1、今天收作业。

下次交作业时间：第8周的周一

2、本周六上午有实验一

3、明天上午有课堂小测。

4、本周作业：

第四章 作业　　第2，3，6题。

第4章 指令级并行

4.1 指令级并行的概念

4.2 指令的动态调度

4.3 动态分支预测技术

4.4 多指令流出技术

4.5 循环展开和指令调度

4.1 指令级并行

4.1.1 指令级并行的概念

- 几乎所有的处理机都利用流水线来使指令重叠并行执行，以达到提高性能的目的。这种指令之间存在的潜在并行性称为指令级并行。

（ILP：Instruction-Level Parallelism）

1. 流水线处理机的实际CPI

- 理想流水线的CPI加上各类停顿的时钟周期数：

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

- 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）

2. 基本程序块

- 基本程序块：一段除了入口和出口以外不包含其他分支的线性代码段。
- 程序平均每5～7条指令就会有一个分支。

3. 循环级并行：使一个循环中的不同循环体并行执行。

- 开发循环体中存在的并行性
  - 最常见、最基本
- 指令级并行研究的重点之一

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他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。

4. 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. 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。

6. 可以从两个方面来解决相关问题：

- 指令调度： 保持相关，但避免发生冲突。
- 通过代码变换(寄存器重命名)，消除相关。

7. 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。
   - 如果我们能做到保持程序的数据相关和控制相关，就能保持程序的数据流和异常行为。

- 举例说明

```
        DADDU     R2，R3，R4
        BEQZ      R2，L1
        LW        R1，0（R2）
L1 ：
```

- 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，因为它使得给定指令的数据可以有多个来源。
  - 仅仅保持数据相关性是不够的，只有再加上保持控制顺序，才能够保持程序顺序。

- 举例：

```
        DADDU    R1，R2，R3
        BEQZ     R4，L1
        DSUBU    R1，R5，R6
    L1 ：…
        OR       R7，R1，R8
```

- 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
```

4.5 循环展开和指令调度

4.5.1 循环展开和指令调度的基本方法

1. 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。
2. 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。

3. 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。
4. 本节中，假定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

例4.6　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；
```

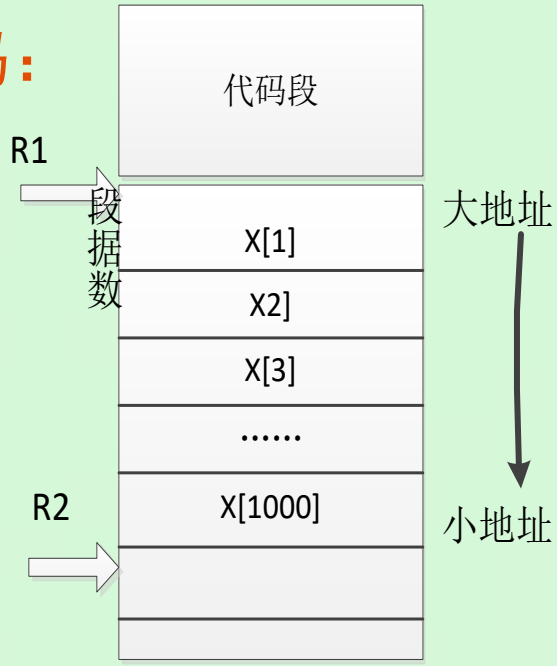
解：把该程序翻译成MIPS汇编语言代码：

假设：

①R1的初值是指向第一个元素

②8（R2）指向最后一个元素。

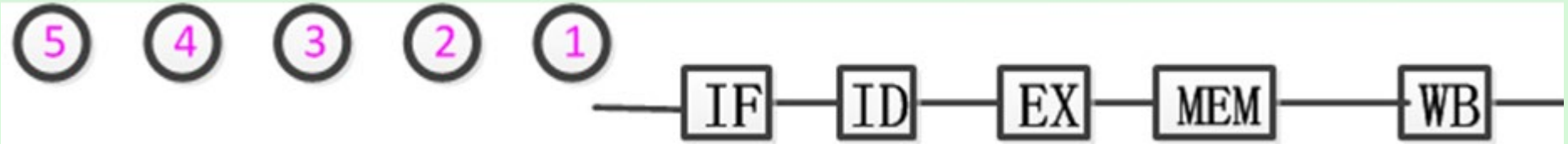
③浮点寄存器F2：用于保存常数s。



```
Loop: L.D    F0,0(R1)      ①
ADD.D        F4,F0,F2      ②
S.D   F4, 0(R1)            ③
 DADDIU  R1,R1, #-8        ④
 BNE         R1,R2,Loop    ⑤
```

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）

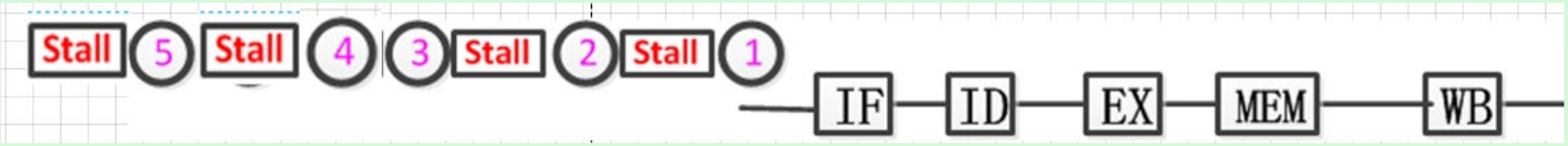


指令解释流水线

- 不进行指令调度的情况下，程序的实际执行情况：

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



1. 如果进行如下的指令调度：

- 把DADDIU指令调度到L.D指令和ADD.D指令之间的“空转”拍；
- 把S.D指令放到了分支指令的延迟槽中；
- 对存储器地址偏移量进行调整；

S.D　F4，8（R1）

指令调度前

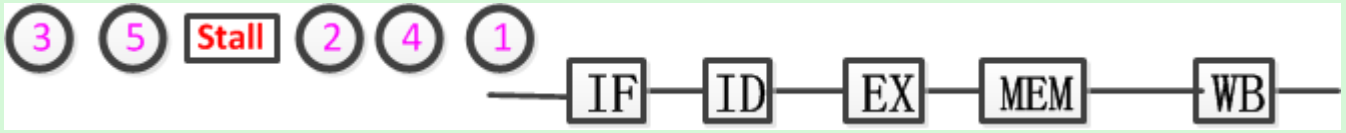
```
Loop: L.D      F0,0(R1)
      (空转)
      ADD.D    F4,F0,F2
      (空转)
      (空转)
      S.D      F4, 0(R1)
      DADDIU   R1,R1,#-8
      (空转)
      BNE      R1,R2,Loop
      (空转)
```

指令调度后

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

			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, Loop	5
	S.D	F4, 8(R1)	6

一轮循环的操作时间从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。



指令送出的顺序已经变为：1, 4, 2, 5, 3

1. 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。

例4.7：　将上述例子中的循环展开4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。

解：无需在循环体后面增加补偿代码

1. 分配寄存器（不重复使用寄存器）：

- F0、F4：用于展开后的第1个循环体
- F2：保存常数s
- F6、F8：展开后的第2个循环体
- F10、F12：第3个循环体
- F14、F16：第4个循环体

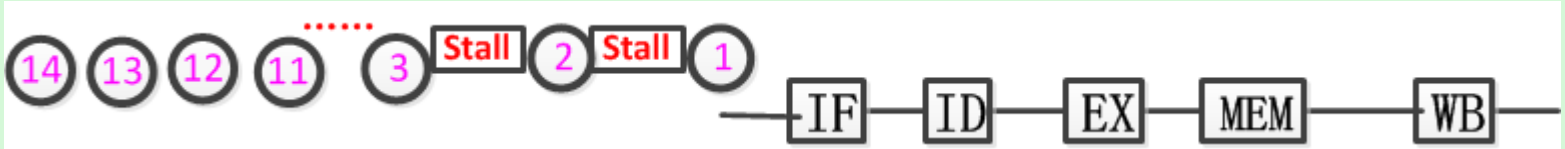
2. 相应的修改偏移地址（ 以R1为参考）：

0（R1），-8（R1），-16（R1），-24（R1）

4个循环体展开后没有调度的代码如下：

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

			指令流出时钟
	ADD.D	F12, F10, F2	⑧ 15
	（空转）		16
	（空转）		17
	S.D	F12, -16(R1)	⑨ 18
	L.D	F14, -24(R1)	⑩ 19
	（空转）		20
	ADD.D	F16, F14, F2	⑪ 21
	（空转）		22
	（空转）		23
	S.D	F16, -24(R1)	⑫ 24
	DADDIU	R1, R1, #-32	⑬ 25
	（空转）		26
	BNE	R1, R2, Loop	⑭ 27
	（空转）		28



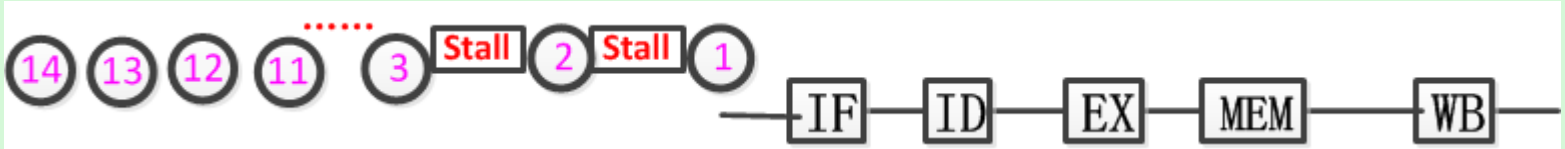
- 结果分析:
  - 这个循环每遍共使用了28个时钟周期。
  - 有4个循环体，完成4个元素的操作。平均每个元素使用28/4=7个时钟周期
  - 原始循环的每个元素需要10个时钟周期。节省的时间：从减少循环控制的开销中获得的。
  - 在整个展开后的循环中，实际指令只有14条，其他14个周期都是空转。

In conclusion, it’s not efficient.

移动时只移动LD指令（往前）和SD指令（往后）：

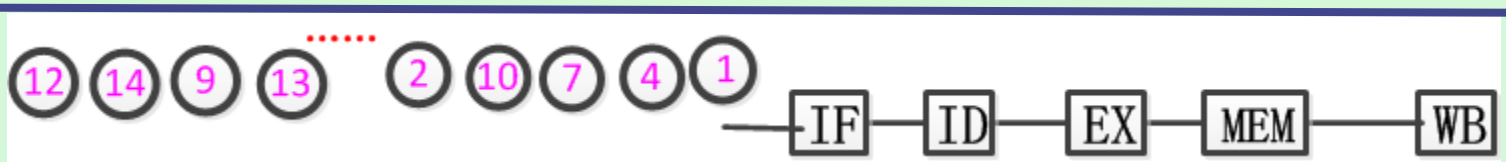
				指令流出时钟
Loop:	L.D	F0, 0(R1)	1	1
	（空转）			2
	ADD.D	F4, F0, F2	2	3
	（空转）			4
	（空转）			5
	S.D	F4, 0 (R1)	3	6
	L.D	F6, -8(R1)	4	7
	（空转）			8
	ADD.D	F8, F6, F2	5	9
	（空转）			10
	（空转）			11
	S.D	F8, -8 (R1)	6	12
	L.D	F10, -16(R1)	7	13
	（空转）			14

			指令流出时钟
ADD.D	F12, F10, F2	8	15
（空转）			16
（空转）			17
S.D	F12, -16(R1)	9	18
L.D	F14, -24(R1)	10	19
（空转）			20
ADD.D	F16, F14, F2	11	21
（空转）			22
（空转）			23
S.D	F16, -24(R1)	12	24
DADDIU	R1, R1, #-32	13	25
（空转）			26
BNE	R1, R2, Loop	14	27
（空转）			28



对指令序列进行优化调度，以减少空转周期：

			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。
  - 注意对存储器数据的相关性分析
  - 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。

4.2 指令的动态调度

- 指令的静态调度
  - 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。

- 指令的动态调度
  - 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他的流水线（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

4.2.1 动态调度的基本思想

到目前为止我们所使用流水线的最大的局限性:

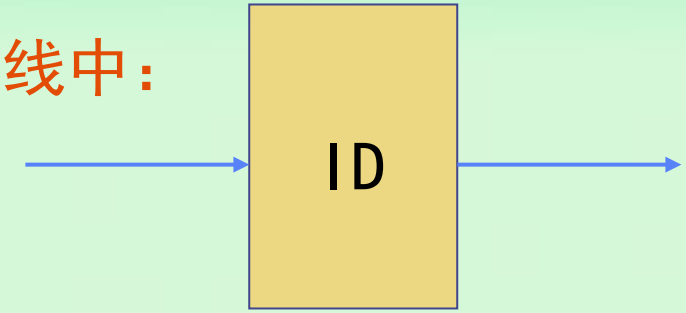
- 指令必须按序流出和执行
- 考虑下面一段代码:

```
DIV.D     F4, F0, F2
SUB.D     F10, F4, F6
ADD.D     F12, F6, F14
```

- SUB.D指令与DIV.D指令关于F4相关，导致流水线停顿。

ADD.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：



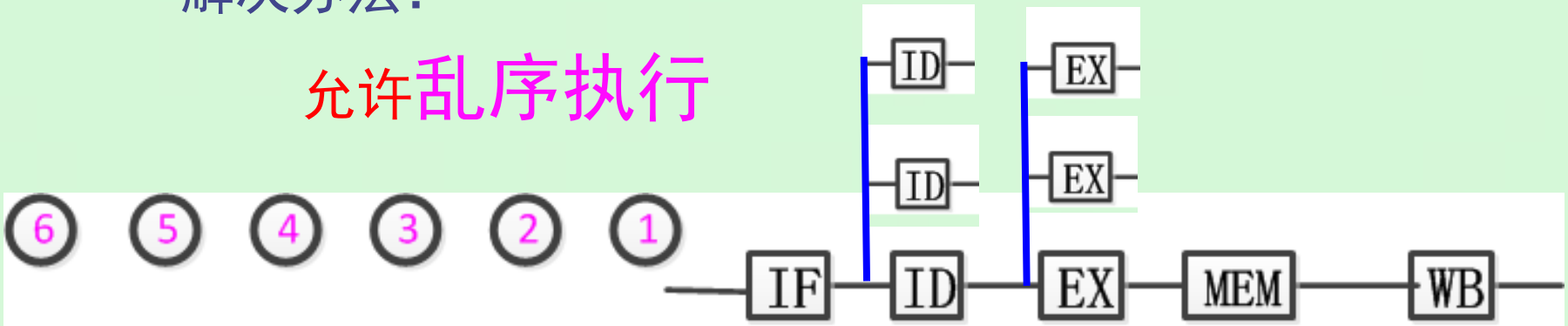
检测结构冲突

检测数据冲突

一旦一条指令受阻，其后的指令都将停顿。

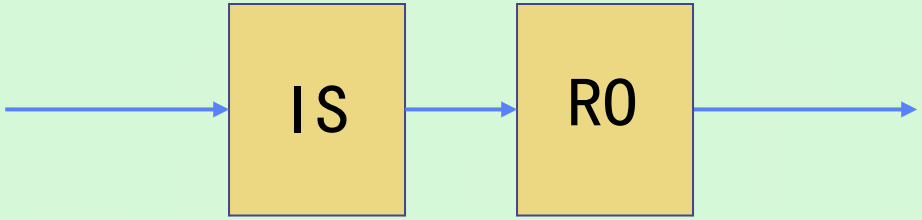
解决办法：

允许乱序执行



1. 为了允许乱序执行，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

2. 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发生WAR冲突和WAW冲突。

- 例如，考虑下面的代码

```
DIV.D   F10, F0, F2
SUB.D   F10, F4, F6
ADD.D   F6, F8, F14
```

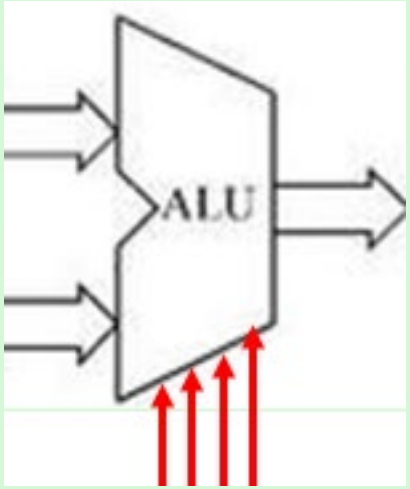
存在输出相关

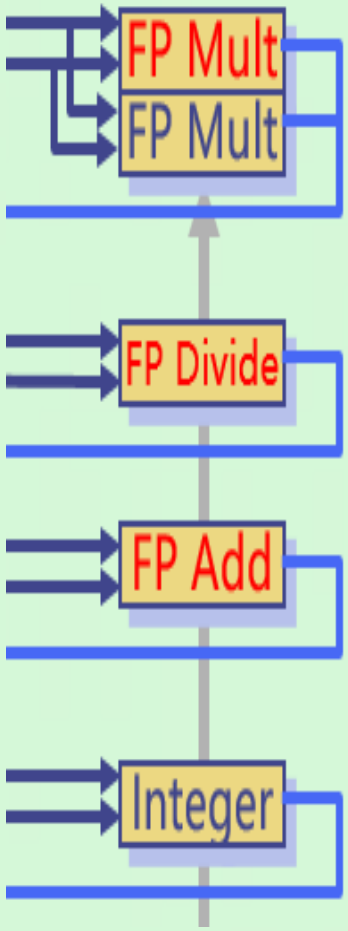
存在反相关

Tomasulo算法可以通过使用寄存器重命名来消除。

3. 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、或者流水功能部件、或者兼而有之。
- 我们假设具有多个功能部件。





4. 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）

- 动态调度要保持正确的异常行为
  - 只有那些在程序严格按程序顺序执行时会发生的异常，才能真正发生。

- 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后，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；

记分牌算法和Tomasulo算法是两种比较典型的动态调度算法。

记分牌调度算法

例：数据先读后写（WAR）相关引起的阻塞

代码序列：

```
DIVD  F0 , F2 , F4
ADDD  F10 , F0 , F8
SUBD  F8 , F8 , F14
```

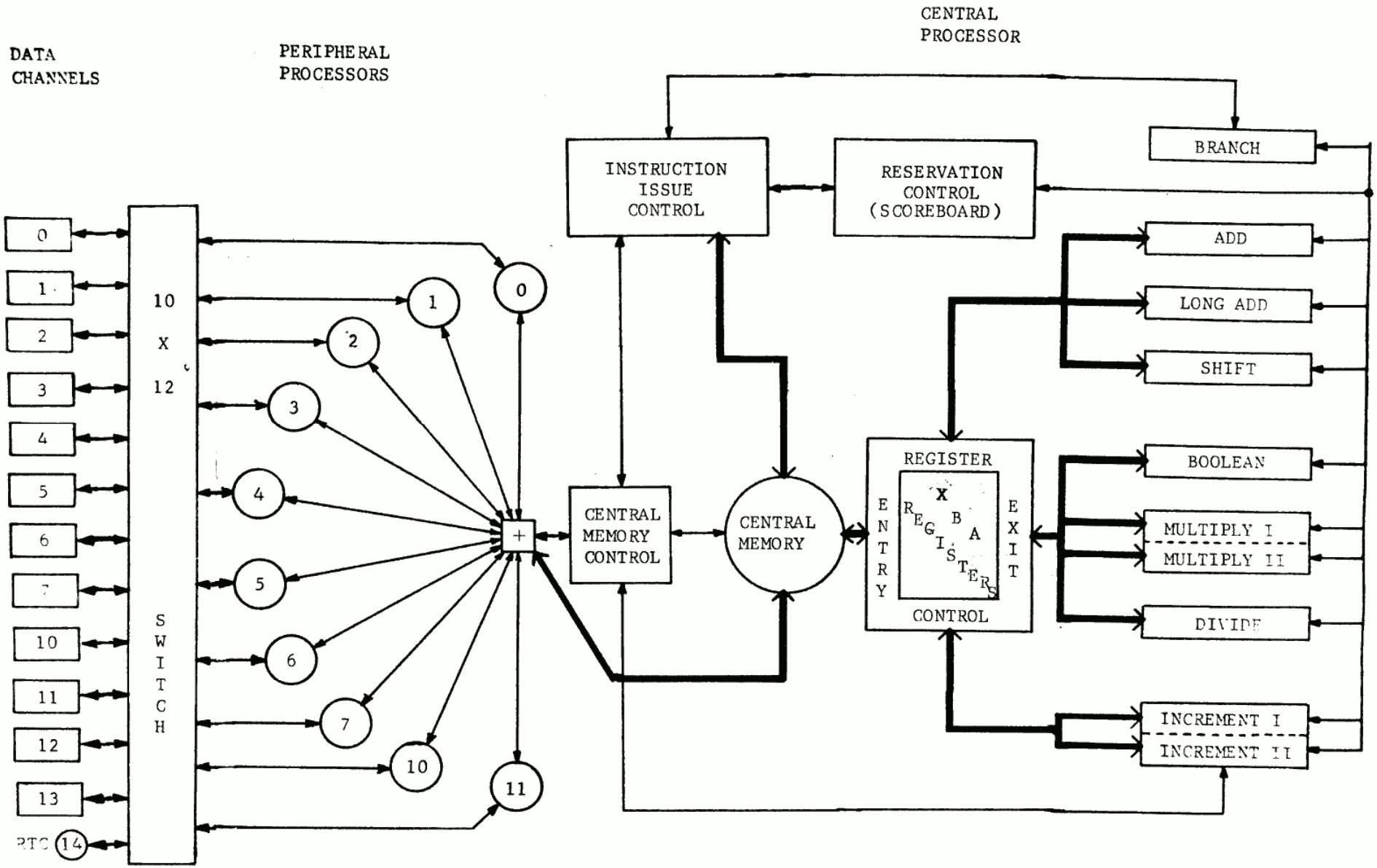
指令乱序执行时就会出现先读后写相关。

记分牌调度算法的目标：

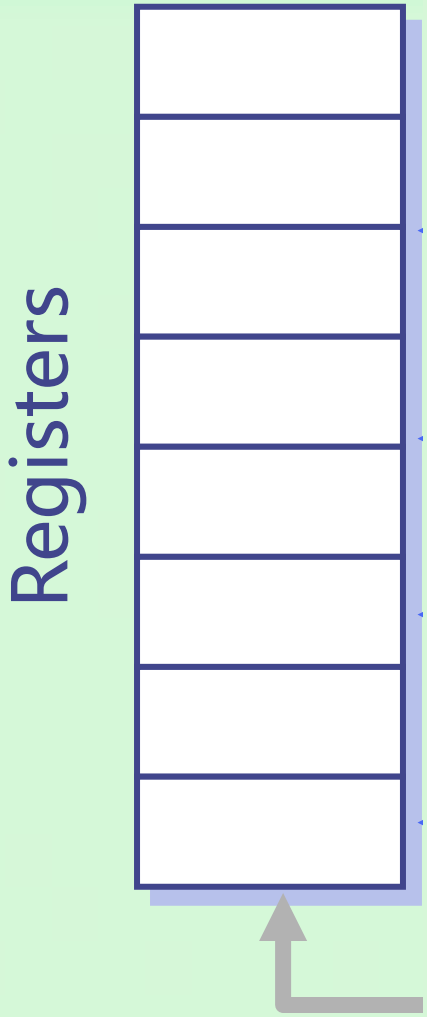
在资源充足时，尽可能早地执行没有数据阻塞的指令，达到每个时钟周期执行一条指令。

CDC 6600

CDC 6600 结构简图



记分牌体系结构



记分牌算法需要记录的信息

SCOREBOARD算法举例

- 指令状态表 （Instruction status）
- 功能部件状态表（Functional unit status）
- 寄存器状态表（Register result status）

①	LD	F6	34（R2）	
②	LD	F2	45（R3）	
③	MULTD	F0	F2	F4
④	SUBD	F8	F6	F2
⑤	DIVD	F10	F0	F6
⑥	ADDD	F6	F8	F2

ADD: 2 cycles

Multd:10 cycles

Divd: 40 cycles

记分牌算法需要记录的信息

1、指令状态表　（Instruction status）

- which of 4 steps the instruction is in

Instruction status

	Instruction	j	k	Issue	Read operands	Execution complete	Write Result
①	LD	F6	34+ R2				
②	LD	F2	45+ R3				
③	MUL	F0	F2	F4			
④	SUB	F8	F6	F2			
⑤	DIVD	F10	F0	F6			
⑥	ADD	F6	F8	F2			

记分牌算法需要记录的信息

2、功能部件状态表（Functional unit status）

Functional unit status—Indicates the state of the functional unit (FU).

