ECE-GY 6913, Computing System Architecture

Project (Phase I)

Komal Niraula (kn2505)

Introduction:

The report outlines the implementation of a single-stage RISC-V processor based on the provided Python code. The processor simulates the RV32I instruction set and incorporates essential components such as instruction memory, data memory, and a register file. These components enable the execution of various instruction types, including arithmetic, logical, branch, and memory access operations.

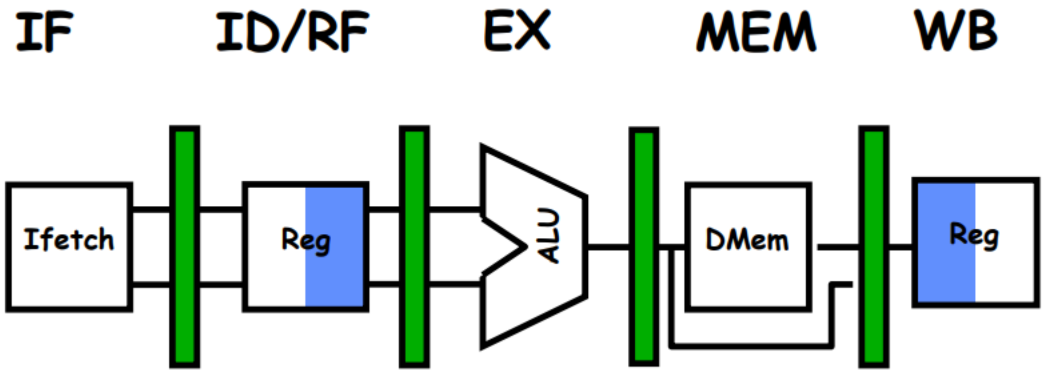
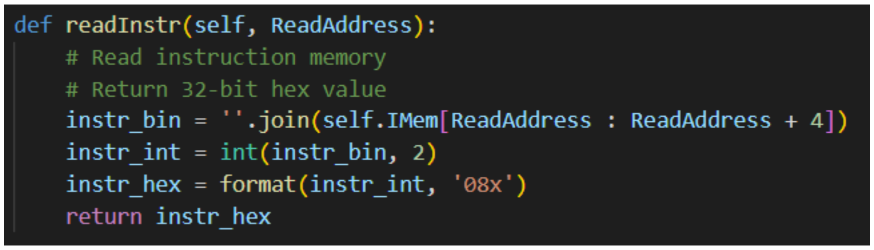


Fig 1.1: Single-Cycle Processor Datapath

Instruction Fetch Stage:

The **Instruction Fetch (IF)** stage retrieves the next instruction to be executed.

- Code Reference: The *readInstr()* method in the *InsMem* class reads instructions from the instruction memory:



```
def readInstr(self, ReadAddress):
    # Read instruction memory
    # Return 32-bit hex value
    instr_bin = ''.join(self.IMem[ReadAddress : ReadAddress + 4])
    instr_int = int(instr_bin, 2)
    instr_hex = format(instr_int, '08x')
    return instr_hex
```

Fig 2.1: Code snippet of instruction fetch

It combines 32-bit instruction data from the instruction memory, converts it to an integer, and formats it as a hex value.

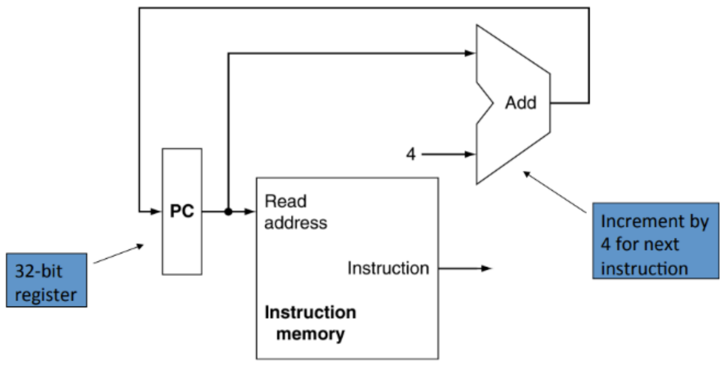


Fig 2.2: Instruction Fetch Datapath

Instruction decode stage:

The Instruction Decode (ID) stage decodes the fetched instruction to determine its type (R-Type, I-Type, etc.) and operands.

