ICSpj项目报告-Y86模拟器

21307130371 丁珈瑶 & 21307130383 杨晨晨

阶段1：实现CPU

模拟器输入输出

- 输入：以包含了机器码和汇编码的 `.yo` 文件作为模拟器输入
- 输出：按照相应文件格式，编写代码输出`yaml`格式文件

顺序实现

测试方式与结果

- 运行`make`：

```
g++ -g -std=c++17 -Wall  -c cpu.cpp -o build/cpu.o
g++ -g -std=c++17 -Wall  -c tool.cpp -o build/tool.o
```

- 运行`python3 test.py --bin ./main`：

```
All correct!
```

```
iceeee@LAPTOP-VSTG409B:/mnt/c/Users/icee/Desktop/CPU_BACK$
python3 test.py --bin ./main
All correct!
```

程序封装

- Y86.h：结构体（CPU类）、宏定义、指令集实现相关函数声明
  - **程序员可见状态**
  - 15个程序寄存器：`long long int REG[15]`
  - 条件码：`int ZF, SF, OG`
  - 状态码 `int Stat`：
  - ```
    #define ADR 3 //遇到非法地址
    #define HLT 2 //遇到器执行halt指令
    #define AOK 1 //正常操作
    ```
  - 程序计数器相关（PC，valP）：`int PC, PCnxt, PCcall`
  - 内存从概念上即为一个很大的字节数组，我们小组通过char数组（一个字节对应两个字符）实现连续内存：`char MEM[10000]`
    ```

    - **Y86-64指令**
    - 指令编码的第一个字节：
        - 指令代码（icode）`int cod`：存储指令第一个字节高4位，判断指令类型
        - 指令功能（ifun）`int B1fn`：存储指令第一个字节低4位，在一组相关指令共用一个代码时判断具体指令
    - 指令存储：`vector<string> instr;`
- tool.h：工具函数声明
- tool.cpp：工具函数
    - char2Int：ascii码转int
    - Int2char：int转ascii码
    - getPos：获取指令地址
    - getval：译码函数，string -> int；并考虑小端法存储方式与负值情况
    - lavteg：反译码函数，int -> string
    - fetch：取指，根据`icode`获取指令字节数，用以计算`valP`
- instr.cpp：指令集相关函数（封装在CPU类中）
    - CPU构造函数：初始化条件码，PC，状态码
    - prep：初始化内存，读入指令后，将文件全部指令机器码存入内存

```
Stat = AOK; OF = SF = 0; ZF = 1;
memset(MEM, '0', 10000 * sizeof(char));
```

    - read：按行读取`.yo`文件中机器码，删除冗余空格，空行
    - printReg/printMem/printCC：按照`yaml`文件格式，打印寄存器（全部）、内存（非零）、条件码
    - halt：状态码`(Stat)`设置为`HLT (2)`，停止指令执行
    - rrmovq rA，rB：译码获得rA，rB，rB指明寄存器赋rA指明寄存器的值
    - irmovq V，rB：译码获得V，rB，rB指明寄存器赋V
    - rmmovq/mrmovq：译码获得rA，rB，D，执行得到内存操作的有效地址，在访存阶段将寄存器值valA写到内存，或者从内存中读出valM
    - OPq rA，rB：
        - 译码rA，rB，switch case选择具体指令：
        - addq/subq/andq/xorq
        - 进行相应操作条件码的维护
    - jXX Dest：
        - 译码Dest，switch case选择具体指令
        - jmp：无条件跳转，直接更新PCnxt
        - 有条件跳转，满足传送条件时更新PCnxt
        -

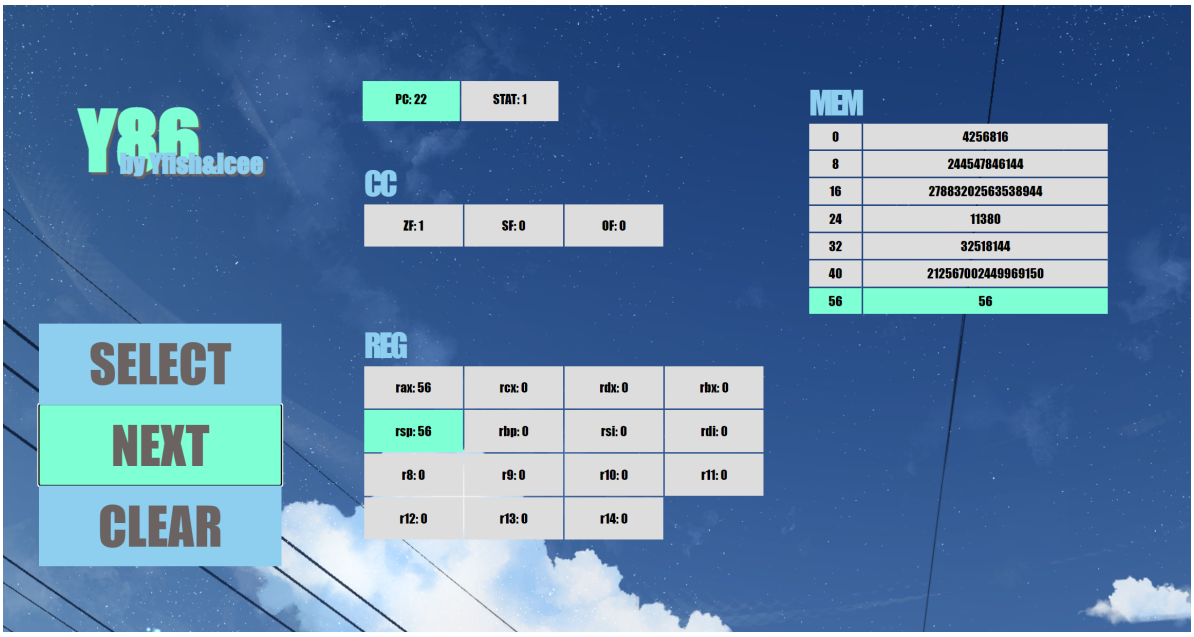
指令	传送条件
jle	`(SF ^ OF) |ZF`
jl	`SF ^ OF`
je	`ZF`
jne	`!ZF`

指令	传送条件
jge	`!(SF ^ OF)`
jg	`(!(SF ^ OF)) & (!ZF)`

  - cmovXX rA，rB：译码rA，rB，switch case选择具体指令，条件判断与jXX相同，满足条件时调用rrmovq
  - call Dest：译码Dest，更新栈顶（-8），访存将返回地址入栈，更新PCnxt，设为call调用的目的地
  - ret：访存得到返回地址，更新栈顶（+8），更新PCnxt，设为返回地址
  - pushq rA：译码fA，更新栈顶，判断地址是否有效（若内存地址为负更新状态码，结束循环），访存将rA指明寄存器的值写入栈
  - popq rA：译码fA，访存获取值，更新栈顶，并将数值写入rA指明寄存器
  - iaddq：译码V，rB，相加值存入rB指明寄存器，并更新条件码
- cpu.cpp：主函数
  - 调用read、prep函数，完成文件读入与相关初始化，处理器进入while无限循环进行指令实现，`switch case`语句跳转指令函数程序执行状态正常时不断更新PC为PCnxt，达成指令的跳转，每一步执行完打印寄存器、内存、条件码，直到状态码异常，执行break退出循环

阶段2：实现前端界面（web应用）

将后端运行得到的yaml文件用yaml2json.py批量转化为json文件。我将这些文件上传到github，再使用`fetch(URL)`与`response.json()`获取文件信息，再读取每一步的cpu状态信息并分块显示。



主要组件

- 三个table：reg、mem、cc、pc&stat。reg包含一行四个的表格内容，显示rax等等。
- 三个button：next、clear、select
- 全局变量pos指示现在位于json文件中的位置

主要功能

页面初始化

- `window.onload`，调用`init()`函数。`init()`调用一次`handleNext()`来显示第一步的cpu状态。
- 默认显示abs-asum-cmov.json文件

显示函数populate()

- 使用`fetch(URL)`与`response.json()`获取文件信息
- 在文件读取结束后再点击next，提示文件结束：判断pos是否已经等于文件的长度。

