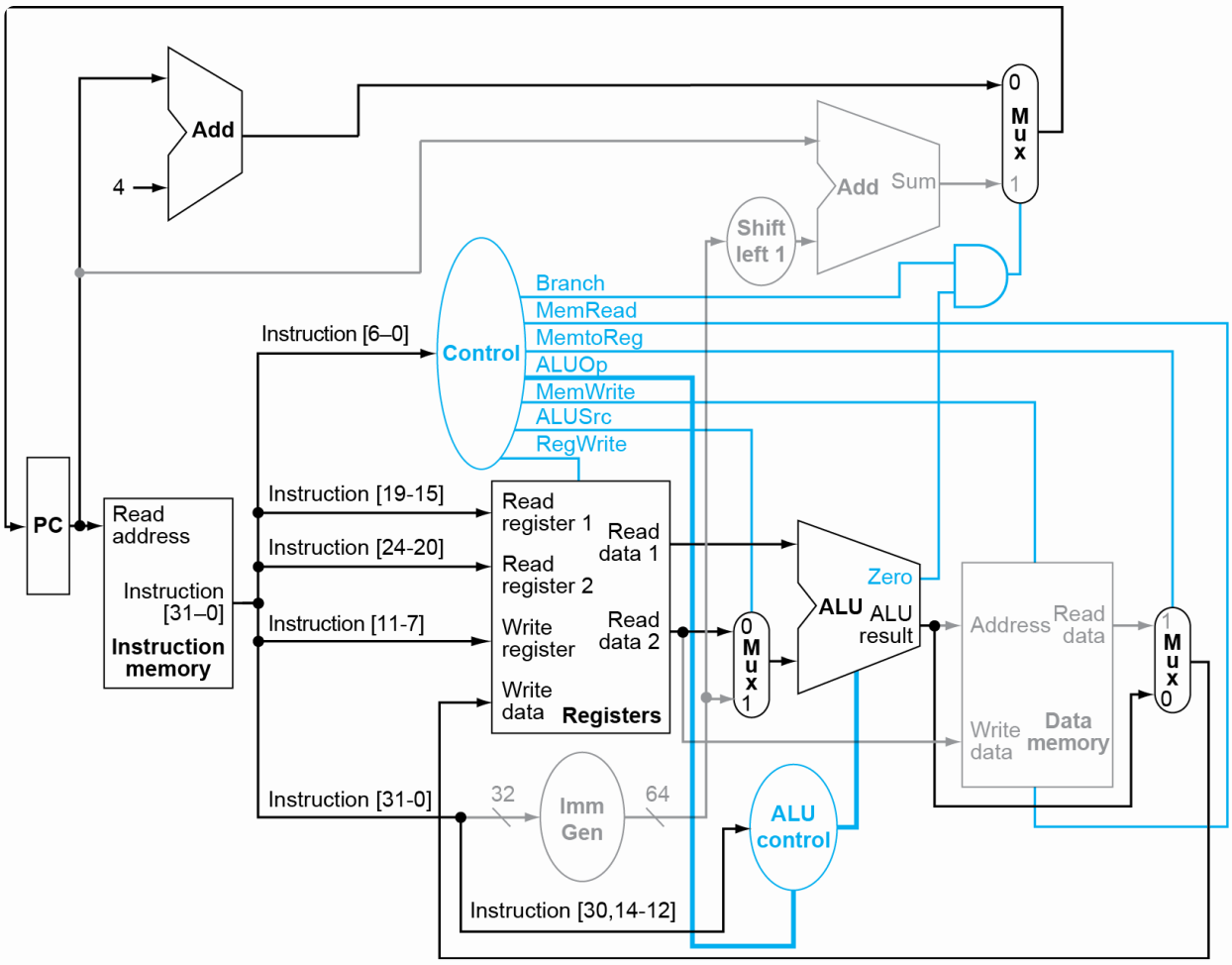
Assignment 4 (Programming Part) - 50%

In this assignment, you will implement a simple RV64I RISC-V single-cycle processor with Verilog. The processor will execute 32-bit instructions, perform corresponding RISC-V operations, and store arithmetic results in the appropriate registers.

Data Path

The CPU datapath is from the textbook.



Since branch and load/store instructions are not required for this assignment, you may omit components that are not needed.

