

第3章 流水线技术

1. 什么是流水线？
2. 如何评价流水线？
3. 怎样才能使流水线的效率最高？
4. 如何处理流水线的资源争用？

3.1 流水线的基本概念

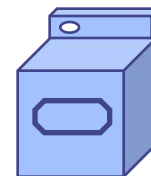
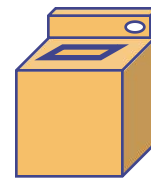
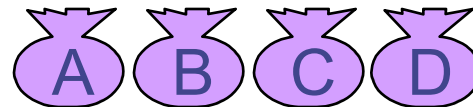
洗衣为例

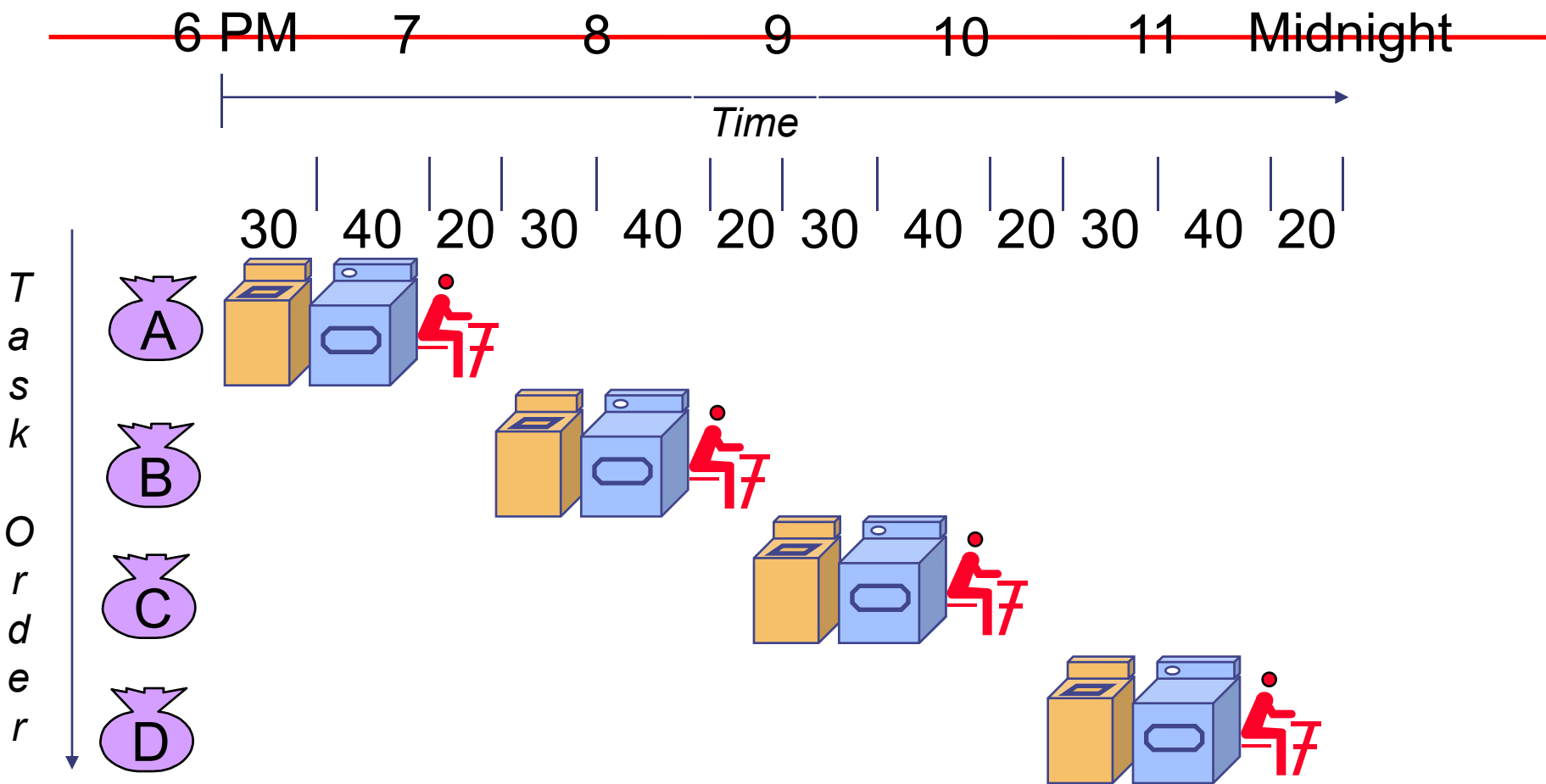
Ann, Brian, Cathy, Dave
每人进行洗衣的动作
wash, dry, and fold

Washer需要 30 minutes

Dryer 需要 40 minutes

Folder 需要 20 minutes

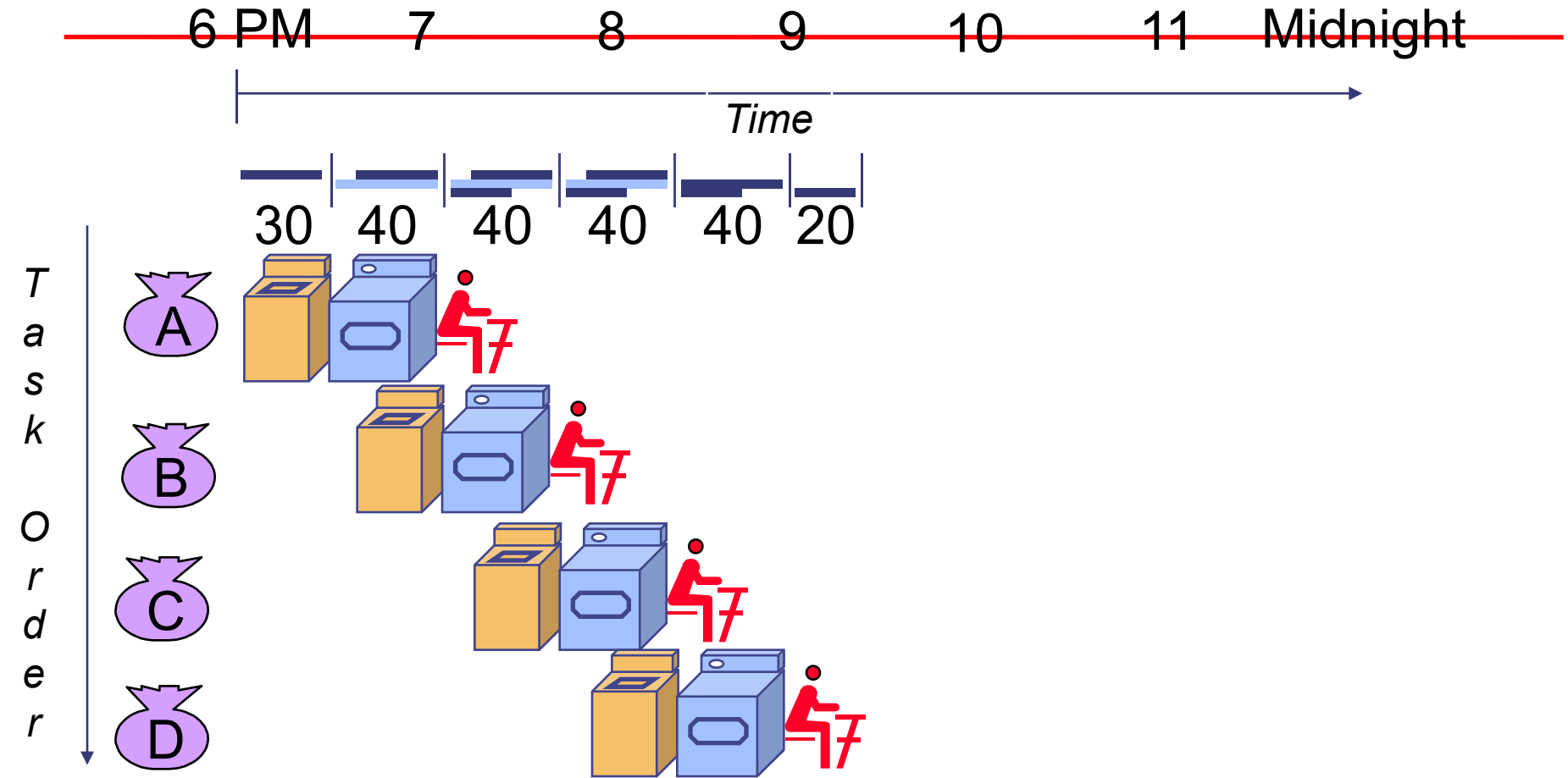




1. 顺序完成这些任务需要 6 hours for 4 loads

2. 如果采用流水作业, 需要多长时间?

流水线作业原则:尽可能早的开始工作



流水作业完成四人的洗衣任务只需要 **3.5 hours**

2. 流水线技术

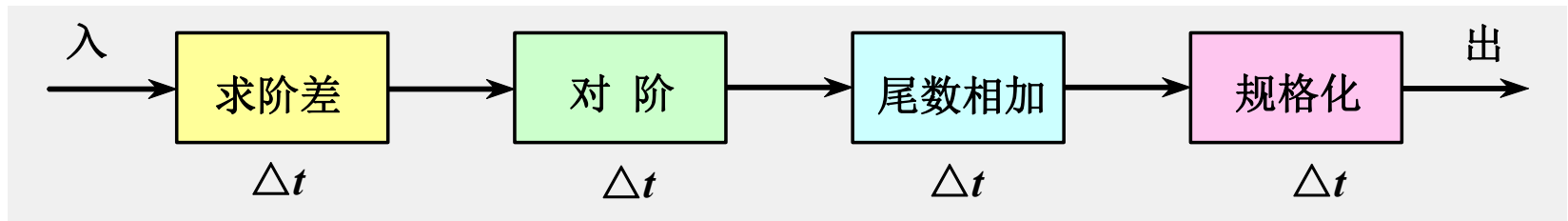
- 把一个重复的过程分解为若干个子过程，每个子过程由专门的功能部件来实现。
- 把多个处理过程在时间上错开，依次通过各功能段，这样，每个子过程就可以与其它子过程并行进行。

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. 浮点加法流水线

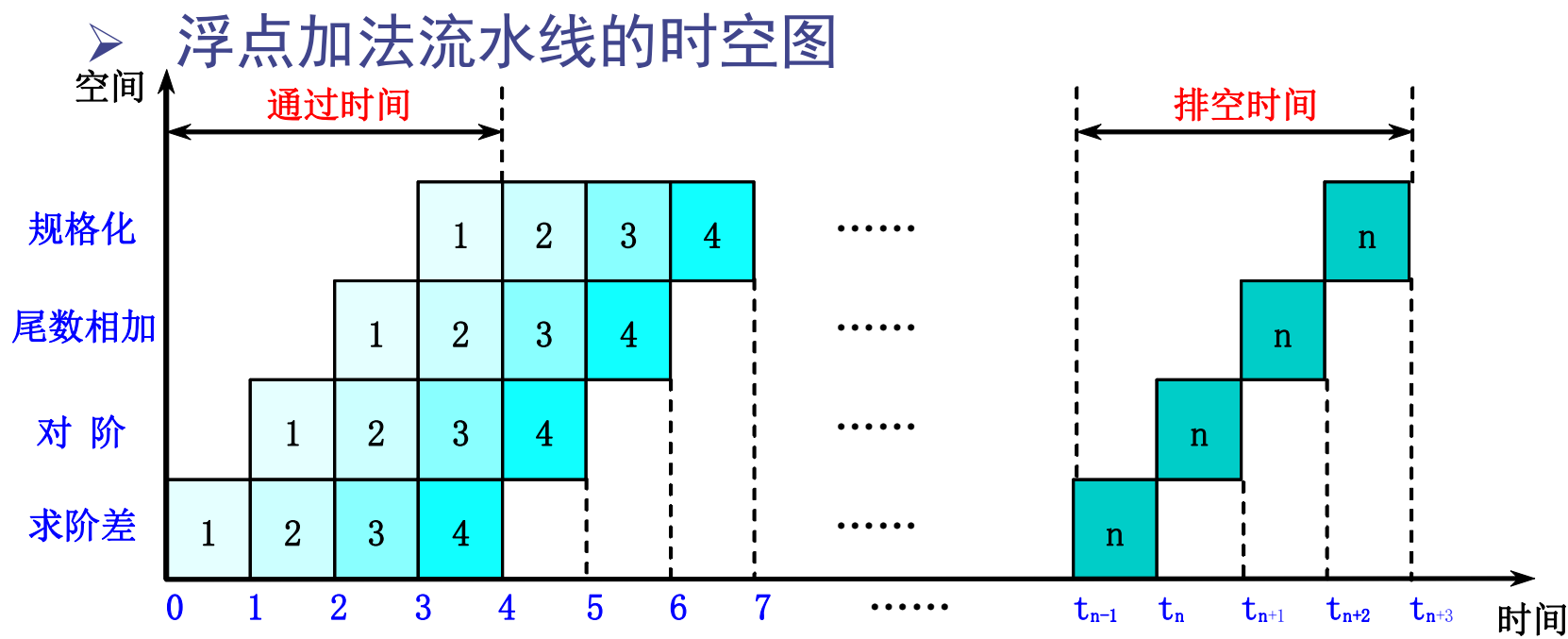
- 把流水线技术应用于运算的执行过程，就形成了运算操作流水线，也称为部件级流水线。
- 把浮点加法的全过程分解为求阶差、对阶、尾数相加、规格化四个子过程。

理想情况：速度提高3倍



5. 时一空图

- 时一空图从时间和空间两个方面描述了流水线的工作过程。时一空图中，横坐标代表时间，纵坐标代表流水线的各个段。



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。

时间最长的段将成为流水线的瓶颈。

- 流水线每一个段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，以提供后面要用到的信息，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。

