第5章 指令级并行及其开发——硬件方法

张晨曦 刘依

www.GotoSchool.net

xzhang2000@sohu.com

- 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 （硬件的方法）

3. 基本程序块
   - 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。
4. 循环级并行：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

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：

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

   - 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）

- 例如，考虑下述语句：

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

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

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. 相关与流水线冲突

- 相关有三种类型：

  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。

  流水线冲突有三种类型：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。

2. 可以从两个方面来解决相关问题：
   - 保持相关，但避免发生冲突。

     指令调度
   - 通过代码变换，消除相关。
3. 程序顺序：由原来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。

4. 控制相关并不是一个必须严格保持的关键属性。
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。
   - 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。

➢ 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
  - 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。

- 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

5.3.1 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:

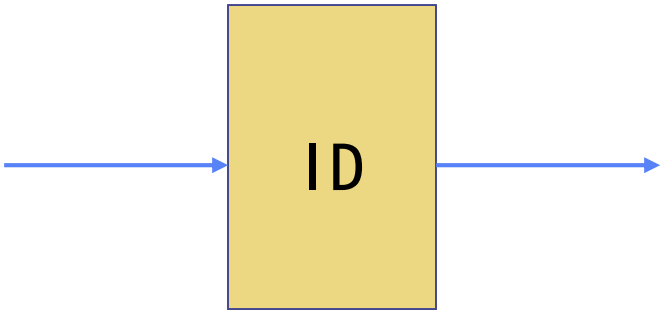
- 指令是按序流出和按序执行的
- 考虑下面一段代码:

```
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。

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。

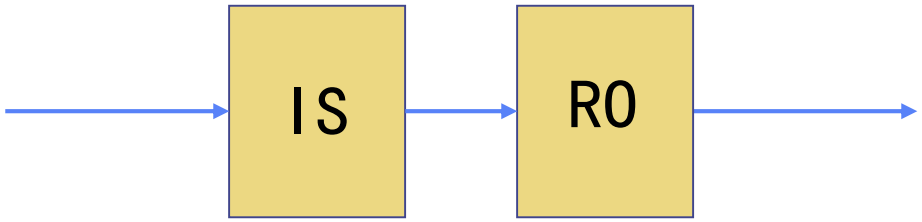
- 为了使上述指令序列中的SUB.D指令能继续执行下去，必须把指令流出的工作拆分为两步：
  - 检测结构冲突
  - 等待数据冲突消失

  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。

3. 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)



检测结构冲突　　检测数据冲突

4. 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
ADD.D   F10, F4, F6
SUB.D   F6, F8, F14
```

存在输出相关：DIV.D 与 ADD.D

存在反相关：ADD.D 与 SUB.D

可以通过使用寄存器重命名来消除。

5. 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。
6. 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。

- 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。
- 不精确异常：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：

    因为当发生异常（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。

➢ 精确异常：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。

记分牌算法和Tomasulo算法是两种比较典型的动态调度算法。

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。

- 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。

➢ 采用了记分牌的MIPS处理器的基本结构

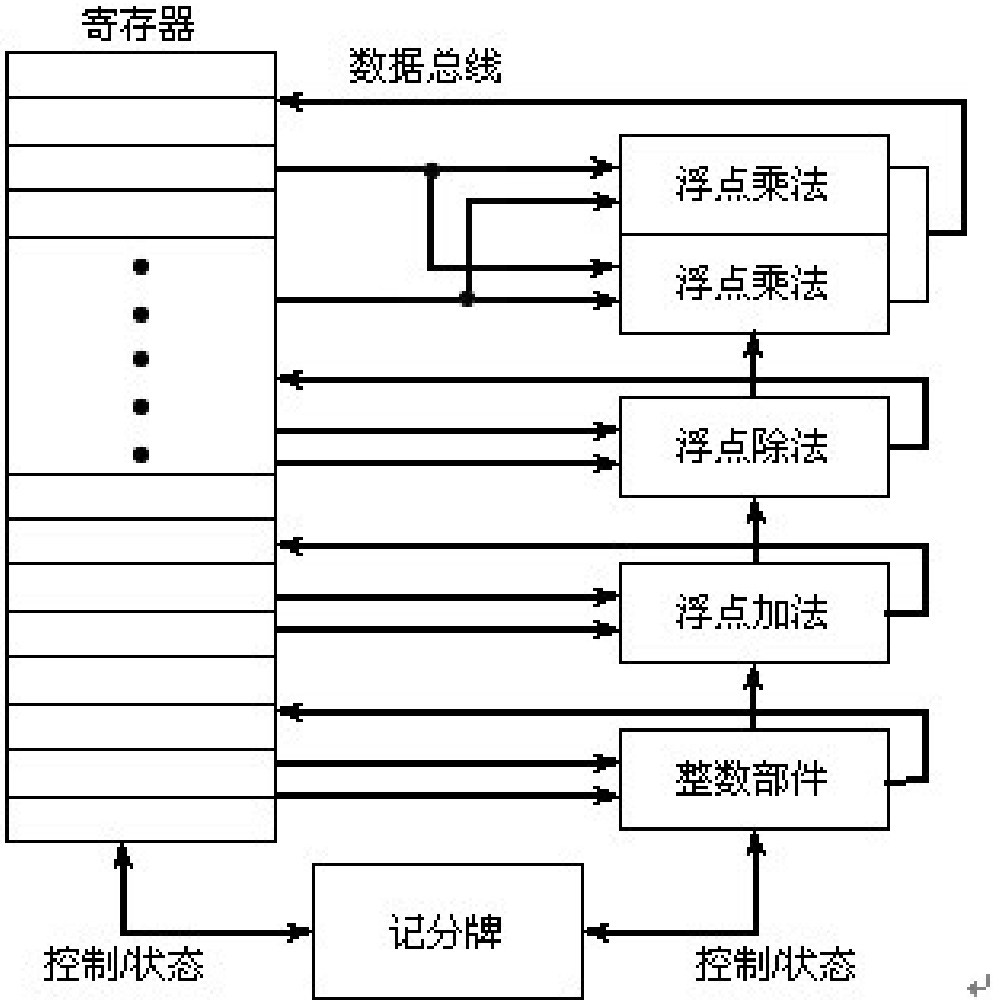
- 每条指令都要经过记分牌。
- 记分牌负责相关检测并控制指令的流出和执行。

➢ 每条指令的执行过程分为4段 （主要考虑浮点操作 ）

- 流出

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


寄存器
数据总线
浮点乘法
浮点乘法
浮点除法
浮点加法
整数部件
记分牌
控制/状态
控制/状态

- 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。

- 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。

- 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。
  - 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）。

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
```

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

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

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。

（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。功能部件状态表中各字段的含义见前面。

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。

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”，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]←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。

（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]←no；　　// 已经读走了就绪的第一个源操作数。

Rk[FU]←no；　　// 已经读走了就绪的第二个源操作数。

Qj[FU]←0；　　// 不再等待其他FU的计算结果。

Qk[FU]←0；

（3）执行

结束条件：

功能部件操作结束。

（4）写结果

进入条件：

∀f((Fj[f]≠Fi[FU] or Rj[f]=no)

& (Fk[f]≠Fi[FU] or Rk[f]=no))；// 不存在WAR冲突。

计分牌内容修改：

∀f(if Qj[f]=FU then Rj[f]←yes)；　// 如果有指令在等待该结果（作为第一源操作数），则将其Rj置为“yes”，表示数据可用。

∀f(if Qk[f]=FU then Rk[f]←yes)；　// 如果有指令在等待该结果（作为第二源操作数），则将其Rk置为“yes”，表示数据可用。

Result(Fi[FU])←0；　// 释放目的寄存器Fi[FU]。

Busy[FU]=no；// 释放功能部件FU。

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。
- 反相关和输出相关。
  - 它们引起计分牌中WAR和WAW冲突 。

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。

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

反相关（F8）导致WAR冲突

输出相关（F6）导致WAW冲突

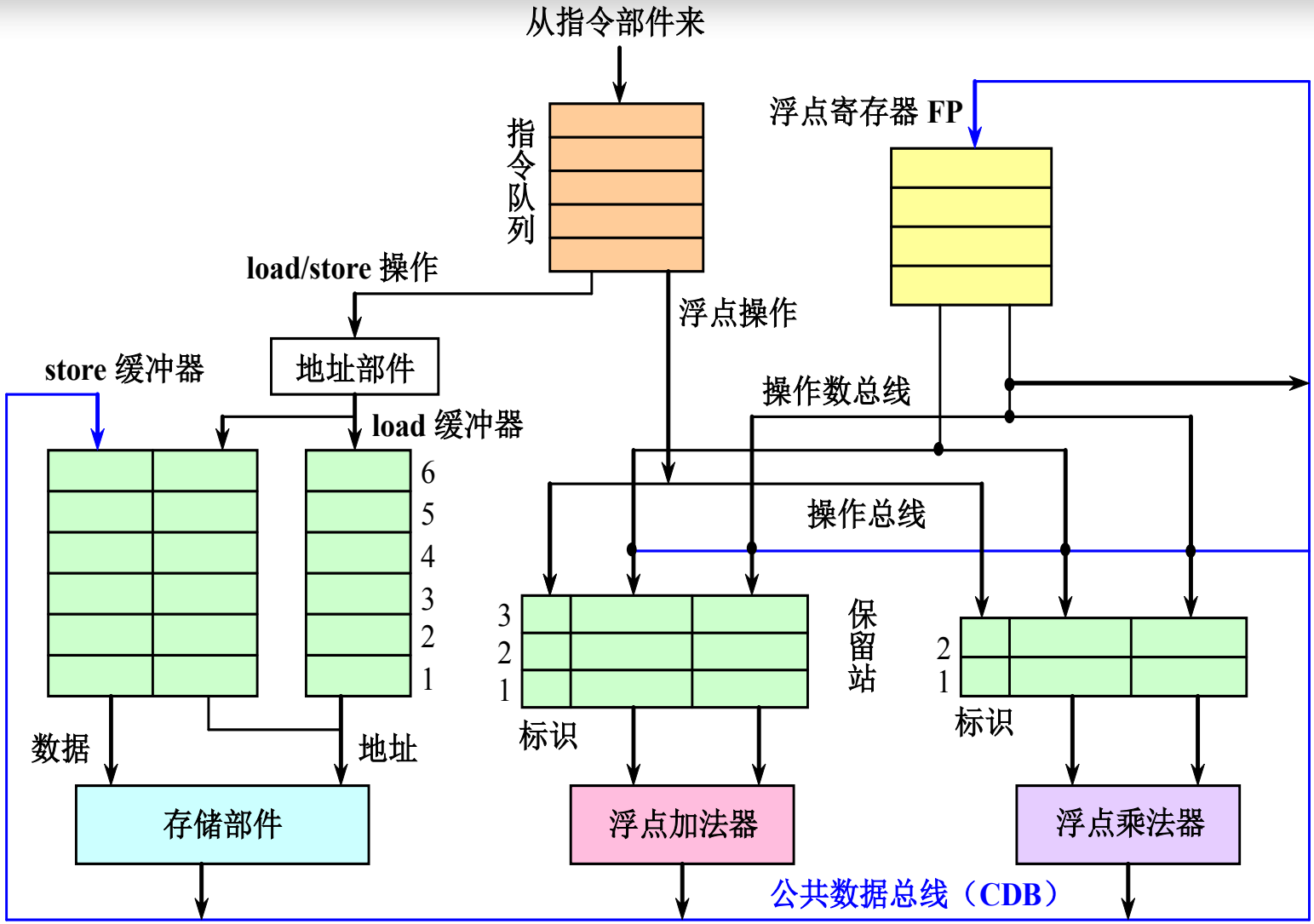
- 消除名相关
  - 引入两个临时寄存器S和T
  - 把这段代码改写为：

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

两个F6都换名为S

两个F8都换名为T

4. 基于Tomasulo算法的MIPS处理器浮点部件的基本结构


从指令部件来
指令队列
浮点寄存器 FP
load/store 操作
浮点操作
地址部件
store 缓冲器
load 缓冲器
操作数总线
操作总线
6
5
4
3
2
1
3
2
1
保留站
2
1
数据
地址
标识
标识
存储部件
浮点加法器
浮点乘法器
公共数据总线（CDB）

➢ 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和解决冲突的信息。

- 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。

- 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。

- 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。

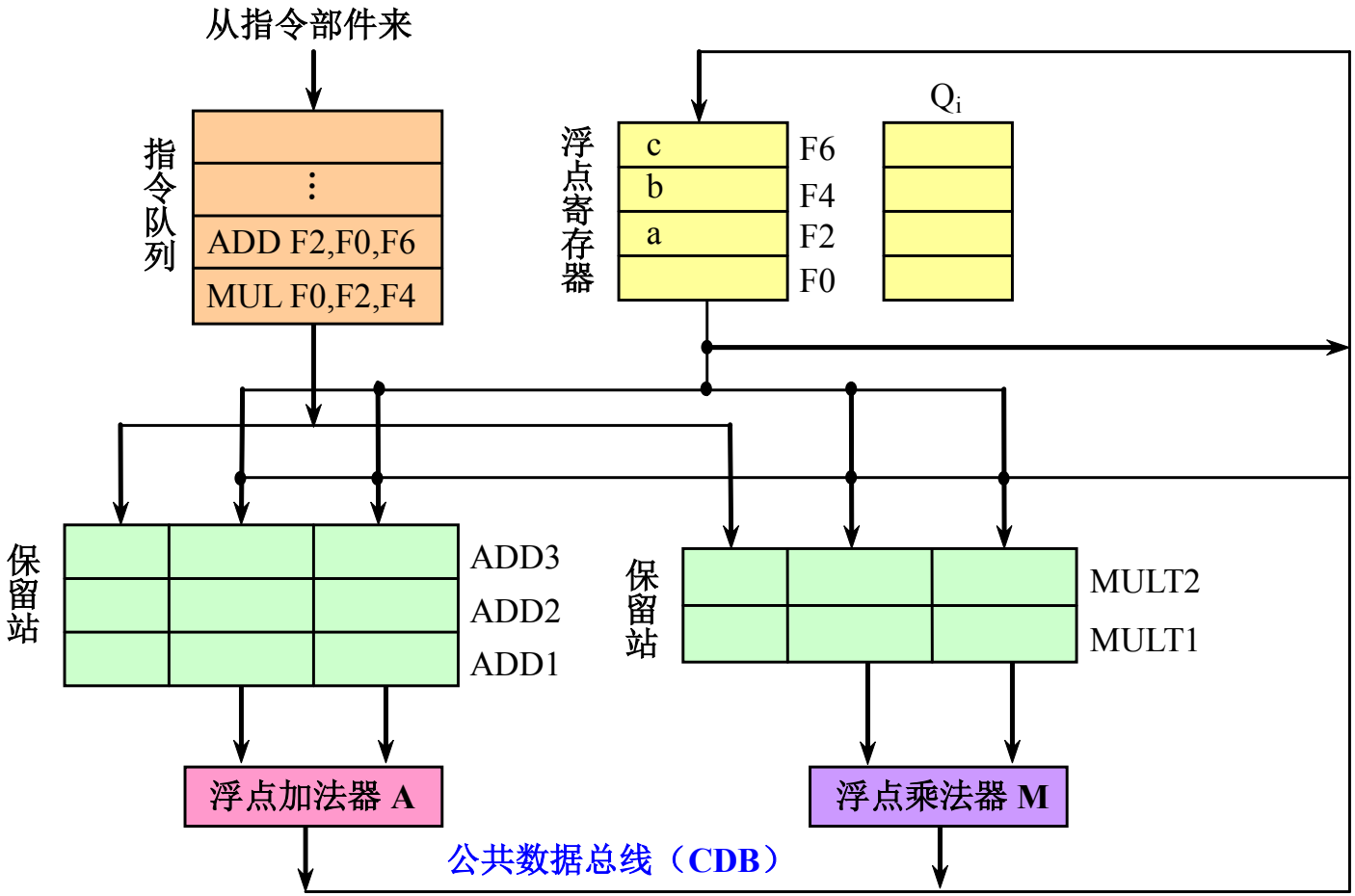
➢ 指令队列

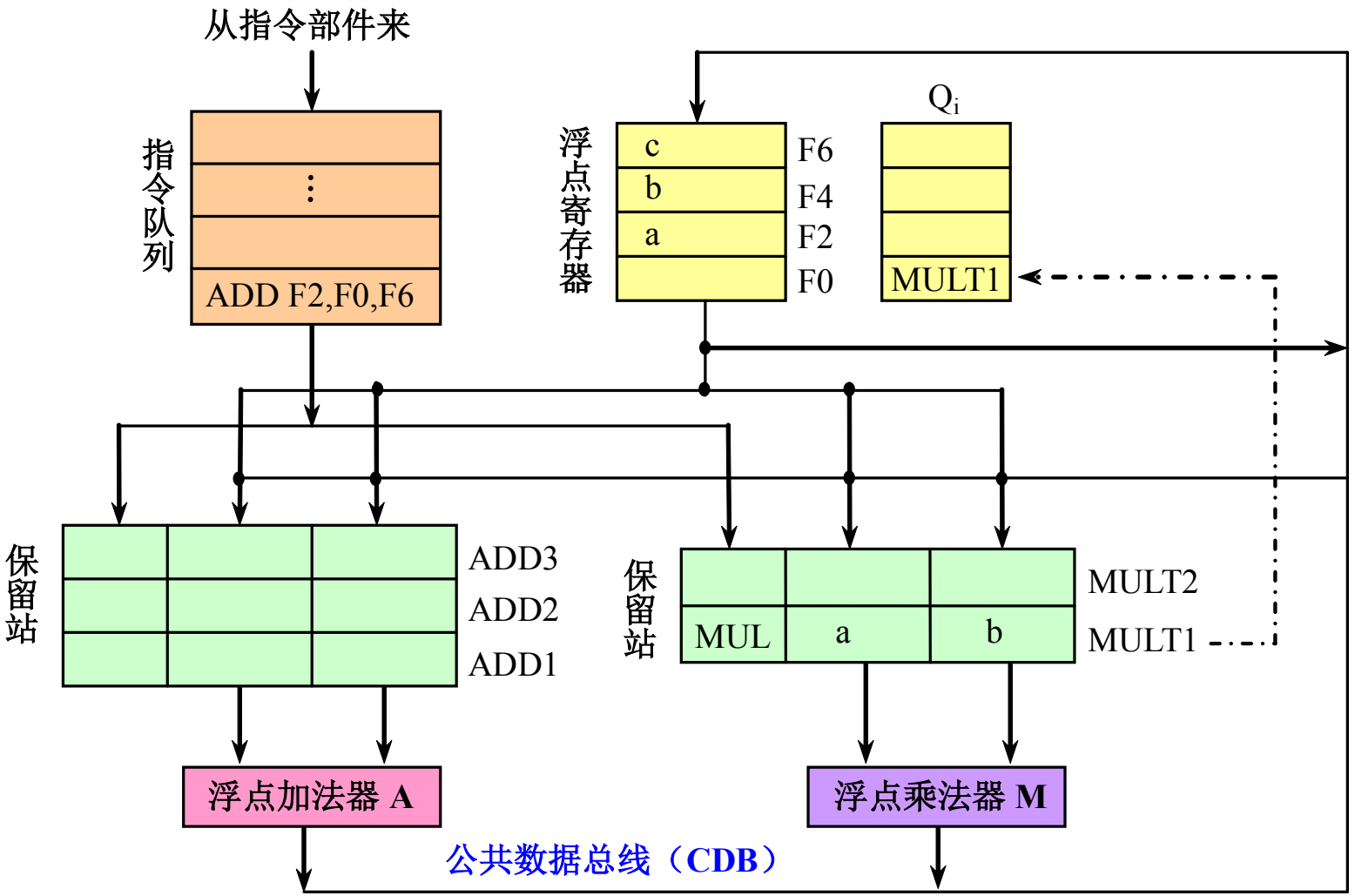
- 指令部件送来的指令放入指令队列
- 指令队列中的指令按先进先出的顺序流出

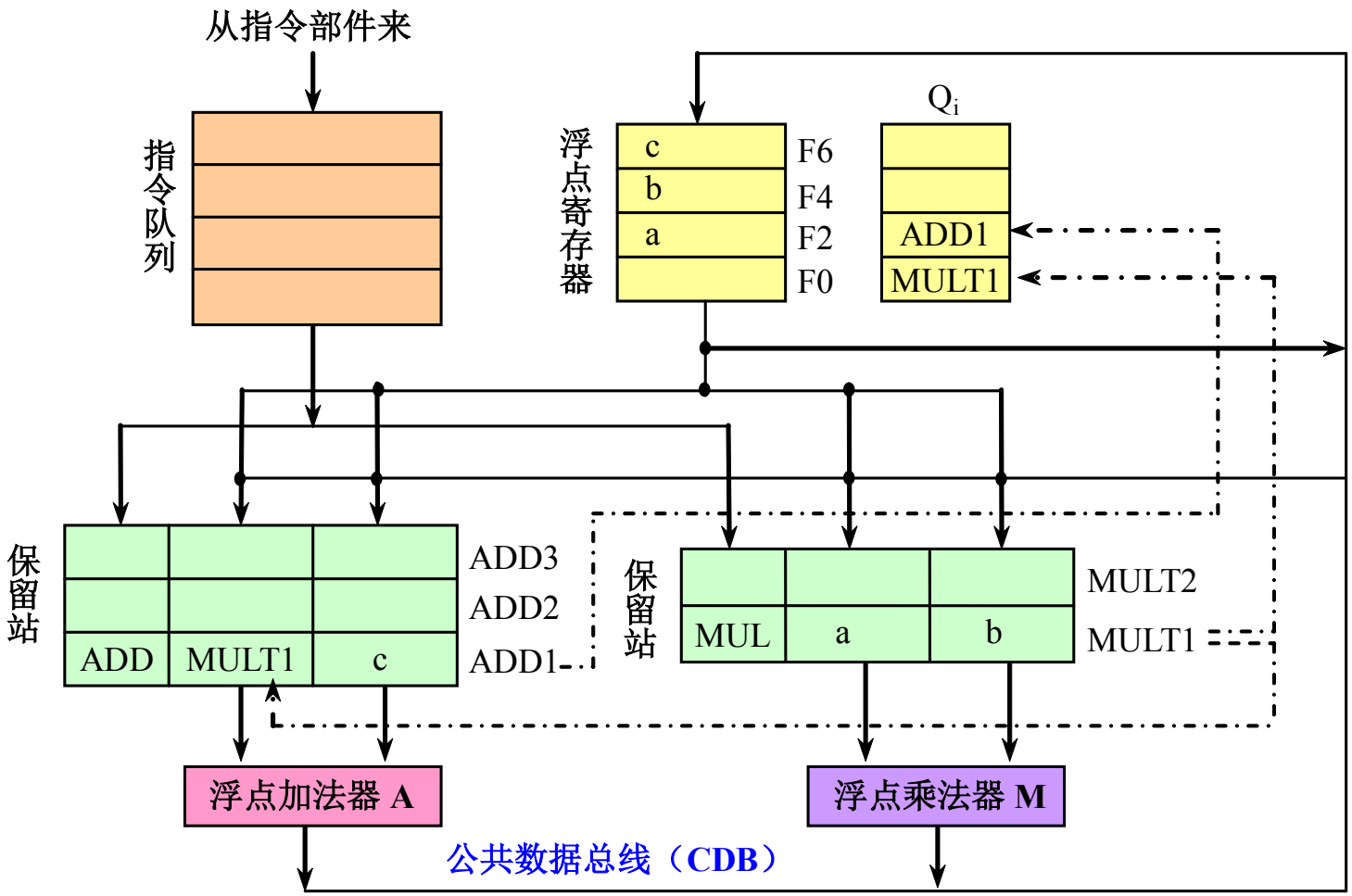
- 运算部件
  - 浮点加法器完成加法和减法操作
  - 浮点乘法器完成乘法和除法操作

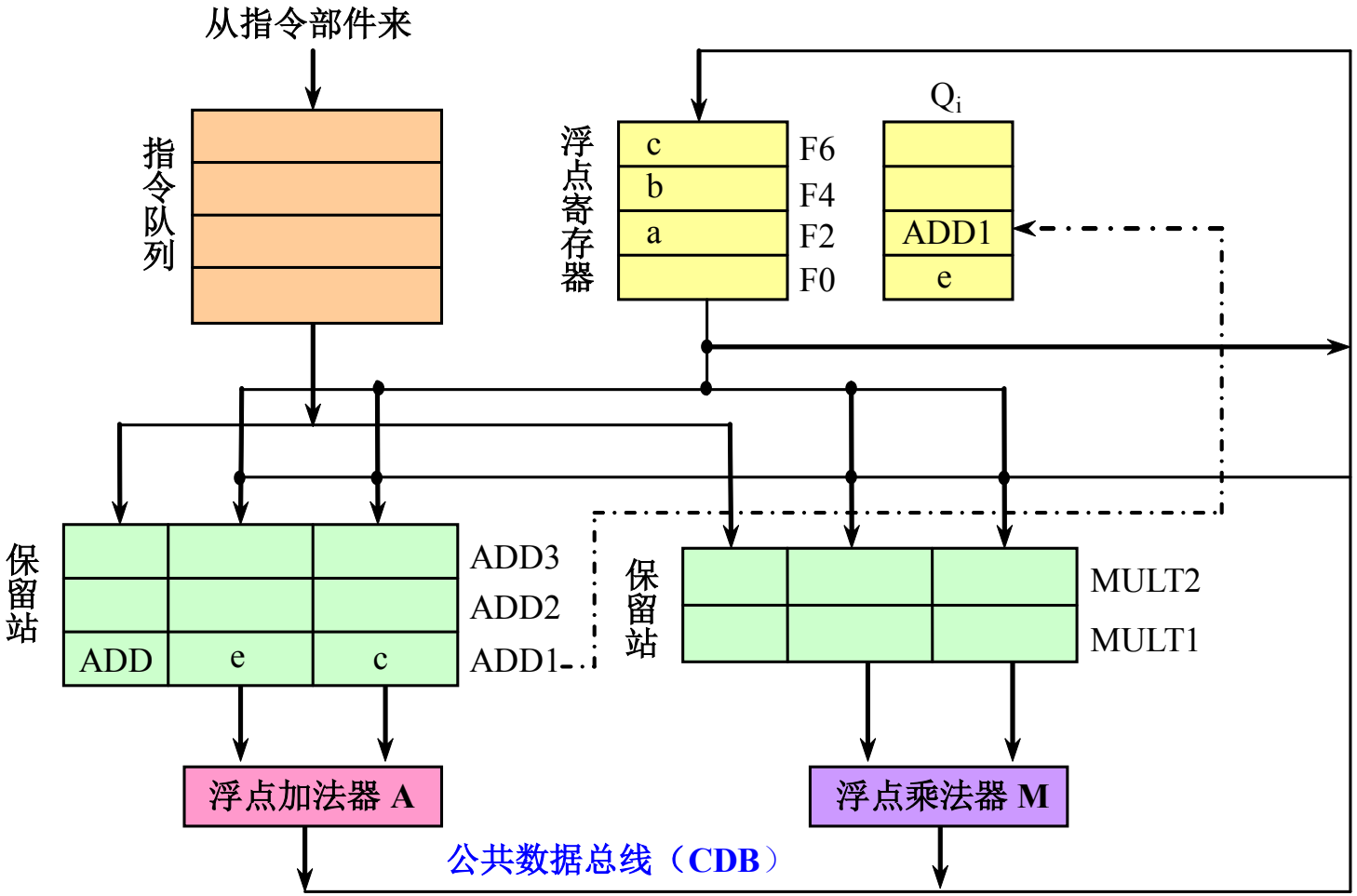
5. 在Tomasulo算法中，寄存器换名是通过保留站和流出逻辑来共同完成的。
   - 当指令流出时，如果其操作数还没有计算出来，则将该指令中相应的寄存器号换名为将产生这个操作数的保留站的标识。
   - 指令流出到保留站后，其操作数寄存器号或者换成了数据本身（如果该数据已经就绪），或者换成了保留站的标识，不再与寄存器有关系。

6. 通过一个简单的例子来说明Tomasulo算法的基本思想




从指令部件来
指令队列
ADD F2,F0,F6
浮点寄存器
c
b
a
F6
F4
F2
F0
Qi
MULT1
保留站
ADD3
ADD2
ADD1
保留站
MUL
a
b
MULT2
MULT1
浮点加法器 A
浮点乘法器 M
公共数据总线（CDB）


从指令部件来
指令队列
浮点寄存器
c
b
a
F6
F4
F2
F0
Qi
ADD1
MULT1
保留站
ADD3
ADD2
ADD1
ADD
MULT1
c
保留站
MULT2
MULT1
MUL
a
b
浮点加法器 A
浮点乘法器 M
公共数据总线（CDB）


从指令部件来
指令队列
浮点寄存器
c
b
a
F6
F4
F2
F0
Qi
ADD1
e
保留站
ADD3
ADD2
ADD1
ADD
e
c
保留站
MULT2
MULT1
浮点加法器 A
浮点乘法器 M
公共数据总线（CDB）

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：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）

- 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。

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。

- 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。

5.3.3.2 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
```