- **Code Reference:** The *decode_execute()* method identifies the opcode and determines the instruction type.
- Depending on the instruction type, it extracts operands and immediate values.

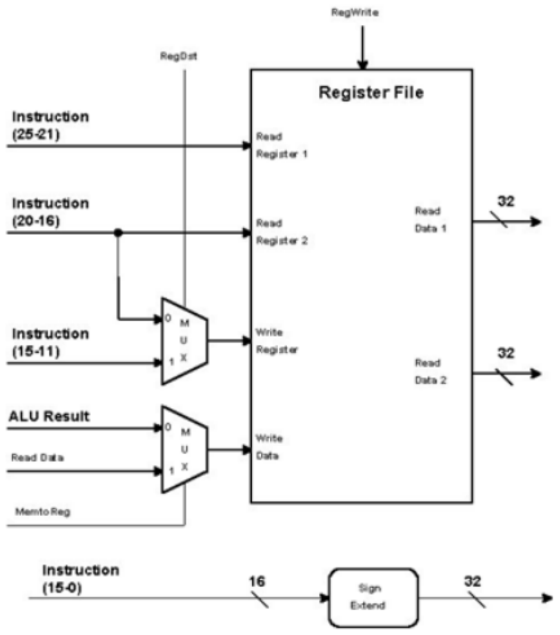


Fig 3.1: Instruction decode stage (Reaz, Jalil, & Rahman, 2012)

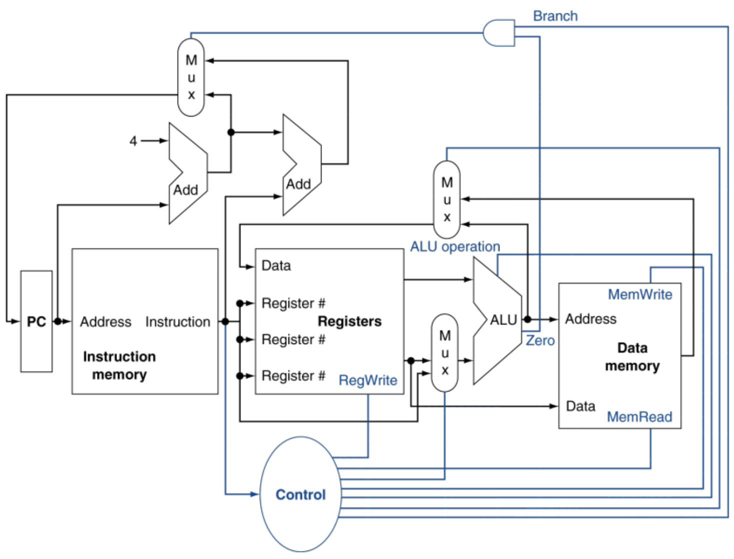
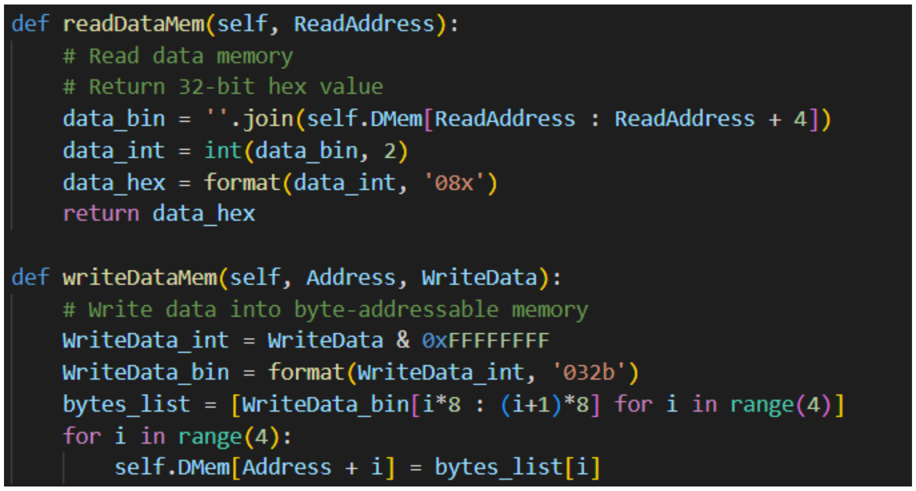


