第5章 指令级并行及其开发——硬件方法

- 指令级并行：指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）

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）

5.3 指令的动态调度

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

5.3.1 动态调度的基本思想

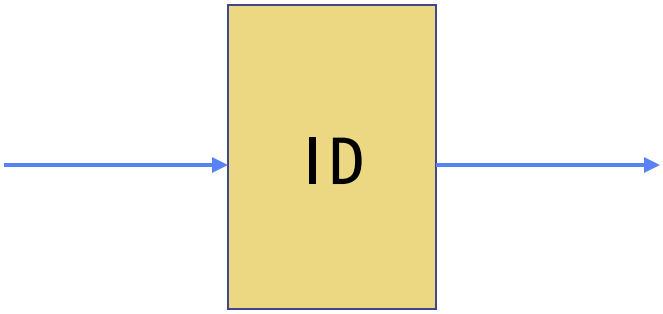
1. 到目前为止,我们所使用流水线的最大的局限性:
   - 指令是按序流出和按序执行的
   - 考虑下面一段代码:

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

ADD.D指令与DIV.D指令关于F4相关，导致流水线停顿。

SUB.D指令与流水线中的任何指令都没有关系，但也因此受阻。

在前面的基本流水线中：



检测结构冲突

检测数据冲突

一旦一条指令受阻，其后的指令都将停顿。

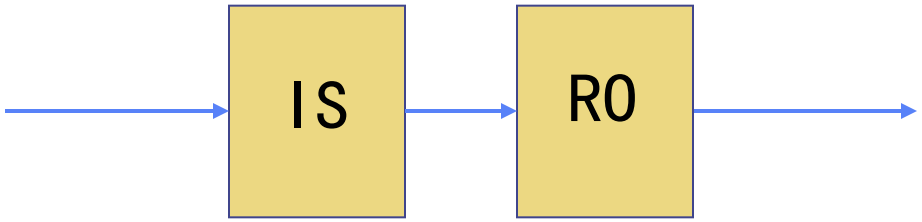
- 把指令流出的工作拆分为两步：
  - **检测结构冲突**
  - **等待数据冲突消失**

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。

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)



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. 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。

5.3.2 记分牌动态调度算法

1. 基本思想

- CDC 6600计算机最早采用此功能
  - 该机器用一个称为记分牌的硬件实现了对指令的动态调度。
  - 该硬件中维护着3张表，分别用于记录指令的执行状态、功能部件状态、寄存器状态以及数据相关关系等。
  - 它把前述5段流水线中的译码段ID分解成了两个段：**流出和读操作数**，以避免当某条指令在ID段被停顿时挡住后面无关指令的流动。

- 记分牌的目标：在没有结构冲突时，尽可能早地执行没有数据冲突的指令，实现每个时钟周期执行一条指令。
- 要发挥指令乱序执行的好处，必须有多条指令同时处于执行阶段。
  - **CDC 6600具有16个独立的功能部件**
    - **4个浮点部件**
    - **5个访存部件**
    - **7个整数操作部件**
- 假设
  - **所考虑的处理器有2个乘法器、1个加法器、1个除法部件和1个整数部件。**