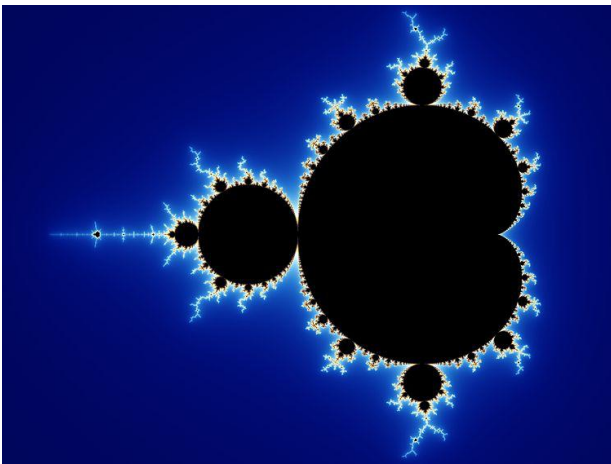
3.1.2 流水线的分类

从不同的角度和观点，把流水线分成多种不同的种类。

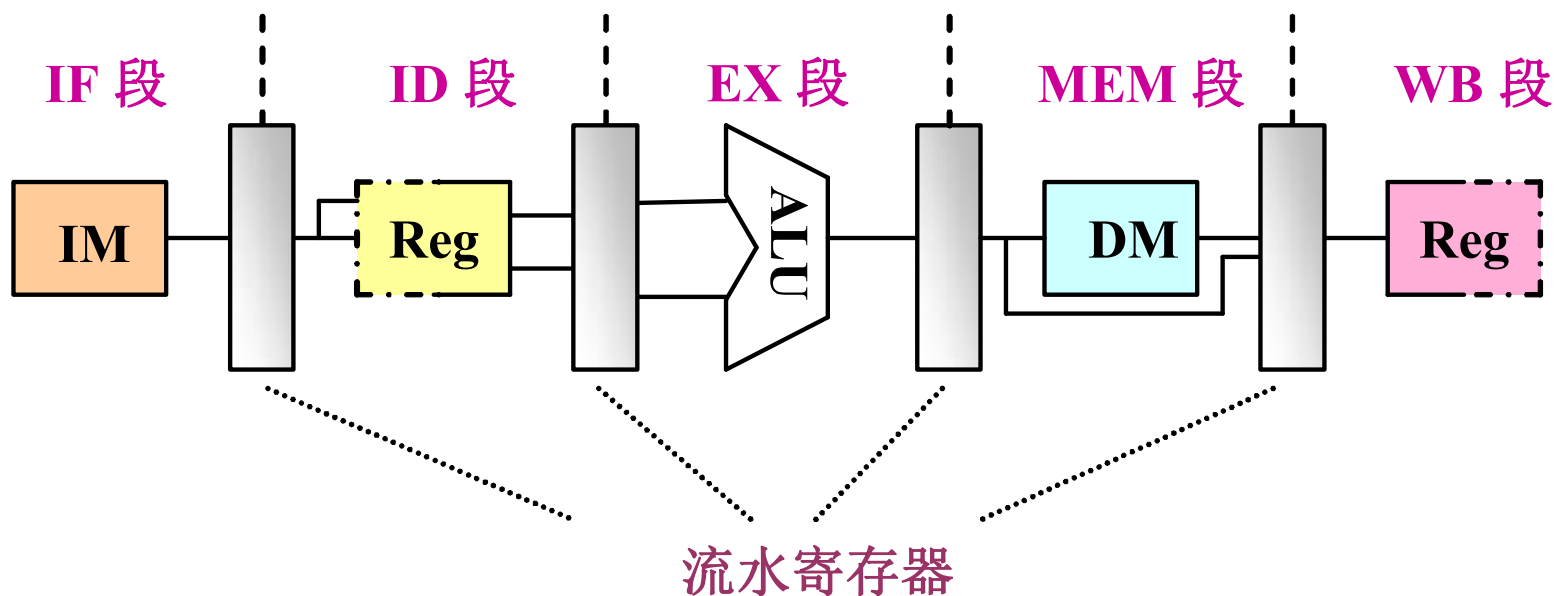
1. 部件级、处理机级及处理机间流水线

（按照流水技术用于计算机系统的等级不同）

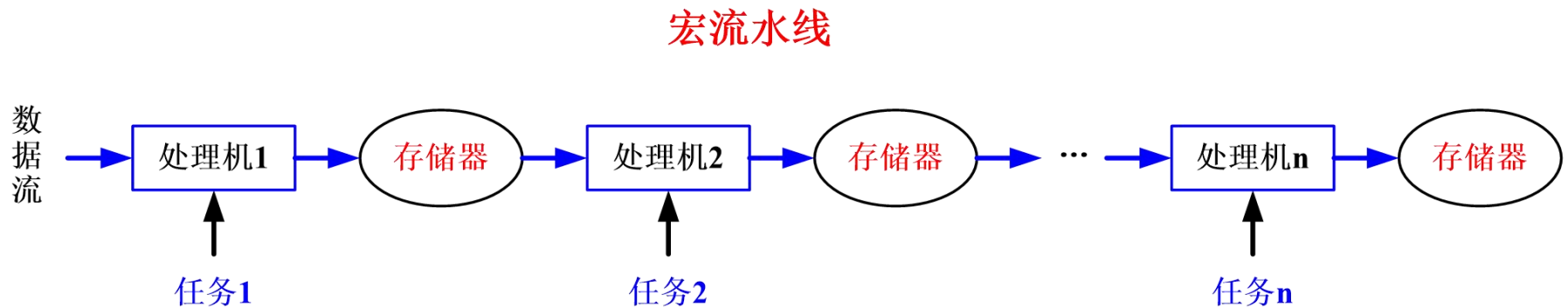
- **部件级流水线**（运算操作流水线）：把处理机中的部件分段，再把这些分段相互连接起来，使得各种类型的运算操作能够按流水方式进行。



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。



- **系统级流水线（宏流水线）**：把多台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。

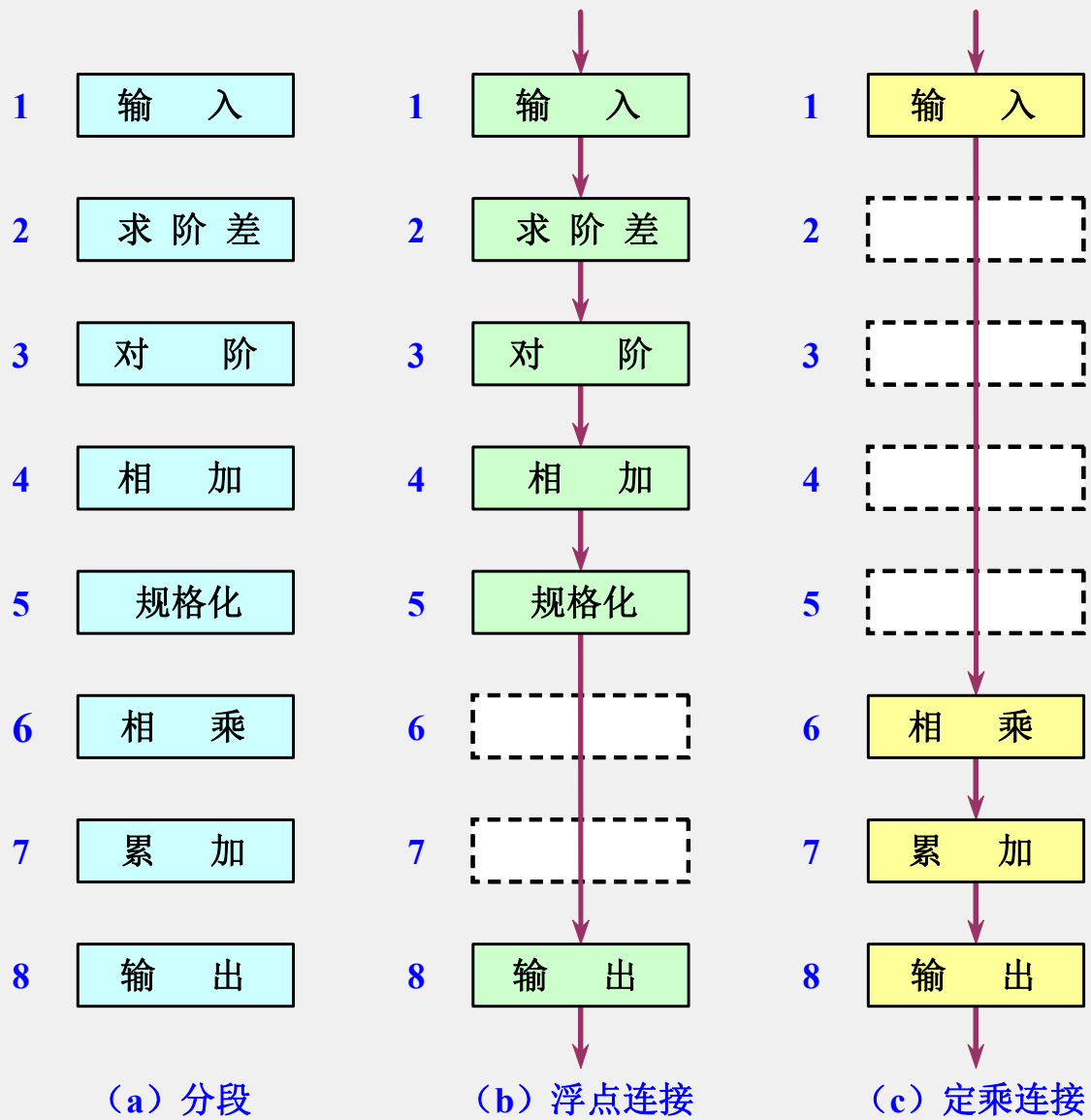


2. 单功能流水线与多功能流水线

（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（图**3.3**）



3. 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作进一步的分类）

➤ **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。

□ **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。

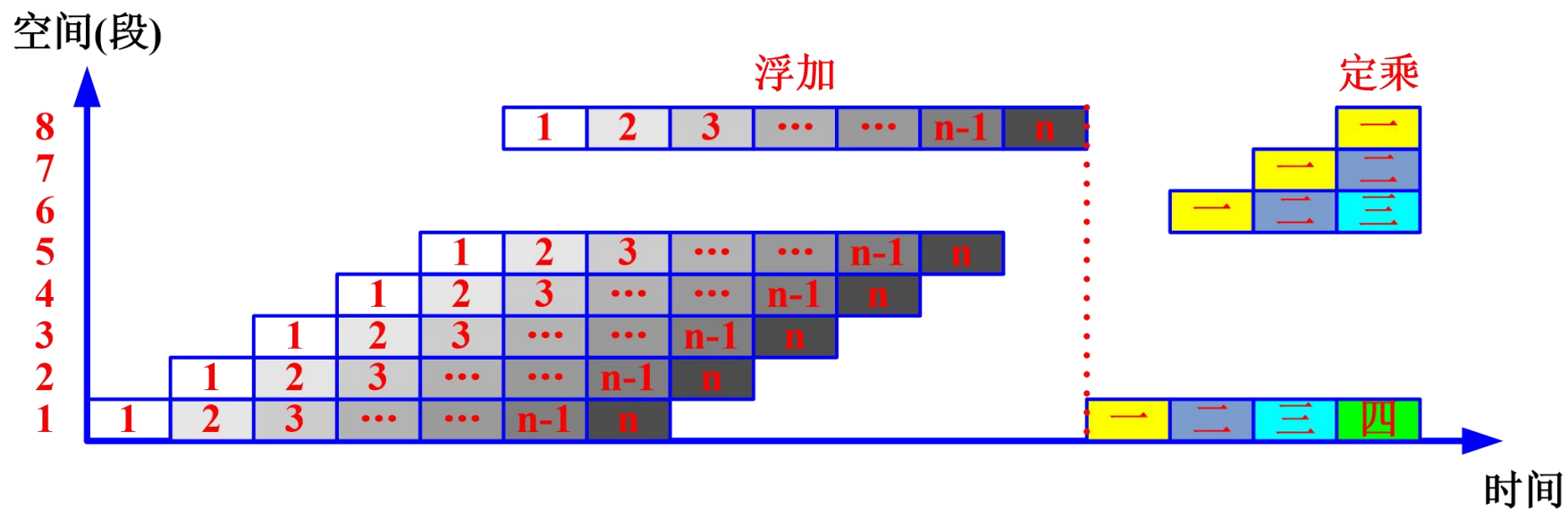
□ **缺点**

控制复杂。

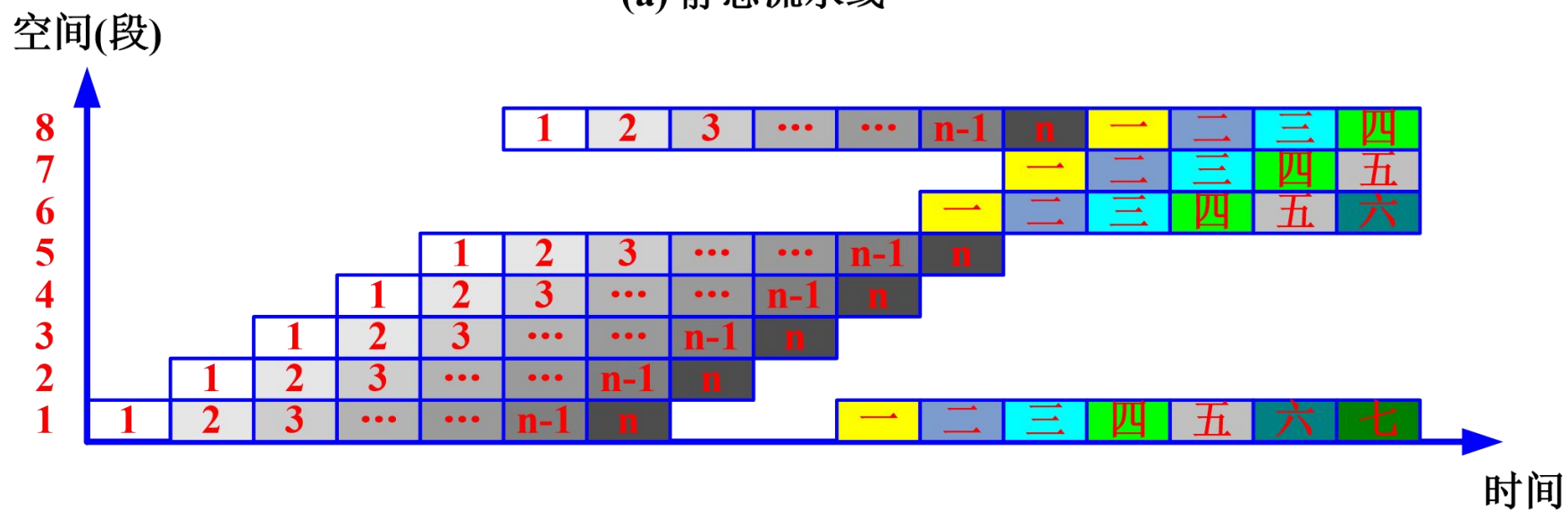
➤ 静、动态流水线时空图的对比

静、动态流水线的时空图

假设该流水线要先做几个浮点加法，然后再做一批定点乘法。



(a) 静态流水线



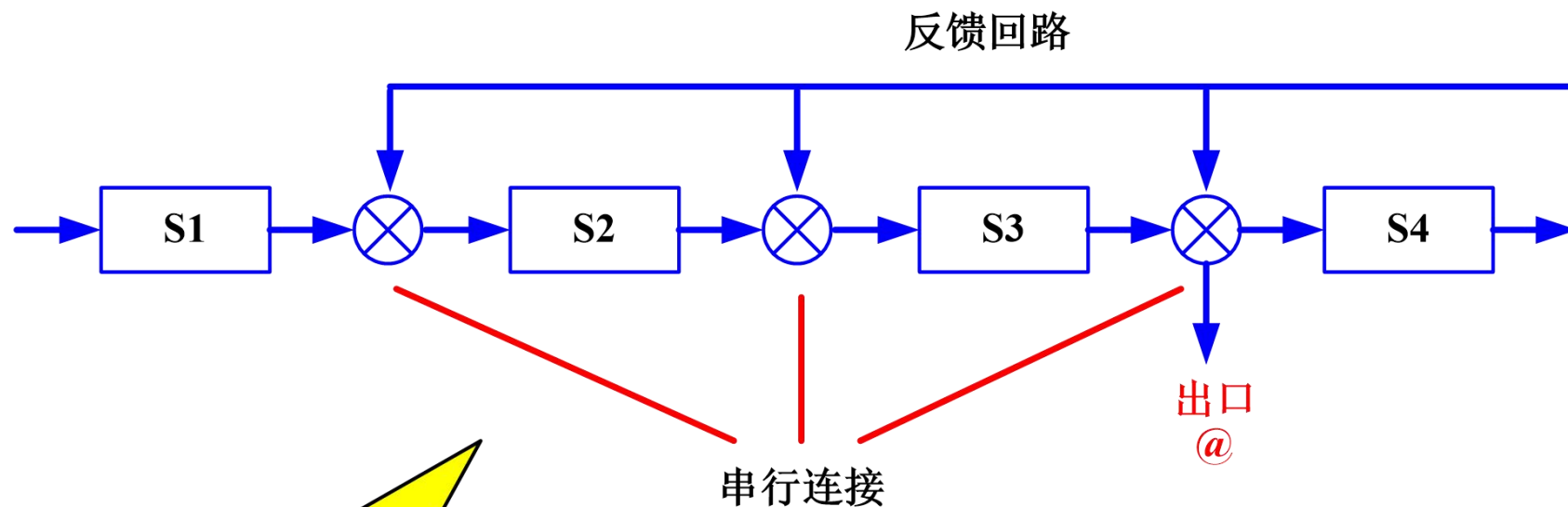
(b) 动态流水线

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。
- 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）



