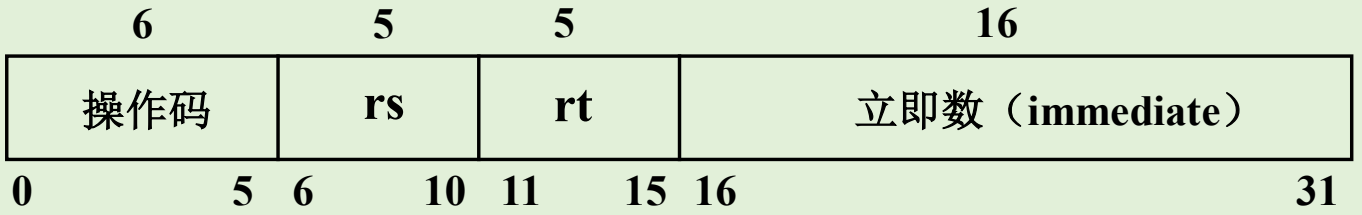
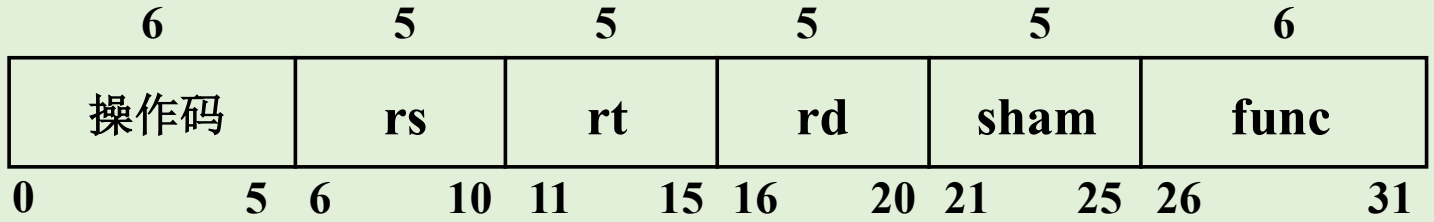
计算机系统结构

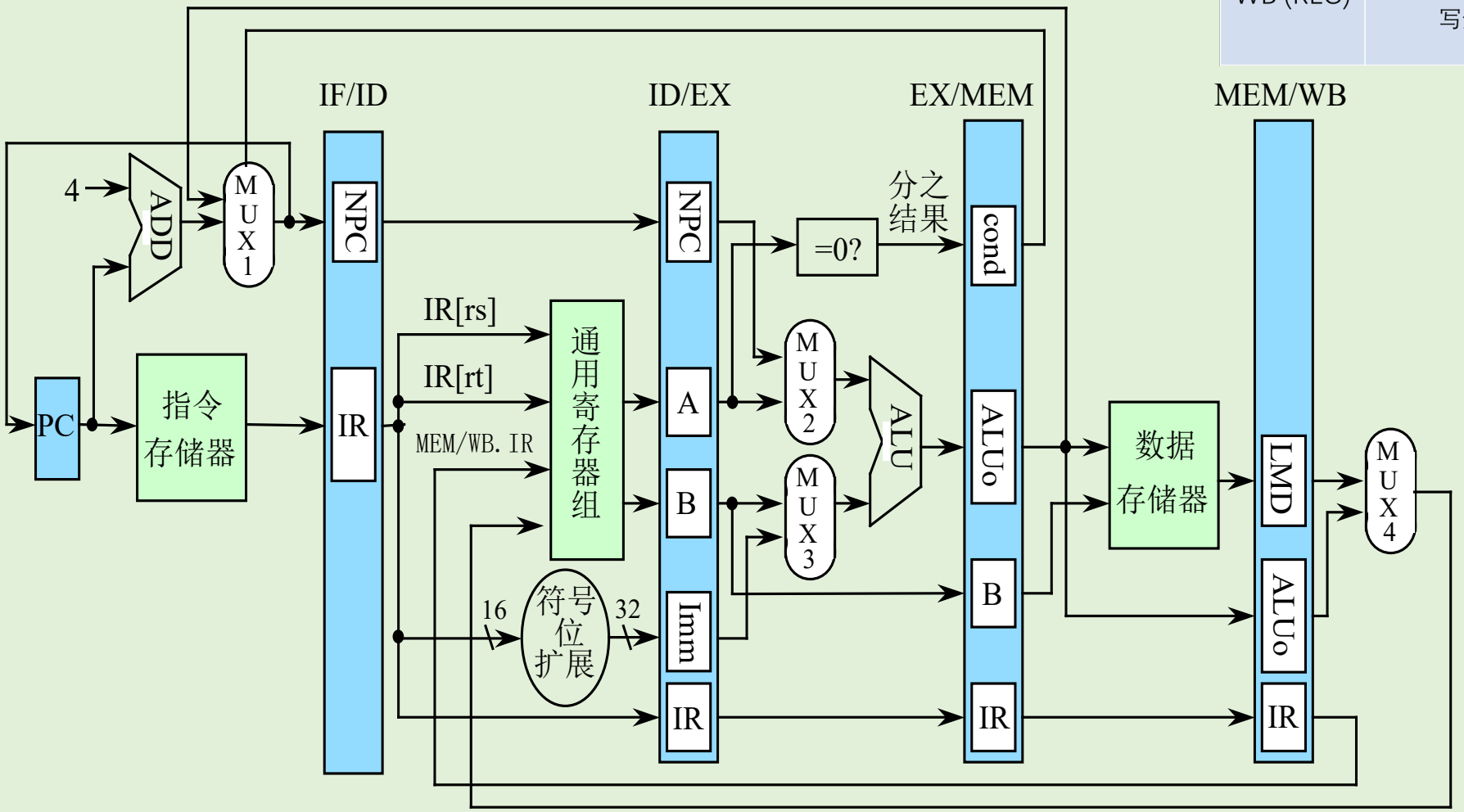
指令级并行

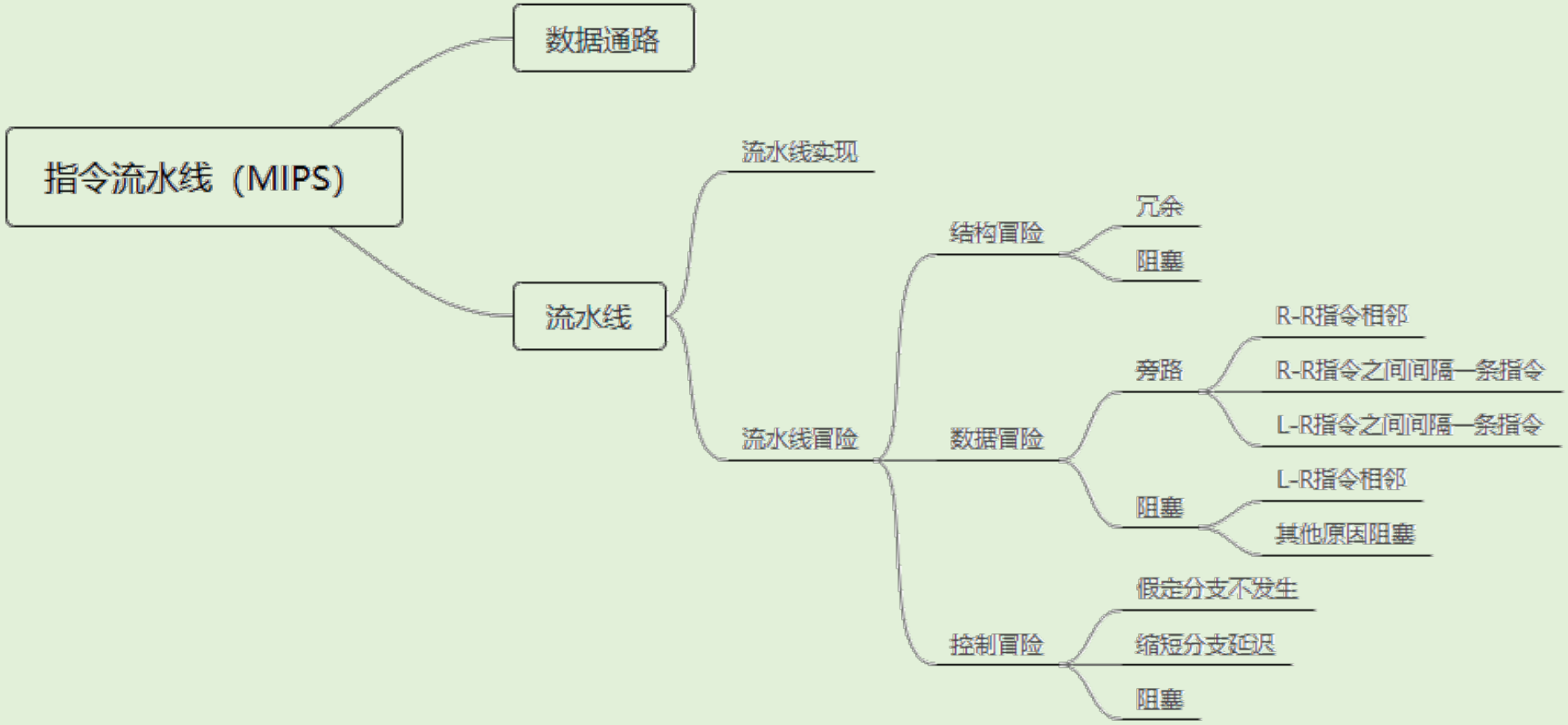
一 指令流水线基础知识复习



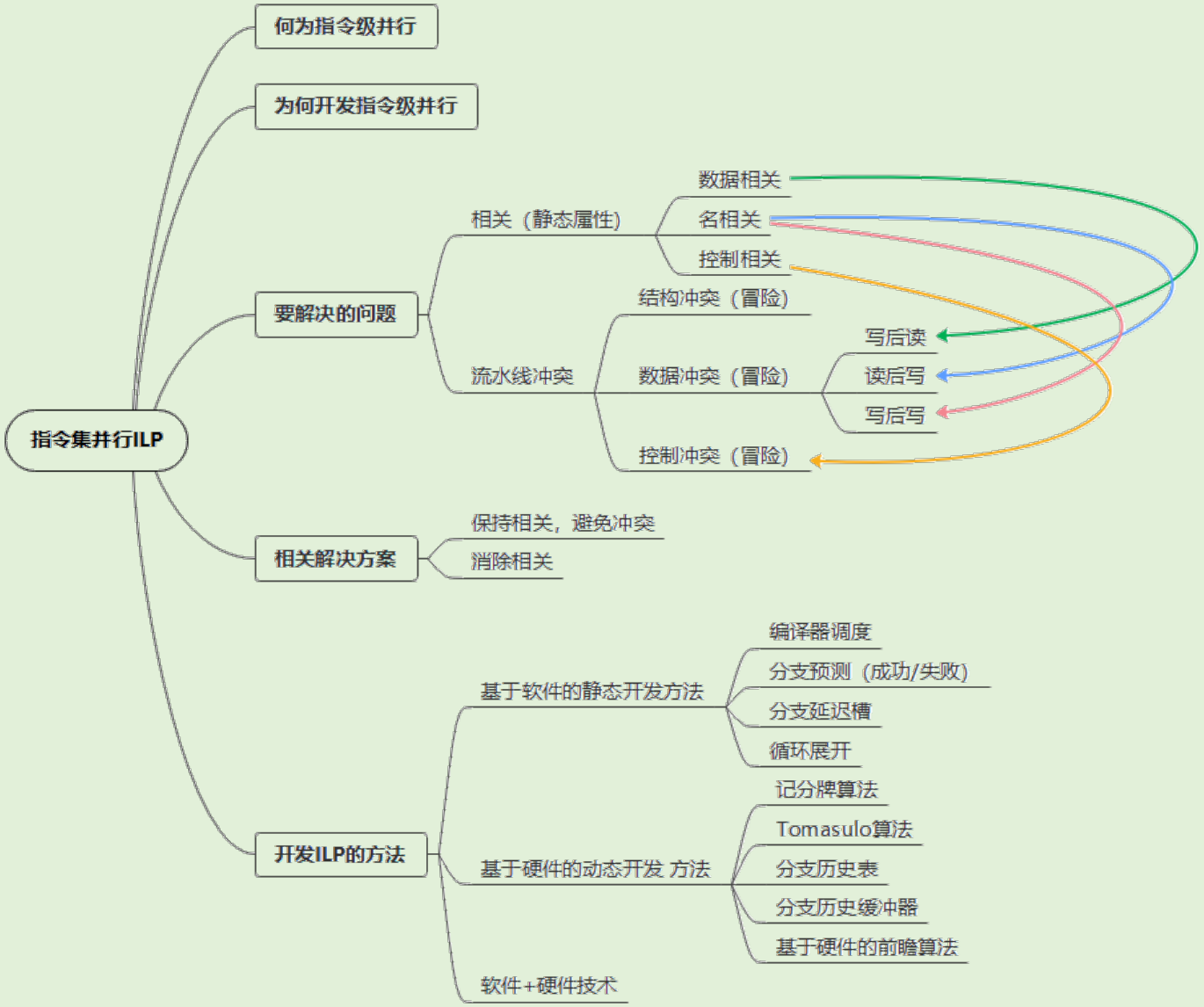


	RR ALU	R-立即数 ALU	访存	分支
IF (IM)	取指令→ IR， PC=PC+4			
ID (REG)	译码，访问通用寄存器，取操作数			
EX (ALU)	ALU	ALU	计算有效地址	计算目标地址；判断分支是否成功
MEM(DM)			访存	目标地址→PC
WB (REG)	ALU运算结果写通用寄存器		Load指令访存结果写通用寄存器	




指令流水线（MIPS）
数据通路
流水线
流水线实现
流水线冒险
结构冒险
冗余
阻塞
数据冒险
旁路
R-R指令相邻
R-R指令之间间隔一条指令
L-R指令之间间隔一条指令
阻塞
L-R指令相邻
其他原因阻塞
控制冒险
假定分支不发生
缩短分支延迟
阻塞

二 指令级并行概念及基本编译技术



2.1 指令级并行的概念

- 指令级并行ILP概念:是指存在于指令一级即指令间的并行性，主要是指机器语言一级，如存储器访问指令、整型指令、浮点指令之间的并行性等。

- 指令级并行主要特点:是并行性由处理器硬件和编译程序自动识别和利用，不需要程序员对顺序程序作任何修改。正是由于这一优点，使得它的发展与处理器的发展紧密相连。

2.2 开发指令级并行的重要性

- 理想CPI（无冲突）是衡量流水线最高性能的一个指标
- 实际CPI往往远大于理想CPI，它是理想流水线的CPI加上各类停顿的时钟周期数：

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

- 减少等式右端的各项就减少了总的CPI,从而提高IPC.（Instructions Per Cycle：每个时钟周期完成的指令条数）

2.3 开发指令级并行需要解决的具体问题

- 相关
  - 两条指令之间存在某种依赖关系
  - 分类：数据相关、名相关、控制相关

```
L.D     F0, 0(R1)
ADD.D   F4, F0, F2
```

反相关：指令*j*写的名＝指令*i*读的名

```
DADDU   R6, R5, R2
DADDU   R5, R1, R2
```

输出相关：指令*j*写的名＝指令*i*写的名

```
DADDU   R5, R3, R2
。。。
DADDU   R5, R1, R2
```

```
BEQZ    R1, name
ADD.D   F7, F3, F4
Name:   ADD.D   F4, F0, F2
        DAADIU R1, R1, #-8
```

- 流水线冲突
  - 对于具体的流水线来说，由于相关等原因的存在，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

- 小结与比较
  - 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。

2.4 相关的两类解决方案

- 保持相关，但避免发生冲突
  - 指令调度 -- 编译器静态调度
- 通过代码变换，消除相关
  - 寄存器重命名 -- 寄存器换名消除WAR和WAW

2.5 开发指令级并行的方法

- 基于软件的静态开发方法
- 基于硬件的动态开发方法
- 硬件＋软件技术

只有硬件技术和软件技术互相配合，才能够最大限度地挖掘出程序中存在的指令级并行。

2.6 基本编译技术 - 循环展开 与 指令调度

- 充分开发指令之间存在的并行性，找出不相关的指令序列，让它们在流水线上重叠并行执行。
- 增加指令间并行性最简单和最常用的方法
  - 开发循环级并行性——循环的不同迭代之间存在的并行性。
  - 在把循环展开后，通过重命名和指令调度来开发更多的并行性。
- 编译器完成这种指令调度的能力受限于两个特性：
  - 程序固有的指令级并行性；
  - 流水线功能部件的执行延迟。

