ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E

COMPUTER ORGANIZATION PROJECT 1 REPORT

CRN	:	21334 / 21336
LECTURER	:	Gökhan İnce / M. Ersel Kamaşak

GROUP MEMBERS:

150230736	:	Umut Özil (Group Representive)
150230737	:	Ege Keklikçi

SPRING 2024

Contents

1 INTRODUCTION [10 points]

In this project registers, register files and Arithmetic Logic Unit are designed and implemented in Verilog Hardware Description Language using Vivado. After the design and implementation, all of the created components are integrated to work under the same single clock. Using this implementation arithmetic and logical operations such as addition, subtraction, logical shift etc. can be done.

1.1 Task Distribution

Ege Keklikçi mainly worked on Part 1 and Part 2, while Umut Özil focused on Part 3 and Part 4. However, we did not strictly assign tasks to each other. Instead, we worked together to get a better idea about the overall project and help each other out whenever needed. Also, we wrote this report together.

2 MATERIALS AND METHODS [40 points]

2.1 PART 1 - Register

In this part, we implemented 16-bit general purpose register to use it in Register File and Address Register File later in part 2. This general purpose register is designed to store data and have 8 different functionalities with 4 wire inputs and 1 reg output.

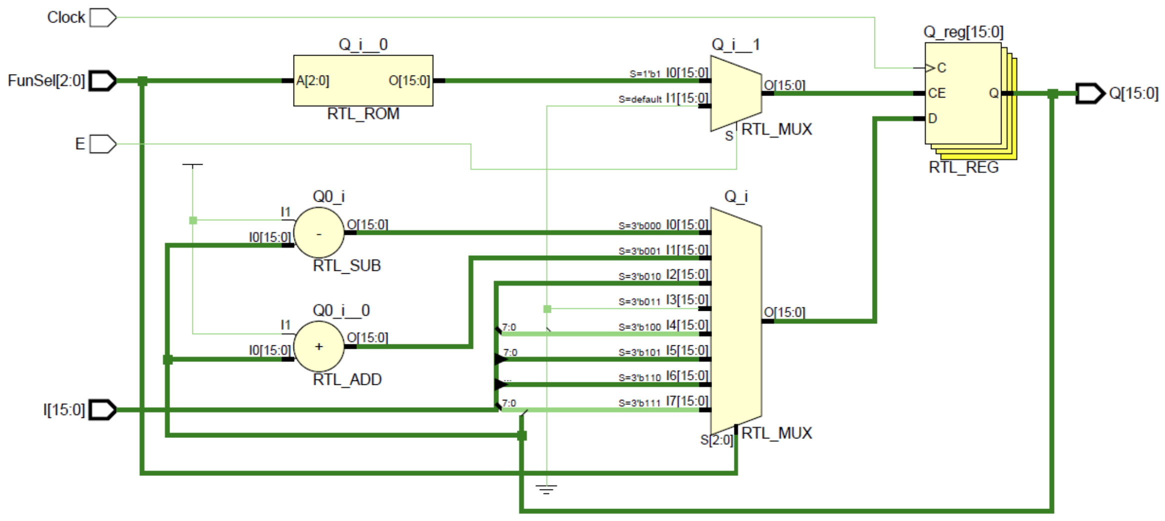


Figure 1: Schematic of Register

Inputs

- E: 1-bit enable input to decide the selected operation will be applied or not.
- FunSel: 3-bit function selection input to decide which function will applied.
- I: 16-bit input to load the register.
- clk: Clock signal to perform operations one by one on rising edge.

Outputs

- Q: 16-bit data stored in the register.

E	**FunSel**	$\mathbf{Q^+}$
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] $\leftarrow$ 0 (Clear MS 8-bit) Q[7-0] $\leftarrow$ I[7-0] (Write Low)
1	101	Q[7-0] $\leftarrow$ I[7-0] (Only Write Low)
1	110	Q[15-8] $\leftarrow$ I[7-0] (Only Write High)
1	111	Q[15-8] $\leftarrow$ I[7] (Sign Extend) Q[7-0] $\leftarrow$ I[7-0] (Write Low

Table 1: Register characteristic table

2.2 PART 2

In Part 2, we implemented Instruction Register, Register File and Address Register File.

2.2.1 Part-2a Instruction Register (IR)

In this part we implemented 16-bit IR that with 4 wire input and 1 reg output. Instruction registers are used to store the instruction that currently being executed by the CPU.

