

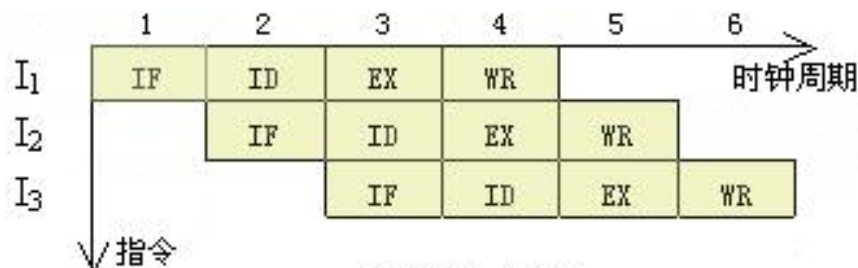
# 计算机体系结构

韩银和

部分内容摘自胡伟武、汪文祥的PPT

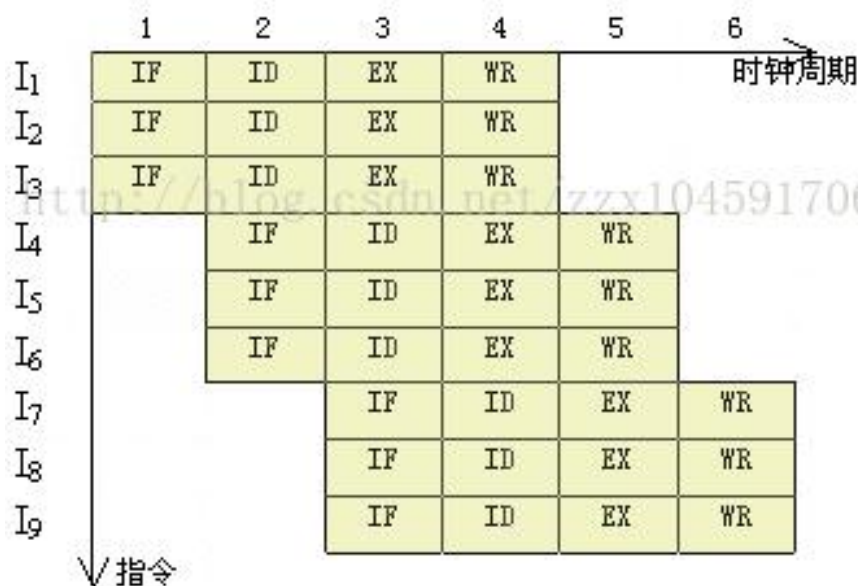
# 单发射和多发射设计及带来的问题

单发射



(a) 单发射时空图

多发射



(b) 多发射时空图

IF: 取指令, ID: 指令译码, EX: 执行指令, WR: 写回结果

# 动态流水线技术

- 影响流水线效率的因素
- 指令调度技术
- 动态调度技术
- Tomasulo算法
- 动态流水线的例外处理

# 影响流水线效率的因素

- 影响指令流水线性能的因素
- 运行时间 = 程序指令数 \* CPI （每条指令时钟周期数）
- **Pipeline CPI=Ideal pipeline CPI + Structural stalls + RAW stalls + WAR stalls + WAW stalls + Control stalls**

技术	作用
循环展开	<b>Control stalls</b>
编译相关性分析	<b>Ideal CPI and data stalls</b>
软件流水技术	<b>Ideal CPI and data stalls</b>
简单流水线	<b>RAW stalls</b>
记分板动态调度	<b>RAW stalls</b>
寄存器重命名	<b>WAR and WAW</b>
动态转移预测	<b>Control stalls</b>
多发射	<b>Ideal CPI</b>
猜测执行	<b>All data and control stalls</b>
非阻塞访存	<b>RAW stall involving memory</b>

# 程序的相关性

- 数据相关（真相关）：导致RAW
- 名字相关：会导致WAW和WAR
- 控制相关：条件转移
- 程序的相关性容易引起流水线堵塞，可以通过软件和硬件的方法

## 避免堵塞或降低堵塞的影响

- 编译调度：如循环展开
- 乱序执行：需要等待的指令不影响其他指令

# 数据相关

- 定义：指令j数据相关于指令i
  - 指令j使用了指令i产生的结果，或
  - 指令j数据相关于指令k，指令k数据相关于指令i
- 数据相关的指令不能并行执行
- 寄存器的数据相关比较容易判断
- 存储器的数据相关不容易判断
  - $100(R4) = 20(R6)?$
  - 对不同循环体， $20(R6) = 20(R6)?$

1 Loop: LD F0, 0(R1)

2 ADD F4, F0, F2  
D

3 SUBI R1, R1, 8

4 BNEZ R1, Loop ;delayed branch

5 SD 8(R1), F4 ;altered when move past SUBI

# 名字相关

- 两条指令使用相同名字（寄存器或存储器），但不交换数据
  - 逆相关（Antidependence）：指令j写指令i所读的存储单元且i先执行，逆相关会导致流水线WAR相关
  - 输出相关（Output Dependence）：指令j与指令i写同一个单元且i先执行，逆相关会导致流水线WAW相关
- 寄存器的名字相关可以通过寄存器重命名（Register Renaming）解决，存储单元的重命名比较困难
  - $100(R4) = 20(R6)?$
  - 对不同循环体， $20(R6) = 20(R6)?$
  - 在前述例子中，编译器必须知道 $0(R1) \neq -8(R1) \neq -16(R1) \neq -24(R1)$
- RISC技术极大地简化了指令之间的相关性

# 单发射和多发射的CPI

- 在单发射处理机中，取指令部件和指令译码部件只各设置一套，而操作部件可以只设置一个多功能操作部件，也可以设置多个独立的操作部件。例如，定点算术逻辑部件ALU、取数存数部件LSU、浮点加法部件FAD、乘法部件MDU等。单发射处理机在指令一级通常采用流水线结构；而在操作部件中，有的机器采用流水线结构，也有的机器不采用流水线结构。
- 单发射处理机的设计目标是每个时钟周期平均执行一条指令，即它的指令级并行度ILP的期望值1。
  - 实际上，它就是一台有k段流水线的普通标量处理机。由于数据相关、条件转移和资源冲突等原因，实际的ILP不可能达到1。
  - 通过优化编译器对指令序列进行重组（reorganizer），以及采用软件与硬件相结合的方法处理数据相关、条件转移和资源冲突等，可以使ILP接近于1。但是，单发射处理机的ILP不可能大于1。

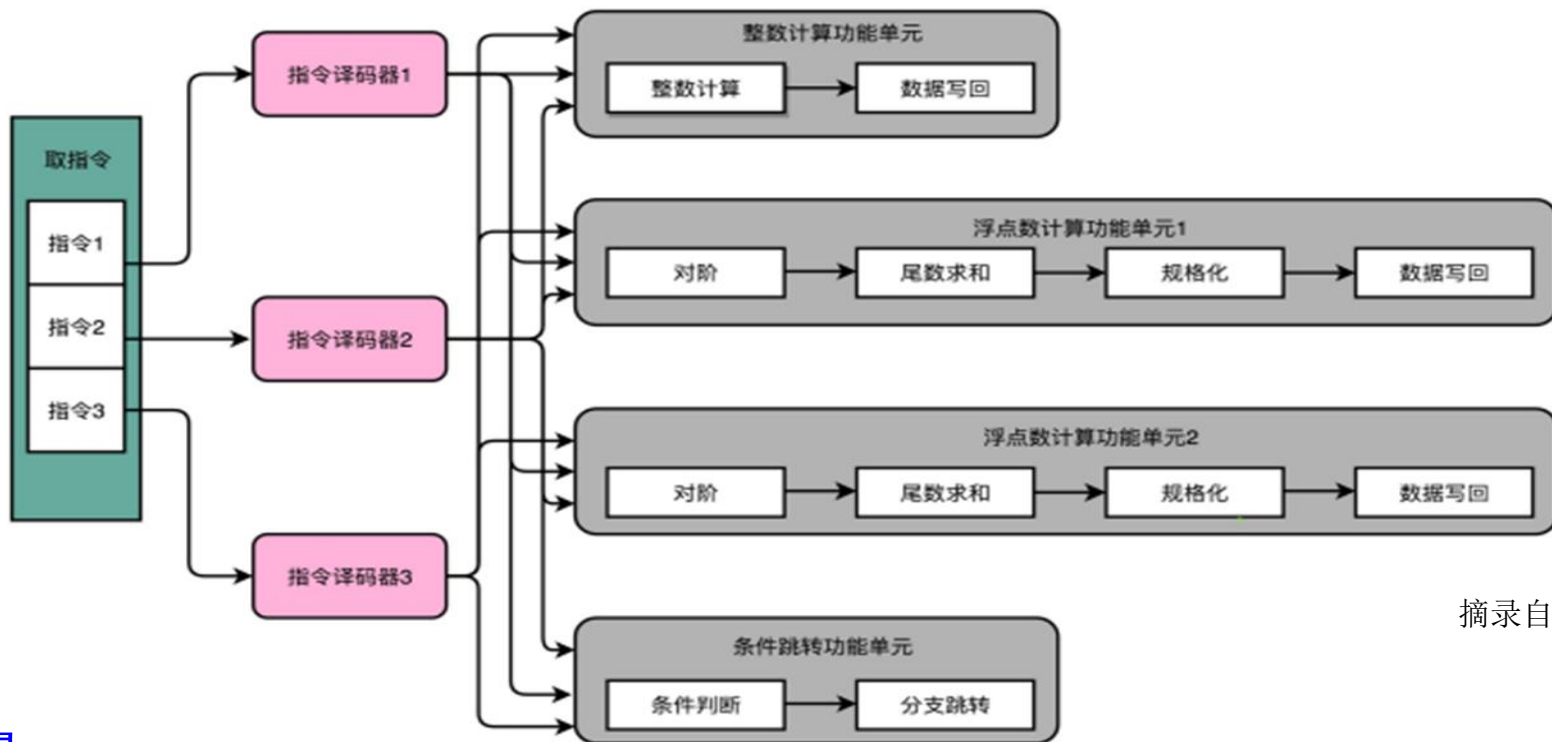


# 多发射的原理

一次性读取多条指令

分发到多个指令译码器并行译码

多个功能单元并行处理，不同功能单元流水线长度不同



摘录自CSDN

## 超标量:

- 超标量是一条流水线中，存在多个执行部件，可以在上一条指令没有执行完成时，执行下一条指令。即乱序执行。可以针对没有前后顺序关系的指令进行并行执行；
- 超标量是对程序员透明的，是指令级别的并行；
- 超标量是否意味着需要多条流水线？（好像不需要，如果有，好像最好？）超
- 标量处理器内部只有一个指令指针，一套控制逻辑，也就是说，对于外界而言，在同一时刻，只能执行一段指令序列。

# 动态调度技术

# 静态流水线的问题

- 在译码阶段把指令“隔开”来解决相关
  - 只要有一条指令停止，后面指令就不能前进，象是一种译码部件的结构相关
- 对编译要求高，最好是编译把相关指令隔开
  - 有些信息在译码时难以确定，如是否发生例外、访存操作需要多少周期等

DIVD f0,f2,f4

ADDD f10,f0,f8

SUBD f12,f8,f14

# 动态调度思想

DIVD f0,f2,f4

ADDD f10,f0,f8

SUBD f8,f8,f14

- 基本思想

- 前面指令的等待不影响后面指令继续前进
- 把相关的解决尽量往后拖延（记得Forward技术吗？）

- 把译码分成两个阶段：发射和读操作数

- 发射：指令译码，检查结构相关
- 读操作数：如果操作数准备好就读数，否则等待（在哪儿等？）
- 当一条指令在读操作数阶段等待时，后面指令的发射可以继续进行

- 乱序执行的基本做法

- 指令进入是有序的
- 执行可以乱序，只要没有相关就可执行，多条指令同时执行
- 结束也是有序的（怎么把乱序变成有序？）

- 与静态调度相比


- 有些相关编译无法检测、编译器更加简单、程序性能对机器依赖少

# 解决WAW和WAR的办法

- 假设只考虑RAW相关，可能发生如下执行次序

- DIV发射，F1，F2都准备好
- MUL1发射，F6没准备好，所以没有读数据
- ADD发射，F3，F4都准备好
- MUL2发射，F8没有准备好，所以没有读数据
- ADD完成，F0写回
- F6准备好，MUL1读数据
- DIV完成，F0写回
- F8准备好，MUL2读数据

DIV F0, F1, F2  
MUL F5, F0, F6  
ADD F0, F3, F4  
MUL F7, F0, F8



The diagram illustrates the data flow of the register F0. A blue arrow originates from the red 'F0' in the 'DIV F0, F1, F2' instruction and points to the red 'F0' in the 'MUL F5, F0, F6' instruction. Another blue arrow originates from the red 'F0' in the 'DIV F0, F1, F2' instruction and points to the red 'F0' in the 'ADD F0, F3, F4' instruction. This visualizes the RAW (Read After Write) dependencies where the value of F0 written by the DIV instruction is read by both the MUL and ADD instructions.

# 解决WAW和WAR的办法

- 为了避免MUL1读回ADD写的F0值，MUL2读回DIV写的F0值
  - 最简单的做法是在MUL读F0之前ADD不能写回，在DIV写回之前ADD不能写回，正是记分板的办法
  - 在上述方法中，F0成为瓶颈，它必须保证DIV写、MUL1读、ADD写、MUL2读的串行次序，这是问题的本质所在
  - 真正相关：MUL1用DIV的结果，MUL2用ADD的结果，F0最终的结果为ADD的结果
  - MUL1用DIV的结果、MUL2用ADD结果不一定通过F0

DIV F0, F1, F2

MUL F5, F0, F6

ADD F0, F3, F4

MUL F7, F0, F8

# 解决WAW和WAR的办法

- 为了避免MUL1读ADD写的F0值，MUL2读DIV写的F0值
  - 也可以在MUL1的输入端指定只接收DIV的输出值，在MUL2的输入端指定只接收ADD的输出值，相当于DIV 直接把结果写到MUL1的输入端，ADD直接把结果写到MUL2的输入端。（记得Forward技术吗？）
  - 要求：(1)DIV的输出连到MUL1的输入，ADD的输出连接到MUL2的输入；(2)MUL1和MUL2的输入端有寄存器，这些寄存器能够指定接收哪个部件的输出作为自己的值
- 同样，为了避免F0的最终值为DIV所写的值
  - 可以在F0记录它当前接收哪个功能部件所写的值
  - 要求F0有一个标志

DIV F0, F1, F2

MUL F5, F0, F6

ADD F0, F3, F4

MUL F7, F0, F8

# 解决WAW和WAR的办法

- 上述解决办法要求
  - 每个功能部件的输入端有一些寄存器
  - 每个寄存器（包括功能部件输入端的寄存器以及通用寄存器）都记录一个功能部件号，指定它当前接收哪个功能部件的值
  - 每个功能部件的输出接到每个功能部件的输入
- 有了上述功能，WAR和WAW相关不用阻塞
  - 寄存器重命名技术

DIV F0, F1, F2

MUL F5, F0, F6

ADD F0, F3, F4

MUL F7, F0, F8



DIV RenFa, F1, F2

MUL F5, RenFa, F6

ADD (F0, RenFb), F3, F4

MUL F7, RenFb, F8



# Tomasulo算法

# Tomasulo算法

- IBM 360/91中首次使用
  - 1966年, 比CDC 6600晚3年
  - Robert Tomasulo提出
  - 目标:即使在没有特殊编译支持的情况下, 也能取得高性能。
- 其主要思想现代处理器中普遍使用
  - 早期Alpha 21264, HP 8000, MIPS 10000, Pentium II, PowerPC 604, ...
  - 现在Intel、AMD、IBM的几乎所有CPU
- 通过寄存器重命名消除WAR和WAW相关

休息

# Tomasulo算法结构

## 保留站内容

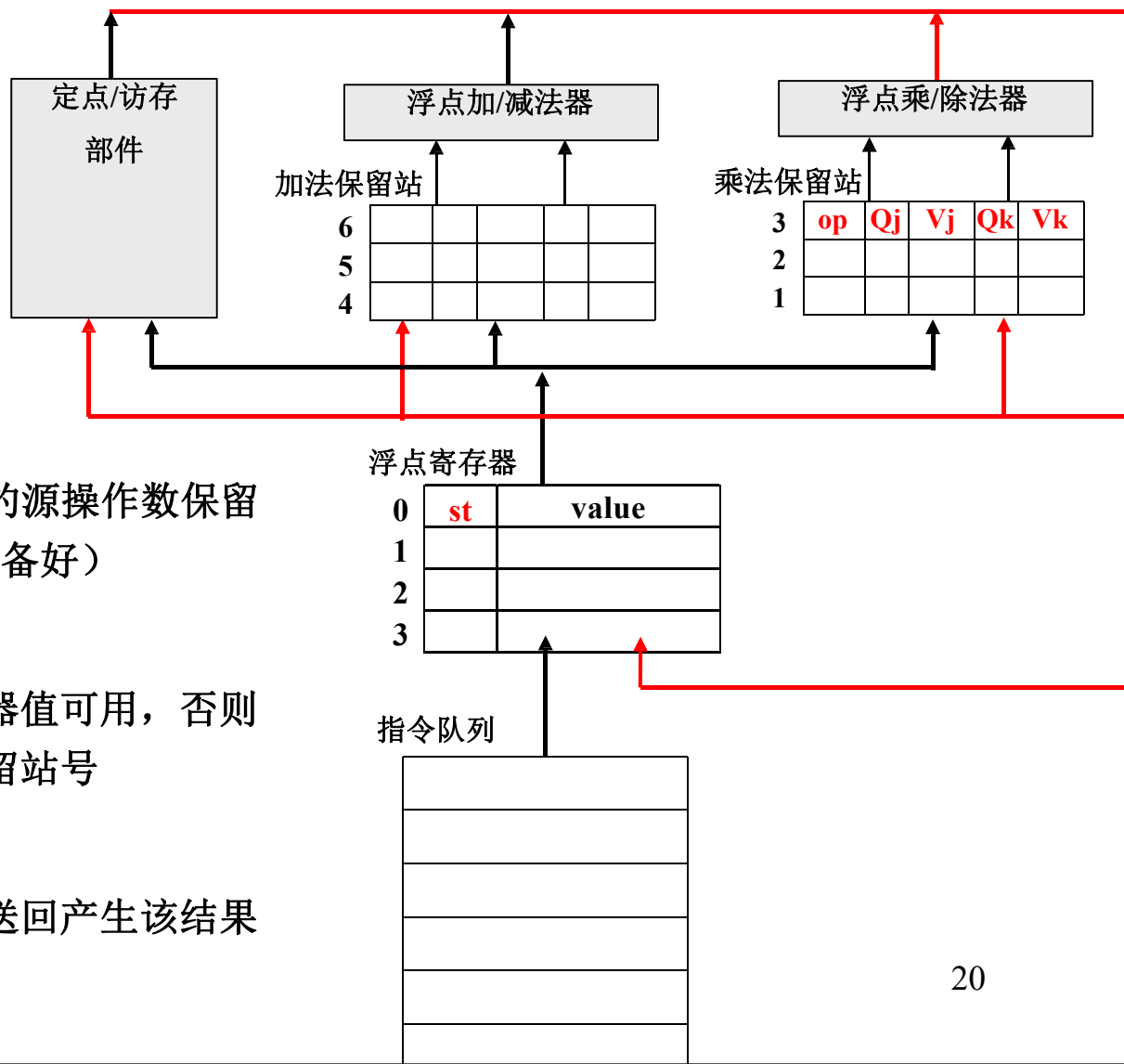
- Busy: 忙位
- Op: 操作码
- $V_j, V_k$ : 源操作数的值
- $Q_j, Q_k$ : 保存没有准备好的源操作数保留站号 (0表示操作数已经准备好)

## 寄存器增加一个域

- 结果状态域: 空表示寄存器值可用, 否则保存产生寄存器结果的保留站号

## 结果总线

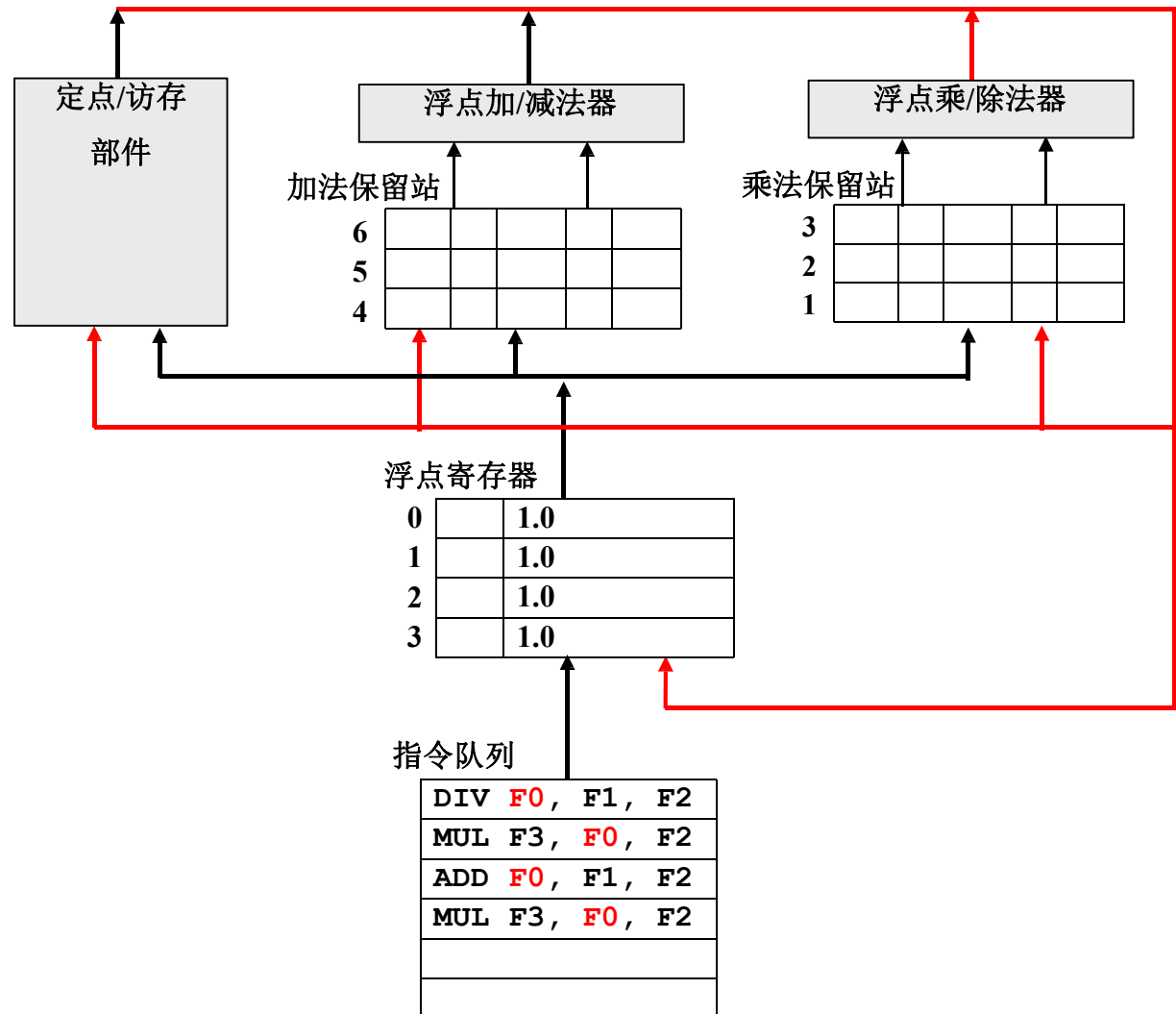
- 除了送回结果值外, 还要送回产生该结果的结果保留站号

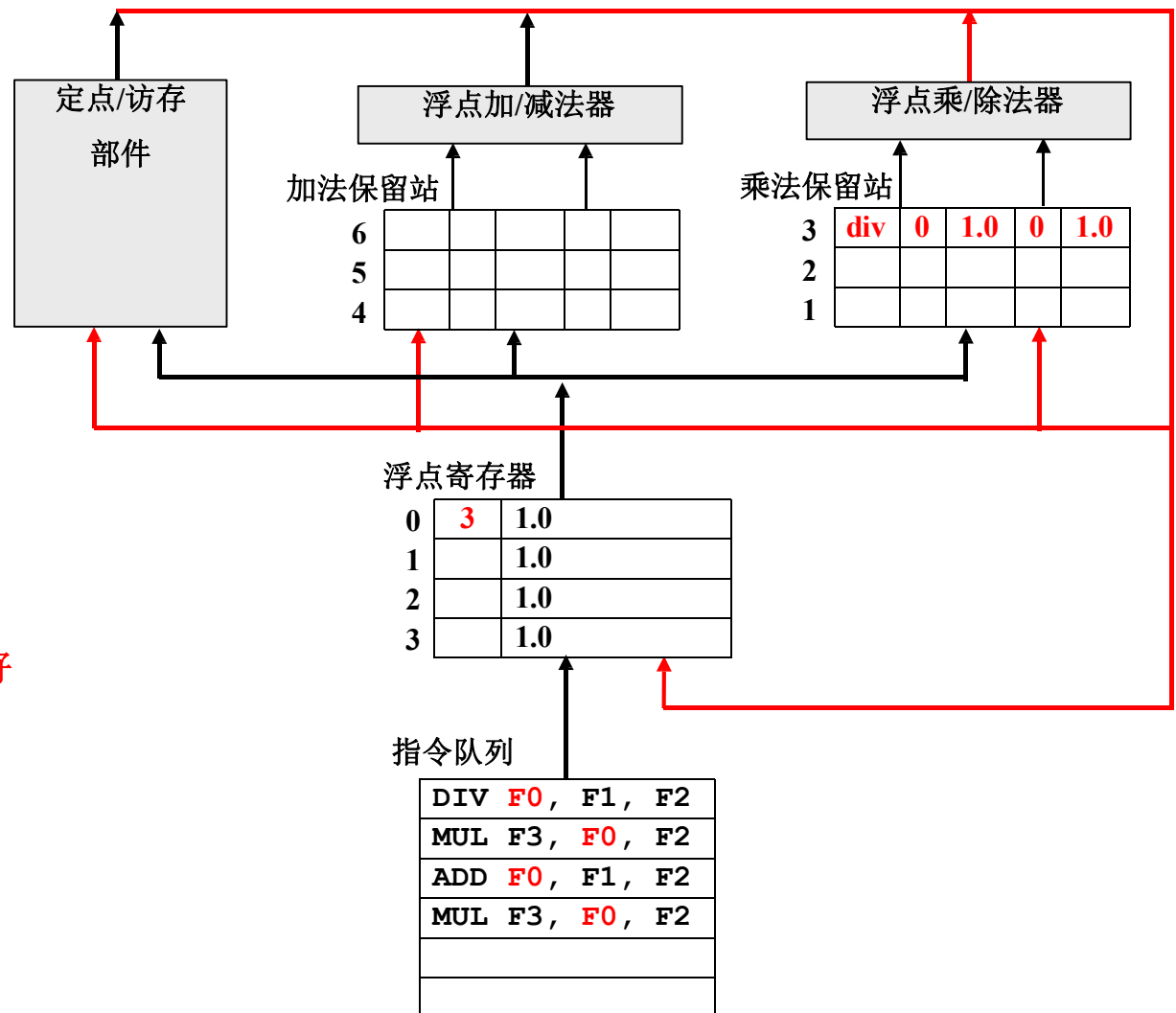


# Tomasulo算法的流水阶段

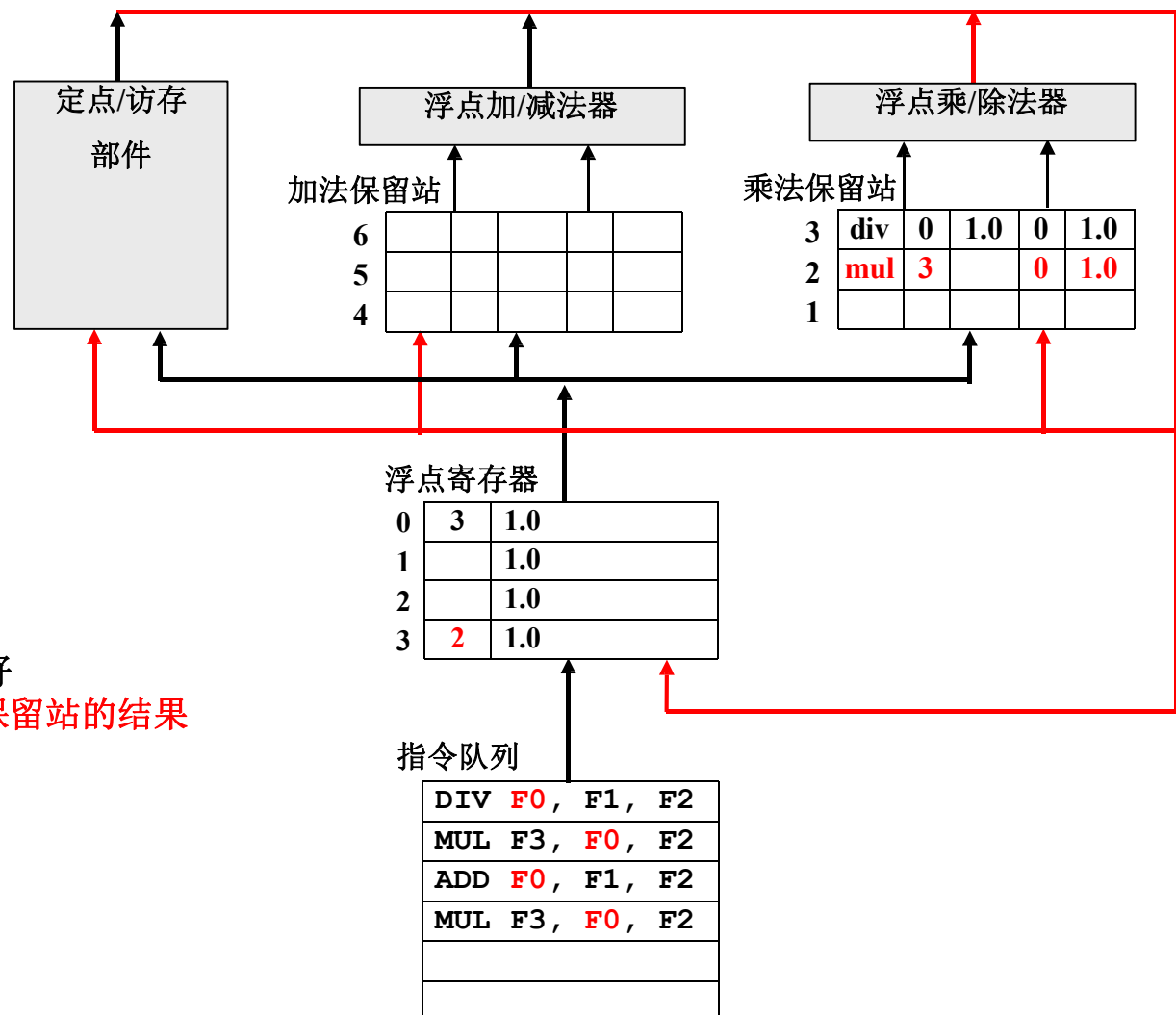
- 发射：把操作队列的指令根据操作类型送到保留站（如果保留站有空），发射过程中读寄存器的值和结果状态域
- 执行：如果所需的操作数都准备好，则执行，否则侦听结果总线并接收结果总线的值。
- 写回：把结果送到结果总线，释放保留站

