实验二：Tomasulo

计71 韩荣 2016010189

1、 实验目的

1、理解流水线中乱序执行的逻辑

2、了解并实现Tomasulo算法

3、增加Tomasulo算法的功能，如添加JUMP指令。

4、与计分牌算法进行比较

2、 实验要求

实验的基本要求为设计实现不带JUMP的Tomasulo算法，输出其log，并对其进行性能测试。

实验的拓展要求为设计实现JUMP的Tomasulo算法，输出其log。

3、运行方法与设计思路

实验环境

MACOS 10.15.1，Apple clang version 11.0.0 (clang-1100.0.33.8)

编译方法

可以使用Makefile，直接make后运行main即可，需要注意命令行参数的使用：

运行方法

```
./main basic 0/1/2/3/4 0/1
```

上述命令第一个参数输入“basic”，第二个参数输入序号，可以对basic进行测试和log输出，第三个参数控制是否输出Cycle信息，Cycle信息打印在屏幕上（1表示要输出）。

```
./main extend filename 0/1
```

上述命令第一个参数输入“extend”，第二个参数输入序号，可以对其他文件进行测试和log输出，第三个参数控制是否输出Cycle信息，Cycle信息打印在屏幕上（1表示要输出）。

实验思路

本次实验一共使用了如下类：

Instruction.h

```
class Instruction{  //指令类，用于记录指令信息
public:
    std::string Op;
    std::string Dst;
    std::string Src1;
    std::string Src2;
    // 记录指令运行时间
    int Issue;
    int ExecComp;
    int WriteResult;
    int Count;
    int fill;
    explicit Instruction(std::string instr);
};
```

Simulator.h

```
class rs{       //保留站的一项
public:
    explicit rs();
    std::string Busy;
    std::string Op;
    std::string Vj;
    std::string Vk;
    std::string Qj;
    std::string Qk;
    int Instr;
    int Allocate;
    int TimeLeft;
    int ExecTime;
    int Status;
    void issue();
    void exec();
    void update();
    void done();
    void writeback();
};
class lb{           //Load Buffer的一项
public:
    explicit lb();
    std::string Busy;
    std::string Address;
```

```
    int Instr;
    int Allocate;
    int ExecTime;
    int TimeLeft;
    int Status;
    void issue();
    void exec();
    void update();
    void done();
    void writeback();
};
class RStation{     //保留站
public:
    rs Ars[6];
    rs Mrs[3];
    void Print();
};
class LoadBuffer{   //Load Buffer
public:
    lb LB[3];
    void Print();
};
class Register{     //Register
public:
    explicit Register();
    std::string R[32];
    void Print();
};
class FU{           //FU
public:
    explicit FU();
    int ld[2];
    int add[3];
    int mul[2];
};
class Simulator{    //Simulator
private:
    std::string IS;
    std::string Ready;
    std::string Done;
    std::string WB;
    RStation RS;
    LoadBuffer L;
    Register R;
    std::vector<Instruction> Instructions;
    FU resource;
```

```
public:
    int Cycle;
    int CanIssue;
    std::string filename;
    std::string stofill;
    int IssuePointer;
    explicit Simulator();
    bool CheckNumber(std::string s);
    bool CheckComplete(rs instr);
    bool CheckLoadComplete(lb instr);
    void Work(std::string file,std::string log);
    void TryIssue(int pointer);
    void Exec();
    void TryExec();
    int IsVacant(std::string op,int pointer);
    bool FindPlace(int pointer);
    bool NotFull();
    void Tomasulo();
    void Print();
    void PrintLog();
    void Clean(std::string op,int fill);
    double WriteBack();
    void ShowInstrStatus();
};
```

其中最重要的是Simulator类，用于模拟器的实现，包括Tomasulo算法的各个部分，以及各个保留站和FU的协调调度。

根据Tomasolu算法，在Simulator类中实现一个Tomasulo函数，可以将一个周期的内容分为如下几个部分：

```
void Simulator::Tomasulo(){
    while(NotFull()){   //如果所有指令没有都至少完成一次
        WriteBack();        //写回
        TryIssue(IssuePointer);   //发射
        Exec(); //执行
        TryExec();  //检查是否就绪
        Print();    //打印Cycle信息
        Cycle += 1; //进入下一个Cycle
    }
}
```