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

3.2 流水线的性能指标

3.2.1 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。

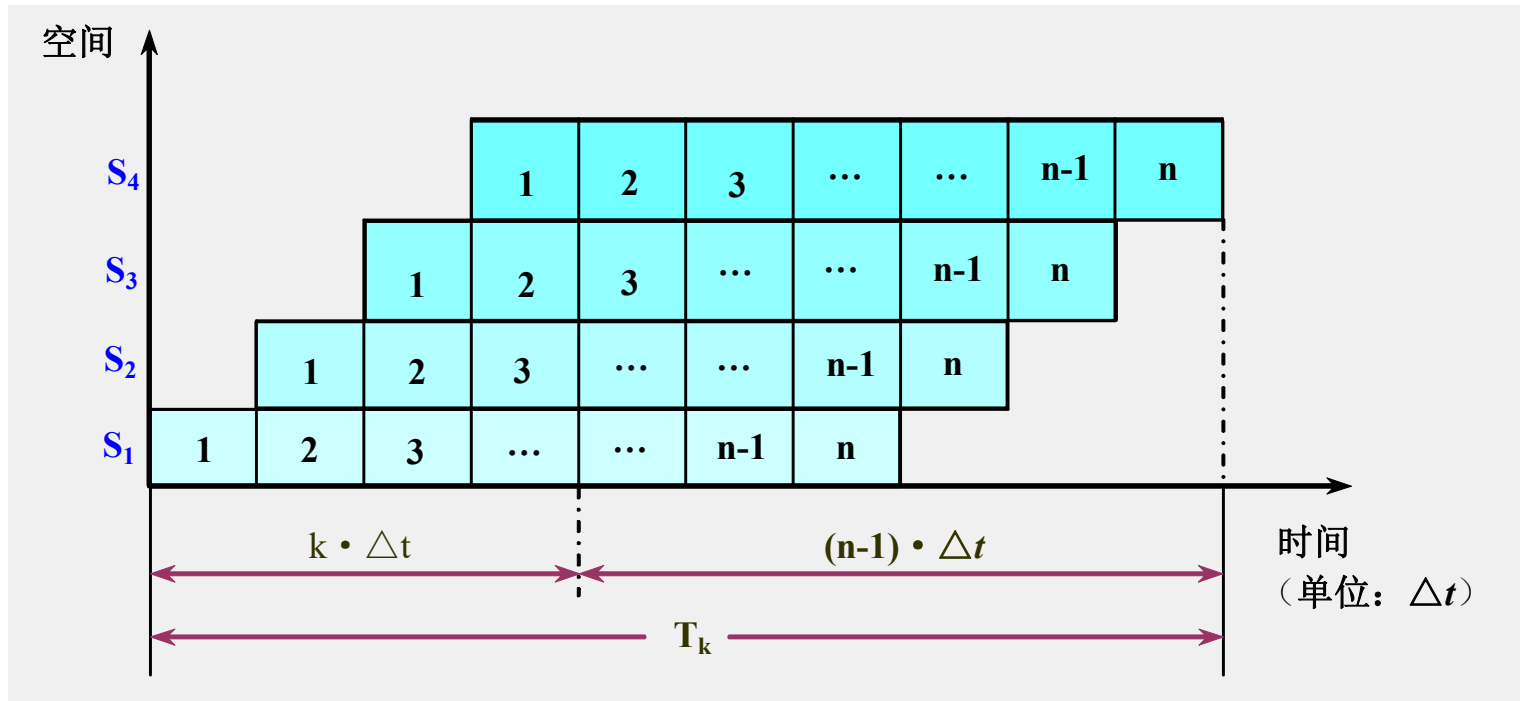
$$TP = \frac{n}{T_k}$$

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

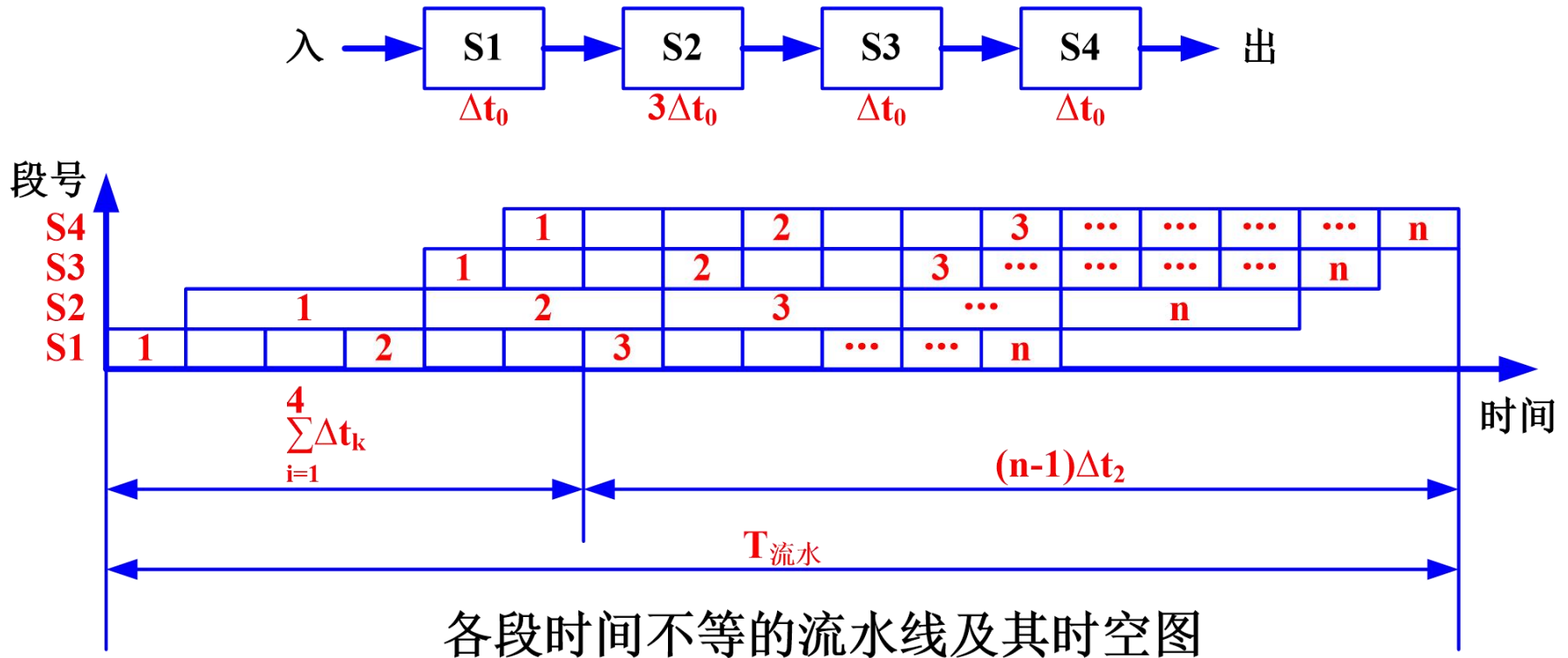
➤ 各段时间不等的流水线及其时空图

举例1 (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**。

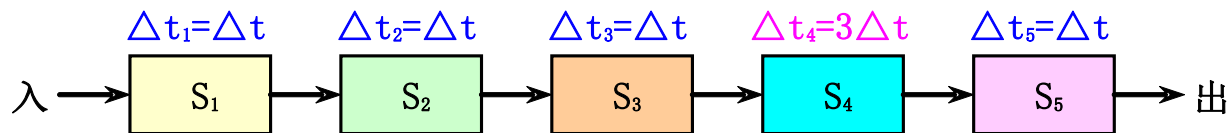
3.2 流水线的性能指标



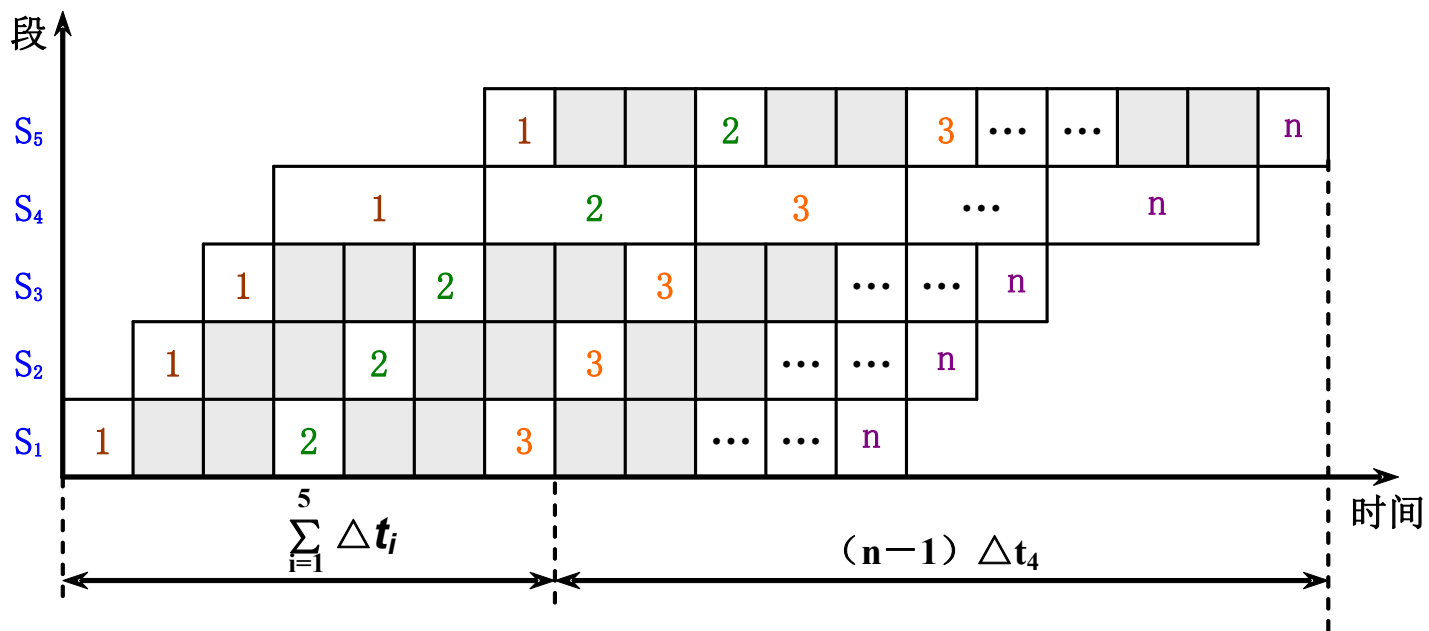
$$T P_{\text{max}} = \frac{1}{\text{max} \{ \Delta t_i \}}$$

举例2：一条5段的流水线

- S_1, S_2, S_3, S_5 各段的时间: Δt
- S_4 的时间: $3\Delta t$ (瓶颈段)



(a) 流水线



(b) 时空图

➤ 各段时间不等的流水线的实际吞吐率为：

（ 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：

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

对前面举例2中的5段流水线

最大吞吐率为：

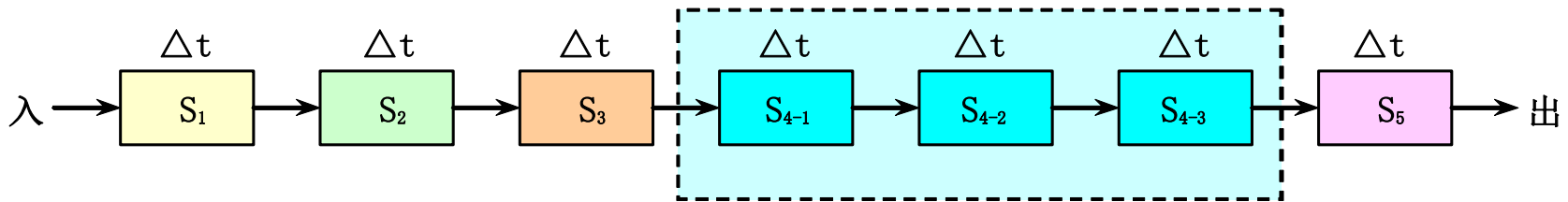
$$TP_{\max} = \frac{1}{3\Delta t}$$

3. 解决流水线瓶颈问题的常用方法

➤ 细分瓶颈段

例如：对前面的5段流水线

把瓶颈段 S_4 细分为3个子流水线段： S_{4-1} ， S_{4-2} ， S_{4-3}



改进后的流水线的吞吐率：

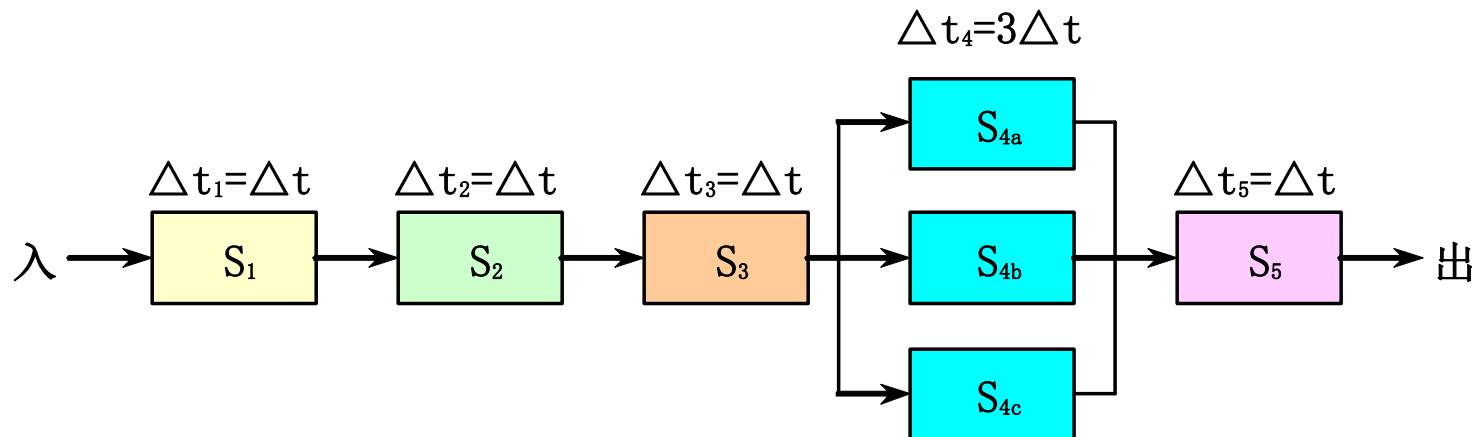
$$TP_{\max} = \frac{1}{\Delta t}$$

➤ 重复设置瓶颈段

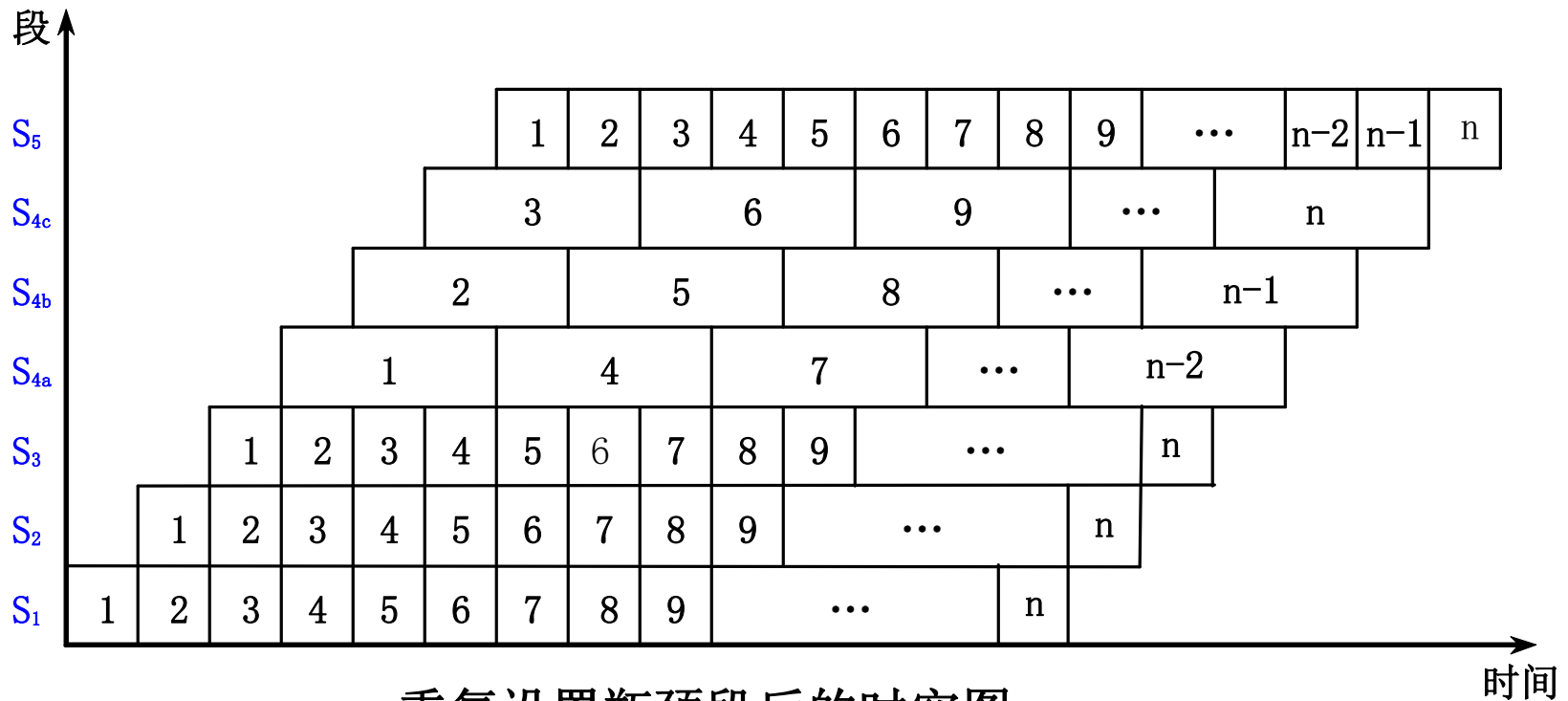
- 举例：[时-空图](#)
- 缺点：控制逻辑比较复杂，所需的硬件增加了。

例如：对前面的5段流水线

重复设置瓶颈段 S_4 ： S_{4a} ， S_{4b} ， S_{4c}



3.2 流水线的性能指标



重复设置瓶颈段后的时空图

改进后的流水线的吞吐率：

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

3.2.2 流水线的加速比

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：

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

3.2.3 流水线的效率

流水线的效率：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

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

- 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。

$$E = TP \cdot \Delta t$$

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：

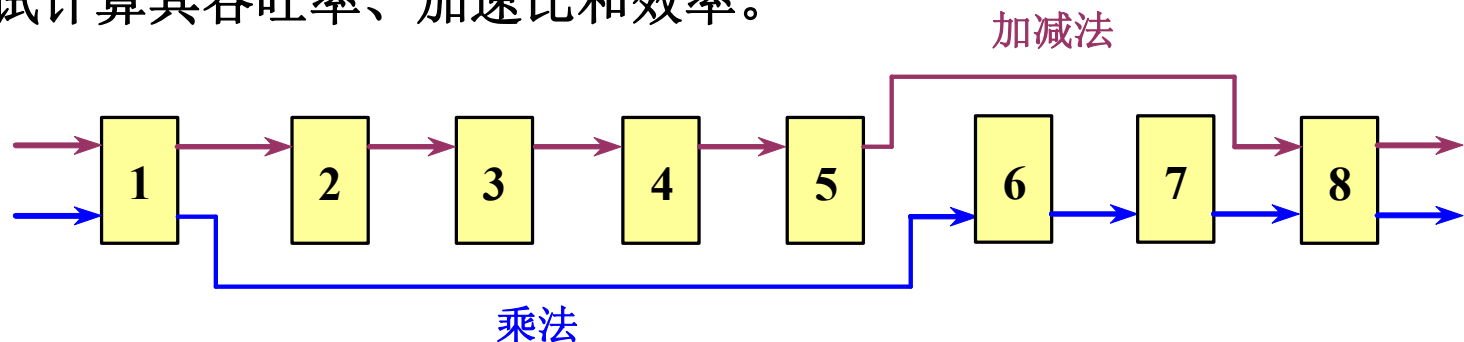
$$E = \frac{n \cdot \sum_{i=1}^k \Delta t_i}{k \left[\sum_{i=1}^k \Delta t_i + (n-1) \cdot \max(\Delta t_1, \Delta t_2, \dots, \Delta t_k) \right]}$$

