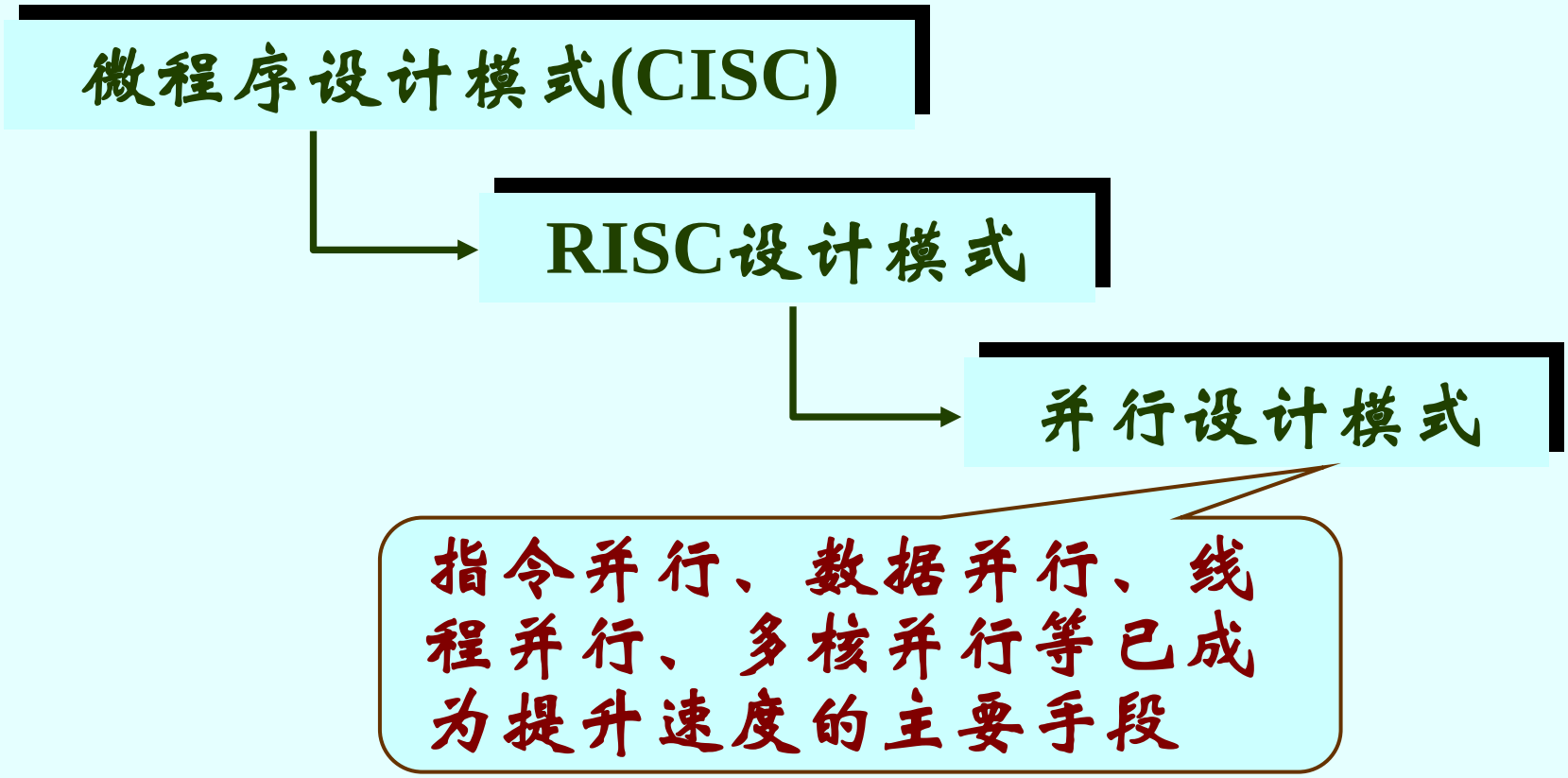
第三章 并行技术和高端处理器

提高计算机性能(速度)所经历的主要历程:



概述: 计算机中的并行性

1、并行性与同时性

- **并行性(parallelism)**
  在同一时刻或是同一时间间隔内完成两件以上的工作, 只要时间上有重叠, 就存在并行性。这一概念引入计算机系统, 成为提高性能的主要技术之一。
- **同时性(simultaneity)**
  两个或多个事件在同一时刻发生则是同时性。

以n位并行加法为例:

由于进位信号的传播延迟, 全部n位加法结果并不是在同一时刻产生, 因此不是同时性, 只是并发性。

但m个存储器模块同时读或写操作, 则属于同时性。

2、指令并行性和数据并行性

对程序运行, 并行性从低到高可分为:

(1) 指令内部并行: 指令内部的微操作之间的并行;

(2) 指令级并行(Instruction Level Parallel, ILP): 并行执行两条或多条指令;

(3) 任务级或过程级并行: 并行执行两个或多个过程或任务(程序段);

(4) 作业或程序级并行: 多个作业或程序间的并行。

对数据处理，并行性从低到高可以分为：

(1) 字串位串：同时只对一个字的一位进行处理；
(2) 字串位并：同时对一个字的全部位进行处理；
(3) 字并位串：同时对多字的同一位进行处理；
(4) 全并行：　同时对多字的全部或部分位进行处理。

在一个系统中，可以有执行程序方面和数据处理方面的多种并行性措施。当并行性到达一定级别，则认为进入并行处理领域

3、提高并行性的三种技术途径

(1) 时间重叠: 即多个处理过程在时间上相互错开, 轮流重叠使用同一硬件设备的不同部分, 以加快硬件周转。原则上不要求重复的硬件设备。

(2) 资源重复: 根据“以数量取胜”的原则, 通过重复设置资源(尤其是硬件资源), 来提高计算机系统的性能。多处理器系统是资源重复的典型。

(3) 资源共享: 用软件方法使多个任务按一定时间顺序轮流使用同一套硬件设备。如分时系统就是遵循资源共享这一思想。

4、系统设计的定量原理

(1) 大概率事件优先原则

大概率事件优先原则是计算机体系结构设计中最重要和最常用的原则。

基本思想是: 对于大概率事件(最常见事件), 赋予优先处理权和资源使用权, 以获得全局的最优结果。

该原则也适用于资源分配。着重改进大概率事件的性能, 能明显提高计算机性能。

(2) 阿姆达尔(Amdahl)定律

阿姆达尔定律是最基本的定量分析原则。该定理指系统中对某一部件采用某种更快的执行方式后, 所能获得的系统性能的改进程度, 取决于这种执行方式被使用的频率, 或所占总时间比例;

也可描述为: 加快某部件执行速度所获得的系统性能加速比, 受限于该部件在系统中所占的重要性。

定量计算过程:

假设:

To: 不采用任何改进措施完成某一任务的时间;

Te: 采取某种改进措施完成同一任务所需的时间;

fe: 改进部分所占系统所有部分的百分比;
(或改进部分在改进前占系统总运行时间的比例)

re: 改进部分在改进后比改进前可加快执行的倍数;

则系统性能加速比为:

$$Sp = \frac{\text{未采用改进措施执行某任务时间}}{\text{采用改进措施后执行某任务的时间}} = \frac{To}{Te}$$

$$= \frac{To(1-fe)+Tofe}{To(1-fe)+\frac{Tofe}{re}} = \frac{1}{(1-fe)+\frac{fe}{re}}$$

$$Sp = \frac{1}{(1-fe) + fe/re}$$

fe: 改进部分的比例
re: 改进部分可加快执行的倍数

式中:

当fe为0时(即没有改进), Sp为1(则性能没有提高);

当$fe \to 100\%$, $Sp \to re$; 即系统性能的提高接近改进部分提高的性能。

当$re \to \infty$时, 则$Sp=1/(1-fe)$, 则改进部分比例越大, 加速比越高(即加速比取决于改进部分的比例)。

当改进倍数re较大时, 式中(fe/re)值很小, 因此加速比主要受限于不可改进部分(1–fe), 仅提高可改进部分的速度, 对于提升系统加速比已经没有什么意义。

系统性能提高幅度受改进部分所占比例的限制

例1: 假设将某系统的某一部件的处理速度加快10倍, 该部件的原处理时间为整个运行时间的40%, 则该部件加速后, 整个系统的性能提高多少? 若该部件的原处理时间为整个运行时间的90%, 则整个系统的性能提高多少?

解: 由题意可知 fe=0.4, re=10, 根据阿姆达尔定律:

$$Sp1=\frac{1}{(1-fe)+\frac{fe}{re}}=\frac{1}{(1-0.4)+\frac{0.4}{10}}=1.56$$

$$Sp2=\frac{1}{(1-fe)+\frac{fe}{re}}=\frac{1}{(1-0.9)+\frac{0.9}{10}}=5.26$$

大概率事件优先原则的体现!

例2: 假设某程序中, 求浮点数平方根FPSQR的操作占整个程序执行时间的20%, 而所有浮点运算指令操作占整个程序执行的50%。有两种措施提高系统性能: 一种是采用FPSQR硬件, 使FPSQR操作的速度提高10倍; 另一种是使所有浮点指令的速度提高2倍, 试比较两种方案。

解: 在两种方案下, re分别是10和2, fe分别是20%和50%, 使用加速比公式:

$$S_{P\text{-}FPSQR} = \frac{1}{(1-fe)+(fe/re)} = \frac{1}{(1-0.2)+(0.2/10)} \approx 1.22$$

$$S_{P\text{-}FP} = \frac{1}{(1-fe)+(fe/re)} = \frac{1}{(1-0.5)+(0.5/2)} \approx 1.33$$

“提高所有浮点指令的速度”比“FPSQR硬件”更好。

根据 $Sp=\frac{1}{(1-fe)+fe/re}$，还可以认为:

Amdahl定律表达了一种性能增加的递减规则：如果仅仅对计算机中的一部分做性能改进，则改进越多，系统获得的效果越小(改进与效果不成线性关系)。

从另外一个方面看，Amdahl定律认为衡量一个“好”的计算机系统的原则是：一个带宽平衡的系统，而不是看它使用的某些部件的性能。

问题:

许多人对该定律的解释表明: 使用大量的处理器求解问题只能获得有限的成功, 但这似乎与大量的并行计算机能显著改进计算性能的现象相冲突。

阿姆达尔定律是在固定应用规模的前提下考虑并行性的增长。但大多数并行计算则是固定并行性但扩展应用的规模。在这种情况下, 随着规模的增加, 顺序代码所占的比例越来越小, 可并行执行的代码部分的比例越来越大。因而使用大量的处理器并行解决复杂问题能显著提升处理速度。

在多处理器(或多核)情况下的加速比:

在多处理器(或多核)情况下的加速比:

假设程序在单处理器上执行时间为t_s, 程序中不能分解为并行计算的部分所占比例为f, 那么程序在n个处理器(或n个内核)上执行时, 串行部分所占的时间为$f \times t_s$, 并行部分的执行时间为$(1-f)\times t_s/n$, 由此得到加速比:

$$S_P(n) = \frac{t_s}{f \times t_s + (1-f)\times t_s/n} = \frac{1}{f + (1-f)/n} < \frac{1}{f}$$

