

第6章 指令级并行及其开发— 软件方法

2023年春季学期

指令级并行

第 3 页

- **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。
- 开发ILP的方法可以分为两大类
 - 基于硬件的**动态开发**方法
 - 基于软件的**静态开发**方法
- 硬件指令调度——解决WAR、WAW相关问题

假设ADD操作需要2个时钟周期

ADD R3, R1, R2
AND R5, R3, R4



真相关



至少延迟一个时钟周期

- 本章研究：如何利用各种技术来开发更多的指令级并行（软件方法）

- 6. 1 [基本指令调度及循环展开](#)
- 6. 2 [跨越基本块的静态指令调度](#)
- 6. 3 [静态多指令发射：VLIW技术](#)
- 6. 4* [显式并行指令计算EPIC](#)
- 6. 5* [开发更多的指令级并行](#)
- 6. 6* [实例：IA-64体系结构](#)

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

第 6 页

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

6.1.1 指令调度的基本方法

第 7 页

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
```

6.1.1 指令调度的基本方法

第 8 页

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：

指令发射时钟		
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

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

完成一个循环体共10个周期，其中5个空转周期

6.1.1 指令调度的基本方法

第 9 页

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：

指令发射时钟		
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

完成一个循环体共6个周期，
其中1个空转周期

注意：（1）把S.D指令放到了分支指令的延迟槽中；

（2）由于修改指令的DADDIU指令（被调度到S.D指令之前），提前对指令进行了减8操作，所以要对S.D指令中的偏移量进行修正，即把“0(R1)”改成“8(R1)”。

（3）编译时指令调度不能消除指令间的相关，而是通过重新安排指令的发射顺序（如S.D、DADDIU指令）尽可能少地引起流水线空转，进而缩短整个指令序列的执行时间。

6.1.1 指令调度的基本方法

第 10 页

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

问题1：在基本指令调度方法中，指令调度不能跨越分支指令。这里需要说明的是，S.D指令被调度到BNE指令的分支延迟槽中不属于跨跃分支指令的情况；

问题2：指令序列中，只有L.D、S.D、ADD.D是有效操作，占3个周期，其他指令为控制循环与解决数据相关等待而附加的，整个指令序列的有效操作比例不高， $3/6=50\%$ 。

如何进一步缩短整个指令序列的执行时间：

- 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？

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，并且不重复使用寄存器。

消除名相关

Loop: LD F0, 0(R1)
ADDD F4, F0, F2
SD 0(R1), F4
LD F0, -8(R1)
ADDD F4, F0, F2
SD -8(R1), F4
LD F0, -16(R1)
ADDD F4, F0, F2
SD -16(R1), F4
LD F0, -24(R1)
ADDD F4, F0, F2
SD -24(R1), F4
SUBI R1, R1, #32
BNEZ R1, Loop

Register Renaming

Loop: LD F0, 0(R1)
ADDD F4, F0, F2
SD 0(R1), F4
LD F6, -8(R1)
ADDD F8, F6, F2
SD -8(R1), F8
LD F10, -16(R1)
ADDD F12, F10, F2
SD -16(R1), F12
LD F14, -24(R1)
ADDD F16, F14, F2
SD -24(R1), F16
SUBI R1, R1, #32
BNEZ R1, Loop

6.1.2 循环展开

第 13 页

展开后没有调度的代码如下（需要分配寄存器）

```
Loop:      LD      F0, 0(R1)
           Stall
           ADDD    F4, F0, F2
           Stall
           Stall
           SD      0(R1), F4 ; drop SUBI&BNEZ
           LD      F6, -8(R1)
           Stall
           ADDD    F8, F6, F2
           Stall
           Stall
           SD      -8(R1), F8 ; drop SUBI&BNEZ
           LD      F10, -16(R1)
           Stall
           ADDD    F12, F10, F2
           Stall
           Stall
           SD      -16(R1), F12 ; drop SUBI&BNEZ
           LD      F14, -24(R1)
           Stall
           ADDD    F16, F14, F2
           Stall
           Stall
           SD      -24(R1), F16
           SUBI    R1, R1, #32
           Stall
           BNEZ    R1, Loop
           Stall
```

注意：

（1）重新分配寄存器F0、F4、F6、F8等（2）DADDIU、L.D和S.D的指令的偏移量进行相应的修正；

（3）这个循环共执行28个时钟周期，平均每个循环体用 $28/4=7$ 个时钟周期；

（4）实际指令14条，空转13个时钟周期，执行效率不高。

6.1.2 循环展开

第 14 页

调度后的代码如下：

这个循环共执行 14 个时钟周期，平均每个循环体用 $14/4=3.5$ 个时钟周期

指令发射时钟			
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！

■ 循环展开和指令调度的注意事项

- 保证**正确性**：循环控制和操作数偏移量的修改
- 注意**有效性**：只有找到不同循环体之间的无关性，才能有效使用循环展开
- 使用不同的寄存器：重新分配寄存器，以避免产生新的冲突
- 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，如新的WAW、WAR等

6.2 跨越基本块的静态指令调度

6.2.1 全局指令调度

第 17 页

■ 循环展开存在问题

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

} 循环体是顺序结构

如果循环体是分支结构，则优化含有分支结构的循环体需要在多个基本块间移动指令，这种调度技术被称为“全局指令调度”，主要包括：

- 踪迹调度
- 超块调度

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。
 - 单发射处理器——减少指令数
 - 多发射处理器——缩短关键路径长度
 - ✓ 关键路径 (critical path)：根据指令间的相关关系构成的数据流图中延迟最长的路径。

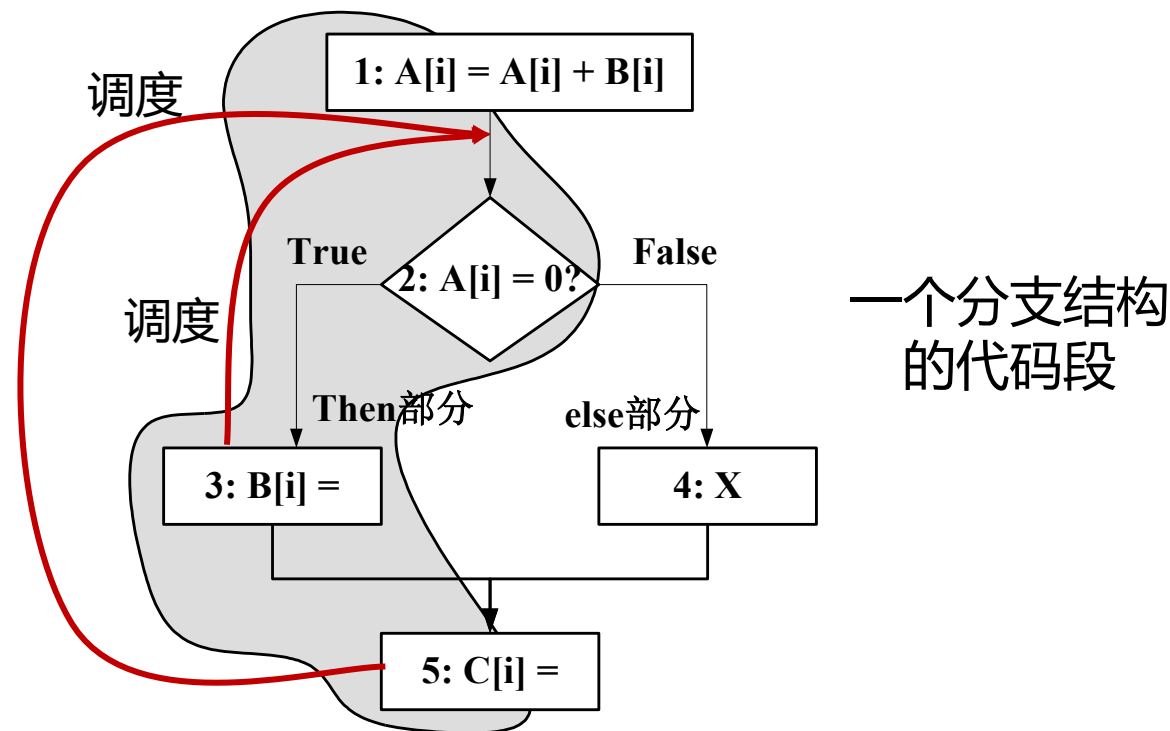
在减少指令数和缩短关键路径长度的基础上，如何进一步利用全局指令调度优化总执行时间？

6.2.1 全局指令调度

第 19 页

2. 实例分析

- 分析指令转移成功和失败的概率一般不同
- 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。



基本思想：（1）优化执行频率较高的基本块；（2）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。

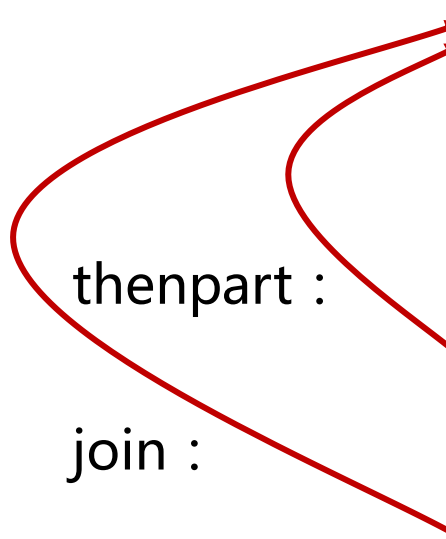
- 如何保证分支条件为false时结果依然正确？
- 如何将语句3和5调度到语句2之前？

6.2.1 全局指令调度

第 20 页

- 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 , thenpart // 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
```

