实验三 Tomasulo实验报告

PB15111662 李双利

- 实验目的
  - 加深对指令级并行性及其开发的理解；
  - 加深对Tomasulo算法的理解；
  - 掌握Tomasulo算法在指令流出、执行、写结果各阶段对浮点操作指令以及load和store指令进行什么处理；
  - 掌握采用了Tomasulo算法的浮点处理部件的结构；
  - 掌握保留站的结构；
  - 给定被执行代码片段，对于具体某个时钟周期，能够写出保留站、指令状态表以及浮点寄存器状态表内容的变化情况。
- 实验要求

  设计和实现一个Tomasulo算法模拟器。

  基本要求：针对程序中直线型代码，可乱序执行、乱序完成。
  - a. 能够正确输出每个周期之后保留站的内容。

    保留站基本信息：

站名	状态	操作码	第一操作数值	第一操作数状态	第二操作数值	第二操作数状态

  - b. 能够正确输出每个周期之后寄存器状态表的内容。

    寄存器状态表基本信息：

寄存器名
寄存器状态（0：不等待保留站的内容；n 表示等待的保留站名n>0）
寄存器内容（状态为0时，该值才有意义）

  - c. 能够正确输出每个周期之后指令状态表的内容（指令分为浮点运算指令和load/store指令），指令状态分为流入，执行和写回。

    指令状态表基本信息

    标志出每条指令流出、执行、写回这三个阶段所在的周期号
  - d. 实现带界面的模拟器
  - e. 执行时间可设置

  较高要求：
  - 支持分支指令

- 代码编译说明

  由于编码问题，在编译该代码时可能出现编码错误的问题，需要按照以下命令指定utf-8进行编译：

```
javac -encoding utf-8 Tomasulo.java
```

- 测试程序&回答

  在实现后的Tomasulo模拟器中指令以下指令进行模拟。

```
L.D     F6, 21（R2）
L.D     F2, 2, 0（R3）
MUL.D   F0, F2, F4
SUB.D   F8, F6, F2
DIV.D   F10, F0, F6
ADD.D   F6, F8, F2
```

  - a. 给出在第5个时钟周期后，保留站的内容

    首先是参考程序的结果：

保留站

Time	名称	Busy	Op	Vj	Vk	Qj	Qk
	Add1	Yes	SUB.D	M1	M2		
	Add2	No					
	Add3	No					
	Mult1	Yes	MULT.D	M2	R[F4]		
	Mult2	Yes	DIV.D		M1	Mult1	

    然后是我的程序的结果：

Time	名称	Busy	Op	Vj	Vk	Qj	Qk
	Add1	yes	SUB.D	M1	M2		
	Add2	no					
	Add3	no					
	Mult1	yes	MULT.D	M2	F4		
	Mult2	yes	DIV.D		M1	Mult1	

保留站

    分析：

    按照 Tomasulo 的流程,5 个时钟周期流出了 5 条指令,其中 load 指令 2 条, mult 指令 1 条,sub 指令 1 条,divd 指令 1 条。其中 mult 和 sub 指令与 ld 产 生生相关,相 关寄存器编号为 F2。M1 与 M2 代表寄存器 F6 和 F2 数据已准 备好。2 条 load 指令执行行 完毕并且已经写回。

  - b. 给出在第10个周期后，保留站，寄存器状态表的信息

    首先是参考程序的结果：

保留站

Time	名称	Busy	Op	Vj	Vk	Qj	Qk
	Add1	No					
	Add2	Yes	ADD.D	M3	M2		
	Add3	No					
5	Mult1	Yes	MULT.D	M2	R[F4]		
	Mult2	Yes	DIV.D		M1	Mult1	

寄存器

字段	F0	F2	F4	F6	F8	F10	F12	F14	F16	F18	F20	F22	F24	F26	F28	F30
Qi	Mult1	Load2		Add2	Add1	Mult2										
值		M2		M1	M3											

然后是我的程序的结果：

Time	名称	Busy	Op	Vj	Vk	Qj	Qk
	Add1	no					
	Add2	yes	ADD.D	M3	M2		
	Add3	no					
5	Mult1	yes	MULT.D	M2	F4		
	Mult2	yes	DIV.D		M1	Mult1	

