

# ICS (Fall 2018) Project

## Phase 3

17307130015 王鑫涛

## 目录

实验内容.....	1
Phase1.....	1
Phase2.....	1
Phase3.....	1
环境.....	1
观察运行状态的方法.....	2
封面页（改动） .....	2
模拟器界面（改动） .....	3
操作和功能（新增） .....	4
控制台 .....	8
开发过程.....	8
多线程.....	8
存储器层次结构的模拟.....	11
对模拟器输入的优化 .....	13
模拟器自动运行速度的控制.....	13
观察模拟器运行到输入文件何处的窗口 .....	14
设置、取消断点 .....	14
模拟器界面的改善 .....	15
分离线程实现 I、M、P、C 窗口的并行.....	15
对展示录像的解释 .....	15

## 实验内容

### Phase 1

正确实现正确模拟测试样例所需的最小指令集。

通过了所给测试样例。

### Phase 2

通过使用 SFML 制作了较为易用简洁美观的图形界面。

### Phase 3

实现多线程的 Y86-64 模拟器。

实现模拟存储器层次结构，添加可视化 cache 窗口。

优化了模拟器的输入，不会再出现一次输入被多个周期读取或未被任何周期读取的情况。

增加了调整模拟器自动运行速度的按钮。

增加了观察模拟器运行到输入文件何处的窗口。

通过上述窗口内的输入实现了设置、取消断点的功能。

改善模拟器界面，使之更为美观、护眼。

使用分离线程将原有的内存窗口和指令窗口以及新添的 cache 窗口和输入文件查看窗口改为和模拟器主窗口并行运行。

## 环境

依赖于文件夹中的包含的 sfml 开头的 dll 文件。

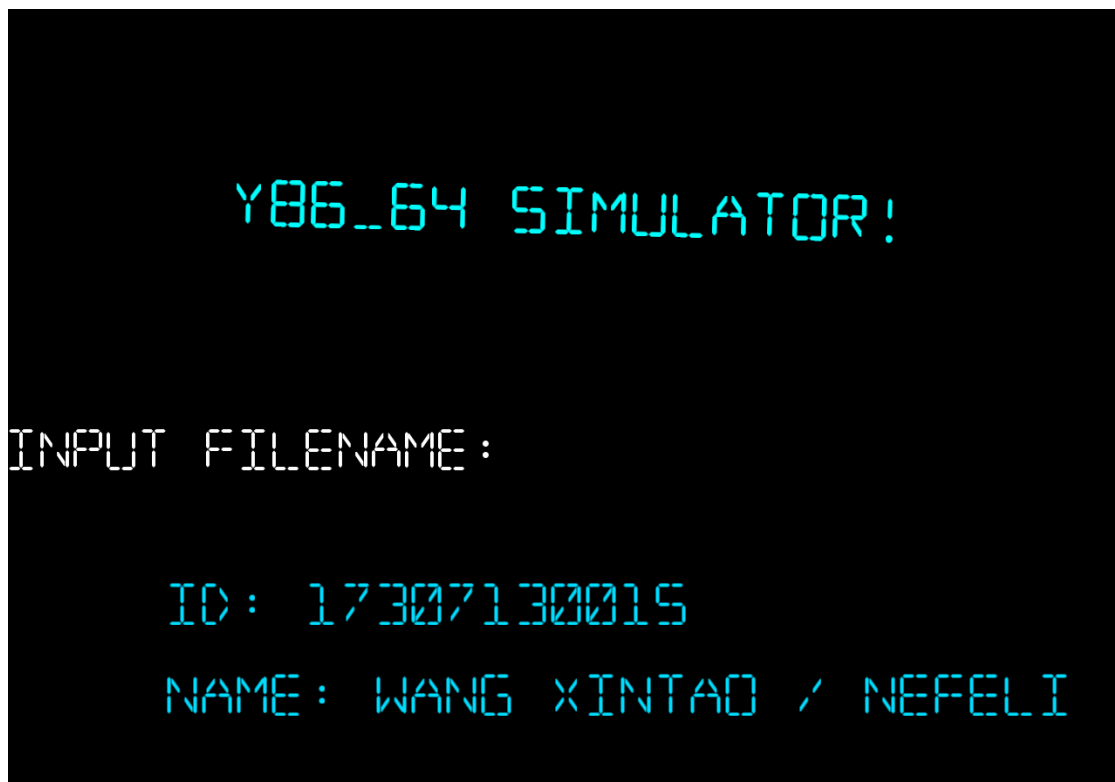
## 观察运行状态的方法

测试样例存储于 test\_code 文件夹中。

找到 SFML\_test 文件夹中的 SFML\_test.exe (或者/bin/Debug 下的同名文件)。

点开可以得到图示界面。

封面页 (改动)

