

第5章 指令级并行及其开发

- 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

- **指令级并行**：指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。

(ILP: Instruction-Level Parallelism)

- 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**：如何利用各种技术来开发更多的指令级并行，以最终提高处理器的运算能力（硬件的方法）

5.1 指令级并行的概念

1. 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

2. 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：

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

- 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)

□ 举例：

请为下列表达式生成没有暂停的指令序列：

$$A = B + C ;$$

$$D = E - F ;$$

假设载入延迟为1个时钟周期。

调度前的代码	调度后的代码
LD Rb, B	LD Rb, B
LD Rc, C	LD Rc, C
DADD Ra, Rb, Rc	LD Re, E
SD Ra, A	DADD Ra, Rb, Rc
LD Re, E	LD Rf, F
LD Rf, F	SD Ra, A
DSUB Rd, Re, Rf	DSUB Rd, Re, Rf
SD Rd, D	SD Rd, D

- 基本程序块：一串连续的代码除了入口和出口以外，没有其他的分支指令和转入点
- 程序平均每4~7条指令就会有一个分支
- 跨块调度

5.2 相关与指令级并行

1. 相关与流水线冲突

- **相关：**两条指令之间存在某种依赖关系。

如果两条指令相关，则它们就有可能不能在流水线中重叠执行或者只能部分重叠执行。

- **相关有3种类型**

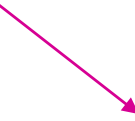


- 数据相关（也称真数据相关）
- 名相关
- 控制相关

2. 数据相关

- 对于两条指令*i*（在前，下同）和*j*（在后，下同），如果下述条件之一成立，则称指令*j*与指令*i*数据相关。
 - 指令*j*使用指令*i*产生的结果；
 - 指令*j*与指令*k*数据相关，而指令*k*又与指令*i*数据相关。
- 数据相关具有传递性。

数据相关反映了数据的流动关系，即如何从其产生者流动到其消费者。

例如：下面这一段代码存在数据相关。（数组增值）

Loop:	L. D	F0, 0 (R1)	// F0为数组元素
			
	ADD. D	F4, F0, F2	// 加上F2中的值
			
	S. D	F4, 0 (R1)	// 保存结果
	DADDIU	R1, R1, #-8	// 数组指针递减8个字节
			
	BNE	R1, R2, Loop	// 如果R1≠R2，则分支

- 如果两条指令之间有数据相关，那么它们就不能同时执行或是完全重叠执行
- 同时执行这些指令会造成正在流水的处理机检测到这种冲突并插入暂停，从而减少甚至取消指令之间的重叠
- 指令序列中存在的數據相关反映出产生该指令序列的程序源代码的相关关系

相关性是程序的一个特性，
是否一个相关会导致实际的冲突，是否该冲突会造成暂停，
这是流水线结构的基本特性。

Loop: L.D F0, 0 (R1) // F0为数组元素

ADD.D F4, F0, F2 // 加上F2中的值

S.D F4, 0 (R1) // 保存结果

DADDIU R1, R1, #-8 // 数组指针递减8个字节

BNE R1, R2, Loop // 如果R1≠R2, 则分支

- 如果分支检测移到了ID流水段，相关会造成1次暂停
- 如果分支检测仍在EX流水段，这个相关就不会引起暂停

➤ 数据相关性会限制可以开发的指令级并行性

➤ 解决方法

- 保持相关关系但避免冲突

- 对代码进行调度是保持相关关系并避免冲突常用的一个最基本方法

- 通过变换代码来消除相关关系

➤ 数据相关的检测

调度前的代码		调度后的代码	
LD	Rb, B	LD	Rb, B
LD	Rc, C	LD	Rc, C
DADD	Ra, Rb, Rc	LD	Re, E
SD	Ra, A	DADD	Ra, Rb, Rc
LD	Re, E	LD	Rf, F
LD	Rf, F	SD	Ra, A
DSUB	Rd, Re, Rf	DSUB	Rd, Re, Rf
SD	Rd, D	SD	Rd, D

数据的流动是经过寄存器时，相关的检测比较直

接

数据的流动是经过存储器时，检测比较复杂。

相同形式的地址其有效地址未必相同；

形式不同的地址其有效地址却可能相同。

2. 名相关

- **名：**指令所访问的寄存器或存储器单元的名称。
- 如果两条指令使用相同的名，但是它们之间并没有数据流动，则称这两条指令存在**名相关**。

➤ 指令j与指令i之间的名相关有两种：

- **反相关：**如果指令j写的名与指令i读的名相同，则称指令i和j发生了反相关。

指令j写的名=指令i读的名

必须保证指令的原来顺序，以确保i能读到正确的值。

- **输出相关：**如果指令j和指令i写相同的名，则称指令i和j发生了输出相关。

指令j写的名=指令i写的名

操作顺序必须得到保证，以使最后写入的值得以保存

- 名相关的两条指令之间并没有数据的传送。
 - 如果一条指令中的名改变了，并不影响另外一条指令的执行。
 - 换名技术
 - **换名技术：**通过改变指令中操作数的名来消除名相关。
 - 对于寄存器操作数进行换名称为**寄存器换名**。
- 既可以用编译器静态实现，也可以用硬件动态完成。

例如：考虑下述代码：

DIV. D F2, F8, F4

ADD. D F8, F0, F12

SUB. D F10, F8, F14

DIV. D和ADD. D存在反相关。

进行寄存器换名（F8换成S）后，变成：

DIV. D F2, S, F4

ADD. D S, F0, F12

SUB. D F10, S, F14

名字相关的指令是可以同时执行或进行重新排序的

- 当指令之间存在相关性，而且它们的执行时间相近时，可能会在流水线中重叠操作，或者指令的重新排序改变了有相关的操作数的访问顺序，这时**数据冲突**就发生了。

程序顺序：

由源程序确定的在完全串行方式下指令的执行顺序
只有在可能会导致错误的情况下，才保持程序顺序。

3. 数据冲突

- 根据指令在流水线中要保留的读访问和写访问的顺序，可以将数据冲突分为3种类型。

考虑两条指令i和j，且i在j之前进入流水线，可能发生的数据冲突有：

- 写后读冲突（RAW）

在 i 写入之前，j 先去读。

j 读出的内容是错误的。

这是最常见的一种数据冲突，它对应于真数据相关。

□ 写后写冲突 (WAW)

在 *i* 写入之前，*j* 先写。

最后写入的结果是 *i* 的。错误！

这种冲突对应于输出相关。

写后写冲突仅发生在这样的流水线中：

- 流水线中不只一个段可以进行写操作；
- 指令被重新排序了。

- 读后写冲突（WAR）

在 *i* 读之前，*j* 先写。

i 读出的内容是错误的！

由反相关引起。

这种冲突仅发生在这样的情况下：

- 有些指令的写结果操作提前了，而且有些指令的读操作滞后了；
- 指令被重新排序了。

4. 控制相关

- **控制相关**是指由分支指令引起的相关。
 - 为了保证程序应有的执行顺序，必须严格按控制相关确定的顺序执行。
- 典型的程序结构是“if-then”结构。
- 例：

```
if p1 {  
    S1;  
};  
S;  
if p2 {  
    S2;  
};
```

```
if p1 {  
    S1;  
};  
S;  
if p2 {  
    S2;  
};
```

➤ 控制相关带来了以下两个限制：

- 与一条分支指令控制相关的指令不能被移到该分支之前。否则这些指令就不受该分支控制了。
对于上述的例子，**then** 部分中的指令不能移到**if**语句之前。
- 如果一条指令与某分支指令不存在控制相关，就不能把该指令移到该分支之后。
对于上述的例子，不能把**S**移到**if**语句的**then** 部分中。

5.2 相关与指令级并行

1. 控制相关并不是一个必须严格保持的关键属性。
2. 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。
 - 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。

DADDU R2, R3, R4

BEQZ R2, L1

LW R1, 0(R2)

L1:

忽略控制相关而把LW指令移到分支指令之前，则有可能引起内存保护异常的错误。

5.2 相关与指令级并行

- **数据流**：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。

```
DADDU    R1, R2, R3↵  
BEQZ     R4, L↵  
DSUBU    R1, R5, R6↵  
L:        ...↵  
OR       R7, R1, R8↵
```


5.3 指令的动态调度

➤ 静态调度

- 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

- 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。

5.3 指令的动态调度

- 动态调度的优点：

- 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. 到目前为止我们所使用流水线的最大的局限性:

- 指令是按序流出和按序执行的
- 考虑下面一段代码:

DIV. D F4, F0, F2

ADD. D F10, F4, F6

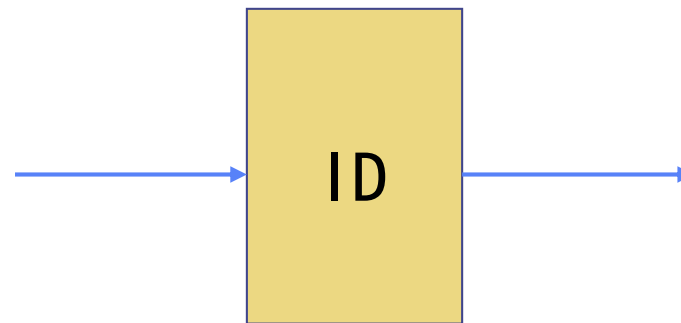
SUB. D F12, F6, F14

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：



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。

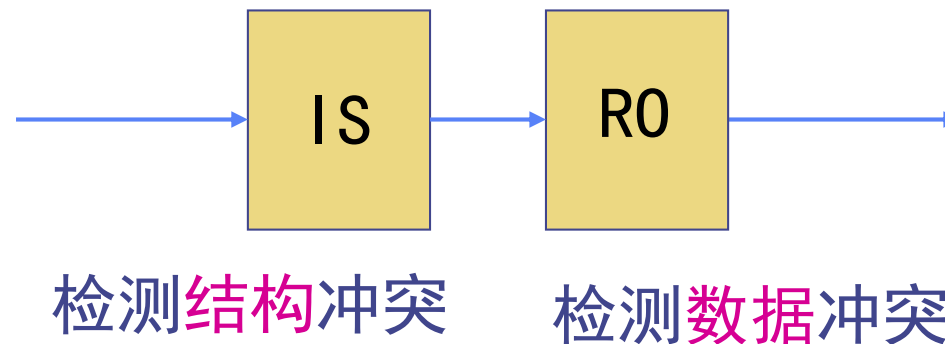
乱序执行

- 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

2. 为了支持乱序执行，我们将5段流水线的译码阶段再分为两个阶段：

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



5.3 指令的动态调度

3. 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
存在反相关 {	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

