第8章 中央处理器（一）

第一讲 中央处理器概述

第二讲 单周期数据通路的设计

第三讲 单周期控制器的设计

第四讲 多周期处理器的设计

第五讲 流水线处理器设计

第六讲 流水线冒险及其处理

第七讲 高级流水线技术

第一讲 中央处理器概述

主 要 内 容

- 指令执行过程
- CPU的基本组成
  - 操作元件（组合逻辑元件）
  - 状态 / 存储元件（时序逻辑元件）
- 数据通路与时序控制
- 计算机性能与CPU时间

CPU执行程序和指令的过程

- CPU执行一条指令的过程
  - 取指令
  - PC+“1”送PC

  } 取指阶段

  - 指令译码
  - 进行主存地址运算
  - 取操作数
  - 进行算术 / 逻辑运算
  - 存结果
  - 以上每步都需检测异常

  } 译码和执行阶段

  - 若有异常，则自动切换到异常处理程序
  - 硬件检测是否有“中断”请求，有则转中断处理
- RTL

问题：

“取指令”一定在最开始做吗?

“PC+1”一定在译码前做吗?

“译码”须在指令执行前做吗?

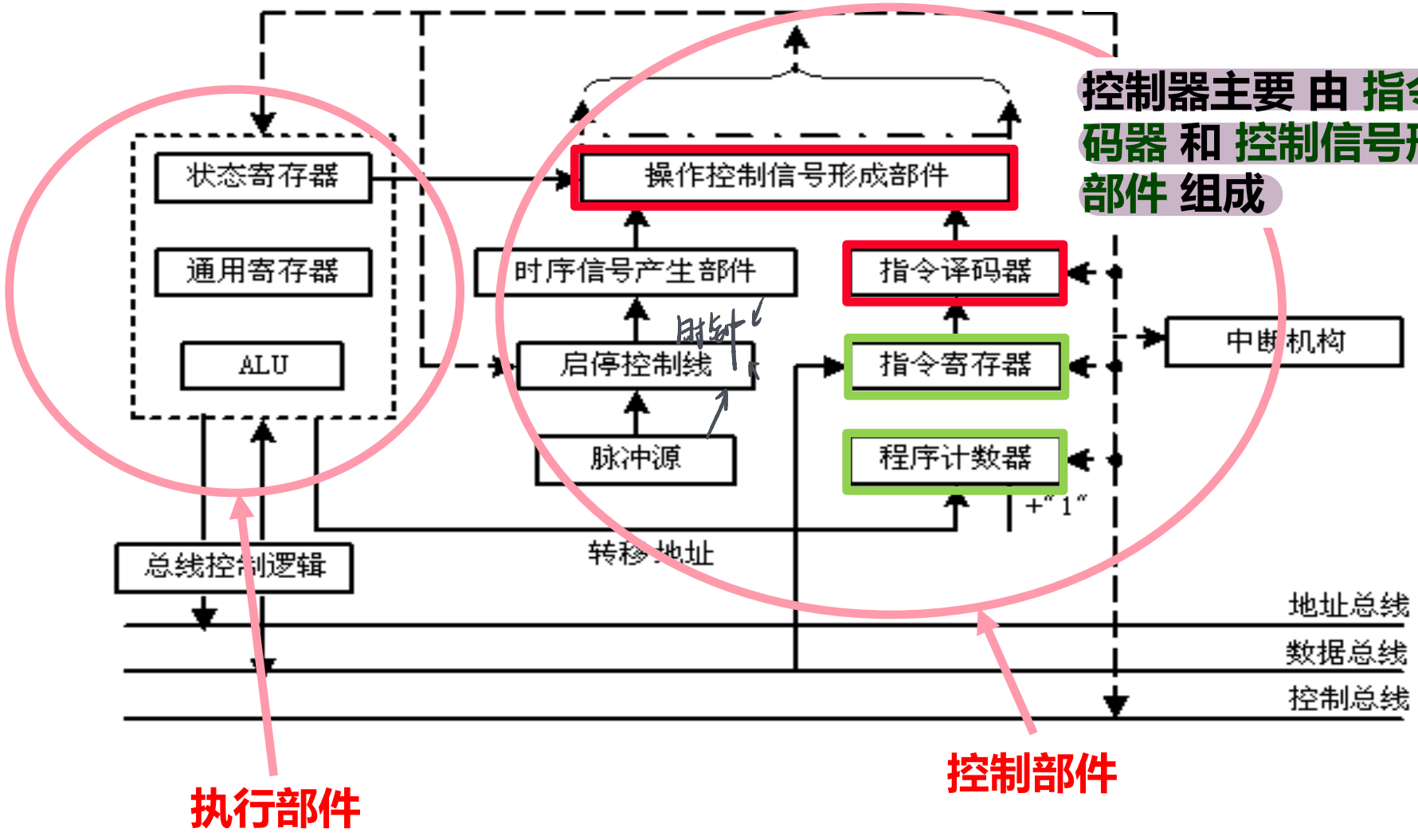
异常和中断的差别是什么?

异常是在CPU内部发生的

中断是由外部事件引起的

CPU基本组成原理图

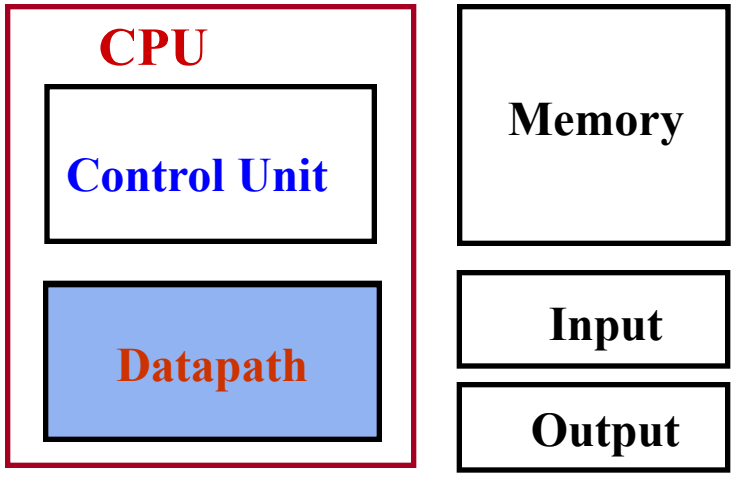
指令寄存器----IR
程序计数器---PC



CPU 由 执行部件 和 控制部件组成

CPU的基本结构

- 计算机的五大组成部分：



- 执行部件（数据通路）
  - 操作元件（如ALU）
  - 存储（状态）元件
    - 寄存器
- 控制部件（控制器）
  - 译码部件 组合逻辑
  - 控制信号生成部件
  - 存储（状态）元件
    - 寄存器

- 数据通路（DataPath）

  ——指令的执行部件

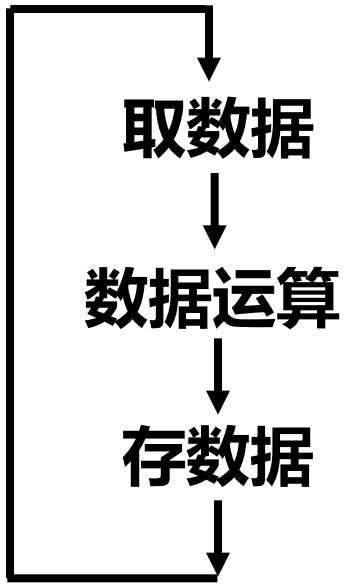
- 控制器（Control Unit）

  ——指令的控制部件

- 控制器负责对执行部件（数据通路）发出控制信号。

数据通路的基本结构

- 数据通路由两类元件组成
  - 组合逻辑元件（也称操作元件）
  - 时序逻辑元件（也称状态元件，存储元件）
- 元件间的连接方式
  - 总线连接方式
  - 分散连接方式
- 数据通路的具体工作是什么？
  - 进行数据存储、处理、传送



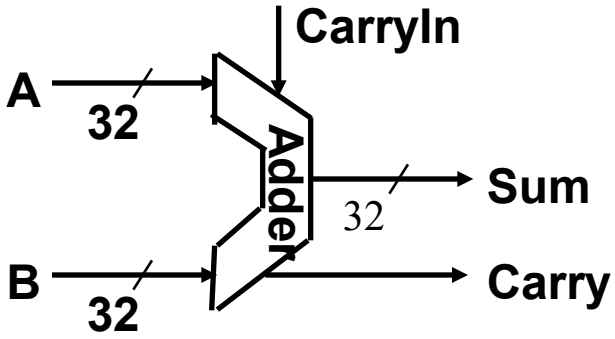
数据通路是由操作元件和存储元件通过总线方式或分散方式连接而成的进行数据存储、处理、传送的路径。

（回忆之前的“运算部件”）

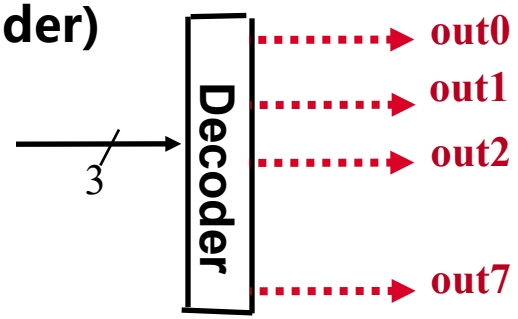
操作元件：组合逻辑电路

控制信号

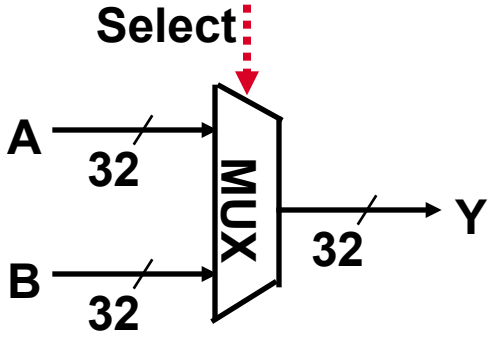
- 加法器 (Adder)



- 译码器 (Decoder)

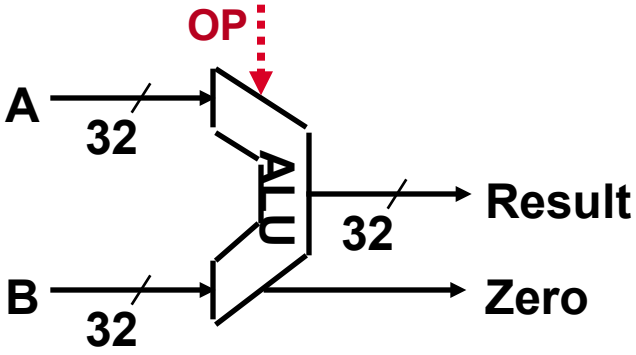


多路选择器 (MUX)



何时要用到adder, ALU, MUX or Decoder?

