实验环境和个人信息

个人信息

杨雅儒, 2017011071, 计85。

实验环境

Ubuntu 20.04, gcc version 9.3.0, cmake version 3.17.2, GNU Make 4.2.1。

运行方法

运行所有例子（由于没有做 performance，为了方便就不运行 Big_test.nel 了）：

```
bash run_all.sh
```

运行单个文件：

```
bash run.sh TestCase/<name>.nel 0/1
```

运行结果将保存在 logs 目录下。

另外**最后一个参数为 1 时将显示每一时刻的瞬时状态**，格式例如：

```
正在执行 pc 34, cycle 57...
打印 cycle 57 的瞬时状态...
Load Buffer:
0:: Busy: False  Address: 2
1:: Busy: False  Address: 1
2:: Busy: False  Address: 3

Units:
unitAdd1:: instId: 31  countdown: 3

Reservation Station:
Ars1:: Busy: False  Op: ADD  Vj: 0  Vk: 0  Qj: None  Qk: None
Ars2:: Busy: False  Op: ADD  Vj: 1  Vk: 0  Qj: None  Qk: None
Ars3:: Busy: False  Op: SUB  Vj: 0  Vk: 0  Qj: None  Qk: None
Ars4:: Busy: False  Op: ADD  Vj: 0  Vk: 0  Qj: None  Qk: None
Ars5:: Busy: True  Op: ADD  Vj: 0  Vk: 0  Qj: None  Qk: None
Ars6:: Busy: False  Op: ADD  Vj: 0  Vk: 0  Qj: None  Qk: None
Mrs1:: Busy: False  Op: None  Vj: 0  Vk: 0  Qj: None  Qk: None
Mrs2:: Busy: False  Op: None  Vj: 0  Vk: 0  Qj: None  Qk: None
Mrs3:: Busy: False  Op: None  Vj: 0  Vk: 0  Qj: None  Qk: None

Register:
0:: value: 0
1:: value: 1
2:: value: 1
3:: value: 0
4:: value: 0
```

```
5:: value: 0
6:: value: 0
7:: value: 0
8:: value: 0
9:: value: 0
10:: value: 0
11:: value: 0
12:: value: 0
13:: value: 0
14:: value: 0
15:: value: 0
16:: value: 0
17:: value: 0
18:: value: 0
19:: value: 0
20:: value: 0
21:: value: 0
22:: value: 0
23:: value: 0
24:: value: 0
25:: value: 0
26:: value: 0
27:: value: 0
28:: value: 0
29:: value: 0
30:: value: 0
31:: value: 0
```

设计

注意事项

- 首先要注意除法在除零的时候只需要 1 个周期，而正常情况下需要 4 个周期；
- 当两条指令同时就绪时，**取时间上更靠前的指令先进入功能部件执行**，注意如果有两个对应功能部件同时空闲，则可以同时分别进入执行——这一点依靠链表 rsQue 来实现；
- 当两条指令同时执行完毕时，可同时写回（尽管这一点我认为使用公共总线的话这样不大合理，但例子中确实是这么给的，为统一就按照例子中的来）；
- 同一周期中不同部分重叠时，依据例子，按照 `写回->取消占用保留站->发射并占用保留站` 的顺序来处理（似乎也不大合理）；
- 存在已发射的 JUMP 指令时不继续进行发射。

指令

定义结构体 Instruction 表示指令，type 表示指令类型, 整型变量 op1, op2, op3 分为表示三个操作数，如果某操作数为 REGISTER，则存储其编号，若为 INTEGER 则直接存储其值。类型分类如下：

- ADD 类型；
- SUB 类型；
- MUL 类型；
- DIV 类型；
- LOAD 类型；
- JUMP 类型。

指令使用一个 `vector<Instruction> instructions` 存储下来，下标从 0 开始，pc 为下一个即将发射的指令下标。

保留站

为了方便统一，这里将课件中的保留站和 Load Buffer 统一认为为广义的保留站，而将狭义的保留站命名为 Functional Buffer。

- ReservationStation 基类：存储保留站类型 type 以及是否忙碌 isBusy；
- FunctionalBuffer 派生类：用于加法、乘法等功能部件的保留站，存储 Instruction::Type 类型变量 op，整型变量 vj 和 vk 表示两个源操作数的值，ReservationStation* 类型变量 qj 和 qk 表示两个源操作数的来源保留站，注意 v/q 只有一个会生效，q 为 NULL 表示已准备好，此时应当用 v；
- LoadBuffer 派生类：用于 Load 的保留站，存储整型变量 addr 表示要 load 的值。

