**CSAPP Y86-64 pipeline simulator**

**stage 2（polished）+ stage 3**

**1.stage2:（polished）：**

* 1. 功能增加介绍：

1. 汇编器：将汇编码(.ys文件)转化为机器码(.yo文件)

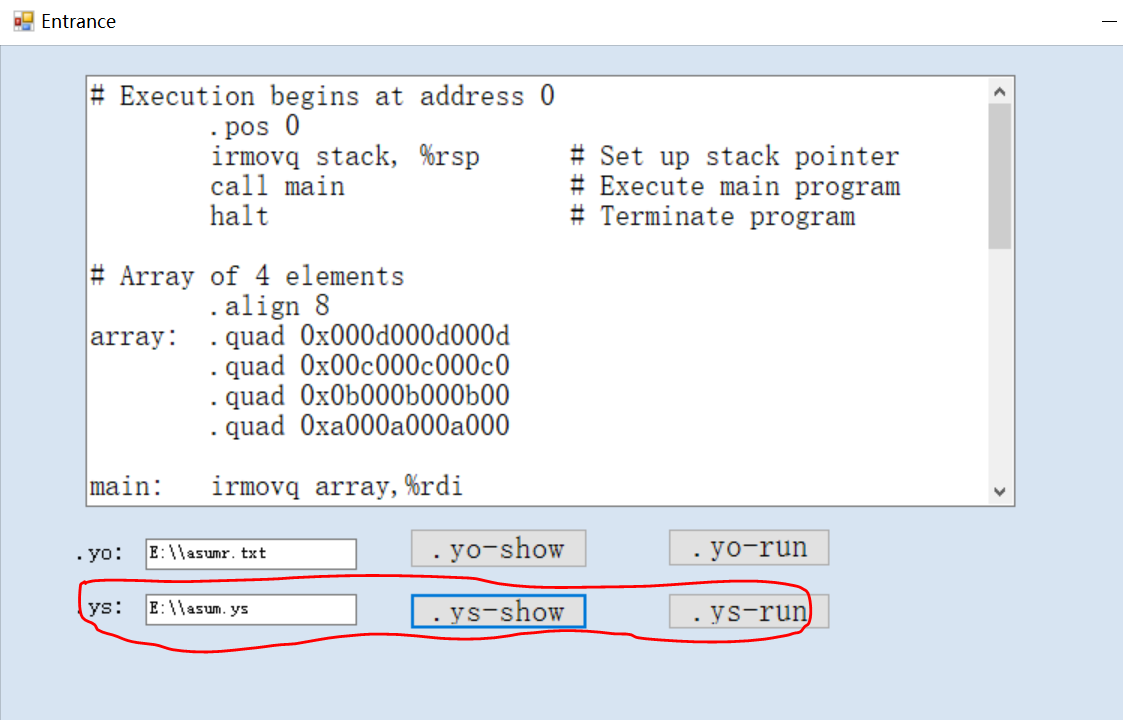
（2） 单步调试(step)功能

1.2 具体介绍：

（1） 汇编器：

1. 界面：

在原来的基础上增加了一个textbox用来输入ys文件名和两个button与yo文件，点击“.yo-show”按钮查看文件内容，点击“.yo-run”按钮运行



2）实现相关：

ReadAssemCode.h:与读取一行汇编码有关的函数，如：

void getNextInstr(char \*save)：得到指令

void GetNextWord(uc buff[][20]);得到参数

int isImme(uc \*c, ul \*val);判断是否为立即数

……

WriteBin.h：与写入机器码有关的函数，如：

针对不同指令名字的处理函数（do\_jxx等）

void write\_imme(uc\* t);写入立即数

ul alignAddr(ul num);使地址对齐

……

Tag.h：单独处理汇编码中用标签代替名字的情况的函数，如：

void add(uc \* name, ul addr);把标签名和地址存入tag数组

void tab(uc \*name, ul pos);参数为标签名且未在tag数组中找到时，存入mark数组

……

3）细节：

1.函数句柄：不同的指令要读入的参数个数不一，参数种类不一，干脆直接对每个指令都写一个处理函数，声明一个函数指针，通过读到的指令名得到相应的函数句柄赋给函数指针。同时，对于jxx、cmov这类的指令的处理函数可以合并成一个。

2.标签：汇编码中有时会用标签代替地址，当行首读到的是标签而不是指令名时，把标签名和地址存入tags数组中以便之后查询；当参数读到的是标签时，首先在tags数组中找该标签名，找到就把相应地址写入，若未找到，就把该标签名和地址放入mark数组，先用0填充当汇编码全部读完时，再rewrite。