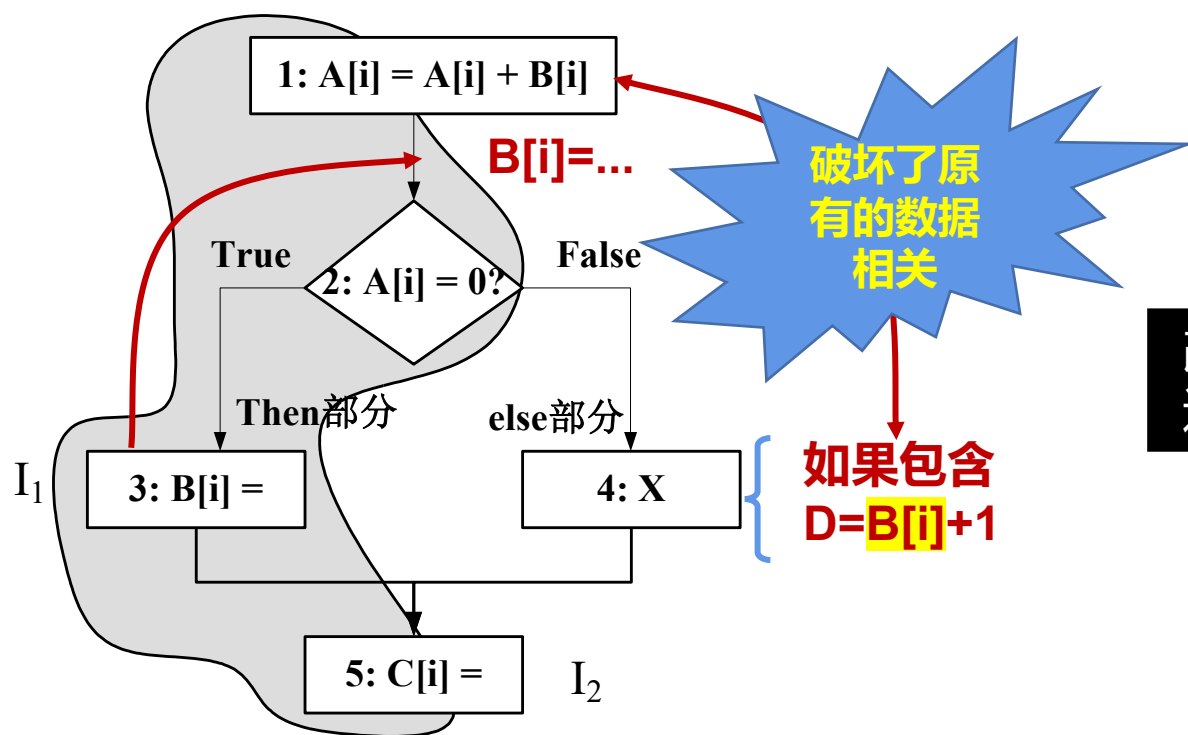


6.2.1 全局指令调度

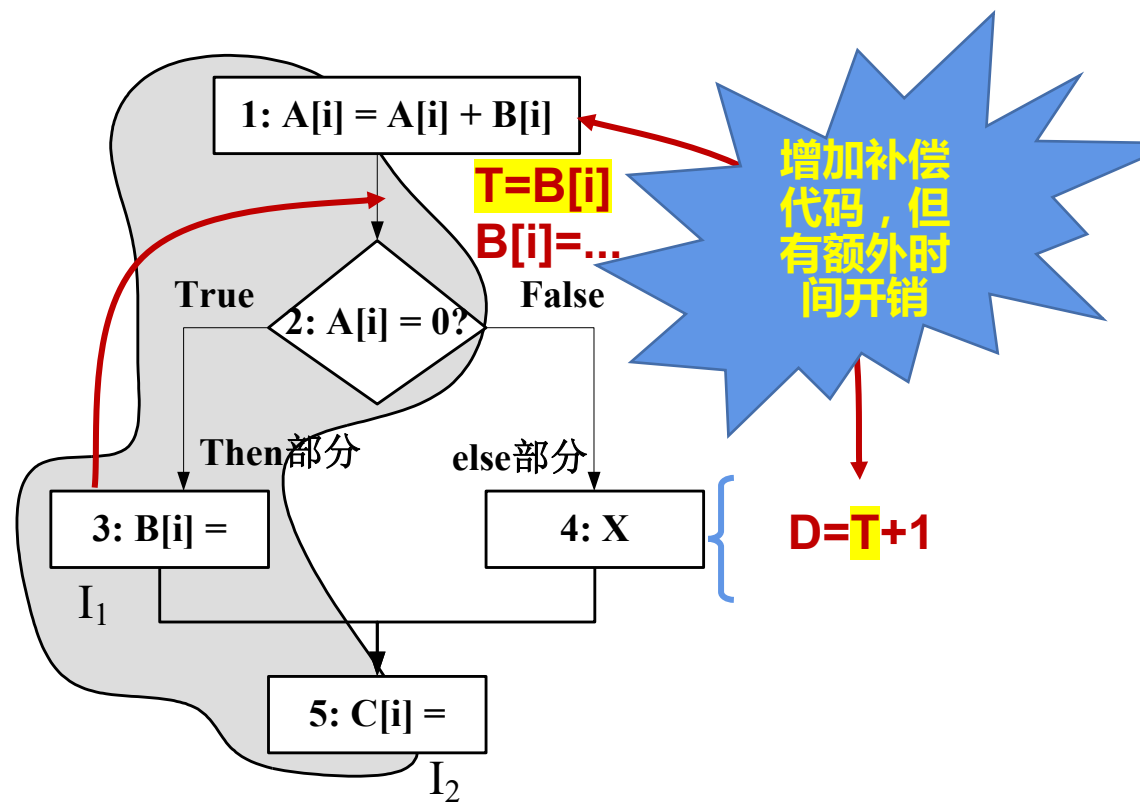
第 21 页

■ 调度指令 I_1 ：直接将 I_1 移到BEQZ前

- 是否会产生错误结果？
- 需不需要向基本块elsepart中增加补偿代码？
- 有没有可能带来额外时间开销？



改进

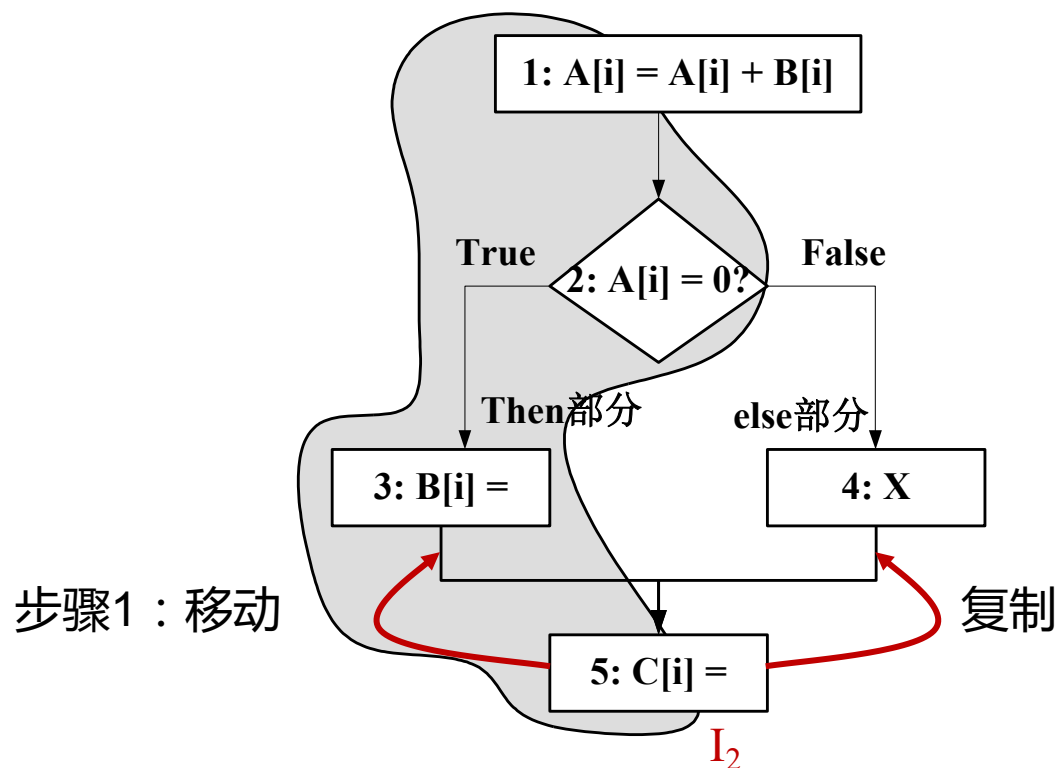


6.2.1 全局指令调度

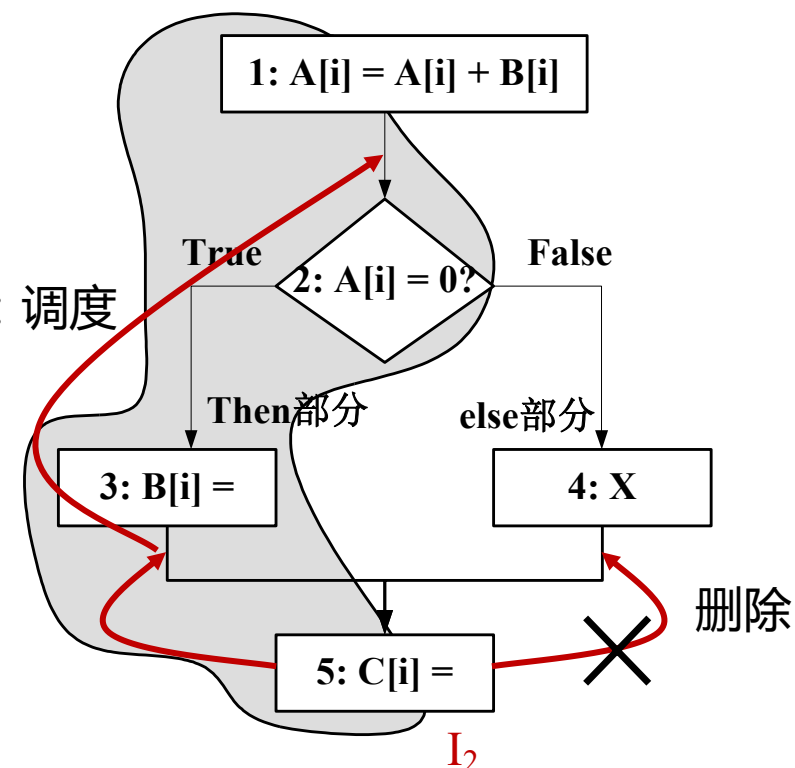
第 22 页

■ 调度指令 I_2

- 将指令 I_2 移动到基本块 thenpart 中，同时复制到 elsepart 中。
- 若不影响执行结果，将 I_2 调度到 BEQZ 前，同时删除 elsepart 中的副本。



步骤2：调度



6.2.1 全局指令调度

第 23 页

3. 全局指令调度是一个很复杂的问题，以 I_1 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？
 - 如果thenpart执行频繁，则调度 I_1 可以提升效率，反之可能降低
- 在分支语句前完成 I_1 所需的**开销**是多大？
 - 如果分支语句前有一些“空转”周期，则 I_1 可以被调度到这样周期内，开销为0
