Tomasulo Algorithm——一次复健

1. 动态调度的思想

简单流水线的一个限制是它们总是按照顺序发射、按照顺序执行。但是，如果指令 j 依赖于长时间运行的指令 i，那么 j 之后的指令必须停顿，直到 i 完成为止，无论它们的依赖性为何。

```
DIV.D       F0,F2,F4
ADD.D       F10,F0,F8
SUB.D       F12,F8,F14
```

考虑这个例子，`SUB.D` 和流水线中的任何指令都没有数据相关性，明明可以提前执行。为了能够执行这个例子，我们需要将发射过程分为两个部分：检查所有结构冒险+等待数据冒险的消失。因此，即便我们仍然使用顺序来发射指令，一条指令却在操作数都准备好的时候就可以开始执行。这样一种流水线实际上是乱序执行的。

回顾我们在流水线那一次中讨论的内容：流水线只会有 RAW 冒险。然而，乱序执行可能会导致 WAW, WAR 等冒险。考虑以下代码序列：

```
DIV.D       F0,F2,F4
ADD.D       F6,F0,F8
SUB.D       F8,F10,F14
MUL.D       F6,F10,F8
```

如果我们先执行 `SUB.D` 再执行 `ADD.D`，那么很明显会发生 WAR 冒险。与之类似，如果 `MUL.D` 先于 `ADD.D` 执行，那么就会出现 WAW。

note: 动态调度的异常处理通常较难，感兴趣的可以参考 CAAQA，Chapter 3.6 和 appendix J

2. 使用 Tomasulo 算法进行动态调度

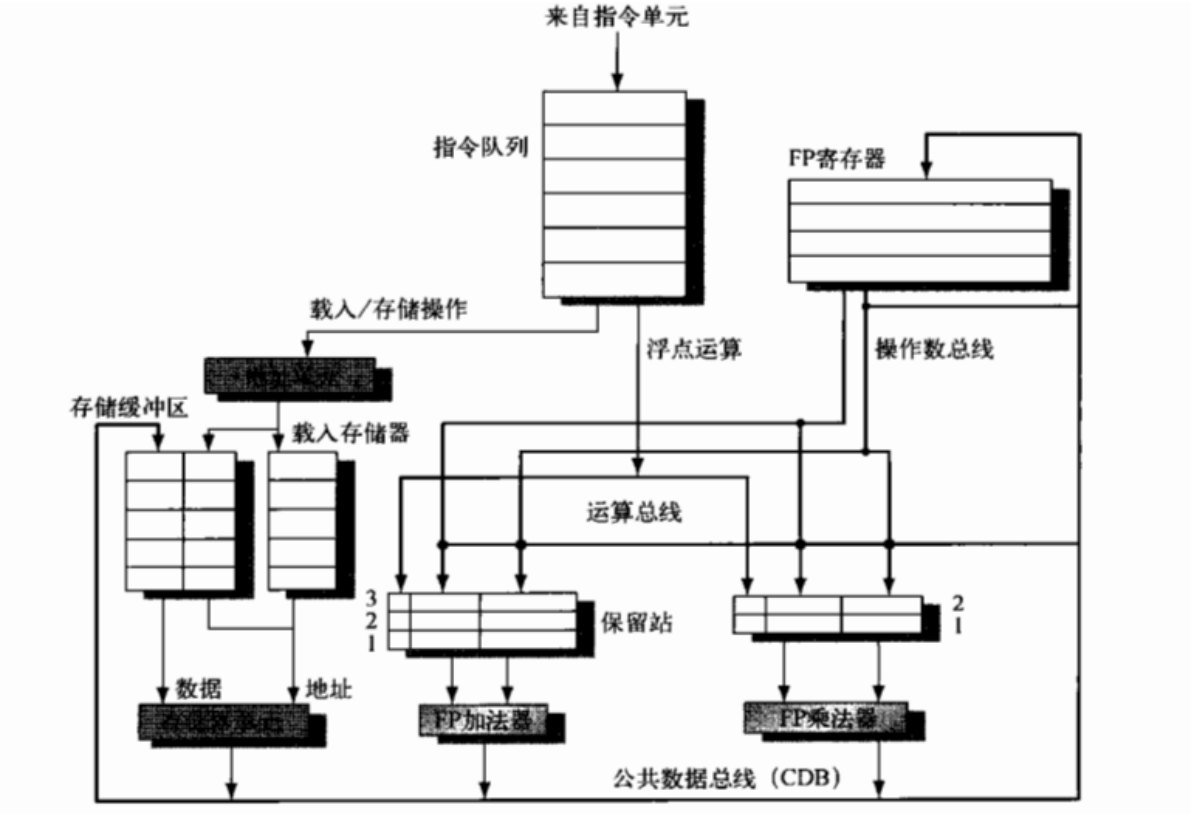
Tomasulo 主要跟踪指令的操作数何时可用，将 RAW 冒险降到最低，并引入寄存器重命名功能，将 WAW 和 WAR 冒险降到最低。我们考虑下面这个例子：

```
DIV.D       F0,F2,F4
ADD.D       F6,F0,F8
S.D         F6,0(R1)
SUB.D       F8,F10,F14
MUL.D       F6,F10,F8
```