# 例子



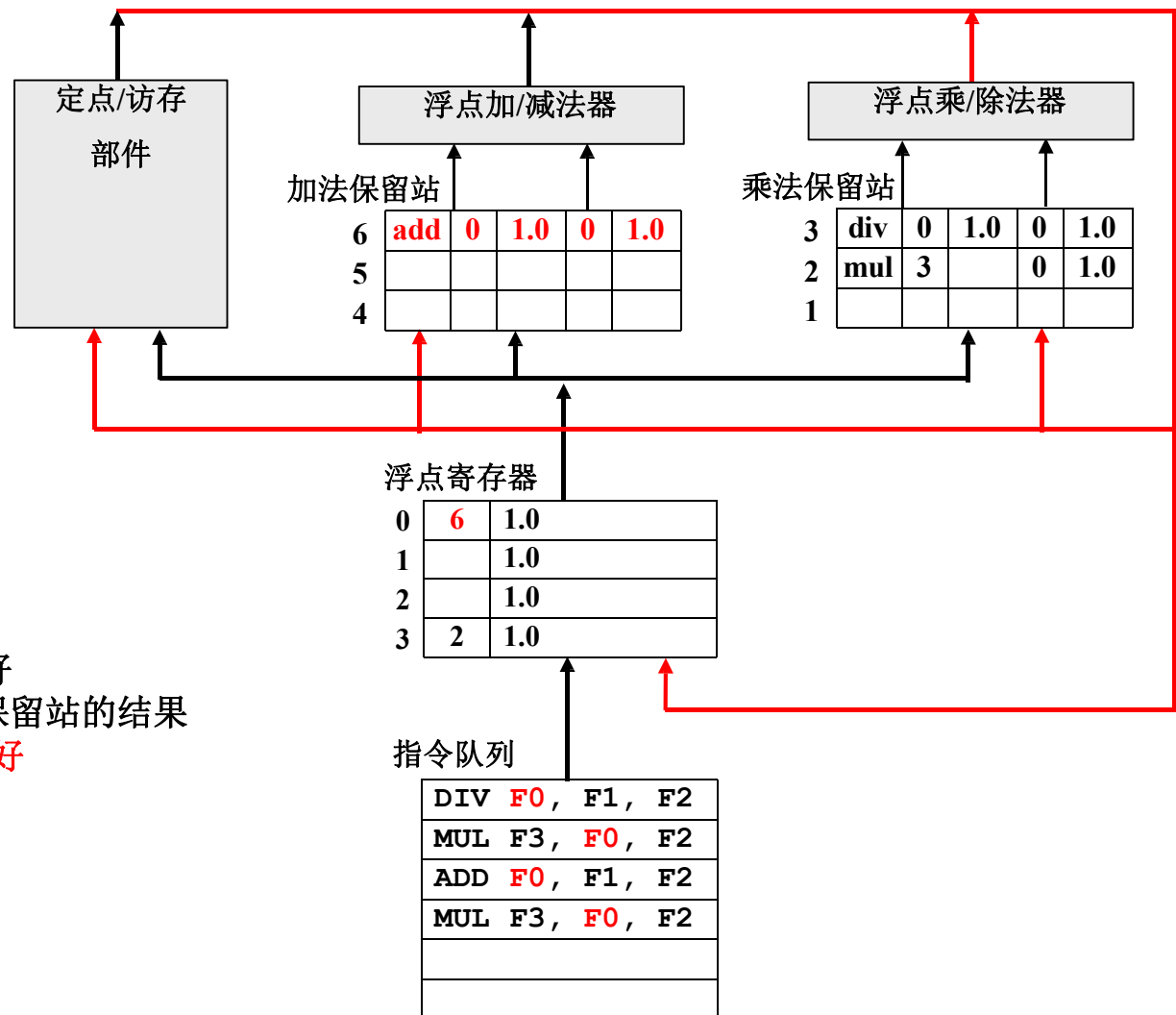


•DIV发射，F1,F2都准备好

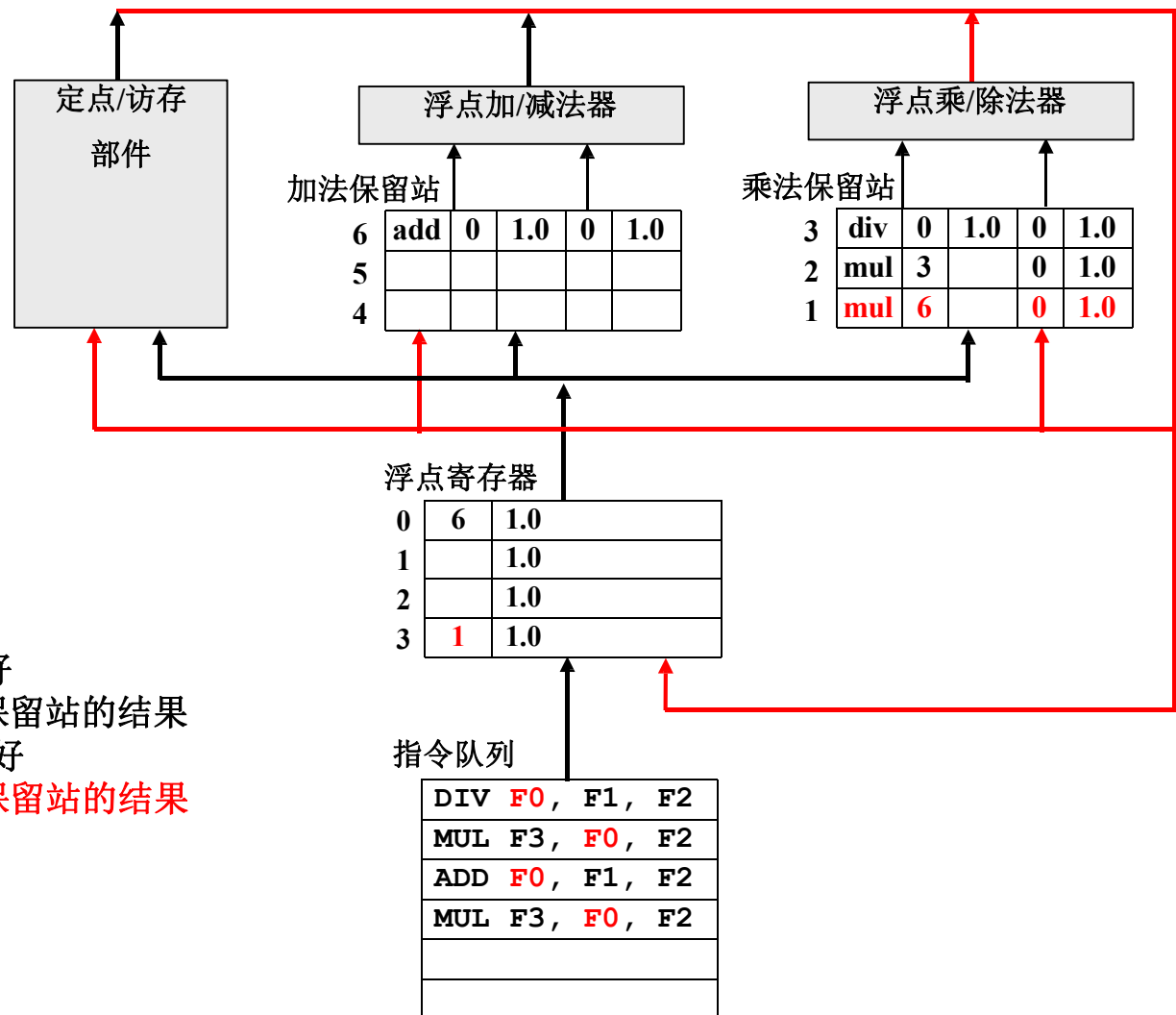


- DIV发射，F1,F2都准备好
- MUL1发射，F0等待3号保留站的结果

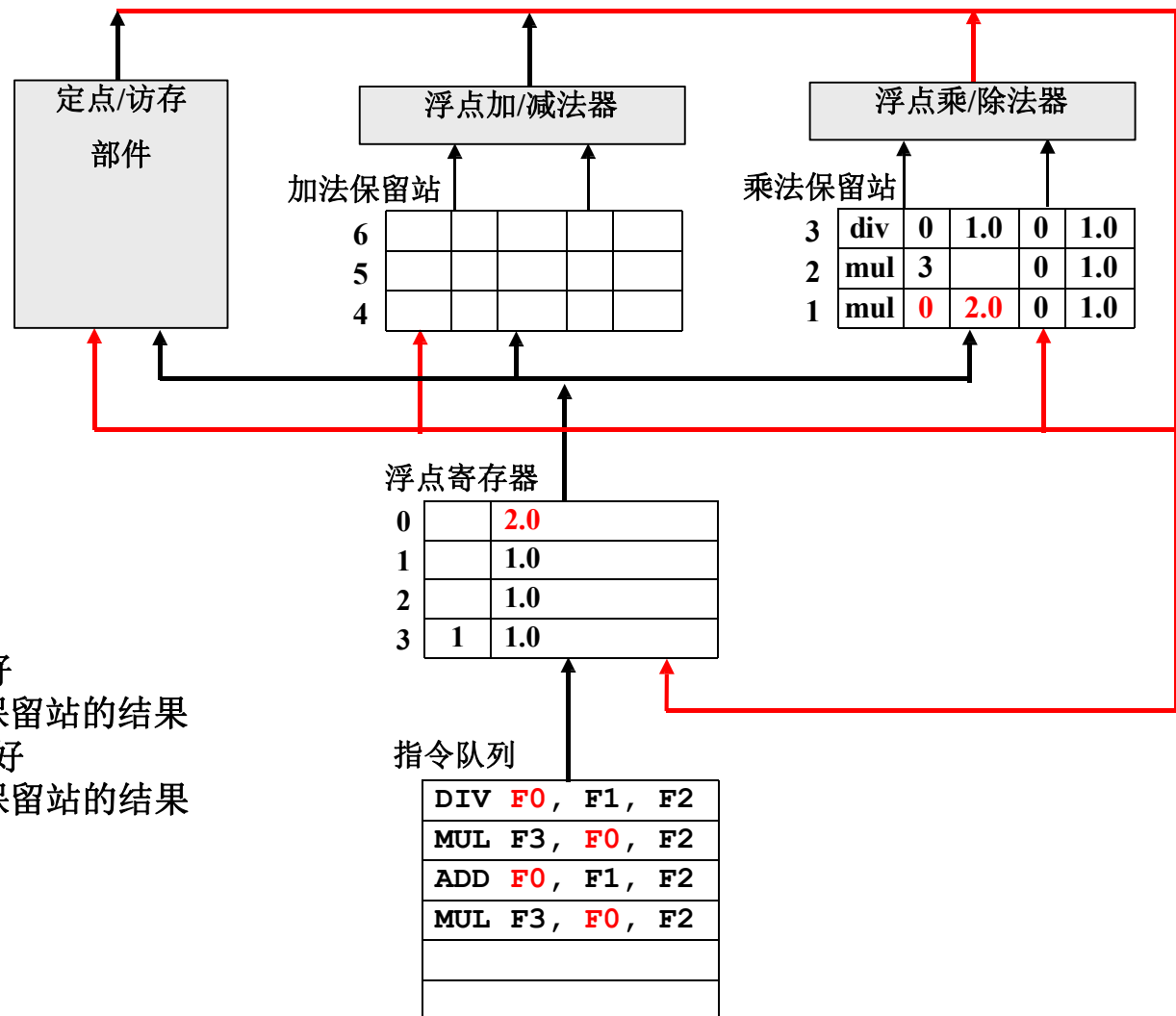




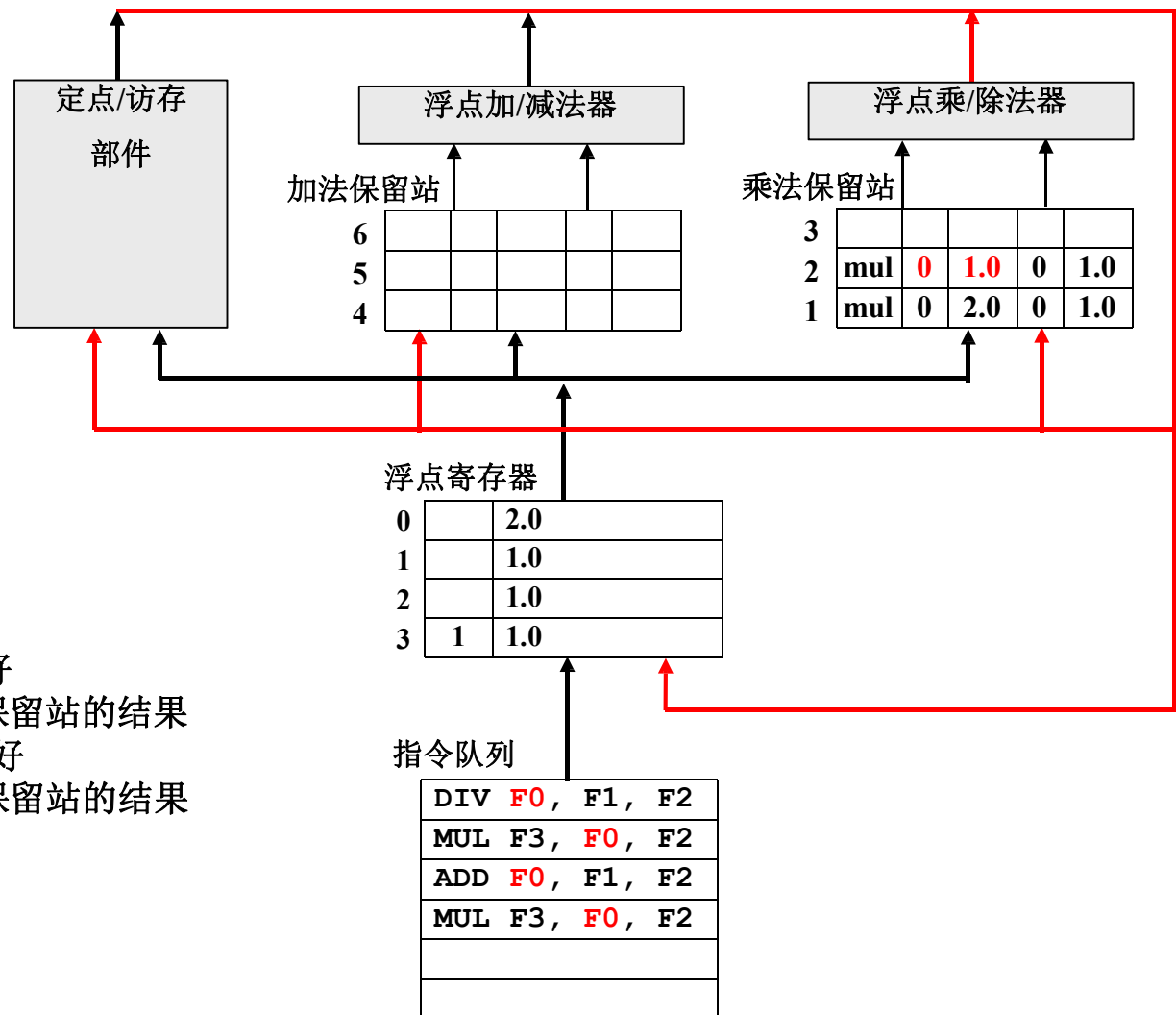
- DIV发射，F1,F2都准备好
- MUL1发射，F0等待3号保留站的结果
- ADD发射，F1,F2都准备好



- DIV发射，F1,F2都准备好
- MUL1发射，F0等待3号保留站的结果
- ADD发射，F1,F2都准备好
- MUL2发射，F0等待6号保留站的结果



- DIV发射，F1,F2都准备好
- MUL1发射，F0等待3号保留站的结果
- ADD发射，F1,F2都准备好
- MUL2发射，F0等待6号保留站的结果
- ADD写回



- DIV发射，F1, F2都准备好
- MUL1发射，F0等待3号保留站的结果
- ADD发射，F1, F2都准备好
- MUL2发射，F0等待6号保留站的结果
- ADD写回
- DIV写回

# Tomasulo算法小结

- 通过动态调度缓解流水线阻塞
  - 例如减少CACHE失效对性能的影响
- 保留站：重命名寄存器+缓存源操作数
  - 避免寄存器成为瓶颈
  - 避免WAW和WAR阻塞
- 缺点
  - 硬件复杂性
  - 结果总线成为瓶颈，多条结果总线增加硬件复杂度
- 在IBM 360/91后被广泛使用
  - 动态调度、寄存器重命名等思想一直被使用：Pentium II; PowerPC 604; MIPS R10000; HP-PA 8000; Alpha 21264

休息

## 动态流水线的例外处理

# 例外（Exception）与流水线

- I/O请求：外部中断
- 指令例外：用户请求中断
  - 系统调用、断点、跟踪调试指令
- 运算部件
  - 整数运算溢出、浮点异常
- 存储管理部件
  - 访存地址不对齐、用户访问系统空间、TLB失效、缺页、存储保护错（写只读页）
- 保留指令错：未实现指令
- 硬件错
- 等等



# 动态流水线的精确例外处理

- 精确例外的要求：在处理例外时，发生例外指令前面的所有指令都执行完，例外指令后面的所有指令还未执行。
- 非精确例外的原因：在乱序执行时，前面的指令发生例外时，后面的指令已经执行完并修改了寄存器或存储单元
  - 在下面的例子中，没有任何相关，ADDF和SUBF指令可以比DIVF先结束。如果在ADDF结束后DIVF发生例外，此时无法恢复例外现场
  - 记分板和Tomasulo算法中都是非精确例外
- 只要保证后面指令修改机器状态时，前面的指令都已不会发生例外即可

DIVF f0,f2,f4

ADDF f10,f10,f8

SUBF f12,f12,f14

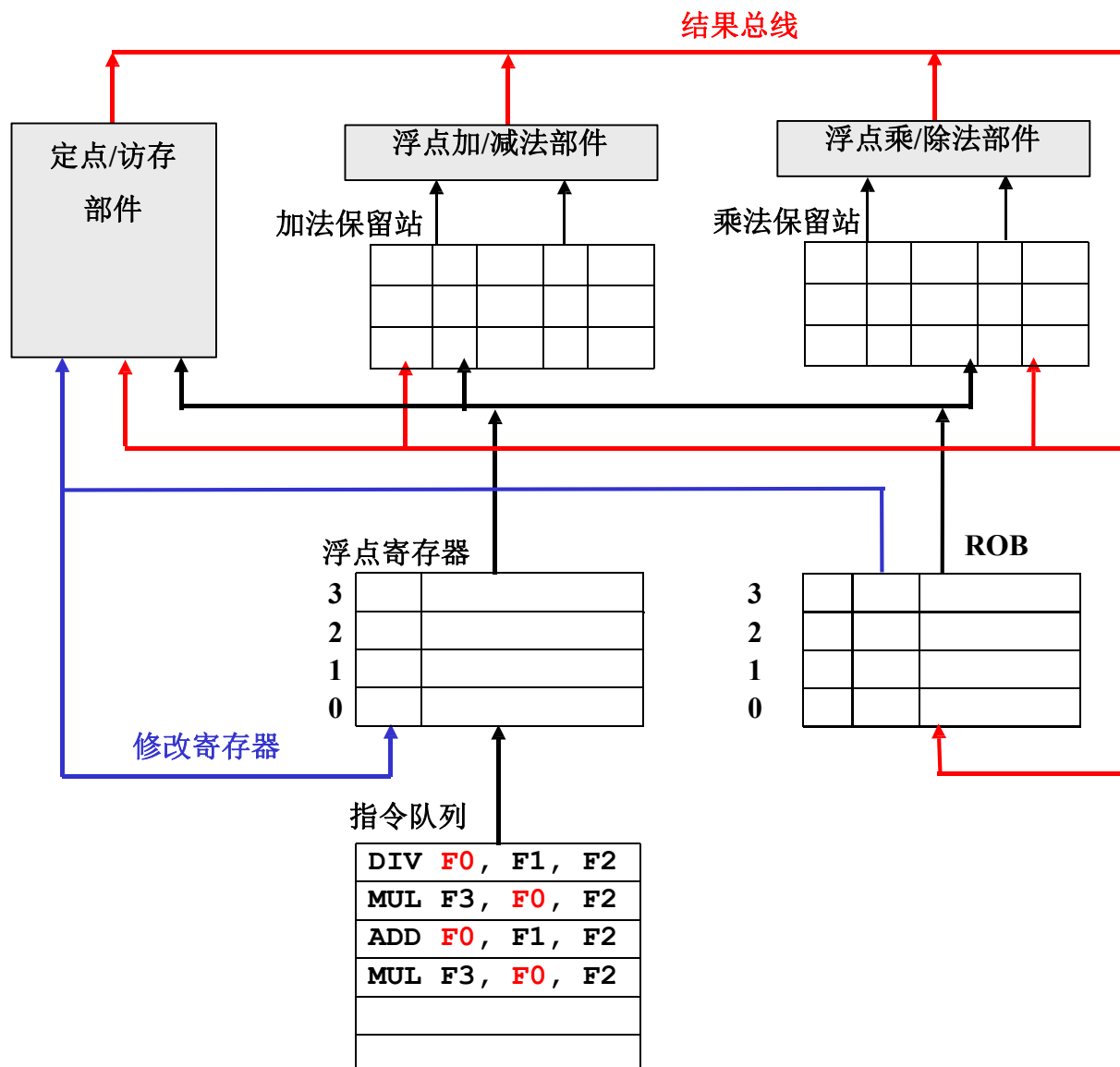
# 硬件支持动态流水线的精确例外处理

- 实现精确例外处理的一个办法是把后面指令对机器状态的修改延迟到前面指令都已经执行完
  - 有些指令在EX阶段也修改机器状态，如运算指令修改结果状态
  - 在执行阶段停止流水线会影响后面的指令执行
- 可以用一些缓冲器来临时保存执行结果，当前面所有指令执行完后，再把保存在缓冲器中的结果写回到寄存器或存储器
  - 在流水线修改机器状态时（在执行或写回阶段）写到缓冲器
  - 增加提交（Commit）阶段，把缓冲器的内容写回到寄存器或存储器
  - 提交阶段只有前面指令都结束后才能进行
  - 有序提交：乱序执行，有序结束
  - 所用的缓冲器通常被称为Reorder Buffer (ROB)
- 在猜测执行中也用上述机制
  - 都是在某些情况不确定的情况下先执行，但留有反悔的余地

