

# 计算机体系结构

胡伟武、汪文祥

# 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

# 影响指令流水线的因素

# 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         ADDD    F4,F0,F2
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

# 指令调度技术



# 编译器的静态调度

- 编译器分析程序中的相关性，并针对目标流水线进行代码优化，以避免程序执行时由于相关引起阻塞
  - 相关不一定引起阻塞，只要隔开足够远
  - 在一个流水线上引起阻塞，在另一个流水线上不一定引起阻塞，所以编译优化与机器有关

# 循环展开技术

- 例：向量的每个元素加常数

For (j=1; j<=1000; j++) x[j]=x[j]+s;

```

Loop:  LD    F0, 0(R1)      ;F0=vector element
        ADDD  F4, F0, F2    ;add scalar from F2
        SD    0(R1), F4    ;store result
        SUBI  R1, R1, 8     ;decrement pointer 8B (DW)
        BNEZ  R1, Loop     ;branch R1!=zero
        NOP                ;delayed branch slot
    
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0
Integer op	Integer op	0

- 每次循环需要9拍

```

1 Loop:  LD      F0, 0(R1)      ;F0=vector element
2          stall
3          ADDD   F4, F0, F2     ;add scalar in F2
4          stall
5          stall
6          SD     0(R1), F4      ;store result
7          SUBI   R1, R1, 8      ;decrement pointer 8B (DW)
8          BNEZ   R1, Loop      ;branch R1!=zero
9          stall                ;delayed branch slot

```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0
Integer op	Integer op	0

- 改变指令次序每次循环需要6拍
  - 注意把SD放在Delay Slot中，偏移量的变化

```

1 Loop:  LD      F0, 0(R1)
2          stall
3        ADDD   F4, F0, F2
4        SUBI   R1, R1, 8
5        BNEZ   R1, Loop      ;delayed branch
6        SD     8(R1), F4      ;altered when move past SUBI

```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0
Integer op	Integer op	0

- 循环展开4次：假设R1的值是4的倍数
  - 注意把SD偏移量的变化

```

1 Loop: LD      F0, 0(R1)
2      ADDD     F4, F0, F2
3      SD       0(R1), F4      ;drop SUBI & BNEZ
4      LD       F0, -8(R1)
5      ADDD     F4, F0, F2
6      SD       -8(R1), F4     ;drop SUBI & BNEZ
7      LD       F0, -16(R1)
8      ADDD     F4, F0, F2
9      SD       -16(R1), F4    ;drop SUBI & BNEZ
10     LD       F0, -24(R1)
11     ADDD     F4, F0, F2
12     SD       -24(R1), F4
13     SUBI     R1, R1, #32     ;alter to 4*8
14     BNEZ     R1, LOOP
15     NOP

```

- 寄存器重命名

- 每4个循环需要  $15 + 4 \times (1+2) = 27$  拍，每个循环 6.8 拍

```

1 Loop: LD      F0, 0(R1)
2      ADDD     F4, F0, F2
3      SD       0(R1), F4
4      LD       F0, -8(R1)
2      ADDD     F4, F0, F2
3      SD       -8(R1), F4
7      LD       F0, -16(R1)
8      ADDD     F4, F0, F2
9      SD       -16(R1), F4
10     LD       F0, -24(R1)
11     ADDD     F4, F0, F2
12     SD       -24(R1), F4
13     SUBI     R1, R1, #32
14     BNEZ     R1, LOOP
15     NOP

```

```

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

```

- 改变指令次序

- 注意把SD与SUBI交换次序时偏移量的变化
- 注意SD与LD交换次序不会影响正确性
- 每4个循环需要14拍，每个循环 3.5 拍

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

- 增加发射宽度
  - 循环展开5次
  - 定点和浮点并行
  - 5个循环需要12拍，每个循环2.4拍

