

Ch6: Instruction Pipeline

指令流水线

第一讲 流水线数据通路和控制逻辑

第二讲 流水线冒险处理

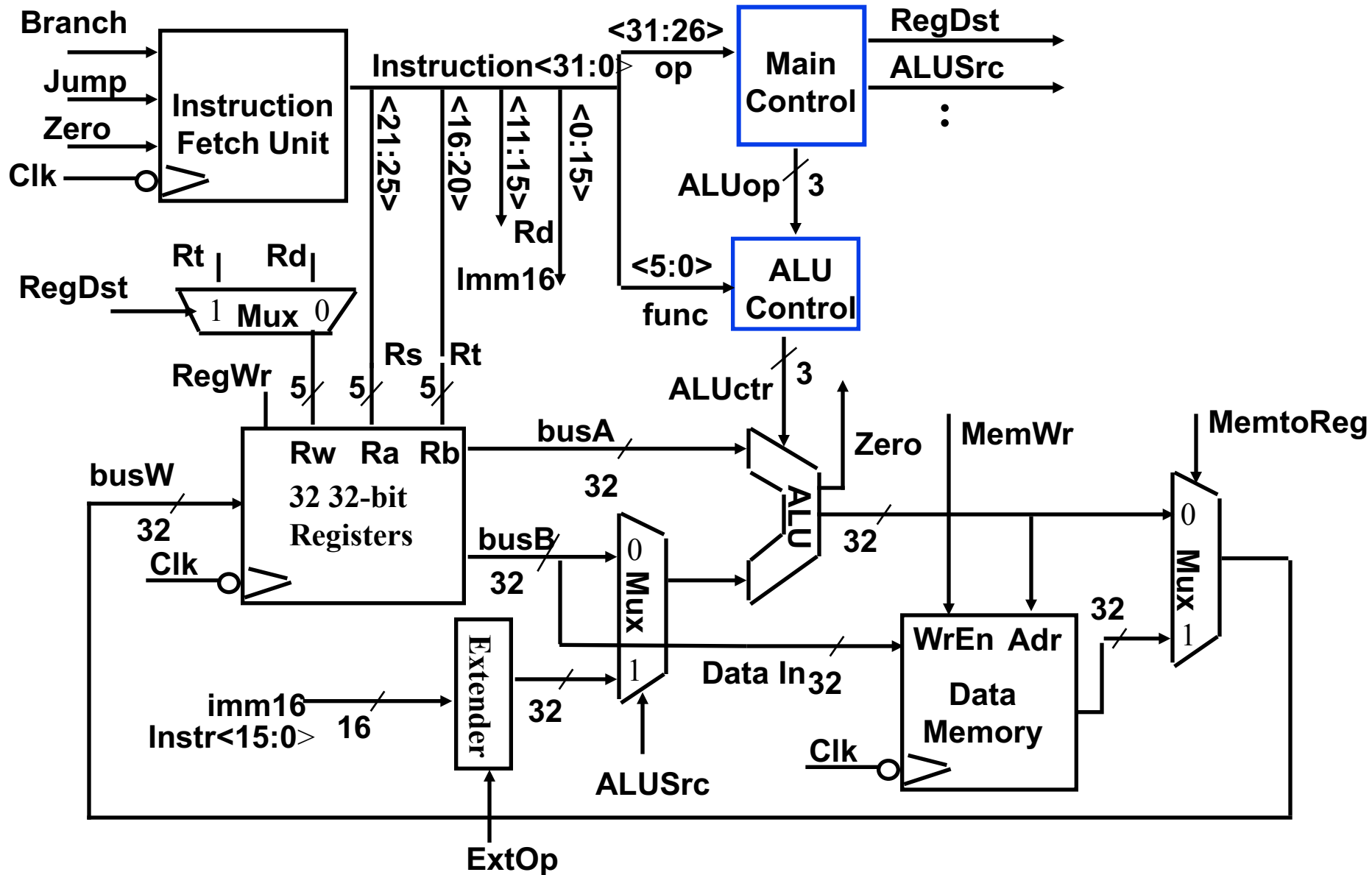
第三讲 高级流水线技术

第一讲 流水线数据通路和控制

主要内容

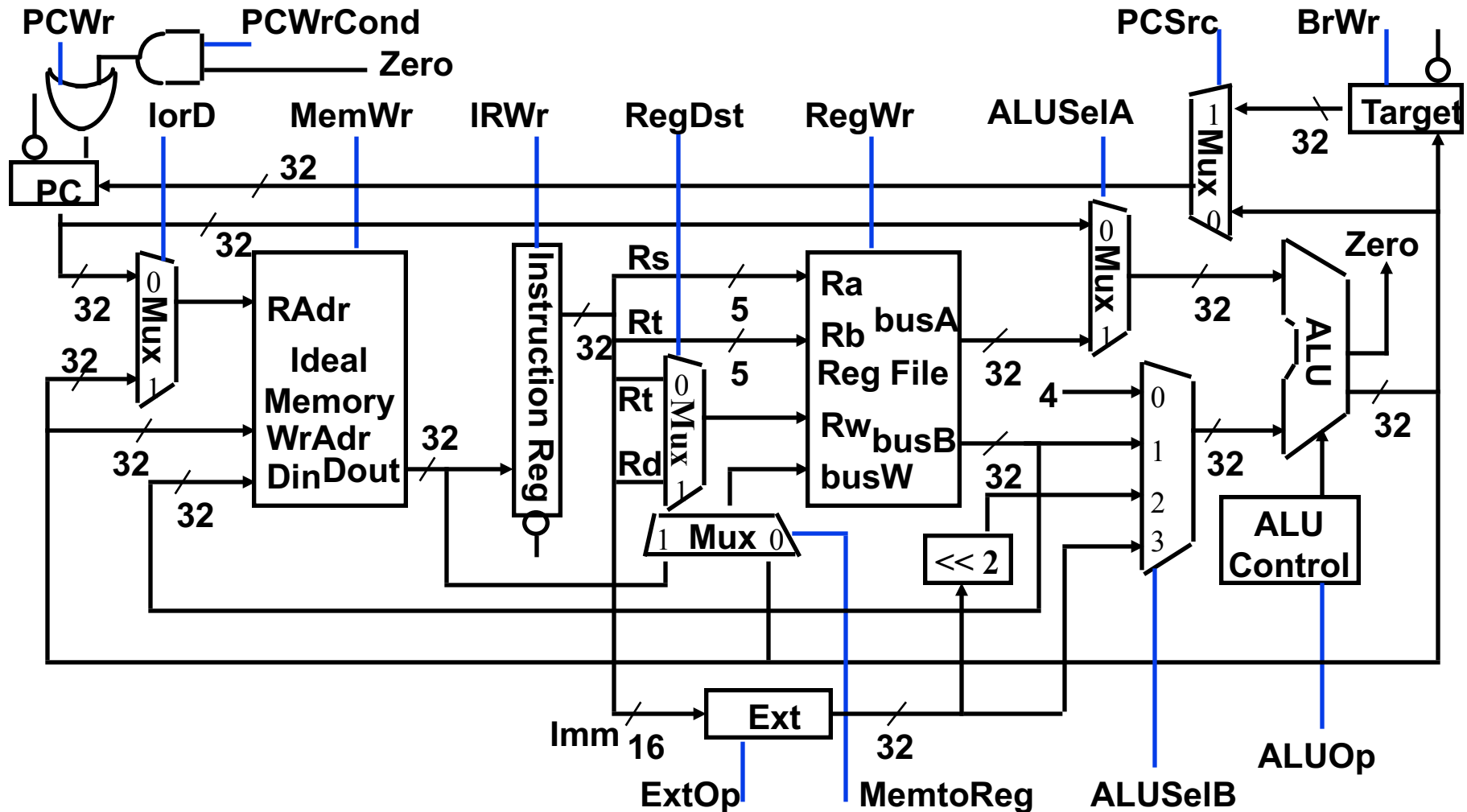
- 日常生活中的流水线处理例子：洗衣服
- 单周期处理器模型和流水线性能比较
- 什么样的指令集适合于流水线方式执行
- 如何设计流水线数据通路
 - 以MIPS指令子集来说明
- 如何设计流水线控制逻辑
 - 分析每条指令执行过程中的控制信号
 - 给出控制器设计过程
- 流水线冒险的概念

复习: A Single Cycle Processor

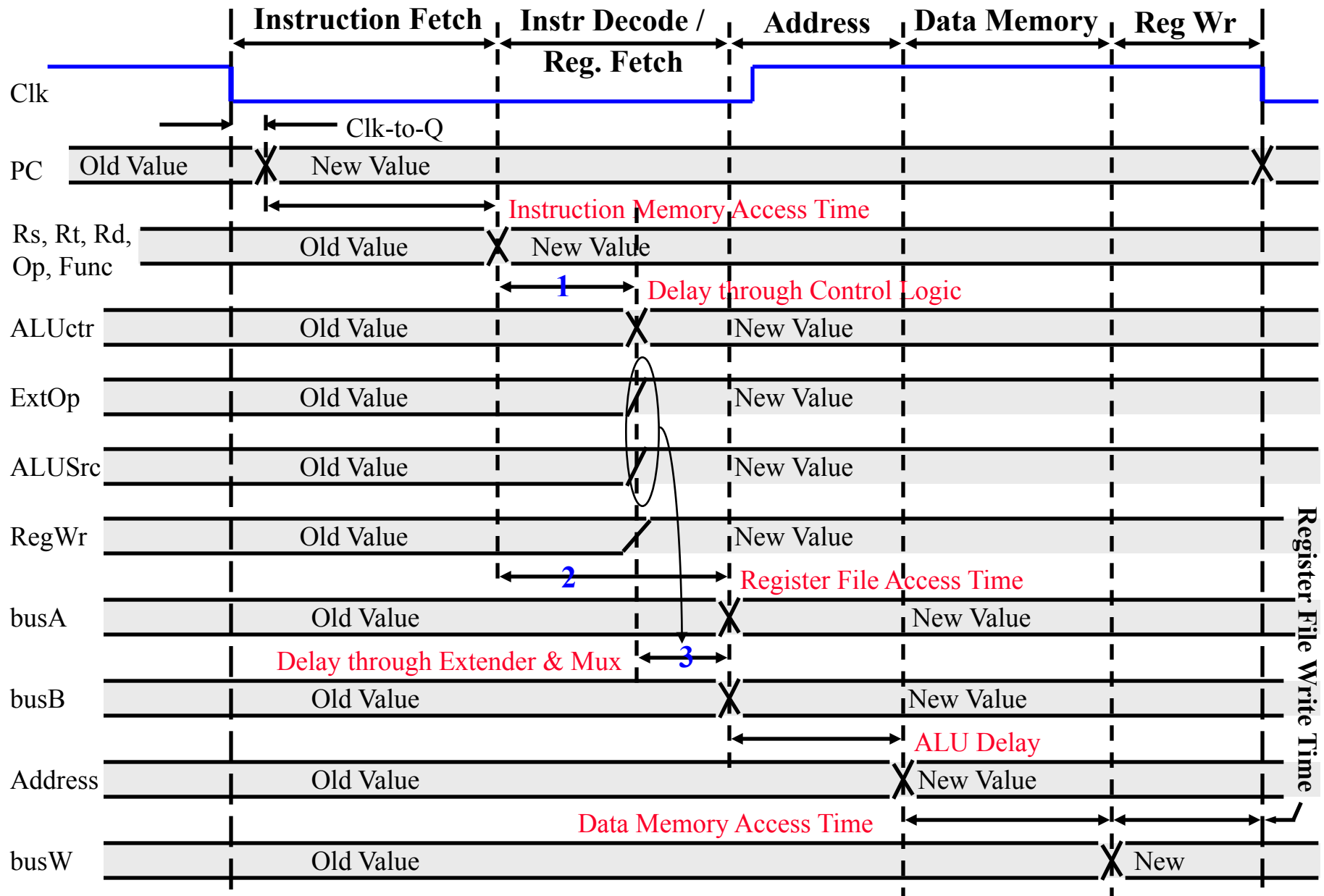


复习: Multiple Cycle Processor

- MCP: 一个功能部件在一个指令周期中可以被使用多次。



复习: Timing Diagram of a Load Instruction

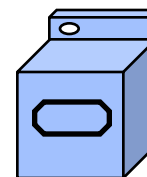
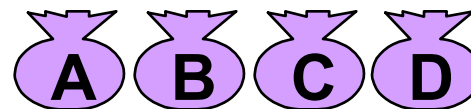


一个日常生活中的例子—洗衣服

◦ Laundry Example

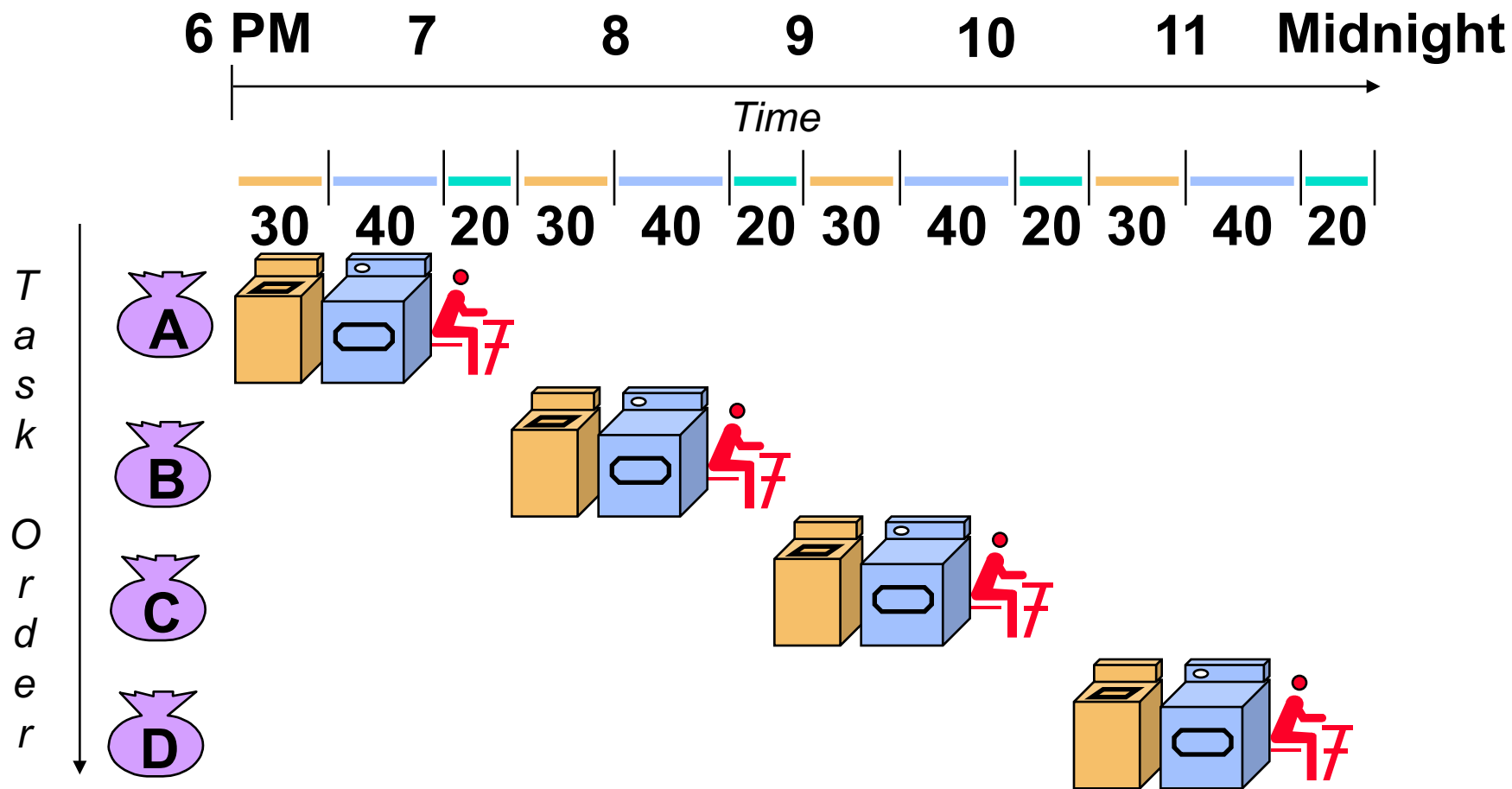
- Ann, Brian, Cathy, Dave each have one load of clothes to **wash, dry, and fold**
- Washer takes **30 minutes**
- Dryer takes **40 minutes**
- “Folder” takes **20 minutes**

如果让你来管理洗衣店，你会如何安排？



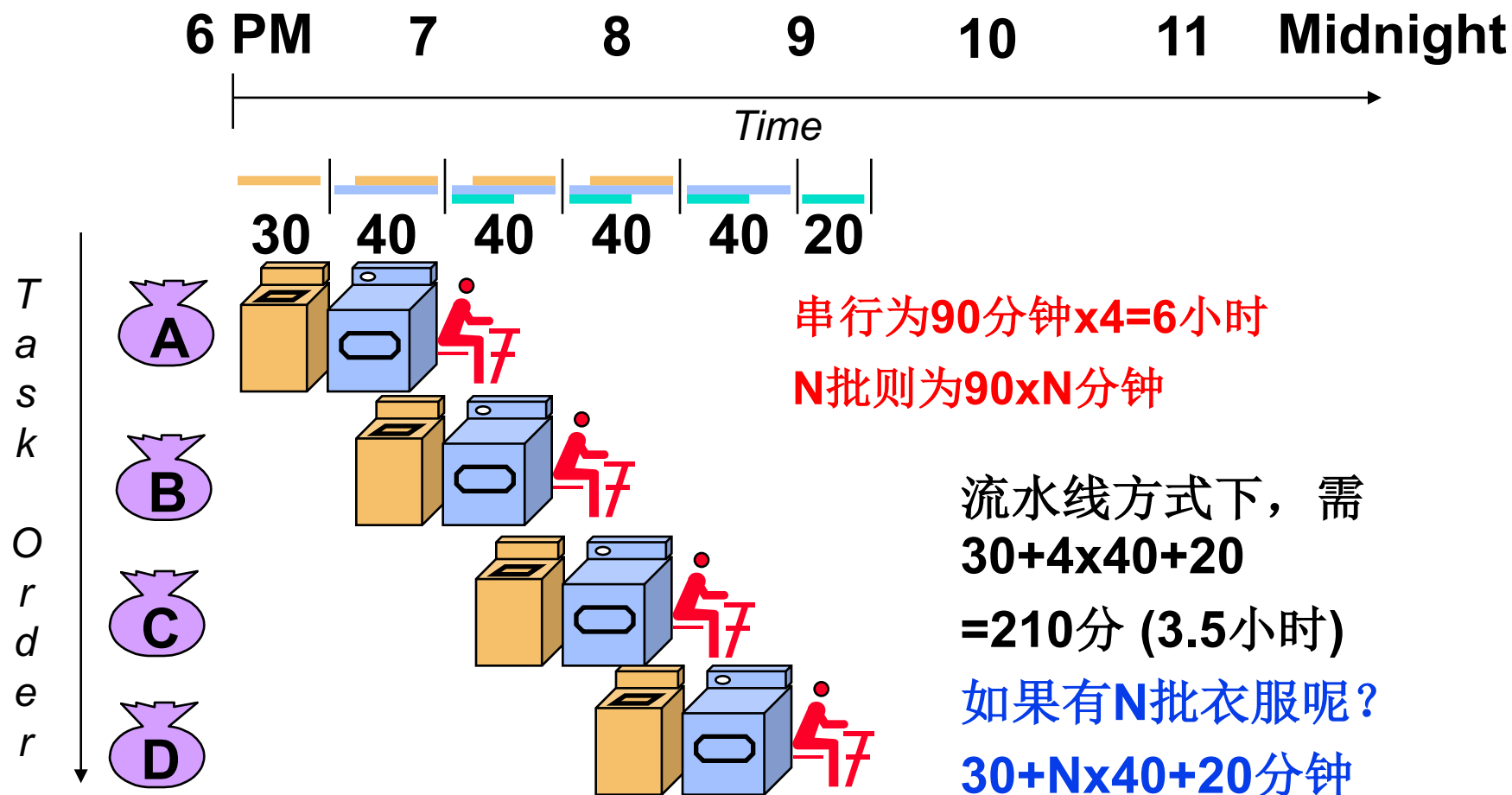
Pipelining: It's Natural !

Sequential Laundry (串行方式)



- 串行方式下，4批衣服需要花费6小时 ($4 \times (30 + 40 + 20) = 360$ 分钟)
- N批衣服，需花费的时间为 $N \times (30 + 40 + 20) = 90N$
- 如果用流水线方式洗衣服，则花多少时间呢？

Pipelined Laundry: (Start work ASAP)



流水方式下，所用时间主要与最长阶段的时间有关！

假定每一步时间均衡，则比串行方式提高约3倍！

复习：Load指令的5个阶段

阶段1	阶段2	阶段3	阶段4	阶段5
Ifetch	Reg/Dec	Exec	Mem	Wr

- **Ifetch (取指)**：取指令并计算PC+4 (用到哪些部件？)
指令存储器、Adder
- **Reg/Dec (取数和译码)**：取数同时译码 (用到哪些部件？)
寄存器堆读口、指令译码器
- **Exec (执行)**：计算内存单元地址 (用到哪些部件？)
扩展器、ALU
- **Mem (读存储器)**：从数据存储器中读 (用到哪些部件？)
数据存储器
- **Wr(写寄存器)**：将数据写到寄存器中 (用到哪些部件？)
寄存器堆写口

这里寄存器堆的读口和写口可看成两个不同的部件。

指令的执行过程是否和“洗衣”过程类似？是否可以采用类似方式来执行指令呢？

单周期指令模型与流水模型的性能比较

- 假定以下每步操作所花时间为：

- 取指：2ns
- 寄存器读：1ns
- ALU操作：2ns
- 存储器读：2ns
- 寄存器写：1ns

Load指令执行时间总计为：8ns
(假定控制单元、PC访问、信号传递等没有延迟)

- 单周期模型

- 每条指令在一个时钟周期内完成
- 时钟周期等于最长的lw指令的执行时间，即：8ns
- 串行执行时，N条指令的执行时间为：8Nns

- 流水线性能

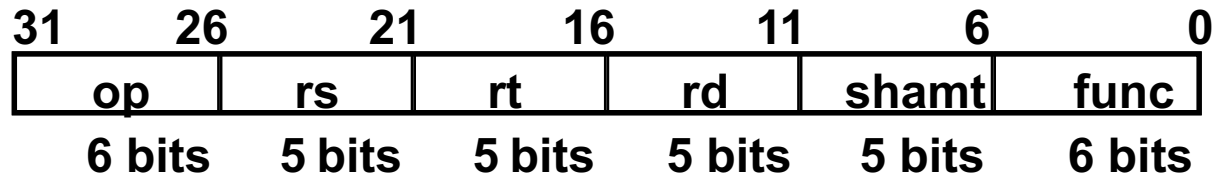
- 时钟周期等于最长阶段所花时间为：2ns
- 每条指令的执行时间为：2nsx5=10ns
- N条指令的执行时间为：(4+N)x2ns
- 在N很大时，比串行方式提高约 4 倍
- 若各阶段操作均衡(例如，各阶段都是2ns)，则提高倍数为5倍。

流水线方式下，单条指令执行时间不能缩短，但能大大提高指令吞吐率！

流水线指令集的设计

○ 具有什么特征的指令集有利于流水线执行呢？

- 长度尽量一致，有利于简化取指令和指令译码操作
 - MIPS指令32位，下址计算方便: $PC+4$
 - X86指令从1字节到17字节不等，使取指部件极其复杂
- 格式少，且源寄存器位置相同，有利于在指令未知时就可取操作数
 - MIPS指令的rs和rt位置一定，在指令译码时就可读rs和rt的值

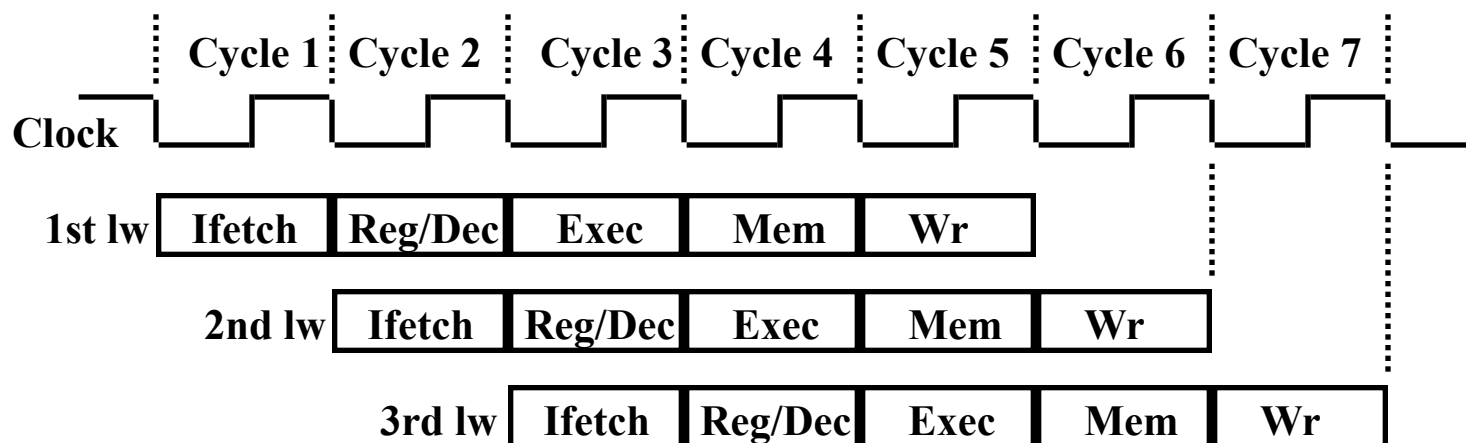


若位置随指令不同而不同，则需先确定指令类型才能取寄存器编号

- load / Store指令才能访问存储器，有利于减少操作步骤，规整流水线
 - lw/sw指令的地址计算和运算指令的执行步骤规整在同一个周期
 - X86运算类指令操作数可为内存数据，需计算地址、访存、执行
- 内存中“对齐”存放，有利于减少访存次数和流水线的规整

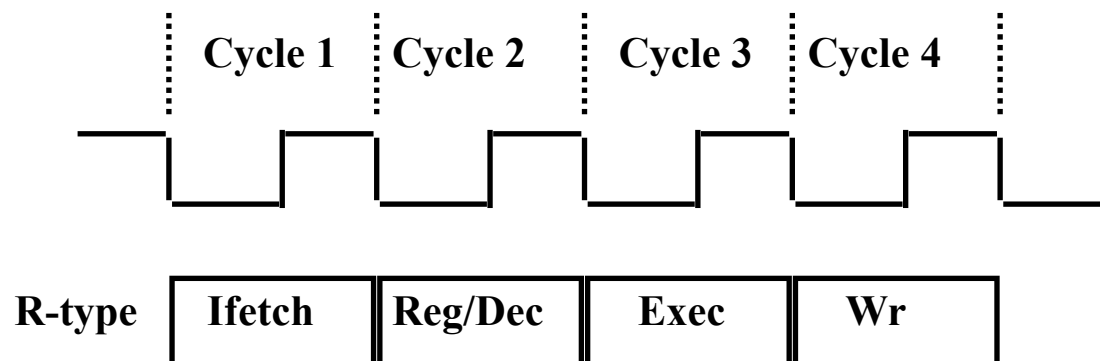
总之，规整、简单和一致等特性有利于指令的流水线执行

Load指令的流水线



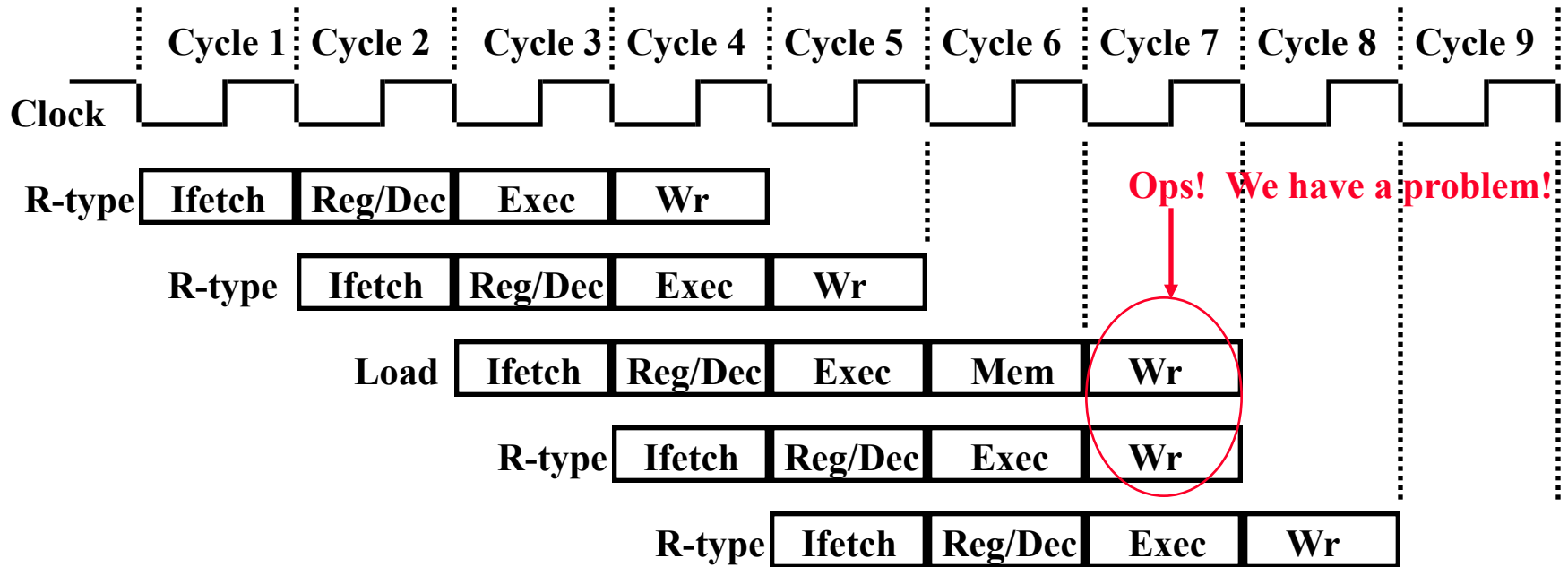
- 每个周期有五个功能部件同时在工作
- 后面指令在前面完成取指后马上开始
- 每个load指令仍然需要五个周期完成
- 但是，吞吐率(throughput)提高许多，理想情况下，有：
 - 每个周期有一条指令进入流水线
 - 每个周期都有一条指令完成
 - 每条指令的有效周期(CPI)为1

R-type指令的4个阶段



- **Ifetch:** 取指令并计算PC+4
- **Reg/Dec:** 从寄存器取数，同时指令在译码器进行译码
- **Exec:** 在ALU中对操作数进行计算
- **Wr:** ALU计算的结果写到寄存器

含R-type和 Load 指令的流水线



上述流水线有个问题：两条指令试图同时写寄存器，因为

- Load在第5阶段用寄存器写口

- R-type在第4 阶段用寄存器写口

或称为资源冲突！

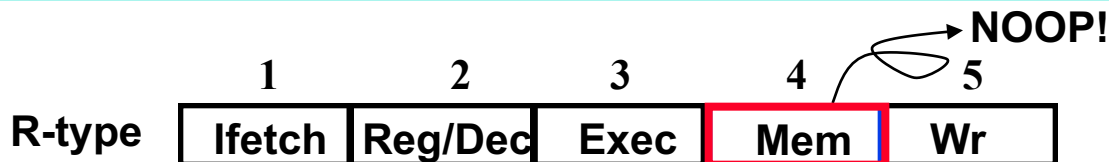
把一个功能部件同时被多条指令使用的现象称为结构冒险(Structure Hazard)

为了流水线能顺利工作，规定：

- 每个功能部件每条指令只能用一次（如：写口不能用两次或以上）

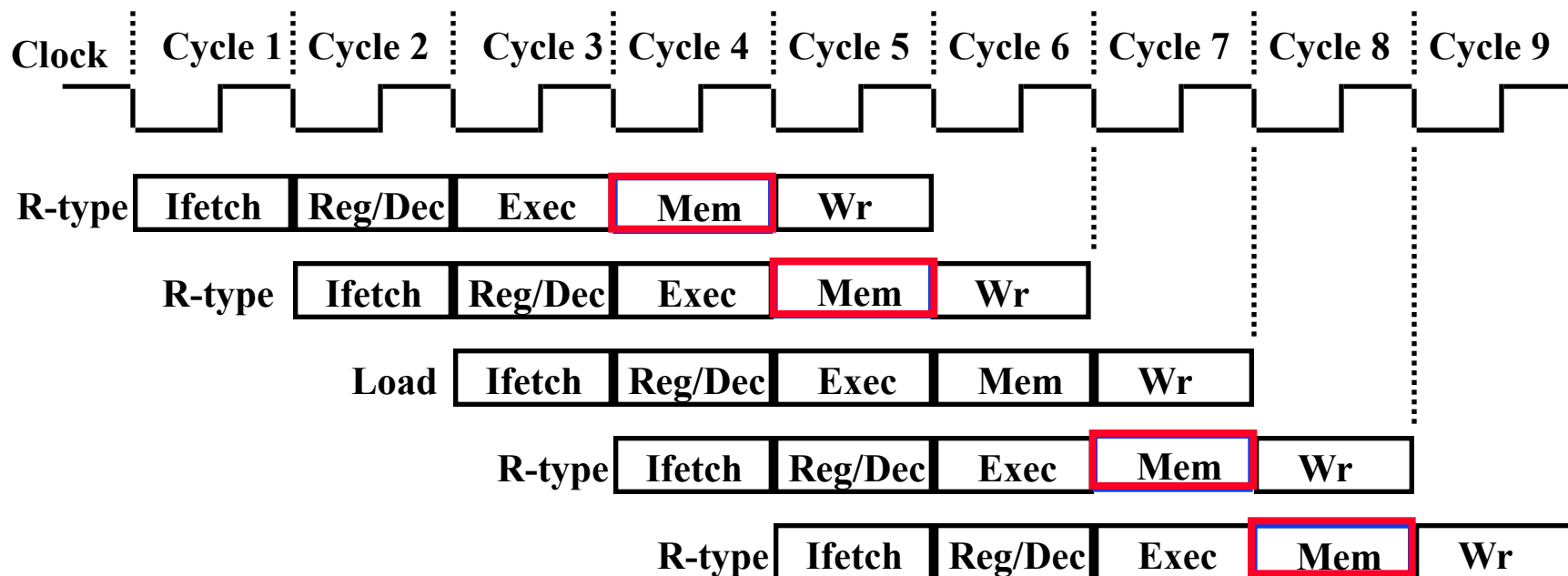
- 每个功能部件必须在相同的阶段被使用（如：写口总是在第五阶段被使用）

R-type的Wr操作延后一个周期执行



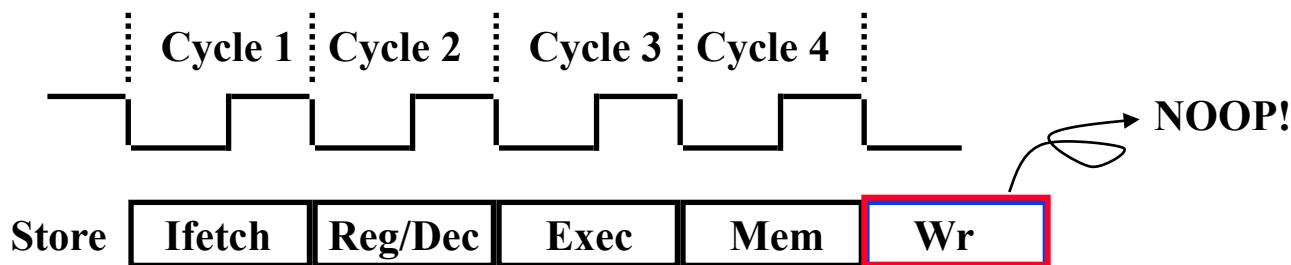
◦ 加一个NOP阶段以延迟“写”操作:

- 把“写”操作安排在第5阶段, 这样使R-Type的Mem阶段为空NOP



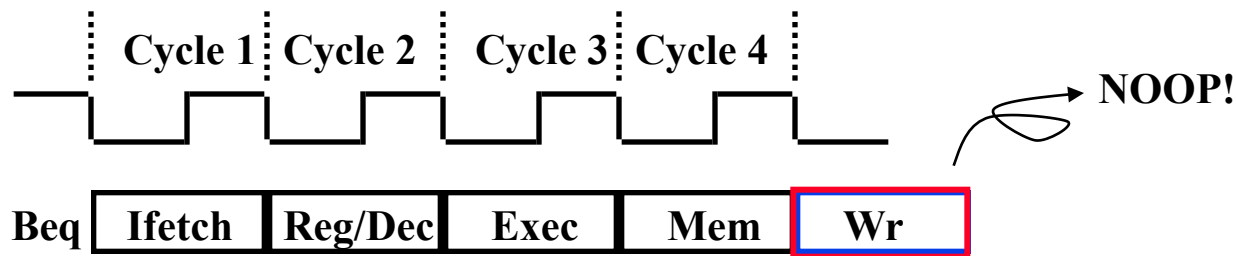
这样使流水线中的每条指令都有相同多个阶段!

Store指令的四个阶段



- **Ifetch:** 取指令并计算PC+4
- **Reg/Dec:** 从寄存器取数，同时指令在译码器进行译码
- **Exec:** 16位立即数符号扩展后与寄存器值相加，计算主存地址
- **Mem:** 将寄存器读出的数据写到主存
- **Wr:** 加一个空的写阶段，使流水线更规整！

Beq的四个阶段



◦ **Ifetch:** 取指令并计算PC+4

◦ **Reg/Dec:** 从寄存器取数，同时指令在译码器进行译码

◦ **Exec:** 执行阶段

- ALU中比较两个寄存器的大小（做减法）
- Adder中计算转移地址

◦ **Mem:** 如果比较相等，则：

- 转移目标地址写到PC

◦ **Wr:** 加一个空写阶段，使流水线更规整！

按照上述方式，把所有指令都按照最复杂的“load”指令所需的五个阶段来划分，不需要的阶段加一个“NOP”操作

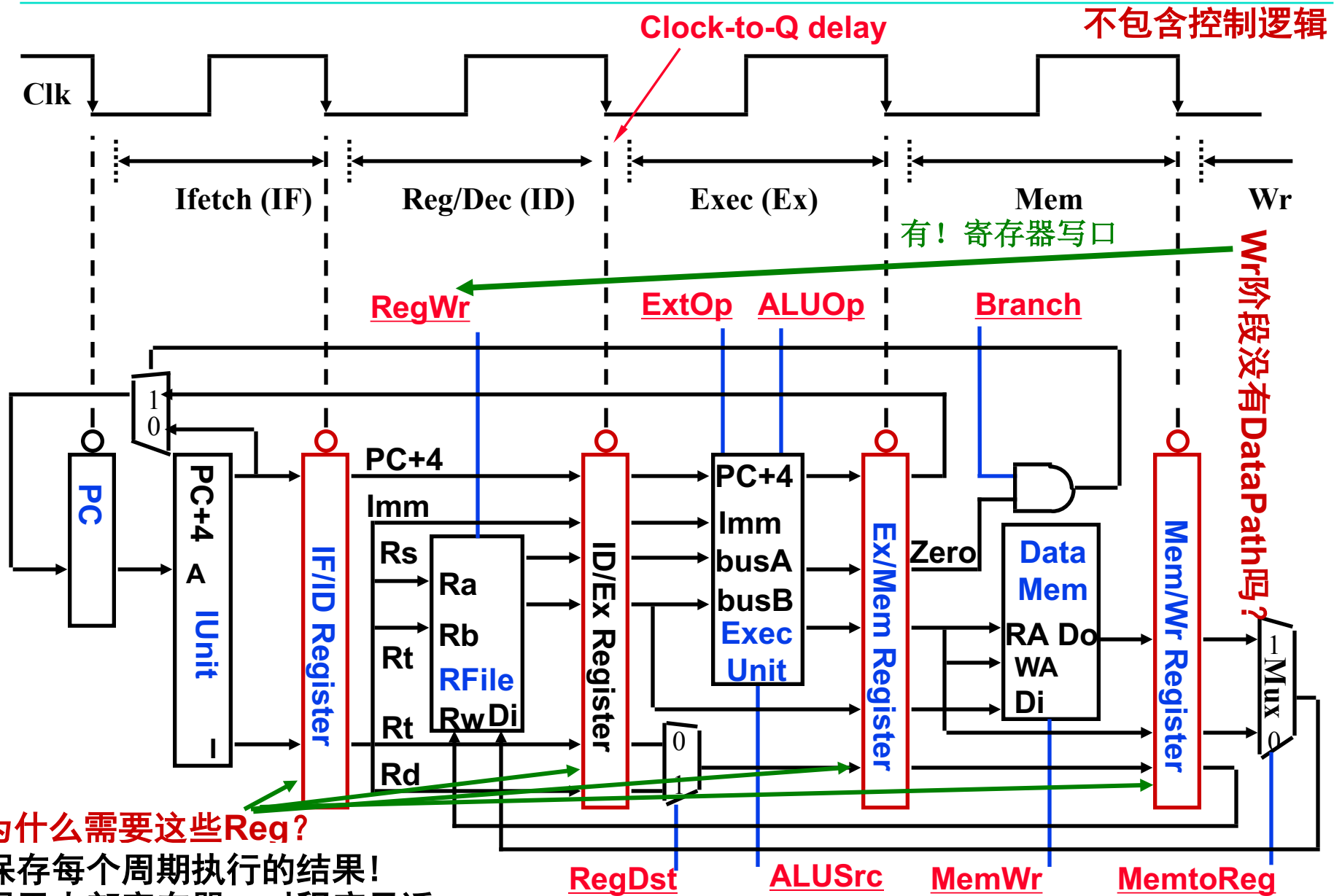
与多周期通路有什么不同？

多周期通路中，在Reg/Dec阶段投机进行了转移地址的计算！可以减少Branch指令的时钟数

为什么流水线中不进行“投机”计算？

因为流水线中所有指令的执行阶段一样多，Branch指令无需节省时钟，因为有比它更复杂的指令。

A Pipelined Datapath (五阶段流水线数据通路)



为什么需要这些Reg?