存储链表 list<ReservationStation*> rsQue，用于实现先发射的先执行。

寄存器状态

每个寄存器用一个结构体 RegisterState 存储，包含：

- 整型变量 value 表示寄存器的值；
- ReservationStation* 类型变量 state 表示当前已经发射的指令中，时间最靠后的写入当前寄存器的指令对应的保留站，若为 NULL 表示当前发射的指令中没有要向该寄存器写入的。

运算部件状态

每个运算部件（包括功能运算部件和Load运算部件）用一个结构体 UnitState 存储，包含：

- 整型变量 instId 表示该运算部件当前正在执行的指令下标；
- 整型变量 countdown 表示剩余周期数；
- ReservationStation* 类型变量 rs 表示对应的保留站（NULL 表示该部件空闲）。

不同类型的运算部件使用不同的数组来区分即可，例如 unitAdd[3], unitMult[2], unitLoad[2]。

控制逻辑

用 Tomasulo 类来控制整个模拟过程。

每个周期的整体步骤分为：

- 尝试写回；
- 尝试发射指令；
- 尝试开始执行已经就绪的指令。

尝试写回

- 枚举每一个运算部件，若正在运行且 countdown > 0 则进行递减操作；
- 若正在运行且 countdown == 0 则进行写回——分不同类型进行操作即可。

尝试发射指令

- 判断是否可发射指令——包括判断是否有 jump 指令，pc 是否在范围内，以及是否有空闲的对应保留站，不可发射则不进行后续判断；
- 对对应的保留站进行修改，并且修改对应目标寄存器的状态，并且将该保留站加入队列 rsQue。

尝试开始执行已经就绪的指令

- 从前到后枚举 rsQue，判断是否就绪，就绪且有空余对应部件，则加入其中。

JUMP 设计思路

这里的设计并没有考虑分支预测。

实际上是一个比较简单的操作，JUMP 与 ADD 操作使用同样类型的运算器，在发射一个 JUMP 之后一直 stall 直到该 JUMP 写回——因为这样才能找到新的正确 pc 值并进行运算。

另外 JUMP 的两个 int 型变量，为了节省功能部件的硬件存储消耗，我便将其存储在全局中了（这也是得益于每个时刻只会有一个 JUMP 指令处于已经被发射而未写回的状态）。

特别地，在 JUMP 写回时，应当向 pc 进行写操作，而非寄存器等。

我的 nel 设计

这里只进行了简单设计，做了一个累加器，从 1 累加至 100，如下：

```
LD,R0,0x0
LD,R1,0x1
LD,R2,0x1
LD,R3,0x65
LD,R30,0x0
ADD,R0,R0,R1
ADD,R1,R1,R2
SUB,R4,R3,R1
JUMP,0x0,R4,0x2
JUMP,0x0,R30,0xFFFFFFFD
LD,R30,0x0
```

输出如下：

```
1 4 5
2 5 6
3 8 9
5 9 10
6 12 13
7 10 11
8 12 13
9 16 17
10 18 19
19 20 21
1207 1210 1211
```

思考与分析——Tomasulo 与记分牌算法

Tomasulo

非常重要的一个特点便是使用了**寄存器换名**，并在此基础上消除了 WAR 冲突以及 WAW 冲突。

寄存器换名实际上就是对于每一个功能部件都增加一定的硬件存储空间来用于存储，例如一个 WAW 的例子：

```
DIV   F0, F1, F2
MUL   F5, F0, F6
ADD   F0, F3, F4
```

假设 ADD 指令先执行完毕，这时向 F0 中写入了，但并不会影响 MUL 操作的正确性——因为 MUL 操作中的 F0 并不来自寄存器，而是直接来自 DIV 指令的广播。

记分牌算法

记分牌算法便没有采用寄存器换名，统一而非分布式地进行处理，仍然看这个例子：

```
DIV   F0, F1, F2
MUL   F5, F0, F6
ADD   F0, F3, F4
```

在记分牌算法中，由于没有采用换名操作，在碰到 WAW 和 WAR 冲突时所做的操作便是停下来等待。