算逻部件 (ALU)



组合逻辑元件的特点：

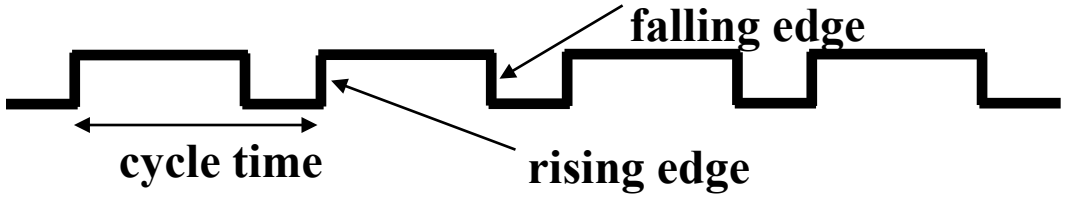
其输出只取决于当前的输入。即：若输入一样，则其输出也一样

定时：所有输入到达后，经过一定的逻辑门延时，输出端改变，并保持到下次改变，不需要时钟信号来定时

状态元件：时序逻辑电路

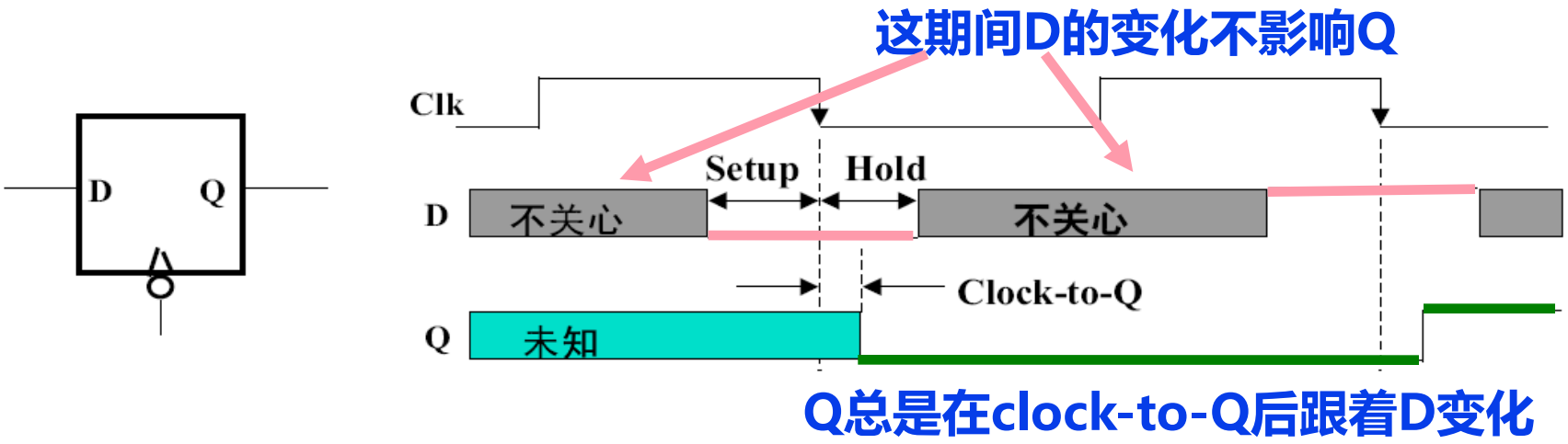
- 状态（存储）元件的特点：
  - 具有存储功能，在时钟控制下输入被写到电路中，直到下个时钟到达
  - 输入端状态由时钟决定何时被写入，输出端状态随时可以读出

- 定时方式：规定信号何时写入状态元件或何时从状态元件读出
  - 边沿触发（edge-triggered）方式：
    - 状态单元中的值只在时钟边沿改变。每个时钟周期改变一次。
      - 上升沿（rising edge） 触发：在时钟正跳变时进行读/写。
      - 下降沿（falling edge）触发：在时钟负跳变时进行读/写。



- 最简单的状态单元：
  - D触发器：一个时钟输入、一个状态输入、一个状态输出

存储元件中何时状态被改变？



- 建立时间（**Setup Time**）：在触发时钟边沿 之前 输入必须稳定
- 保持时间（**Hold Time**）：在触发时钟边沿 之后 输入必须保持
- **Clock-to-Q time:（Latch Prop - 锁存延迟）**
  - 在触发时钟边沿，输出并不能立即变化

切记：状态单元的输入信息总是在一个时钟边沿到达后的“Clk-to-Q”时才被写入到单元中，此时的输出才反映新的状态值

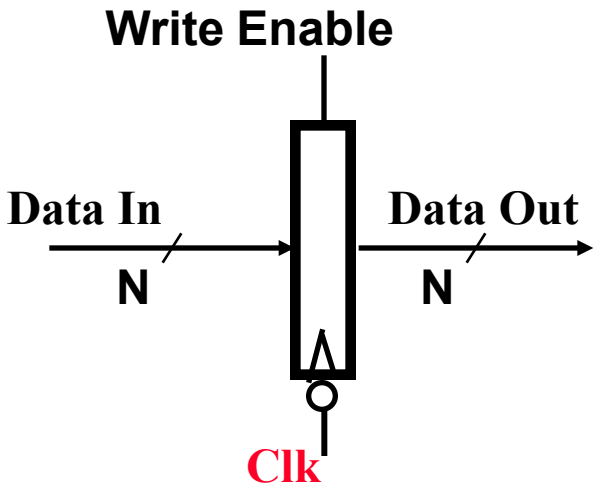
数据通路中的状态元件有两种：寄存器(组) + 理想存储器

存储元件：寄存器和寄存器组

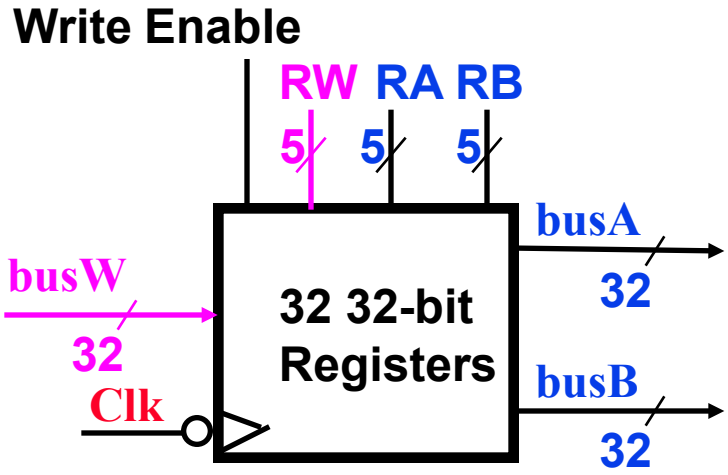
- 寄存器（Register）
  - 有一个写使能（Write Enable-WE）信号

    0: 时钟边沿到来时，输出不变

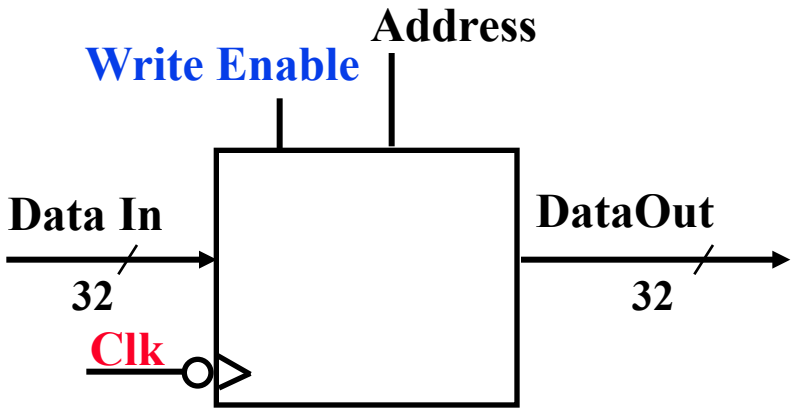
    1: 时钟边沿到来时，输出开始变为输入
  - 一定需要写使能信号吗？



- 寄存器组（Register File）
  - 两个读口（组合逻辑操作）：busA和busB分别由RA和RB给出地址。地址RA或RB有效后，经一个“取数时间(AccessTime)”，busA和busB有效。
  - 一个写口（时序逻辑操作）：写使能为1的情况下，时钟边沿到来时，busW传来的值开始被写入RW指定的寄存器中。



额外假设的状态元件: 理想存储器



- 理想存储器（idealized memory）
  - Data Out：32位读出数据
  - Data In： 32位写入数据
  - Address：读写公用一个32位地址

- 读操作（组合逻辑操作）：地址Address有效后，经一个“取数时间AccessTime”，Data Out上数据有效。
- 写操作（时序逻辑操作）：写使能为1的情况下，时钟Clk边沿到来时，Data In传来的值开始被写入Address指定的存储单元中。

为简化数据通路操作说明，把存储器简化为带时钟信号Clk的理想模型。并非真实存在于CPU中！

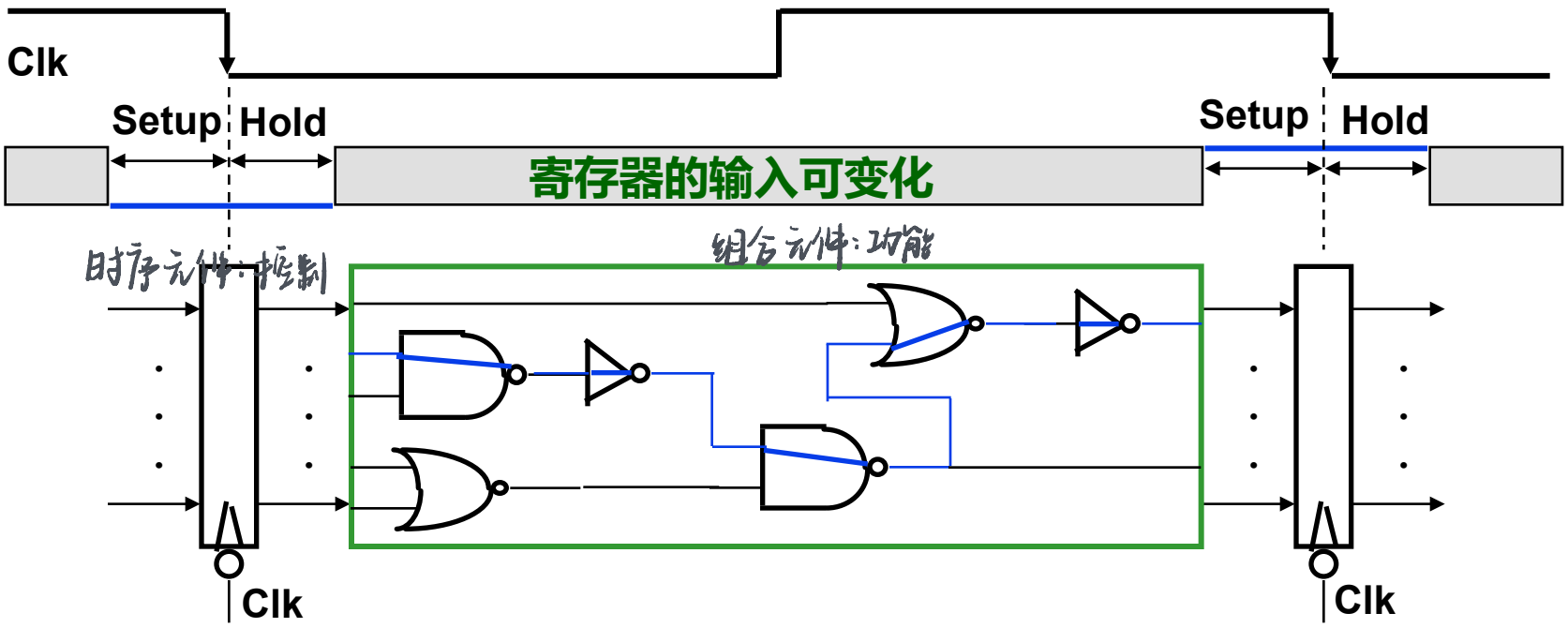
数据通路与时序控制

- 同步系统(Synchronous system)
  - 所有动作有专门时序信号来定时
  - 由时序信号规定何时发出什么动作

  例如，指令执行过程每一步都有控制信号控制，由时序信号确定控制信号何时发出、作用时间多长
- 什么是时序信号?
  - 早期计算机的三级时序系统（自行了解）
  - 用于进行同步控制的定时信号——时钟信号
- 什么叫指令周期?
  - CPU取并执行一条指令的时间
  - 每条指令的指令周期肯定一样吗？ ——不一定