写回:可以发现这个算法首先进行的是写回操作，在写回操作中，首先需要将结果计算出来，并更新保留站中等待这个结果的信息，之后如果寄存器中该寄存器对应的保留站与写回指令相同，则写回寄存器，否则不写，避免WAW冲突。最后，将相关的FU和RS/LD Buffer信息清空，方便下一条指令进行操作。

发射：Simulator类中维护一个IssuePointer，这个IssuePointer指向下一条将要发射的指令。发射时需要检查是否还有剩余的保留站，如果有的话，就直接发射，并将目标寄存器中的值修改为对应保留站的内容。值得注意的是，由于寄存器重命名，所以无论上一个是否已经写回，都可以直接修改目标寄存器中的值。

执行：对所有正处于执行状态的保留站中的剩余时间-1，需要处理两种特殊情况：1、执行完毕，则跳转到完毕状态并记录下这条指令的Exec Comp，2、除法的除数为0，则直接结束除法，跳转到完毕状态并记录下这条指令的Exec Comp。

检查是否就绪：因为写回的原因，可能会改变保留站中指令的就绪情况，对他们进行检查，如果Vj，Vk都具备，则可以进入就绪状态，在下一周期进行执行。

打印Cycle信息：便于打印每个Cycle的详细信息。

以上便是Tomasulo Basic算法的大致思路，通过上述思路可以完成一个不带JUMP指令的Tomasulo算法。

4、拓展功能

1、带JUMP的Tomasulo算法设计

为了实现JUMP指令，需要将其放在Ars保留站中，且在Simulator类中额外增加一个信息量CanIssue，在Instruction中也增加一个计数量Count，仅有在Count == 1时，更新Instruction的Issue、Exec Comp、Write Back值，实现只记录第一次运行指令的信息。

JUMP指令的本质和RS中的运算相似，也可以按照Tomasulo算法的上述五个流程进行，需要注意的是，在Issue阶段，它是不会修改Register的值的，并且会关闭CanIssue，阻塞发射，在Write Back阶段，它可能会修改IssuePointer的值，会打开CanIssue，重新允许发射。

由于支持JUMP新增的一些元素

```
class Simulator{
    int CanIssue;     //用于控制发射
    int LogicPointer;    //增加只增不减的逻辑指针，用于指示发射的先后顺序
    int FindEarliest(std::vector<int> READY,std::string op);  //FU有限时，用于寻找最早发射的指令使用FU
};
class rs{
    int LogicPointer;    //记录本指令发射的逻辑指针
}
class lb{
    int LogicPointer;    //记录本指令发射的逻辑指针
}
```

实际上，在由Basic到支持JUMP的过程中，我对代码进行了很大的重构。之前的指令信息（如Status，TimeLeft）被记录在Instruction类中，这样的话会有很大的问题，因为在引入JUMP后，按照这个逻辑，应该新添加Instruction来记录他的执行信息，但是这样Instructions里面的总指令数就会增多，且很难判断什么时候该结束，因为原本的指令由于JUMP可能很久后才执行，其序号不可预测。

因此在支持JUMP时，我采用了新的记录方法——使用RS和LB来记录一项指令的Status和TimeLeft的信息，这样的话可以放心地修改IssuePointer的值，不需要担心新添加指令的问题，Instructions类的功能变成了一个存放指令的表，顺便记录指令第几次被访问，除此之外没有别的功能。这样的重构大大简化了引入JUMP后的逻辑。

2、关于乱序执行与效率

在支持JUMP的尝试中，我发现了一个有趣的新问题，即在实验要求中我们要求先就绪的先执行。由于偶然，在重构代码时我没有按照这个规则，而是按照序号靠前的RS先执行，但令我意外的是，对于我们的basic测例，这种乱序执行的方法竟然能够提高代码运行的效率。

最后修改为逻辑指针靠前的指令先执行后，发现普遍用了更多的周期数，我便对这个点进行了一些自己思考。

从理论上来说，逻辑指针靠前的指令先执行，则它的依赖项先执行完，其位置相对靠前这种情况其实用先就绪先执行能够更快完成靠前的指令，能够在大部分情况下提高靠前指令的运行效率。

