CSE 210

Computer Architecture Sessional

Assignment-3: MIPS Design and Simulation

Section - A2
Group - 02

Members of the Group:

1. 2105091 - Shemanty Mahjabin
2. 2105096 - Fabliha Afia
3. 2105103 - Mahabuba Sharmin Mim

1 Introduction

A processor, also referred to as a processing unit, is a digital circuit designed to execute operations on data from external sources, such as memory or other data streams. The term is commonly associated with the Central Processing Unit (CPU) within a computer system. The CPU is responsible for interpreting and executing the instructions that form a computer program. It carries out fundamental arithmetic, logical, control, and input/output (I/O) operations as specified by the program's instructions.

Key components of a CPU include the Arithmetic Logic Unit (ALU), which handles arithmetic and logical operations, processor registers that provide operands to the ALU and store its results, and the control unit, which manages the overall operation of the CPU by coordinating the fetching and execution of instructions.

In the field of processor design, MIPS (Microprocessor without Interlocked Pipelined Stages) is a widely used Reduced Instruction Set Computing (RISC) architecture. A processor that implements the MIPS instruction set is referred to as a MIPS processor.

For this project, we have designed an 8-bit processor that follows the MIPS architecture. Each instruction in this design is executed in a single clock cycle. The system includes key components such as instruction memory, data memory, a register file, the ALU, and a control unit, all of which work together to carry out the processor's functions.

The processor consists of the following five main components:

1. **Program Counter (PC)**: This 8-bit register increments by 1 after each clock cycle and stores the address of the next instruction to be fetched from memory.
2. **Register File**: This component houses seven registers, namely $zero, $t0, $t1, $t2, $t3, $t4, $t5, and $sp. The $sp register functions as the stack pointer, initialized to 0x00, while the $zero register always holds the value 0x00. The other registers serve as general-purpose storage.
3. **Arithmetic Logic Unit (ALU)**: The ALU is responsible for performing all arithmetic and logical operations, controlled by a 3-bit ALUop code that determines the type of operation. It works with two 8-bit binary inputs to perform calculations.
4. **Control Unit**: The control unit decodes incoming instructions and directs the selection of inputs for the MUXs, the register file, data memory, and the ALU.
5. **Data Memory**: The data memory acts as the main memory for the processor, storing stack values and providing 256 bytes of storage capacity. It plays a crucial role in storing data during the execution of instructions.

2 Instruction Set

2.1 Instruction Set With Instruction ID

Instruction ID	Instruction Type	Instruction
A	Arithmetic	add
B	Arithmetic	addi
C	Arithmetic	sub
D	Arithmetic	subi
E	Logic	and
F	Logic	andi
G	Logic	or
H	Logic	ori
I	Logic	sll
J	Logic	srl
K	Logic	nor
L	Memory	sw
M	Memory	lw
N	Control	beq
O	Control	bneq
P	Control	j

2.2 Instruction Set With Op-Code

No	Binary	Type	Instruction	Category
0	0000	Logic	and	R-Type
1	0001	Arithmetic	addi	I-Type
2	0010	Arithmetic	sub	R-Type
3	0011	Arithmetic	subi	I-Type
4	0100	Control	j	J-Type
5	0101	Control	bneq	I-Type
6	0110	Memory	sw	I-Type
7	0111	Memory	lw	I-Type
8	1000	Logic	or	R-Type
9	1001	Logic	srl	R-Type
10	1010	Logic	sll	R-Type
11	1011	Logic	andi	I-Type
12	1100	Logic	nor	R-Type
13	1101	Logic	ori	I-Type
14	1110	Control	beq	I-Type
15	1111	Arithmetic	add	R-Type

