ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E

COMPUTER ORGANIZATION
PROJECT 1 REPORT

CRN : 21335 - 21336

LECTURER : Deniz Turgay Altılar, Mustafa Ersel Kamaşak

GROUP MEMBERS:

150220046 : Emre Çağrı Dalcı

150210090 : Emre Safa Yalçın

SPRING 2024

Contents

1 INTRODUCTION **1**

1.1 **Task Distribution** 1

2 MATERIALS AND METHODS **1**

2.1 PART 1 1

2.2 PART 2-A 2

2.3 PART 2-B 3

2.4 PART 2-C 4

2.5 PART 3 5

2.6 PART 4 7

3 RESULTS **9**

3.1 PART 1 9

3.2 PART 2-A 9

3.3 PART 2-B 10

3.4 PART 2-C 10

3.5 PART 3 11

3.6 PART 4 11

4 DISCUSSION **12**

4.1 Discussion on the `Register.v` Module 12

4.1.1 Operational Overview 12

4.1.2 Functional Selector Analysis 12

4.1.3 Implications and Applications 12

4.1.4 Conclusion 13

4.2 Discussion on the `InstructionRegister.v` module 13

4.2.1 Implementation Details 13

4.3 Discussion on the `RegisterFile.v` Module 14

4.3.1 Module Overview 14

4.3.2 Inputs, Outputs, and Functional Mapping 14

4.3.3 Functional Behavior 15

4.4 Discussion on the `AddressRegisterFile.v` Module 15

4.4.1 Module Overview 16

4.4.2 Inputs, Outputs, and Functional Mapping 16

4.4.3 Functional Behavior 16

4.5 Discussion on the `ArithmeticLogicUnit.v` Module 17

4.5.1 Module Inputs and Outputs 17

4.5.2 Operational Modes and Function Selection 17
4.5.3 Flags and Their Implications 19
4.5.4 Conclusion 19
4.6 Discussion on the `ArithmeticLogicUnitSystem.v` Module 19
4.6.1 System Overview 19
4.6.2 Operational Behavior 20

5 Conclusion **20**

1 INTRODUCTION

In this project, we have started the exciting journey of building a simple computer system using Verilog, which is a language used to describe how electronic systems work. Our main goal was to design and test parts of a computer that can do basic math and logic operations. We broke down our project into four main parts, each focusing on different pieces of the computer, like registers and an Arithmetic Logic Unit (ALU). Registers are like small storage spaces in the computer where we can keep numbers for a short time, and the ALU is the part that does all the calculations. First, we worked on creating a 16-bit register that can do eight different operations, like adding or subtracting one from a number or storing a number. This register is very important because we use it in the other parts of our project. Then, we made a group of registers that can work together to hold and manage data. After that, we focused on the ALU, which is where all the action happens. It can do basic math like adding and subtracting, and it can also do logic operations that help the computer make decisions. We tested each part by giving them different inputs to make sure they work as expected. The aim was to have a system where we can not only do calculations and logic operations but also save these results in memory or registers and use them later. We hope that by the end of our tests, everything works smoothly.

1.1 Task Distribution

When we were preparing this project, we did almost everything together. We tried to implement and build each part individually. There is no distribution of task.We tried to find solutions to all the problems we encounter while building and testing the components. That is why each of us participated in every part of the project.

2 MATERIALS AND METHODS

2.1 PART 1

In the first stage of our project, we designed and built a complex 16-bit register, which is controlled by 3-bit control signals (named FunSel) and an enable signal (E). This important register will be used in later stages for building more advanced modules. It has eight different functions, controlled by the FunSel input, which include keeping the same value, decreasing, increasing, loading, and clearing, along with specific bit-level operations as shown in the characteristic table. To construct the design of the register, we employed the "parameter" keyword in Verilog to accurately describe the module's behavior. The

detailed design of this 16-bit register is shown in the schematic below, highlighting the register's flexibility and its crucial role in the upcoming parts of our design:

Control Inputs	Description
OutCSel	2-bit selector to choose the output for OutC
OutDSel	2-bit selector to choose the output for OutD
FunSel	3-bit selector to choose the function operation on the input data
RegSel	3-bit selector to enable writing to the PC, AR, or SP registers

Figure 1: schematic for general registers

FunSel (Binary)	Function Description
000	Decrement Q by 1
001	Increment Q by 1
010	Load Q with input I (16-bit)
011	Clear Q (set all bits to 0)
100	Load Q[7:0] with I[7:0] and clear Q[15:8]
101	Load Q[7:0] with I[7:0] (keep Q[15:8] unchanged)
110	Load Q[15:8] with I[7:0] (keep Q[7:0] unchanged)
111	Sign-extend I[7:0] to Q[15:8] and load I[7:0] to Q[7:0]

Figure 2: Control Inputs and Corresponding Functions for general registers

2.2 PART 2-A

In the second stage of our project, we created and put into place a 16-bit Instruction Register (IR). This register, like the 16-bit register we mentioned earlier, has four main functions when it is used: clear, load, decrement, and increment. It can hold 16-bit binary data but has only an 8-bit data input. To manage this, we added another input to the IR called L'H, which decides if the write operation affects the lower (bits 0-7) or upper (bits 8-15) half of the register. In designing this module, we used behavioral Verilog and the 'parameter' keyword to define how the register works. We also used an 'if statement' to ensure the operation targets the correct half of the register. The IR is mainly used to store the binary instruction that the CPU fetches and holds temporarily. These instructions are in binary code and tell the CPU what action to perform. Below, you can see the schematic that illustrates how we implemented the IR:

