ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E
COMPUTER ORGANIZATION
PROJECT 1 REPORT

CRN : 21335

LECTURER : Prof. Dr. Deniz Turgay ALTILAR

GROUP MEMBERS:

150220733 : Özden KARS

150210067 : Alper DAŞGIN

SPRING 2024

Contents

1 INTRODUCTION **1**

2 MATERIALS AND METHODS **2**

2.1 PART-1 2

2.2 PART-2 3

2.2.1 PART-2(a) 3

2.2.2 PART-2(b) 4

2.2.3 PART-2(c) 5

2.3 PART-3 6

2.4 PART-4 7

3 RESULTS **8**

3.1 PART-1 8

3.2 PART-2 8

3.2.1 PART-2(a) 8

3.2.2 PART-2(b) 9

3.2.3 PART-2(c) 9

3.3 PART-3 10

3.4 PART-4 11

4 DISCUSSION **12**

4.1 PART-1 12

4.2 PART-2 13

4.2.1 PART-2(a) 13

4.2.2 PART-2(b) 14

4.2.3 PART-2(c) 15

4.3 PART-3 17

4.4 PART-4 18

5 CONCLUSION **19**

6 REFERENCES **20**

1 INTRODUCTION

This tasks presents the comprehensive development and simulation of a CPU architecture in Verilog, spanning four distinct parts. In part-1, we developed a module for a basic register with functionalities like decrement, increment, load, clear, and byte manipulation. Part-2 introduces the Verilog implementation of an instruction register, capable of loading either high or low bytes based on control signals (Write and L_H) and clock signals (Clock). Additionally, a register file is created, enabling operations such as incrementing, decrementing, loading, or clearing based on function selection and clock signal. Part-2 also includes an address register file module, which selects and operates on different registers using control signals and input data behaviorally. Part-3 introduces an Arithmetic Logic Unit (ALU) performing arithmetic and logic operations on 16-bit operands, updating status flags accordingly. Operations are determined by FunSel, with results stored in temp and flags updated when the WF (Write Flags) signal is asserted. Part-4 presents Verilog code depicting a CPU structure, emphasizing the ALU, with a simulation environment testing its functionality under various scenarios, verifying outputs against expected values.

2 MATERIALS AND METHODS

Verilog is a hardware description language (HDL) used for designing and modeling digital circuits and systems. While implementing every part step by step, we constantly used Verilog HDL(Hardware Description Language) to make bitwise operations and create constructions like ALU and registers.

2.1 PART-1

A register is a small amount of storage within the CPU (central processing unit) used for temporarily holding data that is being processed or manipulated. Registers are the fastest type of memory in a computer system and are located directly on the CPU chip itself. In part-1 we created a module to represent a simple register with basic functionalities such as decrement, increment, load, clear, and byte manipulation.

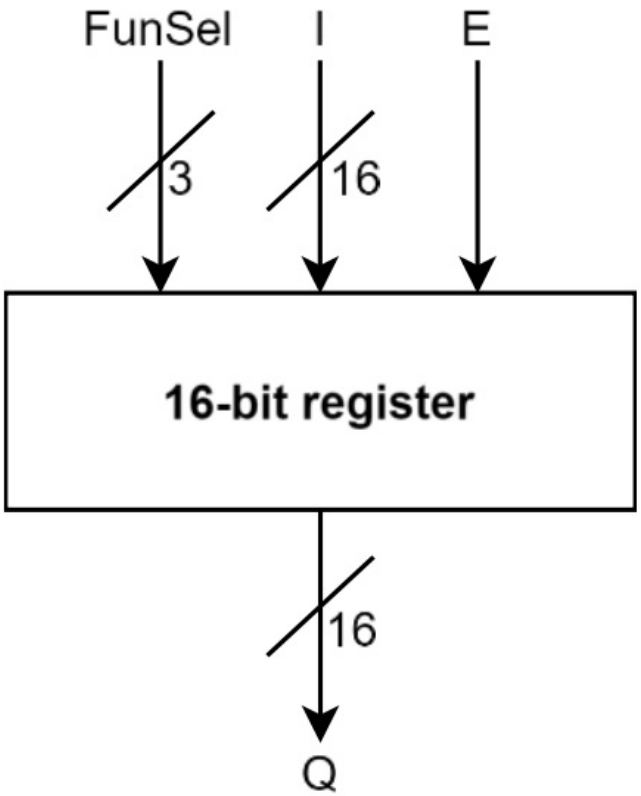

Figure 1: 16-bit register.

2.2 PART-2

2.2.1 PART-2(a)

An instruction register is a type of register found in a CPU (Central Processing Unit) that holds the instruction currently being executed or decoded. When a program is executed, instructions are fetched from memory and loaded into the instruction register for decoding and execution by the CPU.cThe first premise of part-2 consists of the Verilog module implementation of an instruction register that can load either the high or low byte of the input data I based on control signals (Write and L_H) and the clock signal (Clock).

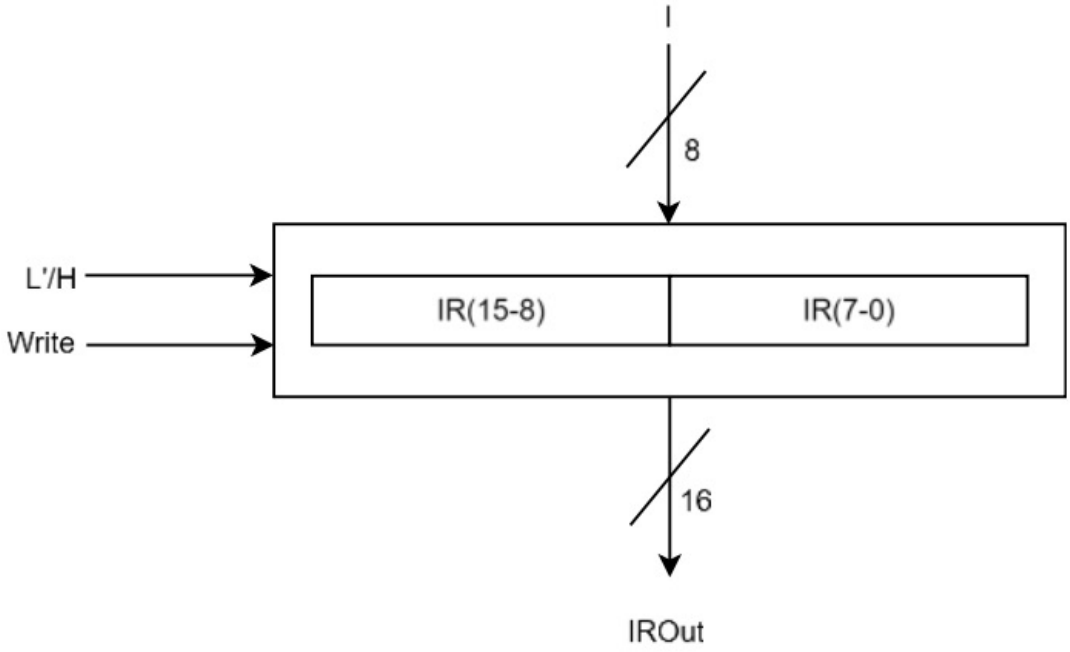

Figure 2: Instruction register(IR).

