ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E
COMPUTER ORGANIZATION
PROJECT 1 REPORT

CRN : 21334

LECTURER : Assoc. Prof. Dr. Gökhan İnce

GROUP MEMBERS:

150210928 : Racha Badreddine

150220061 : Melike Beşparmak

SPRING 2024

CONTENTS

Contents

1 INTRODUCTION [10 points]

In this project, we used Verilog HDL (Hardware Description Language) in the Vivado environment to implement registers and register-based designs.

Initially, we created a versatile 16-bit general-purpose register capable of executing 8 different functionalities based on the Function Selector (FunSel) input.

Next, we implemented a 16-bit Instruction Register (IR), followed by constructing a register file (RF) with 4 general-purpose and 4 scratch registers, and an Address Register File (ARF) containing Program Counter (PC), Address Register (AR), and Stack Pointer (SP) registers.

Subsequently, we designed and implemented an Arithmetic Logic Unit (ALU), capable of performing 16 distinct arithmetic and logic operations on both 8 and 16-bit data, determined by the FunSel.

Finally, we integrated all modules into an Arithmetic Logic Unit System (ALUSys).

1.1 Task Distribution

While working on this project, we ensured that both of us could understand and implement the provided designs. To achieve this, we met regularly whenever we needed to work on the project, collaborating to develop all parts together and to overcome all the obstacles we faced.

2 MATERIALS AND METHODS [40 points]

2.1 Part 1: 16-Bit Register

In part 1, a 16-bit register was designed based on the table provided. E is the enable input, and the register will only perform the selected function if E equals 1. The second input is the 3-bit FunSel which selects the function out of 8 possibilities. Additionally, since registers are sequential circuits, the clock signal is provided as an input during implementation. The Register operates specifically on the positive edge of the clock signal.

E	FunSel	Q+
0	Dont care	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) ← 0, 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

Table 1: Input and Output Table of Register

2.2 Part 2

2.2.1 a: 16-Bit Instruction Register

In this part, a 16-bit IR was designed. The IR operates on either the least significant (LSB) 8 bits or the most significant (MSB) 8 bits. L'H input is used to control the bits to be loaded as least significant (7-0) if it is 0, or most significant (15-8) if it is 1. The write flag works similarly to an enable signal. If it is 0, the IR will retain its value and will not load any data.

L'H	Write	IR+
X	0	IR (Retain Value)
0	1	IR (7-0) ← I (Load LSB)
1	1	IR (15-8) ← I (Load MSB)

Table 2: Input and Output Table of IR

2.2.2 b: Register File System

In this section, an RF was implemented using the registers designed in the first part. This system consists of 8 registers, divided into four general-purpose registers and four scratch registers. Additionally, it has 16-bit input data, two 16-bit outputs labeled A and B, and 5 selectors. Among these selectors, three are 3-bit selectors: OutASel and OutBSel determine the register to write to each output, while Funsel selects the function to be executed by the enabled registers. The remaining two selectors are 4-bit selectors, RegSel and ScrSel. They are used to enable specific registers based on the complement of the given data. For instance, when RegSel is 0001, the first 3 registers corresponding to 0 bits are going to be enabled and R4 will retain its value.

FunSel	R_x+
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)

Table 3: Input and Output Table of RF

2.2.3 c: Address Register File

Similarly to the previous part, another register file was implemented with more specificity. This time, an ARF was created using the registers designed in the first part. It comprises 3 registers, PC, AR, and SP. It has 16-bit input data, two 16-bit outputs labeled C and D along with 4 selectors. Two of them are 3-bit selectors -OutCSel and OutDSel- used to determine the registers to write to the outputs. In addition to two 3-bit selectors, FunSel selects the function to be executed, and RegSel enables specific registers to perform the operation depending on its complement.

2.3 Part 3: Arithmetic Logic Unit

At this stage of the project, we implemented an ALU that performs the operations provided in the table. The ALU accepts two 16-bit inputs A and B along with carry-in input and FunSel. Its output is a 16-bit data and a 4-bit flags to be stored in a register.

The FunSel is a 5-bit data, according to its most significant bit the ALU performs operation on either 8-bit or 16-bit input data. If the most significant bit is 0, the operations will be performed on 8-bit data then apply sign extension depending on the sign to produce the 16-bit output. However, if it is 1 all the operations will be performed on 16-bit data to provide the output.

Since the operations are similar for both data sizes, we used a set of 16 cases to cover all arithmetic and logic operations. To achieve this, we created an additional variable (integer) that indicates the sign bit, 7th or 15th bit depending on the data size. This allowed us to generalize our approach effectively. However, in some cases, specifically the arithmetic ones, we were obliged to operate in two stages. For instance, addition is initially performed on 8 bits. If the most significant bit of FunSel is 1, the operation continues on the remaining bits. This condition is used to update the carry flag correctly.

Moreover, we have 4 flags, Zero (Z), Carry (C), Negative (N), and Overflow (O), to be updated each time an operation is executed, then depending on both the arrival of the positive edge of the clock signal and the write flag, they are stored in the 4-bit Flags Register. The Z flag is 1 when the ALU output is zero, it is checked for all the operations performed. The C flag is 1 when we have a carry resulted from the operation executed by the unit or when we have a borrow in the subtraction operation (No extra bit), this flag is checked only after the arithmetic and shift operations. The N flag is 1 when the ALU output is a negative data which means when the sign bit is equal to 1. This flag is checked after all the operations except the arithmetic shift since it remains unchanged by this operation. Lastly, the O flag is specific to arithmetic operations and indicates whether the result can be represented within the given data size for signed numbers. If not, the flag is set to 1.