4. 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. 指令乱序完成带来的最大问题：

异常处理比较复杂

（精确异常处理、不精确异常处理）

- 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。

5.3 指令的动态调度

5.3.2 记分牌动态调度算法

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

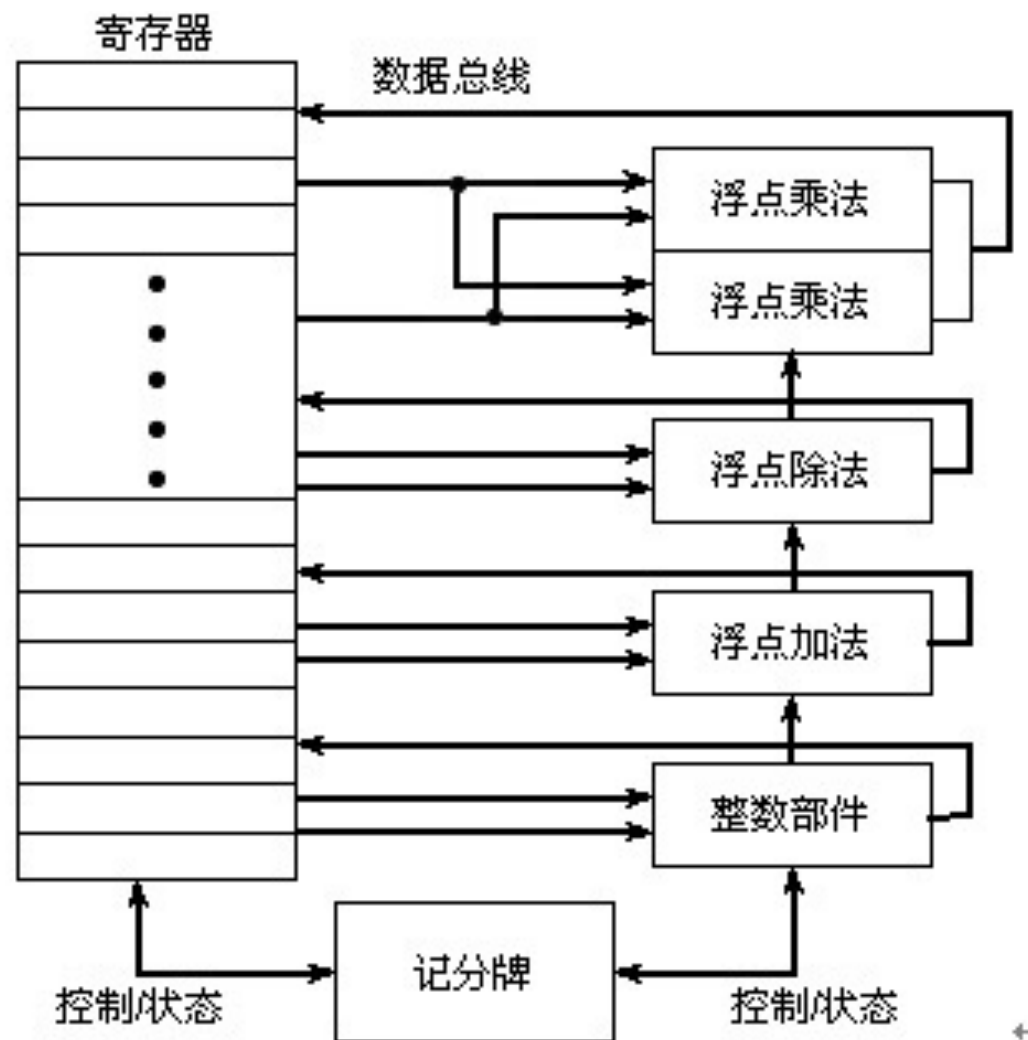
- 记分牌的**目标**：在没有结构冲突时，尽可能早地执行没有数据冲突的指令，实现每个时钟周期执行一条指令。
- 要发挥指令乱序执行的好处，必须有多条指令同时处于执行阶段。
 - CDC 6600具有**16**个独立的功能部件
 - **4**个浮点部件
 - **5**个访存部件
 - **7**个整数操作部件
- 假设
 - 所考虑的处理器有**2**个乘法器、**1**个加法器、**1**个除法部件和**1**个整数部件。
 - 整数部件用来处理所有的存储器访问、分支处理和整数操作。

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。

5.3 指令的动态调度



5.3 指令的动态调度

- 每条指令的执行过程分为4段（主要看浮点操作数，因而不考虑涉及访问内存的阶段）
 - 流出

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。

解决了结构冲突与**WAW**冲突

该指令若无法流出，也就阻止了后面指令的流出。

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。

5.3 指令的动态调度

□ 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。

➤ 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”。

5.3 指令的动态调度

- 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。
 - 如果当前正在运行的指令都不以它为目的寄存器, 则其相应项置为 “no”。
 - **Result**各项的初值为 “no” (全0)。

5.3 指令的动态调度

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

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。

5.3 指令的动态调度

例5.1 假设浮点流水线中

加法需2个时钟

乘法需10个时钟

除法需40个时钟

代码段和记分牌信息的起始

DIV.D准备写结果之前的记分牌

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

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

1. 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

$Fk[FU] \leftarrow S2;$ // 记录第二个源寄存器编号。

$Qj[FU] \leftarrow \text{Result}[S1];$ // 记录将产生第一个源操作数的部件。

$Qk[FU] \leftarrow \text{Result}[S2];$ // 记录将产生第二个源操作数的部件。

$Rj[FU] \leftarrow \text{not } Qj[FU];$ // 置第一个源操作数是否可用的标志。

如果 $Qj[FU]$ 为“no”，就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”。

$Rk[FU] \leftarrow \text{not } Qk[FU];$ // 置第二个源操作数是否可用的标志。

$\text{Result}[D] \leftarrow FU;$ // 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

■ 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。
- 反相关和输出相关。
 - 它们引起记分牌中更多的WAR和WAW冲突。

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)

5.3 指令的动态调度

1. 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

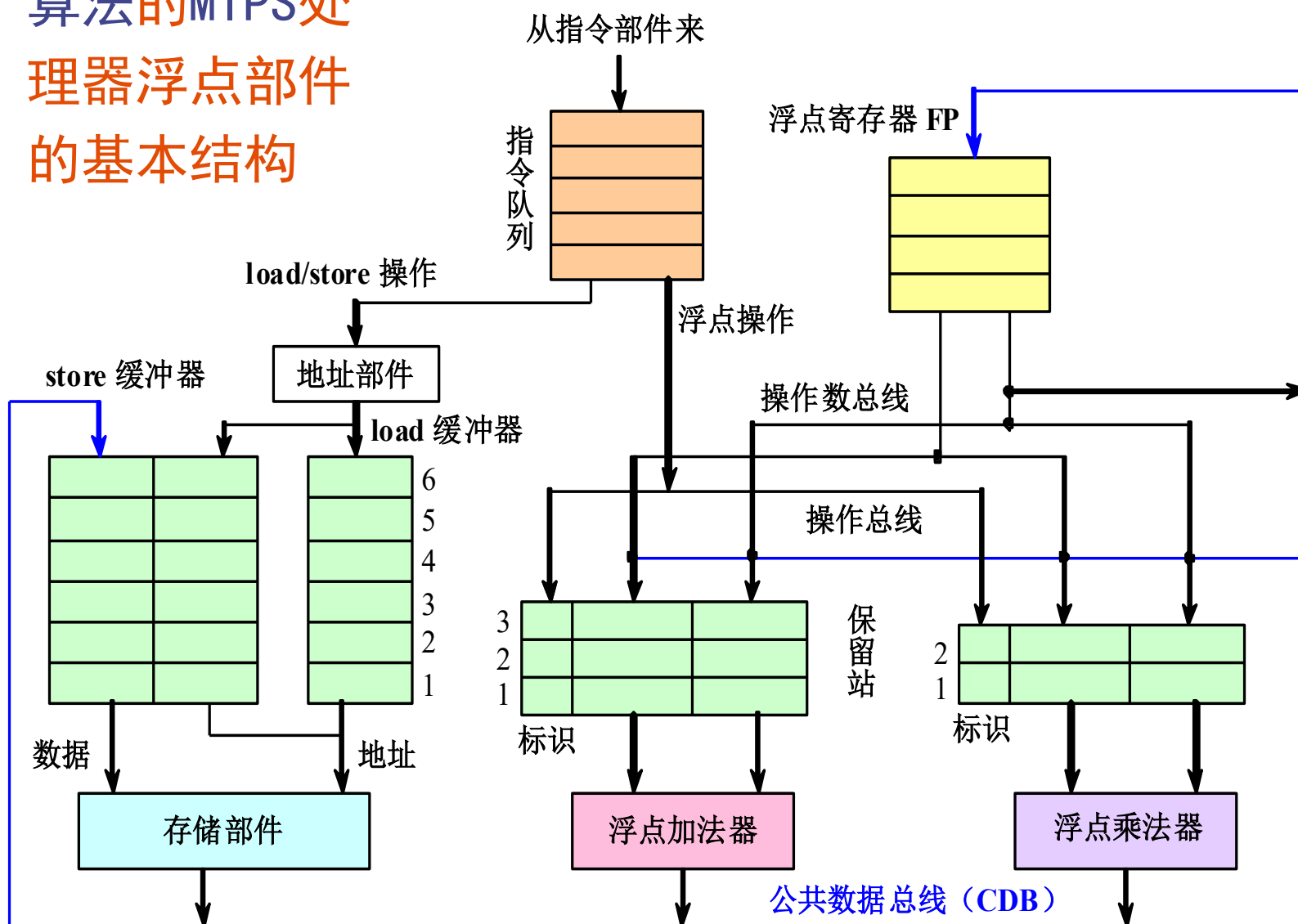
➤ 消除名相关

- 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	} 两个F6都换名为S
	S. D	S, 0 (R1)	
{ 两个F8都换名为T	SUB. D	T, F10, F14	
	MUL. D	F6, F10, T	

- 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。

5.3 指令的动态调度

➤ 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。
- 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。

