Report on miniCPU Project

Mingxuan Li [2025533130] Ruiyang Xu [2025531125] Kaiwen Yang [2025531126]

1 Task Description

miniCPU Design and Implementation is a cutting-edge educational initiative focused on computer architecture.

In this project, our work revolves around building the datapath based on `Logisim`, developing an assembler in `Python`, and writing test cases.

2 Team's Timeline

2.1 Week 1

In week 1, we gained preliminary understanding of the concept of electrical level and sequential logic, and have studied the structure of registers. We also preliminarily developed a Python-based assembler.

2.2 Week 2

In week 2, we studied the fundamentals of computer architecture, including learning RISC-V instruction set and gaining an understanding of the datapath of a single-cycle CPU. Simultaneously, we refined our assembler, enabling it to read assembly files in the terminal, convert assembly instructions into hexadecimal machine code, and output the results via a text file.

2.3 Week 3

In week 3, we implemented a single-cycle CPU in Logisim capable of executing 13 RV32I instructions: `add, addi, sub, and, andi, or, ori, slt, slti, lw, sw, bne` and `jal`.

2.4 Week 4

In week 4, we designed basic test cases and conducted debugging and modifications on the CPU.

2.5 Week 5

In week 5, we extended our previous basic CPU to support more instructions, including the entire RV32I instruction set and RV32M instruction set.

2.6 Week 6

In week 6, we designed additional test cases and performed debugging and modifications on the CPU to ensure its correctness and stability. Finally, we completed this report.

3 CPU Implementation

3.1 Overall Design Plan

- **Instructions Set**

Type	Instructions
R	`add, sub, and, or, slt, sll, srl, sra, xor, sltu`
I1	`addi, andi, ori, slti, lw, xori, sltui, lb, lh, lbu, lhu, jalr`
I2	`slli, srli, srai`
S	`sw, sb, sh`
B	`bne, beq, blt, bltu, bge, begu`
J	`jal`
U	`lui, auipc`
M(R)	`mul, mulh, mulsu, mulu, div, divu, rem, remu`

- **Modules Design**
  - *Instruction Fetch*: IFU includes the PC, NPC, and IM, where the PC register has an asynchronous reset function with a starting address of 0x00000000.
  - *Instruction Memory*: IM is a read-only memory module that stores the machine code instructions in hexadecimal format generated by the assembler.
  - *Register File*: RF contains 32 registers, each 32 bits wide. It supports two asynchronous read ports and one synchronous write port, and has an asynchronous reset function that initializes all registers to 0.
  - *ALU*: ALU performs arithmetic and logical operations, supporting operations such as addition, subtraction, bitwise `AND`, bitwise `OR`, and set less than.
  - *Data Memory*: DM is a synchronous read-write memory module. It has an asynchronous reset function that initializes all memory locations to 0.
  - *Extend*: The extend module is responsible for extension of immediate values.
  - *SProcessor*: This module handles the processing of store instructions.
  - *LProcessor*: This module handles the processing of load instructions.
  - *AddrProcessor*: This module handles the control of store data address.
  - *Control Unit*: The control unit generates control signals for each module and data path based on the opcode and funct3/funct7 fields of the instruction.

- **Overall Structure**

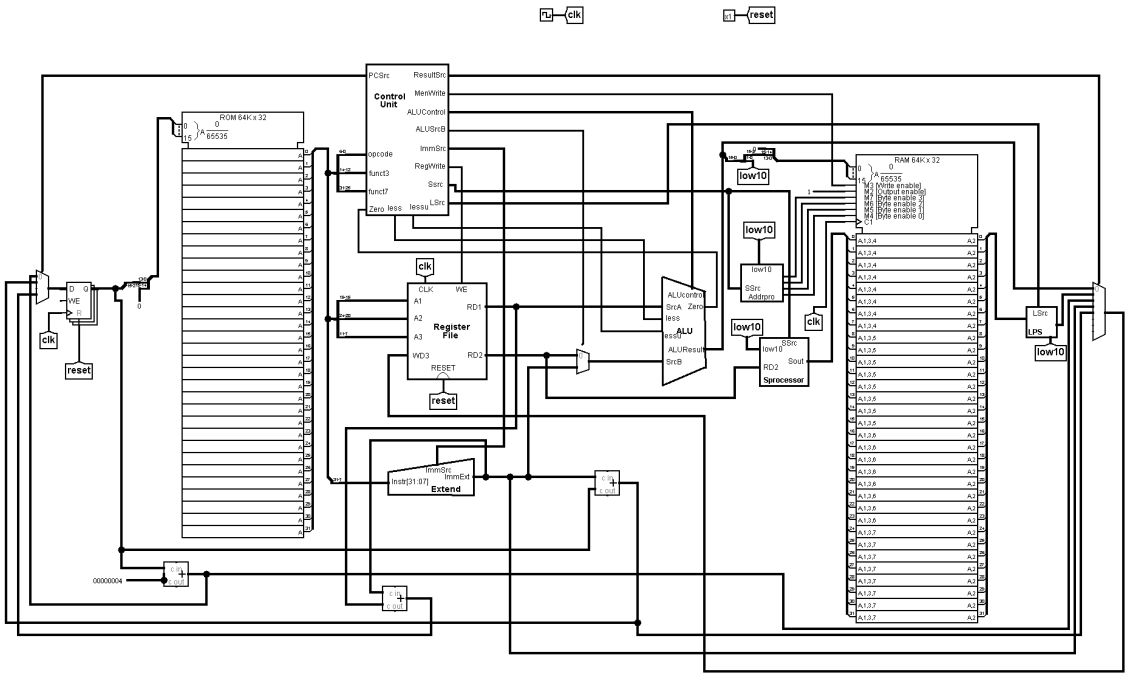

Figure 1: The overall structure

- **Key Modules**

1. *PC*

Port	**Description**
clk	Receives the clock signal (rising edge triggered).
reset	Receives an asynchronous reset signal, which sets the current instruction to the starting position when it is $1'b1$.
D	Receives the next instruction address.
Q	Outputs the current instruction address.

Table 1: PC Module Ports

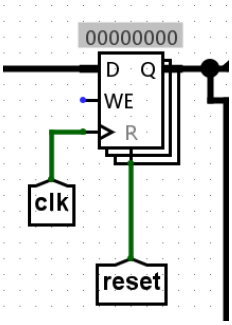

Figure 2: PC Module Implementation

2. *Instruction Memory*

   We use a ROM as the instruction memory.

   The physical address space of the instruction memory (i.e., the actual address range connected to the ROM) is 0x0000 to 0xFFFF (a total of 64KB of space). Since the Program Counter (PC) is 32 bits wide, but the instruction ROM only supports a 16-bit address input (a 32-bit address would require excessively large memory and is unnecessary), we only use the lower 16 bits of the PC (PC[15:0]) as the access address for the ROM. Each instruction is 32 bits wide (4 bytes), but the memory is word-addressable (32 bytes per word), where each address location stores 32 bits. Therefore, the address of the next instruction in this ROM is the current address plus 1.

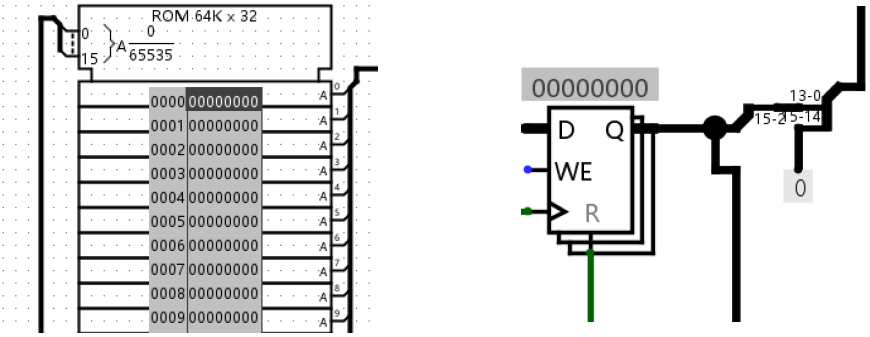

Figure 3: Instruction Memory Addressing Method

3. *Register File*

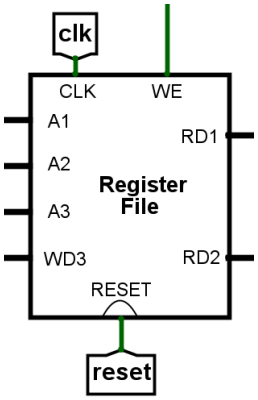

Figure 4: Register File Overall Structure

Port	Description
A1	Address of the first read register.
A2	Address of the second read register.
A3	Address of the write register.
WD	Data to be written to the write register.
RD1	Data output from the first read register.
RD2	Data output from the second read register.
WE	Write enable signal. When $1'b1$, data is written to the register A3.
clk	Receives the clock signal (rising edge triggered).
reset	Receives an asynchronous reset signal.