Fig 3.2: Control Signal Flow (How the instruction is decoded & control signals are generated)

Memory Access Stage:

The Memory Access (MEM) stage reads or writes data to the memory.

- **Code Reference:** Memory access is handled by the *readDataMem()* and *writeDataMem()* methods in the *DataMem* class



```
def readDataMem(self, ReadAddress):
    # Read data memory
    # Return 32-bit hex value
    data_bin = ''.join(self.DMem[ReadAddress : ReadAddress + 4])
    data_int = int(data_bin, 2)
    data_hex = format(data_int, '08x')
    return data_hex

def writeDataMem(self, Address, WriteData):
    # Write data into byte-addressable memory
    WriteData_int = WriteData & 0xFFFFFFFF
    WriteData_bin = format(WriteData_int, '032b')
    bytes_list = [WriteData_bin[i*8 : (i+1)*8] for i in range(4)]
    for i in range(4):
        self.DMem[Address + i] = bytes_list[i]
```

Fig 4.1: Code snippet of read and write methods of DataMem class

- The *DataMem* outputs the final memory state after program execution.

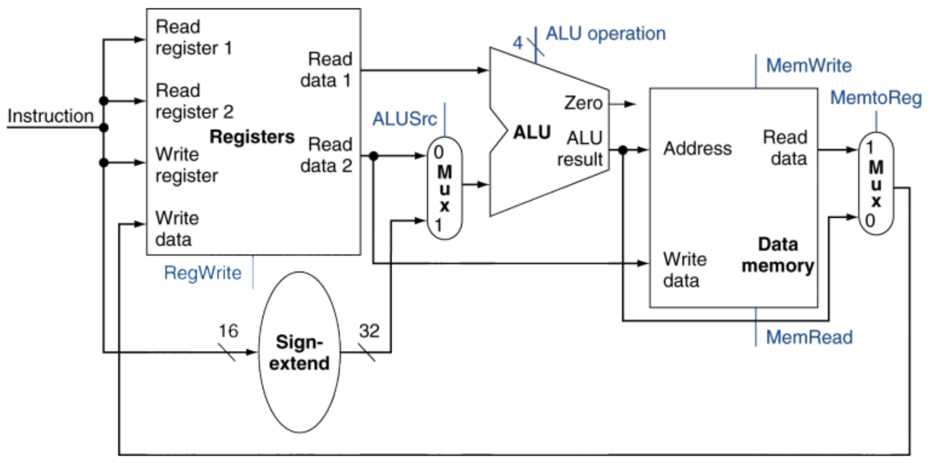


Fig 4.2: Datapath for Memory Access in R and I-Type Instructions

ALU Execution Stage:

The ALU Execution (EX) stage performs arithmetic or logical operations using the operands decoded in the previous stage.

- **Code Reference:** The *calculate_R()* and *calculate_I()* methods implement R-Type and I-Type instructions.

