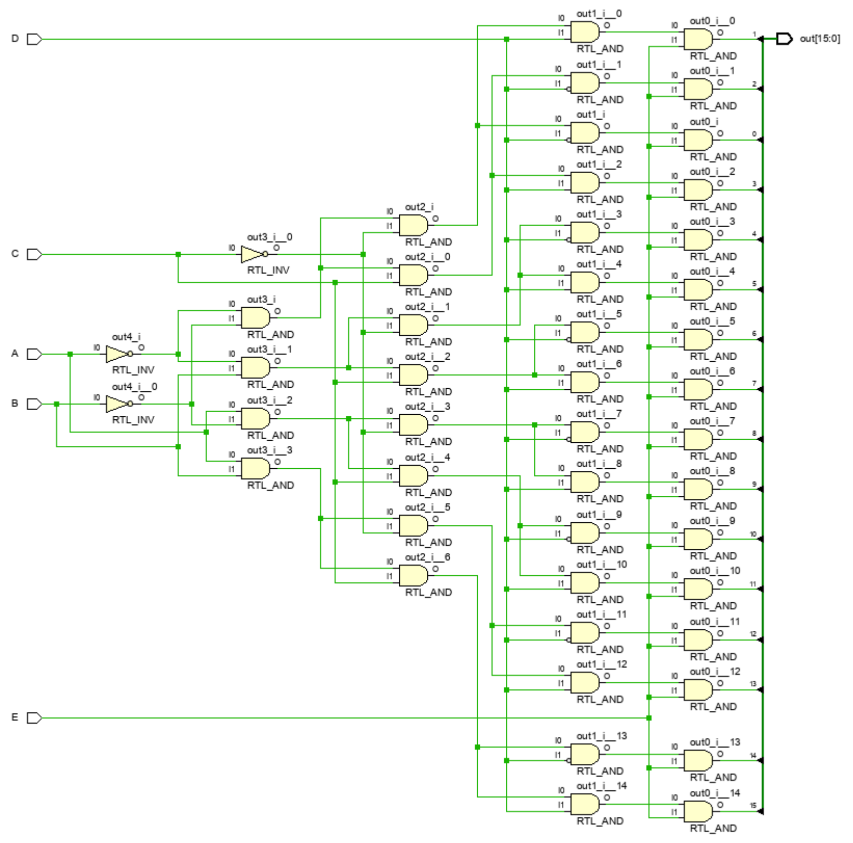


Figure 1: RTL Schematic of 4:16 Decoder

The 16-bit register line module that we implemented has 4 inputs. These are lineselect, clock, reset, and 16-bit dataIn. If lineselect is 1, our register line module stores dataIn in it. Then gives 16-bit stored data as output. That operation works at the rising edge of the clock signal. At the falling edge of the reset signal data will be cleared and 0.

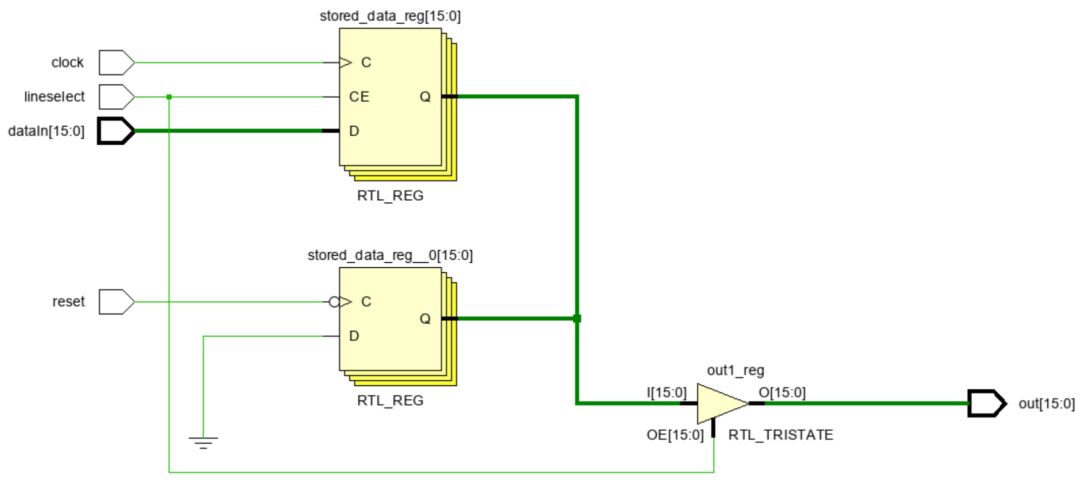


Figure 2: RTL Schematic of Register Line

2.2 PART 2

In this part, we implemented a (32 byte) register file module using 4:16 decoder module and 16-bit register line module that we implemented in Part 1. A register file is memory storage in a CPU. Register files have bits of data and mapping locations. These locations show the addresses that are inputs of a register file. Our register file module takes 4-bit selA, 4-bit selB, 4-bit selWrite, 16-bit dataIn, reset, writeEnable, and a clock signal as inputs. The module gives 16-bit dataA and 16-bit dataB as outputs. The module works at the rising edge of the clock signal. At the falling edge of the reset, the register is cleared. The selWrite input corresponds to the address of the line and it provides to update the value at the selected line and store dataIn input there. selA and selB select the line of the register file for dataA and dataB output. To implement it, we used 2 different decoder that takes selA and selB bits separately as input.

