Y86-64 阶段三实验报告

姓名：罗旭川　　学号：17307130162　　实验时间：2019.1.11-2019.1.12

一、 实验目的

补充 Y86-64 流水线 CPU 模拟器，的功能，包括多线程和 cache

二、 实验代码实现具体过程

A、多线程的实现

1、分析

本 CPU 的多线程早已经在阶段一就实现了，主要是通过 C++的 thread 库和 mutex 库实现。多线程，简单来讲，就是能够实现多个函数的同时运行，为了能正确地使用 thread 库，在实现 Y86 的多线程之前，我先进行了一个小实验：

```cpp
#include<iostream>
#include<thread>
#include<mutex>
using namespace std;
mutex mt;
void fetch(int a)
{
//  lock_guard<mutex> lck(mt);
    cout<<"fetch"<<endl;
}
void decode(int a)
{
//  lock_guard<mutex> lck(mt);
    cout<<"decode"<<endl;
}
void execute(int a)
{
//  lock_guard<mutex> lck(mt);
    cout<<"execute"<<endl;
}
void memory(int a)
{
//  lock_guard<mutex> lck(mt);
    cout<<"memory"<<endl;
}
void write_back(int a)
{
//  lock_guard<mutex> lck(mt);
    cout<<"write_back"<<endl;
}
```

```cpp
int main()
{
    int i=0;
    while(i<10)
    {
        thread t1(fetch,1);
        thread t2(decode,2);
        thread t3(execute,3);
        thread t4(memory,4);
        thread t5(write_back,5);
        t1.join();
        t2.join();
        t3.join();
        t4.join();
        t5.join();
        cout<<"circle "<<i++<<" finish."<<endl;
        cout<<endl;
    }
}
```

上述代码是我写的一个 Y86 的多线程实现框架，当不加锁时会得到混乱的输出结果：

```
fetchexecute
decode
memory

write_back
circle 0 finish.

fetchwrite_backdecode

memory
execute

circle 1 finish.

fetchmemory
execute
decode

write_back
circle 2 finish.
```

如图所示，当使用多线程来执行Y86模拟器的5个stage时，五个阶段的执行顺序将无法保证，这是由于 5 个线程运行的快慢不同导致的。为了避免这种情况，我们需要用到锁。

给这 5 个函数对象加上互斥锁后，会使得每个函数之间不会相互冲突，只有当第一个 cout 执行完后，下一个才能开始 cout。

在这里，我用的是 C++11 方便的自解锁 lock_guard，代码就如左上图被注释掉的部分所示，即：

lock_guard<mutex> lck(mt);

这东西是干什么的呢？它是与 mutex 配合使用，把锁放到 lock_guard 中时，mutex 自动上锁，lock_guard 析构时，同时把 mutex 解锁。因此只需要在每个函数开头上锁就好了，函数调用结束后会**自动解锁**。

上述代码加完锁后，就能按照预想得顺序执行了：

```
fetch
decode
execute
memory
write_back
circle 0 finish.

fetch
decode
execute
memory
write_back
circle 1 finish.
```

注意到：我们回到 Y86-64 的实现上，由于每个 stage 执行完后，都会把值传到 pipe 寄存器中，然后还会进行一个值传递，比如 m_valA→M_valA，并且这个值传递的过程是在所有 5 个 stage 执行完当前语句后才执行的，因此实际上 5 个 stage 用多线程并行时并不会发生取值错误，而**加了锁后反而降低了这几个程序的并行性**。

但是在我的代码实现上，为了能保证 cout 的内容按照 Fetch、Decode、Execute、Memory、Write_back 的顺序打印出来，我才必须要加锁。

2、结果

```
thread t1(fetch, point_f);
thread t2(decode, point_d);
thread t3(execute, point_e);
thread t4(memory, point_m);
thread t5(write_back, point_w);
t1.join();
t2.join();
t3.join();
t4.join();
t5.join();
```

```
void fetch(int point);

void decode(int point);

void execute(int point);

void memory(int point);

void write_back(int point);
```