3.2.4 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。

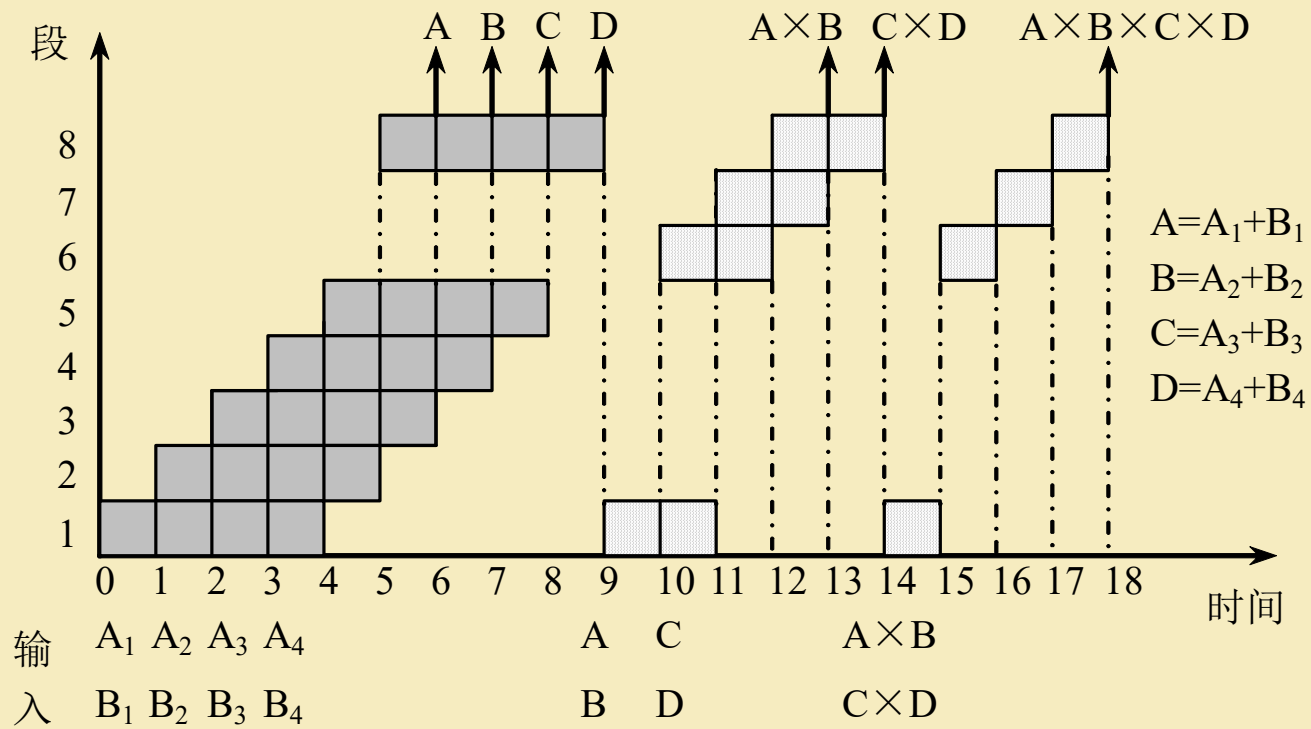


(每段的时间都为 Δt)

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



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

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

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

(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。

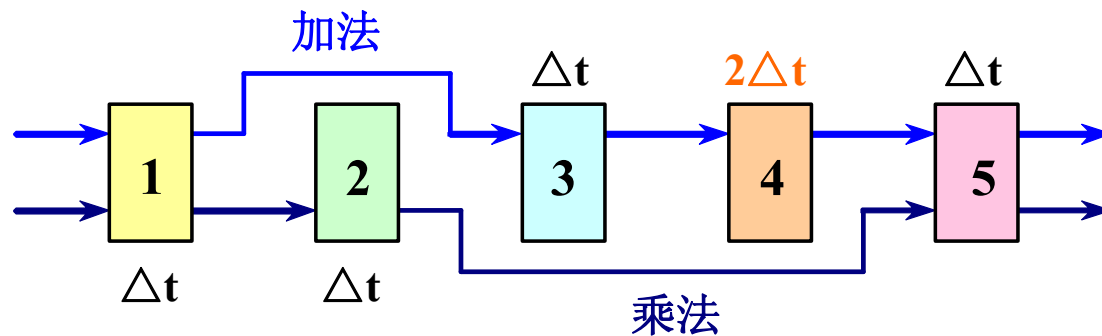
➤ 主要原因

- ❑ 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4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。



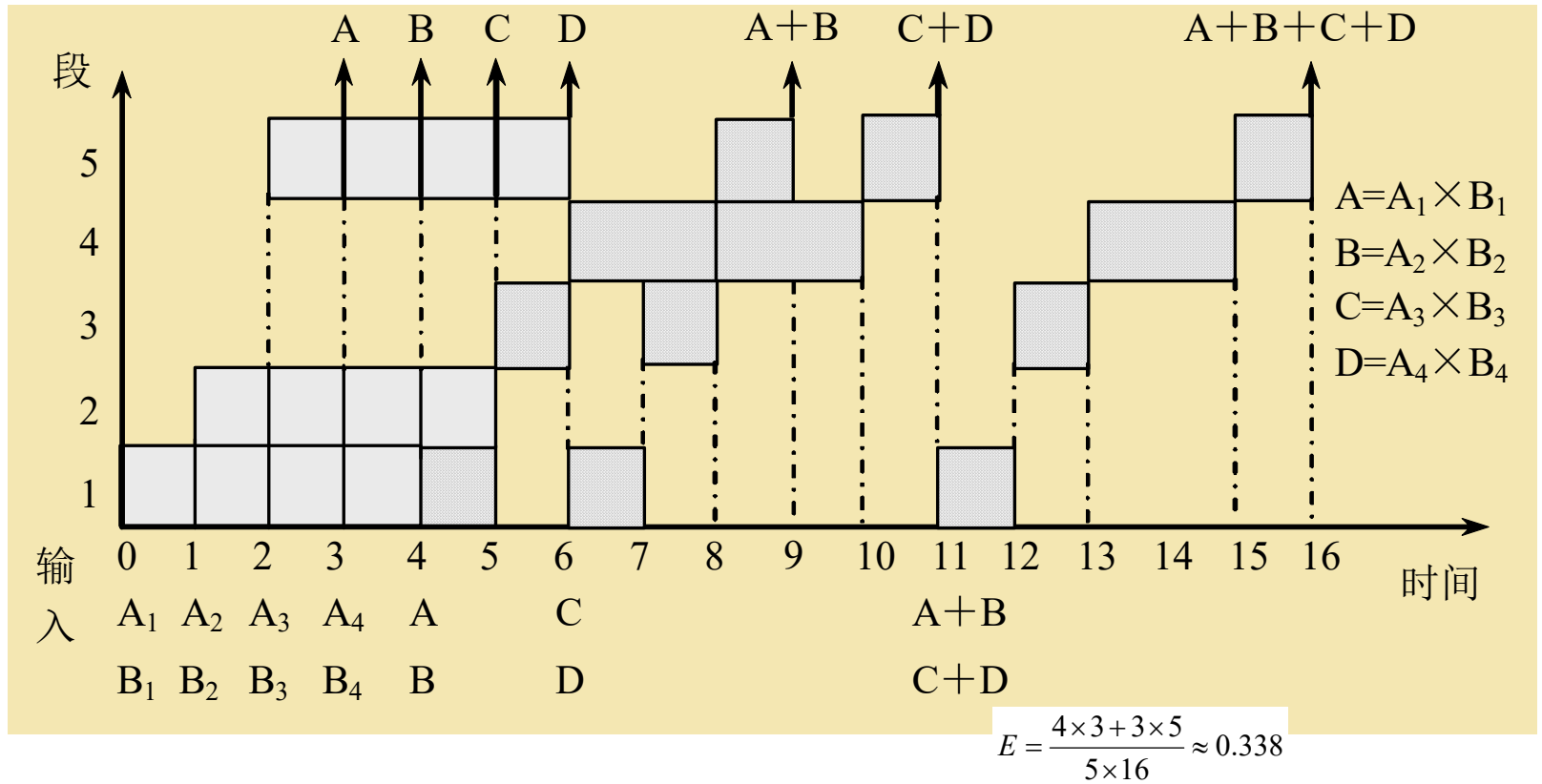
解：(1) 选择适合于流水线工作的算法

- 应先计算 $A_1 \times B_1$ 、 $A_2 \times B_2$ 、 $A_3 \times B_3$ 和 $A_4 \times B_4$ ；
- 再计算 $(A_1 \times B_1) + (A_2 \times B_2)$
 $(A_3 \times B_3) + (A_4 \times B_4)$ ；
- 然后求总的累加结果。

