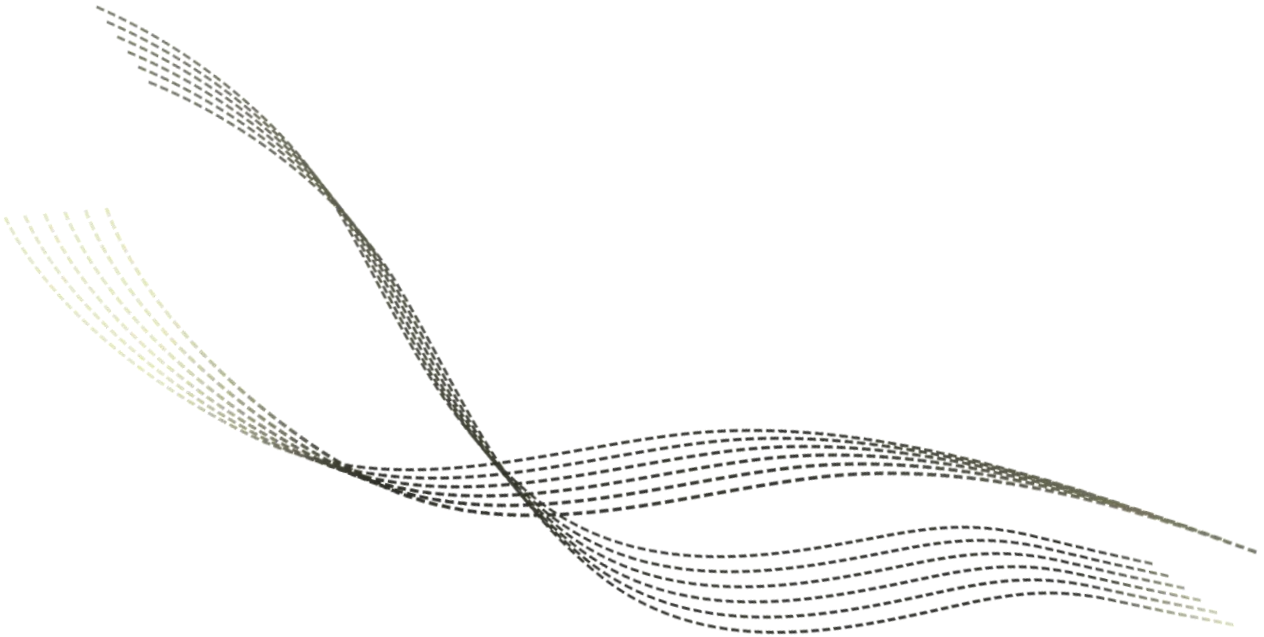
高级计算机体系结构

Advanced Computer Architecture



指令级并行II

沈明华

目录

CONTENTS

PART 01 超标量流水线

■ 背景

◘ 标量流水线(scalar instruction pipeline)的局限性

– 标量指令流水线是一种采用线性顺序组织的多级单流水线结构

标量流水线吞吐量的理论上限

- 每个机器周期仅能完成一条指令
- 过深的流水线分级会降低收益

单一流水线整合存在效率问题

- 需要不同的硬件资源支持
- 指令执行需要长周期/可变延迟

严格执行机制导致的效率问题

- 单个指令阻塞会影响后续所有指令

■ 超标量流水线

□ 提升简单标量流水线的性能

- 通过扩展架构缓解前述三大局限性
- 扩展1：实现并行处理(超标量流水线)
  - 每个机器周期提取多条指令
  - 并行解码指令
- 扩展2：实现多样化(如tomasulo配置多个计算部件)
  - 在执行阶段配置多组异构功能单元，比如为较慢的乘法除法配置单独计算单元
  - 配备数量合适的功能单元，减少功能单元不足导致的指令阻塞
- 扩展3：实现动态执行(乱序发射+ROB)
  - 后续指令可越过阻塞的前导指令继续执行
  - 采用复杂多条目缓冲器对传输中指令进行缓存

■ 超标量流水线

□ 多发射处理器(Multiple Issue Processor)

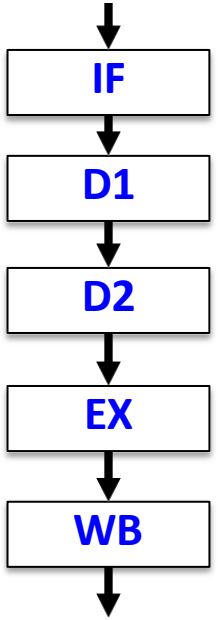
- 为实现CPI<1的目标，CPU必须每周期能发射多条指令
- k-发射处理器：每个流水级可同时处理k条指令
  - Alpha 21264：四发射架构
  - MIPS R10000：四发射架构
  - Intel P4：三发射架构
- 多发射处理器主要有两种实现方式
  - 超标量处理器
  - 超长指令字(**VLIW**, very long instruction word)处理器

■ 超标量流水线

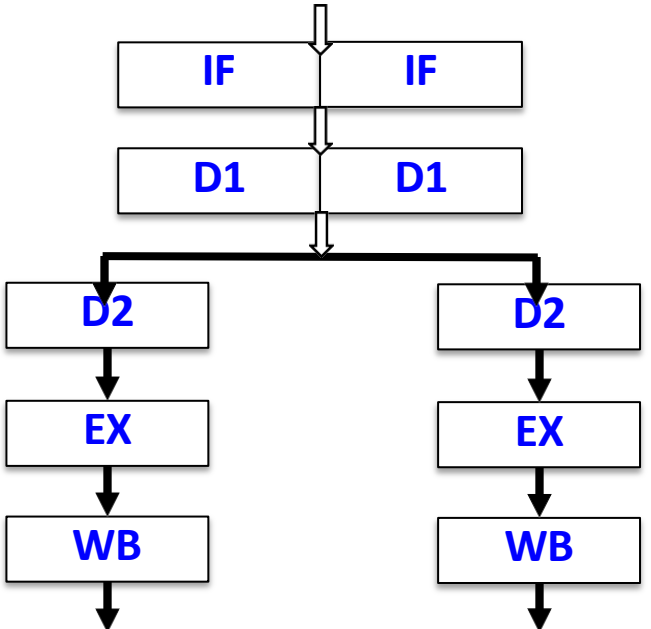
□ 从标量到超标量

- 加速比：
  - ➢ 标量流水线：相对于无流水线设计
    - 主要取决于标量流水线的深度
  - ➢ 超标量流水线：相对于标量流水线
    - 主要取决于并行流水线的宽度

i486的5阶段标量流水线



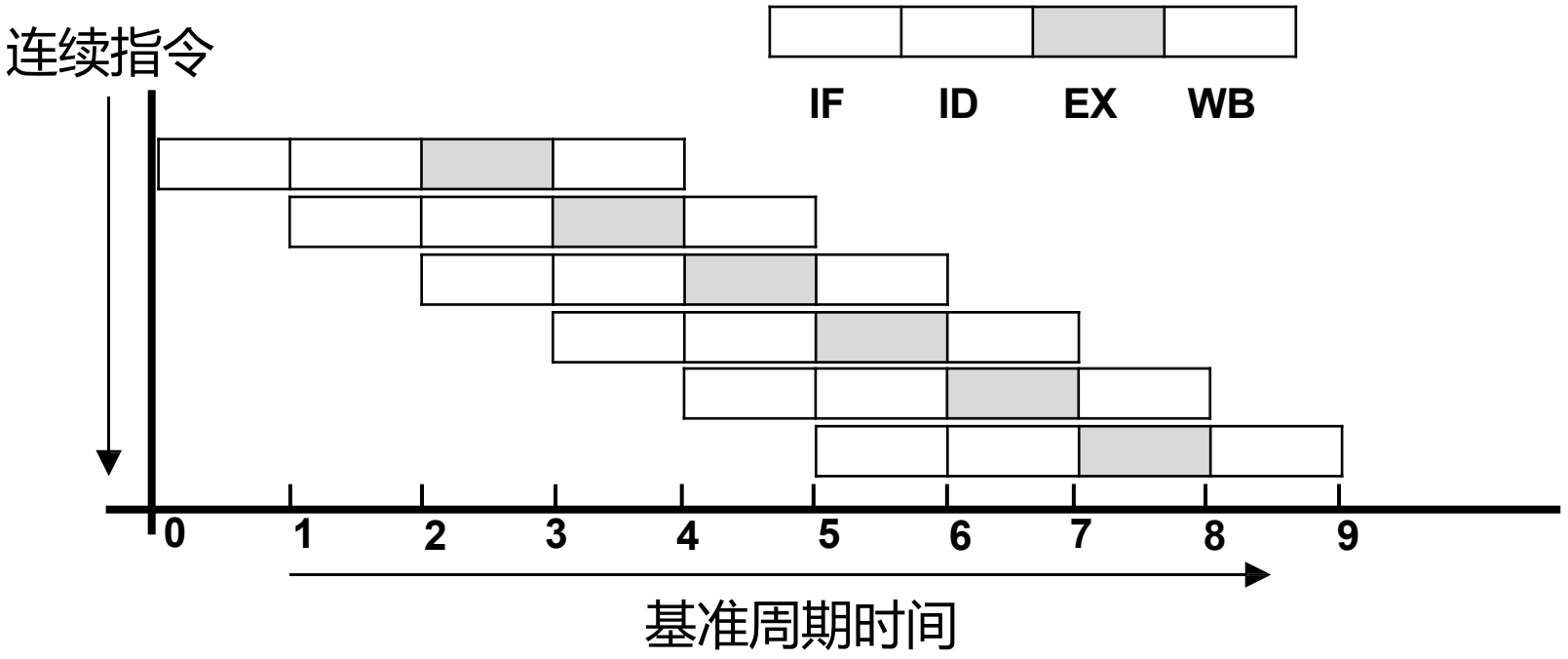
Pentium的5阶段超标量流水线，宽度=2



■ 超标量流水线

◘ 指令级并行机器的分类

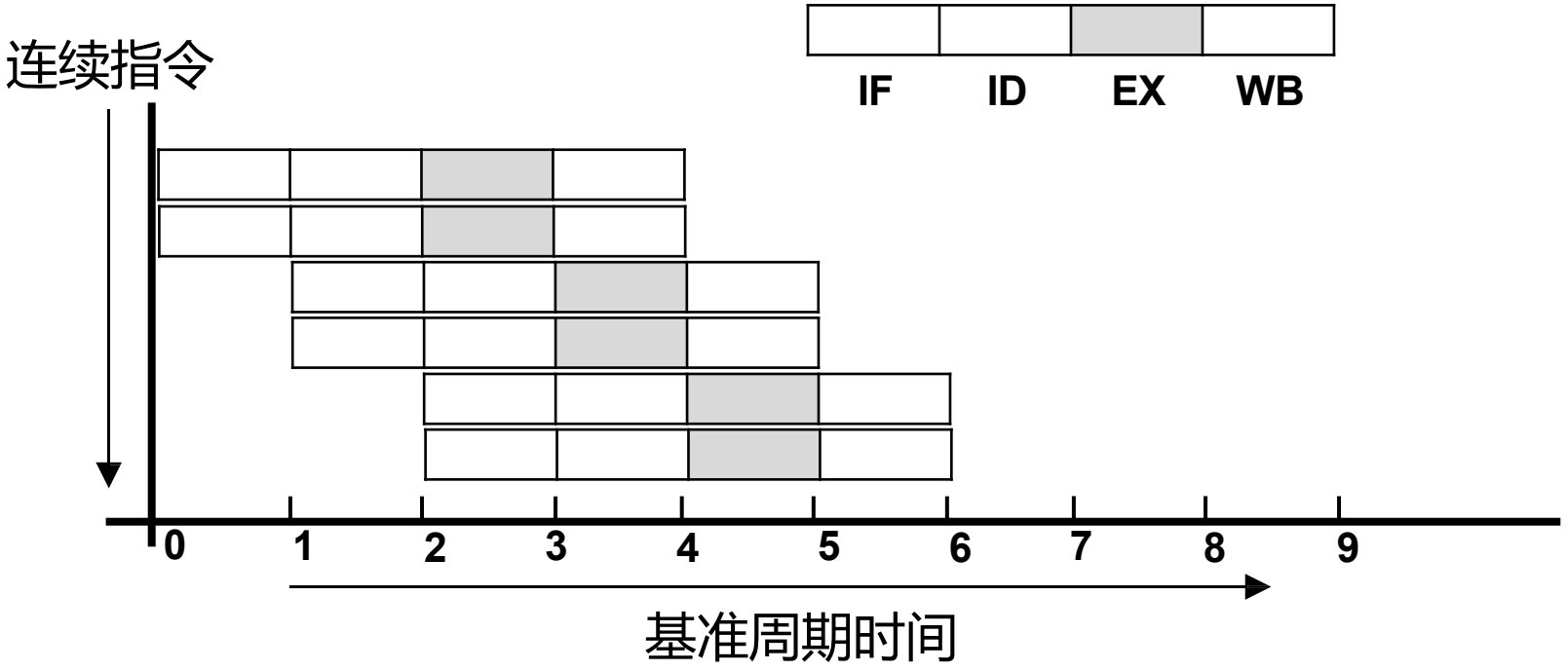
- 基准(简单标量)
  - 充分利用所需指令级并行度 = 1/周期
  - 每周期发射指令数(IPC)= 1



■ 超标量流水线

◘ 指令级并行机器的分类

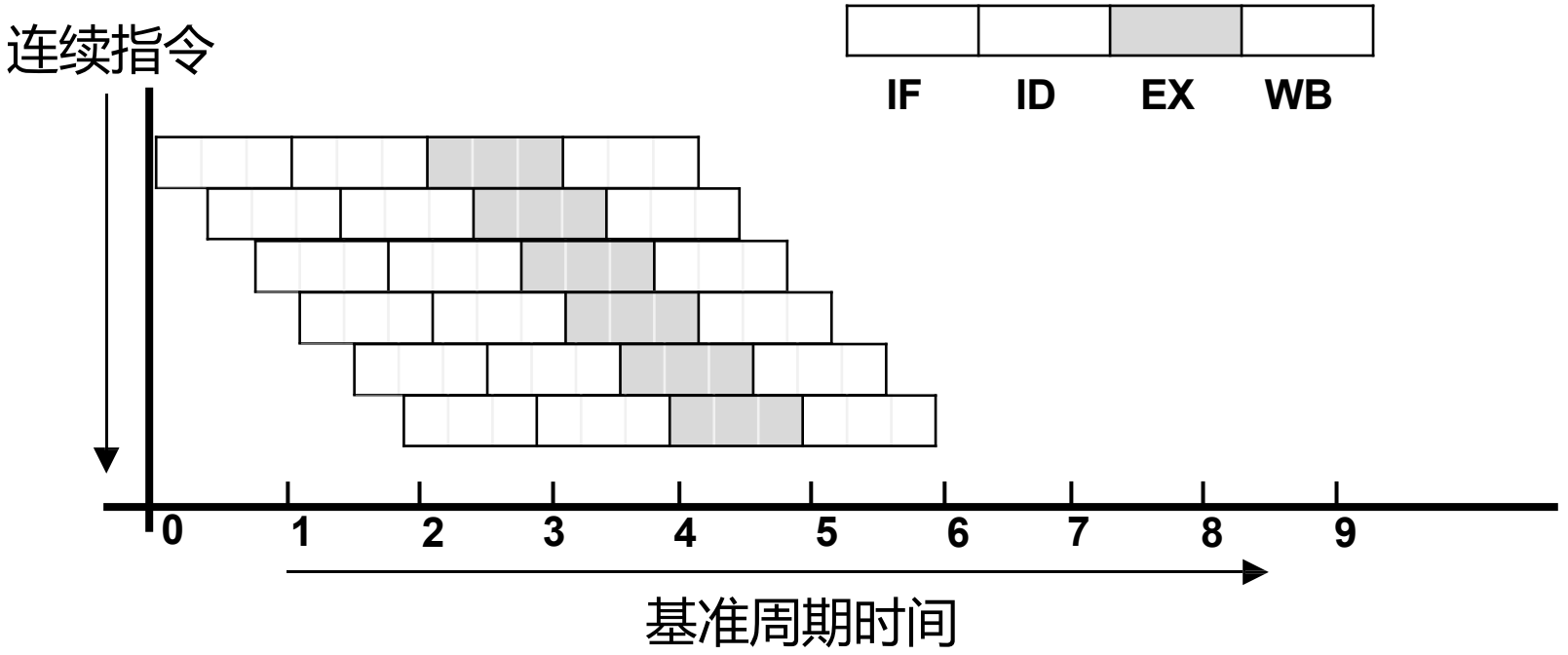
- 超标量(宽度为n)
  - ➢ 充分利用所需指令级并行度 = n/周期
  - ➢ 每周期发射指令数(IPC)= n