Table 2: Register File Module Ports

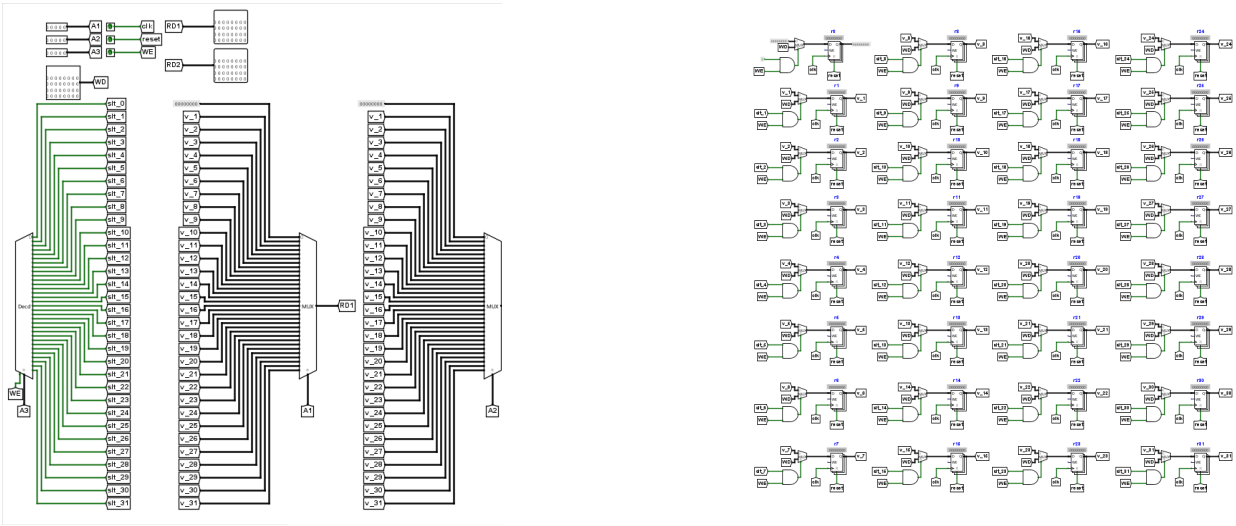

Figure 5: Register File Internal Structure

The specific implementation idea of a Register File is relatively straightforward. The `slt` signal, obtained after decoding the $A3$ input signal via a Decoder, represents the selection signal for each individual register cell. v denotes the value stored in each register cell. The $A1$ and $A2$ input signals are routed through Multiplexers (MUX) to produce $RD1$ and $RD2$, respectively.

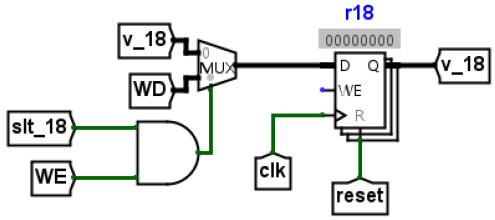

Figure 6: Register Cell Internal Structure

For each individual register cell:

clk receives the clock signal, and **reset** receives an asynchronous reset signal.

A **MUX** selects the content to be written into the register on the rising edge.

The **write data (WD)** will be written into the register only if both the `select signal (slt)` and the `global write enable (WE)` are at a high level. Otherwise, the existing value will be rewritten.

4. *ALU*

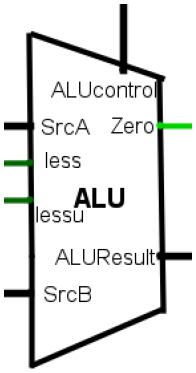

Figure 7: ALU Module Overall Structure

Port	Description
`SrcA`	First operand input (32-bit).
`SrcB`	Second operand input (32-bit).
`ALUcontrol`	ALU operation control signal (5-bit). The encoding is as follows: $5'b00000$: A + B (add) $5'b00001$: A - B (subtract) $5'b00010$: A & B (bitwise AND) $5'b00011$: A — B (bitwise OR) $5'b00100$: A < B (set less than, signed) $5'b00101$: A << B (shift left logical) $5'b00110$: A >> B (shift right logical) $5'b00111$: A >>> B (shift right arithmetic) $5'b01000$: A < B (set less than, unsigned) $5'b01001$: A $\oplus$ B (bitwise XOR) $5'b01010$: A * B (multiply, lower 32 bits) $5'b01011$: A * B (multiply, high 32 bits, signed×signed) $5'b01100$: A * B (multiply, high 32 bits, signed×unsigned) $5'b01101$: A * B (multiply, high 32 bits, unsigned×unsigned) $5'b01110$: A / B (divide, signed) $5'b01111$: A % B (remainder, signed) $5'b10000$: A / B (divide, unsigned) $5'b10001$: A % B (remainder, unsigned)
`ALUResult`	ALU operation result output (32-bit).
`Zero`	Zero flag indicating if the result is zero (1-bit). Provides the basis for branch instructions like `beq` and `bne`.

Table 3: ALU Module Ports and Control Signals

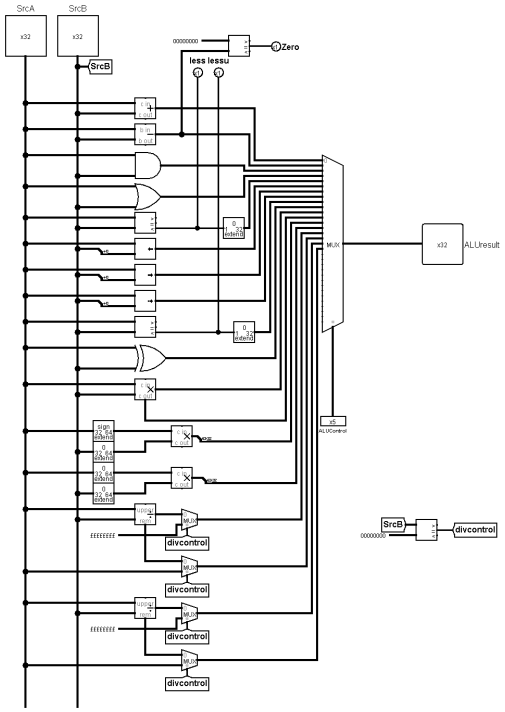

Figure 8: ALU Module Internal Structure

5. *Extend*

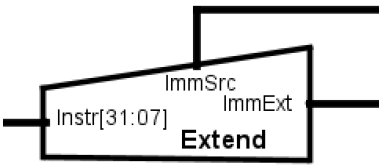

Figure 9: Extend Module Overall Structure

Port	**Description**
Instr[31 : 07]	Remove the opcode from the machine code and extract the immediate part
ImmSrc	Receives the immediate type code. $3'b000$ I1-type $3'b001$ S-type $3'b010$ B-type $3'b011$ J-type $3'b100$ I2-type $3'b101$ U-type
ImmExt	Output the extended immediate value.