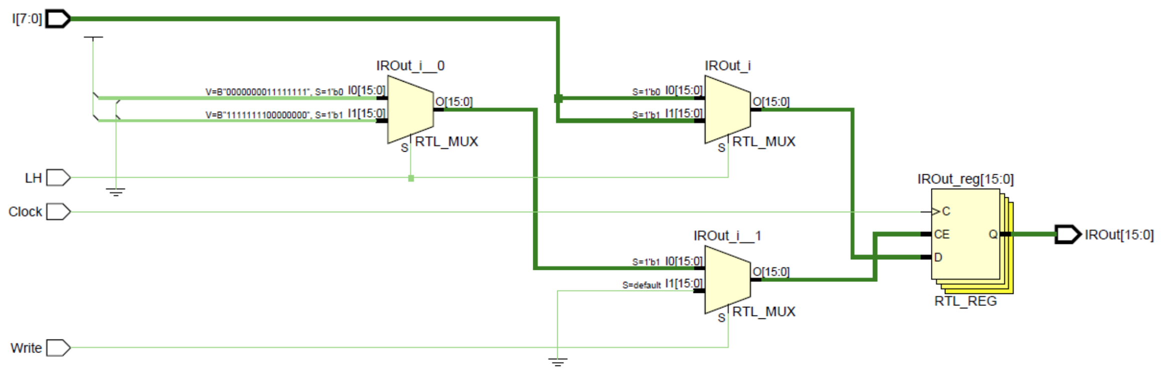


Figure 2: Schematic of Instruction Register

Inputs

- L'H: 1-bit Low/High flag to determine whether the data is written to LS 8-bit or MS 8-bit of IR
- Write: 1-bit write enable input to determine whether the register will be set or not.
- I: 8-bit input to load the register.
- clk: Clock signal to perform operations one by one on rising edge.

Outputs

- IROut: 16-bit data stored in the register.

L'H	Write	IR$^+$
Φ	0	IR (Retain Value)
0	1	IR[7-0] ← I (Load LS 8-bit)
1	1	IR[15-8] ← I (Load MS 8-bit)

Table 2: Instruction Register characteristic table

2.2.2 Part-2b Register File (RF)

In this part, we implemented Register File consist of four 16-bit general purpose registers R1, R2, R3, R4, and four 16-bit scratch registers S1, S2, S3, S4 with 7 wire inputs and 2 reg outputs. The RF is used to store the output values generated by the ALU and to supply operands to ALU.

The general purpose registers and scracth resgisters used in RF are the registers that we made in part 1.

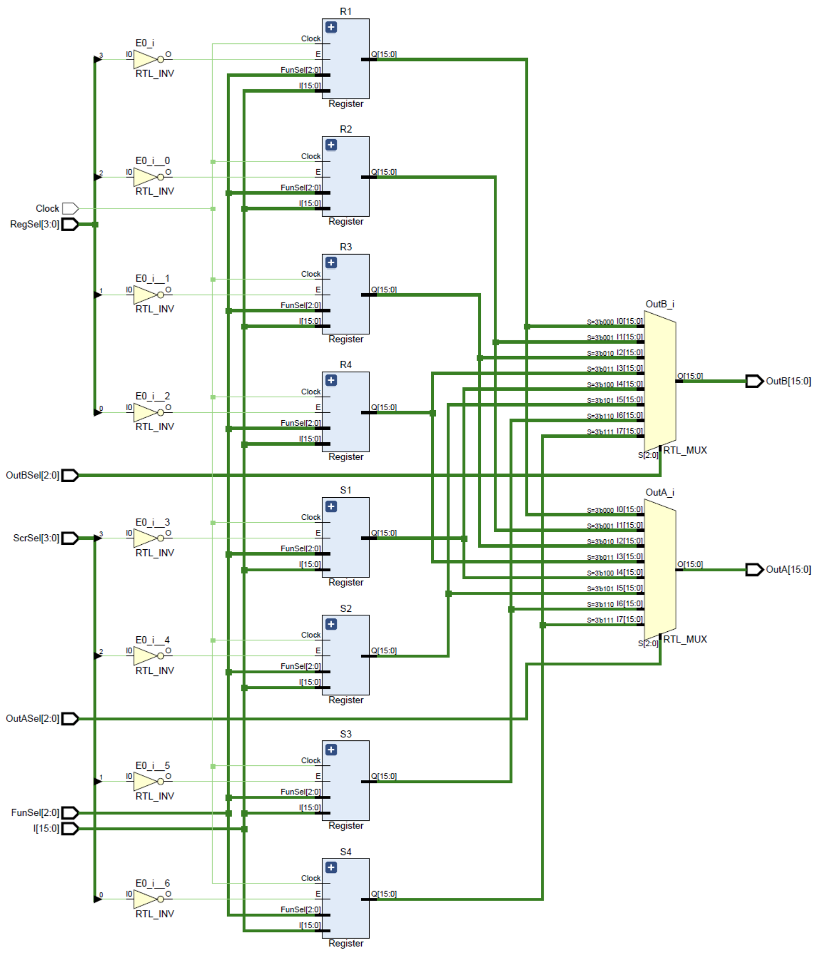


Figure 3: Schematic of Register File

Inputs

- RegSel: 4-bit register selection input to select which general purpose registers will be enabled.

RegSel	**Enable General Registers**	**RegSel**	**Enable General Registers**
0000	R1, R2, R3, R4	1000	R2, R3, R4
0001	R1, R2, R3	1001	R2, R3
0010	R1, R2, R4	1010	R2, R4
0011	R1, R2	1011	R2
0100	R1, R3, R4	1100	R3, R4
0101	R1, R3	1101	R3
0110	R1, R4	1110	R4
0111	R1	1111	-

Table 3: RegSel control input

- ScrSel: 4-bit register selection input to select which scratch registers will be enabled.

