CSAPP Y86-64 pipeline simulator

1.实验介绍：

1.1 实验内容：编写一个 Y86-64 流水线 CPU 模拟器：

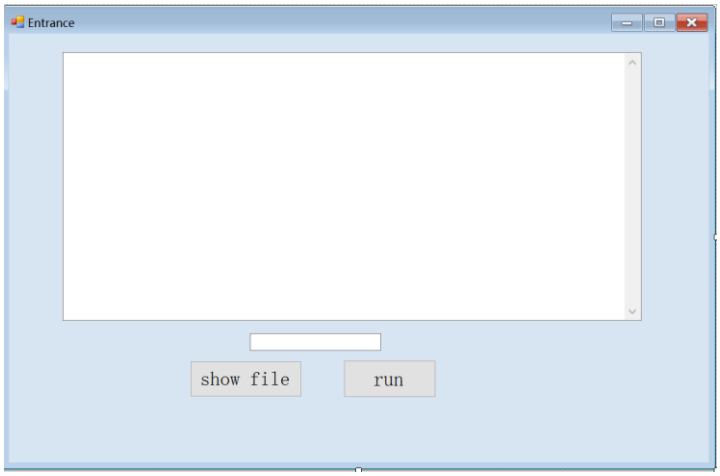
（1）正确实现正确模拟测试样例所需的最小指令集。

（2）提供基本的观察 CPU 运行状态与代码运行结果的功能。

1.2 模拟器介绍：

（1） 界面介绍：

1）初始界面：显示文件内容的“show file”按钮，产生下一界面的“run”按钮，一个输入文件路径的输入框，一个显示文件内容的输出框

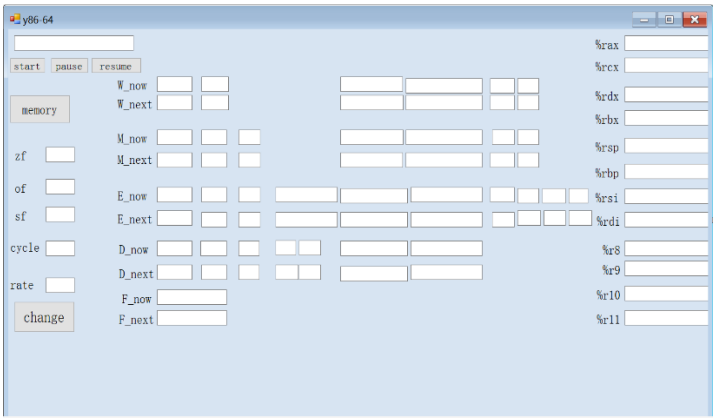


2）流水线界面：

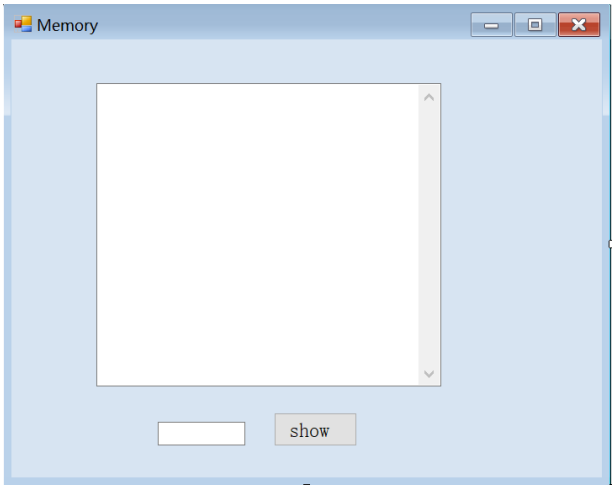
1/左部:一个输出框是文件路径；三个控制进程的按钮：“start”“pause”“resume”；一个“memory”按钮产生内存界面；三个输出框显示条件码;一个输出框显示时钟周期；一个用来输入速率的输入框，改变速率的“change”按钮