Table 4: Extend Module Ports

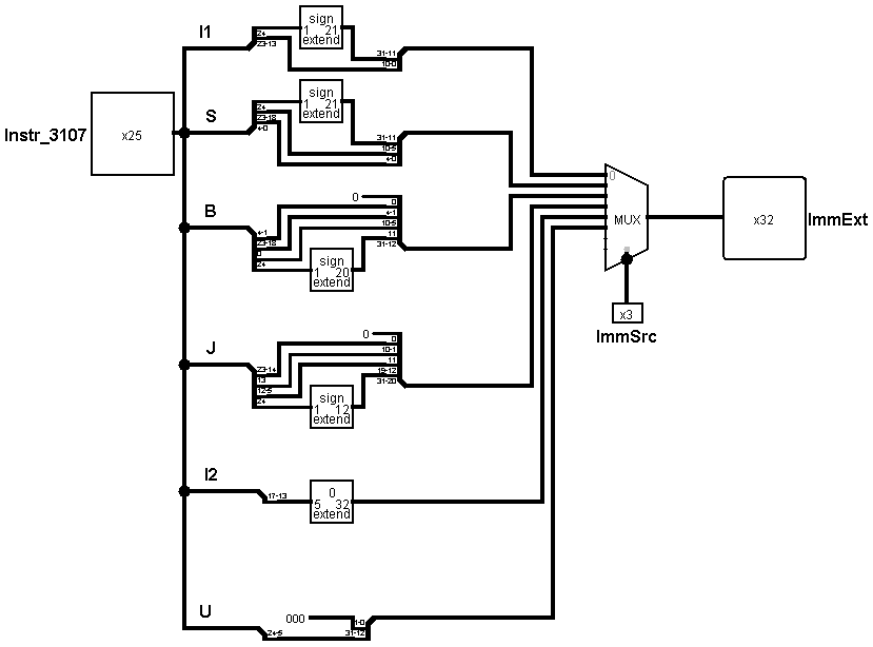

Figure 10: Extend Module Internal Structure

Immediate Type	Extraction Method
I1-type	20Instr[31],Instr[31 : 20]
S-type	20Instr[31],Instr[31 : 25],Instr[11 : 7]
B-type	20Instr[31],Instr[07],Instr[30 : 25],Instr[11 : 08],1′b0
J-type	12Instr[31],Instr[19 : 12],Instr[20],Instr[30 : 21],1′b0
I2-type	27′b0,Instr[24 : 20]
U-type	Instr[31 : 12], 12′b0

Table 5: Immediate Extraction Methods

6. *Data Memory*
   The idea is also relatively simple; just use a RAM.
   The address width is 16 bits, and the data width is 32 bits.

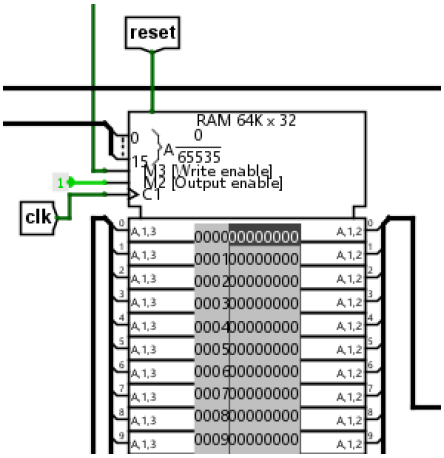

Figure 11: Data Memory Module Implementation

7. *AddrProcessor* To enable the RAM to store data byte by byte, we need to add a module before the interface to control the bit enable of the RAM — that is, to generate the mask. This can be achieved by calculating the remainder of the address divided by 4 and then combining it with the Ssrc signal to output the mask.

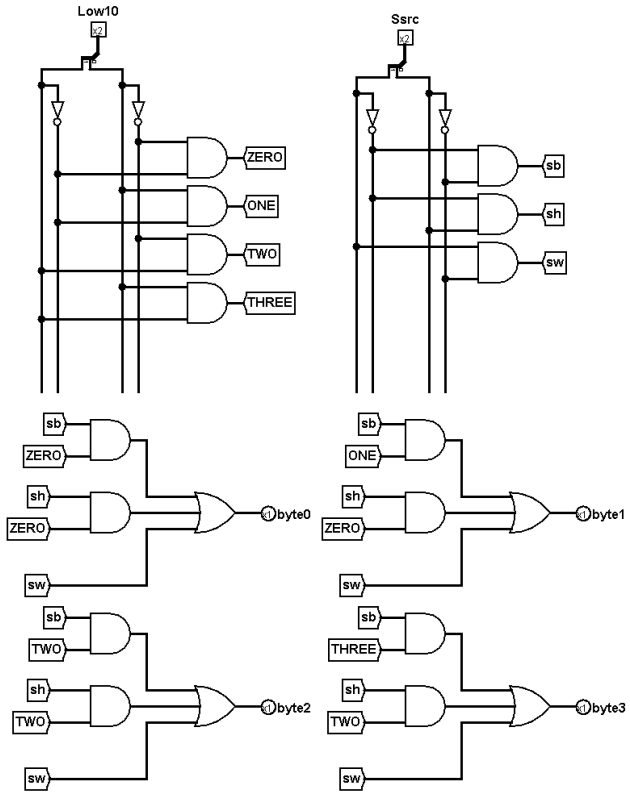

Figure 12: Extend Module Internal Structure

8. *SProcessor*

   This module handles the processing of store instructions.

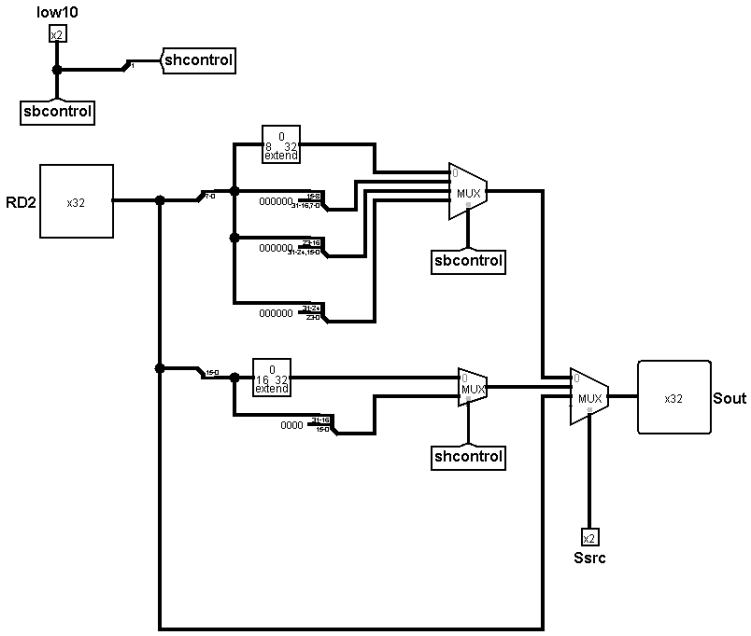

Figure 13: SProcessor Module

9. *LProcessor*

   This module handles the processing of load instructions.