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	…	

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。

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。

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	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	…	

5.3.3.3 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；

- 与rs对应的保留站字段：Vj，Qj
- 与rt对应的保留站字段：Vk，Qk
- Qi、Qj、Qk的内容或者为0，或者是一个大于0的整数。
  - Qi为0表示相应寄存器中的数据就绪。
  - Qj、Qk为0表示保留站或缓冲器单元中的Vj或Vk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。

符号说明：（举例）

MUL.D F4, F0, F2

rd rs rt

i j k

L.D F2, 45（R3）

rt imm rs

k j

S.D F3, 40（R4）

rt imm rs

k 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。

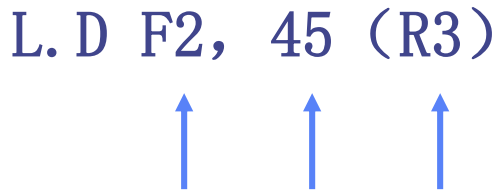
else { RS[r].Vk ← Regs[rt] ；　//第二操作数就绪。把寄存器rt中
//的操作数取到当前保留站的Vk。

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。

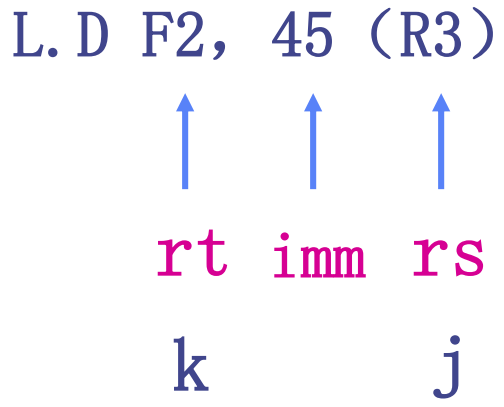


- **load和store指令**

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。
```

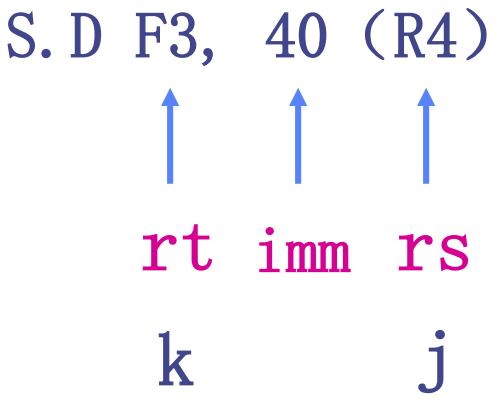


RS[r].Busy ← yes; // 置当前缓冲器单元为“忙”

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。



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。

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 ← 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：

