ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E
COMPUTER ORGANIZATION PROJECT 1 REPORT

CRN	:	21336
LECTURER	:	Prof. Dr. Mustafa Ersel Kamaşak

GROUP MEMBERS:

150210030	:	EMRE BOZKURT
150210022	:	HARUN YAHYA DEMİRPENÇE

SPRING 2024

Contents

1 INTRODUCTION

In this project, we used Verilog HDL to design and build a simple computer system. The system includes an ALU for arithmetic and logical operations, a register file for general-purpose and scratch registers, an address register file for the program counter and address register, and a 16-bit register with eight functions. Each component was developed as a separate Verilog module, underwent simulation testing, and then integrated into a functional whole. This project demonstrates the design and implementation of essential computer system components, serving as a practical exercise in computer organization concepts.

1.1 Task Distribution

We designed Part-1 and Part-3 together. Part-2a and Part-2b were done by Harun Yahya Demirpençe. Part-2c and Part-4 were done by Emre Bozkurt.

2 MATERIALS AND METHODS

2.1 Design and Development Tools

The project was developed using the Verilog hardware description language. Code development, simulation, and Verilog module synthesis were carried out using the Vivado Design Suite.

2.2 Design Approach

To ensure modularity and reusability, each component of the project was designed as a separate Verilog module. Initially, we designed a 16-bit register with 8 functions. Subsequently, we designed a register file, a 16-bit instruction register (IR), an address register file (ARF), an arithmetic logic unit (ALU), and an ALU system.

2.2.1 Part-1

In the first part, we were tasked with designing a 16-bit register. This register module serves as a fundamental building block for storing data within our system. It takes an 8-bit I input, a 1-bit enable input to enable the register, and a 3-bit FunSel input to select from the following 8 different functionalities:

- Increment, decrement
- Load, clear
- Write low, clear high
- Only write low, only write high
- Write low, sign extend

This module will be used in the register file and the address register file in the following parts. The elaborated design for the implementation of the register can be seen in Figure 1.