2.2.2 PART-2(b)

In the second premise of part-2 we created a register file that consists of multiple registers, each capable of performing various operations such as incrementing, decrementing, loading, or clearing based on function selection and a clock signal. The register files are part of the CPU's architecture that consists of a set of registers used for temporarily storing data. The register file is often organized as an array of registers, each capable of holding a single binary value. These registers are generally used for holding operands, intermediate results, addresses, and other data needed for executing instructions. The register file serves as a high-speed storage unit within the CPU, allowing for quick access to data by the CPU's execution units (such as the arithmetic logic unit or ALU). Instead of fetching data from slower main memory (RAM) every time an operation is performed, the CPU can quickly access data stored in registers within the register file.

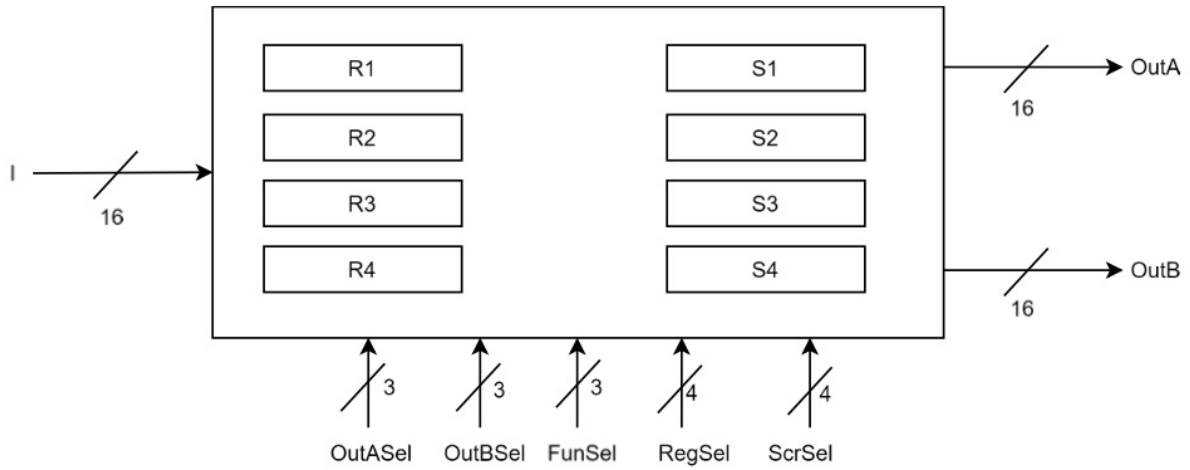

Figure 3: 16-bit general purpose and scratch registers, inputs, and outputs.

2.2.3 PART-2(c)

The Address Register (AR) is a type of register used to store memory addresses. These registers are used to hold the addresses of memory locations for data access or for addressing instructions during program execution. The memory address of the next instruction that needs to be fetched and performed by the CPU is stored in the Program Counter (PC), a specially purpose register. It basically maintains track of where you are in the running software. The PC is usually incremented to point to the next instruction in memory when each instruction is fetched, enabling the CPU to run the program sequentially. Another special-purpose register used to monitor the top of the stack in memory is called the Stack Pointer (SP). A stack is a data structure used in computer architecture that adheres to the Last In, First Out (LIFO) principle. The memory address of the most recent item pushed onto the stack or the next place a push operation can be performed is stored in the stack pointer. For the third premise of part-2 we created an address register file module. This Verilog module has the capability to select and perform operations on different registers based on control signals and input data as behavioral. The behavior of this module is defined within an always block triggered by changes in the RegSel signal. Based on the value of RegSel, corresponding enable signals (EAR, EPC, ESP) are set to enable the appropriate registers. Registers AR (Address Register), PC (Program Counter), and SP (Stack Pointer) perform operations based on the input data I, function selection FunSel, and clock signal Clock.

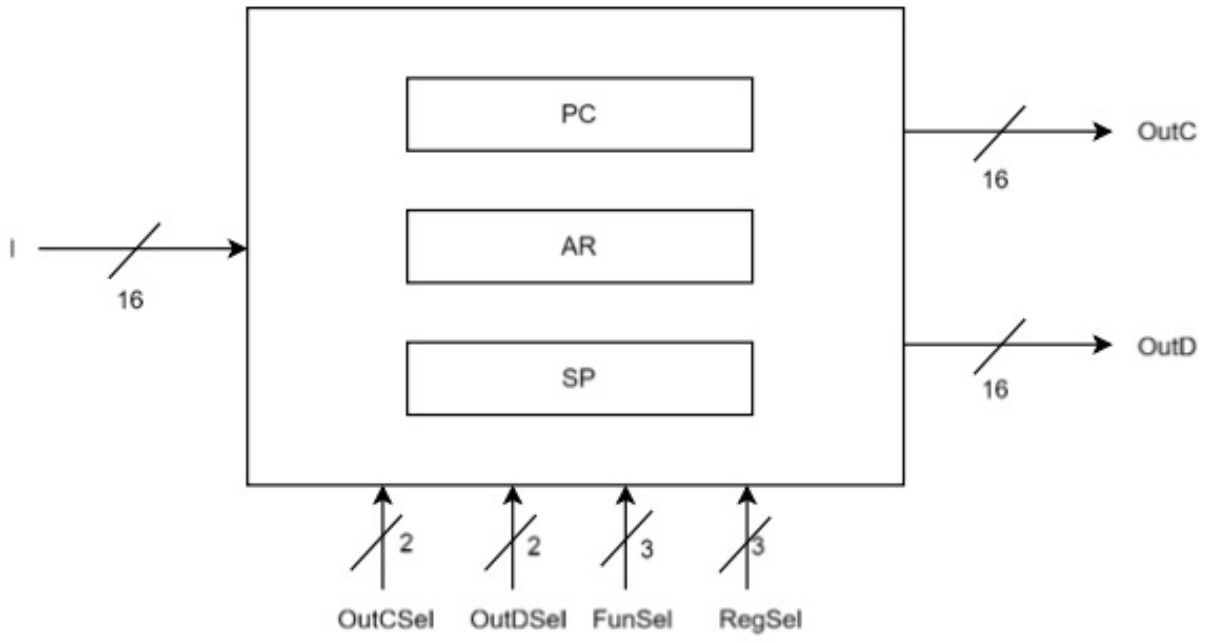

Figure 4: 16-bit address registers, inputs, and outputs.

2.3 PART-3

using the ALU, Arithmetic operations including addition, subtraction, multiplication, and division can all be carried out. Moreover, it is capable of carrying out logical operations like AND, OR, XOR, and NOT. These are fundamental processes for running computer programs and modifying data. The module created In part-3 implements an Arithmetic Logic Unit (ALU) capable of performing arithmetic and logic operations on 16-bit operands, with support for updating status flags. The events proceed as follows: We defined within an always @(*) block which triggers whenever any of the inputs change. The operation performed depends on the value of FunSel: If the most significant bit of FunSel is 0, it treats the inputs (A and B) as 8-bit operands and performs the operation accordingly. If the most significant bit of FunSel is 1, it treats the inputs as 16-bit operands. The results of the operation are stored in temp. Based on the operation performed, status flags (Zero, Negative, Carry, Overflow) are updated accordingly. The updated status flags are stored in FlagsOut when the WF (Write Flags) signal is asserted.