- 整数部件用来处理所有的存储器访问、分支处理和整数操作。

➢ 采用了记分牌的MIPS处理器的基本结构

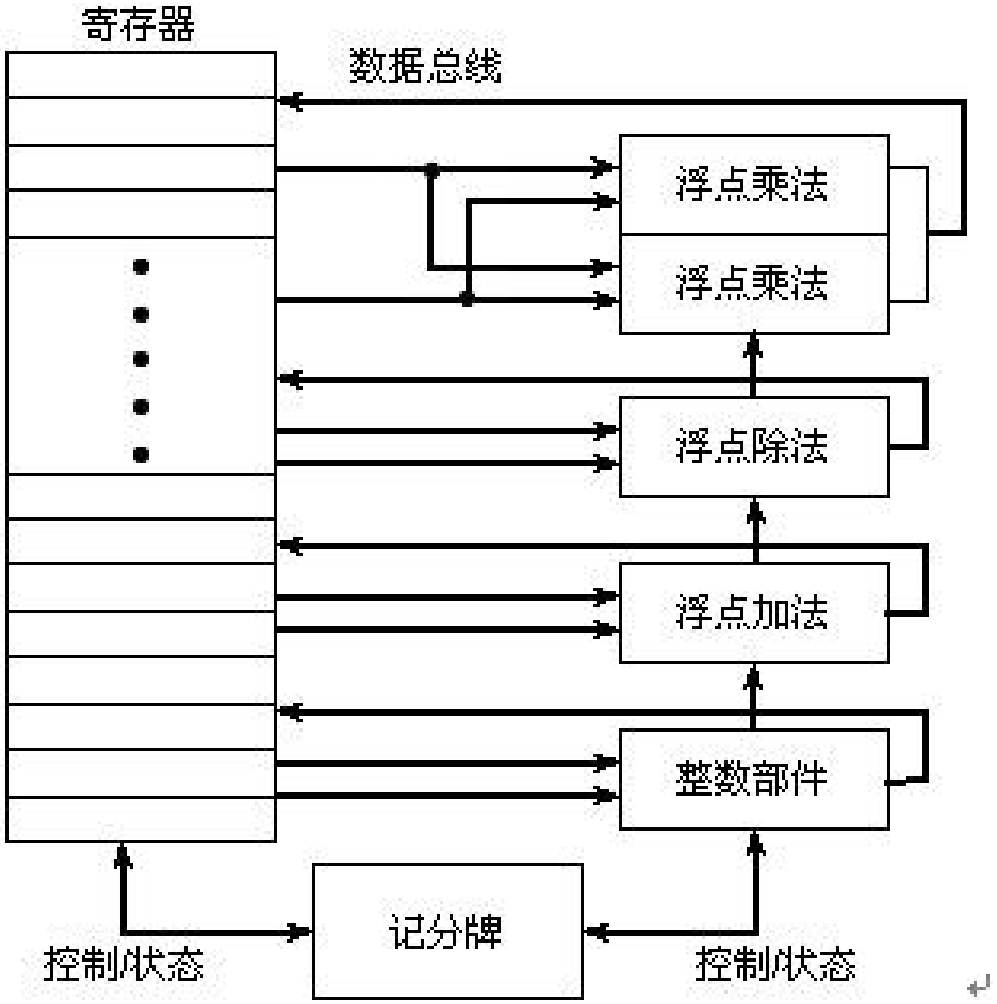
- 每条指令都要经过记分牌。
- 记分牌负责相关检测并控制指令的流出和执行。

➢ 每条指令的执行过程分为4段 （主要考虑浮点操作 ）

- 流出

  如果当前流出指令所需的功能部件空闲，并且所有其他正在执行的指令的目的寄存器与该指令的不同，记分牌就向功能部件流出该指令，并修改记分牌内部的记录表。

  解决了WAW冲突


寄存器
数据总线
浮点乘法
浮点乘法
浮点除法
浮点加法
整数部件
记分牌
控制/状态
控制/状态

- 读操作数

记分牌监测源操作数的可用性，如果数据可用，它就通知功能部件从寄存器中读出源操作数并开始执行。

动态地解决了RAW冲突，并导致指令可能乱序开始执行。

- 执行

取到操作数后，功能部件开始执行。当产生出结果后，就通知记分牌它已经完成执行。

在浮点流水线中，这一段可能要占用多个时钟周期。

- 写结果

记分牌一旦知道执行部件完成了执行，就检测是否存在WAR冲突。如果不存在，或者原有的WAR冲突已消失，记分牌就通知功能部件把结果写入目的寄存器，并释放该指令使用的所有资源。

- 如果检测到WAR冲突，就不允许该指令将结果写到目的寄存器。这发生在以下情况：
  - 前面的某条指令（按顺序流出）还没有读取操作数；而且：其中某个源操作数寄存器与本指令的目的寄存器相同。
  - 在这种情况下，记分牌必须等待，直到该冲突消失。

- 记分牌中记录的信息由3部分构成
  - 指令状态表：记录正在执行的各条指令已经进入到了哪一段。
  - 功能部件状态表：记录各个功能部件的状态。每个功能部件有一项，每一项由以下9个字段组成：
    - **Busy**：忙标志，指出功能部件是否忙。初值为“no”；
    - **Op**：该功能部件正在执行或将要执行的操作；
    - **Fi**：目的寄存器编号；
    - **Fj，Fk**：源寄存器编号；
    - **Qj，Qk**：指出向源寄存器Fj、Fk写数据的功能部件；

- **Rj，Rk**：标志位，为“yes”表示Fj，Fk中的操作数就绪且还未被取走。否则就被置为“no”。

- 结果寄存器状态表Result：每个寄存器在该表中有一项，用于指出哪个功能部件（编号）将把结果写入该寄存器。
  - 如果当前正在运行的指令都不以它为目的寄存器，则其相应项置为“no”。
  - Result各项的初值为“no”（全0）。

2. 举例

- MIPS记分牌所要维护的数据结构
- 下列代码运行过程中记分牌保存的信息

```
L.D      F6, 34(R2)
L.D      F2, 45(R3)
MULT.D   F0, F2, F4
SUB.D    F8, F6, F2
DIV.D    F10, F0, F6
ADD.D    F6, F8, F2
```

指　　令		指令状态表			
		流出	读操作数	执行	写结果
L.D	F6,34(R2)	√	√	√	√
L.D	F2, 45(R3)	√	√	√	
MULT.D	F0, F2, F4	√			
SUB.D	F8, F6, F2	√			
DIV.D	F10, F0, F6	√			
ADD.D	F6, F8, F2				

部件名称	功能部件状态表								
	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	yes	L.D	F2	R3				no	
Mult1	yes	MULT.D	F0	F2	F4	Integer		no	yes
Mult2	no								
Add	yes	SUB.D	F8	F6	F2		Integer	yes	no
Divide	yes	DIV.D	F10	F0	F6	Mult1		no	yes

	结果寄存器状态表							
	F0	F2	F4	F6	F8	F10	…	F30
部件名称	Mult1	Integer			Add	Divide		

MIPS记分牌中的信息

例5.1 假设浮点流水线中各部件的延迟如下：

加法需2个时钟周期

乘法需10个时钟周期

除法需40个时钟周期

代码段和记分牌信息的起始点状态如上图。分别给出MULT.D和DIV.D准备写结果之前的记分牌状态。

解 图中的代码段存在以下相关性：

（1）先写后读相关：第二条L.D指令到MULT.D和SUB.D之间，MULT.D到DIV.D之间，SUB.D到ADD.D之间；

（2）先读后写相关：DIV.D和ADD.D之间，SUB.D和ADD.D之间；

（3）结构相关：ADD.D和SUB.D指令关于浮点加法部件。

指　　令		指令状态表			
		流出	读操作数	执行	写结果
L.D	F6, 34(R2)	√	√	√	√
L.D	F2, 45(R3)	√	√	√	√
MULT.D	F0, F2, F4	√	√	√	
SUB.D	F8, F6, F2	√	√	√	√
DIV.D	F10, F0, F6	√			
ADD.D	F6, F8, F2	√	√	√	

部件名称	功能部件状态表								
	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	no								
Mult1	yes	MULT.D	F0	F2	F4			no	no
Mult2	no								
Add	yes	ADD.D	F6	F8	F2			no	no
Divide	yes	DIV.D	F10	F0	F6	Mult1		no	yes

	结果寄存器状态表							
	F0	F2	F4	F6	F8	F10	…	F30
部件名称	Mult1			Add		Divide		

程序段执行到MULT.D将要写结果时记分牌的状态

指　　令		指令状态表			
		流出	读操作数	执行	写结果
L.D	F6, 34(R2)	√	√	√	√
L.D	F2, 45(R3)	√	√	√	√
MULT.D	F0, F2, F4	√	√	√	√
SUB.D	F8, F6, F2	√	√	√	√
DIV.D	F10, F0, F6	√	√	√	
ADD.D	F6, F8, F2	√	√	√	√

部件名称	功能部件状态表								
	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	no								
Mult1	no								
Mult2	no								
Add	no								
Divide	yes	DIV.D	F10	F0	F6			no	no

	结果寄存器状态表							
	F0	F2	F4	F6	F8	F10	…	F30
部件名称						Divide		

程序段执行到DIV.D将要写结果时记分牌的状态

5.3.3 Tomasulo算法

Tomasulo算法基本思想

1. 核心思想
   - 记录和检测指令相关，操作数一旦就绪就立即执行，把发生RAW冲突的可能性减少到最小；
   - 通过寄存器换名来消除WAR冲突和WAW冲突。
2. IBM 360/91首先采用了Tomasulo算法。
   - IBM 360/91的设计目标是基于整个360系列的统一指令集和编译器来实现高性能，而不是设计和利用专用的编译器来提高性能。

