第6章 指令级并行的开发
——软件方法

沈 立

www.GotoSchool.net

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

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？

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！

2. 循环展开和指令调度的注意事项

- 保证正确性
- 注意有效性
- 使用不同的寄存器
- 删除多余的测试指令和分支指令，并对循环结束代码和新的循环体代码进行相应的修正。
- 注意对存储器数据的相关性分析
- 注意新的相关性

6.2 跨越基本块的静态指令调度

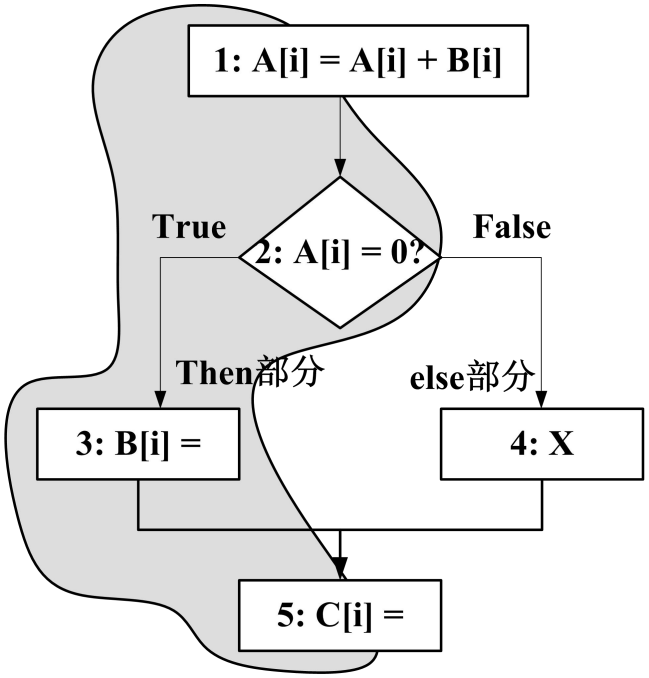
6.2.1 全局指令调度

1. 概述

- 目标：在保持原有数据相关和控制相关不变的前提下，尽可能地缩短包含分支结构的代码段的总执行时间。
  - 单流出处理器——减少指令数
  - 多流出处理器——缩短关键路径长度
- 基本思想：在循环体内的多个基本块间移动指令，扩大那些执行频率较高的基本块的体积。

1. 实例分析

- 由于分支条件为true(转移)的概率大，全局指令调度时会将语句1、2、3、5合并为一个更大的基本块。
  - 如何保证分支条件为**false**时结果依然正确？
  - 如何将语句**3**和**5**调度到语句**2**之前？



一个分支结构的代码段

- 将上图中的代码转换为下面的MIPS汇编指令

```
            LD      R4，0(R1)       // 取A
            LD      R5，0(R2)       // 取B
            DADDU   R4，R4，R5      // A=A+B
            SD      0(R1)，R4       // 存A
            BNEZ    R4，elsepart    // A=0则转移
            X                       // 代码段X，基本块elsepart
            J       join
thenpart：                          // 基本块thenpart
            SD      …，0(R2)        // 指令I1，对应语句3
join：
            SD      …，0(R3)        // 指令I2，对应语句5
```

- 调度指令I1
  - 直接将**I1**移到**BEQZ**前是否会产生错误结果？
  - 向基本块**elsepart**中增加**补偿代码**
  - 补偿代码有可能带来额外时间开销
- 调度指令I2
  - 将**I2**移动到基本块**thenpart**中，同时复制到**elsepart**中。
  - 若不影响执行结果，将**I2**调度到**BEQZ**前，同时删除**elsepart**中的副本。

3. 全局指令调度是一个很复杂的问题

以I1的调度为例：

- 需要确定分支中基本块thenpart和elsepart的执行频率各是多少？
- 在分支语句前完成I1所需的开销是多大？
- 调度I1是否能够缩短thenpart块的执行时间？
- I1是否是最佳的被调度对象？
- 是否需要向elsepart块中增加补偿代码，补偿代码开销如何？怎样生成补偿代码？

6.2.2 踪迹调度

1. 概述

- 踪迹（trace）：程序执行的指令序列，通常由一个或多个基本块组成，trace内可以有分支，但一定不能包含循环。
- 踪迹调度（trace scheduling）会优化执行频率高的trace，减少其执行开销。由于需要添加补偿代码以确保正确性，那些执行频率较低的trace的开销反而会有所增加。
- 踪迹调度非常适合多流出处理器。