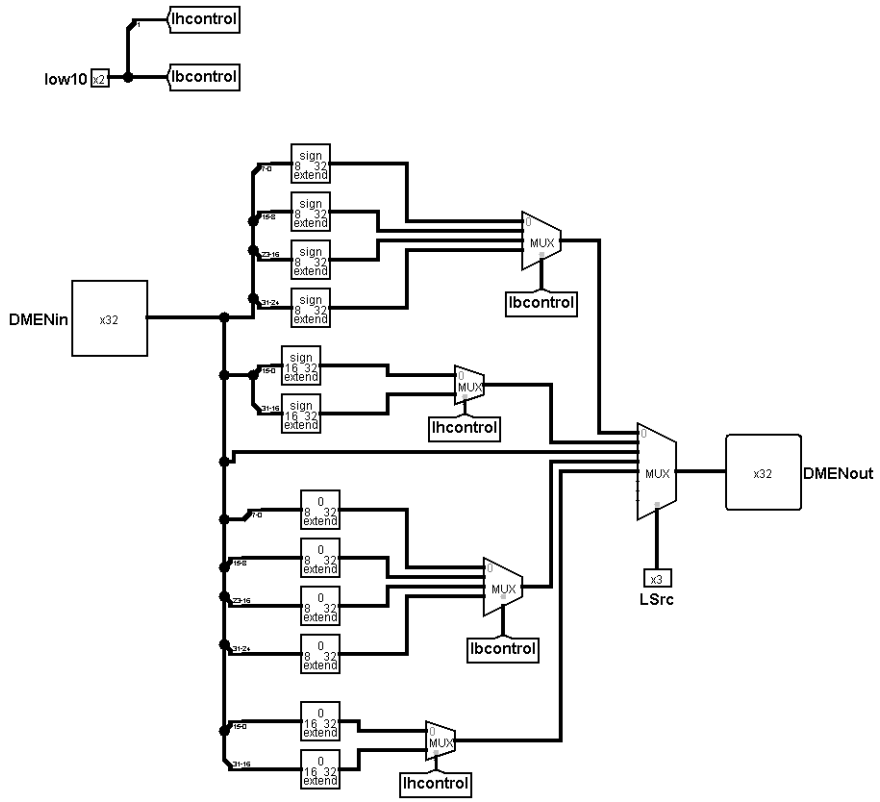

Figure 14: LProcessor Module

10. *Control Unit*

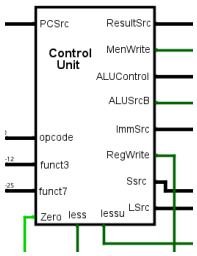

Figure 15: Control Unit Overall Structure

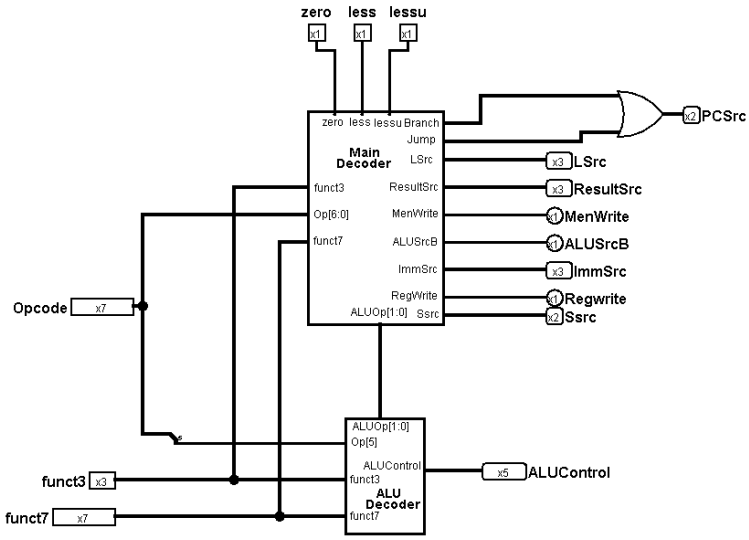

Figure 16: Control Unit Secondary Structure

Port	Description
opcode	Receives the opcode field from the instruction.
funct3	Receives the funct3 field from the instruction.
funct7	Receives the funct7 field from the instruction.
ALUop	Output ALU decoder control signal.
PCSrc	Control signal for PC MUX.
ResultSrc	Control signal for Result MUX.
MenWrite	Data Memory write enable signal.
ALUControl	ALU operation control signal.
ALUSrcB	ALU second operand MUX control signal.
ImmSrc	Immediate extension type control signal.
RegWrite	Register File write enable signal.
LSrc	Load instruction type control signal.
SSrc	Store instruction type control signal.
less and lessu	Branch signal

Table 6: Control Unit Module Ports

The Control Unit is divided into the **Main Decoder** and the **ALU Decoder**.

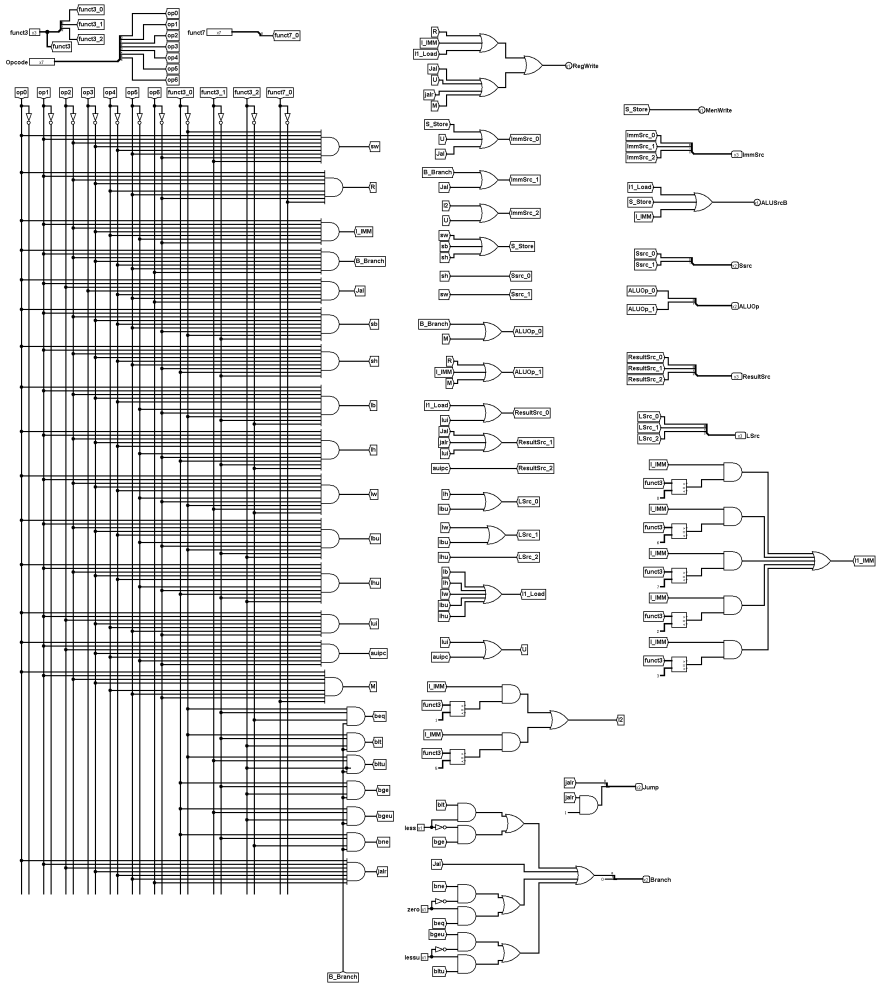

Figure 17: Main Decoder Internal Structure

The logical approach of the Main Decoder is to first split opcode[6 : 0] and funct3 and funct7 into 7-bit or 3-bit binary numbers, determine the instruction type through `AND` logic, and then generate the output status signals via `OR` logic.

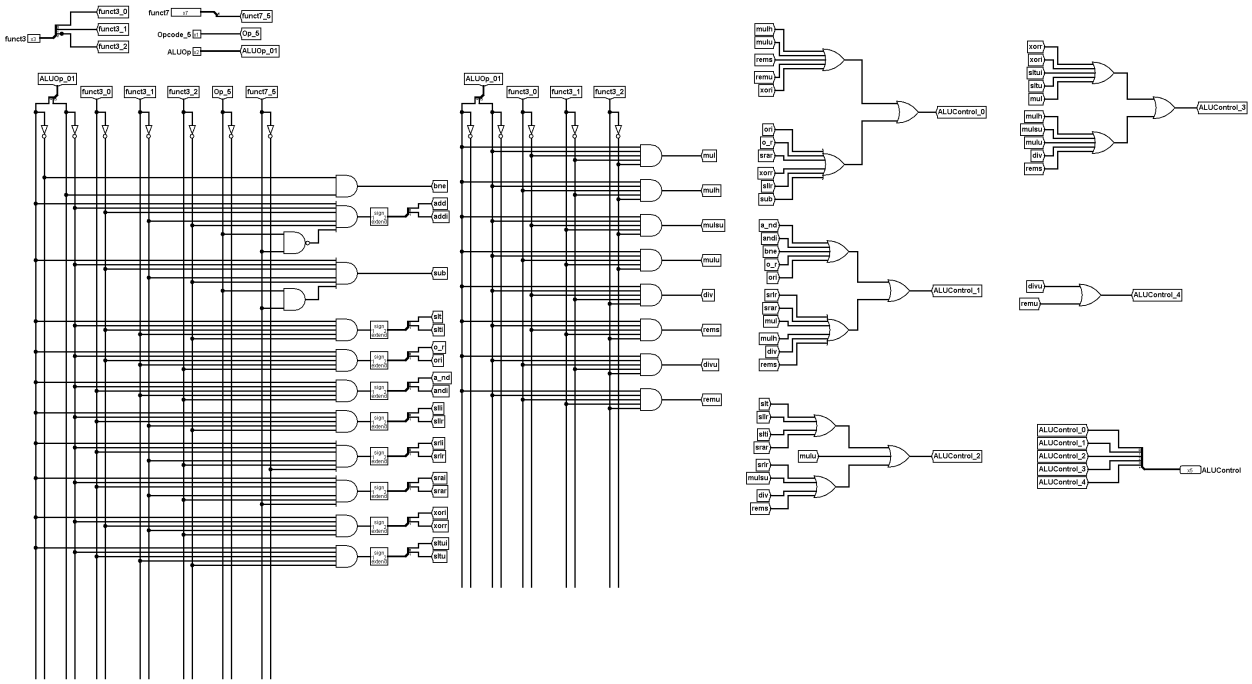

Figure 18: ALU Decoder Internal Structure

The logical approach of the ALU Decoder is to first split funct3 and funct7 into a 3-bit and a 7-bit binary number, respectively. It then determines the exact instruction name by performing `AND` logic operations on opcode[5], funct7[5], and the ALUOp signal output by the Main Decoder. Finally, it generates the output status signals via `OR` logic operations.