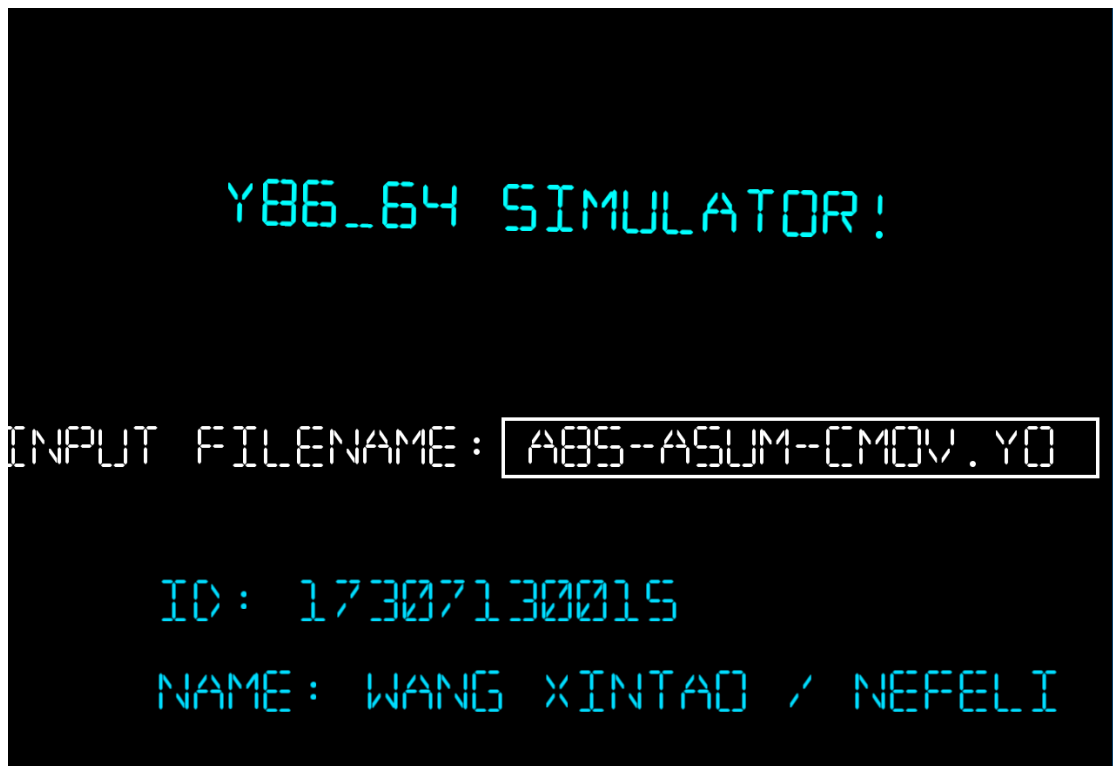


以及一个控制台。

```
press space and enter your filename
```

轻点空格进入输入模式输入文件名。

请输入不包含路径的文件名，如 abs-asum-cmov.yo。



可以输入 26 个小写英文字母、'-'、'.'和数字（可以使用小键盘）。

可以按 backspace 修改文件名，输入完后按空格键，在输入文件是正确的机器语言文件时会进入模拟器界面，否则清除输入。

### 模拟器界面（改动）

输出包含：

- 1.各个寄存器的值
2. ZF、SF、OF
- 3.F、D、E、M、W 各阶段的可见状态

```

RAX 00
RCX 00
RDX 00
R8X 00
RSP 00
RBP 00
RSI 00
RDI 00
R8 00
R9 00
R10 00
R11 00
R12 00
R13 00
R14 00

FOR AUTO-MODE PRESS M: OPEN MEMORY WINDOW
CYCLE CHANGE PRESS O: ENTER/QUIT AUTO MODE
PRESS:
Q: LIGHT-LIKE PRESS I: OPEN INSTRUCTION WINDOW
W: QUICK PRESS P: SEE INSTRUCTION POSITION
E: DEFAULT PRESS C: SEE CACHE SITUATION
R: SLOW

ZF 00 SF 00 OF 00 CF 00
F_PREDPC 00 F_STALL 00 F_BUBBLE 00
D_STAT 01 D_ICODE 01 D_IFUN 00 D_RA 15 D_RB 15
D_VALC 00 D_VALP 00 D_STALL 00 D_BUBBLE 00
E_STAT 01 E_ICODE 01 E_VALA 00
E_VALB 00 E_VALC 00 E_DSTE 15 E_DSTM 15
E_SRC A 15 E_SRC B 15 E_STALL 00 E_BUBBLE 00
M_STAT 01 M_ICODE 01 M_END 00
M_VALC 00 M_VALA 00
M_DSTE 15 M_DSTM 15 M_STALL 00 M_BUBBLE 00
W_STAT 01 W_ICODE 01 W_END 00
W_VALC 00 W_VALM 00
W_DSTE 15 W_DSTM 15 W_STALL 00 W_BUBBLE 00

```

按键盘上的 1-9 键中的数字键 i 将使流水线执行 i 个时钟周期。

可以看到各个阶段的状态和变量。

STALL 为 0 时亮白灯，为 1 时亮红灯。

BUBBLE 为 0 时亮白灯，为 1 时亮蓝灯，会发出气泡音。



