

计算机系统结构

第5章 指令级并行及其开发

目录

- 5.1 [指令级并行的概念](#)
- 5.2 [相关与指令级并行](#)
- 5.3 [指令的动态调度](#)
- 5.4 [动态分支预测技术](#)
- 5.5 [多指令流出技术](#)

- **指令级并行（ILP: Instruction-Level Parallelism）**：指令之间存在的一种并行性，利用它，计算机可以并行执行两条或两条以上的指令。
- 开发**ILP**的途径有两种
 - ▣ 资源重复，重复设置多个处理部件，让它们同时执行相邻或相近的多条指令；
 - ▣ 采用流水线技术，使指令重叠并行执行。
- **本章研究**：运用流水线以及相关优化技术，使得指令能够重叠并行执行。



并行：


指令级别

线程级别

处理器之间

独立计算机之间

计算机并行：有不同的粒度（指令级、线程级、…数据）。



5.1 指令级并行的概念

1. 流水线处理机的实际CPI

- 理想流水线的CPI加上各类停顿的时钟周期数：

$$CPI_{\text{流水线}} = CPI_{\text{理想}} + \text{停顿}_{\text{结构冲突}} + \text{停顿}_{\text{数据冲突}} + \text{停顿}_{\text{控制冲突}}$$

- 理想CPI是衡量流水线最高性能的一个指标。

- **IPC**: Instructions Per Cycle

（每个时钟周期完成的指令条数，CPI的倒数）

2. 开发ILP的方法可以分为两大类

- 主要基于硬件的动态开发方法
- 基于软件的静态开发方法

5.1 指令级并行的概念

3. 基本程序块(Basic Block)

- **基本程序块**：一串连续的代码除了入口和出口以外，没有其他分支指令和转入点。
- 程序平均每4~7条指令就会有一个分支。
- 基本程序块中能开发的并行性有限；
- 为了明显提高性能，必须跨越多个基本块开发ILP。

5.1 指令级并行的概念

4. 循环级并行(Loop-Level Parallelism): 使一个循环中的不同循环体并行执行。

- 开发循环的不同叠代之间存在的并行性(最常见、最基本)
- 是指令级并行研究的重点之一
- 例:

```
for (i=1; i<=500; i=i+1)
    a[i]=a[i]+s;
```

 - ▣ 每一次循环的内部, 没有任何的并行性;
 - ▣ 每一次循环都可以与其它的循环重叠并行执行。

5. 最基本的开发循环级并行的技术

- 循环展开 (loop unrolling) 技术
- 采用向量指令和向量数据表示

5.2 相关与指令级并行

1. 开发指令级并行需要解决的具体问题——相关与流水线冲突

- **相关**：两条指令之间存在某种依赖关系

三种类型：**数据相关、名相关、控制相关**

- **流水线冲突**是指对于具体的流水线来说，由于相关的存在，使得指令流中的下一条指令不能在指定的时钟周期执行。

三种类型：**结构冲突、数据冲突、控制冲突**

- 相关是程序固有的一种属性，它反映了程序中指令之间的相互依赖关系。
- 具体的一次相关是否会导致实际冲突的发生以及该冲突会带来多长的停顿，则是流水线的属性。

5.2 相关与指令级并行

2. 相关的两类解决方案：

- 保持相关，但避免发生冲突。
- 通过代码变换，消除相关。

➡ 指令调度
不改变相关，避免冲突

3. 程序顺序（Program Order）：由原来程序确定的在完全串行方式下指令的执行顺序。并不需要在所有存在相关的地方保持程序顺序，只有在可能会导致错误的情况下，才保持程序顺序。

4. 控制相关并不是一个必须严格保持的关键属性。

5.2 相关与指令级并行

5. 对于正确地执行程序来说，必须保持的最关键的两个属性是：数据流和异常行为。

- **保持异常行为 (Exception Behavior)** 是指：无论怎么改变指令的执行顺序，都不能改变程序中异常的发生情况。
 - 即原来程序中是怎么发生的，改变执行顺序后还是怎么发生。
 - 常弱化为：指令执行顺序的改变不能导致程序中发生新的异常。
- **数据流 (Data Flow)**：指数据值从其产生者指令到其消费者指令的实际流动。
 - 分支指令使得数据流具有动态性，因为一条指令有可能数据相关于多条先前的指令。
 - 分支指令的执行结果决定了哪条指令真正是所需数据的产生者。

5.2 相关与指令级并行

➤ 有时，不遵守控制相关既不影响异常行为，也不改变数据流。

- 可以大胆地进行指令调度，把失败分支中的指令调度到分支指令之前。

- 举例：

DADDU	R1, R2, R3
BEQZ	R12, Skipnext
DSUBU	R4, R5, R6
DADDU	R5, R4, R9
Skipnext: OR	R7, R8, R9



5.3 指令的动态调度

- **静态调度 (Static Scheduling)** ==指令调度的分类==
 - 依靠编译器对代码进行静态调度，以减少相关和冲突。
 - 它不是在程序执行的过程中、而是在编译期间进行代码调度和优化。
 - 通过把相关的指令拉开距离来减少可能产生的停顿。
- **动态调度 (Dynamic Scheduling)**
 - 在程序的执行过程中，依靠专门硬件对代码进行调度，减少数据相关导致的停顿。
 - **优点：**
 - *能够处理一些在编译时情况不明的相关（比如涉及到存储器访问的相关），并简化了编译器；
 - *能够使本来是面向某一流水线优化编译的代码在其它的流水线（动态调度）上也能高效地执行。
 - 以硬件复杂性的显著增加为代价

5.3 指令的动态调度

5.3.1 动态调度的基本思想

1. 经典（顺序）流水线的最大的局限性:

- 指令 按序流出（In-Order-Issue）
按序执行（In-Order-Execution）

- 考虑下面一段代码：

DIV. D F4, F0, F2 (10个周期)

ADD. D F10, F4, F6 (1个周期)

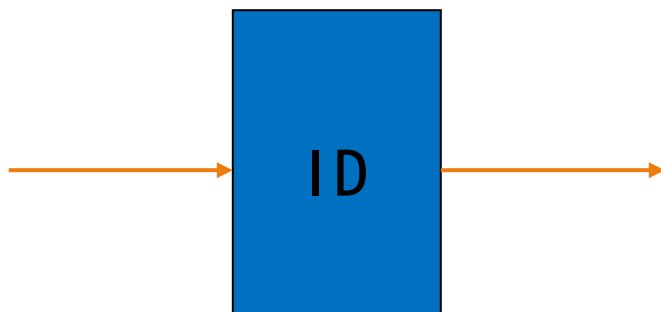
SUB. D F12, F6, F14 (1个周期)

