Tomasulo 模拟器实验报告

李晨昊 2017011466

2020 年 5 月 4 日

目录

1 简介

在Linux平台上,执行make即可编译,要求编译器支持C++17。编译生成可执行文件tomasulo，它接受如下命令行参数:

```
$ ./tomasulo
Usage: ./tomasulo <nel file> <print state (0/1~32, number of registers to
    print)> <print log (0/1)>
```

第一个参数是输入的NEL文件的路径；第二个参数是 0，表示不输出每一步的状态，或者一个 1~32 间的整数n_reg，表示每执行一步，都输出保留站，Load Buffer和前n_reg个寄存器的状态 (之所以有这个设定，是因为输出所有寄存器的状态会占用太多空间，不方便阅读)；第三个参数是 0，表示运行结束后不输出 log，或者 1，表示输出 log。

运行结束后，它总会在stderr里输出总时钟周期数和耗时。

我注意到部分测例是 CRLF 的行尾格式 (具体来说是Fact.nel和Fabo.nel)，这不符合指导书给出的 CFG，因为按照定义，这里面的空字符不能随意忽略，每个字符都是有意义的。为了我的代码的简单 (主要是美观)，我在运行测例前将它们修改成了 LF 的格式，而不是修改我代码来兼容它们。如果助教在测试我的代码的时候遇到了错误，也应该尝试修改文件行尾格式，毕竟我只需要按照要求完成了文件的解析，任何要求之外的格式我都没有义务支持。

2 设计思路

2.1 NEL 文件解析

我的实现比较高效 (解析最长的Big_test.nel耗时 50ms 左右，解析速度在 2×10^7 条每秒的级别)，但同时也非常脆弱，如果输入不符合指导书给出的 CFG，我不对它的结果做任何保证，哪怕只是多了一个空格也不行。

我首先把文件mmap到内存中，然后直接操作字符指针。对于每条指令，先读到一个非字母的字符，然后判断这个范围内的单词，从而确定指令的操作符。根据不同的操作符类别读取后续的操作数。中间的辅助字符，如'R'，','和' '，我都是直接通过指针加法跳过，所以也碰巧支持了不符合规范的0.basic.nel。不过这只是一个意外，还是希望助教以后能够确保数据的合法性。

解析一条指令后，我直接把指针加 1 来跳过换行符，这也是我不能支持 CRLF 格式文件的原因。

2.2 流水线设计

模拟 Tomasulo 算法本质上就是在模拟一个三级的流水线：发射，执行和写回。对此我也分别定义了三个函数，它们的具体功能分别如下：

1. issue：发射一条指令到保留站 (或者Load Buffer，后面不再重复) 中，从寄存器中读值 (可能是有效的值或者保留站编号)，并将目标寄存器标记为等待这个保留站的结果。

2. exec：对于保留站中的指令，如果它已经开始执行，则把执行的剩余时间减 1；如果它已经就绪且有空余的功能部件，则让它开始执行。在我的设计中，没有专门表示功能单元的数据结构，必要的信息都记录在了保留站中。

3. write_back: 对于保留站中的指令，如果它执行的剩余时间为 0，则把它的结果广播到等待这个结果的寄存器和保留站中，并且将这个保留站标记为空闲。

接下来一个问题是应该按照什么顺序执行这三个函数。理论上如果没有数据的依赖关系，应该是可以以任意顺序执行的，只要保证这三个一组一起执行即可 (就是流水线时空图中的一列，如果没有数据依赖就可以以任意顺序执行)。需要考虑数据依赖的情况下，就要按照规定，看同一个周期内哪个阶段对公共数据结构的修改可以被其他阶段感知到。通过文档知道有以下几条规则：

1. 一个周期内发射的指令，在没有其他限制的情况下在本周期开始执行。
2. 如果遇见写回结果和发射指令冲突时 (其实这叫冲突并不是很合适，表达的意思是写回的寄存器恰好发射的指令需要读取)，默认先写回结果，后发射指令。
3. JUMP指令写回前不能发射指令，而写回的同一个周期就可以发射。