5.3 指令的动态调度

➤ 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

➤ 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

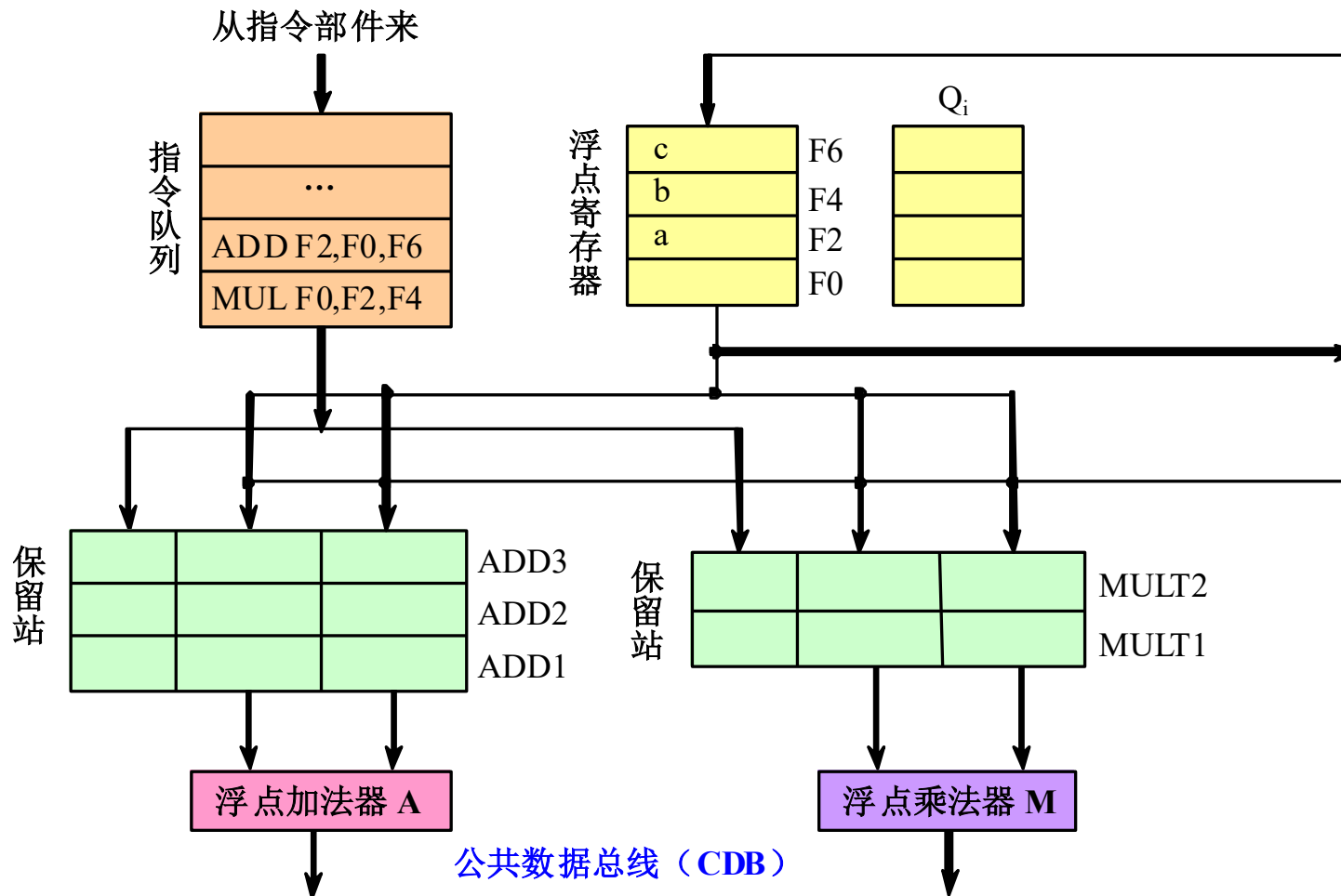
5.3 指令的动态调度

2. 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。

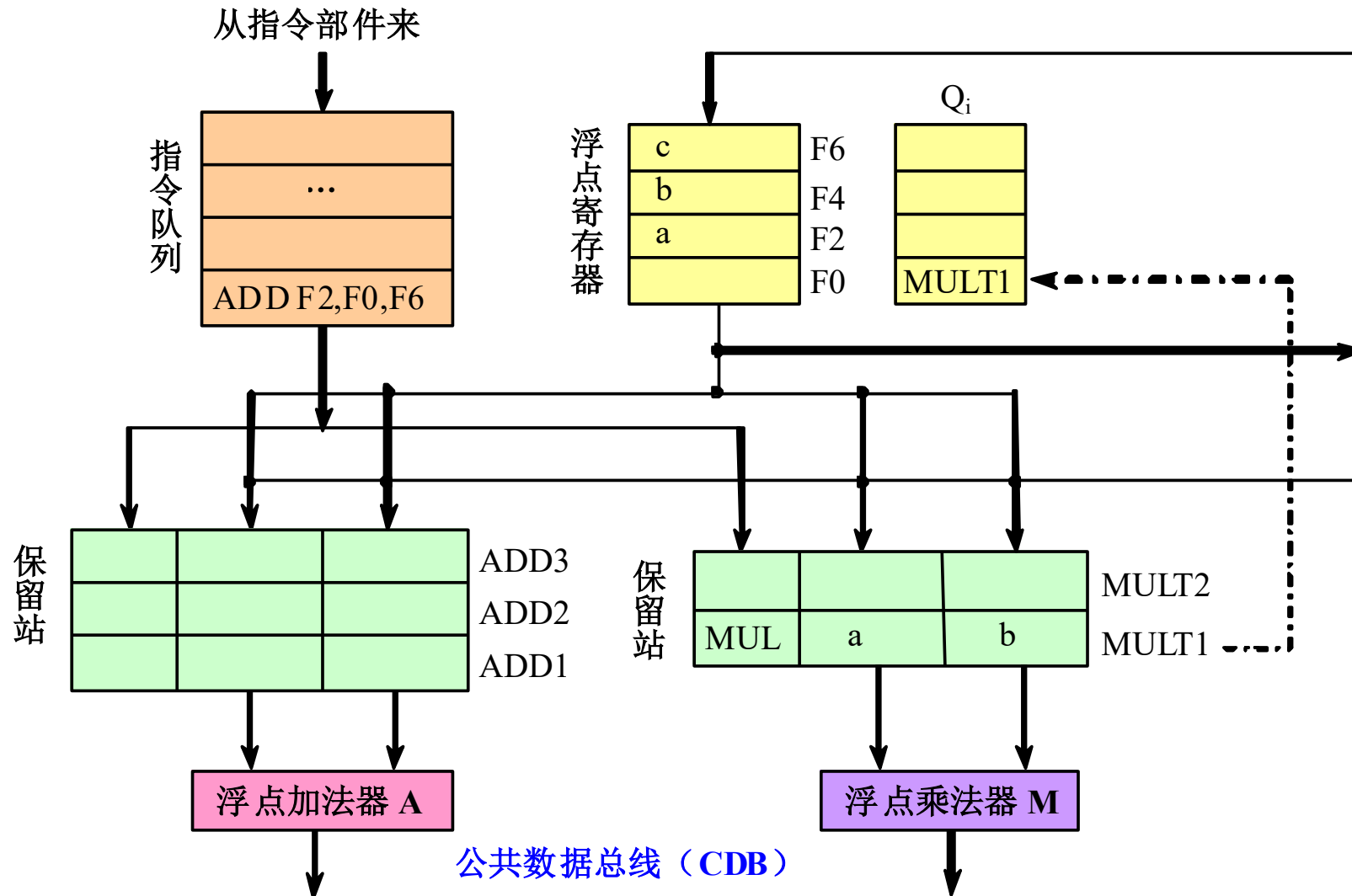
- 当指令流出时，如果其操作数还没有计算出来，则将该指令中相应的寄存器号换名为将产生这个操作数的保留站的标识。
- 指令流出到保留站后，其操作数寄存器号或者换成了数据本身（如果该数据已经就绪），或者换成了保留站的标识，不再与寄存器有关系。