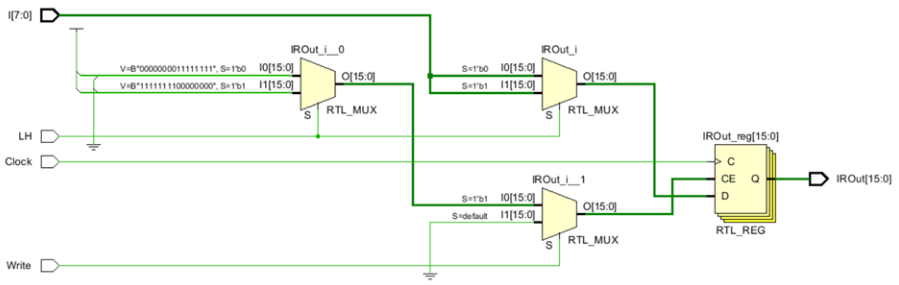

Figure 3: schematic for Instruction Register

Write	LH	Function Description
0	X	No operation (IROut remains unchanged)
1	0	Load IROut[7:0] with the 8-bit input I
1	1	Load IROut[15:8] with the 8-bit input I

Figure 4: Control Inputs and Corresponding Functions for Instruction Registers

2.3 PART 2-B

In the current stage of our design, we have constructed a Register File which includes four main registers, named R1, R2, R3, and R4, and four additional registers for temporary storage, labeled S1, S2, S3, and S4. This setup, illustrated in Figure 3, involves seven key inputs: the data input (I), two selection signals for the output registers (OutASel and OutBSel), two selectors for determining the register operation (RegSel and ScrSel), a function selector (FunSel), and the clock signal. Each register can be engaged through a specific combination of selection signals, allowing us to direct where and how the data will be processed. For instance, using the OutASel and OutBSel, we can dictate which register's content is sent to the output lines OutA and OutB. The RegSel and ScrSel signals, composed of four bits each, enable the registers selectively, allowing the function set by FunSel to be applied appropriately as indicated in Tables 3 and 4. Our design's logic is such that every bit of the RegSel and ScrSel contributes to activating the desired register. After activation, the data load and the function to be executed on the register are determined by the state of FunSel. The system is designed with two outputs, OutA and OutB, where the data from the selected registers is channeled through multiplexers controlled by OutASel and OutBSel. The multiplexers are crafted using an 'always block' in our Verilog code to ensure dynamic selection of the outputs based on the current needs of the operation. This schematic representation provides a clear overview of the interactions between the different components and the data flow within the system.

You can see the schematic in Figure 3 for a detailed depiction of our Register File design.

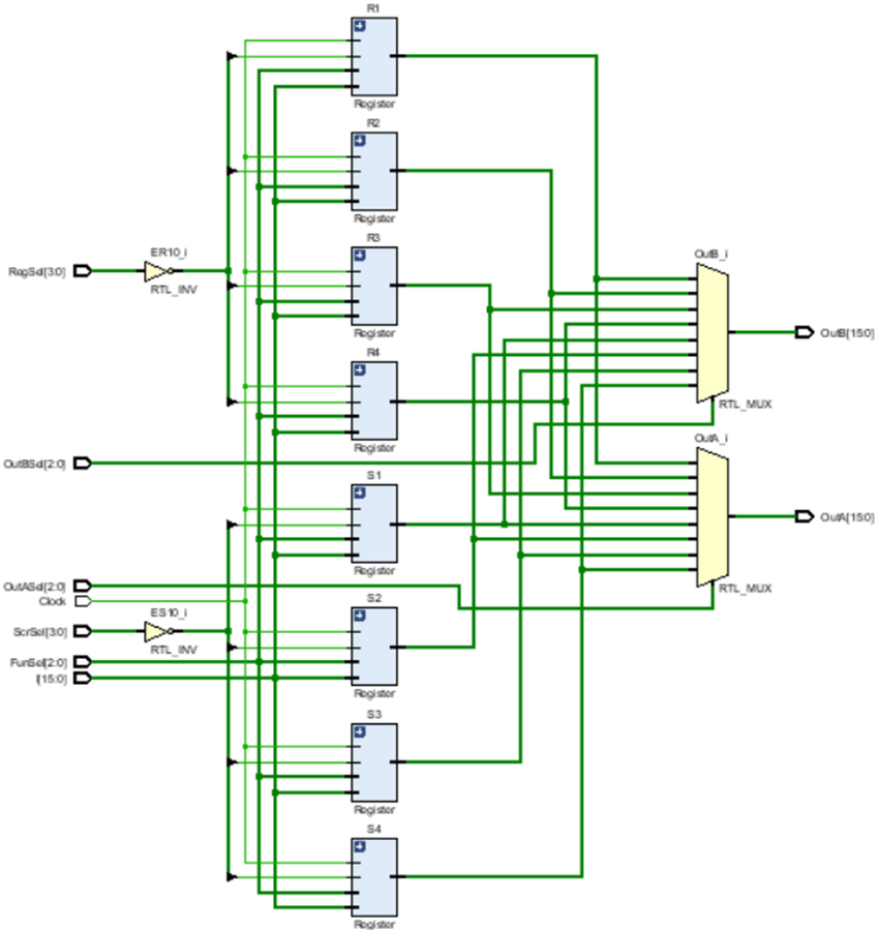

Figure 5: schematic for Register File

Control Inputs	Description
OutASel	Selects which register's value is output to OutA
OutBSel	Selects which register's value is output to OutB
FunSel	Selects the function to perform on the input data before storing
RegSel	Determines which register (R1 to R4) will be enabled for writing
ScrSel	Determines which register (S1 to S4) will be enabled for writing