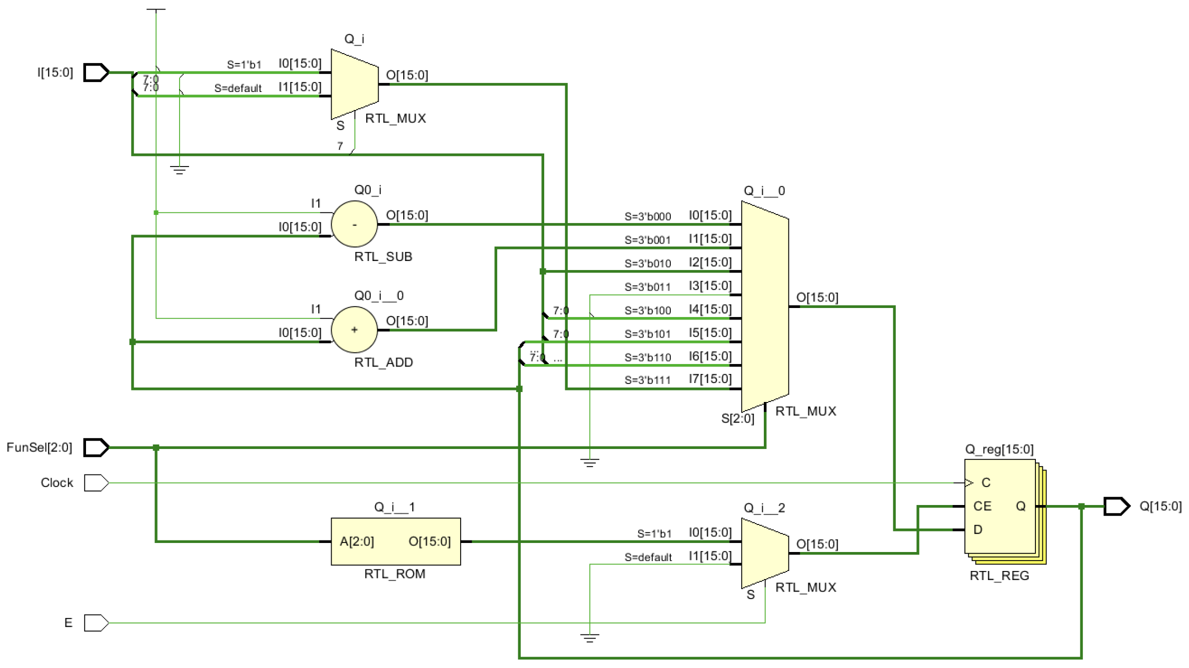


Figure 1: Schematic for 16-bit General Purpose Register

2.2.2 Part-2

- **Part-2a**

  We've designed a 16-bit instruction register in this part. The instruction register serves as a critical component for storing instructions within our system, particularly for the arithmetic logic unit system. The instruction register module features functionality to load either the first 8 bits or the last 8 bits of an instruction, depending on the value of the L'H input. Additionally, it supports the capability to write 8-bit data into the remaining portion of the register. When the write input is low, the instruction register retains its current value. This module will be used in the arithmetic logic unit system. The elaborated design for the instruction register is in Figure 2.

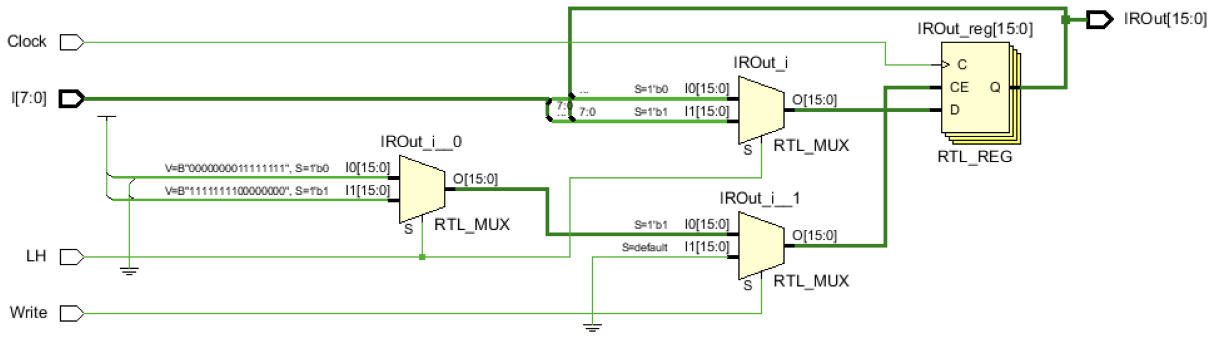


Figure 2: Schematic for 16-bit Instruction Register

- **Part-2b**

  In this part of the project, we have implemented a register file system consisting of four 16-bit general-purpose registers and four 16-bit scratch registers. The register file system features 16-bit RegSel and ScrSel inputs for selecting which register/s are enabled, allowing dynamic control over register access. Additionally, it supports inputs for driving register outputs to the system output based on the given selection criteria with 3-bit OutASel and OutBSel inputs. This module will be used in the arithmetic logic unit system. The elaborated design for the register file system can be seen in Figure 3.