- 时钟周期又是什么呢？

数据通路与时序控制　　现代计算机的时钟周期

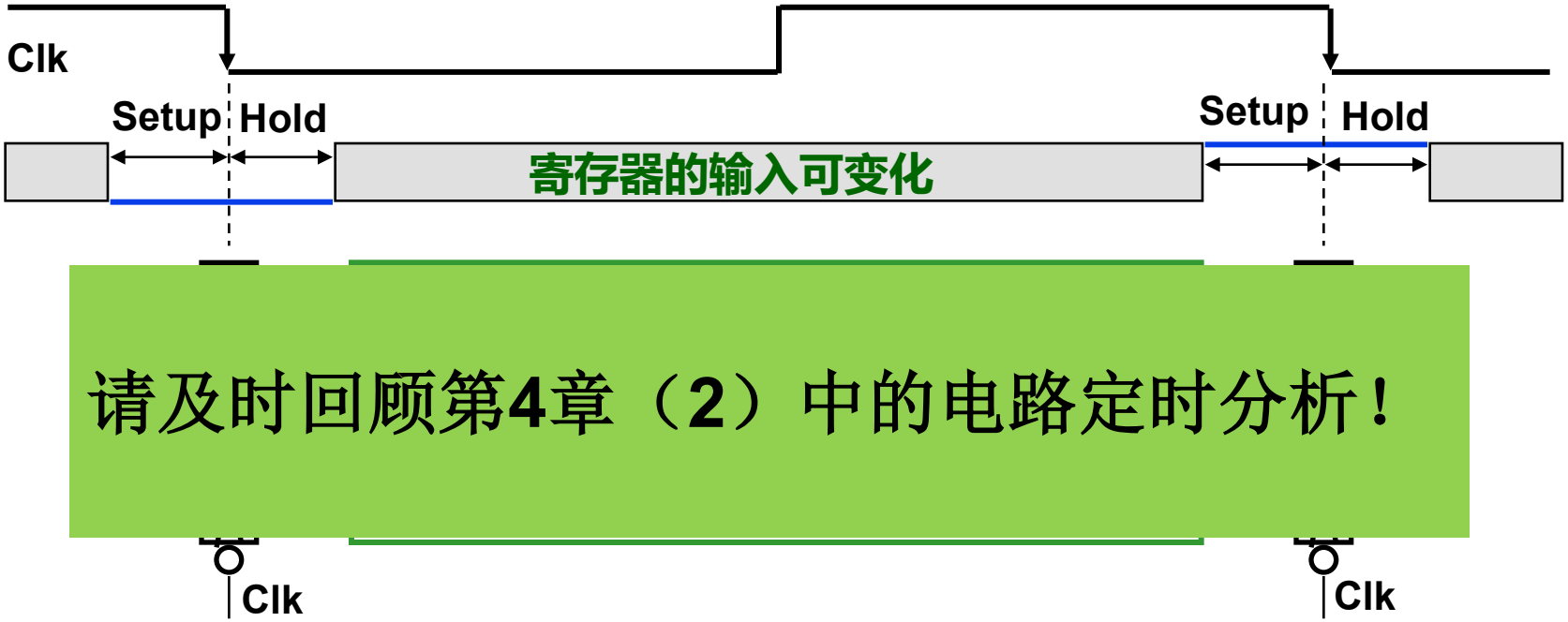


数据通路由 “ … + 状态元件 + 操作元件(组合电路) + 状态元件 + …” 组成

只有状态元件能存储信息，所有操作元件都须从状态单元接收输入，并将输出写入状态单元中。其输入为前一时钟生成的数据，输出为当前时钟所用的数据

- 假定采用下降沿触发（负跳变）方式（也可以是上升沿方式）
  - 所有状态单元在下降沿写入信息，经过Latch Prop (clk-to-Q) 后输出有效
  - Cycle Time = Latch Prop + Longest Delay Path + Setup + Clock Skew
- 约束条件: (Latch Prop + Shortest Delay Path - Clock Skew) > Hold Time

数据通路与时序控制　　现代计算机的时钟周期

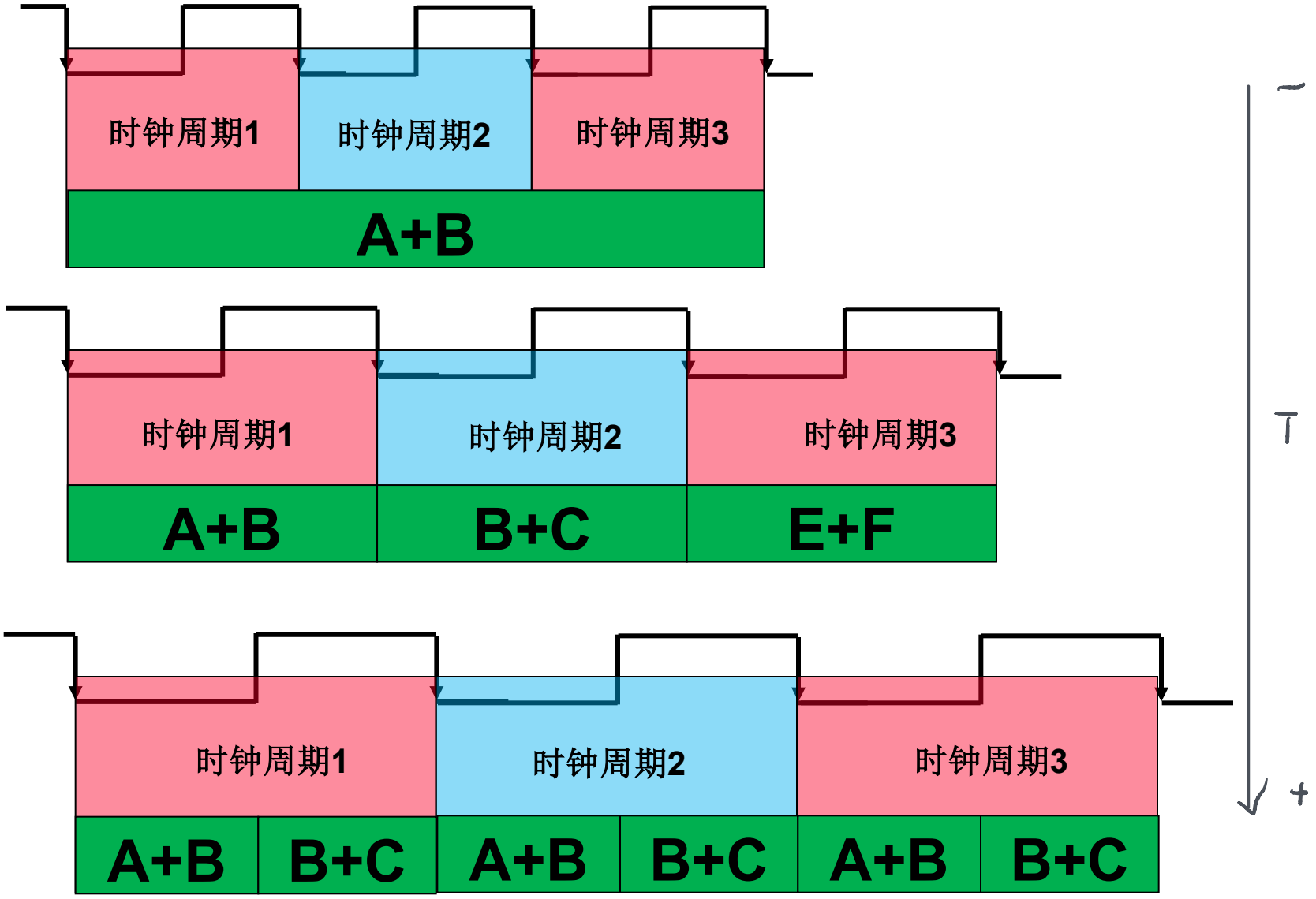


数据通路由 “ … + 状态元件 + 操作元件(组合电路) + 状态元件 + …” 组成

只有状态元件能存储信息，所有操作元件都须从状态单元接收输入，并将输出写入状态单元中。其输入为前一时钟生成的数据，输出为当前时钟所用的数据

- 假定采用下降沿触发（负跳变）方式（也可以是上升沿方式）
  - 所有状态单元在下降沿写入信息，经过Latch Prop (clk-to-Q) 后输出有效
  - Cycle Time = Latch Prop + Longest Delay Path + Setup + Clock Skew
- 约束条件：(Latch Prop + Shortest Delay Path - Clock Skew) > Hold Time

哪一种更好呢？——不能简单下结论



计算机（系统）性能

- 计算机有两种不同的性能
  - Time to do the task
    - 响应时间（response time）
    - 执行时间（execution time）
    - 等待时间或时延（latency）
  - Tasks per day, hour, sec, ns. ..
    - 吞吐率（throughput）
    - 带宽（bandwidth）

