

计算机体系结构

新型计算机研究所

陈 衡

西一楼 A 段415室

Email: hengchen@xjtu.edu.cn

QQ群号: 786750600

第4章 指令级并行

- ◆ 4.1 指令级并行的概念
- ◆ 4.2 指令的动态调度
- ◆ 4.3 动态分支预测技术
- ◆ 4.4 多指令流出技术
- ◆ 4.5 循环展开和指令调度

4.1 指令级并行的概念

◆ 概念

- 几乎所有的处理机都利用流水线来使指令重叠并行执行，以达到提高性能的目的。这种指令之间存在的潜在并行性称为**指令级并行**。
- ILP: Instruction-Level Parallelism
- 可以使用ILP来衡量指令间的相关程度
- Average ILP = no. instruction / no. cyc required

```
code1: r1 ← r2 + 1  
       r3 ← r1 / 17  
       r4 ← r0 - r3
```

ILP = 1

串行执行

```
code2: r1 ← r2 + 1  
       r3 ← r9 / 17  
       r4 ← r0 - r10
```

ILP = 3

并行执行

4.1 指令级并行的概念

- ◆ 获得更多的指令级并行性。
 - 硬件+软件技术
- ◆ 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：
 - $CPI_{\text{流水线}} = CPI_{\text{理想}} + \text{停顿}_{\text{结构冲突}} + \text{停顿}_{\text{数据冲突}} + \text{停顿}_{\text{控制冲突}}$
 - 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
- ◆ 基本程序块
 - 基本程序块：一段除了入口和出口以外不包含其他分支的线性代码段。
 - 程序平均每5~7条指令就会有一个分支。

4.1 指令级并行的概念

- ◆ 循环级并行：使一个循环中的不同循环体并行执行
 - 开发循环体中存在的并行性
 - 最常见、最基本
 - 指令级并行研究的重点之一
 - 例如，考虑下述语句：
 for (i=1; i<=500; i=i+1)
 a[i]=a[i]+s;
● 每一次循环都可以与其他的循环重叠并行执行；
● 在每一次循环的内部，却没有任何的并行性。
- ◆ 最基本的开发循环级并行的技术
 - 循环展开 (loop unrolling) 技术
 - 采用向量指令和向量数据表示

4.1 指令级并行的概念

◆ 相关与流水线冲突

- 相关有三种类型：数据相关、名相关、控制相关
- 流水线冲突是指对于具体的流水线来说，由于相关的存在，使得指令流中的下一条指令不能在指定的时钟周期执行
 - 流水线冲突有三种类型：结构冲突、数据冲突、控制冲突
- 相关是程序固有的一种属性，它反映了程序中指令之间的相互依赖关系。
- 具体的一次相关是否会导致实际冲突的发生以及该冲突会带来多长的停顿，则是流水线的属性。

◆ 可以从两个方面来解决相关问题：

- 保持相关，但避免发生冲突。指令调度
- 通过代码变换，消除相关。


影响ILP的因素分析

- ◆ 程序顺序：由源程序确定的在完全串行方式下指令的执行顺序。
 - 只有在可能会导致错误的情况下，才保持程序顺序。
- ◆ 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**。
 - 保持异常行为是指：无论怎么改变指令的执行顺序，都不能改变程序中异常的发生情况。
 - 即原来程序中是怎么发生的，改变执行顺序后还是怎么发生。
 - 弱化为：指令执行顺序的改变不能导致程序中新发生异常。
 - 如果我们能做到**保持程序的数据相关和控制相关**，就能保持程序的数据流和异常行为。

影响ILP的因素分析

- ◆ 数据流：指数据值从其产生指令到其消费指令的实际流动
 - 分支指令使得数据流具有动态性，因为它使得给定指令的数据可以有多个来源。
 - 仅仅保持数据相关性是不够的，只有再加上保持控制顺序，才能够保持程序顺序。
- ◆ 例1：不能调整的指令顺序

```
DADDU    R1, R2, R3
BEQZ     R4, L1
DSUBU    R1, R5, R6
L1 : ...
OR       R7, R1, R8
```



影响ILP的因素分析

- ◆ 可以调整的指令顺序：有时不遵守控制相关，**既不影响异常行为，也不改变数据流**。可以大胆地进行指令调度，把失败分支中的指令调度到分支指令之前。

◆ 例2 DADDU R1, R2, R3
 BEQZ R12, Skipnext
 DSUBU R4, R5, R6
 DADDU R5, R4, R9

Skipnext:

 OR R7, R8, R9

- ◆ 注意：**假设已知R4在Skipnext之后不再用到**，而且DSUBU不会产生异常，那么就可以挪到BEQZ之前。

4.2 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。
- 不是在程序执行的过程中、而是在**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。
- 能够处理一些在编译时情况不明的相关（比如涉及存储器访问的相关），并简化了编译器；
- 能够使本来是面向某一流水线优化编译的代码在其他的流水线（动态调度）上也能高效地执行。
- **缺点**：以硬件复杂性的显著增加为代价

4.2.1 动态调度的基本思想

- ◆ 流水线的最大的局限性：

- 指令必须按序流出和执行

- ◆ 考虑下面一段代码：

DIV.D **F4**, F0, F2

SUB.D F10, **F4**, F6

ADD.D F12, F6, F14

- SUB.D指令与DIV.D指令关于**F4相关**，导致流水线停顿。
- ADD.D指令与流水线中的任何指令都没有关系，但也因此受阻。

4.2.1 动态调度的基本思想

- ◆ 到目前为止，介绍的MIPS指令执行都需要5个周期
 - 仅限于整数操作指令
 - 浮点数的操作比整数操作慢
- ◆ 概念
 - **延迟(latency)**：指产生结果的指令与使用结果的指令之间的周期数
 - **启动间隔(initiation interval)**是相同种类的两条指令之间的周期数。

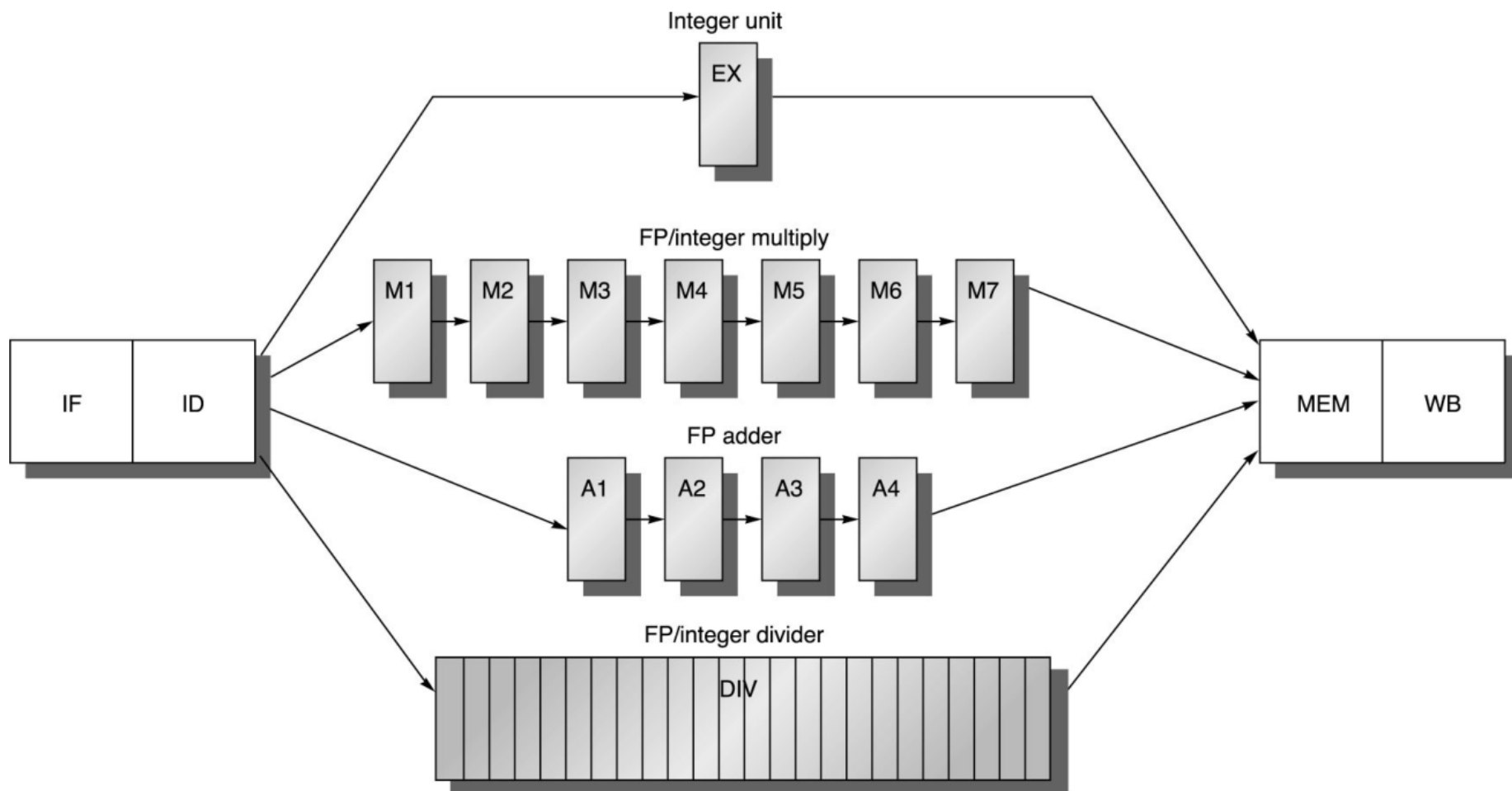
指令	延迟	启动间隔
ADD, SUB	0	1
LOAD, STORE	1	1
ADD.D, SUB.D	3	1
MUL.D	6	1
DIV.D	24	25

流水线指令举例

- ◆ 如果有如下的整数指令序列
 - ADD, SUB, AND, OR, SLLI等
- ◆ 则MIPS的5段流水线间没有延迟
 - **启动间隔为1**，意味着每个时钟周期可以开始下一条指令
 - **延迟为0**，意味这当下一条指令需要数据时，前一条指令已经生成了必要的结果数据。
- ◆ 但是，如果是如下的浮点指令序列
 - ADD.D, SUB.D
 - **启动间隔为1**，意味着SUB.D可以在ADD.D后开始
 - **延迟为3**，意味着如果SUB.D不需要使用ADD.D生成的数据，则流水线可以正常流动；
 - 但**如果有RAW依赖**，则必须在SUB.D前插入3个气泡

流水线指令举例

◆ 执行单元



流水线指令举例

- ◆ 浮点加法和乘法部件使用了流水线，但是除法部件没有使用流水线
 - 如果程序中有大量的除法指令，则执行时间会很长
 - 如果除法指令生成的数据很快就要被使用，程序的执行时间也会比较长
- ◆ **红色斜体字**表示需要读取数据
- ◆ **蓝色实体字**表示数据可用

MUL.D	IF	ID	<i>M1</i>	M2	M3	M4	M5	M6	M7	MEM	WB
ADD.D		IF	ID	<i>A1</i>	A2	A3	A4	MEM	WB		
L.D			IF	ID	<i>EX</i>	MEM	WB				
S.D				IF	ID	<i>EX</i>	<i>MEM</i>	WB			

流水线指令举例

◆ 由于RAW冲突引起的FP指令的停顿

指令	1	2	3	4	5	6	7	8
L.D F4 , 0(R2)	IF	ID	EX	MEM	WB			
MUL.D F0 , F4 , F6		IF	ID	stall	M1	M2	M3	M4
ADD.D F2, F0 , F8			IF	stall	ID	stall	stall	stall
S.D F2, 0(R2)				stall	IF	stall	stall	stall

指令	9	10	11	12	13	14	15	16	17
L.D F4, 0(R2)									
MUL.D F0 , F4, F6	M5	M6	M7	MEM	WB				
ADD.D F2 , F0 , F8	stall	stall	stall	A1	A2	A3	A4	MEM	WB
S.D F2 , 0(R2)	stall	stall	stall	ID	EX	stall	stall	stall	MEM

流水线指令举例

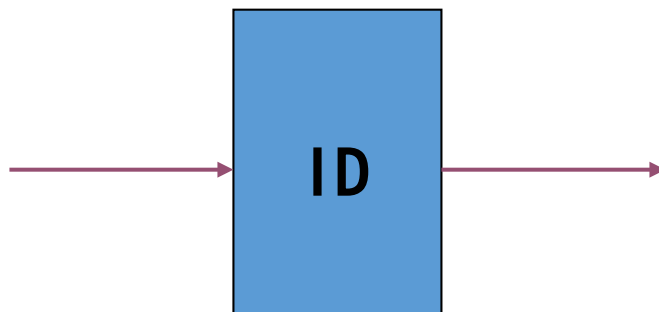
- ◆ 有可能出现2条指令同时执行到WB段
 - 出现**结构冲突**

指令	1	2	3	4	5	6	7	8
ADD.D	IF	ID	A1	A2	A3	A4	MEM	WB
LD		IF	ID	EX	MEM	WB		
DADD			IF	ID	EX	MEM	WB	
DADD				IF	ID	EX	MEM	WB

- ◆ 指令的完成时间可能与执行顺序不同
 - 引起**WAW冲突**

4.2.1 动态调度的基本思想

- ◆ 在前面的基本流水线中：

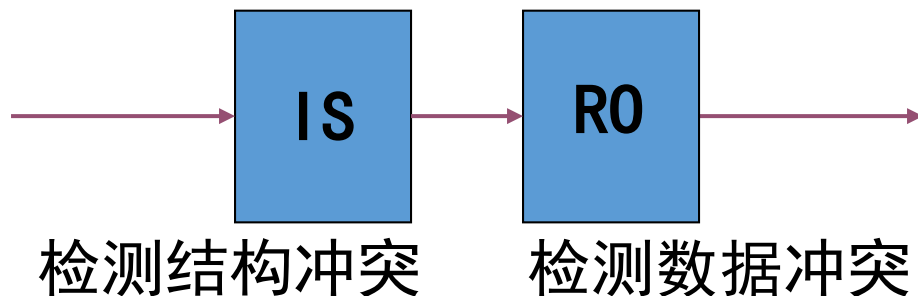


检测结构冲突
检测数据冲突

- ◆ 一旦一条指令受阻，其后的指令都将停顿。
 - 解决办法：允许**乱序**执行

4.2.1 动态调度的基本思想

- ◆ 为了允许乱序执行，我们将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。
 - **乱序执行** (out of order execution, OoO)



4.2.1 动态调度的基本思想

- ◆ 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
存在反相关 { SUB.D **F10**, F4, **F6**
 ADD.D **F6**, F8, F14 } 存在输出相关

- ◆ 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、或者流水功能部件、或者兼而有之。
 - 我们假设具有多个功能部件。

4.2.1 动态调度的基本思想

- ◆ 指令乱序完成带来的最大问题：
 - 异常处理比较复杂
 - 精确异常处理、不精确异常处理
- ◆ 动态调度要**保持正确的异常行为**
 - 只有那些在程序严格按程序顺序执行时会发生的异常，才能真正发生。
 - 保持正确的异常行为：对于一条会产生异常的指令来说，只有当处理机确切地知道该指令将被执行后，才允许它产生异常。
- ◆ 即使保持了正确的异常行为，动态调度处理机仍可能发生不精确异常。

4.2.1 动态调度的基本思想

- ◆ **不精确异常**：当执行指令 i 导致发生异常时，处理机的现场（状态）与严格按程序顺序执行时指令 i 的现场不同。
- ◆ **精确异常**：如果发生异常时，处理机的现场跟严格按程序顺序执行时指令 i 的现场相同。
- ◆ 不精确异常使得在异常处理后难以接着继续执行程序。发生不精确异常的原因：
 - 因为当发生异常（设为指令 i ）时：
 - 流水线可能已经执行完按程序顺序是位于指令 i 之后的指令；
 - 流水线可能还没完成按程序顺序时指令 i 之前的指令。

4.2.1 动态调度的基本思想

◆ 两种常用技术

- 动态调度，也称为乱序执行(out-of-order)
 - 记分牌算法
- 寄存器重命名
 - Tomasulo算法

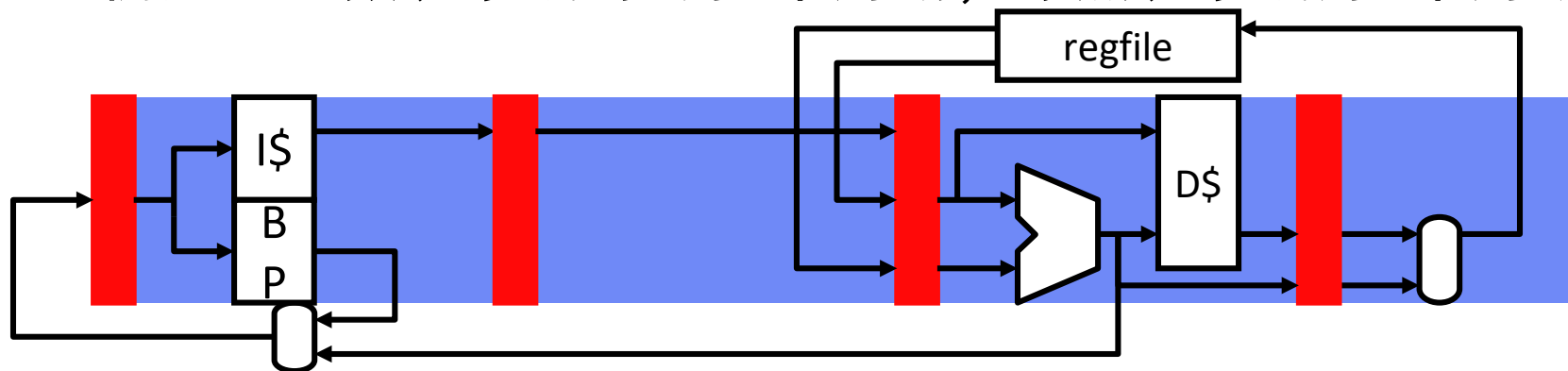
◆ 动态调度

- 指令在取指/译码后放入指令缓冲区(Instruction Buffer)
 - Instruction Window, Instruction Scheduler
- 硬件在每个时钟周期都需要检查指令中的源操作数寄存器是否就绪
- 当源数据就绪后，指令可以按任意顺序离开缓冲区

4.2.1 动态调度的基本思想

◆ MIPS的5段流水线

- IF、ID、EX、MEM、WB
- 也可以写成F、D、X、W，其中X是多时钟周期，包括了M
- 例如：整数流水线的X为1个周期；浮点流水线为3个周期



◆ 存在的问题

- **结构冲突**：每个阶段只有1个指令寄存器，后续指令无法“越过”前面停顿的指令
- ◆ **乱序流水线**：消除了结构冲突，后续指令先于前边的指令执行

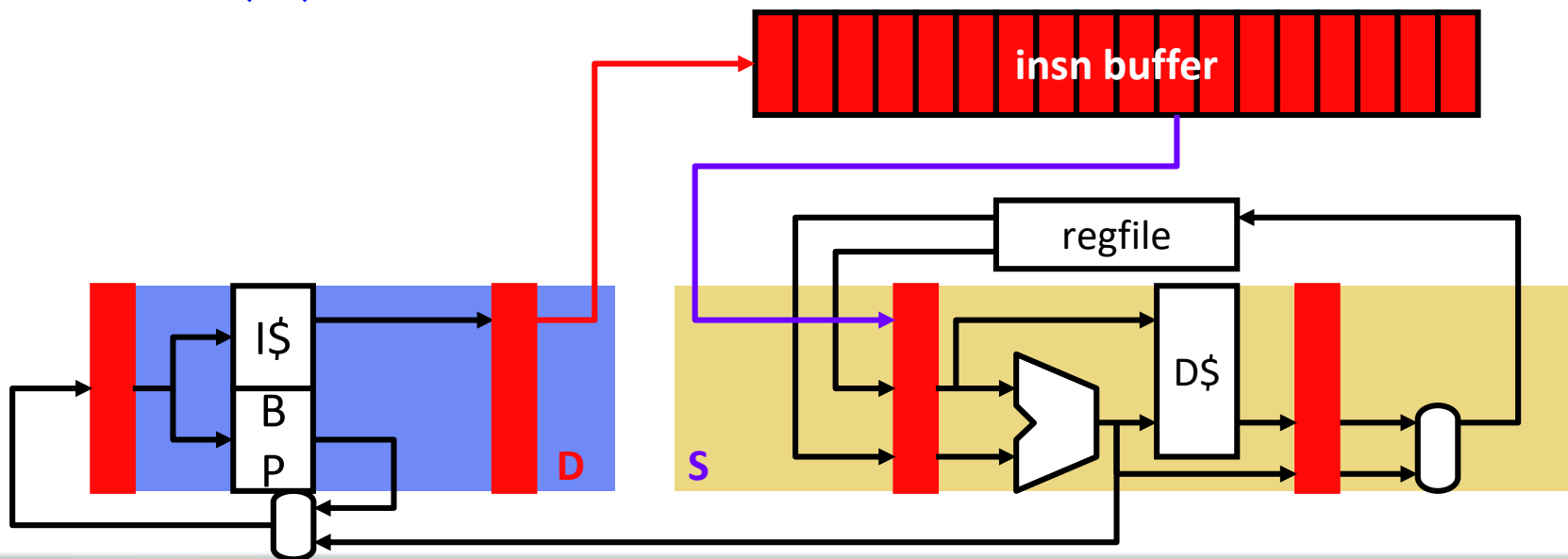
指令调度缓冲区

◆ 乱序流水线

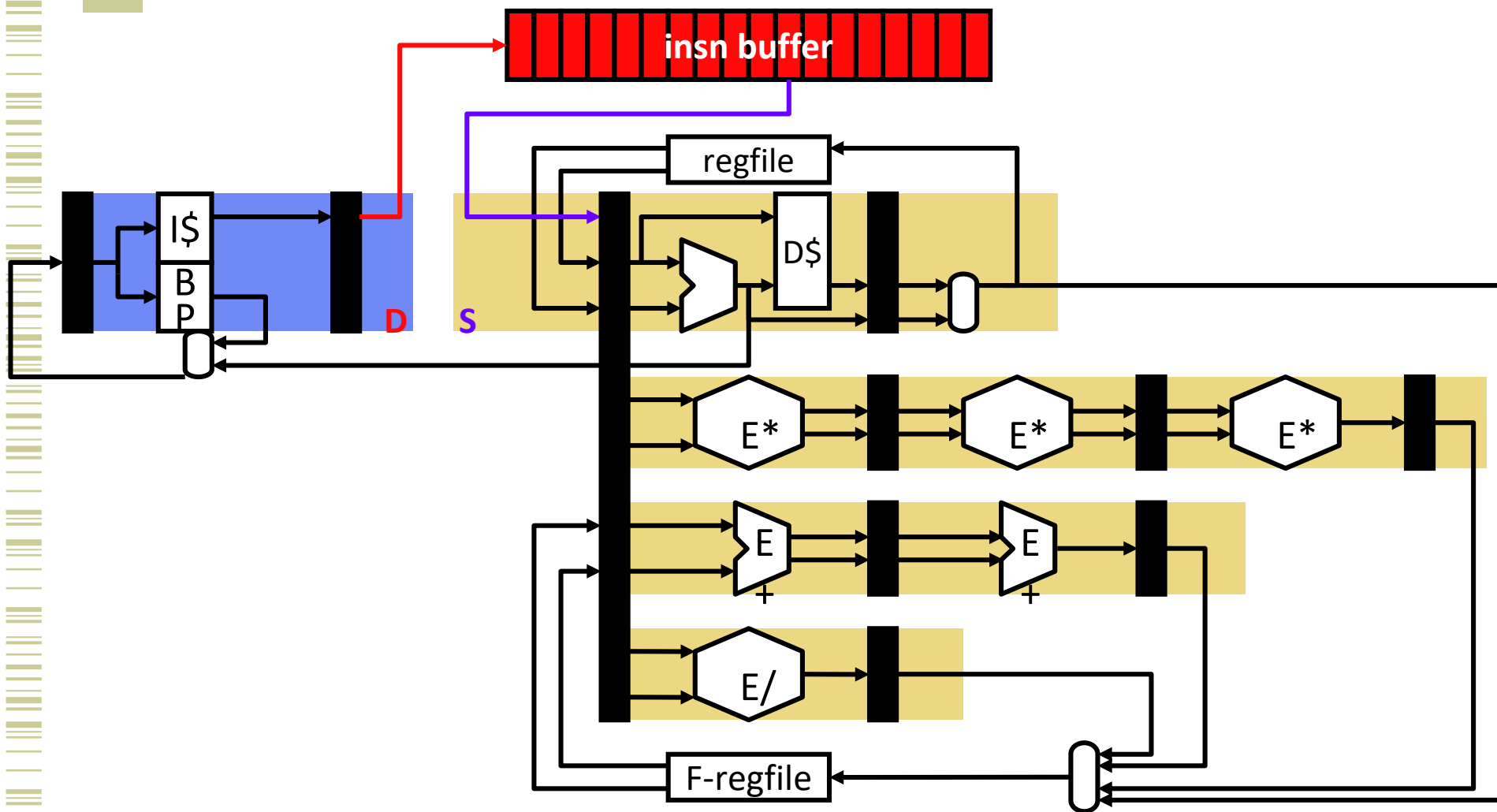
- 基于**指令调度缓冲区**
- 本质上是大量锁存器用来保存指令

◆ 把ID分为两段

- **Dispatch(D)**: 按序将译码后的指令保存在指令缓冲区
- **Issue(S)**: 乱序执行已经就绪的指令

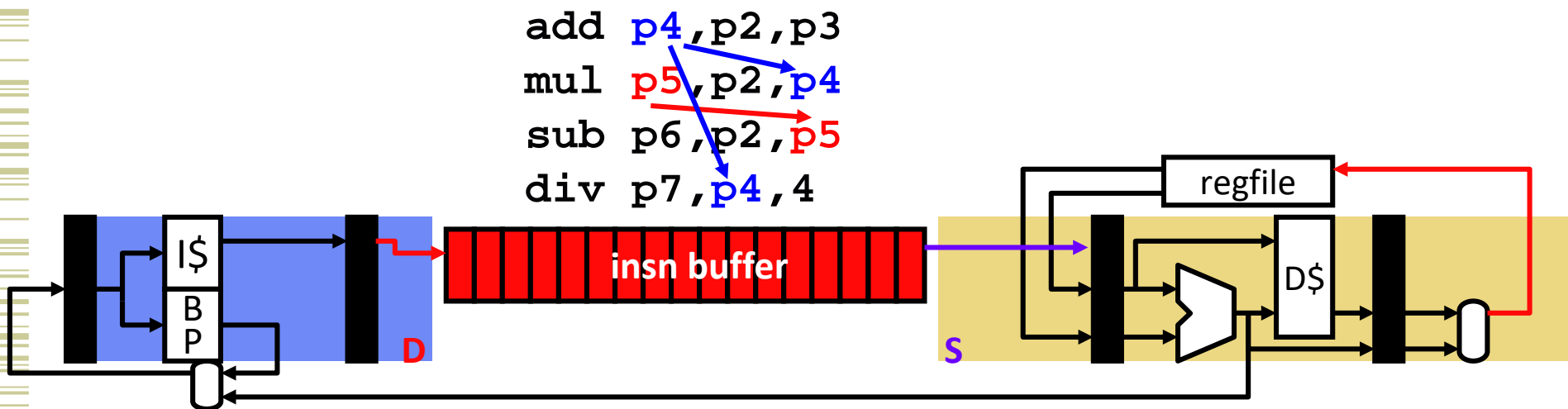


浮点流水线



◆ 不同的操作使用对应的执行段

动态调度



- ◆ 上例中，指令div可以先于指令sub执行
 - 指令mul执行时间比较长
- ◆ 乱序执行会引发新的数据冲突
 - **WAR** (Write-after-Read)，反相关
 - **WAW** (Write-after-Write)，输出相关