```
∀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] ← result;               // 向该寄存器写入结果
      Qi[x] ← 0 };                    // 把该寄存器的状态置为数据就绪
∀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 ← result;                // 向该保留站的Vj写入结果
    RS[x].Qj ← 0 };                   // 置Qj为0，表示该保留站的
                                      // Vj中的操作数就绪
```

∀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 ← result;　　// 向该保留站的Vk写入结果
　 RS[x].Qk ← 0 };　　// 置Qk为0，表示该保留站的Vk中的
　　// 操作数就绪。
RS[r].Busy ← 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] ← 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 ← no;　　//释放当前缓冲器单元，将之置为空闲状态。

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。

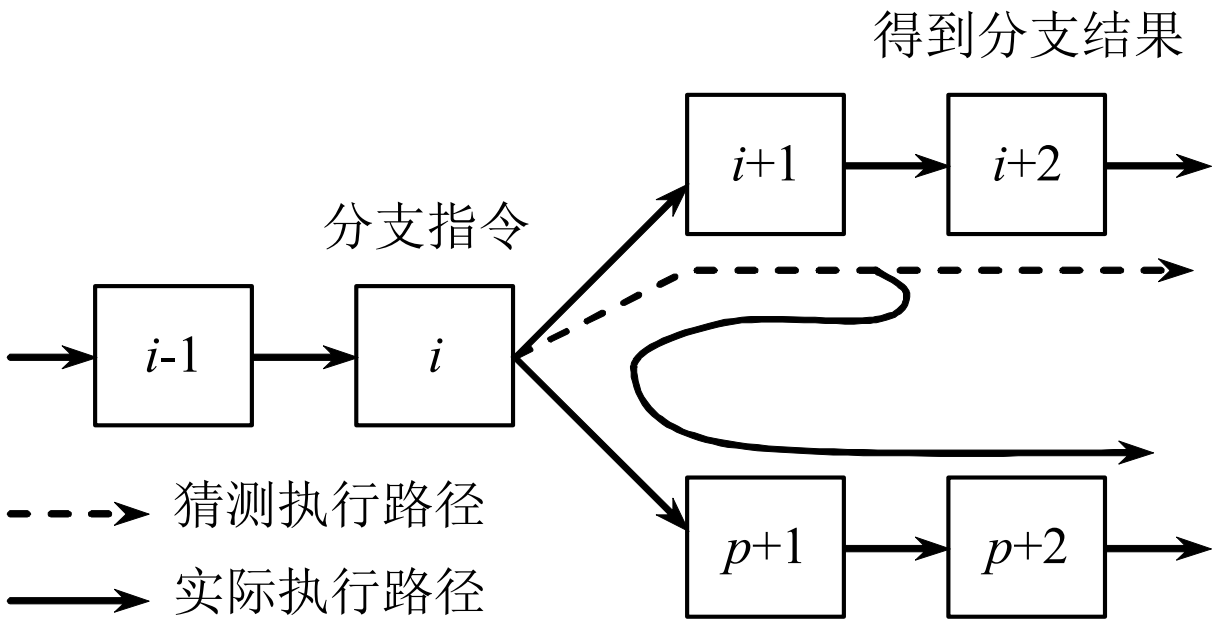
3. 动态分支预测：在程序运行时，根据分支指令过去的表现来预测其将来的行为。
   - 如果分支行为发生了变化，预测结果也跟着改变。
   - 有更好的预测准确度和适应性。
4. 分支预测的有效性取决于：
   - 预测的准确性
   - 预测正确和不正确两种情况下的分支开销

   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. 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？

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。



5.4.1 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

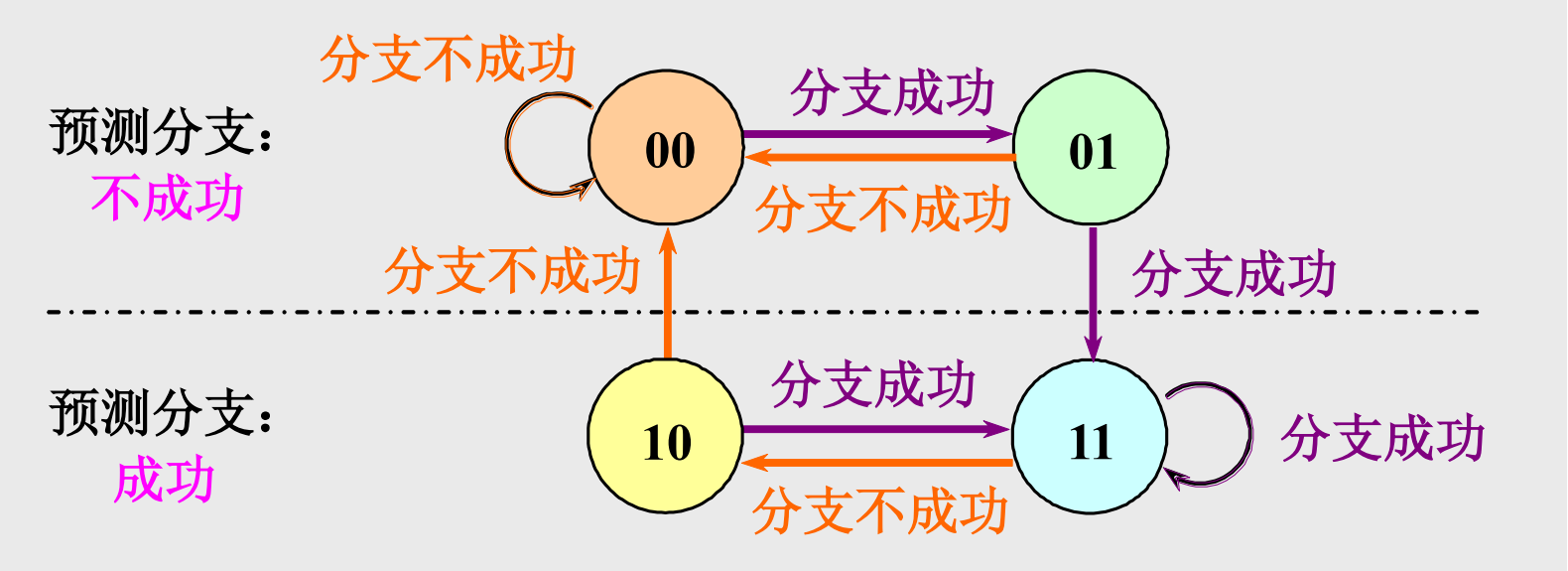
   记录分支指令最近一次的历史，BHT中只需要1位二进制位。

   （最简单）

3. 采用两位二进制位来记录历史

- 提高预测的准确度
- 研究结果表明：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：



- 两位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。

4. 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. 研究结果表明：对于SPEC89测试程序来说，具有大小为4KB的BHT的预测准确率为82%～99%。

一般来说，采用4KB的BHT就可以了。

6. 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。

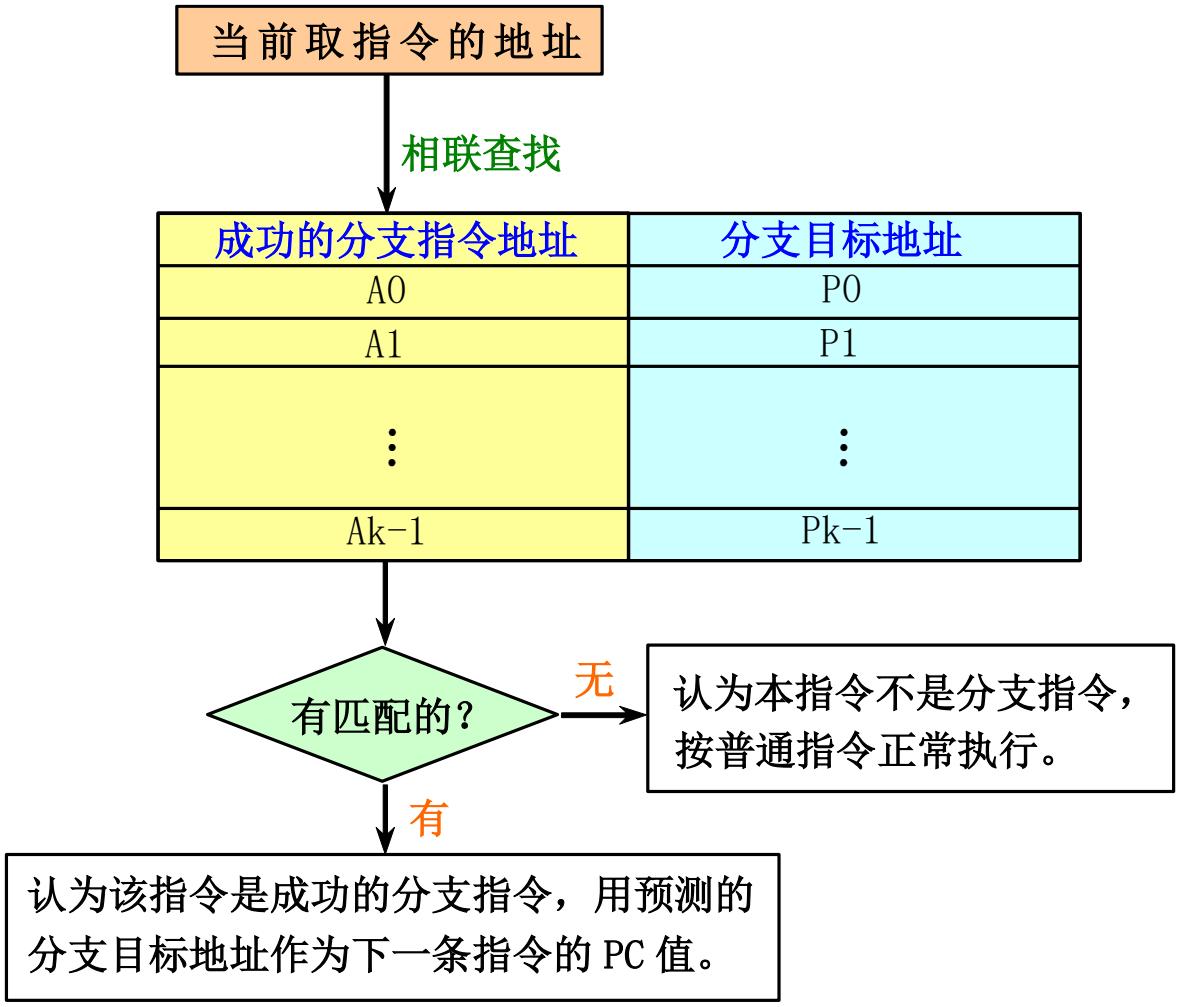
5.4.2 采用分支目标缓冲器BTB

目标：将分支的开销降为 0

方法：分支目标缓冲

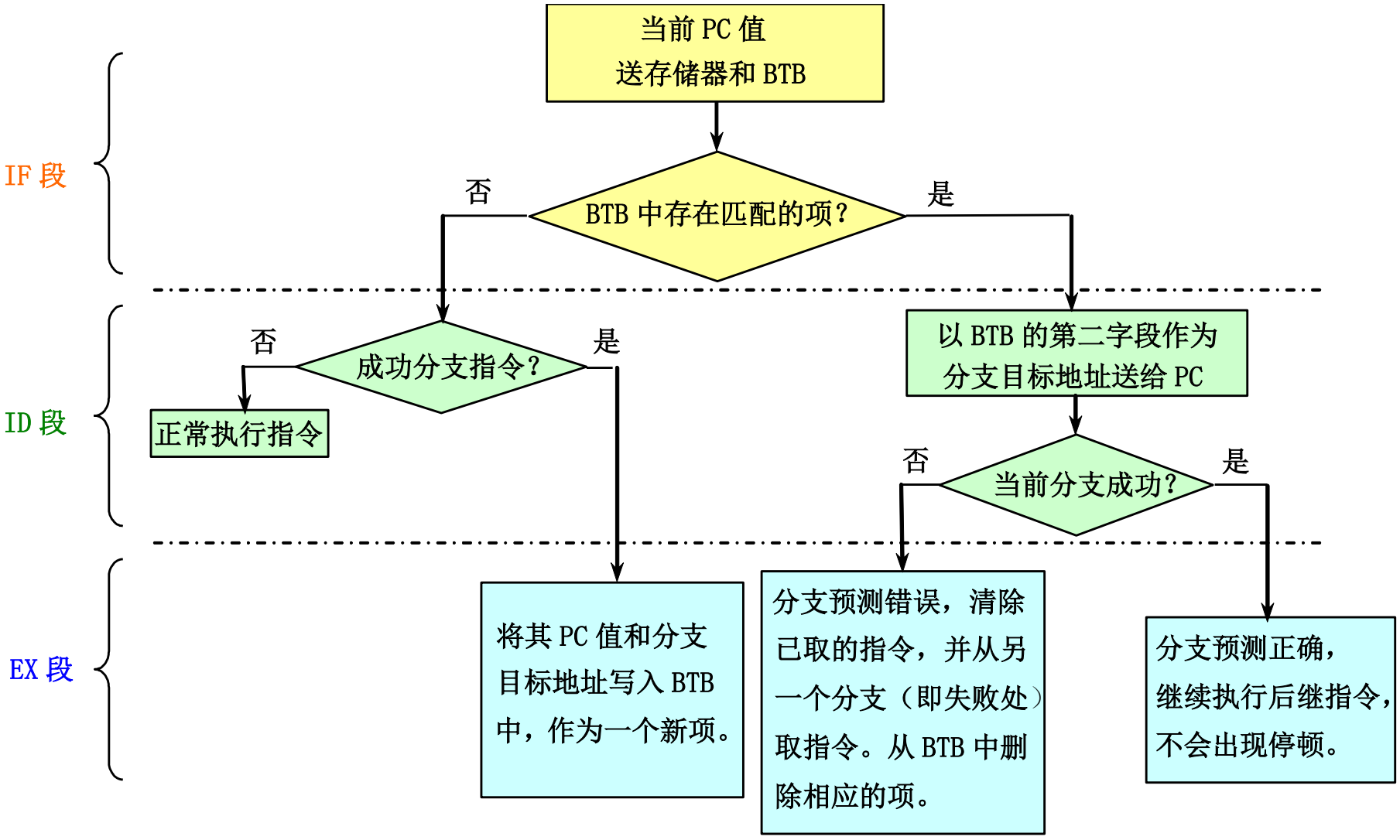
- 将分支成功的分支指令的地址和它的分支目标地址都放到一个缓冲区中保存起来，缓冲区以分支指令的地址作为标识。
- 这个缓冲区就是分支目标缓冲器（Branch-Target Buffer，简记为BTB，或者分支目标Cache （ranch-Target Cache）。

1. BTB的结构


当前取指令的地址
相联查找
成功的分支指令地址
分支目标地址
A0
P0
A1
P1
Ak-1
Pk-1
有匹配的？
无
认为本指令不是分支指令，
按普通指令正常执行。
有
认为该指令是成功的分支指令，用预测的
分支目标地址作为下一条指令的 PC 值。

- 看成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。

2. 采用BTB后，在流水线各个阶段所进行的相关操作：

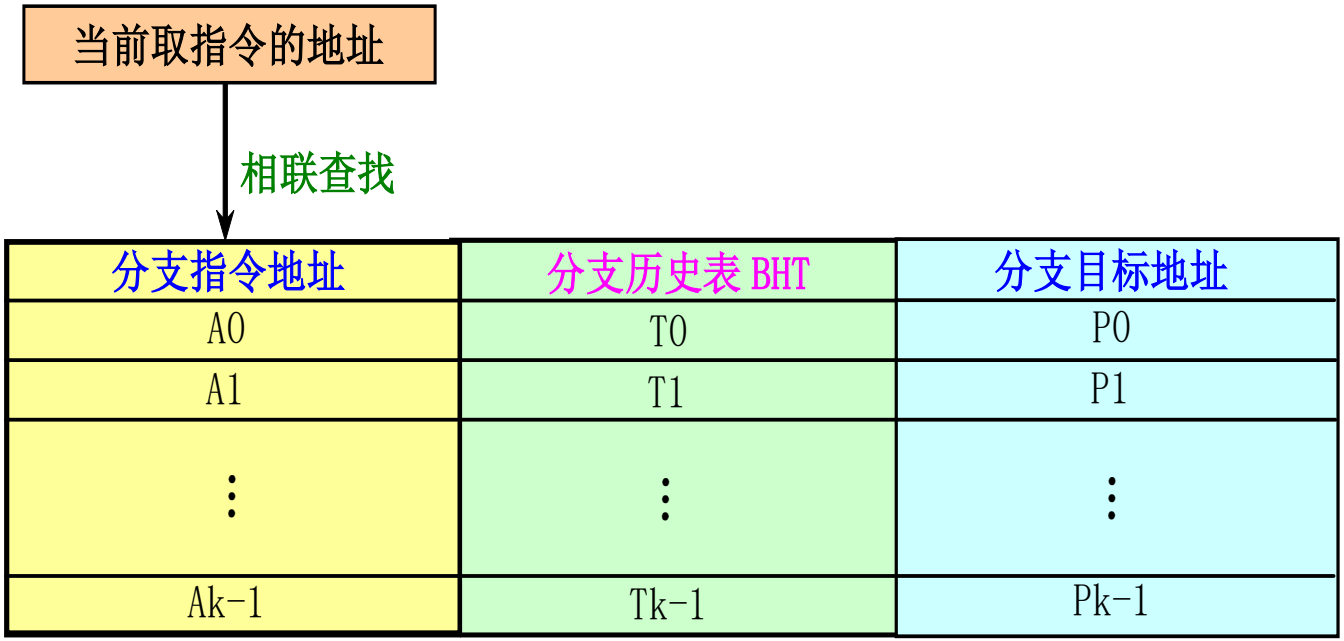

当前 PC 值
送存储器和 BTB
IF 段
否
BTB 中存在匹配的项？
是
ID 段
否
成功分支指令？
是
正常执行指令
以 BTB 的第二字段作为
分支目标地址送给 PC
否
当前分支成功？
是
EX 段
将其 PC 值和分支
目标地址写入 BTB
中，作为一个新项。
分支预测错误，清除
已取的指令，并从另
一个分支（即失败处）
取指令。从 BTB 中删
除相应的项。
分支预测正确，
继续执行后继指令，
不会出现停顿。

3. 采用BTB后，各种可能情况下的延迟 ：

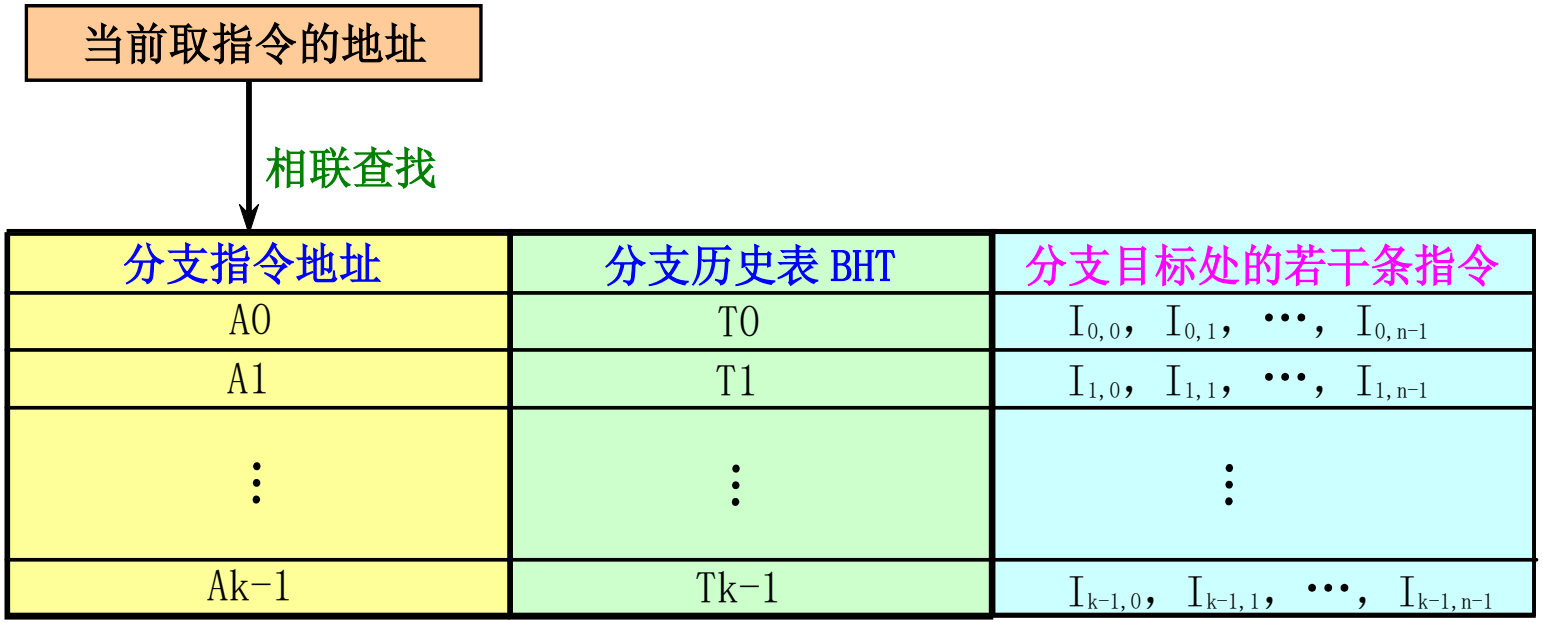
指令在BTB中？	预测	实际情况	延迟周期
是	成功	成功	0
是	成功	不成功	2
不是		成功	2
不是		不成功	0

4. 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. 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。



5.4.3 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

- 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

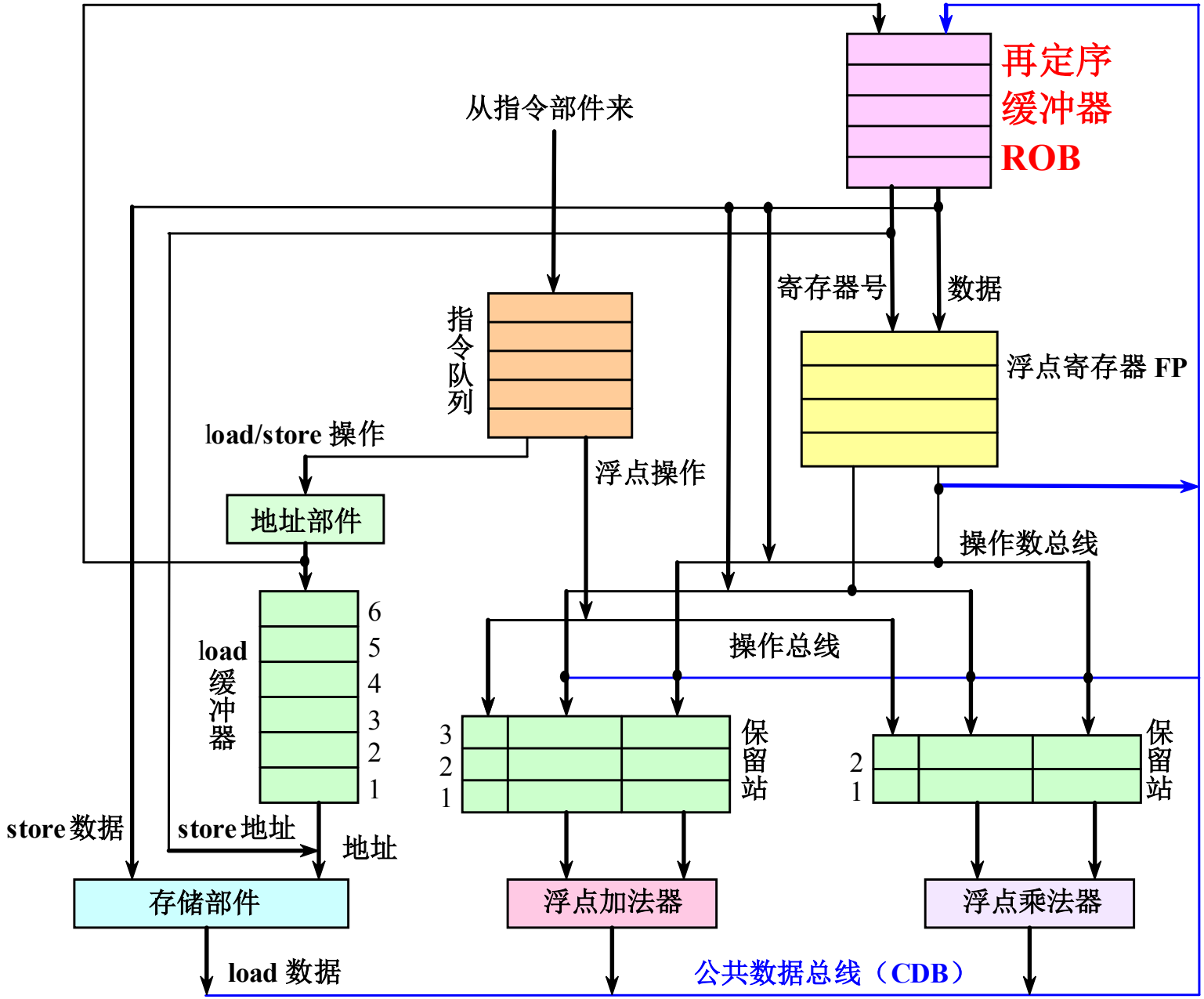
  在分支指令的结果出来后，对相应指令的前瞻执行给予确认。
  - 如果前面所做的猜测是对的，把在ROB中的结果写到寄存器或存储器。
  - 如果发现前面对分支结果的猜测是错误的，那就不予以确认，并从那条分支指令的另一条路径开始重新执行。

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。

4. 支持前瞻执行的浮点部件的结构


再定序
缓冲器
ROB
从指令部件来
寄存器号
数据
指令队列
浮点寄存器 FP
