ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E

COMPUTER ORGANIZATION PROJECT REPORT

PROJECT NO	:	1
DUE DATE	:	11.05.2022
GROUP NO	:	G15

GROUP MEMBERS:

150180073	:	SEYFÜLMÜLÜK KUTLUK
150210734	:	AHMET BARIŞ EMRE DURAK
150180118	:	BURAK ENGİN AŞIKLAR

1 PROJECT PARTS

1.1 PART 1

1.1.1 PART 1A

For the first part of part A, an 8-bit register has been created. This register has different functionalities depending on the input given in FunSel, and checks whether enable is on in order to execute the operation.

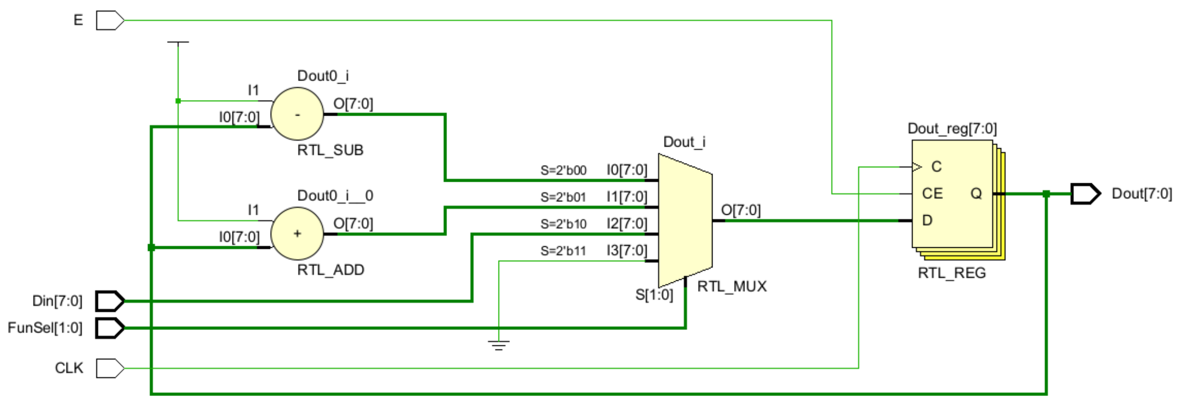
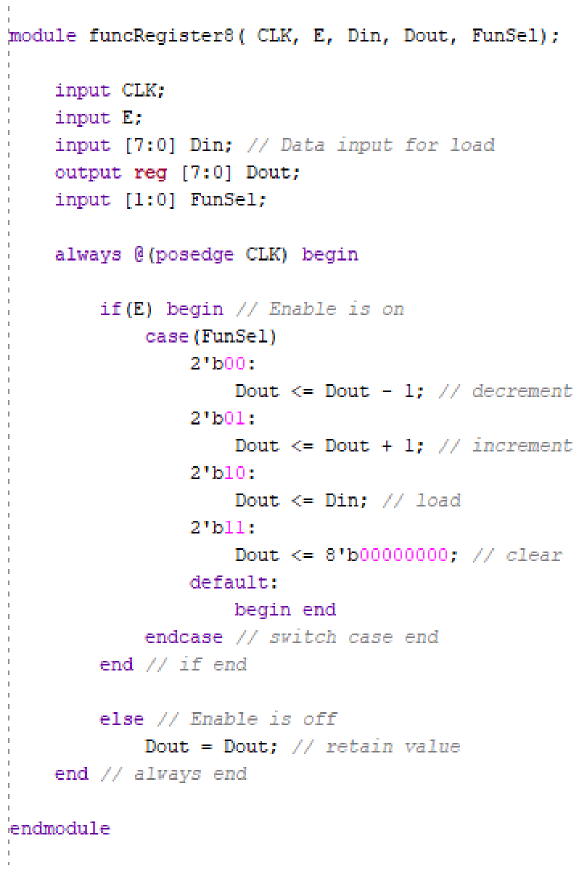


Figure 1: 8-bit register's design



```
module funcRegister8( CLK, E, Din, Dout, FunSel);

    input CLK;
    input E;
    input [7:0] Din; // Data input for load
    output reg [7:0] Dout;
    input [1:0] FunSel;

    always @(posedge CLK) begin

        if(E) begin // Enable is on
            case(FunSel)
                2'b00:
                    Dout <= Dout - 1; // decrement
                2'b01:
                    Dout <= Dout + 1; // increment
                2'b10:
                    Dout <= Din; // load
                2'b11:
                    Dout <= 8'b00000000; // clear
                default:
                    begin end
            endcase // switch case end
        end // if end

        else // Enable is off
            Dout = Dout; // retain value
    end // always end

endmodule
```

Figure 2: Code of the 8-bit register

CLK input is for the clock, E is the enable, Din is the data input that's going to be used for load, and lastly FunSel is to select operations. **1.1.2 PART 1B**

Similar to the earlier part, a register with multiple functionalities has been created, with the difference being input and output having 16-bits.