Figure 6: Control Inputs and Corresponding Functions for Register File

2.4 PART 2-C

For this segment of the project, we've established an Address Register File (ARF), consisting of three critical 16-bit address registers: the Program Counter (PC), Address Register (AR), and Stack Pointer (SP). Much like our previous designs, this configuration is supported by a series of inputs including a load, a single register selector, two outputs selectors, a function selector, and the all-important clock.These components function in harmony to initialize the ARF, with the load, function selector, and clock signal feeding

into each register, similar to the system we devised in Part-2b. The register selector, enabled by the clock, dictates which register will be operational, based on the pattern set out by the RegSel signal.Each address register, upon activation, is capable of being modified by a selected function, thanks to the FunSel control. Owing to the design which incorporates two outputs, we've implemented a pair of multiplexers, each controlled by their respective selectors, OutDSel for OutD, and OutCSel for OutC, allowing us to direct the outputs efficiently.The illustrated design below provides an insight into our ARF implementation, showcasing the schematic interplay of the registers with the input and control signals, and how they collectively contribute to the output channels.

You can see the schematic provided in Figure 4 for a detailed visualization of our Address Register File system.

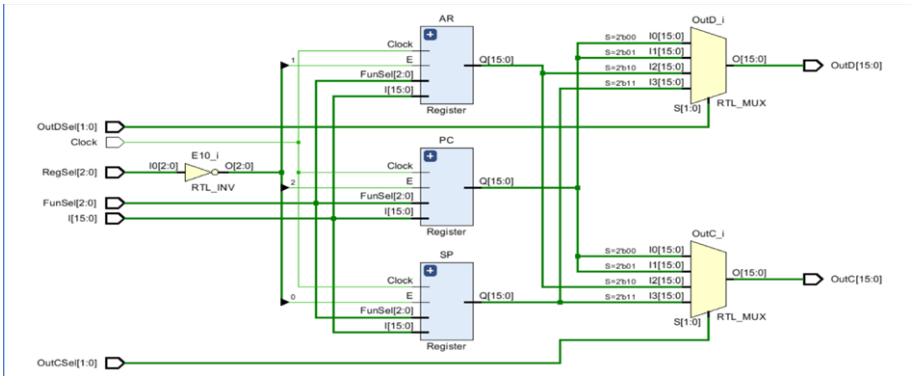

Figure 7: schematic for Address Register File

Control Inputs	Description
OutCSel	2-bit selector to choose the output for OutC
OutDSel	2-bit selector to choose the output for OutD
FunSel	3-bit selector to choose the function operation on the input data
RegSel	3-bit selector to enable writing to the PC, AR, or SP registers

Figure 8: Control Inputs and Corresponding Functions for Address Register File

2.5 PART 3

For this part, we created an ALU, which is a key part of the computer that does different math and logic jobs. We used 2's complement logic for the ALU to work out problems correctly, and we set up flags to keep track of what the ALU is doing. These flags are like signals that let us know if the result of an operation is zero, if there's a leftover number after an addition, or if the result is less than zero. In our programming with Verilog, which is a language for making electronic systems, we used something like a

switch that decides what the ALU should do based on the FunSel input. Then, depending on what kind of operation it is doing, the ALU changes the flags as needed. We put in special checks—if case statements—to handle these changes. The details of how we put all the parts of the ALU together are quite complex, but you can think of it as a big puzzle that fits together just right to do the calculations and decisions that a computer needs. Even though we're not getting into the details of the design, it's carefully made to work well and reliably do the jobs we need it to do.

You can see the schematic provided in Figure 5 for a detailed visualization of ALU:

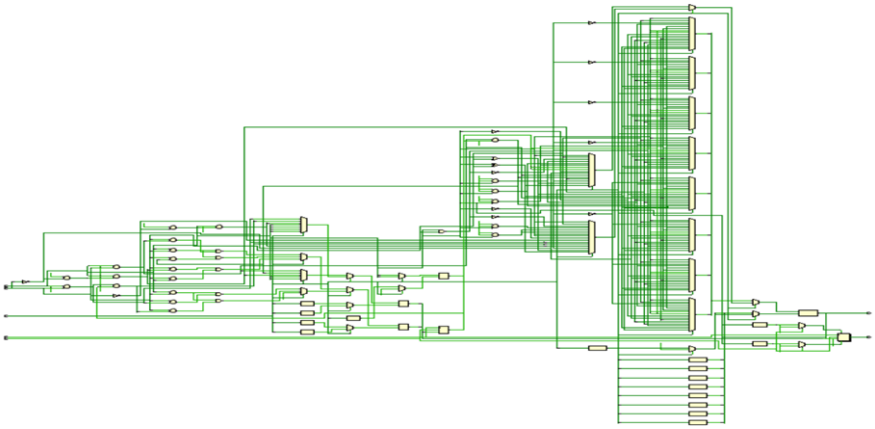

Figure 9: schematic for Arithmetic Logic Unit

FunSel (Binary)	Operation	Description	8-bit/16-bit
0xxxx	Various	Operations defined below are performed on the lower 8 bits of A and B	8-bit
1xxxx	Various	Operations defined below are performed on all 16 bits of A and B	16-bit