- 调度 I_1 是否能够缩短thenpart块的执行时间？
 - 如果 I_1 是关键路径的第一条语句，则调度该指令可以减少执行开销
- I_1 是否是最佳被调度对象？调度 I_2 或thenpart内的其他指令是否获得**更大性能提升**。
- 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？

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踪迹，减少其执行开销。由于存在多个踪迹，踪迹调度非常适合多发射处理器。
- 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踪迹的开销反而会有所增加，所以仅当不同踪迹的执行频率差别较大且各条踪迹的执行频率受输入集的影响较小（补偿代码小）时，才能取得较好的效果。

2. 踪迹调度的步骤

分为两步：踪迹选择和踪迹压缩

■ 踪迹选择

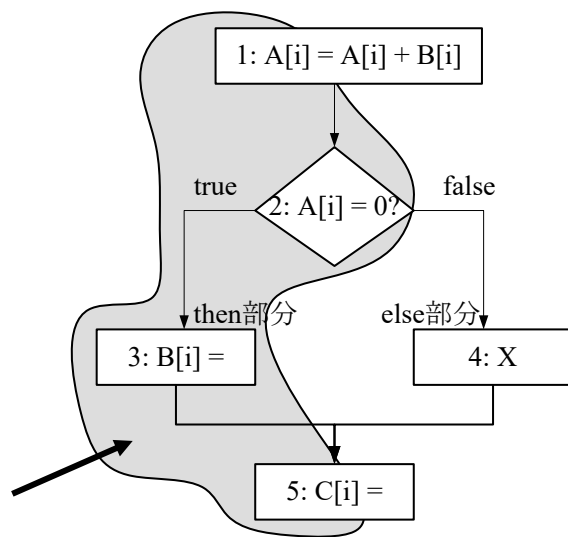
- 作用：从程序的控制流图中选择执行频率较高的路径，每条路径就是一条踪迹；
- 原则：处理转移成功与失败概率相差较大的情况；相近的情况可以采用谓词执行技术进行优化；
- 方法一：如果是循环结构，则采用循环展开，一般循环体的执行效率高于循环体外的基本模块的执行效率；
- 方法二：如果是分支结构，则根据典型输入集下的运行统计信息，若转移成功（或失败）的概率高，则视作转移总是成功（或失败）。

