ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E
COMPUTER ORGANIZATION
PROJECT 1 REPORT

CRN : 22599

LECTURER : Prof. Dr. Mustafa Ersel Kamaşak

GROUP MEMBERS:

150230083 : Mehmet Fuat Karakaya

150220049 : Furkan Kalay

SPRING 2024

Contents

1 INTRODUCTION [10 points] **1**

2 MATERIALS AND METHODS [40 points] **1**

2.1 Materials 1

2.2 Methods 2

2.2.1 Project Part 1 2

2.2.2 Project Part 2 4

2.2.3 Project Part 3 9

2.2.4 Project Part 4 12

3 RESULTS [15 points] **14**

4 DISCUSSION [25 points] **16**

5 CONCLUSION [10 points] **17**

1 INTRODUCTION [10 points]

In this project, we designed various components of a simple computer system including registers, register files, an arithmetic logic unit (ALU), and the overall ALU system using Verilog HDL. Each component was simulated individually to ensure correct functionality. The task distribution is as follows:

- Part 1: Mehmet Fuat Karakaya
- Part 2: Furkan Kalay
- Part 3: Furkan Kalay
- Part 4: Mehmet Fuat Karakaya
- Report: Furkan Kalay - Mehmet Fuat Karakaya

2 MATERIALS AND METHODS [40 points]

2.1 Materials

- Verilog HDL
- 16-bit Register
- 32-bit Register
- Instruction Register
- Data Register
- Register File
- Address Register File
- Arithmetic Logic Unit
- Multiplexer (MUX)
- Memory Unit

2.2 Methods

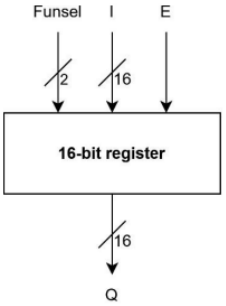
2.2.1 Project Part 1

In the first part of the project, we designed two fundamental hardware modules: a 16-bit register and a 32-bit register. These modules function as synchronous storage elements that update their contents on the rising edge of the clock signal. The update behavior is determined by the control signals `FunSel` and `E`.

16-bit Register The 16-bit register is controlled by a 2-bit `FunSel` input and an enable signal `E`. Based on these inputs, the register can increment, decrement, load a new value, or clear its contents. When `E` is low, the register retains its previous value. All state changes occur on the rising edge of the clock. The schematic of the module is shown in Figure 2a, and its function table is provided in Figure 1a.

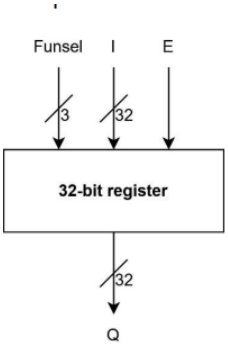
32-bit Register The 32-bit register extends the functionality with a 3-bit `FunSel` control input, allowing additional operations such as bit shifts, zero-extension, and sign-extension. Like the 16-bit register, it updates its value only on the rising edge of the clock and when `E` is asserted. The schematic of this module is shown in Figure 2b, and the function table is given in Figure 1b.

Both modules were tested using the provided simulation files to ensure correct functionality across all combinations of control inputs.



E	FunSel	Q^+
0	ϕ	Q (Retain value)
1	00	Q-1 (Decrement)
1	01	Q+1 (Increment)
1	10	I (Load)
1	11	0 (Clear)

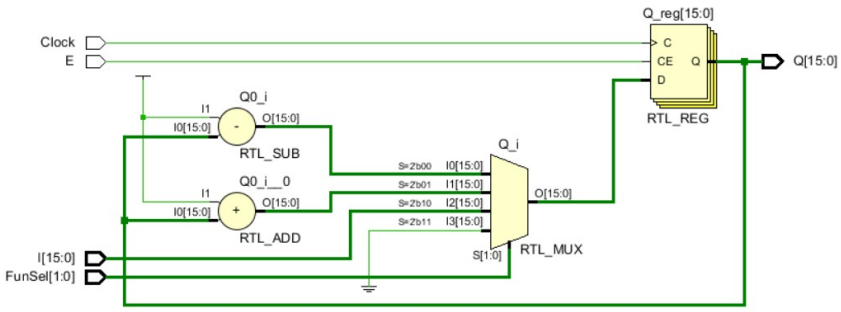
(a) 16-bit register symbol.



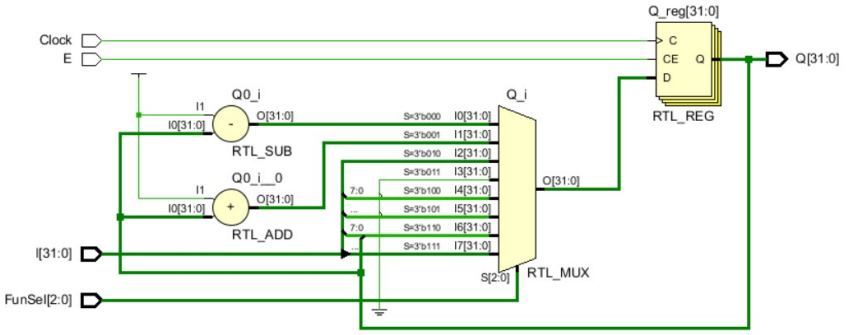
E	FunSel	Q^+
0	ϕ	Q (Retain value)
1	000	Q-1 (Decrement)
1	001	Q+1 (Increment)
1	010	I (Load)
1	011	0 (Clear)
1	100	Q (31-8) ← Clear, Q (7-0) ← I (7-0)
1	101	Q (31-16) ← Clear, Q (15-0) ← I (15-0)
1	110	Q (31-8) ← Q (23-0) (8-bit Left Shift) Q (7-0) ← I (7-0)
1	111	Q (31-16) ← Sign Extend (I (15)) (16-bit sign extend) Q (15-0) ← I (15-0)

(b) 32-bit register symbol.

