PJ项目报告

ps. 由于cache创新后影响内存存放的数据，故main分支中只含要求实现功能的CPU，以保证能通过测试。cache部分和前端部分代码放在新开的cache分支中。

一、CPU的设计

本项目实现了一个简化的CPU模拟器，旨在模拟基本的计算机体系结构和指令执行过程。以下将详细介绍CPU的设计，包括指令集架构、寄存器组、内存模型、条件码和状态码、以及各个功能模块的实现。

一、指令集架构（ISA）

1. 支持的指令

CPU模拟器支持一系列基本的指令集，这些指令涵盖数据传输、算术逻辑运算、控制流和栈操作等。主要指令包括：

- **数据传输指令**：
  - `rrmovq`：寄存器到寄存器的数据传送。
  - `irmovq`：立即数加载到寄存器。
  - `rmmovq`：寄存器的数据存储到内存。
  - `mrmovq`：从内存加载数据到寄存器。
- **算术逻辑运算指令**：
  - `opq`：对两个寄存器进行算术或逻辑运算，支持操作包括加法、减法、按位与、按位异或等。
  - `iopq`：立即数与寄存器进行算术或逻辑运算。
- **控制流指令**：
  - `jmp`：无条件跳转。
  - `jxx`：条件跳转，支持条件包括小于、等于、大于等。
  - `call`：函数调用，将返回地址压入栈中。
  - `ret`：函数返回，从栈中弹出返回地址并跳转。
- **栈操作指令**：
  - `pushq`：将寄存器压入栈。
  - `popq`：从栈中弹出数据到寄存器。

2. 指令格式

每条指令由操作码（icode）和功能码（ifun）组成：

- **icode**：表示指令的类型，例如数据传输、算术运算、控制流等。
- **ifun**：表示具体的功能，例如算术运算中的加减乘除，条件跳转中的具体条件等。

指令可能还包含：

- **寄存器标识符（rA和rB）**：指定源和目的寄存器。
- **立即数（valC）**：用于立即数指令，存储具体的数值。
- **程序计数器（PC）**：指向当前指令的地址。

二、CPU流程设计

CPU分为6层设计，分别是：

1. **取指（Fetch）阶段**：
   - 从指令缓存或内存中读取指令字节。
   - 提取操作码（icode）和功能码（ifun）。
   - 计算下一个程序计数器（PC）的值。
2. **译码（Decode）阶段**：
   - 读取指令中的寄存器标识符（rA和rB）。
   - 从寄存器文件中获取源操作数的值。
3. **执行（Execute）阶段**：
   - 根据指令类型执行ALU运算。
   - 对于算术逻辑指令，执行相应的计算操作。
   - 更新条件码（CC），如零标志（ZF）、符号标志（SF）和溢出标志（OF）。
4. **访存（Memory）阶段**：
   - 对需要访问内存的指令，进行内存读写操作。
   - 处理数据缓存，检查缓存命中，进行加载或存储操作。
5. **写回（Write Back）阶段**：
   - 将执行结果或从内存读取的数据写回目的寄存器。
   - 更新寄存器文件中的值。
6. **更新（Update_PC）阶段**：
   - 更新程序计数器（PC）的值

三、寄存器组设计

1. 通用寄存器

CPU包含16个64位的通用寄存器，编号从0到15，对应的寄存器名称如下：

- `%rax`（0）
- `%rcx`（1）
- `%rdx`（2）
- `%rbx`（3）
- `%rsp`（4）：栈指针寄存器
- `%rbp`（5）：基址指针寄存器
- `%rsi`（6）
- `%rdi`（7）
- `%r8`（8）
- `%r9`（9）
- `%r10`（10）
- `%r11`（11）

- `%r12`（12）
- `%r13`（13）
- `%r14`（14）
- `%r15`（15）：无效寄存器（用于表示无效的寄存器操作）

2. 寄存器文件

- 使用一个数组或映射来存储所有寄存器的值，支持根据寄存器编号快速访问和更新。
- 在译码和写回阶段，读取和写入寄存器文件。

四、内存模型设计

1. 主存储器

- 使用一个字节数组Memory模拟主存储器，大小为`MAXSIZE`字节。
- 支持按字节寻址，存储指令和数据。

2. 内存访问

- 读取或写入内存时，首先检查地址的合法性，防止越界访问。
- 对于访存指令，在访存阶段进行实际的内存操作。

五、条件码和状态码

1. 条件码（CC）

- **零标志（ZF）**：结果为零时设置为1，否则为0。
- **符号标志（SF）**：结果为负数时设置为1，否则为0。
- **溢出标志（OF）**：有符号运算发生溢出时设置为1。