6.2.2 踪迹调度

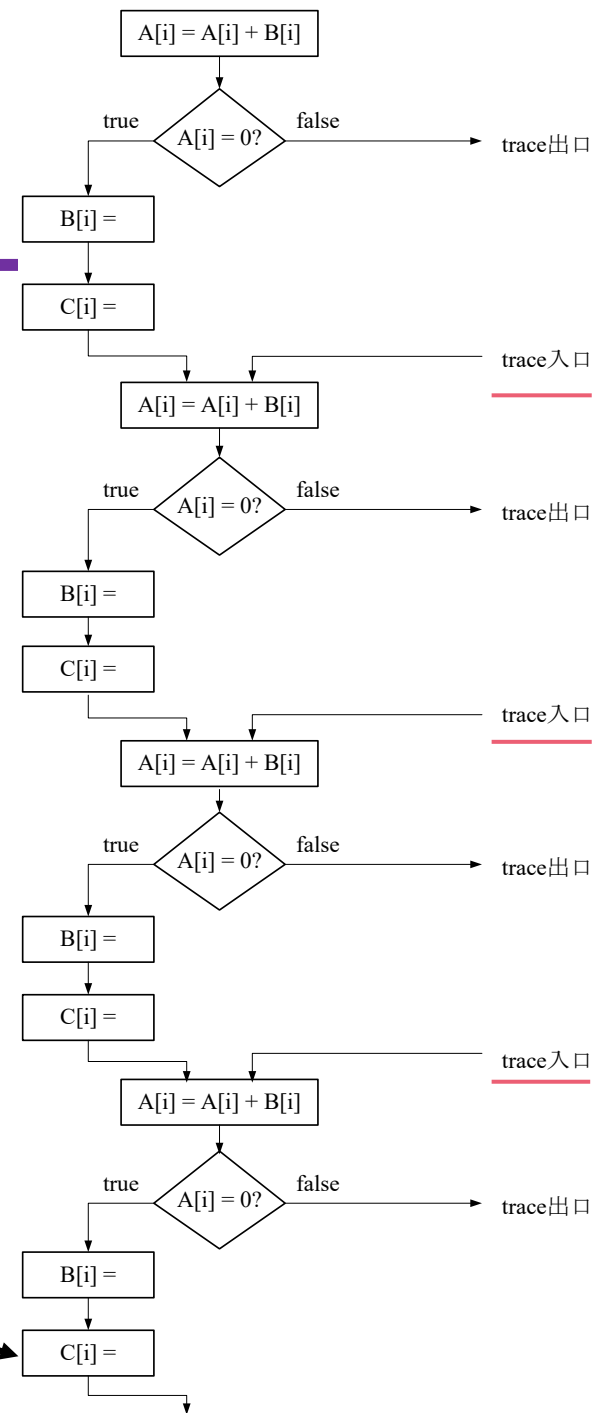
■ 踪迹选择——实例分析

- 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踪迹；
- 一条踪迹可以有多个入口和多个出口。

假设阴影部分的执行频率高



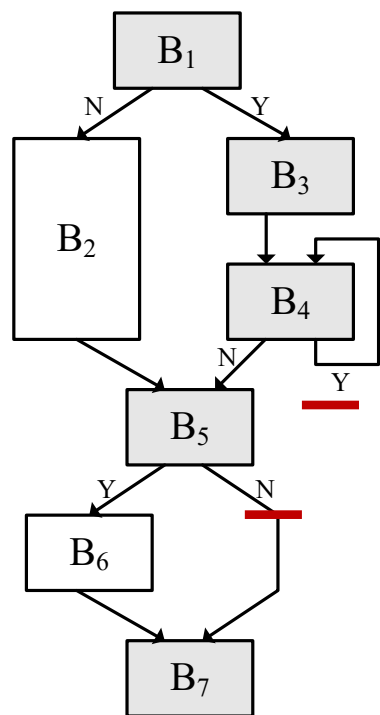
C[i]=语句是该踪迹的一个出口，执行该语句后控制流将无条件退出该踪迹



6.2.2 踪迹调度

第 27 页

- **踪迹压缩**：对已生成的踪迹进行指令调度和优化，尽可能地缩短其执行时间
 - 注意：跨越踪迹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，确保执行结果的正确性。



(a) 控制流图

假设控制流离开基本块B1和B4后发生转移（分支条件为Y）的概率大，而离开B5后发生转移的概率小

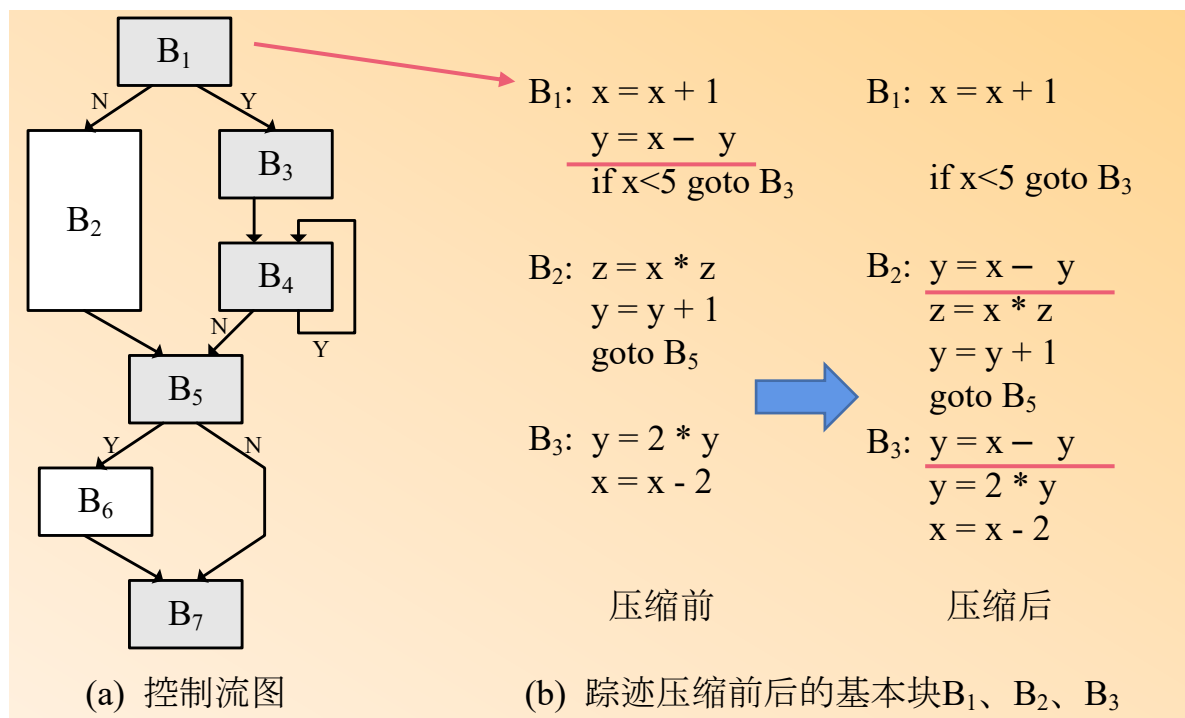
第一步：经过选择可以确认得到**3条踪迹**B₁-B₃、B₄以及B₅-B₇

理由：B4是循环体，因此这3条踪迹无法合并在一起。

6.2.2 踪迹调度

第 28 页

- **踪迹压缩**：对已生成的踪迹进行指令调度和优化，尽可能地缩短其执行时间
 - 注意：跨越踪迹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，确保执行结果的正确性。



目标：3条踪迹： B_1-B_3 、 B_4 以及 B_5-B_7

踪迹压缩：

- (1) 指令 “ $y = x - y$ ” 被从 B_1 调度到 B_3 中，跨越了踪迹的一个出口；
- (2) 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. 踪迹调度的特点总结

- 踪迹调度能够提升性能的最根本原因在于**选出的踪迹 (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**编译器实现复杂度**，特别是指令调度跨跃踪迹的一个入口或出口，或踪迹的入口或出口在踪迹内部。



如何实现？

1. 概述

- **存在问题**：在踪迹调度中，如果踪迹入口或出口位于踪迹内部，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。
- **解决方法**：增加对踪迹拓扑的约束，限制**只能拥有一个入口**
- **定义**：超块（superblock）是只能拥有一个入口，但可以拥有多个出口的结构
- **理由**：由于只能有一个入口，所以只需要考虑跨跃踪迹出口的指令调度
- **接下来的问题**：超块的构造过程与踪迹相似，但怎样确保只有一个入口？

6.2.3 超块调度

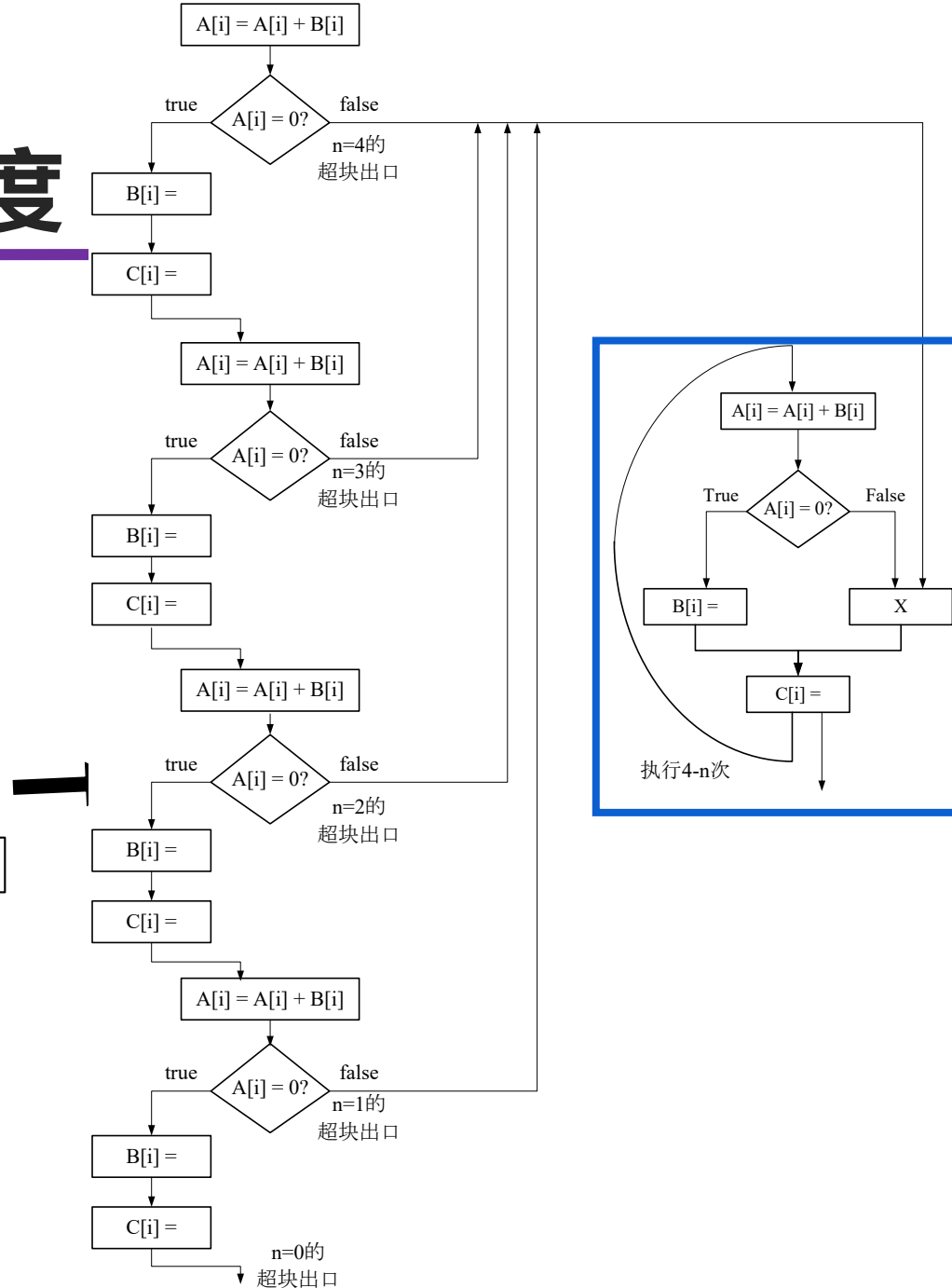
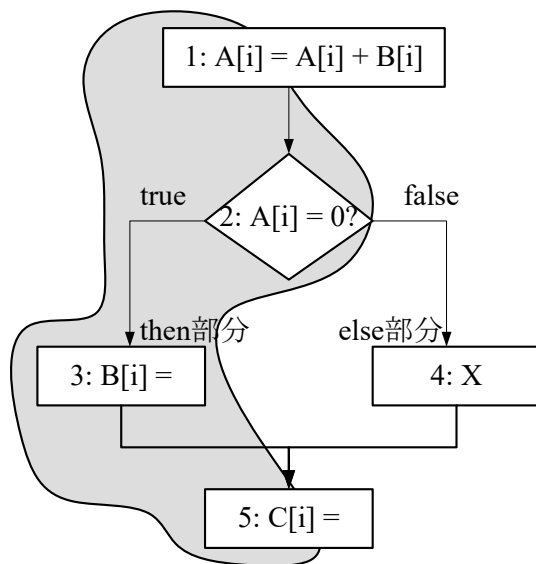
2. 超块构造——尾复制技术