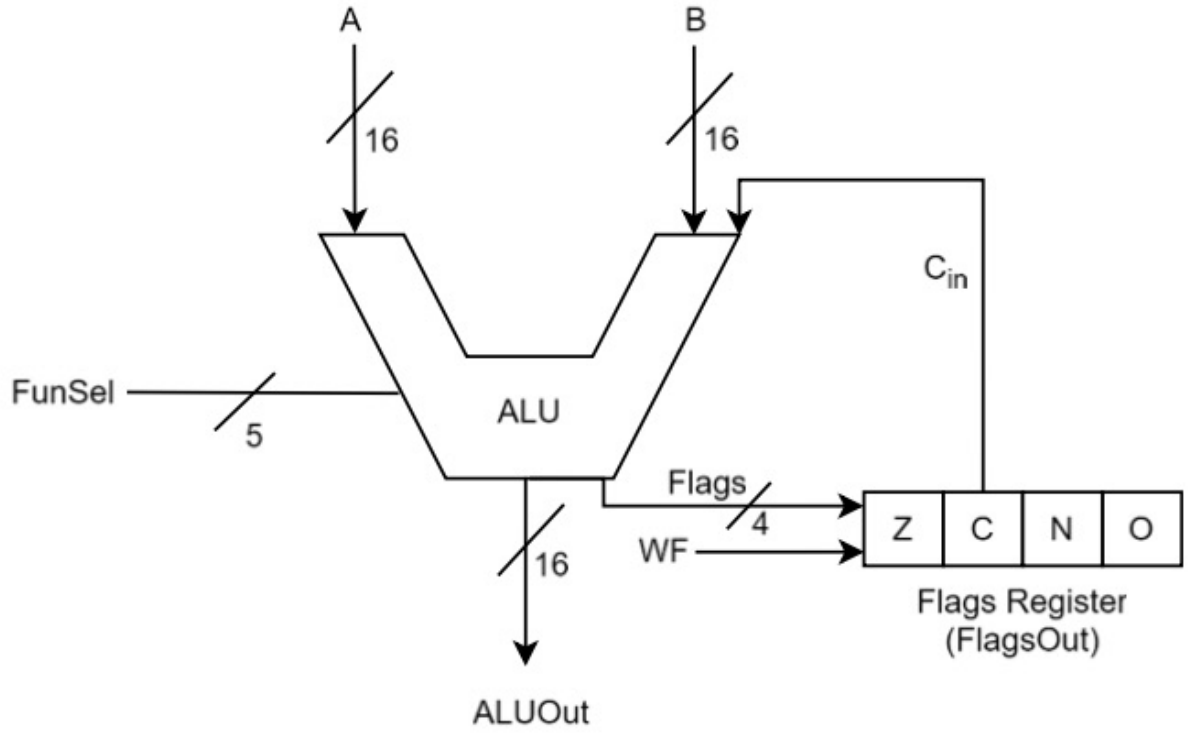

Figure 5: Arithmetic Logic Unit graphic.

2.4 PART-4

In part-4 also the last part Verilog code represents the structure of a CPU, specifically focusing on the Arithmetic Logic Unit (ALU) and its simulation environment. This simulation environment consists of merging previous parts cumulatively and effectively tests the functionality of the "ArithmeticLogicUnit" module under different scenarios and verifies its output against expected values.

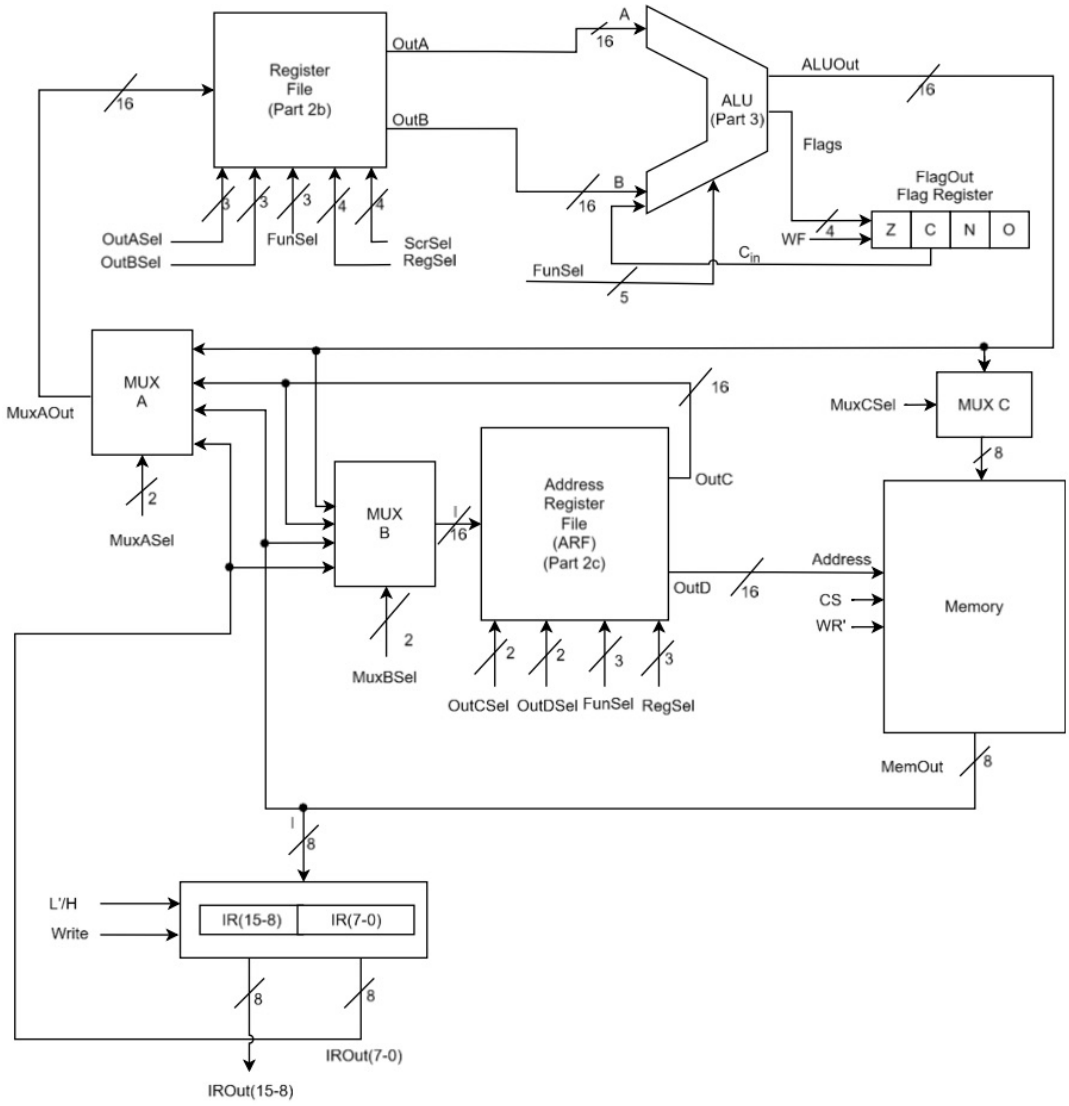

Figure 6: Arithmetic Logic Unit System.