3 Complete Circuit Diagram (of all components)

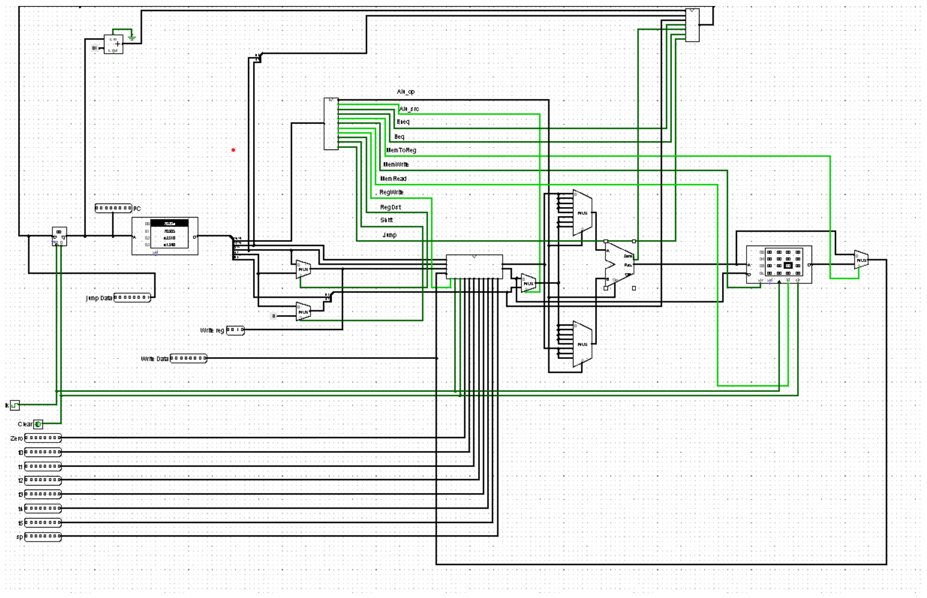

Figure 1: Complete circuit diagram

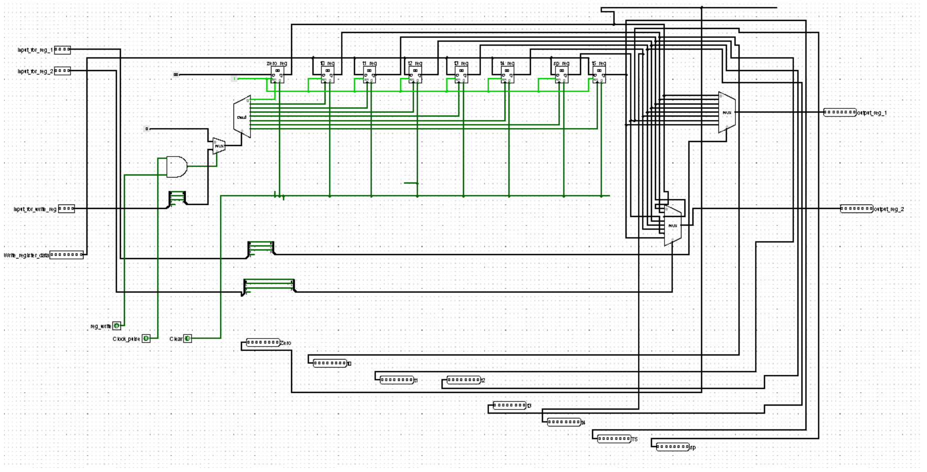

Figure 2: Register circuit

