Detailed Explanation of Pipeline Simulation in Python

Mahyar Mohammadian
Parsa Haghighatgo

July 16, 2024

1 Introduction

This document provides a detailed explanation of a Python program designed to simulate the pipeline stages of a CPU. The program handles basic arithmetic instructions and memory operations, incorporating mechanisms for detecting and handling hazards and stalls. It outputs the pipeline stages for each instruction and the final register values after executing each instruction.

2 Code Listing and Explanation

2.1 Initialization

The program starts by importing necessary libraries and initializing the registers and main memory.

Listing 1: Initialization

```
from colorama import init, Fore, Style

# Initialize colorama
init()

# Define the initial register values and main memory
registers = {
    "$s0": -10,
    "$s1": -10,
    "$s2": -10,
    "$s3": -10,
    "$s4": -10,
    "$s5": -10,
    "$s6": -10,
    "$s7": -10,
    "$t0": -10,
    "$t1": -10,
    "$t2": -10,
    "$t3": -10,
    "$t4": -10,
    "$t5": -10,
    "$t6": -10,
    "$t7": -10
}
mainMem = [0] * 1000
```

Explanation:

- `colorama` is used for colored terminal output.
- The `registers` dictionary initializes general-purpose registers with a value of -10.
- `mainMem` initializes the main memory with 1000 zeroes.

2.2 Stall Detection in Fetch and Decode Stages

Functions to detect stalls in the fetch and decode stages.

Listing 2: Stall Detection

```
def stallFetch(pipOut):
    nStart = 0
    clockcyle = 0
    for pip in pipOut:
        if len(pip) > clockcyle:
            clockcyle = len(pip)
    for i in range(1, clockcyle + 1):
        for pip in pipOut:
            if i < 2:
                if pip[i - 1] == "IF":
                    nStart += 1
            else:
                if pip[i - 1] == "IF" or (pip[i - 1] == "st" and pip[i - 2] ==
                    "IF"):
                    nStart += 1
        if nStart < 3:
            return i
        else:
            nStart = 0
    return nStart

def stallDecode(pipOut, start):
    nStart = 0
    clockcyle = 0
    for pip in pipOut:
        if len(pip) > clockcyle:
            clockcyle = len(pip)
    for i in range(start + 1, clockcyle + 1):
        for pip in pipOut:
            if i < 2:
                if pip[i - 1] == "ID":
                    nStart += 1
            else:
                if i < len(pip):
                    if pip[i - 1] == "ID" or (pip[i - 1] == "st" and pip[i - 2]
                        == "ID"):
                        nStart += 1
                else:
                    continue
        if nStart < 2:
            return i
        else:
            nStart = 0
    return nStart
```

Explanation:

- `stallFetch`: Checks for stalls in the fetch stage by counting the number of "IF" stages and determining if it exceeds the allowable limit.
- `stallDecode`: Similar to `stallFetch`, but checks for stalls in the decode stage.

2.3 Hazard Detection

Function to detect hazards between instructions.

Listing 3: Hazard Detection

```
def hazard(input, index, inputList):
    flagr1 = 0
    if input[2] == inputList[index - 2][1]:
        flagr1 = 1
    elif input[3] == inputList[index - 2][1]:
        flagr1 = 1
    return flagr1
```

Explanation:

- `hazard`: Compares the operands of the current instruction with the destination register of previous instructions to detect data hazards.

2.4 Pipeline Simulation

Function to simulate the pipeline stages for each instruction.

Listing 4: Pipeline Simulation

```
def pipline(input, index, inputList, pipOut):
    output = []
    if index == 1:
        output.append("IF")
        output.append("ID")
        output.append("EX")
        output.append("ME")
        output.append("WB")
    elif index == 2:
        flagH = hazard(input, index, inputList)
        if flagH == 1:
            output.append("IF")
            output.append("ID")
            output.append("st")
            output.append("EX")
            output.append("ME")
            output.append("WB")
        else:
            output.append("IF")
            output.append("ID")
            output.append("EX")
            output.append("ME")
            output.append("WB")
    else:
        flagF = stallFetch(pipOut)
        flagH = hazard(input, index, inputList)
        if flagF > 1:
            for i in range(flagF - 1):
                output.append("st")
        output.append("IF")
        start = len(output)
        flagD = stallDecode(pipOut, flagF)
        dstalls = flagD - len(output) - 1
        if dstalls > 0:
            for i in range(dstalls):
                output.append("st")
        output.append("ID")
        if flagH == 1:
            output.append("st")
        output.append("EX")
        output.append("ME")
        output.append("WB")
    return output
```

Explanation:

- `pipline`: Determines the pipeline stages for each instruction. It handles stalls and hazards by inserting stall cycles ("st") when necessary.

2.5 Output Printing

Functions to print the pipeline stages, main memory, and register values.

