ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E
COMPUTER ORGANIZATION
PROJECT 1 REPORT

CRN : 21335 | 21336

LECTURER : Deniz Turgay Altılar | Mustafa Ersel Kamaşak

GROUP MEMBERS:

820220334 : İREM KALAY(Group Representative)

820220506 : FEYZA DEVELİOĞLU

SPRING 2024

Contents

1 INTRODUCTION [10 points]

In this project, we have implemented basic computer step by step using Verilog hardware description language and simulated each parts. In the first part, we implemented general usage 16 bit-register for using in the structure later on. In the second part, we implemented a structures called "Instruction Register", "Register File" and "Adress Register File". Then, in the third part of the project, we implemented the Arithmetic Logic Unit based on the given characteristic table and observed correct results with wanted operations. Finally at the last part, all the elements were connected to each other as an organization that works with same clock called Arithmetic Logic Unit System.

1.1 Task Distribution

We examined and discussed each part together at the start of every part. The 1(Register), 2a(IR),2b(RF),2c(ARF) is done by İrem. The part 3(ALU) done by Feyza, the control of it done by İrem and the last part 4 (ALU System) we did it together.

2 MATERIALS AND METHODS [40 points]

2.1 Part 1

In the first part, The 16-bit register is implemented by using Verilog. The 16-bit register inputs were: 3-bit control signal 'FunSel', enable input 'E', Load "I" and clock. With 3 bit function selection, the register is performing the selected operation and also it is synchronous with respect to the system clock and only updates its content on the rising edge of the clock due to selected function when the enable (E) signal is asserted.

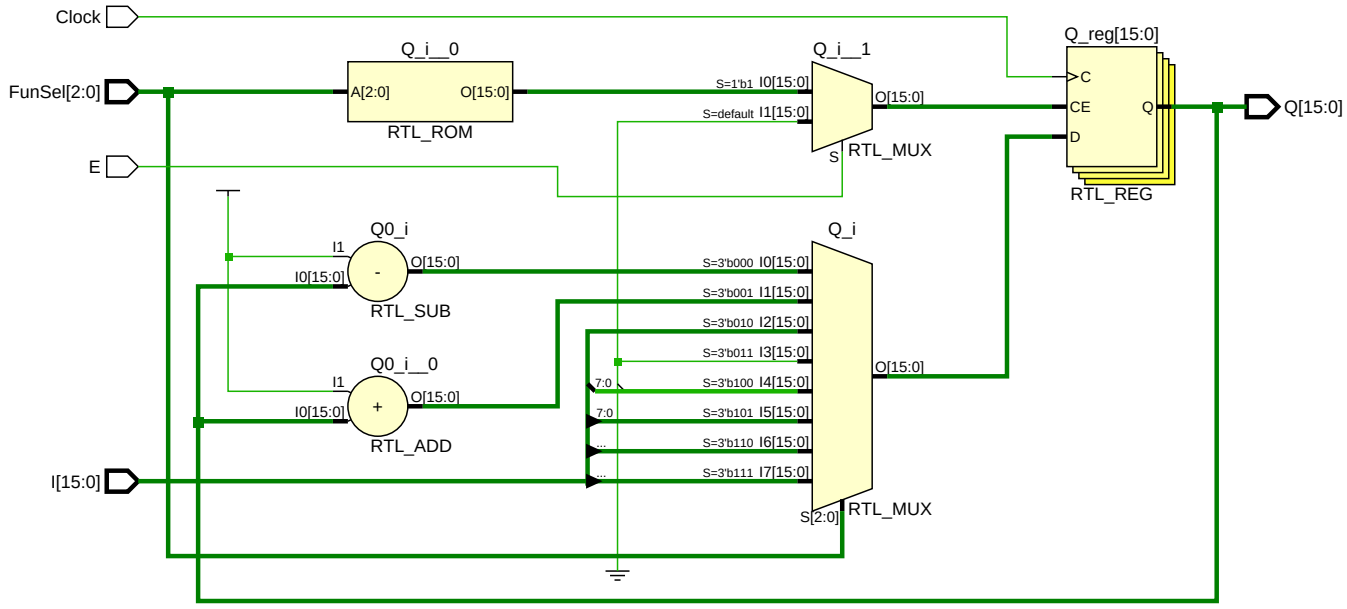

Figure 1: Register Schematic

2.2 Part 2

2.2.1 Part 2a)

For Part 2's a part of the project, we were wanted to design and implement 16-bit Instruction Register that can store 16 bit binary data. Its inputs are as follows: 8 bit Load(input), 1 bit LH(Low/High to decide where to write), 1 bit "write" and a "clock". It has 8 bit binary input but it can store 16 bit, that is the reason why by Low High (L/H) signals , the decision was given as either the 8 bit-data will be written to the bits of(Low bits) 7-0 or 15-8(High bits). There was also "write" input that controls when to write. If write signal is low, it does not writes into the register's anywhere. The Schematic of the IR register is as below.

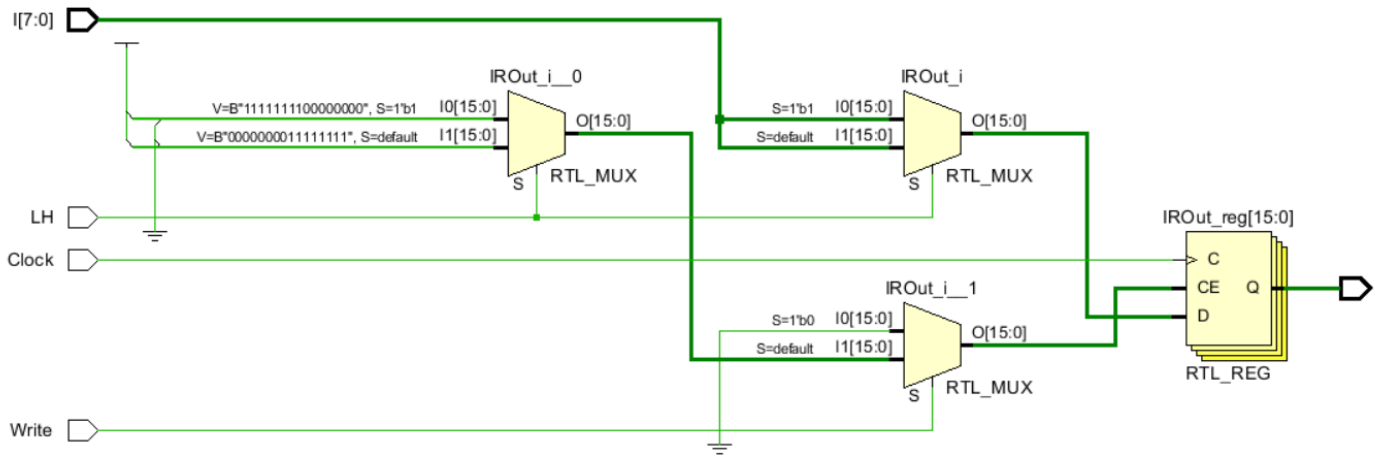

Figure 2: Instruction Register Schematic

2.2.2 Part 2b)

For b part of section 2, we were wanted to design and implement a Register File system.In this part we had inputs as load(the input), two output selectors, two register selectors, one function selector and a clock. Here , we created four 16-bit general purpose registers and four 16-bit scratch registers using our previous Register.v module so that all registers had already their own properaties declared before. We had the inputs 3-bit OutASel and OutBSel to get the OutA and OutB output registers. Also we had 4-bit RegSel and ScrSel as inputs to select the general purpose registers and scratch registers. We noticed a pattern while enabling the registers so that RegSel and ScrSel were able to enable which register needed according to the given table in the project. How the registers are enabled will be disscussed in disscussion part in more detailed. Then register's load is modified by selected function by FunSel after it's enabled.The schematic of our design can be seen below.

