ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E
COMPUTER ORGANIZATION
PROJECT 1 REPORT

CRN	:	21335
LECTURER	:	Prof. Dr. Deniz Turgay Altılar

GROUP MEMBERS:

150210083	:	Kemal Tahir Bıcılıoğlu
150210097	:	Ali Emre Kaya

SPRING 2024

Contents

1 INTRODUCTION [10 points]

In this project, our purpose is to create an Arithmetic Logic Unit (ALU) using Verilog HDL. In computing, an ALU is a combinational digital circuit that performs arithmetic and bit-wise operations on integer binary numbers. It is the fundamental building block of many types of computing circuits. In the first part of the project, we implement the 16-bit register module, which can make eight different operations. We will use that module in the following parts. In the second part, we implement three different register files, which will do instruction and output sender, respectively, to the selectors. In the third part, we create the arithmetic logic unit, which is the most important part. It will calculate the proper operands and pay attention to flags like overflow and carry. In the last part, all the modules are combined to create an arithmetic logic unit system that uses an external memory.

The task distribution is as follows:

- **Part 1:** Ali Emre
- **Part 2.a:** Ali Emre
- **Part 2.b:** Kemal Tahir
- **Part 2.c:** Kemal Tahir
- **Part 3:** Kemal Tahir
- **Part 4:** Ali Emre
- **Report:** Kemal Tahir - Ali Emre

2 MATERIALS AND METHODS [40 points]

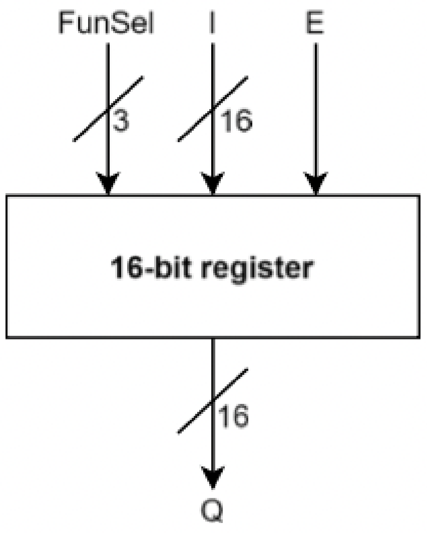
2.1 Materials

- Verilog
- Register
- Instruction Register
- Register File
- Address Register File
- Arithmetic Logic Unit
- 2 to 1 Multiplexer
- 4 to 1 Multiplexer
- Memory Unit

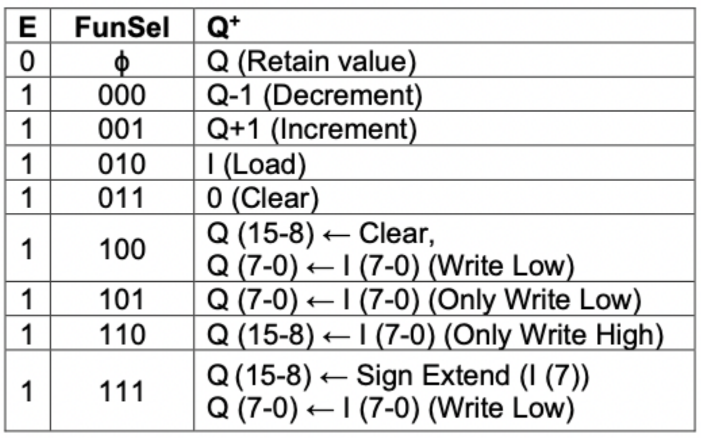
2.2 Methods

2.2.1 Project Part 1

In the first part, we should implement a 16-bit register, the most fundamental part of the whole experiment. A register, a crucial component in digital electronics, functions as a memory device capable of holding a defined quantity of data bits. Its primary roles include storing instructions while the processor handles other operations or retaining arithmetic outcomes until they are needed for subsequent computational steps. To implement a 16-bit register, we should use Enable, FunSel, Input, Output, and Clock as variables. The graphic symbol of the register is given below:



(a) 16-bit Register



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)

(b) Characteristic Table

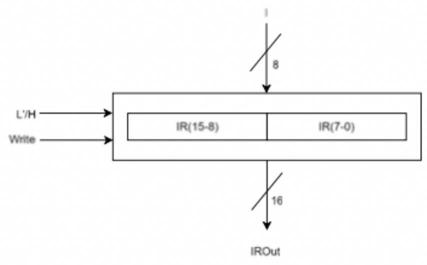
Figure 1

The clock is not shown in the graphic, but it is a requirement because it triggers the operations, respectively, on the rising edge. The rising edge is an important concept in electronics because it is generally used to trigger or synchronize other circuit elements.

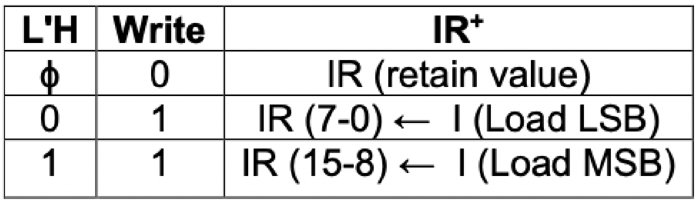
The working principle of this register depends on FunSel's and Enable's values, which are the given characteristic tables (*figure 1*): FunSel will select the proper operation and apply that to the output.

2.2.2 Project Part 2.1

In the first part of Part 2, our mission is to implement a basic register file with respect to the given function table. We need an 8-bit input, a 2-bit L'H signal, a case selector, a clock, and a 16-bit output.



(a) Instruction Register



L'H	Write	IR^{+}
ϕ	0	IR (retain value)
0	1	IR (7-0) $\leftarrow$ I (Load LSB)
1	1	IR (15-8) $\leftarrow$ I (Load MSB)

(b) Characteristic Table

Figure 2: Graphic symbol of IR register and its characteristic table, respectively.

