实验报告

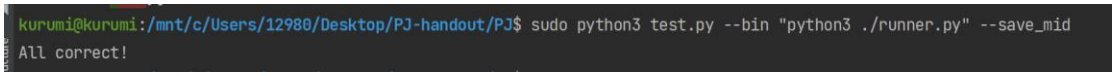
组号：3

组员：谢志康、俞浩珅

贡献度与分工情况：贡献度大致各 50%，我们项目是两人协力完成的，从最初的思路架构到代码编写，大约各写了一半的函数。最后整合时碰到许多问题，这个耗费了我们比较多的时间，最后也是一起在 debug 解决的。

Note:
压缩文件中 CPU.py 是我们 cpu 的所有实现逻辑， runner.py 是最终运行程序。
_debug 后缀程序可以算作简陋的前端（显示每条指令 cpu 的内部状态）
使用命令 python3 test.py --bin "python3 ./runner.py" --save_mid 在 linux 下运行

0. 功能正确性截图



1. CPU 架构概览
参考文章——
CSAPP：第四章——处理器体系结构(上)_csapp 第四章-CSDN 博客
Y86-64 的顺序实现_取指译码执行访存写回-CSDN 博客

简要分析——
在 Y86-64 模拟器中，CPU 的主要组成部分包括：

寄存器 (Register)**: 用于存储临时数据和指令执行的中间结果。寄存器可以快速地被 CPU 访问，以支持高效的数据处理和计算。
内存 (Memory)**: 用于存储程序的指令和数据。CPU 通过读取和写入内存中的数据来执行程序。
条件代码 (Condition Code)**: 存储关于最近执行的运算的状态信息，例如运算结果是否为零（Zero Flag, ZF）、是否为负（Sign Flag, SF）以及是否溢出（Overflow Flag, OF）。
程序计数器 (Program Counter, PC)**: 存储当前正在执行或即将执行的指令的地址。

此外还应有：
状态（STATE）
Y86-64 状态码可以取以下值，1 表示执行正常，2 表示执行一条 halt 指令，3 遇到非法读写，4 表示遇到非法指令代码，其中 2、3、4 则为异常状态码。Y86-64 的状态码为异常时，程序会停止（没有异常处理），一般完整的指令集定义都会有异常处理程序.

根据以上思路，我们可以定义三个类：
class Memory　class Register　class cc
PC 由于操作太过简单可以无需定义类，后续定义函数实现更新即可

```
# 条件代码类，用于存储和管理处理器的条件代码（Zero Flag, Sign Flag, Overflow Flag）。
1 usage
class ConditionCode:...
# 内存类，用于模拟内存的读写操作。
2 usages
class Memory:...
# 寄存器类，用于模拟寄存器的读写操作。
2 usages
class Register:...
```

2. 指令执行流程

CPU 的操作可以划分为以下几个阶段：

取指 (Fetch)

在这一阶段，CPU 从内存中读取指令。这涉及到以下函数：

`fetch(cpu)`: 这个函数从内存中读取当前程序计数器（PC）指向的指令，并对其进行初步解析，以确定后续操作。
`handle_fetch_errors(cpu, instr_valid)`: 用于处理取指阶段可能出现的错误，并根据指令的有效性更新 CPU 状态。

译码 (Decode)

在译码阶段，CPU 解析指令并准备必要的操作数。这包括以下函数：

`set_src(cpu)`: 根据当前指令设置源操作数寄存器 `srcA` 和 `srcB`。
`set_dst(cpu)`: 根据当前指令设置目标操作数寄存器 `dstE` 和 `dstM`。
`decode(cpu)`: 解析指令并从寄存器中读取必要的值以准备执行。

执行 (Execute)

执行阶段是指令逻辑的核心实现。这一阶段的函数包括：

`execute(cpu)`: 执行指令的逻辑，这可能包括算术运算、逻辑运算等。该函数根据指令类型处理不同的操作逻辑。

访存 (Memory Access)

如果指令需要，这一阶段会对内存进行读写操作。相关函数包括：

`memory(cpu)`: 根据指令需求处理内存的读写操作。例如，加载指令会从内存中读取数据，而存储指令会向内存写入数据。
`read_memory(cpu, mem_addr)`: 从指定的内存地址读取数据。
`write_memory(cpu, mem_addr, mem_data)`: 向指定的内存地址写入数据。

写回 (Write Back)