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($n=4/3/2/1/0$)，表示从该出口离开已经执行了 $4-n$ 次循环；

除了最后一个出口 $n=0$ 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(蓝框部分)；

蓝框部分复制代码总是作为退出超块后必须执行的补偿代码。



3. 超块调度的特点总结

- 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 静态多指令发射：VLIW技术

多指令发射

- Pipeline CPI = 理想流水线 (Ideal pipeline) CPI + 结构冲突 (Structural stalls) + RAW 冲突 + WAR 冲突 + WAW 冲突 + 控制冲突 (Control stalls)
- 减小 理想CPI
- 多发射
 - 超标量
 - 静态调度(编译技术)
 - 动态调度(Tomasulo 算法)
 - VLIW (Very Long Instruction Word)
 - Crusoe VLIW processor [www.transmeta.com]
 - Intel Architecture-64 (IA-64) 64-bit address (EPIC)

6.3 静态多指令发射：VLIW技术

第 35 页

1. 基本概念：

把同时发射的或者满足特定约束的一组操作打包在一起，得到一条更长的（64位、128位或更长）的指令。



2. 与超标量的对比与分析

- 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。

超标量流水线的静态调度--Unrolling

	Integer Instr.	FP Instr.
1	Loop: LD F0,0(R1)	
2	LD F6,-8(R1)	
3	LD F10,-16(R1)	ADDD F4,F0,F2
4	LD F14,-24(R1)	ADDD F8,F6,F2
5	LD F18,-32(R1)	ADDD F12,F10,F2
6	SD 0(R1),F4	ADDD F16,F14,F2
7	SD -8(R1),F8	ADDD F20,F18,F2
8	SD -16(R1),F12	
9	SUBI R1,R1,#40	
10	SD -24(R1),F16	
11	BNEZ R1,Loop	
12	SD 8(R1),F20	-40+8=-32

展开 4 次 (5 个循环)

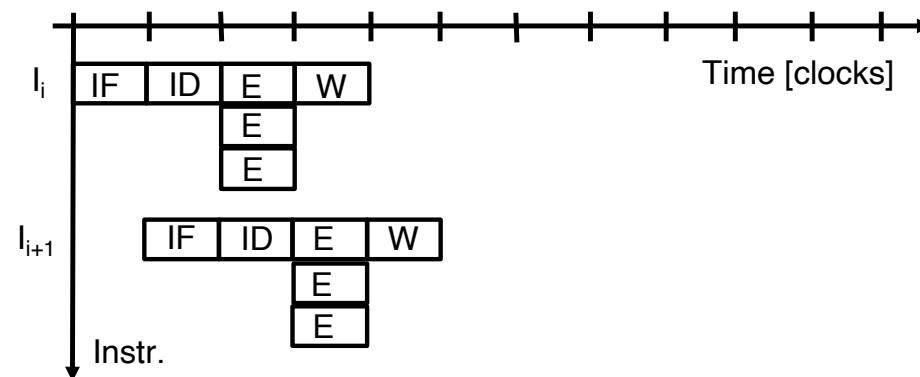
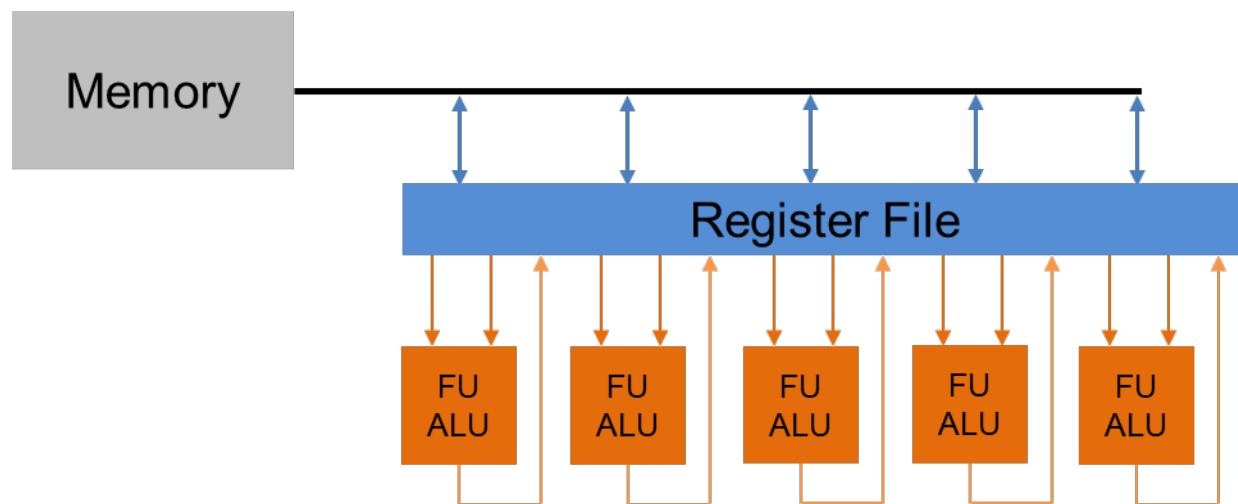
12 cc / 5 = 2.4

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

6.3 静态多指令发射：VLIW技术

第 37 页

要求：VLIW处理器需要多个功能单元才能支持多个操作同时执行；



分析：

- (1) 相关检测和指令调度工作全部由编译器完成；
- (2) VLIW处理器的功能单元利用率。

6.3 静态多指令发射：VLIW技术

3. 实例分析

例6.3 假设某VLIW处理器每个时钟周期可以同时发射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，但至少要是能够保证消除所有流水线“空转”周期，同时不考虑分支延迟。

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

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

6.3 静态多指令发射：VLIW技术

第 39 页

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。

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	DADDUI R1 , R1 , #56
S.D -32 (R1) , F20	S.D -40 (R1) , F24	nop	nop	BNE R1 , R2 , Loop
S.D 8 (R1) , F28	nop	nop	nop	nop