■ 超标量流水线

□ 指令级并行机器的分类

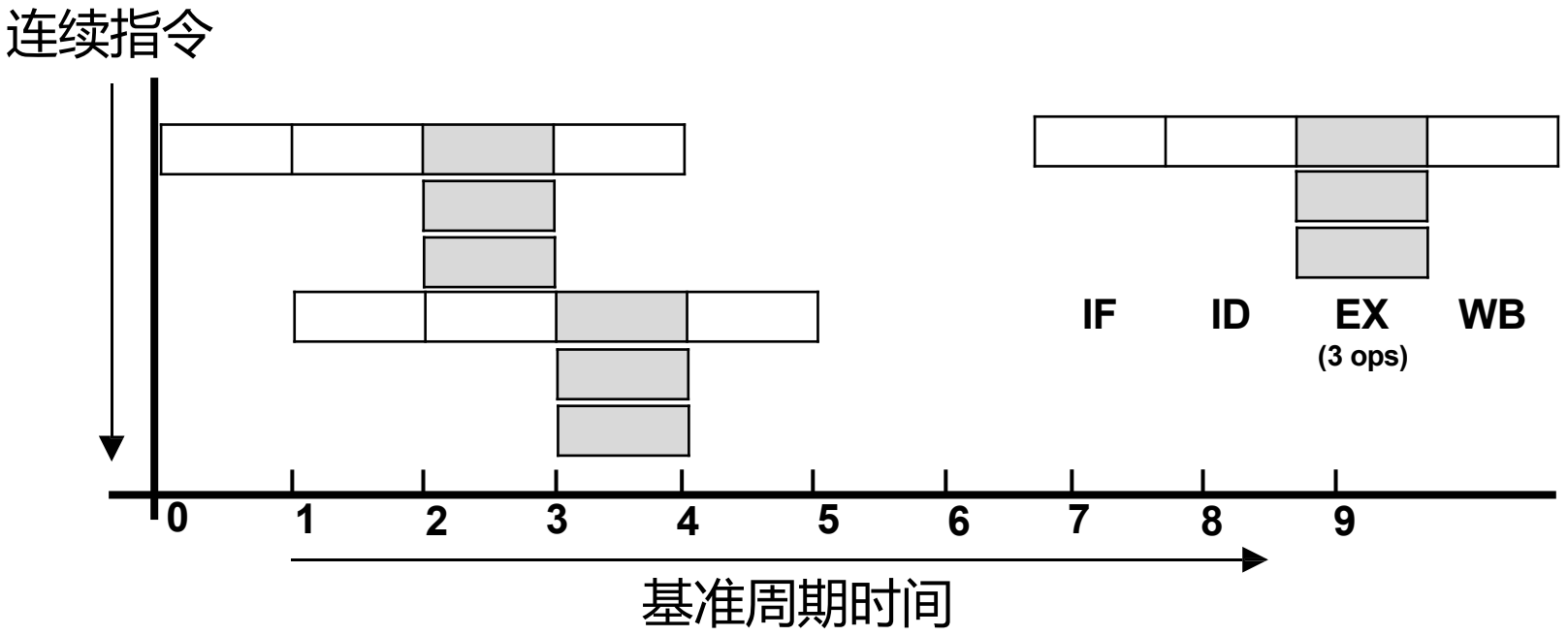
- 超流水线(深度为m)
  - 充分利用所需指令级并行度 = m/基准周期
  - 每周期发射指令数(IPC)= 1，但时钟周期为基准周期的1/m



■ 超标量流水线

□ 指令级并行机器的分类

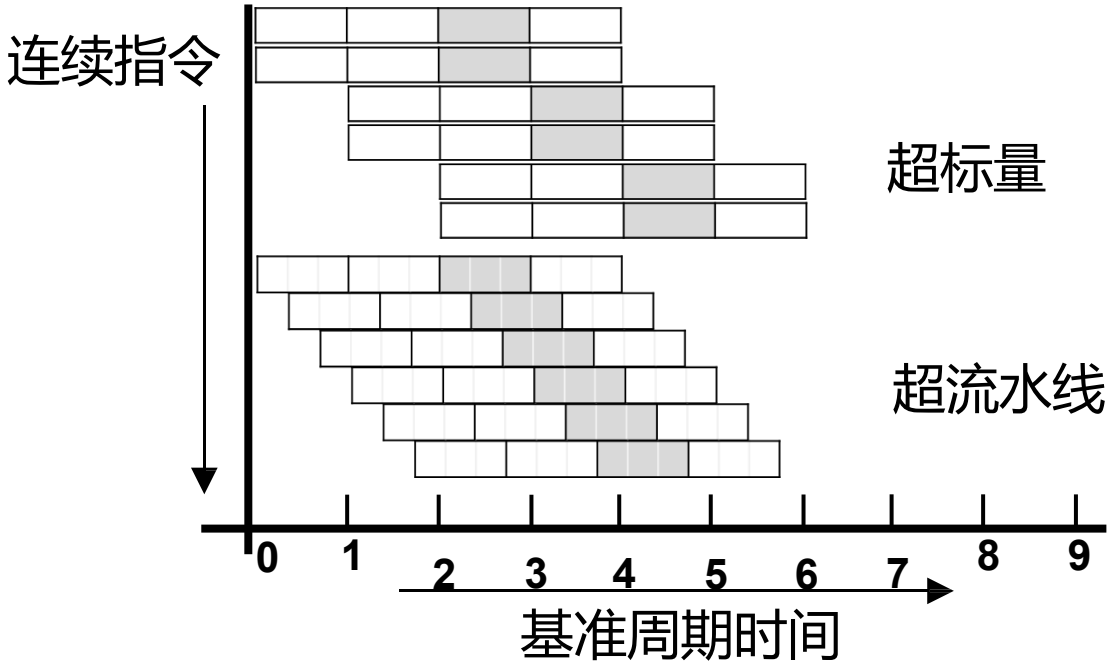
- VLIW(假设打包n个操作)
  - 充分利用所需指令级并行度 = n/周期
  - 每周期执行指令数(IPC)= n条指令/周期 = 1条超长指令字/周期



■ 超标量流水线

□ 超标量与超流水线的对比

- 大致等效的性能
- 如果超标量的宽度等于超流水线的深度，则两者吞吐量相近
- 并行性呈现的方式不同（空间并行 vs 时间并行）



PART 02 分支预测

■ 背景

□ 控制冒险

- 上一章提到流水线存在控制冒险
  - ➢ 无论是tomasulo还是scoreboard都没有很好解决控制冒险问题
  - ➢ 原因在于必须等到分支跳转指令完成计算，才知道下一条指令是什么
    - 在确定下一条指令前，不发送任何指令，引入流水线暂停
  - ➢ 发送哪一条指令成为关键问题
- 现代处理器引入分支预测(branch prediction)技术
  - ➢ 在分支跳转指令完成计算前，预测下一条发送的指令

■ 分支预测

□ 分支预测

- gcc中17%的指令是条件分支指令

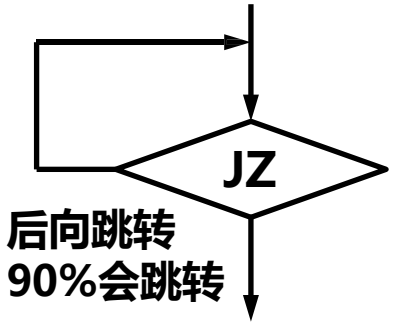
指令	跳转状态已知时机	目标地址已知时机
BEQZ/BNEZ	寄存器读取后	指令读取后
J	恒定跳转	指令读取后
JR	恒定跳转	寄存器读取后

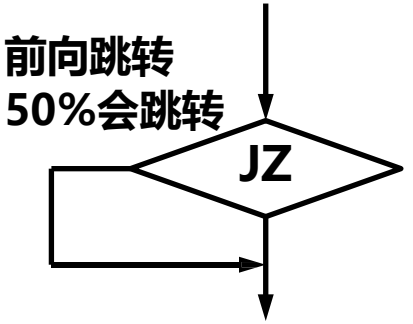
- 分支预测的两大核心组件
  - 分支目标预测与分支条件预测
- 分支惩罚降低CPU的性能表现
  - 性能表现 = f(预测准确率，预测错误代价)

■ 分支预测

□ 分支预测

- 静态预测：在编译时预测分支行为
  - 单个分支往往先天存在跳转或不跳转的偏向，据此有以下三种静态预测策略
    - 预测分支为必定跳转
    - 基于分支方向进行预测
    - 基于先前收集的配置文件信息进行预测



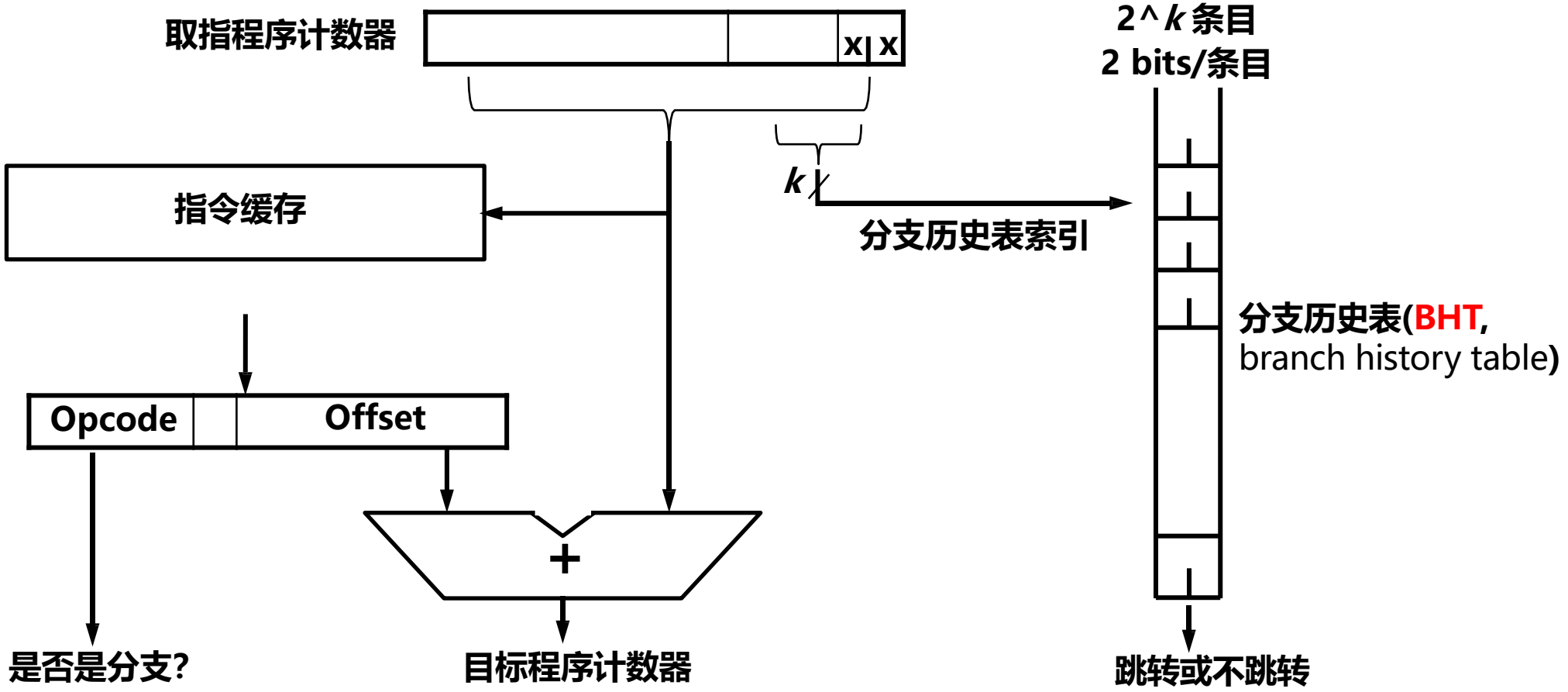


- 动态预测：基于每个分支的实际行为
  - 可在程序运行期间动态调整分支预测策略
  - 硬件支持：分支历史表(BHT)、分支目标缓冲器(BTB)等
- 总体而言一条分支指令发生跳转的概率大约60~70%
  - ISA也可以向转移指令附加偏向，例如Motorola MC88110的
  - bne0(prefered taken) beq0(not taken)

■ 分支预测

□ 动态预测：基于历史记录的跳转

– 4K条目的分支历史表，每条目2位：约80-90%预测准确率



■ 分支预测

□ 分支预测效果较好的原因

- 分支指令在时间和空间上有类似局部性的性质
- 时间局部性
  - 同一条分支指令上次未跳转，这次大概率也不跳转
  - 典型案例，如右图for循环所示，i<100会连续一百次跳转，最后一次才不跳转
  - 可以为这条指令单独记录最近n次执行的跳转方向

```
for (int i = 0; i < 100; i++)
    do some compute;
```