The clock is not described in the symbol, but it has the same purpose as Part 1's clock. Respectively to the function table, there are only two cases; input will be arranged to the most significant 8-bit or least significant 8-bit. We will use the always block to select which operations will operate. We will check whether these cases can be handled according to the value of L'H; we will use the if-else statement for this.

2.2.3 Project Part 2.2

In Part 2-b, we will create a register file (RF) module. The purpose of the register file module is to control the 4 16-bit general-purpose registers and the 4 16-bit scratch registers. To achieve this, the module has six inputs, which are Input, OutASel, OutBSel, FunSel, RegSel, and ScrSel. Also, as a register, we will use the register module, which we implemented in the first part of the project. The graphic symbol of the RF module is given below.

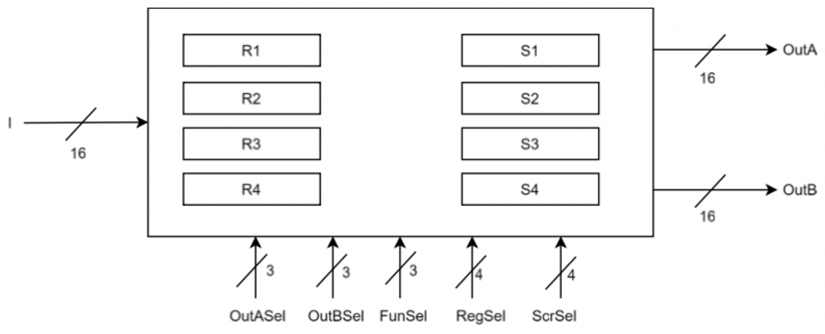


Figure 3: 16-bit general purpose and scratch registers, inputs, and outputs.

OutASel and OutBSel will select the output of the register for the output of the RF module. The control table is given below.

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 4: OutASel and OutBSel controls.

FunSel will determine the operation for the selected registers, depending on this table.

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

Figure 5: FunSel Control Input.

Lastly, RegSel and ScrSel will determine the active and passive registers, depending on these tables.

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.)

(a) RegSel Control Input

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.)

(b) ScrSel Control Input

Figure 6

2.2.4 Project Part 2.3

Address Register File (ARF) contains FunSel, RegSel, OutcSel, OutDSel, 16-bit input, 2 16-bit outputs, and three new registers: program counter (PC), address register (AR), and stack pointer (SP). A graphic symbol is given below.

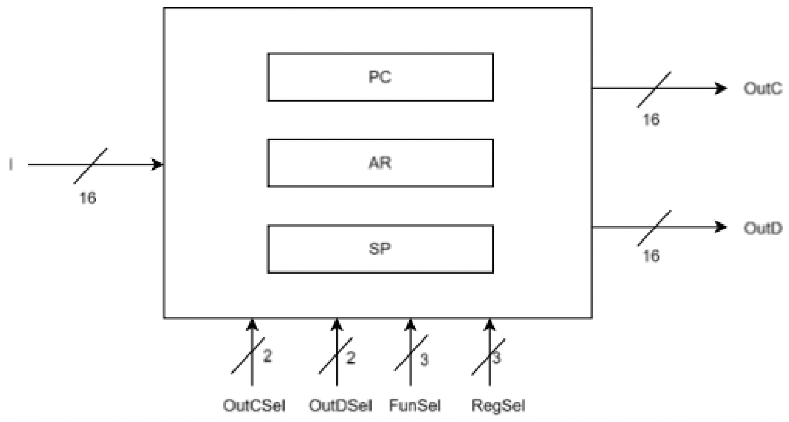


Figure 7: 16-bit address registers, inputs, and outputs.

RegSel will select enabled and disabled registers again; if we look carefully, that can be shown as registers selected by RegSel's inverse. For example, (011) is enabled for only the first one (PC). It can be noticed better by examining the control table.

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	SP is enabled. (Function selected by FunSel will be applied to SP.)
111	NO address register is enabled. (All registers retain their values.)

Figure 8: RegSel control table

Outputs will be selected by the output selectors, which are OutCSel and OutDSel, corresponding to the outputs given in the in the table.

OutCSel	OutC
00	PC
01	PC
10	AR
11	SP

OutDSel	OutD
00	PC
01	PC
10	AR
11	SP

Figure 9: OutCSel and OutDSel controls.

2.2.5 Project Part 3

This part of the project is designing the arithmetic logic unit (ALU). It has six ports: A and B, which are 16-bit inputs; FunSel, which is the case selector; WF, the signal controller; Clock; ALUOut, which is a 16-bit output; and 4-bit FlagsOut, which holds the zero, carry, negative, and overflow variables. Those are compulsory variables; we also defined ourselves variables, which are scratch (17-bit), next_flag (4-bit), extendedA (16-bit), and extendedB (16-bit). scratch is used for determining the overflow; we need to check its most significant bit to understand the overflow. This is not possible with 16-bit variables because we need to access the 17th bit, so we need this scratch variable. next_flags is a 4-bit variable that holds the next state of the flags. extended variables will be clarified in the next page. The graphic of ALU is given below.