ScrSel	**Enable Scratch Registers**	**ScrSel**	**Enable Scratch Registers**
0000	S1, S2, S3, S4	1000	S2, S3, S4
0001	S1, S2, S3	1001	S2, S3
0010	S1, S2, S4	1010	S2, S4
0011	S1, S2	1011	S2
0100	S1, S3, S4	1100	S3, S4
0101	S1, S3	1101	S3
0110	S1, S4	1110	S4
0111	S1	1111	-

Table 4: ScrSel control input

- FunSel: 3-bit function selection input to decide which function will applied to selected registers.
- OutASel: 3-bit register selection input to select which register output will be fed to OutA of RF.
- I: 16-bit input to load the selected registers.
- OutBSel: 3-bit register selection input to select which register output will be fed to OutB of RF.

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

Table 5: OutASel control input

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

Table 6: OutBSel control input

- clk: Clock signal to perform operations one by one on rising edge.

Outputs

- OutA: 16-bit data driven by OutASel.
- OutA: 16-bit data driven by OutBSel.

2.2.3 Part-2c Address Register File (ARF)

In this part, we implemented Address Register file consist of 3 16-bit address registers called program counter (PC), address register (AR), and stack pointer (SP) with 6 wire inputs and 2 reg outputs. PC stores the address of the instruction to be executed next, AR stores a memory address to fetch data from memory, and SP points the top of the stack in memory.

The address registers used in ARF are the registers that we made in part 1.

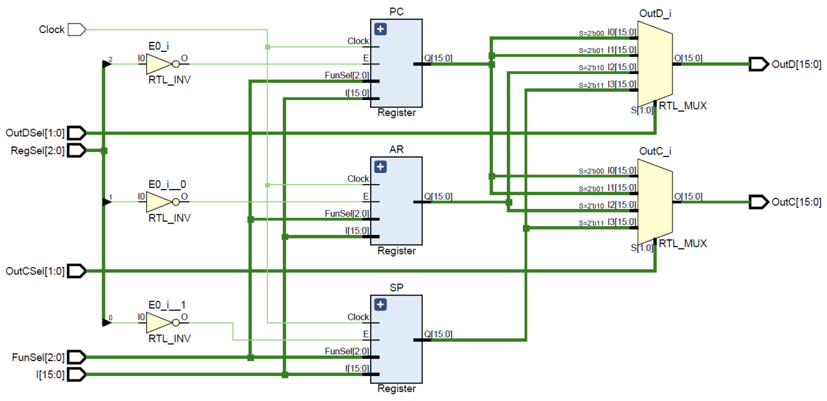


Figure 4: Schematic of Address Register File

Inputs

- FunSel: 3-bit function selection input to decide which function will applied to selected registers.
- I: 16-bit input to load the selected registers.
- OutCSel: 2-bit register selection input to select which register output will be fed to OutC of ARF.
- OutDSel: 2-bit register selection input to select which register output will be fed to OutD of ARF.

OutCSel	OutC
00	PC
01	PC
10	AR
11	SP

Table 7: OutCSel control input

OutDSel	OutD
00	PC
01	PC
10	AR
11	SP

Table 8: OutDSel control input

- RegSel: 3-bit register selection input to select which address registers will be enabled.

RegSel	**Enable Address Registers**
000	PC, AR, SP
001	PC, AR
010	PC, SP
011	PC
100	AR, SP
101	AR
110	SP
111	-

Table 9: RegSel control input

- clk: Clock signal to perform operations one by one on rising edge.

Outputs

- OutC: 16-bit data driven by OutCSel.
- OutD: 16-bit data driven by OutDSel.

2.3 PART 3 - Arithmetic Logic Unit (ALU)

In this part, we implemented 16-bit Arithmetic Logic Unit that can perform variues operations in both 8-bit and 16-bit with 5 wire input, 1 reg output and 4 reg flags (Z, C, N, O).

It can handle arithmetic operations such as addition and subtraction in 2's complement representation, arithmetic and logic shifts and logical operations such as AND, OR, XOR, NAND.

