Tomasulo模拟器实验报告

计63 肖朝军 2016011302

概述

- **实现功能**
  - 接受符合文法定义的 `NEL` 汇编语言的指令序列作为输入
  - 正确输出每条指令的发射时间周期、执行完成时间周期、写回结果时间周期
  - 正确输出各时间周期寄存器的数值
  - 正确输出各个时间周期保留站的状态、LoaderBuffer状态和寄存器结果状态
  - UI交互界面

指令格式	指令说明	运算周期	保留站数	运算部件数
ADD,F1,F2,F3	F1=F2+F3	3	6	3
SUB,F1,F2,F3	F1=F2-F3	3	6	3
MUL,F1,F2,F3	F1=F2*F3	12	3	2
DIV,F1,F2,F3	F1=F2/F3	40	3	2
LD ,F1, addr	F1=addr	3	3	2
JUMP,val,F1,0x1	if(F1!=val) pc+=addr;	1	6	3

- **实现原理**

  Tomasulo算法以硬件方式实现了寄存器重命名，允许指令乱序执行，可以有效提高流水线吞吐率与效率，该算法核心思想是：记录和检测指令相关，操作数一旦就绪就立即执行，把发生RAW（写后读）冲突的可能性降到最低，通过寄存器换名来消除WAR（读后写）和WAW（写后写）冲突。

- **实验环境**
  - 语言：Java
  - 库需求：Java标准库
  - 软件需求：Intellij
- **代码运行**
  - 使用intellij进行编译，修改main.java中的测试程序的地址即可。

实验设计思路

我们整体采用了面向对象的设计方法，将不同的功能部件抽象成一个模块，再将这些功能部件组成一个Tomasulo算法系统。基于Java的对象机制，我们实现了多个不同的模块：

- **Inst**：指令

  > 该模块是对指令的一个抽象，主要作用是将string类型的指令转化为Inst对象，并存储对应的指令类型、数值、发射时间、完成时间、写回时间等

```
public class Inst {
    public static final int ADD = 0;
    public static final int SUB = 1;
    public static final int MUL = 2;
    public static final int DIV = 3;
    public static final int LD = 4;
    public static final int JUMP = 5;

    static final String[] InstName = {"Add", "Sub", "Mul", "Div", "LD",
"Jump"};
    public static final int[] CYCLE = {3, 3, 12, 40, 3, 1}; // 每条指令对应的
运行周期


    public int opid; // 指令类型
    public ArrayList<Operator> operators = new ArrayList<>(); // 指令中操作
数

    public boolean jump; //专门为Jump指令保留的

    String strinst;

    public int issuetime = -1;
    public int runingtime = -1;
    public int wbtime = -1;

    public Inst(String inst){ // 用字符串类型初始化 Inst 类
      ...
    }

    public Inst copy(){ // 类的深拷贝
        ...
    }

}
```

- **Buffer**：保留站

> 该模块是对保留站的一个抽象，记录了保留站的名称，状态，流入该保留站的指令等。

```
class Buffer{
    public Inst inst = null; // 指令
    public boolean occupied = false; //是否被占用
    public boolean completed = false; // 指令是否已经完成
    public boolean running = false; //指令是否正在跑
    boolean waitWB = false; // 是不是等待写回，在刚运行完时为true，写回之后为
false

    String bufferName; //保留站名称

    class RegQ{ // 用于存储两个用于运算的寄存器的状态
        public boolean ready;
        public Buffer buffer = null;
        int value;
    }

    public RegQ qj = new RegQ();
    public RegQ qk = new RegQ();

    public int targetRegId;
    int tagetValue;

    void setBufferName(String _name){
        bufferName = _name;
    }

    public void issueInst(Inst _inst){ //将指令 _inst 发射到该保留站中
        ...
    }

    public boolean checkReady(){ // 检查两个寄存器 qj/qk 是否已经就绪
        ...
    }

    public void complete(int cycles){ // write back
    ...
    }
```

