ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E
COMPUTER ORGANIZATION
PROJECT 1 REPORT

CRN : 21334

LECTURER : Doç. Dr. Gökhan İnce

GROUP MEMBERS:

150230739 : MUHAMMED ALİ AÇIKGÖZ

150220032 : YAVUZ SELİM KARA

SPRING 2024

Contents

1 INTRODUCTION [10 points] **1**

1.1 TASK DISTRIBUTION 1

2 MATERIALS AND METHODS [40 points] **2**

2.1 REGISTER DESIGN 2

2.2 INSTRUCTION REGISTER DESIGN 3

2.3 REGISTER FILE DESIGN 3

2.4 ADDRESS REGISTER FILE DESIGN 5

2.5 ARITHMETIC LOGIC UNIT DESIGN 6

2.6 ARITHMETIC LOGIC UNIT SYSTEM DESIGN 6

3 RESULTS [15 points] **7**

4 DISCUSSION [25 points] **10**

5 CONCLUSION [10 points] **11**

REFERENCES **11**

1 INTRODUCTION [10 points]

The field of computer organization encompasses the study of how computer systems are structured and operate at the hardware level. One fundamental aspect of computer organization is the design and implementation of various components such as registers, register files, arithmetic logic units (ALUs), and memory systems. This project focuses on the design and simulation of these components using Verilog HDL, a hardware description language commonly used for digital circuit design. The project is divided into five main parts:

1. Designing a 16-bit register with multiple functionalities controlled by control signals.
2. Designing a register file containing multiple registers with different functionalities.
3. Designing an address register file (ARF) system comprising address registers such as the program counter (PC), address register (AR), and stack pointer (SP).
4. Implementing an arithmetic logic unit (ALU) that performs various arithmetic and logic operations on input data.
5. Combining all the components designed in the previous parts in an arithmetic logic unit system (ALUSystem).

Each part involves designing and simulating specific hardware modules according to provided specifications. The goal is to create efficient and reusable components that can be integrated into larger computer systems.

1.1 TASK DISTRIBUTION

Throughout the duration of our project, we collaborated effectively, utilizing Discord as our primary communication platform. Our meetings were structured around finding suitable times for both of us, accommodating our schedules to ensure productive work sessions. Over the course of the project, we held approximately 5 to 6 meetings, each lasting between 3 to 4 hours. During these sessions, we handled the problems together and helped each other to understand the missing parts.

Each member had contributed for all parts, including designing the 16-bit register module, register file module, address register file (ARF) system, arithmetic logic unit (ALU) module, and integration of all modules into the final system. Furthermore, responsibilities encompassed simulation testing and ensuring compatibility with the overall system architecture. By collectively contributing to the design, simulation, and testing phases, we ensured the successful completion of all tasks, achieving our project objectives effectively.

2 MATERIALS AND METHODS [40 points]

To accomplish the objectives outlined in the project instructions, we utilized Verilog HDL to design and simulate each hardware module. The design process involved the following steps:

1. Understanding the project requirements and specifications outlined in the provided instructions.

2. Decomposing each part of the project into smaller modules to facilitate modular design and reusability.

3. Implementing the Verilog code for each module, adhering to the defined functionality and control signals.

4. Simulating each module using the provided simulation files to verify correct functionality under various input conditions.

5. Debugging and refining the design based on simulation results to ensure proper operation.

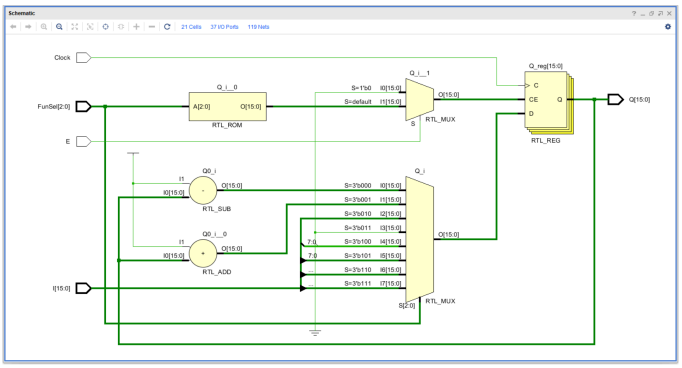
6. Integrating the individual modules to create the complete system according to the system architecture provided in the project instructions.