条件码用于条件跳转和条件数据传输指令，以决定是否满足特定条件。

2. 状态码（Stat）

表示CPU的当前状态，包括：

- **AOK（0）**：正常执行。
- **HLT（1）**：程序停止（遇到`halt`指令）。
- **ADR（2）**：内存地址非法或越界。
- **INS（3）**：非法指令编码。

状态码用于在程序执行过程中检测异常情况并进行相应处理。

六、功能模块的实现

1. 取指模块

- **功能**：从指令缓存或主存中读取指令字节，提取操作码和功能码，计算valP（下一个指令地址）。
- 实现细节
  :
  - 计算指令地址对应的缓存索引和标签，检查指令缓存是否命中。
  - 如果命中，从缓存中读取指令字节；否则，从主存加载指令到缓存。

- 提取icode和ifun。
- 根据指令类型，确定valP的增量。

2. 译码模块

- **功能**：解析指令操作数，读取源寄存器的值。
- 实现细节：
  - 提取寄存器标识符rA和rB。
  - 从寄存器文件中读取 `valA` 和 `valB`。
  - 对于某些指令，valC需要从指令后续字节中读取。

3. 执行模块

- **功能**：执行算术逻辑运算，计算内存地址，更新条件码。
- 实现细节：
  - 根据icode确定操作类型。
  - 对于算术逻辑指令，调用ALU执行计算，如 `valE = valA op valB`。
  - 更新条件码 `CC`。
  - 对于内存访问指令，计算有效地址 `valE`。

4. 访存模块

- **功能**：对内存进行读取或写入操作。
- 实现细节：
  - 对于读取操作，从数据缓存或内存中获取数据 `valM`。
  - 对于写入操作，将数据写入数据缓存或内存。
  - 处理数据缓存的命中和未命中情况。

5. 写回模块

- **功能**：将执行结果或内存读取的数据写回目的寄存器。
- 实现细节：
  - 根据指令类型，确定需要写回的寄存器和数据。
  - 更新寄存器文件中的对应寄存器值。

6. PC更新模块

- **功能**：更新程序计数器PC，指向下一条指令。
- 实现细节：
  - 对于顺序执行的指令，PC = valP。
  - 对于跳转和调用指令，PC可能更新为valC或 `valM`。
  - 根据条件码和跳转条件，决定是否执行跳转。

七、ALU（算术逻辑单元）设计

- **功能**：执行基本的算术和逻辑运算，包括加法、减法、按位与、按位异或等。
- 实现细节
  ：
  - 接受操作数`valA`和`valB`，以及操作类型。
  - 输出结果`valE`，并更新条件码`CC`。

八、控制流的处理

1. 无条件跳转

- 对于`jmp`指令，直接更新`PC = valC`。

2. 条件跳转

- 根据条件码`CC`和指令指定的条件ifun，判断跳转条件是否满足。
- 如果满足，PC = valC；否则，`PC = valP`。

3. 函数调用和返回

- **调用（call）**：
  - 将当前的valP（下一条指令的地址）压入栈，`rsp = rsp - 8`。
  - 更新`PC = valC`，跳转到被调用函数的地址。
- **返回（ret）**：
  - 从栈中弹出返回地址，`rsp = rsp + 8`。
  - 更新`PC = valM`，跳转到返回地址。

九、异常和错误处理

- **非法指令**：如果icode不在支持的指令集范围内，设置状态码`Stat = INS`，停止执行。
- **内存越界**：在内存访问时，如果地址不合法，设置`Stat = ADR`。
- **程序结束**：遇到`halt`指令或执行完成，设置`Stat = HLT`。