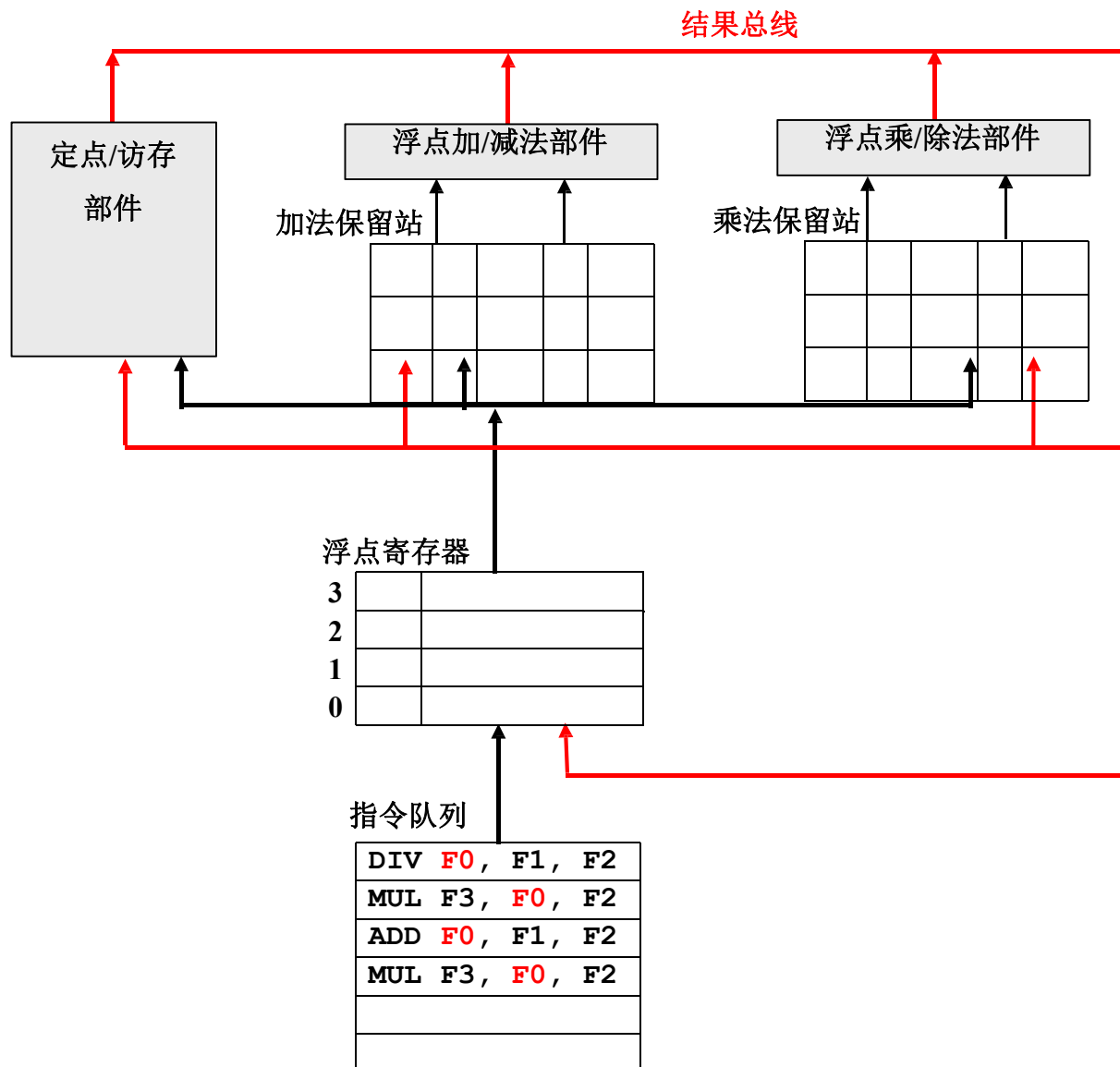
# 指令重排序缓存Reorder Buffer (ROB)

- 内容：目标地址（存数地址或寄存器号）、 值、 操作类型
- 写回时写回到ROB， 因此后面指令有可能从ROB读操作数
- 使用ROB号作为重命名号（原来使用保留站号）， 一条指令的结果寄存器被重命名为其结果ROB号
  - 保留站重命名源寄存器号，
  - ROB重命名结果寄存器号
- 提交时把结果写回寄存器或存储器
- 只要一条指令没有提交， 它就不会对寄存器或存储器的内容进行修改， 在一条指令没有提交之前很容易取消该指令（由于前面指令发生了例外或由于猜测执行不正确）
- ROB可以和Write Buffer合并

- 增加Reorder Buffer的流水线



- 没有Reorder Buffer的流水线



## 例子

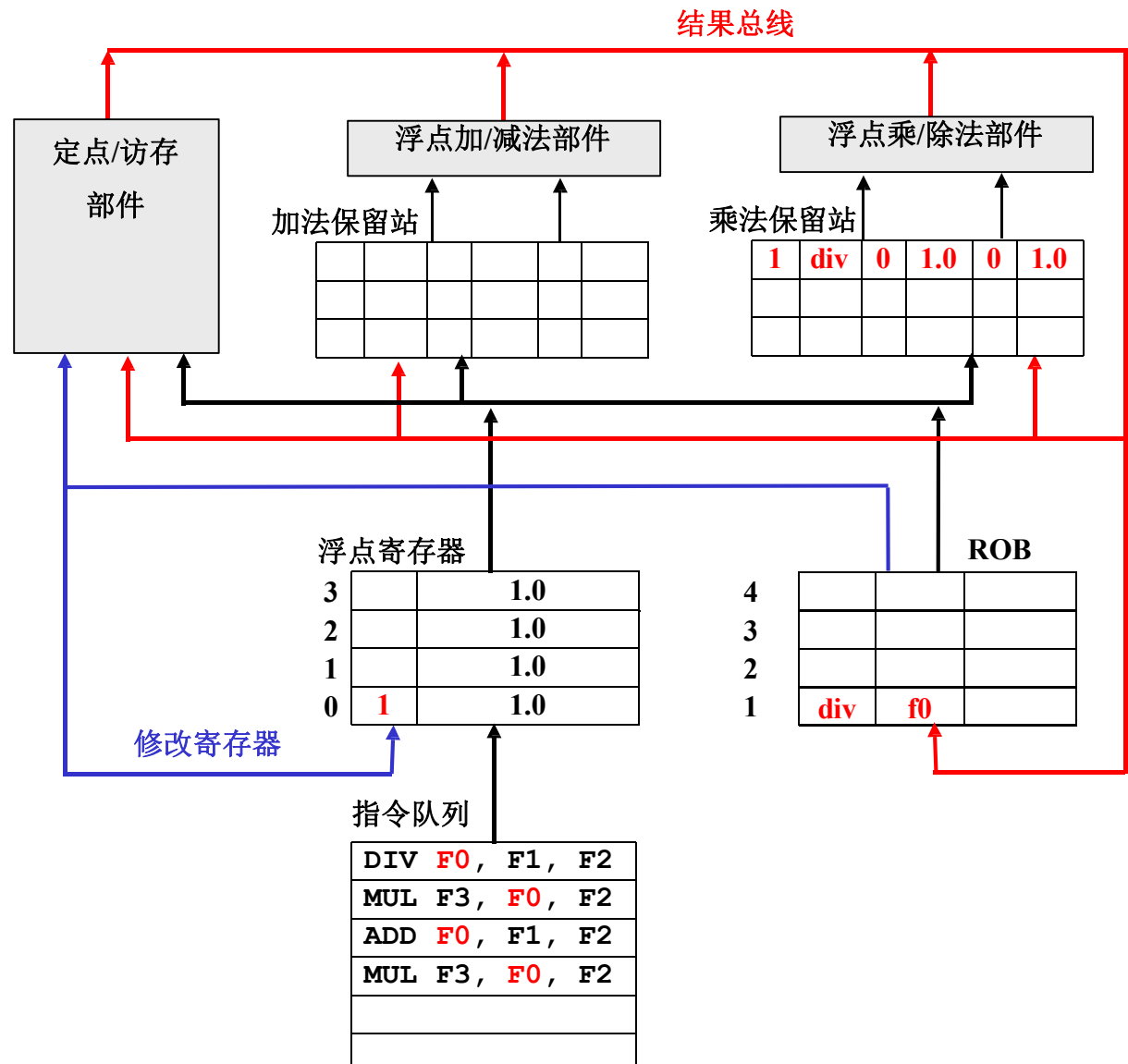
DIV F0, F1, F2

MUL F3, F0, F2

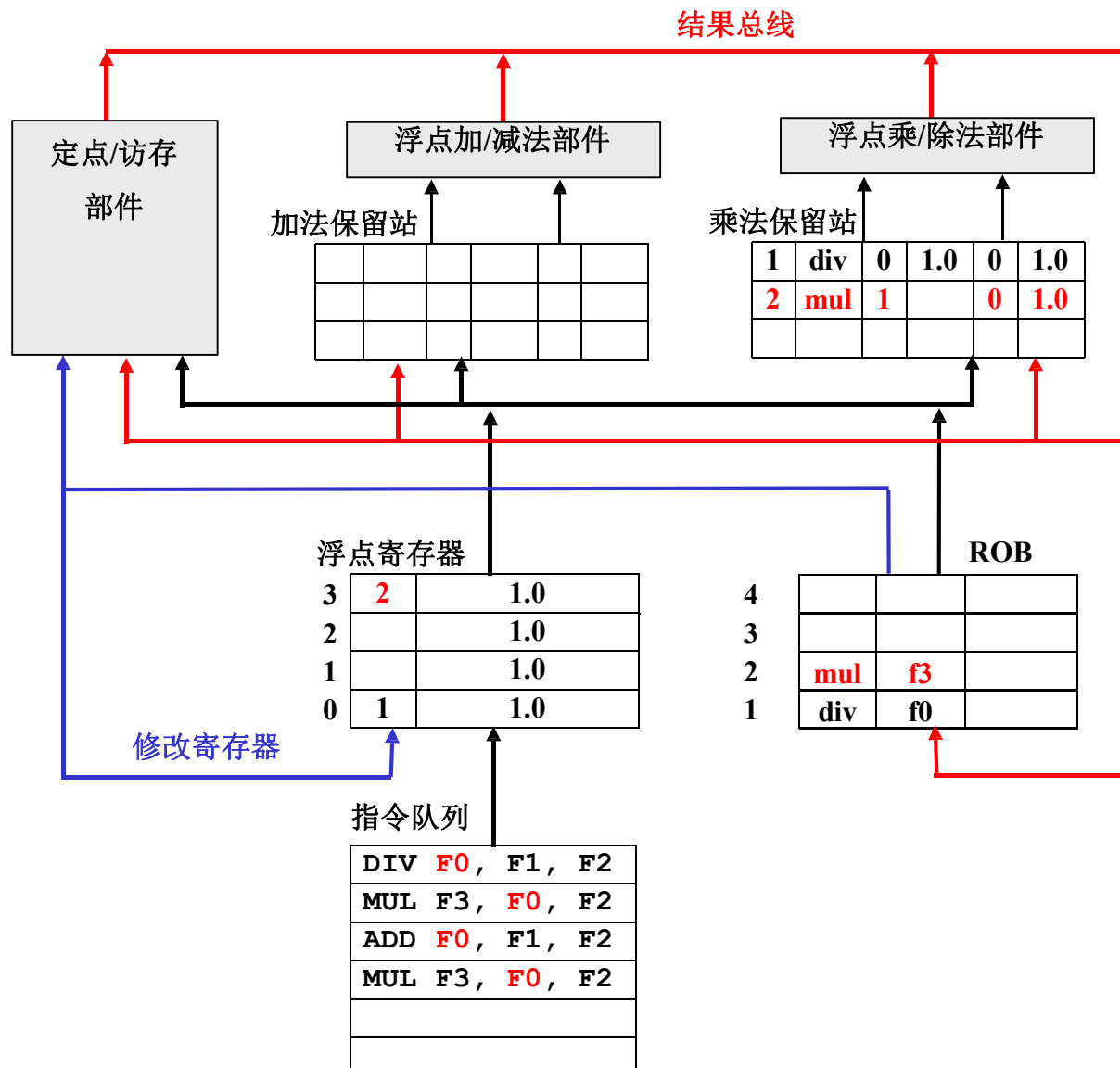
ADD F0, F1, F2

MUL F3, F0, F2

•DIV发射，F1, F2都准备好

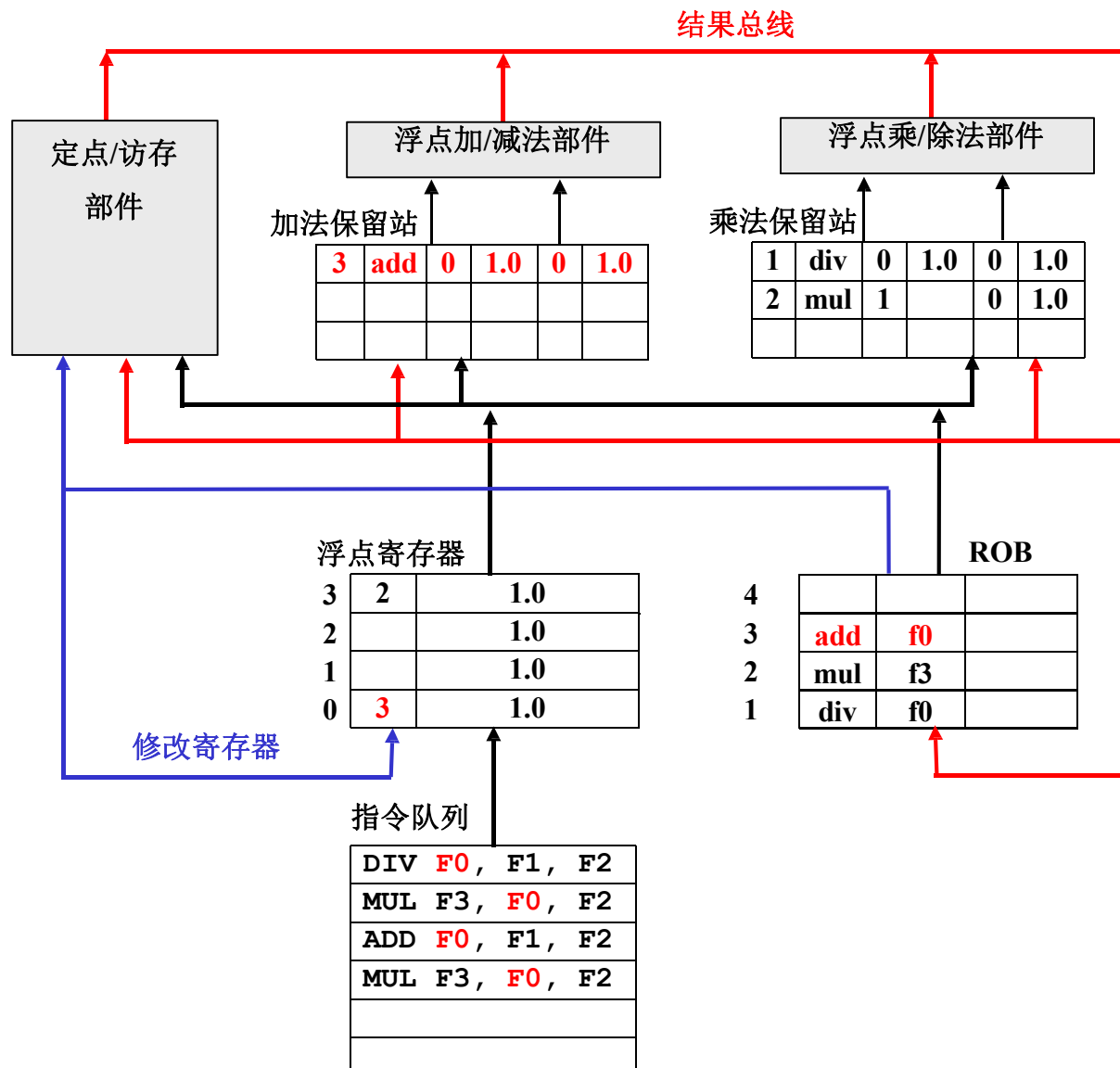


- DIV发射，F1, F2都准备好
- MUL1发射，F0等待1号RB的结果

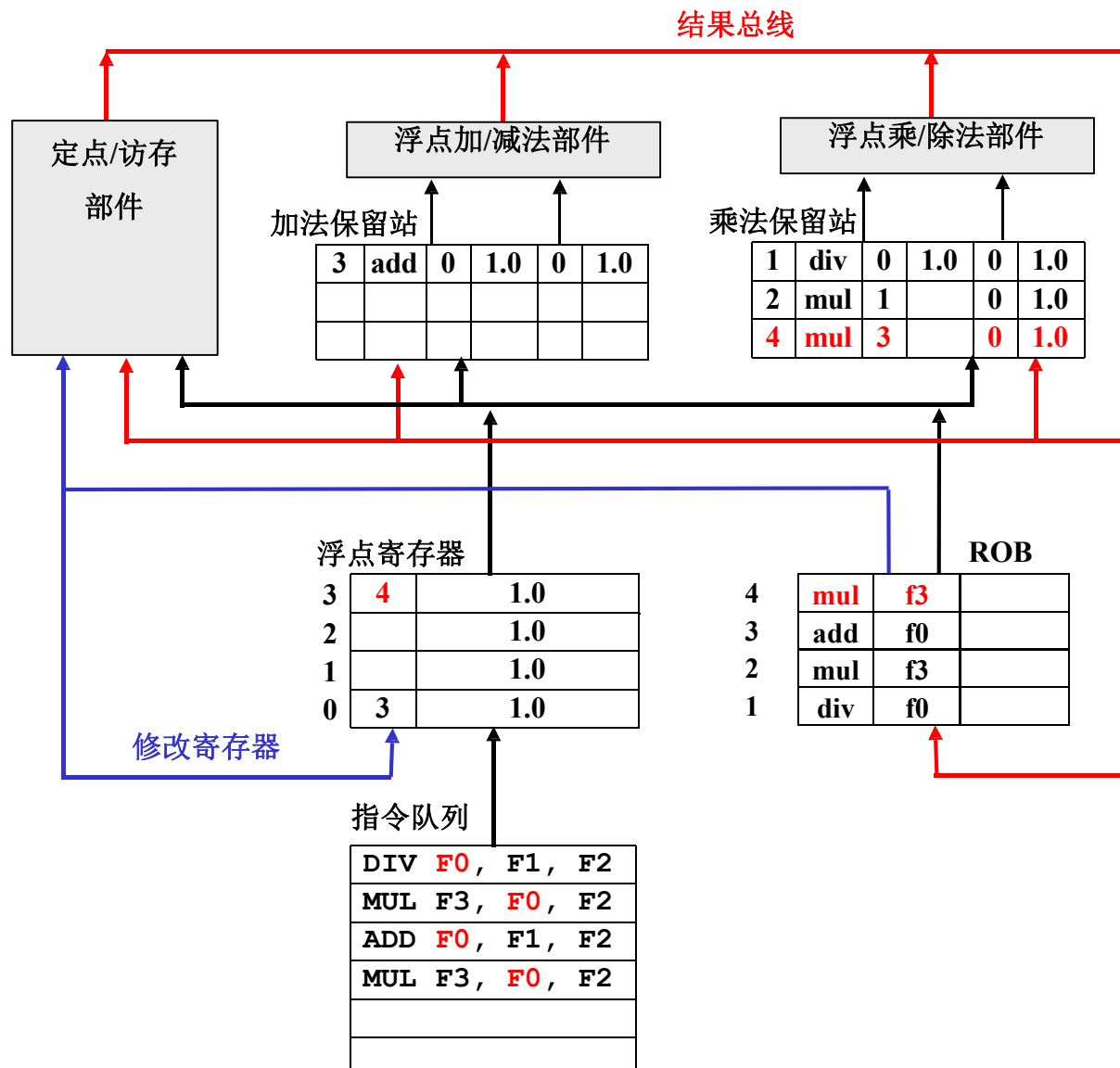




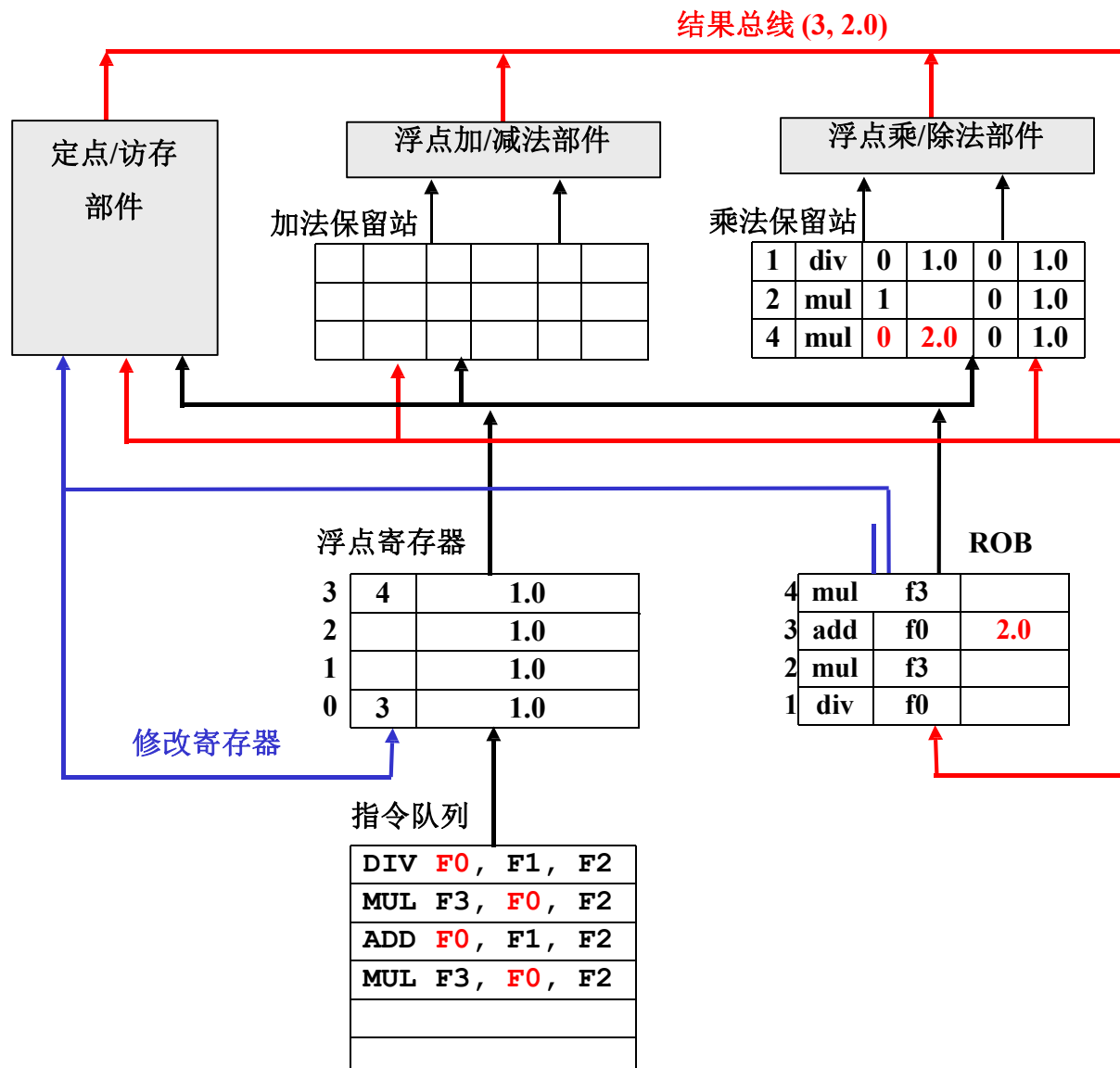
- DIV发射，F1,F2都准备好
- MUL1发射，F0等待1号RB的结果
- ADD发射，F1,F2都准备好



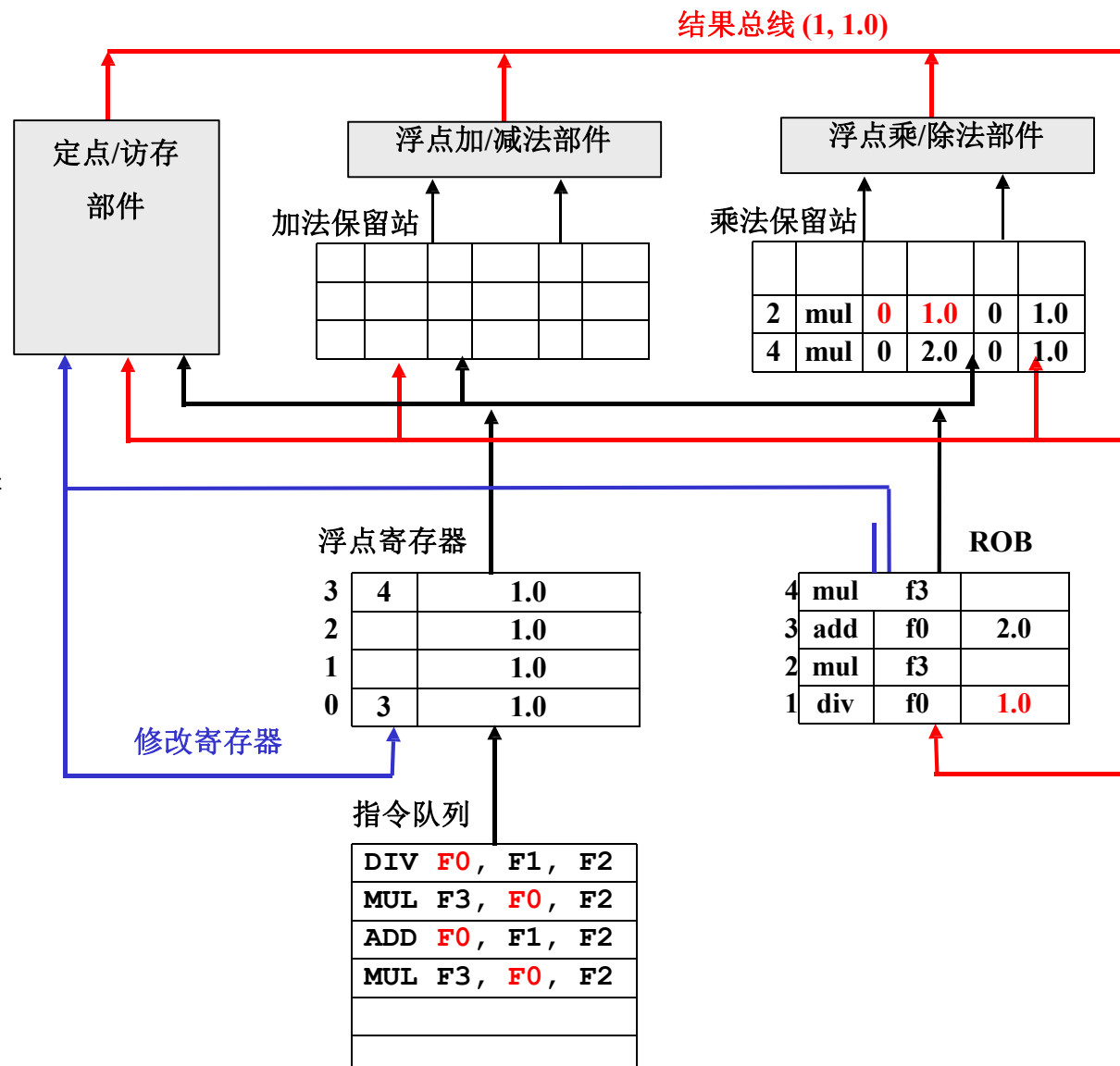
- DIV发射, F1, F2都准备好
- MUL1发射, F0等待1号RB的结果
- ADD发射, F1, F2都准备好
- MUL2发射, F0等待3号RB的结果