```

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

```



# 软件调度与硬件调度

- 既然软件可以在保持程序正确性的前提下改变指令的执行次序来提高性能，硬件能不能做到呢？
  - 可以做到，但要有特定的机制保证程序行为不被改变
  - 关键是程序的数据和结构相关性得以保持
- 软件调度与硬件调度的比较
  - 软件调度的范围大，可以在上万条指令的范围内进行调度；硬件一般只能在百条指令的范围内进行调度
  - 硬件调度可以掌握一些软件编译时还不明确的相关性信息，尤其是对访存相关以及控制相关

# 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

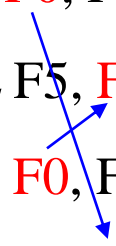
SUBD f8,f8,f14

- 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



# 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
  - 设计目标: 编译器在360系列中通用
- 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

# Tomasulo算法结构

## 保留站内容

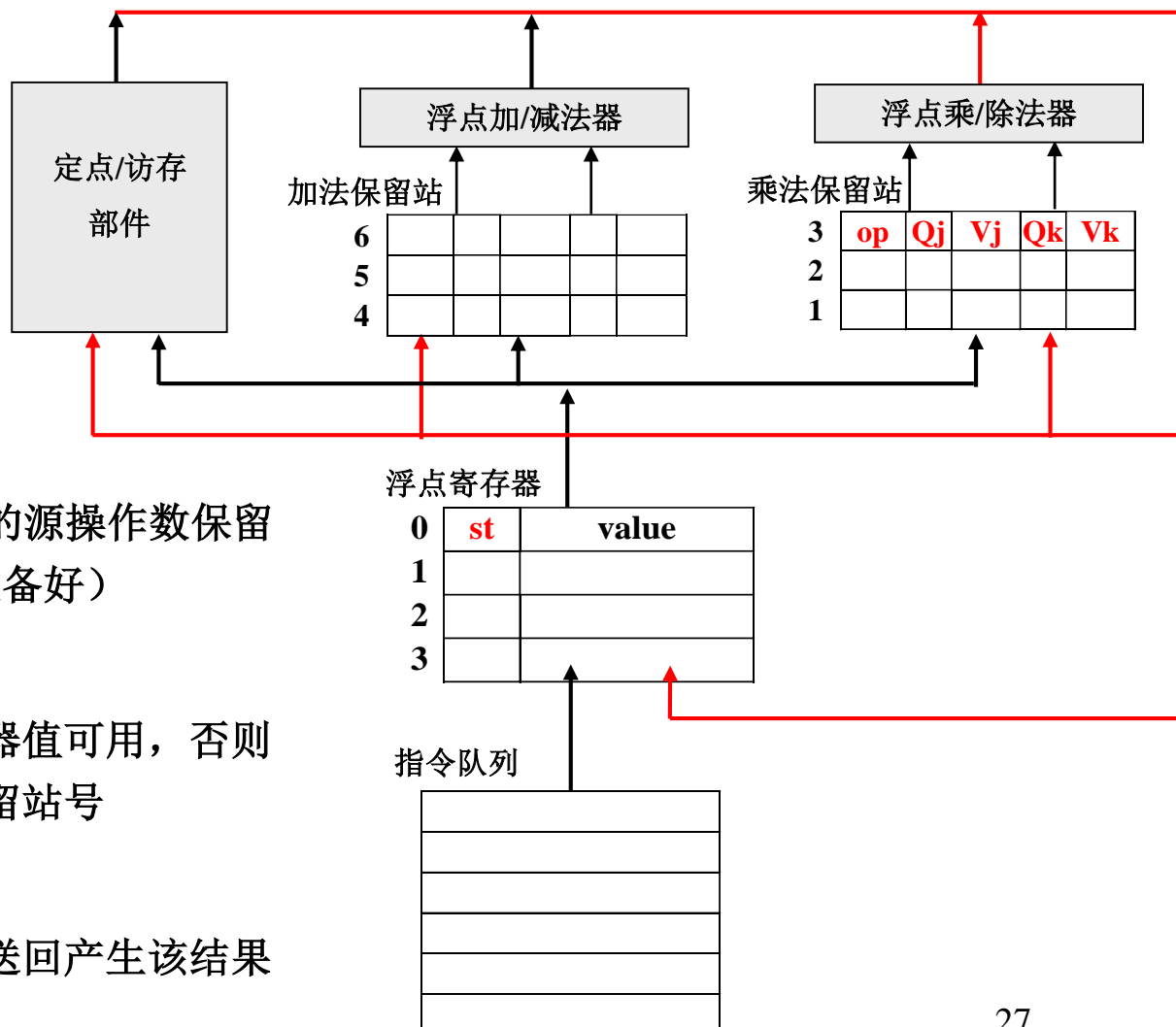
- Busy: 忙位
- Op: 操作码