The C_{in} input is taken from the Flags register, which means that it takes the current stored value in that register.

2.4 Part 4: ALUSys

In the last part of the project, we were asked to implement the system given which basically consists of all modules and designs implemented before in addition to the memory module and RAM file given. Firstly, we implemented two multiplexer modules. The first multiplexer accepts two 16-bit and two extended 8-bit inputs (0 extension for memory output and sign extension for fourth input), as required. The output of this multiplexer is 16-bit data. It was used to represent the MuxA and MuxB in the system. The second multiplexer, representing MuxC in the design, accepts a 16-bit input, then depending on its selector, it outputs either the 8 most significant or least significant bits of the input data.

In this part, by using some extra wires as necessary to establish the correct connection and providing the appropriate parameters to the modules, we could successfully implement the ALU System.

3 RESULTS [15 points]

The schematics of the implementations were generated by Vivado itself. While simulations were conducted, they were found insufficient to evaluate most cases. Therefore, we created additional test cases to ensure comprehensive coverage for all modules. The results are as follows.

3.1 Part 1: 16-Bit Register

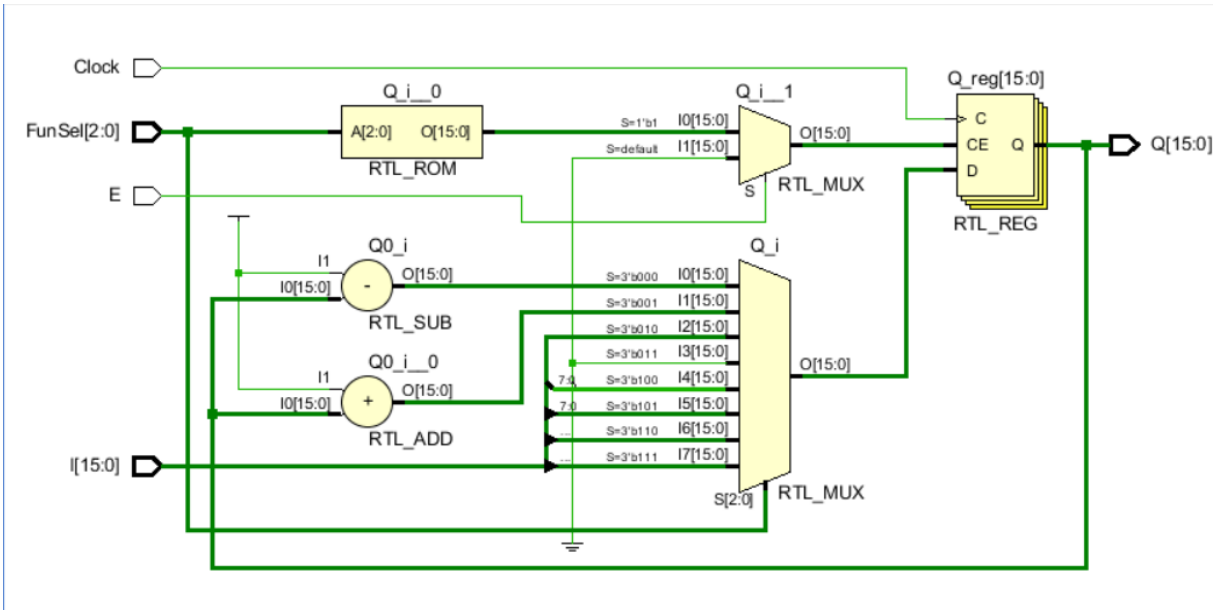

Figure 1: Schematic of Part -1- Register

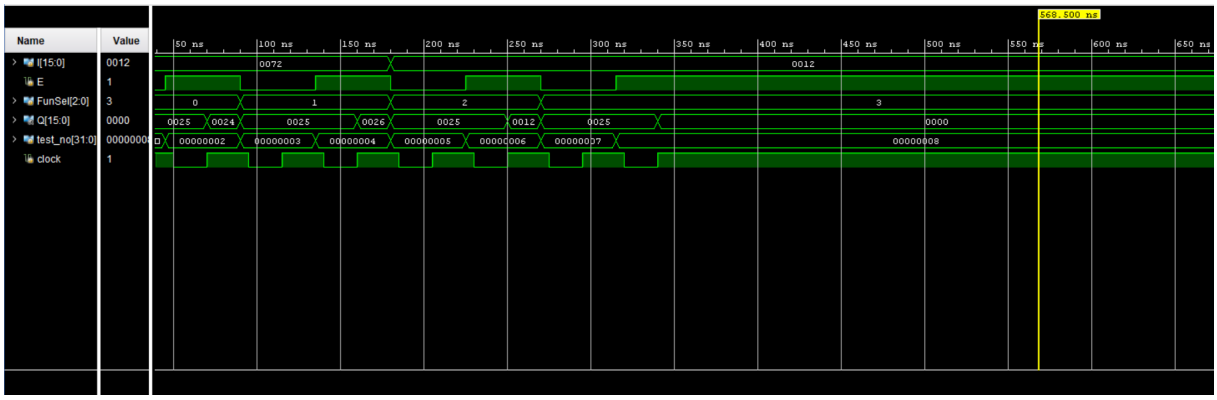

Figure 2: Simulation of Part -1- Register

3.2 Part 2

3.2.1 a: 16-Bit Instruction Register

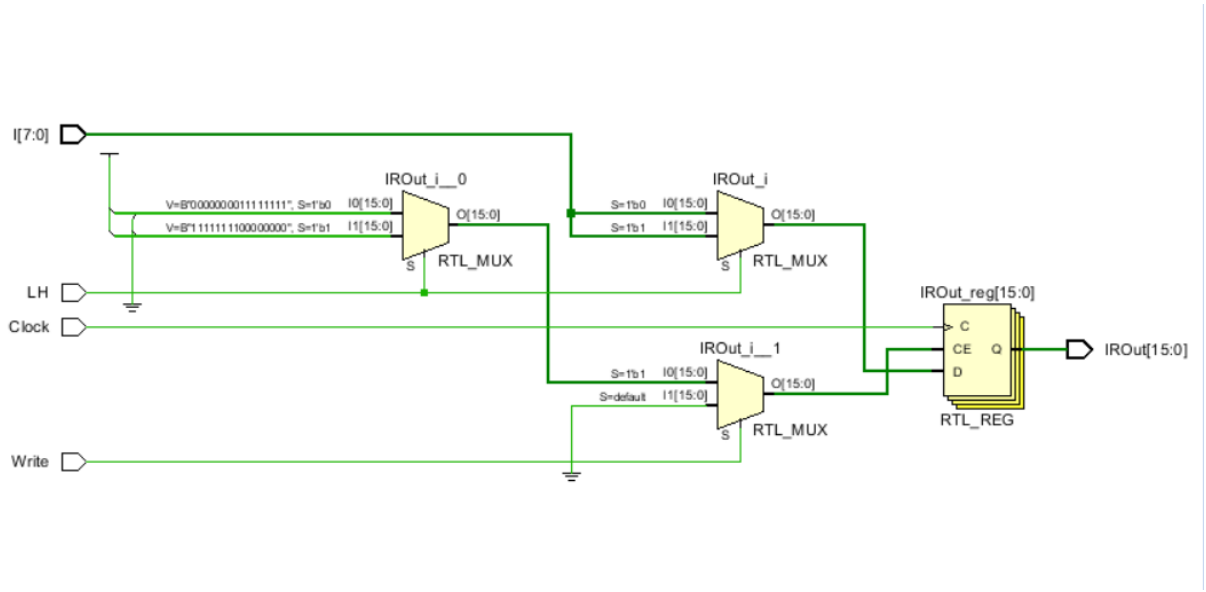

Figure 3: Schematic of Part -2a- IR

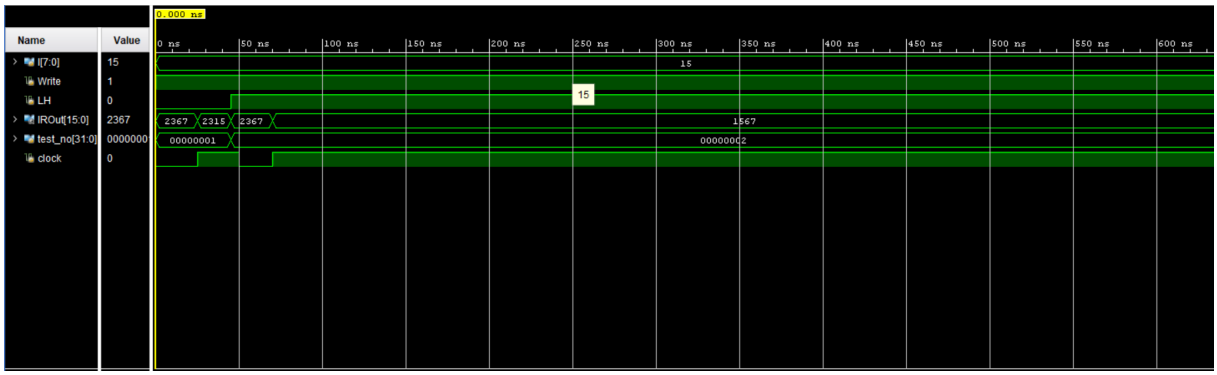

Figure 4: Simulation of Part -2a- IR

3.2.2 b: Register File System

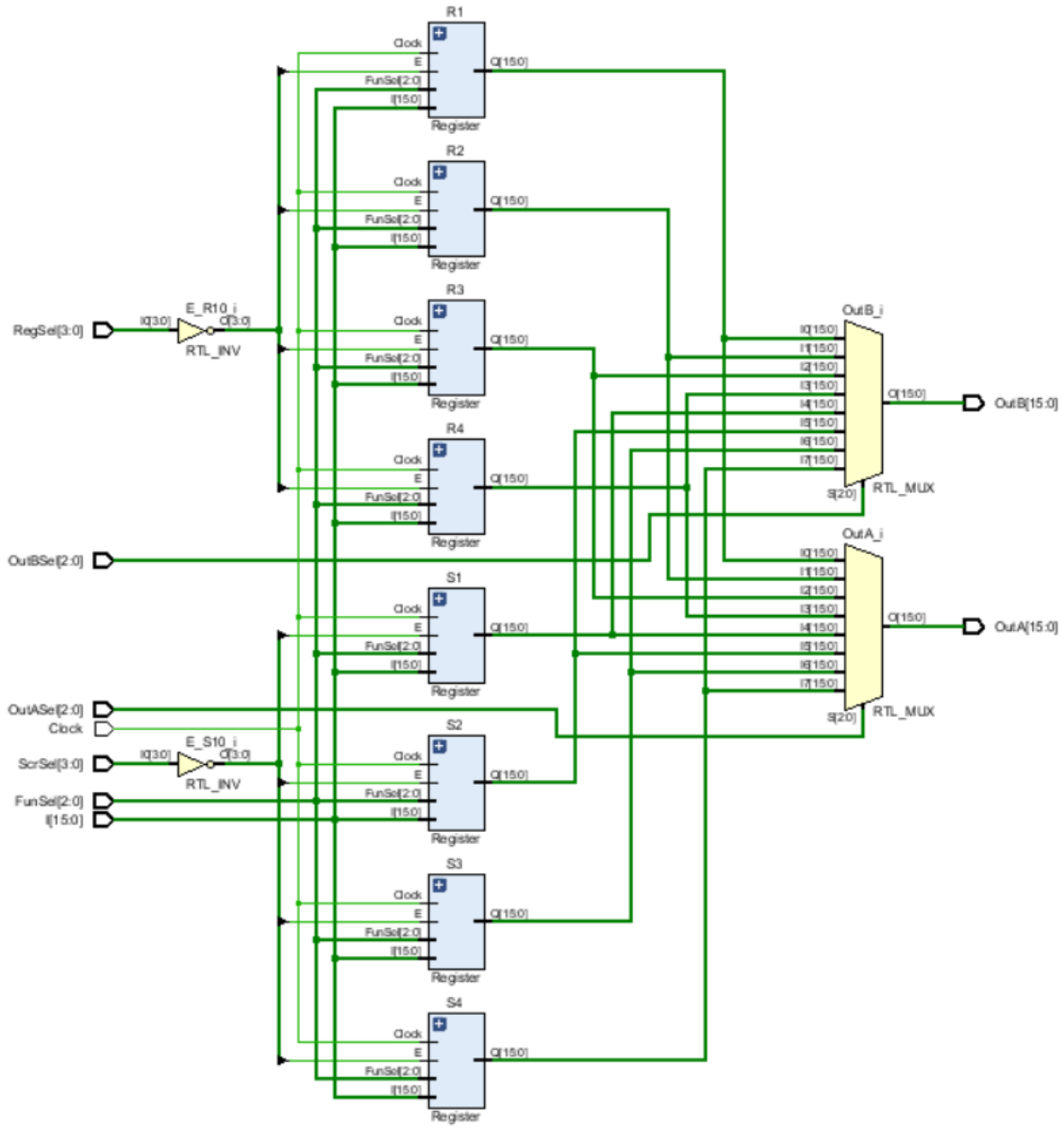

Figure 5: Schematic of Part -2b- RF

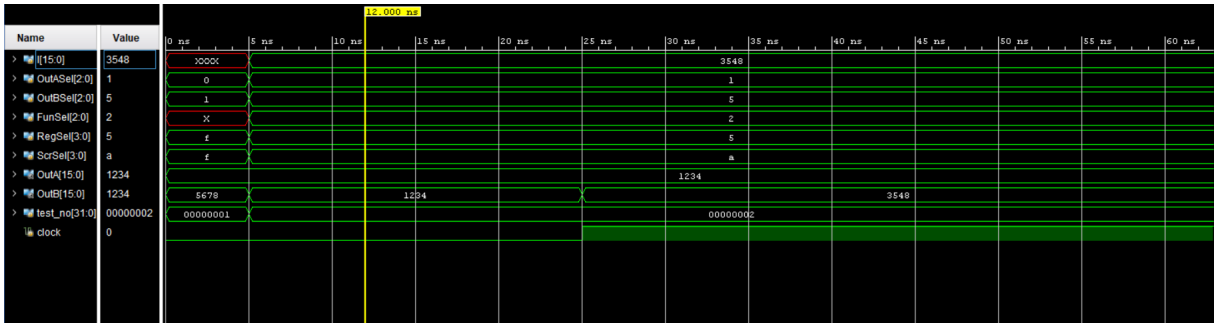

Figure 6: Simulation of Part -2b- RF

3.2.3 c: Address Register File

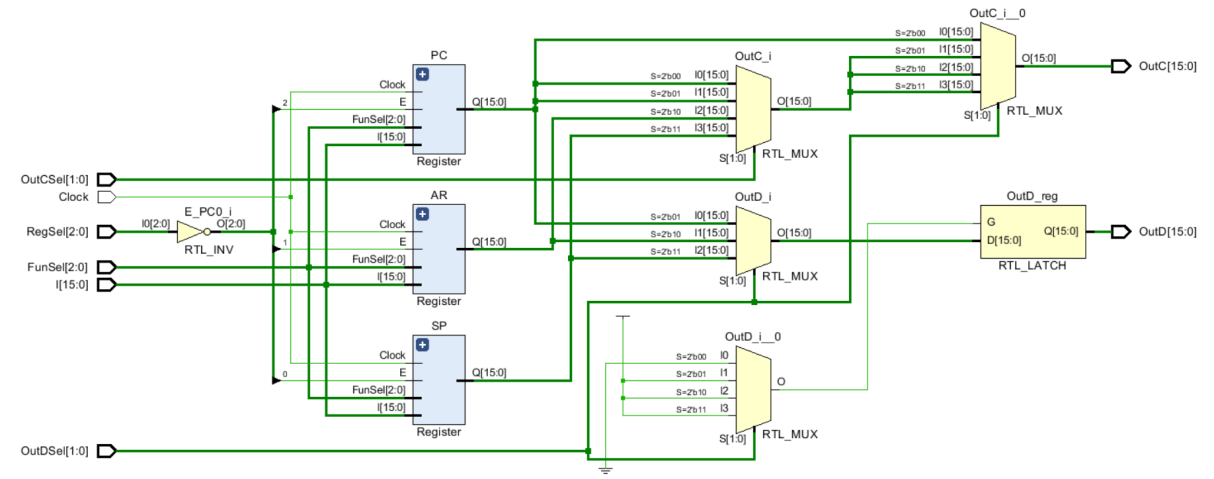

Figure 7: Schematic of Part -2c- ARF

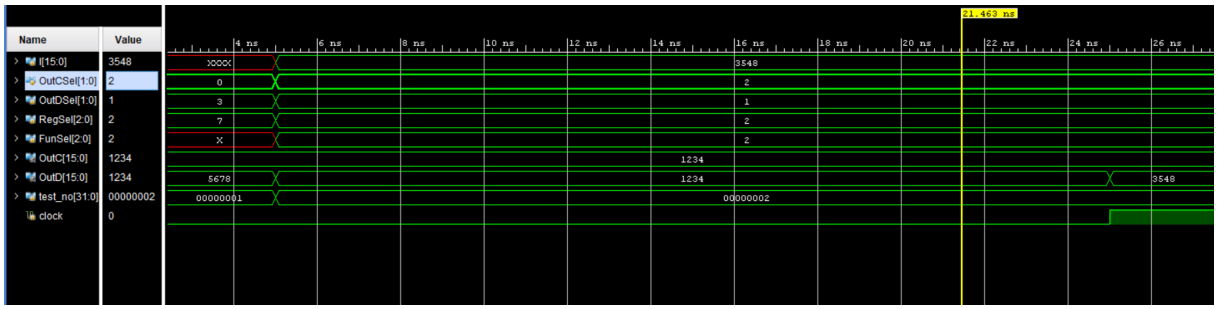

