Tomasulo Simulation 实验报告

刘丰源 2017011313

2020 年 5 月 14 日

目录

第一节 编译运行检察 **2**

第二节 设计思路 **3**

2.1 结构体设计 3

2.2 接口设计 4

2.3 算法设计 4

第三节 拓展功能 **4**

3.1 JUMP 指令 4

3.2 性能优化 5

3.2.1 测试方式 5

3.2.2 测试结果 5

3.2.3 优化方案 5

第四节 记分牌与 Tomasulo 比较 **5**

4.1 记分牌 5

4.2 Tomasulo 6

第五节 实验总结 **6**

第一节　编译运行检察

./run.sh Example.nel 1

第一个参数为文件名；第二个参数选择是否输出中间状态，1 为输出，其它为不输出。

结果位于 out.md 文件中，使用 markdown-pdf 浏览器打开后可以更美观的格式查看中间状态（效果如下）：

保留站状态

Name	Busy	Op	Vj	Vk	Qj	Qk
Ars 1	Yes	SUB	2	1		
Ars 2	Yes	JUMP			Ars 1	
Ars 3	No					
Ars 4	No					
Ars 5	No					
Ars 6	No					
Mrs 1	Yes	DIV			LB 3	Ars 1
Mrs 2	No					
Mrs 3	No					

Load Buffer 状态

Name	Busy	Imm
LB 1	No	
LB 2	No	
LB 3	Yes	-1

寄存器状态

R0	R1	R2	R3	R4	R5	R6	R7
0	Ars 1	1	LB 3	Mrs 1	0	0	0

运算部件状态

部件名称	当前执行指令	当前还剩几个周期
Add1	SUB,R1,R1,R2	2
Add2		
Add3		
Mult1		
Mult2		
Load1	LD,R3,0xFFFFFFFF	1
Load2		

输出结果整理于 Log 目录。

第二节　设计思路

2.1　结构体设计

代码位于 structs.hpp

- **Code**

保存指令的操作符和操作数。

- **FU**

功能部件，使用枚举类型。

- **ReservationStation**

保留站，保存了忙碌、剩余周期、操作符、操作数、等待的保留站（指针）等信息。

- **Register**

寄存器，保存了其数值、更新时间、等待保留站和保留站给出的数值。

- **CodeState**

指令状态，保存了发射、执行结束、写回时间，以及使用的保留站。

- **Log**

类似 CodeState ，但是只保留第一次执行的信息。

2.2　接口设计

核心代码位于 tomasulo.hpp ，主要提供以下功能：

1. set_nel(string s)：设置程序代码。
2. run(int n)：模拟运行 n 个周期。
3. print_clock/amrs/lb/reg/fu()：输出当前周期的保留站、寄存器、功能部件等信息。

2.3　算法设计

核心函数为 step() ，每次执行为一个时钟周期，依次执行以下函数：

1. clear_cdb：将所有执行结束的指令占用的保留站、功能部件释放，并且通知正在等待该保留站的保留站准备执行。如果指令为 JUMP ，则更新当前执行指令的位置。
2. issue：检察当前指令是否能够发射，如果为 JUMP 并且正在执行，则进行等待，否则按顺序执行。
3. lookup_fu：检察保留站中是否有指令可以就绪，若可以，则分配功能部件。
4. update：将保留站中正在执行的指令的剩余时钟周期减一；如果执行完成，则将其信息写入 cdb ，等待资源回收。

将 clear_cdb 放于最前面是因为，指令在 writeResult 状态的时候，其它用到该指令占用资源、或在等待该指令结果的指令，是可以就绪的，所以将释放资源放在最前面。

指令在刚就绪的时候不能执行，需要再额外等待一个周期，所以给保留站设置了 wait 变量。

第三节　拓展功能

3.1　JUMP 指令

1. 指令执行结束后，如果该指令为 JUMP ，则更新当前执行指令的位置（加一或跳转）。
2. 功能部件和保留站的分配与 ADD 相同，但是执行时间不同。
3. 指令发射时，需要判断当前指令是否是 JUMP 。如果是，且尚未执行完毕，则跳过发射，进行等待。
4. 指令发射成功后，如果发射指令是 JUMP ，则不更新当前执行指令的位置，否则按顺序加一。

3.2　性能优化

3.2.1　测试方式

1. 修改 src/main.cpp ，将其中的 `cout « ”time: ” « microtime() - start « ”s” « endl` 取消注释。
2. ./run.sh Big_test.nel 0 。第二个参数一定要是 0 ，关闭输出中间状态，否则会浪费大量时间在 IO 上。

3.2.2　测试结果

- Big_test：8.9 秒。
- Mul：0.014 秒。

3.2.3　优化方案

1. 传递指针，不使用拷贝构造，性能提高。保留站及其内部的数据需要被多个函数使用，为了保证数据的一致性，并且提高数据传递的性能，全部使用指针传递。
2. 将字符串比较变为整数比较。创建 enum FU ，对 fu 进行比较、判断时，原本需要使用字符串比较，改进后变为整数比较。原本查找某个部件的信息需要进行遍历，复杂度为 $O(n*len(m))$ ；改进后可以通过 fus[(int)fu] 直接访问，复杂度为 $O(1)$ 。
3. 为 ars、mrs、lb 开辟连续的内存，使得遍历更加便捷、快速。

第四节　记分牌与 Tomasulo 比较

4.1　记分牌

1. 逻辑电路仅相当于一个功能部件，结构简单，耗费低，但是指令流出慢。
2. 并行性取决于程序，如果程序中的指令均与前面相关，则性能不升反降。
3. 容量决定能在多大范围内寻找不相关指令，也称指令窗口。
4. 功能部件的总数决定了结构冲突的严重程度，采用动态调度后结构重读会更加频繁。
5. 乱序执行会产生更多的名相关，导致写后写、先读后写阻塞增多。

4.2 Tomasulo

1. 将记分牌的关键部分和寄存器换名技术结合，通过保留站实现换名。
2. 只要操作数有效，就将指令取到保留站中，避免指令流出时才到寄存器中取数据。即将执行的指令从相应的保留站中取得操作数，而不是从寄存器中取。
3. 一条指令流出时，存放操作数的操作数的寄存器名被换成对应于该寄存器保留站的名称（编号）。
4. 冲突检测和指令执行控制机制分开。一个功能部件的指令何时开始执行，有该功能部件的保留站控制，而记分牌则是集中控制。
5. 计算的结果通过相关专用通路直接从功能部件进入对应的保留站中进行缓冲，而不一定是写到寄存器。这个相关专用通路通过公共数据总线来实现（Common Data Bus, CDB）。所有等待这个结果的功能部件（指令）可以同时读取。与之相比，记分牌将结果首先写到寄存器，等待此结果的功能部件要通过竞争，在记分牌的控制下使用。
6. 具有分布的阻塞检测机制。
7. 消除了数据的写后写和先读后写相关导致的阻塞。

第五节　实验总结

1. 理论与实践结合，加深了对 tomasulo 算法的理解。
2. 对 tomasulo 算法的细节有了更详细的了解，回顾了 cpu 运行的时序问题。
3. 通过这次实验，我收获良多，感谢老师和助教给我这次学习和锻炼的机会。