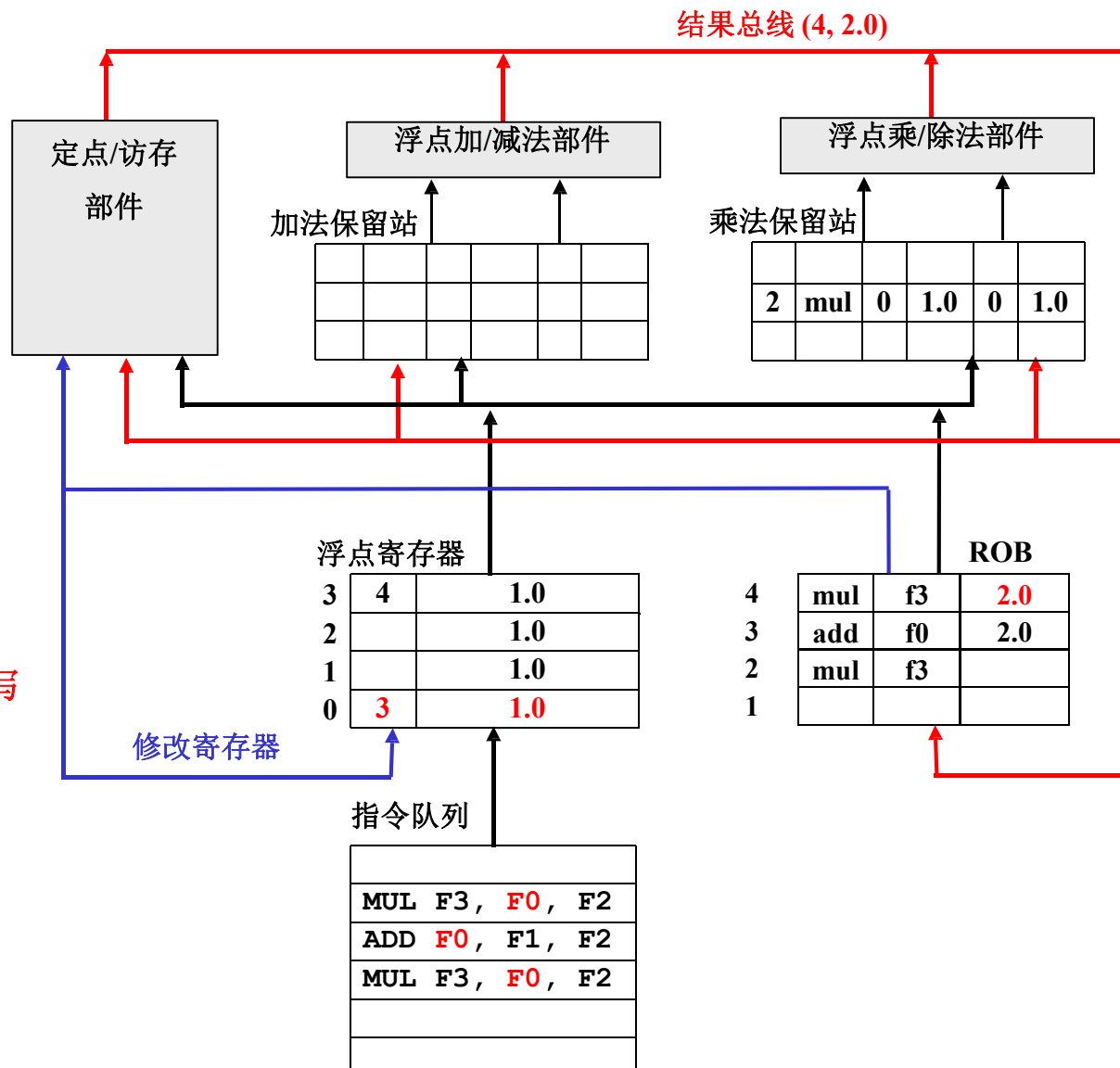
- DIV发射, F1, F2都准备好
- MUL1发射, F0等待1号RB的结果
- ADD发射, F1, F2都准备好
- MUL2发射, F0等待3号RB的结果
- ADD写回



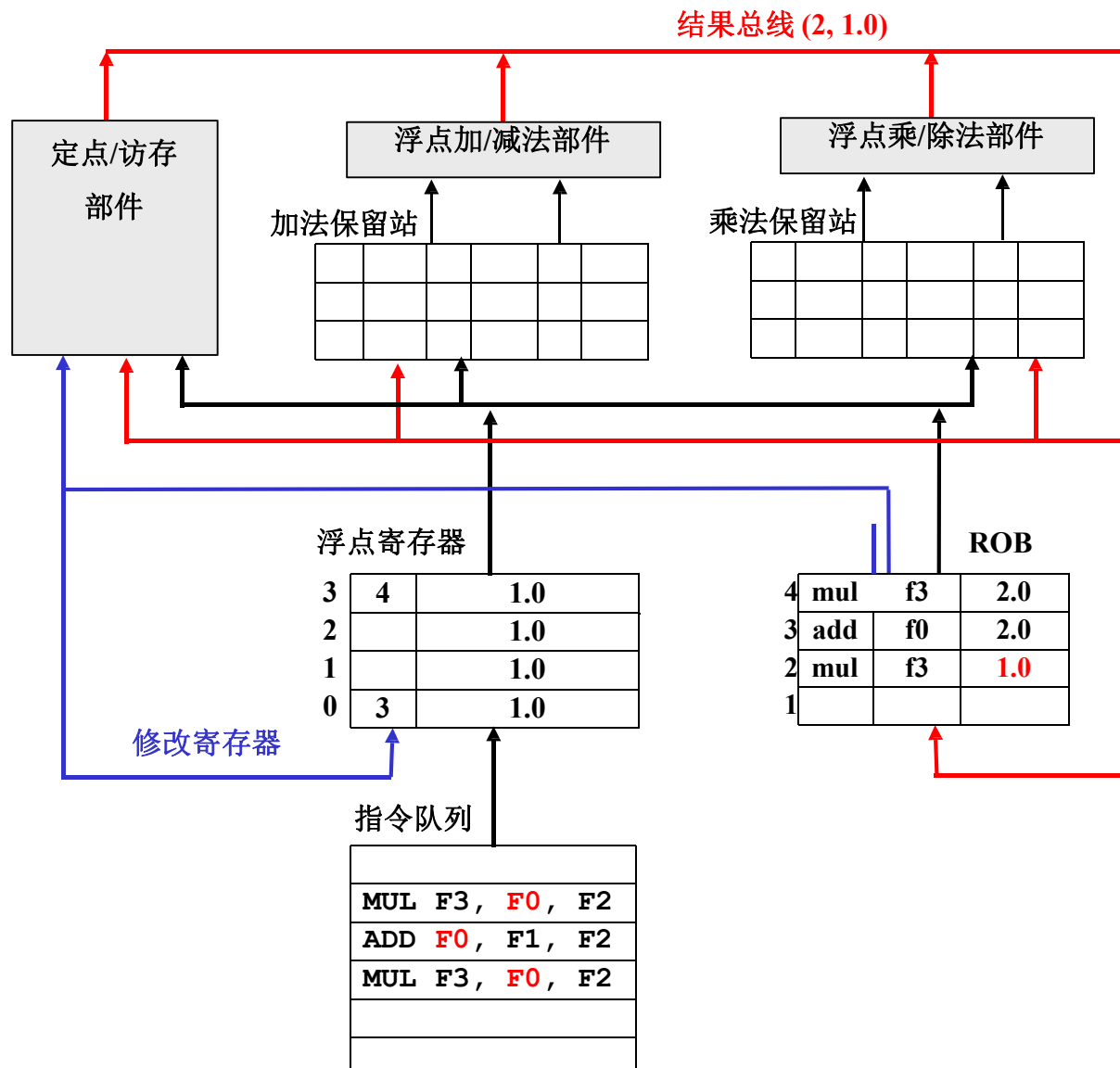
- DIV发射, F1, F2都准备好
- MUL1发射, F0等待1号RB的结果
- ADD发射, F1, F2都准备好
- MUL2发射, F0等待3号RB的结果
- ADD写回
- DIV写回



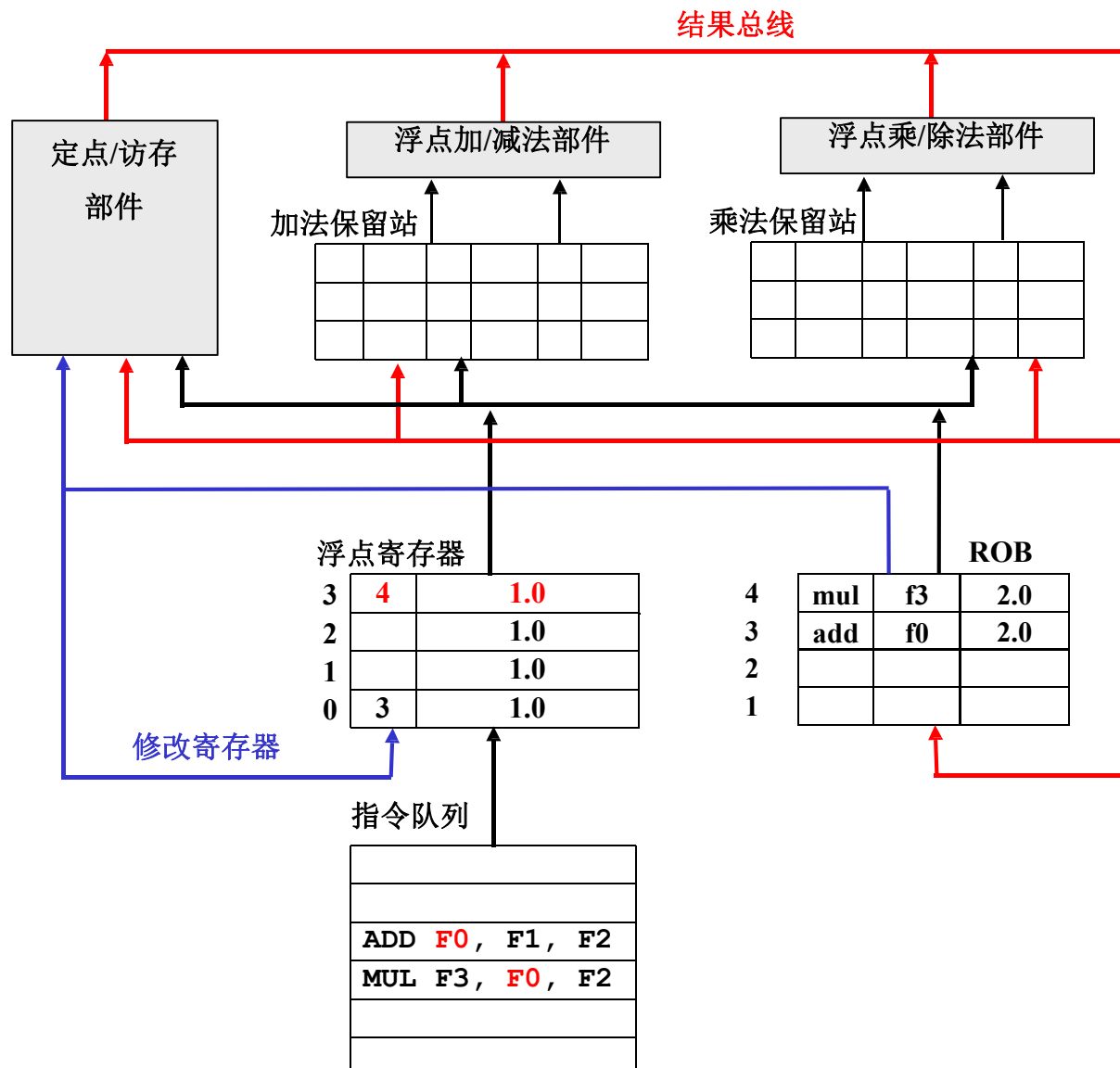
- DIV发射, F1, F2都准备好
- MUL1发射, F0等待1号RB的结果
- ADD发射, F1, F2都准备好
- MUL2发射, F0等待3号RB的结果
- ADD写回
- DIV写回
- DIV 提交 (写回寄存器), MUL2写回



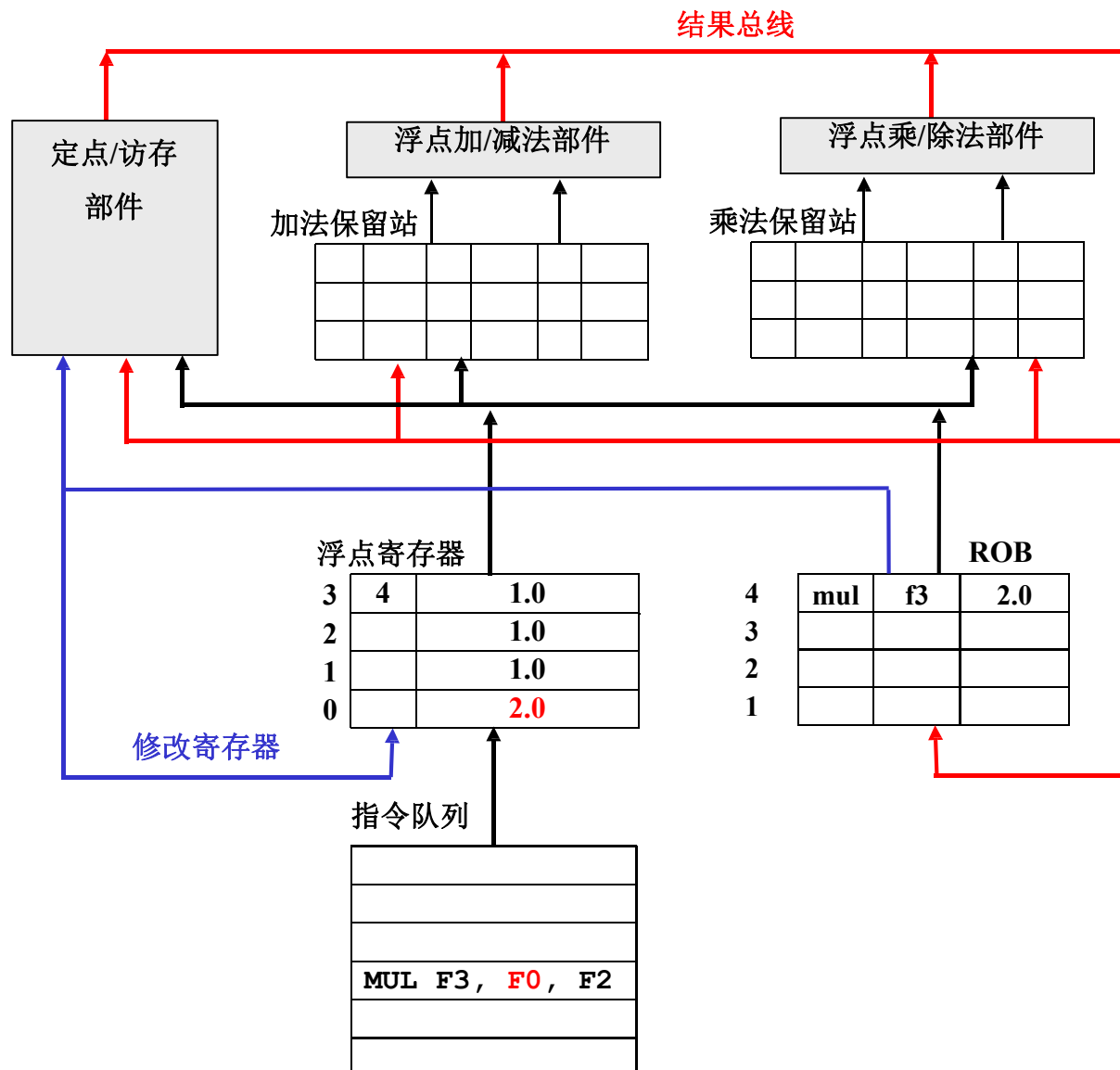
- DIV发射, F1, F2都准备好
- MUL1发射, F0等待1号RB的结果
- ADD发射, F1, F2都准备好
- MUL2发射, F0等待3号RB的结果
- ADD写回
- DIV写回
- DIV Commit, MUL2写回
- MUL1写回



- DIV发射, F1, F2都准备好
- MUL1发射, F0等待1号RB的结果
- ADD发射, F1, F2都准备好
- MUL2发射, F0等待3号RB的结果
- ADD写回
- DIV写回
- DIV Commit, MUL2写回
- MUL1写回
- MUL1 Commit**

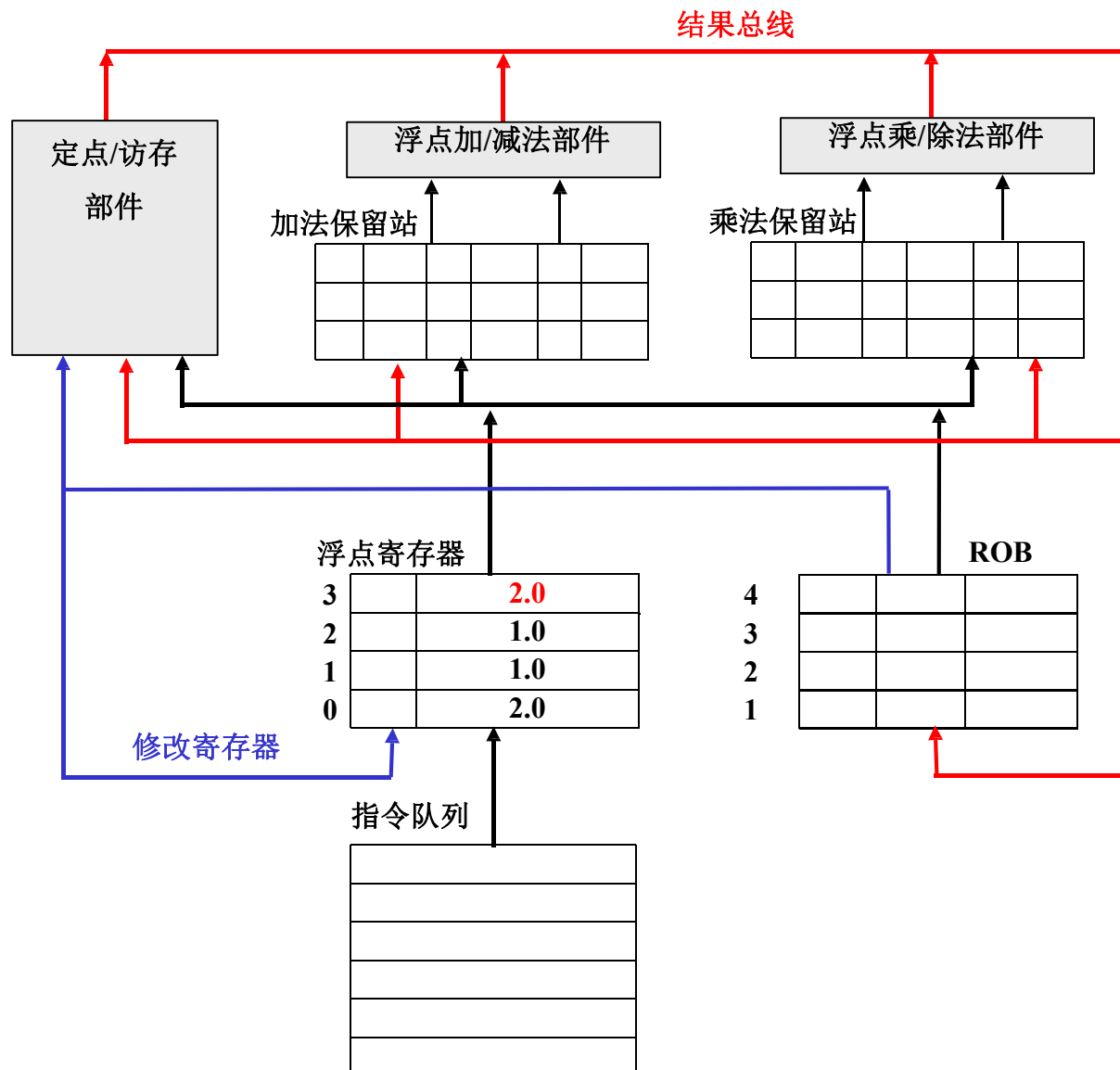


- DIV发射, F1, F2都准备好
- MUL1发射, F0等待1号RB的结果
- ADD发射, F1, F2都准备好
- MUL2发射, F0等待3号RB的结果
- ADD写回
- DIV写回
- DIV Commit, MUL2写回
- MUL1写回
- MUL1 Commit
- ADD Commit



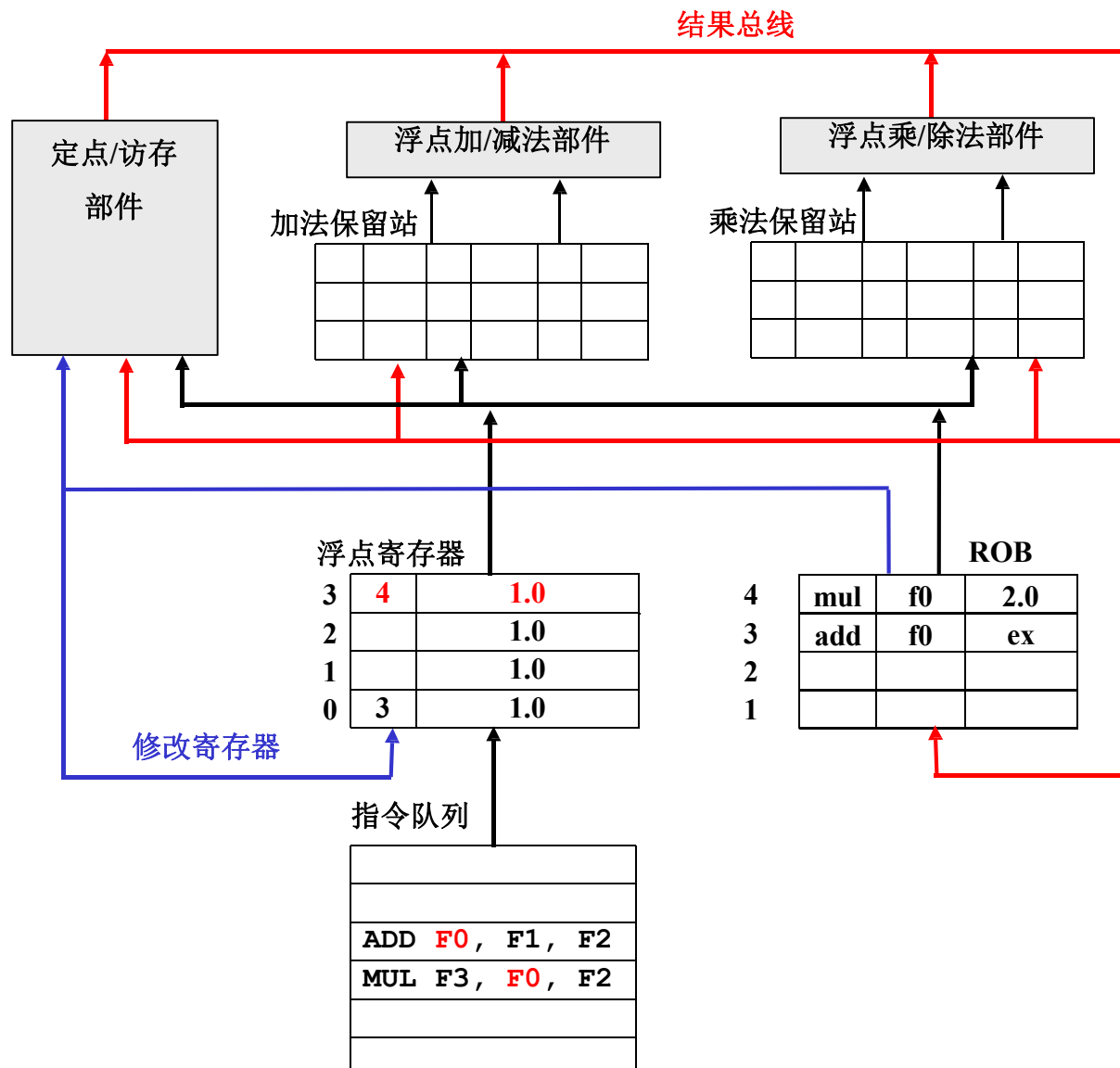


- DIV发射, F1, F2都准备好
- MUL1发射, F0等待1号RB的结果
- ADD发射, F1, F2都准备好
- MUL2发射, F0等待3号RB的结果
- ADD写回
- DIV写回
- DIV Commit, MUL2写回
- MUL1写回
- MUL1 Commit
- ADD Commit
- MUL2 Commit

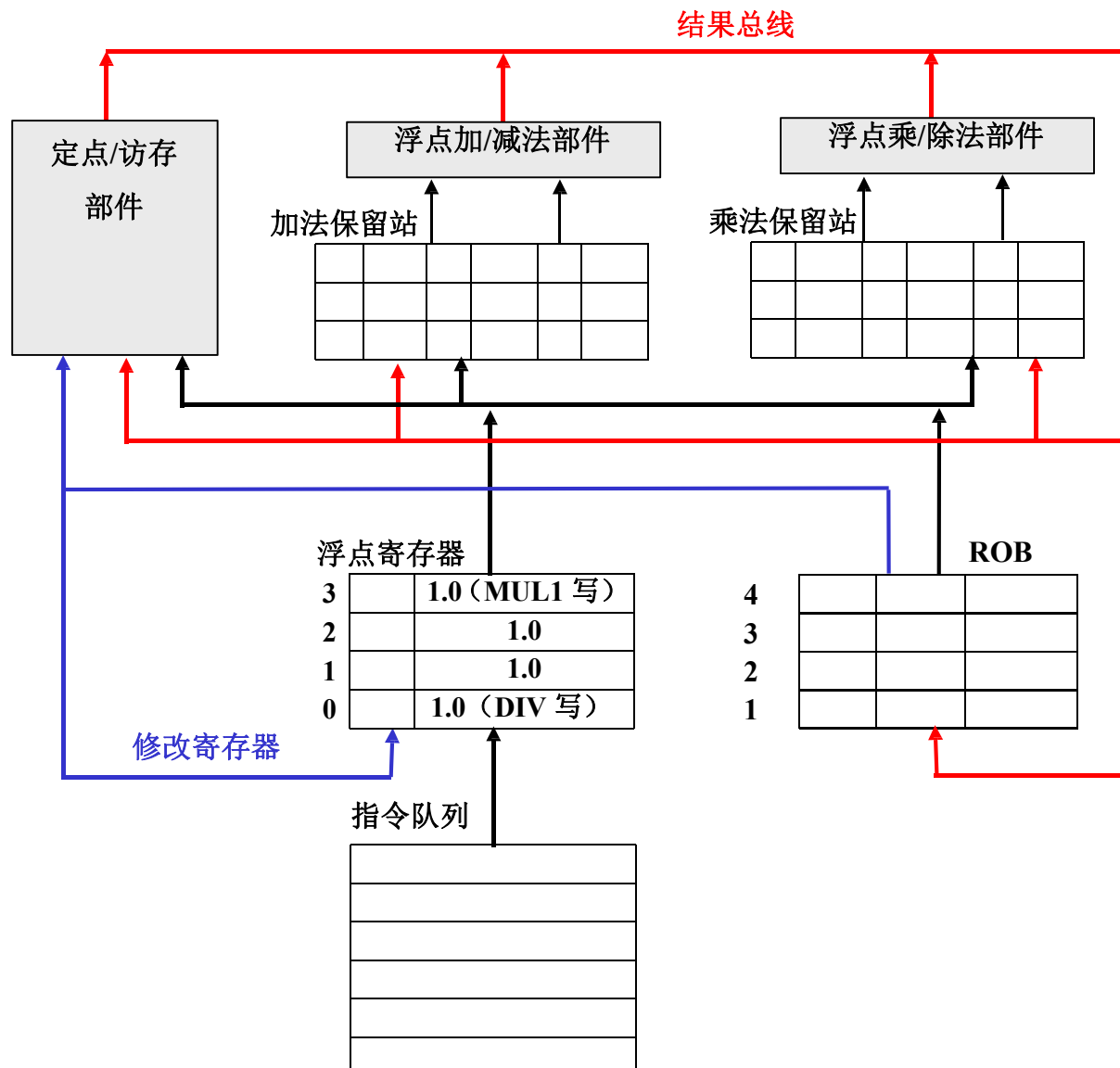


假设ADD发生了溢出例外

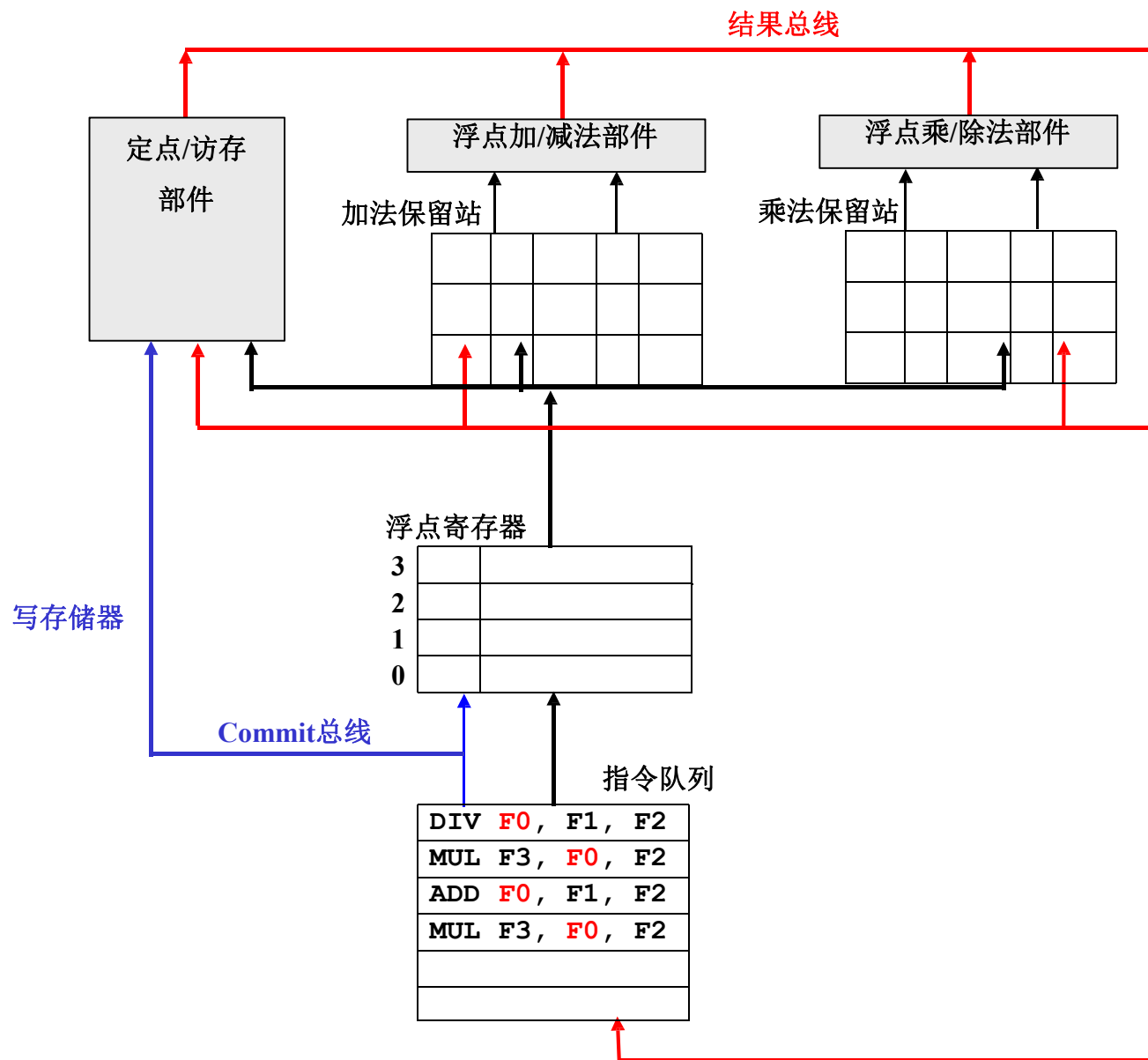
- DIV发射, F1, F2都准备好
- MUL1发射, F0等待1号RB的结果
- ADD发射, F1, F2都准备好
- MUL2发射, F0等待3号RB的结果
- ADD写回
- DIV写回
- DIV Commit, MUL2写回
- MUL1写回
- MUL1 Commit**



