ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E

COMPUTER ORGANIZATION PROJECT 1 REPORT

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

GROUP MEMBERS:

150210006	:	Yusuf Yıldız (Group Representative)
150210016	:	Şafak Özkan Pala

SPRING 2024

Contents

1 INTRODUCTION [10 points]

In this project, using Verilog hardware description language, and the Vivado platform, we have taken the first steps of implementing (or simulating) a basic computer. Inside this implementation, we can make arithmetic and logical operations. Moreover, we can store the results of the operations in memory or registers and make these operations sequentially. The system is controlled by a Clock signal and all of the blocks are interconnected to constitute a basic computer that can handle 8-bit inputs and 16-bit inputs. We dived into binary logical and arithmetic operations with this project, and with the simulations that we have done, we validated our knowledge and our implementation. With the given test inputs, the system hopefully works.

1.1 Task Distribution

Part 1 and Part 2 were done by Şafak Özkan Pala and Part3 and Part4 were done by Yusuf Yıldız. The additional test cases are added, and simulated by the group as a whole, also, debugging was done as a whole group.

2 MATERIALS AND METHODS [40 points]

In this part, we will introduce our implementation details and the solutions that we produced for every task. The main focuses covered in this part are as follows:

- list of control inputs and corresponding functions for your design
- explanations for each constructed part
- task distribution of each group member(given in the first section)

2.1 Part-1

This task involves designing a 16-bit register with eight distinct functionalities controlled by a 3-bit control signal (FunSel) and an enable input (E). Each functionality corresponds to a specific combination of the control signals and enables various operations such as incrementing, decrementing, loading, clearing, and selectively writing to different parts of the register based on the control signals and enable status. The register operates according to the defined characteristic table and is represented by a list of functions as depicted below. We implemented our functions inside of an always posedge Clock block since registers are dependent on the Clock signal of the system. Further details of our implementation in RTL schematic form can be found in Figure 5 in the appendix section.

- 4'b0_XXX: Q+ (Retain value)
- 4'b1_000: Q-1 (Decrement) just subtracted 1 from Q
- 4'b1_001: Q+1 (Increment) just added 1 from Q
- 4'b1_010: I (Load) assigned I to Q
- 4'b1_011: 0 (Clear) override the Q by 16'h0000
- 4'b1_100: Q (15-8) $\leftarrow$ Clear, Q (7-0) $\leftarrow$ I (7-0) (Write Low)
- 4'b1_101: Q (7-0) $\leftarrow$ I (7-0) (Only Write Low)
- 4'b1_110: Q (15-8) $\leftarrow$ I (7-0) (Only Write High)
- 4'b1_111: Q (15-8) $\leftarrow$ Sign Extend (I (7)), Q (7-0) $\leftarrow$ I (7-0) (Write Low)

The first bit before the "_" is Enable(E) input and it controls whether the current value is retained or not.

2.2 Part-2

2.2.1 Part-2a

This task involves designing a 16-bit Instruction Register (IR) with a characteristic table provided. The register can store 16-bit binary data but accepts only 8 bits of input. Depending on the L'H signal, the input bus loads either the lower (bits 7-0) or higher (bits 15-8) half of the register. The characteristic table represented below details actions like retaining the current value, and loading the least or most significant byte based on control signals and the L'H signal state. Again the functions are implemented inside of the always posedge Clock block, and further details of our implementation in RTL schematic form can be found in Figure 6 in the appendix section.

The IR is used to store the instruction word. When the CPU fetches an instruction from memory, it is temporarily stored in the IR. The instruction is a binary word or code that defines a specific operation to be performed.

- 2'bX0: IR+ (Retain value)
- 2'b01: IR (7-0) $\leftarrow$ I (Load LSB) loads the 8-bit input(I) to the least significant 8 bits of IR

- 2'b11: IR (15-8) ← I (Load MSB) loads the 8-bit input(I) to the most significant 8 bits of IR

The second bit of the control input is Write flag(W) and it controls whether the current value is retained or not in the IR. Moreover, the location of the input(I) to be loaded is controlled by the first bit, which is L'H flag.

2.2.2 Part-2b

This task involves designing a system comprising four 16-bit general-purpose registers (R1, R2, R3, R4) and four 16-bit scratch registers (S1, S2, S3, S4). The system operates based on control signals such as OutASel, OutBSel, RegSel, ScrSel, and FunSel. OutASel and OutBSel control the selection of registers for output, while FunSel determines the operation to be performed on the selected registers, and the detailed function control inputs are given below. RegSel and ScrSel signals specify which general-purpose and scratch registers are enabled for the operation, respectively. Further details of our implementation in RTL schematic form can be found in Figure 7 in the appendix section.

- 3'b000: Rx-1 (Decrement)
- 3'b001: Rx +1 (Increment)
- 3'b010: I (Load)
- 3'b011: 0 (Clear)
- 3'b100: Rx (15-8) ← Clear, Rx (7-0) ← I (7-0) (Write Low)
- 3'b101: Rx (7-0) ← I (7-0) (Only Write Low)
- 3'b110: Rx (15-8) ← I (7-0) (Only Write High)
- 3'b111: Rx (15-8) ← Sign Extend (I (7)), Rx (7-0) ← I (7-0) (Write Low)

The behavior of the register file is encapsulated within an `always @(posedge Clock)` block, where specific general-purpose and scratch registers are enabled or disabled based on the control signals `RegSel` and `ScrSel`. These signals dictate which registers receive the incoming data input (`I`) and determine the operations to be executed on them. Enable signals (`R1_Enable` to `R4_Enable` and `S1_Enable` to `S4_Enable`) are generated according to the control signals, enabling or disabling specific registers for data input and operations. Each enabled register is instantiated as a `Register` module, receiving the data input, the corresponding enable signal, the function selection control signal (`FunSel`), and the clock signal (`Clock`). The outputs of these registers (`Q_R1` to `Q_S4`) hold the stored values. Finally, the values of `OutA` and `OutB` are selected based on the control signals `OutASel` and `OutBSel`, respectively, with the chosen register values assigned to these outputs. Thus, the `RegisterFile` module offers a flexible register file architecture, allowing for dynamic selection of registers, execution of operations, and output of register values as determined by the control signals.

2.2.3 Part-2c

The address register file (ARF) system comprises three 16-bit address registers: the program counter (PC), address register (AR), and stack pointer (SP). Control signals FunSel and RegSel determine the operations to be performed on the address registers and which registers are enabled. Based on RegSel, specific combinations of address registers are enabled for the application of functions selected by FunSel. The detailed function control inputs and corresponding operations are the same with the one in Part-2b. OutCSel and OutDSel control signals dictate the selection of registers whose values are output to OutC and OutD, respectively. This architecture offers flexibility in addressing operations and output selection, facilitating efficient address management within a digital system. Further details of our implementation in RTL schematic form can be found in Figure 8 in the appendix section.

The behavior of the address register file is governed by an `always @(posedge Clock)` block, indicating that the operations within the block are triggered on the rising edge of the clock signal. Within this block, specific combinations of address registers are enabled based on the `RegSel` control signal. For instance, if `RegSel` corresponds to `4'b001`, only the PC and AR registers are enabled to undergo operations determined by `FunSel`. This enables flexible configuration of the register file to suit various processing requirements.

The output ports `OutC` and `OutD` are determined by the `OutCSel` and `OutDSel` control signals, respectively. These control signals dictate which registers' values are routed to the output ports. For example, if `OutCSel` is `2'b10`, the value of the AR register is selected and assigned to the `OutC` output port. Similarly, the selected register value is assigned to the `OutD` output port based on `OutDSel`.

