ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E
COMPUTER ORGANIZATION
PROJECT 1 REPORT

CRN : 30307

LECTURER : Dr. Kadir Özlem

GROUP MEMBERS:

150210032 : ARİF EREN YOLDAŞ

110190202 : ANIL ARDA TEVEK

SUMMER 2025

Contents

1 INTRODUCTION

The main objective of the project is to build functional modules including registers, register files, an Arithmetic Logic Unit (ALU), and an ALU System to perform the desired operations in the assignment. Each module was implemented in Verilog Hardware Description Language and tested using simulation files provided in the assignment. Vivado Design Suite was used to code and simulate the designed modules. As a group we did not have any task distribution. We helped each other where we can.

2 MATERIALS AND METHODS

In this section, the design and implementation of the modules are explained with schematics and verilog code blocks for each of the parts.

2.1 16-bit Register

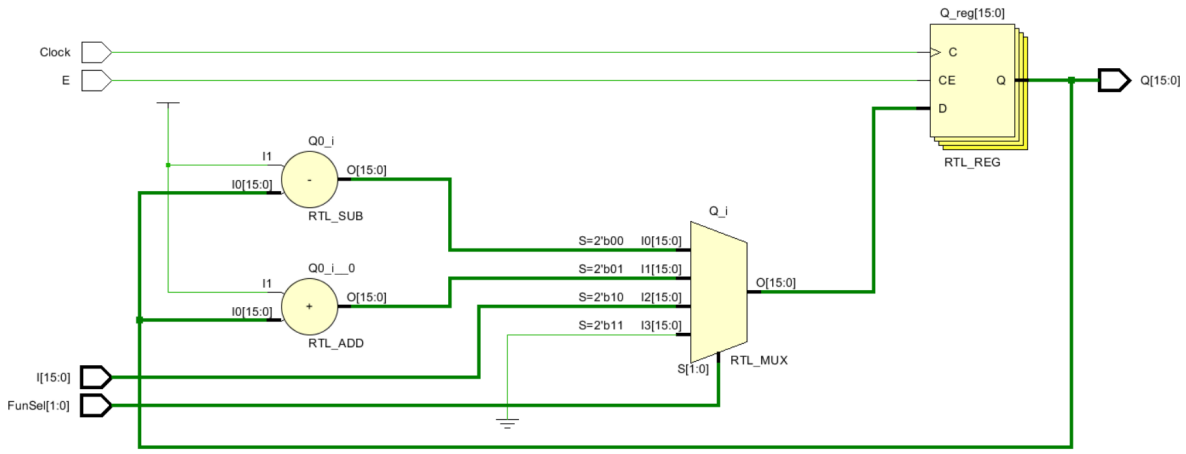

Figure 1: 16-bit Register Schematic

This module is a register designed for 4 operations controlled by a 2-bit FunSel input and an enable signal (E).The register takes Clock, Enable (E), FunSel, and 16-bit input denoted I, as input and 16-bit output denoted Q, as output.

```
module Register16bit (
    input wire Clock,
    input wire E,
    input wire [1:0] FunSel,
    input wire [15:0] I,
    output reg [15:0] Q
);
```

Figure 2: 16-bit Register module definition

The behavior of the module is as follows. When E is low (0), the register maintains its current value regardless of the FunSel input. When E is high (1), the functionality of the register is determined by the value of FunSel. FunSel : 00, decrements the current value of the register. FunSel : 01, increments the current value of the register. FunSel : 10, loads a new 16-bit input to the register. FunSel: 11, clears the register. This register is a pos-edge triggered module where operations are performed only on the rising edge of the clock signal.

```
always @(posedge Clock) begin
    if (E) begin
            case (FunSel)
                2'b00: Q <= Q - 1;          // Decrement
                2'b01: Q <= Q + 1;          // Increment
                2'b10: Q <= I;              // Load input I
                2'b11: Q <= 16'b0;          // Clear
            endcase
    end
end
```

Figure 3: 32-bit Register Sequential Logic Block

The logic is implemented using a case statement that is enabled only if E input is high to evaluate the FunSel input, and synchronous behavior is accomplished using a clock-triggered always block shown in Figure 3.

2.2 32-bit Register

This module is a register designed for 8 operations controlled by a 3-bit FunSel input and an enable signal (E). The register takes Clock, Enable (E), FunSel, and 32-bit input denoted I as input, and outputs 32-bit output denoted Q.