7. Verifying the functionality of the integrated system through comprehensive simulation testing.

8. Throughout the design process, we paid close attention to efficient resource utilization, modularity, and adherence to the given specifications. Additionally, we documented our design decisions and implementation details to provide clarity and facilitate understanding.

2.1 REGISTER DESIGN

Firstly, a 16-bit register is developed, featuring 8 functionalities regulated by 3-bit control signals (FunSel) and an enable input (E). The graphical representation of the registers and their characteristic table is depicted in Figure 1. The symbol Φ denotes "don't care." This register is subsequently utilized in subsequent sections of the project.



(a) Register RTL Design

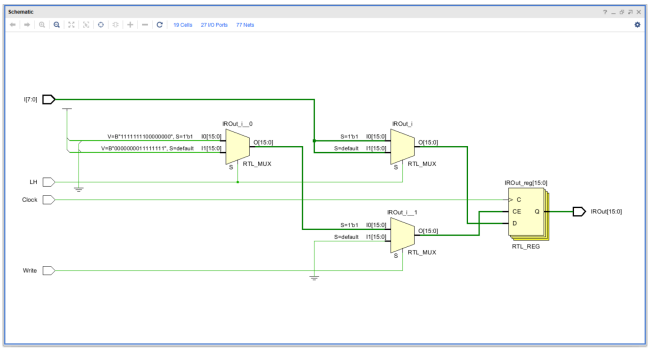
E	FunSel	Q⁺
0	ϕ	Q (Retain value)
1	000	Q-1 (Decrement)
1	001	Q+1 (Increment)
1	010	I (Load)
1	011	0 (Clear)
1	100	Q (15-8) ← Clear, Q (7-0) ← I (7-0) (Write Low)
1	101	Q (7-0) ← I (7-0) (Only Write Low)
1	110	Q (15-8) ← I (7-0) (Only Write High)
1	111	Q (15-8) ← Sign Extend (I (7)) Q (7-0) ← I (7-0)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

(b) Control Inputs of the Register[1]

Figure 1: Register and its Control Inputs

2.2 INSTRUCTION REGISTER DESIGN

Secondly, a 16-bit Instruction Register (IR) register has been designed, as illustrated by its block diagram and function table in Figure 2. This register is capable of storing 16-bit binary data. However, its input is restricted to only 8 bits. Consequently, utilizing the 8-bit input bus, it becomes possible to load either the lower half (bits 7-0) or the upper half (bits 15-8) of the register. This determination is governed by the L'H signal.



(a) Instruction Register RTL Design

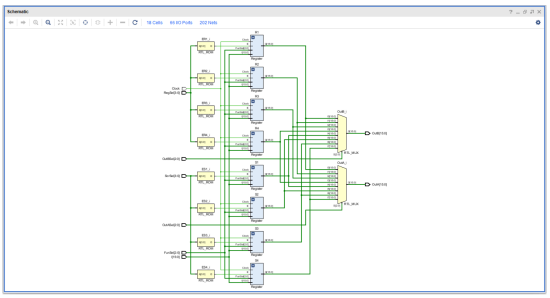
L'H	Write	IR⁺
ϕ	0	IR (retain value)
0	1	IR (7-0) ← I (Load LSB)
1	1	IR (15-8) ← I (Load MSB)

(b) Control Inputs of the Instruction Register[1]

Figure 2: Instruction Register and its Control Inputs

2.3 REGISTER FILE DESIGN

Followingly, a register file (RegisterFile) has been designed, as depicted by its block diagram and functional table in Figure 3, encompassing four 16-bit general-purpose registers: R1, R2, R3, R4, and four 16-bit scratch registers: S1, S2, S3, and S4. The specifics of inputs and outputs are outlined as follows: OutASel and OutBSel are utilized to supply the output lines OutA and OutB, respectively. Sixteen bits from the selected registers are channeled to OutA and OutB. Table 1 delineates the selection of output registers



(a) Register File RTL Design

Table 1: OutASel and OutBSel controls.

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

(b) Control Inputs of the Register File - OutASel and OutCSel[1]

Table 2: FunSel Control Input.

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

(c) Control Inputs of the Register File - FunSel[1]

Table 3: RegSel Control Input

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.)

(d) Control Inputs of the Register File - RegSel[1]

Table 4: ScrSel Control Input

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.)

