ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E

COMPUTER ORGANIZATION
PROJECT-1 REPORT

PROJECT NO : 1

PROJECT DATE : 04.04.2024

GROUP MEMBERS:

820210315 : MELİSA GÜLER

150210053 : NUREFŞAN ALTIN

SPRING 2024

Contents

1 INTRODUCTION [10 points]

The aim of this project is to design and implement various components related to registers and register files in a digital system. By dividing the project into four parts, each with specific objectives, such as understanding register design, register file implementation, ALU functionality, system integration, and modular design; the project aims to achieve comprehensive knowledge and practical experience in digital system design. The first part focuses on designing a 16-bit register with eight functionalities controlled by three-bit control signals and an enable input. The second part involves creating a register file structure, while the third part focuses on developing an Arithmetic Logic Unit (ALU) capable of performing various operations and updating flags accordingly. Lastly, in the fourth part, the project aims to implement the organizational structure of the entire system, which operates on a single clock, ensuring seamless integration and functionality of all components. Overall, this project aims to provide participants with hands-on experience in designing and implementing essential components of digital systems, preparing them for real-world applications in fields such as computer architecture, embedded systems, and digital signal processing.

2 MATERIALS AND METHODS [40 points]

2.1 Part 1

In this part, 16 bit general purpose register which has 8 functionalities is designed and implemented.

This register is designed to perform decrement, increment, load, clean and write operations.There is also a clock in the register system. The register works synchronously with this clock and performs all operations within a certain time period. The enable signal (E) acts as the main on/off switch for the circuit.

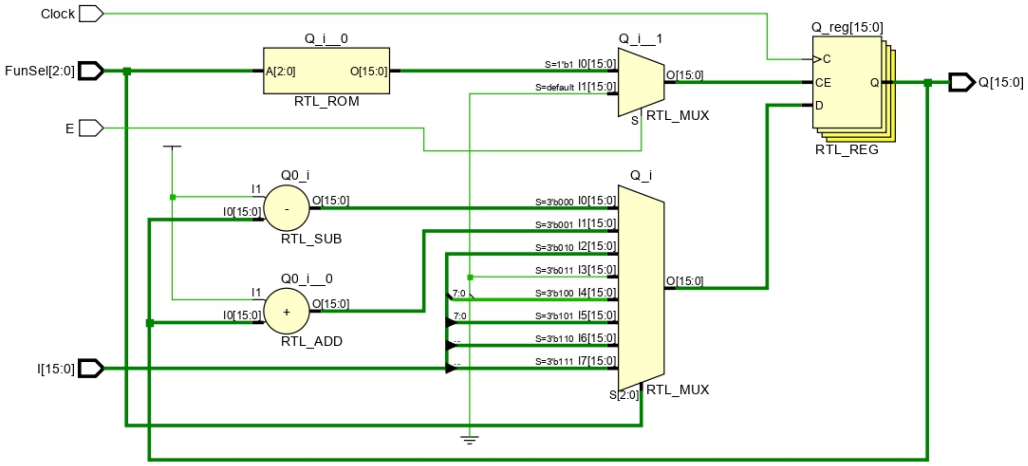

Figure 1: Register

Inputs:

I: 16-bit input to loading the register.

E: Enable input for the register. If E is high, apply the operation which is selected otherwise do nothing.

Clock: Clock signal for the register. When clock is at the rising edge, apply the operation which is selected otherwise do nothing.

FunSel: 3-bit function selection input to select which operation will be applied to the register.

000: Decrement the register by 1 /Q=Q-1

001: Increment the register by 1 /Q=Q+1

010: Load / I

011: Clear the register / Q=0

100: Load input I to the lower 8 bit and clear the upper 8 bit of register

101: Load the lower 8 bit of register

110: Load the upper 8 bit of register

111: Apply sign extension to upper 8 bit and load the lower 8 bit of register

Output:

Q: 16-bit data stored in the register.

How it works:

Always block was operated with posedge Clock to have coherent results.

An if block was used to enable the register.

A 8:3 MUX was created to choose which operation will be performed and directed to Q.

2.2 PART 2.a

16 bit Instruction Register is designed and implemented.

This instruction register circuit allows controlled storage of 16 bit data in the register. Since we only have 8 bit input, we can load only 8 bit either to the lower or upper 8 bit at once. The LH signal chooses the part of the register will be loaded, the write enable (Write) signal acts as a write permission, and the clock synchronizes the data transfer process. This design enables different part of the register to be loaded with new data under the control of the LH input and Write enable (Write) signals.

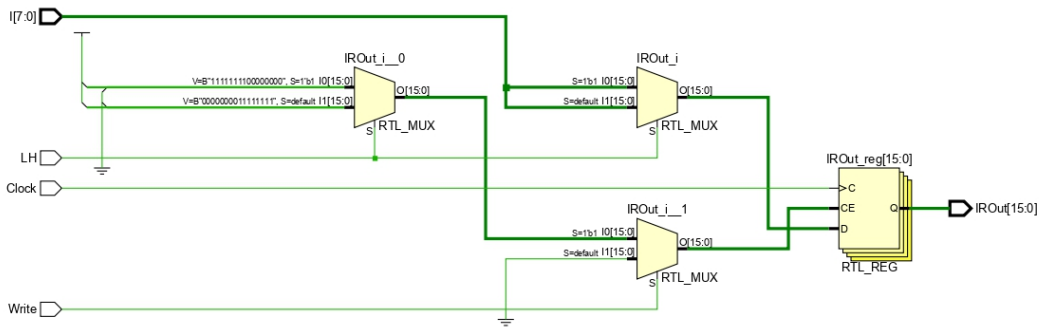

Figure 2: Instruction Register

Inputs:

Clock : Clock signal for the register. When clock is at the rising edge, apply the operation which is selected otherwise do nothing.

I : 8 bit input

LH : Chooses the part that is going to be loaded

Write : Enables the circuit

Outputs:

IROut : 16 bit output

How it works:

Always block was operated with posedge Clock to have coherent results.

An if block was created to allow write operation. In order to determine which half of the register will be loaded, another if block ,which decides according to LH, was also added.

2.3 PART 2.b

16 bit Register File is designed and implemented.