不同应用场合用户关心的性能不同：

-要求吞吐率高的场合，例如：

多媒体应用（音/视频播放要流畅）

-要求响应时间短的场合：例如：

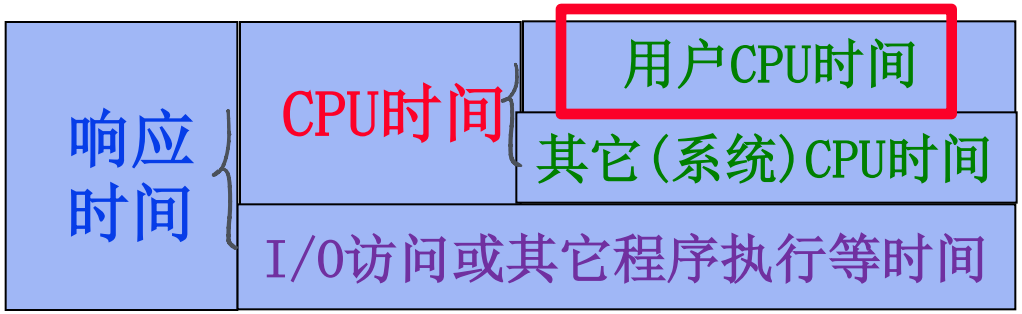
事务处理系统（存/取款速度要快）

-要求吞吐率高且响应时间短的场合：

文件服务器、Web服务器等

系统性能 和 CPU性能

CPU总是被多个程序（OS、多个应用软件等）共享使用，所以：



程序由指令构成。

CPU性能(CPU performance)：用户CPU时间 → 就是执行用户程序中各条指令的总时间

系统性能(System performance)：

一般指没有其它负载时的响应时间

吞吐率 = 单位时间内运行的作业（指令）数（有或无负载/干扰）

“机器X的速度（性能）是Y的n倍”的含义：

$$\frac{\text{ExTime(Y)}}{\text{ExTime(X)}} = \frac{\text{Performance(X)}}{\text{Performance(Y)}} = n$$

CPU执行时间（用户CPU时间，下同）的计算

CPU 执行时间 = CPU时钟周期数 / 程序 ÷ 时钟频率

= CPU时钟周期数 / 程序 × 时钟周期

= 指令条数 / 程序 × CPI × 时钟周期

CPI：Cycles Per Instruction

对于某一条特定的指令而言，其CPI是一个确定值——与CPU设计有关。无需计算。

但是，对于某一个程序或一台机器而言，其CPI是一个平均值，表示该程序或该机器指令集中的1条指令执行时平均需要多少时钟周期。

如何计算平均CPI?

假定CPI_i和C_i分别为第i类指令的CPI和指令条数，则程序的总时钟数为：

$$\text{总时钟数} = \sum_{i=1}^{n} CPI_i \times C_i$$

所以，

$$\text{CPU时间} = \text{时钟周期} \times \sum_{i=1}^{n} CPI_i \times C_i$$

（1）一种计算程序综合CPI的方法:

CPI = 总时钟周期数 / 指令条数 = (CPU 时间×时钟频率) / 指令条数

（2）另一种计算程序综合CPI的方法：假定CPI_i、F_i是各指令CPI和在程序中的出现频率（所有频率总和为1）：

$$\text{CPI} = \sum_{i=1}^{n} CPI_i \times F_i$$

加权和

$$F_i = \frac{C_i}{\text{程序的总指令条数}}$$

一台机器的CPI：考虑该机器的指令集（每类指令的CPI 不同），也需要考虑该机器上运行何种程序（用到的指令频率会不同），最终进行平均

CPI和CPU性能的关系？

由上页可知（以下反映的是“程序运行在某CPU上的平均情况”）：

——CPI = CPU时钟周期数 ÷ 指令条数

有了CPI，反过来可以计算：

——CPU 执行时间 = 指令条数 × CPI × 时钟周期

CPI 常用来衡量以下各方面的综合结果

- Instruction Set Architecture（ISA）
- Implementation of the architecture
  （Organization & Technology）
- Program（Compiler、Algorithm）

但是，单靠CPI的大小，并不能绝对准确的反映CPU性能！

影响CPU性能的各个方面

指令条数 / 程序 × CPI × 时钟周期

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

	instr. count	CPI	clock rate
编程 Programming	X	X	
编译 Compiler	X	(X)	
ISA Instr. Set Arch.	X	X	
CPU设计 Organization		X	X
硬件 Technology			X

计算机硬件性能与ISA、计算机组织、计算机实现技术息息相关

Marketing Metrics （产品宣称指标）

MIPS = Instruction Count / Second x1/10^6

= Clock Rate / CPI x 1/10^6

Million Instructions Per Second （定点指令执行速度）

因为每条指令执行时间不同，所以MIPS总是一个平均值。

MFLOPS = FP Operations / Second x 1/10^6

Million Floating-point Operations Per Second（浮点操作速度）

都有各自的局限。

- 一个GFLOPS（gigaFLOPS）等於每秒拾亿（=10^9）次的浮点运算，
- 一个TFLOPS（teraFLOPS）等於每秒万亿（=10^12）次的浮点运算，
- 一个PFLOPS（petaFLOPS）等於每秒千万亿（=10^15）次的浮点运算，
- 一个EFLOPS（exaFLOPS）等於每秒百亿亿（=10^18）次的浮点运算

基准测试程序

基准测试程序——专门用来进行性能评价的一组程序

浮点运算实际上包括了所有涉及小数的运算，在某类应用软件中常常出现，比整数运算更费时间。现今大部分的处理器中都有浮点运算器。因此每秒浮点运算次数所量测的实际上就是浮点运算器的执行速度。

最常用来测量每秒浮点运算次数的基准程序（benchmark）之一，就是Linpack。

基准测试程序开发：SPEC (Systems Performance Evaluation Committee) http://www.spec.org

例子1

程序P在机器A上运行需10 s， 机器A的时钟频率为400MHz。现在要设计一台机器B，希望该程序在B上运行只需6 s.

机器B时钟频率的提高导致了其CPI的增加，使得程序P在机器B上时钟周期数是在机器A上的1.2倍。机器B的时钟频率达到A的多少倍才能使程序P在B上执行速度是A上的10/6=1.67倍？

分析:

CPU时间A （ 10s ）= 时钟周期数A / 时钟频率A

所以，时钟周期数A = 10 sec x 400MHz = 4000M个

时钟频率B = 时钟周期数B / CPU时间B

= 1.2 x 4000M / 6 sec = 800 MHz

机器B的频率是A的两倍，但机器B的速度并不是A的两倍！

例子2：如何给出综合评价结果？

问题：如果用一组基准程序在不同机器上测出了运行时间，那么如何综合评价机器的性能呢？

先看一个例子：

程序 1: A上运行1秒，　　B上运行10秒

程序2: A上运行1000秒，　　B上运行100秒

- A 速度是B的10倍？（如果看程序1）
- B速度是A的10倍？（如果看程序2）

可以计算总时间作为综合度量值：

B上运行110秒，A上运行1001秒（B是 A 的9.1倍）

可考虑每个程序在作业中的使用频度，即加权平均

综合评价（续）

- 可用以下两种平均值来评价：
  - Arithmetic mean(算术平均)：求和后除n
  - Geometric mean(几何平均)：求积后开根号n
- 根据算术平均执行时间能得到总平均执行时间
- 根据几何平均执行时间不能得到程序总的执行时间
- ----------
- 执行时间的规格化（测试机器相对于参考机器的性能）：
  - 参考机器上执行时间÷ 待测机器上执行时间
- 平均规格化执行时间不能用算术平均计算，而应该用几何平均
  - 程序1 从2秒变成1秒 等价于 程序2 从2000秒变成1000秒.
    （算术平均值不能反映这一点）

综上所述，算术平均和几何平均各有长处，可灵活使用

第8章 中央处理器（一）

第一讲 中央处理器概述

第二讲 单周期数据通路的设计

第三讲 单周期控制器的设计

第四讲 多周期处理器的设计

第五讲 流水线处理器设计

第六讲 流水线冒险及其处理

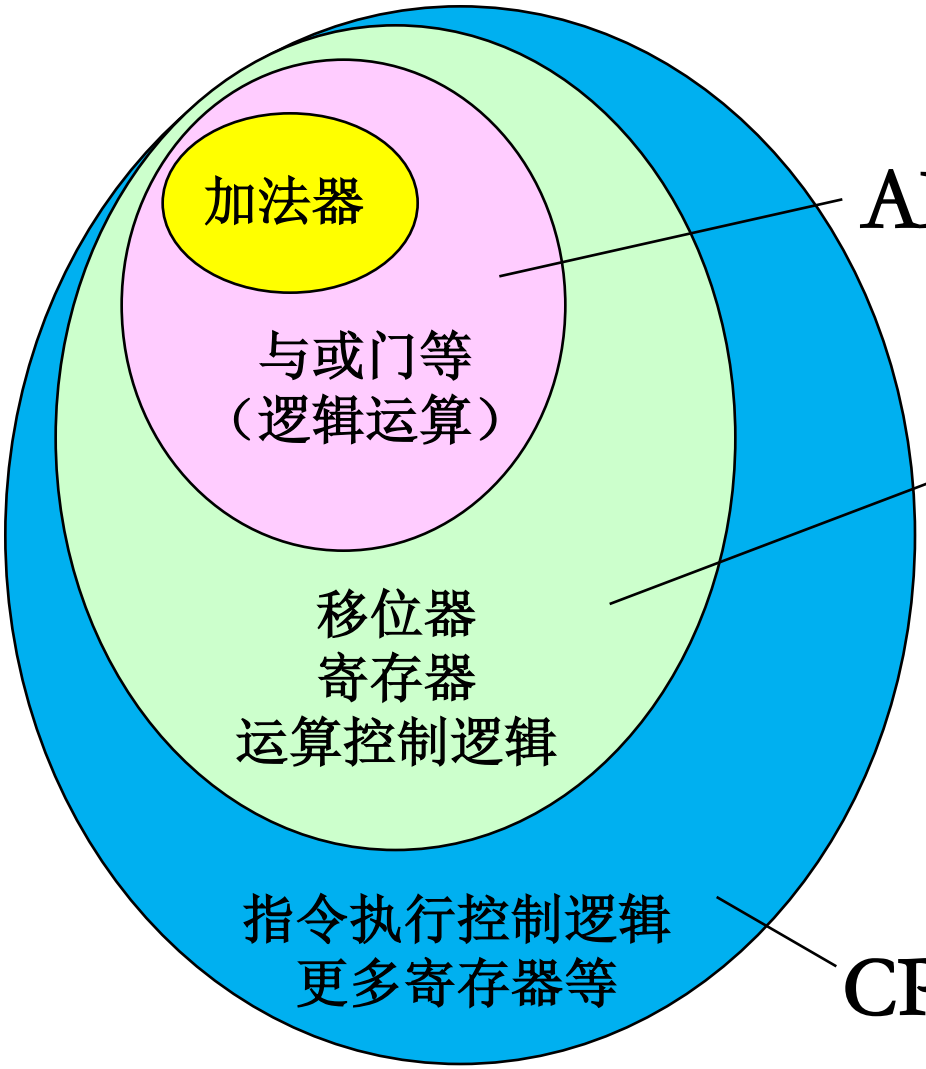
第七讲 高级流水线技术

第二讲 单周期数据通路设计

主 要 内 容

- 实现目标（RV32I中的9条指令）概述
- 扩展器部件的设计
- 算术逻辑部件的设计
- 取指令部件的设计
- R-型指令的数据通路
- I-型指令的数据通路
- U-型指令的数据通路
- Load/Store指令的数据通路
- B-型指令的数据通路
- J-型指令的数据通路
- 完整的单周期数据通路