- **Calculator**：运算器

> 该类主要是对运算器的一个抽象， 该运算器可以抽象 加减运算器、乘除运算器、以及 loader

```
class Calculator{
    public Inst inst = null;
```

```
    public boolean busy = false;
    Buffer buffer;

    int time;

    public void getInst(Buffer _buffer){ // _buffer中指令已经就绪了，分配到该运算器上进行运算
        ...
    }

    public void run(final int cycles){ // 在第 cycles个时钟周期运行
        ...
    }
}
```

- **ComponentsManager**

> 该模块主要是对 保留站buffer和运算器的一个管理，包括指令发射、写回、检测等一系列操作

```
public class ComponentsManager {
    public ArrayList<Buffer> buffers;
    public ArrayList<Calculator> calculators;

    String name;

    public ComponentsManager(int bufferNum, int calNum, String _name){
        ...
    }

    public boolean issueInst(Inst inst, int cycles){ // 发射指令，有可能发射失败
        ...
    }

    public void match(){ // 为buffer匹配运算器
       ...
    }

    public void checkReady(){ // 检测buffer中的指令是否已经执行完毕
    ...
    }

    public void timeCycleRun(int cycles){ //所有的运算器进行计算
    ...
    }

    public void writeBack(int cycles){ // 写回，将所有的运行完的指令写回
```

```
        ...
    }

    public boolean checkFree(){ // 检测是否已经全部空了，用来判断程序是否已经运算
结束
        ...
    }

}
```

- **Registers**：寄存器

> 该类主要是对寄存器进行的一个抽象，保留了所有寄存器的值、状态、等待的保留站等信息。该类使用了设计模式中的单例模式，全局只有一个实例。

```
public class Registers{
    class Reg{ //单个寄存器
        public boolean wait = false;
        public Buffer waitBuffer;
        int value;

        public Reg(){}

        public void setWait(Buffer buffer){
            ...
        }

        public void setArrive(int _value){
            ...
        }
    }

    private static Registers ins = new Registers(32);

    public ArrayList<Reg> regs = new ArrayList<>();
    private Registers(int num){
        ...
    }
    static public Registers getInstance(){
        ...
    }

    public void setWait(int index, Buffer buffer){ //设置等待
        ...
    }

    public void setArrive(int index, int value){ // 运算完成，写到寄存器中
        ...
```

```
    }

    public boolean getWait(int index){ // 获取状态
        ...
    }

    public Buffer getBuffer(int index){ //获取等待的保留站
        ...
    }

    public int getValue(int index){ //获取值
        ...
    }
}
```

上述就是我在实现tomasulo模拟器时，设计的主要的一些类，每一个周期实现的顺序是：**发射指令 -> 检测是否有buffer指令就绪 -> 运算器运行 -> 写回**。

- 指令发射阶段：检测指令类型，根据类型选择不同的 元件管理器进行检测，检测是否有保留站为空，如果有则将该指令发射到该保留站中，否则该阶段不发射指令。
- 检测就绪阶段：在该阶段中，会遍历每一个被占用的保留站及loadbuffer，查看其中的指令所涉及的操作数是否已经就绪，如果就绪则该阶段可以开始运行，否则还需要继续等待。
- 运行阶段：每一个运算器都进行运算，如果运算时间周期没到，则继续等待下一个周期的运算，如果运算时间周期已经到了，则通知相应的保留站要在下一个周期进行写回操作，并空出相应的运算器资源。
- 写回阶段：该阶段将便利所有的保留站及loadbuffer，如果上一个周期运算器通知需要写回，则判断目的寄存器是否是在等待该保留站的值，如果是则更新寄存器的值，否则更新本身buffer的值即可。并释放该buffer资源。

实验结果

- 如下为对于test0.nel在第7个周期的运行结果图，分别给出了保留站状态、loadbuffer状态、指令状态、寄存器状态等值。

next end cycle: 7

保留站状态	Busy	Op	Vj	Vk	Qj	Qk
Ars1	Yes	Sub	2	1		
Ars2	Yes	Jump		0	ADDs0	
Ars3	No					
Ars4	No					
Ars5	No					
Ars6	No					
Mrs1	Yes	Div			LD2	ADDs0
Mrs2	No					
Mrs3	No					