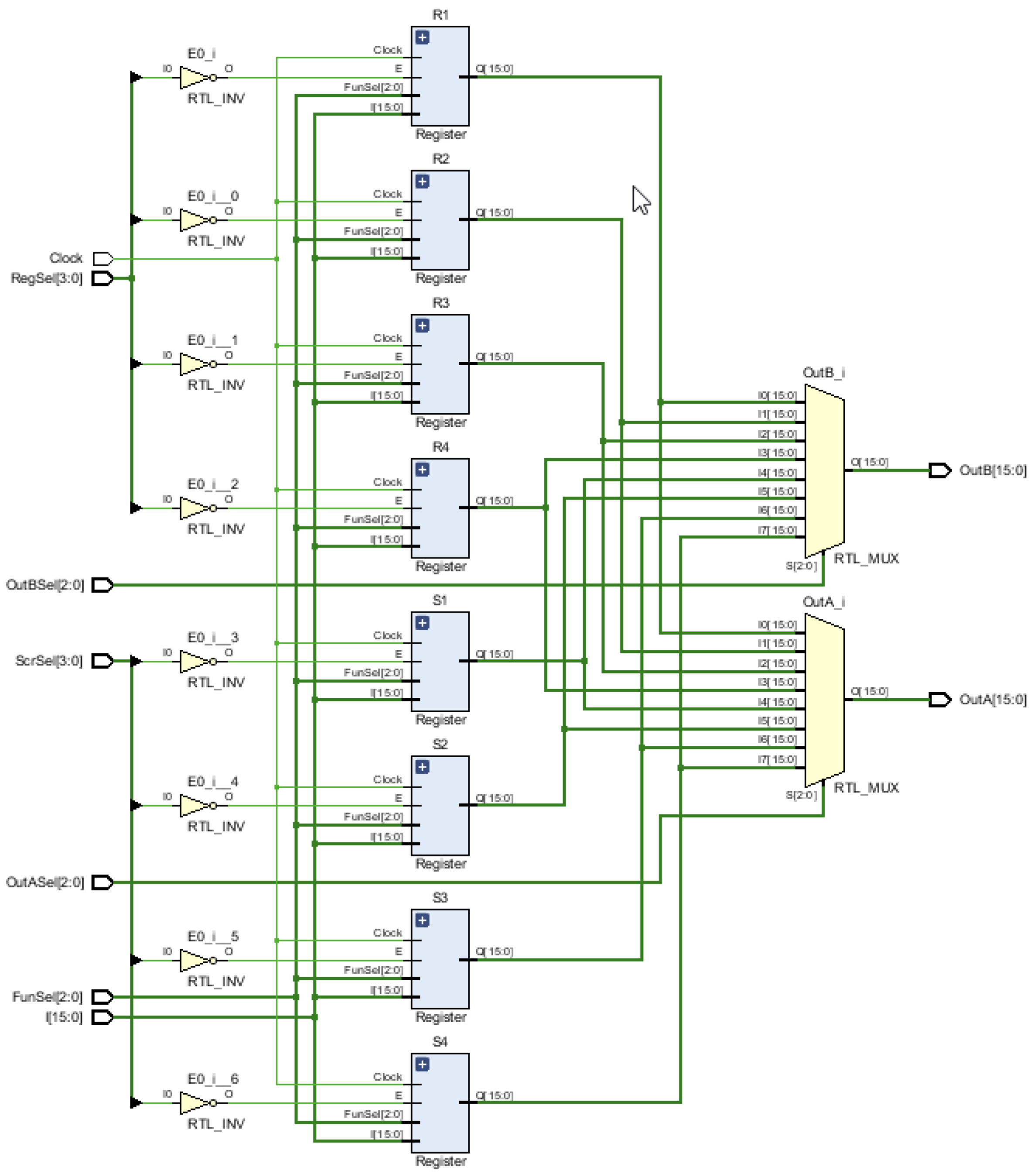


Figure 3: Schematic for Register File

- **Part-2c**

  In this part, we have implemented the address register file system. The address register file system serves as a crucial component for managing memory addresses and control flow within our system. The system consists of 3 16-bit address registers: program counter, address register, stack pointer. The 3-bit RegSel input determines which register will be enabled. The 2-bit OutCSel and OutDSel inputs drive the selected register output to the system output. This module will be used in the arithmetic logic unit system. The elaborated design for the address register file system can be seen in Figure 4.