回顾： CPU功能及其与计算机性能的关系



- 计算机性能(程序执行快慢)由三个关键因素决定（回顾）
  - 指令数目、CPI、时钟周期
    - 指令数目由编译器和ISA决定
    - CPI由ISA和CPU的实现来决定
    - 时钟周期由CPU的实现来决定

因此，CPU的设计与实现非常重要！它直接影响计算机的性能。

ALU以加法器为核心

运算部件以ALU/加法器为核心

（大家都需要：多路选择器和传输线路等）

单总线数据通路

四种基本操作的时序

- **在寄存器之间传送数据**

  取指令　**R0out，Yin**

- **完成算术、逻辑运算**

  R1out，Yin

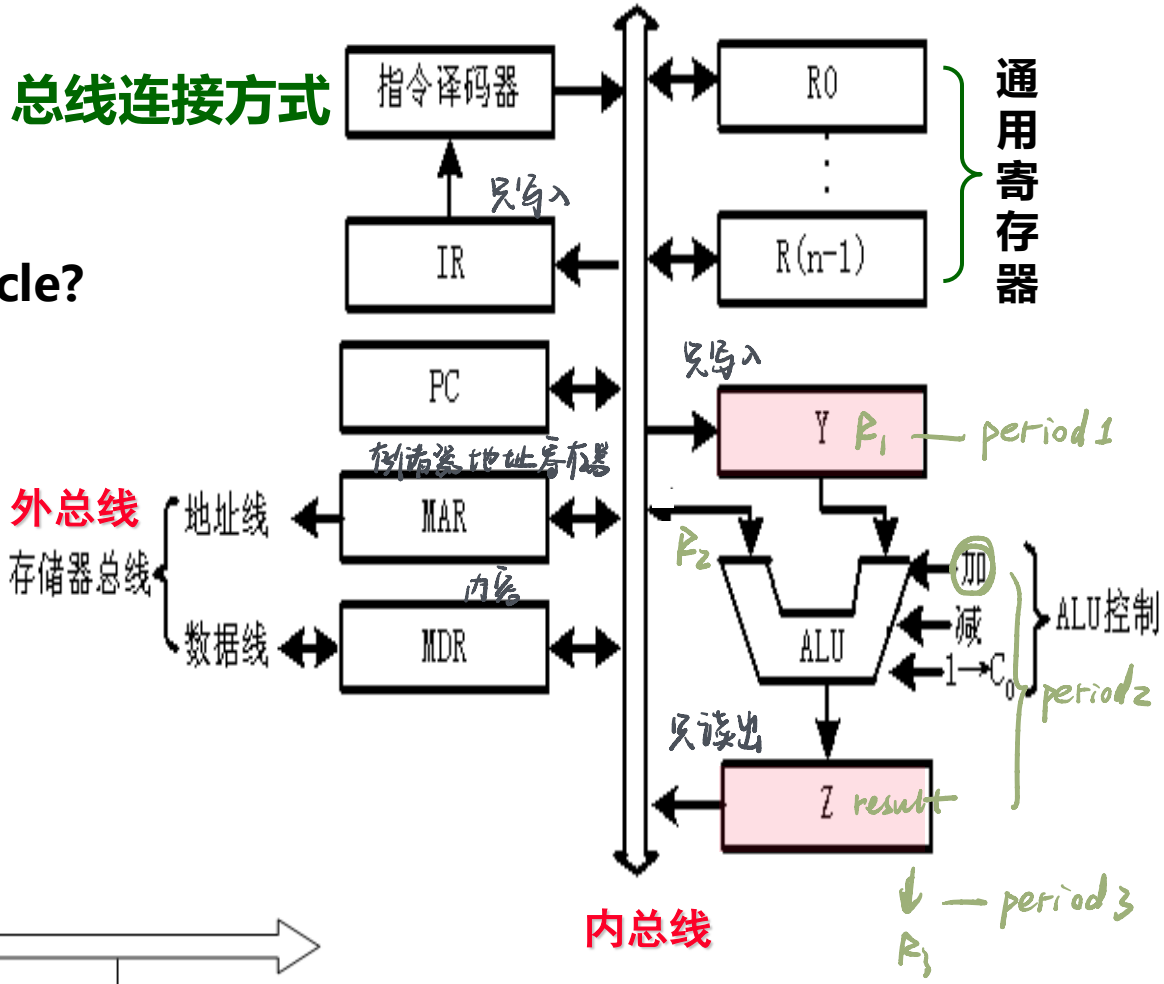
  R2out，Add，Zin

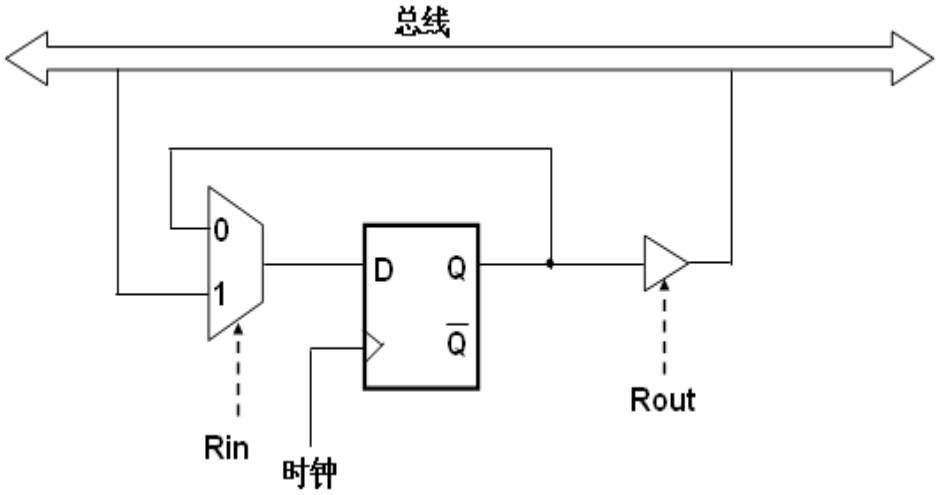
  Zout，R3in

1Cycle?

3Cycles?

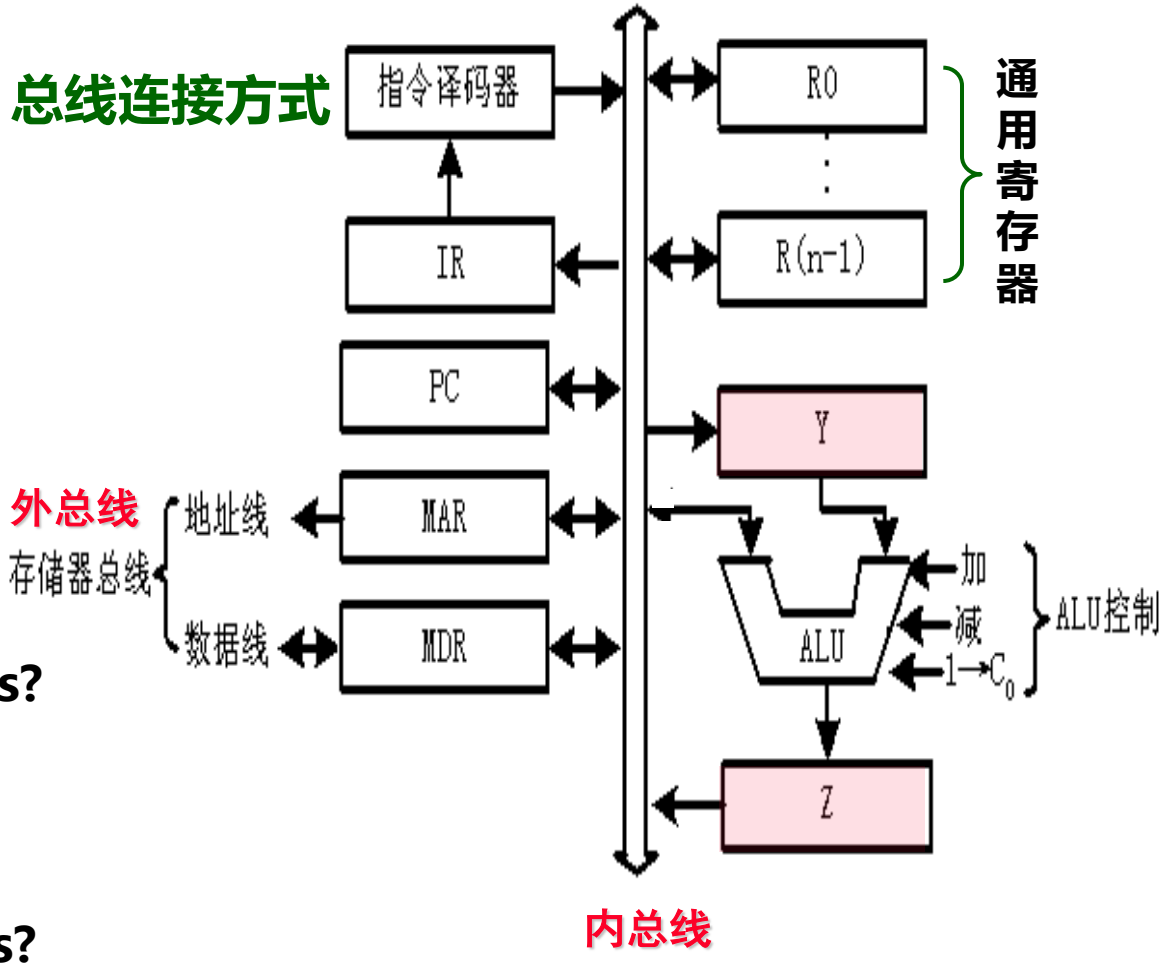
总线连接方式





单总线数据通路

四种基本操作的时序（续）



- 从主存取字 R[R2]←M[R[R1]]
  R1out，MARin　3+？ Cycles?
  Read, WMFC (等待MFC)
  MDRout，R2in
- 写字到主存 M[R[R1]] ← R[R2]
  R1out，MARin　3+？ Cycles?
  R2out，MDRin，
  Write, WMFC

