

计算机组织与系统结构

利用流水线改进性能

Enhancing Performance with Pipelining

(第十三讲)

程旭

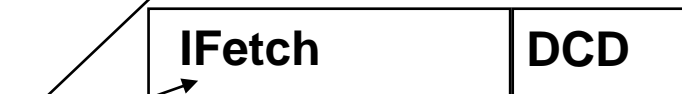
2020.12.17

- 流水线向下传递控制信息，就象向下传递数据一样
- 通过局部控制解决 前递/暂停
- 意外事件会导致流水线停止
- **MIPS**指令系统体系结构中流水线是可见的（延迟转移、延迟装入）
- 更深的流水线、更多的并行度可能获得出更高的性能
- 中断、指令系统、浮点操作加大流水线的难度
- 编译器可以减少数据和控制冒险的代价
 - 装入延迟槽
 - 转移延迟槽
 - 转移预测

再谈流水线冒险



Structural Hazard



I-Fet ch DCD MemOpFetch OpFetch Exec Store



Control Hazard



RAW (read after write) Data Hazard



WAW Data Hazard (write after write)



WAR Data Hazard (write after read)

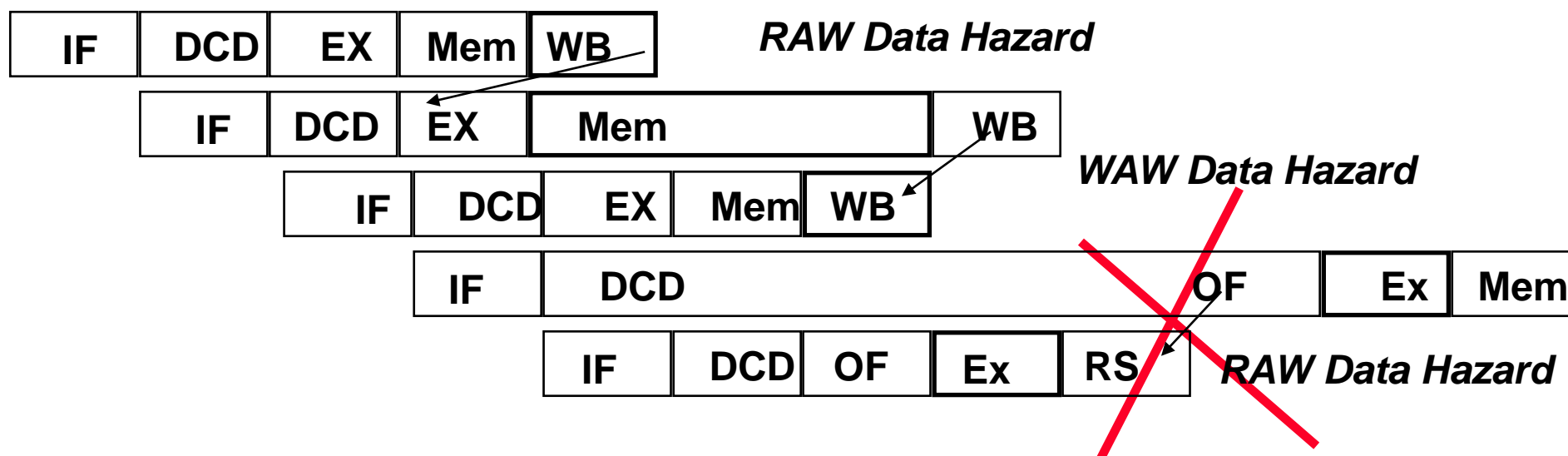
数据冒险

如何避免一些冒险

- 通过总是在流水线的前段(DCD)取操作数, 来消除WAR
- 通过按序完成所有回写操作(在最后一级,静态), 来消除WAW

检测并解决RAW

- 暂停, 并尽可能前递



意外事件中的问题



◦ 意外事件/中断: 在5段流水线中执行着5条指令

- 如何停止流水线?
- 重启?
- 哪些问题产生中断?

段名 可能出现的中断问题

IF 取指页失效、未对准存储器访问、存储保护违例

ID 未定义或非法操作码

EX 算术意外事件

MEM 取数据页失效、未对准存储器访问、存储保护违例、存储器错误

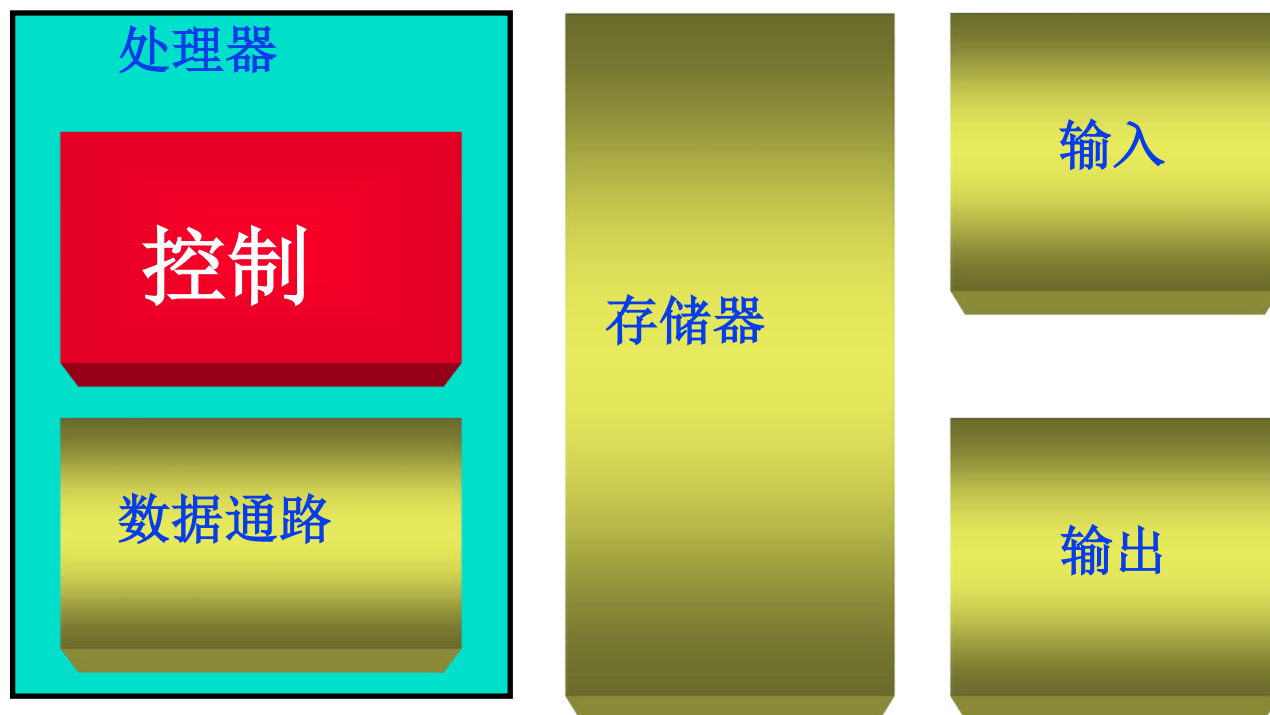
◦ 产生数据页失效的**Load**指令、产生指令页失效的 **Add**指令?

◦ 解决方案1: 中断向量/指令

◦ 解决方案2: 尽可能早地中断执行, 之后, 重启所有未执行完的操作

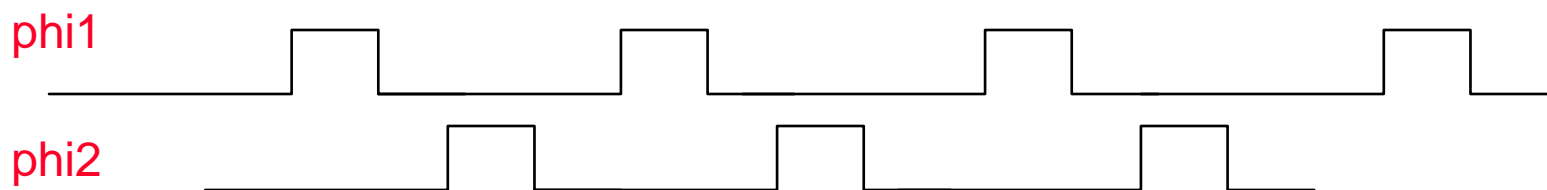
教学目标：已经掌握的内容

。 计算机的五个基本部件

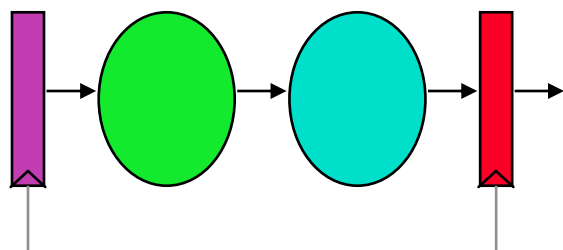


。 本讲主题:高级流水线技术

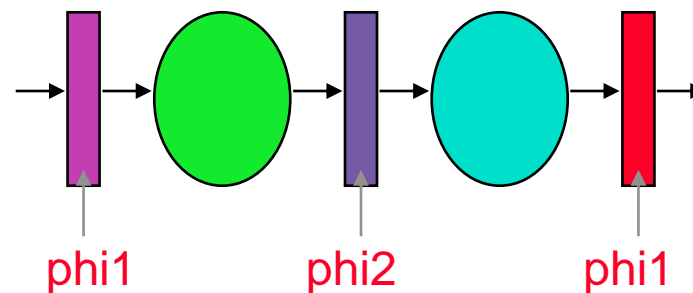
参考：MIPS R3000的时钟定时策略



- ° 双相无重叠时钟（2-phase non-overlapping clocks）
- ° 流水线段两级锁存（电平使能）



边沿触发





MIPS R3000 指令流水线

Inst Fetch		Decode Reg. Read		ALU / E.A		Memory	Write Reg	
TLB	I-Cache	RF		Operation			WB	
				E.A.	TLB	D-Cache		