- $V_j, V_k$ : 源操作数的值
- $Q_j, Q_k$ : 保存没有准备好的源操作数保留站号 (0表示操作数已经准备好)

## 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

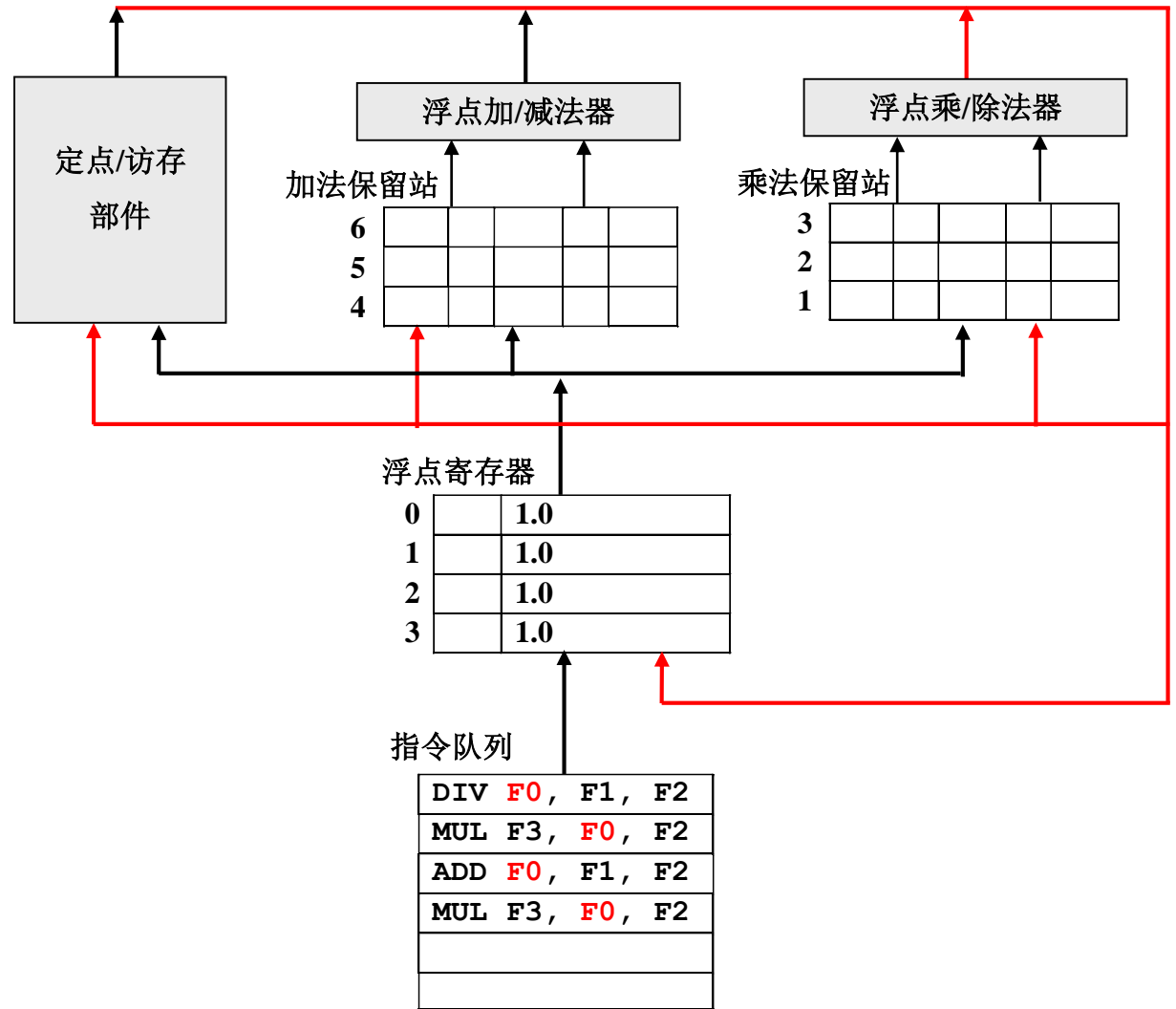
- 除了送回结果值外, 还要送回产生该结果的结果保留站号

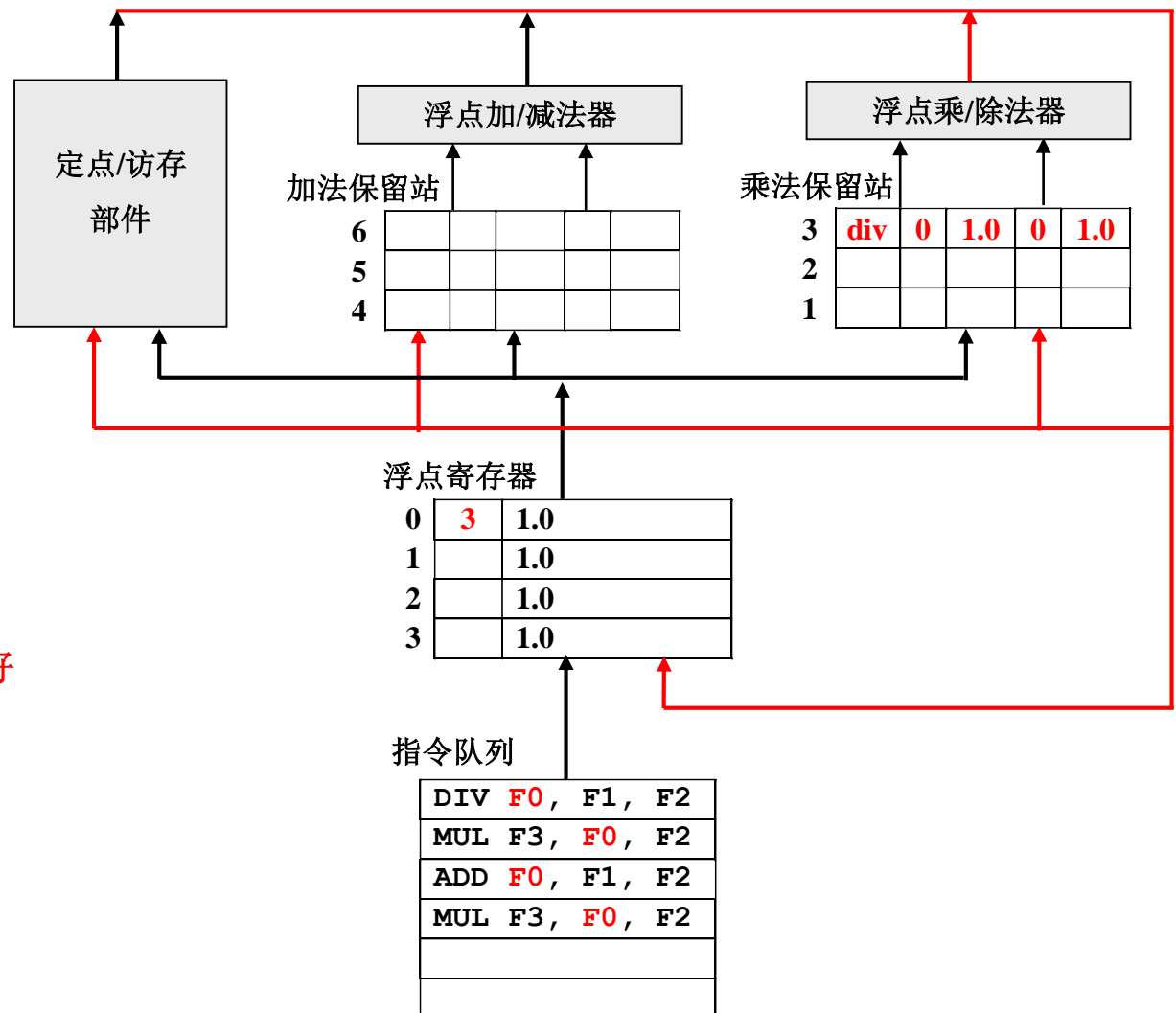


# Tomasulo算法的流水阶段

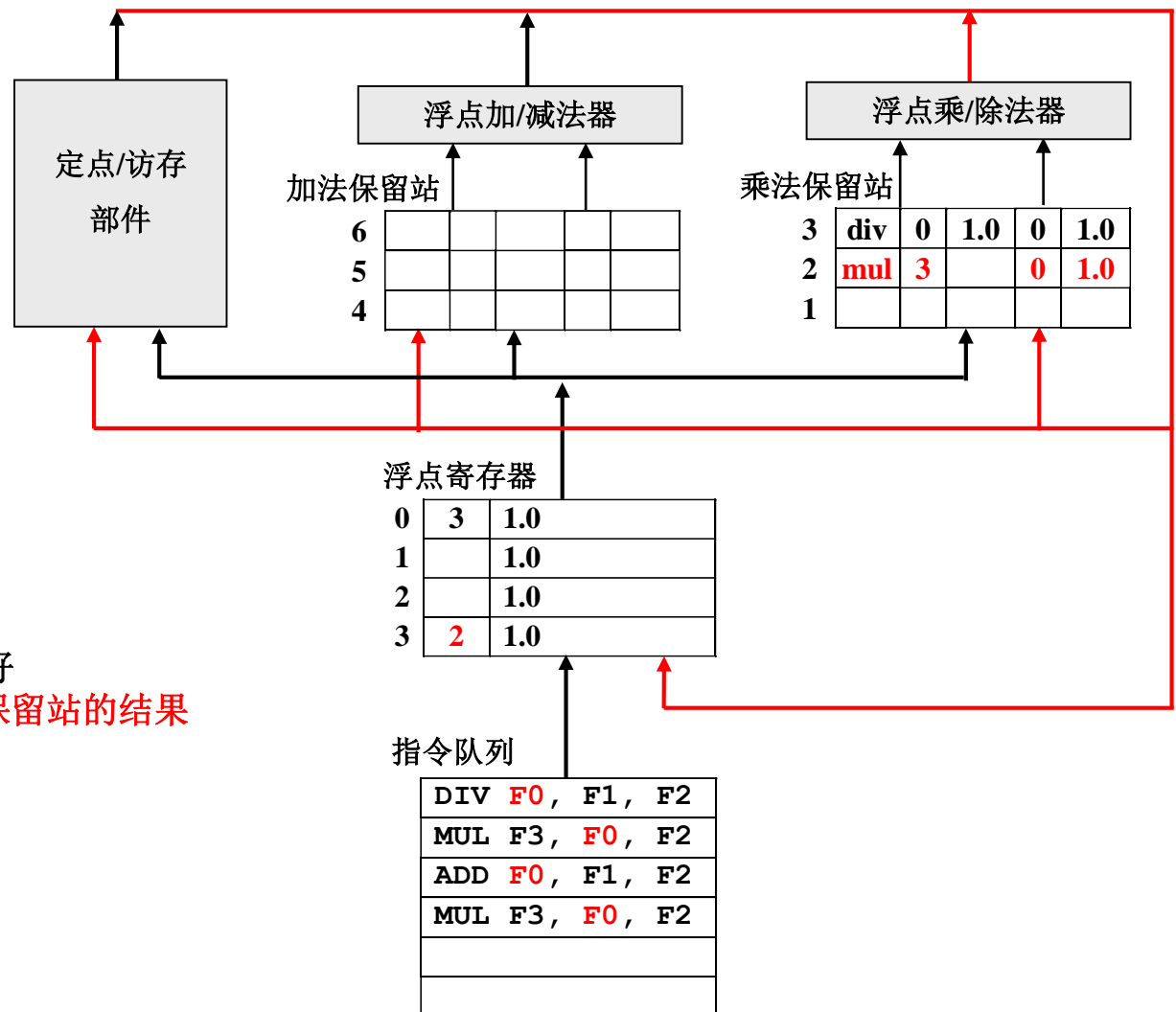
- 发射：把操作队列的指令根据操作类型送到保留站（如果保留站有空），发射过程中读寄存器的值和结果状态域
- 执行：如果所需的操作数都准备好，则执行，否则侦听结果总线并接收结果总线的值。
- 写回：把结果送到结果总线，释放保留站

# 例子

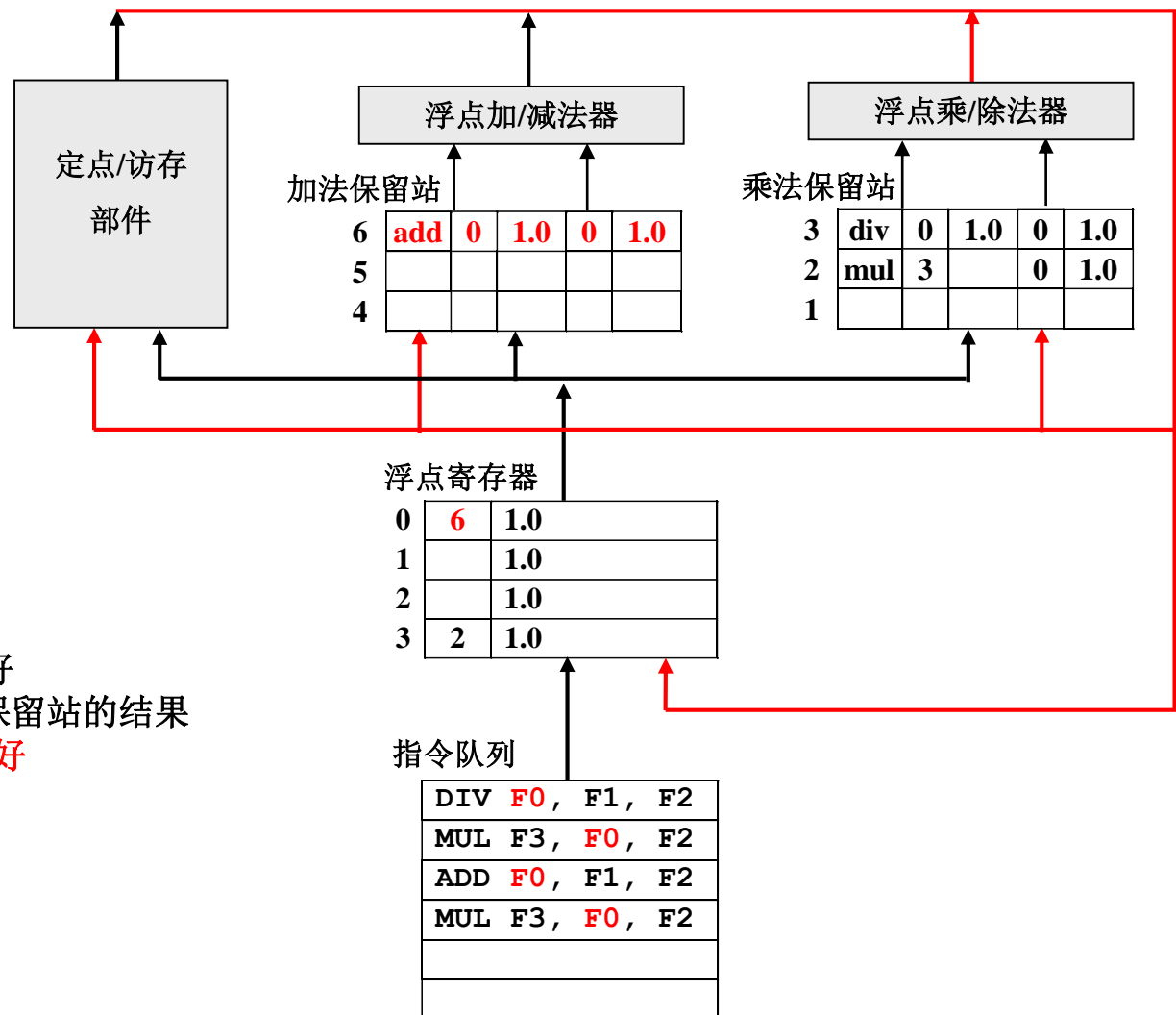




•DIV发射，F1,F2都准备好

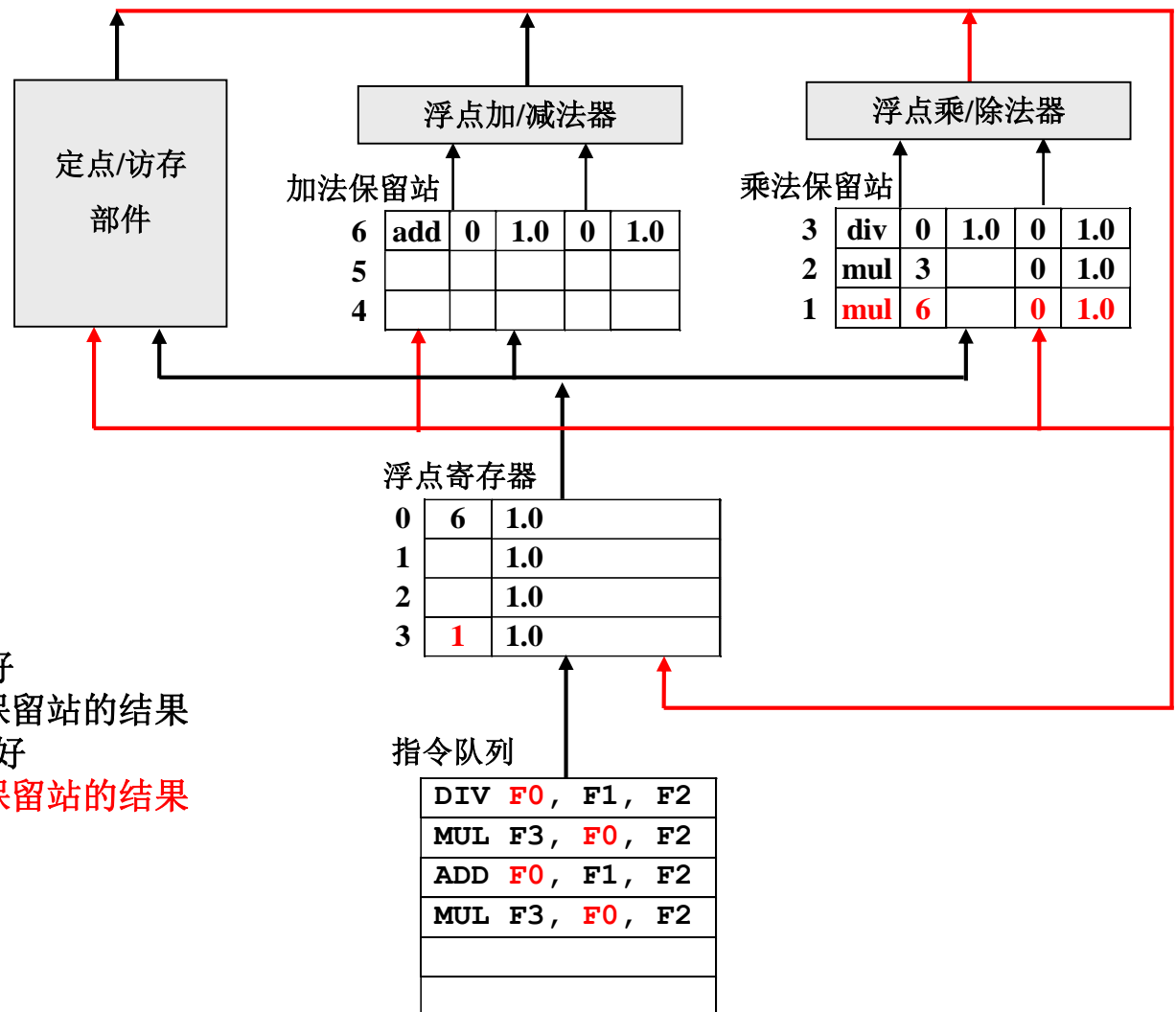


- DIV发射，F1,F2都准备好
- MUL1发射，F0等待3号保留站的结果

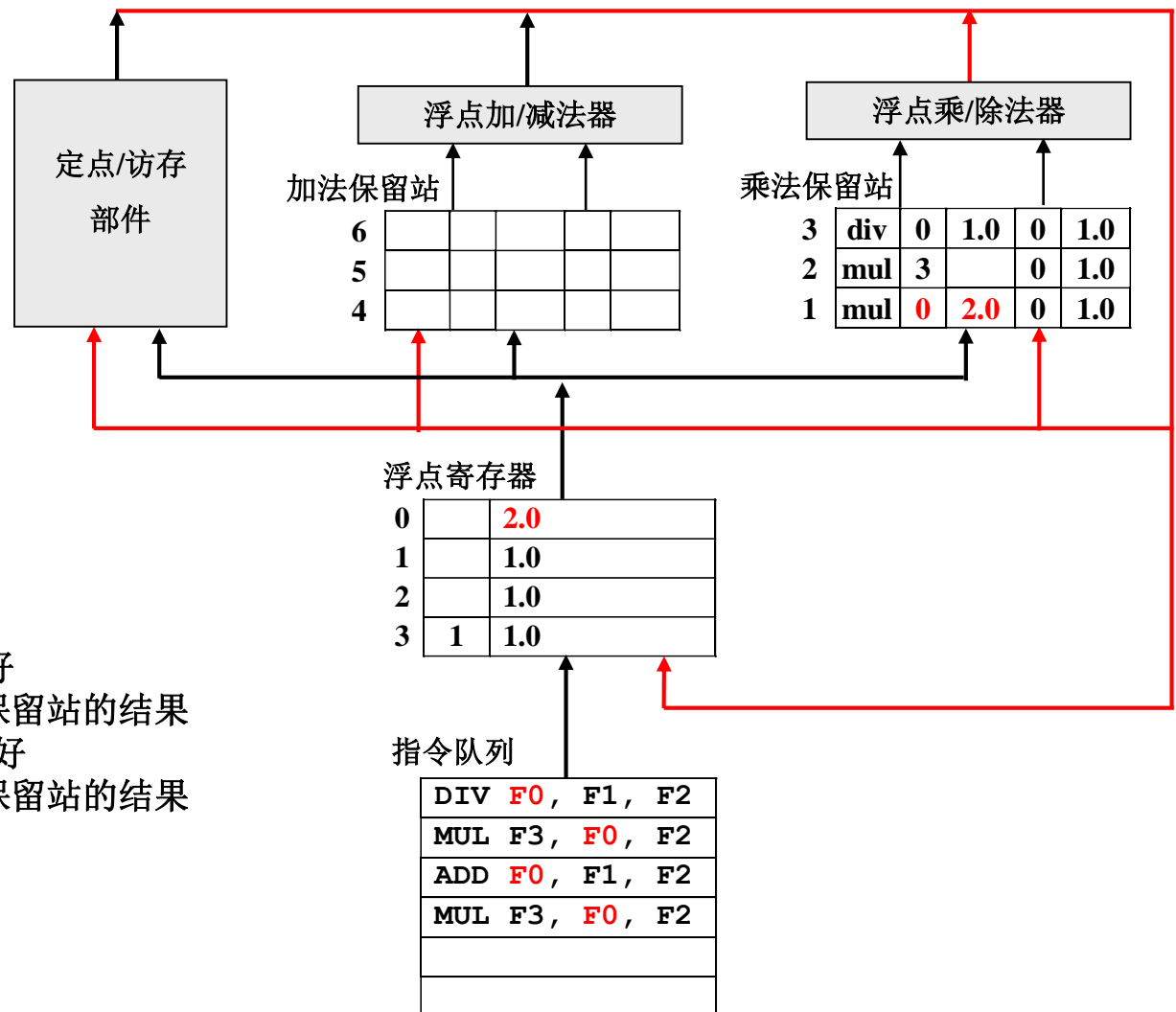


- DIV发射，F1,F2都准备好
- MUL1发射，F0等待3号保留站的结果
- ADD发射，F1,F2都准备好