This register file circuit allows controlled writing of data into specific 16 bit registers. The RegSel and ScrSel signals chooses the registers to be enabled. The enable signal acts as a switch, controlling the participation of registers in operations and the clock synchronizes the data transfer process. This design enables multiple registers to be loaded with new data independently under the control of the RegSel, ScrSel and enable signals.

The register circuit operates by utilizing two sets of registers: R1-R4 and S1-S4. In addition, control signals OutASel and OutBSel chooses the register that are going to be directed to the OutA and OutB respectively. Together, these components create a versatile register system capable of storing, manipulating, and outputting data efficiently within the circuit.

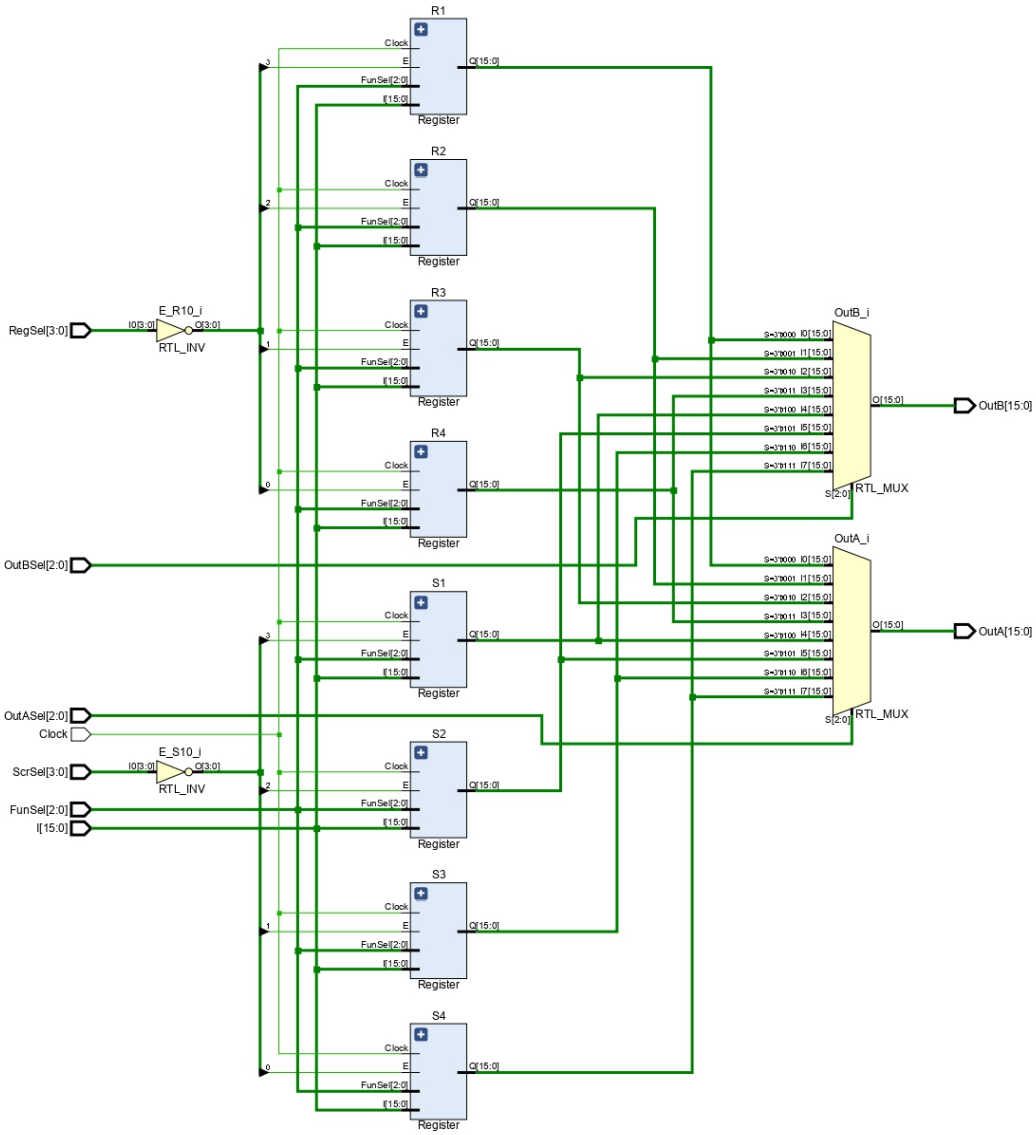

Figure 3: Register File

Inputs:

Clock : Clock signal for the general-purpose registers.

OutASel : 3 bit input to route data to the OutA

OutBSel : 3 bit input to route data to the OutB

FunSel : Chooses the function that is going to be implemented

RegSel : Enables the registers R1-R4

ScrSel : Enables the registers S1-S4

I : 16 bit input

Outputs:

OutA : 16 bit output

OutB : 16 bit output

How it works:

In this part, the register designed in the 1st part is used as 16 bit general purpose register to make the process more efficient.

8 enable wire were created to mark enabled registers. To assign their values complements of RegSel and SrcSel inputs were used.

Always block was operated with * to have coherent results and two 8:3 MUX created to direct data of OutA and OutB in this block.

2.4 PART 2.c

16 bit Address Register File is designed and implemented.

This address register file design enables multiple address registers to be loaded with new data after applying different functionalities independently. It works under the control of the RegSel and enable signals specified for each register.

The register circuit can operate 3 registers: program counter (PC), address register (AR), and stack pointer (SP). The RegSel signal decides which of these will be enabled.

In addition, the control signals OutCSel and OutDSel act as selectors, allowing the circuit to choose correct register's data to direct to the output lines OutC and OutD.