可以看出发射阶段可以感知到写回阶段的影响，执行阶段可以感知到发射阶段的影响，所以合适的顺序应该是：

```
bool step() {
  ++clock;
  write_back();
  issue();
  exec();
  ...
}
```

3 Tomasulo 算法分析

Tomasulo 算法中只要保留站和功能单元充足以及操作数就绪，总是可以进行发射，执行和写回。而记分牌算法会受限于 WAR 和 WAW 冲突，在事实上可以进行计算的时候进行无谓的等待。

Tomasulo 算法的核心在于寄存器重命名，一条指令被发射后，它的目标寄存器的信息并不被保存在保留站中，而是体现在寄存器组中对应位置对这个保留站的数据依赖。WAW 和 WAR 冲突分别是以如下的机制解决的：

- 保留站可以暂存一个寄存器中的数据，所以遇到 WAR 的情景时仍然可以向寄存器中写入，因为所需要的值已经被读出来保存起来了，不会像记分牌算法一样因为担心写入破坏了别人要读的值而延迟写入。

- 保留站和寄存器可以暂存一个未来就绪的数据 (保留站编号)，计算结果写回的时候会检查公共数据总线上的每一个接受处是不是正在等待自己的数据，不会在等待别人的数据的时候写入自己的数据，所以不会像记分牌算法一样因为担心 WAW 导致写入数据不正确而推迟指令的发射。

4 拓展

虽然可以说我做了两个拓展，但是我认为我做的都不算深入，这里都列出来，希望助教可以按照认为价值更高的一个来评分。

4.1 支持 JUMP

我实现了JUMP指令，但是没有做任何的分支预测。
相比于其他指令，JUMP指令的特别之处在于:

1. 它不会写入寄存器，所以将它放入保留站时，不会影响任何寄存器的状态。
2. 它存在于保留站中时，不能发射任何指令 (因为我没有实现分支预测)。
3. 它的写回阶段不会在公共数据总线上发送数据，而是修改当前的pc。
4. 它暂存在加法保留站中，只支持加减法的加法保留站只有保留两个操作数的空间，而JUMP需要三个记录三个操作数 (两个判断相等的操作数，一个跳转偏移量)，所以必须额外添加一个字段来保存多出来的操作数。虽然保留站中也有记录指令的编号 (为了输出 log)，但是在JUMP的时候再读一次指令并不现实，真实的硬件不太可能这样做，而应该在第一次读指令的时候把全部信息都记录在保留站里。

根据样例，必须在JUMP指令的写回的周期就发射下一条指令，而不能等到下一个周期。这一点也在之前的章节中论述了，造成的影响就是模拟执行一步的时候，写回阶段必须在发射阶段的前面。

还有一点需要注意的是，NEL规定的JUMP的语义是从以JUMP这一条语句为基准进行跳转，其实这与大多数指令集都不一样，一般都是以JUMP语句的下一条语句为基准进行跳转。因为在我的实现中，取了指令之后就会递增pc，所以实际跳转的时候需要把偏移量减 1。

4.2 性能优化

其实我也没有刻意地做什么优化，对性能的追求是我写任何程序时都不会放下的，已经成为了我的代码风格的一部分。性能测试的结果如下，我个人觉得还是可以接受的，从比较

0.basic.nel	0.01ms
1.basic.nel	0.03ms
2.basic.nel	0.02ms
3.basic.nel	0.02ms
4.basic.nel	0.02ms
Big_test.nel	93.57ms
Example.nel	0.01ms
Fabo.nel	0.02ms
Fact.nel	0.05ms
Gcd.nel	2295.20ms
Mul.nel	0.64ms

大的几个测例来看每秒执行的指令数在 $10^7 \sim 3 \times 10^7$ 量级，但是因为缺乏和别人的比较，我也不清楚它到底处于什么水平，希望不会贻笑大方：

第一是我广泛地应用了 tagged-union 来定义数据结构，这当然是明显地受到了Rust的影响才养成的编程风格。例如我的指令 (Inst) 的定义是这样的，注释中的指令类型表示op是对应类型的时候，对应字段有效：

```
struct Inst {
  Op op;
  union {
    struct {
      u32 dst;
      union {
        struct { u32 l, r; }; // Add, Sub, Mul, Div
        struct { u32 imm; }; // Ld
      };
    };
    struct { u32 cond, cmp, offset; }; // Jump
  };
  ...
}
```