2.3 Part-3

The Arithmetic Logic Unit (ALU) is a computational circuit capable of performing various arithmetic and logical operations on its two 16-bit inputs, resulting in a 16-bit output. Controlled by FunSel, options of the FunSel are given below, the ALU executes operations such as addition, subtraction, bitwise operations (AND, OR, XOR, NAND),

and shifts (logical left/right, arithmetic right). Operations are performed using 2's complement logic, enabling arithmetic operations like addition and subtraction. The ALU provides versatility in computation, allowing for a wide range of arithmetic and logical operations to be executed efficiently. The ALU flags, including zero (Z), carry (C), negative (N), and overflow (O), are updated based on the result (ALUOut) of the selected operation when the Write Flag (WF) is 1. Also, the ALU can take a function control option to perform with 8-bit input, and the result is calculated considering the first 8 bits of the input, which is I(7-0). Flags and the ALUOut are calculated according to the 8-bit function outputs, the ALUOut is padded with the sign bit of the resulting 8-bit value so that the ALUOut is confirmed to be 16-bit. Further details of our implementation in RTL schematic form can be found in Figure 9 in the appendix section.

- 5'b00000: A (8-bit)
- 5'b10000: A (16-bit)
- 5'b00010: NOT A (8-bit)
- 5'b10010: NOT A (16-bit)
- 5'b00011: NOT B (8-bit)
- 5'b10011: NOT B (16-bit)
- 5'b00100: A + B (8-bit)
- 5'b10100: A + B (16-bit)
- 5'b00101: A + B + Carry (8-bit)
- 5'b10101: A + B + Carry (16-bit)
- 5'b00110: A - B (8-bit)
- 5'b10110: A - B (16-bit)
- 5'b00111: A AND B (8-bit)
- 5'b10111: A AND B (16-bit)
- 5'b01000: A OR B (8-bit)
- 5'b11000: A OR B (16-bit)
- 5'b01001: A XOR B (8-bit)

- 5'b11001: A XOR B (16-bit)
- 5'b01010: A NAND B (8-bit)
- 5'b11010: A NAND B (16-bit)
- 5'b01011: LSL A (8-bit)
- 5'b11011: LSL A (16-bit)
- 5'b01100: LSR A (8-bit)
- 5'b11100: LSR A (16-bit)
- 5'b01101: ASR A (8-bit)
- 5'b11101: ASR A (16-bit)
- 5'b01110: CSL A (8-bit)
- 5'b11110: CSL A (16-bit)
- 5'b01111: CSR A (8-bit)
- 5'b11111: CSR A (16-bit)

The `ArithmeticLogicUnit` module is designed to perform various arithmetic and logical operations on two input operands (A and B) based on the control signal `FunSel`. Here are the implementation details of the module:

2.3.1 Module Inputs

- **A (16-bit)**: Input operand A.
- **B (16-bit)**: Input operand B.
- **FunSel (5-bit)**: Function selection control signal.
- **WF (1-bit)**: Write Flag for controlling the update of output flags.
- **Clock (1-bit)**: Clock input for synchronous operations.

2.3.2 Module Outputs

- **ALUOut (16-bit)**: Result of the ALU operation.
- **FlagsOut (4-bit)**: Output flags including Zero (Z), Carry (C), Negative (N), and Overflow (O).

2.3.3 Internal Signals

- **Result (17-bit)**: Result of ALU operation in 16-bit.
- **Result8Bit (9-bit)**: Result of ALU operation in 8-bit.
- **Flags (4-bit)**: Internal flags including Z, C, N, and O.
- **Operation16Bit, OperationArithmetic, AdditionFlag, ASRFlag, OperationShiftFlag, CircularShiftFlag**: Internal control signals for operation type, arithmetic operation, addition flag, arithmetic shift right flag, shift operation flag, and circular shift flag respectively.

The details of the internal flags we added are as below:

- **Operation16Bit**: This flag indicates whether the current operation is operating on 16-bit operands. If set to 1, it signifies that the operation is performed on 16-bit operands; otherwise, it indicates an 8-bit operation and is used for setting the FlagsOut and ALUOut according to internal temporary values.
- **OperationArithmetic**: This flag is set to 1 during arithmetic operations, such as addition and subtraction. It helps differentiate between arithmetic and logical operations. Since just the arithmetic operations and the shift operations change the Carry(C) flag, it indicates whether to change its value.
- **AdditionFlag**: This flag is set during addition operations to assist in overflow detection. It is used to determine if overflow occurs when two operands with the same sign result in a result with a different sign. The control mechanism of overflow changes with subtraction and addition. For the addition operation, the overflow only occurs in 2 scenarios: (+) + (+) = (-) and (-) + (-) = (+). Meanwhile, for the subtraction operation, overflow only occurs in 2 scenarios: (+) - (-) = (-) and (-) - (+) = (+).
- **ASRFlag**: This flag is specific to arithmetic shift right operations. It is set when performing arithmetic right shifts to maintain the sign bit. This flag is added because shift operations do not affect the negative flag(N) like all other shift operations, so this flag distinguishes the ASR operation.
- **OperationShiftFlag**: This flag is set when any type of shift operation is performed, including logical and arithmetic shifts. It helps control the behavior of certain operations, especially with respect to updating flags. This flag is added because shift operations do not affect the overflow flag(O).

- **CircularShiftFlag**: This flag is set during circular shift operations. It distinguishes circular shifts from other types of shifts and helps control the operation's behavior. This flag is used to set the ALUOut inside of `always @(posedge Clock)` block. If the ALUOut is set inside of `always @(*)` block like all other operations, when the carry flag(C) changes it operates again and shifts the result again, so the result changes incorrectly.

These internal flags play crucial roles in controlling the behavior of the ArithmeticLogicUnit module and ensuring correct operation execution and flag handling.

2.3.4 ALU Operations

The module supports various ALU operations including addition, subtraction, bitwise AND/OR/XOR, logical shifts, and circular shifts. The operations are determined based on the value of `FunSel` signal.

2.3.5 Flags Calculation

- **Zero Flag (Z)**: Set if the result is zero.
- **Carry Flag (C)**: Set based on carry out from addition or borrow in from subtraction.
- **Negative Flag (N)**: Set based on the sign of the result.
- **Overflow Flag (O)**: Set in case of overflow during arithmetic operations.

2.3.6 Operation Control

The module sets internal control signals based on the operation type and size. Shift operations also update the corresponding flags based on the shifted result.

2.3.7 Clock Synchronization

The output `FlagsOut` is updated synchronously with the positive edge of the clock signal when `WF` is asserted.

Overall, the `ArithmeticLogicUnit` module provides a flexible and efficient implementation of various arithmetic and logical operations with support for both 8-bit and 16-bit operands.

2.4 Part-4

In this part, we have implemented the ALU System, which is a system of digital circuit components and is capable of computing arithmetical and logical operations on binary signed integers. In this design, we have several interconnected components like the memory module, which was provided to us in the homework files, and the modules that we have implemented in the previous parts of the project; the ALU, the Register File, the Address Register File (ARF), and the Instruction Register (IR). The system load is read from memory, where the data and instructions are stored. The IR in the system is not utilized fully, in this implementation, it uses half of the bits stored inside it. operations. The RF inside the system has 8 registers inside the circuitry for ALU to make consecutive operations faster. The Address Register File contains the Program Counter (PC), which is to represent the current address in progress, Address Register (AR), which points to an address, and Stack Pointer (SP), which currently does not do anything. The current diagram also has some control inputs (selectors) for choosing the operations that are going to happen in ALU and the flow of the inputs to the ALU, memory, and the register files. All in all, although this system can give outputs with some help outside (most of the selector inputs come into play from the outside environment) it seems it is not completed, yet.