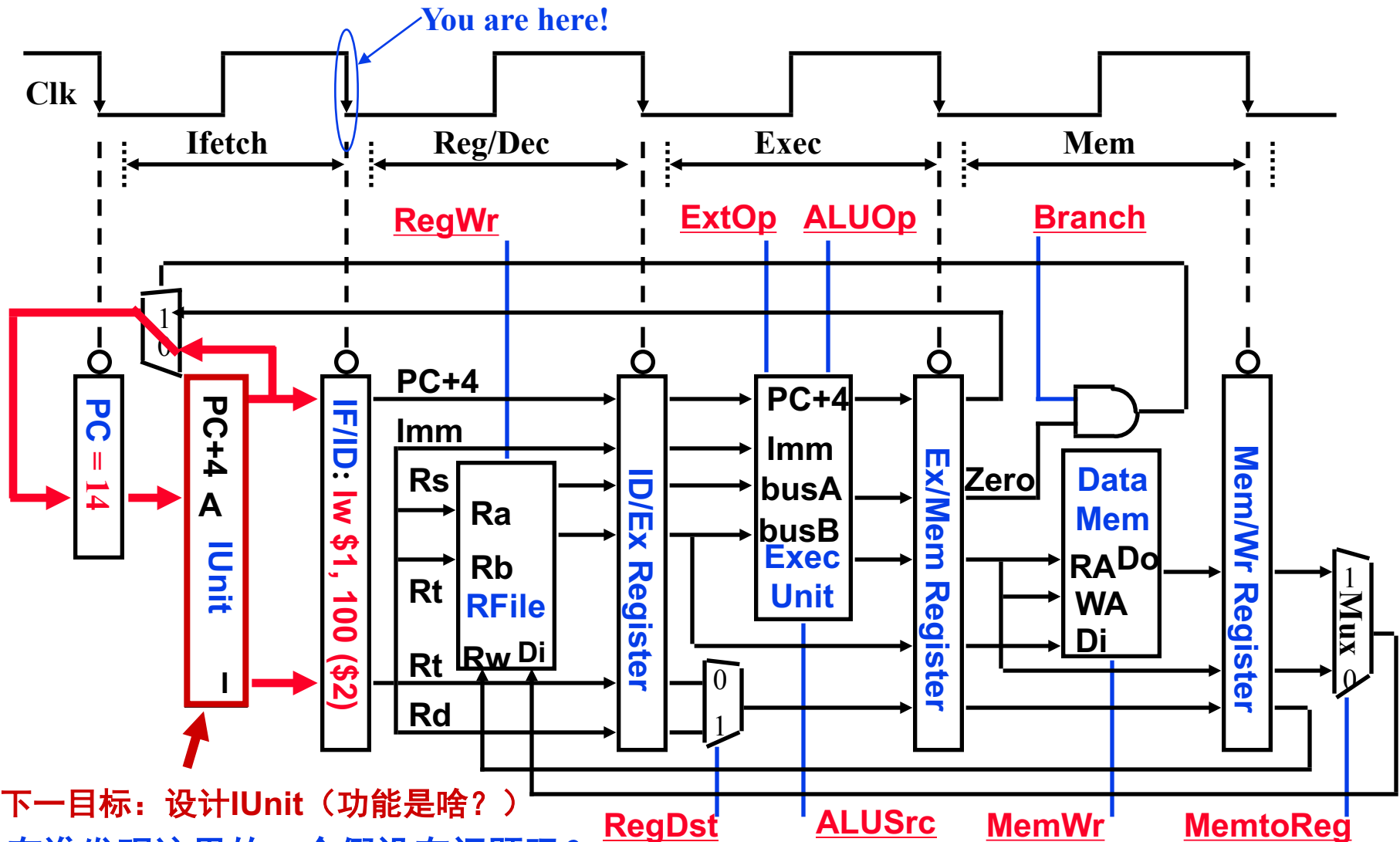
保存每个周期执行的结果!
属于内部寄存器, 对程序员透明, 不需作为现场保存

下面看每条指令在流水线通路中的执行过程

取指令 (Ifetch) 阶段

◦ 第10单元指令: `lw $1, 0x100($2)`

功能: $\$1 \leftarrow \text{Mem} [(\$2) + 0x100]$



下一目标: 设计IUnit (功能是啥?)

有谁发现这里的一个假设有问题吗?

MIPS指令的地址可能是10吗?

先猜一下IUnit中有哪些功能部件?

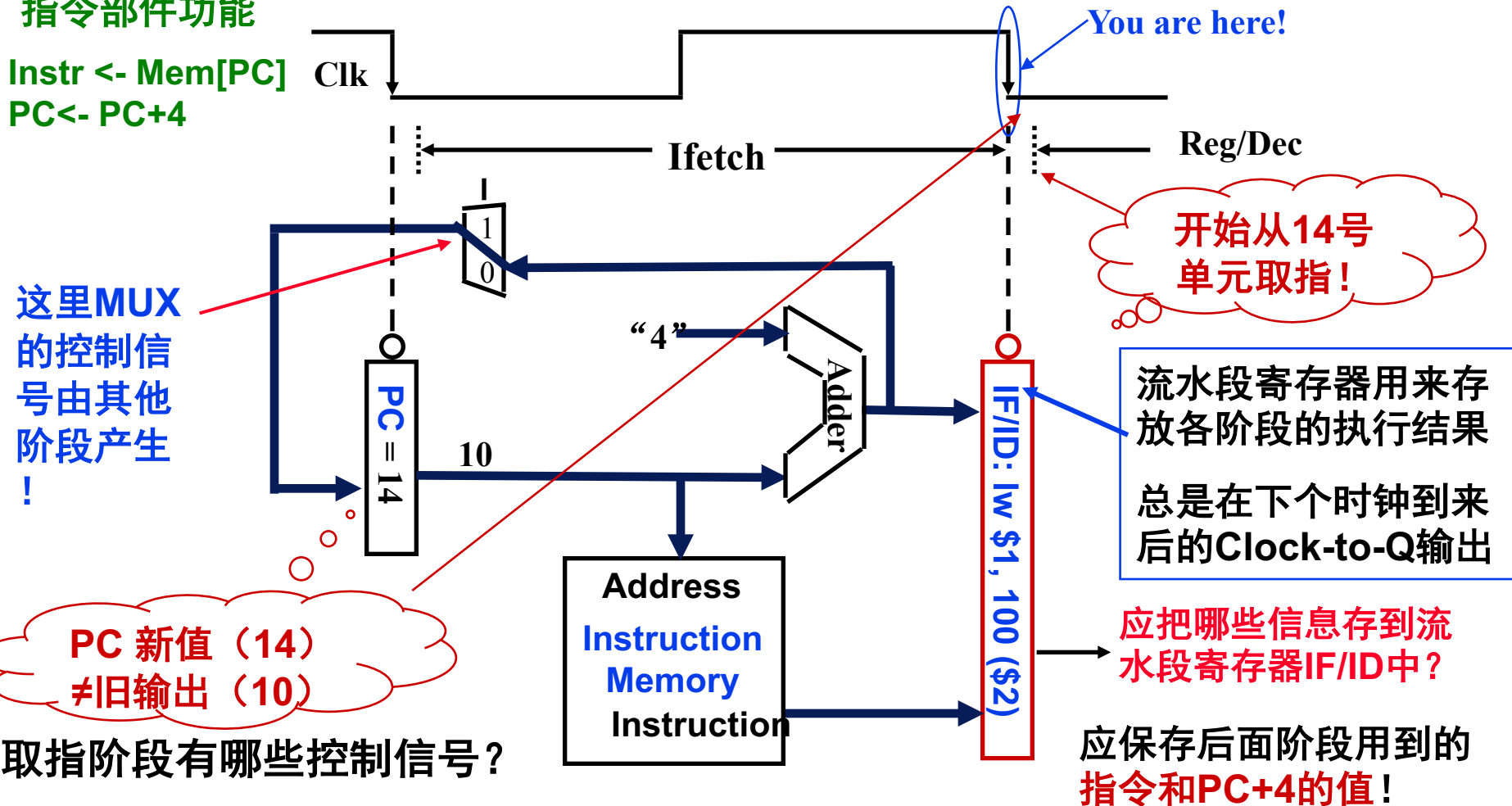
指令部件 IUnit的设计

◦ 第10单元指令: `lw $1, 0x100($2)`

随后的指令在14号单元中!

指令部件功能

$\text{Instr} \leftarrow \text{Mem}[\text{PC}]$
 $\text{PC} \leftarrow \text{PC} + 4$

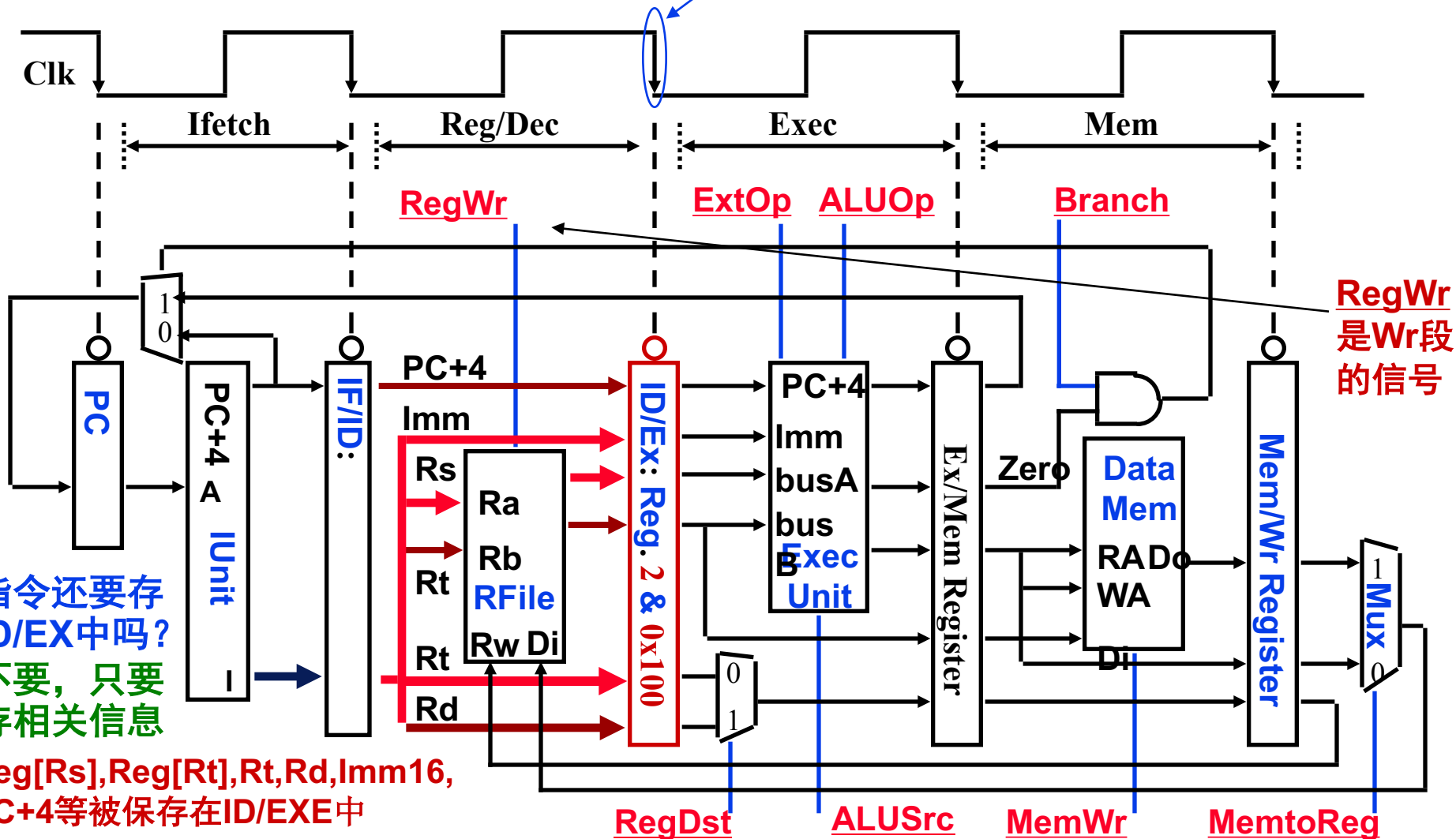


译码/取数 (Reg/Dec) 阶段

Location 10: lw \$1, 0x100(\$2)

功能: $\$1 \leftarrow \text{Mem}[(\$2) + 0x100]$

You are here!



指令还要存ID/EX中吗?
不要, 只要存相关信息

Reg[Rs], Reg[Rt], Rt, Rd, Imm16, PC+4等被保存在ID/EXE中

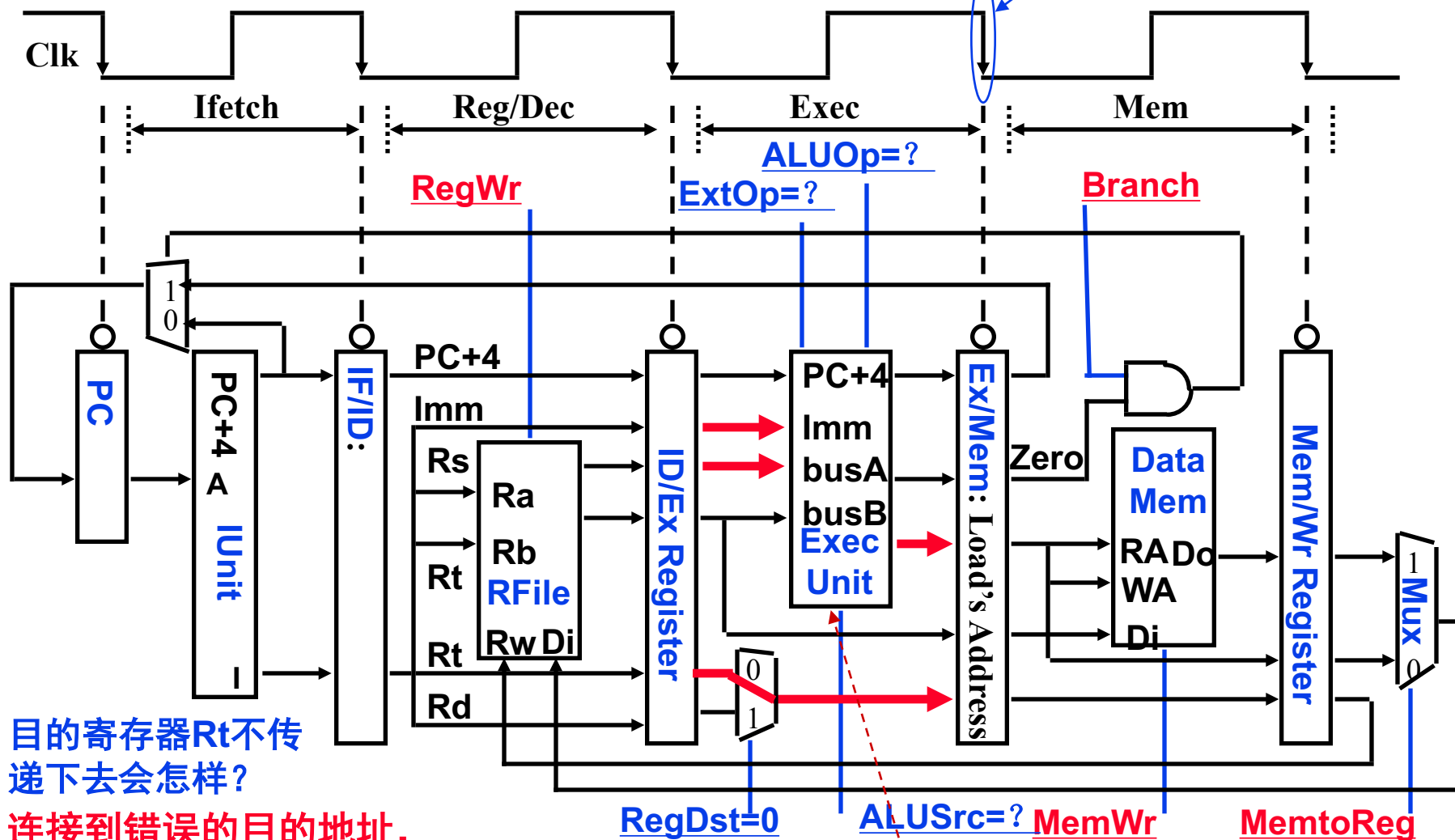
该阶段有哪些控制信号? 没有! 因是所有指令的公共操作, 故无控制信号!

Load指令的地址计算阶段

- Location 10: lw \$1, 0x100(\$2) 功能: $\$1 \leftarrow \text{Mem}[(\$2) + 0x100]$

指令已被译码，可确定执行部件的控制信号！

✓ You are here!



目的寄存器Rt不传递下去会怎样？

连接到错误的目的地址，
指令执行错误！

下一目标：设计执行部件(Exec Unit)

猜有哪些部件？

执行部件（Exec Unit）的设计

执行部件功能？

- 计算内存地址
- 计算转移目标地址
- 一般ALU运算

Load指令的各控制信号取值？

RegDes=0, ALUSrc=1
ALUOp=addu, ExtOp=1

Store指令呢？

RegDes=x, ALUSrc=1
ALUOp=addu, ExtOp=1

Branch指令呢？

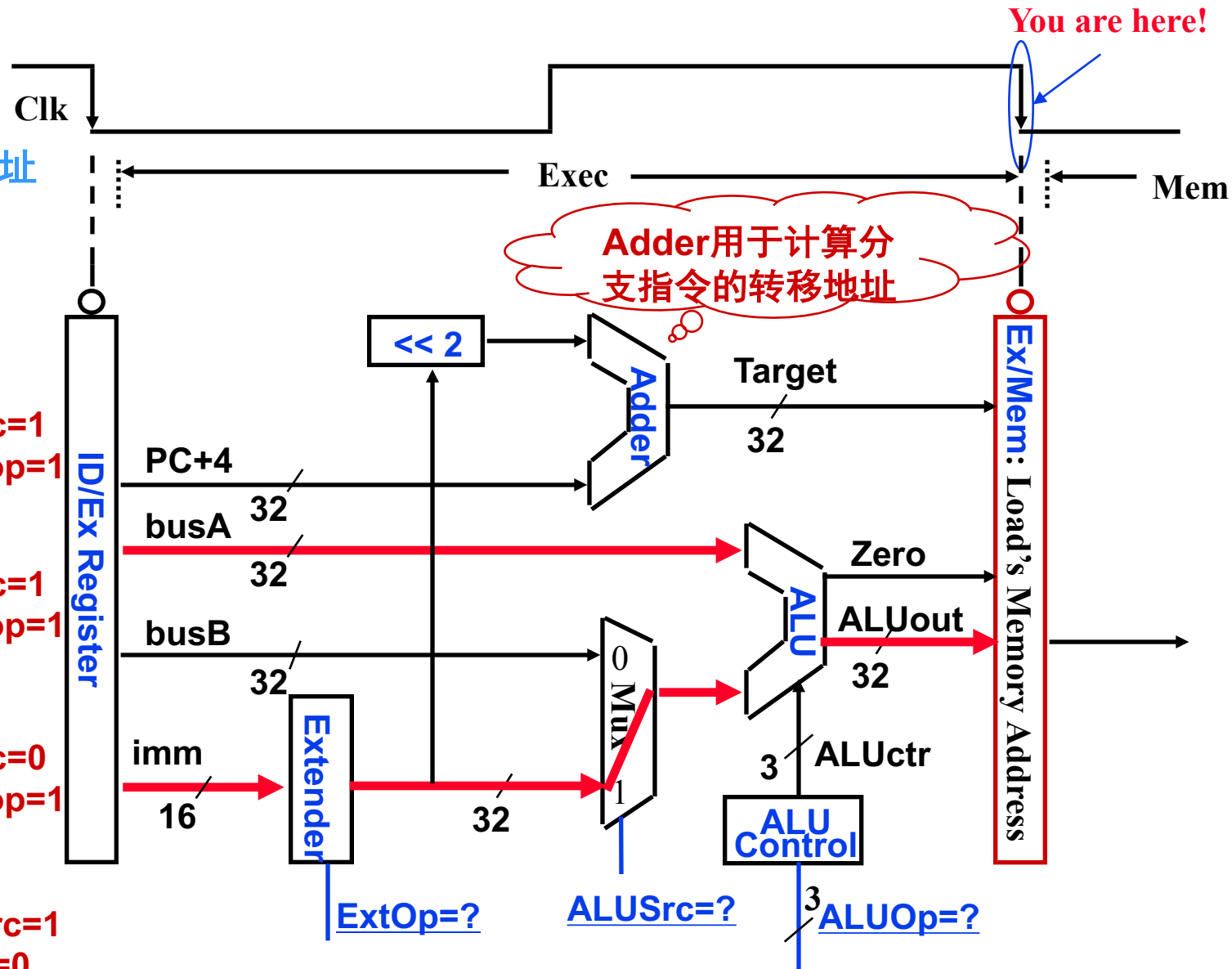
RegDes=x, ALUSrc=0
ALUOp=subu, ExtOp=1

Ori指令呢？

RegDes=0, ALUSrc=1
ALUOp=or, Extop=0

R型指令呢？

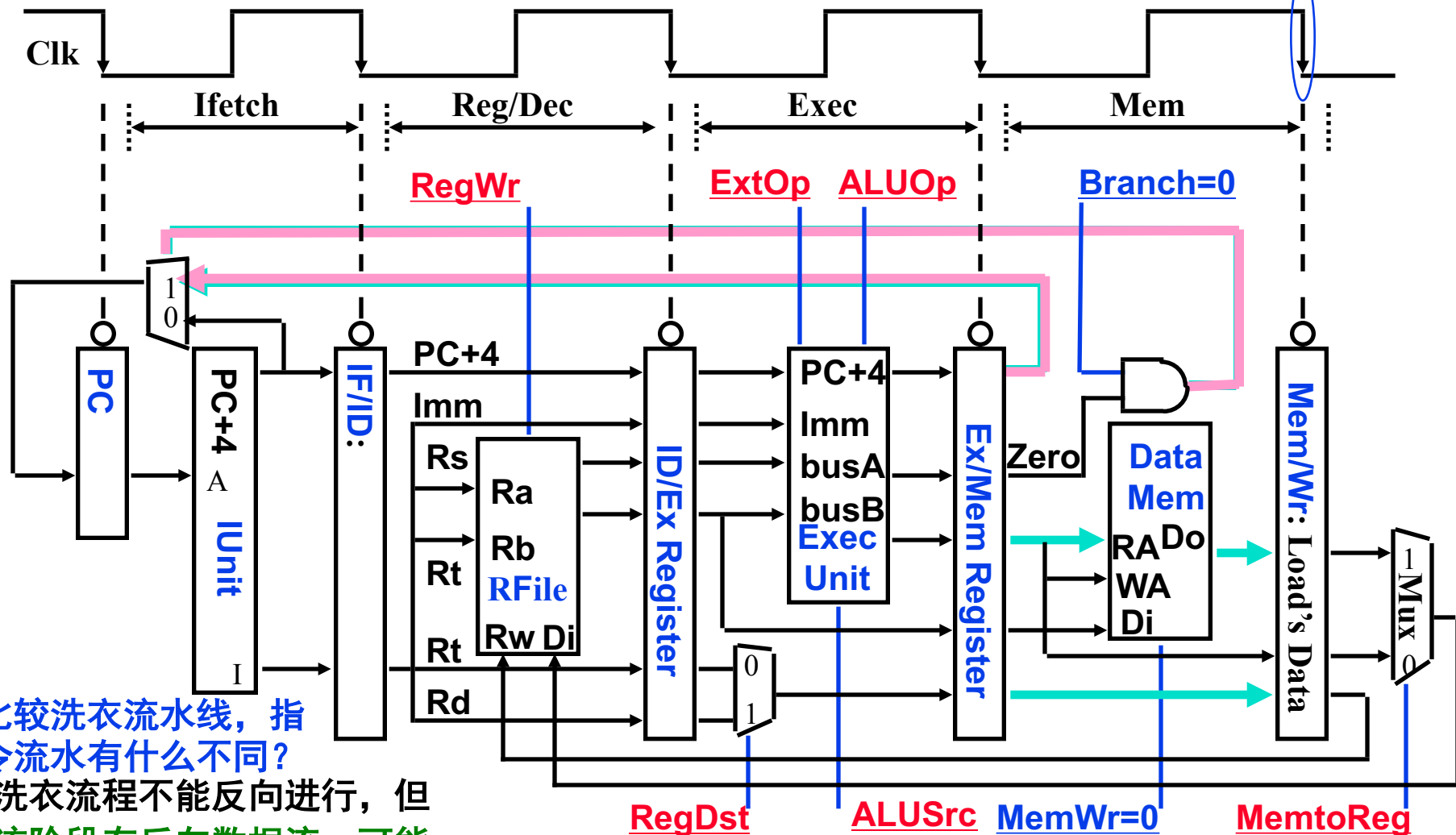
RegDes=1, ALUSrc=0
ALUop='func', Extop=x



Load指令的存储器读(Mem)周期

◦ **Location 10:** lw \$1, 0x100(\$2) 功能: $\$1 \leftarrow \text{Mem}[(\$2) + 0x100]$

You are here!



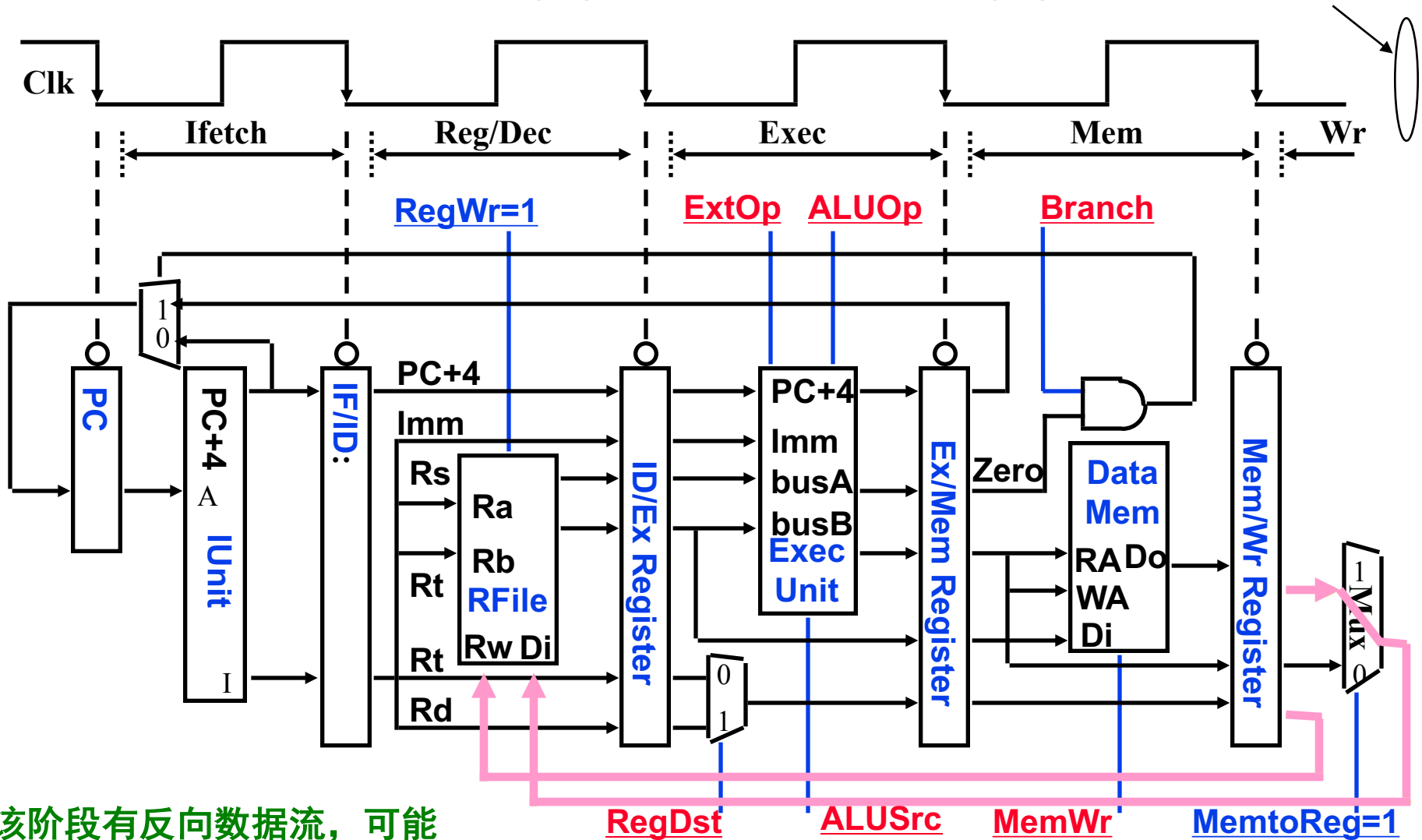
比较洗衣流水线，指令流水有什么不同？

洗衣流程不能反向进行，但该阶段有反向数据流，可能会引起冒险！以后介绍。

周期以最长操作为准设计 $\text{Cycle} > T_{\text{read}}$

Load指令的回写 (Write Back) 阶段

◦ **Location 10:** lw \$1, 0x100(\$2) **功能:** $\$1 \leftarrow \text{Mem}[(\$2) + 0x100]$



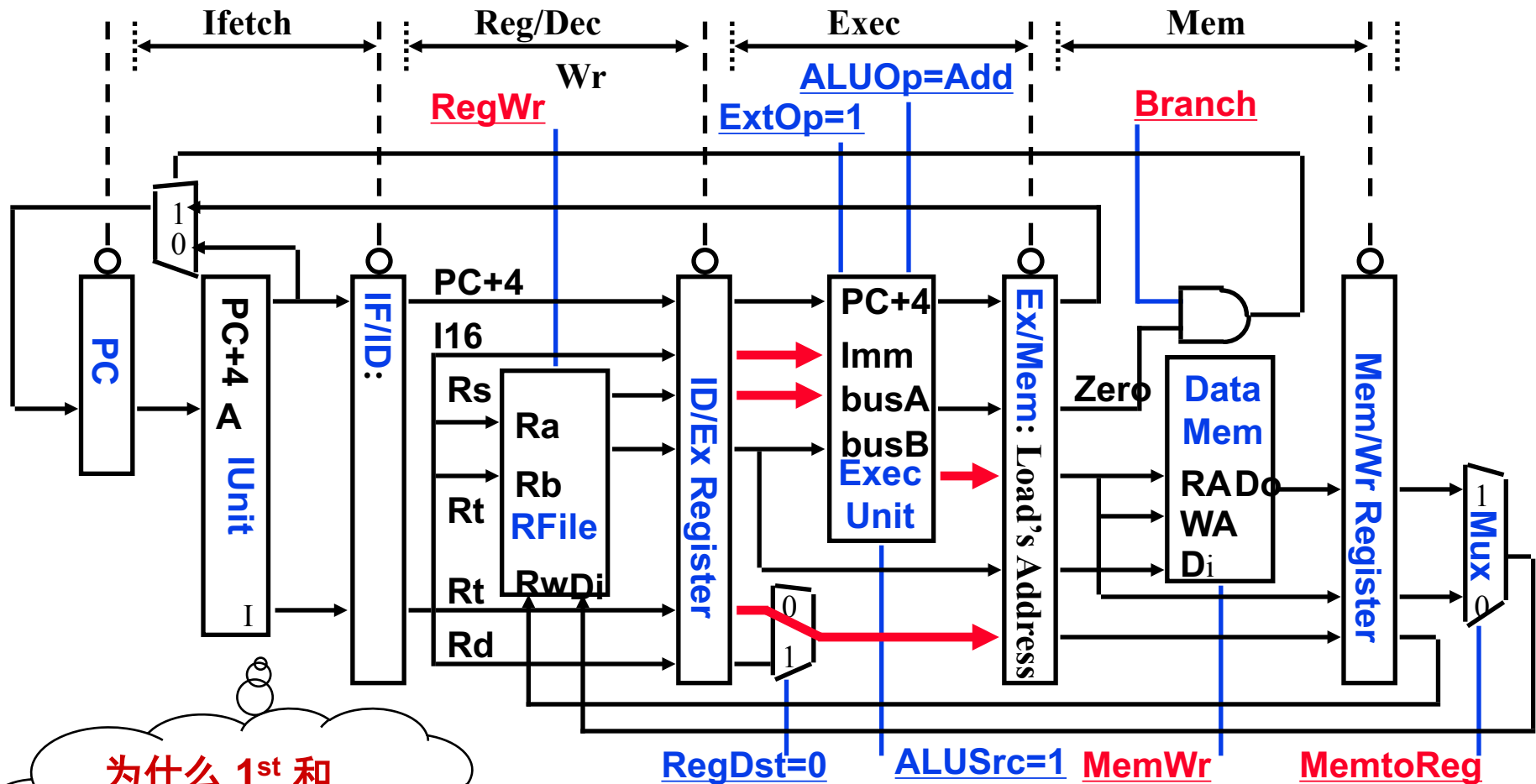
该阶段有反向数据流，可能会引起冒险！以后介绍。

各阶段所经DataPath已有，控制信号如何得到？

流水线中的Control Signals如何获得?

◦ 主要考察: 第N阶段的控制信号, 它取决于某条指令的某个阶段。

- $N = \text{Exec, Mem, Wr}$ (只有这三个阶段有控制信号)
- 例: Load的Exec段的控制信号 = Func (Load's Exec)



为什么 1st 和 2nd 阶段没有控制信号?

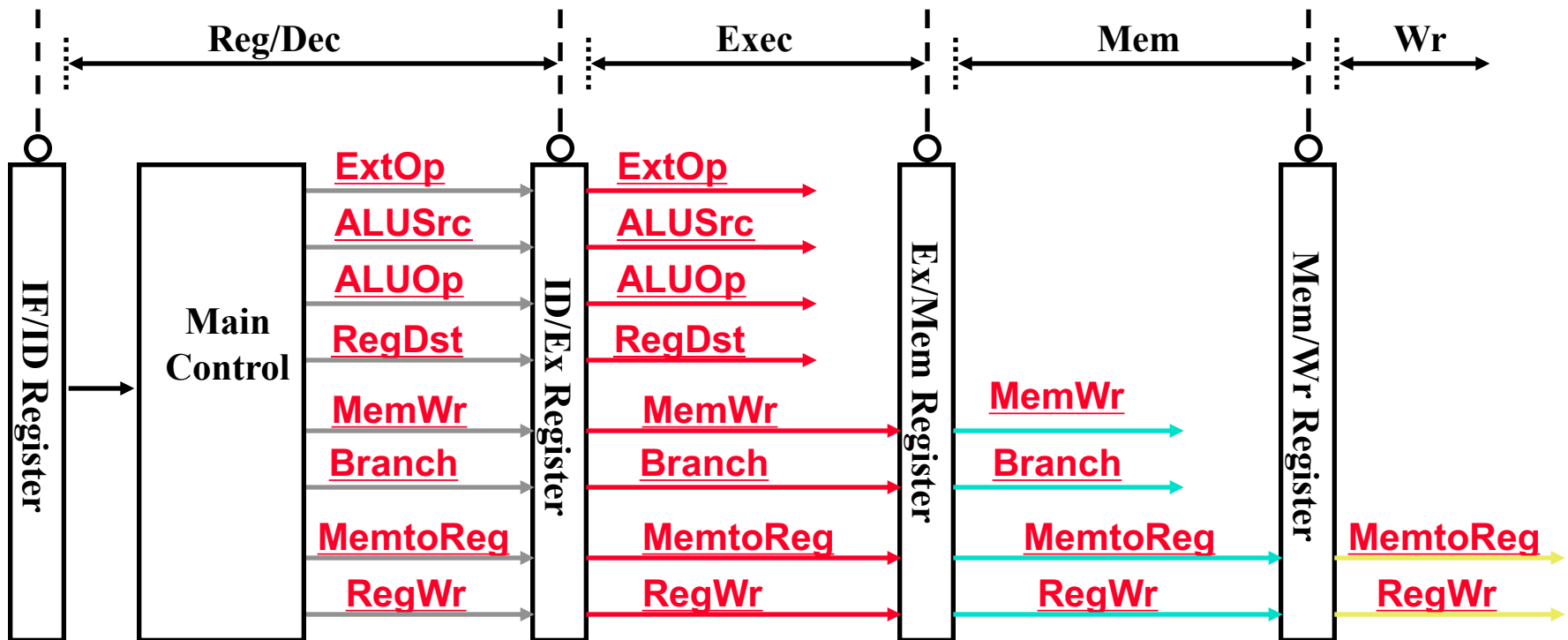
IF和ID阶段的功能对每条指令来说都一样

Load指令:流水线中的控制信号

◦ 在取数/译码 (Reg/Dec) 阶段产生本指令每个阶段的所有控制信号

- Exec信号 (ExtOp, ALUSrc, ...) 在1个周期后使用
- Mem信号 (MemWr, Branch) 在2个周期后使用
- Wr信号 (MemtoReg, RegWr) 在3个周期后使用

所以，控制信号也要保存在流水段寄存器中！



各流水段部件在一个时钟内完成某条指令的某个阶段的工作！

在下个时钟到达时，把执行结果以及前面传递来的后面各阶段要用到的所有数据（如：指令、立即数、目的寄存器等）和控制信号保存到流水线寄存器中！

流水线中的Control Signals

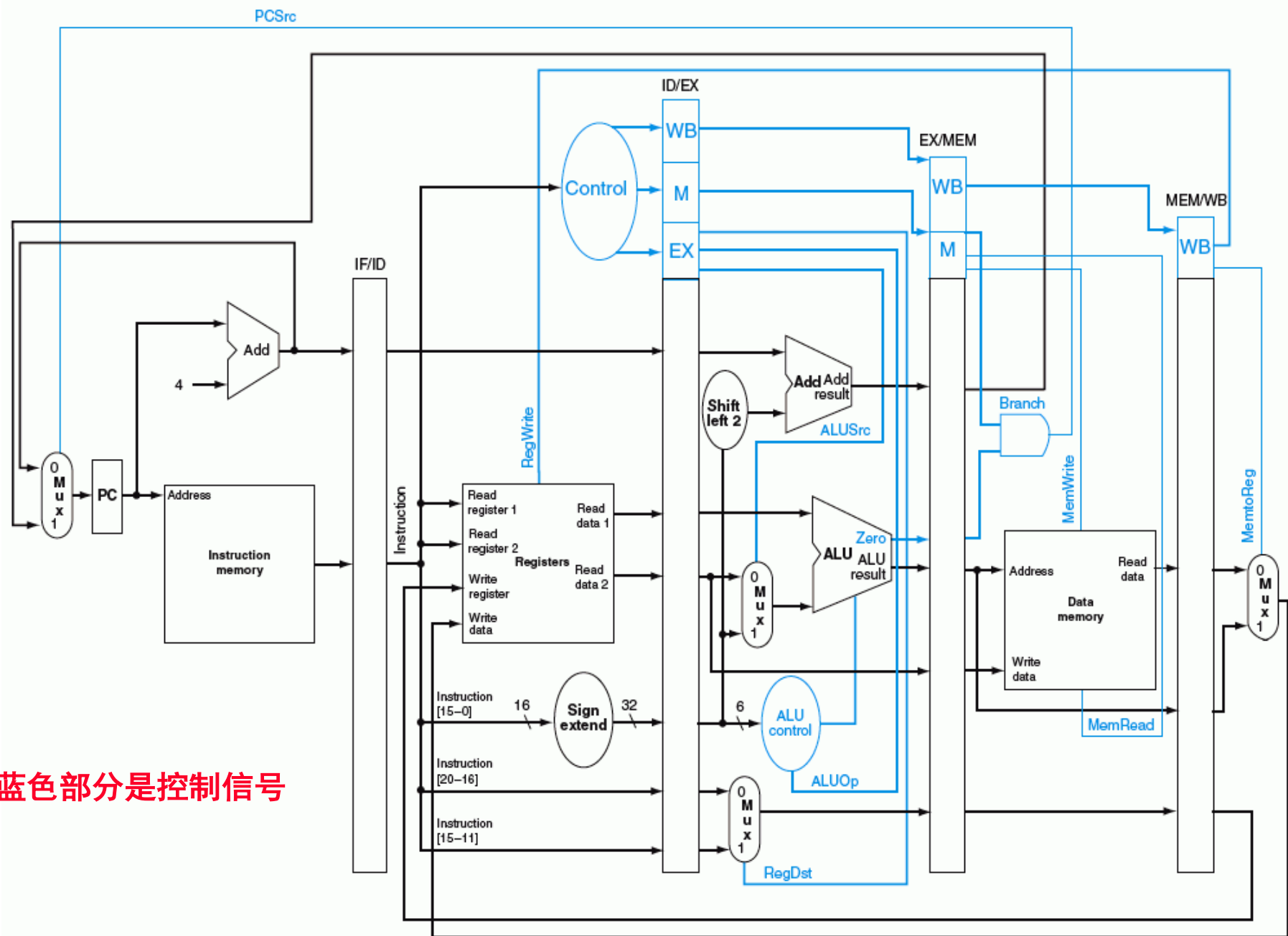
- 通过对前面流水线数据通路的分析，得知：
 - PC需要写使能吗？ 每个时钟都会改变PC，故不需要!
 - 流水段寄存器需要写使能吗？ 每个时钟都会改变流水段寄存器，故不需要!
 - Ifecth阶段和Dec/Reg阶段都没有控制信号
 - Exec阶段的控制信号有四个
 - ExtOp (扩展器操作): 1- 符号扩展; 0- 零扩展
 - ALUSrc (ALU的B口来源): 1- 来源于扩展器; 0- 来源于BusB
 - ALUOp (主控制器输出, 用于辅助局部ALU控制逻辑来决定ALUCtrl)
 - RegDst (指定目的寄存器): 1- Rd; 0- Rt
 - Mem阶段的控制信号有两个
 - MemWr (DM的写信号): Store指令时为1, 其他指令为0
 - Branch (是否为分支指令): 分支指令时为1, 其他指令为0
 - Wr阶段的控制信号有两个
 - MemtoReg (寄存器的写入源): 1- DM输出; 0- ALU输出
 - RegWr (寄存器堆写信号): 结果写寄存器的指令都为1, 其他指令为0

控制逻辑 (Control) 的设计

◦ 流水线控制逻辑的设计

- 每条指令的控制信号在该指令执行期间变不变? **不变!**
(谁记得单周期和多周期时各是怎样的情况?)
- 与单周期还是多周期的控制逻辑设计类似? **单周期!**
(谁记得单周期和多周期控制逻辑各是怎样设计的?)
- 设计过程
 - **控制逻辑分成两部分**
 - 主控制逻辑: 生成ALUop和其他控制信号
 - 局部ALU控制逻辑: 根据ALUop和func字段生成ALUCtrl
 - **用真值表建立指令和控制信号之间的关系**
 - **写出每个控制信号的逻辑表达式**
- 控制逻辑的输出(控制信号)在ID阶段生成, 并存放在ID/EX流水段寄存器中, 然后每来一个时钟跟着指令传送到下一级流水段寄存器
- 某时刻在不同阶段同时执行不同指令, 不同的指令得到不同控制信号

忘记单周期和多周期控制器设计的同学, 复习一下第六章!