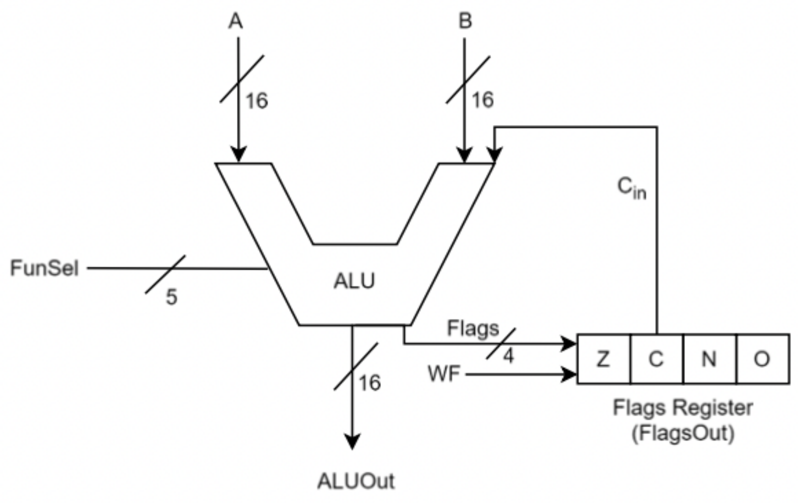
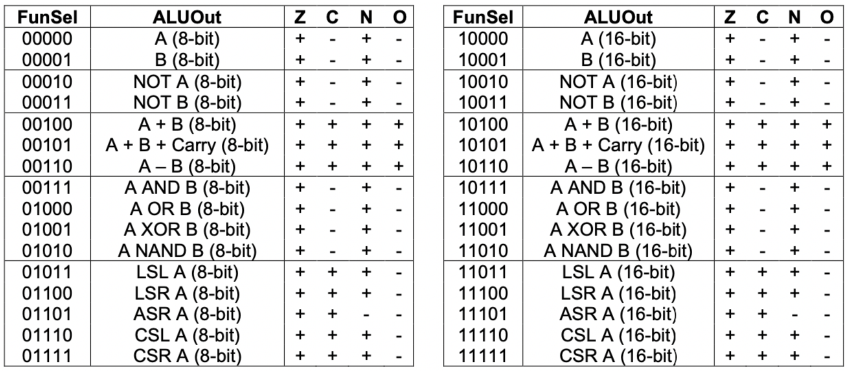


Figure 10: Arithmetic Logic Unit graphic.

FunSel's task is similar to that in other parts: determining what is to happen according to cases. The case table is given below.

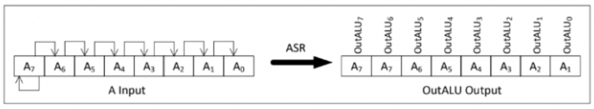


FunSel	ALUOut	Z	C	N	O	FunSel	ALUOut	Z	C	N	O
00000	A (8-bit)	+	-	+	-	10000	A (16-bit)	+	-	+	-
00001	B (8-bit)	+	-	+	-	10001	B (16-bit)	+	-	+	-
00010	NOT A (8-bit)	+	-	+	-	10010	NOT A (16-bit)	+	-	+	-
00011	NOT B (8-bit)	+	-	+	-	10011	NOT B (16-bit)	+	-	+	-
00100	A + B (8-bit)	+	+	+	+	10100	A + B (16-bit)	+	+	+	+
00101	A + B + Carry (8-bit)	+	+	+	+	10101	A + B + Carry (16-bit)	+	+	+	+
00110	A – B (8-bit)	+	+	+	+	10110	A – B (16-bit)	+	+	+	+
00111	A AND B (8-bit)	+	-	+	-	10111	A AND B (16-bit)	+	-	+	-
01000	A OR B (8-bit)	+	-	+	-	11000	A OR B (16-bit)	+	-	+	-
01001	A XOR B (8-bit)	+	-	+	-	11001	A XOR B (16-bit)	+	-	+	-
01010	A NAND B (8-bit)	+	-	+	-	11010	A NAND B (16-bit)	+	-	+	-
01011	LSL A (8-bit)	+	+	+	-	11011	LSL A (16-bit)	+	+	+	-
01100	LSR A (8-bit)	+	+	+	-	11100	LSR A (16-bit)	+	+	+	-
01101	ASR A (8-bit)	+	+	-	-	11101	ASR A (16-bit)	+	+	-	-
01110	CSL A (8-bit)	+	+	+	-	11110	CSL A (16-bit)	+	+	+	-
01111	CSR A (8-bit)	+	+	+	-	11111	CSR A (16-bit)	+	+	+	-

Figure 11: Characteristic table of ALU.

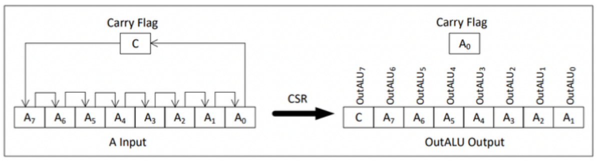
The tricky part is doing operations with different count bits. We need to extend binary numbers, respectively, to their most significant bits to correctly do the operations when we work on 8-bit numbers. extendedA and extendedB variables tasks is holding those extended binary numbers.

Besides the logic operations, there are also shift operations, which are the circular shift operation (CSO), the logical shift operation (LSO), and the arithmetic shift operation (ASO). Those are defined as shown in the tables.



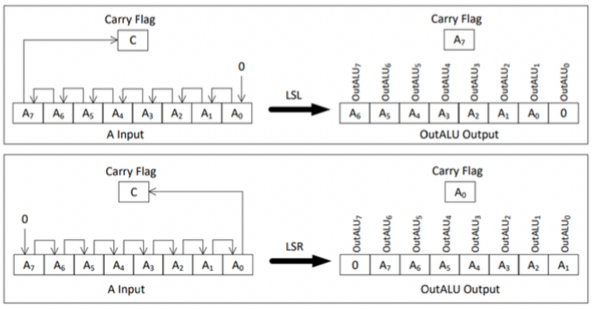
A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0
A Input
ASR
$OutALU_7$ $OutALU_6$ $OutALU_5$ $OutALU_4$ $OutALU_3$ $OutALU_2$ $OutALU_1$ $OutALU_0$
A_7 A_7 A_6 A_5 A_4 A_3 A_2 A_1
OutALU Output

Figure 12: Arithmetic shift operation.



Carry Flag
C
A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0
A Input
CSR
Carry Flag
A_0
$OutALU_7$ $OutALU_6$ $OutALU_5$ $OutALU_4$ $OutALU_3$ $OutALU_2$ $OutALU_1$ $OutALU_0$
C A_7 A_6 A_5 A_4 A_3 A_2 A_1
OutALU Output

Figure 13: Circular shift operation.