3 RESULTS

3.1 PART-1

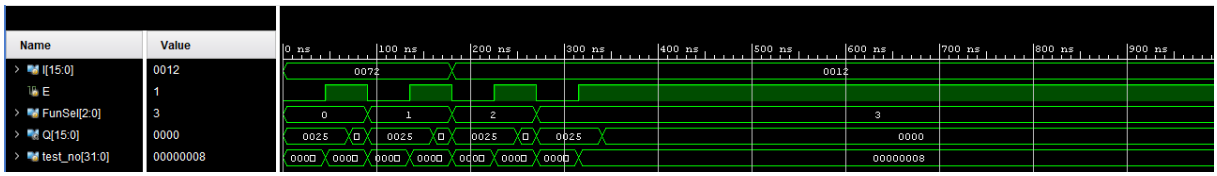

Figure 7: Simulation of general purpose register

```
Register Simulation Started
[PASS] Test No: 1, Component: Q, Actual Value: 0x0025, Expected Value: 0x0025
[PASS] Test No: 2, Component: Q, Actual Value: 0x0024, Expected Value: 0x0024
[PASS] Test No: 3, Component: Q, Actual Value: 0x0025, Expected Value: 0x0025
[PASS] Test No: 4, Component: Q, Actual Value: 0x0026, Expected Value: 0x0026
[PASS] Test No: 5, Component: Q, Actual Value: 0x0025, Expected Value: 0x0025
[PASS] Test No: 6, Component: Q, Actual Value: 0x0012, Expected Value: 0x0012
[PASS] Test No: 7, Component: Q, Actual Value: 0x0025, Expected Value: 0x0025
[PASS] Test No: 8, Component: Q, Actual Value: 0x0000, Expected Value: 0x0000
Register Simulation Finished
```

Figure 8: Simulation result of general purpose register

3.2 PART-2

3.2.1 PART-2(a)

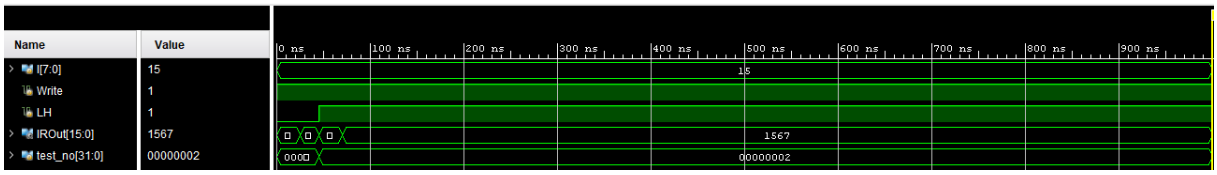

Figure 9: Simulation of instruction register

```
InstructionRegister Simulation Started
[PASS] Test No: 1, Component: IROut, Actual Value: 0x2315, Expected Value: 0x2315
[PASS] Test No: 2, Component: IROut, Actual Value: 0x1567, Expected Value: 0x1567
InstructionRegister Simulation Finished
```

Figure 10: Simulation result of instruction register

3.2.2 PART-2(b)

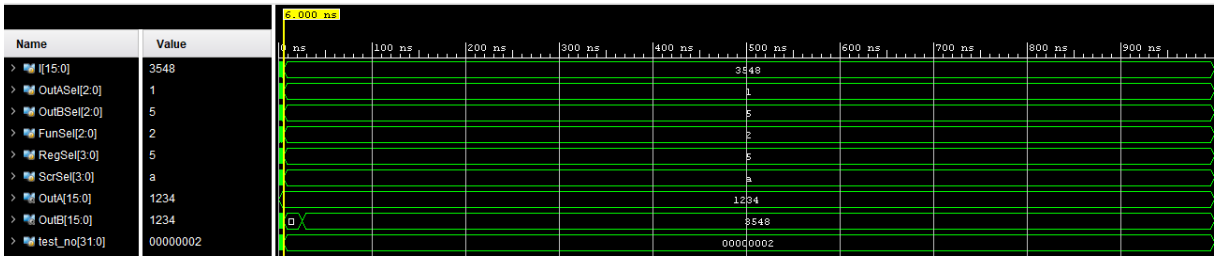

Figure 11: Simulation of register file

```
RegisterFile Simulation Started
[PASS] Test No: 1, Component: OutA, Actual Value: 0x1234, Expected Value: 0x1234
[PASS] Test No: 1, Component: OutB, Actual Value: 0x5678, Expected Value: 0x5678
[PASS] Test No: 2, Component: OutA, Actual Value: 0x1234, Expected Value: 0x1234
[PASS] Test No: 2, Component: OutB, Actual Value: 0x3548, Expected Value: 0x3548
RegisterFile Simulation Finished
```

Figure 12: Simulation result of register file

3.2.3 PART-2(c)

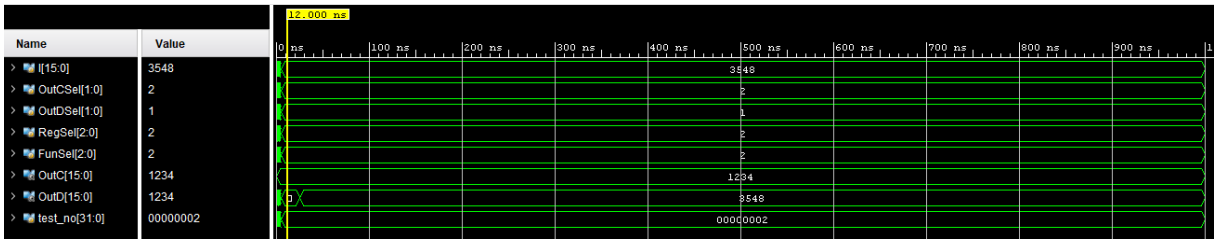

Figure 13: Simulation of address register file

```
AddressRegisterFile Simulation Started
[PASS] Test No: 1, Component: OutC, Actual Value: 0x1234, Expected Value: 0x1234
[PASS] Test No: 1, Component: OutD, Actual Value: 0x5678, Expected Value: 0x5678
[PASS] Test No: 2, Component: OutA, Actual Value: 0x1234, Expected Value: 0x1234
[PASS] Test No: 2, Component: OutB, Actual Value: 0x3548, Expected Value: 0x3548
AddressRegisterFile Simulation Finished
```

Figure 14: Simulation result of address register file

3.3 PART-3

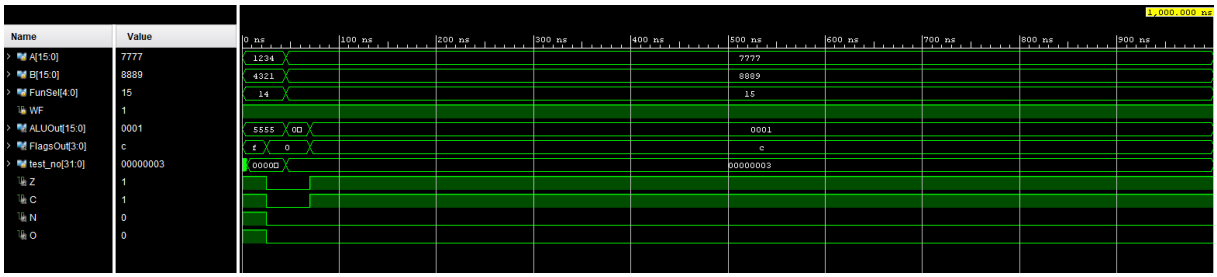

Figure 15: Simulation of arithmetic logic unit

```
ArithmeticLogicUnit Simulation Started
[PASS] Test No: 1, Component: ALUOut, Actual Value: 0x5555, Expected Value: 0x5555
[PASS] Test No: 1, Component: Z, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 1, Component: C, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 1, Component: N, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 1, Component: O, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 2, Component: ALUOut, Actual Value: 0x5555, Expected Value: 0x5555
[PASS] Test No: 2, Component: Z, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 2, Component: C, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 2, Component: N, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 2, Component: O, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 3, Component: ALUOut, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 3, Component: Z, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 3, Component: C, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 3, Component: N, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 3, Component: O, Actual Value: 0x0000, Expected Value: 0x0000
ArithmeticLogicUnit Simulation Finished
```

