计算机系统结构

第5章 指令级并行及其开发

目录

5.4 动态分支预测技术

1. 现有方法的不足与分支预测的重要性

A. 所开发的ILP越多，控制相关的制约就越大，分支预测就要有更高的准确度。

B. 本节中介绍的方法对于每个时钟周期流出多条指令（若为n条，就称为n流出）的处理机来说非常重要。

因为：

- 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。

2. 解决方法——动态分支预测

A. 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。
- 有更好的预测准确度和适应性。

B. 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

3.目标与关键问题

A. 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）

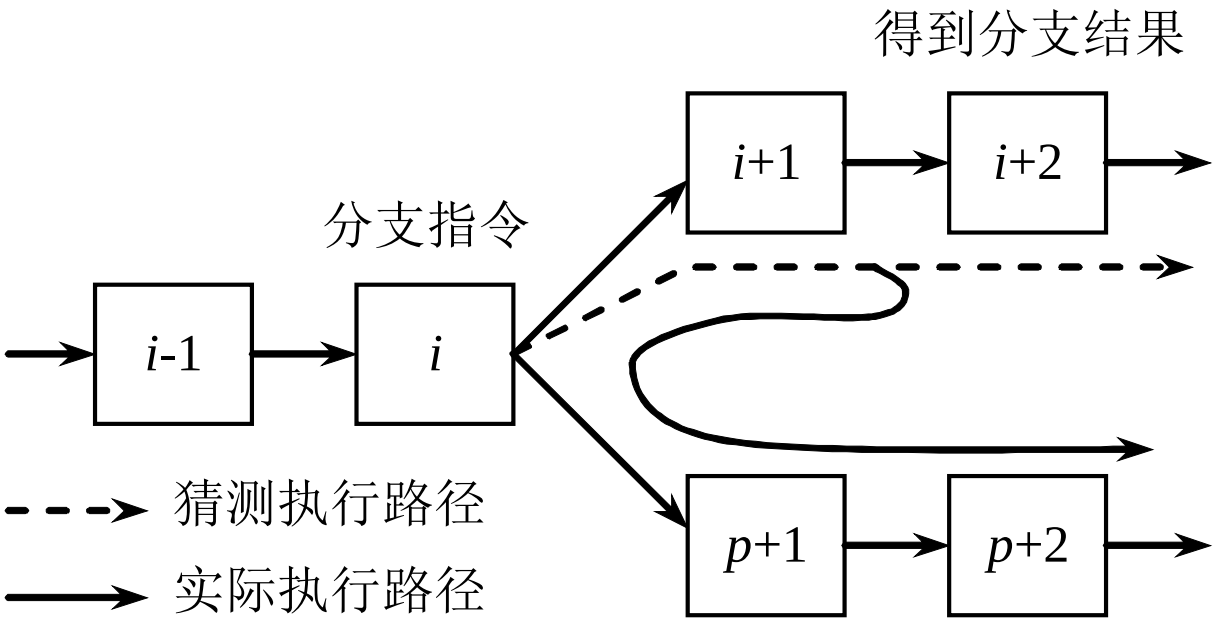
（避免控制相关造成流水线停顿）

B. 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？

4.预测错误时的处理方法

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

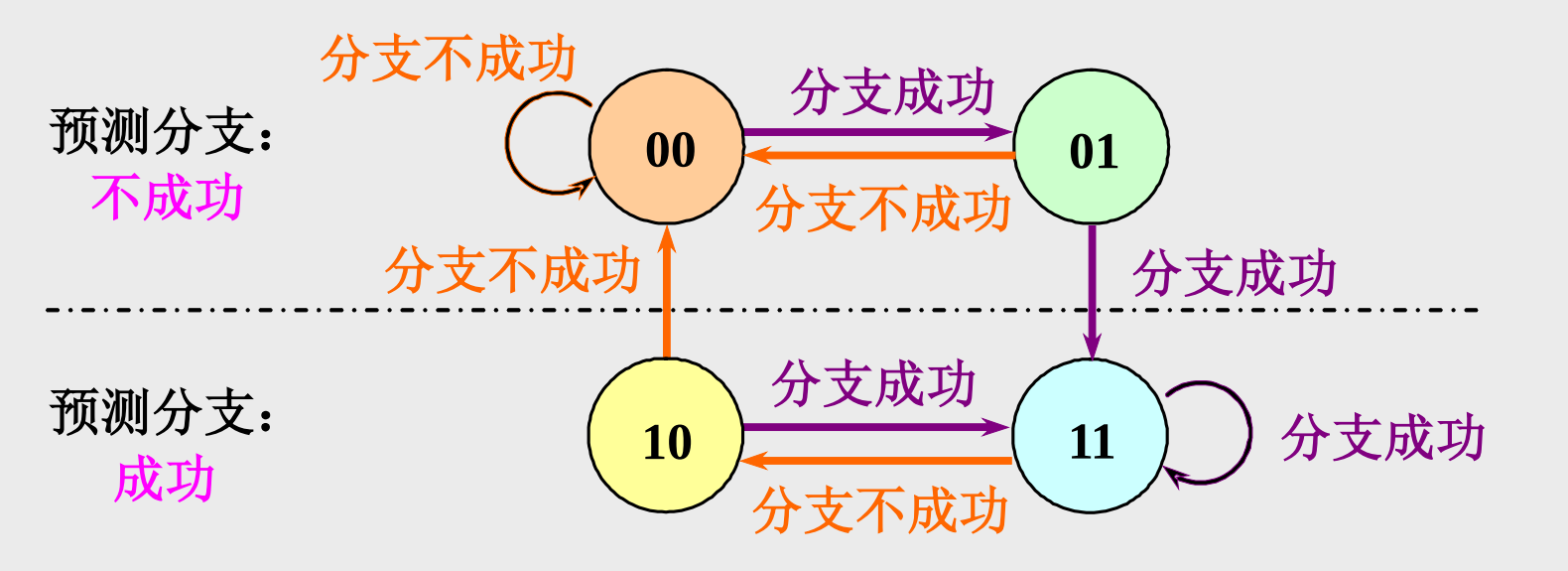
   记录分支指令最近一次的历史，BHT中只需要1位二进制位。

   （最简单）

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：



4. 两位分支预测中的操作步骤：

➢ 两位分支预测中的操作步骤：

- 分支预测；
  - 当分支指令到达译码段（ID）时，根据从BHT读出的信息进行分支预测 。
  - 若预测正确，就继续处理后续的指令，流水线没有断流。否则，就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。
- 状态修改。