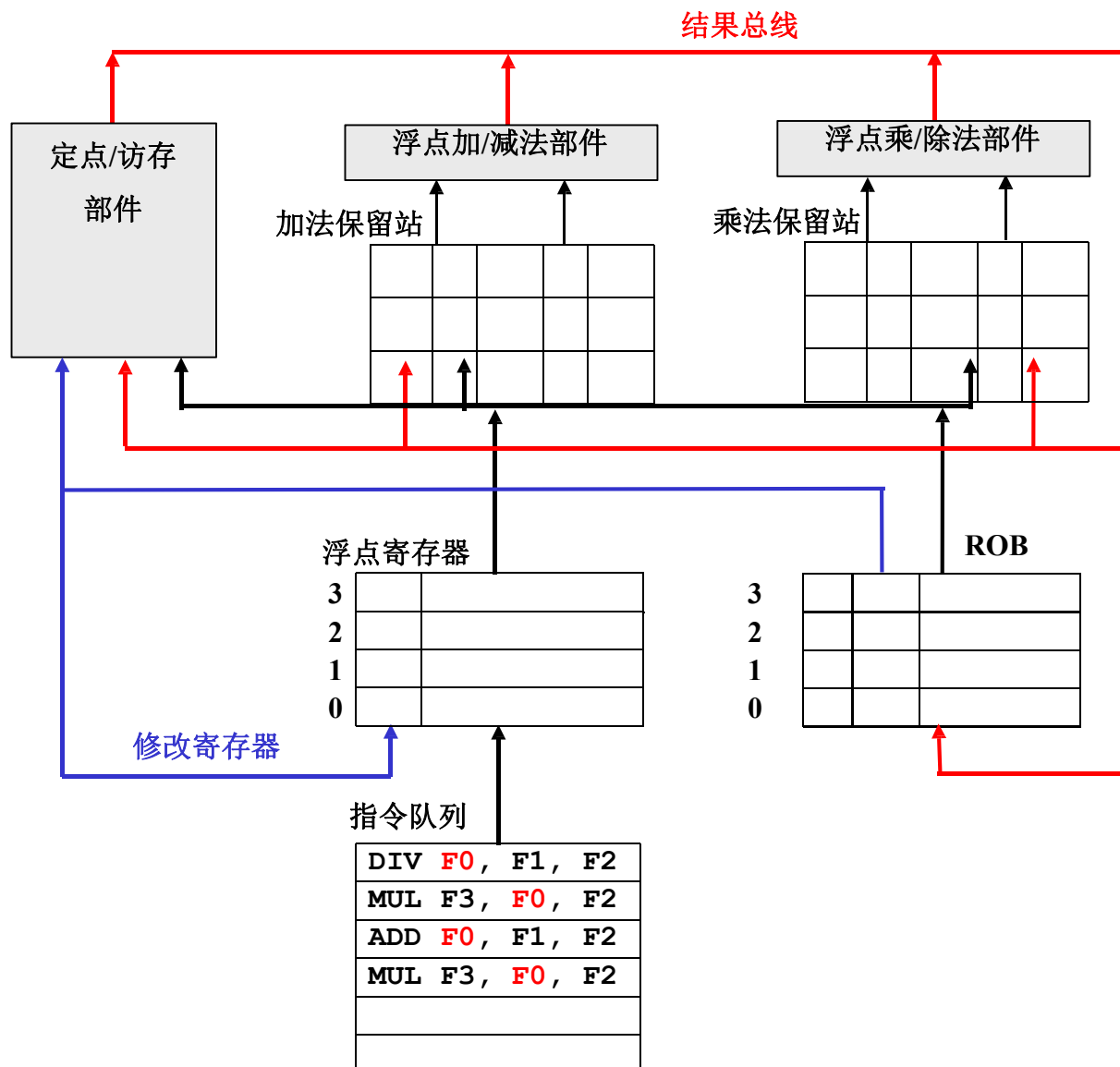
- DIV发射, F1, F2都准备好
- MUL1发射, F0等待1号RB的结果
- ADD发射, F1, F2都准备好
- MUL2发射, F0等待3号RB的结果
- ADD写回
- DIV写回
- DIV Commit, MUL2写回
- MUL1写回
- MUL1 Commit
- ADD Commit



- 龙芯1号把ROB和队列合并



- 增加Reorder Buffer的流水线



# 增加ROB的流水线

- 发射：把操作队列的指令根据操作类型送到保留站（如果保留站以及ROB有空），并在ROB中指定一项作为临时保存该指令结果之用；发射过程中读寄存器的值和结果状态域，如果结果状态域指出结果寄存器已被重命名到ROB，则读ROB
- 执行：如果所需的操作数都准备好，则执行，否则根据结果ROB号侦听结果总线并接收结果总线的值
- 写回：把结果送到结果总线，释放保留站；ROB根据结果总线修改相应项
- 提交：如果队列中第一条指令的结果已经写回且没有发生例外，把该指令的结果从ROB写回到寄存器或存储器，释放ROB的相应项。如果队列头的指令发生了例外或是猜测错误的转移指令，清除操作队列以及ROB等