3.2 Data Path Explanation

The data path controls the flow trajectory of data among various modules from the fetch to the execution of each instruction. Controlling the data flow direction according to different instructions is the key to instruction implementation, which is achieved by using Multiplexers (MUX) and their selection signals.

We use 3 MUXes in data path design:

- **MUX for Next PC selection (PCSrc)**: This MUX selects between the next sequential PC address (PC + 4) and the branch target address (calculated by adding ImmExt to the current PC) and jalr jump target as the next PC value. For non-branch instructions.

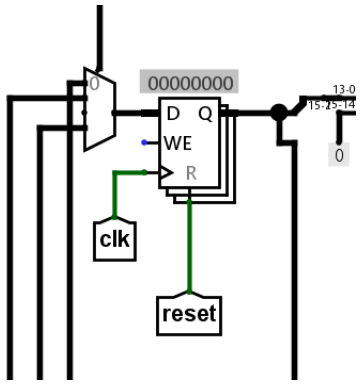

Figure 19: MUX for Next PC selection (PCSrc)

- **MUX for ALU second operand (ALUSrcB)**: This MUX selects between the second read data from the Register File (RD2) and the extended immediate value (ImmExt) as the second operand for the ALU. For R-type instructions, it selects RD2; for I1-type and S-type instructions, it selects ImmExt.

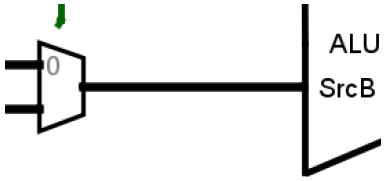

Figure 20: MUX for ALU second operand (ALUSrcB)

ALUSrcB has two input options: RD2 and ImmExt. The selection is controlled by the ALUSrcB signal. For R-type instructions, ALUSrcB is $1'b0$, selecting RD2 as the second operand for the ALU. For I1-type and S-type instructions, ALUSrcB is $1'b1$, selecting ImmExt as the second operand for the ALU.

- **MUX for Result selection (ResultSrc)**: This MUX selects between different data sources as the data to be written back to the Register File.

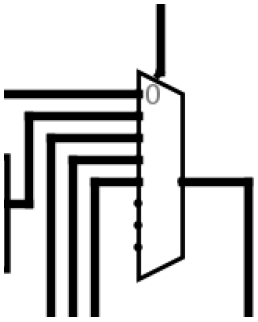

Figure 21: MUX for Result selection (ResultSrc)

ResultSrc has five input optional: ALUResult, ReadData, PC +4, high-order immediate value

and PC + high-order immediate value. The selection is controlled by the ResultSrc signal.

In summary, we present the respective control signals corresponding to each instruction, which also serves as an important basis for the CTRL connection.

Instr	Type	opcode	funct3	funct7	ResultSrc	Regwrite	Menwrite	ALUSrcB	ImmSrc	ALUop	ALUContro	LSrc	SSrc	Branch	Jump	PCSrc	opcode[5]	funct7[5]
add	R	0110011	000	0000000	000	1	0	0	\	10	000	\	\	00	00	00	1	0
sub	R	0110011	000	0100000	000	1	0	0	\	10	001	\	\	00	00	00	1	1
or	R	0110011	110	0000000	000	1	0	0	\	10	011	\	\	00	00	00	1	0
and	R	0110011	111	0000000	000	1	0	0	\	10	010	\	\	00	00	00	1	0
slt	R	0110011	010	0000000	000	1	0	0	\	10	100	\	\	00	00	00	1	0
slti	I1/IMM	0010011	010	\	000	1	0	1	000	10	100	\	\	00	00	00	0	\
lw	I1/LOAD	0000011	010	\	001	1	0	1	000	00	000	010	\	00	00	00	0	\
addi	I1/IMM	0010011	000	\	000	1	0	1	000	10	000	\	\	00	00	00	0	\
ori	I1/IMM	0010011	110	\	000	1	0	1	000	10	011	\	\	00	00	00	0	\
andi	I1/IMM	0010011	111	\	000	1	0	1	000	10	010	\	\	00	00	00	0	\
sw	S	0100011	010	\	\	0	1	1	001	00	000	\	10	00	00	00	1	\
bne	B	1100011	001	\	\	0	0	0	010	01	010	\	\	01	00	01	1	\
jal	J	1101111	\	\	010	1	0	\	011	\	\	\	\	01	00	01	1	\
sll	R	0110011	001	0000000	000	1	0	0	\	10	101	\	\	00	00	00	1	0
srl	R	0110011	101	0000000	000	1	0	0	\	10	110	\	\	00	00	00	1	0
sra	R	0110011	101	0100000	000	1	0	0	\	10	111	\	\	00	00	00	1	1
xor	R	0110011	100	0000000	000	1	0	0	\	10	1001	\	\	00	00	00	1	0
sltu	R	0110011	011	0000000	000	1	0	0	\	10	1000	\	\	00	00	00	1	0
slli	I2	0010011	001	0000000	000	1	0	1	100	10	101	\	\	00	00	00	0	0
srli	I2	0010011	101	0000000	000	1	0	1	100	10	110	\	\	00	00	00	0	0
srai	I2	0010011	101	0100000	000	1	0	1	100	10	111	\	\	00	00	00	0	1
xori	I1/IMM	0010011	100	0000000	000	1	0	1	000	10	1001	\	\	00	00	00	0	0
sltui	I1/IMM	0010011	011	0000000	000	1	0	1	000	10	1000	\	\	00	00	00	0	0
lb	I1/LOAD	0000011	000	\	001	1	0	1	000	00	000	000	\	00	00	00	0	\
lh	I1/LOAD	0000011	001	\	001	1	0	1	000	00	000	001	\	00	00	00	0	\
lbu	I1/LOAD	0000011	100	\	001	1	0	1	000	00	000	011	\	00	00	00	0	\
lhu	I1/LOAD	0000011	101	\	001	1	0	1	000	00	000	100	\	00	00	00	0	\
sb	S	0100011	000	\	\	0	1	1	001	00	000	\	00	00	00	00	1	\
sh	S	0100011	001	\	\	0	1	1	001	00	000	\	01	00	00	00	1	\
lui	U	0110111	\	\	011	1	0	\	101	\	\	\	\	00	00	00	1	\
auipc	U	0010111	\	\	100	1	0	\	101	\	\	\	\	00	00	00	0	\
mul	M	0110011	000	0000001	000	1	0	0	\	11	1010	\	\	00	00	00	1	0
mulh	M	0110011	001	0000001	000	1	0	0	\	11	1011	\	\	00	00	00	1	0
mulsu	M	0110011	010	0000001	000	1	0	0	\	11	1100	\	\	00	00	00	1	0
mulu	M	0110011	011	0000001	000	1	0	0	\	11	1101	\	\	00	00	00	1	0
div	M	0110011	100	0000001	000	1	0	0	\	11	1110	\	\	00	00	00	1	0
divu	M	0110011	101	0000001	000	1	0	0	\	11	10000	\	\	00	00	00	1	0
rem	M	0110011	110	0000001	000	1	0	0	\	11	1111	\	\	00	00	00	1	0
remu	M	0110011	111	0000001	000	1	0	0	\	11	10001	\	\	00	00	00	1	0
beq	B	1100011	000	\	000	0	0	0	010	\	\	\	\	01	00	01	\	\
blt	B	1100011	100	\	000	0	0	0	010	\	\	\	\	01	00	01	\	\
bltu	B	1100011	110	\	000	0	0	0	010	\	\	\	\	01	00	01	\	\
bge	B	1100011	101	\	000	0	0	0	010	\	\	\	\	01	00	01	\	\
bgeu	B	1100011	111	\	000	0	0	0	010	\	\	\	\	01	00	01	\	\
jalr	I1/IMM	1100111	\	\	010	1	1	0	000	\	\	\	\	00	11	11	\	\

Figure 22: Truth Table for Each Instruction

We will package and upload the complete truth table together with the engineering files.

