Tomasulo模拟器PA

陶天骅 2017010255 计81

文件和编译方法

提供了一个 Makefile，可以使用 `make` 或者 `g++ main.cpp Tomasulo.cpp Logger.cpp Instruction.cpp -o Tomasulo -O3 -std=c++11` 编译。

编译完成后，可以 `./Tomasulo [-v] input.nel` 运行某个测例，`-v` 是打印每个周期的情况。或者也可以使用我提供的脚本运行所有的测例（9个作业测例，2个我自己写的）。

```
./run_all.sh
```

完成后，会在 `./Log` 下有所有的需要提交的 log 文件（作业打包的时候已经有了），在 `./detailed_log` 下会有更详细的每个 cycle 的内部 log，内容可能如下：

```
Cycle 31 (PC=17)
Reservation Station
-------------------------------------------------------------------------------
|         |  Busy  |   Op   |     Vj     |     Vk     |    Qj    |    Qk    |
-------------------------------------------------------------------------------
|  Ars 1  |   *    |  SUB   |            | 0x00000001 |   MRS1   |          |
|  Ars 2  |   *    |  ADD   | 0x00000001 | 0x00000000 |          |          |
|  Ars 3  |        |        |            |            |          |          |
|  Ars 4  |        |        |            |            |          |          |
|  Ars 5  |        |        |            |            |          |          |
|  Ars 6  |        |        |            |            |          |          |
|  Mrs 1  |   *    |  MUL   | 0x00000001 | 0x00000000 |          |          |
|  Mrs 2  |   *    |  MUL   | 0x00000003 |            |          |   MRS1   |
|  Mrs 3  |   *    |  MUL   |            | 0x00000000 |   ARS2   |          |
-------------------------------------------------------------------------------

Load Buffer
-----------------------------------------------------------------
|                    |        Busy        |      Address       |
-----------------------------------------------------------------
|        LB 1        |                    |                    |
|        LB 2        |                    |                    |
|        LB 3        |                    |                    |
-----------------------------------------------------------------

Registers
-----------------------------------------------------------------
|                    |       State        |       Value        |
```

```
-----------------------------------------------------------------
|         R0          |        ARS1         |                   |
|         R1          |                     |     0x00000000    |
|         R2          |        MRS3         |                   |
|         R3          |                     |     0x00000000    |
|         R4          |                     |     0x00000001    |
|         R5          |                     |     0x00000000    |
|         R6          |        MRS2         |                   |
|         R7          |                     |     0x00000000    |
...... (more) ......
|        R31          |                     |     0x00000000    |
-----------------------------------------------------------------

Arithmetic Component
-------------------------------------------------------------------------------
|          |      Current       |          RS          |   Cycles remain    |
-------------------------------------------------------------------------------
|  Add 1   |        ADD         |        ARS2          |          2         |
|  Add 2   |                    |                      |                    |
|  Add 3   |                    |                      |                    |
|  Mult 1  |        MUL         |        MRS1          |          3         |
|  Mult 2  |                    |                      |                    |
|  Load 1  |                    |                      |                    |
|  Load 2  |                    |                      |                    |
-------------------------------------------------------------------------------
```

`Gcd.nel` 、`Big_test.nel` 、`Mul.nel` 的 log 太大，所以没有输出。

实现方法

有如下类：

```
class Tomasulo;                         // 主类
  struct LoadBuffer;                    // Load 缓存
  struct RegisterStatus;                // 寄存器
  struct ReservationStation;            // 保留站
  struct ArithmeticComponentStatus;     // 运算部件
class Logger;                           // 负责格式化输出和记录，可以打印漂亮的表格
class Instruction;                      // 一个指令的包装类，包含读文件，处理字符串，返
回序列
  enum class Operation;                 // 操作
  class Operand;                        // 操作数
```