(e) Control Inputs of the Register File - ScrSel[1]

Figure 3: Register File and its Control Inputs

predicated on the OutASel and OutBSel control inputs. RegSel and ScrSel constitute 4-bit signals that designate the registers to apply the function determined by the 3-bit FunSel signal, as delineated in Table 2. The registers selected by RegSel and ScrSel are elucidated in Tables 3 and 4, respectively.

2.4 ADDRESS REGISTER FILE DESIGN

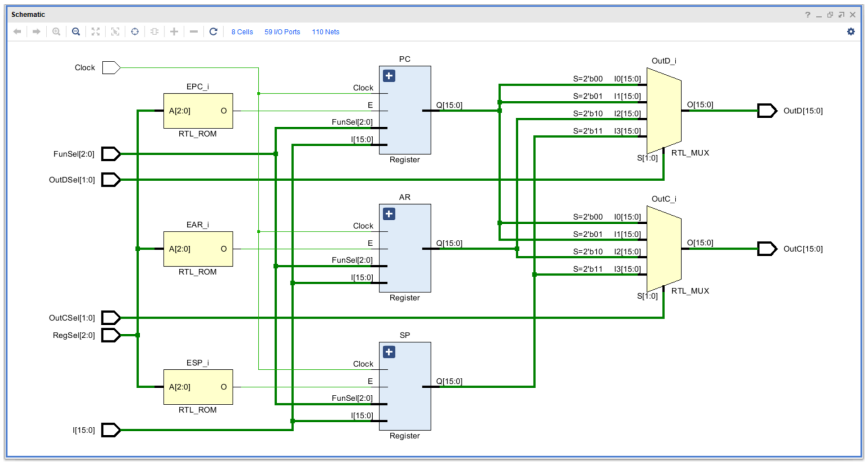


Figure 4: Address Register File RTL Design

The system described consists of three 16-bit address registers: the program counter (PC), address register (AR), and stack pointer (SP). The system is controlled by two inputs: FunSel and RegSel.

- FunSel determines the function applied to the selected address registers.
- RegSel enables specific combinations of address registers for the selected function.

Here's how RegSel operates:

When RegSel is 000, all address registers (PC, AR, and SP) are enabled, allowing the selected function from FunSel to be applied to all of them. Depending on the value of RegSel, different combinations of address registers are enabled:

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 5: Control Inputs of the Address Register File - Enable Address Registers[1]

Additionally, the system has two output lines, OutC and OutD, controlled by OutCSel and OutDSel, respectively. These controls determine which registers' contents are output:

OutCSel selects the register whose contents will be output on OutC. OutDSel selects the register whose contents will be output on OutD. The mappings between the values

of OutCSel and OutDSel and the corresponding output registers (PC, AR) are shown in Table 5.

Table 5: OutCSel and OutDSel controls.

OutCSel	OutC
00	PC
01	PC
10	AR
11	SP

OutDSel	OutD
00	PC
01	PC
10	AR
11	SP

Figure 6: Control Inputs of the Address Register File - OutCSel and OutDSel controls[1]

2.5 ARITHMETIC LOGIC UNIT DESIGN

Subsequently, an arithmetic logic unit (ALU) has been designed, featuring two 16-bit inputs, a 16-bit output, and a 4-bit output designated for zero, negative, carry, and overflow flags.

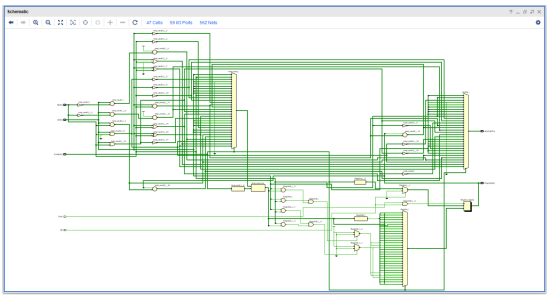
The ALU, depicted in Figure 7, is capable of executing various functions, with the corresponding flags updated as delineated in Table 6 when WF (Write Flag) is set to 1. The FunSel parameter selects the operation to be performed by the ALU. ALUOut denotes the outcome of the operation, which is selected by FunSel and applied to the A and/or B inputs. Arithmetic computations are conducted utilizing 2's complement logic. The Z (zero) bit is asserted if ALUOut equals zero, while the C (carry) bit is asserted if ALUOut sets the carry flag. The N (negative) bit is set if the ALU operation yields a negative result, and the O (overflow) bit is set in case of overflow. The Z, C, N, O flags are stored in a register, with the Z flag occupying the most significant bit (MSB) and the O flag occupying the least significant bit (LSB) of the register.