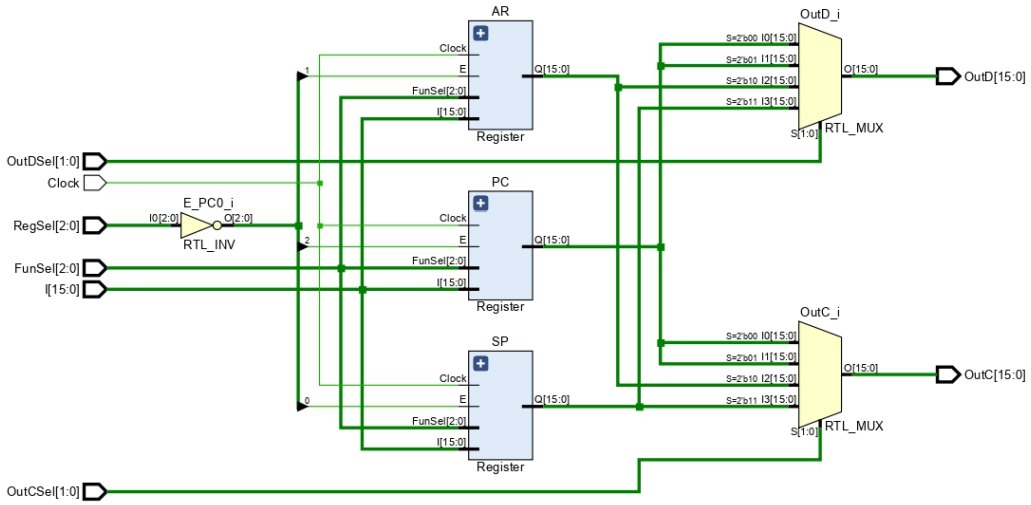

Figure 4: Address Register

Inputs:

Clock: Clock signal for the register. When clock is at the rising edge, apply the operation which is selected otherwise do nothing.

I : 16 bit input

OutCSel : 2 bit input to choose routed the data to the OutC

OutDSel : 2 bit input to choose routed the data to the OutD

FunSel : Chooses the function that is going to be implemented

RegSel : Enables the registers PC, AR and SP

Outputs:

OutC : 16 bit input

OutD : 16 bit input

How it works:

The register designed in the 1st part is also used in this part as 16 bit general purpose register to make the process more efficient.

3 enable wire were created to mark enabled registers. To assign their values complements of RegSel input is used.

Always block was operated with * to have coherent results and two 4:2 MUX created to direct data of OutC and OutD in this block.

2.5 PART 3

In this part Arithmetic Logic Unit (ALU) was designed and implemented.

This ALU performs 32 different arithmetic and logical operations on different size data. It also stores 4 flags which are zero, negative, carry, and overflow to obtain meaningful results from these operations.

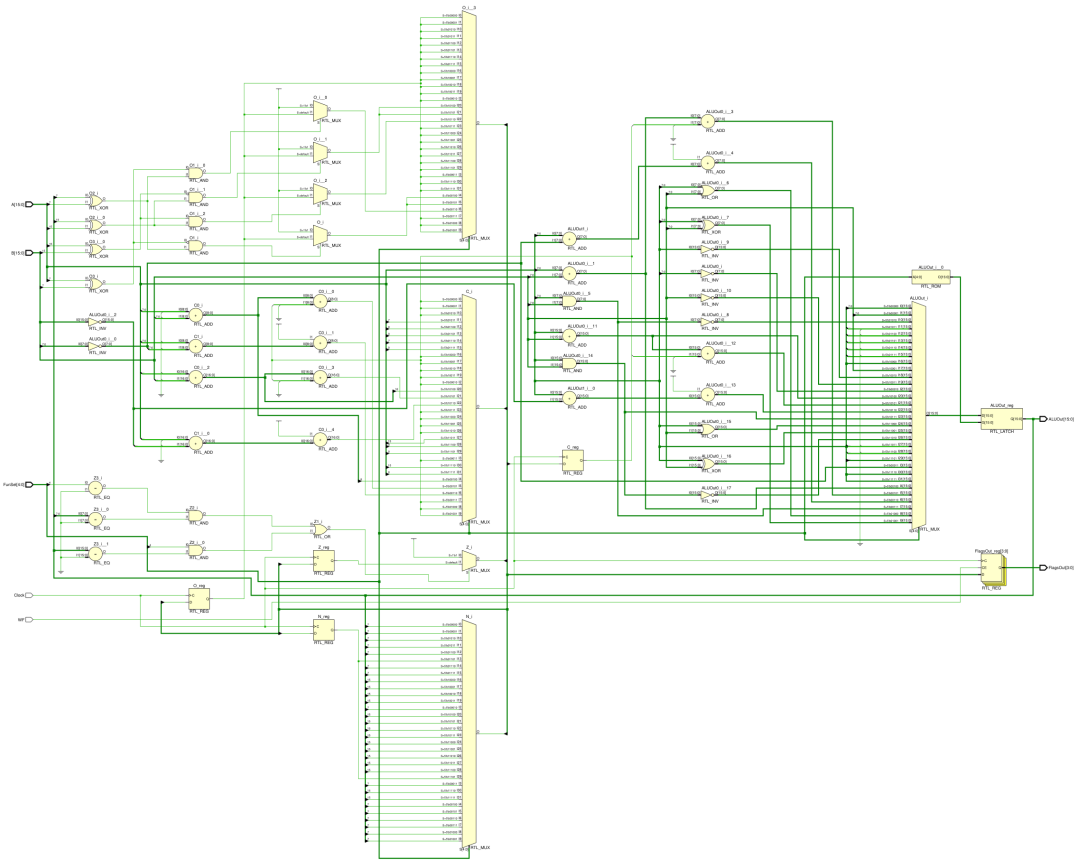

Figure 5: ALU

Inputs:

A: 16 bit input register

B: 16 bit input register

FunSel: 5 bit selection input of ALU. We use it for choosing the function that is going to be implemented in ALU. Since it is 5 bit, we have 32 differerent arithmetic and logical functionalities in this ALU. 16 of them will be carried out on 8 bit numbers and 16 of them will be carried out on 16 bit numbers.

WF: input for the ALU's flag register. We use it to check when we will update flags. When WF is 1 flags will be updated.

Clock: Clock signal for the ALU. When clock is at the rising edge, apply the operation which is seclected otherwise do nothing.

Internal elements:

We used some internal components to increase efficiency in the code:

Z, C, N and O: They are used to store flag's next values. We will use them when flags need to be changed.

Internal result: It is used to determine the value of C(carry) when we are applying addition, addition with carry and subtraction operations. We store temporarily the results of these operations in this register.

Outputs:

ALUOut: 16 bit output

FlagsOut: 4 bit register to store zero, negative, carry, and overflow flags.