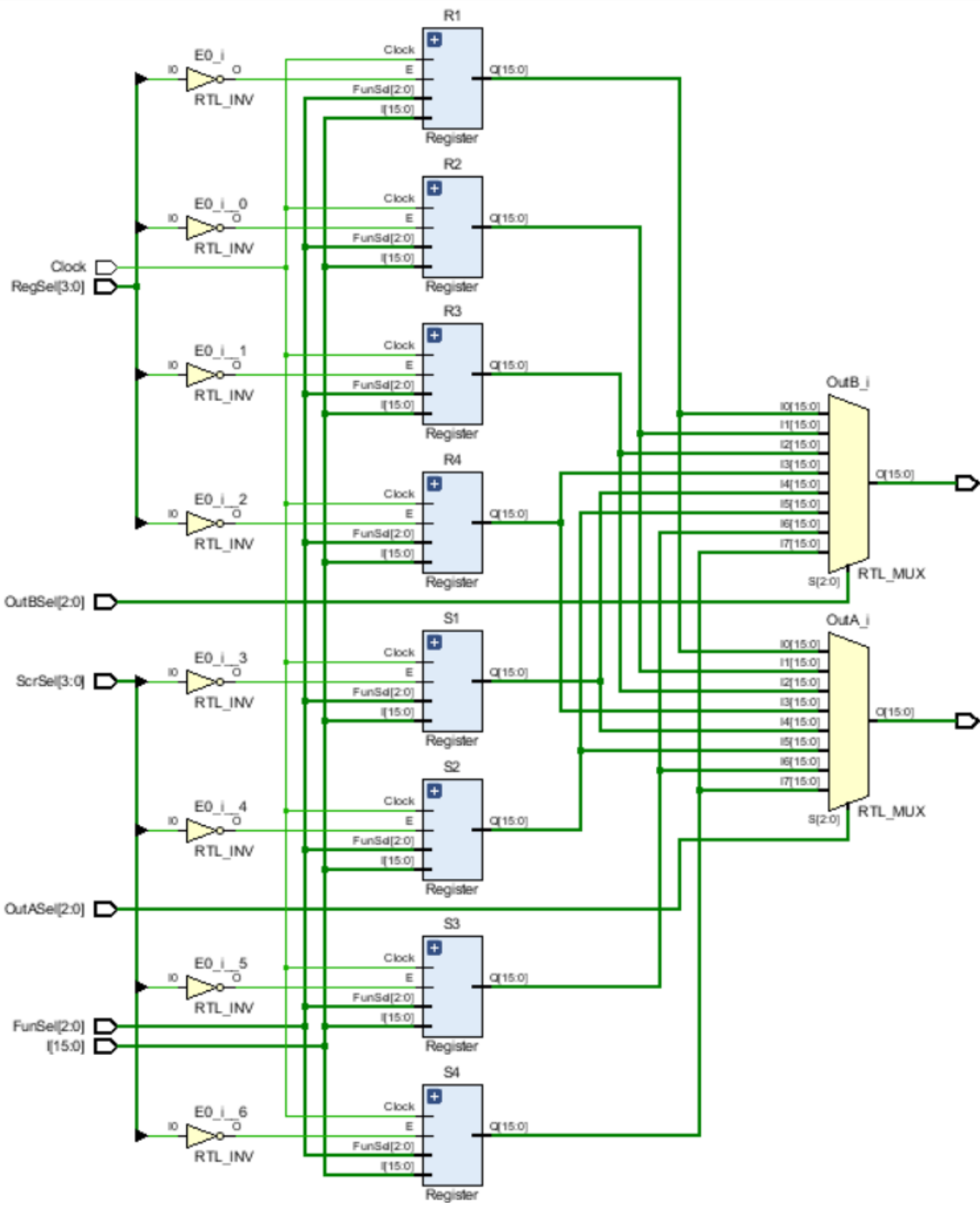

Figure 3: Register File

2.2.3 Part 2c)

In the c part of second section, we have designed an address register file (ARF) module which consists of three 16-bit address registers called program counter(PC), address register(AR) and the stack pointer(SP). Again we used the Register.v module to declare the 16-bit registers. As inputs we had Load, two output selections(2 bit OutCSel and OutDSel), one function selection(3-bit FunSel), one register selection (3 bit RegSel) and a clock. For enable input of registers, we used the pattern that we noticed in the given table like in the Register File part (Part 2b) and as in part 2b, registers are modified by FunSel when they are enabled. To get the Outputs as wanted in the table, we used an always block and inisde of it, the "case" statement to implement two multiplexers for OutC and OutD outputs as in the table. The schematic of our design is below.

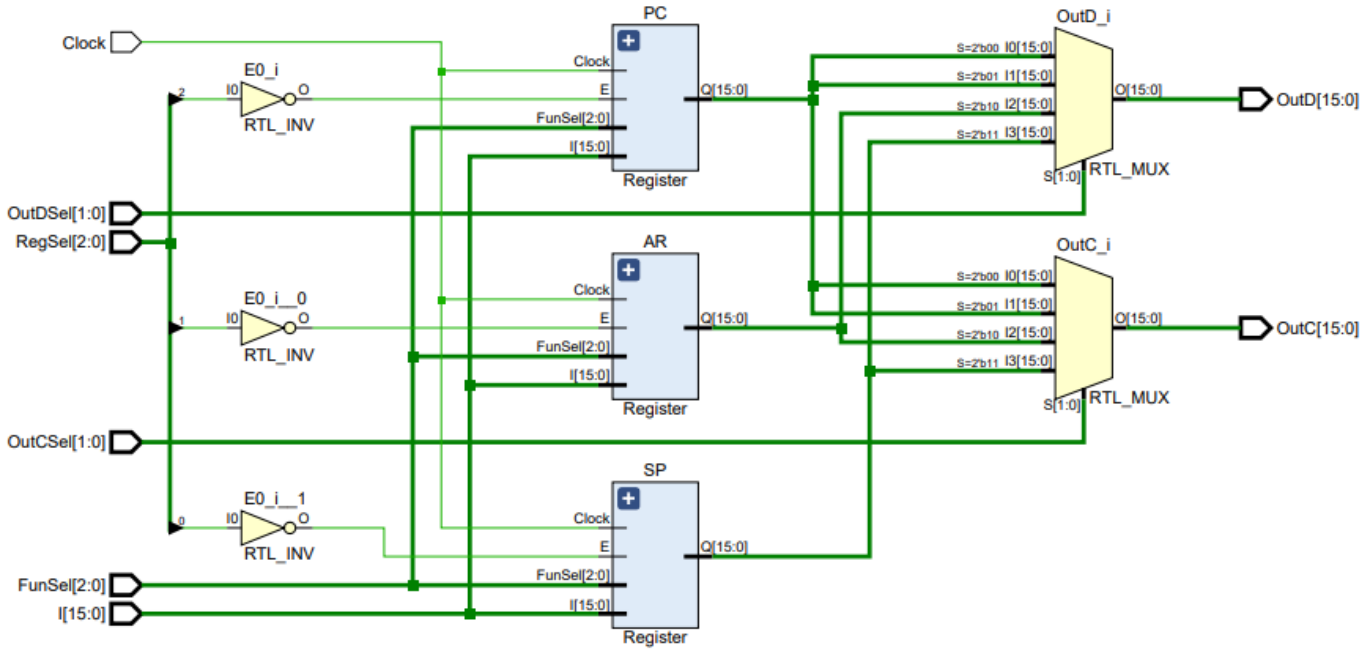

Figure 4: Address Register File

2.3 Part 3

In the part 3, we have implemented the wanted ALU due to the given tables. It had inputs as: A , B, function selector 5 bit FunSel and Flags. The structure that we implemented is able to perform both 8-bit and 16-bit operations. After performing the operation, it updates the flags when the clock comes (positive edge triggered). The flags were "Zero", "Carry", "Negative", "Over Flow".