通过调用 C++的 thread 库，将 Fetch、Decode、Execute、Memory、Write back 五个阶段分别放在 5 个线程中，并用 5 个全局的指针 point_f、point_d、point_e、point_m、point_w 分别指示每个线程需要执行的语句，并将相应指针传入相应的 5 个 stage 函数中

B、cache 的实现

1、分析

Cache 机制，简单来讲，就是将 CPU 最近调用到的地址和值存放在一个 CPU 能快速调用的地方，然后每次需要访问地址时，都先在这个 cache 中寻找。若找到则称为 hit，直接取值即可；找不到则称为 miss，需要去内存访问，并且更新 cache 中的值，方便下次的快速取值。

按照书本的实现方法，需要将地址分为 tag、group_index、offset 三个部分，group_index 用于找到 cache 中对应的组，tag 为匹配标记，offset 则为高速缓存块，里面存储了一段区间的字节块。

但是，考虑到我的内存实现方式并没有按照字节存储，因此完全一摸一样地按照这样的

```
struct Cache{
    int avail=0; //该组cache中空的副本位置下标（<4）
    pair<long long, long long> copy[4];
};
extern Cache cache[4];
//由于内存储存形式的不同，这里我的cache没有按照书本的实现，而是通过4组cache来模拟，每组可放4个值
```

规则实现 cache，因此**我用了其他方法来实现对 cache 地模拟**：
我设置了一个组数为 4 的 cache，分别“管理”内存的 1/4，pair 型数组表示每组 cache 允许存放 4 个最近使用的地址及在该地址上对应的值，avail 用于指示每组中空余的位置

```
int get_group(long long locate)
{
    return (locate & 0x30)>>4;
}
```

如何判断一个地址应该分到哪个 cache 组呢？在这里，我简单的取了地址值的倒数 5、6 位来作为该地址的 group_index，并经过尝试发现能取到较好的效果，即每组数量平均，并且 hit 较多

在代码实现上，cache 主要影响的是内存的读写，因此我将源代码中 Memory 阶段中对内存的读写操作都归结为了两个函数如下，在函数中实现 cache 机制即可：

1）read_to_memory()函数

```
long long read_from_memory(long long locate)
{
    if (No_cache)
    {
        Sleep(100);
        return Memory[locate];
    }
    int index = get_group(locate);
//hit
    for (int i = cache[index].avail - 1; i >= 0;--i)
        if (cache[index].copy[i].first == locate) return cache[index].copy[i].second;
//miss
    Sleep(5);
    long long value = Memory[locate];//读内存
    if (cache[index].avail <= 3)//如果还有空位
    {
        cache[index].copy[cache[index].avail] = make_pair(locate, value);
        cache[index].avail++;
    }
    else
        cache[index].copy[0] = make_pair(locate, value);//覆盖第一个
    return value;
}
```

代码的主要框架为：

①根据地址找到该地址对应的 cache 组号。

②遍历相应 cache 中的四个储存位，若地址在 cache 中 hit，则直接读取 cache 中的内存副本。**注意到，这里遍历四个储存位时我用了倒序的遍历顺序**，因为放在后面的储存值是最近一次存放进来的值，因此其被访问到的机会更大一些，因此先判断它能更高效一些

③若 miss，则只能从内存中读取值，。

如果 cache 组中还有储存位，则将该地址和取到的内存值储存在 cache 中，方便下次读取

如果 cache 组已经存满了 4 个值，则**用新的地址和值覆盖第一个储存值**，因为第一个为最先储存进来的值，因此其被调用的可能性更小，所以可以将其从 cache 中踢除

