上章回顾

1. 有哪些相关（两条指令之间存在某种依赖关系）？
2. 有哪几种冲突？
3. 如何减少分支延迟？
4. 如何调度才能使流水线获得最高效率？
5. 第三章讲的流水线有哪些局限性？

1、尽早判断(猜测)分支是否成功
2、尽早知道分支目标地址

预测分支失败
预测分支成功
分支延迟(槽)

In-order

数据相关、名相关、控制相关

结构冲突、数据冲突、控制冲突

非线性单功能流水线调度

并行的类型

1. **指令级并行(ILP)**
   - 以并行方式执行某个指令流中的独立无关的指令 (pipelining, superscalar, VLIW)
2. **线程级并行 (TLP)**
   - 以并行方式执行多个独立的指令流 (multithreading, multiple cores)
3. **数据级并行(DLP)**
   - 以并行方式执行多个相同类型的操作 (vector/SIMD execution)

指令级并行

- 指令级并行：指令之间存在的一种并行性，利用它，计算机可以并行执行两条或两条以上的指令

  （ILP：Instruction-Level Parallelism）

- 开发ILP的途径有两种
  - **资源重复，重复设置多个处理部件，让它们同时执行相邻或相近的多条指令；**
  - **采用流水线技术，使指令重叠并行执行**
- 本章研究：如何利用各种技术来开发更多的指令级并行（硬件的方法）

第5章 指令级并行及其开发——硬件方法

本章作业

5.8

5.9

5.11

5.1 指令级并行的概念

1. 开发ILP的方法可以分为两大类
   - 基于软件的**静态**开发方法
   - 基于硬件的**动态**开发方法
2. 流水线处理机的实际CPI
   - 理想流水线的CPI加上各类停顿的时钟周期数：

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

Pipeline CPI = Ideal pipeline CPI + Structural stalls + Data hazard stalls + Control stalls

   - 理想CPI是衡量流水线最高性能的一个指标
   - IPC：Instructions Per Cycle

（每个时钟周期完成的指令条数）

5.1 指令级并行的概念

3. 基本程序块

- 基本程序块：一串连续的代码除了入口和出口以外，没有其他的分支指令和转入点
- 程序平均每4～7条指令就会有一个分支

4. 循环级并行：使一个循环中的不同循环体并行执行

- 开发循环的不同叠代之间存在的并行性
  - **最常见、最基本**

循环实例：

```
for （i=1； i<=500； i=i＋1）
    a[i]=a[i]＋s；
```

- 每一次循环都可以与其它的循环重叠并行执行
- 在每一次循环的内部，却没有任何的并行性

5.3 指令的动态调度

- 静态调度

C语言里if(likely (x == 0))/ if(unlikely (x == 0))
编译时的静态预测优化

  - 依靠编译器对代码进行静态调度，以减少相关和冲突
  - 它不是在程序执行的过程中、而是在编译期间进行代码调度和优化
  - 通过把相关的指令拉开距离来减少可能产生的停顿

- 动态调度

  - 在程序的执行过程中，依靠专门硬件对代码进行调度，减少数据相关导致的停顿

5.3 指令的动态调度

5.3.1 动态调度的基本思想

1. 到目前为止,我们所使用流水线的最大的局限性:

- 指令是按序流出和按序执行的
- 考虑下面一段代码:

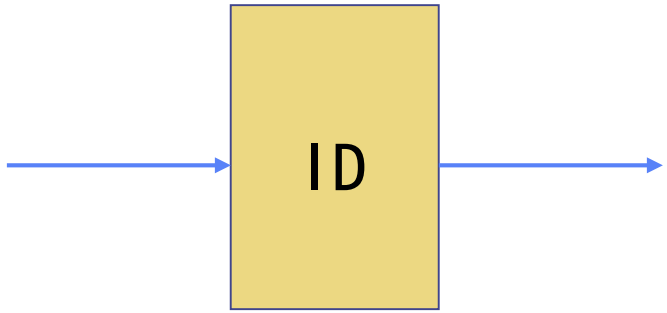
```
LD    F4, 0 (F2)
ADD   F10, F4, F6
SUB   F12, F6, F14
```

ADD指令与DIV指令关于F4相关，导致流水线停顿

SUB指令与流水线中的任何指令都没有关系，但也因此受阻

5.3 指令的动态调度

在前面的基本流水线中：



检测结构冲突

检测数据冲突

一旦一条指令受阻，其后的指令都将停顿

改进：把指令流出的工作拆分为两步

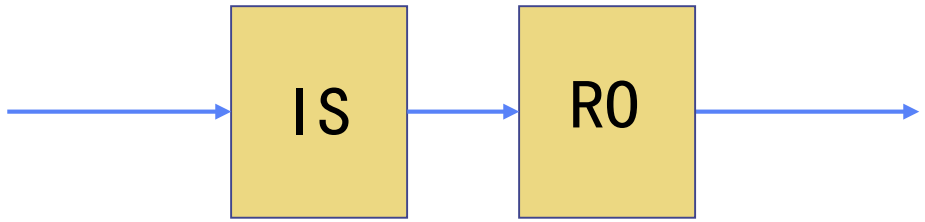
- 检测结构冲突
- 等待数据冲突消失

只要检测到没有结构冲突，就可以让指令流出；并且流出后的指令一旦其操作数就绪就可以立即执行

5.3 指令的动态调度

2. 乱序执行
   - 指令的执行顺序与程序顺序不相同
   - 指令的完成也是乱序完成的：即指令的完成顺序与程序顺序不相同

3. 为了支持乱序执行，将5段流水线的译码阶段再分为两个阶段：
   - 流出(Issue，IS)：指令译码，检查是否存在结构冲突(in-order issue)
   - 读操作数(Read Operands，RO)：等待数据冲突消失，然后读操作数(Out of order execution)



检测**结构**冲突　检测**数据**冲突

5.3 指令的动态调度

4. 在前述5段流水线中，是不会发生WAR冲突和WAW冲突的，但乱序执行就使得它们可能发生了

- 例如，考虑下面的代码

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

存在输出相关：DIV.D 与 ADD.D

存在反相关：ADD.D 与 SUB.D

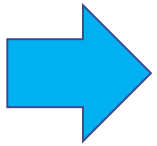
5.3 指令的动态调度

5. 动态调度的流水线支持多条指令同时处于执行当中

- 要求：具有多个功能部件、或者功能部件流水化、或者兼而有之
- 我们假设具有多个功能部件

记分牌算法和Tomasulo算法(自学)