How it Works:

We wrote 2 seperate always blocks here, one is to write the result of the operation to the ALUOut and the other one is to decide and update flag variables. For ALUOut, we use * in the sensitivity window because it should update the ALUOut immediately anytime it changes. However, we use posedge Clock to update FlagsOut because it should be updated after the clock pulse for correctness of our results.

For 8 bit operations:

We decide the sign according to the element in the 7th index of ALUOut.

We determine the carry by looking at the 8th index of the result.

For 16 bit operations:

We decide the sign according to the element in the 15th index of ALUOut.

We determine the carry by looking at the 16th index of the result.

Since we update Z after each operation, we assigned its value at once at the end of the block.

For overflow we have 4 different cases in total. 2 of them occur in addition and 2 of them occur in subtraction. Here we checked the signs of operands and the output. For the addition case, if both operand have same sign but different than the output than there will be overflow. For the subtraction case, if operand's signs are different and the sign of the 1st operand is different than the result's sign than there will be overflow.

In the end, we assign Z,C,N,O values to the FlagsOut register if write flag WF=1.

2.6 PART 4

I this part, we implemented an organization which consist of all the units we designed previous parts. This organization consist of the following units:

Register, was designed in Part-1.

Instruction Register, was designed in Part-2a.

Register File, was designed in Part-2b.

Address Register File, was designed in Part-2c.

Arithmetic Logic Unit(ALU), was designed in Part-3.

Memory, was received from pre-prepared design of logisim.

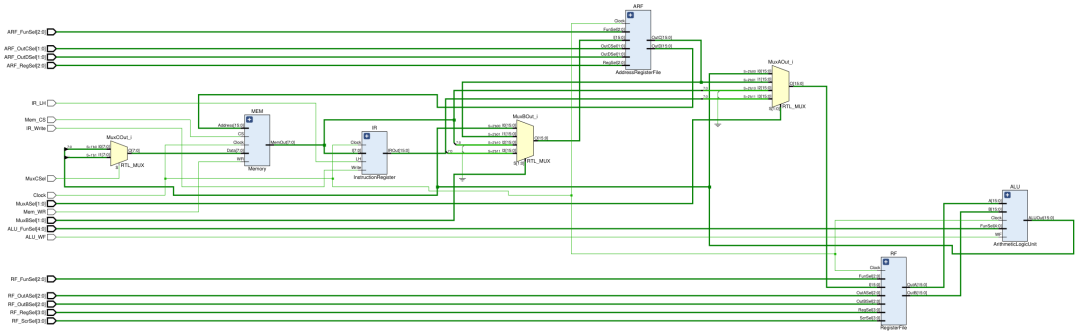

Figure 6: ALU System

Inputs:

Inputs-Outputs of Register File:

RF OutASel : 3 bit input

RF OutBSel : 3 bit input

RF FunSel : 3 bit input

RF RegSel : 4 bit input

RF ScrSel : 4 bit input

OutA : 16 bit output

OutB : 16 bit output

Inputs-Outputs of Address Register File:

ARF FunSel : 3 bit input

ARF OutCSel : 2 bit input

ARF OutDSel : 2 bit input

ARF RegSel : 3 bit input

OutC : 16 bit output

Address/ OutD : 16 bit output

Inputs-Outputs of ArithmeticLogic Unit:

ALU FunSel : 5 bit selection input

ALU WF : Write Flag

ALUOut : 16 bit output

FlagsOut : 4 bit output to store flags

Inputs-Outputs of Instruction Register:

MemOut

IR LH

IR Write : Enable of IR

IROut

Inputs-Outputs of Memory:

Address/ OutD : 16 bit input

Mem WR

Mem CS

MemOut : 8 bit output

Inputs-Outputs of MUXA:

ALUOut

OutC

MemOut

IROut(7-0)

MuxASel

Inputs-Outputs of MUXB:

ALUOut

OutC

MemOut

IROut(7-0)

MuxBSel

Inputs-Outputs of MUXC:

ALUOut(7-0)

ALUOut(15-8)

MuxCSel

How it Works:

Clock : When a rising edge is detected on the clock signal, the circuit activates. The circuit uses one single clock signal to schnronize everything.

We also designed 3 more components to construct connections between inputs and outputs of these units:

MUX A, is utilized to choose the input of the Register File. The ALU's output(ALUOut) is linked to the 0th index, one output of Address Register File which is labeled as OutC is connected to the 1st index, the memory's output(MemOut) is connected to the 2nd index and the least significant 8 bits of the Instruction Register(IROut[7:0]) is connected to the 3rd index of a 4:2 multiplexer as inputs.

MuxASel forms a 2 bit select input to choose one of the inputs of MUX A as the output MuxAOut.

MUX B, is employed to select the input of the Address Register File. The ALU's output(ALUOut) is directed to the 0th index, one output of Address Register File which is labeled as OutC is connected to the 1st index, the memory's output(MemOut) to the 2nd index and the least significant 8 bits of the Instruction Register(IROut[7:0]) is connected to the 3rd index of a 4:2 multiplexer as inputs.

MuxBSel forms a 2 bit select input to choose one of the inputs of MUX B as the output MuxBOut.

MUX C, is used to direct either lower 8 bit or higher 8 bit of ALU output (ALUOut) to the input of Memory module.

MuxCSel forms a 1 bit select input to choose one of the inputs of MUX C as the output MuxCOut.

