静态调度和动态调度

静态调度

简单流水线提取一条指令并发射它，除非流水线中的已有指令和被提取的指令之间存在数据相关，而且不能通过旁路来隐藏。旁路逻辑降低了实际流水线延迟，使特定的相关性不会导致冒险。如果存在不可避免的冒险，则冒险检测硬件会使流水线停顿（从使用该结构的冒险开始）。在清除这种相关性之前，不会提取或发射新指令。为了弥补这些性能损失，编译器可以尝试调度指令来避免冒险。

动态调度

Scoreboard

Q1：为什么要使用动态调度？

思考下面的一段代码：

```
fdiv.d  f0,f2,f4
fadd.d  f10,f0,f8
fsub.d  f12,f8,f14
```

减法指令与前面的指令无关却被阻塞。为使一条指令在其操作数可用是立即开始执行，不受其先前停顿指令的影响，必须将发射分为两部分：检查结构性冒险，等待数据冒险消失。我们希望指令在其数据操作数可用时立即开始执行，即流水线是乱序执行的。

为了实现乱序执行，我们必须将ID流水级分为两级：

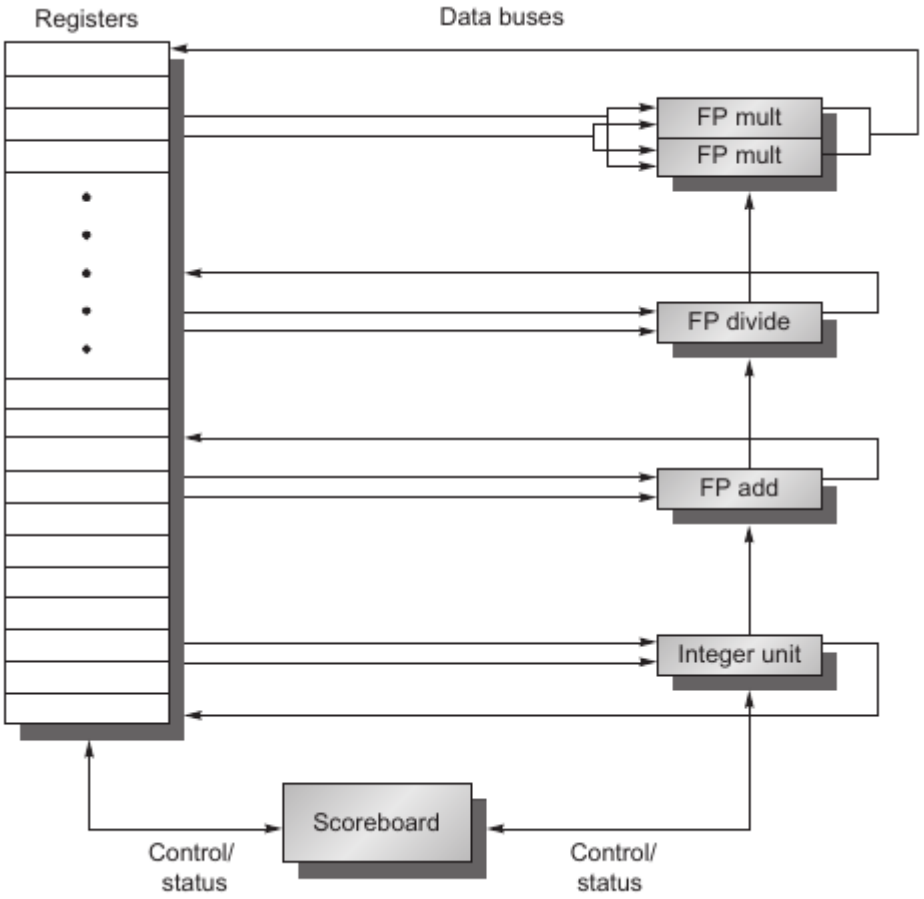
- **发射**——指令译码，检查结构性冒险
- **读取操作数**——等到没有数据冒险，随后读取操作数

> 要充分利用乱序执行，需要在其EX级中同时有多条指令。这一点可以通过多个功能单元、流水化功能单元或同时利用两者来实现

Q2:记分卡的概念

当下一条要执行的指令停顿时，如果其他指令不依赖与任何活动指令或停顿指令，则发射和执行这些指令。记分卡全面负责指令发射与执行，包括所有冒险检测任务。

1.每条指令都进入记分卡，在这里构建一条数据相关性记录（这一步与指令发射相对应，并替换流水线中的ID步骤）。2.记分卡随后判断指令什么时候能够读取它的操作数并开始执行。如果记分卡判断该指令不能立即执行，它监控硬件中的所有变化，以判断该指令何时能够执行。记分卡还控制一条指令什么时候能将其结果写到目标寄存器中。因此，所有冒险检测与解决都集中在记分卡。