Figure 16: Simulation result of arithmetic logic unit

3.4 PART-4

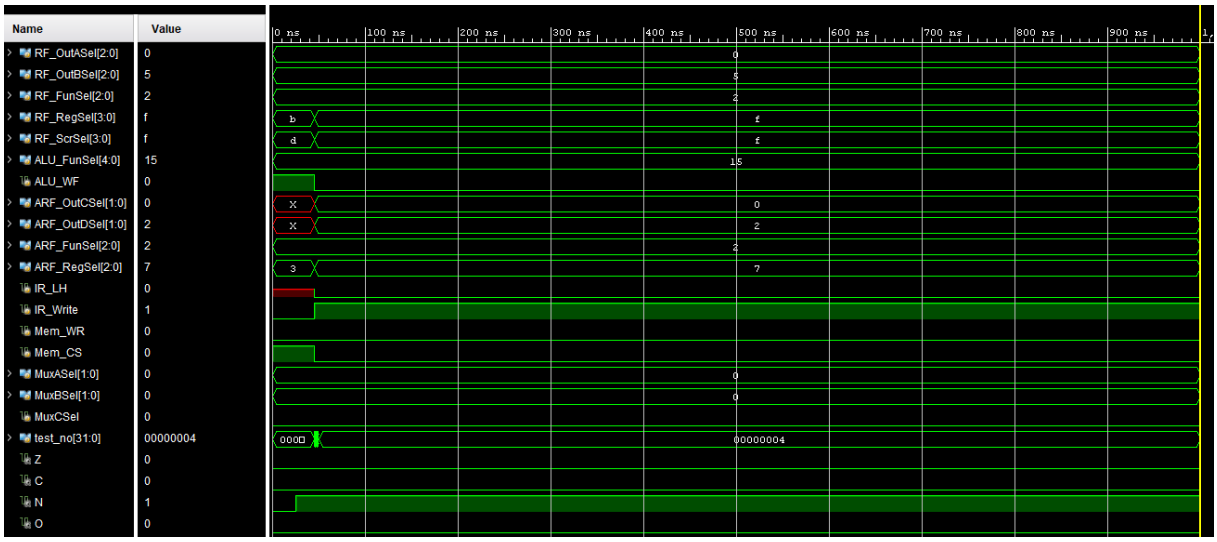

Figure 17: Simulation of arithmetic logic unit system

```
ArithmeticLogicUnitSystem Simulation Started
[PASS] Test No: 1, Component: OutA, Actual Value: 0x7777, Expected Value: 0x7777
[PASS] Test No: 1, Component: OutB, Actual Value: 0x8887, Expected Value: 0x8887
[PASS] Test No: 1, Component: ALUOut, Actual Value: 0xfffe, Expected Value: 0xfffe
[PASS] Test No: 1, Component: Z, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 1, Component: C, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 1, Component: N, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 1, Component: O, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 1, Component: MuxAOut, Actual Value: 0xfffe, Expected Value: 0xfffe
[PASS] Test No: 1, Component: MuxBOut, Actual Value: 0xfffe, Expected Value: 0xfffe
[PASS] Test No: 1, Component: MuxCOut, Actual Value: 0x00fe, Expected Value: 0x00fe
[PASS] Test No: 1, Component: R2, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 1, Component: S3, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 2, Component: OutA, Actual Value: 0x7777, Expected Value: 0x7777
[PASS] Test No: 2, Component: OutB, Actual Value: 0x8887, Expected Value: 0x8887
[PASS] Test No: 2, Component: ALUOut, Actual Value: 0xfffe, Expected Value: 0xfffe
[PASS] Test No: 2, Component: Z, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 2, Component: C, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 2, Component: N, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 2, Component: O, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 2, Component: MuxAOut, Actual Value: 0xfffe, Expected Value: 0xfffe
[PASS] Test No: 2, Component: MuxBOut, Actual Value: 0xfffe, Expected Value: 0xfffe
[PASS] Test No: 2, Component: MuxCOut, Actual Value: 0x00fe, Expected Value: 0x00fe
[PASS] Test No: 2, Component: R2, Actual Value: 0xfffe, Expected Value: 0xfffe
[PASS] Test No: 2, Component: S3, Actual Value: 0xfffe, Expected Value: 0xfffe
[PASS] Test No: 2, Component: PC, Actual Value: 0xfffe, Expected Value: 0xfffe
[PASS] Test No: 3, Component: OutC, Actual Value: 0x1254, Expected Value: 0x1254
[PASS] Test No: 3, Component: Address, Actual Value: 0x0023, Expected Value: 0x0023
[PASS] Test No: 3, Component: Memout, Actual Value: 0x0015, Expected Value: 0x0015
[PASS] Test No: 3, Component: IROut, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 4, Component: OutC, Actual Value: 0x1254, Expected Value: 0x1254
[PASS] Test No: 4, Component: Address, Actual Value: 0x0023, Expected Value: 0x0023
[PASS] Test No: 4, Component: Memout, Actual Value: 0x0015, Expected Value: 0x0015
[PASS] Test No: 4, Component: IROut, Actual Value: 0x0015, Expected Value: 0x0015
ArithmeticLogicUnitSystem Simulation Finished
```

Figure 18: Simulation result of arithmetic logic unit system

4 DISCUSSION

In this project, we design a basic CPU from scratch. First, we implement a general-purpose register. After that, we implement an instruction register and a register file using a register module. The arithmetic register file consists of a program counter (PC), an address register (AR), and a stack pointer (SP), which are implemented using the general-purpose register. Next, we create an arithmetic logic unit (ALU) that applies arithmetic and logical operations to two 16-bit inputs, producing a 16-bit output along with bit flags (carry, zero, negative, and overflow). The last part of this project involves connecting all components that we have created so far. Here, we create a basic CPU using the given memory, instruction register, register file, arithmetic logic unit, and address register file.

4.1 PART-1

As Figure 19 indicates, we firstly look at E, which indicates enable. If E equals zero, then the default value will be retained; if not, the FunSel will perform on the input data, which is I in the diagram. I is a 16-bit binary number. Also, there is a Clock connected to the general-purpose register, which decides when the output will be updated. Operation will be performed on data chosen with the FunSel input, which is a 3-bit input. The FunSel input is used as a selector of an 8:1 MUX. The input of the 8:1 MUX is created by the FunSel table (Figure 20). In Verilog, to create a MUX we use "case".