3 RESULTS [15 points]

3.1 PART 1

Figure 7: Simulation of Register

3.2 PART 2.a

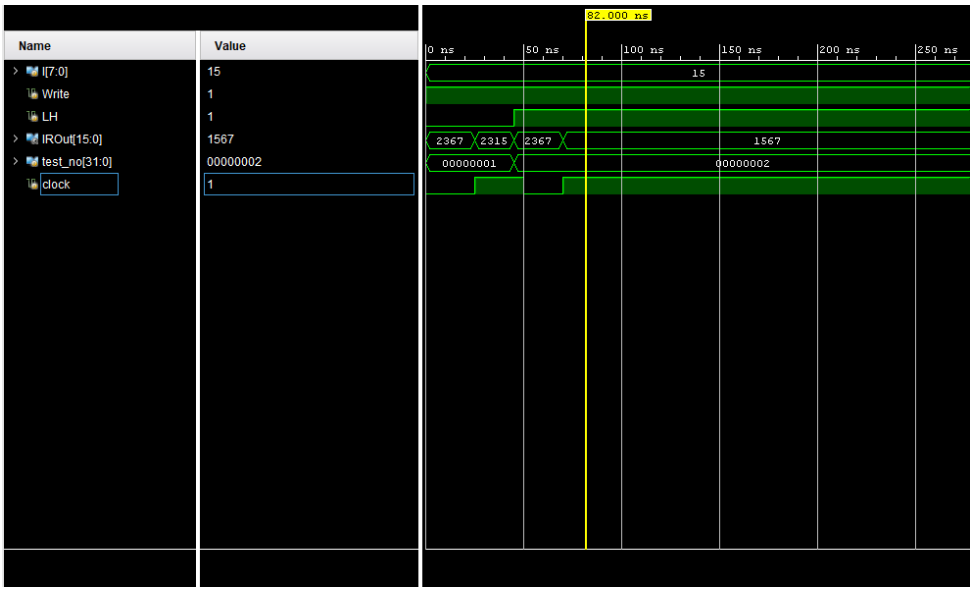

Figure 8: Simulation of Instruction Register

3.3 PART 2.b

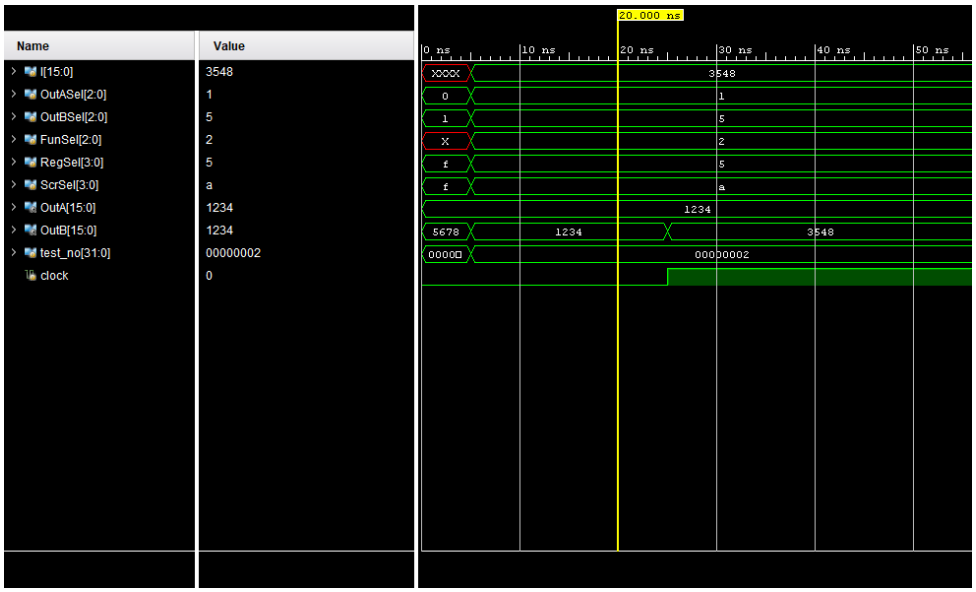

Figure 9: Simulation of Register File

3.4 PART 2.c

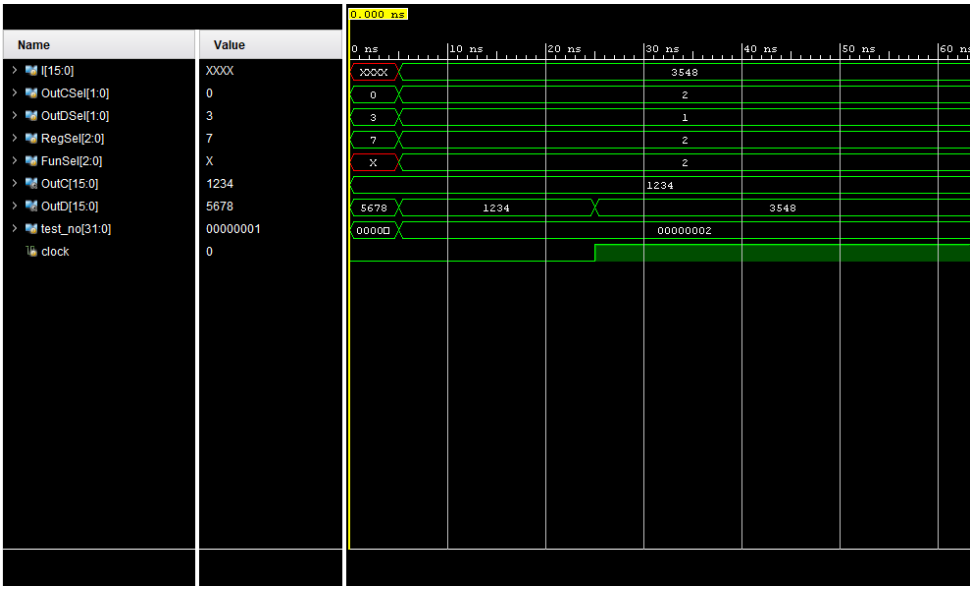

Figure 10: Simulation of Address Register

3.5 PART 3

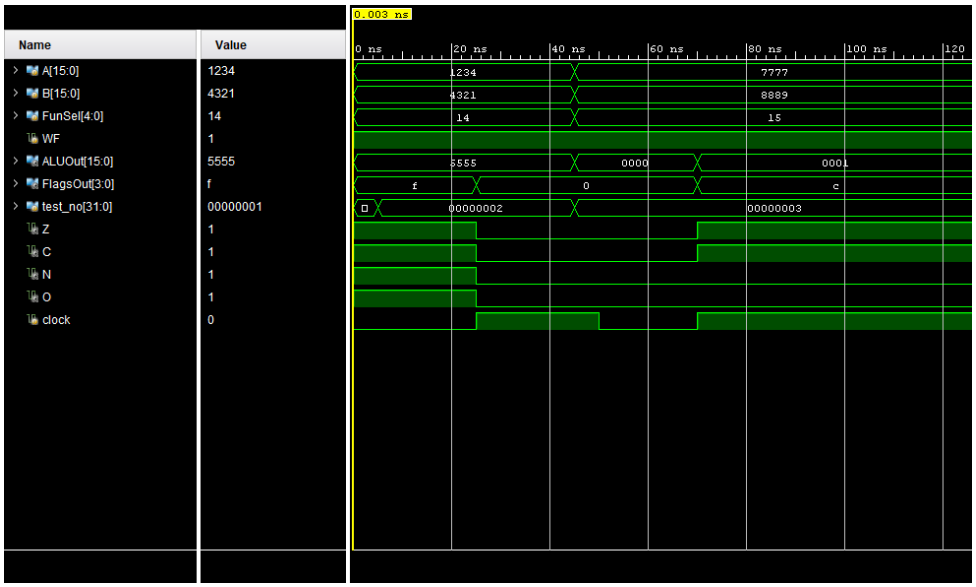

Figure 11: Simulation of ALU

3.6 PART 4

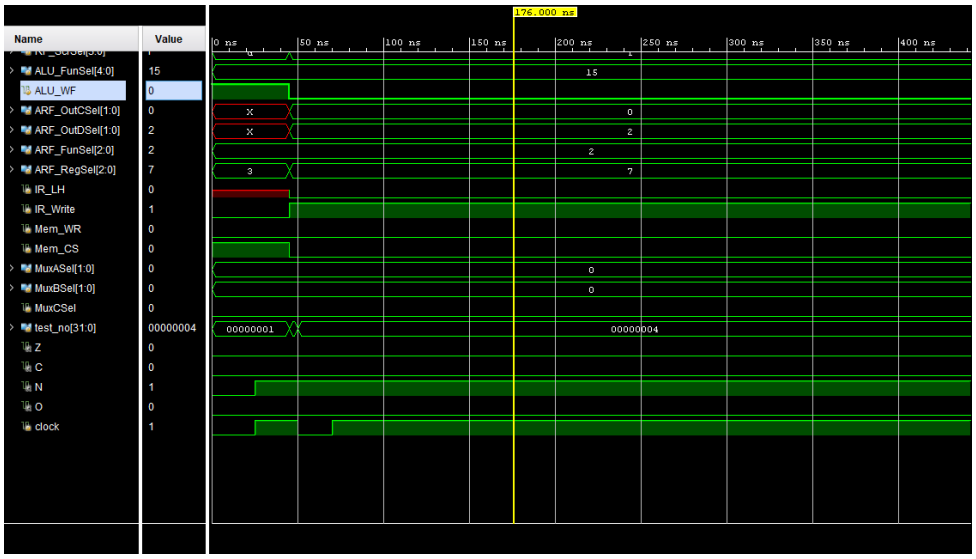

Figure 12: Simulation of ALU System

4 DISCUSSION [25 points]

4.1 PART 1

In this part we focused on how to develop a 16-bit register that can perform 8 different operations based on control signals and an enable input.
This design offers a versatile building block for various system operations and serves as a foundation for implementations in later sections of our project.
These kind of registers allows for data storage, manipulation, and control based on external signals. By combining multiple of them with specific control logic, you can create more complex structures like register files and control units used in processors.

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 13: Characteristic table of the Register

E represents Enable, which determines when the register will carry out the function selected by FunSel. Accordingly the function will be performed if E is 1 and will not be performed when it is 0.
FunSel is selection input for choosing the functions we will operate on given inputs and outputs. It is 3-bit binary value which corresponds to different operations that the register can perform.
Q+ refers to the next output of register after the selected operation is implemented. It depends on the current state and also given inputs.
If E = 0, the register retains its value (no change).
If E = 1, the FunSel value decides what happens in the register:
FunSel = 000 (Decrement): A binary 1 is subtracted from the value stored in the register. This essentially counts down.

FunSel = 001 (Increment): A binary 1 is added to the value stored in the register. This essentially counts up.
FunSel = 010 (Load): The data present on the I input is loaded into the register, overwriting the previous value.
FunSel = 011 (Clear): All bits in the register are set to 0.
FunSel = 100 : The upper 8 bit[15-8] are set to 0 and the lower 8 bit are loaded with new 8-bit data input I[7:0].
