Project: 20-bit single cycle MIPS CPU

Course: CSE 332 - Computer Organization & Design

Section – 04

Summer 2024

Group- 11

Mohammed Arif Mainuddin (2211577042)

Najifa Tabassum (2211578042)

Contents

Introduction

Design a 20-bit MIPS CPU

Objective

Designing a 20-bit MIPS CPU based on ISA with particular operations following R-type, I-type, J-type format

How many operands

There are three main operands named as `d, s, t`. Represented as:

1. Rd = Destination Register
2. Rs = Source Register
3. Rt = Temporary Register (Source Register2)

Besides immediate, address (j-type), shamt – Shift Amount.

Type of Operands

1. Register-based: Operands are stored in registers.
2. Memory-based: Operands are fetched from memory.

How many Operations

There are 7 types of operations.

Types of Operations

Categories of 7 type of operations are:

1. Arithmetic Operations
2. Logical Operations
3. Conditional /Unconditional Operations
4. Shift Operations
5. Memory Operations
6. Comparison Operations
7. No Operation

Category and Sequence wise

Table 1: Name and Category of the instructions

Category	Operation	Name	opcode	Type/Format
	No Operation	NOP	000	
Logical	Nor	NOR	000	R
Conditional	Branch On Equal	BEQ	001	I
Logical	Bitwise OR	OR	000	R
Arithmetic	Add	ADD	000	R
Conditional	Branch on Not Equal	BNE	010	I
Shift	Shift Right Logical	SRL	000	R
Memory	Store Word	SW	011	I
Comparison	Set on less than	SLT	000	R
Shift	Shift Left Logical	SLL	000	R
Arithmetic	Subtract	SUB	000	R
Unconditio-nal	Jump	JMP	100	J
Logical	Bitwise AND	AND	000	R
Memory	Load word	LW	101	I
Arithmetic	Add immediate	ADDi	110	I

No. of formats of instruction (how many different formats)

There are 3 type of format in ISA

1. Register type – (R-Type)
2. Immediate Type – (I- Type)
3. Jump Type – (J- Type)

Describe each of the format (field and field length)

- R-type (ISA format)

opcode	rs	rt	rd	shamt	function
3 bit	4 bit	4 bit	4 bit	2 bit	3 bit

- I-type (ISA format)

opcode	rs	rd	shamt	function
3 bit	4 bit	4 bit	2 bit	3 bit

- J-type (ISA format)

opcode	Address
3 bit	17 bit

Control Signals

Instructions	RegDst	ALUSrc	Memto-Reg	RegWrite	MemRead	MemWrite	Branch	Jump	ALUOp	Opcode
Nop	0	0	0	0	0	0	0	0	xx	000
NOR	1	0	0	1	0	0	0	0	10	000
BEQ	0	0	0	0	0	0	1	0	00	001
OR	1	0	0	1	0	0	0	0	10	000
ADD	1	0	0	1	0	0	0	0	10	000
BNE	0	0	0	0	0	0	1	0	××	010
SRL	1	0	0	1	0	0	0	0	10	000
SW	0	1	0	0	0	1	0	0	01	011
SLT	1	0	0	1	0	0	0	0	10	000
SLL	1	0	0	1	0	0	0	0	10	000
SUB	1	0	0	1	0	0	0	0	10	000
JMP	0	0	0	0	0	0	0	1	xx	100
AND	1	0	0	1	0	0	0	0	10	000
LW	0	1	1	1	1	0	0	0	01	101
ADDi	0	1	0	1	0	0	0	0	01	110

ALU Control Signals

Instructions	Function	ALUOp	Pin 0	Pin 1	Pin 2
NOR	000	10	0	0	0
OR	001	10	1	0	0
ADD	010	10	0	1	0
LW/SW/ADDi		01	0	1	0
SRL	011	10	1	1	0
SLT	100	10	0	0	1
SLL	101	10	1	0	1
BEQ		00	0	1	1
AND	111	10	1	1	1
SUB	110	10	0	1	1

Here, R-type has fixed opcode 000, fixed ALUOp 10 because R-type instructions will be identified according to theirs functions. LW, SW, Addi have similar ALUOp 01 because they are I-type instructions, and ALU need to perform the addition for each of them. BNE, BEQ have identical ALUOp 10, because they are I-type instructions and ALU need to perform the subtraction for each of them. We left the ALUOp 11 blank, because we

do not need assign 11 bit to any of the assigned instructions.