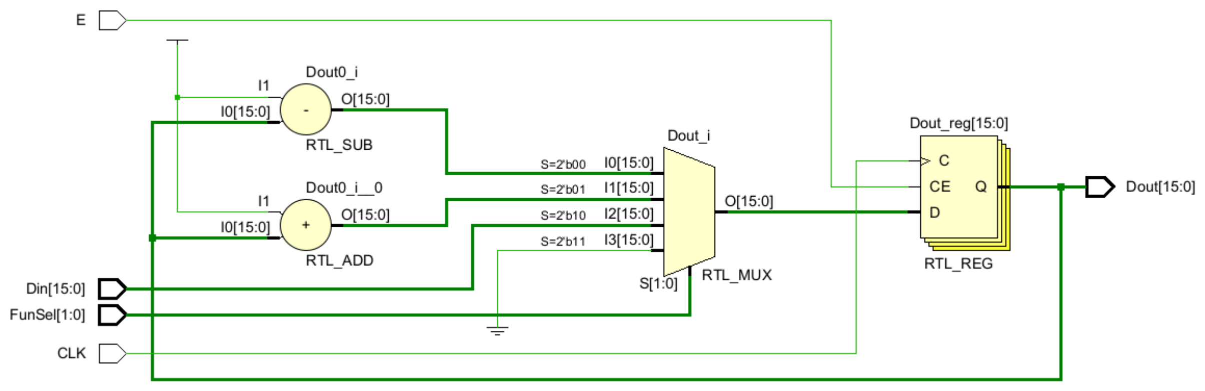
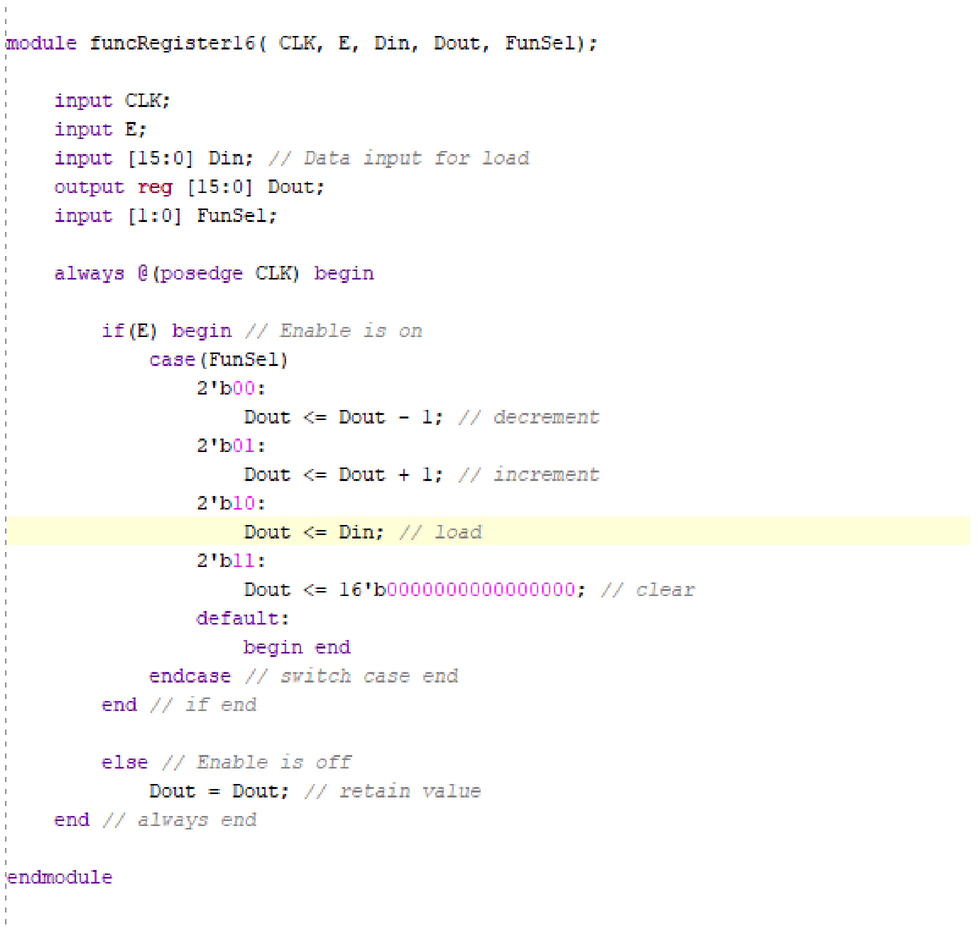


Figure 3: 16-bit register's design



```
module funcRegister16( CLK, E, Din, Dout, FunSel);

    input CLK;
    input E;
    input [15:0] Din; // Data input for load
    output reg [15:0] Dout;
    input [1:0] FunSel;

    always @(posedge CLK) begin

        if(E) begin // Enable is on
            case(FunSel)
                2'b00:
                    Dout <= Dout - 1; // decrement
                2'b01:
                    Dout <= Dout + 1; // increment
                2'b10:
                    Dout <= Din; // load
                2'b11:
                    Dout <= 16'b0000000000000000; // clear
                default:
                    begin end
            endcase // switch case end
        end // if end

        else // Enable is off
            Dout = Dout; // retain value
    end // always end

endmodule
```

Figure 4: Code of the 16-bit register

Pretty much the same inputs as part 1A, only Din is now 16-bit instead of 8.

1.2 PART 2

1.2.1 PART 2A

In this part, utilizing the registers created earlier, a register file, which is essentially a structure that contains many registers, will be created. For this part, the following parts were used:

• Multiplexer: It is used in order to select from the data that is given by the registers, the multiplexer selects according to the OutA and OutB inputs for their respective multiplexers.

• 8-bit Register: It is used for simple functions and operations with the given input, similar to the one made in part 1.

• Splitter: It is used to seperate the 4-bit RegSel input into 4 1-bit parts to send into each register's enable input.