Figure 10: Funsel MSB Input and Division of Operations for Arithmetic Logic Unit

FunSel (Binary)	Operation	Description
xxxx0000	Pass A	ALUOut = A
xxxx0001	Pass B	ALUOut = B
xxxx0010	NOT A	ALUOut = ~A
xxxx0011	NOT B	ALUOut = ~B
xxxx0100	A + B	ALUOut = A + B, set carry and overflow flags as needed
xxxx0101	A + B + Carry	ALUOut = A + B + C, set carry and overflow flags as needed
xxxx0110	A - B	ALUOut = A - B, set carry and overflow flags as needed
xxxx0111	A AND B	ALUOut = A AND B
xxxx1000	A OR B	ALUOut = A OR B
xxxx1001	A XOR B	ALUOut = A XOR B
xxxx1010	A NAND B	ALUOut = A NAND B, only lower 8 bits for 8-bit operation
xxxx1011	LSL A	Logical Shift Left of A, setting carry out as needed
xxxx1100	LSR A	Logical Shift Right of A, setting carry out as needed
xxxx1101	ASR A	Arithmetic Shift Right of A
xxxx1110	ROL A	Rotate Left through Carry A
xxxx1111	ROR A	Rotate Right through Carry A

Figure 11: Funsel Inputs and Corresponding Operation for Arithmetic Logic Unit

2.6 PART 4

In this section, we detail the interconnected architecture of the Control Unit System, which orchestrates the operation of the CPU by managing the flow of data and control signals within the system. This schematic diagram illustrates the intricate layout of the Control Unit that includes multiple key components, such as the Instruction Register (IR), which holds the current instruction being executed, and the Control Logic, which interprets the instruction and generates the necessary control signals. These signals are distributed through control lines to various parts of the CPU, such as the ALU, Register File, and Memory, to control operations like data transfer, arithmetic operations, and memory access.

The Control Logic employs decoders, which translate instruction bits into control signals. These signals activate the necessary paths in the multiplexers (MUX) and enable the correct operation to be performed. The multiplexer outputs are linked to input lines of various components, ensuring that data is routed according to the control unit's logic. For example, the MUX near the ALU selects between immediate values from the instruction or data from the Register File for ALU operations. The Condition Code Register updates flags based on the results of ALU operations, which can be used for branching decisions.

Moreover, the Program Counter (PC) is also influenced by the Control Unit, determining the next instruction to fetch based on the current operation. The Feedback Loop

allows the system to adjust the PC in case of branch or jump instructions. The Status Register, updated by the Control Unit, reflects the current state of the CPU, including flags for zero, carry, and overflow, which are crucial for decision-making processes in program execution.

The diagram further indicates the bi-directional data paths, signified by the double-headed arrows, allowing for two-way communication between components, ensuring that data can be read from or written to registers, and memory as dictated by the control signals. The outlined system is a vital part of the CPU, enabling it to execute a sequence of instructions in a program through the systematic control of data and operations within the CPU.

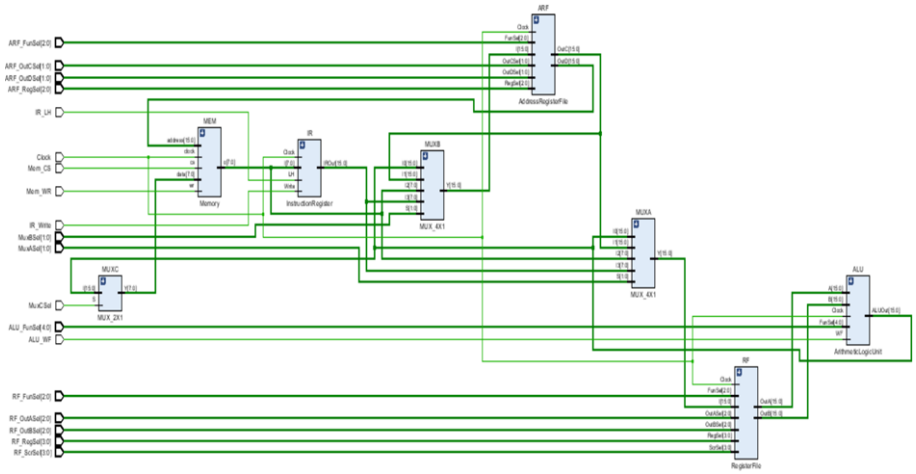

Figure 12: schematic for Arithmetic Logic Unit System

Control Input	Function
RF_OutASel	Selects the output from Register File to ALU input A
RF_OutBSel	Selects the output from Register File to ALU input B
RF_FunSel	Selects the function operation within the Register File
RF_RegSel	Selects which register to write to in the Register File
RF_ScrSel	Selects which scratch register to write to in the Register File
ALU_WF	Write flag to update the FlagsOut in the ALU
ALU_FunSel	Selects the operation to perform in the ALU
ARF_OutCSel	Selects the output from Address Register File to the system
ARF_OutDSel	Selects the output from Address Register File to the memory
ARF_FunSel	Selects the function operation within the Address Register File
ARF_RegSel	Selects which address register to write to in the Address Register File
IR_LH	Selects loading either the low or high byte of the Instruction Register
IR_Write	Enables writing to the Instruction Register
Mem_WR	Write enable signal for the Memory component
Mem_CS	Chip select signal for the Memory component
MuxASel	Selects the input to the MuxA that feeds Register File
MuxBSel	Selects the input to the MuxB that feeds Address Register File
MuxCSel	Selects the high or low byte of the ALUOut to be written to Memory
Clock	System clock that synchronizes the operations