     需要更多地依赖于硬件。

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。

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

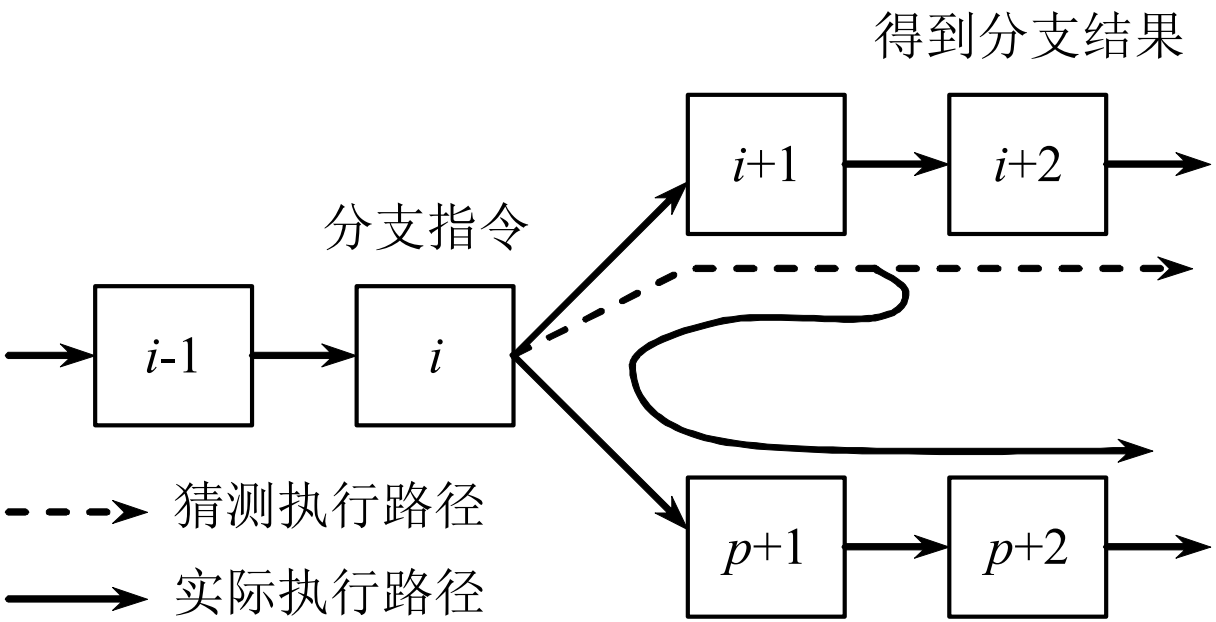
采用动态分支预测技术的目的

- 预测分支是否成功
- 尽快找到分支目标地址（或指令）

需要解决的关键问题

- **如何记录分支的历史信息，要记录哪些信息？**
- **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？**