动态调度引发的新冲突

◆ WAW, 输出相关

- 乱序执行处理器
- 两条指令同时写一个寄存器

add **r1**, r2, r3

sub r3, r2, r1

mul r3, r2, r3

div **r1**, r1, 4

- 如果div指令在add指令前先执行完, 并写了寄存器r1。则r1的数据错误

◆ WAR, 反相关

- 乱序执行处理器
- 在前边指令未读取寄存器数据时, 后续指令先完成写寄存器操作

add r1, r2, r3

sub r3, r2, **r1**

mul r3, r2, r3

div **r1**, r1, 4

- 如果div指令在sub指令前先执行完, 并写了寄存器r1。sub指令执行出错

寄存器重命名

- ◆ 当出现WAR、WAW冲突时，只需让**流水线停顿**就可以避免这两种数据冲突
- ◆ 但，WAR和WAW的相关是“**假相关**”
 - 相关是基于名称/位置而不是数据值
 - 如果有无数的寄存器，则WAR和WAW是可以消除的
 - 思路：增加物理寄存器的数量（对程序员不可见）
 - **动态重命名**使指令使用新的寄存器，消除WAR和WAW
- ◆ 例如：寄存器r1, r2, r3；物理寄存器p1—p7
 - 初始映射关系：r1→p1, r2→p2, r3→p3

映射表

r1	r2	r3
p1	p2	p3
p4	p2	p3
p4	p2	p5
p4	p2	p6

未使用列表

p4, p5, p6, p7
p5, p6, p7
p6, p7
p7

初始指令

```
add r1, r2, r3
sub r3, r2, r1
mul r3, r2, r3
div r1, r1, 4
```

重命名后指令

```
add p4, p2, p3
sub p5, p2, p4
mul p6, p2, p5
div p7, p4, 4
```

4.2.1 动态调度的基本思想

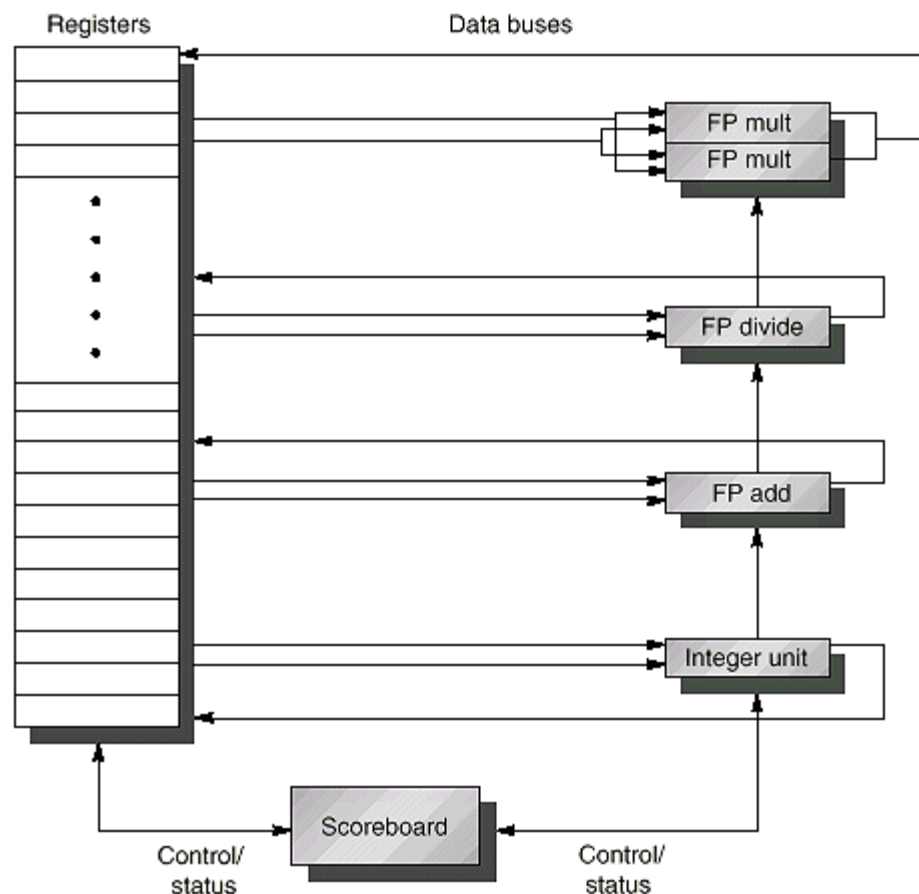
- ◆ 动态调度
 - 全部基于硬件，常称为**乱序执行**(**OoO**: Out-of-Order)
- ◆ 读入大量指令保存到指令缓冲区
 - 使用**分支预测**技术进行前瞻执行
 - 若分支预测错误则清空流水线
- ◆ **重命名技术**避免WAW和WAR的假相关
- ◆ 尽可能快的执行指令
 - 寄存器依赖容易获取
 - 内存依赖的处理比较棘手
- ◆ 按需完成指令
- ◆ 目前计算机的指令调度缓冲区长度: 100+

记分牌算法技术要点

- ◆ 核心思想
 - 允许位于stall指令后面的指令继续执行
 - 乱序执行 (OoO)
- ◆ 记分牌算法采用**集中式硬件**机制，通过互锁机制消除相关，指令执行时满足以下条件：
 - 没有冲突
 - 指令的操作数就绪
- ◆ 通过硬件为指令调度缓冲区中的指令动态构造依赖关系图
- ◆ 可将记分牌相关硬件理解为是一种“**数据结构**”，它为处理器的所有部件协同工作提供必要的信息。

典型的记分牌结构

- ◆ 记分牌算法在1963年首次出现在CDC6600中
 - 4个FP units
 - 5个memory reference units
 - 7个integer units
- ◆ 课堂讲授的例子在此基础上进行了修改，使其更适合MIPS
 - 2个FP multiply
 - 1个FP adder
 - 1个FP divider
 - 1个integer



记分牌算法技术要点

- ◆ 乱序执行会引发WAR、WAW冲突
- ◆ **WAR的一般解决方案**
 - 对操作排队
 - 仅在读操作数阶段读寄存器
- ◆ 对**WAW**而言，检测到相关后，停止发射前一条指令，直到前一条指令完成
- ◆ 要提高效率，需要有多条指令进入执行阶段
 - 必须有多个执行部件或执行部件是流水化的
- ◆ 记分牌保存相关操作和状态
- ◆ 记分牌用四段代替ID，EX，WB 三段

记分牌控制的四阶段

◆ 1、Issue—指令译码，检测结构相关

- 如果当前指令所使用的功能部件空闲，并且没有其他活动的指令使用相同的目的寄存器（WAW），记分牌发射该指令到功能部件，并更新记分牌内部数据，**如果有结构相关或WAW相关，则该指令的发射暂停**，并且也不发射后继指令，直到相关解除。

◆ 2、Read operands—没有数据相关时，读操作数

- 如果先前已发射的正在运行的指令不对当前指令的源操作数寄存器进行写操作，或者一个正在工作的功能部件已经完成了对该寄存器的写操作，则该操作数有效。操作数有效时，记分牌控制功能部件读操作数，准备执行。
- 记分牌在这一步动态地解决了RAW相关，指令可能会乱序执行。

记分牌控制的四阶段

◆ 3.Execution—取到操作数后执行 (EX)

- 接收到操作数后，功能部件开始执行。当计算出结果后，它通知记分牌，可以结束该条指令的执行。

◆ 4.Write result—执行完成 (WB)

- 一旦记分牌得到功能部件执行完毕的信息后，记分牌检测WAR相关，如果没有WAR相关，就写结果，如果有 WAR 相关，则暂停该条指令。

Example:

DIVD F0,F2,F4

ADDD F10,F0,**F8**

SUBD **F8**,F8,F14

- CDC 6600 scoreboard 将暂停 SUBD 直到ADDD 读取操作数后，才进入WR段处理。

记分牌的数据结构

◆ 1、指令状态表

- 记录正在执行的各条指令所处的状态步

◆ 2、功能部件(FU)状态表。用9个域记录

Busy	– 指示该部件是否空闲
Op	– 该部件所完成的操作
Fi	– 其目的寄存器编号
Fj, Fk	– 源寄存器编号
Qj, Qk	– 产生源操作数Fj, Fk的功能部件
Rj, Rk	– 标识源操作数Fj, Fk是否就绪的标志

◆ 3、寄存器结果状态 **Result**

- 如果存在功能部件对某一寄存器进行**写操作**，指示具体是哪个功能部件对该寄存器进行写操作。
- 如果没有指令对该寄存器进行写操作，则该域为Blank(0)

记分牌流水线控制

Instruction status	Wait until	Bookkeeping
Issue	Not Busy(FU) && not Result(D)	Busy(FU) ← yes; Op(FU) ← op; Fi(FU) ← 'D'; Fj(FU) ← 'S1'; Fk(FU) ← 'S2'; Qj ← Result('S1'); Qk ← Result('S2'); Rj ← not Qj; Rk ← not Qk; Result('D') ← FU;
Read operands	Rj == Yes && Rk == Yes	Rj ← No; Rk ← No
Execution complete	Functional unit done	
Write result	$\forall f($ $(Fj(f) \neq Fi(FU) \parallel Rj(f) == No)$ $\&\&$ $(Fk(f) \neq Fi(FU) \parallel Rk(f) == No)$ $)$	$\forall f(\text{if } Qj(f) = FU \text{ then } Rj(f) \leftarrow \text{Yes});$ $\forall f(\text{if } Qk(f) = FU \text{ then } Rk(f) \leftarrow \text{Yes});$ Result(Fi(FU)) ← 0; Busy(FU) ← No

记分牌算法举例

◆ MIPS中各功能单元基本参数信息

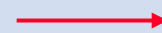
- 各执行单元没有使用流水线技术

执行单元 (FU)	FU的数量	EX时钟周期
Integer	1	1
FP Multiply	2	10
FP Add	1	2
FP Divide	1	40

◆ 举例

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**

真实的数据相关 (RAW)



反相关 (WAR)



输出相关 (WAW)



记分牌算法举例

◆ 真实数据依赖关系RAW

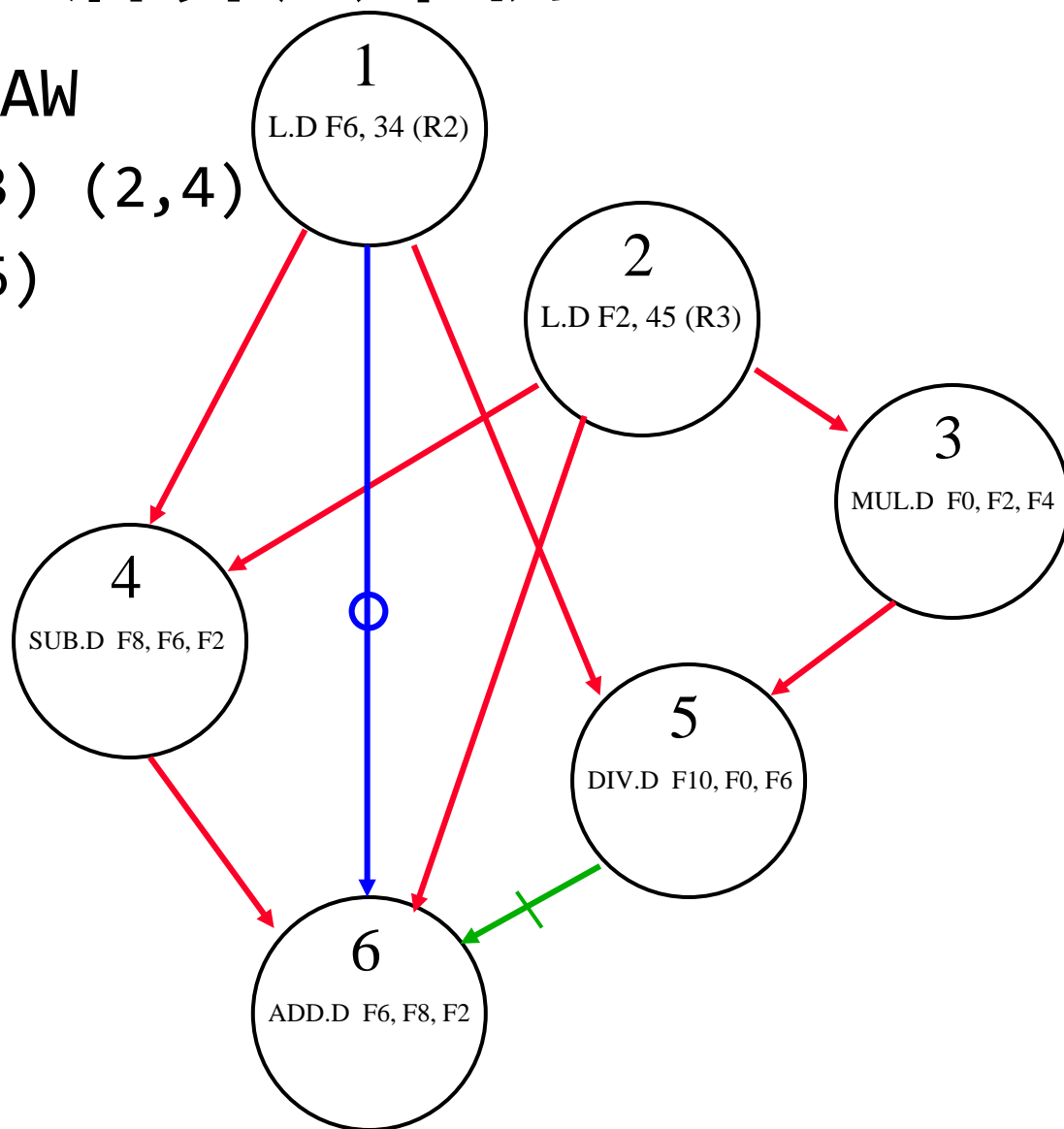
- (1,4) (1,5) (2,3) (2,4)
- (2,6) (3,5) (6,6)

◆ 反相关WAR

- (5,6)

◆ 输出相关WAW

- (1,6)



1	L.D	F6, 34(R2)
2	L.D	F2, 45(R3)
3	MUL.D	F0, F2, F4
4	SUB.D	F8, F6, F2
5	DIV.D	F10, F0, F6
6	ADD.D	F6, F8, F2

记分牌算法举例

<u>Instruction status</u>			
Instruction		j	k
LD	F6	34+	R2
LD	F2	45+	R3
MULT	F0	F2	F4
SUBD	F8	F6	F2
DIVD	F10	F0	F6
ADDD	F6	F8	F2

	Read	Execute	Write
Issue	operand	complete	Result

Functional unit status

<i>Time</i>	<i>Name</i>
Integer	
Mult1	
Mult2	
Add	
Divide	

		<i>dest</i>	<i>S1</i>	<i>S2</i>	<i>FU for j</i>	<i>FU for k</i>	<i>Fj?</i>	<i>Fk?</i>
<i>Busy</i>	<i>Op</i>	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>	<i>Rk</i>

No
No
No
No
No

Register result status

Clock	$F0$	$F2$	$F4$	$F6$	$F8$	$F10$	$F12$	\dots	$F30$
FU									

记分牌算法举例-时钟周期1

Instruction status

Instruction	<i>j</i>	<i>k</i>
LD F6 34+ R2		
LD F2 45+ R3		
MULT F0 F2 F4		
SUBD F8 F6 F2		
DIVD F10 F0 F6		
ADDD F6 F8 F2		

Functional unit status

Time	Name
	Integer
	Mult1
	Mult2
	Add
	Divide

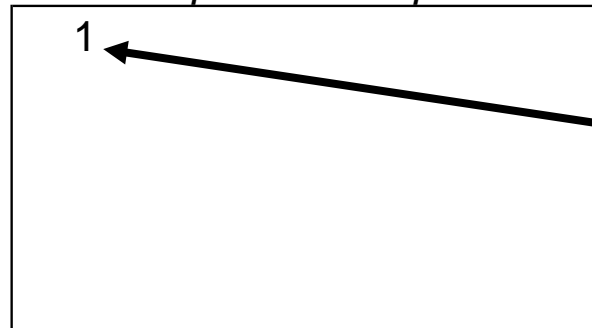
Register result status

Clock

1

FU

Read Executic Write
Issue operand complet Result



Issue LD #1

Shows in which cycle the operation occurred.

		<i>dest</i>	<i>S1</i>	<i>S2</i>	<i>FU for j</i>	<i>FU for k</i>	<i>Fj?</i>	<i>Fk?</i>
<i>Busy</i>	<i>Op</i>	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>	<i>Rk</i>
Yes	Load	F6		R2				Yes
No								
No								
No								
No								

<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
Integer								

记分牌算法举例-时钟周期2

Instruction status

Instruction	<i>j</i>	<i>k</i>
LD F6 34+ R2		
LD F2 45+ R3		
MULT F0 F2 F4		
SUBD F8 F6 F2		
DIVD F10 F0 F6		
ADDD F6 F8 F2		

Read Executic Write

Issue operand complet Result

1	2
---	---

LD #2 can't issue since integer unit is busy.
MULT can't issue because we require in-order issue.

Functional unit status

Time Name

Integer
Mult1
Mult2
Add
Divide

<i>Busy</i>	<i>Op</i>	<i>dest</i> <i>Fi</i>	<i>S1</i> <i>Fj</i>	<i>S2</i> <i>Fk</i>	<i>FU for j</i> <i>Qj</i>	<i>FU for k</i> <i>Qk</i>	<i>Fj?</i> <i>Rj</i>	<i>Fk?</i> <i>Rk</i>
Yes	Load	F6		R2				Yes
No								
No								
No								
No								

Register result status

Clock

2

FU

<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
Integer								

记分牌算法举例-时钟周期3

Instruction status

Instruction	<i>j</i>	<i>k</i>
LD F6 34+ R2		
LD F2 45+ R3		
MULT F0 F2 F4		
SUBD F8 F6 F2		
DIVD F10 F0 F6		
ADDD F6 F8 F2		

Read Executic Write

Issue operand complet Result

1	2	3
---	---	---

Functional unit status

Time Name

Integer
Mult1
Mult2
Add
Divide

<i>Busy</i>	<i>Op</i>	<i>dest</i> <i>Fi</i>	<i>S1</i> <i>Fj</i>	<i>S2</i> <i>Fk</i>	<i>FU for j</i> <i>Qj</i>	<i>FU for k</i> <i>Qk</i>	<i>Fj?</i> <i>Rj</i>	<i>Fk?</i> <i>Rk</i>
Yes	Load	F6		R2				Yes
No								
No								
No								
No								

Register result status

Clock

3

FU

<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
Integer								

记分牌算法举例-时钟周期4

Instruction status

Instruction	<i>j</i>	<i>k</i>
LD F6 34+ R2		
LD F2 45+ R3		
MULT F0 F2 F4		
SUBD F8 F6 F2		
DIVD F10 F0 F6		
ADDD F6 F8 F2		

Read Executic Write

Issue operand complet Result

1	2	3	4

Functional unit status

Time Name

Integer
Mult1
Mult2
Add
Divide

<i>Busy</i>	<i>Op</i>	<i>dest</i> <i>Fi</i>	<i>S1</i> <i>Fj</i>	<i>S2</i> <i>Fk</i>	<i>FU for j</i> <i>Qj</i>	<i>FU for k</i> <i>Qk</i>	<i>Fj?</i> <i>Rj</i>	<i>Fk?</i> <i>Rk</i>
Yes	Load	F6		R2				Yes
No								
No								
No								
No								

Register result status

Clock

4

FU

F0 F2 F4 F6 F8 F10 F12 ... F30

Integer

记分牌算法举例-时钟周期5

Instruction status

Instruction	<i>j</i>	<i>k</i>
LD F6 34+ R2		
LD F2 45+ R3		
MULT F0 F2 F4		
SUBD F8 F6 F2		
DIVD F10 F0 F6		
ADDD F6 F8 F2		

Functional unit status

Time Name

Integer
Mult1
Mult2
Add
Divide

Read Executic Write
Issue operand complet Result

1	2	3	4
5			

Issue LD #2 since integer unit is now free.

		<i>dest</i>	<i>S1</i>	<i>S2</i>	<i>FU for j</i>	<i>FU for k</i>	<i>Fj?</i>	<i>Fk?</i>
<i>Busy</i>	<i>Op</i>	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>	<i>Rk</i>
Yes	Load	F2		R3				Yes
No								
No								
No								
No								

Register result status

Clock

5

FU

<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
	Integer							

记分牌算法举例-时钟周期6

Instruction status

Instruction	<i>j</i>	<i>k</i>
LD F6 34+ R2		
LD F2 45+ R3		
MULT F0 F2 F4		
SUBD F8 F6 F2		
DIVD F10 F0 F6		
ADDD F6 F8 F2		

Read Executic Write

Issue	operand	complet	Result
1	2	3	4
5	6		
6			

Issue MULT.

Functional unit status

Time Name

Integer
Mult1
Mult2
Add
Divide

Busy	Op	dest <i>F_i</i>	<i>S1</i> <i>F_j</i>	<i>S2</i> <i>F_k</i>	<i>FU for j</i> <i>Q_j</i>	<i>FU for k</i> <i>Q_k</i>	<i>F_j?</i> <i>R_j</i>	<i>F_k?</i> <i>R_k</i>
Yes	Load	F2		R3				Yes
Yes	Mult	F0	F2	F4	Integer		No	Yes
No								
No								
No								

Register result status

Clock

6

FU

<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
Mult1	Integer							

记分牌算法举例-时钟周期7

Instruction status

Instruction	<i>j</i>	<i>k</i>
LD F6 34+ R2		
LD F2 45+ R3		
MULT F0 F2 F4		
SUBD F8 F6 F2		
DIVD F10 F0 F6		
ADDD F6 F8 F2		

Functional unit status

Time Name

Integer
Mult1
Mult2
Add
Divide

Issue	Read operand	Executic complet	Write Result
1	2	3	4
5	6	7	
6			
7			

MULT can't read its operands (F2) because LD #2 hasn't finished.

		<i>dest</i>	<i>S1</i>	<i>S2</i>	<i>FU for j</i>	<i>FU for k</i>	<i>Fj?</i>	<i>Fk?</i>
<i>Busy</i>	<i>Op</i>	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>	<i>Rk</i>
Yes	Load	F2		R3				Yes
Yes	Mult	F0	F2	F4	Integer		No	Yes
No								
Yes	Sub	F8	F6	F2		Integer	Yes	No
No								

Register result status

Clock

7

FU

<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
Mult1	Integer			Add				

记分牌算法举例-时钟周期8a

Instruction status

Instruction	<i>j</i>	<i>k</i>
LD F6 34+ R2		
LD F2 45+ R3		
MULT F0 F2 F4		
SUBD F8 F6 F2		
DIVD F10 F0 F6		
ADDD F6 F8 F2		

Read Executic Write

Issue operand complet Result

1	2	3	4
5	6	7	
6			
7			
8			

DIVD issues.
MULT and SUBD both
waiting for F2.

Functional unit status

Time Name

Integer
Mult1
Mult2
Add
Divide

<i>Busy</i>	<i>Op</i>	<i>dest</i> <i>Fi</i>	<i>S1</i> <i>Fj</i>	<i>S2</i> <i>Fk</i>	<i>FU for j</i> <i>Qj</i>	<i>FU for k</i> <i>Qk</i>	<i>Fj?</i> <i>Rj</i>	<i>Fk?</i> <i>Rk</i>
Yes	Load	F2		R3				Yes
Yes	Mult	F0	F2	F4	Integer		No	Yes
No								
Yes	Sub	F8	F6	F2		Integer	Yes	No
Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

Clock

8

FU

<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
Mult1	Integer			Add	Divide			

记分牌算法举例-时钟周期8b

Instruction status

Instruction	<i>j</i>	<i>k</i>
LD F6 34+ R2		
LD F2 45+ R3		
MULT F0 F2 F4		
SUBD F8 F6 F2		
DIVD F10 F0 F6		
ADDD F6 F8 F2		

Read Executic Write

Issue operand complet Result

1	2	3	4
5	6	7	8
6			
7			
8			

LD #2 writes F2.

Functional unit status

Time Name

Integer
Mult1
Mult2
Add
Divide

<i>Busy</i>	<i>Op</i>	<i>dest</i> <i>Fi</i>	<i>S1</i> <i>Fj</i>	<i>S2</i> <i>Fk</i>	<i>FU for j</i> <i>Qj</i>	<i>FU for k</i> <i>Qk</i>	<i>Fj?</i> <i>Rj</i>	<i>Fk?</i> <i>Rk</i>
No								
Yes	Mult	F0	F2	F4			Yes	Yes
No								
Yes	Sub	F8	F6	F2			Yes	Yes
Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

Clock

8

FU

<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
Mult1				Add	Divide			

记分牌算法举例-时钟周期9

Instruction status

Instruction	<i>j</i>	<i>k</i>
LD F6 34+ R2		
LD F2 45+ R3		
MULT F0 F2 F4		
SUBD F8 F6 F2		
DIVD F10 F0 F6		
ADDD F6 F8 F2		

Read Executic Write

Issue operand complet Result

1	2	3	4
5	6	7	8
6	9		
7	9		
8			

Now MULT and SUBD can both read F2.
How can both instructions do this at the same time??

Functional unit status

Time Name

Integer	
10 Mult1	
Mult2	
2 Add	
Divide	

<i>Busy</i>	<i>Op</i>	<i>dest</i> <i>Fi</i>	<i>S1</i> <i>Fj</i>	<i>S2</i> <i>Fk</i>	<i>FU for j</i> <i>Qj</i>	<i>FU for k</i> <i>Qk</i>	<i>Fj?</i> <i>Rj</i>	<i>Fk?</i> <i>Rk</i>
-------------	-----------	--------------------------	------------------------	------------------------	------------------------------	------------------------------	-------------------------	-------------------------

No								
Yes	Mult	F0	F2	F4			Yes	Yes
No								
Yes	Sub	F8	F6	F2			Yes	Yes
Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

Clock

9

FU

<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
-----------	-----------	-----------	-----------	-----------	------------	------------	-----	------------

Mult1				Add	Divide			
-------	--	--	--	-----	--------	--	--	--

记分牌算法举例-时钟周期11

Instruction status

Instruction	<i>j</i>	<i>k</i>
LD F6 34+ R2		
LD F2 45+ R3		
MULT F0 F2 F4		
SUBD F8 F6 F2		
DIVD F10 F0 F6		
ADDD F6 F8 F2		

Read Executic Write

Issue operand complet Result

1	2	3	4
5	6	7	8
6	9		
7	9	11	
8			

ADDD can't start because add unit is busy.

Functional unit status

Time Name

Integer
8 Mult1
Mult2
0 Add
Divide

<i>Busy</i>	<i>Op</i>	<i>dest</i> <i>Fi</i>	<i>S1</i> <i>Fj</i>	<i>S2</i> <i>Fk</i>	<i>FU for j</i> <i>Qj</i>	<i>FU for k</i> <i>Qk</i>	<i>Fj?</i> <i>Rj</i>	<i>Fk?</i> <i>Rk</i>
-------------	-----------	--------------------------	------------------------	------------------------	------------------------------	------------------------------	-------------------------	-------------------------

No								
Yes	Mult	F0	F2	F4			Yes	Yes
No								
Yes	Sub	F8	F6	F2			Yes	Yes
Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

Clock

11

FU

<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
-----------	-----------	-----------	-----------	-----------	------------	------------	-----	------------

Mult1				Add	Divide			
-------	--	--	--	-----	--------	--	--	--

记分牌算法举例-时钟周期12

Instruction status

Instruction	<i>j</i>	<i>k</i>
LD F6 34+ R2		
LD F2 45+ R3		
MULT F0 F2 F4		
SUBD F8 F6 F2		
DIVD F10 F0 F6		
ADDD F6 F8 F2		

Read Executic Write

Issue operand complet Result

1	2	3	4
5	6	7	8
6	9		
7	9	11	12
8			

SUBD finishes.
DIVD waiting for F0.

