Python CPU Emulator Code Report

1. Overview

The code defines a simple CPU emulator that simulates the basic operations of a CPU. The CPU has:

- Eight general-purpose registers.
- Main memory with 1024 words.
- A program counter.
- A total clock cycle counter.
- Map for opcode and registers to encode
- Suppoort basic operations of mov, add, sub, mul, div, and end operation.

In this project, I have demonstrated both a simple and a pipelined execution on the same program.

2. Instruction Set Architecture (ISA)

The supported instructions are:

- mov: Move an immediate value or the content of one register to another.
- add: Add two registers' values or a register's value and an immediate value and store the result in a destination register.
- sub: Subtract two registers' values or a register's value and an immediate value and store the result in a destination register.
- mul: Multiply two registers' values or a register's value and an immediate value and store the result in a destination register.
- div: Divide a register's value by another register's value or an immediate value and store the result in a destination register. The function also handles division by zero error.
- end: End the program execution.

Note: For the clock cycle, I assumed that mov = 0.5 clock cycle, add/sub = 0.75 clock cycle, and mul/div = 1 clock cycle.

3. Functions

- encode_instruction: convert human-readable instruction to a 32-bit binary instruction.
- execute: read instructions from the program and executes them sequentially, updating the state of the CPU accordingly.
- run: execute the program and provides an overview of the CPU's performance, calculating cycles per instruction (CPI).
- execute_instruction: similar to execute function but specifically designed for pipeline_execute.
- pipeline_execute: emulates a basic 4-stage pipeline (Fetch, Decode, Execute, and Write Back) and prints the state of the pipeline after each cycle.
- run_pipeline: resets the CPU state and executes the program using the pipeline.

3.1. Encoding Design

- ✓ Registers and Opcodes have already been mapped.
- 3-bit for opcode
- 3-bit for destination_register
- 8-bit for operand_1
- 8-bit for operand_2

Result: 32-bit Encoded Form

Example:

```
Decoded       Encoded Form
mov r1 1      00000100000001000000000000000000
mov r2 0      00001000000000000000000000000000
mov r3 3      00001100000011000000000000000000
add r1 r1 1   00100100000001000000010000000000
```

4. Pipelining

The emulator also has a basic pipelining mechanism. Pipelining is a technique used in modern processors to increase instruction throughput. This CPU emulator supports a 4-stage pipeline:

1. IF/ID (Instruction Fetch/ Instruction Decode)
2. ID/EX (Instruction Decode/ Execute)
3. EX/MEM (Execute/ Memory Access)
4. MEM/WB (Memory Access/ Write Back)

Some details of each stage:

- IF Stage (Instruction Fetch): In the IF stage, instructions are fetched from memory to keep a steady flow of tasks for the CPU.
- ID Stage (Instruction Decode): During the ID stage, fetched instructions are understood and prepared for execution by identifying their type and associated data.
- EX Stage (Execute): The EX stage is where the actual computations and logical operations take place, making instructions come to life.
- MEM Stage (Memory Access): In the MEM stage, memory operations are performed, enabling communication between the CPU and memory for data retrieval or storage.
- WB Stage (Write Back): The WB stage marks the end of an instruction's journey, where results are written back to registers, making them accessible for future instructions.

Here's how exactly the function in my code works:

1. Initialization: The function starts by initializing a loop that continues until the program reaches its end. Inside the loop, it prints the current state of each pipeline stage for debugging purposes.
2. Pipeline Register Movement: The instructions in the pipeline are moved from one stage to the next. Each stage's contents are copied to the next stage. This is done in reverse order, starting from the MEM/WB stage, moving to the EX/MEM stage, and so on. The idea is to propagate the instructions through the pipeline stages.

3. Instruction Fetch (IF) Stage: The next instruction to be executed is fetched from the program memory (using self.pc) and placed in the IF/ID pipeline register. This simulates the instruction fetching stage.
4. Execution (EX) Stage: If there is an instruction in the EX/MEM stage (which represents the execution stage), it is executed using the execute_instruction function. This allows the actual execution of instructions in the pipeline.
5. Program Termination: The loop checks if all pipeline stages are empty (contain None). If they are all empty, it means that all instructions have completed their execution, and the program can terminate.
6. Cycle Count: The number of clock cycles (self.cycles) is incremented at each iteration of the loop to keep track of the total clock cycles consumed during pipeline execution.
7. Final State and Metrics: After the loop exits, the function prints the final state of each pipeline stage and the total number of clock cycles consumed during pipeline execution.

In summary, the pipeline_execute function simulates the movement of instructions through the pipeline stages, fetches and executes instructions, and keeps track of clock cycles. Please keep in mind that this is a very simple simulation of an actual CPU, so not all the intricate processes such as memory operations (lw/sw) are not being performed in this demonstration.

5. File Input

A function read_program_from_file is used to read instructions from a file and add them to the program. The file name is input.txt and should be located in the same directory as the project.

6. Conclusion

This CPU emulator provides a simple representation of a CPU's functionality, demonstrating both the sequential execution of instructions and the benefits of pipelining to increase instruction throughput.