5. BHT的有效范围：

- 判定分支是否成功所需的时间大于确定分支目标地址所需的时间。

前述5段经典流水线：由于判定分支是否成功和计算分支目标地址都是在ID段完成，所以BHT方法不会给该流水线带来好处。

6. BHT的作用效果

研究结果表明：对于SPEC89测试程序来说，具有大小为4KB的BHT的预测准确率为82%～99%。

一般来说，采用4KB的BHT就可以了。

7. BHT的实现

可以跟分支指令一起存放在指令Cache中；

也可以用一块专门的硬件来实现。

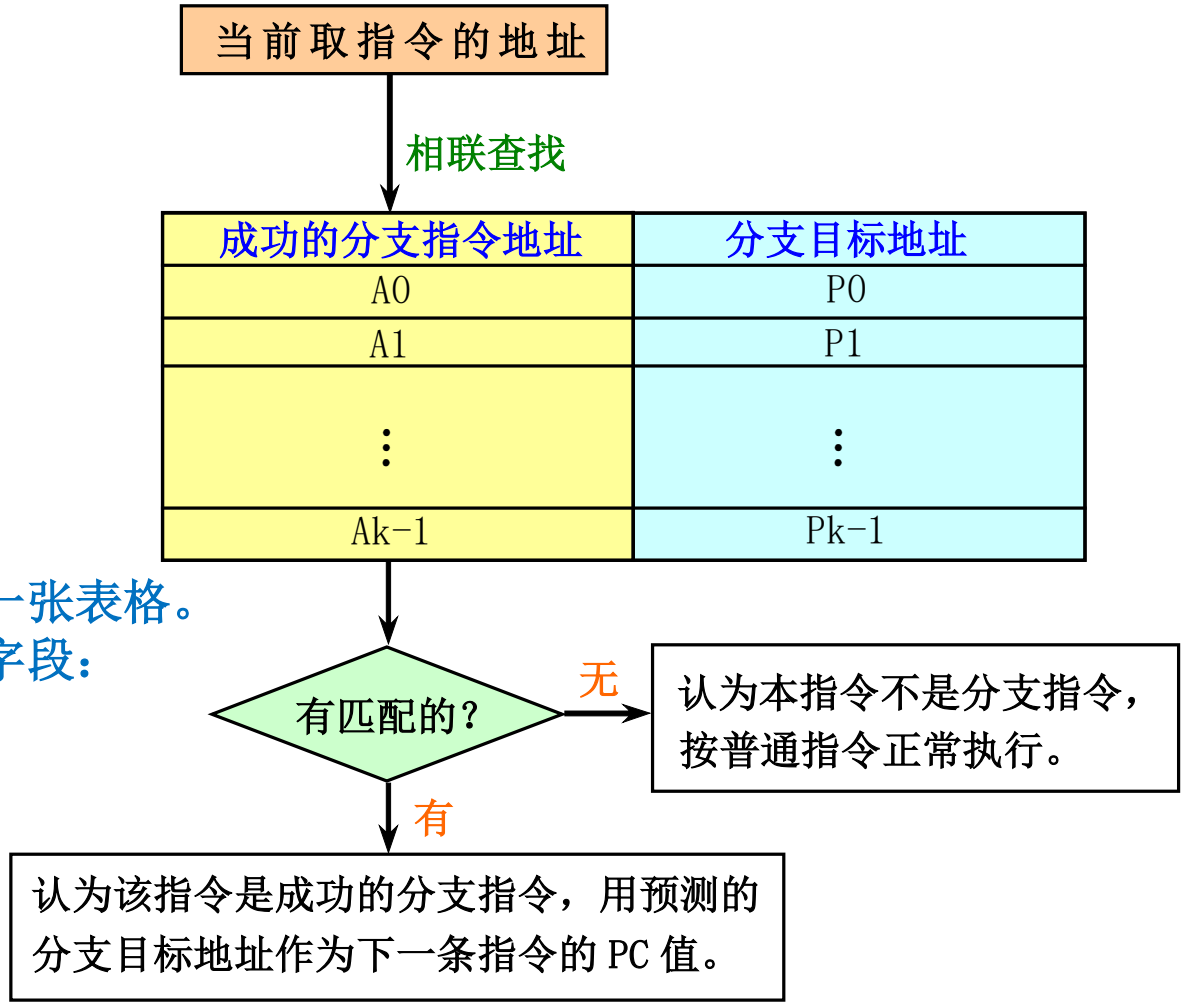
5.4.2 分支目标缓冲器BTB

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 （Branch-Target Cache）。

1. BTB的结构

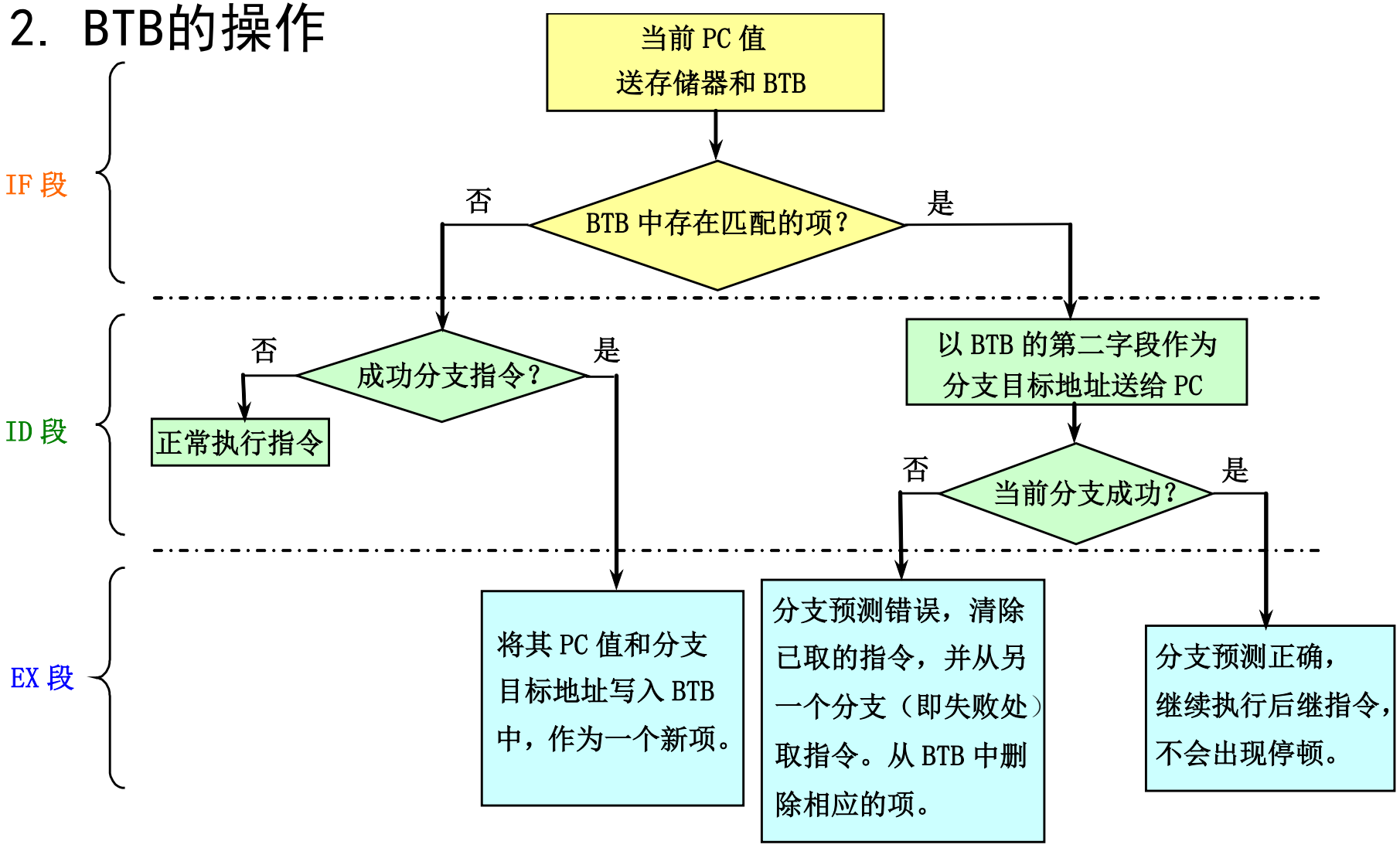


看成是用专门的硬件实现的一张表格。
表格中的每一项至少有两个字段：

*执行过的成功分支指令的地址；
（作为该表的匹配标识）

* 预测的分支目标地址：

2. BTB的操作

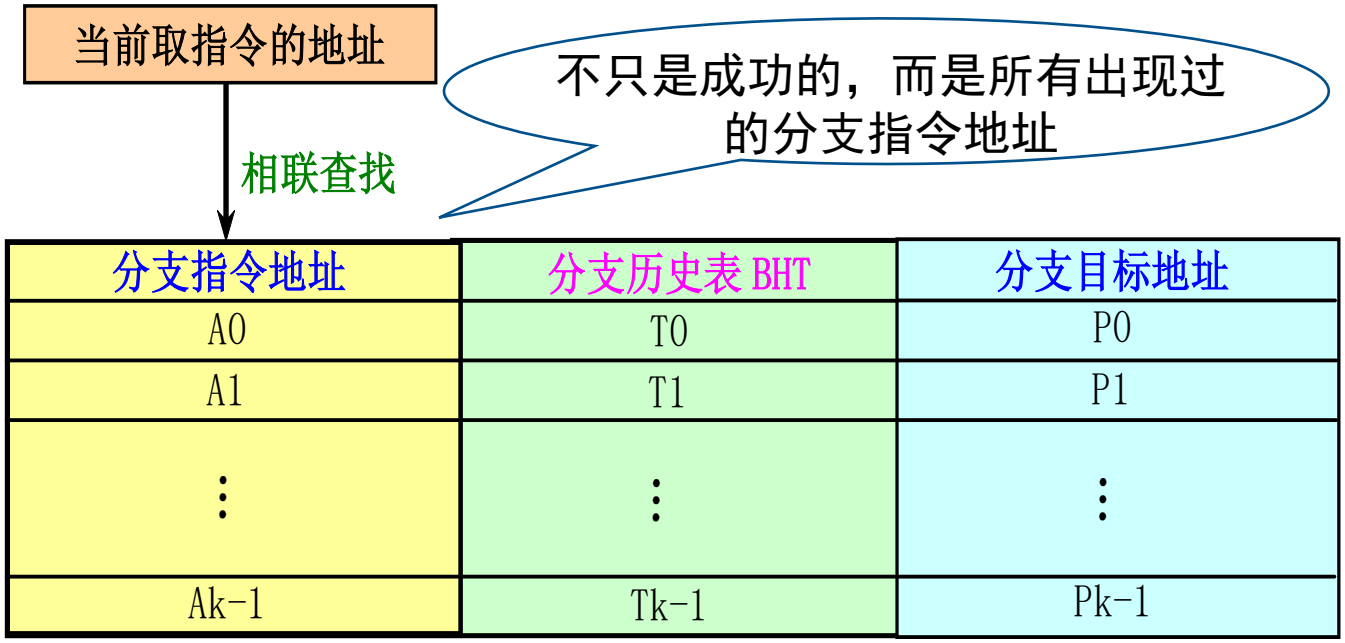

当前 PC 值
送存储器和 BTB
IF 段
否
BTB 中存在匹配的项？
是
ID 段
否
成功分支指令？
是
正常执行指令
以 BTB 的第二字段作为
分支目标地址送给 PC
否
当前分支成功？
是
EX 段
将其 PC 值和分支
目标地址写入 BTB
中，作为一个新项。
分支预测错误，清除
已取的指令，并从另
一个分支（即失败处）
取指令。从 BTB 中删
除相应的项。
分支预测正确，
继续执行后继指令，
不会出现停顿。

3. BTB各种情况下的延迟 ：

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