- Busy — Indicates whether the unit is busy or not
- Op — Operation to perform in the unit (e.g., + or –)
- Fi — Destination register
- Fj, Fk — Source-register numbers
- Qj, Qk— Functional units producing source registers Fj, Fk
- Rj, Rk— Flags indicating when Fj, Fk are ready

Functional unit status

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU for j Qj	FU for k Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

记分牌算法需要记录的信息

3、寄存器状态表（Register result status）

Register result status

- Indicates which functional unit will write each register, if one exists. Blank when no pending instructions will write that register

Register result status

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

4.2.2 Tomasulo算法

Tomasulo算法基本思想

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。
   - 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

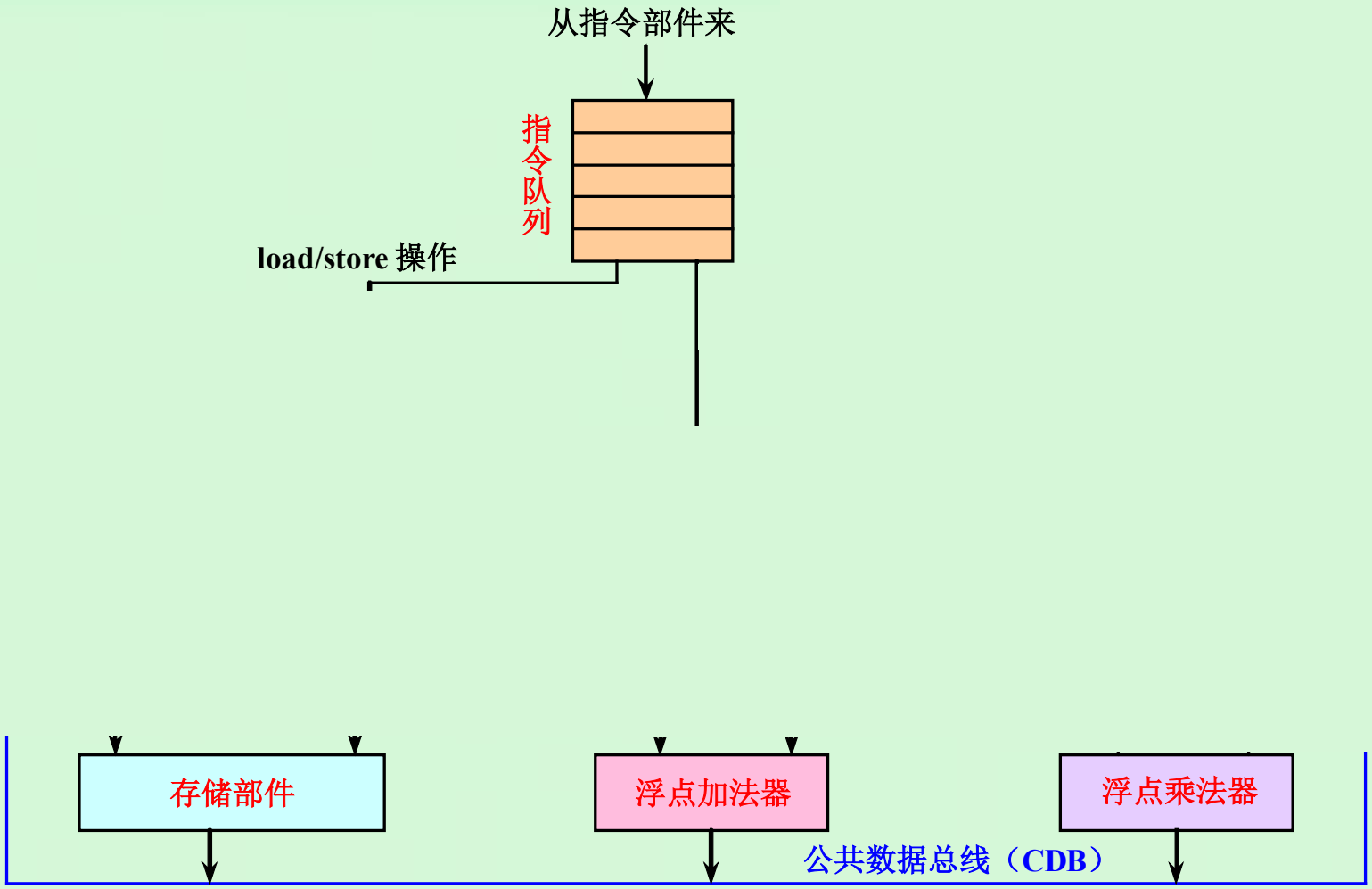
- 消除名相关
  - 引入两个临时寄存器S和T
  - 把这段代码改写为：

```
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
```

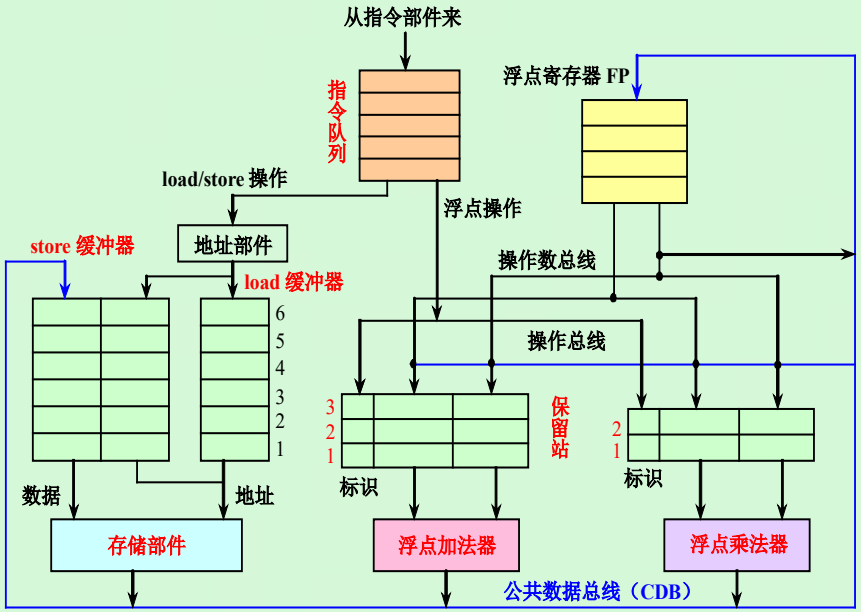
两个F6都换名为S

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
load/store 操作
存储部件
浮点加法器
浮点乘法器
公共数据总线（CDB）

4.2 指令的动态调度


从指令部件来
指令队列
浮点寄存器 FP
load/store 操作
浮点操作
store 缓冲器
地址部件
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）

人民网江西
jx.people.cn

面试

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. 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。

7、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）

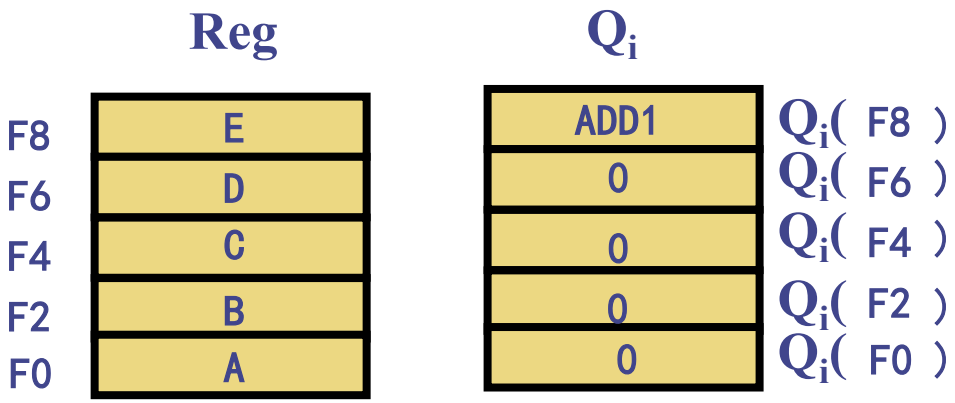
- 执行
  - 当两个操作数都就绪后，本保留站就用相应的功能部件开始执行指令规定的操作。
  - load和store指令的执行需要两个步骤：
    - 计算有效地址（要等到基地址寄存器就绪）
    - 把有效地址放入load或store缓冲器

- 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。

名称	保留站						
	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	no						
Load2	no						
Add1	no						
Add2	no						
Add3	no						

8. 每个保留站有以下几个字段：

- 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。

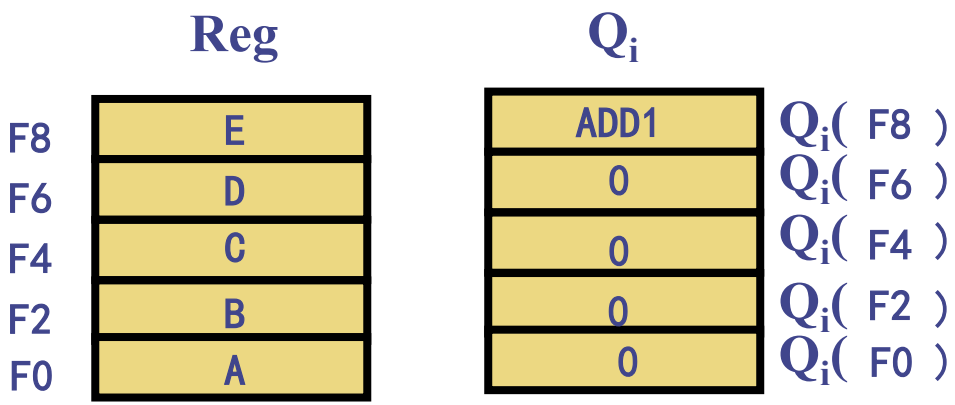


此外：

- 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。

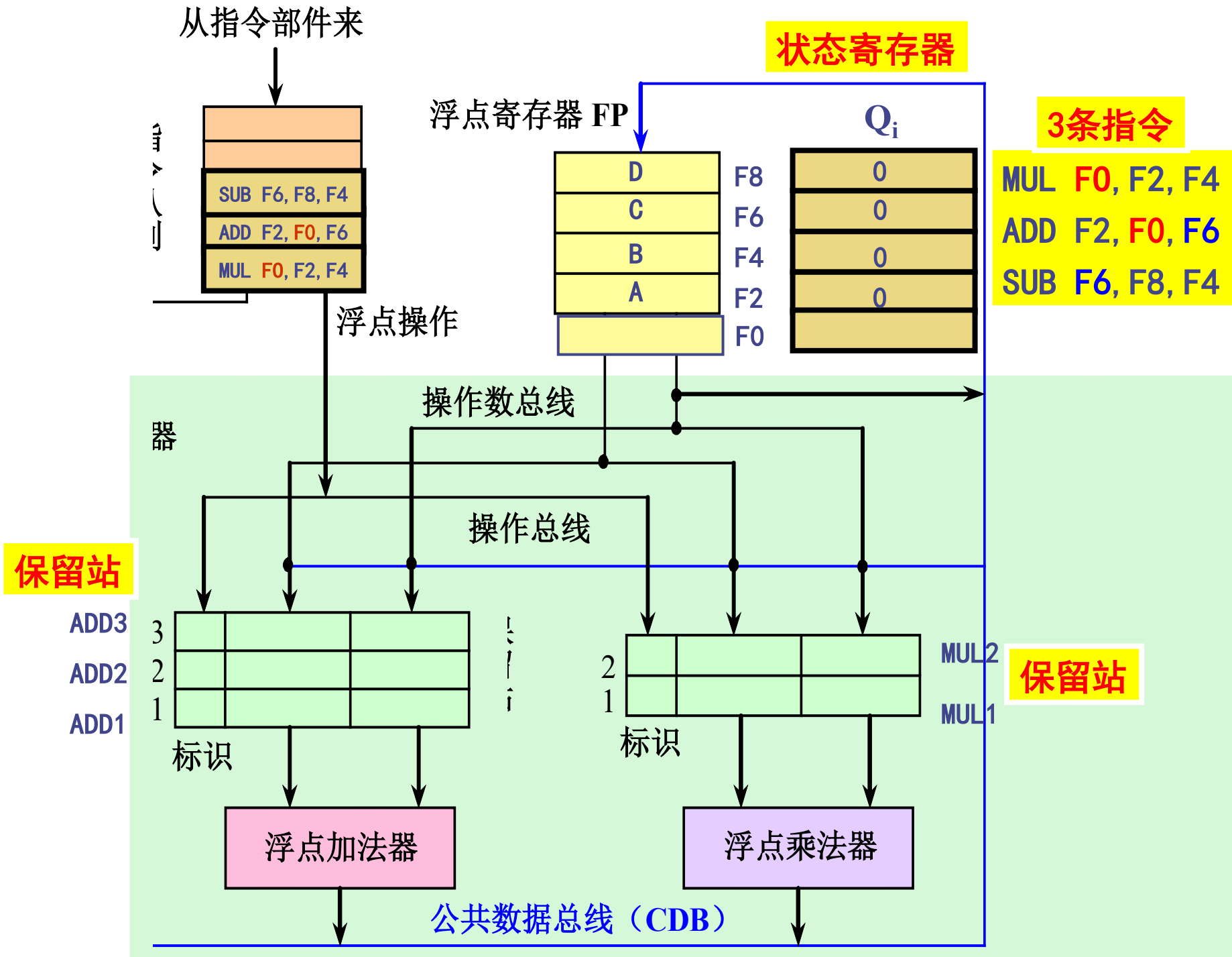
	寄存器状态表							
	F0	F2	F4	F6	F8	F10	…	F30
Qi	0	0	0	0	ADD1		…	

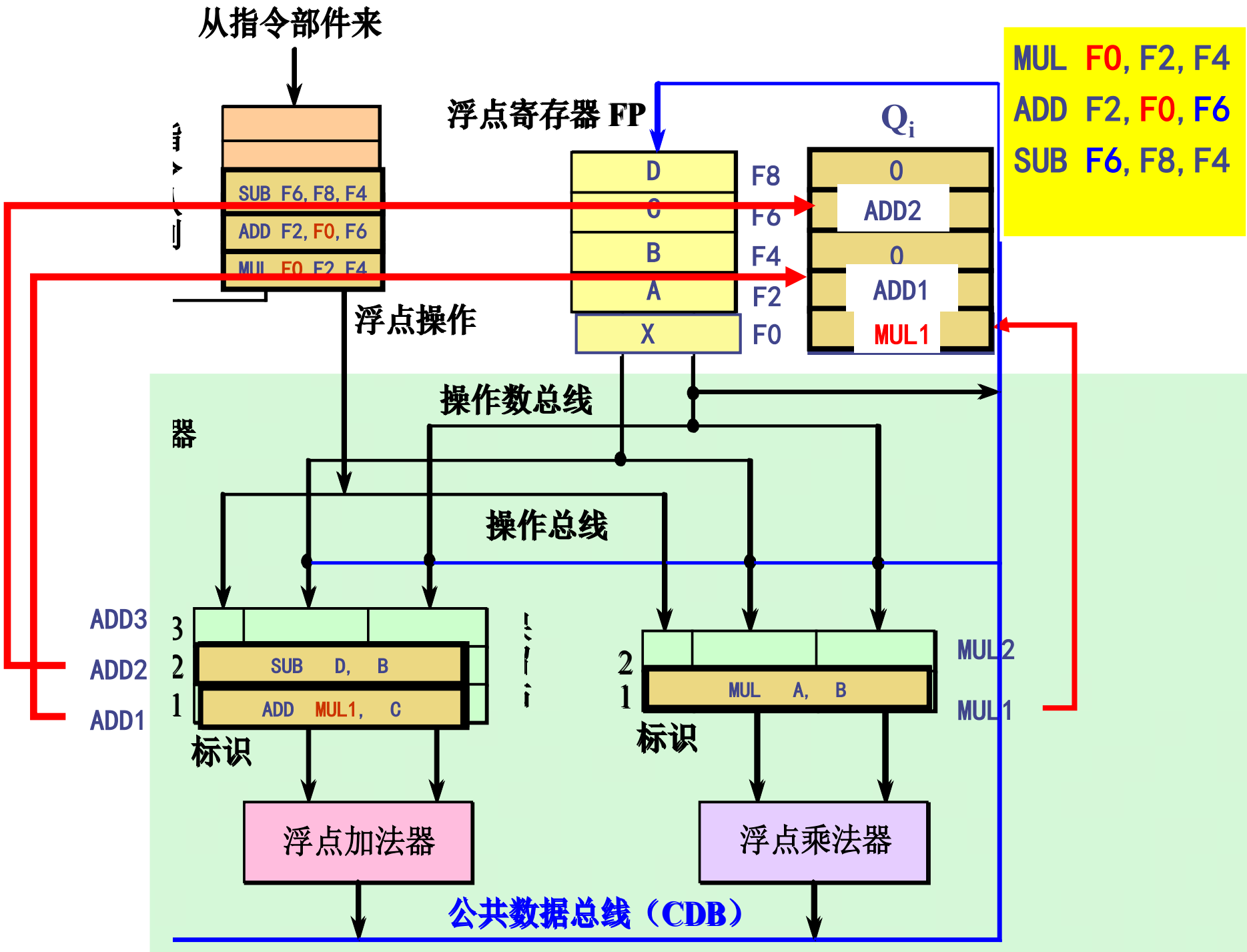
Q_i(F0) Q_i(F2) Q_i(F4) Q_i(F6) Q_i(F8)

