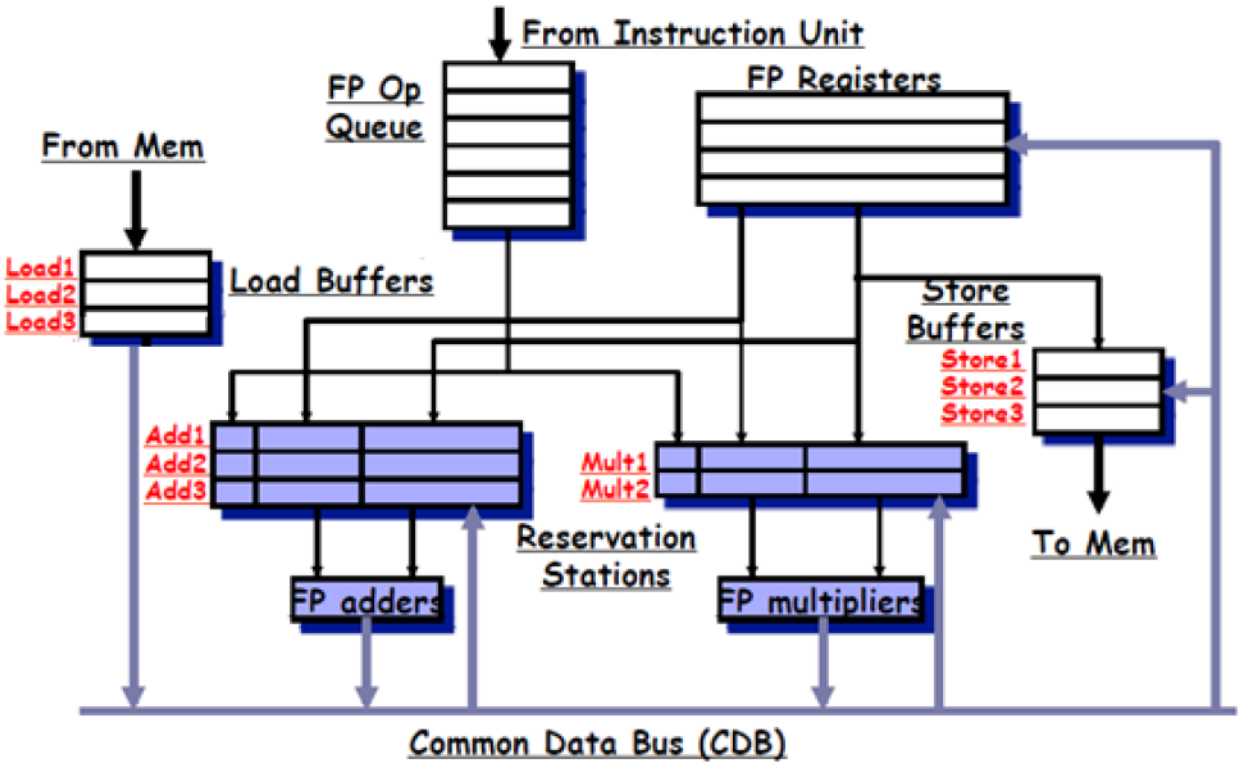
Tomasulo Simulator Exp Report

石景宜 2016011395

一、 实验背景

Tomasulo算法以硬件方式实现寄存器重命名，允许指令乱序执行，是提高流水线的吞吐率和效率的一种有效方式，该算法实现了寄存器重命名，允许指令乱序执行，是提高流水线吞吐率和效率的一种有效方式。该算法首次出现在IBM360/91处理机的浮点处理部件中，后广泛运用于现代处理器设计中。



二、 完成的功能

基础要求

- [x] 能够正确接受任意NEL 汇编语言编写的指令序列作为输入.
- [x] 能够正确输出每一条指令发射的时间周期，执行完成的时间周期，写回结果的时间周期。
- [x] 能够正确输出各时间周期的寄存器数值
- [x] 能够正确输出各时间周期保留站状态、LoadBuffer 状态和寄存器结果状态。

拓展要求

- [x] 设计美观的交互界面。
- [x] 丰富NEL 语言，为它添加更多的指令支持，并能够模拟这些指令的执行。

三、 实验设计

设计思路

- 用一个状态表来存储各个指令运行状态。
- 用不同的类来实现运算类保留站和装载类保留站。
- ISSUE阶段都从指令存储表中取出一条指令，检查对应保留站是否有空闲，若有，初始化指令状态和空闲保留站后放入状态表。
- EXCUTE阶段检查每一个保留站是否可以执行/正在执行/执行完毕，用一个计时器控制执行周期，用等待队列控制执行顺序。若有执行完毕，则立即释放对应运算器。
- WB阶段写回对应寄存器并广播，释放对应保留站。

扩展指令内容

除了实验要求的指令之外，我还实现了两条内存操作指令`LDM、ST`,CFG文法定义扩展如下：

```
Program := InstList
InstList := Inst
InstList := Inst ‘\n’ InstList
Inst := OPR ',' REGISTER ',' REGISTER ',' REGISTER
Inst := OPT ',' REGISTER ',' INTEGER
OPT := "LD" | "LDM" | "ST"
Inst := "JUMP" ',' INTEGER',' REGISTER ','INTEGER
OPR := "ADD"|"MUL”| "SUB"| "DIV"
```

新添加的两条指令有如下形式和含义：

```
"LDM" ',' REGISTER ',' INTEGER //将地址为INTEGER的内存单元装载到寄存器REGISTER
"ST" ',' REGISTER ',' INTEGER  //将寄存器REGISTER的值存储到地址为INTEGER的内存单元
```

