ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E

COMPUTER ORGANIZATION
PROJECT REPORT

PROJECT NO : 1

DUE DATE : 12.04.2023

GROUP NO : 51

GROUP MEMBERS:

150200917 : Erblina Nivokazi (Group Representative)

150220760 : Onur Yavri

150200919 : Abdullah Jafar Mansour Shamout

1 INTRODUCTION

In this project, using Verilog hardware description language, 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 into a memory or registers and make these operations sequentially. With the given test inputs, the system hopefully works.

1.1 Task Distribution

Part 1 and 2a were done by Abdullah Jafar Mansour Shamout, part 2b and 2c were done by Erblina Nivokazi, part 3 and 4 were done by Onur Yavri and the throughout debugging was done as whole group

2 PROJECT PARTS

2.1 PART 1

For the first part we designed and implemented a n-bit general register that is utilized later on in the implementation of the Register File and the Address Register File (ARF). The generated n-bit register has 4 functions when enabled; clear, load, decrement, and increment. To design this module we used the keyword "parameter" with behavioral verilog to define the behavior of the register. The elaborated design for our implementation can be seen below:

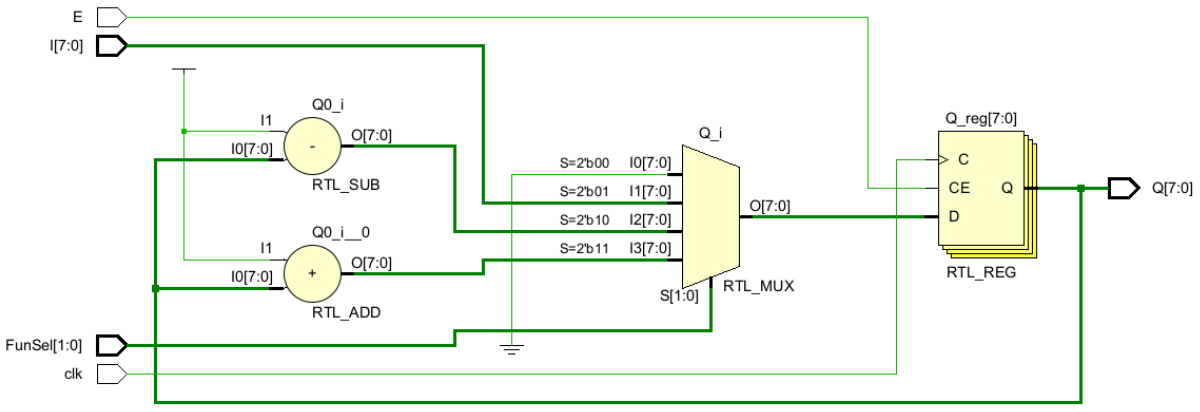

Figure 1: schematic for general registers

2.2 PART 2

2.2.1 PART-2a

For this part we designed and implemented a 16-bit Instruction Register (IR).The IR, just like the n-bit register, has 4 functions when enabled; clear, load, decrement, and increment, however the IR has a 16 bit storage capacity but only 8 bits of input. That is why we have another input to the IR that states which part of it will be effected by the operation to be carried out (L/H). To design this module we used the behavioral verilog to define the behavior of the register, we also utilized an if statement to check whether our operation will affect the low part or the high part. 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. The elaborated design for our implementation can be seen below:

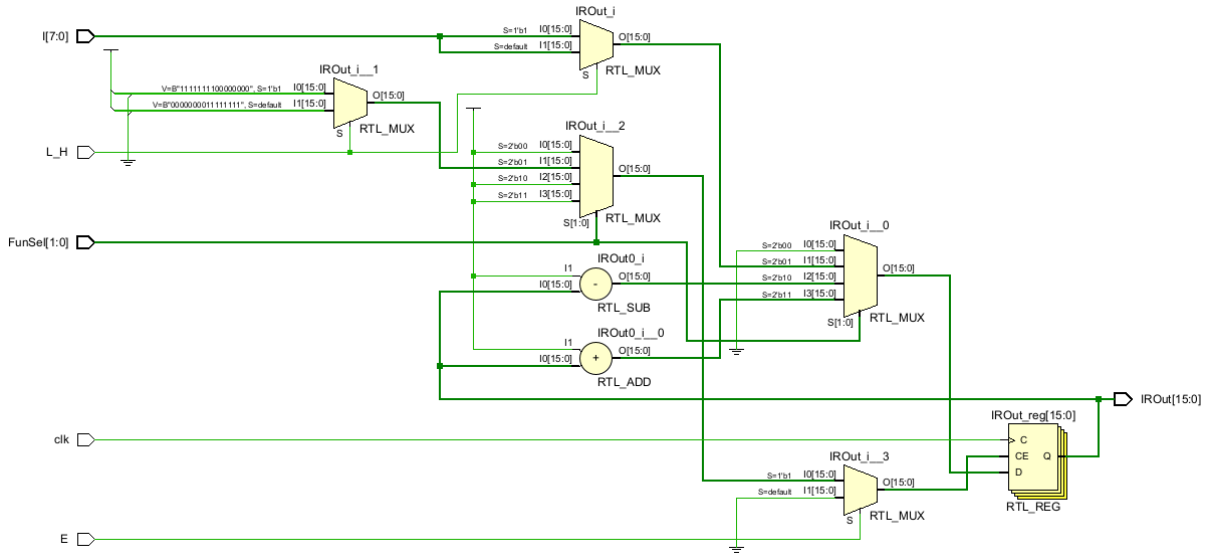

Figure 2: schematic for Instruction register (IR)

2.2.2 PART-2b

For this part we designed and implemented a Register File made of four general purpose registers and four temporary registers. In this design there are a total of seven inputs; one load, two register selectors, two output selectors, a function selector, and the clock. To initialize the registers we used the module created in the first part, the n_bit_register. The load, the function selector (FunSel) and the clock serve as input to each register. The selectors Rsel and Tsel determine which register gets enabled. Through the table given to us in the project file we could see a pattern of how these registers got enabled while the selector's value changed, each bit in the selector served as an enabler for a specific

register. Due to this we assigned each bit of Rsel and Tsel to a distinct wire, these wires then served as an enable input to each register. After a register gets enabled its' load gets modified or changed by a function selected by FunSel. This design has two outputs, O1 and O2, for this part we used an always block to create two multiplexers that have the outputs of every register as inputs, and O1Sel and O2Sel as selectors. The elaborated design for our implementation can be seen below:

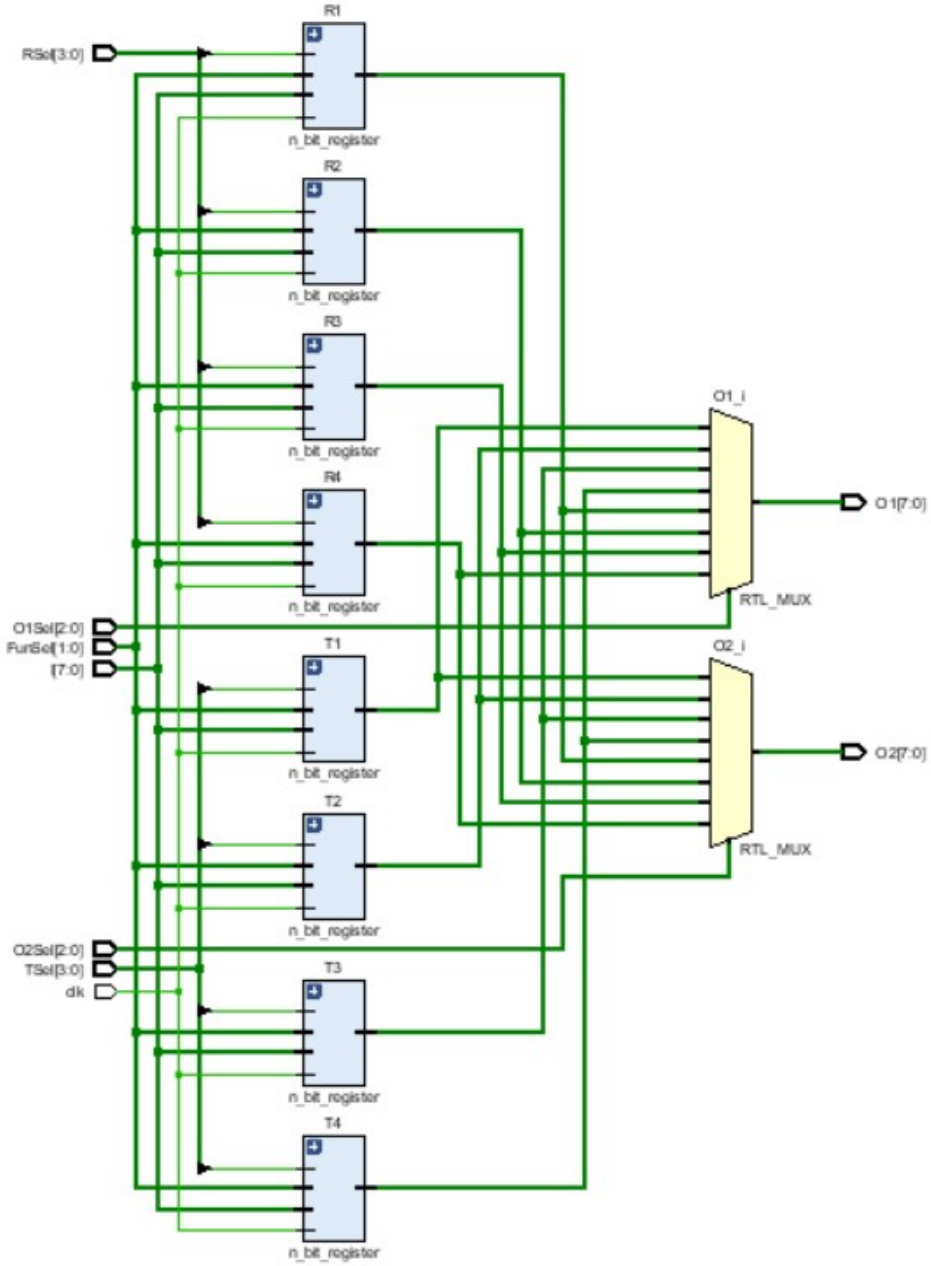

Figure 3: schematic for Register File

2.2.3 PART-2c

For this part we designed and implemented an Address Register File (ARF), made of four registers, PC, AR, SP, and PCPast. In this design we have a total of six inputs; one load, one register selector, two output selectors, a function selector, and the clock. To initialize the registers we used the n_bit_register module created in the first part. Just as in the Register file, the load, the function selector and the clock will serve as input to each register. As for the enable input in the register, we again used the same pattern like in the Register File and assigned each bit of the register selector Rsel to a distinct wire, which then serves as an enable input to each register. Each register, when enabled, gets modified by a function selected by FunSel. Since this design has two outputs we used an always block to create two multiplexers that have the outputs of all registers as input, and OutA and OutB as selectors. The elaborated design for our implementation can be seen below:

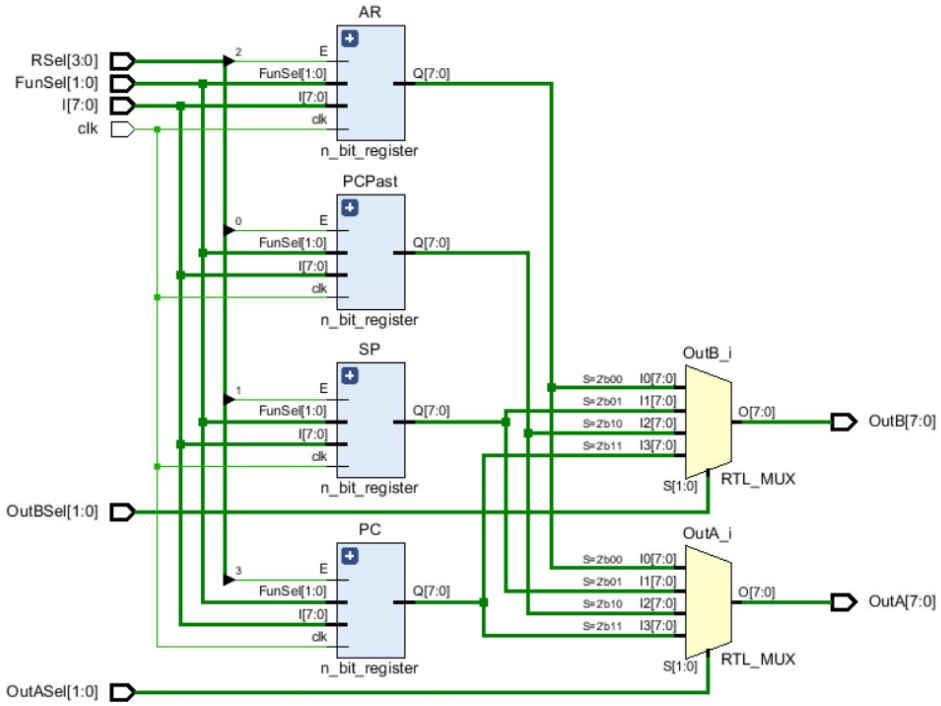

Figure 4: schematic for Address Register File

2.3 PART 3

For this part we designed and implemented the ALU. The ALU is a combinational circuit that does one of the microoperations specified to it by the FunSel input that it has. We implemented the ALU using 2's complement logic, we also created register flags that are controlled depending on the output of the ALU. Those flags describe what kind of output we received. To design the module we used switch logic (cases) to detect the type

of function, then according to which flags this function changes we would have an (if) statement control it. The elaborated design for our implementation can be seen below:

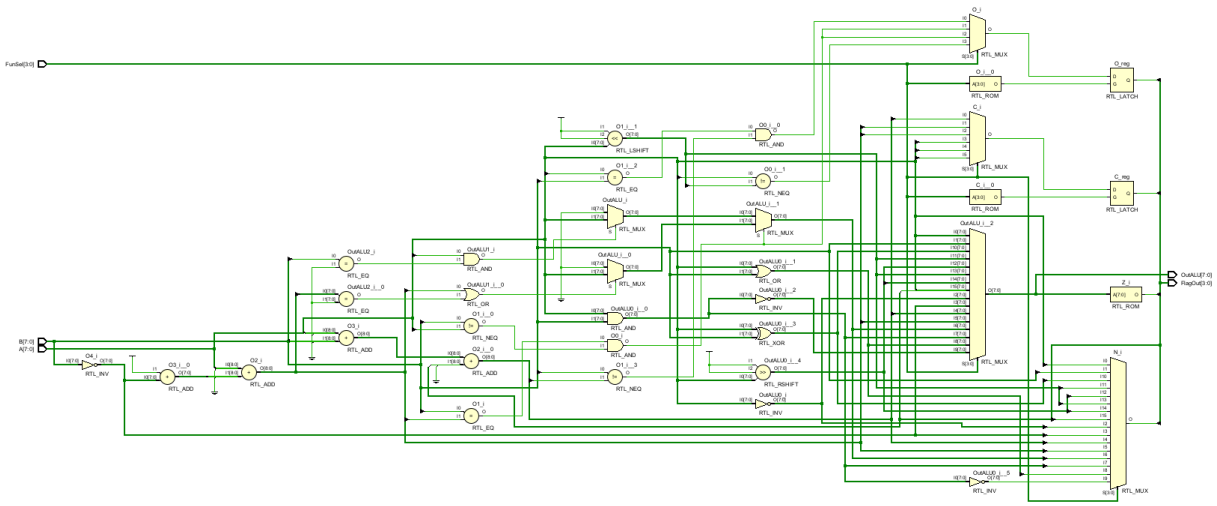

Figure 5: schematic for ALU

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. In Mano machine example we have seen in the lectures, IR also stores the encoded operations. The RF inside the system has 8 registers inside the circuitry for ALU to make consecutive operations faster. The Address Register File contains Program Counter (PC), which is to represent current address in progress, Address Register (AR), which points to an address, Stack Pointer (SP), which currently does not do anything, and Past Program Counter (PCPast), which exists in order to store the PC's last address in case of jump calls. Current diagram also has some control inputs (selectors) for choosing the operations that is 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 to play from the outside environment) it seems it is not completed, yet.

3 Simulations

In the process for the development of each module, we wrote some test files 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. Below are the results of our functional modules passing the tests we wrote:

3.1 PART 1

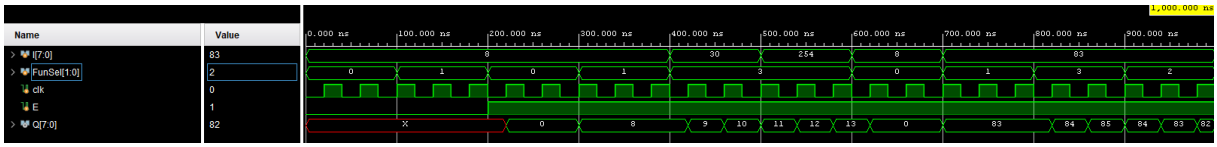

Figure 6: Simulation for part 1

3.2 PART-2a

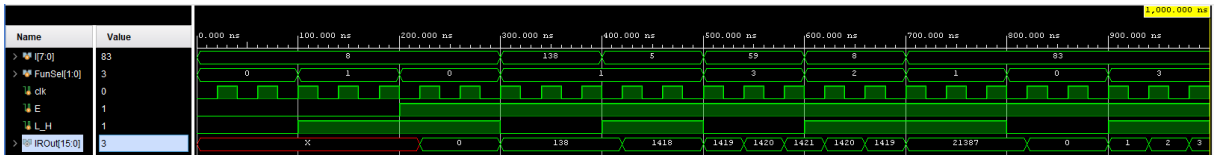

Figure 7: Simulation for part-2a

3.3 PART-2b

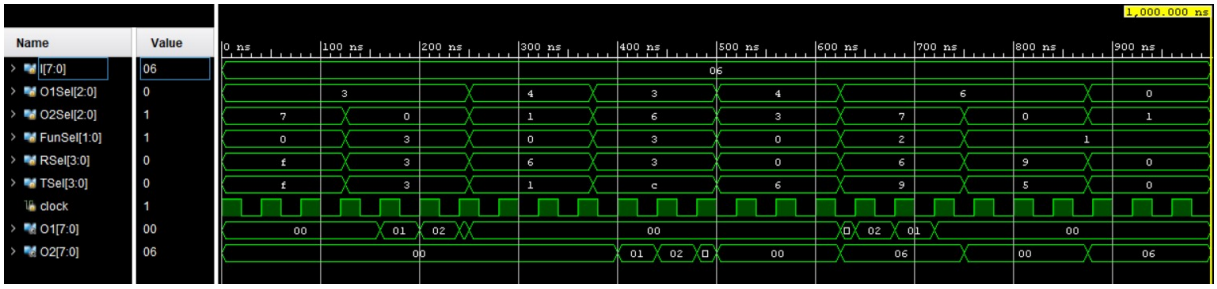

Figure 8: Simulation for part-2b

3.4 PART-2c

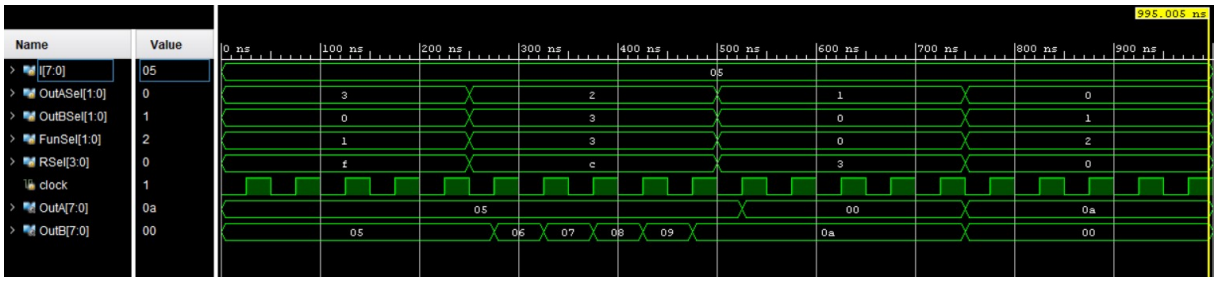

Figure 9: Simulation for part-2c

3.5 PART 3

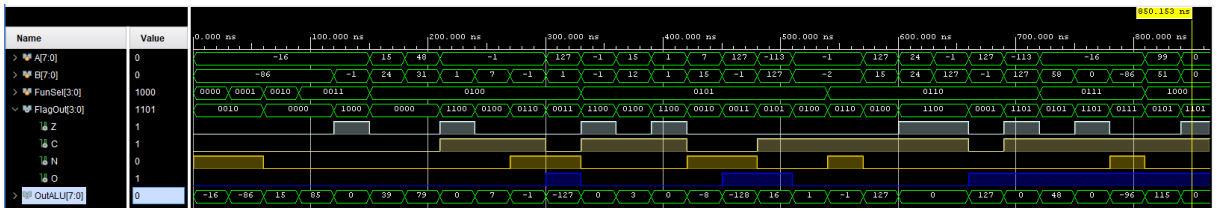

Figure 10: Simulation for part 3

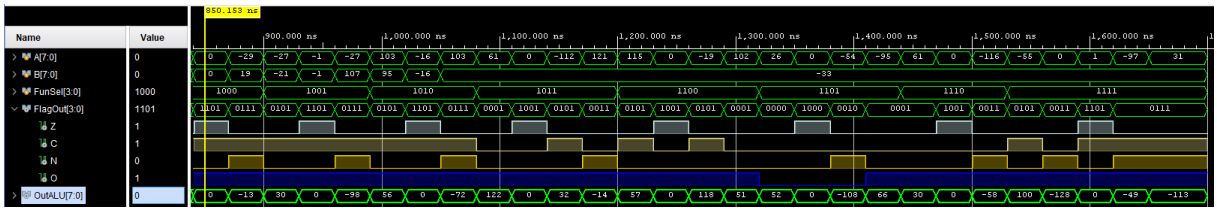

Figure 11: Simulation for part 3 continued

```
Input Values:
Operation: 0
Register File: O1Sel: 0, O2Sel: 0, FunSel: 0, RSel: 15, TSel: 15
ALU FunSel:  0
Addres Register File: OutASel: 0, OutBSel: 0, FunSel: 0, Regsel: 15
Instruction Register: LH: 1, Enable: 1, FunSel: 0
Memory: WR: 0, CS: 1
MuxASel: 3, MuxBSel: 3, MuxCSel: 1

Output Values:
Register File: AOut:   x, BOut:   x
ALUOut:   x, ALUOutFlag:  0, ALUOutFlags: Z:0, C:0, N:0, O:0,
Address Register File: AOut:   x, BOut (Address):   x
Memory Out:   z
Instruction Register: IROut:     x
MuxAOut:   x, MuxBOut:   x, MuxCOut:   x
```