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。



5.4.1 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。

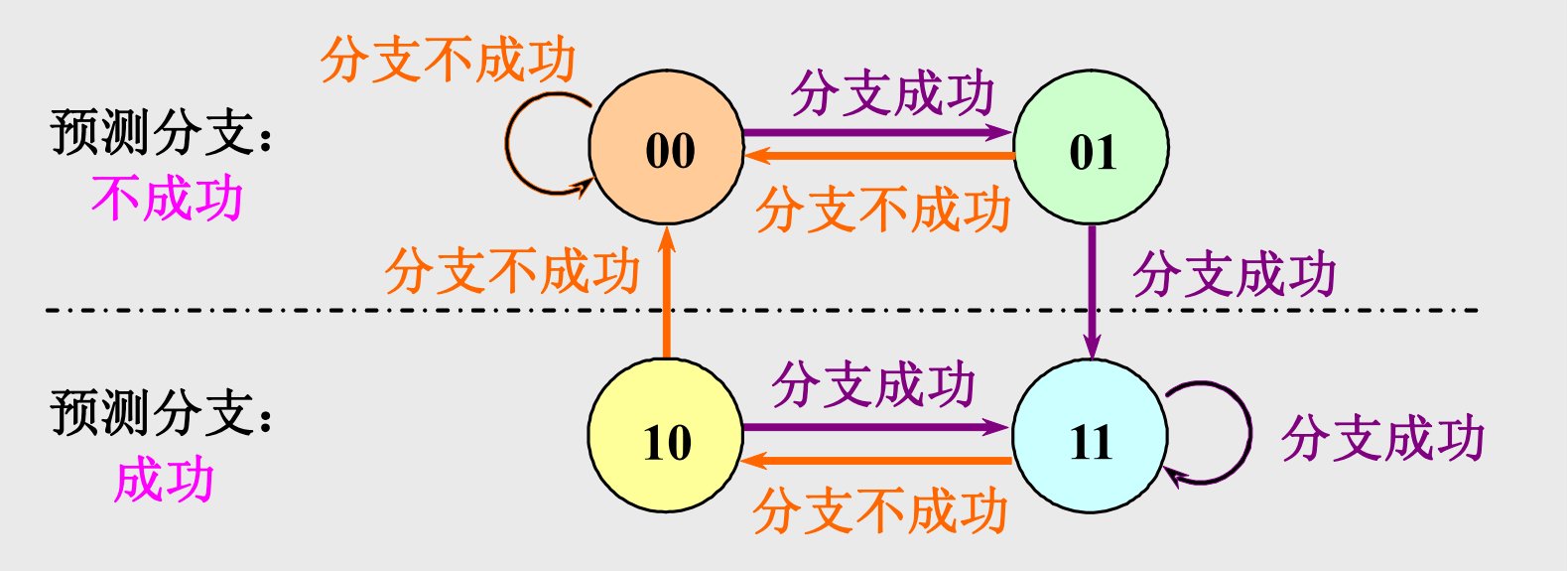
2. 只有1个预测位的分支预测

   记录分支指令最近一次的历史，BHT中只需要1位二进制位。

3. 采用两位二进制位来记录历史

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

➢ 两位分支预测的状态转换如下所示：



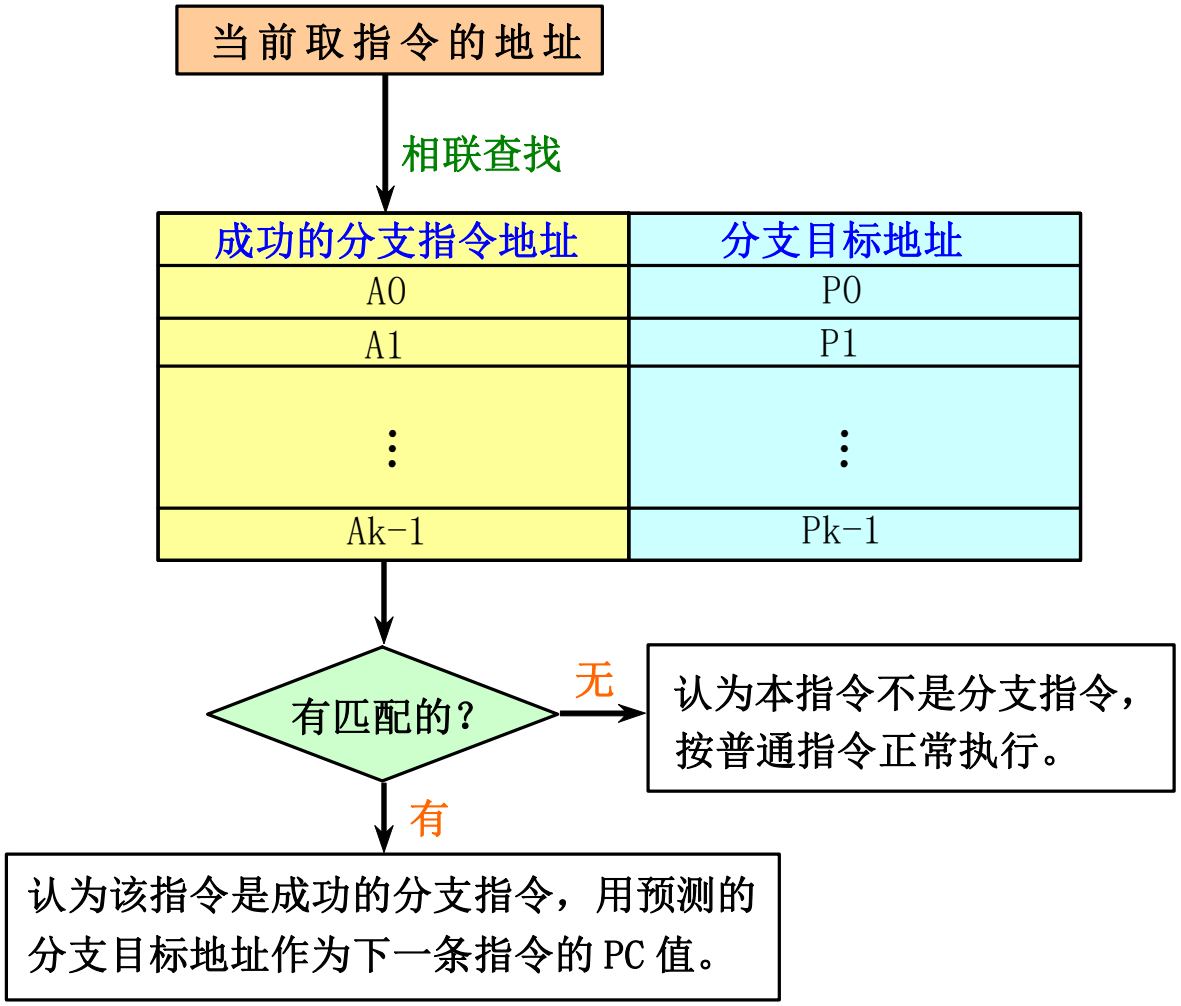
5.4.2 采用分支目标缓冲器BTB

目标：将分支的开销降为 0

方法：分支目标缓冲

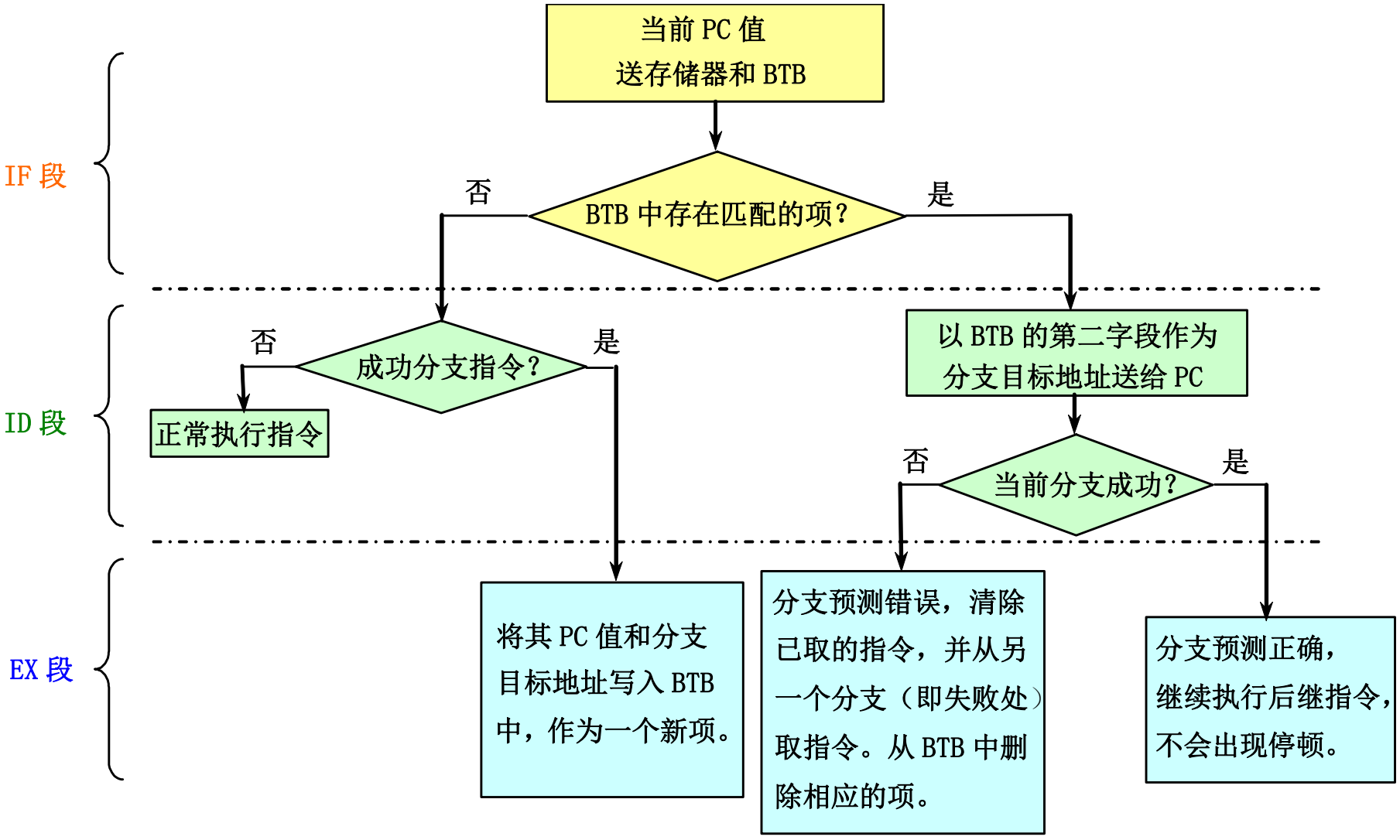
- 将分支成功的分支指令的地址和它的分支目标地址都放到一个缓冲区中保存起来，缓冲区以分支指令的地址作为标识。
- 这个缓冲区就是分支目标缓冲器（Branch-Target Buffer，简记为BTB，或者分支目标Cache （ranch-Target Cache）。