Further details of our implementation in RTL schematic form can be found in Figure 10 in the appendix section.

3 RESULTS [15 points]

Give your results what you get during the project. You can also add the table, image, etc.

In the process of the development of each module, we used the simulation files given to check whether the module behaves as it is supposed to. This made it easier for us to debug the whole organization when we faced problems. In addition to this, we wrote additional 28 test cases to the ArithmeticLogicUnit module, since this module is the main operational unit and has a high amount of operation options. Below are the results of our functional modules passing the tests both we wrote and the given:

1. Part-1: Simulation for Figure 11
2. Part-2a: Simulation for Figure 12
3. Part-2b: Simulation for Figure 13
4. Part-2c: Simulation for Figure 14

5. Part-3: Simulation with the given tests for Figure 15
6. Part-3(continued): Simulation with our additional tests for Figure 16
7. Part-4: Simulation for Figure 17

All images are provided in the appendices for a more organized view, so they are just referred to in this section.

These simulation cases and results will be investigated in detail in the discussion section below.

4 DISCUSSION [25 points]

Please explain, analyze, and interpret what you have done during the project.

tum testler icin tablo seklinde bir sey olusturulabilir

4.1 Part-1

Test Case	Q (Actual Value)	I	E	FunSel	Q (Expected Value)
1	16’h0025	16’h0072	0	3’b000	16’h0025
2	16’h0025	16’hXXXX	1	3’b000	16’h0024
3	16’h0025	16’hXXXX	0	3’b001	16’h0025
4	16’h0025	16’hXXXX	1	3’b001	16’h0026
5	16’h0025	16’h0012	0	3’b010	16’h0025
6	16’h0025	16’h0012	1	3’b010	16’h0012
7	16’h0025	16’hXXXX	0	3’b011	16’h0025
8	16’h0025	16’hXXXX	1	3’b011	16’h0000

Table 1: Test Cases for RegisterSimulation Module

In the first test, since enable 0, 3b’000, i.e. the decrement operation was not written to the result. In the second test, since enable 1 was given, the decrement operation was applied and written to the result. Again, in the third test, since the enable was 0, the Q set at the beginning kept the value 16’h0025, but in the next test, the value was increased by 1 by giving the enable 1. In the fifth test, although FunSel was given as 3b’010, i.e. Load, the value of 16’h0025 given at the beginning was maintained because the enable was not 1. But in the sixth test, enable is also given and the result is input loaded to Q. In the last two tests the clear function of the register is tested and again when enable is 1 the result is cleared as 16’h0000. As a result, the register module we wrote successfully passed all the tests given with the assignment.

4.2 Part-2a

Test Case	IROut (Actual Value)	I	Write	LH	IROut (Expected Value)
1	16’h2367	8’h15	1	0	16’h2315
2	16’h2367	8’h15	1	1	16’h1567

Table 2: Test Cases for InstructionRegisterSimulation Module

Since Write 1 is given in both tests in this section, both operations will be reflected in the result. Our results are correct as expected and in the first test the input is written to LSB 8-bit because LH is 0, similarly in the second test the input is successfully written to MSB 8-bit because LH is 1.

4.3 Part-2b

Test Case	RegSel	ScrSel	FunSel	I	OutASel	OutBSel	OutA	OutB
1	4’b1111	4’b1111	3’bXXX	16’hXXXX	3’b000	3’b001	16’h1234	16’h5678
2	4’b0101	4’b1010	3’b010	16’h3548	3’b001	3’b101	16’h1234	16’h3548

Table 3: Test Cases for RegisterFileSimulation Module

In the first test OutASel was set to 3’b000 and R1 was selected, OutBSel was set to 3b’001 and R2 was selected, but since RegSel was 4’b1111, i.e. all registers were disable, the function would not be executed no matter what and the results came as set in the simulation. In the second test, RegSel 4’b0101 was used to enable R1 and R3, in the same way RegSel 4’b1010 was used to enable S2 and S4 registers and FunSel 3’b010, i.e. Load operation was applied to these enabled registers. Since OutASel is set to read R2 and OutBSel is set to read S2, no load operation is applied from OutA and S2 is read while reading the 16’h1234 value set in the previous test.

4.4 Part-2c

Test Case	RegSel	FunSel	I	OutCSel	OutDSel	OutC	OutD
1	3’b111	3’bXXX	16’hXXXX	2’b00	2’b11	16’h1234	16’h5678
2	4’b010	3’b010	16’h3548	2’b10	2’b01	16’h1234	16’h3548

Table 4: Test Cases for AddressRegisterFileSimulation Module

In this section, since RegSel is given as 3’b111, all registers will retain their current values. The values of PC and SP registers are entered by the user, and AR is selected for OutC and SP for OutD, which are read correctly. In the second test, all registers are initially set to 16'h1234, and then PC and SP registers are enabled by RegSel. The Load function is selected by FunSel and applied to these two registers. Since no operation is applied to AR, its initial value is preserved, and an input is written to SP. As a result, in this section, we successfully obtained the correct result in the simulation provided.

4.5 Part-3

Test Case	A	B	FlagsOut (Actual)	FunSel	WF	ALUOut	FlagsOut (Expected)
1	16'h1234	16'h4321	4'b1111	5'b10100	1	16'h5555	4'b1111
2	16'h1234	16'h4321	4'b0000	5'b10100	1	16'h5555	4'b0000
3	16'h7777	16'h8889	4'b0000	5'b10101	1	16'h0001	4'b1100

Table 5: Given Test Cases for ArithmeticLogicUnitSimulation Module

The second test case only includes a Clock signal, which is used to control whether FlagsOut are updated with the clock or not. In the table above, the basic operations of the ALU are represented, and the final results are obtained from the implemented ALU itself. Since some of the flags can be updated during these operations, the chosen operations demonstrate the possibilities of changes in flags. Here, the provided test cases are presented in tabular form, showing that our code, according to the implementation details provided above, correctly produces the expected results.

Test Case	A	B	FlagsOut (Actual)	FunSel	WF	ALUOut	FlagsOut (Expected)
4	8'b1010_1010	8'b1100_1100	4'b0111	A XOR B 8 BIT	1	16'h0066	4'b0101
5	8'b0011_0011	8'b1111_1111	4'b0000	A NAND B 8 BIT	1	16'hffcc	4'b0010
6	8'b1011_0101	8'b1011_0101	4'b0001	NOT A 8 BIT	1	8'b0100_1010	4'b0001
7	16'hacd5	16'hf0f0	4'b0101	A AND B 16 BIT	1	16'ha0d0	4'b0111

Table 6: Logical Operation Test Cases for ArithmeticLogicUnitSimulation Module

The logical operations section has been implemented correctly according to the requirements provided in the assignment. In this section, we benefited from the knowledge acquired in our previous course, Digital Circuits. The results matched our expectations based on the simulation.

Test Case	A	B	FlagsOut (Actual)	FunSel	WF	ALUOut	FlagsOut (Expected)
8	8'b1011_0101	8'b1011_0101	4'b0000	A + B + Carry 8 BIT	1	8'b0110_1011	4'b0101
9	8'b1011_1101	8'b0011_0101	4'b0000	A - B 8 BIT	1	16'hff88	4'b0110
10	8'b1110_0010	8'b1100_1110	4'b0000	A - B 8 BIT	1	8'b0001_0100	4'b0100
11	16'hffbd	16'h0035	4'b0000	A - B 16 BIT	1	16'hff88	4'b0110
12	8'b1111_1101	8'b0111_1111	4'b0000	A - B 8 BIT	1	8'b0111_1110	4'b0101
13	16'h4e20	16'h9e58	4'b0000	A - B 16 BIT	1	16'hafc8	4'b0011

Table 7: Arithmetic Operation Test Cases for ArithmeticLogicUnitSimulation Module

In the arithmetic operations part of ALU, the operations were designed to be interpreted as signed bits (2's complement form), therefore, all of the operations' results are depending on the signed bit representations. The test cases are chosen deliberately as edge cases and to show flag updates clearly.