Figure 13: Control Inputs and Corresponding Functions for Arithmetic Logic Unit System

3 RESULTS

During the development of each module, we crafted test files to ensure that each part functioned correctly. This approach simplified the troubleshooting process across the entire system whenever issues arose. Here are the outcomes of our tests, showing that our modules worked as intended.

3.1 PART 1

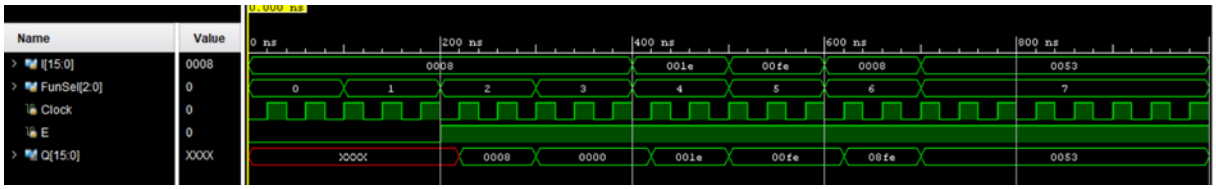

Figure 14: Simulation for 16-bit register

```
-----------------------------------
Register Simulation Started
[PASS] Test No: 1, Component: Q, Actual Value: 0x0029, Expected Value: 0x0029
[PASS] Test No: 2, Component: Q, Actual Value: 0xffff, Expected Value: 0xffff
[PASS] Test No: 3, Component: Q, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 4, Component: Q, Actual Value: 0xf12b, Expected Value: 0xf12b
[PASS] Test No: 5, Component: Q, Actual Value: 0x0012, Expected Value: 0x0012
[PASS] Test No: 6, Component: Q, Actual Value: 0xfa12, Expected Value: 0xfa12
[PASS] Test No: 7, Component: Q, Actual Value: 0x3125, Expected Value: 0x3125
[PASS] Test No: 8, Component: Q, Actual Value: 0xff8c, Expected Value: 0xff8c
Register Simulation Finished
0 Test Failed
8 Test Passed
```

Figure 15: Results of tests for 16-bit register

3.2 PART 2-A

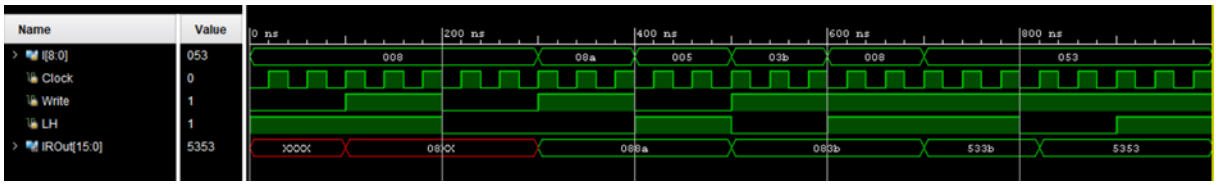

Figure 16: Simulation for 16-bit IR register

```
-----------------------------------
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
0 Test Failed
2 Test Passed
-----------------------------------
```

Figure 17: Results of tests for 16-bit IR register

3.3 PART 2-B

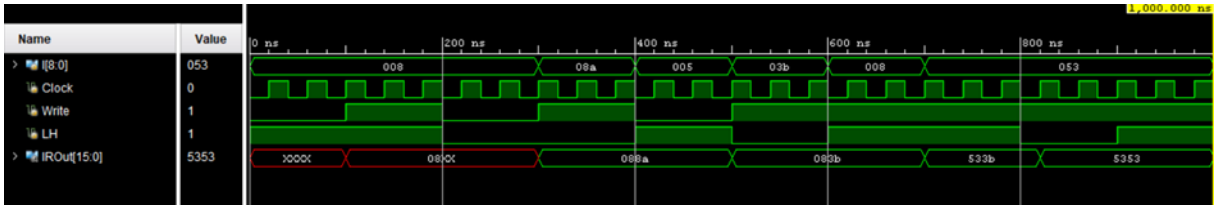

Figure 18: Simulation for 16-bit purpose registers and scratch registers

```
------------------------------------
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
0 Test Failed
4 Test Passed
------------------------------------
```

Figure 19: Results of tests for 16-bit purpose registers and scratch registers

3.4 PART 2-C

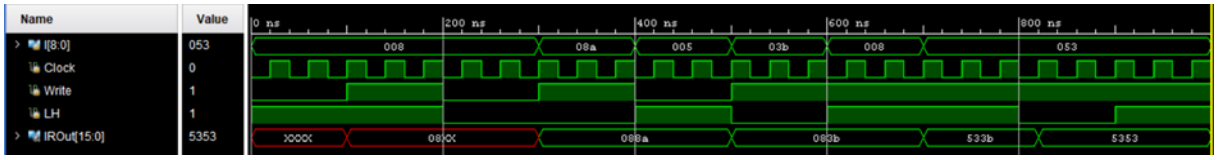

Figure 20: Simulation for Address Registers

```
------------------------------------
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
0 Test Failed
4 Test Passed
------------------------------------
```