(2) 画出时空图

(3) 计算性能

3.2 流水线的性能指标



$$TP = \frac{7}{16\Delta t}$$

$$S = \frac{27\Delta t}{16\Delta t} \approx 1.69$$

下面我们再看一个例子：

例 在静态流水线上计算：

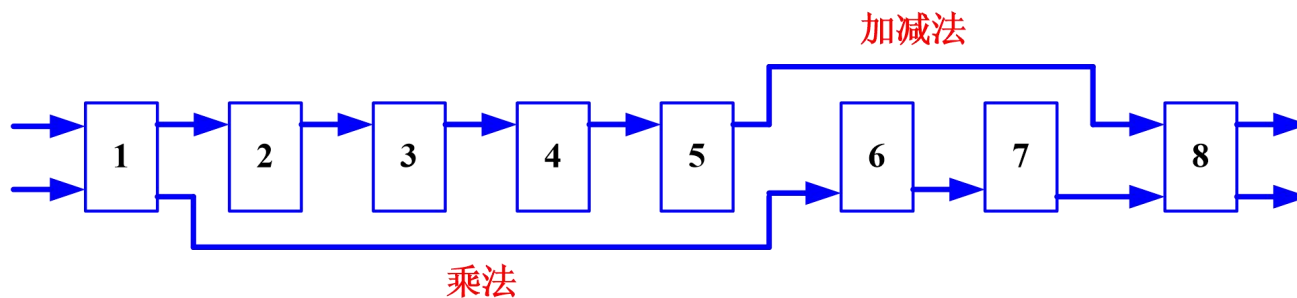
$$\sum_{i=1}^4 (A_i \times B_i)$$

$$\sum_{i=1}^4 A_i B_i = A_1 B_1 + A_2 B_2 + A_3 B_3 + A_4 B_4$$

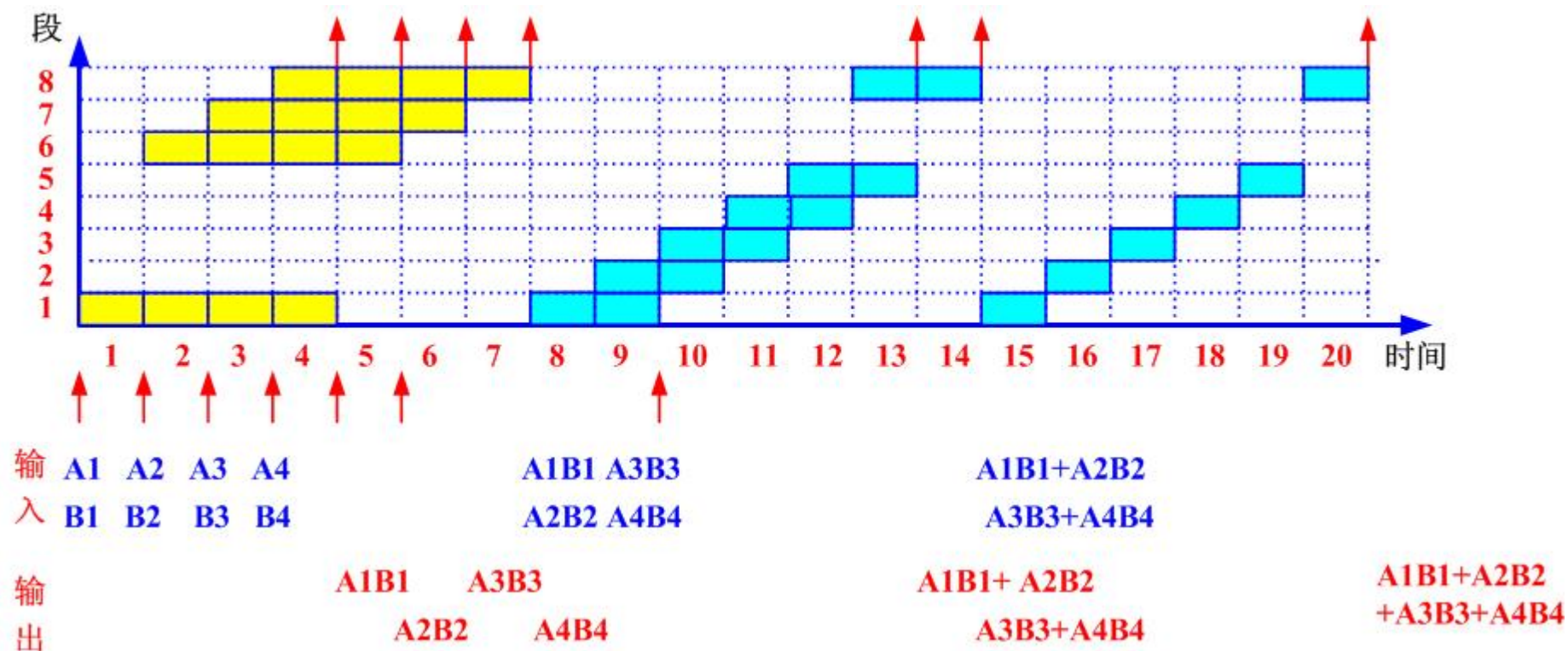
求：吞吐率，加速比，效率。

解： (1) 确定适合于流水处理的计算过程

(2) 画时空图



静态流水线时空图



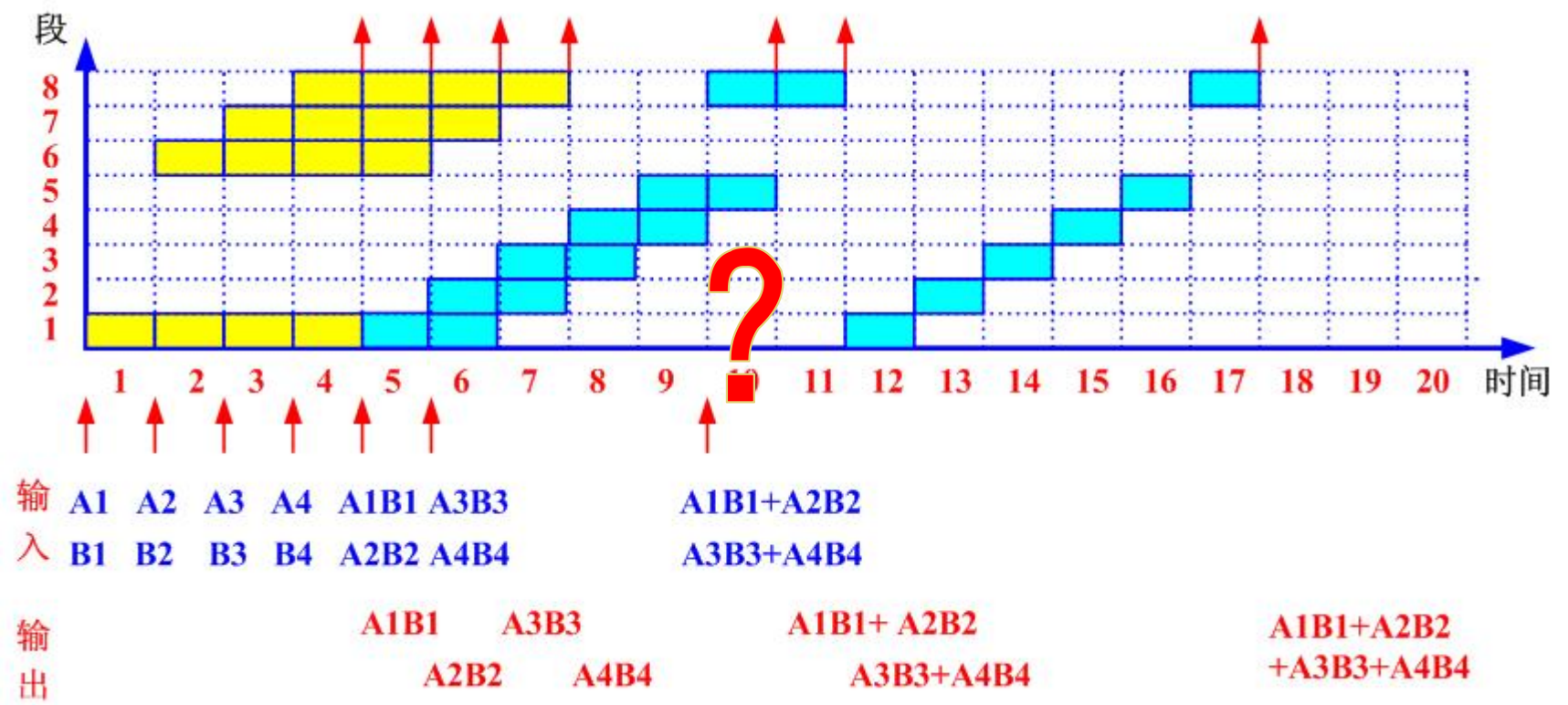
(3) 计算性能

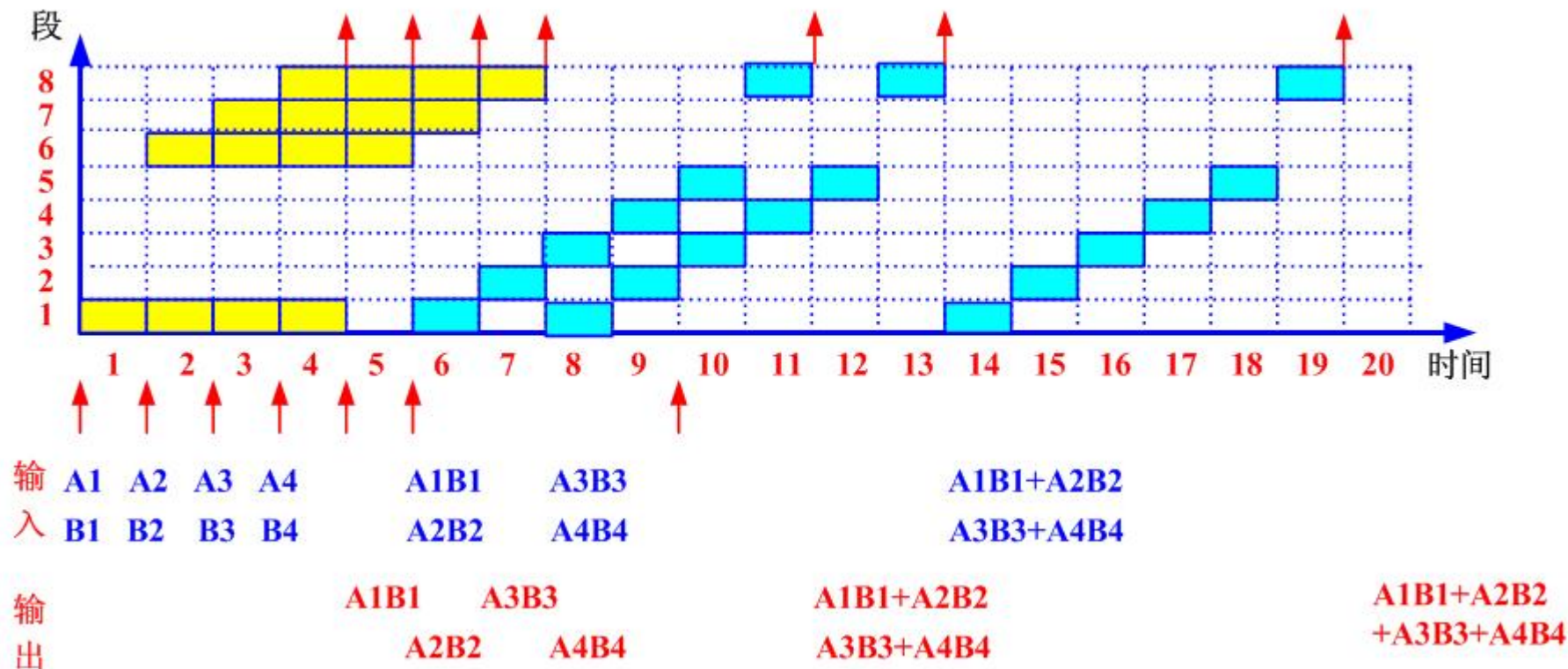
吞吐率 $TP = 7 / (20 \Delta t)$

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

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

动态流水线时空图-1





动态流水线时空图-2

$$T_p = 7 / (19 \Delta t)$$

$$S = 34 / 19$$

$$E = (4 * 4 + 3 * 6) / (8 * 19)$$

3.2.5 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。

3. 冲突问题

流水线设计中要解决的重要问题之一。

3.3 非线性流水线的调度

- 在非线性流水线中，存在反馈回路，当一个任务在流水线中流过时，可能要多次经过某些段。
- 流水线调度要解决的问题：

应按什么样的时间间隔向流水线输入新任务，才能既不发生功能段使用冲突，又能使流水线有较高的吞吐率和效率？

3.3.1 单功能非线性流水线的最优调度

- 向一条非线性流水线的输入端连续输入两个任务之间的时间间隔称为非线性流水线的**启动距离**。
- 会引起非线性流水线功能段使用冲突的启动距离则称为**禁用启动距离**。
- 启动距离和禁用启动距离一般都用时钟周期数来表示，是一个正整数。
- **预约表**
 - 横向（向右）：时间（一般用时钟周期表示）
 - 纵向（向下）：流水线的段

例：一个5功能段非线性流水线预约表

功能段 \ 时间	1	2	3	4	5	6	7	8	9
S1	√								√
S2		√	√					√	
S3				√					
S4					√	√			
S5							√	√	

- 如果在第n个时钟周期使用第k段，则在第k行和第n列的交叉处的格子里有一个√。
- 如果在第k行和第n列的交叉处的格子里有一个√，则表示在第n个时钟周期要使用第k段。

1. 根据预约表写出禁止表F

➤ **禁止表F**：一个由禁用启动距离构成的集合。

➤ **具体方法**

对于预约表的每一行的任何一对√，用它们所在的列号相减（大的减小的），列出各种可能的差值，然后删除相同的，剩下的就是禁止表的元素。

➤ **在上例中**

- 第一行的差值只有一个：8；
- 第二行的差值有3个：1, 5, 6；
- 第3行只有一个√，没有差值；
- 第4和第5行的差值都只有一个：1；

其禁止表是： $F = \{ 1, 5, 6, 8 \}$

2. 根据禁止表F写出初始冲突向量 C_0

(从一个集合到一个二进制位串的变换)

➤ 冲突向量C：一个N位的二进制位串。

➤ 设 $C_0 = (c_N c_{N-1} \dots c_i \dots c_2 c_1)$ ，则：

$$c_i = \begin{cases} 1 & i \in F \\ 0 & i \notin F \end{cases}$$

□ $c_i = 0$ ：允许间隔i个时钟周期后送入后续任务

□ $c_i = 1$ ：不允许间隔i个时钟周期后送入后续任务

➤ 对于上面的例子

$$F = \{ 1, 5, 6, 8 \}$$

$$C_0 = (10110001)$$

3. 根据初始冲突向量 C_0 画出状态转换图

- 当第一个任务流入流水线后，初始冲突向量 C_0 决定了下一个任务需间隔多少个时钟周期才可以流入。
- 在第二个任务流入后，新的冲突向量是怎样的呢？
 - 假设第二个任务是在与第一个任务间隔 j 个时钟周期流入，这时，由于第一个任务已经在流水线中前进了 j 个时钟周期，其相应的禁止表中各元素的值都应该减去 j ，并丢弃小于等于0的值。
 - 对冲突向量来说，就是逻辑右移 j 位（左边补0）。
 - 在冲突向量上，就是对它们的冲突向量进行“或”运算。

$$\text{SHR}^j(C_0) \vee C_0$$

其中： SHR^j 表示逻辑右移 j 位

➤ 推广到更一般的情况

假设: C_k : 当前的冲突向量

j : 允许的时间间隔

则新的冲突向量为:

$$SHR^{(j)}(C_k) \vee C_0$$

- 对于所有允许的时间间隔都按上述步骤求出其新的冲突向量, 并且把新的冲突向量作为当前冲突向量, 反复使用上述步骤, 直到不再产生新的冲突向量为止。

- 从初始冲突向量 C_0 出发，反复应用上述步骤，可以求得所有的冲突向量以及产生这些向量所对应的时间间隔。由此可以画出用冲突向量表示的流水线状态转移图。
- 有向弧：表示状态转移的方向
- 弧上的数字：表示引入后续任务（从而产生新的冲突向量）所用的时间间隔（时钟周期数）

对于上面的例子

(1) $C_0 = (10110001)$

引入后续任务可用的时间间隔为：2、3、4、7个时钟周期

如果采用2，则新的冲突向量为：

$$(00101100) \vee (10110001) = (10111101)$$

如果采用3，则新的冲突向量为：

$$(00010110) \vee (10110001) = (10110111)$$

如果采用4，则新的冲突向量为：

$$(00001011) \vee (10110001) = (10111011)$$

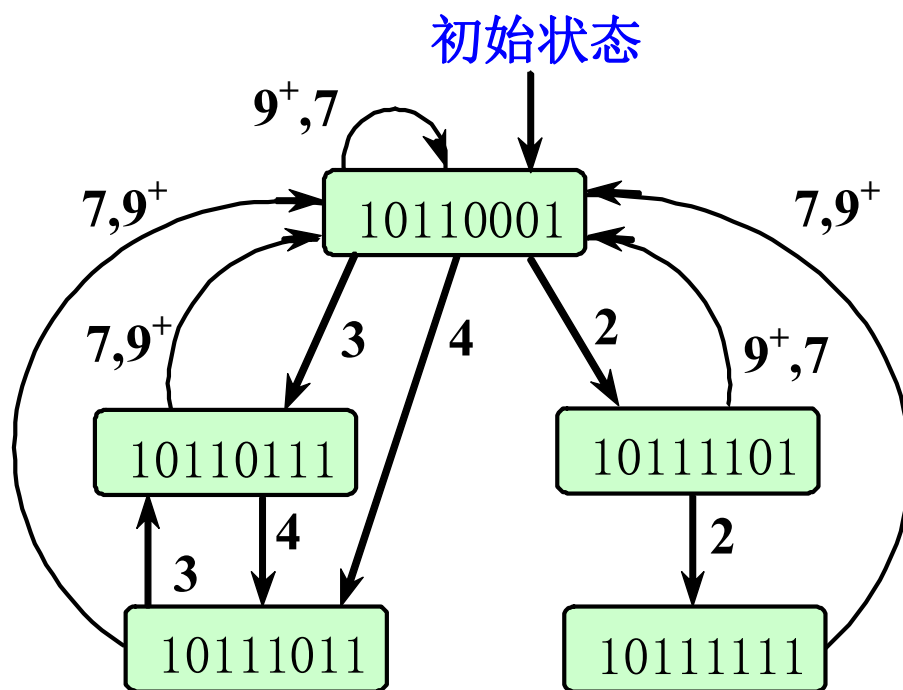
如果采用7，则新的冲突向量为：

$$(00000001) \vee (10110001) = (10110001)$$

(2) 对于新向量 (10111101)，其可用的时间间隔为2个和7个时钟周期。用类似上面的方法，可以求出其后续的冲突向量分别为 (10111101) 和 (10110001)。

(3) 对于其他新向量，也照此处理。

(4) 在此基础上，画出状态转移示意图。



4. 根据状态转换图写出最优调度方案

- 根据流水线状态图，由初始状态出发，任何一个闭合回路即为一种调度方案。
- 列出所有可能的调度方案，计算出每种方案的平均时间间隔，从中找出其最小者即为最优调度方案。
- 上例中，各种调度方案及其平均间隔时间。
 - 最佳方案：（3，4）
平均间隔时间：3.5个时钟周期（吞吐率最高）

各种调度策略及平均延迟拍数

调度策略	平均延迟拍数
(2,7)	4.5
(2,2,7)	3.67
(3,7)	5
(3,4)	3.5
(3,4,3,7)	4.25
(3,4,7)	4.67
(4,3,7)	4.67
(4,7)	5.5
(7)	7

关于算法

3.4 流水线的相关与冲突

3.4.1 一条经典的5段流水线

- 介绍一条经典的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）
 - 以程序计数器PC中的内容作为地址，从存储器中取出指令并放入指令寄存器IR；
 - 同时PC值加4（假设每条指令占4个字节），指向顺序的下一条指令。

➤ 指令译码/读寄存器周期（ID）

对指令进行译码，并用IR中的寄存器地址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：

- **load和store指令**：ALU把指令中所指定的寄存器的内容与偏移量相加，形成访存有效地址。
- **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出的数据进行运算。

- **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出的操作数和指令中给出的立即数进行运算。
- **分支指令**：ALU把指令中给出的偏移量与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。
 - 其它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。