使用资源情况

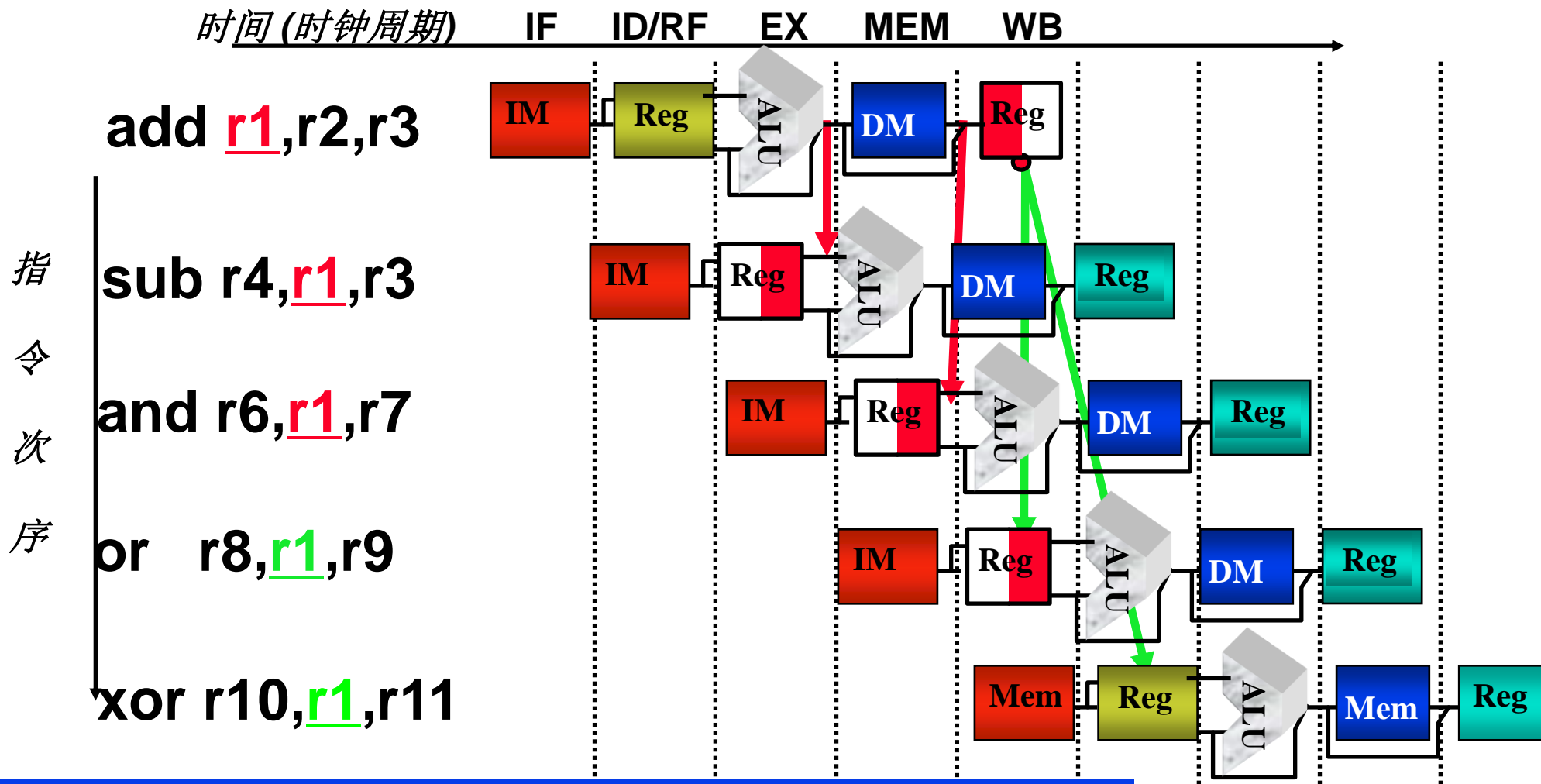
TLB					TLB				
	I-cache								
			RF					WB	
				ALU	ALU				
						D-Cache			

在第一相进行写，在第二相进行读 => 消除了从WB段的旁路

关于r1的数据冒险

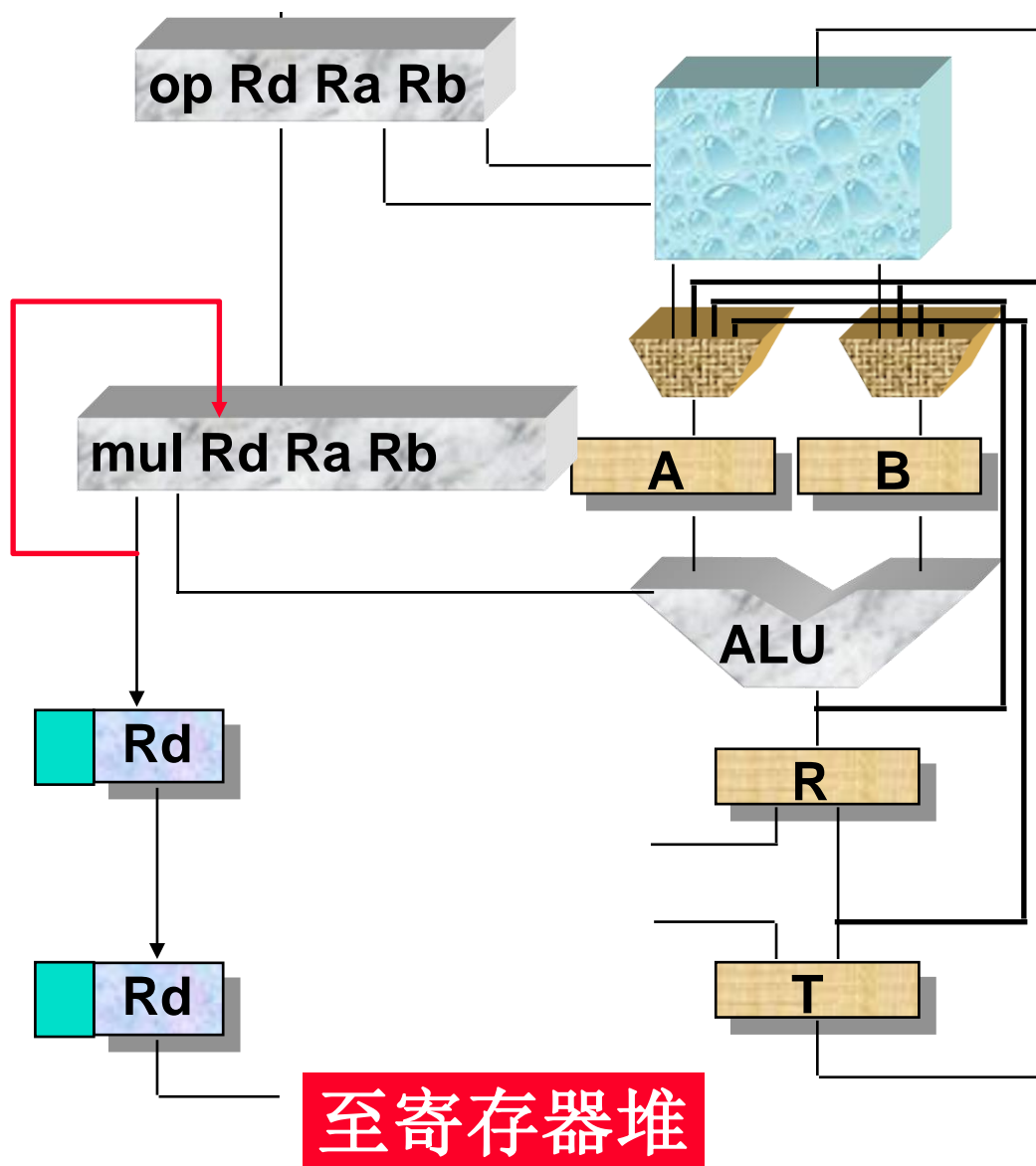


- 立即向后相关就可能出现冒险



在MIPS R3000流水线中， 无需从WB段进行前递！

MIPS R3000 的多周期操作



例如：乘法、除法、**Cache**失效

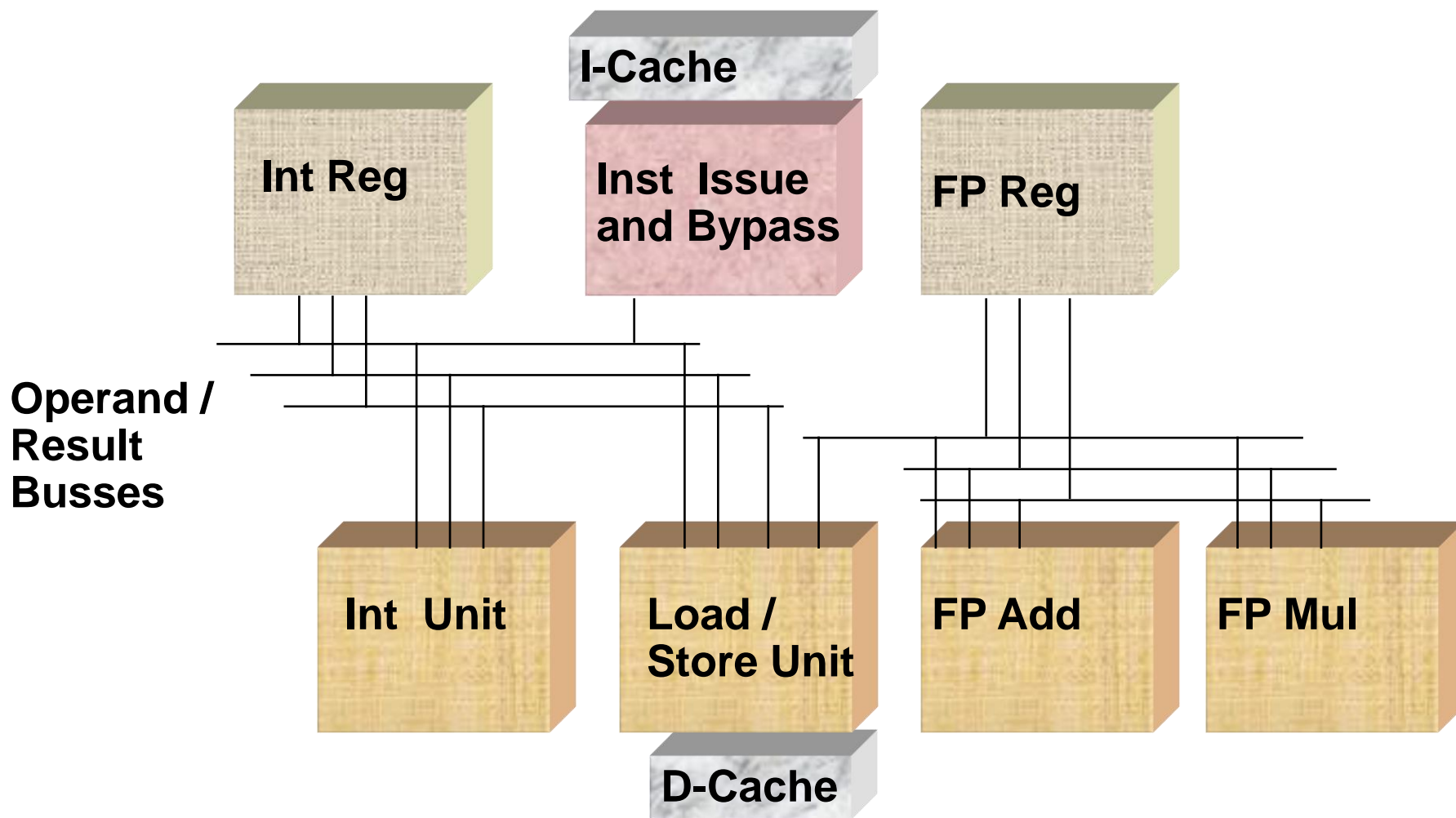
暂停流水线中多周期操作之上的所有流水段

排空（空泡）它之下的所有段

使用本地流水段状态的控制字来一步步执行多周期操作

简单的超标量

不相关整数指令和浮点指令被发送到不同的流水线



使CPI < 1: 每一周期发射多条指令

- 两种主要结构: 超标量和超长指令字
- 超标量: 每一周期可能发射不同数量的指令 (从1条到 6条)
 - 视并行度和相关的具体情况而定。由硬件处理
 - **IBM PowerPC 604、Sun UltraSparc、DEC Alpha 21164、HP 7100**
- 超长指令字: 固定数量的指令; 编译器确定可开发的并行度
 - 流水线可见; 编译器必须对延迟槽进行指令调度, 以确保结果正确
- **Itanium: Explicit Parallel Instruction Computer (EPIC)**
 - **128位指令包 (packets)** 包含三条指令(可以串行执行)
 - 可以将**128位指令包**联接起来, 允许更高的并行性
 - 编译器来决定并行度,
硬件检测指令间的相关和前递/暂停