Figure 8: Simulation of Part -2c- ARF

3.3 Part 3: Arithmetic Logic Unit

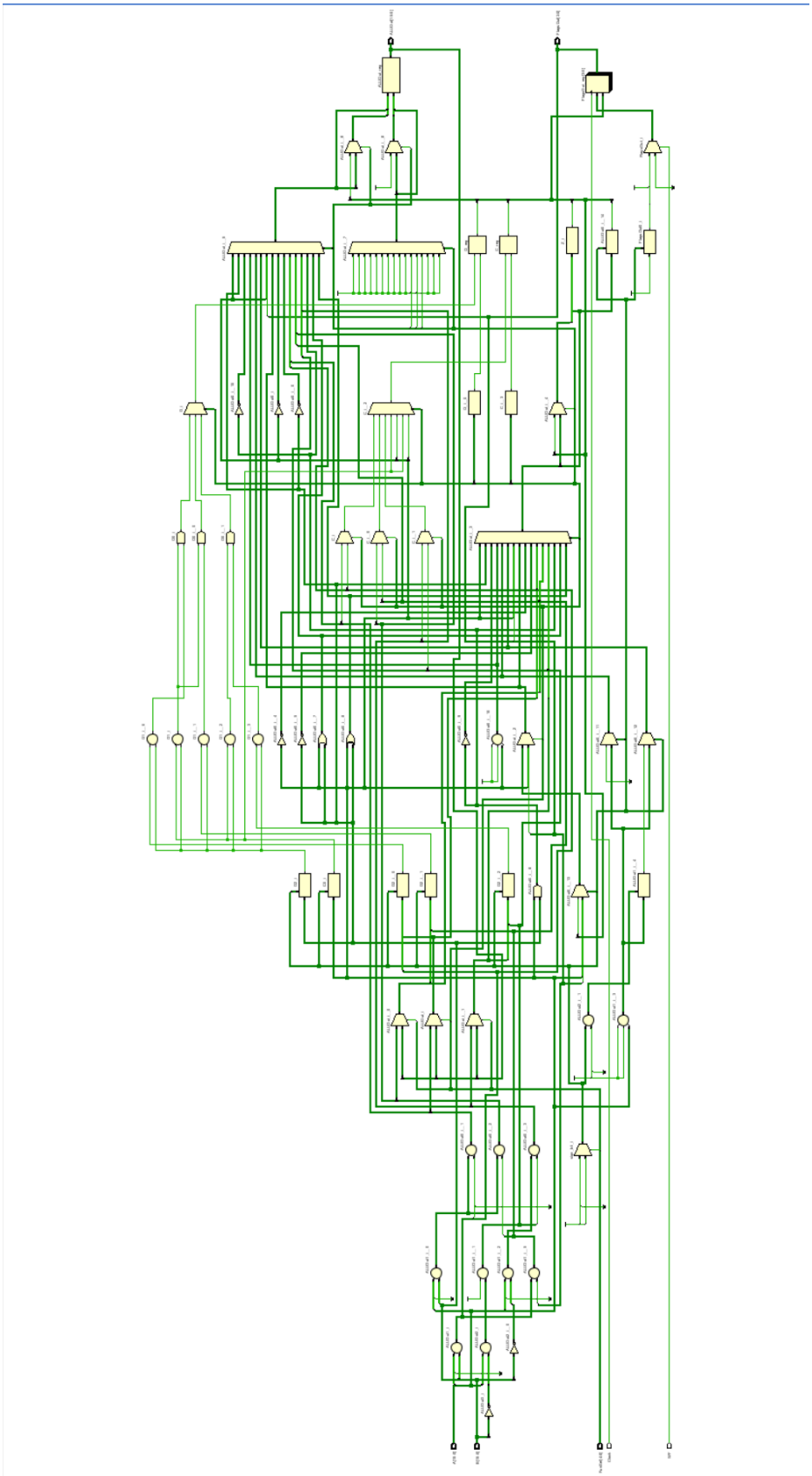

Figure 9: Schematic of Part -3- ALU

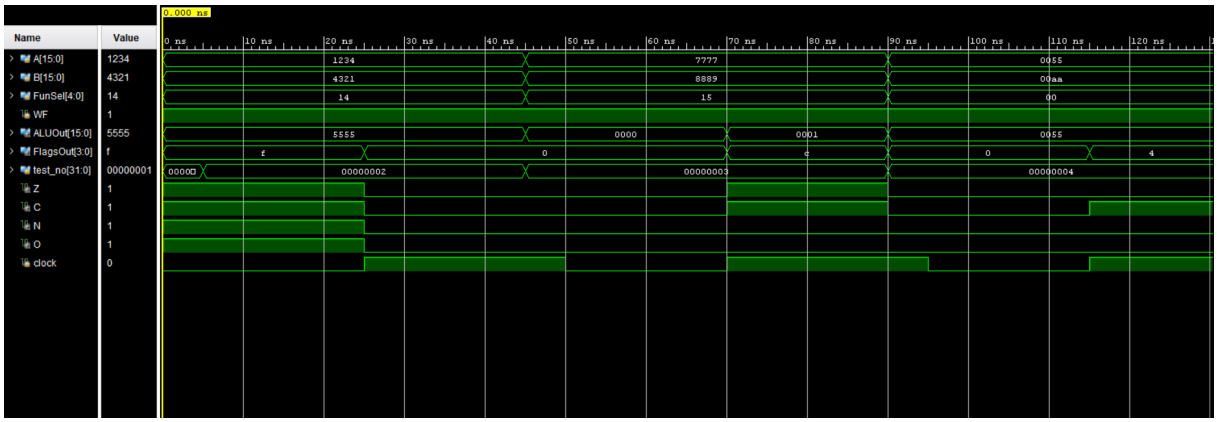

Figure 10: Simulation of Part -3- ALU

3.4 Part 4: ALUSys

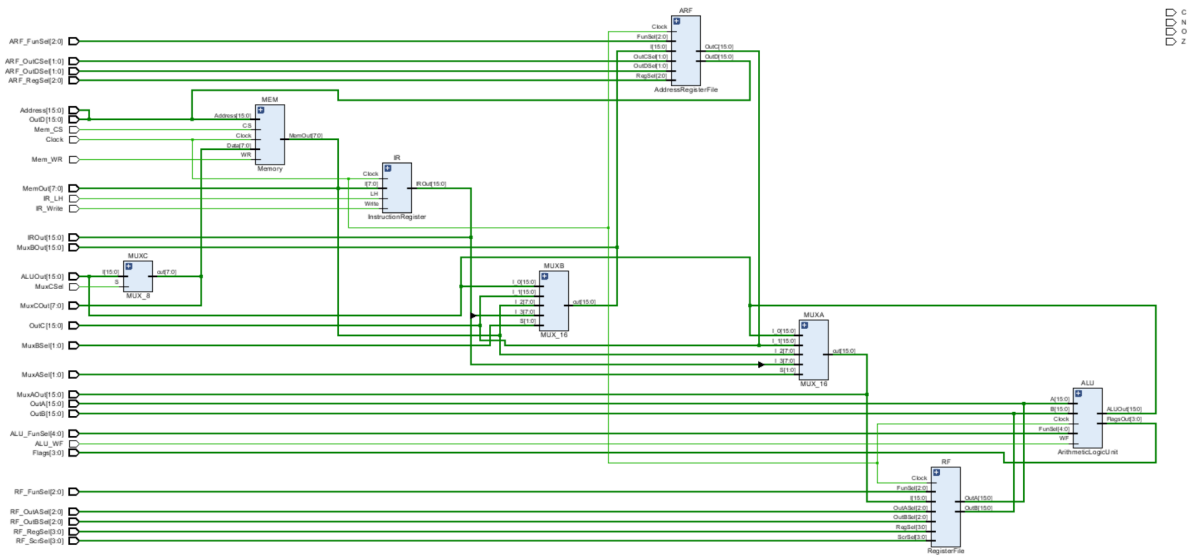

Figure 11: Schematic of Part -4- ALUSys

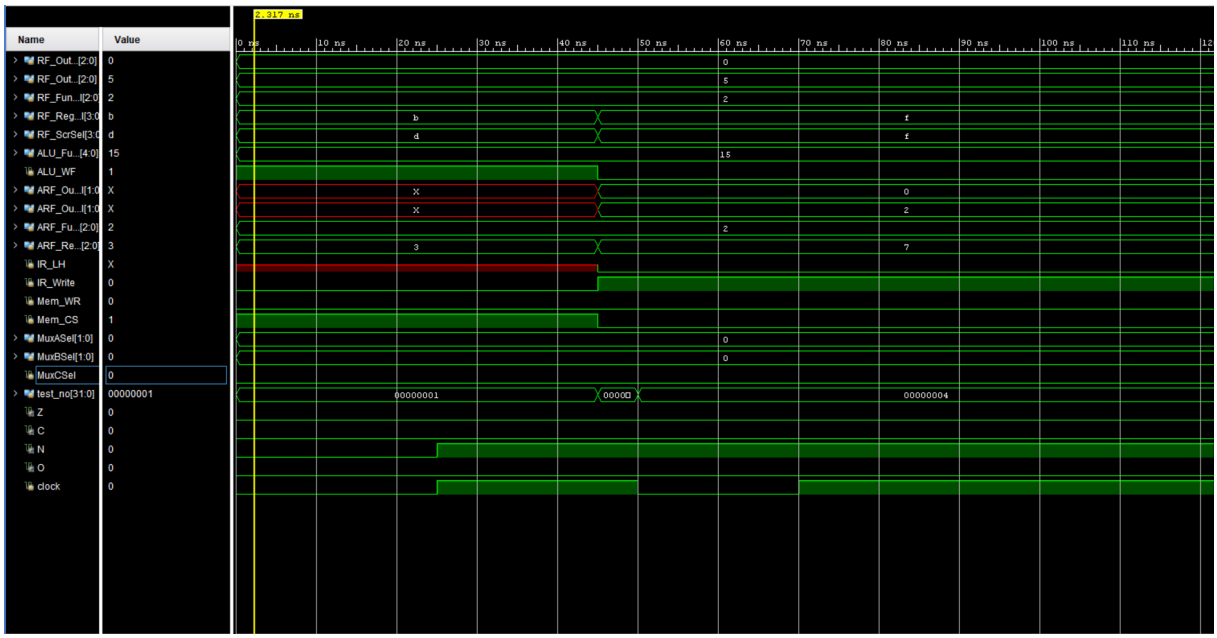

Figure 12: Simulation of Part -4- ALUSys

4 DISCUSSION [25 points]

4.1 Part 1: 16-Bit Register

The table below shows the simulation tests we used to check our module:

E	FunSel	Function	Q	Input	Input Dec.	Output	Output Dec.
0	000	Retain Value	h0025	h0072	114	h0025	37
1	000	Decrement	h0025	X	X	h0024	36
0	001	Increment	h0025	X	X	h0025	37
1	001	Increment	h0025	X	X	h0026	38
0	010	Load	h0025	h0012	18	h0025	37
1	010	Load	h0025	h0012	18	h0012	18
0	011	Clear	h0025	X	X	h0025	37
1	011	Clear	h0025	X	X	h000	0
0	100	Write Low	h0025	hFFFF	-1	h0025	37
1	100	Write Low	h0025	hFFFF	-1	h00FF	255
0	101	Only Write Low	hFF25	h0000	0	hFF25	-219
1	101	Only Write Low	hFF25	h0000	0	hFF00	-256