十、整体执行流程

1. 初始化CPU的状态，包括寄存器、内存、缓存和条件码。
2. 进入主循环，在`Stat == AOK`的条件下反复执行每个流水线阶段。
3. 在每个周期内，按照取指、译码、执行、访存和写回的顺序处理指令。
4. 根据指令执行的结果和状态码，决定继续执行还是停止。

通过上述设计，我构建了一个功能完整的CPU模拟器，能够模拟指令的取指、译码、执行、访存和写回过程。该模拟器支持基本的指令集、寄存器操作和内存访问，整个设计遵循计算机体系结构的基本原理，为深入理解CPU内部的工作机制提供了实践基础。

二、前端的设计

本项目旨在构建一个用于展示 Y86-64 CPU 仿真器状态的前端页面。该页面能够读取 YAML 文件中的数据，并以可视化的方式展示 CPU 的各种状态信息，包括寄存器（REG）、条件码（CC）、内存（MEM）和状态（STAT）等。用户可以通过切换按钮查看不同的运行段落，并选择不同的文件路径来加载数据。

文件结构

项目主要由以下三个文件组成：

1. `index.html`：定义了网页的结构和布局，包括标题、按钮、显示区域等。
2. `style.css`：提供了页面的样式和布局设计，使页面更加美观和用户友好。
3. `render.js`：包含了主要的交互逻辑，包括数据的加载、解析，以及页面内容的更新。

---

index.html 详细说明

该文件定义了网页的基本结构，主要包括以下部分：

1. 页面的头部信息

- **语言和字符集设置**：指定了页面的语言为英语，字符集为 UTF-8，确保了页面的兼容性。
- **标题**：页面的标题设置为 "Y86-64 CPU"。
- **样式表链接**：引入了 style.css 文件，用于定义页面的样式。

2. 页面主体

- **名称显示**：右上角显示了开发者的名字 "MHchen"。
- **页面标题**：使用 `<h1>` 标签显示了页面的主标题。
- 按钮区域
  :
  - **输入框**：用于输入文件名。
  - **操作按钮**：`Operate`、`Prev`、`Next`、`Cache`、`Memory`，提供了用户交互的功能。

3. 显示区域

- **PC 显示区域**：展示当前的程序计数器（PC）值和段落信息。
- 数据展示栏
  ：分为四个主要部分：
  - **REG**：显示寄存器内容。
  - **CC**：显示条件码状态。
  - **MEM**：显示内存内容。
  - **STAT**：显示当前状态。

4. 时间和文件路径显示

- **Time**：显示当前的时间值。
- **Filepath**：显示当前加载的文件路径。

5. 脚本引入

- **js-yaml 库**：用于解析 YAML 文件。
- `render.js` **脚本**：包含主要的交互逻辑。

---

style.css 详细说明

该文件定义了页面的样式，使页面具有良好的视觉效果和用户体验。

1. 全局样式

- **背景颜色**：设置了整体的背景颜色为淡棕色（#cfc2c2）。
- **字体**：统一使用了 Times New Roman 字体。
- **标题样式**：调整了主标题的大小和颜色。
- **名称显示**：定位在页面右上角，显示开发者的名字。

2. 按钮和输入框样式

- **按钮样式**：设定了按钮的大小、颜色、字体和交互效果。
- **输入框样式**：调整了输入框的尺寸、边框和背景颜色，使其与整体风格统一。

3. 显示区域样式

- **PC 显示区域**：设置了布局方式，使 PC 值和段落信息以合理的方式排列。

4. 数据展示栏样式

- **整体布局**：使用 Flex 布局，使数据展示栏的四个板块（REG、CC、MEM、STAT）并列排列。
- **样式细节**：设定了每个板块的背景、边框、字体和滚动条等属性。

5. 时间和文件路径显示区域

- **显示设置**：定位 Time 和 Filepath 信息的位置和样式。
- **隐藏文件路径**：通过 CSS，将文件路径相关的元素隐藏，只显示时间信息。