本节中，我们使用的浮点流水线延迟为：

产生结果的指令	使用结果的指令	延迟（时钟周期数）
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

假设采用MIPS的5段整数流水线：

- 分支的延迟：1个时钟周期。
- 整数load指令的延迟：1个时钟周期。
- 整数运算部件是全流水或者重复设置了足够的份数。

例　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=1;  i<=1000;  i++)
        x[i] = x[i] + s;
```

把该程序翻译成MIPS汇编语言代码：假设R1的初值是指向第一个元素，8（R2）指向最后一个元素。

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

其中：

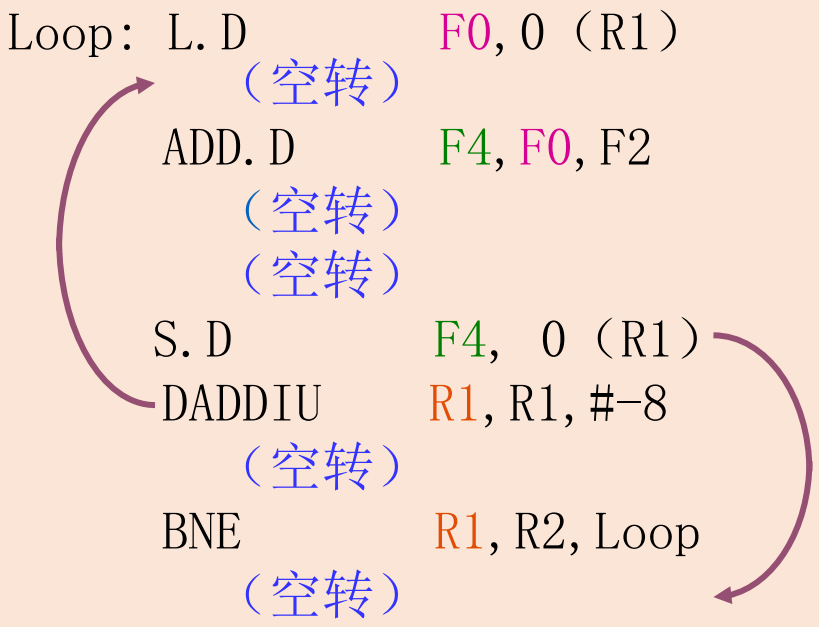
- 整数寄存器R1：指向向量中的当前元素。
  （初值为向量中最高端元素的地址）
- 浮点寄存器F2：用于保存常数s。

情况 I ： 不进行调度

			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

每个元素的操作需要10个时钟周期，其中5个是空转周期。

情况 II ： 进行指令调度



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

一个元素的操作时间从10个时钟周期减少到6个，其中5个周期是有指令执行的，1个为空转周期。

- 例子中的问题及解决方案
  - 只有L.D、ADD.D和S.D这3条指令是有效操作。
  - 占用3个时钟周期。
  - 而DADDIU、空转和BEN这3个时钟周期都是附加的循环控制开销。
- 循环展开技术
  - 把循环体的代码复制多次并按顺序排列，然后相应调整循环的结束条件。
  - 这给编译器进行指令调度带来了更大的空间。

情况 III ： 将情况 I 每四个循环展开成一个循环

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

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

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

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

- 去掉多余的循环控制语句
- 寄存器换名
- 修改下标

			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

结果分析：

- 这个循环每遍共使用了28个时钟周期。
- 有4个循环体，完成4个元素的操作。

  平均每个元素使用28/4=7个时钟周期
- 原始循环的每个元素需要10个时钟周期。

  节省的时间：从减少循环控制的开销中获得的。
- 在整个展开后的循环中，实际指令只有14条，其他14个周期都是空转。

  效率并不高

情况 IV ： 将情况 III 进行指令调度

			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

结果分析：

- 没有数据相关引起的空转等待。整个循环仅仅使用了14个时钟周期。平均每个元素的操作使用14/4=3.5个时钟周期。
- 通过循环展开、寄存器重命名和指令调度，可以有效地开发出指令级并行。

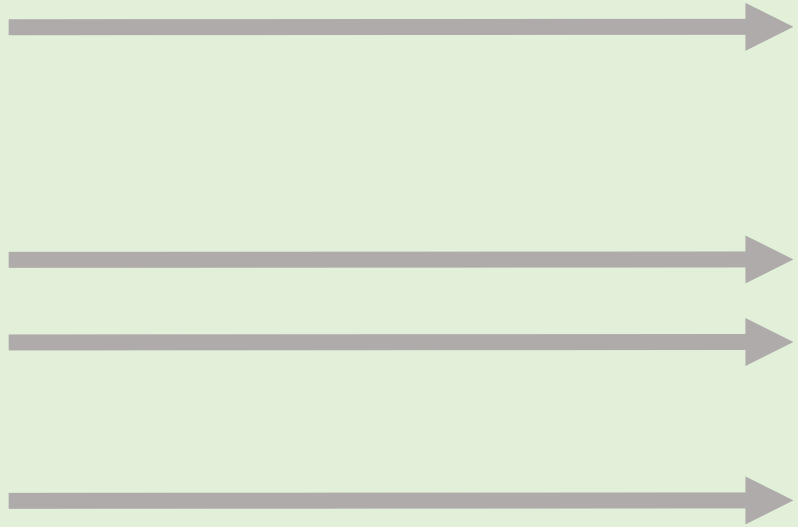
- 循环展开和指令调度时要注意以下几个方面：
  - 保证正确性。在循环展开和调度过程中尤其要注意两个地方的正确性：循环控制，操作数偏移量的修改。
  - 注意有效性。只有能够找到不同循环体之间的无关性，才能有效地使用循环展开。
  - 使用不同的寄存器。（否则可能导致新的冲突）
  - 删除多余的测试指令和分支指令，并对循环结束代码和新的循环体代码进行相应的修正。
  - 注意对存储器数据的相关性分析例如：对于load指令和store指令，如果它们在不同的循环迭代中访问的存储器地址是不同的，它们就是相互独立的，可以相互对调。
  - 注意新的相关性。由于原循环不同次的迭代在展开后都到了同一次循环体中，因此可能带来新的相关性。

三 动态调度解决数据冒险

3.1 基本思想

1. 相关与流水线冲突

- 相关：两条指令之间存在的某种依赖关系
- 相关有3种类型
  - 数据相关
  - 名相关
    - 反相关
    - 输出相关
  - 控制相关



- 控制冲突（冒险）
  - 停顿
  - 重复设置资源
- 数据冲突（冒险）之写后读
  - 定向/旁路 （forwarding）
  - 编译器指令调度
- 数据冲突（冒险）之读后写
- 数据冲突（冒险）之写后写
- 控制冲突（冒险）
  - 预测分支失败
  - 预测分支成功
  - 延迟分支

静态调度

2. 几个基本概念

- 静态调度 V.S. 动态调度
  - 静态调度：是依靠编译器对代码进行调度，也就是在代码被执行之前进行调度；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。

- 指令顺序执行 V.S. 指令乱序执行
  - 指令顺序执行：指令放入流水线的顺序和指令完成的顺序一致；
  - 指令乱序执行：指令放入流水线的顺序和指令完成的顺序不一致，也就是说有些指令进入流水线后被阻塞的，而在其后进入流水线的指令先完成了。

3. 为什么要进行动态调度？ -- 第一个问题

首先，思考下面一段代码：

指令1　DIV.D　F4，F0，F2

指令2　SUB.D　F10，F4，F6

指令3　ADD.D　F12，F6，F14

这段代码放到 MIPS 的五段流水线上，假如没有用任何定向等技术，会怎样执行？

大概会是这样的：

	1	2	3	4	5	6	7	8	9	10	11	12
DIV.D F4, F0, F2	IF	ID	EX	MEM	WB							
SUB.D F10, F4, F6		IF	S	S	S	ID	EX	MEM	WB			
ADD.D F12, F6, F14						IF	ID	EX	MEM	WB		

指令2和指令1是关于F4数据相关，因此进入流水线之后会产生数据冲突，导致指令2被阻塞，指令3也同时被阻塞了。指令3是很冤枉的，明明和前面指令什么关系都没有。

思考：怎么办？

解决第一个问题

指令1　DIV.D　F4，F0，F2

指令2　SUB.D　F10，F4，F6

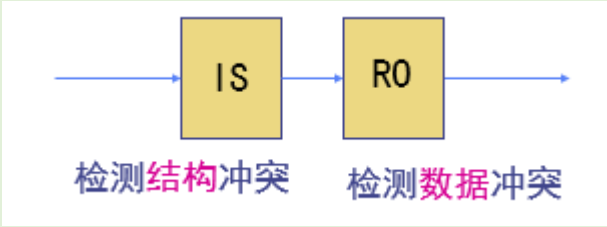
指令3　ADD.D　F12，F6，F14

指令2被阻塞了，停下来等待指令1。能否让无辜的指令3先执行呢？

指令的动态调度，也就是在指令的执行过程中进行调度。

将ID段又分成了两个阶段：

- 流出（Issue，IS）阶段：指令译码，检查是否存在结构冲突。
- 读操作数（Read Operands，RO）阶段：检测数据冲突，如果没有，继续执行；如果有，等待数据冲突消失，然后读操作数。



4. 为什么要进行动态调度？ -- 第二个问题

假如代码是这样的，第1条指令是个除法指令，执行阶段可能会比较长（假设执行需要3个周期时间）；第2和第3条有反相关。

指令1　DIV.D　F4，F0，F2

指令2　SUB.D　F10，F4，F6

指令3　ADD.D　F6，F12，F14

那么把这段代码放入无动态调度的流水线中，恐怕是这样的：

	1	2	3	4	5	6	7	8	9	10	11	12	
DIV.D F4, F0, F2	IF	ID	EX	EX	EX	MEM	WB						
SUB.D F10, F4, F6		IF	S	S	S	S	S	ID	EX	MEM	WB		
ADD.D F6, F12, F14								IF	ID	EX	MEM	WB	

除了时间长点，没啥错误。

但是如果进行指令动态调度呢？恐怕是这样的：

	1	2	3	4	5	6	7	8	9	10	11	12
DIV.D··F4，F0，F2	IF	ID	EX	EX	EX	MEM	WB					
SUB.D··F10，F4，F6		IF	S	S	S	S	S	ID	EX	MEM	WB	
ADD.D··F6，F12，F14			IF	ID	EX	MEM	WB					

请思考：有没有问题？

指令3先写入F6， 指令2后读出F6，就产生了错误的执行结果。

也就是说，指令的动态调度导致了指令的乱序执行，指令的乱序执行导致了有反相关和输出相关的指令进入流水线之后产生读后写冲突和写后写冲突。

解决第二个问题

采用寄存器换名技术，消除名相关。

于是下面这段代码：

```
指令1    DIV.D  F4，F0，F2
指令2    SUB.D  F10，F4，F6
指令3    ADD.D  F6，F12，F14
```

可以变成：