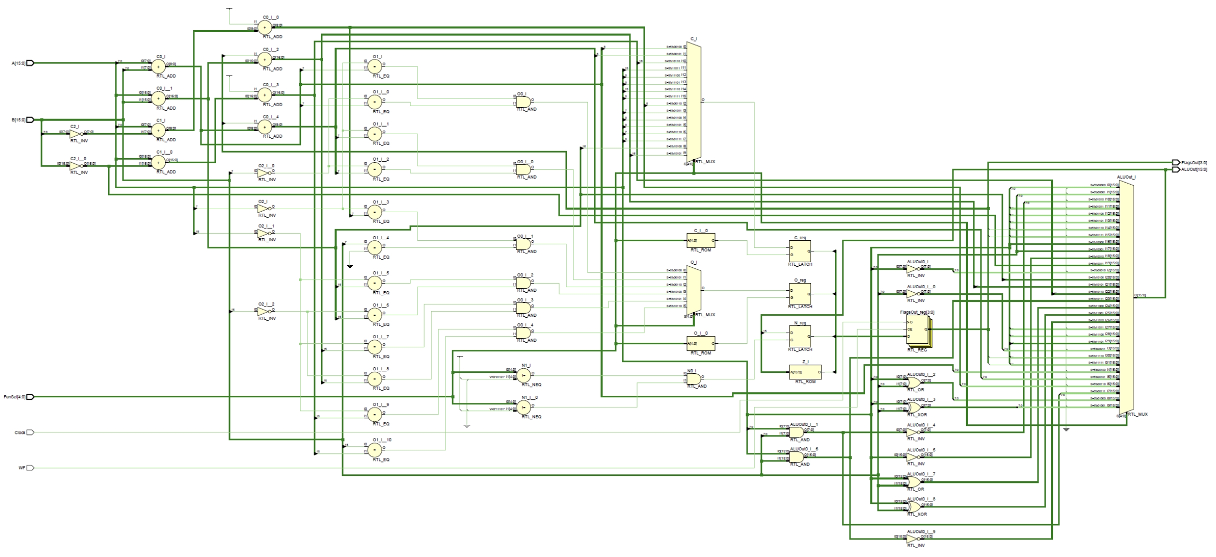


Figure 5: Schematic of Arithmetic Logic Unit

Inputs

- A: 16-bit operand A.
- B: 16-bit operand B.
- FunSel: 5-bit function selection input to decide which function will applied.
- WF: 1-bit flag write enable input to determine whether the flags will be set or not.
- clk: Clock signal to perform operations one by one on rising edge.

Outputs

- ALUOut: 16-bit output of the function applied.
- Flags:
  - Z: Indicates that whether the ALUOut is zero.
  - C: Indicates that whether a carry is occurred after the operation.
  - N: Indicates that whether the ALUOut is negatife.
  - O: Indicates that whether a overflow is occurred after the operation.

FunSel	**ALUOut**	**Z C N O**	**FunSel**	**ALUOut**	**Z C N O**
00000	A (8-bit)	$+ - + -$	10000	A	$+ - + -$
00001	B (8-bit)	$+ - + -$	10001	B	$+ - + -$
00010	NOT A (8-bit)	$+ - + -$	10010	NOT A	$+ - + -$
00011	NOT B (8-bit)	$+ - + -$	10011	NOT B	$+ - + -$
00100	A + B (8-bit)	$+ + + +$	10100	A + B	$+ + + +$
00101	A + B + Carry (8-bit)	$+ + + +$	10101	A + B + Carry	$+ + + +$
00110	A - B (8-bit)	$+ + + +$	10110	A - B	$+ + + +$
00111	A AND B (8-bit)	$+ - + -$	10111	A AND B	$+ - + -$
01000	A OR B (8-bit)	$+ - + -$	11000	A OR B	$+ - + -$
01001	A XOR B (8-bit)	$+ - + -$	11001	A XOR B	$+ - + -$
01010	A NAND B (8-bit)	$+ - + -$	11010	A NAND B	$+ - + -$
01011	LSL A (8-bit)	$+ + + -$	11011	LSL A	$+ + + -$
01100	LSR A (8-bit)	$+ + + -$	11100	LSR A	$+ + + -$
01101	ASR A (8-bit)	$+ + - -$	11101	ASR A	$+ + - -$
01110	CSL A (8-bit)	$+ + + -$	11110	CSL A	$+ + + -$
01111	CSR A (8-bit)	$+ + + -$	11111	CSR A	$+ + + -$

Table 10: Characteristic table of ALU

2.4 Part 4 - Arithmetic Logic Unit System

In this part, we implemented the Arithmetic Logic Unit System by making connections between the ALU, Address Register File (ARF), Register File (RF) and Instruction Register (IR) modules that we implemented in the previous parts and the memory module that provided to us in the project files with 3 multiplexers.

ALU System can perform arithmetic and logic operations with the data in the memory or registers and store the results of these operations in the registers or memory.

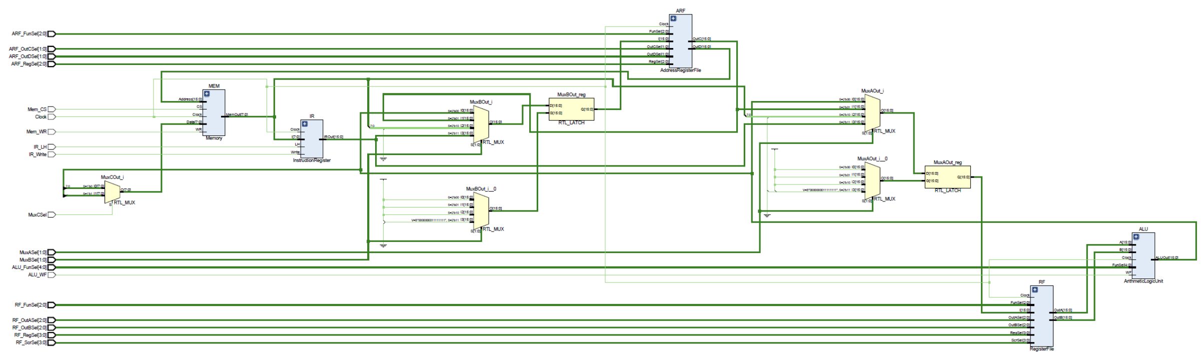


Figure 6: Schematic of Arithmetic Logic Unit System

Inputs

- MuxASel: 2-bit selection input to select which interconnection will be connected to RF.

- MuxBSel: 2-bit selection input to select which interconnection will be connected to ARF.

MuxASel	MuxAOut
00	ALUOut
01	ARF OutC
10	Memory Output
11	IR[7-0]

Table 11: Multiplexer-A control input

MuxBSel	MuxBOut
00	ALUOut
01	ARF OutC
10	Memory Output
11	IR[7-0]

Table 12: Multiplexer-B control input

- MuxCSel: 2-bit selection input to select which interconnection will be connected to Memory.