1. BTB的结构


当前取指令的地址
相联查找
成功的分支指令地址
分支目标地址
A0
P0
A1
P1
Ak-1
Pk-1
有匹配的？
无
认为本指令不是分支指令，
按普通指令正常执行。
有
认为该指令是成功的分支指令，用预测的
分支目标地址作为下一条指令的 PC 值。

- 用专门的硬件实现的一张表格。
- 表格中的每一项至少有两个字段：
  - **执行过的成功分支指令的地址；**
  - **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：

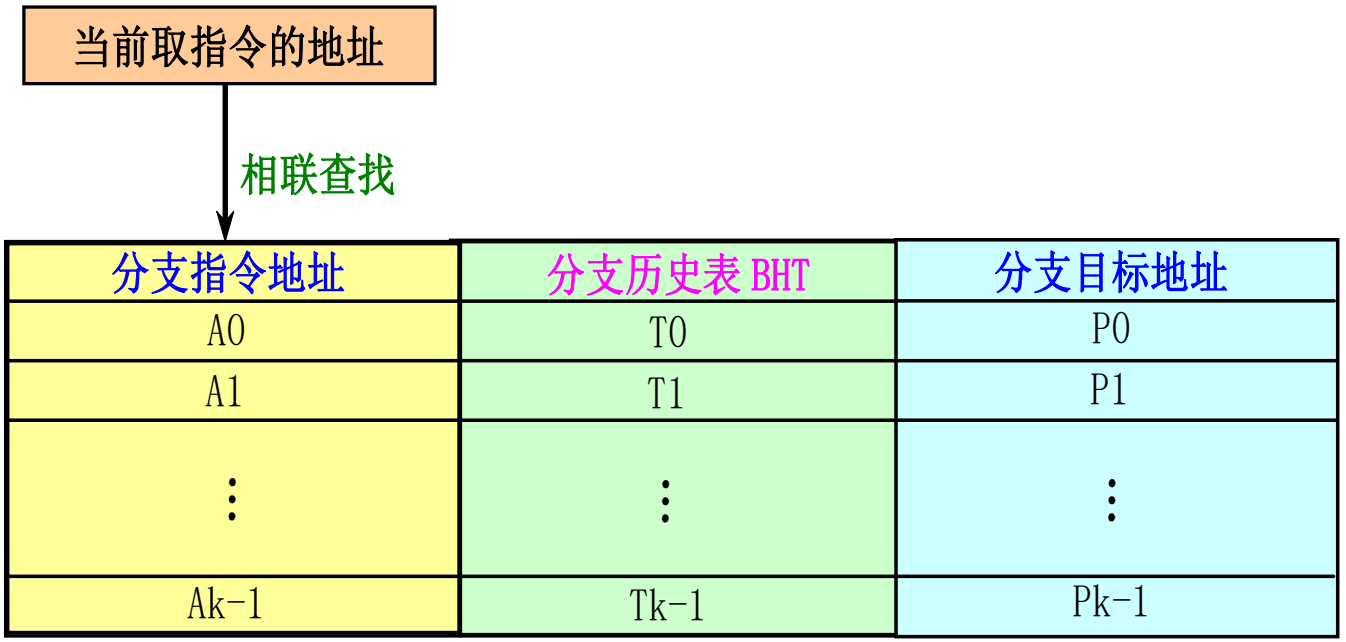

当前 PC 值
送存储器和 BTB
IF 段
否
BTB 中存在匹配的项？
是
ID 段
否
成功分支指令？
是
正常执行指令
以 BTB 的第二字段作为
分支目标地址送给 PC
否
当前分支成功？
是
EX 段
将其 PC 值和分支
目标地址写入 BTB
中，作为一个新项。
分支预测错误，清除
已取的指令，并从另
一个分支（即失败处）
取指令。从 BTB 中删
除相应的项。
分支预测正确，
继续执行后继指令，
不会出现停顿。