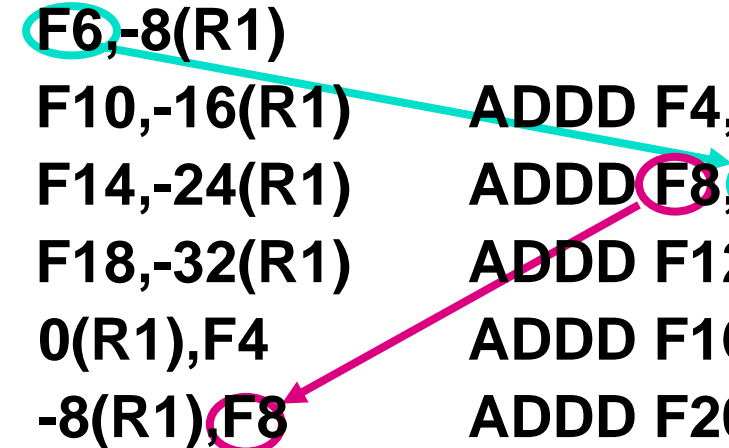
循环展开：减小标量延迟

1	Loop:	LD	F0, 0 (R1)	
2		LD	F6, -8 (R1)	
3		LD	F10, -16 (R1)	
4		LD	F14, -24 (R1)	LD to ADDD: 1 Cycle
5		ADDD	F4, F0, F2	
6		ADDD	F8, F6, F2	
7		ADDD	F12, F10, F2	
8		ADDD	F16, F14, F2	ADDD to SD: 2 Cycles
9		SD	0 (R1), F4	
10		SD	-8 (R1), F8	
11		SD	-16 (R1), F12	
12		SUBI	R1, R1, #32	
13		BNEZ	R1, LOOP	
14		SD	8 (R1), F16 ; $8 - 32 = -24$	

14个时钟周期 或者 每次迭代3.5个周期

超标量中的循环展开

整数指令	浮点指令	时钟周期
Loop: LD F0,0(R1)		1
LD F6,-8(R1)		2
LD F10,-16(R1)	ADDD F4,F0,F2	3
LD F14,-24(R1)	ADDD F8,F6,F2	4
LD F18,-32(R1)	ADDD F12,F10,F2	5
SD 0(R1),F4	ADDD F16,F14,F2	6
SD -8(R1),F8	ADDD F20,F18,F2	7
SD -16(R1),F12		8
SD -24(R1),F16		9
SUBI R1,R1,#40		10
BNEZ R1,LOOP		11
SD -32(R1),F20		12

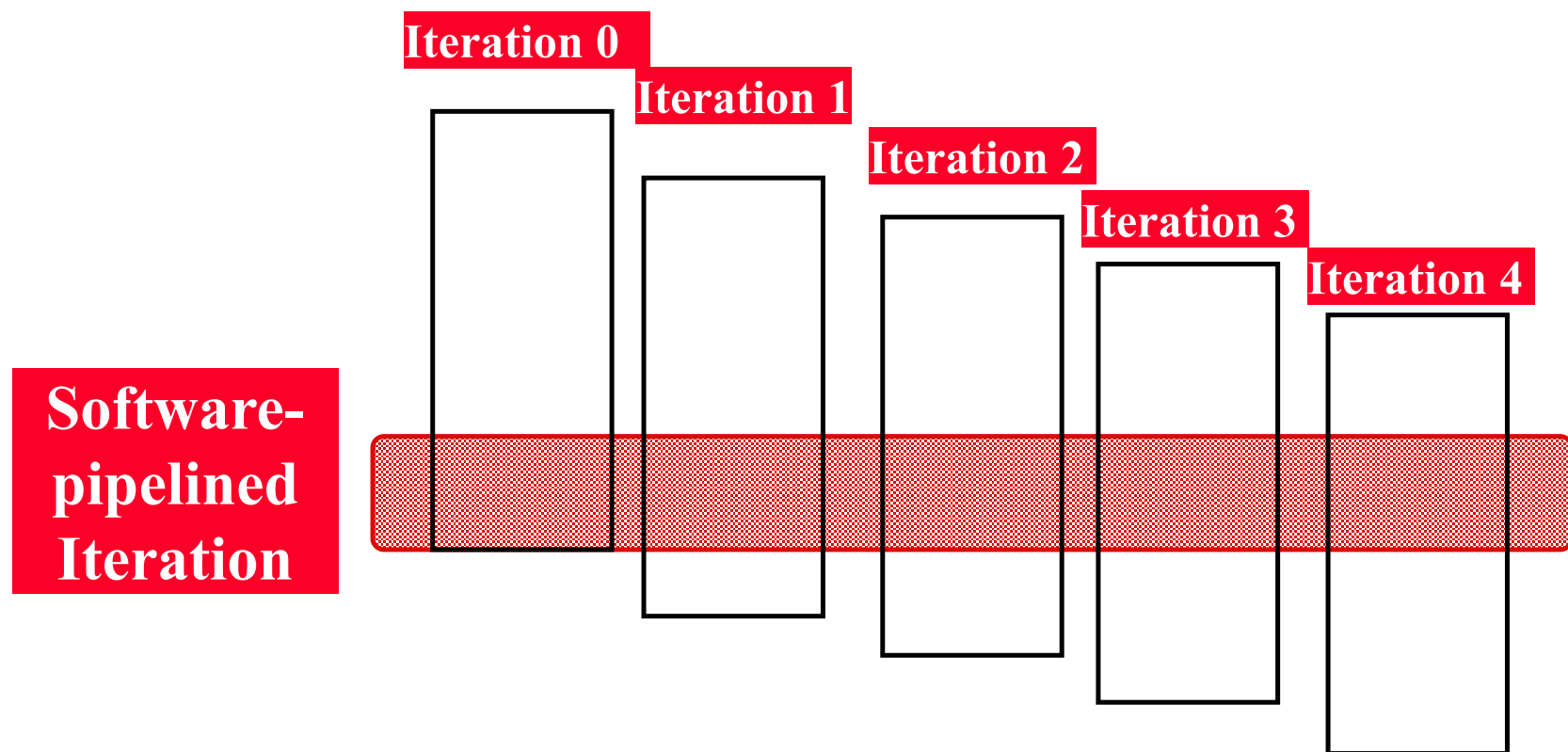


为了避免延迟(超标量需要+1), 把循环展开5次

12个时钟周期, 每次迭代2.4周期

软件流水技术

- 发现: 如果循环的每次迭代之间没有相关, 那么通过从不同的迭代中抽取指令来获得更高的指令级并行性。
- 软件流水: 对循环进行重构, 使得每次迭代执行的指令是属于原循环的不同迭代过程的。(软件形式的Tomasulo算法)



软件流水示例

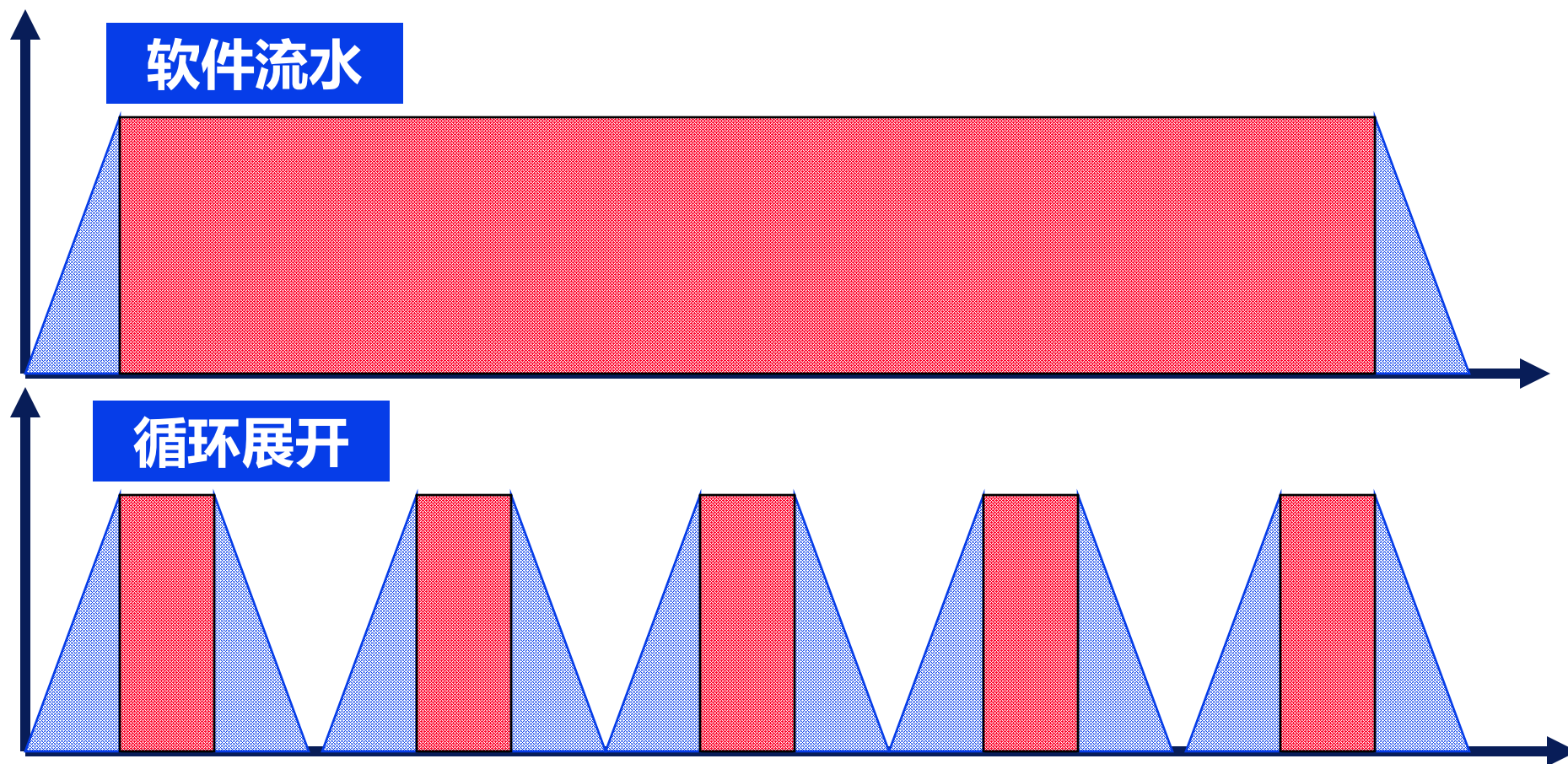
软件流水化之前: 展开3次

```
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 SUBI  R1, R1, #24
11 BNEZ  R1, LOOP
```

软件流水化之后

```
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  SUBI  R1, R1, #8
5  BNEZ  R1, LOOP
```