Figure 1: Graphic symbols for 16-bit and 32-bit registers.



(a) 16-bit Register Schematic.



(b) 32-bit Register Schematic.

Figure 2: Schematics for 16-bit and 32-bit registers.

2.2.2 Project Part 2

In Part 2 of the project, we implemented four key modules, each corresponding to a specific subtask: the *Instruction Register (IR)*, *Data Register (DR)*, *Register File (RF)*, and *Address Register File (ARF)*. Each module was designed according to its description in the project specification and was verified using simulation.

Instruction Register (IR) – (Part-2a) The Instruction Register is a 16-bit module that loads 8-bit input data in two steps, using the `LH` control signal. When `LH` is low, the lower byte of the register is written; when high, the upper byte is written. A write occurs only when the `Write` signal is asserted on the rising edge of the clock. This module was implemented as described in Part-2a. The schematic and function table are shown in Figures 3 and 4, respectively.

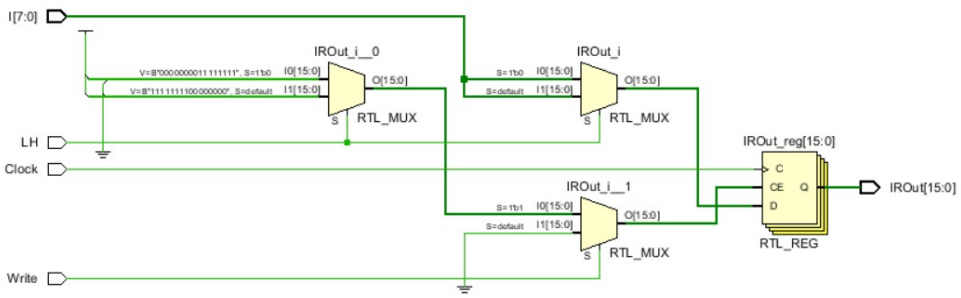
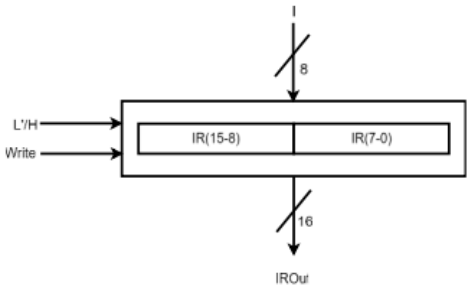


Figure 3: Instruction Register (IR) schematic



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 4: Graphic symbol of IR register and its function table

Data Register (DR) – (Part-2b) The Data Register is a 32-bit module that handles 8-bit input and supports four different operations depending on the `FunSel` control signal. It includes options for sign extension, zero extension, left shift, and right shift. The register updates on the rising edge of the clock when `E` is high. This module corresponds to Part-2b. The schematic and symbol are shown in Figures 5 and 6.

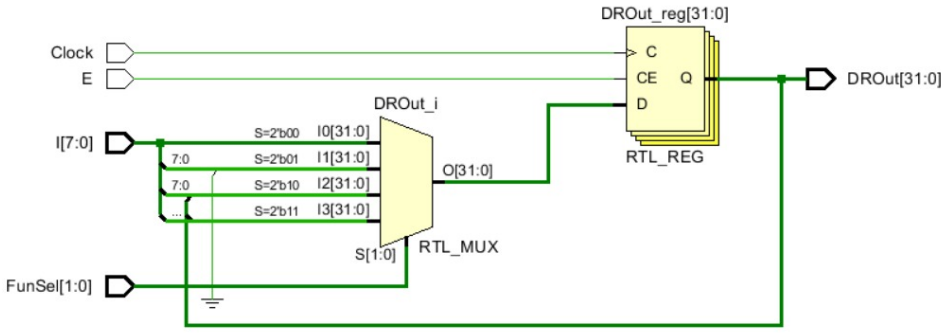
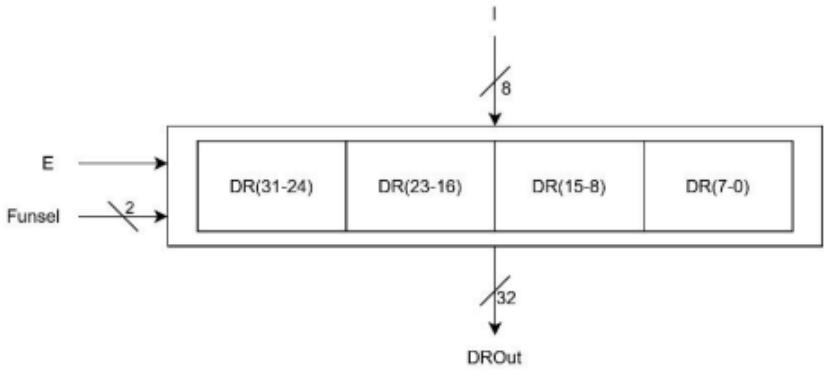


Figure 5: Data Register (DR) schematic



E	FunSel	DR⁺
0	ϕ	DR (Retain value)
1	00	DR (31-8) ← Sign Extend (I (7)) (8-bit sign extend) DR (7-0) ← I (7-0)
1	01	DR (31-8) ← Clear, DR (7-0) ← I (7-0)
1	10	DR (31-8) ← DR (23-0) (8-bit Left Shift) DR (7-0) ← I (7-0)
1	11	DR (23-0) ← DR (31-8) (8-bit Right Shift) DR (31-24) ← I (7-0)

Figure 6: Graphic symbol of DR register and its function table

Register File (RF) – (Part-2c) This module implements a 32-bit register file consisting of four general-purpose registers (R1–R4) and four scratch registers (S1–S4). Each register is an instance of the previously implemented `Register32bit` module. Therefore, the `FunSel` signal in the Register File directly maps to the same control logic described earlier: it enables operations such as increment, decrement, load, clear, bit shifts, zero-extension, and sign-extension, depending on the value of the 3-bit input.