Carry Flag
C
0
A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0
A Input
LSL
Carry Flag
A_7
$OutALU_7$ $OutALU_6$ $OutALU_5$ $OutALU_4$ $OutALU_3$ $OutALU_2$ $OutALU_1$ $OutALU_0$
A_6 A_5 A_4 A_3 A_2 A_1 A_0 0
OutALU Output
Carry Flag
C
0
A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0
A Input
LSR
Carry Flag
A_0
$OutALU_7$ $OutALU_6$ $OutALU_5$ $OutALU_4$ $OutALU_3$ $OutALU_2$ $OutALU_1$ $OutALU_0$
0 A_7 A_6 A_5 A_4 A_3 A_2 A_1
OutALU Output

Figure 14: Logical shift operation.

2.2.6 Project Part 4

This section is a combination of the sections we did before. We already designed all the modules to use in this part, except multiplexers. A multiplexer (mux) is a digital circuit that selects one of several input signals and forwards it to a single output. Controlled by selection inputs, it efficiently routes data, facilitating tasks like data routing, signal selection, and memory address decoding within electronic systems. We need two different types of multiplexers to design the arithmetic logic unit system (ALUSys), which are 4 to 1 and 2 to 1.

Unlike the other parts, we will use the memory module in this section. In the memory module, CS (Chip Select) and WR (Write) are vital signals. CS enables the memory module, directing it to respond to incoming address and data signals. WR, when activated with CS, indicates data should be written into the specified memory location. These signals manage data flow, ensuring accurate reading and writing operations within the computer system.

The itemization of the variables in the code is prepared according to the names of the variables in the test file.

Module: mux4to1 (muxA)

- **Inputs:** in1, in2, in3, in4, sel
- **Output:** out

Module: mux2to1 (muxB)

- **Inputs:** in1, in2, sel
- **Output:** out

Module: RegisterFile (RF)

- **Inputs:** I (output from MuxA), RF_OutASel, RF_OutBSel, RF_FunSel, RF_RegSel, RF_ScrSel, Clock
- **Outputs:** OutA, OutB

Module: AddressRegisterFile (ARF)

- **Inputs:** I (output from MuxB), ARF_OutCSel, ARF_OutDSel, ARF_FunSel, ARF_RegSel, Clock
- **Outputs:** OutC, OutD

Module: Memory (MEM)

- **Inputs:** Data (output from MuxC), Address (output from ARF), Mem_WR, Mem_CS, Clock
- **Output:** MemOut

Module: InstructionRegister (IR)

- **Inputs:** I (output from MEM), IR_LH, IR_Write, Clock
- **Output:** IROut

Graphic symbol of arithmetic logic unit system is given below,

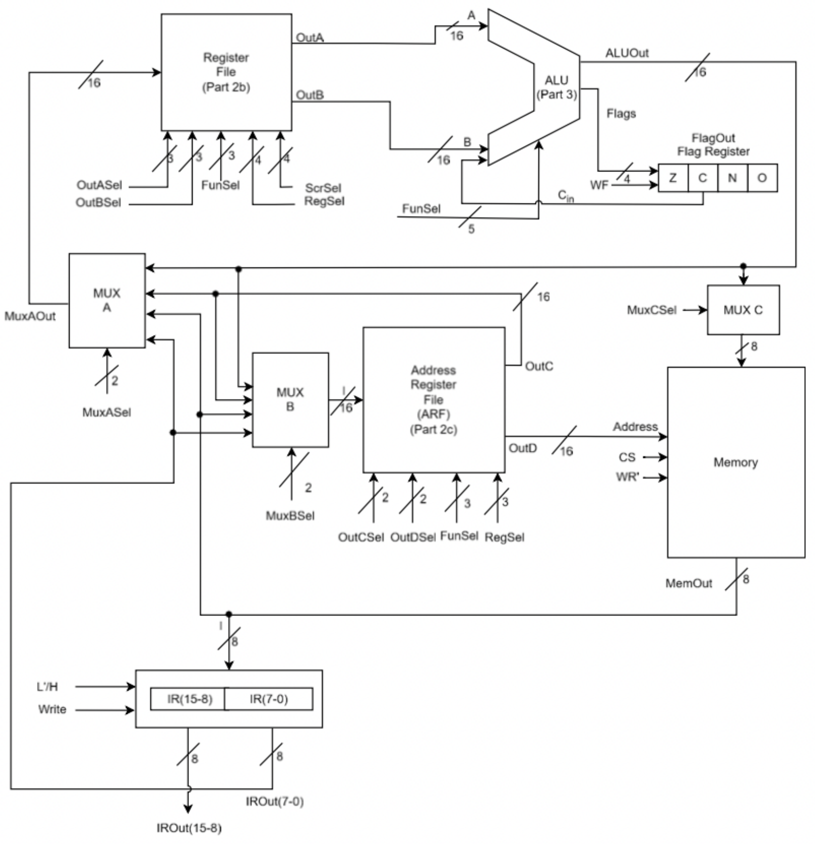


Figure 15: Arithmetic Logic Unit System.

3 RESULTS [15 points]

After we implemented each part, we executed the corresponding behavioral simulation files and examined the outputs. All of the test cases have been passed correctly. After that, we also ensured that our project could successfully execute run.bat and pass all the provided cases.