(小圆放大的图形)

## 操作和功能

如橙字和粉字所示，提供以下功能：

1.按 M：打开记载程序运行过程中写入数据的 Memory 窗口，每页至多显示十个数据，可按左右键翻页。



2.按 O：进入或退出自动模式。

3.按 I：打开记载了输入文件的 Instruction 窗口，每页最多显示 10 条指令，可按左右键翻页。



4.按 P（新）：打开 Place 窗口，显示当前指令在输入 yo 文件中的位置。窗口页显示 7 条指令，当前 F\_predPC 对应的指令在中间，黄色标记。该窗口和主窗口并行运行，会随模拟器的运行而更改。

```

                                | ;# EXECUTION BEGINS AT ADDRESS 0
0x000:                          | ;# .POS 0
0x000: 30F400020000000000000000 | ;# IRMOVB STACK, %RSP ;# SET UP STACK POINTER
0x00A: 803800000000000000000000 | ;# CALL MAIN ;# EXECUTE MAIN PROGRAM
0x013: 00                          | ;# HALT ;# TERMINATE PROGRAM
                                | ;#
                                | ;# ARRAY OF 4 ELEMENTS

PRESS I FROM [1,2,3,4,5,6,7] TO SET BREAKPOINT IN THAT POSITION

```

5.在 P 窗口打开时，按小键盘上的 1-7 键后按空格（新）：在 Place 窗口中第 i 行（i 表示你按的键）对应的地址处设置或取消一个断点，用蓝色表示这个指令对应的地址处有一个断点。

```

0x038:                          | ;# MAIN:
0x038: 30F718000000000000000000 | ;# IRMOVB ARRAY,%RDI
0x042: 30F604000000000000000000 | ;# IRMOVB $4,%RSI
0x04C: 805600000000000000000000 | ;# CALL ABSUM ;# ABSUM[ARRAY, 4]
0x055: 90                          | ;# RET
                                | ;#

PRESS I FROM [1,2,3,4,5,6,7] TO SET BREAKPOINT IN THAT POSITION