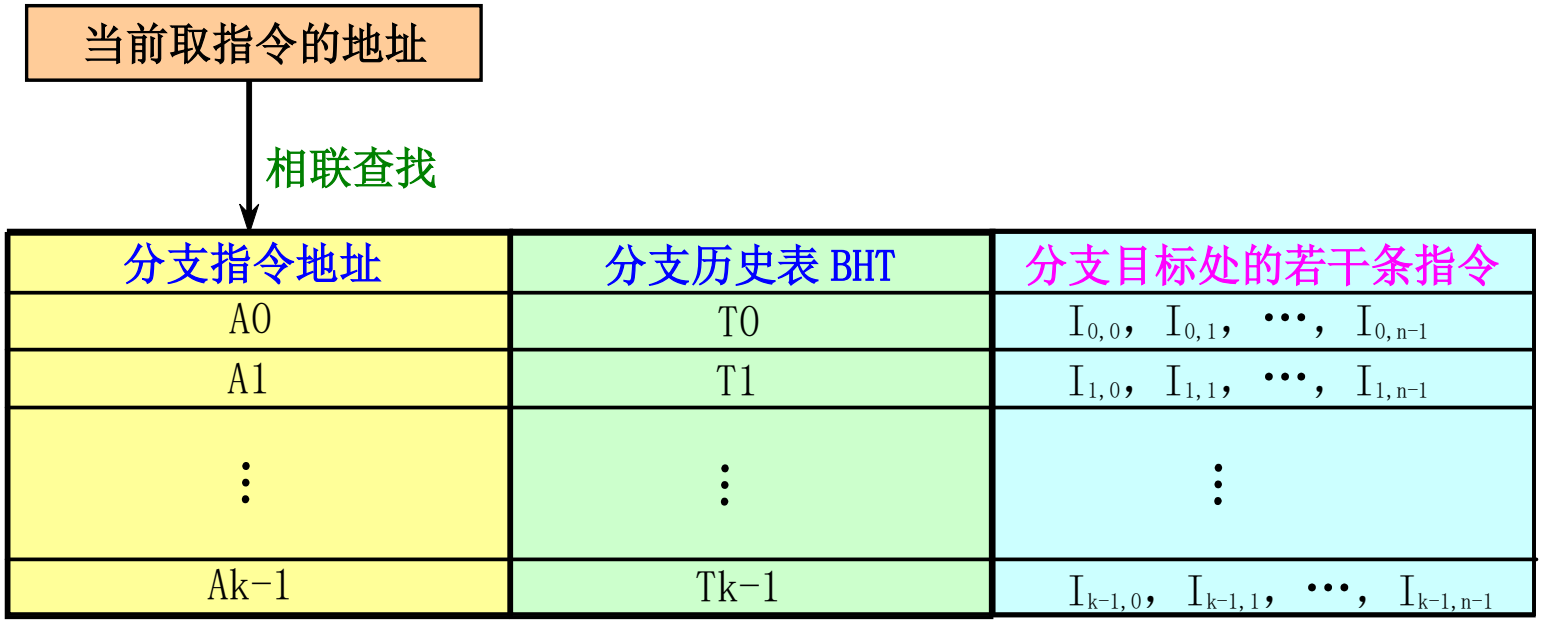
4. 改进BTB——提升预测准确率

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. 改进BTB——降低访存开销

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。



这里存指令的好处：IF是通过指令地址取指令，分支成功，程序的空间局部性被破坏，取指令的时延很可能会增加（1级i-cache miss）。

5. 4. 3 基于硬件的前瞻执行

前瞻执行（Speculation）的基本思想：

对分支指令的结果进行猜测，并假设这个猜测总是对的，然后按这个猜测结果继续取、流出和执行后续的指令。只是执行指令的结果不是写回到寄存器或存储器，而是写入一个称为再定序缓冲器ROB（ReOrder Buffer）中 。等到相应的指令得到“确认”（Commit）（即确实是应该执行的）之后，才将结果写入寄存器或存储器。

1. 基于硬件的前瞻执行结合了3种思想：
   - 动态分支预测。用来选择后续执行的指令。
   - 在控制相关的结果尚未出来之前，前瞻地执行后续指令。
   - 用动态调度对基本块的各种组合进行跨基本块的调度。
2. 对Tomasulo算法加以扩充，就可以支持前瞻执行。

   把Tomasulo算法的写结果和指令完成加以区分，分成两个不同的段：

   写结果 + 指令确认

- 写结果段
  - 把前瞻执行的结果写到ROB中；
  - 通过CDB在指令之间传送结果，供需要用到这些结果的指令使用。
- 指令确认段

  在分支指令的结果出来后，对相应指令的前瞻执行给予确认。
  - 如果前面所做的猜测是对的，把在ROB中的结果写到寄存器或存储器。
  - 如果发现前面对分支结果的猜测是错误的，那就不予以确认，并从那条分支指令的另一条路径开始重新执行。