Figure 12: Inputs and outputs for part 4

```
Input Values:
Operation: 1
Register File: O1Sel: 4, O2Sel: 5, FunSel: 1, RSel: 12, TSel:  0
ALU FunSel:  3
Addres Register File: OutASel: 3, OutBSel: 3, FunSel: 3, Regsel: 15
Instruction Register: LH: 0, Enable: 0, FunSel: 0
Memory: WR: 0, CS: 1
MuxASel: 3, MuxBSel: 1, MuxCSel: 1

Output Values:
Register File: AOut:   0, BOut:   0
ALUOut: 255, ALUOutFlag:  2, ALUOutFlags: Z:0, C:0, N:1, O:0,
Address Register File: AOut:   0, BOut (Address):   0
Memory Out:   z
Instruction Register: IROut:     0
MuxAOut:   0, MuxBOut:   z, MuxCOut:   0
```

Figure 13: Inputs and outputs for part 4, continued

```
Input Values:
Operation: 1
Register File: O1Sel: 4, O2Sel: 6, FunSel: 1, RSel: 10, TSel:  0
ALU FunSel: 12
Addres Register File: OutASel: 3, OutBSel: 3, FunSel: 3, Regsel: 15
Instruction Register: LH: 0, Enable: 0, FunSel: 0
Memory: WR: 0, CS: 1
MuxASel: 3, MuxBSel: 3, MuxCSel: 1

Output Values:
Register File: AOut:   1, BOut:   1
ALUOut:   0, ALUOutFlag: 12, ALUOutFlags: Z:1, C:1, N:0, O:0,
Address Register File: AOut:   1, BOut (Address):   1
Memory Out:   z
Instruction Register: IROut:     0
MuxAOut:   1, MuxBOut:   1, MuxCOut:   1
```

Figure 14: Inputs and outputs for part 4, continued

4 DISCUSSION

4.1 Part 1

For running the simulation for this part we used 10 cases as it can be seen in the table below:

Input in binary	Input in decimal	Enable signal	FunSel	Function	Output at Posedge	clk cycles
8'b00001000	8	0	2'b00	clear	XX	
8'b00001000	8	0	2'b01	load	XX	
8'b00001000	8	1	2'b00	clear	0	
8'b00001000	8	1	2'b01	load	8	
8'b00011110	30	1	2'b11	increment	9,10	2
8'b11111110	254	1	2'b11	increment	11,12,13	3
8'b00001000	8	1	2'b00	clear	0	
8'b01010011	83	1	2'b01	load	83	
8'b01010011	83	1	2'b11	increment	84,85	2
8'b01010011	83	1	2'b10	decrement	84,83,82	3

Figure 15: Simulation table for part 1

as the table shows, for the first two cases when the enable is 0, whatever the state that the register was in, will continue. Since the register had no previous values set, it is expected to receive XX. After that by clearing the register we are setting it to 0, as we can see this takes effect regardless of the input. Then we loaded the value 8 and for two clock cycles we incremented it to get 9 then 10. We did it again for 3 clock cycles to get 11,12 and 13, again we see input has no effect on the increment function. After that we cleared the register and loaded 83. We incremented it for two clock cycles to get 84 and 85. then we decremented it for 3 clock cycles to get 84, 83 and 82. As it can be seen per our test results, the general n-bit register does in fact behave as we expected it to. Since it is a n-bit register module, we had to give the parameter a value when calling it in the test file. We decided on giving it 8

4.2 Part-2a

For running the simulation for this part we used 10 cases again as it can be seen in the table below:

Input in binary	FunSel	Function	Enable signal	L/H	Output at Posedege	registers 16 bit in binary
8'b00001000	2'b00	clear	0	0	XX	
8'b00001000	2'b01	load	0	1	XX	
8'b00001000	2'b00	clear	1	1	0	00000000 00000000
8'b10001010	2'b01	load	1	0	138	00000000 10001010
8'b00000101	2'b01	load	1	1	1418	00000101 10001010
8'b00111011	2'b11	increment	1	0	1419	00000101 10001011
8'b00111011	2'b11	increment	1	0	1420	00000101 10001100
8'b00111011	2'b11	increment	1	0	1421	00000101 10001101
8'b00001000	2'b10	decrement	1	1	1420	00000101 10001100
8'b00001000	2'b10	decrement	1	1	1419	00000101 10001011
8'b01010011	2'b01	load	1	1	21387	01010011 10001011
8'b01010011	2'b00	clear	1	0	0	00000000 00000000
8'b01010011	2'b11	increment	1	1	1	00000000 00000001
8'b01010011	2'b11	increment	1	1	2	00000000 00000010
8'b01010011	2'b11	increment	1	1	3	00000000 00000011