CPU访存有两种通信方式

早期：直接访问MM，“异步”方式，用MFC应答信号；

现在：先Cache后MM，“同步”方式，无需应答信号。

时钟周期的宽度如何确定？以“Riout,OP（Add、Sub、And等）,Zin”还是以“Read/Write”所花时间来确定？

Read/Write时间长，机器周期以此为准！

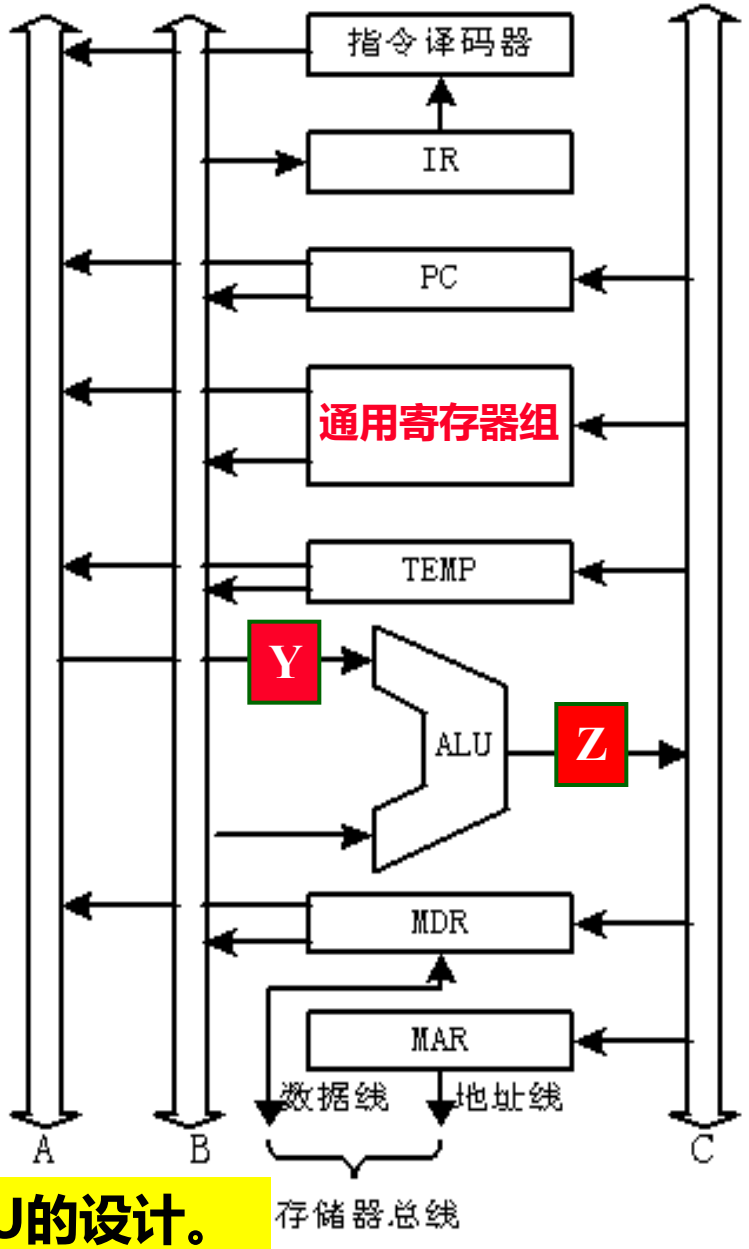
三总线数据通路（自学）

- 单总线中一个时钟内只允许传一个数据，因而指令执行效率很低
- 可采用多总线方式，同时在多个总线上传送不同数据，提高效率
- 例如：三总线数据通路
  - 总线A、B分别传送两个源操作数，总线C传送结果。单总线中的暂存器Y和Z在此可取消。

    三个总线各自传不同的数据，不会发生冲突，故无需Y和Z
  - 采用双口通用寄存器组
  - 如何实现： R[R3] ←R[R1] op R[R2]

    R1outA，R2outB，op，R3inC

    只要一个时钟周期（节拍）即可！



目前大都采用流水线方式执行指令，单总线或三总线的总线式数据通路很难实现指令流水执行。

以下以RISC-V指令系统为例介绍非总线式CPU的设计。

设计处理器的步骤

ISA确定后，进行处理器设计的大致步骤

第一步：分析每条指令的功能，并用RTL(Register Transfer Language)来表示。

第二步：根据指令的功能给出所需的元件，并考虑如何将他们互连。

第三步：确定每个元件所需控制信号的取值。

第四步：汇总所有指令所涉及到的控制信号，生成一张反映指令与控制信号之间关系的表。（真值表）

第五步：根据表得到每个控制信号的逻辑表达式，据此设计控制器电路。

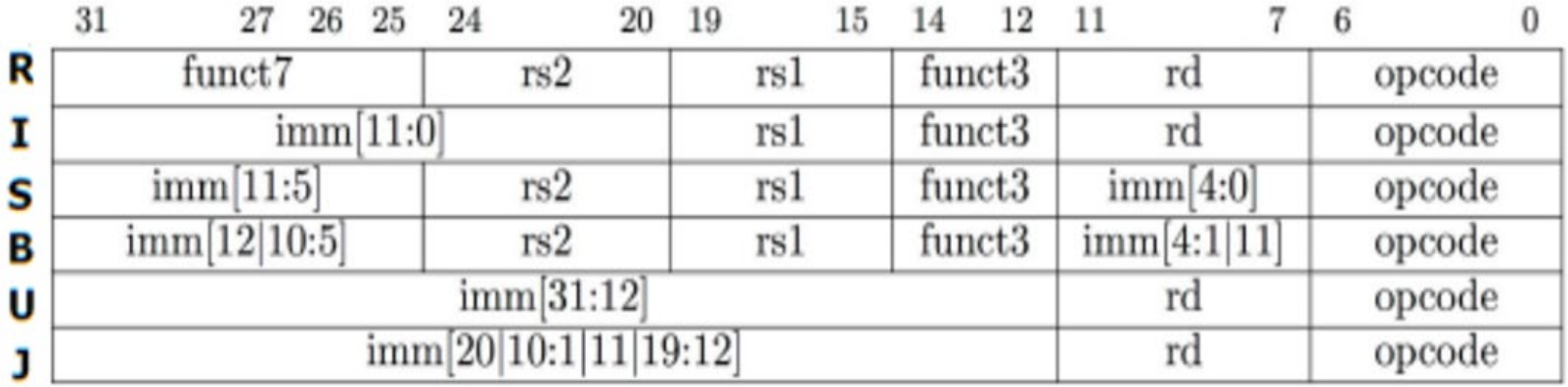
- **处理器设计涉及到数据通路的设计和控制器的设计**
- **数据通路中有两种元件**
  - **操作元件：由组合逻辑电路实现**
  - **存储（状态）元件：由时序逻辑电路实现**

RISC-V指令格式

- **所有指令都是32位宽，须按字地址对齐**

  字地址为4的倍数或2的倍数（RV32C压缩指令格式）！

- **有6种指令格式**



	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	opcode
I	imm[11:0]			rs1	funct3	rd	opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
B	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
J	imm[20|10:1|11|19:12]					rd	opcode

数据通路实现目标包括：

3条R-型指令：add rd, rs1, rs2、slt rd, rs1, rs2、sltu rd, rs1, rs2

2条I-型指令：ori rd, rs1, imm12、lw rd, imm12(rs1)

1条S-型指令：sw rs2, imm12(rs1)

1条B-型指令：beq rs1, rs2, imm12

1条U-型指令：lui rd, imm20

1条J-型指令：jal rd, imm20

具有很强的代表性：

算术/逻辑运算，取数/存数；

短立即数，长立即数；

条件转移，无条件转移；

带符号数判断大小，无符号数判断大小

目标指令功能描述

注意：每条指令的第一步都是取指令并PC加4，使PC指向下条指令（表中除第一条add指令外，其余指令都省略了对第一步的描述）

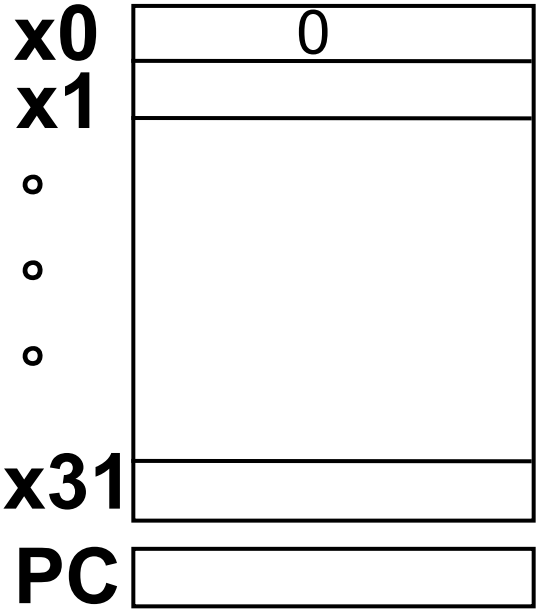
注意：I-型指令的立即数为符号扩展，即使是逻辑运算，立即数也是符号扩展！

指　令	功　能
add rd, rs1, rs2	M[PC]，PC←PC + 4 R[rd]←R[rs1] + R[rs2]
slt rd, rs1, rs2	if (R[rs1] < R[rs2]) R[rd]←1 else R[rd]←0
sltu rd, rs1, rs2	if (R[rs1] < R[rs2]) R[rd]←1 else R[rd]←0
ori rd, rs1, imm12	R[rd]←R[rs1] | SEXT(imm12)
lui rd, imm20	R[rd]←imm20||000H
lw rd, rs1, imm12	Addr←R[rs1] + SEXT(imm12) R[rd]←M[Addr]
sw rs1, rs2, imm12	Addr←R[rs1] + SEXT(imm12) M[Addr]←R[rs2]
beq rs1, rs2, imm12	Cond←R[rs1] – R[rs2] if (Cond eq 0) PC←PC+(SEXT(imm12)×2)
jal rd, imm20	R[rd]←PC + 4 PC←PC + (SEXT(imm20)×2)

符号扩展后左移一位

RV32I指令中寄存器数据和存储器数据的指定

- 寄存器数据指定：
  - 31 x 32-bit GPRs （x0 = 0）
  - 寄存器编号占5 bit
  - PC: 程序计数器（无需编号）
  - 寄存器功能和2种汇编表示方式
- 存储器数据指定
  - 32- bit machine --> 可访问空间: 2^{32}bytes
  - Little Endian(小端方式)



- 只能通过Load/Store指令访问存储器数据
- 数据地址通过一个32位寄存器内容加12位偏移量得到
- 12位偏移量是带符号整数，采用符号扩展
- lw rd, imm12(rs1)， sw rs2, imm12(rs1)

- 数据不要求按边界对齐，执行到一条不按边界对齐的访存指令时，硬件抛出异常，由软件进行处理

SKIP

RV32I寄存器的功能定义和两种汇编表示

BACK to last

寄存器	ABI 名	功能描述	被调用过程保存?
x0	zero	硬编码 0	—
x1	ra	返回地址	否
x2	sp	栈指针	是
x3	gp	全局指针	—
x4	tp	线程指针	—
x5	t0	临时寄存器	否
x6~x7	t1~t2	临时寄存器	否
x8	s0/fp	保存寄存器/帧指针	是
x9	s1	保存寄存器	是
x10~x11	a0~a1	过程参数/返回值	否
x12~x17	a2~a7	过程参数	否
x18~x27	s2~s11	保存寄存器	是
x28~x31	t3~t6	临时寄存器	否

Registers are referenced either by number—x0, … x31, or by name —zero, ra, s1… t0.