Instruction Assembly Convention during input

R-type instructions: NOR, OR, ADD, SRL, SLT, SLL, SUB, AND

	Instructions	rd	rs	rt	shamt
Example 1	sub	r1	r2	r3	
Example 2	srl	r5	r2		2

I-type instructions: LW, SW, BEQ, BNE, ADDi

	Instructions	rd	rs	immediate
Example 1	lw	r1	r0	5
Example 2	addi	r5	r0	2

J-type instructions: JMP

c	Instructions	Address
Example 1	jmp	5

NOP Operation

NOP instruction focuses on ensuring the control unit disables register/memory writes. So we take 20 bit 0 as input for both A and compare it using a comparator where another input is default 0 , if two inputs matches output will be 1, we took the output and implement a basic AND operation where one input is from NOP operation connected via a NOT gate another input is RegWrite , if NOP operation output is 1 then AND gate output will be 0 Regwrite will get a 0 so it will perform NO Operation (opcode for both NOP and R-type is 000 so RegWrite will be always 1 as it is also working for R-type difference will make by NOP output). Similarly, a basic AND gate used in Data Memory part as Data Memory has a feature named disables component which will be activate while getting zero , therefore two inputs of AND gate was a default 1 and output of NOP via not gate. As a result when NOP will be activated output will be 1

but for AND gate via NOT gate it will be 0, finally AND gate output will be 0 which will activate the Data Memory disables component and Data Memory will perform NO Operation as a result no changes will be seen and program counter will go for fetching next instruction.