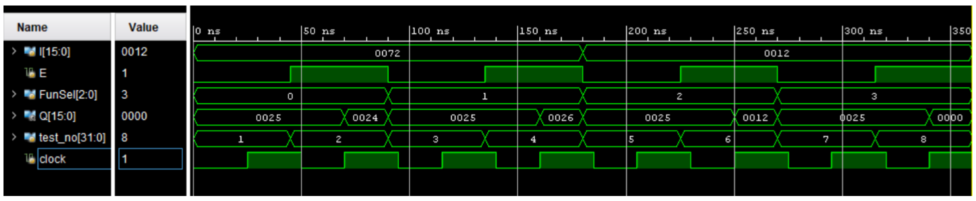
MuxCSel	**MuxCOut**
0	ALUOut[7-0]
1	ALUOut[15-8]

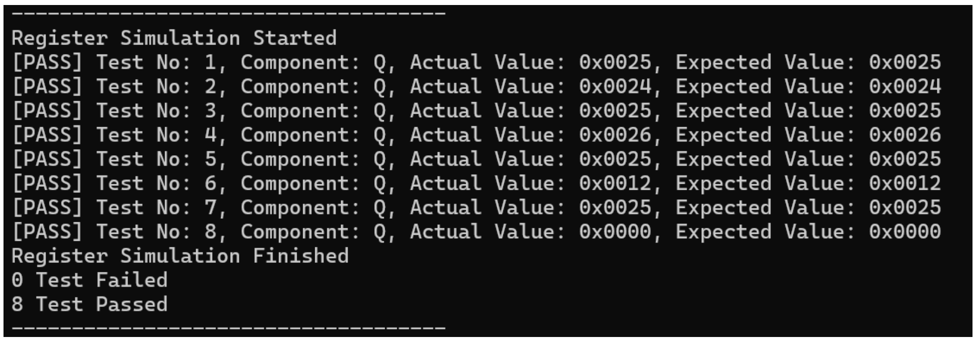
Table 13: Multiplexer-C control input

3 RESULTS [15 points]

3.1 PART 1 - Register Simulation

Results of the cases in the simulation file RegisterSimulation.v that provided to us in the project files.





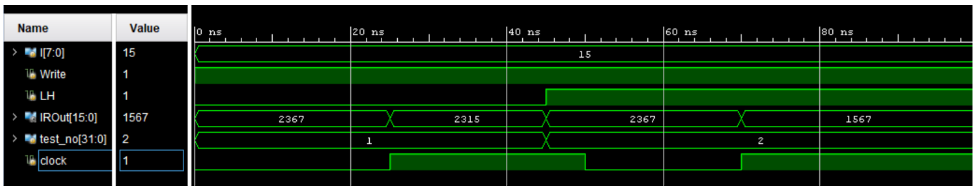
```
------------------------------------
Register Simulation Started
[PASS] Test No: 1, Component: Q, Actual Value: 0x0025, Expected Value: 0x0025
[PASS] Test No: 2, Component: Q, Actual Value: 0x0024, Expected Value: 0x0024
[PASS] Test No: 3, Component: Q, Actual Value: 0x0025, Expected Value: 0x0025
[PASS] Test No: 4, Component: Q, Actual Value: 0x0026, Expected Value: 0x0026
[PASS] Test No: 5, Component: Q, Actual Value: 0x0025, Expected Value: 0x0025
[PASS] Test No: 6, Component: Q, Actual Value: 0x0012, Expected Value: 0x0012
[PASS] Test No: 7, Component: Q, Actual Value: 0x0025, Expected Value: 0x0025
[PASS] Test No: 8, Component: Q, Actual Value: 0x0000, Expected Value: 0x0000
Register Simulation Finished
0 Test Failed
8 Test Passed
------------------------------------
```

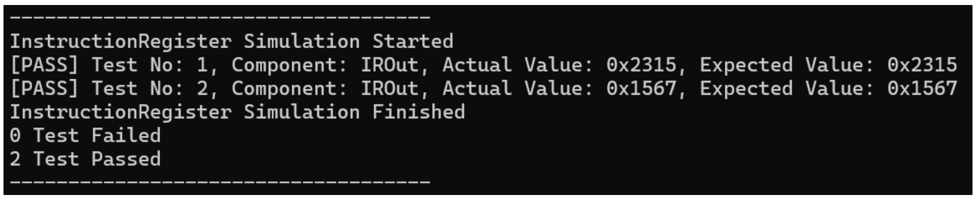
Figure 7: Results of simulation

3.2 PART 2

3.2.1 Part-2a Instruction Register Simulation

Results of the cases in the simulation file InstructionRegisterSimulation.v that provided to us in the project files.