7. Limitations

- Extend Functionality: The current ISA is very basic. It can be extended to support more instructions like logical operations, jumps, or branching.
- Error Handling: While there is some error handling (e.g., division by zero), more extensive error handling and reporting would make the emulator more robust.
- Memory Operations: Incorporate memory operations to simulate load and store operations from/to main memory.
- Enhance Pipelining: Introduce hazard detection and resolution to handle cases where instructions depend on the results of previous ones still in the pipeline.

Overall, this emulator serves as a solid foundation for understanding and simulating basic CPU operations.

Demonstration

Input.txt (the input file)

input.txt

```
mov r1 1
mov r2 0
mov r3 3
add r1 r1 1
add r1 r1 r2
mul r1 r1 2
div r4 r3 r2  # Division by zero here
add r2 r1 4
end
```

The main class:

```
cpu = CPU()
# cpu.run(program1)
# cpu.run(program2)
print()
user_program = read_program_from_file('input.txt')
# print(user_program)
cpu.run(user_program)
print("*" * 75)
print("\nPipelined execution:")
cpu.run_pipeline(user_program)
```

Output for normal execution:

```
   Decoded      Encoded Form                        Updated Registers          Cycles
1. mov r1 1     00000100000001000000000000000000  [0, 1, 0, 0, 0, 0, 0, 0]   0.5
2. mov r2 0     00000100000000000000000000000000  [0, 1, 0, 0, 0, 0, 0, 0]   1.0
3. mov r3 3     00000110000001100000000000000000  [0, 1, 0, 3, 0, 0, 0, 0]   1.5
4. add r1 r1 1  00100100000001000000010000000000  [0, 2, 0, 3, 0, 0, 0, 0]   2.25
5. add r1 r1 r2 00100100000001000000100000000000  [0, 2, 0, 3, 0, 0, 0, 0]   3.0
6. mul r1 r1 2  01100100000001000000100000000000  [0, 4, 0, 3, 0, 0, 0, 0]   4.0
Division by zero error! Skipping instruction: ['div', 'r4', 'r3', 'r2']
8. add r2 r1 4  00101000000001000001000000000000  [0, 4, 8, 3, 0, 0, 0, 0]   4.75

Final Registers: [0, 4, 8, 3, 0, 0, 0, 0]
Number of Cycles: 4.75
Number of Executed Instructions: 8
Cycles per Instruction (CPI): 0.59
***************************************************************************
```

Output for pipelined execution:

```
**************************

Pipelined execution:

Current Pipeline State:
IF/ID: None
ID/EX: None
EX/MEM: None
MEM/WB: None

Current Pipeline State:
IF/ID: ['mov', 'r1', 1]
ID/EX: None
EX/MEM: None
MEM/WB: None

Current Pipeline State:
IF/ID: ['mov', 'r2', 0]
ID/EX: ['mov', 'r1', 1]
EX/MEM: None
MEM/WB: None
```

```
Current Pipeline State:
IF/ID: ['mov', 'r3', 3]
ID/EX: ['mov', 'r2', 0]
EX/MEM: ['mov', 'r1', 1]
MEM/WB: None

Current Pipeline State:
IF/ID: ['add', 'r1', 'r1', 1]
ID/EX: ['mov', 'r3', 3]
EX/MEM: ['mov', 'r2', 0]
MEM/WB: ['mov', 'r1', 1]

Current Pipeline State:
IF/ID: ['add', 'r1', 'r1', 'r2']
ID/EX: ['add', 'r1', 'r1', 1]
EX/MEM: ['mov', 'r3', 3]
MEM/WB: ['mov', 'r2', 0]
```

```
Current Pipeline State:
IF/ID: ['mul', 'r1', 'r1', 2]
ID/EX: ['add', 'r1', 'r1', 'r2']
EX/MEM: ['add', 'r1', 'r1', 1]
MEM/WB: ['mov', 'r3', 3]

Current Pipeline State:
IF/ID: ['div', 'r4', 'r3', 'r2']
ID/EX: ['mul', 'r1', 'r1', 2]
EX/MEM: ['add', 'r1', 'r1', 'r2']
MEM/WB: ['add', 'r1', 'r1', 1]

Current Pipeline State:
IF/ID: ['add', 'r2', 'r1', 4]
ID/EX: ['div', 'r4', 'r3', 'r2']
EX/MEM: ['mul', 'r1', 'r1', 2]
MEM/WB: ['add', 'r1', 'r1', 'r2']
Division by zero error! Skipping instruction: ['div', 'r4', 'r3', 'r2']
```

```
Current Pipeline State:
IF/ID: None
ID/EX: ['add', 'r2', 'r1', 4]
EX/MEM: ['div', 'r4', 'r3', 'r2']
MEM/WB: ['mul', 'r1', 'r1', 2]

Current Pipeline State:
IF/ID: None
ID/EX: None
EX/MEM: ['add', 'r2', 'r1', 4]
MEM/WB: ['div', 'r4', 'r3', 'r2']

Final Pipeline State:
IF/ID: None
ID/EX: None
EX/MEM: None
MEM/WB: ['add', 'r2', 'r1', 4]

Total Clock Cycles: 14.75
Registers: [0, 4, 8, 3, 0, 0, 0, 0]
Total Clock Cycles: 14.75
```