2/中部：输出框：FDEMF 各个阶段此时和下一时刻各个状态变量的值

3/右部：寄存器值输出框



3）内存界面：一个输入框，一个输出框，一个按钮



1.3 功能介绍：

（1） 显示每条指令经过流水线时，FDEMF 各个阶段此时和下一时刻各个状态变量的值、寄存器的值、条件码、当前时钟周期、内存的状态。

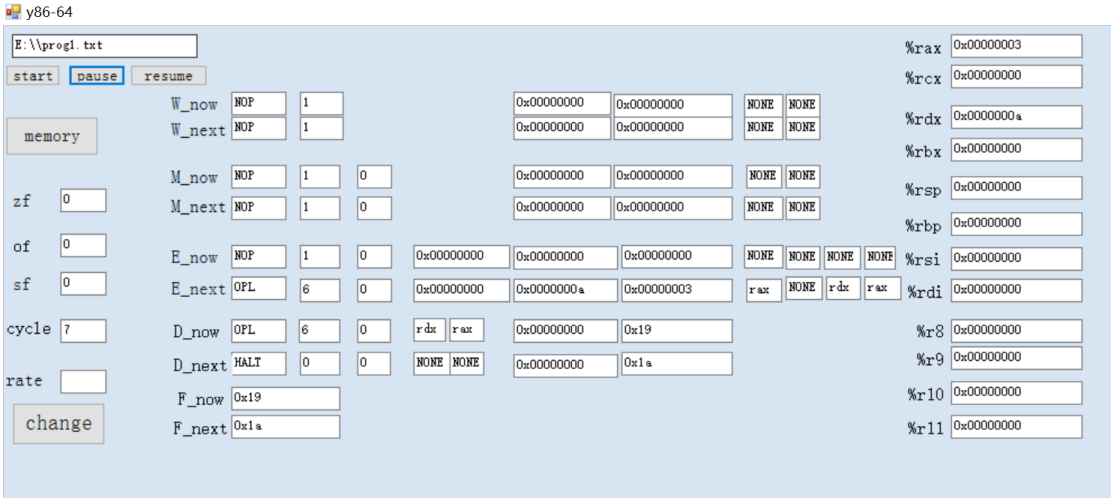
（2） 支持对时钟频率的修改。

（3） 支持 start、pause、resume 操作。

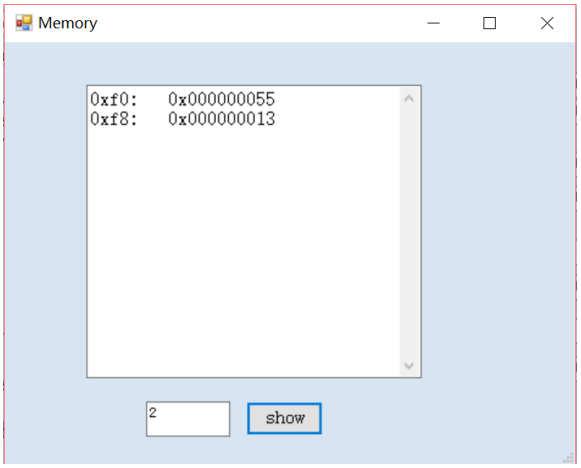
2. 使用手册：

1）输入文件完整路径（!!! 务必是完整路径），单击“show file”，上方的输出框会出现文件内容；单击“run”产生下一窗口

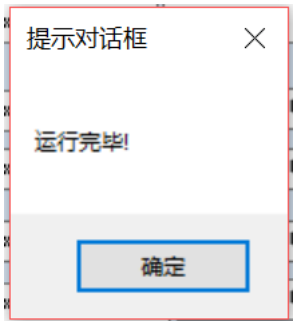
2）产生下一窗口后，单击“start”，各寄存器值会随着运行而变化。



3）单击“memory”，产生下一窗口，输入想要查看的栈上的值数目，单击“show”



4）当代码运行结束之后，会弹出对话框，显示运行完毕。



5）想要暂停的时候点击 pause 按钮即可暂停，点击 resume 即可继续。

3. 实现相关

Y86 流水线机制

3.1 准备工作：

1、在“mm.h”中开辟一个 0x200+5 的数组作为虚拟内存，指令从 0x0 开始存

2、在“mm.h”头文件中声明与读取内存相关的函数：

0）load_file：读取文件

1）atoi64_t：将字符数组转化为相应的long long型数据

2）transform：将16进制表示的字符转化成十进制数字

3）ctoc：把两个16进制字符合并成一个字符

4）addr：把字符数组表示的地址转化成long long

5）load_instr：保留文件中的机器码，并储存在开辟的虚拟内存中，每一个位置存储两个字符（8位）

6）read_mem：从虚拟内存中读取

7）read8Bytes：从虚拟内存中读取8位

8）write_mem：写入内存

3、在mm.cpp文件中实现上述函数

4、在“Fetch.h”“Decode.h”“Execute.h”“Memory.h”“Write.h”t头文件中定义各阶段处理时的中间变量以及初始化函数、NOP操作函数；在“INFO”头文件中把定义操作名称数组、寄存器名称数组；

5、新建“Entrance.h”作为初始界面，“pipe.h”作为流水线界面，“show_mem.h”作为内存显示界面；在头文件设计页面加入控件

3.2具体实现

1、用上一周期中间过程的值更新各阶段寄存器此刻的值

2、y86 机制：

1）y86 的过程执行是以周期为时间单位的，在每个周期中，Y86 流水线是按照 Fetch、Decode、Execute、Memory、Write Back 顺序执行。