基本思想：集中信息进行调度，让没有冲突的指令尽早执行，如果某条指令被暂停，其后续不相关的指令仍可继续执行



5.4 动态分支预测技术

所开发的ILP越多，控制相关的制约就越大，分支预测要有更高的准确度

- 在n-流出的处理机中，遇到分支指令的可能性增加了n倍
- 要给处理器连续提供指令，就需要准确地预测分支

5.4 动态分支预测技术

动态分支预测：在程序运行时，根据分支指令过去的表现来预测其将来的行为

- **如果分支行为发生了变化，预测结果也跟着改变**

分支预测的有效性取决于：

- 预测的准确性
- 预测正确和不正确两种情况下的分支开销

决定分支开销的因素：

- **流水线的结构**
- **预测的方法**
- **预测错误时的恢复策略等**

5.4 动态分支预测技术

采用动态分支预测技术的目的

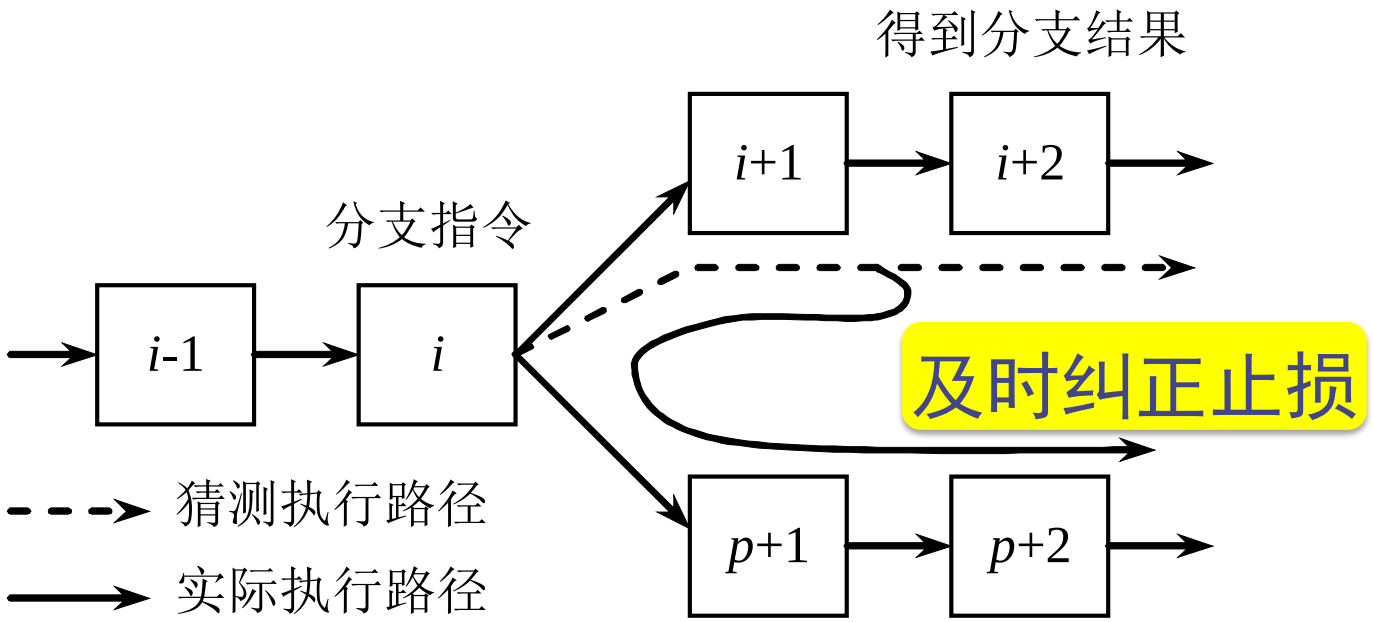
- 提高预测分支成功率
- 尽快找到分支目标地址（或指令）

需要解决的3个关键问题

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去向，甚至提前取出分支目标处的指令？

5.4 动态分支预测技术

3. 在预测错误时，要作废已经预取和分析的指令，恢复现场，并从另一条分支路径重新取指令



5.4 动态分支预测技术

5.4.1 采用分支历史表 BHT

1. 分支历史表BHT（Branch History Table）

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

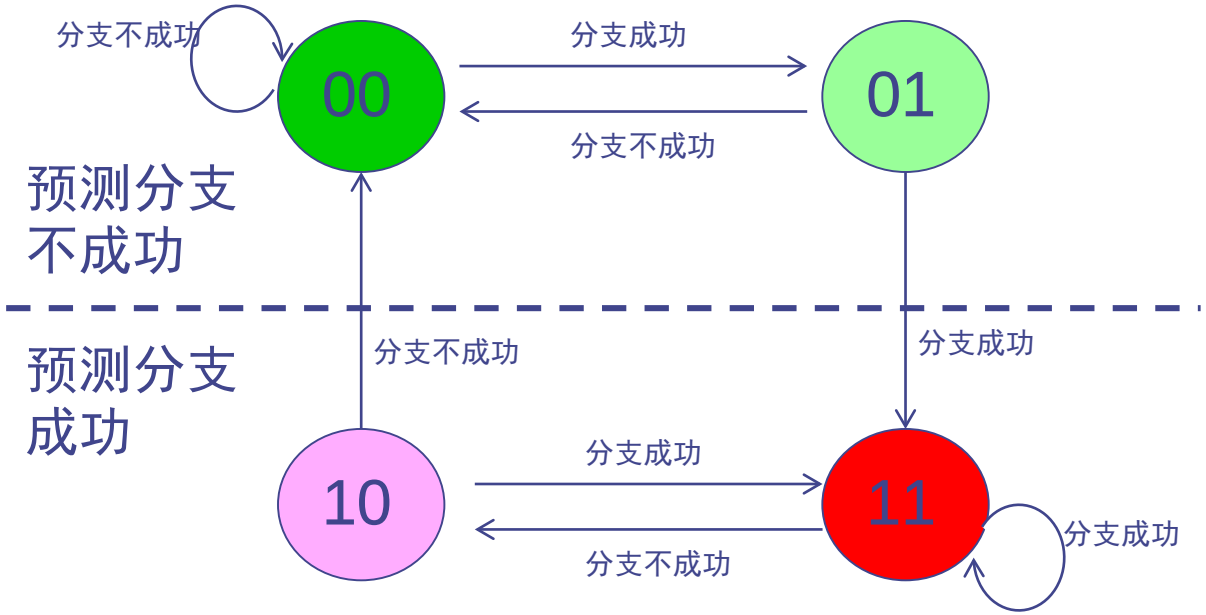