1. 踪迹调度的步骤

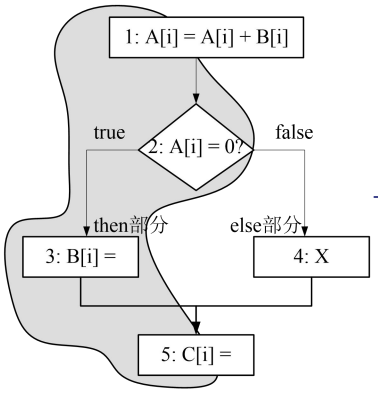
分为两步：踪迹选择和踪迹压缩

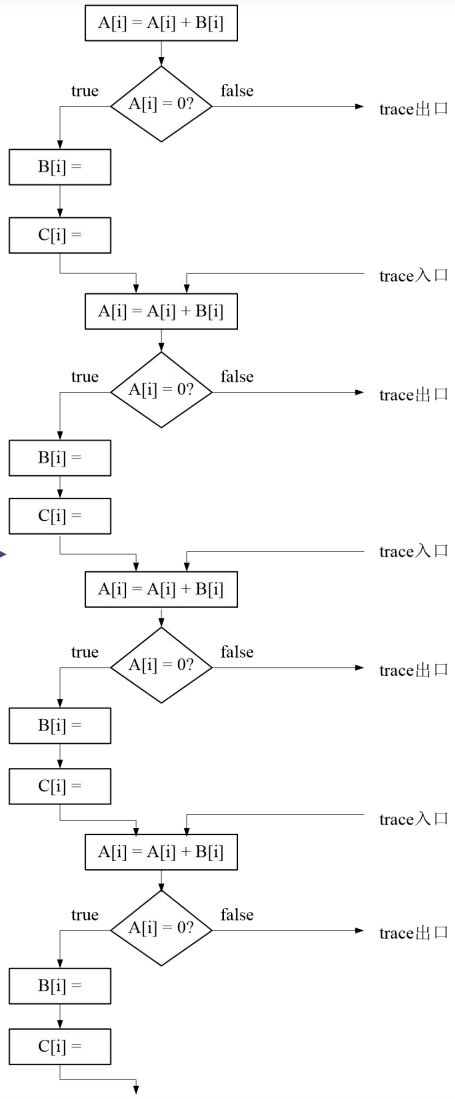
- 踪迹选择
  - 从程序的控制流图中选择执行频率较高的路径，每条路径就是一条trace。
  - 处理转移成功与失败概率相差较大的情况
  - 循环结构：循环展开
  - 分支结构：根据典型输入集下的运行统计信息

➢ 踪迹选择——实例分析

将左边的循环展开4次并把阴影部分(执行频率高)拼接在一起就可以得到一条trace；

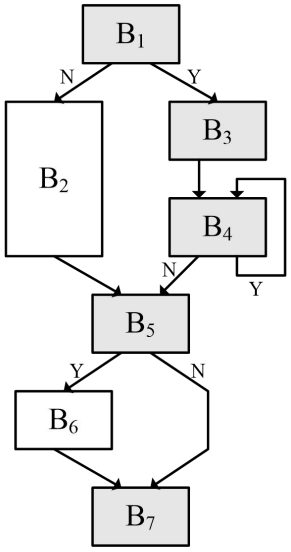
一条trace可以有多个入口和多个出口。





踪迹压缩

- 对已生成的trace进行指令调度和优化，尽可能地缩短其执行时间；
- 跨越trace内部的入口或出口调度指令时必须非常小心，有时还需要增加补偿代码 。



(a) 控制流图

压缩前	压缩后
B_1: x = x + 1 y = x - y if x<5 goto B_3	B_1: x = x + 1 if x<5 goto B_3
B_2: z = x * z y = y + 1 goto B_5	B_2: y = x - y z = x * z y = y + 1 goto B_5
B_3: y = 2 * y x = x - 2	B_3: y = x - y y = 2 * y x = x - 2

(b) 踪迹压缩前后的基本块B_1、B_2、B_3

三条trace：B_1-B_3、B_4以及B_5-B_7

指令“y = x - y”被从B_1调度到B_3中，跨越了trace的一个出口；

需要向块B_2中增加补偿代码，即将指令“y = x - y”复制到B_2的第一条指令之前 。

3. 踪迹调度的性能特点

- 踪迹调度能够提升性能的最根本原因在于选出的trace都是执行频率很高的路径，减少它们的执行开销有助于缩短程序的总执行时间。
- 对于某些应用，补偿代码引起的开销很有可能降低踪迹调度的优化效果。
- 踪迹调度会大大增加编译器的实现复杂度。

6.2.3 超块调度

1. 概述

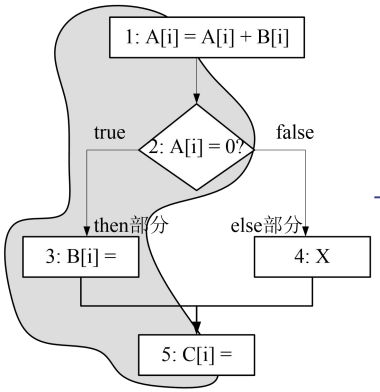
- 在踪迹调度中，如果trace入口或出口位于trace内部，编译器生成补偿代码的难度将大大增加，而且编译器很难评估这些补偿代码究竟会带来多少性能损失。
- 超块（superblock）是只能拥有一个入口，但可以拥有多个出口的结构
- 超块的构造过程与trace相似，但怎样确保只有一个入口？