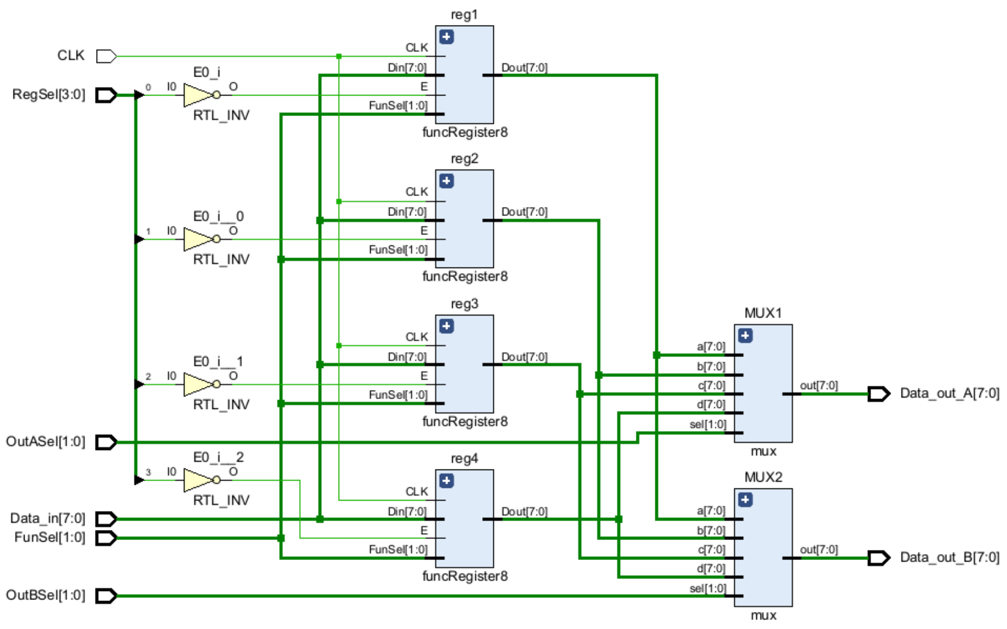
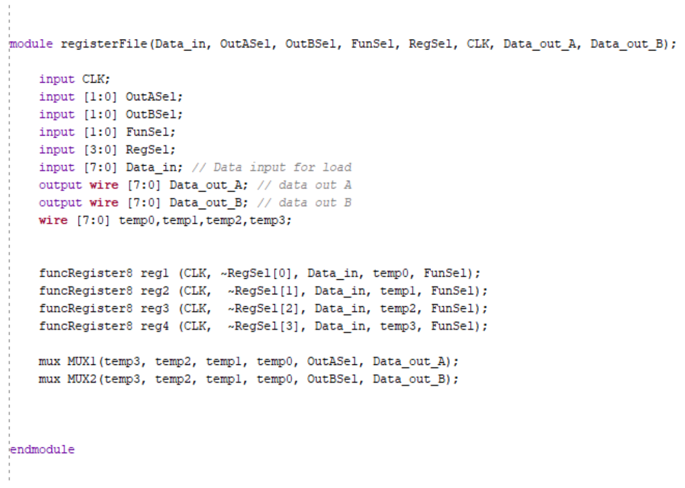


Figure 5: Design of the register file



```
module registerFile(Data_in, OutASel, OutBSel, FunSel, RegSel, CLK, Data_out_A, Data_out_B);

    input CLK;
    input [1:0] OutASel;
    input [1:0] OutBSel;
    input [1:0] FunSel;
    input [3:0] RegSel;
    input [7:0] Data_in; // Data input for load
    output wire [7:0] Data_out_A; // data out A
    output wire [7:0] Data_out_B; // data out B
    wire [7:0] temp0,temp1,temp2,temp3;


    funcRegister8 reg1 (CLK, ~RegSel[0], Data_in, temp0, FunSel);
    funcRegister8 reg2 (CLK,  ~RegSel[1], Data_in, temp1, FunSel);
    funcRegister8 reg3 (CLK,  ~RegSel[2], Data_in, temp2, FunSel);
    funcRegister8 reg4 (CLK,  ~RegSel[3], Data_in, temp3, FunSel);

    mux MUX1(temp3, temp2, temp1, temp0, OutASel, Data_out_A);
    mux MUX2(temp3, temp2, temp1, temp0, OutBSel, Data_out_B);



endmodule
```

Figure 6: Code of the register file

CLK is for the clock, OutASel and OutBSel are for selecting whether the output will be given by the multiplexer, FunSel has the same purpose as it had in the previous part, RegSel is for deciding which Registers will be active and lastly Data$_i nisforgettingtheinputtoload.$

1.2.2 PART 2B

Building up on the design that was created in part 2A, the file register is now modified to be a address register file. It has a program counter (PC), an address register (AR), and a stack pointer (SP) and one less register compared to previous part. Other than that there is no notable difference.