功能设计需求的分析

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	opcode
I	imm[11:0]			rs1	funct3	rd	opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
B	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
J	imm[20|10:1|11|19:12]					rd	opcode

除R-型外，其他5类都带有立即数

——立即数扩展器

核心运算类功能的实现 ——ALU

根据PC取指令和PC+4 ——取指令部件

指令的RTL最终实现 ——完整数据通路

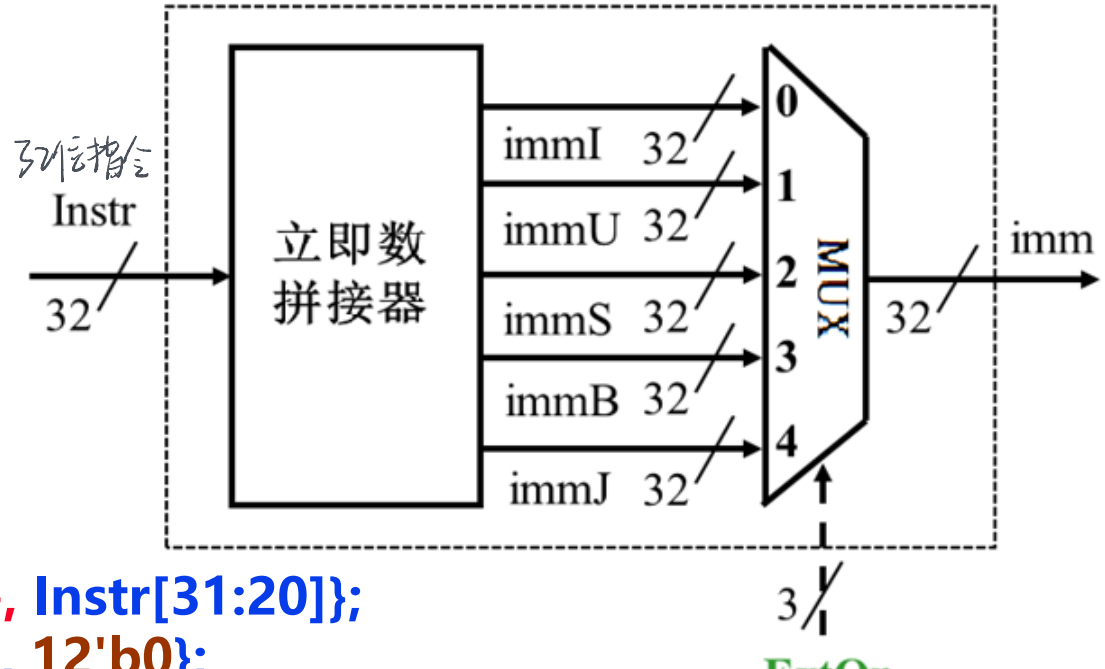
add rd, rs1, rs2	M[PC]，PC←PC + 4 R[rd]←R[rs1] + R[rs2]

扩展器部件的设计

除R-型外，其他5类都带有立即数

立即数拼接器：根据指令格式对指令中的立即数进行拼接和扩展，形成32位立即数

ExtOp：控制选择和输入指令相匹配的立即操作数输出




```
assign immI = {20{Instr[31]}, Instr[31:20]};
assign immU = {Instr[31:12], 12'b0};
assign immS = {20{Instr[31]}, Instr[31:25], Instr[11:7]};
assign immB = {20{Instr[31]}, Instr[7], Instr[30:25], Instr[11:8], 1'b0};
assign immJ = {12{Instr[31]}, Instr[19:12], Instr[20], Instr[30:21], 1'b0};
```

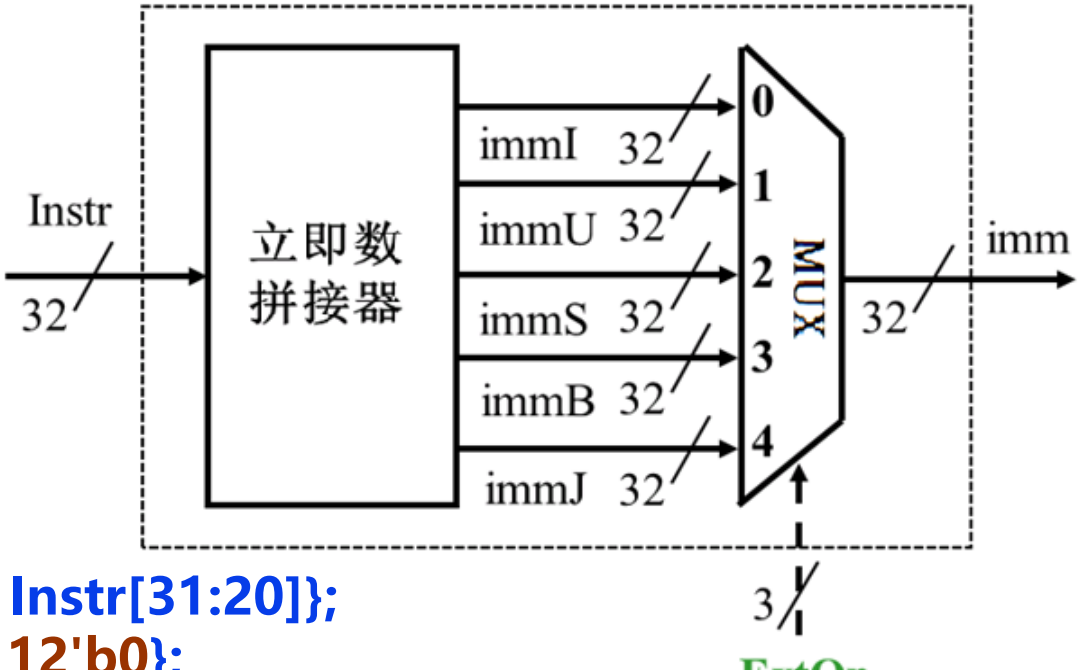
	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	opcode
I	imm[11:0]			rs1	funct3	rd	opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
B	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
J	imm[20|10:1|11|19:12]					rd	opcode

扩展器部件的设计

除R-型外，其他5类都带有立即数

立即数拼接器：根据指令格式对指令中的立即数进行拼接和扩展，形成32位立即数

ExtOp：控制选择和输入指令相匹配的立即操作数输出




```
assign immI = {20{Instr[31]}, Instr[31:20]};
assign immU = {Instr[31:12], 12'b0};
assign immS = {20{Instr[31]}, Instr[31:25], Instr[11:7]};
assign immB = {20{Instr[31]}, Instr[7], Instr[30:25], Instr[11:8], 1'b0};
assign immJ = {12{Instr[31]}, Instr[19:12], Instr[20], Instr[30:21], 1'b0};
```

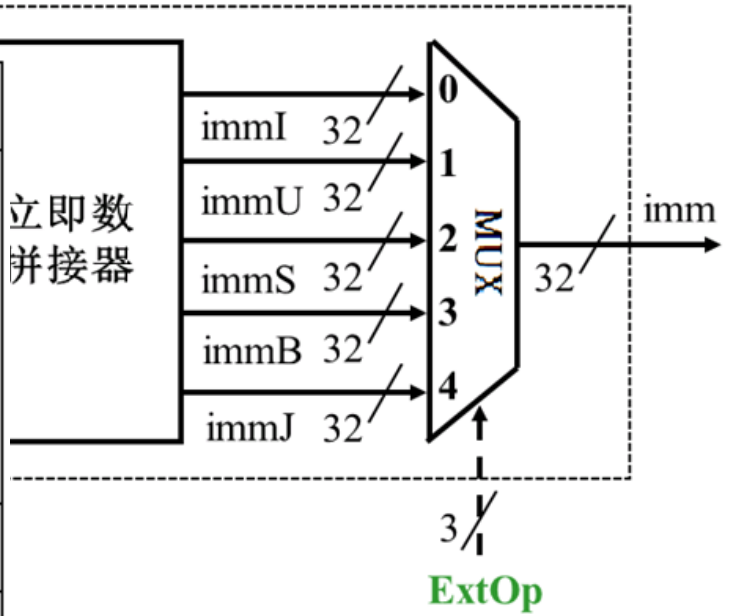
由于所有指令格式是预知的，每种指令中的立即数的位置是固定的，因此拼接器拿到一条32位指令之后，就可以同时完成五种拼接（把这个32位分别当成IUSBJ类指令）

但只有一个是正确的，其它都是错的，最终会按ExtOP输出那个正确的。

——不用等待其它信号，在译码结果（ ExtOP）还没出来之前，就可以把五种立即数拼好。

扩展器部件的设计

指　令	立即数编码类型	ExtOp<2:0>
add rd, rs1, rs2	无立即数	× × ×
slt rd, rs1, rs2		
sltu rd, rs1, rs2		
ori rd, rs1, imm12	I-型立即数（immI）	0 0 0
lui rd, imm20	U-型立即数（immU）	0 0 1
lw rd, rs1, imm12	I-型立即数（immI）	0 0 0
sw rs1, rs2, imm12	S-型立即数（immS）	0 1 0
beq rs1, rs2, imm12	B-型立即数（immB）	0 1 1
jal rd, imm20	J-型立即数（immJ）	1 0 0