1	110	Only Write High	hFF25	h0000	0	h0025	37
1	111	Sign Extend, Write Low	hFF25	h15F0	5616	hFFF0	-16

Table 4: Test Cases For Part 1

As seen in the table, All the results we found are consistent with the expected ones. When the Enable of the Register is 0, no matter what data is provided it will retain its current state. Operations such as Clear, Increment, and Decrement are independent of the input data, as they are executed based on the current data stored in the register. For the other functions, the Registers perform the operations as expected.

4.2 Part 2

4.2.1 a: 16-Bit Instruction Register

For the IR implemented in this part, we tested the module using the following cases:

Write	L'H	Q	Input	Output
1	0	h2367	h15	h2315
1	1	h2367	h15	h1567
0	0	h2367	h15	h2367

Table 5: Test Cases For IR

As shown in the table, When the Write flag is 0, the register is disabled and it keeps storing the current data despite the input. However, when it is 1, it either writes the input to the 8 most significant or least significant bits depending on the L'H Flag.

4.2.2 b: Register File System

For this part, following the method we explained in the second section we could obtain the following results:

RegSel	ScrSel	OutASel	OutBSel	Q	I	Funsel	OutA	OutB
b1111	b1111	b000	b001	R1: h1234, R2: h5678	X	b111	h1234	h5678
b0101	b1010	b001	b101	h1234	h3548	b010	h1234	h3548
b1111	b1111	b001	b101	h1234	h3548	b111	h1234	h1234
b1001	b1001	b001	b110	h1234	h3548	b111	h0048	h0048
b1001	b1001	b110	b000	h1234	h3548	b110	h1234	h4834

Table 6: Test Cases For RFS

While testing this module, we provided the current state (Q) to all registers. Then, based on the complements of RegSel and ScrSel, the registers either function if enabled or retain their values. OutASel and OutBSel determine which register outputs are routed to the respective outputs.

For instance, in the first test case, all registers were disabled, so both outputs A and B displayed the current states of registers R1 and R2. However, in the second test, the system was tasked with loading the input I. Output A was taken from the disabled register R2, thus maintaining the current state. Meanwhile, output B was taken from the enabled scratch register S2, allowing us to obtain the input I as the output.

4.2.3 c: Address Register File

This part is similar to the Register File, it just contains fewer registers which are PC, AR, and SP. The results of the tests are as follows:

RegSel	OutCSel	OutDSel	Q	I	Funsel	OutC	OutD
b111	b00	b11	PC: h1234, SP: h5678	X	b111	h1234	h5678
b101	b10	b01	h1234	h3548	b010	h1234	h3548
b111	b10	b11	h1234	h3548	b111	h1234	h0048
b110	b10	b11	h1234	h3548	b111	h1234	h4834
b011	b00	b10	h1234	h3548	b111	h1234	h0000

Table 7: Test Cases For ARF

The functioning of the ARF is similar to the RF; however, we modified some of the functions executed in the registers, and the results are consistent with expectations. For example, in the last test case, the PC was disabled, so it was not cleared, and we obtained its current state as the C output. Meanwhile, the D output was sourced from the enabled AR register, which was cleared.

