North South University

Department of Electrical & Computer Engineering

Project Report

Course ID & Name: CSE332 Computer Organization and Architecture

Section: 02

Semester: Spring 2025

Project Name: **15Bits Single Cycle MIPS CPU**

Submitted to: Ms. Tanjila Farah (TnF)

Report Submission Date: 12 April 2025

Group Number: 13

Group Members ID & Name:

1. Shefa Tabassum	2232993042
2. Easfiah Hossain Lamia	2231699642

Remarks:

Score

Project Title: **15Bits Single Cycle MIPS CPU**

Introduction:

This report presents the design and implementation of a **15-bit CPU** built using **Logisim**, based on a custom-defined **Instruction Set Architecture (ISA)**. The primary objective of this project is to understand the fundamental components and working principles of a basic processor by designing and integrating its core modules—such as the register file, ALU, control units, and memory units—within a simplified CPU framework.

By following a structured ISA with **R-type**, **I-type**, and **J-type** instruction formats, the project aims to simulate how a processor fetches, decodes, and executes instructions. The design emphasizes modular construction, clear control logic, and efficient data flow, offering practical insight into the inner workings of CPU architecture.

1) How many operands?

In our ISA, the number of operands varies based on the instruction format:

- **R-Type Instructions** use **three operands** (three registers).
- **I-Type Instructions** use **two operands** (two registers and an immediate value).
- **J-Type Instructions** use **one operand** (a memory address).

Operand Count Breakdown

Operand Count	Instructions	Operands (Format)
0 operands	nop	No operands (does nothing).
1 operand	jmp	One address (J-Type).
2 operands	sw, lw ,beq,ori,addi	Two registers + one memory address (I-Type).
3 operands	and, xor, sub	Three registers (R-Type).

2) Types of operand? (Register based or Memory based??)

Operand Type Classification

Type	Instructions	Description
Register-Based	and, xor, ori, addi, sub	Operate mainly on registers, with addi,ori using an immediate value.
Memory-Based	sw, lw	Access memory or load/store data from/to memory.
Branching	beq	Branch instruction that compares values and branches if they are equal.
Jump	jmp	Uses an address for jumping to another instruction.

3) How many operations?

Our ISA consists of 10 operations, as assigned:

1. BEQ (Branch Equal)

2. AND (Logical AND)

3. XOR (Logical XOR)

4. NOP (No Operation)

5. ORI (Logical OR Immediate)

6. ADDI (Add Immediate)

7. SW (Store Word)

8. JMP (Jump)

9. SUB (Subtract)

10. LW (Load Word)

4) Types of operations? (Arithmetic, logical, branch type?? How many from each category? List the opcodes and respective binary values

Our ISA consists of three categories of operations: **Arithmetic**, **Logical**, **Branching**, **Memory**, and **Jump**. Below is a breakdown of how many instructions belong to each category, along with their opcodes and binary values.

1. Arithmetic Operations (2 Instructions)

Instruction	Opcode	Binary Value
ADDI (Add Immediate)	011	011xxxxxxxxxxxx
SUB (Subtract)	000	000xxxxxxxxxxxx

2. Logical Operations (3 Instructions)

Instruction	Opcode	Binary Value
AND (Logical AND)	000	000xxxxxxxxxxxx
XOR (Logical XOR)	000	000xxxxxxxxxxxx
ORI (Logical OR Immediate)	010	010xxxxxxxxxxxx

3. Branch Operations (1 Instruction)

Instruction	Opcode	Binary Value
BEQ (Branch if Equal)	001	001xxxxxxxxxxxx

4. Memory Operations (2 Instructions)

Instruction	Opcode	Binary Value
LW (Load Word)	110	110xxxxxxxxxxxx
SW (Store Word)	100	100xxxxxxxxxxxx

5. Jump Operation (1 Instruction)

Instruction	Opcode	Binary Value
JMP (Jump)	101	101xxxxxxxxxxxx

6. No Operation (NOP)

Instruction	Opcode	Binary Value
NOP (No Operation)	000	000000000000000

5) No. of the format of instruction (how many different formats?) (R, I , J)

There are **3 different instruction formats**:

1. **R-Type (Register Format)**
2. **I-Type (Immediate Format)**
3. **J-Type (Jump Format)**

Format	Used For	Instructions
R-Type	Register-based operations	AND, XOR, SUB, NOP
I-Type	Immediate arithmetic, logical, memory access, and branching	ORI, ADDI, LW, SW, BEQ
J-Type	Unconditional jumps	JMP

6) Description of each of the formats (fields and field length)

R-type (Register type):

opcode	rs	rt	rd	function
3 bit	3 bit	3 bit	3 bit	3 bit

I-type (Immediate type):

opcode	rs	rt	immediate
3 bit	3 bit	3 bit	6 bit

J-type (Jump type):

opcode	Address
3 bit	12 bit

7) **Table with assigned function/operation list and our assigned opcode/ function bit.**

Category	Operation	Name	Opcode	Function	Type/Format
Branch	Branch Equal	BEQ	001	-	I-type
Logical	Logical AND	AND	000	000	R-type
Logical	Logical XOR	XOR	000	001	R-type
No Operation	No Operation	NOP	000	000	R-type
Logical	Logical OR Immediate	ORI	010	-	I-type
Arithmetic	Add Immediate	ADDI	011	-	I-type
Memory	Store Word	SW	100	-	I-type
Jump	Jump	JMP	101	-	J-type
Arithmetic	Subtract	SUB	000	010	R-type
Memory	Load Word	LW	110	-	I-type

8) **Table with number of registers we will use.**

Register name	**Register number**	**Usage**
R0	000	Zero register
R1	001	General purpose
R2	010	General purpose
R3	011	General purpose
R4	100	General purpose
R5	101	General purpose
R6	110	General purpose
R7	111	General purpose

Control Signal Table

Instruction	**RegDst**	**ALU Src**	**MemtoReg**	**RegWrite**	**Mem Read**	**Mem Write**	**Branch**	**Jump**	**ALU Op**	**Opcode**	**Function**
BEQ	0	0	0	0	0	0	1	0	00	001	-
AND	1	0	0	1	0	0	0	0	10	000	000
XOR	1	0	0	1	0	0	0	0	10	000	001
NOP	0	0	0	0	0	0	0	0	xx	000	-
ORI	0	1	0	1	0	0	0	0	11	010	-
ADDI	0	1	0	1	0	0	0	0	01	011	-
SW	x	1	x	0	0	1	0	0	01	100	-
JMP	0	0	0	0	0	0	0	1	xx	101	-
SUB	1	0	0	1	0	0	0	0	10	000	010
LW	0	1	1	1	1	0	0	0	01	110	-

ALU Control Signals

Instruction	ALUOp	Function	Pin 2	Pin 1	Pin 0
AND	10	000	0	0	0
XOR	10	001	0	0	1
SUB	10	010	0	1	0
addi/sw/lw	01	-	0	1	1
ori	11	-	1	0	0
beq	00	-	0	1	0

Explanation of ALU control signals

In our design, R-type instructions have a fixed opcode of 000 and a fixed ALUOp of 10, since the specific operation is determined by the function field.