实验框架

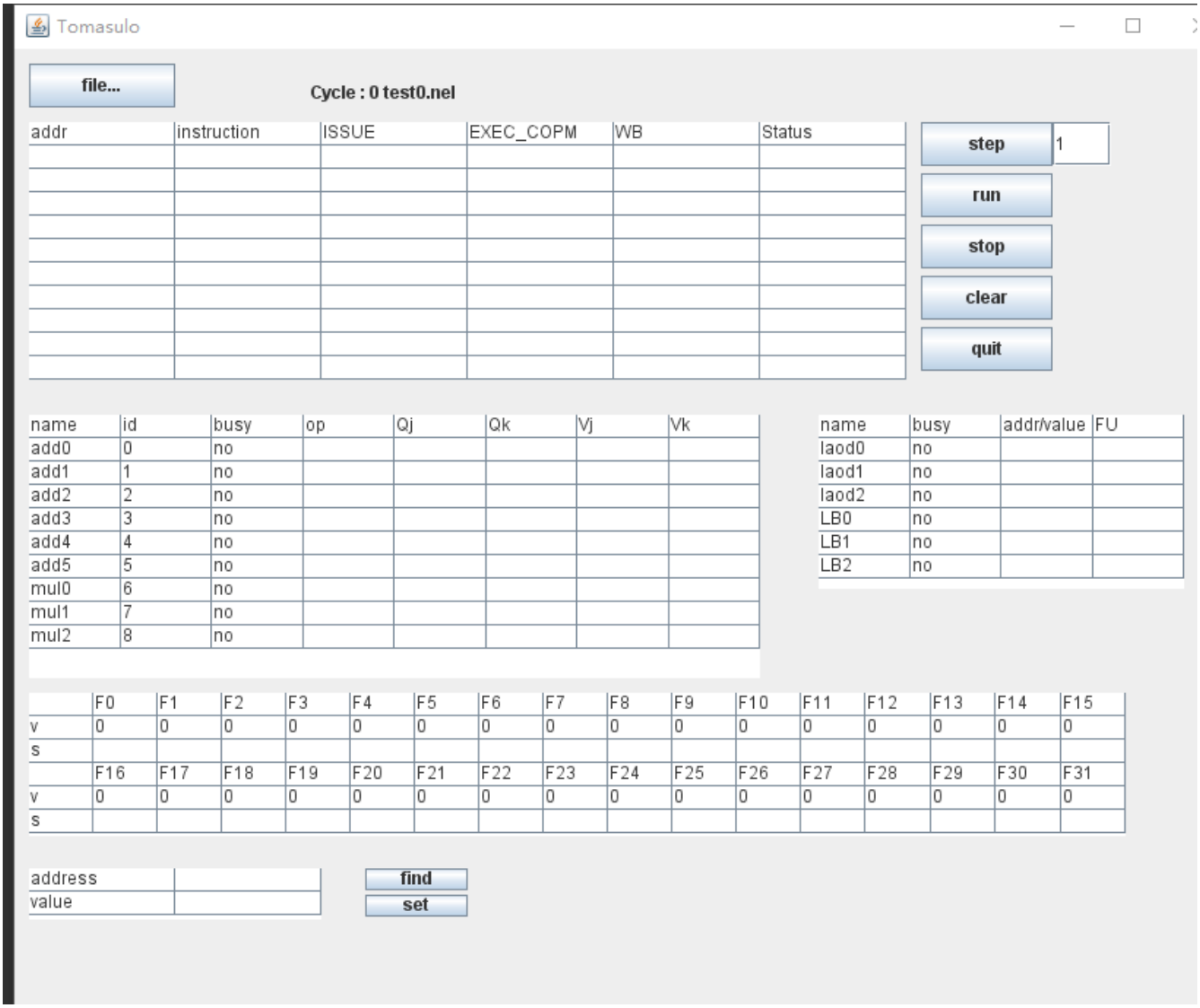
代码框架

本次实验没有引用和参考任何开源框架，使用java实现了Tomasulo算法和ui，代码共约1500行，，所有代码类功能如下：

- `CalculateStation.java` :算术型保留站类。通过该类可以实现MUL和ADD两种保留站实例。
- `InstructionState.java`：指令执行状态类，存储了指令的执行过程中的所有信息。
- `LSStation.java`：装载、存储保留站，通过该类实现了LOAD和LOAD_BUFFER两种保留站，分别对应于指令`LDM,ST`和`LD`。
- `Tomasulo.java`：主算法类，其中包含一个内部类来创建图形界面和根据用户操作控制内部算法执行。其中的`step_next`函数是主要算法执行函数，用来进入下一个周期。

图形界面

图形界面如下：



其中的表格，按照从上到下，从左到右的顺序，分别为：

- 指令状态：每一行的值为：指令地址、指令内容、发射周期，执行完毕周期、写回周期、当前状态。当前状态共*ISSUE*，*EXECUTE*，*WB*，*FINISHED*四种。**注意：该表格只显示最后执行的十条指令。**
  - 实际执行中还引入了*READY*状态，用来表示已经得到需要操作数，正在等待运算部件空闲的指令。
- 运算保留站：add为加减保留站，mul为乘除保留站。
- 装载保留站：laod为内存读写操作的保留站，LB为寄存器装载操作的保留站。
- 寄存器状态：v为寄存器当前值，s为寄存器当前状态。
- 内存查询表：address、value均可修改。

一共八个按钮，他们功能如下：

- `file...`：浏览文件系统选择nel源码文件。默认读取`testcases/test0.nel`
- `step`：前进N步，N为其后文本框内数值，可修改。
- `run`：连续运行直到结束。
- `stop`：打断连续运行，暂停。
- `clear`：清空当前执行状态，下次执行时从头开始。
- `quit`：退出程序。
- `find`：查找左侧address对应值，值显示在value对应表格。

- `set`：按照左侧表格指定值设置内存。

注意：

- **所有图形界面显示的值和需要输入的值均直接使用十进制，仅源代码需要用十六进制表示。**
- **在填写查询/写入内存的表格时，填入数字后一定要按回车，否则无法生效**