2. 只有1个预测位的分支预测

记录分支指令最近一次的历史，BHT中只需要1位二进制位

5.4 动态分支预测技术

3. 采用两位二进制位来记录历史

- 提高预测的准确度
- 研究结果表明：两位分支预测的性能与n位（n>2）分支预测的性能差不多
- 两位分支预测的状态转换：



连续两次预测错误才改变预测方向

前述5段流水中，在ID段判断分支计算目标地址，BHT也是在ID段进行预测，无法带来好处

5.4 动态分支预测技术

5.4.2 采用分支目标缓冲器BTB

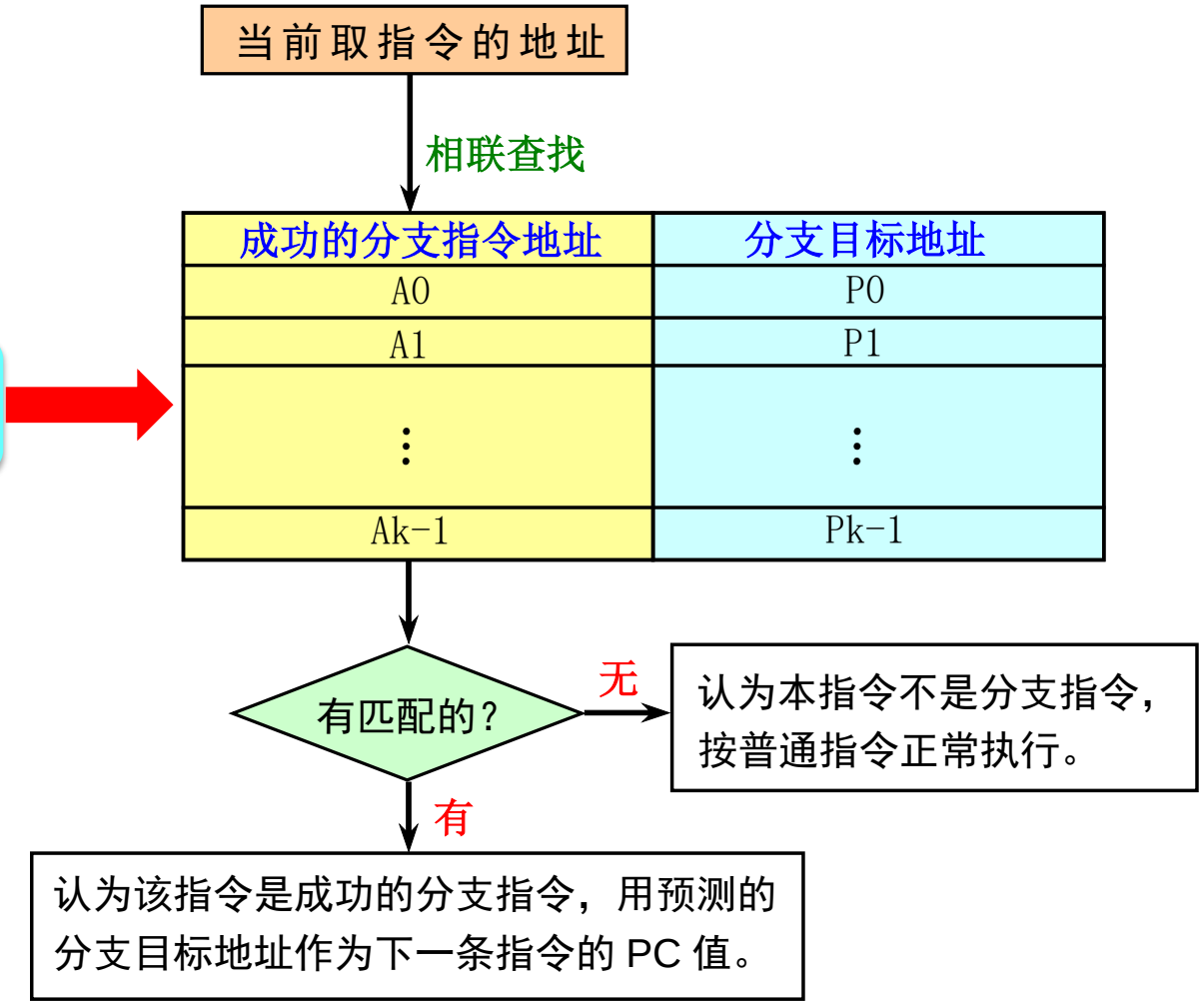
目标：将分支的开销降为 0

方法：分支目标缓冲

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

5.4 动态分支预测技术

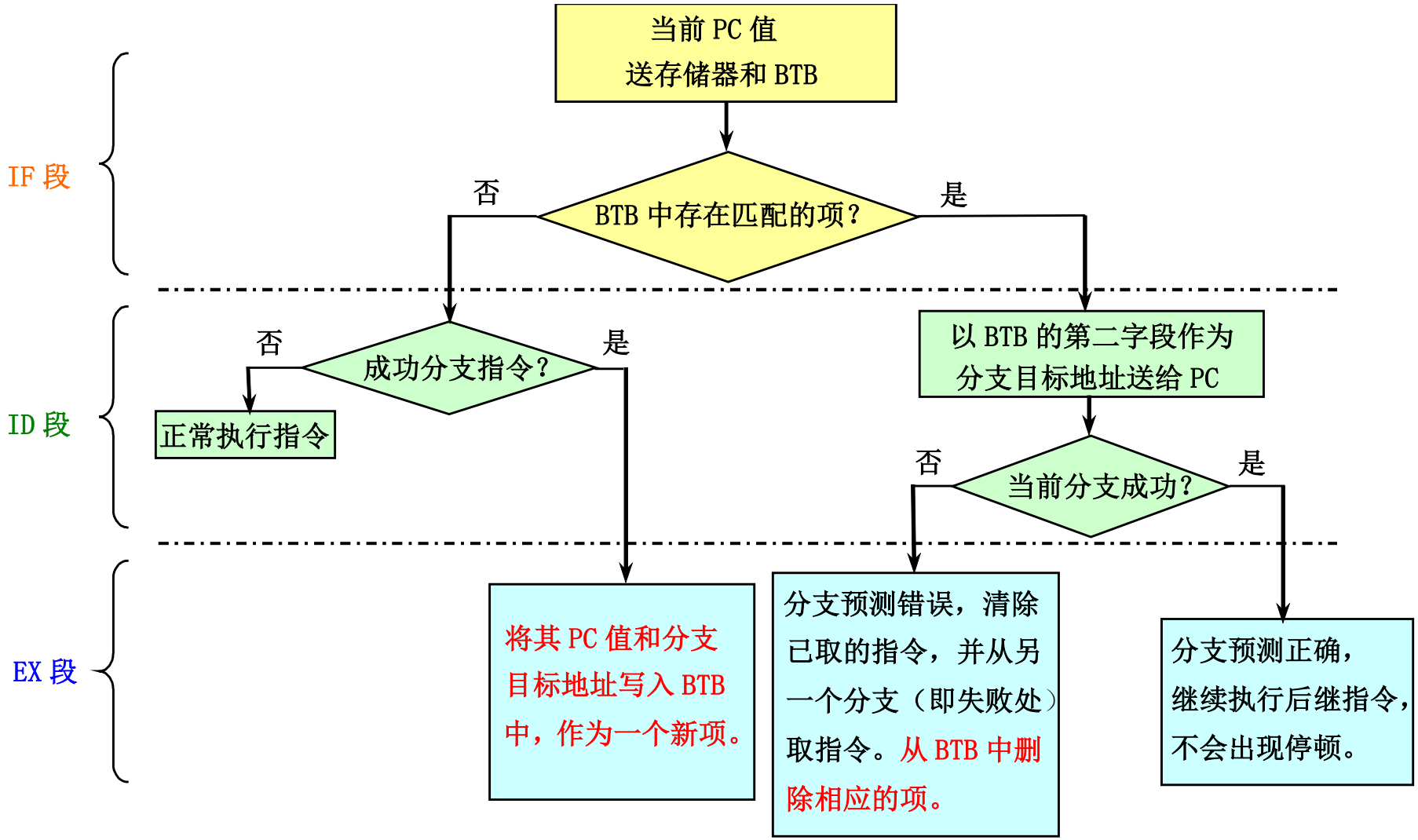
BTB的结构



用专门的硬件实现的一张表格，表格中的每一项至少有两个字段：

1. 执行过的成功分支指令的地址
2. 预测的分支目标地址

2. 采用BTB后，在流水线各个阶段所进行的相关操作



5.4 动态分支预测技术

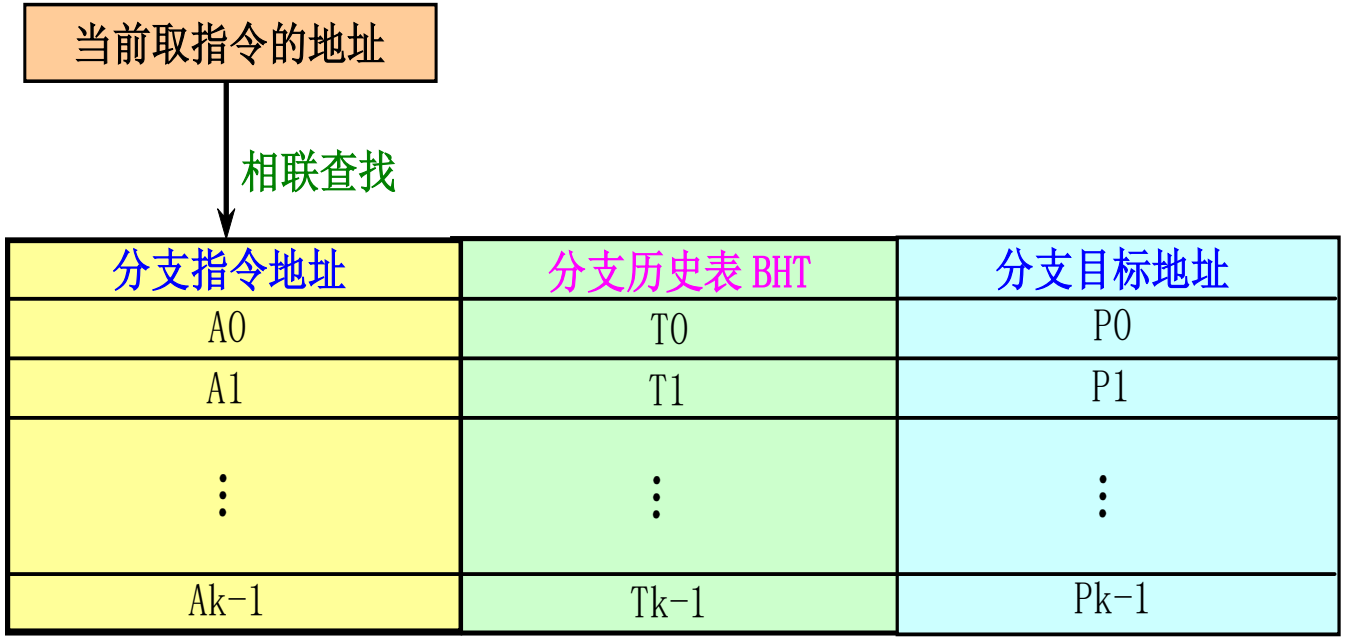
3. 采用BTB后，各种可能情况下的延迟

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

5.4 动态分支预测技术

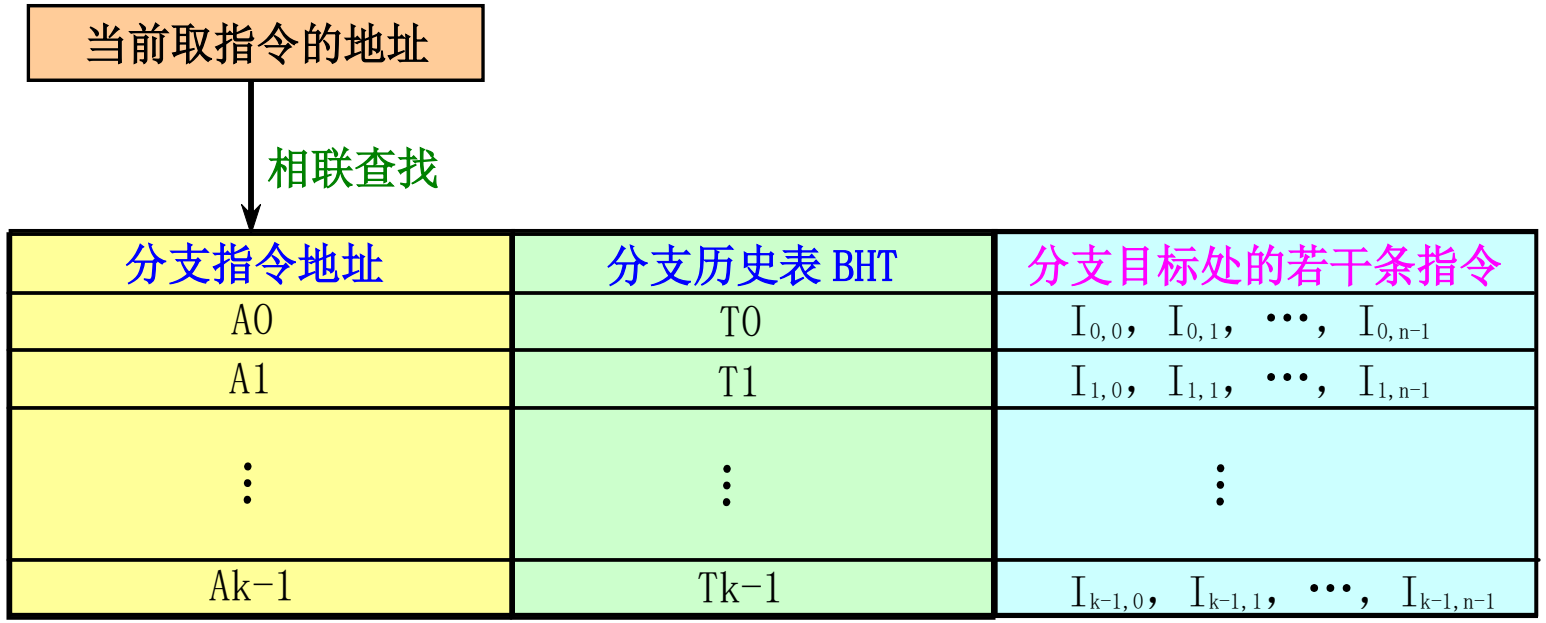
4. BTB的另一种形式

- 在分支目标缓冲器中增设一个至少是两位的“分支历史表”字段