load/store 操作
浮点操作
地址部件
操作数总线
操作总线
load
缓冲器
6
5
4
3
2
1
3
2
1
保留站
2
1
保留站
store 数据
store 地址
地址
存储部件
浮点加法器
浮点乘法器
load 数据
公共数据总线（CDB）

- 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。

5. 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，其操作为：

- 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

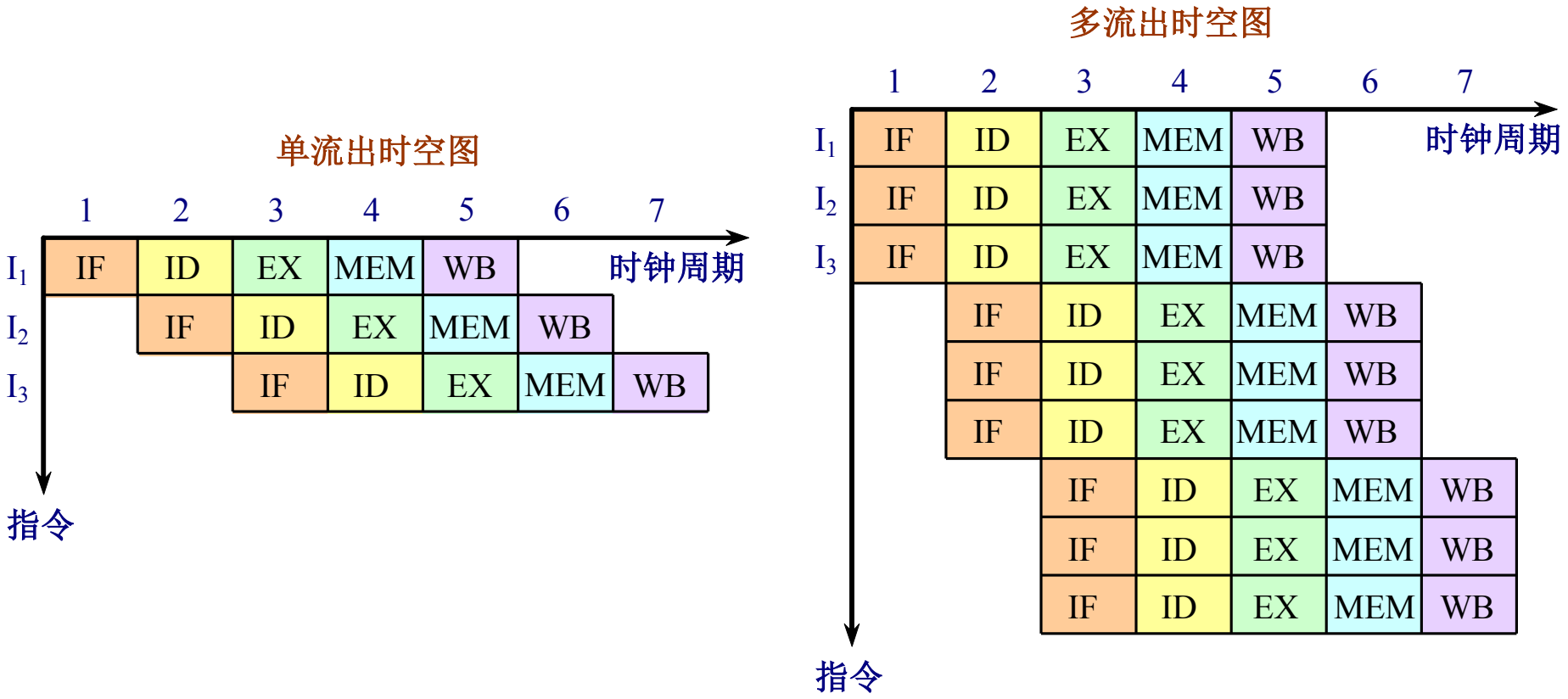
字段	浮点寄存器状态							
	F0	F2	F4	F6	F8	F10	…	F30
ROB项编号	3			6	4	5		
Busy	yes	no	no	yes	yes	yes	…	no

6. 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。
- 主要缺点：所需的硬件太复杂。

5.5 多指令流出技术

- 在每个时钟周期内流出多条指令， CPI＜1。
- 单流出和多流出处理机执行指令的时空图对比



单流出和多流出处理机执行指令的时空图

1. 多流出处理机有两种基本风格：

- 超标量（Superscalar）
  - 在每个时钟周期流出的指令条数不固定，依代码的具体情况而定。（有个上限）
  - 设这个上限为n，就称该处理机为n-流出。
  - 可以通过编译器进行静态调度，也可以基于Tomasulo算法进行动态调度。
- 超长指令字VLIW（Very Long Instruction Word）
  - 在每个时钟周期流出的指令条数是固定的，这些指令构成一条长指令或者一个指令包。
  - 指令包中，指令之间的并行性是通过指令显式地表示出来的。
  - 指令调度是由编译器静态完成的。

2. 超标量处理机与VLIW处理机相比有两个优点：
   - 超标量结构对程序员是透明的，处理机能自己检测下一条指令能否流出，不需要由编译器或专门的变换程序对程序中的指令进行重新排列；
   - 即使是没有经过编译器针对超标量结构进行调度优化的代码或是旧的编译器生成的代码也可以运行，当然运行的效果不会很好。
     - 要想达到很好的效果，方法之一：

       使用动态超标量调度技术。

3. 下表列出了一些基本的多流出技术、这些技术的特点以及采用这些技术的处理机实例。

技 术	流出结构	冲突检测	调 度	主要特点	处理机实例
超标量（静态）	动态	硬件	静态	按序执行	Sun UltraSPARCⅡ/Ⅲ
超标量（动态）	动态	硬件	动态	部分乱序执行	IBM Power2
超标量（猜测）	动态	硬件	带有前瞻的动态调度	带有前瞻的乱序执行	Pentium Ⅲ/4, MIPS R10K, Alpha 21264, HP PA 8500, IBM RS64Ⅲ
VLIW /LIW	静态	软件	静态	流出包之间没有冲突	Trimedia，i860
EPIC	主要是静态	主要是软件	主要是静态	相关性被编译器显式地标记出来	Itanium

5.5.1 基于静态调度的多流出技术

- 在典型的超标量处理器中，每个时钟周期可流出1到8条指令。
- 指令按序流出，在流出时进行冲突检测。

  由硬件检测当前流出的指令之间是否存在冲突以及当前流出的指令与正在执行的指令是否有冲突。

举例：一个4-流出的静态调度超标量处理机

- 在取指令阶段，流水线将从取指令部件收到1～4条指令（称为流出包）。
  - 在一个时钟周期内，这些指令有可能是全部都能流出，也可能是只有一部分能流出。

- 流出部件检测结构冲突或者数据冲突。

  一般分两阶段实现：

  - 第一段：进行流出包内的冲突检测，选出初步判定可以流出的指令；
  - 第二段：检测所选出的指令与正在执行的指令是否有冲突。

MIPS处理机是怎样实现超标量的呢？

假设：每个时钟周期流出两条指令：

1条整数型指令＋1条浮点操作指令

- 其中：把load指令、store指令、分支指令归类为整数型指令。

1. 要求：

   同时取两条指令（64位），译码两条指令（64位）。

2. 对指令的处理包括以下步骤：
   - 从Cache中取两条指令；
   - 确定那几条指令可以流出（0～2条指令）；
   - 把它们发送到相应的功能部件。

3. 双流出超标量流水线中指令执行的时空图
   - 假设：所有的浮点指令都是加法指令，其执行时间为两个时钟周期。
   - 为简单起见，图中总是把整数指令放在浮点指令的前面。

指令类型	流水线工作情况							
整数指令	IF	ID	EX	MEM	WB			
浮点指令	IF	ID	EX	EX	MEM	WB		
整数指令		IF	ID	EX	MEM	WB		
浮点指令		IF	ID	EX	EX	MEM	WB	
整数指令			IF	ID	EX	MEM	WB	
浮点指令			IF	ID	EX	EX	MEM	WB
整数指令				IF	ID	EX	MEM	WB
浮点指令				IF	ID	EX	EX	MEM

4. 采用“1条整数型指令＋1条浮点指令”并行流出的方式，需要增加的硬件很少。
5. 浮点load或浮点store指令将使用整数部件，会增加对浮点寄存器的访问冲突。

   增设一个浮点寄存器的读/写端口。
6. 由于流水线中的指令多了一倍，定向路径也要增加。

7. 限制超标量流水线的性能发挥的障碍

- load指令
  - load后续3条指令都不能使用其结果，否则就会引起停顿。
- 分支延迟
  - 如果分支指令是流出包中的第一条指令，则其延迟是3个时钟周期；
  - 否则就是流出包中的第二条指令，其延迟就是两个时钟周期。

5.5.2 基于动态调度的多流出技术

- 扩展Tomasulo算法：支持双流出超标量流水线
  - 每个时钟周期流出两条指令；
  - 一条是整数指令，另一条是浮点指令。

1. 采用一种比较简单的方法：
   - 指令按顺序流向保留站，否则会破坏程序语义。
   - 将整数所用的表结构与浮点用的表结构分离开，分别进行处理，这样就可以同时地流出一条浮点指令和一条整数指令到各自的保留站。

2. 有两种不同的方法可以实现多流出

关键在于：对保留站的分配和对流水线控制表格的修改。

- 在半个时钟周期里完成流出步骤，这样一个时钟周期就能处理两条指令。
- 设置一次能同时处理两条指令的逻辑电路。

现代的流出4条或4条以上指令的超标量处理机经常是两种方法都采用。

例5.5 对于采用了Tomasulo算法和多流出技术的MIPS流水线，考虑以下简单循环的执行。该程序把F2中的标量加到一个向量的每个元素上。