第二天

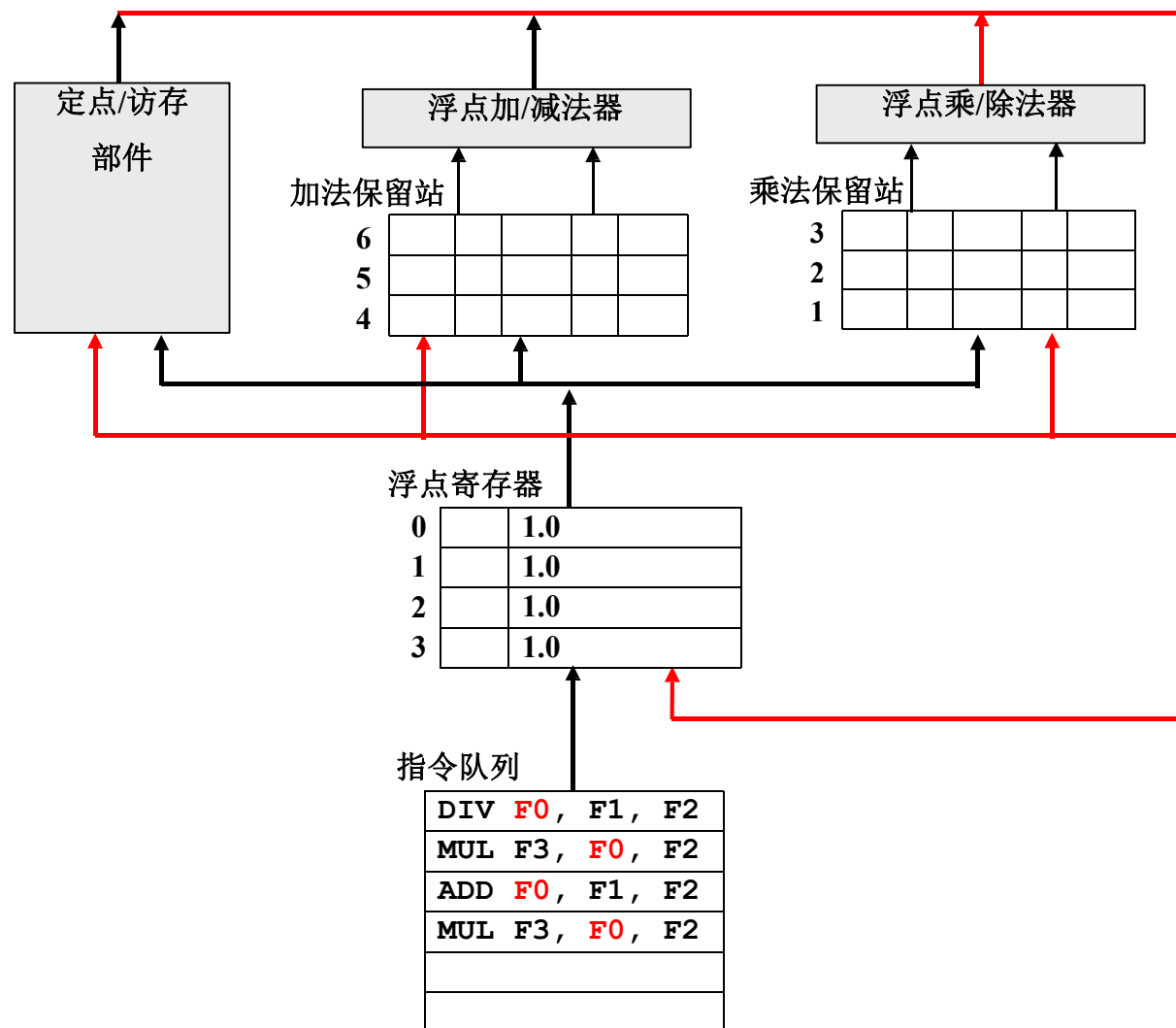


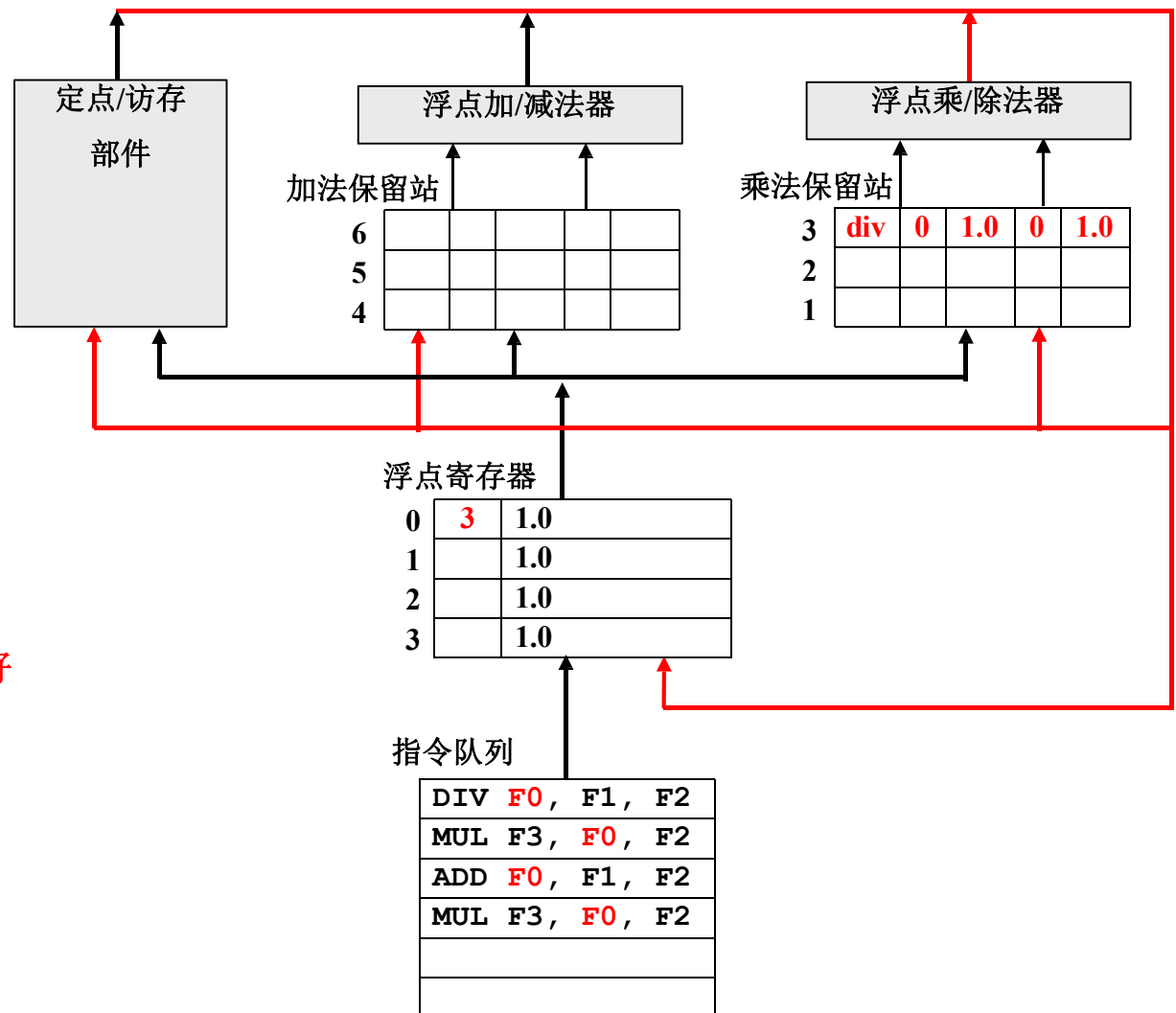
# 计算机体系结构

韩银和

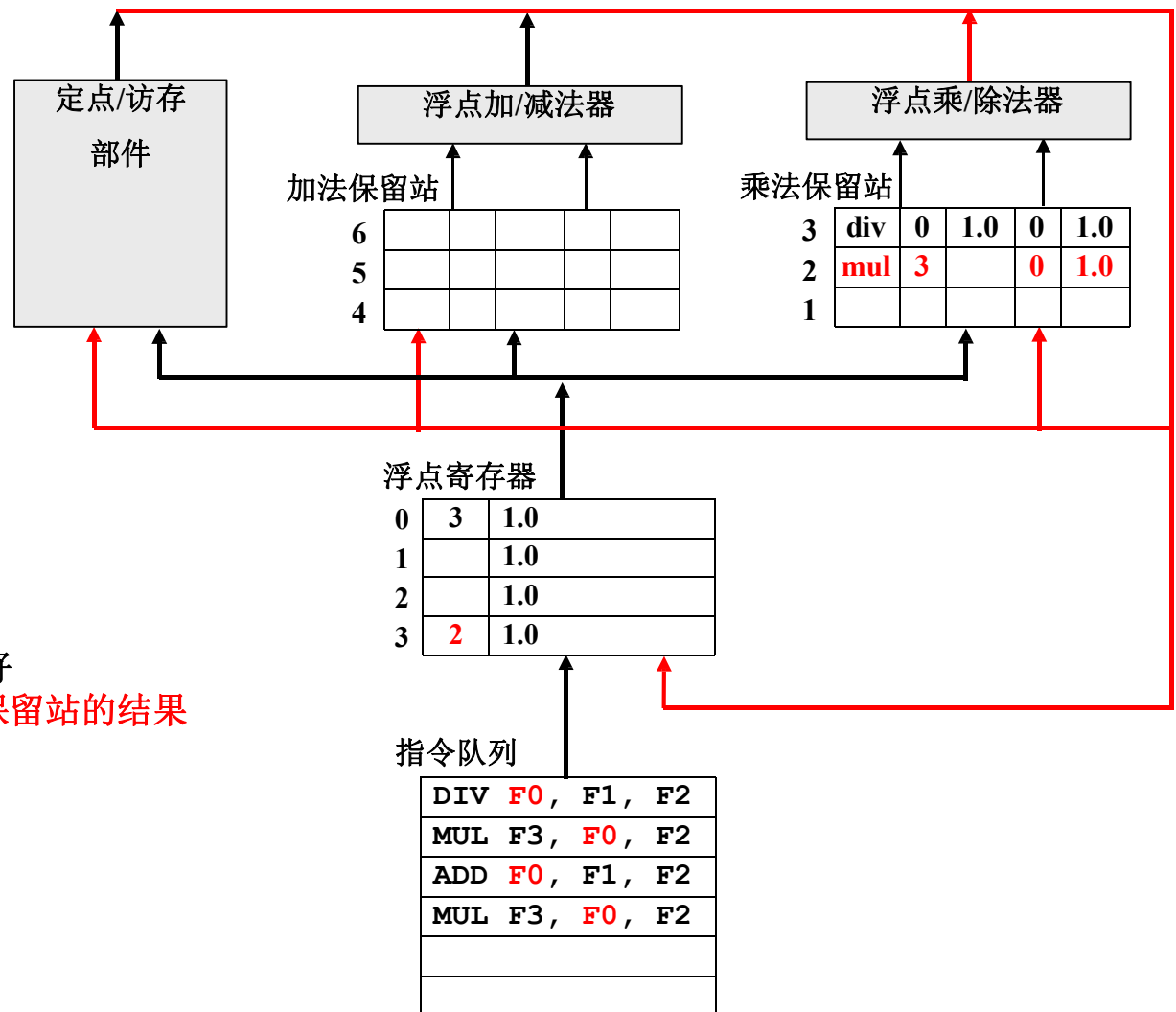
PPT部分摘自：胡伟武、汪文祥老师的讲义

# 回顾一下Tomasulo算法

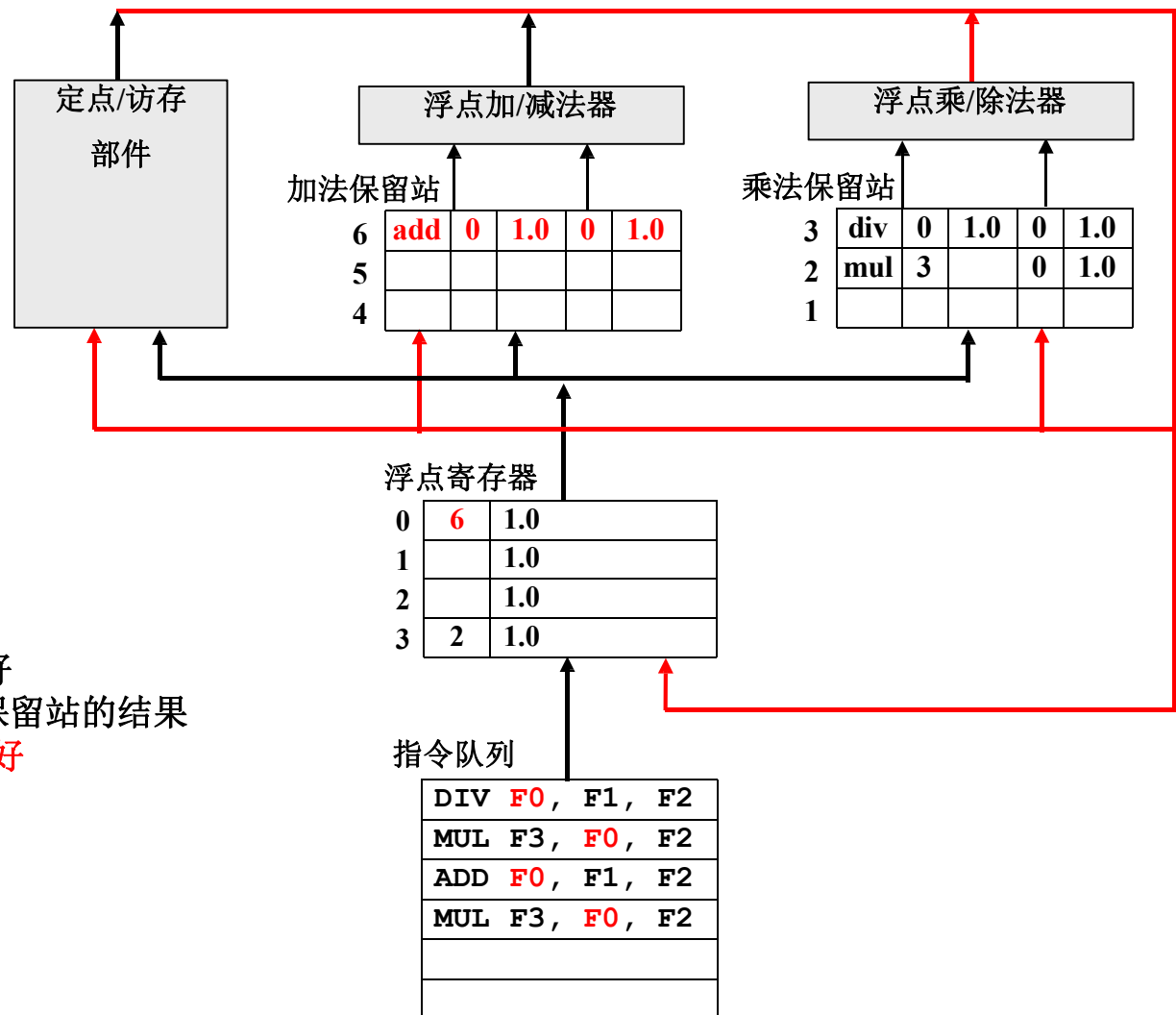




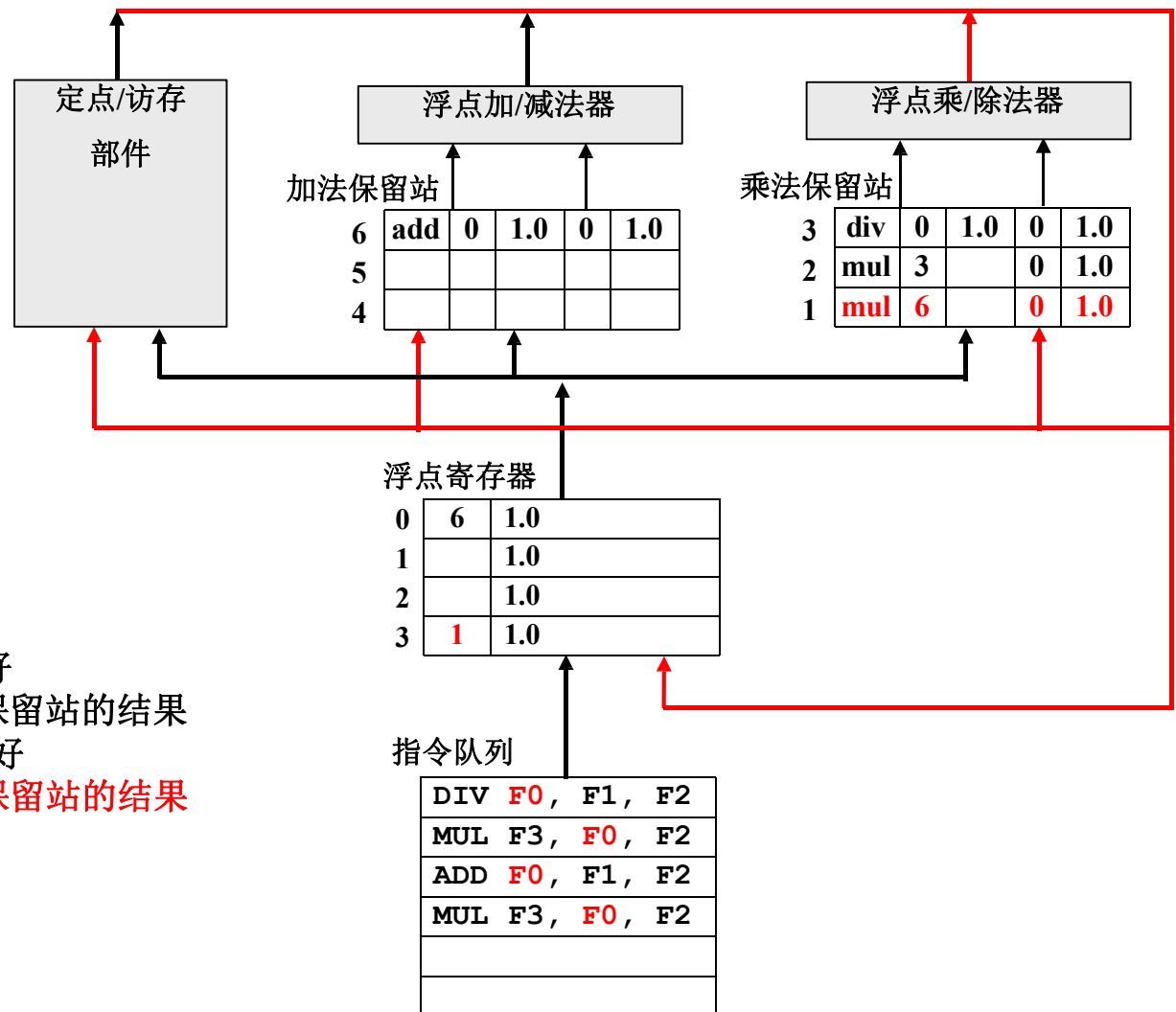
•DIV发射，F1,F2都准备好



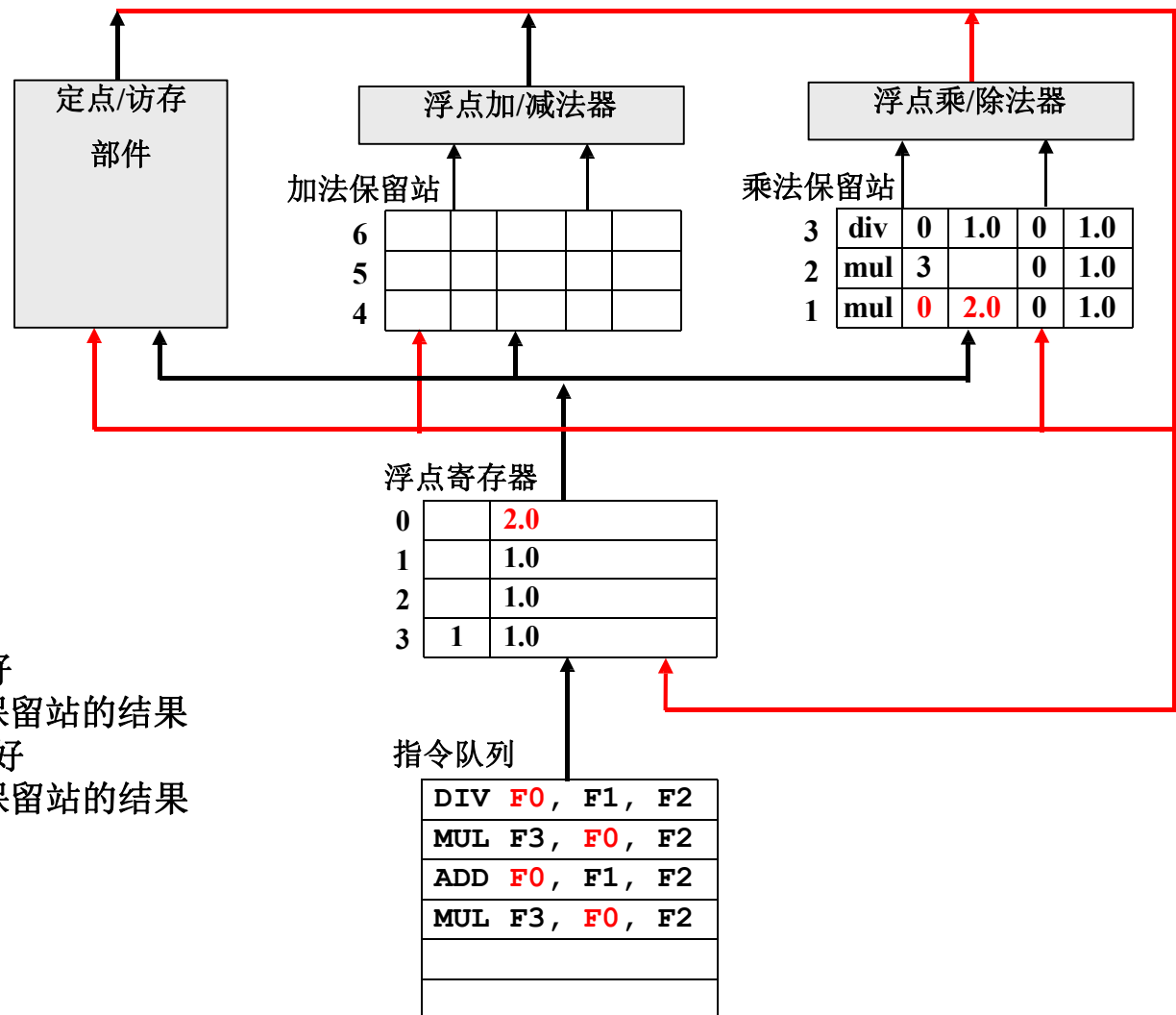
- DIV发射，F1,F2都准备好
- MUL1发射，F0等待3号保留站的结果



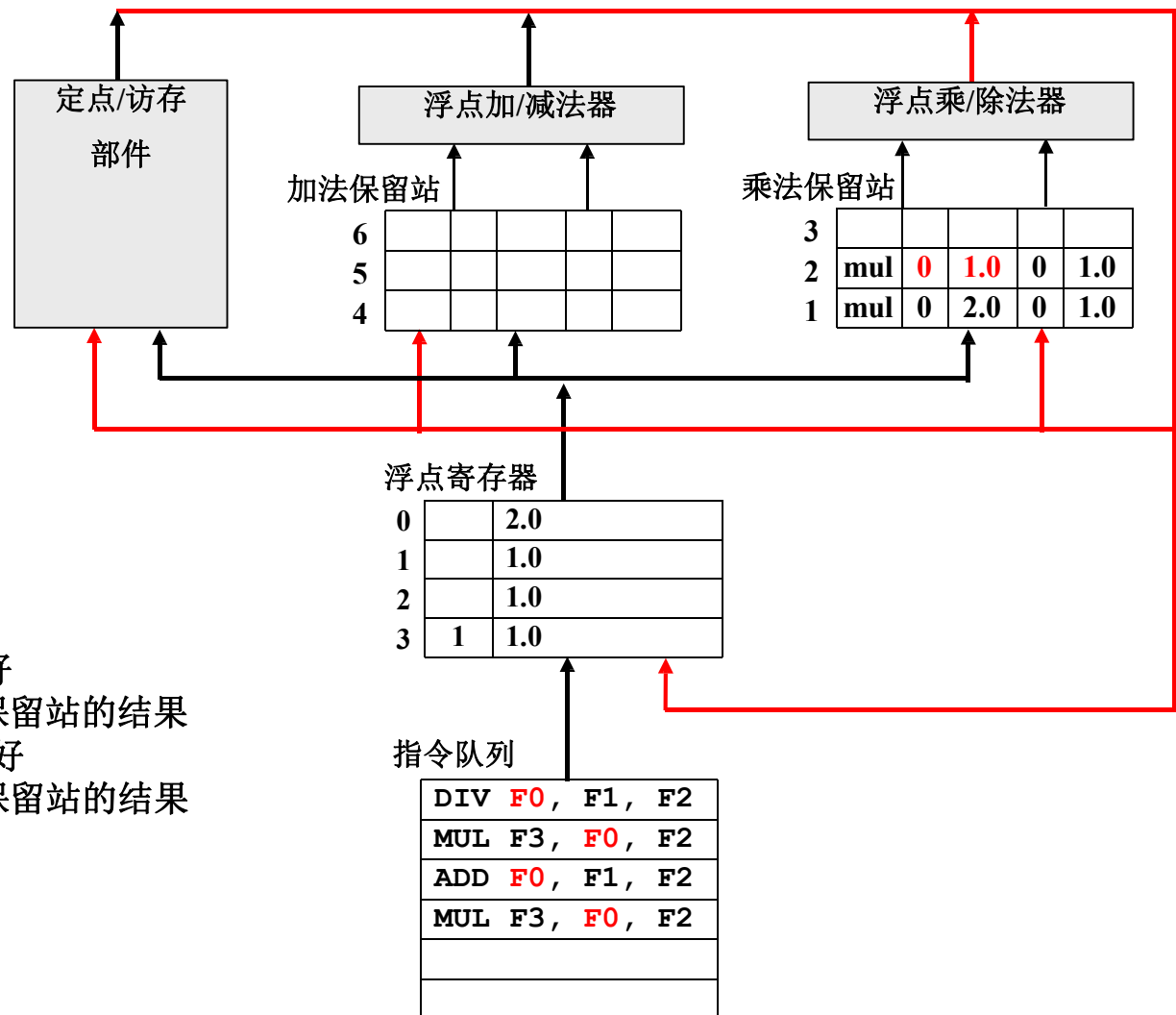
- DIV发射，F1,F2都准备好
- MUL1发射，F0等待3号保留站的结果
- ADD发射，F1,F2都准备好



- DIV发射，F1,F2都准备好
- MUL1发射，F0等待3号保留站的结果
- ADD发射，F1,F2都准备好
- MUL2发射，F0等待6号保留站的结果



- DIV发射，F1,F2都准备好
- MUL1发射，F0等待3号保留站的结果
- ADD发射，F1,F2都准备好
- MUL2发射，F0等待6号保留站的结果
- ADD写回



- DIV发射，F1,F2都准备好
- MUL1发射，F0等待3号保留站的结果
- ADD发射，F1,F2都准备好
- MUL2发射，F0等待6号保留站的结果
- ADD写回
- DIV写回



# 多发射数据通路

- 乱序执行的关键技术
- 动态调度流水线数据通路
  - 保留站的组织
  - 寄存器与保留站的关系
  - 寄存器重命名方法
- 常见处理器的数据通路
- 多发射结构数据通路
- 龙芯2号多发射结构简介

# 乱序执行的关键技术

# 指令级并行的关键技术

- 指令流水线: 时间重叠
- 多发射: 空间重复
- 乱序执行（有序结束）: 充分利用资源
  - **动态调度**: 前面指令因相关而等待时, 后面的可继续前进。
  - **转移猜测**: 在转移条件确定前, 猜测某个分支取指并执行
  - **非阻塞访存**: 提高访存指令执行效率, 减少访存阻塞
  - 乱序执行可以提高性能**1.5-2倍**。

- 动态调度的主要作用及其思想
  - 把读寄存器从译码中区分开来，并利用保留站等指令缓存技术避免前面的阻塞的指令影响后面指令的执行
  - 通过ROB对执行完的指令重新排序实现有序结束
  - 利用寄存器重命名技术保存未提交的临时结果，消除WAW和WAR相关并支持猜测性执行
- 有序进入、乱序执行、有序结束

# 影响动态调度的主要因素

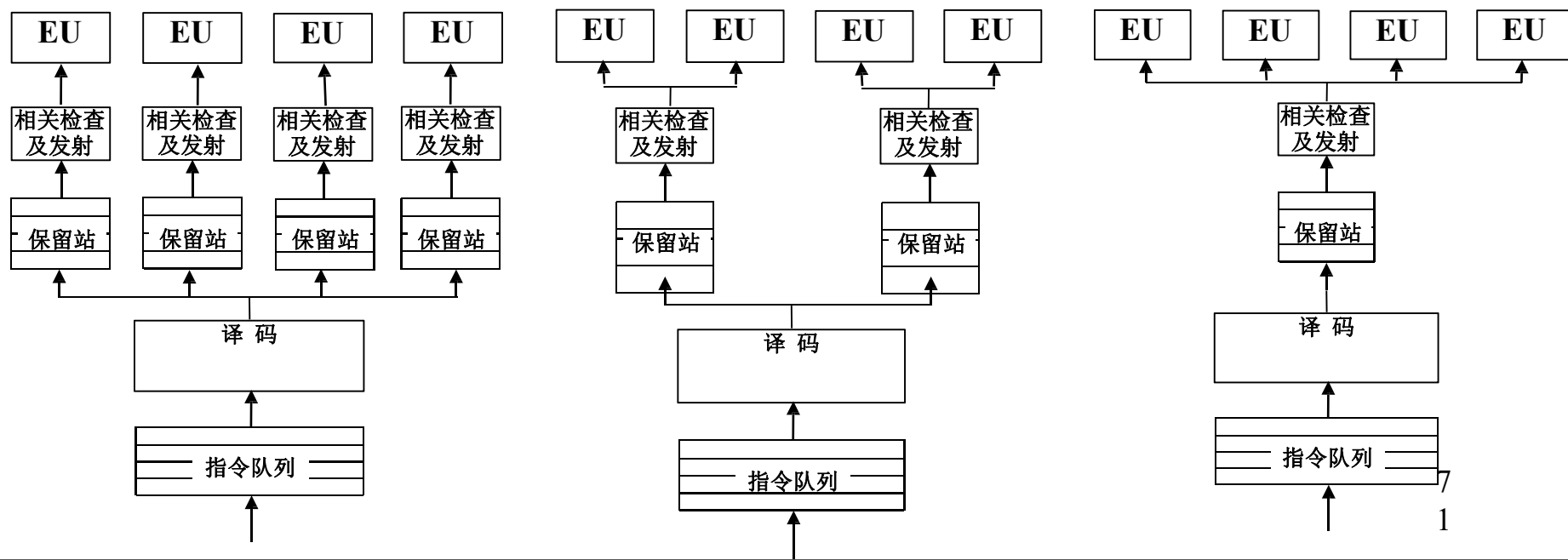
- 指令缓存的结构
  - 独立保留站
  - 组保留站
  - 全局保留站
- 读取寄存器内容的时间
  - 保留站前读
  - 保留站后读
- 寄存器重命名的方法
  - 重命名寄存器和物理寄存器分开
  - 重命名寄存器和物理寄存器合并

# 保留站的组织

# 指令缓存结构

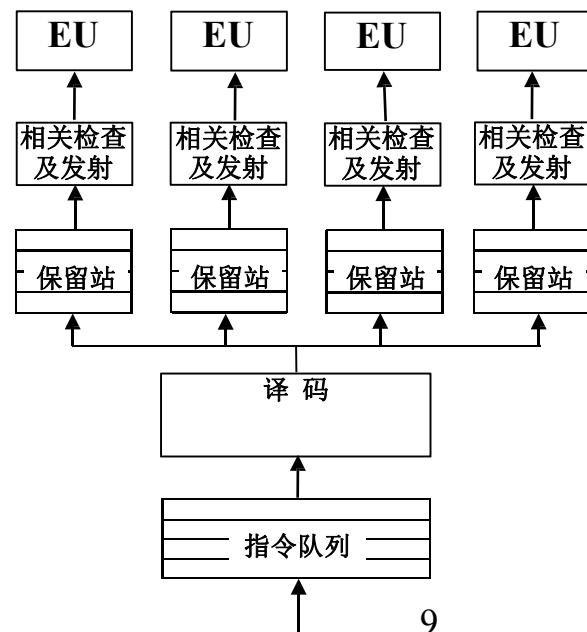
## • 保留站组织

- 独立保留站：每个功能部件一个保留站
- 分组保留站：多个功能部件共享保留站
- 全局保留站：所有功能部件共享保留站
- 比较：数据通路复杂度（结果总线传送）、保留站项数、保留站效率、发射复杂度



# 独立保留站

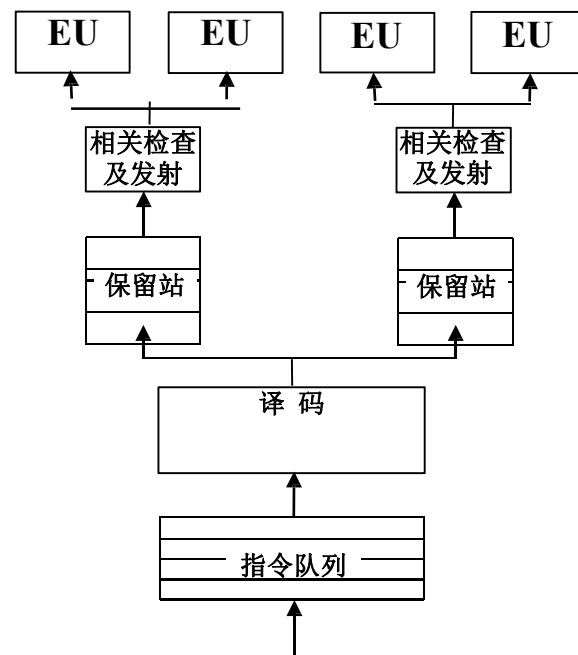
- 每个功能部件都有自己的保留站
  - 每个保留站项数较少（2-4项），只要一个写入端口一个读出端口，输出选择比较简单
  - 保留站利用率低，可能忙的忙死甚至引起堵塞，闲的闲死
  - 结果总线送到所有保留站，连线长，结果写回可能需要单独一拍
  - CDC6600每个EU一项保留站
  - IBM360/91中浮点加减部件两项，浮点乘法部件3项保留站
  - PowerPC 620三个定点一个浮点部件每个两项保留站
  - 龙芯1号三个定点两个浮点部件每个两项保留站





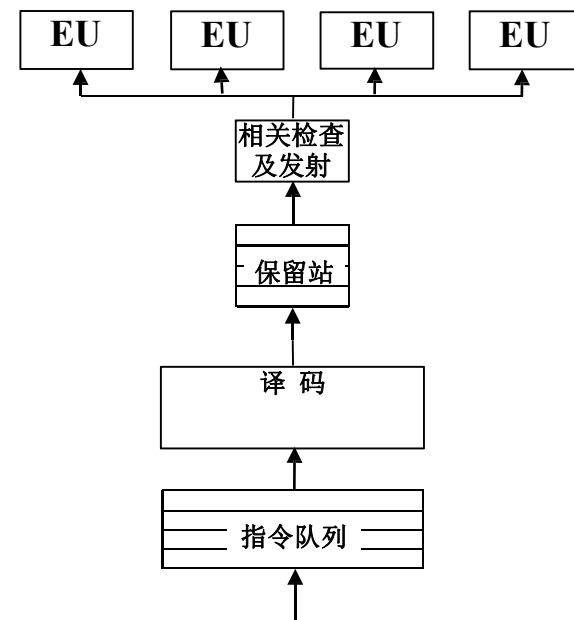
# 分组保留站

- 同组的功能部件共享保留站
  - 每个保留站项数较多，每个保留站需要多个写入端口多个读出口，保留站读出可能需要单独一拍
  - 保留站效率较高
  - 结果总线不用送到每个功能部件
  - 一般划分时考虑定点、浮点、访存
  - R10000定点、浮点、访存各16项
  - Alpha 21264定点20项、浮点15项
  - PA8700运算和访存各28项，还起到ROB作用
  - Godson-2采用组保留站



# 全局保留站

- 所有功能部件共享一个保留站
- 保留站项数很多，读出写入端口都很多，保留站读出时间长，保留站控制很复杂
- 保留站效率很高
- 结果总线只送到全局保留站
- **Pentium Pro**的全局保留站有20项



# 不同处理器的保留站组织

独立保留站	分组保留站	全局保留站
Power1 (1990) Nx586 (1994) PowerPC 603 (1993) PowerPC 604 (1995) PowerPC 620 (1996) Am 2900 sup (1995) Am K5 (1995) Godson-1 (2001)	ES/9000 Power2 (1993) R10000 (1996) PM1 (Sparc64) (1995) HP PA8700 (1998) Alpha 21264 (1998) Godson-2 (2003) AMD K7 (1999) AMD K8 (2003)	Penium Pro/II/III (1995) Pentium-M (2003) Core (2006)

# 不同处理器的保留站总项数比较

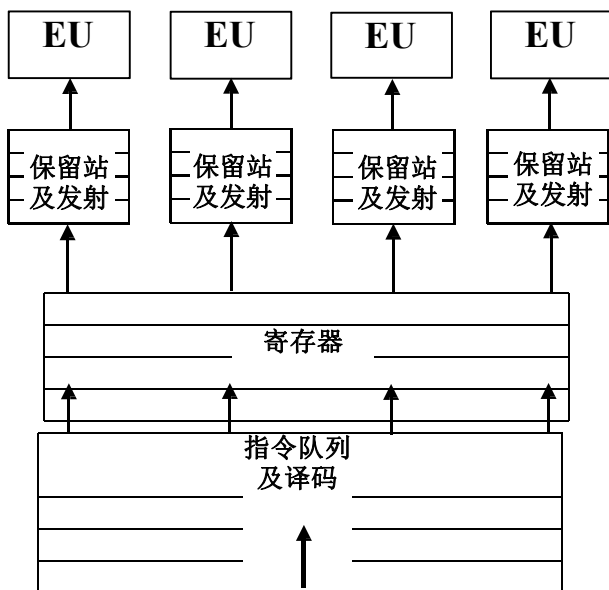
处理器	保留站总项数
PowerPC 603 (1993)	3
PowerPC 604 (1995)	12
PowerPC 620 (1996)	15
Nx586 (1994)	42
K5 (1995)	14
PM1 (Sparc64) (1995)	36
Pentium Pro (1995)	20
R10000 (1996)	48
PA8000 (1996)	56
Alpha 21264	35
AMD K7/K8	54/60
Intel Core	32
Godson-2	32

# 寄存器与保留站的关系

# 读取寄存器值的时机

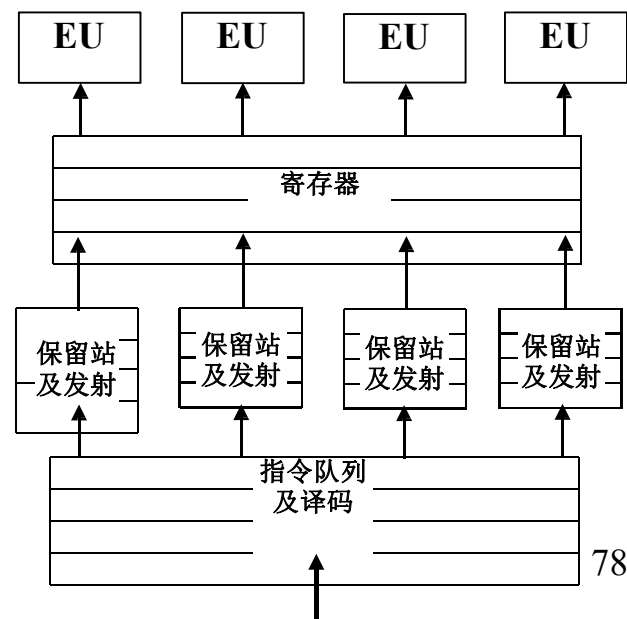
## • 保留站前读寄存器

- 保留站中有值域，较复杂
- 操作数没准备好就读寄存器，保留站侦听结果总线
- 有序发射



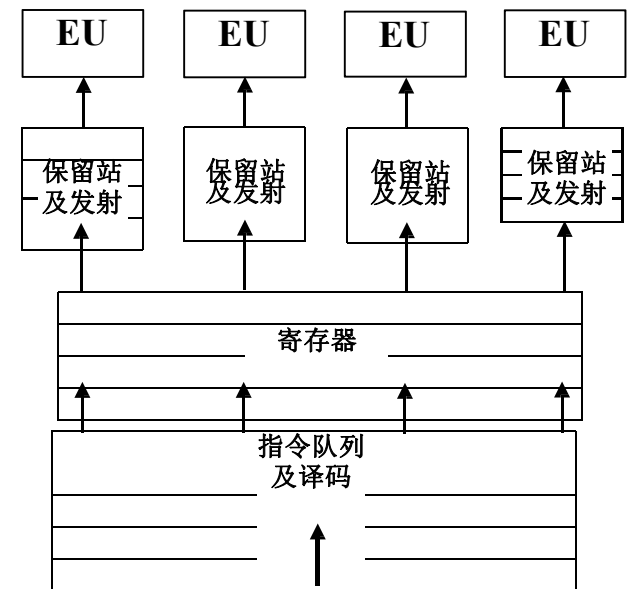
## • 保留站后读寄存器

- 保留站中无值域，较简单
- 操作数全部准备好后才能读寄存器
- 乱序发射



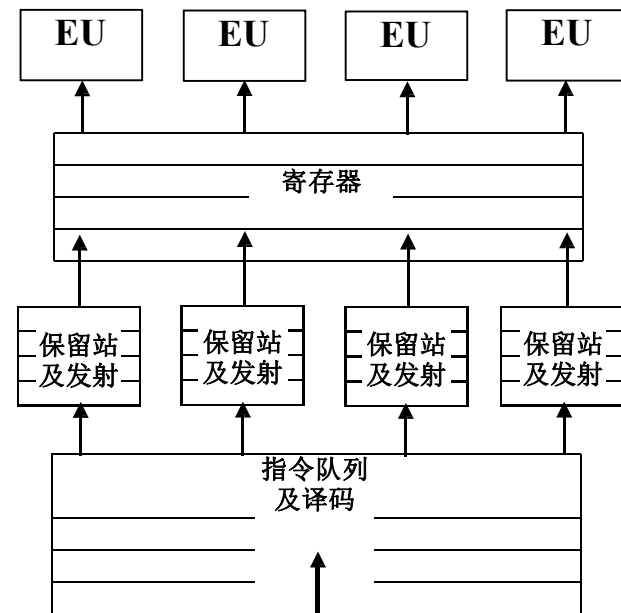
# 保留站前读寄存器

- 寄存器的输出作为保留站的输入
- 操作数没有准备好就读寄存器，保留站侦听结果总线获取没写回的值
- 有序读寄存器
- 保留站中值的来源：寄存器、重命名寄存器、进入保留站时侦听、进入保留站后侦听
- 保留站中有值域，因此较复杂
- 寄存器读端口数为发射宽度



# 保留站后读寄存器

- 保留站的输出作为寄存器的输入
- 保留站确信所有值都已经准备好后再读寄存器，有可能有Forward的情况
- 乱序读寄存器
- 保留站中没有值域，比较简单
- 寄存器读端口数为相应功能部件数





# 保留站前（后）读寄存器的处理器

保留站前读寄存器	保留站后读寄存器
IBM360/91 (1967)	CDC 6600 (1964)
PowerPC 603 (1993)	Power1 (1990)
PowerPC 604 (1993)	Power2 (1993)
PowerPC 605 (1993)	Lighting (1991)
PowerPC 620 (1996)	ES/9000 (1992)
Am 2900 sup. (1995)	Nx586 (1994)
Am K5 (1995)	PA8000 (1996)
PM-1 (Sparc64) (1995)	R10000 (1996)
Pentium Pro (1995)	Alpha 21264 (1997)
Intel Core (2006)	Intel Netburst (2001)
Godson-1	AMD K7/K8 (2006)
	Godson-2

- 似乎保留站后读寄存器占了上风

休息

# 寄存器重命名方法

# 寄存器重命名技术

- 双重作用
  - 例外或转移猜测错误时取消后面操作
  - 解决WAR和WAW相关
- 核心思想
  - 一个操作写寄存器时重命名到其他寄存器
  - 一个操作结束时再写到结构寄存器

# 重命名的方法

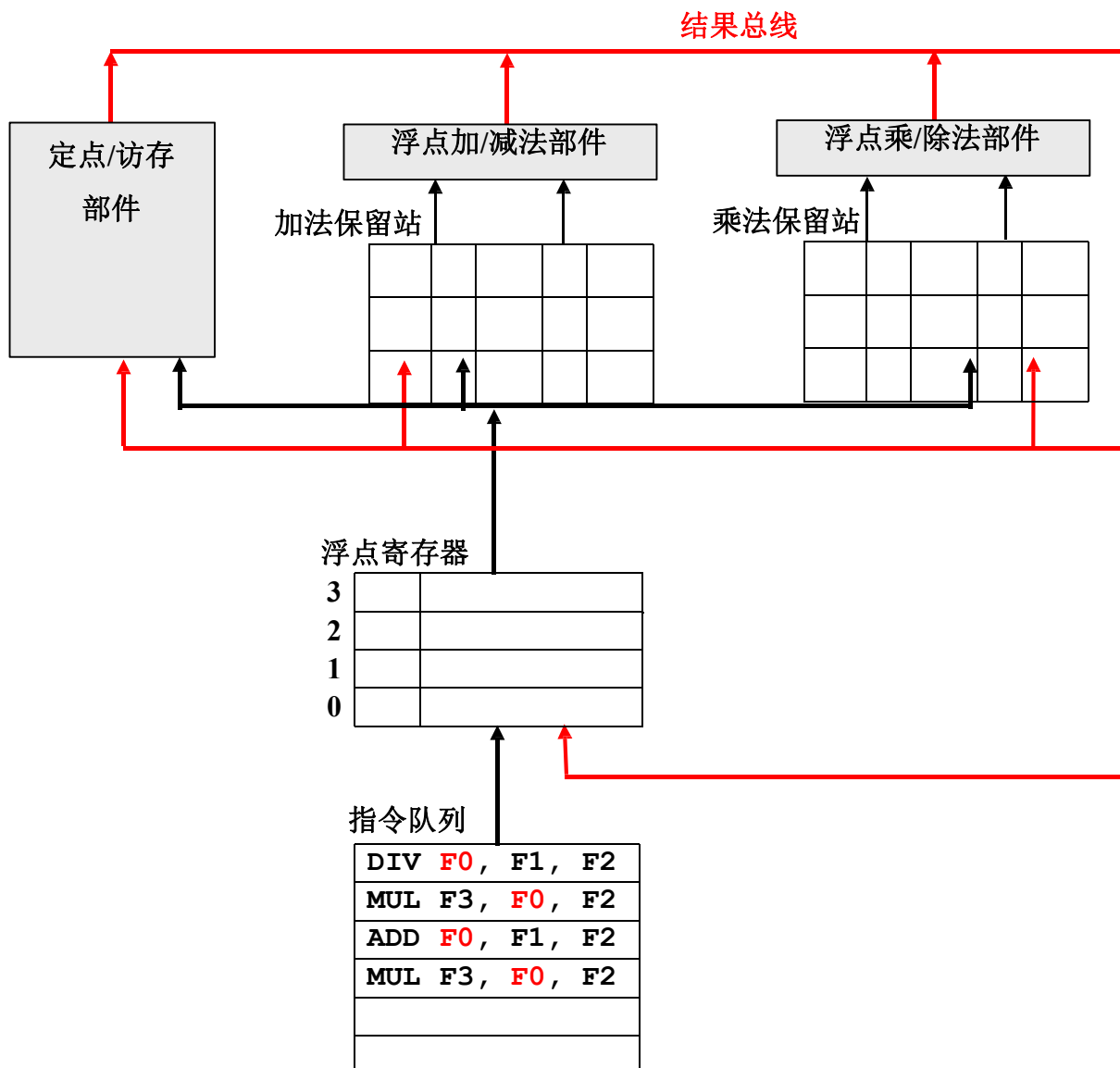
- 重命名的方法多种多样，不拘一格
  - 软件重命名
  - 重命名到保留站
  - 重命名到ROB
  - 重命名到发射队列
  - 建立物理寄存器到逻辑寄存器的映射
- 总之只要找到一个地方临时放一下数据