LoadBuffer	Busy	Address		Issue	Exec	WB
LB1	No		LD,F1,0x2	1	4	5
LB2	No		LD,F2,0x1	2	5	6
LB3	Yes	0xFFFFFFFF	LD,F3,0xFFFFFFFF	3	7	
			SUB,F1,F1,F2	4		
			DIV,F4,F3,F1	5		
			JUMP,0x0,F1,0x2	6		
			JUMP,0xFFFFFFFF,F3,0xFFFFFFFD			
			MUL,F3,F1,F4			

Reg	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
State		ADDs0		LD2	MULs0											
Value	0x0	0x0	0x1	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0
Reg	F16	F17	F18	F19	F20	F21	F22	F23	F24	F25	F26	F27	F28	F29	F30	F31
State																
Value	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0

- 点击"end"按钮，可以直接跳到程序运行结尾。其余结果可通过运行程序检查，暂不列出。

finish end cycle: 25

保留站状态	Busy	Op	Vj	Vk	Qj	Qk
Ars1	No					
Ars2	No					
Ars3	No					
Ars4	No					
Ars5	No					
Ars6	No					
Mrs1	No					
Mrs2	No					
Mrs3	No					

LoadBuffer	Busy	Address		Issue	Exec	WB
LB1	No		LD,F1,0x2	1	4	5
LB2	No		LD,F2,0x1	2	5	6
LB3	No		LD,F3,0xFFFFFFFF	3	7	8
			SUB,F1,F1,F2	4	9	10
			DIV,F4,F3,F1	5	14	15
			JUMP,0x0,F1,0x2	6	11	12
			JUMP,0xFFFFFFFF,F3,0xFFFFFFFD	12	13	14
			MUL,F3,F1,F4	20	24	25

Reg	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
State																
Value	0x0	0x0	0x1	0x0	0xfff...	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0
Reg	F16	F17	F18	F19	F20	F21	F22	F23	F24	F25	F26	F27	F28	F29	F30	F31
State																
Value	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0

- 自己编写测例：我自己编写了一个测试样例，代码如下。该代码计算累加，从0开始每次加3，加够0x1000次。在循环结束之后，F1中应该为0x9000。

```
LD,F1,0x0
LD,F2,0x3
LD,F3,0x1000
LD,F4,0x1
LD,F5,0x1000
ADD,F1,F1,F2
SUB,F3,F3,F4
JUMP,0x0,F3,0x2
JUMP,0x1000,F5,0xfffffffc
MUL,F1,F1,F2
```

运行结果如下：可以看到结果是正确的。

finish end cycle: 40971

保留站状态	Busy	Op	Vj	Vk	Qj	Qk
Ars1	No					
Ars2	No					
Ars3	No					
Ars4	No					
Ars5	No					
Ars6	No					
Mrs1	No					
Mrs2	No					
Mrs3	No					

LoadBuffer	Busy	Address
LB1	No	
LB2	No	
LB3	No	

	Issue	Exec	WB
LD,F1,0x0	1	4	5
LD,F2,0x3	2	5	6
LD,F3,0x1000	3	7	8
LD,F4,0x1	6	9	10
LD,F5,0x1000	7	10	11
ADD,F1,F1,F2	8	11	12
SUB,F3,F3,F4	9	13	14
JUMP,0x0,F3,0x2	10	15	16
JUMP,0x1000,F5,0xfffffffc	16	17	18
MUL,F1,F1,F2	40966	40970	40971

Reg	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
State																
Value	0x0	0x9000	0x3	0x0	0x1	0x1000	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0
Reg	F16	F17	F18	F19	F20	F21	F22	F23	F24	F25	F26	F27	F28	F29	F30	F31
State																
Value	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0

总结

该次试验从0开始搭建了一个tomasulo模拟器，通过亲手实现该程序，学到了很多之前没有关注到的算法的细节，Tomasulo通过乱序执行代码，检测指令相关来消除冲突，提高了吞吐率。学习到了不少。