为了保存状态，类内有很多的字段，比如 `ReservationStation`

```
struct ReservationStation {
  bool busy[TOTAL_STATION_COUNT];
  Instruction::Operation op[TOTAL_STATION_COUNT];
  int tagSource0[TOTAL_STATION_COUNT];        // Qj
  int tagSource1[TOTAL_STATION_COUNT];        // Qk
  int source0[TOTAL_STATION_COUNT];           // Vj
  int source1[TOTAL_STATION_COUNT];           // Vk
  bool executing[TOTAL_STATION_COUNT];
  int pc[TOTAL_STATION_COUNT];
  int issueCycle[TOTAL_STATION_COUNT];
  int readyTime[TOTAL_STATION_COUNT];
};
```

Tomasulo 是主逻辑类，有 `run()` 方法，具体可以看注释

```
/**
 * @brief Tomasulo 的主方法
 * 分为以下阶段：Write Result, Issue, Execute, RS update, Load Buffer update
 * 其中重要的是 Write Result 要最先，这样才可以释放部件，并且其他指令会需要 Write Result
的结果
 * Issue 要在 RS update 和 Load Buffer update之前，
 * 因为有的指令Issue完之后直接就可以进入 RS 和 LB 了
 * 其他的顺序没有太大的关系
 *
 * 详细内容：
 * Write Result --  取 arithmeticComponentStatus 中完成的指令，如果是JUMP，释放保留
站，
 *                  修改pc；如果是其他的，释放保留站，计算结果，并广播
 * Issue --         进入保留站或者是 Load Buffer（上个周期执行完指令的已经从中清除了）
 *                  如果是 JUMP 的话，要设置 jumpIssued，暂停后续发射
 *                  由于 JUMP 会暂停发射了，因此不会有多个 JUMP 都发射了的情况
 * Execute --       递减剩余周期数，并做 log
 * RS update --     取就绪的指令，进入运算部件，操作的时候如下：
 *                  把所有就绪的行序号放进一个 vector，按照 ready time 然后 issue
time
 *                  排序，按顺序进入运算部件
 * Load Buffer update --    和 RS 相同
 * @param step 原本是用来单步运行的，实际没用用到
 */
void Tomasulo::run(bool step);
```

拓展内容

实现 Jump 指令

有基本的 Jump，没有分支预测。

实现的时候和 ADD 类似对待，但某个 JUMP 发射后，会暂停后续的发射，直到 JUMP 的结果出来。符合跳转条件时，修改 pc 的值。

性能测试

没有使用特别的优化技巧，但是在程序中设置了很多状态字段。

测试环境: macOS 10.15.4, i7 3.1 GHz，单线程，clang++ -O3，运行时有 log，但关闭 stdout 输出。

```
% time ./Tomasulo TestCase/Gcd.nel
./Tomasulo TestCase/Gcd.nel  3.80s user 0.02s system 99% cpu 3.854 total
% time ./Tomasulo TestCase/Big_test.nel
./Tomasulo TestCase/Big_test.nel  1.62s user 0.14s system 98% cpu 1.786 total
% time ./Tomasulo TestCase/Mul.nel
./Tomasulo TestCase/Mul.nel  0.00s user 0.00s system 41% cpu 0.013 total
```

