

第3章 流水线技术

- 3. 1 [重叠执行和先行控制](#)
- 3. 2 [流水线的基本概念](#)
- 3. 3 [流水线的性能指标](#)
- 3. 4 [流水线的相关与冲突](#)
- 3. 5 [流水线的实现](#)
- 3. 6 [向量处理机](#)

3.1 重叠执行和先行控制

3.1.1 重叠执行

1. 将一条指令的执行过程分为三个阶段



一条指令的执行过程

➤ 取指令

- 按照指令计数器PC的内容访问主存，取出一条指令送到指令寄存器。

➤ 指令分析

- 对指令的操作码进行译码，按照给定的寻址方式和地址字段形成操作数的地址，并用这个地址读取操作数。

➤ 指令执行

- 按照操作码的要求，完成指令规定的功能。

在指令的执行过程中还要更新PC值，为读取下一条指令做好准备。

2. 三种执行方式

- 顺序执行方式
- 一次重叠执行方式
- 二次重叠执行方式

3. 顺序执行方式

➤ 指令的执行过程



➤ 执行 n 条指令所花的时间

$$T = \sum_{i=1}^n (t_{\text{取指令}i} + t_{\text{分析}i} + t_{\text{执行}i})$$

- 如果取指令、指令分析和指令执行的时间相等，都是 t ，则

$$T=3nt$$

- 优点

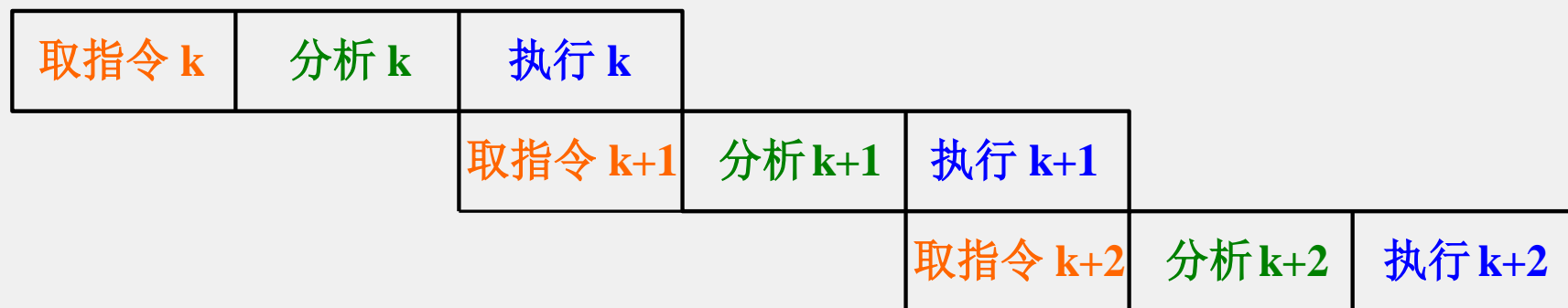
- 控制简单，节省设备。

- 主要缺点

- 处理机执行指令的速度慢
 - 功能部件的利用率很低

4. 一次重叠执行方式

➤ 指令的执行过程



执行第 k 条指令与取第 $k+1$ 条指令同时进行。

(一种最简单的重叠方式)

- 如果执行一条指令的3个阶段的时间相等，都是 t ，则执行 n 条指令所花的时间为

$$T = (1 + 2n) t$$

- 优点

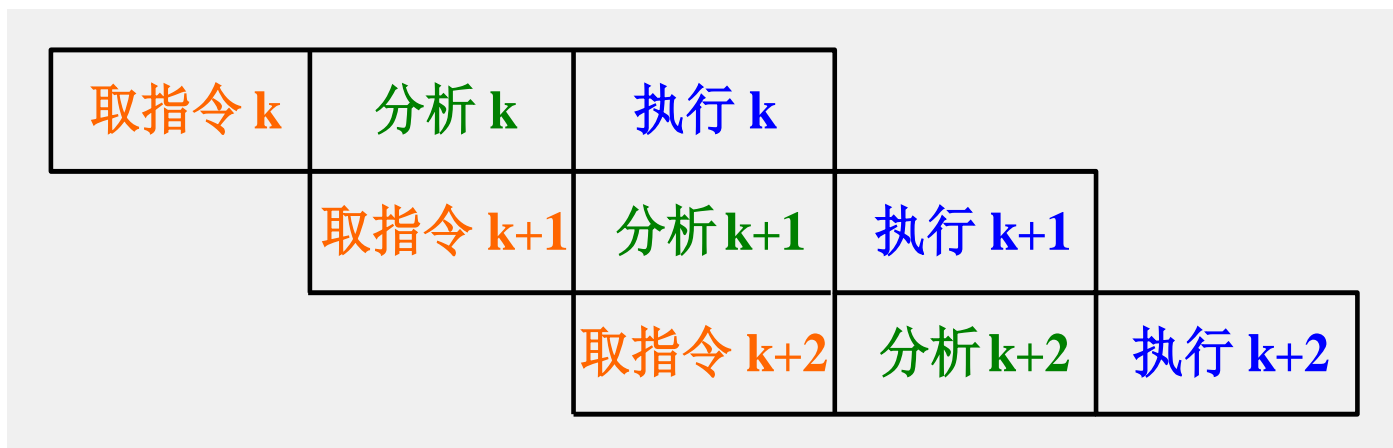
- 程序的执行时间减少了近1/3。
- 功能部件的利用率明显提高。

- 缺点

- 需要增加一些硬件，控制过程变复杂了。需要增加一个指令寄存器,原来的指令寄存器存储当前正执行的第K条指令,新增的存储取出来的第K+1条指令。

5. 二次重叠执行方式

➤ 指令的执行过程



取第 $k+1$ 条指令提前到与分析第 k 条指令同时进行，
分析第 $k+1$ 条指令与执行第 k 条指令同时进行。

- 如果执行一条指令的3个阶段的时间相等，都是 t ，则执行 n 条指令所花的时间为

$$T = (2+n) t$$

- 优点

- 与顺序执行方式相比，执行时间缩短了近 $2/3$ 。
- 部件的利用率有了进一步的提高。

- 缺点

- 需要增加更多的硬件。
- 需要设置独立的取指令部件、指令分析部件和指令执行部件。要解决好访问主存的冲突问题。
- 原因是取指令要访问主存,分析指令是可能从主存取操作数,执行指令时可能要写结果到主存。

➤ 访问主存的冲突问题

4种解决方法

- 设置两个独立编址的存储器：
指令存储器（存放指令）、数据存储器（存放数据）
- 指令和数据仍然混合存放在同一个主存中，但设置两个Cache：

指令Cache、数据Cache

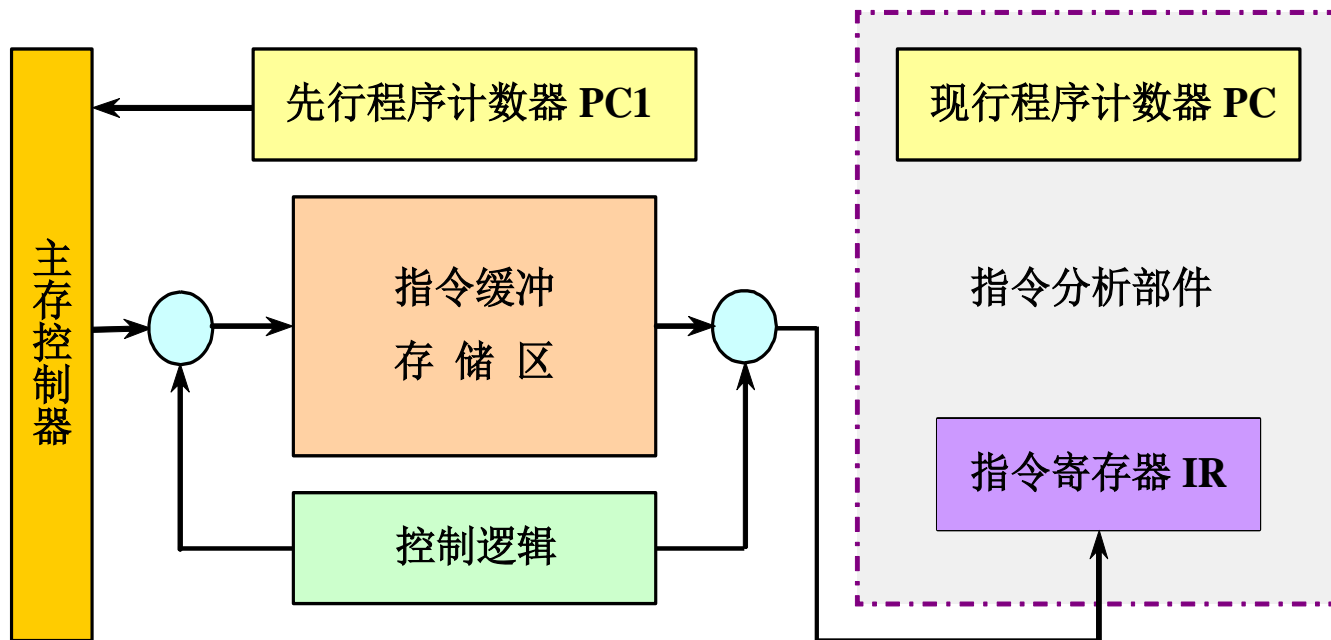
程序空间和数据空间相互独立的系统结构被称为哈佛结构。

- 指令和数据仍然混合存放在同一个主存中，但主存采用多体交叉结构。取指令和读操作数访问不同的存储体,可以实现指令重叠执行。
(有一定的局限性)

- 在主存和指令分析部件之间增设**指令缓冲站**
（又被称为**先行指令缓冲站**）
 - 有些指令将结果存在通用寄存器, 在执行阶段不访问主存. 主存不是满负荷工作的, 插空从主存中预先把后面将要执行的指令取出来, 存放到指令缓冲站中。
 - 在“取指令”阶段从指令缓冲站读取指令（如果指令缓冲站不为空），而不用去访问主存。

6. 先行指令缓冲站

➤ 先行指令缓冲站的组成



包含一个指令缓冲存储区和控制逻辑，按队列方式保证原来指令顺序。只要指令缓冲站不满,就发出请求预取指令.指令分析部件每分析完一条,就向指令缓冲站发出取下一条指令的请求.取出后在指令缓冲站的该指令作废.指令缓冲站存放的指令条数动态变化.

- 指令缓冲存储区和相应的控制逻辑
 - 按队列方式工作。
 - 只要指令缓冲站不满，它就自动地向主存控制器发取指令请求，不断地预取指令。
- 指令分析部件
 - 每分析完一条指令，就自动向指令缓冲站发出取下一条指令的请求。指令取出之后就把指令缓冲站中的该指令作废。
 - 指令缓冲站中存放的指令的条数是动态变化的。
- 两个程序计数器

- 先行程序计数器PC1：用于从主存预取指令；
- 现行程序计数器PC：用来记录指令分析部件当前正在分析的指令的地址。

7. 先行控制方式中的一次重叠执行

- 若取指令阶段的时间很短，可以把这个操作合并到分析指令中。
- 上述的二次重叠就演变成了一次重叠
 - 把一条指令的执行过程分为分析和执行两个阶段；
 - 让前一条指令的执行与后一条指令的分析重叠进行。

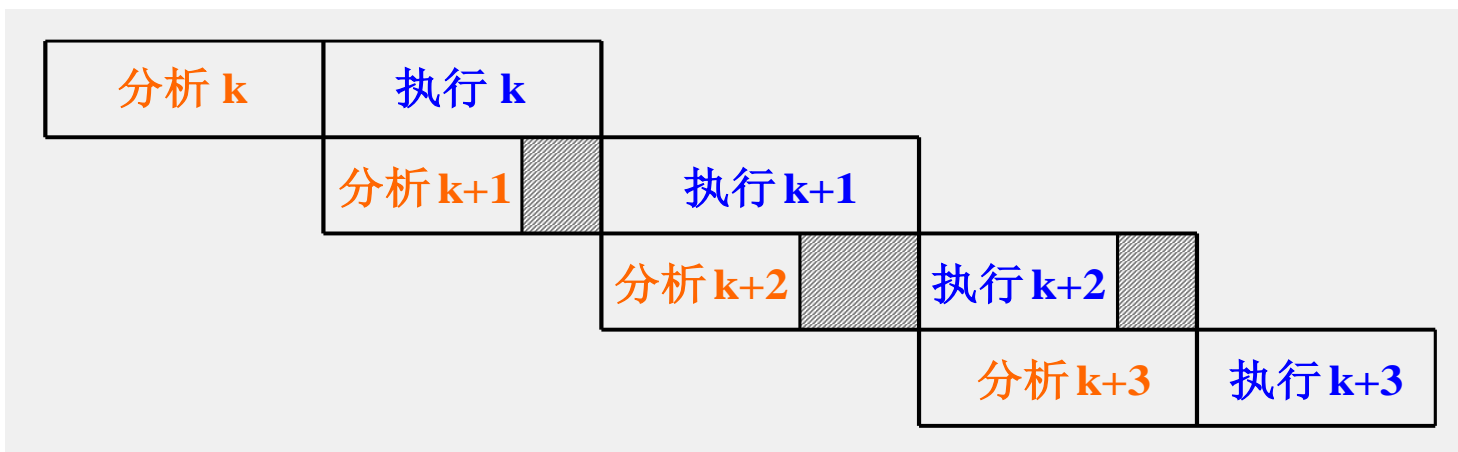


- 如果指令分析和指令执行所需要的时间都是 t ，则采用这种方式连续执行 n 条指令所需要的时间为：

$$T = (1+n) t$$

- 控制方式比较简单，得到了广泛应用。

- 当指令分析和指令执行所需要的时间不相等时，其执行过程为：



指令分析部件和指令执行部件存在相互等待的时候，会出现部件空闲的情况。为充分发挥这两套部件的作用，采用多种缓冲技术。

3.1.2 先行控制

1. **先行控制技术：缓冲技术和预处理技术的结合**
 - **缓冲技术：**在工作速度不固定的两个功能部件之间设置缓冲器，用以平滑它们的工作。
 - **预处理技术：**预取指令、对指令进行加工以及预取操作数等。
2. **采用先行控制方式的处理机结构**

➤ 设置了4个缓冲站

（平滑主存、指令分析部件、运算器三者之间的工作）

- 先行指令缓冲站（前面已讲述）
- 先行操作站
- 先行读数站
- 后行写数站

共同特点：按先进先出的方式工作，而且都是由一组若干个能快速访问的存储单元和相关的控制逻辑组成。独立工作,不互相等待,提高指令执行速度和部件效率。

➤ 先行操作站

- 在指令分析部件和运算器之间提供缓冲
- **先行**：因为其中的指令对于运算器正在执行的指令来说是后续的，但却被**先行**取出并预处理。

➤ 指令分析部件

- 从先行指令缓冲站取指令，并进行预处理，加工成统一格式的**RR型操作命令**，然后送入先行操作站。
- 对于不同指令做不同的处理。

- ❑ **寄存器-寄存器型（RR型）指令：**可以不作任何处理，直接送入。
- ❑ **操作数来自主存的运算指令：**计算出操作数的有效地址，并将该地址送入先行读数站的某个存储单元（设其地址为 i ），同时用 i 替换原来指令中的操作数地址码字段。
- ❑ **向主存“写数”的指令：**把形成的有效地址送入后行写数站的某个存储单元（设其地址为 j ），同时用 j 替换原来指令中的目标地址码字段。
- ❑ **立即数型指令：**把指令中的立即数送入读数站（设为第 l 个存储单元），同样也用 l 替换原来指令中的立即数字段。

➤ 运算器

- 从先行操作站取出RR型操作命令并执行。
- 每执行完一条，将运算结果写入通用寄存器组或者后行写数站。
- 继续执行先行操作站中的后续命令。

➤ 先行读数站

- **作用：**接收指令分析部件送来的访问主存的有效地址，按顺序依次从主存读取操作数，提供给运算器使用。
- **先行：**因为对于正在执行的指令来说，先行读数站中的操作数是先行取出的。

- 每个存储单元由3部分组成：

先行地址字段、先行操作数字段、标志字段

- 每当从指令分析部件接收有效地址时，将之放入先行地址字段，并将地址有效标志置位。
- 等到该单元成为队列的第一项时，先行读数站会用该地址向主存发出读请求，把取来的操作数放入该单元的先行操作数字段，同时将数据有效标志置位。
- 当以后运算器需要该操作数时，就可以直接从先行读数站取得，而不必去访问主存。

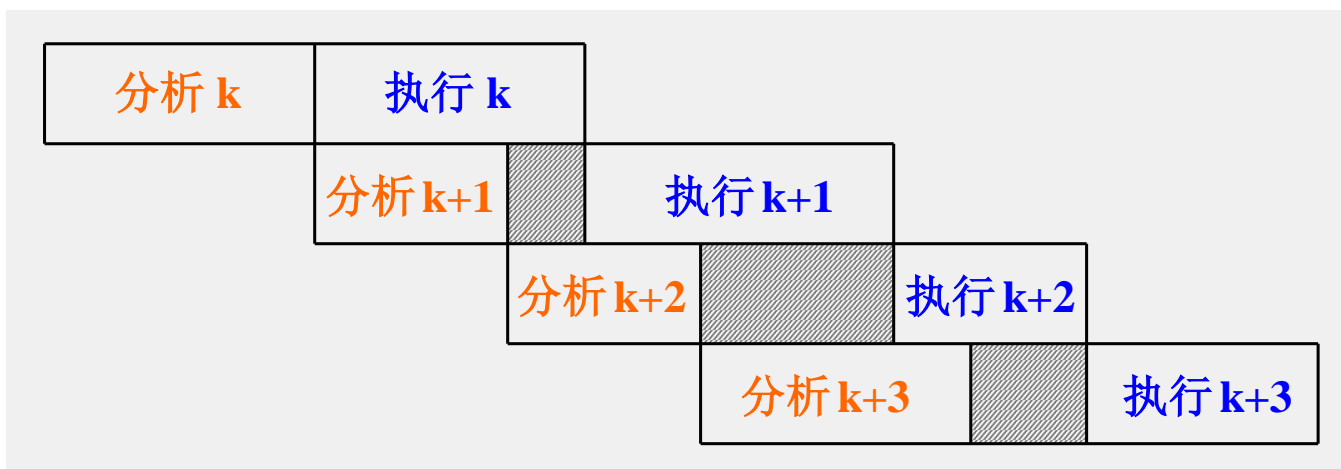
➤ 后行写数站

- **作用：**接收从运算器送来的结果数据，并负责将之写入主存。
- **后行：**因为站在运算器的角度来看，结果数据不是在相应的指令运算完后立即写入主存，而是由后行写数站**滞后**写入的。
- 每一个存储单元由**3**部分组成：

后行地址字段、后行数据字段、标志字段

每当从运算器接收数据时，将之放入后行数据字段，并把相应的数据有效标志置位。后行写数站的控制逻辑自动向主存发出写数请求。当写数据操作完成后，也要置位有关标志。

3. 采用先行控制后的一次重叠执行



指令分析部件在不间断地分析指令，而指令执行部件则在不间断地执行指令，它们都始终处于忙碌状态。

- 理想情况下，指令执行部件应该是一直忙碌的。
- 处理机连续执行 n 条指令所需要的时间为

$$T_{\text{先行}} = t_{\text{分析1}} + \sum_{i=1}^n t_{\text{执行}i} \approx \sum_{i=1}^n t_{\text{执行}i}$$

3.2 流水线的基本概念

3.2.1 什么是流水线

1. 工业生产流水线

下面通过一个例子来说明流水线的好处：

- [两种方案](#)
- [两种方案的工作过程对比](#)
- [流水线生产过程的抽象描述](#)
- [这种流水工作方式的主要特点](#)

2. 流水线技术

- 把一个重复的过程分解为若干个子过程，每个子过程由专门的功能部件来实现。
- 把多个处理过程在时间上错开，依次通过各功能段，这样，每个子过程就可以与其他的子过程并行进行。

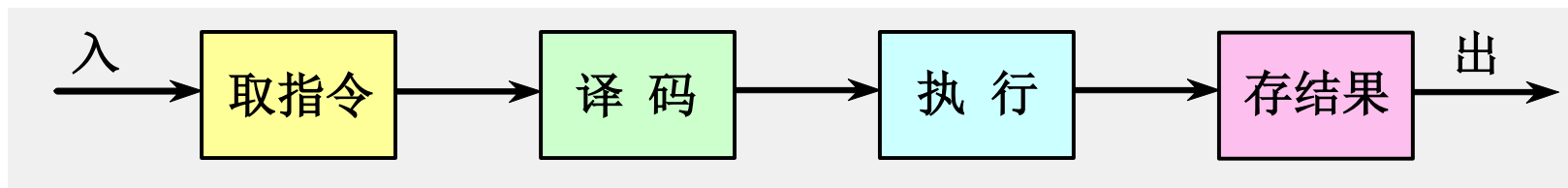
3. 流水线中的每个子过程及其功能部件称为流水线的级或段，段与段相互连接形成流水线。流水线的段数称为流水线的深度。

4. 指令流水线

- 把指令的解释过程分解为分析和执行两个子过程，并让这两个子过程分别用独立的分析部件和执行部件来实现。

理想情况：速度提高一倍

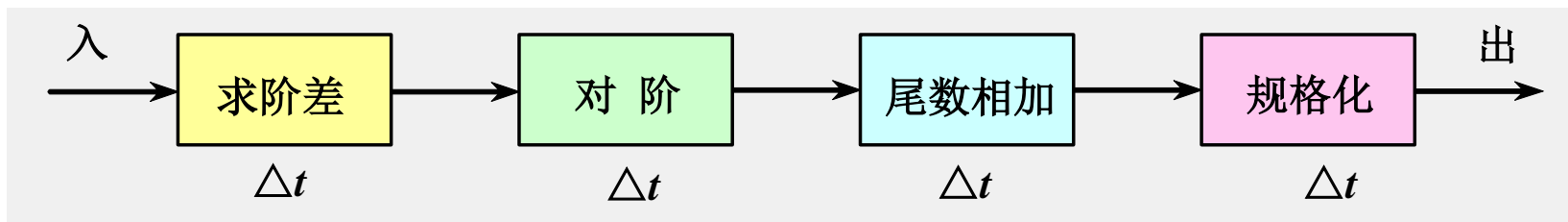
- 4段指令流水线



5. 浮点加法流水线

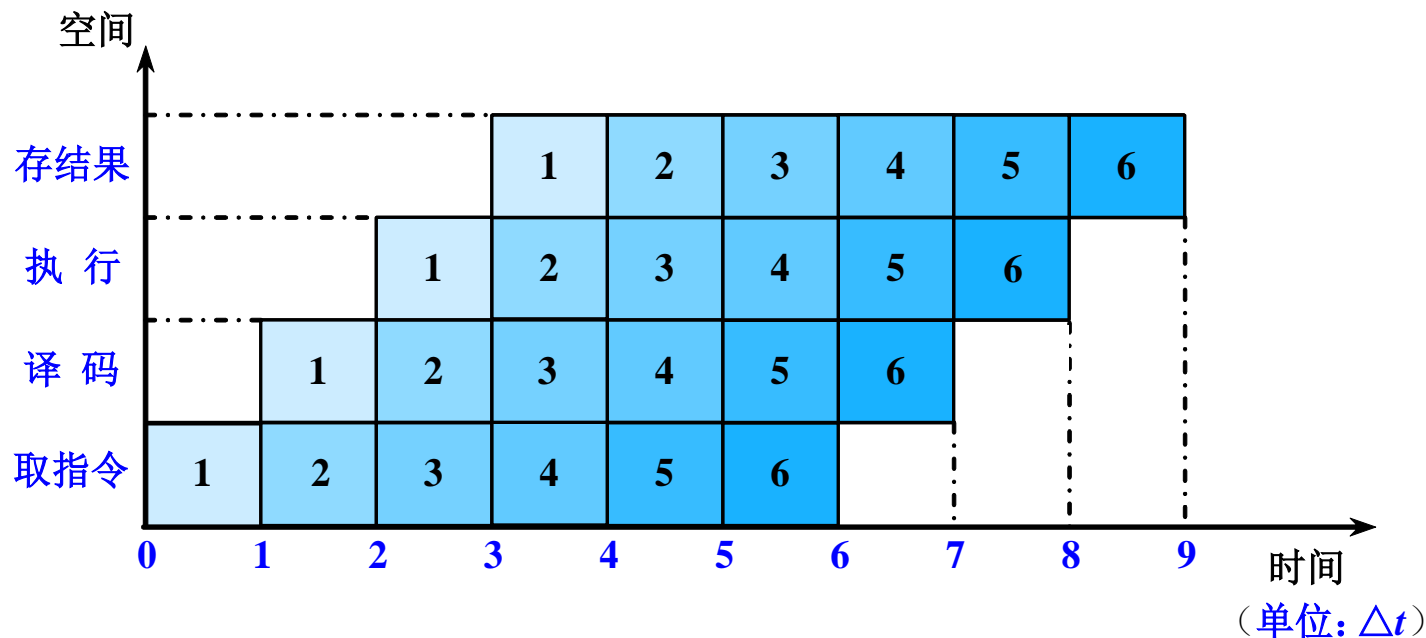
- 把流水线技术应用于运算的执行过程，就形成了运算操作流水线，也称为部件级流水线。
- 把浮点加法的全过程分解为求阶差、对阶、尾数相加、规格化4个子过程。