ADD. D指令与DIV. D指令关于F4相关，导致流水线停顿。

SUB. D指令与流水线中的任何指令都没有关系，但也因此受阻。

5.3 指令的动态调度

在前面的基本流水线中译码阶段：



检测结构冲突

检测数据冲突

一旦一条指令受阻，其后的指令都将停顿。

5.3 指令的动态调度

- 为了使上述指令序列中的SUB.D指令能继续执行下去，必须把指令流出的工作拆分为两步：
 - 检测结构冲突
 - 等待数据冲突消失

只要检测到没有结构冲突，就可以让指令流出。并且流出后的指令一旦其操作数就绪就可以立即执行。

2. 乱序执行

- 指令的执行顺序与程序顺序不相同
- 指令的完成也是乱序完成的
 - 即指令的完成顺序与程序顺序不相同。

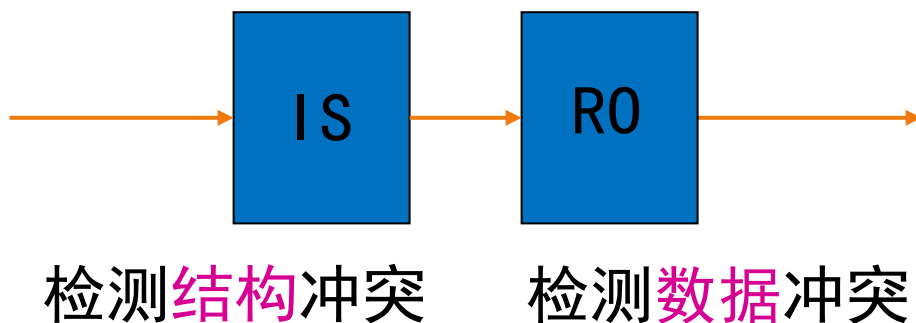
5.3 指令的动态调度

3. 为了乱序执行，将5段流水线的译码阶段（ID）再分为两个阶段：

- 流出（Issue, IS）：指令译码，检查是否存在结构冲突。
- 读操作数（Read Operands, RO）：等待数据冲突消失，然后读操作数。

按序流出（in-order issue）

乱序执行（out of order execution）



5.3 指令的动态调度

4. 引入指令缓冲区，直到冲突消除
5. 部署更多的执行部件，使多条指令能同时执行或访存
6. 在前述5段流水线中，是不会发生WAR冲突和WAW冲突的。但乱序执行就使得它们可能发生了。

➤ 例如，考虑下面的代码

	DIV. D	F10, F0, F2	} 存在输出相关 (WAW)
存在反相关 (WAR)	{ ADD. D	F10, F4, F6	
	SUB. D	F6, F8, F14	

可以通过使用寄存器重命名来消除。

5.3 指令的动态调度

7. 动态调度的流水线支持多条指令同时处于执行当中。

➤ 要求：具有

多个功能部件、

或者 功能部件流水化、

或者兼而有之。

➤ 我们假设具有多个功能部件。

5.3 指令的动态调度

8. 复杂的异常处理

指令乱序完成带来的最大问题: 异常处理比较复杂

- 动态调度的处理机要保持正确的异常行为
 - 对于一条会产生异常的指令来说，只有当处理机确切地知道该指令将被执行时，才允许它产生异常。
- 即使保持了正确的异常行为，动态调度处理机仍可能发生不精确异常。

5.3 指令的动态调度

- **精确异常 (Precise Exception)** : 如果发生异常时, 处理机的现场跟严格按程序顺序执行时指令*i*的现场相同。
- **不精确异常 (Imprecise Exception)** : 当执行指令*i*导致发生异常时, 处理机的现场 (状态) 与严格按程序顺序执行时指令*i*的现场不同。
- 发生不精确异常的原因: 当发生异常 (设为指令*i*) 时
 - 流水线可能已经执行完按程序顺序是位于指令*i*之后的指令;
 - 流水线可能还没完成按程序顺序是指令*i*之前的指令。
- 不精确异常使得在异常处理后难以接着继续执行程序。

5.3 指令的动态调度

两种典型的动态调度算法：

记分牌算法和Tomasulo算法

5.3 指令的动态调度

5.3.2 记分牌（Scoreboard）动态调度算法

1. 基本思想

➤ CDC 6600计算机最早采用此功能

- 该机器用一个称为记分牌的硬件实现了对指令的动态调度。
- 该硬件中维护着3张表，分别用于记录指令的执行状态、功能部件状态、寄存器状态以及数据相关关系等。
- 它把前述5段流水线中的译码段ID分解成了两个段：流出和读操作数，以避免当某条指令在ID段被停顿时挡住后面无关指令的流动。

5.3 指令的动态调度

- 记分牌的**目标**：在没有结构冲突时，尽可能早地执行没有数据冲突的指令，实现每个时钟周期执行一条指令。
- 要发挥指令乱序执行的好处，必须有多条指令同时处于执行阶段。
 - CDC 6600具有**16**个独立的功能部件
 - **4**个浮点部件
 - **5**个访存部件
 - **7**个整数操作部件
- 假设
 - 所考虑的处理器有**2**个乘法器、**1**个加法器、**1**个除法部件和**1**个整数部件。
 - 整数部件用来处理所有的存储器访问、分支处理和整数操作。

5.3 指令的动态调度

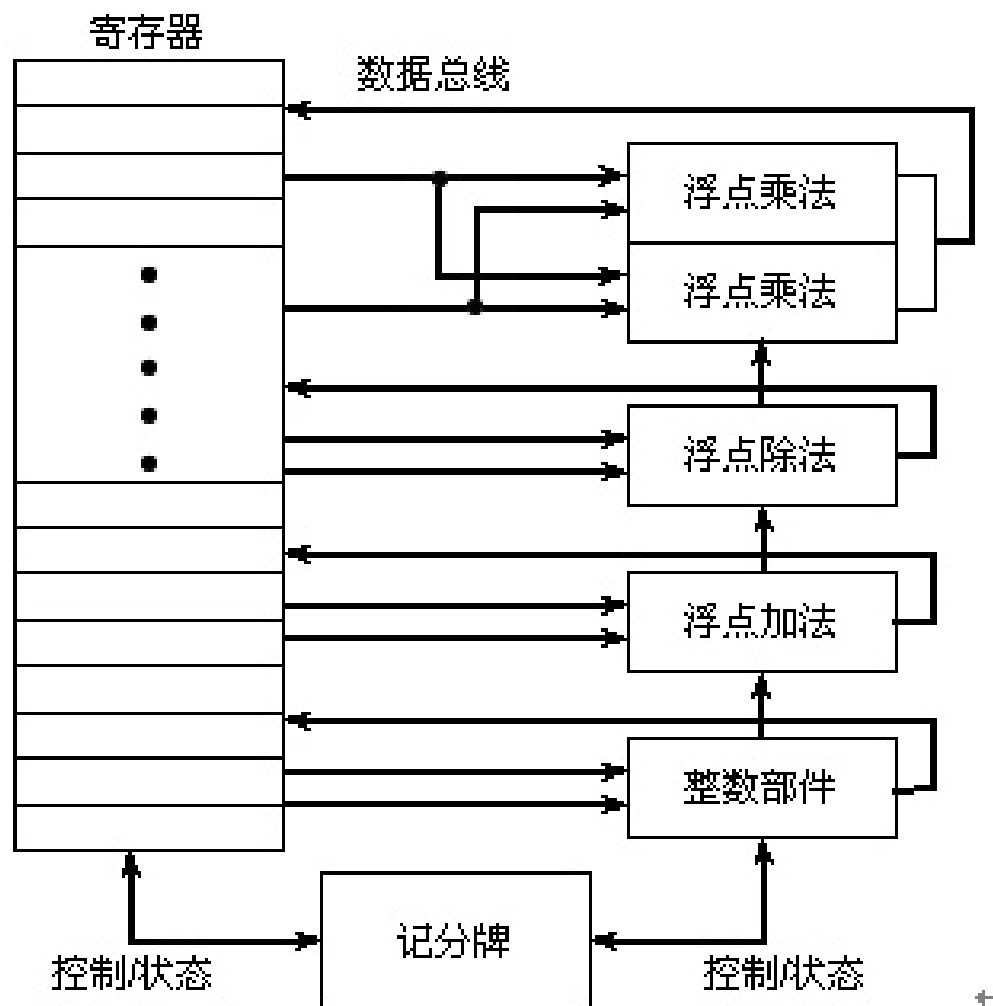
- 采用了记分牌的MIPS处理器的基本结构
 - ▣ 每条指令都要经过记分牌。
 - ▣ 记分牌负责相关检测并控制指令的流出和执行。
- 每条指令的执行过程分为4段（主要考虑浮点操作）