以上代码共有两处反相关：ADD.D 与 SUB.D 之间，S.D 和 MUL.D 之间。在 ADD.D 和 MUL.D 之间还有一处输出相关，从而一共可能存在 3 处冒险：ADD.D 使用 F8 和 SUB.D 使用 F6 时的 WAR 冒险，以及因为 ADD.D 可能在 MUL.D 之后完成所造成的 WAW 冒险。还有 3 个真正的数据相关：DIV.D 和 ADD.D 之间、SUB.D 和 MUL.D 之间、ADD.D 和 S.D 之间。

我们假定存在临时寄存器 S 和 T，我们用它们来进行重命名。这是由保留站来完成的。保留站再一个操作数可用的时候马上提取它，此外，等待执行的指令会指定保留站，来为自己提供输入（如果保留站中的操作数已经算好了），最后，在对寄存器连续进行多次写入的时候，只会实际使用最后一个操作来更新寄存器。在发射指令的时候，我们将待用操作数的寄存器使用重命名，变为保留站中的名字。

例如刚刚这个例子。在 F8 被写入之后，我们就用 T 来进行重命名，来表示接下来的操作都是在对 T 进行。



上图给出了基于 Tomasulo 算法的处理器的基本结构。保留站中每个位置保存一条已经被发射、正在等待执行的指令，如果已经计算出这一指令的操作数值，则保留该数值，如果还没有计算出，则保留提供这些操作数值的保留站名称。

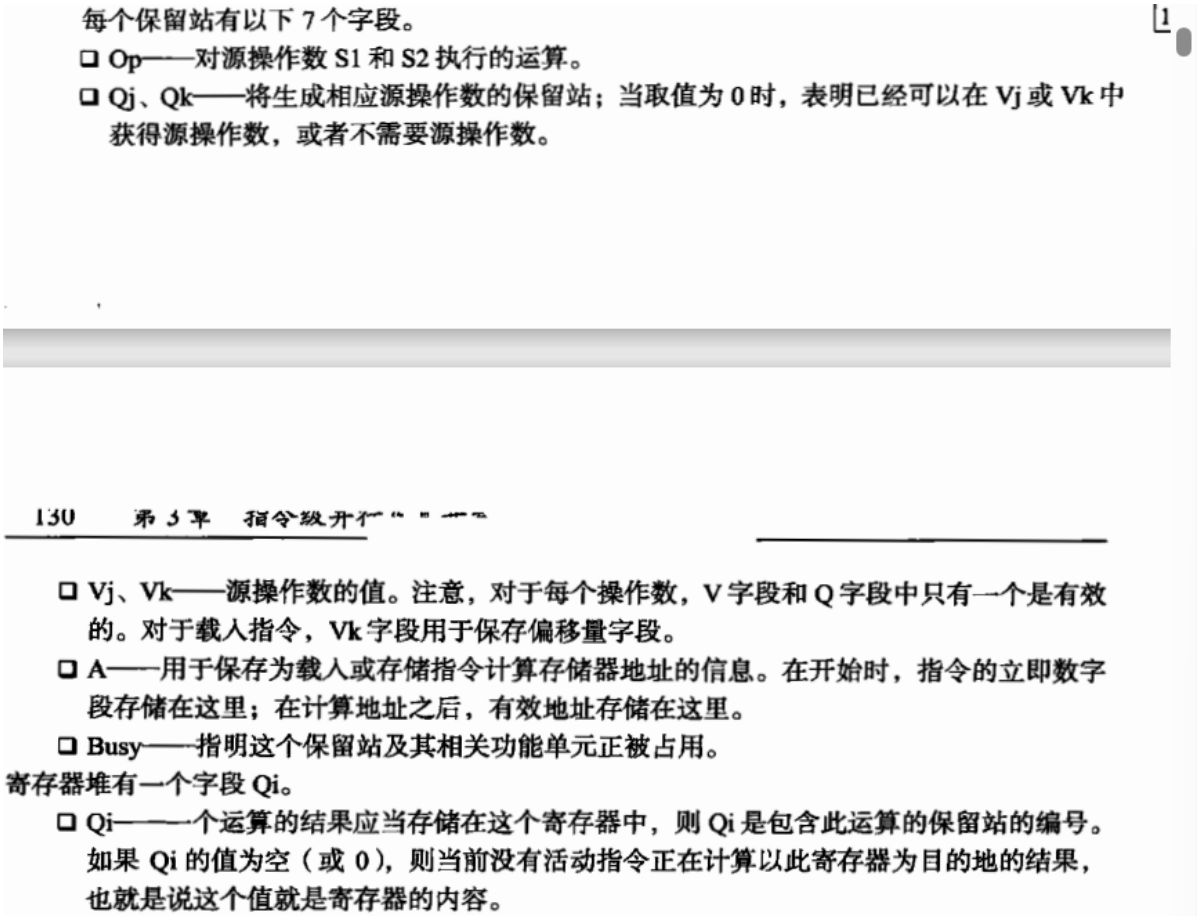
Load Store Buffer 保存来自和进入存储器的数据或地址。细化地说，它的功能包括：1）保存有效地址的各个部分，直到计算完成 2）跟踪正在等待存储器的未完成载入过程 3）保存正在等待 CDB 的已完成载入过程的结果。