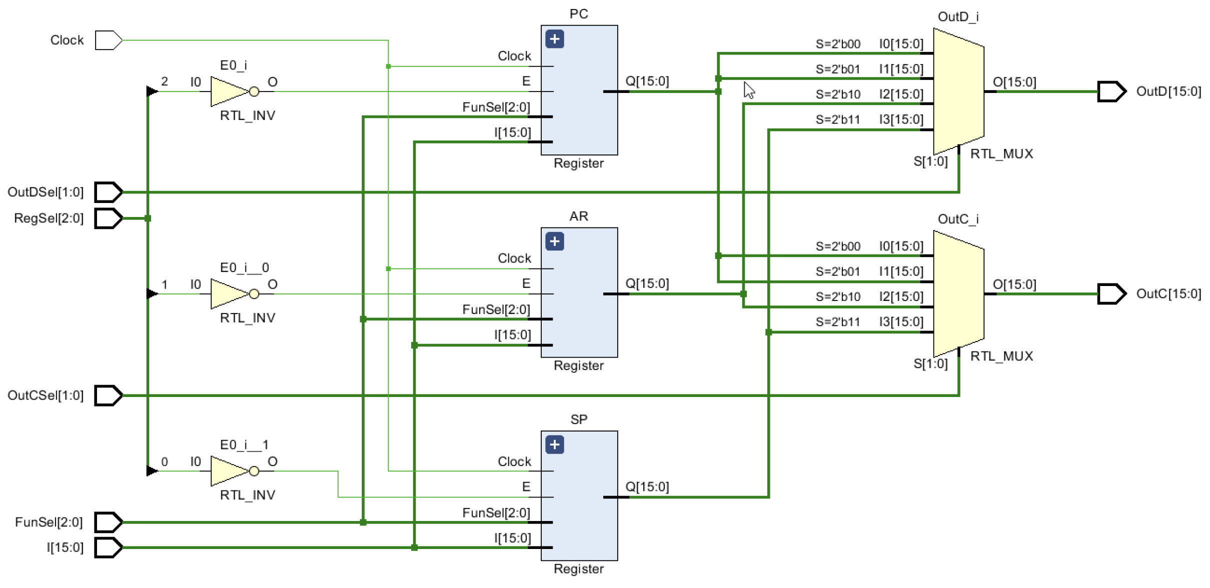


Figure 4: Schematic for Address Register File

2.2.3 Part-3

We are requested to design an arithmetic logic unit in part-3. The ALU is responsible for performing various arithmetic and logic operations on input data. The ALU module takes two 16-bit inputs, A and B, and a 5-bit ALU operation select input, FunSel. The FunSel input determines which operation the ALU should perform from the following operations.

- The MSB of FunSel determines whether the operation is 8-bit or 16-bit.
- Buffer A, B
- Negation A, B
- $A + B, A + B + Carry, A - B$

- $A \wedge B, A \vee B, A \oplus B, A$ NAND B
- Logical, arithmetic, circular shift operations

The ALU produces a 16-bit output, ALUOut represents the result of the selected operation. A 4-bit FlagsOut output is also updated if the input WF is 1, and synchronously with the clock signal according to the ALUOut value and selected operation.

- Z is the MSB, and O is the LSB.
- Z bit is set if ALUOut equals zero.
- C bit is set if ALUOut produces carry.
- N bit is set if ALUOut is negative.
- O bit is set if overflow occurs.

This module will be used in the arithmetic logic unit system. The elaborated design for the arithmetic logic unit can be seen in Figure 5.