循环展开和软件流水的示例图



- 软件流水
 - 代码空间较小
 - 只需填充和排空流水线一次
 - 而循环展开每次迭代就需要一次

超标量结构的限制

- 虽然分离整数部件和浮点部件对硬件并不困难，但是为了实现 **CPI=0.5**，执行的程序必须满足：
 - 浮点操作恰好占整个指令总数的 **50%**
 - 指令间没有冒险
- 如果同时发射更多的指令，译码和发射机制都将更加困难
 - 即使对于 双发射超标量 => 检测 **2**个操作码, **6**个寄存器标识符, 并且确定是发射单条指令，还是发射两条指令

超长指令字结构

- **VLIW:** 指令空间和简化译码之间的权衡
 - 长指令字具有存放多个操作的空间
 - 编译程序放在同一长指令字中的操作可以并行执行
 - 例如， **2个整数操作、2个浮点操作、2个存储器访问操作、1个转移操作**
 - 每一场位**16 ~ 24位**
 $\Rightarrow 7 \times 16$ (112) 位 $\sim 7 \times 24$ (168) 位宽度
 - 需要编译技术来在多条转移指令之间进行指令调度

VLIW中的循环展开

存储器访问1	存储器访问2	浮点操作1	浮点操作2	整数/转移	时钟
LD F0,0(R1)	LD F6,-8(R1)				1
LD F10,-16(R1)	LD F14,-24(R1)				2
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F2		3
LD F26,-48(R1)		ADDD F12,F10,F2	ADDD F16,F14,F2		4
		ADDD F20,F18,F2	ADDD F24,F22,F2		5
SD 0(R1),F4	SD -8(R1),F8	ADDD F28,F26,F2			6
SD -16(R1),F12	SD -24(R1),F16				7
SD -32(R1),F20	SD -40(R1),F24			SUBI R1,R1,#48	8
SD -0(R1),F28				BNEZ R1,LOOP	9

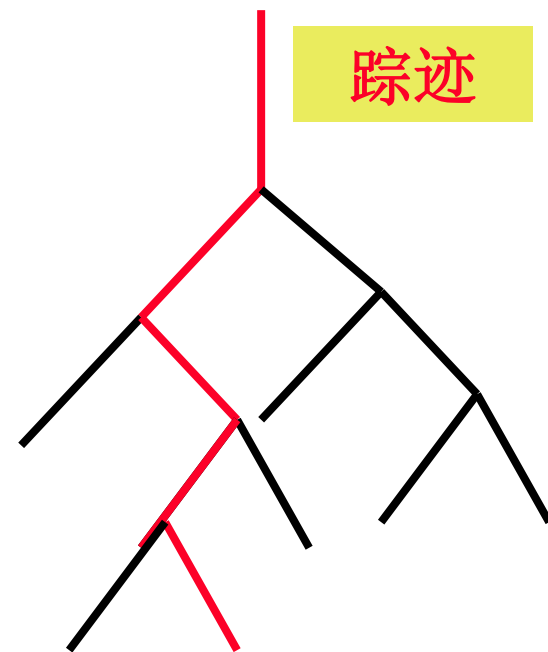
为了避免延迟，展开7次

9个周期产生7个结果 每次迭代1.3个周期

VLIW需要更多的寄存器(EPIC \Rightarrow 128int + 128FP)

踪迹调度 (Trace Scheduling)

- 跨越IF转移指令（不一定是LOOP转移指令）的并行度
- 两步：
 - 踪迹选择 (*Trace Selection*)
 - 发现（静态分析预测的）执行代码序列中最可能执行的基本块序列 (*trace*)
 - 踪迹压缩 (*Trace Compaction*)
 - 将踪迹挤压成一些VLIW指令
 - 需要增加一些标记代码 (*bookkeeping code*)，以防预测错误



硬件策略：指令并行

- 为什么需要硬件在程序执行过程中处理？
 - 对于在编译时不能准确辨别的实际相关，硬件策略可以很好地工作
 - 编译器更加简单
 - 为一台机器编写的代码可以在另外的机器上很好地工作
- 主要思想：希望暂停指令的后续指令继续处理

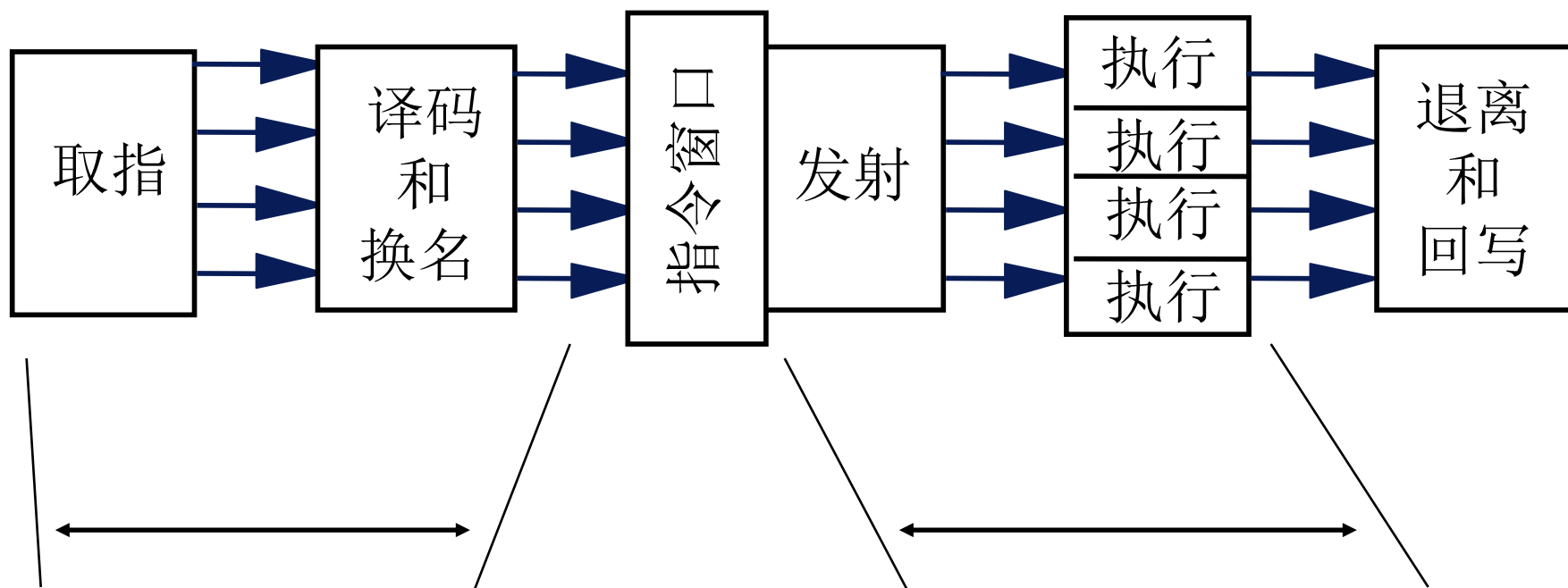
DIVD F0, F2, F4

ADDD F10, F0, F8

SUBD F12, F8, F14

- 允许乱序执行 => 乱序完成
- 指令译码段检测结构相关I

超标量流水线



■按序将指令递交到乱序执行的内核!

支持按序发射指令的记分板技术

Scoreboard for In-order Issues

Busy[FU#] : a bit-vector to indicate FU's availability.
(FU = Int, Add, Mult, Div)

These bits are hardwired to FU's.

WP[reg#] : a bit-vector to record the registers for which
writes are pending.

These bits are set to true by the Issue stage and set to false by
the WB stage

Issue checks the instruction (opcode dest src1 src2)
against the scoreboard (Busy & WP) to dispatch

FU available?

RAW?

WAR?

WAW?

Busy[FU#]

WP[src1] or WP[src2]

cannot arise

WP[dest]

硬件策略：指令并行

- 乱序执行需要进一步划分指令译码段：
 1. 发射：指令译码, 检测结构冒险
 2. 读操作数：等待到没有数据冒险, 再读取操作数
- 记分板（**Scoreboards**）允许满足上述两个条件的指令被立即执行，而无需等待到前面的指令执行完毕
- **CDC 6600**: 按序发送、乱序执行、乱序提交（也称为乱序完成）

记分板控制的四级

- **发射**— 指令译码 并 检测结构冒险(ID1)
 - 按照程序的次序发射指令 (进行冒险检测)
 - 如果存在 **结构冒险** 暂停发射
 - 如果 带发射的指令与 已发射但尚未完成的指令之间存在输出相关, 则暂停发射 (无WAW冒险)
- **读操作数**—等待到没有数据冒险, 再读取操作数 (ID2)
 - 由于将等待未完成指令写回其结果, 因而在该阶段, 可解决所有的真数据相关(RAW冒险)
 - 在该模型中, **无数据前递!**

记分板控制的四级（续一）

◦ 执行——对操作数进行操作 (EX)

- 接收到操作数之后，功能部件开始执行。当产生结果之后，它通报记分板：已经完成执行。

◦ 写结果——完成执行 (WB)

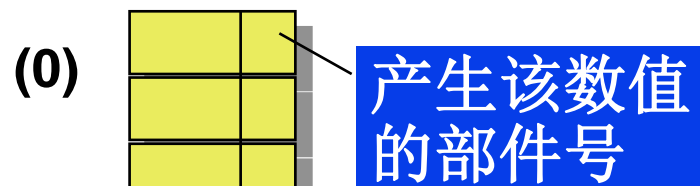
- 暂停直到与以前的指令没有WAR冒险：

示例：

```
DIVD  F0, F2, F4  
ADDD  F10, F0, F8  
SUBD  F8, F8, F14
```

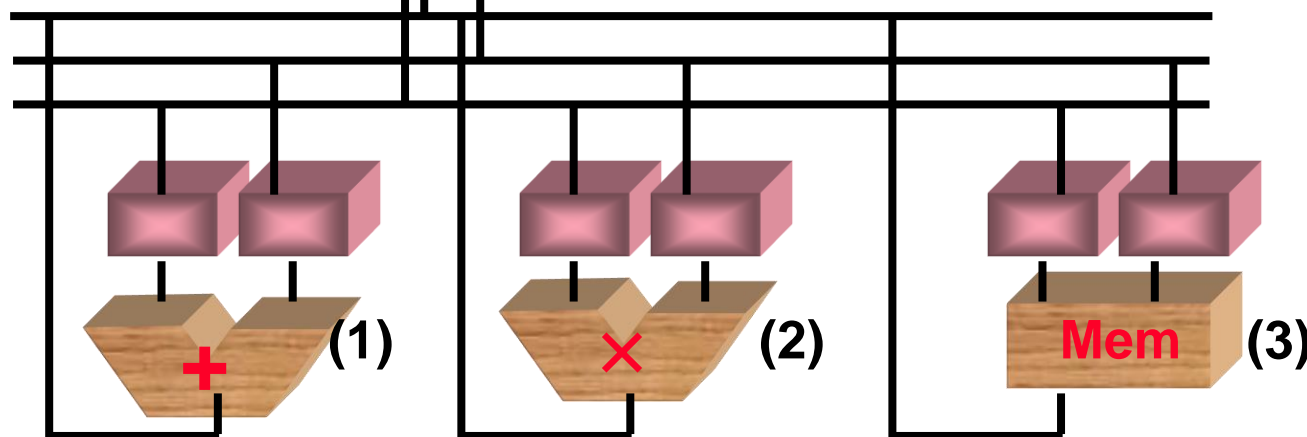
CDC 6600的记分板将暂停**SUBD**指令，直到**ADDD**指令读取了操作数。

