Tomasulo算法实现及演示系统

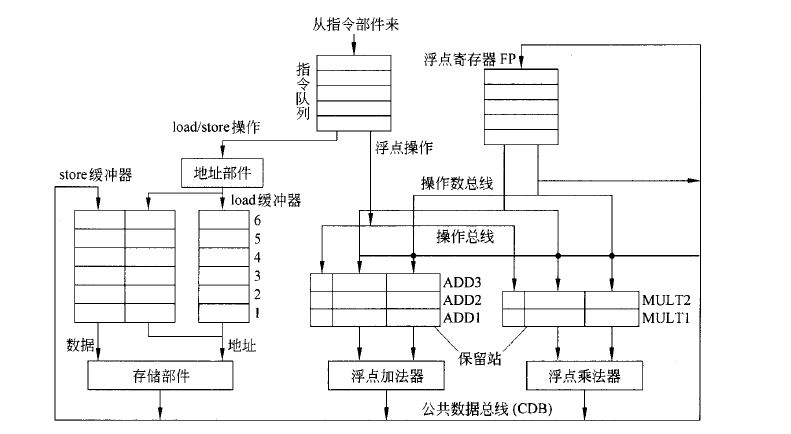
|  |  |  |
| --- | --- | --- |
| 2010011281 | 王凝枰 | [iwnking@gmail.com](mailto:iwnking@gmail.com) |
| 2010011293 | 刘彪 | [Liubiao2638@gmail.com](mailto:Liubiao2638@gmail.com) |
| 2010011297 | 胡沛筠 | [Penn.peiyun@gmail.com](mailto:Penn.peiyun@gmail.com) |

# Tomasulo算法原理

## 核心思想

1. 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冲突的的可能性减小到最小；
2. 通过寄存器换名来消除WAR冲突和WAW冲突

## 调度算法示意图



其中load，store，加法器和乘法器都有自己的保留站。这些保留站的作用就是用来作为寄存器换名的目标地址。由于指令时不可能直接使用这些保留站的，所以换名之后不会引起和其他指令对寄存器使用的冲突。

另外，Tomasulo 算法采用分布的保留站，因而具有以下两个特点：

1. 冲突检测和指令执行控制是分布的。每个功能部件的保留站中的信息决定了什么时候指令可以在该功能部件中执行。
2. 计算结果通过CDB直接从产生它的保留站传送到需要它的功能部件，而不用通过寄存器。

## 算法执行步骤

1. **流出**

从指令队列的头部取一条指令，如果该指令的操作所要求的保留站是空闲的，就把该指令送到该保留站，并且如果其操作数在寄存器中已经就绪，就把这些操作数从寄存器中送入保留站，如果操作数还没有就绪，就把将产生该操作数的保留站的标识送入该保留站。这样，一旦该数据计算出来，这条指令便可以立即执行。

1. **执行**

如果某个操作数还没有计算出来，本保留站将监视CBD，等待所需的计算结果。一旦那个结果产生，它将本放到CBD上，本保留站将立即获得该数据。当两个操作数都已经就绪之后，便可以开始执行。

1. **写结果**

功能部件计算完毕之后，就将计算结果放到CBD上，所有等待该计算结果的寄存器和保留站都同时从CBD上读取该数据。

此外，每个保留站有以下几个字段：

* Op：要对源操作数进行的操作
* Qj,Qk:将产生源操作数的保留站号，等于0表示操作数已经就绪且在Vj,Vk中，或者不需要操作数。
* Vj,Vk：源操作数的值，对于每一个操作数来说，V或Q字段只有一个有效。
* Busy：表示本保留站是否忙。
* A：仅load和store用，用来保存立即数的值。
* Qi：寄存器状态表，用来保存将要把结果写入其中的保留站号。

## Tomasulo算法伪代码

1. **指令流出**
2. 浮点运算指令

**进入条件**：有空闲保留站（设为r）

操作和状态表内容修改：

if (Qi[rs] != 0)

RS[r].Qj = Qi[rs];

else {

RS[r].Vj = Regs[rs];

RS[r].Qj = 0;

}

if (Qi[rt] != 0)

RS[r].Qk = Qi[rt];

else {

RS[r].Vk = Regs[rt];

RS[r].Qk = 0;

}

RS[r].Busy = yes;

RS[r].Op = Op;

Qi[rd] = r;

1. load和store指令

**进入条件**：缓冲器有空闲单元

操作和状态表内容修改：

if (Qi[rs] != 0)