# 软件寄存器重命名

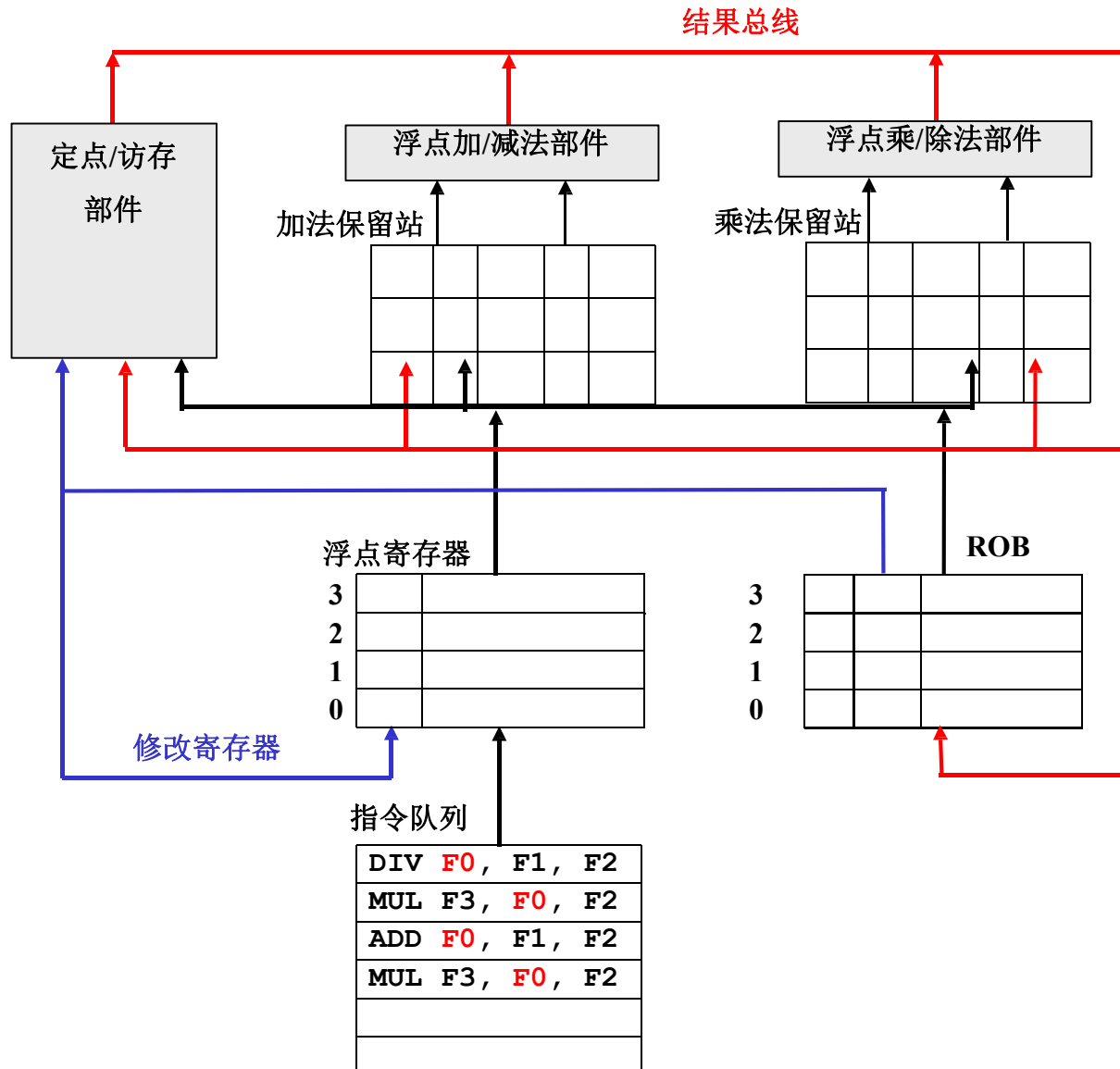
LD	<b>F0</b>	0	R1
MULTD	<b>F4</b>	<b>F0</b>	F2
SD	<b>F4</b>	0	R1
LD	<b>F0</b>	0	R1
MULTD	<b>F4</b>	<b>F0</b>	F2
SD	<b>F4</b>	0	R1

LD	<b>F0</b>	0	R1
MULTD	<b>F4</b>	<b>F0</b>	F2
SD	<b>F4</b>	0	R1
LD	<b>F6</b>	0	R1
MULTD	<b>F8</b>	<b>F6</b>	F2
SD	<b>F8</b>	0	R1

# 重命名到保留站

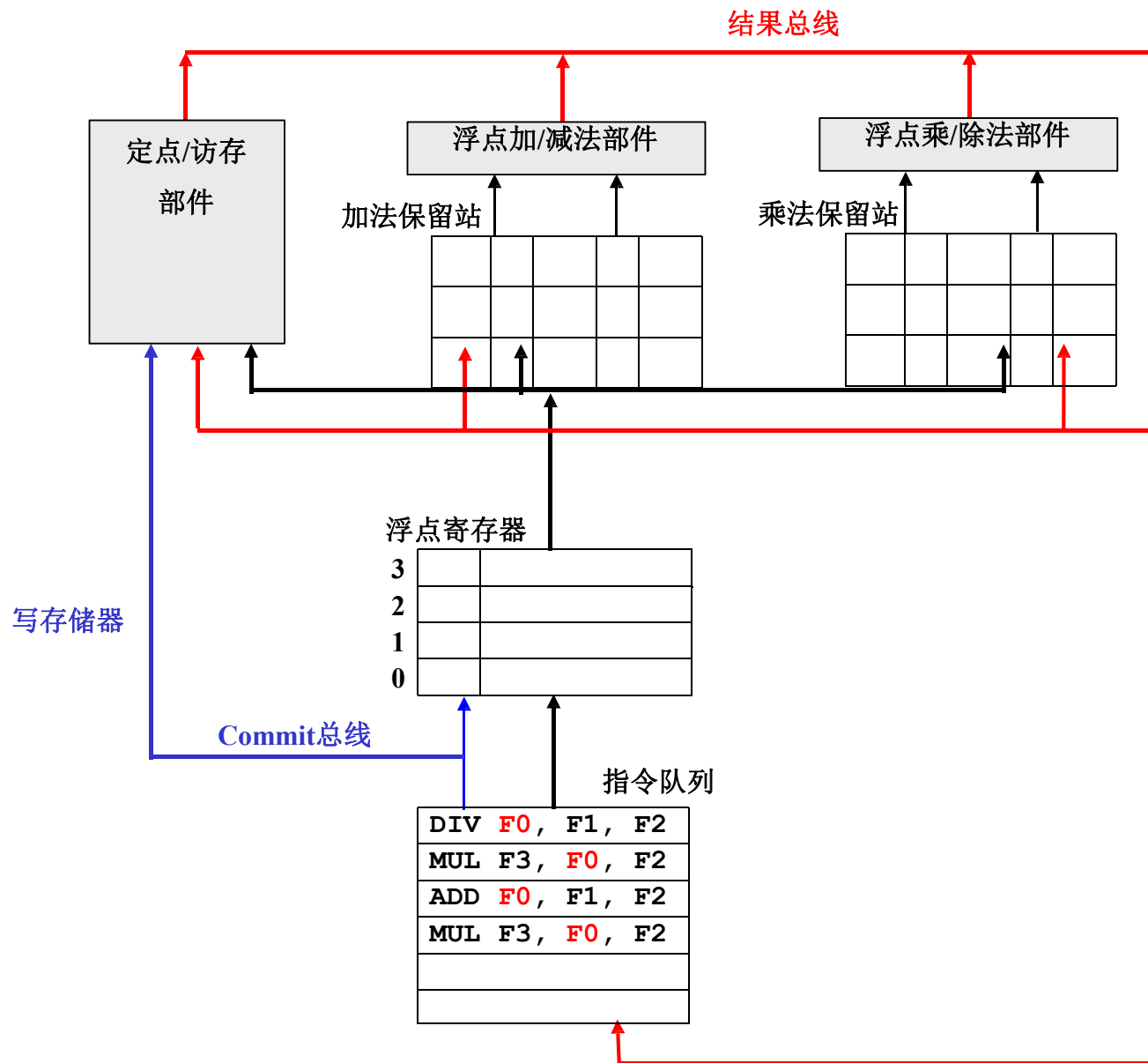


# 重命名到ROB





# 重命名到发射队列



# 建立逻辑寄存器到物理寄存器的映射

<b>ADD</b>	<b>R1</b>	<b>R1</b>	<b>1</b>
<b>ADD</b>	<b>R1</b>	<b>R1</b>	<b>1</b>
<b>ADD</b>	<b>R1</b>	<b>R1</b>	<b>1</b>
<b>ADD</b>	<b>R1</b>	<b>R1</b>	<b>1</b>
<b>ADD</b>	<b>R1</b>	<b>R1</b>	<b>1</b>
<b>ADD</b>	<b>R1</b>	<b>R1</b>	<b>1</b>

<b>ADD</b>	<b>PR32</b>	<b>PR1</b>	<b>1</b>
<b>ADD</b>	<b>PR33</b>	<b>PR32</b>	<b>1</b>
<b>ADD</b>	<b>PR34</b>	<b>PR33</b>	<b>1</b>
<b>ADD</b>	<b>PR35</b>	<b>PR34</b>	<b>1</b>
<b>ADD</b>	<b>PR36</b>	<b>PR35</b>	<b>1</b>
<b>ADD</b>	<b>PR37</b>	<b>PR36</b>	<b>1</b>

# 硬件重命名的分类

- 重命名寄存器和结构寄存器分开
  - 重命名到保留站、ROB、专门的重命名寄存器、发射队列
- 重命名寄存器和结构寄存器不分开
  - 为每个逻辑寄存器动态分配物理寄存器，需要建立逻辑寄存器和物理寄存器之间的映射表。
  - 映射表可以是RAM的方式，即有逻辑寄存器那么多项，也可以是CAM的方式，即有物理寄存器那么多项

# 独立重命名寄存器的重命名算法（1）

## 假设保留站前读寄存器

- 译码阶段
  - 为目标寄存器分配一个空闲的重命名寄存器
  - 为源寄存器找到相应的重命名或结构寄存器号
  - 读操作数（注意三种操作数来源）并送入保留站
- 发射阶段
  - 在保留站中找操作数准备好的操作进行运算，不涉及重命名
- 执行阶段
  - 执行结果写回到重命名寄存器
  - 写回到侦听该重命名寄存器值的保留站
  - 不写回结构寄存器

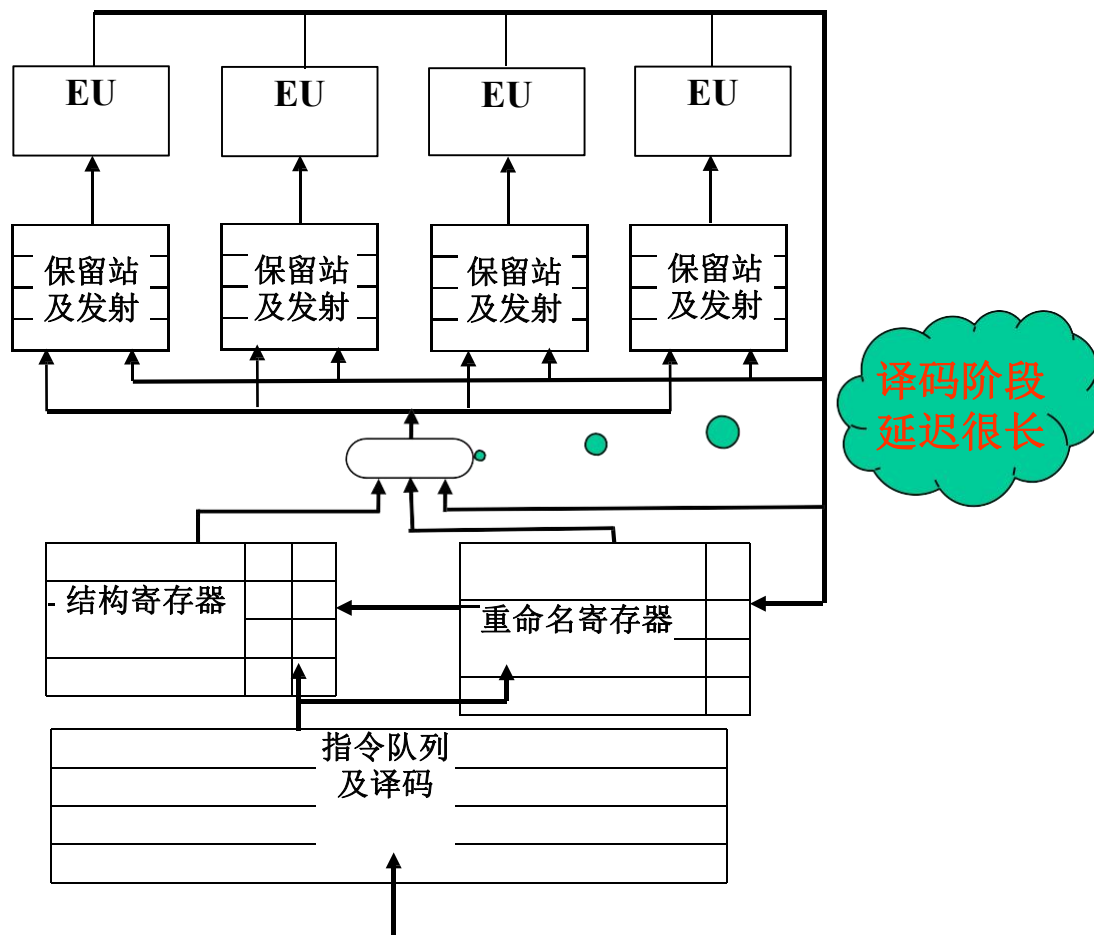
# 独立重命名寄存器的重命名算法（2）

- 提交阶段
  - 把重命名寄存器的值写回到相应的结构寄存器
  - 释放相应的重命名寄存器
- 转移猜错或例外
  - 取消后面的已经建立的重命名关系
  - 把结构寄存器的状态都置为有效
  - 把重命名寄存器的状态都置为空

# 独立重命名寄存器的重命名算法（3）

- 重命名寄存器的状态
  - **EMPTY**: 表示该寄存器没有被重命名（重命名后又已经被释放）
  - **MAPPED**: 表示已经被重命名但结果没有写回
  - **WRITEBACK**: 表示结果已经写回重命名寄存器但没有Commit到结构寄存器
- 结构寄存器的状态
  - **VALID**: 表示相应寄存器的值可用
  - **INVALID**: 表示相应寄存器的值不可用
- 映射关系
  - 可在结构寄存器中增加一个指向重命名寄存器的重命名寄存器号域

# 独立重命名寄存器的重命名算法（4）



# 使用物理寄存器堆的重命名算法（1）

## 假设保留站后读寄存器

- 译码阶段
  - 把目标寄存器映射到一个空闲的物理寄存器
  - 为源寄存器根据依赖关系找到相应的物理寄存器号
  - 把重命名后的寄存器号写入保留站
- 发射阶段
  - 判断所需的操作数是否已经准备好，或（Forwarding情况下）正在写回
  - 从物理寄存器或结果总线中读寄存器的值
- 执行阶段
  - 执行结果根据目标物理寄存器号写回到物理寄存器
  - 不用写回到保留站



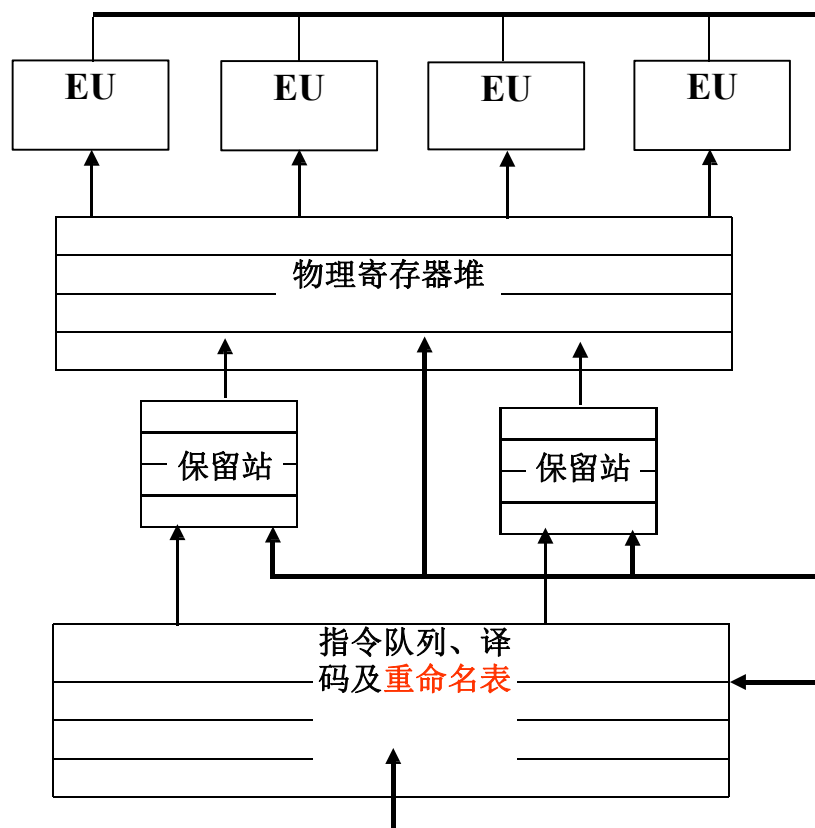
# 使用物理寄存器堆的重命名算法（2）

- 提交阶段
  - 修改重命名表确认目标寄存器的重命名关系
  - 释放老的目标寄存器重命名关系
- 转移猜错或例外
  - 修改重命名表取消后面的已经建立的重命名关系
  - 把状态为**MAPPED**或**WRITEBACK**的都置为**EMPTY**
  - 确认状态为**COMMIT**的映射为相应逻辑寄存器的最新映射

# 使用物理寄存器堆的重命名算法（3）

- 核心是重命名表
  - 可以用CAM或RAM的方法，以CAM的方法为例
  - 项数与物理寄存器一样
- 主要包括三个域
  - **name**: 相应的逻辑寄存器号
  - **state**: 状态，**EMPTY**: 表示该寄存器没有被重命名（重命名后又已经被释放）；**MAPPED**: 表示已经被重命名但结果没有写回；**WRITEBACK**: 表示结果已经写回重命名寄存器但没有Commit到结构寄存器；**COMMIT**: 表示结果已经被确认
  - **valid**: 在一个逻辑寄存器对应多个物理寄存器的情况下表示最新映射

# 物理寄存器堆的重命名算法（4）



# 不同处理器的重命名方法

使用物理寄存器堆	单独的重命名寄存器	
	专设重命名寄存器	重命名到ROB
Power1 (1990) Power2 (1993) ES/9000 nX586 PM1 (Sparc64, 1995) R10000 R12000 Alpha 21264 (1998) AMD K8 FP Godson-2 (2003)	PowerPC 603 (1993) PowerPC 604 (1995) PowerPC 620 (1996) Power3 (1998) PA8000 PA8200 PA8500 AMD K8 Int	Am29000 (1995) K5 (1995) M1 (1995) K6 (1997) Pentium Pro (1995) Pentium II (1997) Pentium III (1999) Intel Core Godson-1

- **UltraSparc**没有使用寄存器重命名

# 常见处理器的数据通路

# 乱序执行的数据通路

- 三方面考虑
  - 保留站结构（独立、分组、全局）
  - 读操作数时机（译码、发射）
  - 重命名方式（重命名寄存器与结构寄存器分开、不分开）
- 因此一共有12种组合

# 数据通路类型

	单项保留站	分组保留站	全局保留站
保留站前读操作数 独立重命名寄存器	PPC603, PPC604, PPC620, K5, Godson-1		
保留站前读操作数 物理寄存器堆			
保留站后读操作数 独立重命名寄存器		PA-8700	Pentium Pro
保留站后读操作数 物理寄存器堆		ES-9000, Power4, Alpha 21264, MIPS R10000, Godson-2	Intel lake 系列处理器

- 保留站后读操作数可以减少一次源操作数拷贝（从寄存器到保留站）
- 物理寄存器堆可以减少一次结果操作数拷贝（从物理寄存器到结构寄存器）

# 常见处理器的数据通路（早期）

CPU	保留栈/队列/ROB	寄存器	运算部件
龙芯2号	In-flight window (64) Int. issue queue (16) FP issue queue(16)	Int. regfile (64 4w8r) FP regfile (64 4w8r)	arith./logic, shift/branch add/logic, shift/mult/div load/store FP madd/div/sqrt/comp/media/ FP madd/
ALPHA 21264	In-flight window (80) Int. issue queue (20) FP issue queue(15)	2 Int. regfile (80 4r6w) FP regfile (72, 4r4w)	arith./logic, shift/branch, mult. add/logic, shift/branch, MVI/PLZ arith./logic load/store arith./logic load/store FP add, div, sqrt FP mult.
MIPS R10000	Integer queue(16) Address queue(16) FP queue(16)	Int. regfile (64 7r3w) Cond. file (64x1 2r3w) FP regfile(64 5r3w)	arith./logic, shift/branch arith./logic, mult/div FP add, sub, comp, conversion FP mult, div, sqrt load/store
HP PA-8700	ALU Reorder Buffer(28) Memory Reorder Buffer(28)	Int. arch. reg. (32, 4w8r) Int. ren. reg. (56, 4w9r) FP arch. reg. (32, 4w8r) FP ren. reg. (56, 4w9r)	2 arith./logic units 2 shift merge units 2 FP MAC units 2 FP div/sqrt units 2 load/store units
ULTRA SPARC-III	Instruction queue(20) Miss queue(4)	Int reg (144 3w7r) FP regfile (32, 5r4w)	2 arith., logic, shift branch load/store FP adder, graphic FP div/sqrt, mult, graphic
POWER 4	fixed point & load/store (4x9) floating-point (4x5) Branch execution(1x12) CR logical(2x5) in flight win(200=20x5)	GPRS (80) FPRS (72) CRS (32) Link/count (16) FPSCR(20) XER(24)	2 fixed-point 2 floating-point 2 load/store 1 branch 1 CR



# 常见处理器的数据通路（最近）



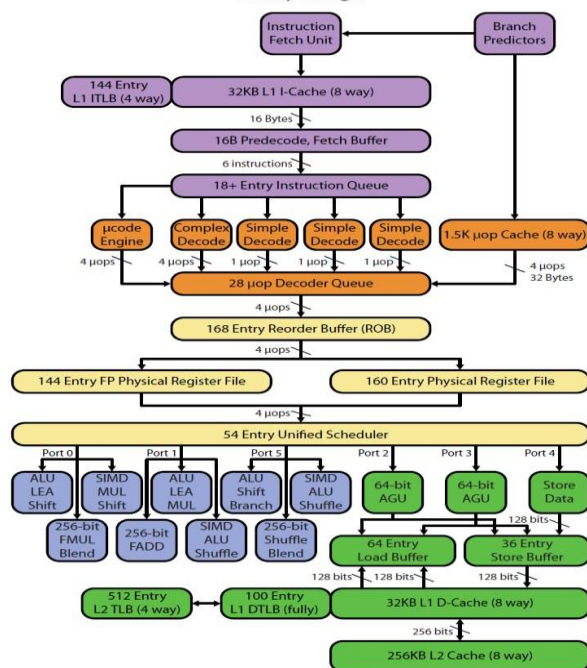
智能计算机研究中心  
Research Center for Intelligent Computing Systems

	Intel Sandybridge	AMD BullDozer	IBM Power7	
<b>前端</b>				
取指宽度	16 字节/时钟周期	32 字节/时钟周期	8 指令/时钟周期	8 指令/时钟周期
一级指令缓存	32KB, 8 路, 64B/行	64KB, 2路, 64B/行	32KB, 4 路, 128B/行	64KB, 4 路, 64B/行
指令TLB	128 项, 4 路, L1 ITLB	72 项, 全相联, L1 ITLB 512 项, 4 路, L2 ITLB	64 项, 2 路, L1 ITLB	64 项, 全相联, L1 ITLB
分支预取	BTB (8K-16K?项) 间接目标队列 (? 项) RAS (? 项) 循环检测	512 项, 4 路 L1 BTB 5120 项, 5 路 L2 BTB 512 项 间接目标队列 24 项 RAS 循环检测	8K 项本地 BHT 队列, 16K 项全局BHT 队列, 8K 项 全局 sel 队列 128 项间接目标队列 16 项 RAS	8K项本地 BHT 队列 8K项全局BHT 队列 8K项 全局 sel 队列 128项间接目标队列 16项 RAS 循环检测
<b>乱序执行</b>				
Reorder缓存	168 项	128 项	120 项	128 项
发射队列	54-项 统一	60-项 浮点(共享) 40-项 定点, 访存	48-项 标准	32-项 浮点; 32-项 定点; 32-项 访存
寄存器重命名	160 定点; 144 浮点	96 定点; 160 浮点(双核共享)	80 定点,浮点; 56 CR; 40 XER, 24 Link&Count	128 定点; 128 浮点/向量; 16 Acc; 32 DSPCtrl; 32 FCR
<b>执行单元</b>				
执行单元个数	ALU/LEA/Shift/128位 MUL/128位 Shift/256位FMUL/256位Blend + ALU/LEA/Shift/128位 ALU /128bit Shuffle/256位 FADD + ALU/Shift/Branch/ 128位 ALU/128bit Shuffle/256 位 Shuffle/256位 Blend	ALU/IMUL/Branch + ALU/IDIV/Count + 128位 FMAC/128位 IMAC + 128位FMAC/128位 XBAR + 128位 MMX + 128位 MMX/128位 FSTO	2 定点 + 2 浮点/向量 + 1 转移 + 1 CR	2 定点/转移/DSP + 2 浮点/向量
向量部件宽度	256位	128位	128位	256位