```
Loop: L.D     F0, 0(R1)        // 取一个数组元素放入F0
      ADD.D   F4, F0, F2       // 加上在F2中的标量
      S.D     F4, 0(R1)        // 存结果
      DADDIU R1,R1, #-8
                               // 将指针减少8（每个数据占8个字节）
      BNE R1,R2,Loop
      // 若R1不等于R2，表示尚未结束，转移到Loop继续。
```

现做以下假设：

- 每个时钟周期能流出一条整数指令和一条浮点指令，即使它们相关也是如此。
- 有一个整数部件，用于整数ALU运算和地址计算；并且对于每一种浮点操作类型都有一个独立的流水化了的浮点功能部件。
- 指令流出和写结果各占用一个时钟周期。
- 具有动态分支预测部件和一个独立的计算分支条件的功能部件。
- 跟大多数动态调度处理器一样，写回段的存在意味着实际的指令延迟会比按序流动的简单流水线多一个时钟周期。

- 所以，从产生结果数据的源指令到使用该结果数据的指令之间的延迟为：整数运算一个周期，load两个周期，浮点加法运算3个周期。

1. 请列出该程序前面3遍循环中各条指令的流出、开始执行和将结果写到CDB上的时间。

2. 如果分支指令单流出，没有采用延迟分支，但分支预测是完美的。请列出整数部件、浮点部件、数据Cache以及CDB的资源使用情况。

解 执行时，该循环将动态展开，并且只要可能就流出两条指令。

表中列出了各指令执行到几个操作点的时间及资源的使用情况。

遍数	指　　令		流出	执行	访存	写CDB	说明
1	L.D	F0,0（R1）	1	2	3	4	流出第一条指令
1	ADD.D	F4,F0,F2	1	5		8	等待L.D的结果
1	S.D	F4,0（R1）	2	3	9		等待ADD.D的结果
1	DADDIU	R1,R1,#-8	2	4		5	等待ALU
1	BNE	R1,R2,Loop	3	6			等待DADDIU的结果
2	L.D	F0,0（R1）	4	7	8	9	等待BNE完成
2	ADD.D	F4,F0,F2	4	10		13	等待L.D的结果
2	S.D	F4,0（R1）	5	8	14		等待ADD.D的结果
2	DADDIU	R1,R1,#-8	5	9		10	等待ALU
2	BNE	R1,R2,Loop	6	11			等待DADDIU的结果
3	L.D	F0,0（R1）	7	12	13	14	等待BNE完成
3	ADD.D	F4,F0,F2	7	15		18	等待L.D的结果
3	S.D	F4,0（R1）	8	13	19		等待ADD.D的结果
3	DADDIU	R1,R1,#-8	8	14		15	等待ALU
3	BNE	R1,R2,Loop	9	16			等待DADDIU的结果

时钟周期	整型ALU	浮点ALU	数据Cache	CDB
2	1/L.D			
3	1/S.D		1/L.D	
4	1/DADDIU			1/L.D
5		1/ADD.D		1/DADDIU
6				
7	2/L.D			
8	2/S.D		2/L.D	1/ADD.D
9	2/DADDIU		1/S.D	2/L.D
10		2/ADD.D		2/DADDIU
11				
12	3/L.D			

时钟周期	整型ALU	浮点ALU	数据Cache	CDB
13	3/S.D		3/L.D	2/ADD.D
14	3/DADDIU		2/S.D	3/L.D
15		3/ADD.D		3/DADDIU
16				
17				
18				3/ADD.D
19			3/S.D	
20				

可以看出：

- 每3个时钟周期就执行一个新循环，每个循环5条指令。

  IPC＝5/3＝1.67条/拍

- 虽然指令的流出率比较高，但是执行效率并不是很高。
  - 16拍共执行15条指令，
  - 平均指令执行速度为15/16＝0.94条/拍。
- 原因是浮点运算少，ALU部件成了瓶颈。

  解决方法：增加一个加法器，把ALU功能和地址运算功能分开。

3. 上述双流出动态调度流水线的性能受限于以下3个因素：

- 整数部件和浮点部件的工作负载不平衡，没有充分发挥出浮点部件的作用。

  应该设法减少循环中整数型指令的数量。

- 每个循环叠代中的控制开销太大。
  - 5条指令中有两条指令是辅助指令；
  - 应该设法减少或消除这些指令。
- 控制相关使得处理机必须等到分支指令的结果出来后才能开始下一条L. D指令的执行。

5.5.3 超长指令字技术

1. 把能并行执行的多条指令组装成一条很长的指令；
   （100多位到几百位）
2. 设置多个功能部件；
3. 指令字被分割成一些字段，每个字段称为一个操作槽，直接独立地控制一个功能部件；
4. 在VLIW处理机中，在指令流出时不需要进行复杂的冲突检测，而是依靠编译器全部安排好了。

5. VLIW存在的一些问题

- 程序代码长度增加了
  - 提高并行性而进行的大量的循环展开；
  - 指令字中的操作槽并非总能填满。

  解决：采用指令共享立即数字段的方法，或者采用指令压缩存储、调入Cache或译码时展开的方法。
- 采用了锁步机制

  任何一个操作部件出现停顿时，整个处理机都要停顿。
- 机器代码的不兼容性

5.5.4 多流出处理器受到的限制

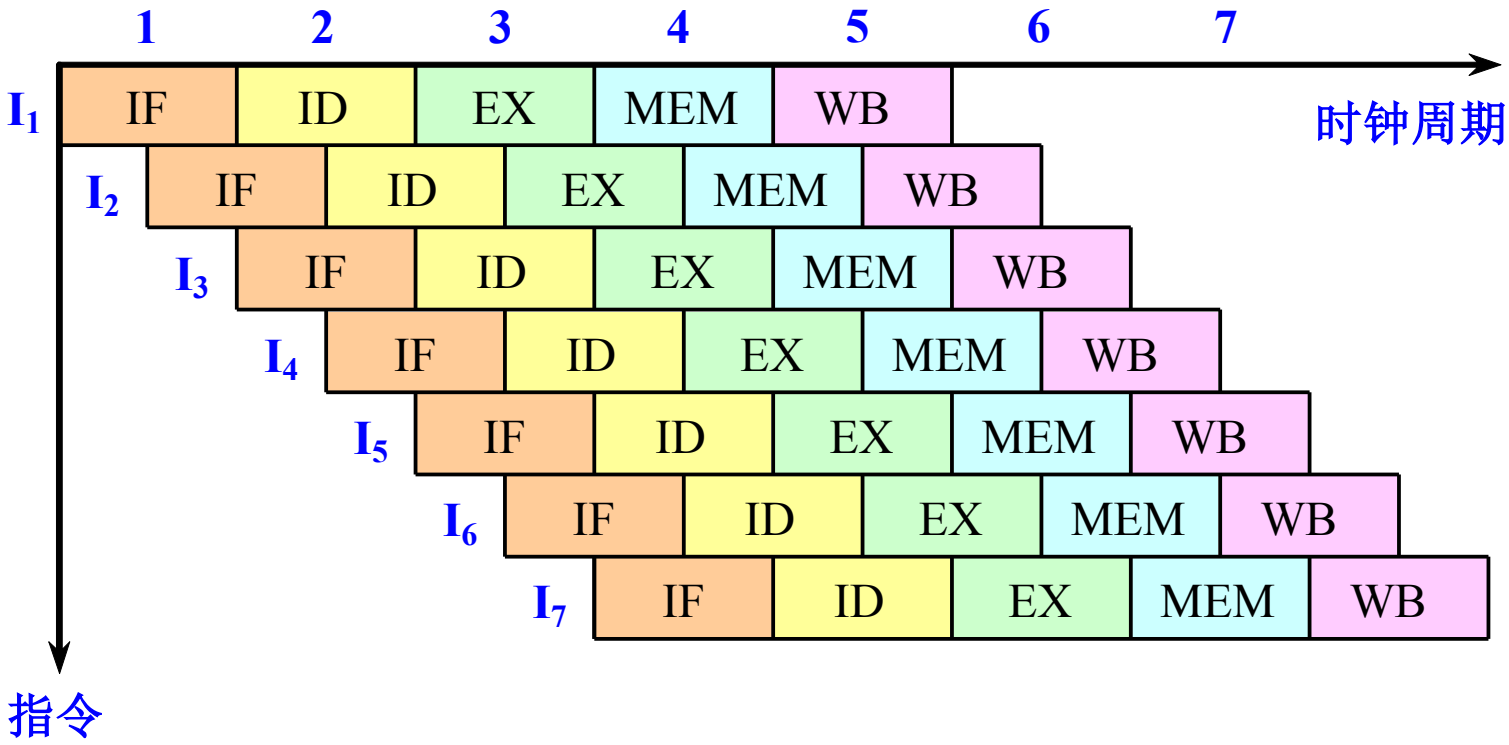
指令多流出处理器受哪些因素的限制呢？

主要受以下3个方面的影响：

- 程序所固有的指令级并行性；
- 硬件实现上的困难；
- 超标量和超长指令字处理器固有的技术限制。

5.5.5 超流水线处理机

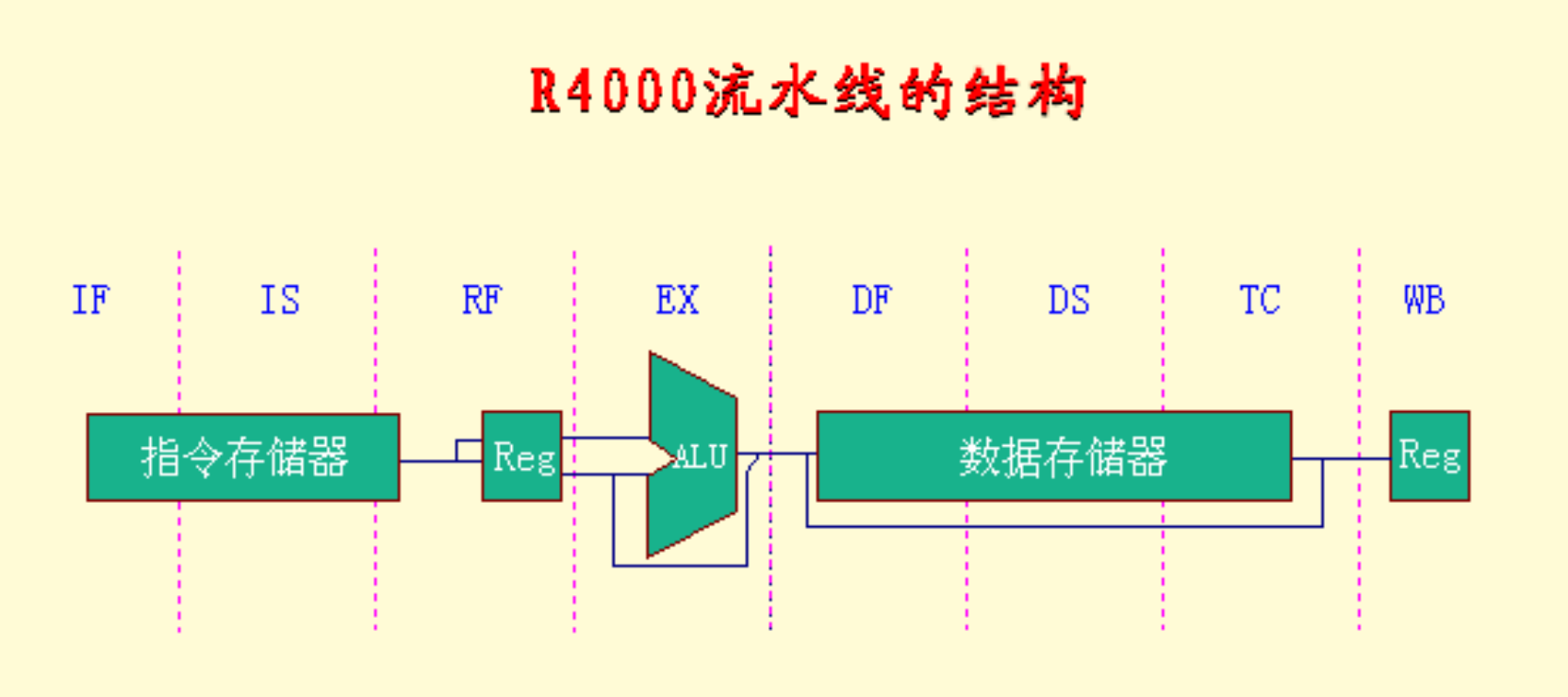
1. 将每个流水段进一步细分，这样在一个时钟周期内能够分时流出多条指令。这种处理机称为超流水线处理机。
2. 对于一台每个时钟周期能流出n条指令的超流水线计算机来说，这n条指令不是同时流出的，而是每隔1/n个时钟周期流出一条指令。
   - 实际上该超流水线计算机的流水线周期为1/n个时钟周期。
3. 一台每个时钟周期分时流出两条指令的超流水线计算机的时空图。


1
2
3
4
5
6
7
时钟周期
I1
I2
I3
I4
I5
I6
I7
IF
ID
EX
MEM
WB
指令

4. 在有的资料上，把指令流水线级数为8或8以上的流水线处理机称为超流水线处理机。
5. 典型的超流水线处理器：SGI公司的MIPS系列R4000
   - R4000微处理器芯片内有2个Cache：
     - 指令Cache和数据Cache
     - 容量都是8KB
     - 每个Cache的数据宽度为64位
   - R4000的核心处理部件：整数部件
     - 一个32×32位的通用寄存器组
     - 一个算术逻辑部件（ALU）
     - 一个专用的乘法/除法部件

- 浮点部件
  - 一个执行部件
    - 浮点乘法部件
    - 浮点除法部件
    - 浮点加法/转换/求平方根部件

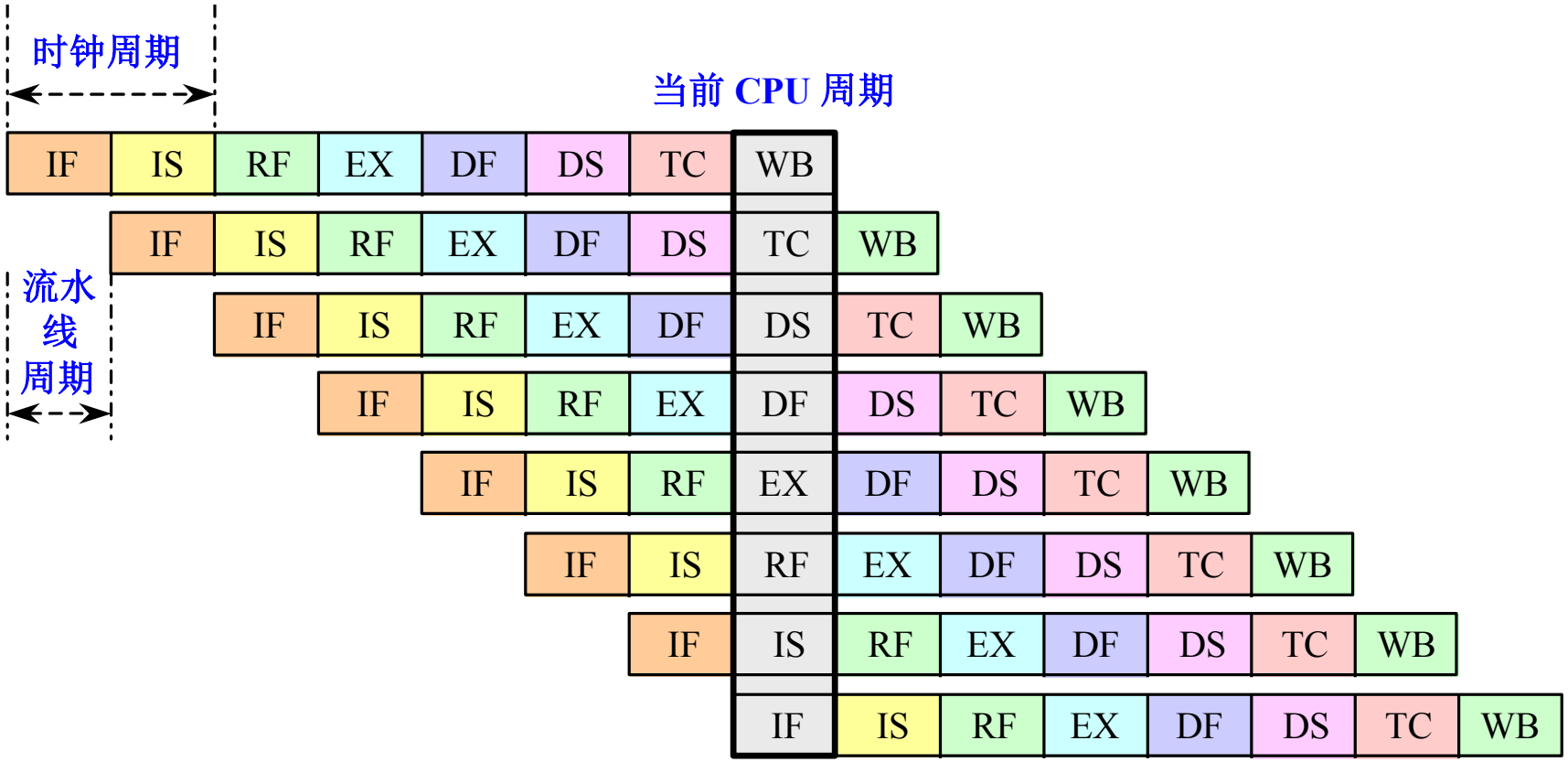
      （它们可以并行工作）
  - 一个16×64位的浮点通用寄存器组。浮点通用寄存器组也可以设置成32个32位的浮点寄存器。
- R4000的指令流水线有8级


R4000流水线的结构
IF
IS
RF
EX
DF
DS
TC
WB
指令存储器
Reg
ALU
数据存储器
Reg

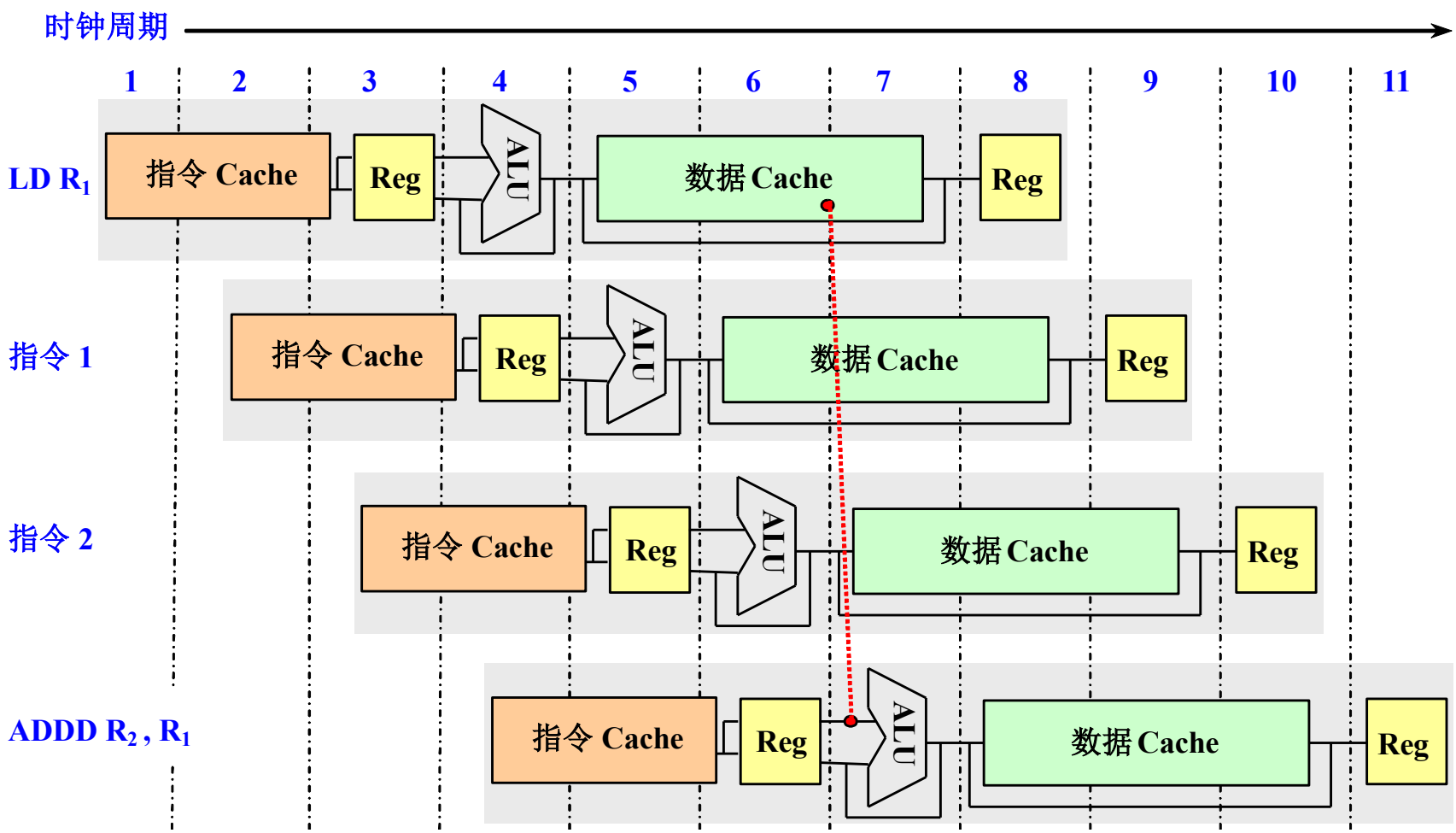
各级的功能

- IF：取指令的前半步，根据PC值去启动对指令Cache的访问。
- IS：取指令的后半步，在这一级完成对指令Cache的访问。
- RF：指令译码，访问寄存器组读取操作数，冲突检测，并判断指令Cache是否命中。
- EX：指令执行。包括：有效地址计算，ALU操作，分支目标地址计算，条件码测试。
- DF：取数据的前半步，启动对数据Cache的访问。
- DS：取数据的后半步，在这一级完成对数据Cache的访问。
- TC：标识比较，判断对数据Cache的访问是否命中。
- WB：load指令或运算型指令把结果写回寄存器组。

➢ MIPS R4000指令流水线时空图



➢ 载入延迟为两个时钟周期



第6章 指令级并行的开发
——软件方法

沈 立

www.GotoSchool.net

6.1 基本指令调度及循环展开

6.1.1 指令调度的基本方法

1. 指令调度：找出不相关的指令序列，让它们在流水线上重叠并行执行。
2. 制约编译器指令调度的因素
   - 程序固有的指令级并行
   - 流水线功能部件的延迟

表6.1本节使用的浮点流水线的延迟

产生结果的指令	使用结果的指令	延迟(cycles)
浮点计算	另一个浮点计算	3
浮点计算	浮点store(S.D)	2
浮点Load(L.D)	浮点计算	1
浮点Load(L.D)	浮点store(S.D)	0

例6.1 对于下面的源代码，转换成MIPS汇编语言，在不进行指令调度和进行指令调度两种情况下，分析其代码一次循环所需的执行时间。