The ALU that we implemented is able to calculate the following operations: Pass-through operations for A and B, Bitwise NOT operation on A and B, Additions and subtractions with and without carry, AND, OR, XOR, and NAND bitwise operations, Shift operations including logical shift left (LSL), logical shift right (LSR), arithmetic shift right (ASR), circular shift left (CSL), and circular shift right (CSR)

The flags are set as follows:

Zero (Z): Set if the ALU output is zero. Negative (N): Set if the result of the operation is negative (MSB of ALUOut is set: if it is 8 bit it sets as the 8th bit. if it is 15 bit, it sets as 15th bit). Carry (C): Set if there is a carry out from the most significant bit in addition or from the least significant bit in a shift operation. Overflow (O): Set if an arithmetic operation results in overflow, which occurs when the result does not fit within the range of a 8 bit numberas and 16-bit numbers. We were choosing which operation to perform by 5-bit function selector (FunSel). The schematic of Arithmetic Logic Unit is in below.

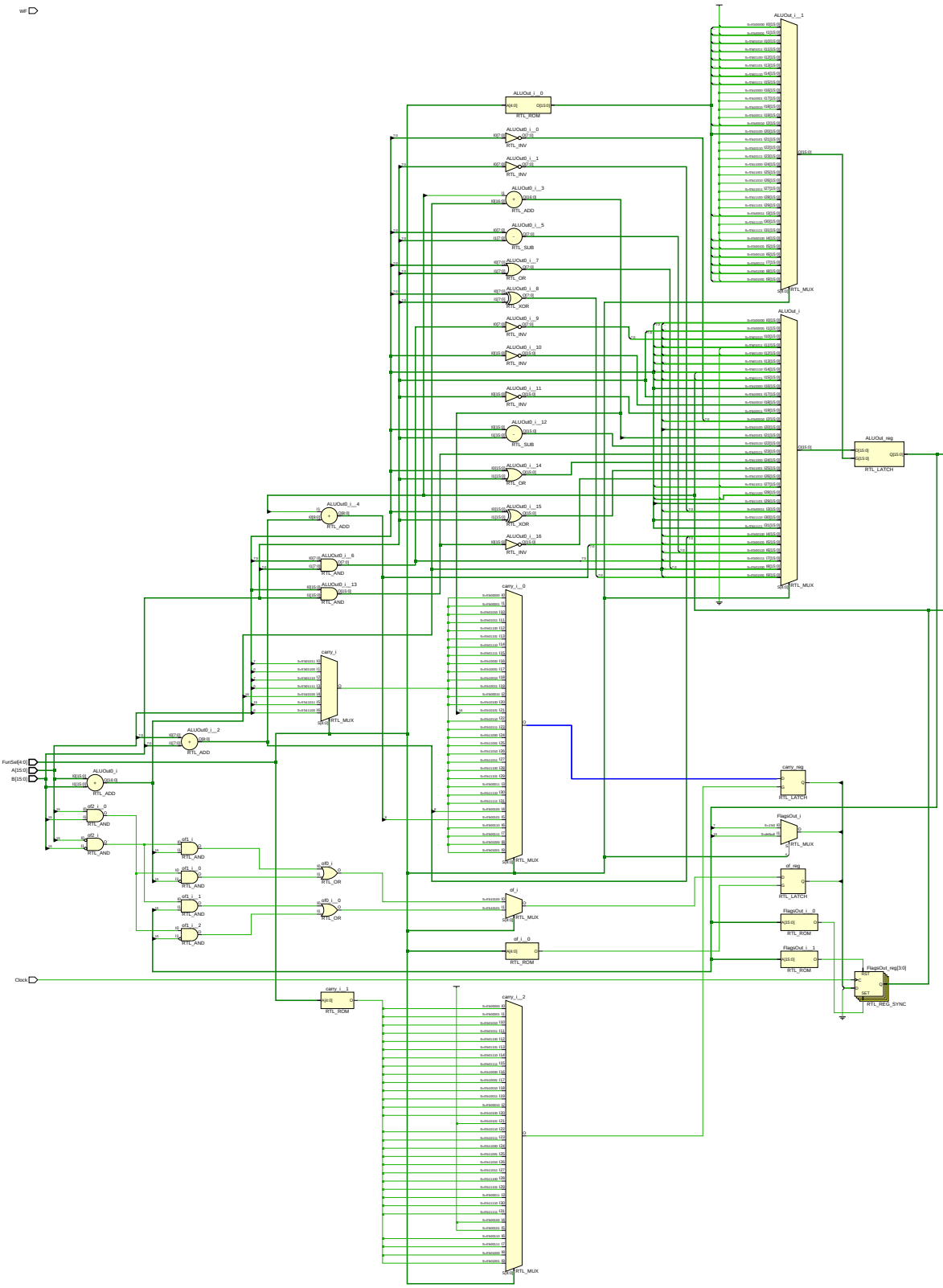

Figure 5: Arithmetic Logic Unit

2.4 Part 4

In the last part of the project, we have connected every component as we wanted to implement the organization called Arithmetic Logic Unit System which is a system that capable of computing logical and arithmetic operations on 8 bit or 16 bit binary numbers. In this part we had Arithmetic Logic Unit (ALU), Register File (RF), Adress Register File (ARF), Memory that provided to us already in the project and instruction register(IR). Here Register File's Outputs of OutA and OutB is connected to ALU's inputs. ALU's output is connected to MuxC, MuxB and MuxA. Memory's output is connected to IR, IR output is connected to MuxA and lastly MuxA's output is to Register File , MuxB's output to ARF and MuxC's output is connected to Memory. We used "wire" type while connecting and assigned them later on to the outputs in the Verilog code. The schematic can be seen below. In the disscussion part, what are they and how they are used will be mentioned.