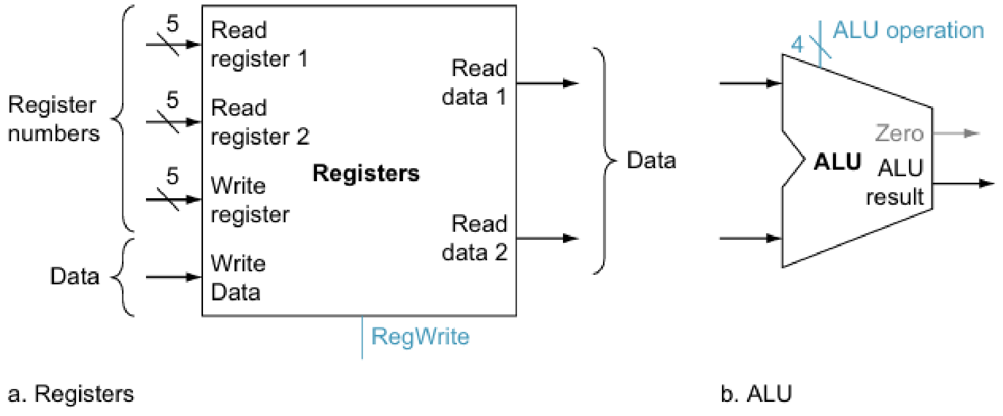


Fig 5.1: The register file and the ALU needed to implement R-format ALU operations

Register Write Back Stage:

The Write-Back (WB) stage writes the result of the computation or memory access back to the register file.

- **Code Reference:** The *writeRF()* method in *RegisterFile* class updates the registers.

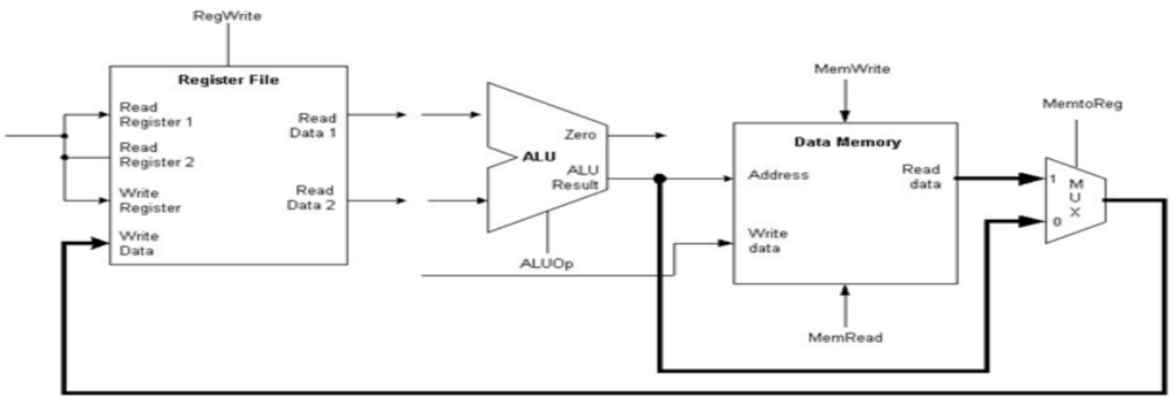


Fig 6.1: Architecture of Write Back Stage (Reaz, Jalil, & Rahman, 2012)

Immediate Sign Extension:

Immediate values are extended to 32 bits using sign-extension to ensure correct arithmetic.

- **Code Reference:** The *sign_extend()* method handles the extension.

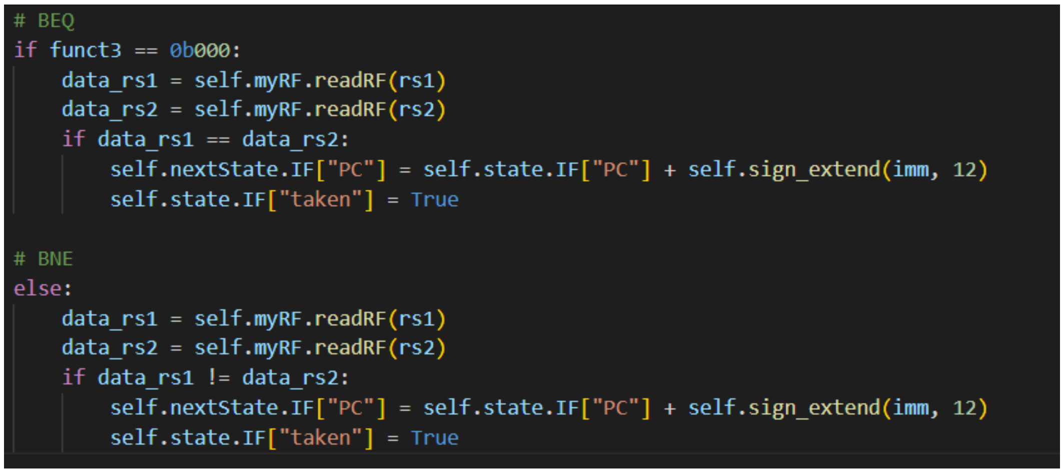
ImmSrc	Immediate Fields	Sign-Bit Location	Bit Range	Description
0	{Instr[31], Instr[30:20]}	11	31:20	12-bit signed immediate (I-type)
1	{Instr[31], Instr[31:25], Instr[11:7]}	11	31:25, 11:7	12-bit signed immediate (S-type)
10	{Instr[31], Instr[7], Instr[30:25], Instr[11:8]}	12	31, 7, 30:25, 11:8	13-bit signed immediate (B-type)
11	{Instr[31], Instr[19:12], Instr[20], Instr[30:21]}	20	31, 19:12, 20, 30:21	21-bit signed immediate (J-type)