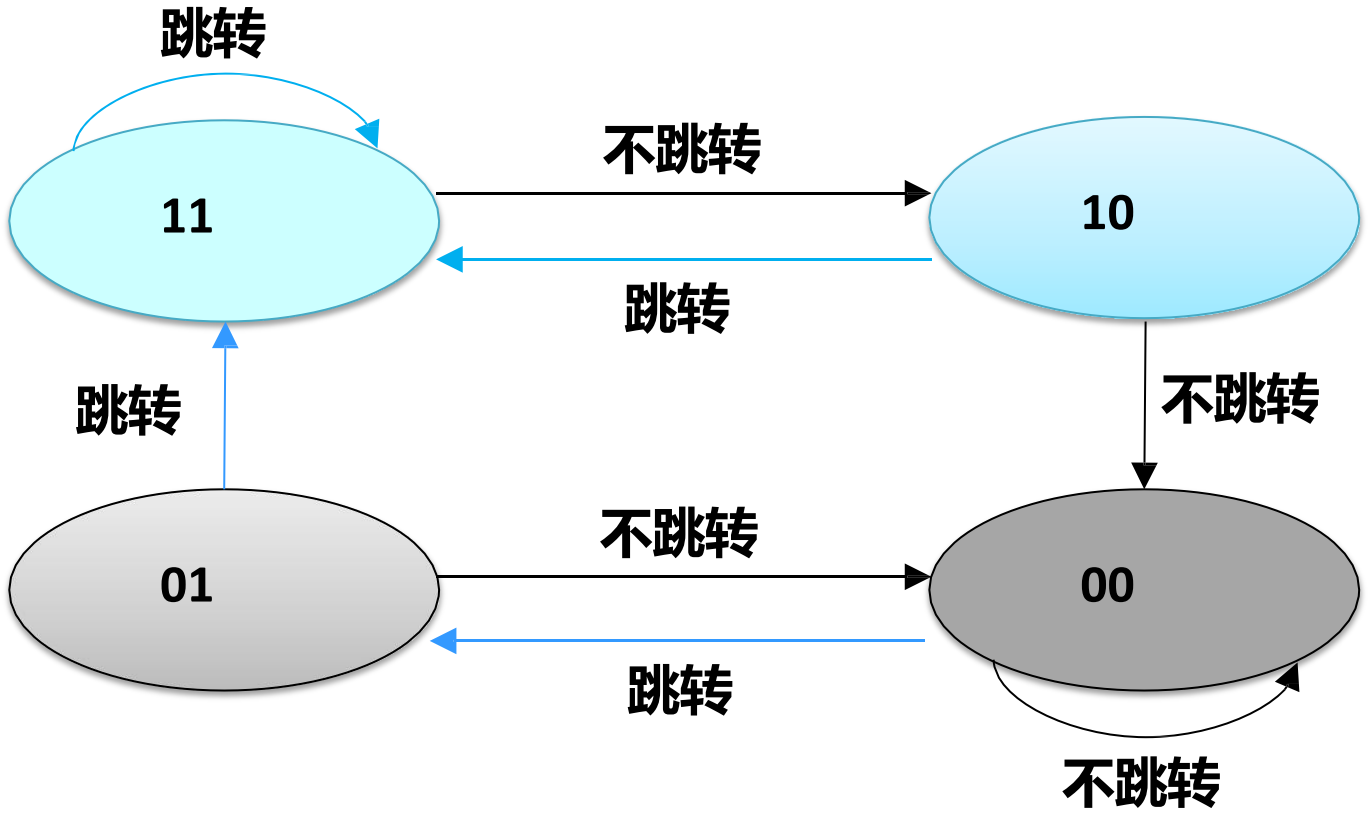
- 空间局部性
  - 相邻的分支指令未跳转，当前指令大概率也不跳转
  - 典型案例，如右图所示，第一条指令不跳转，第二条指令一定不跳转
  - 可以利用队列记录最近执行分支指令的跳转方向

```
if (x[i] < 5)
    y ++;
if (x[i] < 7)
    c ++;
```

■ 分支预测

□ 2 bit分支预测

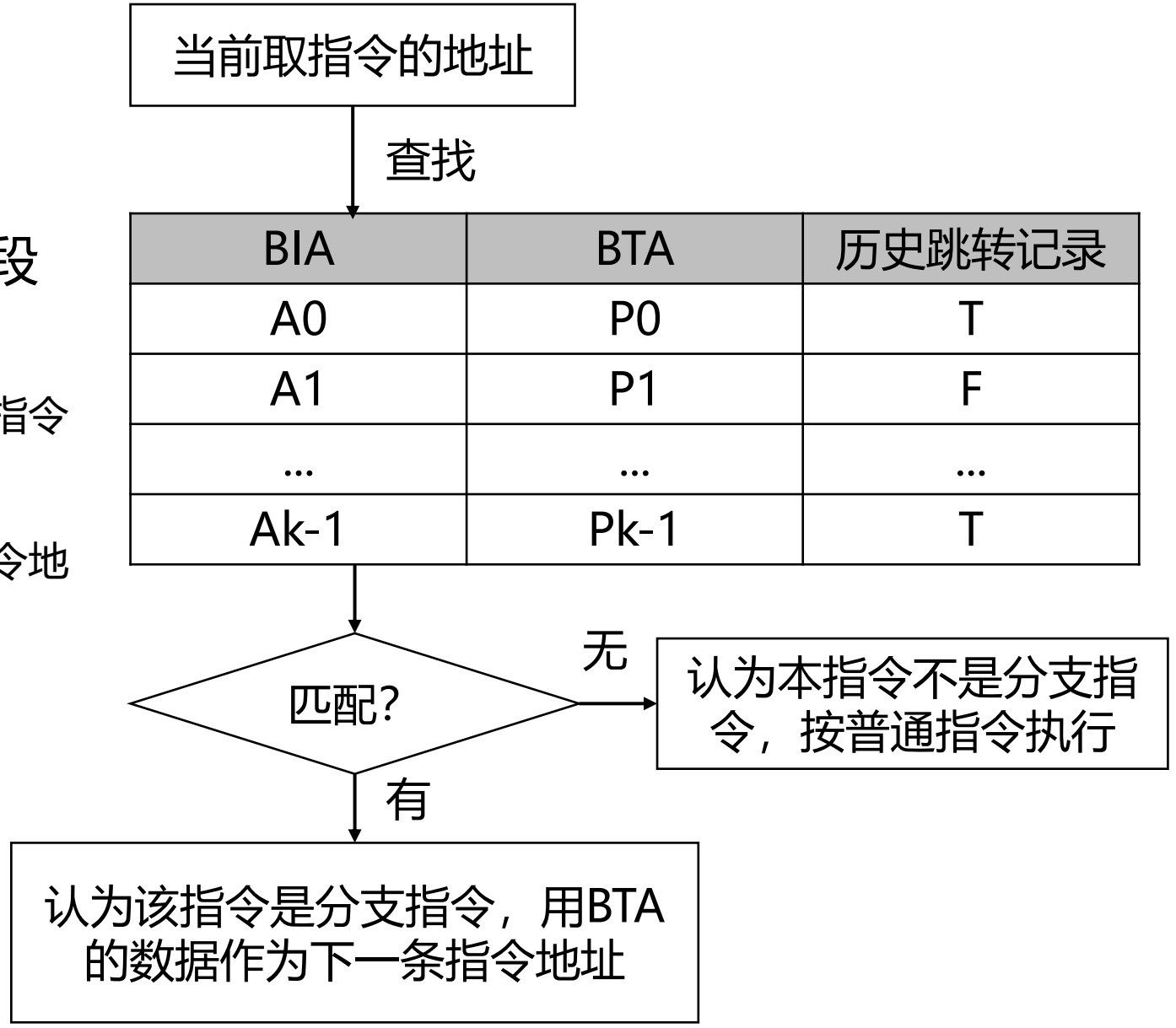
- 在连续两次预测都错误时才更改预测，利用时间局部性
- n bit的分支预测准确率并不比2 bit版本高很多



■ 分支预测

□ 分支目标缓冲(**BTB**, branch target buffer)

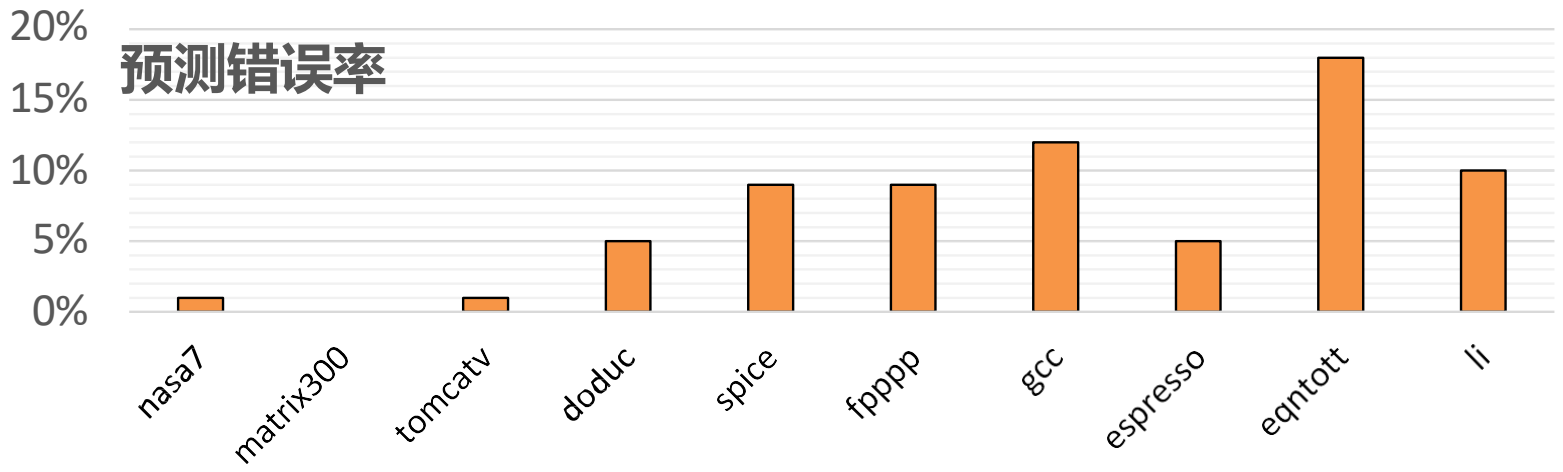
- 分支目标缓存包含两个字段
  - ➢ 分支指令地址：BIA
    - 判断当前指令是否为分支指令
  - ➢ 跳转目标地址：BTA
    - 这条分支指令的下一条指令地址
    - 是预测的下一条地址



■ 分支预测

□ 错误预测恢复

- 顺序处理器
  - ➢ 清除流水线中分支指令后的所有错误执行指令
- 乱序处理器
  - ➢ 释放ROB中错误指令占用的表项



■ 分支预测

□ 精确中断(Precise exception)

- 精确中断需要满足以下条件
  - ➢ 中断与特定的故障指令 Fi 相关联
  - ➢ Fi 之前的所有指令(按程序顺序)均已完成执行
  - ➢ Fi 及之后的所有指令表现为从未开始
- 精确中断与错误预测恢复属于同一类问题
- 保持精确中断需要按序完成
- 需要硬件缓冲区存储未提交指令的结果：
  - ➢ 保证指令完成的顺序和程序逻辑一致
  - ➢ 保证内存访问的顺序和程序逻辑一致

■ 分支预测

□ 重排序缓冲(ROB)

- 简单回忆上一章ROB的设计与工作原理
- 一个以循环队列(FIFO)方式管理的指令缓冲区
  - 包含所有正在执行的指令
  - 在指令执行完成到提交之间，临时保存结果
- 重排序缓冲区的字段：
  - 指令类型：分支/存储/ALU指令
  - 目的地址：提供目标寄存器编号或内存地址
  - 值：在指令提交之前，保存其执行结果

■ 分支预测

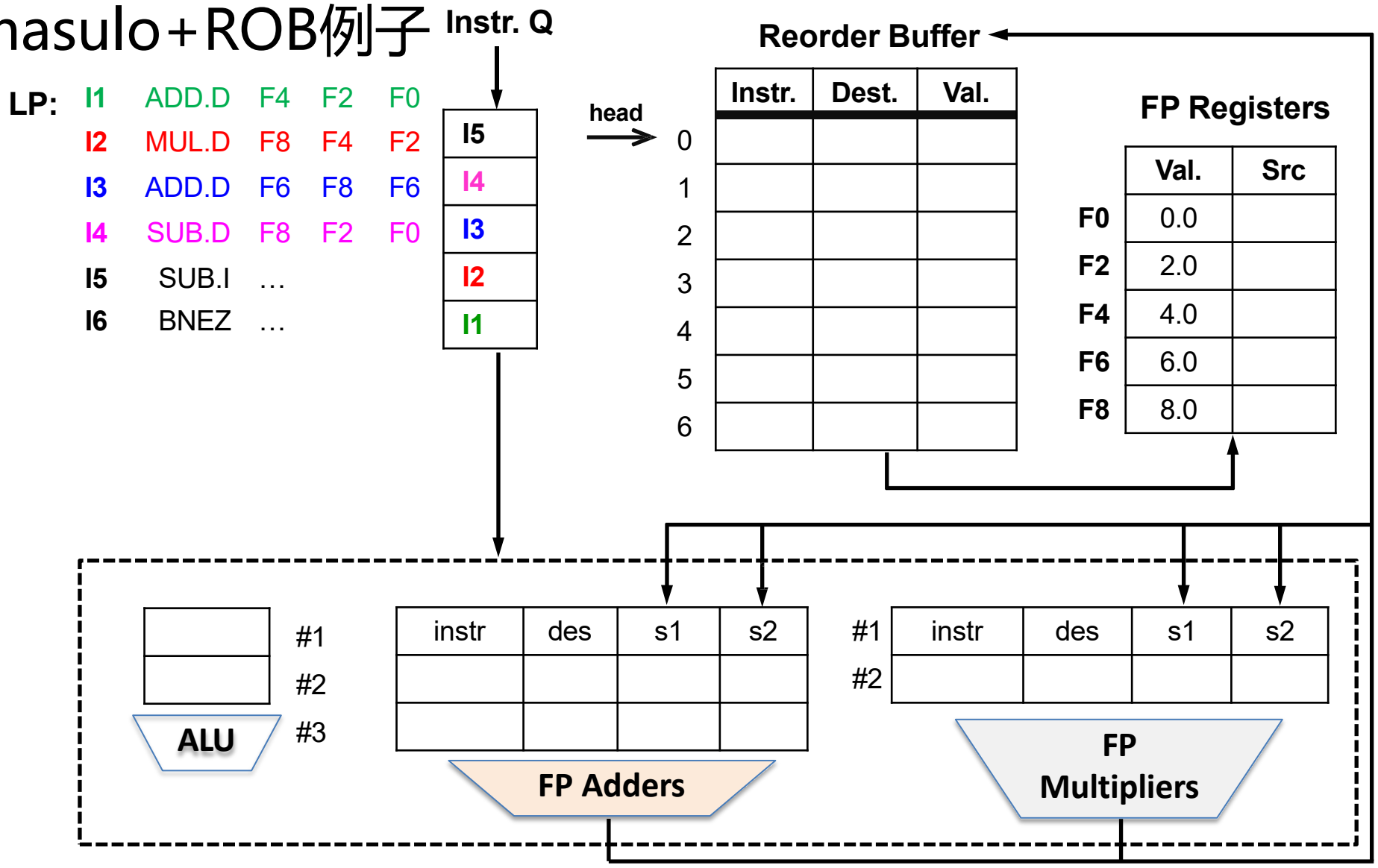
□ Tomasulo+ROB

- 对Tomasulo硬件进行扩展以支持分支预测，产生以下代表性产品
  - PowerPC 603/604/G3/G4
  - MIPS R10000/R12000
  - Intel Pentium II/III/4
  - Alpha 21264
  - AMD K5/K6/Athlon