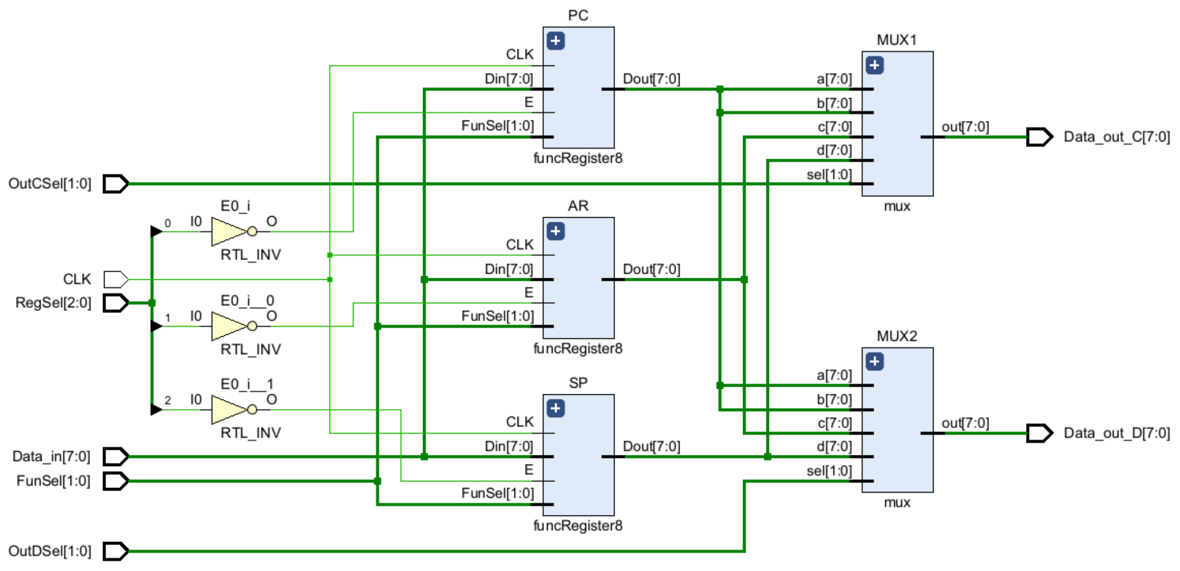
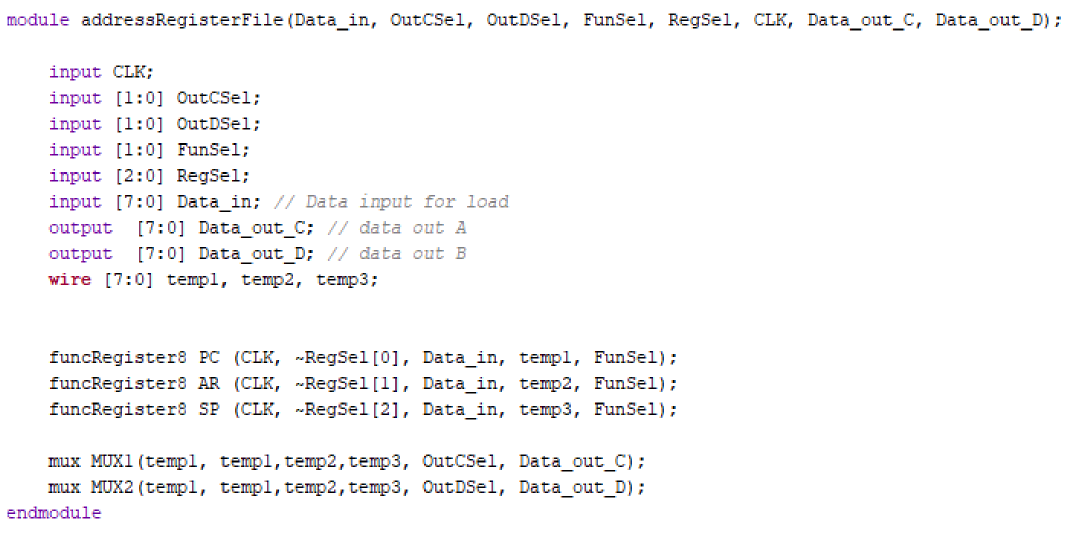


Figure 7: Design of address register file



```
module addressRegisterFile(Data_in, OutCSel, OutDSel, FunSel, RegSel, CLK, Data_out_C, Data_out_D);

    input CLK;
    input [1:0] OutCSel;
    input [1:0] OutDSel;
    input [1:0] FunSel;
    input [2:0] RegSel;
    input [7:0] Data_in; // Data input for load
    output  [7:0] Data_out_C; // data out A
    output  [7:0] Data_out_D; // data out B
    wire [7:0] temp1, temp2, temp3;


    funcRegister8 PC (CLK, ~RegSel[0], Data_in, temp1, FunSel);
    funcRegister8 AR (CLK, ~RegSel[1], Data_in, temp2, FunSel);
    funcRegister8 SP (CLK, ~RegSel[2], Data_in, temp3, FunSel);

    mux MUX1(temp1, temp1,temp2,temp3, OutCSel, Data_out_C);
    mux MUX2(temp1, temp1,temp2,temp3, OutDSel, Data_out_D);
endmodule
```

Figure 8: Code of address register file

Pretty much the same inputs as the previous part.

1.2.3 PART 2C

For this part an 16-bit IR register has been designed, purpose of this register is to store an 8-bit input on the front or back half of a 16-bit bit register. In addition to this the 16-bit register also has to retain it's functionality and must be able to load 8-bit parts from its 16-bit memory.

A 16-bit register is connected to multiple multiplexers in order to manage the operations. LowHigh signal is used to determine whether the multiplexer makes the input the front half or the back half of the output.