We have handled 8 bit operations in 2 parts. Firstly, we have used sign extension for arithmetic operations like (A + B), A, etc. Secondly, we have used the extension with adding 0's for the logical operations like LSL, CSL, AND, etc.

In addition, we have added some test cases for shift operations to see the validty of our code.

2.6 ARITHMETIC LOGIC UNIT SYSTEM DESIGN

Finally, an arithmetic logic unit system (ALUSystem) has been designed that implements the organization in Figure 8. The whole system uses the same single clock.



(a) Arithmetic Logic Unit RTL Design

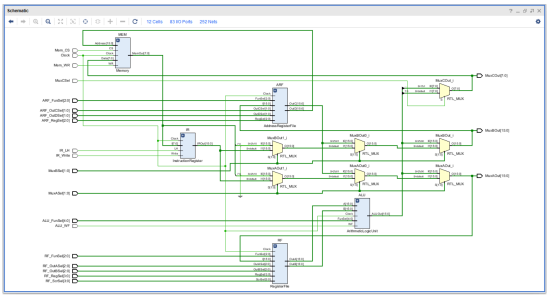
Table 6: Characteristic table of ALU.

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)	+	+	+	-

(b) Control Inputs of the Arithmetic Logic Unit[1]

Figure 7: Arithmetic Logic Unit and its Control Inputs



(a) Arithmetic Logic Unit System RTL Design

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)

(b) Control Inputs of the Arithmetic Logic Unit System - MuxASel, MuxBSel, and MuxCSel[1]

Figure 8: Arithmetic Logic Unit and its Control Inputs

3 RESULTS [15 points]

The results of our project include the successful design and simulation of each hardware module as well as the integrated system. Each module demonstrated correct functionality and responded appropriately to control signals and input data combinations. Simulation results confirmed the proper operation of the designed components under various test scenarios.

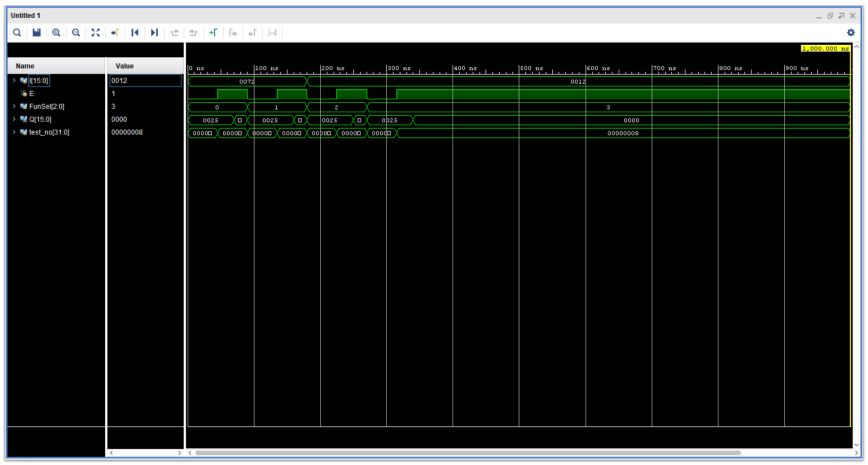


Figure 9: Register Simulation Results in Vivado[1]

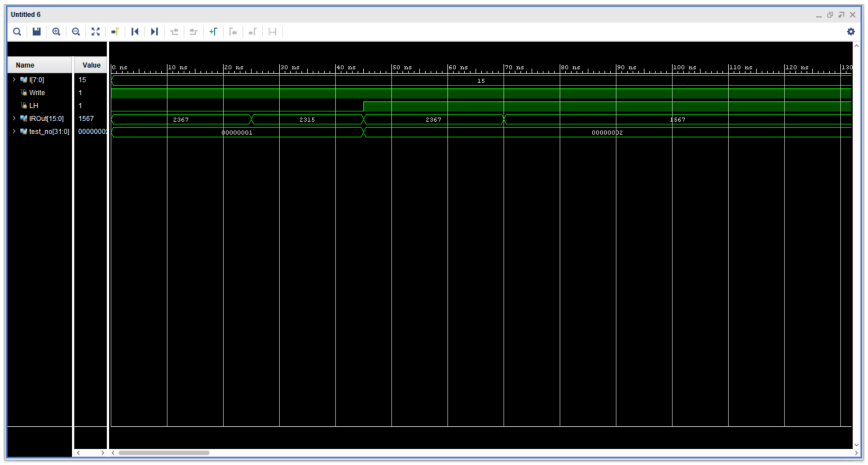


Figure 10: Instruction Register Simulation Results in Vivado[1]