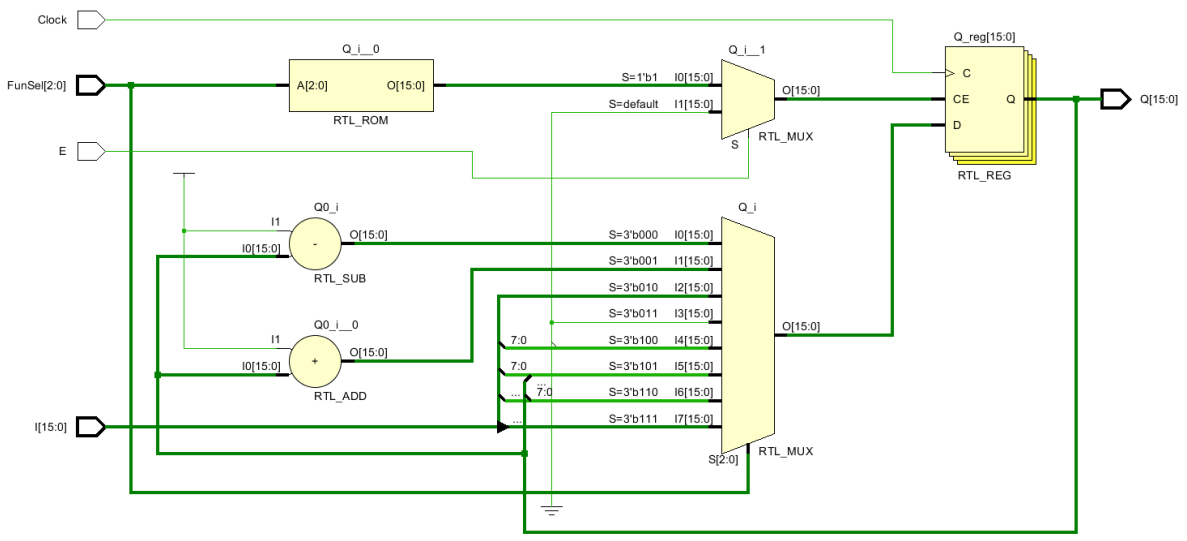

Figure 19: RTL design of general purpose register

E	FunSel	Q^+
0	ϕ	Q (Retain value)
1	000	Q-1 (Decrement)
1	001	Q+1 (Increment)
1	010	I (Load)
1	011	0 (Clear)
1	100	Q (15-8) ← Clear, Q (7-0) ← I (7-0) (Write Low)
1	101	Q (7-0) ← I (7-0) (Only Write Low)
1	110	Q (15-8) ← I (7-0) (Only Write High)
1	111	Q (15-8) ← Sign Extend (I (7)) Q (7-0) ← I (7-0) (Write Low)

Figure 20: Function Selector of general purpose register

4.2 PART-2

4.2.1 PART-2(a)

As shown in Figure 21, there are four inputs: I, an 8-bit input, and Write, Clock, LH, which are 1-bit inputs. LH decides whether the given inputs will be written to the high [15:8] or low [7:0] part of the existing number; it works as a FunSel but is 1 bit. Write functions as an enable in this register. Clock serves the same purpose as above. The output will be 16 bits. Since there are two options for Write and LH, we use a nested if statement to perfom these operations in the Verilog code.

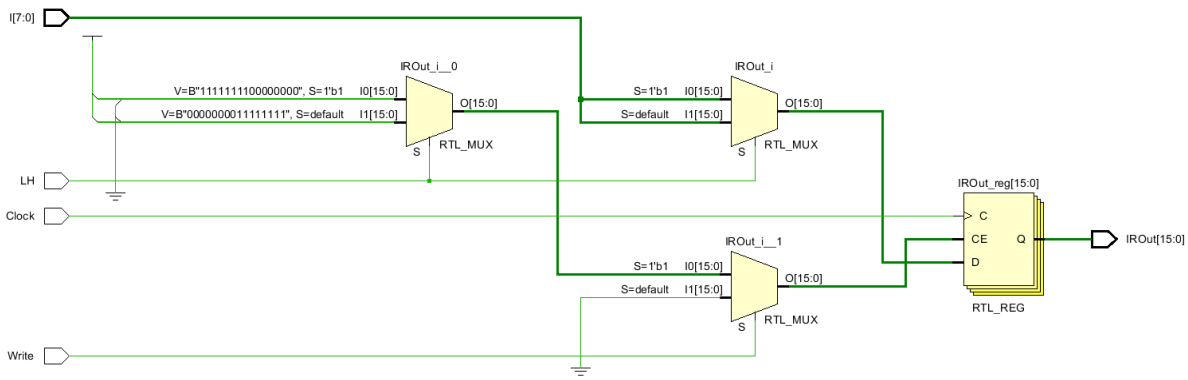

Figure 21: RTL design of instruction register

L'H	Write	IR^+
ϕ	0	IR (retain value)
0	1	IR (7-0) ← I (Load LSB)
1	1	IR (15-8) ← I (Load MSB)

Figure 22: Function Selector of instruction register

4.2.2 PART-2(b)

In this part, we use the general-purpose registers that we created in part 1 along with their functions, as described in the FunSel table from Figure 20. Figure 24 and Figure 25 determine the enable input that will be used in the registers. Here, the leftmost digit represents R1, the second leftmost R2, and so on, similar to the Scratch register (S). By taking the complement of the given four-bit inputs RegSel and ScrSel and assigning them to their corresponding places, we can determine the enable signal for each register. Other than that, "I" input, FunSel and Clock have the same responsibilities as in the previous parts. In the end, we have two 16-bit outputs. OutASel and OutBSel from Figure 26 determine which register's output will be our actual output.

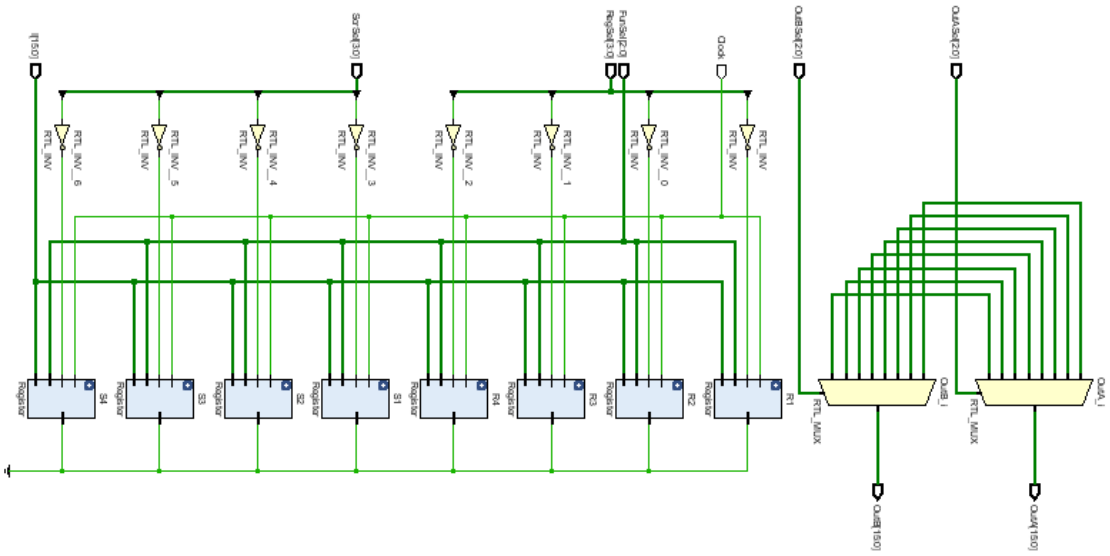

Figure 23: RTL design of register file

RegSel	Enable General Purpose Registers	RegSel	Enable General Purpose Registers
0000	All general purpose registers are enabled. (Function selected by FunSel will be applied to R1, R2, R3 and R4.)	1000	R2, R3, and R4 are enabled. (Function selected by FunSel will be applied to R2, R3, and R4.)
0001	R1, R2 and R3 are enabled. (Function selected by FunSel will be applied to R1, R2, and R3.)	1001	R2 and R3 are enabled. (Function selected by FunSel will be applied to R2 and R3.)
0010	R1, R2, and R4 are enabled. (Function selected by FunSel will be applied to R1, R2, and R4.)	1010	R2 and R4 are enabled. (Function selected by FunSel will be applied to R2 and R4.)
0011	R1 and R2 are enabled. (Function selected by FunSel will be applied to R1 and R2.)	1011	Only R2 is enabled. (Function selected by FunSel will be applied to R2.)
0100	R1, R3, and R4 are enabled. (Function selected by FunSel will be applied to R1, R3, and R4.)	1100	R3 and R4 are enabled. (Function selected by FunSel will be applied to R3 and R4.)
0101	R1 and R3 are enabled. (Function selected by FunSel will be applied to R1 and R3.)	1101	Only R3 is enabled. (Function selected by FunSel will be applied to R3.)
0110	R1 and R4 are enabled. (Function selected by FunSel will be applied to R1 and R4.)	1110	Only R4 is enabled. (Function selected by FunSel will be applied to R4.)
0111	Only R1 is enabled. (Function selected by FunSel will be applied to R1.)	1111	NO general purpose register is enabled. (All registers retain their values.)