执行部件少于保留站的处理

为了保证进入保留站且做好执行前准备的指令能够严格规定顺序执行，采用了四个先进先出的等待队列来控制执行顺序。

```
// wait queue
Queue<Integer> addWQ = new LinkedList<Integer>();   //加减
Queue<Integer> mulWQ = new LinkedList<Integer>();   //乘除
Queue<Integer> lswQ = new LinkedList<Integer>();    //内存
Queue<Integer> loadWQ = new LinkedList<Integer>();  //寄存器装载
```

在EXCUTE阶段：

- 首先按照指令顺序检查指令是否已经做好执行前准备，若是，加入相应等待队列。
- 若等待队列不为空，检查是否有空闲运算部件，若有，则取出等待队列队尾保留站，开始执行。直到等待队列为空或者对应空闲运算部件数为0。

控制冲突处理

没有采用分支预测技术，故在JUMP指令需要跳转的时候直接暂停流水线。

内存设置

模拟器硬件中设置了4096个内存单元，每一个内存单元都是一个int。任何在[0,4096)之外的内存访问都是非法访问，对应的LDM操作会得到0值，ST操作会无效，但执行时间不会变化。

为了解决WAW,WAR,RAW冲突，在LD和ST进入EXCUTE阶段之前进行检查，若检查到冲突，直接暂停流水线。

四、 正确性测试

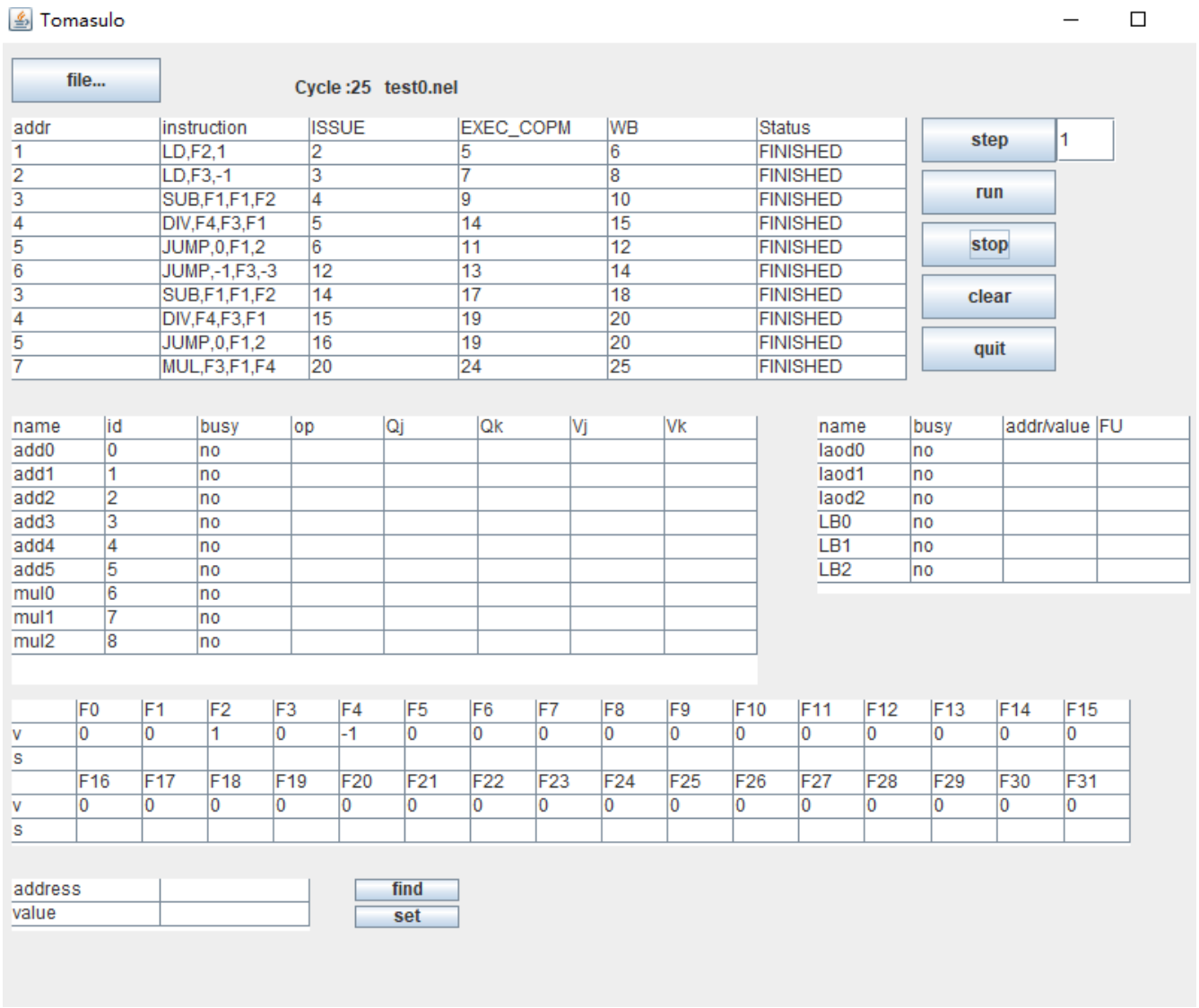
首先测试给定测试文件。

`test0.nel`

首先，在本次实验中将乘法和除法执行周期都改为四。

执行结果和每周期执行结果和`例子.pdf`中给出的基本一致。

（唯一的不同之处在于我将寄存器写回后清空了寄存器状态，而pdf中是寄存器状态保存了寄存器值，这个问题在ui显示上很好解决，但是在寄存器状态实际值不能改变为寄存器实际值，因为这样可能和保留站编号混淆。）