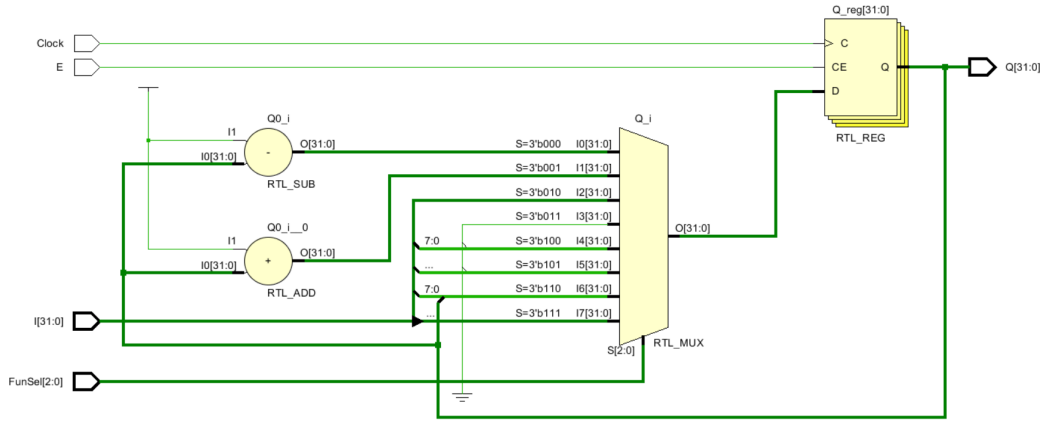

Figure 4: 32-bit Register Schematic

```
1 module Register32bit (
2     input wire Clock,
3     input wire E,
4     input wire [2:0] FunSel,
5     input wire [31:0] I,
6     output reg [31:0] Q
7 );
```

Figure 5: 32-bit Register module definition

The behavior of the module is as follows. When E is low (0), the register maintains its current value regardless of the FunSel input. When E is high (1), the functionality of the register is determined by the value of FunSel. FunSel : 000, decrements the current value of the register. FunSel : 001, increments the current value of the register. FunSel : 010, loads a new 32-bit input to the register.

```
always @(posedge Clock) begin
if (E) begin
    case (FunSel)
        3'b000: Q <= Q - 1;              // Decrement
        3'b001: Q <= Q + 1;              // Increment
        3'b010: Q <= I;                  // Load input I
        3'b011: Q <= 32'b0;              // Clear
        3'b100: Q <= {24'b0, I[7:0]};    // Load lower 8 bits of I
        3'b101: Q <= {16'b0, I[15:0]};   // Load lower 16 bits of I
        3'b110: Q <= {Q[23:0], I[7:0]}; //(8-bit Left Shift)
        3'b111: Q <= {{16{I[15]}}, I[15:0]}; //Sign extend (16bit)
    endcase
end
end
```

Figure 6: 32-bit Register Sequential Logic Block

FunSel : 011, clears the register. FunSel : 100, clears the upper 24 bits and loads the lower 8 bits of the input to Q[7:0]. FunSel : 101, clears the upper 16 bits and loads the lower 16 bits of the input to Q[15:0]. FunSel : 110, shifts the lower 24 bits to the left by 8 and loads the lower 8 bits of the input to Q[7:0]. FunSel : 111, sign-extends the most significant bit of the 16-bit input to Q[31:16] and loads the lower 16 bits to Q[15:0].

The logic is implemented using a case statement that is enabled only if E input is high to evaluate the FunSel input, and synchronous behavior is accomplished using a clock-triggered always block shown in Figure 6.