Q3:记分卡的实现步骤

- **发射**——如果指令的一个功能单元空闲，没有其他活动指令以同一寄存器为目标寄存器，则记分卡向功能单元发射指令，并更新其内部数据结构。这一步替代了流水线中的ID步骤的一部分。*解决WAW冒险，确保没有其他活动功能单元希望将结果写入目标寄存器。如果存在结构性冒险或WAW冒险，则阻塞指令发射，在清除这些冒险之前，不会再发射其他指令。*
- **读取操作数**——记分卡监视源操作数的可用性。如果先前发射的活动指令都不再写入源操作数，该源操作数可用。*解决RAW冒险，当源操作数可用时，记分卡告诉功能单元继续从寄存器读取操作数，并开始执行。这一步和发射步骤一起，完成了流水线中ID级的功能。*
- **执行**——功能单元接收到操作数后开始执行。结果准备就绪后，通知记分卡已经完成执行。
- **写结果**——一旦记分卡知道功能单元已经完成执行，则检查WAR冒险，并在必要时停顿正在完成的指令

```
fdiv.d      f0,f2,f4
fadd.d      f10,f0,f8
fsub.d      f8,f8,f14
```

将先前的代码中ADD和SUB都使用F8,则存在WAR冒险。记分卡仍将SUB阻塞在写结果阶段，直到ADD读取操作数

在以下情况下，不允许一条正在执行的指令写入其结果：

- 在正在执行的指令前面（按发射顺序）有一条指令还没有读取其操作数
- 这些操作数之一与正执行指令的结果是同一寄存器

如果不存在这一WAR冒险，或者已经清除，则记分卡会告诉功能单元将其结果存储到目标寄存器中。

Tomasulo

Q1：算法核心思想

1. 允许在操作数可用时立即执行指令，避免RAW冒险
2. 重命名寄存器以避免WAR和WAW冒险

保留站的概念

在tomasulo算法中，寄存器重命名功能由保留站提供，保留站会为等待发射的指令缓冲操作数。

基本思想

保留站在一个操作数可用时马上提取并缓冲它，这样就不再需要从寄存器中获取该操作数。此外，等待执行的指令会指定保留站，为自己既提供输入。在对寄存器连续进行写入操作并且重叠执行时，只会实际使用最后一个操作更新寄存器。在发射指令时，会将待用操作数的寄存器说明符更名，改为保留站的名字，这就实现了寄存器重命名。

字段组成

- Op–对源操作数S1和S2执行的运算。
- Qj和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。
- Vj和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字段用于保存偏移字段。
- A–用于保存载入或者存储指令计算储存器地址的信息。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。

