ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E

COMPUTER ORGANIZATION
PROJECT REPORT

PROJECT NO : 1

PROJECT DATE : 28.04.2022 - 11.05.2022

LAB SESSION : FRIDAY - 10.30

GROUP NO : G56

GROUP MEMBERS:

150180089 : Harun ÇİFCİ

150190724 : Umut TÖLEK

150200705 : Helin Aslı AKSOY

SPRING 2022

Contents

1 INTRODUCTION [10 points]

In this project, firstly, we are asked to design 8-bit and 16-bit register. Then, by using those registers we have designed Regular Register File and Adress Register File. Following that design we have designed an ALU with 16 different Operations included. After designing all components, we have created a basic computer that can read a value from memory, that can make Arithmetic and Logical operations and save it back to memory.

2 MATERIALS AND METHODS [40 points]

2.1 PART 1

In this part, we have designed 2 types of registers, one of them is 8-bit, other one is 16-bit. Registers has decrement, increment, clear and load functionalities that are controlled by 2-bit control signals (FunSel) and an enable input (E).

2.1.1 8-Bit Register

We have implemented 8-Bit register with 8-bit adder for increment function, 8-bit subtractor for decrement function. We have used 4 to 1 8-bit Multiplexer to select functionalities such as increment(Output of adder), decrement(Output of subtractor), load(takes direct 8-Bit Input), clear(connected to ground). Then we have connected output of Multiplexer to 8-Bit D Flip Flop system.

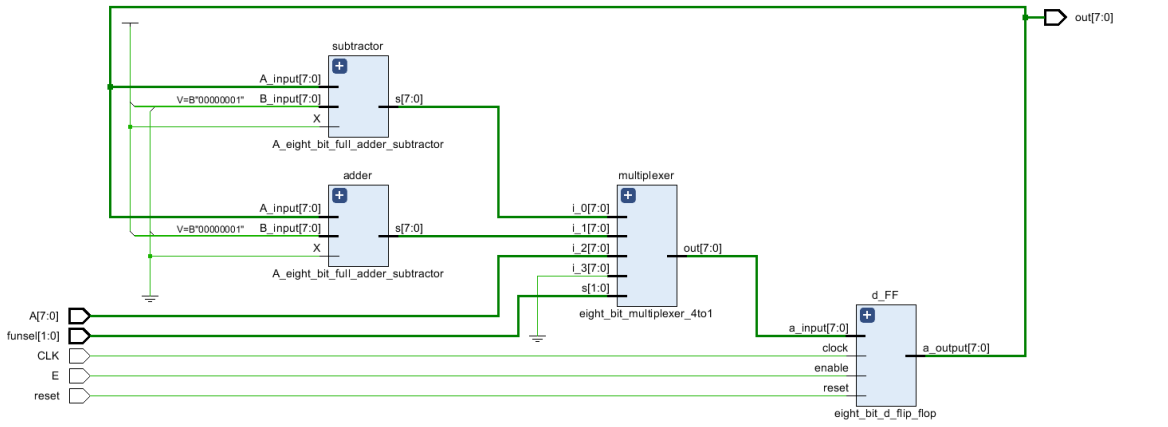

Figure 1: Elaborated Design of 8-Bit Register[]

2.1.2 16-Bit Register

We have implemented 16-Bit register with 16-bit adder for increment function, 16-bit subtractor for decrement function. We have used 4 to 1 16-bit Multiplexer to select functionalities such as increment(Output of adder), decrement(Output of subtractor), load(takes direct 16-Bit Input), clear(connected to ground). Then we have connected output of Multiplexer to 16-Bit D Flip Flop system.

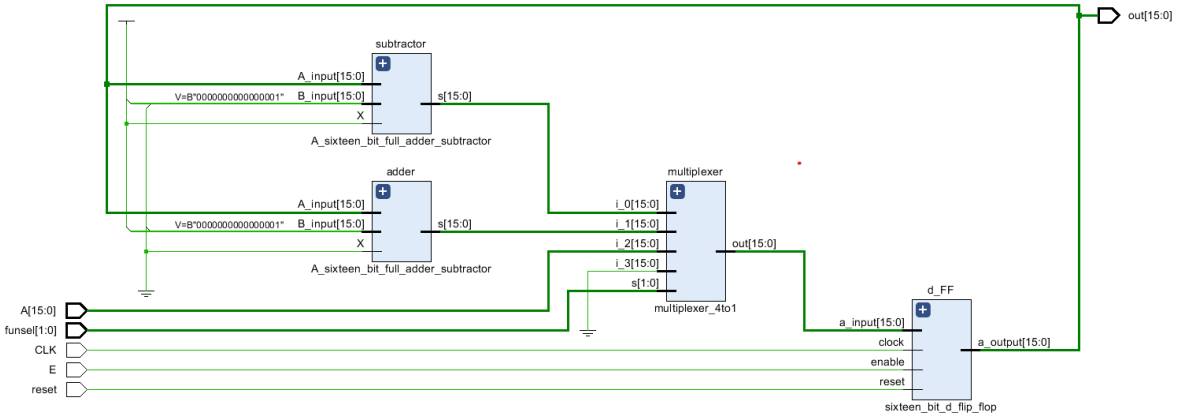

Figure 2: Elaborated Design of 16-Bit Register[]

2.2 PART 2

In this part, we have designed a regular register file and a address register file.

2.2.1 Part 2.a

We have used 4 8-bit register, two 4 to 1 multiplexers for different outputs A and B, 4-bit Register Selector Input, 2-bit Output A Selector and 2-bit Output B Selector Inputs.

Register selector input is responsible for enable/disable registers. All 4 registers connected to both output channels A and B. Multiplexers are responsible for which of the register outputs will be the output of File. In order to select File outputs, we have used A and B output selectors.