2）Fetch 阶段，开始时会先判断状态是不是 HALT，如果是 NOP，反之则以 F_now_predPC 为始读取下一条指令。f_predPC 值由 PC 的预测值 f_valP/f_valC、寄存器 M 中的 M_now_valA、寄存器 W 中的 W_now_valM 共同决定。

3)Decode 阶段，开始时会先判断状态是不是 NOP/HALT，如果是则进行相应处理。反之，则根据 D 寄存器此刻的指令类型进行相应操作。

4)Excute 阶段，开始时会先判断状态是不是 NOP/HALT，如果是则进行相应处理。反之，根据 E 寄存器此刻的指令类型对 aluA 和 aluB 进行运算，以及设置条件码 CC。

5）Memory 阶段，开始时会先判断状态是不是 NOP/HALT，如果是则进行相应处理。反之，根据 M 寄存器此刻的指令类型进行读写，会产生 valM 值。因为此时 valM 已经更新，应当更新 f_predPC，d_valA,d_valB 实现数据转发。

6）Write 阶段，根据 W 寄存器此刻的指令类型写回寄存器

7）当中间变量都更新完之后，要进行异常判断，如果 stall=1 则下一时钟周期时 X_now 寄存器变量不更新，如果是 bubble=1，则插入“气泡”——把中间变量都设为无效值

3、用中间变量更新寄存器 X_next 的值

3.3 附加功能

1、速率控制、pause/resume：声明一个时钟对象，在时钟 tick 事件中完成一条指令，通过改变时钟的 interval 控制速率，通过控制时钟的 enable 实现 pause/resume

4.过程中的困难&感想：

1、指令存储到内存中：指令是 16 进制的形式，因为内存的变化应该是以 8 为单位，所以存储指令时应当把每两位 16 进制数合并成一位存在内存数组中。我的实现是把每两位转换成 10 进制数，存储到 char 类型的内存数组中，而在解析 icode、ifun、ra、rb 时因为 char 和 int 类似可直接移位分别取高四位和第四位。

2、内存写：把 long long 类型通过移位转化成 char[8]再写入。这两点都遇到了同一个问题，在移位时高位可能自动补 1，意识到之后我把所有 char 类型都改成了 unsigned char 类型。

3、数据类型转换：除了上述 char*和 long long 的相互转换，因为 textbox 类型是 String^,寄存器中间变量数据类型又是 string，涉及到 long long 和 string、String^(窗体中的 textbox 的 text 类型)的相互转换，所以设计了很多中间函数：比如 ll_to_str（ll num）：long long 是十进制，在转换时通过 stringstream s;s <<hex<< num;直接转换成 16 进制字符串

4、逻辑：一开始设计的时候，是把每个寄存器更新和处理阶段放在了一起，结果下一个寄存器的数据是上一个寄存器刚刚处理完的指令即每个寄存器处理的都是同一条指令，完全不是流水线，于是一通爆改，把寄存器的更新和处理分开。并且我一开始只有一个 mm.h 和 cpu.h 头文件，和别人交流之后，借鉴了把寄存器单独封装成一个类的做法，把中间变量、初始化函数和 NOP 函数放进去，实现起来思路清晰很多。在最初设计时退出标志是 fetch 到了“halt”指令，但这样会导致前面的几条指令都没有执行完，于是改成了以更新之后 W_now_instruc 的状态作为退出标志。

5、转发、异常：一开始直接翻译的 hcl 代码，果不其然出现了各种问题，比如分支错误时 f_predPC 没有更新，于是在访存之后增加了对 f_predPC 的更新和 d_valA、d_valB 的更新；在异常处理中，一开始我是分类讨论把 bubble 的判断放在相应的寄存器处理中，但是此时一些判断条件还没有更新，所以干脆把异常判断和处理都放在所有寄存器中间阶段之后。

6、窗体：第一次设计前端，很茫然，并且 windows form 的教程很少，最后还是在室友的帮助下上手。其实和 c++差不多，一个窗体就是一个类（只是此处为托管类），触发通过 button_click 事件，输出通过 textbox，产生新的窗口就实例化相应的类。

7、感想：一开始真的是一脸懵逼，没有写过这么庞大的一个 pj，所以一开始不知道如何利用头文件包装；逻辑上的不足导致很多问题想不清楚，没有想清楚就开始做导致经历多次“一通爆改”。虽然过程非常曲折，耗了很多时间，但是也加深了对于 Y86 的理解，学会了基本的前端开发。