Fig 7.1: Immediate Sign Extension Table

Branch Instructions:

Branch instructions like beq and bne alter the program counter if conditions are met.

- **Code Reference:** They are included inside *decode_execute()* function.



```
# BEQ
if funct3 == 0b000:
    data_rs1 = self.myRF.readRF(rs1)
    data_rs2 = self.myRF.readRF(rs2)
    if data_rs1 == data_rs2:
        self.nextState.IF["PC"] = self.state.IF["PC"] + self.sign_extend(imm, 12)
        self.state.IF["taken"] = True

# BNE
else:
    data_rs1 = self.myRF.readRF(rs1)
    data_rs2 = self.myRF.readRF(rs2)
    if data_rs1 != data_rs2:
        self.nextState.IF["PC"] = self.state.IF["PC"] + self.sign_extend(imm, 12)
        self.state.IF["taken"] = True
```

Fig 8.1: Code snippet of branch instructions

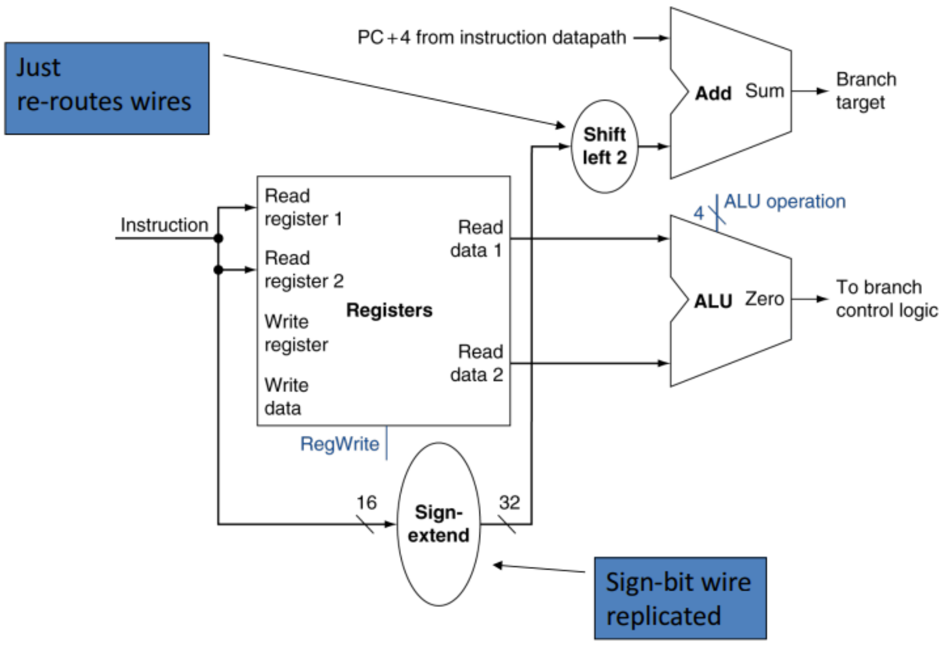


Fig 8.2: Branch Instructions

Instruction Categories:

All the following instructions are handled in *decode_execute()* function.