2.3 Instruction Register

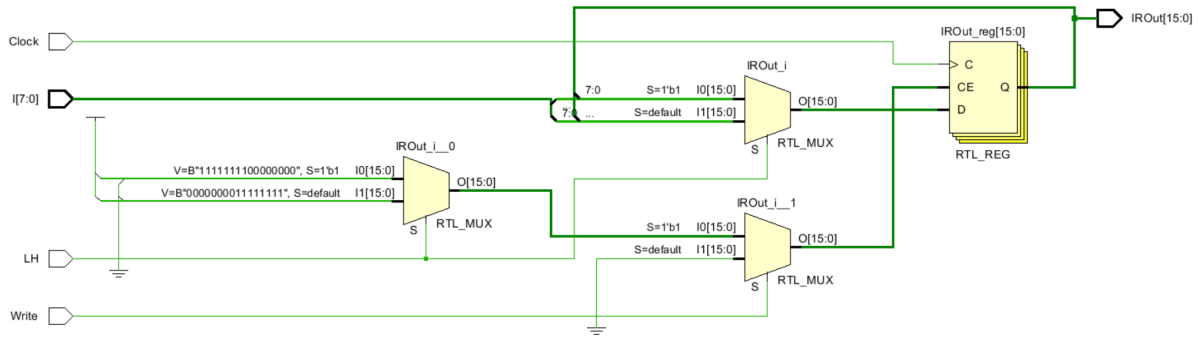

Figure 7: Instruction Register Schematic

This module is a register designed to store a 16-bit instruction value using an 8-bit input bus. The register takes Clock, Write, LH, and 8-bit input denoted I as input, and outputs a 16-bit value denoted IROut.

```
module InstructionRegister (
    input wire Clock,
    input wire Write,
    input wire LH,
    input wire [7:0] I,
    output reg [15:0] IROut
);
```

Figure 8: Instruction Register module definition

The behavior of the module is as follows. When Write is low (0), the register maintains its current value regardless of the LH signal. When Write is high (1), the functionality of the register is determined by the value of LH. LH : 0, loads the lower byte of the register (IROut[7:0]) with the 8-bit input I while preserving the upper byte (IROut[15:8]). LH : 1, loads the upper byte of the register (IROut[15:8]) with the 8-bit input I while preserving the lower byte (IROut[7:0]).

```
always @(posedge Clock) begin
    if (Write) begin
        if (LH) begin
            IROut <= {I, IROut[7:0]};
        end else begin
            IROut <= {IROut[15:8], I};
        end
    end
end
```

Figure 9: Instruction Register Sequential Logic Block

This register is a pos-edge triggered module where operations are performed only on the rising edge of the clock signal. The logic is implemented using nested if-else conditions to evaluate LH, and synchronous behavior is accomplished using a clock-triggered always block shown in Figure 9.

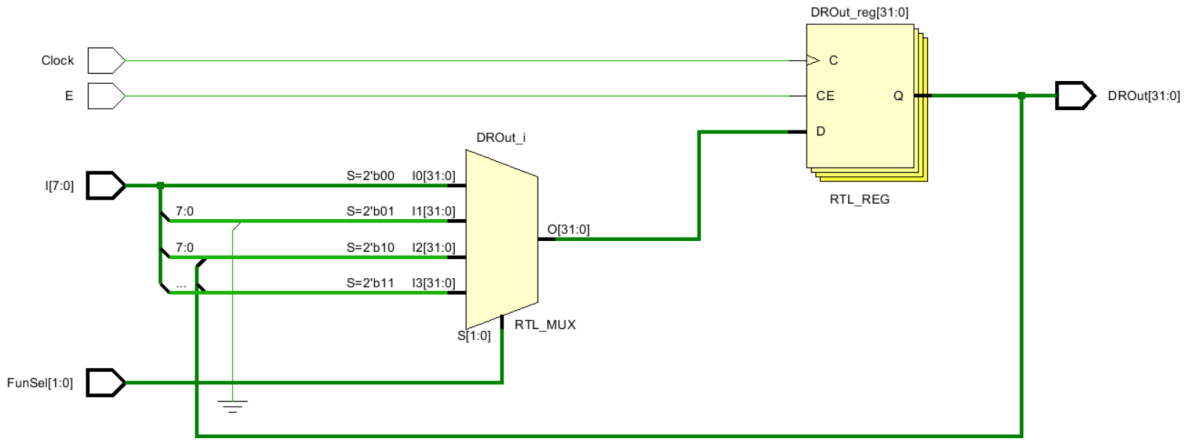

Figure 10: Data Register Schematic

2.4 Data Register

This module is a register designed to store a 32-bit data value using an 8-bit input bus. The register takes Clock, Enable (E), FunSel, and 8-bit input denoted I as input, and outputs a 32-bit value denoted DROut.

```
1 module DataRegister (
2     input wire Clock,
3     input wire E,
4     input wire [1:0] FunSel,
5     input wire [7:0] I,
6     output reg [31:0] DROut
7 );
```