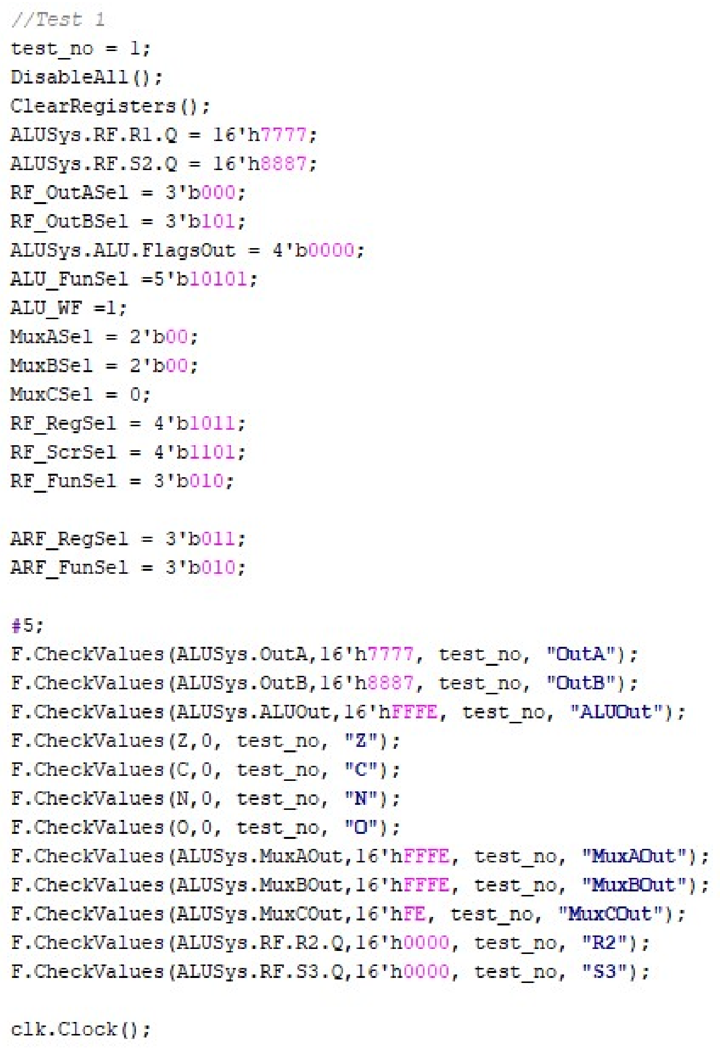
What we want to clearly explain here is our own method used in all 8-bit ALU operations. There is no definitive information on how the 8-bit operations are written to ALUOut, as discussed both with the assistant and within our course group on WhatsApp. Based on an answer provided by the assistant, according to what we learned in our previous course, Digital Circuits, when converting an 8-bit number to a 16-bit one, sign extension should be applied. Therefore, we applied sign extension to all 8-bit operations at the end, ensuring that in the overall system where the ALU is used, no data corruption occurs. We believe that padding the beginning of an 8-bit number with zeros would alter its actual value, so we consider this method incorrect, hence we are making this clarification openly. Since there is no explanation regarding this detail in the assignment instructions, we consider it crucial to avoid losing points on this issue.

Test Case	**A**	**B**	**FlagsOut (Actual)**	**FunSel**	**WF**	**ALUOut**	**FlagsOut (Expected)**
14	16'hc000	16'h0000	4'b0000	LSL A 16 BIT	1	16'h8000	4'b0110
15	16'h0001	16'h0000	4'b0001	LSR A 16 BIT	1	16'h0000	4'b1101
16	16'he003	16'h0000	4'b0001	ASR A 16 BIT	1	16'hf001	4'b0101
17	16'he003	16'h0000	4'b0110	ASR A 16 BIT	1	16'hf001	4'b0110
18	16'h0000	16'h0000	4'b0101	CSR A 16 BIT	1	16'8000	4'b0011
19	16'h0001	16'h0000	4'b0010	CSR A 16 BIT	1	16'h0000	4'b1100
20	16'h8000	16'h0000	4'b0001	CSL A 16 BIT	1	16'h0000	4'b1101
21	16'hc000	16'h0000	4'b0100	CSL A 16 BIT	1	16'h8001	4'b0110
22	16'hca30	16'h0000	4'b0100	CSL A 16 BIT	1	16'h9461	4'b0110
23	8'b1100_0001	8'b0000_0000	4'b0001	LSL A 8 BIT	1	16'hff82	4'b0111
24	8'b0000_0001	16'h0000	4'b0011	LSR A 8 BIT	1	16'h0000	4'b1101
25	8'b1100_0011	16'h0000	4'b0011	ASR A 8 BIT	1	16'hffe1	4'b0111
26	8'b0001_1100	16'h0000	4'b0111	ASR A 8 BIT	1	8'b0000_1110	4'b0011
27	8'b0000_0000	16'h0000	4'b0101	CSR A 8 BIT	1	16'hff80	4'b0011
28	8'b0000_0001	16'h0000	4'b0010	CSR A 8 BIT	1	16'h0000	4'b1100
29	8'b1000_0110	16'h0000	4'b0101	CSL A 8 BIT	1	16'h000d	4'b0101
30	8'b1000_0000	16'h0000	4'b0000	CSL A 8 BIT	1	16'h0000	4'b1100
31	8'b0110_0001	16'h0000	4'b0100	CSL A 8 BIT	1	16'hffc3	4'b0010

Table 8: Shift Operation Test Cases for ArithmeticLogicUnitSimulation Module

Shift operations have been tested with our own challenging tests, and the outputs provided above should be expected based on our understanding of logical circuits. We would like to add something: in Circular Shift operations, a kind of bug occurs if a different control mechanism is not implemented. When the carry changes, the shift operation is repeated, and the result does not match what is shown in the assignment PDF. However, this situation was also observed in test3 from the provided ALU tests. If the goal is to design a properly functioning ALU, then in this test, the result should not increase again when the carry changes. As mentioned in the PDF regarding Circular Shift results and as observed in external operations, I believe only the version shifted once should be expected from us. To address this, we prevented this bug in our code by keeping track of the necessary flags, and our simulations work perfectly fine. We wanted to provide this detail due to potential disagreements during the evaluation process. We believe that in all operations where Carry directly affects the result, the result should be written in the ALU's positive Clock edge, so that the changed Carry does not inadvertently set our result incorrectly.