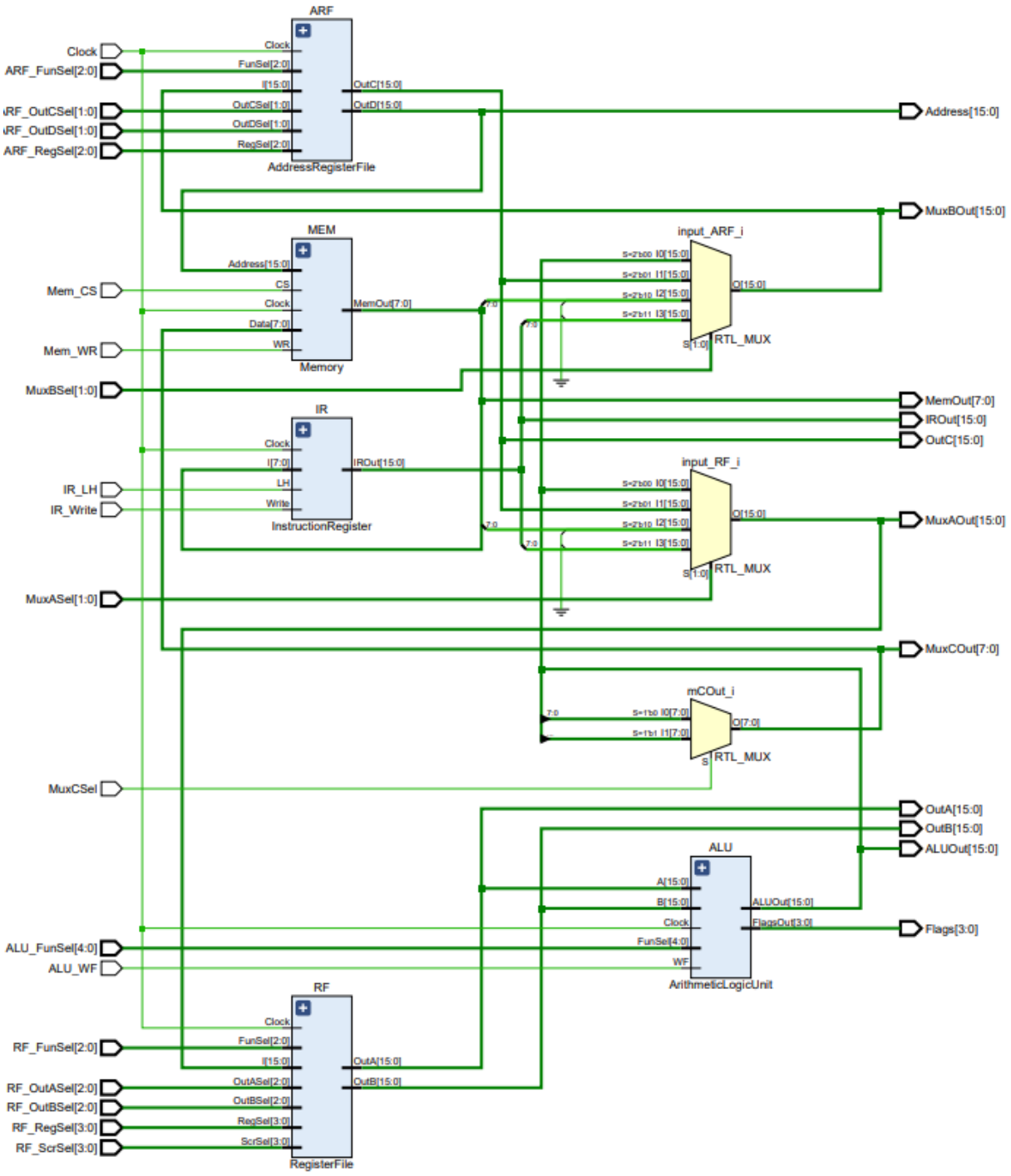

Figure 6: Arithmetic Logic Unit System

3 RESULTS [15 points]

. The tables of inputs with corresponding wanted functions are in discussion part. Here the simulations of given tests and some extra tests. Making extra tests helped us during debugging proccess.

3.1 Part 1

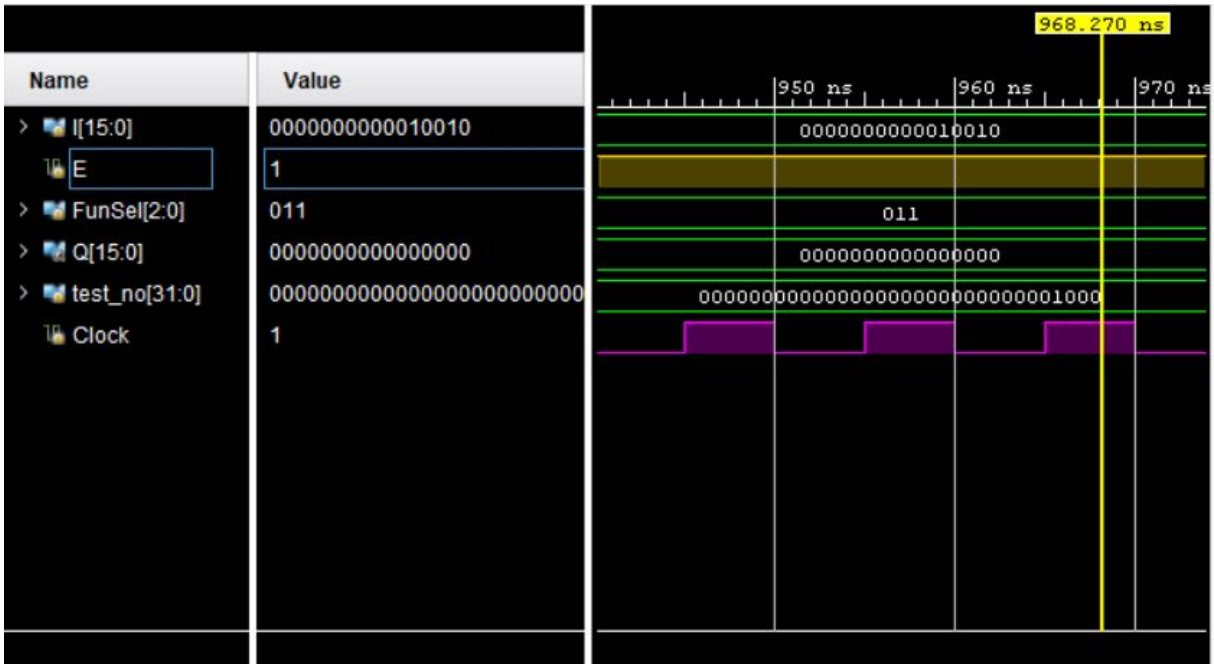

Figure 7: 16 Bit Register that performs clear operation

Our own Register Test and its following table shows inputs and outputs according to functions

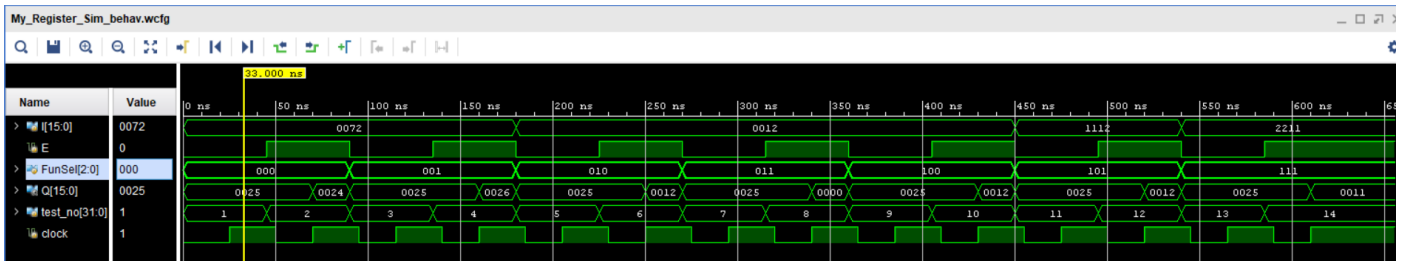

Figure 8: Register Simulation Test

Input	Enable	FunSel	Function	Output at Posedge
R.Q= 0025	0	000	Decrement	0025
R.Q=0025	1	F000	Decrement	0024
R.Q=0025	0	001	Increment	0025
R.Q=0025	1	001	Increment	O026
R.Q=0025, I=0012	0	010	Load	0025
R.Q=0025, I=0012	1	010	Load	0012
R.Q=0025	0	011	Clear	0025
R.Q=0025	1	011	Clear	O000
R.Q=0025, I=0012	0	100	15-8Clear, 7-0 write low	0025
R.Q=0025, I=0012	1	100	15-8Clear, 7-0 write low	0012
R.Q=0025, I=1112	0	101	Only write low	0025
R.Q=0025, I=1112	1	101	Only write low	0012
R.Q=0025, I=2211	0	111	15-8 sign extend, 7-0 write low	0025
R.Q=0025, I=2211	1	111	15-8 sign extend, 7-0 write low	0011