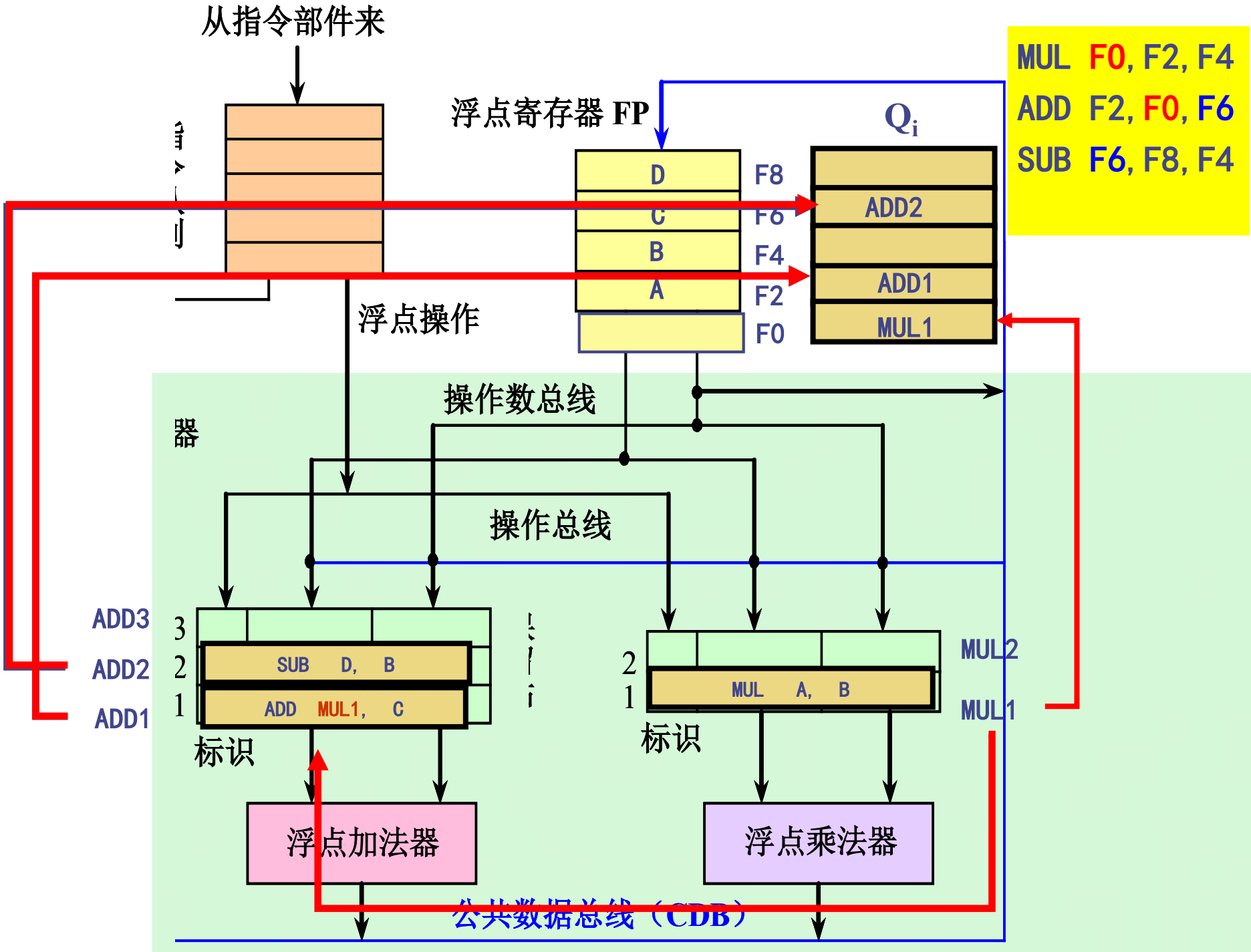


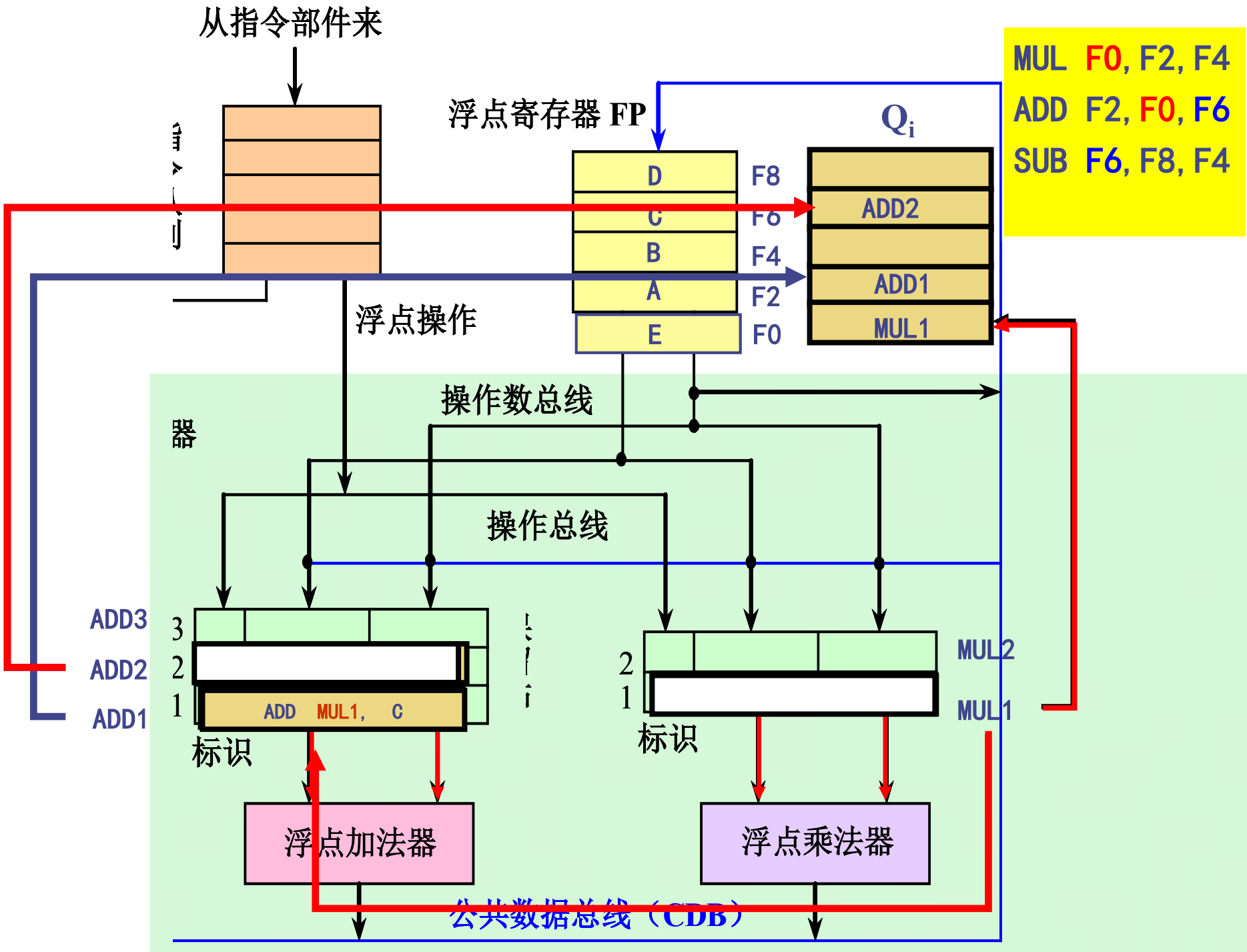
Tomasulo算法举例

- 指令状态表
- 保留站
- 寄存器状态表


从指令部件来
SUB F6, F8, F4
ADD F2, F0, F6
MUL F0, F2, F4
浮点操作
状态寄存器
浮点寄存器 FP
D
C
B
A
F8
F6
F4
F2
F0
Qi
0
0
0
0
3条指令
MUL F0, F2, F4
ADD F2, F0, F6
SUB F6, F8, F4
操作数总线
操作总线
保留站
ADD3
ADD2
ADD1
3
2
1
标识
2
1
标识
MUL2
MUL1
保留站
浮点加法器
浮点乘法器
公共数据总线（CDB）


从指令部件来
SUB F6, F8, F4
ADD F2, F0, F6
MUL F0, F2, F4
浮点寄存器 FP
D
C
B
A
X
F8
F6
F4
F2
F0
Q_i
0
ADD2
0
ADD1
MUL1
MUL F0, F2, F4
ADD F2, F0, F6
SUB F6, F8, F4
浮点操作
操作数总线
操作总线
ADD3
ADD2
ADD1
3
2
1
SUB D, B
ADD MUL1, C
标识
2
1
MUL A, B
标识
MUL2
MUL1
浮点加法器
浮点乘法器
公共数据总线（CDB）


从指令部件来
浮点寄存器 FP
Q_i
MUL F0, F2, F4
ADD F2, F0, F6
SUB F6, F8, F4
D
C
B
A
F8
F6
F4
F2
F0
ADD2
ADD1
MUL1
浮点操作
操作数总线
器
操作总线
ADD3
ADD2
ADD1
3
2
1
SUB D, B
ADD MUL1, C
标识
2
1
MUL A, B
MUL2
MUL1
标识
浮点加法器
浮点乘法器
公共数据总线（CDB）


从指令部件来
浮点寄存器 FP
Q_i
MUL F0, F2, F4
ADD F2, F0, F6
SUB F6, F8, F4
D
C
B
A
E
F8
F6
F4
F2
F0
ADD2
ADD1
MUL1
浮点操作
操作数总线
器
操作总线
ADD3
ADD2
ADD1
3
2
1
ADD MUL1, C
标识
2
1
MUL2
MUL1
标识
浮点加法器
浮点乘法器
公共数据总线（CDB）

例4.1　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。

① L.D　F6,34（R2）

② L.D　F2,45（R3）

③ MUL.D　F0,F2,F4

④ SUB.D　F8,F2,F6

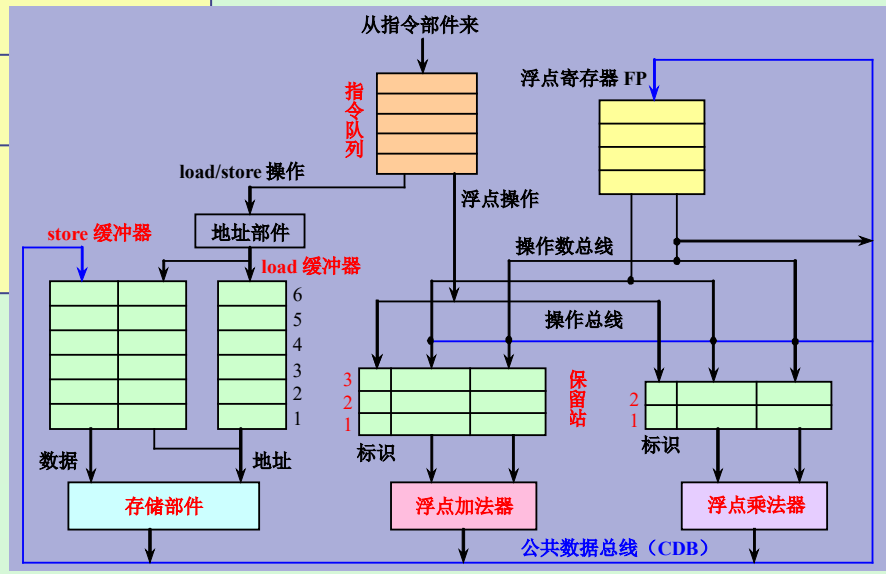
⑤ DIV.D　F10,F0,F6

⑥ ADD.D　F6,F8,F2

Tomasulo算法的核心
- 指令状态表
- 保留站
- 寄存器状态表

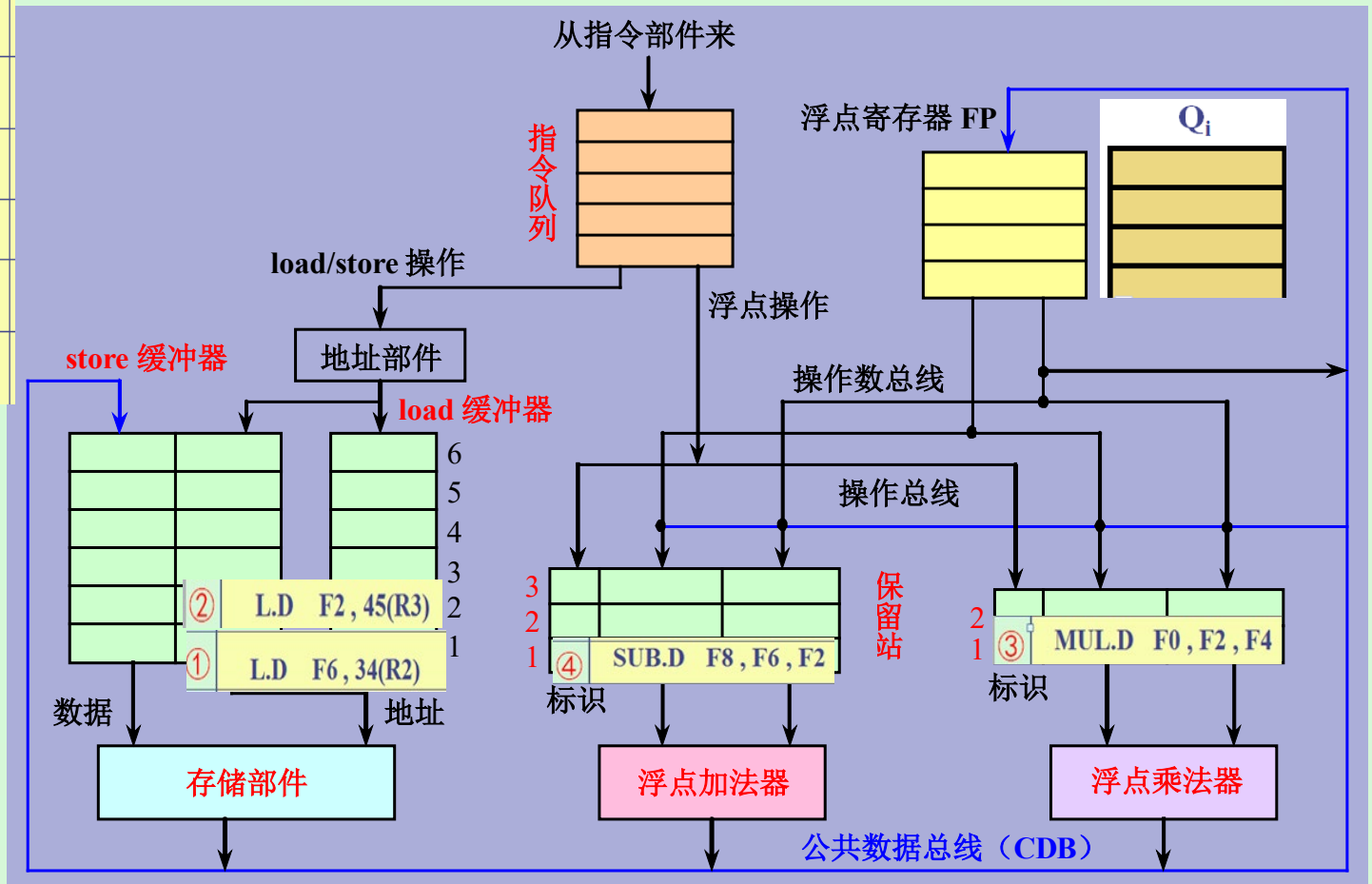
当采用Tomasulo算法时，在上述给定的时刻，指令状态表、保留站、load缓冲器以及寄存器状态表中的内容分别介绍如下。

	指　　令	指令状态表		
		流出	执行	写结果
①	L.D　F6 , 34(R2)	√	√	√
②	L.D　F2 , 45(R3)	√	√	
③	MUL.D　F0 , F2 , F4	√		
④	SUB.D　F8 , F6 , F2			
⑤	DIV.D　F10 , F0 , F6			
⑥	ADD.D　F6 , F8 , F2			



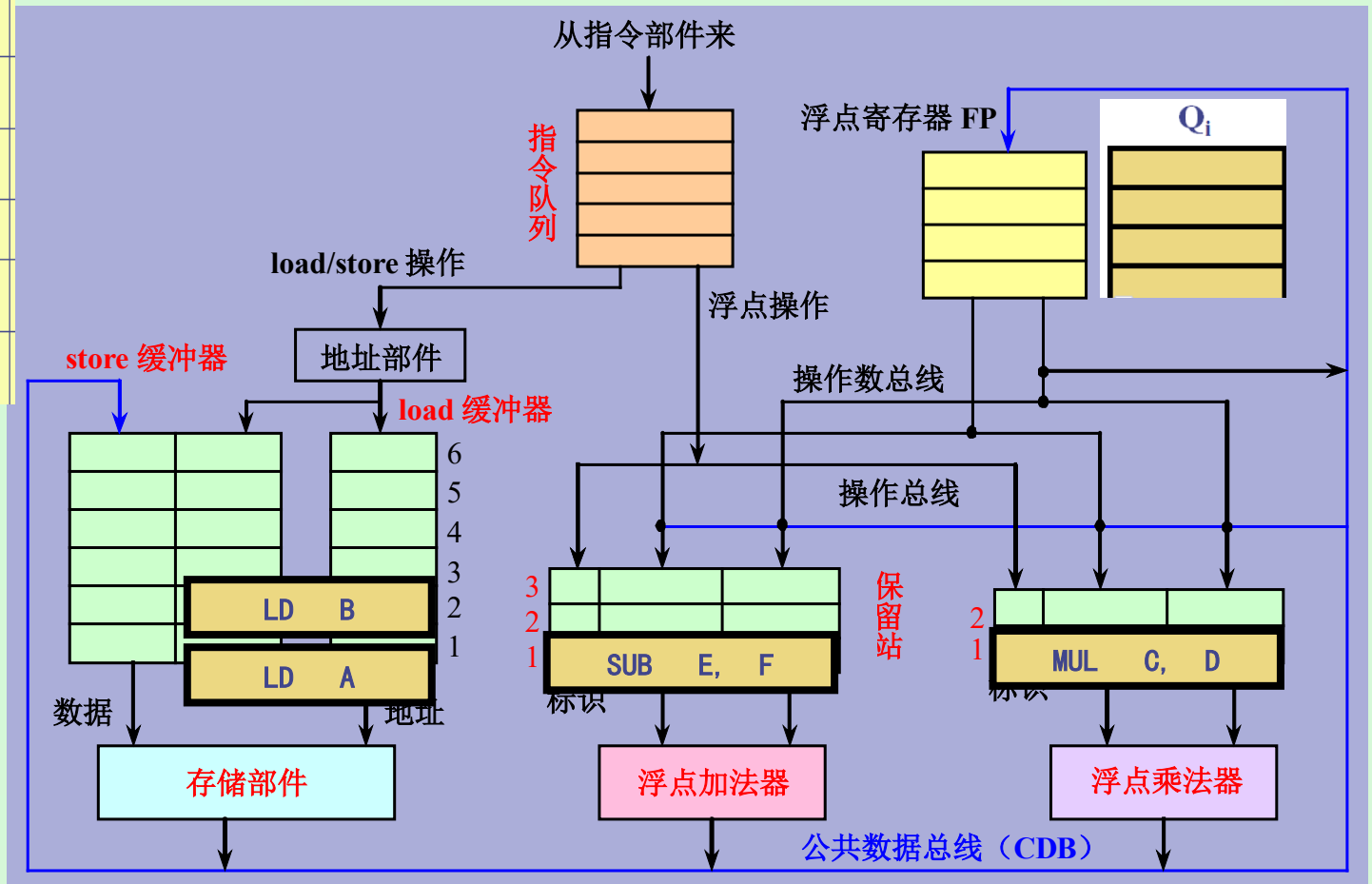
当采用Tomasulo算法时，在上述给定的时刻，指令状态表、保留站、load缓冲器以及寄存器状态表中的内容分别介绍如下。

	指　　令
①	L.D　F6 , 34(R2)
②	L.D　F2 , 45(R3)
③	MUL.D　F0 , F2 , F4
④	SUB.D　F8 , F6 , F2
⑤	DIV.D　F10 , F0 , F6
⑥	ADD.D　F6 , F8 , F2



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

	指　　令
①	L.D　F6 , 34(R2)
②	L.D　F2 , 45(R3)
③	MUL.D　F0 , F2 , F4
④	SUB.D　F8 , F6 , F2
⑤	DIV.D　F10 , F0 , F6
⑥	ADD.D　F6 , F8 , F2

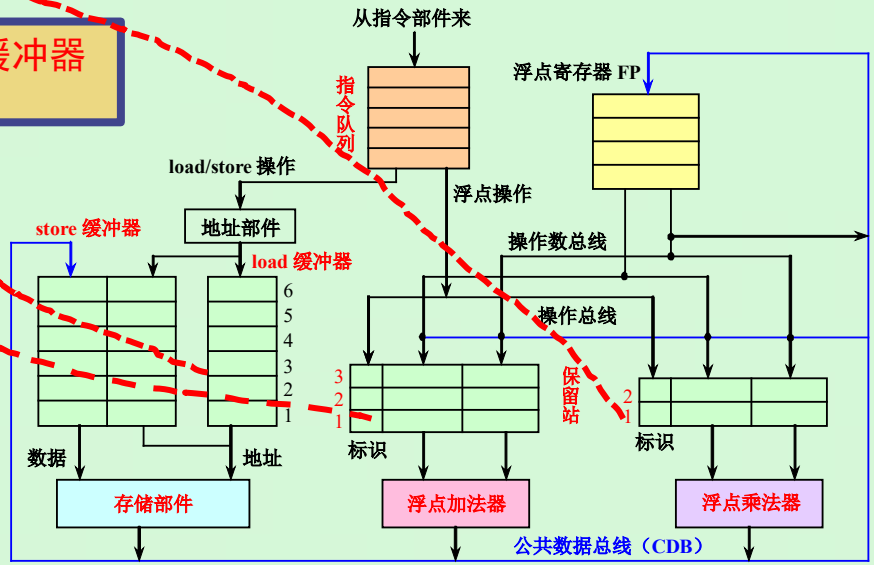


名称	保留站						
	Busy		Vj	Vk	Qj	Qk	
Load1	no						
Load2	yes						
Add1	yes			Mem[34+Regs[R2]]	Load2		
Add2	no						
Add3	no						
Mult1	yes			Reg[F4]	Load2		
Mult2	no						