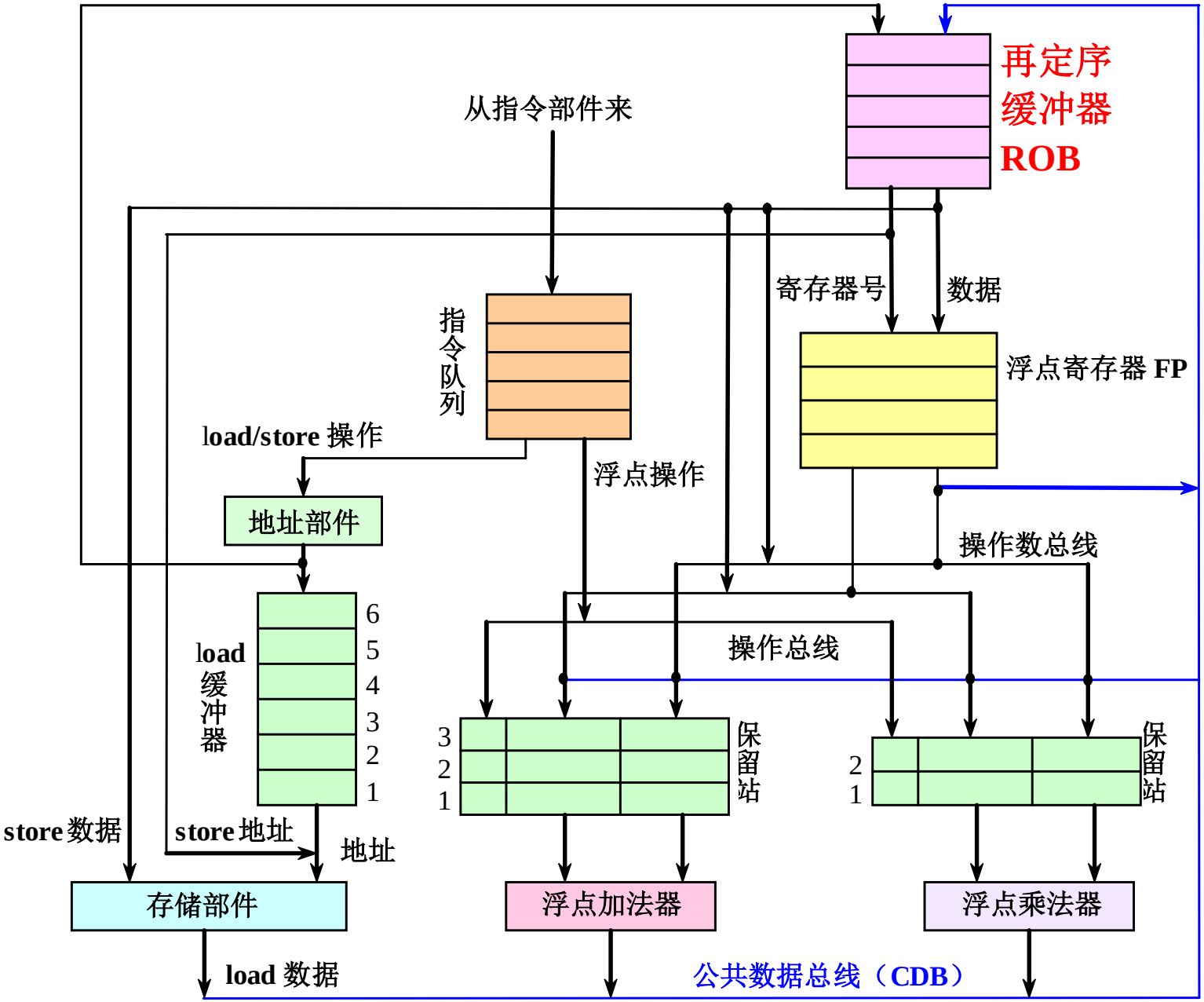
3. 实现前瞻的关键思想：

允许指令乱序执行，但必须顺序确认；

在指令被确认之前，不允许它进行不可恢复的操作。

4. 支持前瞻执行的浮点部件的结构

增加ROB缓冲器，并将其合并store缓冲器的功能。


再定序
缓冲器
ROB
从指令部件来
寄存器号
数据
指令队列
浮点寄存器 FP
load/store 操作
浮点操作
地址部件
操作数总线
操作总线
load
缓冲器
6
5
4
3
2
1
保留站
3
2
1
保留站
2
1
store 数据
store 地址
地址
存储部件
浮点加法器
浮点乘法器
load 数据
公共数据总线（CDB）

- ROB中的每一项由以下4个字段组成：
  - 指令类型

    指出该指令是分支指令、store指令或寄存器操作指令。
  - 目标地址

    给出指令执行结果应写入的目标寄存器号（如果是load和ALU指令）或存储器单元的地址（如果是store指令）。
  - 数据值字段

    用来保存指令前瞻执行的结果，直到指令得到确认。
  - 就绪字段

    指出指令是否已经完成执行并且数据已就绪。

- Tomasulo算法中保留站的换名功能是由ROB来完成的。

5. 采用前瞻执行机制后，指令的执行步骤：

（在Tomasulo算法的基础上改造的 ）

- 流出
  - 从浮点指令队列的头部取一条指令。
  - 如果有空闲的保留站（设为r）且有空闲的ROB项（设为b），就流出该指令，并把相应的信息放入保留站r和ROB项b。
  - 如果保留站或ROB全满，便停止流出指令，直到它们都有空闲的项。

- 执行
  - 如果有操作数尚未就绪，就等待，并不断地监测CDB。

    （检测RAW冲突）
  - 当两个操作数都已在保留站中就绪后，就可以执行该指令的操作。
- 写结果
  - 当结果产生后，将该结果连同本指令在流出段所分配到ROB项的编号放到CDB上，经CDB写到ROB以及所有等待该结果的保留站。
  - 释放产生该结果的保留站。
  - store指令在本阶段完成，其操作为：
    - 如果要写入存储器的数据已经就绪，就把该数据写入分配给该store指令的ROB项。
    - 否则，就监测CDB，直到那个数据在CDB上播送出来，才将之写入分配给该store指令的ROB项。

➢ 确认

对分支指令、store指令以及其它指令的处理不同：

- 其它指令（除分支指令和store指令）

  当该指令到达ROB队列的头部而且其结果已经就绪时，就把该结果写入该指令的目的寄存器，并从ROB中删除该指令。

- store指令

  处理与上面的类似，只是它把结果写入存储器。

- 分支指令
  - 当预测错误的分支指令到达ROB队列的头部时，清空ROB，并从分支指令的另一个分支重新开始执行。

    （错误的前瞻执行）
  - 当预测正确的分支指令到达ROB队列的头部时，该指令执行完毕。

例5.4 假设浮点功能部件的延迟时间为：加法2个时钟周期，乘法10个时钟周期，除法40个时钟周期。对于下面的代码段，给出当指令MUL.D即将确认时的状态表内容。

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

前瞻执行中MUL.D确认前，保留站和ROB的状态

名称	保留站							
	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	no							
Load2	no							
Add1	no							
Add2	no							
Add3	no							
Mult1	no	MUL.D	Mem[45+ Regs[R3]]	Regs[F4]			#3	
Mult2	yes	DIV.D		Mem[34+Regs[R2]]	#3		#5	

增加Dest字段，记录哪个ROB项将接收该保留站的结果。#x表示ROB项x中的数字字段。

项号	ROB				
	Busy	指令	状态	目的	Value
1	no	L.D F6, 34（R2）	确认	F6	Mem[34+Regs[R2]]
2	no	L.D F2, 45（R3）	确认	F2	Mem[45+Regs[R3]]
3	yes	MUL.D F0, F2, F4	写结果	F0	#2×Regs[F4]
4	yes	SUB.D F8, F6, F2	写结果	F8	#1－#2
5	yes	DIV.D F10, F0, F6	执行	F10	
6	yes	ADD.D F6,F8,F2	写结果	F6	#4＋#2

字段	浮点寄存器状态							
	F0	F2	F4	F6	F8	F10	…	F30
ROB项编号	3			6	4	5		
Busy	yes	no	no	yes	yes	yes	…	no