3.mrmov：写入寄存器编号时是相反顺序，先写‘r’代表的寄存器，再写‘m’中包含的寄存器。

4.写入立即数时小端法

4)感想：

通过把汇编码翻译成机器码，学到了直接读机器码时没有注意过的事情。align 8意味着地址8位对齐；比如.pos xx是从xx地址开始……写汇编的时候感觉没有之前做cpu的时候困难，大概是有后者打了一下基础，已经知道了小端法之类的细节。但是还是会遇到有很多小问题，比如我写入立即数的时候0补位对于奇数长度和偶数长度有差别，但是这种问题在前端因为不能输出调试根本不好debug，于是我另外开了一个控制台程序才一点一点调试好。

(2) 单步调试(step):

这个就比较简单，在”step”按钮的函数中进行一个时钟周期即可。

**2.stage 3：**

2.1实验内容：

（1）实现多线程的Y86-64模拟器。

（2）模拟存储器层次结构、虚拟内存等

2.2具体内容：

2.2.1多线程：

（1）原理：

在单周期doWork（）函数中把每个阶段的函数各开一个线程包装起来，然后每个线程都调用start()函数开始;并调用join()阻止主线程;

（2）细节&困难：

一开始感觉多线程听起来很高级，不知道要怎么实现。询问助教之后，得知就是用thread这个类来给FDEMW五个阶段各开一个线程。因为在stage2中我对于数据转发的处理是当被转发的数据更新完之后再对d\_valA,d\_valB以及f\_predPC更新，这样的做法对于实现流水线是可行的，但对于多线程来说，这样就不能将与该阶段相关的所有变量都包装到一个线程里面（比如d\_valA对于其他d\_xx变量的更新是滞后的）。对于这种线程对同一个变量的访问速度不一致的问题，我想到的是——对几个会被多个线程访问的数据设flag，当更新了就把flag设为true，对其他阶段变量有依赖性的变量我就用了一个while（！（flag1&&flag2……））来卡住，等待某些其他的线程走到某一步再继续。但是一开始我把flag设为true的语句放在了if里面，导致有的情况因为不会进入if导致flag还是false，结果线程卡住。

然后，托管c++是一门没什么人用的语言(捂脸),资料很少，并且开线程的方式和其他语言不一样，一开始怎么写语法都错误，最后在google上搜到正确方法。

2.2.2模拟存储器层次结构：

（1）实现：

1）高速缓存（Cache）：

struct Line：每一行的结构体，含有valid有效位，tag标记位，byte\_num字节数，\*bytes字符数组，times记录出现次数，time记录出现时间。

struct Set：每一组的结构体，含有\*lines“行”组

struct Cache：Cache结构体，含有set\_num组数，line\_num每一组行数，\*sets

2）虚拟内存：即上一阶段中的字符数组

3）hit or miss or replace：

分别提取出地址的set位、tag位和byte位，set作为索引在cache中找到对应的组，遍历该组，若存在标记位有效并且tag一致的行，则hit，通过byte位偏移获得字节，读取或写入；若找不到，则miss，再次遍历，查看是否有无效的行（即空行），若找到则从内存中读取该行到cache中，通过byte位偏移获得字节，读取或写入；若每一行都有效，则需要根据不同的替换策略replace某一行，行被替换时要重写会内存。

4）替换策略：

LFU（最不常使用的被替换）：对于未被访问到的、有效的、times大于被访问行的行times--，被访问的行times重置为line\_num,当要替换时，选择times=1的行替换。

LRU（最近最少使用）：对于未被访问到的、有效的行time++,被访问的行time重置为0，要替换使，选择time最大的行替换。

1. 细节与困难：

因为组数、行数可自行设置，其实是把直接映射、组相联、全相联的实现统一成三个结构体；加入了cache之后原来的read\_mem函数要修改成read\_cache,原来的读取内存函数用于从virtual memory读到cache中；一开始实现的时候只考虑了读，没考虑写和写回；一开始运行的时候老是“未经处理的异常”，debug好久，最后还是换成控制台看输出，才发现是在去set位数时先左移再右移导致左边填充1成了负数，所以在移位操作时还是都转化成unsigne类型比较好。

ps：因某些原因，没有录下关于cache的视频，详细代码请见：

1. vcache.h&&vcache.cpp文件：关于上述cache实现
2. mm.cpp文件：将原本直接从内存中读取数据的函数read\_mem转变成从cache中读取，增加了read函数实现cache从内存中读取；将原本直接写入内存的函数write\_mem转变成从写入cache，增加了write函数实现从cache写入内存（如下图）