```

```

In auto mode
breakpoint!
D_VALC 115
E_STAT 01
E_VALB 00
0x06E: 708000000000000000000000 | ;# JMP
0x077:                          | ;# /X $BEGIN
0x077: 50A700000000000000000000 | ;# LOOP:
                                | ;# MRMOVB

```

6.按 C（新）：打开 cache 窗口，其中有 4 个块，对应的 TAG、GRP 表示这一块在底层存储器（也就是我用 map 模拟的 memory）中的标记和索引，而 TIM 表示这一块 cache 有多久没被使用过（读或写），这里的单位时间并非一个时钟周期，而是一次对 cache 的访问。窗口页和主窗口并行运行，会随模拟器的运行而更改。





## 控制台

保留了阶段 1 中用于输出的控制台作为用于调试观察的额外功能。

```
30
rax 880481730765 rcx 0 rdx 0 rbx 0
rsp 496 rbp 0 rsi 2 rdi 40
r8 8 r9 1 r10 824646303936 r11 824646303936
r12 0 r13 0 r14 0
stat 1
ZF 1 SF 0 OF 0
F_predPC 0x87 F_STALL 0 F_BUBBLE 0
D_stat 1 D_icode 2 D_ifun 6 D_rA 11 D_rB 10 D_valC 0 D_valP 135 D_STALL 0 D_BUBBLE 0
E_stat 1 E_icode 6 E_valA 12094812457728 E_valB 0 E_valC 0 E_dstE 11 E_dstM 15 E_srcA 10 E_srcB 11 E_STALL 0 E_BUBBLE 0
M_stat 1 M_icode 6 M_Cnd 1 M_valE 0 M_valA 824646303936 M_dstE 11 M_dstM 15 M_STALL 0 M_BUBBLE 0
W_stat 1 W_icode 5 W_Cnd 0 W_valE 40 W_valM 12094812457728 W_dstE 15 W_dstM 10 W_STALL 0 W_BUBBLE 0
0x1f8 19
0x1f0 85
10
rax 12975294188493 rcx 0 rdx 0 rbx 0
rsp 496 rbp 0 rsi 1 rdi 48
r8 8 r9 1 r10 -175924544839680 r11 0
r12 0 r13 0 r14 0
stat 1
ZF 0 SF 0 OF 0
F_predPC 0x8b F_STALL 0 F_BUBBLE 0
D_stat 1 D_icode 6 D_ifun 0 D_rA 8 D_rB 7 D_valC 0 D_valP 139 D_STALL 0 D_BUBBLE 0
E_stat 1 E_icode 6 E_valA 175924544839680 E_valB 12975294188493 E_valC 0 E_dstE 0 E_dstM 15 E_srcA 10 E_srcB 0 E_STALL 0 E_BUBBLE 0
M_stat 1 M_icode 2 M_Cnd 1 M_valE 175924544839680 M_valA 175924544839680 M_dstE 10 M_dstM 15 M_STALL 0 M_BUBBLE 0
W_stat 1 W_icode 6 W_Cnd 1 W_valE 175924544839680 W_valM 0 W_dstE 11 W_dstM 15 W_STALL 0 W_BUBBLE 0
0x1f8 19
0x1f0 85
10
rax 188899839028173 rcx 0 rdx 0 rbx 0
rsp 504 rbp 0 rsi 0 rdi 56
r8 8 r9 1 r10 175924544839680 r11 175924544839680
r12 0 r13 0 r14 0
stat 1
ZF 1 SF 0 OF 0
F_predPC 0x60 F_STALL 1 F_BUBBLE 0
D_stat 5 D_icode 1 D_ifun 0 D_rA 15 D_rB 15 D_valC 0 D_valP 0 D_STALL 0 D_BUBBLE 1
E_stat 1 E_icode 9 E_valA 504 E_valB 504 E_valC 0 E_dstE 4 E_dstM 15 E_srcA 4 E_srcB 4 E_STALL 0 E_BUBBLE 0
M_stat 5 M_icode 1 M_Cnd 1 M_valE 0 M_valA 0 M_dstE 15 M_dstM 15 M_STALL 0 M_BUBBLE 0
W_stat 5 W_icode 1 W_Cnd 1 W_valE 0 W_valM 0 W_dstE 15 W_dstM 15 W_STALL 0 W_BUBBLE 0
0x1f8 19
0x1f0 85
10
rax 188899839028173 rcx 0 rdx 0 rbx 0
rsp 512 rbp 0 rsi 0 rdi 56
r8 8 r9 1 r10 175924544839680 r11 175924544839680
r12 0 r13 0 r14 0
stat 4
ZF 1 SF 0 OF 0
F_predPC 0x14 F_STALL 1 F_BUBBLE 0
D_stat 4 D_icode 0 D_ifun 0 D_rA 15 D_rB 15 D_valC 0 D_valP 20 D_STALL 0 D_BUBBLE 0
E_stat 4 E_icode 0 E_valA 0 E_valB 0 E_valC 0 E_dstE 15 E_dstM 15 E_srcA 15 E_srcB 15 E_STALL 0 E_BUBBLE 0
M_stat 5 M_icode 1 M_Cnd 0 M_valE 0 M_valA 0 M_dstE 15 M_dstM 15 M_STALL 0 M_BUBBLE 1
W_stat 4 W_icode 0 W_Cnd 1 W_valE 0 W_valM 0 W_dstE 15 W_dstM 15 W_STALL 1 W_BUBBLE 0
0x1f8 19
0x1f0 85
```