- ▣ **流出 没有结构冲突和WAW(写后写)冲突**

如果当前流出指令所需的功能部件空闲，并且所有其他正在执行的指令的目的寄存器与该指令的不同，记分牌就向功能部件流出该指令，并修改记分牌内部的记录表。

解决了WAW冲突

5.3 指令的动态调度



具有记分牌的MIPS处理器的基本结构

5.3 指令的动态调度

□ 读操作数

记分牌监测源操作数的可用性，如果数据可用，它就通知功能部件从寄存器中读出源操作数并开始执行。

动态地解决了RAW冲突，并导致指令可能乱序开始执行。

□ 执行

取到操作数后，功能部件开始执行。当产生出结果后，就通知记分牌它已经完成执行。

在浮点流水线中，这一段可能要占用多个时钟周期。

□ 写结果

记分牌一旦知道执行部件完成了执行，就检测是否存在WAR冲突。如果不存在，或者原有的WAR冲突已消失，记分牌就通知功能部件把结果写入目的寄存器，并释放该指令使用的所有资源。

5.3 指令的动态调度

- 如果检测到WAR冲突，就不允许该指令将结果写到目的寄存器。这发生在以下情况：
 - 前面的某条指令（按顺序流出）还没有读取操作数；而且：其中某个源操作数寄存器与本指令的目的寄存器相同。
 - 在这种情况下，记分牌必须等待，直到该冲突消失。

5.3 指令的动态调度

➤ 记分牌中记录的信息由3部分构成

- **指令状态表**：记录正在执行的各条指令已经进入到了哪一段。
- **功能部件状态表**：记录各个功能部件的状态。每个功能部件有一项，每一项由以下9个字段组成：
 - **Busy**：忙标志，指出功能部件是否忙。初值为“no”；
 - **Op**：该功能部件正在执行或将要执行的操作；
 - **Fi**：目的寄存器编号；**Fj**，**Fk**：源寄存器编号；
 - **Qj**，**Qk**：指出向源寄存器Fj、Fk写数据的功能部件；
 - **Rj**，**Rk**：标志位，为“yes”表示Fj，Fk中的操作数就绪且还未被取走。否则就被置为“no”。
- **结果寄存器状态表Result**：每个寄存器在该表中有一项，用于指出哪个功能部件（编号）将把结果写入该寄存器。
 - 正在运行的指令都不以它为目的寄存器，则置为“no”。
 - **Result**各项的初值为“no”（全0）。

5.3 指令的动态调度

2. 举例

- MIPS记分牌所要维护的数据结构
- 下列代码运行过程中记分牌保存的信息

L. D F6, 34(R2)

L. D F2, 45(R3)

MULT. D F0, F2, F4

SUB. D F8, F6, F2

DIV. D F10, F0, F6

ADD. D F6, F8, F2

没有 写后写和
结构冲突

指令	指令状态表			
	流出	读操作数	执行	写结果
L.D F6,34(R2)	√	√	√	√
L.D F2, 45(R3)	√	√	√	
MULT.D F0, F2, F4	√			
SUB.D F8, F6, F2	√			
DIV.D F10, F0, F6	√			
ADD.D F6, F8, F2				

部件名称	功能部件状态表								
	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	yes	L.D	F2	R3				no	
Mult1	yes	MULT.D	F0	F2	F4	Integer		no	yes
Mult2	no								
Add	yes	SUB.D	F8	F6	F2		Integer	yes	no
Divide	yes	DIV.D	F10	F0	F6	Mult1		no	yes

	结果寄存器状态表							
	F0	F2	F4	F6	F8	F10	...	F30
部件名称	Mult1	Integer			Add	Divide		

MIPS记分牌中的信息

5.3 指令的动态调度

例5.1 假设浮点流水线中各部件的延迟如下：

加法需2个时钟周期

乘法需10个时钟周期

除法需40个时钟周期

代码段和记分牌信息的起始点状态如上图。分别给出MULT.D和DIV.D准备写结果之前的记分牌状态。

解 图中的代码段存在以下相关性：

- (1) 先写后读相关：第二条L.D指令到MULT.D和SUB.D之间，
MULT.D到DIV.D之间， SUB.D到ADD.D之间；
- (2) 先读后写相关： DIV.D和ADD.D之间， SUB.D和ADD.D之间；
- (3) 结构相关： ADD.D和SUB.D指令关于浮点加法部件。

指 令	指令状态表			
	流出	读操作数	执行	写结果
L.D F6, 34(R2)	√	√	√	√
L.D F2, 45(R3)	√	√	√	√
MULT.D F0, F2, F4	√	√	√	
SUB.D F8, F6, F2	√	√	√	√
DIV.D F10, F0, F6	√			
ADD.D F6, F8, F2	√	√	√	

部件名称	功能部件状态表								
	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	no								
Mult1	yes	MULT.D	F0	F2	F4			no	no
Mult2	no								
Add	yes	ADD.D	F6	F8	F2			no	no
Divide	yes	DIV.D	F10	F0	F6	Mult1		no	yes

	结果寄存器状态表							
	F0	F2	F4	F6	F8	F10	...	F30
部件名称	Mult1			Add		Divide		

程序段执行到MULT. D将要写结果时记分牌的状态

指 令	指令状态表			
	流出	读操作数	执行	写结果
L.D F6, 34(R2)	✓	✓	✓	✓
L.D F2, 45(R3)	✓	✓	✓	✓
MULT.D F0, F2, F4	✓	✓	✓	✓
SUB.D F8, F6, F2	✓	✓	✓	✓
DIV.D F10, F0, F6	✓	✓	✓	
ADD.D F6, F8, F2	✓	✓	✓	✓

部件名称	功能部件状态表								
	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	no								
Mult1	no								
Mult2	no								
Add	no								
Divide	yes	DIV.D	F10	F0	F6			no	no

	结果寄存器状态表							
	F0	F2	F4	F6	F8	F10	...	F30
部件名称						Divide		

程序段执行到DIV. D将要写结果时记分牌的状态

5.3 指令的动态调度

3. 具体算法

约定：

- **FU**：表示当前指令所要用的功能部件；
- **D**：目的寄存器的名称；
- **S1、S2**：源操作数寄存器的名称；
- **Op**：要进行的操作；
- **Fj[FU]**：功能部件FU的Fj字段（其他字段依此类推）；
- **Result[D]**：结果寄存器状态表中与寄存器D相对应的内容。
其中存放的是将把结果写入寄存器D的功能部件的名称。

5.3 指令的动态调度

(1) 指令流出

进入条件：

not Busy[FU] & not Result[D]; // 功能部件空闲且没有
//写后写（WAW）冲突。

计分牌内容修改：

Busy[FU] ← yes; // 把当前指令的相关信息填入
// 功能部件状态表。

Op[FU] ← Op; // 记录操作码。

Fi[FU] ← D; // 记录目的寄存器编号。

Fj[FU] ← S1; // 记录第一个源寄存器编号。

5.3 指令的动态调度

Fk[FU]←S2; // 记录第二个源寄存器编号。

Qj[FU]←Result[S1]; // 记录将产生第一个源操作数的部件。

Qk[FU]←Result[S2]; // 记录将产生第二个源操作数的部件。

Rj[FU]←not Qj[FU]; // 置第一个源操作数是否可用的标志。
//如果Qj[FU]为“no”，表示没有操作部件要写S1，数据可用，
//置Rj[FU]为“yes”。否则置Rj[FU]为“no”。