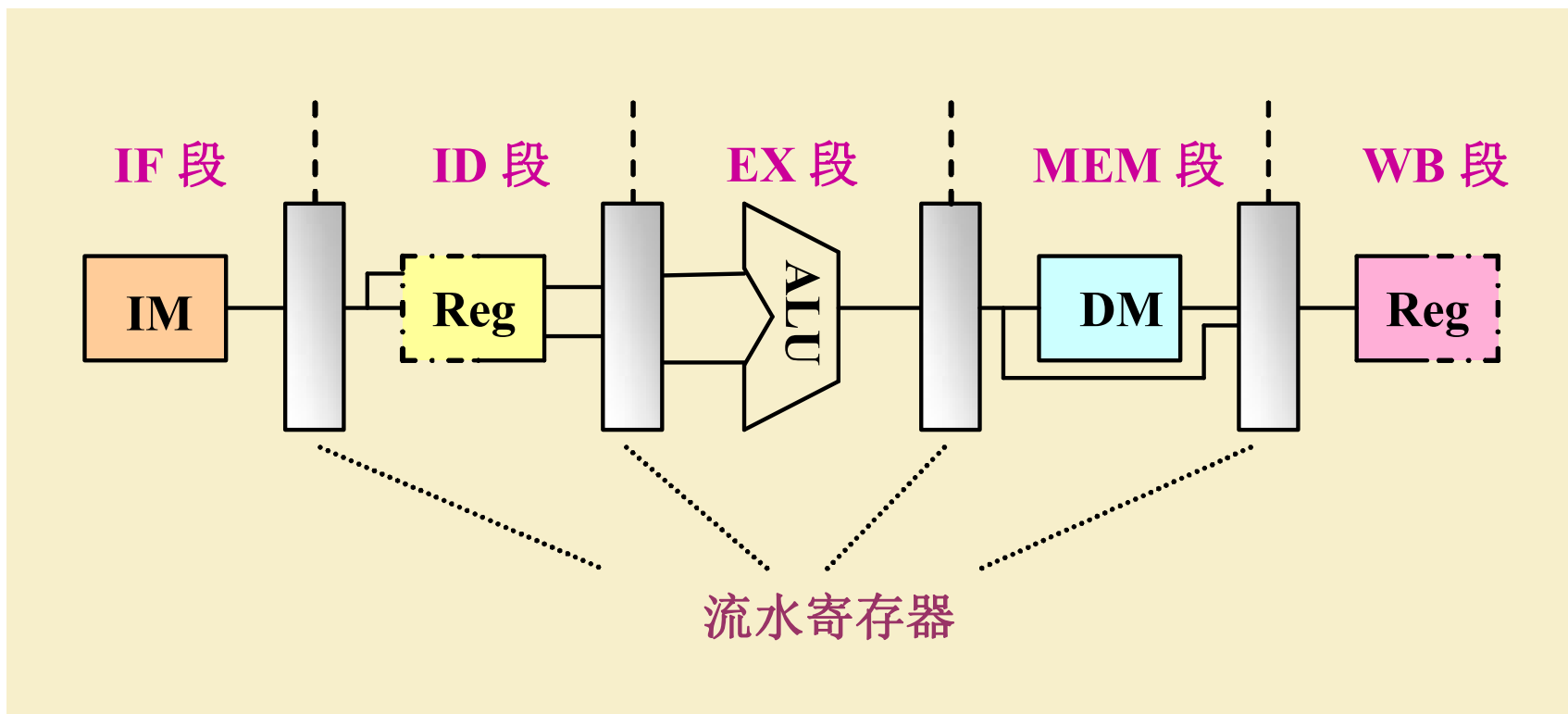
在这个实现方案中：

- 分支指令需要4个时钟周期；
- store指令需要4个周期；
- 其它指令需要5个周期才能完成。

2. 将上述实现方案修改为流水线实现

➤ 一条经典的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）。



3. 采用流水线方式实现时，应解决好以下几个问题：

- 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。（为什么这样安排？）

➤ 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？

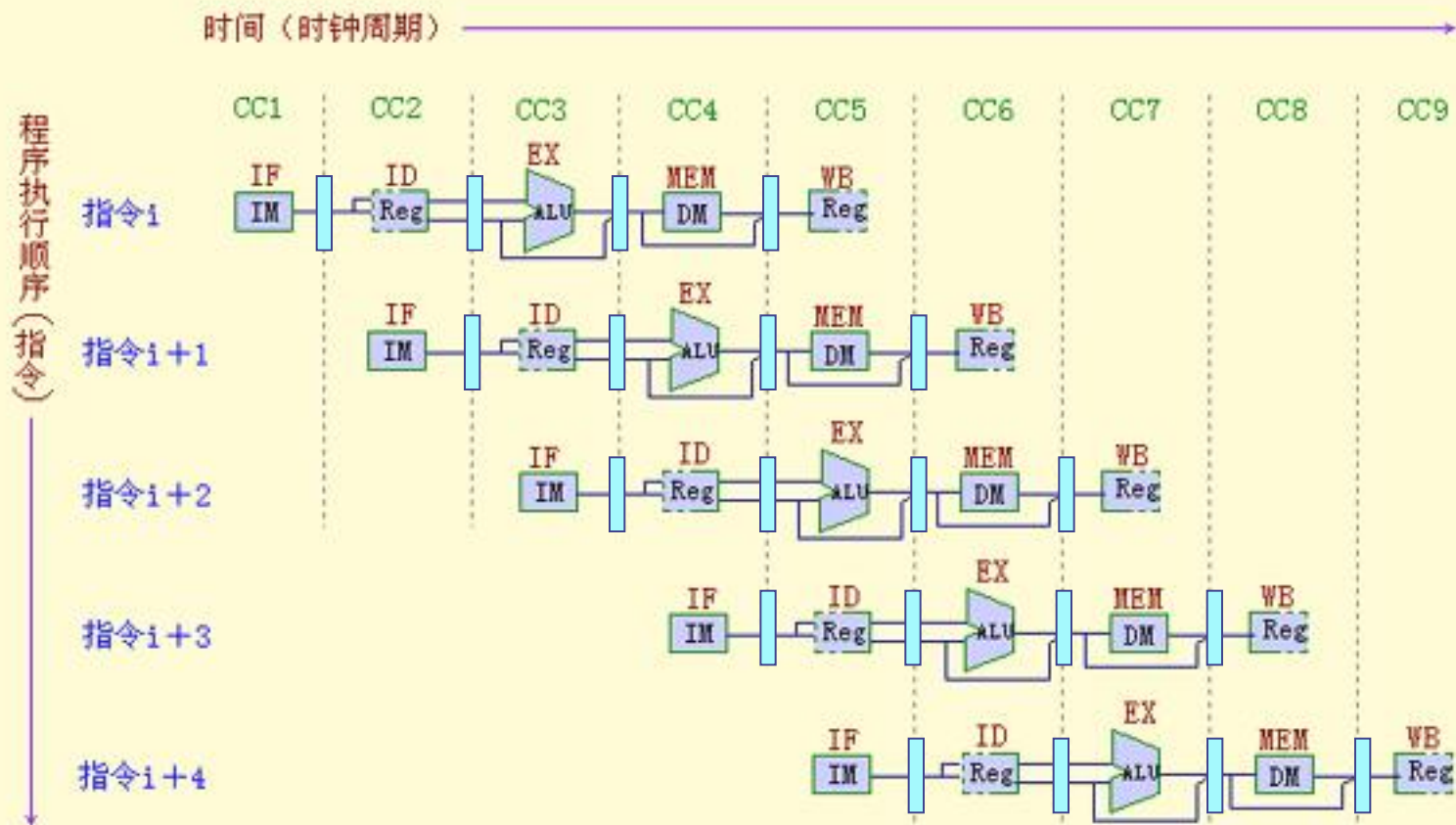
4. 5段流水线的两种描述方式

- 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



3.4.2 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

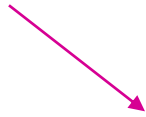


- 数据相关（也称真数据相关）
- 名相关
- 控制相关

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，即如何从其生产者流动到其消费者。

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, F8, F4

ADD. D F8, F0, F12

SUB. D F10, F8, F14

DIV. D和ADD. D存在反相关。

进行寄存器换名（F8换成S）后，变成：

DIV. D F2, F8, F4

ADD. D S, F0, F12

SUB. D F10, 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它会改变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）。

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**。
- 常见的导致结构冲突的原因：
 - 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。

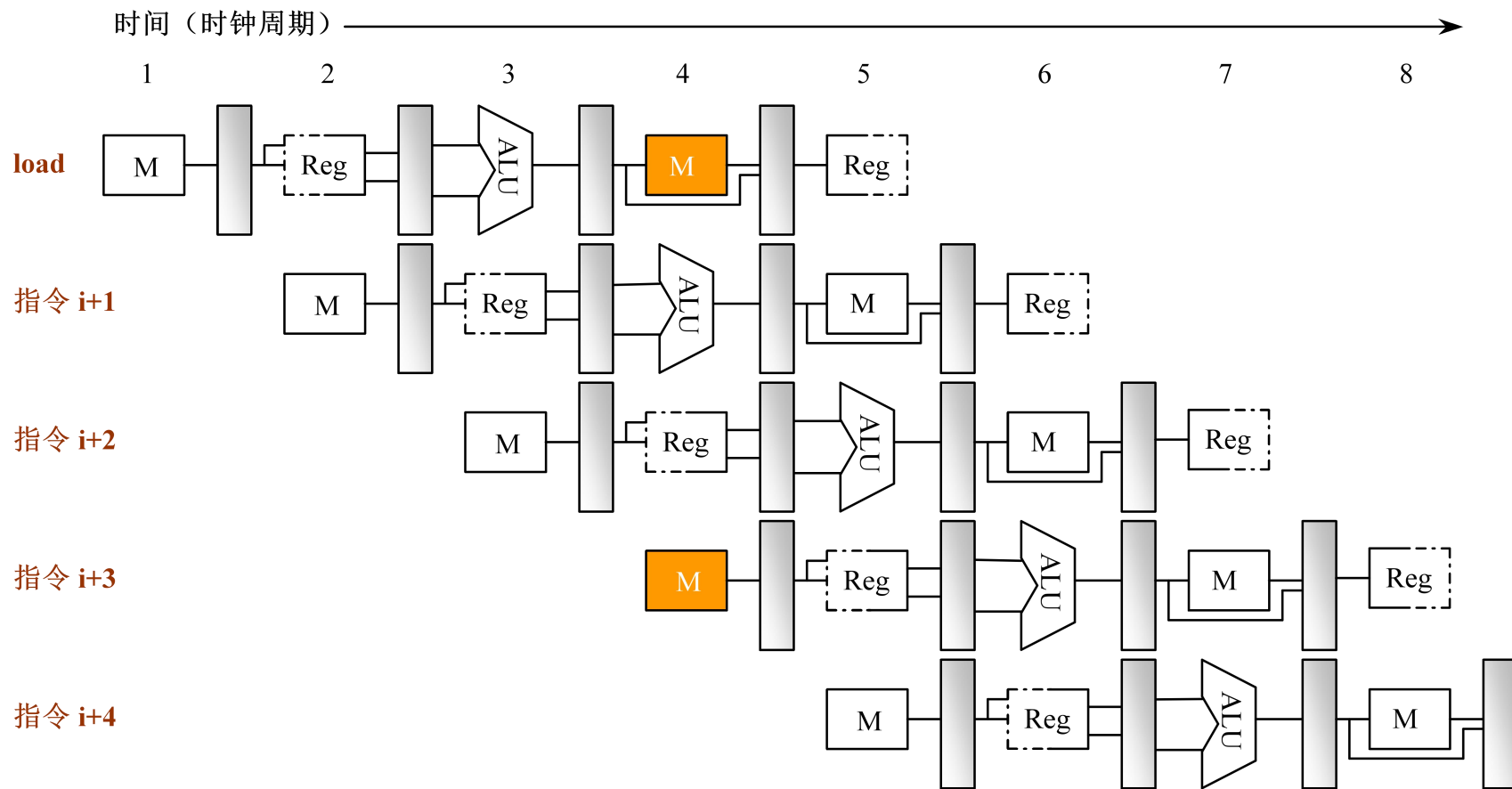
□ 解决办法I：插入暂停周期

（“流水线气泡”或“气泡”）

□ 解决方法II：

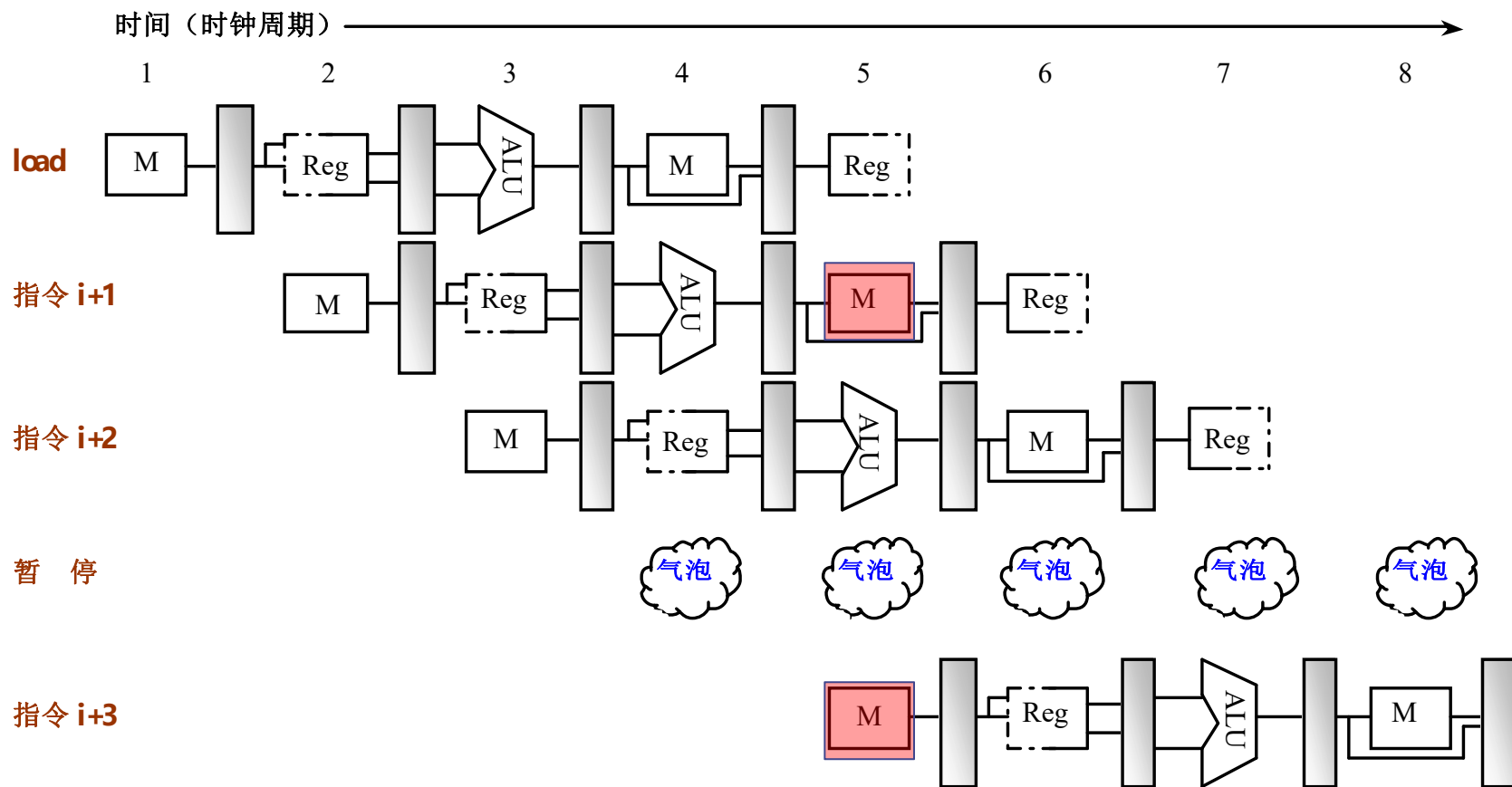
设置相互独立的指令存储器和数据存储器
或设置相互独立的指令Cache和数据Cache。

3.4 流水线的相关与冲突



由于访问同一个存储器而引起的结构冲突

3.4 流水线的相关与冲突



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
load	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，使之不同于它们串行执行时的顺序，则发生了**数据冲突**。