```
指令1    DIV.D  F4，F0，F2
指令2    SUB.D  F10，F4，S
指令3    ADD.D  F6，F12，F14
```

4. 动态调度基本思想小结

- 相对于静态指令调度 ，动态指令调度是在指令的执行过程中进行调度，使得无关的指令得以先执行，减少阻塞；
- 能够处理一些在编译时情况不明的相关（如存储器访问的相关）；
- 能够使本来是面向某一流水线优化编译的代码在其他的流水线（动态调度）上也能高效地执行；
- 动态指令调度将会引起指令乱序执行，因此，使用换名技术消除名相关（包括反相关和输出相关）；
- 指令乱序完成使得异常处理困难；
- 以硬件复杂性的显著增加为代价。

3.3 Tomasulo算法

知识要点:

1. 核心思想
2. 基本结构
3. Tomasulo算法的执行步骤
4. 一个简单的例子
5. Tomasulo算法特点
6. Tomasulo详细解读
7. Tomasulo算法举例
8. Tomasulo要点总结

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。

- 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

为了消除反相关和输出相关请进行换名

- 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

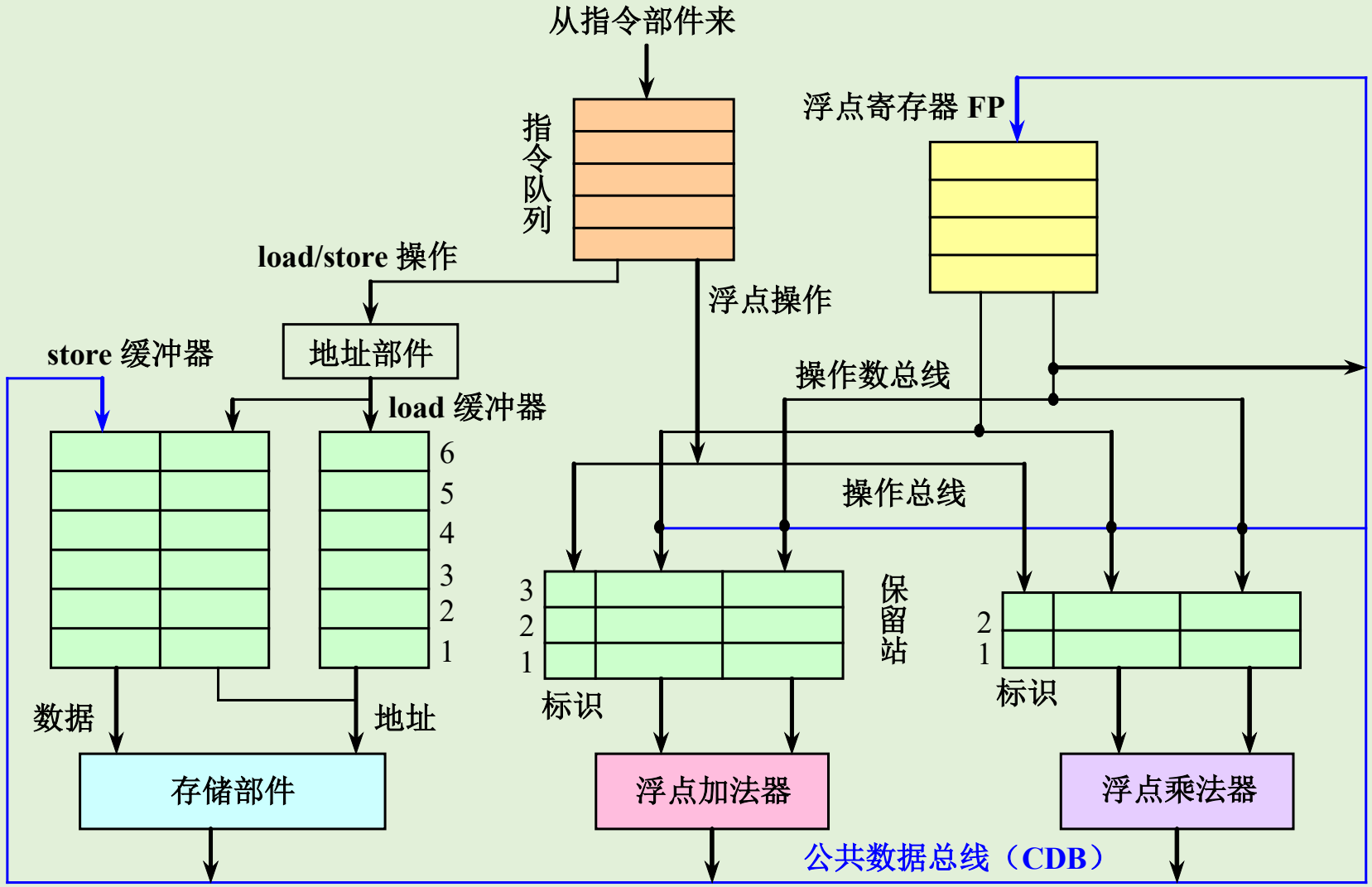
S消除ADD与MUL的WAW冲突

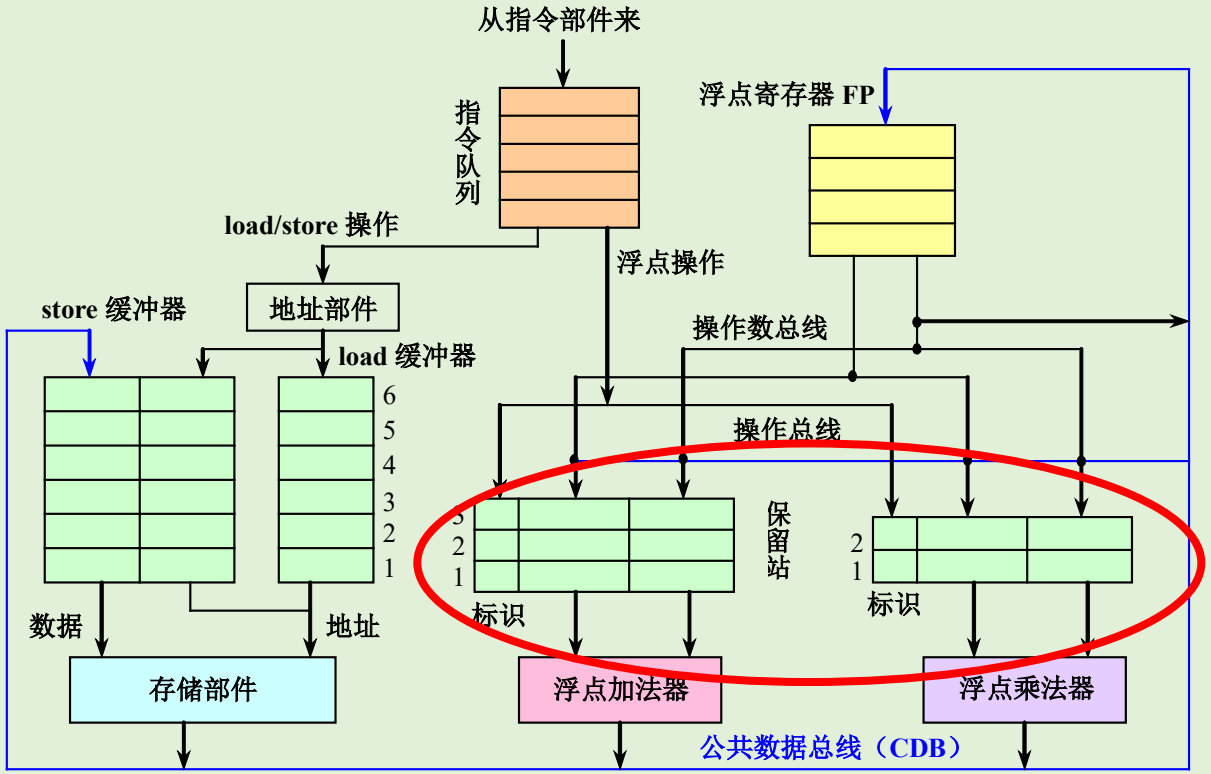
T消除ADD与SUB的WAR冲突

2. Tomasulo的基本结构

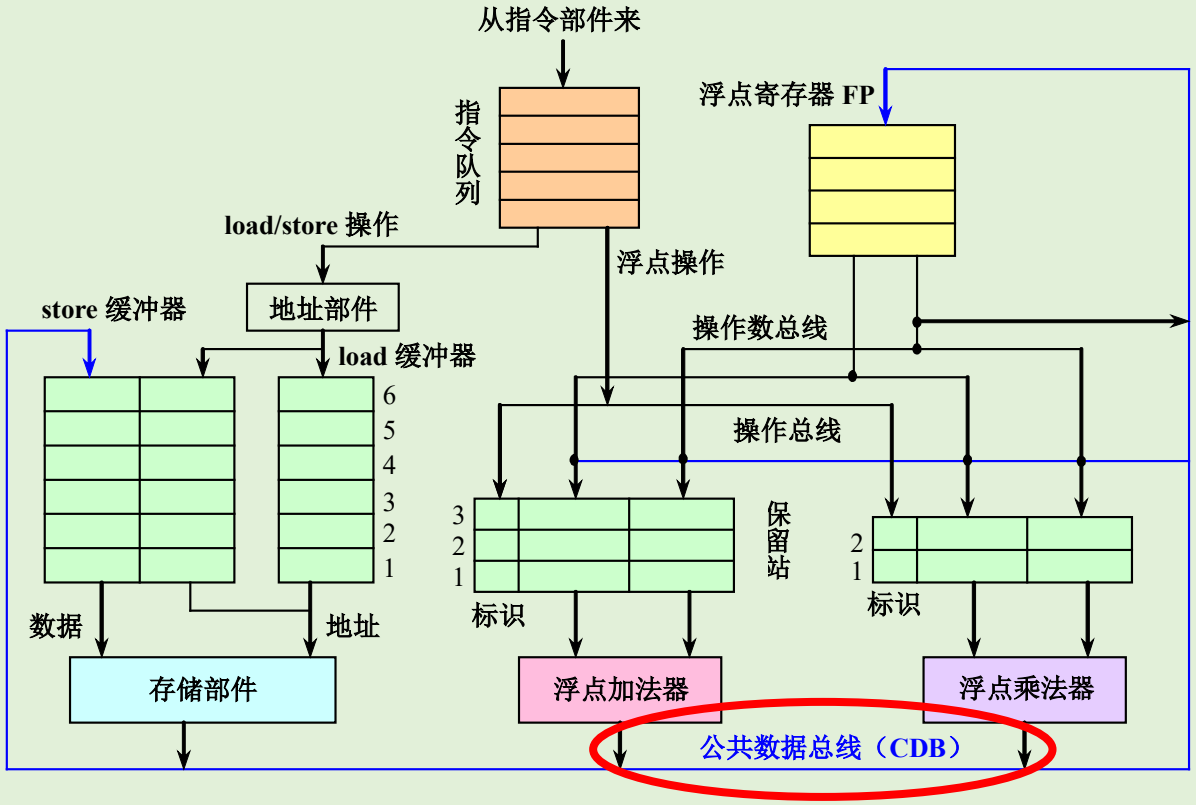
主要部件：

- 指令队列
- 保留站
- Store/load缓冲器
- 公共数据总线CDB
- 运算部件
- 存储部件
- 浮点寄存器

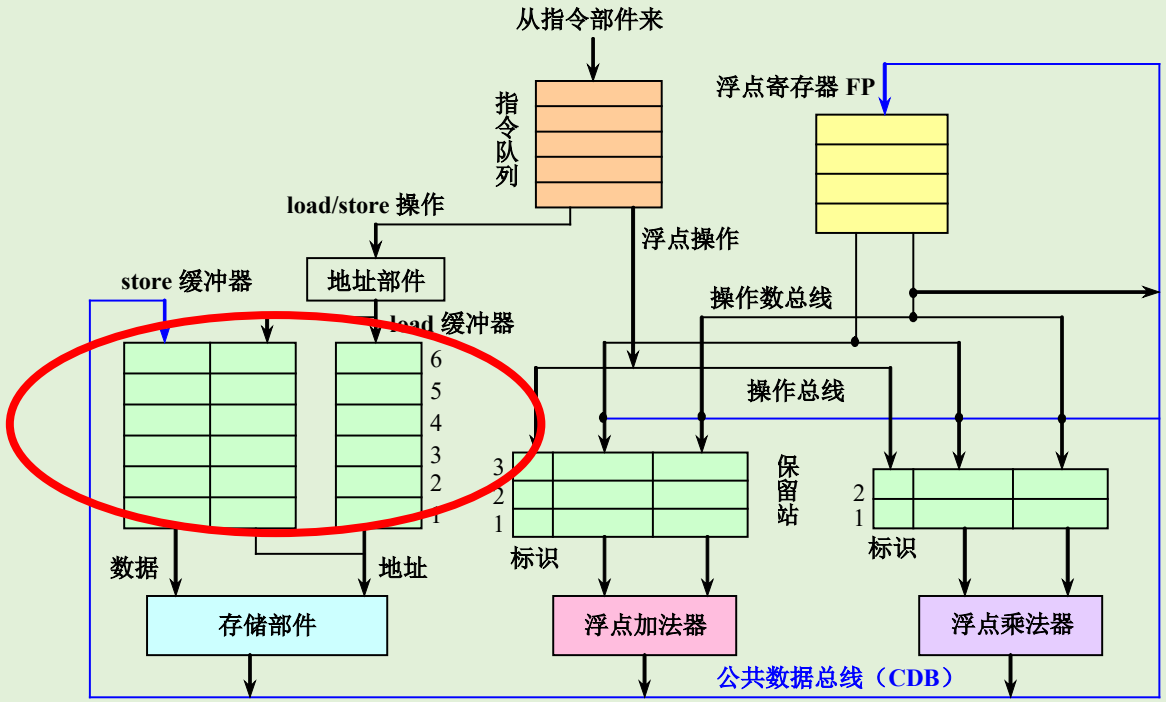




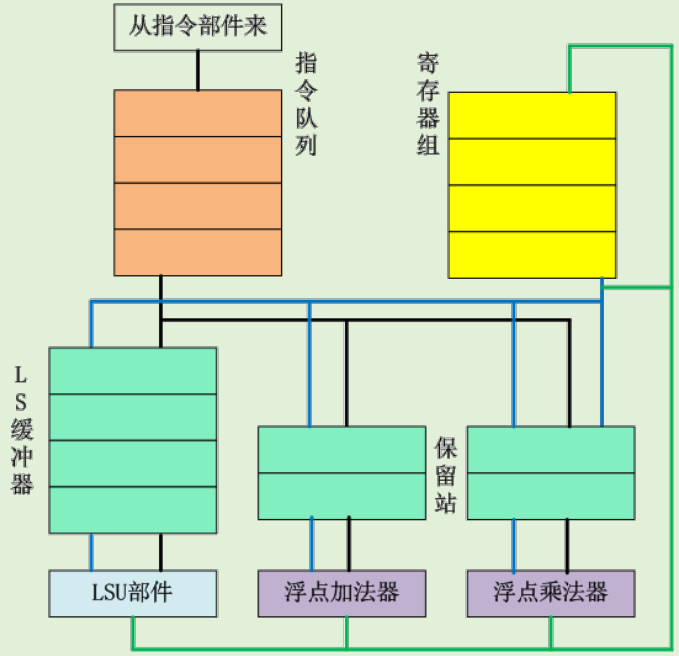
- 保留站（reservation station）
  - 每个保留站中保存一条已经流出并等待到本功能部件执行的指令（相关信息）。
  - 每个保留站包括操作码、操作数以及用于检测和解决冲突的信息。
    - 在一条指令流出送到保留站的时候，如果该指令的源操作数已经在寄存器中就绪，则将操作数取到该保留站中。
    - 如果操作数还没有计算出来，则在该保留站中记录即将产生这个操作数的保留站的标识。
  - 浮点加法器有3个保留站：ADD1，ADD2，ADD3
  - 浮点乘法器有2个保留站：MULT1，MULT2
  - 每个保留站都有一个标识字段，唯一地标识了该保留站。



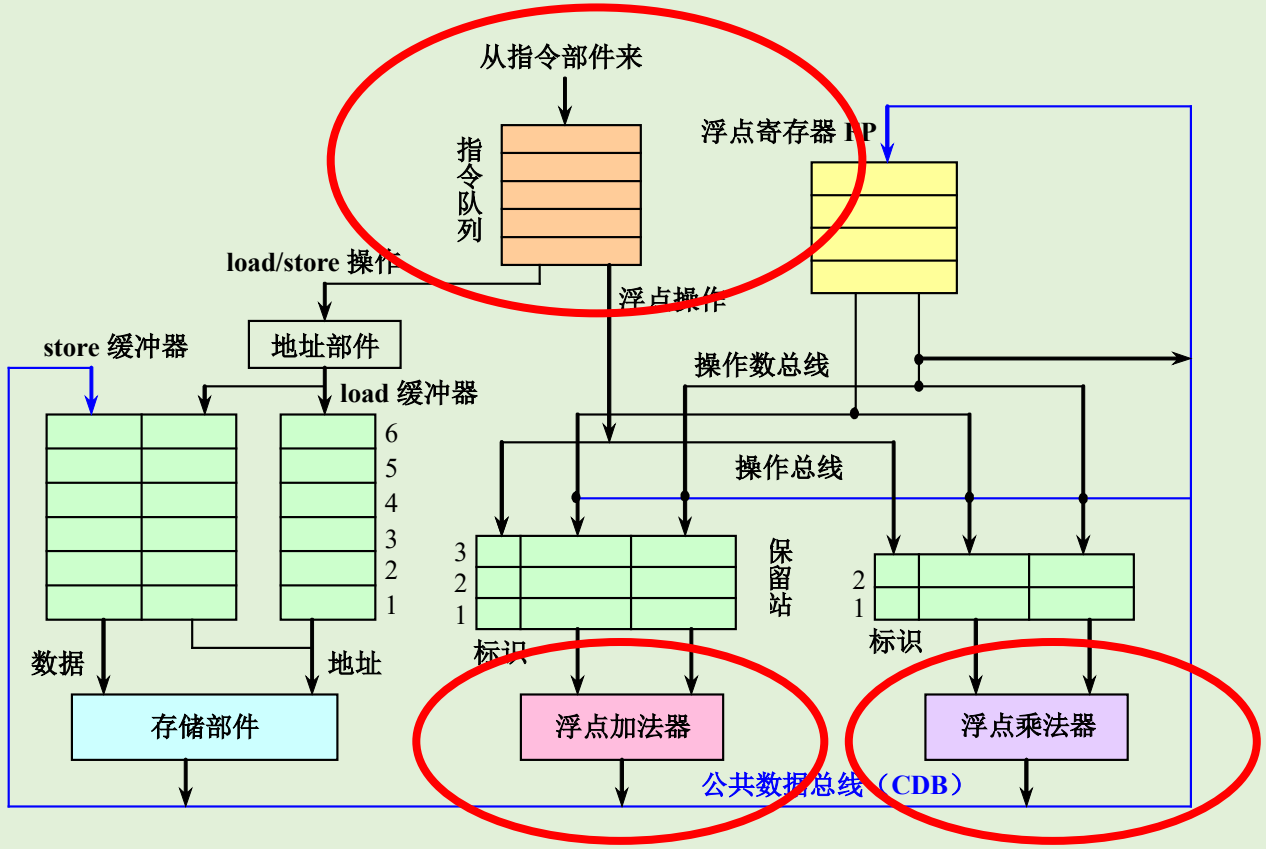
- 公共数据总线CDB （一条重要的数据通路）
  - 所有功能部件的计算结果都是送到CDB上，由它把这些结果直接送到（播送到）各个需要该结果的地方。
  - 在具有多个执行部件且采用多流出（即每个时钟周期流出多条指令）的流水线中，需要采用多条CDB。
  - 从存储器读取的数据也送到CDB。
  - CDB连接到除了load缓冲器以外的所有部件的人口。
  - 浮点寄存器通过一对总线连接到功能部件，并通过CDB连接到store缓冲器的人口



- 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

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。这样 后面指令对寄存器的写入操作就不可能产生WAR冲突了

3.Tomasulo算法的执行步骤

要点三步：

- 流出
- 执行
- 写结果

4. 一个简单的例子说明 Tomasulo算法的基本思想。

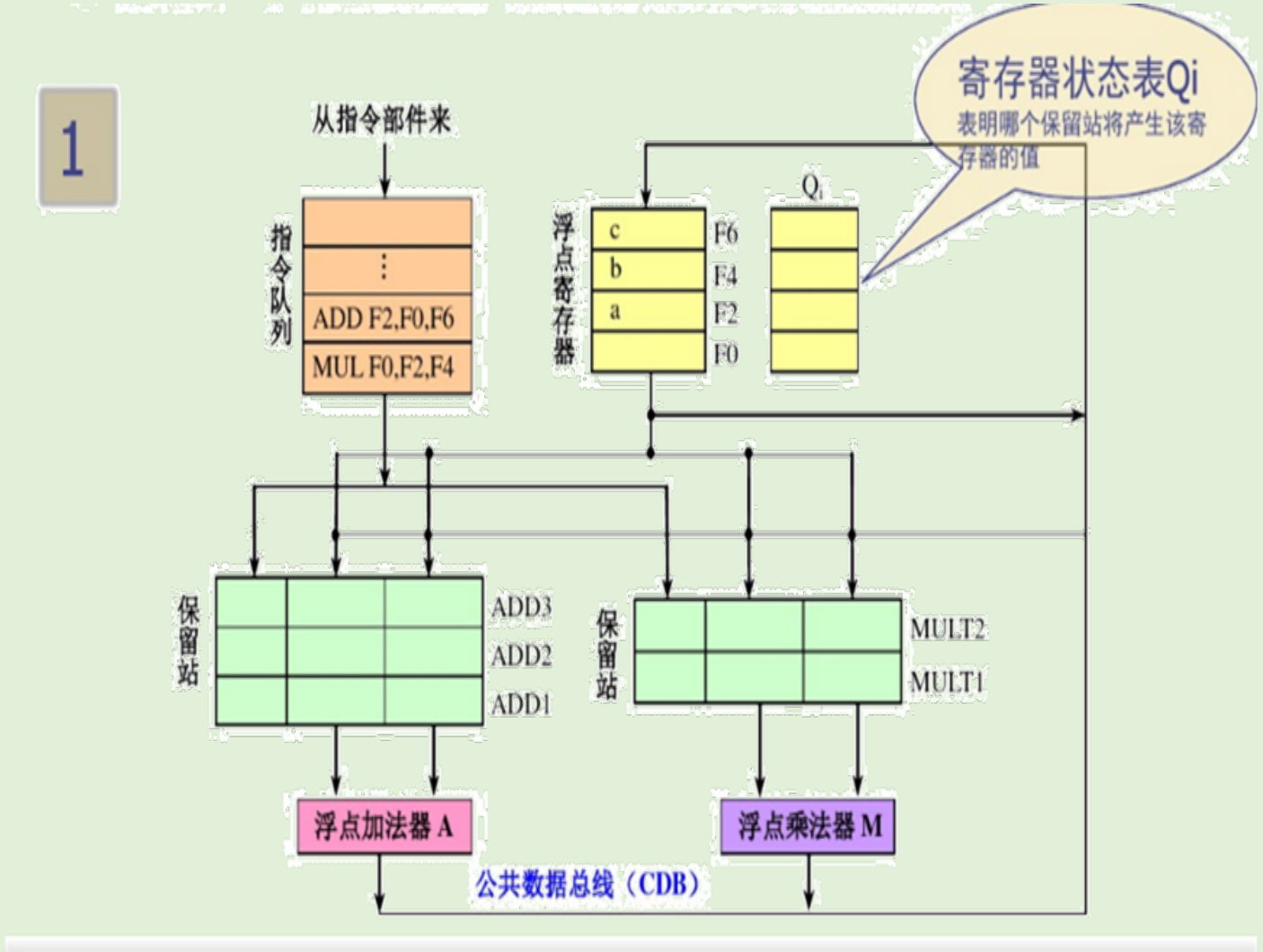


图3.2(a)

MUL F0,F2,F4
ADD F2,F0,F6