记分板 (Scoreboard, CDC 6600)



如果有空闲的功能部件并且没有未决的对其目标进行修改的指令，就立即发射

- 保持直到寄存器可用
- 当准备好后，取操作数、执行
- 在回写段修改记分板



$r1 \leftarrow M[r1 + r2]$
 $r2 \leftarrow r2 * r3$
 $r4 \leftarrow r2 + r5$
 $r2 \leftarrow r0$

op Ra ? Rb ? Rd S1 S2

op Ra ? Rb ? Rd S1 S2

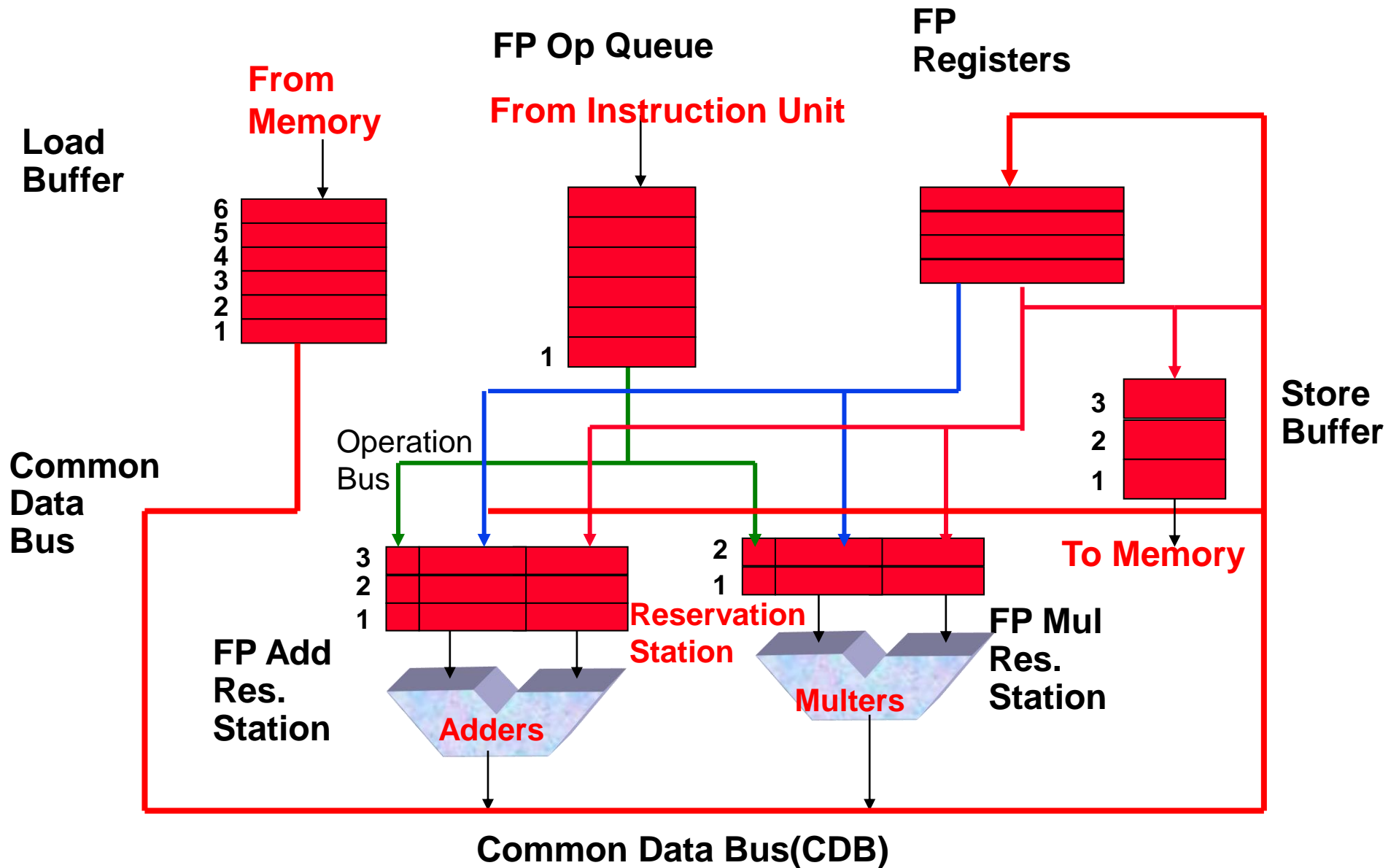
op Ra ? Rb ? Rd S1 S2

指令

Tomasulo算法 与 记分板

- 控制&缓冲器 分布于 功能部件(FU) 与 集中于记分板;
 - 功能部件缓冲器称为 “保留站(reservation stations)”; 存放未决的操作数
- 指令中的寄存器 被 数值或者指向保留站的指针 代替; 这一过程称为 寄存器换名(register renaming) ;
 - 消除WAR、WAW冒险
 - 保留站 比 实际寄存器 多, 因而可以完成优化编译器所不能完成的一些工作
- 结果 从 RS 直接 到 FU, 无需通过寄存器, 而是通过 公共数据总线(Common Data Bus) 把结果广播到所有FU
- 装入 (Load) 和 存储 (Stores) 也象 其他功能部件一样使用保留站

Tomasulo的结构图



Tomasulo算法的三段

1. **Issue**—从FP Op Queue中取出指令

如果保留站空闲(无结构冒险),
控制机制 发射 指令&发送操作数(对寄存器进行换名)。

2. **Execution**—对操作数执行操作(EX)

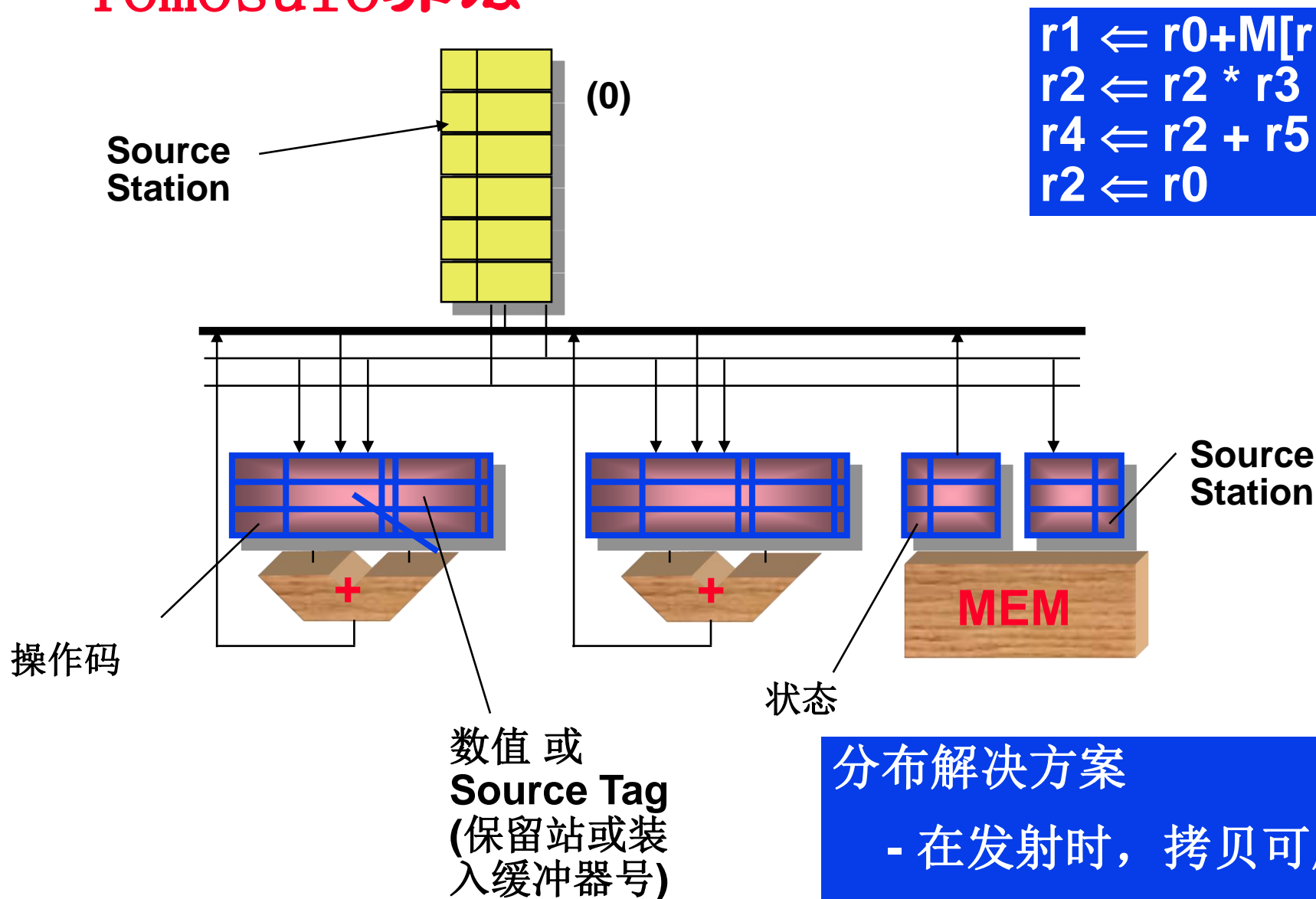
如果两个操作数都已就绪, 就执行;
如果没有就绪, 就观测 公共数据总线 等待所需结果

3. **Write result**—完成执行(WB)

通过公共数据总线将结果写入到所有等待的部件;
标记 保留站 可用

- 正常的数据总线: 数据 + 目的 (“去向” 总线)
- 公共数据总线: 数据 + 源 (“来源” 总线)
 - 64位数据 + 4位功能部件 源 地址
 - 如果与期望的功能部件匹配, 就 “写” (产生结果)
 - 进行广播

Tomosulo算法



分布解决方案

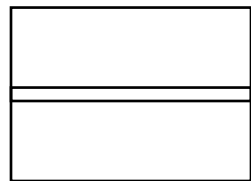
- 在发射时，拷贝可用参数
- 对等待的操作数从功能部件直接前递

寄存器换名

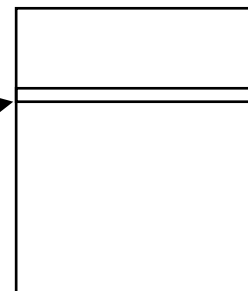
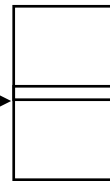
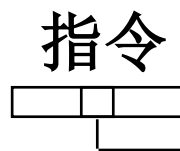
使用很大一组通用寄存器，编译可以通过换名技术来消除WAR冒险

- 有时，需要增加移动操作（**move**）
- 硬件可以在运行过程中解决这一问题（但是却不能考察程序的其他部分）

体系结构定义的
寄存器



映射表



很大的内部寄存器

•取操作数

所有的源寄存器通过映射表进行换名

•发射时:

为目标寄存器赋予一个新的伪寄存器
修改映射表

- 在下一次写入之前，适用于所有后续指令

记分板产生的问题

- 乱序完成 => **WAR**、**WAW**冒险
- **WAR**冒险的解决方案
 - 把操作和它们所需操作数的拷贝排队等待
 - 只有在读操作数段才读寄存器
- 对于**WAW**冒险, 必须检测该冒险: 暂停直到其他指令完成
- 在执行阶段需要支持多条指令同时执行 => 多套执行部件 或者流水化执行部件
- 记分板纪录指令间相关情况、状态或操作信息
- 记分板用四段流水替代了流水线的**ID**、 **EX**、 **WB**段