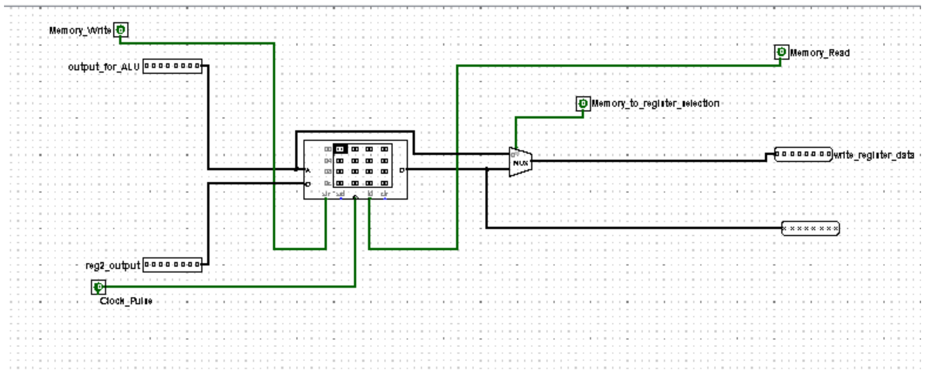

Figure 3: Datamemory circuit

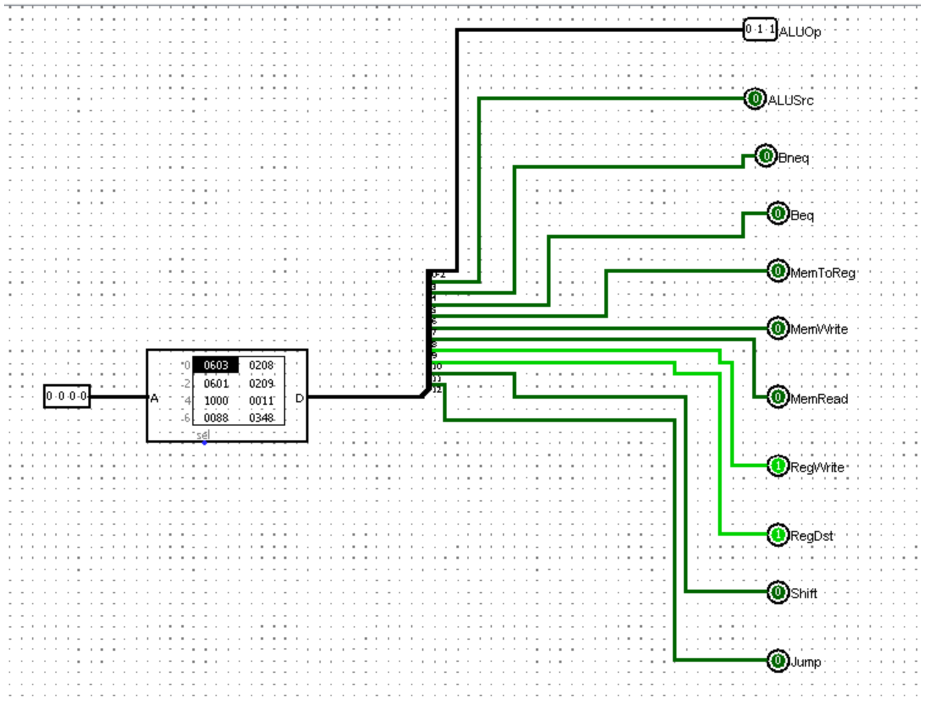

Figure 4: Control unit circuit

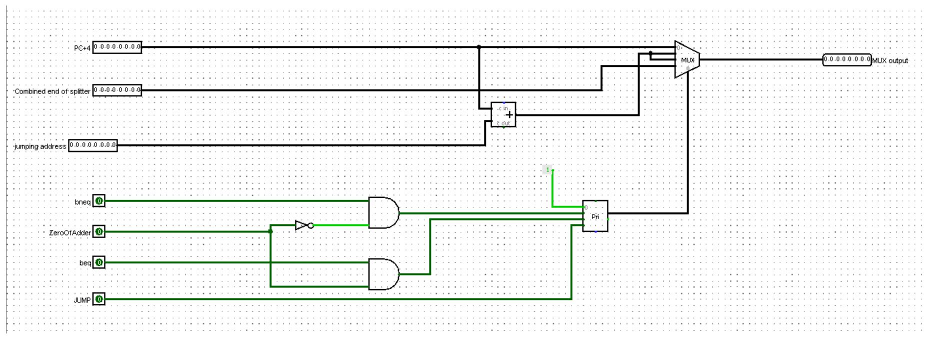

Figure 5: AddressHandler circuit