举例：

DADD R1, R2, R3

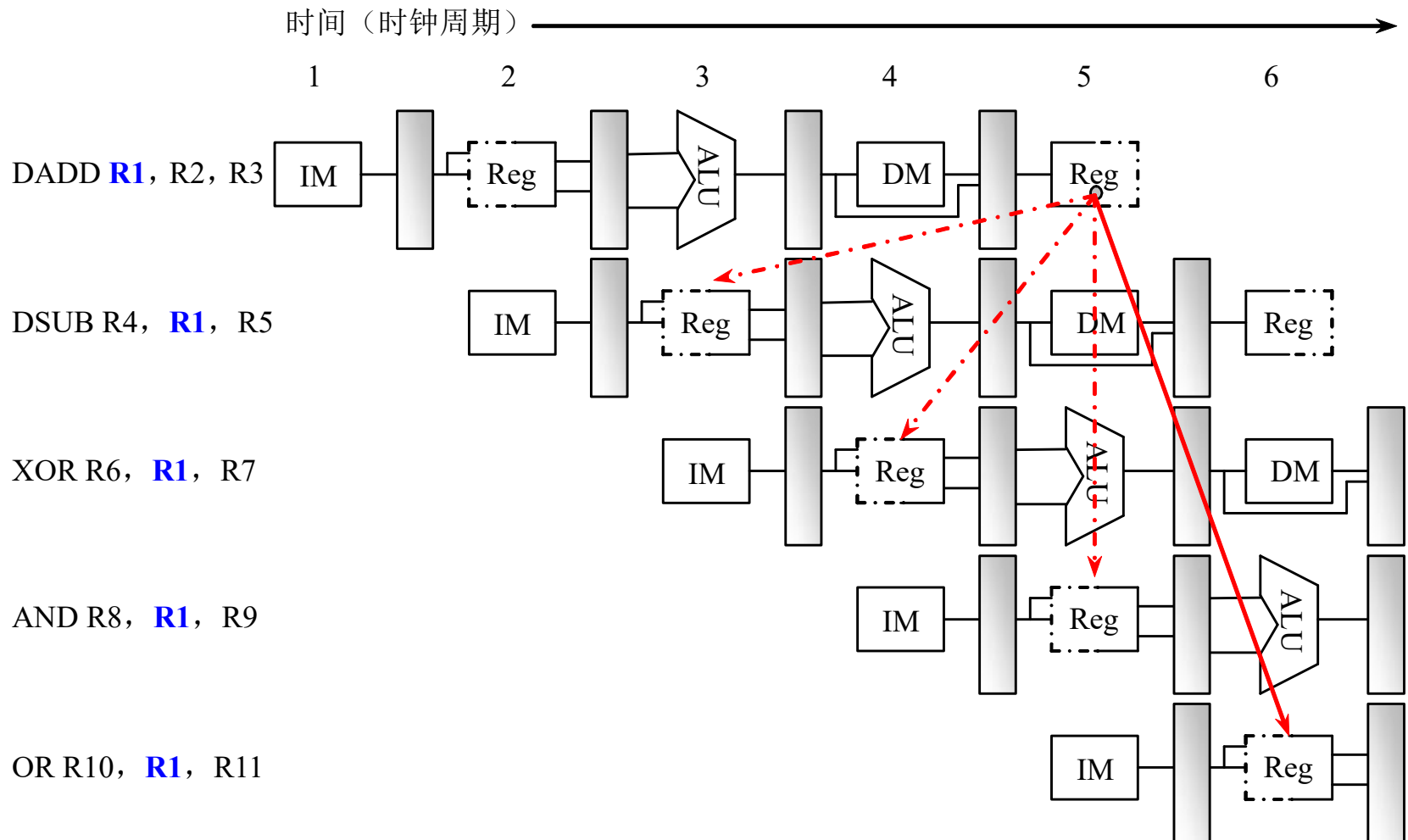
DSUB R4, R1, R5

XOR R6, R1, R7

AND R8, R1, R9

OR R10, R1, R11

3.4 流水线的相关与冲突



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;
- 指令被重新排序了。

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：

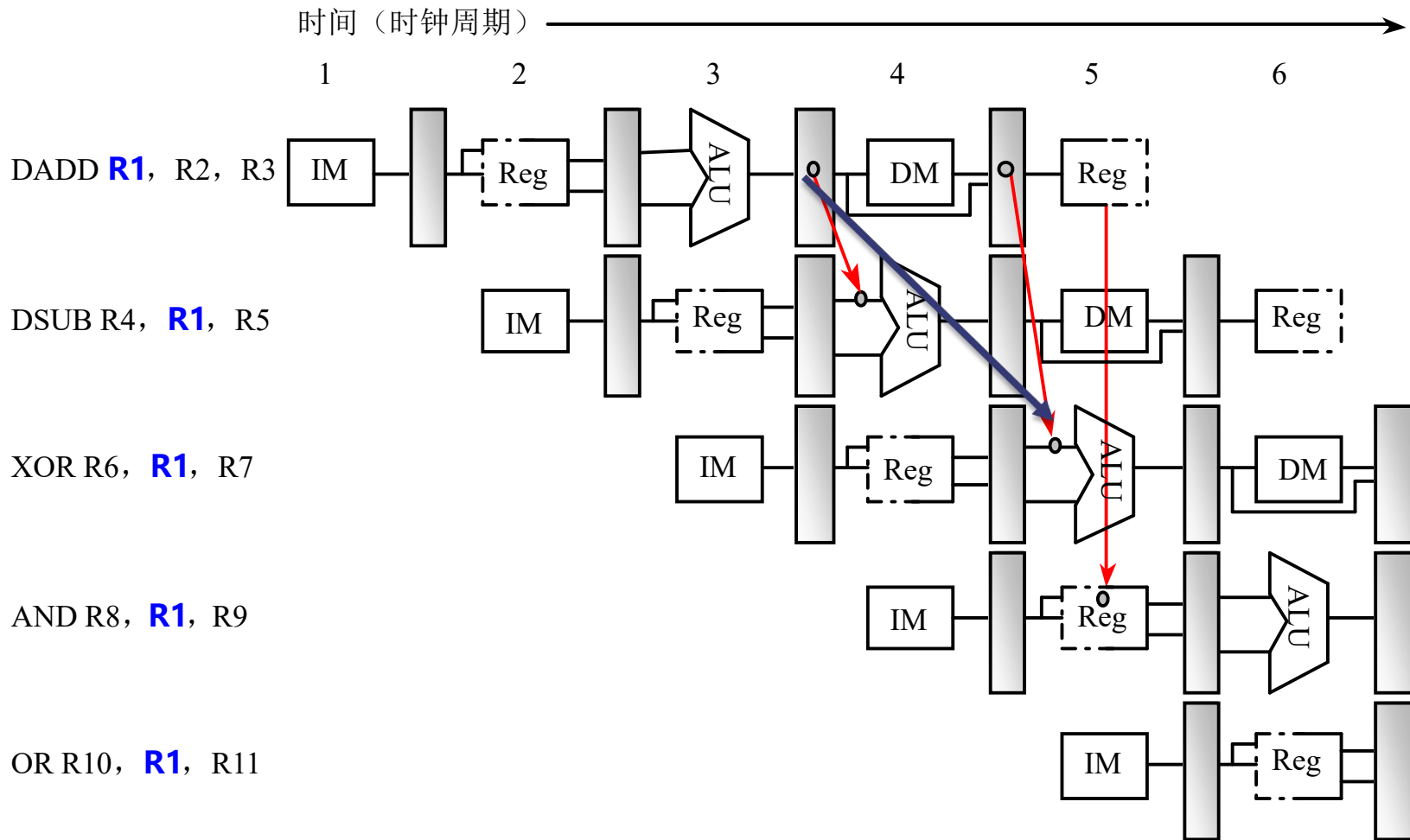
- 有些指令的写结果操作提前了，而且有些指令的读操作滞后了；
- 指令被重新排序了。

➤ 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：**在计算结果尚未出来之前，后面等待使用该结果的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它指令需要它的地方，那么就可以避免停顿。

3.4 流水线的相关与冲突



采用定向技术后的流水线数据通路

- 定向的实现
 - EX段和MEM段之间的流水寄存器中保存的ALU运算结果总是回送到ALU的入口。
 - 当定向硬件检测到前一个ALU运算结果写入的寄存器就是当前ALU操作的源寄存器时，那么控制逻辑就选择定向的数据作为ALU的输入，而不采用从通用寄存器组读出的数据。
- 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它功能部件的输入。

➤ 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:

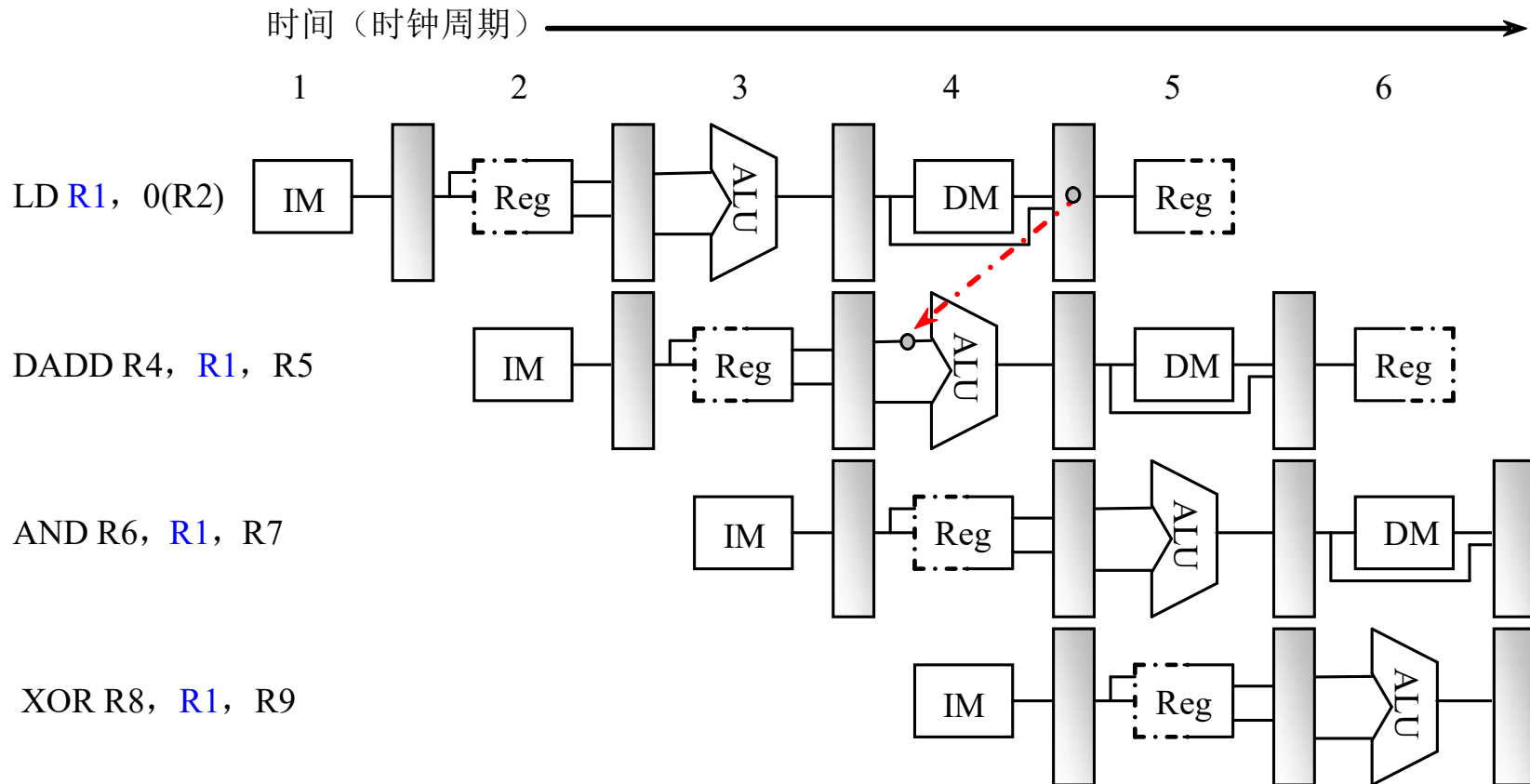
```
LD      R1, 0(R2)
DADD    R4, R1, R5
AND      R6, R1, R7
XOR      R8, R1, R9
```

- 增加流水线互锁机制（Pipeline Interlock），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。

MIPS的全称?

3.4 流水线的相关与冲突



无法将LD指令的结果定向到DADD指令

➤ 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**。

□ 举例：

请为下列表达式生成没有暂停的指令序列：

$A = B + C$ ；

$D = E - F$ ；

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，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

分支指令	IF	ID	EX	MEM	WB					
分支目标指令		IF	stall	stall	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
分支目标指令+3								IF	ID	EX

- 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，就好像什么都没发生似的；
- 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，并按分支目的地重新取指令执行。

要保证：分支结果出来之前不能改变处理机的状态，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”了流水线原来必需插入的暂停周期。

分支延迟指令的调度

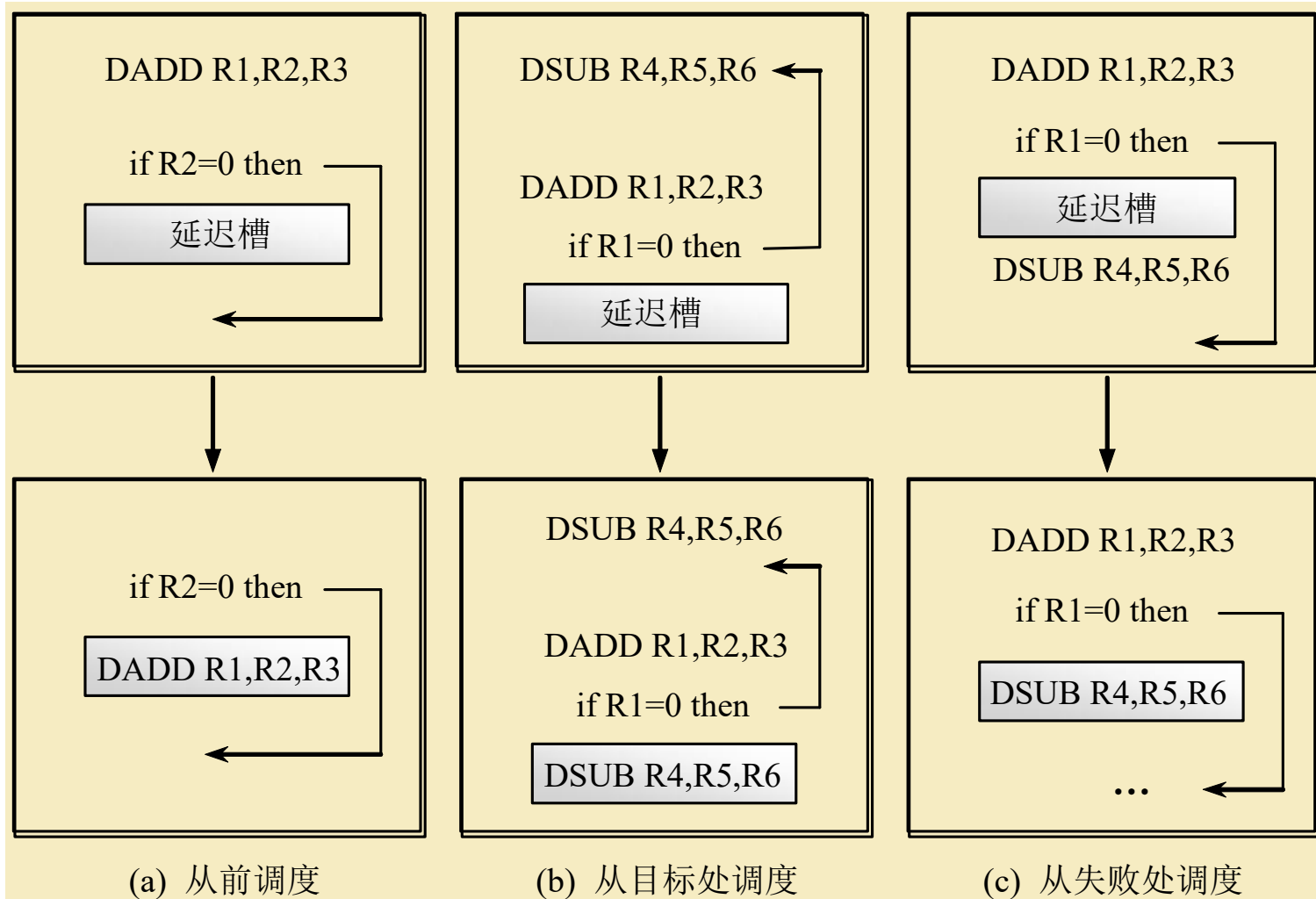
任务：在延迟槽中放入有用的指令

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为一个空操作。

“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 i+1		IF	idle	idle	idle	idle			
	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