- 假设一条指令有4个状态：发射(issue)、执行(execute)、写结果(write)、提交(commit)

■ 分支预测

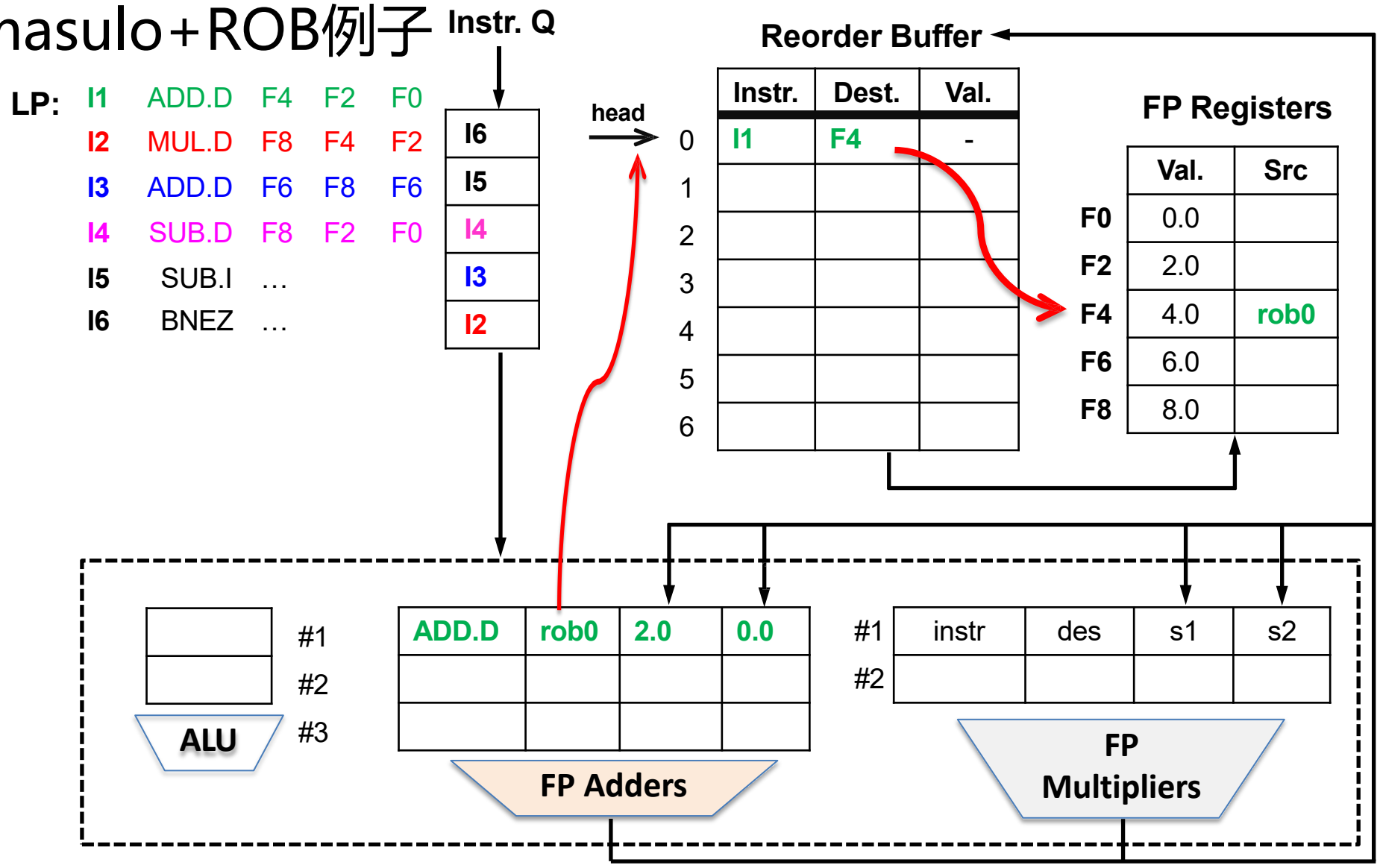
□ Tomasulo+ROB例子



■ 分支预测

□ Tomasulo+ROB例子

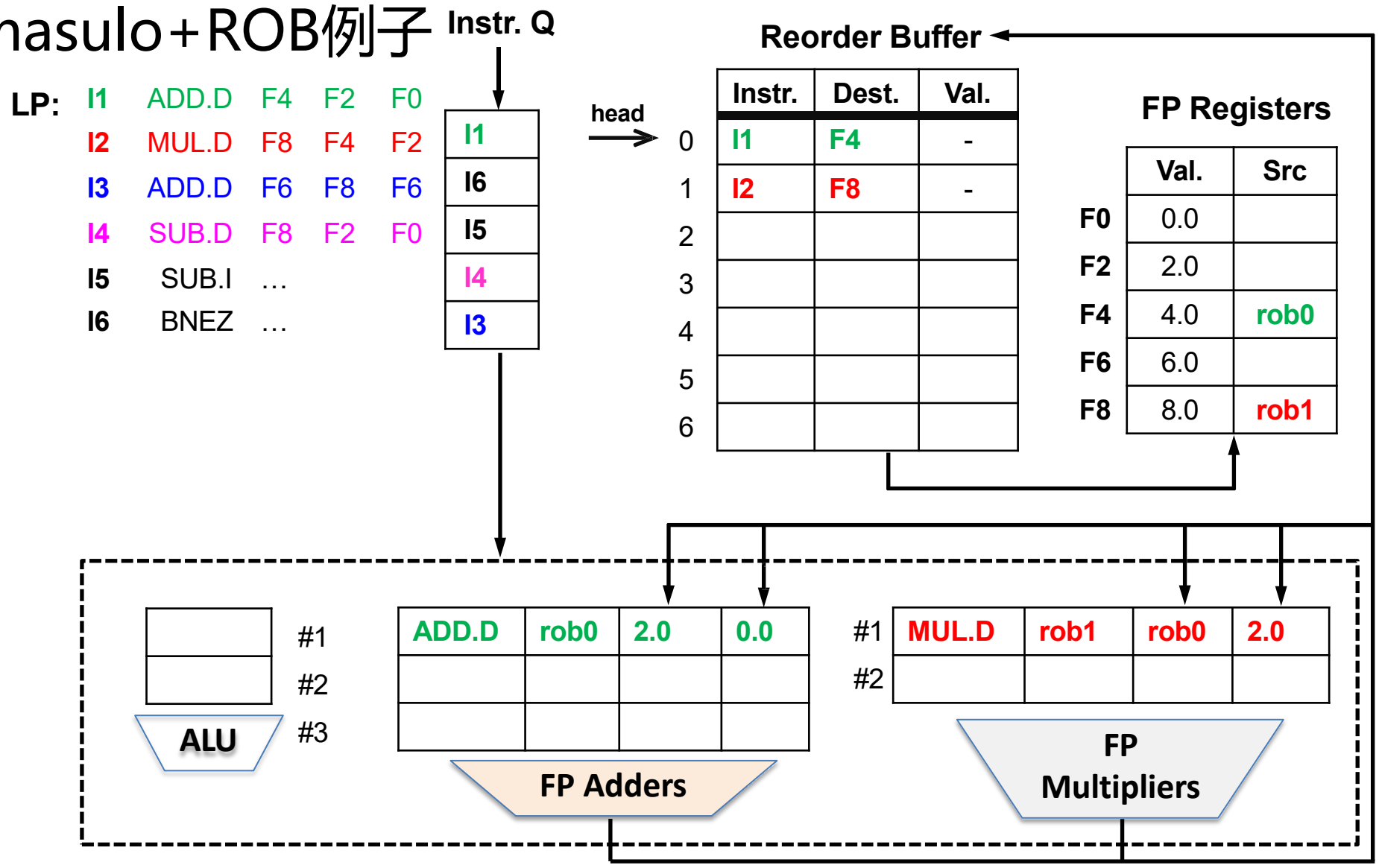
时钟周期 1



■ 分支预测

□ Tomasulo+ROB例子

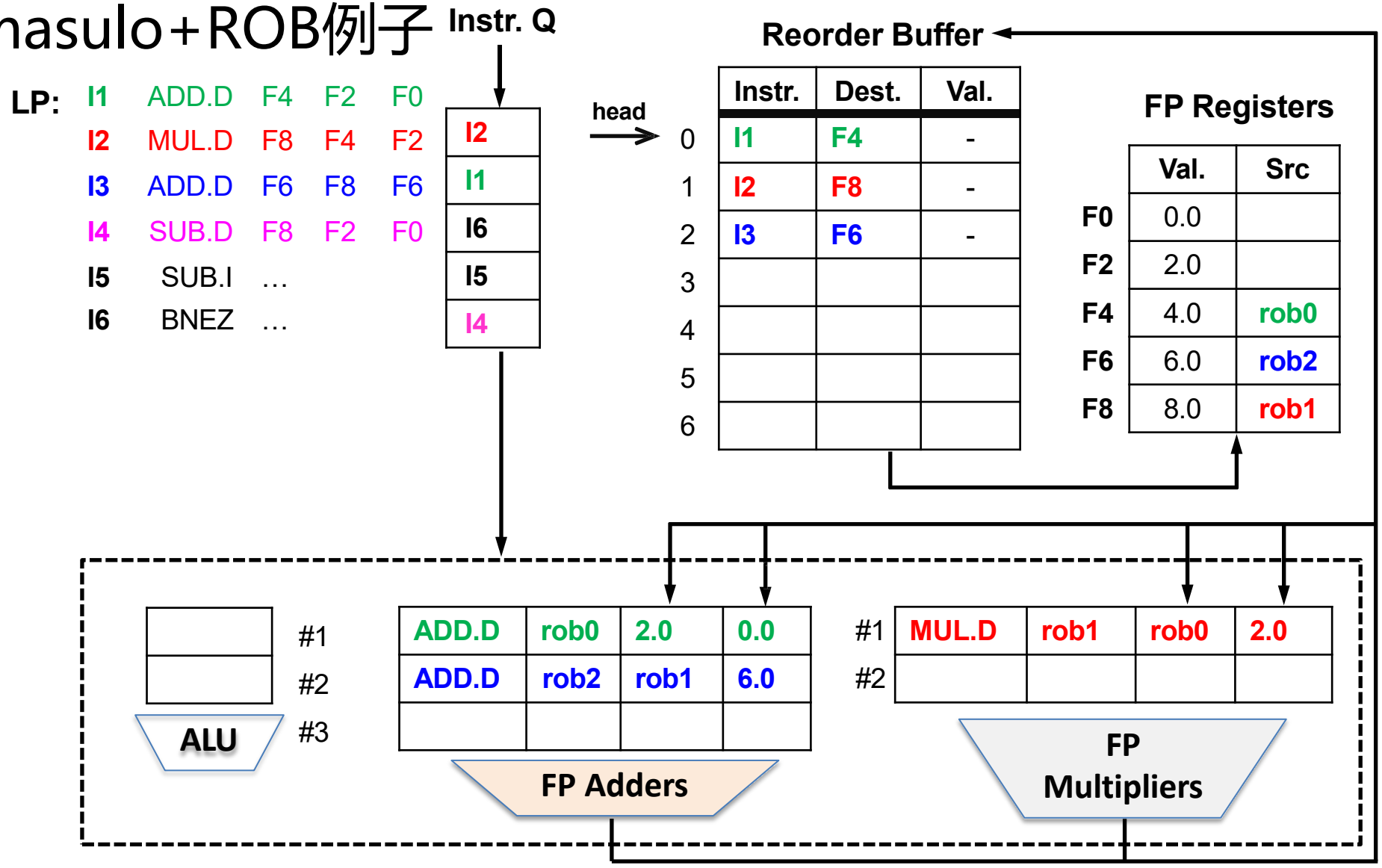
时钟周期 2



■ 分支预测

□ Tomasulo+ROB例子

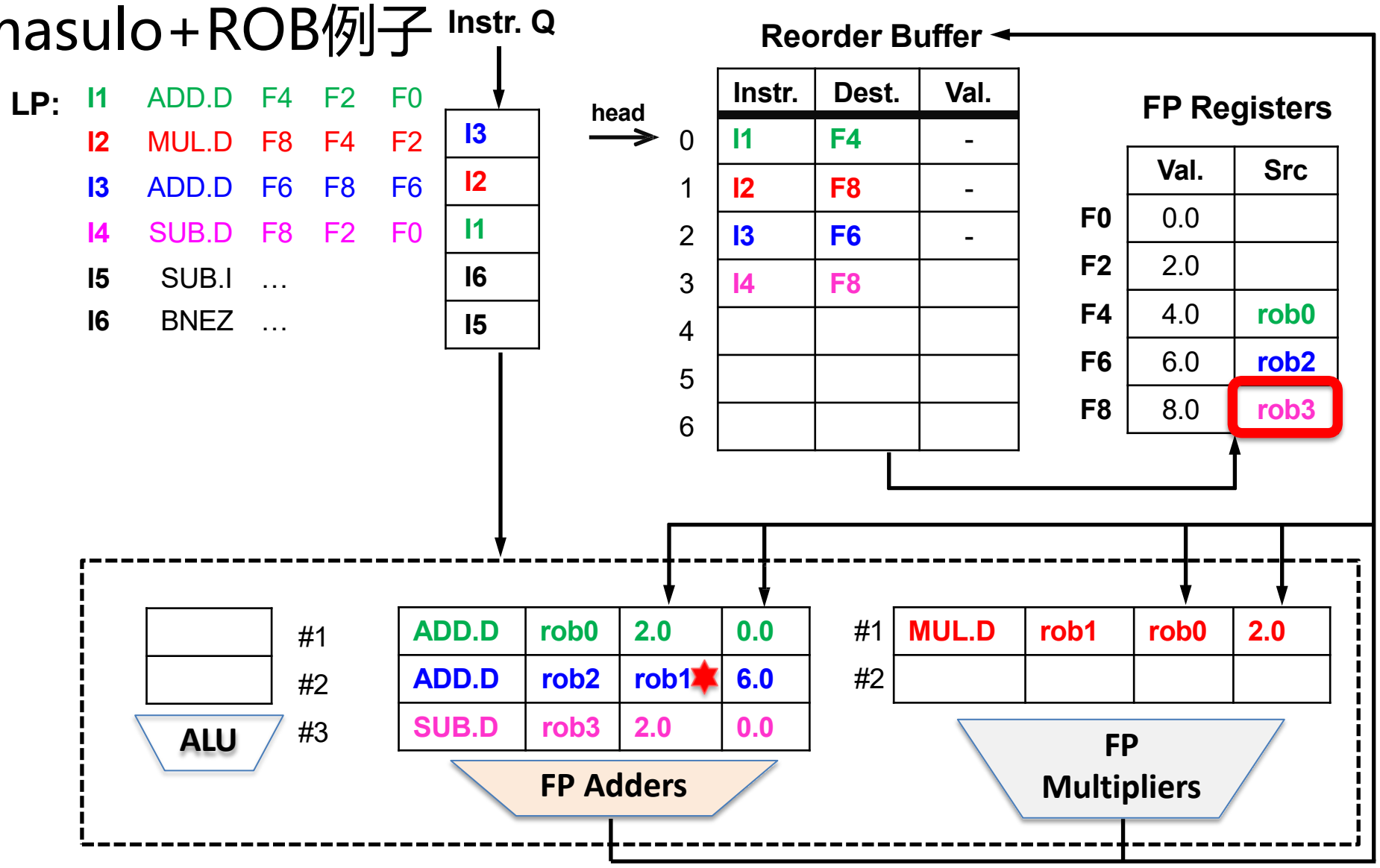
时钟周期 3



■ 分支预测

□ Tomasulo+ROB例子

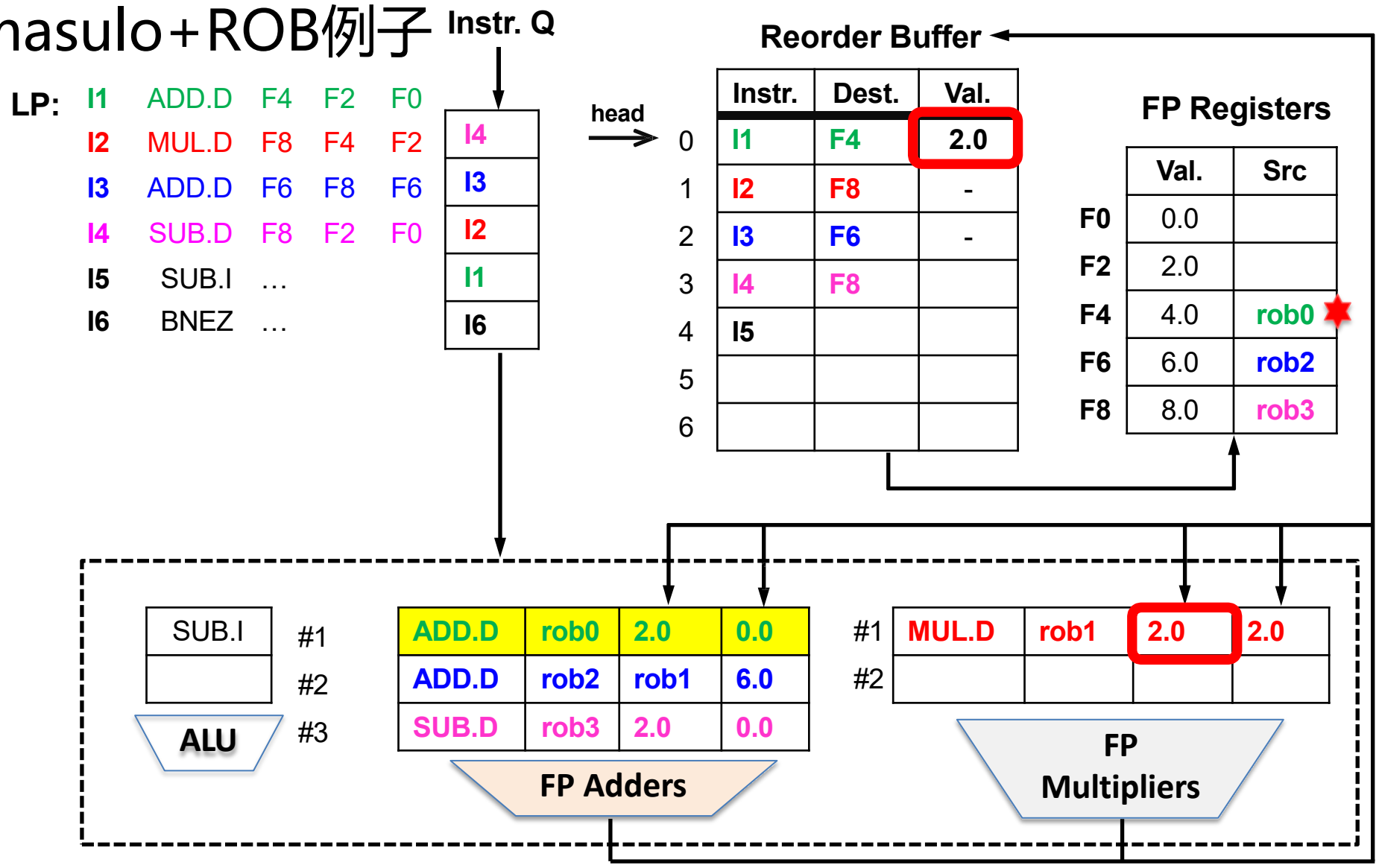