	Instructions
Example 1	nop

Screenshot of Logisim

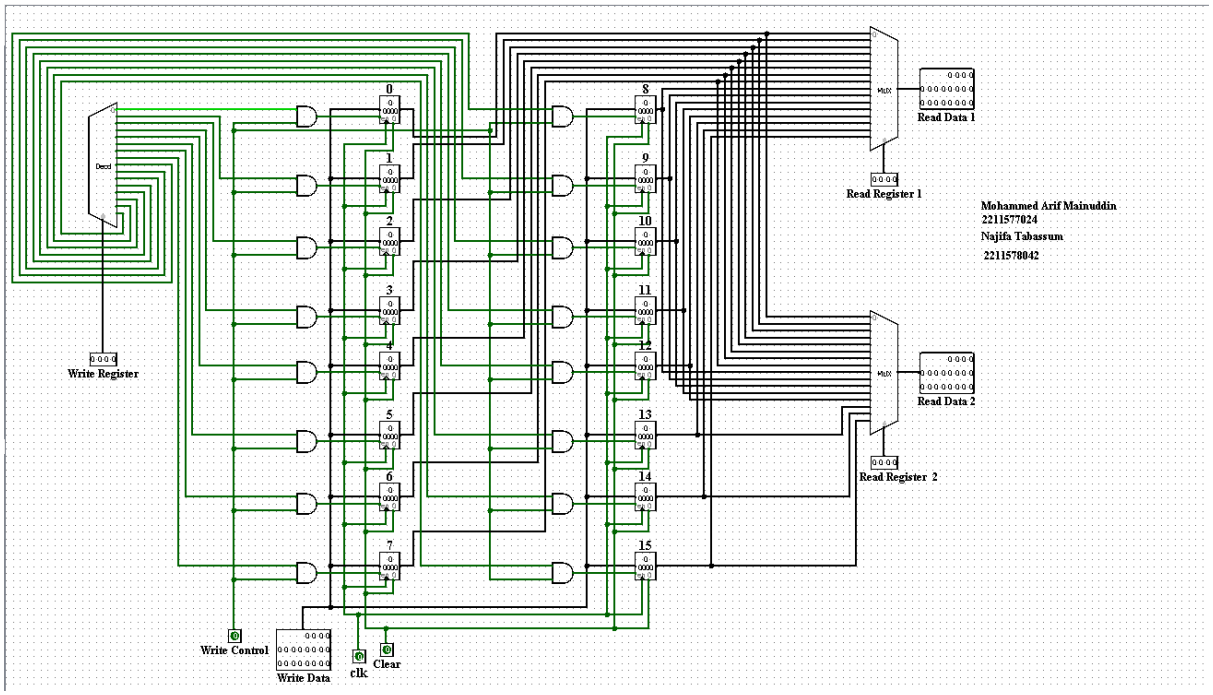

Figure 1: 20 bit register

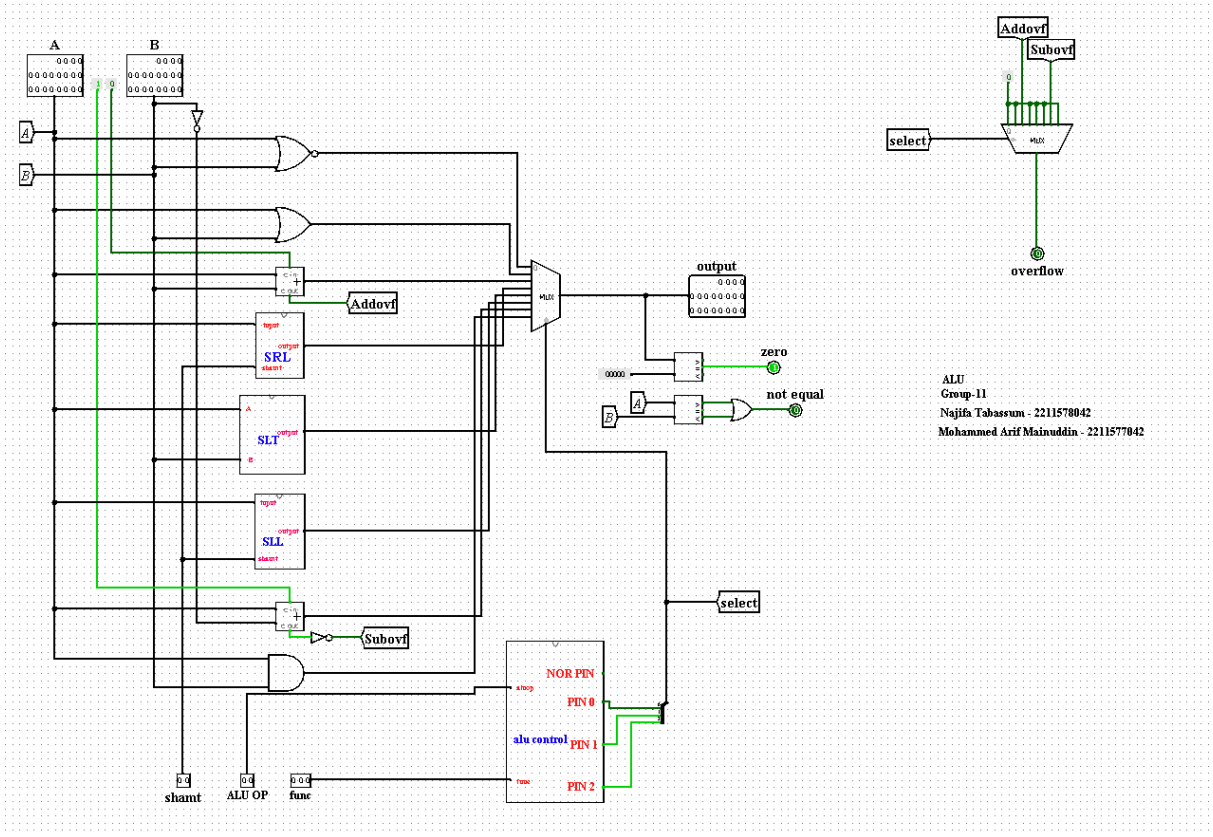

Figure 2: ALU (main)