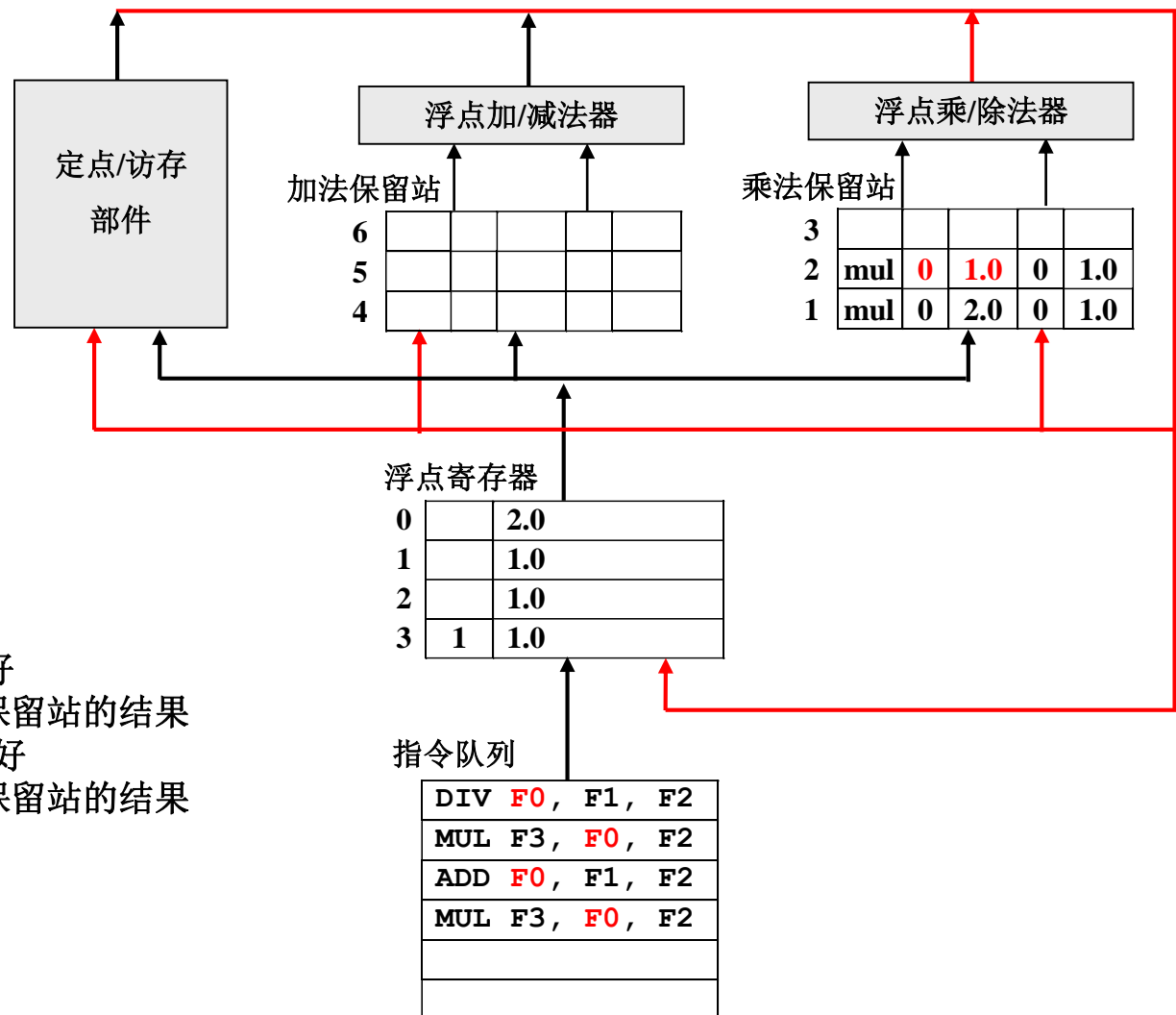




- DIV发射，F1,F2都准备好
- MUL1发射，F0等待3号保留站的结果
- ADD发射，F1,F2都准备好
- MUL2发射，F0等待6号保留站的结果



- 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

## 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

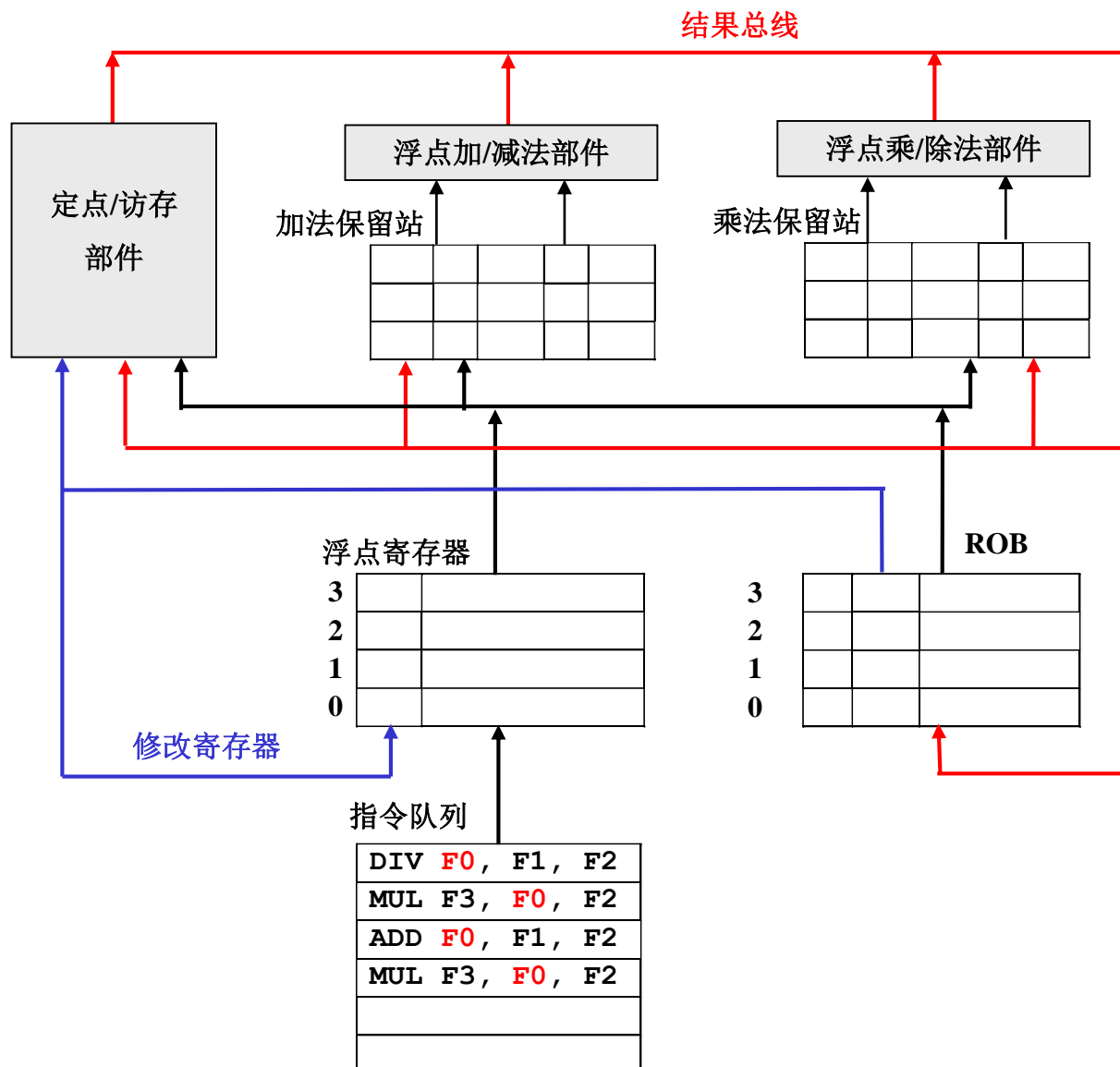
- 实现精确例外处理的一个办法是把后面指令对机器状态的修改延迟到前面指令都已经执行完
  - 有些指令在EX阶段也修改机器状态，如运算指令修改结果状态
  - 在执行阶段停止流水线会影响后面的指令执行
- 可以用一些缓冲器来临时保存执行结果，当前面所有指令执行完后，再把保存在缓冲器中的结果写回到寄存器或存储器
  - 在流水线修改机器状态时（在执行或写回阶段）写到缓冲器
  - 增加提交（Commit）阶段，把缓冲器的内容写回到寄存器或存储器
  - 提交阶段只有前面指令都结束后才能进行
  - 有序提交：乱序执行，有序结束
  - 所用的缓冲器通常被称为Reorder Buffer (ROB)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



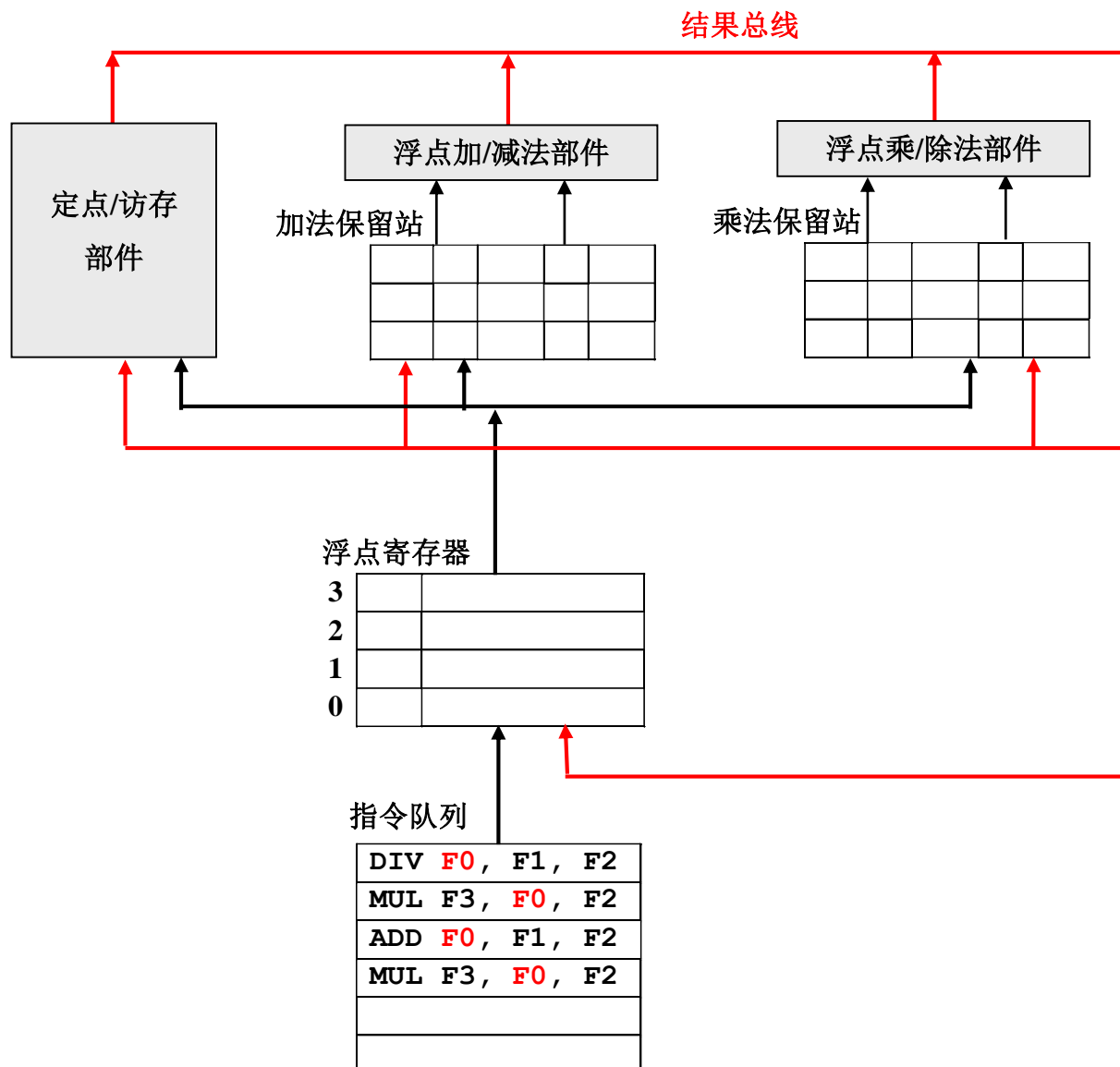