In part 1, we are asked to create a 16-bit register that performs specific operations based on the corresponding function selection ($FunSel$) applied to the register's input, as outlined in the provided table (*figure 1*). Its behavioral simulation is as the following:

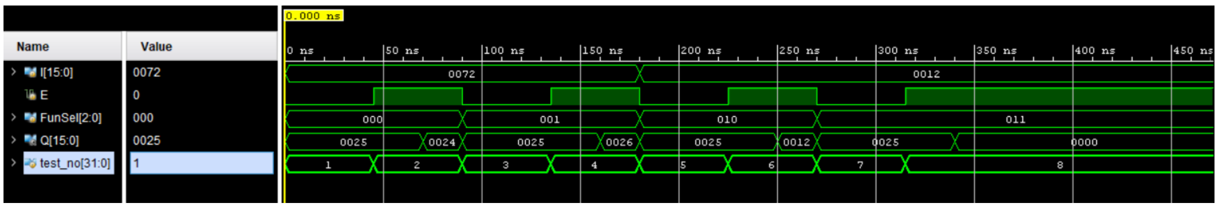


Figure 16: Register.v Simulation

In part 2-a, we are asked to create a 16-bit instruction register that stores binary data. Additionally, we need to determine whether the incoming data will be loaded into the lower or higher bits of the stored binary data (*figure 2*). Its behavioral simulation is as the following:

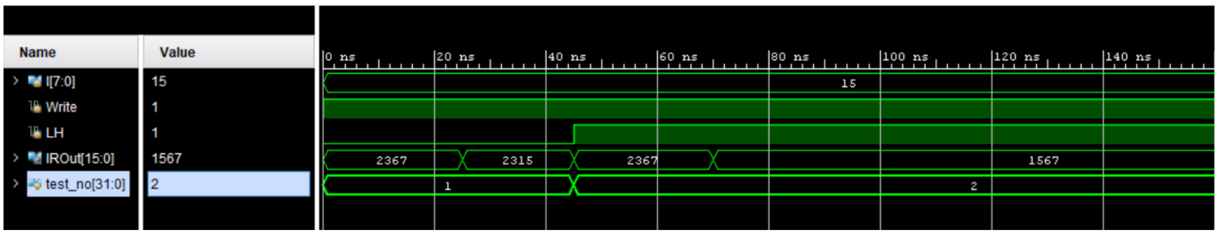
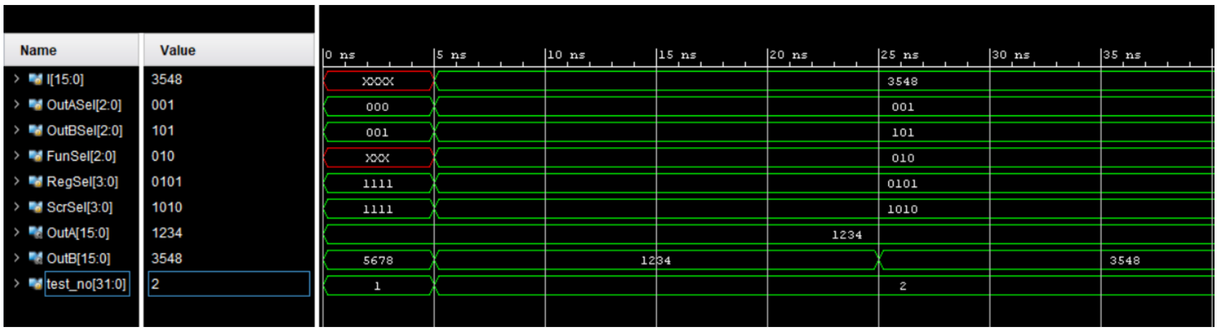


Figure 17: InstructionRegister.v Simulation

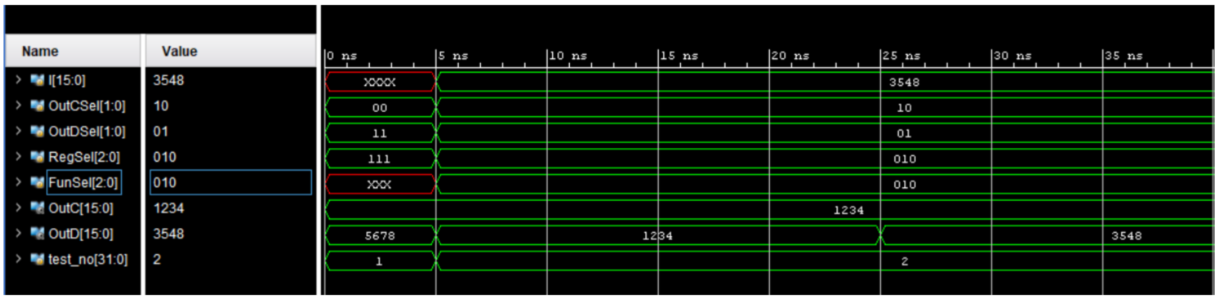
In part 2-b, the task is to create a 16-bit register file with eight registers, four labeled with R and four with S. This register file should be capable of performing the functions of the previously implemented registers. It includes selection components to choose which registers from the R and S groups will be active, as well as to select which registers' outputs will serve as the outputs of the register file. The corresponding selection components are provided in the figure (*figure 3*). Its behavioral simulation is as the following:



Name	Value
I[15:0]	3548
OutASel[2:0]	001
OutBSel[2:0]	101
FunSel[2:0]	010
RegSel[3:0]	0101
ScrSel[3:0]	1010
OutA[15:0]	1234
OutB[15:0]	3548
test_no[31:0]	2

Figure 18: RegisterFile.v Simulation

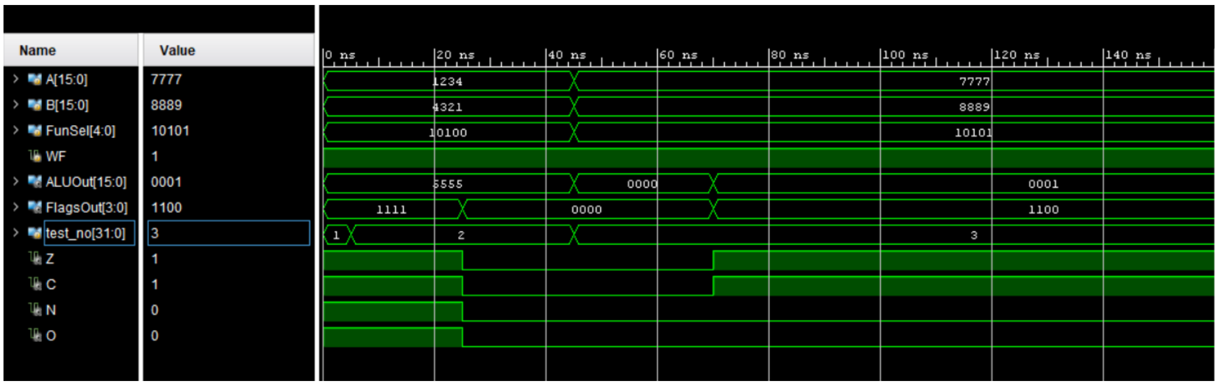
In part 2-c, our task is to create an address register file *ARF* system comprising three 16-bit registers intended for use as address registers. The system includes selection bits to determine which address registers will be active and which register's output will serve as the output of the *ARF*. The corresponding components are illustrated in the provided (*figure 7*). Its behavioral simulation is as the following:



Name	Value
I[15:0]	3548
OutCSel[1:0]	10
OutDSel[1:0]	01
RegSel[2:0]	010
FunSel[2:0]	010
OutC[15:0]	1234
OutD[15:0]	3548
test_no[31:0]	2

Figure 19: AddressRegisterFile.v Simulation

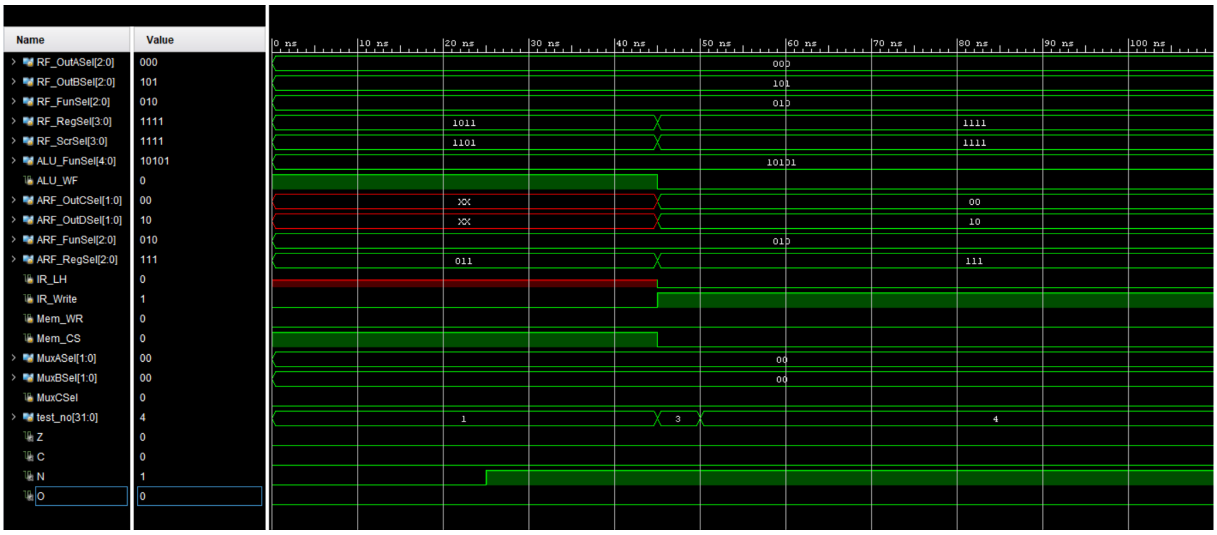
In part 3, our objective is to design an arithmetic logic unit *ALU* capable of performing various mathematical and bit shifting operations, while also generating specific flags associated with certain operations. The *ALU* functions as a combinational circuit, selecting operations based on input selection bits and producing both the operation output and the required flags (*figure 10*). Its behavioral simulation is as the following:



Name	Value
A[15:0]	7777
B[15:0]	8889
FunSel[4:0]	10101
WF	1
ALUOut[15:0]	0001
FlagsOut[3:0]	1100
test_no[31:0]	3
Z	1
C	1
N	0
O	0

Figure 20: Register.v Simulation

In part 4, our task is to integrate all the previously implemented modules into a system that operates using a single clock signal. This integrated system is depicted in the provided (*figure 15*). Its behavioral simulation is as the following:



Name	Value
RF_OutASel[2:0]	000
RF_OutBSel[2:0]	101
RF_FunSel[2:0]	010
RF_RegSel[3:0]	1111
RF_ScrSel[3:0]	1111
ALU_FunSel[4:0]	10101
ALU_WF	0
ARF_OutCSel[1:0]	00
ARF_OutDSel[1:0]	10
ARF_FunSel[2:0]	010
ARF_RegSel[2:0]	111
IR_LH	0
IR_Write	1
Mem_WR	0
Mem_CS	0
MuxASel[1:0]	00
MuxBSel[1:0]	00
MuxCSel	0
test_no[31:0]	4
Z	0
C	0
N	1
O	0

Figure 21: ArithmeticLogicUnitSystem Simulation

4 DISCUSSION [25 points]

In this project, we have learned how to use Verilog to implement hardware designs and how to analyze our implementations by the given simulation files. Overall earnings of dealing with Verilog and hardware design are more detailed knowledge and more understanding about what we have learned in digital circuits and computer organization classes. For all the parts, we first analyzed the simulation files to be able to pass the test cases successfully. The whole implementation works with a clock signal which synchronizes all the parts of the modules to work properly without any hazards. After we had completed each part of the project, we ensured all of the parts with the provided simulation files and saw they passed all the test cases successfully.

In part 1, We are asked to design a 16-bit register that applies desired operations when corresponding function selection (*FunSel*) is applied to the input of the register according to the given table (*figure 1*). To accomplish that, we first analyzed the simulation file to check the names of the input and output lines of the register and how it should be implemented to pass the test cases. We connected the input and output lines of the register and used case structure, which is similar to the switch case structure in C programming language, to apply the desired operations for given *FunSel* input combinations. The implemented register is able to execute the operations when enable input (*E*) is high because enable input is meant to decide whether the register should work or not. If it is low, the register will not be working regardless of the *FunSel* input. Register has also a clock input line to trigger to operate in the rising edge of the clock signal because the integrated circuits in a hardware design should work in a synchronized way otherwise asynchronous signals can lead to hazards because of the various time delays of the logical operations. Its schematic (*Figure 22*) and its behavioral simulation (*Figure 16*) also validate our implementation.