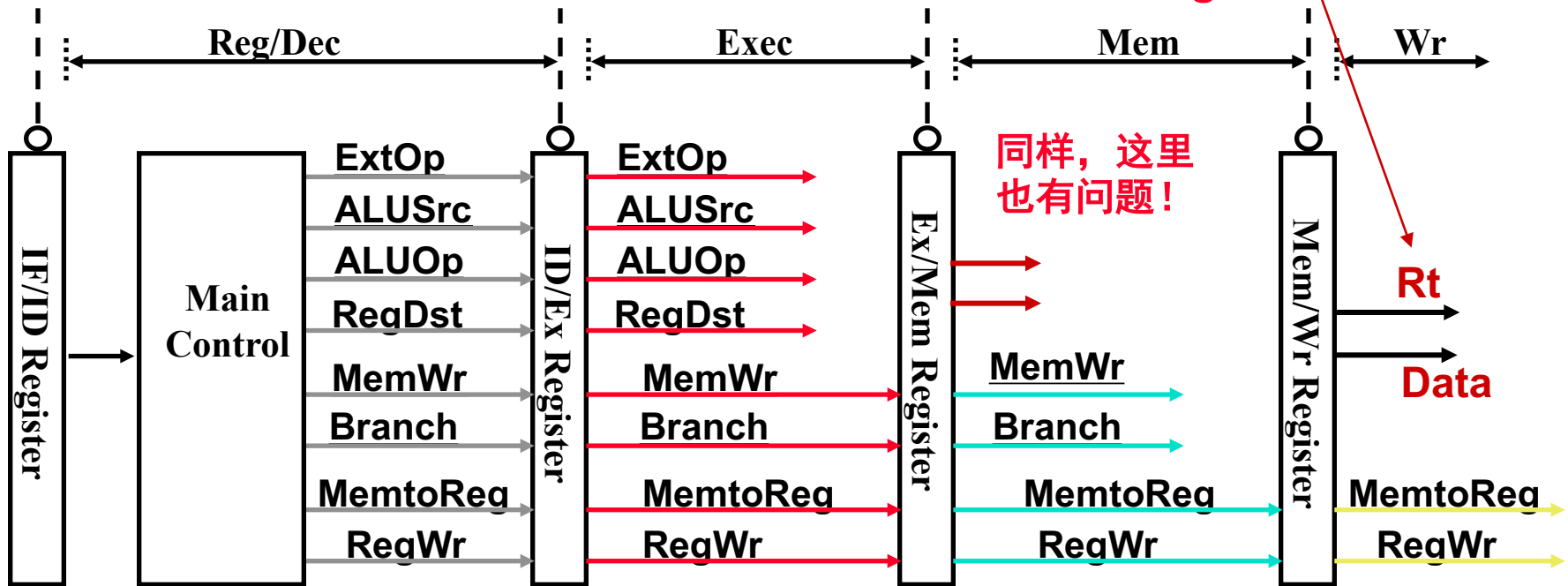
蓝色部分是控制信号

Load指令:流水线中的控制信号

- 在取数/译码 (Reg/Dec) 阶段产生本指令每个阶段的所有控制信号
 - Exec信号 (ExtOp, ALUSrc, ...) 在1个周期后使用
 - Mem信号 (MemWr, Branch) 在2个周期后使用
 - Wr信号 (MemtoReg, RegWr) 在3个周期后使用 (这里是否会有问题?)

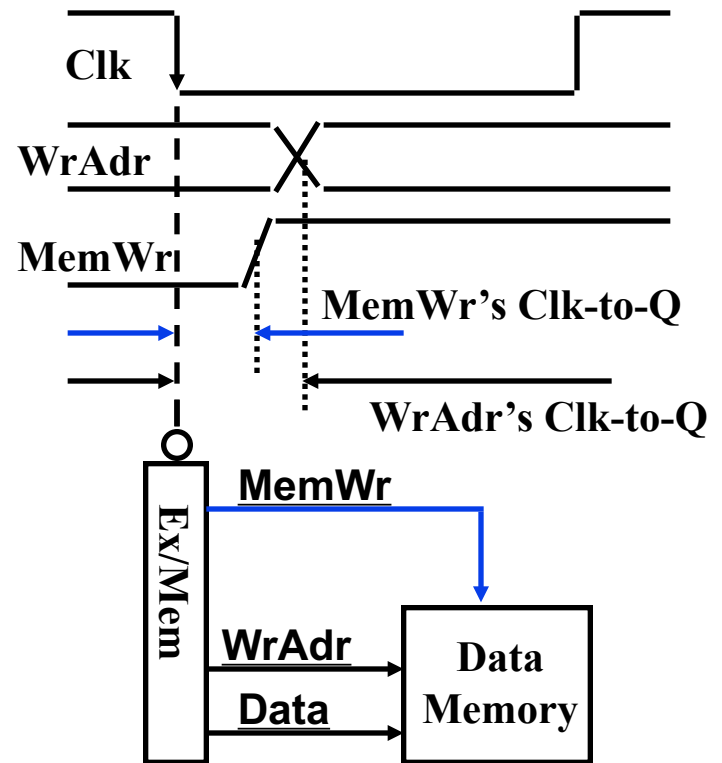
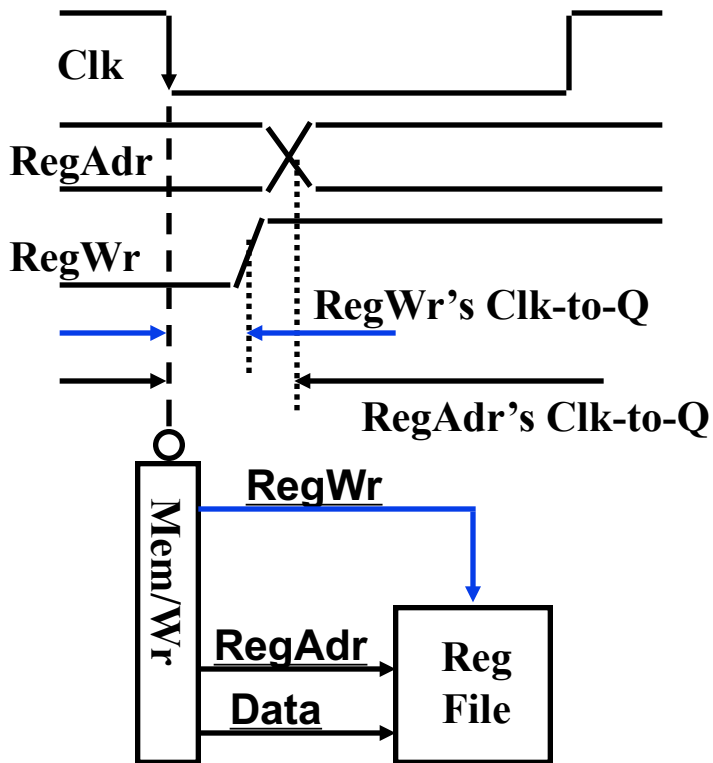
所以, 控制信号也要保存在流水段寄存器中!

Rt和Data在RegWr后到达咋办?



保存在流水段寄存器中的信息 (包括前面阶段传递来或执行的结果及控制信号) 一起被传递到下一个流水段!

Wr阶段的开始: 存在一个实际的问题!



- 在流水线中也存在地址 和 写使能之间的“竞争”问题
 - Wr段开始时若 $\text{RegAdr's (Rd/Rt) Clk-to-Q} > \text{RegWr's Clk-to-Q}$, 则错写寄存器!
 - Mem 阶段开始时若 $\text{WrAdr's Clk-to-Q} > \text{MemWr's Clk-to-Q}$, 则错写存储器!
- 不能用多周期中的方法! 为什么?

哪个同学记得多周期中是如何处理“竞争”问题的?

流水线中的“竞争”问题

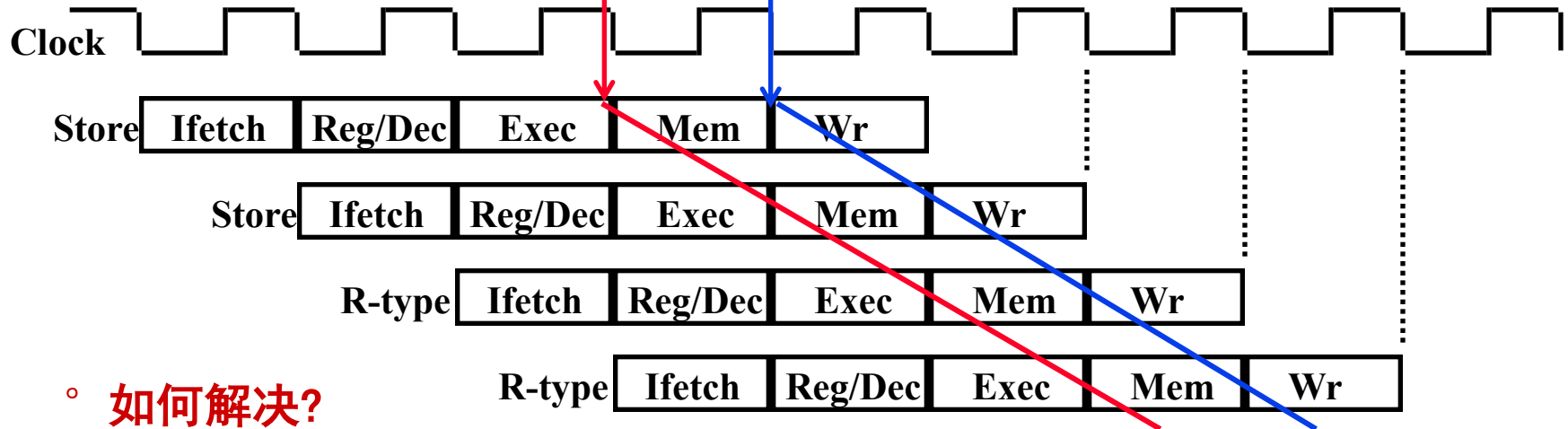
◦ 多周期中解决 Addr 和 WrEn 之间竞争问题的方法:

- 在第 N 周期结束时, 让 Addr 信号有效
- 在第 N + 1 周期让 WrEn 有效

} 保证 Addr 信号
在 WriteEnable
信号之前到达

◦ 上述方法在流水线设计中不能用, 因为:

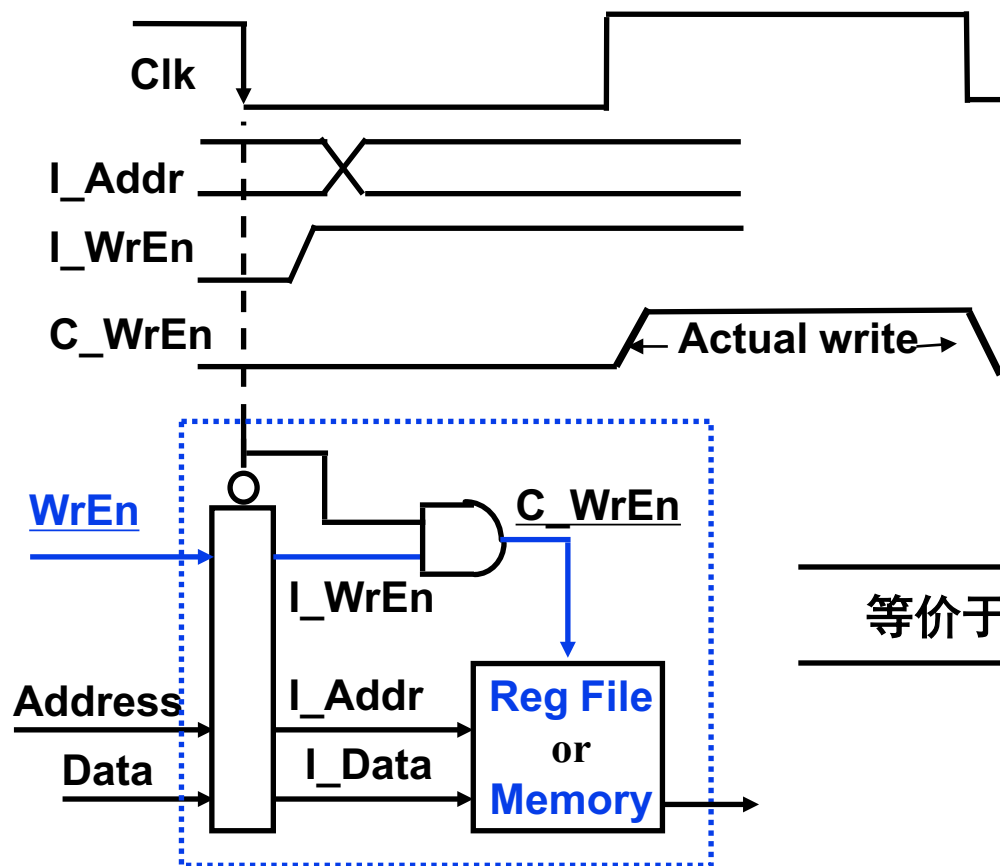
- 每个周期必须能够写 Register
- 每个周期必须能够写 Memory



寄存器组的同步和存储器的同步

- 解决方案：将Write Enable和时钟信号相“与”

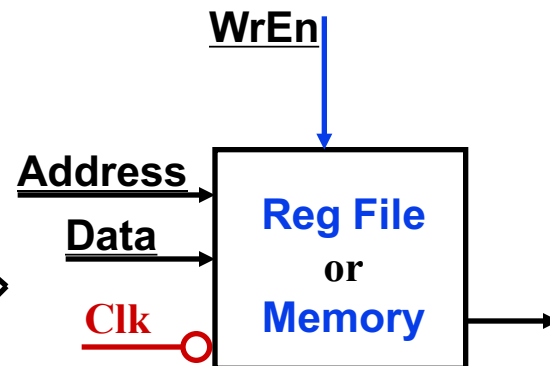
须由电路专家确保不会发生“定时错误”（即：能合理设计“Clock”!）



1. Addr, Data和 WrEn 必须在Clk边沿到来后至少稳定一个 set-up时间

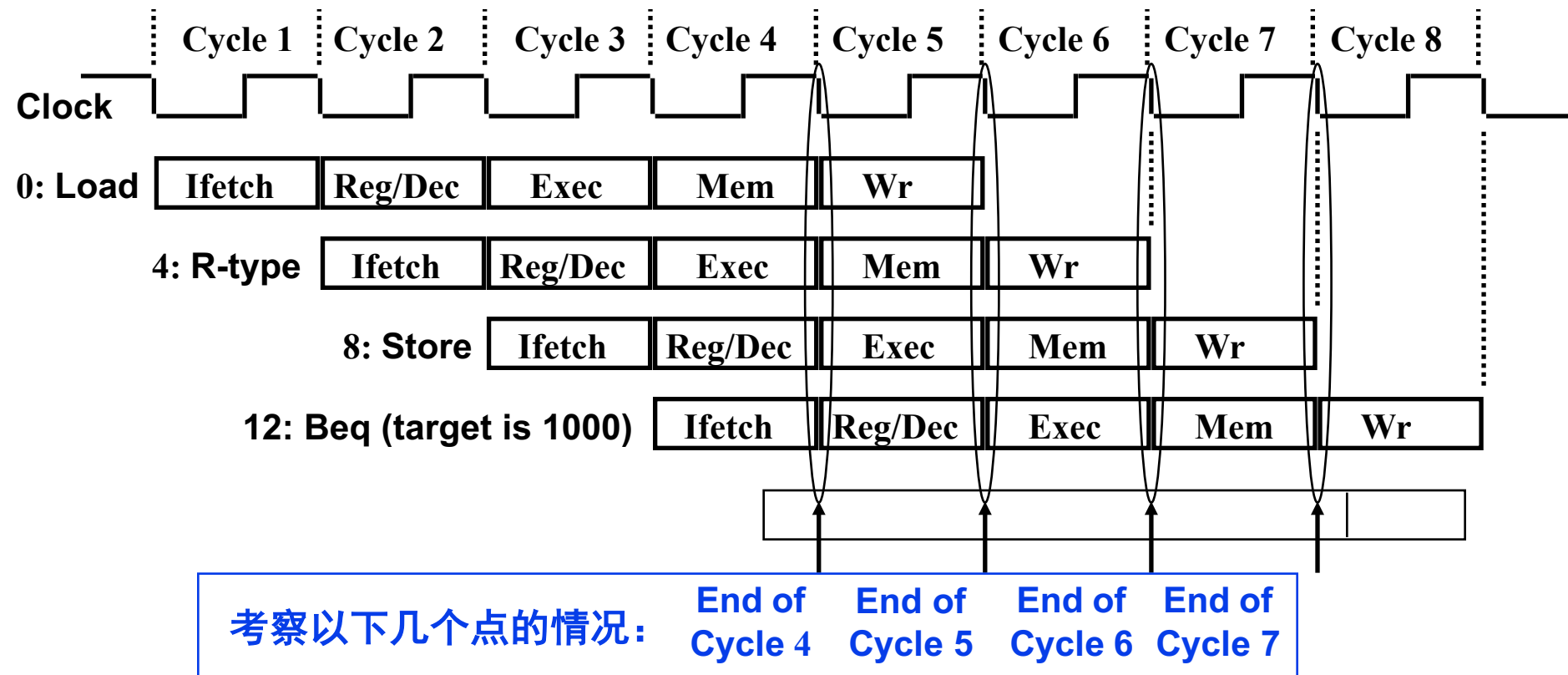
2. Clk高电平时间 大于 写入时间

等价于



相当于单周期通路中的理想寄存器和理想存储器

流水线举例：考察流水线DataPath的数据流动情况

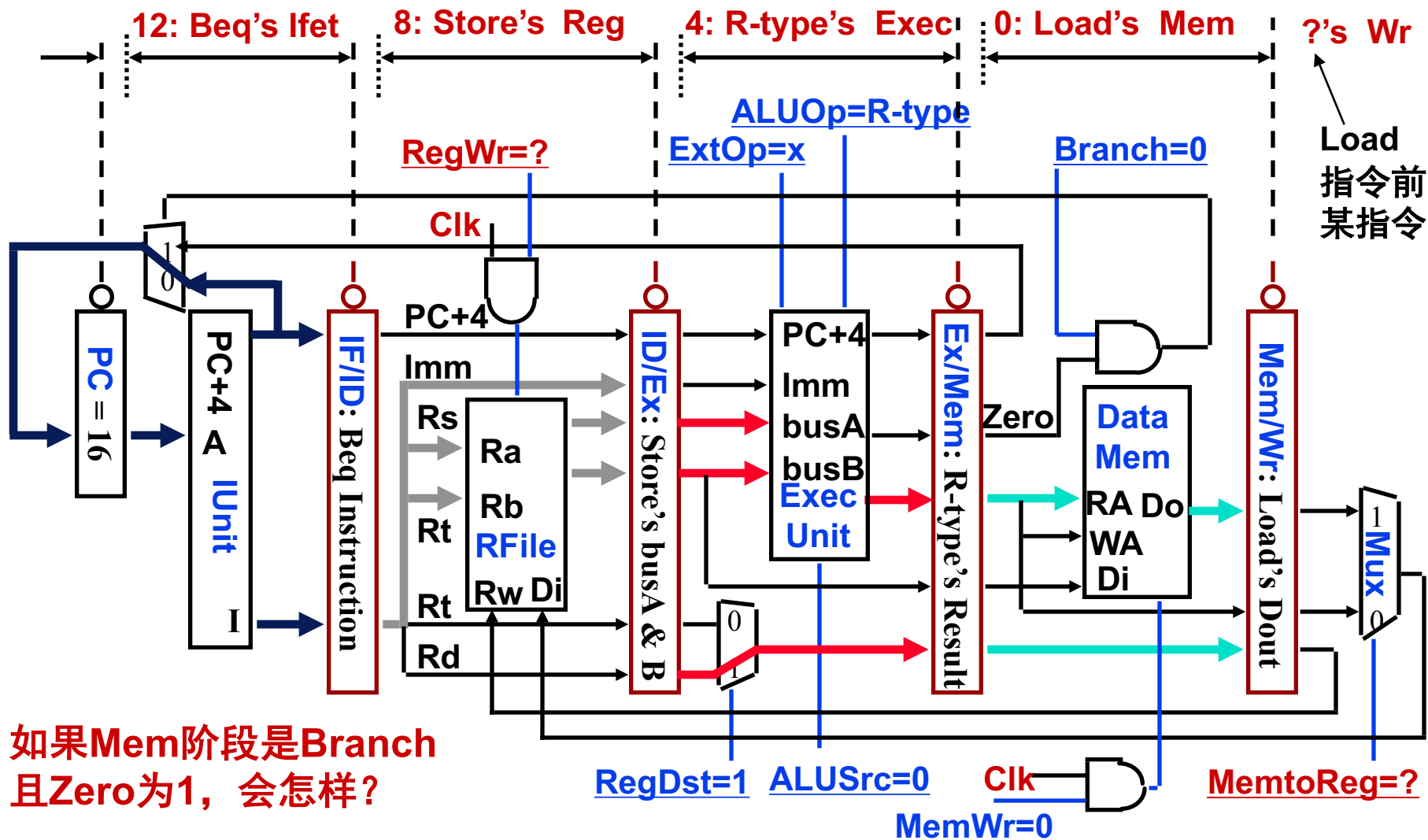


- End of Cycle 4: Load's Mem, R-type's Exec, Store's Reg, Beq's Ifetch
- End of Cycle 5: Load's Wr, R-type's Mem, Store's Exec, Beq's Reg
- End of Cycle 6: R-type's Wr, Store's Mem, Beq's Exec
- End of Cycle 7: Store's Wr, Beq's Mem

说明：后面仅考察数据流动情况，控制信号随数据同步流动不再说明。

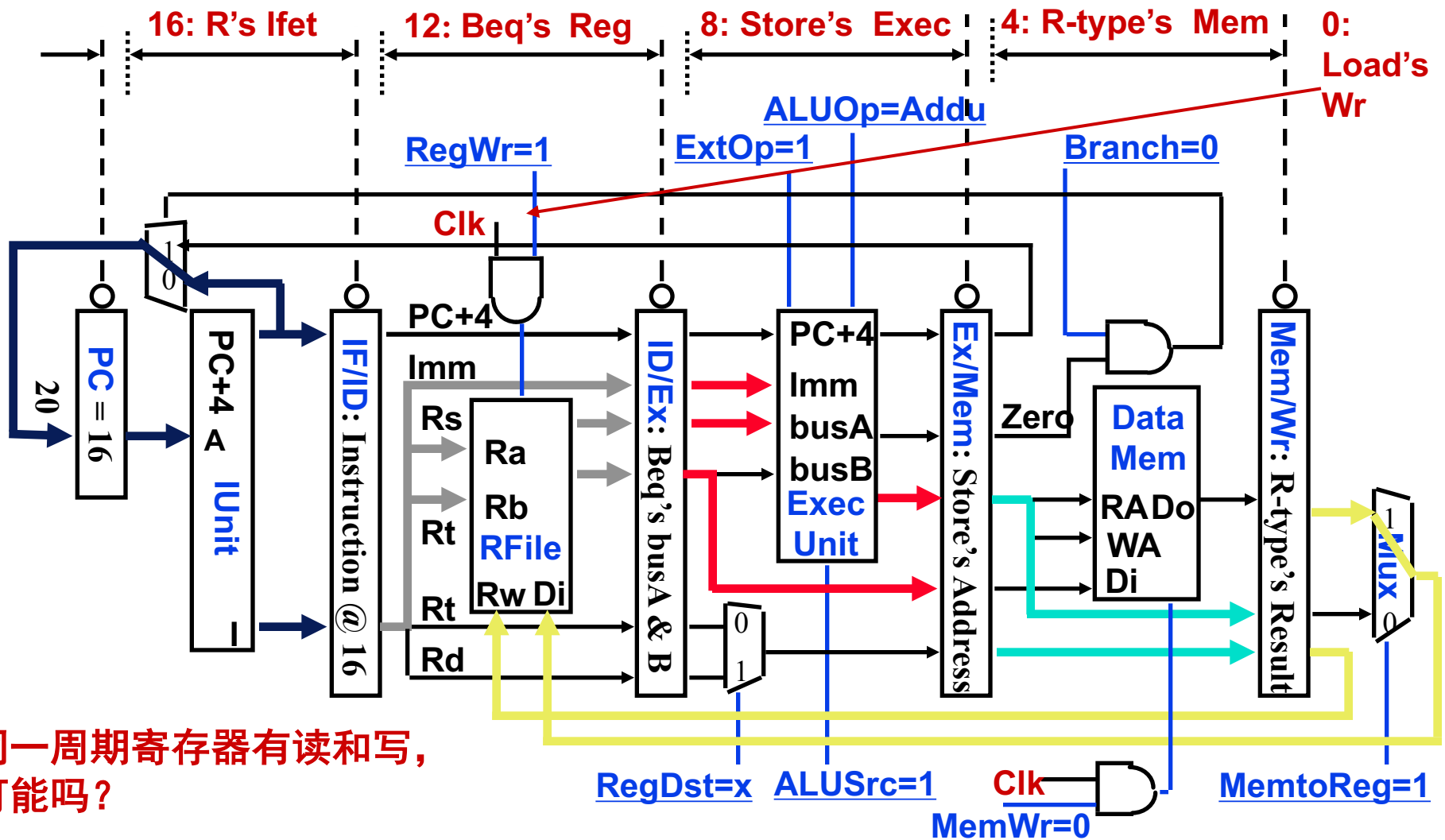
第四周期结束时的状态:

- 0: Load's Mem 4: R-type's Exec 8: Store's Reg 12: Beq's Ifetch



第五周期结束时的状态:

- 0: Lw's Wr 4: R's Mem 8: Store's Exec 12: Beq's Reg 16: R's Ifetch

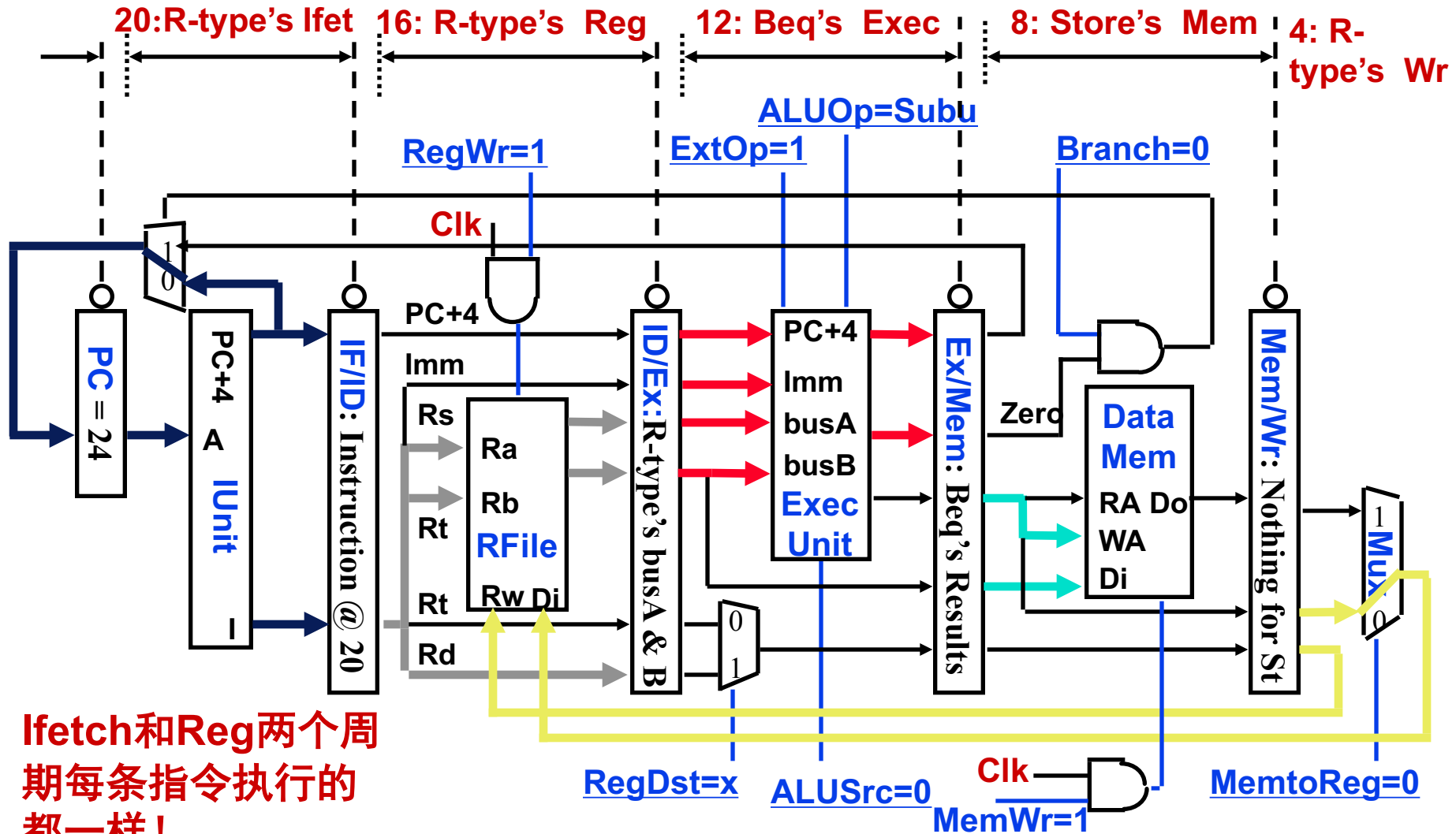


寄存器的写口和读口可看成是独立的两个部件！

利用时钟上升和下降沿两次触发，能做到前半周期写，后半周期读

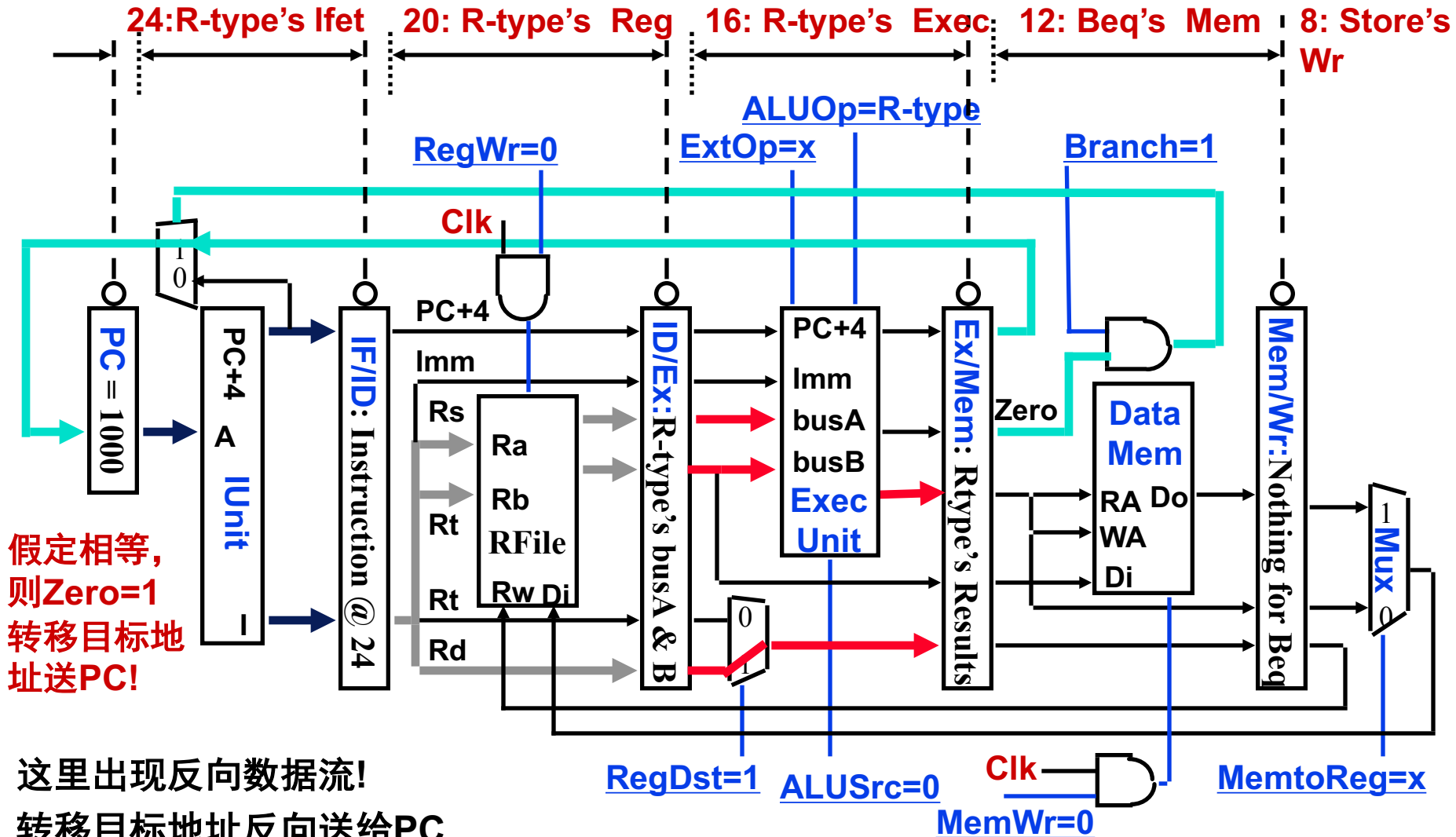
第六周期结束时的状态:

- 4: R's Wr 8: Store's Mem 12: Beq's Exec 16: R's Reg 20: R's Ifet



第七周期结束时的状态:

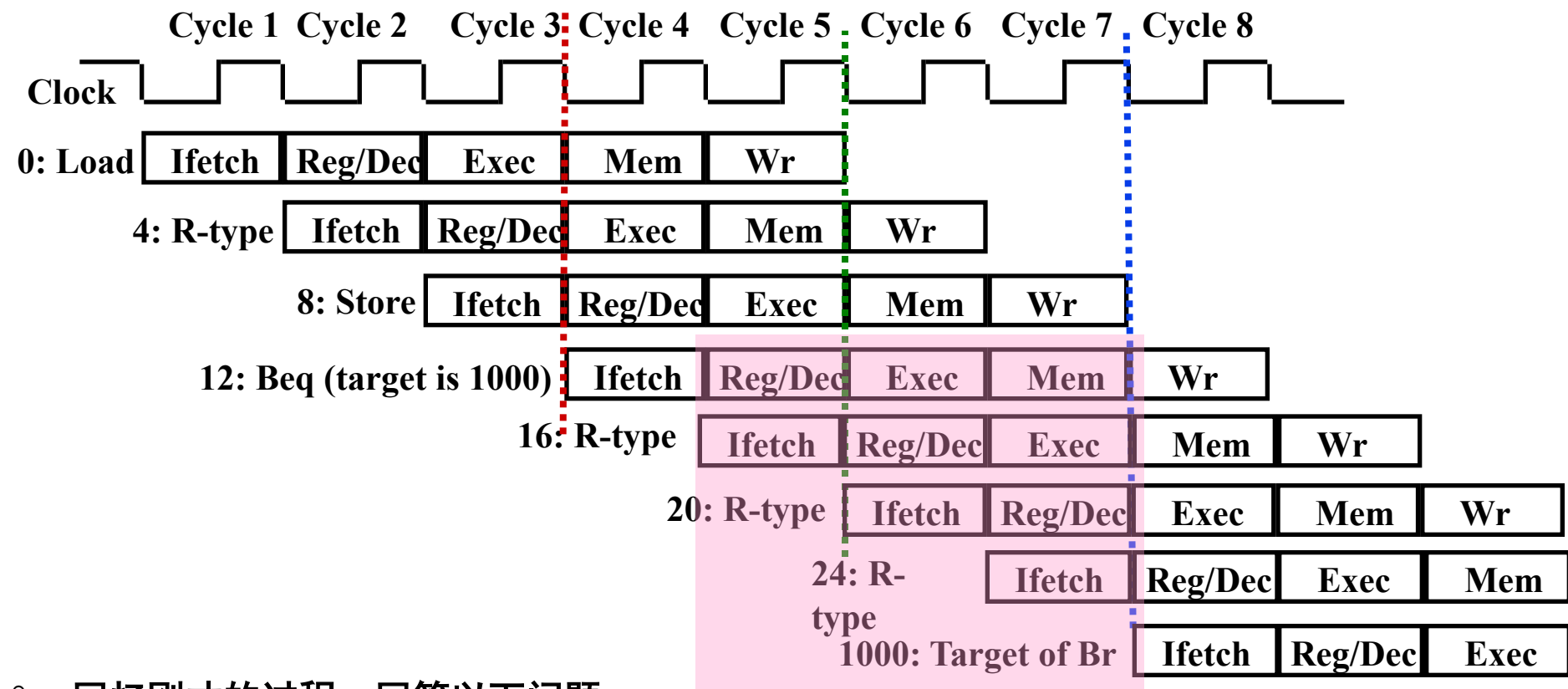
◦ 8: Store's Wr 12: Beq's Mem 16: R's Exec 20: R's Reg 24: R's Ifet



假定相等,
则Zero=1
转移目标地
址送PC!

这里出现反向数据流!
转移目标地址反向送给PC
可能会导致控制冒险!