5.3 指令的动态调度

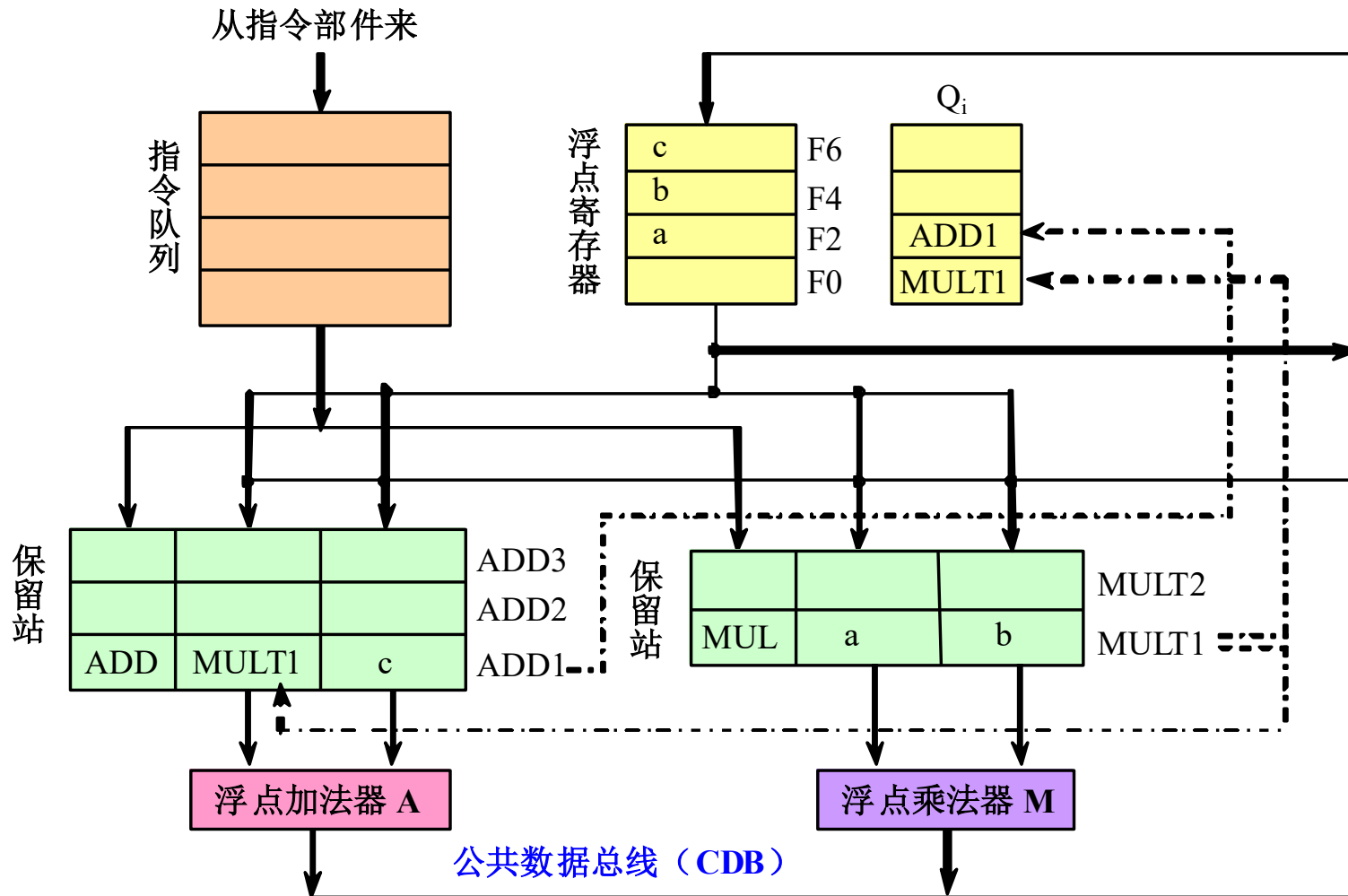
3. 通过一个简单的例子来说明Tomasulo算法的基本思想



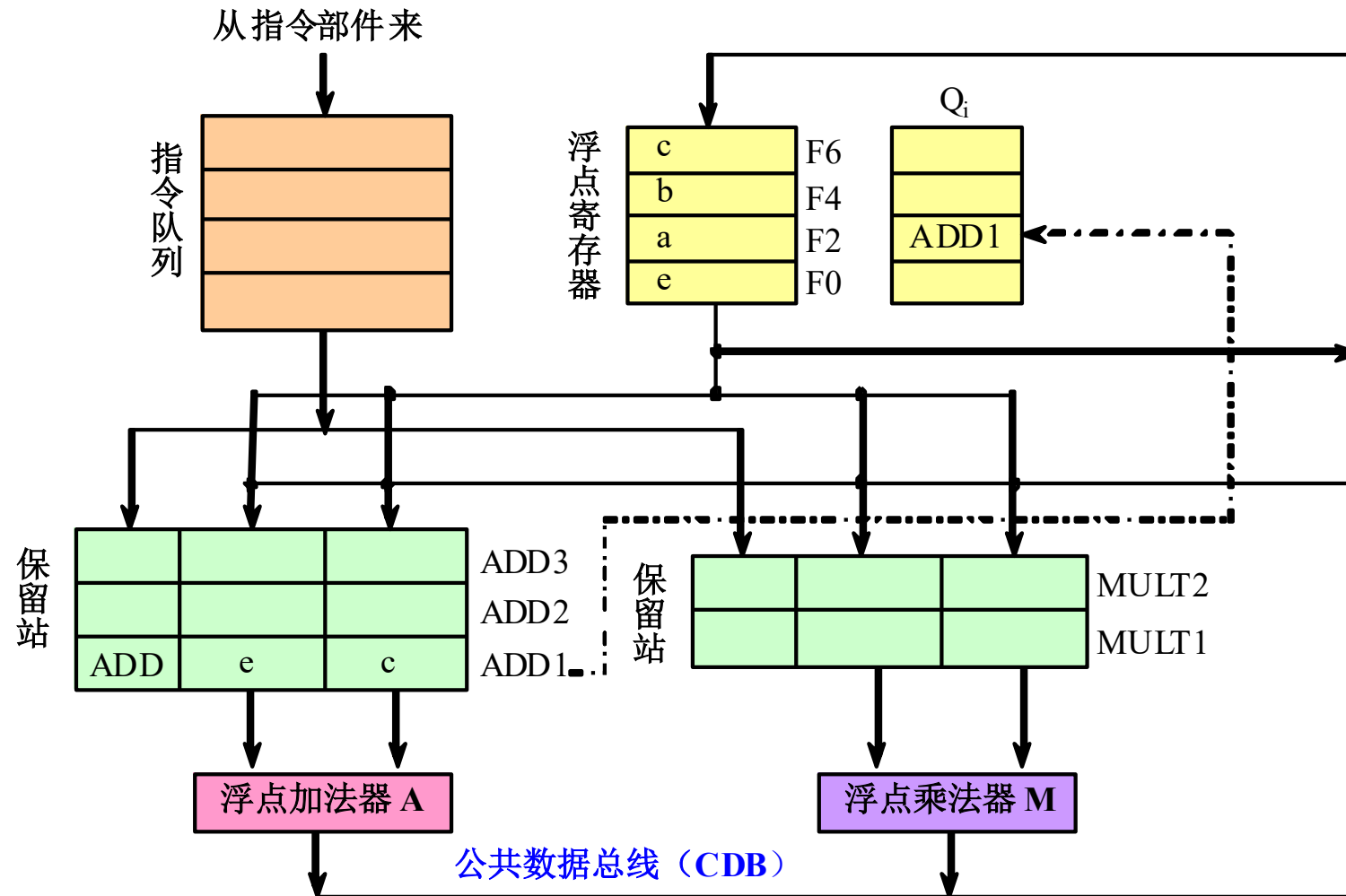
5.3 指令的动态调度



5.3 指令的动态调度



5.3 指令的动态调度



5.3 指令的动态调度

4. 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。

5. 指令执行的步骤

使用Tomasulo算法的流水线需3段：

- 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）

5.3 指令的动态调度

- 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

6. 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。

5.3 指令的动态调度

➤ 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

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	LD					45+Regs[R3]
Add1	yes	SUB		Mem[34+Regs[R2]]	Load2		
Add2	yes	ADD			Add1	Load2	
Add3	no						
Mult1	yes	MUL		Reg[F4]	Load2		
Mult2	yes	DIV		Mem[34+Regs[R2]]	Mult1		

	寄存器状态表							
	F0	F2	F4	F6	F8	F10	...	F30
Qi	Mult1	Load2		Add2	Add1	Mult2	...	

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	√	√	√

与记分牌的对比

↵

指 令↵	指令状态表↵			
	流出↵	读操作数↵	执行↵	<u>写结果</u> ↵
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↵	√↵	√↵	√↵	↵

1

MUL. D指令准备写结果时各状态表的内容

名称	保留站						
	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	no						
Load2	no						
Add1	no						
Add2	no						
Add3	no						
Mult1	yes	Mu1	Mem[45+Regs[R3]]	Reg[F4]			
Mult2	yes	DIV		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;

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。

5.3 指令的动态调度

符号说明： （举例）

MUL.D F4 , F0 , F2

 ↑ ↑ ↑

rd **rs** **rt**

i **j** **k**

L.D F2 , 45 (R3)

 ↑ ↑ ↑

rt **im** **rs**

i **m** **j**

S.D F3, 40 (R4)

 ↑ ↑ ↑

rt **im** **rs**

k **m** **j**

MUL.D

F4 , F0 , F2



rd rs rt

j k

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) // 检测第一操作数是否就绪

{ RS[r].Qj ← Qi[rs] }; //第一操作数没有就绪，进行寄存器
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Qj。
该编号是一个大于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)           // 检测第二操作数是否就绪
{ RS[r].Qk ← Qi[rt] ;      //第二操作数没有就绪，进行寄存器换
    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 im rs
i m j

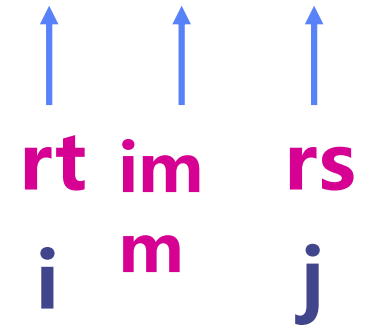
➤ load和store指令

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)           // 检测第一操作数是否就绪
    {RS[r].Qj ← Qi[rs] }   //第一操作数没有就绪，进行寄存器
                           // 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)



RS[r].Busy \leftarrow yes;

// 置当前缓冲器单元为“忙”

RS[r].A \leftarrow 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:

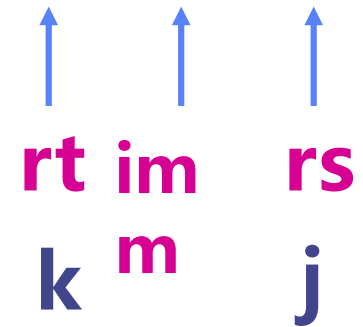
Qi[rt] \leftarrow r;

// 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)



对于store指令:

```
if (Qi[rt] ≠ 0)           // 检测要存储的数据是否就绪
{RS[r].Qk ← Qi[rt] }      //该数据尚未就绪，进行寄存器换名，
                           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。
```

1. 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

1. 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：

$\forall x \text{ (if (Qi}[x] = r)$	// 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
$\forall x \text{ (if (RS}[x].Qj = r)$	// 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。
-----------------------------	------------------------

1. Tomasulo算法的缺陷是需要大量的硬件

- 需要大量高速运转的存储器件及复杂的控制逻辑
- 使用单条完整的总线（CDB），所以反过来会限制性能的提高。即使可以增加CDB，但是增加CDB即意味着需要增加与流水线中所有硬件的连接，包括保留站

2. 对于单发射流水线，Tomasulo算法比起编译器调度所带来的有利之处，则会少于实现起来所需要的成本的增加

- 但是随着处理机在发射能力方面的提高及代码复杂度的增加，将使寄存器换名、动态调度、推测执行技术变得越来越重要

3. 对指令执行存在一个限制，如果流水中存在分支指令未完成，当前指令不能进入“执行”

5.4 动态分支预测技术

1. 所开发的ILP越多，控制相关的制约就越大，分支预测就要有更高的准确度。
2. 本节中介绍的方法对于每个时钟周期流出多条指令（若为n条，就称为n流出）的处理机来说非常重要。

因为：

- 在n-流出的处理机中，遇到分支指令的可能性增加了n倍
- 要给处理器连续提供指令，就需要准确地预测分支

5.4 动态分支预测技术

3. 动态分支预测：在程序运行时，根据分支指令过去的表现来预测其将来的行为。

- 如果分支行为发生了变化，预测结果也跟着改变。
- 有更好的预测准确度和适应性。

4. 分支预测的有效性取决于：

- 预测的准确性
- 预测正确和不正确两种情况下的分支开销

决定分支开销的因素：

- 流水线的结构
- 预测的方法
- 预测错误时的恢复策略等

5.4 动态分支预测技术

5. 采用动态分支预测技术的目的

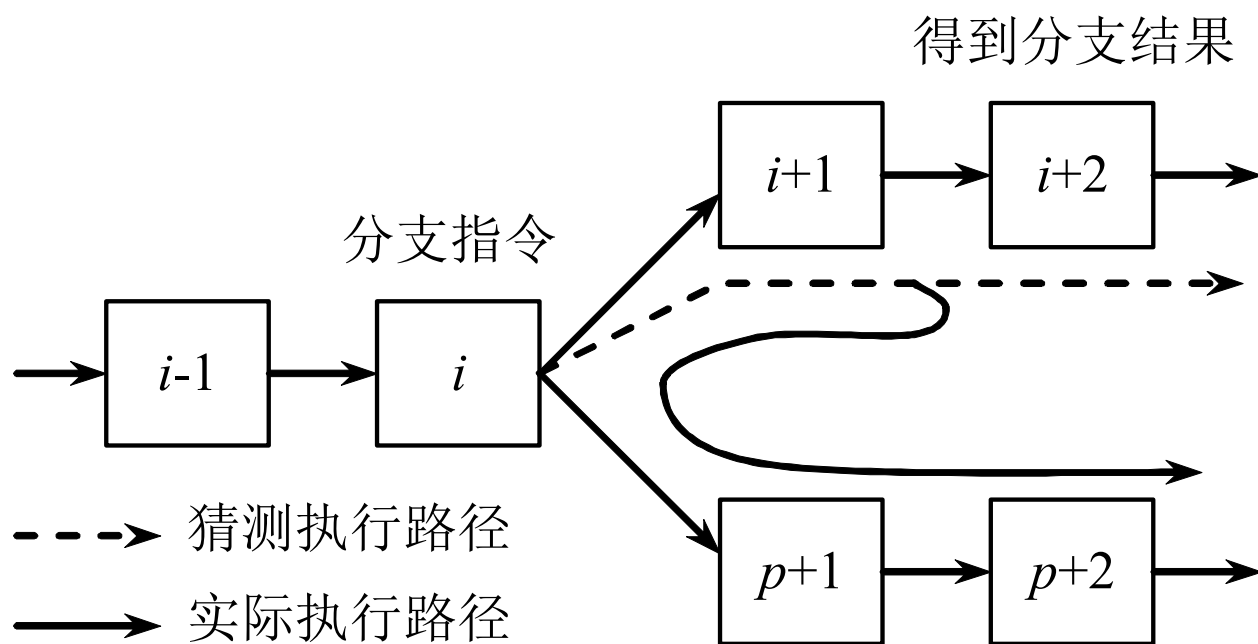
- 预测分支是否成功
- 尽快找到分支目标地址（或指令）
（避免控制相关造成流水线停顿）

6. 需要解决的关键问题

- 如何记录分支的历史信息，要记录哪些信息？
- 如何根据这些信息来预测分支的去向，甚至提前取出分支目标处的指令？

5.4 动态分支预测技术

7. 在预测错误时，要作废已经预取和分析的指令，恢复现场，并从另一条分支路径重新取指令。



5.4 动态分支预测技术

5.4.1 采用分支历史表 BHT

1. 分支历史表BHT (Branch History Table)