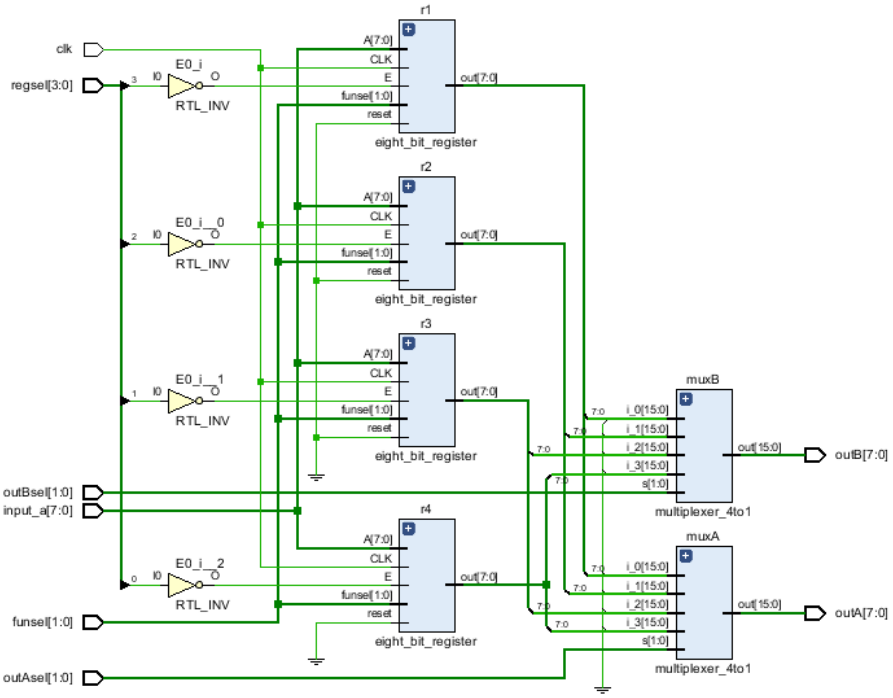

Figure 3: Elaborated Design of Regular Register File[]

2.2.2 Part 2.b

We have used 3 8-bit registers, two 4 to 1 multiplexers for different outputs C and D, 3-bit Register Selector Input, 2-bit Output A Selector and 2-bit Output B Selector Inputs.

Register selector input is responsible for enable/disable registers. All 3 registers connected to both output channels C and D. Multiplexers are responsible for which of the register outputs will be the output of File. In order to select File outputs, we have used C and D output selectors. Both Multiplexers gives PC output with 00 and 01 selector values, and gives AR and SP outputs respectively with 10 and 11 selector values.

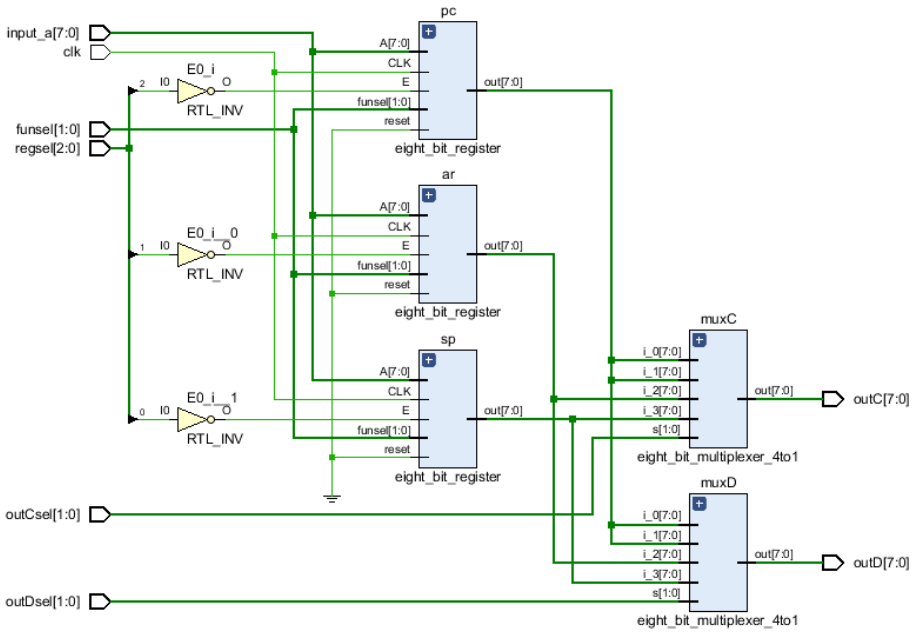

Figure 4: Elaborated Design of Addres Register File[]

2.2.3 Part 2.c

We have implemented 16-Bit Instruction Register with 16-bit adder for increment function, 16-bit subtractor for decrement function. We have used 4 to 1 16-bit Multiplexer to select functionalities such as increment(Output of adder), decrement(Output of subtractor), load(takes direct 8-Bit Input), clear(connected to ground). Then we have connected output of Multiplexer to 16-Bit D Flip Flop system.

Different from regular 16-bit register, we have designed assignbits module. With assignbits module according to L/H signal we have assigned 8 bits input to the appropriate place. Then we have connected its output to "load" port of Multiplexer.