把图3.1简化为图3.2(a)，并且增加了一个寄存器状态表Q_i。每个寄存器在Q_i中有一项，用于指出哪个保留站将产生该寄存器的值。

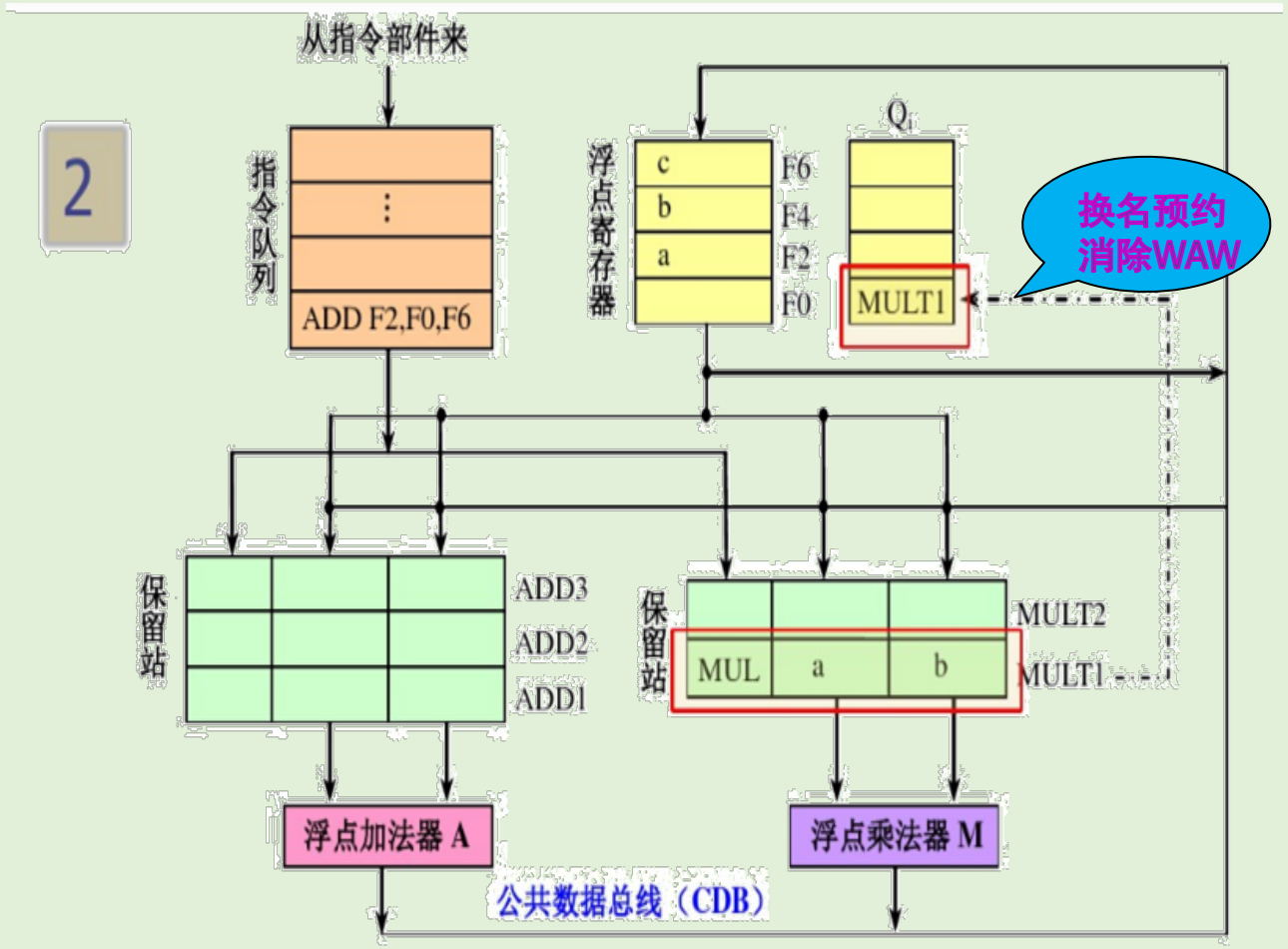


图3.2(b)

MUL F0,F2,F4
ADD F2,F0,F6

图3.2(b)，当指令MULF0，F2，F4流出到保留站 Mult1时，由于其操作数a和b就绪(在F2和F4中)，就将它们从寄存器取到保留站，这样该指令以后就跟F2和F4没有关系了，执行时直接从保留站中取数据。同时将目的寄存器F0对应的Q_i标志置为Mut1，表示该寄存器的内容将由保留站 Mult1提供(图3.2(b)中的虚线所示)。

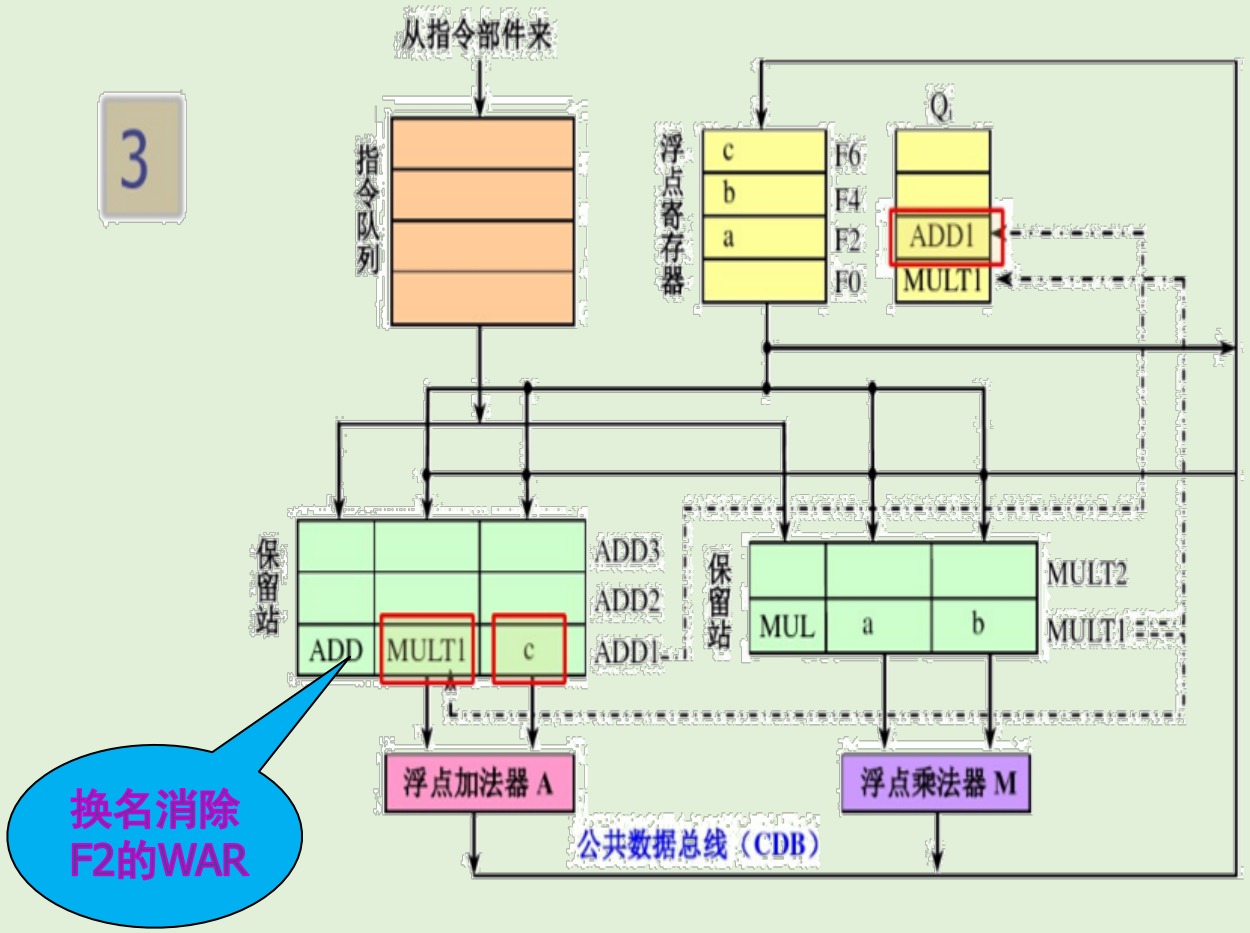


图3.2(c)

MUL F0,F2,F4
ADD F2,F0,F6

图3.2(c)，当指令ADDF2,F0,F6流出到保留站Add1时，也将操作数c取到保留站，但发现F0中的操作数还没有就绪，于是就把其提供者 Mult1的标识取到保留站中。这样就有两个地方在等 Mult1的结果。同时，它将目的寄存器F2对应的Q_i标志置为Add1表示该寄存器的内容将由保留站Add1提供。

4

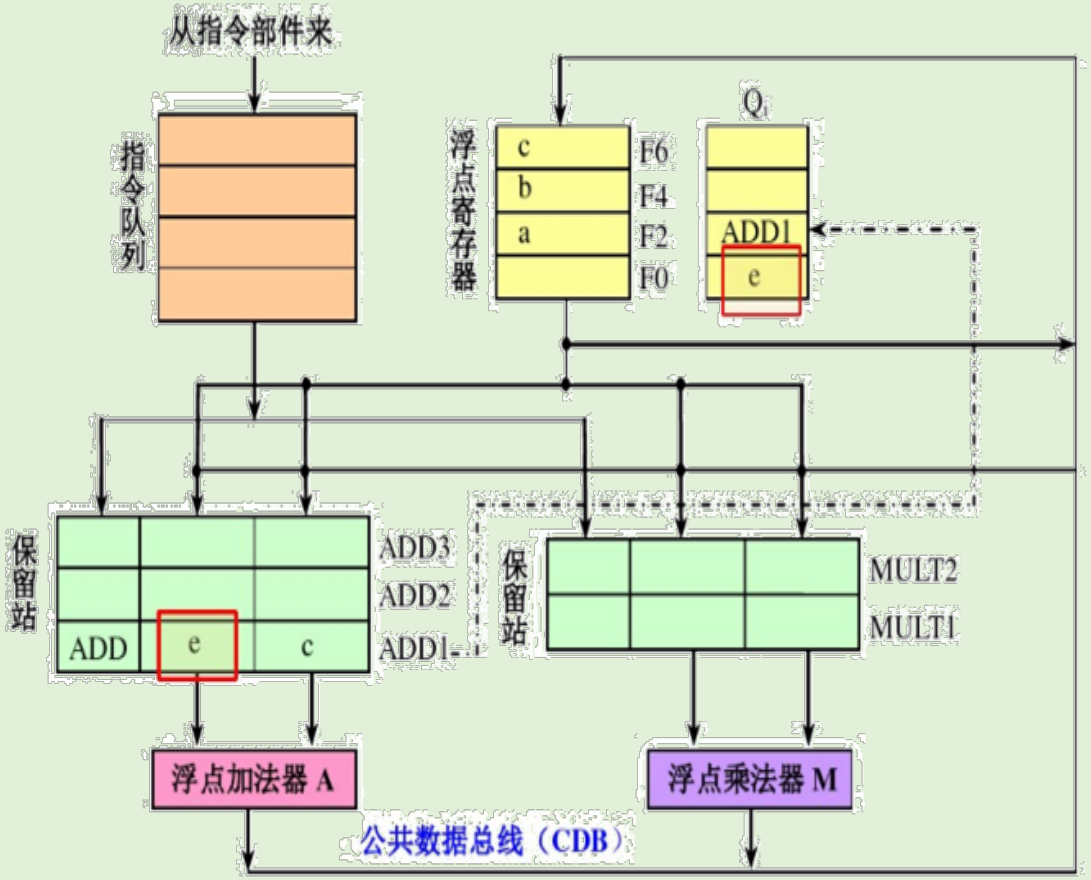


图3.2(d)

MUL F0,F2,F4
ADD F2,F0,F6

图3.2(d)，当 Mult1的运算结果产生后(设为e)，就把数据放到总线上(广播)，所有等待该数据的地方都会自动把数据取走。Add1中的ADD指令得到该数据后，马上就可以开始执行。

5. Tomasulo算法具有两个主要的优点：

①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。

②消除了WAW冲突和WAR冲突导致的停顿(详情见后面例题4.1)

使用保留站进行寄存器换名，并且操作数一旦就绪就将之放入保留站。

6. 详细算法解读

① 首先，搞清楚所有的字段是什么含义；

② 其次，搞清楚 ALU指令、 Load指令 和 Store指令进入每个阶段的条件，以及每个阶段结束时，保留站和浮点寄存器状态表的状态；

③ 没有捷径，耐着性子往下看，注意细节。

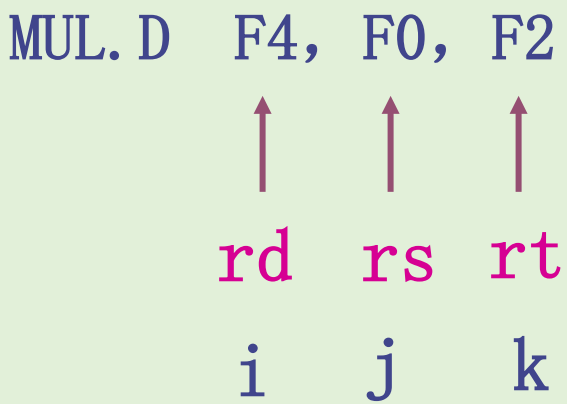
Tomasulo具体算法详细解读：

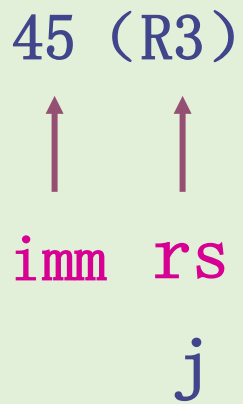
各符号的意义

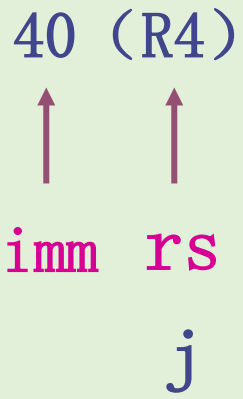
- r：分配给当前指令的保留站或者缓冲器单元编号；
- rs、rt、rd：参考MIPS指令系统字段含义；
- imm：符号扩展后的立即数；
- RS：保留站；
- result：浮点部件或load缓冲器返回的结果；
- Qi：寄存器状态表；
- Regs[ ]：寄存器组；
- Op:当前指令的操作码

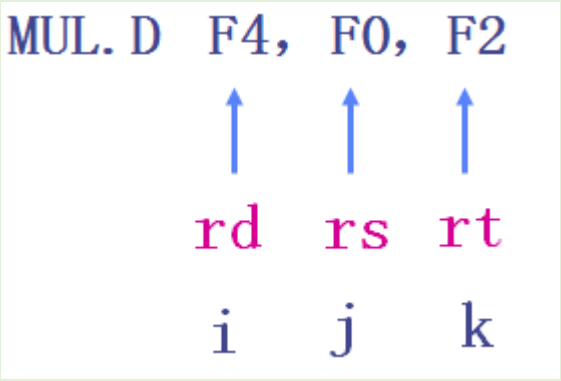
- 与rs对应的保留站字段：Vj，Qj
- 与rt对应的保留站字段：Vk，Qk
- Load指令：rt是保存所取数据的寄存器号
- Store指令：rt是保存所要存储的数据的寄存器号
- Qi、Qj、Qk的内容或者为0，或者是一个大于0的整数。
  - Qi为0表示相应寄存器中的数据就绪。
  - Qj、Qk为0表示保留站或缓冲器单元中的Vj或Vk字段中的数据就绪。
  - 当它们为正整数时，表示相应的寄存器、保留站或缓冲器单元正在等待结果。

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



- **指令流出**

①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].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。

else // 第一操作数没有就绪

{ RS[r].Qj ← Qi[rs] } // 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。