Rk[FU]←not Qk[FU]; // 置第二个源操作数是否可用的标志。

Result[D]←FU; // D是当前指令的目的寄存器。
// 功能部件FU将把结果写入D。

5.3 指令的动态调度

(2) 读操作数

进入条件：

$Rj[FU] \ \& \ Rk[FU]$; // 两个源操作数都已就绪。

计分牌内容修改：

$Rj[FU] \leftarrow no$; // 已经读走了就绪的第一个源操作数。

$Rk[FU] \leftarrow no$; // 已经读走了就绪的第二个源操作数。

$Qj[FU] \leftarrow 0$; // 不再等待其他FU的计算结果。

$Qk[FU] \leftarrow 0$;

5.3 指令的动态调度

(3) 执行

结束条件：

功能部件操作结束。

(4) 写结果

进入条件：

$\forall f((Fj[f] \neq Fi[FU] \text{ or } Rj[f] = \text{no})$

$\& (Fk[f] \neq Fi[FU] \text{ or } Rk[f] = \text{no}))$; // 不存在WAR冲突。

记分牌内容修改：

5.3 指令的动态调度

$\forall f(\text{if } Q_j[f]=FU \text{ then } R_j[f] \leftarrow \text{yes});$ // 如果有指令在等待该结果（作为第一源操作数），则将其 R_j 置为“yes”，表示数据可用。

$\forall f(\text{if } Q_k[f]=FU \text{ then } R_k[f] \leftarrow \text{yes});$ // 如果有指令在等待该结果（作为第二源操作数），则将其 R_k 置为“yes”，表示数据可用。

$\text{Result}(F_i[FU]) \leftarrow 0;$ // 释放目的寄存器 $F_i[FU]$ 。

$\text{Busy}[FU] = \text{no};$ // 释放功能部件FU。

5.3 指令的动态调度

4. 记分牌的性能受限于以下几个方面：

- 程序代码中可开发的并行性，即是否存在可以并行执行的不相关的指令。
- 记分牌的容量。
 - 记分牌的容量决定了流水线能在多大范围内寻找不相关指令。流水线中可以同时容纳的指令数量称为**指令窗口**。
- 功能部件的数目和种类。
 - 功能部件的总数决定了结构冲突的严重程度。
- 名相关（反相关和输出相关）。
 - 反相关和输出相关引起记分牌中**WAR**和**WAW**冲突。
乱序流出引起更多的名相关。采用分支预测技术就会出现循环的多个迭代同时执行，名相关将更加严重。

5.3 指令的动态调度

5.3.3 Tomasulo算法

5.3.3.1 基本思想

1. 核心思想

- 记录和检测指令相关，操作数一旦就绪就立即执行，把发生RAW冲突的可能性减少到最小；
- 通过寄存器换名来消除WAR冲突和WAW冲突。

2. IBM 360/91首先采用了Tomasulo算法。

- IBM 360/91的设计目标是基于整个360系列的统一指令系统和编译器来实现高性能，而不是设计和利用专用的编译器来提高性能。

需要更多地依赖于硬件。

5.3 指令的动态调度

- IBM 360体系结构只有4个双精度浮点寄存器，限制了编译器调度的有效性。
- 360/91的访存时间和浮点计算时间都很长。
(也是Tomasulo算法要解决的问题)

3. 寄存器换名可以消除WAR冲突和WAW冲突。

- 考虑以下代码：

	DIV. D	F0, F2, F4			
反相关 (F8) 导致WAR冲突	{	ADD. D	F6, F0, F8	}	输出相关 (F6) 导致WAW冲突
		S. D	F6, 0 (R1)		
		SUB. D	F8, F10, F14		
		MUL. D	F6, F10, F8		

5.3 指令的动态调度

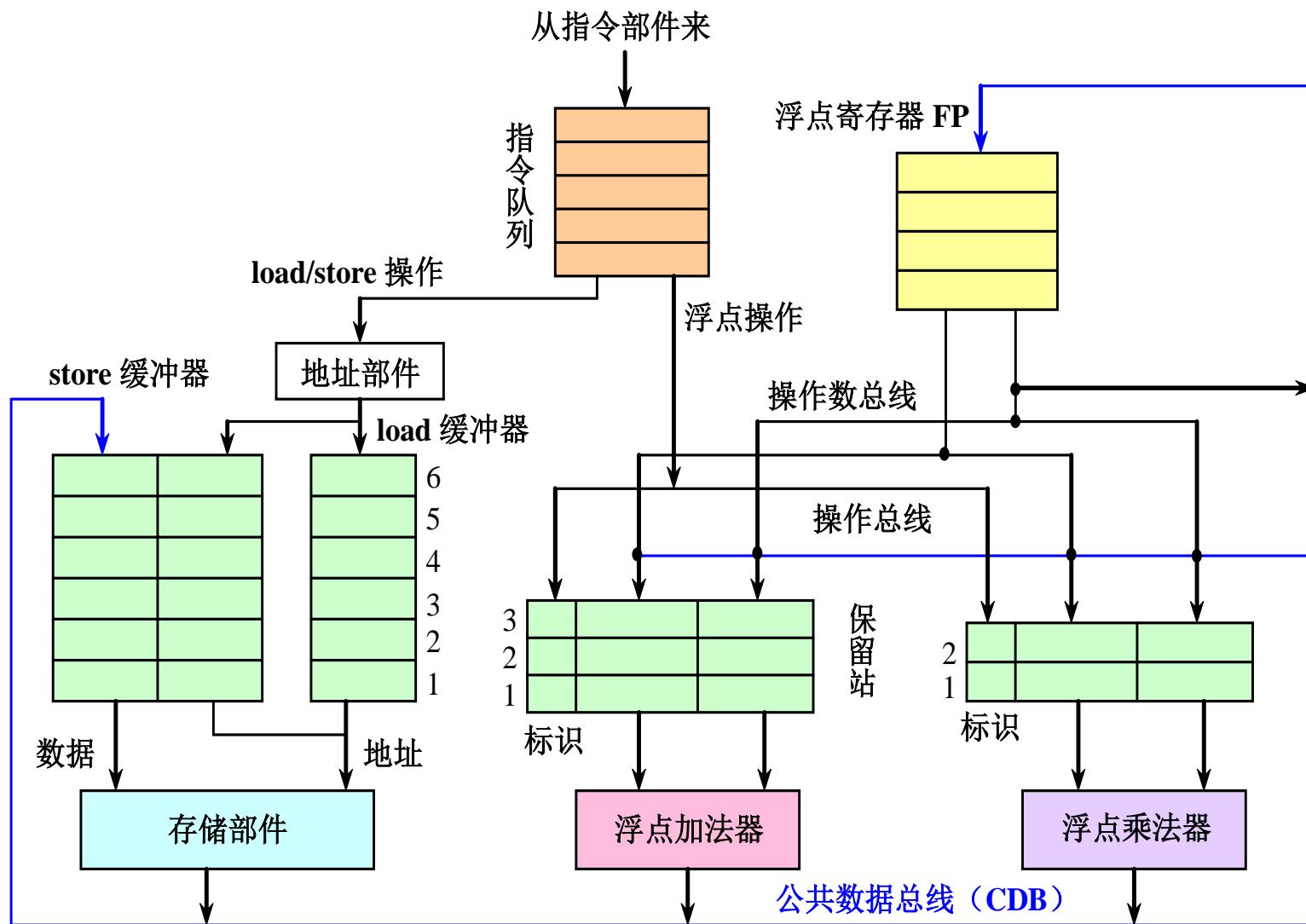
➤ 消除名相关

- 引入两个临时寄存器S和T
- 把这段代码改写为:

	DIV. D	F0, F2, F4	
	ADD. D	S, F0, F8	
	S. D	S, 0 (R1)	} 两个F6都换名为S
两个F8都换名为T {	SUB. D	T, F10, F14	
	MUL. D	F6, F10, T	