addr	instruction	ISSUE	EXEC_COPM	WB	Status
1	LD,F2,1	2	5	6	FINISHED
2	LD,F3,-1	3	7	8	FINISHED
3	SUB,F1,F1,F2	4	9	10	FINISHED
4	DIV,F4,F3,F1	5	14	15	FINISHED
5	JUMP,0,F1,2	6	11	12	FINISHED
6	JUMP,-1,F3,-3	12	13	14	FINISHED
3	SUB,F1,F1,F2	14	17	18	FINISHED
4	DIV,F4,F3,F1	15	19	20	FINISHED
5	JUMP,0,F1,2	16	19	20	FINISHED
7	MUL,F3,F1,F4	20	24	25	FINISHED

name	id	busy	op	Qj	Qk	Vj	Vk
add0	0	no					
add1	1	no					
add2	2	no					
add3	3	no					
add4	4	no					
add5	5	no					
mul0	6	no					
mul1	7	no					
mul2	8	no					

name	busy	addr/value	FU
laod0	no		
laod1	no		
laod2	no		
LB0	no		
LB1	no		
LB2	no		

	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
v	0	0	1	0	-1	0	0	0	0	0	0	0	0	0	0	0
s																
	F16	F17	F18	F19	F20	F21	F22	F23	F24	F25	F26	F27	F28	F29	F30	F31
v	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
s																

test1.nel

所有配置和文档中一致，执行结果如下：

file...

Cycle :54 test1.nel

addr	instruction	ISSUE	EXEC_COPM	WB	Status
0	LD,F1,3	1	4	5	FINISHED
1	LD,F2,0	2	5	6	FINISHED
2	LD,F3,-1	3	8	9	FINISHED
3	ADD,F2,F1,F2	4	9	10	FINISHED
4	MUL,F4,F1,F3	5	21	22	FINISHED
5	DIV,F2,F3,F1	6	49	50	FINISHED
6	SUB,F4,F2,F1	7	53	54	FINISHED
7	JUMP,0,F1,-2	8	9	10	FINISHED

step 1

run

stop

clear

quit

name	id	busy	op	Qj	Qk	Vj	Vk
add0	0	no					
add1	1	no					
add2	2	no					
add3	3	no					
add4	4	no					
add5	5	no					
mul0	6	no					
mul1	7	no					
mul2	8	no					

name	busy	addr/value	FU
laod0	no		
laod1	no		
laod2	no		
LB0	no		
LB1	no		
LB2	no		

	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
v	0	3	0	-1	-3	0	0	0	0	0	0	0	0	0	0	0
s																
	F16	F17	F18	F19	F20	F21	F22	F23	F24	F25	F26	F27	F28	F29	F30	F31
v	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
s																