- 最简单的动态分支预测方法。
- 用BHT来记录分支指令最近一次或几次的执行情况（成功还是失败），并据此进行预测。

2. 只有1个预测位的分支预测

记录分支指令最近一次的历史，BHT中只需要1位二进制位。

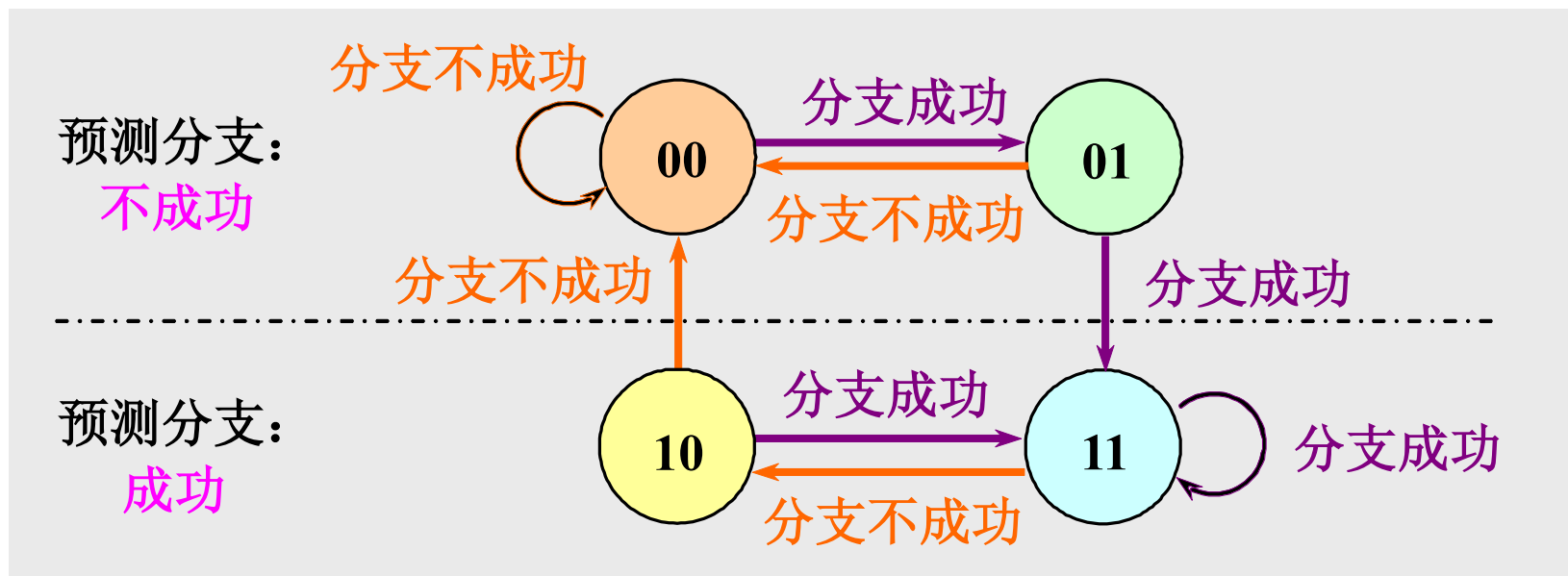
（最简单）

5.4 动态分支预测技术

3. 采用两位二进制位来记录历史

- 提高预测的准确度
- 研究表明：两位分支预测的性能与 n 位 ($n > 2$) 分支预测的性能差不多。

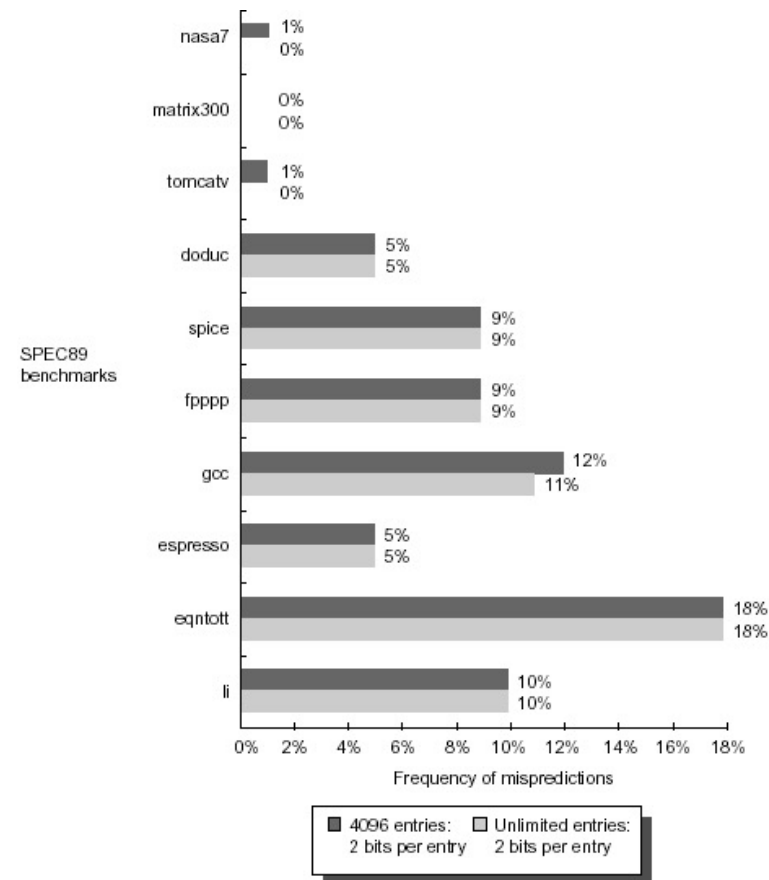
➤ 两位分支预测的状态转换如下所示：



- BHT可以跟分支指令一起存放在指令Cache中，也可以用一块专门的硬件来实现。
 - 可以作为一对指令位附于指令缓冲站的每一块中，并随指令一起读取。
 - 可以用存放指令地址的小的专用Cache来实现
 - 通过指令地址的低位访问BHT

5.4 动态分支预测技术

- 研究表明：对于SPEC89测试程序来说，具有大小为4K的BHT的预测准确率为82%~99%。



4. 高级分支预测技术（选看）

① 相关分支预测器

- ### 使用其它分支行为来进行预测的分支预测器

```
if (aa == 2)
    aa = 0;
if (bb == 2)
    bb = 0;
if (aa != bb) {
```

```

DSUBUIR3, R1, #2
BNEZ    R3, L1          ; 分支b1 (aa!=2)
DADD    R1, R0, R0      ; aa==0
L1:     DSUBUIR3, R2, #2
BNEZ    R3, L2          ; 分支b2 (bb!=2)
DADD    R2, R0, R0      ; bb==0
L2:     DSUBU  R3, R1, R2 ; R3=aa-bb
BEQZ    R3, L3          ; 分支b3 (aa==bb)

```

4. 高级分支预测技术

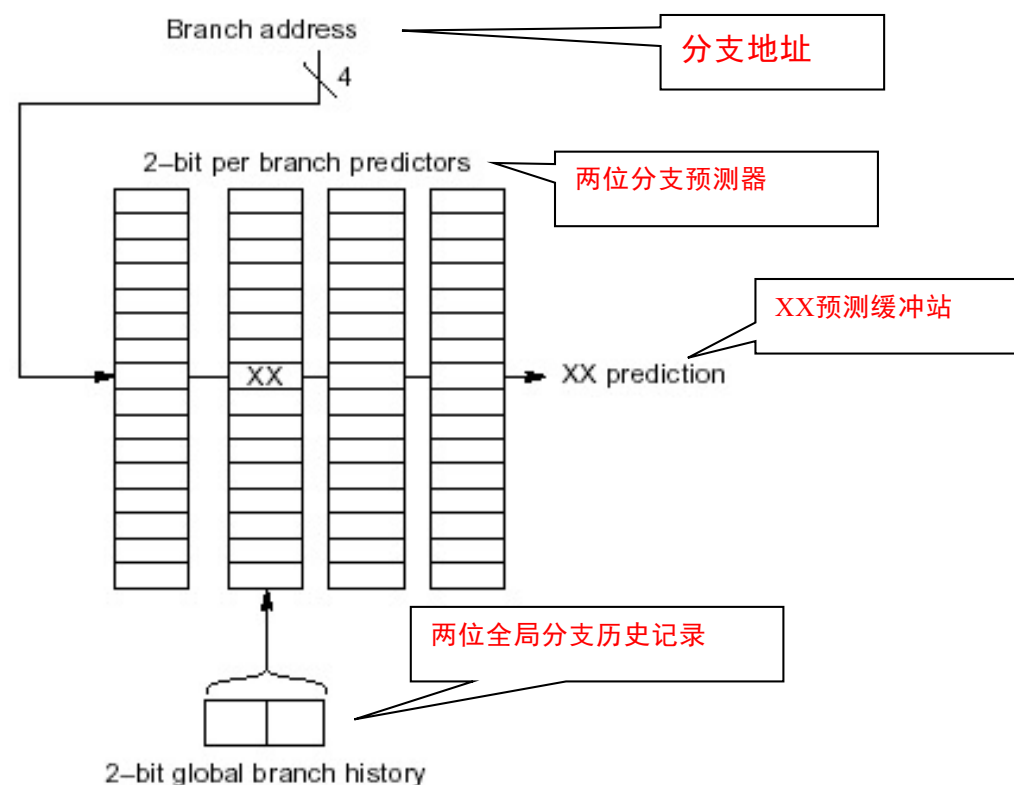
① 相关分支预测器

- 使用其它分支行为来进行预测的分支预测器
- (m, n) 预测器表示使用前 m 个分支行为去从 2^m 个分支预测中进行选择，每一个预测是对应于单个分支的 n 位预测器
- 最近 m 个分支的全局历史记录可以记录在一个 m 位移位寄存器中，每一位记录着该分支是被执行还是未被执行（增加的硬件并不复杂）

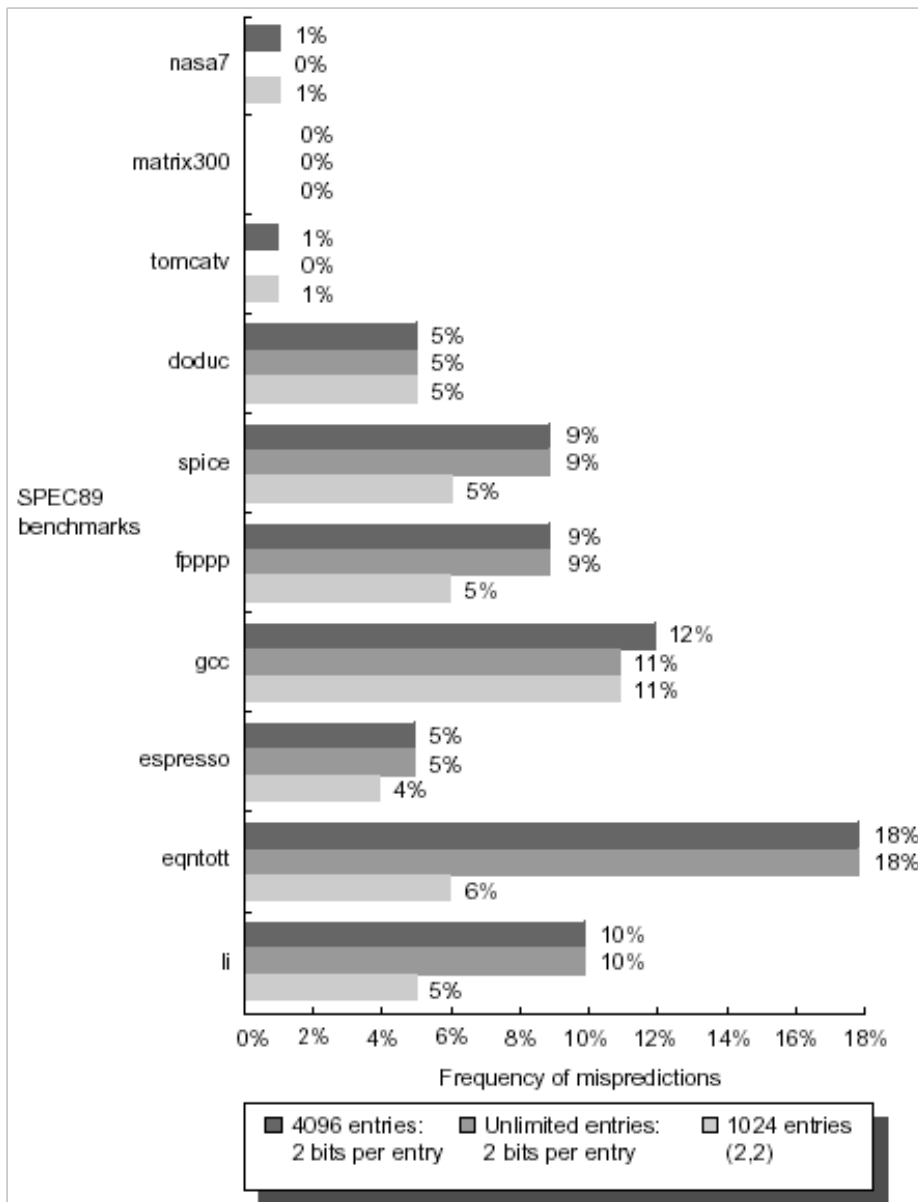
4. 高级分支预测技术

① 相关分支预测器

对分支预测缓冲站的访问可由分支地址的低位拼接上m位全局历史记录而得到



在一个 (m, n) 预测器中，所需要的位数是：
 $2^m \times n \times$ 分支地址所能选择的预测数目



- 一个4K个入口的两位BHT共占用8K个数据位。
- 而一个 (2,2) 相关预测期如果占用8K个数据位，则，

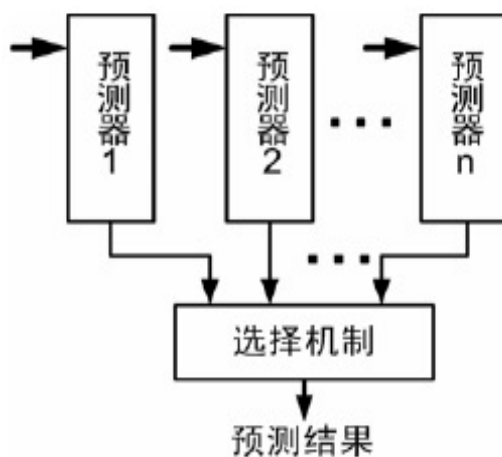
$$2^2 \times 2 \times \text{分支选择的入口数目} = 8K$$