寄存器重命名如何消除WAR和WAW冒险？

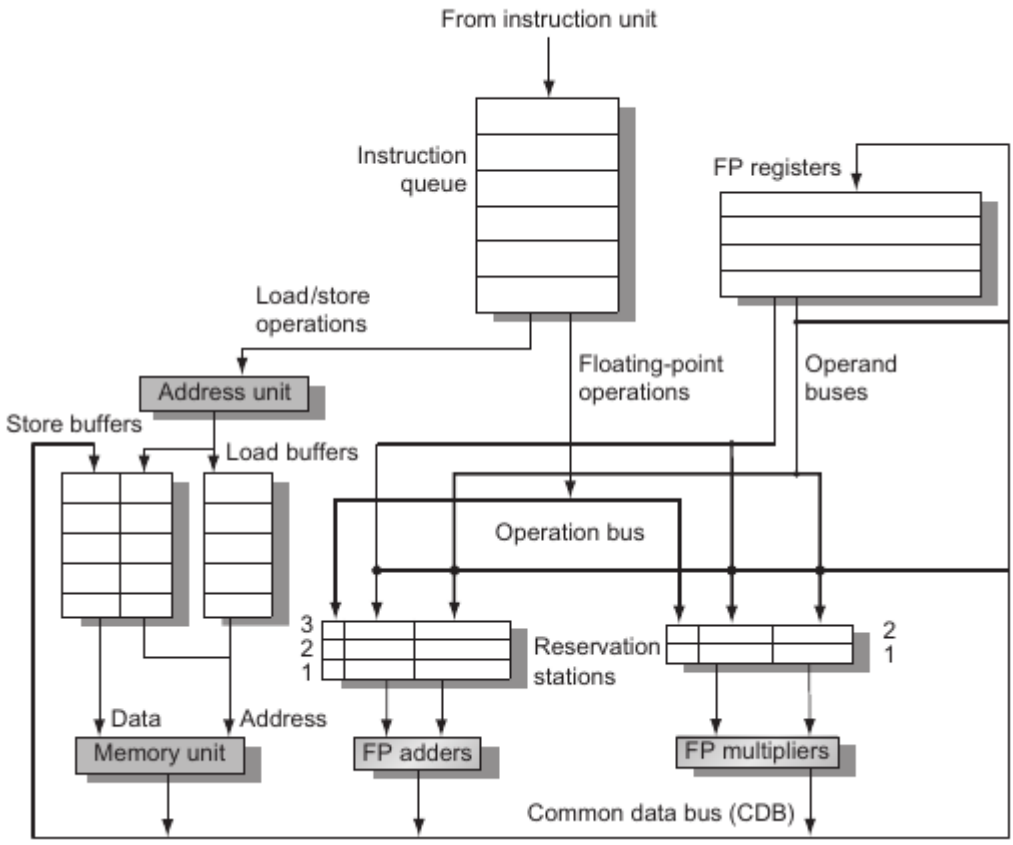
考虑下面可能出现WAR和WAW冒险的代码：

```
fdiv.d  f0,f2,f4
fadd.d  f6,f0,f8
fsd     f6,0(x1)
fsub.d  f8,f10,f14
fmul.d  f6,f10,f8
```

两处反相关，ADD和SUB之间，SD和MUL之间；一处输出相关，ADD和MUL之间；三处真数据相关，DIV和ADD之间，SUB和MUL之间，ADD和SD之间。假定存在两个临时寄存器：S和T。利用S和T对这段代码进行改写，使其没有任何相关，如下图所示：

```
fdiv.d  f0,f2,f4
fadd.d  S,f0,f8
fsd     S,0(x1)
fsub.d  T,f10,f14
fmul.d  f6,f10,T
```

Q2:算法的基本步骤



1. 发射–从指令队列的头部获取下一条指令，指令队列按FIFO顺序维护，以确保能够保持数据流的正确性。如果有一个匹配保留站为空，则将这条指令发送到这个站中，如果操作数值当前已经存在于寄存器，也一起发送到站中。如果没有空保留站，则存在结构冒险，该指令会被阻塞，直到有保留站或缓冲区被释放为止。如果操作数不在寄存器中，则一直跟踪将生成这些操作数的功能单元。这一部将对寄存器进行重命名，消除WAR和WAW冒险（有时这一步被称为派遣）
2. 执行–如果还有一个或多个操作数不可用，则在等待计算的同时监视公共数据总线。当一个操作数变为可用时，就将它放到任何一个正在等待它的保留站中。当所有操作数都可用时，则可以

在相应功能单元执行运算。通过延迟指令执行，直到操作数可用为止，可以避免RAW冒险（有时这一步被称为发射）

3. 写结果–在计算出结果后，将其写到CDB，再从CDB传递给寄存器和任意等待这一结果的保留站。存储指令一直缓存在缓存缓冲区中，直到待存储值和存储地址可用为止，然后在有空闲存储单元时，写入结果

载入和存储指令：载入和存储指令需要两个步骤。第一步，使用基址寄存器计算有效地址，将有效地址放在载入缓冲区或存储缓冲区。第二步，载入指令在存储器单元可用时立即执行；存储指令等待要存储的值，然后将其发送到存储单元。

分支指令：为了保持异常行为，对于任何一条指令，必须要等到根据程序顺序排在这条指令之前的所有分支全部完成之后，确保异常指令被正确执行，而不是在预测错误的方向上执行。在使用分支预测的处理器中，处理器在允许分支之后的指令开始执行之前，必须知道分支预测是正确的，否则分支之后的指令可能先写结果导致写回错误结果，不能消除分支预测错误的影响。

旁路：采用CDB，再由保留站从总线中提取结果，共同实现了静态调度流水线中使用的旁路机制。所以在动态调度中，在生成结果的指令与使用结果的指令之间至少要比生成该结果的功能单元的延迟长一个时钟周期。