Figure 16: Simulation table for part-2a

as the table shows, for the first two cases when the enable is 0, whatever the state that the register was in, will continue. Since the register had no previous values set, it is expected to receive XX. After that by clearing the register we are setting it to 0, as we can see this takes effect regardless of the input. Then we loaded the value 138 in binary to the low part of the register thus we get that value as an output. Afterwards we load the binary value 00000101 for the upper part of the register. By combining both the upper part and the lower part, we get a value of 1418 in decimal as an output. Then for three clock cycles we incremented the value in IR to get 1419, 1420, and 1421. Afterwards we decremented it for two clock cycles to get 1420 and 1419. Then by overwriting the High part of the IR we loaded a binary 01010011 this gave us the decimal value 21387. After that we decided to clear the IR and increment the IR for three clock cycles to get 1, 2, 3. Just like in the register part, we see that when the function is increment, decrement or clear it is independent of the input and the L/H input. Despite us using this table to show the output as decimal, we only used it like this to show how the incrementation and decrementation operations affect the value of the IR as a whole. However the IR utilizes its High and Low parts differently.

4.3 Part-2b

To test the Register File, we used eight different cases as it can be seen down below:

RSel	TSel	FunSel	R1	R2	R3	R4	T1	T2	T3	T4	O1Sel	O1	O2Sel	O2
1111	1111	00 – Clear	1	1	1	1	1	1	1	1	011	T4 – 0	111	R4 – 0
0011	0011	11 – Increment			1	1			1	1	011	T4 – incrementing	000	T1 – 0
0110	0001	00 – Clear		1	1					1	100	R1 – 0	001	T2 – 0
0011	1100	11 – Increment			1	1	1	1			011	T4 – 0	110	R3 – incrementing
0000	0110	00 – Clear						1	1		100	R1 – 0	011	T4 – 0
0110	1001	10 - Decrement		1	1		1			1	110	R3 – decrementing	111	R4 – incremented
1001	0101	01 – Load	1			1		1		1	110	R3 – decremented	000	T1 – decremented
0000	0000	01 – Load									000	T1 – decremented	001	T2 – load

Figure 17: Simulation table for part-2b

In each case we changed the inputs of RSel, TSel, FunSel, O1Sel, and O2Sel, whereas the load was kept the same throughout the testbench. In the first case both RSel and TSel are set to 1111, meaning that all registers are enabled to be modified by our selected function from FunSel. FunSel in the first case is set to 00, which clears all the registers to zero. In the second case RSel and TSel are set to 0011, enabling R3, R4 and T3, T4. The FunSel in this case is 11 meaning that the enabled registers will start to increment. O1Sel is 011 meaning that T4 will be output O1 (the output of T4 will show the number incrementing 01,02,03...), and O2Sel is 000 making T1 output O2. In the third case R3, R4, and T4 get cleared to zero. In the fourth case R3, R4, T1, and T2 are enabled to be incremented, the output O1 is going to be T4 which was set to zero the previous case, and O2 will output R3 which is getting incremented. In the fifth case we cleared T2 and T3. In the sixth case we decremented R2, R3, T1 and T4, and as output in O1 we have R3 which is getting decremented, and in O2 we have R4 which got incremented in the fourth case. In the seventh case R1, R4, T2 and T4 get the value of load, and as output in O1 there is R3 and in O2 there is T1, both of which got decremented in the previous case. In the last case none of the registers are enabled so nothing changes, as output in O1 we have T1, and as output in O2 we have T2 which is equal to the load of the system.

4.4 Part-2c

To test the Address Register File, we used four different cases as it can be seen down below:

RSel	FunSel	PC	AR	SP	PCPast	OutASel	OutA	OutBSel	OutB
1111	01 - Load	1	1	1	1	11	PC - load	00	AR – load
1100	11 – Increment	1	1			10	PCPrev - load	11	PC – load incrementing
0011	00 – clear			1	1	01	SP – 0	00	AR – incremented
0000	10 - Decrement					00	AR - incremented	01	SP - 0

Figure 18: Simulation table for part-2c

In each case we changed the inputs of RSel, FunSel, OutASel and OutBSel, whereas the load was kept the same throughout the testbench. In the first case RSel is 1111 which

means all registers are enabled, since the FunSel in the first case is 01, all registers get assigned the load. OutASel is set to 11, meaning it will output the PC which right now is holding the load, and OutBSel is set to 00 meaning it will output AR which is also carrying the load. In the second case only PC and AR are enabled, since the FunSel in the second case is 11, PC and AR get incremented. OutASel is set to 10 meaning it will output PCPrev in OutA and in Outb the output will be PC. In the third case SP and PCPast get cleared, in the output OutA we selected SP which got cleared to zero in this case, and in the OutB we have AR which got incremented in the second case. In the last case none of the registers are enabled, in OutA we have AR which got incremented in the second case and in OutB we have SP which got set to zero in the third case.