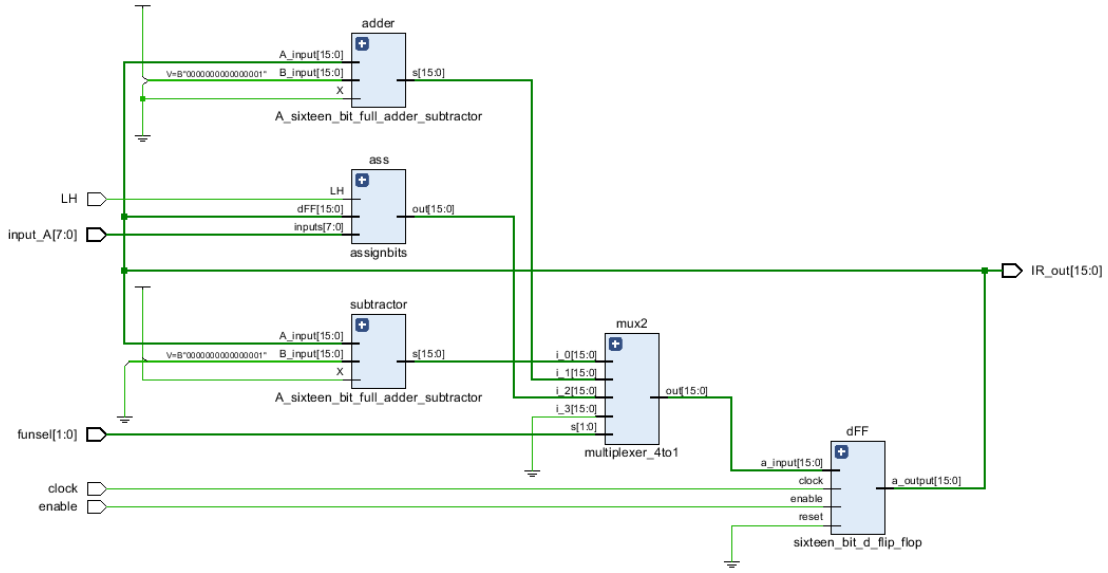

Figure 5: Elaborated Design of 16-bit Instruction Register[]

2.3 PART 3

In this part, we have designed ALU with 16 operations. Result of these operations we have 4-bit flag register that includes Z, N, C and O single bit representation. Z means the result is zero, C means the result has carry output, N means the result is negative, and O means the result has overflow. The operations are listed below;

FunSel	OutALU	Z	C	N	O
0000	A	√	–	√	–
0001	B	√	–	√	–
0010	NOT A	√	–	√	–
0011	NOT B	√	–	√	–
0100	A + B	√	√	√	√
0101	A + B + Carry	√	√	√	√
0110	A - B	√	√	√	√
0111	A AND B	√	–	√	–
1000	A OR B	√	–	√	–
1001	A XOR B	√	–	√	–
1010	LSL A	√	√	√	–
1011	LSR A	√	√	√	–
1100	ASL A	√	–	√	√
1101	ASR A	√	–	–	–
1110	CSL A	√	√	√	√
1111	CSR A	√	√	√	√

Figure 6: Operations of ALU[]

In order to Implement Arithmetic and Logical Unit, we have used Behavioral modelling approach. In an always block,the case is being determined with 4-Bit Funsel input. Once the case is being selected, by checking some special "if" conditions, ALU does the related operations.Instruction

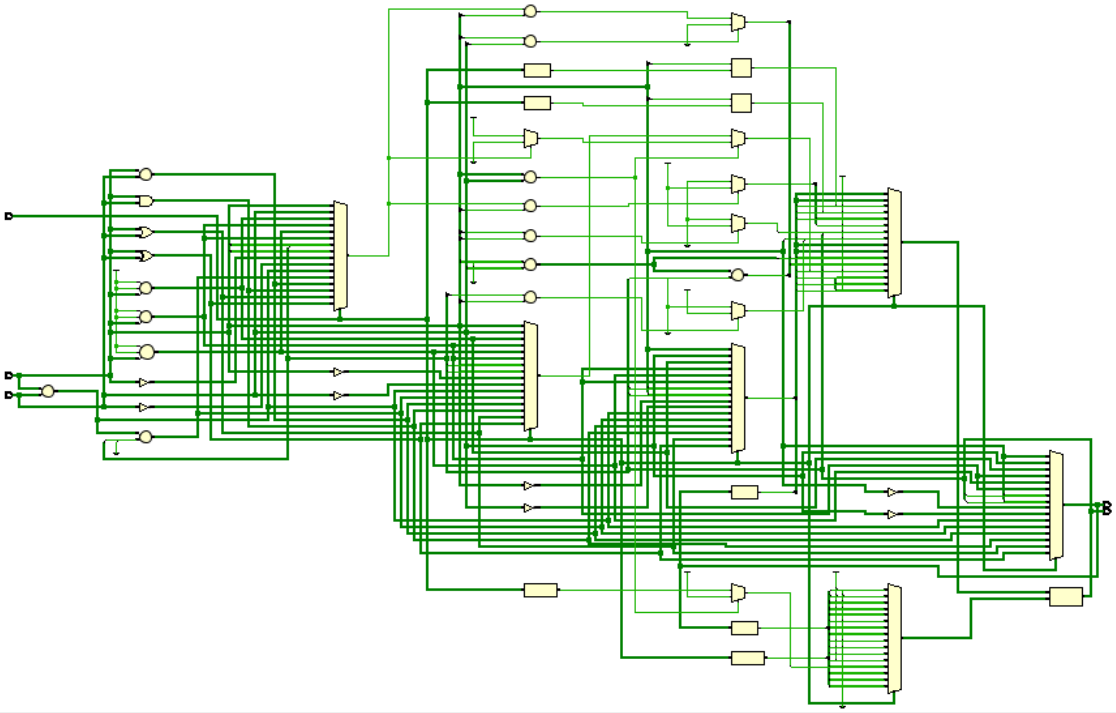

Figure 7: Elaborated Design of Arithmetic and Logical Unit[]

2.4 PART 4

In this part, we have implemented the memory module that you have given us and connected into Instruction Register. From there with the L/H signal, we got an output which we send to Multiplexer A. This Multiplexer A determines what input goes into Register File. Multiplexer C is responsible for which of the outputs goes into ALU. After necessary processes are done Output of ALU becomes the input of Memory and goes into Multiplexer B. From which Multiplexer decides that what goes into Address Register File. Output of ARF is the address which goes into Memory.