共有6个Load缓冲器

共有三个加法保留站

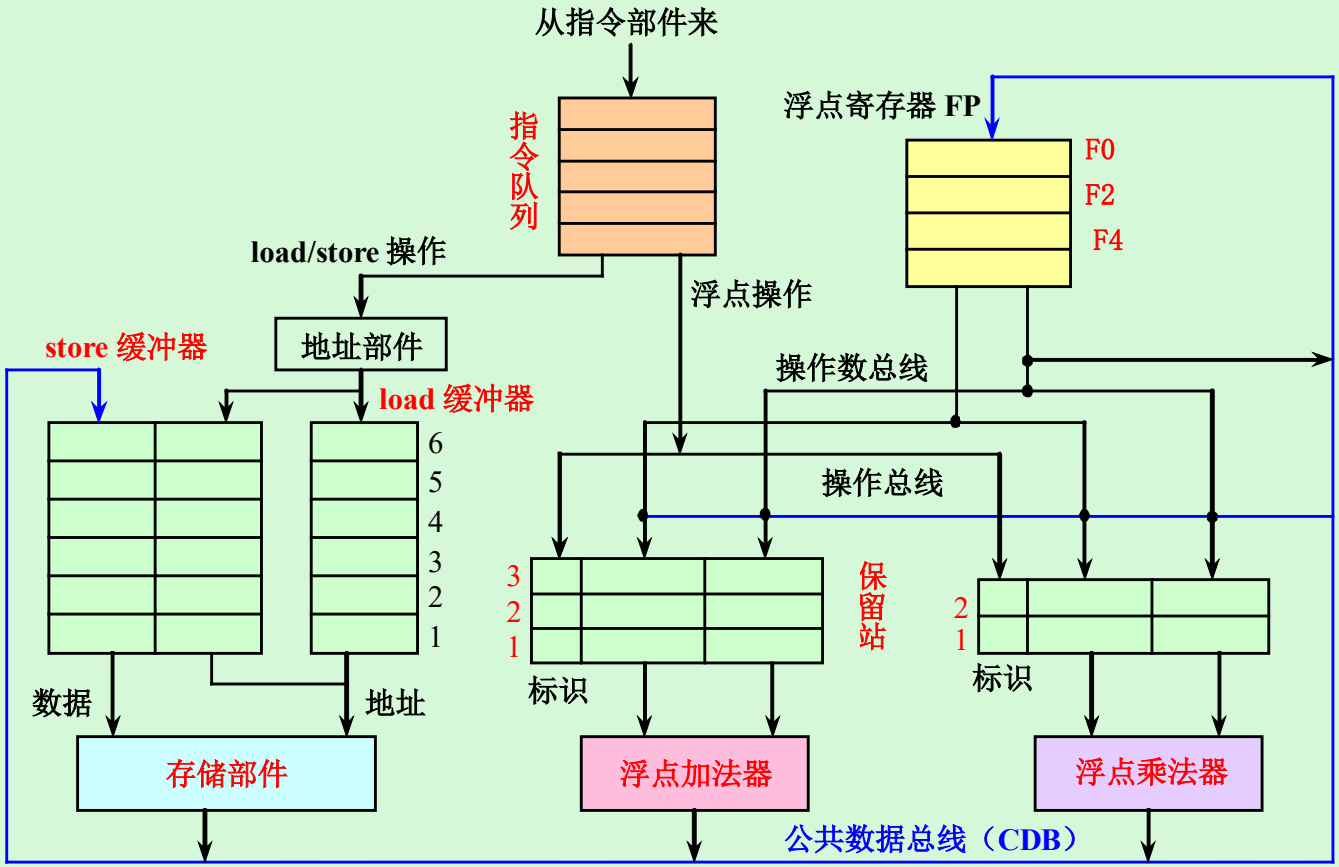
共有两个乘法保留站



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。

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



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，并且操作数一旦就绪就将之放入保留站。

例4.2 对于例4.1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

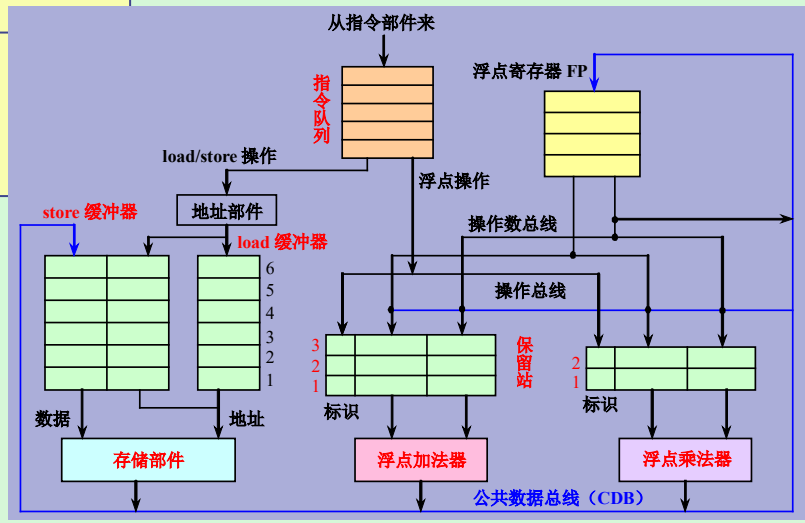
给出MUL.D指令准备写结果时各状态表的内容。

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
①	L.D	F6 , 34(R2)	√	√	√
②	L.D	F2 , 45(R3)	√	√	√
③	MUL.D	F0 , F2 , F4	√	√	
④	SUB.D	F8 , F6 , F2	√	√	√
⑤	DIV.D	F10, F0, F6	√		
⑥	ADD.D	F6 , F8 , F2	√	√	√

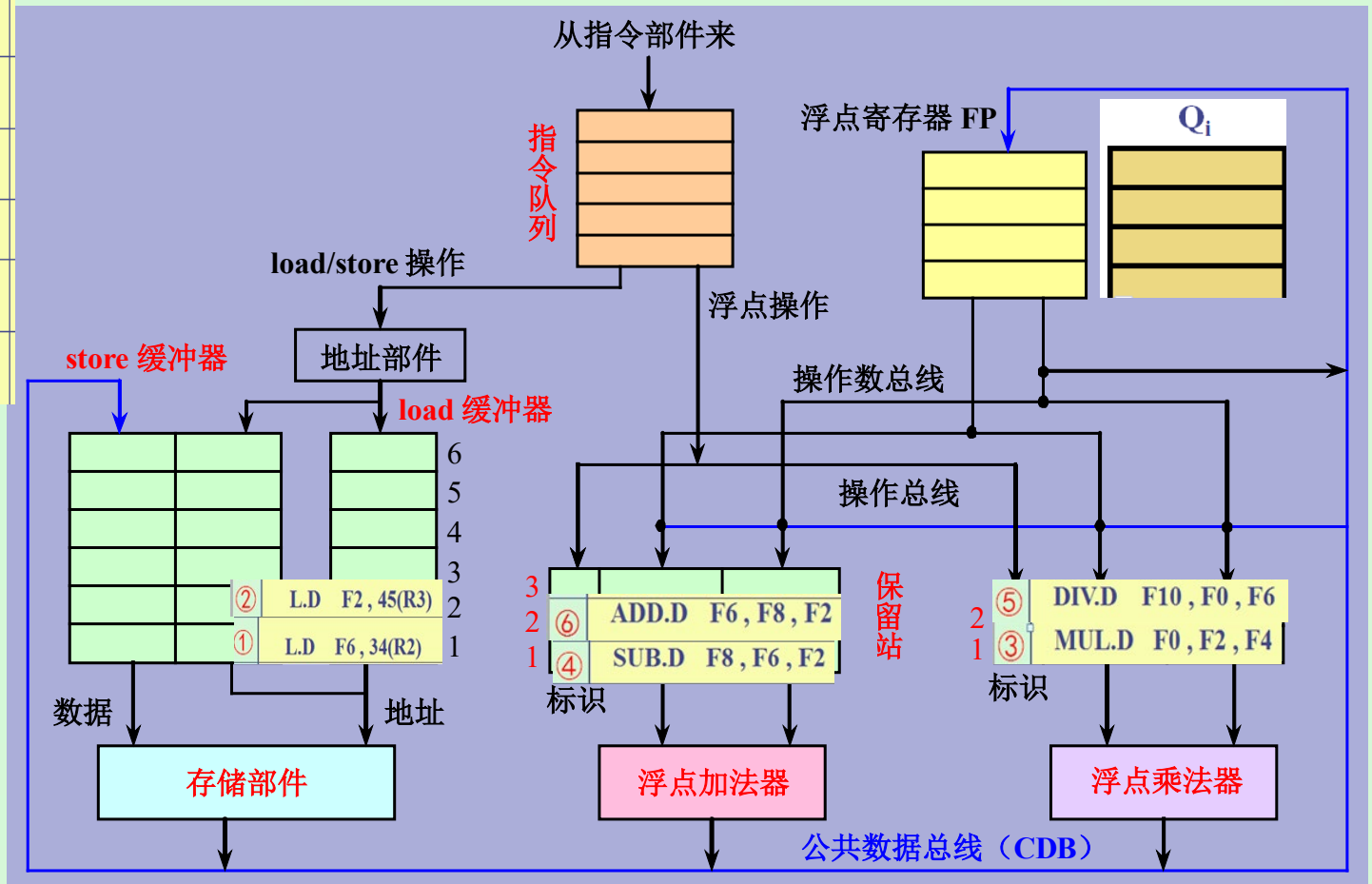
当采用Tomasulo算法时，在上述给定的时刻，指令状态表、保留站、load缓冲器以及寄存器状态表中的内容分别介绍如下。

	指　　令	指令状态表		
		流出	执行	写结果
①	L.D　F6 , 34(R2)	√	√	√
②	L.D　F2 , 45(R3)	√	√	√
③	MUL.D　F0 , F2 , F4	√	√	
④	SUB.D　F8 , F6 , F2	√	√	√
⑤	DIV.D　F10 , F0 , F6	√		
⑥	ADD.D　F6 , F8 , F2	√	√	√



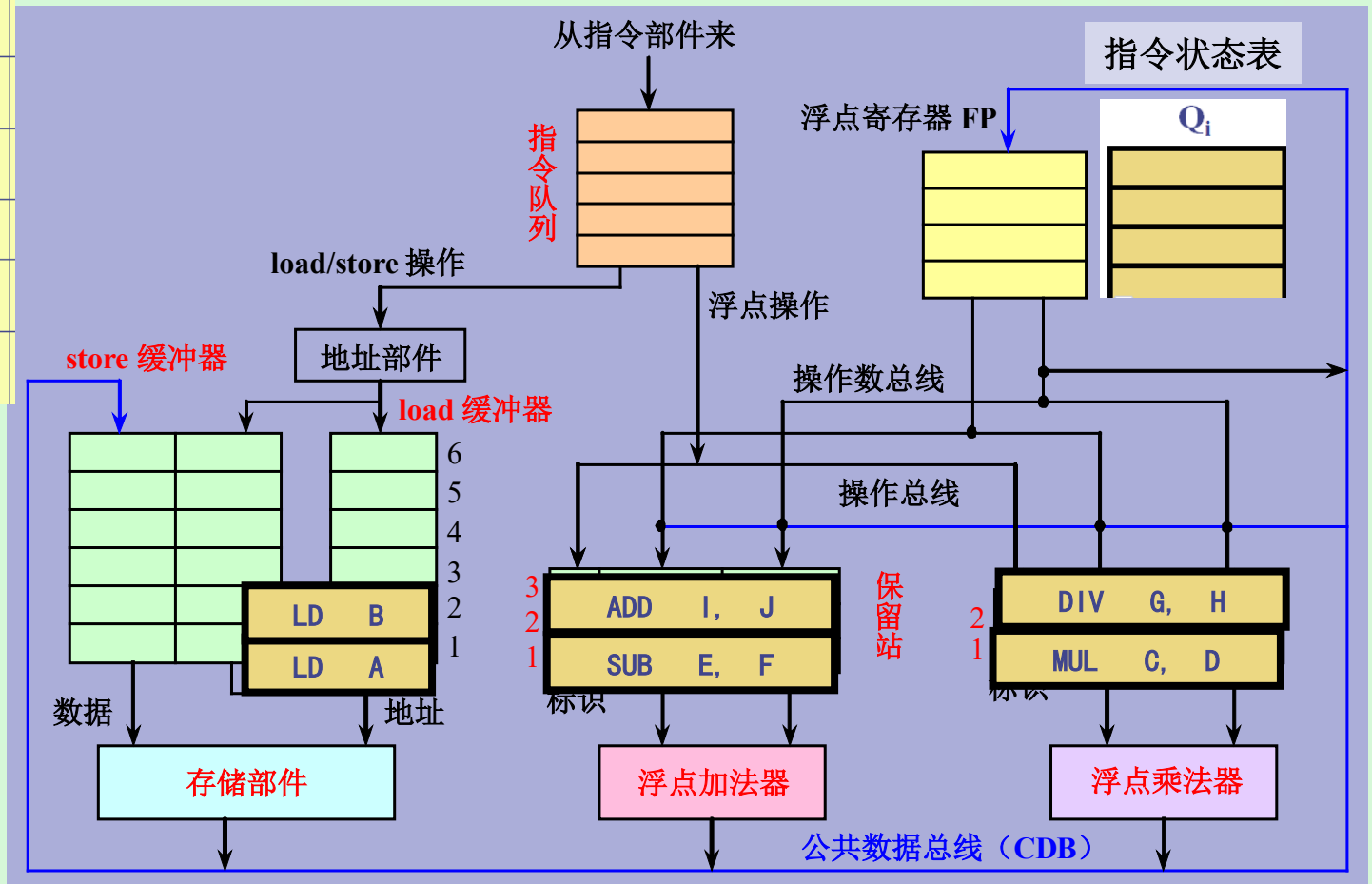
当采用Tomasulo算法时，在上述给定的时刻，指令状态表、保留站、load缓冲器以及寄存器状态表中的内容分别介绍如下。

	指　　令
①	L.D　F6 , 34(R2)
②	L.D　F2 , 45(R3)
③	MUL.D　F0 , F2 , F4
④	SUB.D　F8 , F6 , F2
⑤	DIV.D　F10 , F0 , F6
⑥	ADD.D　F6 , F8 , F2



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

	指　　令
①	L.D　F6 , 34(R2)
②	L.D　F2 , 45(R3)
③	MUL.D　F0 , F2 , F4
④	SUB.D　F8 , F6 , F2
⑤	DIV.D　F10 , F0 , F6
⑥	ADD.D　F6 , F8 , F2

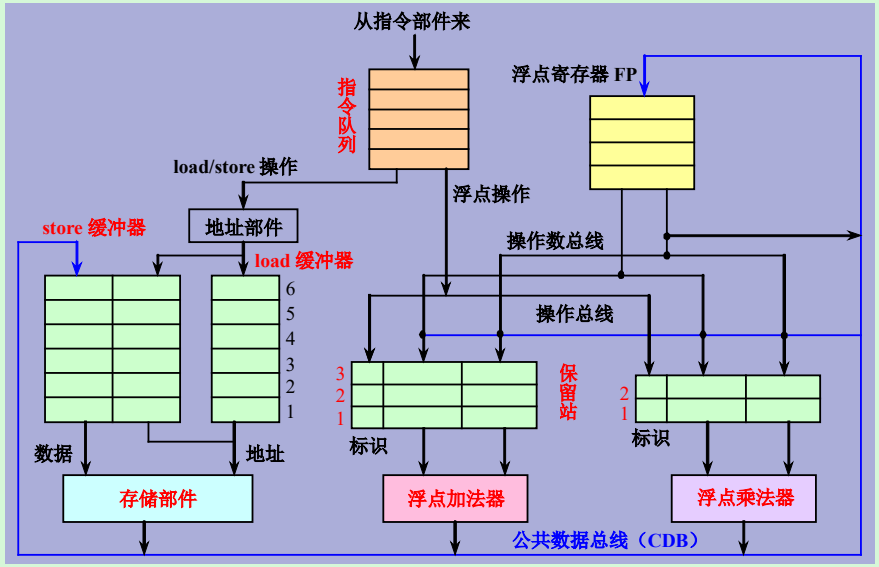


名称	保留站						
	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	no						
Load2	no						
Add1	no						
Add2	no						
Add3	no						
Mult1	yes	Mul	Mem[45+Regs[R3]]	Reg[F4]			
Mult2	yes	DIV		Mem[34+Regs[R2]]	Mult1		

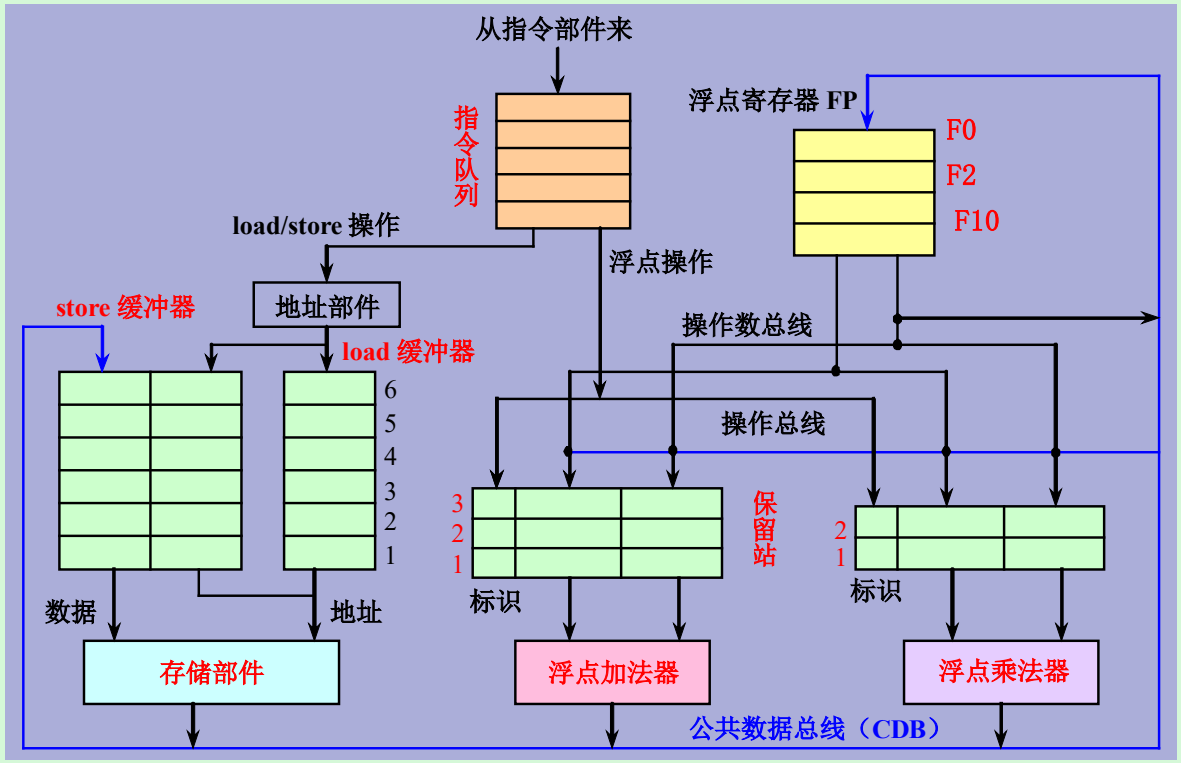
共有6个Load缓冲器

共有两个乘法保留站

共有三个加法保留站



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



Tomasulo具体算法描述

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

- I类指令
  - 包括所有的load和store指令、立即数指令、，分支指令、寄存器跳转指令、寄存器链接跳转指令。
  - 立即数字段为16位，用于提供立即数或偏移量。

6	5	5	16
操作码	rs	rt	立即数（immediate）
0 5	6 10	11 15	16 31

- R类指令
  - 包括ALU指令、专用寄存器读/写指令、move指令等。
  - ALU指令

    Regs[rd]← Regs[rs] funct Regs[rt]

    func为具体的运算操作编码

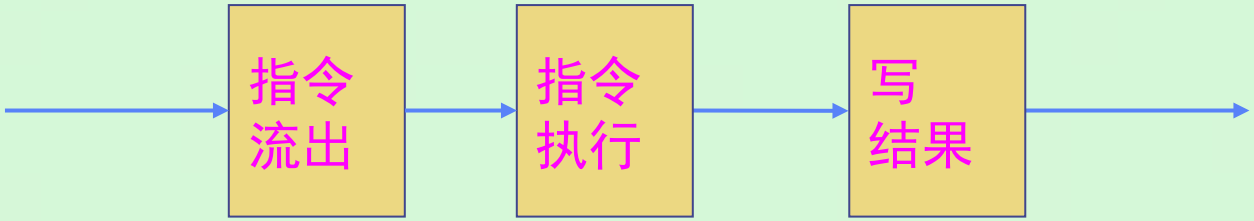
6	5	5	5	5	6
操作码	rs	rt	rd	sham	func
0 5	6 10	11 15	16 20	21 25	26 31

J类指令

- 包括跳转指令、跳转并链接指令、自陷指令、异常返回指令。
- 在这类指令中，指令字的低26位是偏移量，它与PC值相加形成跳转的地址。

Tomasulo具体算法描述

1. tomasulo算法中指令处理的3个阶段分别是：



2.什么情况下可以指令流出？

答：当有空闲的保留站，就可以填充空闲的保留站的V_J,V_K,Q_J,Q_K

根据状态寄存器Q_I的值，就知道填充的是保留站VJ,VK,QJ,QK中的哪两个。

如果Qi[rs] = 0,说明**第一操作数是否就绪；**

如果Qi[rt] = 0,说明**第二操作数是否就绪。**

保留站：Reservation Station：RS

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].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 imm rs

k j

➢ load和store指令

进入条件：缓冲器有空闲单元（设为r）

操作和状态表内容修改：

```
if （Qi[rs] = 0）                    // 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。
```

L.D F2，45（R3）

rt imm rs

k j

RS[r].Busy ← yes；　　// 置当前缓冲器单元为“忙”

RS[r].A ← Imm；　　// 把符号位扩展后的偏移量放入

// 当前缓冲器单元的A

对于load指令：

Qi[rt] ← r；　　// 把当前缓冲器单元的编号r放入

// load指令的目标寄存器rt所对应的寄存器

// 状态表项，以便rt将来接收所取的数据。

S.D F3, 40 (R4)

rt imm rs

k j

对于store指令：

if （Qi[rt] = 0） // 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

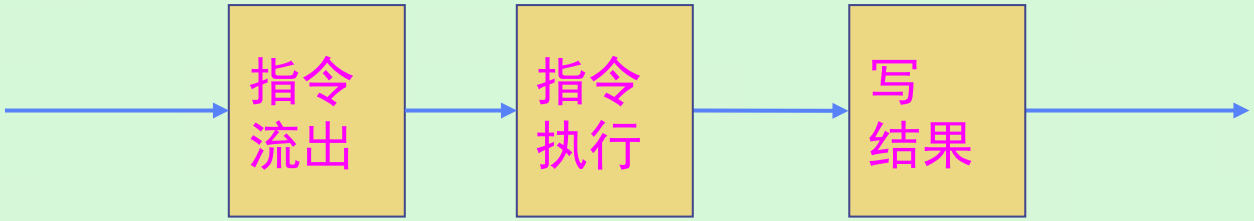
// 中的数据就绪。

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。

Tomasulo具体算法描述

1. tomasulo算法中指令处理的3个阶段分别是：



2.什么情况下可以指令执行？

答：保留站V_J,V_K非0 或者Q_J=0,Q_K=0。

说明两个**操作数已经准备就绪了；**

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：

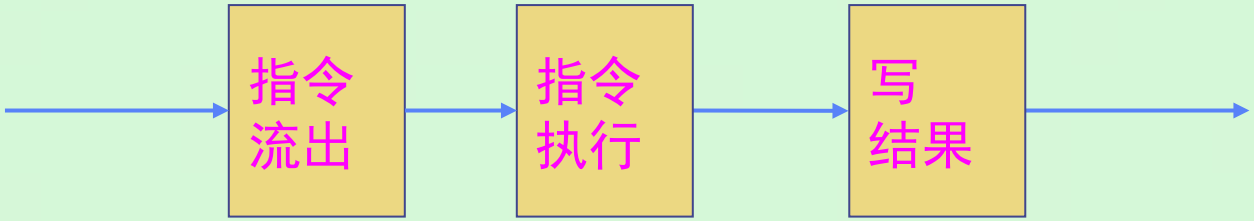
RS[r].A ← RS[r].Vj + RS[r].A;　　//计算有效地址

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

Tomasulo具体算法描述

1. tomasulo算法中指令处理的3个阶段分别是：



2.什么情况下可以写结果？

答：保留站r执行结束，且CDB就绪；

结果写入对应的寄存器或者保留站中。

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。

4.3 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，就需要预测分支的结果；
   - Amdahl定律告诉我们，机器的CPI越小，控制停顿的相对影响就更大。