输入	时间
Gcd.nel	3.854s
Big_test.nel	1.786s
Mul.nel	0.013s

编写测例nel文件

我编写了两个nel文件，他们的功能都是求1到10的和，并将结果存在R1中。

`sum.nel` 是使用 JUMP 的版本

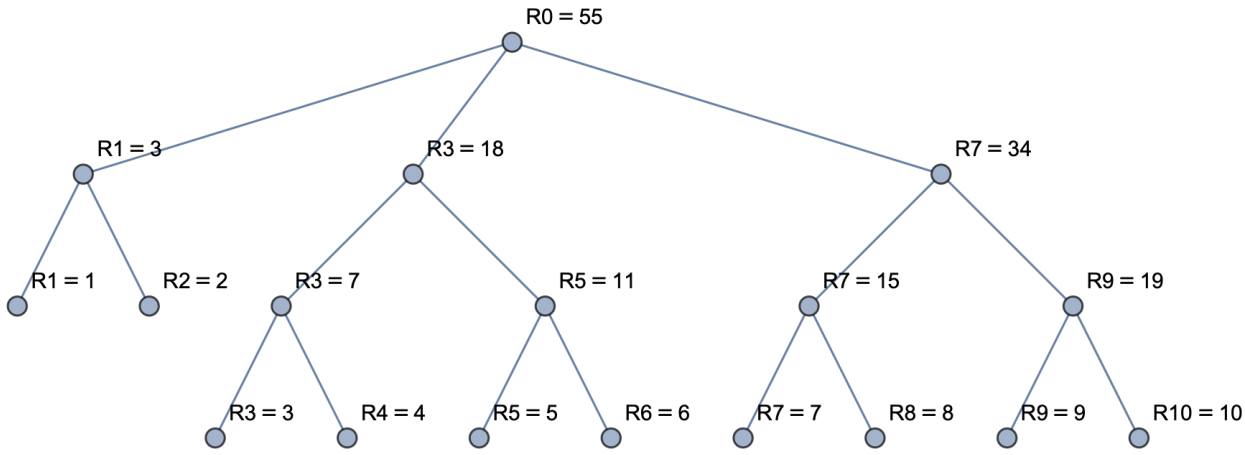
```
LD,R0,0xA
LD,R1,0x0
LD,R2,0x0
LD,R3,0x1
JUMP,0x00,R0,0x4
ADD,R1,R0,R1
SUB,R0,R0,R3
JUMP,0x00,R2,0xFFFFFFFD
```

`sum_no_jump.nel` 是不使用 JUMP 的版本

```
LD,R0,0x0
LD,R1,0x1
LD,R2,0x2
ADD,R1,R1,R2
LD,R3,0x3
LD,R4,0x4
ADD,R3,R3,R4
LD,R5,0x5
```

```
LD,R6,0x6
ADD,R5,R5,R6
LD,R7,0x7
LD,R8,0x8
ADD,R7,R7,R8
LD,R9,0x9
LD,R10,0xA
ADD,R9,R9,R10
ADD,R0,R0,R1
ADD,R3,R3,R5
ADD,R7,R7,R9
ADD,R0,R0,R3
ADD,R0,R0,R7
```

`sum_no_jump.nel` 按照如下图顺序求和，即累加 R1 到 R10 的值。



最后检查R0的值，应当为 55 即 $0x37$，正确：

```
Registers (sum_no_jump.nel)
-----------------------------------------------------------------
|                    |       State        |       Value        |
-----------------------------------------------------------------
|         R0         |                    |     0x00000037     |
|         R1         |                    |     0x00000003     |
|         R2         |                    |     0x00000002     |
|         R3         |                    |     0x00000012     |
|         R4         |                    |     0x00000004     |
|         R5         |                    |     0x0000000b     |
|         R6         |                    |     0x00000006     |
|         R7         |                    |     0x00000022     |
|         R8         |                    |     0x00000008     |
|         R9         |                    |     0x00000013     |
|        R10         |                    |     0x0000000a     |
-----------------------------------------------------------------

Registers (sum.nel)
-----------------------------------------------------------------
```

```
|                  |       State        |       Value        |
-------------------------------------------------------------
|         R0       |                    |     0x00000037     |
|         R1       |                    |     0x00000000     |
|         R2       |                    |     0x00000000     |
|         R3       |                    |     0x00000001     |
|         R4       |                    |     0x00000000     |
|         R5       |                    |     0x00000000     |
|         R6       |                    |     0x00000000     |
|         R7       |                    |     0x00000000     |
|         R8       |                    |     0x00000000     |
|         R9       |                    |     0x00000000     |
|        R10       |                    |     0x00000000     |
-------------------------------------------------------------
```