Write operations to registers are selectively controlled using two 4-bit inputs: `RegSel` (for R1–R4) and `ScrSel` (for S1–S4). These control signals determine which registers are

enabled to receive the operation specified by `FunSel`. Simultaneously, two registers can be read and routed to output ports `OutA` and `OutB`, which are selected via the 3-bit inputs `OutASel` and `OutBSel`, respectively.

The schematic and symbolic view of the Register File are shown in Figures 7 and 8. The detailed behavior of the control signals is presented in the following figures.

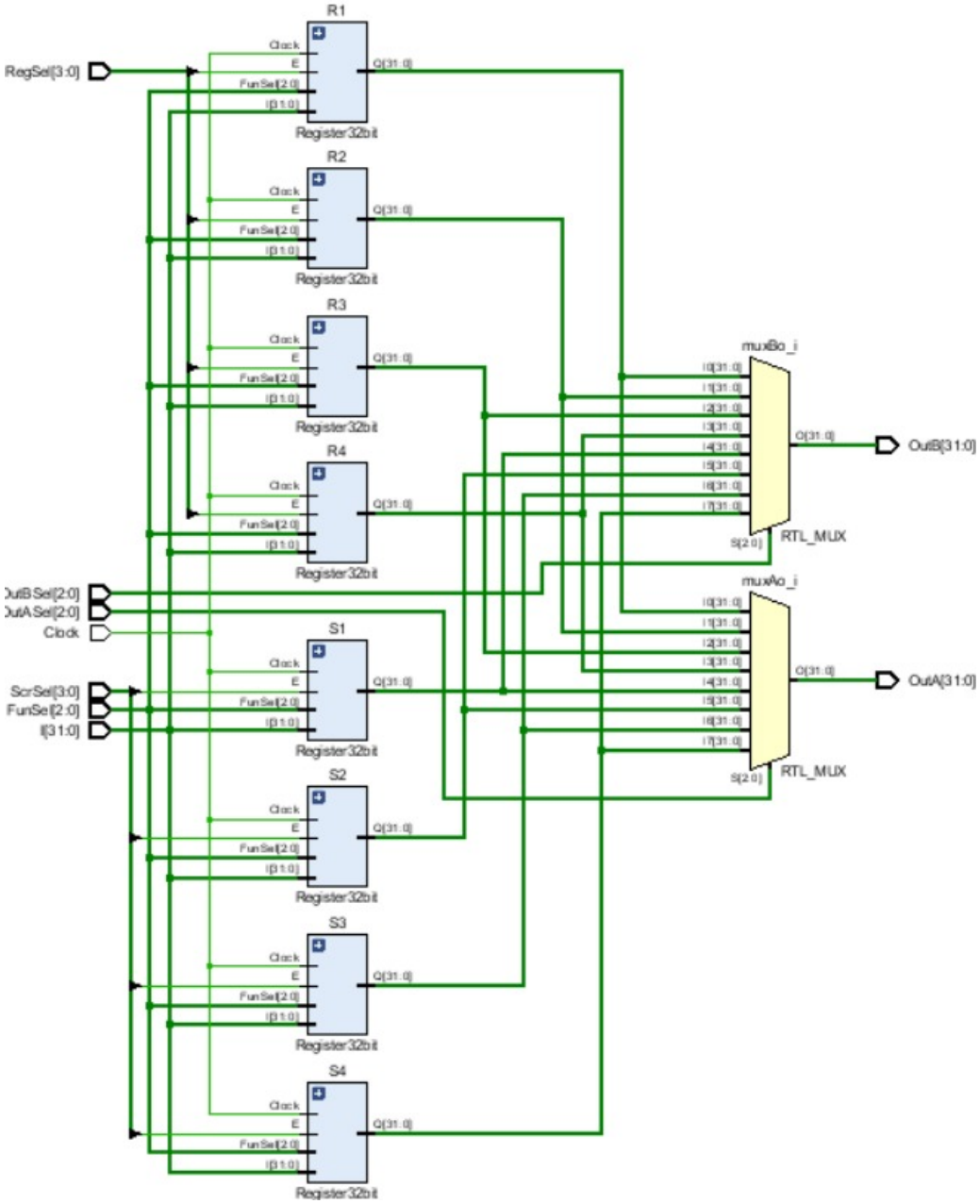


Figure 7: 32-bit Register File (RF) schematic

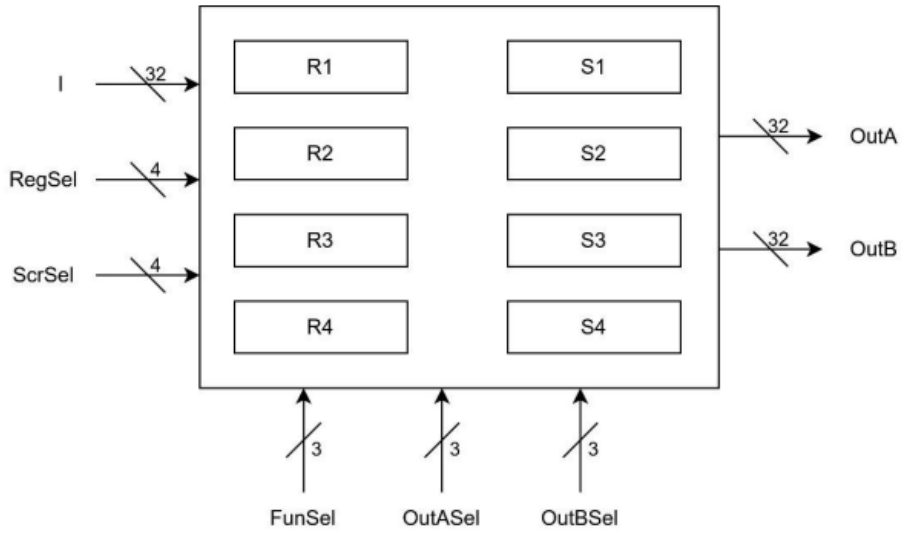


Figure 8: Graphic symbol of Register File and control interface

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

Figure 9: RegSel control input table: general-purpose register enable combinations