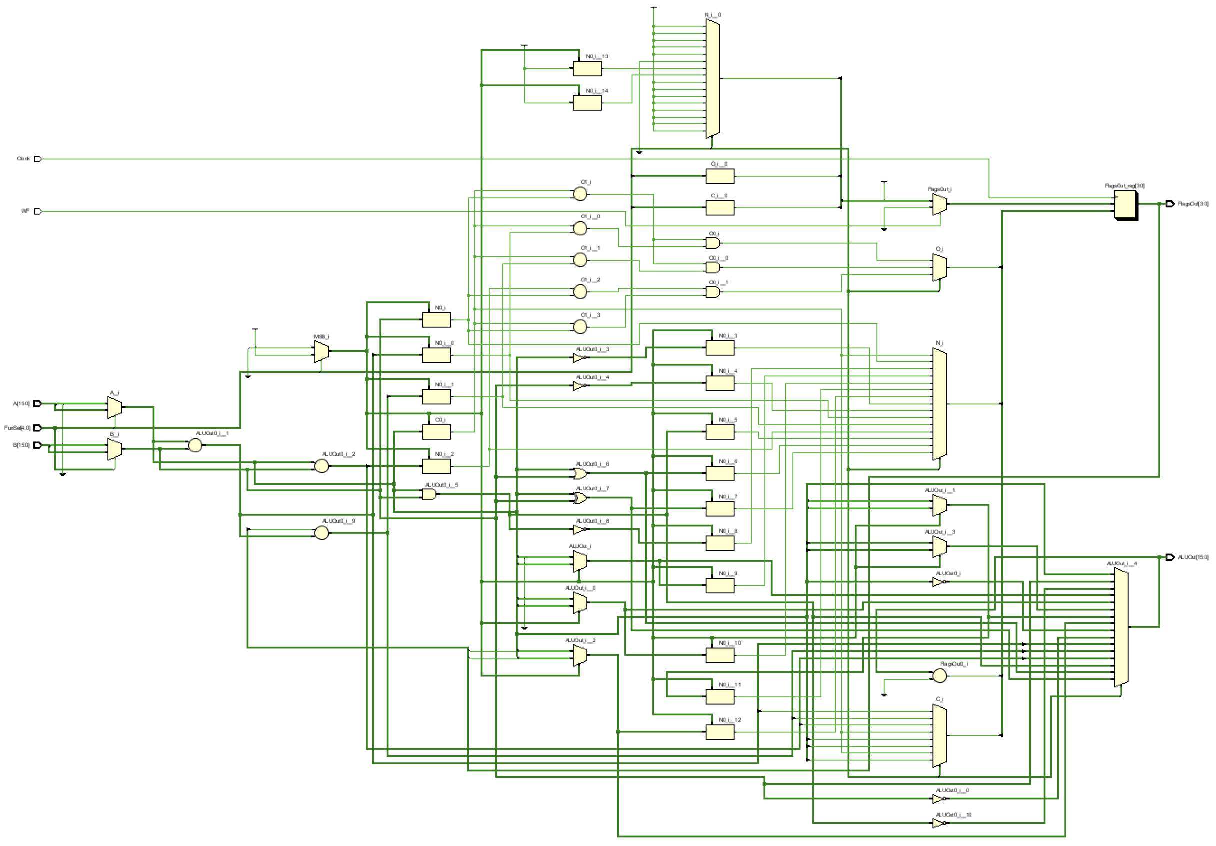


Figure 5: Schematic for Arithmetic Logic Unit

2.2.4 Part-4

In section 4, we are asked to design an arithmetic logic unit system. For the entire computer organization project, the ALU system is essential. To ensure synchronized functioning throughout the system, it runs on a single clock signal.

We have previously designed and implemented a number of modules that are required to implement the ALU system, such as the register, register file, address register file, instruction register, and ALU. These modules make up the fundamental parts needed to run the ALU system. At this point, in order to enable data routing within the ALU system, we must incorporate these components by putting the multiplexers (MuxA, MuxB, and MuxC) into place:

- MuxA: The input for the ALU's first operand is chosen by this multiplexer. Depending on the MuxASel control signal, we will set MuxA to choose amongst ALUOut, ARF output C, MemOut, and the lower 8 bits of IROut.
- MuxB: MuxB chooses the input for the second operand of the ALU, much like MuxA does. Depending on the MuxBSel control signal, we will set MuxB to choose amongst ALUOut, ARF output C, MemOut, and the least significant 8 bits of IROut.
- MuxC: MuxC selects between different parts of the ALU output. We will configure MuxC to select between the lower 8 bits ALUOut or the upper 8 bits ALUOut based on the MuxCSel control signal.

Keep in mind that extending memory output with zeros for the third input of MuxA and MuxB is required.

The ALU System will also communicate with the memory to retrieve information for processing. The memory output, which will supply the required data input for the ALU operations, will enable this connection.

We will finish the ALU system integration by putting these multiplexers into practice and making use of the memory output, which will enable effective data processing and manipulation inside the computer organization. The elaborated design for the arithmetic logic unit system can be seen in Figure 6