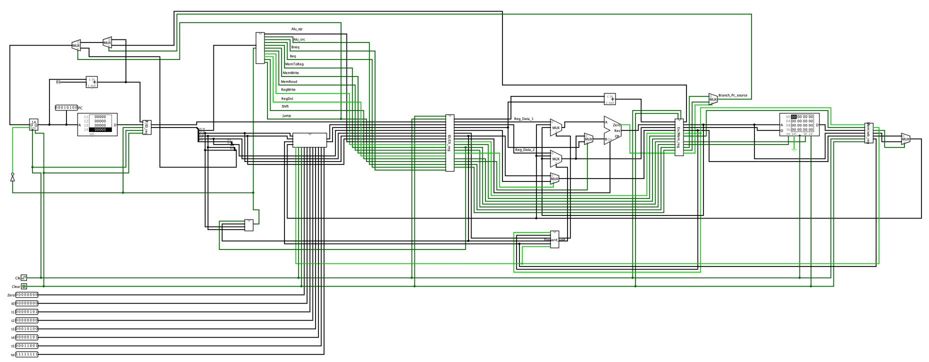

Figure 6: Complete figure with pipelining

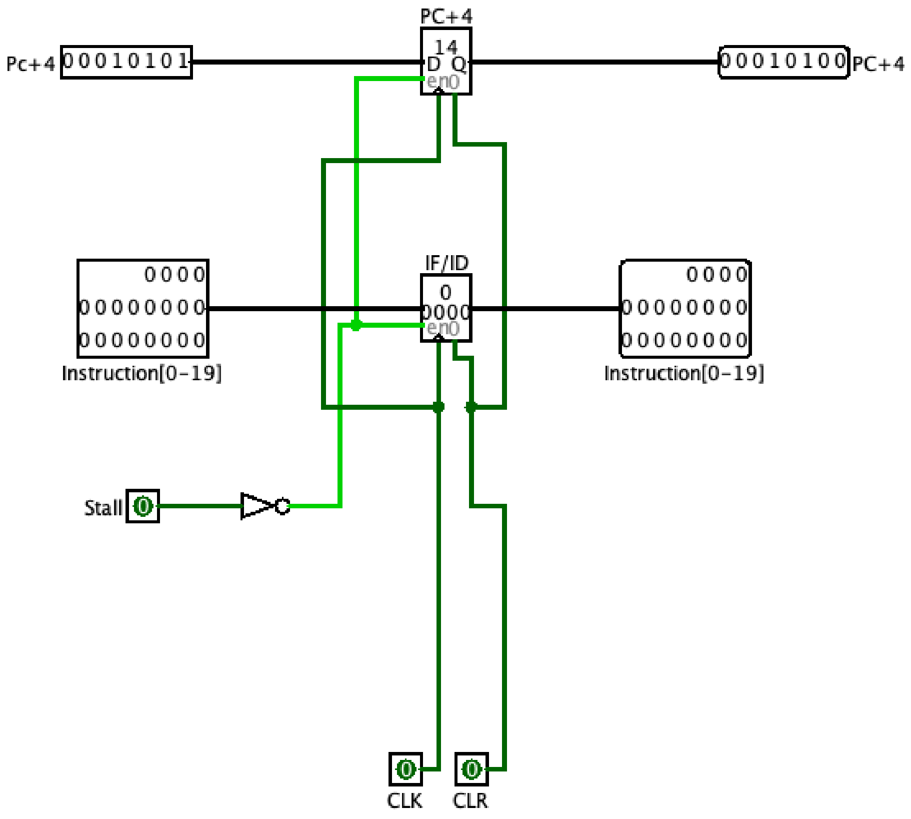

Figure 7: IF/ID Register

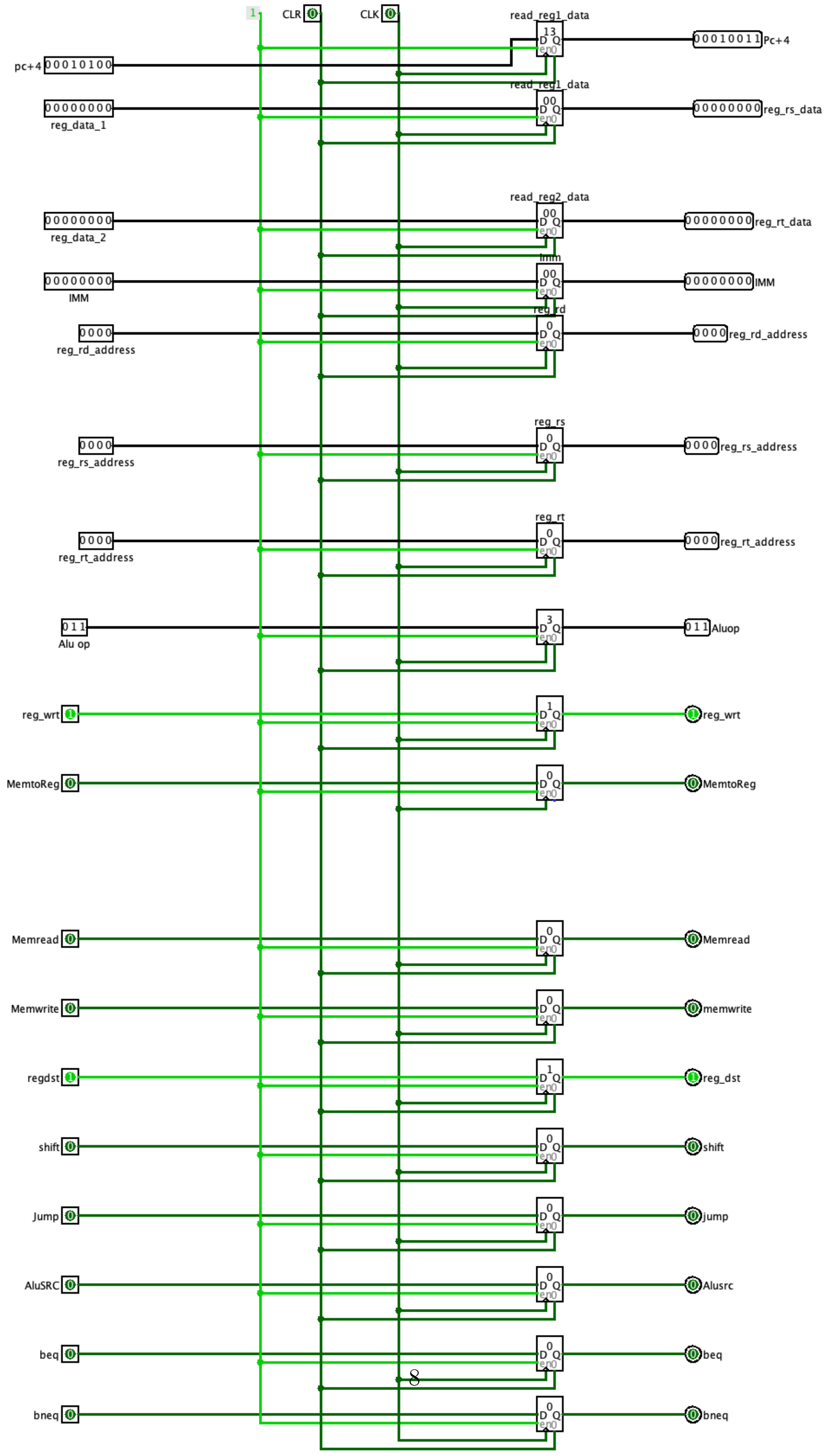

Figure 8: ID/EX Register