```
if  (Qi[rt] = 0)                    // 检测第二操作数是否就绪
  { 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。
else                                //第二操作数没有就绪
  { RS[r].Qk ← Qi[rt] }             //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。
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

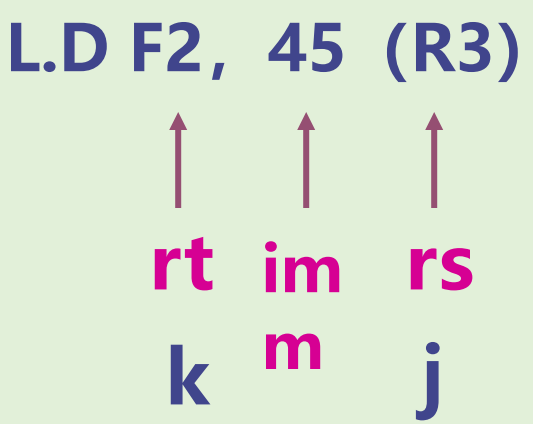
k　m　j

② 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].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。
 else                               // 第一操作数没有就绪
  {RS[r].Qj ← Qi[rs] }              // 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。
```



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 im rs

k m j

对于store指令：

if （Qi[rt] = 0） // 检测要存储的数据是否就绪

{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。

else // 该数据没有就绪

{RS[r].Qk ← Qi[rt] } // 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。

2. 执行

① 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 。

② 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

此处理解为，Load和Store指令需要在同一队列中，且按照FIFO的次序访存

注意：Load指令在该阶段完成的任务是两个：计算访存地址，访存

注意：Store指令在该阶段，只需要第一个操作数就绪。

请思考：为什么Load和Store指令要按顺序访存?

3. 写结果

① 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。
```

② store指令

进入条件：保留站r执行结束，且RS[r].Qk = 0

注意：Store指令在该阶段，需要第二个操作数就绪。

```
                                // 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。
```

7. Tomasulo算法总结

	流出	执行	访存	写结果
浮点ALU指令	进入条件：保留站有空闲； 完成任务：进入保留站，换名，目标寄存器预约；	进入条件：两个操作数就绪，且部件不冲突； 完成任务：浮点运算		进入条件：功能部件执行结束，CDB就绪； 完成任务：将结果通过CDB写入浮点寄存器和保留站，修改Qi，Qj，Qk状态，释放保留站；
Load指令	进入条件：L/S缓冲区有空闲； 完成任务：进入缓冲区，立即数写入A字段；	进入条件：RS寄存器就绪，计算部件不冲突； 完成任务：计算访存地址，并写入A字段；	进入条件：L/S队列头部，地址已经计算完，写入A字段； 完成任务：访存，结果放到CDB上；	进入条件：访存结束，CDB就绪； 完成任务：将结果通过CDB写入浮点寄存器和保留站，修改Qi，Qj，Qk状态，释放缓冲区；
Store指令	进入条件：L/S缓冲区有空闲； 完成任务：进入缓冲区，立即数写入A字段；	进入条件：RS寄存器就绪，计算部件不冲突； 完成任务：计算访存地址，并写入A字段；	进入条件：L/S队列头部，地址已经计算完，写入A字段，要写入的数据就绪； 完成任务：写入存储器；	这个访存就是它的写结果阶段

需要注意的地方：

- 用硬件的思维理解硬件的算法；
- 一个clock，也就是一个时钟周期，打开的是一个存储设备到另一个存储设备之间的数据通路；
- 对于任意一条指令，任意一个阶段，能否进入（或者说开始执行它要执行的任务）的唯一审视标准是：**进入条件是否满足。**

8 .Tomasulo算法实例动态演示

load：2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

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2			
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
0	FU									

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1		
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
1	FU			Load1					

周期1：取出第一条指令L.D F6, 34(R2)，地址偏移量34,写入LOAD1，LOAD1名存入寄存器F6。

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1		
LD	F2	45+	R3	2		
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	~~Yes~~	~~34+R2~~
Load2	Yes	45+R3
Load3	~~No~~	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
2	*FU*		Load2		Load1					

周期2：取出第二条指令L.D F2, 45(R3)，地址偏移量45写入LOAD2，LOAD2名存入寄存器F2，同时第一条指令开始执行，LOAD1上写入绝对地址

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	
LD	F2	45+	R3	2		
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
3	*FU*	Mult1	Load2		Load1					

周期3：取出第三条指令MUL.D F0, F2,F4，第一条指令完成，第二条指令开始执行,LOAD2上写入绝对地址。保留站中存入待运算的操作数和操作。寄存器F0上写入保留站中待运算命令的名称。

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	Yes	SUBD	M(A1)			Load2
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
4	FU	Mult1	Load2		M(A1)	Add1				

周期4：取出第四条指令SUB.D F8,F6,F2，第二条指令执行完成，第一条指令写入结果M(A1)到寄存器F6，保留站中存入第四条指令的待运算操作数和操作。LOAD1清空。

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
2	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	No					
	Add3	No					
10	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock **5**

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
FU	Mult1	M(A2)		M(A1)	Add1	Mult2			

周期5：取出第五条指令DIV.D F10,F0,F6，第二条指令写结果M(A2)到寄存器F2，LOAD2清空。保留站中存入第五条指令的待运算操作数和操作

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
1	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
9	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
6	*FU*	Mult1	M(A2)		Add2	Add1	Mult2			

周期6：取出第六条指令ADD.D F6,F8,F2，第三条和第四条指令开始执行，相关的操作数和操作符被存入保留站

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
0	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
8	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
7	*FU*	Mult1	M(A2)		Add2	Add1	Mult2			

周期7：第四条指令执行完成，保留站中的第三条指令继续执行

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
2	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
7	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
8	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

周期8：第四条指令写结果M-M到寄存器F8，保留站中存放第四条指令的位置清空，第三条指令继续执行。

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
1	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
6	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
9	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

周期9：第六条指令开始执行。第三条指令继续执行。

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
0	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
5	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
10	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

周期10：第六条指令执行完成，第三条指令继续执行。

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
4	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
11	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

周期11：第六条指令写结果M-M+M(A2)到寄存器F6中，清空保留站中原来存放第六条指令的位置。第三条指令继续执行。

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
3	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
12	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

周期12~15：第三条指令继续执行，直到完成。

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
2	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
13	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
1	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
14	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
0	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
15	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

in-order issue, out-of-order execution, completion

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
40	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
16	*FU*	M*F4	M(A2)		(M-M+M	(M-M)	Mult2			

周期16：第三条指令写结果M*F4到寄存器F0，保留站中原来存放第三条指令的位置清空。

周期17~56：第五条指令开始执行，直到结束；

周期57：第五条指令执行结束后写结果M6到F10,保留站中原来存放第五条指令的位置清空。

例 单流出处理器采用基于Tomasulo算法进行指令调度。有一个LSU部件（部件内有加法器），2个LS缓冲器（私有时间戳实现FIFO），一个加/减法运算部件，一个乘/除法运算部件，2个ALU保留站。指令序列执行前，指令均未流出，所有缓冲器/保留站均空闲。

（1）各个硬件操作流水段及指令通过的时钟周期如下表

Issue	WtCDB	Mem	Execute				
			LD	ST	SUB	ADD	MUL
1	1	3	1	1	4	4	10

（2）待执行指令序列如下：

指令	对应变量
LD R2, (R1)	Rt=R2, Rs=R1, Imm=0
MUL R2, R2, #2	Rd=R2, Rs=R2, Rt= #2
ST R2, (R1)	Rt=R2, Rs=R1, Imm=0
SUB R1, R1, #4	Rd=R1, Rs=R1, Rt= #4
ADD R5, R3, R4	Rd=R5, Rs=R3, Rt=R4

问：

（1）请给出指令执行状态时钟周期表

（2）请给出第14周期末尾各个状态表内容

请挑错

指令	IS	EX	MEM	WtCDB	备注	14末
LD R2, (R1)	1	2	3-5	6		已完成
MUL R2, R2, #2	2	7-16	Null	17	数据相关	已执行
ST R2, (R1)	3	4	18-20	Null	数据相关	已执行
SUB R1, R1, #4	4	5-8	Null	9	R1已换名	已写结果
ADD R5, R3, R4	5	9-13	Null	14	结构冲突	已执行

指令	IS	EX	MEM	WtCDB	备注	14末
LD R2, (R1)	1	2	3-5	6		已完成
MUL R2, R2, #2	2	7-16	Null	17	数据相关	已执行
ST R2, (R1)	3	4	18-20	Null	数据相关	已执行
SUB R1, R1, #4	4	5-8	Null	9	R1已换名	已写结果
ADD R5, R3, R4	10	11-14	Null	15	保留站冲突	已执行

Label	Busy	Op	Vj(rs)	Vk(rt)	Qj(rs)	Qk(rt)	A(Imm)
Load1	N						
Load2	Y	ST	Reg[R1]			ALU1	Reg[R1]
ALU1	Y	MUL	Reg[R2]	#2			
ALU2	Y	SUB->ADD	Reg[R3]	Reg[R4]			

	R1	R2	R3	R4	R5
Qi		ALU1			ALU2

9. Tomasulo的要点总结

① 指令按照程序原有的顺序进入指令队列，并按照原有的顺序流出指令队列，但是并不一定按照原有的顺序执行。

② 操作数未就绪的指令在保留站中等待，操作数就绪的指令可以先被执行。

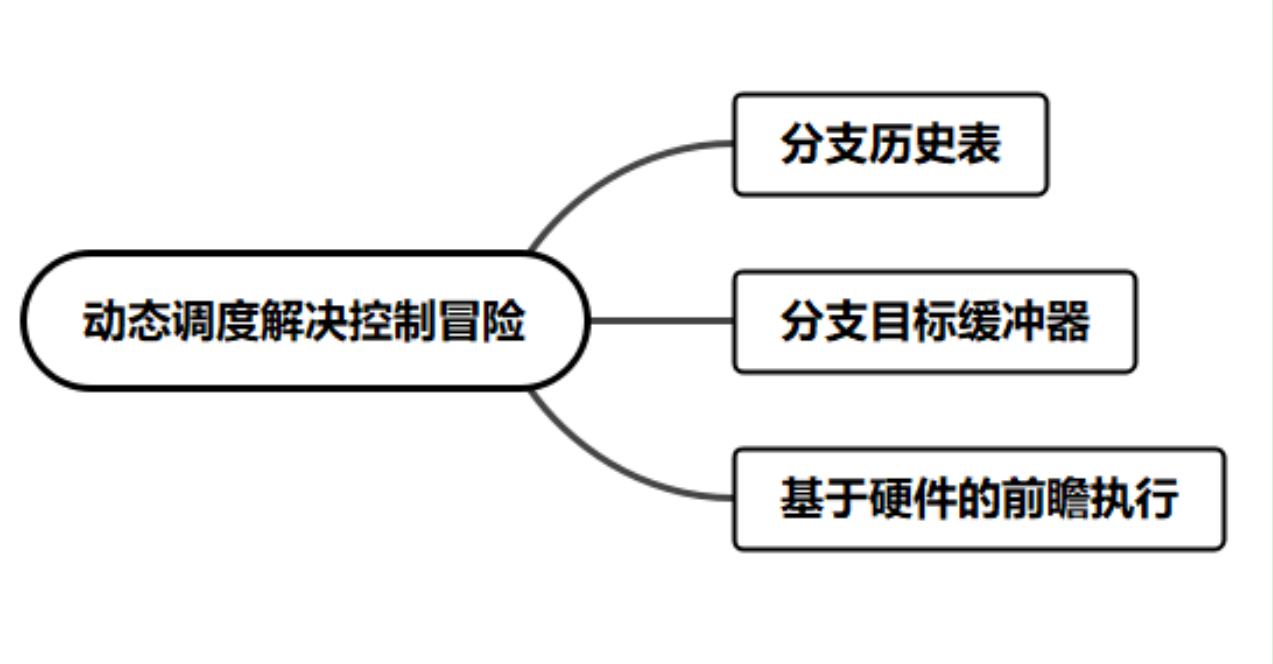
③ 通过寄存器换名技术消除了名相关，从而解决了读后写和写后写冲突。换了什么名呢？把流出到保留站中的指令的操作数寄存器换成了操作数，或者产生该操作数的保留站号。

④ 指令能够流出、执行和写结果这三步的进入条件请看详细的算法，但是还有一个隐含条件，即没有结构冲突。

⑤ Tomasulo算法只能处理结构冲突和数据冲突，不能处理异常和控制冲突。指令队列里有分支指令，分支指令之后的指令不能流出，直到确定分支是否成功。

⑥ 在算法中，有两个地方容易忽视，需要格外注意：一个是从Load/store指令要按照FIFO次序访存；另一个是，store指令执行阶段和写结果阶段进入的条件。

四 动态调度解决控制冒险



什么是动态分支预测

动态分支预测技术：

通过硬件技术，在程序执行时根据每一条转移指令过去的转移历史记录来预测下一次转移的方向。通过提前预测分支方向，减少或消除控制相关导致的流水线停顿。

优点：

- 根据程序的执行过程动态地改变转移的预测方向，因此有更好的准确度和适应性。
- 程序每次执行时，可能预测的分支方向与前次相同或不同。

从简单到复杂的动态转移预测技术如下：

- 分支预测缓存器（BHT）
- 分支目标缓冲器（BTB）
- 基于硬件的前瞻执行(ROB)

以上预测机制的性能随着复杂性与硬件的增加而有效地提高。

要点：

- 动态分支预测技术和静态分支预测技术的区别
- 什么叫“分支指令过去的表现”？
- 如何衡量分支预测的有效性？
- 分支预测要解决的问题是什么？
- 需要解决的关键问题？