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未来的行为。
   - 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

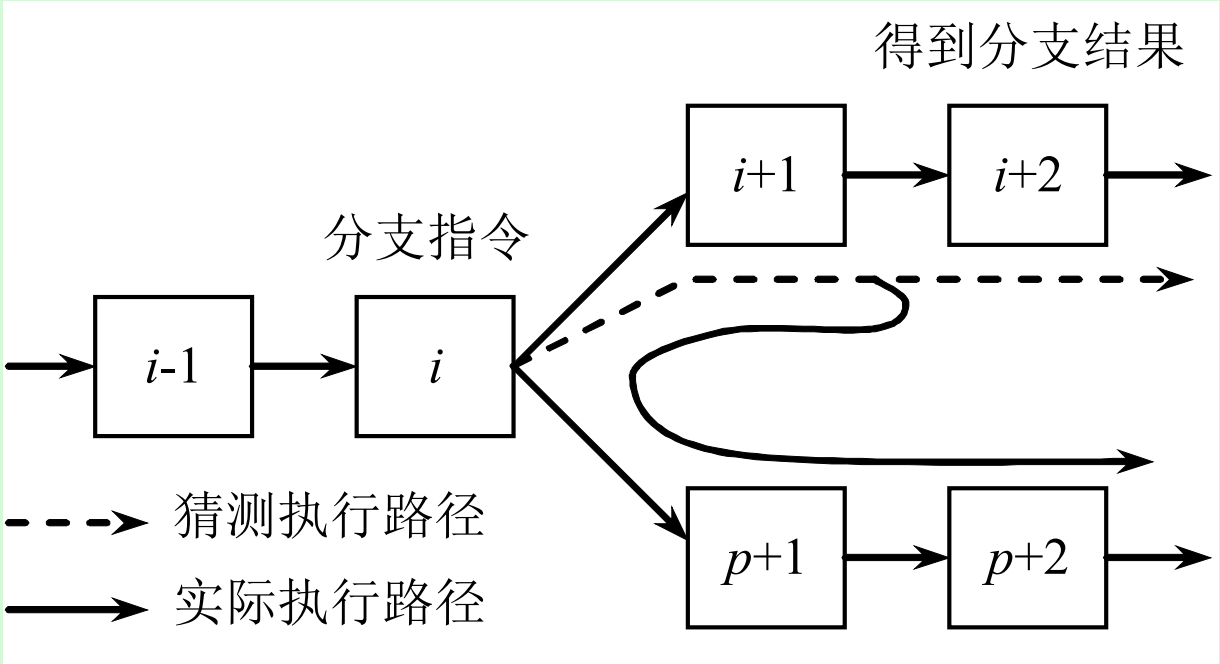
- 预测分支是否成功
- 尽快找到分支目标地址（或指令）

（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；
- 如何根据这些信息来预测分支的去向（甚至取到指令）。

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。



4.3.1 采用分支历史表 BHT

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。

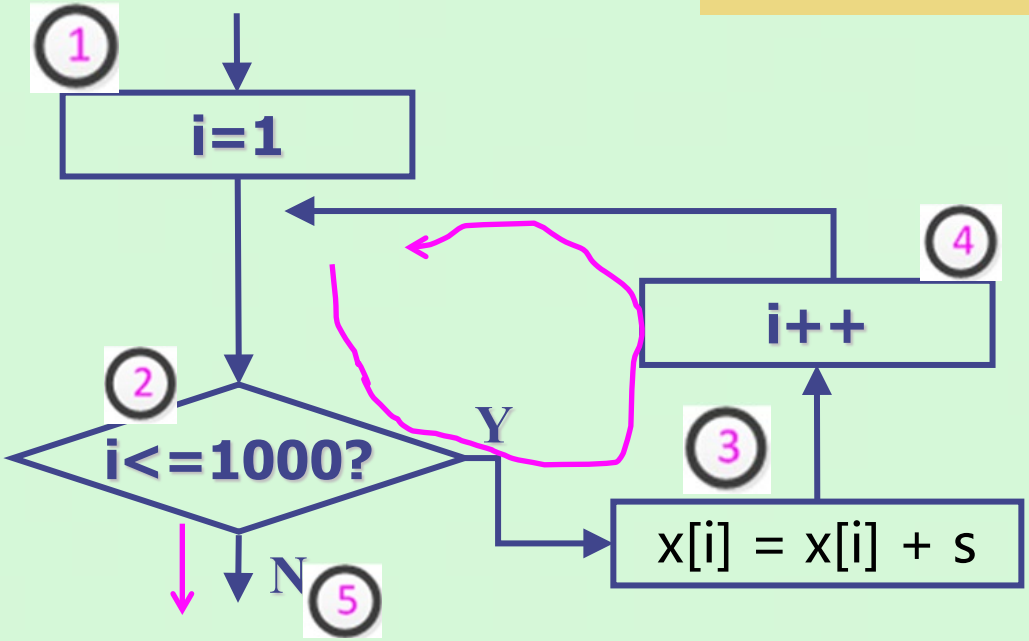
   （最简单）

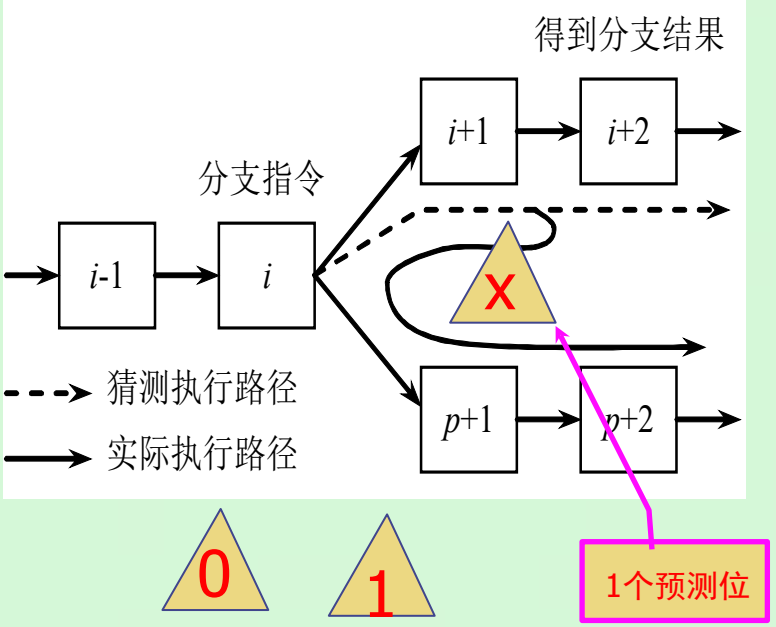
4.3.1 采用分支历史表 BHT

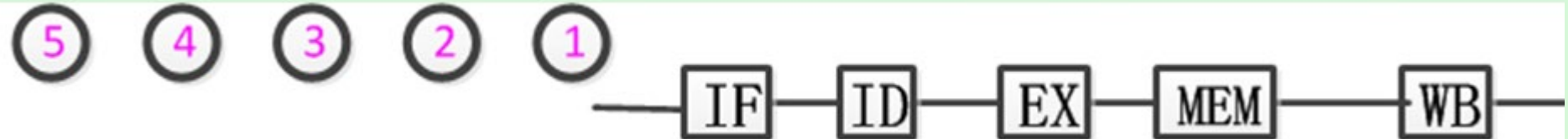
1. 只有1个预测位的分支预测缓冲

记录分支指令最近一次的历史，BHT中只需要1位二进制位。

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



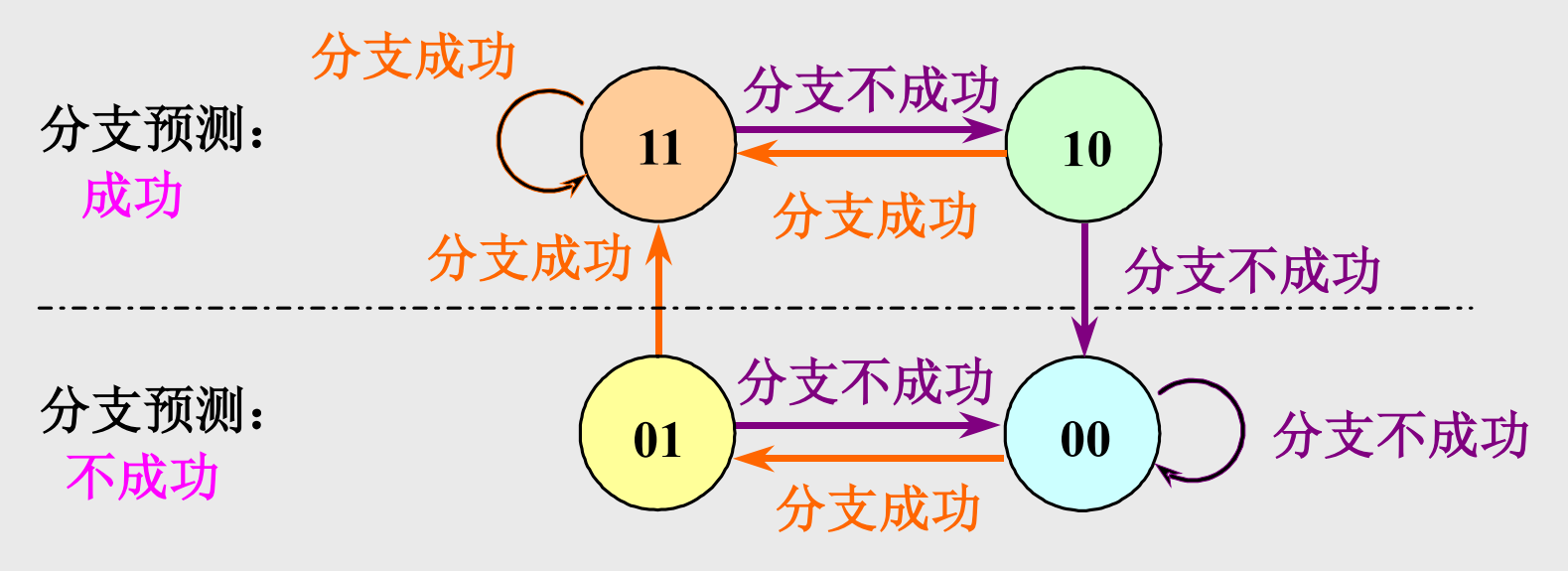




3. 采用两位二进制位来记录历史

- 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：



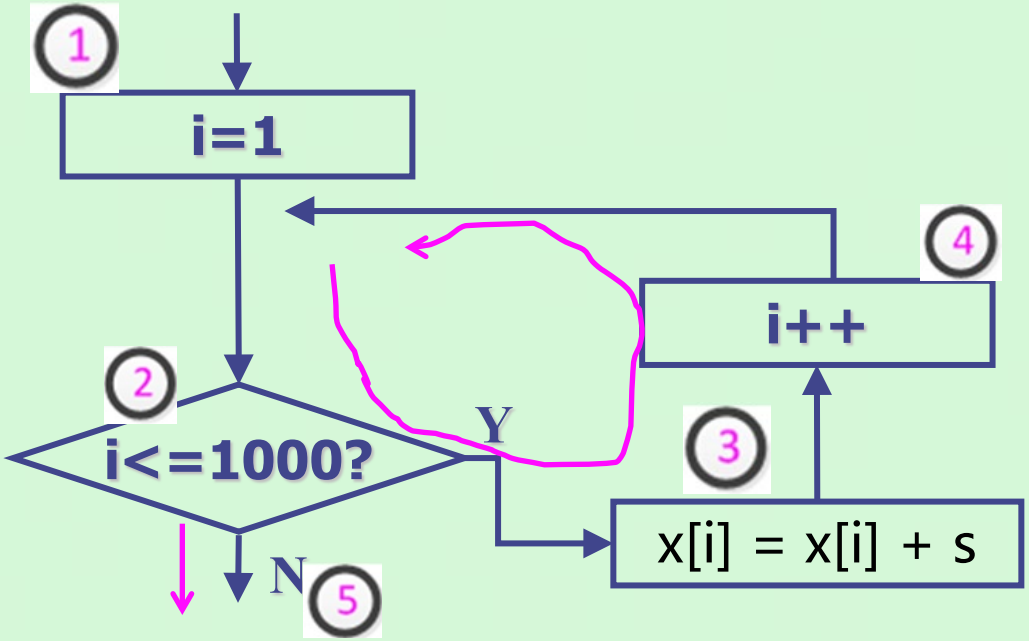
如果一次预测错误，还保持原来预测。如果连续两次预测错误，就改变预测方案。

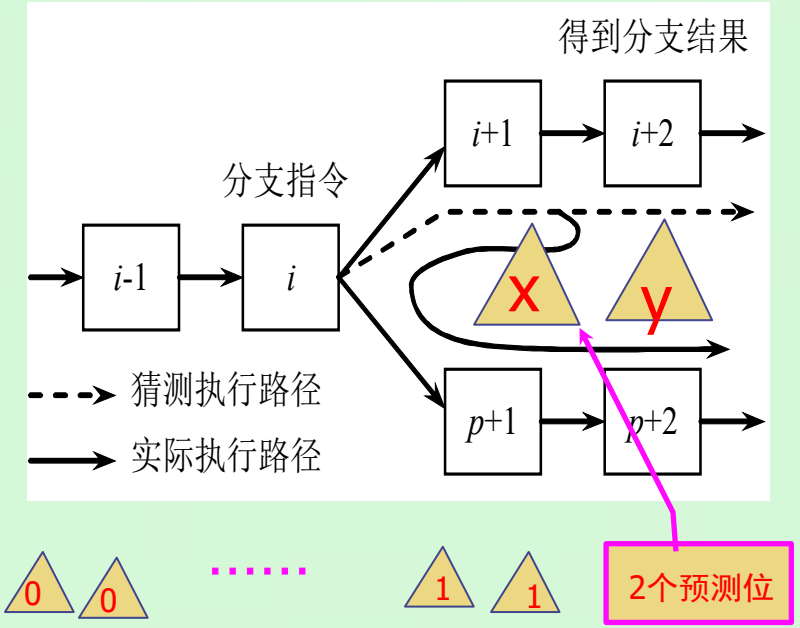
4.3.1 采用分支历史表 BHT

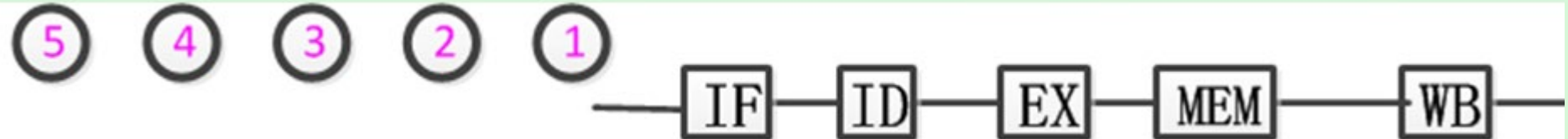
3. 采用两位二进制位来记录历史

- 提高预测的准确度

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







- 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。

6. 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。

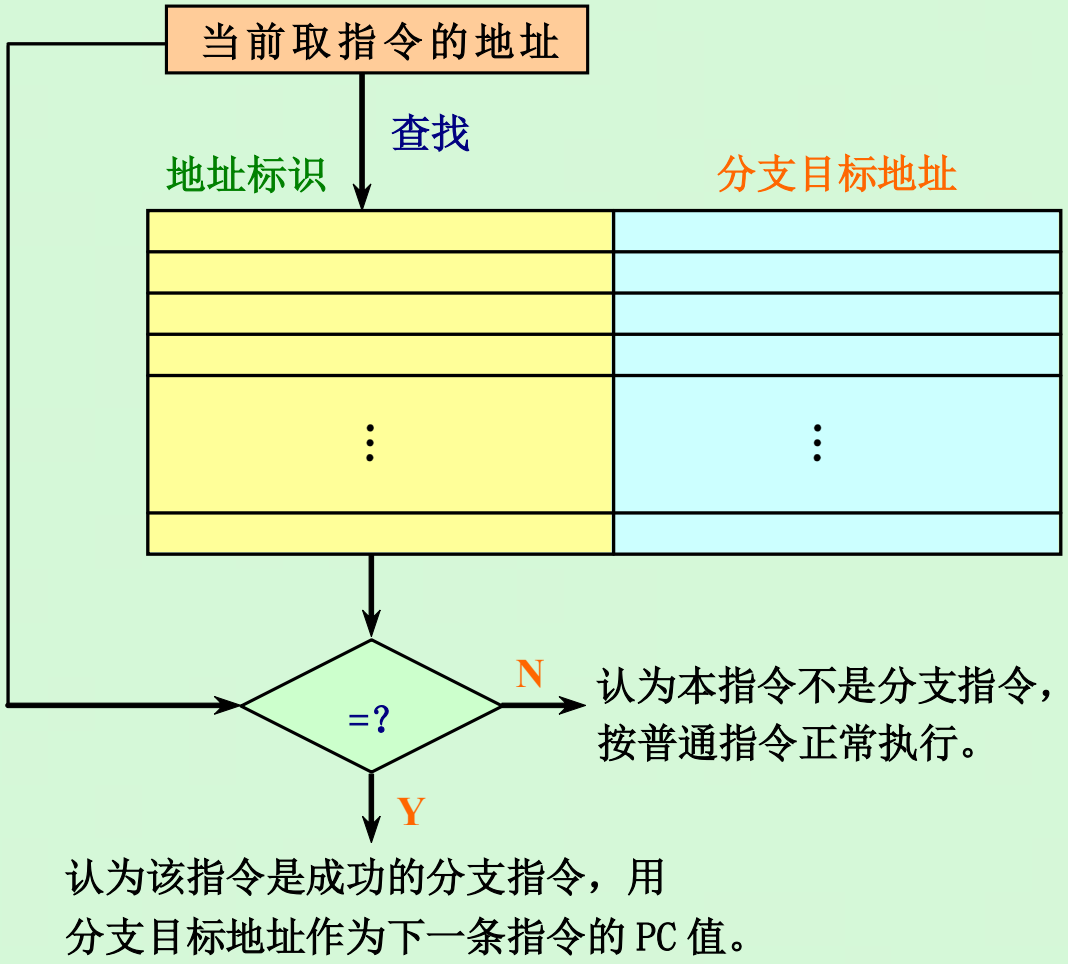
4.3.2 采用分支目标缓冲器BTB

目标：将分支的开销降为 0

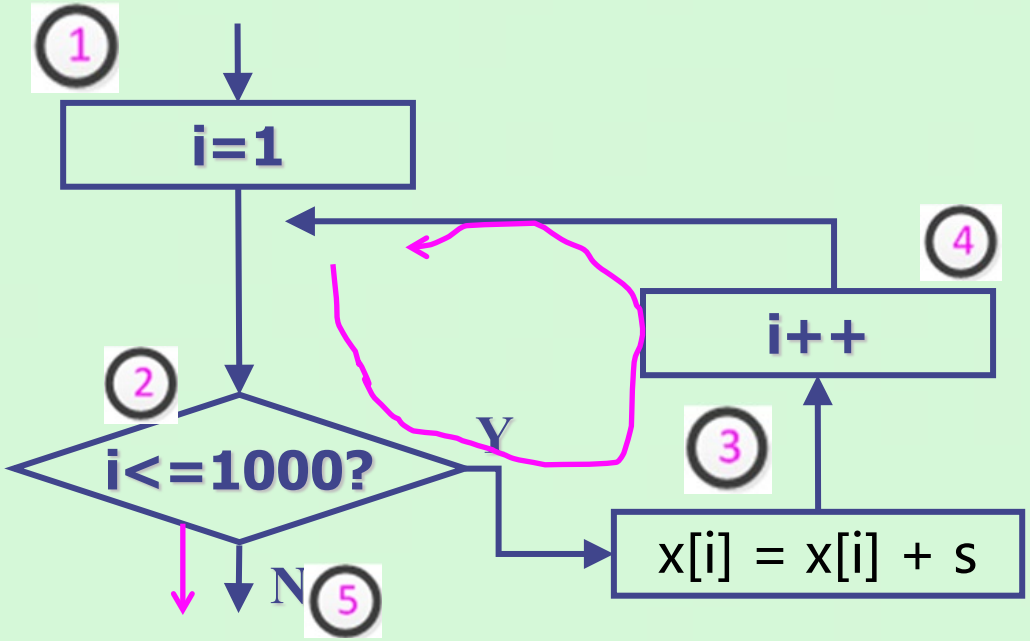
方法：分支目标缓冲

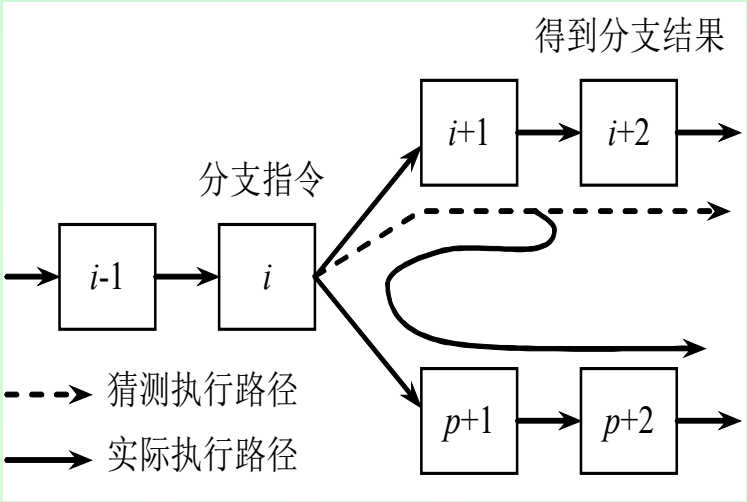
- 将分支成功的分支指令的地址和它的分支目标地址都放到一个缓冲区中保存起来，缓冲区以分支指令的地址作为标识。
- 这个缓冲区就是分支目标缓冲器（Branch-Target Buffer，简记为BTB，或者Branch-Target Cache）。

1. BTB的结构


当前取指令的地址
查找
地址标识
分支目标地址
=?
N
认为本指令不是分支指令，
按普通指令正常执行。
Y
认为该指令是成功的分支指令，用
分支目标地址作为下一条指令的 PC 值。