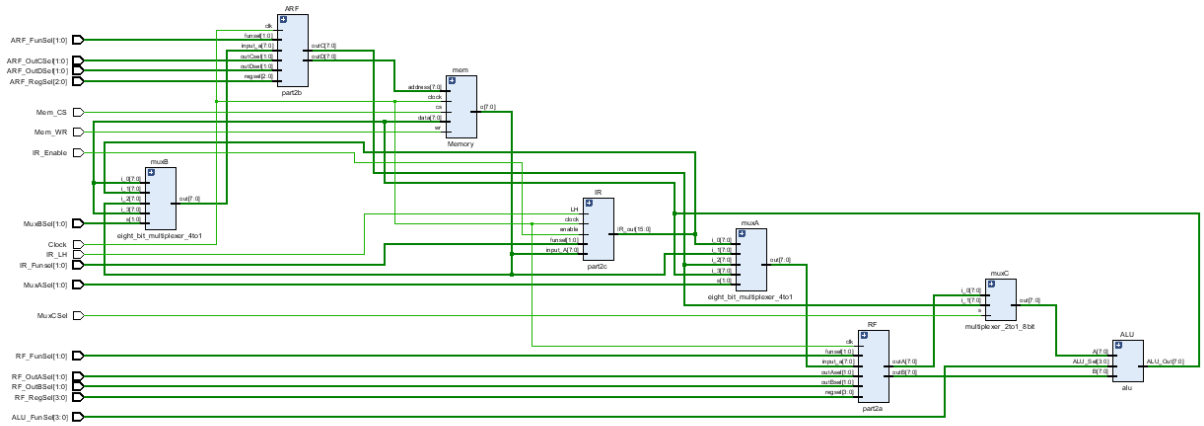

Figure 8: Elaborated Design of ALU System[]

3 RESULTS [15 points]

3.1 PART 1

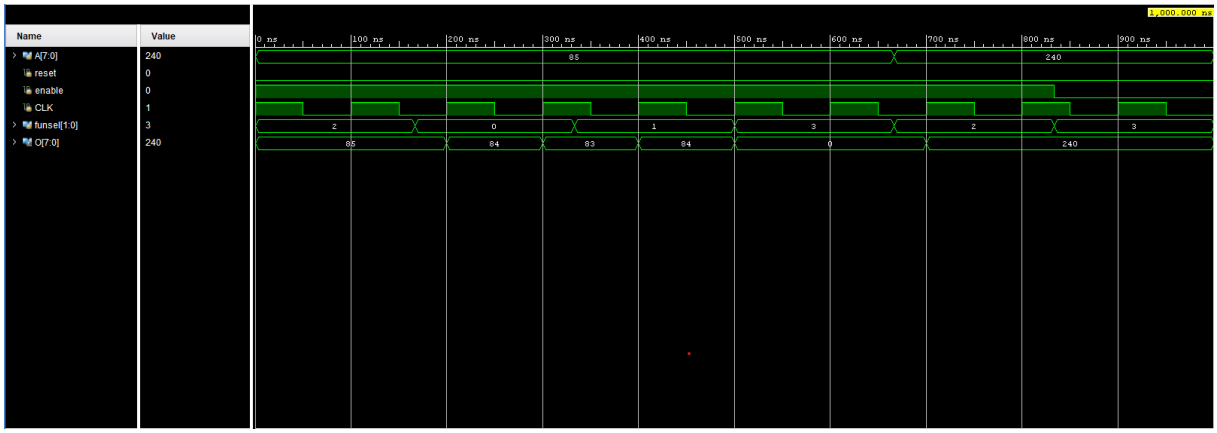

Figure 9: Simulation of 8-Bit Register[]

```
module eight_bit_register_test();
   reg [7:0] A;
   reg reset;
   reg enable;
   reg CLK;
   reg [1:0]funsel;
   wire [7:0] O;
   eight_bit_register uut(A,funsel,enable,CLK,reset,O);

   initial begin
       CLK = 0; enable=1; reset=0; A = 8'b01010101; funsel=2'b10;#166.66;//load
       funsel=2'b00; #166.66;//decrement
       funsel=2'b01; #166.66;//increment
       funsel=2'b11; #166.66;//Clear
      A = 8'b11110000; funsel=2'b10; #166.66;//load
       enable = 0; funsel=2'b11; #166.66; // Clear

   end
   always
   begin
       CLK <= ~CLK; #50;
   end

endmodule
```

Figure 10: Test bench code of 8-Bit Register[]

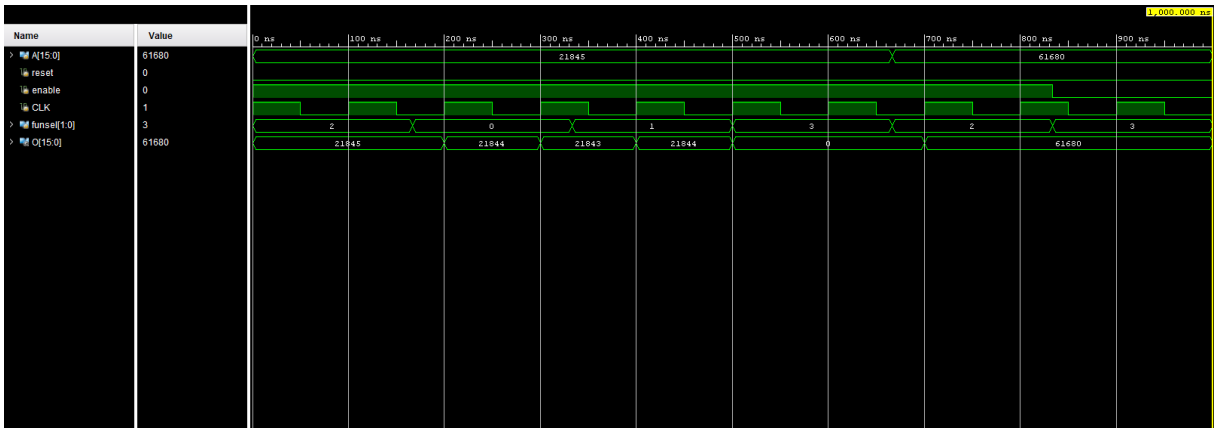

Figure 11: Simulation of 16-Bit Register[]

```
module sixteen_bit_register_test();
   reg [15:0] A;
   reg reset;
   reg enable;
   reg CLK;
   reg [1:0]funsel;
   wire [15:0] O;
   sixteen_bit_register uut(A,funsel,enable,CLK,reset,O);

   initial begin
       CLK = 0; enable=1; reset=0; A = 16'b0101010101010101; funsel=2'b10;#166.66;//load
       funsel=2'b00; #166.66;//decrement
       funsel=2'b01; #166.66;//increment
       funsel=2'b11; #166.66;//Clear
      A = 16'b1111000011110000; funsel=2'b10; #166.66;//load
       enable = 0; funsel=2'b11; #166.66; // Clear

   end
   always
   begin
       CLK <= ~CLK; #50;
   end
endmodule
```