理想情况：速度提高3倍



6. 时空图

- 时空图从时间和空间两个方面描述了流水线的工作过程。时空图中，横坐标代表时间，纵坐标代表流水线的各个段。
- 4段指令流水线的时空图



7. 流水技术的特点

- 流水线把一个处理过程分解为若干个子过程（段），每个子过程由一个专门的功能部件来实现。
- 流水线中各段的时间应尽可能相等，否则将引起流水线堵塞、断流。
 - 时间长的段将成为流水线的瓶颈。
- 流水线每一个功能部件的后面都要有一个缓冲寄存器（锁存器），称为流水寄存器。
 - 作用：在相邻的两段之间传送数据，以保证提供后面要用到的数据，并把各段的处理工作相互隔离。

- 流水技术适合于大量重复的时序过程，只有在输入端不断地提供任务，才能充分发挥流水线的效率。
- 流水线需要有通过时间和排空时间。
 - **通过时间**：第一个任务从进入流水线到流出结果所需的时间。
 - **排空时间**：最后一个任务从进入流水线到流出结果所需的时间。

3.2.2 流水线的分类

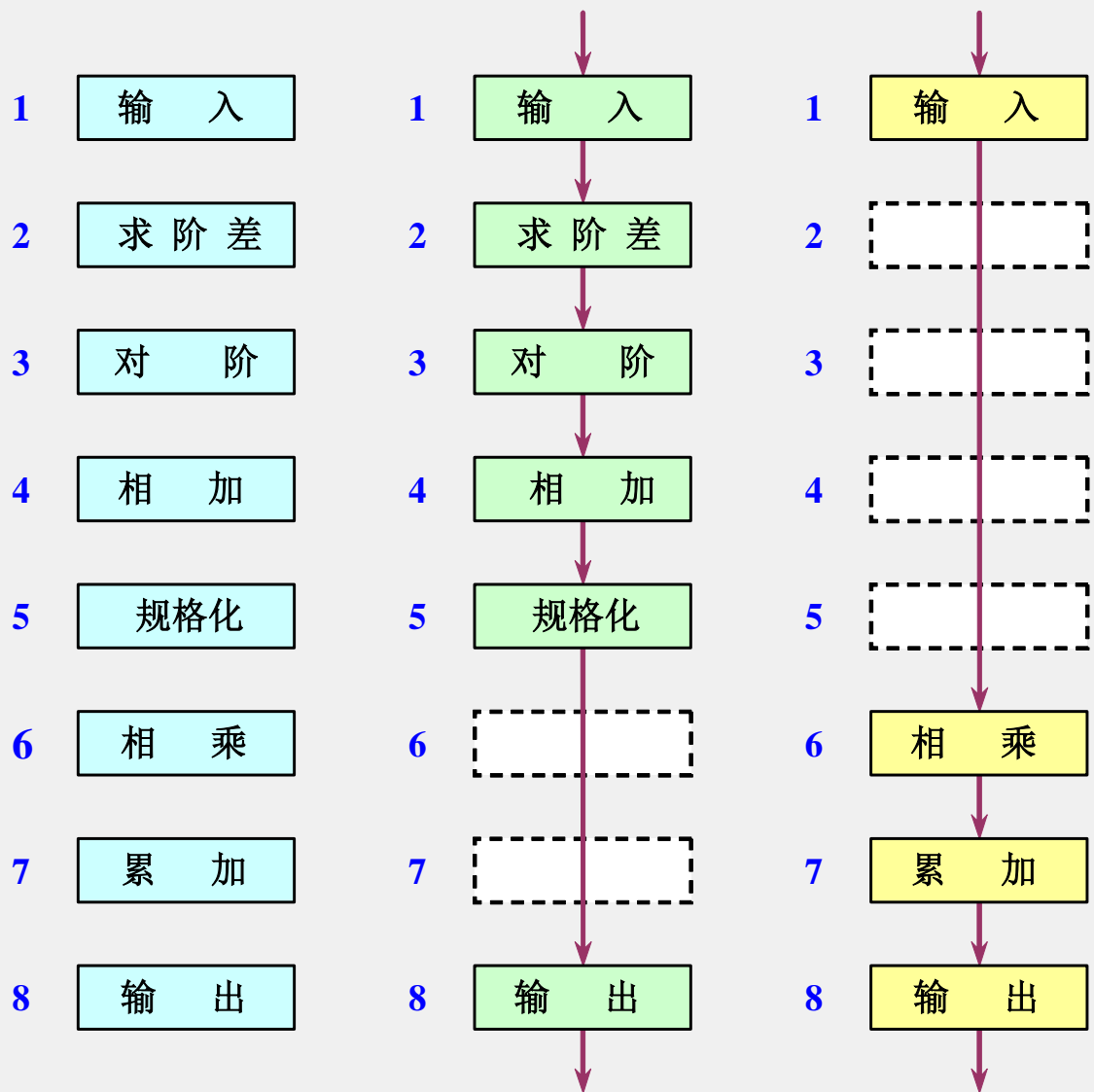
从不同的角度和观点，把流水线分成多种不同的种类。

1. 单功能流水线与多功能流水线

（按照流水线所完成的功能来分类）

- **单功能流水线：**只能完成一种固定功能的流水线。
- **多功能流水线：**流水线的各段可以进行不同的连接，以实现不同的功能。

例： [ASC的多功能流水线](#)



(a) 分段

(b) 浮点连接

(c) 定乘连接

2. 静态流水线与动态流水线

（按照同一时间内各段之间的连接方式对多功能流水线做进一步的分类）

➤ **静态流水线：**在同一时间内，多功能流水线中的各段只能按同一种功能的连接方式工作。

- 对于静态流水线来说，只有当输入的是一串相同的运算任务时，流水的效率才能得到充分的发挥。

例如：[ASC的8段流水线](#)

- **动态流水线：**在同一时间内，多功能流水线中的各段可以按照不同的方式连接，同时执行多种功能。
[动画](#)
 - **优点**

灵活，能够提高流水线各段的使用率，从而提高处理速度。
 - **缺点**

控制复杂。
- [静、动态流水线时空图的对比](#)

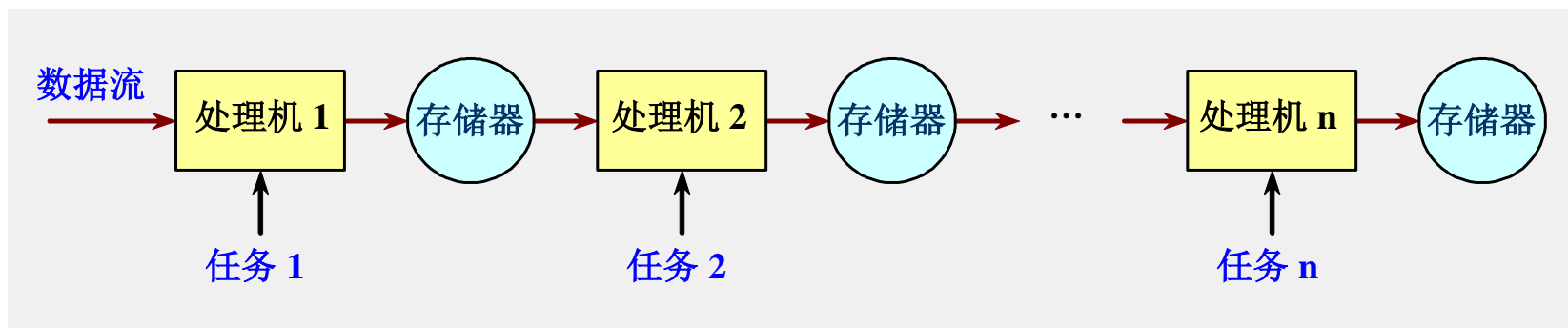
3. 部件级、处理机级及处理机间流水线

（按照流水的级别来进行分类）

- **部件级流水线**（运算操作流水线）：把处理机的算术逻辑运算部件分段，使得各种类型的运算操作能够按流水方式进行。
- **处理机级流水线**（指令流水线）：把指令的解释执行过程按照流水方式处理。把一条指令的执行过程分解为若干个子过程，每个子过程在独立的功能部件中执行。

例如：前面的4段指令流水线

- **处理机间流水线**（宏流水线）：它是由两个或者两个以上的处理机串行连接起来，对同一数据流进行处理，每个处理机完成整个任务中的一部分。 [动画解析](#)



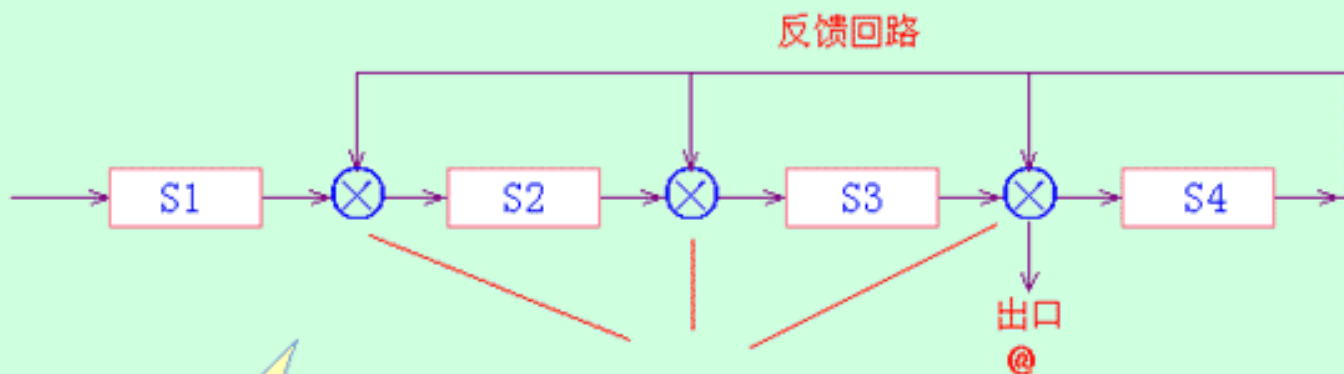
4. 线性流水线与非线性流水线

（按照流水线中是否有反馈回路来进行分类）

- **线性流水线：**流水线的各段串行连接，没有反馈回路。数据通过流水线中的各段时，每一个段最多只流过一次。
- **非线性流水线：**流水线中除了有串行的连接外，还有反馈回路。（[举例](#)）
- [非线性流水线的调度问题](#)
 - 确定什么时候向流水线引进新的任务，才能使该任务不会与先前进入流水线的任务发生冲突——争用流水段。

非线性流水线

(举例)



虽然流水线仅由四段构成，
但有些段可能要重复通过。

串行连接

例如任务 @：

→ S1 → S2 → S3 → S4 → S2 → S3 → S4 → S3 →

5. 顺序流水线与乱序流水线

（根据任务流入和流出的顺序是否相同来进行分类）

- **顺序流水线：**流水线输出端任务流出的顺序与输入端任务流入的顺序完全相同。每一个任务在流水线的各段中是一个跟着一个顺序流动的。
- **乱序流水线：**流水线输出端任务流出的顺序与输入端任务流入的顺序可以不同，允许后进入流水线的任务先完成（从输出端流出）。

也称为无序流水线、错序流水线、异步流水线

6. 标量处理机与向量流水处理机

- 把指令执行部件中采用了流水线的处理机称为**流水线处理机**。
- **标量处理机**：处理机不具有向量数据表示和向量指令，仅对标量数据进行流水处理。
- **向量流水处理机**：具有向量数据表示和向量指令的处理机。

向量数据表示和流水技术的结合。

3.3 流水线的性能指标

3.3.1 吞吐率

吞吐率：在单位时间内流水线所完成的任务数量或输出结果的数量。

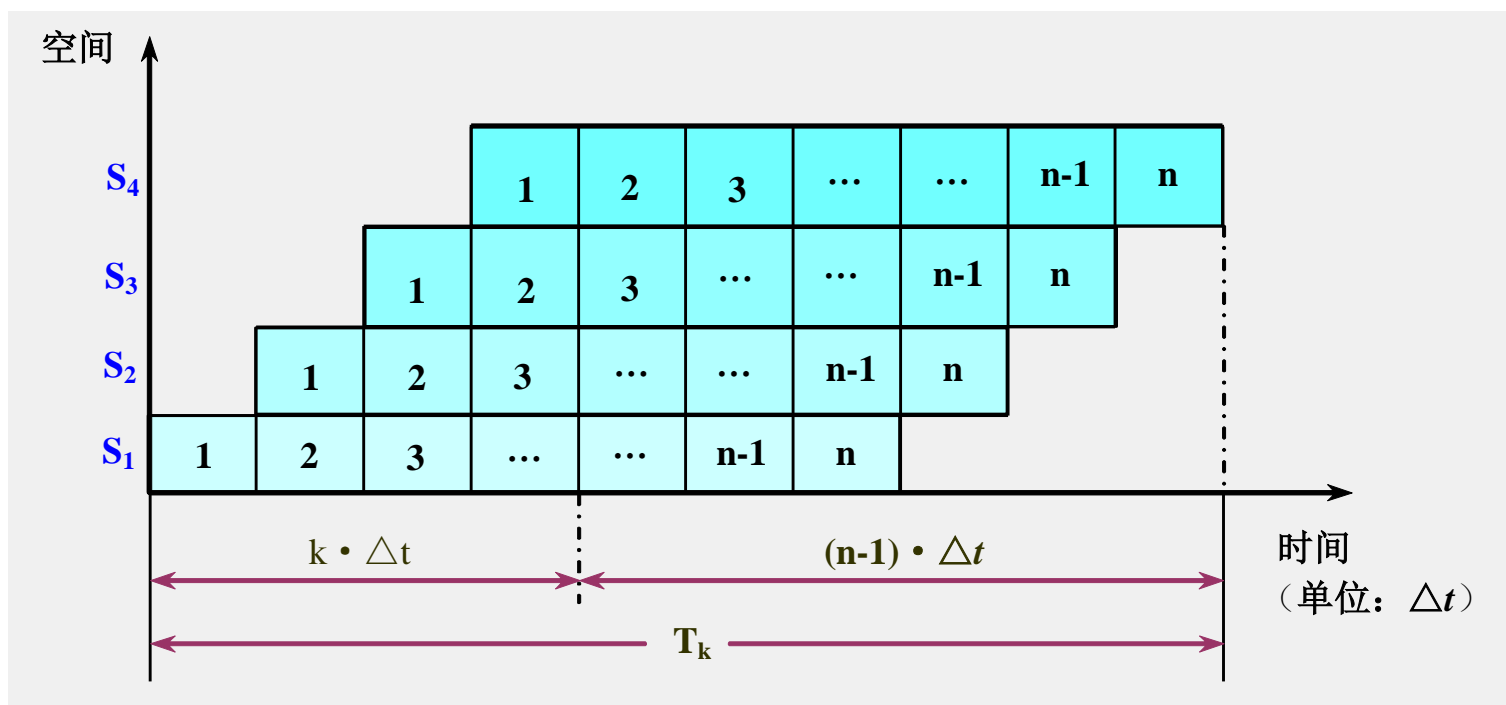
$$TP = \frac{n}{T_K}$$

n ：任务数

T_k ：处理完成 n 个任务所用的时间

1. 各段时间均相等的流水线

➤ 各段时间均相等的流水线时空图



- 流水线完成 n 个连续任务所需要的总时间为
(假设一条 k 段线性流水线)

$$T_k = k \Delta t + (n-1) \Delta t = (k+n-1) \Delta t$$

- 流水线的实际吞吐率

$$TP = \frac{n}{(k+n-1)\Delta t}$$

- 最大吞吐率

$$TP_{\max} = \lim_{n \rightarrow \infty} \frac{n}{(k+n-1)\Delta t} = \frac{1}{\Delta t}$$

➤ 最大吞吐率与实际吞吐率的关系

$$TP = \frac{n}{k + n - 1} TP_{\max}$$

- 流水线的实际吞吐率小于最大吞吐率，它除了与每个段的时间有关外，还与流水线的段数 k 以及输入到流水线中的任务数 n 等有关。
- 只有当 $n \gg k$ 时，才有 $TP \approx TP_{\max}$ 。

2. 各段时间不完全相等的流水线

➤ 各段时间不等的流水线及其时空图

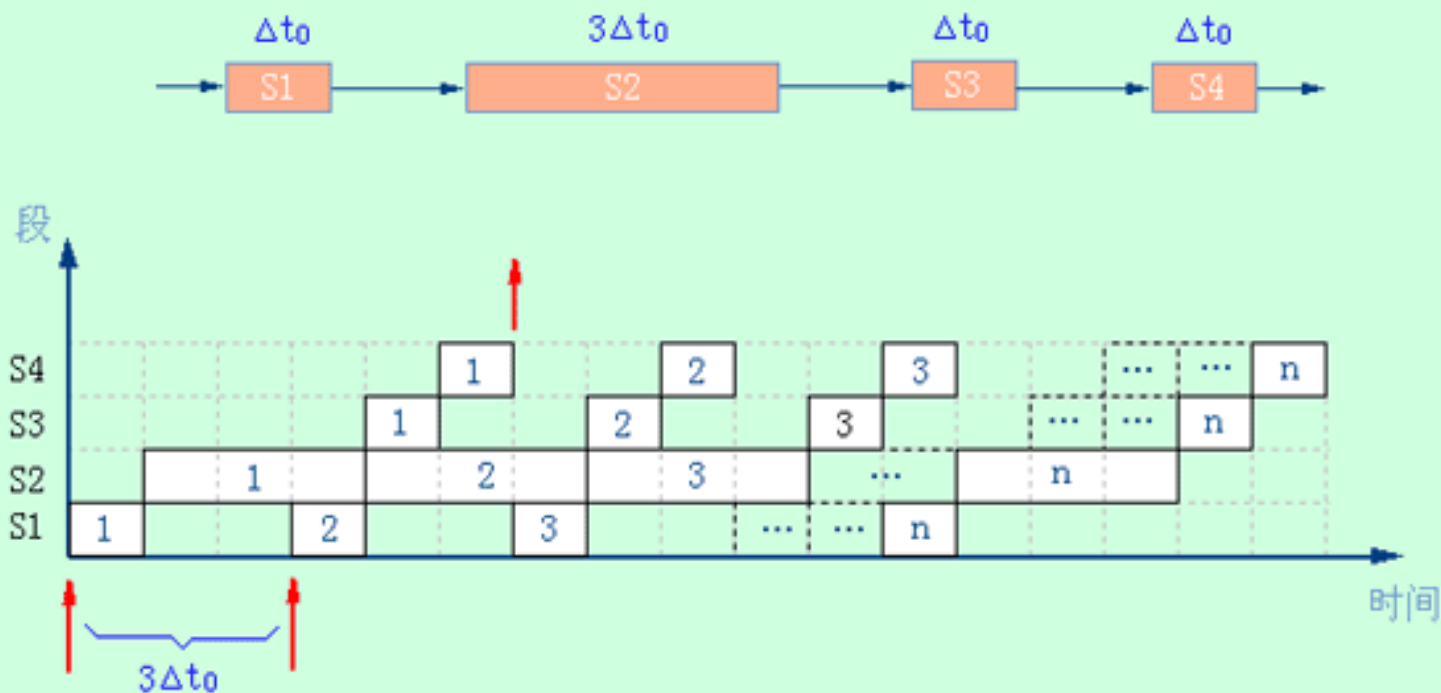
举例（时空图）

- 一条4段的流水线
- S1, S3, S4各段的时间: Δt
- S2的时间: $3\Delta t$ （瓶颈段）

流水线中这种时间最长的段称为流水线的**瓶颈段**。

流水线的时-空图

(各段时间不等)



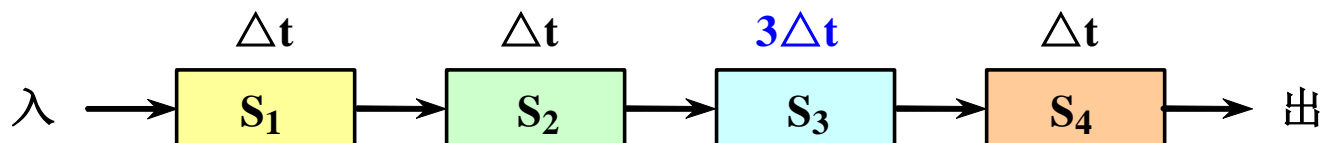
- 各段时间不等的流水线的实际吞吐率：
(Δt_i 为第 i 段的时间，共有 k 个段)

$$TP = \frac{n}{\sum_{i=1}^k \Delta t_i + (n-1) \max(\Delta t_1, \Delta t_2, \dots, \Delta t_k)}$$