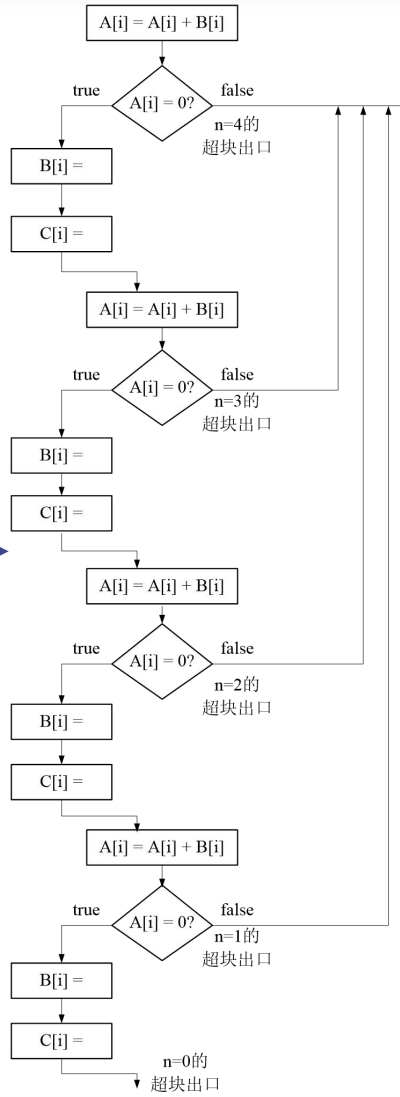
1. 超块构造——尾复制技术

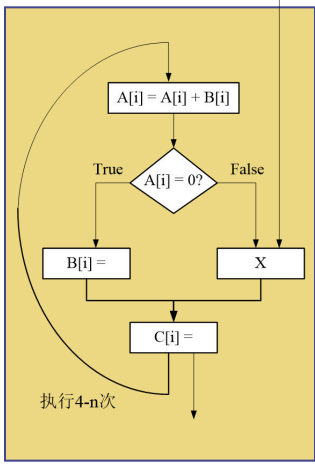
将左边的循环展开4次并把阴影部分(执行频率高)拼接在一起就可以得到一个超块；

超块有1个入口和5个出口(n=4/3/2/1/0)。

除了n=0外，从其他4个出口退出超块后，还需要继续完成余下的n次叠代(黄色部分)。







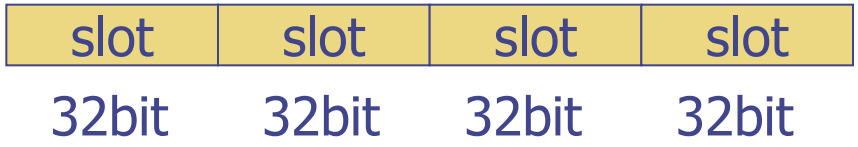
3. 超块调度的性能特点

- 尾复制技术简化了补偿代码的生成过程，并降低了指令调度的复杂度。
- 超块结构目标代码的体积也大大增加。
- 补偿代码的生成使得编译过程更加复杂，而且由于无法准确评估由补偿代码引起的时间开销，这限制方法超块调度的应用范围。

6.3 静态多指令流出：VLIW技术

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

1. 实例分析

例6.3 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。将例6.1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

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

4. 性能比较——多流出处理器 vs. 向量处理器

- 即使对于一些结构不规则的代码，多流出处理器也能从中挖掘出一些指令级并行。
- 多流出处理器对存储系统没有过高的要求，价格较便宜、由Cache和主存构成的多层次存储子系统即可满足其对性能的要求。

结论： 多流出处理器已成为当前实现指令级并行的主要选择，而向量处理器则通常是作为协处理器集成到计算机系统中，以加速特定类型的应用程序。

6.4 显示并行指令计算EPIC

- 超标量和VLIW结构都存在严重不足
  - 超标量硬件复杂度太高，8流出成为极限；
  - VLIW存在代码兼容问题，编译器智能程度不够。
- EPIC技术在VLIW基础上融合了超标量的一些优点
  - 编译器通过踪迹调度、超块调度等带有极强猜测性的优化技术尽可能多地挖掘指令级并行。
  - 流水线硬件则提供丰富的计算资源实现这些指令级并行，并通过专门的机制确保在程序执行过程中出现预测错误时仍然能得到正确的运行结果，尽量减少由此引起的额外开销。

- 什么是EPIC?
  - 指令级并行主要由编译器负责开发，处理器为保证代码正确执行提供必要的硬件支持，只有在这些硬件机制的辅助下这些优化技术才能高效完成。
  - 系统结构必须提供某种通信机制，使得流水线硬件能够了解编译器“安排”好的指令执行顺序。
- EPIC编译器的高级优化技术
  - 非绑定分支
  - 谓词执行
  - 前瞻执行

6.4.1 非绑定分支

1. 分支指令在传统流水线上的执行过程

- 计算分支转移条件
- 生成分支目标地址
- 取下一条指令
- 译码并流出下一条指令

在传统流水线上，分支指令都具有“原子性”，即上述各操作被绑定在一起，不能分开。

2. 非绑定分支技术

- 核心思想：将分支指令划分为多条粒度更小的指令，独立执行。
  - 准备操作：计算分支目标地址
  - 比较操作：计算分支转移条件
  - 转移操作：根据分支转移条件是**true**还是**false**，改变控制流或执行顺序的下一条指令。
- 运行时，流水线硬件根据前两个操作的结果，动态地将第三个操作转换为空操作或无条件转移。
  - 前两个操作应尽早完成

6.4.2 谓词执行

1. 条件执行机制

- 条件执行：指指令的执行依赖于一定的条件，当条件为真时指令将正常执行，否则将什么也不做。
  - 实例：条件传输指令

例6.4 在下面的语句中，

if（A=0）{S=T；}

假设变量A、S、T的值分别保存在寄存器R1、R2和R3内。请用分支指令和条件传输指令编写功能相同的汇编代码。

解：包含分支指令的MIPS汇编代码如下：

```
        BNEZ    R1，L
        ADDU    R2，R3，R0
    L：
```

而使用条件传输指令CMOVZ时的汇编代码为：

```
        CMOVZ  R2，R3，R1
```

指令CMOVZ有3个操作数，R2为目的操作数，R1和R3是源操作数，执行条件保存在寄存器R1中。

当R1=0时，R3的值被复制到R2中，否则R2的内容不变。

- 分析
  - 条件传输指令将分支指令引起的控制相关转换为相对于分支转移条件（R1）的数据相关。
  - 条件执行机制能够删除代码中那些行为难以预测的分支指令，提高分支预测准确率，并减少由于分支预测错误带来的性能损失。
  - 无论指令的执行条件是否为真，指令都将被读出、译码并执行。
  - 这种编译优化技术叫做条件转换。

1. 条件传输指令的应用——计算绝对值

- 求绝对值的运算A = abs(B)，对应的C语句为：

  if（B<0）{A=-B;} else {A=B;}

- 使用条件传输指令后，代码段如下（假设变量A和B分别被保存在寄存器R1和R2中）：

```
SUB  R1，R0，R2          // A = -B
SLT  R3，R2，R0          // 若B<0，R3=1，
                         // 否则R3=0
CMOVZ  R1，R2，R3        // R3=0时，A = B
```

1. 谓词执行机制

- 条件传输指令的性能问题
  - 随着指令数的增加，经过条件转换得到的条件传输指令和条件计算指令的数量也将增加，这会大大降低目标代码的效率。
- 谓词执行（predicated execution）
  - 给指令集中的每条指令都增加一个执行条件，这个执行条件就叫做谓词（predicate）。
  - 若谓词为真，指令正常执行，否则什么也不做。

1. 谓词执行机制——实例分析

例6.5 假设在一个周期内，某双流出的超标量处理器可以同时执行一个访存操作和一个ALU操作，或者仅执行一个分支操作。受此限制，下面这段汇编代码的执行效率并不高，表现在：