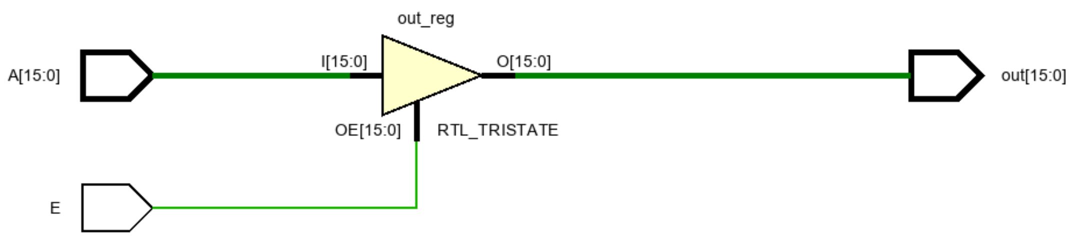


Figure 3: RTL Schematic of Three-State Buffer

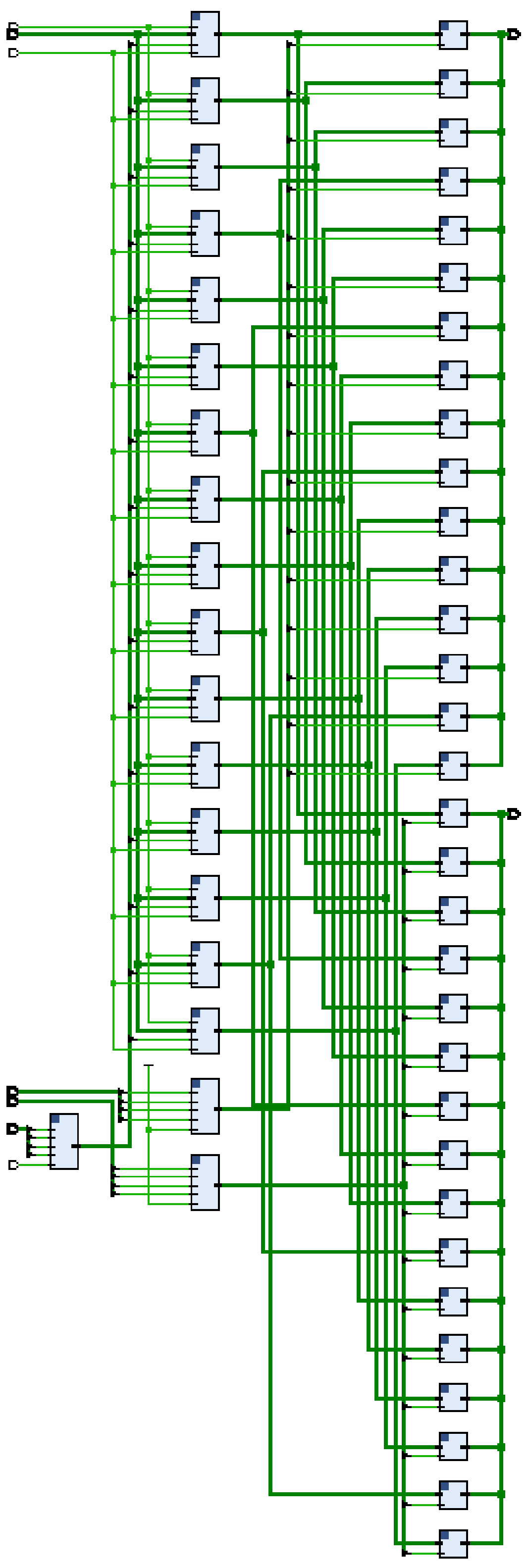


Figure 4: RTL Schematic of Part 2

2.3 PART 3

In this part, implemented an Arithmetic Logic Units (ALU) that does some operations in the tablel in Figure 5. An ALU is the part of a computer CPU that does arithmetic and logic operations. ALU takes srcA, srcB, and 3-bit op as inputs and gives 16-bit dst and zeroFlag as output. 3-bit op input determines the operation that operates. zeroFlag is only updated when op is 2 or 3. ALU works at the rising edge of the clock signal. At the falling edge of the reset signal, zeroFlag will be cleared and 0.

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 ˆ srcB	No
5	Logical Shift Right	dst = srcA >> srcB	No
6	Logical Shift Left	dst = srcA << srcB	No
7	LOAD	dst = srcB	No