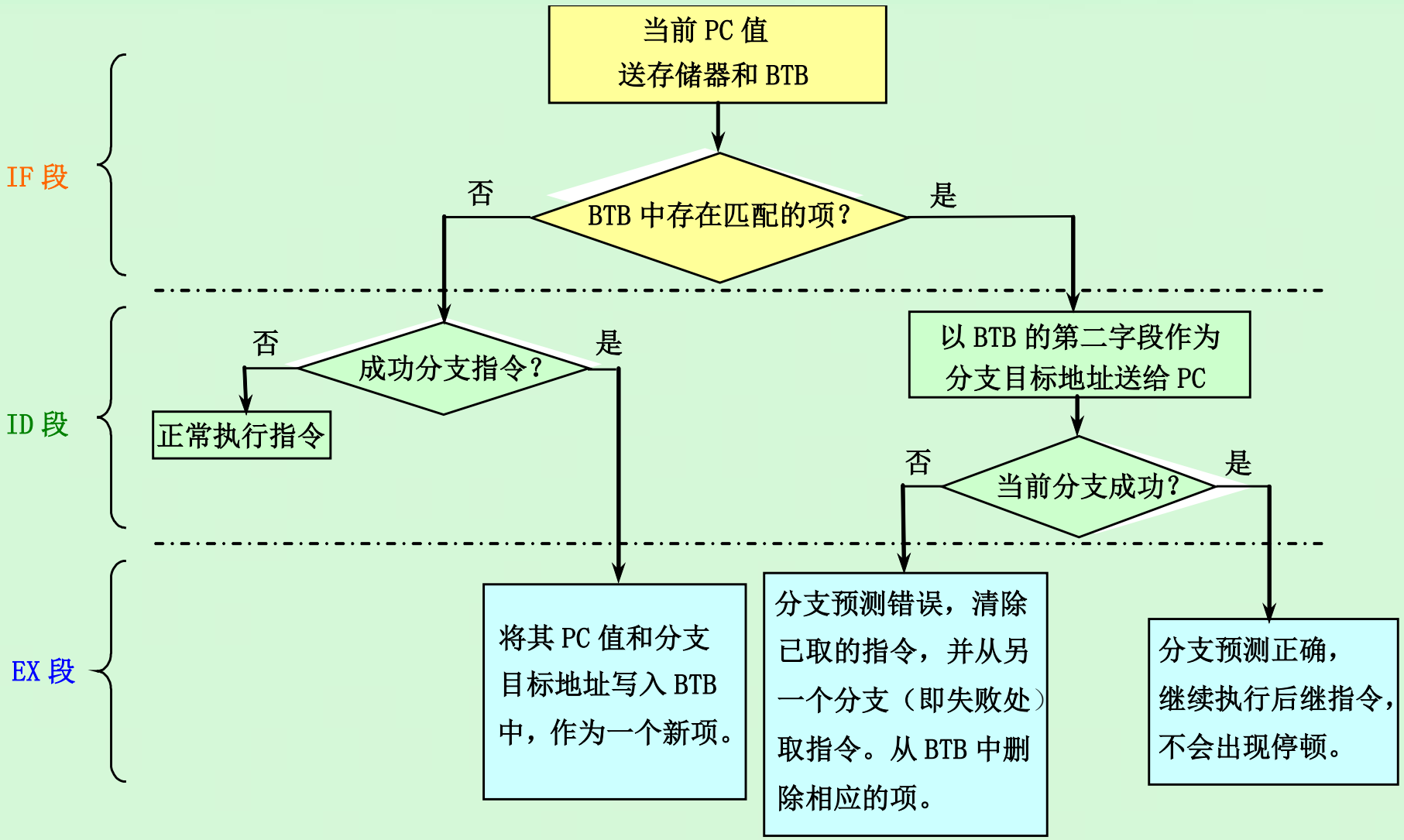
```
for (i=1;  i<=1000;  i++)
                x[i] = x[i] + s;
```





- 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：


IF 段
ID 段
EX 段
当前 PC 值
送存储器和 BTB
BTB 中存在匹配的项？
否
是
成功分支指令？
否
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
当前分支成功？
否
是
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 ：

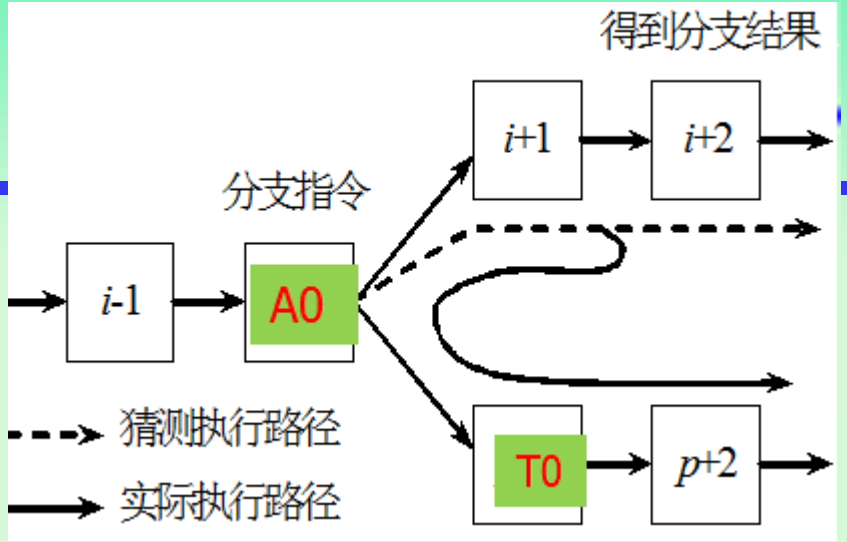
指令在BTB中？	预测	实际情况	延迟周期
是	成功	成功	0
是	成功	不成功	2
不是		成功	2
不是		不成功	0

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。

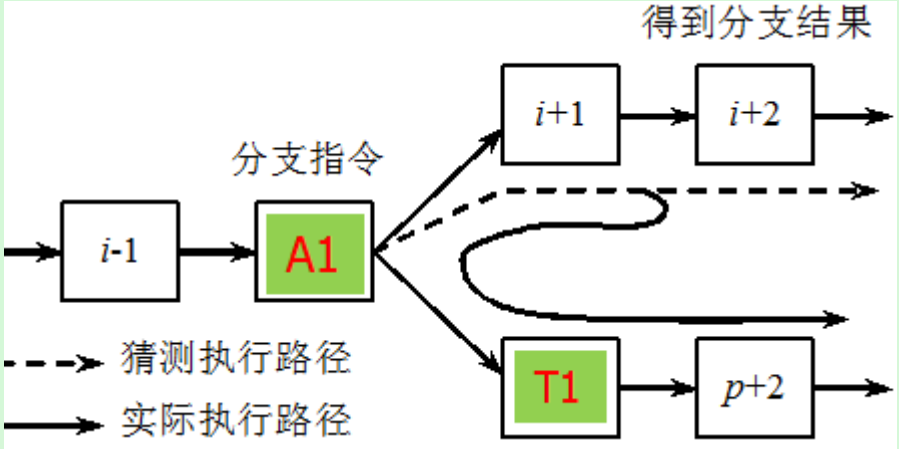
实现零延迟无条件分支，甚至有时还可以做到零延迟条件分支。



当前取指令的地址 PC

相联查找

分支指令地址	分支历史表 BHT	分支目标处的若干条指令
A0	T0	$I_{0,0}$, $I_{0,1}$, …, $I_{0,n-1}$
A1	T1	$I_{1,0}$, $I_{1,1}$, …, $I_{1,n-1}$
⋮	⋮	⋮
Ak-1	Tk-1	$I_{k-1,0}$, $I_{k-1,1}$, …, $I_{k-1,n-1}$



4.3.3 基于硬件的前瞻执行

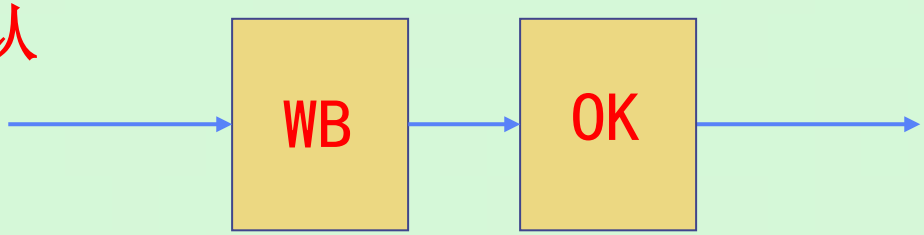
前瞻执行（speculation）的基本思想：

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。

1. 基于硬件的前瞻执行结合了三种思想：
   - 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(写到ROB中)
  - 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。

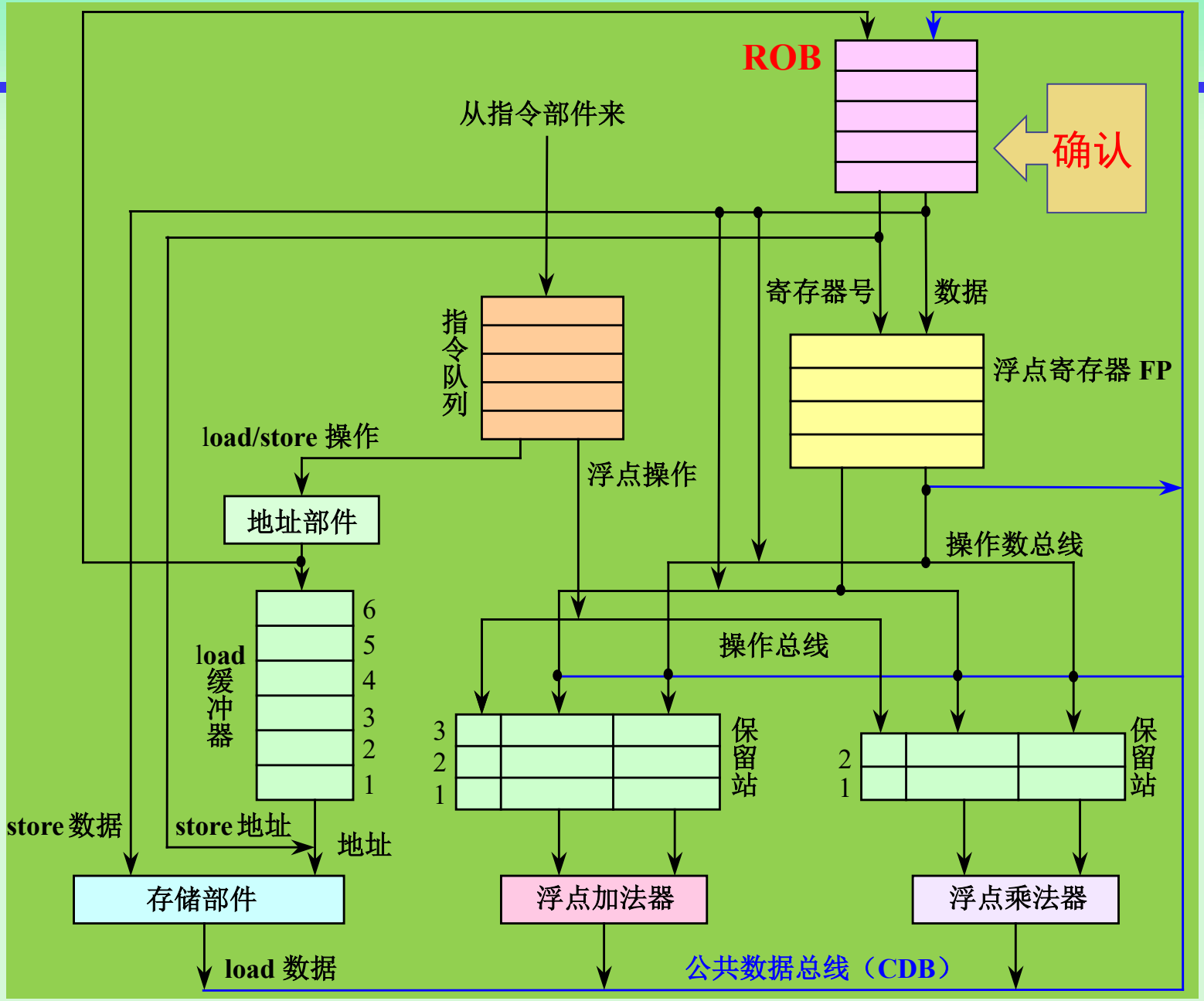
- 指令确认段 （ ROB中指令依次确认）

  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。

4. 支持前瞻执行的浮点部件的结构如下页图形所示。


ROB
确认
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 (写到ROB中)
  - 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:

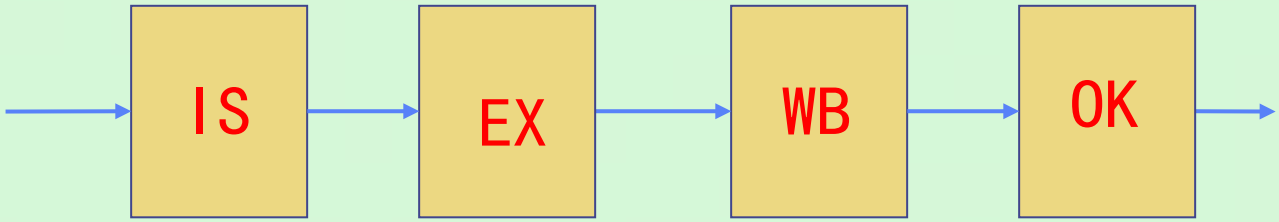
- 如果要写入存储器的数据已经就绪，就把该数据写入分配给该store指令的ROB项。
- 否则，就监测CDB，直到那个数据在CDB上播送出来，这时才将之写入分配给该store指令的ROB项。

➢ 确认 （ROB中指令依次确认）

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。

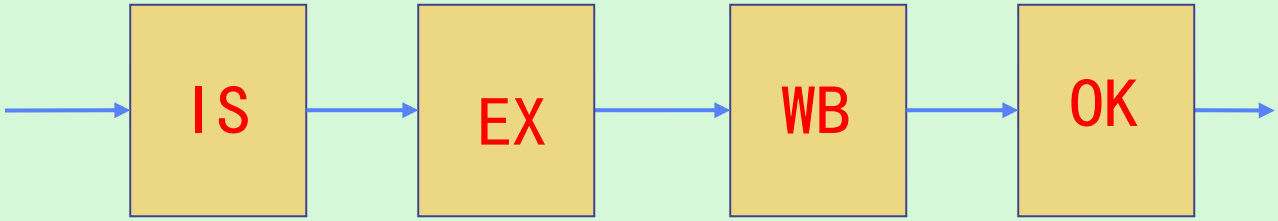
- 分支指令
  - 当预测错误的分支指令到达ROB队列的头部时，清空ROB，并从分支指令的另一个分支重新开始执行。

    （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。

例4.3 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（保留站和ROB）内容。

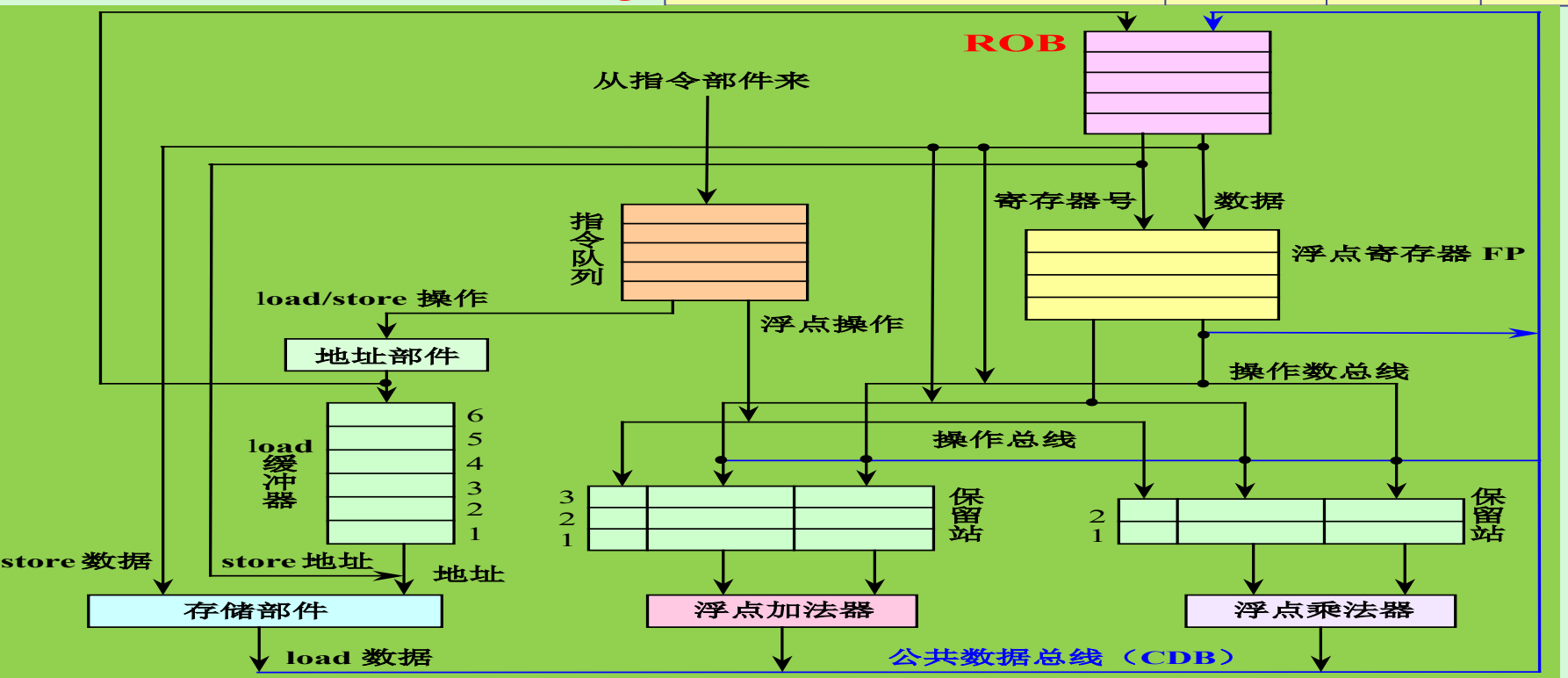
① L.D F6,34（R2）
② L.D F2,45（R3）
③ MUL.D F0,F2,F4
④ SUB.D F8,F6,F2
⑤ DIV.D F10,F0,F6
⑥ ADD.D F6,F8,F2

加法：2个时钟周期
乘法：10个时钟周期
除法：40个时钟周期



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

	指令	指令状态表		
		流出	执行	写结果
①	L.D F6 , 34(R2)	√	√	√
②	L.D F2 , 45(R3)	√	√	√
③	MUL.D F0 , F2 , F4	√	√	√
④	SUB.D F8 , F6 , F2	√	√	√
⑤	DIV.D F10 , F0 , F6	√		
⑥	ADD.D F6 , F8 , F2	√	√	√



前瞻执行中MUL.D确认前，保留站和ROB的状态

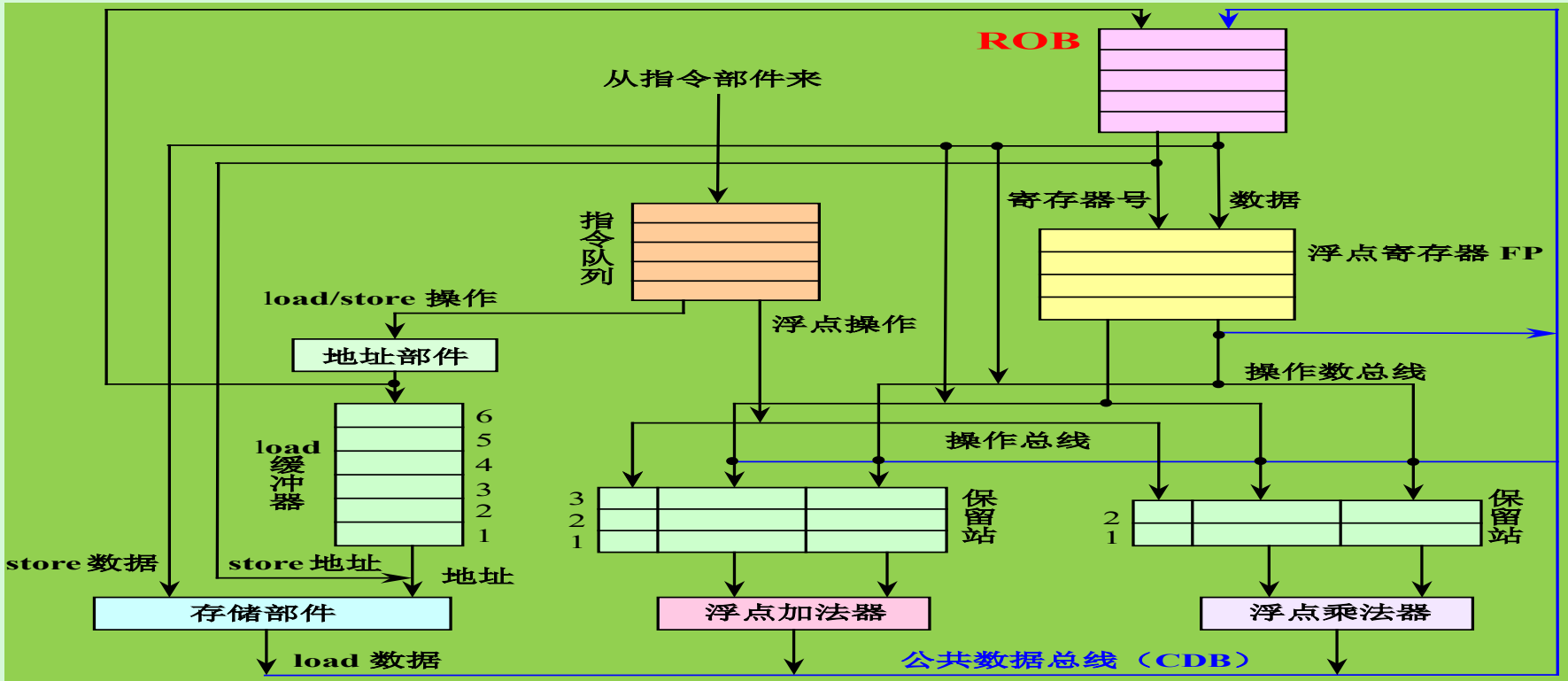
名称	保留站							
	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Add1	no							
Add2	no							
Add3	no							
Mult1	no	MUL	Mem[45+ Regs[R2]]	Regs[F4]			#3	
Mult2	yes	DIV		Mem[34+Regs[R2]]	#3		#5	