（1）第二个周期只能流出一条ALU指令，访存单元空闲；

（2）当分支转移不成功时，BEQZ指令后的两条LW指令之间存在的数据相关将引起流水线暂停。

试通过谓词执行机制解决这两个问题，减少此段代码的执行开销。

例6.5的代码段

周期	指令1	指令2
1	LW R1，40（R2）	ADD R3，R4，R5
2		ADD R6，R3，R7
3	BEQZ R10，L	
4	LW R8，0（R10）	
5	LW R9，0（R8）	

解 我们用LWC表示带谓词的LW指令，并假设该指令的执行条件为谓词不等于0。这样，BEQZ后的第一条LW指令就可以被转换为LWC指令，并被调度到第二个周期执行，如下所示：

周期	指令1	指令2
1	LW R1，40（R2）	ADD R3，R4，R5
2	LWC R8，20（R10），R10	ADD R6，R3，R7
3	BEQZ R10，L	
4	LW R9，0（R8）	

- 分析
  - 调度后代码的执行时间缩短了。
  - 若分支转移成功，**LWC**将被转换为空操作，这不影响结果的正确性，但也不会缩短执行时间。
  - 周期**4**中的**LW**指令转换为**LWC**，结果如何？

周期	指令1	指令2
1	LW R1，40（R2）	ADD R3，R4，R5
2	LWC R8，20（R10），R10	ADD R6，R3，R7
3	BEQZ R10，L	
4	LW R9，0（R8）	

1. 谓词执行机制——异常处理

- 谓词执行增加了异常处理的复杂度
- 例若LWC指令执行时发生缺页中断，如何处理？
  - **LWC的谓词为true，中断本应发生；**
  - **LWC的谓词为false，中断不应发生。**

周期	指令1	指令2
1	LW R1，40（R2）	ADD R3，R4，R5
2	LWC R8，20（R10），R10	ADD R6，R3，R7
3	BEQZ R10，L	
4	LW R9，0（R8）	

1. 如何将谓词为假的指令转换为空操作？
   - 两种方法
     - 流水线前端指令流出时
     - 流水线后端结果确认时
   - 一般采用第二种方法
     - 为什么？

6.4.3 前瞻执行

1. 概述

- 谓词执行与全局指令调度的不足之处
  - 仅有少量结构全面实现谓词执行
  - 全局指令调度往往需要补偿代码
- EPIC通过前瞻执行提高猜测执行的效果
- 什么是前瞻执行？
  - 在数据相关或控制相关尚未消除时，将指令调度到相关指令前猜测执行。
  - 通过硬件机制完成异常处理，确保正确性。

- 影响前瞻执行效果的因素
  - 编译器能力的高低：能否准确识别可以被前瞻执行的指令。
  - 异常处理机制：能否推迟处理由被前瞻执行的指令引起的异常，直到确定前瞻指令确实被执行后。
  - 如何避免前瞻引起的错误？

1. 实例分析

例6.6 下面是一个if-then-else结构的C程序段以及相应的MIPS汇编代码段，其中变量A和B分别被保存在地址为0(R3)和0(R2)的存储单元中。若分支转移不成功的概率很大，请利用前瞻执行技术将第二条LD指令调度到分支指令BNEZ前执行。假设寄存器R14空闲。

C语句：

```
if （A≠0）A=A+4；else  A=B；
```

汇编指令：

```
    LD  R1，0（R3）              // 取A
    BNEZ       R1，L1           //（A≠0）？
    LD  R1，0（R2）              // A=B（else部分）
    J   L2
L1：DADDI R1，R1，#4            // A=A+4(then部分)
L2：SD    R1，0（R3）           // 存A
```

解 调度结果如下：

```
        LD      R1, 0（R3）          // 取A
        sLD     R14, 0（R2）         // 取B，前瞻执行
        BEQZ    R1, L3
        DADDI R14, R1, #4            // A=A+4
L3:     SD      R14     , 0（R3）    // A=B
```

1. 前瞻执行——异常处理机制

问题：若执行sLD时发生异常应该如何处理？

- 有四种方法
- 终止性异常 vs 可继续异常

➢ 方法一： 立即处理

- 此前瞻指令引起的异常只是简单地返回一个未定义值即可，而不是立即结束程序的运行。
- 前瞻正确时，正在执行的程序不会被终止，但它的执行结果肯定是错误的。
- 前瞻错误时，程序也将继续执行下去，只是处理该异常的返回值不会被使用。

- 方法二：借助专门的检测指令判断是否需要处理
  - 代码实例

```
LD      R1，0（R3）          // 取A
sLD R14，0（R2）             // 取B，前瞻执行
BEQZ    R1，L1
SPECCK  0（R2）              // sLD是否产生异常
J       L2
L1：DADDI   R14，R1，#4      // A=A+4
L2：SD R14，0（R3）          // A=B
```

  - SPECCK的执行条件与sLD指令相同（分支转移失败）
  - 基本思想：推迟处理由前瞻指令sLD引发的异常，直到已确定该指令确实应该被执行 。

- 方法三：借助寄存器状态位判断是否需要处理
  - 为每个通用寄存器增加一个特殊的状态标志位：“poison”位
  - 前瞻指令引发的可继续异常都将被立即处理。
  - 前瞻指令引发终止性异常时，其目的寄存器R的poison位将被置1，否则该位将被清0。
  - 当前瞻指令之后的另一条指令访问R时，若R的poison位为1将触发一个终止性异常。
  - 基本思想：将前瞻指令引起异常的处理推迟到另一条指令访问前瞻指令的目的寄存器时。

- 方法四：借助再定序缓冲器完成
  - 将指令的执行结果保存在再定序缓冲器内，并按指令流出的顺序依次确认。
  - 前瞻指令的确认时机被推迟，直至能够确定该指令的前瞻执行是正确（或错误）的。
  - 除了需要再定序缓冲器等硬件机制的支持外，也需要在编译时标出所有被前瞻的指令，以及这些指令所跨越的条件分支。