Figure 9: Table of our Simulation Tests

3.2 Part 2

3.2.1 Part 2a)

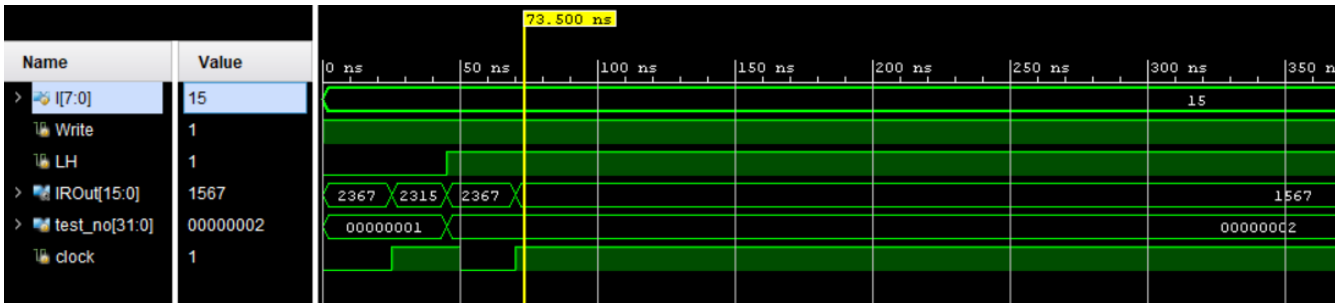

Figure 10: Instruction Register

3.2.2 Part 2b)

Our Own Test bench that checks more conditions for Register File:

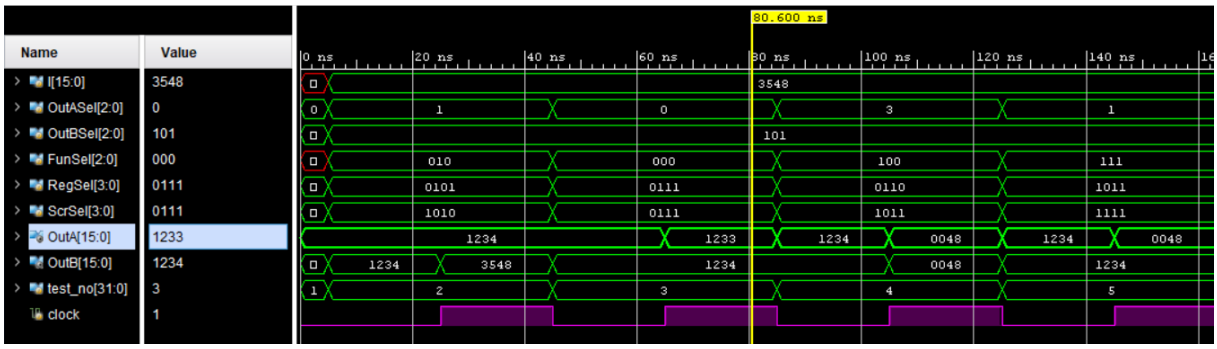

Figure 11: More tests for RF

The Table of inputs and outputs due to our test:

Initial value	I	FunSel	RegSel	ScrSel	OutASel	OutBSel	OutA	OutB
R1= 1234 R2=5678	xxxx	xxx	1111	1111	000	001	1234	5678
all registers set to 1234	3548	010(load)	0101	1010	001	101	1234	3548
all registers set to 1234	3548	000(decrement)	0111	0111	000	101	1233	1234
all registers set to 1234	3548	100(Write low)	0110	1011	011	101	0048	0048
all registers set to 1234	3548	111(15-8 sign extend, 7-0 write low)	1011	1111	001	101	0048	1234

Figure 12: Table of our Simulation Tests

3.2.3 Part 2c)

The given test:

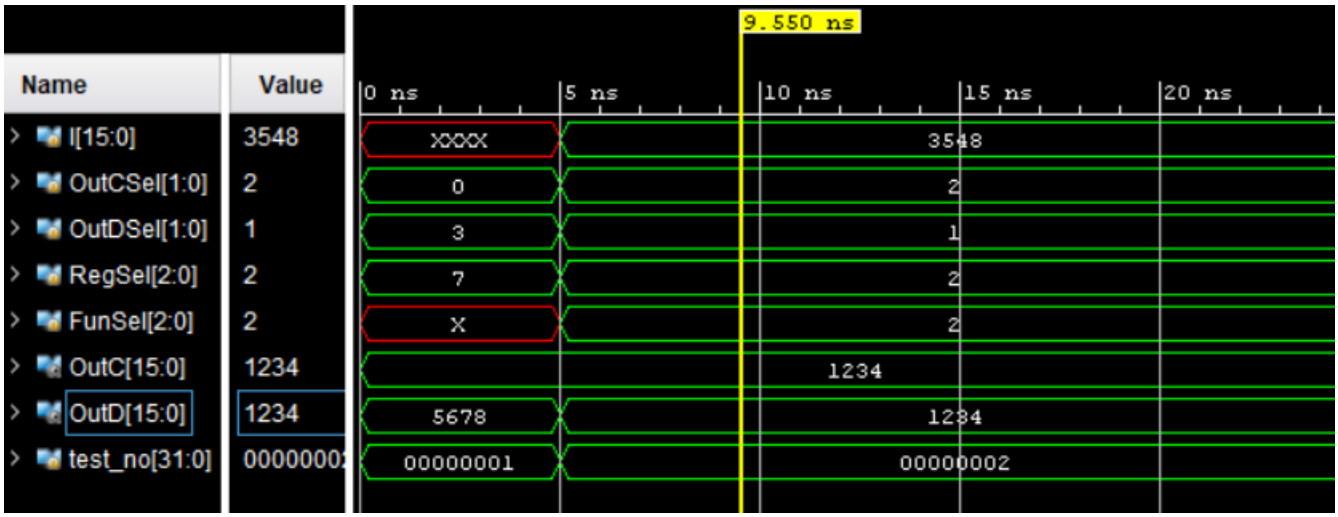

Figure 13: Address Register File

Our own test bench that checks more conditions on ARF:

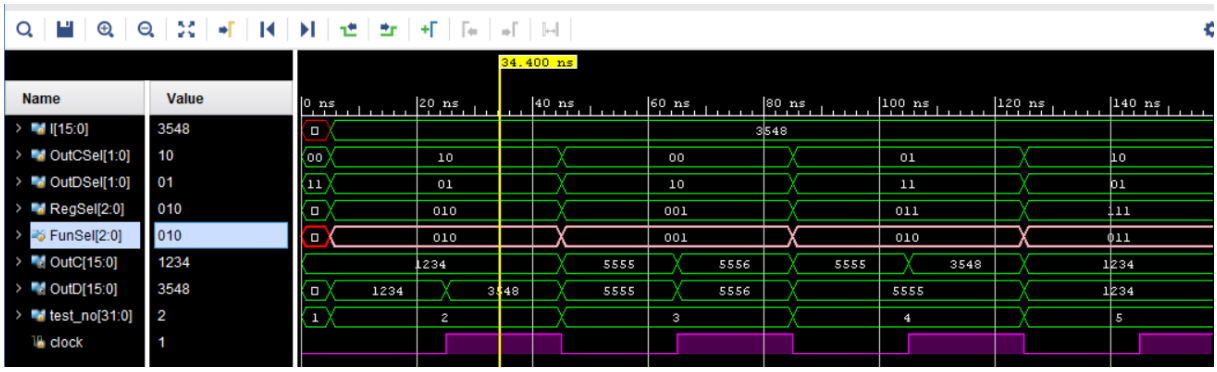

Figure 14: More ARF tests

and it's corresponding inputs and outputs table:

Initial value	I	FunSel	RegSel	OutCSel	OutDSel	OutC	OutD
PC=1234, SP=5678	xxxx	xxx	111	00	11	1234	5678
set all registers to 1234	3548	010(Load)	010	10	01	1234	3548
set all registers to 5555	3548	001(increment)	001	00	10	5556	5556
set all registers to 5555	3548	010(Load)	011	01	11	3548	5555
set all register to 1234	3548	011(Clear)	111	10	01	1234	1234

Figure 15: Inputs and outputs table of our tests

3.3 Part 3

This is the results of the given test bench, the selected FunSels can be seen from organge Funsels. (The table correspounds to them is at discussion part.) Here 10100 and 10101 operations was performed which is 16-bit addition and 16-bit addition with carry respectively.

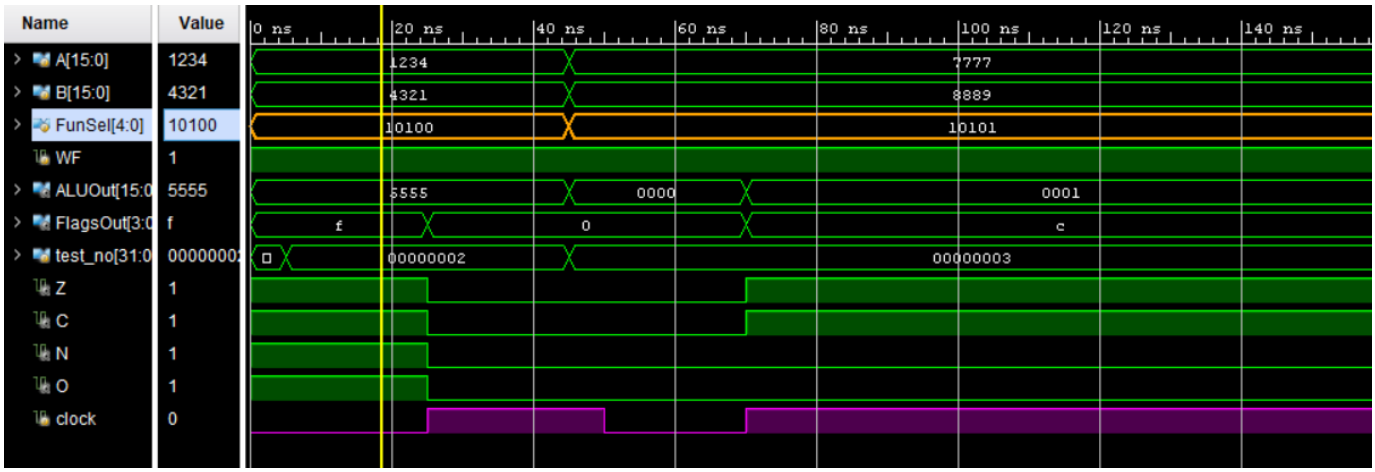

Figure 16: Arithmetic Logic Unit Simulation

Figure 17: Inputs and the outputs of the test cases

and the output of Tcl console as follows

```
-----------------------------------
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 18: Tcl Console of Figure

This is the result of tests that written by us.The operations of FunSel: 00000 (A 8-bit), 10010(16 -bit NOT A), 00010 (NOT A 8-bit), 00100 (8-Bit A+B), 10110 (16-bit A-B), 00110 (8-bit A-B), 11000 (16-bit A OR B), 01000 (8 bit A OR B) was performed in this one.

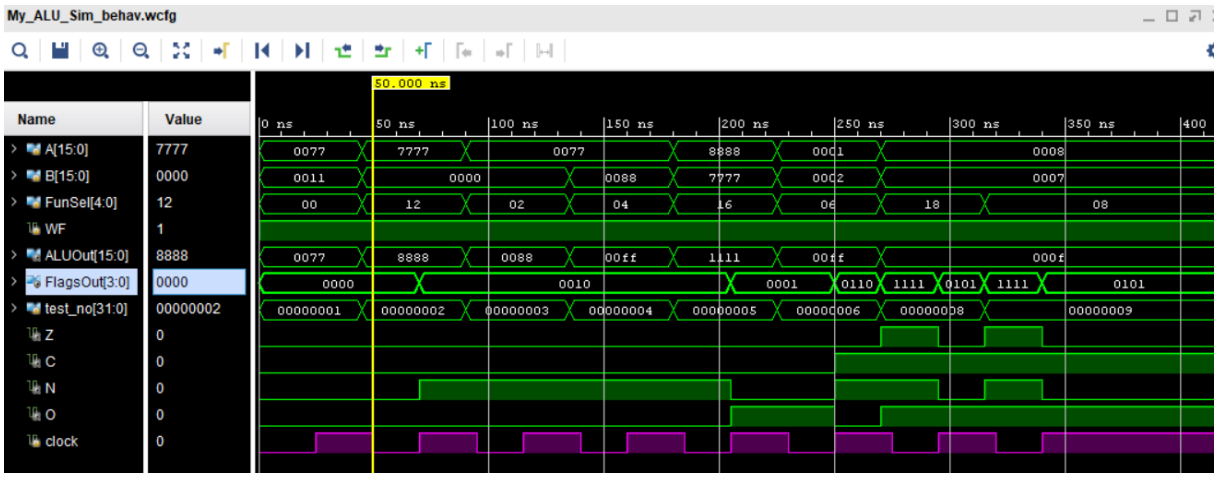

Figure 19: Arithmetic Logic Unit Simulation 2

Table of inputs and corresponding outputs

InputA	InputB	Function	ALUOut	FlagsOut(ZCNO)
h'77	h'11	A(8 bit)	h'0077	0000
h'7777	h'0000	NOT A (16 bit)	h'8888	0010
h'77	h'00	NOT A (8 bit)	h'88	0010
h'77	h'88	A+B (8 bit)	h'ff	0010
h'8888	h'7777	A-B (16 bit)	h'1111	0001
h'01	h'02	A-B (8 bit)	h'ff	0110
h'0008	h'0007	A OR B (16 bit)	h'000f	0101
h'08	h'07	A OR B (8 bit)	h'0f	0101

and the Tcl consol as follows