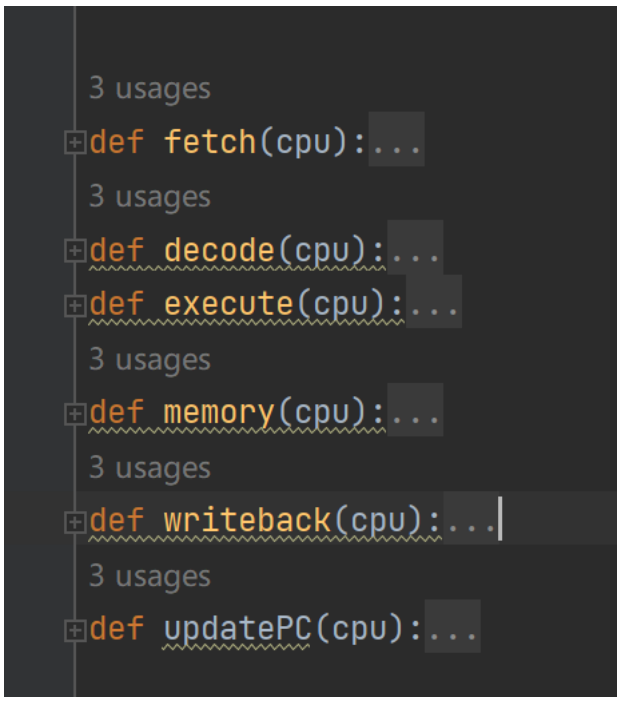
这一阶段将执行结果写回到寄存器中。相关函数：

`writeback(cpu)`: 将计算结果或内存读取的数据写回到寄存器。

更新程序计数器 (Update PC)

在每个指令周期的最后，更新程序计数器，以指向下一条要执行的指令。这涉及到：

`updatePC(cpu)`: 更新程序计数器(PC)，以指向下一条指令或根据跳转指令更新指令地址。



3. 运行
使用 `runner.py` 运行模拟器。这个脚本负责实例化 CPU 对象，加载指令到内存，然后按顺序执行每个指令周期。

4. 流水线
在每个阶段中引入流水寄存器，存储上一阶段的输出和下一阶段的输入

```
self.valP = 0            # 下一指令地址
self.valA = ZERO
self.valB = ZERO
self.valE = ZERO
self.valM = ZERO
self.valC = ZERO
```

思路参考图——

举例说明——

```
2 usages
def decode(cpu):
    set_src(cpu)
    set_dst(cpu)
    # 读取寄存器中的 valA 和 valB
    cpu.valA, cpu.valB = cpu.Reg.read(cpu.srcA, cpu.srcB)
```

我们在译码阶段就同时取出寄存器中的两个源操作数，当后续需要使用时，就可以直接获取。

5. 前端

没有用 html 或 qt 实现比较精美的前端，我们就在 python 下运行算是也写了个前端（？）

在任意输入的 yo 文件中，我们 cpu 执行时能告诉我们运行的每一条信息，并且返回执行该语句后 cpu 的内部所有状态。

图例如下——

```
0
正在执行指令: irmovq
Fetch: icode = 3, ifun = 0, next PC = 0xa
Debug: Before setting valA for CALL - valP (Next Address): 10
Debug: Execute - After ALU operation, valA: 1
Debug: After Execute - valE: 01000000, valA (Return Address): 1
Writeback: Wrote 01000000 to RAX
{'PC': 0, 'Register': {'RAX': 0, 'RCX': 0, 'RDX': 0, 'RBX': 0, 'RSP': 0, 'RBP': 0, 'RSI': 0, 'RDI': 0, 'R8': 0, 'R9': 0, 'R10': 0, 'R11':
 0, 'R12': 0, 'R13': 0, 'R14': 0}, 'Memory': {0: 127024, 8: 27038778780745728, 16: 192560, 24: 16914579456}, 'CC': {'ZF': 1, 'SF': 0, 'OF':
  0}, 'STAT': '1'}
10
正在执行指令: xorq
Fetch: icode = 6, ifun = 3, next PC = 0xc
Debug: Before setting valA for CALL - valP (Next Address): 12
ALU Add: ZF = 1, SF = 0, OF = 0
Debug: Execute - After ALU operation, valA: 00000000
Debug: After Execute - valE: 00000000, valA (Return Address): 00000000
Writeback: Wrote 00000000 to RSP
{'PC': 10, 'Register': {'RAX': 1, 'RCX': 0, 'RDX': 0, 'RBX': 0, 'RSP': 0, 'RBP': 0, 'RSI': 0, 'RDI': 0, 'R8': 0, 'R9': 0, 'R10': 0, 'R11':
 0, 'R12': 0, 'R13': 0, 'R14': 0}, 'Memory': {0: 127024, 8: 27038778780745728, 16: 192560, 24: 16914579456}, 'CC': {'ZF': 1, 'SF': 0, 'OF':
  0}, 'STAT': '1'}
12
正在执行指令: pushq
Fetch: icode = 10, ifun = 0, next PC = 0xe
地址: 00000000
write值 0 地址: 00000000
Debug: write_memory - Successfully wrote 01000000 to address 0
```