Figure 5: ALU Instructions

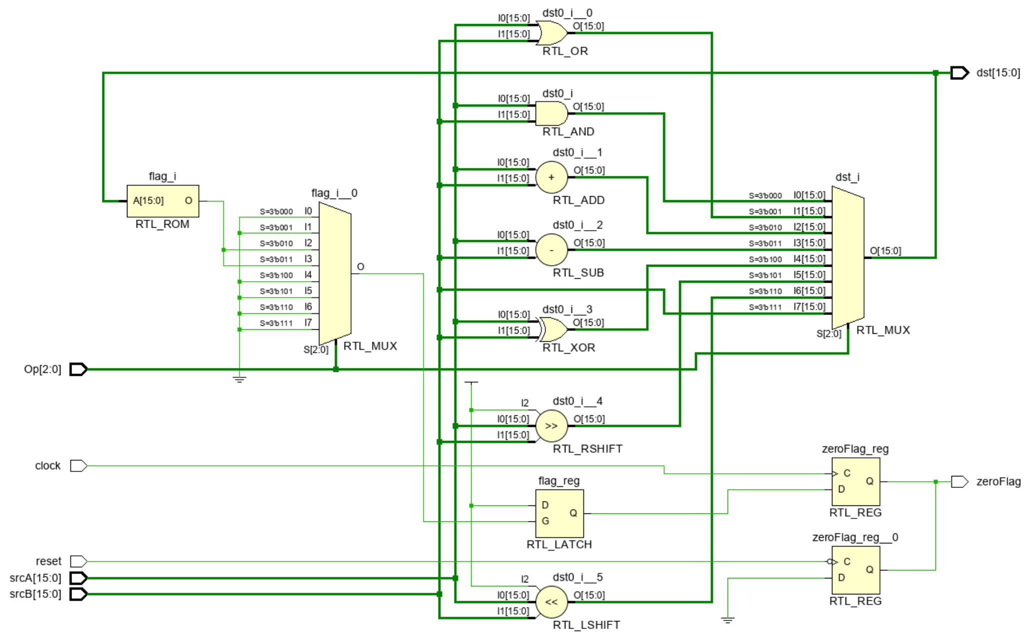


Figure 6: RTL Schematic of Part 3

2.4 PART 4

In this part, we implemented an instruction decoder. It creates the control signals of our digital system according to 16-bit instruction input that comes from the ROM. Figure x shows the instruction set of the digital system. There are 4 types of instructions that are register, load, immediate, and branch. The meaning of the input bits changes according to the instruction type. The first 4 bit is always operation code. It determines which operation operates. The second 4 bit is selWrite which is the destination address for writing the operation result. The third and fourth 4 bits are selA and selB which are operand address information.

If the instruction type is register, the third 4 bit is srcA, the fourth 4 bit is srcB. The second 4 bit is dst which is the result of the operation between the values at srcA and srcB.

If the instruction type is immediate, the third 4 bit is srcA, the fourth 4 bit is four-BitImmediate . The second 4 bit is dst

If the instruction type is load, the last 8 bit is immediate, the second 4 bit is dst.

If the instruction type is branch, the last 8 bit is immediate, the second 4 bit is unused.

Some of the outputs are part of the instruction input, some are not. There are other 1 bit outputs. writeEnable, isLoad, isImmediate, isBranch. IsBranchNotEqual, and BranchEqual.

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 ˆ 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