# 指令重排序缓存Reorder Buffer (ROB)

- 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

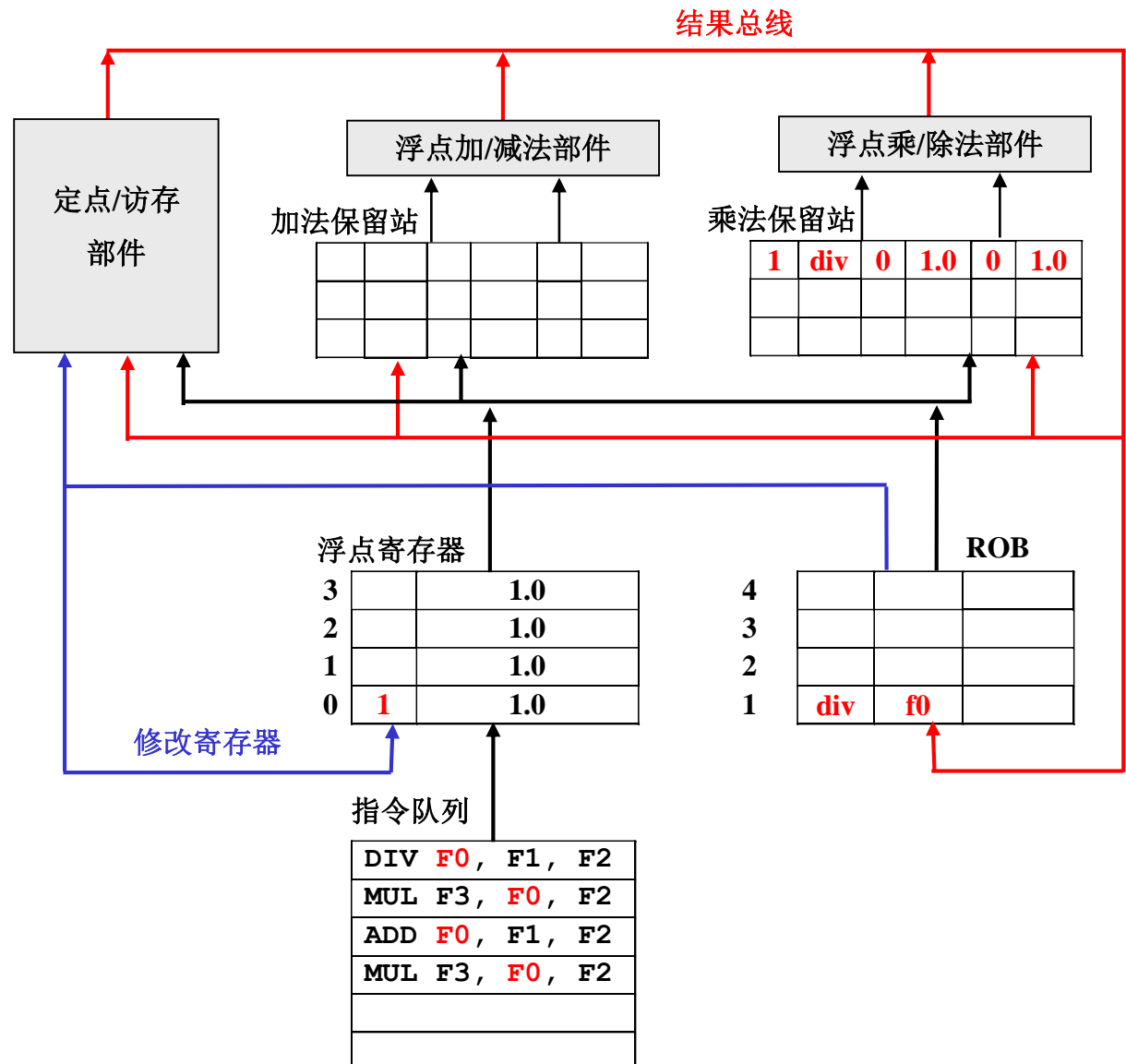
DIV F0, F1, F2

MUL F3, F0, F2

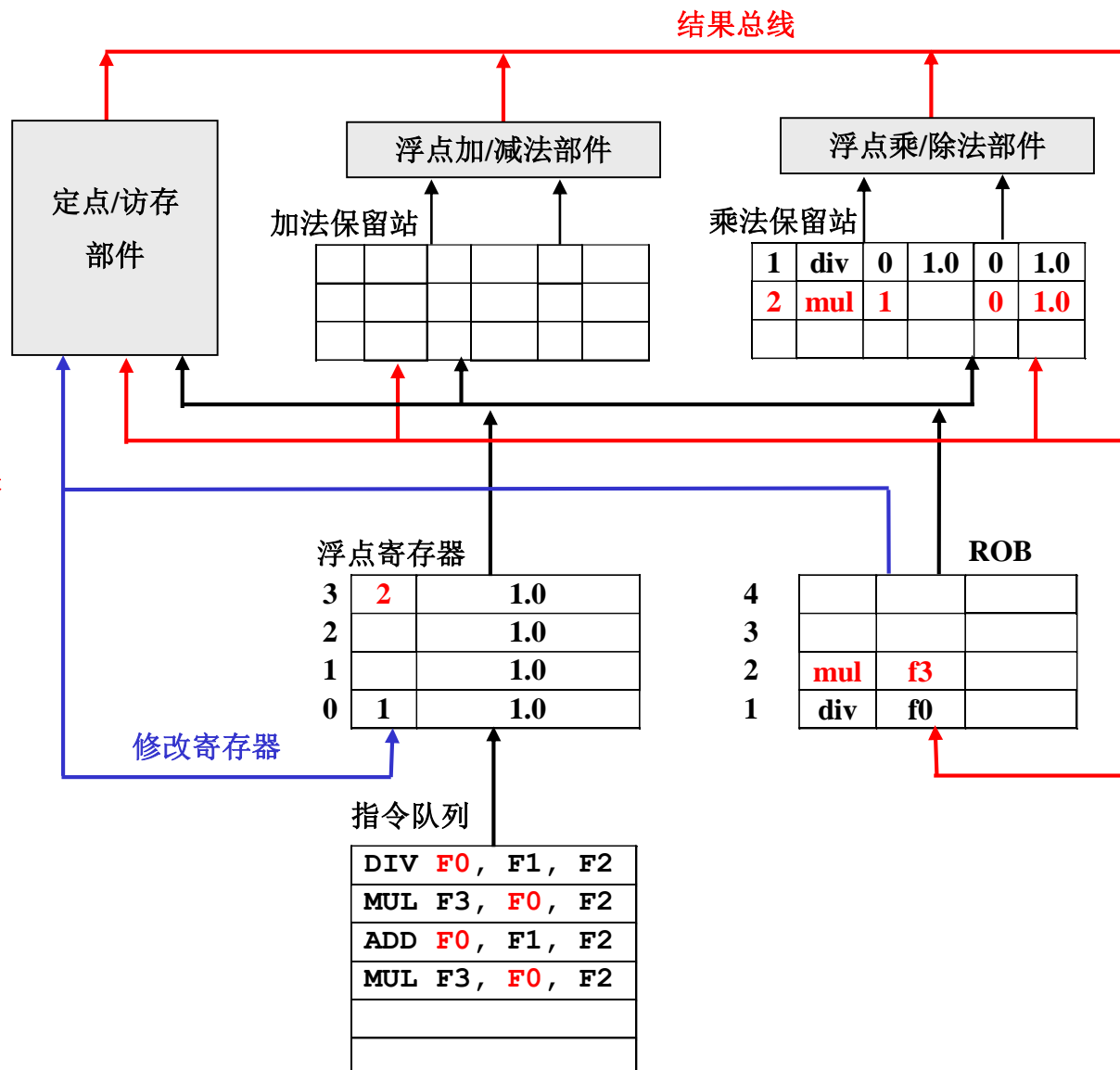
ADD F0, F1, F2

MUL F3, F0, F2

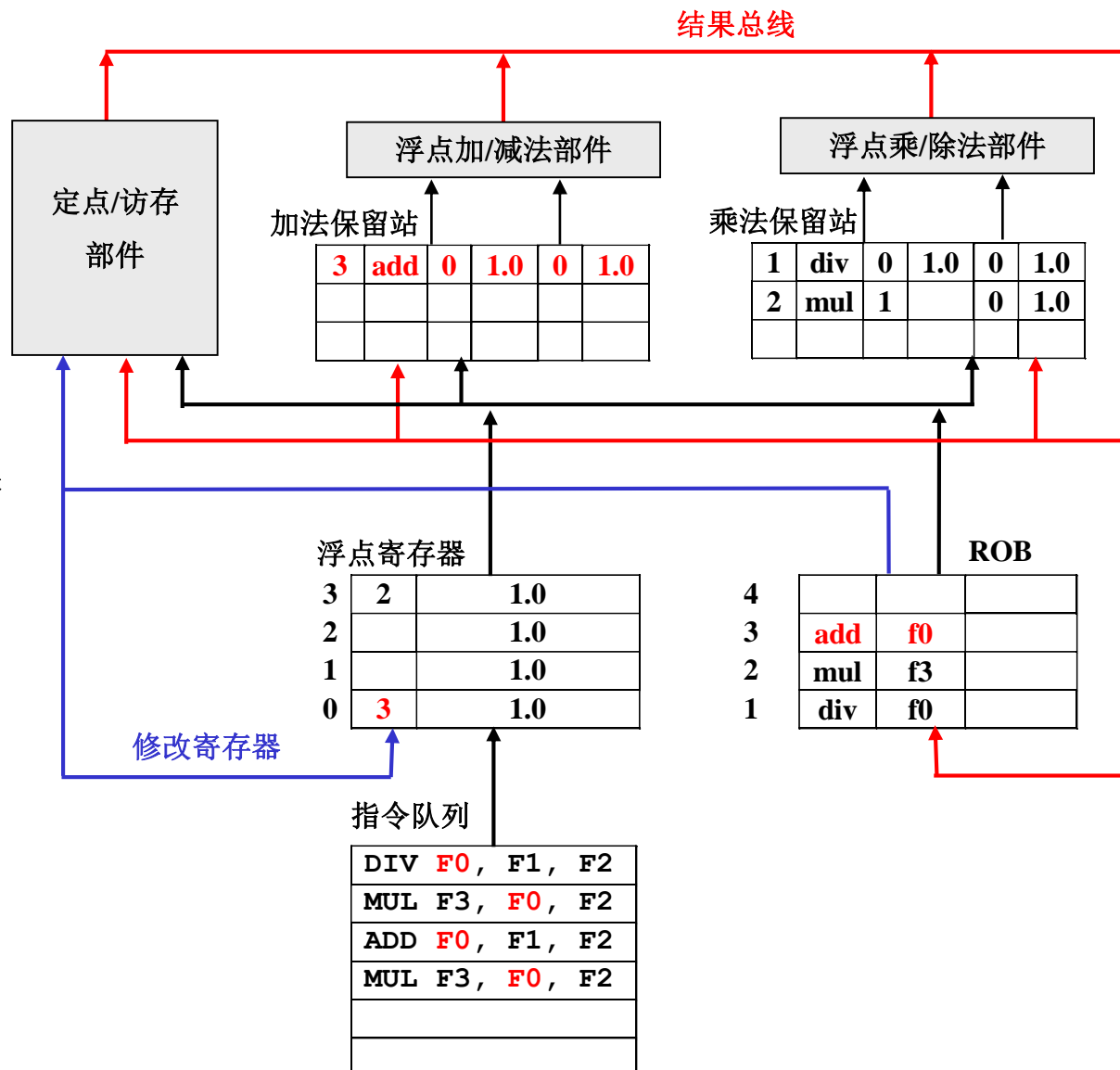
•DIV发射，F1,F2都准备好



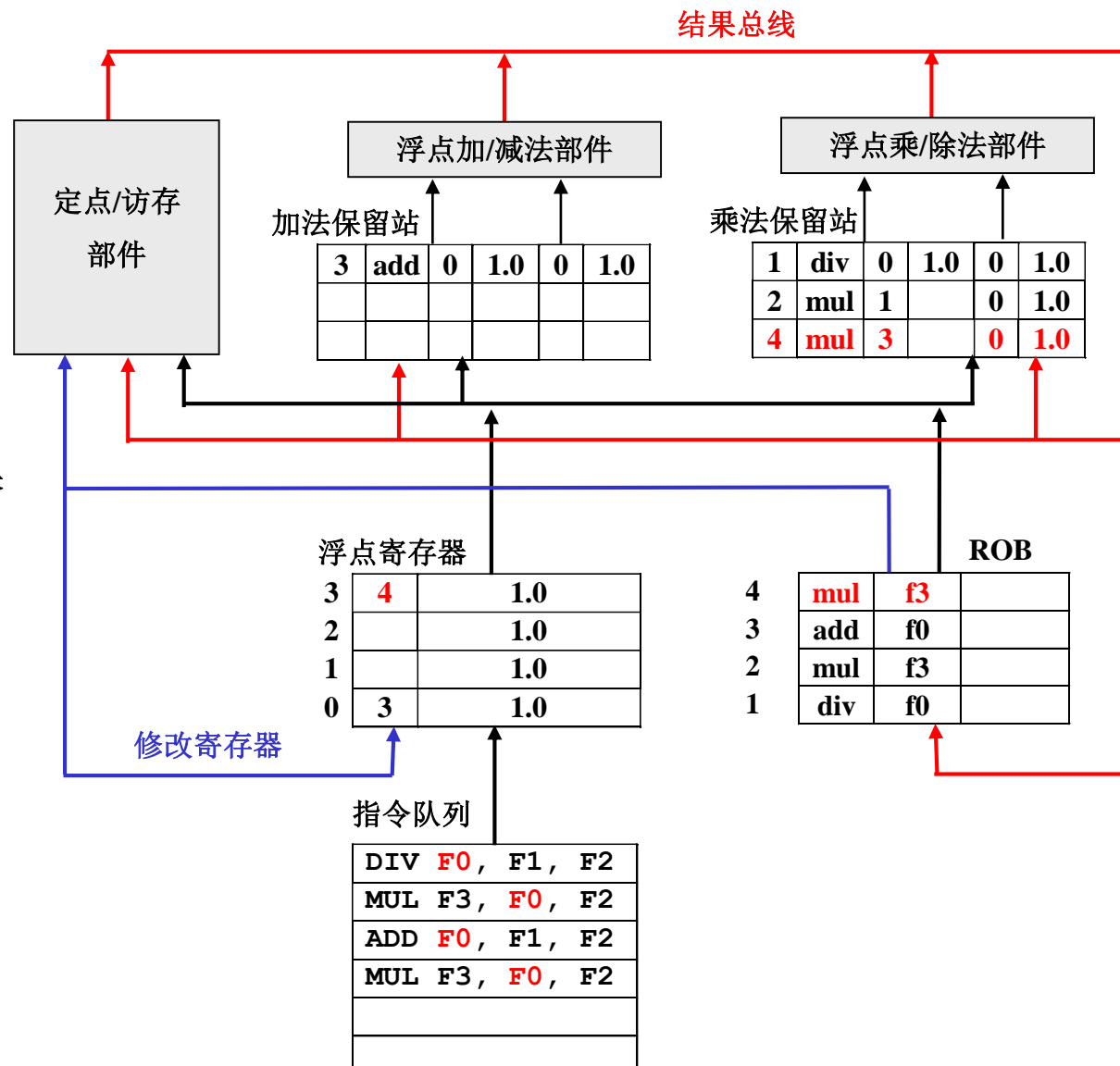
- DIV发射，F1,F2都准备好
- MUL1发射，F0等待1号RB的结果



- 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

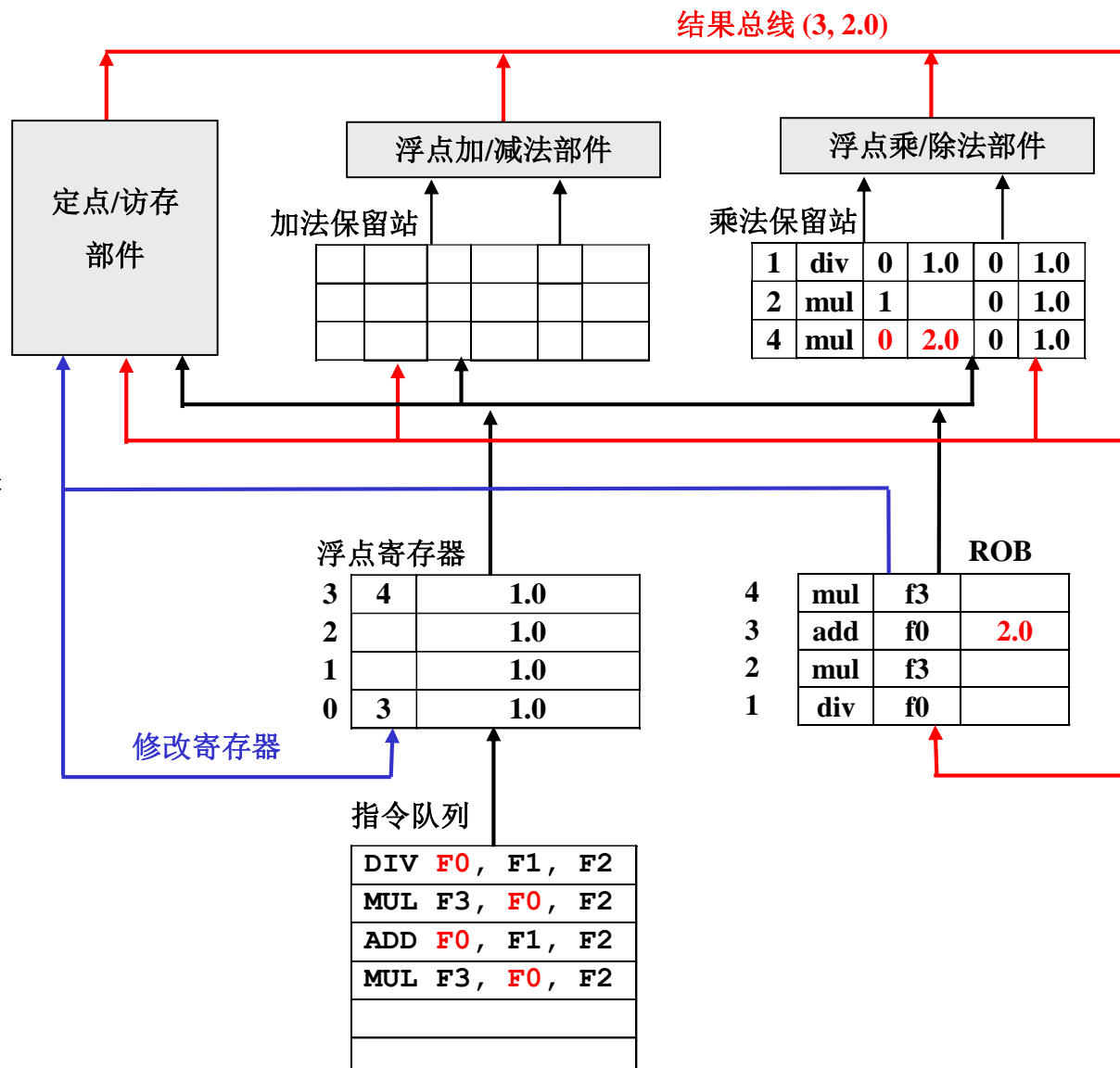


- DIV发射，F1,F2都准备好
- MUL1发射，F0等待1号RB的结果
- ADD发射，F1,F2都准备好
- MUL2发射，F0等待3号RB的结果

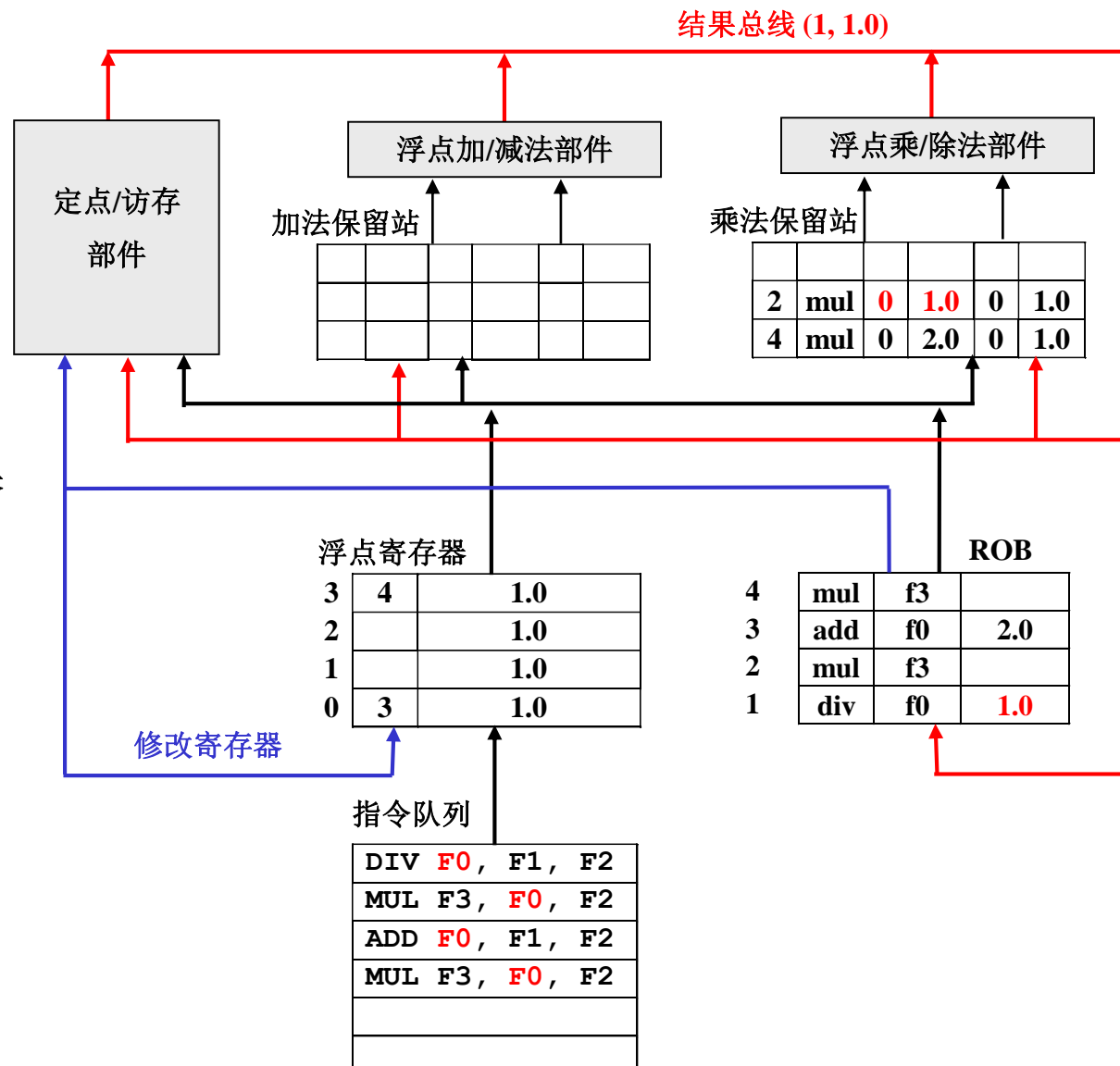




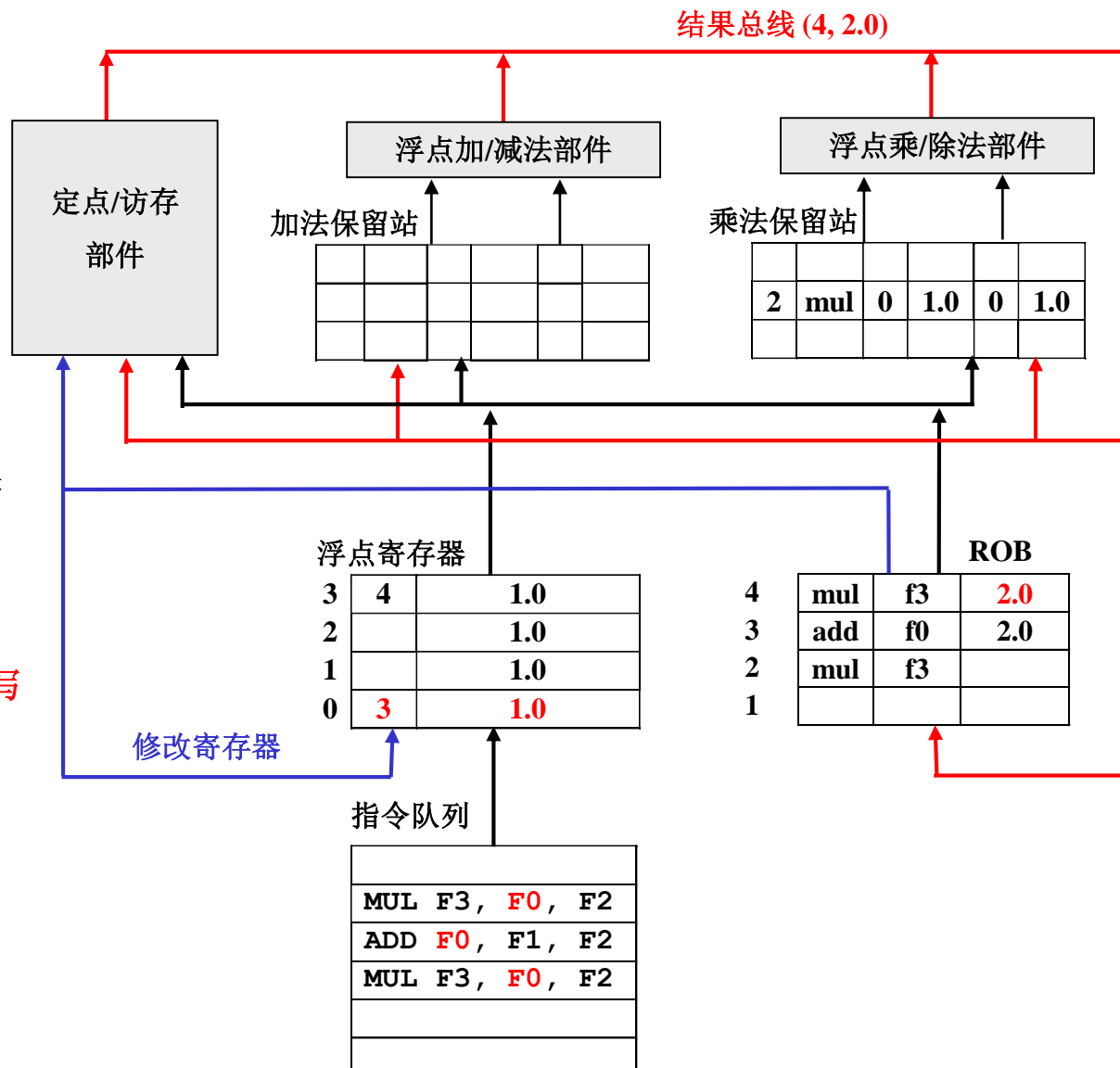
- DIV发射, F1,F2都准备好
- MUL1发射, F0等待1号RB的结果
- ADD发射, F1,F2都准备好