- R-Type Instructions
  Operations: ADD, SUB, AND, OR, XOR
- I-Type Instructions
  Operations: ADDI, XORI, ORI, ANDI
- Branch Instructions (B-Type)
  Operations: BEQ, BNE
- Jump Instructions (J-Type)
  Operation: JAL.
- Load and Store Instructions
  Operations: LW, SW.

State Management:

The processor operates in a structured manner by maintaining the status of each stage using a state dictionary. Each stage of the pipeline (Instruction Fetch, Decode, Execute, Memory Access, and Write Back) is represented with specific attributes in the *State* class.

- IF State: Tracks the program counter (PC) and nop status for the fetch stage.
- ID State: Holds the instruction details, hazard flags, and relevant data for decoding.
- EX State: Manages operand values, ALU operations, and immediate values.
- MEM State: Tracks memory operations like read or write, including data addresses.
- WB State: Stores results for write-back operations.

Halt mechanism:

The halt mechanism is implemented to stop the processor execution gracefully once all instructions are executed or if a halt condition is encountered. This is controlled by a specific flag (nop) within the state class.

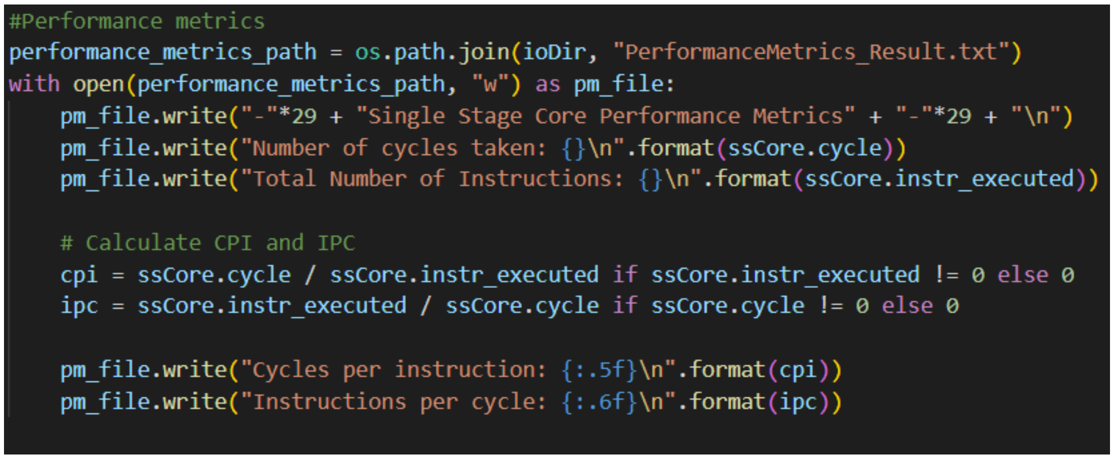
How Halt Works?

- **Initialization**: All nop flags are set to False at the beginning of execution.
- **Condition for Halt**: The processor halts when the nop flag in the instruction fetch (IF) stage is True, or when the halt opcode is detected.
- **Final State Capture**: When the halt condition is met, the final state of the register file and memory is output to files.

Performance Metrics:

To analyze the efficiency of the single-stage processor, the following metrices are calculated:

- Number of Cycles: Total cycles taken for program execution are tracked.
- Instructions Executed: Count of instructions completed during execution.
- Cycles Per Instruction (CPI): Indicates the average number of cycles needed to execute one instruction.
- Instructions Per Cycle (IPC): Reflects the instruction throughput of the processor.



```python
#Performance metrics
performance_metrics_path = os.path.join(ioDir, "PerformanceMetrics_Result.txt")
with open(performance_metrics_path, "w") as pm_file:
    pm_file.write("-"*29 + "Single Stage Core Performance Metrics" + "-"*29 + "\n")
    pm_file.write("Number of cycles taken: {}\n".format(ssCore.cycle))
    pm_file.write("Total Number of Instructions: {}\n".format(ssCore.instr_executed))

    # Calculate CPI and IPC
    cpi = ssCore.cycle / ssCore.instr_executed if ssCore.instr_executed != 0 else 0
    ipc = ssCore.instr_executed / ssCore.cycle if ssCore.cycle != 0 else 0

    pm_file.write("Cycles per instruction: {:.5f}\n".format(cpi))
    pm_file.write("Instructions per cycle: {:.6f}\n".format(ipc))
```

Fig 9.1: Code snippet of performance metrics

Input:

The program accepts the following files as inputs:

- imem.txt: Contains the instruction memory data required for execution.
- dmem.txt: Contains the data memory information, used for load and store operations.