Supported Instructions

Your CPU must support the following RISC-V instructions:

- `add rd, rs1, rs2`

- `addi rd, rs1, imm`
- `sub rd, rs1, rs2`
- `and rd, rs1, rs2`
- `andi rd, rs1, imm`
- `or rd, rs1, rs2`
- `ori rd, rs1, imm`
- `xor rd, rs1, rs2`
- `xori rd, rs1, imm`
- `sll rd, rs1, rs2`
- `slli rd, rs1, shamt`
- `srl rd, rs1, rs2`
- `srli rd, rs1, shamt`
- `sra rd, rs1, rs2`
- `srai rd, rs1, shamt`

Note:

For `sll`, `srl`, and `sra`, the shift amount is determined by the lowest 6 bits of `rs2`.

Refer to *The RISC-V Instruction Set Manual* for detailed specifications.

Instruction Format

Refer to *The RISC-V Instruction Set Manual*, particularly the "RV32/64G Instruction Set Listings" section, for details about opcodes and instruction formats. Below are the formats and opcodes for the instructions you need to implement:

R-Type

funct7	rs2	rs1	funct3	rd	opcode	instruction
0000000	rs2	rs1	000	rd	0110011	`add`
0100000	rs2	rs1	000	rd	0110011	`sub`
0000000	rs2	rs1	100	rd	0110011	`xor`
0000000	rs2	rs1	110	rd	0110011	`or`
0000000	rs2	rs1	111	rd	0110011	`and`
0000000	rs2	rs1	001	rd	0110011	`sll`
0000000	rs2	rs1	101	rd	0110011	`srl`

0100000	rs2	rs1	101	rd	0110011	`sra`

I-Type

imm[11:0]	rs1	funct3	rd	opcode	instruction
imm[11:0]	rs1	000	rd	0010011	`addi`
imm[11:0]	rs1	100	rd	0010011	`xori`
imm[11:0]	rs1	110	rd	0010011	`ori`
imm[11:0]	rs1	111	rd	0010011	`andi`

Immediate Shift

imm[11:6]	imm[5:0]	rs1	funct3	rd	opcode	instruction
000000	shamt	rs1	001	rd	0010011	`slli`
000000	shamt	rs1	101	rd	0010011	`srli`
010000	shamt	rs1	101	rd	0010011	`srai`

Module Implementation

You are required to implement the following modules. Templates for **CPU**, **PC**, **instruction memory**, and **register file** are provided. Ensure your implementation follows the comments in the templates to maintain compatibility with the testbench.

Required Modules

1. **CPU**: The main module integrating all components.
2. **Register File**: Contains 32 64-bit registers. Register 0 must always return zero, regardless of write operations.
3. **PC (Program Counter)**: Tracks the address of the current instruction.
4. **Instruction Memory**: Stores and retrieves program instructions. This module has already been implemented for you; do not modify it.
5. **Control**: Decodes the instruction opcode and generates control signals for other components.
6. **ALU**: Executes arithmetic and logical operations based on control signals.
7. **ALU Control**: Decodes `funct7` and `funct3` fields to specify the operation for the ALU.
8. **Imm Gen**: Extracts and sign-extends immediate values from instructions.

You may implement additional modules if needed.

Testbench

We have provided the testbench file, `Testbench.v`, for your use. You are not permitted to modify the testbench, except for debugging purposes. When grading your submission, we will replace `Testbench.v` with the original version.

The testbench reads instructions from `instruction.txt`, loads them into the instruction memory, and then starts the CPU. It logs the program counter (PC) and all register values at the end of each cycle, saving this data to `output.txt`. Your submission will be graded by comparing your generated `output.txt` with the correct reference output.

The `instruction.txt` file is a plain text file where each line contains a 32-bit binary instruction (ASCII '0' or '1'), with one instruction per line.

Sample `instruction.txt` and `output.txt` files are provided in the `testcase` folder within the downloaded template files. For grading, we will use a private `instruction.txt` file.

Submission Guidelines

Submit a **zip file** named `<Student_ID>_HW4.zip` through NTU COOL, with the following directory structure:

```
<Student_ID>_HW4
├── CPU.v
├── Registers.v
└── <all other .v files>
```

Please make sure the directory `<Student_ID>_HW4` is included in the zip file. Replace `<Student_ID>` with your student ID in uppercase letters.

Ensure your code can be compiled with the following command:

```
iverilog -o CPU.out *.v
```

If your code cannot be compiled, you will receive **zero points**.

Policies

- **Plagiarism**: Strictly prohibited. Copying or sharing work will result in **zero points**.
- **AI-Generated Content**: Not allowed. Submitting AI-generated code will result in **zero points**.