Listing 5: Output Printing

```
def print_pipeline_output(pipOut):
    print("Pipeline for these Instructions")
    # Find the maximum length of any pipeline stage to align columns
    max_length = max(len(pip) for pip in pipOut)

    # Color map
    color_map = {
        "IF": Fore.GREEN,
        "ID": Fore.YELLOW,
        "EX": Fore.CYAN,
        "ME": Fore.MAGENTA,
        "WB": Fore.BLUE,
        "st": Fore.RED
    }

    # Print header
    header = "Cycle".ljust(6) + "\t|\t" + "\t|\t".join([f"Stage {i + 1}".ljust(4) for i in range(max_length)])
    print(header)
    print("-" * len(header))
    separator = "-" * len(header)

    # Print each pipeline stage
    for i, pip in enumerate(pipOut, start=1):
        cycle_str = str(i).ljust(6)
        stages_str = "\t|\t".join([color_map.get(stage, "") + stage.ljust(4) + Style.RESET_ALL for stage in pip])
        print(f"{cycle_str} | {stages_str}")
    return separator

def memoryPrint(memory, sep):
    print(sep)
    print()
    print("Main Memory after executing these Instructions")
    print(memory)
    print()

def print_register_values(register_history):
    print(sep)
    print()
    print("Register values after executing each instruction")
    print()

    headers = ["Register"] + [f"Instr {i + 1}" for i in range(len(register_history))]
    header_row = "\t|\t".join(headers)
    print(Fore.YELLOW + header_row + Style.RESET_ALL)
    print(Fore.YELLOW + "-" * len(header_row) + Style.RESET_ALL)

    for reg in registers.keys():
        row = [Fore.CYAN + reg + Style.RESET_ALL] + [str(registers_after_exec[reg]) for registers_after_exec in
                                                     register_history]
        print("\t|\t".join(row))
```

Explanation:

- `print_pipeline_output`: Prints the pipeline stages for each instruction with appropriate colors.

- `memoryPrint`: Prints the main memory after executing all instructions.
- `print_register_values`: Prints the values of all registers after executing each instruction.

2.6 Memory Handling and Instruction Execution

Functions to handle memory operations and execute instructions.

Listing 6: Memory Handling and Execution

```
def memoryHandle(input):
    if input[0] == "lw":
        address = registers[input[3]] + int(input[2][1:])
        registers[input[1]] = mainMem[address]
        return 1
    elif input[0] == "sw":
        address = registers[input[3]] + int(input[2][1:])
        mainMem[address] = registers[input[1]]
        return 2
    return 0

def execute_instruction(instruction):
    opcode = instruction[0]
    dest = instruction[1]
    src1 = instruction[2]
    src2 = instruction[3]

    if src2.startswith("#"):
        immediate = int(src2[1:])
        if opcode == "add":
            registers[dest] = registers[src1] + immediate
        elif opcode == "sub":
            registers[dest] = registers[src1] - immediate
        elif opcode == "mul":
            registers[dest] = registers[src1] * immediate
        elif opcode == "div":
            if immediate != 0:
                registers[dest] = registers[src1] // immediate
            else:
                print(Fore.RED + "Error: Division by zero" + Style.RESET_ALL)
        elif opcode == "ori":
            registers[dest] = registers[src1] | immediate
    else:
        if opcode == "add":
            registers[dest] = registers[src1] + registers[src2]
        elif opcode == "sub":
            registers[dest] = registers[src1] - registers[src2]
        elif opcode == "mul":
            registers[dest] = registers[src1] * registers[src2]
        elif opcode == "div":
            if registers[src2] != 0:
                registers[dest] = registers[src1] // registers[src2]
            else:
                print(Fore.RED + "Error: Division by zero" + Style.RESET_ALL)
        elif opcode == "addi":
            registers[dest] = registers[src1] + int(src2)
        elif opcode == "subi":
            registers[dest] = registers[src1] - int(src2)
        elif opcode == "muli":
            registers[dest] = registers[src1] * int(src2)
        elif opcode == "divi":
            if int(src2) != 0:
                registers[dest] = registers[src1] // int(src2)
            else:
                print(Fore.RED + "Error: Division by zero" + Style.RESET_ALL)
        elif opcode == "ori":
            registers[dest] = registers[src1] | int(src2)
    return 1
```

Explanation:

- `memoryHandle`: Handles `lw` and `sw` instructions, updating the registers or main memory.
- `execute_instruction`: Executes arithmetic and logical instructions, updating the appropriate registers.

2.7 Main Execution

The main execution part reads the input file, simulates the pipeline, and prints the results.

Listing 7: Main Execution

```
# File path
file_path = "C:\\Users\\beta\\Downloads\\pp.txt"

# Open file for reading
with open(file_path, 'r') as file:
    code = file.read()
lines = code.splitlines()
inputList = []
pipOut = []

# Iterate the input file
for line in lines:
    input = line.split(" ")
    ins = input[0]
    args = input[1].split(",")
    listCons = [ins, args[0], args[1], args[2]]
    inputList.append(listCons)

register_history = []

for instruction in inputList:
    index = inputList.index(instruction) + 1
    output = pipline(instruction, index, inputList, pipOut)
    pipOut.append(output)
    memSetFlag = memoryHandle(instruction)
    regSetFlag = execute_instruction(instruction)
    register_history.append(registers.copy())  # Store a copy of register
        values after each instruction

# Print the pipeline output beautifully
sep = print_pipeline_output(pipOut)

# Print the main memory
memoryPrint(mainMem, sep)

# Print final register values after each instruction
print_register_values(register_history)
```

Explanation:

- Reads the input file containing instructions.
- Parses each instruction and simulates its execution through the pipeline.
- Stores register values after each instruction.
- Prints the pipeline stages, main memory, and final register values.