```
[PASS] Test No: 1, Component: Z, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 1, Component: N, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 2, Component: ALUOut, Actual Value: 0x8888, Expected Value: 0x8888
[PASS] Test No: 2, Component: Z, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 2, Component: N, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 3, Component: ALUOut, Actual Value: 0x0088, Expected Value: 0x0088
[PASS] Test No: 3, Component: Z, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 3, Component: N, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 4, Component: ALUOut, Actual Value: 0x00ff, Expected Value: 0x00ff
[PASS] Test No: 4, Component: Z, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 4, Component: N, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 4, Component: Z, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 4, Component: N, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 5, Component: ALUOut, Actual Value: 0x1111, Expected Value: 0x1111
[PASS] Test No: 5, Component: Z, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 5, Component: C, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 5, Component: N, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 5, Component: O, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 6, Component: ALUOut, Actual Value: 0x00ff, Expected Value: 0x00ff
[PASS] Test No: 6, Component: Z, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 6, Component: C, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 6, Component: N, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 6, Component: O, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 8, Component: ALUOut, Actual Value: 0x000f, Expected Value: 0x000f
[PASS] Test No: 8, Component: Z, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 8, Component: C, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 8, Component: N, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 8, Component: O, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 9, Component: ALUOut, Actual Value: 0x000f, Expected Value: 0x000f
[PASS] Test No: 9, Component: Z, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 9, Component: C, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 9, Component: N, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 9, Component: O, Actual Value: 0x0001, Expected Value: 0x0001
ArithmeticLogicUnit Simulation Finished
0 Test Failed
34 Test Passed
```

Figure 20: Tcl Console of ALU Simulation 2

3.4 Part 4

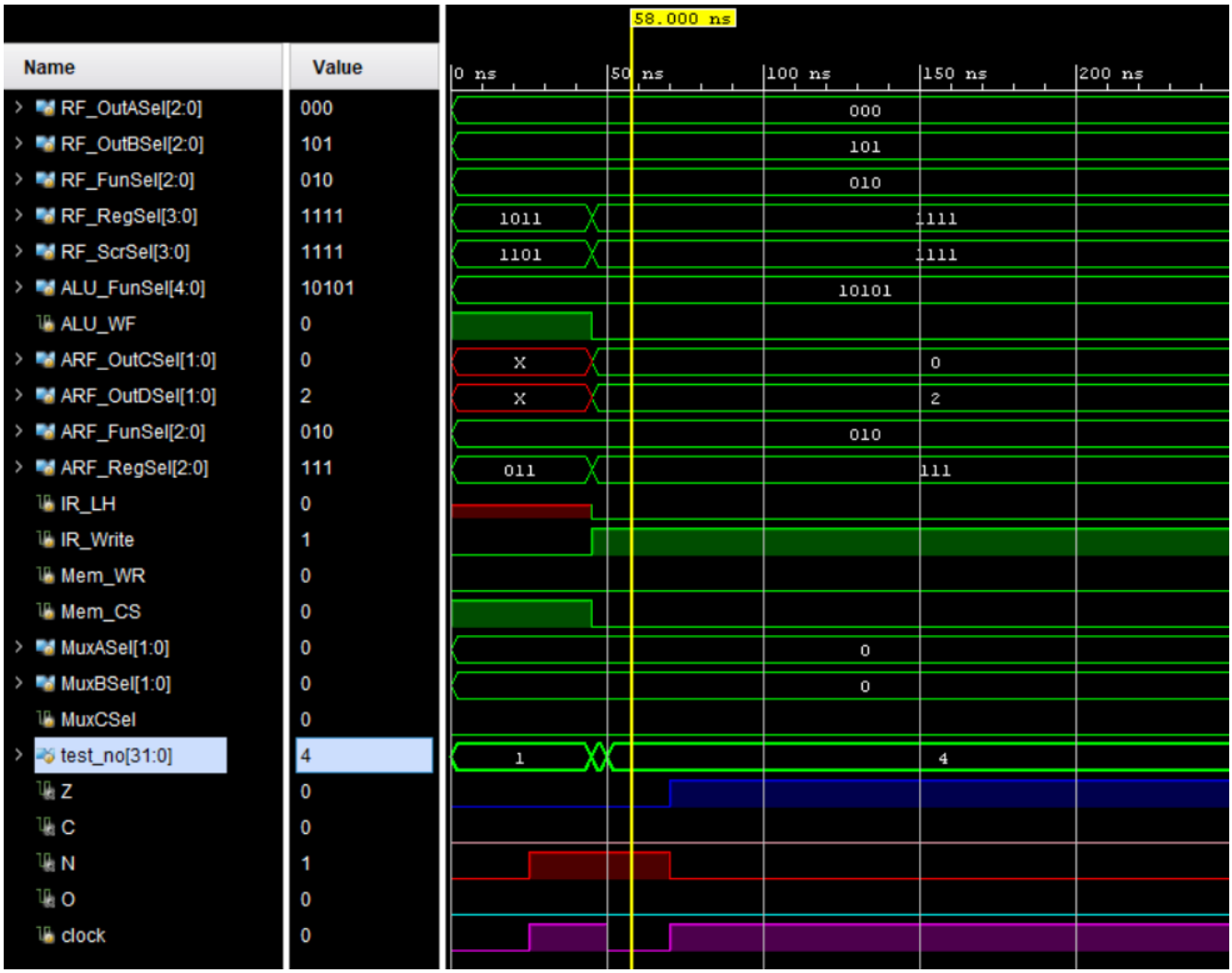

Figure 21: Arithmetic Logic Unit System

4 DISCUSSION [25 points]

4.1 Part 1

In part 1, the design and implementation of the 16-bit register was wanted. The implementation was aimed at achieving the functionalities outlined in the project description, controlled through a combination of the enable input (E) and 3-bit control signals (FunSel).

The characteristic table provided in the project description (Figure 22) served as a critical reference throughout the design process, guiding the configuration of the register's functionalities. These included nine operations such as retain value, decrementing, incrementing, loading specific values, clearing, and various modes of writing data to either the high or low bits of the register or performing sign extension based on a 8-bit input. Based on the essential characteristic table whenever the enable (E) input was 0, the FunSel function acted as "Dont care" and the output retained it's value. When the Enable is 1, the Funsel functions are as follows in the table.

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

Figure 22: Funnction Selection inputs and Operations

4.2 Part 2

4.2.1 Part 2a)

In this part, we implemented Instruction Register that plays crucial role for the ALU system. Since, when the CPU fetches instruction from memory, it loads it into the instruction register so it can decode and execute it as we have learned from our lecture classes. We used the test cases that we are already given and made some extra cases to be sure of it working right. As it can be seen from Results part 2a, it Loads when the write signal is 1 to the appropriate bits according to the L/H signal being 1 or 0. The input table is below.

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 23: The table of IR

4.2.2 Part 2b)

In this part, for enabling the registers we noticed the pattern that each register was being enabled when the RegSel bits are "zero". As it can be seen from the Figure 25, when 0001 R1, R2 and R3 is enabled which corresponds to the zeroes of RegSel's bits. Because of this , while writing the code we put the Regsel to the enable of the Registers as "not(tilda) Regsel[n]" (n corresponds to 0-4). We again tested our implementation firstly via given test cases then we have written similar test cases to be sure of the correctness. For the selections of OutA and OutB multiplexers are used for implementing the following given tables. As it can be seen in the results part the test case was at first disabling and clearing all registers then was giving to R1 1234 value and to R2 5678 value. For OutASel the selection 000 was indicating that R1 was selected for OutA and for OutBSel the selection 001 was indicating that R2 was selected for OutB as can be seen in the first 5 ns. The OutASel and OutBSel tables and the RegSel tables are available below. Operations are done due to those tables.

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 24: Selection of OutA and OutB

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

Figure 25: RegSel Control Input and according enabling operation

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

Figure 26: ScrSel Control Input and according enabling operation

4.2.3 Part 2c)

While implementing Address Register File(ARF) , we noticed the same pattern as in the Part 2b. The 3 bit RegSel was XXX corresponding PC AR SP from left to right, and when the RegSel's bit becomes zero, it was enabling the corresponding Register. For example for the case 010 PC and SP enabled. We did the given test cases and also added some extra test cases to our test bench to control whether we have an error or not. For instance,the first test case was doing that at first disabling all, then assigning PC and SP hexadecimal 1234 and 5678 values respectively, and then Selecting OutCSel as 00 and OutDSel as 11. Afterwards we saw the result in the simulation for OutC 1234 and for OutD 5678.

The tables that defines the structure in the project as seen in the below.

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