## 开发过程

### 多线程：

参照课本上线程的使用方法，将 F、D、E、M、W 五个阶段的执行放到五个不同的线程当中执行。

为了处理转发问题，实际上 W、M、E 阶段需要先行执行，不过即使是伪并行也希望做成并行的样子，于是参考书上 P702 页中信号量的使用方法，设置了 4 个信号：W\_instr, M\_instr, E\_instr\_D, E\_instr\_F，实现 W、M 并行，E 执行，再 D、F 并行，但这样其实是分为了三个部分，所以改为 6 个信号，W\_instr, W\_instr1, W\_instr2 和 M\_instr, M\_instr1, M\_instr2，实现 W、M 先并行，E、D、F 再并行。

```
void * y86_64::W_thread(void *__this)
{
    //y86_64 * _this = (y86_64 *) __this;
    P.Writeback();
    sem_post(&W_instr);
    sem_post(&W_instr1);
    sem_post(&W_instr2);
    //cout<<"W_"<<W_instr<<endl;
}

void * y86_64::M_thread(void *__this)
{
    //y86_64 * _this = (y86_64 *) __this;
    P.Memory();
    sem_post(&M_instr);
    sem_post(&M_instr1);
    sem_post(&M_instr2);
}

void * y86_64::E_thread(void *__this)
{
    //y86_64 * _this = (y86_64 *) __this;
    sem_wait(&W_instr);
    sem_wait(&M_instr);
    P.Execute();
    //sem_post(&E_instr_D);
    //sem_post(&E_instr_F);
}

void * y86_64::D_thread(void *__this)
{
    //y86_64 * _this = (y86_64 *) __this;
    //sem_wait(&E_instr_D);
    sem_wait(&W_instr1);
    sem_wait(&M_instr1);
    P.Decode();
}
```