---

render.js 详细说明

该文件包含了页面的主要交互逻辑，包括数据的加载、解析和展示。

1. 变量和元素获取

- **输入框元素**：用于获取用户输入的文件名。
- 全局变量
  :
  - timeValue：存储当前的时间值。

    - currentIndex：当前段落的索引。
    - paragraphs：存储解析后的段落数据数组。

2. 格式化数据函数

- **功能**：将对象的数据格式化为可读的多行字符串，用于在页面上显示。

3. 元素获取

- **功能**：获取页面上各个需要操作的元素，方便后续的内容更新和事件绑定。

4. 按钮事件处理

- **按钮状态设置**：默认将运行按钮设置为可点击状态。
- 文件路径切换
  :
    - 点击 `Cache` 按钮后，文件的读取路径设置为 y86_cache。
    - 点击 `Memory` 按钮后，文件的读取路径设置为 y86。

5. 更新显示内容函数

- **功能**：根据当前的数据更新页面上的显示内容。
- **按钮状态**：根据当前索引，调整 `Prev` 和 `Next` 按钮的可用状态。

6. 加载 YAML 文件并解析

- **功能**：异步加载指定的 YAML 文件，并解析数据。
- 数据处理
  :
    - 过滤有效的段落数据。
    - 提取时间信息。
- **错误处理**：如果加载或解析失败，向用户提示错误信息。

7. 按钮事件绑定

- `Operate` **按钮**：当用户点击后，读取输入的文件名，加载并展示数据。
- `Prev` **和** `Next` **按钮**：用于在不同的段落之间导航。

8. 页面初始化

- **功能**：在页面加载时，初始化所有的显示内容和按钮状态。

本项目通过 HTML、CSS 和 JavaScript 实现了一个简洁直观的前端页面，用于展示 Y86-64 CPU 仿真器的运行状态。页面提供了良好的用户体验，用户可以方便地输入文件名、切换数据路径，并查看不同的 CPU 状态信息。代码结构清晰，模块化程度高，易于维护和扩展。

通过本次设计，我学习了前端开发的基础知识，对 HTML 的结构设计、CSS 的样式调整以及 JavaScript 的交互逻辑有了更深入的理解。同时，使用第三方库（如 `js-yaml`）进行数据解析，也提高了项目的实用性和功能性。

三、代码运行的方法

1. **编译程序**

   在终端中进入代码所在的目录，执行以下命令：

   ```
   make
   ```

   该命令会根据Makefile文件编译源代码，生成可执行文件cpu。

2. **运行CPU模拟器**

   使用以下命令运行CPU模拟器，将`.yo`指令文件作为输入：

   ```
   ./cpu < [指令文件路径] > output/[输出文件名].yml
   ```

   示例：

   ```
   ./cpu < testcases/prog1.yo > output/prog1.yml
   ```

   以上命令会将`prog1.yo`中的指令加载到CPU模拟器中执行，并将结果输出到`output/prog1.yml`文件中。

四、Cache模块设计

在本CPU模拟器中，缓存系统是一个关键组件，我实现了**指令缓存（Instruction Cache 和 数据缓存（Data Cache）**，以提高指令和数据访问的效率。下面将详细介绍缓存的设计和实现原理。

一、缓存概述

1. 缓存类型

- **指令缓存（I-Cache）**：用于缓存指令，减少取指阶段对主存的访问次数。
- **数据缓存（D-Cache）**：用于缓存数据，减少读写操作对主存的访问。

2. 缓存结构

- **组相联缓存（Set Associative Cache）**：采用组相联方式，兼顾直接映射和全相联缓存的优点，提高缓存的命中率和访问速度。

二、指令缓存的设计

1. 参数设置

- **缓存大小**：共有 **8 个集合（Set）**。
- **相联度**：每个集合包含 **2 条缓存线（Line）**。
- **块大小**：每条缓存线的块大小为 **2 字节**。

2. 缓存控制变量

- **有效位（valid）**：表示缓存线是否存储了有效的数据。
- **标签（tag）**：存储地址的高位部分，用于匹配访问地址。
- **数据块（data）**：存储实际的指令数据。
- **年龄（age）**：用于实现 LRU（最近最少使用）替换策略。
- **3. 工作原理**
  - 地址解析

    ：

    将指令地址划分为标签（tag）、索引（set index）和 块内偏移（block offset）。
    - **标签（tag）**：地址的高位部分，用于缓存线匹配。
    - **索引（set index）**：决定该地址映射到哪个集合。
    - **块内偏移（block offset）**：定位块内具体的数据字节。
  - 访问流程