4 Assembler Building

```
INSTRUCTIONS_SET = {
    "add": {"opcode": "0110011", "funct3": "000", "funct7": "0000000",
   "type": "R"},
    "sub": {"opcode": "0110011", "funct3": "000", "funct7": "0100000",
   "type": "R"},
    "or": {"opcode": "0110011", "funct3": "110", "funct7": "0000000", "
   type": "R"},
    "and": {"opcode": "0110011", "funct3": "111", "funct7": "0000000",
   "type": "R"},
    "slt": {"opcode": "0110011", "funct3": "010", "funct7": "0000000",
   "type": "R"},
    "slti": {"opcode": "0010011", "funct3": "010", "type": "I1"},
```

```
    "lw": {"opcode": "0000011", "funct3": "010", "type": "I1"},
    "addi": {"opcode": "0010011", "funct3": "000", "type": "I1"},
    "ori": {"opcode": "0010011", "funct3": "110", "type": "I1"},
    "andi": {"opcode": "0010011", "funct3": "111", "type": "I1"},
    "sw": {"opcode": "0100011", "funct3": "010", "type": "S"},
    "bne": {"opcode": "1100011", "funct3": "001", "type": "B"},
    "beq": {"opcode": "1100011", "funct3": "000", "type": "B"},
    "blt": {"opcode": "1100011", "funct3": "100", "type": "B"},
    "bge": {"opcode": "1100011", "funct3": "101", "type": "B"},
    "bltu": {"opcode": "1100011", "funct3": "110", "type": "B"},
    "bgeu": {"opcode": "1100011", "funct3": "111", "type": "B"},
    "jal": {"opcode": "1101111", "type": "J"},
    "sll": {"opcode": "0110011", "funct3": "001", "funct7": "0000000",
   "type": "R"},
    "srl": {"opcode": "0110011", "funct3": "101", "funct7": "0000000",
   "type": "R"},
    "sra": {"opcode": "0110011", "funct3": "101", "funct7": "0100000",
   "type": "R"},
    "xor": {"opcode": "0110011", "funct3": "100", "funct7": "0000000",
   "type": "R"},
    "sltu": {"opcode": "0110011", "funct3": "011", "funct7": "0000000",
    "type": "R"},
    "slli": {"opcode": "0010011", "funct3": "001", "funct7": "0000000",
    "type": "I2"},
    "srli": {"opcode": "0010011", "funct3": "101", "funct7": "0000000",
    "type": "I2"},
    "srai": {"opcode": "0010011", "funct3": "101", "funct7": "0100000",
    "type": "I2"},
    "xori": {"opcode": "0010011", "funct3": "100", "type": "I1"},
    "sltiu": {"opcode": "0010011", "funct3": "011", "type": "I1"},
    "lb": {"opcode": "0000011", "funct3": "000", "type": "I1"},
    "lh": {"opcode": "0000011", "funct3": "001", "type": "I1"},
    "lbu": {"opcode": "0000011", "funct3": "100", "type": "I1"},
    "lhu": {"opcode": "0000011", "funct3": "101", "type": "I1"},
    "sb": {"opcode": "0100011", "funct3": "000", "type": "S"},
    "sh": {"opcode": "0100011", "funct3": "001", "type": "S"},
    "lui": {"opcode": "0110111", "type": "U"},
    "auipc": {"opcode": "0010111", "type": "U"},
    "mul": {"opcode": "0110011", "funct3": "000", "funct7": "0000001",
   "type": "R"},
    "mulh": {"opcode": "0110011", "funct3": "001", "funct7": "0000001",
    "type": "R"},
    "mulsu": {"opcode": "0110011", "funct3": "010", "funct7": "0000001"
   , "type": "R"},
    "mulu": {"opcode": "0110011", "funct3": "011", "funct7": "0000001",
    "type": "R"},
    "div": {"opcode": "0110011", "funct3": "100", "funct7": "0000001",
   "type": "R"},
    "divu": {"opcode": "0110011", "funct3": "101", "funct7": "0000001",
    "type": "R"},
    "rem": {"opcode": "0110011", "funct3": "110", "funct7": "0000001",
   "type": "R"},
    "remu": {"opcode": "0110011", "funct3": "111", "funct7": "0000001",
    "type": "R"},
```

```
}
```