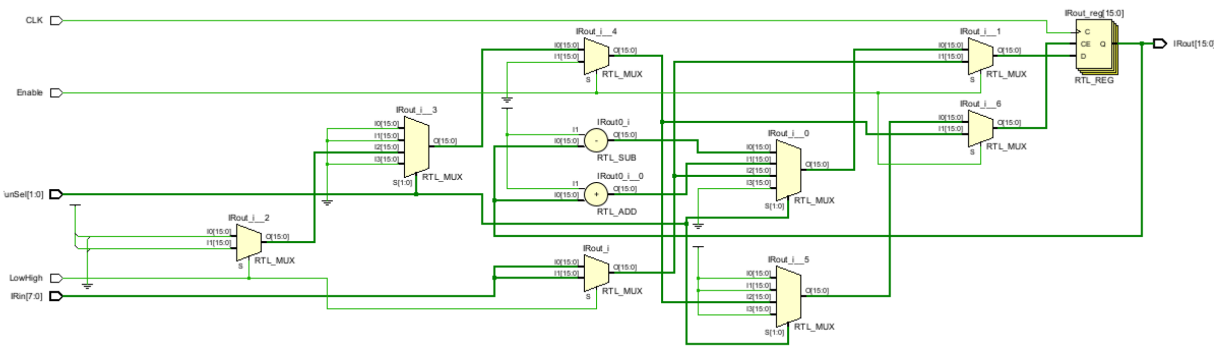
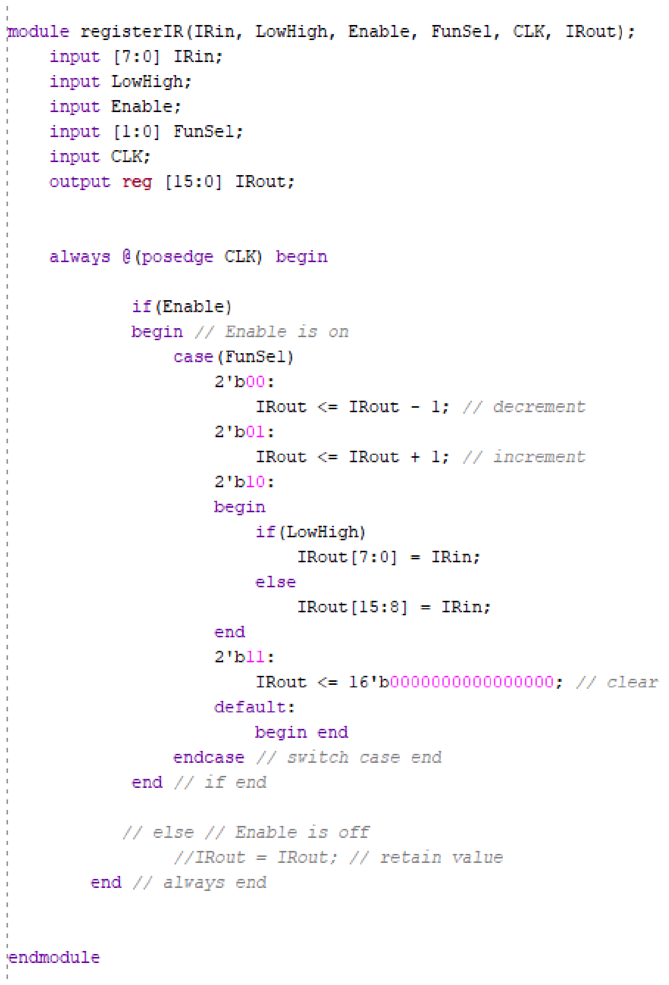


Figure 9: Design of IR register



```
module registerIR(IRin, LowHigh, Enable, FunSel, CLK, IRout);
    input [7:0] IRin;
    input LowHigh;
    input Enable;
    input [1:0] FunSel;
    input CLK;
    output reg [15:0] IRout;


    always @(posedge CLK) begin

            if(Enable)
            begin // Enable is on
                case(FunSel)
                    2'b00:
                        IRout <= IRout - 1; // decrement
                    2'b01:
                        IRout <= IRout + 1; // increment
                    2'b10:
                    begin
                        if(LowHigh)
                            IRout[7:0] = IRin;
                        else
                            IRout[15:8] = IRin;
                    end
                    2'b11:
                        IRout <= 16'b0000000000000000; // clear
                    default:
                        begin end
                endcase // switch case end
            end // if end

           // else // Enable is off
                //IRout = IRout; // retain value
        end // always end


endmodule
```

Figure 10: Code of IR register

CLK input is for the clock, Enable is the enable signal, IRin is the data input that's going to be used for load, FunSel is to select operations, LowHigh is for selecting whether the 8-bit data will be written to the first or second half.

1.3 PART 3

For this part an ALU has been designed, this module takes two 8-bit inputs and applies an operation depending on FunSel which gives the output as an 8-bit data. While the operations commence, the ZCNO flags can trigger to signify certain situations happening.

The parts that were used in this module are as follows:

- Multiplexer: It is used to choose instructions according to FunSel input.

- ADDER : It is used to construct A+B instruction.

- SUBTRACTER : It is used to construct A-B instruction.

- NOT : It is usedto construct Not A, Not B instructions.

- AND : It is used to construct A.B instruction.
- OR : It is used to construct A+B instruction.
- XOR : It is used to construct AB instruction.
- Logical Left : It is used to construct lsl A, asl A instructions.
- Logical Right : It is used to construct lsr A instruction.
- Logical Left : It is used to construct lsl A instruction.
- Arithmetic Right : It is used to construct asr A instruction.
- Rotate left : It is used to construct csl A instruction.
- Rotate Right : It is used to construct csr A instruction. ://www.overleaf.com/project/60a4eafd654c9df