Figure 7: Digital System Instruction Set

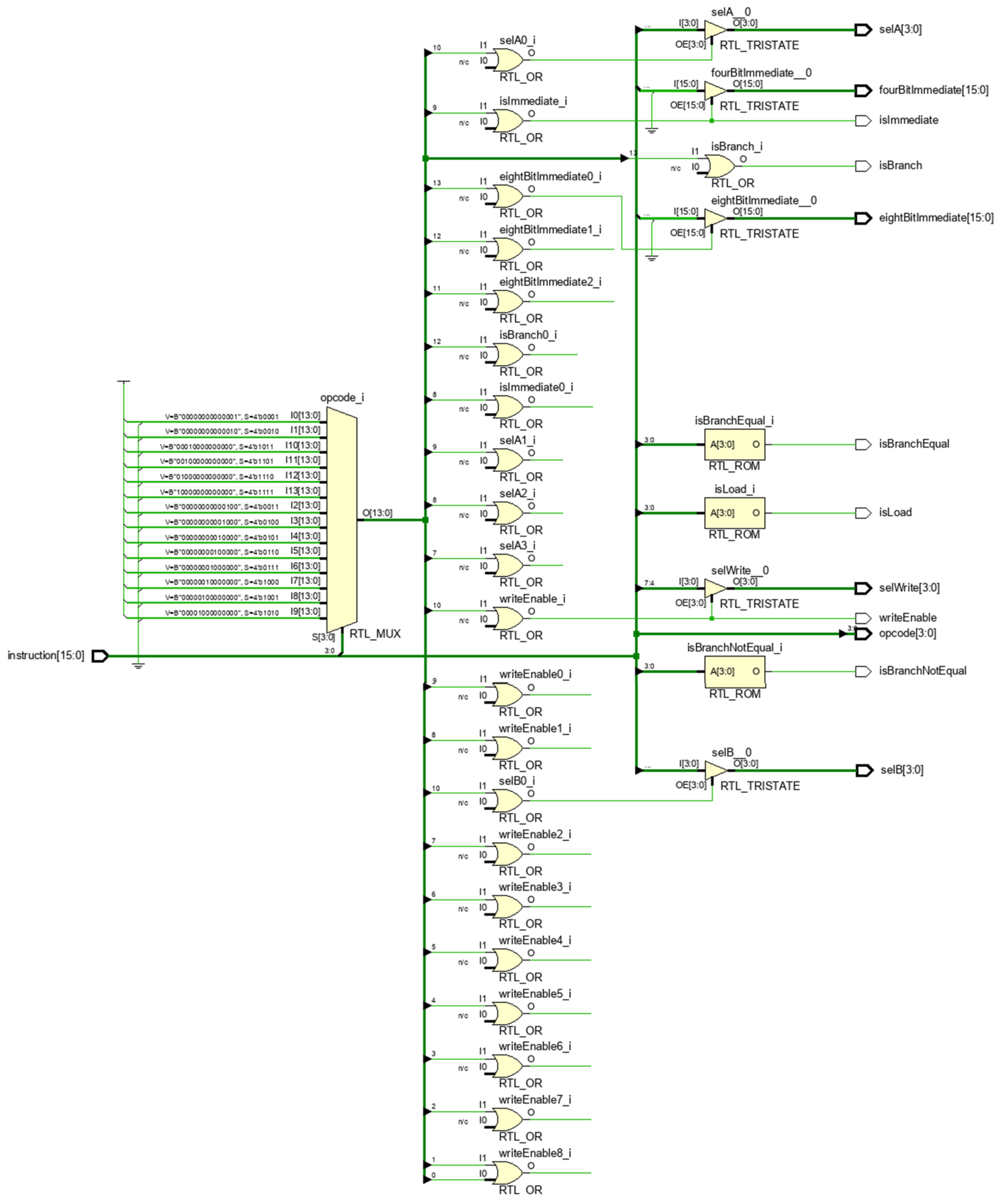


Figure 8: RTL Schematic of Part 4

2.5 PART 5

In this part, we implemented a program counter. It takes reset signal, clock signal, zeroFlag, isBranch, isBranchNotEqual, isBranchEqual, and 8-bit immediateAddress as inputs. It gives PC output. PC will send the address of the next process. The circuit operates when the clock is rising edge. isBranch, isBranchNotEqual, and isBranchEqual inputs decide what PC is going to be. If reset input is falling edge, the immediateAdress input is cleared.

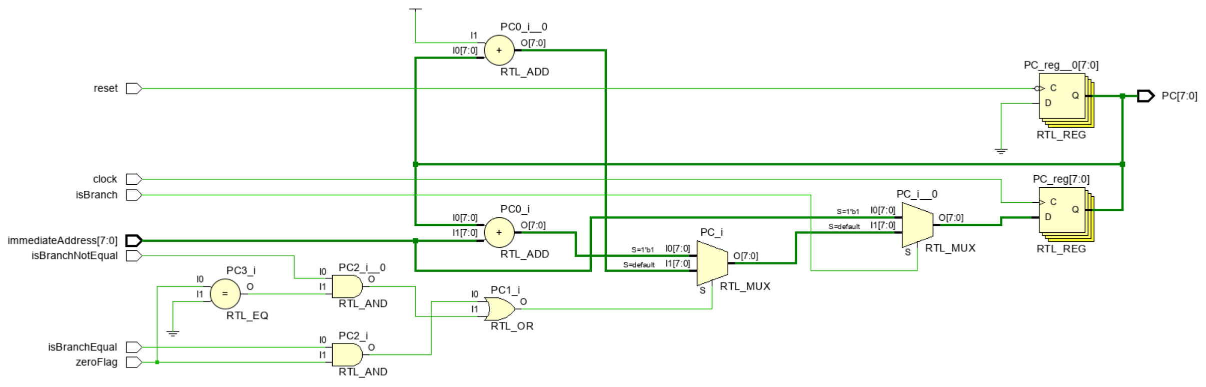


Figure 9: RTL Schematic of Part 5

2.6 PART 6

In this part, we implemented a mini-computer using the modules that we implemented in the previous parts. The mini-computer takes the clock signal and reset signal as input. The computer contains a program memory, an instruction decoder, a register file, an ALU, and a program counter. The instruction comes from program memory. 16-bit instruction is the input of the instruction decoder. The details of the mini-computer are in the discussion part.

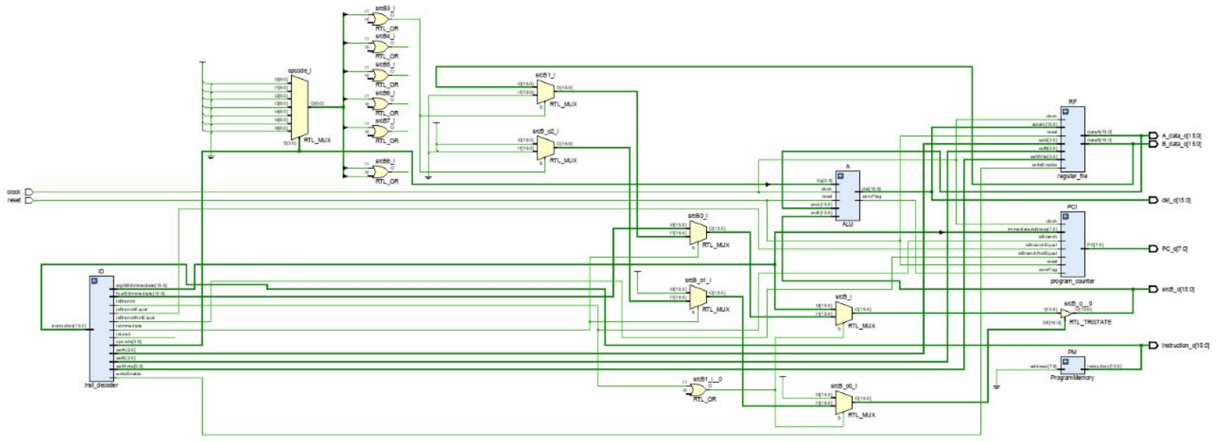


Figure 10: RTL Schematic of Part 6

3 RESULTS [15 points]

3.1 PART 1

In decoder simulation results, A, B, C, D are selector inputs and E is enable input. Our 16:4 decoder produces 2^n minterms, here n is the number of selector inputs which is 4. In first block of simulation picture, A=0, B=0, C=0, D=0, E=1; enable is high, allows to produce 16 minterms. According to selector inputs, only A'B'C'D' minterm will be 1, rest of them will be 0.