一开始能明显发现流水线的执行有问题，找不到问题，只好先设置成这样。

```
sem_init(&W_instr,0,0);
sem_init(&M_instr,0,0);
sem_init(&E_instr_D,0,0);
sem_init(&E_instr_F,0,0);

pthread_create(&W_pth,NULL,W_thread,NULL);
pthread_create(&M_pth,NULL,M_thread,NULL);
pthread_create(&E_pth,NULL,E_thread,NULL);

pthread_join(W_pth,NULL);
pthread_join(M_pth,NULL);
pthread_join(E_pth,NULL);

pthread_create(&D_pth,NULL,D_thread,NULL);
pthread_create(&F_pth,NULL,F_thread,NULL);
pthread_join(D_pth,NULL);
pthread_join(F_pth,NULL);
```

后来发现，各个 thread 函数中 sem\_post 的返回值竟然是-1，原来我的修改没有成功。通过面向搜索引擎的编程，我发现这可能是我没有使用 sem\_init 的原因。在使用 sem\_init 之后，问题就迎刃而解了。

```
sem_init(&W_instr,0,0);
sem_init(&M_instr,0,0);
sem_init(&W_instr1,0,0);
sem_init(&M_instr1,0,0);
sem_init(&W_instr2,0,0);
sem_init(&M_instr2,0,0);
//sem_init(&E_instr_D,0,0);
//sem_init(&E_instr_F,0,0);

pthread_create(&W_pth,NULL,W_thread,NULL);
pthread_create(&M_pth,NULL,M_thread,NULL);
pthread_create(&E_pth,NULL,E_thread,NULL);
pthread_create(&D_pth,NULL,D_thread,NULL);
pthread_create(&F_pth,NULL,F_thread,NULL);

pthread_join(D_pth,NULL);
pthread_join(F_pth,NULL);
pthread_join(W_pth,NULL);
pthread_join(M_pth,NULL);
pthread_join(E_pth,NULL);
```

## 存储器层次结构的模拟

先是确定了对地址位的划分：用后四位表示块偏移，从右数第五、第六位表示组索引，其余位表示标记。

将原来的内存 Memory 作为底层存储器，新建 Cache 类。

```
class cache
{
    public:

        cache();
        int dirty;
        int valid;
        int tag;
        int group;
        long long data[16]={};
        int time_since_lastuse;
};
```

其中 dirty 表示是否被写过，tag、group 表示当前块在底层存储器中的组索引和标记。Time 表示有多久没被使用过，这里的单位并非模拟器运行的周期，而是一次对 cache 的访问。

在模拟器中共有 4 块 cache，每块大小和底层存储器的模拟中一块的大小相同，有 16 个数据（位）。因为 memory 本身是用小端法存储，所以 cache 的实现就比较方便。