- MUL2发射, F0等待3号RB的结果
- ADD写回



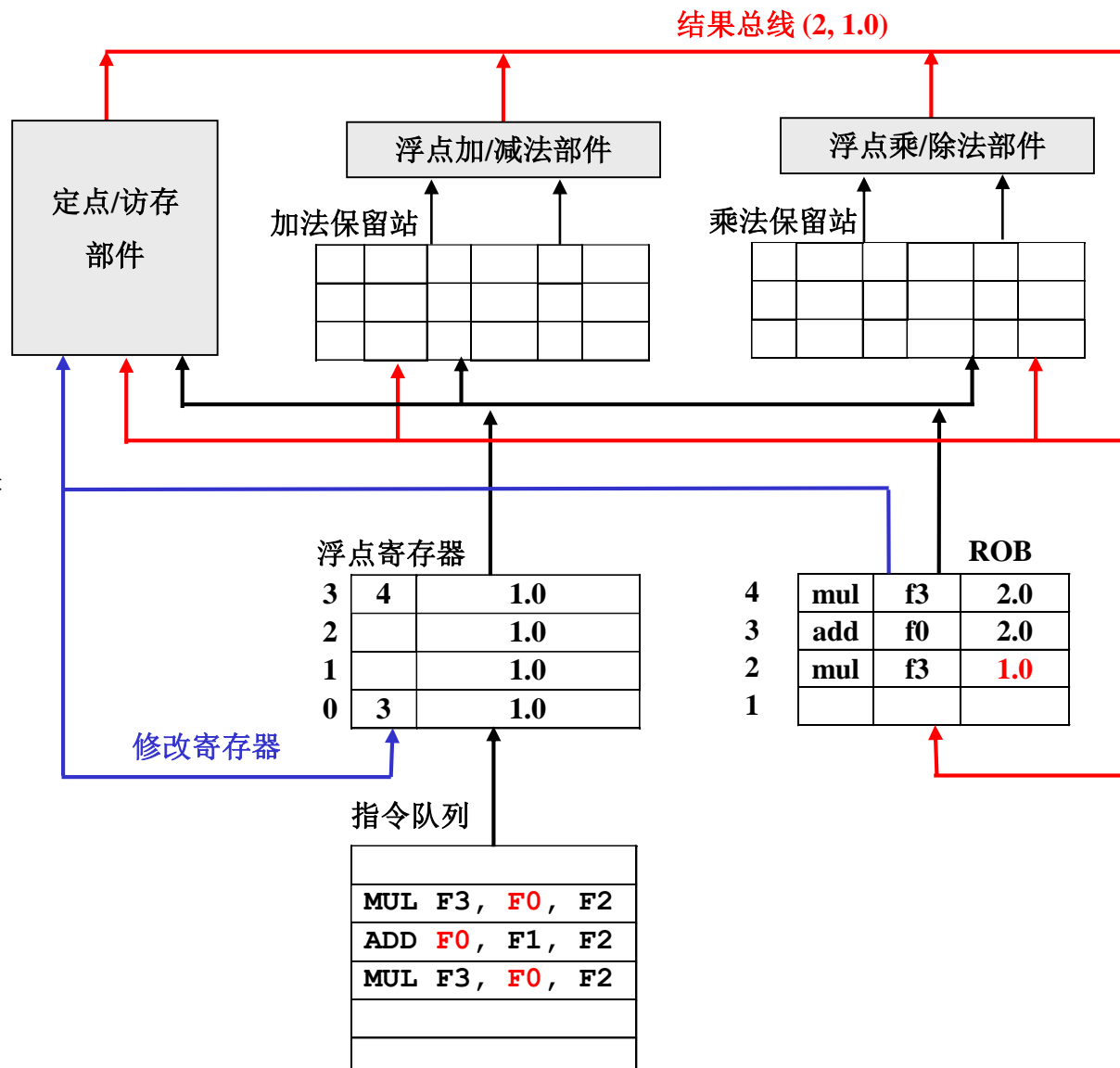
- DIV发射, F1,F2都准备好
- MUL1发射, F0等待1号RB的结果
- ADD发射, F1,F2都准备好
- MUL2发射, F0等待3号RB的结果
- ADD写回
- DIV写回



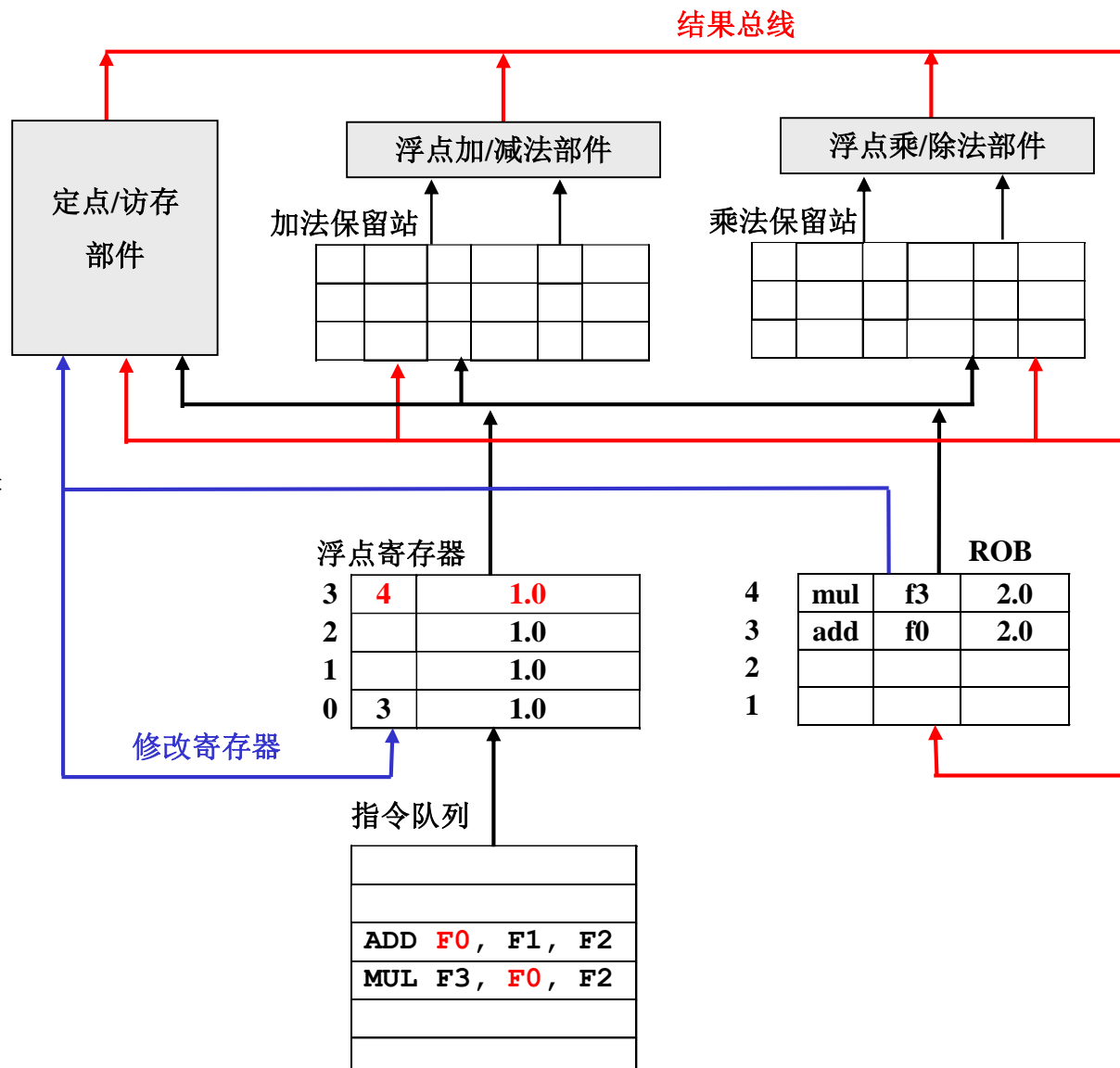
- DIV发射, F1,F2都准备好
- MUL1发射, F0等待1号RB的结果
- ADD发射, F1,F2都准备好
- MUL2发射, F0等待3号RB的结果
- ADD写回
- DIV写回
- DIV 提交 (写回寄存器), MUL2写回



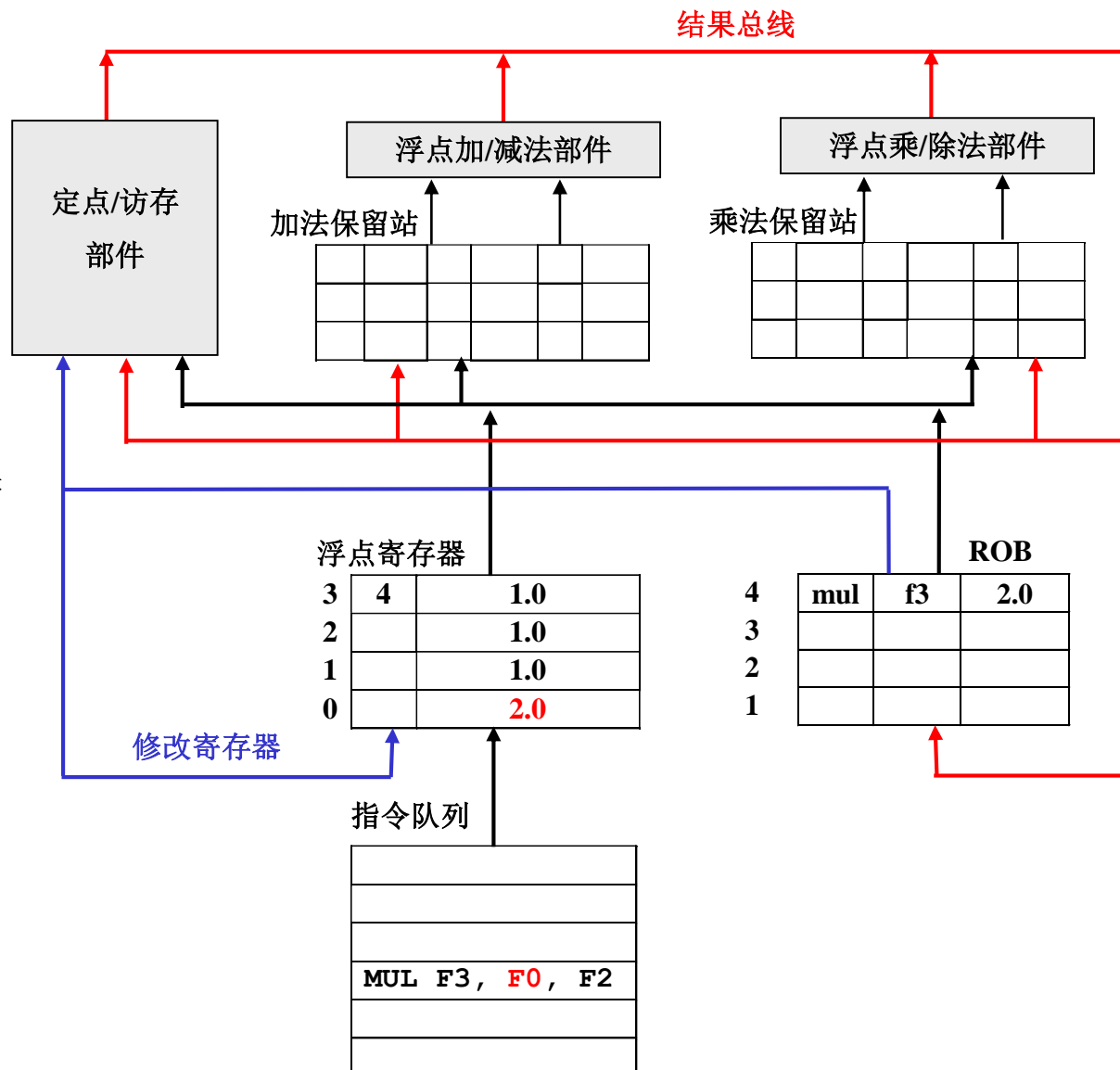
- DIV发射, F1,F2都准备好
- MUL1发射, F0等待1号RB的结果
- ADD发射, F1,F2都准备好
- MUL2发射, F0等待3号RB的结果
- ADD写回
- DIV写回
- DIV Commit, MUL2写回
- MUL1写回



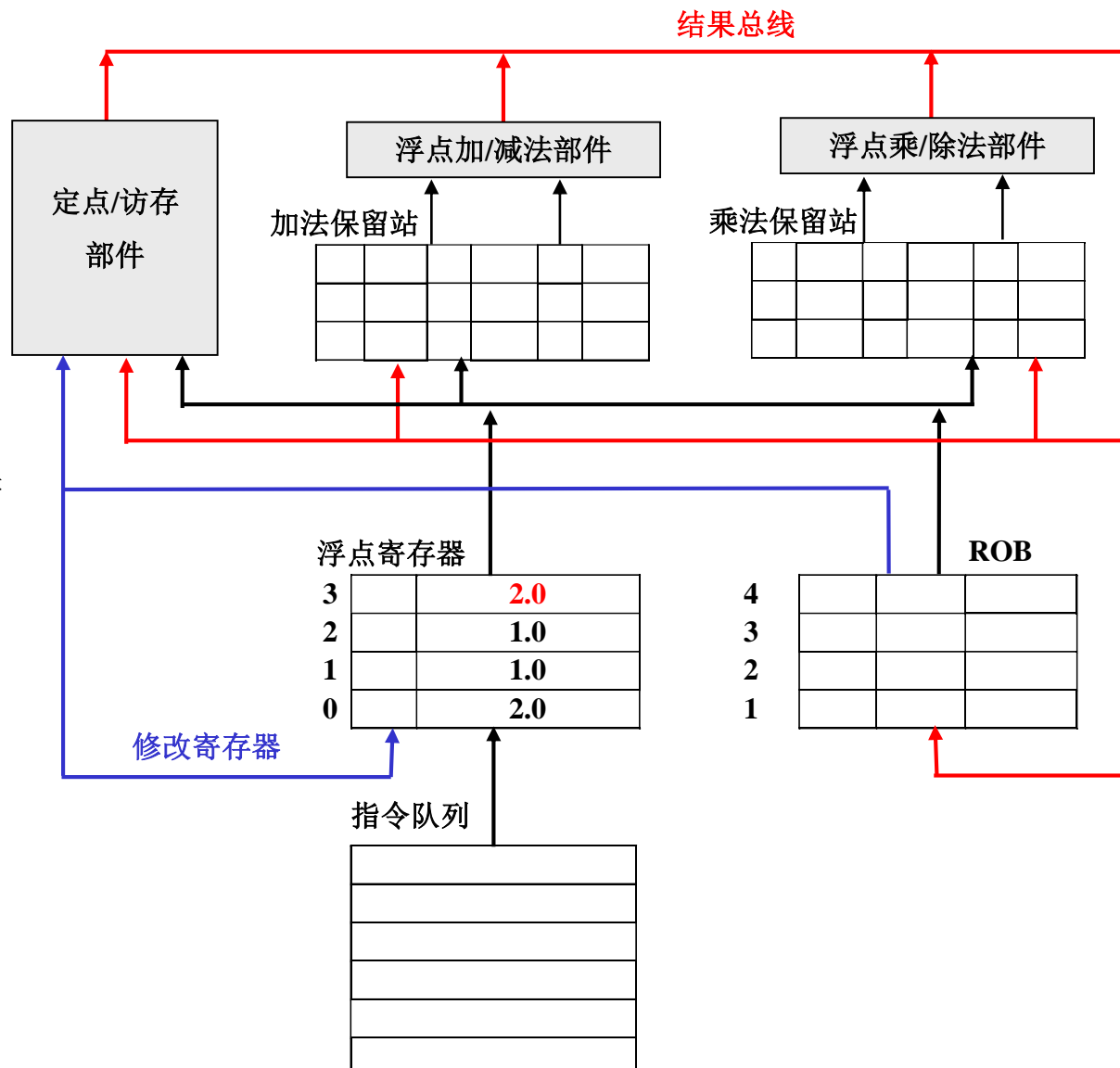
- DIV发射, F1,F2都准备好
- MUL1发射, F0等待1号RB的结果
- ADD发射, F1,F2都准备好
- MUL2发射, F0等待3号RB的结果
- ADD写回
- DIV写回
- DIV Commit, MUL2写回