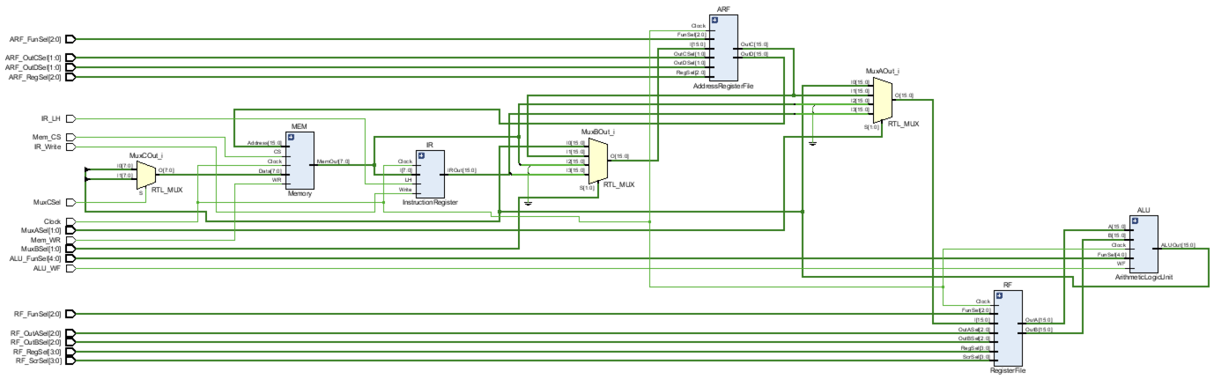


Figure 6: Schematic for Arithmetic Logic Unit System

2.3 Integration

Once all individual modules were verified, we integrated them into a complete system. Further simulations were conducted using Vivado to verify the interaction between modules and to ensure that the integrated system produced the desired results.

3 RESULTS

3.1 Simulation and Testing

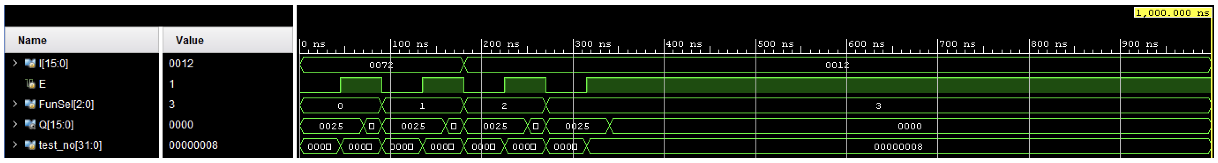
After implementing each module, we conducted simulations using Vivado to verify its functionality. Testbench files were used to provide input stimuli and to monitor the output signals, ensuring that each module performed as expected.

3.2 Testing and Validation

The complete system was tested using various test cases to validate its functionality. Testing ensured that the system could perform arithmetic and logical operations, store and retrieve data from memory and registers, and execute instructions sequentially.