加速比受限于串行部分所占比例

例: (1) 用四核处理器, 须串行部分的比例50%, 其加速比:

$$S_P(n) = \frac{1}{f+(1-f)/n} = 1/(0.5+(1-0.5)/4) = 1.6$$

(2) 改用八核处理器, 串行部分仍50%, 其加速比:

$$S_P(n) = 1.77$$

(3) 不管用多少内核(从2核开始), 只要串并行比例不变: $S_P(n) \in (1.333, 2)$

(4) 用八核处理器(假设串行比例降为30%), 则加速比: $S_P(n) = 1/(0.3+(1-0.3)/8) = 2.58$

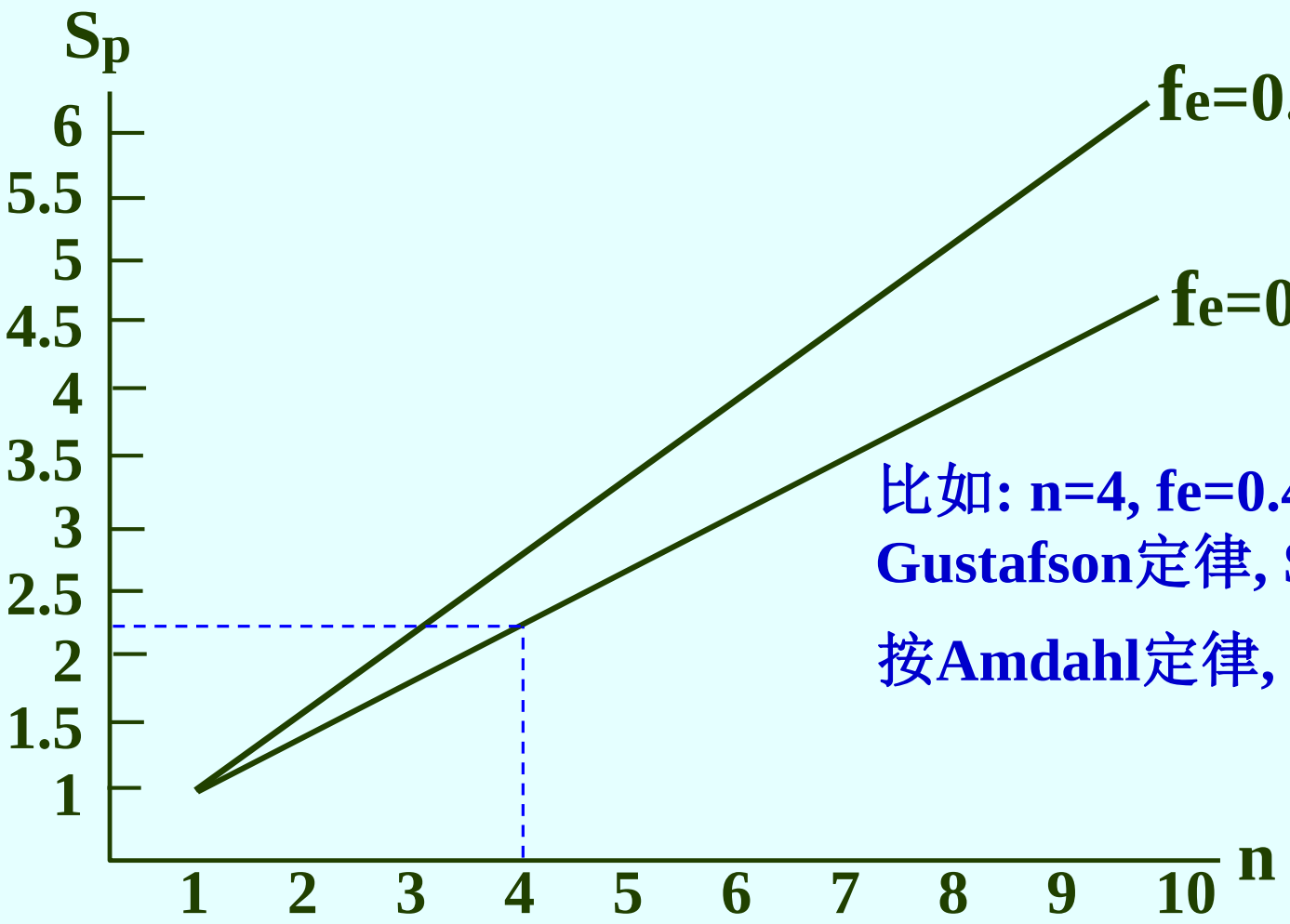
“乐观”的Gustafson定律

1988年, 美国Sandia国家实验室在1024个处理器的超立方体结构上观察了3个实际应用程序, 发现随着处理器的增加, 其性能线性增加的现象。John L. Gustafson基于此实验数据, 提出了一个新的计算加速比的公式:

$$S_p = n+(1–n)(1–fe)$$

S_p是加速比, n是处理器数量, fe是改进措施部分所占系统的比列。

以fe分别为0.4和0.6为例:



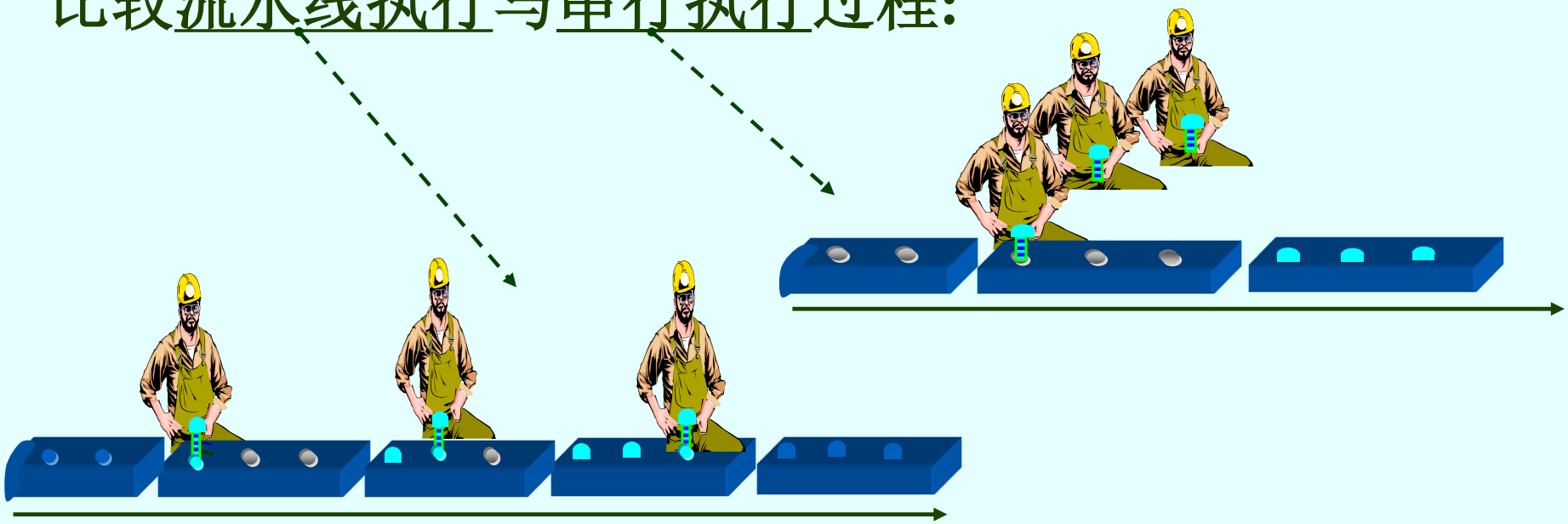
Gustafson定律说明: 在许多实际应用程序中得到接近线性的加速效果是可能的。

“Gustafson定律拯救了人们对并行处理的信心”

第一节 流水线技术

流水技术是指：将一个重复的时序过程分解成为若干个子过程，而每个子过程都可以在其专用功能段上与其它子过程同时执行。

比较流水线执行与串行执行过程：

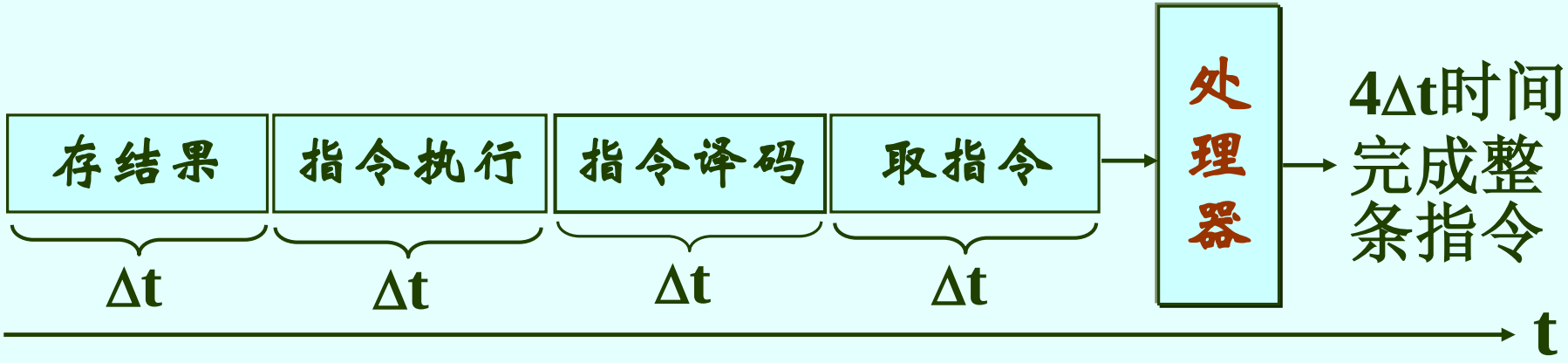


理论上讲：效率为串行的3倍

一、指令流水线(整数)

假设: 将一条指令的整个过程分解为四个阶段:

在非流水线方式下(假设各阶段时间等长):

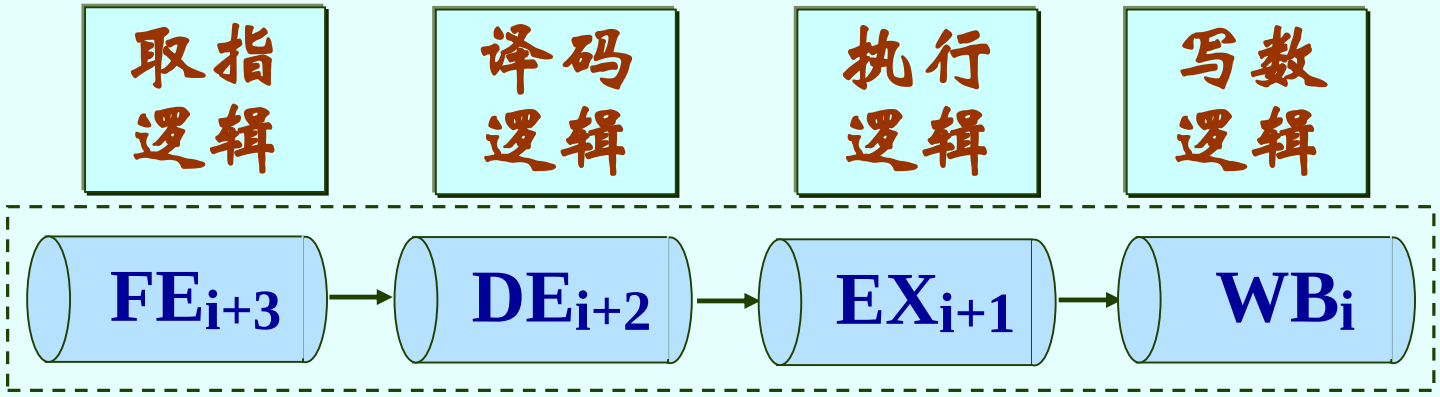


完成n条指令需要n×4Δt 时间。

如果将处理器按照指令各阶段的功能重新设计:

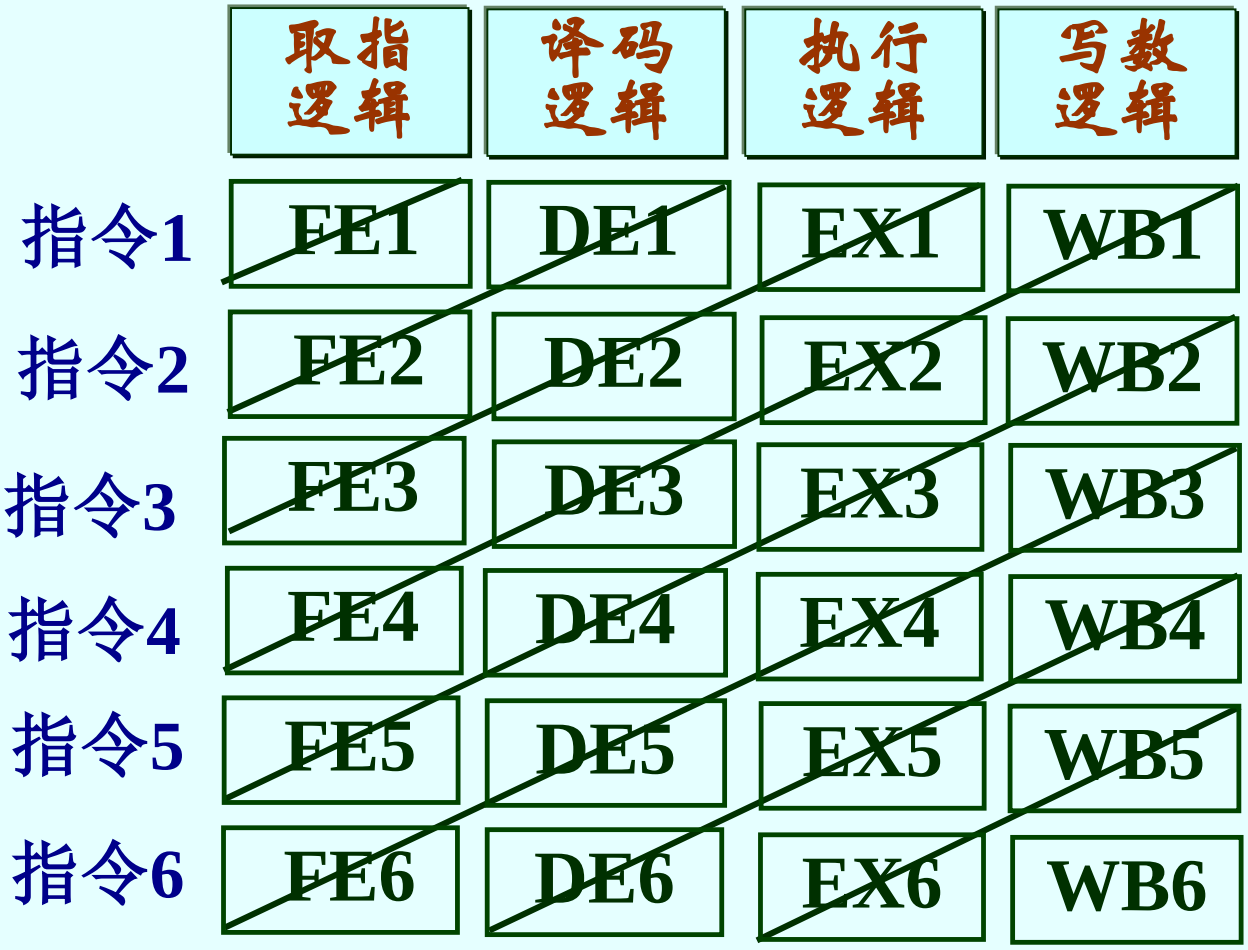
让完成不同功能所涉及的功能部件在逻辑上相互独立,让不同功能部件在时间上并行工作,从而使多条指令的不同阶段的功能在不同的功能部件上并行完成,以提高指令执行速度。

如下图所示:

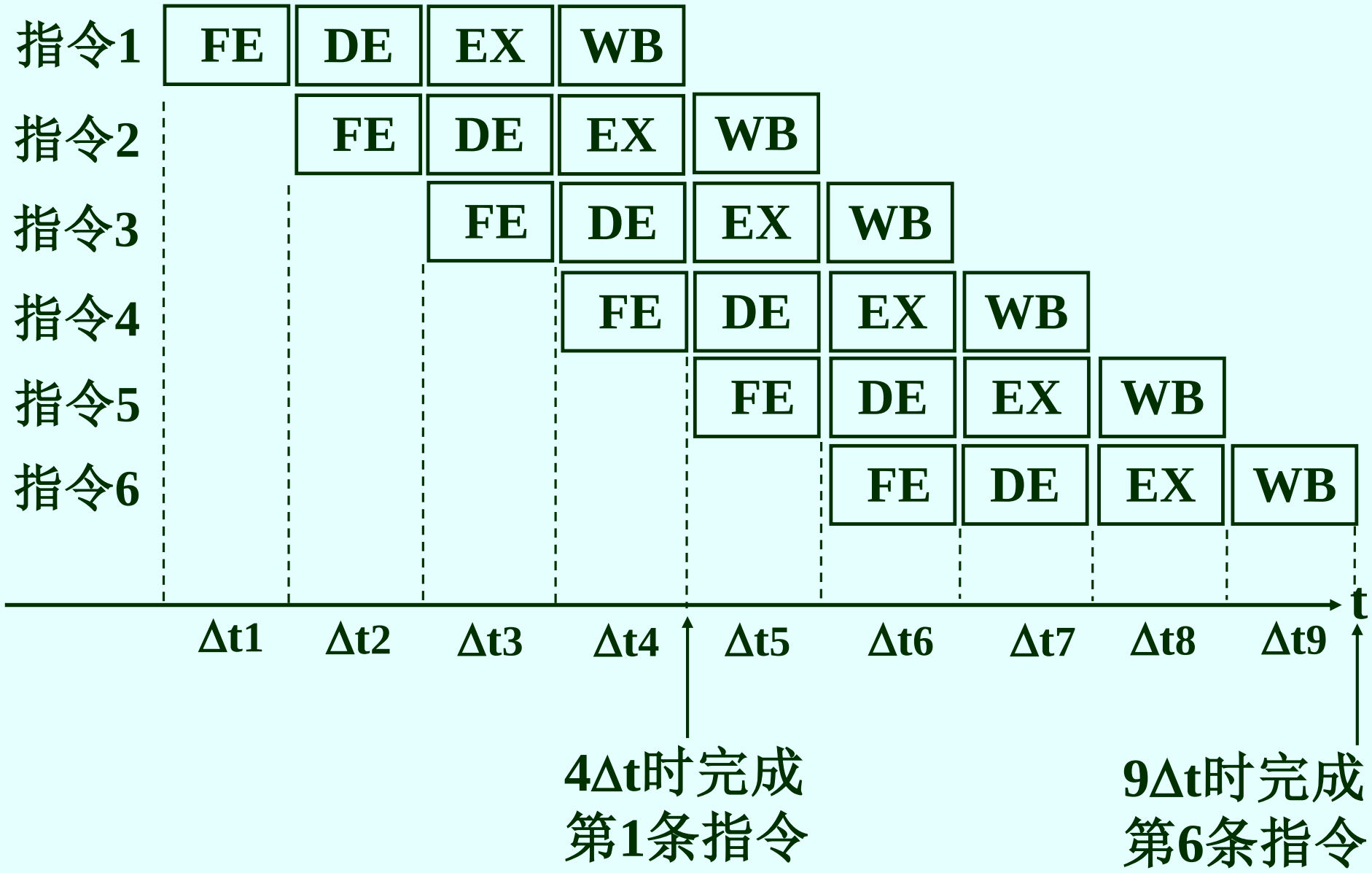


i是指令在程序中序号

例: 以下几条指令的执行过程:



注: 该图不反映流水线时间重叠关系, 只反映指令在不同阶段的功能在什么部件上完成。将该图按执行的时间顺序改写为下图:


指令1
指令2
指令3
指令4
指令5
指令6
FE
DE
EX
WB
t
Δt1
Δt2
Δt3
Δt4
Δt5
Δt6
Δt7
Δt8
Δt9
4Δt时完成
第1条指令
9Δt时完成
第6条指令

二、浮点加法流水线

浮点加法的全过程为求阶差、对阶、尾数相加、规格化四个子过程, 分别用不同的部件来实现。

假设每一阶段的时间为Δt, 采用流水线以后, 虽然每个加法操作的总时间仍然是4Δt , 但在加法器的输出端, 每一个Δt就输出一个加法结果。

三、流水线分类

1、线性流水线与非线性流水线

按照流水线的各个流水段之间是否有反馈信号, 可把流水线分为线性流水线和非线性流水线两类。

(1) 线性流水线(Linear Pipelining)

线性流水线是将流水线的各段逐个串接起来。输入数据从流水线的一端进入, 从另一端输出。数据在每一个流水段都流过一次, 而且仅流过一次。

如: 指令流水线

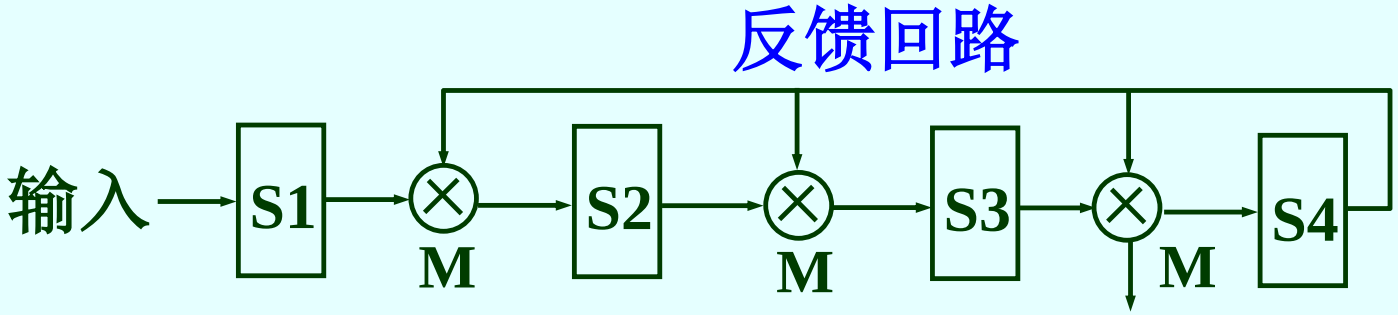
(取指→译码→ 执行→写结果)

浮点加法器流水线

(阶差→对阶→尾数相加→规格化)等

(2) 非线性流水线(Nonlinear Pipelining)

即: 在流水线的各段之间除了有串行的连接外, 还有前馈和反馈连接, 每一个功能段都可以输出, 并在各功能段之后, 还增加了一个多路选择器(M), 可以选择不同的输入。如图所示:



由于反馈回路的存在, 在一次流水过程中, 有的段可能被多次使用。比如: 从输入到输出可能依次流过: S1→S2→S3→S4→S2→S3, 其中S2和S3使用了两次。

非线性流水线, 通常是一个多功能流水线, 多用于递归处理, 这种结构对控制的要求很高。

2、单功能与多功能流水线

(1) 单功能流水线(Unifunction Pipelining)

一条流水线只能完成一种固定的功能的流水线称为单功能流水线(Unifunction Pipelining)。例如:浮点加法器流水线专门完成浮点加法运算, 浮点乘法器流水线专门完成浮点乘法运算。当要实现多种不同功能时, 可以采用多条单功能流水线。