- 流水线的最大吞吐率为

$$TP_{\max} = \frac{1}{\max(\Delta t_1, \Delta t_2, \dots, \Delta t_k)}$$

例如：一条4段的流水线中， S_1 ， S_2 ， S_4 各段的时间都是 Δt ，唯有 S_3 的时间是 $3\Delta t$ 。



最大吞吐率为

$$TP_{\max} = \frac{1}{3\Delta t}$$

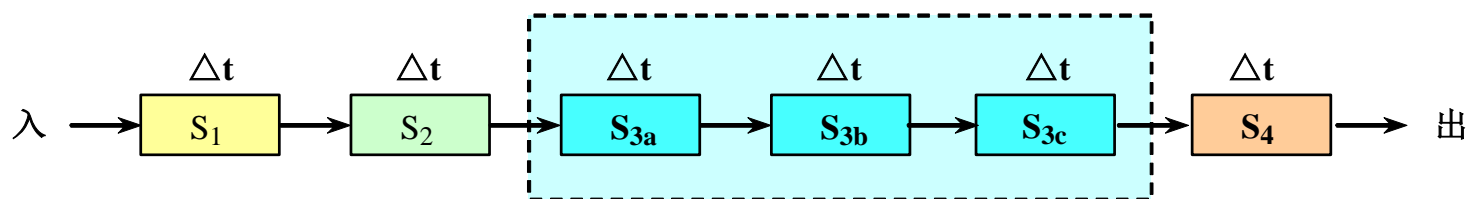
3. 解决流水线瓶颈问题的常用方法

举例

➤ 细分瓶颈段

例如：对前面的4段流水线

把瓶颈段 S_3 细分为3个子流水线段： S_{3a} ， S_{3b} ， S_{3c}



改进后的流水线的吞吐率：

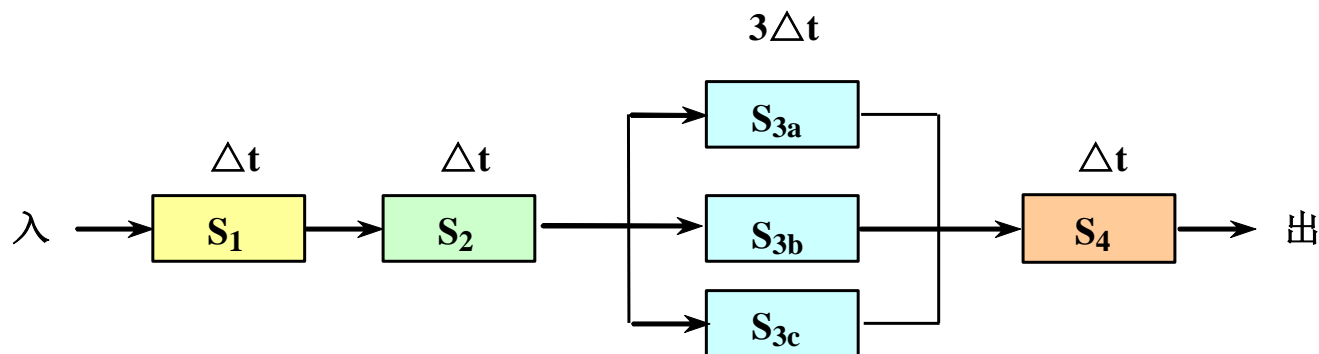
$$TP_{\max} = \frac{1}{\Delta t}$$

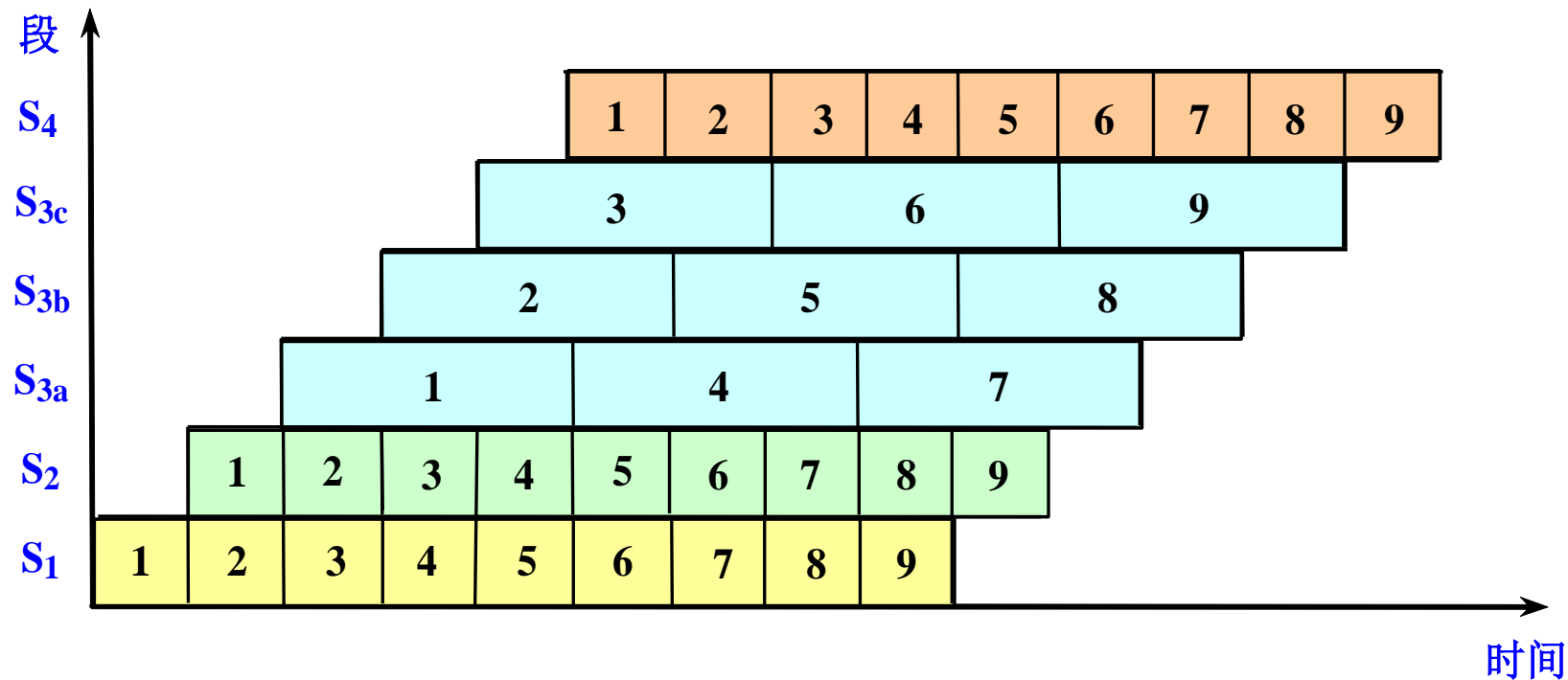
➤ 重复设置瓶颈段

- 举例：时空图
- 缺点：控制逻辑比较复杂，所需的硬件增加了。

例如：对前面的4段流水线

重复设置瓶颈段 S_3 ： S_{3a} ， S_{3b} ， S_{3c}





重复设置瓶颈段后的时空图

3.3.2 加速比

加速比：完成同样一批任务，不使用流水线所用的时间与使用流水线所用的时间之比。

假设：不使用流水线（即顺序执行）所用的时间为 T_s ，使用流水线后所用的时间为 T_k ，则该流水线的加速比为

$$S = \frac{T_s}{T_k}$$

1. 流水线各段时间相等（都是 Δt ）

- 一条 k 段流水线完成 n 个连续任务
所需要的时间为

$$T_k = (k + n - 1)\Delta t$$

- 顺序执行 n 个任务

所需要的时间： $T_s = nk\Delta t$ （解释）

- 流水线的实际加速比为

$$S = \frac{nk}{k + n - 1}$$

➤ 最大加速比

$$S_{\max} = \lim_{n \rightarrow \infty} \frac{nk}{k + n - 1} = k$$

当 $n \gg k$ 时, $S \approx k$

思考：流水线的段数愈多愈好？

2. 流水线的各段时间不完全相等时

- 一条 k 段流水线完成 n 个连续任务的实际加速比为

$$S = \frac{n \sum_{i=1}^k \Delta t_i}{\sum_{i=1}^k \Delta t_i + (n-1) \max(\Delta t_1, \Delta t_2, \dots, \Delta t_k)}$$

3.3.3 效率

效率：流水线中的设备实际使用时间与整个运行时间的比值，即流水线设备的利用率。

由于流水线有通过时间和排空时间，所以在连续完成 n 个任务的时间内，各段并不是满负荷地工作。

1. 各段时间相等

- 各段的效率 e_i 相同
(解释)

$$e_1 = e_2 = \cdots = e_k = \frac{n\Delta t}{T_k} = \frac{n}{k + n - 1}$$

- 整条流水线的效率为

$$E = \frac{e_1 + e_2 + \cdots + e_k}{k} = \frac{ke_1}{k} = \frac{kn\Delta t}{kT_k}$$

- 可以写成

$$E = \frac{n}{k + n - 1}$$

- 最高效率为

$$E_{\max} = \lim_{n \rightarrow \infty} \frac{n}{k + n - 1} = 1$$

当 $n \gg k$ 时, $E \approx 1$ 。

- 当流水线各段时间相等时，流水线的效率与吞吐率成正比。

$$E = TP \Delta t$$

2. 流水线的效率是流水线的实际加速比 S 与它的最大加速比 k 的比值。

$$E = \frac{S}{k}$$

当 $E=1$ 时， $S=k$ ，实际加速比达到最大。

3. 从时空图上看，效率就是 n 个任务占用的时空面积和 k 个段总的时空面积之比。

$$E = \frac{n \text{ 个任务实际占用的时空区}}{k \text{ 个段总的时空区}}$$

当各段时间不相等时

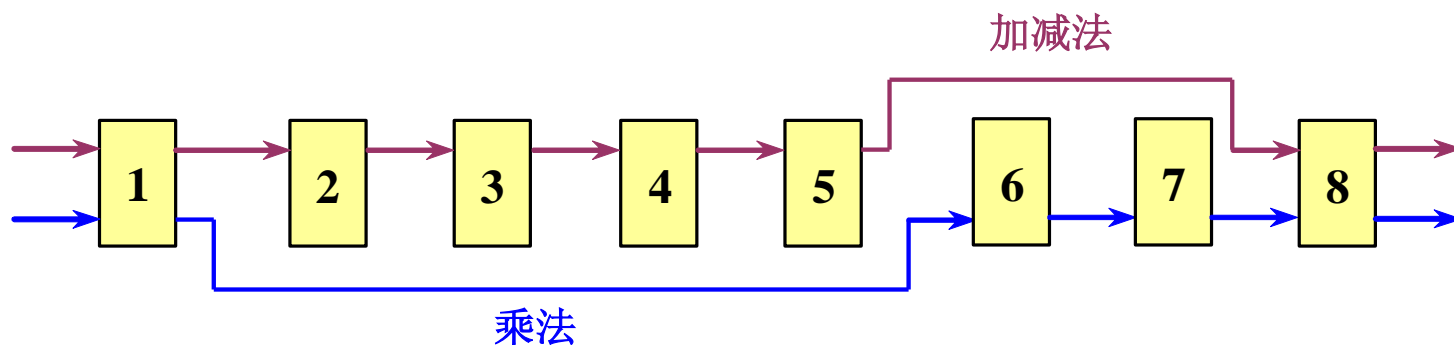
$$E = \frac{n \cdot \sum_{i=1}^k \Delta t_i}{k \left[\sum_{i=1}^k \Delta t_i + (n-1) \cdot \max(\Delta t_1, \Delta t_2, \dots, \Delta t_k) \right]}$$

3.3.4 流水线的性能分析举例

例3.1 设在下图所示的静态流水线上计算：

$$\prod_{i=1}^4 (A_i + B_i)$$

流水线的输出可以直接返回输入端或暂存于相应的流水寄存器中，试计算其吞吐率、加速比和效率。



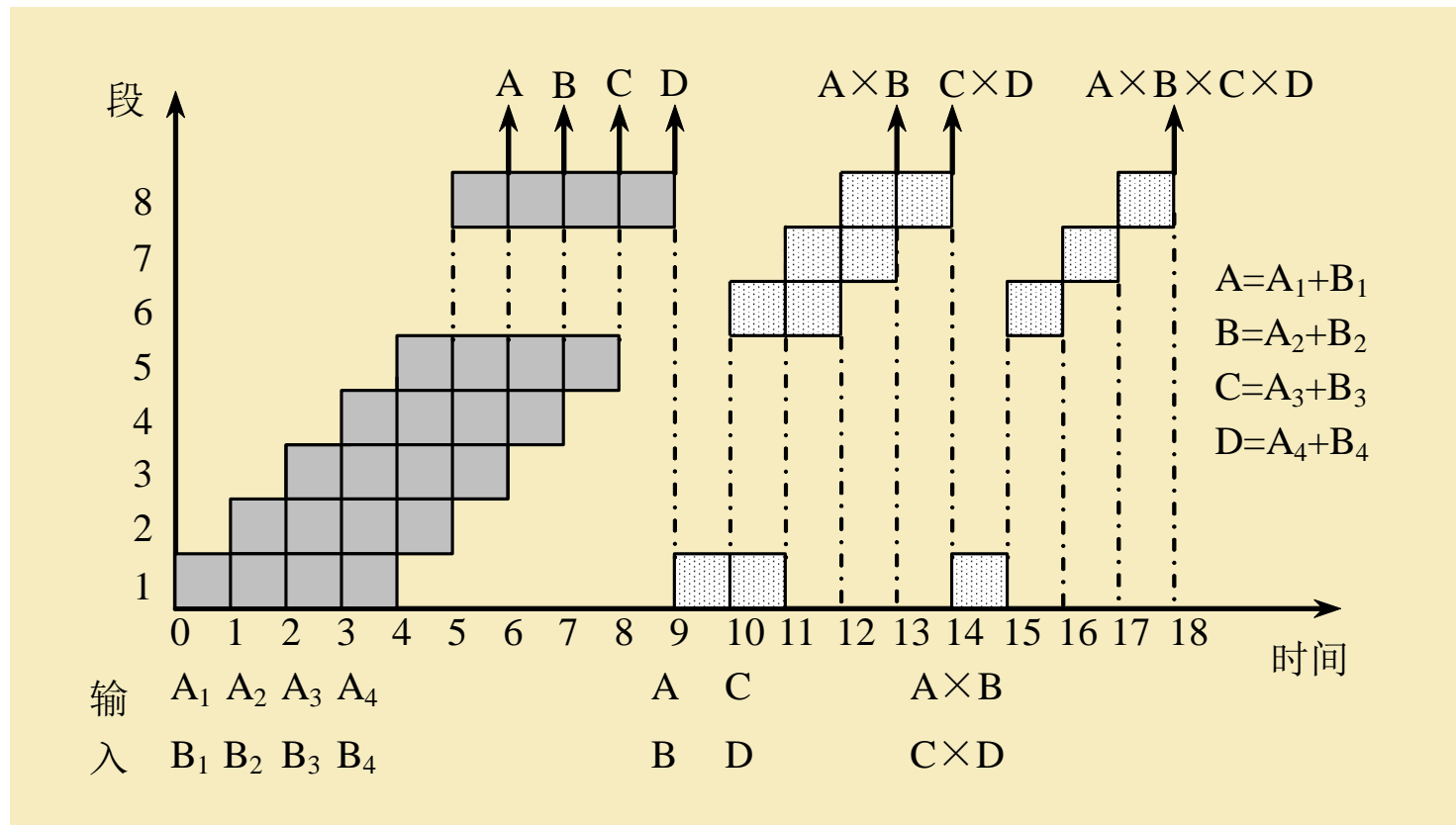
(每段的时间都为 Δt)

解：（1）选择适合于流水线工作的算法

- 先计算 A_1+B_1 、 A_2+B_2 、 A_3+B_3 和 A_4+B_4 ；
- 再计算 $(A_1+B_1) \times (A_2+B_2)$ 和 $(A_3+B_3) \times (A_4+B_4)$ ；
- 然后求总的乘积结果。

（2）画出时空图

3.3 流水线的性能指标



$$TP = \frac{7}{18\Delta t}$$

$$S = \frac{36\Delta t}{18\Delta t} = 2$$

$$E = \frac{4 \times 6 + 3 \times 4}{8 \times 18} = 0.25$$

(3) 计算性能

- 在18个 Δt 时间中，给出了7个结果。吞吐率为：

$$TP = \frac{7}{18\Delta t}$$

- 不用流水线，由于一次求和需 $6\Delta t$ ，一次求积需 $4\Delta t$ ，则产生上述7个结果共需 $(4 \times 6 + 3 \times 4) \Delta t = 36\Delta t$
加速比为

$$S = \frac{36\Delta t}{18\Delta t} = 2$$

▣ 流水线的效率

$$E = \frac{4 \times 6 + 3 \times 4}{8 \times 18} = 0.25$$

可以看出，在求解此问题时，该流水线的效率不高。

(原因)：多功能流水线在做某一运算时，总有一些段空闲；静态流水线功能切换时，要等前一种运算结果流出后才进行后一种运算；运算之间存在数据相关；存在建立与排空过程。

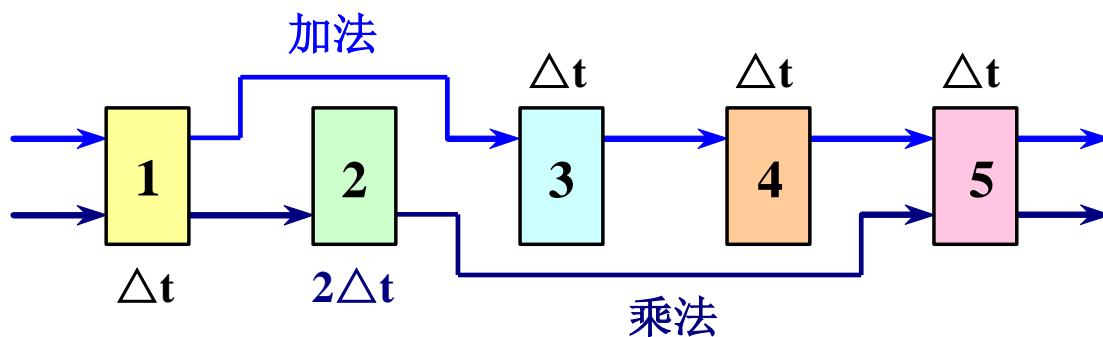
➤ 主要原因

- ❑ 多功能流水线在做某一种运算时，总有一些段是空闲的。
- ❑ 静态流水线在进行功能切换时，要等前一种运算全部流出流水线后才能进行后面的运算。
- ❑ 运算之间存在关联，后面有些运算要用到前面运算的结果。
- ❑ 流水线的工作过程有建立与排空部分。

例3.2 有一条动态多功能流水线由5段组成，加法用1、3、4、5段，乘法用1、2、5段，第2段的时间为 $2\Delta t$ ，其余各段时间均为 Δt ，而且流水线的输出可以直接返回输入端或暂存于相应的流水寄存器中。若在该流水线上计算：

$$\sum_{i=1}^4 (A_i \times B_i)$$

试计算其吞吐率、加速比和效率。

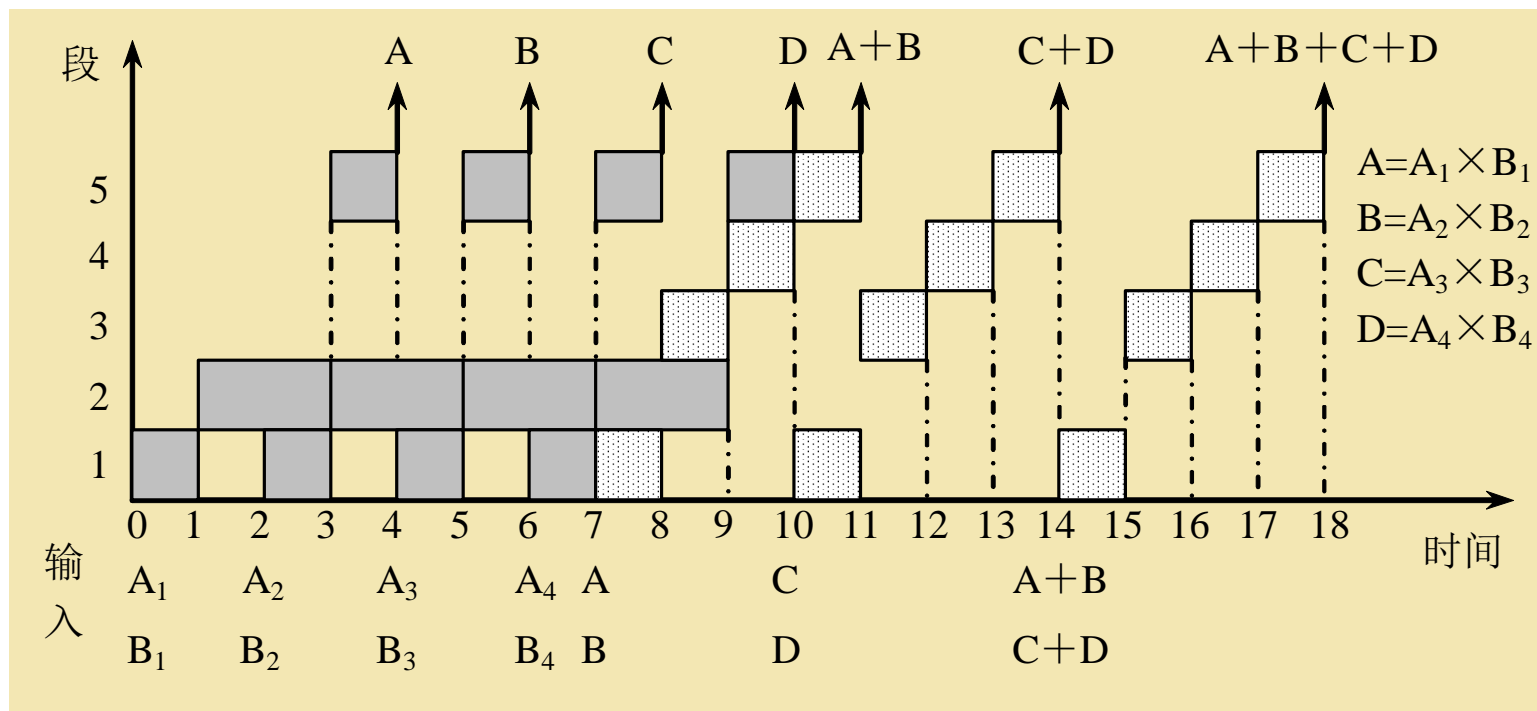


解：(1) 选择适合于流水线工作的算法

- 应先计算 $A1 \times B1$ 、 $A2 \times B2$ 、 $A3 \times B3$ 和 $A4 \times B4$;
- 再计算 $(A1 \times B1) + (A2 \times B2)$
 $(A3 \times B3) + (A4 \times B4)$;
- 然后求总的累加结果。

(2) 画出时空图

(3) 计算性能



$$TP = \frac{7}{18\Delta t}$$

$$S = \frac{28\Delta t}{18\Delta t} \approx 1.56$$

$$E = \frac{4 \times 4 + 3 \times 4}{5 \times 18} \approx 0.31$$

下面我们再看一个例子：

例 在静态流水线上计算：

$$\sum_{i=1}^4 (A_i \times B_i)$$

求：吞吐率，加速比，效率。

解：（1）确定适合于流水处理的计算过程

（2）画时空图

（3）性能计算

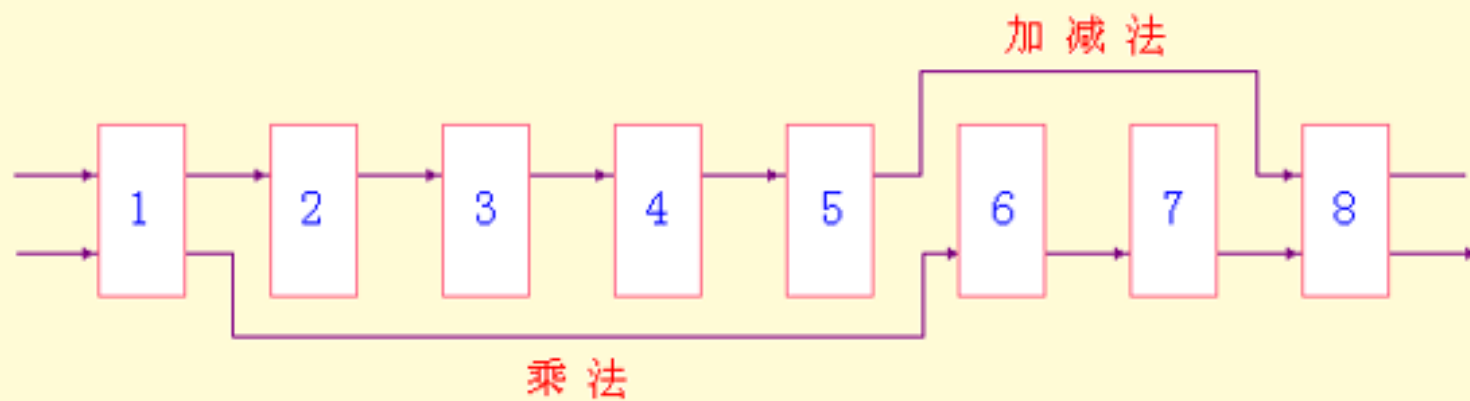
$$\text{吞吐率 } TP = 7 / (20 \Delta t)$$

$$\text{加速比 } S = (34 \Delta t) / (20 \Delta t) = 1.7$$

$$\text{效率 } E = (4 \times 4 + 3 \times 6) / (8 \times 20) = 0.21$$

静态流水线

(举 例)



可以看出，在求解此问题时，该流水线的效率不高。

动态流水线的时空图 举例 I

举例 II：这样行不行？ 正确答案

3.3.5 流水线设计中的若干问题

1. 瓶颈问题

- 理想情况下，流水线在工作时，其中的任务是同步地每一个时钟周期往前流动一段。
- 当流水线各段不均匀时，机器的时钟周期取决于瓶颈段的延迟时间。
- 在设计流水线时，要尽可能使各段时间相等。

2. 流水线的额外开销

- ▣ 流水寄存器延迟
- ▣ 时钟偏移开销

- 流水寄存器需要建立时间和传输延迟
 - **建立时间**：在触发写操作的时钟信号到达之前，寄存器输入必须保持稳定的时间。
 - **传输延迟**：时钟信号到达后到寄存器输出可用的时间。
- 时钟偏移开销
 - 流水线中，时钟到达各流水寄存器的最大差值时间。（时钟到达各流水寄存器的时间不是完全相同）

➤ 几个问题

- ❑ 流水线并不能减少（而且一般是增加）单条指令的执行时间，但却能提高吞吐率。
- ❑ 增加流水线的深度（段数）可以提高流水线的性能。
- ❑ 流水线的深度受限于流水线的额外开销。
- ❑ 当时钟周期小到与额外开销相同时，流水已没意义。因为这时在每一个时钟周期中已没有时间来做有用的工作。

额外开销对流水线性能影响较大，特别是流水线深度比较大、时钟周期较小时，应选择高性能的锁存器作流水寄存器。

3. 冲突问题

流水线设计中要解决的**重要问题之一**。

3.4 流水线的相关与冲突

3.4.1 一个经典的5段流水线

- 介绍一个经典的5段RISC流水线
- 首先讨论在非流水情况下是如何实现的

1. 一条指令的执行过程分为以下5个周期：

- 取指令周期（IF）
 - $IR \leftarrow Mem[PC]$ 。
 - PC值加4。（假设每条指令占4个字节）

➤ 指令译码/读寄存器周期 (ID)