Q3:算法细节

Instruction state	Wait until	Action or bookkeeping
Issue FP operation	Station r empty	`if (RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; if (RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rt].Qi else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0}; RS[r].Busy ← yes; RegisterStat[rd].Q ← r;`
Load or store	Buffer r empty	`if (RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; RS[r].A ← imm; RS[r].Busy ← yes;`
Load only		`RegisterStat[rt].Qi ← r;`
Store only		`if (RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rs].Qi} else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};`
Execute FP operation	`(RS[r].Qj = 0)` and `(RS[r].Qk = 0)`	Compute result: operands are in Vj and Vk
Load/storestep 1	RS[r].Qj = 0 & r is head of load-store queue	`RS[r].A ← RS[r].Vj + RS[r].A;`
Load step 2	Load step 1 complete	Read from `Mem[RS[r].A]`
Write result FP operation or load	Execution complete at r & CDB available	`∀x(if (RegisterStat[x].Qi=r) {Regs[x] ← result; RegisterStat[x].Qi ← 0}); ∀x(if (RS[x].Qj=r) {RS[x].Vj ← result;RS[x].Qj ← 0}); ∀x(if (RS[x].Qk=r) {RS[x].Vk ← result;RS[x].Qk ← 0}); RS[r].Busy ← no;`
Store	Execution complete at r & `RS[r].Qk = 0`	`Mem[RS[r].A] ← RS[r].Vk; RS[r].Busy ← no;`

Q4:算法举例

考虑下面的简单循环，将一个数组的元素乘以f2中的标量：

```
Loop:  fld     f0,0(x1)
       fmul.d  f4,f0,f2
       fsd     f4,0(x1)
       addi    x1,x1,-8
       bne     x1,x2,Loop  // branches if x1≠x2
```

如果我们预测会执行这些分支，那么保留站可以同时执行多次循环的多条指令；假设已经发射了该循环两个连续迭代中的所有指令，但一个浮点载入/存储指令或运算也没有完成。

		Instruction status			
Instruction		From iteration	Issue	Execute	Write result
fld	f0,0(x1)	1	√	√	
fmul.d	f4,f0,f2	1	√		
fsd	f4,0(x1)	1	√		
fld	f0,0(x1)	2	√	√	
fmul.d	f4,f0,f2	2	√		
fsd	f4,0(x1)	2	√		

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	Yes	Load					Regs[x1] + 0
Load2	Yes	Load					Regs[x1] − 8
Add1	No						
Add2	No						
Add3	No						
Mult1	Yes	MUL		Regs[f2]	Load1		
Mult2	Yes	MUL		Regs[f2]	Load2		
Store1	Yes	Store	Regs[x1]			Mult1	
Store2	Yes	Store	Regs[x1] − 8			Mult2	

	Register status								
Field	f0	f2	f4	f6	f8	f10	f12	…	f30
Qi	Load2		Mult2						

载入和存储指令:只要载入指令和存储指令访问的是不同地址，就可以放心地乱序执行它们。如果载入指令和存储指令访问相同地址，交换顺序就会导致冒险。

在给定时刻是否可以执行一条载入指令。如果按程序顺序执行有效地址计算，那么当一条载入指令完成有效地址计算时，就可以通过查看所有活动存储缓冲区的A字段来确定是否存在地址冲突。如果载入地址与存储缓冲区中任何活动项目的地址匹配，则在发生冲突的存储器指令完成之前，不要将载入指令发送到载入缓冲区。对于存储指令也是如此。后面考虑一种去除这一限制的方法

针对分支预测扩展Tomasulo算法

分支预测减少了由于分支导致的直接停顿，但是就如上一节中提到的，处理器要确保异常指令的正确执行，并且有消除分支预测错误的影响的能力，在此基础上提高并行能力需要克服控制相关的局限性。

考虑分支指令的不确定性，我们无法知道指令是否提供了正确值，如果我们将指令结果的旁路从一条指令的实际完成操作中分离出来。这样就可以允许执行一条指令，并将其结果旁路给其他指令，但不允许这条指令执行任何不能撤销的更新操作，知道确定这条指令不再具有不确定性为止。当一个指令不再具有不确定性时，允许它更新寄存器堆或存储器，这个步骤称为指令提交。

当指令提交时，需要添加一组硬件缓冲区，用来保存已经完成执行但还没有提交的指令结果，这一硬件缓冲区称为重排序缓冲区（ROB）。ROB会在一定时间内保存指令的结果，这段时间