动态超标量的性能

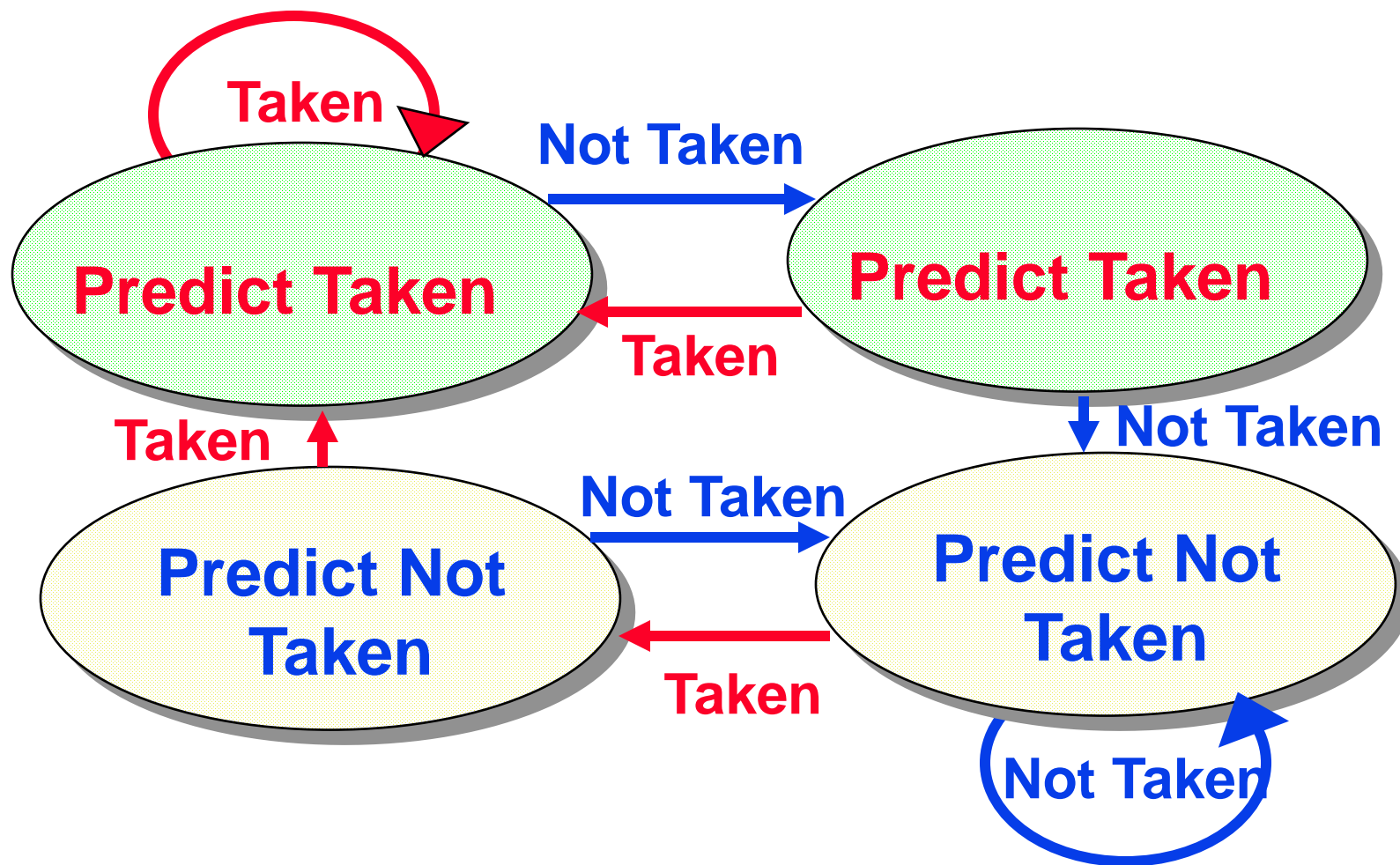
迭代 序号	指令	发射	执行 时钟周期数目	写结果
1	LD F0,0(R1)	1	2	4
1	ADDD F4,F0,F2	1	5	8
1	SD 0(R1),F4	2	9	
1	SUBI R1,R1,#8	3	4	5
1	BNEZ R1,LOOP	4	5	
2	LD F0,0(R1)	5	6	8
2	ADDD F4,F0,F2	5	9	12
2	SD 0(R1),F4	6	13	
2	SUBI R1,R1,#8	7	8	9
2	BNEZ R1,LOOP	8	9	

每次迭代4个周期

转移仍然需要1个时钟周期

动态转移预测

- 解决方案: 2位策略, 只有当连续两次预测错误后才改变预测方向



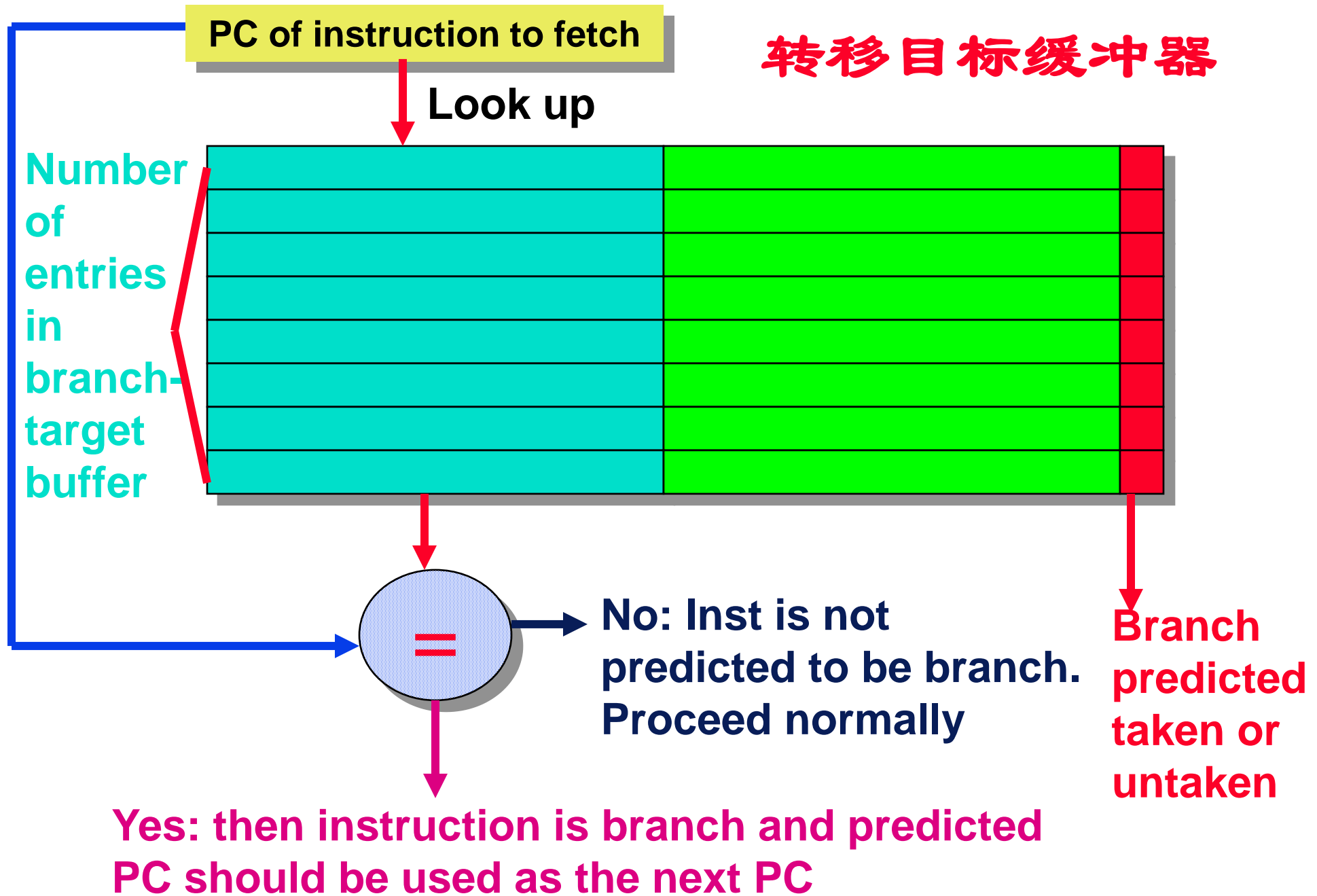
转移历史表的正确率

- 产生错误预测的原因：
 - 对该指令产生了错误猜测
 - 在对转移历史表进行检索时，使用了错误转移指令的转移历史信息
- 对于**4096**表项的情况，不同程序的转移预测率从**99% (nasa7, tomcatv)** 到 **82% (eqntott)**, 其中 **spice**为 **91%**、**gcc** 为**88%**
- **4096**个表项已经基本上可以与无穷表项的情况做的一样好！

预测的同时还需要地址

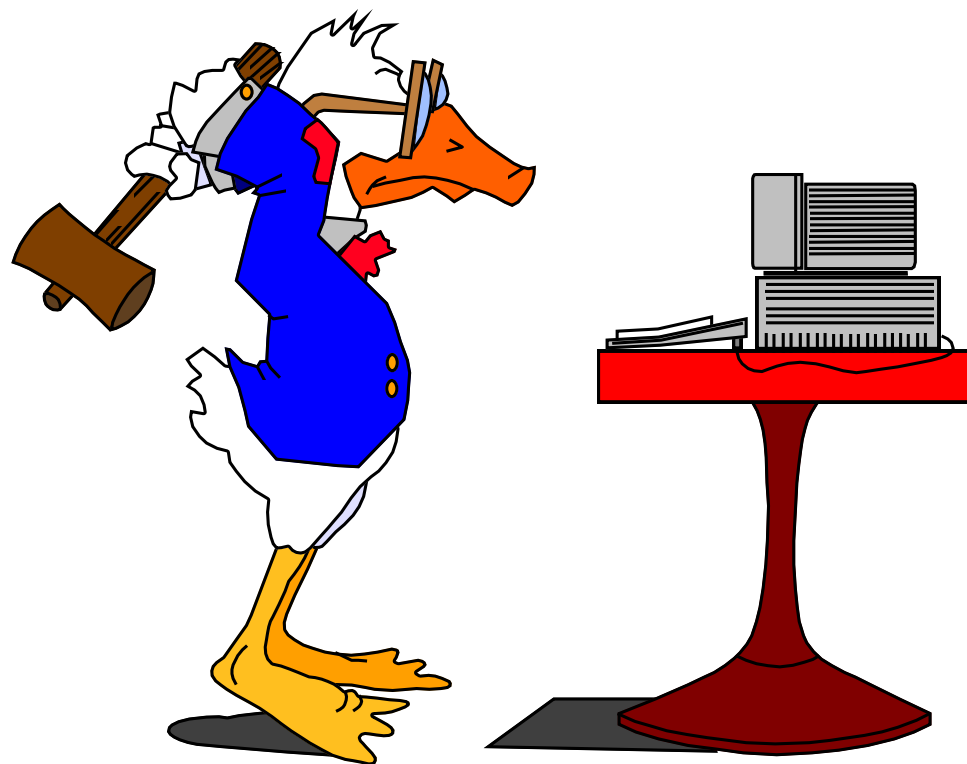
- 转移目标缓冲器 (BTB): 转移索引的地址可以得到预测方向 和 转移目标地址 (如果发生)
 - 注: 由于不能使用错误的转移地址, 现在必须对转移进行检测
- 返回预测的转移地址