时钟周期 4



■ 分支预测

□ Tomasulo+ROB例子

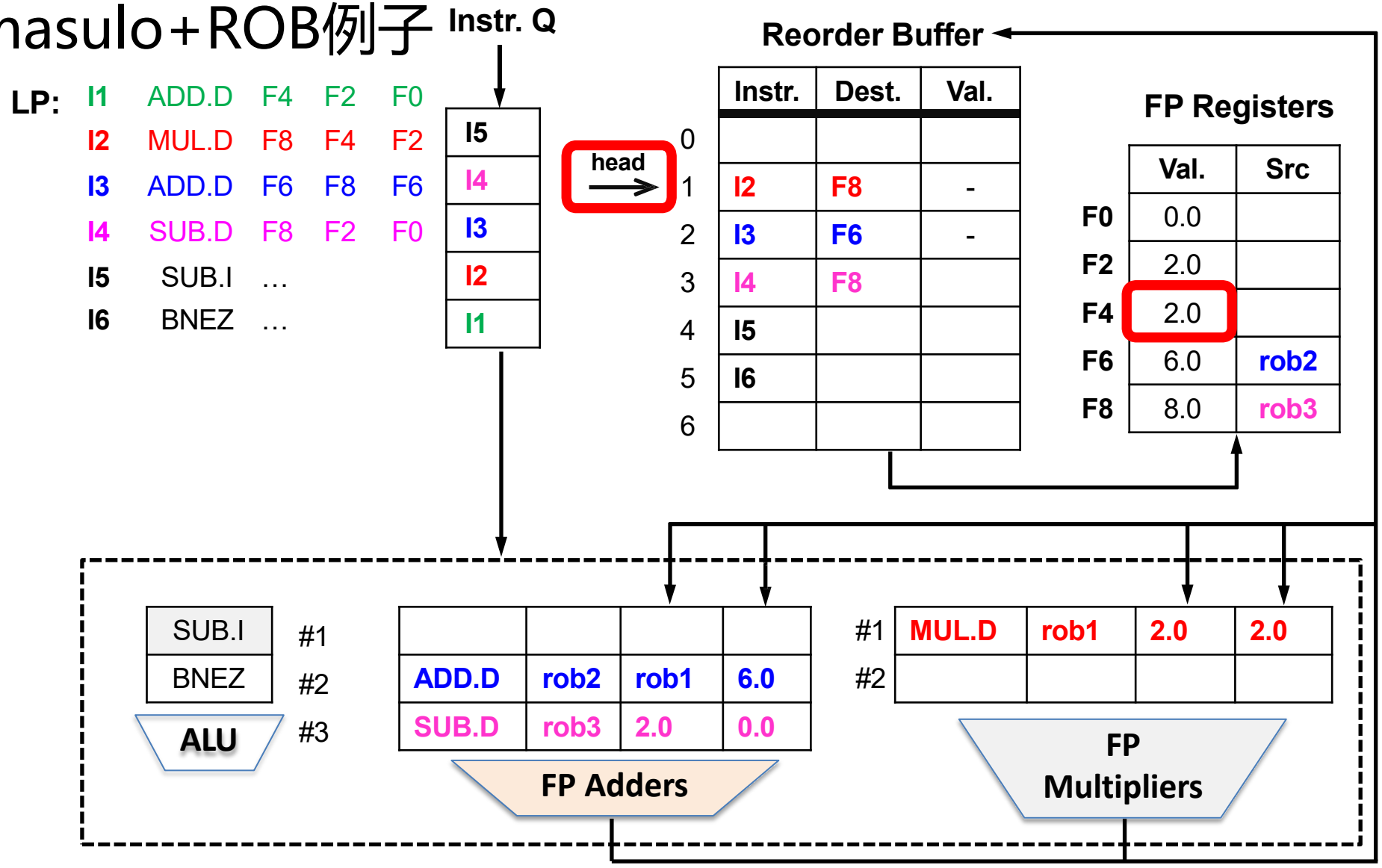
时钟周期 5



■ 分支预测

□ Tomasulo+ROB例子

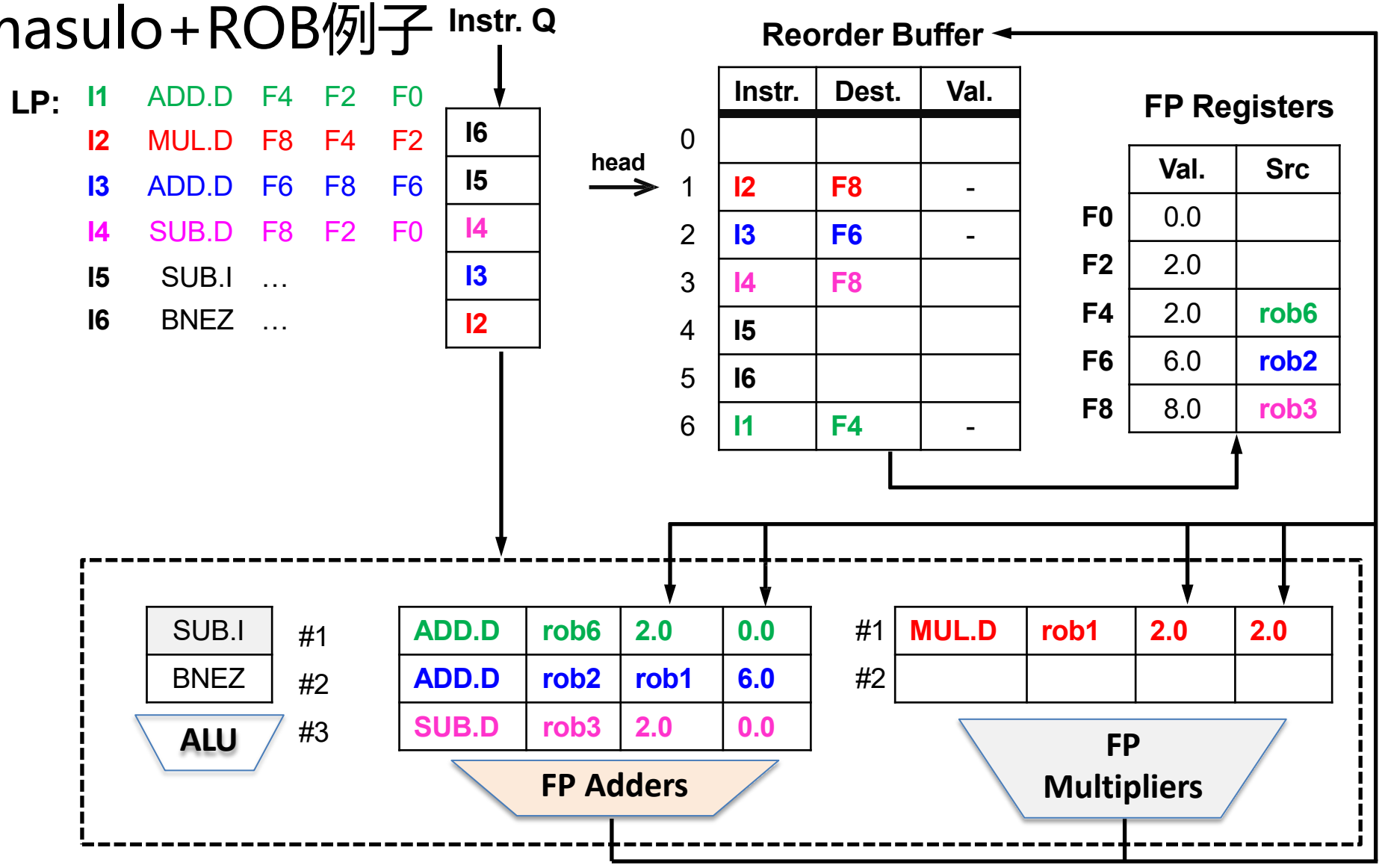
时钟周期 6



■ 分支预测

□ Tomasulo+ROB例子

时钟周期 7

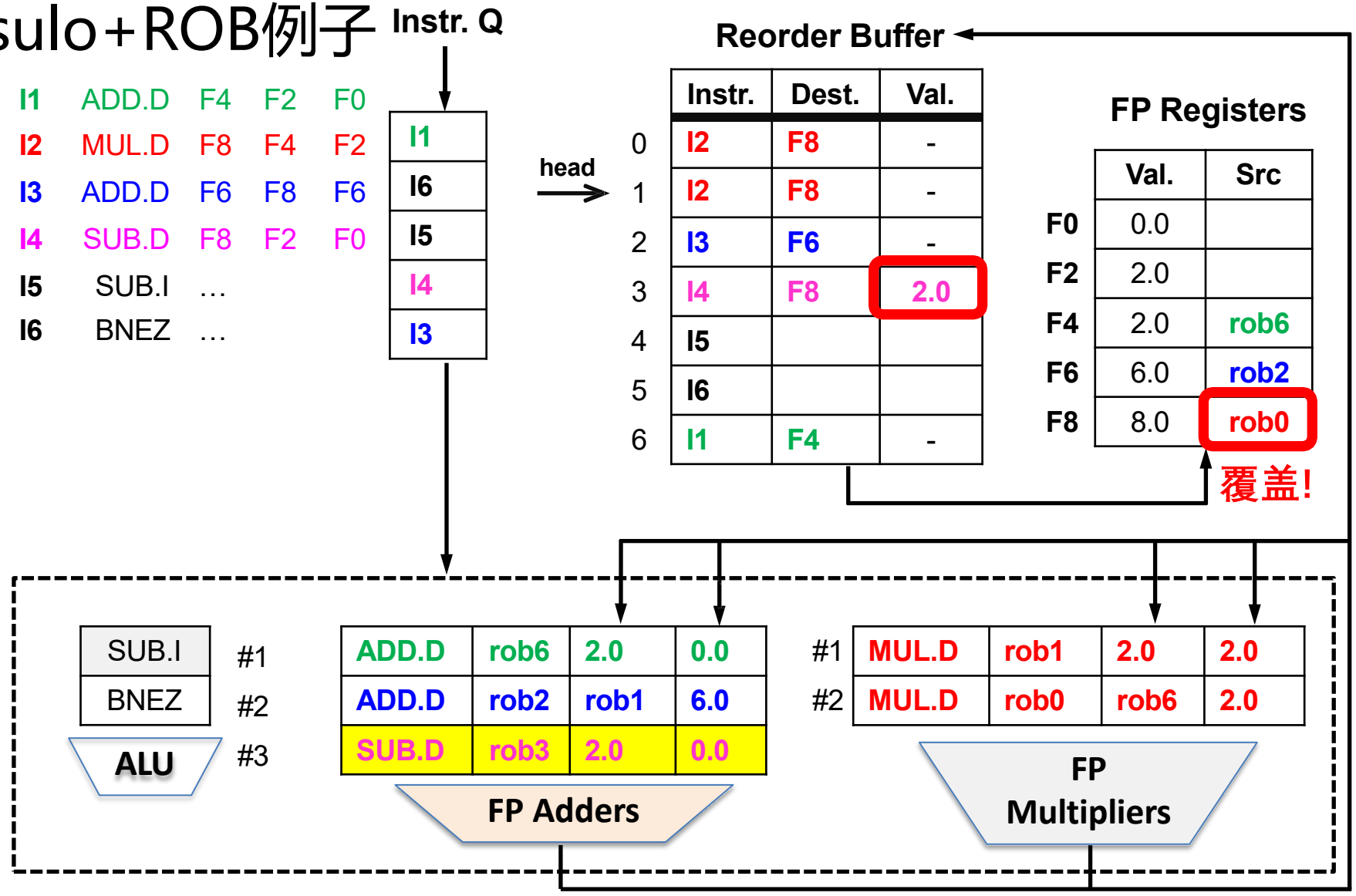


■ 分支预测

□ Tomasulo+ROB例子

时钟周期 8

```
LP:  I1   ADD.D  F4  F2  F0
     I2   MUL.D  F8  F4  F2
     I3   ADD.D  F6  F8  F6
     I4   SUB.D  F8  F2  F0
     I5   SUB.I  …
     I6   BNEZ   …
```