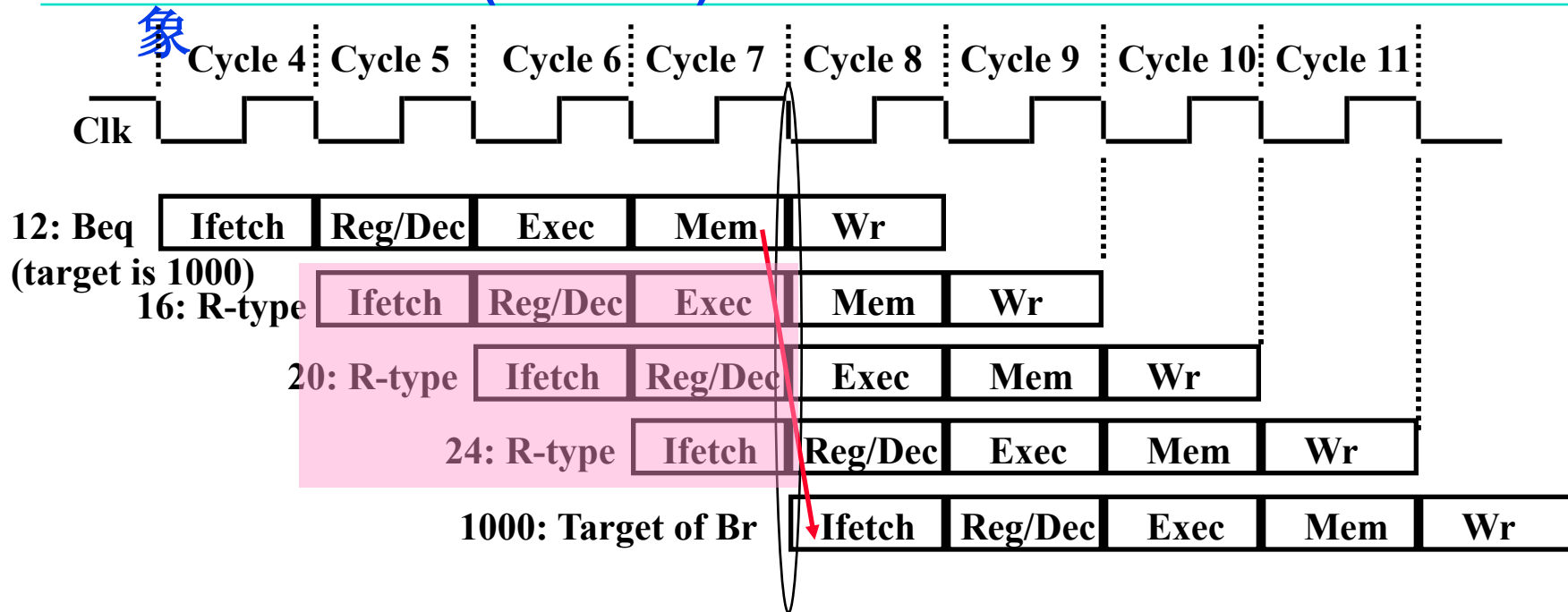
总结前面的流水线执行过程



回忆刚才的过程，回答以下问题：

- Branch指令何时确定是否转移？转移目标地址在第几周期计算出来？
 - 第六周期得到Zero和转移地址、第七周期控制转移地址送到PC输入端、第八周期开始才能根据转移地址取指令
 - 如果Branch指令执行结果是需要转移（称为taken），则流水线会怎样？
- Load指令何时能把数据写到寄存器？第几周期开始写数据？
 - 第五周期写入、第六周期开始才能使用
 - 如果后面R-Type的操作数是load指令目标寄存器的内容，则流水线怎样？

转移分支指令(Branch)引起的“延迟”现象



◦ 虽然Beq指令在第四周期取出，但：

- 目标地址在第七周期才被送到PC的输入端
- 第八周期才能取出目标地址处的指令执行

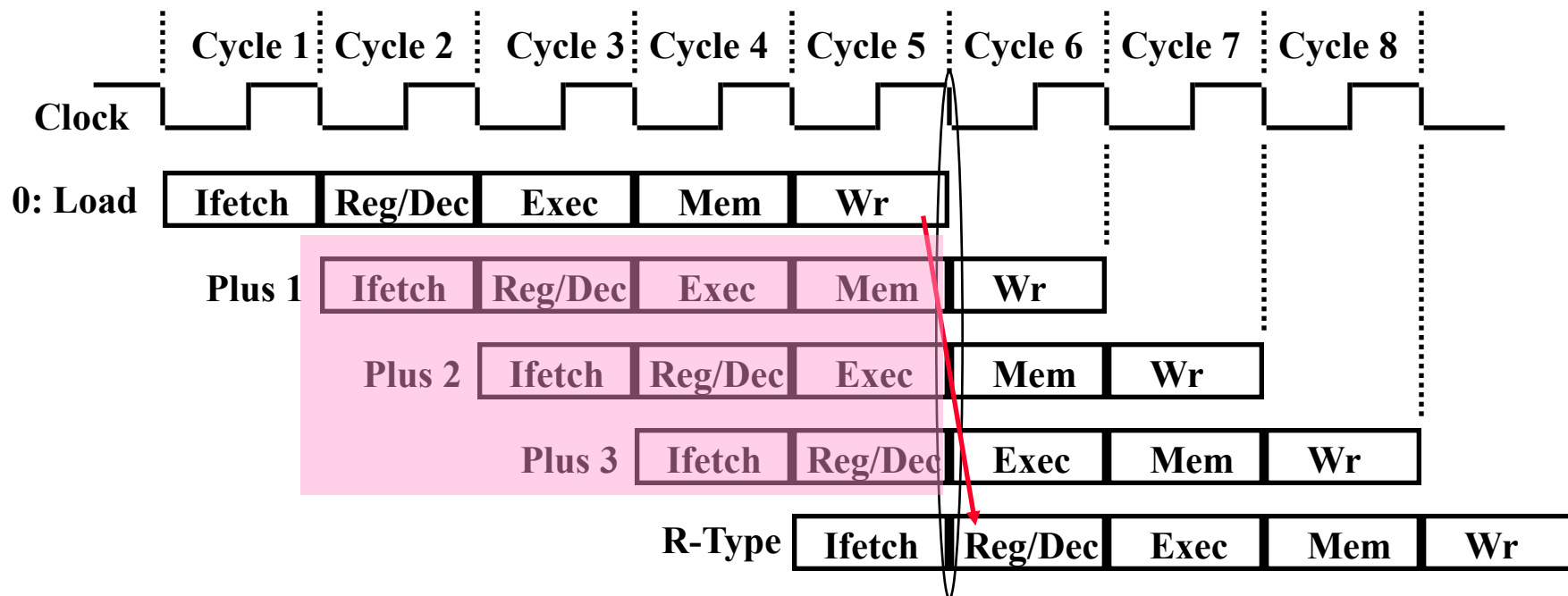
结果：在取目标指令之前，已有三条指令被取出，取错了三条指令！

◦ 这种现象称为控制冒险（Control Hazard）

（注：也称为分支冒险或转移冒险（Branch Hazard））

[BACK](#)

装入指令(Load)引起的“延迟”现象



◦ 尽管Load指令在第一周期就被取出，但：

- 数据在第五周期结束才被写入寄存器
- 到第六周期写入的数据才能被用

结果：如果随后指令要用到Load的数据的话，就需延迟三条指令！

◦ 这种现象被称为 **数据冒险 (Data Hazard)** 或 **数据相关(Data Dependency)**

第一讲内容小结

- 指令的执行可以像洗衣服一样，用流水线方式进行
 - 均衡时指令吞吐率提高N倍，但不能缩短一条指令的执行时间
 - 流水段数以最复杂指令所需步骤数为准（有些指令的某些阶段为空操作），每个阶段的宽度以最复杂阶段所需时间为准（尽量调整使各阶段均衡）
- 以Load指令为准，分为五个阶段
 - 取指令段(IF)
 - 取指令、计算PC+4（IUnit: Instruction Memory、Adder）
 - 译码/读寄存器(ID/RF)段
 - 指令译码、读Rs和Rt（寄存器读口）
 - 执行(EXE)段
 - 计算转移目标地址、ALU运算（Extender、ALU、Adder）
 - 存储器(MEM)段
 - 读或写存储单元（Data Memory）
 - 写寄存器(Wr)段
 - ALU结果或从DM读出数据写到寄存器（寄存器写口）
- 流水线控制器的实现
 - IF和ID/RF段不需控制信号控制，只有EXE、MEM和Wr需要
 - ID段生成所有控制信号，并随指令的数据同步向后续阶段流动
- 寄存器和存储器的竞争问题可利用时钟信号来解决
- 流水线冒险：结构冒险、控制冒险、数据冒险
（下一讲主要介绍解决流水线冒险的数据通路如何设计）

第二讲 流水线冒险的处理

主要内容

- 流水线冒险的几种类型
- 数据冒险的现象和对策
 - 数据冒险的种类
 - 相关的数据是ALU结果：可以通过转发解决
 - 相关的数据是DM读出的内容：随后的指令需被阻塞一个时钟
 - 数据冒险和转发
 - 转发检测 / 转发控制
 - 数据冒险和阻塞
 - 阻塞检测 / 阻塞控制
- 控制冒险的现象和对策
 - 静态分支预测技术
 - 动态分支预测技术
 - 缩短分支延迟技术
- 流水线中对异常和中断的处理
- 访问缺失对流水线的影响

总结：流水线的三种冲突/冒险（Hazard）情况

○ **Hazards**：指流水线遇到无法正确执行后续指令或执行了不该执行的指令

- **Structural hazards** (hardware resource conflicts):

现象：同一个部件同时被不同指令所使用

- 一个部件每条指令只能使用1次，且只能在特定周期使用
- 设置多个部件，以避免冲突。如指令存储器IM 和数据存储器DM分开

- **Data hazards** (data dependencies):

现象：后面指令用到前面指令结果数据时，前面指令的结果还没产生

- 采用转发(Forwarding/Bypassing)技术
- Load-use冒险需要一次阻塞(stall)
- 编译程序优化指令顺序

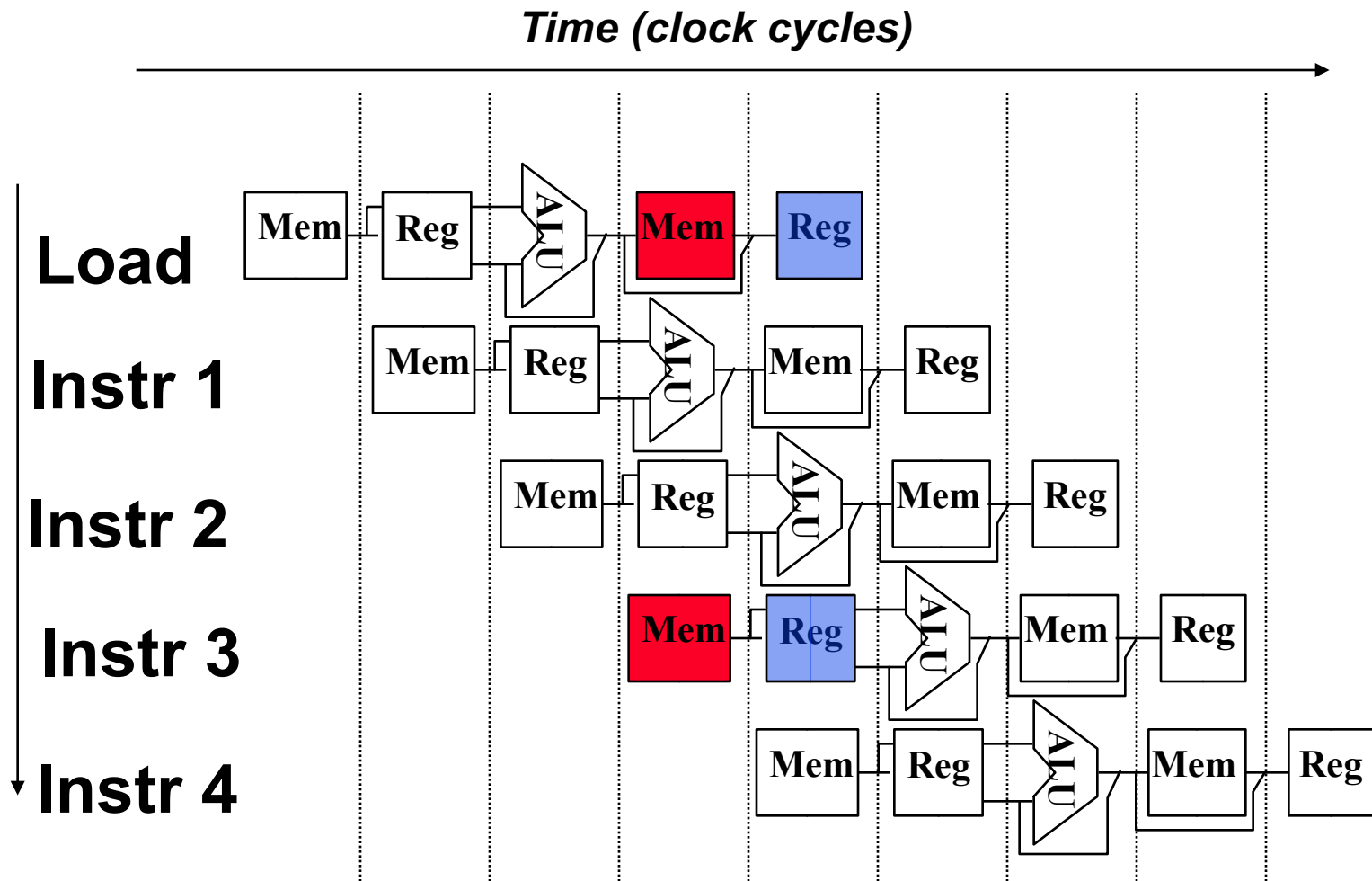
- **Control (Branch) hazards** (changes in program flow):

现象：转移或异常改变执行流程，后继指令在目标地址产生前已被取出

- 采用静态或动态分支预测
- 编译程序优化指令顺序(分支延迟)

SKIP

Structural Hazard (结构冒险) 现象



只有一个存储器时，在Load指令取数据同时又取指令的话，则发生冲突！

如果不对寄存器堆的写口和读口独立设置的话，则发生冲突！

结构冒险也称硬件资源冲突：同一个执行部件被多条指令使用。

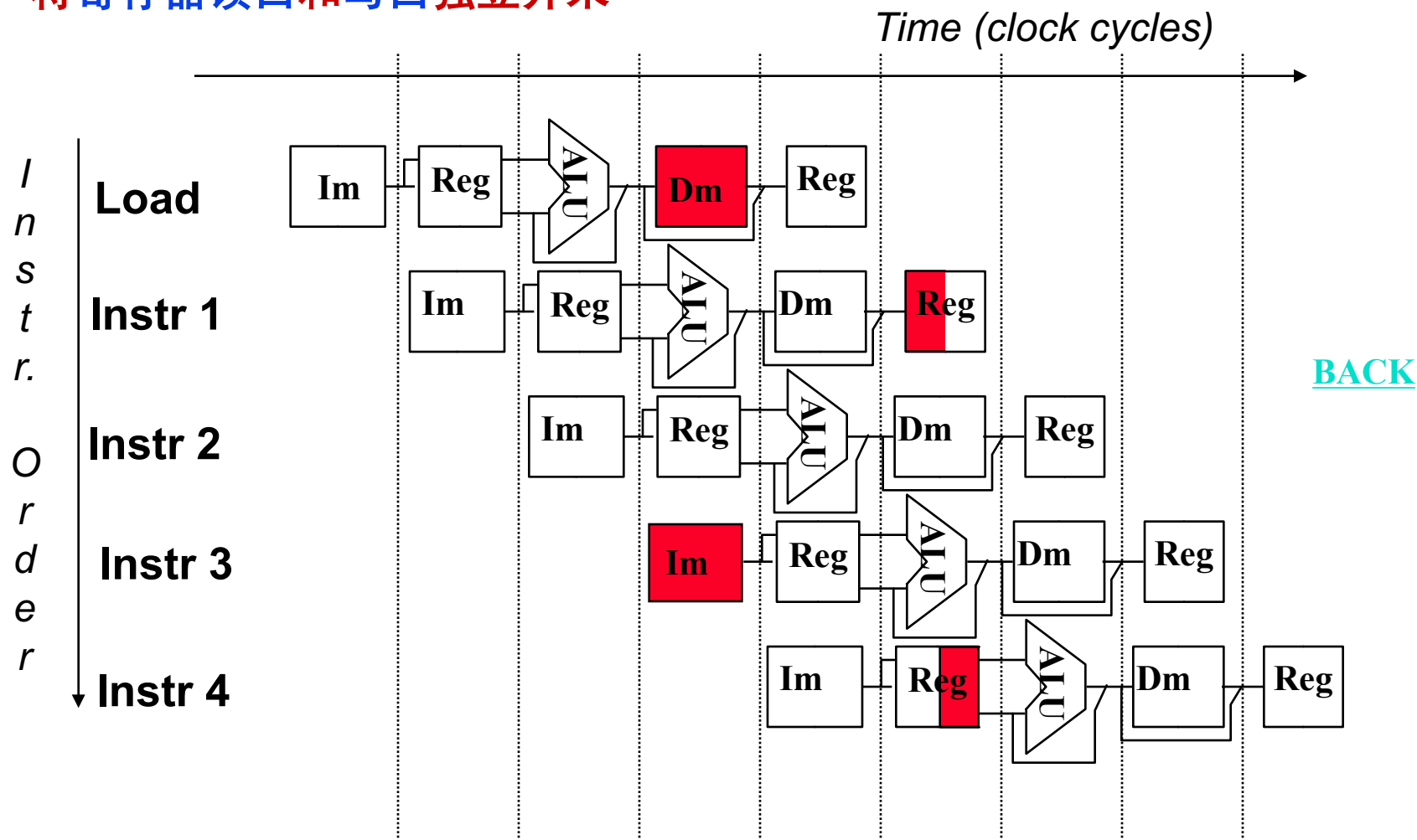
Structural Hazard的解决方法

为了避免结构冒险，规定流水线数据通路中功能部件的设置原则为：

每个部件在特定的阶段被用！（如：ALU总在第三阶段被用！）

将Instruction Memory (Im) 和 Data Memory (Dm)分开

将寄存器读口和写口独立开来



Data Hazard现象

举例说明：以下指令序列中，寄存器r1会发生数据冒险

想一下，哪条指令的r1是老的值？

哪条是新的值？

add r1, r2, r3

sub r4, r1, r3 读r1时，add指令正在执行加法(EXE)，老值！

and r6, r1, r7 读r1时，add指令正在传递加法结果(MEM)，老值！

or r8, r1, r9 读r1时，add指令正在写加法结果到r1(WB)，老值！

xor r10, r1, r11 读r1时，add指令已经把加法结果写到r1，新值

画出流水线图能很清楚理解！

补充：三类数据冒险现象

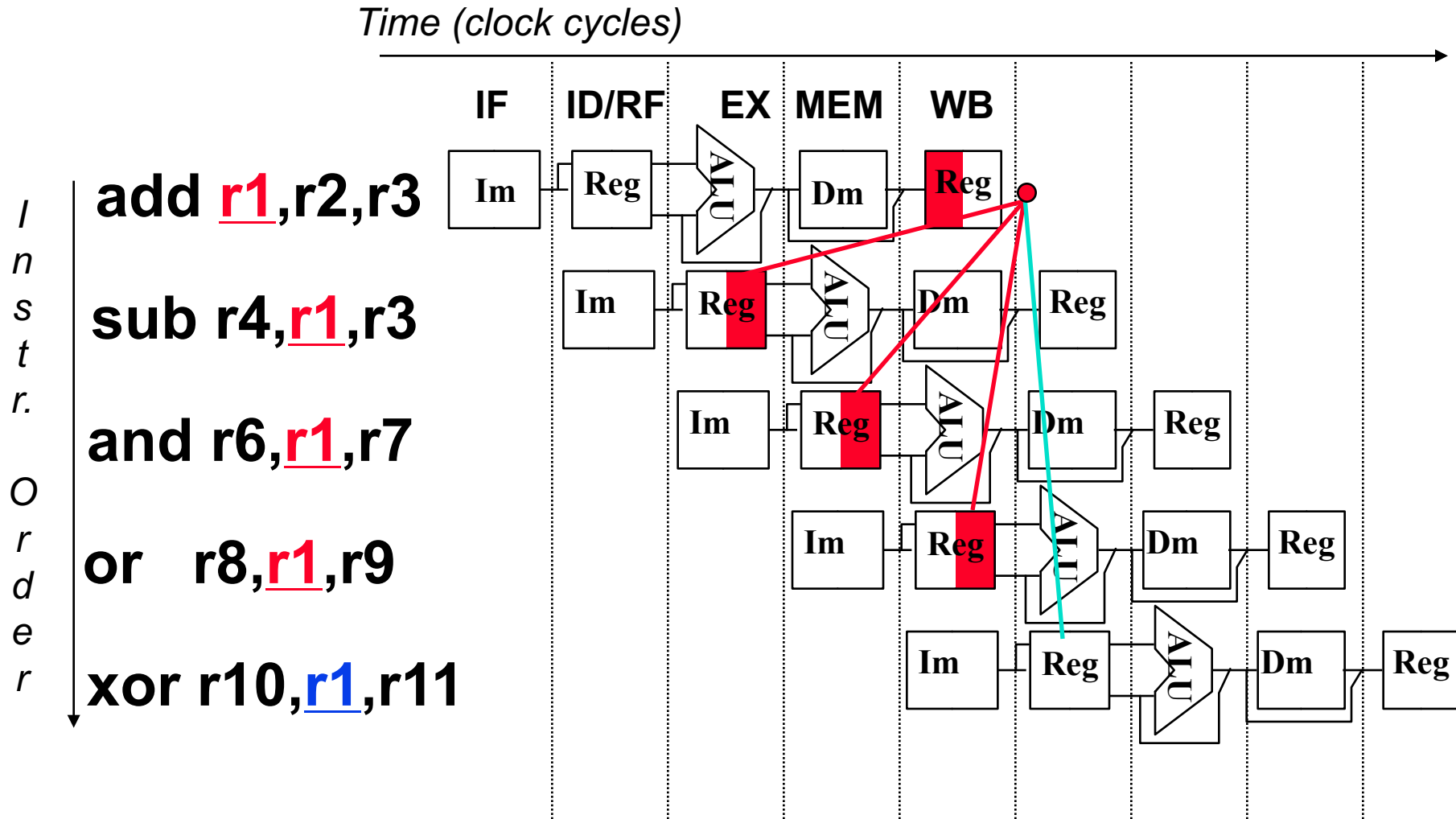
RAW: 写后读（基本流水线中经常发生，如上例）

WAR: 读后写（基本流水线中不会发生，乱序执行时会发生）

WAW: 写后写（基本流水线中不会发生，乱序执行时会发生）

本讲介绍基本流水线，所以仅考虑RAW冒险

Data Hazard on r1



最后一条指令的r1才是新的值！

如何解决这个问题？

数据冒险的解决方法

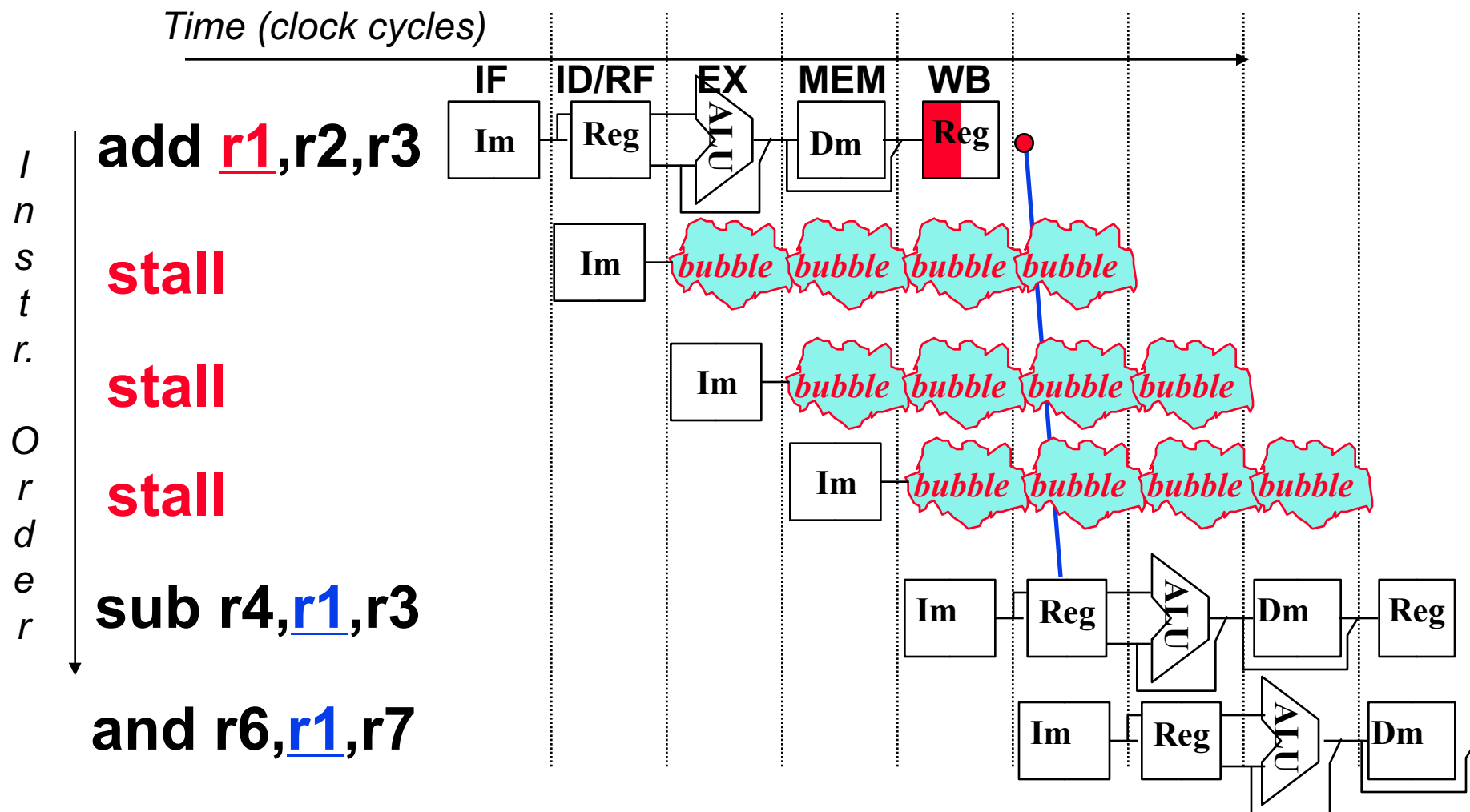
- 方法1：硬件阻塞（stall）
- 方法2：软件插入“NOP”指令
- 方法3：合理实现寄存器堆的读/写操作（不能解决所有数据冒险）
 - 前半时钟周期写，后半时钟周期读，若同一个时钟内前面指令写入的数据正好是后面指令所读数据，则不会发生数据冒险
- 方法4：转发（Forwarding或Bypassing 旁路）技术
 - 若相关数据是ALU结果，则如何？
可通过转发解决
 - 若相关数据是上条指令DM读出内容，则如何？
不能通过转发解决，随后指令需被阻塞一个时钟 或 加NOP指令
称为Load-use数据冒险！
- 方法5：编译优化：调整指令顺序（不能解决所有数据冒险）

实现“转发”和“阻塞”要修改数据通路：

- 检测何时需要“转发”，并控制实现“转发”
- 检测何时需要“阻塞”，并控制实现“阻塞”

方案1: 在硬件上采取措施, 使相关指令延迟执行

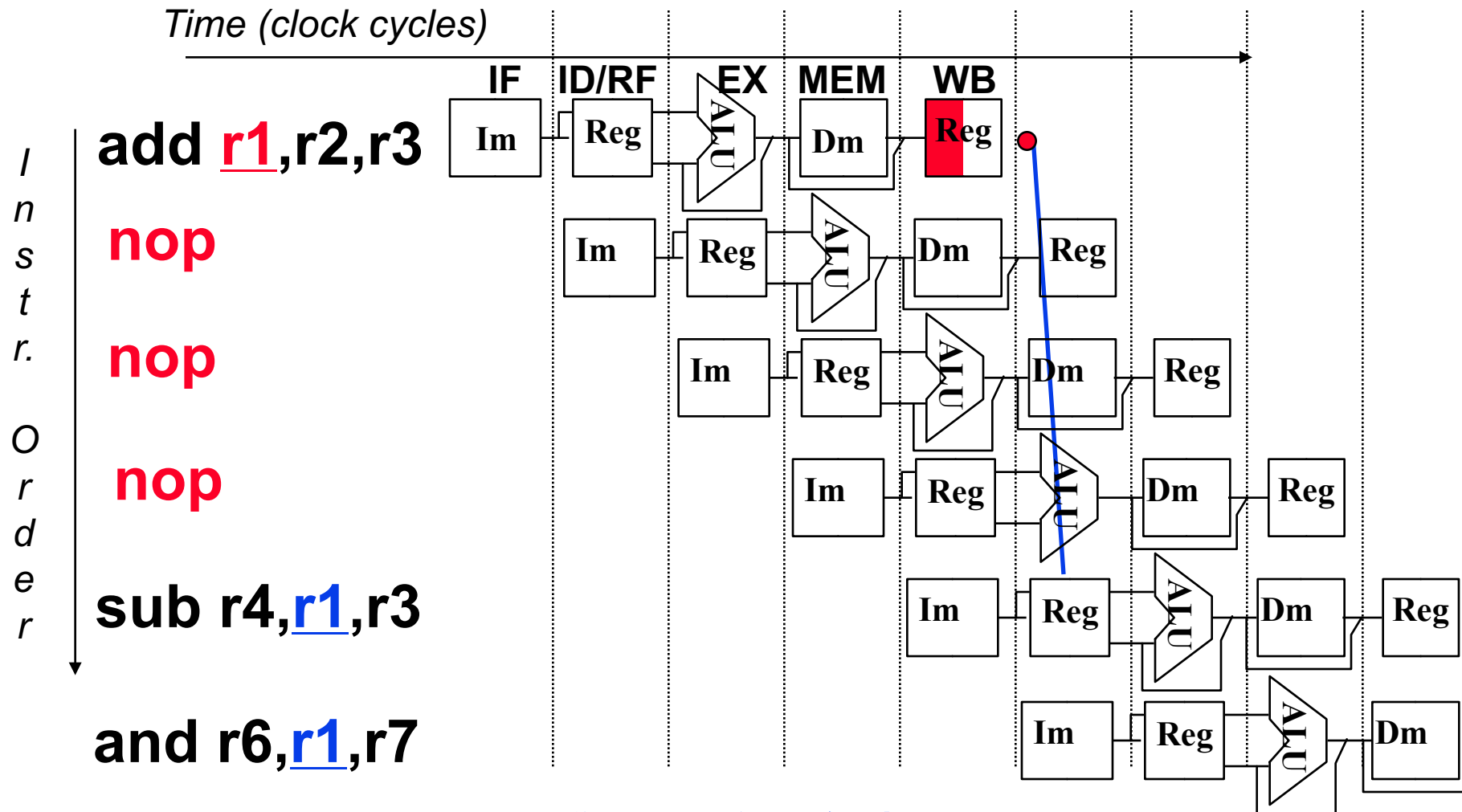
- 硬件上通过阻塞(stall)方式阻止后续指令执行, 延迟到有新值以后!
这种做法称为流水线阻塞, 也称为插入“气泡Bubble”



- 缺点: 控制比较复杂, 需要改数据通路; 指令被延迟三个时钟执行。

方案 2: 软件上插入无关指令

- 由编译器插入三条NOP指令，浪费三条指令的空间和时间，是最差的做法。
好处：数据通路简单，即无需改数据通路。

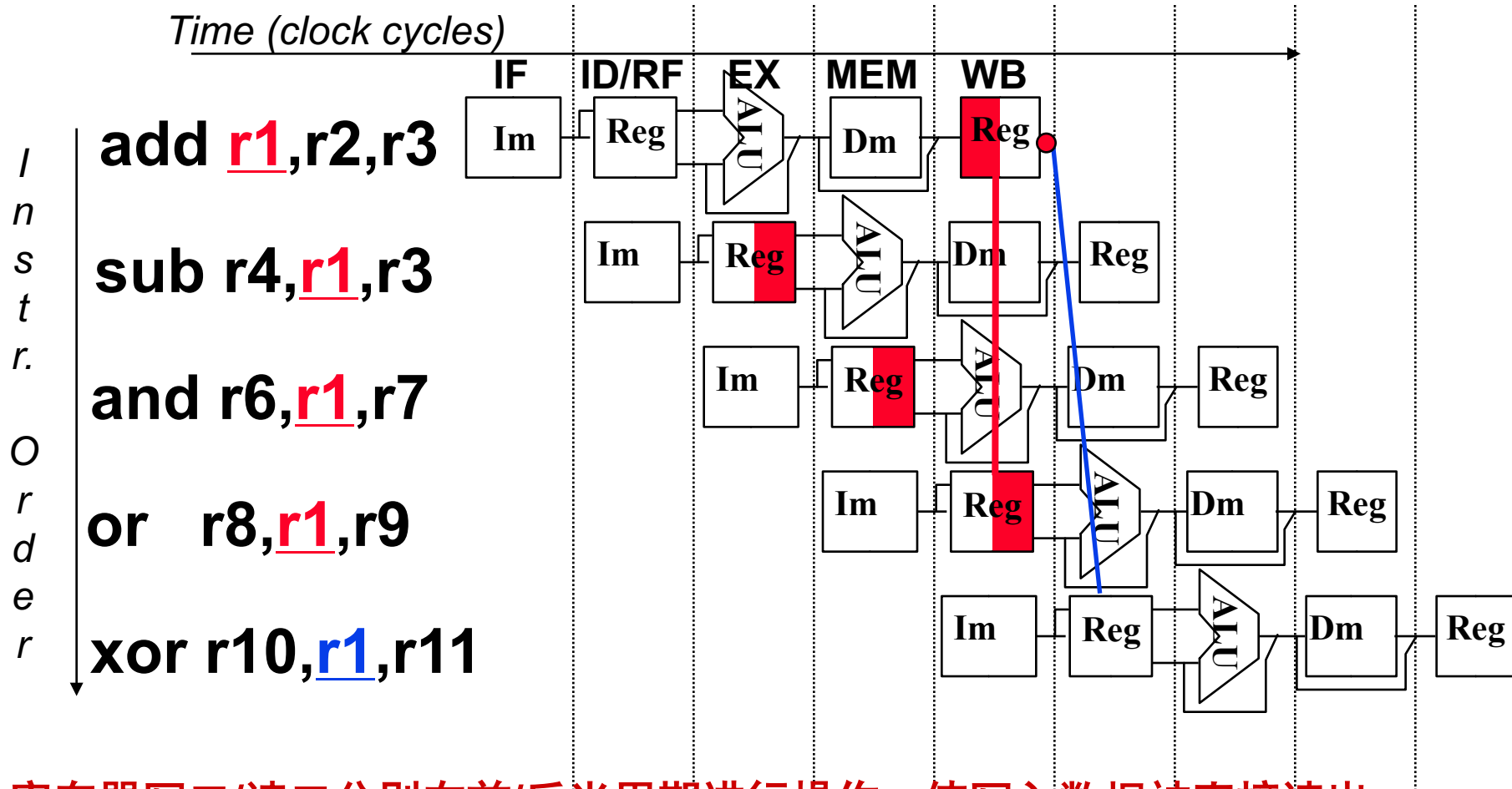


与方案1比，哪个更快？

一样，都是多三个时钟周期！

方案3: 同一周期内寄存器堆先写后读

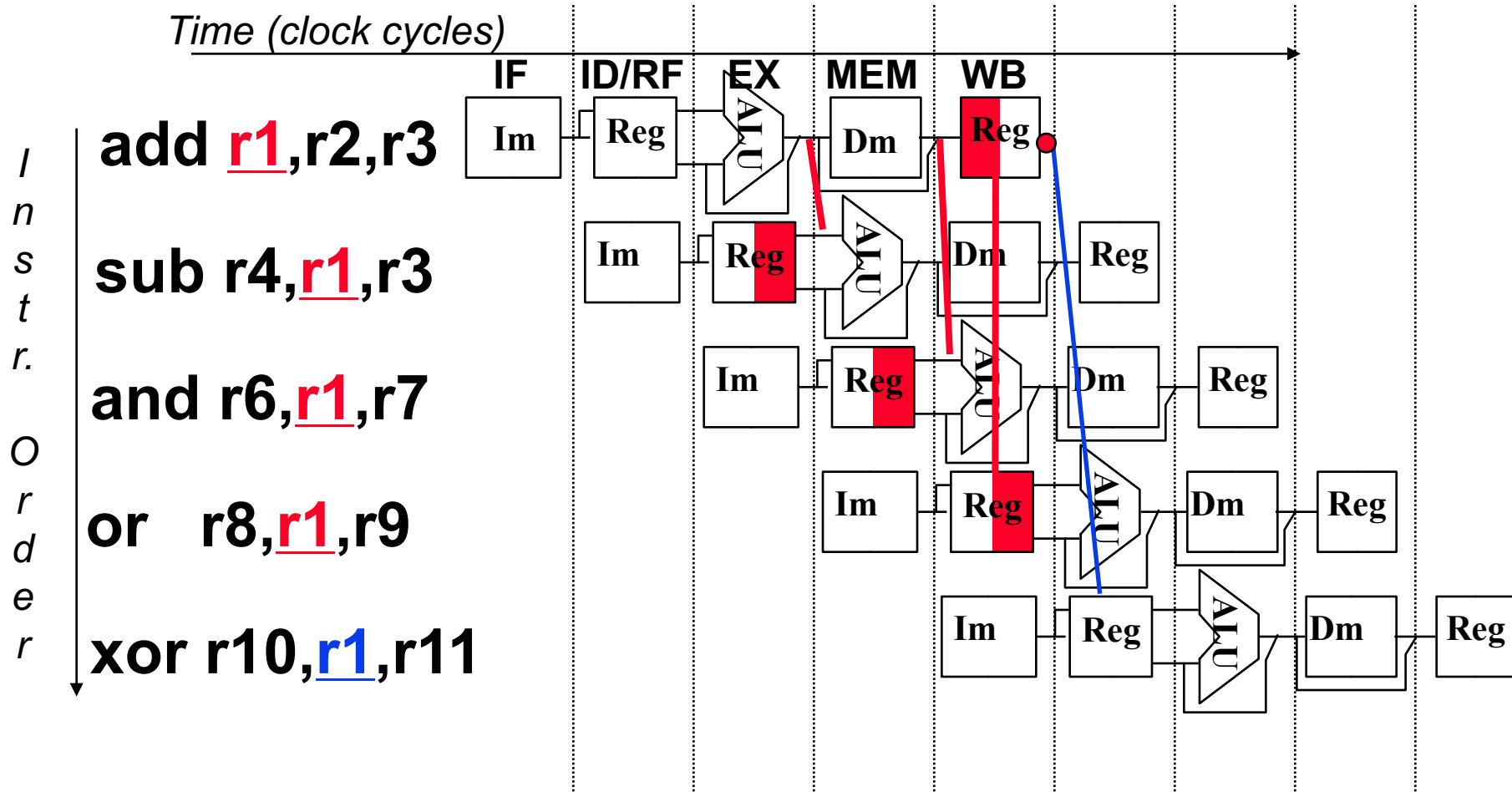
- 寄存器堆的读口和写口是相互独立的部件！



寄存器写口/读口分别在前/后半周期进行操作，使写入数据被直接读出
但是，只能解决部分数据冒险！

方案4: 利用DataPath中的中间数据: 转发+阻塞

- 仔细观察后发现: 流水段寄存器中已有需要的值r1! 在哪个流水段R中?