$$\text{分支选择的入口数目} = 1K$$

4. 高级分支预测技术

② Tournament预测器（竞争预测器）

- 将全局预测和局部预测结合，通过一个选择器从二者中选择
- 可以为特定的转移选择正确的预测器

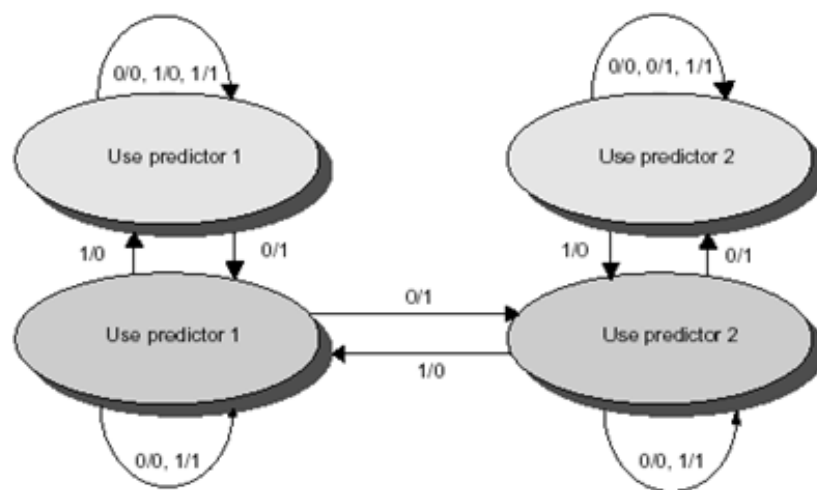


Alpha21264的Tournament预测器：

- **全局预测器：**使用最后12个分支跳转情况进行索引，即全局预测器也有 $2^{12}=4K$ 个入口，每个入口都是一个标准的两位预测器。
- **局部预测器：**设计为两层，上面一层是一个局部历史记录，使用指令地址的低10位进行地址索引，即有 $2^{10}=1K$ 个入口，每个入口10位（10位的BHT，对循环支持较好），分别对应这个入口最近的10个分支，即该分支指令最近10次的跳转情况，这种10位历史记录允许对10个分支进行记录和预测，从局部历史记录选择出的入口对一个1K的入口表进行索引，这些入口由3位计数器构成，以提供本地预测。

Alpha21264的Tournament预测器：

- **选择器：** 使用分支局部地址的低12位分支局部地址索引，即有 $2^{12}=4K$ 个选择器，每个索引得到一个两位计数器，用来选择使用局部预测器还是使用全局预测器的预测结果。在设计时默认使用局部预测器，当两个预测器都正确或都不正确时，不改变计数器；当全局预测器正确而局部预测器预测错误时，计数器加1，否则减1。



5.4 动态分支预测技术

➤ 分支预测中的操作有两个步骤：

□ 分支预测；

- 当分支指令到达译码段（ID）时，根据从BHT读出的信息进行分支预测。
- 若预测正确，就继续处理后续的指令，流水线没有断流。否则，就要作废已经预取和分析的指令，恢复现场，并从另一条分支路径重新取指令。

□ 状态修改。

5.4 动态分支预测技术

5. BHT方法只在以下情况下才有用：

- 判定分支是否成功所需的时间大于确定分支目标地址所需的时间。

前述5段经典流水线：由于判定分支是否成功和计算分支目标地址都是在ID段完成，所以BHT方法不会给该流水线带来好处。

5.4 动态分支预测技术

5.4.2 采用分支目标缓冲器BTB

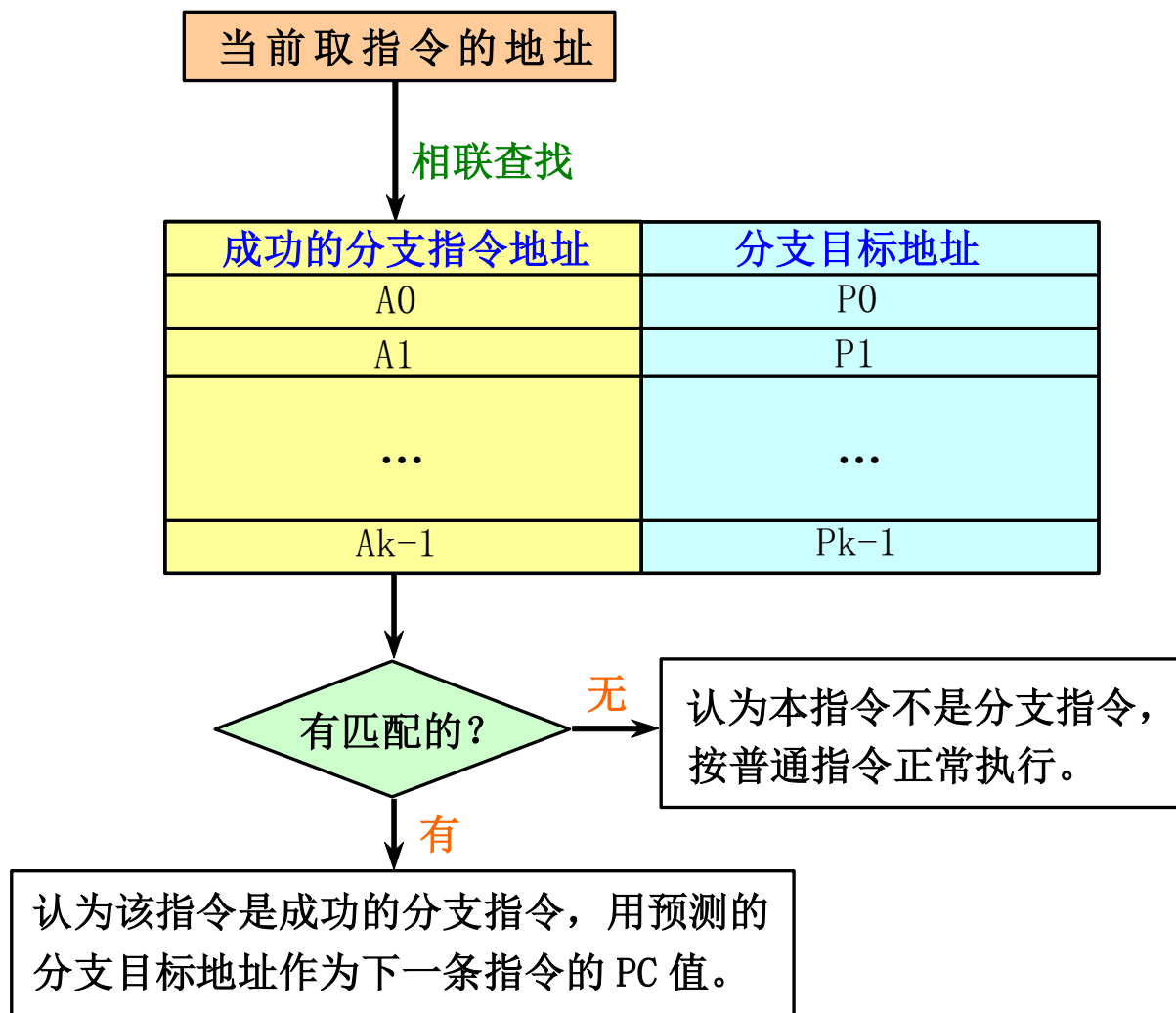
目标：将分支的开销降为 0

方法：分支目标缓冲

- 将分支成功的分支指令的地址和它的分支目标地址都放到一个缓冲区中保存起来，缓冲区以分支指令的地址作为标识。
- 这个缓冲区就是分支目标缓冲器（Branch-Target Buffer，简记为BTB，或者分支目标Cache（Branch-Target Cache）。

5.4 动态分支预测技术

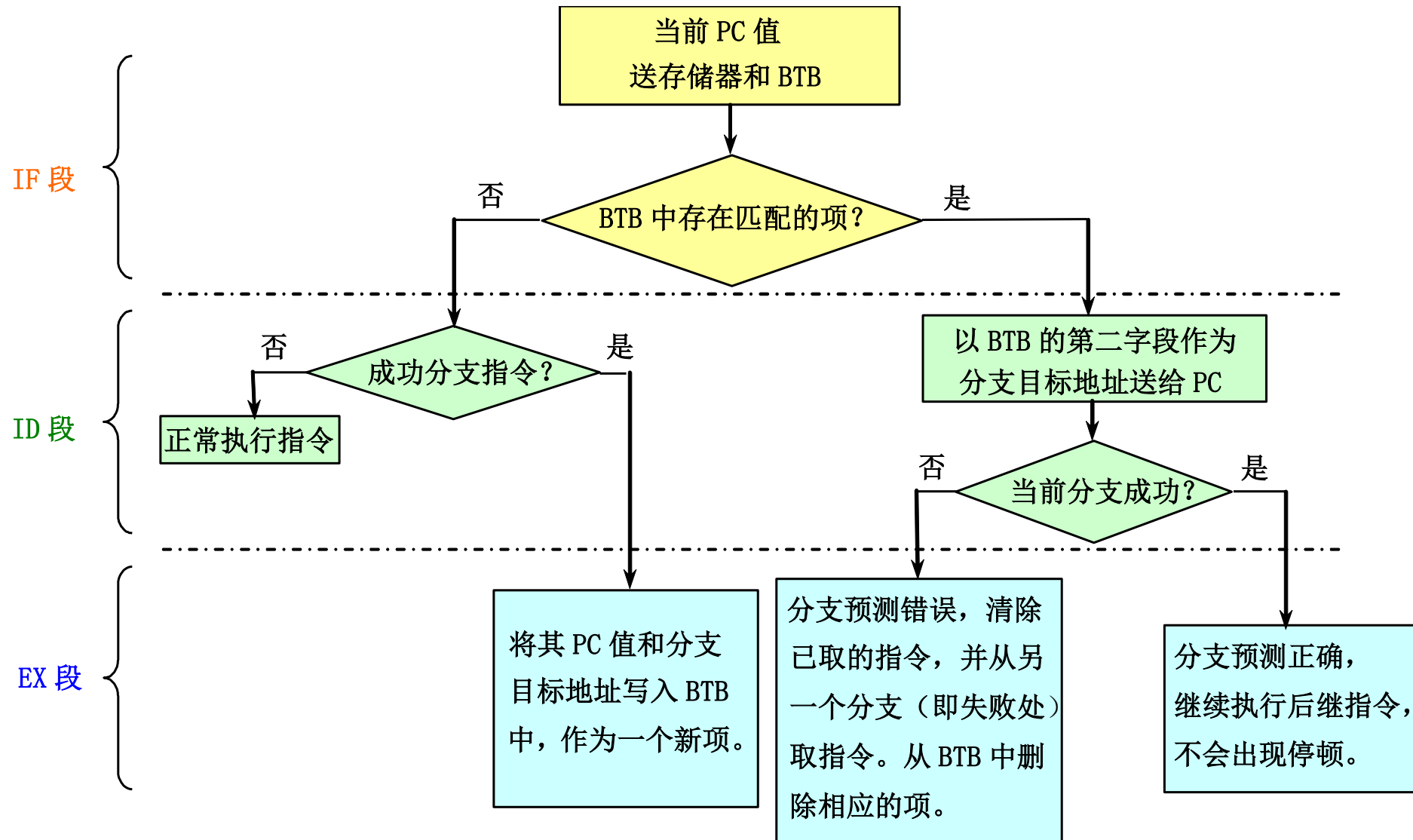
1. BTB的结构



5.4 动态分支预测技术

- BTB是用专门的硬件实现的一张表格。
- 表格中的每一项至少有两个字段：
 - 执行过的成功分支指令的地址；
（作为该表的匹配标识）
 - 预测的分支目标地址。