1. 动态分支预测技术和静态分支预测技术的区别

静态分支预测技术所进行的操作事先预定好的 ，与分支的实际执行情况无关；

动态分支预测技术的方法在程序运行时根据分支执行过去的表现预测其将来的行为（如果分支行为发生了变化，预测结果也跟着改变，此外有更好的预测准确度和适应性）。

2. 什么是“分支指令过去的表现”？

就是记录分支的历史信息，在预测错误时，要作废已经预取和分析的指令，恢复现场，为了恢复现场，需要在执行预测的目标指令之前将现场保存起来。

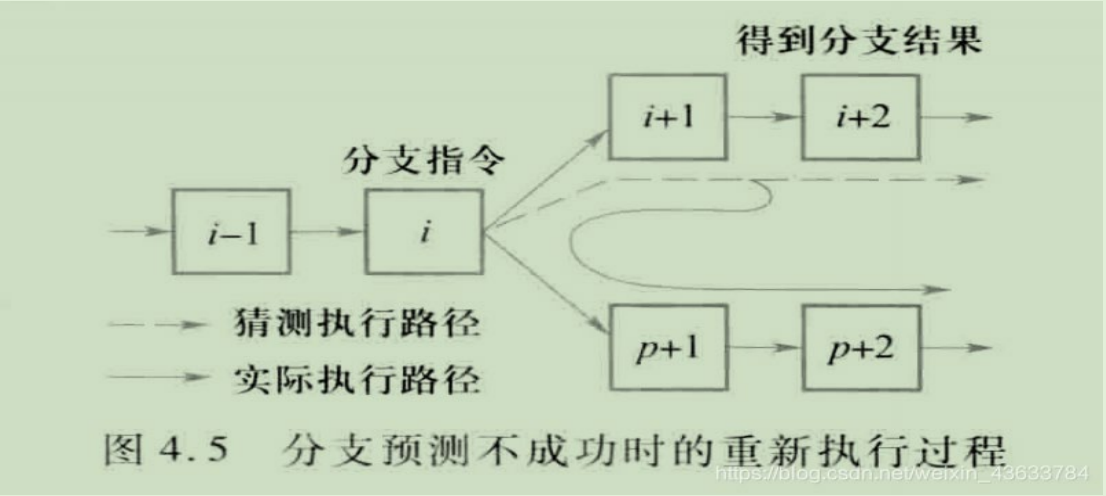


图 4.5　分支预测不成功时的重新执行过程

3. 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（5段流水，分支处理由MEM段提到ID）
  - 预测的方法
  - 预测错误时的恢复策略等

4. 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）

5. 需要解决的关键问题

- 如何记录分支的历史信息；
- 如何根据这些分支的历史信息来预测分支的去向（甚至取到指令）。

4.1 分支历史表 BHT

要点：

1. 分支历史表
2. 只有1个预测位的分支历史表
3. 采用两位二进制位记录历史
4. 转移历史表BHT的逻辑结构
5. BHT方法适用情况

1. 分支历史表BHT（Branch History Table）

或分支预测缓冲器（Branch Prediciton Buffer）

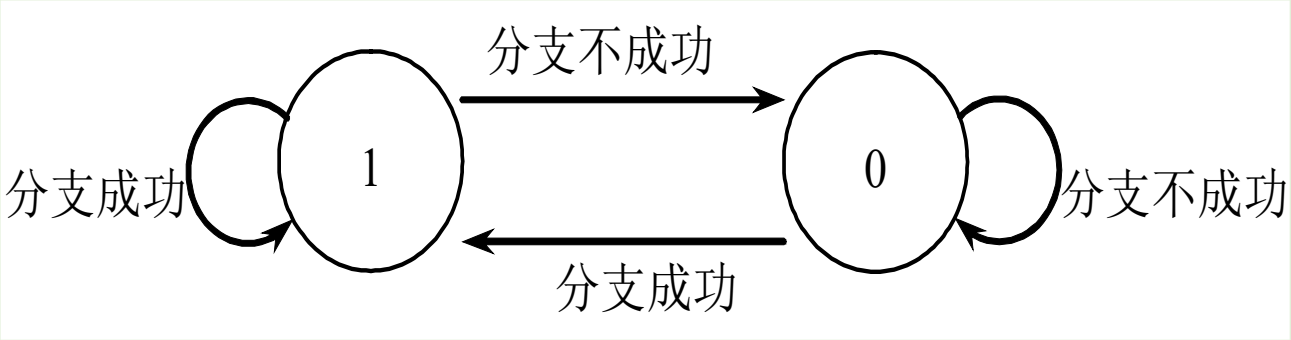
- 最简单的动态分支预测方法。
- 用BHT来记录分支指令最近一次或几次的执行情况（成功或不成功），并据此进行预测。

2. 只有1个预测位的分支预测缓冲

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）

- “0”记录分支不成功
- “1”记录分支成功
- 遇见1，预测成功。实际成功则保持1，实际失败置0
- 遇见0，预测失败。实际失败则保持0，实际成功置1



只有1个预测位的分支预测缓冲状态转换图

① 分支预测缓冲技术包括两个步骤

- 分支预测

  如果当前缓冲记录的预测位为“1”，则预测分支为成功；如果预测位为“0”，则预测分支为不成功。

- 预测位修改

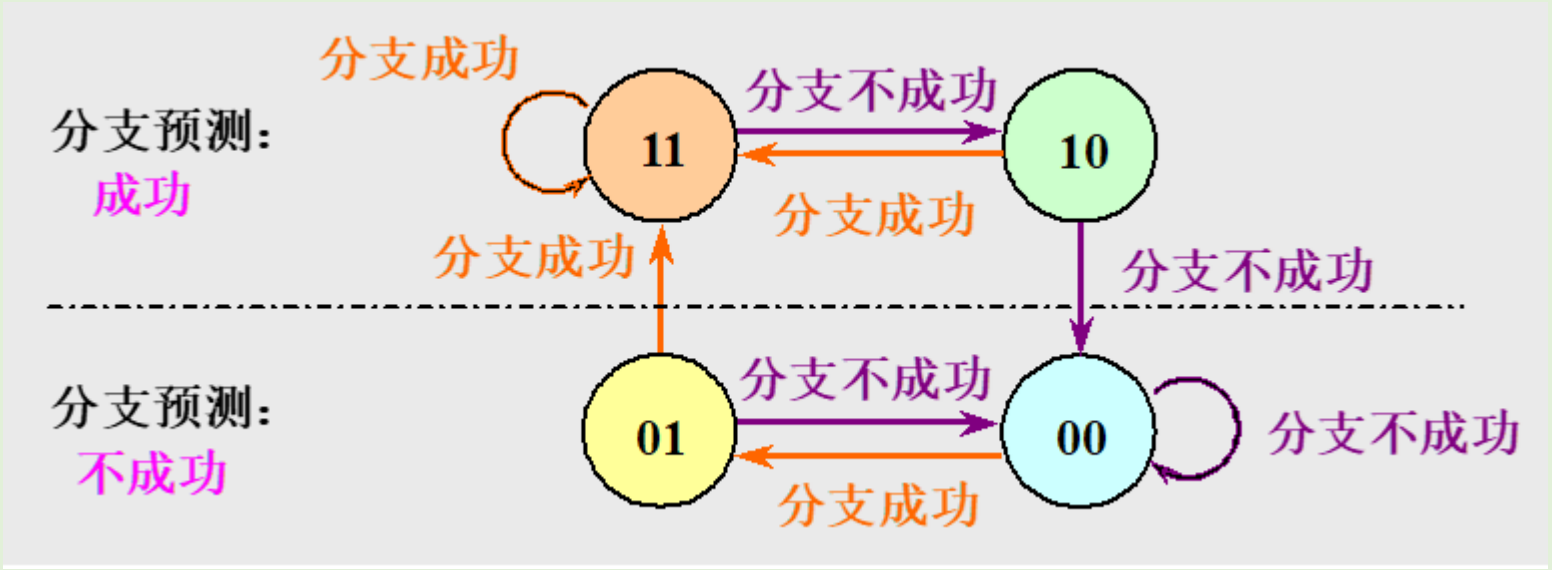
  如果当前分支成功，则预测位置为“1”； 如果当前分支不成功，预测位置为“0”。

② 分支预测错误时，预测位就被修改，并且需要恢复现场，程序从分支指令处重新执行。

实例：

```
Loop : L.D     F0, 0(R1)
       ADD.D   F4, F0, F2
       S.D     F4, 0(R1)
       DADDIU  R1, R1, #-1
       BNE     R1, R2, Loop   (0)
       L.D     F8, 0(R1)

初始：(R1) = 1A H ； (R2) = 10 H
    BHE 指令后面括号里的（0）代表它的分支历史表 BHT 表项。以下用 BHT(BNE) 来表示。

第一次循环：
Loop : L.D     F0, 0(R1)
       ADD.D   F4, F0, F2
       S.D     F4, 0(R1)
       DADDIU  R1, R1, #-1          //(R1) = 19 H
       BNE     R1, R2, Loop   (0)   //由于 BHT（BNE）=0，因此预测分支失败，但是实际是成功的!! 将 BHT（BNE）置为“1”。
       L.D     F8, 0(R1)

第二次循环：
Loop : L.D     F0, 0(R1)
       ADD.D   F4, F0, F2
       S.D     F4, 0(R1)
       DADDIU  R1, R1, #-1          //(R1) = 18 H
       BNE     R1, R2, Loop   (1)   //由于 BHT（BNE）=1，因此预测分支成功，实际也是成功的，保持 BHT（BNE）为“1”。
       L.D     F8, 0(R1)

第十次循环：
    Loop : L.D     F0, 0(R1)
       ADD.D   F4, F0, F2
       S.D     F4, 0(R1)
       DADDIU  R1, R1, #-1          //(R1) = 10 H
       BNE     R1, R2, Loop   (1)   //由于 BHT（BNE）=1，因此预测分支成功，实际上不成功，执行后将 BHT（BNE）置为“0”。
       L.D     F8, 0(R1)


整个过程，预测错误 2 次。
```

3. 采用两位二进制位来记录历史



上图读解：

（a）约定一下，左边为第一位，右边为第二位，比如“01”，“0”为第一位，“1”位第二位。

（b）当前状态下，预测啥看第一位，比如“01”，第一位是“0”，则预测分支是失败的（不成功）。

（c）上图分成上下两部分：上半部分两个状态第一位都是“1”，处在该状态的分支指令，将会被预测分支成功；下半部分两个状态第一位都是“0”，处在该状态的分支指令，将会被预测分支失败。

（d）连续2次预测错误会导致翻转，也就是从上半部分进入下半部分，或者从下半部分进入上半部分。比如当前处于“01”状态，也就是前一次已经预测错误了，此次将仍然预测分支是失败的，但是如果分支成功了（也就是预测错了），将会更改为“11”状态。

（e）没看懂多看几遍。

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。

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。

研究结果表明：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%。

一般来说，采用4K的BHT就可以了。

BHT可以跟分支指令一起存放在指令Cache中，也可以用一个专门的硬件来实现。

4.2 分支目标缓冲器BTB

要点：

1. 背景、目标、方法
2. BTB的结构
3. 采用BTB在流水线各个阶段操作
4. 采用BTB各种可能情况下的延迟
5. 举例
6. 分支指令可能存在3种情况
7. BTB的另一种形式

1. 为什么要用分支目标缓冲器

背景：

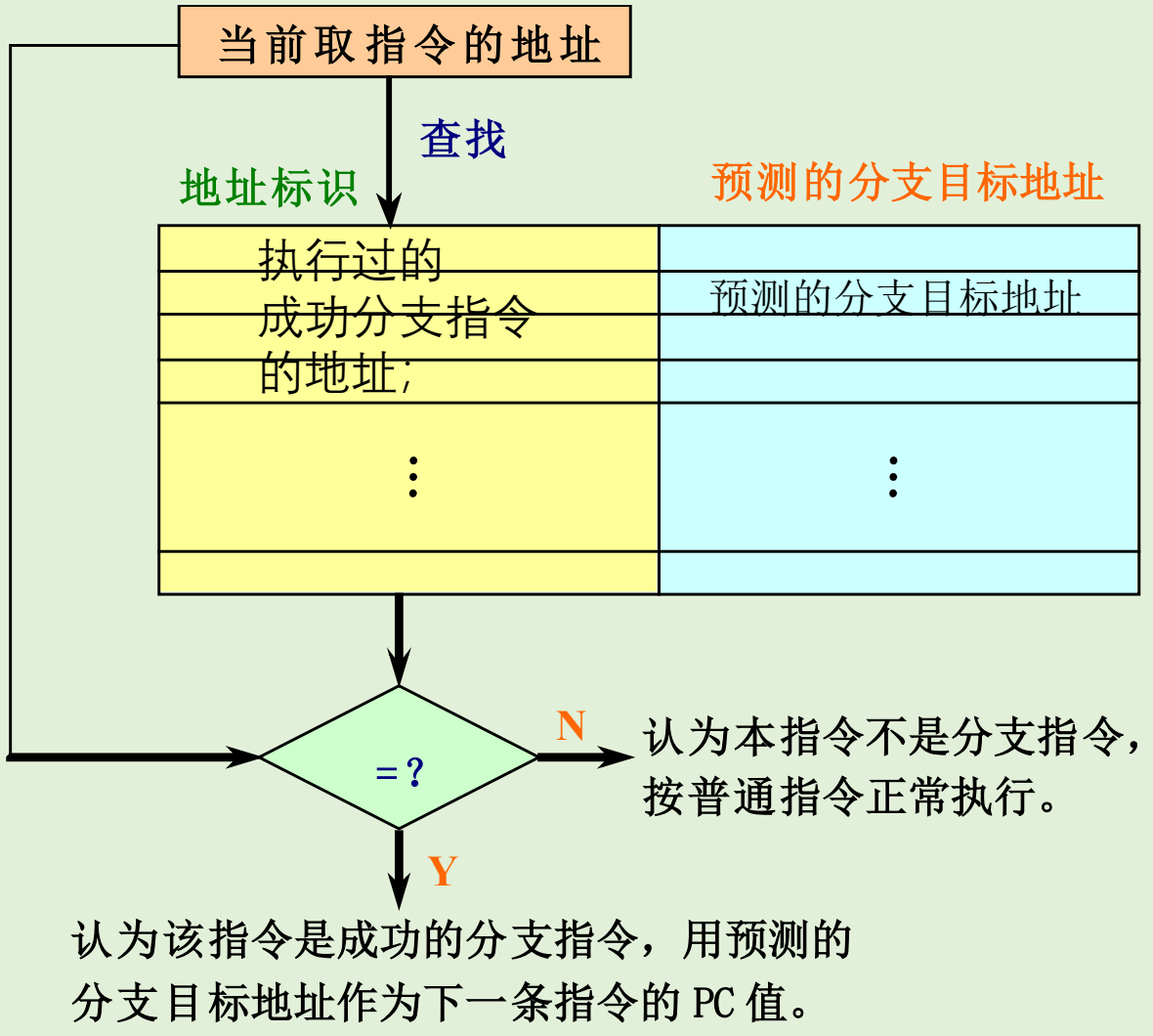
在高性能流水线中，特别是多流出的处理机中，只准确地预测分支还不够，还要能够提供足够的指令流。许多现代的处理器要求每个时钟周期能提供4～8条指令。这需要**尽早**知道分支是否成功、**尽早**知道分支目标地址、**尽早**获得分支目标指令。