4.5 Part 3

ALU Basic Operations Test Cases

Input A	Input B	Function	Output	Flags (ZCNO)	Decimal (signed)
11110000	10101010	A	11110000	0010	-16
11110000	10101010	B	10101010	0010	-86
11110000	10101010	$\bar{A}$	00001111	0000	15
11110000	10101010	$\bar{B}$	01010101	0000	85
11110000	11111111	$\bar{B}$	00000000	1000	0

In the table above, basic operations of ALU has been represented and the final results are taken from implemented ALU itself. As some of the flags can be updated, the operations chosen to show possibilities of the change in flags.

ALU Arithmetic Operations Test Cases

Input A	Input B	Function	Output	Flags (ZCNO)	Decimal (signed)
00001111	00011000	$A + B$	00100111	0000	37
00110000	00011111	$A + B$	01001111	0000	79
11111111	00000001	$A + B$	00000000	1100	0
11111111	00000111	$A + B$	00000111	0100	7
11111111	11111111	$A + B$	11111111	0110	-1
01111111	00000001	$A + B$	10000001	0011	-127
11111111	11111111	$A - B$	00000000	1100	0
00001111	00001100	$A - B$	00000011	0100	3
00000001	00000001	$A - B$	00000000	1100	0
00000111	00001111	$A - B$	11111000	0010	-8
01111111	11111111	$A - B$	10000000	0011	-128
10001111	01111111	$A - B$	00010000	0101	16
11111111	11111110	$A - B$	00000001	0100	1
11111111	11111110	$A > B$	11111111	0110	-1
01111111	00001111	$A > B$	01111111	0100	127
00011000	00011000	$A > B$	00000000	1100	0
11111111	01111111	$A > B$	00000000	1100	0
01111111	11111111	$A > B$	00000000	1101	0
10001111	01111111	$A > B$	00000000	1101	0

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.

ALU Logical Operations Test Cases

Input A	Input B	Function	Output	Flags (ZCNO)	Decimal (signed)
11110000	00111010	A AND B	00110000	0100	48
11110000	00000000	A AND B	00000000	1100	0
11110000	10101010	A AND B	10100000	0110	-96
01100011	00110011	A OR B	01110011	0100	115
00000000	00000000	A OR B	00000000	1100	0
11100011	00010011	A OR B	11110011	0110	-13
11100101	11101011	A NAND B	00011110	0100	30
11111111	11111111	A NAND B	00000000	1100	0
11100101	01101011	A NAND B	10011110	0110	-98
01100111	01011111	A XOR B	00111000	0100	56
11110000	11110000	A XOR B	00000000	1100	0
01100111	11011111	A XOR B	10111000	0110	-72

Logical operations of implemented ALU works fine, even though it is not supposed to represent outputs as decimals, in order not to make the tables in the same shape, they are also included in the table's last column. Basic logical operations work properly and changes the flags correctly.

ALU Shifting Operations Test Cases

Input A	Input B	Function	Output	Flags (ZCNO)	Decimal (signed)
00111101	11011111	$LSL\,A$	01111010	0000	122
00000000	11011111	$LSL\,A$	00000000	1000	0
10010000	11011111	$LSL\,A$	00100000	0100	32
01111001	11011111	$LSL\,A$	11110010	0010	-14
01110011	11011111	$LSR\,A$	00111001	0100	57
00000000	11011111	$LSR\,A$	00000000	1000	0
11101101	11011111	$LSR\,A$	01110110	0100	118
01100110	11011111	$LSR\,A$	00110011	0010	51
00011010	11011111	$ASL\,A$	00110100	0000	52
00000000	11011111	$ASL\,A$	00000000	1000	0
11001010	11011111	$ASL\,A$	10010100	0010	-108
10100001	11011111	$ASL\,A$	01000010	0001	66
00111101	11011111	$ASR\,A$	00011110	0000	30
00000000	11011111	$ASR\,A$	00000000	1000	0
10001100	11011111	$ASR\,A$	11000110	0010	-58
11001001	11011111	$CSR\,A$	01100100	0110	100
00000000	11011111	$CSR\,A$	10000000	0010	-128
00000001	11011111	$CSR\,A$	00000000	1000	0
10011111	11011111	$CSR\,A$	01001111	0100	79
00011111	11011111	$CSR\,A$	10001111	0110	-113

Shifting operations of implemented ALU works properly but because the design is ambiguous, it might not the proper result. As some of the operations require carry flag to work, after an operation that is not related to current shifting operation and has changed the carry flag, shifting operation might not result as the user intended. This problem is also valid for arithmetic operations. As stated in the previous part, even though it is not meaningful for shifting operations (except arithmetic shifting operations) to be represented as decimals, they are included to preserve the shape of the table.

5 CONCLUSION

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 especially the comparison one because in the homework it was not specified how the comparison operation 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 understood how an ALU system behaves using 2's complement logic.