Functional unit status

Time Name

Integer
7 Mult1
Mult2
Add
Divide

<i>Busy</i>	<i>Op</i>	<i>dest</i> <i>Fi</i>	<i>S1</i> <i>Fj</i>	<i>S2</i> <i>Fk</i>	<i>FU for j</i> <i>Qj</i>	<i>FU for k</i> <i>Qk</i>	<i>Fj?</i> <i>Rj</i>	<i>Fk?</i> <i>Rk</i>
-------------	-----------	--------------------------	------------------------	------------------------	------------------------------	------------------------------	-------------------------	-------------------------

No								
Yes	Mult	F0	F2	F4			Yes	Yes
No								
No								
Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

Clock

12

FU

<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
-----------	-----------	-----------	-----------	-----------	------------	------------	-----	------------

Mult1						Divide		
-------	--	--	--	--	--	--------	--	--

记分牌算法举例-时钟周期13

Instruction status

Instruction	<i>j</i>	<i>k</i>
LD F6 34+ R2		
LD F2 45+ R3		
MULT F0 F2 F4		
SUBD F8 F6 F2		
DIVD F10 F0 F6		
ADDD F6 F8 F2		

Read Executic Write

Issue	operand	complet	Result
1	2	3	4
5	6	7	8
6	9		
7	9	11	12
8			
13			

ADDD issues.

Functional unit status

Time Name

Integer
6 Mult1
Mult2
Add
Divide

Busy	Op	dest <i>Fi</i>	<i>S1</i> <i>Fj</i>	<i>S2</i> <i>Fk</i>	<i>FU for j</i> <i>Qj</i>	<i>FU for k</i> <i>Qk</i>	<i>Fj?</i> <i>Rj</i>	<i>Fk?</i> <i>Rk</i>
No								
Yes	Mult	F0	F2	F4			Yes	Yes
No								
Yes	Add	F6	F8	F2			Yes	Yes
Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

Clock

13

FU

<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
Mult1			Add		Divide			

记分牌算法举例-时钟周期14

Instruction status

Instruction	<i>j</i>	<i>k</i>
LD F6 34+ R2		
LD F2 45+ R3		
MULT F0 F2 F4		
SUBD F8 F6 F2		
DIVD F10 F0 F6		
ADDD F6 F8 F2		

Issue	Read operand	Execution complet	Write Result
1	2	3	4
5	6	7	8
6	9		
7	9	11	12
8			
13	14		

Functional unit status

Time Name

Integer
5 Mult1
Mult2
2 Add
Divide

Busy	Op	dest <i>Fi</i>	<i>S1</i> <i>Fj</i>	<i>S2</i> <i>Fk</i>	<i>FU</i> for <i>j</i> <i>Qj</i>	<i>FU</i> for <i>k</i> <i>Qk</i>	<i>Fj</i> ? <i>Rj</i>	<i>Fk</i> ? <i>Rk</i>
No								
Yes	Mult	F0	F2	F4			Yes	Yes
No								
Yes	Add	F6	F8	F2			Yes	Yes
Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

Clock

14

FU

<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
Mult1			Add		Divide			

记分牌算法举例-时钟周期15

Instruction status

Instruction	<i>j</i>	<i>k</i>
LD F6 34+ R2		
LD F2 45+ R3		
MULT F0 F2 F4		
SUBD F8 F6 F2		
DIVD F10 F0 F6		
ADDD F6 F8 F2		

Issue	Read operand	Execution complete	Write Result
1	2	3	4
5	6	7	8
6	9		
7	9	11	12
8			
13	14		

Functional unit status

Time Name

Integer
4 Mult1
Mult2
1 Add
Divide

Busy	Op	dest <i>Fi</i>	<i>S1</i> <i>Fj</i>	<i>S2</i> <i>Fk</i>	<i>FU</i> for <i>j</i> <i>Qj</i>	<i>FU</i> for <i>k</i> <i>Qk</i>	<i>Fj</i> ? <i>Rj</i>	<i>Fk</i> ? <i>Rk</i>
No								
Yes	Mult	F0	F2	F4			Yes	Yes
No								
Yes	Add	F6	F8	F2			Yes	Yes
Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

Clock

15

FU

<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
Mult1			Add		Divide			

记分牌算法举例-时钟周期16

Instruction status

Instruction	<i>j</i>	<i>k</i>
LD F6 34+ R2		
LD F2 45+ R3		
MULT F0 F2 F4		
SUBD F8 F6 F2		
DIVD F10 F0 F6		
ADDD F6 F8 F2		

Read Executic Write

Issue	operand	complet	Result
1	2	3	4
5	6	7	8
6	9		
7	9	11	12
8			
13	14	16	

Functional unit status

Time Name

Integer
3 Mult1
Mult2
0 Add
Divide

Busy Op dest S1 S2 FU for j FU for k Fj? Fk?
Fj Fk Qj Qk Rj Rk

No							
Yes	Mult	F0	F2	F4			Yes Yes
No							
Yes	Add	F6	F8	F2			Yes Yes
Yes	Div	F10	F0	F6	Mult1		No Yes

Register result status

Clock

16

FU

F0	F2	F4	F6	F8	F10	F12	...	F30
Mult1			Add		Divide			

记分牌算法举例-时钟周期17

Instruction status

Instruction	<i>j</i>	<i>k</i>
LD F6 34+ R2		
LD F2 45+ R3		
MULT F0 F2 F4		
SUBD F8 F6 F2		
DIVD F10 F0 F6		
ADDD F6 F8 F2		

Read Executic Write

Issue	operand	complet	Result
1	2	3	4
5	6	7	8
6	9		
7	9	11	12
8			
13	14	16	

ADDD can't write because of DIVD. RAW!

Functional unit status

Time Name

Integer
2 Mult1
Mult2
Add
Divide

Busy Op dest S1 S2 FU for j FU for k Fj? Fk?
Fj Fk Qj Qk Rj Rk

No							
Yes	Mult	F0	F2	F4		Yes	Yes
No							
Yes	Add	F6	F8	F2		Yes	Yes
Yes	Div	F10	F0	F6	Mult1	No	Yes

Register result status

Clock

17

FU

F0	F2	F4	F6	F8	F10	F12	...	F30
Mult1			Add		Divide			

记分牌算法举例-时钟周期18

Instruction status

Instruction	<i>j</i>	<i>k</i>
LD F6 34+ R2		
LD F2 45+ R3		
MULT F0 F2 F4		
SUBD F8 F6 F2		
DIVD F10 F0 F6		
ADDD F6 F8 F2		

Read Executic Write

Issue	operand	complet	Result
1	2	3	4
5	6	7	8
6	9		
7	9	11	12
8			
13	14	16	

Nothing Happens!!

Functional unit status

Time Name

Integer
1 Mult1
Mult2
Add
Divide

Busy	Op	dest <i>Fi</i>	<i>S1</i> <i>Fj</i>	<i>S2</i> <i>Fk</i>	<i>FU for j</i> <i>Qj</i>	<i>FU for k</i> <i>Qk</i>	<i>Fj?</i> <i>Rj</i>	<i>Fk?</i> <i>Rk</i>
No								
Yes	Mult	F0	F2	F4			Yes	Yes
No								
Yes	Add	F6	F8	F2			Yes	Yes
Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

Clock

18

FU

<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
Mult1			Add		Divide			

记分牌算法举例-时钟周期19

Instruction status

Instruction	<i>j</i>	<i>k</i>
LD F6 34+ R2		
LD F2 45+ R3		
MULT F0 F2 F4		
SUBD F8 F6 F2		
DIVD F10 F0 F6		
ADDD F6 F8 F2		

Read Executic Write

Issue	operand	complet	Result
1	2	3	4
5	6	7	8
6	9	19	
7	9	11	12
8			
13	14	16	

MULT completes execution.

Functional unit status

Time Name

Integer
0 Mult1
Mult2
Add
Divide

Busy	Op	dest <i>Fi</i>	<i>S1</i> <i>Fj</i>	<i>S2</i> <i>Fk</i>	<i>FU for j</i> <i>Qj</i>	<i>FU for k</i> <i>Qk</i>	<i>Fj?</i> <i>Rj</i>	<i>Fk?</i> <i>Rk</i>
No								
Yes	Mult	F0	F2	F4			Yes	Yes
No								
Yes	Add	F6	F8	F2			Yes	Yes
Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

Clock

19

FU

<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
Mult1			Add		Divide			

记分牌算法举例-时钟周期20

Instruction status

Instruction	<i>j</i>	<i>k</i>
LD F6 34+ R2		
LD F2 45+ R3		
MULT F0 F2 F4		
SUBD F8 F6 F2		
DIVD F10 F0 F6		
ADDD F6 F8 F2		

Read Executic Write

Issue	operand	complet	Result
1	2	3	4
5	6	7	8
6	9	19	20
7	9	11	12
8			
13	14	16	

MULT writes.

Functional unit status

Time Name

Integer
Mult1
Mult2
Add
Divide

Busy	Op	dest <i>Fi</i>	<i>S1</i> <i>Fj</i>	<i>S2</i> <i>Fk</i>	<i>FU</i> for <i>j</i> <i>Qj</i>	<i>FU</i> for <i>k</i> <i>Qk</i>	<i>Fj</i> ? <i>Rj</i>	<i>Fk</i> ? <i>Rk</i>
No								
No								
No								
Yes	Add	F6	F8	F2			Yes	Yes
Yes	Div	F10	F0	F6			Yes	Yes

Register result status

Clock

20

FU

<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
			Add		Divide			

记分牌算法举例-时钟周期21

Instruction status

Instruction	<i>j</i>	<i>k</i>
LD F6 34+ R2		
LD F2 45+ R3		
MULT F0 F2 F4		
SUBD F8 F6 F2		
DIVD F10 F0 F6		
ADDD F6 F8 F2		

Read Executic Write

Issue	operand	complet	Result
1	2	3	4
5	6	7	8
6	9	19	20
7	9	11	12
8	21		
13	14	16	

DIVD loads operands

Functional unit status

Time Name

Integer
Mult1
Mult2
Add
Divide

Busy	Op	dest <i>Fi</i>	<i>S1</i> <i>Fj</i>	<i>S2</i> <i>Fk</i>	<i>FU for j</i> <i>Qj</i>	<i>FU for k</i> <i>Qk</i>	<i>Fj?</i> <i>Rj</i>	<i>Fk?</i> <i>Rk</i>
No								
No								
No								
Yes	Add	F6	F8	F2			Yes	Yes
Yes	Div	F10	F0	F6			Yes	Yes

Register result status

Clock

21

FU

<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
			Add		Divide			

记分牌算法举例-时钟周期22

Instruction status

Instruction	<i>j</i>	<i>k</i>
LD F6 34+ R2		
LD F2 45+ R3		
MULT F0 F2 F4		
SUBD F8 F6 F2		
DIVD F10 F0 F6		
ADDD F6 F8 F2		

Read Executic Write

Issue	operand	complet	Result
1	2	3	4
5	6	7	8
6	9	19	20
7	9	11	12
8	21		
13	14	16	22

Now ADDD can write since WAR removed.

Functional unit status

Time Name

Integer
Mult1
Mult2
Add
40 Divide

Busy	Op	dest <i>Fi</i>	S1 <i>Fj</i>	S2 <i>Fk</i>	FU for <i>j</i> <i>Qj</i>	FU for <i>k</i> <i>Qk</i>	<i>Fj?</i> <i>Rj</i>	<i>Fk?</i> <i>Rk</i>
No								
No								
No								
No								
Yes	Div	F10	F0	F6			Yes	Yes

Register result status

Clock

22

FU

F0	F2	F4	F6	F8	F10	F12	...	F30
					Divide			



记分牌算法举例-时钟周期XX

快速跳过一些时钟周期

记分牌算法举例-时钟周期61

Instruction status

Instruction	<i>j</i>	<i>k</i>
LD F6 34+ R2		
LD F2 45+ R3		
MULT F0 F2 F4		
SUBD F8 F6 F2		
DIVD F10 F0 F6		
ADDD F6 F8 F2		

Issue	Read operand	Execution complete	Write Result
1	2	3	4
5	6	7	8
6	9	19	20
7	9	11	12
8	21	61	
13	14	16	22

DIVD completes execution

Functional unit status

Time Name

Integer
Mult1
Mult2
Add
0 Divide

Busy	Op	dest <i>Fi</i>	S1 <i>Fj</i>	S2 <i>Fk</i>	FU for <i>j</i> <i>Qj</i>	FU for <i>k</i> <i>Qk</i>	<i>Fj?</i> <i>Rj</i>	<i>Fk?</i> <i>Rk</i>
No								
No								
No								
No								
Yes	Div	F10	F0	F6			Yes	Yes

Register result status

Clock

61

FU

F0	F2	F4	F6	F8	F10	F12	...	F30
					Divide			

记分牌算法举例-时钟周期62

Instruction status

Instruction	<i>j</i>	<i>k</i>
LD F6 34+ R2		
LD F2 45+ R3		
MULT F0 F2 F4		
SUBD F8 F6 F2		
DIVD F10 F0 F6		
ADDD F6 F8 F2		

Issue	Read operand	Execution complete	Write Result
1	2	3	4
5	6	7	8
6	9	19	20
7	9	11	12
8	21	61	62
13	14	16	22

DONE!!

Functional unit status

Time Name

Integer
Mult1
Mult2
Add
0 Divide

Busy	Op	dest <i>Fi</i>	S1 <i>Fj</i>	S2 <i>Fk</i>	FU for <i>j</i> <i>Qj</i>	FU for <i>k</i> <i>Qk</i>	<i>Fj?</i> <i>Rj</i>	<i>Fk?</i> <i>Rk</i>
No								
No								
No								
No								
No								

Register result status

Clock

62

FU

F0 F2 F4 F6 F8 F10 F12 ... F30

记分牌流水线-冲突消除总结

消除冲突类型	Instruction status	Wait until	Bookkeeping
结构冲突和WAW冲突	Issue	Not Busy(FU) && not Result(D)	Busy(FU) ← yes; Op(FU) ← op; Fi(FU) ← 'D'; Fj(FU) ← 'S1'; Fk(FU) ← 'S2'; Qj ← Result('S1'); Qk ← Result('S2'); Rj ← not Qj; Rk ← not Qk; Result('D') ← FU;
RAW冲突	Read operands	Rj == Yes && Rk == Yes	Rj ← No; Rk ← No
	Execution complete	Functional unit done	
WAR冲突	Write result	$\forall f($ $(Fj(f) \neq Fi(FU) \parallel Rj(f) == No)$ $\&\&$ $(Fk(f) \neq Fi(FU) \parallel Rk(f) == No)$ $)$	$\forall f(\text{if } Qj(f) = FU \text{ then } Rj(f) \leftarrow \text{Yes});$ $\forall f(\text{if } Qk(f) = FU \text{ then } Rk(f) \leftarrow \text{Yes});$ Result(Fi(FU)) ← 0; Busy(FU) ← No

CDC 6600 scoreboard的主要缺陷

- ◆ 没有定向数据通路
- ◆ 指令窗口较小，仅局限于基本块内的调度
- ◆ 功能部件数较少，容易产生**结构相关**，特别是其Load/store操作也是用IU部件完成的
- ◆ 结构冲突时不能发射
- ◆ **WAR相关**是通过等待解决的
- ◆ **WAW相关**时，不会进入IS阶段

动态调度方案二：Tomasulo算法

- ◆ 该算法首次在 IBM 360/91上使用
 - 比CDC6600晚三年
 - 目标：在没有专用编译器的情况下，提高系统性能
- ◆ Tomasulo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。
- ◆ IBM 360 & CDC 6600 ISA的差别
 - IBM360只有 2位寄存器描述符 vs. CDC 6600寄存器描述符3位
 - IBM360 4个FP 寄存器 vs. CDC 6600 8个
 - IBM 360 有memory-register 操作
- ◆ Alpha 21264, HP 8000, MIPS 10000, Pentium II, PowerPC 604, ...

Tomasulo vs. 记分牌算法

- ◆ 控制和缓存**分布**在各部件中 vs. 控制和缓存**集中**在记分牌
 - FU 缓存称为“保留站RS”；保存待用操作数
- ◆ **指令中的寄存器**在RS中用**寄存器值**或**指向RS的指针**代替（称为寄存器换名，register renaming）
 - 避免 WAR, WAW冲突
 - **RS多于寄存器**，因此可以做更多编译器无法做的优化
- ◆ 传给FU的数据从RS来而不是从寄存器来，FU的计算结果通过CDB以**广播方式**发向所有功能部件
- ◆ Load和Store部件也看作带有RS的功能部件
- ◆ 可以**跨越分支**，允许FP操作队列中FP操作不仅仅局限于基本块

寄存器换名

◆ 考虑如下代码

反相关 (F8)
导致WAR冲突

DIV.D F0 , F2 , F4

ADD.D F6 , F0 , F8

S.D F6 , 0 (R1)

SUB.D F8 , F10 , F14

MUL.D F6 , F10 , F8

输出相关 (F6)
导致WAW冲突

◆ 消除名相关

- 引入两个临时寄存器S 和 T
- 把这段代码改写为:

DIV.D F0 , F2 , F4

ADD.D S , F0 , F8

S.D S , 0 (R1)

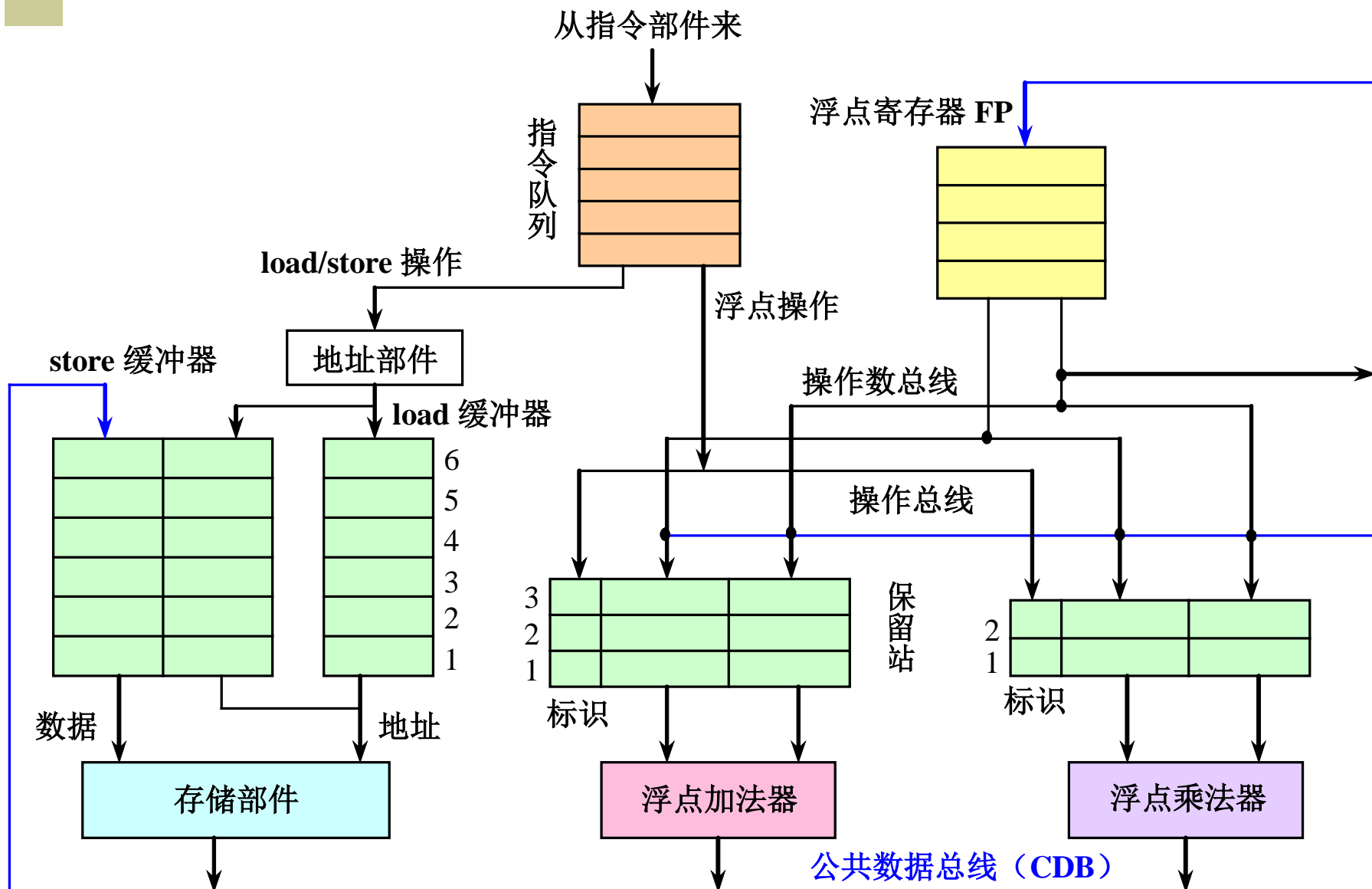
SUB.D T , F10 , F14

MUL.D F6 , F10 , T

} 两个F6都换名为S

两个F8都换名为T {

MIPS处理器浮点部件的基本结构



Tomasulo算法基本硬件结构

◆ 保留站 (Reservation Station)

- 每个保留站中保存一条已经流出并等待到本功能部件执行的指令（相关信息）。
- 包括：操作码、操作数以及用于检测 and 解决冲突的信息。
 - 在一条指令流出到保留站的时候，如果该指令的源操作数已经在寄存器中就绪，则将之取到该保留站中。
 - 如果操作数还没有计算出来，则在该保留站中记录将产生这个操作数的保留站的标识。
- 浮点加法器有3个保留站：ADD1, ADD2, ADD3
- 浮点乘法器有2个保留站：MULT1, MULT2
 - 每个保留站都有一个标识字段，唯一地标识了该保留站

Tomasulo算法基本硬件结构

- ◆ 公共数据总线CDB：一条重要的数据通路
 - 所有功能部件的计算结果都是送到CDB上，由它把这些结果直接送到（播送到）各个需要该结果的地方。
 - 在具有多个执行部件且采用多流出（即每个时钟周期流出多条指令）的流水线中，需要采用多条CDB。
- ◆ load缓冲器和store缓冲器
 - 存放读/写存储器的数据或地址
 - load缓冲器的作用有3个：
 - 存放用于计算有效地址的分量
 - 记录正在进行的load访存，等待存储器的响应；
 - 保存已经完成了的load的结果（即从存储器取来的数据），等待CDB传输。
 - store缓冲器的作用有3个
 - 存放用于计算有效地址的分量
 - 保存正在进行的store访存的目标地址，该store正在等待存储数据的到达；
 - 保存该store的地址和数据，直到存储部件接收。

Tomasulo算法基本硬件结构

◆ 浮点寄存器FP

- 共有16个浮点寄存器：F0, F2, F4, ..., F30。
- 它们通过一对总线连接到功能部件，并通过CDB连接到store缓冲器。

◆ 指令队列

- 指令部件送来的指令放入指令队列
- 指令队列中的指令按**先进先出**的顺序流出

◆ 运算部件

- 浮点加法器完成加法和减法操作
- 浮点乘法器完成乘法和除法操作

Tomasulo算法的特点

- ◆ 在Tomasulo算法中，**寄存器换名**是通过保留站和流出逻辑来共同完成的。
 - 当指令流出时，如果其操作数还没有计算出来，则将该指令中相应的寄存器号换名为将产生该操作数的保留站标识
 - 指令流出到保留站后，其操作数寄存器号或者换成了数据本身（如果该数据已经就绪），或者换成了保留站的标识，不再与寄存器有关系。
- ◆ Tomasulo算法具有以下两个特点：
 - **冲突检测和指令执行控制是分布的**
 - 每个功能部件的保留站中的信息决定了什么时候指令可以在该功能部件开始执行
 - 计算结果通过CDB直接从产生它的保留站传送到所有需要它的功能部件，而不用经过寄存器。

Tomasulo算法执行步骤

- ◆ 使用Tomasulo算法的流水线需3段：
- ◆ **1、流出**：从指令队列的头部取一条指令
 - 如果该指令的操作所要求的**保留站有空闲**的，就把该指令送到该保留站（设为r）
 - 如果其操作数在寄存器中已经就绪，就将这些操作数送入保留站r
 - 如果其操作数还没有就绪，就把将产生该操作数的保留站的标识送入保留站r。
 - 一旦被记录的保留站完成计算，它将直接把数据送给保留站r。
 - 寄存器换名和对操作数进行缓冲，**消除WAR冲突**
 - 完成对目标寄存器的预约工作
 - **消除了WAW冲突**
 - 如果没有空闲的保留站，指令就不能流出
 - 发生了**结构冲突**

Tomasulo算法执行步骤

◆ 2、执行

- 当两个操作数都就绪后，本保留站就用相应的功能部件开始执行指令规定的操作
- load和store指令的执行需要两个步骤：
 - 计算有效地址（要等到基地址寄存器就绪）
 - 把有效地址放入load或store缓冲器

◆ 3、写结果

- 功能部件计算完毕后，就将计算结果放到CDB上，所有等待该计算结果的寄存器和保留站（包括store缓冲器）都同时从CDB上获得所需要的数据

Tomasulo算法保留站字段

- ◆ **Op**: 要对源操作数进行的操作。
- ◆ **Qj, Qk**: 将产生源操作数的保留站号。
 - 等于0表示操作数已经就绪且在Vj或Vk中，或者不需要操作数
- ◆ **Vj, Vk**: 源操作数的值。
 - 对于每一个操作数来说，V或Q字段只有一个有效
- ◆ **Busy**: 为“yes”表示本保留站或缓冲单元“忙”。
- ◆ **A**: 仅load和store缓冲器有该字段。开始时存放指令中的立即数字段，地址计算后存放有效地址。
- ◆ **Qi**: 寄存器状态表。
 - 每个寄存器在该表中有对应的一项，用于存放将把结果写入该寄存器的保留站的站号
 - 为0表示当前没有正在执行的指令要写入该寄存器，也即该寄存器中的内容就绪

Tomasulo算法的具体实现

◆ 各符号的意义

- **r**: 分配给当前指令的保留站或者缓冲器单元编号;
- **rd**: 目标寄存器编号;
- **rs**、**rt**: 操作数寄存器编号;
- **imm**: 符号扩展后的立即数;
- **RS**: 保留站;
- **result**: 浮点部件或load缓冲器返回的结果;
- **Qi**: 寄存器状态表;
- **Regs[]**: 寄存器组;
- 与rs对应的保留站字段: **Vj**, **Qj**
- 与rt对应的保留站字段: **Vk**, **Qk**

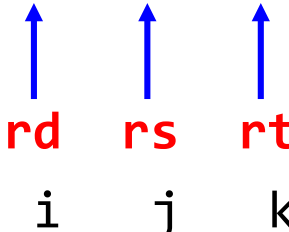
Tomasulo算法的具体实现

◆ 各符号的意义

- Q_i 、 Q_j 、 Q_k 的内容或者为 0，或者是一个大于0的整数
- Q_i 为0表示相应寄存器中的数据就绪。
- Q_j 、 Q_k 为0表示保留站或缓冲器单元中的 V_j 或 V_k 字段中的数据就绪
- 当它们为正整数时，表示相应的寄存器、保留站或缓冲器单元正在等待结果。

◆ 符号说明：（举例）

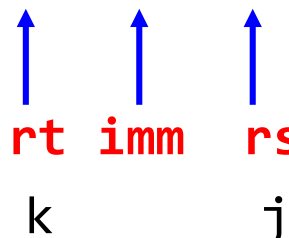
MUL.D F4, F0, F2



rd rs rt
i *j* *k*

S.D F3, 40(R4)

L.D F2, 45(R3)



rt imm rs
k *j*

Tomasulo算法—1、指令流出

MUL.D F4, F0, F2

↑ ↑ ↑

rd rs rt

i j k

◆ 浮点运算指令

- 进入条件：有空闲保留站（r），操作和状态表内容修改：

// (1) 检测第一操作数是否就绪

if (Qi[rs] == 0) // 第一操作数**就绪**

{ //把寄存器 rs中的操作数取到当前保留站的Vj

RS[r].Vj ← Regs[rs];