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

Figure 10: ScrSel control input table: scratch register enable combinations

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 11: Output selection for OutA and OutB based on OutASel and OutBSel

Address Register File (ARF) – (Part-2d) The Address Register File (ARF) consists of three 16-bit registers: the Program Counter (PC), Stack Pointer (SP), and Address Register (AR). These registers are instantiated using the previously implemented `Register16bit` module. Therefore, the functionality of the `FunSel` control input is identical to that of the 16-bit register: it supports increment, decrement, load, and clear operations. All updates occur on the rising edge of the clock and are gated by the `RegSel` input.

The ARF allows parallel read operations on two output ports: `OutC` and `OutD`. The data routed to these outputs is determined by the 2-bit signals `OutCSel` and `OutDSel`, which select among the three internal registers.

Figures 12 and 13 show the schematic and symbolic representation of the ARF module. The detailed operation of the control signals `OutCSel`, `OutDSel`, and `RegSel` are presented in the figures below.

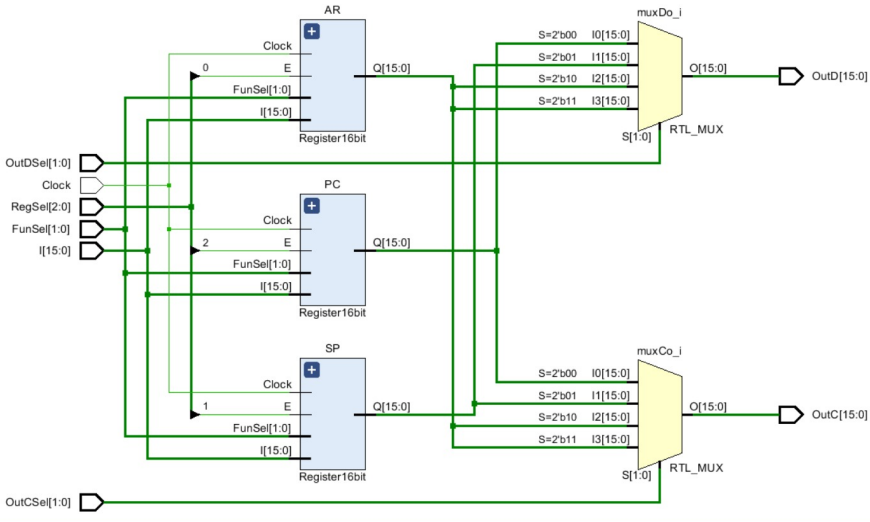


Figure 12: Address Register File (ARF) schematic

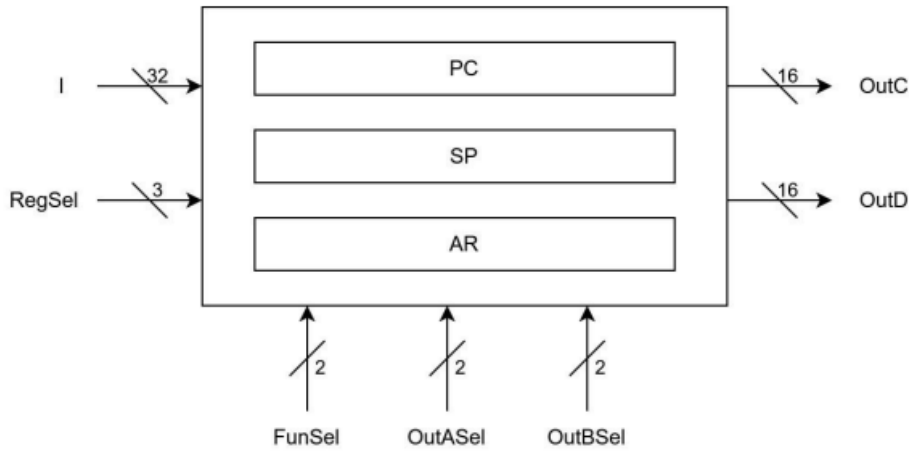


Figure 13: Graphic symbol of Address Register File

OutCSel	OutC
00	PC
01	SP
10	AR
11	AR

OutDSel	OutD
00	PC
01	SP
10	AR
11	AR

Figure 14: Output selection for OutC and OutD based on OutCSel and OutDSel

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

Figure 15: RegSel control input table: register enable combinations for PC, SP, and AR

2.2.3 Project Part 3

The Arithmetic Logic Unit (ALU) module was designed to perform both 16-bit and 32-bit arithmetic and logical operations. It takes two 32-bit inputs `A` and `B`, and a 5-bit function selector `FunSel`, which determines the operation to be performed. The ALU supports a wide range of functions including addition, subtraction, bitwise logic operations (AND, OR, XOR, NAND), as well as logical, arithmetic, and circular shift operations. These operations can be executed in either 16-bit or 32-bit mode, depending on the value of `FunSel`.

The primary output of the ALU is `ALUOut`, which holds the result of the selected operation. Additionally, the ALU produces a 4-bit `FlagsOut` output, which encodes the following status flags:

- `Z` (Zero): Set when the result is zero
- `C` (Carry): Set when a carry or borrow occurs
- `N` (Negative): Set when the result is negative
- `O` (Overflow): Set when a signed overflow occurs

These flags are stored in the format `{Z, C, N, O}` and are updated synchronously with the clock on its rising edge, but only when the `WF` (Write Flag) signal is asserted.