4. 基于Tomasulo算法的MIPS处理器浮点部件的基本结构

5.3 指令的动态调度



5.3 指令的动态调度

➤ 保留站 (reservation station)

每个保留站中保存一条已经流出并等待到本功能部件执行的指令（相关信息）。

包括：操作码、操作数以及用于检测 and 解决冲突的信息。

- 在一条指令流出到保留站的时候，如果该指令的源操作数已经在寄存器中就绪，则将之取到该保留站中。
 - 如果操作数还没有计算出来，则在该保留站中记录将产生这个操作数的保留站的标识。
- 浮点加法器有3个保留站：ADD1, ADD2, ADD3
 - 浮点乘法器有两个保留站：MULT1, MULT2
 - 每个保留站都有一个标识字段，唯一地标识了该保留站。

5.3 指令的动态调度

➤ 公共数据总线CDB

（一条重要的数据通路）

- 所有功能部件的计算结果都是送到CDB上，由它把这些结果直接送到（播送到）各个需要该结果的地方。
- 在具有多个执行部件且采用多流出（即每个时钟周期流出多条指令）的流水线中，需要采用多条CDB。

➤ load缓冲器和store缓冲器

- 存放读/写存储器的数据或地址
- load缓冲器的作用有3个：
 - 存放用于计算有效地址的分量；
 - 记录正在进行的load访存，等待存储器的响应；
 - 保存已经完成了的load的结果（即从存储器取来的数据），等待CDB传输。

5.3 指令的动态调度

- **store缓冲器的作用有3个：**
 - 存放用于计算有效地址的分量；
 - 保存正在进行的store访存的目标地址，该store正在等待存储数据的到达；
 - 保存该store的地址和数据，直到存储部件接收。
- **浮点寄存器FP**
 - 共有16个浮点寄存器：F0, F2, F4, ..., F30。
 - 它们通过一对总线连接到功能部件，并通过CDB连接到store缓冲器。
- **指令队列**
 - 指令部件送来的指令放入指令队列
 - 指令队列中的指令按先进先出的顺序流出

5.3 指令的动态调度

➤ 运算部件

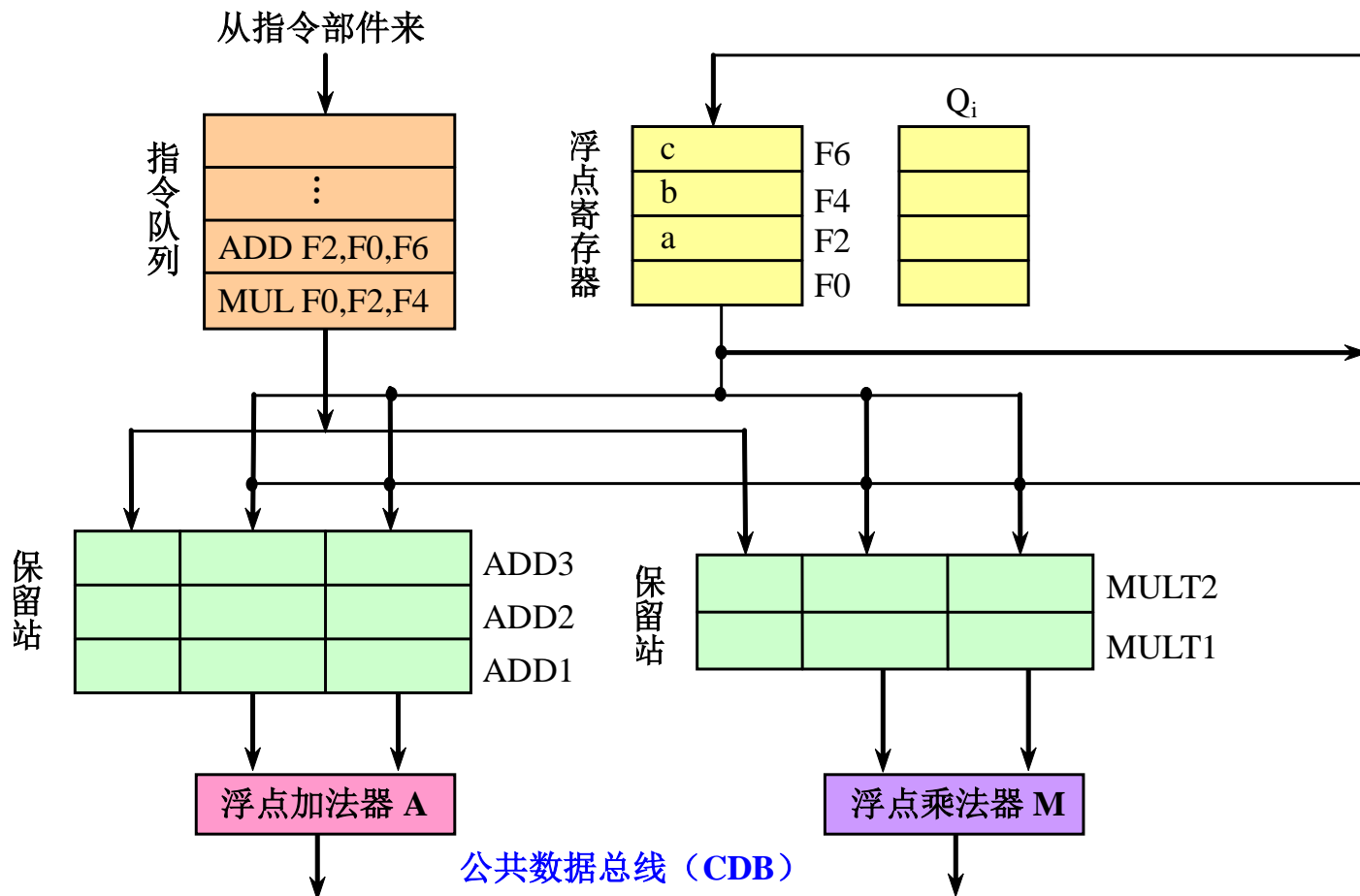
- 浮点加法器完成加法和减法操作
- 浮点乘法器完成乘法和除法操作

5. 在Tomasulo算法中，寄存器换名是通过保留站和流出逻辑来共同完成的。

- 当指令流出时，如果其操作数还没有计算出来，则将该指令中相应的寄存器号换名为将产生这个操作数的保留站的标识。
- 指令流出到保留站后，其操作数寄存器号或者换成了数据本身（如果该数据已经就绪），或者换成了保留站的标识，不再与寄存器有关系。