RS[r].Qj ← 0; // 置Qj为0,

} //表示当前保留站的Vj中的操作数就绪。

else // 第一操作数**没有就绪**

{ // 进行寄存器换名，即把将产生该操作数的保留站的编号

RS[r].Qj ← Qi[rs]; // 放入当前保留站的Qj

}

Tomasulo算法—1、指令流出

// (2) 检测第二操作数是否就绪

if ($Q_i[rt] = 0$) // 第二操作数**就绪**。

{ //把寄存器rt中的操作数取到当前保留站的Vk

$RS[r].Vk \leftarrow Regs[rt];$

// 置Qk为0，表示当前保留站的Vk中的操作数就绪。

$RS[r].Qk \leftarrow 0;$

}

else //第二操作数**没有就绪**

{ //进行寄存器换名，把将产生该操作数的保留站编号放入当前保留站Qk

$RS[r].Qk \leftarrow Q_i[rt];$

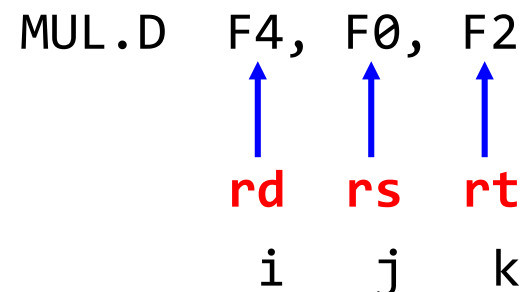
}

$RS[r].Busy \leftarrow yes;$ //置当前保留站为“忙”

$RS[r].Op \leftarrow Op;$ //设置操作码

$Q_i[rd] \leftarrow r;$ // 把当前保留站的编号r放入rd 所对应

// 的寄存器状态表项，以便rd 将来接收结果



Tomasulo算法—1、指令流出

◆ load和store指令

- 进入条件：缓冲器有空闲单元（r）操作和状态表内容修改

// 检测第一操作数是否就绪

L.D F2, 45(R3)

if (Qi[rs] = 0) // 第一操作数**就绪**

{ // 把寄存器rs中的操作数取到当前缓冲器单元的Vj

RS[r].Vj ← Regs[rs];

RS[r].Qj ← 0; // 置Qj为0，表示当前缓冲器单元的Vj

} // 中的操作数就绪

else // 第一操作数**没有就绪**

{ // 进行寄存器换名，即把将产生该操作数的保留站的编号

RS[r].Qj ← Qi[rs]; // 存入当前缓冲器单元的Qj

}

RS[r].Busy ← yes; // 置当前缓冲器单元为“忙”

RS[r].A ← Imm; // 把符号位扩展后的偏移量放入

// 当前缓冲器单元的A

↑ ↑ ↑
rt **imm** **rs**
k j

Tomasulo算法—1、指令流出

◆ 对于load指令:

$Q_i[rt] \leftarrow r;$ // 把当前缓冲器单元的编号 r 放入
 // load指令的目标寄存器 rt 所对应的寄存器状态表项,
 // 以便 rt 将来接收所取的数据。

L.D F2, 45(R3)
 ↑ ↑ ↑
 rt imm rs
 k j

◆ 对于Store指令:

// 检测要存储的数据是否就绪
 if ($Q_i[rt] = 0$) // 该数据**就绪**
 { // 把它从寄存器 rt 取到store缓冲器单元的 V_k
 $RS[r].V_k \leftarrow Regs[rt];$
 $RS[r].Q_k \leftarrow 0;$ // 置 Q_k 为0,
 //表示当前缓冲器单元的 V_k 中的数据就绪
 }
 else // 该数据**没有就绪**
 { // 进行寄存器换名, 即把将产生该数据的保留站的编号
 $RS[r].Q_k \leftarrow Q_i[rt];$ // 放入当前缓冲器单元的 Q_k
 }

S.D F3, 40(R4)
 ↑ ↑ ↑
 rt imm rs
 k j

Tomasulo算法—2、执行

◆ 浮点操作指令

- 进入条件： $(RS[r].Qj = 0)$ **且** $(RS[r].Qk = 0)$;
// 两个源操作数就绪
- 操作和状态表内容修改：进行计算，产生结果。

◆ load/store指令，关键是计算有效地址

- 进入条件： $(RS[r].Qj = 0)$ // r为保留站编号
且 r成为load/store 缓冲队列的头部
- 操作和状态表内容修改：
 $RS[r].A \leftarrow RS[r].Vj + RS[r].A;$ // 计算有效地址
- 对于load指令**，在完成有效地址计算后，还要进行：
从Mem[RS[r].A]读取数据； //从存储器中读取数据

Tomasulo算法—3、写结果

◆ 浮点运算指令和load指令

- 进入条件：保留站r执行结束，且CDB就绪。
- 操作和状态表内容修改：

// (1) 对于任何一个正在等该结果的浮点寄存器x

$\forall x \text{ (if (Qi[x] == r))}$

{

 Regs[x] \leftarrow result; // 向该寄存器写入结果

 Qi[x] \leftarrow 0; // 把该寄存器的状态置为数据就绪

}

// (2) 对于任何一个正在等该结果作为第一操作数的保留站x

$\forall x \text{ (if(RS[x].Qj == r))}$

{

 RS[x].Vj \leftarrow result; // 向该保留站的Vj写入结果

 RS[x].Qj \leftarrow 0; // 置Qj为0,

}

 // 表示该保留站的Vj中的操作数就绪

Tomasulo算法—3、写结果

// (3) 对于任何一个正在等该结果作为**第二操作数的保留站x**

$\forall x$ (if (RS[x].Qk == r)

{

 RS[x].Vk \leftarrow result; // 向该保留站的Vk写入结果

 RS[x].Qk \leftarrow 0; // 置Qk为0,

} // 表示该保留站的Vk中的操作数就绪

RS[r].Busy \leftarrow no; // 释放当前保留站, 将之置为空闲状态

◆ store指令

- 进入条件: 保留站r执行结束, **且** RS[r].Qk == 0

// 要存储的数据已经**就绪**

- 操作和状态表内容修改:

// 数据写入存储器, 地址由store缓冲器单元的A字段给出。

Mem[RS[r].A] \leftarrow RS[r].Vk

RS[r].Busy \leftarrow no; // 释放当前缓冲器单元, 将之置为空闲状态

Tomasulo算法举例

Instruction status:

				Exec	Write		
Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Comp</i>	<i>Result</i>	Busy	Address
LD	F6	34+	R2			Load1	No
LD	F2	45+	R3			Load2	No
MULTD	F0	F2	F4			Load3	No
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Reservation Stations:

on Stations:

				<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
0	<i>FU</i>								

Tomasulo算法举例-时钟周期1

Instruction status:

				Exec	Write		
Instruction	<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Address
LD	F6	34+	R2	1		Load1	Yes 34+R2
LD	F2	45+	R3			Load2	No
MULTD	F0	F2	F4			Load3	No
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Reservation Stations:

on Stations:

				$S1$	$S2$	RS	RS
$Time$	$Name$	$Busy$	Op	V_j	V_k	Q_j	Q_k
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
1	FU Load1								

Tomasulo算法举例-时钟周期2

Instruction status:

				Exec	Write		
Instruction	<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Address
LD	F6	34+	R2	1		Load1	Yes 34+R2
LD	F2	45+	R3	2		Load2	Yes 45+R3
MULTD	F0	F2	F4			Load3	No
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Reservation Stations:

on Stations:

				<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
2	<i>FU</i>	Load2		Load1					

Tomasulo算法举例-时钟周期3

Instruction status:

				Exec Write			
Instruction	<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Address
LD	F6	34+	R2	1	3	Load1	Yes 34+R2
LD	F2	45+	R3	2		Load2	Yes 45+R3
MULTD	F0	F2	F4	3		Load3	No
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Reservation Stations:

on Stations:

				<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
3	FU Mult1 Load2 Load1								

Tomasulo算法举例-时钟周期4

Instruction status:

				Exec Write			
Instruction	<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Address
LD	F6	34+	R2	1	3	4	Load1
LD	F2	45+	R3	2	4		Load2
MULTD	F0	F2	F4	3			Load3
SUBD	F8	F6	F2	4			
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Reservation Stations:

				<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
Time	Name	Busy	Op	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
Add1	Yes	SUBD	M(A1)				Load2
Add2	No						
Add3	No						
Mult1	Yes	MULTD			R(F4)	Load2	
Mult2	No						

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
4	FU	Mult1	Load2		M(A1)	Add1			

Tomasulo算法举例-时钟周期5

Instruction status:

				Exec	Write		
Instruction	<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Address
LD	F6	34+	R2	1	3	4	Load1
LD	F2	45+	R3	2	4	5	Load2
MULTD	F0	F2	F4	3			Load3
SUBD	F8	F6	F2	4			
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2				

Reservation Stations:

on Stations:

				<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
2	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	No					
	Add3	No					
10	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
5	FU	Mult1	M(A2)		M(A1)	Add1	Mult2		

Tomasulo算法举例-时钟周期6

Instruction status:

				Exec	Write		
Instruction	<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Address
LD	F6	34+	R2	1	3	4	Load1
LD	F2	45+	R3	2	4	5	Load2
MULTD	F0	F2	F4	3			Load3
SUBD	F8	F6	F2	4			
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	6			

Reservation Stations:

on Stations:

				<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
1	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
9	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
6	FU	Mult1	M(A2)		Add2	Add1	Mult2		

Tomasulo算法举例-时钟周期7

Instruction status:

				Exec	Write		
Instruction	<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Address
LD	F6	34+	R2	1	3	4	Load1
LD	F2	45+	R3	2	4	5	Load2
MULTD	F0	F2	F4	3			Load3
SUBD	F8	F6	F2	4	7		
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	6			

Reservation Stations:

on Stations:

				<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
0	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
8	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
7	FU								
	Mult1	M(A2)		Add2	Add1	Mult2			

Tomasulo算法举例-时钟周期8

Instruction status:

				Exec	Write		
Instruction	<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Address
LD	F6	34+	R2	1	3	4	Load1
LD	F2	45+	R3	2	4	5	Load2
MULTD	F0	F2	F4	3			Load3
SUBD	F8	F6	F2	4	7	8	
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	6			

Reservation Stations:

on Stations:				<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
	Add1	No					
2	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
7	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
8	<i>FU</i>	Mult1	M(A2)		Add2	(M-M)	Mult2		

Tomasulo算法举例-时钟周期9

Instruction status:

				Exec	Write		
Instruction	<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Address
LD	F6	34+	R2	1	3	4	Load1
LD	F2	45+	R3	2	4	5	Load2
MULTD	F0	F2	F4	3			Load3
SUBD	F8	F6	F2	4	7	8	
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	6			

Reservation Stations:

on Stations:

				<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
	Add1	No					
1	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
6	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
9	<i>FU</i>	Mult1	M(A2)		Add2	(M-M)	Mult2		

Tomasulo算法举例-时钟周期10

Instruction status:

				Exec	Write		
Instruction	<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Address
LD	F6	34+	R2	1	3	4	Load1
LD	F2	45+	R3	2	4	5	Load2
MULTD	F0	F2	F4	3			Load3
SUBD	F8	F6	F2	4	7	8	
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	6	10		

Reservation Stations:

on Stations:

				<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
	Add1	No					
0	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
5	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
10	<i>FU</i>	Mult1	M(A2)		Add2	(M-M)	Mult2		

Tomasulo算法举例-时钟周期11

Instruction status:

				Exec	Write		
Instruction	<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Address
LD	F6	34+	R2	1	3	4	Load1
LD	F2	45+	R3	2	4	5	Load2
MULTD	F0	F2	F4	3			Load3
SUBD	F8	F6	F2	4	7	8	
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	6	10	11	

Reservation Stations:

on Stations:

				<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
	Add1	No					
	Add2	No					
	Add3	No					
4	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
11	FU	Mult1	M(A2)		(M-M+M	(M-M)	Mult2		

Tomasulo算法举例-时钟周期12

Instruction status:

				Exec	Write		
Instruction	<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Address
LD	F6	34+	R2	1	3	4	Load1
LD	F2	45+	R3	2	4	5	Load2
MULTD	F0	F2	F4	3			Load3
SUBD	F8	F6	F2	4	7	8	
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	6	10	11	

Reservation Stations:

				<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
Time	Name	Busy	Op	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
	Add1	No					
	Add2	No					
	Add3	No					
3	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD	M(A1)	Mult1		

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
12	FU	Mult1	M(A2)		(M-M+M	(M-M)	Mult2		

Tomasulo算法举例-时钟周期13

Instruction status:

				Exec	Write		
Instruction	<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Address
LD	F6	34+	R2	1	3	4	Load1
LD	F2	45+	R3	2	4	5	Load2
MULTD	F0	F2	F4	3			Load3
SUBD	F8	F6	F2	4	7	8	
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	6	10	11	

Reservation Stations:

on Stations:

				<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
	Add1	No					
	Add2	No					
	Add3	No					
2	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
13	FU	Mult1	M(A2)		(M-M+M	(M-M)	Mult2		

Tomasulo算法举例-时钟周期14

Instruction status:

				Exec	Write		
Instruction	<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Address
LD	F6	34+	R2	1	3	4	Load1
LD	F2	45+	R3	2	4	5	Load2
MULTD	F0	F2	F4	3			Load3
SUBD	F8	F6	F2	4	7	8	
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	6	10	11	

Reservation Stations:

on Stations:

				<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
	Add1	No					
	Add2	No					
	Add3	No					
1	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
14	FU	Mult1	M(A2)		(M-M+M	(M-M)	Mult2		

Tomasulo算法举例-时钟周期15

Instruction status:

				Exec	Write		
Instruction	<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Address
LD	F6	34+	R2	1	3	4	Load1
LD	F2	45+	R3	2	4	5	Load2
MULTD	F0	F2	F4	3	15		Load3
SUBD	F8	F6	F2	4	7	8	
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	6	10	11	

Reservation Stations:

on Stations:

				<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
	Add1	No					
	Add2	No					
	Add3	No					
0	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
15	FU	Mult1	M(A2)		(M-M+M	(M-M)	Mult2		

Tomasulo算法举例-时钟周期16

Instruction status:

				Exec	Write		
Instruction	<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Address
LD	F6	34+	R2	1	3	4	Load1
LD	F2	45+	R3	2	4	5	Load2
MULTD	F0	F2	F4	3	15	16	Load3
SUBD	F8	F6	F2	4	7	8	
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	6	10	11	

Reservation Stations:

on Stations:

				<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
40	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
16	<i>FU</i>	M*F4	M(A2)		(M-M+M	(M-M)	Mult2		



Tomasulo算法举例-时钟周期XXX

快速跳过一些时钟周期

Tomasulo算法举例-时钟周期55

Instruction status:

				Exec	Write		
Instruction	<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Address
LD	F6	34+	R2	1	3	4	Load1
LD	F2	45+	R3	2	4	5	Load2
MULTD	F0	F2	F4	3	15	16	Load3
SUBD	F8	F6	F2	4	7	8	
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	6	10	11	

Reservation Stations:

				<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
Time	Name	Busy	Op	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
1	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
55	FU	M*F4	M(A2)		(M-M+M	(M-M)	Mult2		

Tomasulo算法举例-时钟周期56

Instruction status:

				Exec	Write		
Instruction	<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Address
LD	F6	34+	R2	1	3	4	Load1
LD	F2	45+	R3	2	4	5	Load2
MULTD	F0	F2	F4	3	15	16	Load3
SUBD	F8	F6	F2	4	7	8	
DIVD	F10	F0	F6	5	56		
ADDD	F6	F8	F2	6	10	11	

Reservation Stations:

				<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
Time	Name	Busy	Op	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
0	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
56	FU	M*F4	M(A2)		(M-M+M	(M-M)	Mult2		

Tomasulo算法举例-时钟周期57

Instruction status:

				Exec	Write		
Instruction	<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Address
LD	F6	34+	R2	1	3	4	Load1
LD	F2	45+	R3	2	4	5	Load2
MULTD	F0	F2	F4	3	15	16	Load3
SUBD	F8	F6	F2	4	7	8	
DIVD	F10	F0	F6	5	56	57	
ADDD	F6	F8	F2	6	10	11	

Reservation Stations:

on Stations:

				<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
56	FU	M*F4	M(A2)		(M-M+N	(M-M)	Result		

Tomasulo算法 vs 记分牌算法

Instruction status:

				Read	Exec	Write	
				Issue	Oper	Comp	Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8	21	61	62
ADDD	F6	F8	F2	13	14	16	22

记分牌算法

			Exec	Write	
			Issue	Comp	Result
			1	3	4
			2	4	5
			3	15	16
			4	7	8
			5	56	57
			6	10	11

Tomasulo算法

- ◆ 为什么记分牌算法/6600所需时间较长？
 - 结构冲突
 - 没有定向技术

Tomasulo算法 vs 记分牌算法

	IBM 360/91	CDC 6600
功能部件工作方式	流水化的功能部件	多个功能部件
功能部件数量	6 load, 3 store, 3 add, 2 multi	1 load/store, 1 add, 2 multi, 1 divide
指令窗口大小	~ 14 指令	~ 5 指令
结构冲突时	不发射	不发射
WAR时	寄存器重命名	stall
WAW时	寄存器重命名	停止发射
写寄存器方式	CDB广播	
控制方式	保留站, 分布式	集中式

4.3 动态分支预测技术

- ◆ 控制相关对流水线的吞吐率和效率影响相对于数据相关要大得多
 - 条件指令在一般程序中所占的比例相当大
- ◆ 处理条件转移控制相关的关键问题：
 - 要确保流水线能够正常工作
 - 减少因断流引起的吞吐率和效率的下降
- ◆ 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
 - 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，就需要预测分支的结果；
 - Amdahl定律告诉我们，机器的CPI越小，控制停顿的相对影响就更大。

4.3 动态分支预测技术

- ◆ **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。
- ◆ 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

4.3 动态分支预测技术

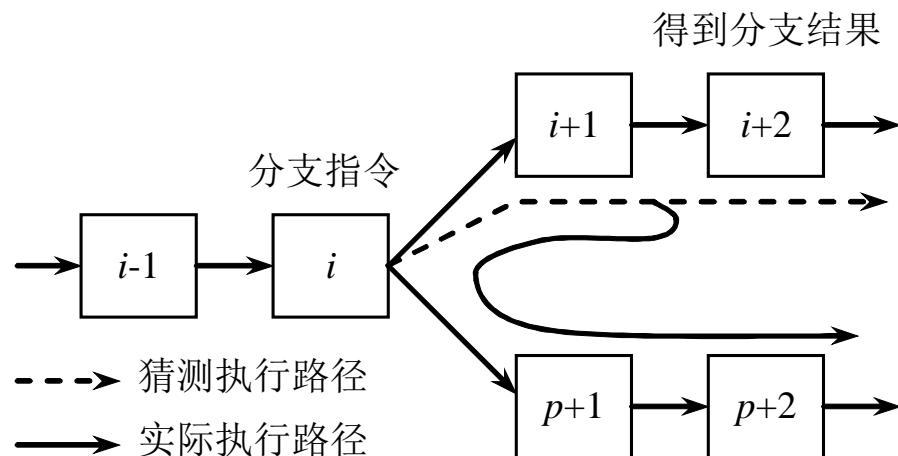
◆ 采用动态分支预测技术的**目的**

- 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

◆ 需要解决的**关键问题**

- 如何记录分支的历史信息；
- 如何根据这些信息来预测分支的去向（甚至取到指令）。

◆ 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。



4.3 动态分支预测技术

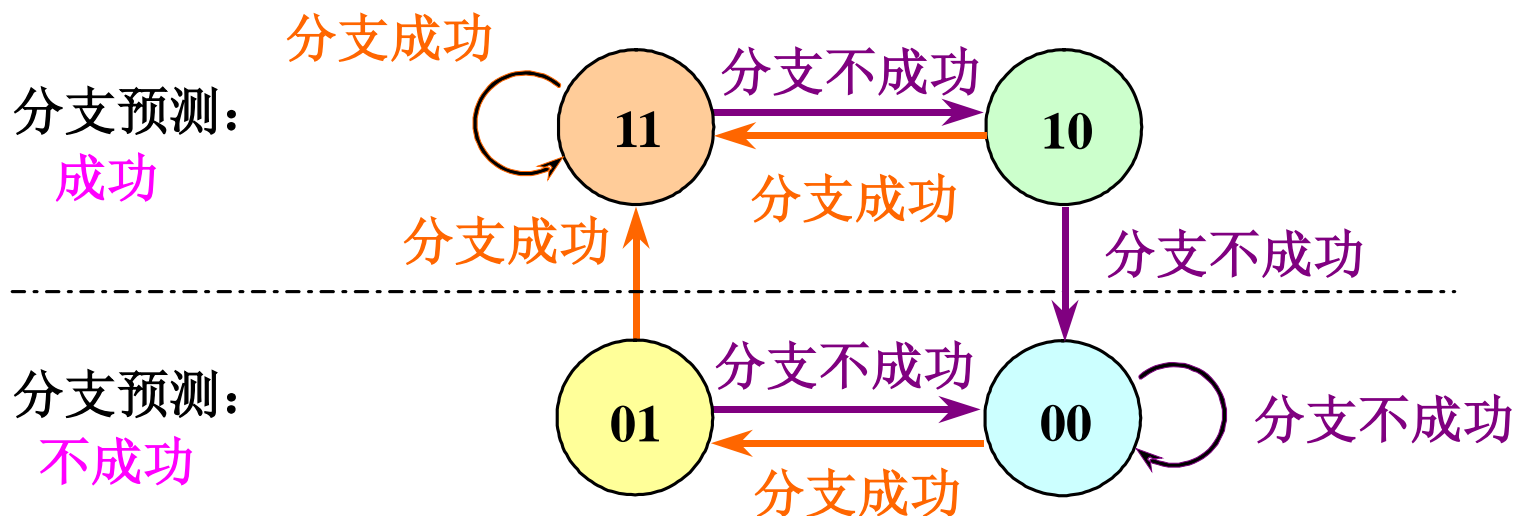
- ◆ 当分支方向预测错误时
 - 流水线中有多个功能段要浪费
 - 可能造成程序执行结果发生错误
 - 因此当程序沿着错误方向运行后，作废这些程序时，一定不能破坏通用寄存器和主存储器的内容。
- ◆ 假设对于一条有K段的流水线，由于条件分支的影响，在最坏情况下，每次条件转移将造成 $k-1$ 个时钟周期的断流。假设条件分支在一般程序中所占的比例为 p ，条件成功的概率为 q 。试分析分支对流水线的影响。
 - 加速比为理想情况的： $1/[1+p(1-q)(k-1)]$
 - 结论：条件转移指令对流水线的影响很大，必须采取相关措施来减少这种影响。

4.3.1 采用分支历史表 BHT

- ◆ **分支历史表BHT** (Branch History Table) 或分支预测缓冲器 (Branch Prediction Buffer)
 - 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 (成功或不成功)，并据此进行预测。
- ◆ **只有1个预测位的分支预测缓冲**：只记录分支指令最近一次的历史，BHT中只需要1位二进制位。
- ◆ **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。

4.3.1 采用分支历史表 BHT

- ◆ 两位分支预测的状态转换如下所示：



- ◆ 两位分支预测中的操作有两个步骤：

- **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**。

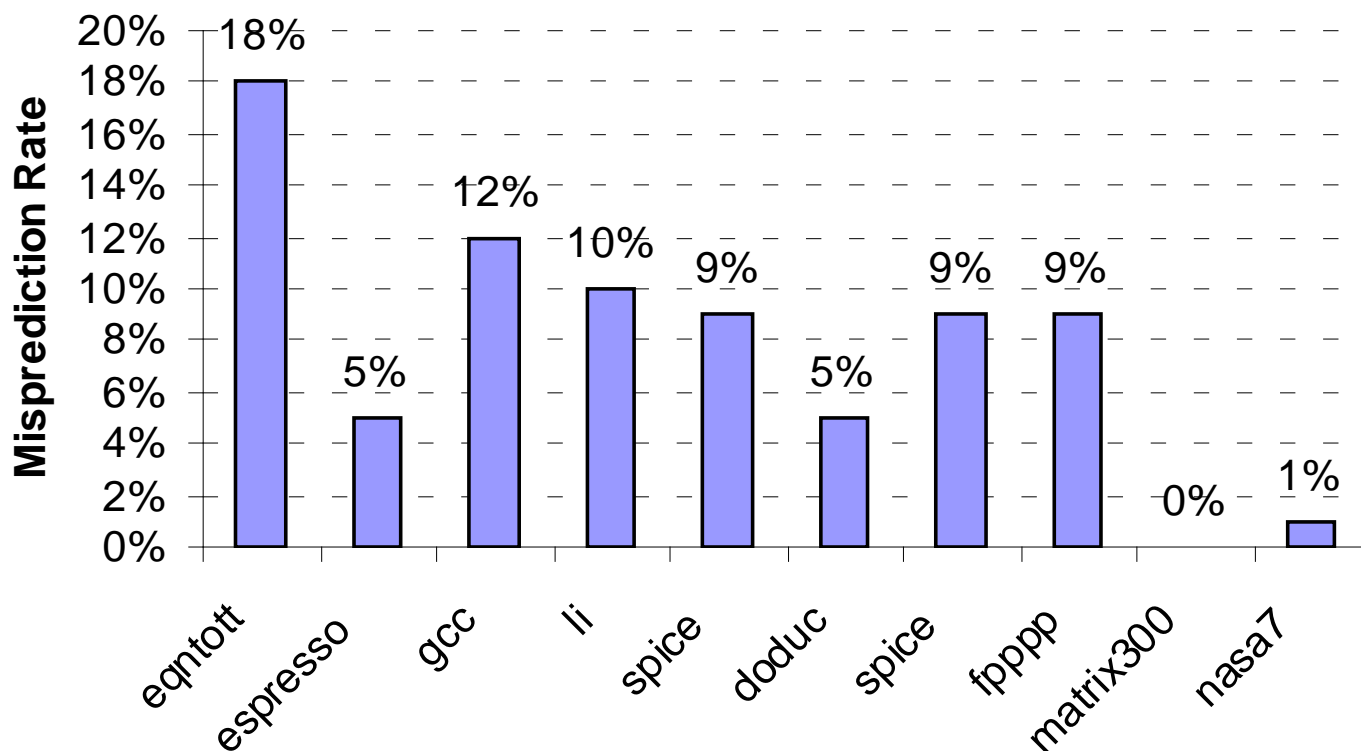
4.3.1 采用分支历史表 BHT

- ◆ BHT方法只在以下情况下才有用：
 - 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。
- ◆ 研究结果表明：对于SPEC89测试程序来说，具有大小为4K的BHT的预测准确率为82%~99%。
 - 一般来说，采用4K的BHT就可以了
 - 再增加项数，对提高预测准确率几乎没有效果
(in Alpha 21164)
- ◆ BHT可以跟分支指令一起存放在指令Cache中，也可以用专门的一个硬件来实现。

4.3.1 采用分支历史表 BHT

◆ 分支预测错误的原因：

- 预测错误
- 由于**使用PC的低位查找BHT表**，可能得到错误的分支历史记录



SPEC89预测结果

关联/两级预测器

◆ 例如：

```
if (aa==2) aa=0;  
if (bb==2) bb=0;  
if (aa!=bb) {
```

◆ 翻译为MIPS

```
    SUBI R3,R1,#2  
    BNEZ R3,L1      ; branch b1 (aa!=2)  
    ADDI R1,R0,R0   ; aa=0  
L1: SUBI R3,R2,#2  
    BNEZ R3,L2      ; branch b2(bb!=2)  
    ADDI R2,R0,R0   ; bb=0  
L2: SUBI R3,R1,R2    ; R3=aa-bb  
    BEQZ R3,L3      ; branch b3 (aa==bb)
```

◆ 观察结果：

- ◆ b3 与分支b2 和b1相关
- ◆ 如果b1和b2都分支失败，则b3一定成功。

两级预测器-举例

- ◆ 假设d的初始值序列为 0, 1, 2
- ◆ b1 如果分支失败, b2一定也分支失败。
- ◆ 前面的两位标准的预测方案就没法利用这一点, 而关联/两级预测方案就可以。

◆ 代码

```
if (d==0) d=1;  
if (d==1) d=0;
```

◆ 翻译为DLX

```
BNEZ R1,L1           ;branch b1(d!=0)  
ADDI R1,R0,#1         ;d==0, so d=1  
L1: ADDI R3,R1,#-1  
    BNEZ R3,L2         ;branch b2(d!=1)  
    ...  
L2:
```

Initial value of d	d==0?	b1	Value of d before b2	d==1?	b2
0	yes	not taken	1	yes	not taken
1	no	taken	1	yes	not taken
2	no	taken	2	no	taken