Figure 12: Test bench code of 16-Bit Register[]

3.2 PART 2

Figure 13: Simulation of Regular Register File[]

```
module part2a_test();
    reg [7:0] A;
    reg [1:0]funsel;
    reg [3:0]regsel;
    reg [1:0]outAsel;
    reg [1:0]outBsel;
    reg CLK;
    wire [7:0] OA;
    wire [7:0] OB;
    part2a uut(A,funsel,regsel,outAsel,outBsel,CLK,OA,OB);

    initial begin
        CLK = 0; regsel=4'b0110; A = 8'b01010101; funsel=2'b10; outAsel=2'b11 ; outBsel=2'b00;#166.66;//R1 AND R4 --> LOAD // OUTPUT-A R4 OUTPUT-B R1
        funsel=2'b00;regsel=4'b1110;outAsel=2'b11 ; outBsel=2'b00; #166.66;// DECREMENT R4 // OUTPUT-A R4 OUTPUT-B R1
        funsel=2'b11; regsel=4'b1110; outAsel=2'b11 ; outBsel=2'b10; #166.66;// CLEAR R4 // OUTPUT-A R4 OUTPUT-B R3
        funsel=2'b10;regsel=4'b1001;A = 8'b11001100;outAsel=2'b01 ; outBsel=2'b10; #166.66;// LOAD R2 AND R3 // OUTPUT-A R2 OUTPUT-B R3
        funsel=2'b01;regsel=4'b1001;outAsel=2'b10 ; outBsel=2'b01; #166.66;// INCREMENT R2 AND R3 // OUTPUT-A R3 OUTPUT-B R2
        funsel=2'b11;regsel=4'b0000;outAsel=2'b11 ; outBsel=2'b10; #166.66;// CLEAR ALL // OUTPUT-A R4, OUTPUT-B R3

    end
    always
    begin
        CLK <= ~CLK; #50;
    end

endmodule
```

Figure 14: Test bench code of Regular Register File[]

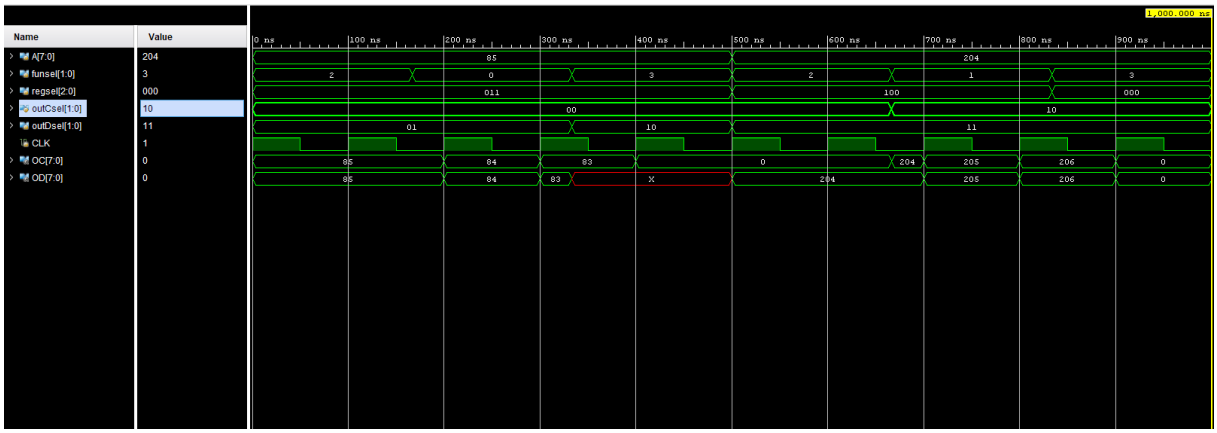

Figure 15: Simulation of Address Register File[]

```
module part2b_test();
   reg [7:0] A;
   reg [1:0]funsel;
   reg [2:0]regsel;
   reg [1:0]outCsel;
   reg [1:0]outDsel;
   reg CLK;
   wire [7:0] OC;
   wire [7:0] OD;
   part2b uut(A,funsel,regsel,outCsel,outDsel,CLK,OC,OD);

   initial begin
       CLK = 0; regsel=3'b011; A = 8'b01010101; funsel=2'b10; outCsel=2'b00 ; outDsel=2'b01;#166.66;// LOAD PC // OUTPUT-C PC // OUTPUT-D PC
       funsel=2'b00;regsel=3'b011;outCsel=2'b00 ; outDsel=2'b01; #166.66;// DECREMENT PC // OUTPUT-C PC // OUTPUT-D PC
       funsel=2'b11; regsel=3'b011; outCsel=2'b00 ; outDsel=2'b10;#166.66; // CLEAR PC // OUTPUT-C PC // OUTPUT-D AR
       funsel=2'b10;regsel=3'b100;A = 8'b11001100;outCsel=2'b00 ; outDsel=2'b11; #166.66; // LOAD AR AND SP // OUTPUT-C PC // OUTPUT-D SP
       funsel=2'b01;regsel=3'b100;outCsel=2'b10 ; outDsel=2'b11; #166.66;// INCREMENT AR AND SP // OUTPUT-C AR // OUTPUT-D SP
       funsel=2'b11;regsel=3'b000;outCsel=2'b10 ; outDsel=2'b11; #166.66;// CLEAR ALL // OUTPUT-C AR // OUTPUT-D SP

   end
   always
   begin
       CLK <= ~CLK; #50;
   end

endmodule
```

Figure 16: Test bench code of Address Regular Register File[]