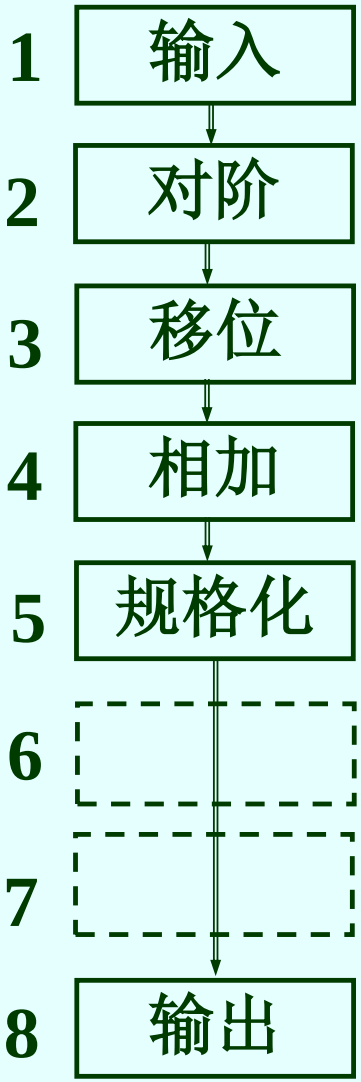
如Cray-1计算机中有12条单功能流水线, Pentium处理器有一条5级整数运算流水线和一条8级浮点运算流水线。Alpha 21064处理机有三条流水线, 其中, 整数操作和访问存储器操作为7级流水线, 浮点运算操作为10级流水线。

(2) 多功能流水线(Multifunction Pipelining)

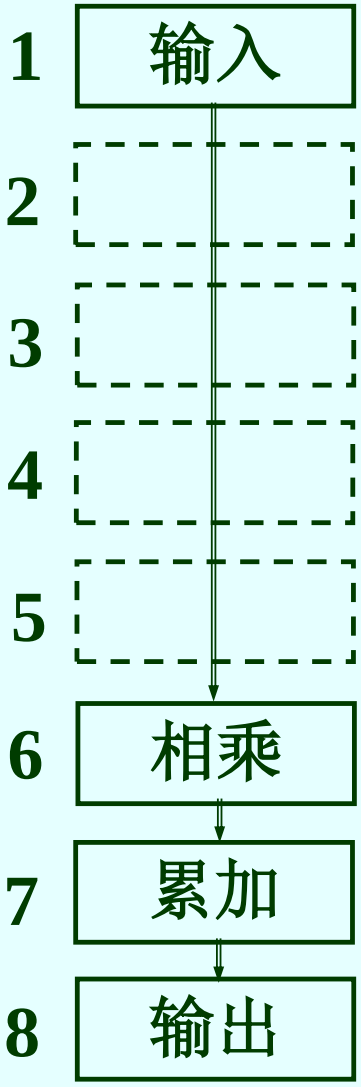
指流水线的各段可以进行不同的连接。通过不同的连接方式实现不同的功能, 使流水线各功能部件利用率比较高。

比如ASC处理器的多功能流水线示意:

ASC处理器中流水线



浮点加减
运算时的连接



定点乘法
运算时的连接

3、静态流水线与动态流水线

在多功能流水线中，按照在同一时间内是否能够连接成多种方式，同时执行多种功能，可以把多功能流水线分为静态流水线和动态流水线两种。

在同一段时间内，各流水段只能按照一种固定的方式连接，实现一种固定的功能

在同一段时间内，流水线的各段可以按照不同方式连接，同时执行多种功能

在一般情况下，动态流水线的效率和功能部件的利用率要比静态流水线高，但是，动态流水线的控制比静态流水线要复杂得多。因此大多数处理器中均采用静态流水线。

四、流水线性能指标

(1) 流水线吞吐率TP

是指在单位时间内，流水线所完成的任务数量或输出结果的数量。

针对指令流水线，则为指令数量

$$TP = \frac{n}{T_k}$$ (吞吐率基本公式)

其中：n为任务数量(对指令流水线，则为完成的指令数)，T_k是处理完n个任务所用的时间。

吞吐率基本公式可根据流水线的具体设计，进一步细分下述两种情况：

① 各时间段均等的流水线

假设指令流水线各段时间均等(=Δt), 级数为k , 第一条指令输入后, 经过k×Δt的时间完成, 此后的每一个Δt完成一条指令, 这样, 流水线完成n条指令所需时间为:

$$T_k = k\times\Delta t + (n-1)\Delta t \quad \text{(流水线排空时间)}$$

代入吞吐率基本公式:

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

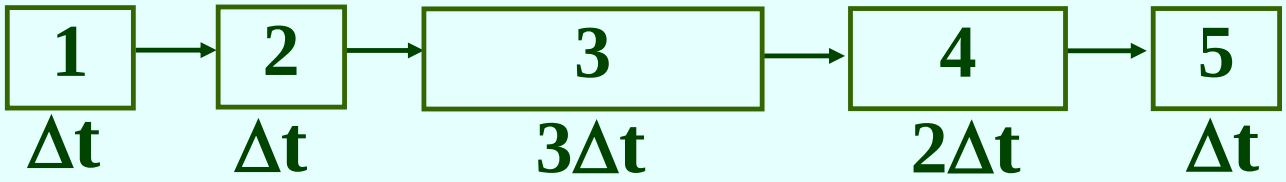
最大吞吐率: (当$n\to\infty$)

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

② 各段时间不完全相等的流水线

一般情况下, 不同的指令以及指令的不同阶段所需时间可能存在差异, 因此可以按实际所需时间分配每一级的时间长度。但存在控制很复杂、时间瓶颈使部分功能部件利用率较低的缺陷。

比如: 假设采用5级流水线, 按以下分配时间长度:

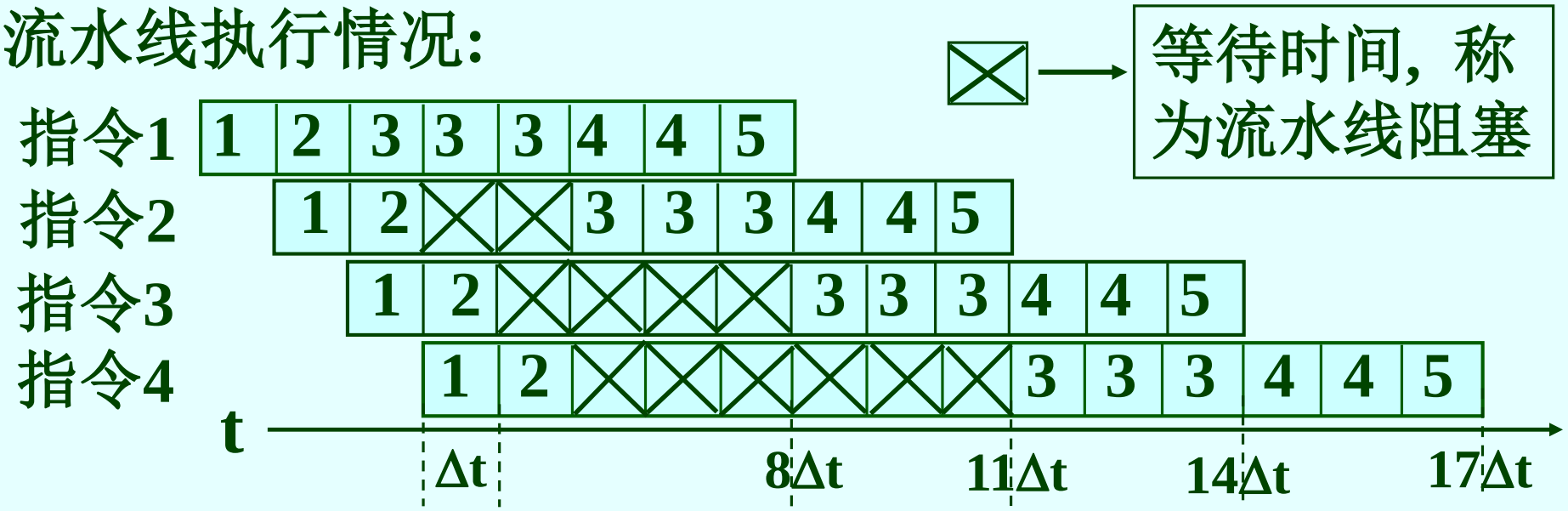


即: $\Delta t_1=\Delta t$, $\Delta t_2=\Delta t$, $\Delta t_3=3\Delta t$, $\Delta t_4=2\Delta t$, $\Delta t_5=\Delta t$

第2级部件需等待第3级部件2Δt的时间之后才能将操作交给第3级部件;

流水线执行情况:

流水线执行情况：



从第2条指令开始，每3Δt完成1条指令，因此有：

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

最大吞吐率：$TP = 1/Max(\Delta t_1, \Delta t_2, \ldots \Delta t_k)$

如上例中，最大吞吐率：$TP = 1/3\Delta t$

(2) 流水线加速比

流水线加速比定义：

按顺序方式(非流水线)执行一批指令所用的时间与按流水线方式执行同一批指令所用时间之比。

加速比：$S_p = \dfrac{T_s}{T_k}$

T_s ← 顺序方式执行所用时间

T_k ← 流水线方式执行所用时间

假设流水线各段时间均等为Δt，则k级流水线排空时间为: $T_k=(k+n-1)\Delta t$。

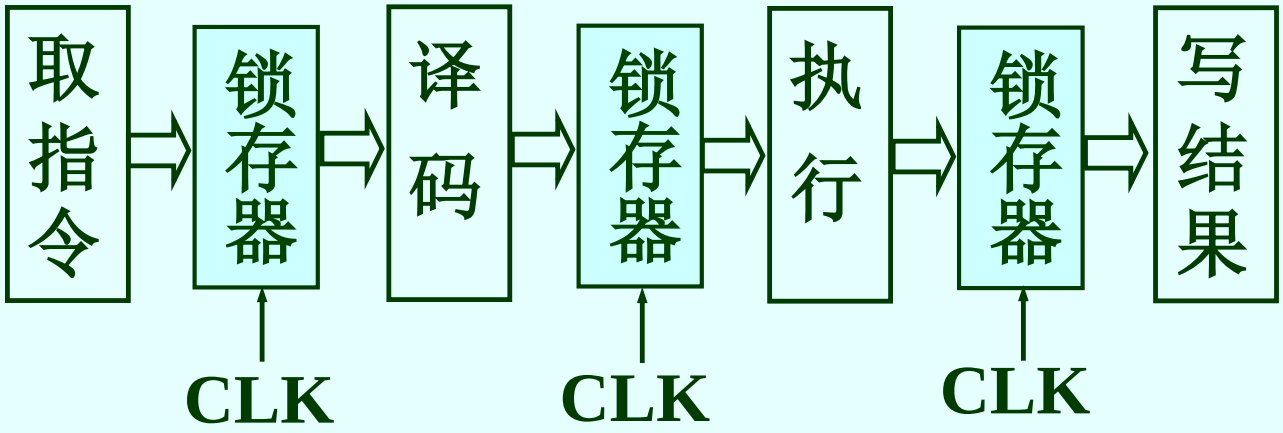
假设按顺序执行n条指令，则完成n条指令所需时间为: $T_s = nk\Delta t$ 。