For I-type instructions like LW, SW, and ADDI, we assign them an ALUOp of 01, as all of them require the ALU to perform addition—whether it's computing memory addresses or immediate arithmetic.

The ORI instruction, while also I-type, has a distinct behavior and thus is assigned a different ALUOp value, 11, along with separate control signal settings, since bitwise operations like ORI require different handling than arithmetic instructions.

Finally, the BEQ instruction uses ALUOp = 00 and the same pin configuration as SUB. The ALU is configured to perform subtraction, which is necessary for comparing register values in case of BEQ . Control pins are set accordingly to ensure the correct ALU operation is executed.

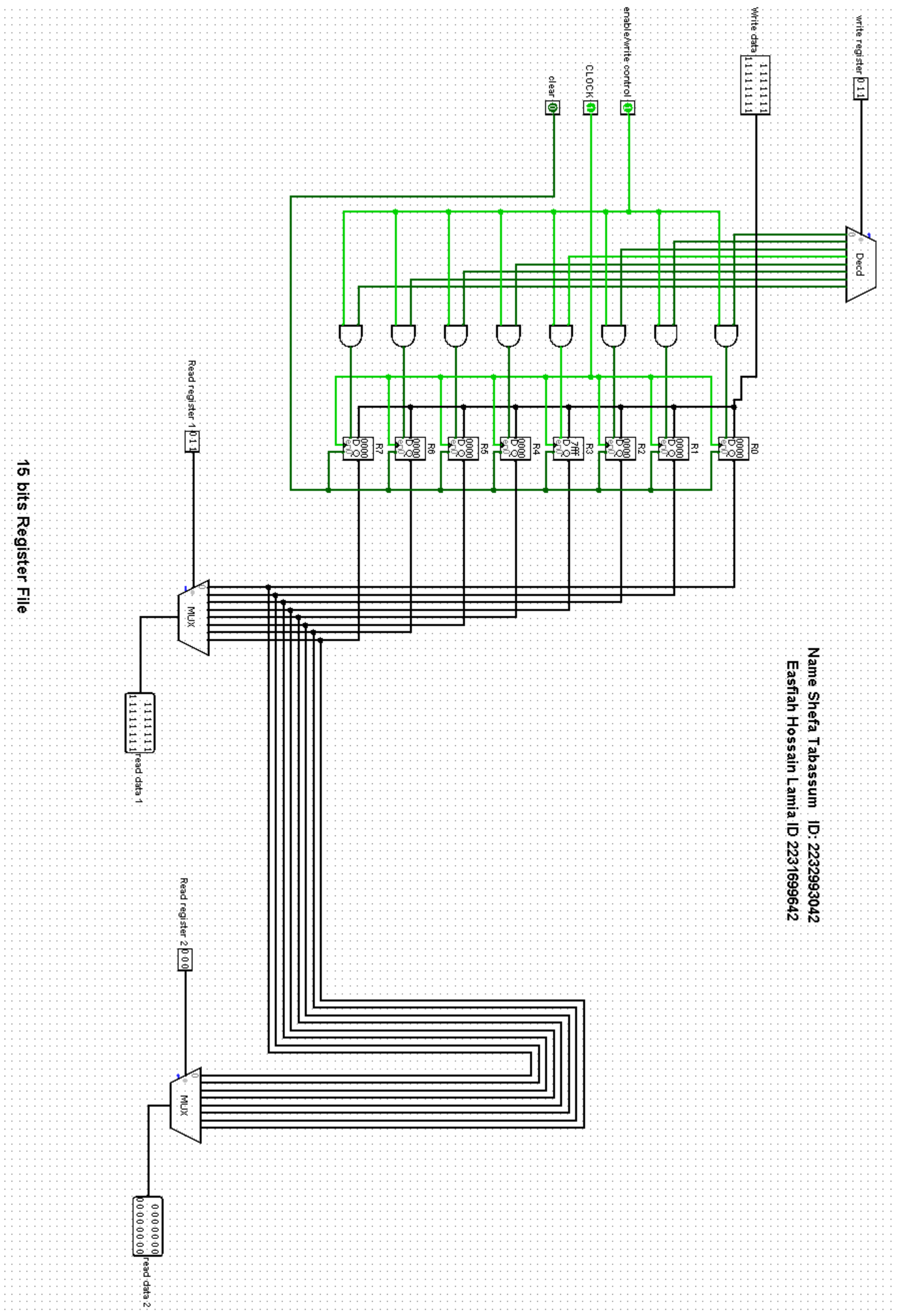


Figure: 15 Bits Register file

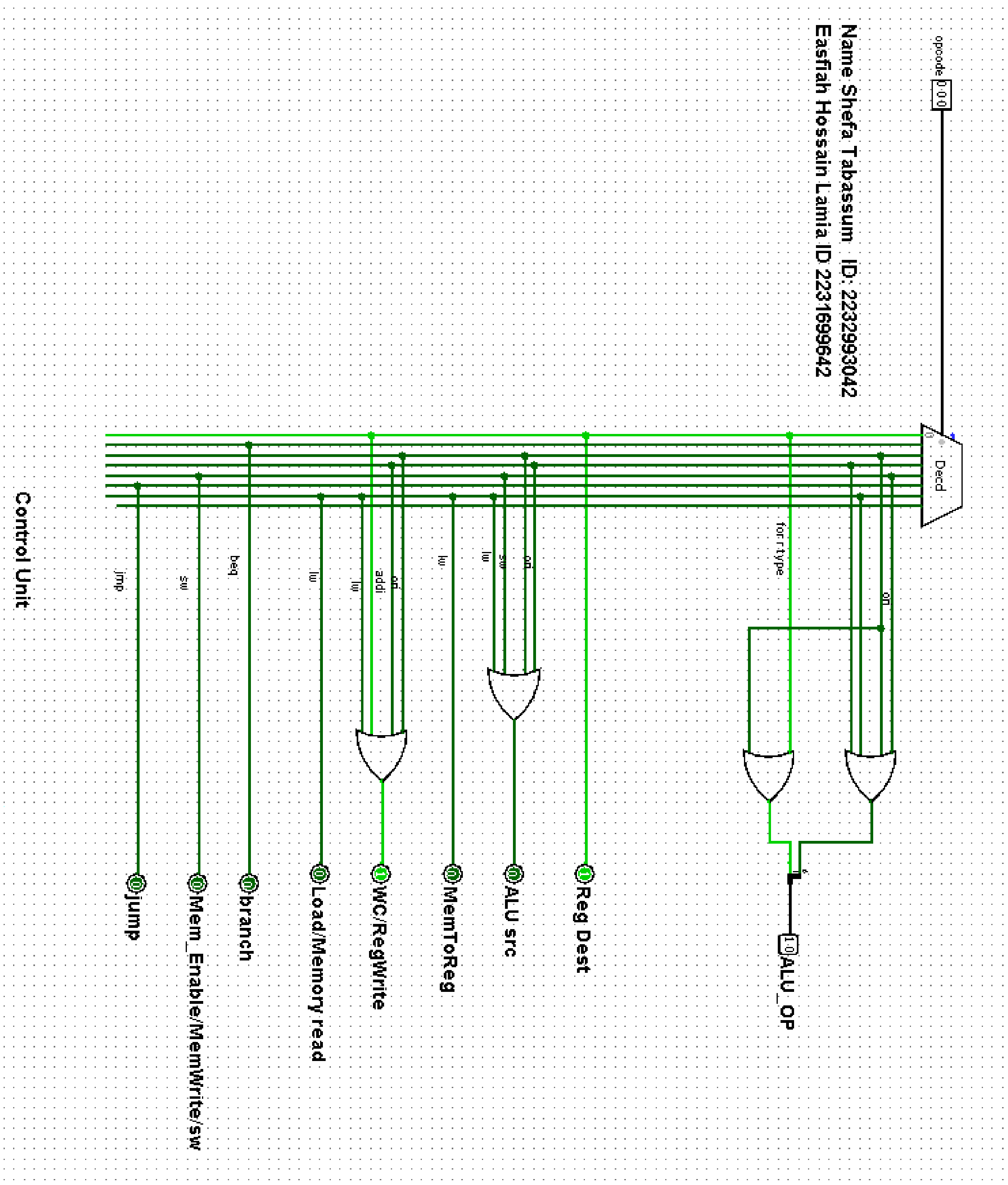


Fig: Control Unit

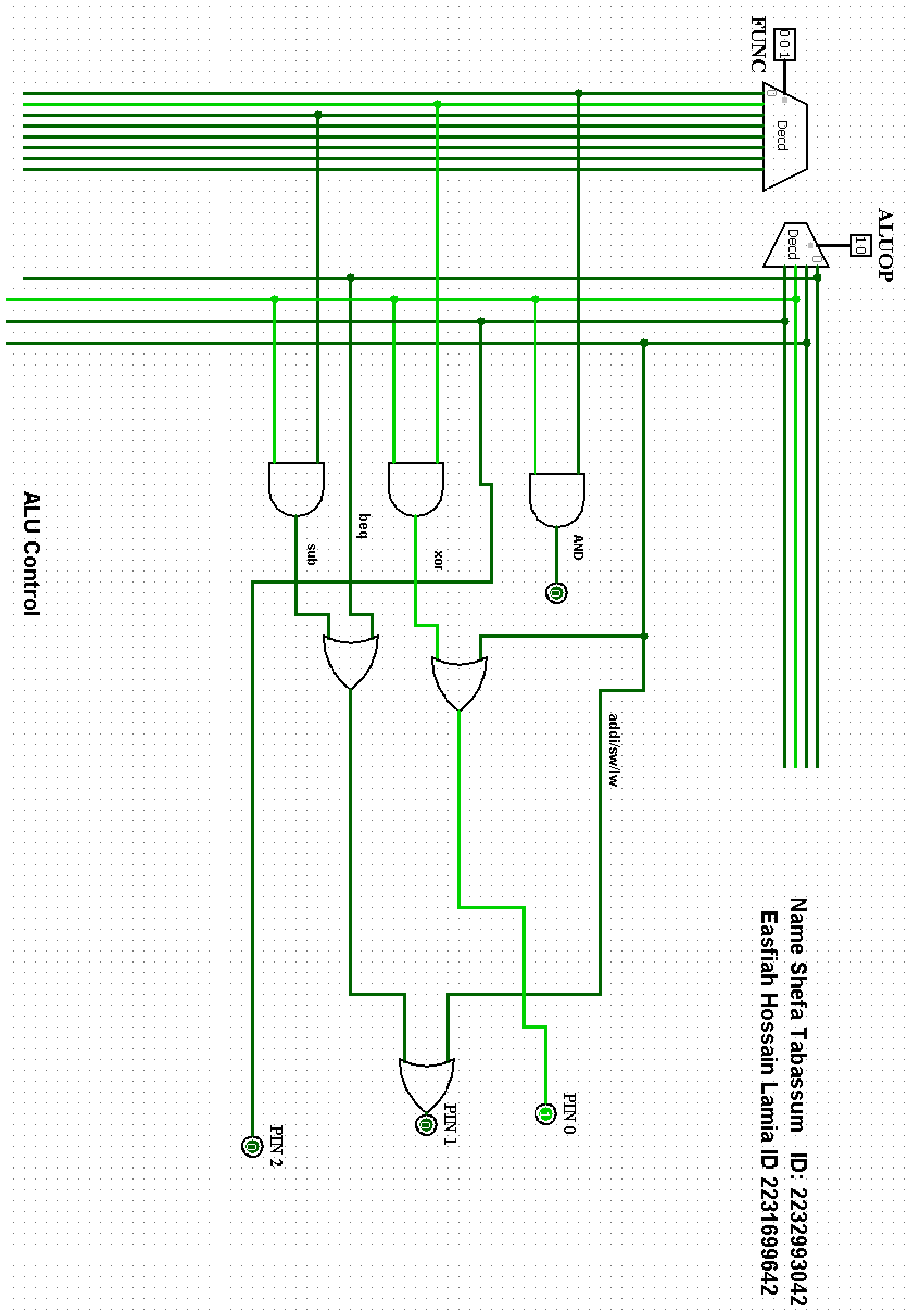


Fig: ALU control

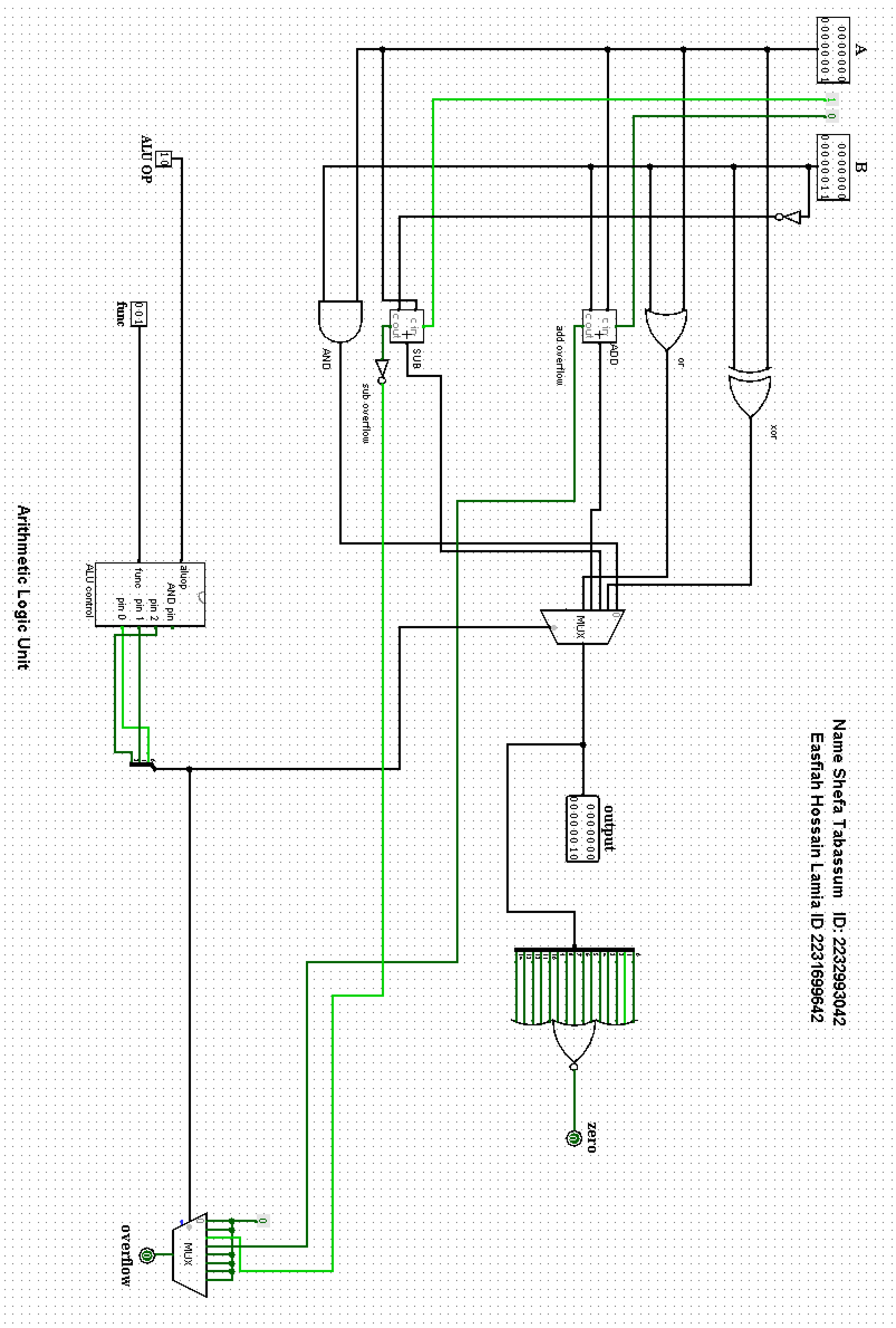


Figure: Arithmetic Logic Unit (ALU)

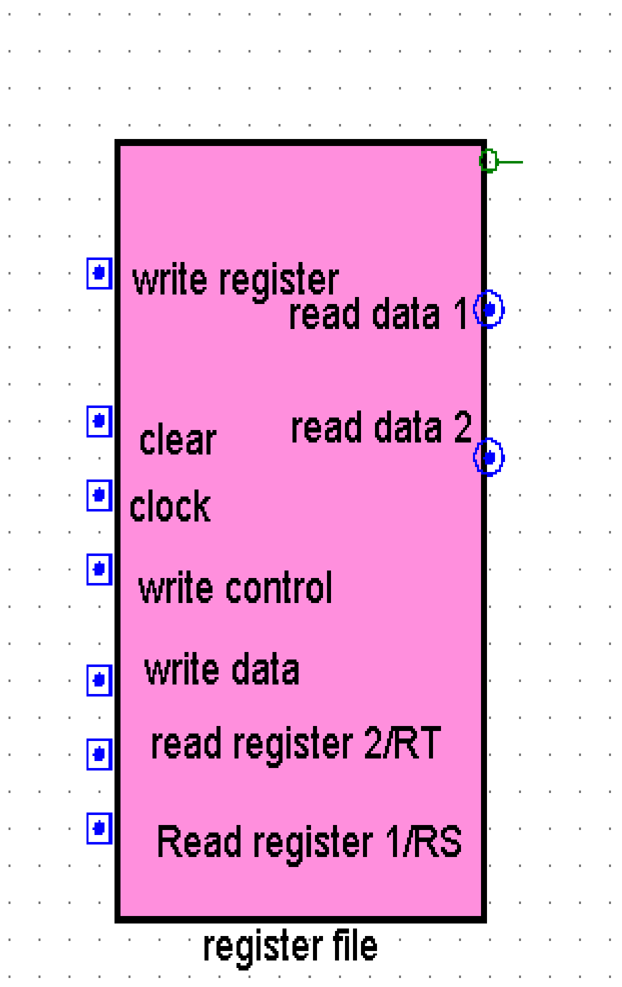


Fig: Subcircuit of register file

Description : Register File

- **Write Register**: Specifies the destination register for writing data.
- **Read Register 1 (rs)**: Address of the first source register to read from.
- **Read Register 2 (rt)**: Address of the second source register to read from.
- **Write Data**: Data to be written into the write register.
- **Read Data 1**: Output data from the first read register.
- **Read Data 2**: Output data from the second read register.
- **Write Control**: Enables or disables writing to the register file.
- **Clock**: Synchronizes the writing operation.
- **Clear**: Resets all register values to default (usually zero)