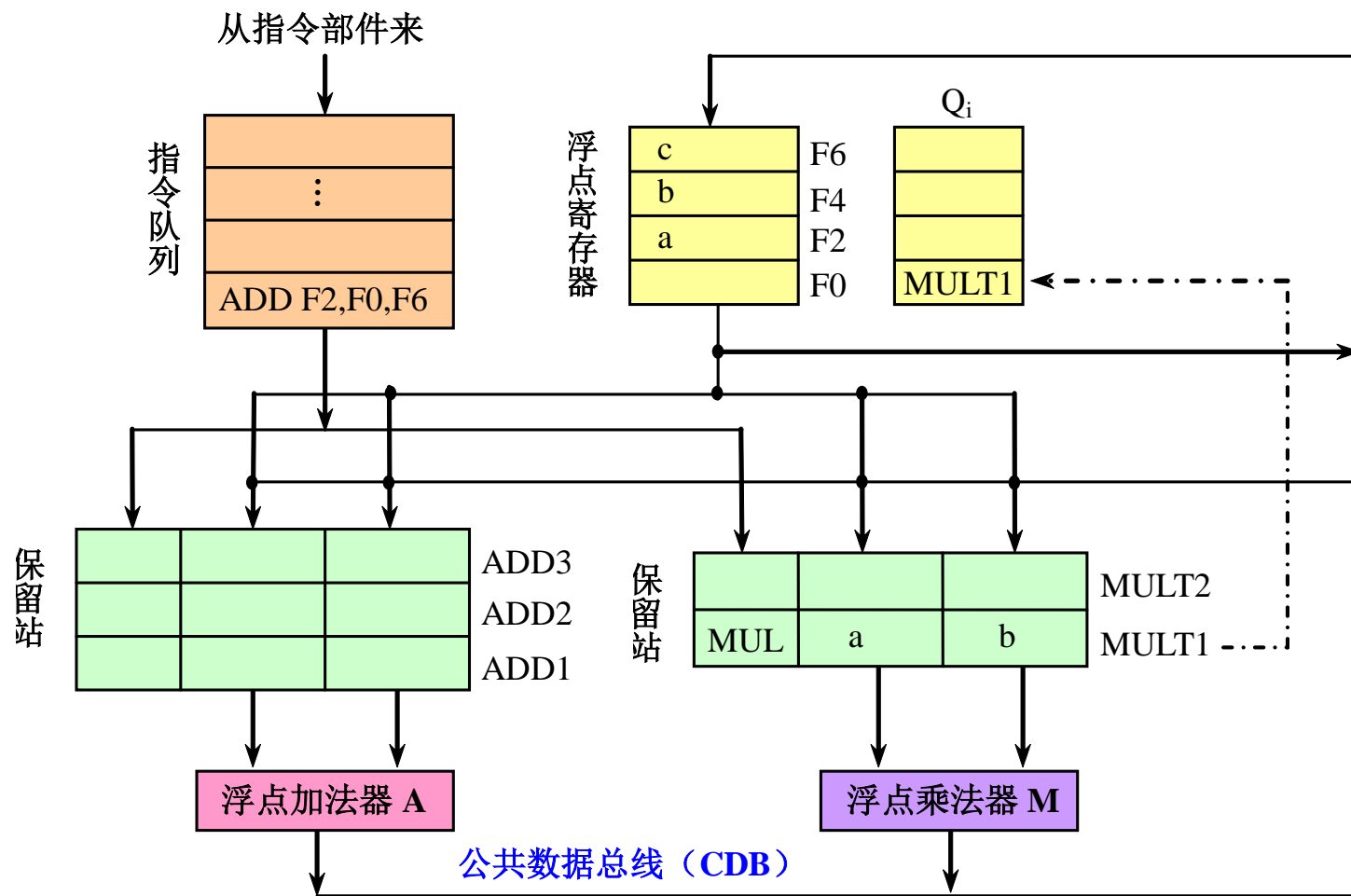
5.3 指令的动态调度

6. 通过一个简单的例子来说明Tomasulo算法的基本思想



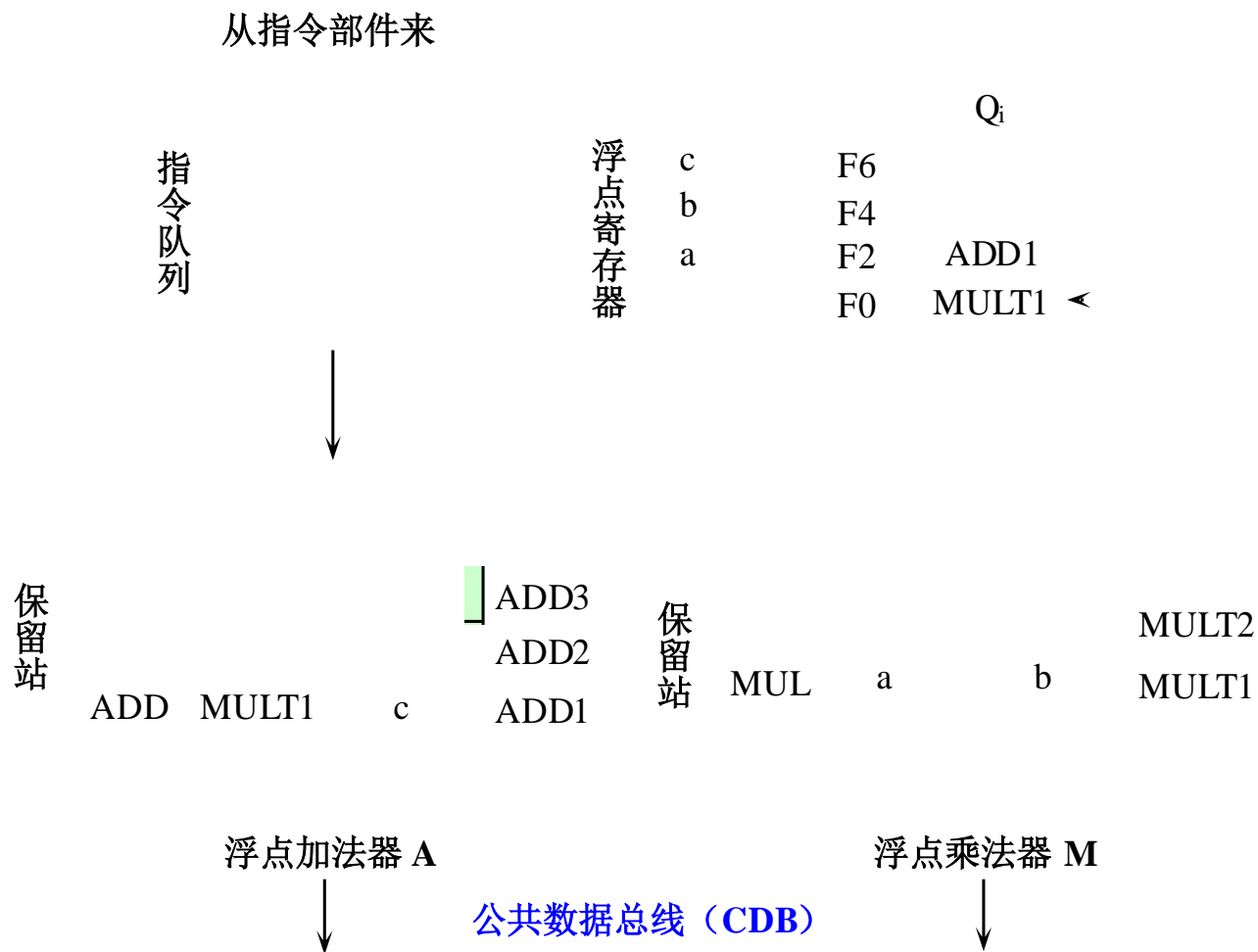
(a)