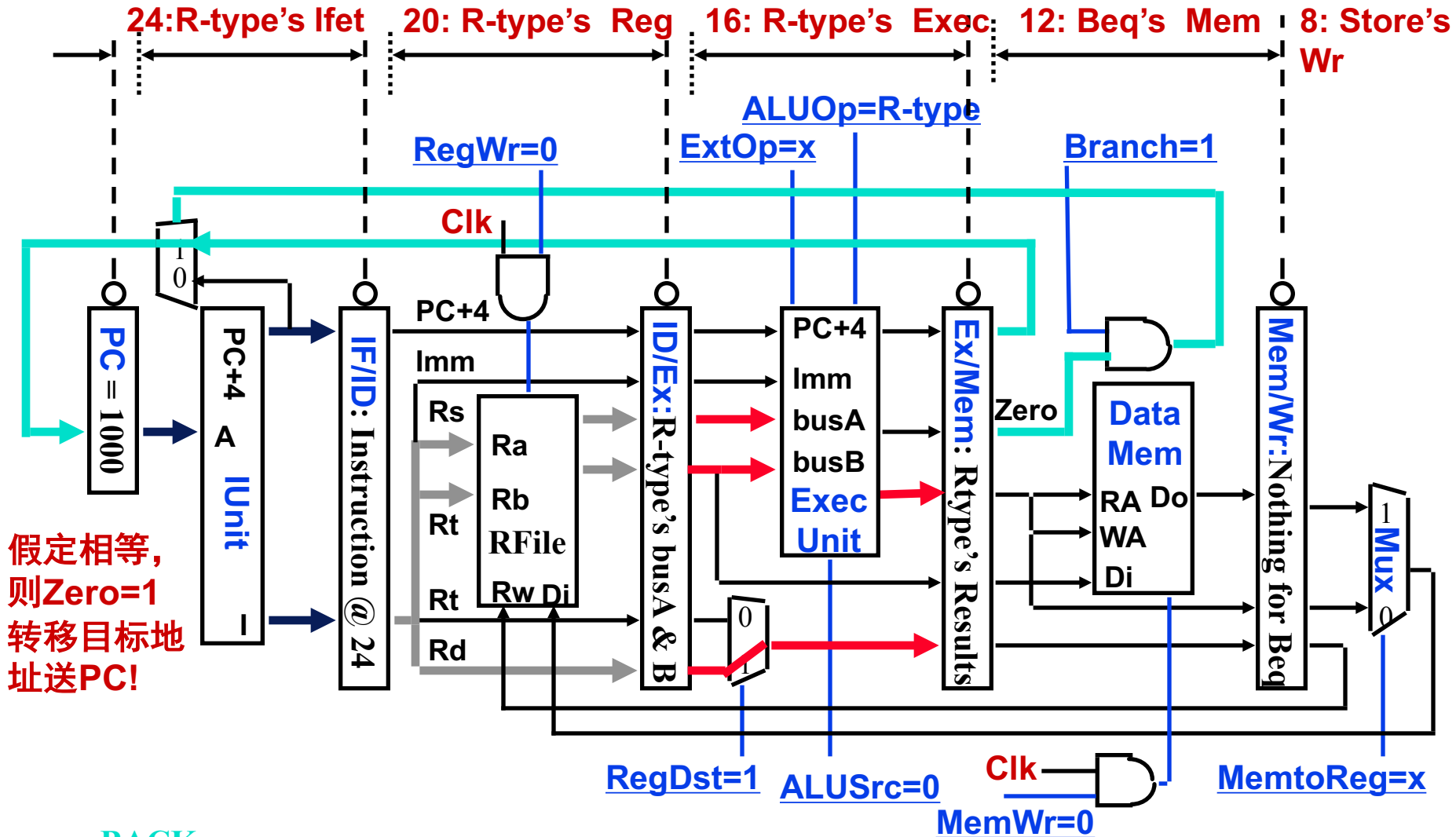


把数据从流水段寄存器中直接取到ALU的输入端

称为转发 (Forwarding)
或旁路 (Bypassing)

第七周期结束时的状态:

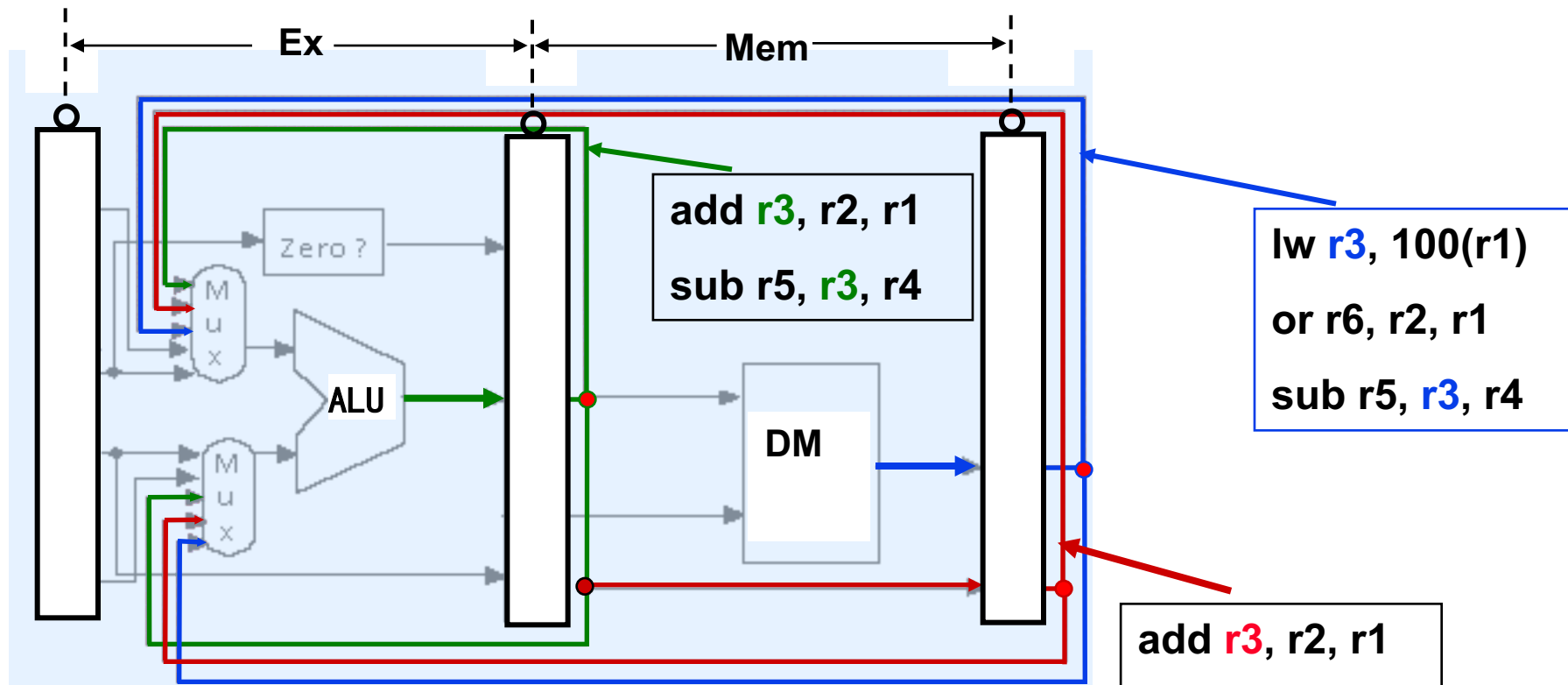
◦ 8: Store's Wr 12: Beq's Mem 16: R's Exec 20: R's Reg 24: R's Ifet



[BACK](#)

硬件上的改动以支持“转发”技术

- 加MUX，使流水段寄存器值返送ALU输入端



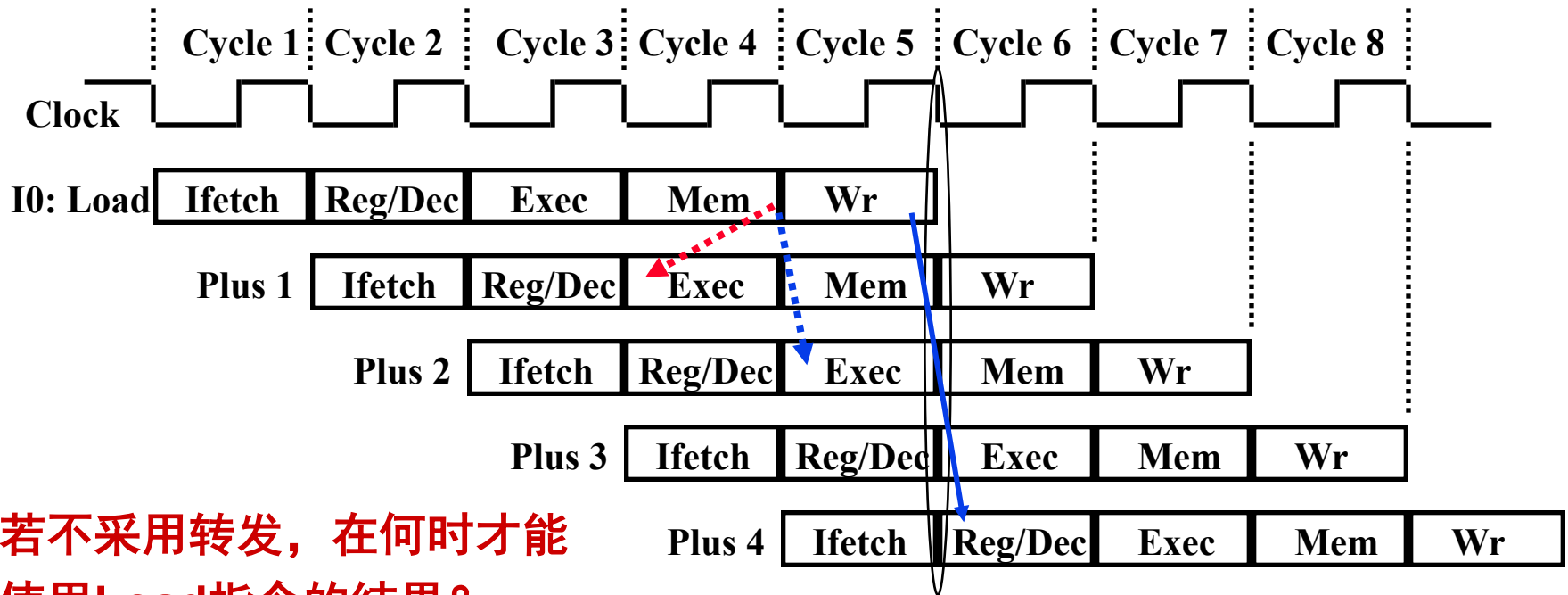
如果指令序列为

```
lw r3, 100(r1)
or r6, r3, r1
sub r5, r3, r4
```

能用“转发”技术
解决第1、2两条指
令间的数据冒险吗
?

请看后面的幻灯片!

复习: Load指令引起的延迟现象



若不采用转发，在何时才能使用Load指令的结果？

◦ Load指令最早在哪个流水线寄存器中开始有后续指令需要的值？

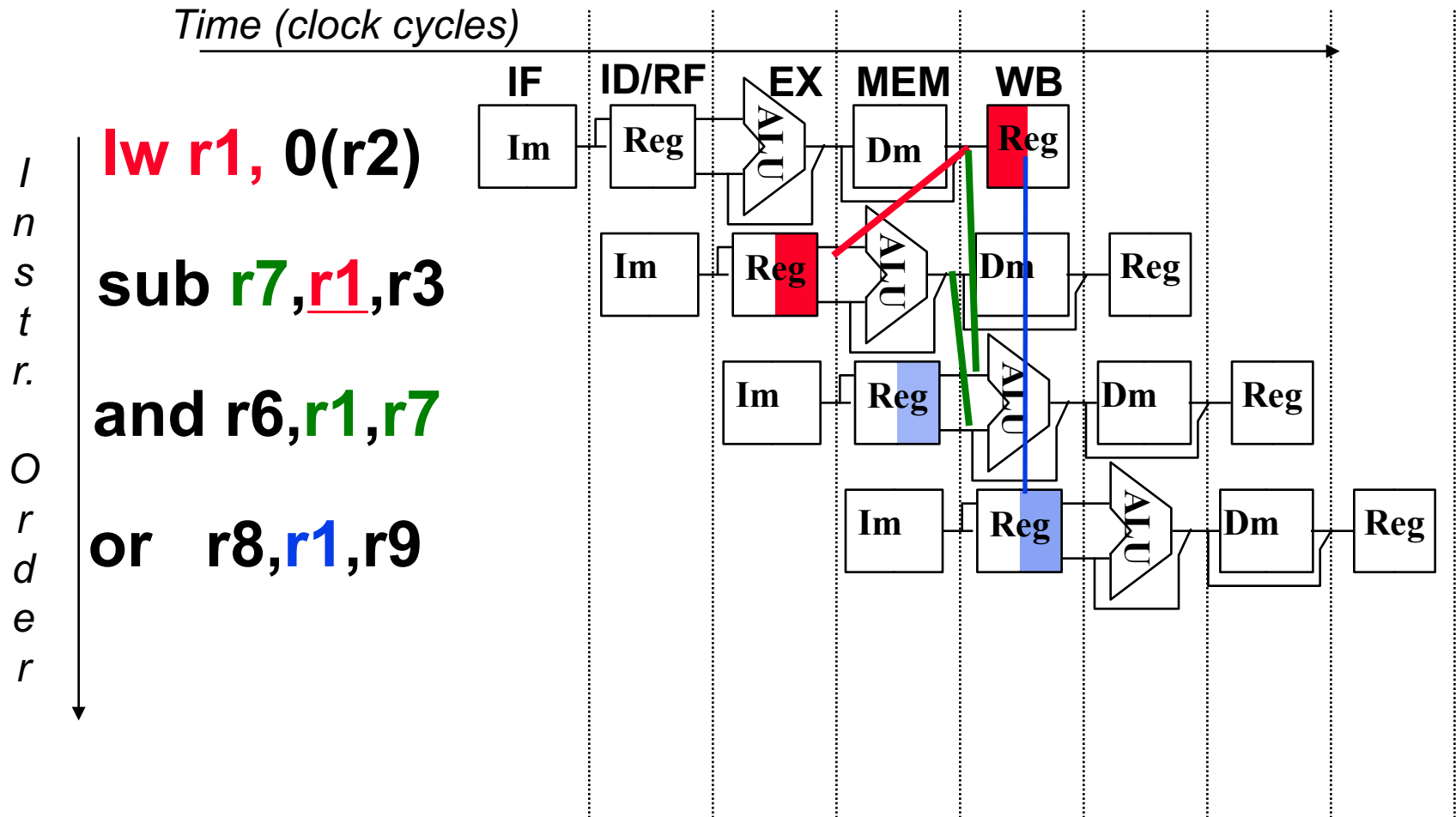
实际上，在第四周期结束时，数据在流水段寄存器中已经有值。

采用数据转发技术可以使load指令后面第二条指令得到所需的值

但不能解决load指令和随后第一条指令间的数据冒险，要延迟执行一条指令！

这种load指令和随后指令间的数据冒险，称为“装入-使用数据冒险 (load- use Data Hazard)”

“Forwarding”技术使Load-use冒险只需延迟一个周期

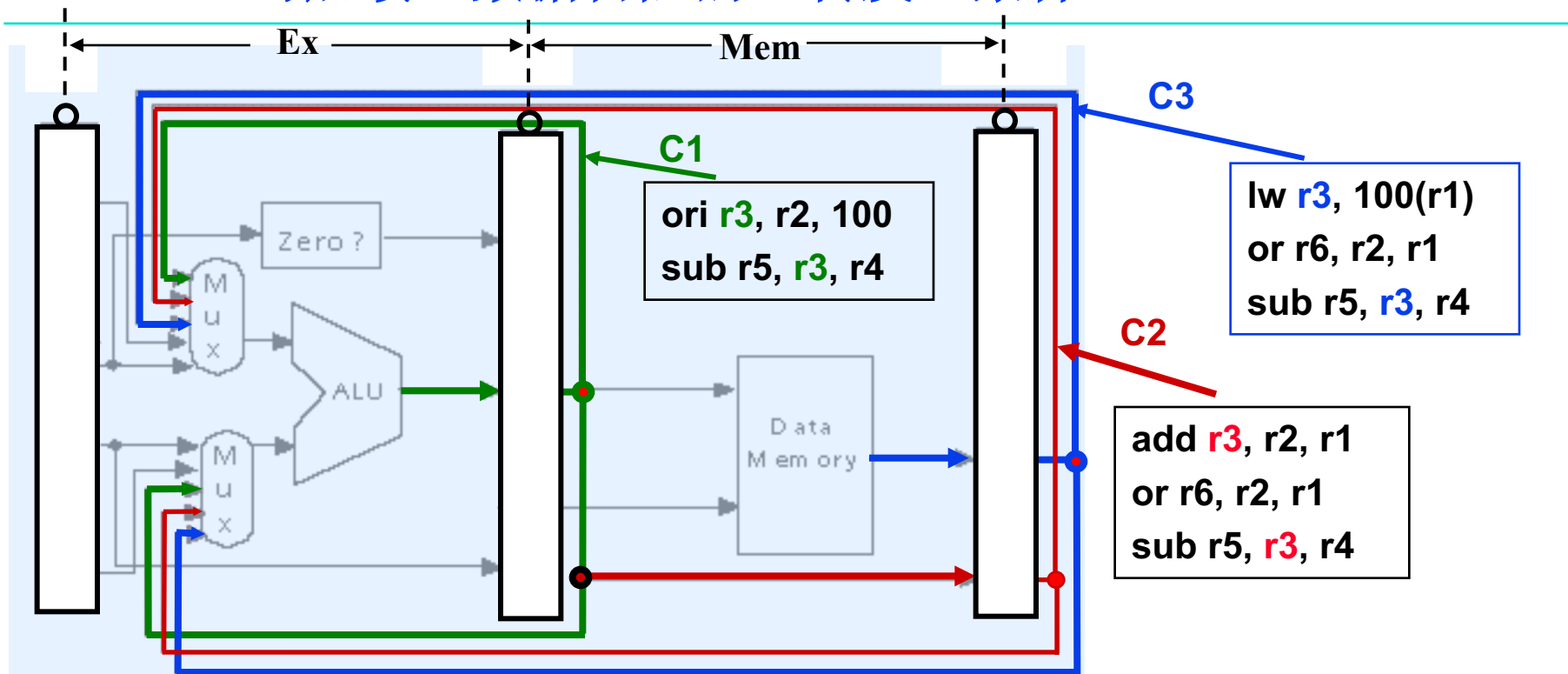


采用“转发”后仅第二条指令 **SUB r7,r1,r3** 不能按时执行！

发生“装入-使用数据冒险”时，需要对load后的指令阻塞一个时钟周期！

数据冒险处理最佳方案：“转发” + “Load-use阻塞”

RAW（写后读）数据冒险的“转发”条件



后面指令需用ALU输出结果

C1: 目寄是后一条指令的源寄

C2: 目寄是后第二条指令的源寄

(例如: R-Type后跟R- / lw / sw / beq等)

后面指令需用从DM读出的结果

C3: 目寄是后第二指令的源寄

(例如: load指令后跟R-Type / beq等)

用流水段寄存器来表示转发条件（C3以后考虑）

C1(a): EX/MEM. RegisterRd=ID/EX. RegisterRs

C1(b): EX/MEM. RegisterRd=ID/EX. RegisterRt

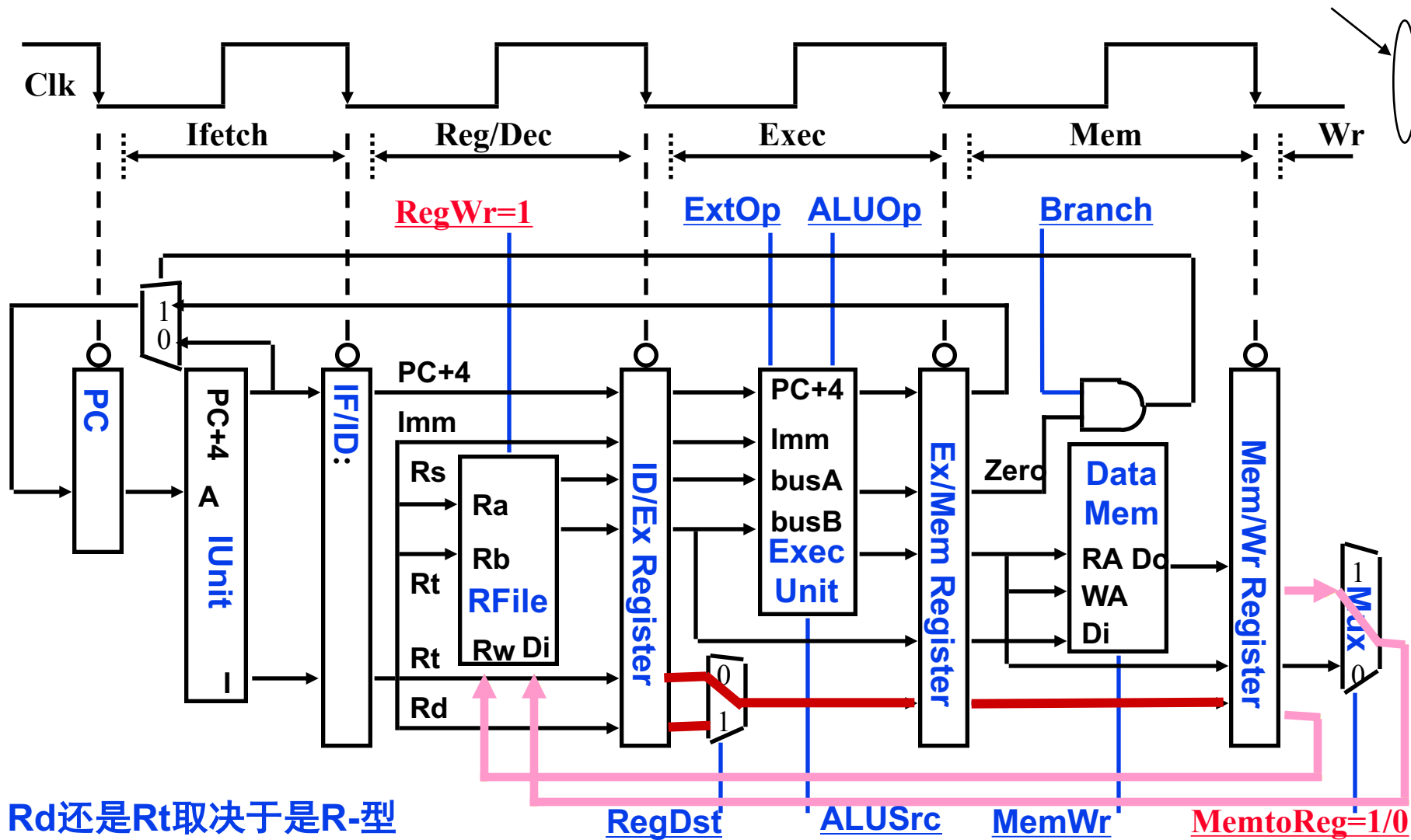
C2(a): MEM/WB. RegisterRd=ID/EX. RegisterRs

C2(b): MEM/WB. RegisterRd=ID/EX. RegisterRt

这里的RegisterRd是指目的寄存器

实际上是R-type的Rd 或 I-Type的Rt

指令的回写 (Write Back) 阶段



Rd还是Rt取决于是R-型指令，还是I-型指令！

若是beq指令会怎样？

```
beq r3, r2, 100
add r4, r3, r2
sub r5, r3, r2
```

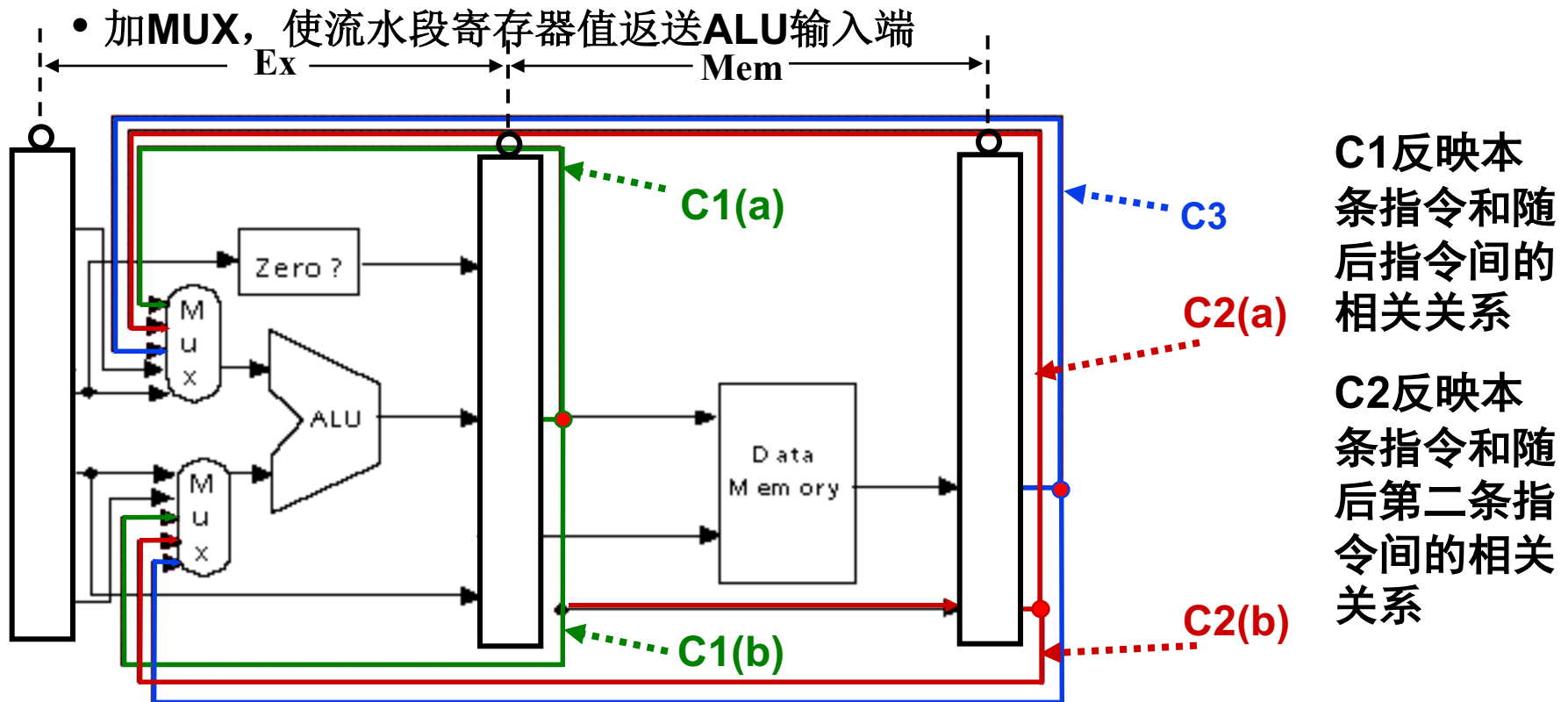
因为beq指令无需写结果，故不能进行转发！

转发条件的进一步完善

```
beq r3, r2, 100
add r4, r3, r2
sub r5, r3, r2
```

- 以下两种情况下，转发会发生错误
 - 指令的结果不写入目的寄存器Rd时
 - 例如，Beq指令只对rs和rt相减，不写结果到目的寄存器Rt
 - 即：EX / MEM 或 MEM / WB 流水段寄存器的RegWr信号为0
 - Rd等于\$0时
 - 例如，指令 sll \$0, \$1, 2 的转发结果为(R[\$1]<<2)，但实际上应该是0
- 因此，修改转发条件为：
 - C1(a): EX/MEM.RegWr
and EX/MEM. RegisterRd \neq 0
and EX/MEM. RegisterRd=ID/EX. RegisterRs
 - C1(b): EX/MEM.RegWr
and EX/MEM. RegisterRd \neq 0
and EX/MEM. RegisterRd=ID/EX. RegisterRt
 - C2(a): MEM/WB.RegWr
and MEM/WB. RegisterRd \neq 0
and MEM/WB. RegisterRd=ID/EX. RegisterRs
 - C2(b): MEM/WB.RegWr
and MEM/WB. RegisterRd \neq 0
and MEM/WB. RegisterRd=ID/EX. RegisterRt

转发路径和转发条件



红线和兰线可以合并，在原数据通路中确实是合并在一起的。记得吗？

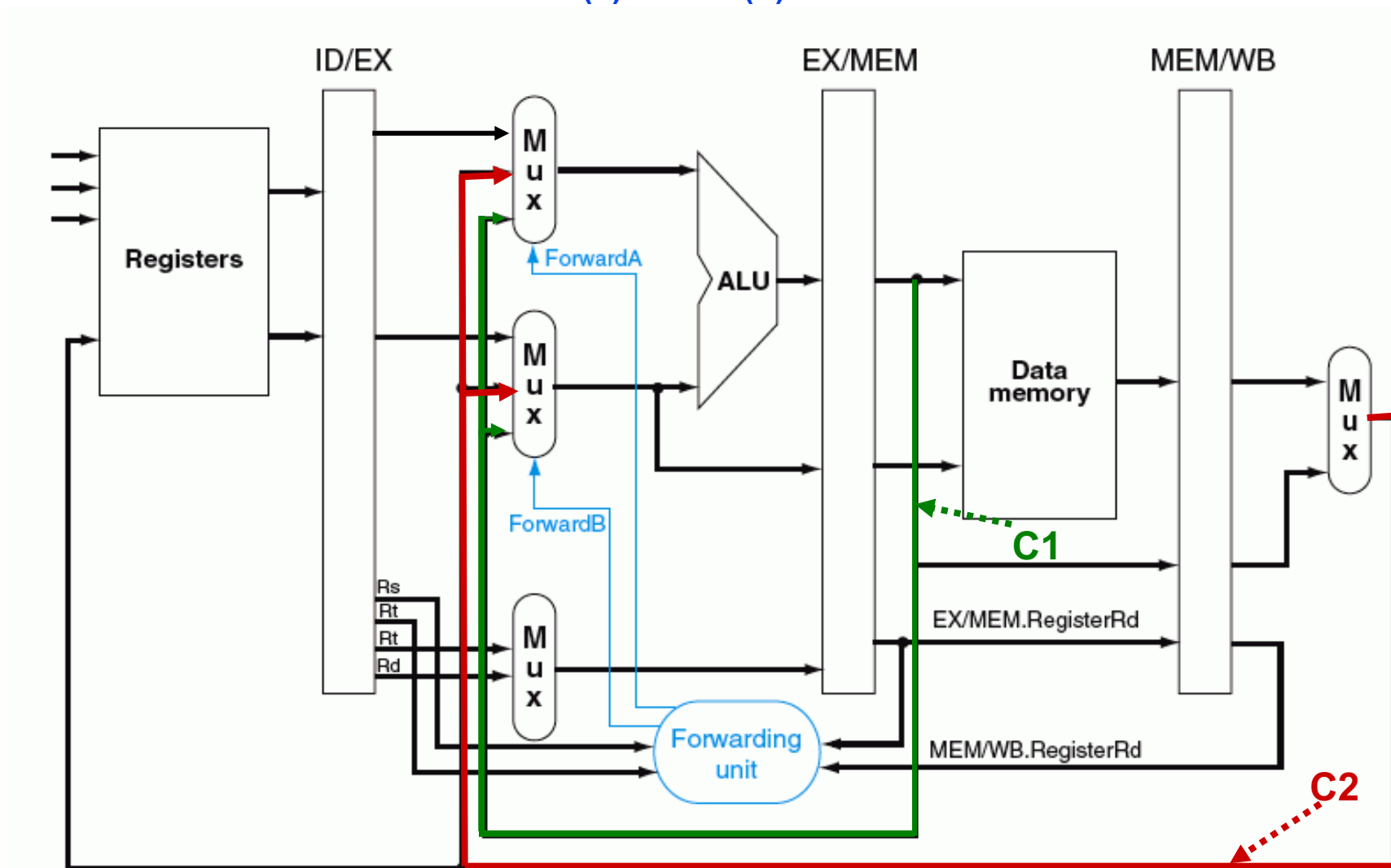
由一个二路选择器（控制端为MemtoReg）合并输出到寄存器堆！

所以，无需另有一个检测条件C3！ C3就是C2！

C1和C2分别反映哪两条指令的关系呢？

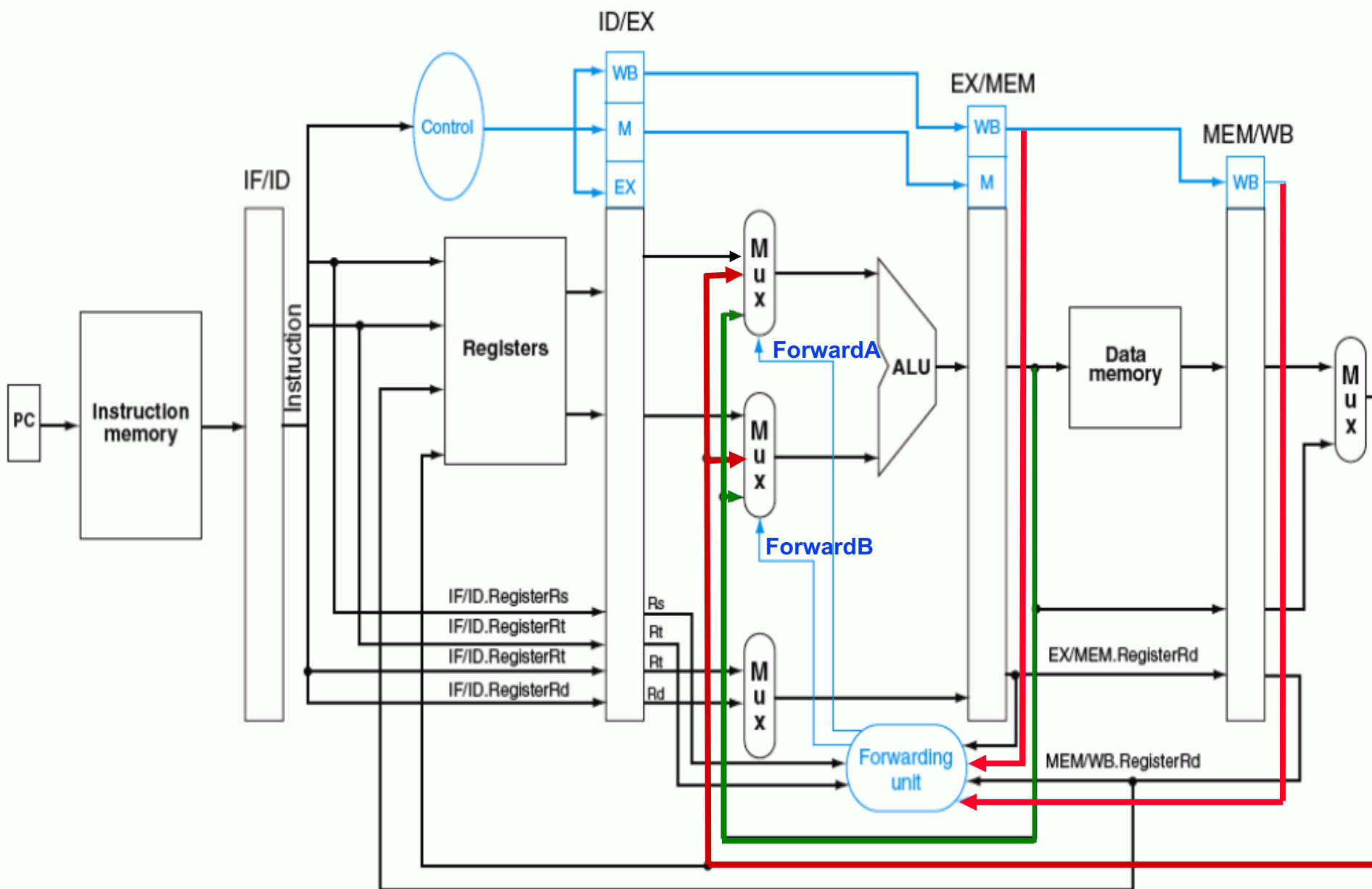
转发路径和转发条件

ForwardA (ForwardB) = $\begin{cases} 01 & \text{当 } c2(a)=1 \text{ 或 } C2(b)=1 \text{ 时} \\ 10 & \text{当 } c1(a)=1 \text{ 或 } C1(b)=1 \text{ 时} \end{cases}$



“转发检测”部件中缺何条件？ MEM/WB.RegWr=1、EX/MEM.RegWr=1

带转发的流水线数据通路



更加复杂的数据冒险问题

- 考察以下指令序列，采用前述转发条件会发生什么情况？

add \$1, \$1, \$2

add \$1, \$1, \$3

add \$1, \$1, \$4

.....

ForwardA (ForwardB) = $\begin{cases} 01 & \text{当 } c2(a)=1 \text{ or } c2(b)=1 \text{ 时} \\ 10 & \text{当 } c1(a)=1 \text{ or } c1(b)=1 \text{ 时} \end{cases}$

← 本条指令 对于左边的指令序列，C1和C2的值各是什么？

C1=C2=1，使得Forward信号取值不确定！

可能会使转发到第3条指令的操作数是第1条指令结果，而不是第2条指令的结果！

怎样改写“转发”检测条件：改C1还是改C2？ 应该让C1=1,C2=0!

需要改写“转发”条件C2(a)和C2(b)为：

C2(a)=MEM/WB.RegWrite and (MEM/WB.RegisterRd \neq 0) and (EX/MEM.RegisterRd \neq ID/EX.RegisterRs) and (MEM/WB.RegisterRd=ID/EX.RegisterRs)

C2(b)=MEM/WB.RegWrite and (MEM/WB.RegisterRd \neq 0) and (EX/MEM.RegisterRd \neq ID/EX.RegisterRt) and (MEM/WB.RegisterRd=ID/EX.RegisterRt)

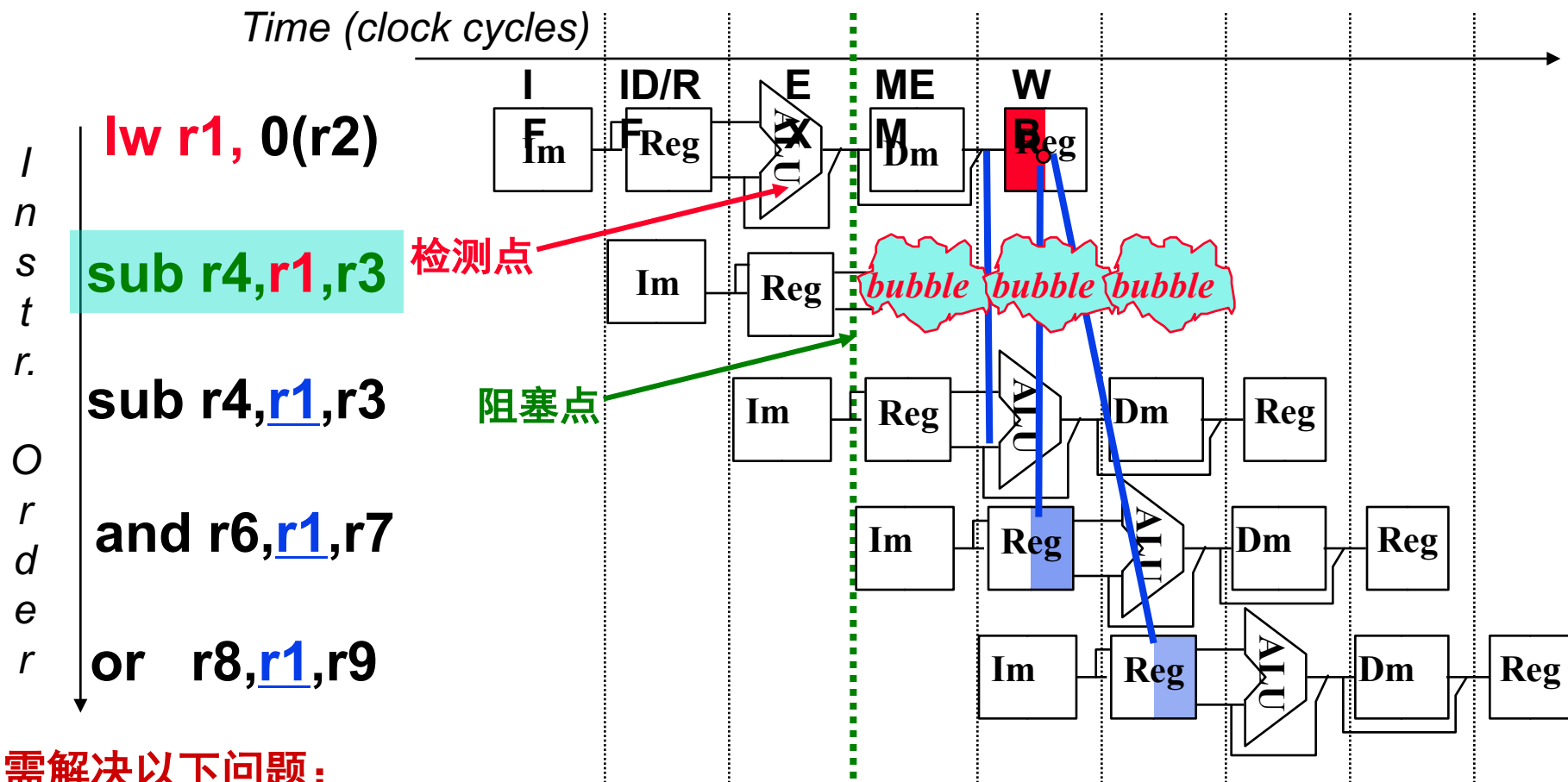
上述公式相当于加了一个条件限制：

如果本条指令源操作数和上条指令的目的寄存器一样，则不转发上上条指令的结果，而转发上条指令的结果（即：此时的C1=1而C2=0）

至此，解决了RAW数据冒险的“转发”处理

所有RAW数据冒险都能通过转发来解决吗？

Load-use Data Hazard (硬件阻塞方式)



需解决以下问题：

(1) 判断什么条件下需要阻塞

ID/EX.MemRead

and (ID/EX.RegisterRt=IF/ID.RegisterRs

or ID/EX.RegisterRt=IF/ID.RegisterRt)

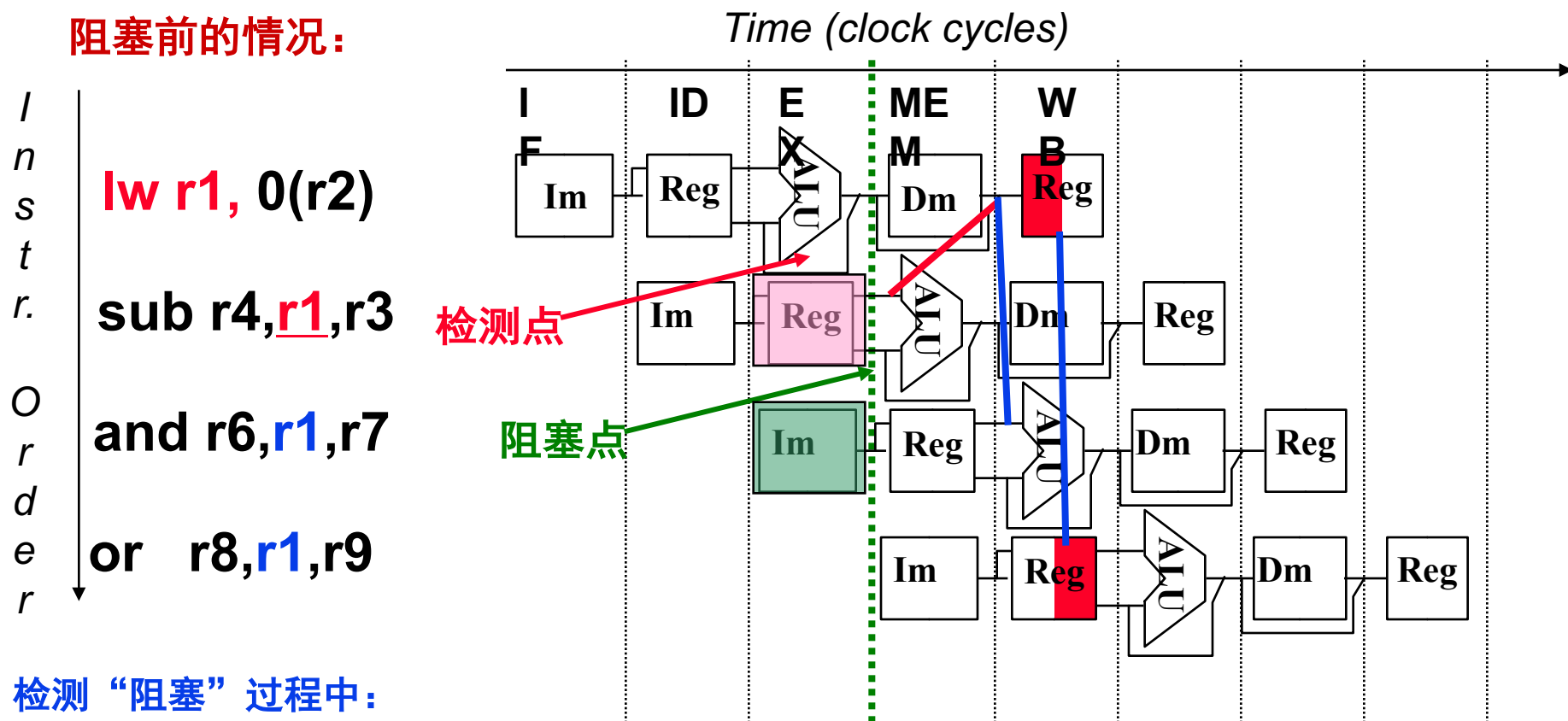
前面指令为Load 并且

前面指令的目的寄存器等于当前刚取出指令的源寄存器

(2) 如何修改数据通路来实现阻塞

Load-use Data Hazard (硬件阻塞方式)

阻塞前的情况:



检测“阻塞”过程中:

1) sub指令在IF/ID寄存器中, 并正被译码, 控制信号和Rs/Rt的值将被写到ID/EX段寄存器

2) and指令地址在PC中, 正被取出, 取出的指令将被写到IF/ID段寄存器中

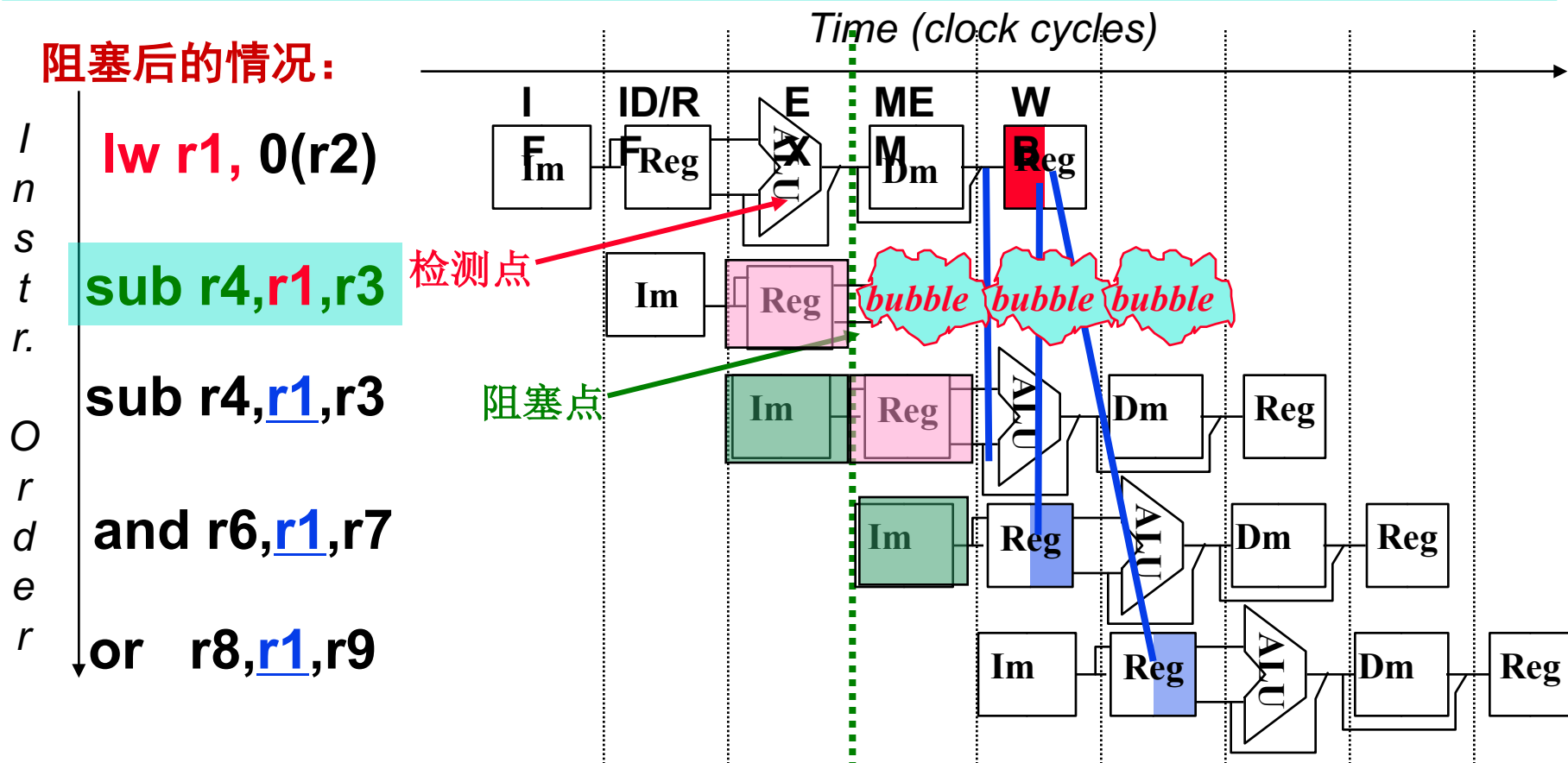
在阻塞点, 必须将上述两条指令的执行结果清除, 并延迟一个周期执行这两条指令

延迟一个周期执行后面指令, 相当于把阻塞点前面一个周期的状态再保持一个周期

lw指令还是继续正常执行下去

想想看, lw后面的指令如何做到继续保持状态?