```
for (i=1000; i>0; i--)
    x[i] = x[i] + s;
```

解：

先把该程序翻译成MIPS汇编语言代码

```
Loop:   L.D     F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
        DADDIU  R1, R1, #-8
        BNE     R1, R2, Loop
```

- 在不进行指令调度的情况下，根据表中给出的浮点流水线中指令执行的延迟，程序的实际执行情况如下：

			指令流出时钟
Loop:	L.D	F0, 0(R1)	1
	(空转)		2
	ADD.D	F4, F0, F2	3
	(空转)		4
	(空转)		5
	S.D	F4, 0(R1)	6
	DADDIU	R1, R1, #-8	7
	(空转)		8
	BNE	R1, R2, Loop	9
	(空转)		10

- 在用编译器对上述程序进行指令调度以后，程序的执行情况如下：

			指令流出时钟
Loop：	L.D	F0, 0(R1)	1
	DADDIU	R1, R1, #-8	2
	ADD.D	F4, F0, F2	3
	(空转)		4
	BNE	R1, R2, Loop	5
	S.D	F4, 8(R1)	6

进一步分析：

- 编译时指令调度是怎样减少整个指令序列在流水线上的执行时间的？
- 指令调度能否跨越分支边界？
- 怎样提高整个执行过程中有效操作的比率？

6.1.2 循环展开

1. 循环展开

- 把循环体的代码复制多次并按顺序排放， 然后相应调整循环的结束条件。
- 开发循环级并行的有效方法

例6.2 将例6.1中的循环展开3次得到4个循环体，然后对展开后的指令序列在不调度和调度两种情况下，分析代码的性能。假定R1的初值为32的倍数，即循环次数为4的倍数。消除冗余的指令，并且不要重复使用寄存器。

- 展开后没有调度的代码如下(需要分配寄存器)

			指令流出时钟
Loop:	L.D	F0, 0(R1)	1
	（空转）		2
	ADD.D	F4, F0, F2	3
	（空转）		4
	（空转）		5
	S.D	F4, 0(R1)	6
	L.D	F6, -8(R1)	7
	（空转）		8
	ADD.D	F8, F6, F2	9
	（空转）		10
	（空转）		11
	S.D	F8, -8(R1)	12
	L.D	F10, -16(R1)	13
	（空转）		14

			指令流出时钟
	ADD.D	F12, F10, F2	15
	（空转）		16
	（空转）		17
	S.D	F12, -16（R1）	18
	L.D	F14, -24（R1）	19
	（空转）		20
	ADD.D	F16, F14, F2	21
	（空转）		22
	（空转）		23
	S.D	F16, -24（R1）	24
	DADDIU	R1, R1, # -32	25
	（空转）		26
	BNE	R1, R2, Loop	27
	（空转）		28

50%是空转周期！

- 调度后的代码如下：

			指令流出时钟
Loop:	L.D	F0, 0（R1）	1
	L.D	F6, -8（R1）	2
	L.D	F10, -16（R1）	3
	L.D	F14, -24（R1）	4
	ADD.D	F4, F0, F2	5
	ADD.D	F8, F6, F2	6
	ADD.D	F12, F10, F2	7
	ADD.D	F16, F14, F2	8
	S.D	F4, 0（R1）	9
	S.D	F8, -8（R1）	10
	DADDIU	R1, R1, # -32	12
	S.D	F12, 16（R1）	11
	BNE	R1, R2, Loop	13
	S.D	F16, 8（R1）	14

结论：通过循环展开、寄存器重命名和指令调度，可以有效开发出指令级并行。

没有空转周期！

2. 循环展开和指令调度的注意事项

- 保证正确性
- 注意有效性
- 使用不同的寄存器
- 删除多余的测试指令和分支指令，并对循环结束代码和新的循环体代码进行相应的修正。
- 注意对存储器数据的相关性分析
- 注意新的相关性

6.2 跨越基本块的静态指令调度

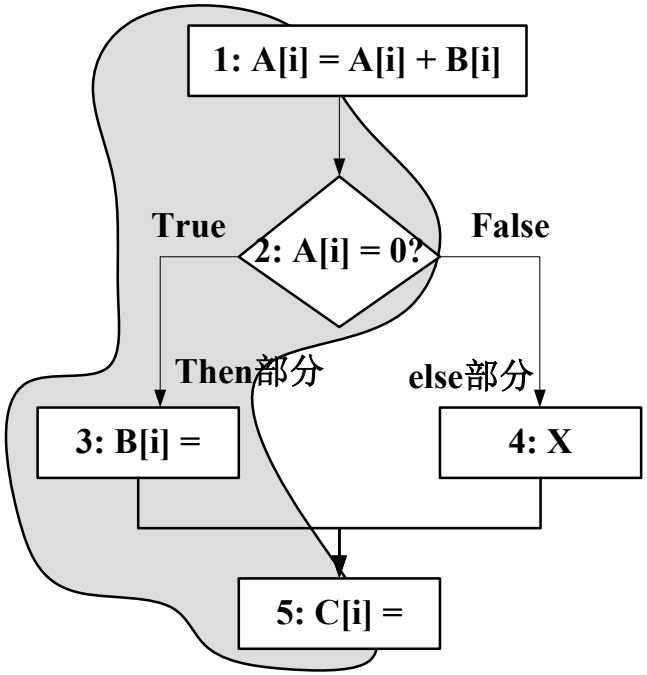
6.2.1 全局指令调度

1. 概述

- 目标：在保持原有数据相关和控制相关不变的前提下，尽可能地缩短包含分支结构的代码段的总执行时间。
  - 单流出处理器——减少指令数
  - 多流出处理器——缩短关键路径长度
- 基本思想：在循环体内的多个基本块间移动指令，扩大那些执行频率较高的基本块的体积。

2. 实例分析

- 由于分支条件为true(转移)的概率大，全局指令调度时会将语句1、2、3、5合并为一个更大的基本块。
  - 如何保证分支条件为**false**时结果依然正确？
  - 如何将语句**3**和**5**调度到语句**2**之前？



一个分支结构的代码段

- 将上图中的代码转换为下面的MIPS汇编指令

```
            LD      R4，0(R1)       // 取A
            LD      R5，0(R2)       // 取B
            DADDU   R4，R4，R5      // A=A+B
            SD      0(R1)，R4       // 存A
            BNEZ    R4，elsepart    // A=0则转移
            X                       // 代码段X，基本块elsepart
            J       join
thenpart：                          // 基本块thenpart
            SD      …，0(R2)        // 指令I1，对应语句3
join：
            SD      …，0(R3)        // 指令I2，对应语句5
```

- 调度指令I1
  - 直接将I1移到BEQZ前是否会产生错误结果？
  - 向基本块elsepart中增加补偿代码
  - 补偿代码有可能带来额外时间开销
- 调度指令I2
  - 将I2移动到基本块thenpart中，同时复制到elsepart中。
  - 若不影响执行结果，将I2调度到BEQZ前，同时删除elsepart中的副本。

3. 全局指令调度是一个很复杂的问题

以I1的调度为例：

- 需要确定分支中基本块thenpart和elsepart的执行频率各是多少？
- 在分支语句前完成I1所需的开销是多大？
- 调度I1是否能够缩短thenpart块的执行时间？
- I1是否是最佳的被调度对象？
- 是否需要向elsepart块中增加补偿代码，补偿代码开销如何？怎样生成补偿代码？

6.2.2 踪迹调度

1. 概述

- 踪迹（trace）：程序执行的指令序列，通常由一个或多个基本块组成，trace内可以有分支，但一定不能包含循环。
- 踪迹调度（trace scheduling）会优化执行频率高的trace，减少其执行开销。由于需要添加补偿代码以确保正确性，那些执行频率较低的trace的开销反而会有所增加。
- 踪迹调度非常适合多流出处理器。

2. 踪迹调度的步骤

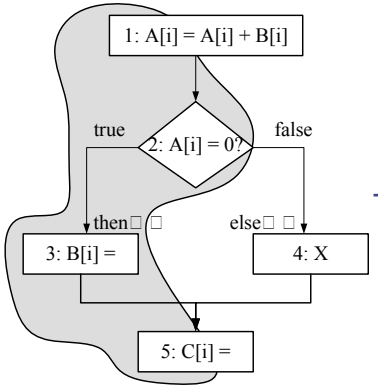
分为两步：踪迹选择和踪迹压缩

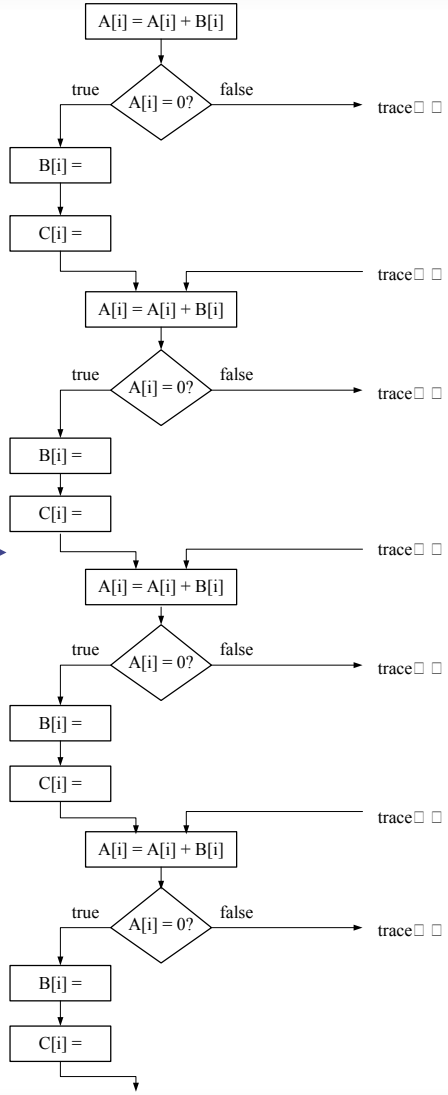
- 踪迹选择
  - 从程序的控制流图中选择执行频率较高的路径，每条路径就是一条trace。
  - 处理转移成功与失败概率相差较大的情况
  - 循环结构：循环展开
  - 分支结构：根据典型输入集下的运行统计信息

➢ 踪迹选择——实例分析

将左边的循环展开4次并把阴影部分(执行频率高)拼接在一起就可以得到一条trace；

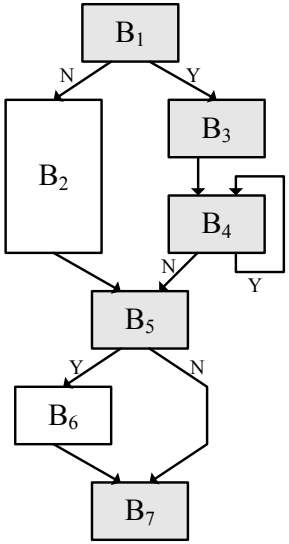
一条trace可以有多个入口和多个出口。





踪迹压缩

- 对已生成的trace进行指令调度和优化，尽可能地缩短其执行时间；
- 跨越trace内部的入口或出口调度指令时必须非常小心，有时还需要增加补偿代码 。



(a) □ □ □ □

B_1: x = x + 1
y = x - y
if x<5 goto B_3

B_2: z = x * z
y = y + 1
goto B_5

B_3: y = 2 * y
x = x - 2

□ □ □

B_1: x = x + 1

if x<5 goto B_3

B_2: y = x - y
z = x * z
y = y + 1
goto B_5

B_3: y = x - y
y = 2 * y
x = x - 2

□ □ □

(b) □ □ □ □ □ □ □ □ □ □ □ B_1□ B_2□ B_3

三条trace：B_1-B_3、B_4以及B_5-B_7

指令“y = x - y”被从B_1调度到B_3中，跨越了trace的一个出口；

需要向块B_2中增加补偿代码，即将指令“y = x - y”复制到B_2的第一条指令之前 。

3. 踪迹调度的性能特点

- 踪迹调度能够提升性能的最根本原因在于选出的trace都是执行频率很高的路径，减少它们的执行开销有助于缩短程序的总执行时间。
- 对于某些应用，补偿代码引起的开销很有可能降低踪迹调度的优化效果。
- 踪迹调度会大大增加编译器的实现复杂度。

6.2.3 超块调度

1. 概述

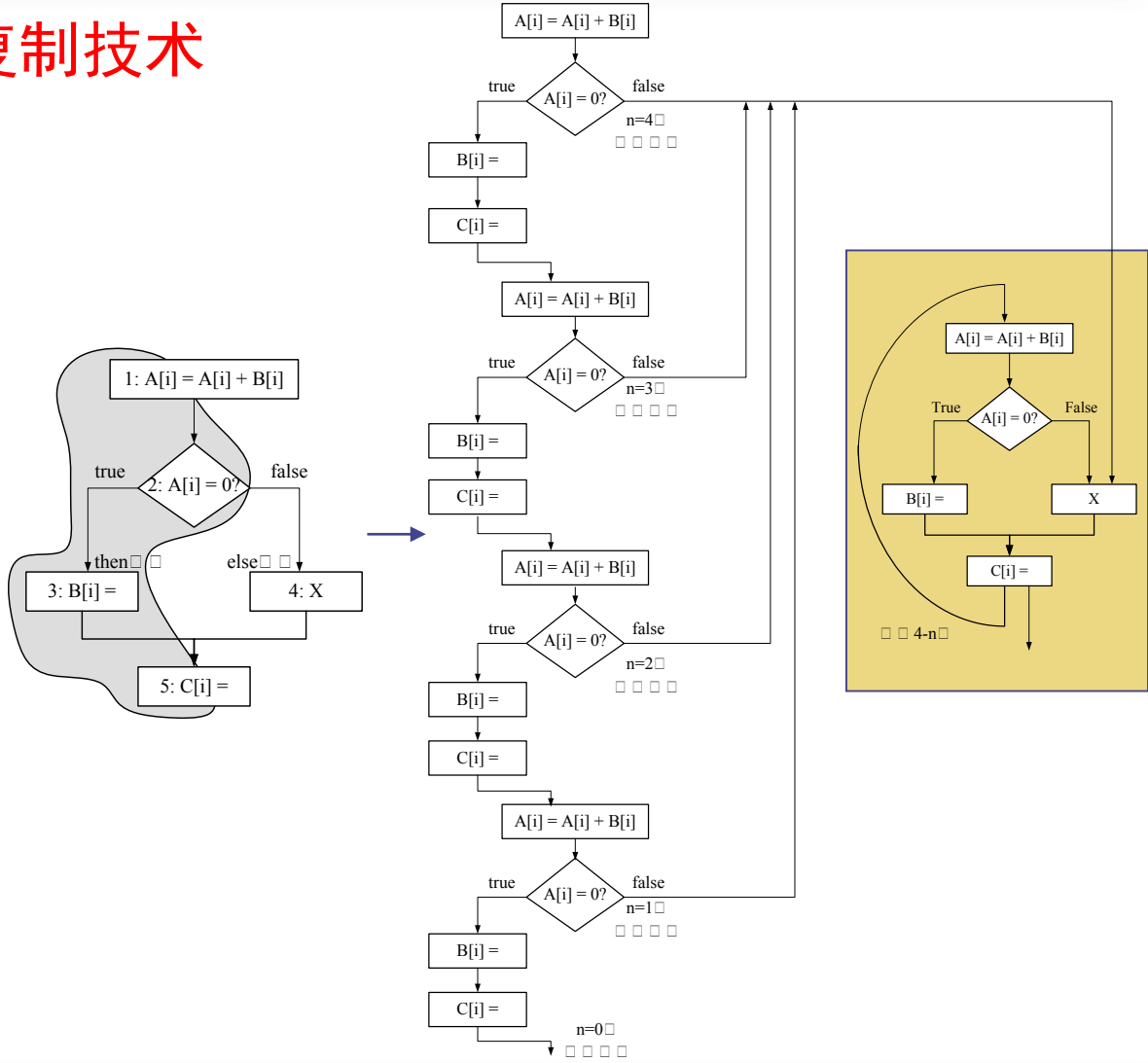
- 在踪迹调度中，如果trace入口或出口位于trace内部，编译器生成补偿代码的难度将大大增加，而且编译器很难评估这些补偿代码究竟会带来多少性能损失。
- 超块（superblock）是只能拥有一个入口，但可以拥有多个出口的结构
- 超块的构造过程与trace相似，但怎样确保只有一个入口？

2. 超块构造——尾复制技术

将左边的循环展开4次并把阴影部分(执行频率高)拼接在一起就可以得到一个超块；

超块有1个入口和5个出口(n=4/3/2/1/0)。

除了n=0外，从其他4个出口退出超块后，还需要继续完成余下的n次叠代(黄色部分)。



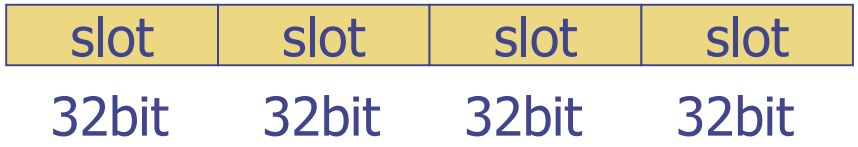
3. 超块调度的性能特点

- 尾复制技术简化了补偿代码的生成过程，并降低了指令调度的复杂度。
- 超块结构目标代码的体积也大大增加。
- 补偿代码的生成使得编译过程更加复杂，而且由于无法准确评估由补偿代码引起的时间开销，这限制方法超块调度的应用范围。

6.3 静态多指令流出：VLIW技术

1. VLIW vs. 超标量

- 在动态调度的超标量处理器中，相关检测和指令调度基本都由硬件完成。
- 在静态调度的超标量处理器中，部分相关检测和指令调度工作交由编译器完成。
- 在VLIW处理器中，相关检测和指令调度工作全部由编译器完成，它需要更“智能”的编译器。



128bit VLIW指令

2. 实例分析

例6.3 假设某VLIW处理器每个时钟周期可以同时流出5个操作，包括2个访存操作，2个浮点操作以及1个整数或分支操作。将例6.1中的代码循环展开，并调度到该VLIW处理器上执行。循环展开次数不定，但至少要能够保证消除所有流水线“空转”周期，同时不考虑分支延迟。

解：循环被展开7次，经调度后可以消除所有流水线“空转”。在不考虑分支延迟的情况下，每执行一个叠代需要9个时钟周期，计算出7个结果，即平均每得到一个结果需要1.29个周期。

VLIW的不足：

- 编码效率仅比50%略高一些
- 所需要的寄存器数量也大大增加

访存操作1	访存操作2	浮点操作1	浮点操作2	整数分支操作
L.D F0，0（R1）	L.D F6，-8（R1）	nop	nop	nop
L.D F10，-16（R1）	L.D F14，-24（R1）	nop	nop	nop
L.D F18，-32（R1）	L.D F22，-40（R1）	ADD.D F4，F0，F2	ADD.D F8，F6，F2	nop
L.D F26，-48（R1）	nop	ADD.D F12，F10，F2	ADD.D F16，F14，F2	nop
nop	nop	ADD.D F20，F18，F2	ADD.D F24，F22，F2	nop
S.D 0（R1），F4	S.D -8（R1），F8	ADD.D F28，F26，F2	nop	nop
S.D -16（R1），F12	S.D -24（R1），F16	nop	nop	nop
S.D -32（R1），F20	S.D -40（R1），F24	nop	nop	DADDUI R1，R1，#56
S.D 8（R1），F28	nop	nop	nop	BNE R1，R2，Loop

3. VLIW性能分析

- VLIW目标代码编码效率低
  - 为消除流水线“空转”需要增加循环展开的次数
  - 很难从应用程序中找到足够多的并行指令填满VLIW指令中的每一个slot
- VLIW流水线的互锁机制
  - VLIW处理器中没有实现任何相关检测逻辑，而是靠互锁机制保证执行结果的正确
  - 这种简单的互锁机制将造成较大的开销
- 目标代码兼容性差
  - 二进制翻译

4. 性能比较——多流出处理器 vs. 向量处理器

- 即使对于一些结构不规则的代码，多流出处理器也能从中挖掘出一些指令级并行。
- 多流出处理器对存储系统没有过高的要求，价格较便宜、由Cache和主存构成的多层次存储子系统即可满足其对性能的要求。

结论：多流出处理器已成为当前实现指令级并行的主要选择，而向量处理器则通常是作为协处理器集成到计算机系统中，以加速特定类型的应用程序。

6.4 显示并行指令计算EPIC

- 超标量和VLIW结构都存在严重不足
  - 超标量硬件复杂度太高，8流出成为极限；
  - VLIW存在代码兼容问题，编译器智能程度不够。
- EPIC技术在VLIW基础上融合了超标量的一些优点
  - 编译器通过踪迹调度、超块调度等带有极强猜测性的优化技术尽可能多地挖掘指令级并行。
  - 流水线硬件则提供丰富的计算资源实现这些指令级并行，并通过专门的机制确保在程序执行过程中出现预测错误时仍然能得到正确的运行结果，尽量减少由此引起的额外开销。

- 什么是EPIC?
  - 指令级并行主要由编译器负责开发，处理器为保证代码正确执行提供必要的硬件支持，只有在这些硬件机制的辅助下这些优化技术才能高效完成。
  - 系统结构必须提供某种通信机制，使得流水线硬件能够了解编译器“安排”好的指令执行顺序。
- EPIC编译器的高级优化技术
  - 非绑定分支
  - 谓词执行
  - 前瞻执行

6.4.1 非绑定分支

1. 分支指令在传统流水线上的执行过程

- 计算分支转移条件
- 生成分支目标地址
- 取下一条指令
- 译码并流出下一条指令

在传统流水线上，分支指令都具有“原子性”，即上述各操作被绑定在一起，不能分开。

2. 非绑定分支技术

- 核心思想：将分支指令划分为多条粒度更小的指令，独立执行。
  - 准备操作：计算分支目标地址
  - 比较操作：计算分支转移条件
  - 转移操作：根据分支转移条件是true还是false，改变控制流或执行顺序的下一条指令。
- 运行时，流水线硬件根据前两个操作的结果，动态地将第三个操作转换为空操作或无条件转移。
  - 前两个操作应尽早完成

6.4.2 谓词执行

1. 条件执行机制

- 条件执行：指指令的执行依赖于一定的条件，当条件为真时指令将正常执行，否则将什么也不做。
  - 实例：条件传输指令

例6.4 在下面的语句中，

if（A=0）{S=T；}

假设变量A、S、T的值分别保存在寄存器R1、R2和R3内。请用分支指令和条件传输指令编写功能相同的汇编代码。

解：包含分支指令的MIPS汇编代码如下：