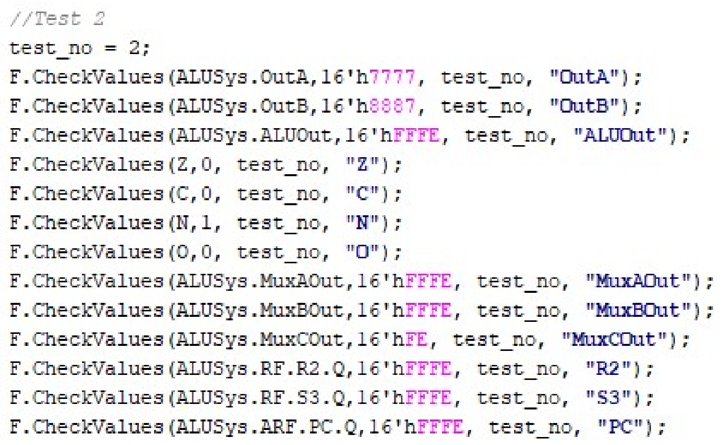
4.6 Part-4



```
//Test 1
test_no = 1;
DisableAll();
ClearRegisters();
ALUSys.RF.R1.Q = 16'h7777;
ALUSys.RF.S2.Q = 16'h8887;
RF_OutASel = 3'b000;
RF_OutBSel = 3'b101;
ALUSys.ALU.FlagsOut = 4'b0000;
ALU_FunSel =5'b10101;
ALU_WF =1;
MuxASel = 2'b00;
MuxBSel = 2'b00;
MuxCSel = 0;
RF_RegSel = 4'b1011;
RF_ScrSel = 4'b1101;
RF_FunSel = 3'b010;

ARF_RegSel = 3'b011;
ARF_FunSel = 3'b010;

#5;
F.CheckValues(ALUSys.OutA,16'h7777, test_no, "OutA");
F.CheckValues(ALUSys.OutB,16'h8887, test_no, "OutB");
F.CheckValues(ALUSys.ALUOut,16'hFFFE, test_no, "ALUOut");
F.CheckValues(Z,0, test_no, "Z");
F.CheckValues(C,0, test_no, "C");
F.CheckValues(N,0, test_no, "N");
F.CheckValues(O,0, test_no, "O");
F.CheckValues(ALUSys.MuxAOut,16'hFFFE, test_no, "MuxAOut");
F.CheckValues(ALUSys.MuxBOut,16'hFFFE, test_no, "MuxBOut");
F.CheckValues(ALUSys.MuxCOut,16'hFE, test_no, "MuxCOut");
F.CheckValues(ALUSys.RF.R2.Q,16'h0000, test_no, "R2");
F.CheckValues(ALUSys.RF.S3.Q,16'h0000, test_no, "S3");

clk.Clock();
```

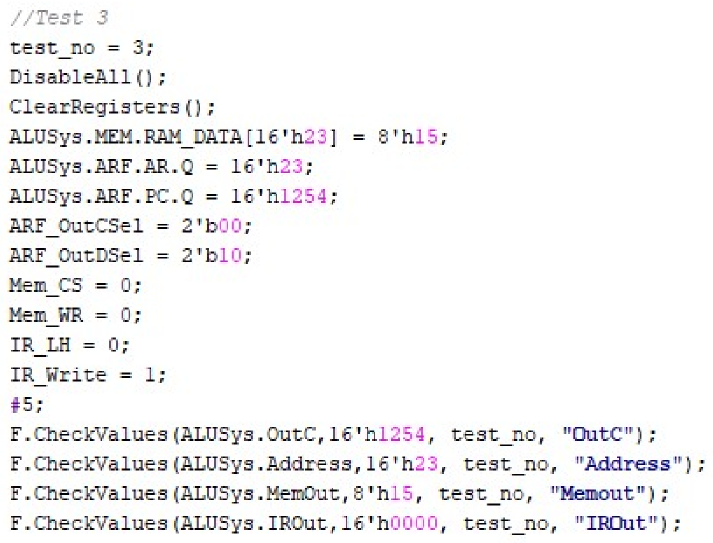
Figure 1: Test 1 for ArithmeticLogicUnitSystemSimulation Module

In the final section of the assignment, we brought together all the modules we wrote in previous sections to design a complete system. In this part, after correctly establishing all wire connections, block designs, and module initializations in our code, we successfully



```
//Test 2
test_no = 2;
F.CheckValues(ALUSys.OutA,16'h7777, test_no, "OutA");
F.CheckValues(ALUSys.OutB,16'h8887, test_no, "OutB");
F.CheckValues(ALUSys.ALUOut,16'hFFFE, test_no, "ALUOut");
F.CheckValues(Z,0, test_no, "Z");
F.CheckValues(C,0, test_no, "C");
F.CheckValues(N,1, test_no, "N");
F.CheckValues(O,0, test_no, "O");
F.CheckValues(ALUSys.MuxAOut,16'hFFFE, test_no, "MuxAOut");
F.CheckValues(ALUSys.MuxBOut,16'hFFFE, test_no, "MuxBOut");
F.CheckValues(ALUSys.MuxCOut,16'hFE, test_no, "MuxCOut");
F.CheckValues(ALUSys.RF.R2.Q,16'hFFFE, test_no, "R2");
F.CheckValues(ALUSys.RF.S3.Q,16'hFFFE, test_no, "S3");
F.CheckValues(ALUSys.ARF.PC.Q,16'hFFFE, test_no, "PC");
```

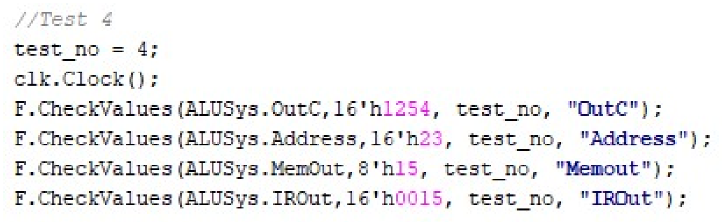
Figure 2: Test 2 for ArithmeticLogicUnitSystemSimulation Module



```
//Test 3
test_no = 3;
DisableAll();
ClearRegisters();
ALUSys.MEM.RAM_DATA[16'h23] = 8'h15;
ALUSys.ARF.AR.Q = 16'h23;
ALUSys.ARF.PC.Q = 16'h1254;
ARF_OutCSel = 2'b00;
ARF_OutDSel = 2'b10;
Mem_CS = 0;
Mem_WR = 0;
IR_LH = 0;
IR_Write = 1;
#5;
F.CheckValues(ALUSys.OutC,16'h1254, test_no, "OutC");
F.CheckValues(ALUSys.Address,16'h23, test_no, "Address");
F.CheckValues(ALUSys.MemOut,8'h15, test_no, "Memout");
F.CheckValues(ALUSys.IROut,16'h0000, test_no, "IROut");
```

Figure 3: Test 3 for ArithmeticLogicUnitSystemSimulation Module

passed all the tests provided with the assignment. We confirmed once again that our previous modules were functioning correctly, and additionally, by implementing the MUX, we reinforced our understanding of logical design concepts.



```
//Test 4
test_no = 4;
clk.Clock();
F.CheckValues(ALUSys.OutC,16'h1254, test_no, "OutC");
F.CheckValues(ALUSys.Address,16'h23, test_no, "Address");
F.CheckValues(ALUSys.MemOut,8'h15, test_no, "Memout");
F.CheckValues(ALUSys.IROut,16'h0015, test_no, "IROut");
```

Figure 4: Test 4 for ArithmeticLogicUnitSystemSimulation Module

5 CONCLUSION [10 points]

Throughout the project we did not face any problems till the ALU part, there we faced a bit of difficulty with writing the checks for the flags and ALUOut especially in the arithmetic and shift operations because in the homework it was not specified how these operations should be carried out. Afterwards we learned how to do it using the updates that were posted on ninova. We also found it troublesome to deal with the faulty testbench to test our whole ALU system. As for what we learned, we gained knowledge in writing modules using behavioral Verilog, and how to develop a software in a simulation platform Vivado, how to design and test adequate and overarching simulations, how the Clock cycles affect the logical design, and understood how an ALU system behaves using 2's complement logic with a quite number of different operations including boolean operations, arithmetic operations, and shifting operations etc for both 8-bit and 16-bit inputs.