转移目标缓冲器



动态转移预测小结

- 转移历史表: 为每个循环需要2位
- 转移目标缓冲器: 包括转移地址 和 预测



可获得更多的指令级并行的硬件支持

- 通过把转移指令转化为条件执行的指令来消除转移预测:

if (x) then A = B op C else NOP

- 如果条件不成立, 那么既不存储结果, 也不产生意外事件
 - **Alpha、MIPS、PowerPC、SPARC**的扩展指令系统体系结构具有 条件执行的移动指令 (**conditional move**); **PA-RISC**可以废止任何后续指令。
 - **EPIC**: 由**64**个可选的**1**位条件场位 (**condition field**) 来支持条件性执行
- 条件性指令的缺点
 - 即使是被废止的指令也需要占用一个时钟周期;
 - 如果在后期进行条件评测, 则产生暂停;
 - 复杂的条件不利于高效处理;
在流水线的晚期才知道条件是否满足

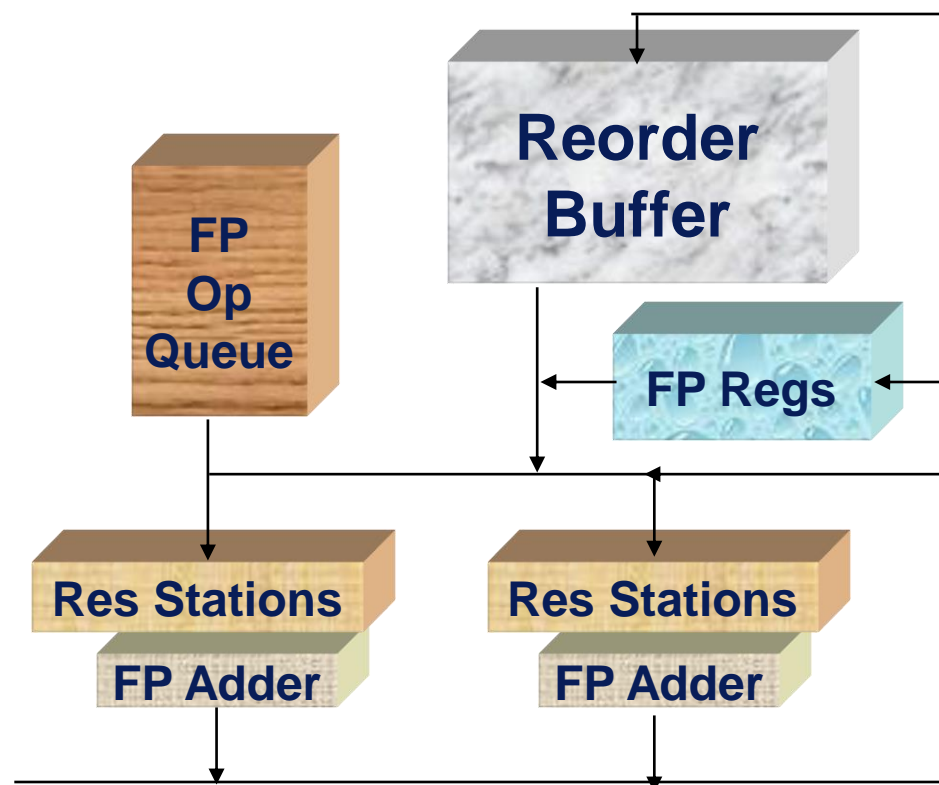
可获得更多的指令级并行的硬件支持（续一）

- **推测式执行 (Speculation)**：如果转移实际没有发生（硬件修复），允许指令不产生任何影响（包括意外事件）
- 通常，与动态指令调度结合
- 将结果的**推测式**旁路 和 结果的**实际**旁路 分离开
 - 当指令不再是推测式后，回写结果（指令提交）
 - 乱序执行，但按序提交

可获得更多的指令级并行的硬件支持（续二）

- 对于还没有提交的指令的结果需要硬件的缓冲器：**重排序缓冲器（*reorder buffer*）**

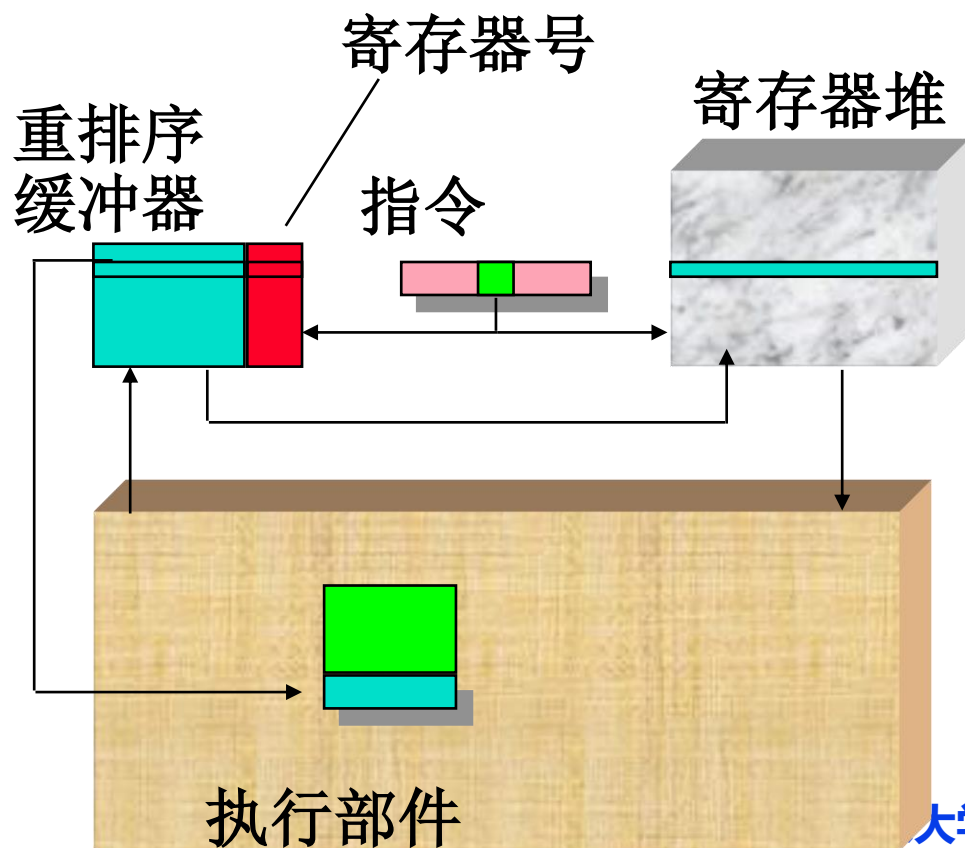
- 重排序缓冲器可以是操作数的源
- 一旦操作数提交，在寄存器中就可以找到结果
- 3个场位：指令类型、目标、数值
- 用重排序缓冲器的编号替代保留站
- 这样，就可以很容易地撤销由于错误预测的转移或意外事件导致的推测式指令



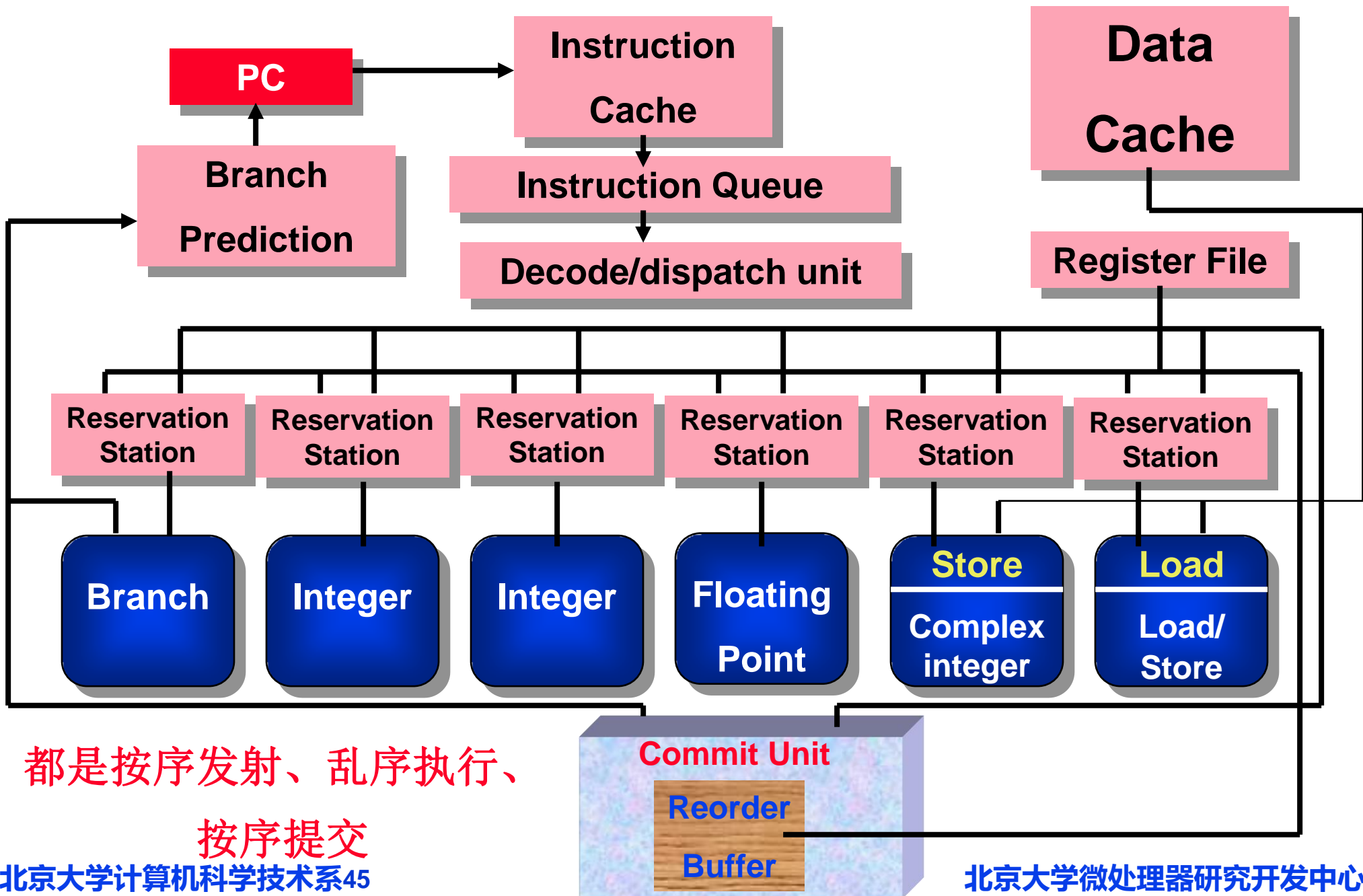
重排序缓冲器

跟踪尚未完成的对寄存器的修改操作

- 与寄存器访问并行, 根据优先级对重排序缓冲器进行并行查找
- 如果命中, 说明寄存器堆的内容是旧值,
 - 重排序缓冲器提供新值
 - 重排序缓冲器向功能部件提供需要旁路的新值
- 当指令完成时, 将职能转让给寄存器堆