RS[r].Qj = Qi[rs];

else

RS[r].Vj = Regs[rs];

RS[r].Qj = 0;

RS[r].Busy = yes;

RS[r].A = Imm;

对于load指令:

Qi[rt] = rt;

对于store指令：

if (Qi[rt] != 0)

RS[r].Qk = Qi[rt];

else {

RS[r].Vk = Regs[rt];

RS[r].Qk = 0;

}

1. **执行**
2. 浮点操作指令

**进入条件**：(RS[r].Qj == 0) && (RS[r].Qk == 0)

操作数和状态修改：进行计算，产生结果

1. load/store指令

**进入条件**：(RS[r].Qj == 0) 且r成为load/store缓冲队列的头部

1. **写结果**
2. 浮点指令和load指令

**进入条件**：保留站r执行结束，且CDB就绪

操作和状态表内容修改：

对任意寄存器x,

if (Qi[x] == r) {

Regs[x] = result;

Qi[x] = 0;

}

对任意保留站x,

if (RS[x].Qj == r) {

RS[x].Vj = result;

RS[x].Qj = 0;

}

if (RS[x].Qk == r) {

RS[x].Vk = result;

RS[x].Qk = 0;

}

RS[r].Busy = no;

1. store指令

**进入条件**：保留站r执行结束，且RS[r].Qk == 0

操作和状态表内容修改：

Mem[Rs[r].A] = RS[r].Vk;

RS[r].Busy = no;

# Tomasulo 模拟器设计及实现

基于上述Tomasulo算法原理，我们用Python语言实现了Tomasulo算法模拟器。该模拟器包括：

1. data\_type.py: 定义了几种数据类型包括：
   1. Ins: 指令类型，其中包括指令执行状态state，耗费周期time\_cost，操作码op，源和目标寄存器的信息；
   2. Reg: 浮点寄存器堆类型，其中保存了状态表以及对应的寄存器值表；
   3. RS: 保留站项类型，其中包含了op, qj, qk, vj, vk, A, busy等信息；此外还保存了占用此保留站的指令；
   4. Mem: 内存块类型，其中包含了数据数组以及访存和设置内存的接口；
2. func\_unit.py: 定义了3种功能元件的类，包括访存元件Memory\_unit、加减法元件Adder\_unit和乘除法元件Multiplier\_unit；元件中包含信息有：
   1. busy: 当前元件是否被占用；
   2. rs\_id: 占用当前元件的保留站编号，默认为-1；
   3. end\_time: 当前元件结束被占用的时间；
   4. result: 缓存当前元件产生的结果，在模拟器中实际结果已经计算得到，等到相应的周期时，才将result中缓存的结果赋给寄存器用于更新；
3. controller.py: 其中包括模拟器的主要控制器，具体来说，包括：
   1. 数据
      1. ins\_list: 用于存放所有的指令序列，另外指令序列将由read\_ins方法读入，当前即将流出的下一条指令序号为ins\_current\_id；
      2. memory: 模拟器所模拟的内存堆；
      3. rs\_list: 存放所有的保留站项，其中rs\_map用于标定每一种保留站在rs\_list中的上下界，需要注意的是，例如每一种保留站在rs\_list中都是连续的一块；
      4. mem\_unit, add\_unit, multi\_unit: 三种功能部件的实例；
      5. memory\_queue: 规定执行Load/Store操作的顺序的队列，遵从先进先出；
      6. clock\_now: 标识当前时钟周期；
   2. 方法
      1. read\_ins(self, ins\_text\_data): 将text形式的指令序列读入并解析存入ins\_list；
      2. write\_memory(self, index, data), read\_memory(self, index): 提供的直接读取和修改内存的借口；
      3. print\_state(self), print\_reg(self), print\_rs(self): 用于输出调试信息的接口；
      4. 关于模拟执行接口
         * step(self): 单步执行一个周期；
         * done(self): 判断所有指令是否执行完毕；
         * run(self): 执行读入的所有指令直到所有的指令执行完毕；
      5. 关于每一次模拟执行的内部过程
         * influx: 检查是否存在指令可以流出到保留站中，如果可以则流出。另外，每周期最多流出一条指令；
         * execute: 检查保留站中是否存在就绪指令，如果存在则执行相应程序。另外，不同功能部件间可以同时执行。
         * update: 检查是否存在功能部件在此周期开始时已经完成工作，如果完成，更新以占用此功能部件的保留站为源的寄存器

# Tomasulo 算法演示系统