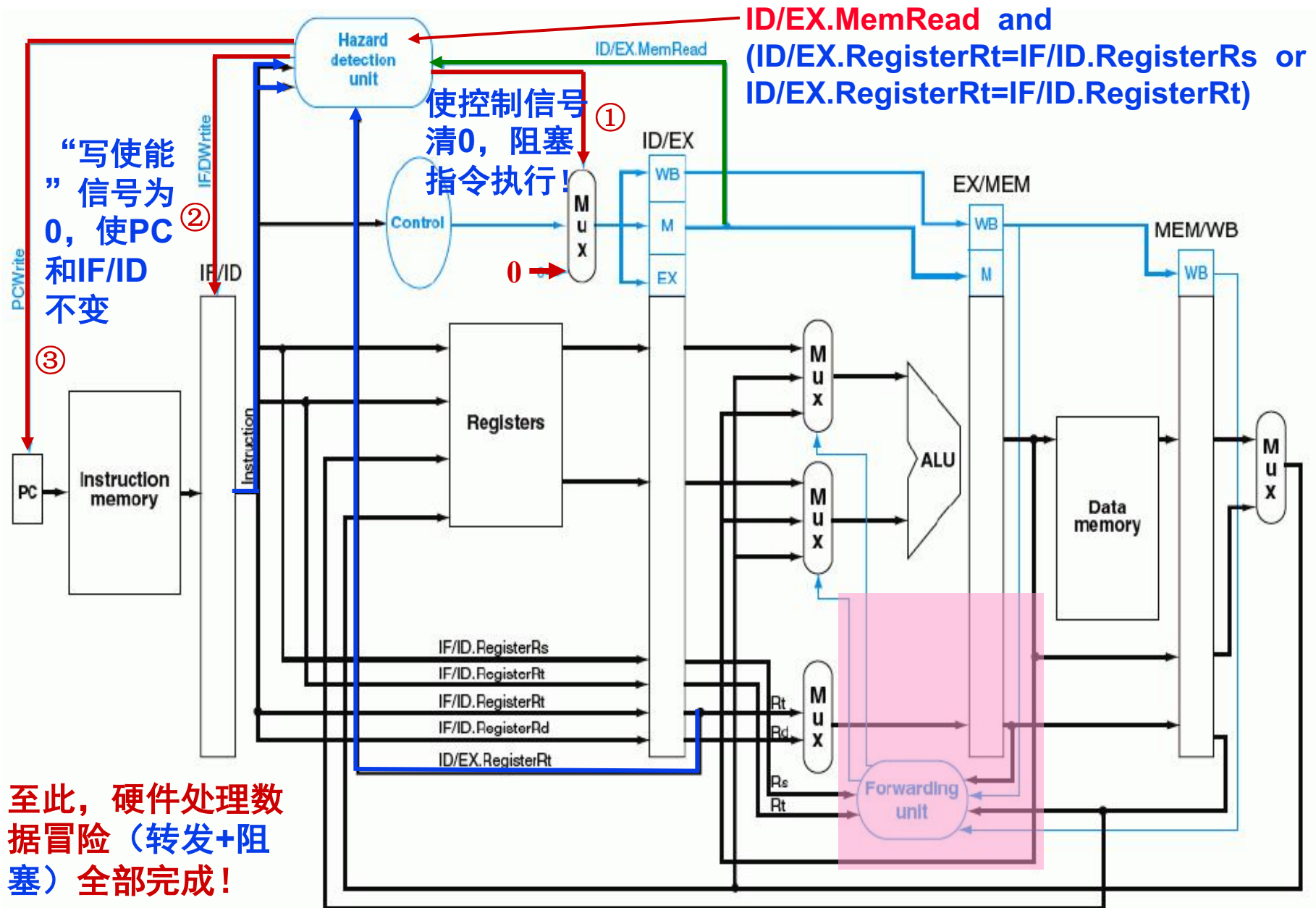
Load-use Data Hazard (硬件阻塞方式)



在阻塞点，将上述两条指令的执行结果清除，并延迟一个周期执行这两条指令

- ① 将ID/EX段寄存器中所有控制信号清0，插入一个“气泡”
- ② IF/ID寄存器中的信息不变（还是sub指令），sub指令重新译码执行
- ③ PC中的值不变（还是and指令地址），and指令重新被取出执行

带“转发”和“阻塞”检测的流水线数据通路



至此，硬件处理数据冒险（转发+阻塞）全部完成！

方案5：编译器进行指令顺序调整来解决数据冒险

以下源程序可生成两种不同的代码，优化的代码可避免Load阻塞

a = b + c;

d = e - f;

假定 a, b, c, d, e, f 在内存

编译器的优化很重要！

Slow code:

```
lw    $2, b
lw    $3, c
add   $1, $2, $3
sw    $1, a
lw    $5, e
lw    $6, f
sub   $4, $5, $6
sw    $4, d
```

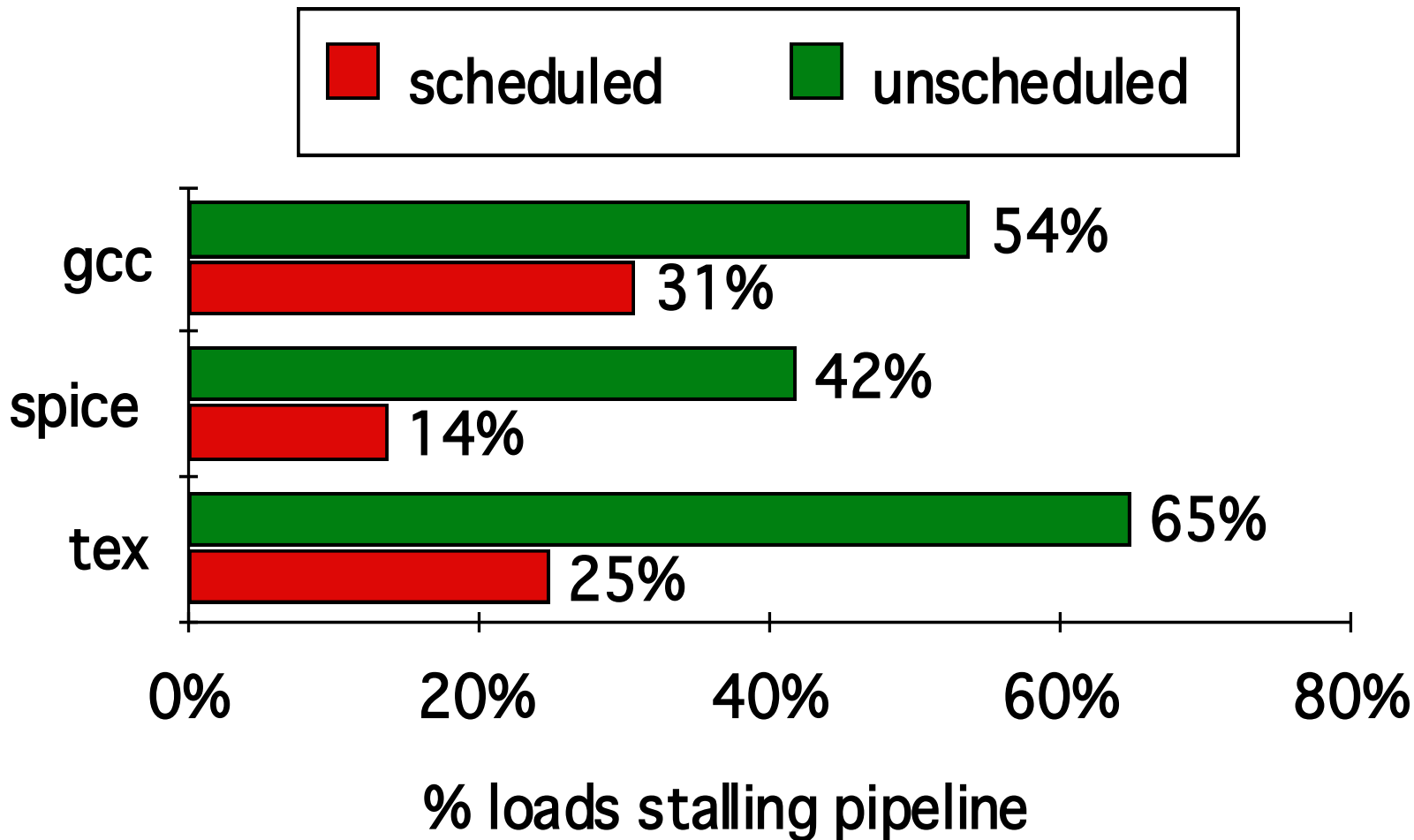
调整后

Fast code:

```
lw    $2, b
lw    $3, c
lw    $5, e
add   $1, $2, $3
lw    $6, f
sw    $1, a
sub   $4, $5, $6
sw    $4, d
```

如果硬件不支持阻塞处理的话，则编译器可以将顺序调整和插入NOP指令结合起来，在找不到可插入的指令时，插入NOP指令！

编译器优化以避免阻塞的情况调查:



由此可见，优化调度后load阻塞现象大约降低了1/2~1/3

[BACK](#)

Control Hazard的解决方法

- 方法1：硬件上阻塞（stall）分支指令后三条指令的执行
 - 使后面三条指令清0或其操作信号清0，以插入三条NOP指令
 - 方法2：软件上插入三条“NOP”指令
(以上两种方法的效率太低，需结合分支预测进行)
 - 方法3：分支预测（Predict）
 - 简单（静态）预测：
 - 总是预测条件不满足(not taken)，即：继续执行分支指令的后续指令
可加启发式规则：在特定情况下总是预测满足(taken)，其他情况总是预测不满足。如：循环顶部（底部）分支总是预测为不满足（满足）。
能达65%-85%的预测准确率
 - 动态预测：
 - 根据程序执行的历史情况进行动态预测调整，能达90%的预测准确率
 - 方法4：延迟分支（Delayed branch）（通过编译程序优化指令顺序！）
 - 把分支指令前面与分支指令无关的指令调到分支指令后执行，也称延迟转移
- 另一种控制冒险：异常或中断控制冒险的处理

简单（静态）分支预测方法

◦ 基本做法

- 总预测条件不满足(not taken), 即: 继续执行分支指令的后续指令
可加启发式规则:
在特定情况下总是预测满足(taken), 其他情况总是预测不满足
- 预测失败时, 需把流水线中三条错误预测指令丢弃掉
 - 将被丢弃指令的控制信号值或指令设置为0

(注: 涉及到当时在IF、ID和EX三个阶段的指令)

◦ 性能

- 如果转移概率是50%, 则预测正确率仅有50%

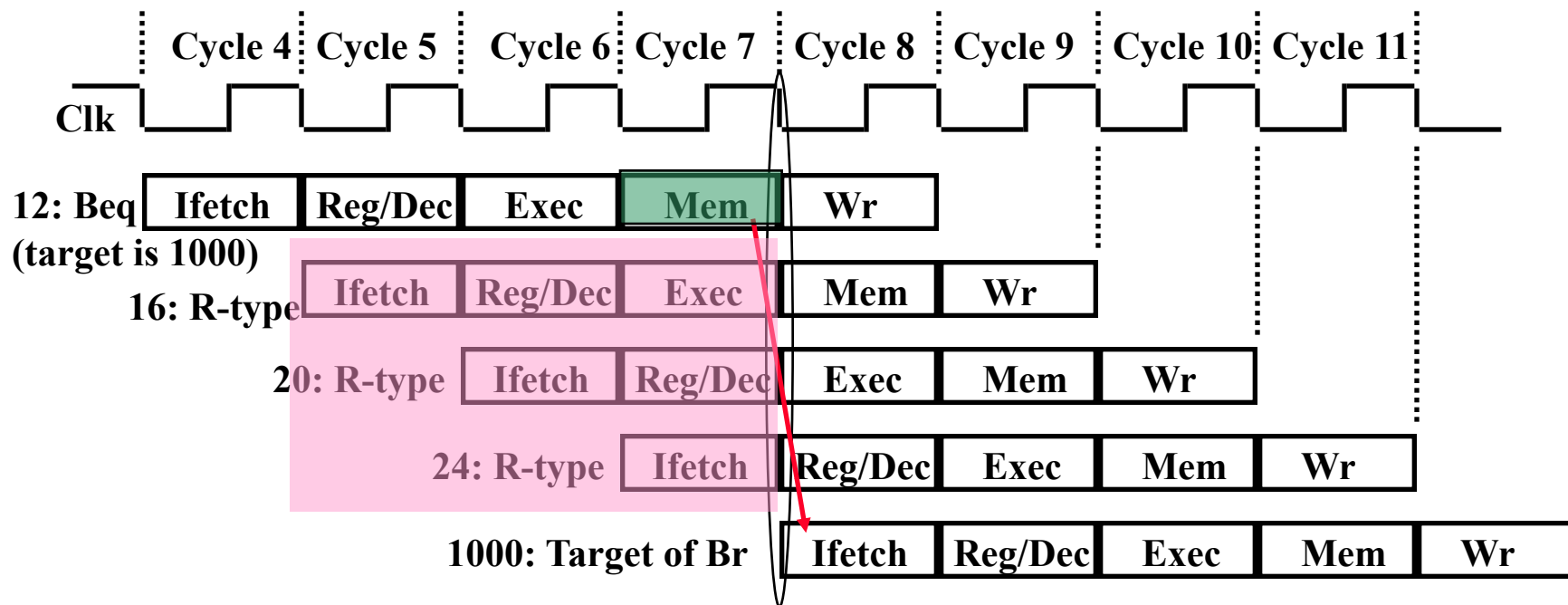
◦ 预测错误的代价

- 预测错误的代价与何时能确定是否转移有关。越早确定代价越少
- 是否可以把“是否转移”的确定工作提前, 而不等MEM阶段才确定
[?]可以!

那最早可以提前到哪个阶段呢?

SKIP

复习: Control Hazard现象



- 虽然Beq指令在第四周期取出，但：
 - “是否转移”在Mem阶段确定，目标地址在第七周期才被送到PC输入端
 - 第八周期才取出目标地址处的指令执行
- 结果：在取目标指令之前，已有三条指令被取出，取错了三条指令！
- 发生转移时，要在流水线中清除Beq后面的三条指令，分别在EXE、ID、IF段中
- 延迟损失时间片C：发生转移时，给流水线带来的延迟损失

[BACK](#)

这里 C=3

简单（静态）分支预测方法

- 缩短分支延迟，减少错误预测代价
 - 可以将“转移地址计算”和“分支条件判断”操作调整到ID阶段来缩短延迟
 - 将转移地址生成从MEM阶段移到ID阶段，可以吗？为什么？
(是可能的：IF/ID流水段寄存器中已经有PC的值和立即数)
 - 将“判0”操作从EX阶段移到ID阶段，可以吗？为什么？
(用逻辑运算(如，先按位异或，再结果各位相或)来直接比较Rs和Rt的值)
(简单判断用逻辑运算，复杂判断可以用专门指令生成条件码)
(许多条件判断都很简单)
- 预测错误的检测和处理（称为“冲刷、冲洗” -- Flush）
 - 当Branch=1并且Zero=1时，发生转移（taken）
 - 增加控制信号：IF.Flush=Branch and Zero，取值为1时，说明预测失败
 - 预测失败(条件满足)时，完成以下两件事（延迟损失时间片C=1时）：
 - ① 将转移目标地址->PC
 - ② 清除IF段中取出的指令，即：将IF/ID中的指令字清0，转变为nop指令

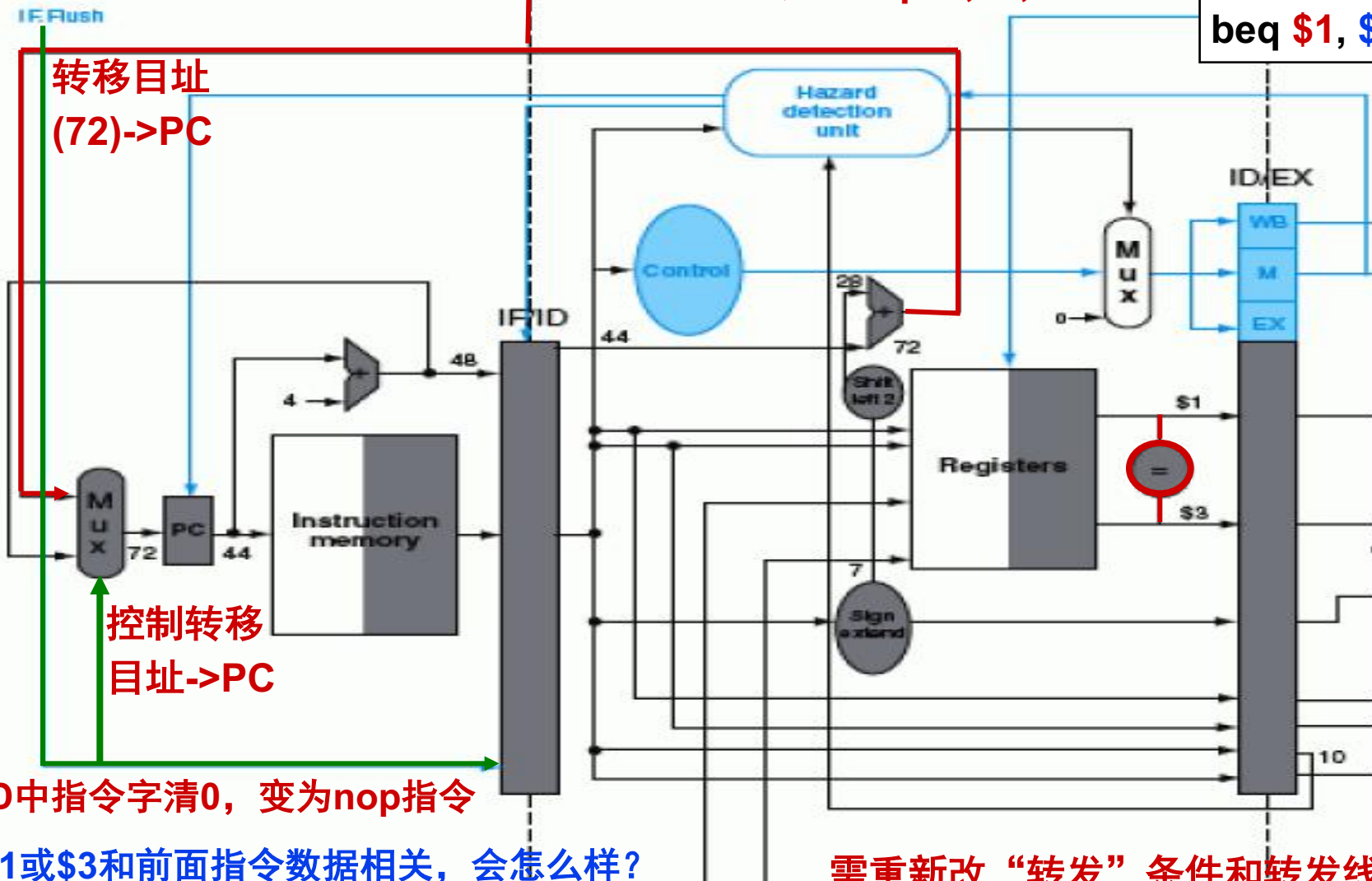
原来要清除三条指令，调整后只需要清除一条指令，因而只延迟一个时钟周期，每次预测错误减少了两个周期的代价！
即：这里 C=1

带静态分支预测处理的数据通路

IF.Flush=Branch and Zero

40#指令 “beq \$1,\$3,7” ID段

```
sub $3, $5, $1
add $1, $5, $2
beq $1, $3, 7
```



若\$1或\$3和前面指令数据相关, 会怎么样?

- 上上条指令的EXE段结果可转发回来进行判断
- 上条指令的EXE段结果来不及转发回来, 引起1次阻塞!

需重新改“转发”条件和转发线路!
作业中有相应的练习

[BACK](#)

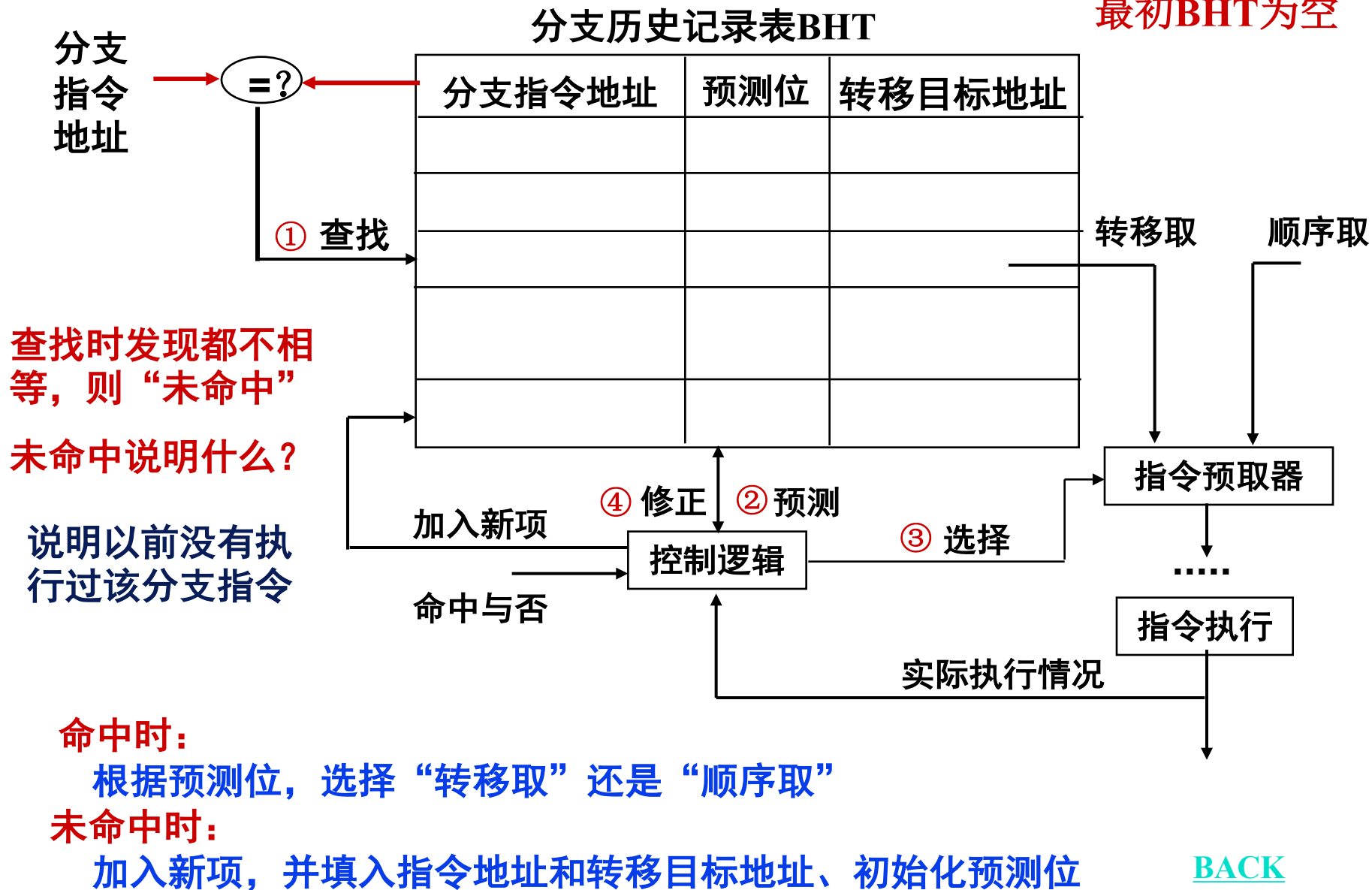
动态分支预测方法

- 简单的静态分支预测方法的预测成功率不高，应考虑动态预测
- 动态预测基本思想：
 - 利用最近转移发生的情况，来预测下一次可能转移还是不转移
 - 根据实际情况来调整预测
 - 转移发生的历史情况记录在BHT中（有多个不同的名称）
 - 分支历史记录表BHT（Branch History Table）
 - 分支预测缓冲BPB（Branch Prediction Buffer）
 - 分支目标缓冲BTB（Branch Target Buffer）
 - 每个表项由分支指令地址低位作索引，故在IF阶段就可以取到预测位
 - 低位地址相同的分支指令共享一个表项，所以，可能取的是其他分支指令的预测位。会不会有问题？
 - 由于仅用于预测，所以不影响执行结果

现在几乎所有的处理器都采用动态预测（dynamic predictor）

分支历史记录表BHT（或BTB、BPB）

最初BHT为空

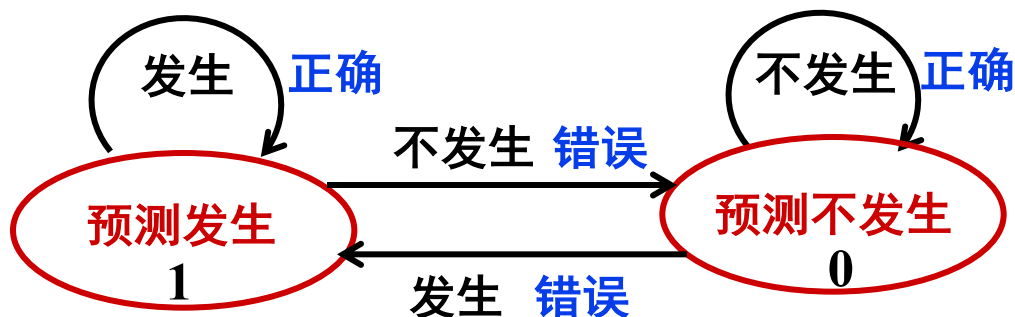


动态预测基本方法

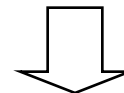
- 采用一位预测位：总是按上次实际发生的情况来预测下次
 - 1表示最近一次发生过转移（taken），0表示未发生（not taken）
 - 预测时，若为1，则预测下次taken，若为0，则预测下次not taken
 - 实际执行时，若预测错，则该位取反，否则，该位不变
 - 可用一个简单的预测状态图表示
 - 缺点：当连续两次的分支情况发生改变时，预测错误
 - 例如，循环迭代分支时，第一次和最后一次会发生预测错误，因为循环的第一次和最后一次都会改变分支情况，而在循环中间的各次总是会发生分支，按上次的实际情况预测时，都不会错。
- 采用二位预测位
 - 用2位组合四种情况来表示预测和实际转移情况
 - 按照预测状态图进行预测和调整
 - 在连续两次分支发生不同时，只会有一次预测错误

采用较多的是二位或二位以上预测位。如：Pentium 4的BTB2采用4位预测位

一位预测状态图



```
Loop:  g = g + A[i];
        i = i + j;
        if (i != h) go to Loop:
Assuming variables g, h, i, j ~
$1, $2, $3, $4 and base address
of array is in $5
```



```
Loop: add $7, $3, $3      ; i*2
      add $7, $7, $7      ; i*4
      add $7, $7, $5
      lw $6, 0($7)        ; $6=A[i]
      add $1, $1, $6       ; g= g+A[i]
      add $3, $3, $4
      bne $3, $2, Loop
      ... ..
```

◦ 指令预取时，按照预测位读取相应分支的指令

- 预测发生时，选择“转移取”
- 预测不发生时，选择“顺序取”

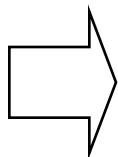
◦ 指令执行时，按实际执行结果修改预测位

- 对照状态转换图来进行修改
- 例如：对于一个循环分支
 - 若初始状态为0(再次循环时为0)，则第一次和最后一次都错
 - 若初始状态为1，则只有最后一次会错。(再次循环时又改为0，还是有两次错)

即：只要本次和上次的发生情况不同，就会出现一次预测错误。

举例：双重循环的一位动态预测

```
into sum (int N)
{
  int i, j, sum=0;
  for (i=0; i < N; i++)
    for (j=0; j < N; j++)
      sum=sum+1;
  return sum;
}
```



```
... ..
Loop-i: beq $t1,$a0, exit-i  # 若( i=N)则跳出外循环
        add $t2, $zero, $zero #j=0
Loop-j: beq $t2, $a0, exit-j  # 若(j=N)则跳出内循环
        addi $t2, $t2, 1      # j=j+1
        addi $t0, $t0, 1      #sum=sum+1
        j Loop-j
exit-j:  addi $t1, $t1, 1      # i=i +1
        j Loop-i
exit-i:  ... ..
```

外循环中的分支指令共执行 $N+1$ 次，
内循环中的分支指令共执行 $N \times (N+1)$ 次。

$N=10$ ，分别90.9%和82.7%

$N=100$ ，分别99%和98%

若预测位初始为0，则外循环只有最后一次预测错误；跳出内循环时预测位变为1，再进入内循环时，第一次总是预测错误，并且任何一次循环的最后一次总是预测错误，因此，总共有 $1+2 \times (N-1)$ 次预测错误（第一次循环有1次预测错，后面 $(N-1)$ 次循环每次有2次预测错）。

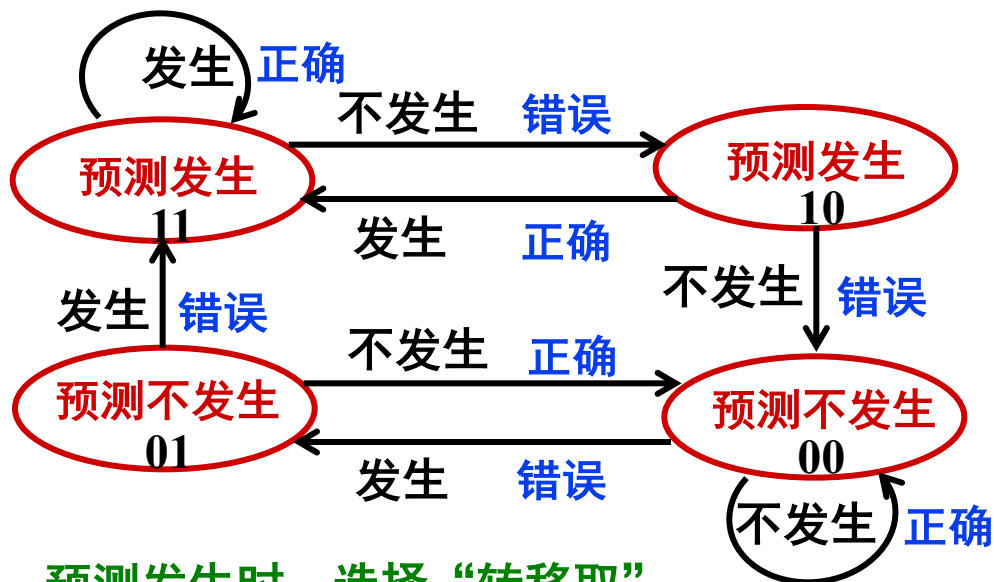
N 越大预测准确率越高！

动态预测基本方法

- 采用一位预测位：总是按上次实际发生的情况来预测下次
 - 1表示最近一次发生过转移（taken），0表示未发生（not taken）
 - 预测时，若为1，则预测下次taken，若为0，则预测下次not taken
 - 实际执行时，若预测错，则该位取反，否则，该位不变
 - 可用一个简单的[预测状态图](#)表示
 - 缺点：当连续两次的分支情况发生改变时，预测错误
 - 例如，循环迭代分支时，第一次和最后一次会发生预测错误，因为循环的第一次和最后一次都会改变分支情况，而在循环中间的各次总是会发生分支，按上次的实际情况预测时，都不会错。
- 采用二位预测位
 - 用2位组合四种情况来表示预测和实际转移情况
 - 按照[预测状态图](#)进行预测和调整
 - 在连续两次分支发生不同时，只会有一次预测错误

采用较多的是二位或二位以上预测位。如：Pentium 4的BTB2采用4位预测位

两位预测状态图



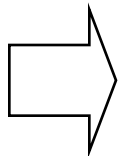
```
Loop: add $7, $3, $3    ; i*2
      add $7, $7, $7    ; i*4
      add $7, $7, $5
      lw $6, 0($7)      ; $6=A[i]
      add $1, $1, $6    ; g= g+A[i]
      add $3, $3, $4
      bne $3, $2, Loop
      ... ..
```

预测发生时，选择“转移取”
预测不发生时，选择“顺序取”

- 基本思想：只有两次预测错误才改变预测方向
 - 11状态时预测发生（强转移），实际不发生时，转到状态10（弱转移），下次仍预测为发生，如果再次预测错误（实际不发生），才使下次预测调整为不发生00
- 好处：连续两次发生不同的分支情况时，会预测正确
 - 例如，对于循环分支的预测（假定预测初始位为11）
 - 第一次：初始态为11（再次进入循环时为10），预测发生，实际也发生，正确
 - 中间：状态为“11”，预测发生，实际也发生，正确
 - 最后一次：状态为“11”，预测发生，但实际不发生，错

举例：双重循环的两位动态预测

```
into sum (int N)
{
  int i, j, sum=0;
  for (i=0; i < N; i++)
    for (j=0; j < N; j++)
      sum=sum+1;
  return sum;
}
```



```
... ..
Loop-i: beq $t1,$a0, exit-i  # 若( i=N)则跳出外循环
        add $t2, $zero, $zero #j=0
Loop-j: beq $t2, $a0, exit-j  # 若(j=N)则跳出内循环
        addi $t2, $t2, 1      # j=j+1
        addi $t0, $t0, 1      #sum=sum+1
        j Loop-j
exit-j:  addi $t1, $t1, 1      # i=i +1
        j Loop-i
exit-i:  ... ..
```

外循环中的分支指令共执行 $N+1$ 次，
内循环中的分支指令共执行 $N \times (N+1)$ 次。

$N=10$ ，分别90.9%和90.9%

$N=100$ ，分别99%和99%

若预测位初始为00，外循环只有最后一次预测错误；跳出内循环时预测位变为01，再进入内循环时，第一次预测正确，只有最后一次预测错误，因此，总共有 N 次预测错误。

N 越大准确率越高！

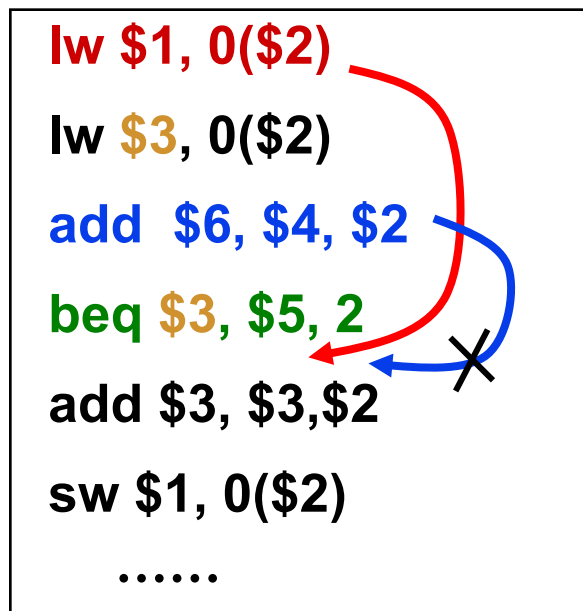
[BACK](#)

分支延迟时间片的调度

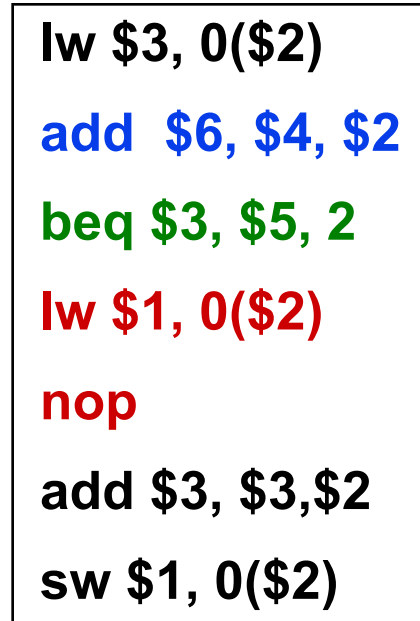
- 属于静态调度技术，由编译程序重排指令顺序来实现
- 基本思想：把分支指令前面的与分支指令无关的指令调到分支指令后面执行，以填充延迟时间片（也称分支延迟槽Branch Delay slot），不够时用nop填充