The symbolic representation of the ALU is provided in Figure 16, while the operations supported for each `FunSel` input are detailed in the characteristic table in Figure 17. The complete schematic of the ALU implementation is shown in Figure 18.

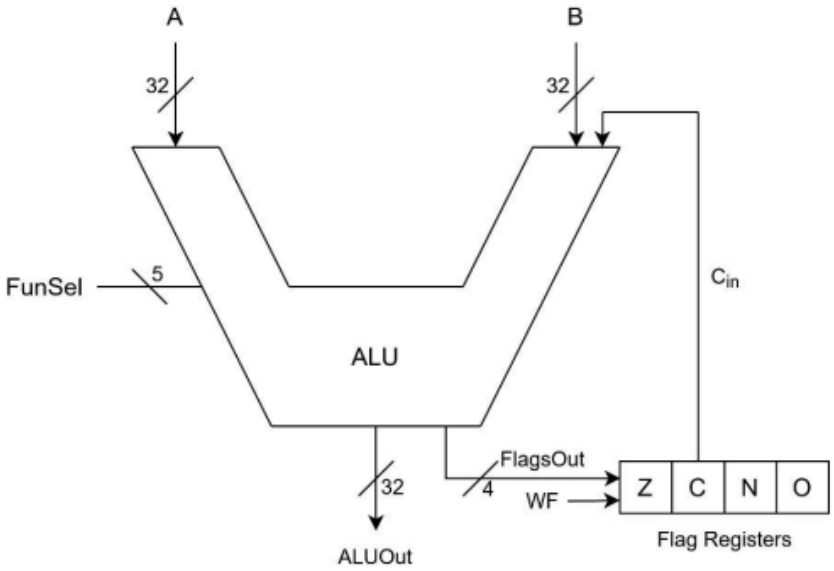


Figure 16: Graphic symbol of Arithmetic Logic Unit

FunSel	ALUOut	Z	C	N	O
00000	A (16-bit)	+	-	+	-
00001	B (16-bit)	+	-	+	-
00010	NOT A (16-bit)	+	-	+	-
00011	NOT B (16-bit)	+	-	+	-
00100	A + B (16-bit)	+	+	+	+
00101	A + B + Carry (16-bit)	+	+	+	+
00110	A – B (16-bit)	+	+	+	+
00111	A AND B (16-bit)	+	-	+	-
01000	A OR B (16-bit)	+	-	+	-
01001	A XOR B (16-bit)	+	-	+	-
01010	A NAND B (16-bit)	+	-	+	-
01011	LSL A (16-bit)	+	+	+	-
01100	LSR A (16-bit)	+	+	+	-
01101	ASR A (16-bit)	+	-	-	-
01110	CSL A (16-bit)	+	+	+	-
01111	CSR A (16-bit)	+	+	+	-

FunSel	ALUOut	Z	C	N	O
10000	A (32-bit)	+	-	+	-
10001	B (32-bit)	+	-	+	-
10010	NOT A (32-bit)	+	-	+	-
10011	NOT B (32-bit)	+	-	+	-
10100	A + B (32-bit)	+	+	+	+
10101	A + B + Carry (32-bit)	+	+	+	+
10110	A – B (32-bit)	+	+	+	+
10111	A AND B (32-bit)	+	-	+	-
11000	A OR B (32-bit)	+	-	+	-
11001	A XOR B (32-bit)	+	-	+	-
11010	A NAND B (32-bit)	+	-	+	-
11011	LSL A (32-bit)	+	+	+	-
11100	LSR A (32-bit)	+	+	+	-
11101	ASR A (32-bit)	+	-	-	-
11110	CSL A (32-bit)	+	+	+	-
11111	CSR A (32-bit)	+	+	+	-