Figure 21: Results of tests for Address Registers

3.5 PART 3

```
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
0 Test Failed
15 Test Passed
```

Figure 22: Results of tests for Arithmetic Logic Unit (ALU)

3.6 PART 4

```
-----------------------------------
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
[PASS] Test No: 5, Component: OutC, Actual Value: 0x1254, Expected Value: 0x1254
[PASS] Test No: 5, Component: Address, Actual Value: 0x0023, Expected Value: 0x0023
[PASS] Test No: 5, Component: Memout, Actual Value: 0x0015, Expected Value: 0x0015
[PASS] Test No: 5, Component: IROut, Actual Value: 0x0000, Expected Value: 0x0000
ArithmeticLogicUnitSystem Simulation Finished
0 Test Failed
37 Test Passed
-----------------------------------
exit
```

Figure 23: Results of tests for Arithmetic Logic Unit System(ALUSys)

4 DISCUSSION

4.1 Discussion on the `Register.v` Module

The `Register.v` Verilog module presents a versatile 16-bit register with a range of functionalities determined by a 3-bit function selector (`FunSel`) and an enable signal (`E`). It operates on the rising edge of the provided clock signal (`Clock`), making it suitable for synchronous digital circuits. The register's behavior can be categorized into basic arithmetic operations, data loading, and bit manipulation, depending on the value of `FunSel` when `E` is asserted.

4.1.1 Operational Overview

At its core, the module performs different operations on the 16-bit register (`Q`)—ranging from increment and decrement operations to conditional loading and bit masking. These operations are crucial for a variety of computational and control tasks within digital systems, such as counters, shift registers, and finite state machines.

4.1.2 Functional Selector Analysis

The table below summarizes the operations performed by the module based on the `FunSel` input.

Table 1: Summary of Operations based on `FunSel` Input

`FunSel`	**Operation**	**Description**
000	Decrement	Subtracts 1 from `Q`
001	Increment	Adds 1 to `Q`
010	Load	Loads input `I` into `Q`
011	Clear	Sets `Q` to 0
100	Load Lower	Loads lower 8 bits of `I` into lower 8 bits of `Q`
101	Update Lower	Updates lower 8 bits of `Q` with lower 8 bits of `I`
110	Update Upper	Updates upper 8 bits of `Q` with lower 8 bits of `I`
111	Sign Extend	Sign extends lower 8 bits of `I` into `Q`

4.1.3 Implications and Applications

The `Register.v` module is a fundamental building block in digital design. Its capacity to perform multiple operations makes it extremely versatile. For example, the

increment and decrement operations can be used to design counters or implement algorithms requiring arithmetic operations. The ability to load and manipulate specific bits of the register is particularly useful for operations that require partial data storage or modification, aligning well with tasks in data processing and control systems.

The design also highlights the importance of the enable signal (`E`), which acts as a gatekeeper for the register's operations, ensuring that modifications to the register occur only when desired. This feature is critical for the controlled execution of operations in larger systems, where the timing of data updates must be precisely managed.

4.1.4 Conclusion

The `Register.v` module exemplifies the versatility and efficiency of Verilog in designing digital components. Its diverse set of operations, controlled via the `FunSel` input, enables a wide range of applications in digital systems. Furthermore, the inclusion of an enable signal adds a layer of control, ensuring operations are performed only when appropriate, which is essential for building reliable and predictable digital circuits.

4.2 Discussion on the `InstructionRegister.v` module

The `InstructionRegister` module is a fundamental component of our project, designed to store instruction codes temporarily during execution. This module operates on a simple principle: capturing and updating the instruction register based on the input signals and the clock's rising edge. It is built to handle 8-bit instructions, with a provision for either the lower half (LH=0) or the upper half (LH=1) of a 16-bit instruction register (`IROut`) to be updated. This design choice allows for a flexible instruction handling mechanism, catering to a variety of instruction formats and lengths.

4.2.1 Implementation Details

The `InstructionRegister` leverages an edge-triggered behavior to update its contents, ensuring that instructions are captured precisely at the clock's rising edge. The register supports partial updates – a crucial feature for operations requiring instruction modification or partial loading. The control signals, namely `Write` and `LH`, dictate the operation mode: whether to write to the register and which half to update.

The following table summarizes the module's inputs and outputs:

The module's behavior is contingent on the `Write` signal. When `Write` is asserted, the `InstructionRegister` updates either its lower or upper 8 bits based on the `LH` signal. This selective update feature provides the system with the flexibility needed for complex

Table 2: Input and Output Summary of `InstructionRegister`

Type	Name	Description
Input	`Clock`	Clock signal
Input	`I`	8-bit input data
Input	`Write`	Control signal to write/update
Input	`LH`	Selects lower or higher byte for update
Output	`IROut`	16-bit instruction register

instruction handling and sequencing. If the `Write` signal is not asserted, the register maintains its current state, ensuring stability and data retention across clock cycles.