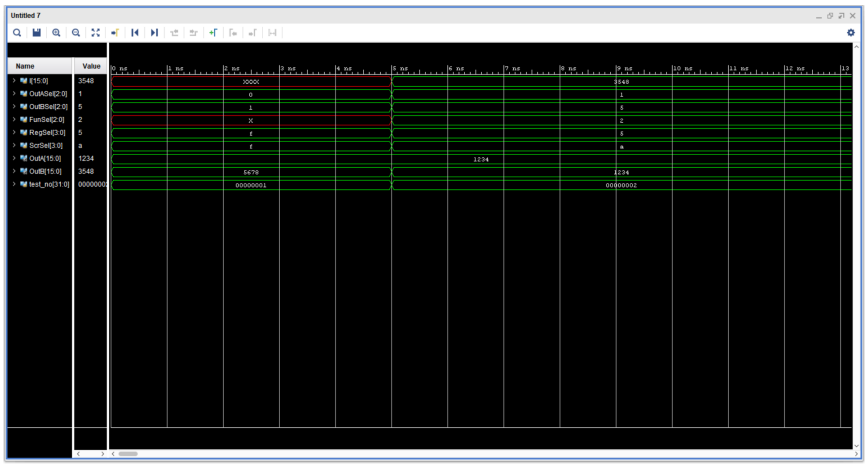


Figure 11: Register File Simulation Results in Vivado[1]

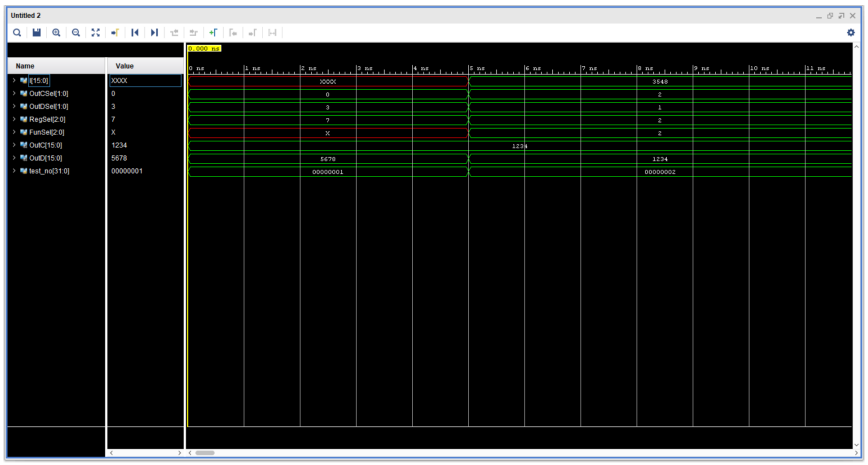


Figure 12: Address Register File Simulation Results in Vivado[1]

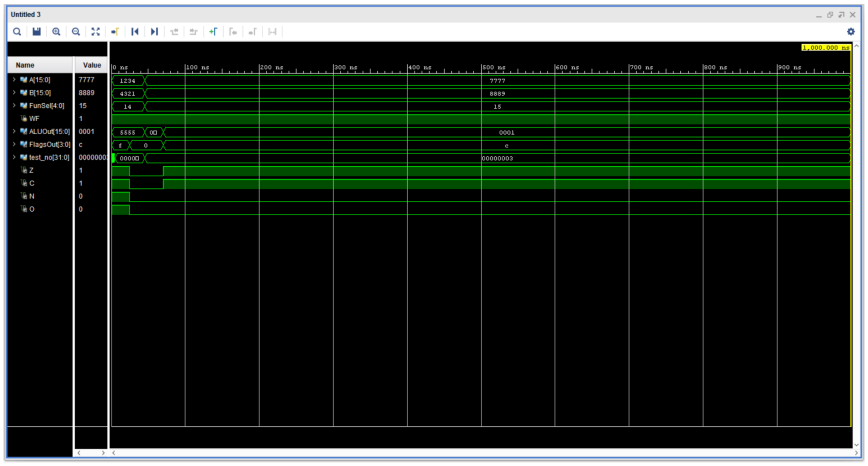


Figure 13: Arithmetic Logic Unit Simulation Results in Vivado[1]