添加从底层存储器向 cache 中写入一个块、以及在 cache 中某块被写过的情况下将其写回底层存储器的操作。

```

void pipe::cache_read(int group, int tag , int icache)
{
    //将DRAM(memory)中的一块读入cache
    Cache[icache].valid = 1;
    Cache[icache].dirty = 0;
    Cache[icache].tag = tag;
    Cache[icache].group = group;
    Cache[icache].time_since_lastuse = 0;
    cout<<"group & tag " <<group << " " <<tag<<" " <<((group<<4)+(tag<<6))<<endl;
    for(int i=0;i < 16 ; i++)
    {
        Cache[icache].data[i] = mem[(tag<<6) + (group<<4) + i];
        //cout<<"icache i " <<icache<<" " <<i<<" " <<Cache[icache].data[i]<<endl;
    }
}

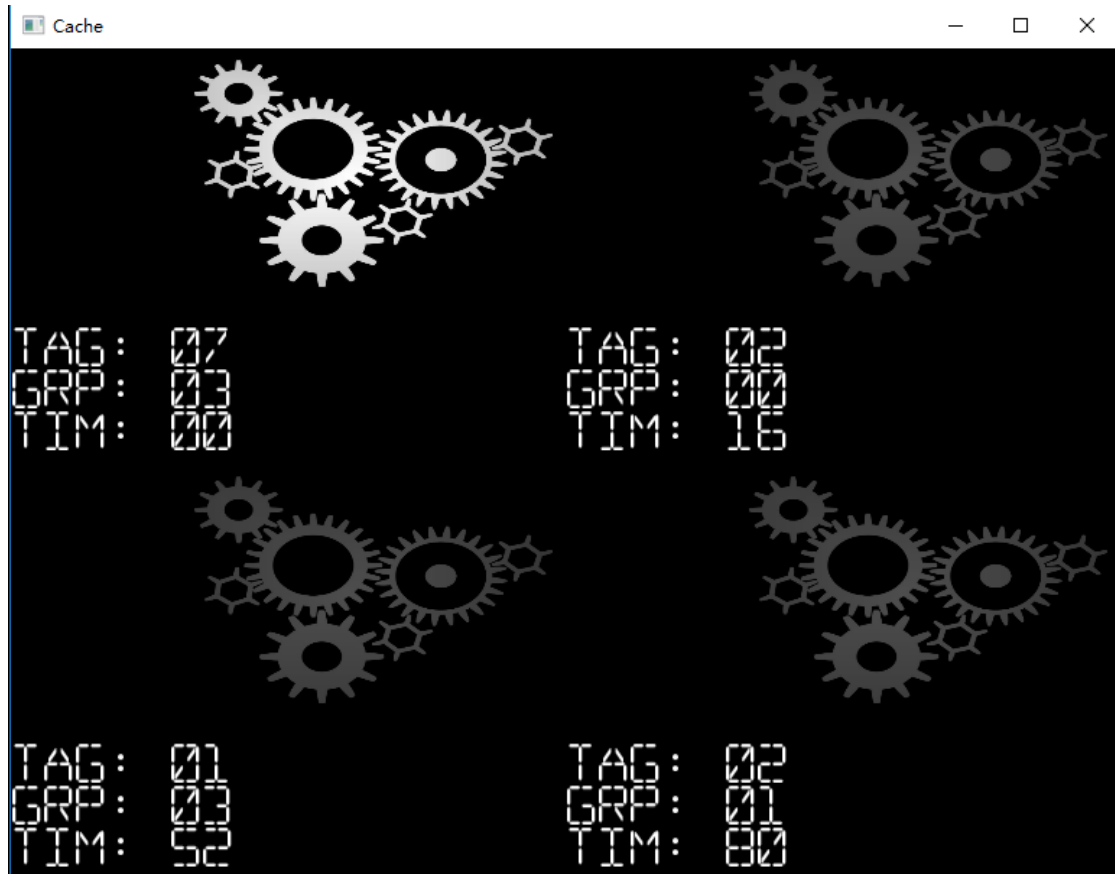
void pipe::cache_writeback(int group, int tag, int icache)
{
    if(Cache[icache].dirty == 1)
    {
        for(int i=0;i < 16 ; i++)
        {
            mem[(tag<<6) + (group<<4) + i] =Cache[icache].data[i];
        }
    }
}

```

每次要写数据时，先通过 tag 和 group 观察对应的那一块是否在 cache 中，在的话 (hit) 就将数据写入对应的 cache 块，并标记其被写过，更新 cache 块最远使用时间为 0。如果不在的话 (miss)，关于写的问题较为复杂，我采用一种简单的策略，直接写入模拟底层存储器的 mem 中。

读数据的情况类似，只是当 miss 的时候，我们先将对应的块读入 cache 中，替换掉最久未被使用的 cache 中的一块。

使用 SFML 创建可视化窗口。(前文中已经介绍过)



### 对模拟器的输入优化

设置整型数据 `wait`，当读取一次输入后将 `wait` 设置为一个整数值，每次进入 `handleInput`（输入处理）函数时，执行 `if(wait-- > 0) return;` 的操作，屏蔽掉这个输入处理函数，保证一次按键输入不会被多次读取。

### 模拟器自动运行速度的控制

在输入处理中新增对 Q、W、E、R 键的检测，根据输入将 `cycle` 变量更改为 1、2、5、50，`cycle` 的初始值是 5。在自动模式下，只有当 `clk%cycle` 的时候，才执行后台流水线的一个周期。