    ：
    1. **计算索引和标签**：从指令地址中提取索引和标签。
    2. **定位集合**：根据索引找到对应的集合。
    3. 缓存命中判断

       ：在集合内的缓存线上，检查有效位和标签是否匹配。
       - **命中（Hit）**：直接从缓存提取指令数据。
       - **未命中（Miss）**：从主存加载指令数据到缓存，再进行访问。
    4. **更新年龄（age）**：在命中或未命中后，更新缓存线的年龄信息。
  - **替换策略**：采用 **LRU（Least Recently Used）** 算法。每次访问后更新缓存线的年龄，替换时选择年龄最大的（最久未使用的）缓存线。

4. 实现细节

- **数据结构**：使用二维数组表示缓存：

```
struct CacheLine {
    bool valid;
    uint64_t tag;
    uint8_t data[BLOCK_SIZE];
    uint64_t age;
};
CacheLine icache[NUM_SETS][LINES_PER_SET];
```

- **取指阶段逻辑**：
  - **计算索引和标签**：

```
uint64_t address = PC; // 当前指令地址
uint64_t block = address % BLOCK_SIZE;
uint64_t set_index = (address / BLOCK_SIZE) % NUM_SETS;
uint64_t tag = address / (NUM_SETS * BLOCK_SIZE);
```

- **遍历集合中的缓存线，判断命中**：

```
bool hit = false;
for (int i = 0; i < LINES_PER_SET; ++i) {
    if (icache[set_index][i].valid && icache[set_index][i].tag ==
tag) {
        // 命中，读取数据
        instruction = icache[set_index][i].data[block];
        // 更新年龄
        renew_age(set_index, i);
        hit = true;
        TIME += HIT_TIME; // 命中耗时
        break;
    }
}
```

- **未命中处理**：

```
if (!hit) {
    // 从主存加载指令到缓存
    int replace_line = find_LRU_line(set_index);
    icache[set_index][replace_line].valid = true;
    icache[set_index][replace_line].tag = tag;
    // 假设块大小为1字节，这里简化处理
    icache[set_index][replace_line].data[block] = Memory[address];
    // 更新年龄
    renew_age(set_index, replace_line);
    // 读取指令
    instruction = icache[set_index][replace_line].data[block];
    TIME += MISS_TIME; // 未命中耗时
}
```

- **年龄更新函数**：

```
void renew_age(uint64_t set_index, int line) {
    icache[set_index][line].age = current_time++;
}
```

- **LRU 替换函数**：

```
int find_LRU_line(uint64_t set_index) {
    uint64_t min_age = UINT64_MAX;
    int lru_line = 0;
    for (int i = 0; i < LINES_PER_SET; ++i) {
        if (icache[set_index][i].age < min_age) {
            min_age = icache[set_index][i].age;
            lru_line = i;
        }
    }
    return lru_line;
}
```

三、数据缓存的设计

1. 参数设置

- **缓存大小**：共有 **64 个集合（Set）**。
- **相联度**：每个集合包含 **8 条缓存线（Line）**。
- **块大小**：每条缓存线的块大小为 **32 字节**。

2. 缓存控制变量

- **有效位（valid）**
- **脏位（dirty）**：表示缓存线的数据是否已修改但未写回主存。
- **标签（tag）**
- **数据块（data）**
- **年龄（age）**

3. 工作原理

- **地址解析**：将数据地址划分为 **标签（tag）**、**索引（set index）** 和 **块内偏移（block offset）**。
- **访问流程**：
    1. **计算索引和标签。**
    2. **定位集合。**
    3. **缓存命中判断**：
        - **命中**：直接进行读/写操作。
        - **未命中**：
            - **读操作**：从主存加载数据到缓存，再读取。
            - **写操作**：采用 **写分配（Write Allocate）** 策略，先加载再写入。
    4. **更新年龄（age）**。
- **替换策略**：同样采用 **LRU** 算法。

4. 写回策略

- **写回（Write-Back）**：在对缓存线进行写操作时，仅修改缓存中的数据并设置脏位为 `true`，不立即写回主存。
- **缓存替换时**：

- 如果被替换的缓存线脏位为 `true`，则需要将数据写回主存。
- 如果脏位为 `false`，直接替换即可。

5. 实现细节

- **数据结构**：

```
struct CacheLine {
    bool valid;
    bool dirty;
    uint64_t tag;
    uint8_t data[BLOCK_SIZE];
    uint64_t age;
};
CacheLine dcache[NUM_SETS][LINES_PER_SET];
```