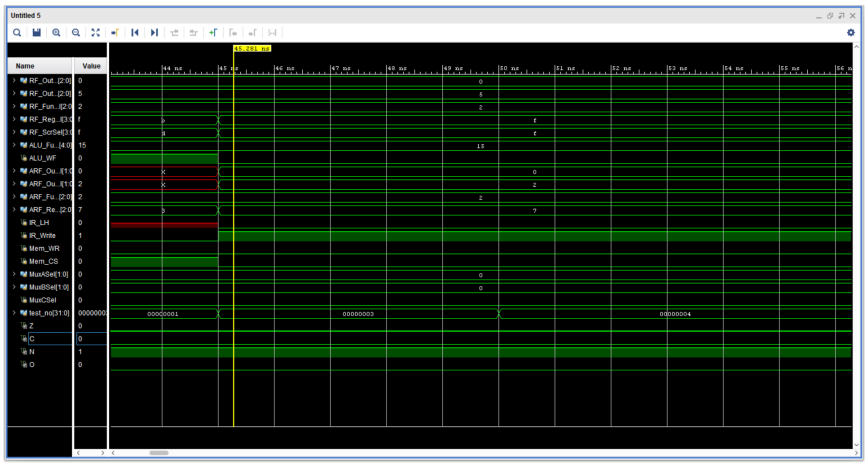


Figure 14: Arithmetic Logic Unit System Simulation Results in Vivado [1]

We have passed all the test cases from the simulation files. We also executed succesfully the given Run.bat file.

4 DISCUSSION [25 points]

The design and simulation of hardware modules for computer organization tasks require careful consideration of functionality, control signals, and data paths. By following the specifications provided in the project instructions, we were able to develop robust and efficient components that meet the requirements of each part.

One key aspect of our design approach was modularity, which allows for easy integration of individual modules into larger systems and promotes reusability. Modular design also facilitates debugging and testing by isolating functional units and simplifying the identification of issues.

During the implementation phase, a significant portion of our effort was dedicated to handling 8-bit inputs for the Arithmetic Logic Unit (ALU). Given that some aspects of the provided project instructions were not fully elucidated, we invested considerable time in research to devise the most suitable solution. One crucial consideration was whether to utilize sign extension to convert 8-bit inputs into 16-bit values, ensuring compatibility with the ALU's operand size. In computer science, when performing logical operations, data are typically treated as sequences of bits and extended with zeros. Conversely, in arithmetic operations, numerical values are considered, and extension is performed by replicating the sign bit to maintain the signedness of the value. Since this information we have decided to go with the solution that using sign extension for artihmetic operations like A+B, and using expanding with 0's for logical operations like A, CSR, LSR, etc.

Sign extension, was indeed incorporated into our design process. By sign-extending the 8-bit inputs to 16 bits, we ensured uniformity in data representation, thereby enabling seamless operation within the ALU. This decision not only aligns with industry standards but also enhances the versatility of our hardware implementation.

Furthermore, the handling of shift operations for 8-bit inputs posed an additional challenge. We deliberated extensively on the optimal placement of shift operations, particularly in relation to extension. ASR (Arithmetic Shift Right) operation should be extended with the sign bit, because of it is an arithmetic operation. CSR, CSL, LSL, and LSR operations extended with 0's due to their operation type is logical. These calculations yielded the desired results.

5 CONCLUSION [10 points]

In conclusion, our project demonstrates the successful design and simulation of hardware modules for computer organization tasks using Verilog HDL in Vivado program. By systematically following the provided instructions and applying sound design principles, we were able to create efficient and reliable components for registers, register files, address registers, and arithmetic logic units.

Through simulation testing, we validated the functionality of our designs and ensured that they meet the specified requirements. The modular nature of our design allows for easy integration into larger computer systems, promoting scalability and flexibility.

Overall, this project provided valuable hands-on experience in digital circuit design and computer organization principles, enhancing our understanding of hardware implementation in computer systems.

REFERENCES

[1] Blg 222 e project 1.pdf. 2024.