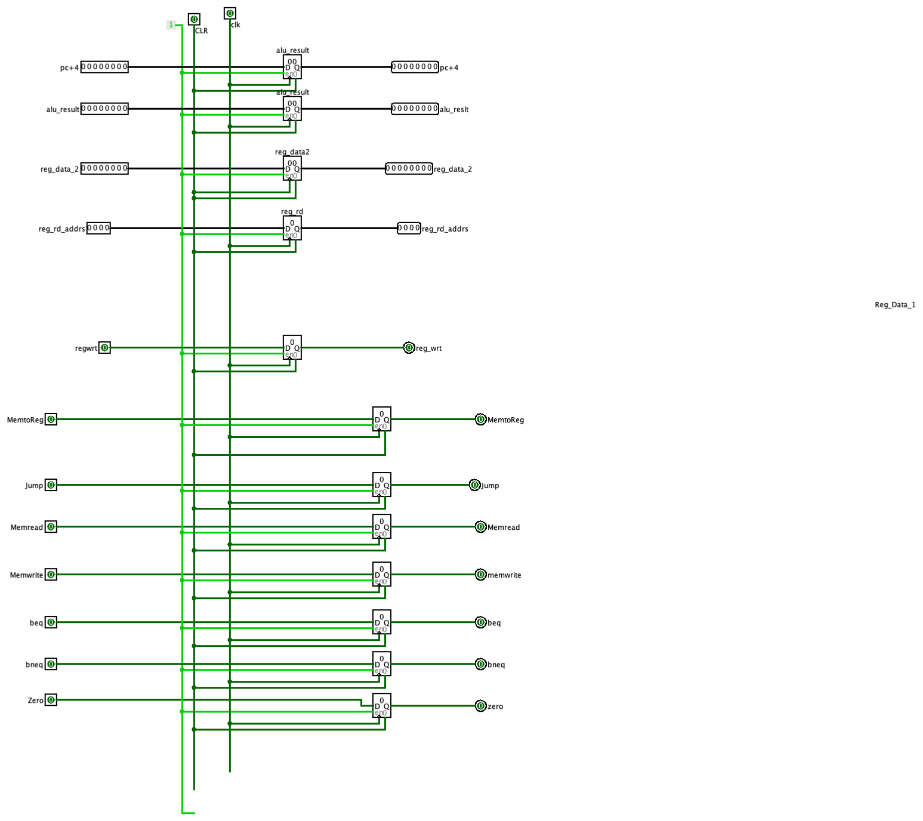

Figure 9: EX/MEM Register

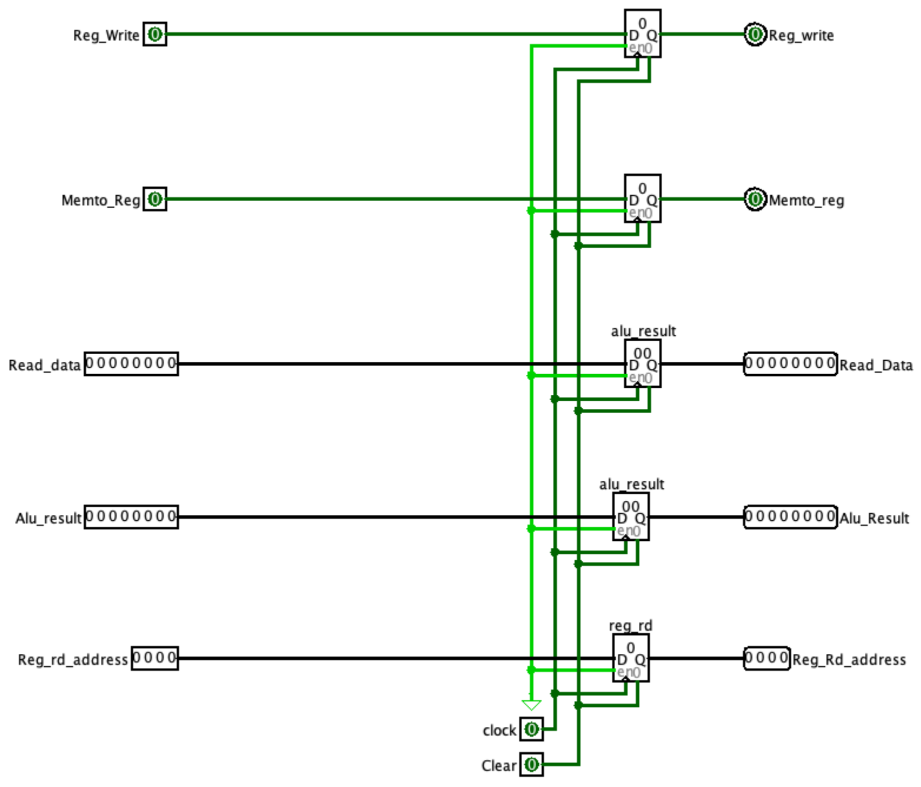

Figure 10: MEM/WB Register

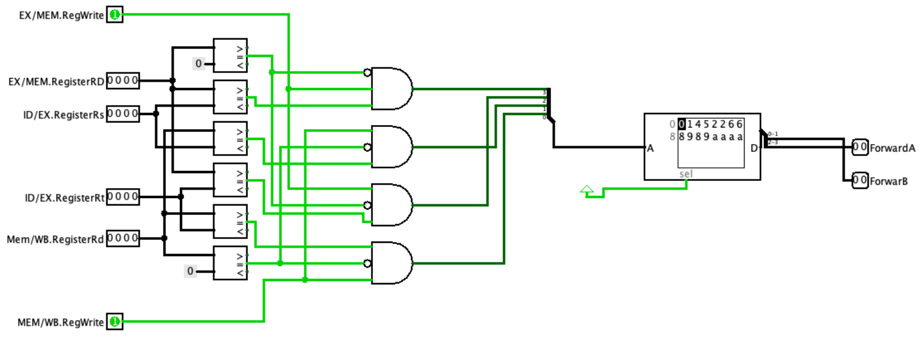

Figure 11: Forward Unit

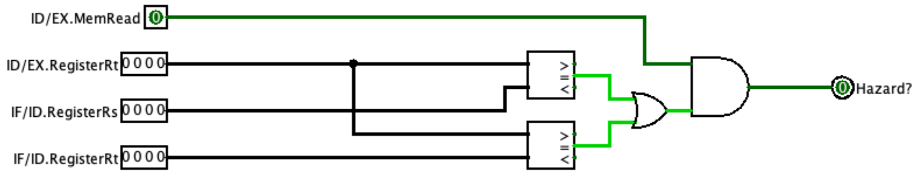

Figure 12: Hazard Detection

4 How to Write and Execute a Program on Your Machine

This section outlines the procedure for writing and running a program on the MIPS processor.