保留站

字段	F0	F2	F4	F6	F8	F10	F12	F14	F16	F18	F20	F22	F24	F26	F28	F30
状态	Mult1	Load2		Add2	Add1	Mult2										
值		M2		M1	M3											

结果分析：

按照 Tomasulo 的流程,10 个时钟周期 6 条指令均已流出,且 2 条 load 指令 执行行完毕并且写回。sub指 令也已执行行完毕并写回,目目前保留站中剩下 add、mult、div3 条指令。add 指令正在执行行阶段最后一一个时钟周期,没有 写回。div 指令等待 mult 指令 的结果。乘法指令还有 5 个时钟周期执行行完毕。

- 设计思想以及算法
  - 类成员定义&变量

    在Tomasulo算法中，每过一个clock需要维护四个表，具体来说，

    - **load**部件状态表负责记录Load指令的执行情况（可以认为该部分是保留站Rs的一部分），包括load的指令以及当前部件的占用情况和load之后得到的value。

      ```
      public class LoadStatus{
          public boolean busy;
          public int count_down;
          public int inst_index;
      }
      ```

    - 指令状态表负责记录每一条指令的各阶段执行时间和流程状态

      ```
      public class InstStatus{
          public boolean executing;
          public int rs_index;
      }
      ```

    - 保留站**RS**负责记录各个功能部件（包含两个浮点加法运算单元和三个浮点乘运算单元）的占用情况，以及操作数是否准备就绪，如果需要等待的话还要记录等待的功能部件序号。

```
public class RsStations{
    public boolean busy;
    public int op;
    public int qj;
    public int qk;
    public int count_down;
    public int inst_index;
}
```

- 寄存器状态表**RegisterStatus**负责记录F0~F30浮点寄存器的占用情况，以及其存储的计算值。

```
public  class RegStatus{
    public int state;
    public int value;
}
```

- **Tomasulo流程core()**

  Tomasuolo算法共有三个执行流程，分别是发射**Issue**、执行**Execute**和写回**Write result**阶段，每个阶段都会对表进行维护，并且在GUI中显示更新状态。具体实现过程我是按照PPT上所讲的思路进行实现。并且在原来框架基础上做了部分改进和优化。

  - **Issue**

    在发射阶段共有两种类型的指令发射，一种是Load型指令，需要占用Load部件，所以每次取指令时若为Load指令则要取检测**Load状态表**看是否有空闲的Load部件可用，没有的话需要等待其他Load指令执行完之后释放部件再进行占用然后发射Load指令，同时更新**Load状态表** 。

    对于FP类型的指令发射过程也是完全类似的，ADD/SUB指令需要占用ADD部件，MULT/DIV指令需要占用Mult部件，所以检查**RS表**中空闲的部件分配给相应地FP指令然后发射，否则需要等待。伪代码如下

```
Issue
FP Operation:
     Wait until :  Station r empty
    Action or bookkeeping:
            if(RegisterStat[rs].Qi!=0) {RS[r].Qj <- RegisterStat[rs].Qi}
             else {RS[r].Vj != Reg[rs]; RS[r].Qj <- 0 }
            if(RegisterStat[rt].Qi!=0) {RS[r].Qk <- RegisterStat[rt].Qi}
            else {RS[r].Vk!=Reg[rt]; RS[r].Qk <- 0 }
            RS[r].Busy <- yes; RegisterStat[rd].Qi = r;
Load:
      Wait until: Buffer r empty
      Action or bookkeeping:
           if(RegisterStat[rs].Qi!=0)
              {RS[r].Qj <- RegisterStat[rs].Qi}
           else {RS[r].Vj <- Reg[rs]; RS[r].Qj <- 0 }
           RS[r].A <- imm;   RS[r].Busy <- yes;
           RegisterStat[rt].Qi = r;
```