Figure 17: Simulation of 16-bit Instruction Register[]

```
module part2c_test();
   reg [7:0] A;
   reg LH;
   reg enable;
   reg CLK;
   reg [1:0]funsel;
   wire [15:0] O;
   part2c uut(A,LH,funsel,CLK,enable,O);

   initial begin
       enable=1;LH=1'b1; A = 8'b00000001; funsel=2'b10;#5;CLK = 0;#166.66;// Load MSB
       LH=1'b0; enable=1; A = 8'b00000001; funsel=2'b10;#166.66;// Load LSB
       LH=1'b0; enable=1; A = 8'b11111111; funsel=2'b10;#166.66;// Load LSB
       funsel=2'b00; #166.66;// Decrement
       funsel=2'b01; #166.66;// Increment
       funsel=2'b11; #166.66;// Clear

   end
   always
   begin
       CLK <= ~CLK; #50;
   end

endmodule
```

Figure 18: Test bench code of Instruction Register[]

3.3 PART 3

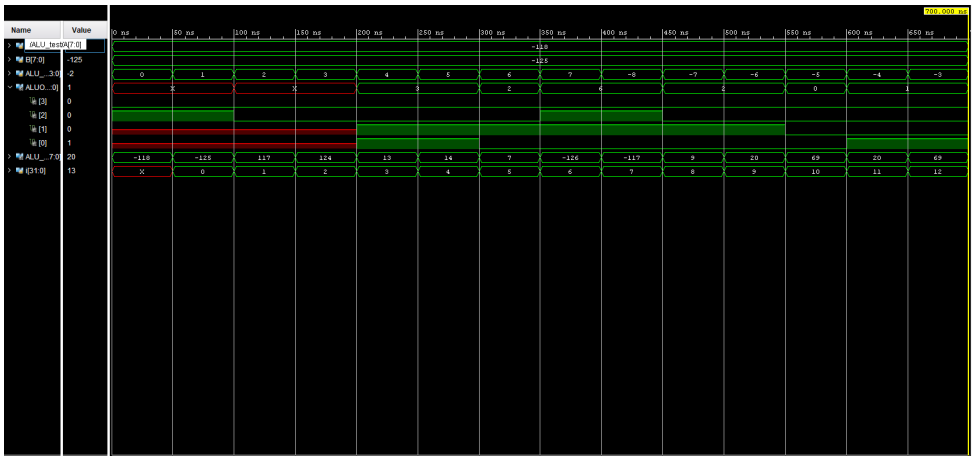

Figure 19: Simulation of Arithmetic and Logical Unit for signed numbers[]

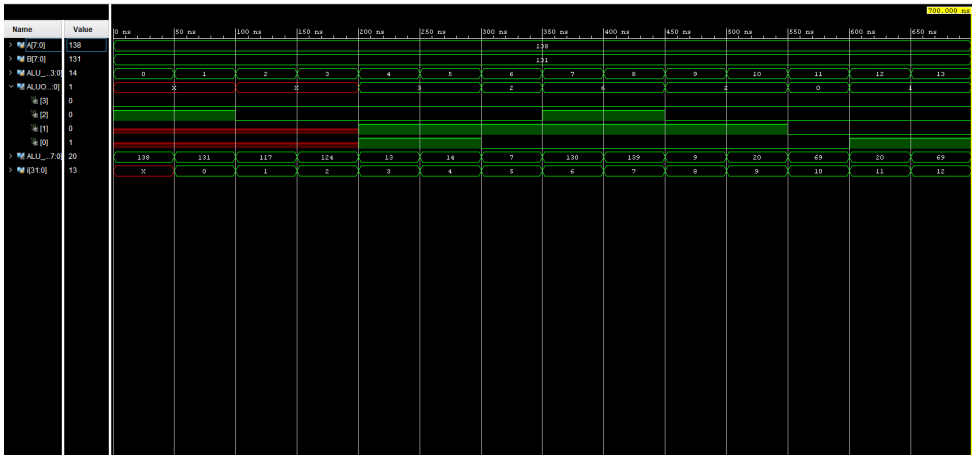

Figure 20: Simulation of Arithmetic and Logical Unit for unsigned numbers[]

```
module ALU_test();
//Inputs
 reg[7:0] A,B;
 reg[3:0] ALU_Sel;
//Outputs
 wire [3:0]ALUOutFlag;
 wire[7:0] ALU_Out;
 integer i;
 alu uut(
            A,B,  // ALU 8-bit Inputs
            ALU_Sel,// ALU Selection
            ALU_Out, // ALU 8-bit Output
            ALUOutFlag // ALU Output Flags (4 bit)
     );
    initial begin
    // hold reset state for 100 ns.
      A = 8'b10001010;
      B = 8'b10000011;
      ALU_Sel = 4'h0;
      #50;
      for (i=0;i<=15;i=i+1)
      begin
       ALU_Sel = ALU_Sel + 4'b0001;
       #50;
      end

      A = 8'hF6;
      B = 8'h0A;

    end
endmodule
```

Figure 21: Test bench code of Arithmetic and Logical Unit[]