4. 控制前瞻

- 将load调度到store之前前瞻执行最常见的数据前瞻
  - 为保证正确性，编译器总是会选择保守的调度方法，认为相邻的store与load间存在地址冲突，但在很多情况下，地址并不冲突。
  - 尽早完成load指令有助于缩短关键路径的长度。
- 硬件地址冲突检测机制
  - 当load指令前瞻执行时，流水线硬件会将它访存的地址记录在一个特殊的地址表中。

- 每执行一条store指令，流水线硬件将该指令的访存地址与地址表中的各有效项进行匹配，命中则说明出现地址冲突，前瞻失败。
- 若控制流抵达load指令原来所在的位置时未出现冲突（编译时在此位置放置检测指令），说明前瞻成功，流水线硬件从地址表中删除对应的项。

➢ 数据前瞻失败时的处理

- 仅**load**指令被前瞻执行：由检测指令重新执行**load**指令即可
- 数据相关于**load**的其他指令也被前瞻执行：需要补偿代码

6.5 开发更多的指令级并行

why?

- 全局指令调度技术已经能够很好地处理由分支指令引起的控制相关
  - 容易预测的——前瞻执行
  - 不容易预测的——谓词执行
- 指令间的数据相关对指令级并行开发的限制作用反而越来越大

6.5.1 挖掘更多的循环级并行

1. 循环携带相关

- 循环携带相关是指一个循环的某个叠代中的指令与其他叠代中的指令之间的数据相关。

例6.7 在下面的循环中，

```
for (i=1; i<=100; i=i+1) {
    A[i+1] = A[i] + C[i];          /* S1 */
    B[i+1] = B[i] + A[i+1];        /* S2 */
}
```

假设数组A、B和C中所有元素的存储地址都互不相同，请问语句S1与S2之间存在哪些数据相关？

解 S1和S2之间存在两种不同类型的数据相关：

- 循环携带RAW数据相关：相邻连词叠代的语句S1之间，相邻两次叠代中的语句S2之间。
- RAW数据相关：同一叠代内的语句S2与S1之间。

分析:

- 循环携带相关迫使指令只能按照所在叠代的先后顺序依次执行。
- 限制了同一叠代内存在数据相关的各语句之间的相对顺序。

➢ 怎样消除循环携带数据相关？

例6.8 在下面的循环中，语句S1和S2之间存在哪些数据相关？该循环的各次叠代是否可以并行执行？如果不能，请修改其代码，使之可以并行。

```
for (i=1; i<=100; i=i+1) {
        A[i] = A[i] + B[i];          /* S1 */
        B[i+1] = C[i] + D[i];        /* S2 */
}
```

解 第i次叠代中语句S1与第i-1次叠代中语句S2之间存在RAW类型的循环携带数据相关，但它们之间没有形成环(S2与上次叠代的S1不相关)。修改后代码

```
A[1] = A[1] + B[1];
for (i=1; i<=99; i=i+1) {
    B[i+1] = C[i] + D[i];          /* 原S2 */
    A[i+1] = A[i+1] + B[i+1];      /* 原S1 */
}
B[101] = C[100] + D[100];
```

修改方法：将存在循环携带相关的各条指令放在同一个叠代中

- 复杂循环携带数据相关的处理

```
for（i=6；i<=100；i=i+1）
    Y[i] = Y[i-5] + Y[i];    // 相关距离为5
for（i=2；i<=100；i=i+1）
    Y[i] = Y[i-1] + Y[i];    // 相关距离为1
```

编译器必须检测出这种递归关系

(1) 某些系统结构（特别是向量计算机）为递归提供了专门的硬件支持

(2) 这样的递归结构中通常隐藏着大量的循环级并行

1. 存储别名分析

- 什么是存储别名
  - 一个元素可能同时拥有多个合法的地址表达式
  - A[i+5]、A[j*2-6]、A[k]
- 数组是仿射的
  - 如果一个一维数组A[m:n]的下标可以被表示为形如a×i+b的形式，那么就称该数组是仿射的（affine）。
  - 一个多维数组，如果它每一维的下标都是仿射的，那么它就是仿射的。

- GCD测试法
  - 算法描述
    - 如果GCD(c, a)可以整除(d-b)，那么有可能存在存储别名。
    - 如果GCD测试的结果为假（不能整除），那么一定没有存储别名存在。

例6.9 使用GCD测试方法判断下面的循环中是否存在存储别名。

```
for（i=1；i<=100；i=i+1）
    x[2*i+3] = x[2*i] * 5.0；
```

解 在这个循环中，a=2，b=3，c=2，d=0，

那么GCD(a, c)=2，而d-b=-3。

由于2不能整除-3，因此没有存储别名，即无论i取何值，x[2*i+3]与x[2*i]都将表示数组x的不同元素。

1. 数据相关分析

- 除了检测指令之间是否存在数据相关外，编译器还会将识别出的数据相关进一步细分为真数据相关、输出相关和反相关等不同类型，以便利用不同的优化技术消除这些相关。
- 常用的优化有重命名、值传播、高度消减等。

➢ 重命名优化实例

例6.10 找出下面循环中的所有数据相关，指出它们究竟是真数据相关、输出相关、还是反相关，并利用重命名技术消除其中的输出相关和反相关。

```
for (i=1; i<=100; i=i+1) {
        Y[i] = X[i] / a;          /* S1 */
        X[i] = X[i] + a;          /* S2 */
        Z[i] = Y[i] + a;          /* S3 */
        Y[i] = a  -  Y[i];        /* S4 */
}
```

解 这4条语句中存在以下相关：

1. S3与S1和S4与S1之间分别存在真数据相关。
2. S2和S1之间存在反相关。
3. S3和S4之间存在反相关。
4. S1和S4之间存在输出相关。

```
for (i=1; i<=100; i=i+1) {
    Y[i] = X[i] / a;          /* S1 */
    X[i] = X[i] + a;          /* S2 */
    Z[i] = Y[i] + a;          /* S3 */
    Y[i] = a - Y[i];          /* S4 */
    }
```