- 译码。
- 用IR中的寄存器编号去访问通用寄存器组，读出所需的操作数。

➤ 执行/有效地址计算周期 (EX)

不同指令所进行的操作不同：

- 存储器访问指令：ALU把所指定的寄存器的内容与偏移量相加，形成用于访存的有效地址。
- 寄存器—寄存器ALU指令：ALU按照操作码指定的操作对从通用寄存器组中读取的数据进行运算。

- **寄存器—立即数ALU指令：**ALU按照操作码指定的操作对从通用寄存器组中读取的第一操作数和立即数进行运算。
 - **分支指令：**ALU把偏移量与PC值相加，形成转移目标的地址。同时，对在前一个周期读出的操作数进行判断，确定分支是否成功。
- **存储器访问 / 分支完成周期（MEM）**
- 该周期处理的指令只有load、store和分支指令。其他类型的指令在此周期不做任何操作。

□ load和store指令

load指令：用上一个周期计算出的有效地址从存储器中读出相应的数据。

store指令：把指定的数据写入这个有效地址所指出的存储器单元。

□ 分支指令

分支“成功”，就把转移目标地址送入PC。
分支指令执行完成。

➤ 写回周期（WB）

ALU运算指令和load指令在这个周期把结果数据写入通用寄存器组。

ALU运算指令：结果数据来自ALU。

load指令：结果数据来自存储器系统。

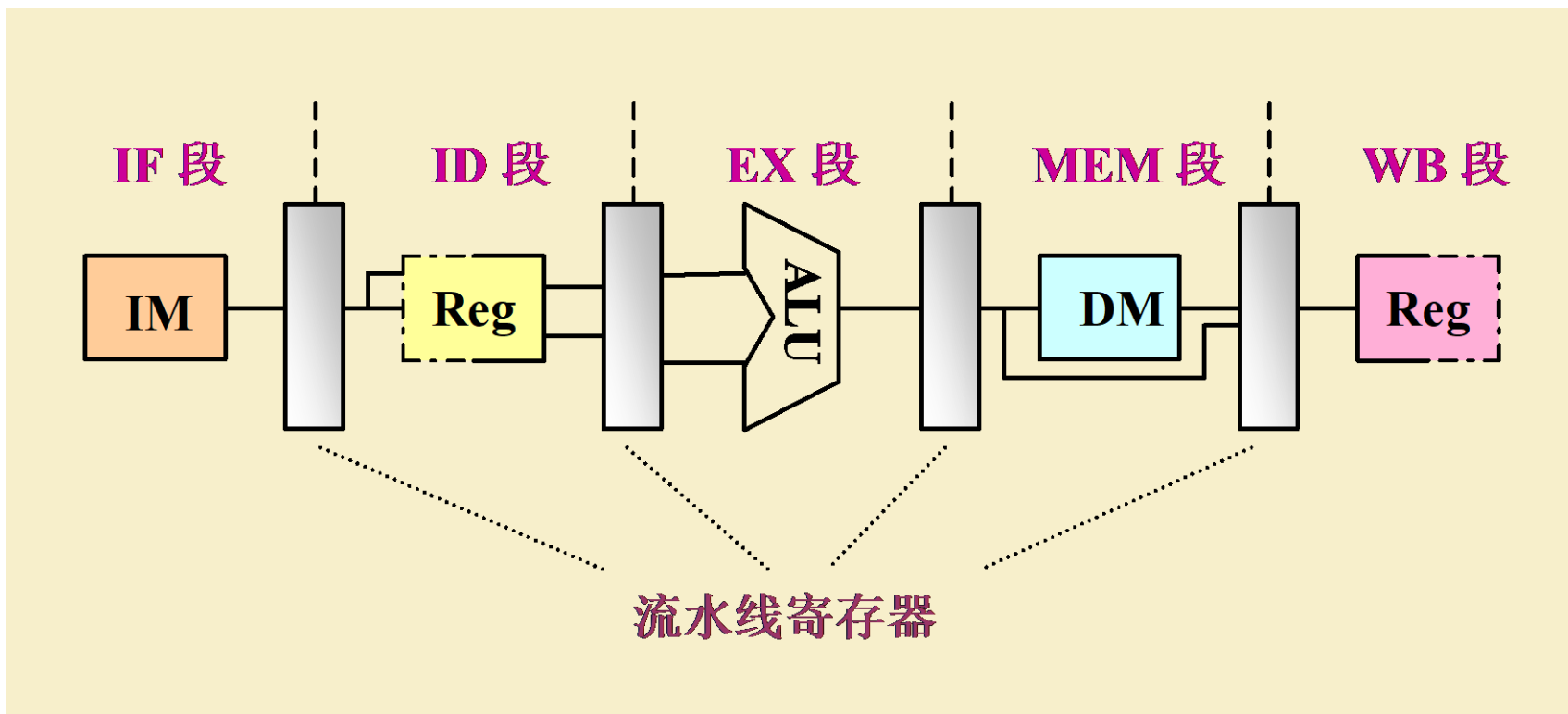
在这个实现方案中：

- 分支指令需要4个时钟周期（如果把分支指令的执行提前到ID周期，则只需要2个周期）。
- store指令需要4个周期。
- 其他指令需要5个周期才能完成。

2. 将上述实现方案修改为流水线实现

➤ 一个经典的5段流水线

- 每一个周期作为一个流水段。
- 在各段之间加上锁存器（流水寄存器）。

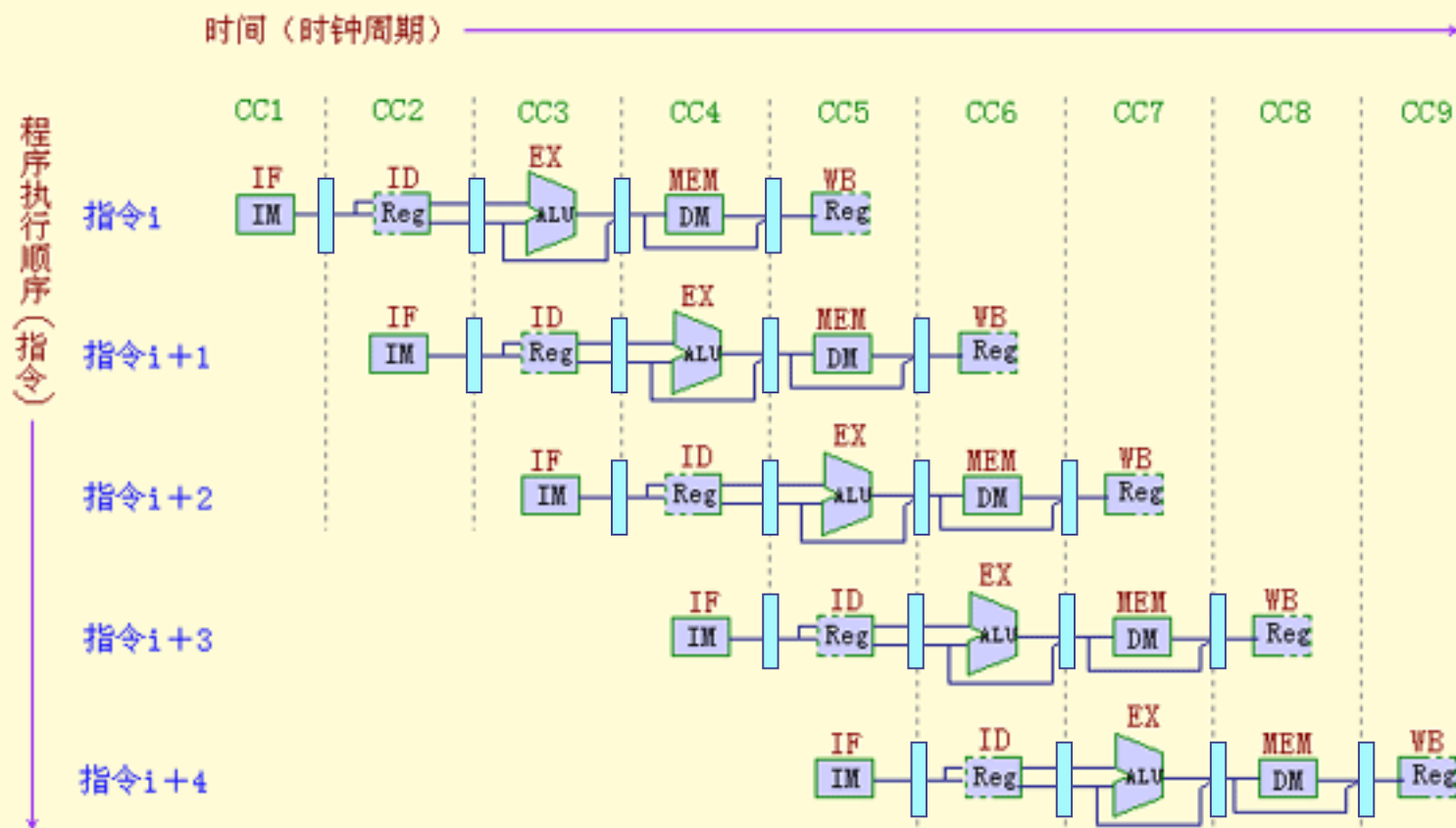


- 5段流水线的两种描述方式
 - 第一种描述（类似于时空图）

指令编号	时钟周期								
	1	2	3	4	5	6	7	8	9
指令i	IF	ID	EX	MEM	WB				
指令i+1		IF	ID	EX	MEM	WB			
指令i+2			IF	ID	EX	MEM	WB		
指令i+3				IF	ID	EX	MEM	WB	
指令i+4					IF	ID	EX	MEM	WB

➤ 第二种描述（按时间错开的数据通路序列）

流水线可以看成是按时间错开的数据通路序列



3. 采用流水线方式实现时，应解决以下几个问题：

- 要保证不会在同一时钟周期要求同一个功能段做两件不同的工作。

例如，不能要求ALU同时做有效地址计算和算术运算。

- 避免IF段的访存（取指令）与MEM段的访存（读/写数据）发生冲突。
 - 可以采用分离的指令存储器和数据存储器；
 - 一般采用分离的指令Cache和数据Cache。
- ID段和WB段都要访问同一寄存器文件。

ID段：读

WB段：写

如何解决对同一寄存器的访问冲突？

把写操作安排在时钟周期的前半拍完成，把读操作安排在后半拍完成。

➤ 考虑PC的问题

- 流水线为了能够每个时钟周期启动一条新的指令，就必须在每个时钟周期进行PC值的加4操作，并保留新的PC值。这种操作**必须在IF段完成**，以便为取下一条指令做好准备。

（需设置一个专门的加法器）

- 但分支指令也可能改变PC的值，而且是在MEM段进行，这会导致冲突。

请考虑一下，如何处理分支指令？

3.4.2 相关与流水线冲突

3.4.2.1 相关

- **相关：**两条指令之间存在某种依赖关系。

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

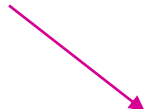


- 相关有3种类型
 - 数据相关（也称真数据相关）
 - 名相关
 - 控制相关

1. 数据相关

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

数据相关反映了数据的流动关系，即如何从其产生者流动到其消费者。

例如：下面这一段代码存在数据相关。

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

- 当数据的流动是经过寄存器时，相关的检测比较直观和容易。
- 当数据的流动是经过存储器时，检测比较复杂。
 - 相同形式的地址其有效地址未必相同。
 - 形式不同的地址其有效地址却可能相同。

2. 名相关

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

➤ 指令 j 与指令 i 之间的名相关有两种：

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

指令 j 写的名 = 指令 i 读的名

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

指令 j 写的名 = 指令 i 写的名

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

例如：考虑下述代码：

```
DIV. D      F2, F6, F4
ADD. D      F6, F0, F12
SUB. D      F8, F6, F14
```

DIV. D和ADD. D存在反相关。

进行寄存器换名（F6换成S）后，变成：

```
DIV. D      F2, F6, F4
ADD. D      S, F0, F12
SUB. D      F8, S, F14
```

3. 控制相关

- **控制相关**是指由分支指令引起的相关。
 - 为了保证程序应有的执行顺序，必须严格按控制相关确定的顺序执行。

- 典型的程序结构是“if-then”结构。

- 请看一个示例：

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

➤ 控制相关带来了以下两个限制：

- 与一条分支指令控制相关的指令不能被移到该分支之前，否则这些指令就不受该分支控制了。

对于上述的例子，`then` 部分中的指令不能移到`if`语句之前。

- 如果一条指令与某分支指令不存在控制相关，就不能把该指令移到该分支之后。

对于上述的例子，不能把`S`移到`if`语句的`then`部分中。

3.4.2.2 流水线冲突

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3种类型：

- **结构冲突：**因硬件资源满足不了指令重叠执行的要求而发生的冲突。
- **数据冲突：**当指令在流水线中重叠执行时，因需要用到前面指令的执行结果而发生的冲突。
- **控制冲突：**流水线遇到分支指令和其他会改变PC值的指令所引起的冲突。

带来的几个问题：

- 导致错误的执行结果。
- 流水线可能会出现停顿，从而降低流水线的效率和实际的加速比。
- 我们约定

当一条指令被暂停时，在该暂停指令之后流出的所有指令都要被暂停，而在该暂停指令之前流出的指令则继续进行（否则就永远无法消除冲突）。
在暂停期间,流水线不启动新的指令。

1. 结构冲突

- 在流水线处理机中，为了能够使各种组合的指令都能顺利地重叠执行，需要对功能部件进行流水或重复设置资源。
- 如果某种指令组合因为资源冲突而不能正常执行，则称该处理机有**结构冲突**。
- 常见的导致结构相关的原因：
 - 功能部件不是完全流水
 - 资源份数不够

➤ 结构冲突举例：访存冲突

有些流水线处理机只有一个存储器，将数据和指令放在一起，访存指令会导致访存冲突。

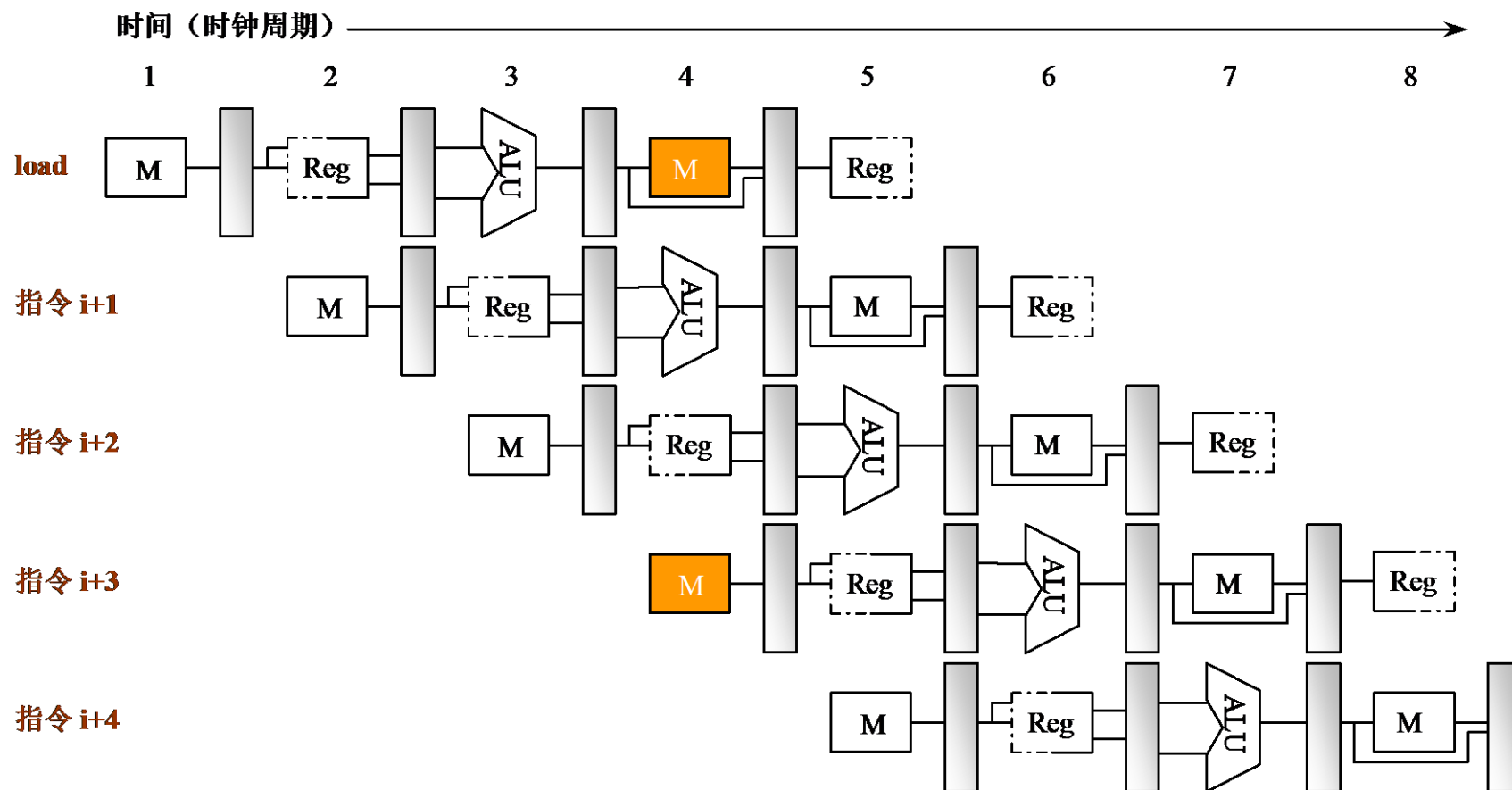
□ 解决办法I：插入暂停周期

（“流水线气泡”或“气泡”）

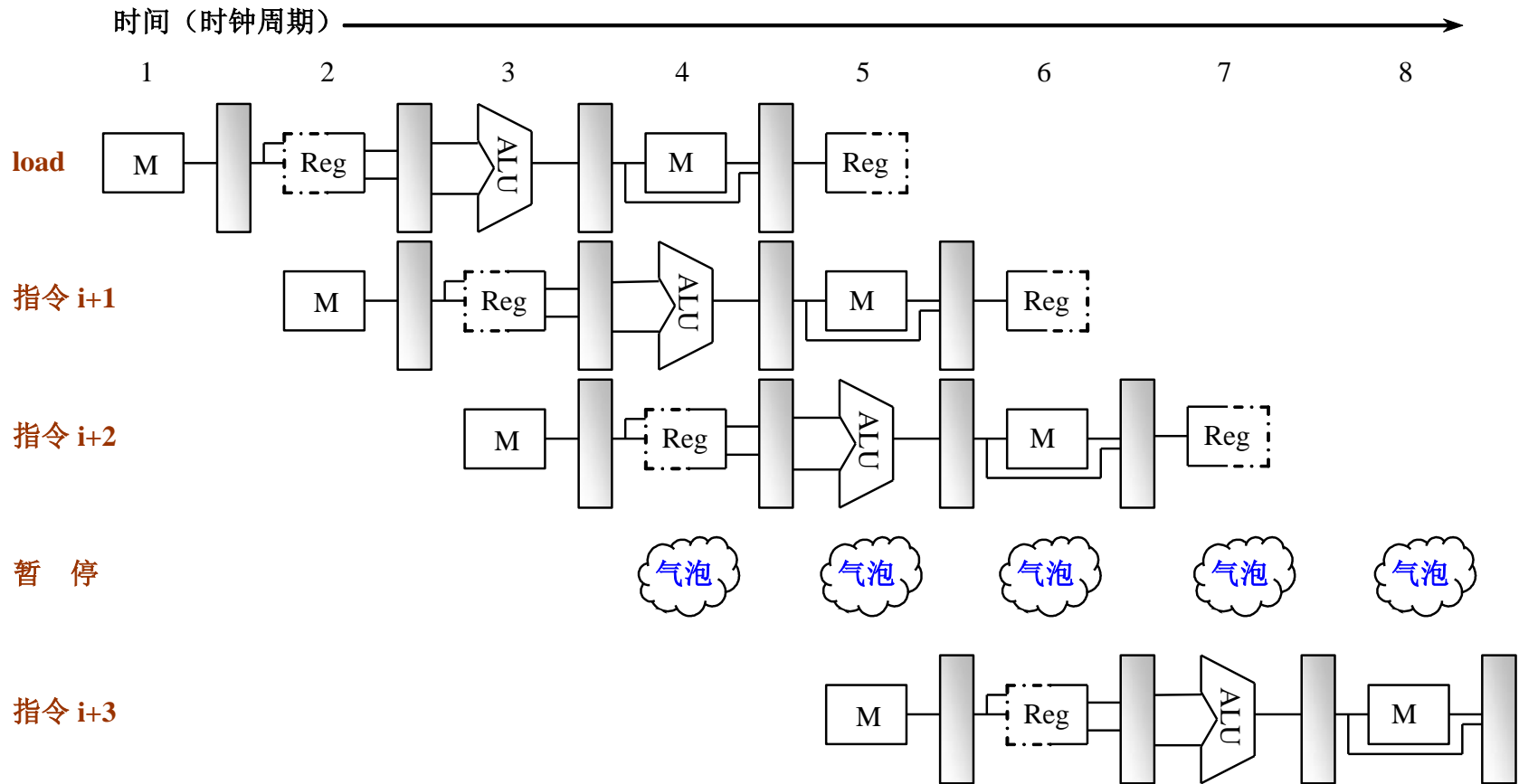
引入暂停后的时空图

□ 解决方法II：

设置相互独立的指令存储器和数据存储器
或设置相互独立的指令Cache和数据Cache。



由于访问同一个存储器而引起的结构冲突



为消除结构冲突而插入的流水线气泡

引入暂停后的时空图

指令编号	时钟周期									
	1	2	3	4	5	6	7	8	9	10
指令i	IF	ID	EX	MEM	WB					
指令i+1		IF	ID	EX	MEM	WB				
指令i+2			IF	ID	EX	MEM	WB	WB		
指令i+3				stall	IF	ID	EX	MEM	WB	
指令i+4						IF	ID	EX	MEM	WB
指令i+5							IF	ID	EX	MEM

➤ 有时流水线设计者允许结构冲突的存在

主要原因：减少硬件成本

- 如果把流水线中的所有功能单元完全流水化，或者重复设置足够份数，那么所花费的成本将相当高。

2. 数据冲突

当相关的指令靠得足够近时，它们在流水线中的重叠执行或者重新排序会改变指令读/写操作数的顺序，使之不同于它们非流水实现时的顺序，则发生了数据冲突。

举例：

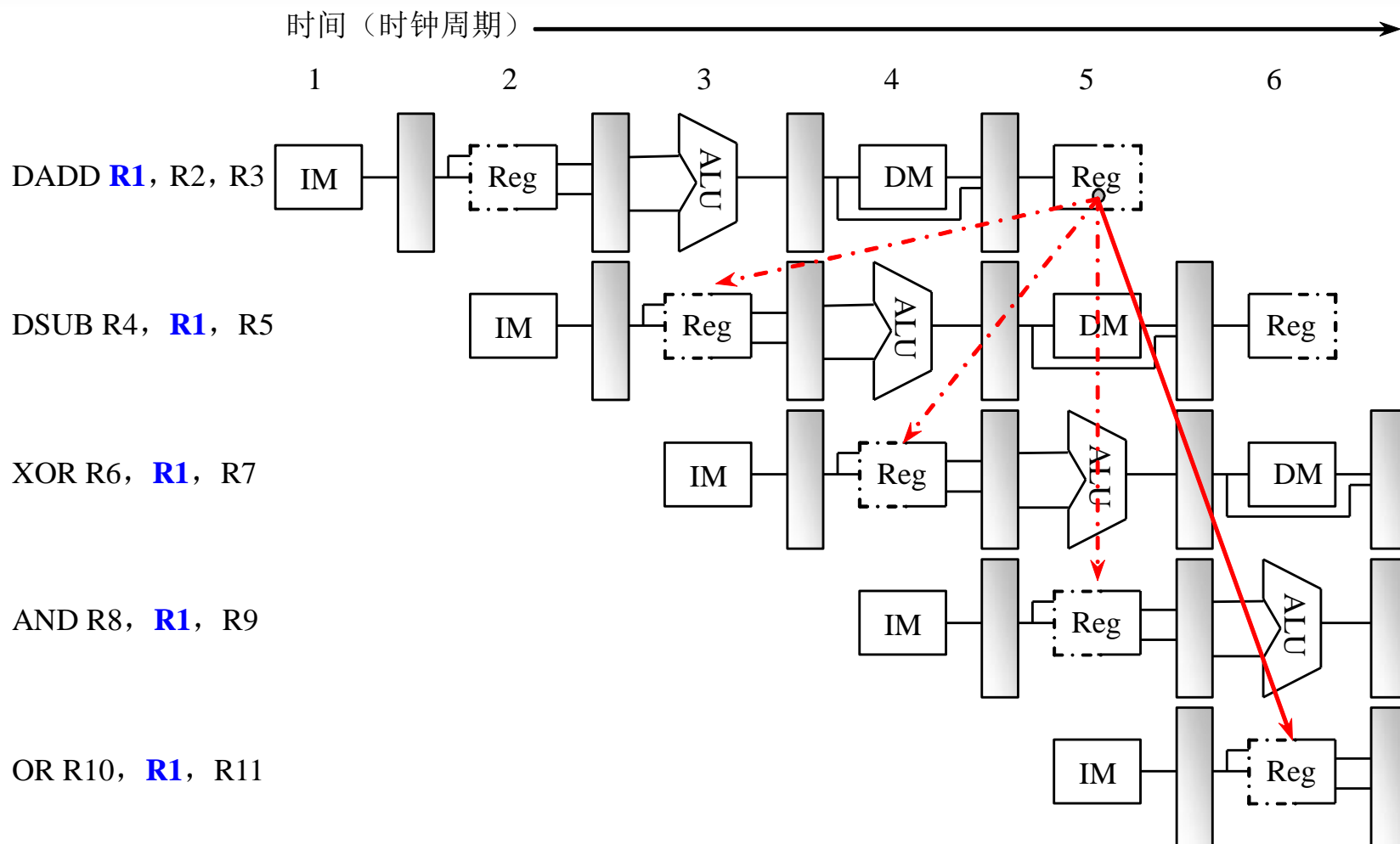
DADD R1, R2, R3

DSUB R4, R1, R5

XOR R6, R1, R7

AND R8, R1, R9

OR R10, R1, R11



流水线的数据冲突举例

- 根据指令读访问和写访问的顺序，可以将数据冲突分为3种类型。

考虑两条指令*i*和*j*，且*i*在*j*之前进入流水线，可能发生的数据冲突有：

- **写后读冲突（RAW）**

在 *i* 写入之前，*j* 先去读。

j 读出的内容是错误的。

这是最常见的一种数据冲突，它对应于真数据相关。

□ 写后写冲突 (WAW)

在 i 写入之前, j 先写。

最后写入的结果是 i 的。错误!