We have successfully written codes based on our own knowledge and what was taught to us in our lessons, and if our codes are examined, as can be seen, we have done this using the most understandable and good coding practices. Our hope is that our results, which we believe to be correct, will also pass the verification stage successfully. It is our humble criticism that more detailed explanations be provided in the next assignments.

6 Appendix

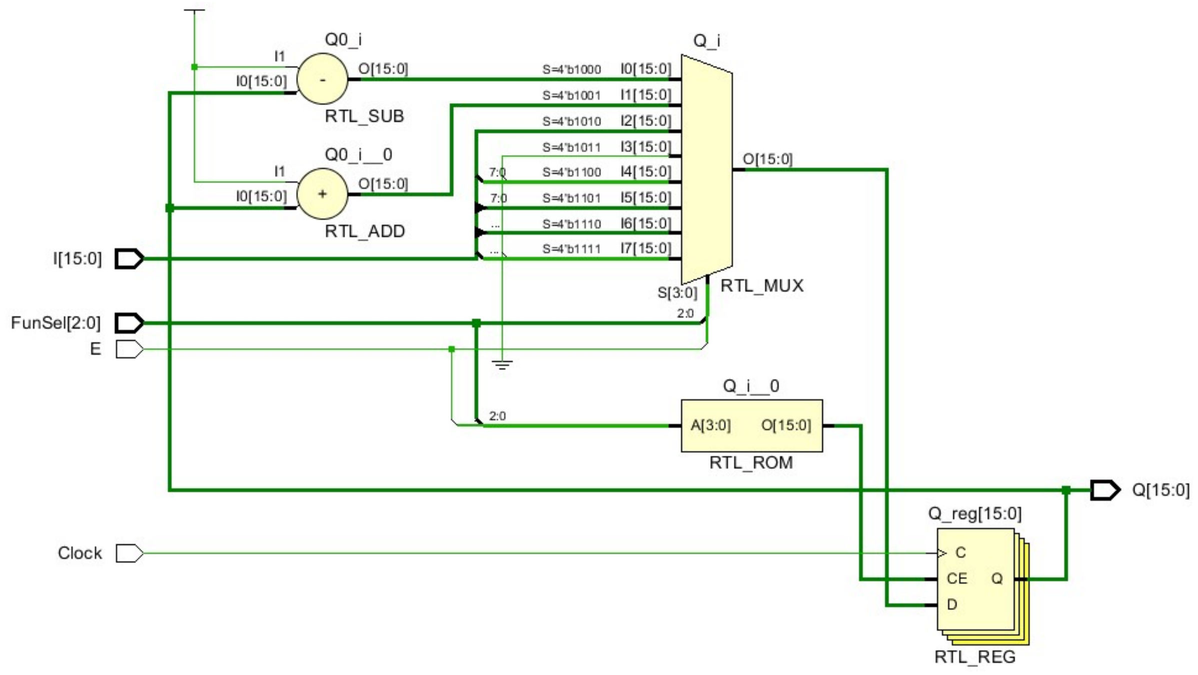


Figure 5: RTL Schematic for Part-1 (Register)

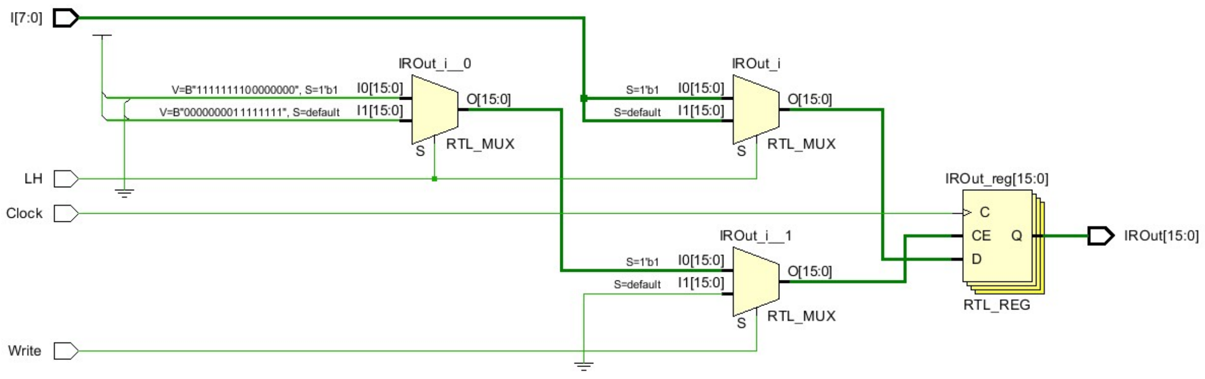


Figure 6: RTL Schematic for Part-2a (InstructionRegister)

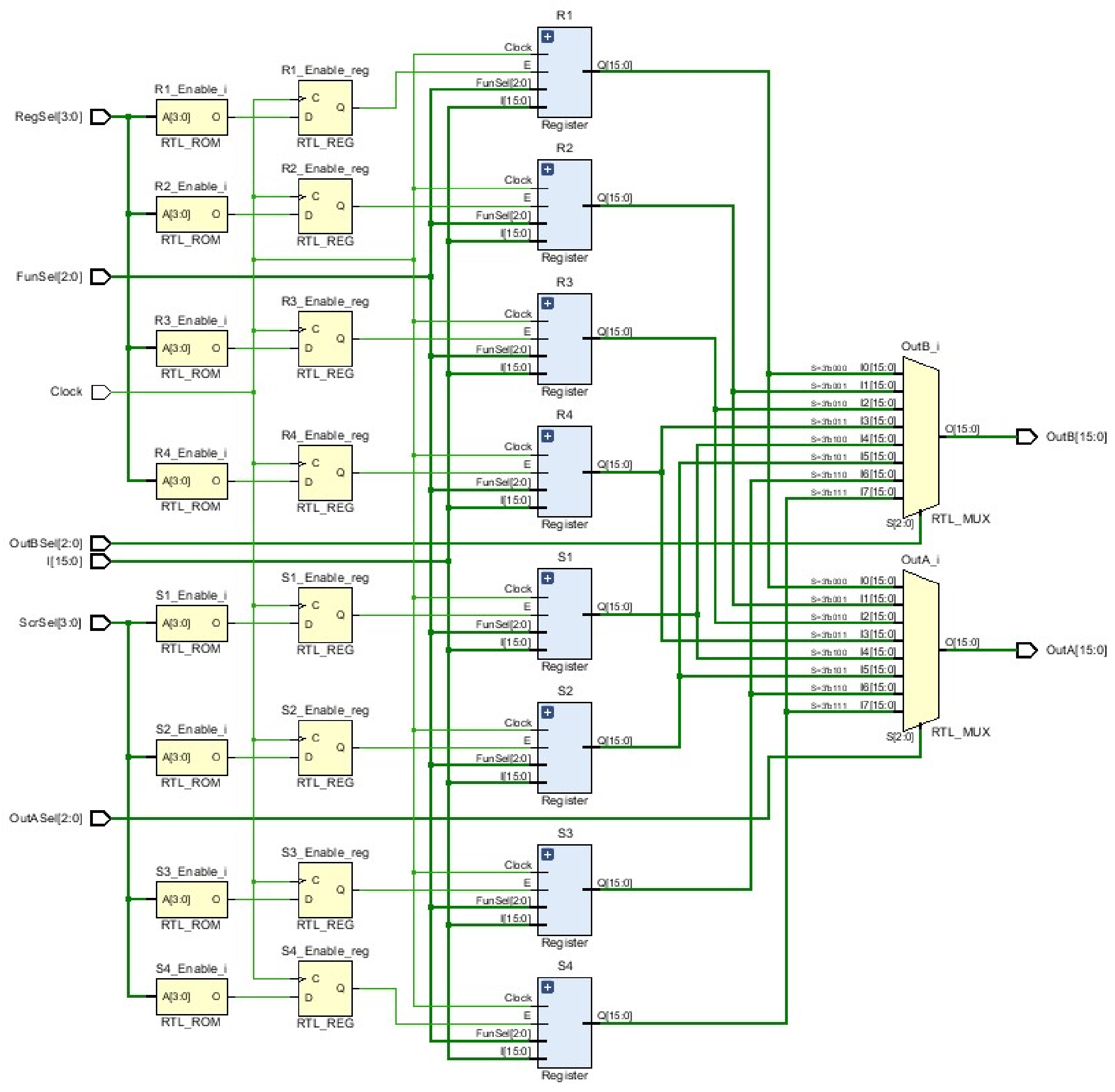


Figure 7: RTL Schematic for Part-2b (RegisterFile)