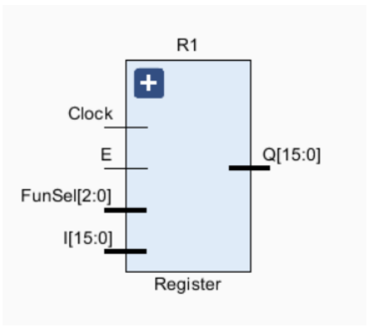


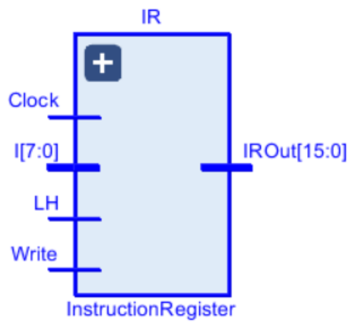
Figure 22: Register Schematic

In part 2, we are given 3 parts a, b, and c.

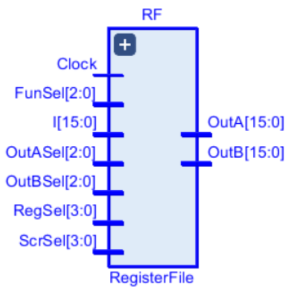
In part 2-a, we are asked to design a 16-bit instruction register which basically stores 16-bit binary data while choosing whether the given input data will be loaded to lower bits or higher bits of the stored binary data. Also, it has a clock signal input line as it should be working in a synchronized way with the other components of the system. Because of that, it will only operate when the clock signal is on the rising edge. It has a write input and it decides if the instruction register will operate or not. If it is low, it will not operate regardless of any input combinations (*Figure 2*). Its schematic (*Figure 23a*) and its behavioral simulation (*figure 17*) validate our implementation.

In part 2-b, we are asked to design a 16-bit register file which will have 8 registers 4 of which are named with R and S, so that it will be able to execute the functionalities of the registers previously implemented. It has selection parts to choose which registers of R and S groups will be operating and also choose which registers' output will be the outputs of the register file. Corresponding selection parts were given (*figure 3*). We interpreted the $RegSel$ control input and $ScrSel$ control input and we realized that R and S group of registers are enabled according to selection bit values. Thus, we basically gave enable input of the registers with the negation of the corresponding selection input. For example, enable input of the $R1$ register is given as~$RegSel[3]$ because it is enabled when $RegSel[3]$ is low and disabled when $RegSel[3]$ is high. So that, we could implement our registers working as desired with the selection bits in a very simple way. After analyzing the simulation file and failing for the first test case, we have understood that the output wires need to be observed always although the registers work according to the rising edge of the clock signal. The reason why we should do that as far as we figured out is we should be monitoring the outputs all the time but we should not be changing which register is connected to it. Other than that we have realized from the simulation file that output wires can directly be determined by just accessing it and in accordance with the test cases even if the clock signal is not called, the output should be changed accordingly. After these realizations, we have changed our always block's sensitivity window to be a $*$ which basically means that the always block will operate whichever wire is changed in the module. After these modifications, our implementation worked correctly. Its schematic (*figure 23b*) and its behavioral simulation (*figure 18*) validate our implementation.

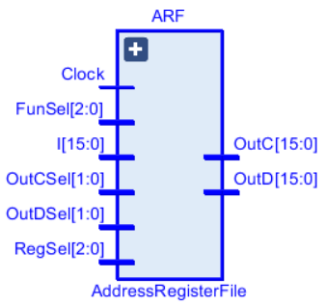
In part 2-c, we are asked to design an address register file ARF system which will consist of three 16-bit registers which are going to be used as address registers and selection bits to choose which address registers will be operated and which register's output will be the outputs of the address register file's outputs (*figure 7*). It has the same implementation logic as part 2-b which we have chosen our registers to operate according to the clock signal and we have chosen the output of the module with an $always*$ block. Its schematic (*figure 23c*) and its behavioral simulation (*figure 19*) validate our implementation.



(a) Instruction Register Schematic for Part 2A



(b) Register File Schematic for Part 2B



(c) Address Register File Schematic for Part 2C

Figure 23: Various Register Schematics

In part 3, we are asked to design an arithmetic logic unit *ALU* which can produce various mathematical operations and bit shifting operations while providing certain flags that can occur with certain operations (*figure 10*). *ALU* is basically a combinational circuit which chooses what operation to do according to selection bits and provides both the output of the operation and its necessary flags. Similar to what we have experienced in part 2-b, we realized that *ALU* operations need not be done with the clock signal though the flags need to be updated according to the clock signal. All of the operations that need to be performed in our *ALU* implementation can be easily done by using the Verilog-provided operations like negation, addition, subtraction, bit shifting, etc. However corresponding flag checks should be done carefully according to desired operations. After easily implementing the operations we started doing flag checks. Since all of the operations have a Z flag check which is a zero check, we simply dealt with it after all the cases were done. It simply checks if the output is zero or not and provides the correct flag. Nevertheless, we realized that we could not pass some cases because of the $A+B+Carry$ case since it meant to do the $A+B$ first and checks for the flag, after that it adds $carry$ to the result. We simply dealt with this operation in its case in the always block instead of dealing with it after all the cases are finished. Similarly, the N flag is updated for all the cases except for the arithmetic shift right operation, so we dealt with it separately in its case. For carry flags of the shifting operations, we were given how it should be done for each shifting operation, so we simply applied them as expected. For carry flags of the addition operations, since we need to check 17. bit for two 16-bit variables, we made use of a temporary wire with 17-bit and checked its last bit if there was a carry or not. For overflow flags, we applied the overflow rules: (positive + positive = negative, negative + negative = positive, positive - negative = negative, negative - positive = positive). For the reason that we should do the same operations with 8-bit inputs, we sign extended the given binary number treating it as if it was given as an 8-bit binary number and using its first 8 bits, so that we would be able to use the same function blocks for the 8-bit inputs. We simply check if the function selection is for an 8-bit operation, if so we sign extend the number and apply the same operations. Its schematic (*figure 24*) and its behavioral simulation (*figure 20*) validate our implementation.