```
        BNEZ    R1，L
        ADDU    R2，R3，R0
    L：
```

而使用条件传输指令CMOVZ时的汇编代码为：

```
        CMOVZ  R2，R3，R1
```

指令CMOVZ有3个操作数，R2为目的操作数，R1和R3是源操作数，执行条件保存在寄存器R1中。

当R1=0时，R3的值被复制到R2中，否则R2的内容不变。

- 分析
  - 条件传输指令将分支指令引起的控制相关转换为相对于分支转移条件（R1）的数据相关。
  - 条件执行机制能够删除代码中那些行为难以预测的分支指令，提高分支预测准确率，并减少由于分支预测错误带来的性能损失。
  - 无论指令的执行条件是否为真，指令都将被读出、译码并执行。
  - 这种编译优化技术叫做条件转换。

2. 条件传输指令的应用——计算绝对值

- 求绝对值的运算A = abs(B)，对应的C语句为：

  if（B<0）{A=-B;} else {A=B;}

- 使用条件传输指令后，代码段如下（假设变量A和B分别被保存在寄存器R1和R2中）：

```
SUB  R1，R0，R2          // A = -B
SLT  R3，R2，R0          // 若B<0，R3=1，
                         // 否则R3=0
CMOVZ  R1，R2，R3        // R3=0时，A = B
```

3. 谓词执行机制

- 条件传输指令的性能问题
  - 随着指令数的增加，经过条件转换得到的条件传输指令和条件计算指令的数量也将增加，这会大大降低目标代码的效率。
- 谓词执行（predicated execution）
  - 给指令集中的每条指令都增加一个执行条件，这个执行条件就叫做谓词（predicate）。
  - 若谓词为真，指令正常执行，否则什么也不做。

4. 谓词执行机制——实例分析

例6.5 假设在一个周期内，某双流出的超标量处理器可以同时执行一个访存操作和一个ALU操作，或者仅执行一个分支操作。受此限制，下面这段汇编代码的执行效率并不高，表现在：

（1）第二个周期只能流出一条ALU指令，访存单元空闲；

（2）当分支转移不成功时，BEQZ指令后的两条LW指令之间存在的数据相关将引起流水线暂停。

试通过谓词执行机制解决这两个问题，减少此段代码的执行开销。

例6.5的代码段

周期	指令1	指令2
1	LW R1，40（R2）	ADD R3，R4，R5
2		ADD R6，R3，R7
3	BEQZ R10，L	
4	LW R8，0（R10）	
5	LW R9，0（R8）	

解 我们用LWC表示带谓词的LW指令，并假设该指令的执行条件为谓词不等于0。这样，BEQZ后的第一条LW指令就可以被转换为LWC指令，并被调度到第二个周期执行，如下所示：

周期	指令1	指令2
1	LW R1，40（R2）	ADD R3，R4，R5
2	LWC R8，20（R10），R10	ADD R6，R3，R7
3	BEQZ R10，L	
4	LW R9，0（R8）	

分析

- 调度后代码的执行时间缩短了。
- 若分支转移成功，**LWC**将被转换为空操作，这不影响结果的正确性，但也不会缩短执行时间。
- 周期**4**中的**LW**指令转换为**LWC**，结果如何？

周期	指令1	指令2
1	LW R1，40（R2）	ADD R3，R4，R5
2	LWC R8，20（R10），R10	ADD R6，R3，R7
3	BEQZ R10，L	
4	LW R9，0（R8）	

5. 谓词执行机制——异常处理

- 谓词执行增加了异常处理的复杂度
- 例若LWC指令执行时发生缺页中断，如何处理？
  - LWC的谓词为true，中断本应发生；
  - LWC的谓词为false，中断不应发生。

周期	指令1	指令2
1	LW R1，40（R2）	ADD R3，R4，R5
2	LWC R8，20（R10），R10	ADD R6，R3，R7
3	BEQZ R10，L	
4	LW R9，0（R8）	

6. 如何将谓词为假的指令转换为空操作？
   - 两种方法
     - 流水线前端指令流出时
     - 流水线后端结果确认时
   - 一般采用第二种方法
     - 为什么？

6.4.3 前瞻执行

1. 概述

- 谓词执行与全局指令调度的不足之处
  - 仅有少量结构全面实现谓词执行
  - 全局指令调度往往需要补偿代码
- EPIC通过前瞻执行提高猜测执行的效果
- 什么是前瞻执行？
  - 在数据相关或控制相关尚未消除时，将指令调度到相关指令前猜测执行。
  - 通过硬件机制完成异常处理，确保正确性。

- 影响前瞻执行效果的因素
  - 编译器能力的高低：能否准确识别可以被前瞻执行的指令。
  - 异常处理机制：能否推迟处理由被前瞻执行的指令引起的异常，直到确定前瞻指令确实被执行后。
  - 如何避免前瞻引起的错误？

2. 实例分析

例6.6 下面是一个if-then-else结构的C程序段以及相应的MIPS汇编代码段，其中变量A和B分别被保存在地址为0(R3)和0(R2)的存储单元中。若分支转移不成功的概率很大，请利用前瞻执行技术将第二条LD指令调度到分支指令BNEZ前执行。假设寄存器R14空闲。

C语句：

```
if （A≠0）A=A+4; else  A=B;
```

汇编指令：

```
    LD  R1, 0（R3）              // 取A
    BNEZ       R1, L1           //（A≠0）?
    LD  R1, 0（R2）              // A=B（else部分）
    J   L2
L1: DADDI R1, R1, #4             // A=A+4(then部分)
L2: SD    R1, 0（R3）            // 存A
```

解 调度结果如下：

```
        LD      R1, 0(R3)          // 取A
        sLD     R14, 0(R2)         // 取B，前瞻执行
        BEQZ    R1, L3
        DADDI R14, R1, #4          // A=A+4
L3:     SD      R14    , 0(R3)     // A=B
```

3. 前瞻执行——异常处理机制

问题：若执行sLD时发生异常应该如何处理？

- 有四种方法
- 终止性异常 vs 可继续异常

➢ 方法一： 立即处理

- 此前瞻指令引起的异常只是简单地返回一个未定义值即可，而不是立即结束程序的运行。
- 前瞻正确时，正在执行的程序不会被终止，但它的执行结果肯定是错误的。
- 前瞻错误时，程序也将继续执行下去，只是处理该异常的返回值不会被使用。

- 方法二：借助专门的检测指令判断是否需要处理
  - 代码实例

```
LD        R1，0（R3）          // 取A
sLD R14，0（R2）               // 取B，前瞻执行
BEQZ      R1，L1
SPECCK    0（R2）              // sLD是否产生异常
J         L2
L1：DADDI     R14，R1，#4      // A=A+4
L2：SD R14，0（R3）            // A=B
```

  - SPECCK的执行条件与sLD指令相同（分支转移失败）
  - 基本思想：推迟处理由前瞻指令sLD引发的异常，直到已确定该指令确实应该被执行 。

- 方法三：借助寄存器状态位判断是否需要处理
  - 为每个通用寄存器增加一个特殊的状态标志位：“poison”位
  - 前瞻指令引发的可继续异常都将被立即处理。
  - 前瞻指令引发终止性异常时，其目的寄存器R的poison位将被置1，否则该位将被清0。
  - 当前瞻指令之后的另一条指令访问R时，若R的poison位为1将触发一个终止性异常。
  - 基本思想：将前瞻指令引起异常的处理推迟到另一条指令访问前瞻指令的目的寄存器时。

- 方法四：借助再定序缓冲器完成
  - 将指令的执行结果保存在再定序缓冲器内，并按指令流出的顺序依次确认。
  - 前瞻指令的确认时机被推迟，直至能够确定该指令的前瞻执行是正确（或错误）的。
  - 除了需要再定序缓冲器等硬件机制的支持外，也需要在编译时标出所有被前瞻的指令，以及这些指令所跨越的条件分支。

4. 控制前瞻

- 将load调度到store之前前瞻执行最常见的数据前瞻
  - 为保证正确性，编译器总是会选择保守的调度方法，认为相邻的store与load间存在地址冲突，但在很多情况下，地址并不冲突。
  - 尽早完成load指令有助于缩短关键路径的长度。
- 硬件地址冲突检测机制
  - 当load指令前瞻执行时，流水线硬件会将它访存的地址记录在一个特殊的地址表中。

- 每执行一条store指令，流水线硬件将该指令的访存地址与地址表中的各有效项进行匹配，命中则说明出现地址冲突，前瞻失败。
- 若控制流抵达load指令原来所在的位置时未出现冲突（编译时在此位置放置检测指令），说明前瞻成功，流水线硬件从地址表中删除对应的项。

➢ 数据前瞻失败时的处理

- 仅**load**指令被前瞻执行：由检测指令重新执行**load**指令即可
- 数据相关于**load**的其他指令也被前瞻执行：需要补偿代码

6.5 开发更多的指令级并行

why?

- 全局指令调度技术已经能够很好地处理由分支指令引起的控制相关
  - 容易预测的——前瞻执行
  - 不容易预测的——谓词执行
- 指令间的数据相关对指令级并行开发的限制作用反而越来越大

6.5.1 挖掘更多的循环级并行

1. 循环携带相关

- 循环携带相关是指一个循环的某个叠代中的指令与其他叠代中的指令之间的数据相关。

例6.7 在下面的循环中，

```
for (i=1; i<=100; i=i+1) {
    A[i+1] = A[i] + C[i];          /* S1 */
    B[i+1] = B[i] + A[i+1];        /* S2 */
}
```

假设数组A、B和C中所有元素的存储地址都互不相同，请问语句S1与S2之间存在哪些数据相关？

解 S1和S2之间存在两种不同类型的数据相关：

- 循环携带RAW数据相关：相邻连词叠代的语句S1之间，相邻两次叠代中的语句S2之间。
- RAW数据相关：同一叠代内的语句S2与S1之间。

分析:

- 循环携带相关迫使指令只能按照所在叠代的先后顺序依次执行。
- 限制了同一叠代内存在数据相关的各语句之间的相对顺序。

- 怎样消除循环携带数据相关？

例6.8 在下面的循环中，语句S1和S2之间存在哪些数据相关？该循环的各次叠代是否可以并行执行？如果不能，请修改其代码，使之可以并行。

```
for (i=1; i<=100; i=i+1) {
        A[i] = A[i] + B[i];          /* S1 */
        B[i+1] = C[i] + D[i];        /* S2 */
}
```

解 第i次叠代中语句S1与第i-1次叠代中语句S2之间存在RAW类型的循环携带数据相关，但它们之间没有形成环(S2与上次叠代的S1不相关)。修改后代码

```
A[1] = A[1] + B[1];
for (i=1; i<=99; i=i+1) {
    B[i+1] = C[i] + D[i];          /* 原S2 */
    A[i+1] = A[i+1] + B[i+1];      /* 原S1 */
}
B[101] = C[100] + D[100];
```

修改方法：将存在循环携带相关的各条指令放在同一个叠代中

- 复杂循环携带数据相关的处理

```
for（i=6；i<=100；i=i+1）
    Y[i] = Y[i-5] + Y[i];    // 相关距离为5
for（i=2；i<=100；i=i+1）
    Y[i] = Y[i-1] + Y[i];    // 相关距离为1
```

编译器必须检测出这种递归关系

(1) 某些系统结构（特别是向量计算机）为递归提供了专门的硬件支持

(2) 这样的递归结构中通常隐藏着大量的循环级并行

2. 存储别名分析

- 什么是存储别名
  - 一个元素可能同时拥有多个合法的地址表达式
  - A[i+5]、A[j*2-6]、A[k]
- 数组是仿射的
  - 如果一个一维数组A[m:n]的下标可以被表示为形如a×i+b的形式，那么就称该数组是仿射的（affine）。
  - 一个多维数组，如果它每一维的下标都是仿射的，那么它就是仿射的。

GCD测试法

算法描述

- 如果GCD(c, a)可以整除(d-b)，那么有可能存在存储别名。
- 如果GCD测试的结果为假（不能整除），那么一定没有存储别名存在。

例6.9 使用GCD测试方法判断下面的循环中是否存在存储别名。

```
for（i=1；i<=100；i=i+1）
    x[2*i+3] = x[2*i] * 5.0；
```

解 在这个循环中，a=2，b=3，c=2，d=0，

那么GCD(a, c)=2，而d-b=-3。

由于2不能整除-3，因此没有存储别名，即无论i取何值，x[2*i+3]与x[2*i]都将表示数组x的不同元素。

3. 数据相关分析

- 除了检测指令之间是否存在数据相关外，编译器还会将识别出的数据相关进一步细分为真数据相关、输出相关和反相关等不同类型，以便利用不同的优化技术消除这些相关。
- 常用的优化有重命名、值传播、高度消减等。

➢ 重命名优化实例

例6.10 找出下面循环中的所有数据相关，指出它们究竟是真数据相关、输出相关、还是反相关，并利用重命名技术消除其中的输出相关和反相关。

```
for (i=1; i<=100; i=i+1) {
        Y[i] = X[i] / a;          /* S1 */
        X[i] = X[i] + a;          /* S2 */
        Z[i] = Y[i] + a;          /* S3 */
        Y[i] = a  -  Y[i];        /* S4 */
}
```

解 这4条语句中存在以下相关：

1. S3与S1和S4与S1之间分别存在真数据相关。
2. S2和S1之间存在反相关。
3. S3和S4之间存在反相关。
4. S1和S4之间存在输出相关。