两级预测器-举例

- ◆ 假设d的初始值在2和0之间切换。
- ◆ 用1-bit预测器，**初始设置为预测失败**，T表示预测成功，NT表示预测失败。
- ◆ 结论：这样的序列每次预测都错，预测错误率100%

```

BNEZ R1,L1           ;branch b1(d!=0)
ADDI R1,R0,#1        ;d==0, so d=1
L1: ADDI R3,R1,#-1
BNEZ R3,L2           ;branch b2(d!=1)
    
```

d=?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT

两级预测器

◆ 基本思想：

- **用1位作为相关位**(correlation)。即每个分支都有两个相互独立的预测位：
 - 一个预测位假设最近一次执行的分支失败时的预测位，
 - 另一个预测位是假设最近一次执行的分支成功时的预测位
- ◆ 最近一次执行的分支与要预测的分支可能不是同一条指令
- ◆ 记为 (1, 1) 前一位表示最近一次分支失败时的预测位，后一位表示最近一次分支成功时的预测位

Prediction bits	Prediction if last branch	
	not taken	Prediction if last branch taken
NT/NT	not taken	not taken
NT/T	not taken	taken
T/NT	taken	not taken
T/T	taken	taken

两级预测器的预测和执行情况

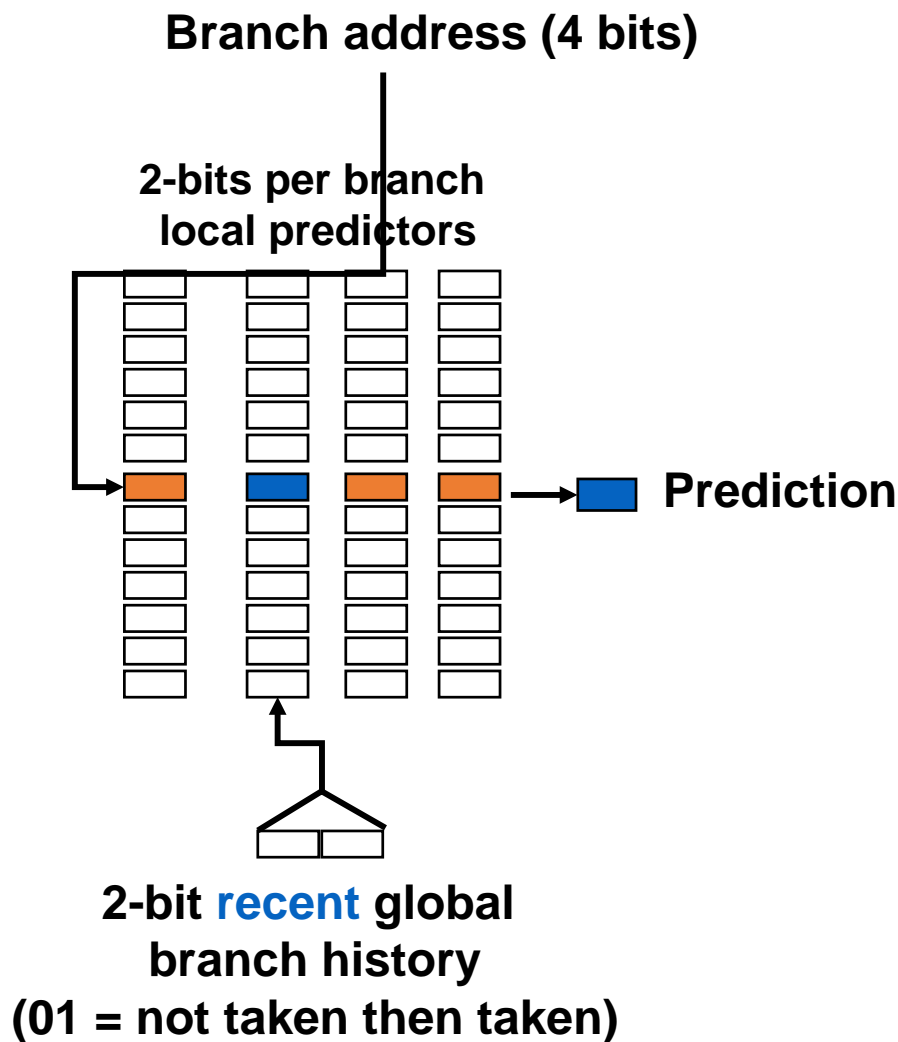
- ◆ 显然只有在第一次 $d=2$ 时，预测错误，其他都预测正确
- ◆ 记为 **(1, 1) 预测器**，即根据最近一次分支的行为来选择一对1-bit预测器中的一个。
- ◆ 更一般的表示为 (m, n) ，即根据最近的 m 个分支，从 2^m 个分支预测器中选择预测器，每个预测器的位数为 n

```

BNEZ R1,L1           ;branch b1(d!=0)
ADDI R1,R0,#1        ;d==0, so d=1
L1: ADDI R3,R1,#-1
BNEZ R3,L2           ;branch b2(d!=1)
    
```

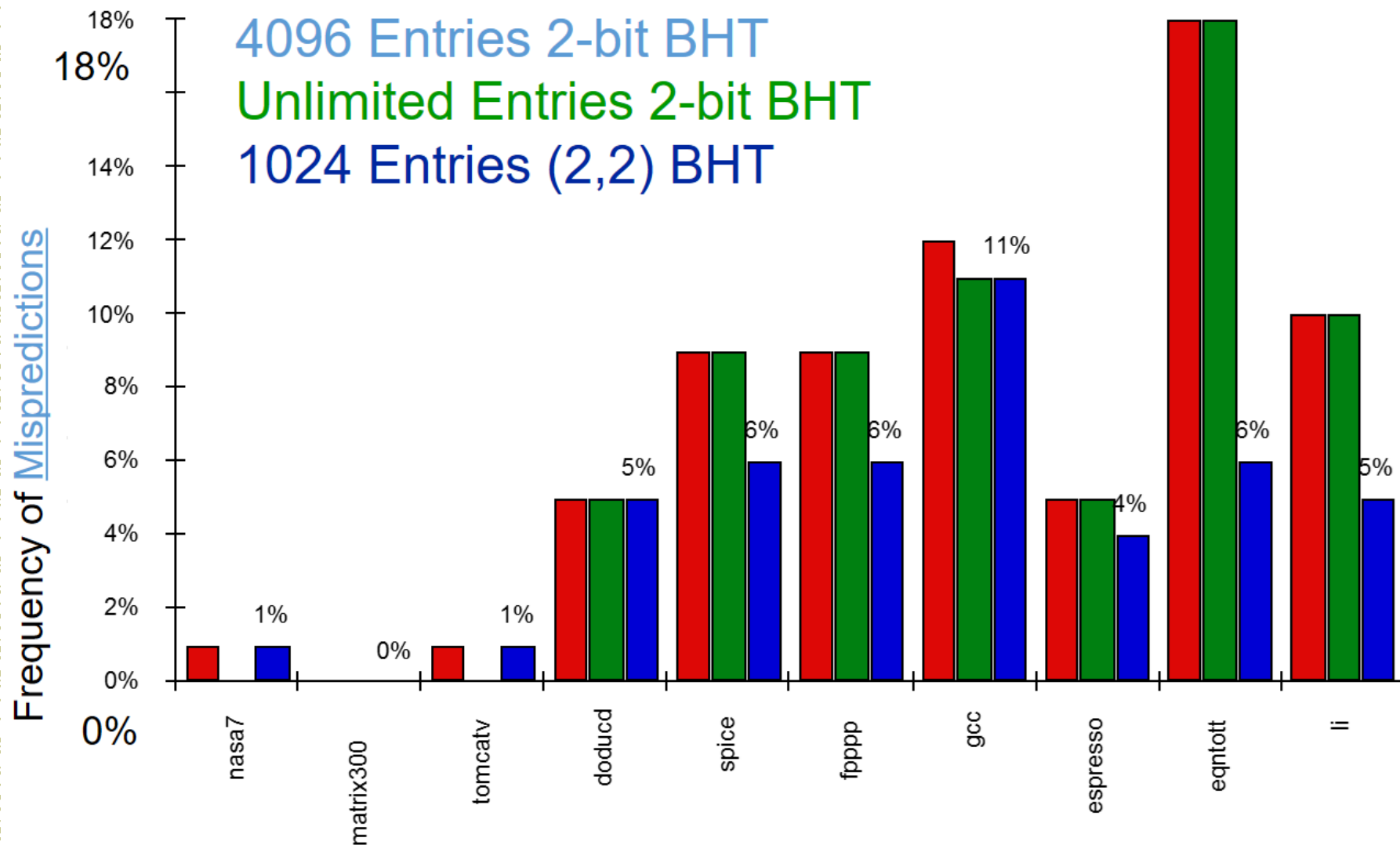
d=?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	NT/NT	T	T/NT	NT/NT	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T
2	T/NT	T	T/NT	NT/T	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T

两级预测器



- ◆ (2,2) 两级预测器：2-bit全局，2-bit局部

不同方案预测精度比较



4.3.2 采用分支目标缓冲器BTB

- ◆ **目标**：将分支的开销降为 0，比BHT更先进
- ◆ **方法**：分支目标缓冲（Branch-Target Buffer，简记为 BTB，或者Branch-Target Cache）
 - 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。
 - 完全由硬件实现，是一个Cache

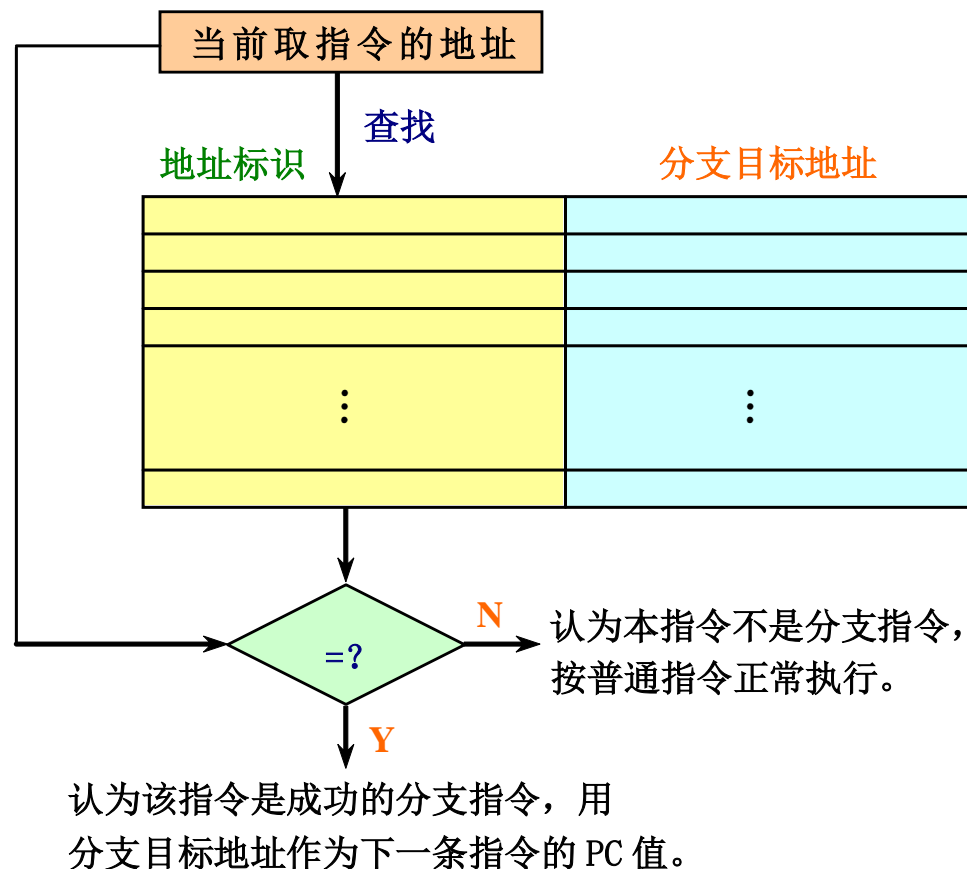
4.3.2 采用分支目标缓冲器BTB

◆ BTB的结构

- 看成是用专门的硬件实现的一张表格。

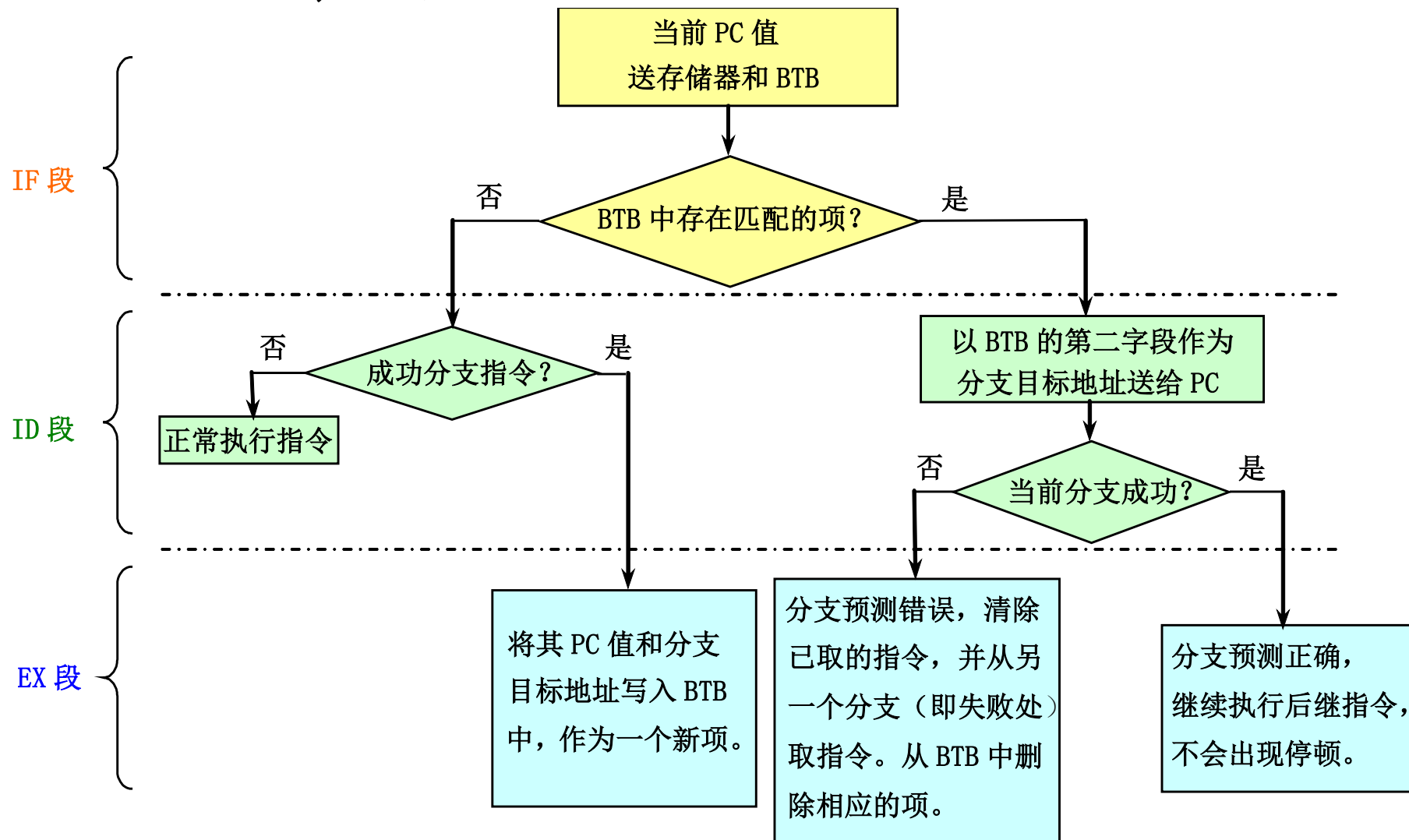
◆ 分支指令的地址作为BTB的索引，以得到分支预测地址

- 必须检测分支指令的地址是否匹配，以免用错误的分支地址
- 从表中得到预测地址
- 分支方向确定后，更新预测的PC



4.3.2 采用分支目标缓冲器BTB

- 采用BTB后，在流水线各个阶段所进行的相关操作：



4.3.2 采用分支目标缓冲器BTB

- ◆ 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

- ◆ BTB的另一种形式：在分支目标缓冲器中存放一条或者多条分支目标处的指令。有以下潜在的好处：
 - 更快地获得分支目标处的指令
 - 可以一次提供分支目标处的多条指令，这对于多流出处理器是很有必要的；（如Case语句）

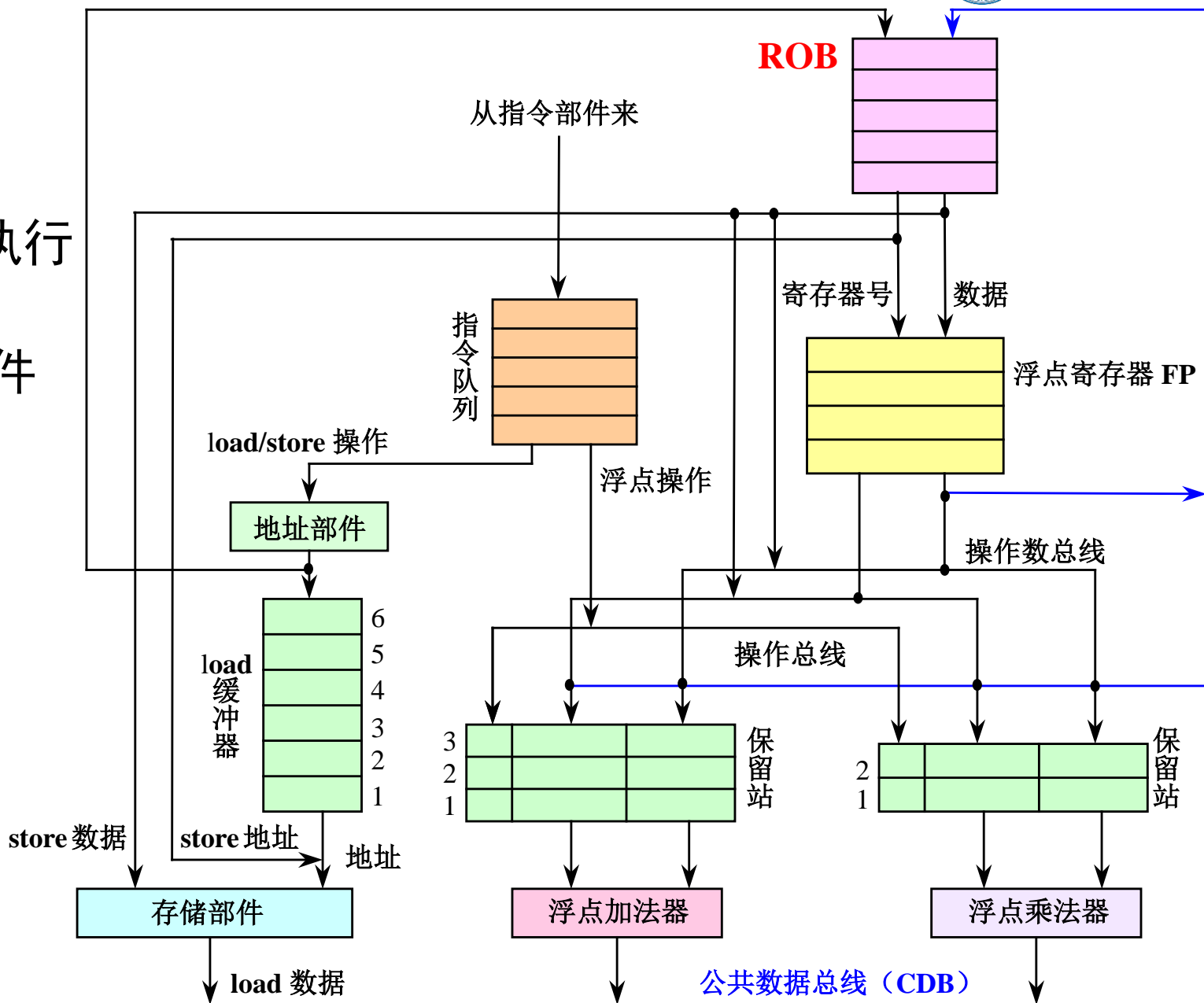
4.3.3 基于硬件的前瞻执行

- ◆ 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，而是放到一个称为**ROB**（ReOrder Buffer）的缓冲器中。
 - 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。
- ◆ 基于硬件的前瞻执行结合了三种思想：
 - 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。

4.3.3 基于硬件的前瞻执行

- ◆ 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。

支持前瞻执行的 浮点部件 的结构



4.3.3 基于硬件的前瞻执行

- ◆ 实现前瞻的关键思想：
 - 允许指令**乱序执行**，但必须**顺序确认**。
- ◆ 支持前瞻执行的浮点部件的结构
- ◆ 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

4.3.3 基于硬件的前瞻执行

- ◆ 采用前瞻执行机制后，在原Tomasulo算法的基础上改造为以下指令执行步骤：
- ◆ **Step1: 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。
- ◆ **Step2: 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。

4.3.3 基于硬件的前瞻执行

◆ Step3: 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的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，这时才将之写入分配给该store指令的ROB项。

4.3.3 基于硬件的前瞻执行

- ◆ **Step4: 确认**，对分支指令、store指令以及其他指令的处理不同：
 - **其他指令**：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标寄存器，并从ROB中删除该指令。
 - **store指令**
 - 处理与上面类似，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。

4.3.3 基于硬件的前瞻执行

◆ Step1、流出 (Issue)

Status	Wait until	Action or bookkeeping
Issue all instructions	Reservation station (r) and ROB (b) both available	<pre> if (RegisterStat[rs].Busy)/*in-flight instr. writes rs*/ {h ← RegisterStat[rs].Reorder; if (ROB[h].Ready)/* Instr completed already */ {RS[r].Vj ← ROB[h].Value; RS[r].Qj ← 0;} else {RS[r].Qj ← h;} /* wait for instruction */ } else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0;}; RS[r].Busy ← yes; RS[r].Dest ← b; ROB[b].Instruction ← opcode; ROB[b].Dest ← rd;ROB[b].Ready ← no; </pre>
FP operations and stores		<pre> if (RegisterStat[rt].Busy) /*in-flight instr writes rt*/ {h ← RegisterStat[rt].Reorder; if (ROB[h].Ready)/* Instr completed already */ {RS[r].Vk ← ROB[h].Value; RS[r].Qk ← 0;} else {RS[r].Qk ← h;} /* wait for instruction */ } else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0;}; </pre>
FP operations		<pre> RegisterStat[rd].Reorder ← b; RegisterStat[rd].Busy ← yes; ROB[b].Dest ← rd; </pre>
Loads		<pre> RS[r].A ← imm; RegisterStat[rt].Reorder ← b; RegisterStat[rt].Busy ← yes; ROB[b].Dest ← rt; </pre>
Stores		<pre> RS[r].A ← imm; </pre>

4.3.3 基于硬件的前瞻执行

◆ Step2、 执行 (Execute)

Status	Wait until	Action or bookkeeping
Execute FP op	$(RS[r].Qj == 0)$ and $(RS[r].Qk == 0)$	Compute results—operands are in Vj and Vk
Load step 1	$(RS[r].Qj == 0)$ and there are no stores earlier in the queue	$RS[r].A \leftarrow RS[r].Vj + RS[r].A;$
Load step 2	Load step 1 done and all stores earlier in ROB have different address	Read from $Mem[RS[r].A]$
Store	$(RS[r].Qj == 0)$ and store at queue head	$ROB[h].Address \leftarrow RS[r].Vj + RS[r].A;$

4.3.3 基于硬件的前瞻执行

◆ Step3、写结果，确认(Write result & Commit)

Status	Wait until	Action or bookkeeping
Write result all but store	Execution done at r and CDB available	$b \leftarrow RS[r].Dest; RS[r].Busy \leftarrow no;$ $\forall x (if (RS[x].Qj == b) \{ RS[x].Vj \leftarrow result; RS[x].Qj \leftarrow 0 \});$ $\forall x (if (RS[x].Qk == b) \{ RS[x].Vk \leftarrow result; RS[x].Qk \leftarrow 0 \});$ $ROB[b].Value \leftarrow result; ROB[b].Ready \leftarrow yes;$
Store	Execution done at r and $(RS[r].Qk == 0)$	$ROB[h].Value \leftarrow RS[r].Vk;$
Commit	Instruction is at the head of the ROB (entry h) and $ROB[h].ready == yes$	$d \leftarrow ROB[h].Dest; /* register dest, if exists */$ $if (ROB[h].Instruction == Branch)$ $\quad \{ if (branch is mispredicted)$ $\quad \quad \{ clear ROB[h], RegisterStat; fetch branch dest; \}; \}$ $else if (ROB[h].Instruction == Store)$ $\quad \{ Mem[ROB[h].Destination] \leftarrow ROB[h].Value; \}$ $else /* put the result in the register destination */$ $\quad \{ Regs[d] \leftarrow ROB[h].Value; \};$ $ROB[h].Busy \leftarrow no; /* free up ROB entry */$ $/* free up dest register if no one else writing it */$ $if (RegisterStat[d].Reorder == h) \{ RegisterStat[d].Busy \leftarrow no; \};$