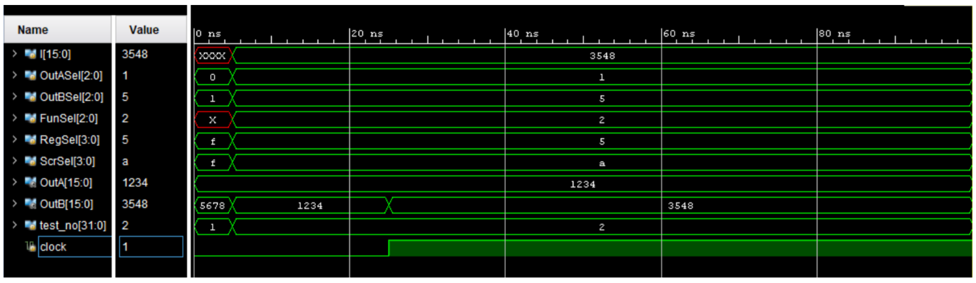


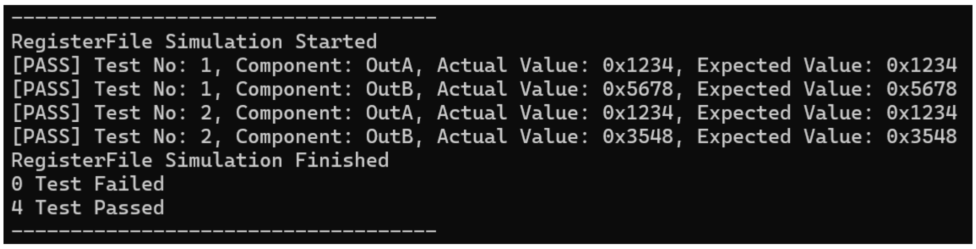
```
------------------------------------
InstructionRegister Simulation Started
[PASS] Test No: 1, Component: IROut, Actual Value: 0x2315, Expected Value: 0x2315
[PASS] Test No: 2, Component: IROut, Actual Value: 0x1567, Expected Value: 0x1567
InstructionRegister Simulation Finished
0 Test Failed
2 Test Passed
------------------------------------
```

Figure 8: Results of simulation

3.2.2 Part-2b Register File Simulation

Results of the cases in the simulation file RegisterFileSimulation.v that provided to us in the project files.



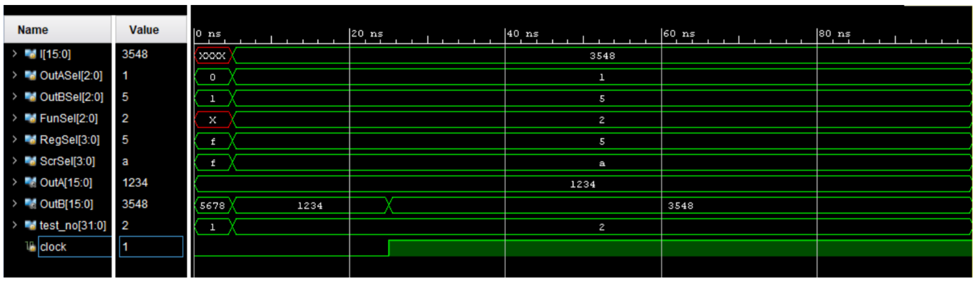


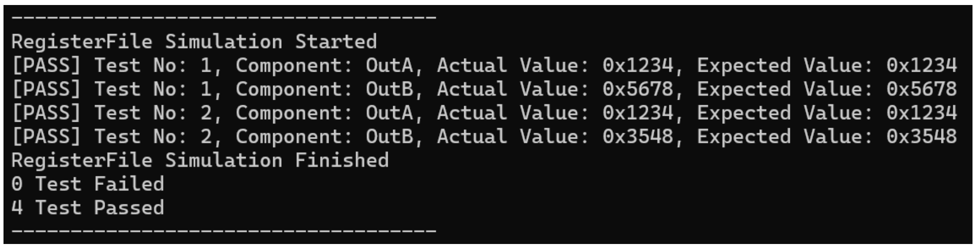
```
------------------------------------
RegisterFile Simulation Started
[PASS] Test No: 1, Component: OutA, Actual Value: 0x1234, Expected Value: 0x1234
[PASS] Test No: 1, Component: OutB, Actual Value: 0x5678, Expected Value: 0x5678
[PASS] Test No: 2, Component: OutA, Actual Value: 0x1234, Expected Value: 0x1234
[PASS] Test No: 2, Component: OutB, Actual Value: 0x3548, Expected Value: 0x3548
RegisterFile Simulation Finished
0 Test Failed
4 Test Passed
------------------------------------
```

Figure 9: Results of simulation

3.2.3 Part-2c Address Register File Simulation

Results of the cases in the simulation file AddressRegisterFileSimulation.v that provided to us in the project files.