```
for (i=1; i<=100; i=i+1) {
    Y[i] = X[i] / a;          /* S1 */
    X[i] = X[i] + a;          /* S2 */
    Z[i] = Y[i] + a;          /* S3 */
    Y[i] = a - Y[i];          /* S4 */
    }
```

将原代码变换为下面的形式，可以消除所有输出相关和反相关。

```
for (i=1; i<=100; i=i+1) {
        /* 将数组Y重命名为T以消除输出相关 */
        T[i] = X[i] / c;
        /* 将数组X重命名为X1以消除反相关 */
        X1[i] = X[i] + c;
        /* 将Y重命名为T以消除反相关 */
        Z[i] = T[i] + c;
        Y[i] = c  -  T[i];
}
```

- 值传播优化实例
  - 优化前代码

```
DADDUI R1，R2，#4
DADDUI R1，R1，#4
```

  - 优化后代码

```
DADDUI R1，R2，#8
```

  - 值传播优化通过将变量替换为已知的值或表达式以达到消除数据相关

高度消减优化实例

- 目的：缩短数据流图中关键路径的长度
- 优化前代码I

```
ADD     R1，R2，R3      /* I1 */
ADD     R4，R1，R6      /* I2 */
ADD     R8，R4，R7      /* I3 */
```

- 优化后代码I

```
ADD     R1，R2，R3
ADD     R4，R6，R7
ADD     R8，R1，R4
```

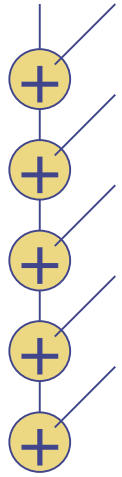
高度消减优化实例

- 优化前代码II

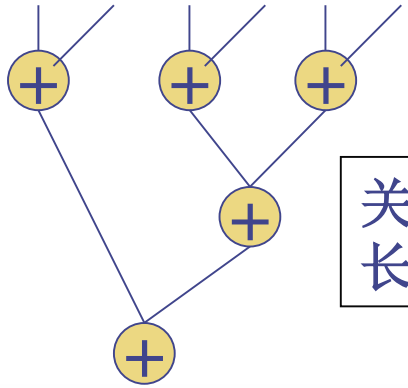
```
sum = sum + X[i];
sum = sum + X[i+1];
sum = sum + X[i+2];
sum = sum + X[i+3];
sum = sum + X[i+4];
```

- 优化后代码II

```
sum = sum + X[i];
t1 = X[i+1] + X[i+2];
t2 = X[i+3] + X[i+4];
t1 = t1 + t2;
sum = sum + t1;
```



关键路径
长度为5

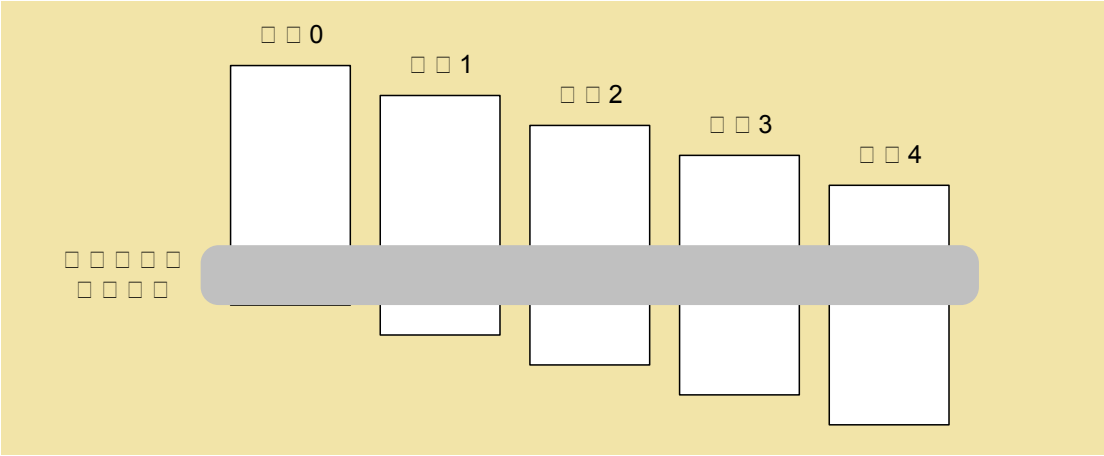


关键路径
长度为3

6.5.2 软流水

1. 简介

- 软流水技术的核心思想是从循环不同的叠代中抽取一部分指令（循环控制指令除外）拼成一个新的循环叠代。
- 目的
  - 将同一叠代中的相关指令分布到不同的叠代中
  - 将不同叠代中的相关指令封装到同一叠代中



2. 实例分析

例6.11 试用软流水技术处理例6.1中的循环，假设数组x有n个元素。

解 软流水需要从原循环的多个叠代中选择指令拼成新的循环，因此我们首先将原循环展开。

```
叠代i:      ; 修改x[i]并保存
            L.D     F0, 0(R1)
            ADD.D   F4, F0, F2
            S.D     F4, 0(R1)
叠代i+1:    ; 修改x[i-1]并保存
            L.D     F0, 0(R1)
            ADD.D   F4, F0, F2
            S.D     F4, 0(R1)
叠代i+2:    ; 修改x[i-2]并保存
            L.D     F0, 0(R1)
            ADD.D   F4, F0, F2
            S.D     F4, 0(R1)
```

从这3个叠代中分别选出一条指令，与原有的循环控制指令拼在一起得到一个新的叠代。

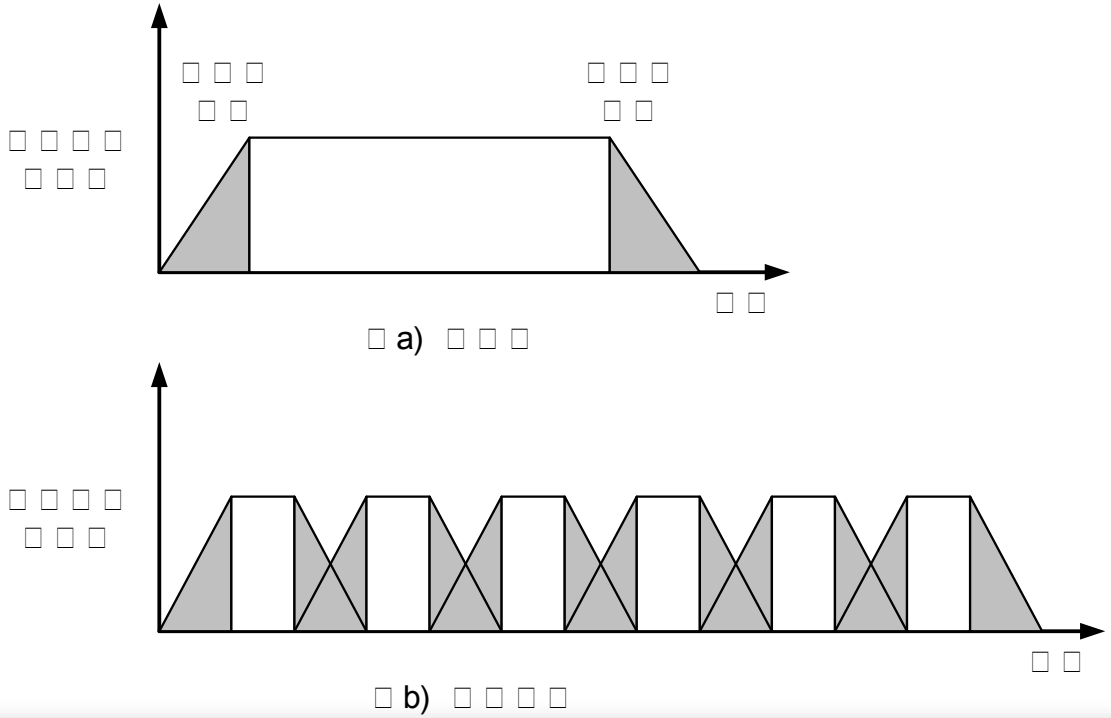
```
        DADDUI      R1, R1, #-16   // I1：R1保存x[n-2]的地址
        L.D   F0, 16 (R1)          // I2：取x[n]
        ADD.D F4, F0, F2           // I3：x[n] = x[n] + F2
        L.D   F0, 8 (R1)           // I4：取x[n-1]
Loop:   S.D   F4, 16 (R1)          // I5：存x[i+2]
        ADD.D F4, F0, F2           // I6：x[i+1] = x[i+1] +
        F2
        L.D   F0, 0 (R1)           // I7：取x[i]
        BNE   R1, R2, Loop         // I8
        DADDUI      R1, R1, #-8    // I9：填充分支延迟槽
        S.D   F0, 8 (R1)           // I10：存x[2]
        ADD.D F4, F0, F2           // I11：x[1] = x[1] + F2
        S.D   F4, 0 (R1)           // I12：存x[1]
```

分析：

- 新循环的结束条件仍为：R1=R2
- 新循环从元素x[n-2]开始从头处理的，元素x[n]与x[n-1]只是从半途开始处理。
- 新循环的最后一次叠代只处理完元素x[3]，x[2]刚被修改完结果尚未写回，x[1] 刚被取出，需要被修改并写回。
- 新循环的每个叠代相当于流水执行了原循环的3个叠代。元素x[n/n-1/2/1]的处理相当于充满和排空这条“流水线”。

3. 性能比较：软流水 vs. 循环展开

- 循环展开主要减少由分支指令和修改循环索引变量的指令所引起的循环控制开销。
- 软流水使叠代内的指令级并行达到最大。



6.6 实例：IA-64体系结构

1. IA-64简介

- IA-64是Intel与HP合作研制出的64位EPIC体系结构，其设计遵循以下原则：
  - 按照EPIC的思想开发指令级并行；
  - 提供大量的硬件资源实现指令级并行；
  - 提供一系列辅助软件指令级并行开发技术的硬件机制。

2. Intel IA-64产品——Itanium

- 主频：800MHz
- 采用3级Cache
  - 第一级为分离的指令和数据Cache
  - 后两级为混合Cache
  - 第三级Cache（在片外）
- 流水线共分为10级，其中实现了一些基于硬件的动态指令调度机制，如分支预测、重命名、记分牌等。
- 提供了丰富的功能单元，所有的功能单元都是流水的。
- 每个周期最多流出6条指令，最多可同时执行3条分支指令和2个访存操作，提高了分支处理和存储访问的能力。

- 整数性能(SPECint 2000)
  - 主频为800MHz的Itanium性能仅为2GHz主频Pentium 4的60%，1GHz主频Alpha 21264的68%。
  - Alpha 21264主频的功耗比Itanium低20%。
- 浮点性能(SPECfp 2000)
  - 800MHz Itanium的性能是2GHz Pentium 4的1.04倍，是1GHz Alpha 21264的1.20倍。

3. Intel IA-64产品——Itanium 2
   - 其性能在不经过任何调试和优化的条件下比Itanium提高50%到100%。

6.6.1 IA-64指令格式

IA-64功能单元和操作的类型

功能单元类型	操作类型	描述	操作实例
I-unit	A	整数ALU运算	加，减，与，或，比较
	I	非ALU整数运算	整数和多媒体移位、位测试、移动
M-unit	A	整数ALU运算	加，减，与，或，比较
	M	访存操作	整数/浮点load, store操作
F-unit	F	浮点操作	各种浮点运算
B-unit	B	分支操作	条件分支、无条件分支、调用
L+X	L+X	扩展操作	空操作、扩展的立即数

- IA-64的目标代码被划分为多个指令组，并按照流出的时间顺序依次排列，编译器指出组边界。
- IA-64的指令被称为“bundle”，长128位，包含123位操作信息（3个操作×41位/操作=123位）和5位模板信息。
- 模板位的作用
  - 简化译码，模板信息指明了bundle中3个操作的类型以及它们分别需要在哪类功能单元上执行。
  - 显式指出指令组边界，模板中定义了停止位。

IA-64模板值及其含义

模板值	操作槽0	操作槽1	操作槽2
0	M	I	I
1	M	I	I（s）
2	M	I（s）	I
3	M	I（s）	I（s）
4	M	L	X
5	M	L	X（s）
8	M	M	I
9	M	M	I（s）
10	M（s）	M	I
11	M（s）	M	I（s）
12	M	F	I
13	M	F	I（s）

模板值	操作槽0	操作槽1	操作槽2
14	M	M	F
15	M	M	F（s）
16	M	I	B
17	M	I	B（s）
18	M	B	B
19	M	B	B（s）
22	B	B	B
23	B	B	B（s）
24	M	M	B
25	M	M	B（s）
28	M	F	B
29	M	F	B（s）

例6.12 将例6.1中的代码展开7次并生成IA-64指令，使得：

(1) bundle数最少；

(2) 执行时间最短。

假设每时钟周期可流出1个bundle，bundle中任何操作的暂停将导致整个流水线暂停。各操作延迟如表6.1所示。

解 调度结果中“s”表示停止位，nop表示空操作。

- 第一种情况下只有9个bundle，15%的槽为空，但需要21个周期才能完成。
- 第二种情况代码体积和空槽均有所增加，需要11个bundle，有30%的槽是空槽，但执行时间大大缩短，仅用12个周期就可以完成 。

解 （1）bundle最少

模板	操作槽0	操作槽1	操作槽2	执行时间
9：M M I	L.D F0，0（R1）	L.D F6，-8（R1）	nop (s)	1
14：M M F	L.D F10，-16（R1）	L.D F14，-24（R1）	ADD.D F4，F0，F2	3
15：M M F	L.D F18，-32（R1）	L.D F22，-40（R1）	ADD.D F8，F6，F2 (s)	4
15：M M F	L.D F26，-48（R1）	S.D F4，0（R1）	ADD.D F12，F10，F2 (s)	6
15：M M F	S.D F8，-8（R1）	S.D F12，-16（R1）	ADD.D F16，F14，F2 (s)	9
15：M M F	S.D F16，-24（R1）	nop	ADD.D F20，F18，F2 (s)	12
15：M M F	S.D F20，-32（R1）	nop	ADD.D F24，F22，F2 (s)	15
15：M M F	S.D F24，-40（R1）	nop	ADD.D F28，F26，F2 (s)	18
15：M M F	S.D F28，-48（R1）	DADDUI R1，R1，#-56	BNE R1，R2，Loop	21

(2) 执行时间最短

模板	操作槽0	操作槽1	操作槽2	执行时间
8：M M I	L.D F0，0（R1）	L.D F6，-8（R1）	nop	1
9：M M I	L.D F10，-16（R1）	L.D F14，-24（R1）	nop (s)	2
14：M M F	L.D F18，-32（R1）	L.D F22，-40（R1）	ADD.D F4，F0，F2	3
14：M M F	L.D F26，-48（R1）	nop	ADD.D F8，F6，F2	4
15：M M F	nop	nop	ADD.D F12，F10，F2 (s)	5
14：M M F	nop	S.D F4，0（R1）	ADD.D F16，F14，F2	6
14：M M F	nop	S.D F12，-16（R1）	ADD.D F20，F18，F2	7
15：M M F	nop	S.D F8，-8（R1）	ADD.D F24，F22，F2 (s)	8
14：M M F	nop	S.D F16，-24（R1）	ADD.D F28，F26，F2	9
9：M M I	S.D F20，-32（R1）	S.D F24，-40（R1）	nop (s)	11
8：M M I	S.D F28，-48（R1）	DADDUI R1，R1，#-56	BNE R1，R2，Loop	12

6.6.2 IA-64的谓词执行机制

- IA-64提供的硬件支持
  - 设置了大量的谓词寄存器（64个），每个谓词寄存器的宽度都是1位。
  - 所有类型的IA-64操作都可以按照谓词执行方式执行，每个操作41位编码的最低6位指明了保存执行条件的谓词寄存器。
  - 增强了比较（compare）操作和测试（test）操作的功能，都可以同时修改多个谓词寄存器。

IA-64所支持的不同比较模式

模式	指令后缀	操作	
		目的谓词寄存器1	目的谓词寄存器2
Normal	无	if (qp) {target=result}	if (qp) {target=!result}
Unconditional	unc	if (qp) {target=result} else {target=0}	if (qp) {target=!result} else {target=0}
AND	and	if (qp && !result) {target=0}	if (qp && !result) {target=0}
	andcm	if (qp && result) {target=0}	if (qp && result) {target=0}
OR	or	if (qp && result) {target=1}	if (qp && result) {target=1}
	orcm	if (qp && !result) {target=1}	if (qp && !result) {target=1}
DeMorgan	or.andcm	if (qp && result) {target=1}	if (qp && result) {target=0}
	and.orcm	if (qp && !result) {target=0}	if (qp && !result) {target=1}

例6.13 将下面的C语句转换为IA-64汇编指令，使其执行时间最短。

```
if （r1==1 || r2==2 || r3==3 || r4==4）s;
```

解 对应的IA-64汇编代码如下。

IA-64 cmp指令的格式为：cmp.op.mod p, q = r1, r2

这里op为要进行的比较操作，mod为上表所示的比较模式中的一种，p，q为两个目的谓词寄存器，r1和r2为要比较的操作数，可以是寄存器名，也可以是立即数。

```
cmp.ne p1, p0 = r0, r0 ; ;
cmp.eq.or p1, p0 = 1, r1
cmp.eq.or p1, p0 = 2, r2
cmp.eq.or p1, p0 = 3, r3
cmp.eq.or p1, p0 = 4, r4     ; ;
（p1） s
```

6.6.3 IA-64的前瞻执行机制

- 为了标识被前瞻执行的load指令并检测前瞻是否成功，IA-64专门设置了3条指令。
  - **ld.a**：表示被前瞻执行的load指令
  - **ld.c**：编译器将ld.c指令放在被前瞻执行的load指令最初所在的位置，ld.c执行则前瞻成功
  - **chk.a**：编译器将它放在与ld.c相同的位置。若执行chk.a时发现对应的load前瞻执行失败，它会将控制流转移到一段补偿代码中，重新执行。
- 前瞻load指令的地址被记录在ALAT表中
- 每个寄存器有一个状态位NaT(Not A Thing)