将原代码变换为下面的形式，可以消除所有输出相关和反相关。

```
for (i=1; i<=100; i=i+1) {
        /* 将数组Y重命名为T以消除输出相关 */
        T[i] = X[i] / c;
        /* 将数组X重命名为X1以消除反相关 */
        X1[i] = X[i] + c;
        /* 将Y重命名为T以消除反相关 */
        Z[i] = T[i] + c;
        Y[i] = c  -  T[i];
}
```

- 值传播优化实例
  - 优化前代码

```
DADDUI R1，R2，#4
DADDUI R1，R1，#4
```

  - 优化后代码

```
DADDUI R1，R2，#8
```

  - 值传播优化通过将变量替换为已知的值或表达式以达到消除数据相关

- 高度消减优化实例
  - 目的：缩短数据流图中关键路径的长度
  - 优化前代码I

```
ADD     R1，R2，R3      /* I1 */
ADD     R4，R1，R6      /* I2 */
ADD     R8，R4，R7      /* I3 */
```

  - 优化后代码I

```
ADD     R1，R2，R3
ADD     R4，R6，R7
ADD     R8，R1，R4
```

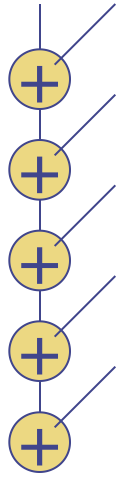
高度消减优化实例

- 优化前代码II

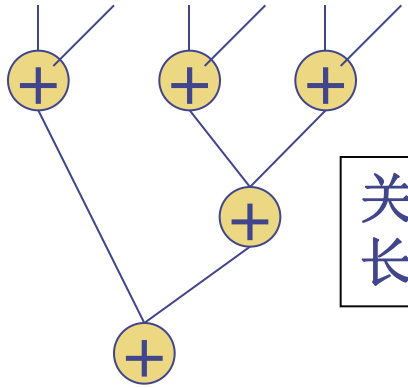
```
sum = sum + X[i];
sum = sum + X[i+1];
sum = sum + X[i+2];
sum = sum + X[i+3];
sum = sum + X[i+4];
```

- 优化后代码II

```
sum = sum + X[i];
t1 = X[i+1] + X[i+2];
t2 = X[i+3] + X[i+4];
t1 = t1 + t2;
sum = sum + t1;
```



关键路径
长度为5

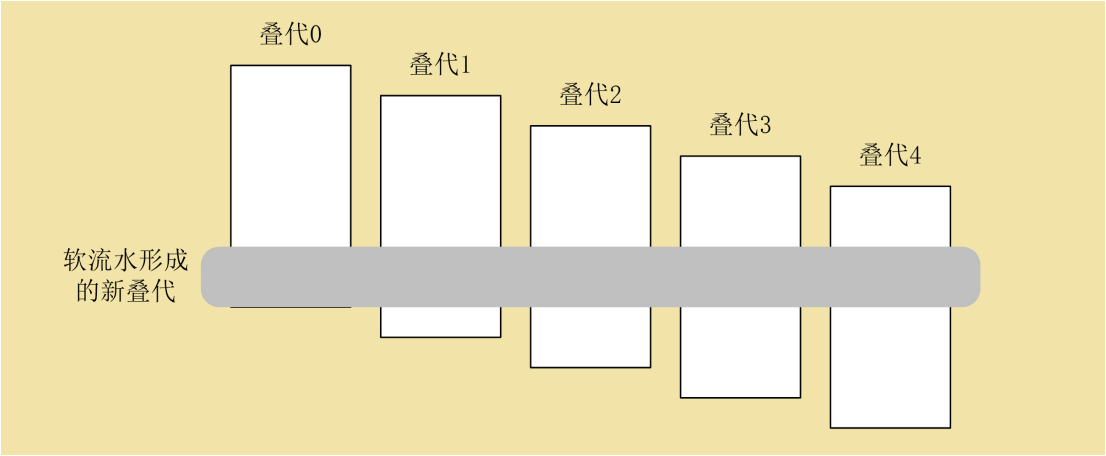


关键路径
长度为3

6.5.2 软流水

1. 简介

- 软流水技术的核心思想是从循环不同的叠代中抽取一部分指令（循环控制指令除外）拼成一个新的循环叠代。
- 目的
  - 将同一叠代中的相关指令分布到不同的叠代中
  - 将不同叠代中的相关指令封装到同一叠代中



1. 实例分析

例6.11 试用软流水技术处理例6.1中的循环，假设数组x有n个元素。

解 软流水需要从原循环的多个叠代中选择指令拼成新的循环，因此我们首先将原循环展开。

```
叠代i:      ; 修改x[i]并保存
            L.D     F0, 0(R1)
            ADD.D   F4, F0, F2
            S.D     F4, 0(R1)
叠代i+1:    ; 修改x[i-1]并保存
            L.D     F0, 0(R1)
            ADD.D   F4, F0, F2
            S.D     F4, 0(R1)
叠代i+2:    ; 修改x[i-2]并保存
            L.D     F0, 0(R1)
            ADD.D   F4, F0, F2
            S.D     F4, 0(R1)
```