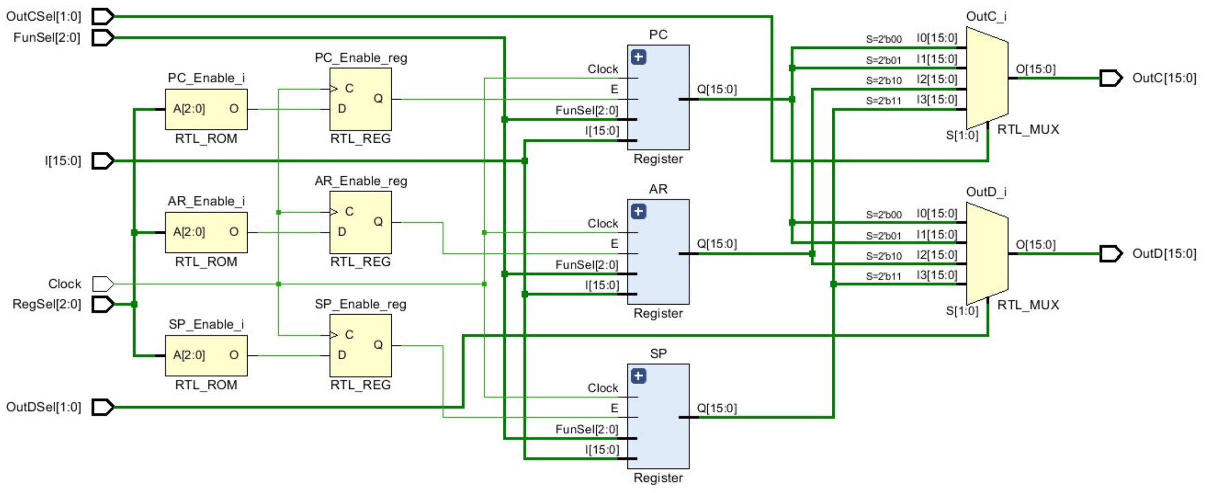


Figure 8: RTL Schematic for Part-2c (AddressRegisterFile)

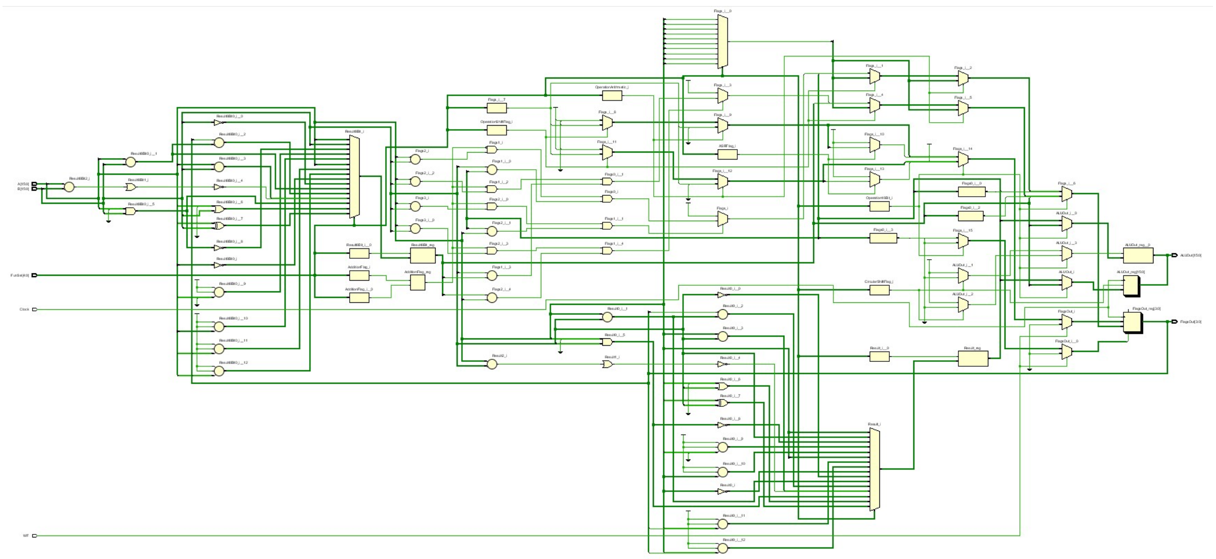


Figure 9: RTL Schematic for Part-3 (ArithmeticLogicUnit)

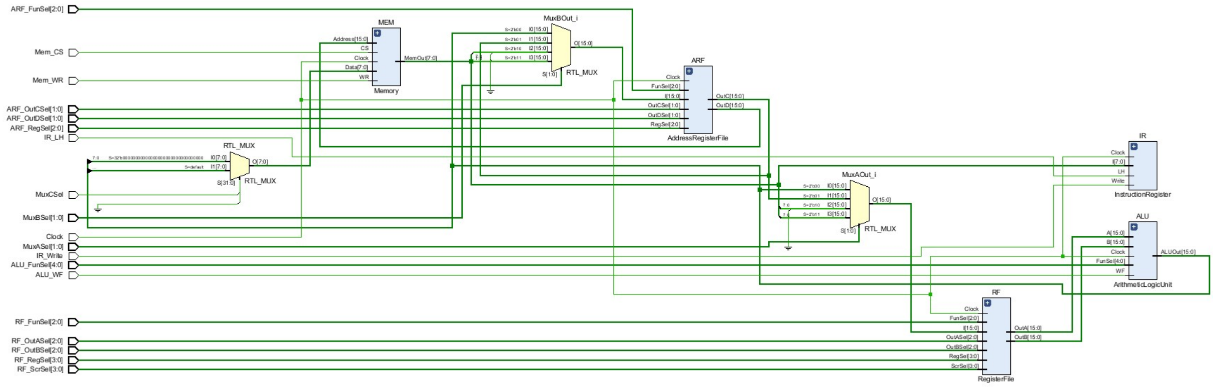


Figure 10: RTL Schematic for Part-4 (ArithmeticLogicUnitSystem)

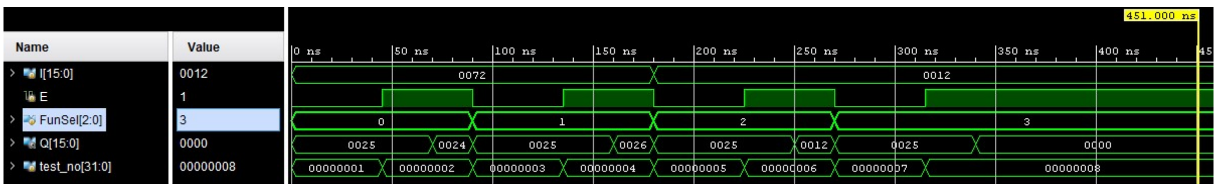


Figure 11: Simulation for Part-1 (Register)