5.3 指令的动态调度



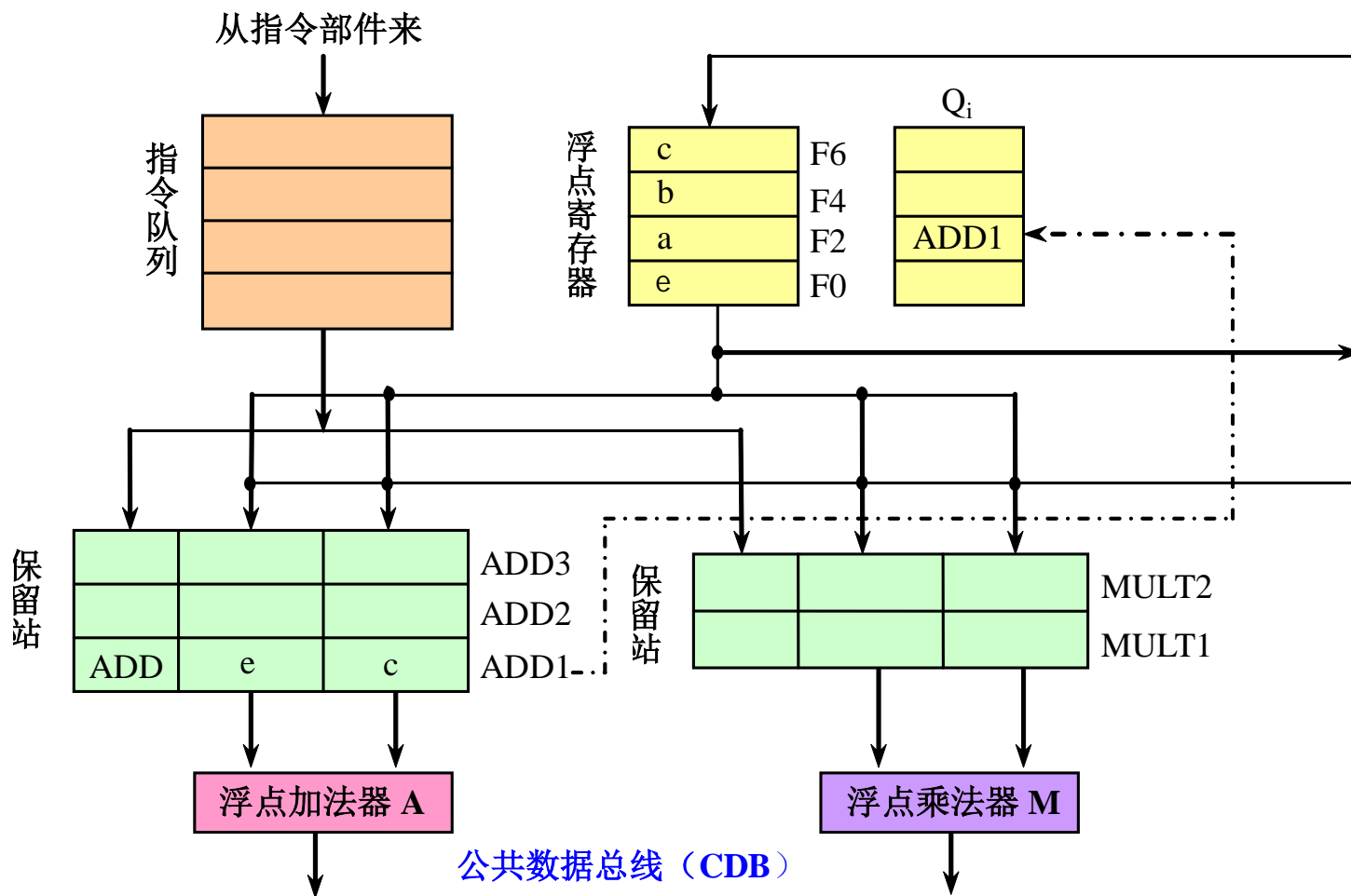
(b)

5.3 指令的动态调度



(c)

5.3 指令的动态调度



(d)

5.3 指令的动态调度

7. Tomasulo算法具有以下两个特点：

- 冲突检测和指令执行控制是分布的。

每个功能部件的保留站中的信息决定了什么时候指令可以在该功能部件开始执行。

- 计算结果通过CDB直接从产生它的保留站传送到所有需要它的功能部件，而不用经过寄存器。

8. 指令执行的步骤

使用Tomasulo算法的流水线需3段：

流出、执行、写结果

5.3 指令的动态调度

- 流出：从指令队列的头部取一条指令。
 - 如果该指令的操作所要求的保留站有空闲的，就把该指令送到该保留站（设为 r ）。
 - 如果其操作数在寄存器中已经就绪，就将这些操作数送入保留站 r 。
 - 如果其操作数还没有就绪，就把将产生该操作数的保留站的标识送入保留站 r 。
 - 一旦被记录的保留站完成计算，它将直接把数据送给保留站 r 。
(寄存器换名和对操作数进行缓冲，消除WAR冲突)
 - 完成对目标寄存器的预约工作
(消除了WAW冲突)
 - 如果没有空闲的保留站，指令就不能流出。
(发生了结构冲突)

5.3 指令的动态调度

➤ 执行

- 当两个操作数都就绪后，本保留站就用相应的功能部件开始执行指令规定的操作。
- **load**和**store**指令的执行需要两个步骤：
 - 计算有效地址（要等到基地址寄存器就绪）
 - 把有效地址放入**load**或**store**缓冲器

➤ 写结果

- 功能部件计算完毕后，就将计算结果放到**CDB**上，所有等待该计算结果的寄存器和保留站（包括**store**缓冲器）都同时从**CDB**上获得所需要的数据。

5.3 指令的动态调度

9. 每个保留站有以下7个字段：

- **Op**：要对源操作数进行的操作
- **Qj, Qk**：将产生源操作数的保留站号
 - 等于0表示操作数已经就绪且在Vj或Vk中，或者不需要操作数。
- **Vj, Vk**：源操作数的值
 - 对于每一个操作数来说，V或Q字段只有一个有效。
 - 对于load来说，Vk字段用于保存偏移量。
- **Busy**：为“yes”表示本保留站或缓冲单元“忙”
- **A**：仅load和store缓冲器有该字段。开始是存放指令中的立即数字段，地址计算后存放有效地址。

10. 寄存器状态表

➤ **Qi**：寄存器状态表

- 每个寄存器在该表中有对应的一项，用于存放将把结果写入该寄存器的保留站的站号。
- 为0表示当前没有正在执行的指令要写入该寄存器，也即该寄存器中的内容就绪。

5.3 指令的动态调度

5.3.3.2 举例

例5.2 对于下述指令序列，给出当第一条指令完成并写入结果时，Tomasulo算法所用的各信息表中的内容。

L. D F6, 34 (R2)

L. D F2, 45 (R3)

MUL. D F0, F2, F4

SUB. D F8, F2, F6

DIV. D F10, F0, F6

ADD. D F6, F8, F2

5.3 指令的动态调度

当采用Tomasulo算法时，在上述给定的时刻，
保留站、load缓冲器以及寄存器状态表中的内容。

指 令		指令状态表		
		流出	执行	写结果
L.D	F6 , 34(R2)	√	√	√
L.D	F2 , 45(R3)	√	√	
MUL.D	F0 , F2 , F4	√		
SUB.D	F8 , F6 , F2	√		
DIV.D	F10 , F0 , F6			
ADD.D	F6 , F8 , F2			

名称	保留站						
	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	no						
Load2	yes	L. D					45+Regs[R3]
Add1	yes	SUB. D		Mem[34+Regs[R2]]	Load2		
Add2	no						
Add3	no						
Mult1	yes	MUL. D		Reg[F4]	Load2		
Mult2	no						

	寄存器状态表							
	F0	F2	F4	F6	F8	F10	...	F30
Qi	Mult1	Load2			Add1		...	