Figure 17: Characteristic table of ALU: operations defined by `FunSel`

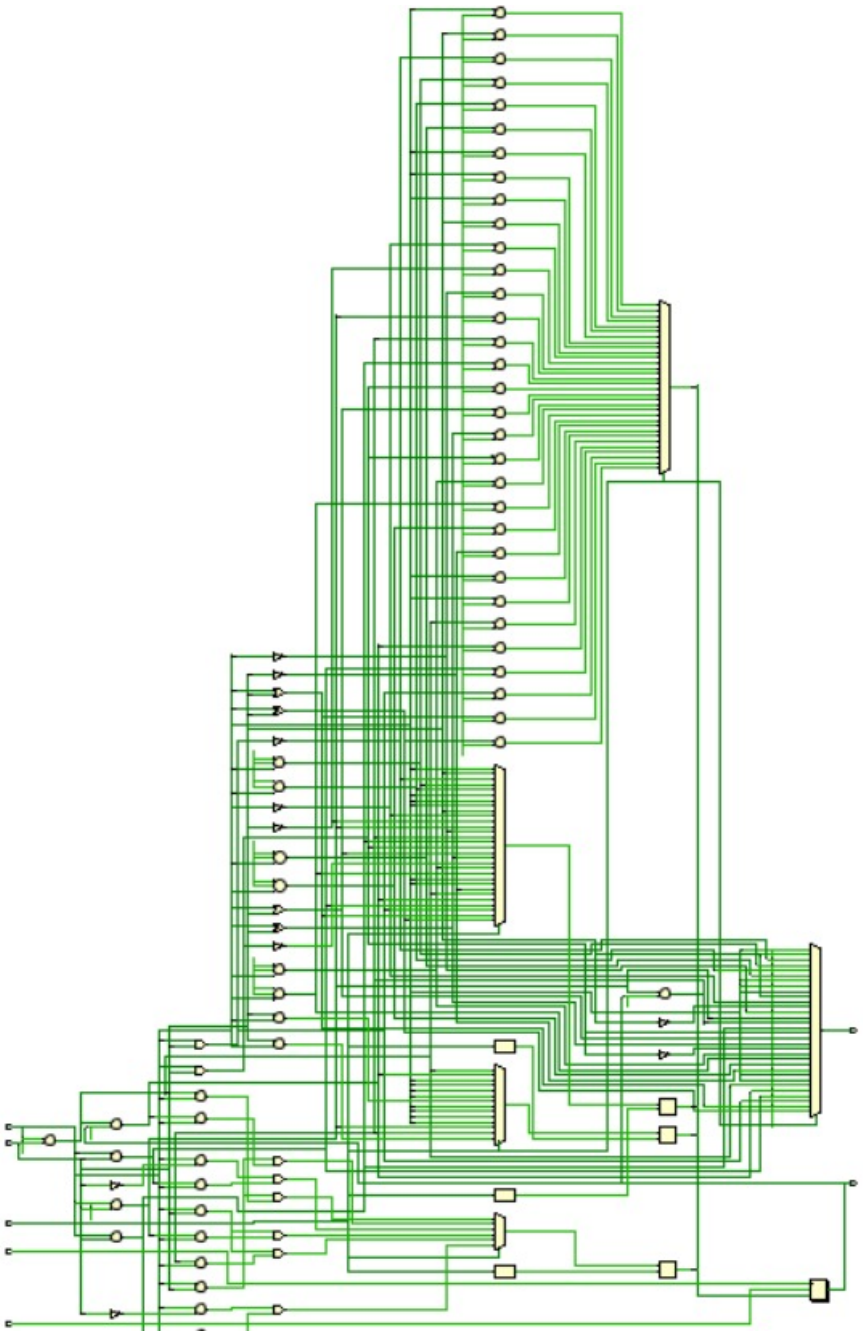


Figure 18: Arithmetic Logic Unit (ALU) schematic

2.2.4 Project Part 4

This final section brings together all previously implemented components into a complete Arithmetic Logic Unit (ALU) system. All modules from earlier parts—such as the Register File, Address Register File, Instruction Register, Data Register, and ALU—are integrated here. The only new elements introduced are the multiplexers (MuxA, MuxB, MuxC, MuxD), which handle data routing between modules. The entire system operates synchronously with a single shared `Clock` input.

The symbolic diagram of the system is shown in Figure 19, and the detailed schematic is provided in Figure 20.