■ 分支预测

□ Tomasulo+ROB例子

时钟周期 9

```
LP:  I1   ADD.D  F4  F2  F0
     I2   MUL.D  F8  F4  F2
     I3   ADD.D  F6  F8  F6
     I4   SUB.D  F8  F2  F0
     I5   SUB.I  …
     I6   BNEZ   …
```

Instr. Q

I1
I6
I5
I4
I3

head

Reorder Buffer

	Instr.	Dest.	Val.
0	I2	F8	-
1	I2	F8	-
2	I3	F6	-
3	I4	F8	2.0
4	I5		
5	I6		
6	I1	F4	-

FP Registers

	Val.	Src
F0	0.0	
F2	2.0	
F4	2.0	rob6
F6	6.0	rob2
F8	8.0	rob0

ROB已满!

SUB.I	#1	ADD.D	rob6	2.0	0.0
BNEZ	#2	ADD.D	rob2	rob1	6.0
ALU	#3				

FP Adders

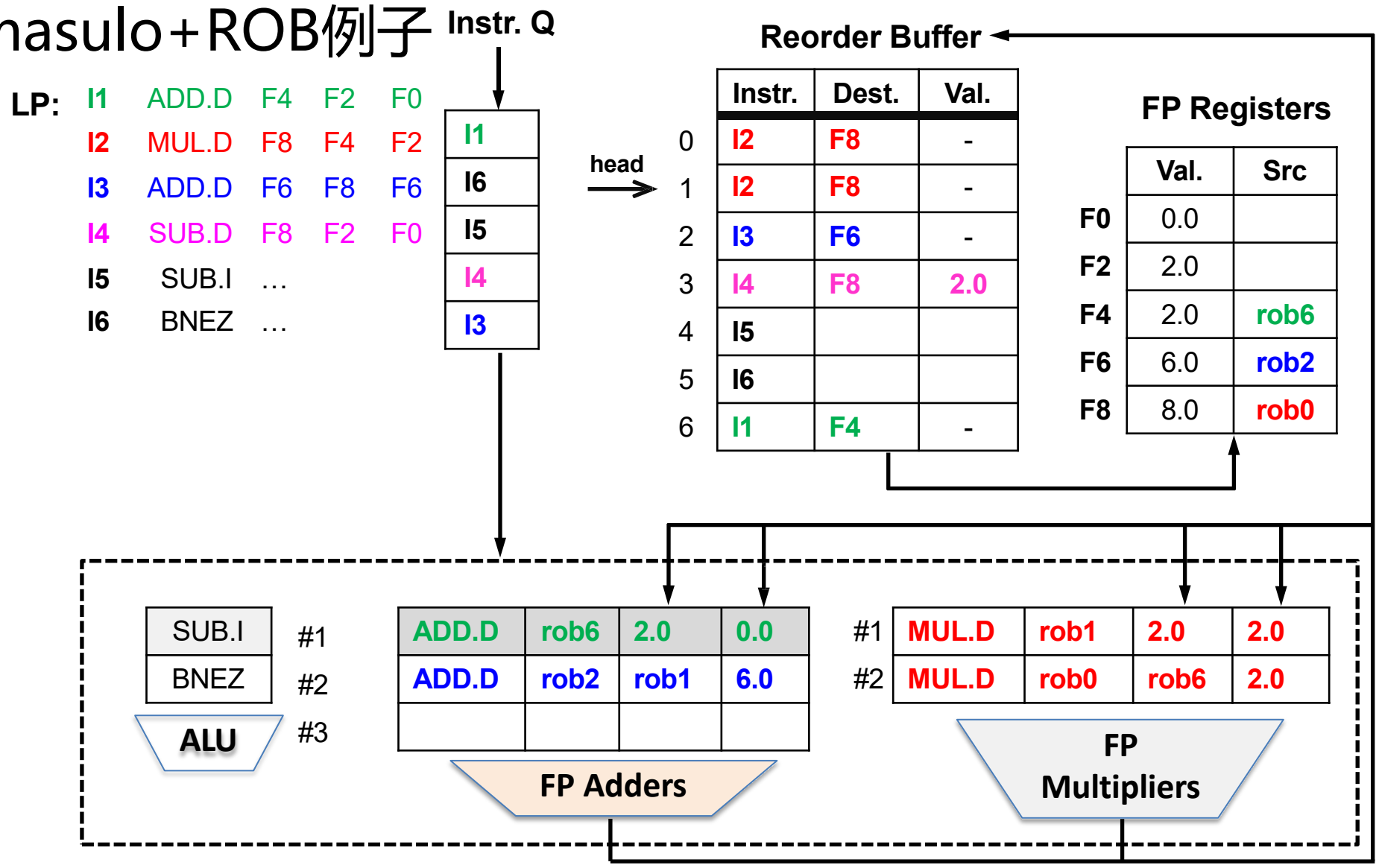
#1	MUL.D	rob1	2.0	2.0
#2	MUL.D	rob0	rob6	2.0