5.3 指令的动态调度

Tomasulo算法具有两个主要的优点：

➤ 冲突检测逻辑是分布的

（通过保留站和CDB实现）

- 如果有多条指令已经获得了一个操作数，并同时在等待同一运算结果，那么这个结果一产生，就可以通过CDB同时播送给所有这些指令，使它们可以同时执行。

➤ 消除了WAW冲突和WAR冲突导致的停顿

使用保留站进行寄存器换名，并且在操作数一旦就绪就将之放入保留站。

5.3 指令的动态调度

例5.3 对于例5.2中的代码，假设各种操作的延迟为：

load: 1个时钟周期

加法: 2个时钟周期

乘法: 10个时钟周期

除法: 40个时钟周期

给出MUL.D指令准备写结果时各状态表的内容。

解：MUL.D指令准备写结果时各状态表的内容如下图所示。

5.3 指令的动态调度

指令		指令状态表		
		流出	执行	写结果
L.D	F6 , 34(R2)	√	√	√
L.D	F2 , 45(R3)	√	√	√
MUL.D	F0 , F2 , F4	√	√	
SUB.D	F8 , F6 , F2	√	√	√
DIV.D	F10, F0, F6	√		
ADD.D	F6 , F8 , F2	√	√	√

名称	保留站						
	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	no						
Load2	yes						
Add1	yes						
Add2	yes						
Add3	no						
Mult1	yes	MUL. D	Mem[45+Regs[R3]]	Reg[F4]			
Mult2	yes	DIV. D		Mem[34+Regs[R2]]	Mult1		

	寄存器状态表							
	F0	F2	F4	F6	F8	F10	...	F30
Qi	Mult1					Mult2	...	

5.3 指令的动态调度

5.3.3.3 具体算法

各符号的意义

- **r**: 分配给当前指令的保留站或者缓冲器单元编号;
- **rd**: 目标寄存器编号;
- **rs、rt**: 操作数寄存器编号;
- **imm**: 符号扩展后的立即数;
- **RS**: 保留站;
- **result**: 浮点部件或load缓冲器返回的结果;
- **Qi**: 寄存器状态表;
- **Regs[]**: 寄存器组;

MUL.D F4, F0, F2

↑ ↑ ↑

rd rs rt

i j k

5.3 指令的动态调度

- 与rs对应的保留站字段： V_j, Q_j
- 与rt对应的保留站字段： V_k, Q_k
- Q_i, Q_j, Q_k 的内容或者为0，或者是一个大于0的整数。
 - Q_i 为0表示相应寄存器中的数据就绪。
 - Q_j, Q_k 为0表示保留站或缓冲器单元中的 V_j 或 V_k 字段中的数据就绪。
 - 当它们为正整数时，表示相应的寄存器、保留站或缓冲器单元正在等待结果。

1. 指令流出

➤ 浮点运算指令

进入条件：有空闲保留站（设为 r ）

操作和状态表内容修改：

```
if (Qi[rs] ≠ 0)                // 第一操作数没就绪
    { RS[r].Qj ← Qi[rs] }; //没就绪, 寄存器换名, 即将产生该操作
                             数的保留站的编号放入当前保留站的Qj。编号是一个大于0的整数。
else { RS[r].Vj ← Regs[rs] ; //第一操作数就绪。把寄存器rs
                             //中的操作数取到当前保留站的Vj。
        RS[r].Qj ← 0 }         //置Qj为0, 表示当前保留站的Vj中
                             //的操作数就绪。
```

```

if (Qi[rt] ≠ 0)                                //第二操作数未就绪

{ RS[r].Qk ← Qi[rt] ;                          //寄存器换名，即把将产生该操作数的
    保留站的编号放入当前保留站的Qk。该编号是一个大于0的整数。

else { RS[r].Vk ← Regs[rt] ;                  //第二操作数就绪。把寄存器rt中
    //的操作数取到当前保留站的Vk。

    RS[r].Qk ← 0 }                            //置Qk为0，表示当前保留站的Vk中
    //的操作数就绪。

RS[r].Busy ← yes;                             //置当前保留站为“忙”

RS[r].Op ← Op;                                //设置操作码

Qi[rd] ← r;                                   //把当前保留站的编号r放入rd所对应
    //的寄存器状态表项，以便rd将来接收结果。

```

L.D F2, 45 (R3)

↑ ↑ ↑
rt imm rs
k j

➤ load和store指令

进入条件：缓冲器有空闲单元（设为r）

操作和状态表内容修改：

```
if (Qi[rs] ≠ 0)                      //第一操作数没就绪
    { RS[r].Qj ← Qi[rs] }            //寄存器换名，即把将产生该操作数
                                   //的保留站的编号存入当前缓冲器单元的Qj。
else
    { RS[r].Vj ← Regs[rs];           // 第一操作数就绪，把寄存器rs中的
                                   // 操作数取到当前缓冲器单元的Vj
      RS[r].Qj ← 0 } ;               // 置Qj为0，表示当前缓冲器单元的Vj
                                   // 中的操作数就绪。
```

L.D F2, 45 (R3)

↑ ↑ ↑
rt imm rs
k j

```
RS[r].Busy ← yes;           // 置当前缓冲器单元为“忙”
```

```
RS[r].A ← Imm;           // 把符号位扩展后的偏移量放入
                           // 当前缓冲器单元的A
```

对于load指令:

```

Qi[rt] ← r;           // 把当前缓冲器单元的编号r放入
                        // load指令的目标寄存器rt所对应的寄存器
                        // 状态表项，以便rt将来接收所取的数据。

```

S.D F3, 40 (R4)

↑ ↑ ↑
rt imm rs
k j

对于store指令:

```
if (Qi[rt] ≠ 0)           //要存储的数据尚未就绪
{ RS[r].Qk ← Qi[rt] }     //寄存器换名，即把将产生该数据的
                           //保留站的编号放入当前缓冲器单元的Qk。

else

{ RS[r].Vk ← Regs[rt];    // 该数据就绪，把它从寄存器rt取到
                           // store缓冲器单元的Vk

  RS[r].Qk ← 0 };         // 置Qk为0，表示当前缓冲器单元的Vk
                           // 中的数据就绪。
```

2. 执行

➤ 浮点操作指令

- ❑ 进入条件： $(RS[r].Qj = 0)$ 且 $(RS[r].Qk = 0)$;
// 两个源操作数就绪
- ❑ 操作和状态表内容修改： 进行计算，产生结果。

➤ load/store指令

- ❑ 进入条件： $(RS[r].Qj = 0)$ 且 r 成为load/store缓冲队列的头部
- ❑ 操作和状态表内容修改：

$RS[r].A \leftarrow RS[r].Vj + RS[r].A;$ //计算有效地址

对于load指令，在完成有效地址计算后，还要进行：

从 $Mem[RS[r].A]$ 读取数据； //从存储器中读取数据

3. 写结果

➤ 浮点运算指令和load指令

进入条件：保留站r执行结束，且CDB就绪。

操作和状态表内容修改：

$\forall x$ (if ($Qi[x] = r$)	// 对于任何一个正在等该结果
	// 的浮点寄存器x
{ Regs[x] \leftarrow result;	// 向该寄存器写入结果
$Qi[x] \leftarrow 0$ };	// 把该寄存器的状态置为数据就绪
$\forall x$ (if ($RS[x].Qj = r$)	// 对于任何一个正在等该结果
	// 作为第一操作数的保留站x
{ $RS[x].Vj \leftarrow$ result;	// 向该保留站的Vj写入结果
$RS[x].Qj \leftarrow 0$ };	// 置Qj为0，表示该保留站的
	// Vj中的操作数就绪

$\forall x$ (if (RS[x].Qk = r) // 对于任何一个正在等该结果作为
 // 第二操作数的保留站x
 { RS[x].Vk \leftarrow result; // 向该保留站的Vk写入结果
 RS[x].Qk \leftarrow 0 } ; // 置Qk为0, 表示该保留站的Vk中的
 // 操作数就绪。
 RS[r].Busy \leftarrow no; // 释放当前保留站, 将之置为空闲状态。

➤ store指令

进入条件: 保留站r执行结束, 且RS[r].Qk = 0
 // 要存储的数据已经就绪

操作和状态表内容修改:

Mem[RS[r].A] \leftarrow RS[r].Vk // 数据写入存储器, 地址由store
 // 缓冲器单元的A字段给出。

RS[r].Busy \leftarrow no; // 释放当前缓冲器单元, 将之置为空闲状态。