5.4 动态分支预测技术

5. 更进一步，在表中对于每条分支指令都存放若干条分支目标处的指令，就形成了分支目标指令缓冲器



5.4 动态分支预测技术

例题：假设没有分支的程序理想CPI为1，实际分支指令比例为10%，现对条件转移指令使用分支目标缓冲进行优化。若分支预测错误的开销为5个时钟周期，缓冲不命中的开销为3个时钟周期，命中率为80%，预测精度为95%。则相对固定的2个时钟周期延迟的分支处理获得的加速比是多少？

解：

程序执行的实际CPI = 没有分支的理想CPI + 分支带来的额外开销

额外开销=10% * (80%命中 * 5%预测错误 * 5 + 20%没命中 * 3)

= 0.08

所以程序执行的实际CPI = 1 + 0.08 = 1.08。

采用固定的2 个时钟周期延迟的分支处理的CPI为：

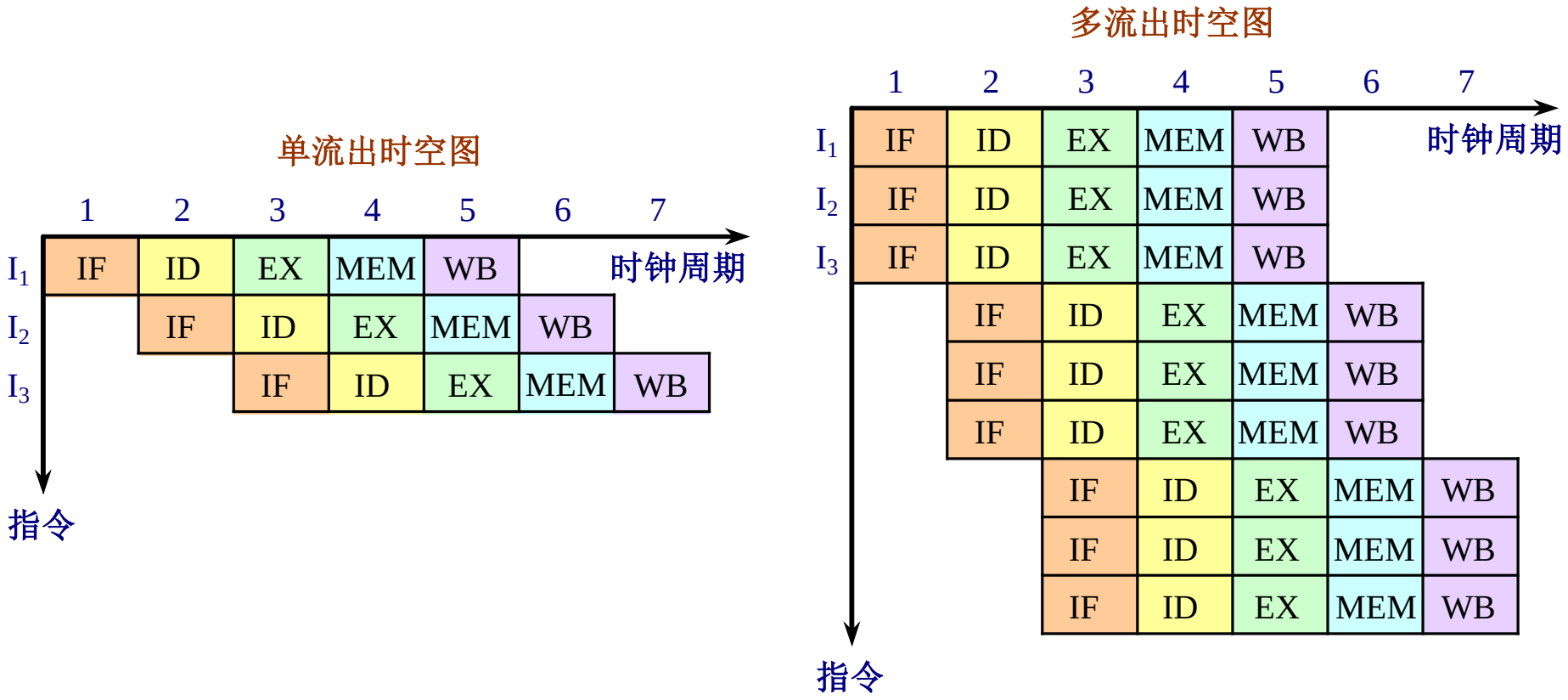
CPI = 1 + 10% * 2 = 1.2

由此可知该方法的加速比为：S = 1.2 / 1.08 = 1.11

5.5 多指令流出技术

- 在每个时钟周期内流出多条指令， CPI＜1
- 单流出和多流出处理机执行指令的时空图对比

5.5 多指令流出技术



单流出和多流出处理机执行指令的时空图

5.5 多指令流出技术

1. 多流出处理机有两种基本风格：

- 超标量（Superscalar）
  - 在每个时钟周期流出的指令条数不固定，依代码的具体情况而定（有个上限）
  - 设这个上限为n，就称该处理机为n-流出
  - 可以通过编译器进行静态调度，也可以使用硬件动态调度
- 超长指令字VLIW（Very Long Instruction Word）