- MUL1写回
- MUL1 Commit



- 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



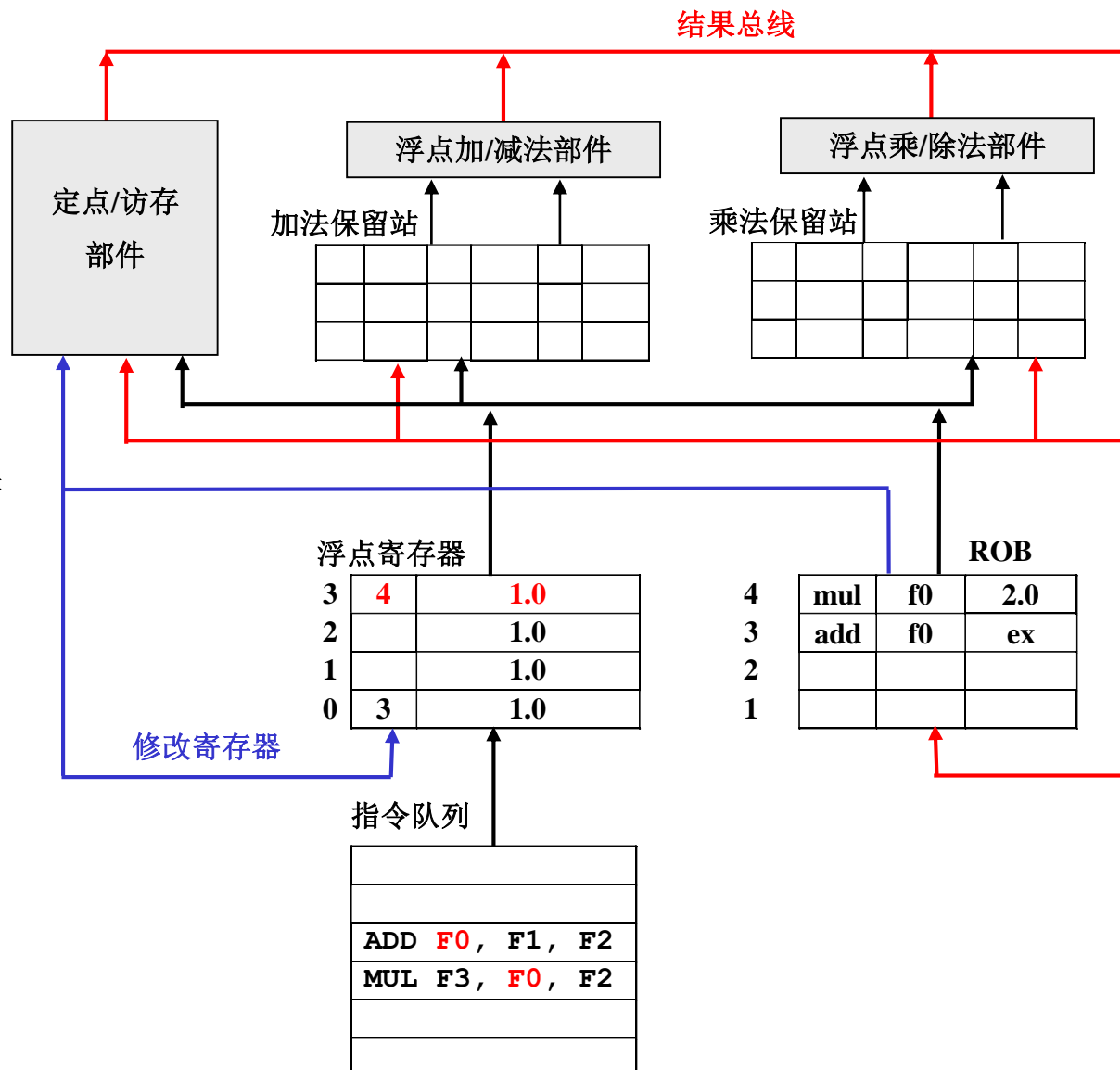
- DIV发射, F1,F2都准备好
- MUL1发射, F0等待1号RB的结果
- ADD发射, F1,F2都准备好
- MUL2发射, F0等待3号RB的结果
- ADD写回
- DIV写回
- DIV Commit, MUL2写回
- MUL1写回
- MUL1 Commit
- ADD Commit
- MUL2 Commit



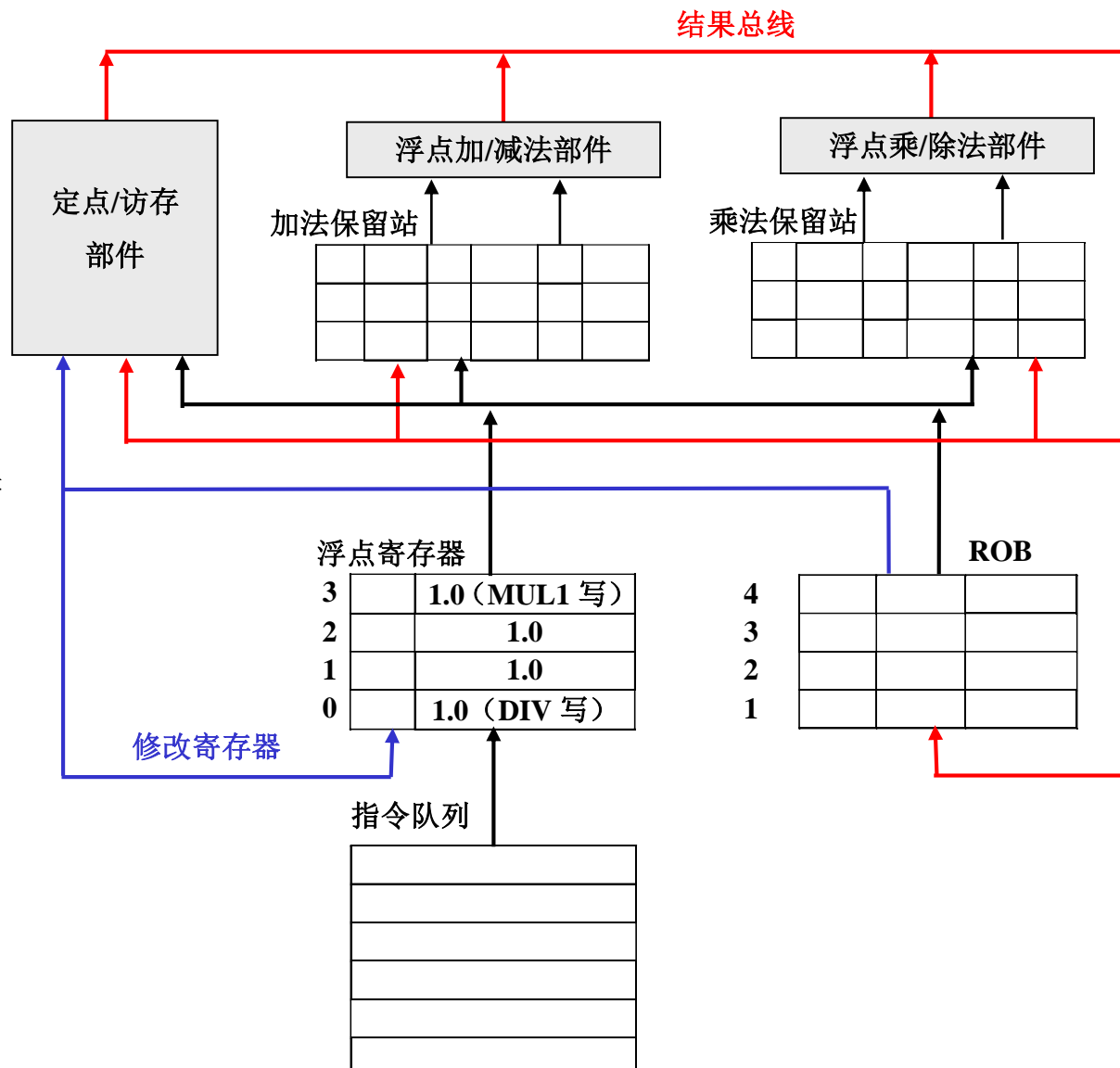
假设ADD发生了溢出例外



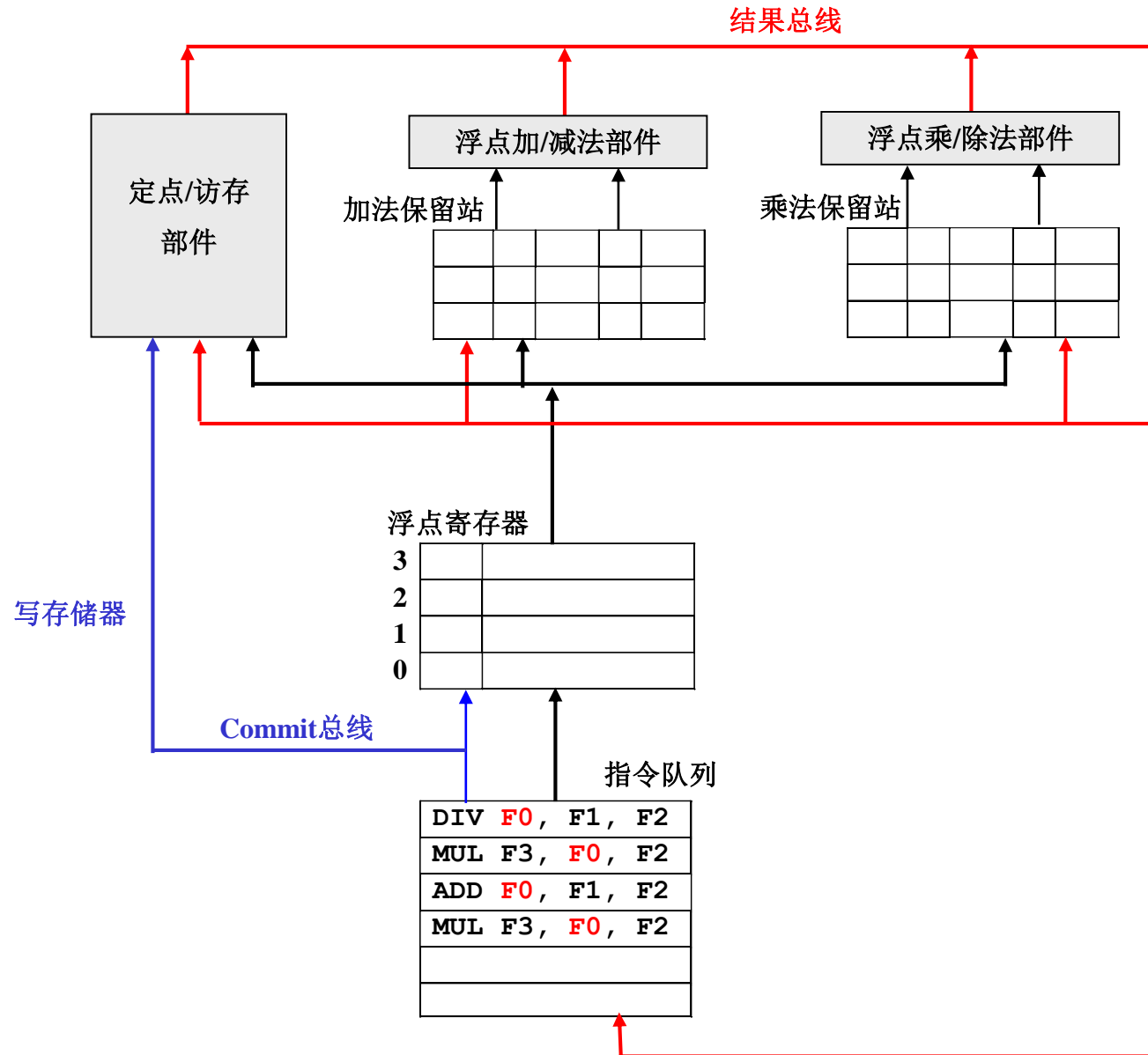
- DIV发射, F1, F2都准备好
- MUL1发射, F0等待1号RB的结果
- ADD发射, F1, F2都准备好
- MUL2发射, F0等待3号RB的结果
- ADD写回
- DIV写回
- DIV Commit, MUL2写回
- MUL1写回
- MUL1 Commit**



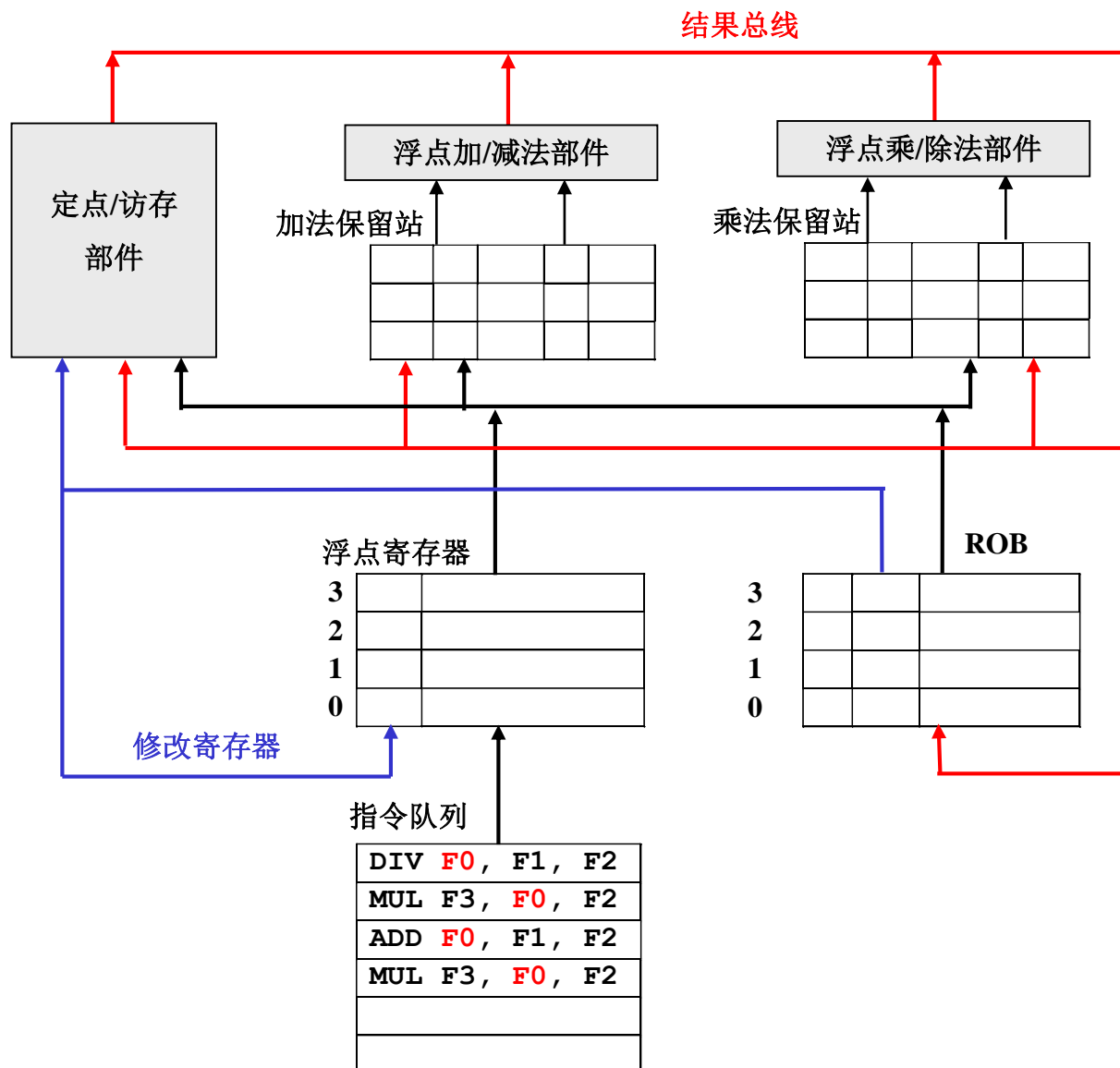
- DIV发射, F1, F2都准备好
- MUL1发射, F0等待1号RB的结果
- ADD发射, F1, F2都准备好
- MUL2发射, F0等待3号RB的结果
- ADD写回
- DIV写回