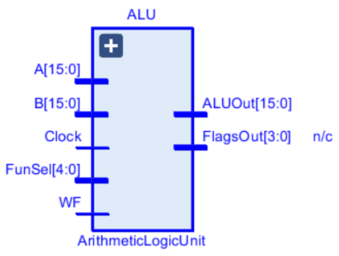


Figure 24: Arithmetic Logic Unit Schematic

In part 4, we are asked to connect all the modules we have implemented before in a system which uses the same single clock (*figure 15*). Since we are provided with which wires will be the inputs of which components, we can easily use the previous modules and connect them as desired. We are taking one clock input signal and we are sending it to all the components which work with the clock signal so that all the components use the same single clock which will avoid hazards and synchronization. We are given multiplexers in the organization figure which we did not build before, so we should implement it in this system file. A multiplexer is a device that selects between input signals and forwards the selected input to a single output line according to its select bits. Also, it should not be triggered with a clock signal because it simply selects one of the provided inputs with the selection input. After implementing the multiplexers, we need to have some wires which can be referred from the system and provide output with the clock signal. After running the simulation, we realized that there should be an address wire different than the input of the memory module. We can understand it since it is provided in the test cases and expected as an output. Its schematic (*figure 25*) and its behavioral simulation (*figure 21*) also validate our implementation.

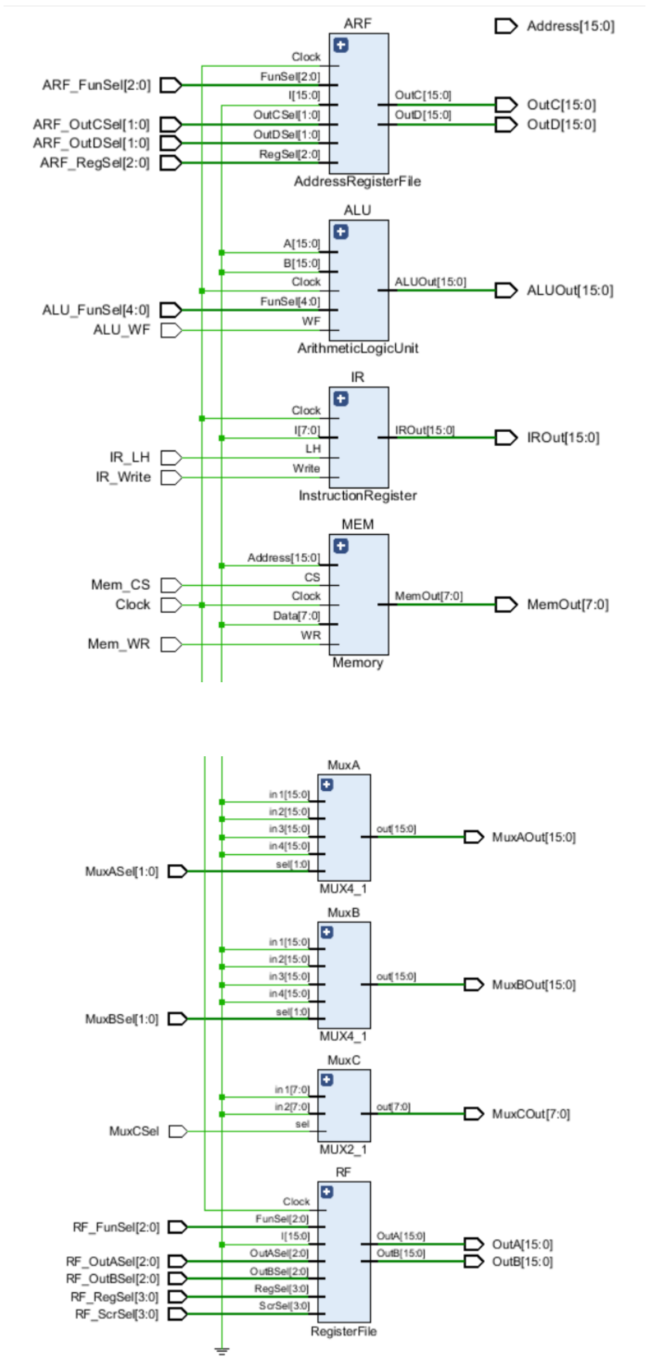


Figure 25: Arithmetic Logic Unit System

5 CONCLUSION [10 points]

We faced difficulties in the project because some variables and components weren't explained well, even though the schematics were mostly easy to understand. For example, in part 4, there was a variable called "address" which wasn't clearly defined in the project instructions or test cases. We think having more test cases would help us know exactly what to do and save time understanding our tasks and working on them. In this situation, we spend more time on understanding what we are responsible for and expected for instead of implementing the modules. Working on this project gave us a lot of experience in how to implement a hardware system and also familiarity with how to use Verilog. We understand the necessity of the clock signal in the hardware systems and apply it in our modules. Also, we experienced that generally we should give the outputs whenever it is asked for but should not do the operations all the time instead we should execute the operations with the clock signal. We grasped the idea of defining modules to be crucial as it simplifies the design process and allows us to use them in more complex designs. It helps generalize ideas and makes them reusable whenever needed. Overall, we implemented the desired project and checked it using the provided simulation files so that we ensured our implementation.

[1] [2]

REFERENCES

[1] Deniz Turgay Altılar. Computer organization: Lecture slides, 2024. Lecture slides provided by the instructor.

[2] ChipVerify. Verilog, 2024. Retrieved from ChipVerify website.