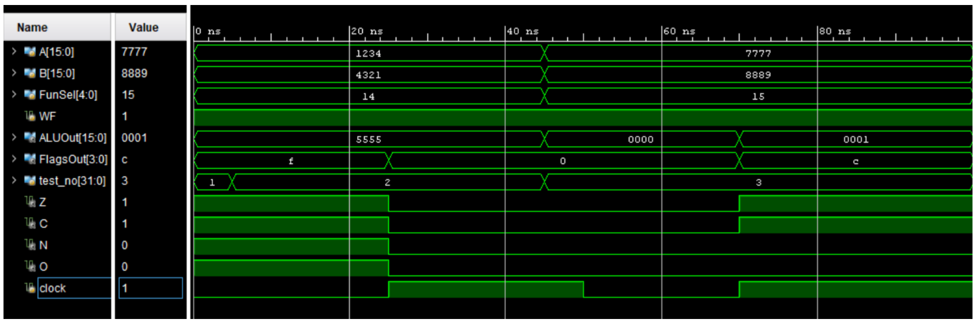


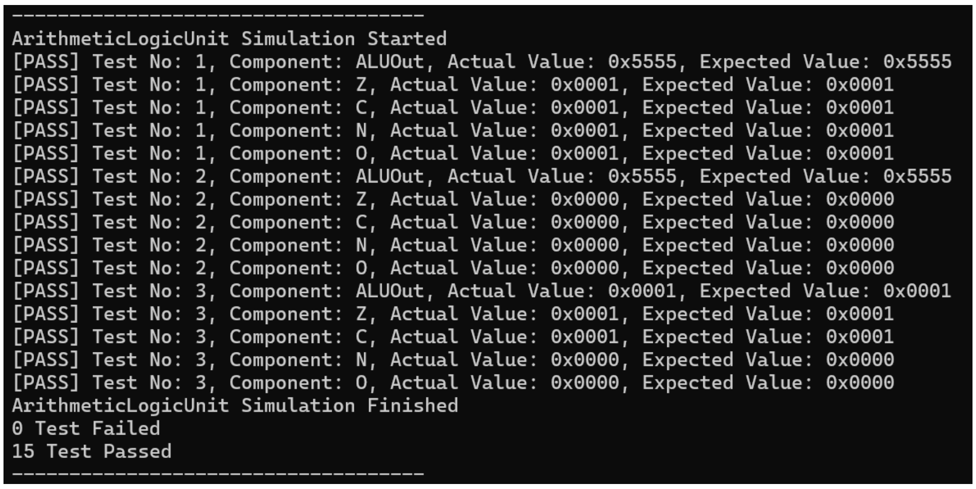
```
------------------------------------
RegisterFile Simulation Started
[PASS] Test No: 1, Component: OutA, Actual Value: 0x1234, Expected Value: 0x1234
[PASS] Test No: 1, Component: OutB, Actual Value: 0x5678, Expected Value: 0x5678
[PASS] Test No: 2, Component: OutA, Actual Value: 0x1234, Expected Value: 0x1234
[PASS] Test No: 2, Component: OutB, Actual Value: 0x3548, Expected Value: 0x3548
RegisterFile Simulation Finished
0 Test Failed
4 Test Passed
------------------------------------
```

Figure 10: Results of simulation

3.2.4 PART 3 - Arithmetic Logic Unit Simulation

Results of the cases in the simulation file ArithmeticLogicUnitSimulation.v that provided to us in the project files.