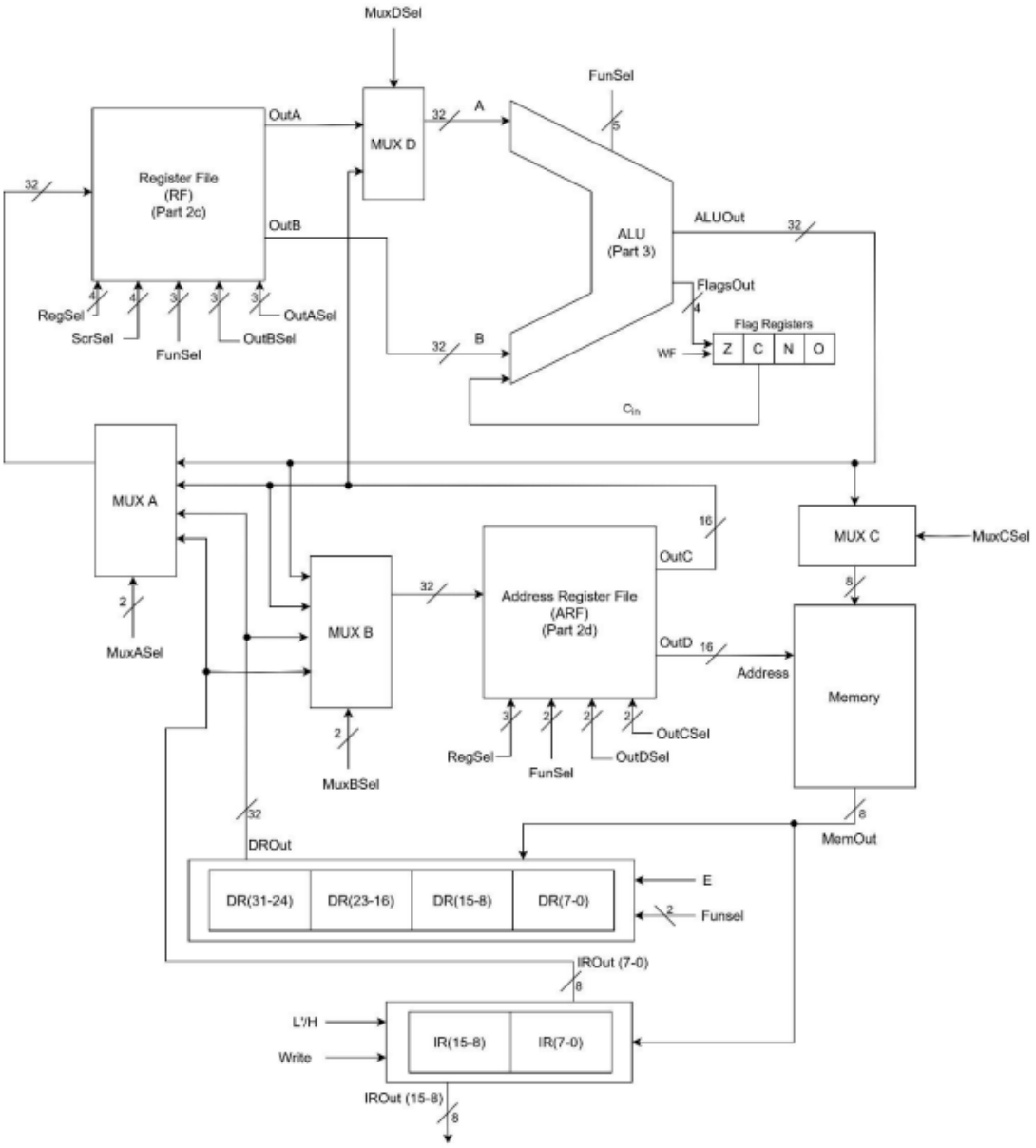


Figure 19: Symbolic representation of the Arithmetic Logic Unit System

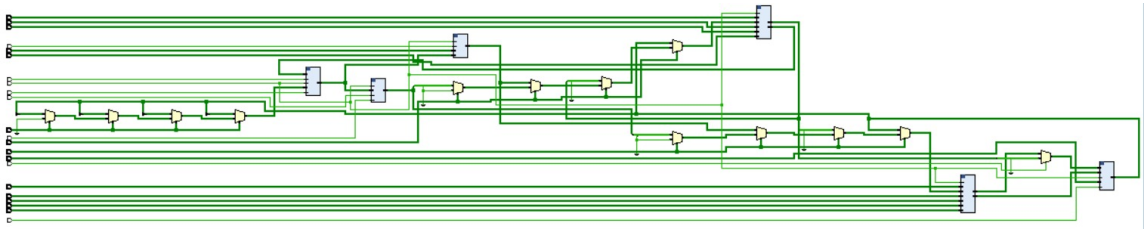


Figure 20: ALU System schematic

Multiplexer Overview

- `MuxA` selects the input for the Register File from four sources:
  - `00` – Previous ALU output (`ALUOut`)
  - `01` – Output from ARF (`OutC`) zero-extended
  - `10` – Output from Data Register (`DROut`)
  - `11` – Lower 8 bits of the Instruction Register, zero-extended
- `MuxB` selects the input for the ARF, using the same four options as MuxA.
- `MuxC` selects the 8-bit value to be written to memory:
  - `00` – Bits [7:0] of `ALUOut`
  - `01` – Bits [15:8] of `ALUOut`
  - `10` – Bits [23:16] of `ALUOut`
  - `11` – Bits [31:24] of `ALUOut`
- `MuxD` determines the first operand of the ALU:
  - `0` – Output A from the Register File (`OutA`)
  - `1` – Output C from the ARF, zero-extended

Memory Module

The system includes a memory module `MEM` that is accessed through the address provided by the ARF. Data is written to or read from memory depending on the control signals `WR` (Write) and `CS` (Chip Select). The memory output is then used as input to both the Instruction Register (IR) and the Data Register (DR), depending on control context.

Clock Synchronization

All components in this system are synchronized using a single clock input. This ensures that data flow, memory operations, and register updates occur in

3 RESULTS [15 points]

All modules were tested individually using the provided simulation environment, and all tests passed successfully. Behavioral simulation results verified that each component functioned correctly under all specified control signal combinations and data input scenarios.

Module-Level Verification

Each module was verified through simulation by checking its output against expected values. Below is a summary of some key results:

- **Register16bit & Register32bit:** Both registers passed increment, decrement, load, and clear operations under the correct enable and control signals.
- **InstructionRegister & DataRegister:** Successfully demonstrated correct loading of 8-bit input data, with appropriate handling of lower/upper byte updates and shifting behavior.
- **RegisterFile & AddressRegisterFile:** Output selections, register updates, and parallel read/write behavior matched expectations.
- **ALU:** All 16-bit and 32-bit arithmetic, logical, and shift operations produced correct results and correctly set the Z, C, N, and O flags.
- **ALU System:** Full integration passed every test case, including memory interaction, mux selections, and register coordination.

Example output from the simulation console confirms the success of the tests, as shown below:

```
[PASS] Test No: 1, Component: Q, Actual: 0x00000025, Expected: 0x00000025
[PASS] Test No: 3, Component: ALUOut, Actual: 0x00000000, Expected: 0x00000000
[PASS] Test No: 2, Component: R2, Actual: 0xfffffffe, Expected: 0xfffffffe
[PASS] Test No: 2, Component: PC, Actual: 0x0000fffe, Expected: 0x0000fffe
```

Behavioral Simulation Visuals

To further demonstrate the correct functionality of each component, waveform outputs from behavioral simulations were captured. Below are selected simulation snapshots for critical modules:

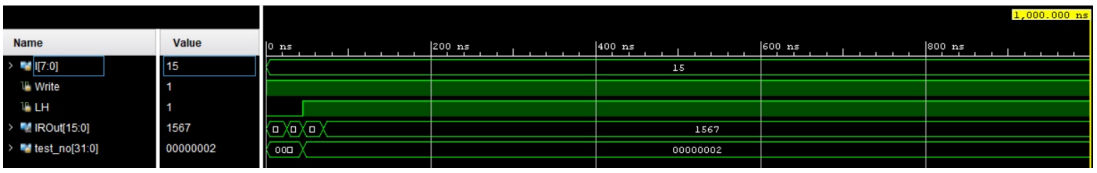


Figure 21: Simulation output of the Instruction Register

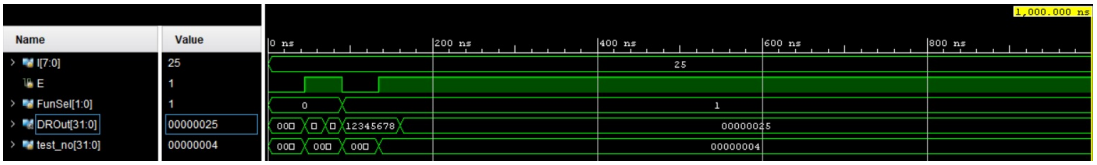


Figure 22: Simulation output of the Data Register

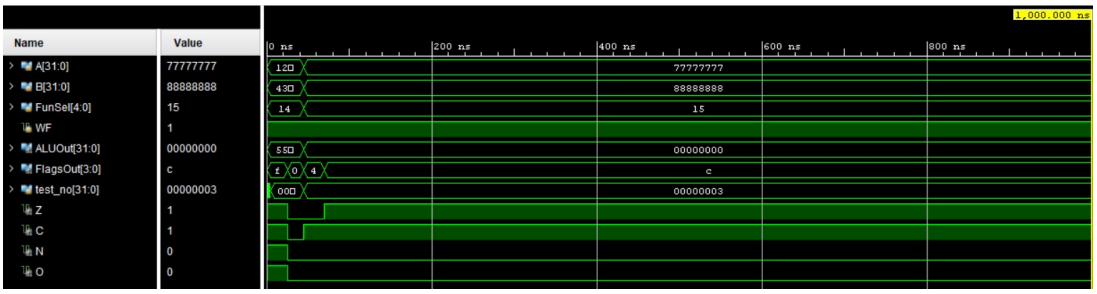


Figure 23: Simulation output of the ALU

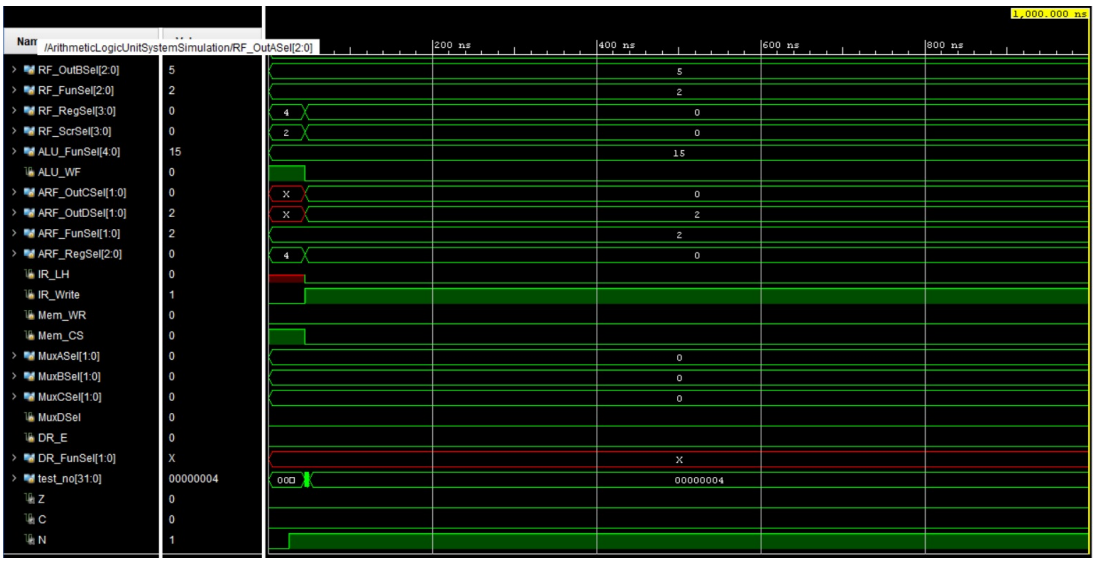


Figure 24: Simulation output of the ALU System

The combination of text-based simulation logs and waveform results confirm that all modules operate as intended, both individually and when integrated into the full ALU system.

4 DISCUSSION [25 points]

Throughout the project, Verilog HDL was used to implement and verify a digital system composed of various functional modules. Each part of the project introduced different challenges and reinforced essential concepts from digital systems and computer organization.

One of the key takeaways was the importance of synchronization using the clock signal. All modules, including registers, register files, and the ALU, were triggered on the rising edge of the clock to ensure reliable, hazard-free operation. Early on, it became clear that correct timing and enable signal management were crucial to maintain predictable module behavior.

Debugging simulation outputs was another significant aspect. While individual modules passed their tests relatively smoothly, integration in Part 4 required careful signal routing and understanding how data flows between components. The use of multiplexers in the final ALU system highlighted how selective data paths and clean modular interfaces simplify complex circuit design.

The ALU flags (Zero, Carry, Negative, Overflow) were also particularly instructive. These outputs required careful handling and served as a valuable exercise in understanding how arithmetic operations interact with status indicators in real hardware.

We also observed how important it was to read and interpret the provided testbenches. Many issues stemmed not from faulty logic but from mismatches between expected I/O names or update timing. These insights reinforced good practices for HDL-based system design, such as monitoring sensitivity lists, separating combinational and sequential logic, and modularizing code for reuse.

Simulations not only verified correctness but also helped us visually trace how each control signal affected system behavior — especially in the integrated ALU system.

5 CONCLUSION [10 points]

This project provided hands-on experience with digital hardware design using Verilog HDL, allowing us to apply theoretical knowledge from lectures in a practical and integrated way. We successfully designed, implemented, and tested components ranging from simple registers to a fully functional ALU system with memory integration.

Challenges included interpreting ambiguous module specifications, understanding how control signals like ‘Enable‘, ‘Write‘, and ‘FunSel‘ interact, and ensuring synchronized updates across clocked elements. Particularly in Part 4, integrating multiple subsystems and ensuring consistent data flow required careful debugging and logical signal tracing.

Working through these challenges improved our understanding of topics such as clock-driven design, register file structures, ALU operation, multiplexing, and memory access protocols. We also learned how to navigate simulation-based verification and respond to failing test cases methodically.

Overall, the project strengthened both our technical and problem-solving skills, and demonstrated the power of modular design and simulation in building reliable digital systems.

REFERENCES

[1] BLG222E Project1 Documentation, Istanbul Technical University, 2024-2025.