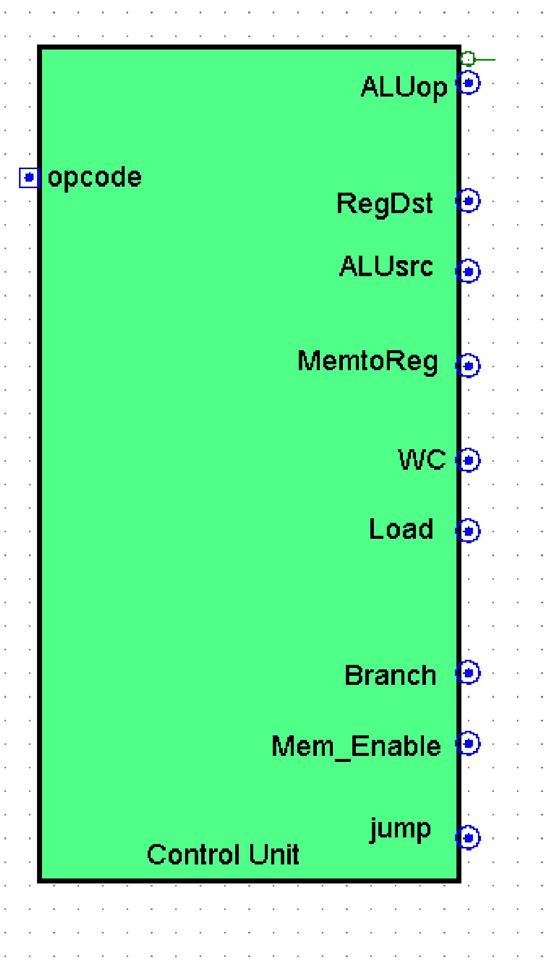


Fig: Subcircuit of control unit

Description : Control Unit

- **Opcode**: Input that determines the type of instruction.
- **RegDst**: Selects the destination register (rt or rd).
- **ALUSrc**: Chooses between register or immediate value as second ALU operand.
- **MemtoReg**: Controls data selection for writing back to registers (from memory or ALU).
- **Write Control (WC)**: Enables writing to the register file.
- **Load**: Activates memory read operation.
- **Branch**: Enables conditional branching.
- **MemEnable**: Enables memory access.
- **Jump**: Controls jump instruction execution.
- **ALUOp**: Determines the type of ALU operation to perform.

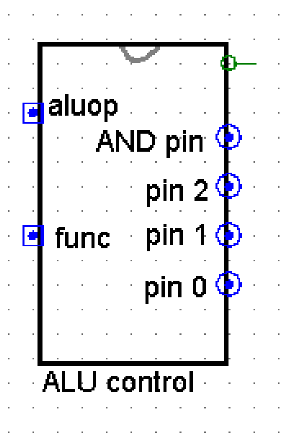


Figure: Subcircuit of alu control

Description: ALU Control

- **ALUOp**: Input from control unit indicating instruction type (e.g., R-type, I-type).
- **Func**: Function field from instruction (used for R-type instructions).
- **Pin0, Pin1, Pin2, Pin:** Outputs that generate the correct ALU operation signal based on ALUOp and Func.

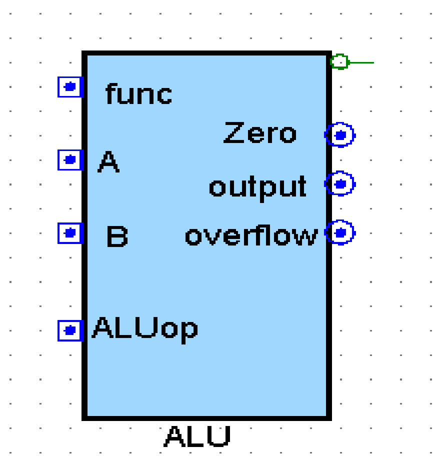
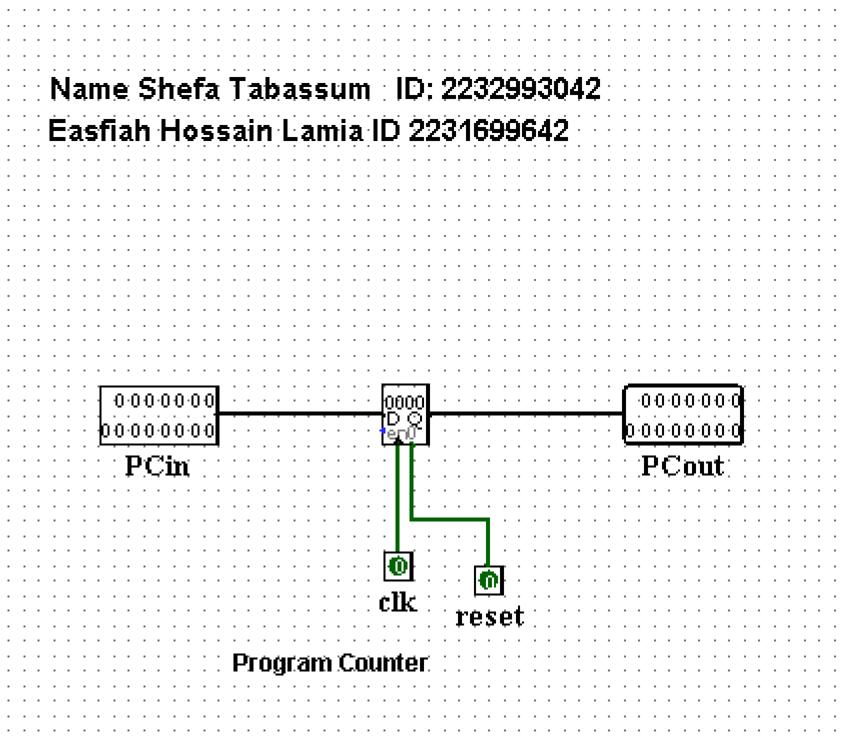


Fig: Subcircuit of ALU

Description: ALU

- **A**: First input operand to the ALU.
- **B**: Second input operand to the ALU.
- **ALUOp / Function**: Specifies the operation to be performed (e.g., add, subtract).
- **Output**: Result of the ALU operation.
- **Zero Flag**: Set when the ALU output is zero (used in branch decisions).
- **Overflow**: Indicates arithmetic overflow during operation.