6. 前瞻执行

- 通过ROB实现了指令的顺序完成。
- 能够实现精确异常。
- 很容易地推广到整数寄存器和整数功能单元上。
- 主要缺点：所需的硬件太复杂。

5.5 多指令流出技术

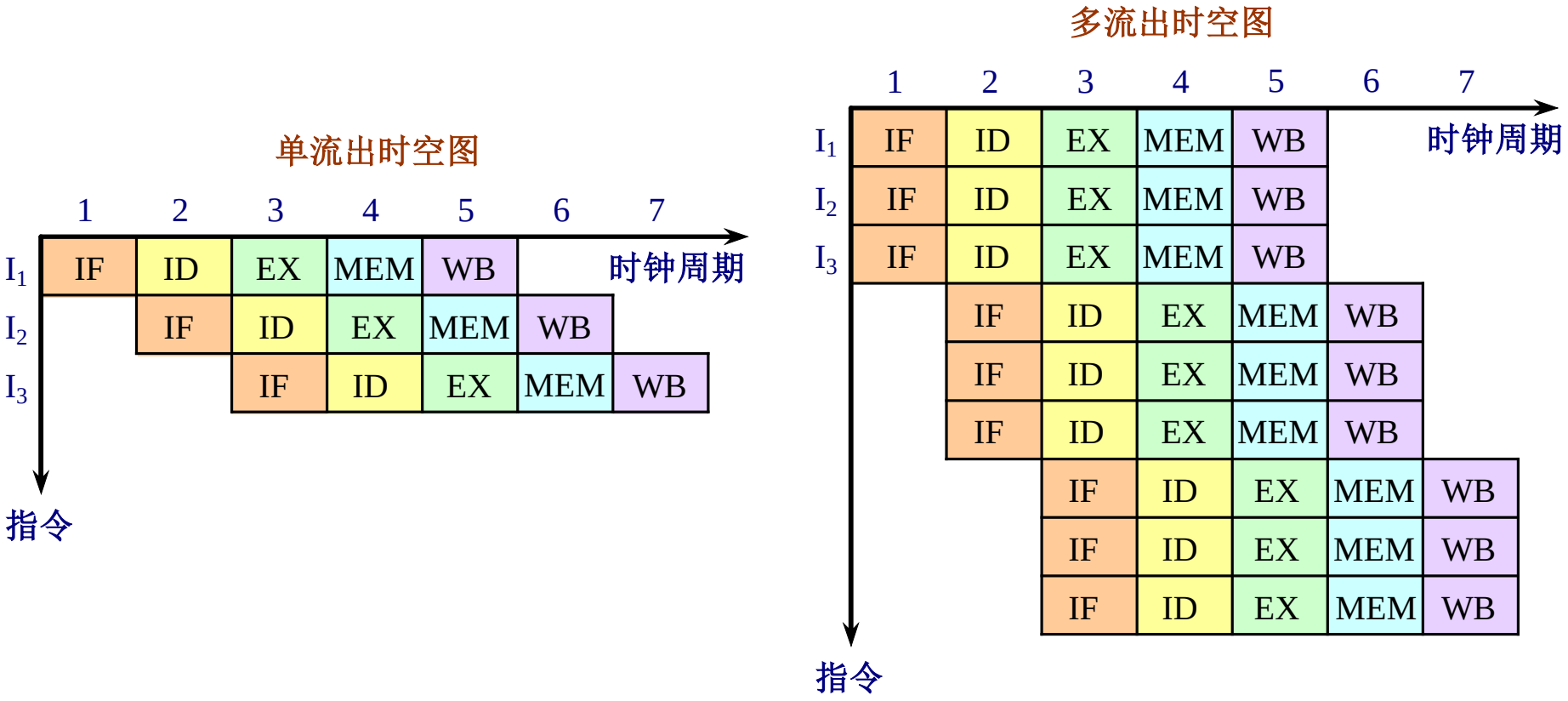
1. 工作动机

- 实际CPI=理想CPI+各种冲突引入的停顿
  - 前面讲的技术主要集中后者（消除/减少各种冲突引入的停顿）
  - 也可以考虑通过提升前者，改进实际CPI

- 前面讲的所有流水线都是单流出的，即一个时钟周期内至多流出一条指令，理想CPI=1

- 本节描述的技术：一个时钟周期内流出多条指令，理想CPI<1。

2. 单流出和多流出对比



单流出和多流出处理机执行指令的时空图

3. 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，也可以基于Tomasulo算法进行动态调度。

- **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。

4. 超标量处理机与VLIW处理机相比有两个优点：

- 超标量结构对程序员是透明的，处理机能自己检测下一条指令能否流出，不需要由编译器或专门的变换程序对程序中的指令进行重新排列；
- 即使是没有经过编译器针对超标量结构进行调度优化的代码或是旧的编译器生成的代码也可以运行，当然运行的效果不会很好。
  - 要想达到很好的效果，方法之一：

    使用动态超标量调度技术。

2. 下表列出了一些基本的多流出技术、这些技术的特点以及采用这些技术的处理机实例。

各种多流出技术

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（安腾）

5.5.1 基于静态调度的多流出技术

基于静态调度的超标量流水线技术

- 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。

- 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。

MIPS处理机实现超标量方法：

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；
   - 确定哪几条指令可以流出（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。

基于静态调度的超标量流水线技术

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

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。

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。
- 分支延迟
  - 如果分支指令是流出包中的第一条指令，则其直接影响后续3条指令；
  - 否则就是流出包中的第二条指令，其直接影响后续2条指令。

5.5.2 基于动态调度的多流出技术

扩展Tomasulo算法：支持双流出超标量流水线

- 每个时钟周期流出两条指令；
- 一条是整数指令，另一条是浮点指令。

1. 采用一种比较简单的方法：

   - 指令按顺序流向保留站，否则会破坏程序语义。
   - 将整数所用的表结构与浮点用的表结构分离开，分别进行处理，这样就可以同时流出一条浮点指令和一条整数指令到各自的保留站。

2. 实现多流出的两种的方法：

动态调度关键在于：对保留站的分配和对流水线控制表格的修改。

- 在半个时钟周期里完成流出步骤，这样一个时钟周期就能处理两条指令。

  --时间/流水

- 设置一次能同时处理两条指令的逻辑电路。

  --空间/加宽流出电路

现代的流出4条或4条以上指令的超标量处理机经常是两种方法都采用。

例5.5 对于采用了Tomasulo算法和多流出技术的MIPS流水线，考虑以下简单循环的执行。该程序把F2中的标量加到一个向量的每个元素上。

```
Loop: L.D     F0, 0（R1）       // 取一个数组元素放入F0
      ADD.D   F4, F0, F2        // 加上在F2中的标量
      S.D     F4, 0（R1）       // 存结果
      DADDIU R1,R1，#-8
                        // 将指针减少8（每个数据占8个字节）
      BNE R1,R2,Loop
              // 若R1不等于R2，表示尚未结束，转移到Loop继续。
```

参考的C代码：

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

现做以下假设：

- 每个时钟周期能流出2条指令，即使它们相关也是如此。
- 有一个整数部件，用于整数ALU运算和地址计算；并且对于每一种浮点操作类型都有一个独立的流水化了的浮点功能部件。
- 指令流出和写结果各占用一个时钟周期。
- 具有动态分支预测部件和一个独立的计算分支条件的功能部件。
- 跟大多数动态调度处理器一样，写回段的存在意味着实际的指令延迟会比按序流动的简单流水线多一个时钟周期。
- 从产生结果数据的源指令到使用该结果数据的指令之间的延迟为：

  整数运算一个周期；

  load两个周期（执行计算地址1+访存1）；

  浮点加法运算3个周期。

5.5 多指令流出技术

1. 请列出该程序前面3遍循环中各条指令的流出、开始执行和将结果写到CDB上的时间。

2. 如果分支指令单流出，没有采用延迟分支，但分支预测是完美的。请列出整数部件、浮点部件、数据Cache以及CDB的资源使用情况。

解：执行时，该循环将动态展开，并且只要可能就流出两条指令。表中列出了各指令执行到几个操作点的时间及资源的使用情况。

遍数	指　　令	流出	执行	访存	写CDB	说明
1	L.D　　F0,0（R1）	1	2	3	4	流出第一条指令
1	ADD.D　　F4,F0,F2	1	5		8	等待L.D的结果
1	S.D　　F4,0（R1）	2	3	9		等待ADD.D的结果
1	DADDIU　　R1,R1,#-8	2	4		5	等待ALU（S.D在用）
1	BNE　　R1,R2,Loop	3	6			等待DADDIU的结果
2	L.D　　F0,0（R1）	4	7	8	9	等待BNE完成
2	ADD.D　　F4,F0,F2	4	10		13	等待L.D的结果
2	S.D　　F4,0（R1）	5	8	14		等待ADD.D的结果
2	DADDIU　　R1,R1,#-8	5	9		10	等待ALU
2	BNE　　R1,R2,Loop	6	11			等待DADDIU的结果
3	L.D　　F0,0（R1）	7	12	13	14	等待BNE完成
3	ADD.D　　F4,F0,F2	7	15		18	等待L.D的结果
3	S.D　　F4,0（R1）	8	13	19		等待ADD.D的结果
3	DADDIU　　R1,R1,#-8	8	14		15	等待ALU
3	BNE　　R1,R2,Loop	9	16			等待DADDIU的结果