因此有加速比: $S_p = \dfrac{nk}{k+n-1}$

最大加速比: $S_{max} = \lim\limits_{n\to\infty} \dfrac{nk}{k+n-1} = k$

即当n>>k时，流水线加速比等于流水线段数，因此理论上流水线段数越多越好，但在设计上会带来许多问题，性能也会受影响。

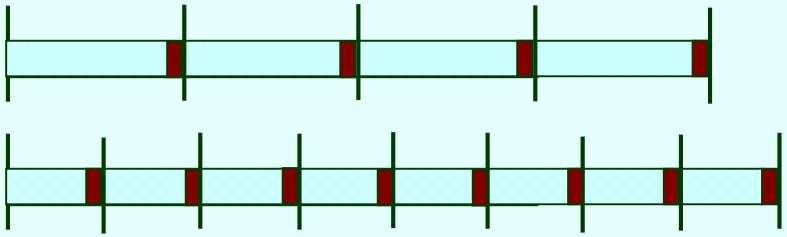
在指令流水线中, 每一段流水的延时时间(即各阶段的实际执行时间)不可能完全相等, 因此,每一级之后都需要有一个缓冲寄存器(也称锁存器)用于保存本流水段的执行结果, 以实现各段的时间同步。的如下图所示:

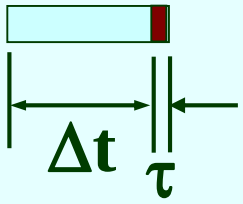


由时钟CLK将流水段的执行结果打入相应锁存器。

可以看出: 流水线级数的增加, 每一级功能完成时间缩短(设功能段时延Δt), 但每一级都有锁存器时延(假设: 一个锁存器时延均值为τ), 因此, 锁存器的总时延 (累加时延)占整个流水线的总时间随之增加, 因此, 流水线级数太多时, 用于完成指令功能的时间比例会减少 。

假设: 流水线级数由四级增加到八级:





每一级时间缩短一倍,但锁存器时延(非功能性时延)所占时间增加了, 因而完成一条指令的时间延长了。

(3) 流水线效率

流水线效率指: 流水线中的功能部件的利用率，其值为流水线功能部件的实际使用时间与整个运行时间之比。

假设流水线级数为k , 各级时间相等(Δt), 完成n条指令总时间为T_k，则每一级的效率为(理想情况):

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

由于各级相同, 则整条流水线利用率:

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

当n>>k 时， E趋于1。

例. 一个四级流水线(各级等长), 仅执行完一条指令时:

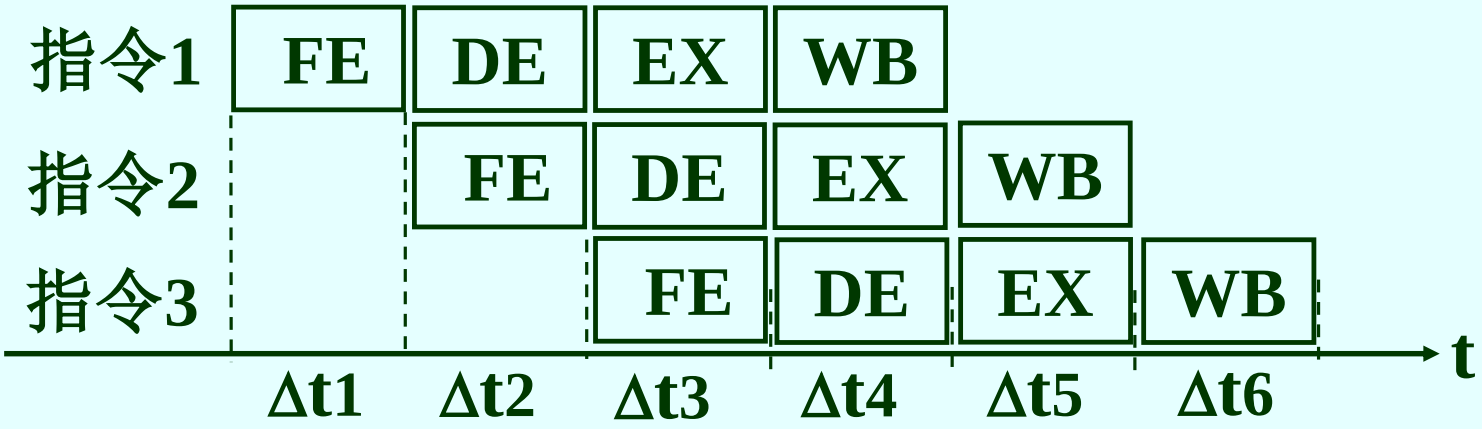
$$E = \frac{n}{(k+n-1)} = \frac{1}{(4+1-1)} = \frac{1}{4}$$

即: 由于仅执行完一条指令, 流水线四段中, 在任何一个Δt时间内, 都只有一个段被使用, 其余三个段为空闲, 因此利用率只有四分之一。

假设执行完三条指令, 且流水线无阻塞, 则:

$$E = \frac{n}{(k+n-1)} = \frac{3}{(4+3-1)} = \frac{1}{2}$$

流水线情况:



在流水线执行过程中, 各段空闲情况:

取指部件: 空闲Δt4、 Δt5、 Δt6

译码部件: 空闲Δt1、 Δt5、 Δt6

执行部件: 空闲Δt1、 Δt2、 Δt6

写数部件: 空闲Δt1、 Δt2、 Δt3

即在整个6Δt的时间里, 任何一个部件都存在3Δt的空闲, 利用率1/2。

如果各级的效率不完全相等, 则流水线效率为:

根据: $e_i = \frac{n\Delta t}{T_k}$ 可以推导出:

$$E = \frac{n\sum_{i=1}^{k}\Delta t_i}{K[\sum_{i=1}^{k}\Delta t_i + (n-1)\times \max(\Delta t_1, \Delta t_2 \ldots \Delta t_k)]}$$

上述计算公式都是流水线在理想情况的计算方法, 在实际情况, 流水线可能存在阻塞, 不能按理想状态实现流水执行。

非理想状况下流水线效率分析:

假设一段程序由n条指令构成的，分为K级流水，用Δt_{ij}表示第i条指令在第j阶段(使用部件j)所需时间。

(1) 如果采用顺序执行，则有：

第1条指令完成时间：$\Delta t_{11} + \Delta t_{12} + \Delta t_{13} \ldots + \Delta t_{1K}$

第2条指令完成时间：$\Delta t_{21} + \Delta t_{22} + \Delta t_{23} \ldots + \Delta t_{3K}$

第3条指令完成时间：$\Delta t_{31} + \Delta t_{32} + \Delta t_{33} \ldots + \Delta t_{3K}$

⋮ ⋮

第n条指令完成时间：$\Delta t_{n1} + \Delta t_{n2} + \Delta t_{n3} \ldots + \Delta t_{nK}$

顺序完成该程序所需时间为上述所有Δt_{ij}的累加：

$$T_S = \sum_{i=1}^{n} \sum_{j=1}^{k} \Delta t_{ij}$$

(2) 如果按流水线方式执行, 则有:

假设完成的总时间为T_K, 所谓效率则是流水线各功能部件被实际使用的时间占总时间的比例。

第1个部件使用时间: $\Delta t_{11}+\Delta t_{21}+\Delta t_{31}\ldots+\Delta t_{n1}$

第2个部件使用时间: $\Delta t_{12}+\Delta t_{22}+\Delta t_{32}\ldots+\Delta t_{n2}$

第3个部件使用时间: $\Delta t_{13}+\Delta t_{23}+\Delta t_{33}\ldots+\Delta t_{n3}$

⋮ ⋮

第K个部件使用时间: $\Delta t_{1K}+\Delta t_{2K}+\Delta t_{3K}\ldots+\Delta t_{nK}$

所有K个部件的平均使用时间为:

$$\frac{(\Delta t_{11}+\Delta t_{21}\ldots+\Delta t_{n1})+(\Delta t_{12}+\Delta t_{22}\ldots+\Delta t_{n2})+\ldots+(\Delta t_{1K}+\Delta t_{2K}\ldots+\Delta t_{nK})}{K}$$

$$=\frac{T_S}{K}$$

所有部件的平均使用时间(T_S/K)除以该程序完成的总时间(T_k)则为利用率, 即:

$$E = \frac{T_S/k}{T_k} = \frac{T_S}{k \times T_k}$$

T_S: 顺序方式执行所用时间

$k \times T_k$: 流水线执行所用时间

上式即为一种简单的实际的流水线效率计算方法。

例. 一个四级流水线(各级等长), 执行完三条指令时:

$$E = \frac{T_S}{k \times T_k} = \frac{12}{4 \times 6} = \frac{1}{2}$$

五、流水线的相关与冲突

1、流水线中的相关

相关是指两条指令之间存在某种依赖关系。

一般来说，指令之间的依赖性越少，流水线各级的并行性越高，流水线效率也越高。

反之，依赖性越多，流水线各级的并行性越低，流水线效率也越低。

有三种类型的相关：

数据相关(真数据相关)、名相关、控制相关

(1) 数据相关

假设两条指令Ins_i和Ins_j (Ins_i在Ins_j之前), 如果有下列情形之一, 则称指令Ins_i和Ins_j数据相关:

① Ins_j使用Ins_i产生的结果

② Ins_j与Ins_k数据相关, 而Ins_k又与Ins_i数据相关

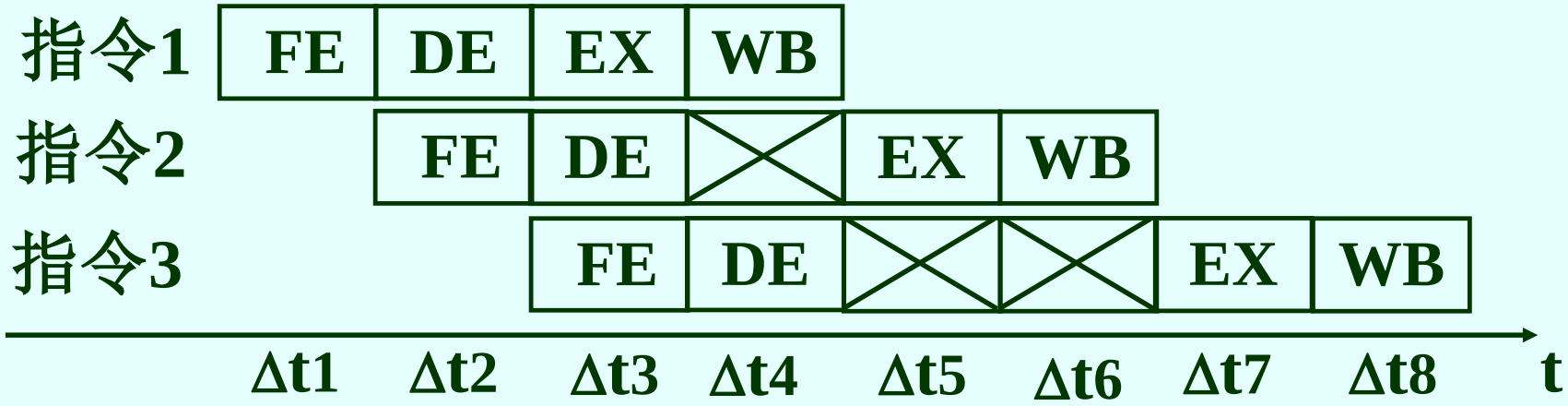
如下程序段存在数据相关:

第2条指令使用第1条指令产生的结果, 第3条指令使用第2条指令产生的结果。

```
MOV Ri , 8    ; 8→Ri
MOV Rj , Ri   ; Ri→Rj
ADD Rk , Rj   ; Rj+Rk→Rk
```

数据相关将引发流水线阻塞。

```
MOV Ri , 8   ; 8→Ri
MOV Rj , Ri  ; Ri→Rj
ADD Rk , Rj  ; Rj →Rk
```



流水线效率: $E = \frac{T_S}{k \times T_k} = \frac{12}{4 \times 8} = \frac{3}{8} \approx 37.5\%$

如果不考虑阻塞: $E = \frac{n}{(k+n-1)} = \frac{3}{(4+3-1)} = 50\%$

(理想情况)

由此可知, 软件算法(结构)对软件性能的影响!