Figure 11: Data Register module definition

Behavior of the module is as follows. When E is low (0), the register retains its current value regardless of the FunSel input. When E is high (1), the operation is determined by the value of FunSel. FunSel : 00, sign-extends the most significant bit of the 8-bit input I to DROut[31:8] and loads I into DROut[7:0]. FunSel : 01, clears the upper 24 bits and loads I into DROut[7:0]. FunSel : 10, performs an 8-bit left shift of DROut[23:0] into DROut[31:8] and loads I into DROut[7:0]. FunSel : 11, performs an 8-bit right shift of DROut[31:8] into DROut[23:0] and loads I into DROut[31:24]. This register is a pos-edge triggered module where operations are executed only on the rising edge of the clock signal.

```
always @(posedge Clock) begin
    if (E) begin
        case (FunSel)
            2'b00: DROut <= {{24{I[7]}}, I};
            2'b01: DROut <= {24'b0, I};
            2'b10: DROut <= {DROut[23:0], I};
            2'b11: DROut <= {I, DROut[31:8]};
        endcase
    end
end
```

Figure 12: Data Register Sequential Logic Block

The logic is implemented using a case statement to evaluate the FunSel input, and synchronous behavior is handled using a clock-triggered always block shown in Figure 12.

2.5 Register File

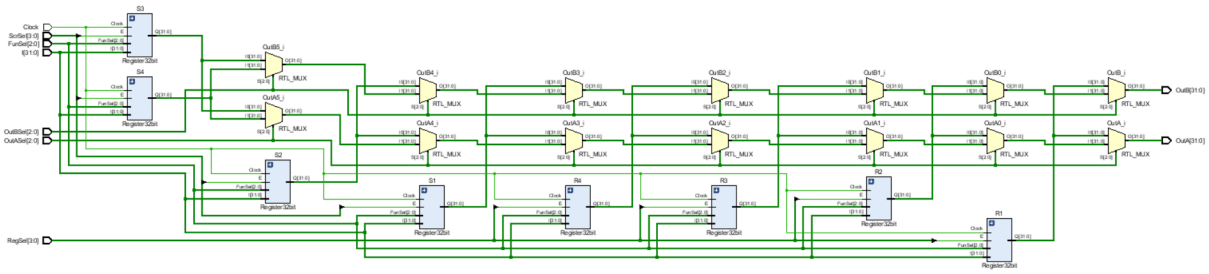

Figure 13: Register File Schematic

This module is a register file composed of four 32-bit general-purpose registers (R1–R4) and four 32-bit scratch registers (S1–S4). The module takes Clock, 4-bit register enable signals RegSel and ScrSel, a 3-bit control input FunSel, a 32-bit input I, and two 3-bit output selectors OutASel and OutBSel, and outputs two 32-bit values through OutA and OutB. The outputs are determined by OutASel and OutBSel, which select one of the eight internal registers to feed to OutA and OutB, respectively.

```
module RegisterFile(
    input   wire        Clock     ,
    input   wire[31:0]  I   ,
    input   wire[2:0]   OutASel ,
    input   wire[2:0]   OutBSel ,
    input   wire[2:0]   FunSel  ,
    input   wire[3:0]   RegSel  ,
    input   wire[3:0]   ScrSel  ,

    output  wire[31:0]  OutA    ,
    output  wire[31:0]  OutB
);
```

Figure 14: Register File module definition

RegSel and ScrSel are 4-bit control signals used to generate individual enable lines for the general-purpose and scratch registers. When a register's enable line is active, the function specified by FunSel is applied to it on the rising edge of the clock. FunSel : 000, decrements the current register value. FunSel : 001, increments the value. FunSel : 010, loads the 32-bit input I. FunSel : 011, clears the register. FunSel : 100, clears the upper 24 bits and loads I[7:0] to Q[7:0]. FunSel : 101, clears the upper 16 bits and loads I[15:0] to Q[15:0]. FunSel : 110, shifts Q[23:0] left by 8 bits and loads I[7:0] into Q[7:0]. FunSel : 111, sign-extends I[15] into Q[31:16] and loads I[15:0] into Q[15:0].

```
    wire ER1, ER2, ER3, ER4, ES1, ES2, ES3, ES4;

    wire[31:0] Q_R1;
    wire[31:0] Q_R2;
    wire[31:0] Q_R3;
    wire[31:0] Q_R4;
    wire[31:0] Q_S1;
    wire[31:0] Q_S2;
    wire[31:0] Q_S3;
    wire[31:0] Q_S4;

    assign {ER1, ER2, ER3, ER4} = RegSel;
    assign {ES1, ES2, ES3, ES4} = ScrSel;
```