举例：如何对以下程序段进行分支延迟调度？（假定时间片为2）

若分支条件判断和目标地址计算提前到ID阶段，则分支延迟时间片减少为1



调度后



调度后可能带来其他问题：
产生新的load-use数据冒险

调度后，无需在硬件线路中阻塞branch指令后面指令的执行

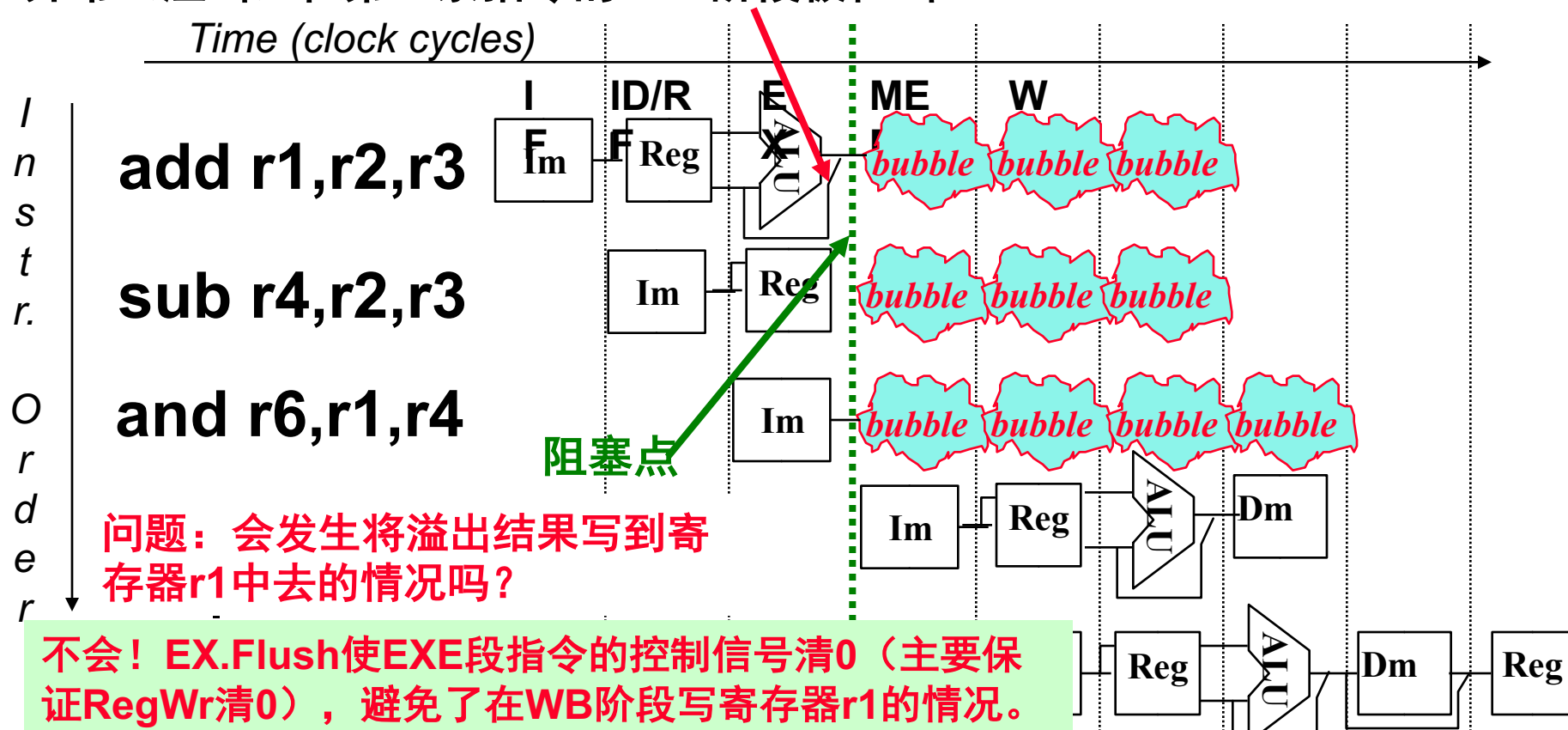
[BACK](#)

另一种控制冒险：异常和中断

- 异常和中断会改变程序的执行流程
- 某条指令发现异常时，后面多条指令已被取到流水线中正在执行
 - 例如ALU指令发现“溢出”时，已经到EX阶段结束了，此时，它后面已有两条指令进入流水线了
- 流水线数据通路如何处理异常？(举例说明)
 - 假设指令add r1,r2,r3产生了溢出
(记住：MIPS异常处理程序的首地址为0x8000 0180)
 - 处理思路：
 - ✓ 清除add指令以及后面的所有已在流水线中的指令
 - ✓ 关中断（将中断允许触发器清0）
 - ✓ 保存PC或PC-4（断点） 到 EPC
 - ✓ 0x8000 0180送PC（从0x8000 0180处开始取指令）

异常的处理

- 异常（溢出）在第一条指令的EXE阶段被检出



异常处理：

IF.Flush使IF段指令在IF/ID寄存器中清为0，变成nop指令

ID.Flush与数据冒险阻塞检测信号相或(or)后，使ID段指令的控制信号清0

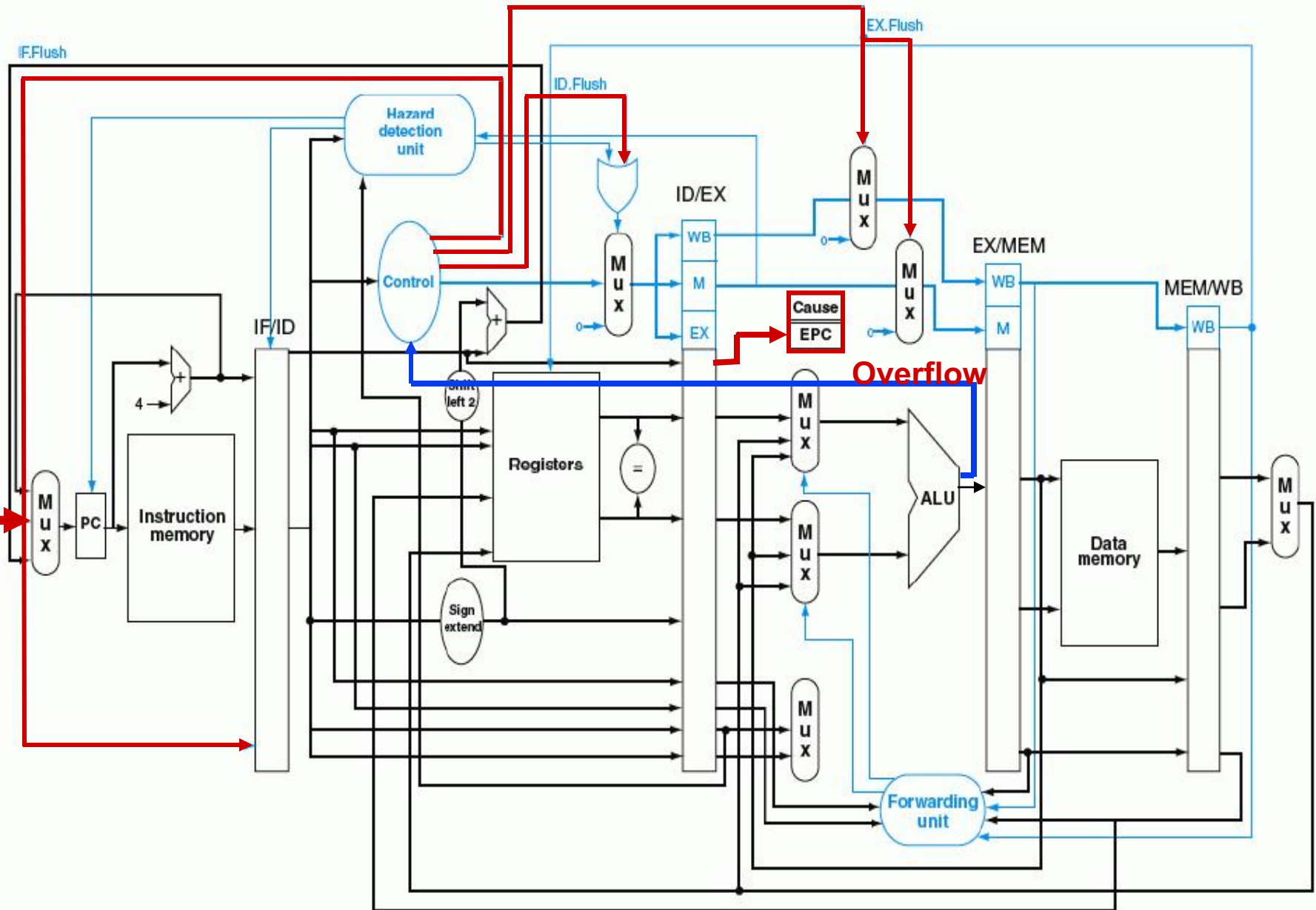
EX.Flush使EX段指令的控制信号清0

关中断，并将断点（可能是PC、可能是PC-4）保存到EPC中

将0x8000 0180作为PC的一个输入，并控制PC输入端的多路选择器

带异常处理的流水线数据通路

80000180H



流水线方式下的异常处理的难点问题

- 流水线中同时有5条指令，到底是哪一条发生异常？
 - 根据异常发生的流水段可确定是哪条指令，因为各类异常发生的流水段不同
 - ✓ “溢出”在EXE段检出
 - ✓ “无效指令”在ID段检出
 - ✓ “除数为0”在ID段检出
 - ✓ “无效指令地址”在IF段检出
 - ✓ “无效数据地址”在Load/Store指令的EXE段检出
- 外部中断与特定指令无关，如何确定处理点？
 - 可在IF段或WB段中进行中断查询，需要保证当前WB段的指令能正确完成，并在有中断发生时，确保下个时钟开始执行中断服务程序
- 检测到异常时，指令已经取出多条，当前PC的值已不是断点，怎么办？
 - 指令地址存放在流水段R，可把这个地址送到EPC保存，以实现精确中断
非精确中断不能提供准确的断点，而由操作系统来确定哪条指令发生了异常
- 一个时钟周期内可能有多个异常，该先处理哪个？
 - 异常：检出异常后，其原因存到专门寄存器中并流到最后阶段处理，使前面指令的异常优先级高于后面指令
 - 中断：在中断查询程序或中断优先级排队电路中按顺序查询
- 系统中只有一个EPC，多个中断发生时，一个EPC不够放多个断点，怎么办？
 - 总是把优先级最高的送到EPC中
- 在异常处理过程中，又发生了新的异常或中断，怎么办？
 - 利用中断屏蔽和中断嵌套机制来处理

后面三个问题中有关中断的问题在第九章中详细介绍！

三种处理器实现方式的比较

◦ 单周期、多周期、流水线三种方式比较

假设各主要功能单元的操作时间为：

- 存储单元：200ps
- ALU和加法器：100ps
- 寄存器堆（读 / 写）：50ps

假设MUX、控制单元、PC、扩展器和传输线路都没有延迟，指令组成为：
25%取数、10%存数、52%ALU、11%分支、2%跳转，则下面实现方式中，
哪个更快？快多少？

- (1) 单周期：每条指令在一个固定长度的时钟周期内完成
- (2) 多周期：每类指令时钟数为取数-5，存数-4，ALU-4，分支-3，跳转-3
- (3) 流水线：每条指令分取指令、取数/译码、执行、存储器存取、写回五阶段
(假定没有结构冒险，数据冒险采用转发处理，分支延迟槽为1，预测准确率为75%；不考虑异常、中断和访存缺失引起的流水线冒险)

三种处理器实现方式的比较

解：CPU执行时间 = 指令条数 x CPI x 时钟周期长度

三种方式指令条数都一样，所以只要比较CPI和时钟周期长度即可。

Instruction class	Functional units used by the instruction class				
	Instruction fetch	Register access	ALU	Register access	
R-type	Instruction fetch	Register access	ALU	Register access	
Load word	Instruction fetch	Register access	ALU	Memory access	Register access
Store word	Instruction fetch	Register access	ALU	Memory access	
Branch	Instruction fetch	Register access	ALU		
Jump	Instruction fetch				

各指令类型要求的时间长度为：

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	50	100	0	50	400 ps
Load word	200	50	100	200	50	600 ps
Store word	200	50	100	200		550 ps
Branch	200	50	100	0		350 ps
Jump	200					200 ps

三种处理器实现方式的比较

对于单周期方式：

时钟周期将由最长指令来决定，应该是load指令，为600ps

所以，N条指令的执行时间为 **$600N(\text{ps})$**

对于多周期方式：

时钟周期将取功能部件最长所需时间，应该是存取操作，为200ps

根据各类指令的频度，计算平均时钟周期数为：

CPU时钟周期= $5 \times 25\% + 4 \times 10\% + 4 \times 52\% + 3 \times 11\% + 3 \times 2\% = 4.12$

所以，N条指令的执行时间为 **$4.12 \times 200 \times N = 824N(\text{ps})$**

对于流水线方式：

Load指令：当发生Load-use依赖时，执行时间为2个时钟，否则1个时钟，
故平均执行时间为1.5个时钟；

Store、ALU指令：1个时钟；

Branch指令：预测成功时，1个时钟，预测错误时，2个时钟，
所以：平均约为： $.75 \times 1 + .25 \times 2 = 1.25$ 个；

Jump指令：2个时钟（总要等到译码阶段结束才能得到转移地址）

平均CPI为： $1.5 \times 25\% + 1 \times 10\% + 1 \times 52\% + 1.25 \times 11\% + 2 \times 2\% = 1.17$

所以，N条指令的执行时间为 **$1.17 \times 200 \times N = 234N(\text{ps})$**

第二讲小结

- 流水线冒险的几种类型：
 - 资源冲突、数据相关、控制相关（改变指令流的执行方向）
- 数据冒险的现象和对策
 - 数据冒险的种类
 - 相关的数据是ALU结果，可以通过转发解决
 - 相关的数据是DM读出的内容，随后的指令被阻塞一个时钟
 - 数据冒险和转发
 - 转发检测 / 转发控制
 - 数据冒险和阻塞
 - 阻塞检测 / 阻塞控制
- 控制冒险的现象和对策
 - 静态分支预测技术
 - 缩短分支延迟技术
 - 动态分支预测技术
- 异常和中断是一种特殊的控制冒险
- 访存缺失（Cache缺失、TLB缺失、缺页）会引起流水线阻塞

第三讲 高级流水线技术

- 高性能流水线 - 指令级并行(ILP)技术
 - 超流水线
 - 多发射流水线
 - 静态多发射 (VLIW处理器 + 编译器静态调度)
 - 动态多发射 (超标量处理器 + 动态流水线调度)
- 静态多发射 (VLIW-超长指令字)
 - 编译器静态推测完成“指令打包”和“冒险处理”
 - MIPS 2-发射流水线数据通路
 - 循环展开指令调度
 - IA-64的EPIC技术
- 动态多发射
 - 动态多发射流水线的通用模型
 - 动态多发射流水线的执行模式
 - 按序发射、按序完成
 - 按序发射、乱序完成
 - 乱序发射、乱序完成
 - Pentium 4 动态多发射流水线
 - 超流水、超标量、动态调度、乱序发射、乱序完成

提高性能措施—实现指令级并行

- 实现指令流内部的并行流水线称为指令级并行（ILP）
- 有两种指令级并行策略

N段流水线说明一个时钟周期内最多有几条指令同时并行执行？

N条！故N越大并行度越高！

- **超流水线（Super-pipelining）**

- 级数更多的流水线 **CPI = ?** **CPI = 1**
- 理想情况下，流水线的加速比与流水段的数目成正比

（即：理想情况下，流水段越多，时钟周期越短，指令吞吐率越高）

但是，它会增加开销，且是有极限的！可以怎样突破极限呢？

- **多发射流水线（Multiple issue pipelining）**

- 多条指令(如整数运算、浮点运算、装入/存储等) 同时启动并独立运行
- 前提：有多个执行部件。如：定点、浮点、乘/除、取数/存数部件等

- 结果：能达到小于1的CPI，定义CPI的倒数为IPC

（例如：理想的四路多发射流水线的IPC为4）

- 需要实现以下两个主要任务 一条流水线变成多条流水线！
 - 指令打包：分析每个周期发射多少条？哪些指令可以同时发射？
 - 冒险处理：由编译器静态调整指令或在运行时由硬件处理
- 实现上述两个主要任务的基础—**推测技术**
- 两种实现方法
 - **静态多发射**：由编译器在编译时静态完成指令打包和冒险处理
 - **动态多发射**：由硬件在执行时动态完成指令打包和冒险处理

增加的开销体现在哪里？

流水段寄存器！

实现多发射技术的基础—推测

- 推测技术：由编译器或处理器猜测指令执行结果，并以此来调整指令执行顺序，使指令的执行能达到最大可能的并行
 - 指令打包的决策依赖于“推测”的结果
 - 可根据指令间的相关性来进行推测
 - 与前面指令不相关的指令可以提前执行
 - 可对分支指令进行推测
 - 可提前执行分支目标处的指令
 - 预测仅是“猜测”，可能推测错误，故需有推测错误检测和回滚机制
 - 推测错误时，会增加额外开销
 - 有“软件推测”和“硬件推测”两种
 - 软件推测：编译器通过推测来静态重排指令（一定要正确！）
 - 硬件推测：处理器在运行时通过推测来动态调度指令

静态多发射处理器

- 由编译器在编译时进行相关性分析和静态分支预测，以静态完成“指令打包”和“冒险处理”
 - 指令打包（将同时发射的多条指令合并到一个长指令中）
 - 将同一个时钟周期内发射的多个指令看成是一条多个操作的长指令，称为一个“发射包”
 - “静态多发射”也被称为“超长指令字”（VLIW-Very Long Instruction Word），采用这种技术的处理器被称为VLIW处理器
 - 在同一个周期内发射的指令类型是受限制的（举例:干洗/水洗）
例如，只能是一条ALU指令/分支指令、一条Load/Store指令
 - IA-64采用这种方法，Intel称其为EPIC（Explicitly Parallel Instruction Computer—显式并行指令计算机）
 - 冒险处理（主要是数据冒险和控制冒险）
 - 做法1：完全由编译器通过代码调度和插入nop指令来消除所有冒险，无需硬件实现冒险检测和流水线阻塞
 - 做法2：由编译器通过静态分支预测和代码调度来消除同时发射指令间内部依赖，由硬件检测数据冒险并进行流水线阻塞
- 即：保证打包指令内部不会出现冒险！

静态多发射处理器实例

实例：MIPS ISA 指令集的静态多发射----2发射处理器

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

要使原MIPS处理器能够同时处理两条流水线，数据通路需要做哪些改进？

1. 同时要能取并译码两条指令，怎么办？

- 将两条指令打包成64位长指令，前面为ALU/Branch，后面为lw/sw
- 没有配对指令时，就用nop指令代替
- 将64位长指令中的两个操作码同时送到控制器（指令译码器）进行译码

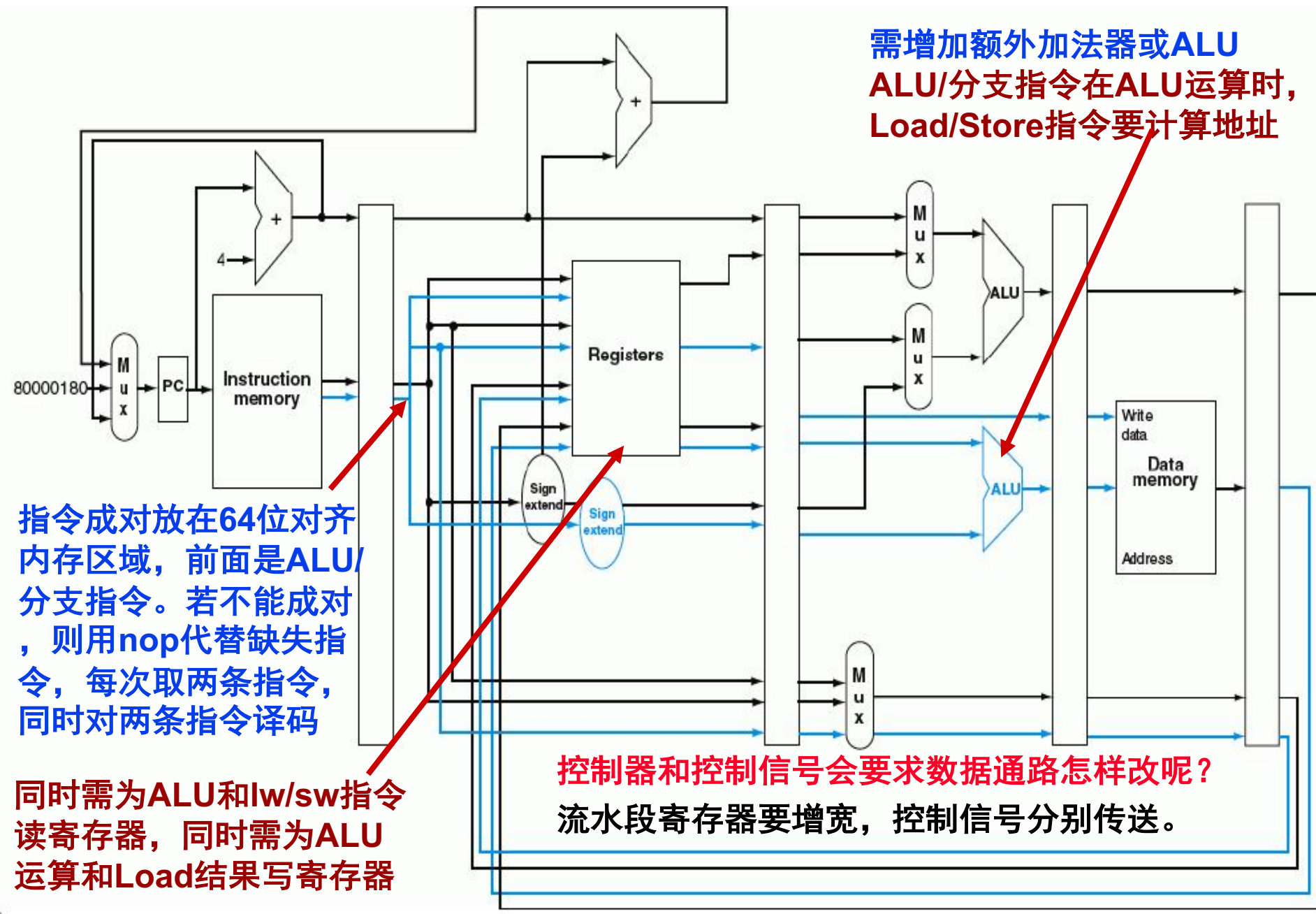
2. 两条指令同时要读两个寄存器或写寄存器（和lw配对时），怎么办？

- 增加两个读口和一个写口

3. 两条指令同时要使用ALU进行运算，怎么办？

- 增加一个ALU（包括2组输入总线和1组输出总线）

2发射流水线数据通路（蓝色是增加部分）



2发射流水线的特点

- 优点：潜在性能将提高大约2倍（实际上达不到！为什么？）
- 缺点：
 - 为消除结构冒险，需增加额外部件
 - 增加了潜在的由于数据冒险和控制冒险导致的性能损失
 - 例1：对于Load-use数据冒险
 - 单发射流水线：只有一条指令延迟
 - 2发射流水线：有一个周期（即2条指令）延迟
 - 例2：对于ALU-Load/Store数据冒险
 - 单发射流水线：可用“转发”技术使ALU结果直接转发到Load/Store指令的EXE阶段
 - 2发射流水线：两条指令同时进行，ALU的结果不能直接转发，因而不能提供给与其配对的Load/Store指令使用，只能延迟一个周期

为更有效地利用多发射处理器的并行性，必须有更强大的编译器，能够充分消除指令间的依赖关系，使指令序列达到最大的并行性！

例：2发射MIPS指令调度

- 以下是一段循环代码段

```
Loop: lw      $t0, 0($s1)
      addu    $t0, $t0, $s2
      sw      $t0, 0($s1)
      addi    $s1, $s1, -4
      bne     $s1, $zero, Loop
```




(能看出这段程序的功能吗?) 循环内进行的是数组访问!

- 为了能在2发射MIPS流水线中有效执行, 该怎样重新排列指令
 - 调度方案如下: 没有指令配对时, 用nop指令代替

前三条和后两条各具有相关性
可把第四条指令调到第一条后面
sw指令是否有问题? 怎么办?
\$s1减4, 故sw指令偏移改为4
能否把addi和lw配成一对?
寄存器\$s1被同时读, 并读后写,
两条指令的操作不会相互影响。

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:	addi \$s1, \$s1, -4 ? ?	lw \$t0, 0(\$s1)	1
	addi \$s1, \$s1, -4		2
	addu \$t0, \$t0, \$s2		3
	bne \$s1, \$zero, Loop	sw \$t0, 4(\$s1)	4



一个循环内, 五条指令在四个时钟内完成, 实际CPI为0.8, 即: IPC=1.25
在循环中访问数组的更好的调度技术是“循环展开”

用“循环展开”技术进行指令调度

- 基本思想：展开循环体，生成多个副本，在展开的指令中统筹调度
- 上例采用“循环展开”后的指令序列是什么？
 - 为简化起见，假定循环执行次数是4的倍数
 - “循环展开”4次后循环内指令（lw, addu, sw与数组访问相关）各有4条，再加1条addi和1条bne，共14条指令 为何第一条指令将\$s1减16？与\$t0关联的指令偏移为何不同？
 - 指令最佳调度序列如下：

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:	addi \$s1,\$s1,-16	lw \$t0, 0(\$s1)	1
		lw \$t1, 12(\$s1)	2
	addu \$t0,\$t0,\$s2	lw \$t2, 8(\$s1)	3
	addu \$t1,\$t1,\$s2	lw \$t3, 4(\$s1)	4
	addu \$t2,\$t2,\$s2	sw \$t0, 16(\$s1)	5
	addu \$t3,\$t3,\$s2	sw \$t1, 12(\$s1)	6
		sw \$t2, 8(\$s1)	7
	bne \$s1,\$zero,Loop	sw \$t3, 4(\$s1)	8

14条指令用了8个时钟，CPI达到 $8/14=0.57$ 。

需要用到“重命名寄存器”技术，多用了三个临时寄存器\$t1,\$t2,\$t3，消除了名字依赖关系（非真实依赖，只是寄存器名相同而已）

代价是什么？ 多用了三个临时寄存器，并增加了代码大小（存储空间变大）

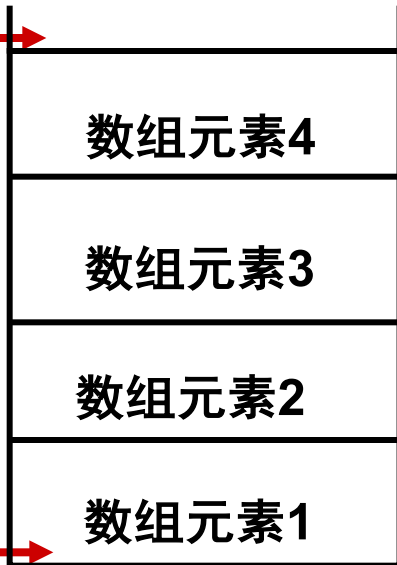
好处：充分利用并行，并消除部分循环分支！

循环展开后的偏移量

- 第一条指令将\$*s1*减16，指令执行后，\$*s1*的值变成了循环结束时\$*s1*的值
- 所以循环体内各数组元素的访问指令的偏移量依次为：
16 - 数组元素1， 12 - 数组元素2， 8 - 数组元素3， 4 - 数组元素4

新的\$*S1*

\$*s1*-16



为什么第一个周期中的lw指令的偏移量为0？
因为第一个周期中的lw指令进行地址计算时，addi指令的执行结果还没有写到\$*s1*中，所以，此时\$*s1*中还是原来的值？

为什么第一条addu指令不放在第二周期？
为了避免load-use数据冒险！

当循环次数不是4的倍数时这样做就有问题！

可见：编译器和机器结构密切相关！系统程序员必须非常了解机器结构！
编译器的好坏直接影响程序执行快慢！

实例：Intel IA-64架构

- IA-64类似于64位MIPS架构，是Register-Register型的RISC风格指令集
- 但有独特性：要求编译器**显式地**给出指令级的并行性，Intel称其为EPIC
Explicitly Parallel Instruction Computer—显式并行指令计算机
- 与MIPS-64架构的区别
 - 更多寄存器：128个整数、128个浮点数、8个专用分支、64个1位谓词
 - 支持寄存器窗口重叠技术
 - 同时发射的指令组织在指令包（bundle）中
 - 引入特殊的谓词化技术，以支持推测执行和消除分支，提高指令级并行度
- EPIC的实现技术
 - 指令组（Instruction Group）：相互间没有寄存器级数据依赖的指令序列
 - 指令组长度任意，用“停止标记”在指令组之间明显标识
 - 指令组内部的所有指令可并行执行，只要有足够硬件且无内存操作依赖
 - 指令包：同时发射的指令重新编码并形成指令包 $14+3\times 7+6=41$ op有14位
 - 长度为128，由5位长的模板字段、三个41位长的指令组成
 - 模板字段对应于以下五类功能部件中的三条指令 $3\times 41+5=128$
整数ALU、非整数ALU（移位和多媒体）、访存、浮点、分支
 - 谓词化：将指令的执行与谓词相关联，而不是与分支指令关联

IA-64是几-发射流水线？ 3-发射流水线！

[BACK](#)

RISC的通用寄存器

- RISC机采用大量寄存器
- 其目的：
 - 减少程序访问存储器的次数
- RISC机寄存器的组织方式有两种：

- 重叠寄存器窗口技术ORW（硬件方法）

- 执行过程调用和返回时，利用寄存器组而不是存储器来完成参数传递
- 通过重叠窗口技术，使得不再需要保存和恢复寄存器内容
- 可大大提高了程序执行的速度

- 优化寄存器分配技术（软件方法）

- 规定一套寄存器分配算法
- 通过编译程序的优化处理来充分利用寄存器资源
- 编译器为那些在一定的时间内使用最多的变量分配寄存器

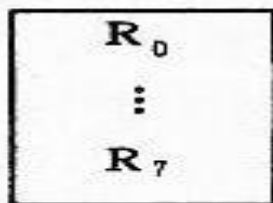
回顾：过程调用时，MIPS是如何传递参数的？x86是如何传递参数的？哪种开销小、效率高？

MIPS通过约定寄存器、X86通过栈（Stack）传递的！用栈传递开销大！

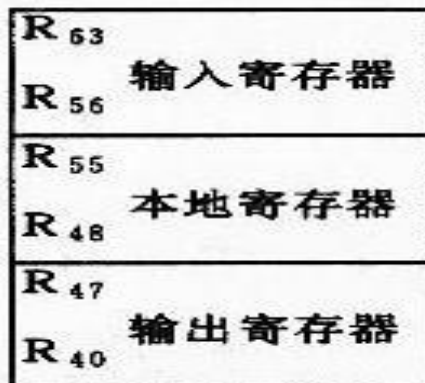
[BACK](#)

重叠寄存器窗口技术 (Overlapped Register Window)

全局寄存器

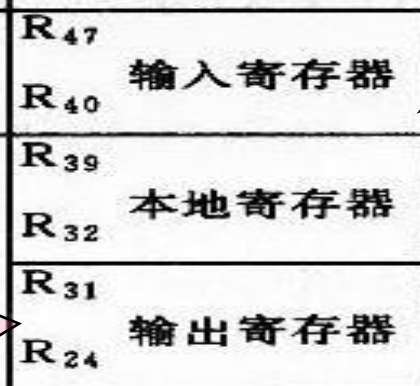


过程A的“窗口”



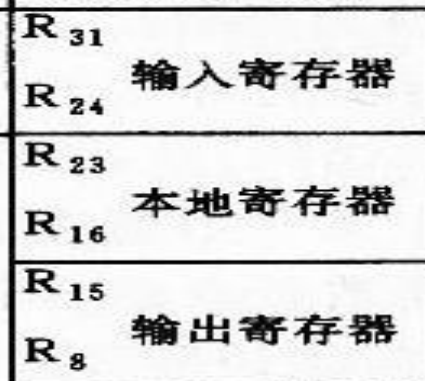
用于保存
局部数据

过程B的“窗口”



与调用自己的
父过程的输出
寄存器重叠

过程C的“窗口”



与被调用子
过程的输入
寄存器重叠

寄存器分为两类：

全局R：所有过程共享

窗口R：用于过程调用

A调用B时，由于各自使用不同的局部寄存器，所以不需保存现场

A过程的输出寄存器可直接把参数送给B

从B返回时，B将返回结果送到其输入寄存器，A可直接得到B返回的结果

[BACK](#)

相当于输入参数和返回参数用同一套（重叠窗口）寄存器！

Intel IA-64架构的谓词和推测执行技术

- 谓词和谓词寄存器
 - 分支指令中的条件称为谓词
 - 每个谓词与一个谓词寄存器相关联
 - 每条指令都可与最后6位标识（64个一位谓词，故谓词寄存器的标号用6位表示）的谓词寄存器相关联，反映条件是否满足
- 可消除循环内if-then-else分支（循环分支可由循环展开部分消除）

例：if (p) { Statement1 } else { Statement2 } 被编译成：

(p) Statement1

(~p) Statement2

括号中的条件为1时，执行后面的代码，否则，转化为nop指令

- 条件分支指令被转化为由谓词寄存器关联的指令，消除了分支
- 通过谓词寄存器可实现指令的推测执行

IA-64是采用静态多发射机制的比较复杂的指令集，对编译器的要求极高

[BACK](#)

动态多发射处理器

- 由硬件在执行时动态完成指令打包或冒险处理
- 通常被称为超标量处理器（Superscalar）
 - 同一个时钟动态发射多条指令，一个周期内可执行一条以上指令
- 与VLIW处理器的不同点：
 - **VLIW处理器**：与机器结构密切相关，在结构有差异的机器上要重新编译
 - **超标量处理器**：编译器仅进行指令顺序调整（还是串行序列），不进行指令打包，而是由硬件根据机器结构决定同时发射哪几条指令。因此，编译后的代码能够被不同结构的机器正确执行
- 超标量处理器多结合动态流水线调度（Dynamic pipeline scheduling）技术
 - 通过指令相关性检测和动态分支预测等手段，投机性地不按指令顺序执行，当发生流水线阻塞时，可以到后面找指令来执行
 - 举例说明动态流水线调度技术：

lw	\$t0, 20(\$s2)
addu	\$t1, \$t0, \$t2
sub	\$s4, \$s4, \$t3
slti	\$t5, \$s4, 20

左边指令序列中，哪条指令可以提前执行？

sub指令可提前到addu指令前执行

如果不将sub调到前面，则会影响slti指令的执行，而且还会发生load-use冒险

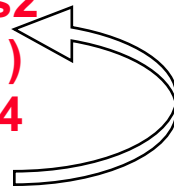
动态流水线调度的通用模型

◦ 动态流水线的一个重要思想：在等待解决阻塞时，到后面找指令提前执行！

◦ 动态流水线的通用模型：

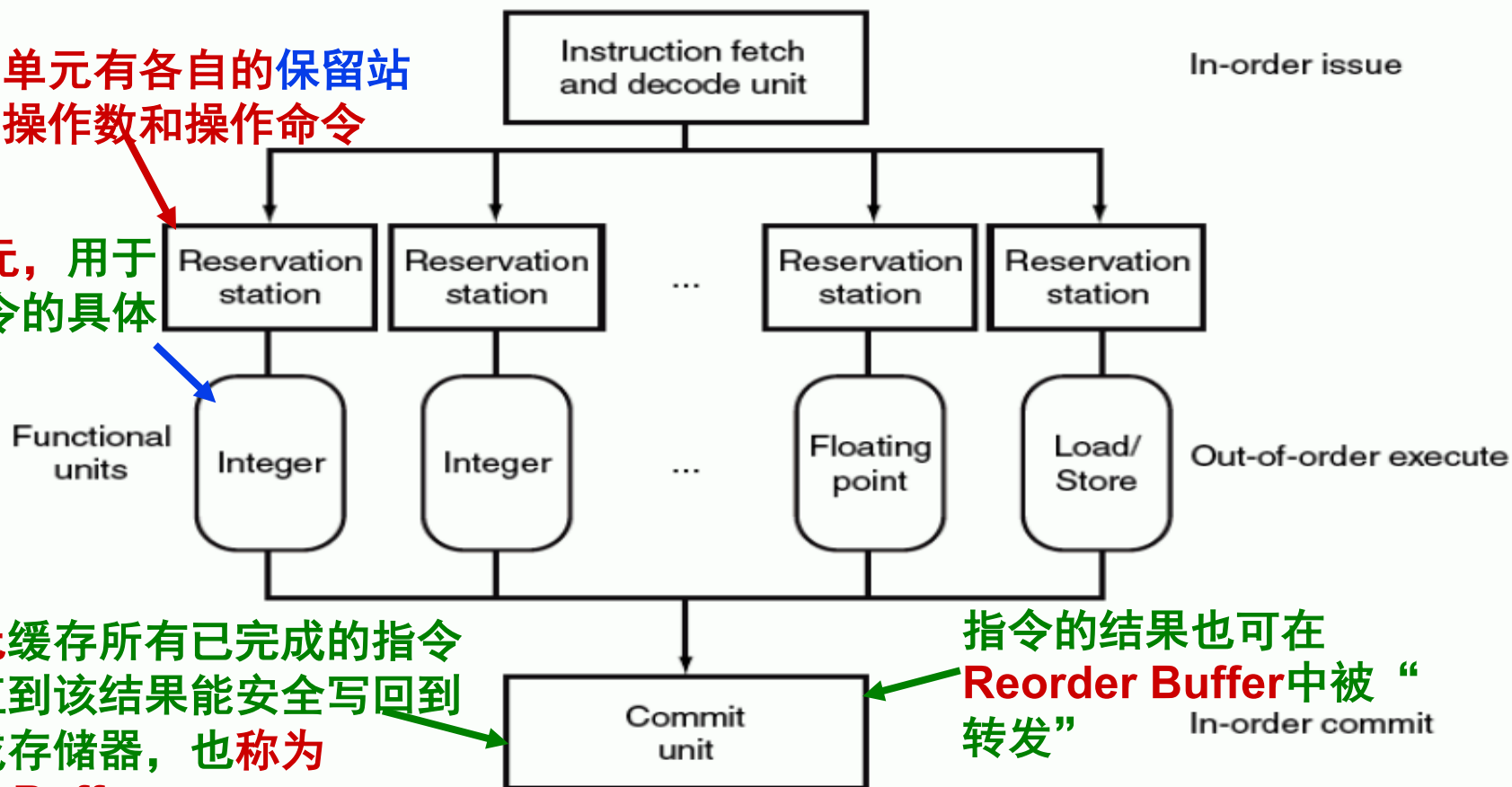
- 一个指令预取和译码单元：有序发射
- 多个并列执行的功能单元：乱序执行
- 一个提交单元：有序提交

```
addu $t0, $t0, $s2  
sw    $t0, 0($s1)  
addi  $s1, $t0, -4  
mul   $t1, $t1  
mflo  $s3
```



每个功能单元有各自的保留站
用于保存操作数和操作命令

功能单元，用于
完成指令的具体
功能



提交单元缓存所有已完成的指令
结果，直到该结果能安全写回到
寄存器或存储器，也称为
Reorder Buffer

指令的结果也可在
Reorder Buffer中被“
转发”

功能单元的性能

- 功能：用来执行特定类型的操作
- 性能：每个功能单元具有基本的操作性能，用两个周期数来刻画
 - 执行周期（Latency）：完成特定操作所花的周期数
 - 发射时间（Issue Time）：连续、独立操作之间的最短周期数

以下是Pentium III 算术功能部件的性能

Operation	Latency	Issue Time
Integer Add	1	1
Integer Multiply	4	1
Integer Divide	36	36
Floating-Point Add	3	1
Floating-Point Multiply	5	2
Floating-Point Divide	38	38
Load (Cache Hit)	3	1
Store (Cache Hit)	3	1