- 采用BTB后，在流水线各个阶段所进行的相关操作：



5.4 动态分支预测技术

2. 采用BTB后，各种可能情况下的延迟：

指令在BTB中？	预测	实际情况	延迟周期
是	成功	成功	0
是	成功	不成功	2
不是		成功	2
不是		不成功	0

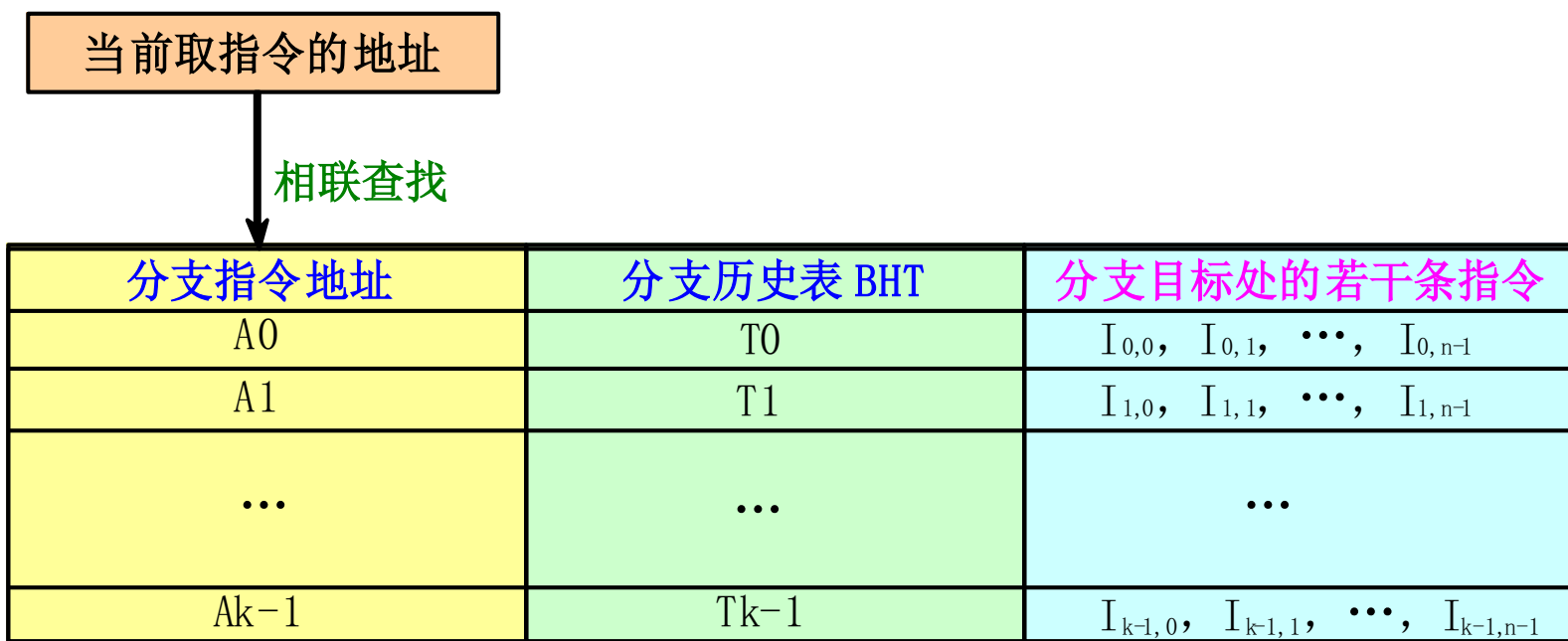
2. BTB的另一种形式

- 在分支目标缓冲器中增设一个至少是两位的“分支历史表”字段



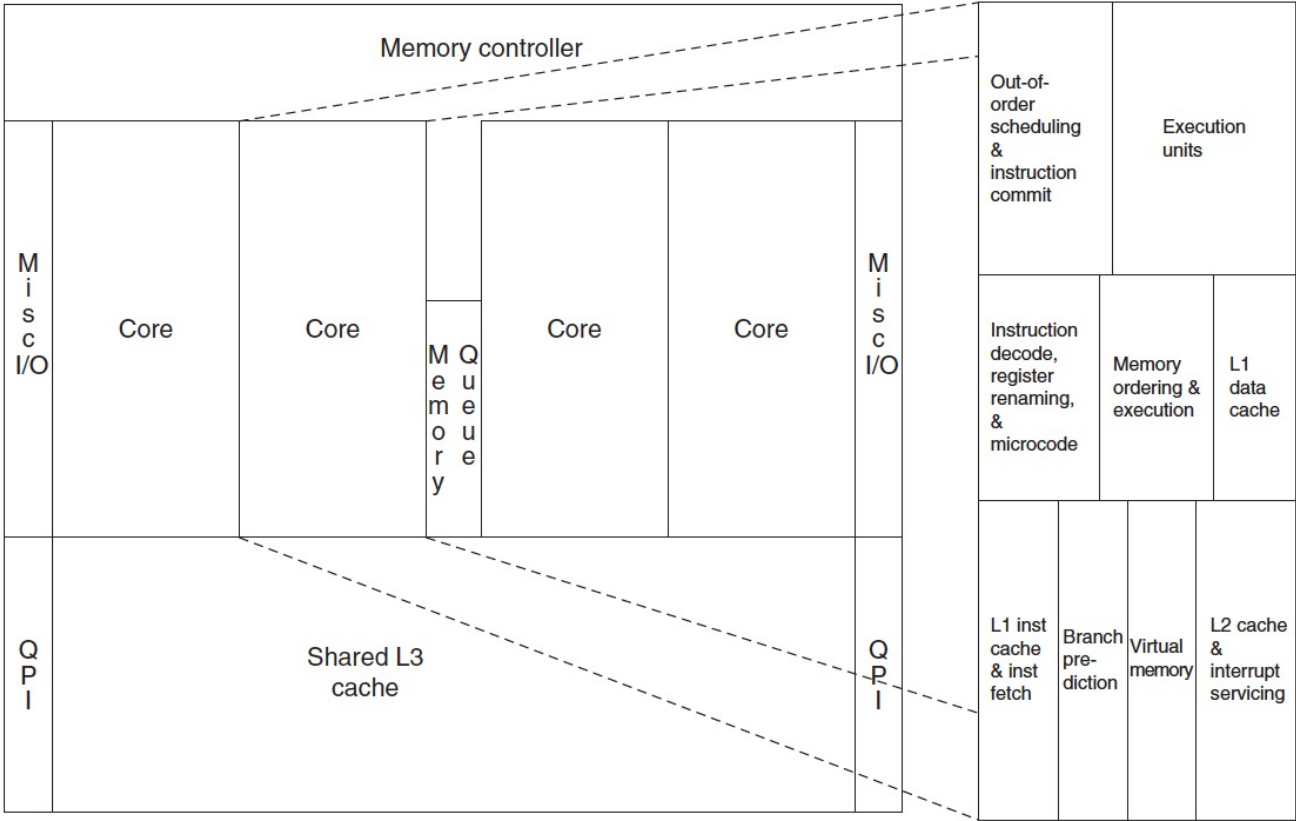
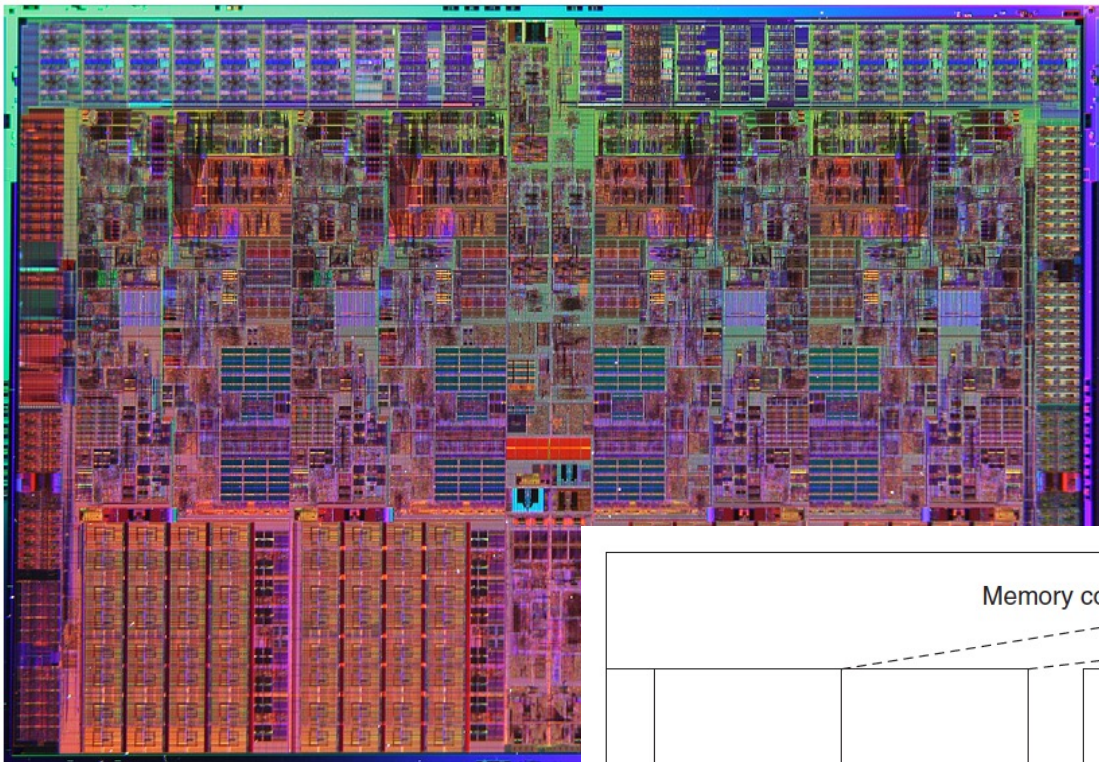
5.4 动态分支预测技术

3. 更进一步，在表中对于每条分支指令都存放若干条分支目标处的指令，就形成了分支目标指令缓冲器。



- 允许分支目标缓冲站访问时间超过相继指令间的取指令时间
 - 从而允许使用更大的分支目标缓冲站
- 分支折叠

i7的晶片照片及结构布置图



5.6 基于硬件的前瞻执行（自学）

5.6.1 基于硬件的前瞻执行

前瞻执行（speculation）的基本思想：

对分支指令的结果进行猜测，并假设这个猜测总是对的，然后按这个猜测结果继续取、流出和执行后续的指令。只是执行指令的结果不是写回到寄存器或存储器，而是写入一个称为再定序缓冲器 ROB（ReOrder Buffer）中。等到相应的指令得到“确认”（commit）（即确实是应该执行的）之后，才将结果写入寄存器或存储器。