FP Multipliers

■ 分支预测

□ Tomasulo+ROB例子

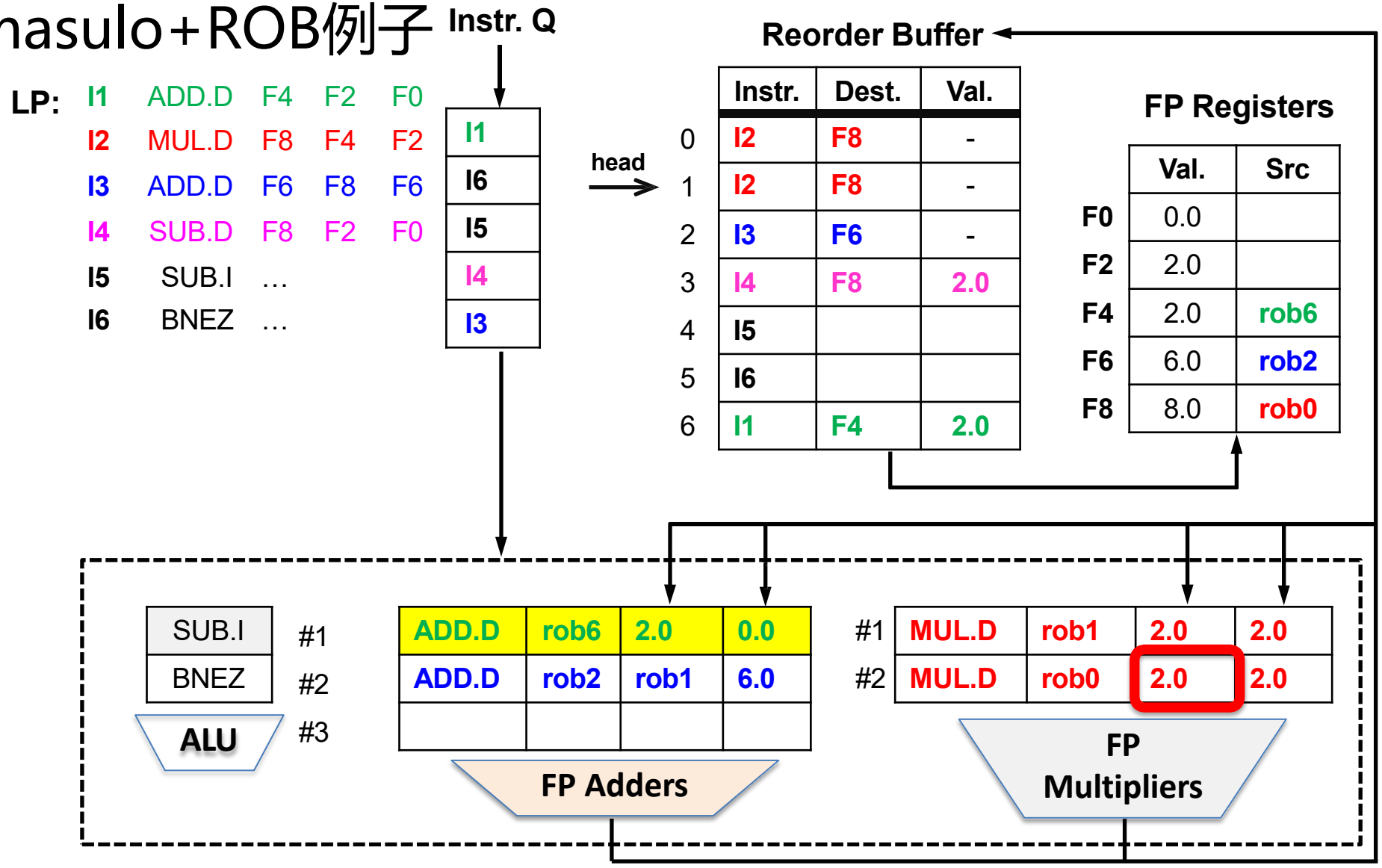
时钟周期 10



■ 分支预测

□ Tomasulo+ROB例子

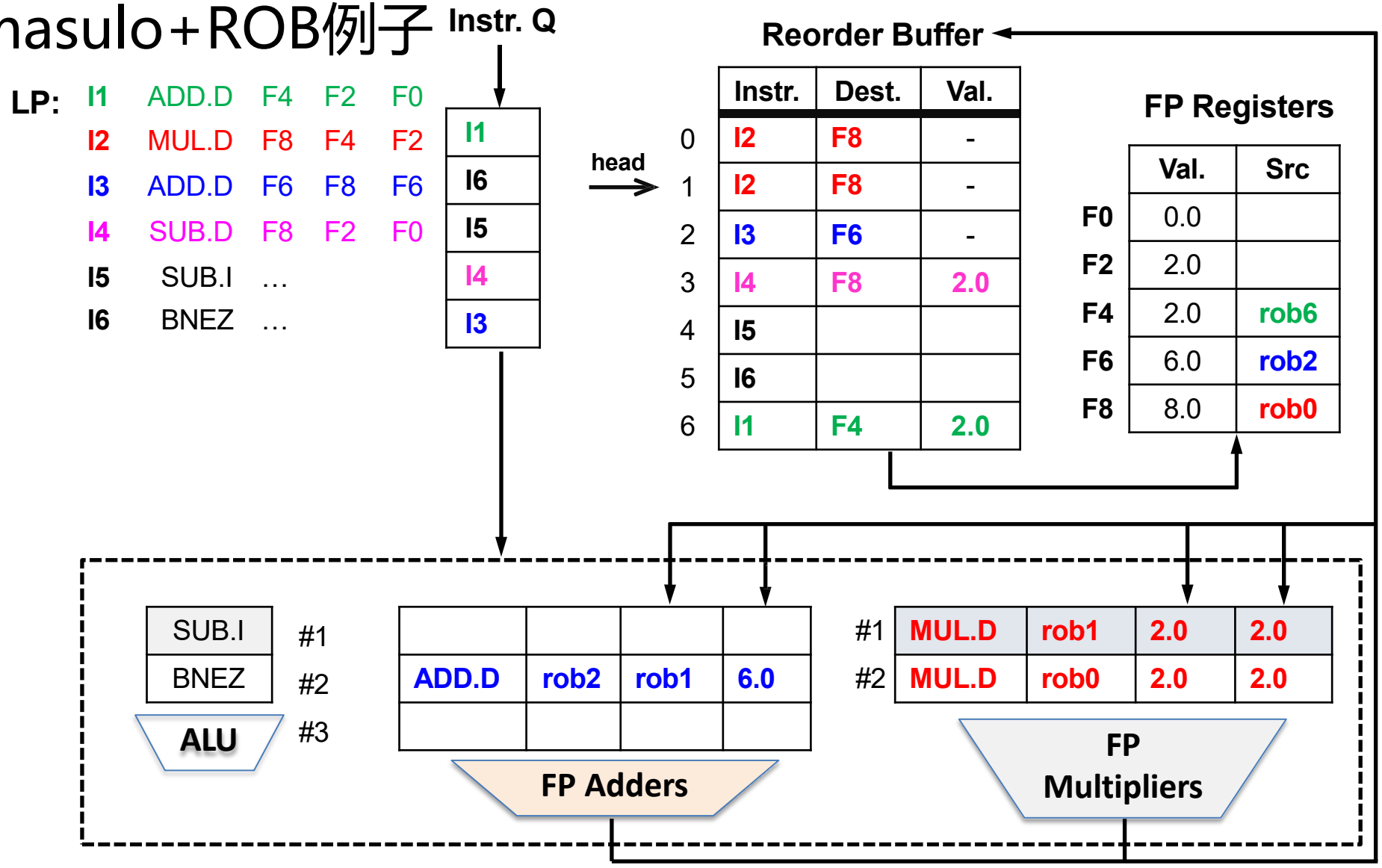
时钟周期 11



■ 分支预测

□ Tomasulo+ROB例子

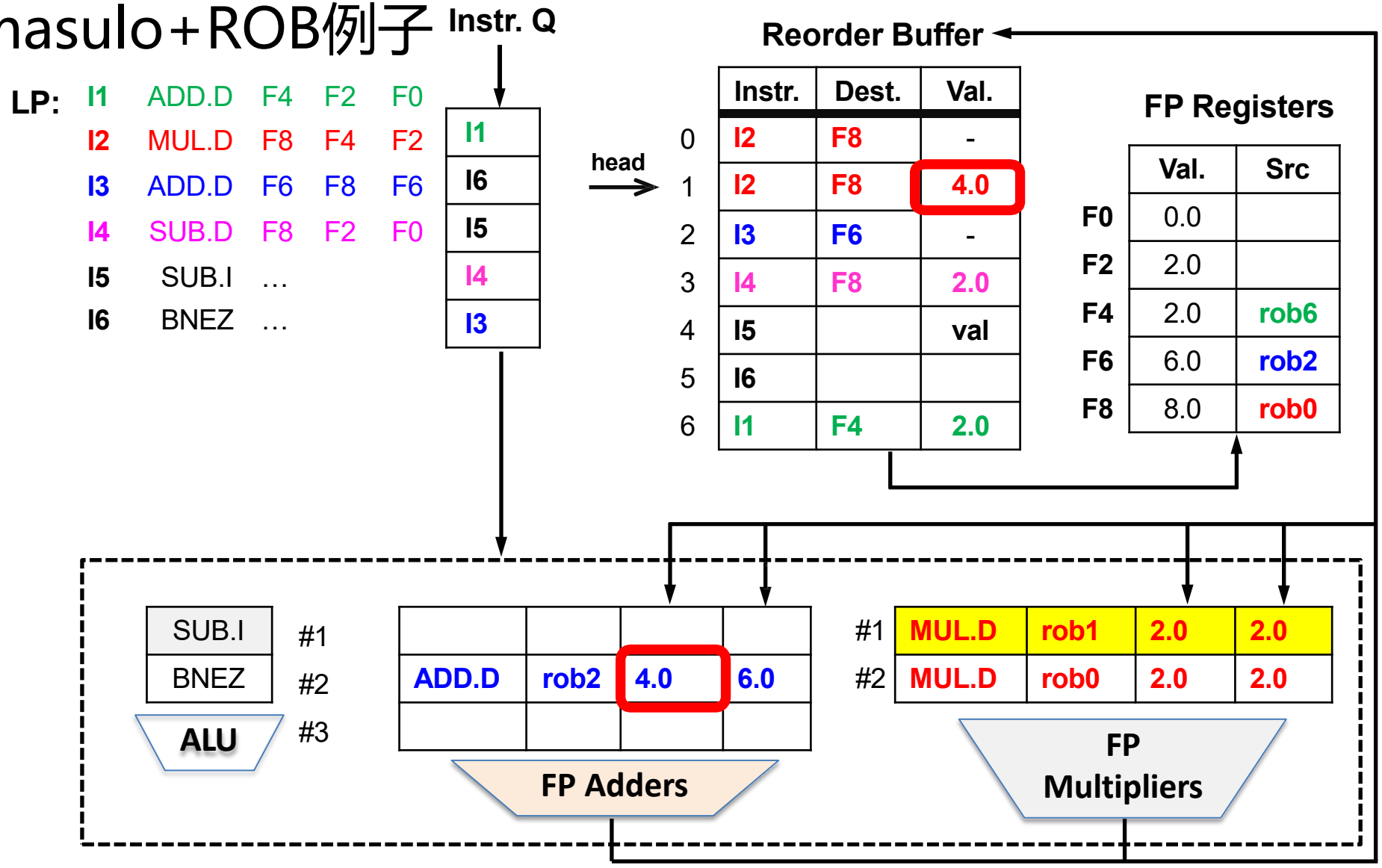
时钟周期 16



■ 分支预测

□ Tomasulo+ROB例子

时钟周期 17



■ 分支预测

□ Tomasulo+ROB例子

时钟周期 18

```
LP:  I1   ADD.D  F4  F2  F0
     I2   MUL.D  F8  F4  F2
     I3   ADD.D  F6  F8  F6
     I4   SUB.D  F8  F2  F0
     I5   SUB.I  …
     I6   BNEZ   …
```

Instr. Q

I2
I1
I6
I5
I4

head

Reorder Buffer

	Instr.	Dest.	Val.
0	I2	F8	-
1	I3	F6	-
2	I3	F6	-
3	I4	F8	2.0
4	I5		val
5	I6		
6	I1	F4	2.0

FP Registers

	Val.	Src
F0	0.0	
F2	2.0	
F4	2.0	rob6
F6	6.0	rob2
F8	4.0	rob0

分支错误预测!