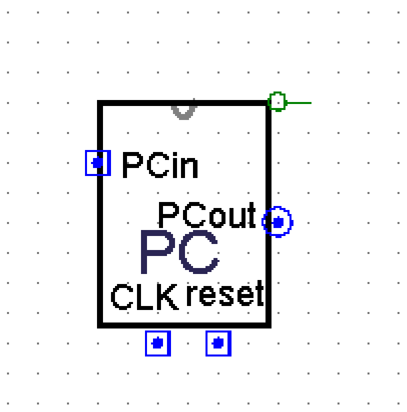


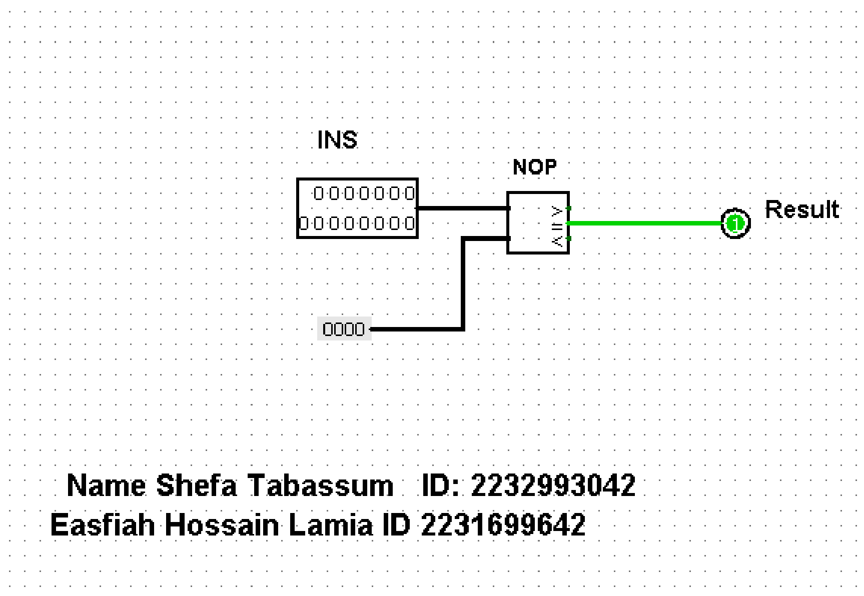
Fig: Program Counter

Description : Program Counter

- **PCin**:
  This is the **input** to the Program Counter. In the datapath, it carries the **next instruction address**, selected from a multiplexer (MUX). The MUX chooses between: A **jump address**, or a value selected from another MUX (branch address or PC + 1).
  So, PCin determines what value the PC will update to on the next clock pulse.
- **PCout**:
  This is the **output** of the Program Counter. It holds the **current instruction address**, which will be used to fetch the next instruction from memory.
- **clk (Clock)**:
  This signal **controls the timing** of when the Program Counter updates. On

each rising edge of the clock, if reset is not active, the PC takes the value from PCin and sends it to PCout.

- **reset**:
  When this signal is active (usually high), it **resets the Program Counter to zero**, regardless of the value on PCin. This is useful at the start of program execution or after a system restart.


INS
0000000
00000000
NOP
0000
Result
Name Shefa Tabassum ID: 2232993042
Easfiah Hossain Lamia ID 2231699642

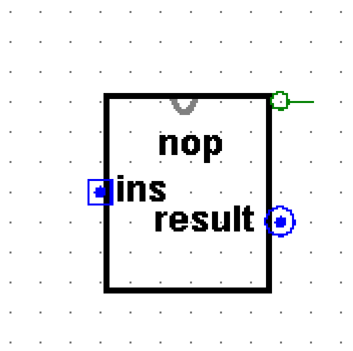


Figure: NOP

Description: NOP

- **Instruction Input**: Takes 15-bit binary 000000000000000 (NOP instruction), which has **opcode = 000** and **func = 000** — same as R-type format.
- **Comparator**: Inputs:
  - First input A = 15-bit instruction.
  - Second input = constant 15-bit 0.
- **Result = 1** if both match, indicating a NOP.