基于硬件的前瞻执行-举例

- ◆ 例4.3 假设浮点功能部件的延迟时间为：加法2个时钟周期，乘法9个时钟周期，除法40个时钟周期。对于下面的代码段，给出指令执行时的状态表内容。

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

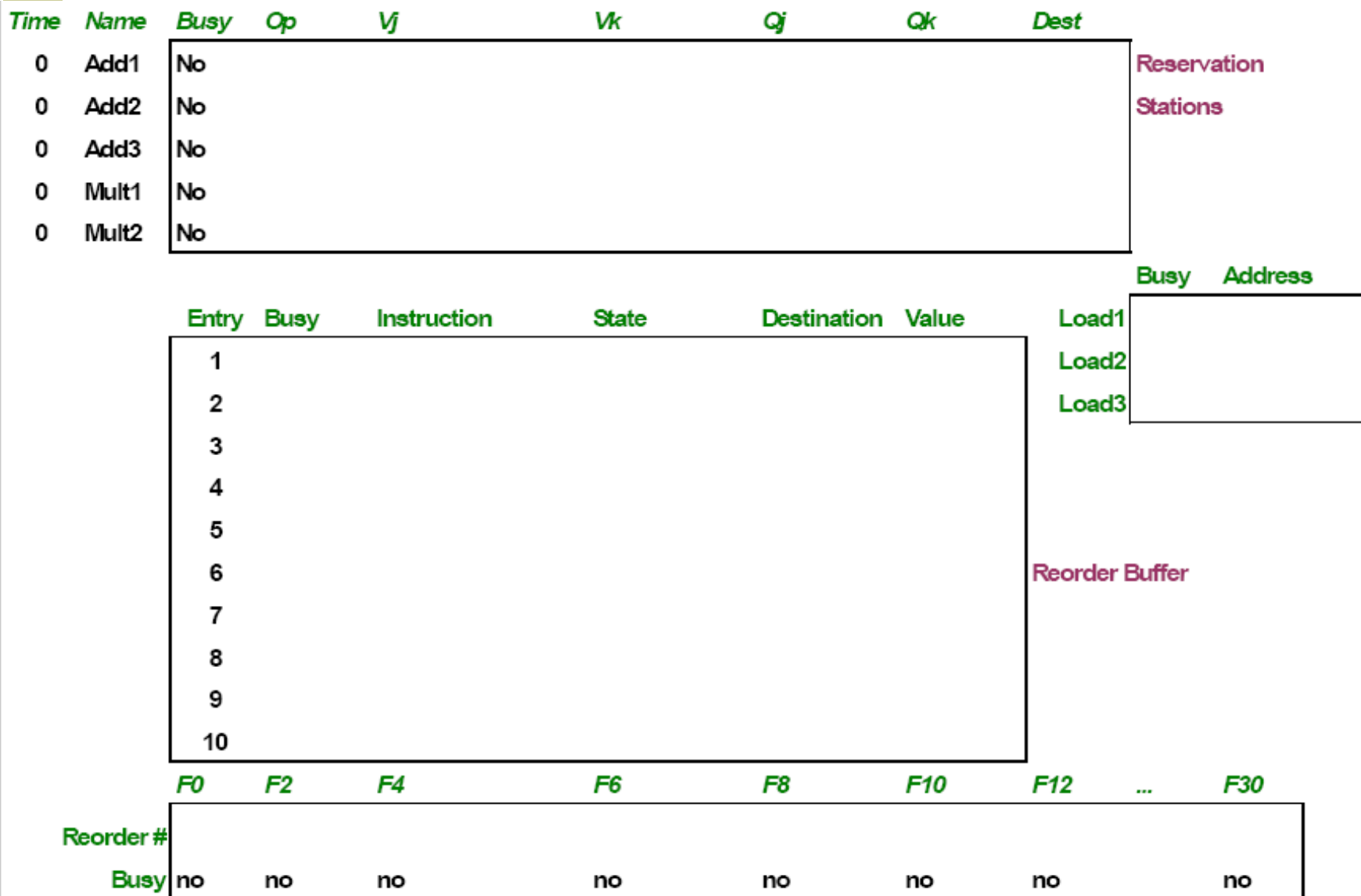
- ◆ 假设：执行阶段的周期数

LD: 1cycles MULT: 9 cycles

SUBD/ADDD: 2cycles DIVD: 40 cycles



基于前瞻执行的Tomasulo-cc 00

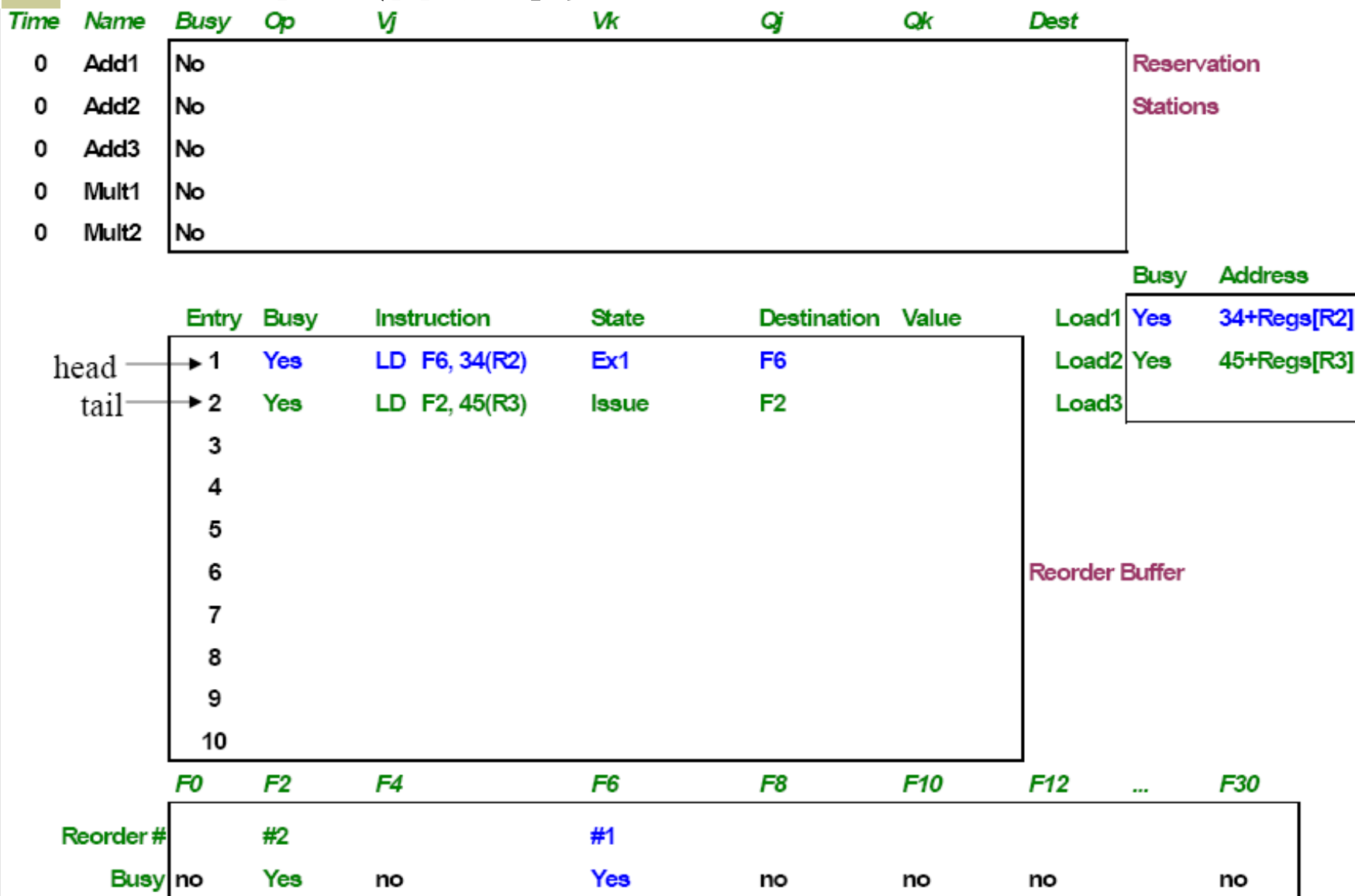


基于前瞻执行的Tomasulo-cc 01

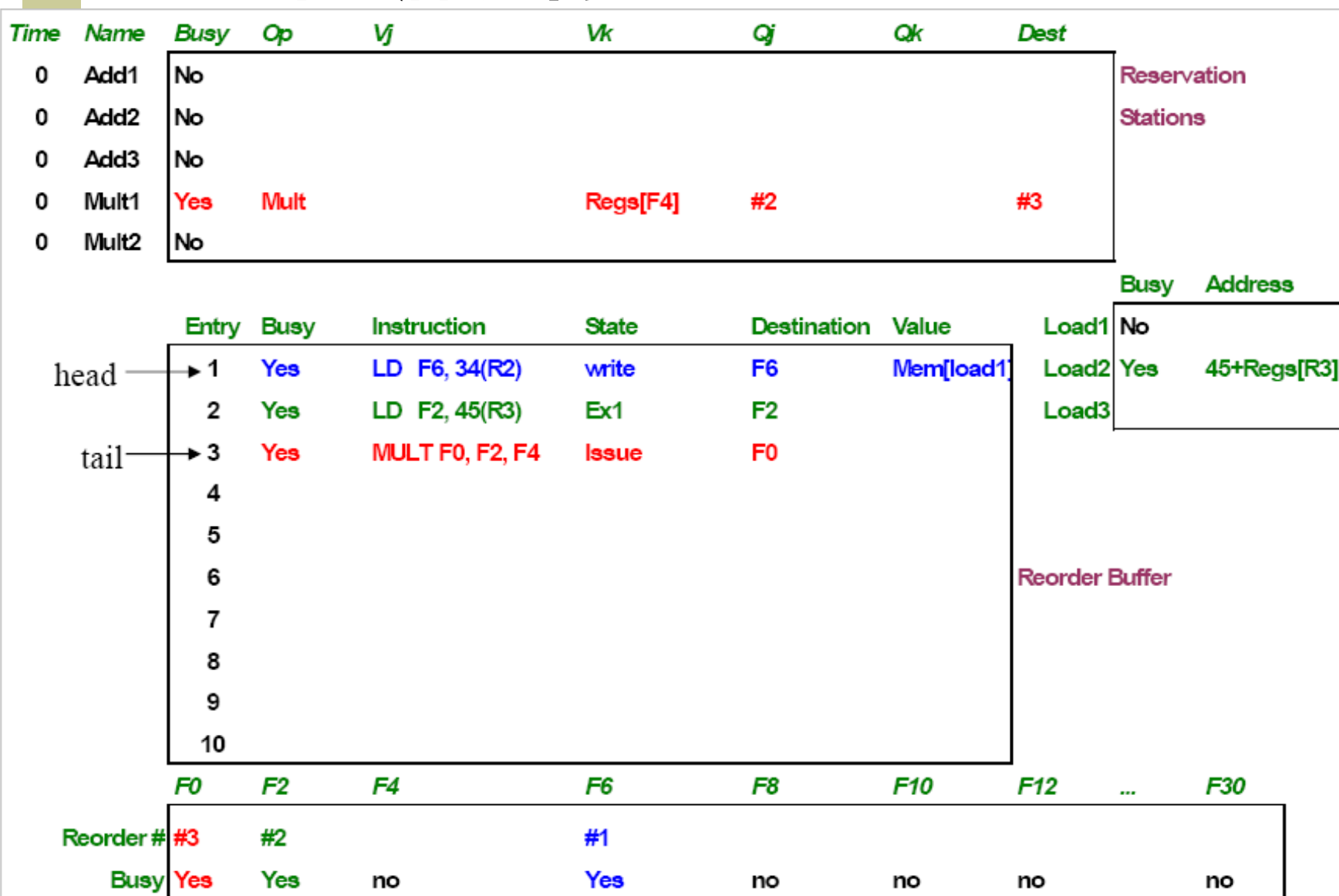
Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest		
0	Add1	No							Reservation Stations	
0	Add2	No								
0	Add3	No								
0	Mult1	No								
0	Mult2	No								
									Busy	Address
Entry	Busy	Instruction		State	Destination	Value	Load1	Load2	Load3	
1	Yes	LD F6, 34(R2)		Issue	F6		Yes		34+Regs[R2]	
2										
3										
4										
5										
6										
7										
8										
9										
10										
		F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #					#1					
Busy		no	no	no	Yes	no	no	no		no

Reorder Buffer

基于前瞻执行的Tomasulo-cc 02



基于前瞻执行的Tomasulo-cc 03



基于前瞻执行的Tomasulo-cc 04

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	Yes	SUB	Regs[F6]	Mem[45+Regs[R3]]			#4
0	Add2	No						
0	Add3	No						
0	Mult1	Yes	Mult	Mem[45+Regs[R3]]	Regs[F4]			#3
0	Mult2	No						

Busy Address

Entry	Busy	Instruction	State	Destination	Value	Load1	Load2	Load3
1	No	LD F6, 34(R2)	commit	F6	Mem[load1]	No	No	No
2	Yes	LD F2, 45(R3)	write	F2	Mem[load2]	No	No	No
3	Yes	MULT F0, F2, F4	EX1	F0				
4	Yes	SUBD F8, F6, F2	Issue	F8				
5								
6								
7								
8								
9								
10								

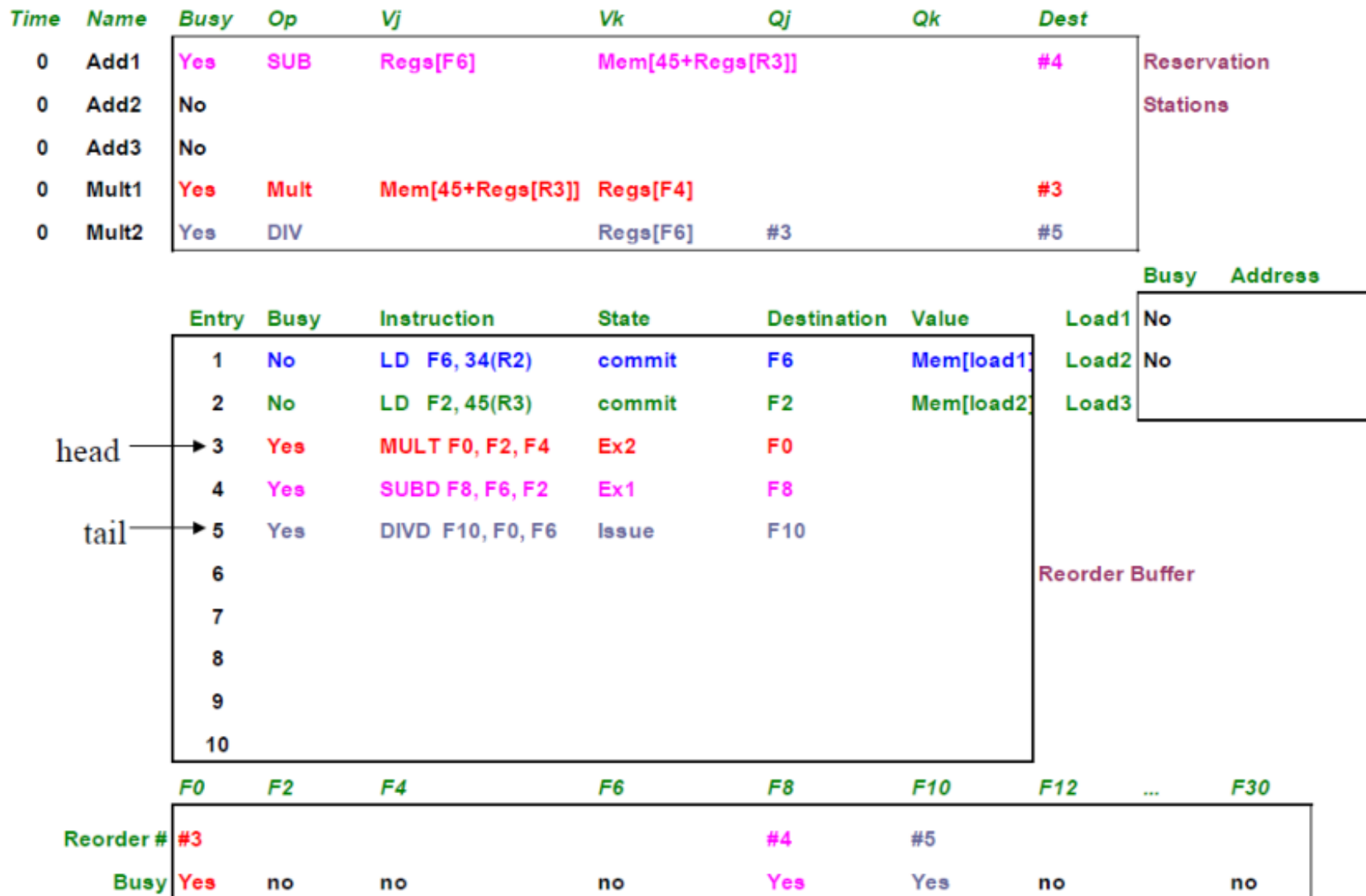
head → 2

tail → 4

Reorder Buffer

F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #	#3	#2		#4				
Busy	Yes	Yes	no	no	Yes	no	no	no

基于前瞻执行的Tomasulo-cc 05



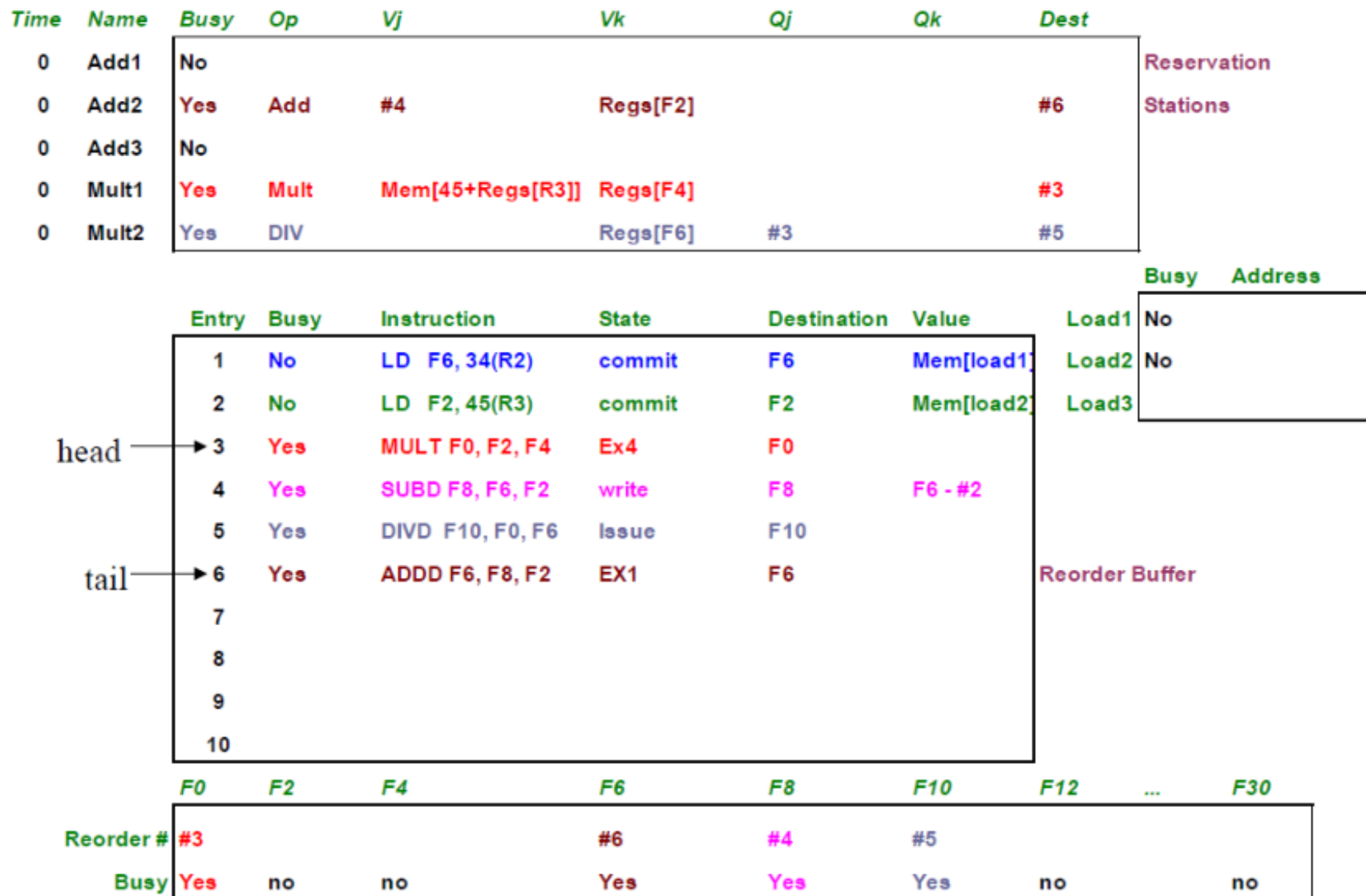
基于前瞻执行的Tomasulo-cc 06

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	Yes	SUB	Regs[F6]	Mem[45+Regs[R3]]			#4
0	Add2	Yes	Add		Regs[F2]	#4		#6
0	Add3	No						
0	Mult1	Yes	Mult	Mem[45+Regs[R3]]	Regs[F4]			#3
0	Mult2	Yes	DIV		Regs[F6]	#3		#5

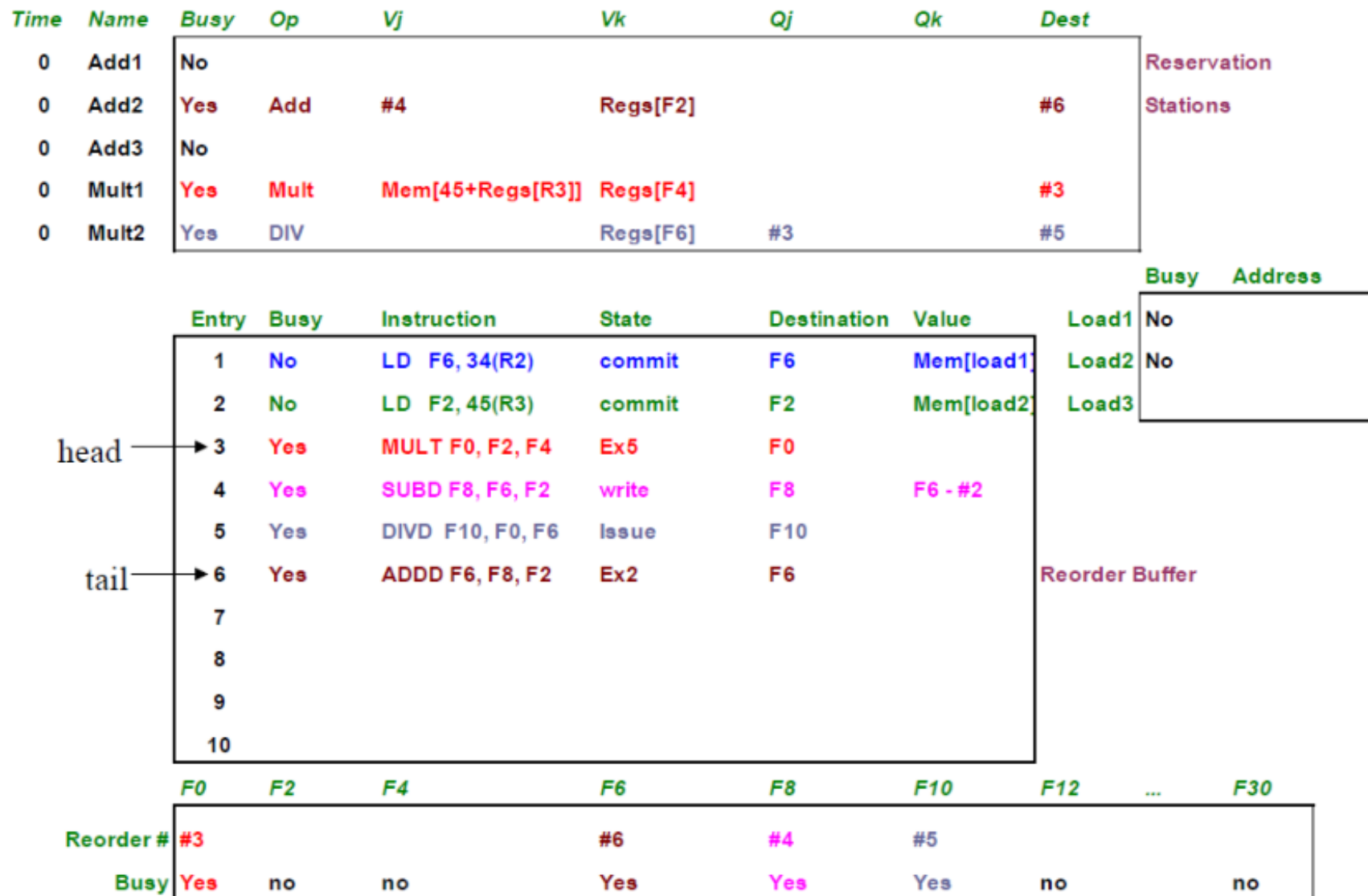
		Entry	Busy	Instruction	State	Destination	Value	Load1	Load2	Load3
		1	No	LD F6, 34(R2)	commit	F6	Mem[load1]	No	No	No
		2	No	LD F2, 45(R3)	commit	F2	Mem[load2]	No	No	No
head	→	3	Yes	MULT F0, F2, F4	Ex3	F0				
		4	Yes	SUBD F8, F6, F2	Ex2	F8				
		5	Yes	DIVD F10, F0, F6	Issue	F10				
tail	→	6	Yes	ADDD F6, F8, F2	Issue	F6				
		7								
		8								
		9								
		10								

Reorder #	F0	F2	F4	F6	F8	F10	F12	...	F30
#3				#6	#4	#5			
Busy	Yes	no	no	Yes	Yes	Yes	no		no

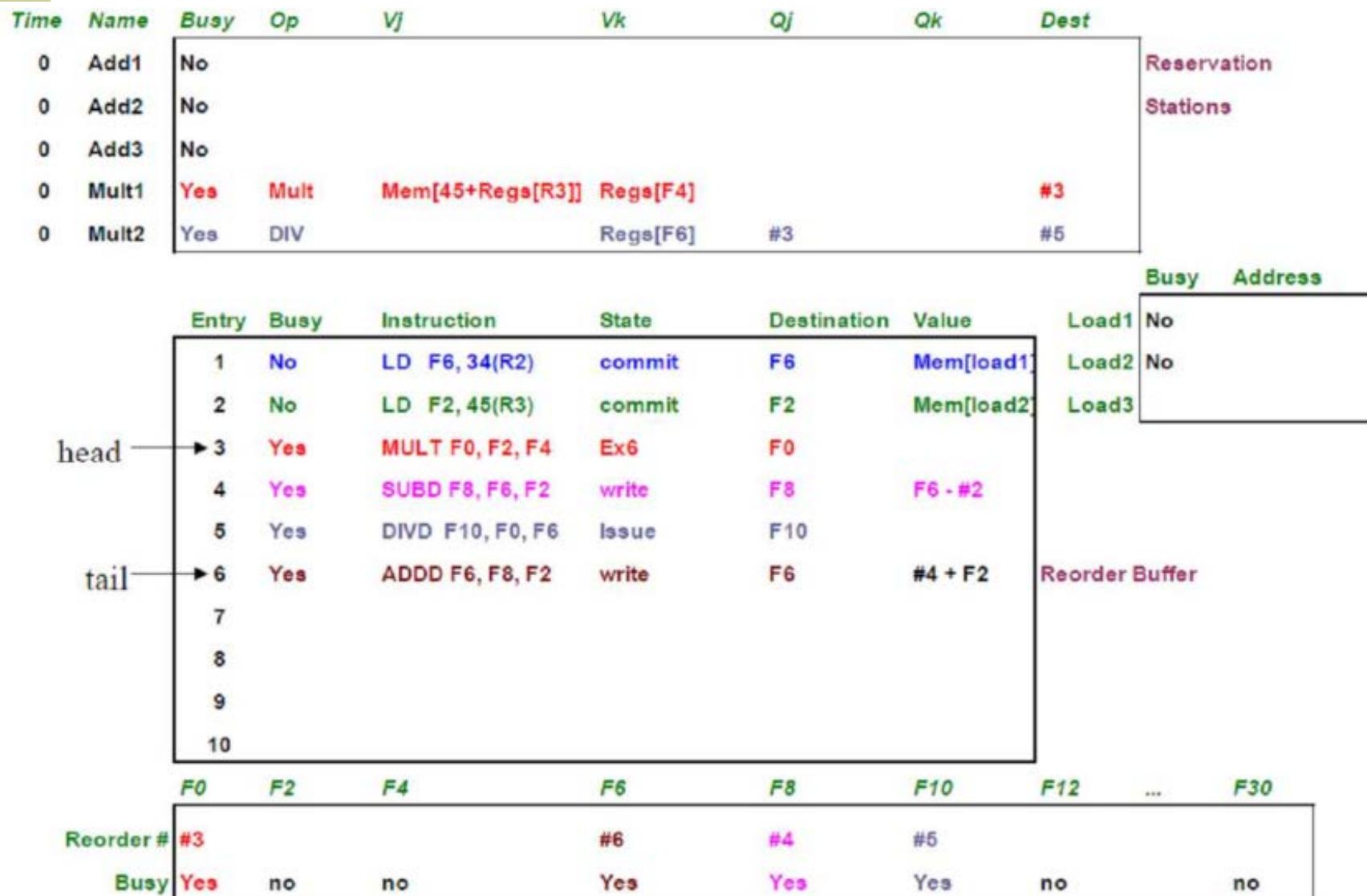
基于前瞻执行的Tomasulo-cc 07



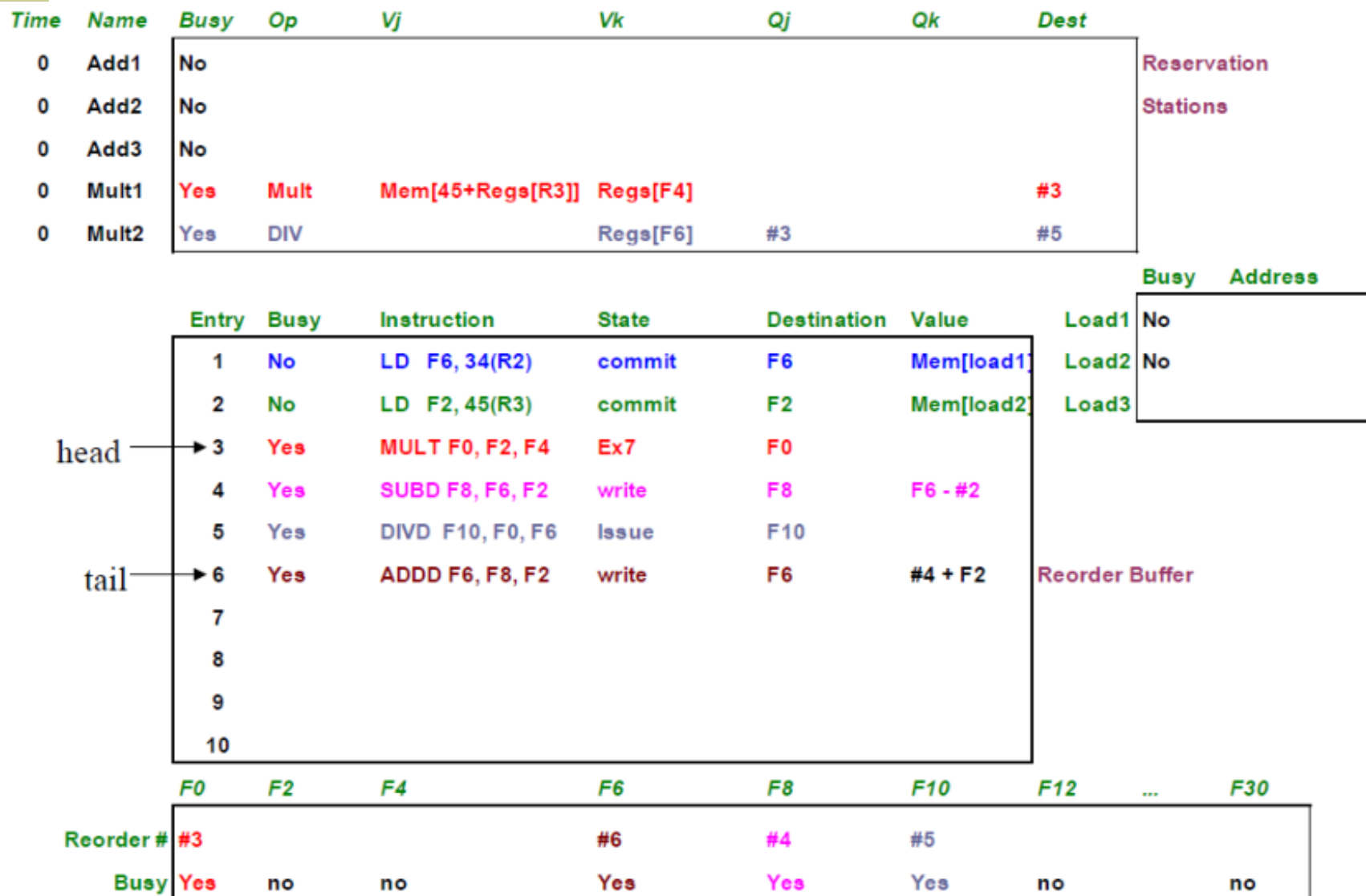
基于前瞻执行的Tomasulo-cc 08



基于前瞻执行的Tomasulo-cc 09



基于前瞻执行的Tomasulo-cc 10



基于前瞻执行的Tomasulo-cc 11

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	No						
0	Add2	No						
0	Add3	No						
0	Mult1	Yes	Mult	Mem[45+Regs[R3]]	Regs[F4]			#3
0	Mult2	Yes	DIV		Regs[F6]	#3		#5