FunSel = 101 (Only Write Low) : The lower 8 bit [7-0] of register is loaded with new 8-bit data input I[7:0].
FunSel = 110 (Only Write High) : The upper 8 bit [15-8] of register is loaded with new 8-bit data input I[7:0].
FunSel = 111 : Sign extension is applied to the upper 8 bit [15-8] of register according to the value in the 7th index of input and load the lower 8 bit [7-0] with new 8-bit data input I[7:0].

Also our register works synchronously with a clock signal and perform all operations that it can carry out compatible with the time. We used posedge clock, which refers to rising edge of the signal, in always block so it updates the state value when the signal transitions from low (0) to high (1) .

4.2 PART 2a

The main topic of part 2-a is the design of a special 16-bit register called the Instruction Register (IR).

This register is used to store instruction codes for the processor.It can hold 16 bits of data, but it can only receive data 8 bits at a time. A control signal named L'H determines whether the loaded 8-bit data goes to the lower or higher half of the register. A Write was included to decide whether the Instruction register performs an operation or not. When Write is 1, it will be active and operations can be performed. But when Write is 0, it will be unavailable for operations. This allows for efficient loading of instructions that might be longer than 8 bits.

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 14: IR register's characteristic table

Ø: This symbol means "don't care". If write is 0, it doesn't matter what the value of L'H. The IR will output the current value in this case.

When Write = 1:

LH=0 : Load the LSB(7-0) of register

LH=1 : Load the MSB(15-8) of register

4.3 PART 2b

In this part, we designed a register file that consists of two sets of 16-bit registers:

There is 4 general-purpose registers (R1, R2, R3, R4) which were designed for general usage and 4 scratch registers (S1, S2, S3, S4) which were designed for temporary data storage. Additionally two 4-bit control signals, RegSel and ScrSel were created. These signals act like decoder keys. By applying a specific binary value to RegSel, one of the four general-purpose registers (R1-R4) can be selected to be involved in an operation. Similarly, a specific binary value on ScrSel selects one of the scratch registers (S1-S4).