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

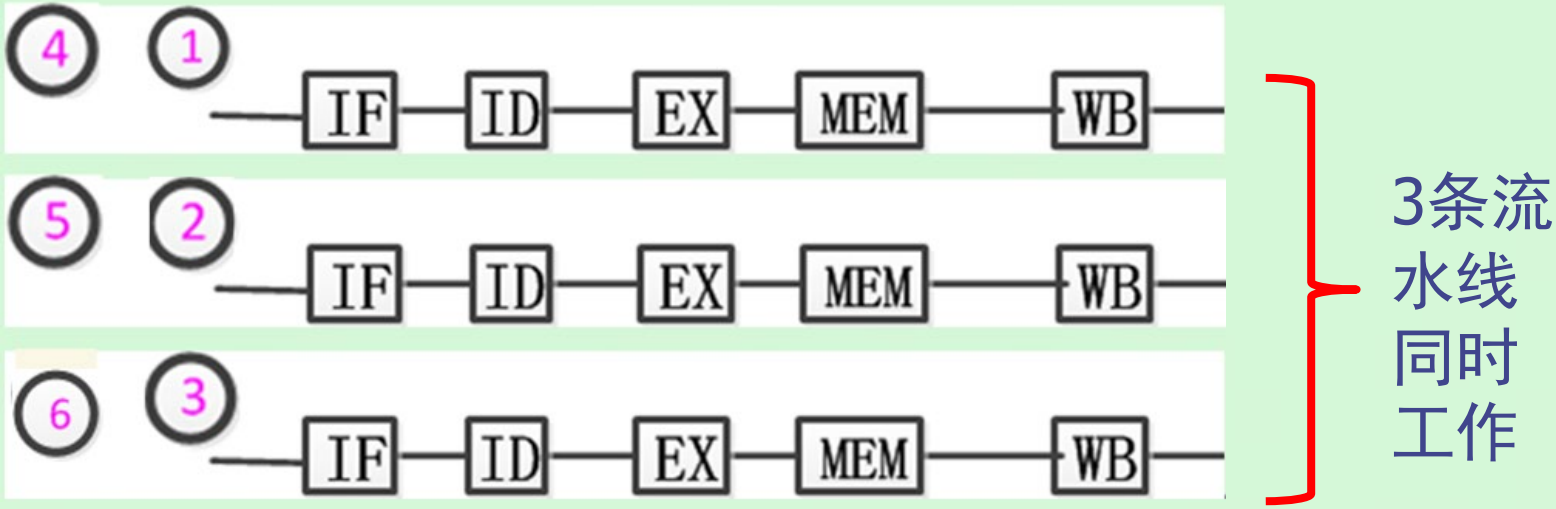
6. 前瞻执行

- 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。



4.4 多指令流出技术

- 一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



单流出时空图

指令 \ 时钟周期	1	2	3	4	5	6	7
I_1	IF	ID	EX	MEM	WB		
I_2		IF	ID	EX	MEM	WB	
I_3			IF	ID	EX	MEM	WB

多流出时空图

指令 \ 时钟周期	1	2	3	4	5	6	7
I_1	IF	ID	EX	MEM	WB		
I_2	IF	ID	EX	MEM	WB		
I_3	IF	ID	EX	MEM	WB		
		IF	ID	EX	MEM	WB	
		IF	ID	EX	MEM	WB	
		IF	ID	EX	MEM	WB	
			IF	ID	EX	MEM	WB
			IF	ID	EX	MEM	WB
			IF	ID	EX	MEM	WB

单流出和多流出处理机执行指令的时空图

1. 多流出处理机有三种基本风格

- 超标量（Superscalar）
  - 一个处理机中有多条流水线，就称为超标量处理机。
  - 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）
  - 把能并行执行的多条指令组装成一条很长的指令字。由这个指令字控制处理机中的多个执行部件同时执行的处理机。
  - 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。

1. 多流出处理机有三种基本风格

- 超流水线处理机（Superpiple）
  - 将每个流水段进一步进行细分，使每个流水段的耗时小于一个时钟周期，这样在一个时钟周期内就能够分时流出多条指令。这种处理机称为超流水线处理机。
  - 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。
  - 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。
  - 典型的超流水线处理器：SGI公司的MIPS系列R4000
  -

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，因为处理机能自己检测下一条指令能否流出，从而不需要重新排列指令来满足指令的流出。
   - 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例子。

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	顺序执行	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	带有猜测的动态执行	带有猜测的乱序执行	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

4.4.1 基于静态调度的多流出技术

- 在典型的超标量处理器中，有多条流水线可以同时工作。
- 指令按序流出，在流出时进行冲突检测。

  在当前流出的指令序列中，不存在数据冲突或者相关冲突。

- 流出部件检测结构冲突或者数据冲突。

  一般分两阶段实现：

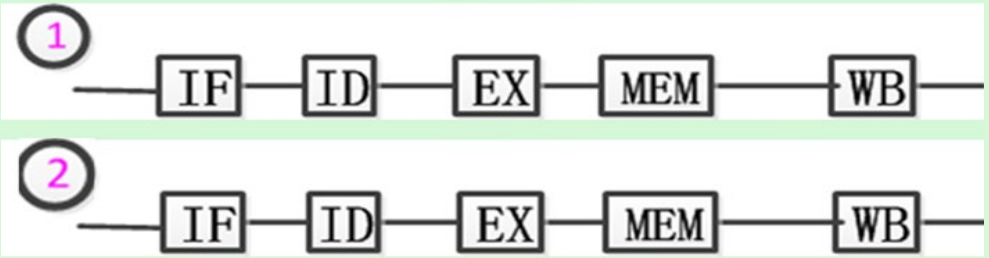
  - 第一阶段：进行流出包内的冲突检测，选出初步判定可以流出的指令。
  - 第二阶段：检测所选出的指令与正在执行的指令是否有冲突。

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。



整数型指令流水线

浮点型指令流水线

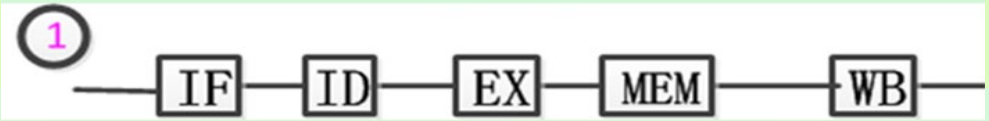
1. 要求：

   同时取两条指令（64位），译码两条指令（64位）。

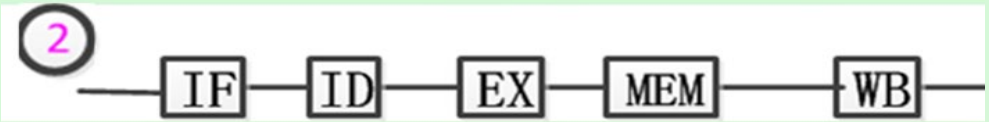
2. 对指令的处理包括以下步骤：
   - 从Cache中取两条指令。
   - 确定哪几条指令可以流出（0～2条指令）。
   - 把它们发送到相应的功能部件。

3. 双流出超标量流水线中指令的执行过程
   - 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，下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



整数型指令流水线



浮点型指令流水线

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条指令；
  - 否则就是流出包中的第二条指令，其延迟就是两条指令。

4.4.2 基于动态调度的多流出技术

- 扩展Tomasulo算法：支持两路超标量
  - 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。

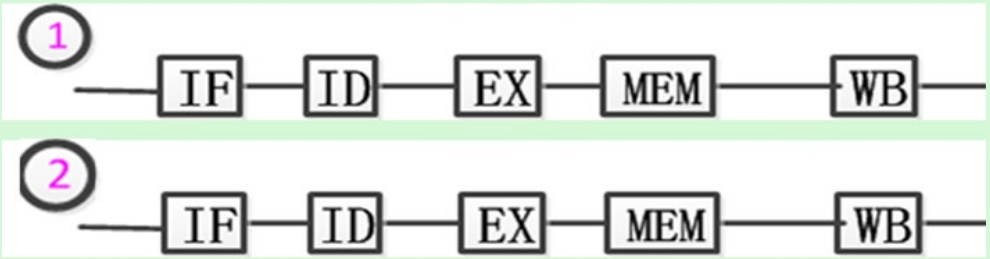
关键在于：对保留站的分配和对流水线控制表格的修改。

- 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4.4　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
```



整数型指令流水线

浮点型指令流水线

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。
- 分支指令单独流出，没有采用延迟分支，但分支预测是完美的。分支指令完成前，其后续指令只能被取出和流出，但不能执行。

- 因为写结果占用一个时钟周期，所以产生结果的延迟为：整数运算一个周期，load两个周期，浮点加法运算3个周期。

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。

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。

为了便于分析，表中列出了访存发生的时刻。运行结果如下图所示。

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

从图中可以看出：

- 程序基本可以达到3拍流出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。

- 每个循环迭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。

4.4.3 超长指令字技术（VLIW）

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，所有的处理和指令安排都是由编译器完成的。

例4.5 假设VLIW处理机每个时钟周期可同时流出5条指令：两条访存指令、两条浮点操作指令和一条整数指令或分支指令。对于例4.4中的循环展开后的代码，给出它在该VLIW中的代码序列。不考虑分支指令的延迟槽。

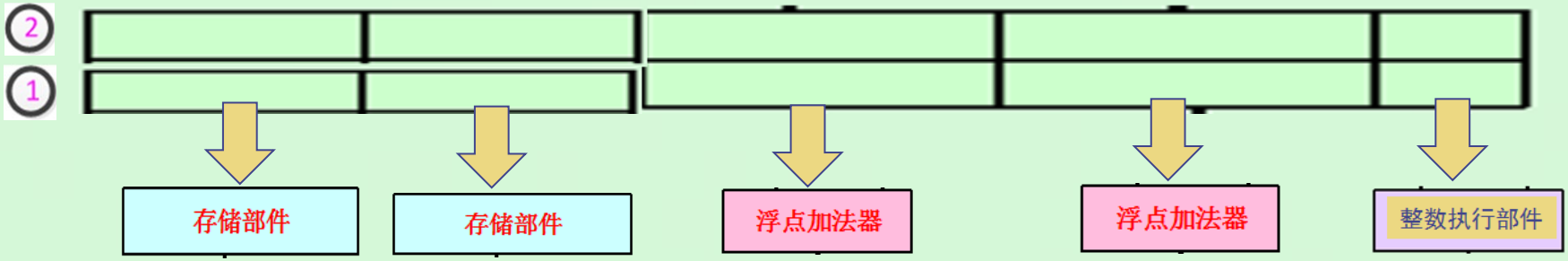
解 代码序列如下图所示。

运行时间为8个时钟周期。

每遍循环平均1.6个时钟周期。

8个时钟周期内流出了17条指令，每个时钟周期2.1条。

8个时钟周期共有操作槽8×5=40个，有效槽的比例为42.5%。

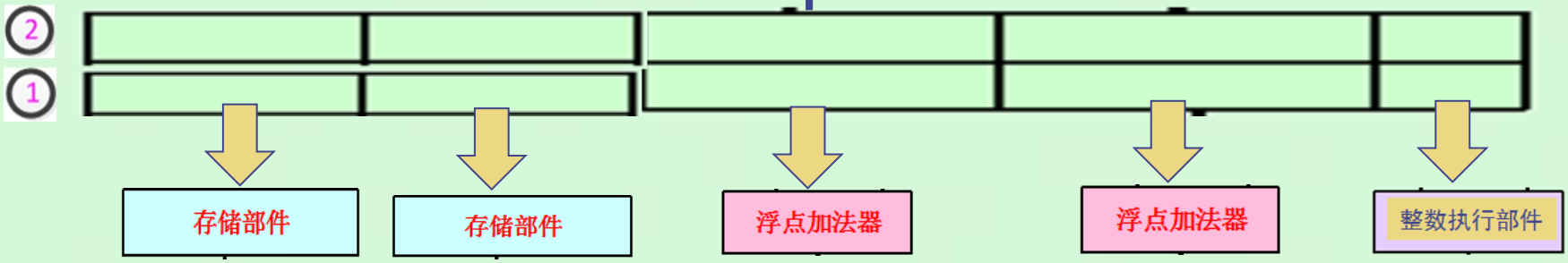


下面是前面使用的循环程序段，对其进行循环展开5次，并在超标量流水线上进行调度。

```
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
```

解 将循环展开5遍并调度，可得如下代码：

```
Loop:  L.D     F0,0(R1)
       L.D     F6,-8(R1)
       L.D     F10,-16(R1)
       L.D     F14,-24(R1)
       L.D     F18,-32(R1)
       ADD.D   F4,F0,F2
       ADD.D   F8,F6,F2
       ADD.D   F12,F10,F2
       ADD.D   F16,F14,F2
       ADD.D   F20,F18.F2
       S.D     F4,0(R1)
       S.D     F8,-8(R1)
       S.D     F12,-16(R1)
       DADDIU  R1,R1,# -40
       S.D     F16,16 (R1)
       BNE     R1,R2,Loop
       S.D     F20,8(R1)
```



	访存指令1	访存指令2	浮点指令1	浮点指令2	整数/转移指令
1	L.D F0,0(R1)	L.D F6,-8(R1)			
2	L.D F10,-16(R1)	L.D F14,-24(R1)			
3	L.D F18,-32(R1)		ADD.D F4,F0,F2	ADD.D F8,F6,F2	
4			ADD.D F12,F10,F2	ADD.DF16,F14,F2	
5			ADD.D F20,F18,F2		
6	S.D F4,0(R1)	S.D F8,-8(R1)			
7	S.D F12,-16(R1)	S.D F16, - 24(R1)			DADDIUR1,R1,#-4
8	S.D F20,8(R1)				BNE R1,R2,Loo

将多条指令组装成一条长指令字时要考虑指令之间的相关。

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

4.4.4 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？

主要受以下三个方面的影响：

- 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。

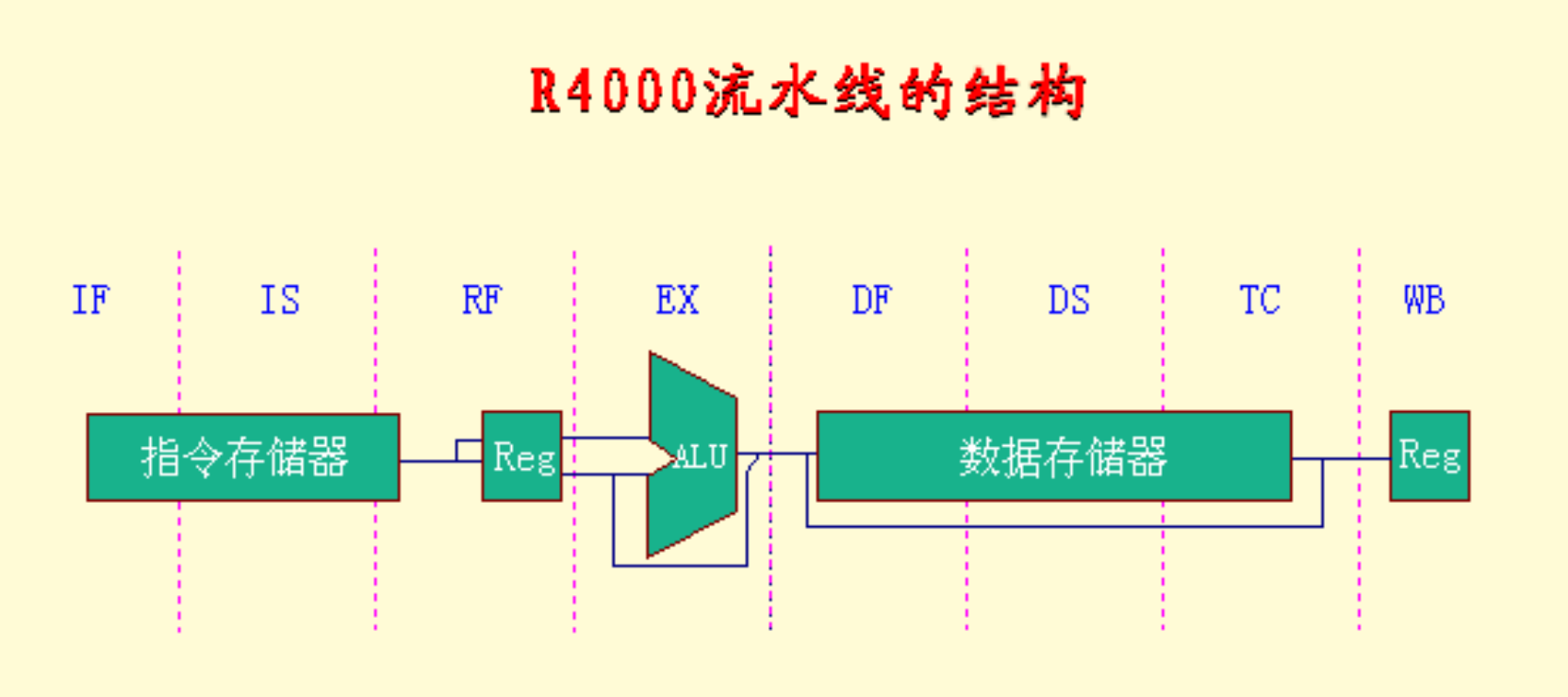
4.4.5 超流水线处理机

1. 将每个流水段进一步进行细分，使每个流水段的耗时小于一个时钟周期，这样在一个时钟周期内就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. 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。
4. 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8 KB
     - 每个Cache的数据宽度为64 b
   - 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的指令流水线由8级组成，如下图

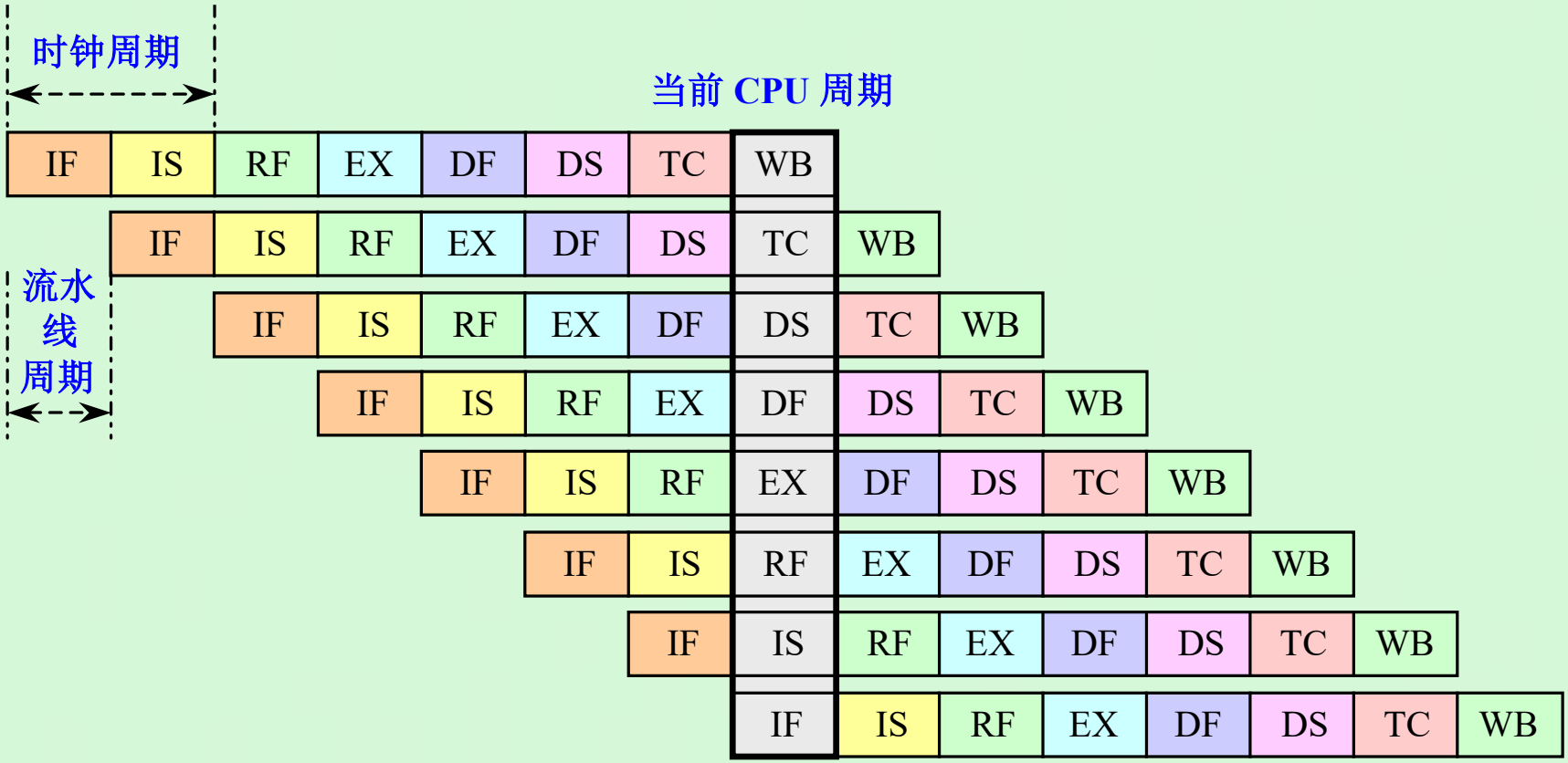


每个段耗费1/2个时钟周期。

各级的功能

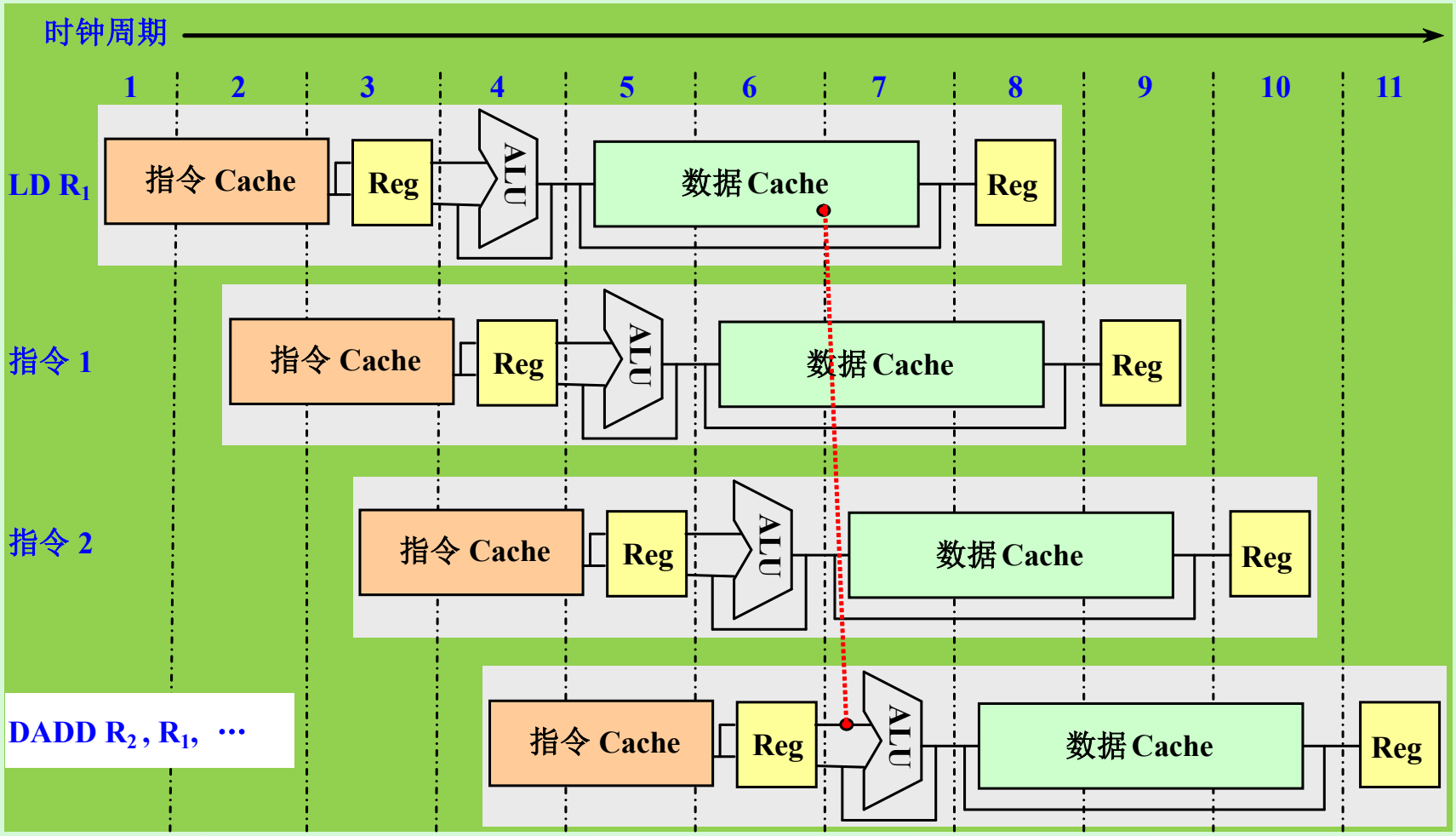
- IF：取指令的前半步，根据PC值去启动对指令Cache的访问。
- IS：取指令的后半步，在这一级完成对指令Cache的访问。
- RF：指令译码，访问寄存器组读取操作数，冲突检测，并判断指令Cache是否命中。
- EX：指令执行。包括有效地址计算，ALU操作，分支目标地址计算，条件码测试。
- DF：取数据的前半步，启动对数据Cache的访问。
- DS：取数据的后半步，在这一级完成对数据Cache的访问。
- TC：标识比较，判断对数据Cache的访问是否命中。
- WB：load指令或运算型指令把结果写回寄存器组。

➢ MIPS R4000指令流水线时空图



每隔1/2个时钟周期流出一条指令。

➢ 载入延迟为两个时钟周期



4.5 循环展开和指令调度

4.5.1 循环展开和指令调度的基本方法

1. 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。
2. 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。

3. 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。
4. 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

假设采用第3章的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4.6 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;
```