Entry	Busy	Instruction	State	Destination	Value	Load1	Load2	Load3
1	No	LD F6, 34(R2)	commit	F6	Mem[load1]	No	No	No
2	No	LD F2, 45(R3)	commit	F2	Mem[load2]	No	No	No
3	Yes	MULT F0, F2, F4	Ex8	F0				
4	Yes	SUBD F8, F6, F2	write	F8	F6 - #2			
5	Yes	DIVD F10, F0, F6	Issue	F10				
6	Yes	ADDD F6, F8, F2	write	F6	#4 + F2			
7								
8								
9								
10								

Reorder #	F0	F2	F4	F6	F8	F10	F12	...	F30
#3				#6	#4	#5			
Busy	Yes	no	no	Yes	Yes	Yes	no		no

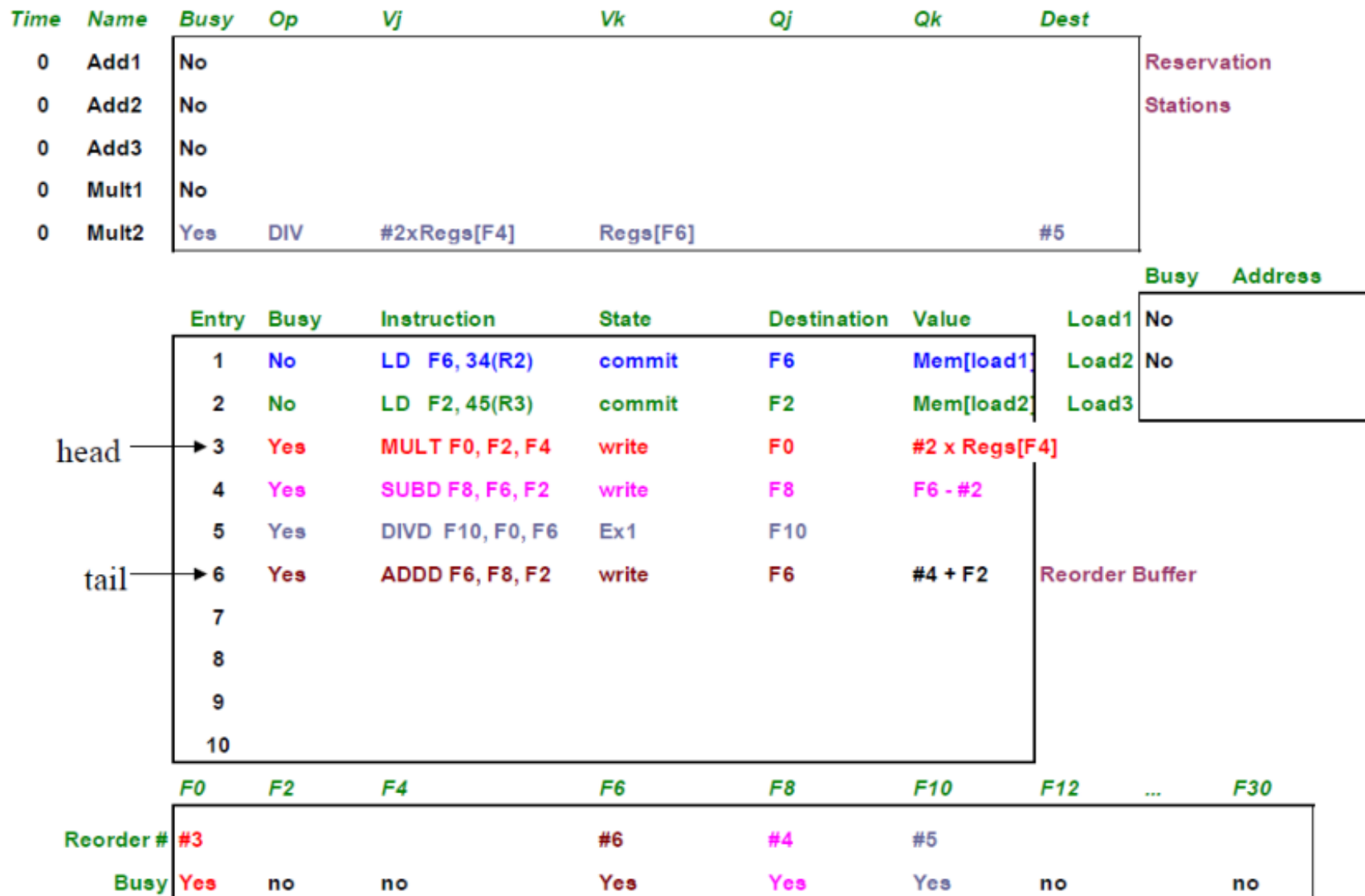
基于前瞻执行的Tomasulo-cc 12

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	No						
0	Add2	No						
0	Add3	No						
0	Mult1	Yes	Mult	Mem[45+Regs[R3]]	Regs[F4]			#3
0	Mult2	Yes	DIV		Regs[F6]	#3		#5

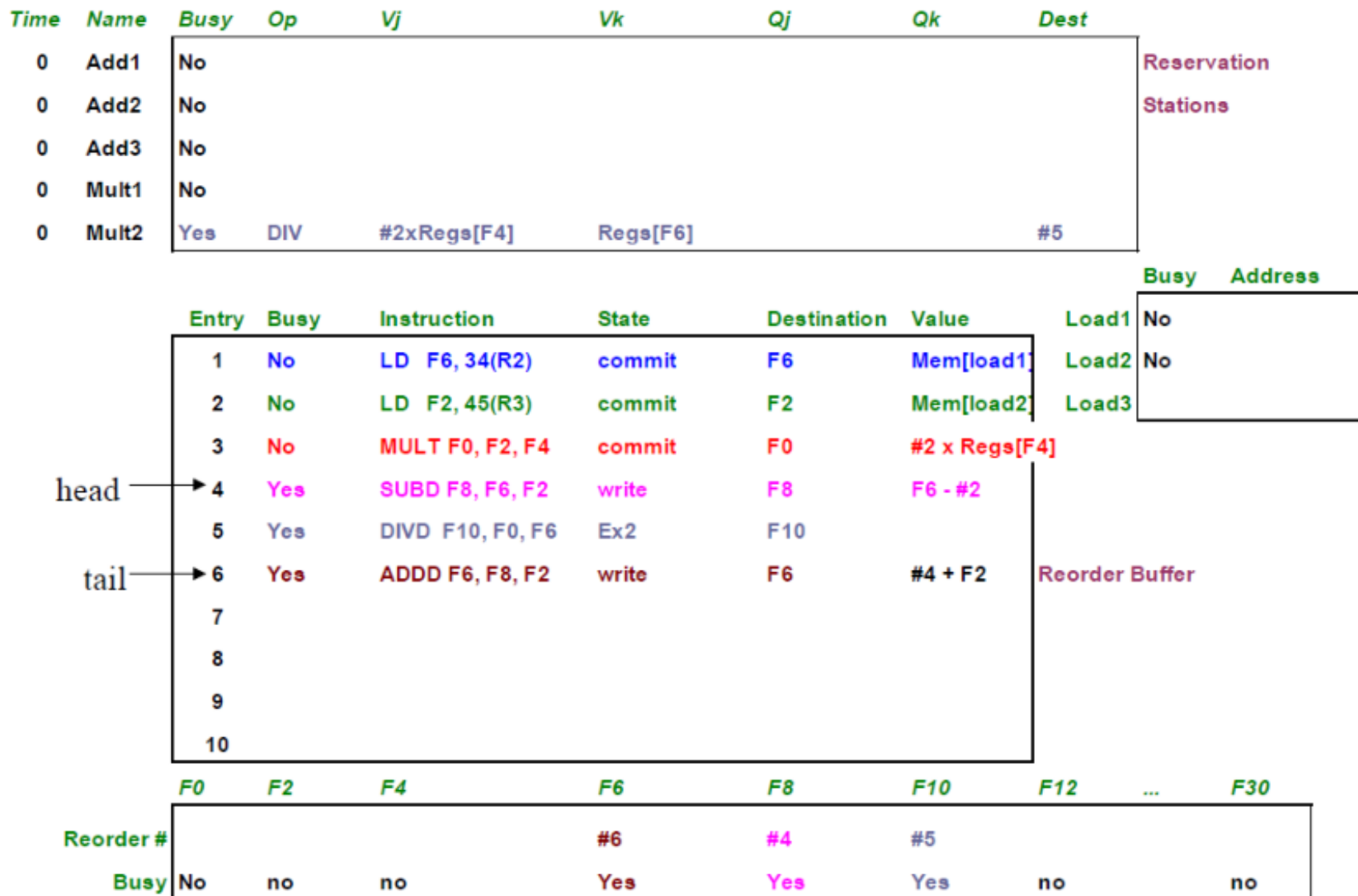
Entry	Busy	Instruction	State	Destination	Value	Load1	Load2	Load3
1	No	LD F6, 34(R2)	commit	F6	Mem[load1]	No	No	No
2	No	LD F2, 45(R3)	commit	F2	Mem[load2]	No	No	No
3	Yes	MULT F0, F2, F4	Ex9	F0				
4	Yes	SUBD F8, F6, F2	write	F8	F6 - #2			
5	Yes	DIVD F10, F0, F6	issue	F10				
6	Yes	ADDD F6, F8, F2	write	F6	#4 + F2			
7								
8								
9								
10								

Reorder #	F0	F2	F4	F6	F8	F10	F12	...	F30
#3				#6	#4	#5			
Busy	Yes	no	no	Yes	Yes	Yes	no		no

基于前瞻执行的Tomasulo-cc 13



基于前瞻执行的Tomasulo-cc 14



基于前瞻执行的Tomasulo-cc 15

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	No						
0	Add2	No						
0	Add3	No						
0	Mult1	No						
0	Mult2	Yes	DIV	#2xRegs[F4]	Regs[F6]			#5

Entry	Busy	Instruction	State	Destination	Value	Load1	Load2	Load3
1	No	LD F6, 34(R2)	commit	F6	Mem[load1]	No	No	No
2	No	LD F2, 45(R3)	commit	F2	Mem[load2]	No	No	No
3	No	MULT F0, F2, F4	commit	F0	#2 x Regs[F4]			
4	No	SUBD F8, F6, F2	commit	F8	F6 - #2			
5	Yes	DIVD F10, F0, F6	Ex3	F10				
6	Yes	ADDD F6, F8, F2	write	F6	#4 + F2			
7								
8								
9								
10								

Reorder #	F0	F2	F4	F6	F8	F10	F12	...	F30
				#6		#5			
Busy	no	no	no	Yes	no	Yes	no		no

基于前瞻执行的Tomasulo-cc 16

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	No						
0	Add2	No						
0	Add3	No						
0	Mult1	No						
0	Mult2	Yes	DIV	#2xRegs[F4]	Regs[F6]			#5

Reservation Stations

Entry	Busy	Instruction	State	Destination	Value	Load1	Load2	Load3
1	No	LD F6, 34(R2)	commit	F6	Mem[load1]	No	No	No
2	No	LD F2, 45(R3)	commit	F2	Mem[load2]	No	No	No
3	No	MULT F0, F2, F4	commit	F0	#2 x Regs[F4]			
4	No	SUBD F8, F6, F2	commit	F8	F6 - #2			
5	Yes	DIVD F10, F0, F6	Ex4	F10				
6	Yes	ADDD F6, F8, F2	write	F6	#4 + F2			
7								
8								
9								
10								

Reorder Buffer

Need 36 more EX cycles for DIV to finish...

Reorder #	F0	F2	F4	F6	F8	F10	F12	...	F30
Busy	no	no	no	Yes	no	Yes	no		no

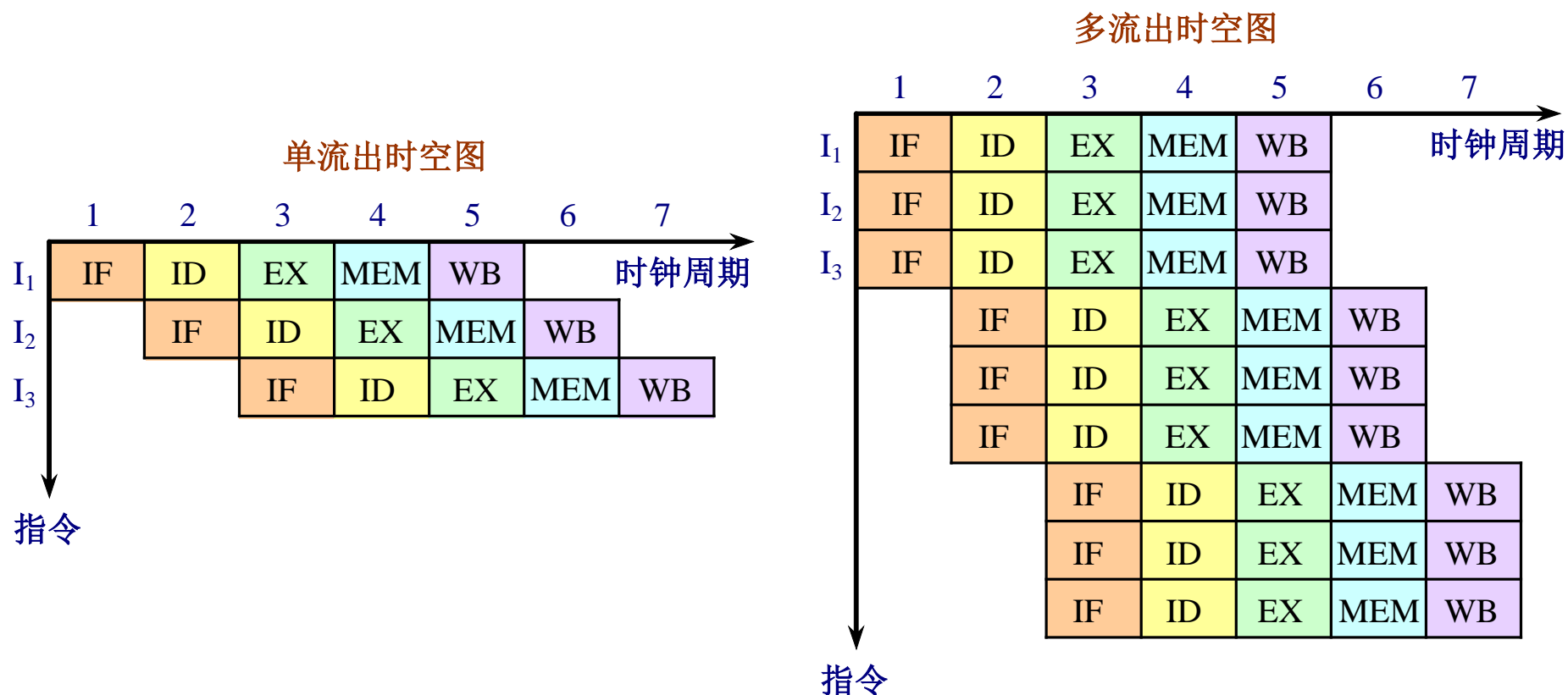
基于前瞻执行的Tomasulo-总结

Instruction	Issue	Exec Comp	Writeback	Commit
LD F6, 34(R2)	1	2	3	4
LD F2, 45(R3)	2	3	4	5
MULT F0, F2, F4	3	12	13	14
SUBD F8, F6, F2	4	6	7	15
DIVD F10, F0, F6	5	52	53	54
ADDD F6, F8, F2	6	8	9	55

In-order Issue/Commit, Out-of-Order Execution/Writeback

4.4 多指令流出技术

- ◆ 一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



4.4 多指令流出技术

◆ 多发射

- 迄今为止介绍的各类提高性能的技术都是围绕使 $CPI=1$ 这一目标展开的。
 - 如：消除数据相关、控制相关、静态调度、动态调度等
- 根据公式 $CPU\ time = IC \times CPI \times cycle\ time$ ，进一步提高性能的启发是使 $CPI < 1$ ，即：必须在一个时钟周期里发射多条指令，即指令的多发射技术。

- ◆ **基本前提**：有足够硬件，即功能单元、寄存器、及存储器带宽的基础上。也就是说 **不存在结构竞争**。

核心技术：指令多发射技术

- ◆ 技术路线：三种高性能的指令级并行处理机
 - 超标量处理机 (Super Scalar Processor)
 - 超流水线处理机 (Super Pipelining Processor)
 - 超标量流水线处理机 (Super Pipelining Super Scalar Processor)
- ◆ 共同特点：
 - 在一个时钟周期内可以发射多条指令，输出多条指令

1、超标量处理机

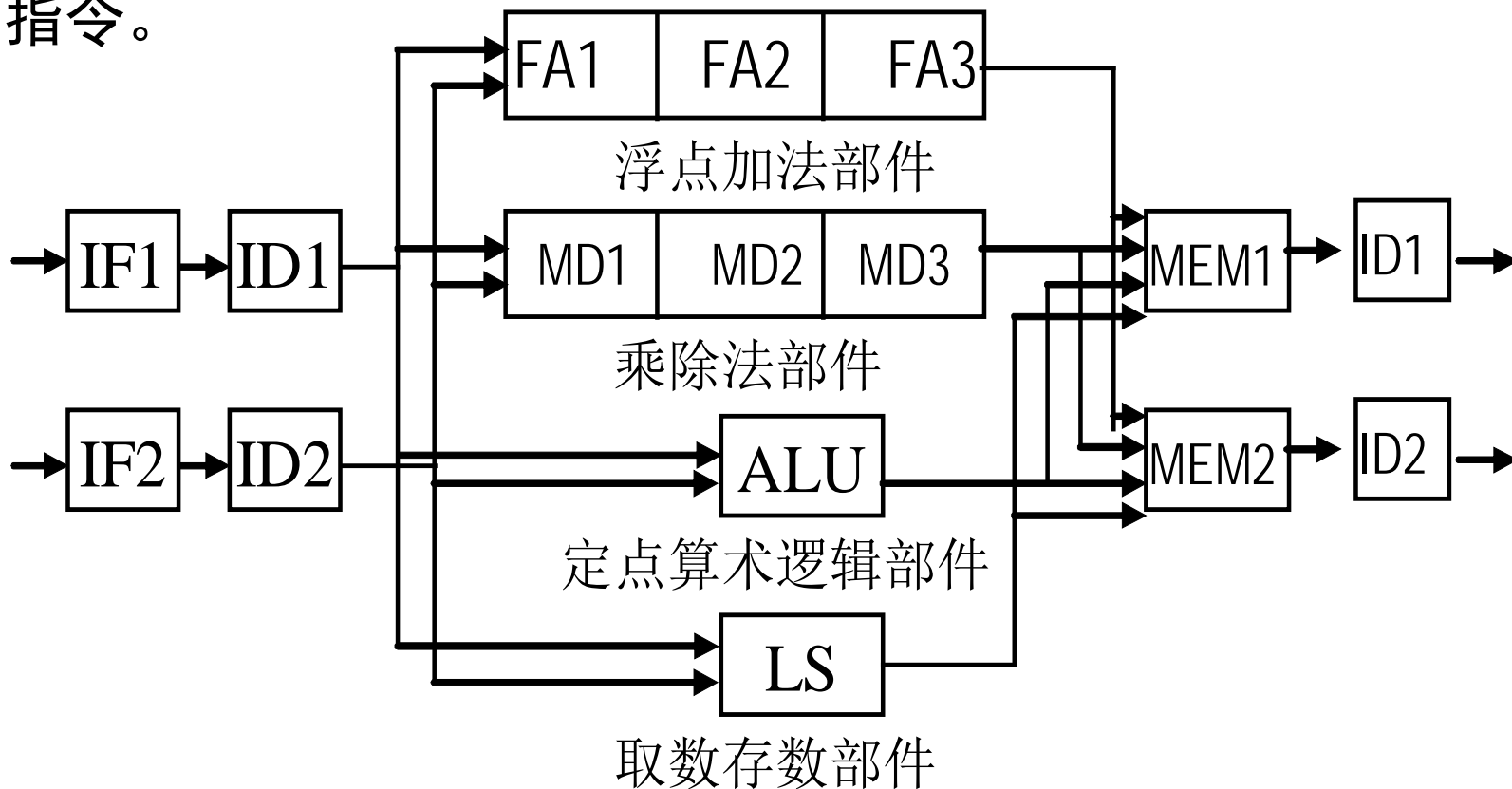
- ◆ 典型特征：有多个操作部件，一个或几个比较大的通用寄存器堆。
- ◆ 典型处理机：Intel 的Pentium, Motorola的MC88110, IBM 的Power PC, 以及AMD的Opteron等。
- ◆ 主要特点：
 - 在一个周期里能发射 n 条指令（ n 一般为1~8），就称该处理机为 **n 流出**；
 - 同时发射的指令**按一定规律搭配**，即有一定限制，不能自由搭配；
 - 用静态调度（compiler完成）和/或动态调度（硬件完成）方法确定可同时发射的指令条数。

4.4.1 基于静态调度的多流出技术

- ◆ 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。**
 - 在当前流出的指令序列中，不存在数据冲突或者相关冲突
- ◆ 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）。
 - 流出部件检测结构冲突或者数据冲突。一般分两阶段：
 - 第一阶段：进行流出包内的冲突检测，选出初步判定可以流出的指令。
 - 第二阶段：检测所选出的指令与正在执行的指令是否有冲突。

双发射流水线结构示意图

- ◆ 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。



双发射处理器的流水时序

- ◆ 双流出超标量流水线中指令的执行过程
 - 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，下图中总是把整数指令放在浮点指令的前面

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

4.4.1 基于静态调度的多流出技术

- ◆ 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。
- ◆ 由于流水线中的指令多了一倍，定向路径也要增加
- ◆ 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，则其延迟是3条指令；
 - 否则就是流出包中的第二条指令，其延迟就是两条指令

4.4.2 基于动态调度的多流出技术

- ◆ **扩展Tomasulo算法**：支持两路超标量
 - 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。
- ◆ 采用一种比较简单的方法
 - 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，这样就可以同时地流出一条浮点指令和一条整数指令到各自的保留站。
- ◆ 有两种不同的方法可以实现多流出。
 - 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。
 - 设置一次能同时处理两条指令的逻辑电路。
 - 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。