```
if (pos > file.length) {
    alert("运行结束！");
    return;
}
```

- 调用显示reg、mem、cc等状态的函数

状态更新并清除原先状态

- 对于REG和PC，只需直接为相应的textContent赋值
- 对于MEM
  - 使用`deleteRow()`循环删除原先的mem表格。进行table.rows.length - 1次循环（不能删除表格的标题（第一行，显示“MEM”）。
  - 使用`insertRow()`得到新的一行，再使用`insertCell()`在新的这一行中添加元素。

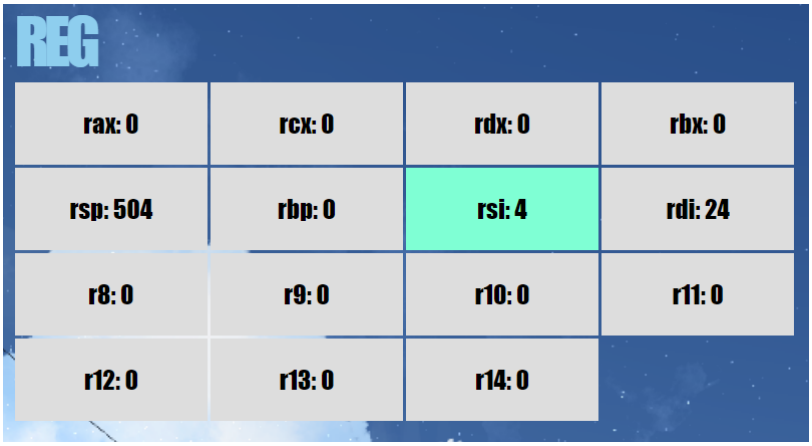
下拉菜单选择需要显示的文件

- hover 可以设置鼠标悬停时触发的操作，例如修改元素属性、调用函数等。下拉菜单悬停触发函数`pickFile()`
- `display: none;` 默认隐藏组件，设置为只有触发hover才显示。

- `pickFile()`：switch，选择不同的网址赋给requestURL，这将被显示函数调用。

高亮每一步的更新

对于REG和PC，只需比较当前元素的textContent与更新的值是否相等。



- 对于MEM，其中的元素个数和key是不固定的。实现方法如下：
    1. 对于每个当前mem中的key，查找前一个mem中是否有这个key，mem[key]值是否相等。更新的部分使用arr[]数组存储。

```
var arr = new Array();
var idx = 0;
//check for change and store changes in arr
if (pos > 0) {
    var pre = obj[pos - 1].MEM;
    var cur = obj[pos].MEM;
    for (let key in cur) {
        if (pre[key] == "undefined" || pre[key] != cur[key]) {
            arr[idx] = key;
            ++idx;
        }
    }
}
```

    2. 显示当前mem时，查找每个key是否在arr[]中。如果查找到，就将这一格的颜色高亮，否则设为普通的颜色。