难点和讨论

1. 关于一个周期内，发射、就绪、写会、开始执行的顺序问题。经过分析，指令 s 写回的时候，s 的保留站可以被后续指令进入；一个指令可以在同一个周期发射并就绪，或者说可以在发射的周期就进入运算部件；指令 $s1$ 在某个周期 c 把值 v 写回，那么某个等待 v 的指令 $s2$ 在周期 c 就可以进入运算部件，不用等到周期 $c+1$.因此，一个周期内的操作顺序为 Write Result, Issue, Execute, RS update, Load Buffer update 可以满足要求。
2. 保留站中就绪的指令按照最先就绪时间，然后是发射时间排序，按顺序进入运算部件。实现的时候操作的时候如下：

   把所有就绪的行序号放进一个 vector，按照 ready time 然后 issue time排序，按顺序进入运算部件：

   ```
   std::sort(ready_rs_id.begin(), ready_rs_id.end(), [&](int lhs, int rhs) {
     if (reservationStation.readyTime[lhs] !=
   reservationStation.readyTime[rhs])
       return reservationStation.readyTime[lhs] <
   reservationStation.readyTime[rhs];
     else
       return reservationStation.issueCycle[lhs] <
   reservationStation.issueCycle[rhs];
   });
   ```

   （用 lambda 很方便）
3. 通过观察运算部件的使用情况，发现 RAW 的情况是并行度下降的主要原因。此时执行指令必须要等待计算结果出来。

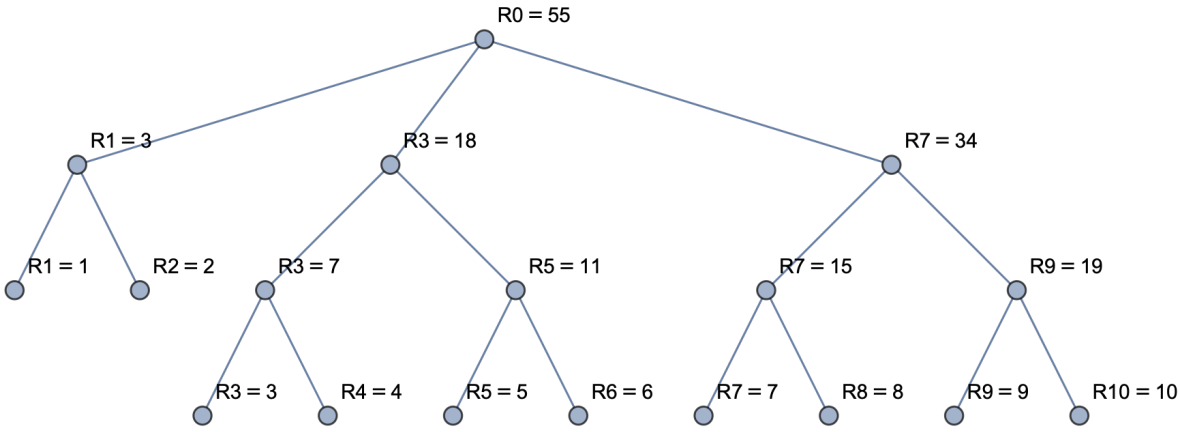
   为了使并行性达到更大，代码应该尽量避免 RAW，考虑如下的两种程序，他们的作用都是求1到10的和，但是求值顺序不一样。

   ### sum_no_jump.nel

   ```
   LD,R0,0x0
   ```

```
LD,R1,0x1
LD,R2,0x2
ADD,R1,R1,R2
LD,R3,0x3
LD,R4,0x4
ADD,R3,R3,R4
LD,R5,0x5
LD,R6,0x6
ADD,R5,R5,R6
LD,R7,0x7
LD,R8,0x8
ADD,R7,R7,R8
LD,R9,0x9
LD,R10,0xA
ADD,R9,R9,R10
ADD,R0,R0,R1
ADD,R3,R3,R5
ADD,R7,R7,R9
ADD,R0,R0,R3
ADD,R0,R0,R7
```