PowerPC 604和Pentium Pro的动态调度



PowerPC 604和Pentium Pro的动态调度 (续)

参数	PowerPC	PentiumPro
发射的最多指令数/时钟	4	3
完成执行的最多指令数/时钟	6	5
提交的最多指令数/时钟	6	3
重排序缓冲器中的指令	16	40
换名缓冲器的数目	12 Int/8 FP	40
保留站的数目	12	20
整数功能部件的数目	2	2
浮点功能部件的数目	1	1
转移功能部件的数目	1	1
复杂整数的功能部件的数目	1	0
存储部件的数目	1	1 load +1 store

Pentium Pro中的动态调度

- PentiumPro并没有对80x86指令直接流水处理
- Pentiumpro译码部件将 Intel指令变换成72位的微操作（相当于MIPS的指令）
- 将这些微操作送到重排序缓冲器 和 保留站
- 需要一个周期来测定 80x86 指令的长度 + 两个以上的周期来创建微操作
- 大多数指令变换成一个到四个微操作
- 复杂的 80x86指令通过一个常规的微程序 (8K x 72 bits)来执行，它将发射一个较长的微操作序列

多发射机器的限制

- 指令级并行性的内在限制
 - 每5条指令1个转移: 如何使 5-路的 VLIW 忙碌?
 - 部件的延迟: 必须调度许多操作
 - 大约需要 流水线深度 \times 独立功能部件数目 的并行指令
- 加大硬件设计实现
 - 为了支持并行执行, 重置功能部件
 - 增加寄存器队的端口
 - 例如, VLIW可能需要整数寄存器堆具有7个读端口和3个写端口
 - 增加存储器的端口
 - 对超标量译码, 以及对时钟频率和流水线深度的影响

多发射机器的限制（续）

- 超标量或超长指令字实现的特定限制
 - 超标量中的译码发射
 - **VLIW**的代码大小：循环展开 + **VLIW**中的空场位
 - **VLIW**锁步（**lock step**） => 1种冒险 **hazard**，所有指令暂停
 - **VLIW**：二进制码兼容问题

总结

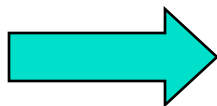
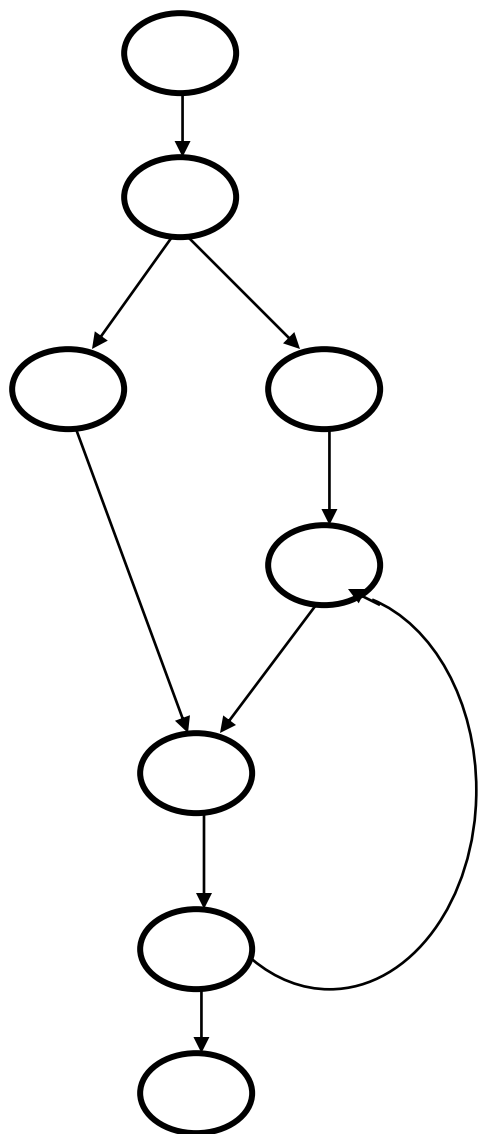
- **MIPS**指令系统体系结构使得流水线可见（延迟转移、延迟装入）
- 利用更深的流水线、并行性来获得更高的性能
- 超标量和超长指令字
 - **CPI < 1**
 - 动态发射 与 静态发射
 - 同时发射更多的指令，使指令间相关的损失有加大的倾向
- 软件流水
 - 可以使得流水线更加有效地工作、代码膨胀小、开销少

思考

如何使处理器更快、更有效？

从算法到程序

□ 算法



□ C Code example

```
typedef enum {
    ADD, MULT, MINUS, DIV, MOD,
    BAD} op_type;

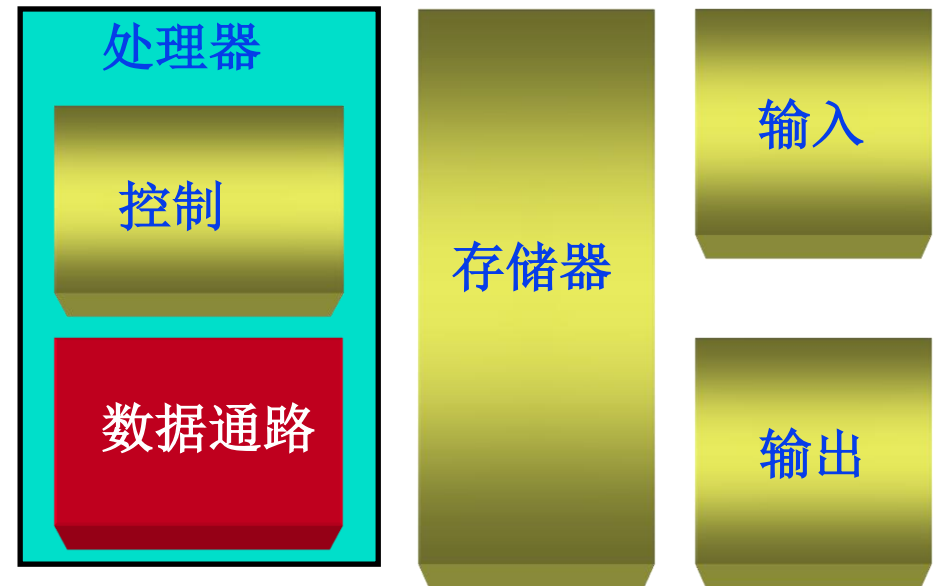
char unparse_symbol(op_type op)
{
    switch (op) {
        case ADD :
            return '+';
        case MULT:
            return '*';
        case MINUS:
            return '-';
        case DIV:
            return '/';
        case MOD:
            return '%';
        case BAD:
            return '?';
    }
}
```

程序在计算机系统中处理

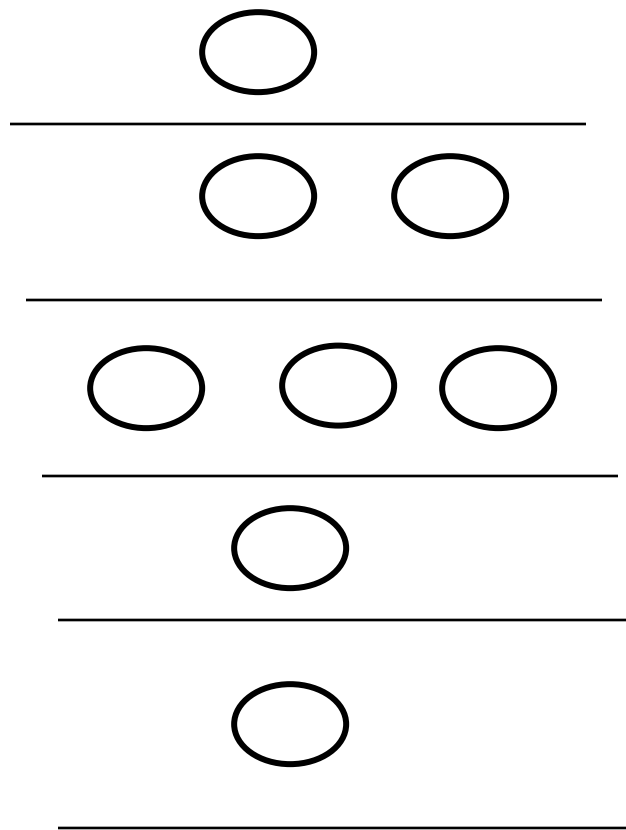
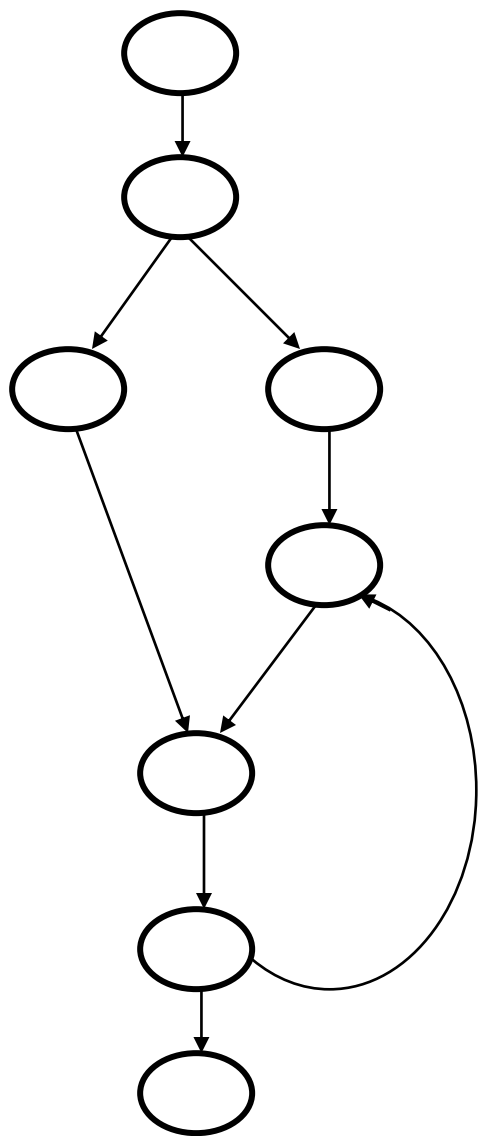
□ C Code

example

```
typedef enum  
{ADD, MULT, MINUS, DIV, MOD,  
BAD} op_type;  
  
char unparse_symbol(op_type op)  
{  
    switch (op) {  
        case ADD :  
            return '+';  
        case MULT:  
            return '*';  
        case MINUS:  
            return '-';  
        case DIV:  
            return '/';  
        case MOD:  
            return '%';  
        case BAD:  
            return '?';  
    }  
}
```



静态程序流程图到动态处理流程图



计算机系统中的七种序列

- 提交序列：指令退离处理器
- 完成序列：指令操作完成
- 执行序列：指令开始执行
- 发送序列：指令发送到执行部件
- 译码序列：指令开始译码
- 取指序列：处理器访问存储器中的指令
- 存储序列：程序在存储器中的存放地址