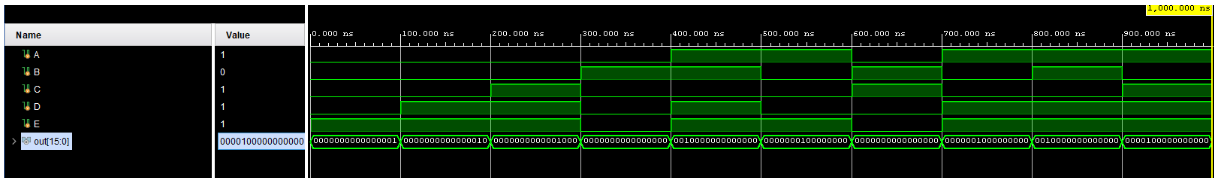


Figure 11: Simulation of Decoder

In register line simulation, there are 16-bit dataIn, 1-bit lineselect, 1-bit clock and 1-bit reset inputs and 16-bit output. At the rising edge of clock signal and high lineselect input, 16-bit output will be equals to input DataIn, at the falling edge of reset signal, 16-bit output will be cleared. Otherwise output will be high impedance.

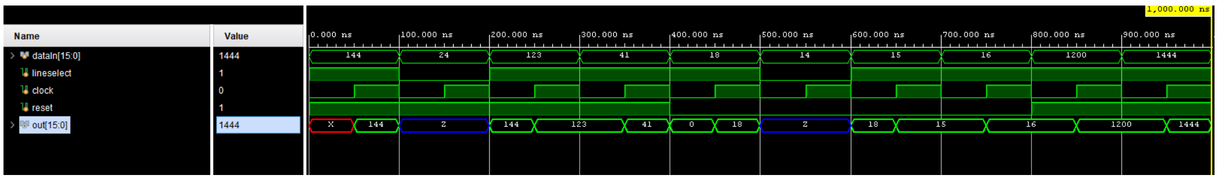


Figure 12: Simulation of Register Line

3.2 PART 2

In part 2 register file simulation, there are inputs selA, selB, selWrite, dataIn, reset, writeEnable, clock and there are outputs dataA and dataB. 4-bit selA and 4-bit selB inputs select which register lines to use for dataA and dataB outputs. selWrite input selects the line to be updated. The selected line stores the 16-bit dataIn at rising edge of clock signal and writeEnable is high. At the falling edge of reset signal register lines will be cleared as shown in 400. ns in simulation picture.

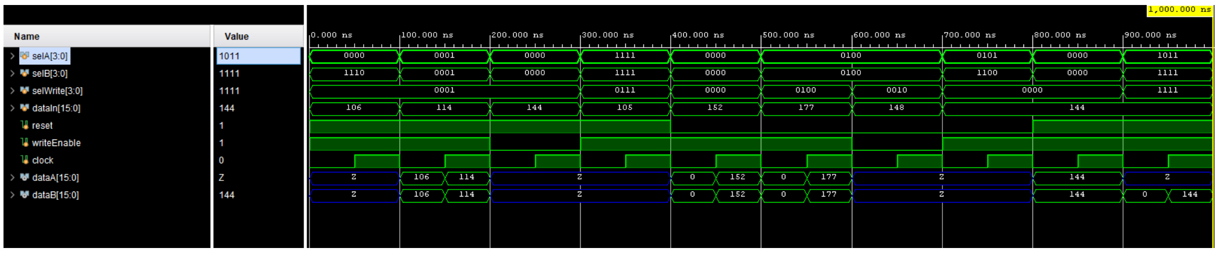


Figure 13: Simulation of Part 2

3.3 PART 3

In part 3 ALU simulation results, there are two 16-bit inputs srcA and srcB. These are our resources to implement arithmetic operations on. There are also 3-bit Op, 1-bit clock, reset and inputs. 3-bit Op input selects the arithmetic operation to implement srcA and srcB according to Table in assignment page at rising edge of clock signal. For example, here at 300. ns of simulation picture, Op=decimal 2, which is addition operation. srcA=decimal 120 and srcB=decimal 100. Result (dst) will be 120+100=220. At falling edge of reset signal 16-bit dst output will be cleared. Unfortunately, our zeroFlag output does not work properly.