8 enable signal (E) also included to control whether the general-purpose and scratch registers participate in operations. When E is high (1), it activates the register selected by RegSel and ScrSel. But when E is low (0), it essentially powers down that register, making it unavailable for operations. This ensures only registers that are needed will be used at a specific time.

E is a kind of switch for the registers. Even if a register is selected by RegSel or ScrSel, it won't be involved in operations unless E is turned on.

Two more control signals were designed, OutASel and OutBSel, to select which registers send their data to two output lines, OutA and OutB. These signals act like multiplexers. Depending on the binary value applied to each signal, which register's data gets routed to the corresponding output line can be selected.

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 15: OutASel and OutBSel controls.

This table shows how the control signals OutASel and OutBSel determine which register outputs data on the OutA and OutB lines.

OutASel/OutBSel: These are the two control signals (each 2 bits) that select which register will output data. R1-R4: These represent the four general-purpose registers. S1-S4: These represent the four scratch registers.

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 16: FunSel Control Input

This table shows what operations will be carried out in registers according to the FunSel's value.

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 17: RegSel Control Input

This table is related to how the RegSel affect the general-purpose registers (R1-R4).

RegSel: This is a 4-bit register select signal that can select one of the four general-purpose registers (R1-R4).

R1, R2, R3, R4's enable values are related with the complement of the value at the corresponding index (3,2,1,0) of RegSel.

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 18: ScrSel Control Input

This table is related to how the ScrSel affect the scratch registers (S1-S4).

ScrSel: This is a 4-bit control signal that selects one of the scratch registers (S1-S4).

S1, S2, S3, S4's enable values are related with the complement of the value at the corresponding index (3,2,1,0) of ScrSel.

4.4 PART 2c

In this part, we designed a address register file that consists of 3 sets of 16-bit registers:

There are Program Counter register, Address Register and, Stack Pointer register.

Additionally a 3-bit control signal, RegSel was created. This signal act like decoder keys. By applying a specific binary value to RegSel, one of the 3 address registers can be selected to be involved in an operation.

3 enable signal specified for each of these registers also included to control whether the address registers participate in operations. When E is high (1), it activates the registers selected by RegSel. But when E is low (0), it essentially powers down that register, making it unavailable for operations. This ensures only registers that are needed will be used at a specific time.

Two more control signals were designed, OutCSel and OutDSel, to select which registers send their data to two output lines, OutC and OutD. These signals act like multiplexers. Depending on the binary value applied to each signal, which register's data gets routed to the corresponding output line can be selected.

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 19: RegSel Control Input

This table is related to how the enable signal and the register select signal (RegSel) affect the address registers.

RegSel: This is a 3-bit register select signal that can select any of the 3 address registers (PC,AR,SP).

PC: Program Counter register.

AR: Address Register.

SP: Stack Pointer register.

OutCSel	OutC
00	PC
01	PC
10	AR
11	SP

OutDSel	OutD
00	PC
01	PC
10	AR
11	SP

Figure 20: OutCSel and OutDSel controls.

This table shows how 2 bit control signals OutCSel and OutDSel select which register outputs data on the OutC and OutD lines, respectively.

4.5 PART 3

In this part of the project, we focused on the Arithmetic Logic Unit (ALU).

This unit performs various arithmetic and logical operations on two 16-bit data inputs, labeled A and B. It also generates flags to indicate the outcomes of the operations. These flags are:

Zero (Z): This flag is set to 1 if the ALU output is zero.

Negative (N): This flag is set to 1 if the ALU output is a negative number.

Carry (C): This flag is set to 1 if there's a carry out from the most significant bit during the operation.

Overflow (O): This flag is set to 1 if an overflow occurs during the operation.

The specific operation performed by the ALU is determined by the FunSel (Function Select) which is 5-bit binary value, as detailed in the provided table. We connected the 16-bit data inputs (A and B) to the ALU data path. The FunSel needs to be connected to the control unit. We also connected the 16-bit output ALUOut to the output port. In addition, since the flags (Z, N, C, O) need to be stored for future use, we created the output register FlagsOut to save them. Also, we defined a write flag (WF) to control when we will update the FlagOut. Finally, the flag register's(FlagsOut) output is connected to the ALU'S input port since we will use it again in next operations.

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 21: Characteristic table of ALU.

This table defines operations will be performed in ALU according to the FunSel's value. There is 32 different operations, 16 of them applied on 8-bit numbers and the remaining 16 are applied on 16-bit numbers. There is also a column that shows which flags will be updated after performing that spesific operation(+ and - stands for the ones we need to update and stands for the ones will remain unchanged respectively).

According to FunSel different operations will be carried out:

00000 A (8-bit) : lower 8 bit of ALUOut is loaded with lower 8 bit of A

Updated Flags = Z and N

00001 B (8-bit) : lower 8 bit of ALUOut is loaded with lower 8 bit of B

Updated Flags = Z and N

00010 NOT A (8-bit) : lower 8 bit of ALUOut is loaded with complement of lower 8 bit of A

Updated Flags = Z and N

00011 NOT B (8-bit) : lower 8 bit of ALUOut is loaded with complement of lower 8 bit of B

Updated Flags = Z and N

00100 A + B (8-bit) : lower 8 bit of ALUOut is loaded with the sum of lower 8 bit of A and lower 8 bit of B

Updated Flags = Z, C, N and O

00101 A + B + Carry (8-bit) : lower 8 bit of ALUOut is loaded with the sum of lower 8 bit of A, lower 8 bit of B and carry bit , which comes from previous steps

Updated Flags = Z, C, N and O

00110 A – B (8-bit) : lower 8 bit of ALUOut is loaded by subtracting lower 8 bit of B from lower 8 bit of A

Updated Flags = Z, C, N and O

00111 A AND B (8-bit) : lower 8 bit of ALUOut is loaded with the result of AND operation applied on lower 8 bit of A and lower 8 bit of B

Updated Flags = Z and N

01000 A OR B (8-bit) : lower 8 bit of ALUOut is loaded with the result of OR operation applied on lower 8 bit of A and lower 8 bit of B

Updated Flags = Z and N

01001 A XOR B (8-bit) : lower 8 bit of ALUOut is loaded with the result of XOR operation applied on lower 8 bit of A and lower 8 bit of B