- DIV Commit, MUL2写回
- MUL1写回
- MUL1 Commit
- ADD Commit



- 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

# Meltdown攻击的基本原理

- 攻击者在用户态执行一条访问系统空间的load指令，发生例外，相关例外信息存在ROB中，访问数据X（假设为一个字节）目标寄存器为R1，重命名到PR1，**返回值存到PR1**，提交时会被取消（不写到R1）
- 攻击者读取R1并从PR1中读到内容X，作为下标访问一个事先准备好的用户态数组A，A中被访问的数据带到Cache中，其它数据仍在内存
- 非法load指令成为ROB头指令被提交，load指令及其后续指令被取消，但A[x]内容已经在Cache
- 攻击者访问数组A的所有数据并判断其延迟，其中有一个数据在Cache中，延迟最短，假设数组A第10号元素访问延迟最短，则X=10
  - 侧信道攻击
- Meltdown攻击确实由乱序执行引起，但很容易规避
  - Load操作执行阶段发生例外时，在ROB中置例外，但不要把返回值写入寄存器

# 动态流水线小结

- 有序进入、乱序执行（允许超车）、有序结束
- 主要数据结构
  - 保留站（发射队列）把有序变成乱序，临时存指令
  - 重命名寄存器用于保存未提交的临时结果，临时存数据
  - ROB把乱序重新变成有序
- 乱序的能力和有关队列大小紧密相关
  - 现代高性能CPU一般都有100条以上指令在流水线中乱序执行

# 作 业