4.4.2 基于动态调度的多流出技术

- ◆ 例4.4 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）
      // 若R1不等于R2，表示尚未结束，转移到Loop继续
      BNE      R1, R2, Loop
```

- ◆ 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。

例4.4 假设条件

- ◆ 每个时钟周期能流出一条整数指令和一条浮点指令，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。
- ◆ 分支指令单独流出，没有采用延迟分支，但分支预测是完美的。**分支指令完成前，其后续指令只能被取出和流出，但不能执行。**
- ◆ 因为写结果占用一个时钟周期，所以**产生结果的延迟**为：整数运算一个周期，load两个周期，浮点加法运算3个周期。



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
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

4.4.2 基于动态调度的多流出技术

- ◆ 执行时，该循环将动态展开，并且只要可能就流出两条指令。
- ◆ 为了便于分析，表中列出了访存发生的时刻。运行结果如上图所示。
 - 程序基本可以达到3拍流出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。
 - 16拍共执行15条指令，
 - 平均指令执行速度为 $15/16 = 0.94$ 条/拍。
 - 原因是浮点运算少，ALU部件成了瓶颈。
 - 解决方法：增加一个加法器，把ALU功能和地址运算功能分开

双流出动态调度流水线的性能分析

◆ 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。
- 每个循环迭代中的**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。

4.4.3 超长指令字技术（VLIW）

- ◆ VLIW（Very Long Instruction Word）在一个时钟周期里发射固定数量的指令，实际为一条长指令，该VLIW按固定格式组织的；
- ◆ **VLIW由Compiler组织的，即不能由硬件动态组织。**
- ◆ 主要特点：
 - 以一条长指令实现多条指令的并行执行，减少存储器访问
 - 单一的控制流：只有一个控制器，每个周期启动一条指令
 - 超长指令字被分成多个控制字段，每个字段直接独立地控制每个功能部件。
 - 在编译阶段完成多个可并行执行操作的调度。

VLIW究竟该有多长？

- ◆ 以一个拥有7个功能单元的VLIW处理器为例，指令长度有多少位？
- ◆ 设7个功能单元可支持2个整数操作，2个FP操作，2个memory访问操作和1个分支操作，则实际上一条VLIW含7条指令。
- ◆ 为支持每一功能单元正常工作，需分配每一个功能单元相应的数据域。一般每一数据域为16-24位。
- ◆ VLIW长度为： $16 \times 7 = 112 \text{ bits}$
或： $24 \times 7 = 168 \text{ bits}$

VLIW存在的一些问题

◆ 程序代码长度增加了

- 提高并行性而进行了大量的循环展开。
- 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的上述问题是致命的，因而没有得到实际应用

理想情况下的超标量处理机的性能

◆ 单流水线性能：

$$T(1,1) = (k + N - 1)\Delta t$$

◆ m度超标量处理机性能：

- $K\Delta t$ 为开始m条指令的执行时间

$$T(m,1) = \left(k + \frac{N-m}{m} \right) \Delta t$$

◆ m度超标量处理机对单流水线的加速比：

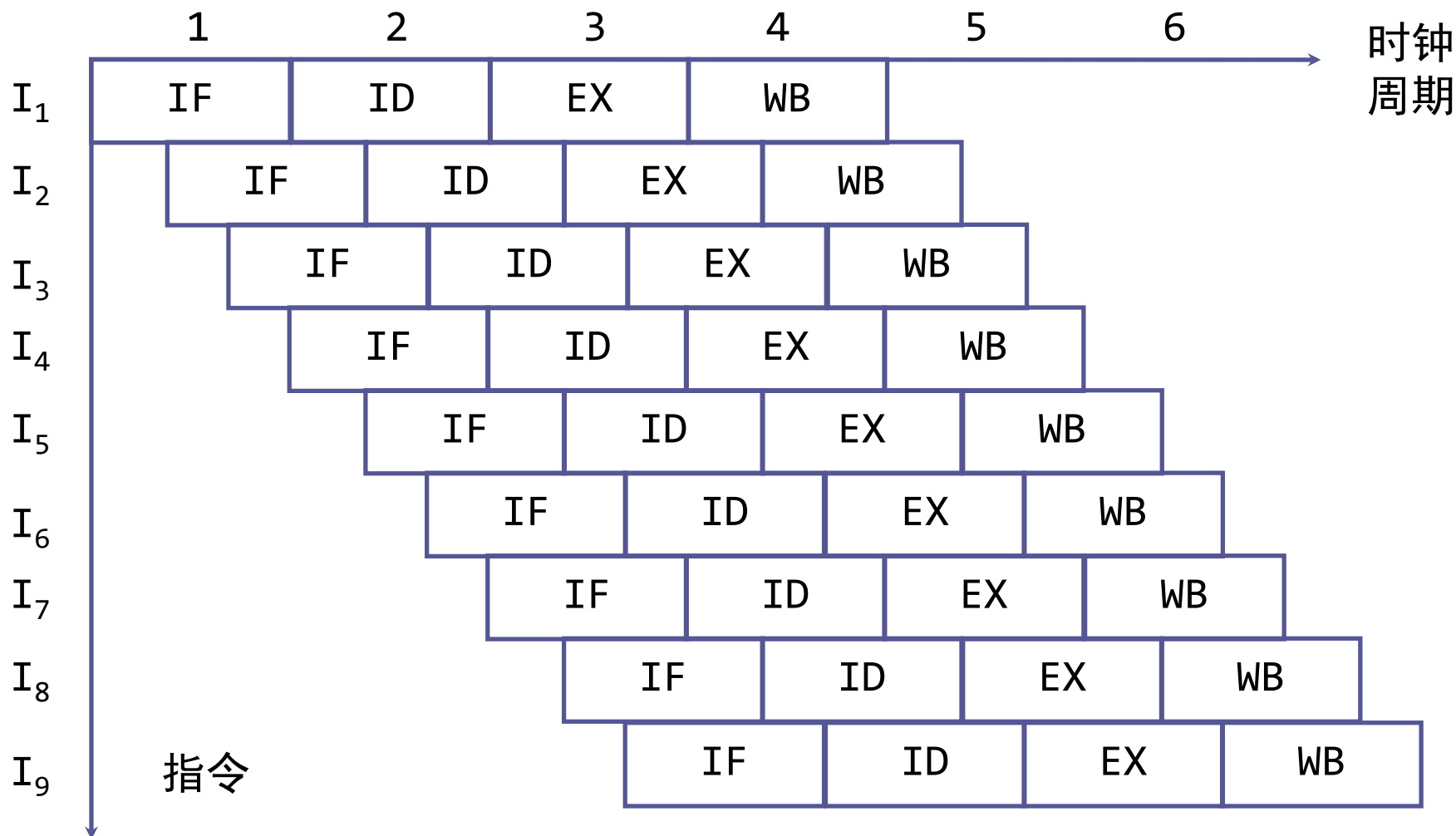
$$S(m,1) = \frac{T(1,1)}{T(m,1)} = \frac{m(k + N - 1)}{N + m(k - 1)}$$

4.4.5 超流水线处理机

- ◆ 定义1：一个周期内能够分时发射多条指令的处理机称为 **超流水线处理机**。
- ◆ 定义2：指令流水线有**8个或更多功能段**的流水线处理机称为超流水线处理机。
- ◆ 超流水线处理机采用的是**时间并行性**
- ◆ 指令执行时序
 - 每隔 $1/n$ 个时钟周期发射一条指令
 - 流水线的有些功能段还可以进一步细分，例如：ID功能段可以再细分为译码、读第一操作数和读第二操作数三个流水段。也有些功能段不能再细分，如WB功能段一般不再细分。

4.4.4 超流水线处理机

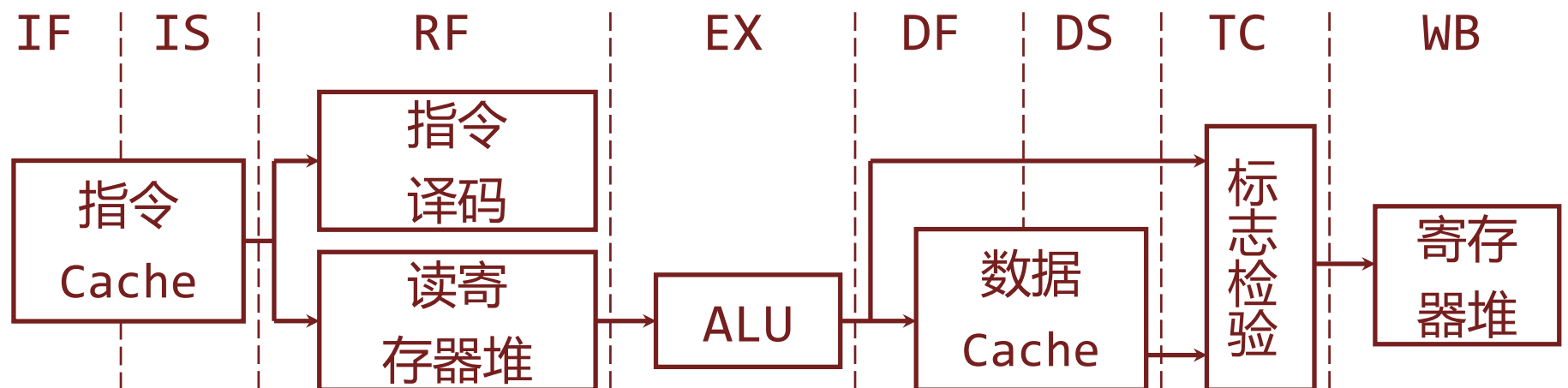
- ◆ 每个时钟周期分时**发送3条指令的超流水线**



典型处理机结构

- ◆ 典型的超流水线处理器：SGI公司的MIPS系列R4000
 - MIPS R4000处理机每个时钟周期包含**两个流水段**，是一种标准的超流水线处理机结构。
 - 有8个流水段
 - 分为指令Cache和数据Cache，容量各8KB，每个时钟周期可以访问Cache两次。
 - 在一个时钟周期内可以从指令Cache中读出两条指令，从数据Cache中读出或写入两个数据。
 - 主要运算部件有整数部件和浮点部件

MIPS R4000处理机的流水线操作



IF：取第一条指令

RF：读寄存器堆，指令译码

DF：取第一个数据

TC：数据标志校验

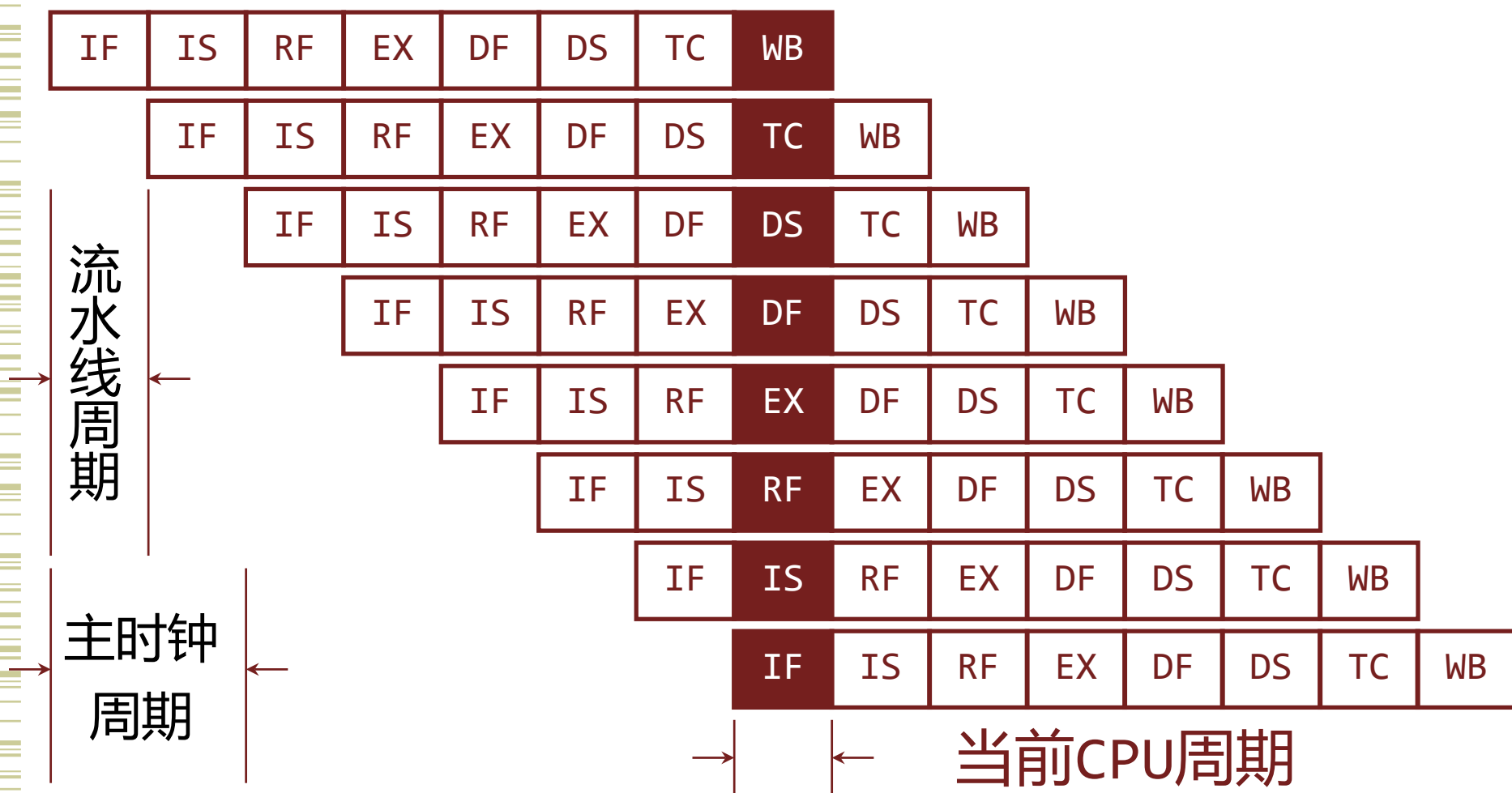
IS：取第二条指令

EX：执行指令

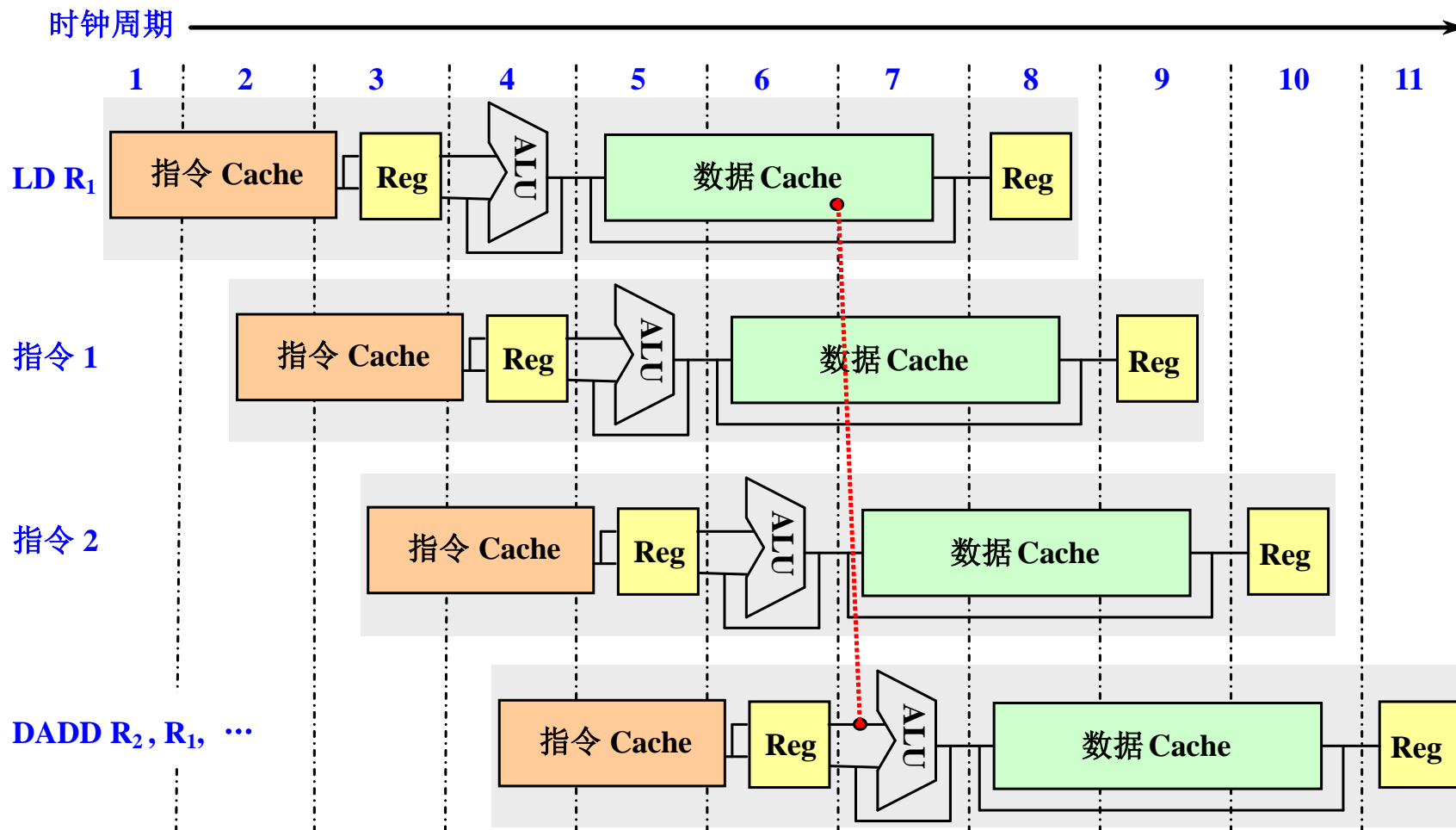
DS：取第二个数据

WB：写回结果

MIPS R4000正常指令流水线工作时序



载入延迟为两个时钟周期



超流水线处理机性能

- ◆ 指令级并行度为(1, n)的超流水线处理机，执行N条指令所需要的时间为：

$$T(1, n) = \left(k + \frac{N-1}{n} \right) \Delta t$$

- ◆ 超流水线处理机相对于单流水线普通标量处理机的加速比为：

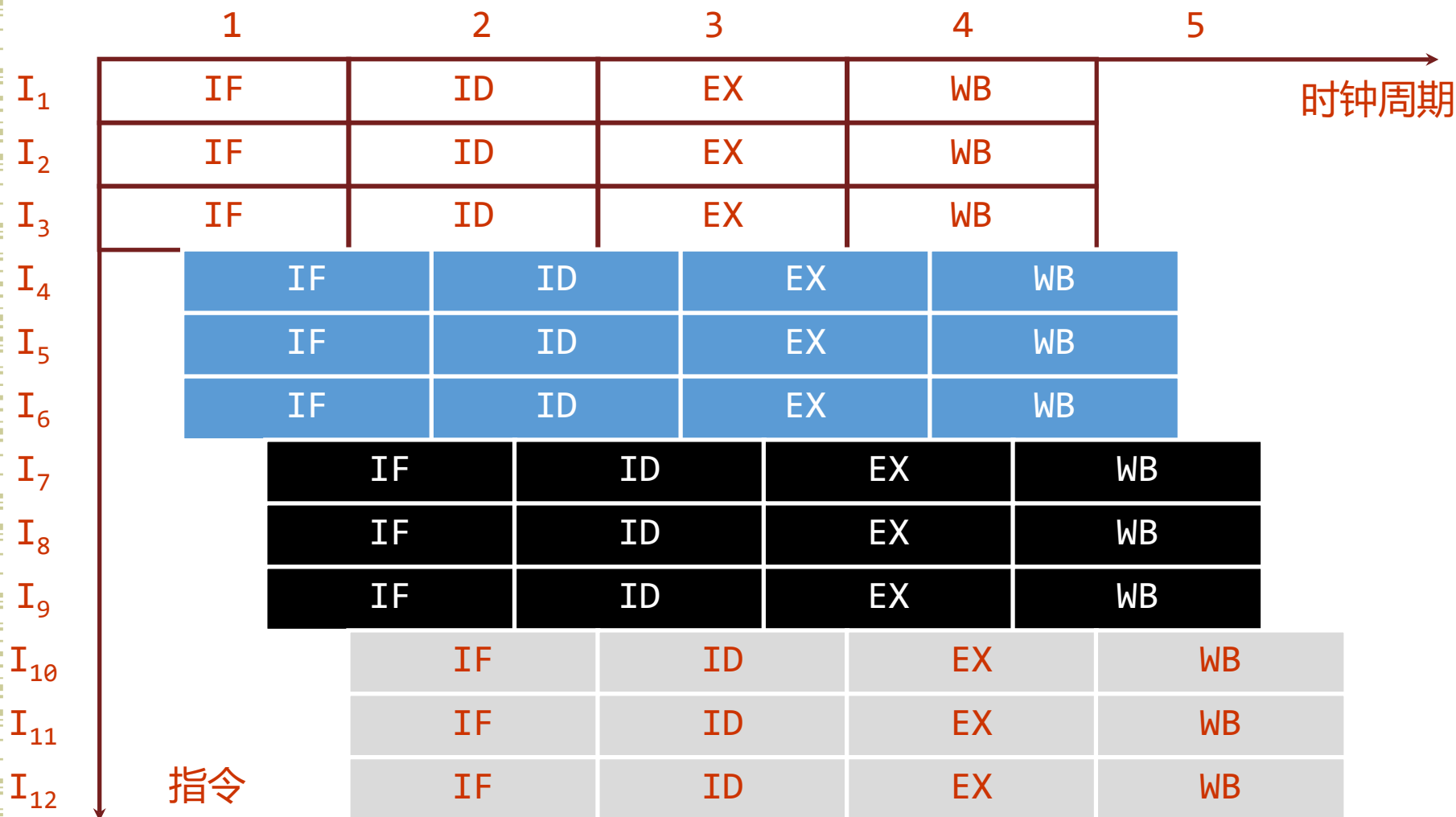
$$\begin{aligned} S(1, n) &= \frac{T(1, 1)}{T(1, n)} = \frac{(k + N - 1) \Delta t}{\left(k + \frac{N-1}{n} \right) \Delta t} \\ &= \frac{n(k + N - 1)}{nk + N - 1} \end{aligned}$$

- ◆ 超流水线处理机的加速比的最大值为： $S(1, n)_{\text{MAX}} = n$

4.4.4 超标量超流水线处理机

- ◆ 技术要点：把超标量与超流水线技术结合在一起，就成为**超标量超流水线处理机**
- ◆ 指令执行时序
 - 超标量超流水线处理机在一个时钟周期内分时发射指令 n 次，每次同时发射指令 m 条，每个时钟周期总共发射指令 $m \times n$ 条。

每时钟周期发射3次, 每次3条指令



超标量超流水线处理机性能

- ◆ 指令级并行度为 (m, n) 的超标量超流水线处理机，连续执行 N 条指令所需要的时间为：

$$T(m, n) = \left(k + \frac{N - m}{m \cdot n}\right) \Delta t$$

- ◆ 超标量超流水线处理机相对于单流水线标量处理机的加速比为：

$$\begin{aligned} S(m, n) &= \frac{S(1, 1)}{S(m, n)} \\ &= \frac{(k + N - 1) \Delta t}{\left(k + \frac{N - m}{mn}\right) \Delta t} \\ &= \frac{m \cdot n \cdot (k + N - 1)}{m \cdot n \cdot k + N - m} \end{aligned}$$

- ◆ 在理想情况下，超标量超流水线处理机加速比的最大值为：

$$S(m, n)_{\text{MAX}} = m * n$$

四种不同处理机性能比较

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
流水线周期	1个	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	1	$1/n$	$1/n$
指令并行度 ILP	1	m	n	$m*n$

4.5 循环展开和指令调度

◆ 4.5.1 循环展开和指令调度的基本方法

- 1、充分开发指令之间存在的并行性，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、增加指令间并行性最简单和最常用的方法
 - 开发循环级并行性——循环的**不同迭代之间存在的并行性**
 - 在把循环展开后，通过重命名和指令调度来开发更多的并行性。
- 3、编译器完成这类指令调度的能力受限于两个特性：
 - 程序固有的指令级并行性；
 - **流水线功能部件的执行延迟。**

4.5.1 循环展开和指令调度的基本方法

- ◆ 本节中，我们使用的浮点流水线延迟为：

产生结果的指令	使用结果的指令	延迟（时钟周期数）
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

- ◆ 假设采用第3章的5段整数流水线：
 - 分支的延迟：1个时钟周期。
 - 整数load指令的延迟：1个时钟周期。
 - 整数运算部件是全流水或者重复设置了足够的份数。

指令调度举例

- ◆ 例4.6 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，分析其一次循环所需的执行时间。

```
for (i=1; i<=1000; i++)  
    x[i] = x[i] + s;
```

- ◆ 解：把该程序翻译成MIPS汇编语言代码：

```
Loop: L.D      F0, 0(R1)      // 取一个向量元数放入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    // 指针减去8  
      BNE      R1, R2, Loop   // 继续
```

- ◆ 其中：

- R1：指向向量中的当前元素，初值为最高端(第1个)元素的地址
- 8(R2)：指向最后一个元素。
- 浮点寄存器F2：用于保存常数s。

指令调度举例

- ◆ 不进行指令调度的情况下，程序的实际执行情况：

标号	指令	时钟周期
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

- ◆ 每个元素的操作需要10个时钟周期，
- ◆ 其中5个是空转周期。

指令调度举例

- ◆ 指令调度以后，程序的执行情况如下：
 - 把DADDIU指令调度到了L.D指令和ADD.D指令之间的“空转”拍
 - 把S.D指令放到了分支指令的延迟槽中
 - 对存储器地址偏移量进行调整

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

标号	指令
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)

指令调度举例

- ◆ 一个元素的操作时间从10个时钟周期减少到6个，其中5个周期是有指令执行的，1个为空转周期
- ◆ 例子中的问题及解决方案
 - 只有L.D、ADD.D和S.D这3条指令是有效操作，占用3个时钟周期。
 - 而DADDIU、空转和BNE这3个时钟周期都是附加的循环控制开销。
 - 要减少这种附加的循环控制开销在程序执行时间中所占的比率，可以使用**循环展开**技术

标号	指令		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

循环展开举例

◆ 循环展开技术：

- 把循环体代码复制多次并按顺序排列，调整循环结束条件，为指令调度带来更大的空间

- ◆ 例4.7 将上述例子中的循环展开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**，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。

- ◆ 解：展开后迭代次数为8，且无需在循环体后面增加补偿代码

- 分配寄存器
- 不重复使用寄存器

寄存器	用途
F2	保存常数
F0、F4	展开后的第1个循环体
F6、F8	展开后的第2个循环体
F10、F12	展开后的第3个循环体
F14、F16	展开后的第4个循环体

循环展开举例——没有调度的代码

标号	指令	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

标号	指令	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

循环展开举例—没有调度的代码

◆ 结果分析：

- 这个循环每遍共使用了28个时钟周期。
- 有4个循环体，完成4个元素的操作。
- 平均每个元素使用 $28/4=7$ 个时钟周期
- 原始循环的每个元素需要10个时钟周期。
- 节省的时间：从减少循环控制的开销中获得的。
- 在整个展开后的循环中，实际指令只有14条，其他14个周期都是空转。

◆ 效率并不高

循环展开举例—优化调度的代码

- ◆ 对指令序列进行优化调度，以减少空转周期
- ◆ 结果分析
 - 没有数据相关引起的空转等待。
 - 整个循环仅仅使用了14个时钟周期。
 - 平均每个元素的操作使用 $14/4=3.5$ 个时钟周期
 - 通过循环展开、寄存器重命名和指令调度，可以有效地开发出指令级并行。

标号	指令		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	11
	S.D	F12, 16(R1)	12
	BNE	R1, R2, Loop	13
	S.D	F16, 8(R1)	14

循环展开和指令调度注意事项

- ◆ **保证正确性**。在循环展开和调度过程中尤其要注意两个地方的正确性：**循环控制**，**操作数偏移量的修改**。
- ◆ **注意有效性**。只有能够找到不同循环体之间的无关性，才能有效地使用循环展开。
- ◆ **使用不同的寄存器**。否则可能导致新的冲突
- ◆ **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**
 - 例如：对于load指令和store指令，如果它们在不同的循环迭代中访问的存储器地址是不同的，它们就是相互独立的，可以相互对调。
- ◆ **注意新的相关性**。由于原循环不同次的迭代在展开后都到了同一次循环体中，因此可能带来新的相关性。

4.5.2 静态超标量处理机中的循环展开

- ◆ 超标量处理器如何进行循环展开和指令调度？
- ◆ 例4.8 下面是前面使用的循环程序段，对其进行循环展开，并在超标量流水线上进行调度。

Loop: L.D	F0, 0(R1)	// 取一个向量元数放入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	// 指针减去8
BNE	R1, R2, Loop	// 继续

静态超标量处理机中的循环展开举例

- ◆ 要达到无延迟的指令调度，须将循环展开5遍并调度
- ◆ 可使S.D指令不会停顿
 - 使用ADD.D的结果

标号	指令
Loop:	L.D F0, 0(R1)
	L.D F6, -8(R1)
	L.D F10, -16(R1)
	L.D F14, -24(R1)
	L.D F18, -32(R1)

标号	指令
	ADD.D F4, F0, F2
	ADD.D F8, F6, F2
	ADD.D F12, F10, F2
	ADD.D F16, F14, F2
	ADD.D F20, F18, F2
	S.D F4, 0(R1)
	S.D F8, -8(R1)
	S.D F12, -16(R1)
	DADDIU R1, R1, #-40
	S.D F16, 16(R1)
	BNE R1, R2, Loop
	S.D F20, 8(R1)

针对超标量进行调度后的指令序列

	整数指令		浮点指令	时钟
Loop:	L.D	F0, 0(R1)		1
	L.D	F6, -8(R1)		2
	L.D	F10, -16(R1)	ADD.D F4, F0, F2	3
	L.D	F14, -24(R1)	ADD.D F8, F6, F2	4
	L.D	F18, -32(R1)	ADD.D F12, F10, F2	5
	S.D	F4, 0(R1)	ADD.D F16, F14, F2	6
	S.D	F8, -8(R1)	ADD.D F20, F18, F2	7
	S.D	F12, -16(R1)		8
	DADDIU	R1, R1, #-40		9
	S.D	F16, 16(R1)		10
	BNE	R1, R2, Loop		11
	S.D	F20, 8(R1)		12

4.5.2 静态超标量处理机中的循环展开

- ◆ 每次循环需12个时钟周期，即计算每个结果需要2.4个时钟周期。（每次循环计算5个结果）
- ◆ 在普通的MIPS流水线上，没有调度的代码迭代一次为10个时钟周期，调度后为6个时钟周期，展开4次并调度后每个迭代为3.5个时钟周期。
- ◆ 与之相比，超标量流水线的性能提高分别为：4.2倍、2.5倍、1.5倍
- ◆ 在这个例子中可以看到，超标量MIPS流水线的性能主要受限于：**整数计算和浮点计算之间的平衡问题**
- ◆ 本例中没有足够的浮点指令来使两路流水线都达到饱和。