从概率上来说，如果采用序号靠前的RS先执行，那么序号靠前的RS很可能是逻辑指令靠后的指令，这样的程序如果具有局部数据依赖，先执行后发射的反而可能会提高指令运行的效率。

因此两种方法互有优劣，但是先执行逻辑指针靠前的指令使得整个过程很稳定可估计，因此往往采用这一种方法。

3、关于程序写法与效率

这里讨论的效率不是CPI，而是机器运行的效率，即Simulator本身的运行效率。开始时我的设计一切以Instruction为中心，这样的方法可以十分简便的表达一条指令的状态，并通过便历指令进行查找和更新，逻辑简洁，非常利于实现。但随着指令数的增加，便历指令的弊端就出现了，因为需要便历，所以效率很低且绝大多数指令都是不需要更新或者不需要写回的，这种方法效率极低，特别是在运行Big_test这种文件时。

因此，将以指令为中心改为以保留站为中心，每次仅需便历保留站，将其中对应的指令拿出来单独分析即可，这样一次便历仅需6+3+3个保留站便可完成，将时间复杂度从0(n^2)变成了0(1)，同时采用保留站中心的思路可以非常方便JUMP指令的实现，虽然basic版本不如以指令为中心的实现直观，但无论是效率还是扩展性都大大提高。

是所谓“牢骚太盛防断肠，风物长宜放眼量”，不计眼前小利，才可能有未来大得。

5、与计分牌算法的异同

计分牌算法：

1、逻辑电路结构简单，耗费低，但指令流出慢

2、直接对FU进行操作，FU的总数会决定结构冲突的频率。

3、并行性取决于程序内容，如果高度相关，则会降低性能

4、乱序执行由于没有重命名，会导致WAW、WAR冲突增加阻塞。

Tomasulo：

1、Tomasolu算法可以根据重命名直接进行发射，利用保留站进行换名，只要还剩RS即可发射。

2、将指令取到保留站中，这样可以及时记录数据依赖，可以在源数据未写回寄存器时，就对目的寄存器进行修改而不会影响结果，提高了效率

3、冲突检测和指令执行控制机制分开，是分布式控制，而计分牌是集中控制。

4、消除了WAW和WAR所导致的阻塞。

6、实验结果

所有问价的实验结果已经在Log文件夹中给出，可以发现运行的效率较高，且结果是正确的。

对于performance的部分，经过测试，两个的文件CPI：

MUL.nel = 5

Big_test = 2

两个文件的运行时间：

MUL = 0.0234s

Big_test = 12.415s

(注：在测试运行时间时，关闭了log的输出，如果加上输出，时间会大于上述时间)

7、设计NEL文件

本NEL文件主要用于讨论与计分牌算法的差异

```
LD R0,0x5           //1
LD R1,0x3           //2
ADD R2,R0,R1        //3
SUB R3,R2,R0        //4
MUL R1,R2,R0        //5
DIV R1,R2,R0        //6
```

可以发现这个测例中包含了WAW冲突和WAR冲突其中第3与第5条指令有WAR冲突，在计分牌算法会等待直至写回后，才可发射第五条指令。而第5、6条指令会产生WAW冲突，在计分牌算法中也会阻塞，但上述两种情况在Tomasulo算法中，由于保留站的存在，可以很好的发射与执行。

其Log如下，可以发现发射是连续的，没有受到阻塞：

```
1 4 5
2 5 6
3 9 10
4 13 14
5 14 15
6 14 15
```

7、实验小结

本次实验对于Tomasolu算法的理解更加透彻了，开始以Instruction为主进行TimeLeft的计算等操作，能够完成Basic，但在性能测试的时候，表现较为糟糕，后来便使用RS为主的计算方式，大大提高了效率。而后再此基础之上完成了JUMP功能， 并对可能的提高效率方法进行了分析，收获很大。

同时，由于没有参考任何代码，本次实验从构思到实现都经过了较为仔细的思考过程，收获颇丰，对于Tomasolu的掌握也很不错。

感谢助教和老师的付出！