从上述图中看出，哪些功能部件是流水化的？哪些是非流水化的？

- 整数加、整数乘、浮点加、装入、存储这五种部件是流水化的
- 浮点乘部件是部分流水化
- 整数除和浮点除是完全没有流水化

CPU设计的一个原则：有限的芯片空间应该在各功能部件之间进行平衡！尽量让大多数资源用于最关键的操作（对大量基准程序进行评估）

从上述图中能否看出，哪些是最重要的操作？哪些是不常用的？

整数加法和乘法、浮点数加法和乘法是重要的操作

除法相对来说不太常用，而且本身难以实现流水线

动态流水线的几种执行模式

根据动态流水线指令发射和完成顺序，可分为三种执行模式：

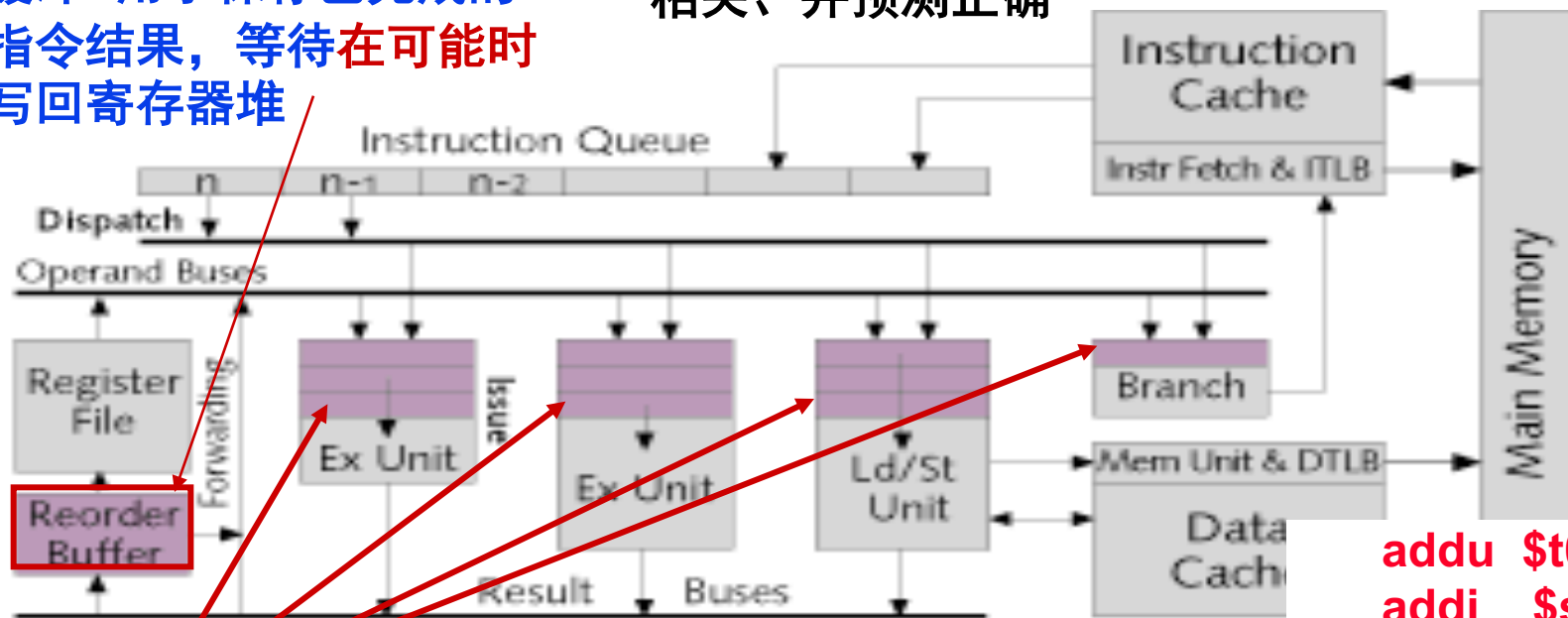
- 按序发射按序完成（Pentium）
- 按序发射无序完成（Pentium II和Pentium III）
- 无序发射无序完成（Pentium 4）

保守的是顺序完成，好处：

- (1) 简化异常检测及其处理
- (2) 能在被推测指令完成前得知推测结果的正确性

ReOrder Buffer 重排序缓冲：用于保存已完成的指令结果，等待在可能时写回寄存器堆

写回条件：与前面的所有指令结果不相关、并预测正确



保留站：存放操作数和操作命令

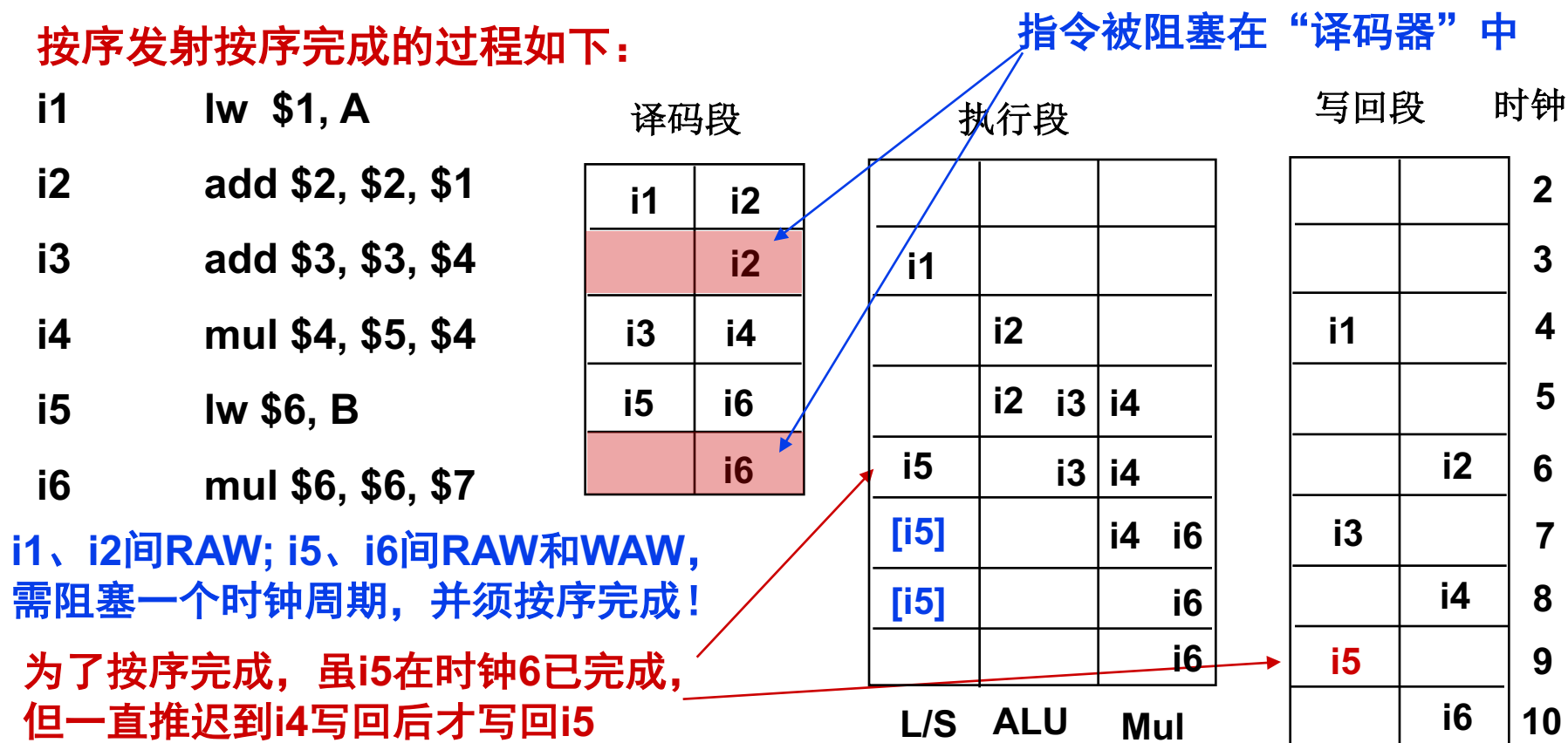
SKIP

```
addu $t0, $t0, $s2
addi $s1, $t0, -4
beq  $s1, $t3, exit
mul  $t1, $t1
mflo $s3
```

按序发射按序完成

- 举例：2发射超标量，分为取指（F）、译码（D）、执行（E）、写回（W）。F、D、W段在一个时钟周期内完成（可同时有两条指令在这三个阶段）；E段有三个执行部件：Load/Store部件完成数据Cache访问需要1个时钟，整数ALU完成简单ALU操作需2个时钟，整数乘法器完成乘法运算需要3个时钟。执行部件采用流水化方式。

按序发射按序完成的过程如下：

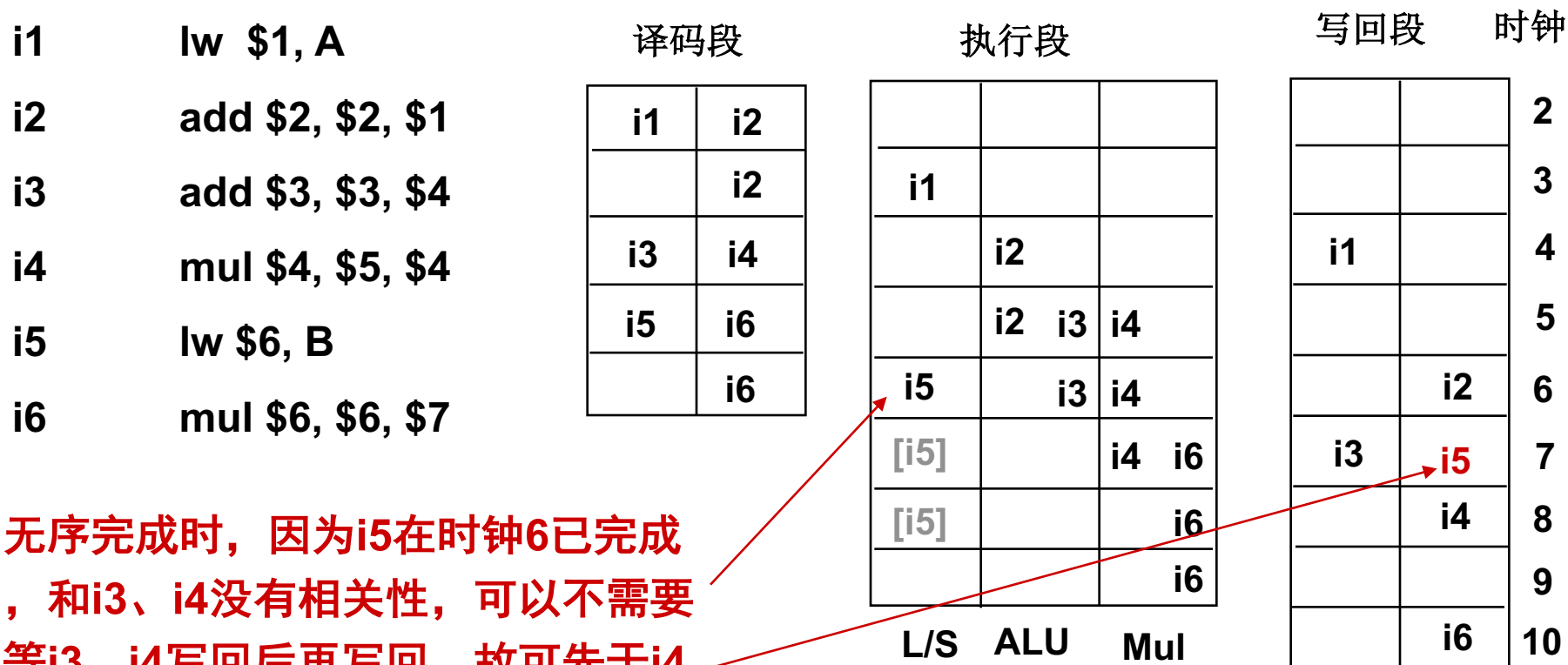


如果还有一条乘法指令, 则最多可有三条乘法指令同时在执行

按序发射无序完成

- 举例：2发射超标量，分为取指（F）、译码（D）、执行（E）、写回（W）。F、D、W段在一个时钟周期内完成（可同时有两条指令在这三个阶段）；E段有三个执行部件：Load/Store部件完成数据Cache访问需要1个时钟，整数ALU完成简单ALU操作只需2个时钟，整数乘法器完成乘法运算需要3个时钟。执行部件采用流水化方式。

按序发射无序完成的过程如下：



[BACK](#)

无序发射无序完成

- 举例：2发射超标量，分为取指（F）、译码（D）、执行（E）、写回（W）。F、D、W段在一个时钟周期内完成（可同时有两条指令在这三个阶段）；E段有三个执行部件：Load/Store部件完成数据Cache访问需要1个时钟，整数ALU完成简单ALU操作只需2个时钟，整数乘法器完成乘法运算需要3个时钟。执行部件采用流水化方式。

取指和译码按顺序进行，发射前进行相关性检测，无关指令可先行发射和先行完成！

无序发射无序完成的过程如下：

例如：i4在i3前面发射！

i1 lw \$1, A
i2 add \$2, \$2, \$1
i3 add \$3, \$3, \$4
i4 mul \$4, \$5, \$4
i5 lw \$6, B
i6 mul \$6, \$6, \$7

译码段	
i1	i2
i3	i4
i5	i6
i7	i8

执行段		
i1		
	i2	i4
i5	i2	i3
		i3
		i4
		i6
		i6
L/S	ALU	Mul

写回段	时钟
	2
	3
i1	4
	5
i2	6
i3	7
	8
	9
	10

无序发射的超标量中，译码后的指令被存放在一个“指令窗口”的缓冲器中，等待发射。当所需功能部件可用、且无冲突或无相关性阻碍指令执行时，就从指令窗口发射，与取指和译码的顺序无关

只要保证i1和i2、i5和i6之间的发射和完成顺序即可！

[BACK](#)

动态流水线调度的必要性

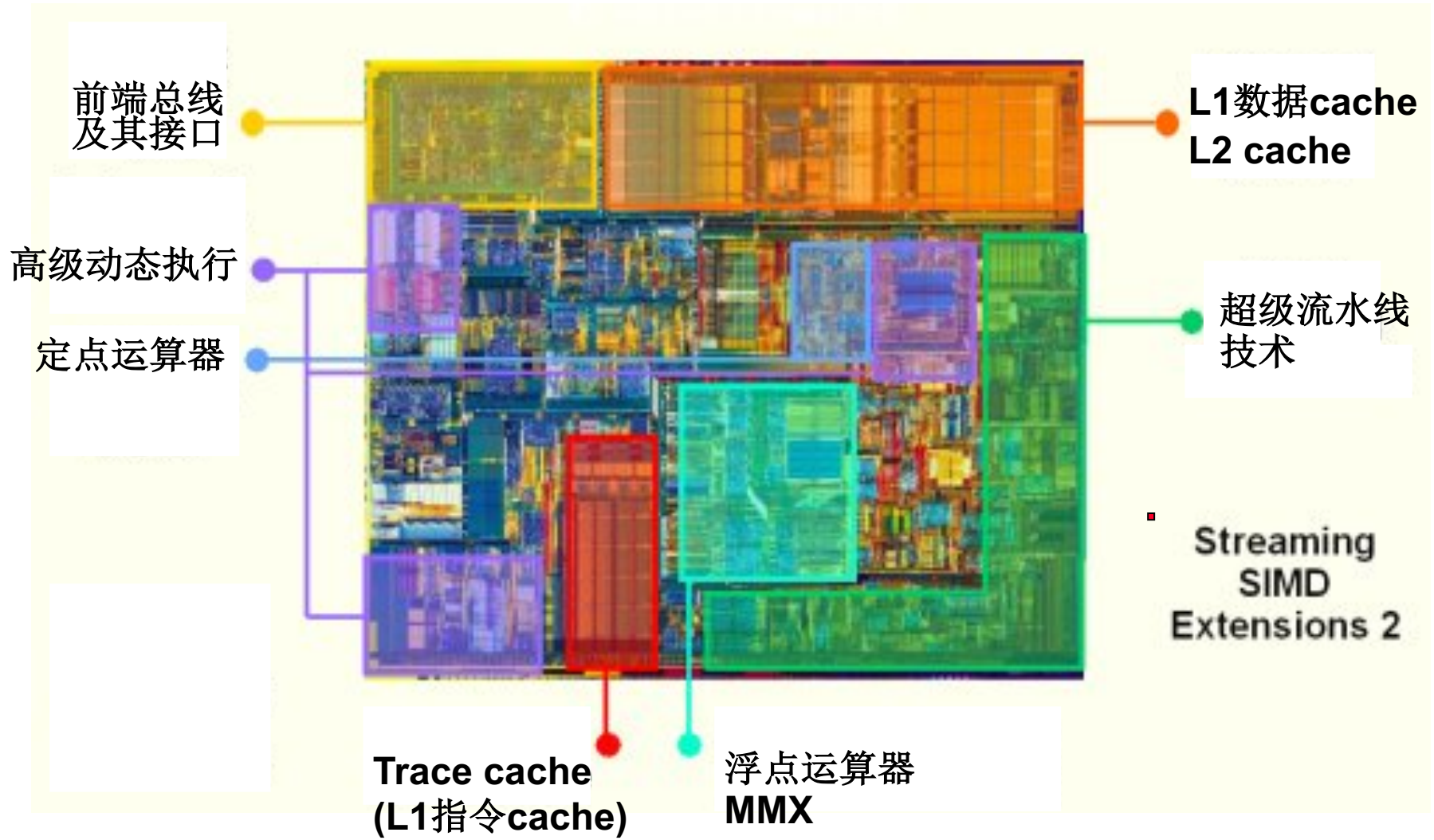
- 编译器可依据数据依赖关系调度代码，为什么还要超标量处理器来动态调度？
 - 并不是所有阻塞都能事先确定，动态调度可在阻塞时，提前执行无关指令
 - 例如，Cache缺失是不可预见的阻塞
 - 动态分支预测需要根据执行的真实情况进行预测
 - 采用动态调度使得硬件将处理器细节屏蔽起来
- (不同处理器的发射宽度、流水线延时等可能不同，流水线的结构也会影响循环展开的深度。通过动态调度使得处理器细节被屏蔽起来，软件发行商无需针对同一指令集的不同处理器发行相应的编译器，并且，以前的代码也可在新的处理器上运行，无需重新编译)

理解程序的性能：

高性能微处理器并不能持续进行多条指令的发射，原因有：

- 指令间的高度依赖关系限制了指令之间的并行执行，特别是隐含依赖关系的存在。例如，使用指针的代码段，存在隐含依赖。
- 分支指令预测错误。
- 内存访问引起的阻塞（Cache缺失、缺页等）使得流水线难以满负荷运转。

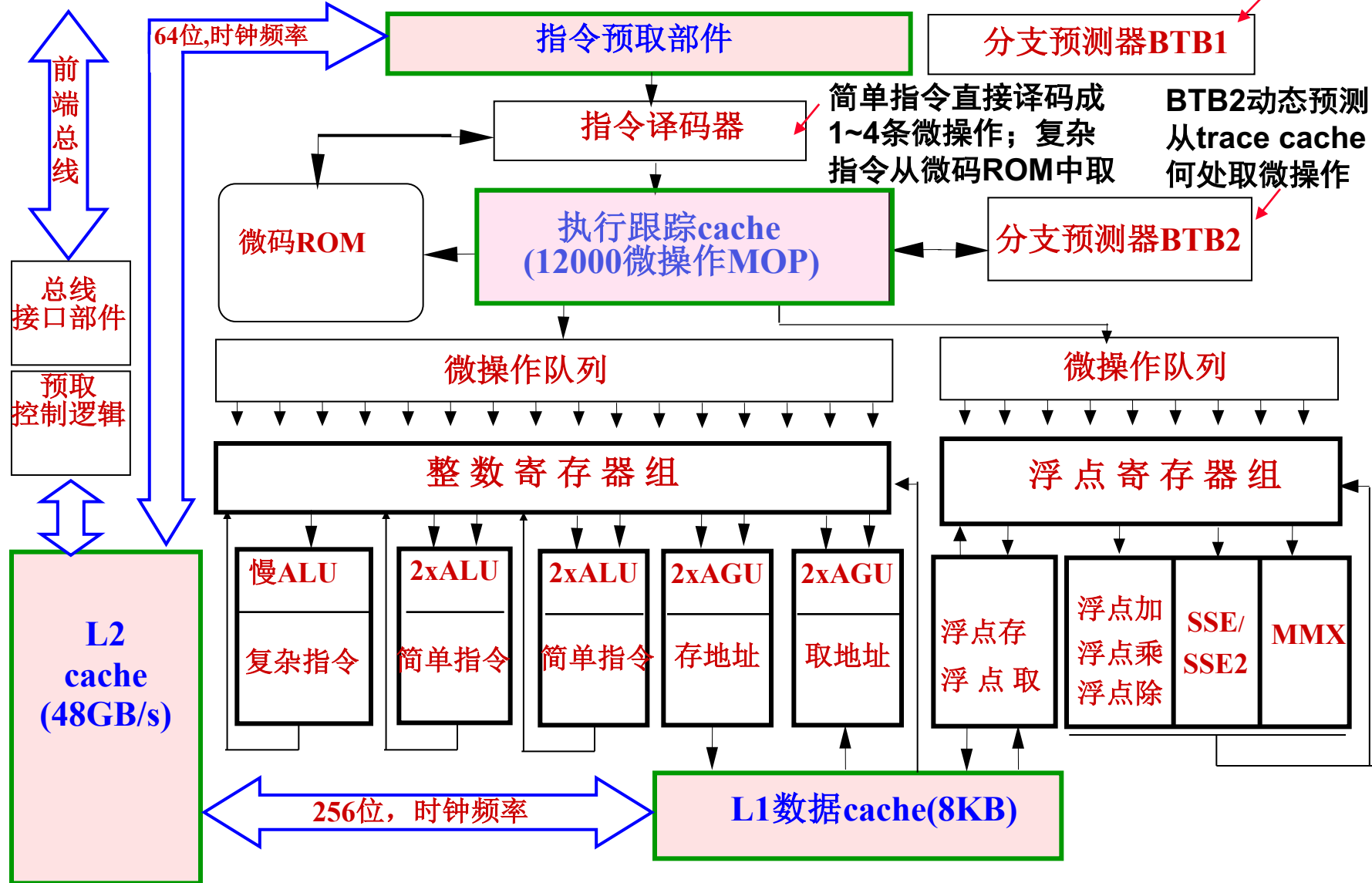
回顾：Pentium 4 处理器的芯片布局



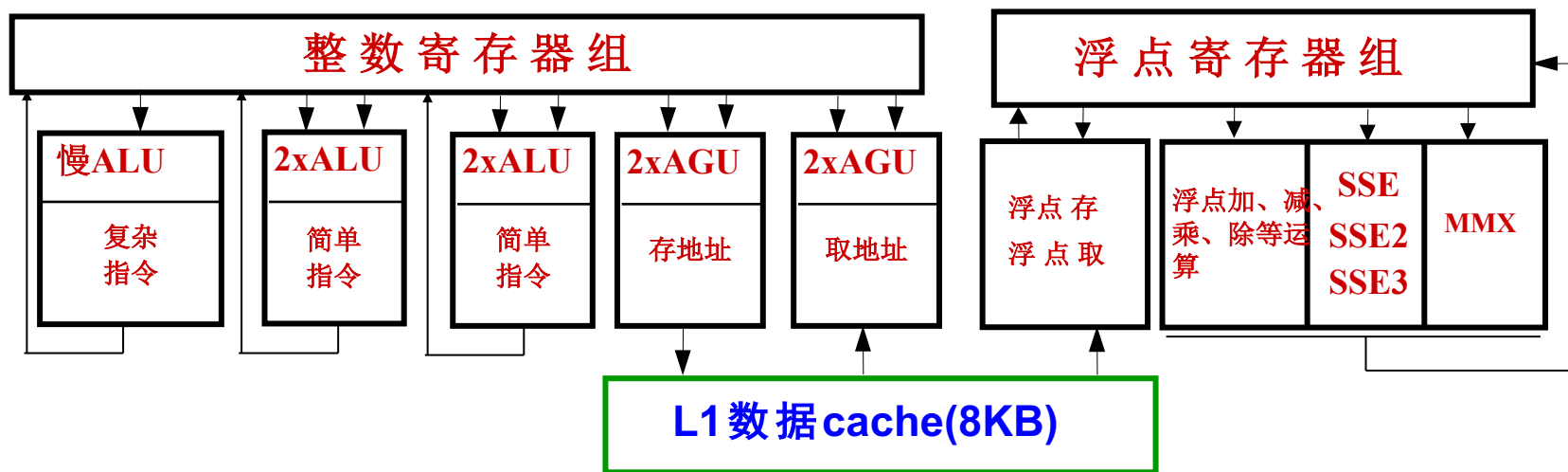
回顾: Pentium 4 处理器的逻辑结构

每个MOP相当于一RISC指令，格式没公开

BTB1静态预测从L2的何处预取指令



Pentium 4的超标量结构运算器



采用超标量（superscalar）结构，共含9个运算部件，可同时工作，所花时钟不同

- 2个高速整数ALU(每个时钟周期进行2次操作)，用于完成简单的整数运算(如加/减法)
- 1个慢速整数ALU(需要多个时钟周期才能完成1次操作)，用于完成整数乘、除法运算
- 2个地址生成部件（AGU），用于计算操作数的有效地址，所生成的地址分别用于从内存存取操作数或向内存保存操作结果
- 1个运算部件用于完成浮点操作数地址的计算
- 1个运算部件用于完成浮点加法、乘法和除法运算
- 1个运算部件用于执行流式的SIMD处理（SSE/SSE2/SSE3指令）
- 1个运算部件用于完成多媒体信号处理（MMX指令）

在运算部件中执行的是微操作，而不是指令！运算器中的操作采用流水方式！

回顾：Pentium 4 的用户可见寄存器组

整数寄存器组

在Pentium4内部，整数和浮点数各有128个物理寄存器

寄存器换名操作：将用户可见的逻辑寄存器换成内部的物理寄存器

寄存器换名时，要确定是真实依赖还是名字依赖（反依赖）

名字依赖时可用不同的物理寄存器替换相同的逻辑寄存器

指令计数器
标志寄存器

P4 Pentium 80386 80486 8086 8088

	31	15	8	7	0
GPR 0	EAX	AX	AH	AL	
GPR 1	ECX	CX	CH	CL	
GPR 2	EDX	DX	DH	DL	
GPR 3	EBX	BX	BH	BL	
GPR 4	ESP	SP			
GPR 5	EBP	BP			
GPR 6	ESI	SI			
GPR 7	EDI	DI			
	EIP	IP			
	EFLAGS	FLAGS			

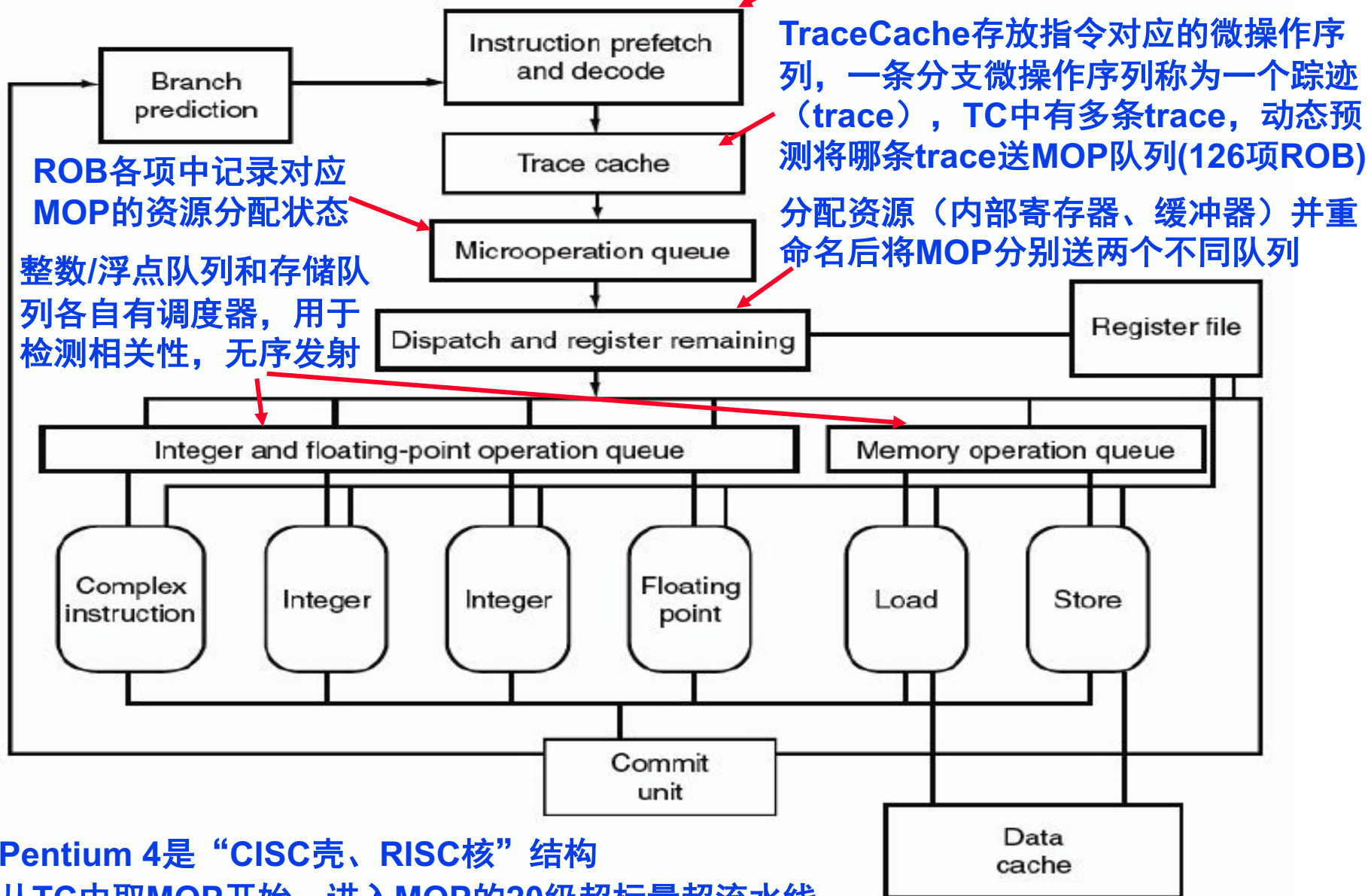
浮点寄存器组

79	FPR 0	0
	FPR 1	
	FPR 2	
	FPR 3	
	FPR 4	
	FPR 5	
	FPR 6	
	FPR 7	

Pentium4 流水线结构部分

每个MOP相当于一RISC指令

译码结果是对应的若干条微操作MOP



Pentium 4是“CISC壳、RISC核”结构

从TC中取MOP开始，进入MOP的20级超标量超流水线

Pentium 4的指令译码 – 对指令功能进行分解

◦ 指令译码逻辑：

- 功能：将指令转换为一组基本操作，称为微操作MOP
- 输入：程序中的指令
- 输出：微操作（简单计算任务）

例1： `addl %eax, %edx` 的译码结果为什么？

一个“加法”操作（对应一个微操作MOP）

例2： `addl %eax, 4(%edx)` 的译码结果呢？

四个简单操作（对应四个微操作MOP）：

“地址计算”： $\text{Reg}[\%edx] + 4 \rightarrow \text{addr}$

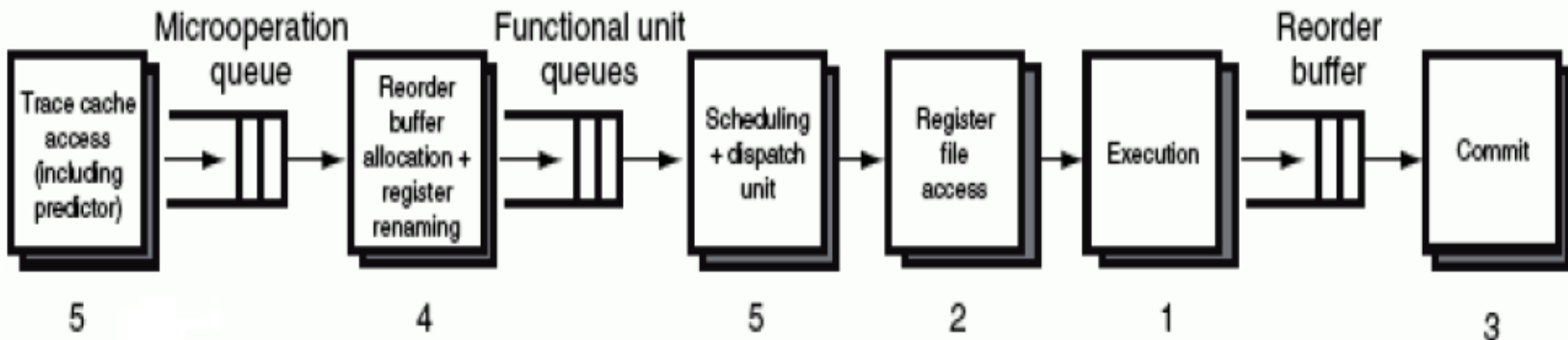
“装入”： $\text{Mem}[\text{addr}] \rightarrow \text{Reg}[\text{Rtemp}]$

“加法”： $\text{Reg}[\text{Rtemp}] + \text{Reg}[\%eax] \rightarrow \text{Reg}[\text{Rtemp}]$

“存数”： $\text{Reg}[\text{Rtemp}] \rightarrow \text{Mem}[\text{addr}]$

一个微操作相当于一组RISC指令，译码生成的微操作序列被存放到TC中
(TC: Trace Cache)

Pentium4 的20级超流水线(Hyper-pipeline)



整数运算微操作流水线为20级，浮点为29级（执行阶段的长度不同）

两个drive段用于芯片内传输信号的驱动，使其保证长距离传输



沿一个踪迹顺序取MOP，直到遇到一条转移MOP，通过BTB2预测下个踪迹开始点，继续取MOP送ROB/Alloc/Ren部件。预测目标处MOP不在时，要通知指令预取器，快从L2中取指令并译码。

一个周期3条MOP送ROB。ROB有126项，记录每个MOP及分配的资源 and 执行状态，根据资源分配情况进行寄存器重命名后，分别送两个MOP队列中进行排队。

每个队列按FIFO将MOP送到各自的调度器，在调度器中进行数据相关性检测，当所有源操作都就绪时，将MOP发射到对应的执行部件。是“无序”发射。

被发射的MOP开始读取物理寄存器中的源操作数，或从旁路由L1-D Cache读取。

在不同的执行部件中执行。每个部件执行时间长短不同

建立标志信息ZF/CF等，并将执行结果写入物理寄存器。对BTB2中预测是否正确进行确认及相应处理

本讲小结

- 有以下两种指令级并行(ILP)技术（即：高性能流水线形式）
 - 超流水线：更多的流水线级数
 - 多发射流水线：同时发射多个指令，有多条流水线同时进行
 - 静态多发射（VLIW处理器+编译器静态调度）
 - 动态多发射（超标量处理器+动态流水线调度）
- 静态多发射（VLIW（超长指令字）处理器）
 - 由编译器静态推测来完成“指令打包”和“冒险处理”
 - MIPS 2-发射Datapath中有2个执行部件，将2条指令打包，并同时译码执行
 - 采用循环展开进行指令调度，能得到很好的性能
 - IA-64采用VLIW技术，Intel称其为EPIC技术，3条指令打包
- 动态多发射（超标量处理器）
 - 指令执行时由硬件动态推测，多个执行部件，同时发射多个指令到执行部件
 - 3种动态多发射流水线的执行模式
 - 按序发射按序完成、按序发射无序完成、无序发射无序完成
 - Pentium 4 动态多发射流水线（无序发射、无序完成）
 - 简单指令由硬件译码器译码执行，复杂指令由微操作ROM产生MOP
 - 指令对应的MOP存放在trace cache中，按一条条trace存放
 - 20级以上超流水线、3发射超标量、2个队列动态调度、126条微操作同时执行
 - 指令静态预测 + 微操作动态预测

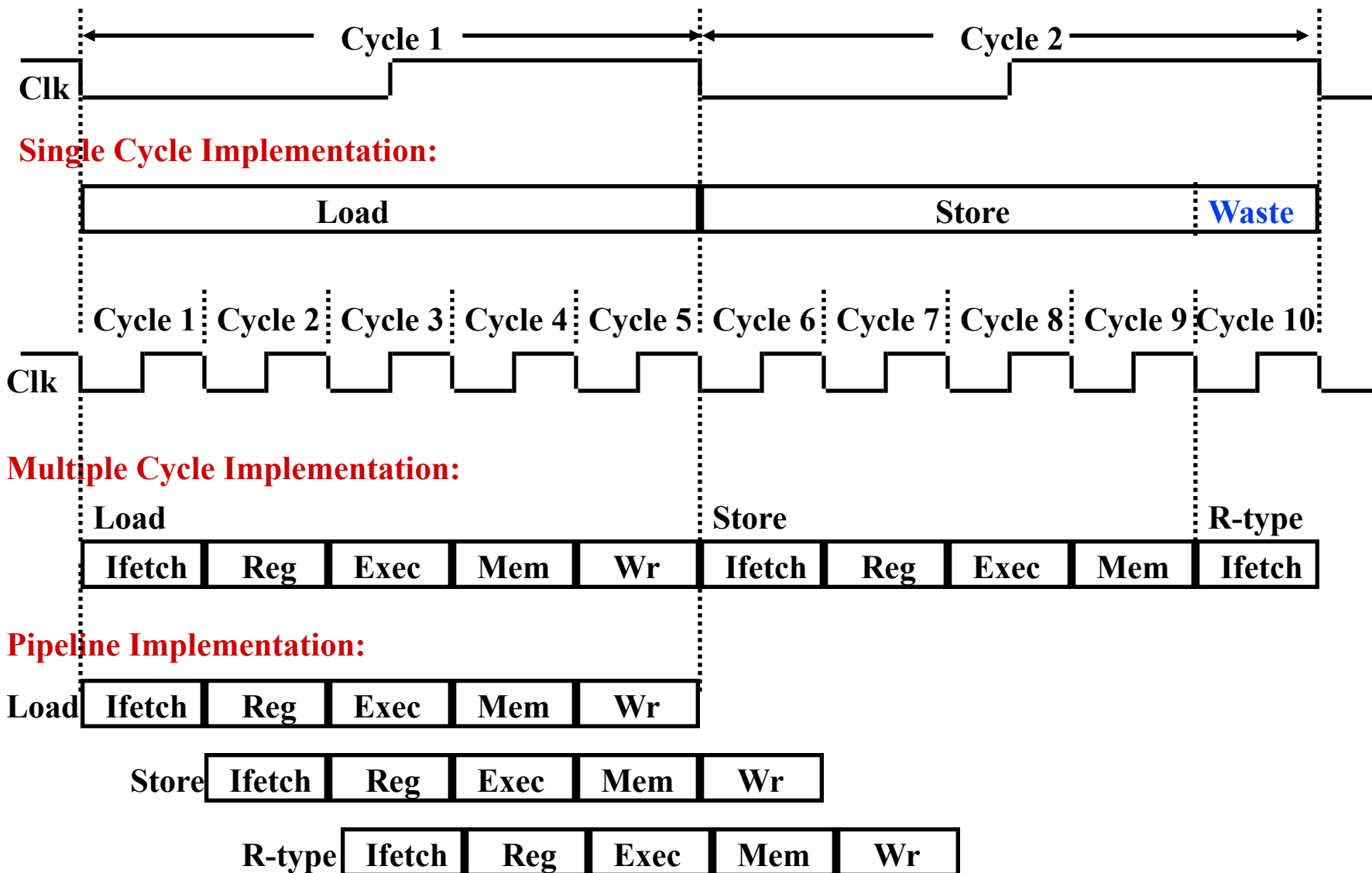
本章总结1

◦ 指令流水线的设计

- 将每条指令的执行规整化为若干个同样的流水阶段
- 每个流水阶段的执行时间一样，都等于一个时钟
- 理想情况下，每个时钟有一条指令进入流水线，也有一条指令执行结束
- 每个流水段中的部件都是组合逻辑加寄存器，组合逻辑中产生的结果在时钟到来时被存储到寄存器（如：程序计数器、条件码寄存器、流水线寄存器等）。
- 每两个相邻流水段之间的流水线寄存器，用以记录所有在后面阶段要用到的各种信息，有哪些呢？
 - 控制信号、指令的代码、参加运算的操作数、指令运算结果、指令异常信息、寄存器读口地址、寄存器写口地址、存储器地址、PC+4、本条指令地址等。
- 指令译码得到的控制信号通过流水线寄存器传送到后面各个流水段中

本章总结2

单周期, 多周期 和 流水线比较



本章总结3

- 指令流水线的局限性
 - 并不是每条指令都有相同多个流水段
 - 并不是每个流水段都一样长
 - 随着流水线深度的增加，流水线寄存器的额外开销比例也增大
 - 指令在资源冲突、数据相关或控制相关时会发生流水线冒险
- 指令流水线的执行效率
 - 吞吐率：比非流水线方式下大大提高
 - 指令执行时间：相对于非流水线方式，一条指令的执行时间延长了
- 提高流水线指令效率的高级流水线技术
 - 超流水线：级数更多的流水线
 - 多发射流水线：同时发射多条指令的流水线
 - 静态多发射：VLIW结构、编译器静态推测
 - 动态多发射：超标量结构、硬件动态推测调度

本章总结4

- 结构冒险（资源冲突）：多条指令同时使用同一个功能部件
 - 规定每个功能部件在一条指令中只能被用一次
 - 规定每个功能部件只能在某个特定的阶段被用
 - 指令存储器(Code Cache)和数据存储器(Data Cache)分开
- 数据冒险（数据相关）：前面指令的结果是后面指令的操作数
 - 软件阻塞：（如：编译器）在后面的数据相关指令前插入nop指令
 - 硬件阻塞：在后面数据相关指令的特定流水段插入“气泡”以“阻塞”指令继续执行，直到取得所需数据为止
 - “转发”（旁路）：把前面指令执行过程中得到的数据直接传送到后面指令相应的流水段中。
 - 对于load-use，采用“阻塞加转发”的方式解决数据冒险

本章总结5

- 控制冒险（控制相关）：返回指令、分支指令等可能改变顺序增量的PC值，由于获取转移目标地址的时间较长，使得在目标地址产生前已经有指令被取到流水线中，如果已经取出执行的指令不是正确的指令，则发生控制冒险。
 - 软件阻塞：（如：编译器）在控制相关指令后面插入nop指令
 - 硬件阻塞：在控制相关指令后面的指令被取出前插入“气泡”，使流水线停顿若干时钟，直到控制相关指令得到正确的PC值为止
 - 采用“分支预测”技术。简单（静态）地预测每次分支结果都一样，或根据分支指令执行历史进行动态预测，动态预测能达到90%以上的成功率
 - 采用延迟分支技术。将前面一条与分支指令无关的指令放到分支指令后面执行，这样，流水线不会发生阻塞现象。这种对指令顺序进行调整的工作在程序编译阶段完成