方法：分支目标缓冲

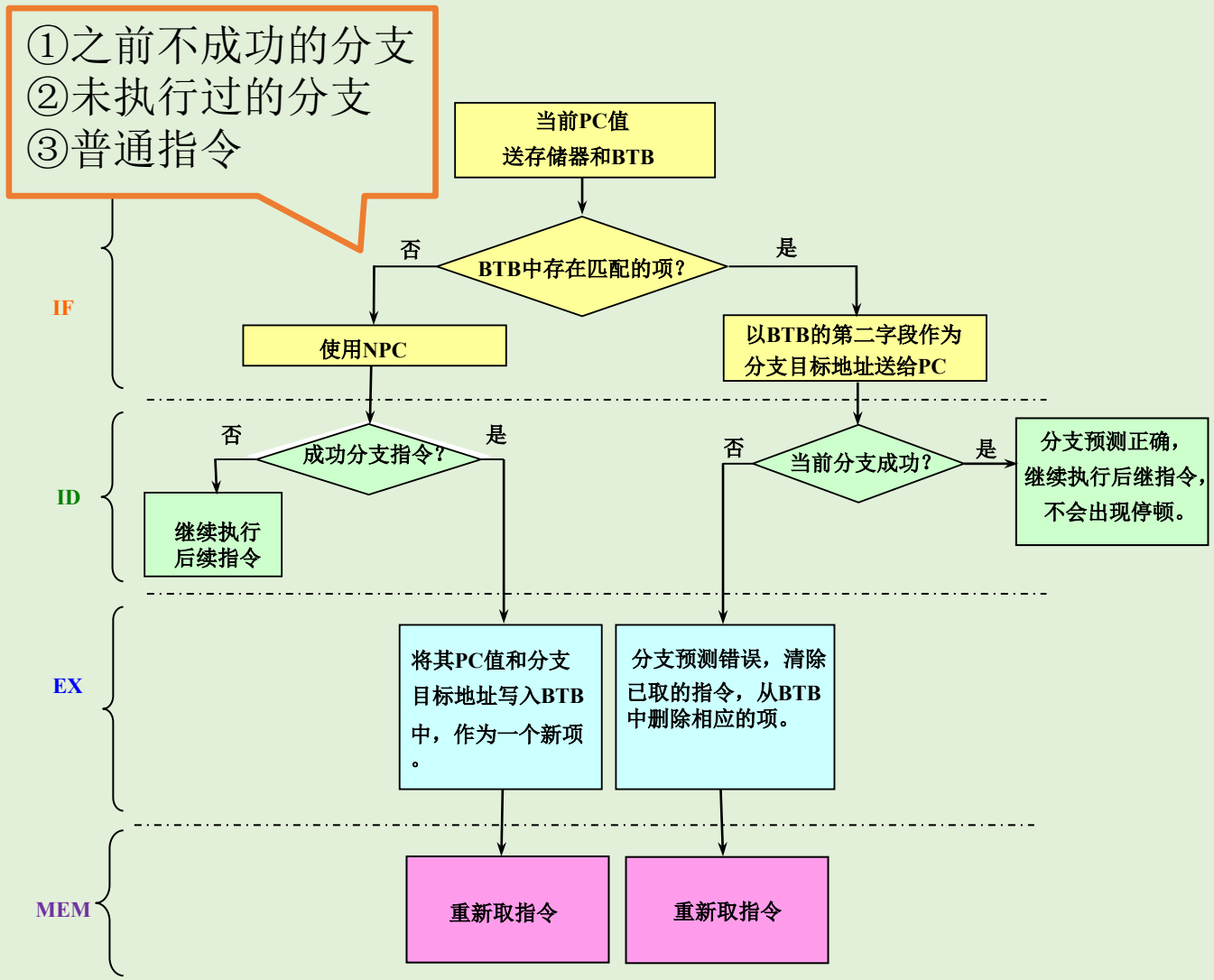
- 将分支成功的分支指令的地址和它的分支目标地址都放到一个缓冲区中保存起来，缓冲区以分支指令的地址作为标识。
- 这个缓冲区就是分支目标缓冲器（Branch-Target Buffer，简记为BTB，或者Branch-Target Cache）。

2. BTB表结构

- **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。



3. 采用BTB后，在流水线各个阶段所进行的相关操作：



如果预测错误或在BTB中没有匹配的项，要有至少2个时钟周期的开销。

因为：

① 需要更新BTB中的项，要花费一个时钟周期；

② 在更新BTB 中的项时，要停止取指令，那么取新的 指令又要花费一个时钟周期。

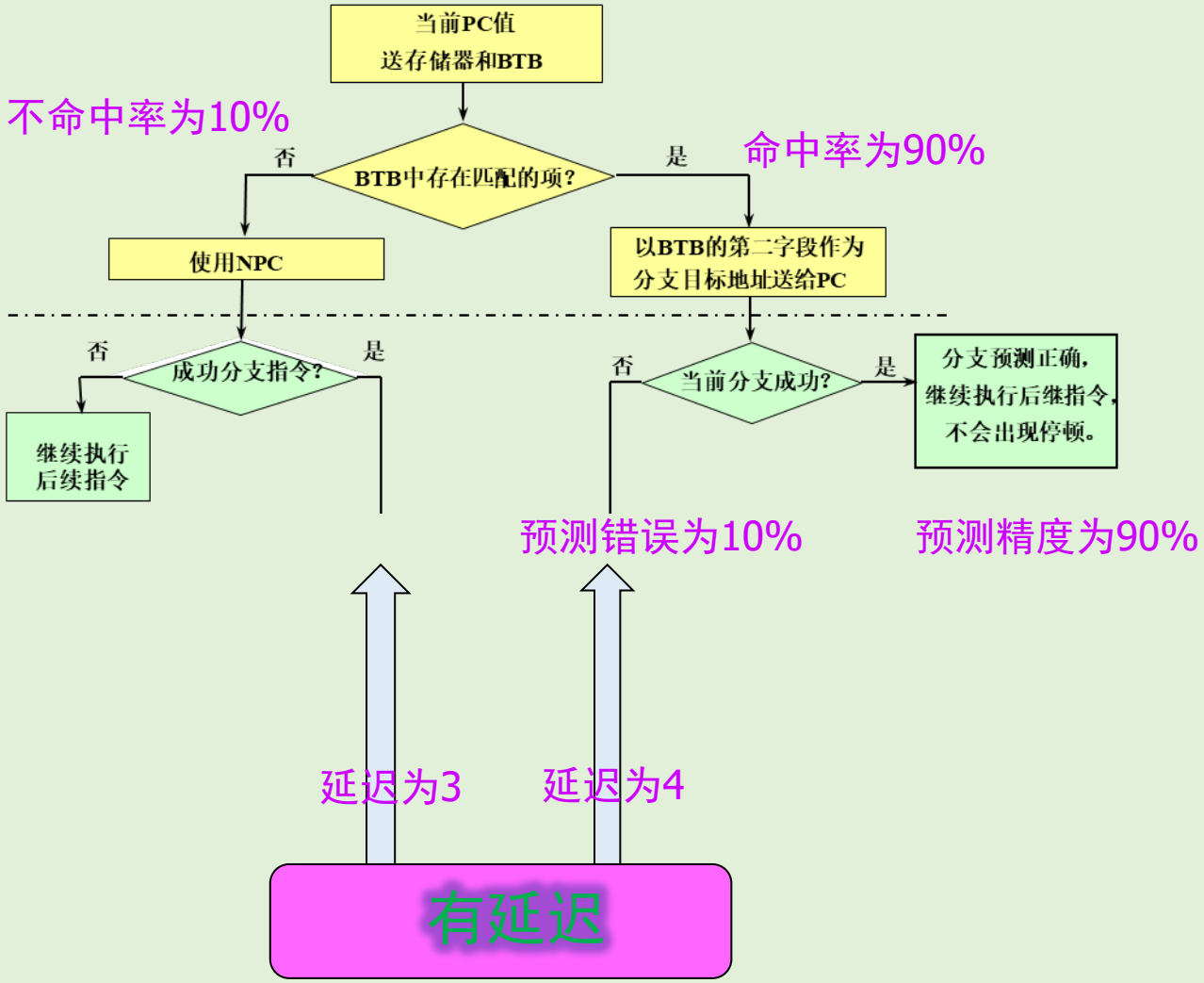
指令在BTB中？	预测	实际情况	延迟周期
是	成功	成功	0
是	成功	不成功	2
不是		成功	2
不是		不成功	0

4. 举例：

假设有一条长流水线，仅仅对条件转移指令使用分支目标缓冲。假设分支预测错误的开销为4个时钟周期，缓冲不命中的开销为3个时钟周期。假设：命中率为90%，预测精度为90%，分支频率为15%，没有分支的基本CPI为1。

（1）求程序执行的CPI。

（2）相对于采用固定的2个时钟周期延迟的分支处理，哪种方法程序执行速度更快？



解：

（1）程序执行的CPI = 没有分支的基本CPI（1）

+ 分支带来的额外开销

分支带来的额外开销是指在分支指令中，缓冲命中但预测错误带来的开销与缓冲没有命中带来的开销之和。

分支带来的额外开销= 15% * (90%命中×10%预测错误×4 + 10%没命中×3)= 0.099

所以，程序执行的CPI = 1 + 0.099 = 1.099

（2）采用固定的2 个时钟周期延迟的分支处理

CPI = 1 + 15%×2 = 1.3

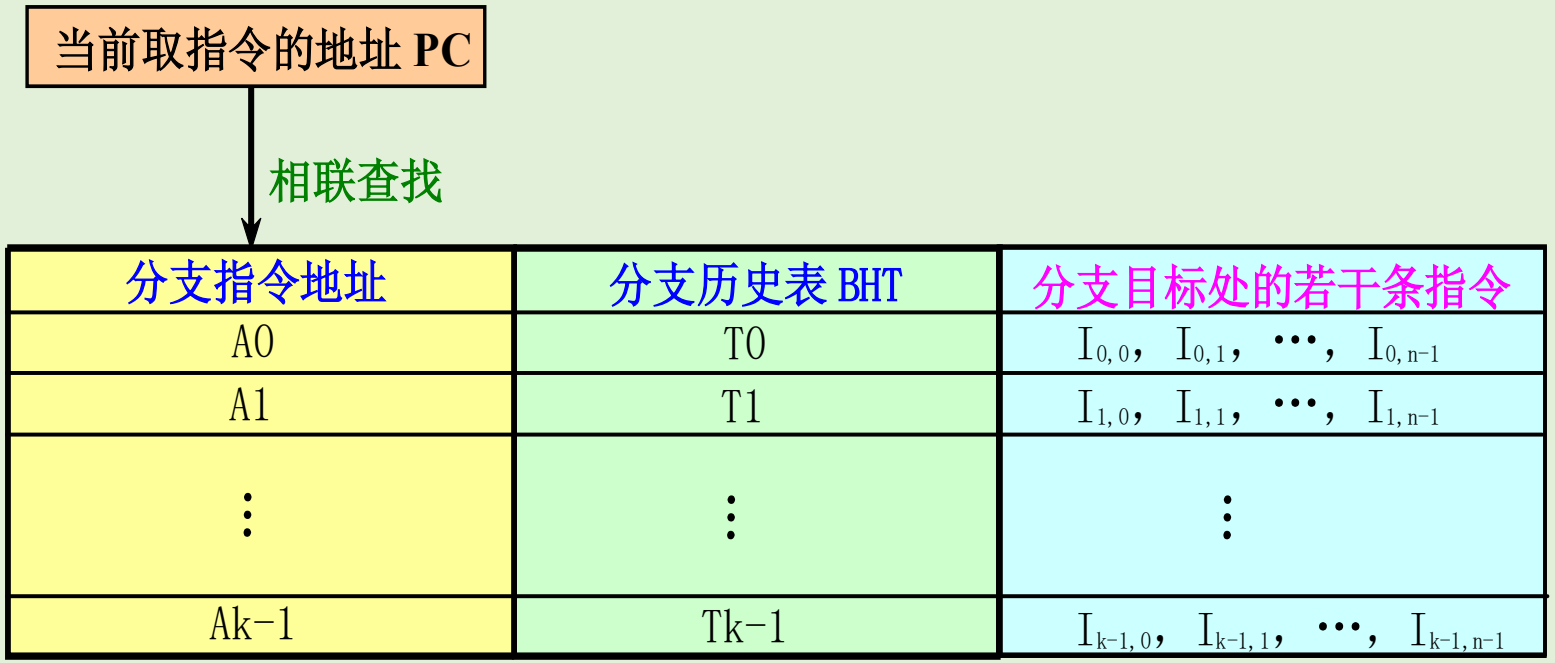
由（1）（2）可知分支目标缓冲方法执行速度快。

5. BTB的另一种形式

在分支目标缓冲器中存放一条或者多条分支目标处的指令。

- 有三个潜在的好处：
  - 更快地获得分支目标处的指令；
  - 可以一次提供分支目标处的多条指令，这对于多流出处理器是很有必要的；
  - 使我们可以进行称为分支折叠（branch folding）的优化。

    实现零延迟无条件分支，甚至有时还可以做到零延迟条件分支。



4.3 基于硬件的前瞻执行

过程要点：

1. 流出、执行、写结果、确认四个阶段
2. 顺序流出、准备好的先执行（乱序执行），顺序确认
3. 分支预测错误，刷新ROB
4. 异常指令到达ROB头部，刷新ROB

Tomasulo算法能够做什么？不能够做什么？

能做：

- 解决RAW冲突 （CDB）
- 消除WAW和WAR冲突(换名)
- 浮点运算，load/store运算

不能做：

- 分支预测
- 处理异常

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出、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，而是放到一个称为ROB（ReOrder Buffer）的缓冲器中。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。

目的：在猜测错误时能够恢复现场（即没有进行不可恢复的写操作）

动态预测+猜测执行+保留站技术

为了得到更高的并行性，将动态转移预测技术与Tomasulo结合，设计出了前瞻执行方法：处理器按转移预测方向，乱序执行推测的指令序列，如果预测正确，就能够消除控制相关的延迟。也解决了数据相关和消除名相关。如果预测错误，则废弃已经执行的指令序列，从另一个分支处重新开始执行。