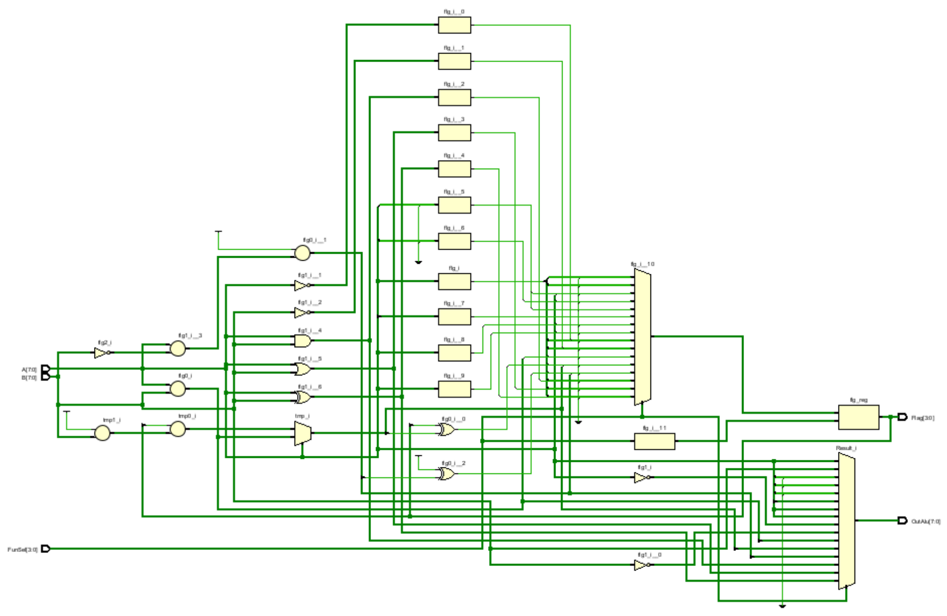
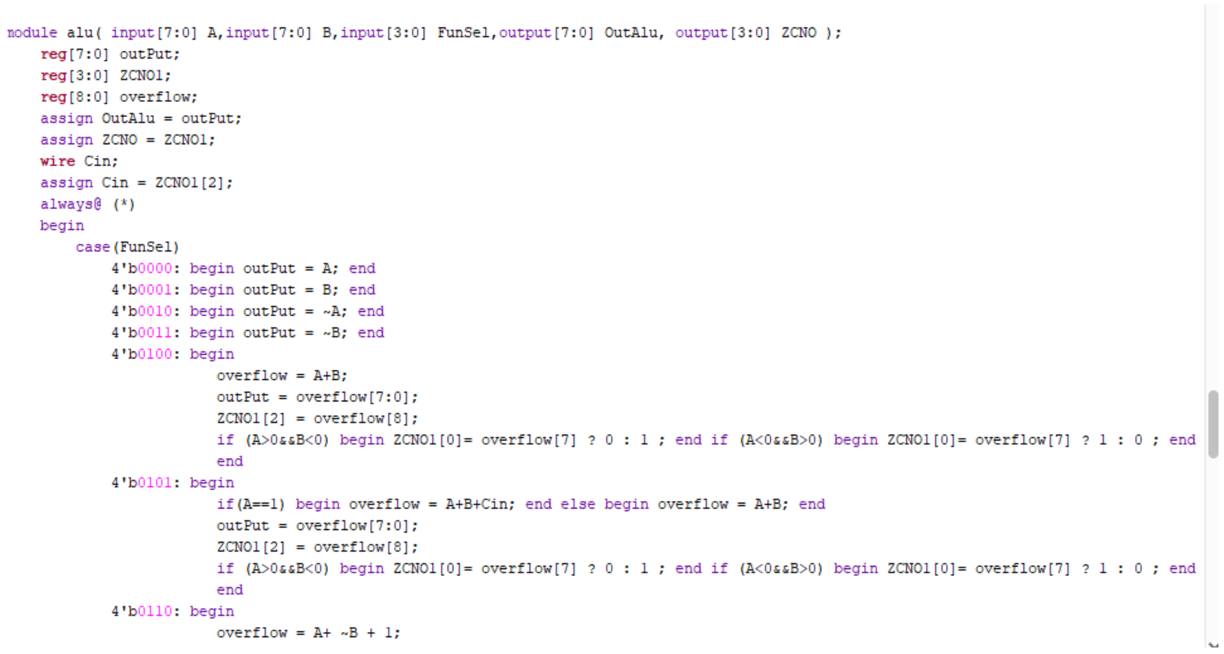


Figure 11: PART 3



```verilog
module alu( input[7:0] A,input[7:0] B,input[3:0] FunSel,output[7:0] OutAlu, output[3:0] ZCNO );
    reg[7:0] outPut;
    reg[3:0] ZCNO1;
    reg[8:0] overflow;
    assign OutAlu = outPut;
    assign ZCNO = ZCNO1;
    wire Cin;
    assign Cin = ZCNO1[2];
    always@ (*)
    begin
        case(FunSel)
            4'b0000: begin outPut = A; end
            4'b0001: begin outPut = B; end
            4'b0010: begin outPut = ~A; end
            4'b0011: begin outPut = ~B; end
            4'b0100: begin
                        overflow = A+B;
                        outPut = overflow[7:0];
                        ZCNO1[2] = overflow[8];
                        if (A>0&&B<0) begin ZCNO1[0]= overflow[7] ? 0 : 1 ; end if (A<0&&B>0) begin ZCNO1[0]= overflow[7] ? 1 : 0 ; end
                        end
            4'b0101: begin
                        if(A==1) begin overflow = A+B+Cin; end else begin overflow = A+B; end
                        outPut = overflow[7:0];
                        ZCNO1[2] = overflow[8];
                        if (A>0&&B<0) begin ZCNO1[0]= overflow[7] ? 0 : 1 ; end if (A<0&&B>0) begin ZCNO1[0]= overflow[7] ? 1 : 0 ; end
                        end
            4'b0110: begin
                        overflow = A+ ~B + 1;
```

Figure 12: PART 3

The only relevant inputs for this module are the A and B 8-bit inputs and FunSel for selecting the operation.

1.4 PART 4

With all the pieces designed only thing left to do is to assemble them. Using the Multiplexer, Register File, Address Register File, IR Register made in the earlier parts in addition to a memory.

The parts in the system are used for the following functions:

• Multiplexer: For selection and the navigation of the data given from various sources in the system.

• Various Registers: Storing the data and accessing and using the various operations available on them whenever they are needed.

• ALU: Used for arithmetic and logic operations on the data it has been given.

• Memory Module: For accessing the data stored in it with an address.