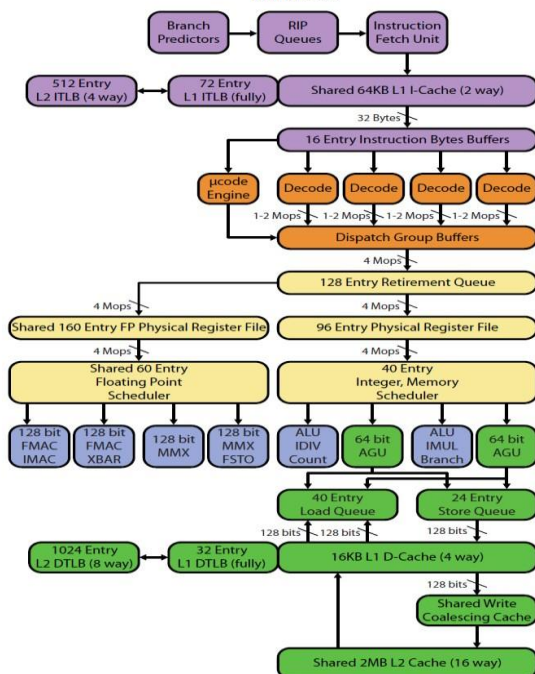
# 常见处理器的数据通路（最近）

	Intel Sandybridge	AMD Bulldozer	IBM Power7	GS464E
访存执行单元	2 取+ 1 存	2 取/存	2 取/存	2 取/存
Load/Store队列	64-项 Load 队列, 36-项 Store队列	40-项 Load 队列, 24-项 Store 队列	32-项 Load队列, 32-项 Store 队列	64-项 取/存 队列
Load/Store 宽度	128 位	128 位	256 位 load, 128 位 Store	256 位
TLB	100项全相联, L1 DTLB, 512项4路, L2 TLB	32项全相联, L1 DTLB, 1024项8路, L2 TLB	64项全相联, L1 DTLB, 512项4路, L2 TLB	256-项全相联, TLB 每项两页
一级数据缓存	32KB, 8 路, 64B/行	16KB, 4路, 64B/行	32KB, 8 路, 128B/行	64KB, 4 路, 64B/行
二级缓存	每核256KB, 8路, 64B/ 行	双核共享2MB, 16 路	每核256KB, 8路, 28B/行	每核256KB, 16路, 64B/行
三级缓存	8个核20 MB	4个核8 MB	8个核32 MB	4个核4 MB~8MB
一级数据缓存同时处理 缺失请求数	10	?	8	Unified 16
二级缓存同时处理缺失 请求数	16	23	24	
一级 Load-to-use 延 迟	定点4时钟周期, 浮点/向量5时钟周期	4时钟周期	定点2时钟周期, 浮点/向量3 时钟周期	定点3时钟周期, 浮点/向量5 时钟周期
二级 Load-to-use 延 迟	12 时钟周期	18-20 时钟周期	8	17 时钟周期

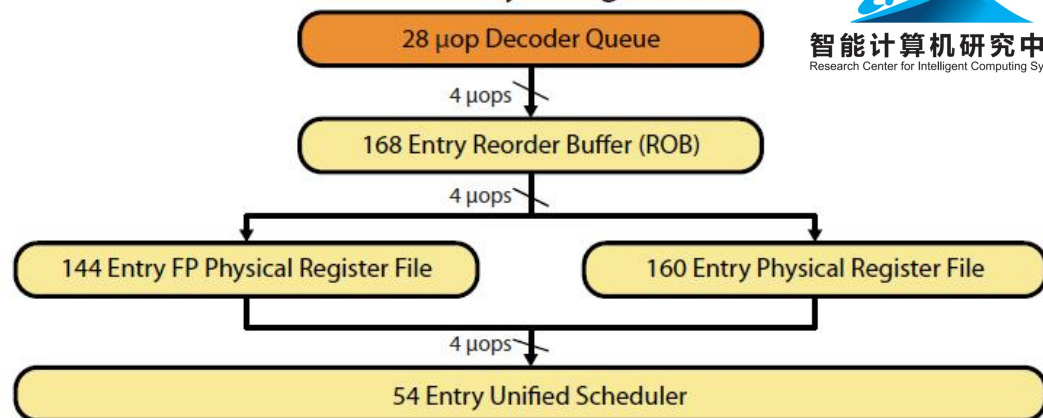
Sandy Bridge



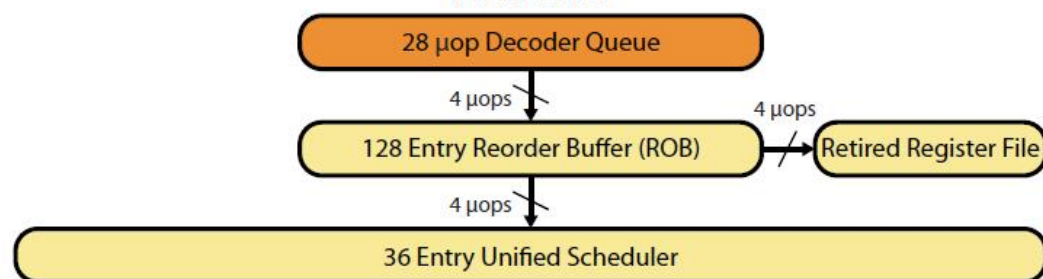
Bulldozer



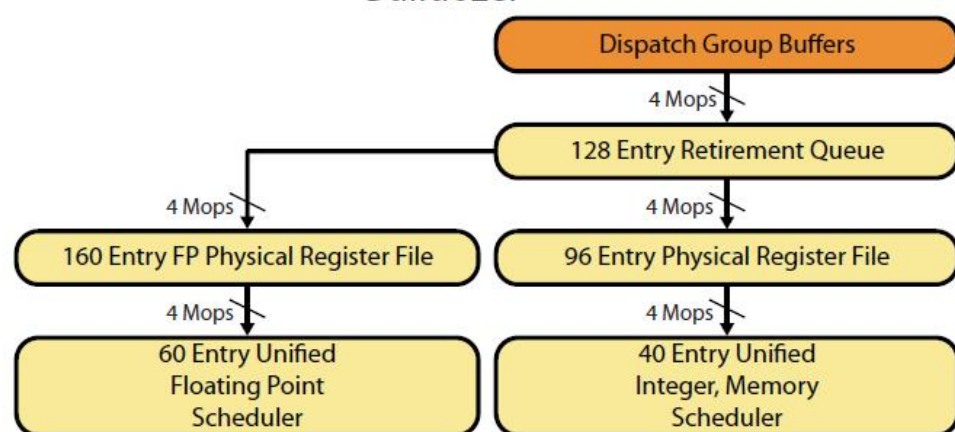
Sandy Bridge



Nehalem










Bulldozer



# Intel 最近处理器的队列大小

## Haswell Buffer Sizes

Extract more parallelism in every generation

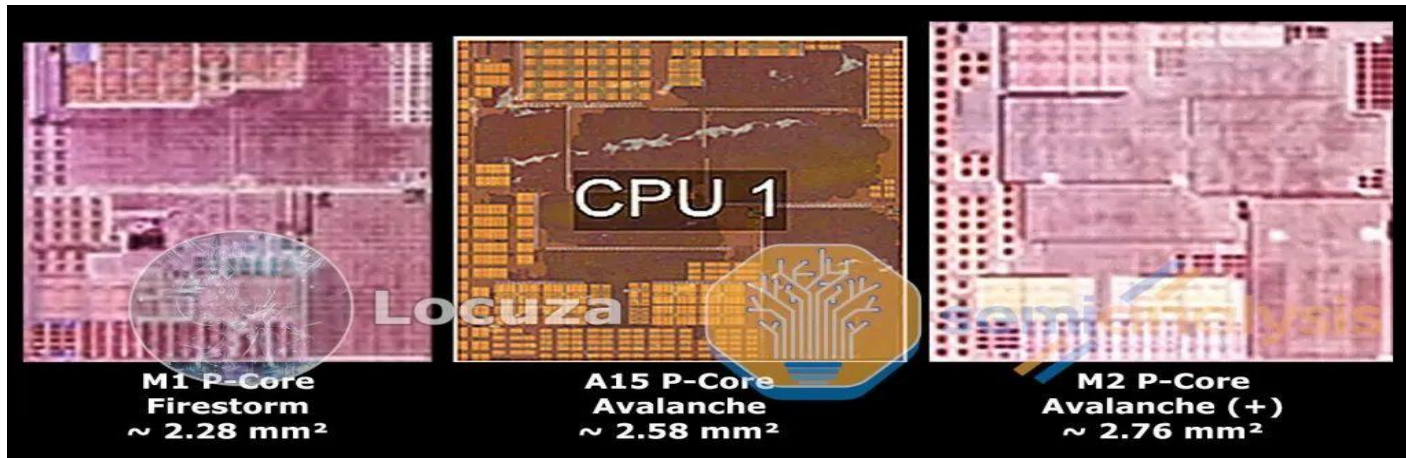
	Nehalem	Sandy Bridge	Haswell	
Out-of-order Window	128	168	192	
In-flight Loads	48	64	72	
In-flight Stores	32	36	42	
Scheduler Entries	36	54	60	
Integer Register File	N/A	160	168	
FP Register File	N/A	144	168	
Allocation Queue	28/thread	28/thread	56	

休 息

# 多发射结构数据通路

## ROB越大就越好吗？

一个非常有趣的变化是，A15 和 M2 中 Avalanche 核心的 ROB 与 M1 和 A14 中 Firestorm 核心相比显得更小。这一点特别有趣，因为为了获得业界最宽、最高的 IPC 内核，苹果拥有业界最大的 ROB？



# 多发射的情况

- 数据通路变宽
  - Alpha21264: 取指4条, 发射6条, 写回6条, Commit 11条
  - 寄存器读端口变多
  - 访存端口要求也增加, 如Alpha有两个访存部件, 通过倍频实现
- 同一拍发射的指令之间的相关
  - 如在重命名阶段发生同一拍之间的相关
- 流水线复杂度与发射宽度成平方关系
  - 如发射队列 (保留站) 变大\* (读端口变多+侦听端口变多)
  - 如寄存器重命名端口变多\*同一拍重命名指令相关



# 多发射时的寄存器重命名

- 假设开始时与逻辑寄存器R1,R2,R3对应的最近的物理寄存器分别为PR1,PR2,PR3, 在同一拍中对如下指令进行寄存器重命名:

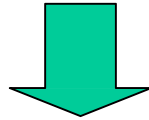
**ADD R1, R1, R1**

**ADD R1, R1, R1**

**ADD R1, R1, R1**

**ADD R1, R1, R1**

```
add r01,r01,r01  
add r01,r01,r01  
add r01,r01,r01  
add r01,r01,r01
```



```
add pr32,pr01,pr01  
add pr33,pr01,pr01  
add pr34,pr01,pr01  
add pr35,pr01,pr01
```

发生错误



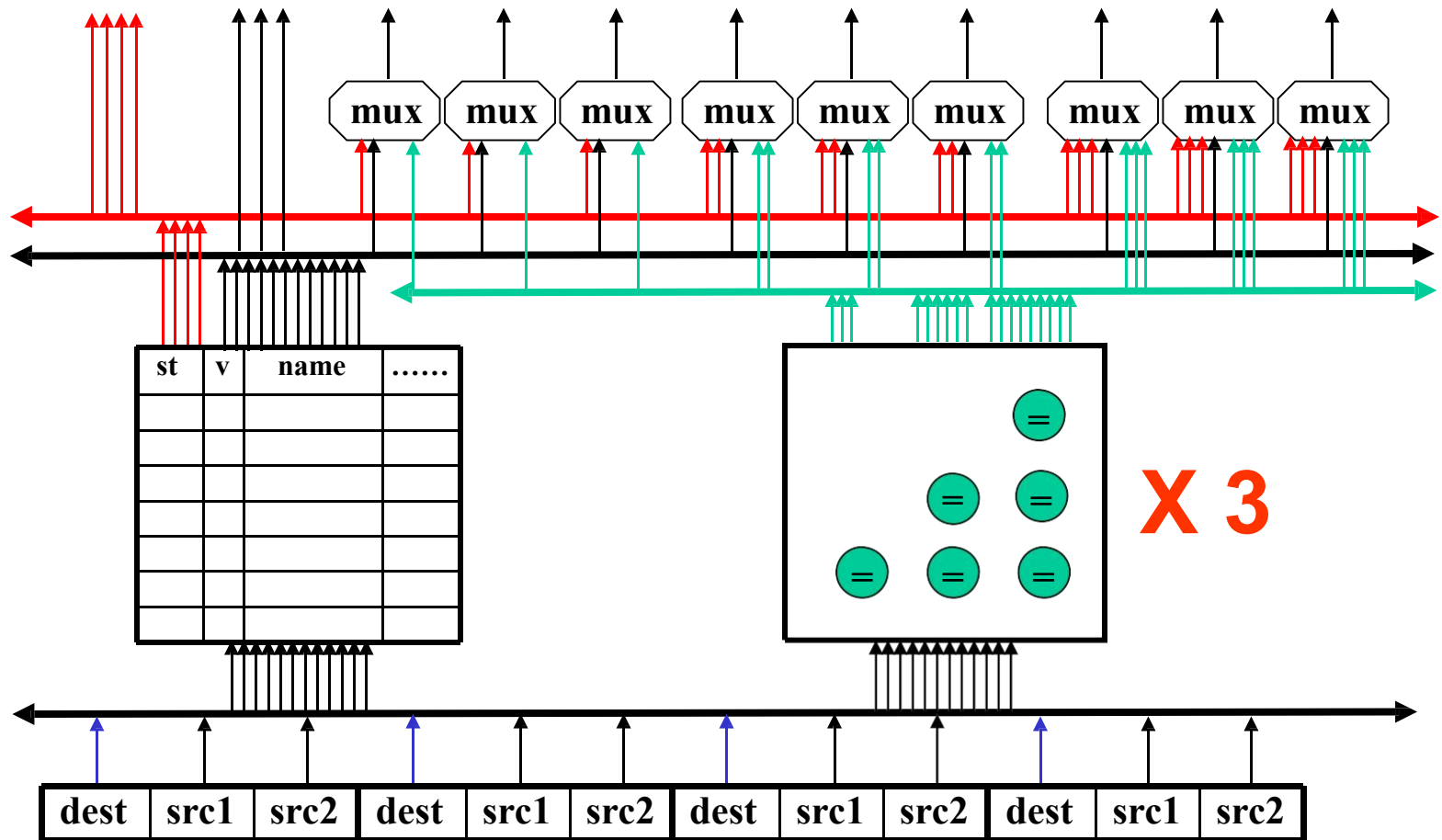
```
add pr32,pr01,pr01  
add pr33,pr32,pr32  
add pr34,pr33,pr33  
add pr35,pr34,pr34
```

正确

# 多发射的其它设计问题

- 同时提交多条指令
  - 重命名把有序变成无序，要除了“前后看”，还要“左右看”
  - 提交把无序变成有序，也要“左右看”：同一拍可以提交多条，第二条是否提交要看第一条是否提交，等等
- 指令发射和读寄存器
  - 多端口寄存器是物理设计的难点：会增加延迟、面积、功耗，四发射结构一般至少需要四写八读的寄存器堆
  - 指令发射逻辑：从保留站中同时找出多条数据准备好的指令，为了性能要适当考虑在数据准备好的情况下前面的指令先发射
- 多功能部件
  - 增加回端口：寄存器堆、重命名寄存器表、ROB、发射队列
  - 尤其是多访存部件会大大增加设计复杂度

# 同时发射多条指令结构



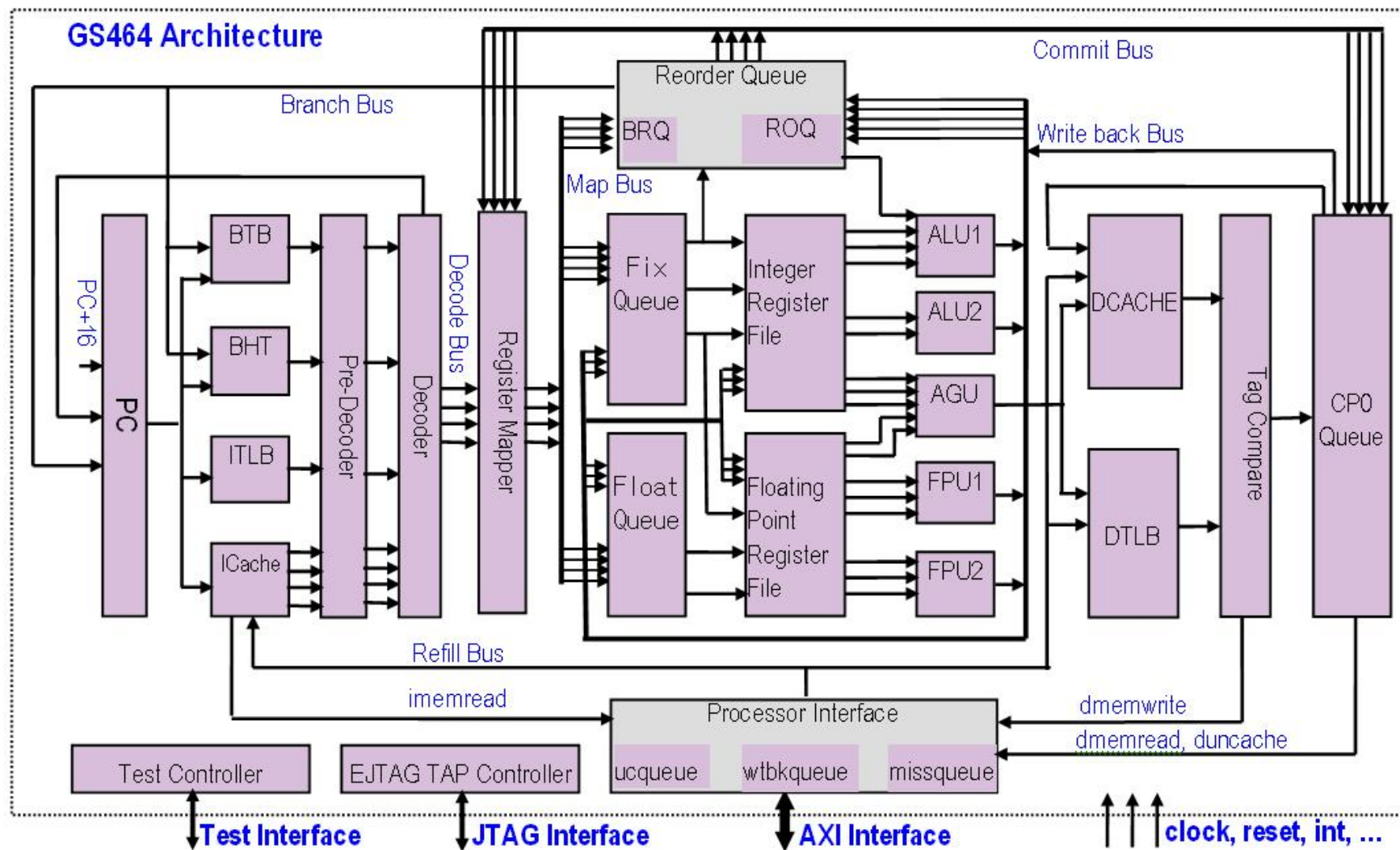
重命名逻辑随发射宽度的平方复杂度！

# 龙芯2号多发射结构简介

# 龙芯2号数据通路

- 组保留站
  - 定点、浮点
- 保留站后读操作数
- 物理寄存器堆

# 龙芯2号处理器核结构图



# 龙芯2号指令流水线

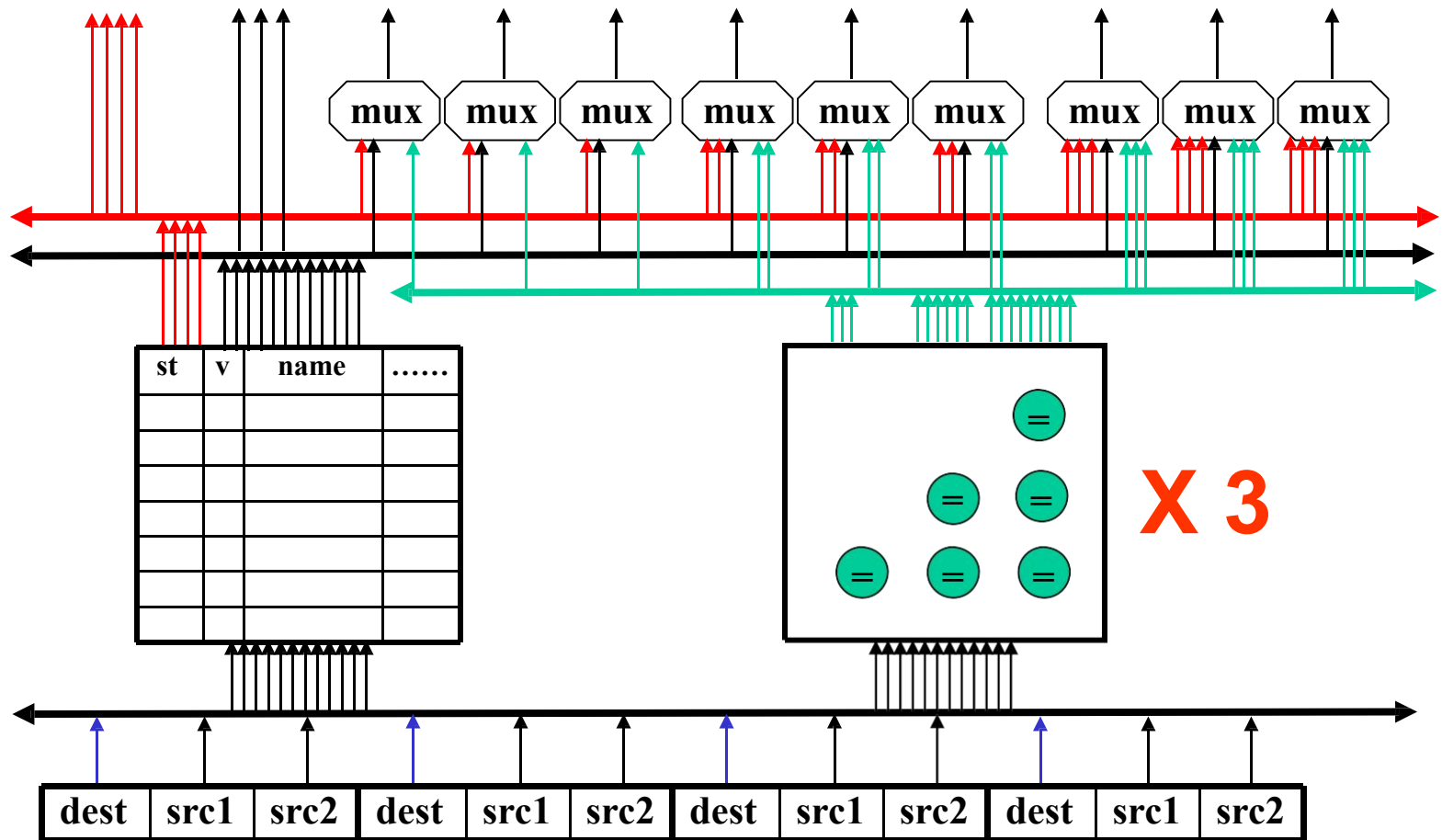
- **取指：**根据程序计数器（PC）来访问指令Cache和指令TLB，如果TLB和Cache命中，就从指令Cache中取四条指令到指令寄存器
- **预译码：**对分支指令进行转移猜测
- **译码：**指令寄存器里的四条指令被译码成处理器内部指令格式
- **寄存器重命名：**为每个逻辑目标寄存器分配一个物理寄存器，并且将逻辑源寄存器重命名为该寄存器最新映射的物理寄存器
- **调度：**重命名后的指令被调度到定点或者浮点保留站中等待执行，同样也送到重排序队列（ROQ）中以保证按序完成，分支指令和访存指令还分别发送到分支队列和访存重排序队列
- **发射：**从定点或浮点保留站中选取操作数都准备好的指令并发射，操作数没有准备好的指令在保留站侦听结果总线和前递总线
- **读寄存器：**已发射的指令从物理寄存器堆中读取源操作数并发送到相应的功能单元
- **执行：**根据指令要求进行运算，运算结果写回寄存器堆，并通知保留站和寄存器重命名表相应的物理寄存器已经写回，执行阶段可能需要多拍
- **提交：**按照ROQ记录的程序顺序提交执行完的指令，每拍可提交四条指令，提交的指令送往寄存器重命名表用于确认它的目的寄存器的重命名关系并释放原来分配给同一逻辑寄存器的物理寄存器，送往访存队列允许那些提交的存数指令写入Cache 或内存



# 寄存器重命名

- 功能
  - 把逻辑寄存器号转化为物理寄存器号
  - 消除WAR和WAW相关，建立RAW相关关系。
- 使用CAM寄存器重命名表，每项包括如下内容
  - **State**: 四个状态EMPTY、MAPPED、WRITEBACK、COMMIT。
  - **Name**: 该物理寄存器对应的逻辑寄存器号。
  - **valid**: 在多个物理寄存器对应一个逻辑寄存器时表示哪个是最近被映射的。
  - 其他。
- 操作
  - 重命名: 为目标寄存器分配一个空闲物理寄存器（状态置为MAPPED），为源寄存器根据依赖关系找到相应的物理寄存器号。
  - 执行: 把重命名寄存器的状态置为WTBK。
  - 提交: 把目标寄存器的值变成软件可见（把重命名寄存器的状态置为COMMIT），释放原来的重命名寄存器（状态置为EMPTY）。
  - 转移猜错或例外: 取消后面的已经建立的重命名关系

# 重命名主要通路



重命名逻辑随发射宽度的平方复杂度！

# 发射队列

- 定点和浮点队列分开，相当于分组保留站
  - 操作寄存器重命名后进入发射队列，取指和译码过程中发生例外的操作不进入发射队列
- 乱序发射
  - 对于每个功能部件每次在已经准备好可以发射的操作中找最早进入队列的进行发射，发射队列中增加age域
- 数据准备好的判断
  - 寄存器重命名阶段根据物理寄存器状态表判断该物理寄存器是否可用；
  - 操作队列侦听有关有关功能部件的写回结果，把等待该结果的操作数置为有效

- 发射队列内容
  - **fu**: 执行该操作的功能部件号，全1表示空。
  - **roqid, brqid**: Reorder队列和转移队列索引，写回时用。
  - **op**: 操作码。
  - **pdest**: 物理目标寄存器号。
  - **psrc1, psrc2**: 物理源寄存器号。
  - **prdy1, prdy2**: 源寄存器准备好标志。
  - **taken**: 转移猜测结果，为1表示猜测跳转。
  - **imm**: 定点队列为16位立即数。
  - **fmt**: 辅助操作码，浮点队列为5位fmt。
  - **age**: 表示一个操作在操作队列中等待时间的长短。
  - .....

- 所有操作在重命名后到提交之前一直在ROQ中
  - 寄存器重命名后有序进入、乱序执行、有序结束
  - ROQ还保存部分执行结果：如evzoui、例外、JR等
- 操作
  - 重命名：每次按序把重命名后的操作送入Reorder队列
  - 写回：根据结果总线的roqid号修改相应操作的状态，执行过程中的例外写回ex和excode，浮点操作写回evzoui，JR和JALR写回PC。
  - 提交：每次按序提交四个已写回的操作，例外操作不提交
  - 形成例外总线：例外操作必须成为ROQ第一个操作才能形成例外总线

- **ROQ的内容**

- **state: EMPTY、MAPPED、BRWTBK和WTBK。**
- **op: 8位，操作码。**
- **odest: 7位，dest原来对应的物理寄存器**
- **pdest: 7位，物理目标地址寄存器。**
- **brqid: 转移队列号。**
- **evzoui: 6位，浮点操作的EVZOUi结果写到此域。**
- **ex: 1位，表示是否发生例外。**
- **excode: 6位，例外原因。**
- **brerr: 表示转移猜测错误。**
- **taken: 表示转移成功。**
- **c: 1位，用于保存浮点比较结果。**
- **bd: 1 位，表示是否为延迟槽操作。**
- **.....**

谢 谢