address	
value	

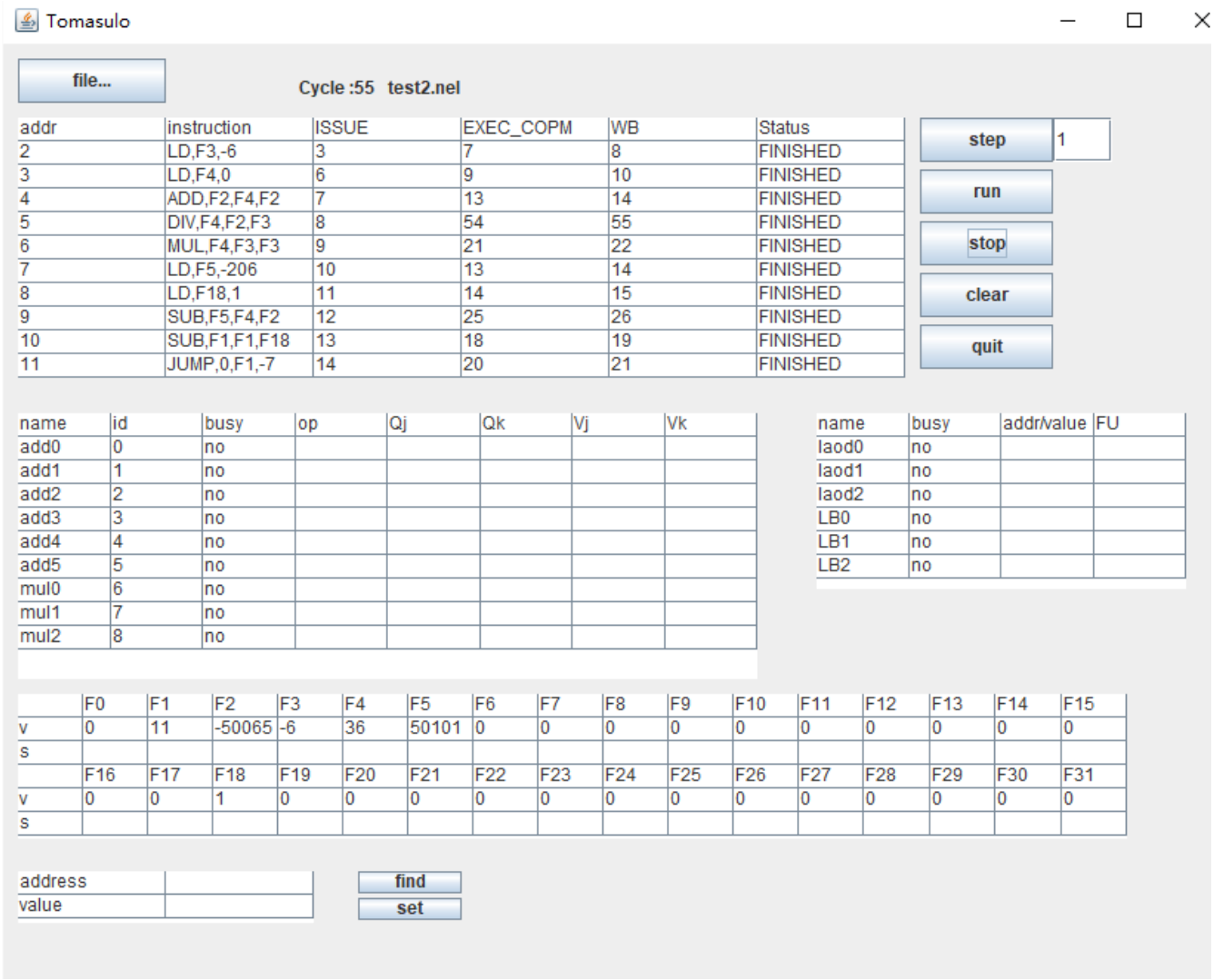
find

set

和预计结果一致。

test2.nel

所有配置和文档中一致，执行结果如下：

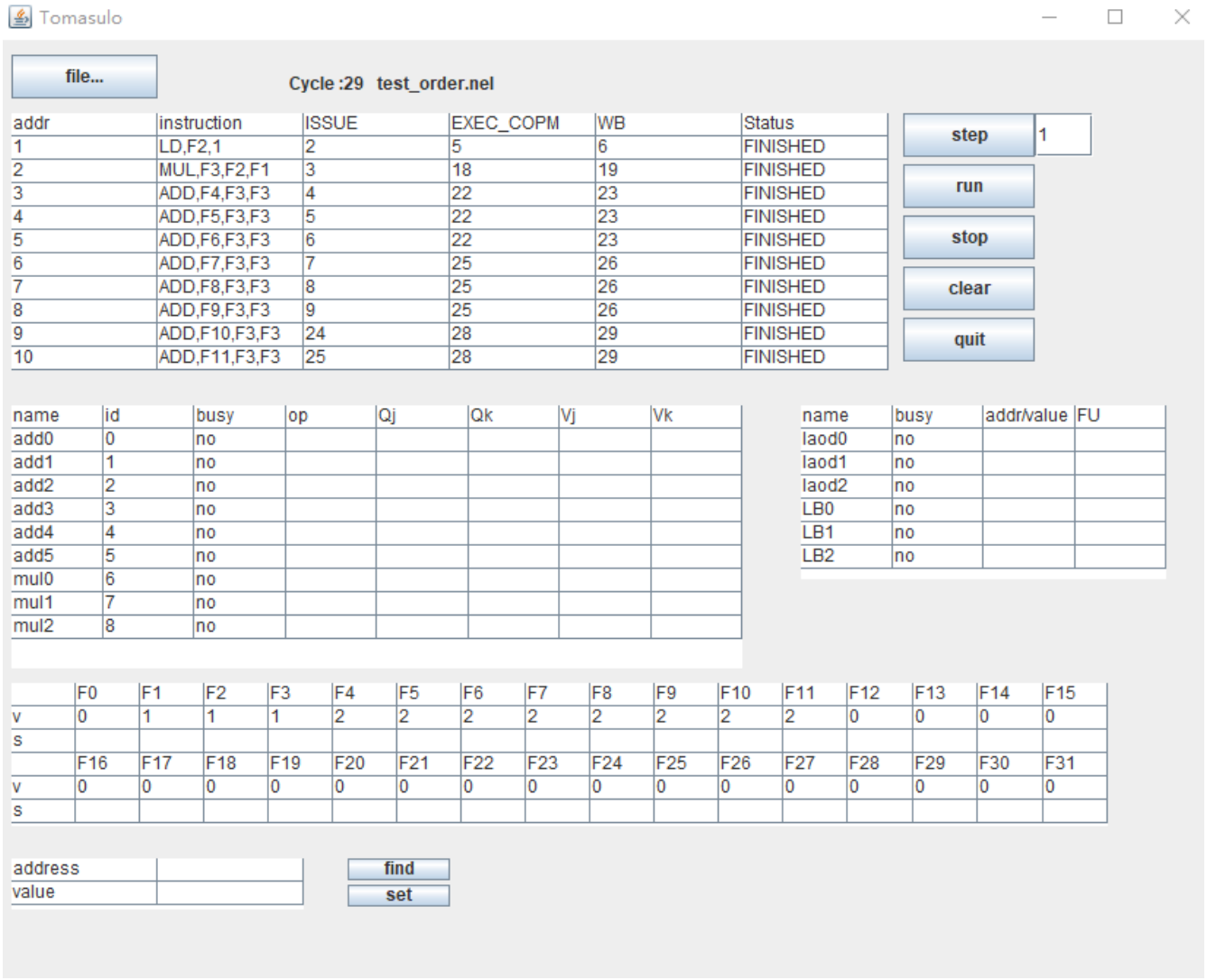


可以看出，相应运算部件满时，地址为3的LD指令等待了两个周期才发射，其余结果也符合预期。

指令执行顺序测试 `test_order.nel`

```
LD,F1,0x1
LD,F2,0x1
MUL,F3,F2,F1
ADD,F4,F3,F3
ADD,F5,F3,F3
ADD,F6,F3,F3
ADD,F7,F3,F3
ADD,F8,F3,F3
ADD,F9,F3,F3
ADD,F10,F3,F3
ADD,F11,F3,F3
```