3. 采用BTB后，各种可能情况下的延迟 ：

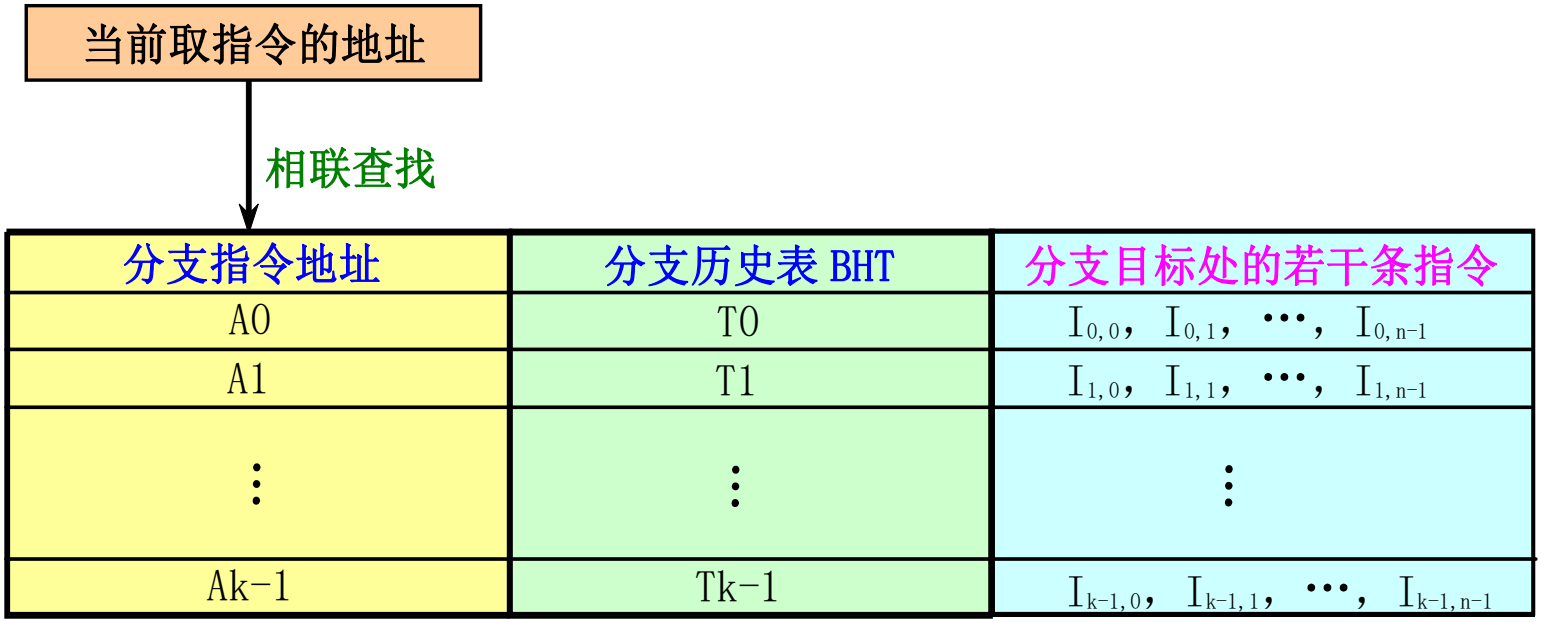
指令在BTB中？	预测	实际情况	延迟周期
是	成功	成功	0
是	成功	不成功	2
不是		成功	2
不是		不成功	0

4. BTB的另一种形式

- 在分支目标缓冲器中增设一个至少是两位的“分支历史表”字段

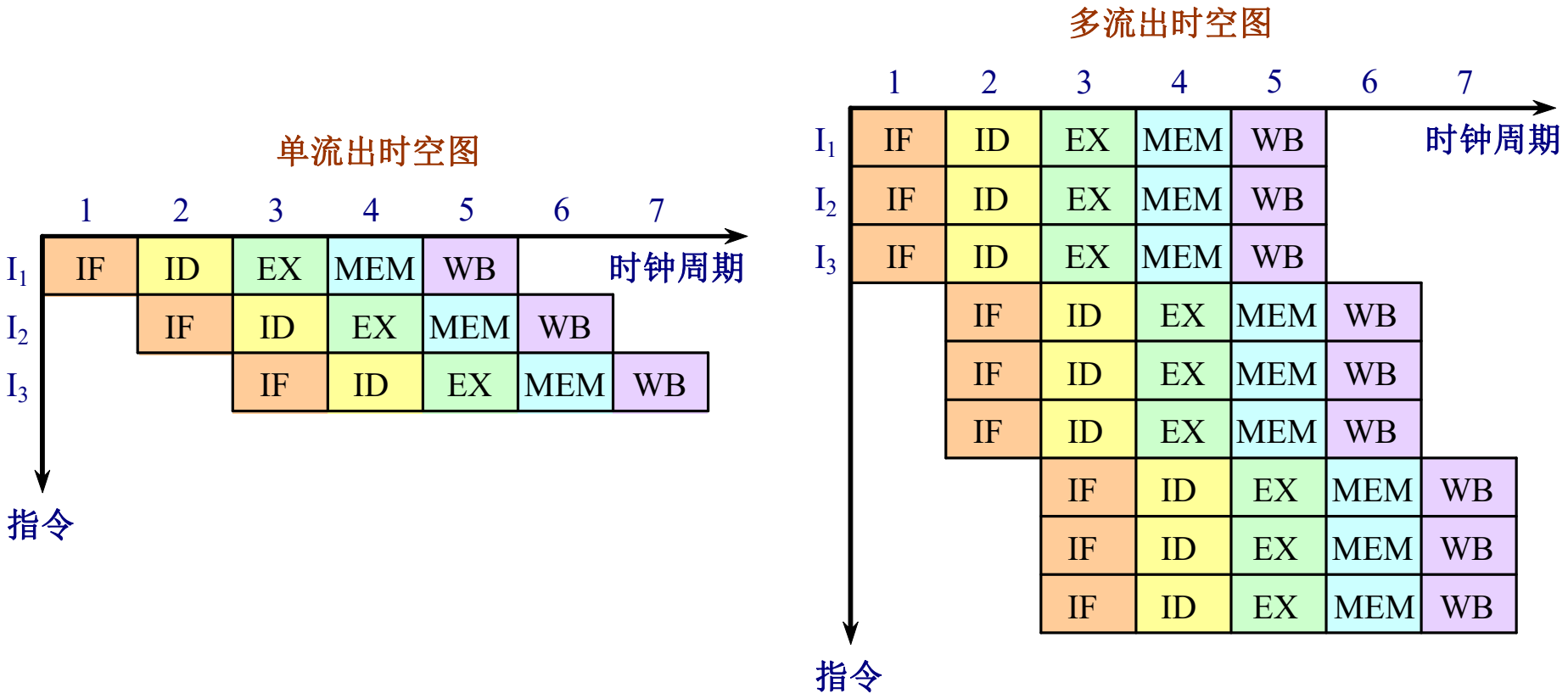


5. 更进一步，在表中对于每条分支指令都存放若干条分支目标处的指令，就形成了分支目标指令缓冲器。



5.5 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



单流出和多流出处理机执行指令的时空图

1. 多流出处理机有两种基本风格：
   - 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。

基本的多流出技术、特点以及实例

技 术	流出结构	冲突检测	调 度	主要特点	处理机实例
超标量（静态）	动态	硬件	静态	按序执行	Sun UltraSPARCII/III
超标量（动态）	动态	硬件	动态	部分乱序执行	IBM Power2
超标量（猜测）	动态	硬件	带有前瞻的动态调度	带有前瞻的乱序执行	Pentium III/4，MIPS R10K，Alpha 21264，HP PA 8500，IBM RS64III
VLIW /LIW	静态	软件	静态	流出包之间没有冲突	Trimedia，i860
EPIC	主要是静态	主要是软件	主要是静态	相关性被编译器显式地标记出来	Itanium