Figure 24: Enable Selection for R1,R2,R3,R4

ScrSel	Enable General Purpose Registers	ScrSel	Enable General Purpose Registers
0000	All general purpose registers are enabled. (Function selected by FunSel will be applied to S1, S2, S3, and S4.)	1000	S2, S3, and S4 are enabled. (Function selected by FunSel will be applied to S2, S3, and S4.)
0001	S1, S2, and S3 are enabled. (Function selected by FunSel will be applied to S1, S2, and S3.)	1001	S2 and S3 are enabled. (Function selected by FunSel will be applied to S2 and S3.)
0010	S1, S2, and S4 are enabled. (Function selected by FunSel will be applied to S1, S2, and S4.)	1010	S2 and S4 are enabled. (Function selected by FunSel will be applied to S2 and S4.)
0011	S1 and S2 are enabled. (Function selected by FunSel will be applied to S1 and S2.)	1011	Only S2 is enabled. (Function selected by FunSel will be applied to S2.)
0100	S1, S3, and S4 are enabled. (Function selected by FunSel will be applied to S1, S3, and S4.)	1100	S3 and S4 are enabled. (Function selected by FunSel will be applied to S3 and S4.)
0101	S1 and S3 are enabled. (Function selected by FunSel will be applied to S1 and S3.)	1101	Only S3 is enabled. (Function selected by FunSel will be applied to S3.)
0110	S1 and S4 are enabled. (Function selected by FunSel will be applied to S1 and S4.)	1110	Only S4 is enabled. (Function selected by FunSel will be applied to S4.)
0111	Only S1 is enabled. (Function selected by FunSel will be applied to S1.)	1111	NO general purpose register is enabled. (All registers retain their values.)

Figure 25: Enable Selection for S1,S2,S3,S4

OutASel	OutA
000	R1
001	R2
010	R3
011	R4
100	S1
101	S2
110	S3
111	S4

OutBSel	OutB
000	R1
001	R2
010	R3
011	R4
100	S1
101	S2
110	S3
111	S4

Figure 26: Output Selection for OutA and OutB

4.2.3 PART-2(c)

In this part, we use the general-purpose registers that we created in part 1 along with their functions, as described in the FunSel table from Figure 20. Figure 28 determine the enable input that will be used in the registers. Here, the leftmost digit represents PC, the second leftmost AR, and so on. "I" input, FunSel and Clock have the same responsibilities as in the previous parts. In the end, we have two 16-bit outputs. OutCSel and OutDSel from Figure 29 determine which register's output will be our actual output.

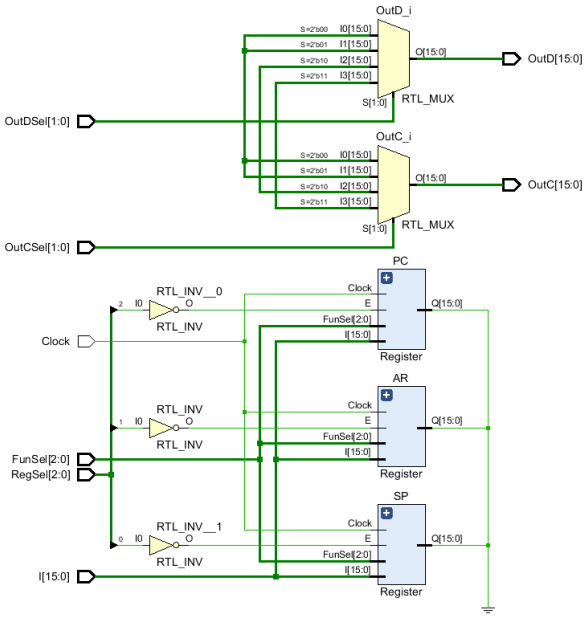

Figure 27: RTL design of address register file

RegSel	Enable Address Registers
000	All address registers are enabled. (Function selected by FunSel will be applied to PC, AR, and SP.)
001	PC and AR are enabled. (Function selected by FunSel will be applied to PC and AR.)
010	PC and SP are enabled. (Function selected by FunSel will be applied to PC and SP.)
011	PC is enabled. (Function selected by FunSel will be applied to PC.)
100	AR and SP are enabled. (Function selected by FunSel will be applied to AR and SP.)
101	AR is enabled. (Function selected by FunSel will be applied to AR.)
110	SP is enabled. (Function selected by FunSel will be applied to SP.)
111	NO address register is enabled. (All registers retain their values.)