预测分支成功的情况下，分支取消机制的执行情况

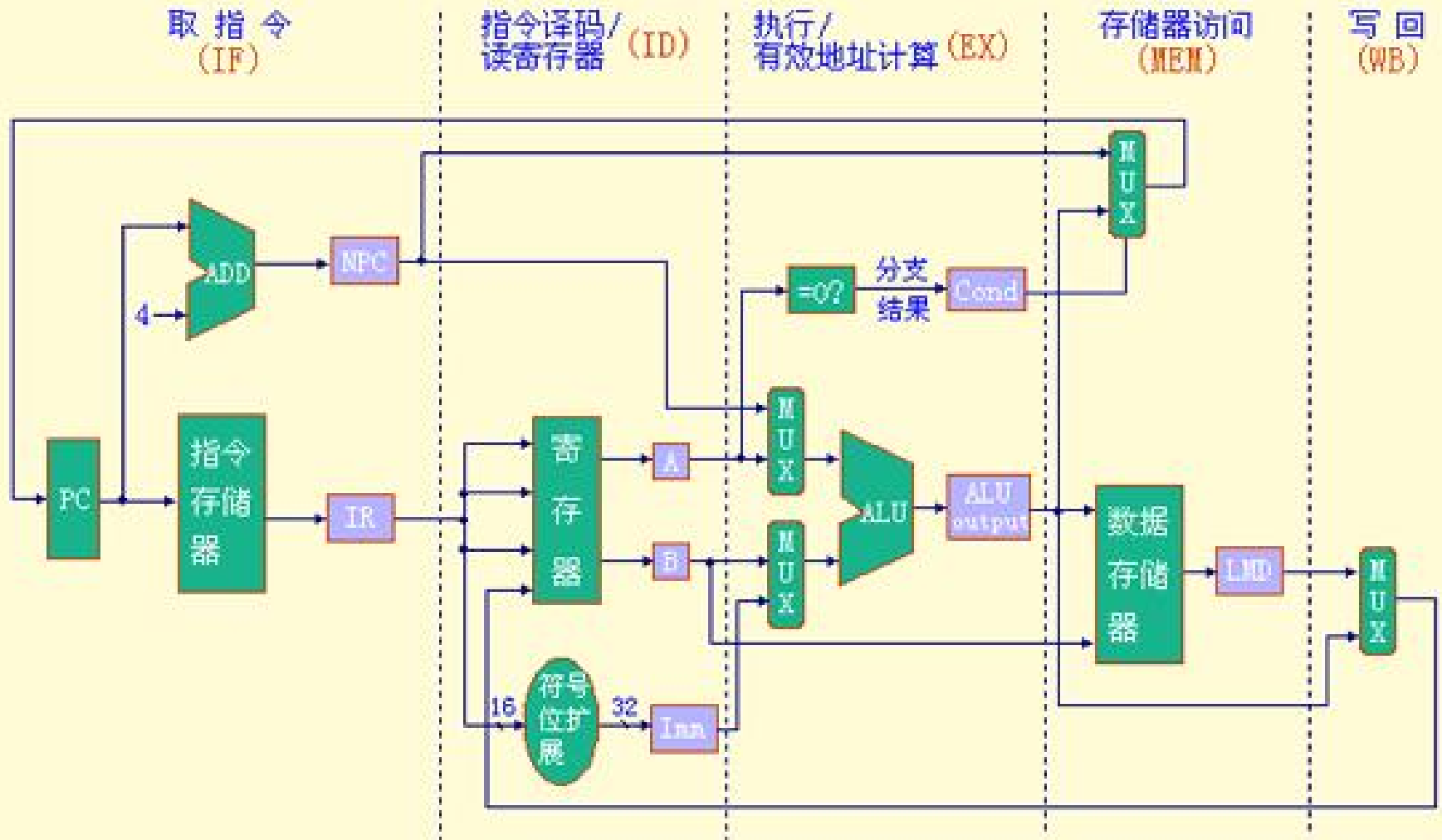
3.5 流水线的实现

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。）

实现MIPS指令的一种简单数据通路



➤ 设置了一些临时寄存器。其作用如下：

- **PC**：程序计数器，存放当前指令的地址。
- **NPC**：下一条程序计数器，存放下一条指令的地址。
- **IR**：指令寄存器，存放当前正在处理的指令。
- **A**：第一操作数寄存器，存放从通用寄存器组读出来的操作数。
- **B**：第二操作数寄存器，存放从通用寄存器组读出来的另一个操作数。
- **Imm**：存放符号扩展后的立即数操作数。
- **Cond**：存放条件判定的结果。为“真”表示分支成功。
- **ALUo**：存放ALU的运算结果。
- **LMD**：存放load指令从存储器读出的数据。

2. 一条MIPS指令最多需要以下5个时钟周期：

➤ 取指令周期（IF）

- $IR \leftarrow \text{Mem}[PC]$
- $NPC \leftarrow PC + 4$

➤ 指令译码/读寄存器周期（ID）

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

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：

- load指令和store指令

$$ALUo \leftarrow A + Imm$$

- 寄存器—寄存器ALU指令

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

- 寄存器—立即值ALU指令

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

- 分支指令

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

$$cond \leftarrow (A == 0)$$

为什么将有效地址计算周期和执行周期合并为一个时钟周期？

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，其它指令都是做： $PC \leftarrow NPC$
- 在该周期内处理的MIPS指令仅仅有load、store和分支三种指令。

- load指令和store指令

$LMD \leftarrow Mem[ALUo]$

或者 $Mem[ALUo] \leftarrow B$

- 分支指令

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

➤ 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

$Regs[rd] \leftarrow ALUo$

- 寄存器—立即数ALU指令

$Regs[rt] \leftarrow ALUo$

- load指令

$Regs[rt] \leftarrow LMD$

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

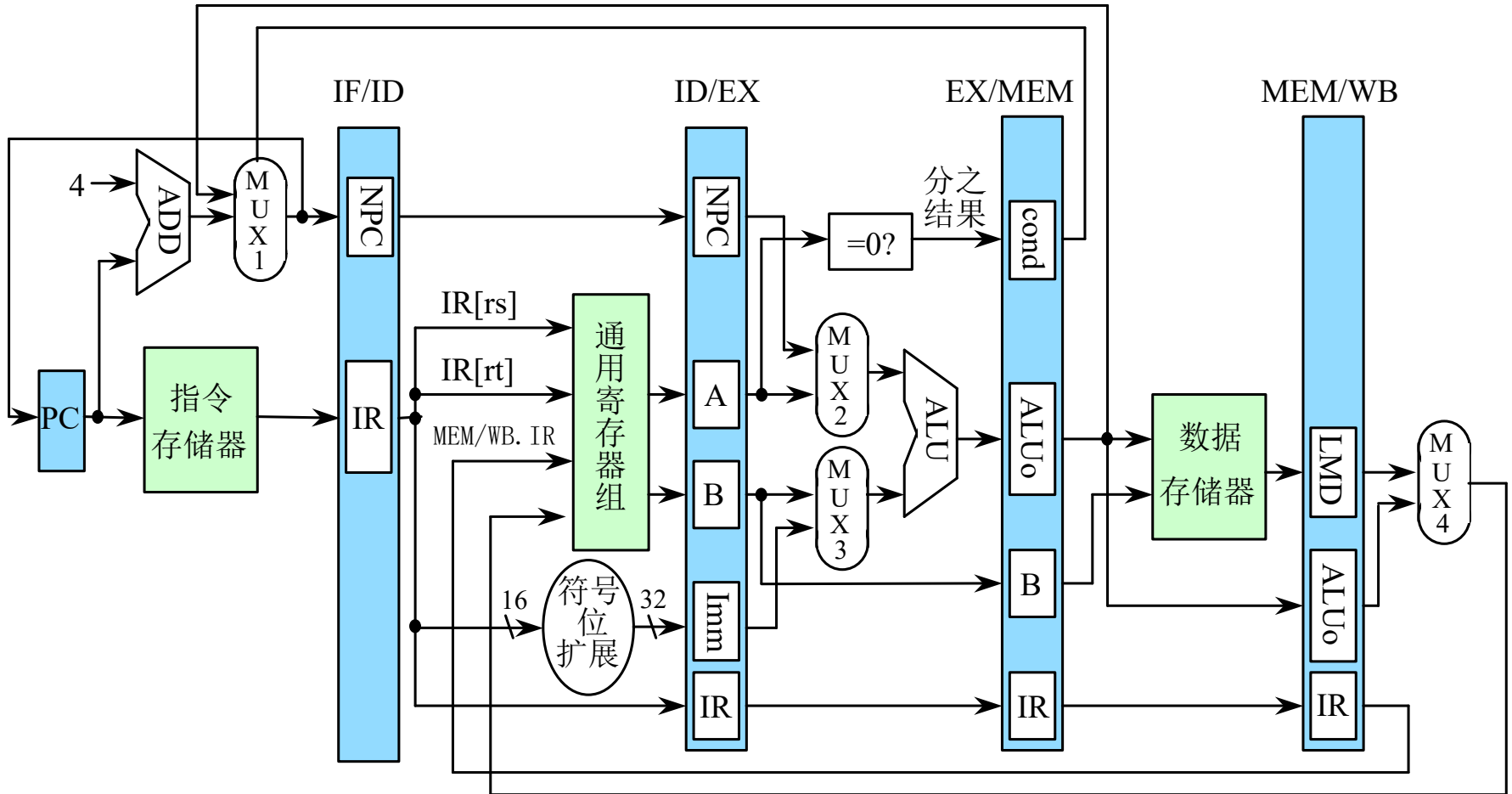
- 设置了流水寄存器
 - 段与段之间设置流水寄存器
 - 流水寄存器的名称

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子寄存器构成的

3.5 流水线的实现



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。

- 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。

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	$\text{IF/ID. IR} \leftarrow \text{Mem}[\text{PC}]$ $\text{IF/ID. NPC, PC} \leftarrow (\text{if} ((\text{EX/MEM. IR}[\text{op}] == \text{branch}) \& \text{EX/MEM. cond}) \{ \text{EX/MEM. ALUo} \} \text{ else } \{ \text{PC}+4 \}) ;$		
ID	$\text{ID/EX. A} \leftarrow \text{Regs}[\text{IF/ID. IR}[\text{rs}]] ; \text{ID/EX. B} \leftarrow \text{Regs}[\text{IF/ID. IR}[\text{rt}]] ;$ $\text{ID/EX. NPC} \leftarrow \text{IF/ID. NPC} ; \text{ID/EX. IR} \leftarrow \text{IF/ID. IR} ;$ $\text{ID/EX. Imm} \leftarrow (\text{IF/ID. IR}_{16})^{16} \# \text{IF/ID. IR}_{16..31} ;$		
	ALU 指令	load/store 指令	分支指令
EX	$\text{EX/MEM. IR} \leftarrow \text{ID/EX. IR} ;$ $\text{EX/MEM. ALUo} \leftarrow$ $\text{ID/EX. A } \textit{funct} \text{ ID/EX. B}$ <p>或</p> $\text{EX/MEM. ALUo} \leftarrow$ $\text{ID/EX. A } \textit{op} \text{ ID/EX. Imm} ;$	$\text{EX/MEM. IR} \leftarrow \text{ID/EX. IR} ;$ $\text{EX/MEM. ALUo} \leftarrow$ $\text{ID/EX. A} + \text{ID/EX. Imm} ;$ $\text{EX/MEM. B} \leftarrow \text{ID/EX. B} ;$	$\text{EX/MEM. IR} \leftarrow \text{ID/EX. IR} ;$ $\text{EX/MEM. ALUo} \leftarrow$ $\text{ID/EX. NPC} +$ $\text{ID/EX. Imm} \ll 2 ;$ $\text{EX/MEM. cond} \leftarrow$ $(\text{ID/EX. A} == 0) ;$

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	$\text{MEM/WB. IR} \leftarrow \text{EX/MEM. IR};$ $\text{MEM/WB. ALU}_o \leftarrow \text{EX/MEM. ALU}_o;$	$\text{MEM/WB. IR} \leftarrow \text{EX/MEM. IR};$ $\text{MEM/WB. LMD} \leftarrow \text{Mem}[\text{EX/MEM. ALU}_o];$ 或 $\text{Mem}[\text{EX/MEM. ALU}_o] \leftarrow \text{EX/MEM. B};$	
WB	$\text{Regs}[\text{MEM/WB. IR}[\text{rd}]] \leftarrow \text{MEM/WB. ALU}_o;$ 或 $\text{Regs}[\text{MEM/WB. IR}[\text{rt}]] \leftarrow \text{MEM/WB. ALU}_o;$	$\text{Regs}[\text{MEM/WB. IR}[\text{rt}]] \leftarrow \text{MEM/WB. LMD};$	

3. 流水线的控制

➤ 主要是如何控制四个多路选择器。

□ MUX2

if (ID/EX.IR[op]==“分支指令”)

{ MUX2_output=ID/EX.NPC };

else MUX2_output=ID/EX.A;

//MUX2_output表示MUX2的输出

□ MUX3

if (ID/EX.IR[op]==“寄存器—寄存器型ALU指令”)

{ MUX3_output=ID/EX.B };

else MUX3_output=ID/EX.Imm;

//MUX3_output表示MUX3的输出

□ MUX1

```
if ( (EX/MEM.IR[op]==“分支指令” ) & EX/MEM.cond )  
{ MUX1_output=EX/MEM.ALUo };  
else MUX1_output=PC+4;
```

//MUX1_output表示MUX1的输出

□ MUX4

```
if (MEM/WB.IR[op]==“load”)  
{ MUX4_output=MEM/WB.LMD };  
else MUX4_output=MEM/WB.ALUo
```

//MUX4_output表示MUX4的输出

- **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： $(ID/EX. IR. op == RR \text{ ALU}) \& (EX/MEM. IR. op == RR \text{ ALU}) \& (ID/EX. IR[rt] == EX/MEM. IR[rd])$

则： $EX/MEM. ALU_o$ 定向到ALU的下面一个输入

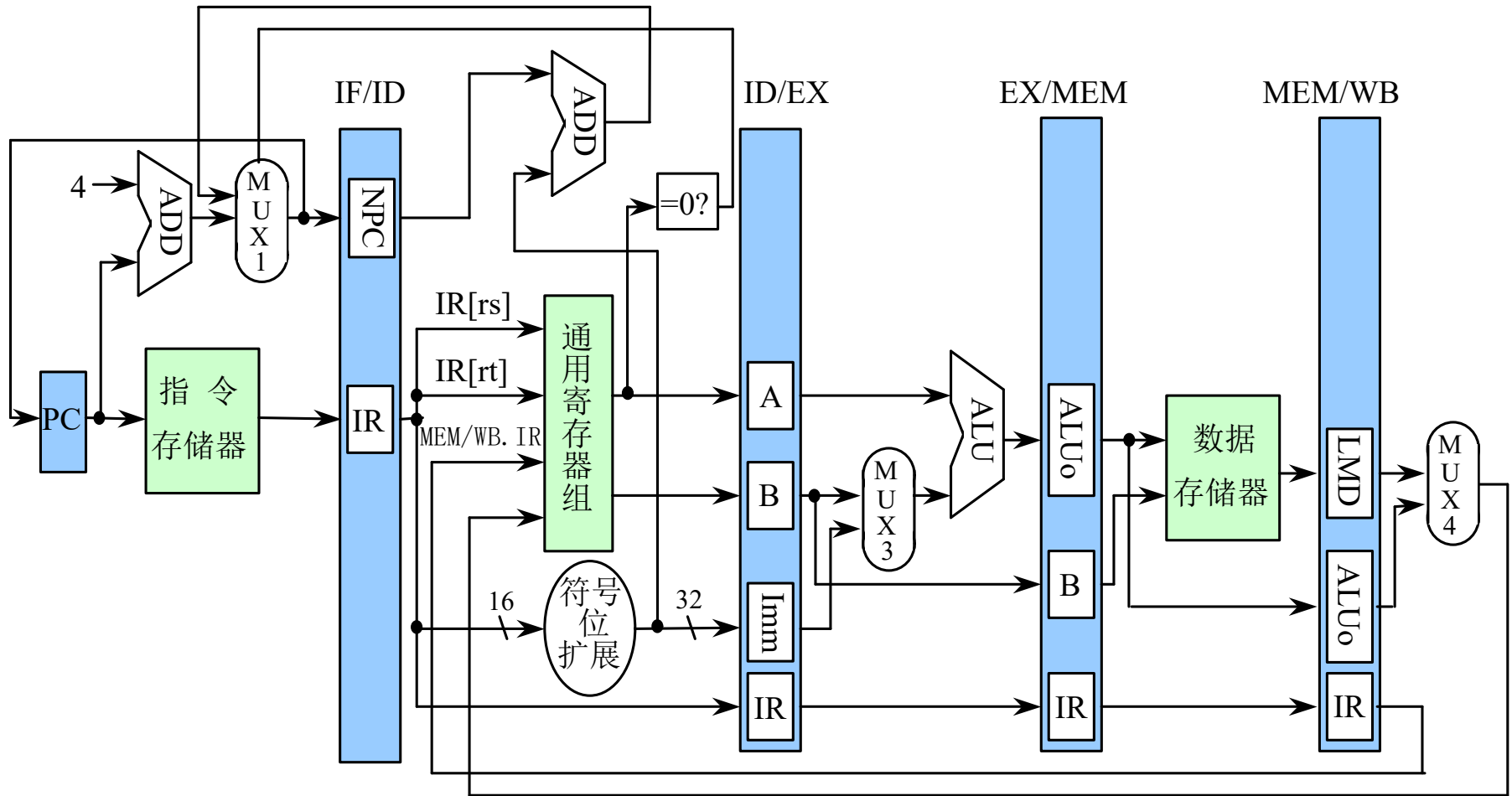
- 若： $(ID/EX. IR[op] == RR \text{ ALU}) \& (MEM/WB. IR[op] == load) \& (ID/EX. IR[rt] == MEM/WB. IR[rt])$

则：把 $MEM/WB. LMD$ 定向到ALU的下面一个输入

4. 控制冲突

- 分支指令的条件测试和分支目标地址计算是在EX段完成，对PC的修改是在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：作如下改进
 - （把上述工作提前到ID段进行）
 - 在ID段增设一个加法器：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

3.5 流水线的实现



改进后流水线的分支操作

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

作业： 3. 8, 3. 11

This data hazard can be detected quite easily when the program's machine code is written by the compiler. The original Stanford RISC machine relied on the compiler to add the NOP instructions in this case, rather than having the circuitry to detect and (more taxingly) stall the first two pipeline stages. Hence the name MIPS: **Microprocessor without Interlocked Pipeline Stages**. It turned out that the extra NOP instructions added by the compiler expanded the program binaries enough that the instruction cache hit rate was reduced. The stall hardware, although expensive, was put back into later designs to improve instruction cache hit rate, at which point the acronym no longer made sense.

——http://en.wikipedia.org/wiki/Classic_RISC_pipeline

[Return](#)