Figure 15: Register File Internal Wiring Block

This part of the module represents the internal structure and wiring of the 32-bit

register file composed of four general-purpose registers (R1–R4) and four scratch registers (S1–S4). The 4-bit control inputs RegSel and ScrSel are used to determine which registers are enabled for operation. These signals are unpacked into individual enable lines (ER1–ER4 and ES1–ES4) corresponding to each register.

```
Register32bit R1 (
    .I(I),
    .E(ER1),
    .FunSel(FunSel),
    .Clock(Clock),
    .Q(Q_R1)
);
Register32bit R2 (
    .I(I),
    .E(ER2),
    .FunSel(FunSel),
    .Clock(Clock),
    .Q(Q_R2)
);
Register32bit R3 (
    .I(I),
    .E(ER3),
    .FunSel(FunSel),
    .Clock(Clock),
    .Q(Q_R3)
);
Register32bit R4 (
    .I(I),
    .E(ER4),
    .FunSel(FunSel),
    .Clock(Clock),
    .Q(Q_R4)
);
```

```
Register32bit S1 (
    .I(I),
    .E(ES1),
    .FunSel(FunSel),
    .Clock(Clock),
    .Q(Q_S1)
);
Register32bit S2 (
    .I(I),
    .E(ES2),
    .FunSel(FunSel),
    .Clock(Clock),
    .Q(Q_S2)
);
Register32bit S3 (
    .I(I),
    .E(ES3),
    .FunSel(FunSel),
    .Clock(Clock),
    .Q(Q_S3)
);
Register32bit S4 (
    .I(I),
    .E(ES4),
    .FunSel(FunSel),
    .Clock(Clock),
    .Q(Q_S4)
);
```

Figure 16: Split 32-bit Register Instantiations

Each register is instantiated from the Register32bit module and shares the Clock, FunSel, and 32-bit input I. When a register's enable signal is active, the function selected by FunSel is applied to it on the rising edge of the clock. Additionally, two 3-bit multiplexer selectors, OutASel and OutBSel, are used to determine which of the eight registers

are routed to the output ports OutA and OutB. This output selection is handled using combinational logic that maps the selector values to the corresponding register outputs.

```
1       assign OutA = (OutASel == 3'b000) ? Q_R1 :
2                     (OutASel == 3'b001) ? Q_R2 :
3                     (OutASel == 3'b010) ? Q_R3 :
4                     (OutASel == 3'b011) ? Q_R4 :
5                     (OutASel == 3'b100) ? Q_S1 :
6                     (OutASel == 3'b101) ? Q_S2 :
7                     (OutASel == 3'b110) ? Q_S3 : Q_S4;
8       assign OutB = (OutBSel == 3'b000) ? Q_R1 :
9                     (OutBSel == 3'b001) ? Q_R2 :
10                    (OutBSel == 3'b010) ? Q_R3 :
11                    (OutBSel == 3'b011) ? Q_R4 :
12                    (OutBSel == 3'b100) ? Q_S1 :
13                    (OutBSel == 3'b101) ? Q_S2 :
14                    (OutBSel == 3'b110) ? Q_S3 : Q_S4;
```