  - **Execute**

在执行阶段，FP指令和Load指令都有一个执行周期时长，在`RsStations`和`LoadStatus`两个表类中都有一个`count_down`成员用于记录剩余的执行周期数。在执行时，首先要检查各个源操作数是否就绪（即考虑`RAW`等相关情况），存在相关的话在**RS表**中会有记录并且处于执行等待，一旦检测到所有的操作数就绪之后开始执行，每一个clock过后`count_down`减去1。伪代码如下。

```
    FP Operation
        wait until: (RS[r].Qj=0) and (RS[r].Qk=0)
        Action or bookkeeping:
            computer result: Operands are in Vj and Vk
    Load-store step1
        wait until: RS[r].Qj =0 & r is head of load-store queue
         Action or bookkeeping:
             RS[r].A <- RS[r].Vj + RS[r].A;
   Load   step2
        wait until: Load Step1 complete
        Action or bookkeeping:
               Read from Mem[RS[r].A]
```

- **Write Result**

  进行写回阶段时，判断其开始的方法是检测到**Load表**或者**RS表**中存在处于busy状态的部件并且其`count_down`减为了0说明执行完毕，那么写回时要寄存器状态表进行更新。伪代码如下。

```
     FP Operation or Load
         Wait until: Execution complete at r  & CDB available
         Action or bookkeeping
            Any x (if (RegisterStat[x].Qi=r) {Regs[x] <- result; RegisterStat[x].Qi <-
0})
            Any x (if(RS[x].Qj =r) {RS[x].Vj <- result; RS[x].Qj <- 0});
            Any x (if(RS[x].Qk =r) {RS[x].Vk <- result; RS[x].Qk <- 0});
           RS[r].Busy <- no;
```

- 改进和优化

  在具体用代码实现时，在给定java框架基础上我遇到了以下三个问题并且针对性的进行了相应地解决。

  - 三个阶段的更新顺序

    在实际实现Tomasulo模拟器时，核心的算法是在一个`core()`函数里实现的，那么每一步（也就是进行一个clock），有的指令处于发射阶段，有的处于执行或者写回阶段，如果直接按照三个阶段的顺序按照伪代码思想进行实现的话，一个问题就是会影响前一个阶段在当前周期内的进展会立刻影响到下一阶段（应该是下一个周期才会影响到），针对这一问题，一个解决思路是发射阶段和执行阶段所操作的指令做一个标记，然后在之后的阶段中进行判断。我的实现是在每次执行`core()`函数时，先从写回阶段开始更新、然后才是执行和发射，这样避免了前一阶段对后一阶段的影响。但是在写回阶段也可能对执行阶段产生影响，所以也有一个标记执行阶段的数组来保证每个执行周期内各个阶段是独立不会互相影响的。

  - 执行结果的M序列问题

    在每一条Load或者FP指令执行完成之后都会有一个M_i来表示结果，但由于乱序执行，其执行结束的时间顺序往往不是指令的排列顺序，为了能够实现M_i是按照指令顺序（指令在同一个周期内写回），在写回阶段时按照指令顺序进行循环然后标记为M_i

- **NOP**指令的插入

  如果直接在给出的java框架上实现Tomasulo算法，在执行的指令序列中插入NOP指令会执行错误（经过测试确实会），所以我对其GUI框架做了部分修改，在**指令状态表**生成时跳过NOP指令。

- 实验结果分析和总结

Tomasulo算法则通过动态调度的方方式，在不影响结果正确性的前提下，重新排列指令实际执行行的顺 序（乱序执行行），提高高时间利用用效率。 该算法与之前同样用用于实现指令流水水线动态调度的计分板不同在于它使用用了寄存器重命名机制。指 令之间具有数据相关性（例如后条指令的源寄存器恰好是前条指令要写入入的⽫目目标寄存器），进行行动 态调度时必须避免三类冒险：写后读（Read-after-Write, RAW）、写后写（Write-after-Write, WAW）、读后写（Write-after-Read, WAR）。第一一种冒险也被称为真数据相关（true data dependence）， ⽽而后两种冒险则并没有那么致命，它们可以由寄存器重命名来予以解决。Tomasulo 算法使用用了一一个共享数据总线（common data bus, CDB）将已计算出的值⼲广广播给所有需要这个值作 为指令源操作数的保留站。该算法尽可能降低了使用用计分板技术导致的流水水线停顿，从⽽而改善了并 行行计算的效率。