解

- 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。

- 不进行指令调度的情况下，程序的实际执行情况：

		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。

- 指令调度以后，程序的执行情况如下：
  - 把DADDIU指令调度到了L.D指令和ADD.D指令之间的“空转”拍。
  - 把S.D指令放到了分支指令的延迟槽中。
  - 对存储器地址偏移量进行调整。

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

				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,	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。

    (取、加、存)
    - 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。

例4.7 （体现循环展开技术的特点）

将上述例子中的循环展开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。

解

无需在循环体后面增加补偿代码

分配寄存器（不重复使用寄存器 ）：

- F0、F4：用于展开后的第1个循环体
- F2：保存常数
- F6、F8：展开后的第2个循环体
- F10、F12：第3个循环体
- F14、F16：第4个循环体

➢ 展开后没有调度的代码如下：

			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
	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

➢ 对指令序列进行优化调度，以减少空转周期：

			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。

5. 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。

4.5.2 静态超标量处理机中的循环展开

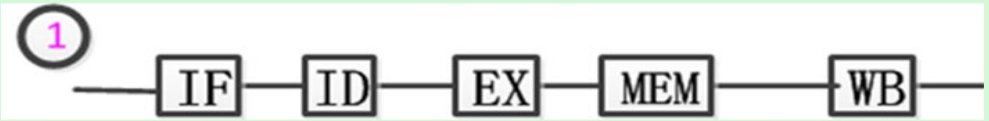
根据表4.4给出的延迟条件，超标量处理器如何进行循环展开和指令调度？

例4.8 下面是前面使用的循环程序段，对其进行循环展开，并在超标量流水线上进行调度。

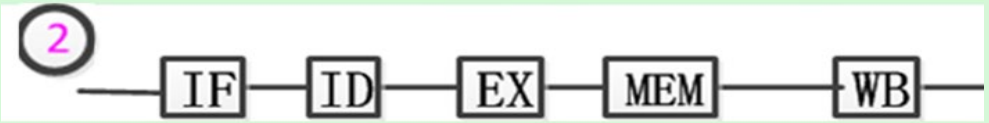
```
Loop: L.D     F0,0(R1)      // 取一个数组元素放入F0
      ADD.D   F4,F0,F2      // 加上在F2中的标量
      S.D     F4,0(R1)      // 存结果
      DADDIU  R1,R1,#-8
                // 将指针减少8（每个数据占8个字节）
      BNE     R1,R2,Loop
      // 若R1不等于R2，表示尚未结束，转移到Loop继续
```

解 将循环展开5遍并调度，可得如下代码：

```
Loop:  L.D     F0,0(R1)
       L.D     F6,-8(R1)
       L.D     F10,-16(R1)
       L.D     F14,-24(R1)
       L.D     F18,-32(R1)
       ADD.D   F4,F0,F2
       ADD.D   F8,F6,F2
       ADD.D   F12,F10,F2
       ADD.D   F16,F14,F2
       ADD.D   F20,F18.F2
       S.D     F4,0(R1)
       S.D     F8,-8(R1)
       S.D     F12,-16(R1)
       DADDIU  R1,R1,# -40
       S.D     F16,16 (R1)
       BNE     R1,R2,Loop
       S.D     F20,8(R1)
```



整数型指令流水线



浮点型指令流水线

- 进一步针对超标量进行调度后的指令序列如下所示

	整数指令	浮点指令	时钟周期
Loop:	L.D F0（R1）		1
	L.D F6,-8（R1）		2
	L.D F10,-16（R1）	ADD.D F4,F0,F2	3
	L.D F14,-24（R1）	ADD.D F8,F6,F2	4
	L.D F18,-32（R1）	ADD.D F12,F10,F2	5
	S.D F4,0（R1）	ADD.D F16,F14,F2	6
	S.D F8,-8（R1）	ADD.D F20,F18,F2	7
	S.D F12,-16（R1）		8
	DADDIU R1,R1,#-40		9
	S.D F16,16 (R1)		10
	BNE R1,R2,Loop		11
	S.D F20,8（R1）		12

- 每次循环需12个时钟周期，即计算每个结果需要2.4个时钟周期。

  （每次循环计算5个结果）

- 在普通的MIPS流水线上，没有调度的代码迭代一次为9个时钟周期，调度后为6个时钟周期，展开4次并调度后每个迭代为3.5个时钟周期。
- 与之相比，超标量流水线的性能提高分别为：

  3.75倍、2.5倍、1.4倍

- 在这个例子中可以看到，超标量MIPS流水线的性能主要受限于：整数计算和浮点计算之间的平衡问题。

本例中没有足够的浮点指令来使两路流水线都达到饱和。

作业

第四章 作业　　第6题。

4.6 下面的一段MIPS汇编程序是计算高斯消去法中的关键一步，用于完成下面公式的计算：

$$Y = a \times X + Y$$

其浮点指令延迟如表4.3所示，整数指令均为1个时钟周期完成，浮点和整数部件均采用流水。整数操作之间以及与其它所有浮点操作之间的延迟为0，转移指令的延迟为0。X中的最后一个元素存放在存储器中的地址为DONE。

```
FOO:  L.D     F2,0(R1)
      MUT.D   F4,F2,F0
      L.D     F6,0(R2)
      ADD.D   F6,F4,F6
      S.D     F6,0[R2]
      DADDIU          R1,R1,#8
      DADDIU          R2,R2,#8
      DSUBIU          R3,R1,#DONE
      BNEZ    R3, FOO
```

(1) 对于标准的MIPS单流水线，上述循环计算一个Y值需要多少时间？其中有多少空转周期？

作业

第四章 作业 第6题。

（2）对于标准的MIPS单流水线，将上述循环顺序展开4次，不进行任何指令调度，计算一个Y值平均需要多少时间？加速比是多少？其加速是如何获得的？

（3）对于标准的MIPS单流水线，将上述循环顺序展开4次，优化和调度指令，使循环处理时间达到最优，计算一个Y值平均需要多少时间？加速比是多少？

（4）对于采用如图4.8前瞻执行机制的MIPS处理器（只有一个整数部件）。当循环第二次执行到

BNEZ R3,FOO

时，写出前面所有指令的状态，包括指令使用的保留站、指令起始节拍、执行节拍和写结果节拍，并写出处理器当前的状态。

（5）对于2路超标量的MIPS流水线，设有两个指令流出部件，可以流出任意组合的指令，系统中的功能部件数量不受限制。将上述循环展开4次，优化和调度指令，使循环处理时间达到最优。计算一个Y值平均需要多少时间？加速比是多少？

（6）对于如图4.13结构的超长指令字MIPS处理器，将上述循环展开4次，优化和调度指令，使循环处理时间达到最优。计算一个Y值平均需要多少时间？加速比是多少？

表4.3，假定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

解：（1）

```
        L.D        F2, 0(R1)      1    ; load x[i]
        Stall                           ; 从浮点取到浮点运算间的空转
        MUT.D      F4, F2, F0     2    ; multiply a * x[i]
        L.D        F6, 0(R2)      3    ; load y[i]
        Stall                           ; 从浮点取到浮点运算间的空转
        Stall                           ;          和          两个浮点运算间共延迟 3 个时钟
        ADD.D      F6, F4, F6     4    ; add a * x[i] + y[i]
        Stall                           ; 从浮点运算到浮点取间的空转
        Stall                           ;
        S.D    F6, 0[R2]          5    ; store y[i]
        DADDIU     R1, R1, #8     6    ; increment x index
        DADDIU     R2, R2, #8     7    ; increment y index
    DSUBIU  R3, R1, #DONE         8    ; test if done    由题设，这之后没有空转
        BNEZ       R3, FOO        9    ; loop if not done    由题设，这之后没有空转
```

所以，共有 14 个时钟周期，其中有 5 个空转周期。

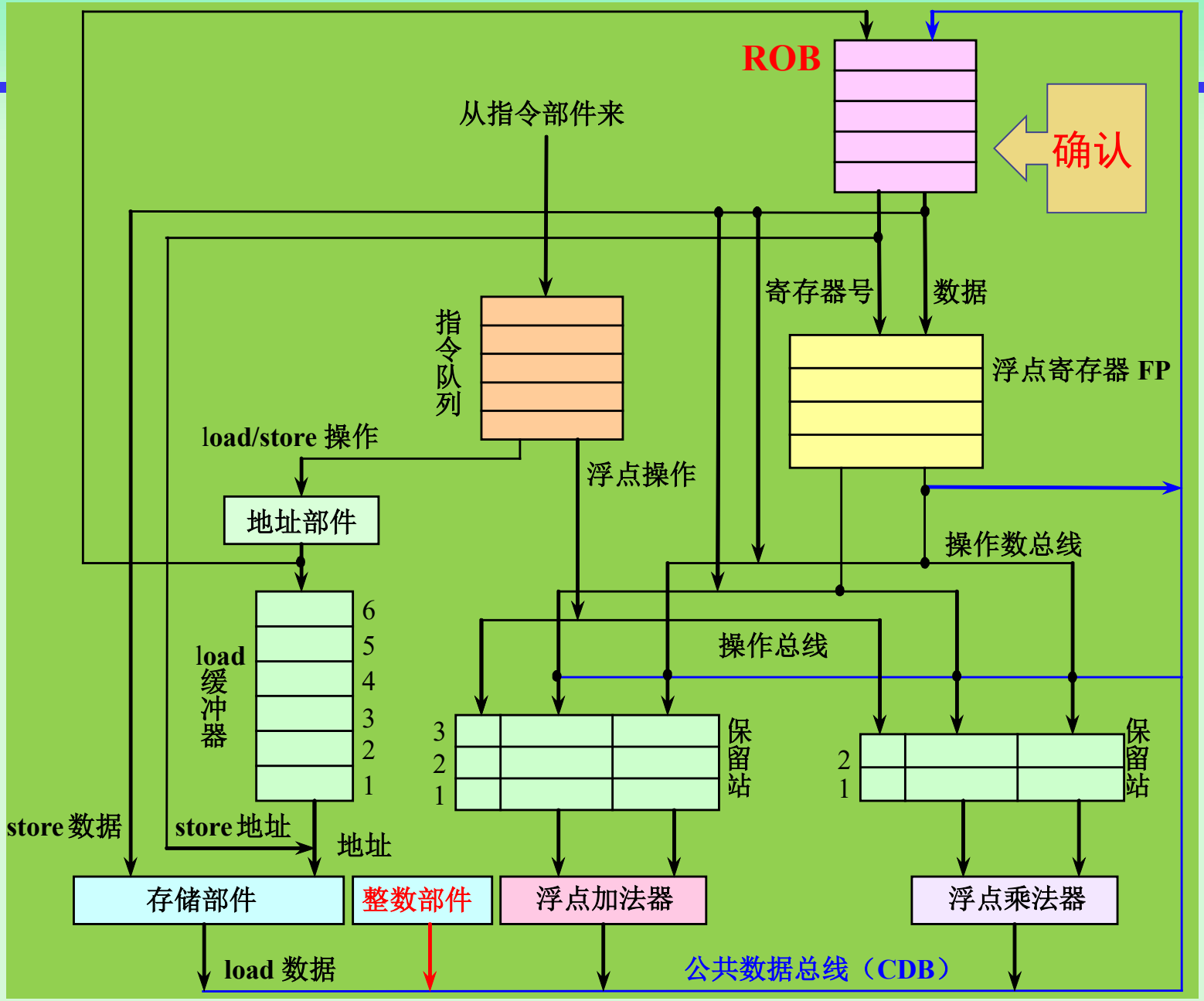
（2）循环顺序展开4 次，不进行任何指令调度，则指令1～5 及其间的stall 都是必要的，只是指令6～9 只需执行一次，因此，共有 10 × 4 ＋ 4 ＝ 44 个时钟周期，计算出4 个Y 值，所以计算一个Y 值需要11 个时钟周期，加速比为：14/11 = 1.27 。加速主要是来自减少控制开销，即减少对R1、R2 的整数操作以及比较、分支指令而来的。

（3）循环顺序展开4 次，优化和调度指令，如下：

```
L.D      F2, 0(R1)
L.D      F8, 8(R1)
L.D      F14, 16(R1)
L.D      F20, 24(R1)
MUT.D    F4, F2, F0
MUT.D    F10, F8, F0
MUT.D    F16, F14, F0
MUT.D    F22, F20, F0
L.D      F6, 0(R2)
L.D      F12, 8(R2)
L.D      F18, 16(R2)
L.D      F24, 24(R2)
ADD.D    F6, F4, F6
ADD.D    F12, F10, F12
ADD.D    F18, F16, F18
ADD.D    F24, F22, F24
S.D      F6, 0[R2]
S.D      F12, 8[R2]
S.D      F18, 16[R2]
S.D      F24, 24[R2]
DADDIU   R1, R1, #32
DADDIU   R2, R2, #32
DSUBIU   R3, R1, #DONE
BNEZ     R3, FOO
```

共用了24 个时钟周期，则计算一个Y 值平均需要 24/4 = 6 个时钟周期，
加速比：14/6 = 2.33

通过 4 次展开，完全消去了 stall，处理时间达到了最优，程序不复杂，需要注意的几点是：1 展开循环后使用不同的 R，并且对于双精度浮点操作，使用以偶数开头的浮点 R；2 偏移地址做相应的转换；3 根据题设，最后几条整数指令不需要做调度，有些同学根据书上的习惯做法，做了调度，这里没有必要。


ROB
确认
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
load 缓冲器
6
5
4
3
2
1
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
整数部件
浮点加法器
浮点乘法器
load 数据
公共数据总线（CDB）

（4）

指令	指令执行时钟			
	流出	执行	写结果	确认
L.D F2, 0（R1）	1	2	3	4
MUL.D F4, F2, F0	2	4	5	6
L.D F6, 0（R2）	3	4	6	7
ADD.D F6, F4, F6	4	~~8~~	9	10
S.D F6, 0（R2）	5	~~11~~	12	13
DADDIU R1, R1, #8	6	7	8	
DADDIU R2, R2, #8	7	8	9	
DSUBIU R3,R1,#DONE	8	9	10	
BNEZ R3, FOO	9	10		
L.D F2, 0（R1）	10	11	13	~~14~~
MUL.D F4, F2, F0	11	13	14	~~15~~
L.D F6, 0（R2）	12	13	15	16
ADD.D F6, F4, F6	13	17	18	19
S.D F6, 0（R2）	14	20	21	22
DADDIU R1, R1, #8	15	16	17	
DADDIU R2, R2, #8	16	17	18	
DSUBIU R3,R1,#DONE	17	18	19	
BNEZ R3, FOO	18			

（4）

名称	保留站							
	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Add1	yes	ADD.D	Regs[F4]	Regs[F6]				
Add2	no							
Add3	no							
Mult1	yes							
Mult2	no							

项号	ROB				
	Busy	指令	状态	目的	Value
1	yes	ADD.D F6, F4, F6	执行	F6	Regs[F4]＋Regs[F6]
2	yes	S.D F6, 0（R2）	流出	Mem[0+Regs[R2]]	#2

字段	浮点寄存器状态							
	F0	F2	F4	F6	F8	F10	…	F30
ROB项编号				1				
Busy				yes			…	

（5）

整数指令	浮点指令	时钟周期数
L.D F2, 0(R1)		1
L.D F8, 8(R1)		2
L.D F14, 16(R1)	MUT.D F4, F2, F0	3
L.D F20, 24(R1)	MUT.D F10, F8, F0	4
L.D F6, 0(R2)	MUT.D F16, F14, F0	5
L.D F12, 8(R2)	MUT.D F22, F20, F0	6
L.D F18, 16(R2)	ADD.D F6, F4, F6	7
L.D F24, 24(R2)	ADD.D F12, F10, F12	8
DADDIU R1, R1, #32	ADD.D F18, F16, F18	9
S.D F6, 0(R2)	ADD.D F24, F22, F24	10
S.D F12, 8(R2)		11
S.D F18,16(R2)		12
S.D F24, 24(R2)		13
DADDIU R2, R2, #32		14
DSUBIU R3, R1, #DONE		15
BNEZ R3, FOO		16

计算一个Y值需要 16/4 = 4 个时钟周期，加速比 = 14/4 = 3.5

（6） 对于如图 4.19 结构的超长指令字 DLX 处理器，将 SAXPY 循环展开 4 次，优化和调度指令，使 SAXPY 循环处理时间达到最优。计算一个 Y 值平均需要多少时间？加速比是多少？

解：

访存 1	访存 2	浮点操作 1	浮点操作 2	整数操作	时钟周期
LD f2, 0 (r1)	LD f8, 8 (r1)				1
LD f14, 16 (r1)	LD f20, 24 (r1)				2
LD f6, 0 (r2)	LD f12, 8 (r2)	MULTD f4,f2, f0	MULTD f10,f8,f0		3
LD f18, 16 (r2)	LD f24, 24 (r2)	MULTD f16,f14,f0	MULTD f22,f20,f0		4
					5
					6
		ADDD f6, f4, f6	ADDD f12,f10,f12		7
		ADDD f18,f16,f18	ADDD f24,f22,f24	ADDI r1,r1,#32	8
				ADDI r2,r2,#32	9
SD 32 (r2), f6	SD 24 (r2), f12			SGTI r3,r1,done	10
SD 16 (r2), f18	SD 8 (r2), f24			BEQZ r3, FOO	11

计算一个 Y 值需要 11/4 个时钟周期，加速比 = 14/(11/4) = 56/11

访存1	访存2	浮点指令1	浮点指令2	整数指令	时钟周期
L.DF2, 0(R1)	L.D F8, 8(R1)				1
L.DF14, 16(R1)	L.DF20, 24(R1)				
L.DF6, 0(R2)	L.DF12, 8(R2)	MUT.DF4, F2, F0	MUT.DF10, F8, F0		3
L.DF18, 16(R2)	L.DF24, 24(R2)	MUT.DF16, F14, F0	MUT.DF22, F20, F0		4
		ADD.DF6, F4, F6	ADD.DF12, F10, F12		5
		ADD.DF18, F16, F18	ADD.DF24, F22, F24	DADDIU R1, R1, #32	6
				DADDIU R2, R2, #32	7
				DSUBIUR3, R1, #DONE	8
				BNEZ R3, FOO	9
S.DF6, -32(R2)	S.DF12, -24(R2)				10
S.DF18,-16(R2)	S.DF24, -8(R2)				11

计算一个Y值需要 11/4 个时钟周期，加速比 = 14/(11/4) = 56/11