Step 1: Writing the MIPS Program

The first task is to write the program in MIPS assembly language. This can be done using any text editor or an Integrated Development Environment (IDE) that supports assembly programming. MIPS assembly consists of various instructions, such as load, store, arithmetic operations, and control instructions, which the processor can decode and execute.

Step 2: Assembling the MIPS Code

Once the program has been written in MIPS assembly, the next step is to assemble the code. The assembler translates the MIPS assembly instructions into machine-readable hexadecimal code. These hexadecimal values represent the binary encoding of the instructions that the processor understands. The output is a sequence of hexadecimal numbers corresponding to the MIPS instructions.

Step 3: Loading the Instruction ROM

After the machine code is generated, it is loaded into the Instruction ROM (Read-Only Memory). The Instruction ROM holds the program instructions that the processor will fetch and execute. During execution, the processor retrieves these instructions sequentially from the Instruction ROM.

Step 4: Loading the Control Signals from the Control File

Along with the instruction ROM, a separate control file is utilized to store the control signals required for the processor's various components, such as the ALU, Register File, and Data Memory. These control signals, which are usually kept in a text file, guide the operation of the processor. When loaded into the processor, the control unit uses these signals to direct the actions of the ALU, registers, and memory.

Step 5: Executing the Program with Clock Pulses

The program execution proceeds with the help of clock pulses. With each clock pulse, the processor fetches the next instruction from the Instruction ROM. After fetching an instruction, the processor decodes it and generates the corresponding control signals based on the control file. These signals determine the necessary actions, such as performing an arithmetic operation with the ALU or retrieving data from memory. The results of the instruction are then written back to the registers or memory. This cycle continues with each clock pulse, and the processor executes each instruction in the sequence.

5 ICs used with count as a chart

Gate	Gate Count	IC	IC Count
MUX(4-to-1)	3	74153	2
MUX(2-to-1)	10	74157	3
MUX(8-to-1)	1	74151	1
MUX(16-to-1)	2	74150	2
Adder(8-bit)	3	7483	6
Decoder(4-to-16)	1	74154	1
AND(2-bit)	14	7408	4
AND(8-bit)	1	7430	1
NOT(1-bit)	4	7404	1
NOR(8-bit)	3	4078	3
Priority Encoder(2-bit)	1	Priority Encoder(2-bit)	1
Register(8-bit)	9	Register(8-bit)	9
ROM(256X20)	1	ROM(256X20)	1
ROM(256X8)	1	ROM(256X8)	1
ROM(16X13)	2	ROM(16X13)	2
Subtractor(8-bit)	1	Subtractor(8-bit)	1
RAM(256X8)	2	RAM(256X8)	2
Clock	2	Clock	2
Right-Shifter(8-bit)	1	Right-Shifter(8-bit)	1
Left-Shifter(8-bit)	1	Left-Shifter(8-bit)	1
comparator	4		

Table 1: ICs used with their counts.

6 Contribution of Each Member

Each group member contributed significantly to the project, ensuring its success.
The specific contributions are as follows:

- **2105091** : address handler,main circuit,IF/ID Register,ID/EX register,EX/Mem register,Hazard detection

- **2105096** : code,data memory,register file.
- **2105103** : control unit,main circuit,ID/EX register,Forward unit,Mem/WB Register,Pipelining

7 Discussion

Throughout the circuit implementation process, we encountered numerous challenges that required us to revise our design multiple times. Some of the initial designs involved a large number of ICs, making the circuit unnecessarily complex and resource intensive. To address this issue, we carefully analyzed alternative approaches and reduced the number of ICs without compromising functionality. This process involved discarding inefficient designs and iterating on new streamlined solutions.

Another major focus was testing for edge cases. We devoted significant time to thoroughly testing the circuit against all possible scenarios to ensure its reliability. Each test case was carefully verified to identify and address potential design flaws or logical inconsistencies. This step proved to be crucial in creating a robust and error-free circuit.

Additionally, we tackled challenges related to control signal synchronization, which required precise timing to avoid conflicts between components. Resolving these issues demanded close attention to the clocking mechanisms and overall timing of operations. Furthermore, reducing the propagation delay across the circuit was a key optimization that significantly improved the performance of the design.

In conclusion, by addressing these challenges and dedicating efforts to optimization, we successfully implemented a circuit that is both efficient and functional. This iterative approach allowed us to achieve a design that balances simplicity, performance, and reliability.