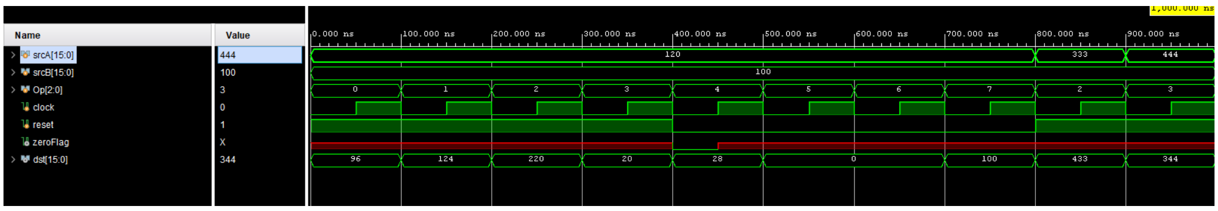
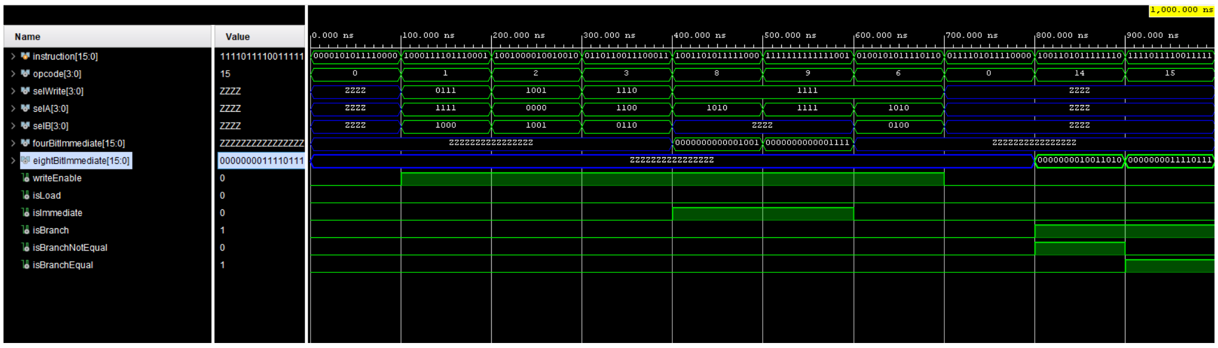


Figure 14: Simulation of Part 3

3.4 PART 4

In part 4 instruction decoder simulation, there are 16-bit instruction input and 16-bit fourBitImmediate, 16-bit eightBitImmediate, 4-bit opcode, 4-bitselWrite, 4-bit selA, 4-bit selB, 1-bit writeEnable, isLoad, isImmediate, isBranch, isBranchnotequal, isBranchequal outputs. First 4 digits of of instruction will be opcode. Rest of them change according to instruction type. For example, in second block opcode is 0001 which is OR operation, second 4-bit unit is selWrite, third 4-bit unit is selA and last 4-bit unit is selB. According to value of opcode fourBitImmediate and eightBitImmediate outputs store extended unused selA and selB.



```
instruction[15:0]       1111011110011111  0000101011110000 1000111101110001 1001000010010010 0110110011100011 1001101011111000 1111111111111001 0100101011110110 0111101011110000 1001101011111110 1111011110011111
opcode[3:0]             15                0 1 2 3 8 9 6 0 14 15
selWrite[3:0]           ZZZZ              ZZZZ 0111 1001 1110 1111 ZZZZ
selA[3:0]               ZZZZ              ZZZZ 1111 0000 1100 1010 1111 1010 ZZZZ
selB[3:0]               ZZZZ              ZZZZ 1000 1001 0110 ZZZZ 0100 ZZZZ
fourBitImmediate[15:0]  ZZZZZZZZZZZZZZZZ  ZZZZZZZZZZZZZZZZ 0000000000001001 0000000000001111 ZZZZZZZZZZZZZZZZ
eightBitImmediate[15:0] 0000000011110111  ZZZZZZZZZZZZZZZZ 0000000010011010 0000000011110111
writeEnable             0
isLoad                  0
isImmediate             0
isBranch                1
isBranchNotEqual        0
isBranchEqual           1
```

Figure 15: Simulation of Part 4

3.5 PART 5

In part 5 program counter simulation, there are inputs reset, clock, zeroFlag, isBranch, isBranchEqual, isBranchNotEqual and 8-bit immediaateAddress. Our output is 8-bit PC. PC stores the current program address of the system. At the falling edge of reset signal PC will be cleared. Assuming operation is branch, at the rising edge of clock signal PC value will be immediateAddress (first block of simulation). At the rising edge of clock signal and isBranchNotEqual is high PC value will be current PC value + immediateAddress (fourth block of simulation). At the rising edge of clock signal, isBranchEqual is high and zeroFlag is shows the result is equal PC value will be current PC value + immediateAddress (fourth block of simulation). Otherwise PC value will be the next address of the memory at the rising edge of clock signal (PC = PC +1).

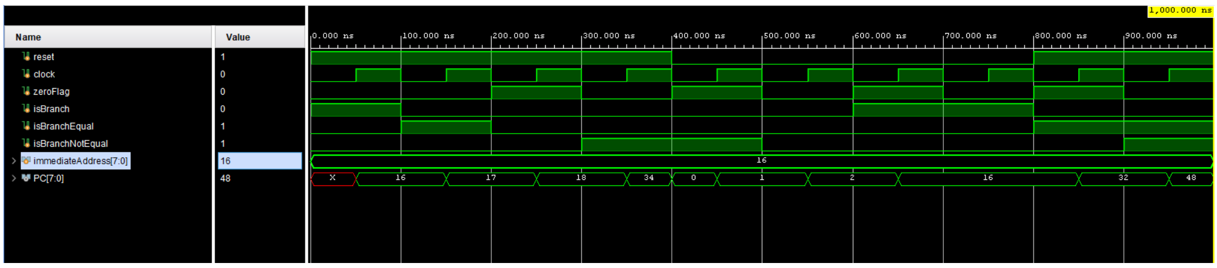


Figure 16: Simulation of Part 5

3.6 PART 6

Unfortunately simulation results shows that Part 6 does not work properly.