## 观察模拟器运行到输入文件何处的窗口

在读输入文件时，将每一行的字符串保存起来。当我们在模拟器界面按下 P 的时候，就生成一个新的窗口，通过当前后台流水线中 F\_predPC 的数据，找到前后一共七条指令字符串。

```
0x000:                                | # EXECUTION BEGINS AT ADDRESS 0
0x000: 30F40002000000000000          | .POS 0
0x000: 30F40002000000000000          | IRMOVD STACK, %RSP      # SET UP STACK POINTE
0x00A: 803B0000000000000000          | CALL MAIN               # EXECUTE MAIN PROGRAM
0x013: 00                                | HALT                    # TERMINATE PROGRAM
                                | # ARRAY OF 4 ELEMENTS
PRESS I FROM [1,2,3,4,5,6,7] TO SET BREAKPOINT IN THAT POSITION
```

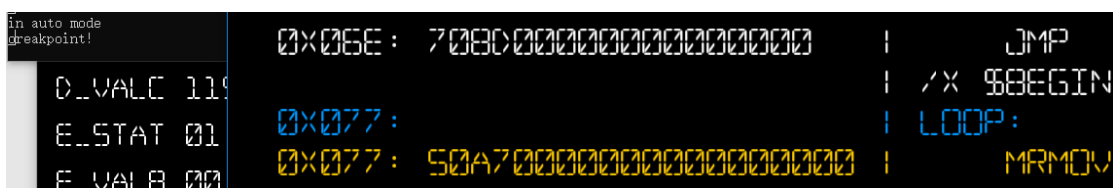
## 设置、取消断点

定义一个判断某个地址是否有断点的 `map<int,int>` 型标记。在 P 窗口打开时，检测小键盘上 1-7 键的输入和空格键，将 P 窗口中对应位置的指令字符串对应的地址标记为断点或取消标记。同时通过定义 `color[line]` 数组，可以将我们想标记断点的那一行的颜色设置为蓝色，表示这里是一个断点。

自动模式下，当 F\_predPC 为断点时就会停下来。

最初是不区分小键盘上的数字键和主键盘上那一排数字键的，但将 P 窗口改为并行线程之后，输入一个数字会同时被模拟器窗口和 P 窗口读取，这当然不是我们希望看到的，所以只好将小键盘分开。

本来设置断点也不需要再按一次空格，但后来考虑到还需要能够取消断点。由于线程分离，P 窗口的运行频率很高，因此用 `wait`—和卡循环来控制都不可行（添加一个过大的循环会造成未响应），所以设定一个小键盘上的数字输入+一次 Space 输入为设置或取消断点的输入。



## 模拟器界面的改善

通过 RGB 颜色值查询，尝试了许多不同的颜色，使界面更为美观。同时调整了一些文字的比例和位置，便于容纳关于新功能的 Tips。

## 分离线程实现 I、M、P、C 窗口的并行

这些窗口原本实现是在 handleInput 中检测输入后打开一个循环，这样的结果就是整个程序将卡在这个循环中，在窗口关闭前模拟器不会再往下运行。为了方便在流水线运行过程中持续观察 Cache 和 P 窗口，将这几个窗口改为新的线程，并将之与主线程分离，实现便利的持续观察功能。

## 对展示录像的解释

展示录像是有声音的，也可以打开声音听语音解释。

[00:00 – 00:36]: 总体介绍。

[00:37 – 00:56]: 封面调整和输入优化。

[00:57 – 01:15]: 模拟器界面调整及按键介绍

[01:15 – 01:44]: 观察运行到输入文件何处的 Place 窗口展示及按键介绍

[01:45 – 02:21]: cache 窗口的介绍和展示

[02:22 – 02:45]: 自动模式下速度调整功能展示



[02:46 – 03:04]: cache 窗口 dirty 标志展示

[03:05 – 04:04]: 断点设置、取消和使用展示

[04:05 – 04:23]: 闲谈