  - 在每个时钟周期流出的指令条数是固定的，这些指令构成一条长指令或者一个指令包
  - 指令包中，指令之间的并行性是通过指令显式地表示出来的
  - 指令调度是由编译器静态完成的

5.5 多指令流出技术

5.5.1 基于静态调度的多流出技术

- 在典型的超标量处理器中，每个时钟周期可流出1到8条指令
- 指令按序流出，在流出时进行冲突检测

由硬件检测当前流出的指令之间是否存在冲突以及当前流出的指令与正在执行的指令是否有冲突

举例：一个4-流出的静态调度超标量处理机

- 在取指令阶段，流水线将从取指令部件收到1～4条指令（称为流出包）
  - **在一个时钟周期内，这些指令有可能是全部都能流出，也可能是只有一部分能流出**

5.5 多指令流出技术

- 流出部件检测结构冲突或者数据冲突

  一般分两阶段实现：

  - **第一段：进行流出包内的冲突检测，选出初步判定可以流出的指令**
  - **第二段：检测所选出的指令与正在执行的指令是否有冲突**

MIPS处理机是怎样实现超标量的呢？

假设：每个时钟周期流出两条指令：

1条整数型指令＋1条浮点操作指令

- **其中：把load指令、store指令、分支指令归类为整数型指令**

5.5 多指令流出技术

1. 要求：

   同时取两条指令（64位），译码两条指令（64位）

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

3. 双流出超标量流水线中指令执行的时空图
   - 假设：所有的浮点指令都是加法指令，其执行时间为两个时钟周期
   - 为简单起见，图中总是把整数指令放在浮点指令的前面