In summary, the `InstructionRegister` module is pivotal for the instruction execution mechanism within our digital system. Its design and implementation underscore our system's capability to handle instructions dynamically, facilitating efficient execution of operations and contributing to the overall flexibility and robustness of our project architecture.

4.3 Discussion on the `RegisterFile.v` Module

The `RegisterFile` module is an integral component of our digital system, designed to facilitate efficient data storage, retrieval, and manipulation. It embodies a collection of registers that can be selectively read from or written to, based on the control signals provided. This flexibility is pivotal for operations requiring multiple data sources or destinations within the system's computational framework.

4.3.1 Module Overview

At its core, the `RegisterFile` manages eight registers, divided into two sets: R (R1 to R4) and S (S1 to S4). Each register is capable of holding a 16-bit value. Selection inputs (`OutASel` and `OutBSel`) determine which registers' values are outputted, thereby enabling dual simultaneous data retrieval. The `FunSel`, `RegSel`, and `ScrSel` inputs further define the operational context, influencing both the data path and the manipulation processes.

4.3.2 Inputs, Outputs, and Functional Mapping

The operation of the `RegisterFile` module is detailed through its inputs and outputs, which are instrumental in its integration and functionality within the larger system.

Table 3: Inputs of the `RegisterFile` Module

Name	Width	Description
`I`	16-bit	Input data for registers
`OutASel`	3-bit	Selects output for `OutA`
`OutBSel`	3-bit	Selects output for `OutB`
`FunSel`	3-bit	Function select for register operations
`RegSel`	4-bit	Register selection for R1 to R4
`ScrSel`	4-bit	Register selection for S1 to S4
`Clock`	1-bit	System clock signal

Table 4: Outputs of the `RegisterFile` Module

Name	Width	Description
`OutA`	16-bit	Data output A
`OutB`	16-bit	Data output B

4.3.3 Functional Behavior

The `RegisterFile` module's functionality hinges on dynamic data selection and manipulation. The selection for output data (`OutA` and `OutB`) is managed through the `OutASel` and `OutBSel` signals, allowing any of the eight registers to be accessed for reading. Writing to the registers is controlled by the `FunSel`, `RegSel`, and `ScrSel` inputs, which enable specific register targeting and function execution based on the operational requirements.

This module exemplifies the project's commitment to versatile and efficient data handling, underpinning the system's capability to execute complex operations with minimal latency and high accuracy.

4.4 Discussion on the `AddressRegisterFile.v` Module

The `AddressRegisterFile` module plays a pivotal role in our digital system, primarily handling the storage and selection of address-related data. This module encapsulates the functionality required to manage crucial address registers within the system, including the Program Counter (PC), Address Register (AR), and Stack Pointer (SP). It is designed to facilitate dynamic data routing and manipulation, catering to the needs of instruction execution and memory management.

4.4.1 Module Overview

The module is engineered to hold and selectively output data from three key registers: PC, AR, and SP. Each of these registers serves a specific function in the control and data flow of the system. The `AddressRegisterFile` ensures that the system can access and modify these registers based on operational requirements, thus supporting efficient program execution and memory operations.

4.4.2 Inputs, Outputs, and Functional Mapping

A clear understanding of the module's inputs and outputs is essential for appreciating its role within the system. The following tables provide a summary of the module interface:

Table 5: Inputs of the `AddressRegisterFile` Module

Name	Width	Description
`I`	16-bit	Input data for the registers
`OutCSel`	2-bit	Output selection for `OutC`
`OutDSel`	2-bit	Output selection for `OutD`
`FunSel`	3-bit	Function selection for register operations
`RegSel`	3-bit	Register selection control
`Clock`	1-bit	System clock signal

Table 6: Outputs of the `AddressRegisterFile` Module

Name	Width	Description
`OutC`	16-bit	Data output C
`OutD`	16-bit	Data output D

4.4.3 Functional Behavior

The `AddressRegisterFile` distinguishes itself through its ability to selectively output data from its internal registers based on the control signals `OutCSel` and `OutDSel`. This selective output capability is crucial for tasks such as address calculation, program flow control, and stack management. The module's design allows for the dynamic selection of either the PC, AR, or SP register data to be routed to its outputs (`OutC` and `OutD`), facilitating versatile system operations and enabling efficient data handling and processing.

In conclusion, the `AddressRegisterFile` module underscores our system's architectural sophistication, highlighting the emphasis on modularity, flexibility, and efficiency. Its role in managing address-related data is indispensable for the seamless execution of programs and the effective management of memory resources.

4.5 Discussion on the `ArithmeticLogicUnit.v` Module

The `ArithmeticLogicUnit` (ALU) is a critical component in our system, designed to execute arithmetic and logic operations. It handles inputs `A` and `B` to perform operations determined by `FunSel`, and outputs the result along with flags that indicate various states like zero, carry, negative, and overflow.