- **读操作流程**：

```
uint64_t address; // 要读取的数据地址
uint64_t block_offset = address % BLOCK_SIZE;
uint64_t set_index = (address / BLOCK_SIZE) % NUM_SETS;
uint64_t tag = address / (NUM_SETS * BLOCK_SIZE);

bool hit = false;
for (int i = 0; i < LINES_PER_SET; ++i) {
    if (dcache[set_index][i].valid && dcache[set_index][i].tag == tag) {
        // 命中，读取数据
        data = dcache[set_index][i].data[block_offset];
        renew_age(set_index, i);
        hit = true;
        TIME += HIT_TIME; // 命中耗时
        break;
    }
}

if (!hit) {
    // 未命中，从主存加载
    int replace_line = find_LRU_line(set_index);
    // 如果脏位为true，写回主存
    if (dcache[set_index][replace_line].dirty) {
        write_back_to_memory(set_index, replace_line);
    }
    // 加载新数据
    dcache[set_index][replace_line].valid = true;
    dcache[set_index][replace_line].dirty = false;
    dcache[set_index][replace_line].tag = tag;
    load_block_from_memory(address, dcache[set_index]
[replace_line].data);
    renew_age(set_index, replace_line);
    // 读取数据
    data = dcache[set_index][replace_line].data[block_offset];
    TIME += MISS_TIME; // 未命中耗时
}
```

- **写操作流程**：

```
// 与读操作类似，但需要设置脏位
if (hit) {
    // 命中，写入数据
    dcache[set_index][i].data[block_offset] = data;
    dcache[set_index][i].dirty = true;
    renew_age(set_index, i);
    TIME += HIT_TIME;
} else {
    // 未命中，写分配
    // 同读操作的未命中处理，加载后写入
    // ...
}
```

四、缓存替换策略的实现

- **LRU算法**：通过维护每条缓存线的年龄（age），在每次访问后更新，将当前时间赋给被访问的缓存线。需要替换时，选择年龄最小的缓存线。

五、缓存性能的模拟

- **时间消耗模拟**：
  - **缓存命中（Hit）**：
    - 指令缓存命中：TIME += 10;
    - 数据缓存命中：TIME += 10;
  - **缓存未命中（Miss）**：
    - 指令缓存未命中：TIME += 100;
    - 数据缓存未命中：TIME += 100;（还需考虑写回时间）
- **写回时间**：如果需要将脏缓存线写回主存，需要额外的时间模拟写回操作。

六、缓存设计的意义

- **提高性能**：通过缓存减少对主存的访问次数，大幅提升CPU的整体性能。
- **模拟真实情况**：缓存机制是现代CPU的重要组成部分，模拟缓存能够更真实地反映程序执行的效率和瓶颈。
- **优化空间**：通过调整缓存参数（如大小、相联度、块大小）和替换策略，可以观察对性能的影响，为进一步优化提供依据。

通过上述设计，实现了一个功能完整的缓存系统，提高了CPU模拟器的性能和真实度。缓存的引入使得指令和数据的访问更加高效，为模拟更复杂的系统奠定了基础。

五、意见&建议

最后一次lab啦，或者说是第一次&最后一次PJ。感觉基础功能的实现难度不大，但是创新的难度蛮大的。手搓测试文件也还是太艰难了。

ICS也是结课了。回想这一个学期，每个lab都做得磕磕盼盼，但是最后都还是能成功做出来。感觉就像我在pre最后说的那样，感觉什么都不会，但又感觉什么都会了。处于一个量子叠加态吧。

虽然ICS硬控了我一整个学期，但它还是教会了我很多东西的。成功把一个什么都不会的转专业小白带上了学习计算机的第一步。（感觉是因为要写lab我才真正弄懂了数据结构和c语言）

希望明年我也能来当助教，带坏下一届同学（bushi。

（要是当上助教，还是会尽心尽力尽职尽责的。金老师看看我~。