开始用一条乘法指令阻塞后面指令的执行，后面加法指令塞满加法保留站后同时释放，看是否按照指令序号顺序执行。



addr	instruction	ISSUE	EXEC_COPM	WB	Status
1	LD,F2,1	2	5	6	FINISHED
2	MUL,F3,F2,F1	3	18	19	FINISHED
3	ADD,F4,F3,F3	4	22	23	FINISHED
4	ADD,F5,F3,F3	5	22	23	FINISHED
5	ADD,F6,F3,F3	6	22	23	FINISHED
6	ADD,F7,F3,F3	7	25	26	FINISHED
7	ADD,F8,F3,F3	8	25	26	FINISHED
8	ADD,F9,F3,F3	9	25	26	FINISHED
9	ADD,F10,F3,F3	24	28	29	FINISHED
10	ADD,F11,F3,F3	25	28	29	FINISHED

name	id	busy	op	Qj	Qk	Vj	Vk
add0	0	no					
add1	1	no					
add2	2	no					
add3	3	no					
add4	4	no					
add5	5	no					
mul0	6	no					
mul1	7	no					
mul2	8	no					

name	busy	addr/value	FU
laod0	no		
laod1	no		
laod2	no		
LB0	no		
LB1	no		
LB2	no		

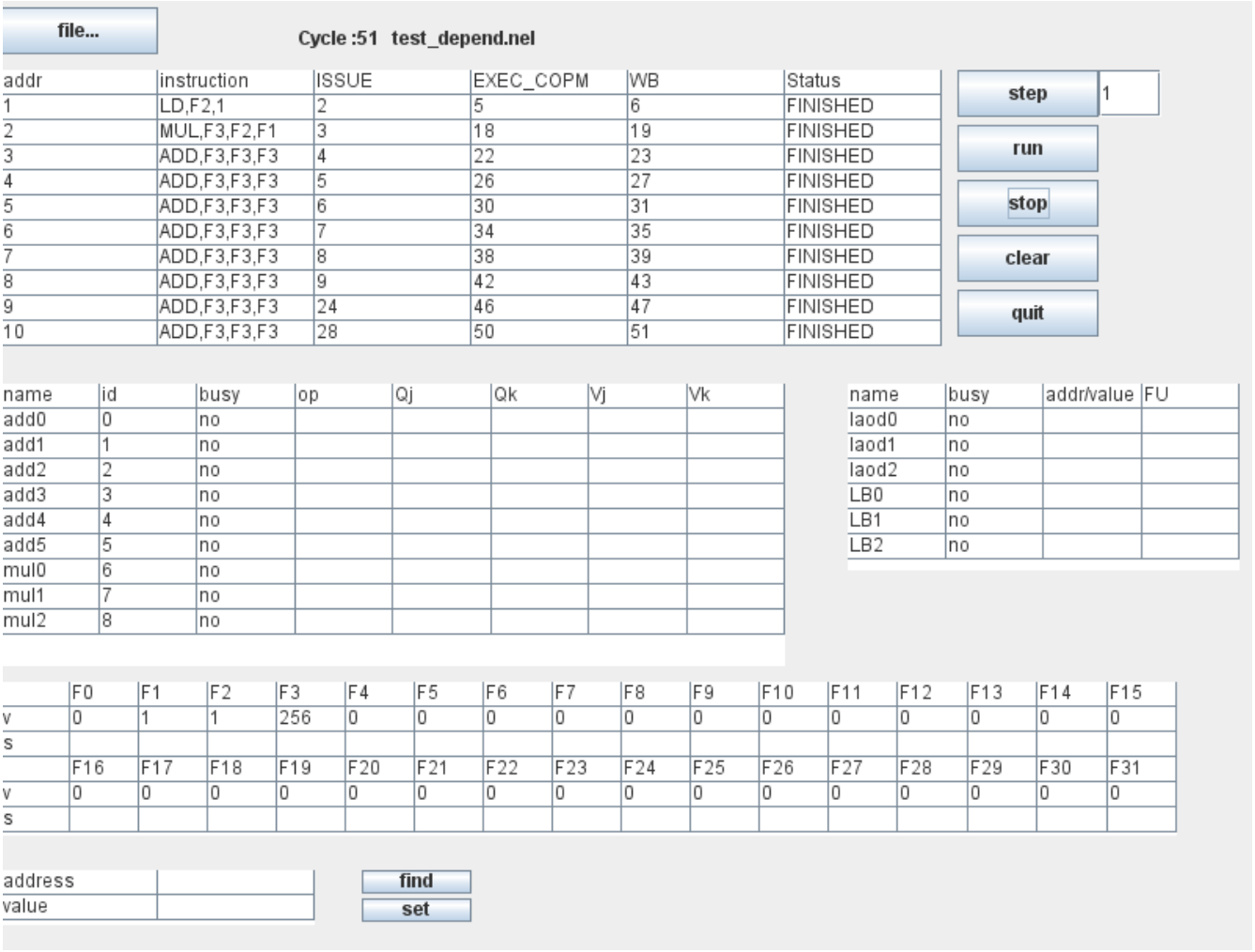
	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
v	0	1	1	1	2	2	2	2	2	2	2	2	0	0	0	0
s																
	F16	F17	F18	F19	F20	F21	F22	F23	F24	F25	F26	F27	F28	F29	F30	F31
v	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
s																

结果和预期一致。

依赖指令测试 `test_depend.nel`

```
LD,F1,0x1
LD,F2,0x1
MUL,F3,F2,F1
ADD,F3,F3,F3
ADD,F3,F3,F3
ADD,F3,F3,F3
ADD,F3,F3,F3
ADD,F3,F3,F3
ADD,F3,F3,F3
ADD,F3,F3,F3
ADD,F3,F3,F3
```