Updated Flags = Z and N

01010 A NAND B (8-bit) : lower 8 bit of ALUOut is loaded with the result of NAND(not and) operation applied on lower 8 bit of A and lower 8 bit of B

Updated Flags = Z and N

01011 LSL A (8-bit) : lower 8 bit of ALUOut is loaded with the value obtained by appliyng Logical Shift Left operation on A

Updated Flags = Z,C and N

01100 LSR A (8-bit) : lower 8 bit of ALUOut is loaded with the value obtained by appliyng Logical Shift Right operation on A

Updated Flags = Z,C and N

01101 ASR A (8-bit) : lower 8 bit of ALUOut is loaded with the value obtained by appliyng Arithmetic Shift Right operation on A

Updated Flags = Z and C

01110 CSL A (8-bit) : lower 8 bit of ALUOut is loaded with the value obtained by appliyng Circular Shift Left operation on A

Updated Flags = Z,C and N

01111 CSR A (8-bit) : lower 8 bit of ALUOut is loaded with the value obtained by appliyng Circular Shift Left operation on A

Updated Flags = Z,C and N

10000 A (16-bit) : ALUOut is loaded with A

Updated Flags = Z and N

10001 B (16-bit) : ALUOut is loaded with B

Updated Flags = Z and N

10010 NOT A (16-bit) : ALUOut is loaded with complement of A

Updated Flags = Z and N

10011 NOT B (16-bit) : ALUOut is loaded with complement of B

Updated Flags = Z and N

10100 A + B (16-bit) : ALUOut is loaded with the sum of A and B

Updated Flags = Z, C, N and O

10101 A + B + Carry (16-bit) : ALUOut is loaded with the sum of A, B and carry bit, which comes from previous steps

Updated Flags = Z, C, N and O

10110 A – B (16-bit) : ALUOut is loaded by subtracting B from A

Updated Flags = Z, C, N and O

10111 A AND B (16-bit) : ALUOut is loaded with the result of AND operation applied on A and B

Updated Flags = Z and N

11000 A OR B (16-bit) : ALUOut is loaded with the result of OR operation applied on A and B

Updated Flags = Z and N

11001 A XOR B (16-bit) : ALUOut is loaded with the result of XOR operation applied on A and B

Updated Flags = Z and N

11010 A NAND B (16-bit) : ALUOut is loaded with the result of NAND(not and) operation applied on A and B

Updated Flags = Z and N

11011 LSL A (16-bit) : ALUOut is loaded with the value obtained by appliyng Logical Shift Left operation on A

Updated Flags = Z,C and N

11100 LSR A (16-bit) : lower 8 bit of ALUOut is loaded with the value obtained by appliyng Logical Shift Right operation on A

Updated Flags = Z,C and N

11101 ASR A (16-bit) : ALUOut is loaded with the value obtained by appliyng Arithmetic Shift Right operation on A

Updated Flags = Z and C

11110 CSL A (16-bit) : ALUOut is loaded with the value obtained by appliyng Circular Shift Left operation on A

Updated Flags = Z,C and N

11111 CSR A (16-bit) : ALUOut is loaded with the value obtained by appliyng Circular Shift Left operation on A

Updated Flags = Z,C and N

4.6 PART 4

In this part, we took all the individual modules we designed previously and integrated them into a cohesive system. This system represents the core building block of our processor. We've incorporated several registers from previous parts, including:

A 16-bit register for general-purpose use (from part 1). An instruction register (from part 2-a). A register file containing four general-purpose registers (R1-R4) and four scratch registers (S1-S4) designed in part 2-b. The address register file (ARF) containing the program counter (PC), address register (AR), and stack pointer (SP) from part 2-c. An ALU that was designed in part 3. This unit performs various arithmetic and logical operations (addition, subtraction, AND, OR, shifting etc.) on two 16-bit inputs. It also produces flags indicating the outcome of the operation (zero, negative, carry, overflow).

Throughout the design process, we created various control signals to manage different functionalities. These signals are represented by the lines with arrows pointing in various directions. They include enable, register select, output select, function select and clock.

Enable (E), controls whether a register participates in an operation.Register Select (RegSel, ScrSel) selects which general-purpose or scratch register is involved in an operation. Output Select (OutASel, OutBSel, OutCSel, OutDSel) determines which registers send their data to specific output lines. Function Select (FunSel), chooses the specific operation for the ALU to perform. Clock, the entire system operates synchronously based on a single clock signal. This clock ensures all the components perform their operations in a coordinated fashion. The specific operation of this system depends on the instructions

being processed and the accompanying control signals. The processor fetches an instruction from memory and stores it in a register.The instruction is decoded to determine the operation to be performed and the registers to be involved. This likely involves the control signals interacting with the instruction register.Based on the control signals, data may be transferred from registers (general-purpose, scratch, or address register file) to the ALU or other destinations.If the instruction requires an arithmetic or logical operation, the ALU performs it on the provided data and updates the flags based on the result.The outcome of the operation (or data from a register) may be written back to a register or memory, again based on the control signals.By working together, these components and control signals form the foundation of our processor, allowing it to execute instructions, perform calculations, and manipulate data. This is a significant step towards building a functional computer system.

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

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

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

Figure 22: Multiplexers of Arithmetic Logic Unit Systems

This table shows how control signals MuxASel, MuxBSel and MuxCSel select which output data will be on the MuxAOut, MuxBOut and MuxCOut lines.

5 CONCLUSION [10 points]

Despite facing obstacles throughout the project, particularly in navigating Verilog, where even simple tasks could lead to complex errors, we found that Parts 1 and 2 served as valuable preparation for the more challenging aspects of the project. Part 3 proved to be the most demanding, given its intricacies in handling both 8-bit and 16-bit operations within the ALU. We did not face any difficulties until the ALU phase of our project, but there were some confusing aspects when we reached that part. We learned how to address these issues through the question-answer sessions on Ninova, and were able to complete our operations.However, through diligent testing, we managed to identify and rectify errors, leading to successful outcomes. Part 4 introduced us to the use of memory and clock, offering a fresh perspective on system design.We used the 8-bit files uploaded while completing the assignment. The inclusion of tables and diagrams in the project files greatly aided our understanding of the assignment. Despite these challenges, we emerged with a wealth of experience in hardware description language (Verilog) and hardware system design.

REFERENCES