Figure 17: Register File Output Multiplexer Logic Block

2.6 Address Register File

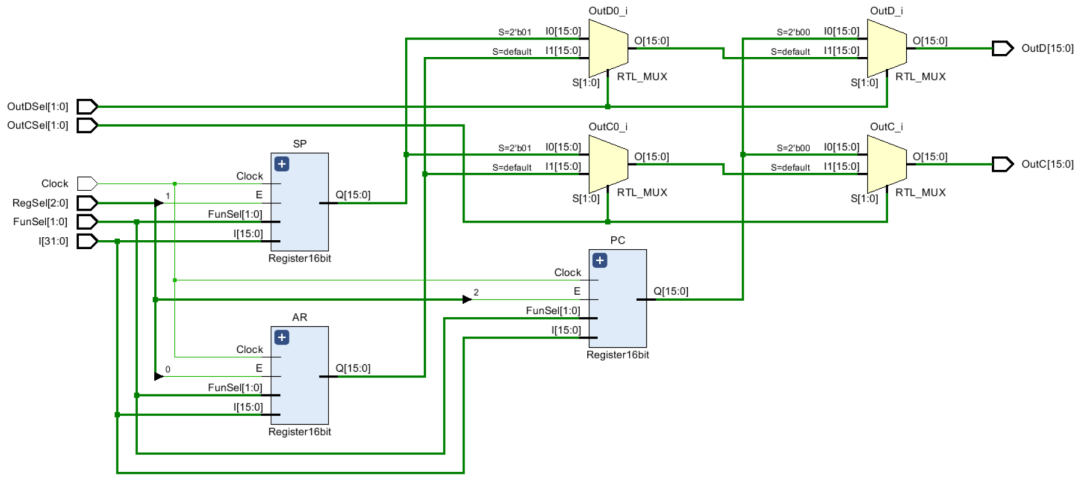

Figure 18: Address Register File Schematic

```
module AddressRegisterFile(
    input wire Clock,
    input wire [31:0] I,
    input wire [2:0] RegSel,
    input wire [1:0] FunSel,
    input wire [1:0] OutCSel,
    input wire [1:0] OutDSel,
    output wire [15:0] OutC,
    output wire [15:0] OutD
);
```

Figure 19: Address Register File module definition

This module is an address register file composed of three 16-bit registers: the program counter (PC), address register (AR), and stack pointer (SP). The module takes Clock, a 3-bit control input RegSel, a 2-bit control input FunSel, and a 16-bit input I, and it outputs two 16-bit values through OutC and OutD.

```
    wire PC_E, AR_E, SP_E;
    wire [15:0] Q_PC, Q_AR, Q_SP;

    assign PC_E = RegSel[2];
    assign SP_E = RegSel[1];
    assign AR_E = RegSel[0];
```

Figure 20: Address Register File Internal Wiring Block

```
    Register16bit PC(.I(I[15:0]), .E(PC_E), .FunSel(FunSel), .
       Clock(Clock), .Q(Q_PC));
    Register16bit AR(.I(I[15:0]), .E(AR_E), .FunSel(FunSel), .
       Clock(Clock), .Q(Q_AR));
    Register16bit SP(.I(I[15:0]), .E(SP_E), .FunSel(FunSel), .
       Clock(Clock), .Q(Q_SP));
```

Figure 21: Address Register File 16-bit Register Instantiations Block

The outputs are determined by 2-bit selectors OutCSel and OutDSel, which select one of the three internal registers to input to OutC and OutD, respectively. RegSel is a 3-bit control signal used to generate individual enable lines for PC, AR, and SP based on specific bit combinations. When the enable signal for a register is active, the function

specified by FunSel is applied to that register on the rising edge of the clock. FunSel : 00, decrements the current register value. FunSel : 01, increments the value. FunSel : 10, loads the 16-bit input I. FunSel : 11, clears the register.

```
1
2
3       assign OutC = (OutCSel == 2'b00) ? PC.Q :
4                     (OutCSel == 2'b01) ? SP.Q : AR.Q;
5
6       assign OutD = (OutDSel == 2'b00) ? PC.Q :
7                     (OutDSel == 2'b01) ? SP.Q : AR.Q;
```

Figure 22: Address Register File Output Multiplexer Logic Block

The module instantiates three Register16bit units, each operating independently under shared control inputs and selected by the RegSel logic, as shown in Figure 22.

2.7 Arithmetic Logic Unit

```
1 module ArithmeticLogicUnit (
2       input wire [31:0] A,
3       input wire [31:0] B,
4       input wire [4:0] FunSel,
5       input wire WF,
6       input wire Clock,
7       output wire [31:0] ALUOut,
8       output reg [3:0] FlagsOut // {Z, C, N, O}
9 );
```

Figure 23: Arithmetic Logic Unit Module Definition

A combinational Arithmetic Logic Unit (ALU) was developed to support both 32-bit and 16-bit operations. The unit performs a set of arithmetic and logical computations based on the provided control signals. Unlike sequential circuits, this ALU does not rely on clocked elements for its core functionality. However, a control input named WF was integrated into the design to handle flag updates. When WF is asserted, the resulting condition flags (Zero, Carry, Negative, and Overflow) are stored in their respective registers.