5.5 多指令流出技术

指令类型	流水线工作情况							
整数指令	IF	ID	EX	MEM	WB			
浮点指令	IF	ID	EX	EX	MEM	WB		
整数指令		IF	ID	EX	MEM	WB		
浮点指令		IF	ID	EX	EX	MEM	WB	
整数指令			IF	ID	EX	MEM	WB	
浮点指令			IF	ID	EX	EX	MEM	WB
整数指令				IF	ID	EX	MEM	WB
浮点指令				IF	ID	EX	EX	MEM

5.5 多指令流出技术

4. 采用“1条整数型指令＋1条浮点指令”并行流出的方式，需要增加的硬件很少
5. 浮点load或浮点store指令将使用整数部件，会增加对浮点寄存器的访问冲突

   增设一个浮点寄存器的读/写端口
6. 由于流水线中的指令多了一倍，定向路径也要增加

5.5 多指令流出技术

7. 限制超标量流水线的性能发挥的障碍

- load指令
  - load后续3条指令都不能使用其结果，否则就会引起停顿（教材P79）
- 分支延迟 （教材P80）
  - 如果分支指令是流出包中的第一条指令，则其延迟是3条指令
  - 否则就是流出包中的第二条指令，其延迟就是2条指令

5.5 多指令流出技术

5.5.3 超长指令字技术

1. 把能并行执行的多条指令组装成一条很长的指令

   （100多位到几百位）

2. 设置多个功能部件

3. 指令字被分割成一些字段，每个字段称为一个操作槽，直接独立地控制一个功能部件

4. 在VLIW处理机中，在指令流出时不需要进行复杂的冲突检测，而是依靠编译器全部安排好了

5.5 多指令流出技术

5. VLIW存在的一些问题

- 程序代码长度增加了
  - 提高并行性而进行的大量的循环展开；
  - 指令字中的操作槽并非总能填满。

  解决：采用指令共享立即数字段的方法，或者采用指令压缩存储、调入Cache或译码时展开的方法。
- 采用了锁步机制

  任何一个操作部件出现停顿时，整个处理机都要停顿。
- 机器代码的不兼容性

5.5 多指令流出技术

5.5.4 多流出处理器受到的限制

指令多流出处理器受哪些因素的限制呢？

主要受以下3个方面的影响：

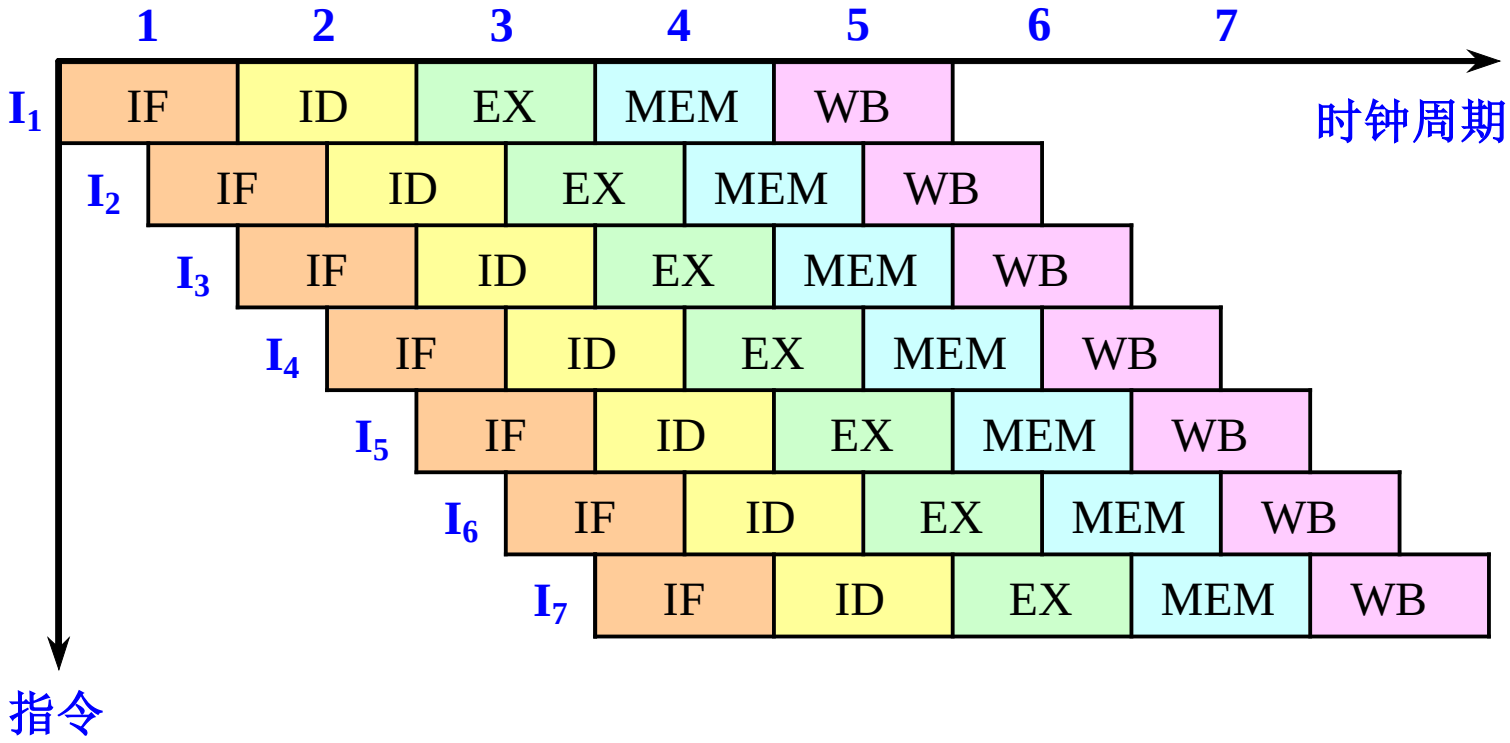
- **程序所固有的指令级并行性**
- **硬件实现上的困难**
- **超标量和超长指令字处理器固有的技术限制**

5.5 多指令流出技术

5.5.5 超流水线处理机

1. 将每个流水段进一步细分，这样在**一个时钟周期内能够分时流出多条指令**。这种处理机称为超流水线处理机
2. 对于一台每个时钟周期能流出n条指令的超流水线计算机来说，这n条指令不是同时流出的，而是每隔1/n个时钟周期流出一条指令
   - 实际上该超流水线计算机的流水线周期为1/n个时钟周期
3. 一台每个时钟周期分时流出两条指令的超流水线计算机的时空图