资源使用情况

时钟周期	整型ALU	浮点ALU	数据Cache	CDB
2	1/L.D			
3	1/S.D		1/L.D	
4	1/DADDIU			1/L.D
5		1/ADD.D		1/DADDIU
6				
7	2/L.D			
8	2/S.D		2/L.D	1/ADD.D
9	2/DADDIU		1/S.D	2/L.D
10		2/ADD.D		2/DADDIU
11				
12	3/L.D			

5.5 多指令流出技术

时钟周期	整型ALU	浮点ALU	数据Cache	CDB
13	3/S.D		3/L.D	2/ADD.D
14	3/DADDIU		2/S.D	3/L.D
15		3/ADD.D		3/DADDIU
16				
17				
18				3/ADD.D
19			3/S.D	
20				

可以看出：

- 每3个时钟周期就执行一个新循环，每个循环5条指令。

  IPC＝5/3＝1.67条/拍

- 虽然指令的流出率比较高，但是执行效率并不是很高。
  - 19拍共执行15条指令，
  - 平均指令执行速度为15/19＝0.79条/拍。
- 原因是浮点运算少，整型ALU部件成了瓶颈。

解决方法：增加一个加法器，把整型ALU功能和地址运算功能分开。

3. 上述双流出动态调度流水线的性能受限于以下3个因素：

- 整数部件和浮点部件的工作负载不平衡，没有充分发挥出浮点部件的作用。

  应该设法减少循环中整数型指令的数量。

- 每个循环叠代中的控制开销太大。
  - 5条指令中有两条指令是辅助指令；
  - 应该设法减少或消除这些指令。
- 控制相关使得处理机必须等到分支指令的结果出来后才能开始下一条L. D指令的执行。

5.5.3 超长指令字技术

（Very Long Instruction Word, VLIW）

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。

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。

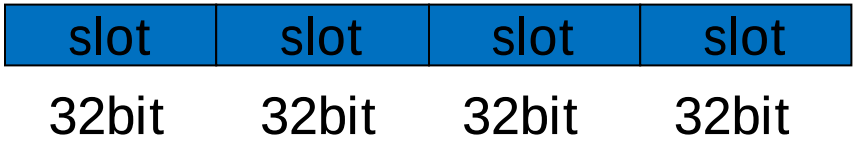
  解决：采用指令共享立即数字段的方法，或者采用指令压缩存储、调入Cache或译码时展开的方法。
- 采用了锁步机制

  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

VLIW技术：静态多指令流出

1. VLIW vs. 超标量

- 在动态调度的超标量处理器中，相关检测和指令调度基本都由硬件完成。
- 在静态调度的超标量处理器中，部分相关检测和指令调度工作交由编译器完成。
- 在VLIW处理器中，相关检测和指令调度工作全部由编译器完成，它需要更“智能”的编译器。



128bit VLIW指令

2. 实例分析

例 假设某VLIW处理器每个时钟周期可以同时流出5个操作，包括2个访存操作，2个浮点操作以及1个整数或分支操作。将前例中的代码循环展开，并调度到该VLIW处理器上执行。循环展开次数不定，但至少要能够保证消除所有流水线“空转”周期，同时不考虑分支延迟。

解：循环被展开7次，经调度后可以消除所有流水线“空转”。在不考虑分支延迟的情况下，每执行一个叠代需要9个时钟周期，计算出7个结果，即平均每得到一个结果需要1.29个周期。

VLIW的不足：

编码效率仅比50%略高一些

所需要的寄存器数量也大大增加

5.5 多指令流出技术

访存操作1	访存操作2	浮点操作1	浮点操作2	整数分支操作
L.D F0，0（R1）	L.D F6，-8（R1）	nop	nop	nop
L.D F10，-16（R1）	L.D F14，-24（R1）	nop	nop	nop
L.D F18，-32（R1）	L.D F22，-40（R1）	ADD.D F4，F0，F2	ADD.D F8，F6，F2	nop
L.D F26，-48（R1）	nop	ADD.D F12，F10，F2	ADD.D F16，F14，F2	nop
nop	nop	ADD.D F20，F18，F2	ADD.D F24，F22，F2	nop
S.D 0（R1），F4	S.D -8（R1），F8	ADD.D F28，F26，F2	nop	nop
S.D -16（R1），F12	S.D -24（R1），F16	nop	nop	nop
S.D -32（R1），F20	S.D -40（R1），F24	nop	nop	DADDUI R1，R1，#56
S.D 8（R1），F28	nop	nop	nop	BNE R1，R2，Loop

3. VLIW性能分析

- VLIW目标代码编码效率低
  - 为消除流水线“空转”需要增加循环展开的次数
  - 很难从应用程序中找到足够多的并行指令填满**VLIW**指令中的每一个**slot**
- VLIW流水线的互锁机制
  - **VLIW**处理器中没有实现任何相关检测逻辑，而是靠互锁机制保证执行结果的正确
  - 这种简单的互锁机制将造成较大的开销
- 目标代码兼容性差
  - 二进制翻译

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。

5.5.5 超流水线处理机（Superpipelining）

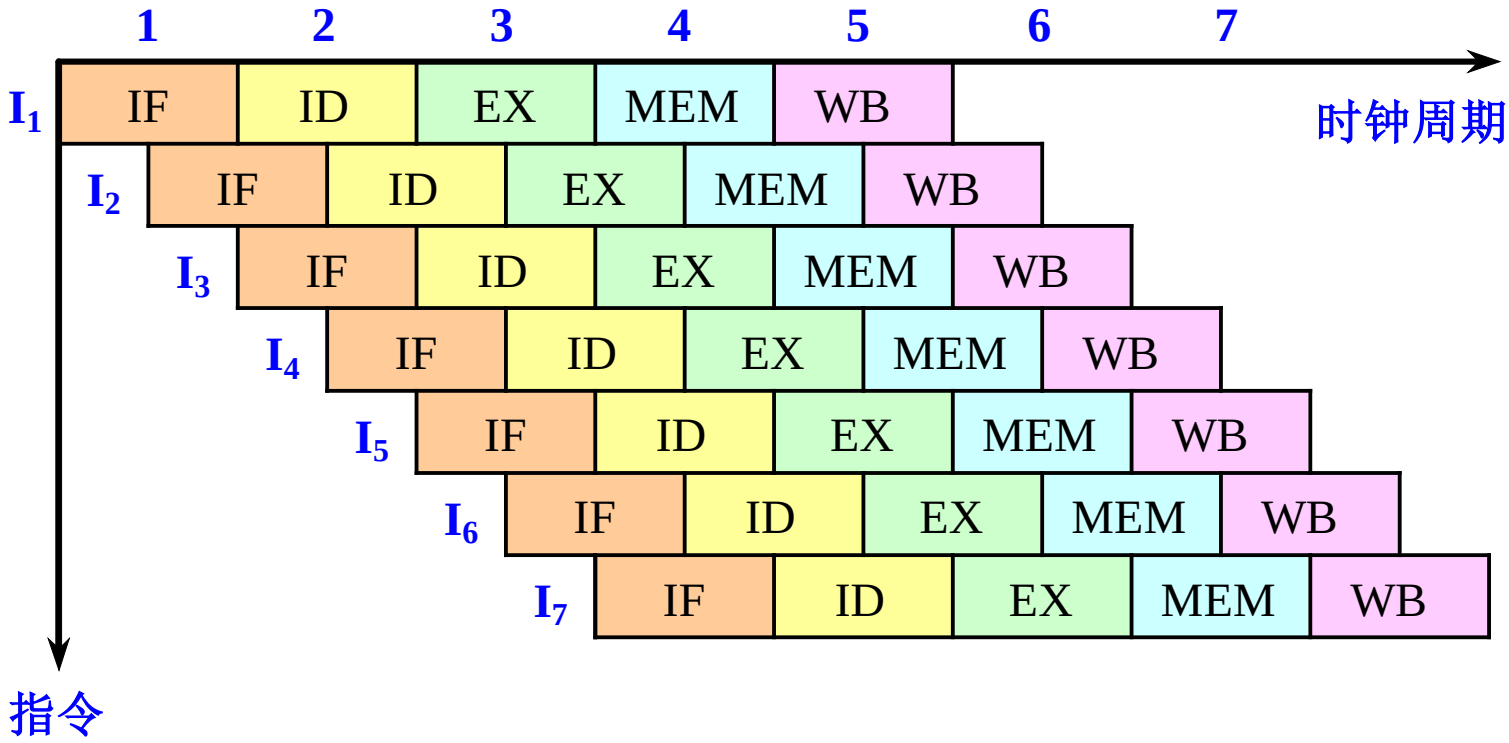
1. 将每个流水段进一步细分，这样在一个时钟周期内能够分时流出多条指令。这种处理机称为超流水线处理机。