SUB.I	#1
BNEZ	#2
ALU	#3

ADD.D	rob1	rob0	rob2
ADD.D	rob2	4.0	6.0

FP Adders

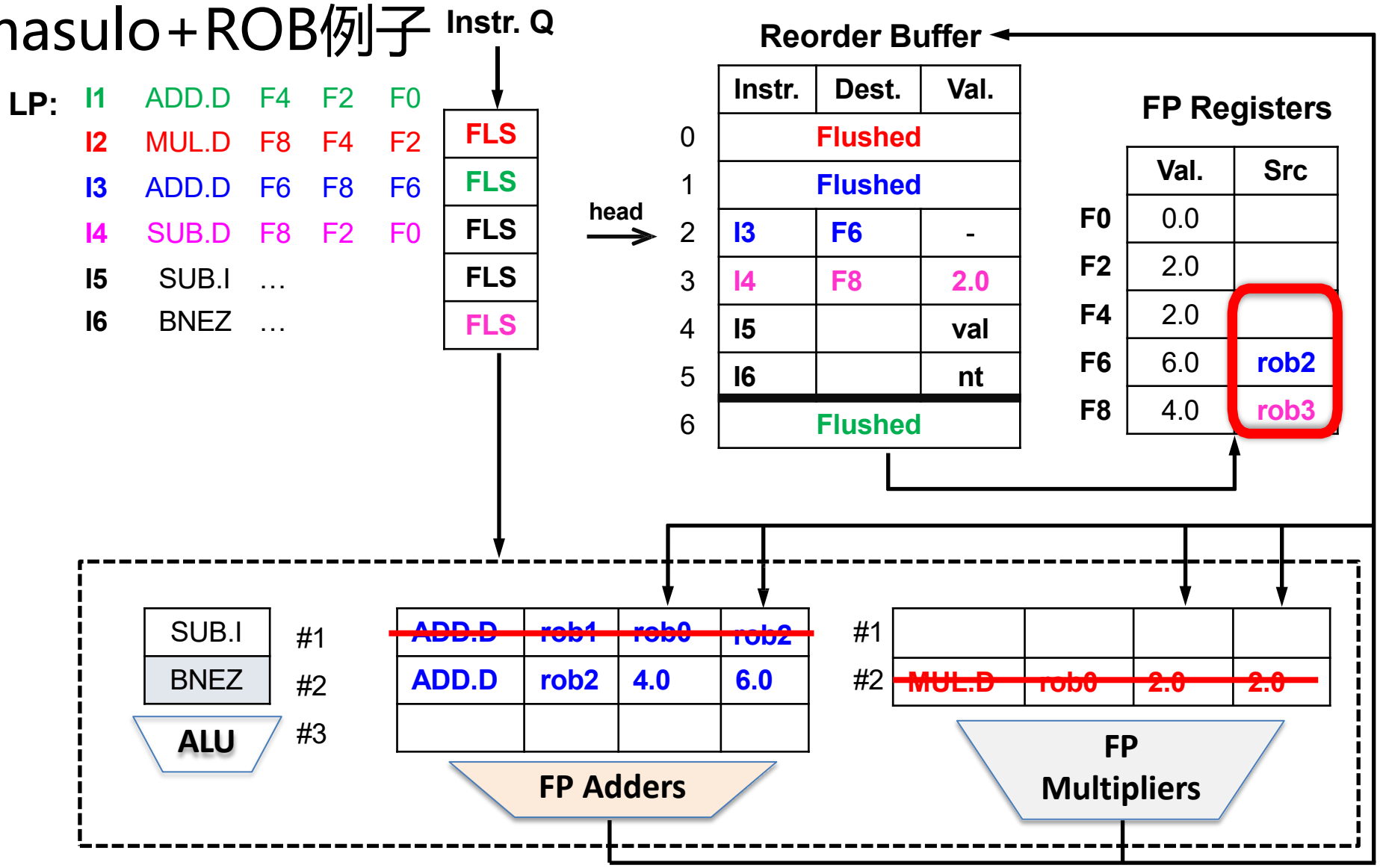
#1				
#2	MUL.D	rob0	2.0	2.0

FP Multipliers

■ 分支预测

□ Tomasulo+ROB例子

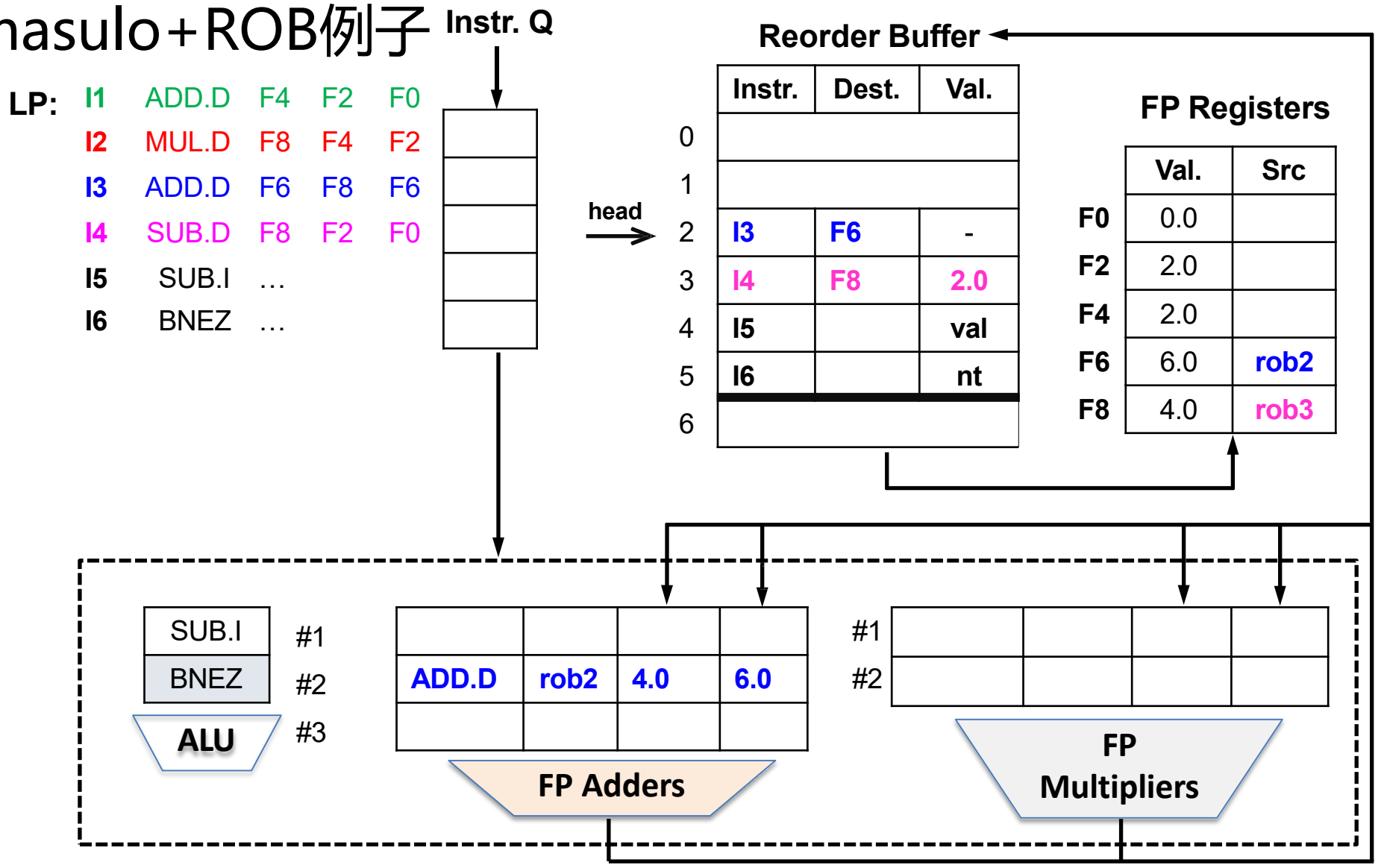
时钟周期 19



■ 分支预测

□ Tomasulo+ROB例子

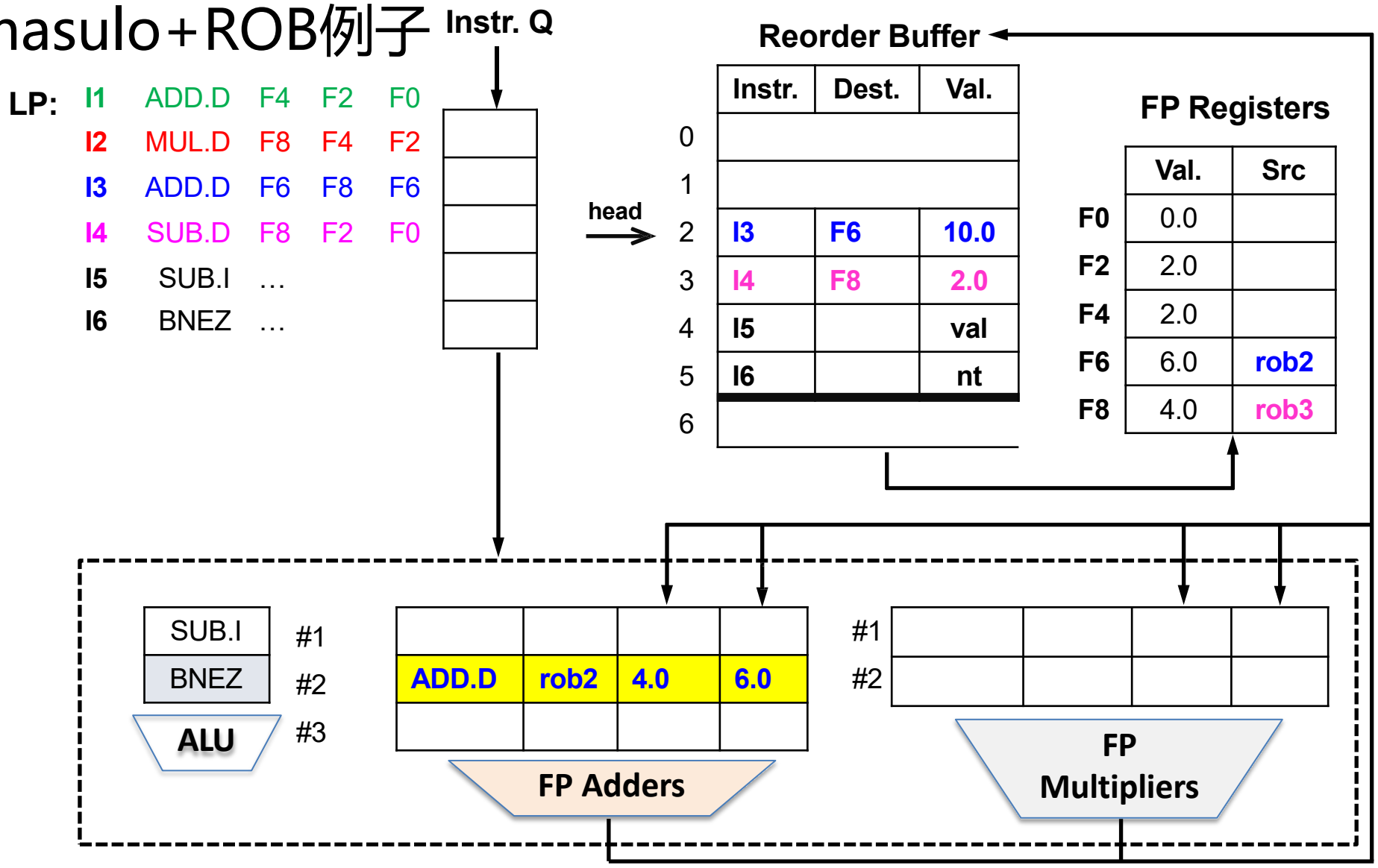
时钟周期 20



■ 分支预测

□ Tomasulo+ROB例子

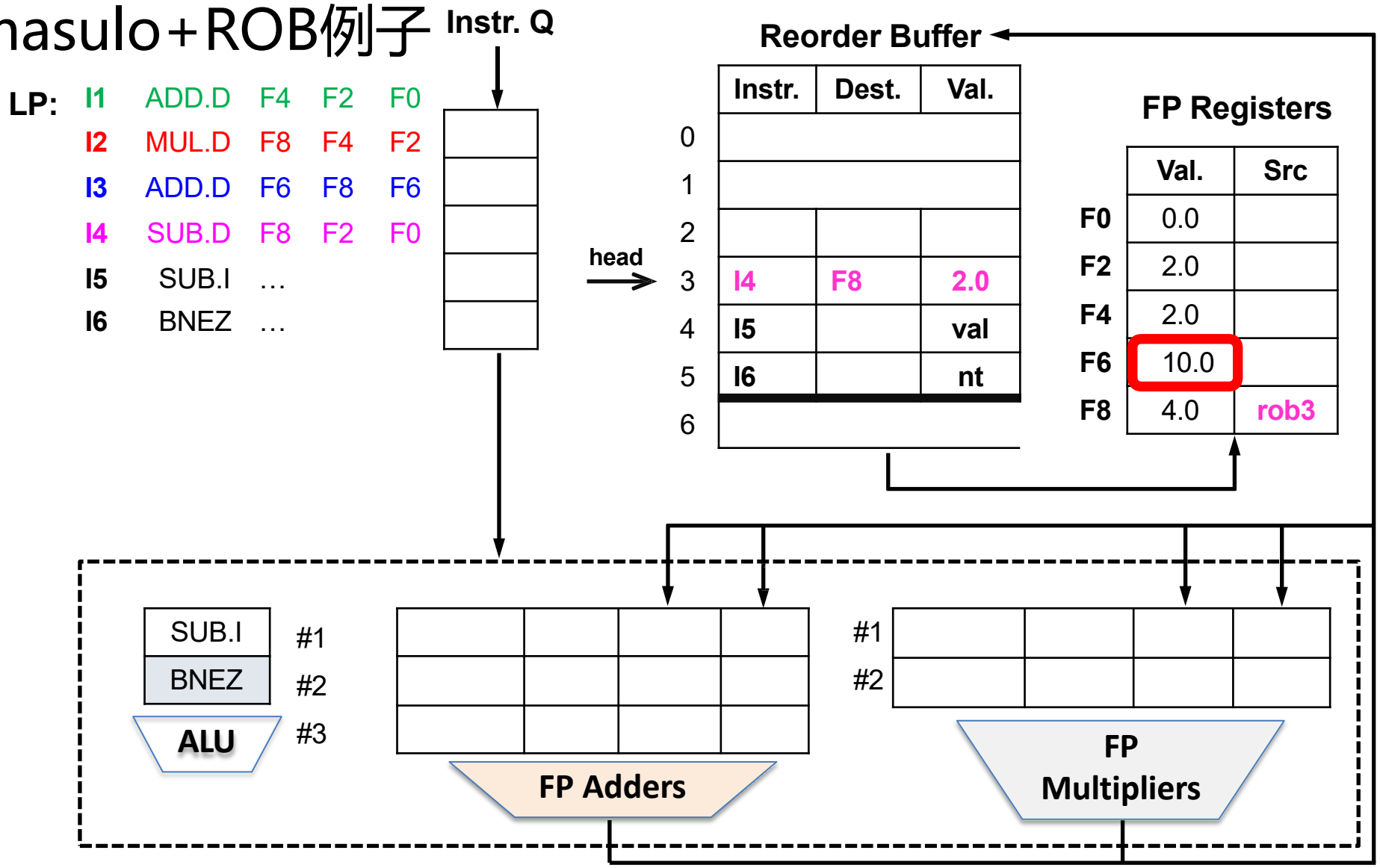
时钟周期 21



■ 分支预测

□ Tomasulo+ROB例子

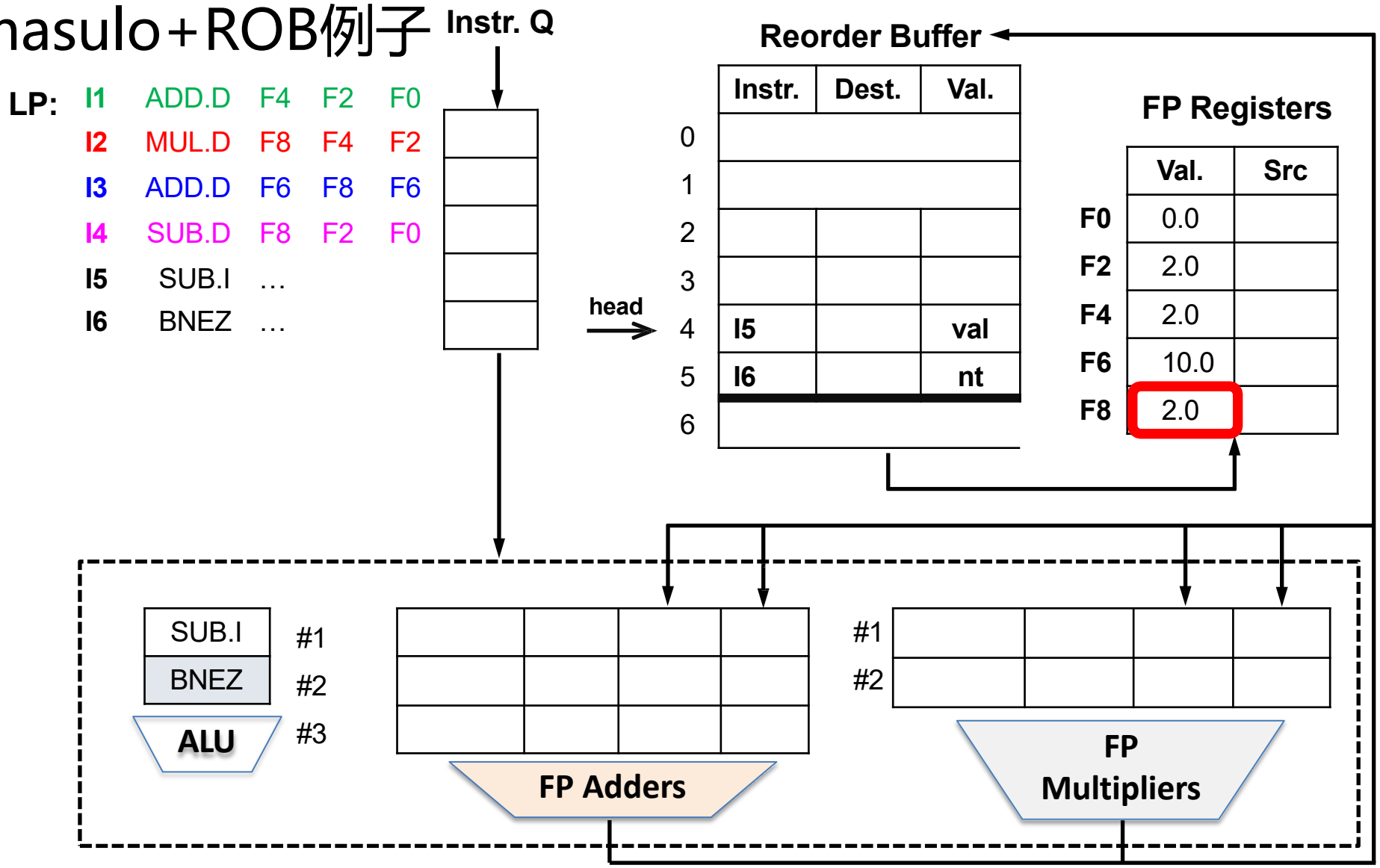
时钟周期 22



■ 分支预测

□ Tomasulo+ROB例子

时钟周期 23



PART 03 VLIW和EPIC

■ VLIW和EPIC

□ VLIW处理器

- 超长指令字处理器
  - 多个固定数量的操作被组合为一条超长指令
  - 如今常见情况是一条指令包含 3 个操作(例如，Intel Itanium 处理器)
  - 在 20 世纪 80 年代，一些初创公司曾尝试将 VLIW 架构商业化并推向市场
- 设计目标：以更低的硬件复杂度实现高性能
  - 减少多发射所需的硬件支持
  - 简化指令派发逻辑
  - 无需结构冒险检测电路
    - 检测数据冒险、结构冒险的工作由编译器完成

■ VLIW和EPIC

□ 架构对比

分类	CISC	RISC	VLIW
指令长度	可变	固定（通常为32位）	固定
指令格式	字段位置可变	规则，字段位置一致	
指令语义	复杂；可能每条指令包含多个相关操作	几乎总是单一简单操作	多个独立的简单操作
寄存器	较少，有时专用	较多，通用	
内存	寄存器-内存架构	加载/存储架构	
硬件	利用微码(microcode)	非微码实现	

■ VLIW和EPIC

□ VLIW处理器的编译器支持

- VLIW架构将调度复杂性从硬件转移到编译器
  - 由编译器生成每条 VLIW 指令字
  - 检测冲突并隐藏延迟
  - 优化以填充指令槽位
- 结构冒险
  - 不能有两个操作被分配到同一个功能单元
  - 不能有两个操作访问同一个内存地址
- 数据冒险
  - 一个指令包内的操作之间不能存在数据依赖
- 控制冒险
  - 静态分支预测

■ VLIW和EPIC

□ 循环展开

C代码

```
for (int i = 1000; i>0; i=i-1)
      x[i] = x[i] + s;
```

没有调度优化的执行顺序 **时钟周期**

```
Loop: L.D       F0, 0(R1)       1
      stall                     2
      ADD.D     F4, F0, F2      3
      stall                     4
      stall                     5
      S.D       F4, 0(R1)       6
      DADDUI    R1, R1, #-8     7
      stall                     8
      BNE       R1, R2, Loop    9
      stall                     10
```

汇编代码

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

■ VLIW和EPIC

□ 循环展开

– 循环展开将循环体复制多次，并相应地调整循环终止条件

```
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
      S.D      F12, -16(R1)   11
      DADDUI   R1, R1, #-32   12
      BNEZ     R1, Loop       13
      S.D      F16, 8(R1)     14
```

L.D到ADD.D需要1个cycle

ADD.D到S.D需要2个cycle

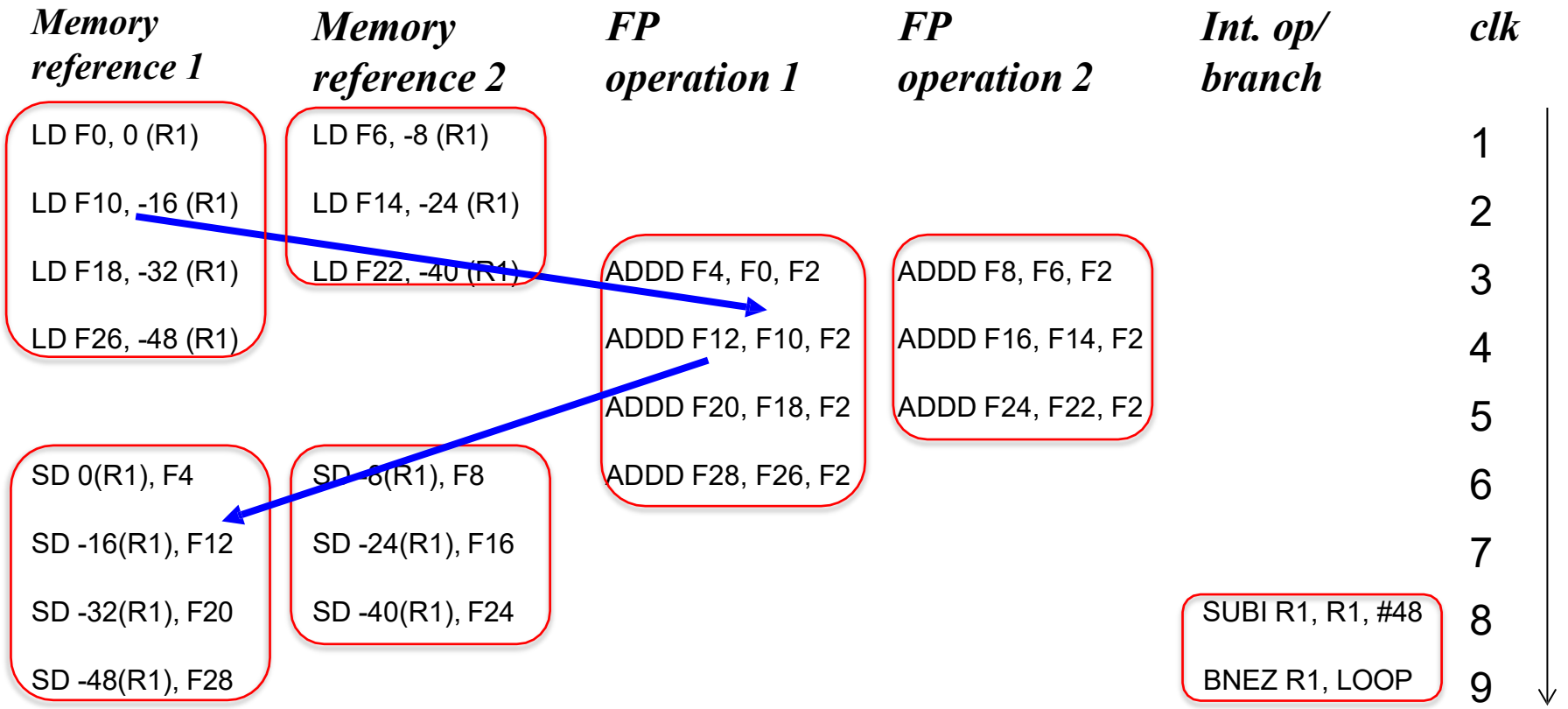
—共需要14个时钟周期，每次循环需3.5个时钟周期

8-32=-24

■ VLIW和EPIC

□ VLIW中的循环展开

– 展开7次避免流水线停顿，9个周期完成7次循环，平均1.3个周期一次循环



■ VLIW和EPIC

- 显式并行指令计算(**EPIC**, Explicitly Parallel Instruction Computing)
  - EPIC是 VLIW 的演进，吸收了许多超标量处理器的优秀设计理念
    - 提供在编译时设计理想执行计划的能力
    - 提供使编译器能够“利用统计规律进行优化”的特性
    - 提供将执行计划传递给硬件的能力
  - Intel/HP Itanium(IA-64)遵循 EPIC 设计理念
    - 6路并行，10级流水线，运行频率800MHz

感谢！