9条指令，最多容纳45个操作，但实际只包含23个操作，其余22个均为空操作，编码效率略大于50%，

6.3 静态多指令发射：VLIW技术

第 40 页

存在问题：

问题一. 编码效率低：如何开发出大量的指令级并级仍是面临的最大挑战；

问题二. 缺少相关检测逻辑：为了简化硬件实现，大多数VLIW处理器没有相关检测逻辑，而是靠互锁机制保证执行结果的正确性。

	Mem. Ref1	Mem Ref. 2	FP1	FP2	Int/Branch
1	LD F2, 0 (R1)	LD F6, -8 (R1)			
2	LD F10, -16 (R1)	LD F14, -24 (R1)			
3	LD F18, -32 (R1)	LD F22, -40 (R1)	ADDD F4, F0, F2	ADDD F8, F0, F6	
4	LD F26, -48 (R1)		ADDD F12, F0, F10	ADDD F16, F0, F14	
5			ADDD F20, F0, F18	ADDD F24, F0, F22	
6	SD 0 (R1), F4	SD -8 (R1), F8	ADDD F28, F0, F26		
7	SD -16 (R1), F12	SD -24 (R1), F16			SUBI R1, R1, #56
8	SD -32 (R1), F20	SD -40 (R1), F24			BNEZ R1, Loop
9	SD 8 (R1), F28				

展开 6 次 (7 个循环)

$9cc / 7 = 1.3$

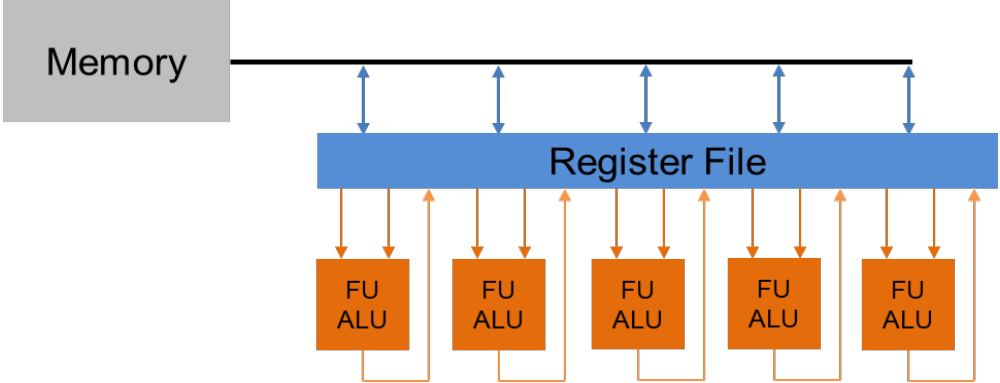
平均: 每个时钟 2.5 操作, 效率 50%

需要更多的寄存器

由于在编译时无法确定一些操作延迟，所以当一功能单元暂停时互锁机制将暂停整个流水线

6.3 静态多指令发射：VLIW技术

问题三. 目标代码兼容性差：VLIW指令格式与操作类型、功能单元数量以及延迟等体系结构参数密切相关；当这些参数变化指令格式变化。



二进制翻译是一种可行的方法，其原理是将某个硬件平台的二进制代码翻译成另一个平台的目标代码的过程。

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 F18 , -32 (R1)	L.D F22 , -40 (R1)	ADD.D F4 , F0 , F2	ADD.D F8 , F6 , F2	nop
.....
S.D 8 (R1) , F28	nop	nop	nop	BNE R1 , R2 , Loop

6.3 静态多指令发射：VLIW技术

第 42 页

4. 性能比较——多发射处理器 vs. 向量处理器

- 即使对于一些结构不规则的代码，多发射处理器也能从中挖掘出一些指令级并行。
- 多发射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：多发射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。

第6章 指令级并行及其开发—软件方法

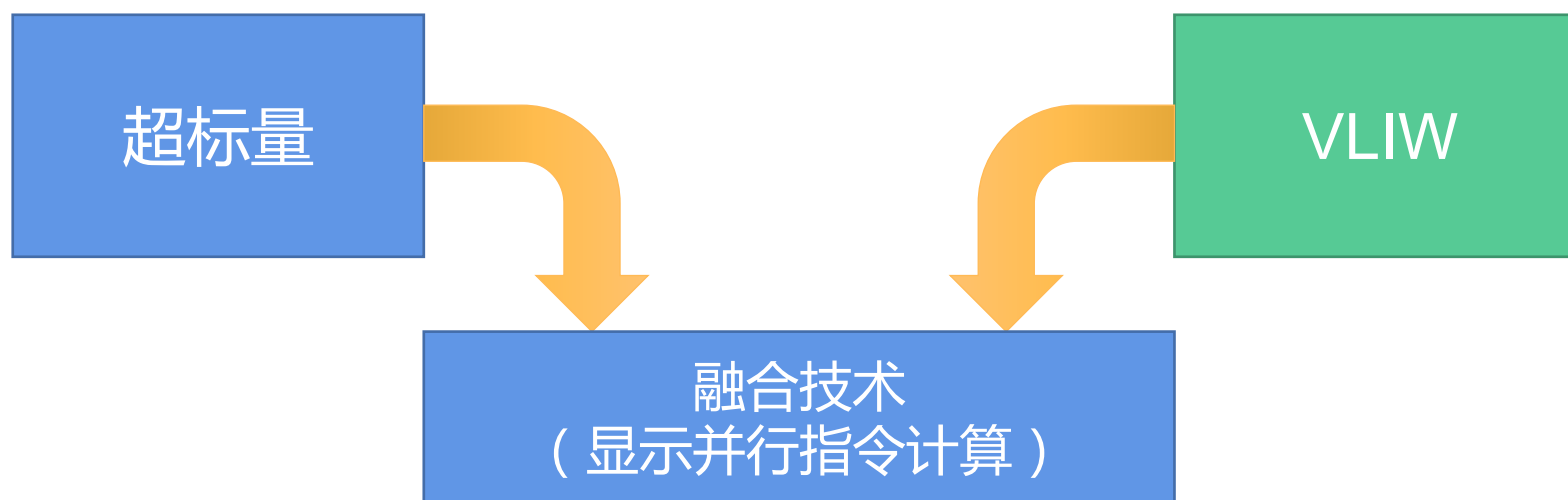
(自学)

6.4 显式并行指令计算EPIC

6.4 显示并行指令计算EPIC

第 44 页

- 超标量和VLIW结构是开发指令级并行的两种极端结构
 - **超标量**依赖流水线硬件动态识别可并行指令，存在硬件复杂度太高等问题，8发射基本上成为极限；
 - **VLIW**依赖编译器实现指令级并行开发，存在**代码兼容**问题，**编译器**的智能程度不够。



6.4 显示并行指令计算EPIC

第 45 页

■ EPIC技术在VLIW基础上融合了超标量的一些优点

- 编译器根据对程序运行特征的统计信息，如分支指令转移成功的概率，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。
- 指令集最重要的思想就是并行处理，如 Itanium和Itanium 2系列。

6.4 显示并行指令计算EPIC

第 46 页

■ 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
- 推测执行



系统结构负责设计流水线硬件执行机制，能够了解编译器“安排”好的指令，动态优化执行顺序

第6章 指令级并行及其开发—软件方法

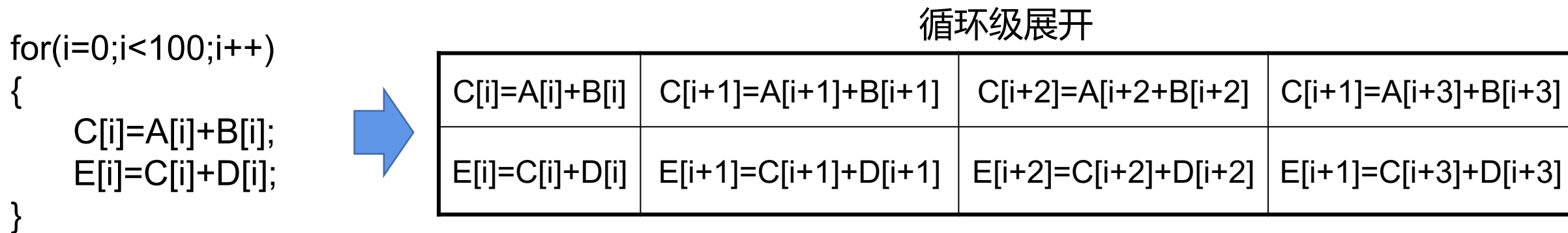
(自学)

6.5 开发更多的指令级并行

6.5 开发更多的指令并行

第 71 页

循环级并行 (Loop-Level Parallelism) : 循环中不同迭代之间的并行。由于每个循环中含有多条指令，所以它的粒度比指令级并行大。



分析：从一个循环中开发出多少并行受到的制约因素

- 迭代内部相关：同一循环迭代内部的指令相关，这属于循环体内部的基本指令调度问题；
- 结论：迭代内各语句间的数据相关对于各次迭代是否能够并行没有影响
- 条件：只要保证同一迭代内存在数据相关的各语句之间的相对顺序不变，多个循环迭代就可以并行执行。

