Tomasulo 实验报告

实验人：陈熠豪

学号：2017011486

邮箱：chenyiha17@mails.tsinghua.edu.cn

代码实现

- Instruction 类
  - 解析和表示指令（op、dst、src1,、src2 等）
  - 记录了指令的行号以及运行时的唯一 id
  - 记录发射周期、执行结束周期、写回周期
- RS_Entry 类
  - 表示保留站内的一条 entry，记录了 Vj、Vk、Qj、Qk 以及 Address 等
- Reservation_Station 类
  - 组合 RS_Entry，形成了各个保留站，包括 loadBuffer、加减法保留站、乘除法保留站
  - 模拟和记录了 32 个 register
  - 实现了包含 jump 指令的模拟运行过程：在每一个 cycle 内依次进行发射指令、写回（上一个 cycle 运行完毕的指令）、运行（已经就绪的指令）和检查就绪（源数据是否写回，是否有空闲 FU）等过程
  - jump 的实现思路：对 jump 指令的处理与其他指令无异，但是在发射时需要记录发射时的指令地址并暂停其他指令的发射（由于没有实现分支预测），在执行结束的周期内确定是否进行跳转，并得到新的指令地址，修改全局的指令寄存器，从而实现执行流的跳转，并回复其他指令的发射

Tomasulo 算法特点

- Tomasulo 采用的是寄存器重命名的方法来消除寄存器数据流之间的假相关：在指令被发射而源数据还未准备好时，先在保留站内记录了源数据的来源指令，实际上就相当于进行了寄存器重命名，而当来源指令执行完毕写回时，再检查所有保留站，将等待结果的源数据给更新了
- 和记分牌算法的差异：
  - Tomasulo 算法通过各个功能单元的保留站来记录和检测冲突，是分布式的，而记分牌算法通过统一的记分牌来管理，是集中式的。（可能分布式的方法便于拓展，而集中式方法效率更高）
  - Tomasulo 算法通过寄存器重命名可以消除 WAR 冲突和 WAW 冲突，但记分牌无法消除
  - Tomasulo 算法通过 CBD 直接访问和修改保留站内的结果，而不必通过寄存器来传递，而记分牌算法通过 FP 寄存器来写结果，可能在这一步出现冲突

实验

- 自己设计的 NEL 测试用例 ./TestCase/myCase.nel

```
LD,R0,0x0000FFFF
ADD,R0,R0,0xFFFFFFFF
JUMP,0x0,R0,0x2
JUMP,0x0,R1,0xFFFFFFFE
```

单纯地进行循环，直到退出

- 运行以下命令测试所有用例。该脚本将会执行 make，并运行 Basic、Extend 的测试用例得到 Log，并运行 Performance 得到运行时间。所有运行时间将存于./Log/time_elapse.json

```
./run.py
```

- 修改./src/macro.h或./Makefile，将 STEP 宏置为 1 后重新 make，则可以逐步运行并查看瞬时状态

  make 生成./bin/tomasulo，执行参数如下：

```
./bin/tomasulo --input_file {INPUT_FILE} [--output_path {OUTPUT_PATH}]
```

  其中 INPUT_FILE 为输入文件路径，OUTPUT_PATH 为输出 Log 路径。如果不指定 output_path，则不会输出 Log

- **注意: 输出的 Log 为所有运行到的指令的首次运行结果，并按照指令在测试用例中的行号排序，但没有运行到的指令将不会出现在 Log 里**

结果

- 性能表现 Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz

TestCase	elapse_time (ms)
0.basic	0.460749
1.basic	1.81993
2.basic	1.40684
3.basic	1.51634
4.basic	2.34449
Fact	1.23715
Fabo	0.81185
Example	0.93785
Gcd	8609.26
myCase	93.3841

TestCase	elapse_time (ms)
Mul	37.5535
Big_test	35969.2