超标量处理机： 空间并行性；

超流水线处理机：时间并行性。

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。



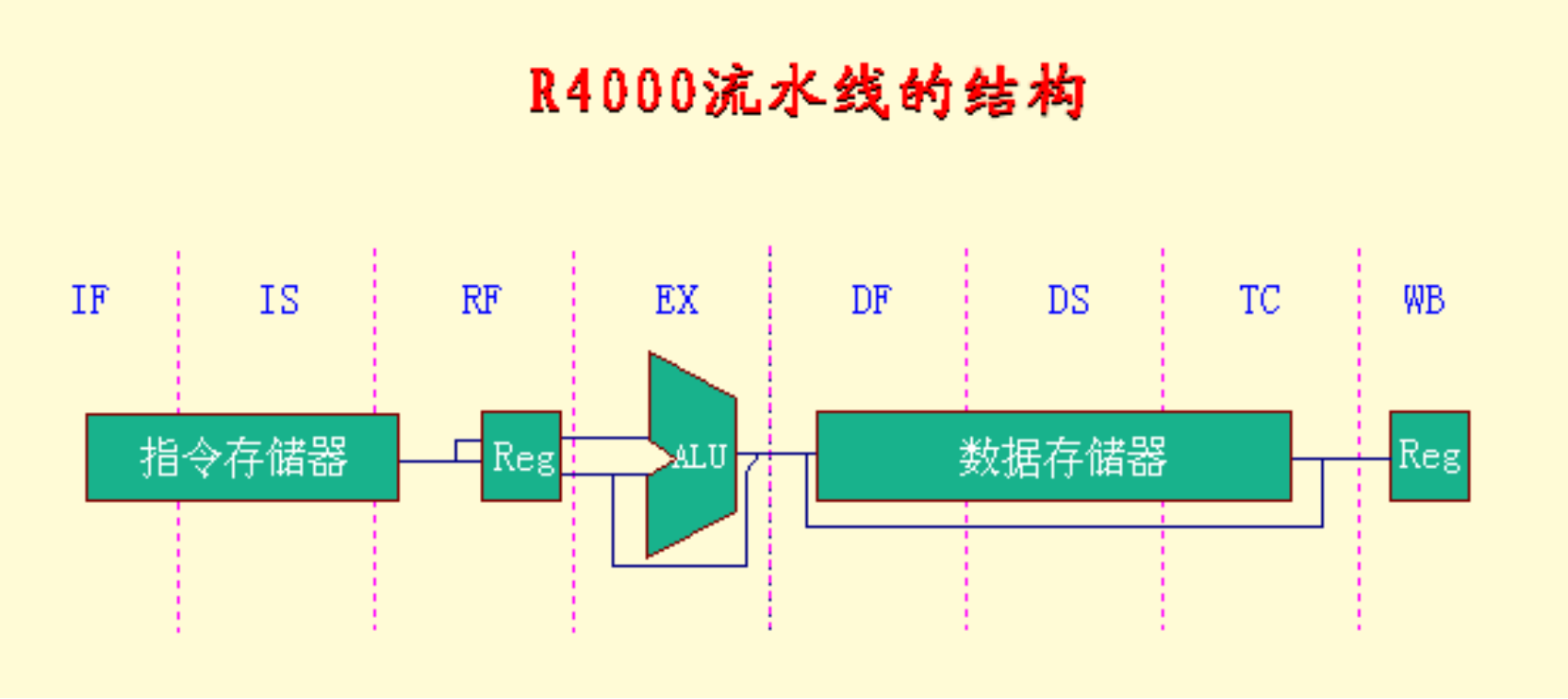
超流水线处理机时空图

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

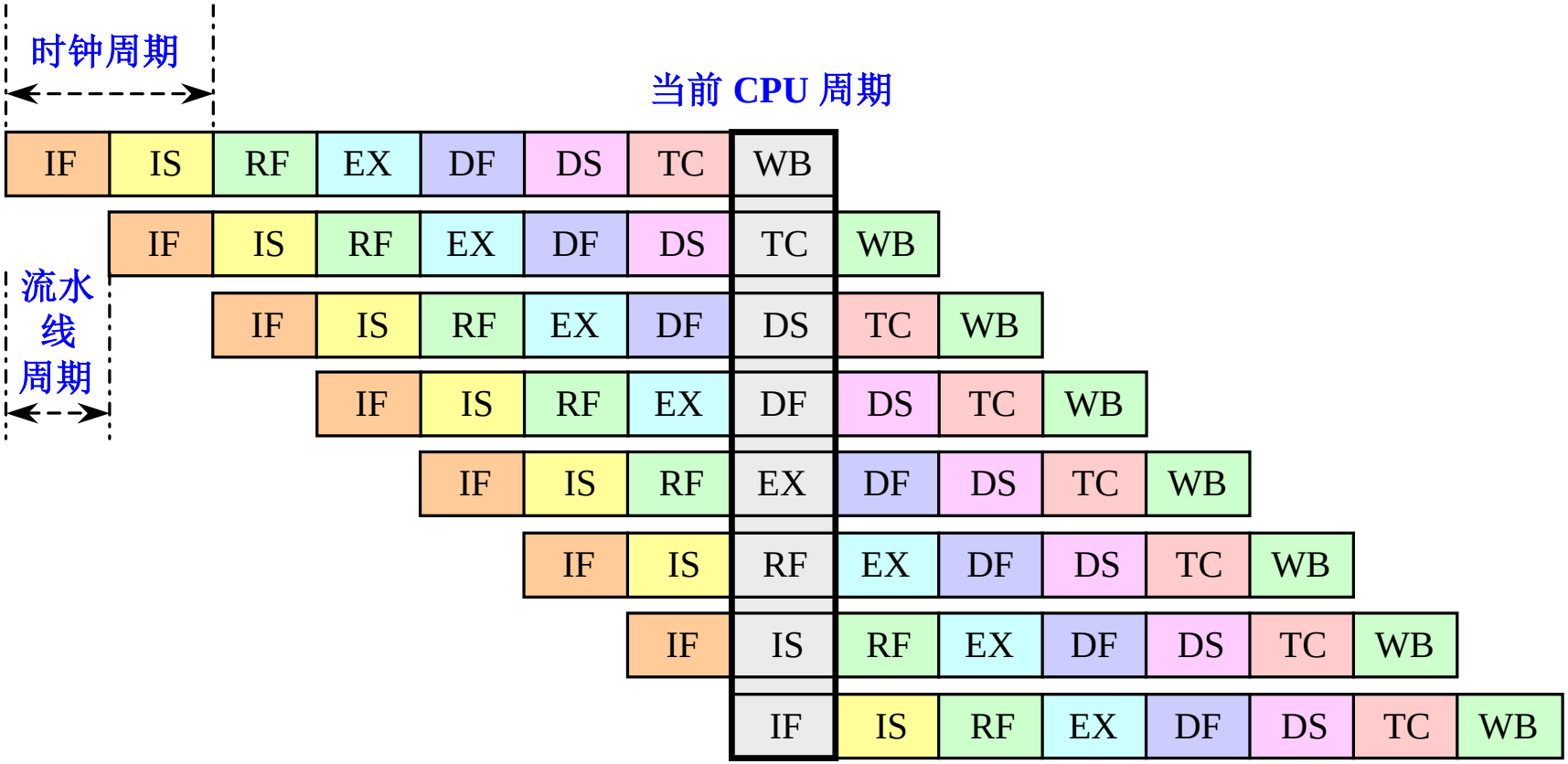
      （它们可以并行工作）
  - 一个16×64位的浮点通用寄存器组。浮点通用寄存器组也可以设置成32个32位的浮点寄存器。
- R4000的指令流水线有8级