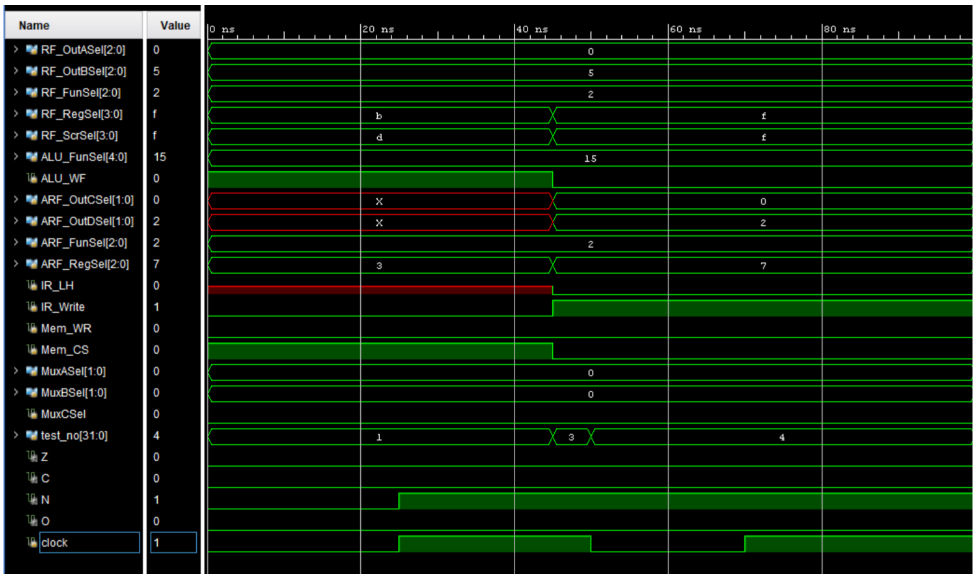


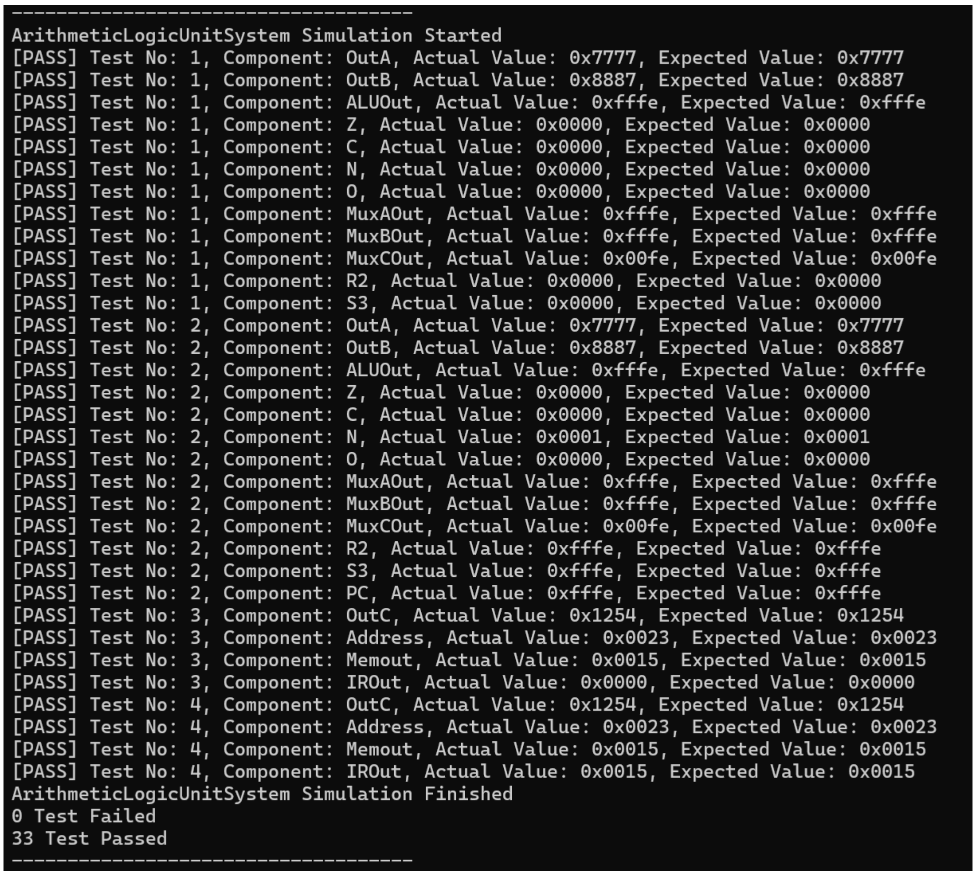
```
-----------------------------------
ArithmeticLogicUnit Simulation Started
[PASS] Test No: 1, Component: ALUOut, Actual Value: 0x5555, Expected Value: 0x5555
[PASS] Test No: 1, Component: Z, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 1, Component: C, Actual Value: 0x0001, Expected 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 11: Results of simulation

3.2.5 PART 4 - Arithmetic Logic Unit System Simulation

Results of the cases in the simulation file ArithmeticLogicUnitSystemSimulation.v that provided to us in the project files.





```
------------------------------------
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
ArithmeticLogicUnitSystem Simulation Finished
0 Test Failed
33 Test Passed
------------------------------------
```

Figure 12: Results of simulation

4 DISCUSSION [25 points]

4.1 PART 1

In this part we implemented 16-bit general purpose register to utilize in the next parts. We used always block utilized with posedge timing specifier, and inside the always block we used case block to apply different functions according to FunSel input.

4.2 PART 2

4.2.1 Part-2a

In this part we implemented 16-bit instruction register (IR). IR functioning similar to general purpose register that we made in part 1. As a result, we adapted the code we wrote for the general purpose register slightly for use in RF.

4.2.2 Part-2b

In this part, we implemented a register file (RF) containing a total of 8 registers. In this module we used always block utilized with * wildcard, because of register file works similar to multiplexer, it determines which registers are enabled and delivers their outputs to the designated output. Notably, it does not need to wait the rising edge of the clock signal.

4.2.3 Part-2c

In this part we implemented address register file (ARF) containing a total of 3 registers. ARF functioning similar to RF that we made in part 2b. As a result, we adapted the code we wrote for the RF slightly for use in ARF.

4.3 PART 3

In this part we implemented Arithmetic Logic Unit (ALU) that can perform both 8-bit and 16-bit arithmetic and logic operations. In this module we used two separate always blocks, one of them utilized with * wildcard and the other one utilized with posedge timing specifier.

In the first always block, utilized with * wildcard, logical and arithmetic operations are applied using case block, and also the flags to be set as a result of the operations are stored in internal wires.

In the second always block, utilized with posedge timing specifier, FlagsOut are set with the values stored in internal wires.

The reason why we use two different always blocks in this way is that, according to the expected values of the test cases in the ArithmeticLogicUnitSimulation.v file, the results of the arithmetic and logic operations are must performed instantly, but the flags to be set as a result of the operations must be done with the rising edge of the clock.

4.4 PART 4

In this part, we implemented an Arithmetic Logic Unit System that can fetch operands from registers or memory, perform arithmetic and logic operations on them, and store the results in registers or memory. In this module we used always block utilized * wildcard, because in the ALU System we just made some interconnections between modules using multiplexers. So, there is no need to wait the rising edge of the clock signal.

5 CONCLUSION [10 points]

We completed the project without any problems in general and successfully passed all the test cases provided to us. We only encountered a few difficulties. One of the challenges we faced was determining where to use the * wildcard and where to use the posedge timing specifier in always blocks. This issue confused us a bit.

We encountered this problem especially in ALU mpdule. While our ALUOut result was correct, the FlagsOut results were coming out incorrectly. After thoroughly examining the simulation file, we realized that the flags were updated with the clock signal. Then, instead of writing the flags we obtained from the operations directly to FlagsOut, we stored them in a internal wires and wrote them to FlagsOut in the second always block with the posedge timing specifier, and we overcame this problem.

In conclusion, it was a hard but an entertaining project that gained us a lot of experience and knowledge about Hardware Description Language Verilog and design of hardware systems.