Listing 1: The Instructions Set

```python
import argparse
from typing import List, Tuple, Optional
import instructions

def preprocess_instruction(line: str) -> Tuple[str, List[str]]:
    parts = line.replace(",", "").replace("(", " ").replace(")", "").split()
    if not parts:
        raise ValueError("Empty instruction")
    mnemonic = parts[0]
    operands = parts[1:]
    return mnemonic, operands

def register_to_binary(register: str) -> str:
    if register.startswith("x"):
        reg_str = register[1:]
    try:
        reg_num = int(reg_str)
    except Exception:
        raise ValueError(f"Invalid register name: {register}")
    if not (0 <= reg_num <= 31):
        raise ValueError(f"Register out of range x0-x31: {register}")
    return format(reg_num, "05b")

def immediate_to_binary(immediate: str, bits: int) -> str:
    if immediate.startswith("x"):
        immediate = immediate[1:]
    try:
        imm = int(immediate, 0)
    except ValueError:
        raise ValueError(f"Invalid immediate: {immediate}")
    if imm < 0:
        imm = (1 << bits) + imm
    return format(imm & ((1 << bits) - 1), f"0{bits}b")

def bin_to_hex32(bin_str: str) -> str:
    if len(bin_str) != 32:
        raise ValueError(f"Expected 32-bit machine code, got {len(bin_str)} bits")
    return format(int(bin_str, 2), "08x")

def assemble_to_machine_code(assembly_instruction: str, label_map: dict, current_pc: int) -> dict:
    mnemonic, operands = preprocess_instruction(assembly_instruction)
    mnemonic = mnemonic.lower().strip()
    inst = instructions.INSTRUCTIONS_SET.get(mnemonic)
    if inst is None:
        raise ValueError(f"Unknown mnemonic: {mnemonic}")
    opcode = inst["opcode"]
```

```
    inst_type = inst["type"]

    if inst_type == "B":
        if len(operands) != 3:
            raise ValueError(f"B-type needs 3 operands, got {len(operands)}: {assembly_instruction}")
        rs1, rs2, label = operands

        if label not in label_map:
            raise ValueError(f"Undefined label: {label}")
        target_pc = label_map[label]

        offset = target_pc - current_pc
        if not (-4096 <= offset <= 4095):
            raise ValueError(
                f"B-type offset out of range (-4096~4095), label: {label}, target_pc: {target_pc}, current_pc: {current_pc}, offset: {offset}"
            )

        imm_binary = immediate_to_binary(str(offset), 13)
        machine_code = (
            f"{imm_binary[0]}"
            f"{imm_binary[2:8]}"
            f"{register_to_binary(rs2)}"
            f"{register_to_binary(rs1)}"
            f"{inst['funct3']}"
            f"{imm_binary[8:12]}"
            f"{imm_binary[1]}"
            f"{opcode}"
        )

    elif inst_type == "J":
        if len(operands) != 2:
            raise ValueError(f"J-type needs 2 operands, got {len(operands)}: {assembly_instruction}")
        rd, label = operands

        if label not in label_map:
            raise ValueError(f"Undefined label: {label}")
        target_pc = label_map[label]

        offset = target_pc - current_pc
        if not (-1048576 <= offset <= 1048575):
            raise ValueError(
                f"J-type offset out of range (-1048576~1048575), label: {label}, target_pc: {target_pc}, current_pc: {current_pc}, offset: {offset}"
            )

        imm_binary = immediate_to_binary(str(offset), 21)
        machine_code = (
            f"{imm_binary[0]}"
            f"{imm_binary[10:20]}"
```

```
            f"{imm_binary[9]}"
            f"{imm_binary[1:9]}"
            f"{register_to_binary(rd)}"
            f"{opcode}"
        )

    elif inst_type == "R":
        if len(operands) != 3:
            raise ValueError(f"R-type needs 3 operands, got {len(operands)}: {assembly_instruction}")
        rd, rs1, rs2 = operands
        machine_code = (
            f"{inst['funct7']}"
            f"{register_to_binary(rs2)}"
            f"{register_to_binary(rs1)}"
            f"{inst['funct3']}"
            f"{register_to_binary(rd)}"
            f"{opcode}"
        )

    elif inst_type == "I1":
        if len(operands) != 3:
            raise ValueError(f"I1-type needs 3 operands, got {len(operands)}: {assembly_instruction}")
        load_inst = ["lw", "lb", "lh", "lbu", "lhu"]
        if mnemonic in load_inst:
            rd, imm, rs1 = operands
        else:
            rd, rs1, imm = operands
        try:
            imm_val = int(imm, 0)
        except ValueError:
            raise ValueError(f"Invalid immediate for I1-type: {imm} (not a number)")
        if not (-2**11 <= imm_val <= 2**11 - 1):
            raise ValueError(f"I1-type immediate out of range (-2048~2047): {imm}")
        imm_binary = immediate_to_binary(imm, 12)
        machine_code = (
            f"{imm_binary}"
            f"{register_to_binary(rs1)}"
            f"{inst['funct3']}"
            f"{register_to_binary(rd)}"
            f"{opcode}"
        )

    elif inst_type == "I2":
        if len(operands) != 3:
            raise ValueError(f"I2-type needs 3 operands, got {len(operands)}: {assembly_instruction}")
        rd, rs1, imm = operands
        try:
            imm_val = int(imm, 0)
        except ValueError:
```

```
144             raise ValueError(f"Invalid immediate for I2-type: {imm} (
    not a number)")
145         if not (0 <= imm_val <= 31):
146             raise ValueError(f"I2-type immediate out of range (0~31): {
    imm}")
147         imm_binary = immediate_to_binary(imm, 5)
148         machine_code = (
149             f"{inst['funct7']}"
150             f"{imm_binary}"
151             f"{register_to_binary(rs1)}"
152             f"{inst['funct3']}"
153             f"{register_to_binary(rd)}"
154             f"{opcode}"
155         )
156 
157     elif inst_type == "S":
158         if len(operands) != 3:
159             raise ValueError(f"S-type needs 3 operands, got {len(
    operands)}: {assembly_instruction}")
160         rs2, imm, rs1 = operands
161         try:
162             imm_val = int(imm, 0)
163         except ValueError:
164             raise ValueError(f"Invalid immediate for S-type: {imm} (not
     a number)")
165         if not (-2**11 <= imm_val <= 2**11 - 1):
166             raise ValueError(f"S-type immediate out of range
    (-2048~2047): {imm}")
167         imm_binary = immediate_to_binary(imm, 12)
168         machine_code = (
169             f"{imm_binary[:7]}"
170             f"{register_to_binary(rs2)}"
171             f"{register_to_binary(rs1)}"
172             f"{inst['funct3']}"
173             f"{imm_binary[7:]}"
174             f"{opcode}"
175         )
176 
177     elif inst_type == "U":
178         if len(operands) != 2:
179             raise ValueError(f"U-type needs 2 operands, got {len(
    operands)}: {assembly_instruction}")
180         rd, imm = operands
181         try:
182             imm_val = int(imm, 0)
183         except ValueError:
184             raise ValueError(f"Invalid immediate for U-type: {imm} (not
     a number)")
185         if not (0 <= imm_val <= 2**20 - 1):
186             raise ValueError(f"U-type immediate out of range
    (0~1048575): {imm}")
187         imm_binary = immediate_to_binary(imm, 20)
188         machine_code = (
189             f"{imm_binary}"
```

```
            f"{register_to_binary(rd)}"
            f"{opcode}"
        )

    else:
        raise ValueError(f"Unknown instruction type: {inst_type}")

    if len(machine_code) != 32:
        raise ValueError(f"Machine code length error: {len(machine_code)} bits (expected 32)")

    return {"Assembly": assembly_instruction, "MachineCode": machine_code, "Type": inst_type}

def strip_comment(line: str) -> str:
    for sep in ("#", ";"):
        if sep in line:
            line = line.split(sep, 1)[0]
    return line.strip()

def assemble_file(
    in_path: str,
    emit_hex: bool = True,
) -> List[str]:
    output_lines: List[str] = []
    label_map: dict[str, int] = {}

    try:
        with open(in_path, "r", encoding="utf-8") as f:
            pc = 0
            pending_labels: List[str] = []

            for lineno, raw_line in enumerate(f, start=1):
                line = strip_comment(raw_line).strip()
                if not line:
                    continue

                inst_part = line
                if ":" in line:
                    label_part, inst_part = line.split(":", 1)
                    label_part = label_part.strip()
                    inst_part = inst_part.strip()

                    if not label_part:
                        raise ValueError(f"Empty label at line {lineno}")
                    if not (label_part[0].isalpha() or label_part[0] == "_"):
                        raise ValueError(f"Invalid label (starts with non-letter/underscore) at line {lineno}: {label_part}")
                    if not all(c.isalnum() or c == "_" for c in label_part):
                        raise ValueError(f"Invalid character in label at line {lineno}: {label_part}")
```

```
237
238                     pending_labels.append(label_part)
239
240                 if inst_part:
241                     for label in pending_labels:
242                         if label in label_map:
243                             raise ValueError(f"Duplicate label at line
    {lineno}: {label}")
244                         label_map[label] = pc
245                     pending_labels.clear()
246                     pc += 4
247
248             if pending_labels:
249                 raise ValueError(f"Labels have no corresponding
    instruction: {', '.join(pending_labels)}")
250
251     except Exception as e:
252         output_lines.append(f"ERROR [Label Collection]: {e}")
253         return output_lines
254
255     try:
256         with open(in_path, "r", encoding="utf-8") as f:
257             pc = 0
258
259             for lineno, raw_line in enumerate(f, start=1):
260                 stripped_line = strip_comment(raw_line).strip()
261                 if not stripped_line:
262                     continue
263
264                 if ":" in stripped_line:
265                     _, inst_part = stripped_line.split(":", 1)
266                     inst_part = inst_part.strip()
267                 else:
268                     inst_part = stripped_line
269
270                 if not inst_part:
271                     continue
272
273                 try:
274                     res = assemble_to_machine_code(inst_part, label_map
    , current_pc=pc)
275                     bin_code = res["MachineCode"]
276                     if emit_hex:
277                         hex_code = bin_to_hex32(bin_code)
278                         output_lines.append(hex_code)
279                     else:
280                         output_lines.append(bin_code)
281                     pc += 4
282                 except Exception as e:
283                     output_lines.append(f"ERROR line {lineno}: {e}")
284
285     except Exception as e:
286         output_lines.append(f"ERROR [Instruction Assembly]: {e}")
287
```

```
    return output_lines

def main():
    parser = argparse.ArgumentParser(description="Simple RV32I/RV32M assembler (supports label for jal/b-type) - batch mode")
    parser.add_argument("-f", "--file", help="Input assembly file (.s)", required=True)
    parser.add_argument("-o", "--out", help="Output text file (optional)")
    parser.add_argument("--bin", action="store_true", help="Emit binary instead of hex")
    args = parser.parse_args()

    output_lines = assemble_file(args.file, emit_hex=not args.bin)
    print("\n".join(output_lines))
    if args.out:
        with open(args.out, "w", encoding="utf-8") as f:
            f.write("\n".join(output_lines))

if __name__ == "__main__":
    main()
```

Listing 2: The Assembler Code

The Assembler converts RISC-V assembly instructions into 32-bit machine code. It first parses each instruction with `preprocess_insturctions` to extract the mnemonic and operands (handling commas, parentheses, and load/store addressing). Then `assemble_to_machine_code` assembles the machine code based on the instruction type: it converts register names to 5-bit binary, converts immediates to the required bit width (using two's complement for negatives),and concatenates fields in the order specified by each instruction format. Finally, `assemble_file` processes assembly files line by line, converts each instruction, and outputs hex or binary while maintaining a PC that increments by 4 bytes per inctruction.