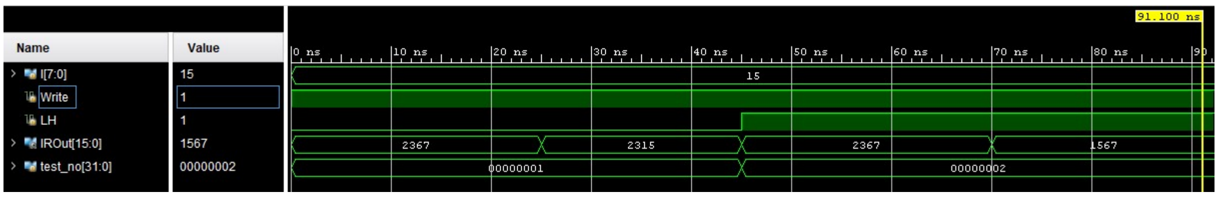


Figure 12: Simulation for Part-2a (InstructionRegister)

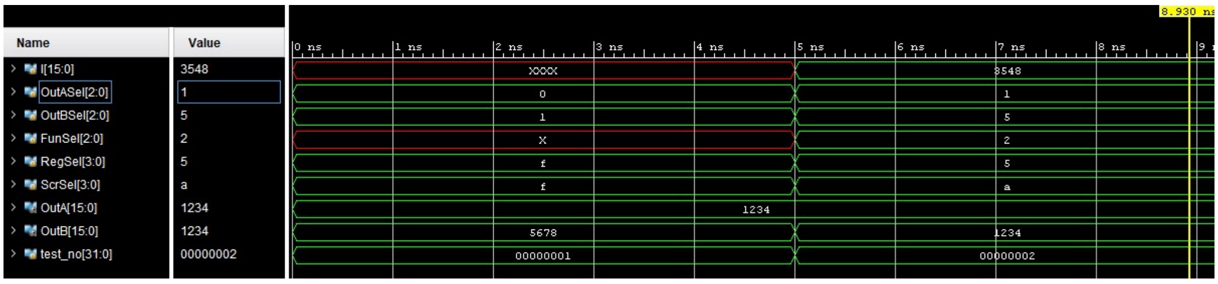


Figure 13: Simulation for Part-2b (RegisterFile)

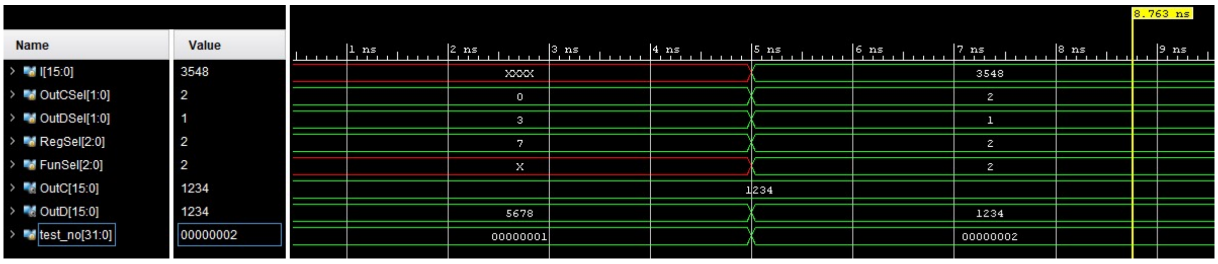


Figure 14: Simulation for Part-2c (AddressRegisterFile)

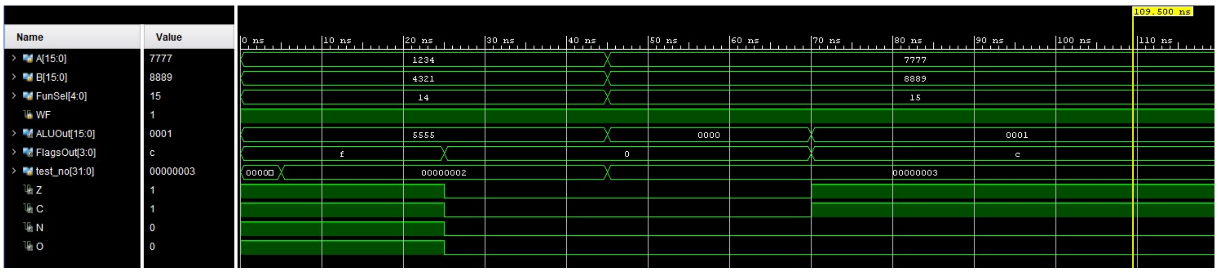


Figure 15: Simulation for Part-3 with the given tests (ArithmeticLogicUnit)

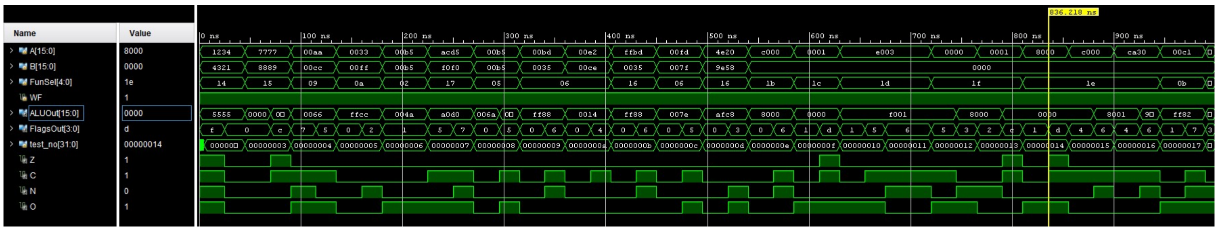
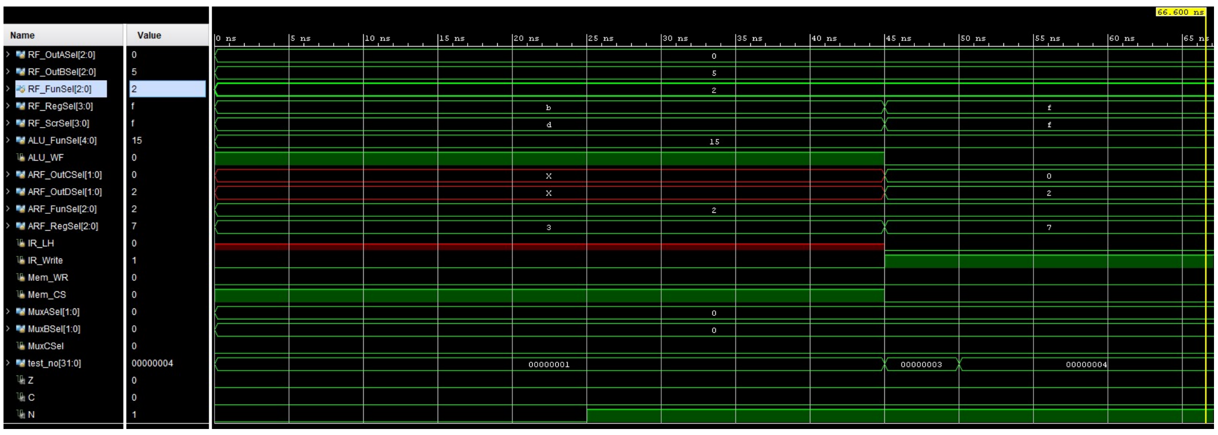


Figure 16: Simulation for Part-3, continued, with additional tests (ArithmeticLogicUnit)



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

Figure 17: Simulation for Part-4 (ArithmeticLogicUnitSystem)