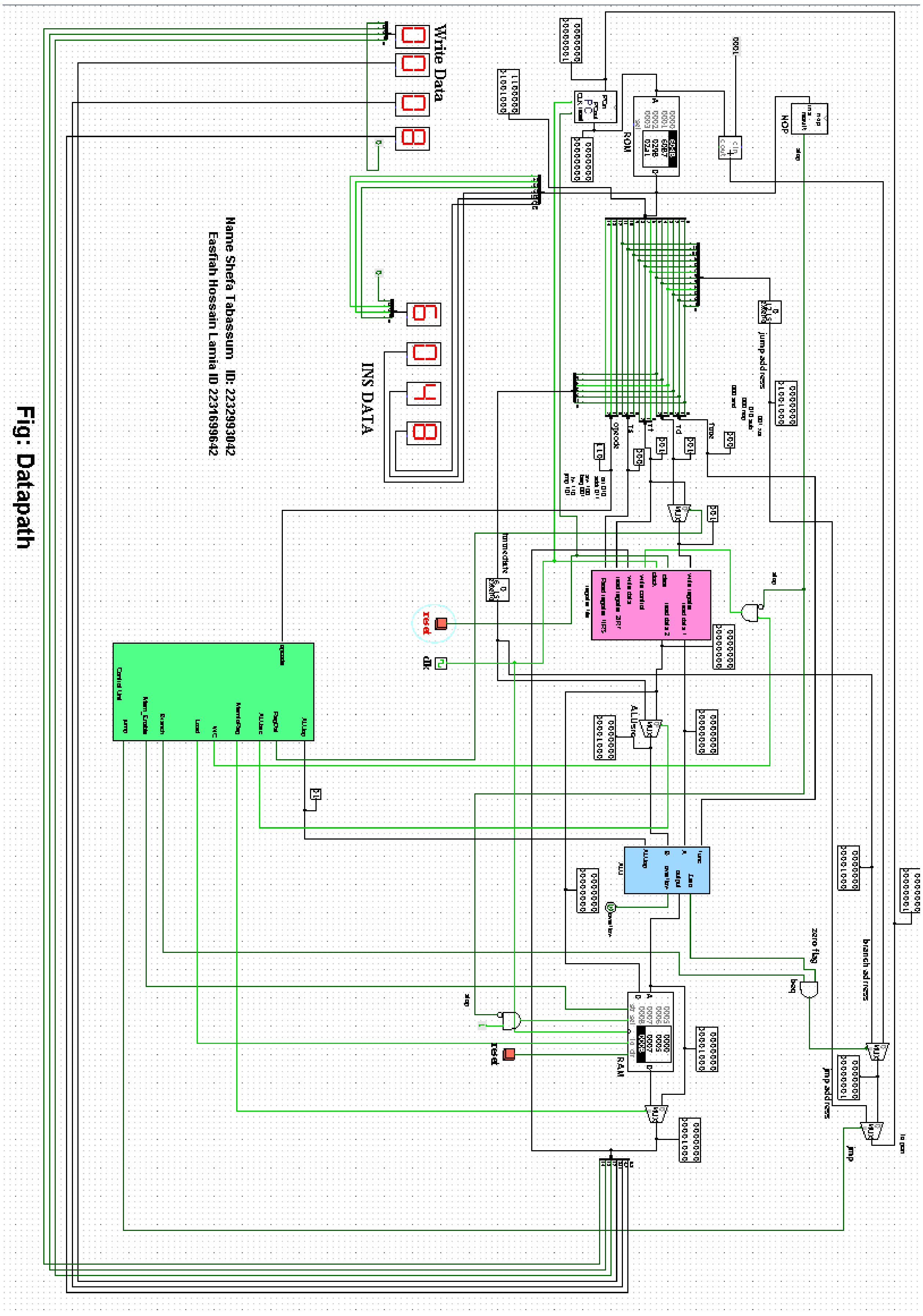


Figure: Datapath

Desccription of Datapath

This single-cycle datapath is capable of executing R-type, I-type, branch, and jump instructions within a single clock cycle. It integrates key components including the Program Counter (PC), Read-Only Memory (ROM), Register File, Arithmetic Logic Unit (ALU), Random Access Memory (RAM), and the Control Unit. Together, these modules form the backbone of the processor, allowing instructions to be fetched, decoded, executed, and their results stored, all in one clock cycle.

Instruction execution begins with the PC fetching a 15-bit instruction from ROM. The instruction is then divided into its fundamental fields: opcode, source registers (rs and rt), destination register (rd), immediate value, and function code (funct). The opcode is sent to the Control Unit, which generates the necessary control signals for the rest of the datapath. The funct field is particularly relevant for R-type instructions, where it is used along with the ALUOp signal to determine the specific ALU operation.

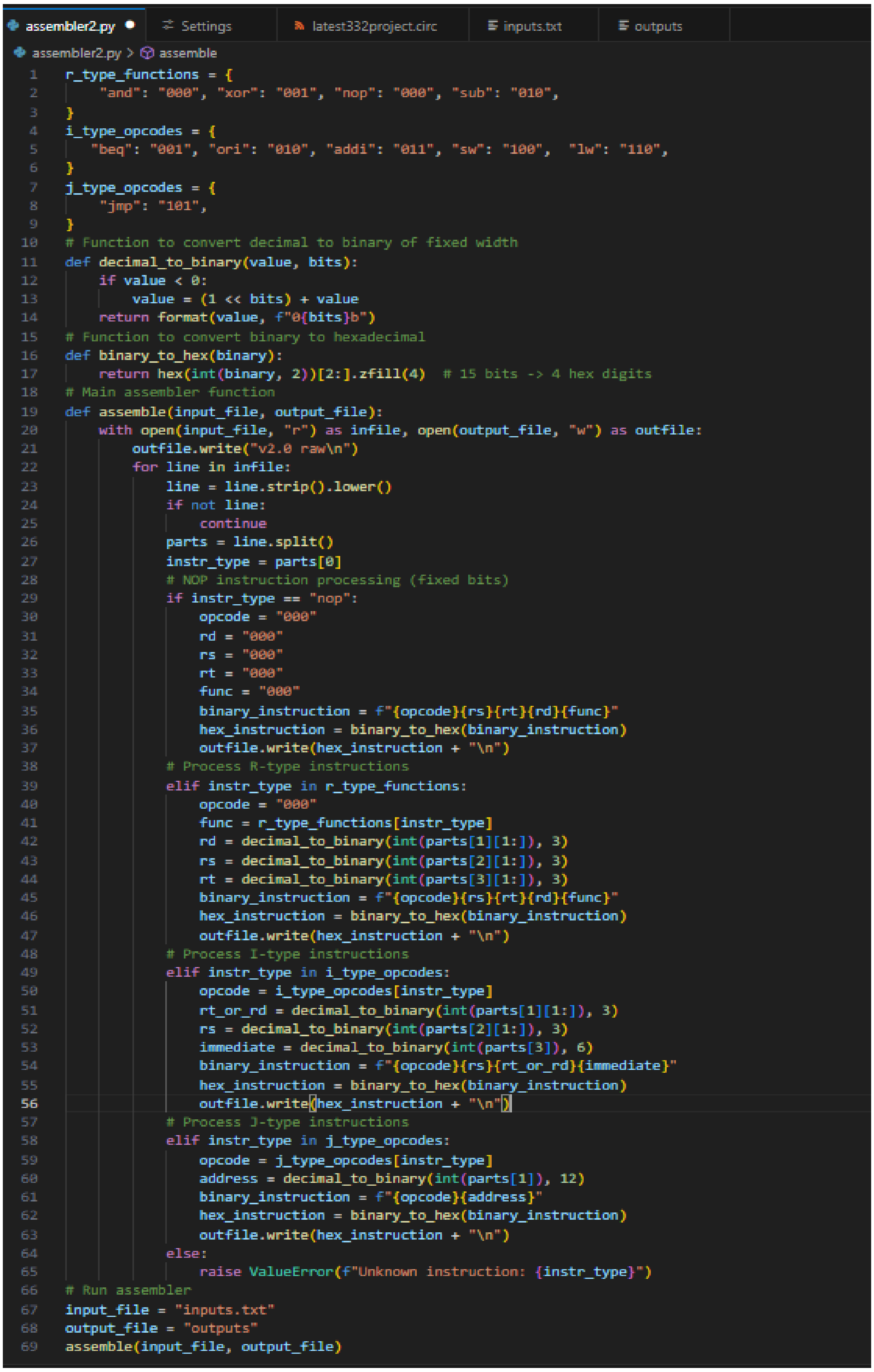
The ALU performs arithmetic and logical operations based on the control signals it receives from the ALU Control Unit. Two multiplexers are used to select the inputs to the ALU. One chooses between the second source register and the sign-extended immediate value, depending on whether the instruction is R-type or I-type. The second multiplexer determines which register will receive the output, selecting between the rd and rt fields. The ALU outputs both the result of the operation and status flags, such as the Zero flag, which is crucial for evaluating branch instructions.