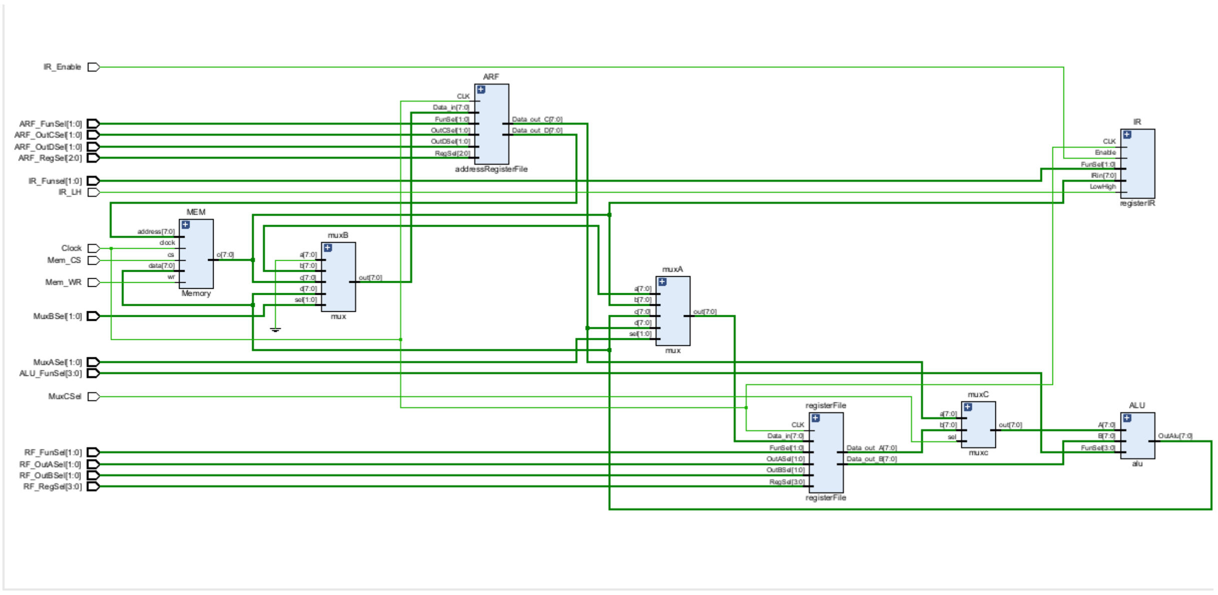
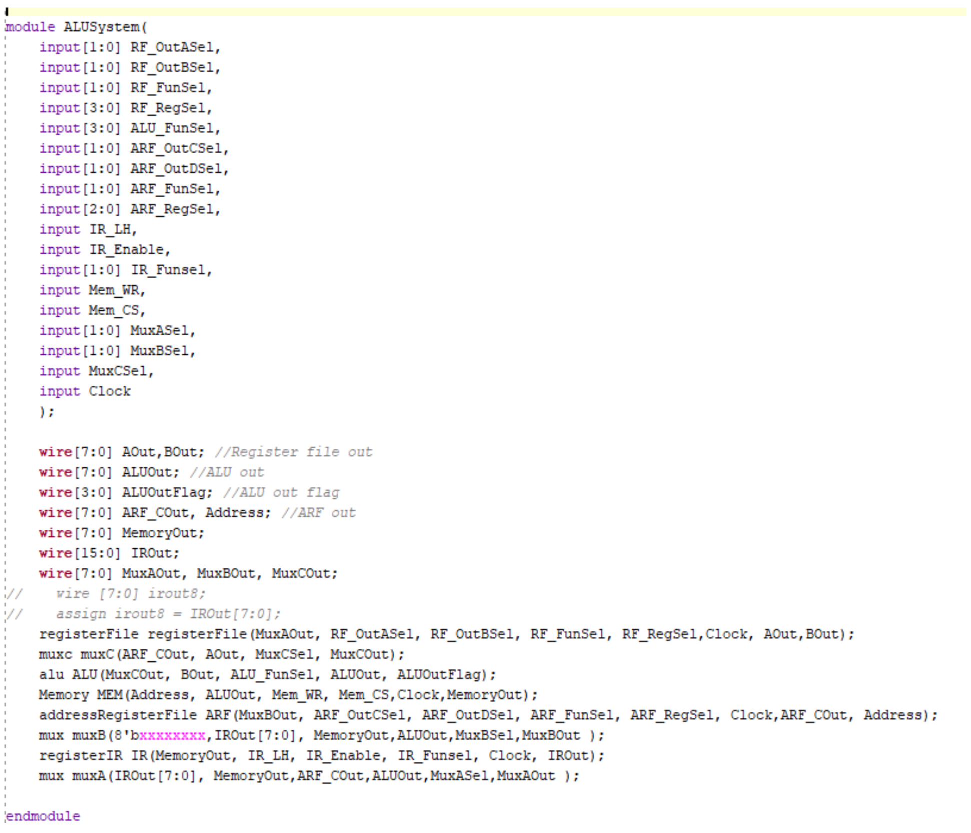