所有加法指令相互依赖，应该等待前一条写回后执行。

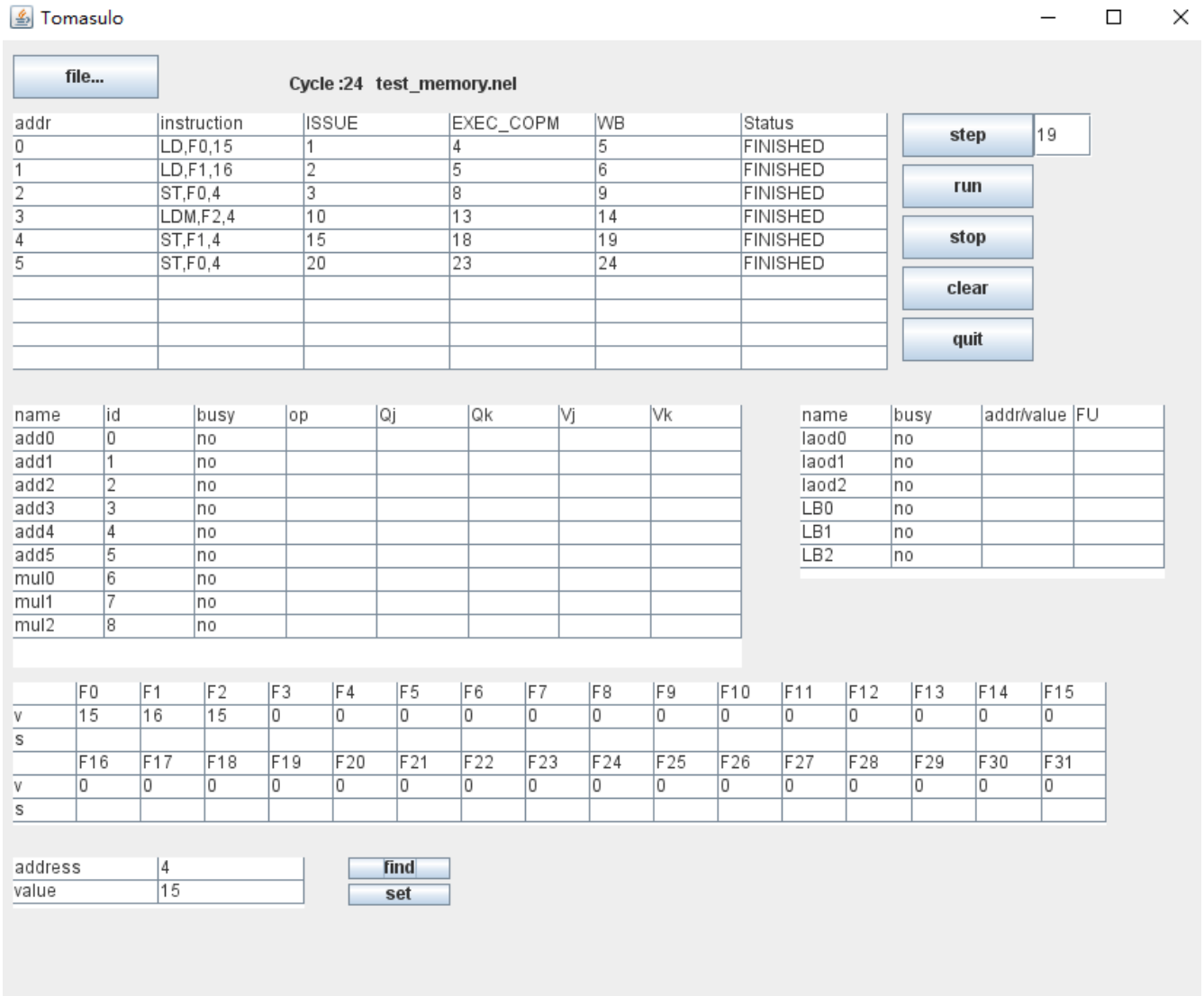


和预期结果一致。

内存读写测试 `test_memory.nel`

```
LD,F0,0xf
LD,F1,0x10
ST,F0,0x4
LDM,F2,0x4
ST,F1,0x4
ST,F0,0x4
```

该代码包含了读后写、写后写、写后读的冲突。



addr	instruction	ISSUE	EXEC_COPM	WB	Status
0	LD,F0,15	1	4	5	FINISHED
1	LD,F1,16	2	5	6	FINISHED
2	ST,F0,4	3	8	9	FINISHED
3	LDM,F2,4	10	13	14	FINISHED
4	ST,F1,4	15	18	19	FINISHED
5	ST,F0,4	20	23	24	FINISHED

name	id	busy	op	Qj	Qk	Vj	Vk
add0	0	no					
add1	1	no					
add2	2	no					
add3	3	no					
add4	4	no					
add5	5	no					
mul0	6	no					
mul1	7	no					
mul2	8	no					

name	busy	addr/value	FU
laod0	no		
laod1	no		
laod2	no		
LB0	no		
LB1	no		
LB2	no		