1. 基于硬件的前瞻执行结合了3种思想：

- 动态分支预测。用来选择后续执行的指令。
- 在控制相关的结果尚未出来之前，前瞻地执行后续指令。
- 用动态调度对基本块的各种组合进行跨基本块的调度。

2. 对Tomasulo算法加以扩充，就可以支持前瞻执行。

把Tomasulo算法的写结果和指令完成加以区分，分成两个不同的段：

写结果，指令确认

已经在一系列处理机上实现了（**PowerPC 620**，**MIPS R10000**，**Intel P6**和**AMD K5**）

➤ 写结果段

- 把前瞻执行的结果写到ROB中；
- 通过CDB在指令之间传送结果，供需要用到这些结果的指令使用。

➤ 指令确认段

在分支指令的结果出来后，对相应指令的前瞻执行给予确认。

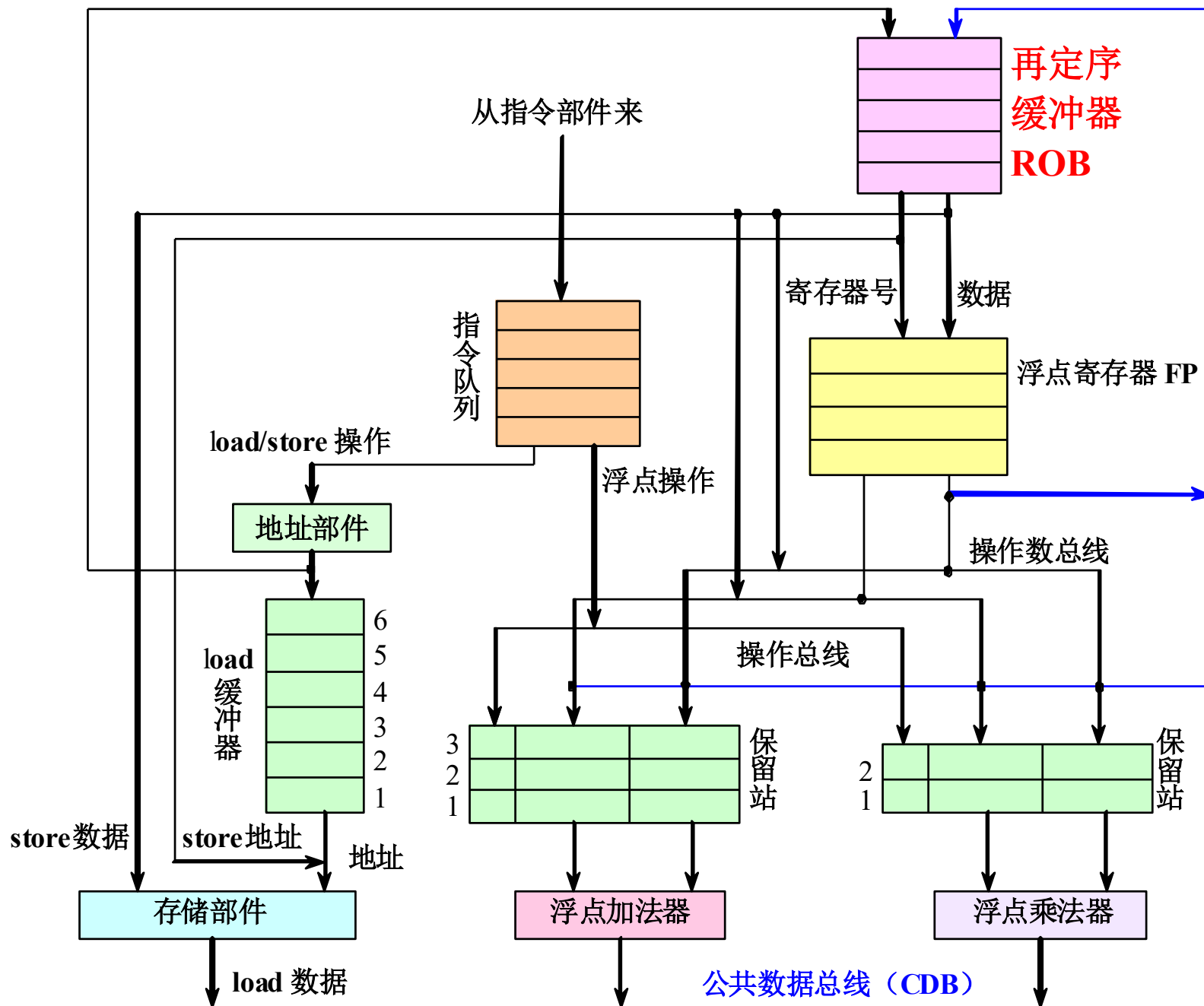
- 如果前面所做的猜测是对的，把在ROB中的结果写到寄存器或存储器。
- 如果发现前面对分支结果的猜测是错误的，那就不予以确认，并从那条分支指令的另一条路径开始重新执行。

3. 实现前瞻的关键思想：

允许指令乱序执行，但必须顺序确认。

在指令被确认之前，不允许它进行不可恢复的操作。

支持前瞻执行的浮点部件的结构



➤ ROB中的每一项由以下4个字段组成：

- 指令类型

指出该指令是分支指令、store指令或寄存器操作指令。

- 目标地址

给出指令执行结果应写入的目标寄存器号（如果是load和ALU指令）或存储器单元的地址（如果是store指令）。

- 数据值字段

用来保存指令前瞻执行的结果，直到指令得到确认。

- 就绪字段

指出指令是否已经完成执行并且数据已就绪。

Tomasulo算法中保留站的换名功能是由ROB来完成的。

4. 采用前瞻执行机制后，指令的执行步骤：

（在Tomasulo算法的基础上改造的）

➤ 流出

- 从浮点指令队列的头部取一条指令。
- 如果有空闲的保留站（设为 r ）且有空闲的ROB项（设为 b ），就流出该指令，并把相应的信息放入保留站 r 和ROB项 b 。
 - 如果寄存器或者ROB中已经含有源操作数，将其发入保留站 r ，否则将产生该源操作数的指令所分配的ROB编号送入 r
 - 将为该指令分配的保留站编号送入 r
- 如果保留站或ROB全满，便停止流出指令，直到它们都有空闲的项。

➤ 执行

- 如果有操作数尚未就绪，就等待，并不断地监测CDB。
(检测RAW冲突)
- 当两个操作数都已在保留站中就绪后，就可以执行该指令的操作。

➤ 写结果

- 当结果产生后，将该结果连同本指令在流出段所分配到的ROB项的编号放到CDB上，经CDB写到ROB以及所有等待该结果的保留站。
- 释放产生该结果的保留站。
- store指令在本阶段完成操作为：
 - 如果要写入存储器的数据已经就绪，就把该数据写入分配给该store指令的ROB项。
 - 否则，就监测CDB，直到那个数据在CDB上播送出来，才将之写入分配给该store指令的ROB项。

➤ 确认

对分支指令、store指令以及其它指令的处理不同：

- 其它指令（除分支指令和store指令）

当该指令到达ROB队列的头部而且其结果已经就绪时，就把该结果写入该指令的目的寄存器，并从ROB中删除该指令。

- store指令

处理与上面的类似，只是它把结果写入存储器。

□ 分支指令

- 当预测错误的分支指令到达ROB队列的头部时，清空ROB，并从分支指令的另一个分支重新开始执行。

（错误的前瞻执行）

- 当预测正确的分支指令到达ROB队列的头部时，该指令执行完毕。

例5.4 假设浮点功能部件的延迟时间为：加法2个时钟周期，乘法10个时钟周期，除法40个时钟周期。对于下面的代码段，给出当指令MUL.D即将确认时的状态表内容。

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

前瞻执行中**MUL.D**确认前，保留站和**ROB**的状态

名称	保留站							
	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	no							
Load2	no							
Add1	no							
Add2	no							
Add3	no							
Mult1	no	MUL.D	Mem[45+ Regs[R2]]	Regs[F4]			#3	
Mult2	yes	DIV.D		Mem[34+Regs[R2]]	#3		#5	

项号	ROB					
	Busy	指令		状态	目的	Value
1	no	L.D	F6, 34 (R2)	确认	F6	Mem[34+Regs[R2]]
2	no	L.D	F2, 45 (R3)	确认	F2	Mem[45+Regs[R3]]
3	yes	MUL.D	F0, F2, F4	写结果	F0	#2×Regs[F4]
4	yes	SUB.D	F8, F6, F2	写结果	F8	#1－#2
5	yes	DIV.D	F10, F0, F6	执行	F10	
6	yes	ADD.D	F6,F8,F2	写结果	F6	#4＋#2

字段	浮点寄存器状态							
	F0	F2	F4	F6	F8	F10	...	F30
ROB项编号	3			6	4	5		
Busy	yes	no	no	yes	yes	yes	...	no

□ 前瞻执行

- 通过ROB实现了指令的**顺序完成**。
 - 能够**实现精确异常**。
- 很容易地推广到整数寄存器和整数功能单元上。
- 可应用于多发射处理器
- **主要缺点：**所需的硬件太复杂。

例：考虑如下循环程序在一个双发射处理器上执行过程，第一次不支持前瞻执行，第二次采用了前瞻执行。

```
Loop: LD          R2, 0(R1)    ; R2为数组元素
      DADDIU      R2, R2, #1    ; R2加1
      SD          R2, 0(R1)    ; 保存结果
      DADDIU      R1, R1, #4    ; 指针加1
      BNE         R2, R3, Loop
```

- 假设有单独整数功能部件用来进行有效地址计算和分支条件计算
- 假设任意两条指令能在一个周期内提交

不使用前瞻执行

循环 体号	指令	发射 时刻	执行 时刻	内存存 取时刻	写CDB 时刻	注释
1	LD R2, 0(R1)	1	2	3	4	第一个发射
1	DADDIU R2, R2, #1	1	5		6	等待LD
1	SD R2, 0(R1)	2	3	7		等待DADDIU
1	DADDIU R1, R1, #4	2	3		4	直接执行
1	BNE R2, R3, LOOP	3	7			等待DADDIU
2	LD R2, 0(R1)	4	8	9	10	等待BNE
2	DADDIU R2, R2, #1	4	11		12	等待LD
2	SD R2, 0(R1)	5	9	13		等待DADDIU
2	DADDIU R1, R1, #4	5	8		9	等待BNE
2	BNE R2, R3, LOOP	6	13			等待DADDIU
3	LD R2, 0(R1)	7	14	15	16	等待 BNE
3	DADDIU R2, R2, #1	7	17		18	等待LD
3	SD R2, 0(R1)	8	15	19		等待DADDIU
3	DADDIU R1, R1, #4	8	14		15	等待BNE
3	BNE R2, R3, LOOP	9	19			等待DADDIU

循环 体号	指令		发射 时刻	执行 时刻	内存 存取 时刻	写CDB 时刻	确认 时刻	注释
1	LD	R2, 0(R1)	1	2	3	4	5	第一次发射
1	DADDIU	R2, R2, #1	1	5		6	7	等待LD
1	SD	R2, 0(R1)	2	3			7	等待DADDIU
1	DADDIU	R1, R1, #4	2	3		4	8	顺序提交
1	BNE	R2, R3, LOOP	3	7			8	等待DADDIU
2	LD	R2, 0(R1)	4	5	6	7	9	无延迟执行
2	DADDIU	R2, R2, #1	4	8		9	10	等待LD
2	SD	R2, 0(R1)	5	6			10	等待DADDIU
2	DADDIU	R1, R1, #4	5	6		7	11	顺序提交
2	BNE	R2, R3, LOOP	6	10			11	等待DADDIU
3	LD	R2, 0(R1)	7	8	9	10	12	尽可能提前
3	DADDIU	R2, R2, #1	7	11		12	13	等待LD
3	SD							DADDIU
3	DADDIU							前
3	BNE	R2, R3, LOOP	9	13			14	等待DADDIU

➤ 分支预测要准确，否则性能将会非常糟糕
ARM Cortex-A8 错误预测的代价为13个周期

第五章作业

➤ 2, 3, 4, 5, 8, 9