很多人喜欢 OOP 那一套，比如定义基类Inst，再定义BinaryInst，LdInst继承它之类的。但是在这种必须了解全部细节才能合理地操作数据结构的情境中，OOP 那一套带来的可拓展性完全无法体现，因为事实上没有办法用一组抽象的操作来定义一条指令，然后在后续的执行中只依赖于这些操作。

避开了 OOP 那一套，也就避开了一般情况下它所要求的动态内存申请，所有数据可以连续地排布在一段内存中。这样既能避免动态内存的额外元数据开销，从而降低内存使用，而且也对 Cache 非常友好，可以提高执行速度。

同样的思路可以用于定义保留站的 $V_{\{j,k\}}$ 和 $Q_{\{j,k\}}$。因为 V 和 Q 中同时只有一个值是有效的，所以不需要用两个u32来表示它们，但是如果用一组bool和u32，因为内存对齐的关系还是会消耗一样多的空间，我的定义是：

```
bool is_q[2]; // `is_q[i] == true` <=> `qv[i] means q[i]`, for i in {0, 1}
u32 qv[2];
```

第二是一些算法上的优化。考虑到保留站，Load Buffer等部件的数目都相当少，如果使用理论上渐进复杂度更小，但是常数更大的算法，耗时很可能还不如看起来更暴力的算法。在真实的硬件里面也往往是这样的情形。

例如在给保留站中就绪的指令分配功能部件时，不能在遍历一遍保留站的同时给就绪的指令功能部件，因为需要考虑同时就绪的编号更小的指令，它们才是应该优先分配功能部件的。理论上渐进复杂度比较好的做法是筛选出所有就绪的指令，按照编号大小选出前剩余的功能部件数个，给它们分配。这个过程 (可以调用std::nth_element来解决) 的渐进复杂度是线性的，但是实际测试发现这样做相当慢，这个操作会在整个时间开销中成为瓶颈。我现在采用的解决方案是遍历一遍保留站，对于每个就绪的指令，统计就绪且编号更小的指令，如果总数小于总功能部件数，就可以为它分配。这样做的渐进复杂度是平方级别的，但是因为最多的保留站数目 (加法保留站) 只有 6，而且这样的常数远小于前一种做法，所以实际性能表现反而更好。

5 测试用例分析

以下是我自己设计的测试用例，在fast_mul.nel中。其目的是模拟竖式乘法，如果我们支持位运算的话就可以不用MUL和DIV而实现MUL的功能。下面的注释是我在报告中手动加上去的，我的解析器不支持这样的语法：

```
LD,R0,0x0 // R0统计结果，计算R0 = R1 * R2
LD,R1,0x123
LD,R2,0x456
LD,R3,0x2 // R3为常数2
LD,R6,0x0 // R6为常数0
JUMP,0x0,R2,0x9 // R2 == 0时运算结束
DIV,R4,R2,R3 // R4 = R2 / 2
MUL,R5,R4,R3 // R5 = R4 * 2
```

```
SUB,R5,R2,R5 // R5 = R2 % 2
JUMP,0x0,R5,0x2 // if (R2 % 2 != 0) { R0 = R0 + R1; }
ADD,R0,R0,R1
ADD,R1,R1,R1 // R1 = R1 * 2;
ADD,R2,R4,R6 // R2 = R2 / 2;
JUMP,0x0,R7,0xFFFFFFF8 // 总是跳回判断R2 == 0处
```

运行结束时可以正确得到`R0 = 0x4EDC2 = 0x123 * 0x456`。

循环体中对`R5`的连续操作的两句，在 Tomasulo 算法中不会引起暂停，会连续地发射出去 (当然执行的时候还是要等待前一条把`R5`的值计算出来，这是本质的 RAW 数据依赖)，这样的 WAW 冲突可以通过保留站顺利解决。