6.5.1 挖掘更多的循环级并行

第 72 页

1. 循环携带相关

- 循环携带相关是指一个循环的某个迭代中的指令与其他迭代中的指令之间的数据相关，会大大限制循环展开的效果。

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 */  
}
```

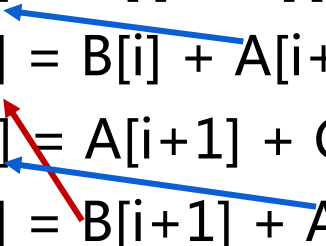
假设数组A、B和C中所有元素的存储地址都互不相同，请问语句S1与S2之间存在哪些数据相关？

6.5.1 挖掘更多的循环级并行

第 73 页

解 S1和S2之间存在两种不同类型的数据相关：

```
for ( i=1 ; i<=100 ; i=i+2 ) {  
    A[i+1] = A[i] + C[i] ;           /* S1 */  
    B[i+1] = B[i] + A[i+1] ;        /* S2 */  
    A[i+2] = A[i+1] + C[i+1] ;      /* S3 */  
    B[i+2] = B[i+1] + A[i+2] ;      /* S4 */  
}
```



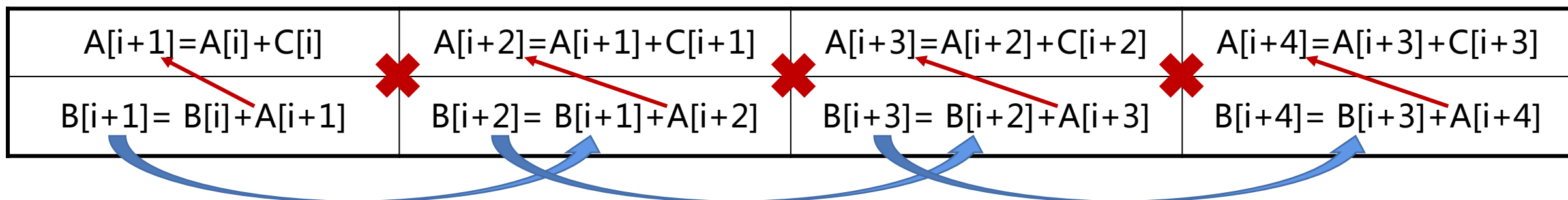
■ RAW数据相关：同一叠代内的语句S2与S1之间；

■ 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。

6.5.1 挖掘更多的循环级并行

第 74 页

```
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 */  
}
```



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。

6.5.1 挖掘更多的循环级并行

第 75 页

■ 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 */  
}
```

6.5.1 挖掘更多的循环级并行

第 76 页

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)。修改后代码如下：

```
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] ;
```

$B[i+1] = C[i] + D[i]$	$B[i+2] = C[i+1] + D[i+1]$	$B[i+3] = C[i+2] + D[i+2]$	$B[i+4] = C[i+3] + D[i+3]$
$A[i+1] = A[i+1] + B[i+1]$	$A[i+2] = A[i+2] + B[i+2]$	$A[i+3] = A[i+3] + B[i+3]$	$A[i+4] = A[i+4] + B[i+4]$

6.5.1 挖掘更多的循环级并行

第 77 页

■ 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
```

$Y[6] = Y[1] + Y[6]$
 $Y[7] = Y[2] + Y[7]$
 $Y[8] = Y[3] + Y[8]$
 $Y[9] = Y[4] + Y[9]$
 $Y[10] = Y[5] + Y[10]$
 $Y[11] = Y[6] + Y[11]$

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

$Y[2] = Y[1] + Y[2]$
 $Y[3] = Y[2] + Y[3]$

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

2. 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
- $A[i+5]$ 、 $A[j*2-6]$ 、 $\&A[k]$

■ 存在的问题：

- 由于缺少足够的运行信息（如不知道索引变量 i 、 j 、 k 的值），编译时很难确定这三个表达式的值是否相同，则无法确定这三条指令是否相关。为了确保正确性，编译器通常会保守地假设它们将访问同一元素，放弃进行优化调度机会。

6.5.2 软件流水

第 89 页

1. 简介

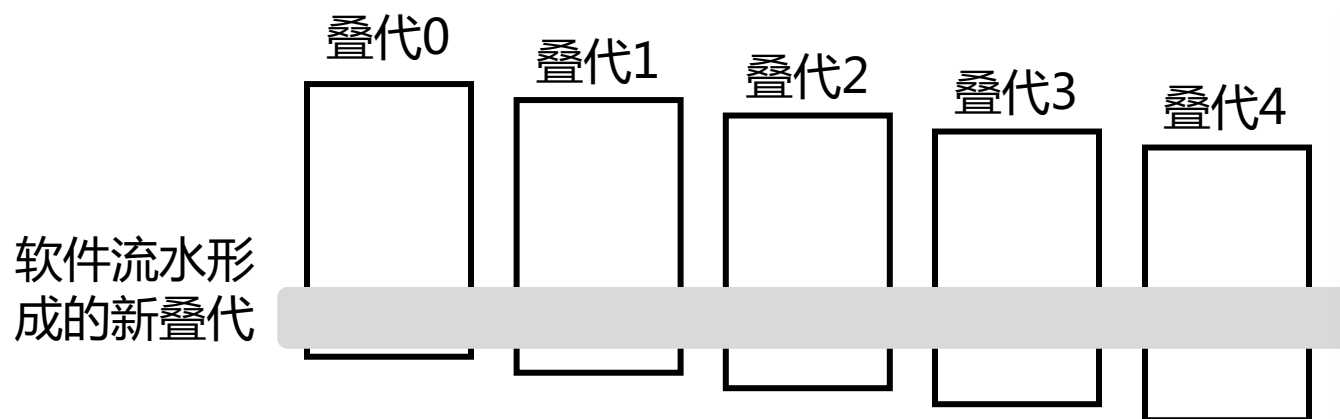
- 软件流水技术的核心思想是从循环不同的叠代中抽取一部分指令（循环控制指令除外）拼成一个新的循环叠代。

- 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

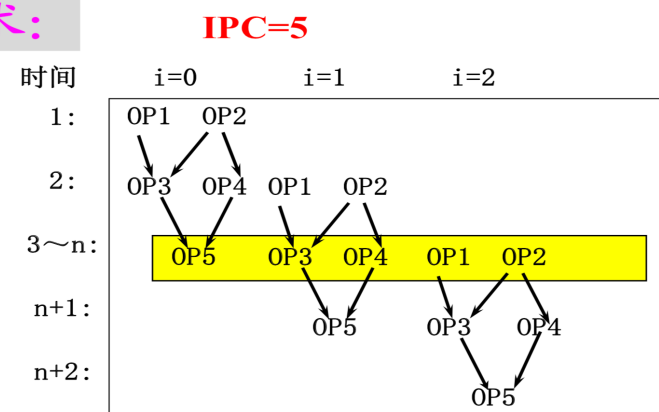
软件流水=虚拟的硬件流水线

注意：与硬件流水线不同的是，充满或排空的指令无法被封装到任何一个新的迭代中，只能放在新循环之前或之后。



软件流水技术:

```
for (i=0; i<n; i++)  
{  
  OP1:  t1=A[i]  
  OP2:  t2=B[i]  
  OP3:  t3=t1*t2  
  OP4:  t4=t2+1  
  OP5:  x=t3+t4  
}
```



6.5.2 软件流水

Before: Unrolled 3 times

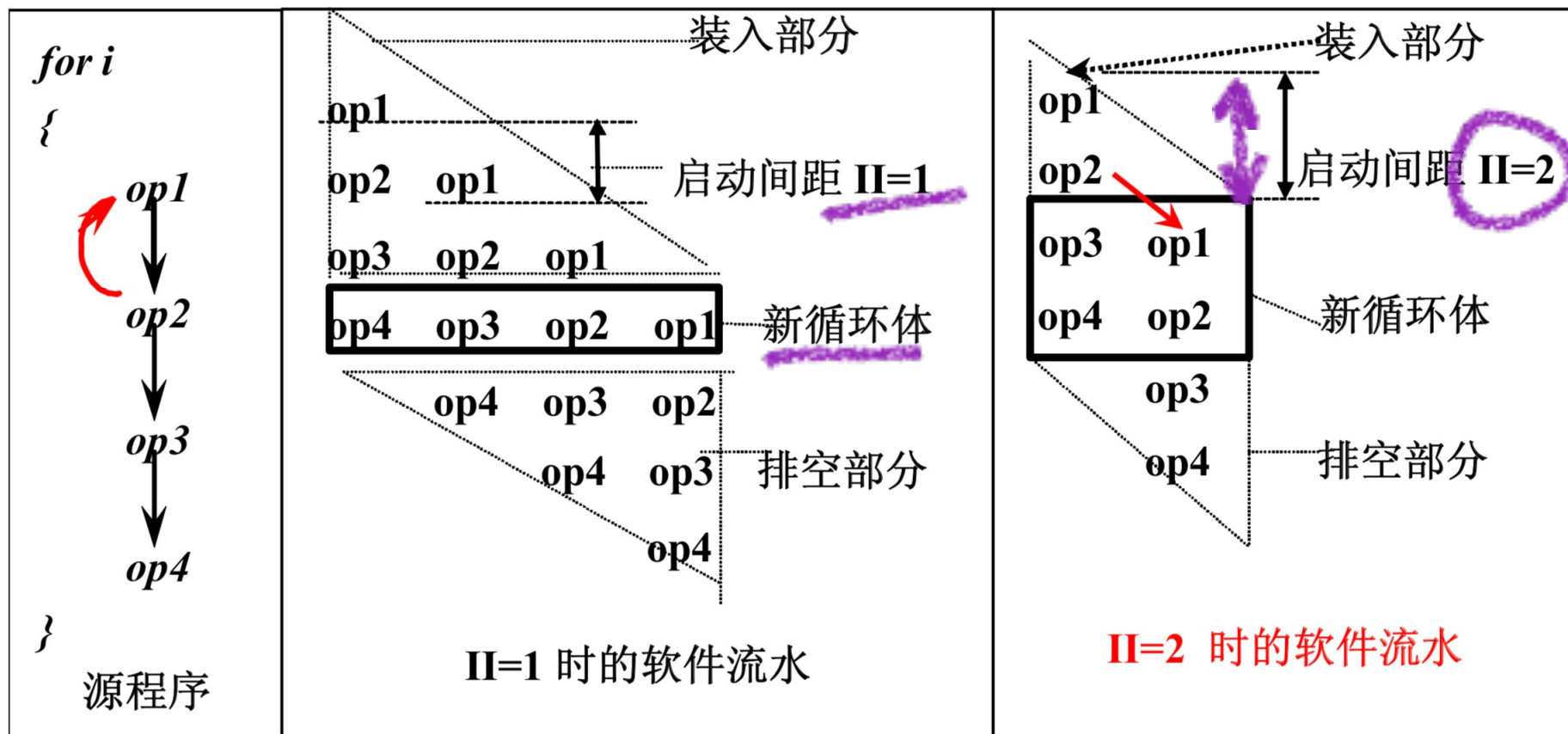
1	LD	F0,0(R1)
2	ADDD	F4,F0,F2
3	SD	0(R1),F4
4	LD	F6,-8(R1)
5	ADDD	F8,F6,F2
6	SD	-8(R1),F8
7	LD	F10,-16(R1)
8	ADDD	F12,F10,F2
9	SD	-16(R1),F12
10	SUBUI	R1,R1,#24
11	BNEZ	R1,LOOP

After: Software Pipelined

1	SD	0(R1),F4 ;	Stores M[i]
2	ADDD	F4,F0,F2 ;	Adds to M[i-1]
3	LD	F0,-16(R1) ;	Loads M[i-2]
4	SUBUI	R1,R1,#8	
5	BNEZ	R1,LOOP	

每次迭代 5 cc

软件流水示意图



2. 实例分析

例6.11 试用软件流水技术处理例6.1中的循环，假设数组x有n个元素。

解 软件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)

6.5.2 软件流水

第 93 页

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。

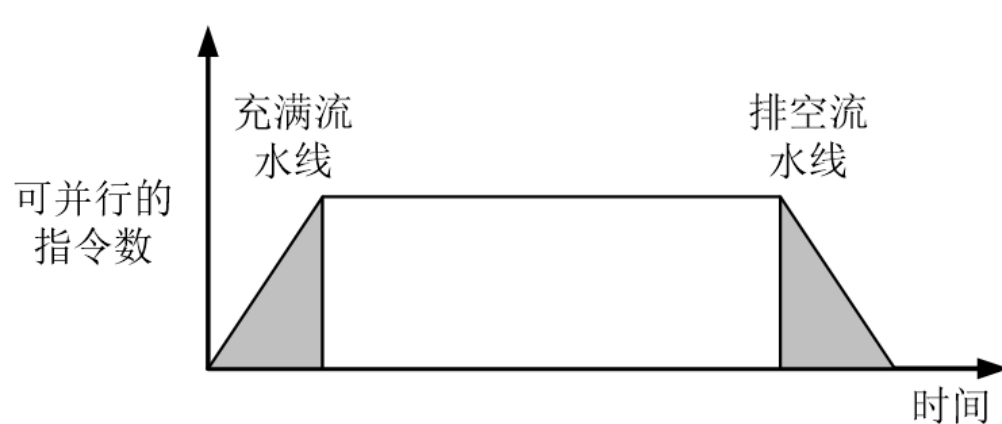
```
DADDUI    R1 , R1 , #-16 // I1 : R1保存x[n-2]的地址
L.D F0 , 16 ( R1 )      // I2 : 取x[n]
ADD.D     F4 , F0 , F2   // I3 : x[n] = x[n] + F2
L.D F0 , 8 ( R1 )       // I4 : 取x[n-1]
Loop : S.D F4 , 16 ( R1 ) // I5 : 存x[i+2]
ADD.D     F4 , F0 , F2   // I6 : x[i+1] = x[i+1] + F2
L.D F0 , 0 ( R1 )       // I7 : 取x[i]
BNER1 , 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：

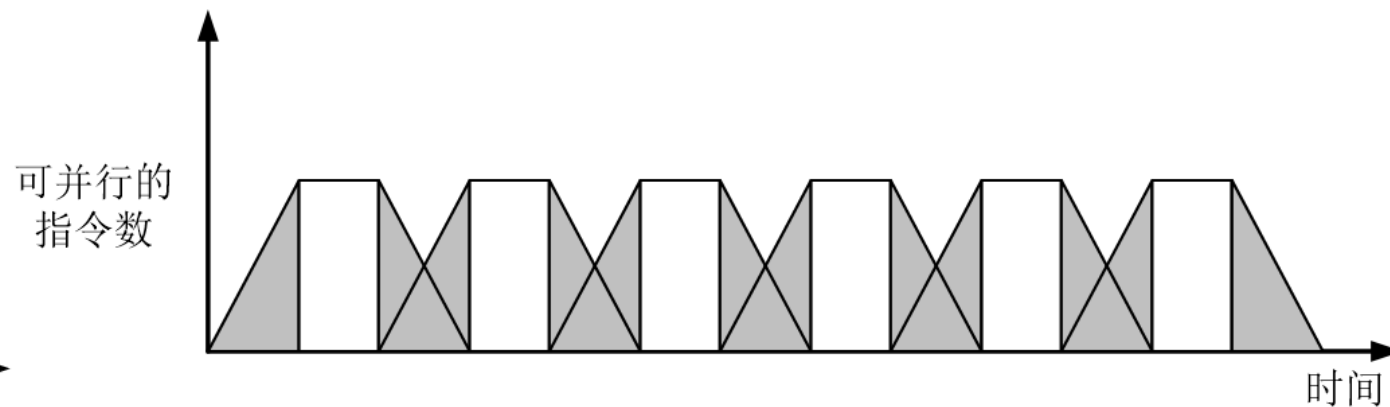
- 新循环的结束条件仍为： $R1 = R2$
- 新循环从元素 $x[n-2]$ 开始从头处理的，元素 $x[n]$ 与 $x[n-1]$ 只是从半途开始处理。
- 新循环的最后一次叠代只处理完元素 $x[3]$ ， $x[2]$ 刚被修改完结果尚未写回， $x[1]$ 刚被取出，需要被修改并写回。
- 新循环的每个叠代相当于流水执行了原循环的3个叠代。元素 $x[n/n-1/2/1]$ 的处理相当于充满和排空这条“流水线”。

3. 性能比较：软件流水 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。
- 软件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

1. 基本指令调度及循环展开：指令调度、循环展开及注意事项
2. 跨越基本块的静态指令调度：全局指令调度、踪迹调度、踪迹选择、踪迹压缩、超块调度、尾复制技术
3. 静态多指令发射：VLIW技术：VLIW vs. 超标量、多发射处理器 vs. 向量处理器
4. 显式并行指令计算EPIC：EPIC、非绑定分支、谓词执行、推测执行
5. 开发更多的指令级并行：循环携带相关、存储别名分析、软流水技术
6. 实例：IA-64体系结构：Itanium、指令格式、谓词执行机制、推测执行机制

第6章习题6.4、6.5、6.6、6.7、6.8

第7章 存储系统