5.5 多指令流出技术



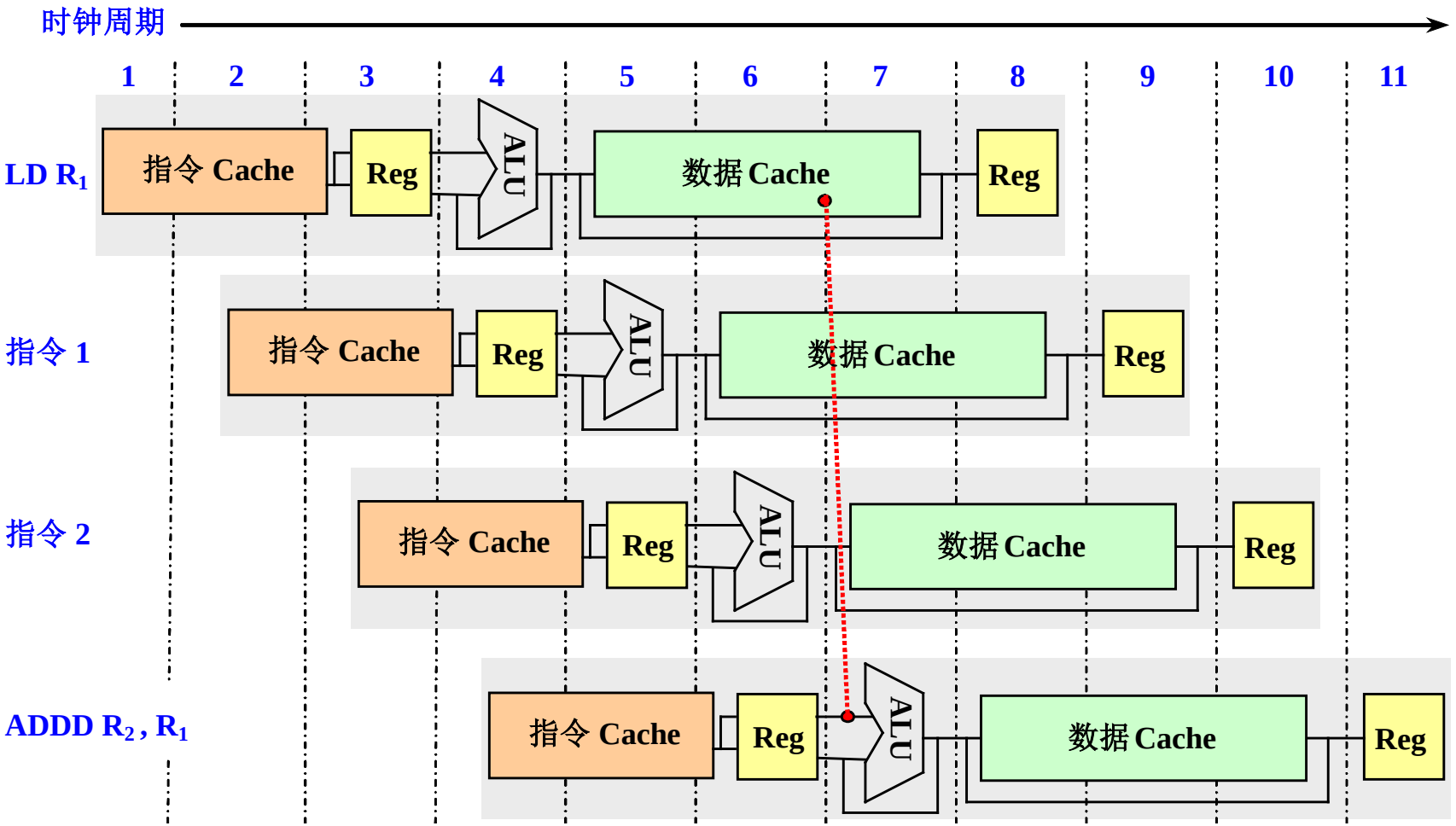
- 各级的功能
  - IF：取指令的前半步，根据PC值去启动对指令Cache的访问。
  - IS：取指令的后半步，在这一级完成对指令Cache的访问。
  - RF：指令译码，访问寄存器组读取操作数，冲突检测，并判断指令Cache是否命中。
  - EX：指令执行。包括：有效地址计算，ALU操作，分支目标地址计算，条件码测试。
  - DF：取数据的前半步，启动对数据Cache的访问。
  - DS：取数据的后半步，在这一级完成对数据Cache的访问。
  - TC：标识比较，判断对数据Cache的访问是否命中。
  - WB：load指令或运算型指令把结果写回寄存器组。

- MIPS R4000指令流水线时空图



- 载入延迟为两个时钟周期

对于load指令，数据要在DS末尾才准备好，紧跟其后的指令要使用需要等两个流水线周期（数据定向）。



Load指令引起的流水线停顿

5.6 指令调度及循环展开

5.6.1 指令调度的基本方法

软件方法--使用编译器的指令级并行开发方法

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

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/说明	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
浮点Store（S.D）	定向路径	0
整数指令（整数运算、分支、整数Load）	假设整数运算部件是全流水的或者重复设置了足够多的分数	1

例 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	时延
Loop:	L.D	F0, 0(R1)	1	(+1)
	DADDIU	R1, R1, #-8	2	(+1)
	ADD.D	F4, F0, F2	3	(+2)
	(空转)		4	
	BNE	R1, R2, Loop	5	(+1)
	S.D	F4, 8(R1)	6	(+0)

进一步分析：

- 编译时指令调度是怎样减少整个指令序列在流水线上的执行时间的？
- 指令调度能否跨越分支边界？
- 怎样提高整个执行过程中有效操作的比率？

5.6.2 循环展开

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：将上例中的循环展开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！

小结　指令级并行

1. 指令的动态调度

乱序执行

对于正确地执行程序来说，必须保持的最关键的两个属性是：数据流和异常行为

两种典型的动态调度算法:

记分牌算法和Tomasulo算法

2. 动态分支预测技术

BHT、BTB

基于硬件的前瞻执行:ROB

3. 多指令流出技术

超标量、超长指令字VLIW、超流水线处理机

小结　指令级并行

2. 动态分支预测技术

BHT、BTB

基于硬件的前瞻执行:ROB

3. 多指令流出技术

超标量处理机:　同时流出、静态调度+动态调度

超长指令字处理机:同时流出、静态调度

超流水线处理机:　分时流出

小结　指令级并行

四种不同类型处理机的性能比较

机器类型	k段流水线基准标量处理机	m度超标量处理机	n度超流水线处理机	(m,n)度超标量超流水线处理机
机器流水线周期	1个时钟周期	1	1 / n	1 / n
同时发射指令条数	1条	m	1	m
指令发射等待时间	1个时钟周期	1	1 / n	1 / n
指令级并行度ILP	1	m	n	m×n

基准标量处理机是一台普通的单流水线处理机。

另外三种指令级并行处理机：

并行度为m的超标量处理机，

并行度为n的超流水线处理机，

并行度为（m，n）的超标量超流水线处理机