4.5.1 Module Inputs and Outputs

The interface of the ALU comprises various inputs and outputs that facilitate its functionality:

Table 7: Inputs of the `ArithmeticLogicUnit` Module

Name	Width	Description
`A`	16-bit	Primary input operand
`B`	16-bit	Secondary input operand
`FunSel`	5-bit	Operation selection code
`WF`	1-bit	Write Flag for updating flags
`Clock`	1-bit	System clock signal

Table 8: Outputs of the `ArithmeticLogicUnit` Module

Name	Width	Description
`ALUOut`	16-bit	Result of the operation
`FlagsOut`	4-bit	Status flags (Zero, Carry, Negative, Overflow)

4.5.2 Operational Modes and Function Selection

The ALU supports a variety of operations, controlled by the `FunSel` input. Each operation is defined by a specific 5-bit code:

Table 9: Function Selection Codes for the ALU

FunSel Code	Operation
8-bit operations:	
00000	Pass A
00001	Pass B
00010	NOT A
00011	NOT B
00100	A + B
00101	A + B + Carry
00110	A - B
00111	A AND B
01000	A OR B
01001	A XOR B
01010	A NAND B
01011	LSL A
01100	LSR A
01101	ASR A
01110	CSL A
01111	CSR A
16-bit operations:	
10000	Pass A
10001	Pass B
10010	NOT A
10011	NOT B
10100	A + B
10101	A + B + Carry
10110	A - B
10111	A AND B
11000	A OR B
11001	A XOR B
11010	A NAND B
11011	LSL A
11100	LSR A
11101	ASR A
11110	CSL A
11111	CSR A

4.5.3 Flags and Their Implications

The ALU module updates its flags based on the outcomes of operations. These flags include Zero (Z), Carry (C), Negative (N), and Overflow (O), each indicating a specific condition that results from the last executed operation.

4.5.4 Conclusion

The `ArithmeticLogicUnit` module's versatility and comprehensive functionality highlight its critical role in the computational architecture. Its ability to perform a wide range of operations efficiently supports complex arithmetic and logic processing necessary for the system's operation.

4.6 Discussion on the `ArithmeticLogicUnitSystem.v` Module

The `ArithmeticLogicUnitSystem` module represents a sophisticated assembly within our project, integrating multiple fundamental modules into a single cohesive unit. This system encapsulates the operations of memory access, arithmetic and logical computations, instruction registration, and dynamic data routing through multiplexers, all governed by a unified clock signal.

4.6.1 System Overview

This complex module leverages the capabilities of various components:

- **Memory (MEM)**: Manages data storage and retrieval.
- **Arithmetic Logic Unit (ALU)**: Performs arithmetic and logic operations.
- **Multiplexers (MUX A, MUX B, MUX C)**: Routes data between different system parts.
- **Register Files (RF) and Address Register File (ARF)**: Store and provide access to data and addresses.
- **Instruction Register (IR)**: Temporarily holds instruction data.

Inputs and Outputs

The system's functionality is directed by a series of inputs, each contributing to the module's operational flow:

Table 10: Inputs of the `ArithmeticLogicUnitSystem` Module

Name	Type	Description
`RF_OutASel, RF_OutBSel`	3-bit	Selects outputs for RF module
`RF_FunSel`	3-bit	Function selection for RF operations
`RF_RegSel, RF_ScrSel`	4-bit	Register selection in RF
`ALU_WF`	1-bit	Write flag for ALU
`ALU_FunSel`	5-bit	Function selection for ALU operations
`ARF_OutCSel, ARF_OutDSel`	2-bit	Selects outputs for ARF module
`ARF_FunSel`	3-bit	Function selection for ARF operations
`ARF_RegSel`	3-bit	Register selection in ARF
`IR_LH, IR_Write`	1-bit	Control signals for IR
`Mem_WR, Mem_CS`	1-bit	Write and chip select for Memory
`MuxASel, MuxBSel`	2-bit	Selection inputs for MUX A and B
`MuxCSel`	1-bit	Selection input for MUX C
`Clock`	1-bit	System clock

4.6.2 Operational Behavior

The system's operation is driven by the clock signal, synchronizing data flow through its components. Memory interactions, ALU computations, and data routing via multiplexers are all conducted in harmony, adhering to the signals provided by the system's inputs. This orchestration ensures that data is processed efficiently, instructions are executed accurately, and the correct data is stored or forwarded as needed.

5 Conclusion

Throughout this report, we have delved into the intricacies of our Verilog project, examining the design, functionality, and interconnectivity of several key modules: the Memory, Arithmetic Logic Unit (ALU), multiplexers (4X1 MUX and 2X1 MUX), register files, and the encompassing 'ArithmeticLogicUnitSystem'. Each module, with its unique role and complexity, contributes significantly to the overarchin g goal of creating a versatile and efficient digital system.

The Memory module serves as the foundation for data storage and retrieval, facilitating dynamic data management. The ALU is at the heart of arithmetic and logical operations, providing the necessary computational power. Multiplexers introduce flexibil-

ity in data routing, enabling conditional data flow based on control signals. Register files, including both general and address-specific types, offer temporary data storage for efficient access and manipulation. The integration of these components into the ‘ArithmeticLogicUnitSystem‘ exemplifies a harmonious orchestration of data processing, showcasing the project's modular design and scalability.

This project not only highlights the practical application of digital design principles but also demonstrates the power of modular architecture in creating complex systems. The ability to break down a system into manageable, reusable components like those discussed in this report is invaluable in the field of digital systems design. It allows for easier debugging, testing, and future enhancements, paving the way for more advanced and robust systems.

In conclusion, the development and discussion of these modules have not only solidified our understanding of fundamental Verilog constructs and digital design concepts but have also prepared us for tackling more complex projects. As we move forward, the lessons learned and the methodologies adopted in this project will undoubtedly serve as a strong foundation for future endeavors in the realm of digital design and computer engineering.