这种冲突对应于输出相关。

写后写冲突仅发生在这样的流水线中:

- 流水线中不只一个段可以进行写操作。
- 当先前某条指令停顿时, 允许其后续指令继续前进。

前面介绍的5段流水线不会发生写后写冲突。

(只在WB段写寄存器)

□ 读后写冲突（WAR）

在 i 读之前， j 先写。

i 读出的内容是错误的！

由反相关引起。

这种冲突仅发生在这样的情况下：

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

读后写冲突在前述5段流水线中不会发生。

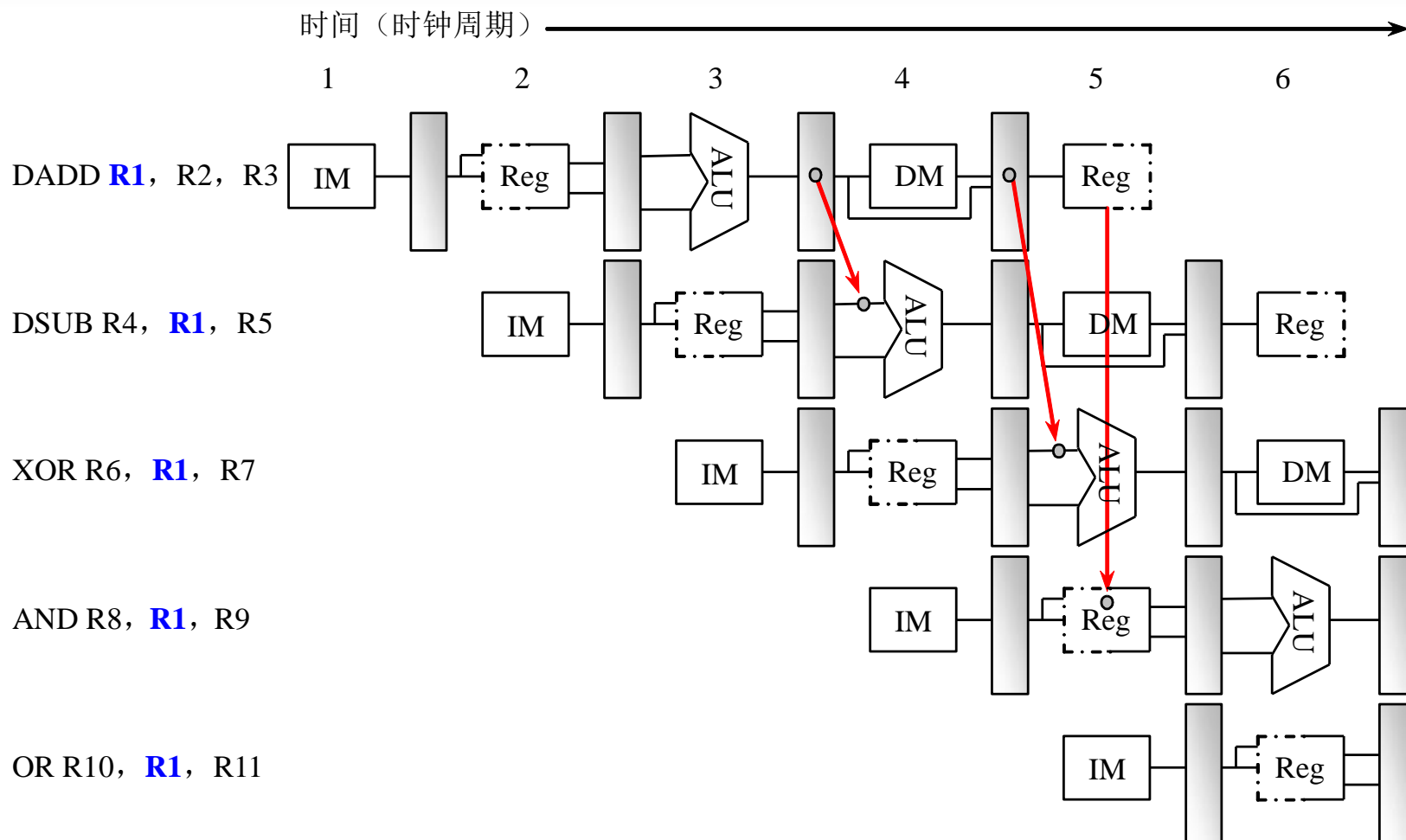
（读操作（在ID段）在写结果操作（在WB段）之前）

➤ 通过**定向技术**减少数据冲突引起的停顿

（定向技术也称为旁路或短路）

- **关键思想：**在某条指令产生计算结果之前，其他指令并不真正立即需要该计算结果，如果能够将该计算结果从其产生的地方直接送到其他指令需要它的地方，那么就可以避免停顿。
- 采用定向技术消除上例中的相关

工作过程演示

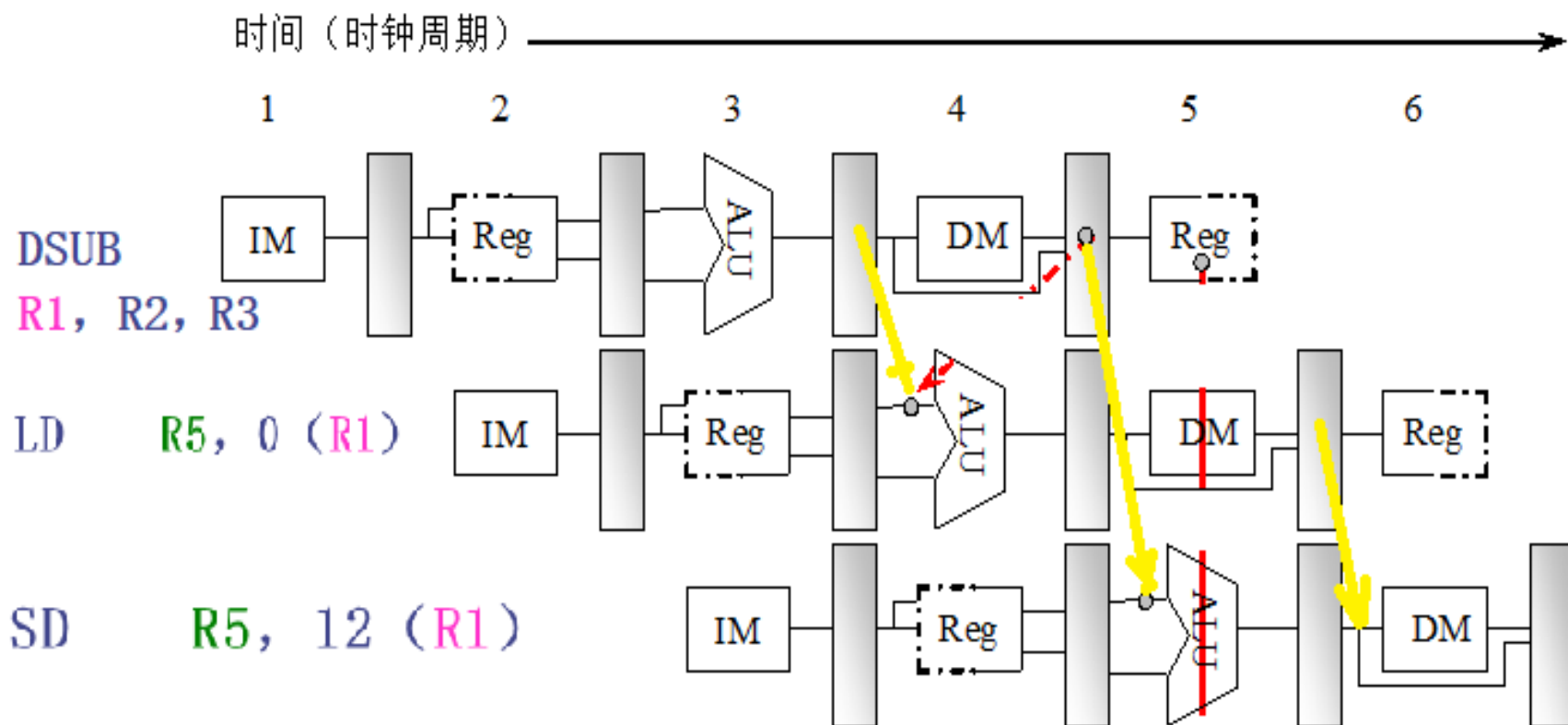


采用定向技术后的流水线数据通路

- 当定向硬件检测到前面某条指令的结果寄存器就是当前指令的源寄存器时，控制逻辑会将前面那条指令的结果直接从其产生的地方定向到当前指令所需的位置。
- 结果数据不仅可以从某一功能部件的输出定向到其自身的输入，而且还可以定向到其他功能部件的输入。

举例:

```
DSUB  R1, R2, R3  
LD     R5, 0 (R1)  
SD     R5, 12 (R1)
```



更多的定向路径

➤ 需要停顿的数据冲突

- 并不是所有的数据冲突都可以用定向技术来解决。

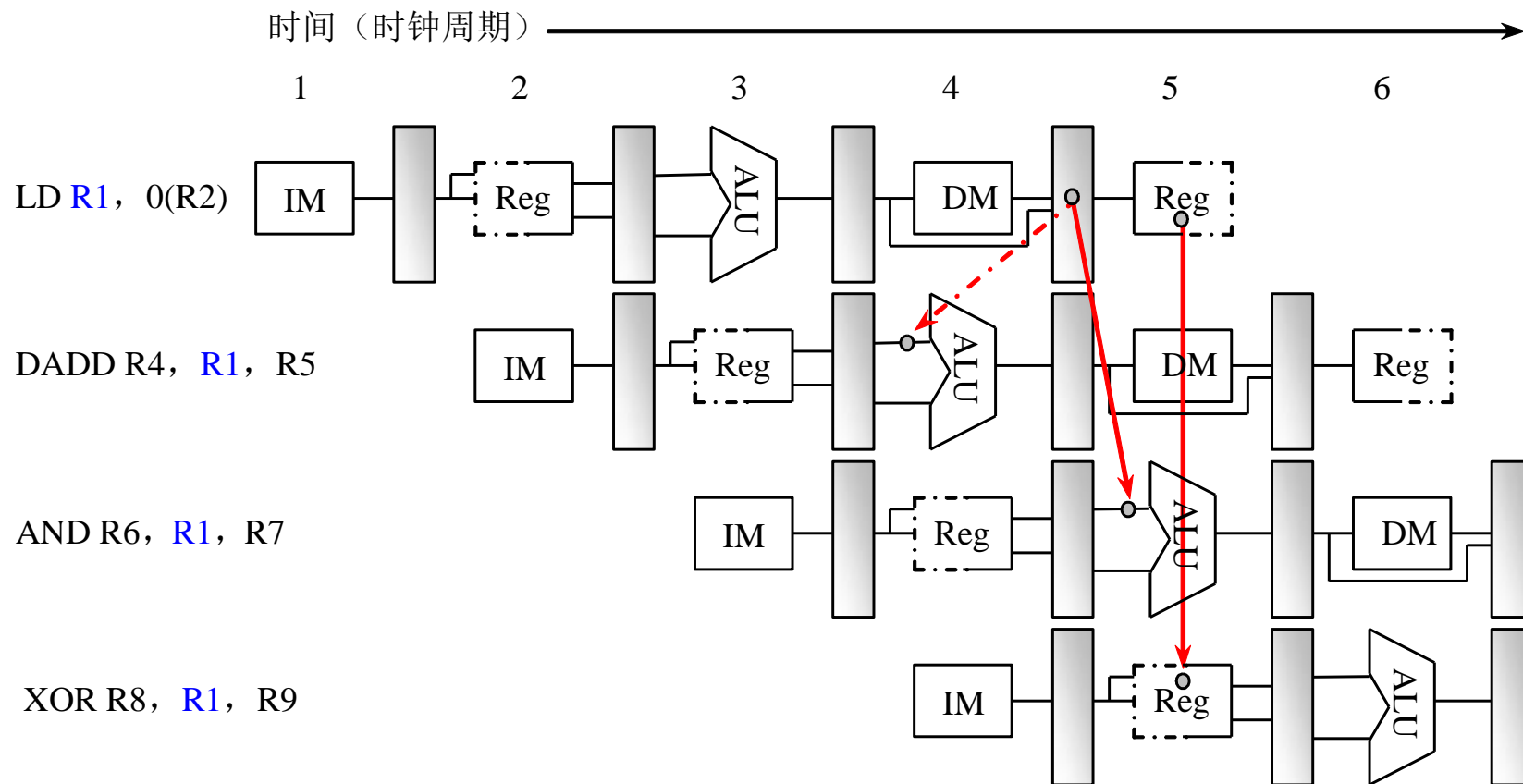
举例:

```
LD      R1, 0 (R2)
DADD    R4, R1, R5
AND      R6, R1, R7
XOR      R8, R1, R9
```

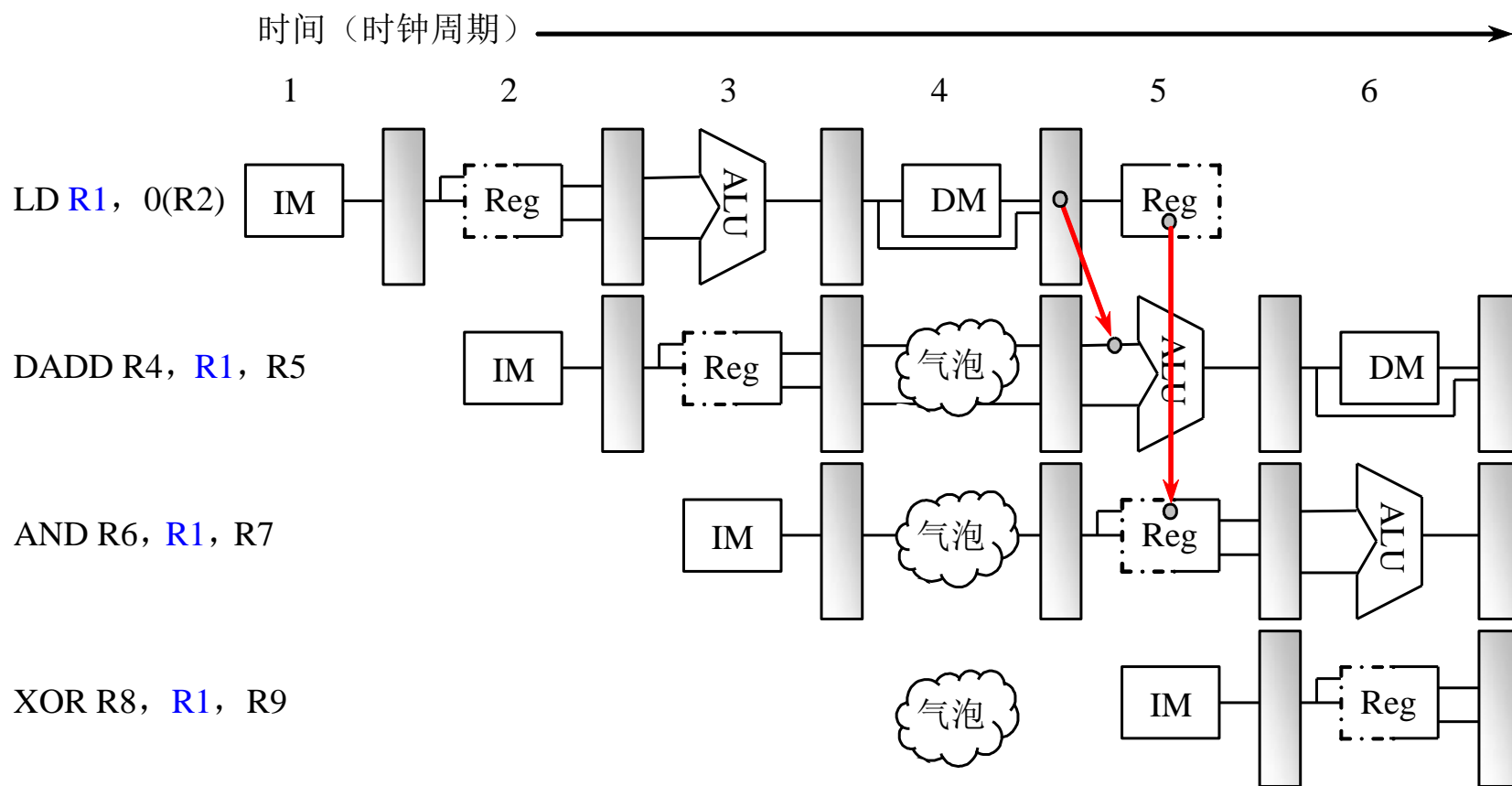
- 增加流水线互锁硬件，插入“暂停”。

作用: 检测发现数据冲突，并使流水线停顿，直至冲突消失。

举例: 演示A 演示B



无法将LD指令的结果定向到DADD指令



流水线互锁机制插入气泡后的执行过程

LD R1, 0 (R2)	IF	ID	EX	MEM	WB			
DADD R4, R1, R5		IF	ID	EX	MEM	WB		
AND R6, R1, R7			IF	ID	EX	MEM	WB	
XOR R8, R1, R9				IF	ID	EX	MEM	WB

LD R1, 0 (R2)	IF	ID	EX	MEM	WB			
DADD R4, R1, R5		IF	ID	stall	EX	MEM	WB	
AND R6, R1, R7			IF	stall	ID	EX	MEM	WB
XOR R8, R1, R9				stall	IF	ID	EX	MEM

插入停顿前后的流水线时空图

➤ 依靠编译器解决数据冲突

让编译器重新组织指令顺序来消除冲突，这种技术称为**指令调度**或**流水线调度**。

- **例如：**采用典型的代码生成方法，
表达式 $A=B+C$ 的代码会导致暂停

LD Rb, B	IF	ID	EX	MEM	WB				
LD Rc, C		IF	ID	EX	MEM	WB			
DADD Ra, Rb, Rc			IF	ID	stall	EX	MEM	WB	
SD Ra, A				IF	stall	ID	EX	MEM	WB

□ 举例：

请为下列表达式生成没有暂停的指令序列：

$A = B + C$ ；

$D = E - F$ ；

假设载入延迟为1个时钟周期。

题解

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

3. 控制冲突

- 执行分支指令的结果有两种
 - **分支成功**：PC值改变为分支转移的目标地址。
在条件判定和转移地址计算都完成后，才改变PC值。
 - **不成功或者失败**：PC的值保持正常递增，
指向顺序的下一条指令。
- 处理分支指令最简单的方法：
“冻结”或者“排空”流水线。
优点：简单。
 - 前述5段流水线中，当在流水线的译码段ID检测到是分支指令，就暂停执行其后的所有指令，直到分支指令到达MEM段，确定分支成功与否，并改变PC值是在MEM段进行的。
给流水线带来了3个时钟周期的延迟。

简单处理分支指令：分支成功的情况

[illegible]

简单处理分支指令：分支失败的情况

[illegible]

- 把由分支指令引起的延迟称为分支延迟。
- 分支指令在目标代码中出现的频度
 - 每3~4条指令就有一条是分支指令。
 - 假设：分支指令出现的频度是30%，
流水线理想 $CPI=1$ ，
那么：流水线的实际 $CPI = 1.9$ 。
- 可采取两种措施来减少分支延迟。
 - 在流水线中尽早判断出分支转移是否成功；
 - 尽早计算出分支目标地址。

下面的讨论中，我们假设：

这两步工作被提前到ID段完成，即分支指令是在ID段的末尾执行完成，所带来的分支延迟为一个时钟周期。

➤ 3种通过软件（编译器）来减少分支延迟的方法

共同点：

- 对分支的处理方法在程序的执行过程中始终是不变的，是静态的。
- 要么总是预测分支成功，要么总是预测分支失败。

□ 预测分支失败

- 允许分支指令后的指令继续在流水线中流动，就好象什么都没发生似的。
- 若确定分支失败，将分支指令看作是一条普通指令，流水线正常流动。

- 若确定分支成功，流水线就把在分支指令之后取出的所有指令转化为空操作，并按分支目地重新取指令执行。

要保证：分支结果出来之前不会改变处理机的状态，以便一旦猜错时，处理机能够回退到原先的状态。

流水线的处理过程

流水线对分支的处理过程

分支指令 i (失败)	IF	ID	EX	MEM	WB				
指令 i+1		IF	ID	EX	MEM	WB			
指令 i+2			IF	ID	EX	MEM	WB		
指令 i+3				IF	ID	EX	MEM	WB	
指令 i+4					IF	ID	EX	MEM	WB

分支指令 i (成功)	IF	ID	EX	MEM	WB				
指令 i+1		IF	ID	idle	idle	idle			
分支目标 j			IF	ID	EX	MEM	WB		
分支目标 j+1				IF	ID	EX	MEM	WB	
分支目标 j+2					IF	ID	EX	MEM	WB

□ 预测分支成功

假设分支转移成功，并从分支目标地址处取指令执行。

起作用的前题：先知道分支目标地址，后知道分支是否成功。

前述5段流水线中，这种方法没有任何好处。

□ 延迟分支

主要思想：

从逻辑上“延长”分支指令的执行时间。把延迟分支看成是由原来的分支指令和若干个延迟槽构成，不管分支是否成功，都要按顺序执行延迟槽中的指令。

延迟分支以及指令的执行顺序

具有一个分支延迟槽的流水线的执行过程

分支失败	分支指令 i	IF	ID	EX	MEM	WB				
	延迟槽指令 i+1		IF	ID	EX	MEM	WB			
	指令 i+2			IF	ID	EX	MEM	WB		
	指令 i+3				IF	ID	EX	MEM	WB	
	指令 i+4					IF	ID	EX	MEM	WB