从这3个叠代中分别选出一条指令，与原有的循环控制指令拼在一起得到一个新的叠代。

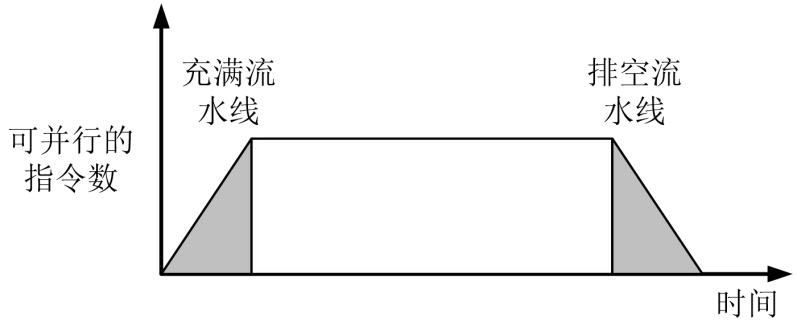
```
        DADDUI      R1, R1, #-16   // I1: R1保存x[n-2]的地址
        L.D   F0, 16(R1)           // I2: 取x[n]
        ADD.D F4, F0, F2           // I3: x[n] = x[n] + F2
        L.D   F0, 8(R1)            // I4: 取x[n-1]
Loop:   S.D   F4, 16(R1)           // I5: 存x[i+2]
        ADD.D F4, F0, F2           // I6: x[i+1] = x[i+1] +
        F2
        L.D   F0, 0(R1)            // I7: 取x[i]
        BNE   R1, R2, Loop         // I8
        DADDUI      R1, R1, #-8    // I9: 填充分支延迟槽
        S.D   F0, 8(R1)            // I10: 存x[2]
        ADD.D F4, F0, F2           // I11: x[1] = x[1] + F2
        S.D   F4, 0(R1)            // I12: 存x[1]
```

分析：

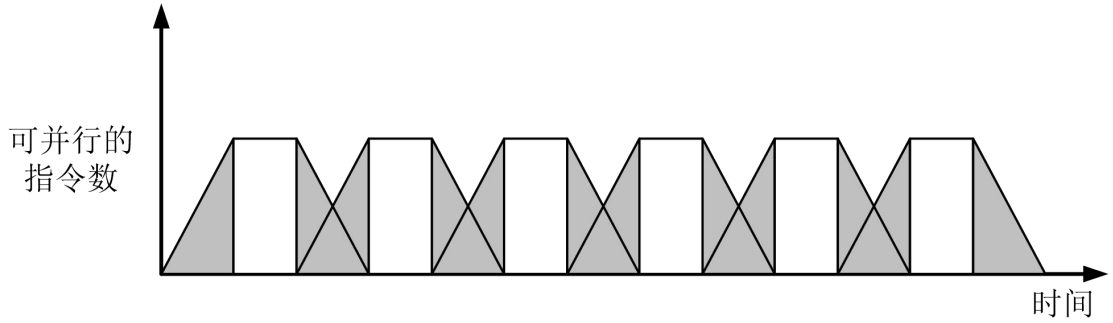
- 新循环的结束条件仍为：R1=R2
- 新循环从元素x[n-2]开始从头处理的，元素x[n]与x[n-1]只是从半途开始处理。
- 新循环的最后一次叠代只处理完元素x[3]，x[2]刚被修改完结果尚未写回，x[1] 刚被取出，需要被修改并写回。
- 新循环的每个叠代相当于流水执行了原循环的3个叠代。元素x[n/n-1/2/1]的处理相当于充满和排空这条“流水线”。

1. 性能比较：软流水 vs. 循环展开

- 循环展开主要减少由分支指令和修改循环索引变量的指令所引起的循环控制开销。
- 软流水使叠代内的指令级并行达到最大。



（a）软流水



（b）循环展开

6.6 实例：IA-64体系结构

1. IA-64简介

- IA-64是Intel与HP合作研制出的64位EPIC体系结构，其设计遵循以下原则：
  - 按照EPIC的思想开发指令级并行；
  - 提供大量的硬件资源实现指令级并行；
  - 提供一系列辅助软件指令级并行开发技术的硬件机制。

2. Intel IA-64产品——Itanium

- 主频：800MHz
- 采用3级Cache
  - 第一级为分离的指令和数据Cache
  - 后两级为混合Cache
  - 第三级Cache（在片外）
- 流水线共分为10级，其中实现了一些基于硬件的动态指令调度机制，如分支预测、重命名、记分牌等。
- 提供了丰富的功能单元，所有的功能单元都是流水的。
- 每个周期最多流出6条指令，最多可同时执行3条分支指令和2个访存操作，提高了分支处理和存储访问的能力。

- 整数性能(SPECint 2000)
  - 主频为800MHz的Itanium性能仅为2GHz主频Pentium 4的60%，1GHz主频Alpha 21264的68%。
  - Alpha 21264主频的功耗比Itanium低20%。
- 浮点性能(SPECfp 2000)
  - 800MHz Itanium的性能是2GHz Pentium 4的1.04倍，是1GHz Alpha 21264的1.20倍。

3. Intel IA-64产品——Itanium 2
   - 其性能在不经过任何调试和优化的条件下比Itanium提高50%到100%。

6.6.1 IA-64指令格式

IA-64功能单元和操作的类型

功能单元类型	操作类型	描述	操作实例
I-unit	A	整数ALU运算	加，减，与，或，比较
	I	非ALU整数运算	整数和多媒体移位、位测试、移动
M-unit	A	整数ALU运算	加，减，与，或，比较
	M	访存操作	整数/浮点load, store操作
F-unit	F	浮点操作	各种浮点运算
B-unit	B	分支操作	条件分支、无条件分支、调用
L+X	L+X	扩展操作	空操作、扩展的立即数

- IA-64的目标代码被划分为多个指令组，并按照流出的时间顺序依次排列，编译器指出组边界。
- IA-64的指令被称为“bundle”，长128位，包含123位操作信息（3个操作×41位/操作=123位）和5位模板信息。
- 模板位的作用
  - 简化译码，模板信息指明了bundle中3个操作的类型以及它们分别需要在哪类功能单元上执行。
  - 显式指出指令组边界，模板中定义了停止位。

IA-64模板值及其含义

模板值	操作槽0	操作槽1	操作槽2
0	M	I	I
1	M	I	I（s）
2	M	I（s）	I
3	M	I（s）	I（s）
4	M	L	X
5	M	L	X（s）
8	M	M	I
9	M	M	I（s）
10	M（s）	M	I
11	M（s）	M	I（s）
12	M	F	I
13	M	F	I（s）