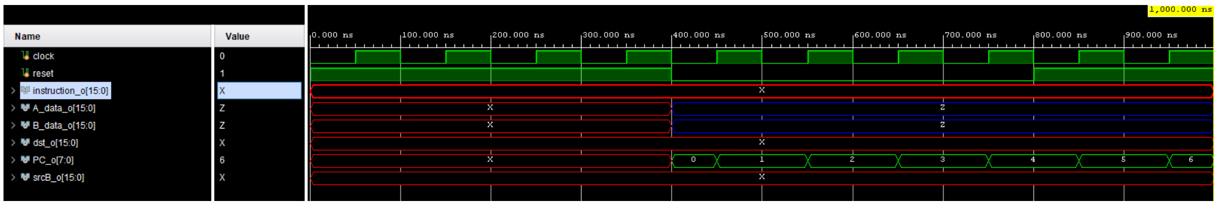


Figure 17: Simulation of Part 6

4 DISCUSSION [25 points]

In Part 1, we implemented a 4:16 decoder with enable input and then a 16-bit register line. We used 3 always block to implement register line.

The first one is for reseting. It clears the stored_data and makes it 0 when the reset signal falls.

The second one is for storing the dataIn in stored_data when the clock rises.

The third one is for transfering stored_data to out1. If lineselect is 1 that operation is going to work. If lineselect is 0, the out1 is going to be high impedence.

```
always @(negedge reset)begin
        stored_data <= 16'h00;
    end

    always @(posedge clock)begin

        if(clock && lineselect) begin
            stored_data <= dataIn;
        end
    end

    always @(*) begin
        if(lineselect) begin
            out1 <= stored_data;
        end
        else begin
            out1 <= 16'hZZ;
        end
    end

    assign out=out1;
```

Listing 1: Register line code

In Part 2, we implemented a (32 byte) register file module using 4:16 decoder module and 16-bit register line module that we implemented in Part 1.

```
 decoder D(selWrite[0], selWrite[1], selWrite[2], selWrite[3],
   writeEnable, dec_out);
```

Listing 2: Decoder

Decoder D has 16 bit output. We used them as enable inputs later.

```
register_line RL1(dataIn, dec_out[0], clock, reset, reg_out1);
register_line RL2(dataIn, dec_out[1], clock, reset, reg_out2);
    .
    .
    .
register_line RL16(dataIn, dec_out[15], clock, reset, reg_out16);
```

Listing 3: Register Line

There are 16 reg_out. Register line that we used uses decoder output bits as lineselect input. Line select acts like Enable here.

```
decoder D1(selA[0], selA[1], selA[2], selA[3], 1, selA_out);
```

Listing 4: Decoder

Again, decoder D1 has 16 bit output. We used them as enable inputs later.

```
tsb TSB0(reg_out1,selA_out[0],dataA);
tsb TSB1(reg_out2,selA_out[1],dataA);
    .
    .
    .
tsb TSB15(reg_out16,selA_out[15],dataA);
```

Listing 5: TSB

There are 16 three-state buffer here. They takes decoder outputs as enable input. They take register outputs as data input. When enable is 1, dataA inputs are equal to register outputs.

```
decoder D2(selB[0], selB[1], selB[2], selB[3], 1, selB_out);
```

Listing 6: Decoder

Again, decoder D2 has 16 bit output. We used them as enable inputs later.

```
tsb TSB16(reg_out1,selB_out[0],dataB);
tsb TSB17(reg_out2,selB_out[1],dataB);
    .
    .
    .
tsb TSB31(reg_out16,selB_out[15],dataB);
```

Listing 7: TSB

There are 16 three-state buffer here. They takes decoder outputs as enable input. They take register outputs as data input. When enable is 1, dataB inputs are equal to register outputs.

In Part 3, we implemented an Arithmetic Logic Unit. We used switch-case block to operate according to 3-bit Op input. There are 8 possibilities.

```
always @(*) begin
    case(Op)
        3'h0: dst <= srcA & srcB;
        3'h1: dst <= srcA | srcB;
        3'h2: begin
            dst <= srcA + srcB;
            if(dst==0) begin
                flag=1;
            end
        end
        3'h3: begin
            dst <= srcA - srcB;
            if(dst==0) begin
                flag=1;
            end
        end

        3'h4: dst <= srcA ^ srcB;
        3'h5: dst <= srcA >> srcB;
        3'h6: dst <= srcA << srcB;
        3'h7: dst <= srcB;

        endcase
end
```

Listing 8: ALU

```
always @(negedge reset) begin
    zeroFlag = 1'h0;
end

always @(posedge clock)begin
    zeroFlag = flag;
```

```
end
```

Listing 9: zeroFlag

zeroFlag is 0 when the reset signal is at a falling edge. When the clock is at a rising edge, zeroFlag is a temporary flag variable that we assigned when needed above.