他就是上面所示的测例之一，对应求值顺序

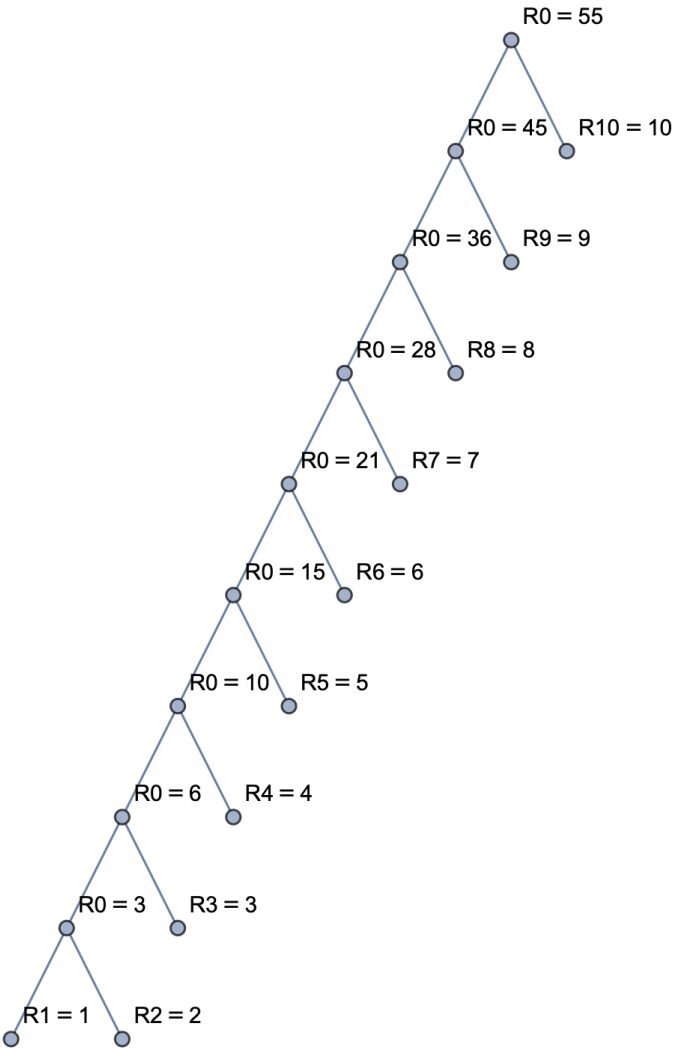


sum_no_jump_slow.nel

```
LD,R0,0x0
LD,R1,0x1
LD,R2,0x2
LD,R3,0x3
LD,R4,0x4
LD,R5,0x5
LD,R6,0x6
LD,R7,0x7
LD,R8,0x8
LD,R9,0x9
LD,R10,0xA
ADD,R0,R1,R2
ADD,R0,R0,R3
```

```
ADD,R0,R0,R4
ADD,R0,R0,R5
ADD,R0,R0,R6
ADD,R0,R0,R7
ADD,R0,R0,R8
ADD,R0,R0,R9
ADD,R0,R0,R10
```

功能一样，对应求值顺序



`sum_no_jump.nel` 运行需要 37 个周期，`sum_no_jump_slow.nel` 需要 55 个周期。可见代码顺序不同对性能有影响。第2种算法反复使用了 R0 的值，导致不能高效并行。

4. JUMP 会使发射暂停，因为没有做分支预测，所以大量 JUMP 会使性能变差。比如`sum.nel`使用 JUMP，要运行 79 周期，`sum_no_jump.nel` 运行只需要 37 个周期。
5. 运算部件计算完结果，需要对外广播。比如需要把所有标为 Ars 1 的地方都变为 `0x01`，实现的方法是：对每一个标记，都使用一个 int 作为 tag，高16位为部件编号（RS，LB），低16位为行号，然后对接收广播的对象加入`void receiveBroadcast(int tag, int value)`方法。
6. Tomasulo 算法比起计分牌算法好在可以避免 structural hazard，并且化解 WAR 和 WAW 依赖。