5.5 多指令流出技术



5.5 多指令流出技术

4. 在有的资料上，把指令流水线级数为8或8以上的流水线处理机称为超流水线处理机

5. 典型的超流水线处理器：SGI公司的MIPS系列R4000

- R4000微处理器芯片内有2个Cache：
  - 指令Cache和数据Cache
  - 容量都是8KB
  - 每个Cache的数据宽度为64位
- R4000的核心处理部件：整数部件
  - 一个32×32位的通用寄存器组
  - 一个算术逻辑部件（ALU）
  - 一个专用的乘法/除法部件
- 浮点部件
  - 一个执行部件
    - 浮点乘法部件
    - 浮点除法部件
    - 浮点加法/转换/求平方根部件（它们可以并行工作）
  - 一个16×64位的浮点通用寄存器组。浮点通用寄存器组也可以设置成32个32位的浮点寄存器。

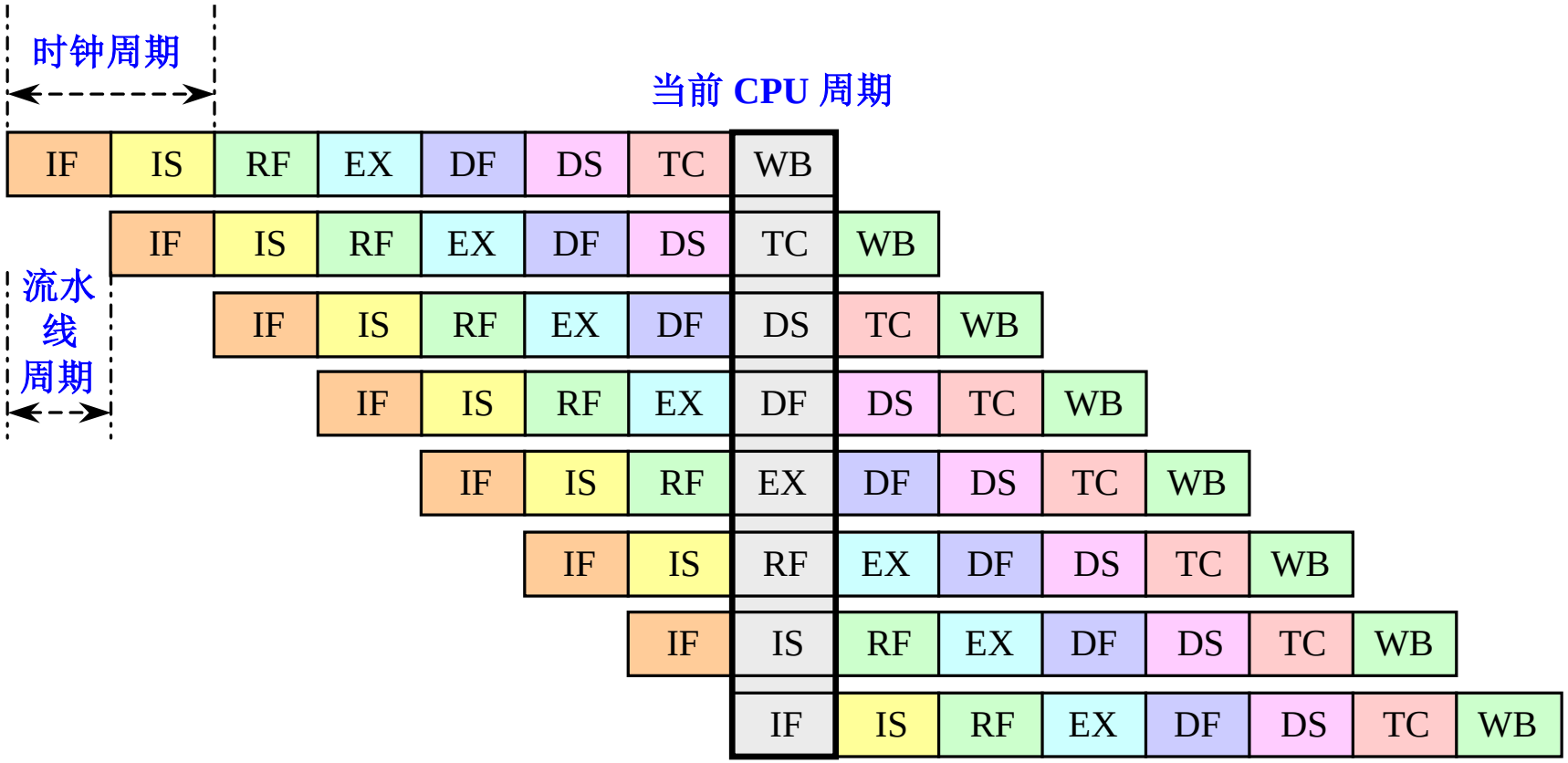
5.5 多指令流出技术

➢ 各级的功能

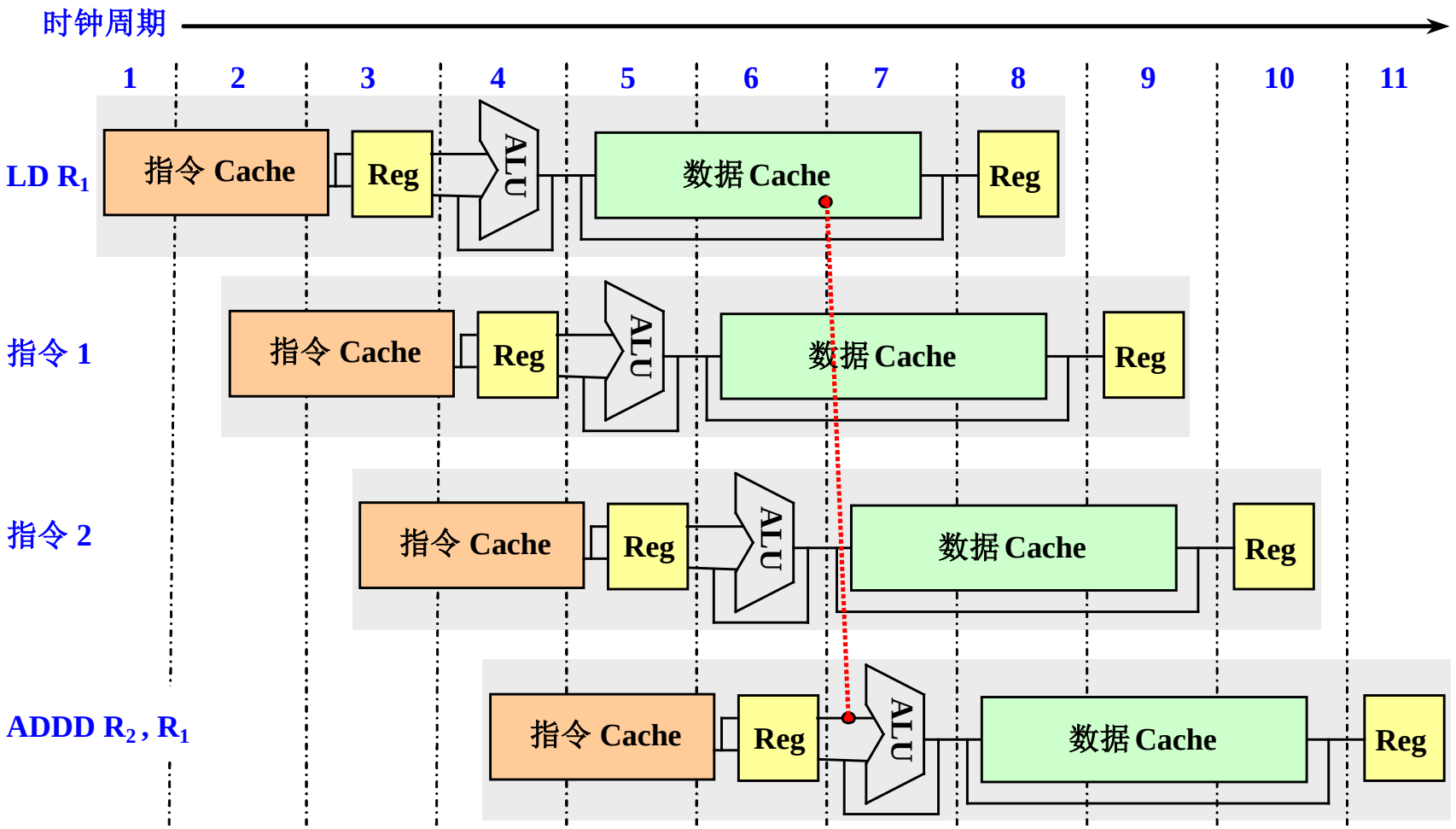
- IF：取指令的前半步，根据PC值去启动对指令Cache的访问。
- IS：取指令的后半步，在这一级完成对指令Cache的访问。
- RF：指令译码，访问寄存器组读取操作数，冲突检测，并判断指令Cache是否命中。
- EX：指令执行。包括：有效地址计算，ALU操作，分支目标地址计算，条件码测试。
- DF：取数据的前半步，启动对数据Cache的访问。
- DS：取数据的后半步，在这一级完成对数据Cache的访问。
- TC：标识比较，判断对数据Cache的访问是否命中。
- WB：load指令或运算型指令把结果写回寄存器组。

5.5 多指令流出技术

➢ MIPS R4000指令流水线时空图



➢ 载入延迟为两个时钟周期



并行的类型

1. 指令级并行(ILP)

- 以并行方式执行某个指令流中的独立无关的指令
  (pipelining, superscalar, VLIW)

2. 线程级并行 (TLP)

- 以并行方式执行多个独立的指令流
  (multithreading, multiple cores)

3. 数据级并行(DLP)

- 以并行方式执行多个相同类型的操作
  (vector/SIMD execution)

DLP的兴起

- 应用需求和技术约束推动着体系结构的发展和选择

  军事研究领域（核武器研制、密码学）、科学研究、天气预报、石油勘探、工业设计 (car crash simulation)、 生物信息学、密码学

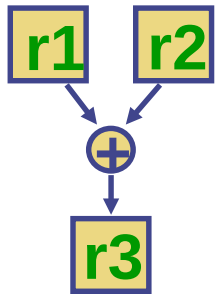
- 图形、机器视觉、语音识别、机器学习等新的应用均需要大量的数值计算，其算法通常具有数据并行
- SIMD-based 结构 (vector-SIMD, subword-SIMD, SIMT/GPUs) 是执行这些算法的最有效途径

  70-80年代Supercomputer = Vector Machine

Alternative Model: Vector Processing