We have also implemented error-checking function, which will automatically output errors when invalid instructions are entered.

5 Test Cases and Verification

5.1 Test Cases

We have designed a series of test cases, and judge whether the CPU executes instructions correctly by monitoring the states of the Register File and the Data Memory.

```
addi x1, x0, 5  #initializes x1 as 5
addi x2, x0, -2 #initializes x2 as -2
add x0, x1, x2 # test if x0 = 0
addi x3, x0, -5 #initializes x3 as -5
addi x4, x0, 3  #initializes x4 as 3
addi x5, x0, 0xA #initializes x5 as 10
add x6, x1, x4  # x6 = 8
sub x7, x1, x4  # x7 = 2
sll x8, x1, x4  # x8 = 0x28
srl x9, x5, x4  # x9 = 1
```

```
sra x10, x2, x4 # x10 = -1
slt x11, x2, x1 # x11 = 1
sltu x12, x2, x1 # x12 = 0
xor x13, x1, x4 # x13 = 6
and x14, x1, x4 # x14 = 1
or x15, x1, x4  # x15 = 7
slti x16, x1, -2 # x16 = 0
sltiu x17, x1, 10 # x17 = 1
xori x18, x5, 5 # x18 = 15
ori x19, x0, 7 # x19 = 7
ori x24, x1, 2047   # x24 = 0x000007FF
andi x20, x5, 3 # x20 = 2
slli x21, x1, 2 # x21 = 0x14
srli x22, x5, 2 # x22 = 2
srai x23, x3, 2 # x23 = -2
addi x4, x4, 1  # bne loop, x4 = 5 when the loop is over
bne x4, x1, -4
jal x30, 8 # x30 = 0x6c
addi x1, x0, 7  # if jal fail, x1 = 7
auipc x31, 1   # x31 = pc = 0x1074
sw x1, 8(x6)          # [0x00000004] = 5
sw x2, 12(x6)         # [0x00000005] = -2
sw x16, 16(x6)         # [0x00000006] = 0
beq x20, x21, 8        # not branch
addi x1, x0, 4         # x1 = 4
bne x20, x21, 8        # branch
addi x1, x0, 2
blt x23, x21, 8        # branch
addi x2, x0, 4
bge x23, x21, 8        # not branch
addi x2, x0, 2         # x2 =2
bltu x21, x23, 8       # branch
addi x16, x0, 2
bgeu x21, x23, 8       # not branch
addi x16, x0, 4        # x16 = 4

slli x1, x2, 32      # Imm out of range(5)
add x1, x2           # Missing operand
add x32, x1, x2      # Register out of range
addi x1, x2, 2048    # Imm out of range(12)
lw x1, abc(x2)       # Imm format error
sw x1, x2, 10        # operand order error
lui x1, 0x100000     # Imm out of range(20)
```

Listing 3: RV32I Test Cases 1

```
addi x1, x0, 1 #x1 = 0x00000001
addi x2, x0, 2 #x2 = 0x00000002
addi x3, x0, 3 #x3 = 0x00000003
addi x4, x0, -1 #x4 = 0xFFFFFFFF
addi x5, x0, -2 #x5 = 0xFFFFFFFE
addi x6, x0, 8 #x6 = 0x00000008
sw x1, -4(x6) #[0x00000001] = 0x00000001
```

```
sh x2, 0(x6) #[0x000000002] = 0x00000002
sh x4, 2(x6) #[0x000000002] = 0xFFFF0002
sb x1, 4(x6) #[0x000000003] = 0x00000001
sb x3, 5(x6) #[0x000000003] = 0x00000301
sb x4, 6(x6) #[0x000000003] = 0x00FF0301
sb x5, 7(x6) #[0x000000003] = 0xFEFF0301
lw x7, -4(x6) #x7 = 0x00000001
lh x8, 0(x6) #x8 = 0x00000002
lh x8, 2(x6) #x8 = 0xFFFFFFFF
lb x9, 4(x6) #x9 = 0x00000001
lb, x10, 5(x6) #x10 = 0x00000003
lb, x11, 6(x6) #x11 = 0xFFFFFFFF
lb, x12, 7(x6) #12 = 0xFFFFFFFE
lhu, x13, 0(x6) #13 = 0x00000002
lhu, x14, 2(x6) #14 = 0x0000FFFF
lbu, x15, 4(x6) #x15 = 0x00000001
lbu, x16, 5(x6) #x16 = 0x00000003
lbu, x17, 6(x6) #x17 = 0x0000000FF
lbu, x18, 7(x6) #x18 = 0d00000000FE
```

Listing 4: RV32i Test Cases2

```
addi x1, x0, 5  #initializes x1 as 5
addi x2, x0, -2 #initializes x2 as -2
addi x3, x0, -5 #initializes x3 as -5
addi x4, x0, 3  #initializes x4 as 3
addi x5, x0, 0xA #initializes x5 as 10
mul x6, x1, x4 # x6 = 0xE
mul x7, x2, x1 # x7 = 0xFFFFFFF6
mulh x8, x1, x2 # x8 = 0xFFFFFFFF
mulsu x10, x2, x1 # x10 = 0xFFFFFFFF
mulsu x11, x1, x2 # x11 = 0x00000004
mulu x12, x2, x3 # x12 = 0xFFFFFFF9
div x13, x5, x4 # x13 = 3
div x14, x3, x4 # x14 = 0xFFFFFFFF
div x15, x1, x0 # x15 = 0xFFFFFFFF
divu x16, x3, x4 # x16 = 0x55555553
rem x17, x5, x4 # x17 = 1
rem x18, x3, x4 # x18 = 0xFFFFFFFE
rem x19, x1, x0 # x19 = 5
remu x20, x3, x4 # x20 = 2
remu x21, x2, x0 # x21 = 0xFFFFFFFE
```

Listing 5: RV32M Test Cases

5.2 Verification

Importing the `Assembler_output.txt` file (generated by converting `RV32I.s` and `RV32M.s` via `Assembler_batch.py`) into `Convert.py` will yield a hexadecimal file compatible with Logisim-Evolution v3.0, which can be directly imported into the ROM for automated testing.

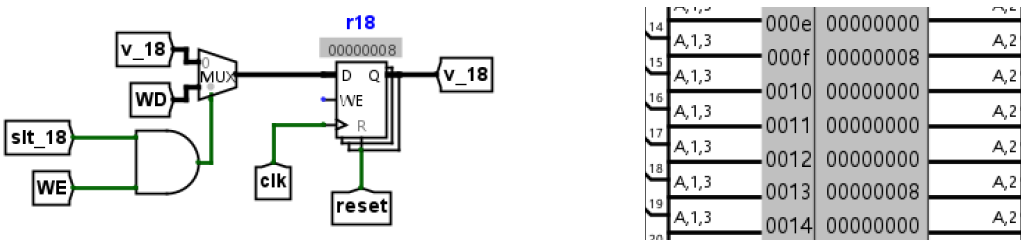

Figure 23: Verification Sign

Testing confirms that our CPU operates normally.

6 Summary of the project

This project involved designing and implementing a simple single-cycle RISC-V CPU that supports a subset of the RV32I instruction set. The CPU architecture includes essential components such as the Program Counter (PC), Instruction Memory, Register File, ALU, Data Memory, and Control Unit. Each module was carefully designed to ensure correct functionality and seamless integration within the CPU.

Students have initially learned and understood the RISC-V instruction set and the basic architecture of computers, laying a solid foundation for their subsequent studies.

7 Team Roles and Responsibilities

- **Mingxuan Li:** Responsible for assembler development and test case design.
- **Ruiyang Xu:** Responsible for the design and implementation of key modules and the writing of the final report.
- **Kaiwen Yang:** Responsible for datapath and module implementation and test case design, CPU and assembler testing(debugging)

8 Acknowledgements

We would like to express our sincere gratitude to our course teaching assistants for their invaluable guidance and support throughout this project. Their expertise and encouragement have been instrumental in helping us navigate the complexities of CPU design and implementation.