3.3 Simulation Results

3.3.1 Part-1



Name	Value
I[15:0]	0012
E	1
FunSel[2:0]	3
Q[15:0]	0000
test_no[31:0]	00000008

Figure 7: Test Results for Register Simulation

3.3.2 Part-2a

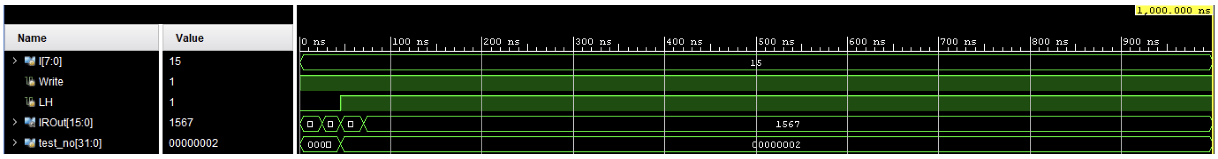


Figure 8: Test Results for Instruction Register Simulation

3.3.3 Part-2b

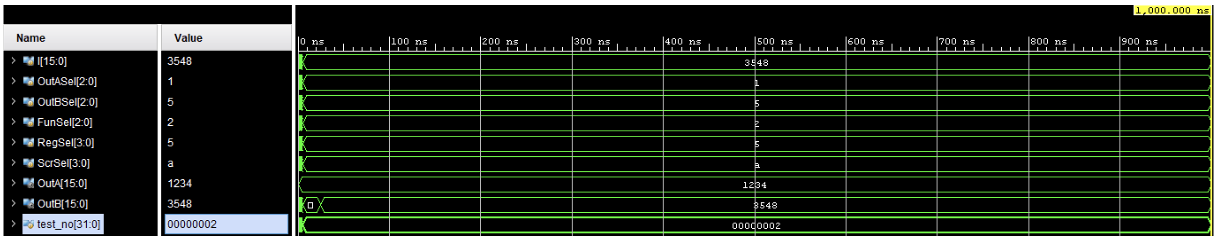


Figure 9: Test Results for Register File Simulation

3.3.4 Part-2c

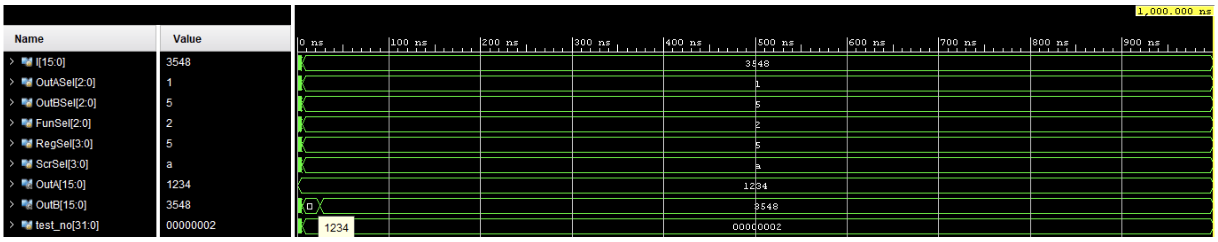


Figure 10: Test Results for Address Register File Simulation

3.3.5 Part-3

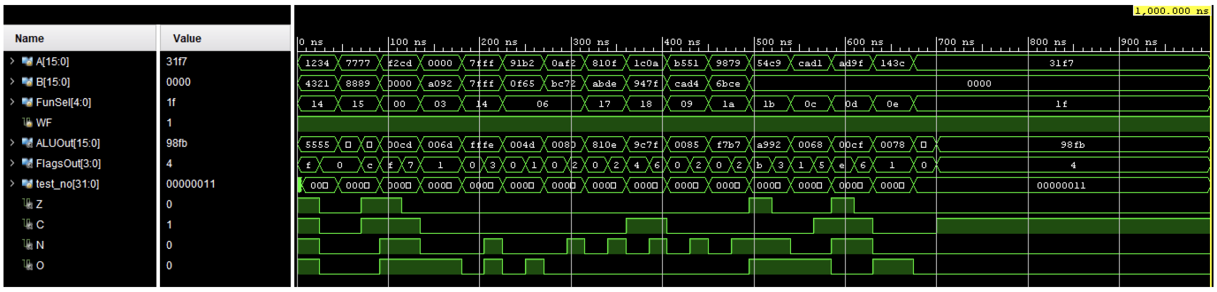
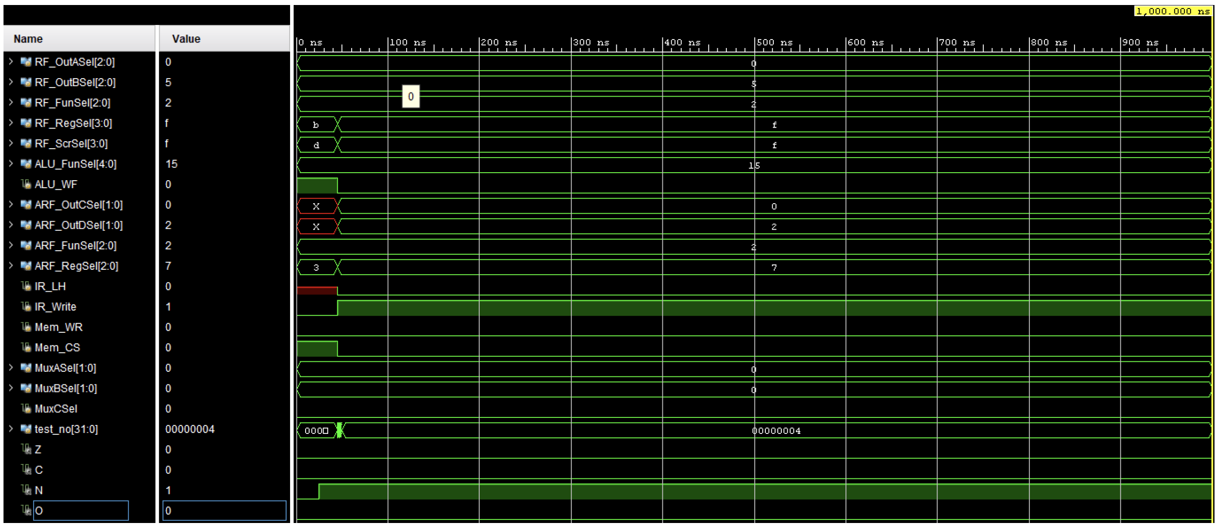