分支成功	分支指令 i	IF	ID	EX	MEM	WB				
	延迟槽指令 i+1		IF	ID	EX	MEM	WB			
	分支目标指令 j			IF	ID	EX	MEM	WB		
	分支目标指令 j+1				IF	ID	EX	MEM	WB	
	分支目标指令 j+2					IF	ID	EX	MEM	WB

分支延迟槽中的指令“掩盖”了流水线原来必须插入的暂停周期。

分支延迟指令的调度

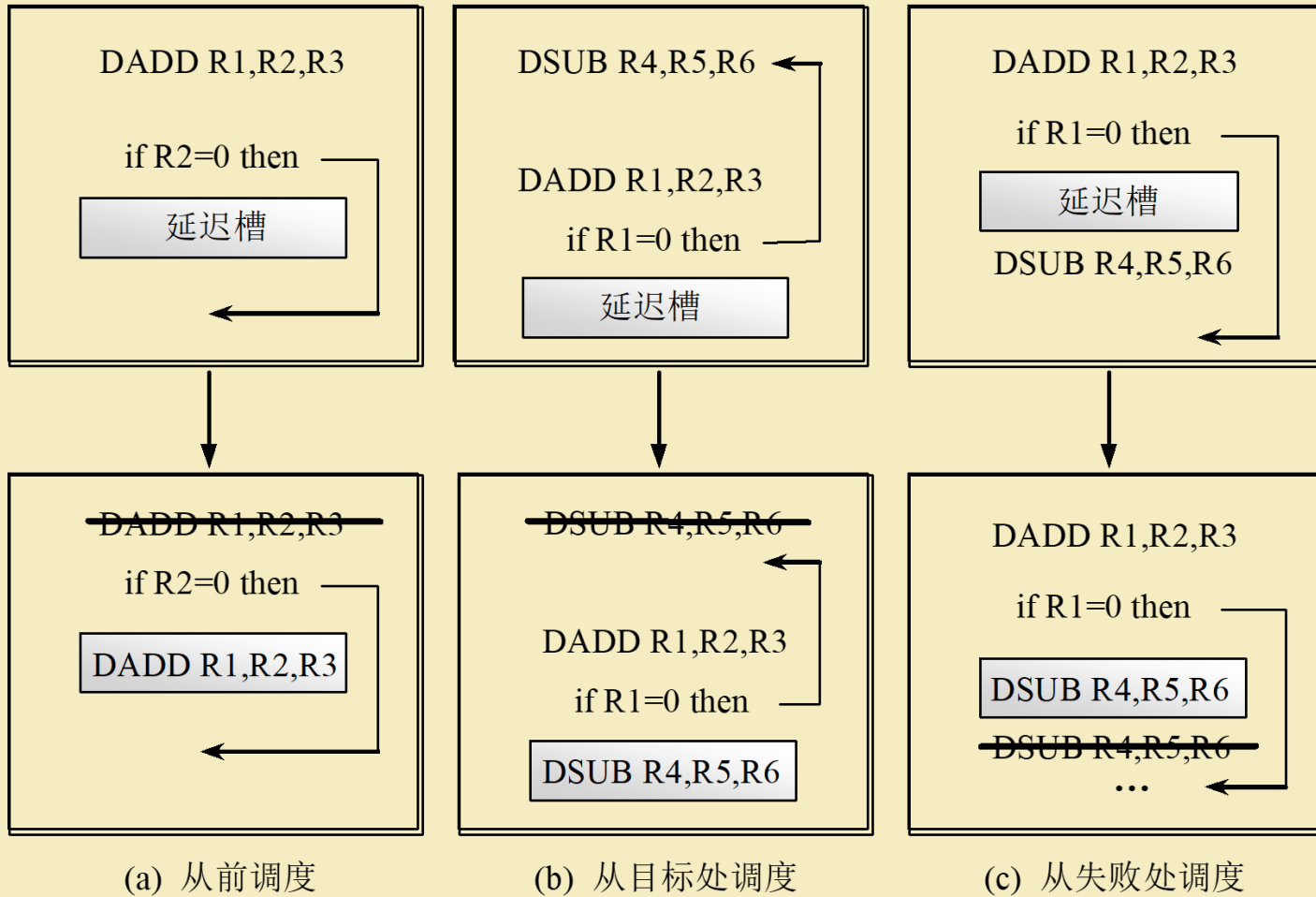
任务：在延迟槽中放入有用的指令。

由编译器完成。能否带来好处取决于编译器能否把有用的指令调度到延迟槽中。

三种调度方法：

- 从前调度
- 从目标处调度
- 从失败处调度

调度前和调度后的代码



三种方法的要求及效果

调 度 策 略	对调度的要求	什么情况下起作用
从 前 调 度	被调度的指令必须与分支无关	任何情况
从目标处调度	必须保证在分支失败时执行被调度的指令不会导致错误。有可能需要复制指令	分支成功时 (但由于复制指令, 有可能会增大程序空间)
从失败处调度	必须保证在分支成功时执行被调度的指令不会导致错误	分支失败时

分支延迟受到两个方面的限制：

- 可以被放入延迟槽中的指令要满足一定的条件。
- 编译器预测分支转移方向的能力。

进一步改进：分支取消机制（[取消分支](#)）

当分支的实际执行方向和事先所预测的一样时，执行分支延迟槽中的指令，否则就将分支延迟槽中的指令转化成一个空操作。

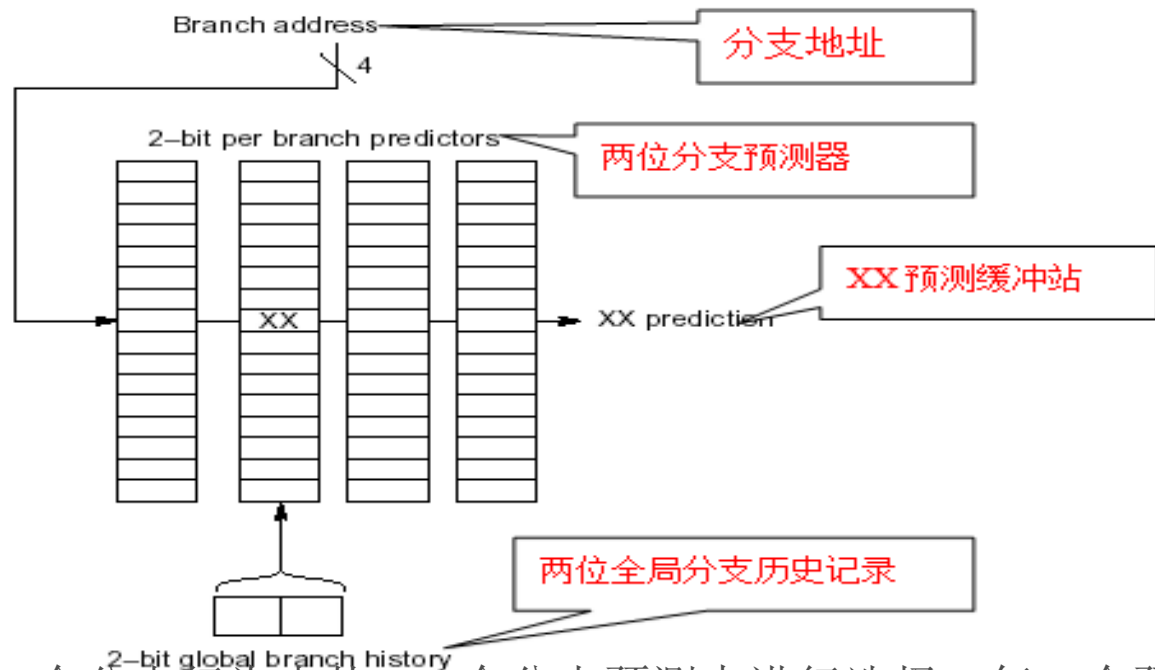
[“预测成功—取消”分支的执行过程](#)

分支失败	分支指令i	IF	ID	EX	MEM	WB				
	延迟槽指令(目标)		IF	idle	idle	idle	idle			
	指令 i+1			IF	ID	EX	MEM	WB		
	指令 i+2				IF	ID	EX	MEM	WB	
	指令 i+3					IF	ID	EX	MEM	WB

分支成功	分支指令i	IF	ID	EX	MEM	WB				
	延迟槽指令(目标)		IF	ID	EX	MEM	WB			
	分支目标指令+1			IF	ID	EX	MEM	WB		
	分支目标指令+2				IF	ID	EX	MEM	WB	
	分支目标指令+3					IF	ID	EX	MEM	WB

预测分支成功的情况下，从目标处调度分支取消机制的执行情况

1. 关联预测器



- 一个 $[m, n]$ 预测器表示使用前 m 个分支行为去从 2^m 个分支预测中进行选择，每一个预测是对应于单个分支的 n 位预测器。
- 这种相关分支预测器的吸引人之处，即在于它与两位预测器相比可以取得更高的预测率，并且只需要少量的额外硬件支持。
- 其硬件的简单性表现在：最近 m 个分支的全局历史记录可以记录在一个 m 位移位寄存器中，每一位记录着该分支是被执行还是未被执行。对分支预测缓冲站的访问可由分支地址的低位拼接上 m 位全局历史记录而得到。

例如：一个 $[2, 2]$ 预测器及如何访问预测器的例子

(2, 2) 分支预测缓冲站使用一个两位全局历史记录去选择4个预测器以获得每一个分支地址。因此所有预测器都对应于相关分支的两位预测器。

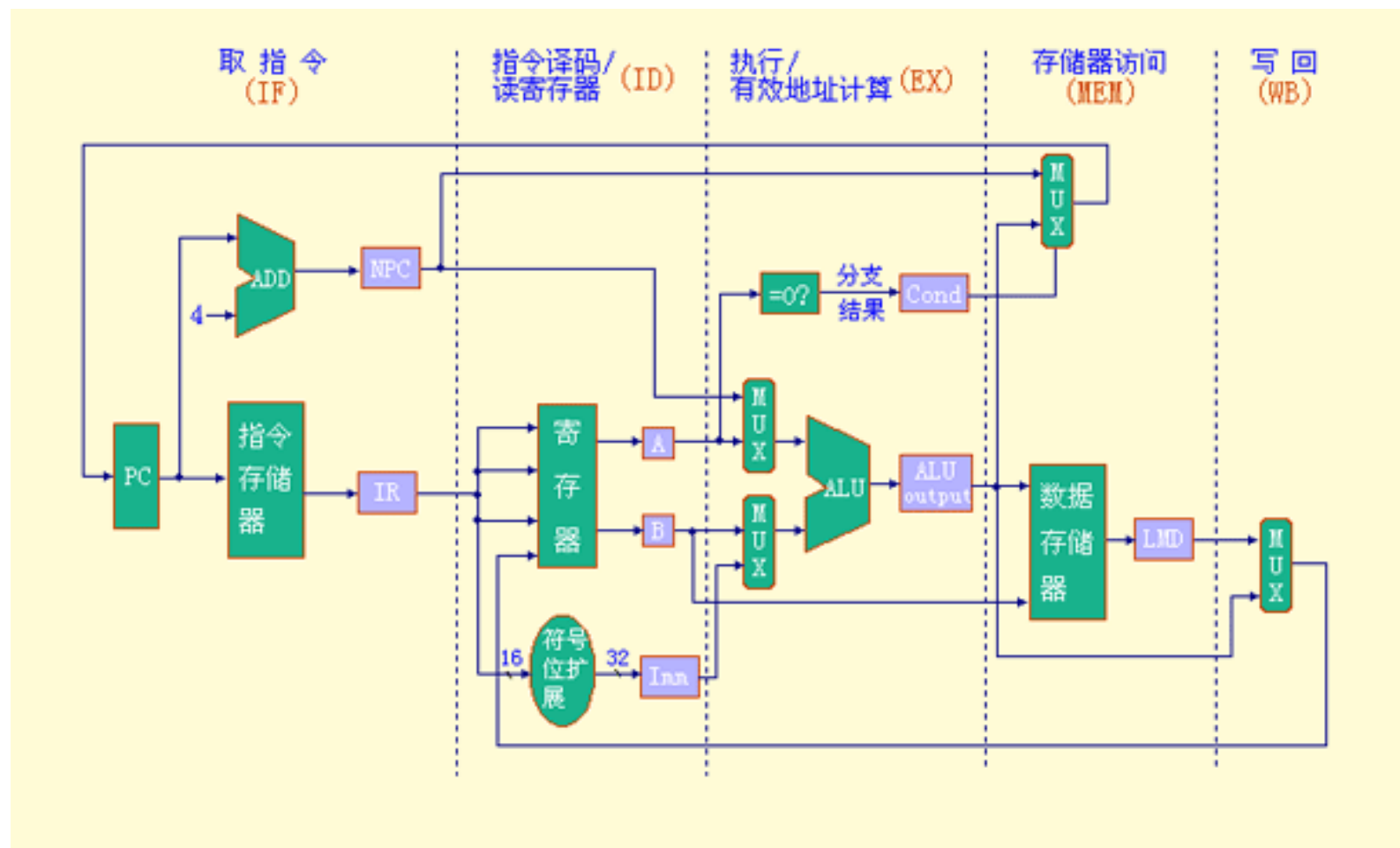
分支预测缓冲站有 $2^2 \times 16 = 64$ 个入口。分支地址用于选择4个入口，全局历史记录从这4个入口中再选择一个，两位的全局历史记录由一个移位寄存器实现且分支地址的低位拼接上两位全局历史记录而得到。

3.5 流水线的实现

3.5.1 MIPS的一种简单实现

1. 实现MIPS指令子集的一种简单数据通路。

- 该数据通路的操作分成5个时钟周期
 - 取指令
 - 指令译码/读寄存器
 - 执行/有效地址计算
 - 存储器访问/分支完成
 - 写回
- 只讨论整数指令的实现（包括：load和store，等于0转移，整数ALU指令等。）



2. 一条MIPS指令最多需要以下5个时钟周期:

➤ 取指令周期 (IF) 操作

- $IR \leftarrow Mem[PC]$
- $NPC \leftarrow PC + 4$ 指向顺序的下一条指令, 其地址放入NPC

➤ 指令译码/读寄存器周期 (ID) 操作

- $A \leftarrow Regs[rs]$
- $B \leftarrow Regs[rt]$
- $Imm \leftarrow ((IR_{16})^{16} \# IR_{16..31})$

指令的译码操作和读寄存器操作是并行进行的。

原因: 在MIPS指令格式中, 操作码字段以及rs、rt字段都是在固定的位置。

这种技术称为**固定字段译码**技术。

➤ 执行/有效地址计算周期 (EX)

不同指令所进行的操作不同:

- 存储器访问指令

[操作](#)

$ALUo \leftarrow A + Imm$ 将操作数相加形成有效地址, 存入临时寄存器ALUo

- 寄存器—寄存器ALU指令

[操作](#)

$ALUo \leftarrow A \text{ func } B$

- 寄存器—立即值ALU指令

[操作](#)

$ALUo \leftarrow A \text{ op } Imm$

- 分支指令

[操作](#)

$ALUo \leftarrow NPC + (Imm \ll 2) ;$

$cond \leftarrow (A == 0)$

将有效地址计算周期和执行周期合并为一个时钟周期，这是因为MIPS指令集采用load / store结构，没有任何指令需要同时进行数据有效地址的计算、转移目标地址的计算和对数据进行运算。

- 存储器访问/分支完成周期（MEM）
 - 所有指令都要在该周期对PC进行更新。
除了分支指令，其他指令都是做 $PC \leftarrow NPC$
 - 在该周期内处理的MIPS指令仅仅有load、store和分支三种指令。Load指令把从存储器读出的数据放入临时寄存器LMD;Store指令把B中的数据写入存储器. 这两种指令都用ALU0中的值作为访存地址, 但在上一周期已经计算好了.

- 存储器访问指令 [操作](#)

$LMD \leftarrow Mem[ALUo]$

或者 $Mem[ALUo] \leftarrow B$

- 分支指令 [操作](#)

if (cond) $PC \leftarrow ALUo$ else $PC \leftarrow NPC$

如cond为真, 表明转移成功, 把ALU0中的转移目标地址送入PC中.

➤ 写回周期 (WB)

不同的指令在写回周期完成的工作也不一样。

- 寄存器—寄存器ALU指令 [操作](#)

$Regs[rd] \leftarrow ALUo$

- 寄存器—立即数ALU指令 [操作](#)

$Regs[rt] \leftarrow ALUo$

- load指令 [操作](#)

$Regs[rt] \leftarrow LMD$

上述结果都写入通用寄存器组, 可能是ALU的计算机结果(ALU0中), 也可能来自存储器(LMD中).

写入目标寄存器由指令中的rd或rt字段支出, 并由操作码决定.

不同操作周期间设置存储单元, 来保存当前指令的执行结果或上周期产生且后面周期需用到的结果.

PC、通用寄存器、存储单元保存指令之间的结果; 临时寄存器IR、NPC、A、B、Imm、cond、ALU0、LMD用于单条指令的执行过程保存中间结果

。如果把IF周期内加法器与EX周期的ALU合并, 共享同一个ALU, 并把指令和数据存储器合并, 方案经济。

3. 不采用单周期实现方案的主要原因

- 对于大多数CPU来说，单周期实现效率很低，因为不同的指令所需完成的操作差别相当大，因而所需要的时钟周期时间也大不一样。
- 单周期实现时，需要重复设置某些功能部件，而在多周期实现方案中，这些部件是可以共享的。

3.5.2 基本的MIPS流水线

- 每一个时钟周期完成的工作看作是流水线的一段，每个时钟周期启动一条新的指令。

1. 流水实现的数据通路

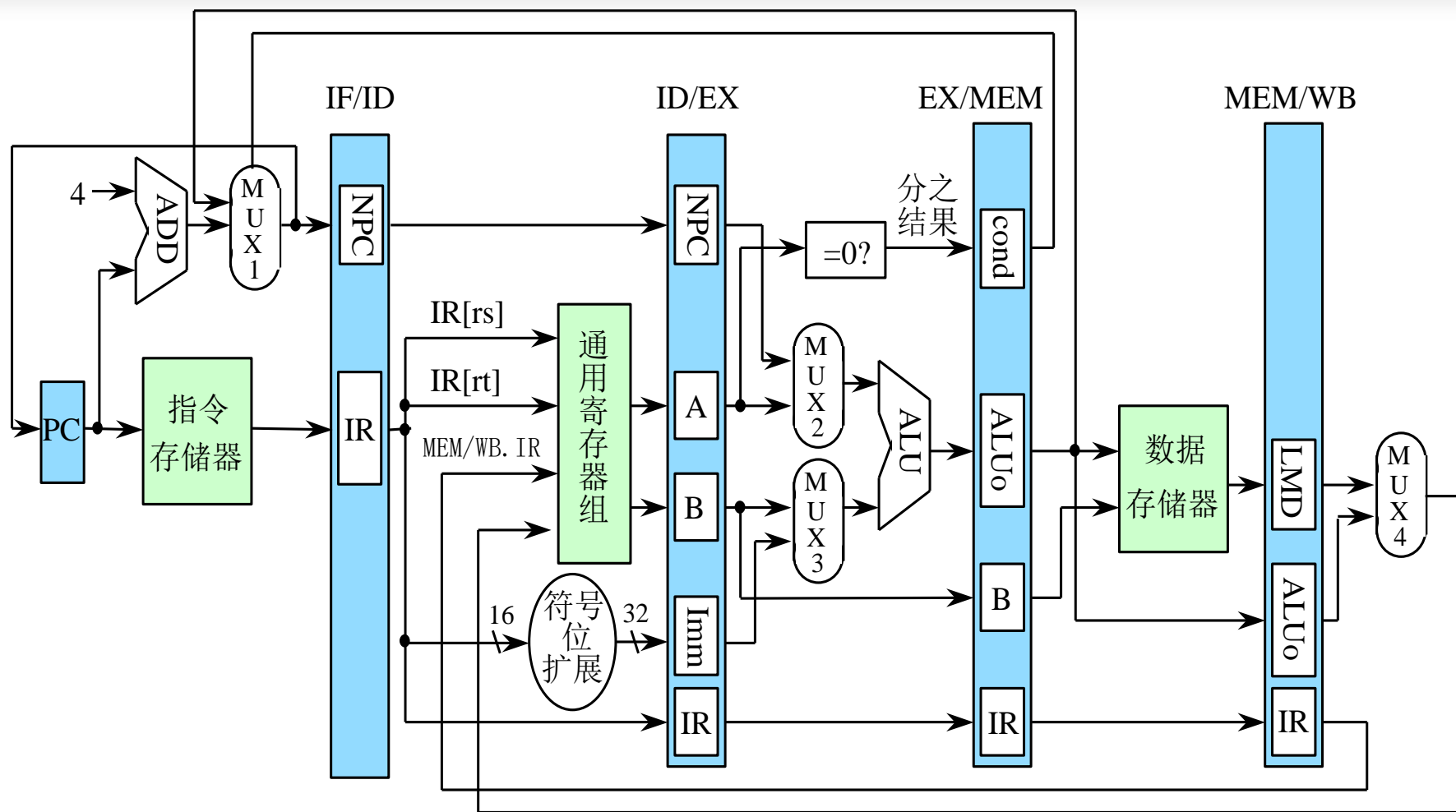
- 设置了流水寄存器

- 段与段之间设置流水寄存器
- 流水寄存器的命名

用其相邻的两个段的名称拼合而成。

例如：ID段与EX段之间的流水寄存器用ID/EX表示

- 每个流水寄存器是由若干个寄存器构成的



流水实现的数据通路

- 寄存器的命名形式为: $x.y$
- 所包含的字段的命名形式为: $x.y[s]$

其中: x : 流水寄存器名称

y : 具体寄存器名称

s : 字段名称

例如:

ID/EX. IR: 流水寄存器ID/EX中的子寄存器IR

IRID/EX. IR[op]: 该寄存器的op字段 (即操作码字段)

- 流水寄存器的作用
 - 将各段的工作隔开, 使得它们不会互相干扰。
 - 保存相应段的处理结果。

例如：

EX/MEM. ALUo：保存EX段ALU的运算结果

MEM/WB. LMD：保存MEM段从数据存储器读出的数据

- 向后传递后面将要用到的数据或者控制信息

所有有用的数据和控制信息每个时钟周期

会随着指令在流水线中的流动往后流动一段。只保存后面需要的数据和信息。如果把PC看作IF段的流水寄存器，则每个段都有一个流水寄存器，位于该流水段的前面，提供指令在该段执行所需的数据和控制信息。

- 增加了向后传递IR和从MEM/WB. IR回送到通用寄存器组的连接。
- 将对PC的修改移到了IF段，以便PC能及时地加4，为取下一条指令做好准备。

在IF和ID段，所有指令的操作一样，从EX段开始才区分不同的指令。

2. 每一个流水段进行的操作

- $IR[rs] = IR_{6..10}$
- $IR[rt] = IR_{11..15}$
- $IR[rd] = IR_{16..20}$

流水线的每个流水段的操作

流水段	所有指令类型		
IF	IF/ID. IR \leftarrow Mem[PC] IF/ID. NPC, PC \leftarrow (if ((EX/MEM. IR[op] == branch) & EX/MEM. cond) {EX/MEM. ALUo} else {PC+4}) ; (动画演示)		
ID	ID/EX. A \leftarrow Regs[IF/ID. IR[rs]]; ID/EX. B \leftarrow Regs[IF/ID. IR[rt]]; ID/EX. NPC \leftarrow IF/ID. NPC; ID/EX. IR \leftarrow IF/ID. IR; ID/EX. Imm \leftarrow (IF/ID. IR ₁₆) ¹⁶ ##IF/ID. IR _{16..31} ; (动画演示)		
	ALU 指令	load/store 指令	分支指令
EX	EX/MEM. IR \leftarrow ID/EX. IR; EX/MEM. ALUo \leftarrow ID/EX. A <i>func</i> ID/EX. B 或 EX/MEM. ALUo \leftarrow ID/EX. A <i>op</i> ID/EX. Imm; (动画演示)	EX/MEM. IR \leftarrow ID/EX. IR; EX/MEM. ALUo \leftarrow ID/EX. A + ID/EX. Imm; EX/MEM. B \leftarrow ID/EX. B; (动画演示)	EX/MEM. IR \leftarrow ID/EX. IR; EX/MEM. ALUo \leftarrow ID/EX. NPC + ID/EX. Imm<<2; EX/MEM. cond \leftarrow (ID/EX. A ==0) ; (动画演示)