In Part 4, we implemented an instruction decoder.

```
assign opcode = instruction[3:0];

assign writeEnable = (opcode == 4'd1) | (opcode == 4'd2) | (opcode == 4'd3) | (opcode == 4'd4) | (opcode == 4'd5) | (opcode == 4'd6) | (opcode == 4'd7) | (opcode == 4'd8) | (opcode == 4'd9) | (opcode == 4'd10) | (opcode == 4'd11) ? 1:0;
assign isLoad = (opcode == 4'd7) ? 1:0;
assign isImmediate = (opcode == 4'd8) | (opcode == 4'd9) | (opcode == 4'd10) ? 1:0;
assign isBranch = (opcode == 4'd13) | (opcode == 4'd14) | (opcode == 4'd15) ? 1:0;
assign isBranchNotEqual = (opcode == 4'd14) ? 1:0;
assign isBranchEqual = (opcode == 4'd15) ? 1:0;

assign selWrite = (opcode == 4'd1) | (opcode == 4'd2) | (opcode == 4'd3) | (opcode == 4'd4) | (opcode == 4'd5) | (opcode == 4'd6) | (opcode == 4'd7) | (opcode == 4'd8) | (opcode == 4'd9) | (opcode == 4'd10) | (opcode == 4'd11) ? instruction[7:4]: 4'bZ;

assign selA = (opcode == 4'd1) | (opcode == 4'd2) | (opcode == 4'd3) | (opcode == 4'd4) | (opcode == 4'd5) | (opcode == 4'd6) | (opcode == 4'd8) | (opcode == 4'd9) | (opcode == 4'd10) | (opcode == 4'd11) ? instruction[11:8] : 4'bZ;
assign selB = (opcode == 4'd1) | (opcode == 4'd2) | (opcode == 4'd3) | (opcode == 4'd4) | (opcode == 4'd5) | (opcode == 4'd6) | (opcode == 4'd11)? instruction[15:12] : 4'bZ;

assign fourBitImmediate = (opcode == 4'd8) | (opcode == 4'd9) | (opcode == 4'd10) ? {12'd0,instruction[15:12]}:16'bZ;
assign eightBitImmediate = (opcode == 4'd7) | (opcode == 4'd13) | (opcode == 4'd14) | (opcode == 4'd15) ?{8'd0, instruction[15:8]}:16'bZ;
```

Listing 10: Instruction decoder

We assigned writeEnable, isLoad, isImmediate, isBranch, isBranchNotEqual, isBranchEqual by using short if block. There are 2 possibilities for them and these are 1 and 0. opcode(first 4 digits of instruction) input decides what they are going to be.

Then we assigned selA and selB using short if block. There are 2 posibilities for them and these are instruction[7:4] for selA, instruction[15:12] for selB and high impedence.

Then we did same things to fourBitImmediate and eightBitImmediate.

In Part 5, we implemented a program counter.

```
always @(negedge reset) begin
        PC <=8'h0;
    end

always @(posedge clock) begin

        if(isBranch==1 && clock)begin
            PC <= immediateAddress;
        end

        else if ((isBranchEqual==1 && zeroFlag==1 && clock) || (
    isBranchNotEqual==1 && zeroFlag==0 && clock))begin
            PC <= PC + immediateAddress;
        end
        else begin
            PC <= PC + 8'd1;
        end
    end
```

Listing 11: PC

In the first always block, it resets and makes the PC output 0 when the reset signal is at the falling edge.

In the second always block, it stores immediateAddress in PC, when isBranch is 1 and the clock is at the rising edge. In the "else if" condition where isBranchNotEqual is 1, zeroFlag is 0, and the clock is rising edge, PC equals PC + immediateAddress. In "else" condition PC equals PC + 1.

In Part 6, we implemented a mini computer.

```

ProgramMemory PM(pc,instruction);

```

```
4  inst_decoder ID(instruction,opcode,selWrite,selA,selB,fourBitImmediate,
       eightBitImmediate,writeEnable,isLoad,isImmediate,isBranch,
       isBranchNotEqual,isBranchEqual);
5
6  register_file RF(selA, selB, selWrite, dst, reset, writeEnable, clock,
       dataA,dataB);
7
8  assign srcB = (isLoad | isBranch) ? eightBitImmediate: (isImmediate)?
       fourBitImmediate:((opcode == 4'd1) | (opcode == 4'd2) | (opcode == 4'
       d3) | (opcode == 4'd4) | (opcode == 4'd5) | (opcode == 4'd6) | (
       opcode == 4'd11))? dataB:4'bZ;
9
10 ALU A(dataA,srcB,opcode[2:0],clock,reset,zeroFlag,dst);
11
12 program_counter PC1(reset, clock, zeroFlag, isBranch,isBranchEqual,
       isBranchNotEqual, eightBitImmediate[8:0],PC);
```

Listing 12: Mini-Computer

In the 2nd line, we used the given module ProgramMemory. It takes PC output as input, then gives instruction as output.

Instruction output is used in inst_decoder. We used the decoder's outputs later in other lines.

We used a register_file module RF. It takes selA,selB, selWrite, and writeEnable which are the outputs of the Instruction Decoder. Also, it takes dst output of the ALU.

We assigned a value to srcB according to the output of the decoder.

We used an ALU. It we used three bit of opcode as input of ALU A. It takes dataA output of RF as its srcA input. Its srcB input is srcB that is assigned before in line 8.

At the end, we used a program counter PC1. It takes zeroFlag output of the ALU as its zeroFlag. Also it takes isBranch, isBranchNotEqual, isBranchEqual and immediateAddress which are the outputs of ID.

5 CONCLUSION [10 points]

In this experiment, we learned how to implement a mini-computer and its components such as register file, instruction decoder, ALU, and Program Counter. We had some difficulties and one of them was in Part 2. We tried to implement Part 2 with a 16:1 multiplexer first, but unfortunately our code didn't work. So we tried to find an alternative solution and decided to three state buffers. In addition, we could not implement Part 6 properly. It's simulation results show us that.