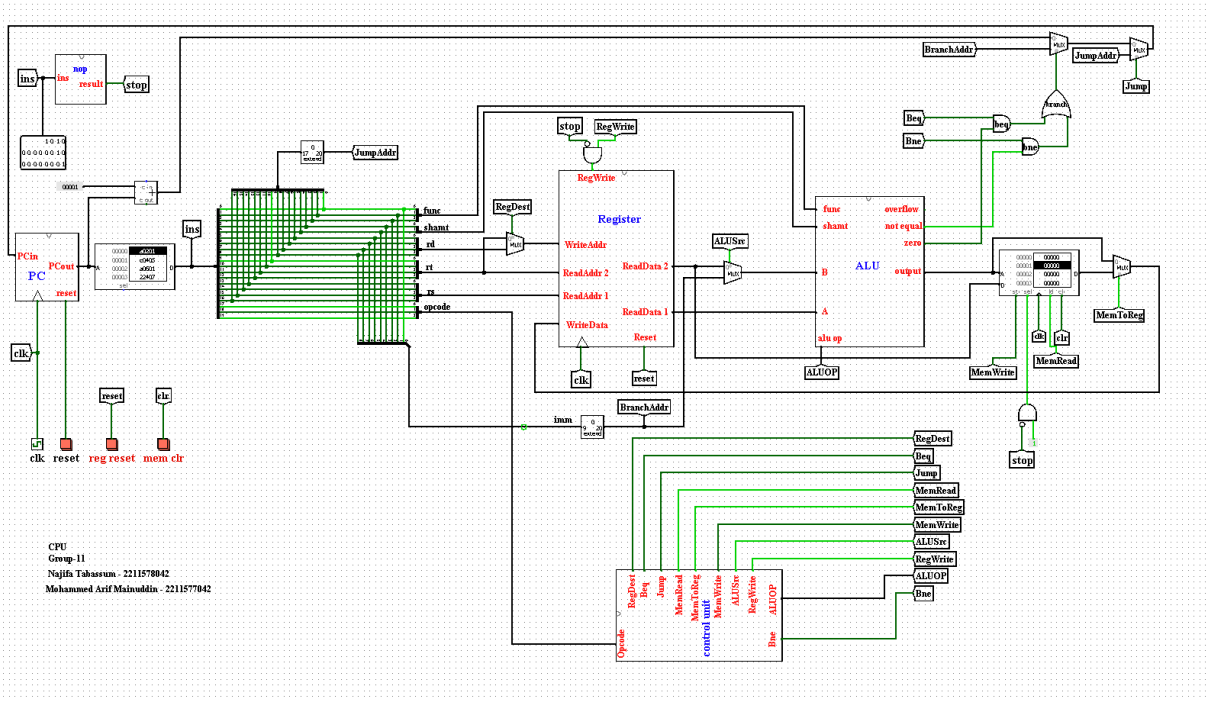

Figure 3: 20 bit CPU

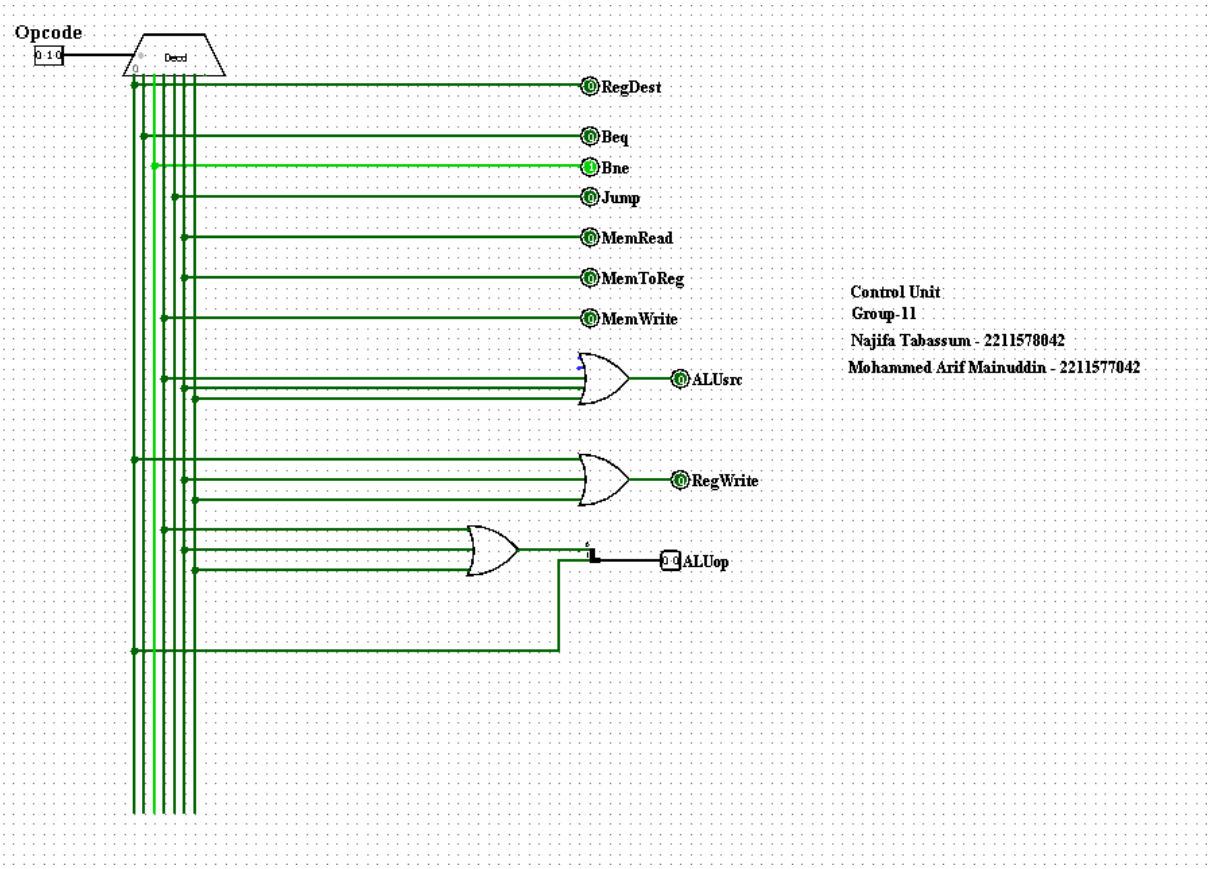

Figure 4: Control Unit

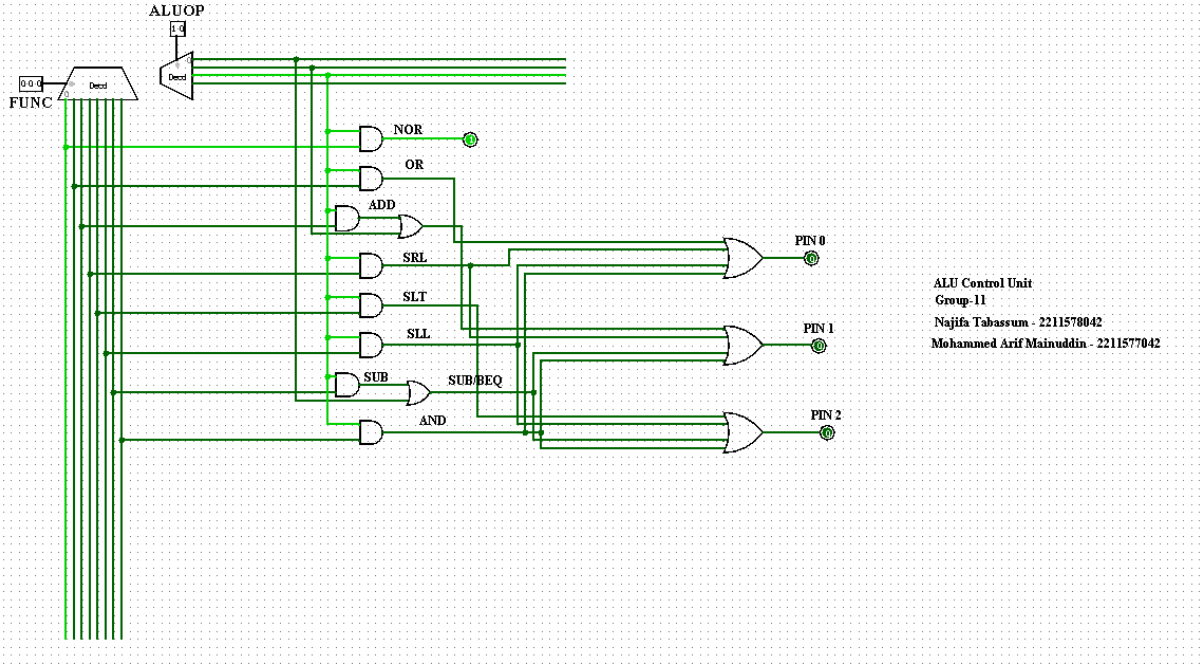

Figure 5: ALU Control Unit

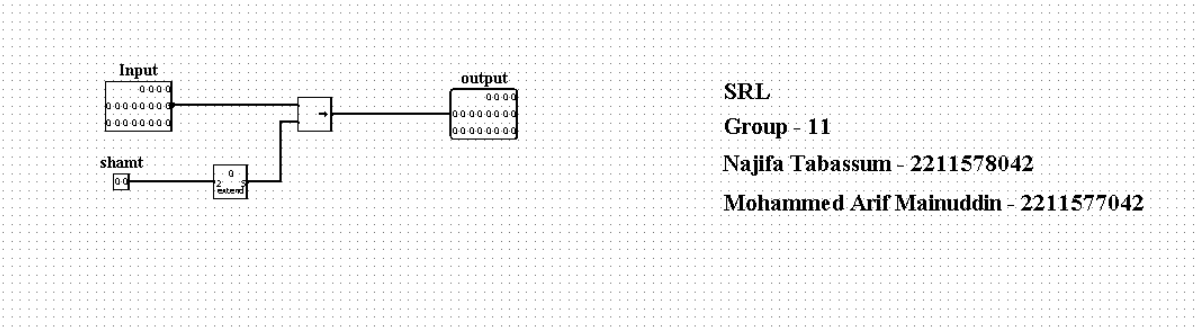

Figure 6: SRL

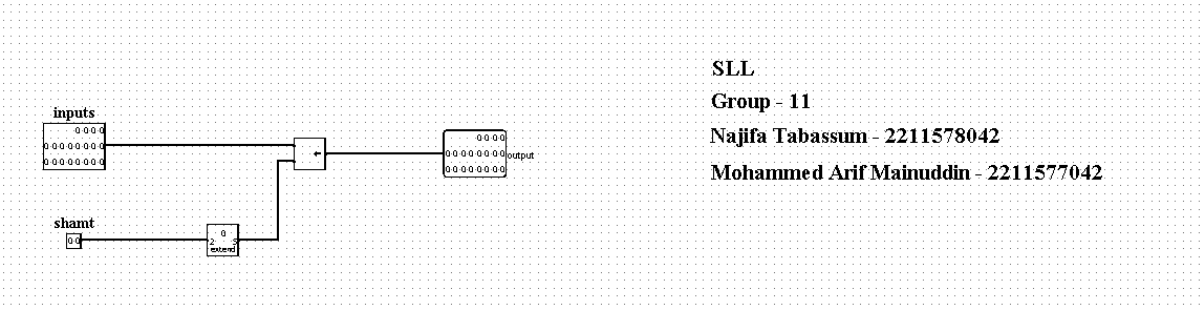

Figure 7: SLL

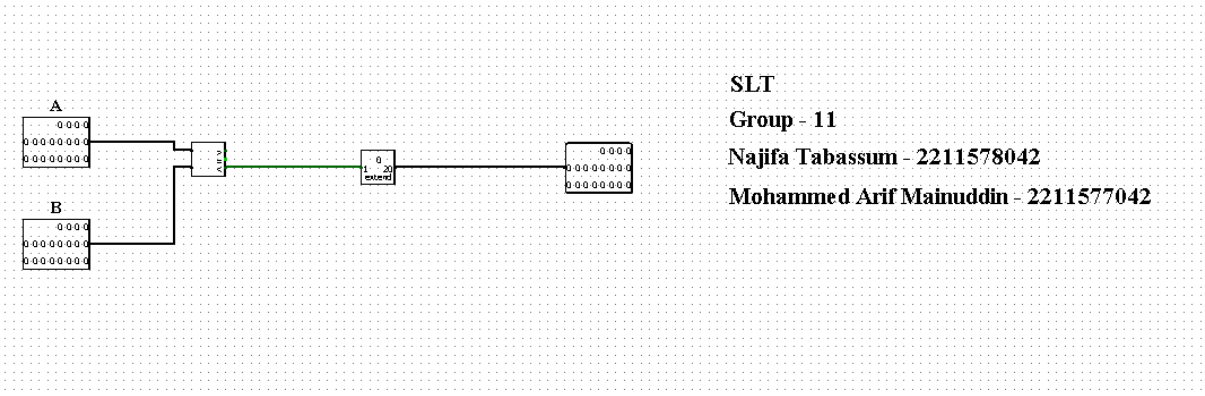

Figure 8: SLT

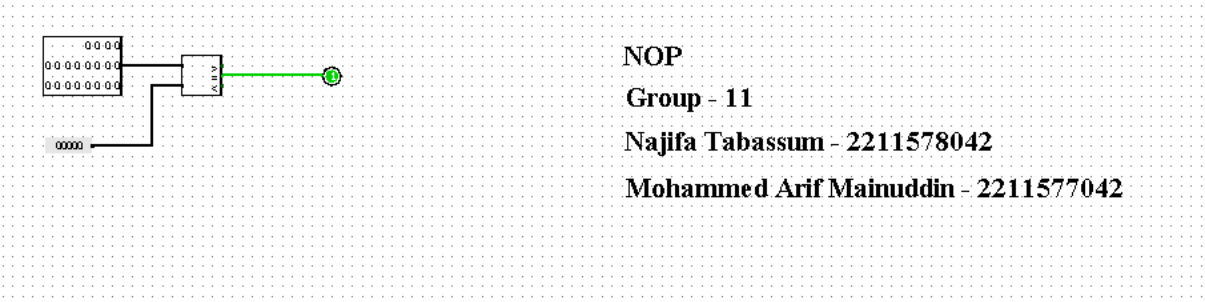

Figure 9: NOP

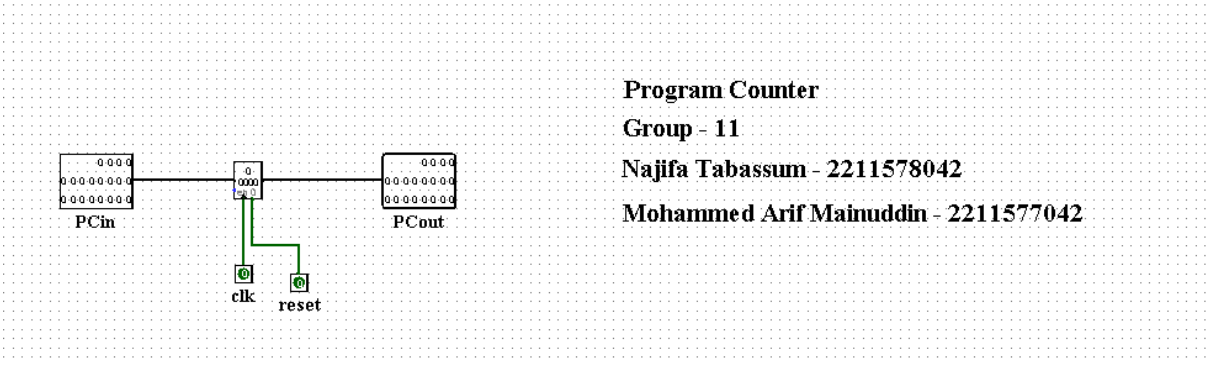

Figure 10: Program Counter

```python
# Define dictionaries for function and opcode mappings
r_type_functions = {
    "nor": "000",
    "or": "001",
    "add": "010",
    "srl": "011",
    "slt": "100",
    "sll": "101",
    "sub": "110",