EPIC: Explicitly Parallel Instruction Computer

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. VLIW存在的一些问题

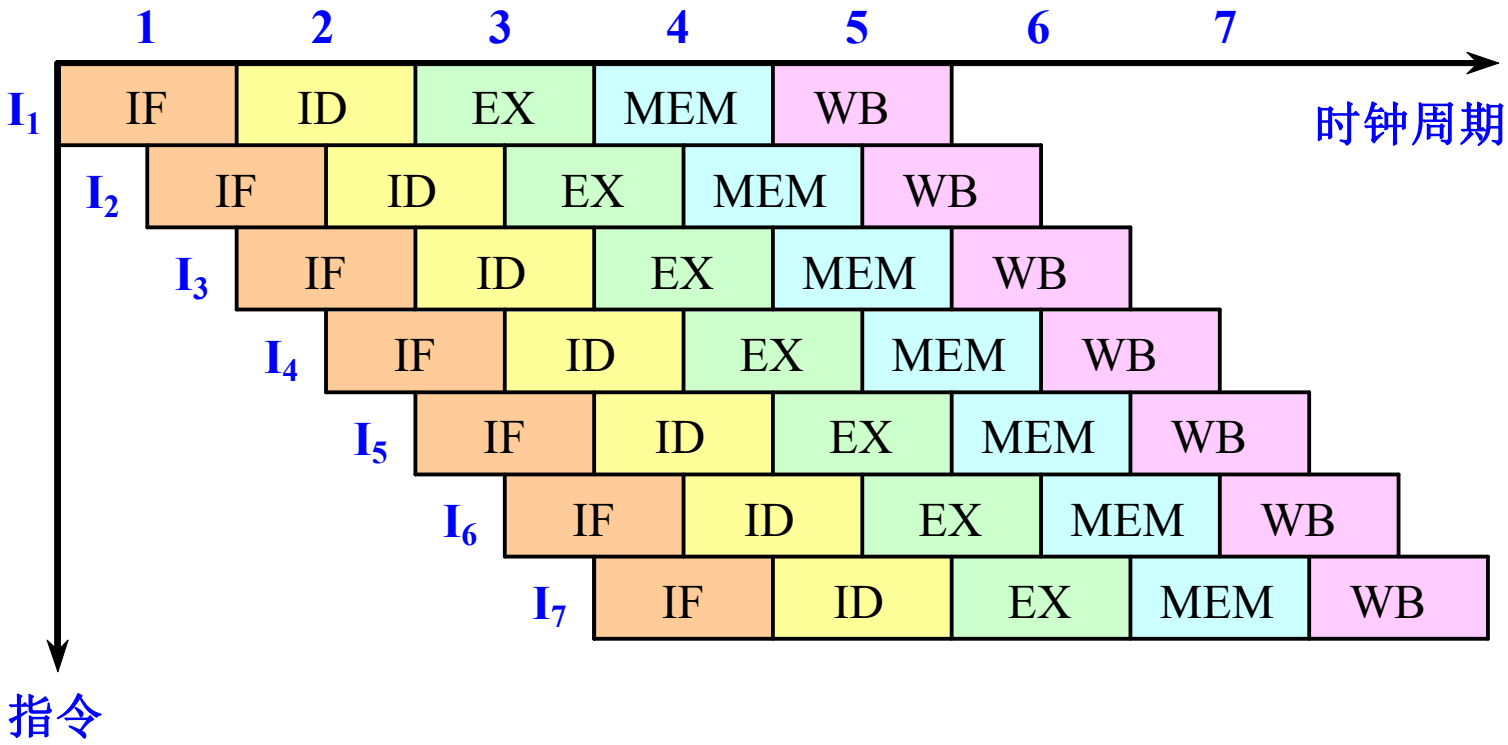
- 程序代码长度增加了
  - 提高并行性而进行的大量的循环展开；
  - 指令字中的操作槽并非总能填满。