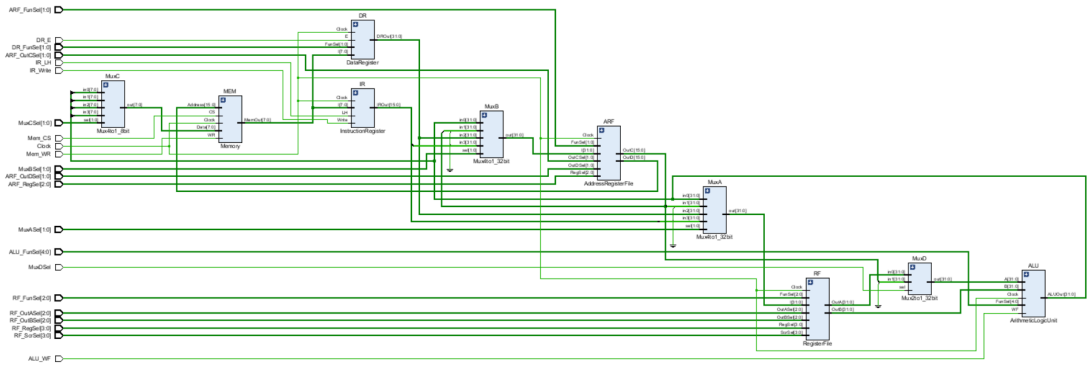

Figure 24: ALU System Schematic

2.8 ALU System

ALU System connects the modules designed previously into a complete system. It includes Arithmetic Logic Unit (ALU), Register File (RF), Address Register File (ARF) and Instruction Register (IR). With the help of Multiplexers inputs for ALU are selected and outputs from the ALU is outputed to different registers to update register values. It is designed to execute arithmetic and logic operations based on instructions loaded from Instruction Register.