目前，该算法的执行过程分为三步：

1. **发射**：从指令队列的头部获取下一条指令。如果有一个匹配保留站为空，就发射。如果没有，就不要发射（结构性危险）。如果操作数值当前已经存在于寄存器，也一并发送到站中。如果操作数不在，则一直跟踪将这些操作数生成的功能单元。
2. **执行**：如果一个操作数可用，就把它放到任何一个等待之的保留站中。当所有操作数都可用，则可以执行运算。对于浮点保留站，如果同时有多条指令准备就绪，可以任意做出选择。对于载入和存储指令，会复杂一点（后面应该会解决）

3. **写结果**：计算出结果后，写道寄存器和保留站中。存储指令一直缓存在存储缓冲区中，直到待存储值和存储地址可用为止。

在指令被发射出去并开始等待源操作数之后，将使用一个保留站编号来引用该操作数，这个编号是对该寄存器进行写操作的指令。如果用一个未用作保留站编号的数来记录（例如 0），那就表示该操作数已经在寄存器中准备就绪。这个保留站编号的记录方式就像"拓展虚拟寄存器"一样。



每个保留站有以下 7 个字段。

- Op——对源操作数 S1 和 S2 执行的运算。
- Qj、Qk——将生成相应源操作数的保留站；当取值为 0 时，表明已经可以在 Vj 或 Vk 中获得源操作数，或者不需要源操作数。

130 第 3 章 指令级并行

- Vj、Vk——源操作数的值。注意，对于每个操作数，V 字段和 Q 字段中只有一个是有效的。对于载入指令，Vk 字段用于保存偏移量字段。
- A——用于保存为载入或存储指令计算存储器地址的信息。在开始时，指令的立即数字段存储在这里；在计算地址之后，有效地址存储在这里。
- Busy——指明这个保留站及其相关功能单元正被占用。

寄存器堆有一个字段 Qi。

- Qi——一个运算的结果应当存储在这个寄存器中，则 Qi 是包含此运算的保留站的编号。如果 Qi 的值为空（或 0），则当前没有活动指令正在计算以此寄存器为目的地的结果，也就是说这个值就是寄存器的内容。

2.1 关于载入和存储指令的进一步讨论

当然，如果载入指令和存储指令访问的是不同地址，我们就可以乱序执行。但如果地址相同会怎么办？一个方法是检测排在某指令之前的所有载入指令或存储指令的地址和它本身的地址。但这样就必须要计算出之前所有指令的地址，实际上这并不总是可以的。

所以，一种简单的有效解决方式是严格按照顺序来执行运算。（实际上，只需要保持存储和其他指令的相对顺序，也就是说，可以随机调整载入指令的顺序）

2.2 Tomasulo 受到欢迎的原因

1. 尽管是在缓存产生之前设计的，但乱序执行能让处理器在等待解决缓存缺失的同时继续执行指令，从而消除了部分缓存缺失的代价。

2. 随着处理器的发射功能越来越强大，设计人员越来越关注难以调度的代码的性能，所以 tomasulo 中的技术（寄存器重命名、动态调度、推测）等技术变得越来越重要。
3. Tomasulo 的高性能并不建立在编译器针对特定流水线结构的优化。