3.4 PART 4

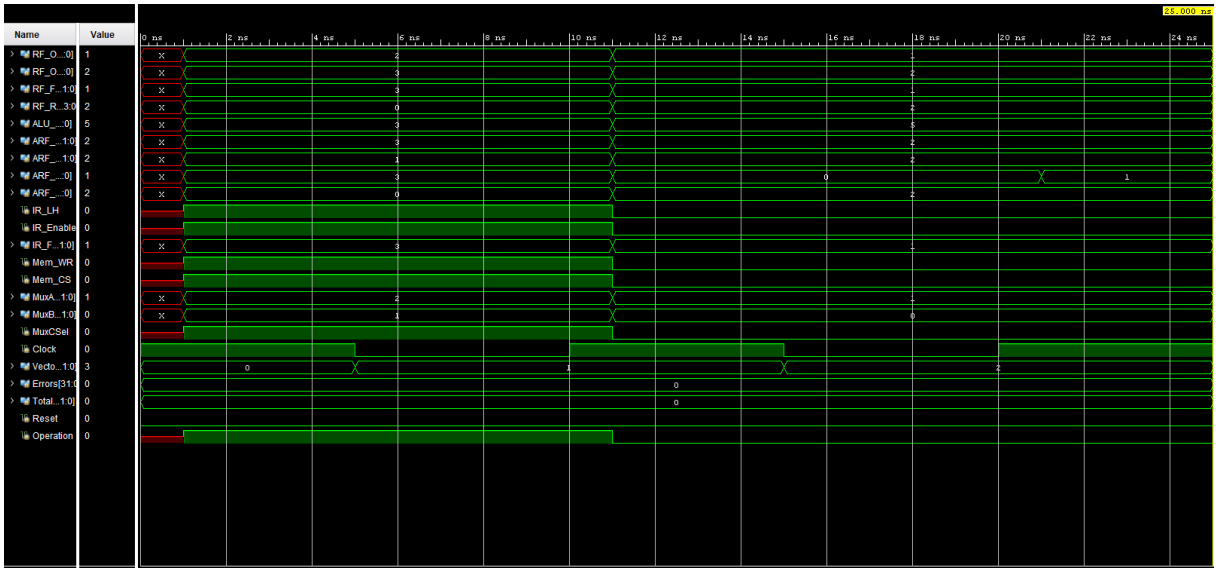

Figure 22: Simulation of ALU System[]

```
# run 1000ns
Input Values:
Operation: 1
Register File: OutASel: 10, OutBSel: 11, FunSel: 11, Regsel: 0000
ALU FunSel: 0011
Addres Register File: OutCSel: 11, OutDSel: 01, FunSel: 11, Regsel: 000
Instruction Register: LH: 1, Enable: 1, FunSel: 11
Memory: WR: 1, CS: 1
MuxASel: 10, MuxBSel: 01, MuxCSel: 1
Ouput Values:
Register File: AOut: xxxxxxxx, BOut: xxxxxxxx
ALUOut: xxxxxxxx, ALUOutFlag: 0x0x, ALUOutFlags: Z:0, C:x, N:0, O:x,
Address Register File: COut: xxxxxxxx, DOut (Address): xxxxxxxx
Memory Out: zzzzzzzz
Instruction Register: IROut: xxxxxxxxxxxxxxxx
MuxAOut: xxxxxxxx, MuxBOut: xxxxxxxx, MuxCOut: xxxxxxxx

Input Values:
Operation: 0
Register File: OutASel: 01, OutBSel: 10, FunSel: 01, Regsel: 0010
ALU FunSel: 0101
Addres Register File: OutCSel: 10, OutDSel: 10, FunSel: 00, Regsel: 010
Instruction Register: LH: 0, Enable: 0, FunSel: 01
Memory: WR: 0, CS: 0
MuxASel: 01, MuxBSel: 00, MuxCSel: 0
Ouput Values:
Register File: AOut: 00000000, BOut: 00000000
ALUOut: 00000000, ALUOutFlag: 1000, ALUOutFlags: Z:1, C:0, N:0, O:0,
Address Register File: COut: 00000000, DOut (Address): 00000000
Memory Out: 00000000
Instruction Register: IROut: 0000000000000000
MuxAOut: 00000000, MuxBOut: 00000000, MuxCOut: 00000000

Input Values:
Operation: 0
Register File: OutASel: 01, OutBSel: 10, FunSel: 01, Regsel: 0010
ALU FunSel: 0101
Addres Register File: OutCSel: 10, OutDSel: 10, FunSel: 01, Regsel: 010
Instruction Register: LH: 0, Enable: 0, FunSel: 01
Memory: WR: 0, CS: 0
MuxASel: 01, MuxBSel: 00, MuxCSel: 0
Ouput Values:
Register File: AOut: 00000001, BOut: 00000000
ALUOut: 00000001, ALUOutFlag: 0000, ALUOutFlags: Z:0, C:0, N:0, O:0,
Address Register File: COut: 00000000, DOut (Address): 00000000
Memory Out: 00000000
Instruction Register: IROut: 0000000000000000
MuxAOut: 00000000, MuxBOut: 00000001, MuxCOut: 00000001

         3 tests completed.
$finish called at time : 25 ns : File "A:/projec_1/projec_1/projec_1.srcs/sim_1/new/project_1_test.v" Line 121
xsim: Time (s): cpu = 00:00:06 ; elapsed = 00:00:05 . Memory (MB): peak = 935.035 ; gain = 1.184
INFO: [USF-XSim-96] XSim completed. Design snapshot 'Project1Test_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
```

Figure 23: TLC Console Output of the original case[]

We moderated the testbench.mem in order to make sure our design works as expected.

```
# run 1000ns
Input Values:
Operation: 1
Register File: OutASel: 10, OutBSel: 11, FunSel: 11, Regsel: 0000
ALU FunSel: 0011
Addres Register File: OutCSel: 11, OutDSel: 01, FunSel: 11, Regsel: 000
Instruction Register: LH: 1, Enable: 1, FunSel: 11
Memory: WR: 1, CS: 1
MuxASel: 10, MuxBSel: 01, MuxCSel: 1
Ouput Values:
Register File: AOut: xxxxxxxx, BOut: xxxxxxxx
ALUOut: xxxxxxxx, ALUOutFlag: 0x0x, ALUOutFlags: Z:0, C:x, N:0, O:x,
Address Register File: COut: xxxxxxxx, DOut (Address): xxxxxxxx
Memory Out: zzzzzzzz
Instruction Register: IROut: xxxxxxxxxxxxxxxx
MuxAOut: xxxxxxxx, MuxBOut: xxxxxxxx, MuxCOut: xxxxxxxx

Input Values:
Operation: 1
Register File: OutASel: 10, OutBSel: 11, FunSel: 01, Regsel: 1100
ALU FunSel: 0100
Addres Register File: OutCSel: 11, OutDSel: 01, FunSel: 01, Regsel: 010
Instruction Register: LH: 1, Enable: 1, FunSel: 11
Memory: WR: 0, CS: 0
MuxASel: 11, MuxBSel: 10, MuxCSel: 1
Ouput Values:
Register File: AOut: 00000000, BOut: 00000000
ALUOut: 00000000, ALUOutFlag: 1000, ALUOutFlags: Z:1, C:0, N:0, O:0,
Address Register File: COut: 00000000, DOut (Address): 00000000
Memory Out: 00000000
Instruction Register: IROut: 0000000000000000
MuxAOut: 00000000, MuxBOut: 00000000, MuxCOut: 00000000

Input Values:
Operation: 1
Register File: OutASel: 10, OutBSel: 11, FunSel: 01, Regsel: 1100
ALU FunSel: 0100
Addres Register File: OutCSel: 11, OutDSel: 01, FunSel: 01, Regsel: 010
Instruction Register: LH: 1, Enable: 1, FunSel: 11
Memory: WR: 0, CS: 0
MuxASel: 11, MuxBSel: 01, MuxCSel: 1
Ouput Values:
Register File: AOut: 00000001, BOut: 00000001
ALUOut: 00000010, ALUOutFlag: 0000, ALUOutFlags: Z:0, C:0, N:0, O:0,
Address Register File: COut: 00000001, DOut (Address): 00000001
Memory Out: 00000000
Instruction Register: IROut: 0000000000000000
MuxAOut: 00000010, MuxBOut: 00000000, MuxCOut: 00000001
```