    "and": "111",
}

i_type_opcodes = {
    "beq": "001",
    "bne": "010",
    "sw": "011",
    "lw": "101",
    "addi": "110",
}

j_type_opcodes = {
    "jmp": "100",
}

# 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(5)  # 20 bits -> 5 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 = "0000"
                rs = "0000"
                rt = "0000"
                shamt = "00"
                func = "000"
                binary_instruction = f"{opcode}{rs}{rt}{rd}{shamt}{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"
                #if it is srl or sll
                if instr_type == 'srl' or instr_type == 'sll':
                    func = r_type_functions[instr_type]
                    rd = decimal_to_binary(int(parts[1][1:]), 4)
                    rs = decimal_to_binary(int(parts[2][1:]), 4)
                    rt = "0000"
                    shamt = decimal_to_binary(int(parts[3]),2) if len(parts) > 3 else "01"

                # if it is not srl or sll
                else:
                    func = r_type_functions[instr_type]
                    rd = decimal_to_binary(int(parts[1][1:]), 4)
                    rs = decimal_to_binary(int(parts[2][1:]), 4)
                    rt = decimal_to_binary(int(parts[3][1:]), 4)
                    shamt = "00"
                # if len(parts) > 4:
                #     shamt = decimal_to_binary(int(parts[4]),2)
                # else:
                #     shamt = "00"
                binary_instruction = f"{opcode}{rs}{rt}{rd}{shamt}{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:]), 4)
                rs = decimal_to_binary(int(parts[2][1:]), 4)
                immediate = decimal_to_binary(int(parts[3]), 9)
                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]), 17)
                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 = "input.txt"
output_file = "output.txt"
assemble(input_file, output_file)
```

Figure 11: Assembler.py

Sample Instructions in Assembly

```
addi r2 r0 10
addi r3 r0 20
sub r1 r2 r3
nop
sll r3 r2 1
sw r1 r0 1
lw r4 r0 1
```

Figure 12: Some Sample Instructions in Assembly Language