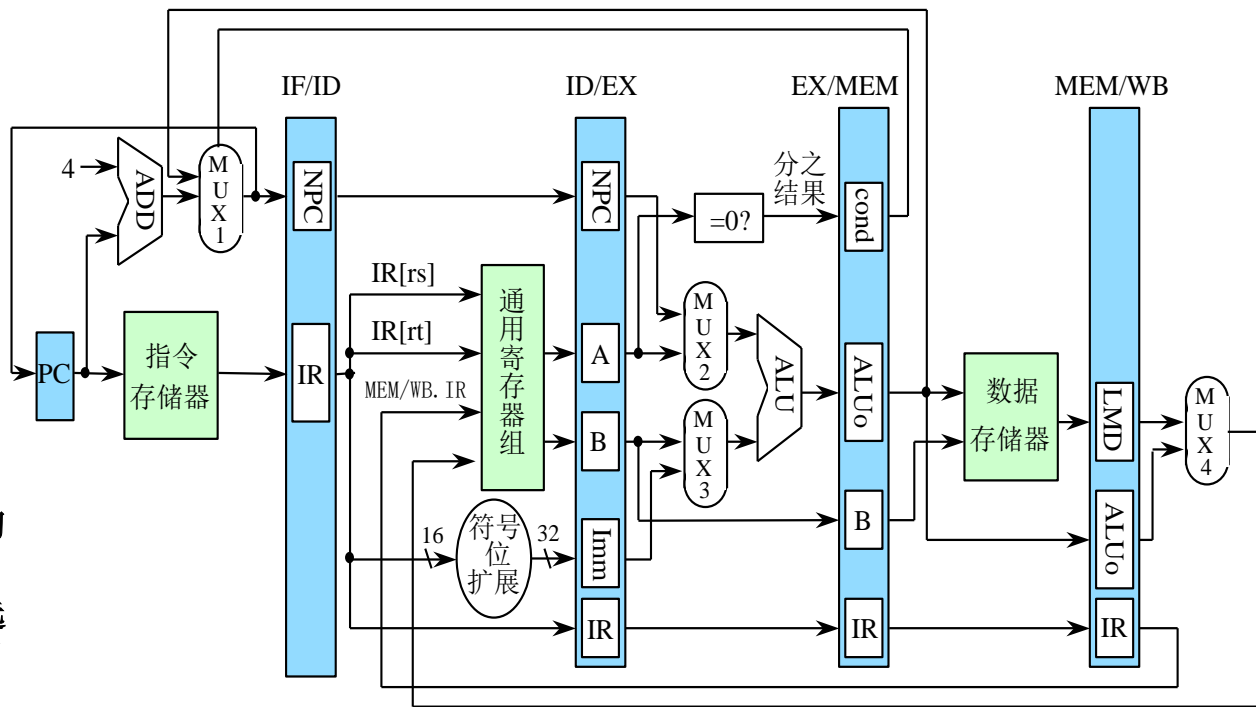
流水线的每个流水段的操作

流水段	任何指令类型		
	ALU 指令	load/store 指令	分支指令
MEM	$\text{MEM/WB. IR} \leftarrow \text{EX/MEM. IR};$ $\text{MEM/WB. ALU}_o \leftarrow$ $\text{EX/MEM. ALU}_o;$ (动画演示)	$\text{MEM/WB. IR} \leftarrow \text{EX/MEM. IR};$ $\text{MEM/WB. LMD} \leftarrow$ $\text{Mem}[\text{EX/MEM. ALU}_o];$ 或 $\text{Mem}[\text{EX/MEM. ALU}_o] \leftarrow$ $\text{EX/MEM. B};$ (动画演示)	
WB	$\text{Regs}[\text{MEM/WB. IR}[\text{rd}]] \leftarrow$ $\text{MEM/WB. ALU}_o;$ 或 $\text{Regs}[\text{MEM/WB. IR}[\text{rt}]] \leftarrow$ $\text{MEM/WB. ALU}_o;$ (动画演示)	$\text{Regs}[\text{MEM/WB. IR}[\text{rt}]] \leftarrow$ $\text{MEM/WB. LMD};$ (动画演示)	

3 流水线的控制

➤ 主要是如何控制4个多路选择器。

- **MUX2:** 若ID/EX. IR中的指令是分支指令, 则选择ID/EX. NPC, 否则选ID/EX. A。
- **MUX3:** 若ID/EX. IR中的指令是寄存器—寄存器型ALU指令, 则选ID/EX. B, 否则选ID/EX. Imm。
- **MUX1:** 若EX/MEM. IR中的指令是分支指令, 而且EX/MEM. cond为真, 则选EX/MEM. ALUo, 即分支目标地址, 否则选PC+4。
- **MUX4:** 若MEM/WB. IR中的指令是load指令, 则选MEM/WB. LMD, 否则选MEM/WB. ALUo。



- **第5个多路器：**从MEM/WB回传至通用寄存器组的写入地址应该是从MEM/WB. IR[rd] 和MEM/WB. IR[rt] 中选一个。
 - 寄存器—寄存器型ALU指令：选择MEM/WB. IR[rd] ；
 - 寄存器—立即数型ALU指令和load指令：选择MEM/WB. IR[rt] 。

➤ 解决数据冲突的问题

- 所有的数据冲突均可以在ID段检测到。

如果存在数据冲突，就在相应的指令流出ID段之前将之暂停。

完成该工作的硬件称为流水线的互锁机制。

- 在ID段确定需要什么样的定向，并设置相应的控制。
降低流水线的硬件复杂度。不必挂起已经改变了机器状态的指令)
- 也可以在使用操作数的那个时钟周期的开始检测冲突和确定必需的定向。
- 检测冲突是通过比较寄存器地址是否相等来实现的。

举例： load互锁

由于使用load的结果而引起的流水线互锁称为load互锁。

在ID段检测是否存在RAW冲突
(这时load指令在EX段)

ID/EX中的操作码 (ID/EX. IR[op])	IF/ID中的操作码 (IF/ID. IR[op])	比较的操作数字段
load	RR ALU	ID/EX. IR[rt]=IF/ID. IR[rs]
load	RR ALU	ID/EX. IR[rt]=IF/ID. IR[rt]
load	load、store ALU立即数或分支	ID/EX. IR[rt]=IF/ID. IR[rs]

- 若检测到RAW冲突，流水线互锁机制必须在流水线中插入停顿，并使当前正处于IF段和ID段的指令不再前进。
 - 将ID/EX. IR中的操作码改为全0
(全0表示空操作)
 - IF/ID寄存器的内容回送到自己的入口

➤ 定向逻辑

- 要考虑的情况更多
- 通过比较流水寄存器中的寄存器地址来确定

例如：

- 若 $(ID/EX. IR. op == RR_ALU) \& (EX/MEM. IR. op == RR_ALU) \& (ID/EX. IR[rt] == EX/MEM. IR[rd])$,

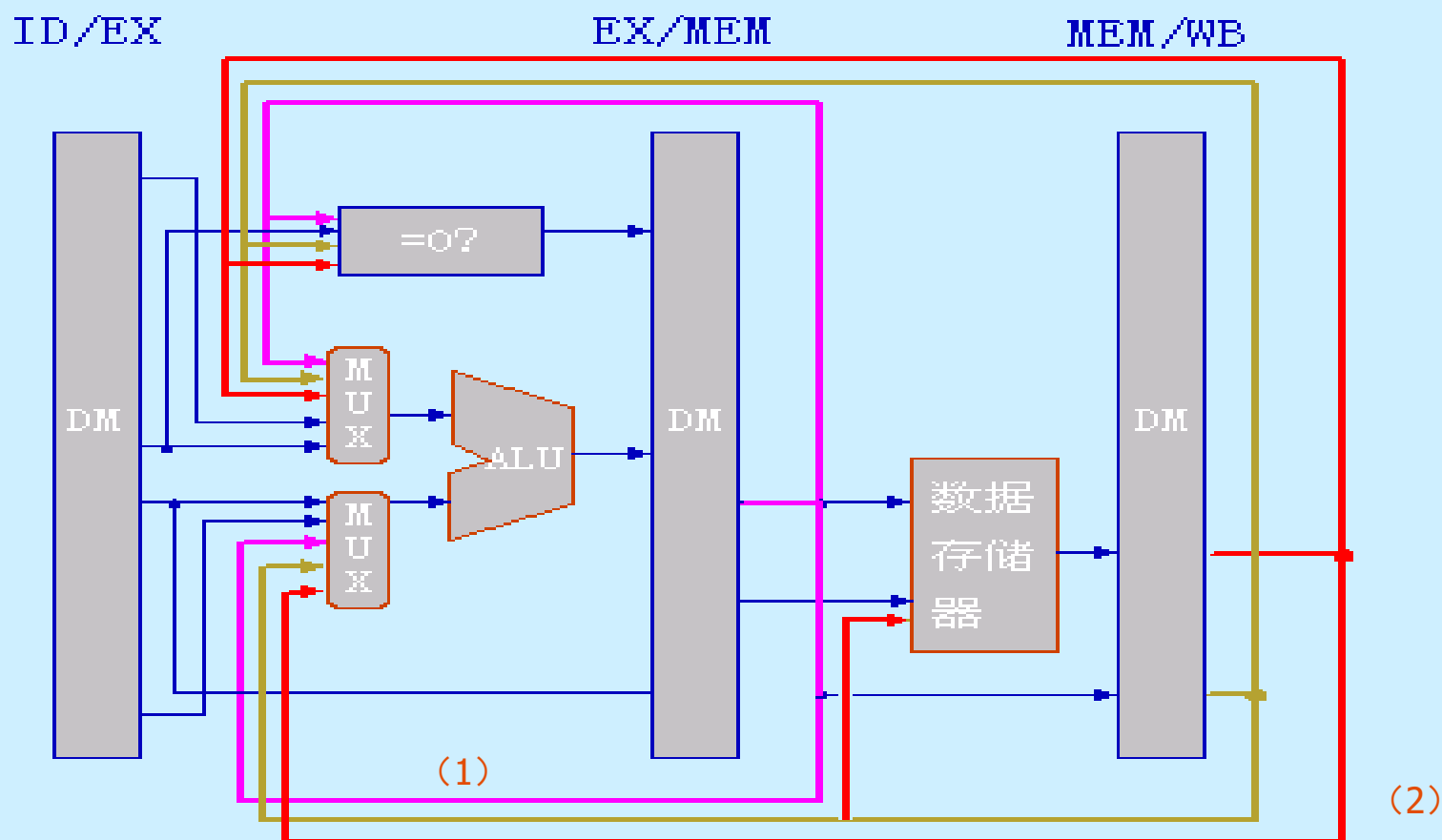
EX和MEM段中的指令都是RR ALU，而且MEM段中指令的目标寄存器地址与EX段中指令的第二个源操作数地址相同，则 EX/MEM. ALU_o定向到ALU的下面一个输入（如图（1））。

- 若 $(ID/EX. IR[op] == RR_ALU) \& (MEM/WB. IR[op] == load) \& (ID/EX. IR[rt] == MEM/WB. IR[rt])$,

EX段中的指令都是RR ALU，WB段中指令是LOAD，且LOAD指令的目标寄存器地址与EX段中指令的第二个源操作数地址相同，则 把MEM/WB. LMD定向到ALU的下面一个输入（如图（2））。

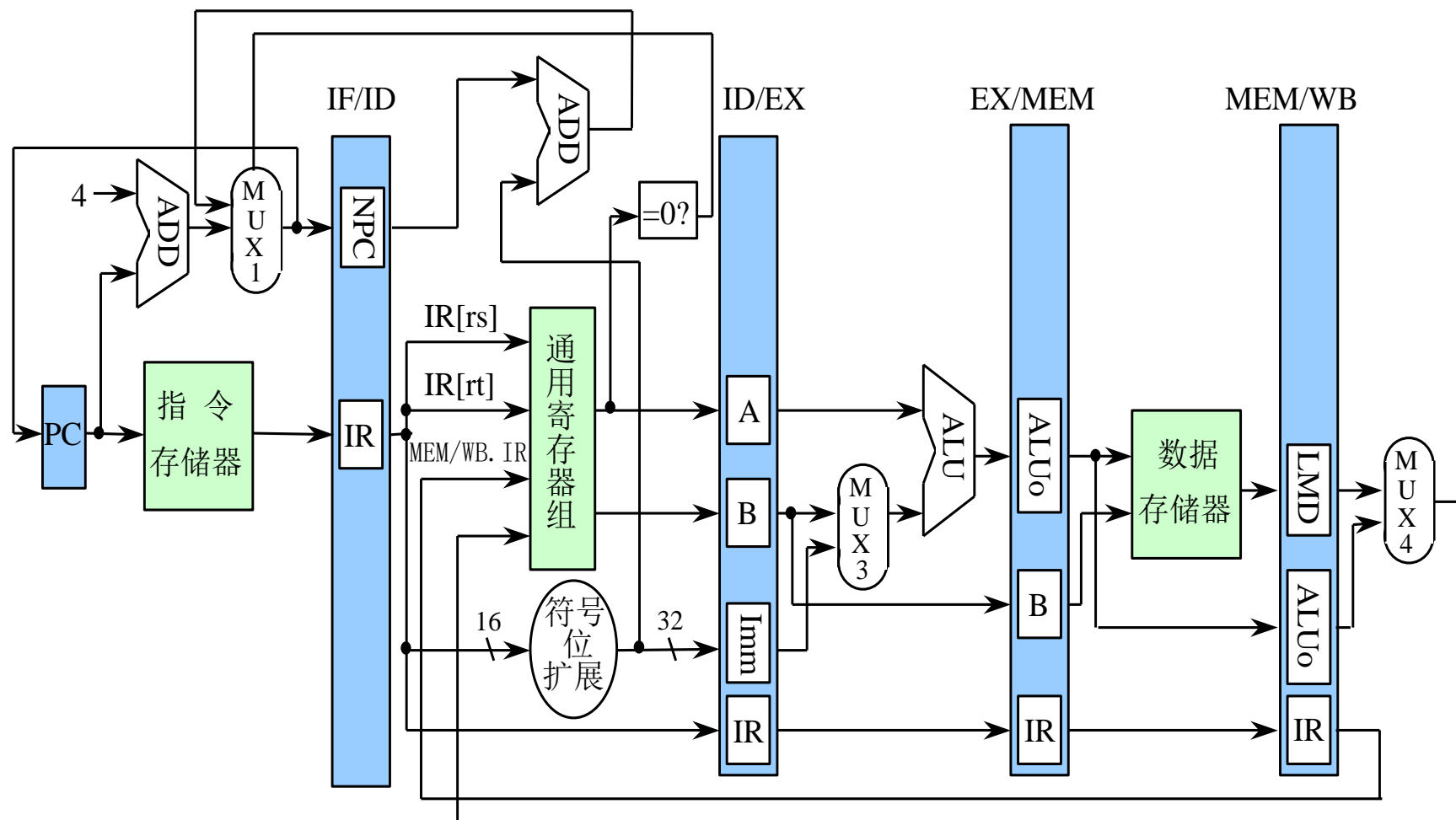
- 流水线的增设的定向路径。分别把ALU的运算结果和访问存储器得到的数据定向到ALU的两个输入端。

定向路径



3. 控制冲突

- 分支指令的条件测试和分支目标地址计算在EX段完成，对PC的修改在MEM段完成。
- 它所带来的分支延迟是3个时钟周期。
- 减少分支延迟：做如下改进
(把上述工作提前到ID段进行)
 - 如果考虑BEQZ和BNEZ，把分支预测提前到ID段进行。在ID段增设一个加法器，用于计算分支目标地址。
 - 把条件测试“=0?”的逻辑电路移到ID段。
 - 这些结果直接回送到IF段的MUX1。
 - 改进后的流水线对分支指令的处理移到ID段。分支延迟变为一个时钟周期。



改进后流水线的分支操作

流水段	分支指令操作
IF	$\begin{aligned} & \text{IF/ID. IR} \leftarrow \text{Mem}[\text{PC}]; \\ & \text{IF/ID. NPC, PC} \leftarrow \\ & \quad (\text{if}((\text{IF/ID. IR}[\text{op}] = \text{branch}) \& ((\text{Regs}[\text{IF/ID. IR}[\text{rs}]] = 0))) \\ & \quad \{ \text{IF/ID. NPC} + (\text{IF/ID. IR}_{16})^{16} \# \# (\text{IF/ID. IR}_{16..31} \ll 2) \} \\ & \quad \text{else } \{ \text{PC} + 4 \}); \end{aligned}$
ID	$\begin{aligned} & \text{ID/EX. A} \leftarrow \text{Regs}[\text{IF/ID. IR}[\text{rs}]]; \quad \text{ID/EX. B} \leftarrow \text{Regs}[\text{IF/ID. IR}[\text{rt}]]; \\ & \text{ID/EX. IR} \leftarrow \text{IF/ID. IR}; \\ & \text{ID/EX. Imm} \leftarrow (\text{IF/ID. IR}_{16})^{16} \# \# \text{IF/ID. IR}_{16..31}; \end{aligned}$
EX	
MEM	
WB	

3.6 向量处理机

- 在流水线处理机中，设置向量数据表示和相应的向量指令，称为**向量处理机**。
- 不具有向量数据表示和相应的向量指令的流水线处理机，称为**标量处理机**。

3.6.1 向量处理方式

以计算表达式 $D=A \times (B+C)$ 为例

A 、 B 、 C 、 D —— 长度为 N 的向量

1. 横向(水平)处理方式

- 向量计算是按行的方式从左到右横向地进行。

- 先计算: $d_1 \leftarrow a_1 \times (b_1 + c_1)$

- 再计算: $d_2 \leftarrow a_2 \times (b_2 + c_2)$

-

- 最后计算: $d_N \leftarrow a_N \times (b_N + c_N)$

- 组成循环程序进行处理。

$$k_i \leftarrow b_i + c_i$$

$$d_i \leftarrow k_i \times a_i$$

- 数据相关: N 次 功能切换: $2N$ 次

- 不适合于向量处理机的并行处理。

2. 纵向(垂直)处理方式

- 向量计算是按列的方式从上到下纵向地进行。

$$\begin{array}{cc} \text{先计算} & \left\{ \begin{array}{l} k_1 \leftarrow b_1 + c_1 \\ \dots\dots \\ k_N \leftarrow b_N + c_N \end{array} \right. & \text{再计算} & \left\{ \begin{array}{l} d_1 \leftarrow k_1 \times a_1 \\ \dots\dots \\ d_N \leftarrow k_N \times a_N \end{array} \right. \end{array}$$

- 表示成向量指令：

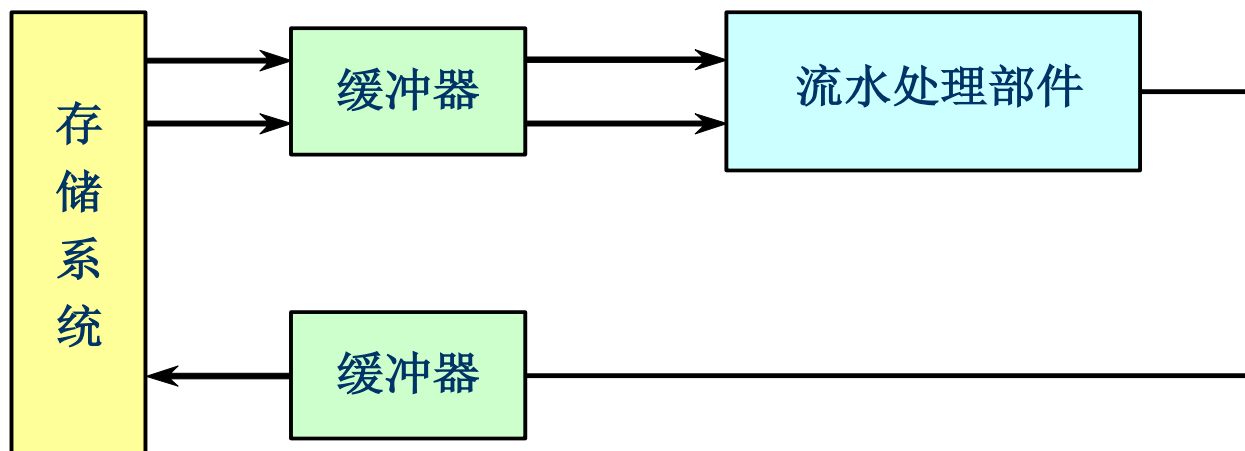
$$K = B + C$$

$$D = K \times A$$

- 两条向量指令之间：

数据相关：1次 功能切换：1次

- 对处理机结构的要求：存储器—存储器结构
- 向量指令的源向量和目的向量都存放在存储器中，运算的中间结果需要送回存储器。
 - 存储器—存储器型操作的运算流水线
 - 例如：STAR-100、CYBER-205



3. 纵横(分组)处理方式

- 又称为**分组处理方式**。
- 把向量分成若干组，组内按纵向方式处理，依次处理各组。
- 对于上述的例子，设：

$$N = S \times n + r$$

- 其中 N 为向量长度， S 为组数， n 为每组的长度， r 为余数。
- 若余下的 r 个数也作为一组处理，则共有 $S+1$ 组。
- 运算过程为：

- 先算第1组:

$$k_{1\sim n} \leftarrow b_{1\sim n} + c_{1\sim n}$$

$$d_{1\sim n} \leftarrow k_{1\sim n} \times a_{1\sim n}$$

- 再算第2组:

$$k_{(n+1)\sim 2n} \leftarrow b_{(n+1)\sim 2n} + c_{(n+1)\sim 2n}$$

$$d_{(n+1)\sim 2n} \leftarrow k_{(n+1)\sim 2n} \times a_{(n+1)\sim 2n}$$

- 依次进行下去, 直到最后一组: 第 $S+1$ 组。
- 每组内各用两条向量指令。

数据相关: 1次 功能切换: 2次

- 对处理机结构的要求：寄存器—寄存器结构
 - 设置能快速访问的向量寄存器，用于存放源向量、目的向量及中间结果，让运算部件的输入、输出端都与向量寄存器相联，构成寄存器—寄存器型操作的运算流水线。
 - 典型的寄存器—寄存器结构的向量处理机
美国的CRAY-1、我国的YH-1巨型机

3.6.2 向量处理机的结构

➤ 以CRAY-1机为例

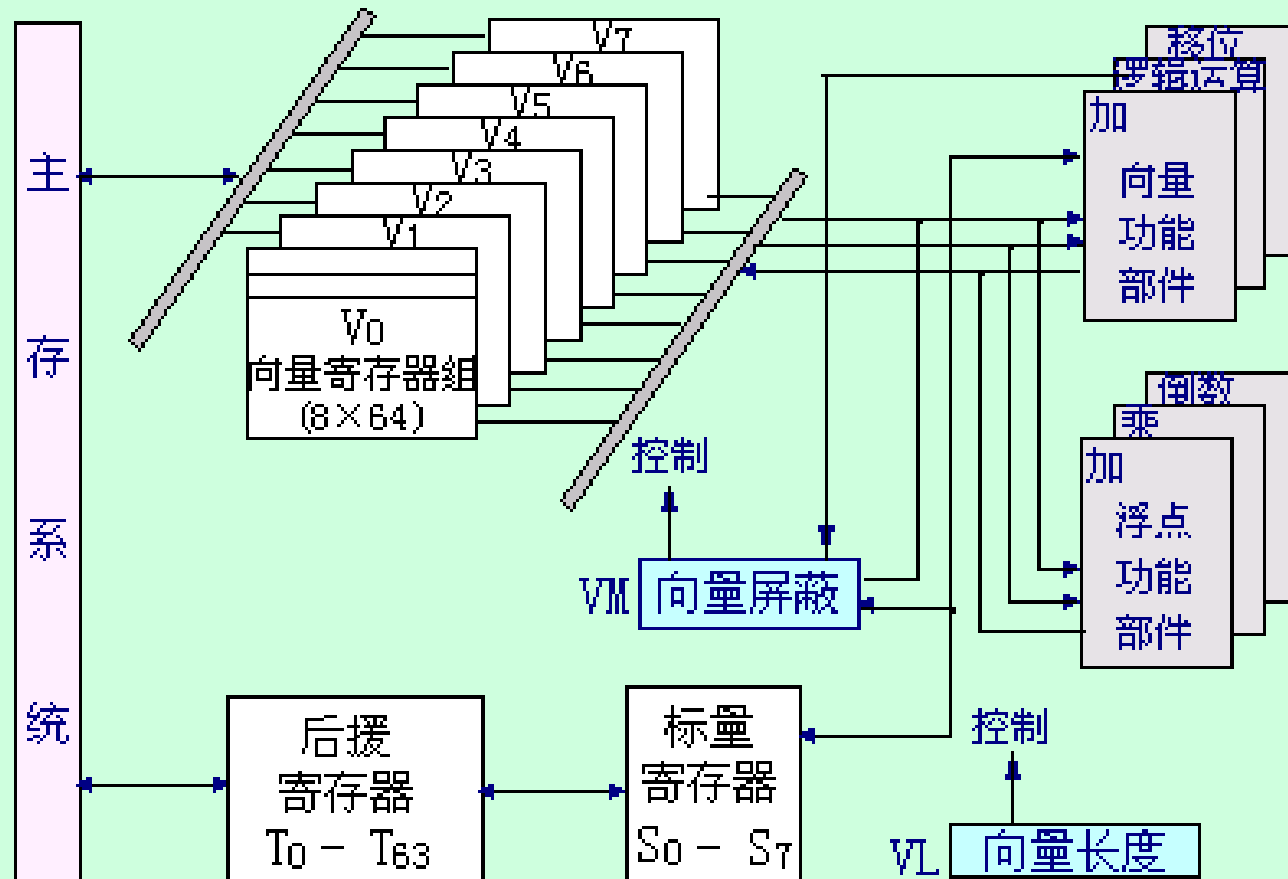
- 美国CRAY公司
- 1976年
- 每秒1亿次浮点运算
- 时钟周期: 12.5 ns