4.3 Part 3: Arithmetic Logic Unit

In the third part, we implemented the ALU as it has been explained before. While writing our tests, some operations were ambiguous for us so we implemented them according to our understanding, the module works correctly according to that as it follows:

A	B	FunSel	Function	ALUOut	Clock	Flags(ZCNO)
h1234	h4321	b10100	A + B (16-bit)	h5555	-	1111
-	-	-	-	h5555	Posedge	0000
h7777	h8889	b10101	A + B + Carry (16-bit)	h0001	Posedge	1100
h0055	h00AA	b00000	A (8-bit)	h00555	Posedge	0100
h0055	h00AA	b00001	B (8-bit)	hFFAA	Posedge	0110
h0055	h00AA	b00010	NOT A (8-bit)	hFFAA	Posedge	0110
h0055	h00AA	b00011	NOT B (8-bit)	h0055	Posedge	0100
h0055	h00AA	b00100	A + B (8-bit)	hFFFF	Posedge	0010
h0055	h00AA	b00101	A + B + Carry (8-bit)	hFFFF	Posedge	0010
h0055	h00AA	b001100	A - B (8-bit)	hFFAB	Posedge	0111
h0055	h00AA	b00111	A AND B (8-bit)	h0000	Posedge	1101
h0055	h00AA	b01000	A OR B (8-bit)	hFFFF	Posedge	0111
h0055	h00AA	b01001	A XOR B (8-bit)	hFFFF	Posedge	0111
h0055	h00AA	b01010	A NAND B (8-bit)	hFFFF	Posedge	0111
h0055	h00AA	b01011	LSL A (8-bit)	hFFAA	Posedge	0011
h0055	h00AA	b01100	LSR A (8-bit)	h002A	Posedge	0101
h0055	h00AA	b01101	ASR A (8-bit)	h002A	Posedge	0101
h0055	h00AA	b01110	CSL A (8-bit)	hFFAA	Posedge	0011
h0055	h00AA	b01111	CSR A (8-bit)	hFFAA	Posedge	0101
h7777	h8889	b10000	A (16-bit)	h7777	Posedge	0100
h7777	h8889	b10001	B (16-bit)	h8889	Posedge	0110
h7777	h8889	b10010	NOT A (16-bit)	h8888	Posedge	0110
h7777	h8889	b10011	NOT B (16-bit)	h7776	Posedge	0100
h7777	h8889	b10110	A - B (16-bit)	hEEEE	Posedge	0111
h7777	h8889	b10111	A AND B (16-bit)	h0001	Posedge	0101
h7777	h8889	b11000	A OR B (16-bit)	5FFFF	Posedge	0111
h7777	h8889	b11001	A XOR B (16-bit)	5FFFE	Posedge	0111
h7777	h8889	b11010	A NAND B (16-bit)	hFFFE	Posedge	0111
h7777	h8889	b11011	LSL A (16-bit)	hEEEE	Posedge	0011
h7777	h8889	b11100	LSR A (16-bit))	h3BBB	Posedge	0101
h7777	h8889	b10100	ASR A (16-bit)	h3BBB	Posedge	0101
h7777	h8889	b11110	CSL A (16-bit)	hEEEE	Posedge	0011
h7777	h8889	b11111	CSR A (16-bit)	hBBBB	two Posedge	0111

Table 8: Test Cases For ALU

After the 3 tests provided in the simulation file, we tested the 8-bit operations, and then the 16-bit test we executed. After the ALU performs the operations provided by the FunSel it sends the Flags to be stored in the Flags Register that works at the positive edge of the Clock Signal.

This register can help in interpreting the results obtained from the ALU, particularly regarding arithmetic operations. Firstly, based on the provided table in the homework, certain flags, such as Overflow, are updated only during specific operations—in the case of overflow, only arithmetic operations, while it retains its current value (unchanged) for other operations.

Secondly, since arithmetic operations are executed in the same manner for both signed and unsigned numbers, the output from the ALU remains consistent. However, depending on the data stored in this register, users can interpret the results differently.

For example, when working with signed numbers, users may ignore the carry flag and focus on the overflow flag to validate the results. However, when dealing with unsigned numbers, they need to check the carry flag. If a carry flag appears during addition, it indicates that the result is invalid. Similarly, if a carry flag emerges during subtraction, it signifies a borrow which means that the result is invalid as well.

4.4 Part 4: ALUSys

The final part of the project aims to create a new module that imitates a basic computer, by interconnecting each of the previous modules. Data is initially sourced from the memory module, and through a cascading system, the output of one module feeds the input of another. Consequently, if the preceding modules are functioning correctly, it's probable that the ALUSys will work as intended. Tests were used to verify whether inputs and outputs are correctly connected. Due to the complexity of simulation cases, we decided not to add additional tests.

5 CONCLUSION [10 points]

We have learned Verilog syntax and how to use the Xilinx/Vivado program for Register Transfer systems. Additionally, we revised registers and the arithmetic logic unit during the implementation process. We grasped the significance of the clock signal in sequential circuits like registers. Alongside the technical learning process, we also acquired collaborative work skills.

Most of the difficulties we faced were due to indeterminate cases of 8-bit operations. It wasn't clear when to perform sign extension or fill with zeros, so we discussed it among ourselves and made decisions based on our assumptions. Secondly, the number of test

cases was insufficient to thoroughly check and understand the system. Consequently, we conducted our own simulations and made modifications to the code accordingly.