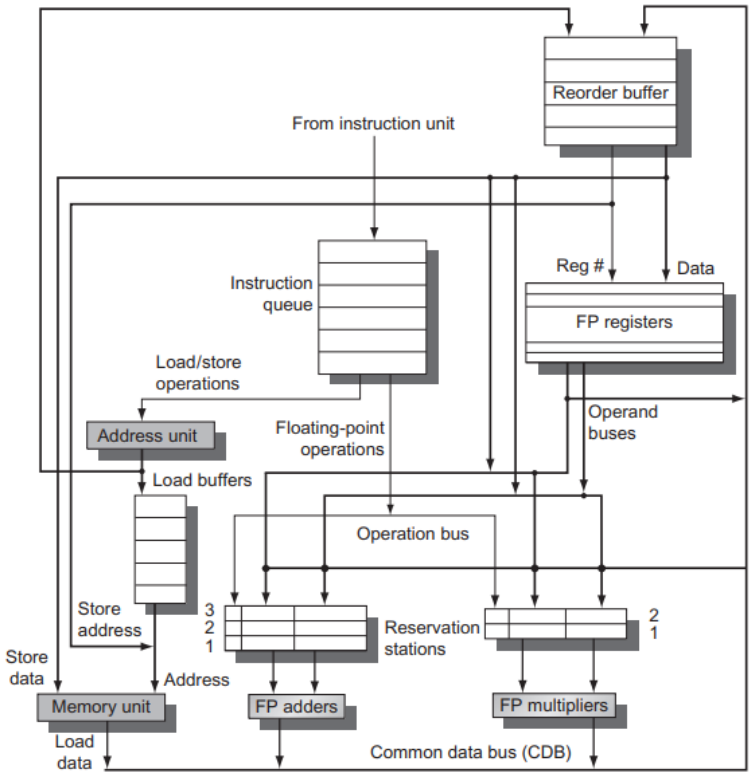
3. 基于硬件的推测

之前我们讨论的情况中，是假设分支预测总是正确的，但事实并非如此，我们需要一些机制来处理推测错误的情景。具体而言，我们不能允许一条分支指令执行任何不能撤销的更新操作，直到确认这条指令不再具有不确定性位置。

Tomasulo 算法的解决方案是**强制指令顺序提交**，以防在指令提交之前出现任何不可挽回的动作。然而，指令执行完毕的时间可能远远早于它们做好准备提交的时间，因而我们需要一组硬件缓冲区，也即**重排序缓冲区（Reorder Buffer）**。

同时，ROB 也是指令的操作数来源（在指令执行完毕到指令提交这段时间）。它里面的每个项目都包含 4 哥字段：指令类型（是分支、存储指令还是寄存器操作）、目的地字段（对于载入和算术指令，存储器地址为何）、值字段（指令的结果值）和就绪字段（结果值准备就绪）。

注意，在 ROB 引入之后，重命名的功能由 ROB 来完成，这个时候，保留站中就需要跟踪为一条指令分配的 ROB。



如此看来，指令执行时涉及以下四个步骤：

1. **发射**：从指令队列获得一条指令。如果保留站和 ROB 中均有空槽，那么发射该指令；如果寄存器或 ROB 含有某些操作数，则将其发送到保留站（如果全部都有我觉得可以直接发到 ROB，置为 Ready）。否则更新控制项，获取为该操作数分配的 ROB 项目编号。
2. **执行**：如果保留站中拥有操作数，执行该运算。指令在这一阶段可能占用多个周期
3. **写结果**：如果结果可用，将它写到 ROB 和任何等待这一结果的保留站上。
4. **提交**：根据要提交的指令是预测错误的分支、存储指令、或是其他指令有分类。如果一条指令在 ROB 的头部且已经 Ready，那么正常田炯。提交存储指令与正常提交类似，但更新的是存储器而不是结果寄存器（这需要你检查这个地址是否可用）。当预测错误的分支到达，你需要刷新 ROB LSB RS，因为类里面的东西肯定都是错误分支，同时，执行过程从该分支的后续正常指令处开始。指令一旦提交完毕，它在 ROB 的相应项都被收回，寄存器或存储器的目的地被更新，不再需要 ROB

4. 费曼学习法

2023.11.8，给 myf 同学等两位同学讲解了 tomasulo 算法，讨论了 i-cache 的设计和 LSB 的特殊设计