Figure 11: Test Results for Arithmetic Logic Unit Simulation

3.3.6 Part-4



```
Name               Value
RF_OutASel[2:0]    0
RF_OutBSel[2:0]    5
RF_FunSel[2:0]     2
RF_RegSel[3:0]     f
RF_ScrSel[3:0]     f
ALU_FunSel[4:0]    15
ALU_WF             0
ARF_OutCSel[1:0]   0
ARF_OutDSel[1:0]   2
ARF_FunSel[2:0]    2
ARF_RegSel[2:0]    7
IR_LH              0
IR_Write           1
Mem_WR             0
Mem_CS             0
MuxASel[1:0]       0
MuxBSel[1:0]       0
MuxCSel            0
test_no[31:0]      00000004
Z                  0
C                  0
N                  1
O                  0
```

Figure 12: Test Results for Arithmetic Logic Unit System Simulation

4 DISCUSSION

In this project, we successfully used Verilog HDL to design and develop a basic computer system that included several necessary parts, including an ALU system, an address register file, an instruction register, a 16-bit register, a register file, and an arithmetic logic unit (ALU). To guarantee modularity and reusability, every component was created as a distinct Verilog module. Each module was created and tested separately before being merged into the overall system as part of a methodical process used in the design process. This method assisted in finding and fixing any problems early on in the development process. Creating the ALU and ALU system was one of the main difficulties encountered in the project. The numerous arithmetic and logical operations that these components had to support, in addition to managing overflow and carry operations, all required careful thought. We carried out extra tests in addition to the provided simulations to guarantee the ALU's proper operation. Arithmetic operations, logical operations, and shift operations were tested on these assessments. We also examined the ALU's capacity to manage overflow and perform operations accurately.

All things considered, this project gave us practical experience utilizing Verilog HDL to design and build computer system components. It aided in the comprehension of computer organization principles and the significance of modular design in complex architectures. Notably, we devised new test scenarios for the ALU, which are detailed in the Table 1, while adhering to the provided test files for other project components.

	Inputs						Outputs	
Test No	A (16-bit)	B (16-bit)	FunSel	WF	FlagsOut	CLK	ALUOut	FlagsOut
1	1234	4321	10100	1	1111	0	5555	1111
2	X	X	X	X	X	1	5555	0000
3	7777	8889	10101	1	0000	1	0001	1100
4	F2CD	0000	00000	1	1111	1	00CD	0111
5	0000	A092	00011	1	0001	1	006D	0001
6	7FFF	7FFF	10100	1	0000	1	FFFE	0011
7	91B2	0F65	00110	1	0000	1	004D	0001
8	0AF2	BC72	00110	1	0000	1	0080	0010
9	810F	ABDE	10111	1	0000	1	810E	0010
10	1C0A	947F	11000	1	0100	1	9C7F	0110
11	B551	CAD4	01001	1	0000	1	0085	0010
12	9879	6BCE	11010	1	0000	1	F7B7	0010
13	54C9	0000	11011	1	1011	1	A992	0011
14	CAD1	0000	01100	1	0001	1	0068	0101
15	AD9F	0000	01101	1	1110	1	00CF	0110
16	143C	0000	01110	1	0001	1	0078	0001
17	31F7	0000	11111	1	0000	1	98FB	0100

Table 1: ALU Test Cases

5 CONCLUSION

To sum up, at the beginning of our project, we faced difficulties when installing Vivado on MacOS, which we overcame with the assistance of friends. We think that if documentation on how to install Vivado on MacOS is prepared, it will be very useful to students who will take this course in the coming semesters. About designing the project, we didn't face any problems until part-3, the ALU part. The first problem was how can we convert 16-bit numbers to 8-bit numbers. Another problem with the ALU design was how to get the carry flag from arithmetic operations. Additionally, a further difficulty was integrating the ALU system with other parts including the memory, instruction register, and register file. For the project to function as a whole, it was vital that data be transferred between these parts accurately and that the ALU performed synchronously with the rest of the system. We solved the issues thanks to co-working and from the Ninova message board. After all these difficulties, this project taught us how to use Vivado, write code with Verilog HDL, and how group work is important.

REFERENCES