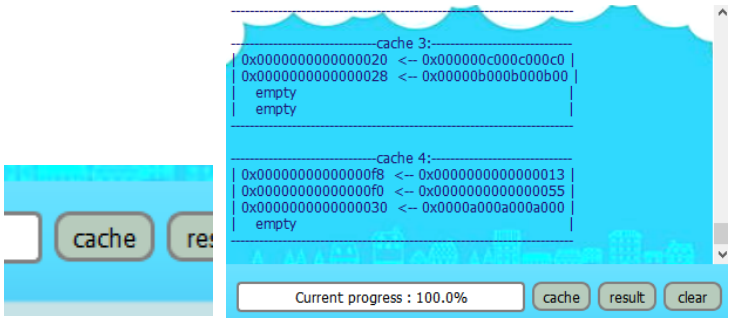
2）write_to_memory()函数【采用直写和非写分配方式】

```
void write_to_memory(long long locate, long long value)
{
    if (No_cache)
    {
        Sleep(100);
        Memory[locate] = value;
        return;
    }
    int i;
    int index = get_group(locate);
//hit
    for (int i = cache[index].avail - 1; i >= 0; --i)
    if (cache[index].copy[i].first == locate) cache[index].copy[i].second = value;
//miss
    Sleep(5);
    if (cache[index].avail <= 3)//如果还有空位就写进去
    {
        cache[index].copy[cache[index].avail] = make_pair(locate, value);
        cache[index].avail++;
    }
    else
        cache[index].copy[0] = make_pair(locate, value);//覆盖第一个
    Memory[locate] = value;
    return;
```

代码的主要框架为：
①根据地址找到该地址对应的 cache 组号。
②同样倒序遍历 4 个储存位，若 hit，则需要改变 cache 中相应位置的值
③若 miss，则如果有空位就把这组新的地址和 value 值写进空的位置上，如果没有空位则同样是覆盖第一个
④注意到为了实现的简便，这里我采用**直写和非写分配**方式，因此最后直接访问内存，将 value 值存入内存
Ps：为了模拟出使用 cache 机制更快，我**运用了 sleep 函数**，若不用 cache 机制，则将程序运行速度放慢；并且，在使用 cache 机制下，若 miss 同样利用 sleep 函数模拟出 miss 要访问内存而因此更慢的特点，而 hit 则不需要 sleep，表示访问地址的速度最快。

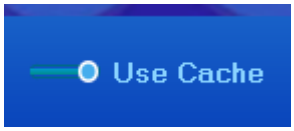
三、 界面实现

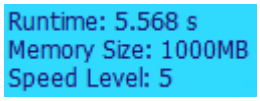
1、在阶段二的基础上增加了 cache 按键，用于查看运行中的 cache 状态

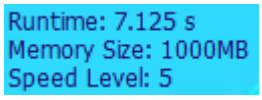


会将每个 cicle 中的 cache 中的所有值打印出来，一共 4 组 cache，每组可以放 4 对值，每对值表示地址和地址中的值，用箭头指向表示值存于地址之中。

2、增加了应用 cache 和不应用 cache 两种模式，便于直观感受 cache 对存取效率的影响

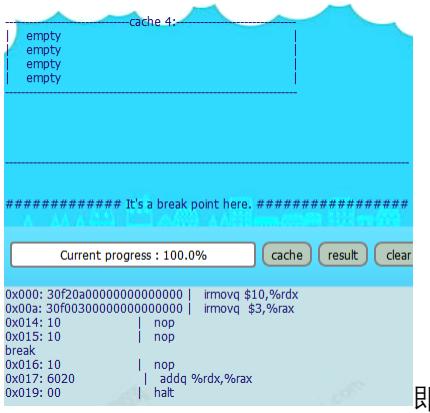






如图，在相同的 Speed Level 下，左边在 Use_Cache 模式下的运行时间要比右边不在 Use_Cache 模式的运行时间短，因此 cache 的存取高效的特点就模拟出来了。

3、将 cache 增加进了调试功能中的可查看内容中



即在断点处打印出 cache 信息以实现调试

四、 实验总结

到这里第三阶段也结束了，整个 Y86-64 模拟器的 PJ 也终于全部结束了，在整个过程中，有 debug 的痛苦，也有实现功能的喜悦，虽然我的模拟器可能没有别人的厉害，但在我心中**我自己做的就是最棒的了**。