Figure 27: RegSel selects Registers

OutCSel	OutC
00	PC
01	PC
10	AR
11	SP

OutDSel	OutD
00	PC
01	PC
10	AR
11	SP

Figure 28: OutCSel and OutDSel controls

4.3 Part 3

In the part 3, we were wanted to implement ALU itself. The wanted ALU required to perform both logical and mathematical calculations by selecting from 5 bit FunSel function selector as in the table (Figure 29). The 8 bit operations performed only using 8 bits, carry and overflow was found accordingly for 8 bit numbers. We resetted the MSB 8 bits to zero before performing the opearations for 8 bit part. In our implementation, we used a temp carry and overflow then at the end of the code , we made them assigned to our flag when the clock pulse comes.Also we checked the negativity and zero conditions at

the end of the code with small if conditions. Our code checks when the flags are needed as in the table and updates them when the clock pulse arrives.

The testing for the ALU took lots of our time in the project. Because we were needed to understand the structure first as the flags are changing after clock positive edge arrives and for each test case we were expected to calculate carry, overflow, the zero condition and the overflow condition. We set the conditions correctly and observed the correct outputs from ALU operations.

As an example highlighted in Figure 16, we executed a test for FunSel set to 10100, corresponding to a 16-bit addition. This operation effectively utilizes the entire 16-bit width of the ALU, allowing us to thoroughly test the ALU's ability to handle wide operations without any loss of data or incorrect flag settings. Initially, all flags were set to 1, simulating a worst-case scenario where overflow, carry, negative, and zero flags might all be asserted due to a previous operation. The result after applying the FunSel operation A+B, with inputs A: 1234 and B: 4321, yielded the correct output of 5555. The subsequent flag results – Z, C, N, O – all reset to 0, aligning with the expected behavior where the operation does not result in an overflow, carry, negative value, or zero result. This specific behavior underscores the importance of the flags' reset mechanism, which occurs on the positive edge of the clock signal, confirming the synchronous nature of our ALU design.

FunSel	ALUOut	Z	C	N	O
00000	A (8-bit)	+	-	+	-
00001	B (8-bit)	+	-	+	-
00010	NOT A (8-bit)	+	-	+	-
00011	NOT B (8-bit)	+	-	+	-
00100	A + B (8-bit)	+	+	+	+
00101	A + B + Carry (8-bit)	+	+	+	+
00110	A – B (8-bit)	+	+	+	+
00111	A AND B (8-bit)	+	-	+	-
01000	A OR B (8-bit)	+	-	+	-
01001	A XOR B (8-bit)	+	-	+	-
01010	A NAND B (8-bit)	+	-	+	-
01011	LSL A (8-bit)	+	+	+	-
01100	LSR A (8-bit)	+	+	+	-
01101	ASR A (8-bit)	+	+	-	-
01110	CSL A (8-bit)	+	+	+	-
01111	CSR A (8-bit)	+	+	+	-

FunSel	ALUOut	Z	C	N	O
10000	A (16-bit)	+	-	+	-
10001	B (16-bit)	+	-	+	-
10010	NOT A (16-bit)	+	-	+	-
10011	NOT B (16-bit)	+	-	+	-
10100	A + B (16-bit)	+	+	+	+
10101	A + B + Carry (16-bit)	+	+	+	+
10110	A – B (16-bit)	+	+	+	+
10111	A AND B (16-bit)	+	-	+	-
11000	A OR B (16-bit)	+	-	+	-
11001	A XOR B (16-bit)	+	-	+	-
11010	A NAND B (16-bit)	+	-	+	-
11011	LSL A (16-bit)	+	+	+	-
11100	LSR A (16-bit)	+	+	+	-
11101	ASR A (16-bit)	+	+	-	-
11110	CSL A (16-bit)	+	+	+	-
11111	CSR A (16-bit)	+	+	+	-

Figure 29: Characteristic table of ALU

4.4 Part 4

In the last part of the project, we achieved to create a system called Arithemetic Logic Unit System that available of making logical and arithmetic operations thanks to ALU.

This system utilizes the ALU and all other components as ARF, Memory, IR, RF and some multiplexers between them with the same clock. We did the given test cases for this part but we checked testing mostly for ALU part because when it resulted wrong , ALU System was resulting wrong either. ALU System is nothing other than the correct connections between it's components. In this implementation the IR register is not used fully, it using half of bits inside it, from memory the load is coming as 8 bits. In the RF there are 8 registers which allows system for a good balance between performance and complexity. In the ARF, there are PC, AR and SR as discussed earlier.

MuxASel	MuxAOut
00	ALUOut
01	ARF OutC
10	Memory Output
11	IR (7:0)

Figure 30: MuxA Selections and Outputs

MuxBSel	MuxBOut
00	ALUOut
01	ARF OutC
10	Memory Output
11	IR (7:0)

Figure 31: MuxB Selections and Outputs

MuxCSel	MuxCOut
0	ALUOut(7-0)
1	ALUOut(15-8)

Figure 32: EMuxC Selections and Outputs

5 CONCLUSION [10 points]

In conclusion, the journey through the four parts of this project has been as challenging as it has been rewarding. At the beginning of project, we tried to recognise the structure of register. At first, it was not easy since we were beginners in Verilog language. Then in Part 2's a, b, c parts, we needed to learn how to use previous Register module and how to combine them togather which we were required for in the last part for the system. While Parts 1 and 2 provided a solid foundation in register operations and control signal processing, it was Part 3, the design of the Arithmetic Logic Unit (ALU), where we encountered significant complexities. Managing the carry flags across various

arithmetic operations and ensuring their correct behavior in sync with the clock cycles was demanding. The precision required to implement the carry logic in both 8-bit and 16-bit operations correctly, and to synchronize it with the clock, tested our understanding of digital design principles and our attention to detail.

When we arrived at part 4 , at first we could not grasp which input to put where. However, later on, we recognised that, it was just wires and nothing more. As in the Part 2 to implement Register File or Adress Register File, we were using the previous module Register. While we using older modules, we noticed small errors and fixed them. This time, only there was too much inputs and outputs that wires was used between them. That is the reason why we needed to keep track on carefully. We used the old modules and connected their ports correctly by checking the given Figure in the Project1 pdf. When we encountered bugs in the system, we noticed that older parts such as ALU was giving errors and when we corrected the issues, our system began to give correct results. At the end of the project, although it seemed to be a though project to us, we managed to succeeded it as a team and learned the concept of computer organization as we were lectured in the BLG222E course at university. Breafly, the implementation of this Project made our understanding stronger.

REFERENCES