Figure 28: Enable Selection for PC, AR and SP

OutCSel	OutC
00	PC
01	PC
10	AR
11	SP

OutDSel	OutD
00	PC
01	PC
10	AR
11	SP

Figure 29: Output Selection for OutC and OutD

4.3 PART-3

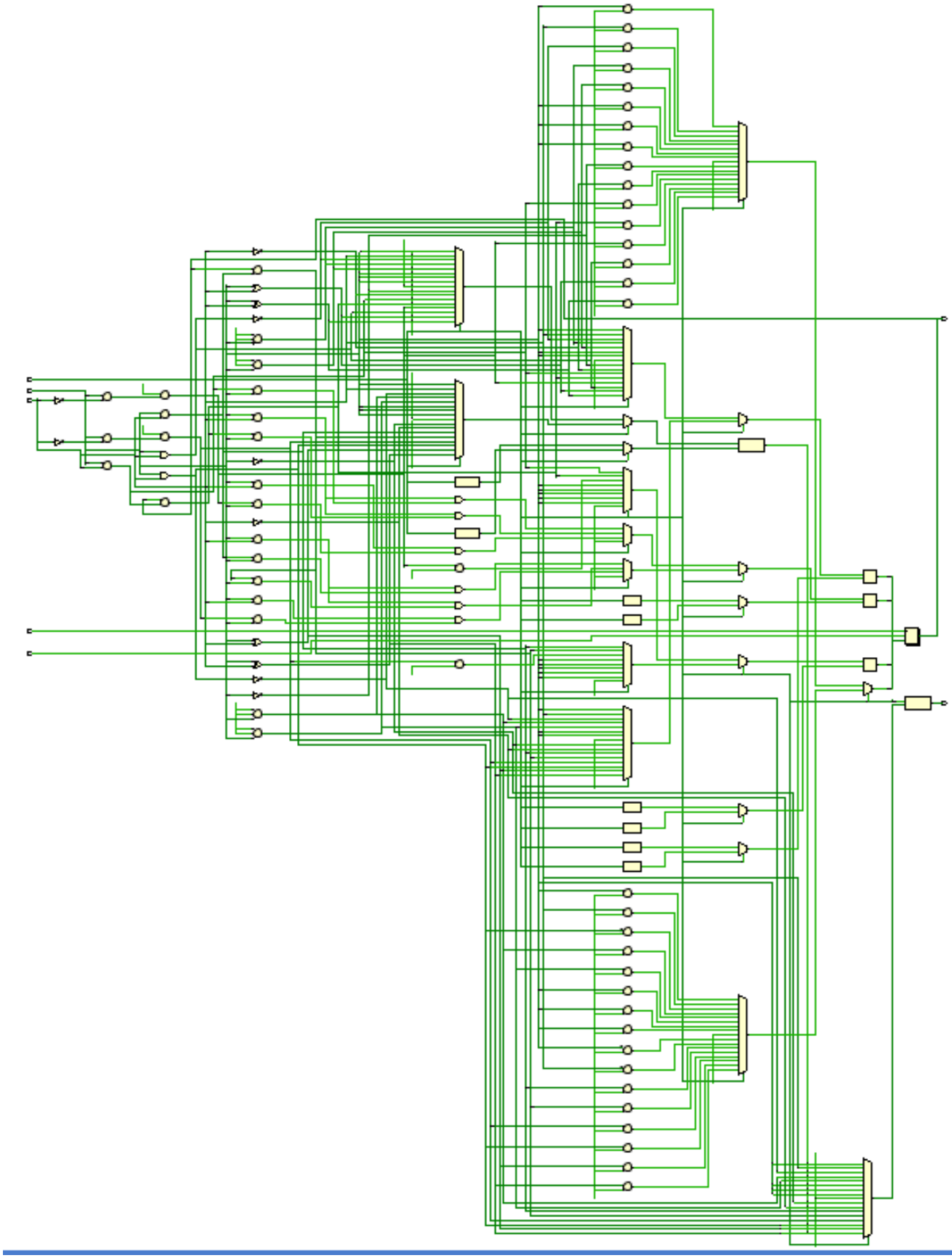

Figure 30: RTL design of arithmetic logic unit

FunSel	ALUOut	Z	C	N	O
00000	A (8-bit)	+	-	+	-
00001	B (8-bit)	+	-	+	-
00010	NOT A (8-bit)	+	-	+	-
00011	NOT B (8-bit)	+	-	+	-
00100	A + B (8-bit)	+	+	+	+
00101	A + B + Carry (8-bit)	+	+	+	+
00110	A – B (8-bit)	+	+	+	+
00111	A AND B (8-bit)	+	-	+	-
01000	A OR B (8-bit)	+	-	+	-
01001	A XOR B (8-bit)	+	-	+	-
01010	A NAND B (8-bit)	+	-	+	-
01011	LSL A (8-bit)	+	+	+	-
01100	LSR A (8-bit)	+	+	+	-
01101	ASR A (8-bit)	+	+	-	-
01110	CSL A (8-bit)	+	+	+	-
01111	CSR A (8-bit)	+	+	+	-

FunSel	ALUOut	Z	C	N	O
10000	A (16-bit)	+	-	+	-
10001	B (16-bit)	+	-	+	-
10010	NOT A (16-bit)	+	-	+	-
10011	NOT B (16-bit)	+	-	+	-
10100	A + B (16-bit)	+	+	+	+
10101	A + B + Carry (16-bit)	+	+	+	+
10110	A – B (16-bit)	+	+	+	+
10111	A AND B (16-bit)	+	-	+	-
11000	A OR B (16-bit)	+	-	+	-
11001	A XOR B (16-bit)	+	-	+	-
11010	A NAND B (16-bit)	+	-	+	-
11011	LSL A (16-bit)	+	+	+	-
11100	LSR A (16-bit)	+	+	+	-
11101	ASR A (16-bit)	+	+	-	-
11110	CSL A (16-bit)	+	+	+	-
11111	CSR A (16-bit)	+	+	+	-

Figure 31: Function Selector of arithmetic logic unit

In Figure 5, we have two 16-bit inputs on which we will perform arithmetic or logic operations. After the operation, we have two outputs: one is a 16-bit output, and the other is a 4-bit flags output. These flags are defined as Zero, Carry, Negative, and Overflow. These flags are not affected by every operation; some operations affect some flags. This is shown with + and - signs in Figure 31. The operations we will perform are also shown in Figure 31. The 16-bit output appears whenever an operation is performed, unlike the flags output. Flags outputs are updated only when the WF flag (write flag) is one. The ALU takes the carry-in input from the flags. In substraction operation carry is actually borrow.(i.e.if carry is zero borrow is one).

4.4 PART-4

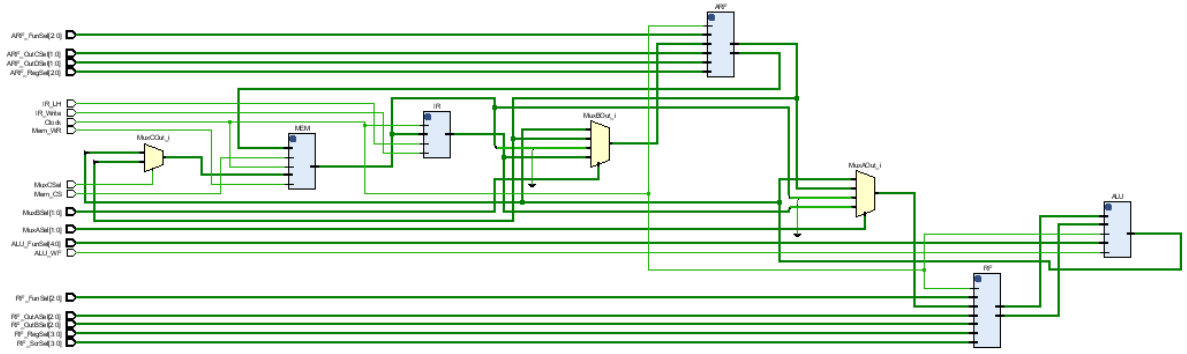

Figure 32: RTL design of ALU system

Table 7: Multiplexers of Arithmetic Logic Unit Systems.

MuxASel	MuxAOut
00	ALUOut
01	ARF OutC
10	Memory Output
11	IR (7:0)

MuxBSel	MuxBOut
00	ALUOut
01	ARF OutC
10	Memory Output
11	IR (7:0)

MuxCSel	MuxCOut
0	ALUOut(7-0)
1	ALUOut(15-8)

Figure 33: Characteristic of MUXa-MUXB-MUXC

In Figure 32, there are 19 inputs: 4 are related to the Address File Register (ARF) explained in Part 2c, 2 are related to the Instruction Register (IR) explained in Part 2a, 2 are related to memory (MEM), which is given, 3 are inputs for Multiplexers A, B, and C, 2 are inputs for the Arithmetic Logic Units (ALU), 5 are inputs for the Register Files (RF), and 1 is an input for the Clock signal, which is common to every register. As shown in Figure 32, the inputs "I" are taken from the output of other components. Specifically, RF's input is taken from MuxAout, ALU's inputs are taken from RF's output, ARF's input is taken from MuxBOut, IR's input is taken from MEM's output, and MEM's inputs (address) are taken from ARF's output, with the other input of MEM taken from MuxCOut (which is selected from ALU's output according to Figure 33 MuxCSel).

In Figure 33, MuxA takes its inputs respectively from the ALU, ARF-OutC, MEM's output, and IR(7:0). MuxB has the same characteristics.

5 CONCLUSION

We completed each part within one or two days consistently. The parts were challenging but also instructive. When we fully understood a part, we grasped the logic behind it. Verilog has a user-friendly and understandable syntax that we were already familiar with from C++. However, Vivado is not compatible with Macbooks, so we had to continue the process with only one computer. This issue wasted time and reduced our workforce.

Additionally, this project is challenging because it requires us to think at the hardware level, which differs significantly from the software level. Because of these differences, we created a more complex system in the ALU.

Lastly, each group member participated equally, and we accomplished all design and report preparation processes together.

6 REFERENCES

REFERENCES

[1] Frank Vahid and Roman Lysecky. *Verilog For Digital Design.* John Wiley and Sons, 2007.

[2] Bob Zeidman. *Verilog Designer's Library.* Prentice Hall PTR, 1999.