  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

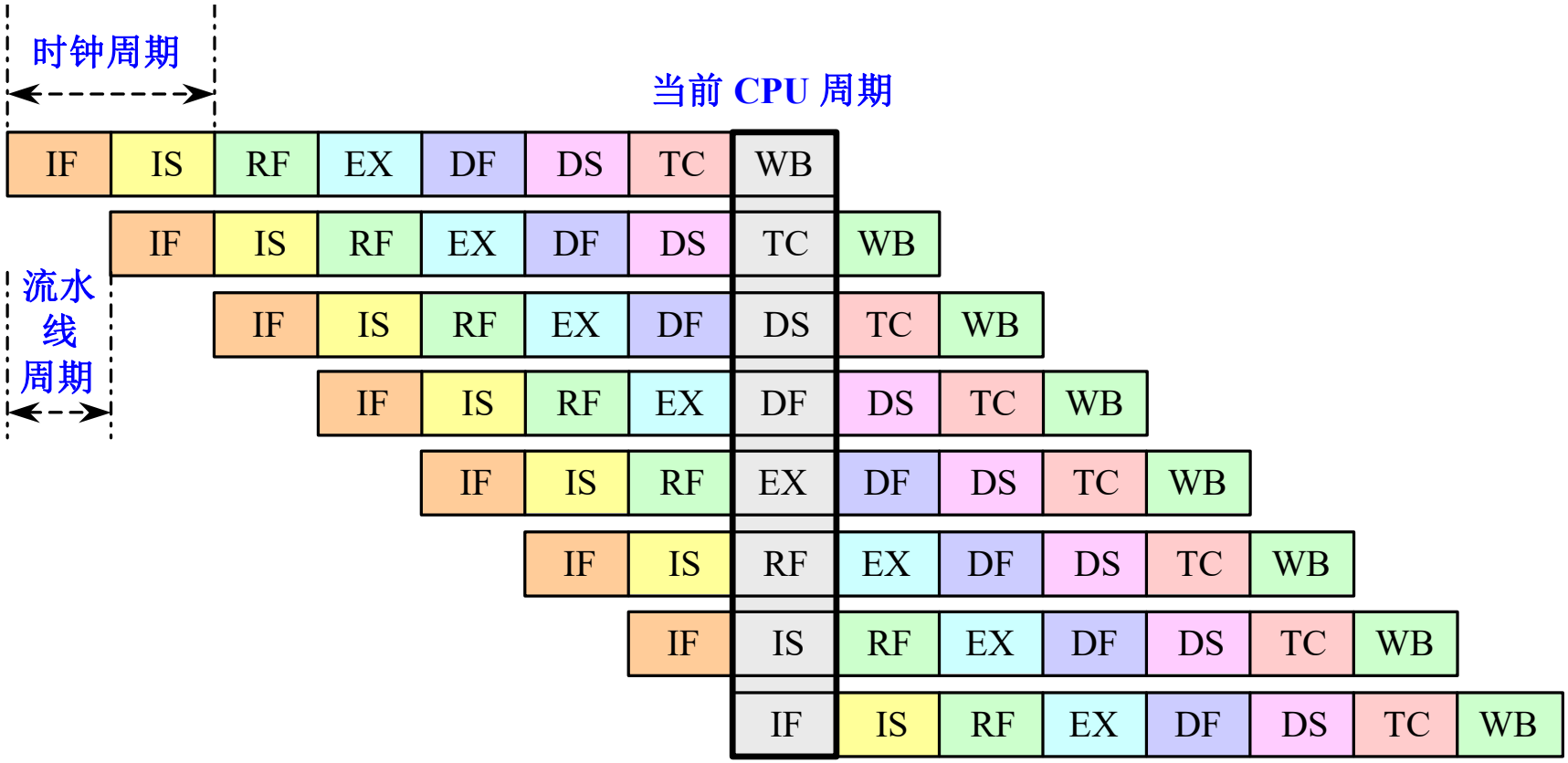
  任何一个操作部件出现停顿时，整个处理机都要停顿。
- 机器代码的不兼容性

5.5.5 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。


1
2
3
4
5
6
7
I1
IF
ID
EX
MEM
WB
时钟周期
I2
IF
ID
EX
MEM
WB
I3
IF
ID
EX
MEM
WB
I4
IF
ID
EX
MEM
WB
I5
IF
ID
EX
MEM
WB
I6
IF
ID
EX
MEM
WB
I7
IF
ID
EX
MEM
WB
指令

➢ MIPS R4000指令流水线时空图



5	①分支预测技术(5.4.1节、5.4.2节)	5.8 5.9	
	②超标量/超长指令字/超流水(5.5节、5.5.3节、5.5.5节)	5.11	

第6章 指令级并行的开发——软件方法

6.1 基本指令调度及循环展开

6.1.1 指令调度的基本方法

1. 指令调度：找出不相关的指令序列，让它们在流水线上重叠并行执行。
2. 制约编译器指令调度的因素
   - 程序固有的指令级并行
   - 流水线功能部件的延迟

表6.1本节使用的浮点流水线的延迟

产生结果的指令	使用结果的指令	延迟(cycles)
浮点计算	另一个浮点计算	3
浮点计算	浮点store(S.D)	2
浮点Load(L.D)	浮点计算	1
浮点Load(L.D)	浮点store(S.D)	0

例6.1 对于下面的源代码，转换成MIPS汇编语言，在不进行指令调度和进行指令调度两种情况下，分析其代码一次循环所需的执行时间。

```
for (i=1000; i>0; i--)
    x[i] = x[i] + s;
```

解：

先把该程序翻译成MIPS汇编语言代码

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

- 在不进行指令调度的情况下，根据表中给出的浮点流水线中指令执行的延迟，程序的实际执行情况如下：

			指令流出时钟
Loop:	L.D	F0, 0(R1)	1
	(空转)		2
	ADD.D	F4, F0, F2	3
	(空转)		4
	(空转)		5
	S.D	F4, 0(R1)	6
	DADDIU	R1, R1, #-8	7
	(空转)		8
	BNE	R1, R2, Loop	9
	(空转)		10

- 在用编译器对上述程序进行指令调度以后，程序的执行情况如下：

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

6.1.2 循环展开

1. 循环展开

- 把循环体的代码复制多次并按顺序排放， 然后相应调整循环的结束条件。
- 开发循环级并行的有效方法

例6.2 将例6.1中的循环展开3次得到4个循环体，然后对展开后的指令序列在不调度和调度两种情况下，分析代码的性能。假定R1-R2的初值为32的倍数，即循环次数为4的倍数。消除冗余的指令，并且不要重复使用寄存器。

- 展开后没有调度的代码如下(需要分配寄存器)

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

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

50%是空转周期！

- 调度后的代码如下：

			指令流出时钟
Loop:	L.D	F0, 0（R1）	1
	L.D	F6, -8（R1）	2
	L.D	F10, -16（R1）	3
	L.D	F14, -24（R1）	4
	ADD.D	F4, F0, F2	5
	ADD.D	F8, F6, F2	6
	ADD.D	F12, F10, F2	7
	ADD.D	F16, F14, F2	8
	S.D	F4, 0（R1）	9
	S.D	F8, -8（R1）	10
	DADDIU	R1, R1, # -32	12
	S.D	F12, 16（R1）	11
	BNE	R1, R2, Loop	13
	S.D	F16, 8（R1）	14

结论：通过循环展开、寄存器重命名和指令调度，可以有效开发出指令级并行。

没有空转周期！

Major techniques with the component of the CPI equation

Technique	Reduces	Section
Forwarding and bypassing	Potential data hazard stalls	C.2
Delayed branches and simple branch scheduling	Control hazard stalls	C.2
Basic compiler pipeline scheduling	Data hazard stalls	C.2, 3.2
Basic dynamic scheduling (scoreboarding)	Data hazard stalls from true dependences	C.7
Loop unrolling	Control hazard stalls	3.2
Branch prediction	Control stalls	3.3
Dynamic scheduling with renaming	Stalls from data hazards, output dependences, and antidependences	3.4
Hardware speculation	Data hazard and control hazard stalls	3.6
Dynamic memory disambiguation	Data hazard stalls with memory	3.6
Issuing multiple instructions per cycle	Ideal CPI	3.7, 3.8
Compiler dependence analysis, software pipelining, trace scheduling	Ideal CPI, data hazard stalls	H.2, H.3
Hardware support for compiler speculation	Ideal CPI, data hazard stalls, branch hazard stalls	H.4, H.5

Five primary approaches in use for multiple-issue processors

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	dynamic	hardware	static	in-order execution	mostly in the embedded space: MIPS and ARM
Superscalar (dynamic)	dynamic	hardware	dynamic	some out-of-order execution, but no speculation	none at the present
Superscalar (speculative)	dynamic	hardware	dynamic with speculation	out-of-order execution with speculation	Pentium 4, MIPS R12K, IBM Power5
VLIW/LIW	static	primarily software	static	all hazards determined and indicated by compiler (often implicitly)	most examples are in the embedded space, such as the TI C6x
EPIC	primarily static	primarily software	mostly static	all hazards determined and indicated explicitly by the compiler	Itanium