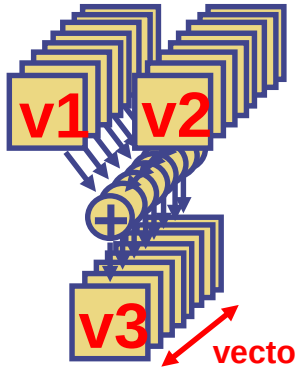
向量处理机具有更高层次的操作，一条向量指令可以处理N个或N对操作数（处理对象是向量）

SCALAR
(1 operation)




```
add r3, r1, r2
```

VECTOR
(N operations)




```
add.vv v3, v1, v2
```

向量处理实例

向量处理的C代码、标量代码和向量代码对比

```
# C 代码
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];
```

```
# 标量代码
  li x4, 64
loop:
  fld f1, 0(x1)
  fld f2, 0(x2)
  fadd.d f3,f1,f2
  fsd f3, 0(x3)
  addi x1, x1, 8
  addi x2, x2, 8
  addi x3, x3, 8
  subi x4, x4, 1
  bnez x4, loop
```

```
# 向量代码
  li x4, 64
  vsetvl x4
  vld v1, (x1)
  vld v2, (x2)
  vadd v3,v1,v2
  vst v3, (x3)
```

Graphics Processing Units (GPUs)

- 原来的GPU是指带有高性能浮点运算部件、可高效生成3D图形的具有固定功能的专用设备 (mid-late 1990s)
  - 让PC机具有类似工作站的图形功能
  - 用户可以配置图形处理流水线，但不是真正的对其编程
- 随着时间的推移，GPU加入了越来越多的可编程性 (2001-2005)
  - 例如新的语言 Cg可用来编写一些小的程序处理图形的顶点或像素，是Windows DirectX的变体
  - 大规模并行（针对每帧上百万顶点或像素）但非常受限于编程模型
- 有些用户注意到通过将输入和输出数据映射为图像，并对顶点或像素渲染计算 可进行通用计算
  - 因为不得不使用图形流水线模型，这对完成通用计算来说是个非常难用的编程模型

General-Purpose GPUs (GP-GPUs)

- 2006年, Nvidia 的 GeForce 8800 GPU 支持一种新的编程语言: CUDA
  - “Compute Unified Device Architecture”
  - 随后工业界推出OpenCL，与CUDA具有相同的ideas, 但独立于供应商
- Idea: 针对通用计算，发挥GPU的计算的高性能和存储器的高带宽来加速一些通用计算中的核心（Kernels）
- 一种附加处理器模型（GPU作为附加设备）：Host CPU发射数据并行的kernels 到GP-GPU上运行

Using CPU+GPU Architecture

- 异构系统（异构多核）
- 针对每个任务选择合适的处理器和存储器
- 通用CPU 适合执行一些串行的线程
  - 串行执行快
  - 带有cache，访问存储器延时低
- GPU 适合执行大量并行的线程
  - 可扩放的并行执行
  - 高带宽的并行存取