例. 一个4级流水线, 完成数据A, B, C, D, E, F, G, H的累加, 假设这8个数据已在寄存器R_0–R_7中, 结果存入R_0。

$R_0+R_1 \rightarrow R_0$ (R_0=A+B)

R_0=A+B+C

程序1:

ADD R_0, R_1 ;

ADD R_0, R_2 ;

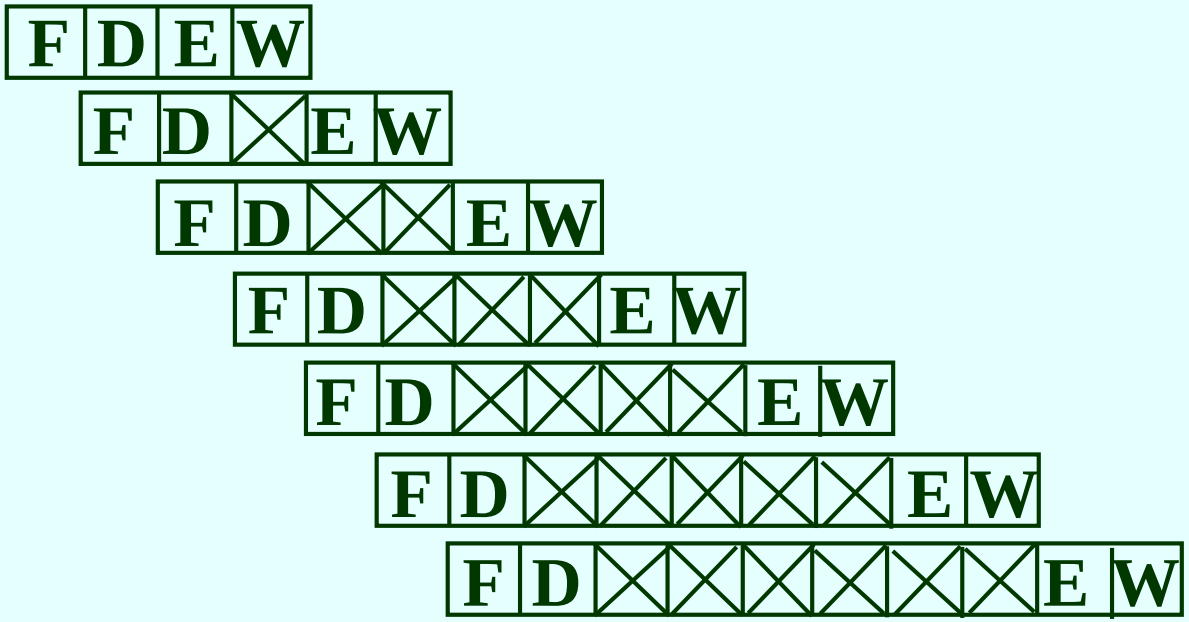
ADD R_0, R_3 ;

ADD R_0, R_4 ;

ADD R_0, R_5 ;

ADD R_0, R_6 ;

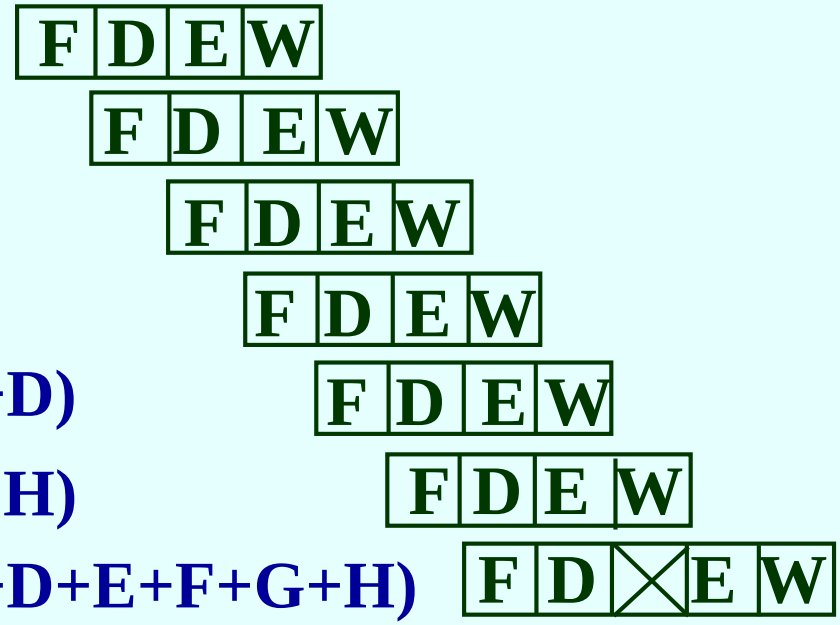
ADD R_0, R_7 ;



流水线效率: $E = \frac{T_S}{k \times T_k} = \frac{28}{4 \times 16} = 43.75\%$

程序2:

ADD R_0, R_1 ;	(R_0=A+B)
ADD R_2, R_3 ;	(R_2=C+D)
ADD R_4, R_5 ;	(R_4=E+F)
ADD R_6, R_7 ;	(R_6=G+H)
ADD R_0, R_2 ;	(R_0=A+B+C+D)
ADD R_4, R_6 ;	(R_4=E+F+G+H)
ADD R_0, R_4 ;	(R_0=A+B+C+D+E+F+G+H)



流水线效率: $E = \frac{T_S}{k \times T_k} = \frac{28}{4 \times 11} = 63.6\%$

(2) 名相关

两条指令使用了相同的寄存器或者存储单元, 但它们之间没有数据流动, 则称这两条指令存在名相关。由于仅使用了相同的名字而没有数据传输, 所以只需通过改变指令中操作数的名称(寄存器名或存储单元号)即可消除名相关(可通过编译或硬件实现)。

(3) 控制相关

控制相关是由分支指令引起的, 程序流向需根据分支指令执行的结果来确定。

由于相关的存在, 可能引发流水线冲突而导致流水线阻塞, 这是妨碍指令流水线高效运行的重要因素, 也是流水线技术要解决的重要问题

2、流水线冲突

(1) 数据冲突：一条指令执行过程中，需要前面的指令的执行结果。

(2) 控制冲突：分支指令未执行完成时，导致流水线不能正常往下执行。

(3) 结构冲突：硬件资源满足不了指令重叠执行的要求，而可能引发流水线阻塞。

六、分支预测技术

在程序运行过程中, 根据分支指令过去的表现来预测其将来的行为。采用分支预测应实现两个目的:

① 判断预测是否成功

② 尽快找到分支目标地址, 以避免流水线阻塞

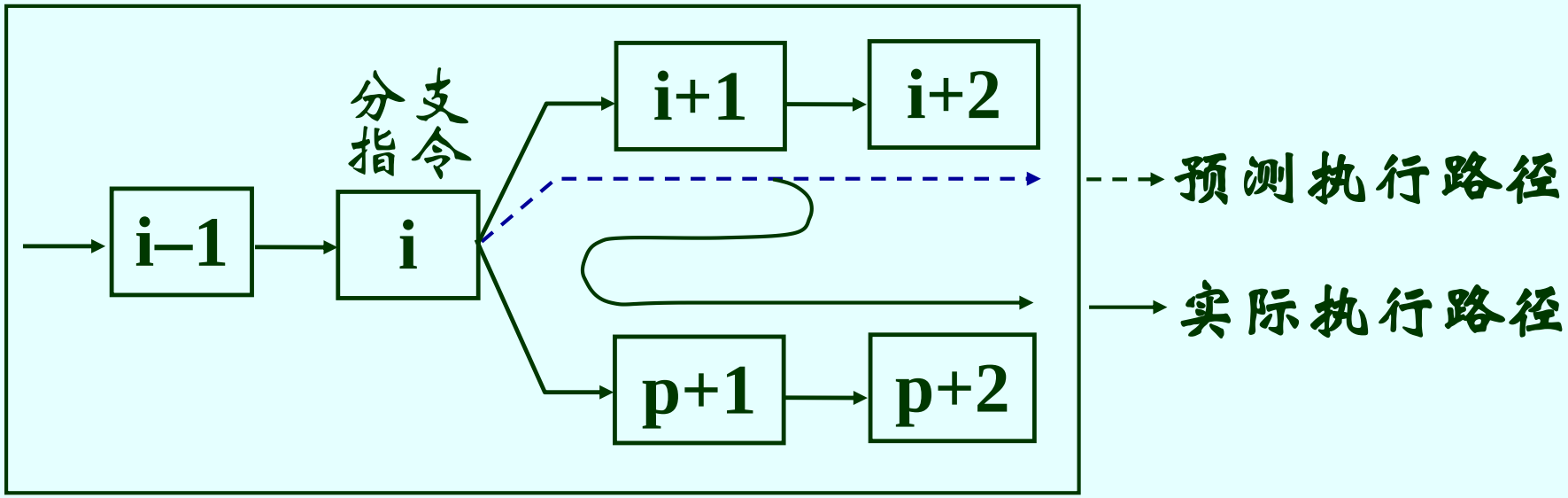
为此, 需解决以下三个问题:

(1) 如何记录分支的历史信息, 要记录哪些信息?

(2) 如何根据这些信息来预测分支的方向, 甚至提前取出分支目标指令?

(3) 分支预测错误时, 废除已经预取和分析的指令, 恢复现场, 并从另一路分支重新取指令。

如下图所示:



显然, 为了能恢复现场, 需要在执行预测目标指令之前将现场保存起来。

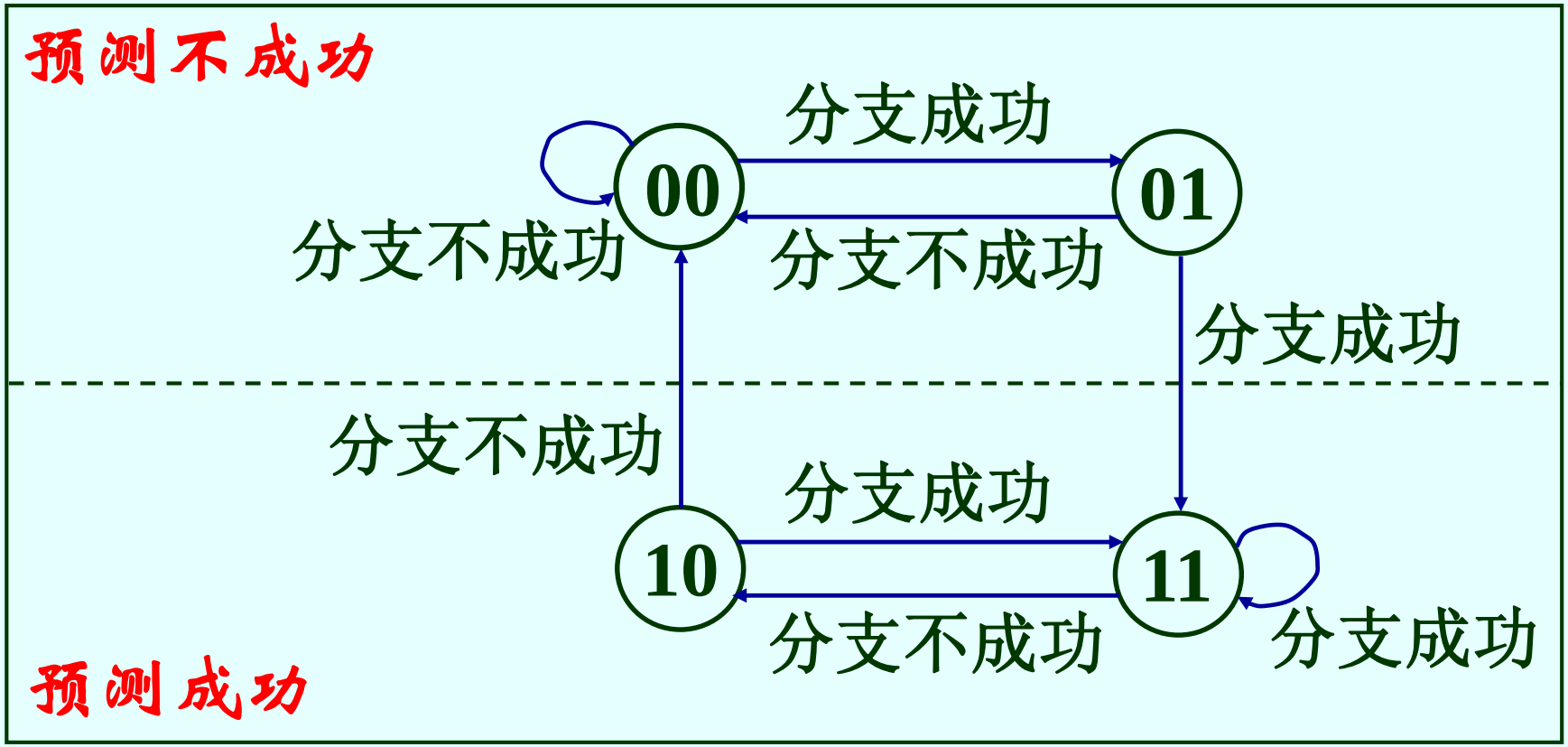
1、分支历史表BHT

采用Branch History Table(BHT)来记录分支指令的历史, 并按该历史情况来进行预测。

所谓“历史”是指最近一次或几次的执行是成功还是失败。如果只记录最近一次的历史, 则BHT中只需要1个二进制位(最简单的一种方式)。

为提高预测精度, 常采用两位二进制来记录历史, 实际测试表明, 采用更多的位与采用两位的预测精度差异不大。

两位分支预测的状态转换图如下:



在00和01状态, 预测不成功(预测不分支), 在10和11状态, 预测成功。线条边的文字说明是指分支指令的实际执行情况。

两位预测操作有两步: ① 分支预测; ② 修改状态

当分支指令到达译码阶段时, 根据从BHT中读出的信息进行预测:

(1) 从BHT中读出的数据为“00”或“01”, 按“分支不成功”处理, 继续分支指令的下一条指令。当分支指令的实际执行结果出来后, 如果发现预测正确, 继续执行指令, 否则, 作废已经预取的和分析的指令, 恢复现场, 并从分支路径重新取指令执行。

对状态的修改操作是: 在原状态为“00”情况下, 如果预测不正确, 则将状态改为“01”; 否则, 表明预测正确, 状态不变; 若原状态为“01”, 如果预测不正确, 则将状态改为“11”; 否则, 就是预测正确, 则将状态改为“00”。

(2) 从BHT中读出的数据为“10”或“11”，按“分支成功”处理处理，即从成功分支路取指令进行处理。待分支指令的实际执行结果出来后，如果预测正确，继续执行指令，否则，作废已经预取的和分析的指令，恢复现场，并从分支路径重新取指令执行。

对状态的修改操作是:若原状态为“11”，如果预测不正确，则将状态改为“10”；否则，表明预测正确，状态不变；若原状态为“10”，如果预测不正确，则将状态改为“00”；否则，预测正确，则将状态改为“11”

BHT可以放在指令高速缓存中，也可以用专门的硬件来实现。

BHT仅仅是预测分支是否成功, 而对分支目标地址不提供支持, 还需要另外的时间来计算目标地址, 因此, 如果确定分支目标地址的时间比较长时(大于或远大于预测时间), BHT方法就没有什么意义。

2、采用分支目标缓冲器BTB

BTB是一个由硬件实现的专用表格。表格中每一项至少由两个字段构成:

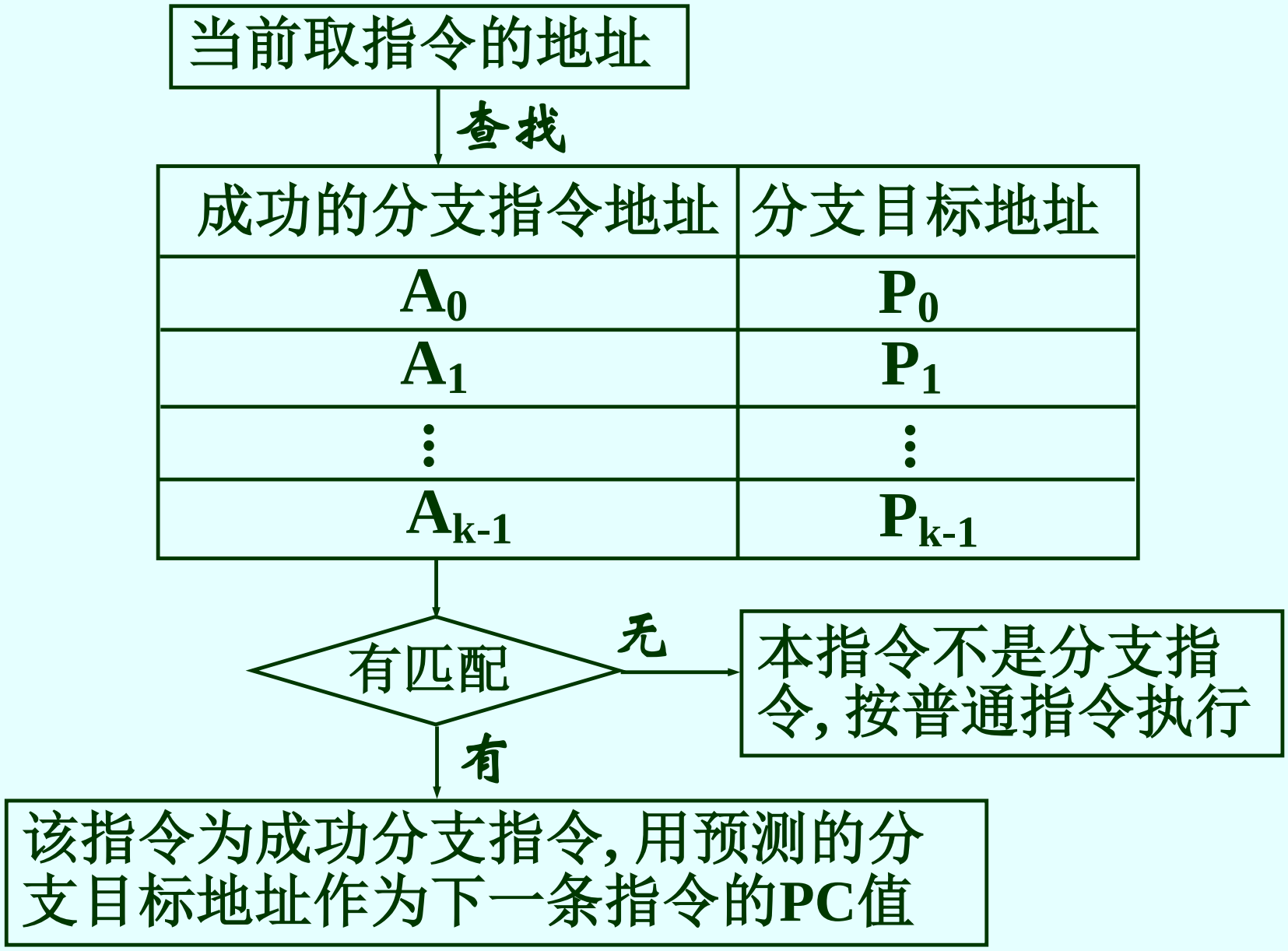
① 近期已执行过的(部分)成功分支指令的地址

② 预测的分支目标地址

在每次取指令的同时, 用该指令的地址与BTB中的所有项目的第一个字段进行比较, 如果有匹配的, 意味着该指令是分支指令并且上一次执行分支是成功的, 据此预测本次执行也将分支成功, 分支地址由该项的第二个字段给出;

如果没有匹配的, 就把当前指令当作普通指令(不是分支指令)来执行。

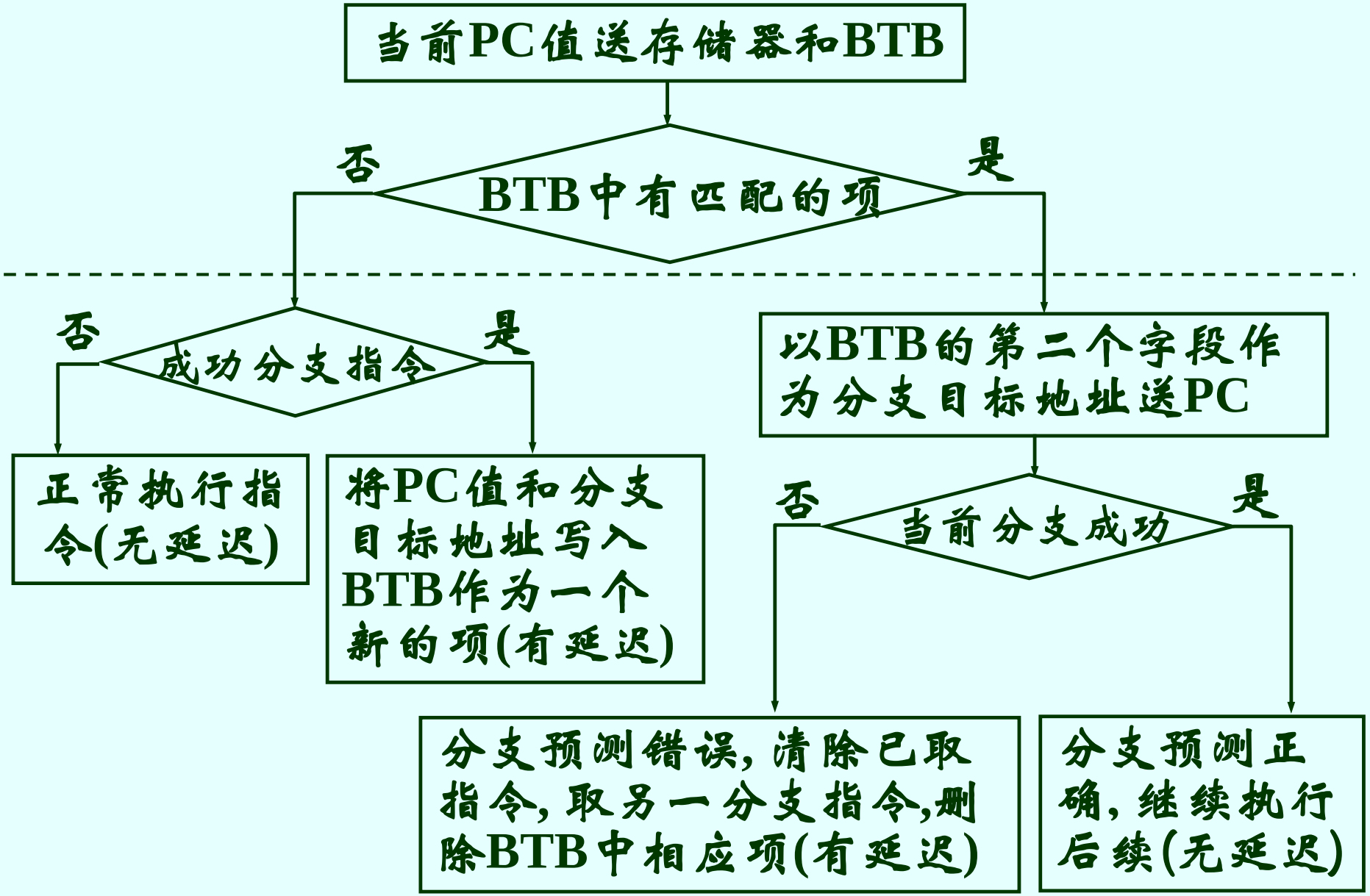
如下图所示:


当前取指令的地址
查找
成功的分支指令地址
分支目标地址
A_0
P_0
A_1
P_1
A_{k-1}
P_{k-1}
有匹配
无
本指令不是分支指令，按普通指令执行
有
该指令为成功分支指令，用预测的分支目标地址作为下一条指令的PC值

由于BTB中存放的是执行过的成功分支指令的地址, 所以如果当前指令的地址与BTB中的第一个字段匹配, 那么就将该匹配项中第二个字段中的地址送往PC, 从分支目标处开始取指令。

如果预测正确, 则不会产生任何分支延迟; 如果预测错误, 或者在BTB中没有匹配的项, 则至少有两个时钟周期的延迟开销 (更新BTB中的项, 在更新期间停止取指令)。

采用BTB时所进行的处理步骤:


当前PC值送存储器和BTB
BTB中有匹配的项
否
是
成功分支指令
否
是
正常执行指令(无延迟)
将PC值和分支目标地址写入BTB作为一个新的项(有延迟)
以BTB的第二个字段作为分支目标地址送PC
当前分支成功
否
是
分支预测错误,清除已取指令,取另一分支指令,删除BTB中相应项(有延迟)
分支预测正确,继续执行后续(无延迟)

BTB的另一种形式是在分支目标缓冲器中增加一个“分支历史表”(也称“转移历史表”) 字段, 存放过去所有执行过的分支指令的转移情况(无论转移成功与否), 该字段用于转移预测(功能类似于签前述的BHT)。

该方法实际上是BTB与BHT相结合

当前预取的指令

↓ 查找

分支指令地址	分支历史表	分支目标地址
A_0	T_0	P_0
A_1	T_1	P_1
⋮	⋮	⋮
A_{k-1}	T_{k-1}	P_{k-1}

第二节 向量处理技术

向量处理机:

具有向量数据表示和向量指令的流水线处理机。

例: 向量计算: D = A×(B + C)

其中A、B、C、D是长度为 N 的向量。

$$A=\begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_N \end{pmatrix} \quad B=\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{pmatrix} \quad C=\begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_N \end{pmatrix} \quad D=\begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_N \end{pmatrix}$$

可以有以下几种处理方式:

1. 水平(横向)处理方式

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

$$\begin{pmatrix} a_1 \times (b_1 + c_1) \\ a_2 \times (b_2 + c_2) \\ \vdots \\ a_N \times (b_2 + c_2) \end{pmatrix} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_N \end{pmatrix} \times \left(\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{pmatrix} + \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_N \end{pmatrix} \right)$$

水平处理方式即为逐个求d[i]:

先计算 d[1]=a[1]×(b[1]+c[1]);

再计算 d[2]=a[2]×(b[2]+c[2]); …,

最后计算 d[N]=a[N]×(b[N]+c[N])。

编程时采用循环程序结构。在每次循环中, 至少要用到如下几条指令:

计算d[i]=a[i]×(b[i]+c[i])

……

$k_i = b_i + c_i$

$d_i = k_i \times a_i$

……

BE(等于“ 0”分支成功)\

程序计算需N次循环, 其中N–1次分支成功, 在每次循环中有一次数据相关。如果用多功能静态流水线, 则要进行2次乘和加的功能转换, 所以共出现N次数据相关和 2N 次功能切换。

因此, 水平处理方式不适合对向量进行流水处理。

2. 垂直(纵向)处理方式

向量的所有分量相同运算处理完后, 再执行别的运算。

$$\begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_N \end{pmatrix} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_N \end{pmatrix} \times \left(\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{pmatrix} + \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_N \end{pmatrix} \right) \quad \text{令: } K = \begin{pmatrix} (b_1 + c_1) \\ (b_2 + c_2) \\ \vdots \\ (b_2 + c_2) \end{pmatrix}$$

则有:

$$D = \begin{pmatrix} a_1 \times (b_1 + c_1) \\ a_2 \times (b_2 + c_2) \\ \vdots \\ a_N \times (b_N + c_N) \end{pmatrix} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_N \end{pmatrix} \times \begin{pmatrix} (b_1 + c_1) \\ (b_2 + c_2) \\ \vdots \\ (b_N + c_N) \end{pmatrix}$$

即: $K = B + C$, $D = K \times A$

只需两条向量指令, 处理过程无分支指令, 无数据相关, 两条向量指令间仅有一次数据相关。仍用静态流水线, 也只需1次功能切换, 所以适合于对向量进行流水处理

3. 分组(纵横)处理方式

分组处理方式是把长度为N的向量, 分成若干组, 每组长度为n , 组内按纵向方式处理, 依次处理各组。

$$
\begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_{n-1} \end{pmatrix} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix} \times \left(\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n-1} \end{pmatrix} + \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{pmatrix} \right) \quad \text{第1组}
$$

$$
\begin{pmatrix} d_n \\ d_{n-1} \\ \vdots \\ d_{2n-1} \end{pmatrix} = \begin{pmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_{2n-1} \end{pmatrix} \times \left(\begin{pmatrix} b_n \\ b_{n+1} \\ \vdots \\ b_{2n-1} \end{pmatrix} + \begin{pmatrix} c_n \\ c_{n+1} \\ \vdots \\ c_{2n-1} \end{pmatrix} \right) \quad \text{第2组}
$$

$$
\begin{pmatrix} d_{2n} \\ d_{2n+1} \\ \vdots \\ d_{3n-1} \end{pmatrix} = \begin{pmatrix} a_{2n} \\ a_{2n+1} \\ \vdots \\ a_{3n-1} \end{pmatrix} \times \left(\begin{pmatrix} b_{2n} \\ b_{2n+1} \\ \vdots \\ b_{3n-1} \end{pmatrix} + \begin{pmatrix} c_{2n} \\ c_{2n+1} \\ \vdots \\ c_{3n-1} \end{pmatrix} \right) \quad \text{第3组}
$$

$$
\begin{pmatrix} \vdots \end{pmatrix} = \begin{pmatrix} \vdots \end{pmatrix} \times \left(\begin{pmatrix} \vdots \end{pmatrix} + \begin{pmatrix} \vdots \end{pmatrix} \right)
$$

一个典型的向量求解例: $Y=a\times X+Y$

其中X和Y是向量, a是一个标量(常量)。

(1) 如果采用非向量方式, 则指令必须对向量中各元素进行一次乘、加和存储操作。每一次循环, 必须对向量中元素位置的下标变量进行增量, 并判断循环是否结束。假设向量长度为64, 每个元素8个字节, 向量首地址分别在R_X和R_Y中, 采用双精度运算(即一次能处理8个字节)。

程序段如下:

Y=a*X + Y

```
        LD   F0 , a              ; a装入F0
        ADDI R4 , RX , #512      ; 元素末地址→R4(各元素8个字节)
LOOP:   LD   F2 , 0(RX)          ; 取向量元素X(i)
        MULD F2 , F0 , F2        ; a与X(i)相乘→F2
        LD   F4 , 0(RY)          ; 取向量元素Y(i)
        ADDD F4 , F2 , F4        ; aX(i)与Y(i)相加→F4
        ST   0(RY), F4           ; 存结果向量元素
        ADDI RX , RX , #8        ; 增量向量元素X下标
        ADDI RY , RY , #8        ; 增量向量元素Y下标
        SUB  R20 , R4 , RX       ; R4−RX→R20 , 计算是否达限界
        BNZ  R20 , LOOP          ; 若循环未结束, 转到LOOP
```

共执行的指令数: 9×64+2=578 条

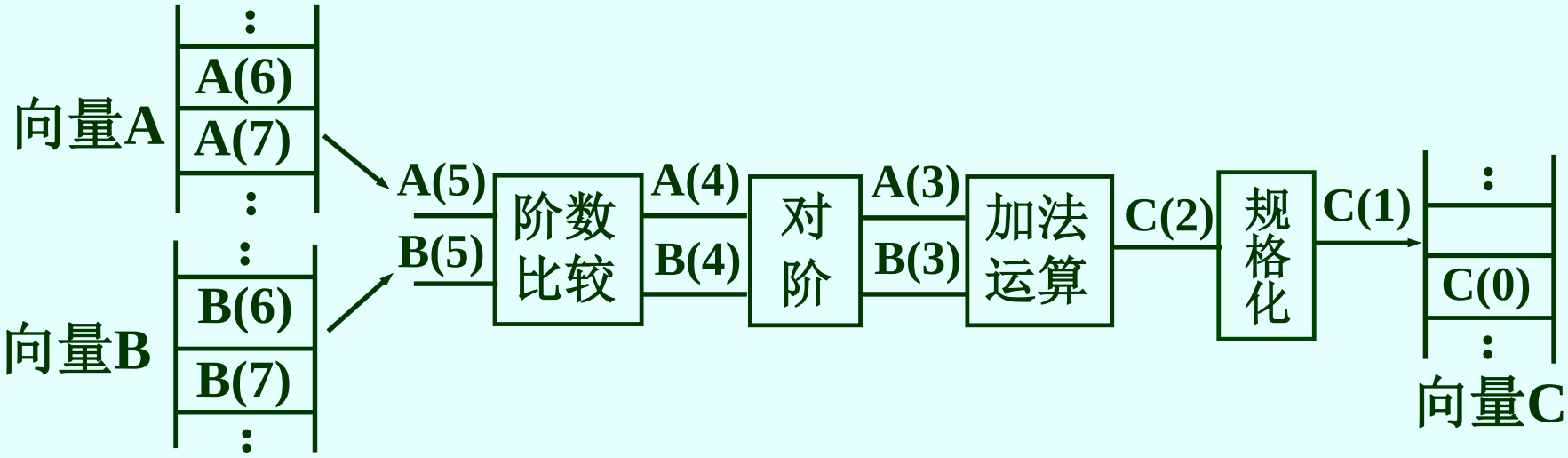
(2) 如果采用向量机来完成同样的操作, 则有:

```
LD      F0 , a          ; a装入F0
LV      V1 , RX         ; 装入向量X, LV为向量取指令
MULV    V2 , F0 , V1    ; a与X(i)相乘
LV      V3 , RY         ; 装入向量Y
ADDV    V4 , V2 , V3    ; 向量加aX+Y
SV      RY , V4         ; 存结果向量元素
```

向量机只需要执行6条指令, 大大降低了对指令带宽的要求, 这是因为向量指令是对64个元素进行操作, 而且没有标量循环中对元素下标量的增量和判断循环是否结束的后4条指令。

注: 向量处理与指令流水线处理的异同:

向量处理属于“运算流水线”类型, 即, 设置几个专用的运算单元, 对数据进行流水线作业处理, 实现对数据的并行处理, (以浮点数处理为例):



指令流水线是将指令执行过程分解成如干个阶段。各阶段的处理分别由专用的硬件来承担, 以达到并行处理的目的。