基本要点：动态预测+猜测执行+保留站技术

基于硬件的前瞻执行结合了三种思想：

① 动态分支预测。用来选择后续执行的指令。

② 在控制相关的结果尚未出来之前，前瞻地执行后续指令。

③ 用动态调度对基本块的各种组合进行跨基本块的调度。（基本程序块：如果 一串连续的代码除了入口和出口以外，不包含其他分支指令和转入点。）

实质是**数据流执行(data flow execution)：只要操作数有效，指令就执行**

对Tomasulo算法加以扩充，就可以支持前瞻执行。

Tomasulo算法的写结果和指令完成都在“写结果”段完成，而在前瞻执行中加以区分，分成两个不同的段：

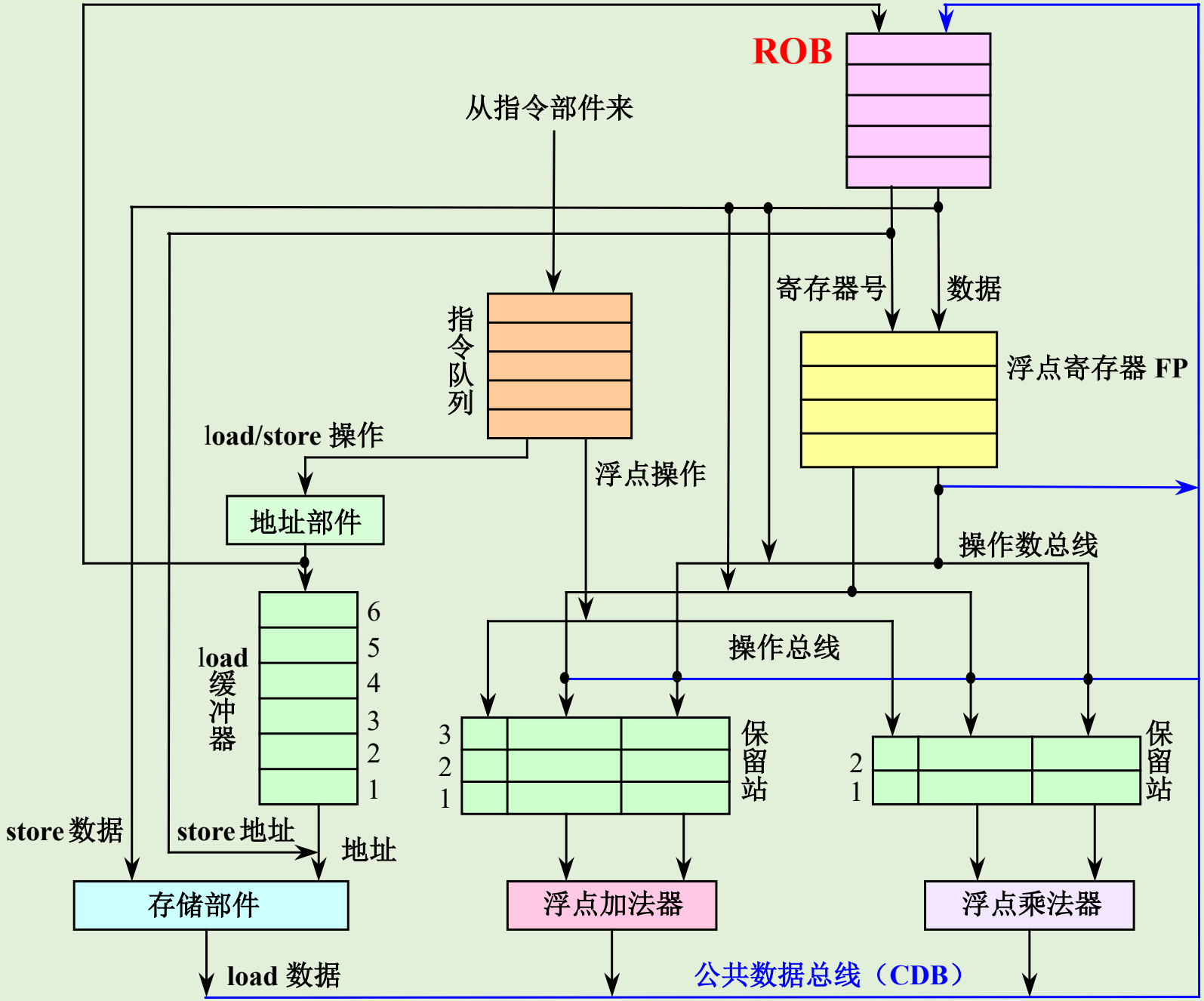
①写结果段

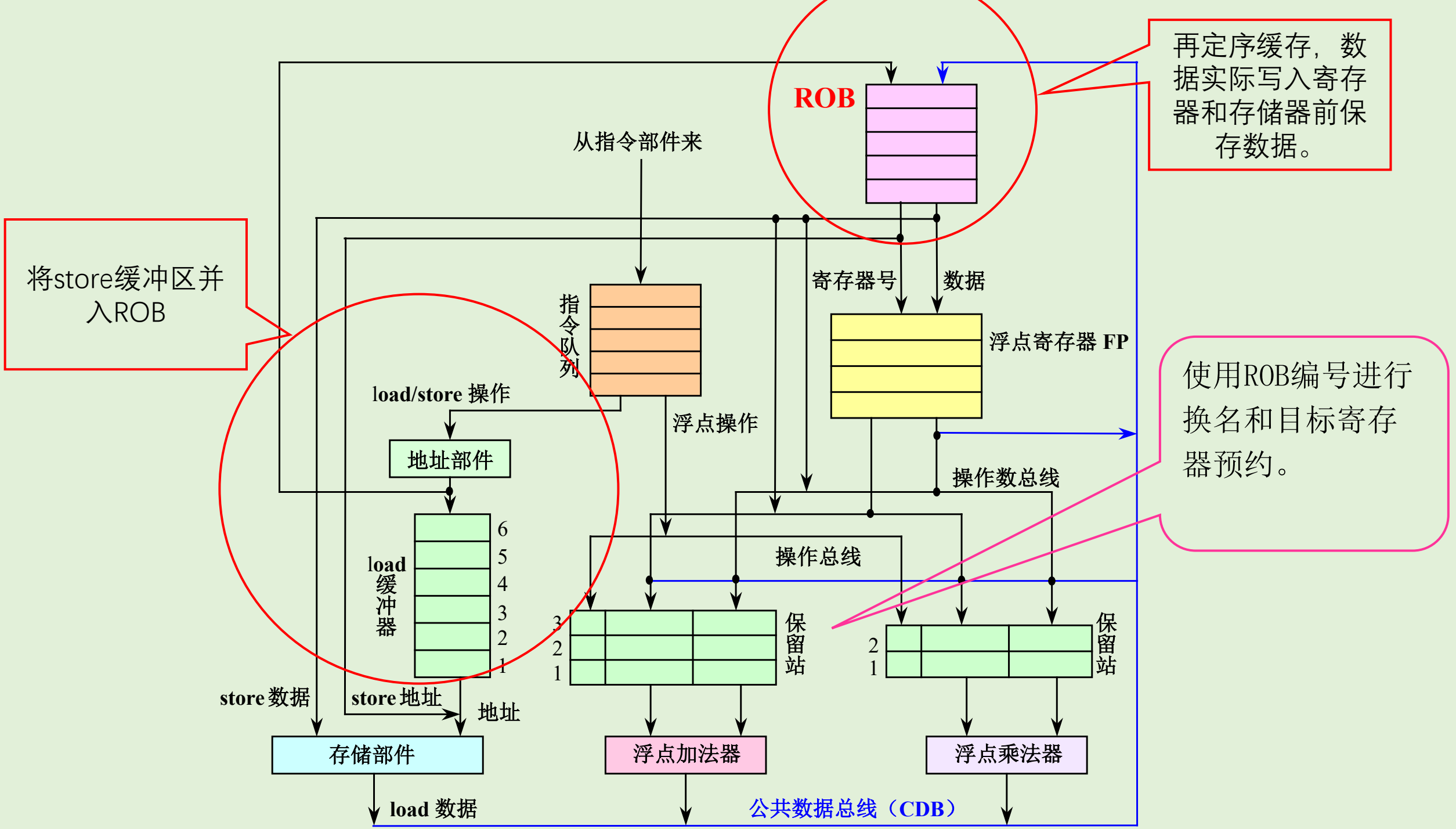
- 把前瞻执行的结果写到ROB中；
- 通过CDB在指令之间传送结果，供需要用到这些结果的指令使用。

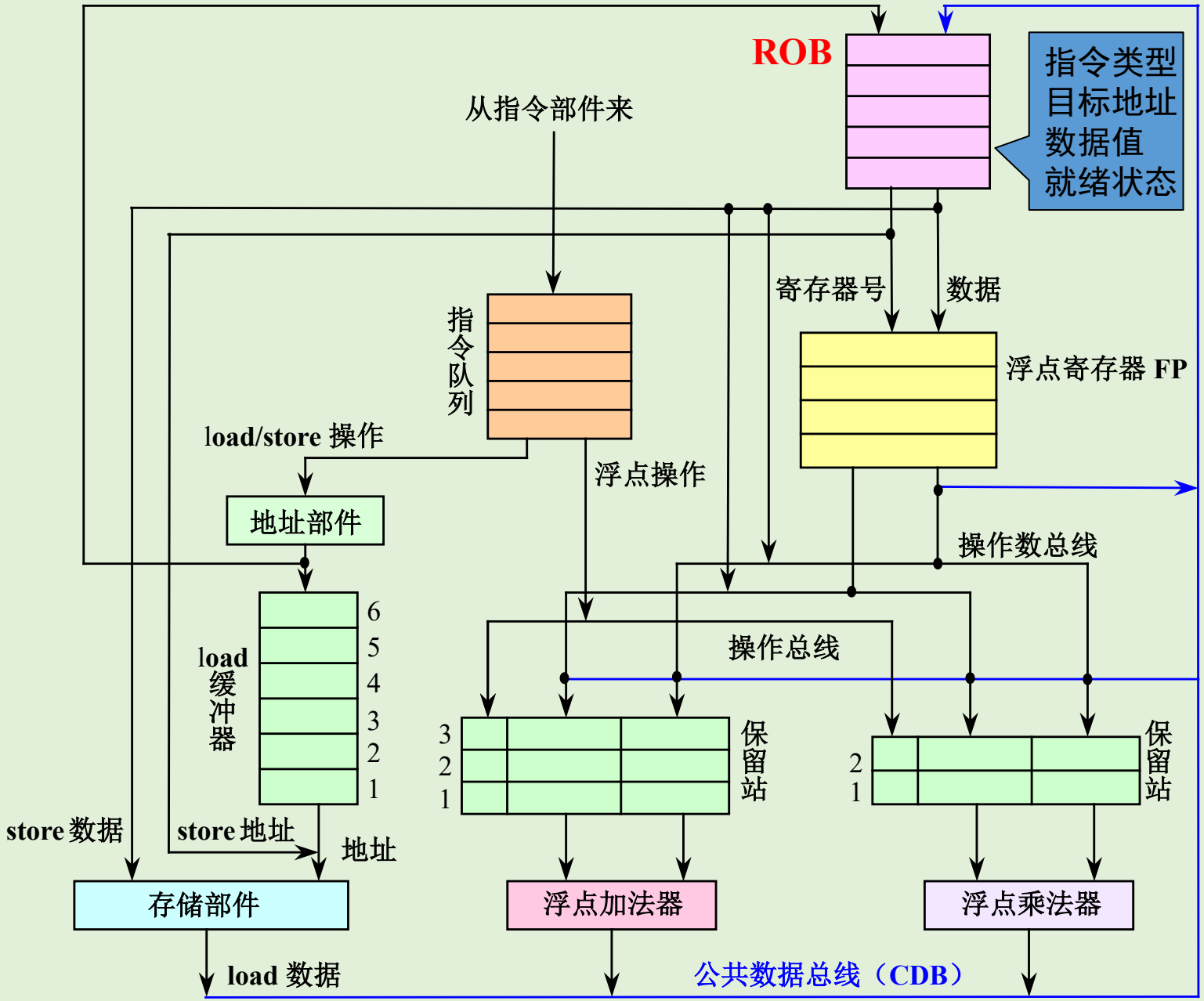
②指令确认段

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。


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
6
5
4
3
2
1
load 缓冲器
操作总线
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
从指令部件来
再定序缓存，数据实际写入寄存器和存储器前保存数据。
将store缓冲区并入ROB
寄存器号
数据
浮点寄存器 FP
指令队列
load/store 操作
浮点操作
地址部件
操作数总线
使用ROB编号进行换名和目标寄存器预约。
load缓冲器
6
5
4
3
2
1
操作总线
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。

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。
- 如果该指令需要的操作数在寄存器或者ROB中就绪，就将其值取到保留站r中；否则，将产生其值的ROB编号写入。
- ROB的编号写入r。
- 用对应的ROB编号对目标寄存器进行预约。

流出（Tomasulo算法）

- 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r。
- 如果该指令需要的操作数在寄存器中就绪，就将值取到保留站r中；如果没就绪，将产生该操作数的保留站的标识送到保留站r。
- 完成对目标寄存器的预约工作。

执行

- 如果有操作数尚未就绪，就等待，并不断地监测CDB。
- 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。
- Load指令做有效地址运算和读取数据；store指令只做有效地址运算。

执行（Tomasulo算法）

- 如果有操作数尚未就绪，就等待，并不断地监测CDB。
- 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。
- Load指令做有效地址运算和读取数据；store指令只做有效地址运算。

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，这时才将之写入分配给该store指令的ROB项。

写结果（tomasulo算法）

- 当结果产生后，将该结果放到CDB上，经CDB写到所有等待该结果的寄存器和保留站中。
- 当写入地址和数据齐备时，Store指令完成存储器的写入。
- 释放产生该结果的保留站。

确认 （Tomasulo算法无该阶段）

对分支指令、**store**指令以及其他指令的处理不同:

- 其他指令（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标寄存器，并从ROB中删除该指令。

- store指令

  处理与上面类似，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。

- 异常指令（实现精确异常）

  在该指令到达ROB头部前不处理，到达ROB头部时，产生中断，清空ROB，此操作能够完成精确的异常处理。但凡刷新ROB，连同指令队列一起刷新，因为假如指令队列中有指令已经是有问题的指令流。

要点汇总：

- 顺序流出指令队列；
- 猜测路径和乱序执行；
- 顺序确认（保证顺序写回）；
- 猜错或发生异常，没写回的全部不算数。

主要优点：

1. 动态调度；
2. 前瞻执行；
3. 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。

例题： 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
```

名称	保留站							
	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Add1	no							
Add2	no							
Add3	no							
Mult1	no	MUL	Mem[45+ Regs[R2]]	Regs[F4]			#3	
Mult2	yes	DIV	#2×Regs[F4]	Mem[34+Regs[R2]]			#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

Dest字段指名是哪个ROB的项将接受该保留站产生的结果

例 单流出处理器采用基于Tomasulo的IS算法进行指令调度。有一个LSU部件（内部自带加法器），2个LS缓冲器（私有时间戳实现FIFO），1个加法ALU部件，1个乘法ALU部件，2个ALU保留站。总是预测分支失败。指令序列执行前，指令均未流出，所有缓冲器/保留站均空闲。分支指令由加法ALU部件执行。ROB空间足够。

（1）各个硬件操作及指令执行的时钟周期如下表

Issue	WtCDB	Mem	Commit	Execute					
				LD	ST	SUB	BNEZ	ADD	MUL
1	1	3	1	1	1	4	4	4	10

（2）待执行指令序列如下：

指令	对应变量
Loop: LD R2, (R1)	Rt=R2, Rs=R1, Imm=0
MUL R2, R2, #2	Rd=R2, Rs=R2, Rt= #2
ST R2, (R1)	Rt=R2, Rs=R1, Imm=0
SUB R1, R1, #4	Rd=R1, Rs=R1, Rt= #4
BNEZ R1, Loop	Rt=Loop, Rs=R1
ADD R5, R3, R4	Rd=R5, Rs=R3, Rt=R4

问：

（1）请给出指令执行状态时钟周期表

（2）给出第16周期末尾各状态表内容

Issue	WtCDB	Mem	Commit	Execute					
				LD	ST	SUB	BNEZ	ADD	MUL
1	1	3	1	1	1	4	4	4	10

指令	IS	EX	MEM	WtCDB	Cmt	备注	16末状态
LD R2, (R1)	1	2	3-5	6	7		已确认
MUL R2, R2, #2	2	7-16	Null	17	18	数据相关	已执行
ST R2, (R1)	3	4	20-22	18	19	数据相关	已执行
SUB R1, R1, #4	4	5-8	Null	9	20	R1已换名	已写结果
BNEZ R1, Loop	10	11-14	Null	15	21	分支失败	已写结果
ADD R5, R3, R4	16	17-20	Null	21	22	保留站冲突	已流出

画圈部分说明：此处的设计，按照ST指令也进入保留站。ST监听CDB，第二个操作数就绪，先写回load2的Vk，在wtCDB阶段写入ROB3的value，然后清空保留站。

Label	Busy	Op	Vj(rs)	Vk(rt)	Qj(rs)	Qk(rt)	A(Imm)	Dest
Load1	N							
Load2	Y	ST	Reg[R1]			ROB2	Reg[R1]	ROB3
ALU1	Y	MUL	Reg[R2]	#2				ROB2
ALU2	Y	SUB ->BNEZ ->ADD	Reg[R3]	Reg[R4]				ROB4 ->ROB5 ->ROB6

保留站状态表

指令	IS	EX	MEM	WtCDB	Cmt	备注	16末状态
LD R2, (R1)	1	2	3-5	6	7		已确认
MUL R2, R2, #2	2	7-16	Null	17	18	数据相关	已执行
ST R2, (R1)	3	4	20-22	18	19	数据相关	已执行
SUB R1, R1, #4	4	5-8	Null	9	20	R1已换名	已写结果
BNEZ R1, Loop	10	11-14	Null	15	21	分支失败	已写结果
ADD R5, R3, R4	16	17-20	Null	21	22	保留站冲突	已流出

ROB状态表

Label	Busy	Op	Target	Value	State
ROB1	N	LD			已确认
ROB2	Y	MUL	R2		已执行
ROB3	Y	ST	Mem(Reg[R1])		已执行
ROB4	Y	SUB	R1	Reg[R1]-4	已写结果
ROB5	Y	BNEZ	#Loop	Reg[R1]-4	已写结果
ROB6	Y	ADD	R5		已流出

指令	IS	EX	MEM	WtCDB	Cmt	备注	16末状态
LD R2, (R1)	1	2	3-5	6	7		已确认
MUL R2, R2, #2	2	7-16	Null	17	18	数据相关	已执行
ST R2, (R1)	3	4	20-22	18	19	数据相关	已执行
SUB R1, R1, #4	4	5-8	Null	9	20	R1已换名	已写结果
BNEZ R1, Loop	10	11-14	Null	15	21	分支失败	已写结果
ADD R5, R3, R4	16	17-20	Null	21	22	保留站冲突	已流出

寄存器状态表

	R1	R2	R3	R4	R5
ROB	ROB4	ROB2			ROB6
Busy	Y	Y			Y