Figure 24: TLC Console Output of our case - p.1[]

```
Input Values:
Operation: 1
Register File: OutASel: 10, OutBSel: 11, FunSel: 01, Regsel: 1100
ALU FunSel: 0100
Addres Register File: OutCSel: 11, OutDSel: 01, FunSel: 11, Regsel: 0010
Instruction Register: LH: 1, Enable: 1, FunSel: 11
Memory: WR: 0, CS: 0
MuxASel: 11, MuxBSel: 10, MuxCSel: 0
Ouput Values:
Register File: AOut: 00000010, BOut: 00000010
ALUOut: 00000100, ALUOutFlag: 0000, ALUOutFlags: Z:0, C:0, N:0, O:0,
Address Register File: COut: 00000010, DOut (Address): 00000010
Memory Out: 00000000
Instruction Register: IROut: 0000000000000000
MuxAOut: 00000100, MuxBOut: 00000000, MuxCOut: 00000010

Input Values:
Operation: 1
Register File: OutASel: 10, OutBSel: 11, FunSel: 01, Regsel: 1100
ALU FunSel: 0100
Addres Register File: OutCSel: 11, OutDSel: 01, FunSel: 01, Regsel: 0010
Instruction Register: LH: 1, Enable: 1, FunSel: 11
Memory: WR: 0, CS: 0
MuxASel: 11, MuxBSel: 01, MuxCSel: 1
Ouput Values:
Register File: AOut: 00000011, BOut: 00000011
ALUOut: 00000011, ALUOutFlag: 0000, ALUOutFlags: Z:0, C:0, N:0, O:0,
Address Register File: COut: 00000000, DOut (Address): 00000000
Memory Out: 00000000
Instruction Register: IROut: 0000000000000000
MuxAOut: 00000011, MuxBOut: 00000000, MuxCOut: 00000000

Input Values:
Operation: 1
Register File: OutASel: 10, OutBSel: 11, FunSel: 01, Regsel: 1100
ALU FunSel: 0100
Addres Register File: OutCSel: 11, OutDSel: 01, FunSel: 01, Regsel: 0010
Instruction Register: LH: 1, Enable: 1, FunSel: 11
Memory: WR: 0, CS: 0
MuxASel: 11, MuxBSel: 10, MuxCSel: 1
Ouput Values:
Register File: AOut: 00000100, BOut: 00000100
ALUOut: 00000101, ALUOutFlag: 0000, ALUOutFlags: Z:0, C:0, N:0, O:0,
Address Register File: COut: 00000001, DOut (Address): 00000001
Memory Out: 00000000
Instruction Register: IROut: 0000000000000000
MuxAOut: 00000101, MuxBOut: 00000000, MuxCOut: 00000001

         6 tests completed.
$finish called at time : 55 ns : File "A:/projec_1/projec_1/projec_1.srcs/sim_1/new/project_1_test.v" Line 121
INFO: [USF-XSim-96] XSim completed. Design snapshot 'Project1Test_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
```

Figure 25: TLC Console Output of our case - p.2[]

4 DISCUSSION [25 points]

4.1 PART 1

For the first part we designed 8-bit and 16-bit registers with enable and FunSel inputs. These registers have different functions such as increment, decrement, load and clear. To implements these functions we used d-flip flips, multiplexer and a adder-subtractor. It was hard to implement all modules to design the registers.

4.2 PART 2

For the second part we designed a register file, address register file and instruction register(IR). Register file has 4 register (R1,R2,R3,R4) and every register has 4 functions such as increment, decrement, load and clear. For the address register file, we did the same implementations in Register File but we named it differently. For the last part we had difficulties while we were implementing the IR. It was hard to load and assign 8-bits input to proper places of 16-bits.

4.3 PART 3

In part3 we have designed an Arithmetic Logic Unit (ALU) that has two 8-bit inputs and an 8-bit output. In order to Implement Arithmetic and Logical Unit, we have used the Behavioral modeling approach. At the beginning of the design, we tried to design with structural modeling such as logic gates but we had some difficulties. Therefore we decided to make it with the behavioral model. The hardest part of the implementation were if else statement blocks. It was complicated to understand and prone to error. Furthermore, we had difficulties while we were setting the flags.

4.4 PART 4

For the final part, at first it was easy to implement the modules because we have designed the previous modules well and organized. After that we had trouble splitting the IR module's 16 bit output to two 8-bits but we managed to solve that problem and the implementation part passed. Our first simulation test we had a problem with the names of the inputs and outputs so we changed them according to Project1Test module. After solving that problem we got expected Outputs in the TCL Console.

5 CONCLUSION [10 points]

In conclusion, we learned to build a basic computer by designing the necessary modules and combining them. At the beginning of the experiment,it was hard to understand the concept of the project but when we get the final part we understand what all of these modules were about. At the first part we design registers. For the second part we used these register to design Register File(RF) and Address Register File (ARF). At the third part of the experiment we design an Arithmetic Logic Unit (ALU) that has two 8-bit inputs and an 8-bit output. At the last part we implement a basic computer system and we used all modules that we designed in the previous parts.