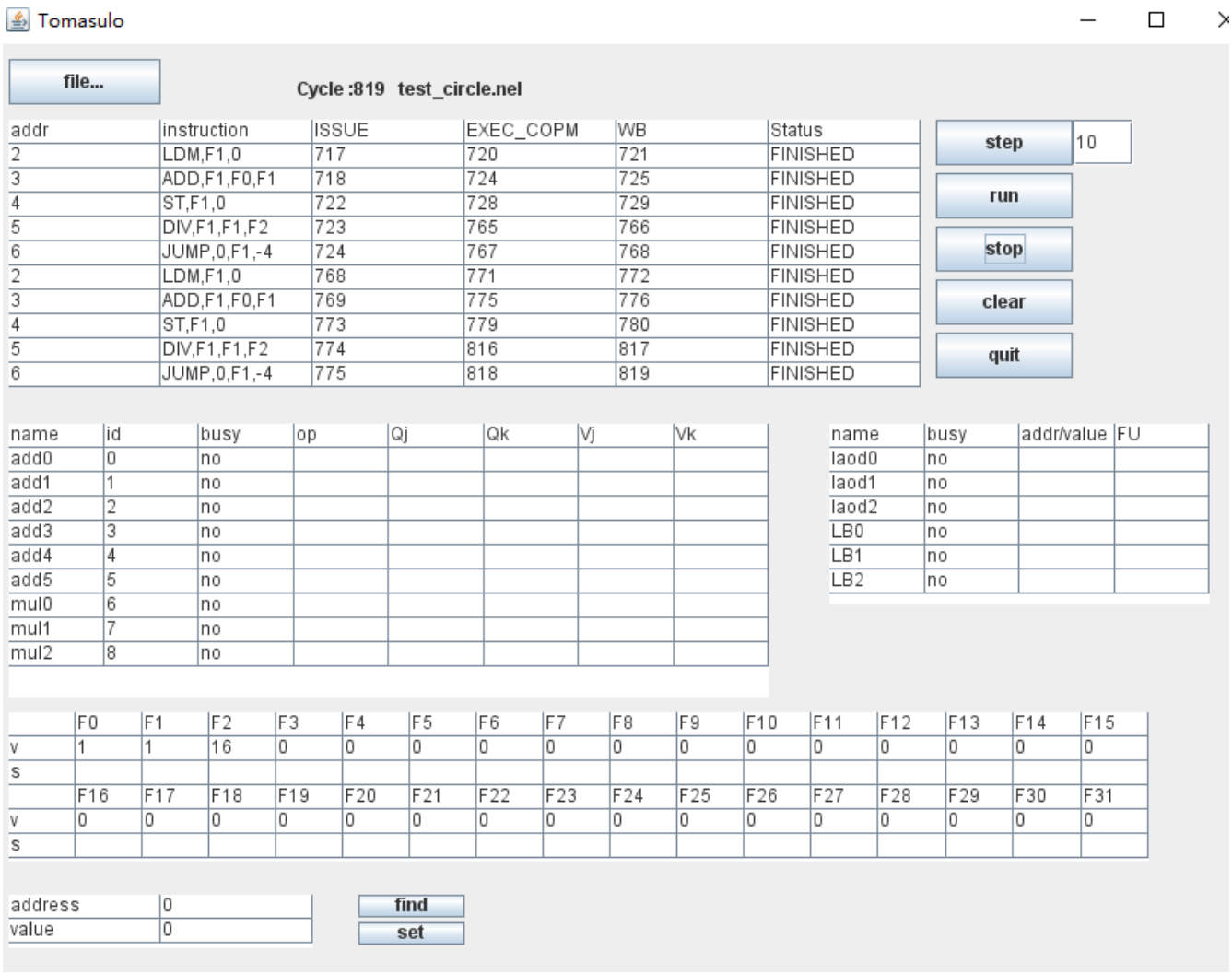
	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
v	15	16	15	0	0	0	0	0	0	0	0	0	0	0	0	0
s																
	F16	F17	F18	F19	F20	F21	F22	F23	F24	F25	F26	F27	F28	F29	F30	F31
v	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
s																

实际运行结果和预期一致。

循环测试

```
LD,F0,0x1
LD,F2,0x10
LDM,F1,0x0
ADD,F1,F0,F1
ST,F1,0x0
DIV,F1,F1,F2
JUMP,0x0,F1,0xfffffffc
```

每一次循环F1增加1，每一次循环需要51个cycle，到F1/F2不为0需要F2个周期，即16个周期，共需要3+51*16个周期。



addr	instruction	ISSUE	EXEC_COPM	WB	Status
2	LDM,F1,0	717	720	721	FINISHED
3	ADD,F1,F0,F1	718	724	725	FINISHED
4	ST,F1,0	722	728	729	FINISHED
5	DIV,F1,F1,F2	723	765	766	FINISHED
6	JUMP,0,F1,-4	724	767	768	FINISHED
2	LDM,F1,0	768	771	772	FINISHED
3	ADD,F1,F0,F1	769	775	776	FINISHED
4	ST,F1,0	773	779	780	FINISHED
5	DIV,F1,F1,F2	774	816	817	FINISHED
6	JUMP,0,F1,-4	775	818	819	FINISHED

name	id	busy	op	Qj	Qk	Vj	Vk
add0	0	no					
add1	1	no					
add2	2	no					
add3	3	no					
add4	4	no					
add5	5	no					
mul0	6	no					
mul1	7	no					
mul2	8	no					

name	busy	addr/value	FU
laod0	no		
laod1	no		
laod2	no		
LB0	no		
LB1	no		
LB2	no		

	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
v	1	1	16	0	0	0	0	0	0	0	0	0	0	0	0	0
s																
	F16	F17	F18	F19	F20	F21	F22	F23	F24	F25	F26	F27	F28	F29	F30	F31
v	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
s																

结果和预期一致。

五、运行方式

要修改保留站个数、指令执行周期、运算器个数，直接修改`Tomasulo.java`中：

```
// excute time
    public static int T_ADD = 3;
    public static int T_SUB = 3;
    public static int T_MUL = 12;
    public static int T_DIV = 40;
    public static int T_LDM = 3;
    public static int T_JUMP = 1;
    public static int T_ST = 3;
    public static int T_LD = 3;
    // reserve station num
    public int LS_STATION_NUM = 3;
    public int MUL_STATION_NUM = 3;
    public int ADD_STATION_NUM = 6;
    public int LOAD_BUFFER_NUM = 3;
    // function unit num
    public static int ADDER = 3;
    public static int MULT = 2;
```

```
18      public static int LOAD = 2; // for memory load and store
19      public static int LOADER = 2; // for load
```

在src文件夹下直接运行`make run`来运行。

默认打开的指令文件是`testcases/test0.nel`。

六、 实验总结

本次实验花的时间和精力较多，很多bug都是在写了ui之后写报告时做测试才找到的。这也是我第一次用软件实现硬件模拟器。

本次实验收获很大，谢谢老师和助教的帮助，希望以后还是减少一点工作量或者增大一点本次实验分值。

//因为提交作业之后和同学交流发现例子修改了，我的实现一开始是在写回时才释放运算器，已经按照最新的例子修改为了在执行完毕后立即释放运算器。希望这一点能够在实验文档中做出规范。