Figure 13: Design of the system

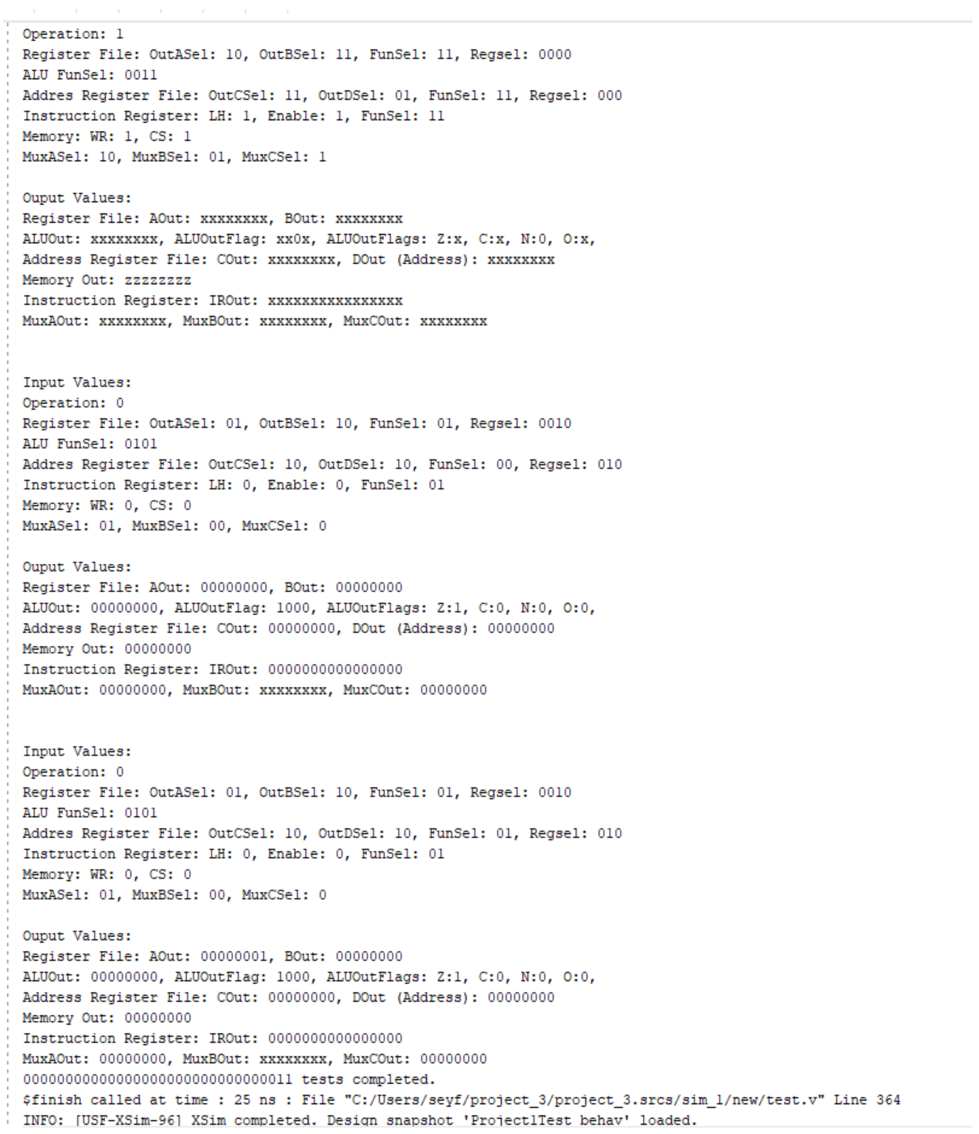


```
module ALUSystem(
    input[1:0] RF_OutASel,
    input[1:0] RF_OutBSel,
    input[1:0] RF_FunSel,
    input[3:0] RF_RegSel,
    input[3:0] ALU_FunSel,
    input[1:0] ARF_OutCSel,
    input[1:0] ARF_OutDSel,
    input[1:0] ARF_FunSel,
    input[2:0] ARF_RegSel,
    input IR_LH,
    input IR_Enable,
    input[1:0] IR_Funsel,
    input Mem_WR,
    input Mem_CS,
    input[1:0] MuxASel,
    input[1:0] MuxBSel,
    input MuxCSel,
    input Clock
    );

    wire[7:0] AOut,BOut; //Register file out
    wire[7:0] ALUOut; //ALU out
    wire[3:0] ALUOutFlag; //ALU out flag
    wire[7:0] ARF_COut, Address; //ARF out
    wire[7:0] MemoryOut;
    wire[15:0] IROut;
    wire[7:0] MuxAOut, MuxBOut, MuxCOut;
//    wire [7:0] irout8;
//    assign irout8 = IROut[7:0];
    registerFile registerFile(MuxAOut, RF_OutASel, RF_OutBSel, RF_FunSel, RF_RegSel,Clock, AOut,BOut);
    muxc muxC(ARF_COut, AOut, MuxCSel, MuxCOut);
    alu ALU(MuxCOut, BOut, ALU_FunSel, ALUOut, ALUOutFlag);
    Memory MEM(Address, ALUOut, Mem_WR, Mem_CS,Clock,MemoryOut);
    addressRegisterFile ARF(MuxBOut, ARF_OutCSel, ARF_OutDSel, ARF_FunSel, ARF_RegSel, Clock,ARF_COut, Address);
    mux muxB(8'bxxxxxxxx,IROut[7:0], MemoryOut,ALUOut,MuxBSel,MuxBOut );
    registerIR IR(MemoryOut, IR_LH, IR_Enable, IR_Funsel, Clock, IROut);
    mux muxA(IROut[7:0], MemoryOut,ARF_COut,ALUOut,MuxASel,MuxAOut );

endmodule
```

Figure 14: Code of the system



```
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: xx0x, ALUOutFlags: Z:x, 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: xxxxxxxx, 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: 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: xxxxxxxx, MuxCOut: 00000000
00000000000000000000000000000011 tests completed.
$finish called at time : 25 ns : File "C:/Users/seyf/project_3/project_3.srcs/sim_1/new/test.v" Line 364
INFO: [USF-XSim-96] XSim completed. Design snapshot 'Project1Test_behav' loaded.
```

Figure 15: Project Test 1 results.

The inputs that were part of other components designed in the previous parts retain the same functionality. Only new inputs are memory related, and the only relevant one being MemWR, which is used for writing into the memory.