The Register File reads data from the source registers indicated by the rs and rt fields and writes data back to the destination register based on the control logic. The write-back data can come either from the ALU or from RAM, depending on the instruction type. Memory access is governed by control signals as well. If MemWrite is active, data from a register is written to RAM at the address produced by the ALU. If MemRead is active, data is retrieved from RAM and can be written back to the register file.

A NOP (No Operation) detection mechanism is integrated to ensure system stability when a NOP instruction is encountered. If the instruction matches a predefined NOP pattern, a comparator activates logic that disables register writing and memory access for that cycle. This is achieved by logically ANDing the RegWrite and memory control signals with the inverse of the NOP detection signal, effectively preventing any change in system state during a NOP.

Control flow is handled through updates to the Program Counter. In normal execution, the PC is simply incremented to point to the next instruction. In the case of branch instructions, the Zero flag from the ALU is evaluated, and if the condition is met, the PC is updated by adding the sign-extended immediate offset. For jump instructions, a separate path allows the PC to be set directly to the jump target, overriding the sequential flow of execution.

This architecture ensures that all instruction types are processed efficiently within one clock cycle, maintaining both performance and simplicity in design.



```python
r_type_functions = {
    "and": "000", "xor": "001", "nop": "000", "sub": "010",
}
i_type_opcodes = {
    "beq": "001", "ori": "010", "addi": "011", "sw": "100",  "lw": "110",
}
j_type_opcodes = {
    "jmp": "101",
}
# Function to convert decimal to binary of fixed width
def decimal_to_binary(value, bits):
    if value < 0:
        value = (1 << bits) + value
    return format(value, f"0{bits}b")
# Function to convert binary to hexadecimal
def binary_to_hex(binary):
    return hex(int(binary, 2))[2:].zfill(4)  # 15 bits -> 4 hex digits
# Main assembler function
def assemble(input_file, output_file):
    with open(input_file, "r") as infile, open(output_file, "w") as outfile:
        outfile.write("v2.0 raw\n")
        for line in infile:
            line = line.strip().lower()
            if not line:
                continue
            parts = line.split()
            instr_type = parts[0]
            # NOP instruction processing (fixed bits)
            if instr_type == "nop":
                opcode = "000"
                rd = "000"
                rs = "000"
                rt = "000"
                func = "000"
                binary_instruction = f"{opcode}{rs}{rt}{rd}{func}"
                hex_instruction = binary_to_hex(binary_instruction)
                outfile.write(hex_instruction + "\n")
            # Process R-type instructions
            elif instr_type in r_type_functions:
                opcode = "000"
                func = r_type_functions[instr_type]
                rd = decimal_to_binary(int(parts[1][1:]), 3)
                rs = decimal_to_binary(int(parts[2][1:]), 3)
                rt = decimal_to_binary(int(parts[3][1:]), 3)
                binary_instruction = f"{opcode}{rs}{rt}{rd}{func}"
                hex_instruction = binary_to_hex(binary_instruction)
                outfile.write(hex_instruction + "\n")
            # Process I-type instructions
            elif instr_type in i_type_opcodes:
                opcode = i_type_opcodes[instr_type]
                rt_or_rd = decimal_to_binary(int(parts[1][1:]), 3)
                rs = decimal_to_binary(int(parts[2][1:]), 3)
                immediate = decimal_to_binary(int(parts[3]), 6)
                binary_instruction = f"{opcode}{rs}{rt_or_rd}{immediate}"
                hex_instruction = binary_to_hex(binary_instruction)
                outfile.write(hex_instruction + "\n")
            # Process J-type instructions
            elif instr_type in j_type_opcodes:
                opcode = j_type_opcodes[instr_type]
                address = decimal_to_binary(int(parts[1]), 12)
                binary_instruction = f"{opcode}{address}"
                hex_instruction = binary_to_hex(binary_instruction)
                outfile.write(hex_instruction + "\n")
            else:
                raise ValueError(f"Unknown instruction: {instr_type}")
# Run assembler
input_file = "inputs.txt"
output_file = "outputs"
assemble(input_file, output_file)
```

Figure: assembler.py

Explanation of the assembler:
We modified the provided sample Python-based assembler to match our custom Instruction Set Architecture (ISA). It reads assembly instructions from inputs.txt, converts them into 15-bit binary machine code based on instruction type (R, I, or J), and writes the output as 4-digit hexadecimal values in a v2.0 raw format to a file named outputs.

The code uses dictionaries to map opcodes and function codes for different instruction types. Helper functions handle binary conversion and two's complement for negative values. I adjusted opcode mappings, field widths, and instruction formats to fit my ISA. This helped me better understand how real processors translate human-readable instructions into machine code.

Discussion

The design process for this 15-bit processor was an interesting and challenging experience. One of the most important aspects was making sure that the control unit worked properly, generating the correct control signals based on the opcode and function code. This was essential for ensuring the correct execution of different instruction types. Another key challenge was the handling of the 6-bit signed immediate in I-type instructions. We had to extend this value to fit the 15-bit datapath, which was necessary for correct ALU and memory operations

One of the main limitations of this single-cycle design is that all instructions are executed in one clock cycle. While this simplifies the design, it means that every instruction, regardless of complexity, must complete in the same amount of time. This can make the processor less efficient for more complex operations. In the future, a multi-cycle or pipelined design could improve performance by allowing different parts of the processor to work on different stages of instructions at the same time.

Overall, this project gave us a solid understanding of how processors work at the hardware level. It also highlighted some of the trade-offs in processor design, especially between simplicity and performance. While this processor is a basic design, it provides a great foundation for learning about more advanced processor architectures.

Conclusion

In this project, we successfully designed and implemented a custom 15-bit processor with a single-cycle datapath. The processor supports R-type, I-type, branch, jump, and NOP instructions. By creating a control unit that decodes the opcode and function code, the processor can correctly execute various types of instructions while keeping the design simple and efficient. The inclusion of NOP detection, immediate extension, and proper memory access handling ensures the processor operates correctly in all situations. Despite the simplicity of the design, it provides a clear example of how processors handle instruction fetching, decoding, and execution.