1. CRAY-1的基本结构

➤ 功能部件

共有12条可并行工作的单功能流水线，可分别流水地进行地址、向量、标量的各种运算。

CRAY-1的基本结构



- 6个单功能流水部件：进行向量运算
 - 整数加（3拍）
 - 逻辑运算（2拍）
 - 移位（4拍）
 - 浮点加（6拍）
 - 浮点乘（7拍）
 - 浮点迭代求倒数（14拍）

括号中的数字为其流水经过的时间，每拍为一个时钟周期，即12.5 ns。

➤ 向量寄存组V

- 由512个64位的寄存器组成，分成8块。
- 编号： $V_0 \sim V_7$
- 每一个块称为一个向量寄存器，可存放一个长度（即元素个数）不超过64的向量。
- 每个向量寄存器可以每拍向功能部件提供一个数据元素，或者每拍接收一个从功能部件来的结果元素。

➤ 标量寄存器S和快速暂存器T

- 标量寄存器有8个： $S_0 \sim S_7$ 64位
- 快速暂存器T用于在标量寄存器和存储器之间提供缓冲。

➤ 向量屏蔽寄存器VM

- 64位，每一位对应于向量寄存器的一个单元。
- 作用：用于向量的归并、压缩、还原和测试操作、对向量某些元素的单独运算等。

2. CRAY-1向量处理的一个显著特点

- 每个向量寄存器 V_i 都有连到6个向量功能部件的单独总线。
- 每个向量功能部件也都有把运算结果送回向量寄存器组的总线。

➤ 只要不出现 V_i 冲突和功能部件冲突，各 V_i 之间和各功能部件之间都能并行工作，大大加快了向量指令的处理。

- V_i 冲突：并行工作的各向量指令的源向量或结果向量使用了相同的 V_i 。

例如：源向量相同

$$V_3 \leftarrow V_1 + V_2$$

$$V_5 \leftarrow V_4 \wedge V_1$$

- 功能部件冲突：并行工作的各向量指令要使用同一个功能部件。

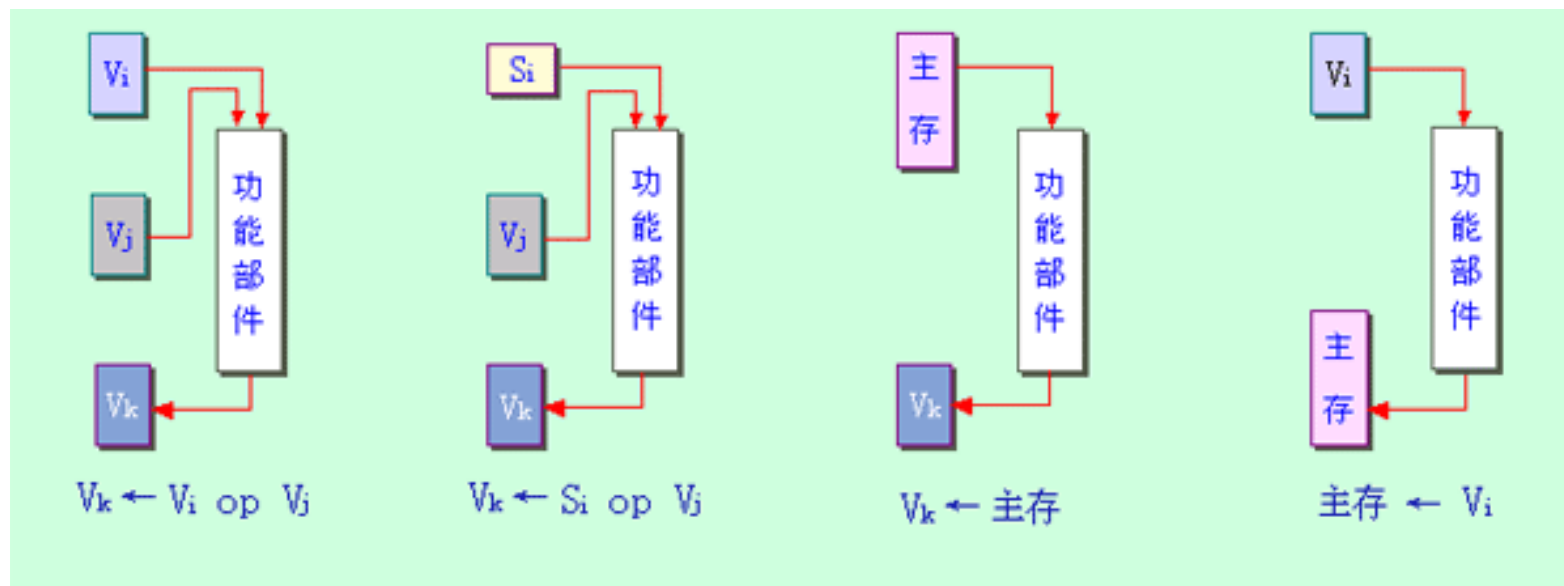
例如：都需使用乘法功能部件

$$V_3 \leftarrow V_1 \times V_2$$

$$V_5 \leftarrow V_4 \times V_6$$

3. CRAY-1向量指令类型

- $V_k \leftarrow V_i \text{ op } V_j$
- $V_k \leftarrow S_i \text{ op } V_j$
- $V_k \leftarrow \text{主存}$
- $\text{主存} \leftarrow V_i$



3.6.3 提高向量处理机性能的方法

提高向量处理机性能的方法

- 设置多个功能部件，使它们并行工作。
- 采用链接技术，加快一串向量指令的执行。
- 采用循环开采技术，加快循环的处理。
- 采用多处理机系统，进一步提高性能。

1. 设置多个功能部件

- 设置多个独立的功能部件。这些部件能并行工作，并各自按流水方式工作，从而形成了多条并行工作的运算操作流水线。

例如：CRAY-1向量处理机有4组12个单功能流水部件：

- 向量部件：向量加，移位，逻辑运算
- 浮点部件：浮点加，浮点乘，浮点求倒数
- 标量部件：标量加，移位，逻辑运算，
数“1”/计数
- 地址运算部件：整数加，整数乘

2. 链接技术

- 链接特征：具有先写后读相关的两条指令，在不出现功能部件冲突和源向量冲突的情况下，可以把功能部件链接起来进行流水处理，以达到加快执行的目的。
- 链接特性的实质
把流水线定向的思想引入到向量执行过程的结果。

例3.3 在CRAY-1上用链接技术进行向量运算

$$D=A \times (B+C)$$

假设向量长度 $N \leq 64$ ，向量元素为浮点数，且向量 B 、 C 已存放在 V_0 和 V_1 中。

画出链接示意图，并分析非链接执行和链接执行两种情况下的执行时间。

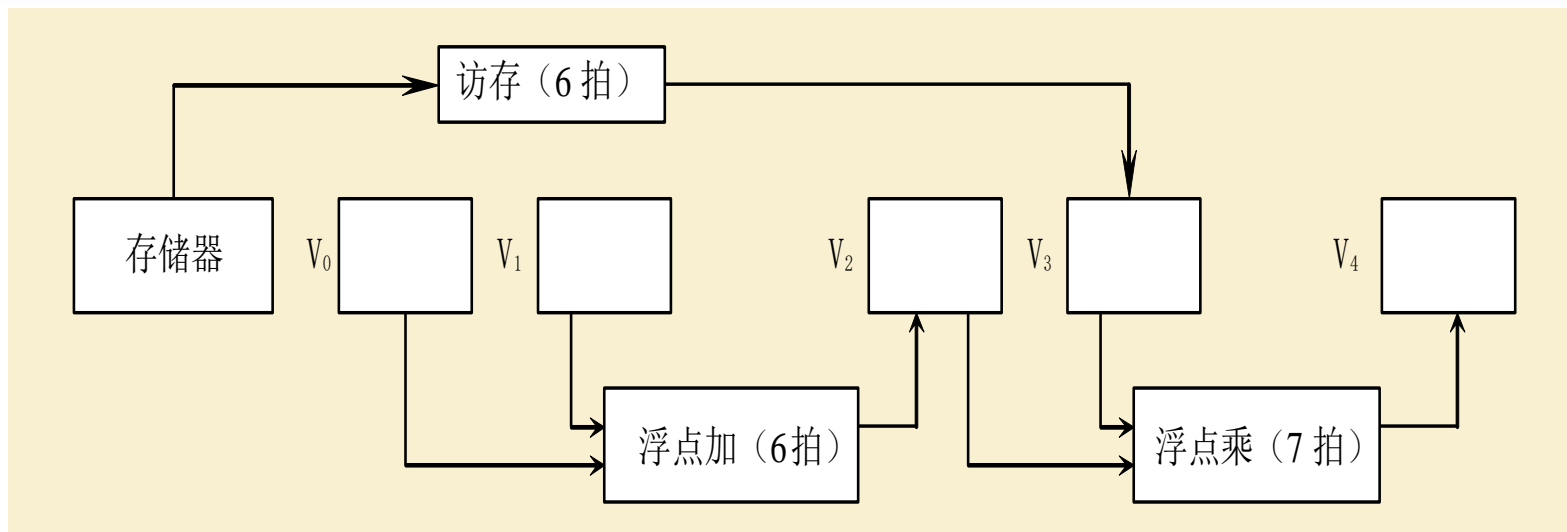
解 用以下三条向量完成上述运算：

$V_3 \leftarrow$ 存储器 // 访存取向量A

$V_2 \leftarrow V_0 + V_1$ // 向量B和向量C进行浮点加

$V_4 \leftarrow V_2 \times V_3$ // 浮点乘，结果存入 V_4

第1、2条指令无寄存器使用冲突，无功能部件冲突，可并行执行。第3条指令与第1、2条指令之间存在先写后读相关，不存在功能部件冲突，因此可以与第1、2条指令链接执行。



- **假设：**把向量数据元素送往向量功能部件以及把结果存入向量寄存器需要一拍时间，从存储器中把数据送入访存功能部件需要一拍时间。

- 3条指令全部用串行方法执行，则执行时间为：

$$\begin{aligned} & [(1+6+1) + N-1] + [(1+6+1) + N-1] \\ & \quad + [(1+7+1) + N-1] = 3N + 22 \quad (\text{拍}) \end{aligned}$$

- 前两条指令并行执行，然后再串行执行第3条指令，则执行时间为：

$$\begin{aligned} & [(1+6+1) + N-1] + [(1+7+1) + N-1] \\ & \quad = 2N + 15 \quad (\text{拍}) \end{aligned}$$

- 第1、2条向量指令并行执行，并与第3条指令链接执行。

- 从访存开始到把第一个结果元素存入 V_4 所需的拍数（亦称为**链接流水线的建立时间**）为：

$$[(1+6+1)] + [(1+7+1)] = 17 \quad (\text{拍})$$

- 3条指令的执行时间为：

$$\begin{aligned} & [(1+6+1)] + [(1+7+1)] + (N-1) \\ & = N+16 \quad (\text{拍}) \end{aligned}$$

➤ 进行向量链接的要求

保证：无向量寄存器使用冲突和无功能部件使用冲突

- 只有在前一条指令的第一个结果元素送入结果向量寄存器的那一个时钟周期才可以进行链接。
- 当一条向量指令的两个源操作数分别是两条先行指令的结果寄存器时，要求先行的两条指令产生运算结果的时间必须相等，即要求有关功能部件的通过时间相等。
- 要进行链接执行的向量指令的向量长度必须相等，否则无法进行链接。

3. 分段开采技术

如果向量的长度大于向量寄存器的长度，
该如何处理呢？

- 当向量的长度大于向量寄存器的长度时，必须把长向量分成长度固定的段，然后循环分段处理，每一次循环只处理一个向量段。
- 这种技术称为分段开采技术。
 - 由系统硬件和软件控制完成，对程序员是透明的。

例3.4 设 A 和 B 是长度为 N 的向量，考虑在Cray-1向量处理器上实现以下的循环操作：

DO 10 I = 1, N

10 A(I) = 5.0 * B(I) + C

➤ 当 $N \leq 64$ 时，可以用以下指令序列：

$S_1 \leftarrow 5.0$	； 将常数5.0送入标量寄存器 S_1
$S_2 \leftarrow 1.0$	； 将常数1.0送入标量寄存器 S_2
$VL \leftarrow N$	； 在向量长度寄存器 VL 中设置向量长度 N
$V_0 \leftarrow B$	； 从存储器中将向量 B 读入向量寄存器 V_0
$V_1 \leftarrow S_1 \times V_0$	； 向量 B 中的每个元素分别和常数 S_1 相乘
$V_2 \leftarrow S_2 + V_1$	； 向量 V_1 中的每个元素分别和常数 S_2 相加
$A \leftarrow V_2$	； 将计算结果从向量寄存器 V_2 存入存储 器的向量 A

➤ 当 $N > 64$ 时，就需要进行分段开采。

□ 循环次数 K ：

$$K = \left\lfloor \frac{N}{64} \right\rfloor$$

□ 余数 L ：

$$L = N - 64 \times \left\lfloor \frac{N}{64} \right\rfloor$$

$S_1 \leftarrow 5.0$; 将常数5.0送入标量寄存器 S_1
 $S_2 \leftarrow 1.0$; 将常数1.0送入标量寄存器 S_2
 $VL \leftarrow L$; 在向量长度寄存器VL中设置向量长度L
 $V_0 \leftarrow B$; 从存储器中将向量 $B[0..L-1]$ 读入向量
; 寄存器 V_0
 $V_1 \leftarrow S_1 * V_0$; 向量B中的每个元素分别和常数 S_1 相乘
 $V_2 \leftarrow S_2 + V_1$; 向量 V_1 中的每个元素分别和常数 S_2 相加
 $A \leftarrow V_2$; 将计算结果从向量寄存器 V_2 存入存储器
; 的向量 $A[0..L-1]$

处理余
数部分,
计算L
个元素

```
For (I=0 to K-1) {  
   $V_0 \leftarrow B$       ; 从存储器中将向量 $B[L+I*64..L+I*64+63]$   
                    ; 读入向量寄存器 $V_0$   
  
   $V_1 \leftarrow S_1 * V_0$  ; 向量B中的每个元素分别和常数 $S_1$   
                    ; 相乘;  
  
   $V_2 \leftarrow S_2 + V_1$  ; 向量 $V_1$ 中的每个元素分别和常数 $S_2$   
                    ; 相加  
  
   $A \leftarrow V_2$       ; 将计算结果 $V_2$ 存入存储器的向量  
                    ;  $A[L+I*64 \cdots L+I*64+63]$   
}
```

循环
K次,
分段
处理

4. 采用多处理机系统

许多新型向量处理机系统采用了多处理机系统结构。例如：

- CRAY-2
 - 包含了4个向量处理机
 - 浮点运算速度最高可达1800MFLOPS
- CRAY Y-MP、C90
 - 最多可包含16个向量处理机

3.6.4 向量处理机的性能评价

衡量向量处理机性能的主要参数：

1. 向量指令的处理时间

➤ 执行一条向量长度为 n 的向量指令所需的时间为

$$T_{vp} = T_s + T_{vf} + (n - 1)T_c$$

□ T_s ：向量流水线的建立时间

□ T_{vf} ：向量流水线的流过时间

它是从向量指令开始译码算起，到第一对向量元素流过流水线直到产生第一个结果元素所需的时间。

□ T_c ：流水线瓶颈段的执行时间

- 如果流水线不存在“瓶颈”，每段的执行时间等于一个时钟周期，则上式可以写为：

$$T_{vp} = [s + e + (n - 1)]T_{clk}$$

- s : 向量流水线的建立时间所对应的时钟周期数
- e : 向量流水线的流过时间所对应的时钟周期数
- T_{clk} : 时钟周期时间

- 也可以将上式改写为：

$$T_{vp} = [T_{start} + n]T_{clk}$$

- T_{start} : 向量功能部件启动所需的时钟周期数

- 对于一组向量指令而言，其执行时间主要取决于三个因素：
 - 向量的长度
 - 向量操作之间是否链接
 - 向量功能部件的冲突和数据的冲突性
- 把几条能在同一个时钟周期内一起开始执行的向量指令集合称为一个编队。
 - 可以看出，同一个编队中的向量指令之间一定不存在流水向量功能部件的冲突和数据的冲突。

例3.5 假设每种向量功能部件只有一个，那么下面的一组向量指令能分成几个编队？

LV	V1, Rx
MULTSV	V2, R0, V1
LV	V3, Ry
ADDV	V4, V2, V3
SV	Ry, V4

解：分为4个编队

- 第一编队：LV
- 第二编队：MULTSV; LV
- 第三编队：ADDV
- 第四编队：SV

- 一个编队内所有向量指令执行完毕所要的时间为：
(假设第 i 个编队中所有向量指令处理的向量元素个数均为 n)

$$\begin{aligned} T_c^i &= \max_j [T_{start}^{ij} + n] T_{clk} \\ &= (T_{start}^i + n) T_{clk} \end{aligned}$$

- T_c^i : 第 i 个编队的执行时间
- T_{start}^{ij} : 第 i 个编队中第 j 条指令所使用向量功能部件的启动时钟周期数

➤ 编队后的向量指令序列总的执行时间为：

$$\begin{aligned}T_v &= \sum_{i=1}^m T_c^i \\&= \sum_{i=1}^m (T_{start}^i + n)T_{clk} \\&= (\sum_{i=1}^m T_{start}^i + mn)T_{clk} \\&= (T_{start} + mn)T_{clk}\end{aligned}$$

- m : 向量指令序列编队的个数
- T_{start} : 向量指令序列编队总的启动时钟周期数

- 编队并采用分段开采技术后，向量指令序列执行所需的总的时钟周期数为：

$$\begin{aligned} T_n &= \left\lfloor \frac{n}{MVL} \right\rfloor \times (T_{loop} + T_{start} + m \times MVL) \\ &\quad + [T_{loop} + T_{start} + m \times (n - \left\lfloor \frac{n}{MVL} \right\rfloor \times MVL)] \\ &= \left\lfloor \frac{n}{MVL} \right\rfloor \times (T_{loop} + T_{start}) + mn \end{aligned}$$

- T_{loop} : 分段开采所需的额外的时间开销
- MVL : 向量处理机的向量寄存器长度

3.6 向量处理机

例3.6 在某向量处理机上执行DAXPY的向量指令序列，也即计算双精度浮点向量表达式。

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

其中 X 和 Y 是双精度浮点向量，最初保存在外部存储器中， a 是一个双精度浮点常数，已存放在浮点寄存器F0中。计算该表达式的向量指令序列如下：

```
LV      V1, Rx //从存储器中载入向量Rx
MULTFV  V2, F0, V1 //向量V1和浮点寄存器F0乘
LV      V3, Ry //从存储器中载入向量Ry
ADDV    V4, V2, V3 //向量V2与V3 加，结果存V4
SV      Ry, V4 //结果保存到存储器向量Ry
```

解：可以把上述5条向量指令按如下方式进行编队：

- 第一编队：LV V1, Rx;
- 第二编队：MULTFV V2, F0, V1; LV V3, Ry;
- 第三编队：ADDV V4, V2, V3;
- 第四编队：SV Ry, V4。

假设：

$$T_{loop}=15$$

向量存储部件的启动：12个时钟周期

向量乘法部件的启动：7个时钟周期

向量加法部件的启动：6个时钟周期

向量寄存器长度：MVL

- 对 n 个向量元素进行计算所需的时钟周期数为

$$\begin{aligned} T_n &= \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + mn \\ &= \left\lceil \frac{n}{64} \right\rceil \times (15 + 12 + 12 + 6 + 12) + 4n = \left\lceil \frac{n}{64} \right\rceil \times 57 + 4n \end{aligned}$$

- 采用向量链接技术，那么指令序列可以编队为
- ❑ 第一编队：LV V1, Rx; MULTFV V2, F0, V1;
 - ❑ 第二编队：LV V3, Ry; ADDV V4, V2, V3;
 - ❑ 第三编队：SV Ry, V4。

- 第一编队启动需要12+7=19个时钟周期
- 第二个编队启动需要12+6=18个时钟周期
- 第三个编队启动仍然需要12个时钟周期

对 n 个向量元素进行计算所需的时钟周期数为

$$\begin{aligned} T_n &= \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + mn \\ &= \left\lceil \frac{n}{64} \right\rceil \times (15 + 19 + 18 + 12) + 3n = \left\lceil \frac{n}{64} \right\rceil \times 64 + 3n \end{aligned}$$

2. 向量处理机的峰值性能 R_{∞}

- R_{∞} 表示当向量长度为无穷大时，向量处理机的最高性能，也称为峰值性能。

$$R_{\infty} = \lim_{n \rightarrow \infty} \frac{\text{向量指令序列中浮点运算次数} \times \text{时钟频率}}{\text{向量指令序列执行所需的时钟周期数}}$$

- 对于上述例题3.6向量指令序列中的操作而言，只有“MULTFV V2, F0, V1”和“ADDV V4, V2, V3”两条浮点操作向量指令。
 - 假设该向量处理机的时钟频率为200 MHz，那么：

$$\begin{aligned} R &= \lim_{n \rightarrow \infty} \frac{\text{向量指令序列中浮点运算次数} \times \text{时钟频率}}{\text{向量指令序列执行所需的时钟周期数}} \quad \text{MFLOPS} \\ &= \lim_{n \rightarrow \infty} \frac{2 \times n \times 200}{\left\lceil \frac{n}{64} \right\rceil \times 64 + 3n} \quad \text{MFLOPS} \\ &= \lim_{n \rightarrow \infty} \frac{2 \times n \times 200}{4n} \quad \text{MFLOPS} \\ &= 100 \quad \text{MFLOPS} \end{aligned}$$

3. 半性能向量长度 $n_{1/2}$

- 半性能向量长度 $n_{1/2}$ 是指向量处理机的运行性能达到其峰值性能的一半时所必须满足的向量长度。
- 对于上面的例子

由于该向量处理机的峰值性能 $R_{\infty}=100$ MFLOPS,
所以根据半性能向量长度的定义有:

$$\frac{2 \times n_{1/2} \times 200}{\left\lceil \frac{n_{1/2}}{64} \right\rceil \times 64 + 3n_{1/2}} = 50$$

假设 $n_{1/2} \leq 64$ ，那么有：

$$64 + 3n_{1/2} = \frac{2 \times n_{1/2} \times 200}{50} = 8n_{1/2}$$

$$5n_{1/2} = 64, \quad n_{1/2} = 12.8$$

$$n_{1/2} = 13$$

4. 向量长度临界值 n_v

- 向量长度临界值 n_v 是指：对于某一计算任务而言，向量方式的处理速度优于标量串行方式处理速度时所需的最小向量长度。
- 对于上述DAXPY的例子
 - ▣ 假设，在标量串行工作方式下实现DAXPY循环的开销为10个时钟周期。那么在标量串行方式下，计算DAXPY循环所需要的时钟周期数为：

$$T_s = (10 + 12 + 12 + 7 + 6 + 12) \times n_v = 59n_v$$

- 在向量方式下，计算DAXPY循环所需要的时钟周期数为：

$$T_v = 64 + 3n_v$$

- 根据向量长度临界值的定义，有：

$$T_v = T_s$$

$$64 + 3n_v = 59n_v$$

$$n_v = \left\lceil \frac{64}{56} \right\rceil = 2$$