Output:

The program generates the following files as outputs:

- SS_RFResult.txt: Tracks the state of the register file after each cycle execution.
- StateResult_SS.txt: This file logs the processor's state, including program counter (PC) and pipeline status, after each cycle.
- SS_DMEMResult.txt: Captures the final state of the data memory after execution.
- PerformanceMetrics_Result.txt: Reports the performance metrics, such as the number of cycles, instructions executed, CPI (Cycles Per Instruction), and IPC (Instructions Per Cycle).

Summary:

The first phase of the project demonstrates the implementation of a single-stage RISC-V processor. The work showcases a fundamental understanding of processor architecture and provides a foundation for implementing advanced architectures.

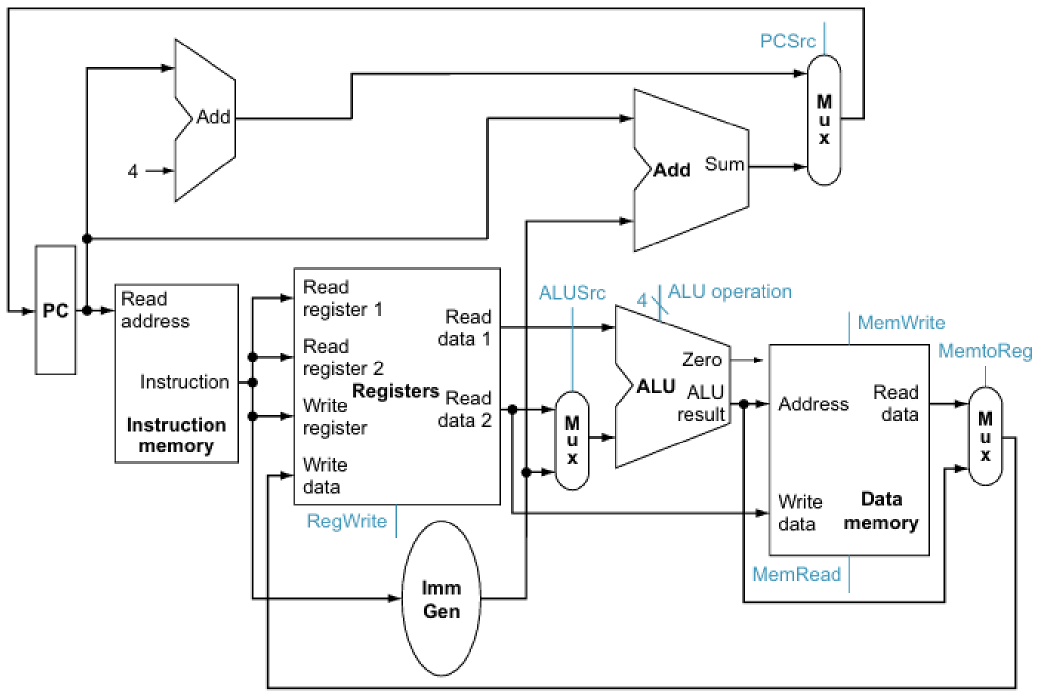


Fig 10.1: Single Stage Processor Implementation (Patterson & Hennessy, 2020)

References:

Patterson, D., & Hennessy, J. (2020). *Computer Organization And Design RISV-V Edition.* Cambridge: Morgan Kaufmann.

Reaz, M., Jalil, J., & Rahman, L. (2012). Single Core Hardware Modeling of 32-bit MIPS RISC Processor with A Single Clock. *Research Journal of Applied Sciences, Engineering and Technology*, 825-832.