模板值	操作槽0	操作槽1	操作槽2
14	M	M	F
15	M	M	F（s）
16	M	I	B
17	M	I	B（s）
18	M	B	B
19	M	B	B（s）
22	B	B	B
23	B	B	B（s）
24	M	M	B
25	M	M	B（s）
28	M	F	B
29	M	F	B（s）

例6.12 将例6.1中的代码展开7次并生成IA-64指令，使得：

(1) bundle数最少；

(2) 执行时间最短。

假设每时钟周期可流出1个bundle，bundle中任何操作的暂停将导致整个流水线暂停。各操作延迟如表6.1所示。

解 调度结果中“s”表示停止位，nop表示空操作。

- 第一种情况下只有9个bundle，15%的槽为空，但需要21个周期才能完成。
- 第二种情况代码体积和空槽均有所增加，需要11个bundle，有30%的槽是空槽，但执行时间大大缩短，仅用12个周期就可以完成 。

解 （1）bundle最少

模板	操作槽0	操作槽1	操作槽2	执行时间
9：M M I	L.D F0，0（R1）	L.D F6，-8（R1）	nop (s)	1
14：M M F	L.D F10，-16（R1）	L.D F14，-24（R1）	ADD.D F4，F0，F2	3
15：M M F	L.D F18，-32（R1）	L.D F22，-40（R1）	ADD.D F8，F6，F2 (s)	4
15：M M F	L.D F26，-48（R1）	S.D F4，0（R1）	ADD.D F12，F10，F2 (s)	6
15：M M F	S.D F8，-8（R1）	S.D F12，-16（R1）	ADD.D F16，F14，F2 (s)	9
15：M M F	S.D F16，-24（R1）	nop	ADD.D F20，F18，F2 (s)	12
15：M M F	S.D F20，-32（R1）	nop	ADD.D F24，F22，F2 (s)	15
15：M M F	S.D F24，-40（R1）	nop	ADD.D F28，F26，F2 (s)	18
15：M M F	S.D F28，-48（R1）	DADDUI R1，R1，#-56	BNE R1，R2，Loop	21

(2) 执行时间最短

模板	操作槽0	操作槽1	操作槽2	执行时间
8：M M I	L.D F0，0（R1）	L.D F6，-8（R1）	nop	1
9：M M I	L.D F10，-16（R1）	L.D F14，-24（R1）	nop (s)	2
14：M M F	L.D F18，-32（R1）	L.D F22，-40（R1）	ADD.D F4，F0，F2	3
14：M M F	L.D F26，-48（R1）	nop	ADD.D F8，F6，F2	4
15：M M F	nop	nop	ADD.D F12，F10，F2 (s)	5
14：M M F	nop	S.D F4，0（R1）	ADD.D F16，F14，F2	6
14：M M F	nop	S.D F12，-16（R1）	ADD.D F20，F18，F2	7
15：M M F	nop	S.D F8，-8（R1）	ADD.D F24，F22，F2 (s)	8
14：M M F	nop	S.D F16，-24（R1）	ADD.D F28，F26，F2	9
9：M M I	S.D F20，-32（R1）	S.D F24，-40（R1）	nop (s)	11
8：M M I	S.D F28，-48（R1）	DADDUI R1，R1，#-56	BNE R1，R2，Loop	12

6.6.2 IA-64的谓词执行机制

- IA-64提供的硬件支持
  - 设置了大量的谓词寄存器（64个），每个谓词寄存器的宽度都是1位。
  - 所有类型的IA-64操作都可以按照谓词执行方式执行，每个操作41位编码的最低6位指明了保存执行条件的谓词寄存器。
  - 增强了比较（compare）操作和测试（test）操作的功能，都可以同时修改多个谓词寄存器。

IA-64所支持的不同比较模式

模式	指令后缀	操作	
		目的谓词寄存器1	目的谓词寄存器2
Normal	无	if (qp) {target=result}	if (qp) {target=!result}
Unconditional	unc	if (qp) {target=result} else {target=0}	if (qp) {target=!result} else {target=0}
AND	and	if (qp && !result) {target=0}	if (qp && !result) {target=0}
	andcm	if (qp && result) {target=0}	if (qp && result) {target=0}
OR	or	if (qp && result) {target=1}	if (qp && result) {target=1}
	orcm	if (qp && !result) {target=1}	if (qp && !result) {target=1}
DeMorgan	or.andcm	if (qp && result) {target=1}	if (qp && result) {target=0}
	and.orcm	if (qp && !result) {target=0}	if (qp && !result) {target=1}

例6.13 将下面的C语句转换为IA-64汇编指令，使其执行时间最短。

```
if （r1==1 || r2==2 || r3==3 || r4==4）s;
```

解 对应的IA-64汇编代码如下。

IA-64 cmp指令的格式为：cmp.op.mod p, q = r1, r2

这里op为要进行的比较操作，mod为上表所示的比较模式中的一种，p，q为两个目的谓词寄存器，r1和r2为要比较的操作数，可以是寄存器名，也可以是立即数。

```
cmp.ne p1, p0 = r0, r0 ; ;
cmp.eq.or p1, p0 = 1, r1
cmp.eq.or p1, p0 = 2, r2
cmp.eq.or p1, p0 = 3, r3
cmp.eq.or p1, p0 = 4, r4     ; ;
（p1） s
```

6.6.3 IA-64的前瞻执行机制

- 为了标识被前瞻执行的load指令并检测前瞻是否成功，IA-64专门设置了3条指令。
  - **ld.a**：表示被前瞻执行的load指令
  - **ld.c**：编译器将ld.c指令放在被前瞻执行的load指令最初所在的位置，ld.c执行则前瞻成功
  - **chk.a**：编译器将它放在与ld.c相同的位置。若执行chk.a时发现对应的load前瞻执行失败，它会将控制流转移到一段补偿代码中，重新执行。
- 前瞻load指令的地址被记录在ALAT表中
- 每个寄存器有一个状态位NaT(Not A Thing)