3 RESULTS

3.1 16-bit Register Simulation

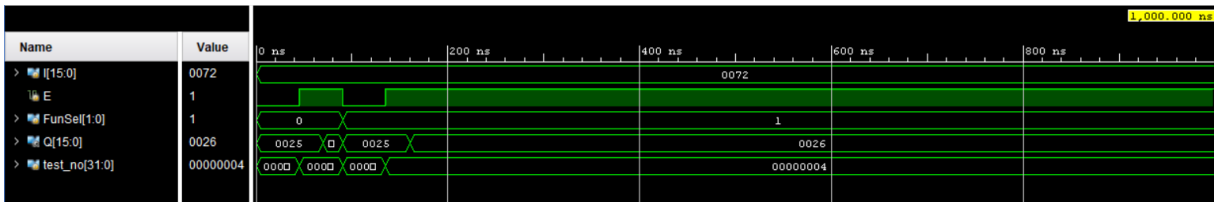

Figure 25: 16-bit Register Simulation Output

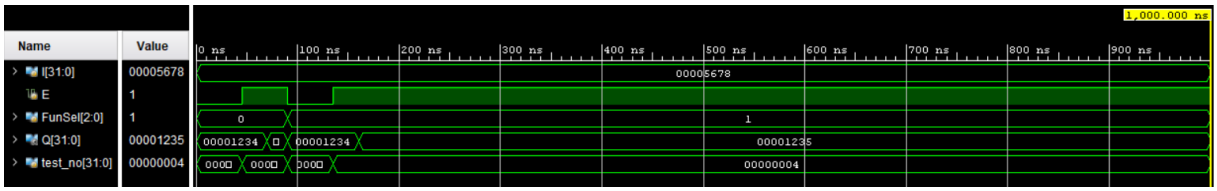

Figure 26: 32-bit Register Simulation Output

3.2 32-bit Register Simulation

3.3 Instruction Register Simulation

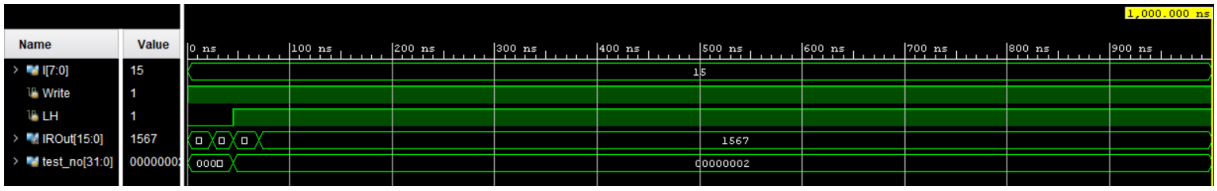

Figure 27: Instruction Register Simulation Output

3.4 Data Register Simulation

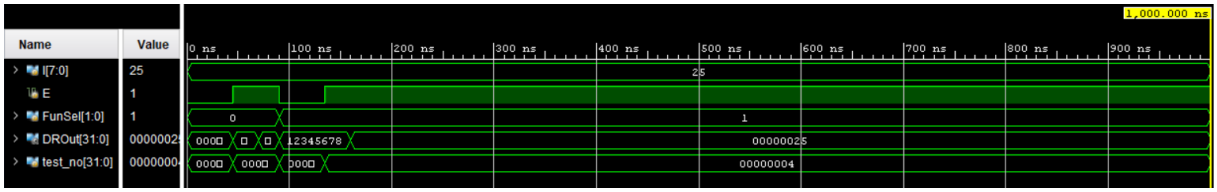

Figure 28: Data Register Simulation Output

3.5 Address Register File Simulation

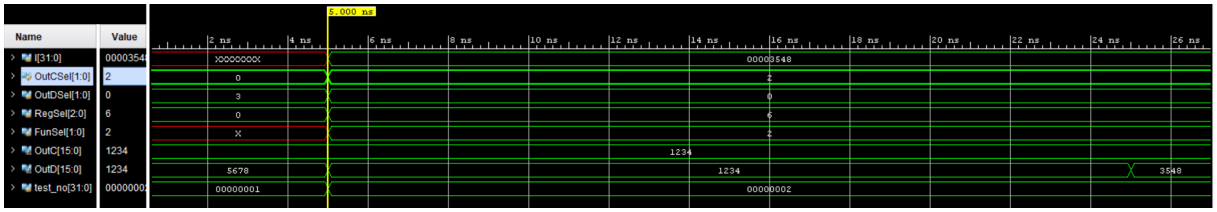

Figure 29: Address Register File Simulation Output

3.6 Arithmetic Logic Unit Simulation

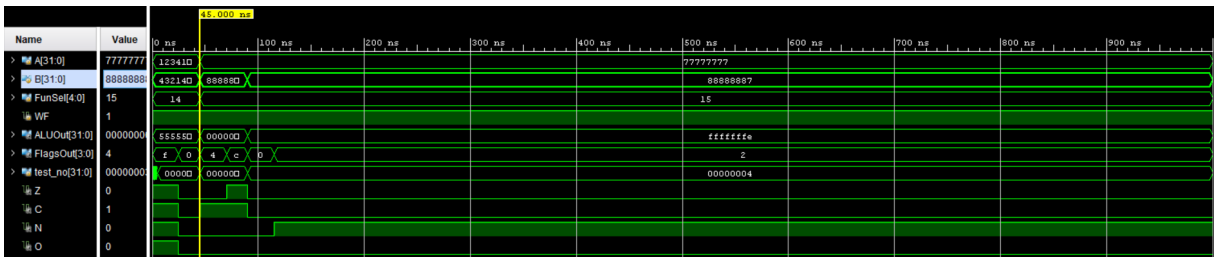

Figure 30: Arithmetic Logic Unit Simulation Output

3.7 ALU System Simulation

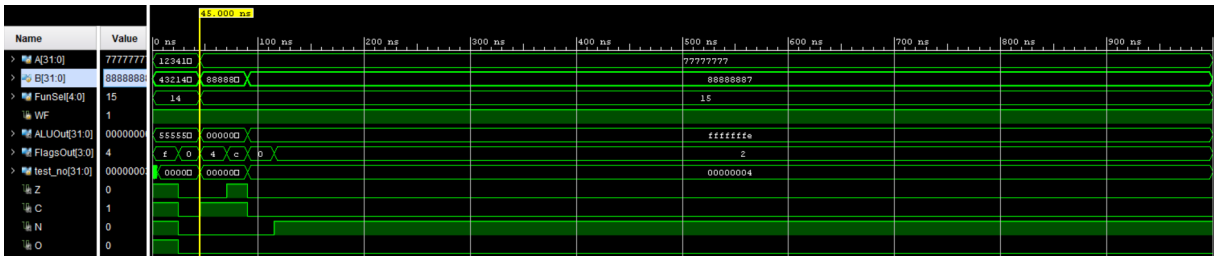

Figure 31: ALU System Simulation Output

4 DISCUSSION

During the development of this project, we learned how to design and implement the smaller building blocks that create the more complex desired system. We implemented registers, register files, and an arithmetic logic unit using Verilog hardware description language. We have seen that the smallest errors in any of the sub-components would result in errors that prevent the operation of any subsequent components. Thus, we have learned to be cautious and diligent while designing and implementing any of the modules. We have also observed that there are more than one solution for designing any system and careful consideration is needed in order to achieve the most effective and efficient designs.

5 CONCLUSION

Most of our challenges stemmed from us being unfamiliar with the Vivado Design Suite program. Although it offers an excellent simulation of the designs which helped us debug many of our errors, setting the simulations and getting used to the program was sometimes frustrating.