问题：各指令的ExtOp取值如何？占几位？

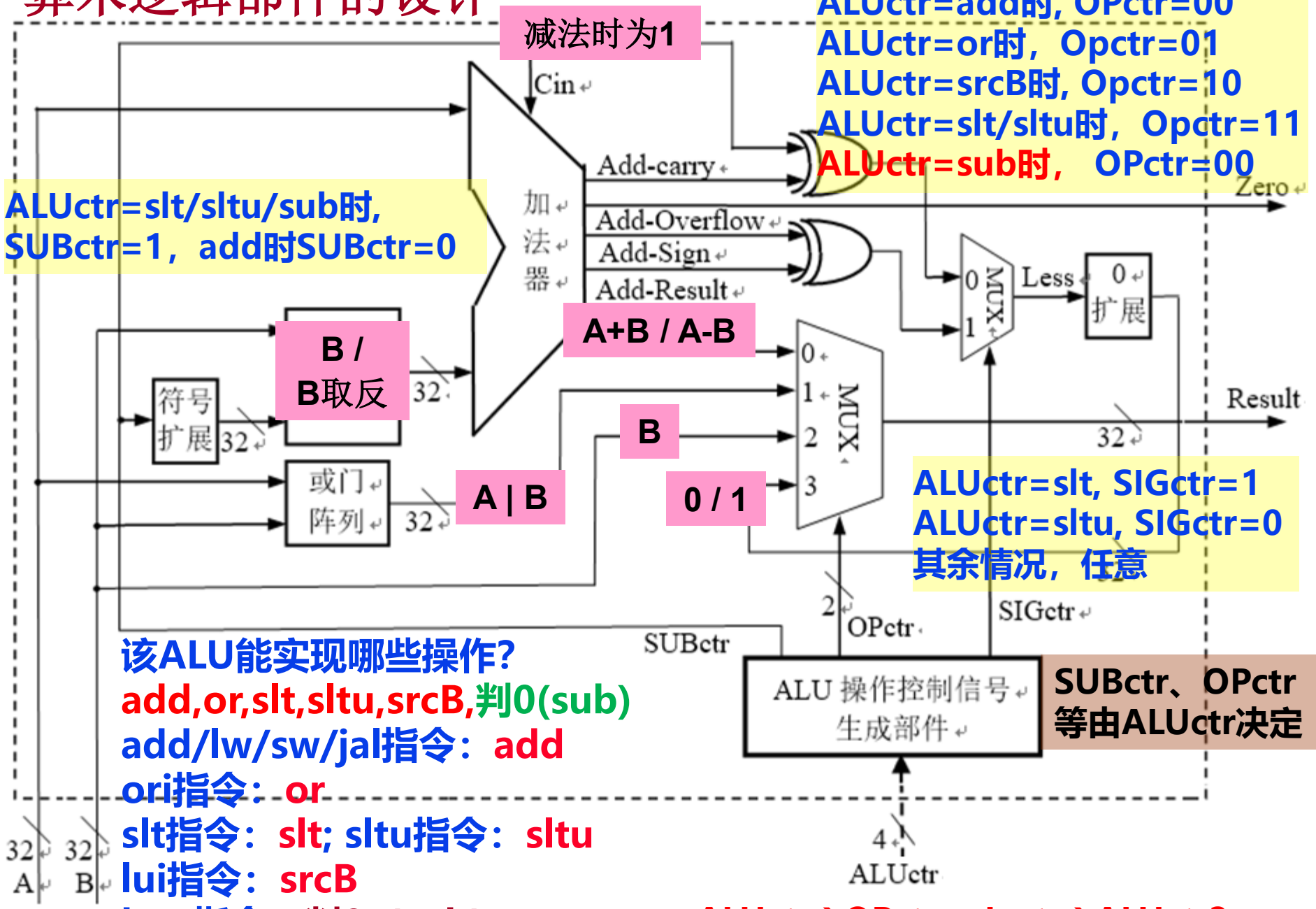
有5种情况，至少占3位！

如何根据指令编码Instr得到ExtOp？

通过指令译码器得到！

将在控制器设计部分介绍。

算术逻辑部件的设计



ALUctr=add时, OPctr=00
ALUctr=or时，Opctr=01
ALUctr=srcB时, Opctr=10
ALUctr=slt/sltu时，Opctr=11
ALUctr=sub时， OPctr=00

ALUctr=slt/sltu/sub时,
SUBctr=1，add时SUBctr=0

ALUctr=slt, SIGctr=1
ALUctr=sltu, SIGctr=0
其余情况，任意

SUBctr、OPctr
等由ALUctr决定

该ALU能实现哪些操作?
add,or,slt,sltu,srcB,判0(sub)
add/lw/sw/jal指令：add
ori指令：or
slt指令：slt; sltu指令：sltu
lui指令：srcB
beq指令：判0（sub）

ALUctr→OPctr, Instr→ALUctr?

算术逻辑部件的设计

(1) ALUctr如何决定OPctr等?

看操作和SUBctr、OPctr等的关系

ALUctr=slt/sltu/sub时, SUBctr=1，add时SUBctr=0

ALUctr=slt, SIGctr=1
ALUctr=sltu, SIGctr=0
其余情况，任意

ALUctr=add时, OPctr=00
ALUctr=or时，Opctr=01
ALUctr=srcB时, Opctr=10
ALUctr=slt/sltu时，Opctr=11
ALUctr=sub时， OPctr=00

ALUctr<3:0>	操作类型	SUBctr	SIGctr	OPctr<1:0>
0 0 0 0	add	0	×	0 0
0 0 0 1	（未用）			
0 0 1 0	slt	1	1	1 1
0 0 1 1	sltu	1	0	1 1
0 1 0 0	（未用）			
0 1 0 1	（未用）			
0 1 1 0	or	×	×	0 1
0 1 1 1	（未用）			
1 0 0 0	sub	1	×	0 0
其余	（未用）			
1 1 1 1	srcB	×	×	1 0

SUBctr = (~ALUctr<3> & ~ALUctr<2> & ALUctr<1>) | ALUctr<3>

SIGctr = ~ALUctr<0>

OPctr<1> = (~ALUctr<3> & ~ALUctr<2> & ALUctr<1>) |
(ALUctr<3> & ALUctr<2> & ALUctr<1> & ALUctr<0>) **(slt, sltu)**

OPctr<0> = (~ALUctr<3> & ~ALUctr<2> & ALUctr<1>) |
(~ALUctr<3> & ALUctr<2> & ALUctr<1> & ~ALUctr<0>) **(or)**

(2) 指令Instr如何决定ALUctr?

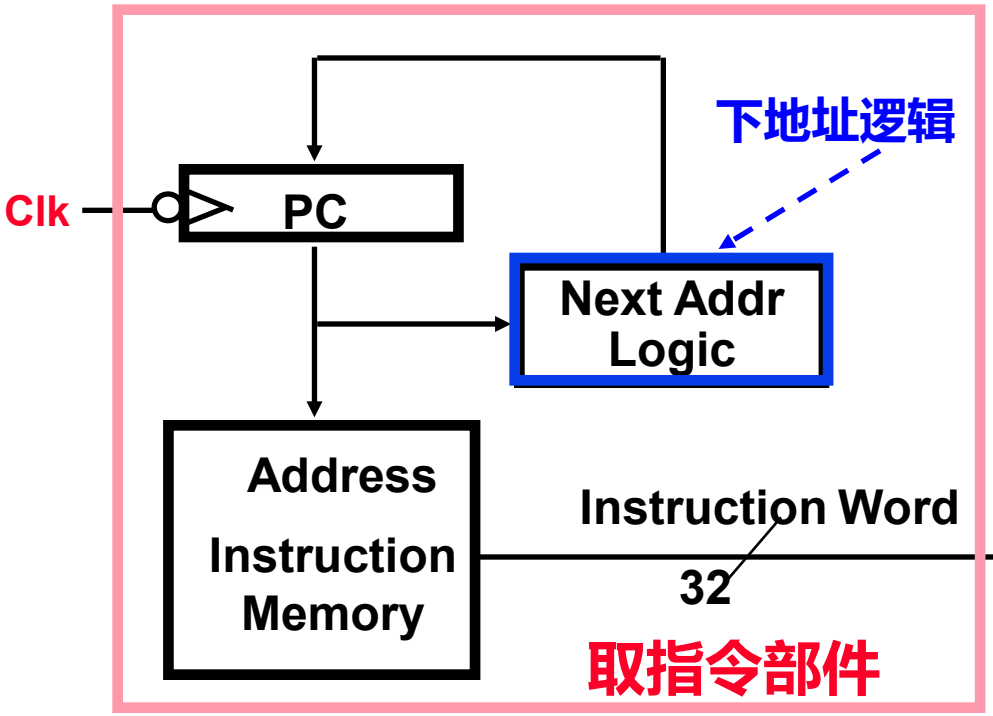
通过指令译码器得到!

在控制器设计部分介绍

取指令部件(Instruction Fetch Unit)的设计

- 每条指令都有的公共操作：
  - 取指令： M[PC]
  - 更新PC：PC ← PC + 4

转移（Branch and Jump）时，PC内容再次被更新为 “转移目标地址”



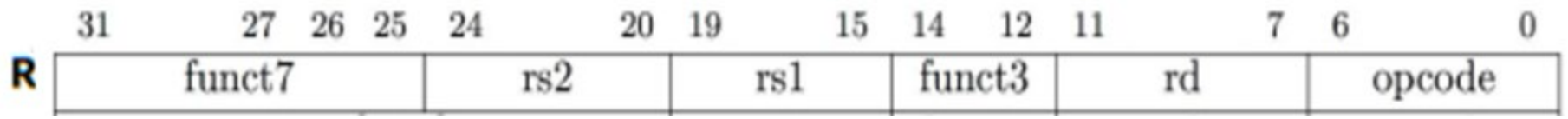
先取指令，再改PC的值（具体实现时，可以并行）

绝不能先改PC的值，再取指令

取指后，各指令功能不同，数据通路中信息流动过程也不同

下面分别对每条指令进行相应数据通路的设计

R-型指令的数据通路



R-型指令功能：

R[rd] ← R[rs1] op R[rs2]

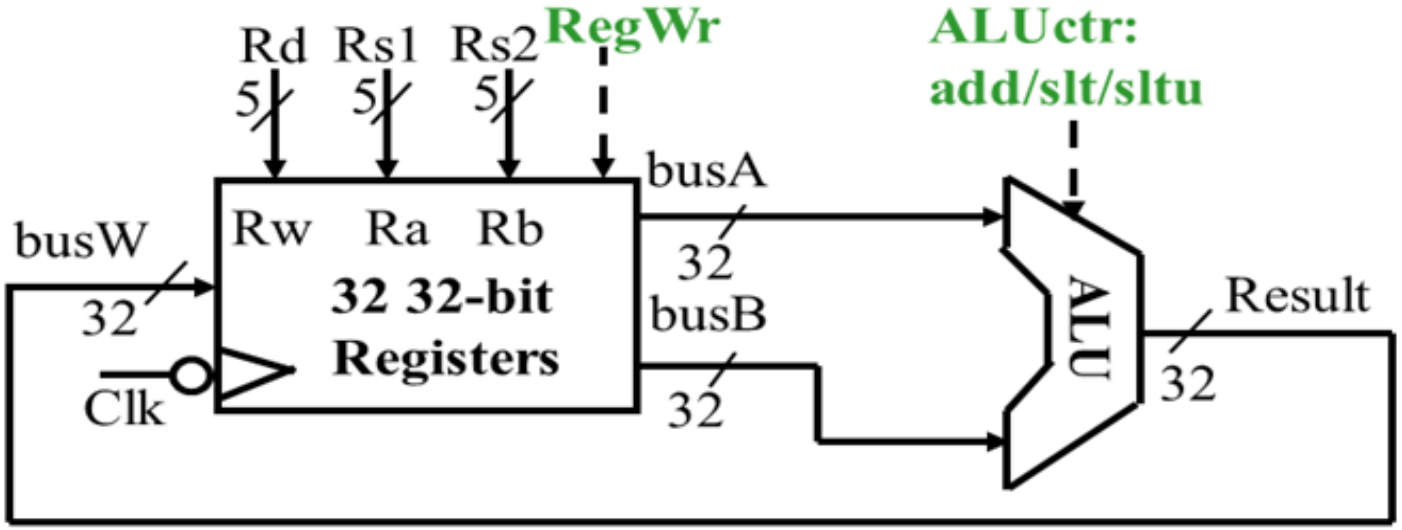
根据PC读取指令

以下相应字段送控制器

- **操作码opcode**
- **功能码funct7和funct3**

以下相应字段送寄存器堆

- **rs1送Rs1输入端**
- **rs2送Rs2输入端**
- **rd送Rd输入端**

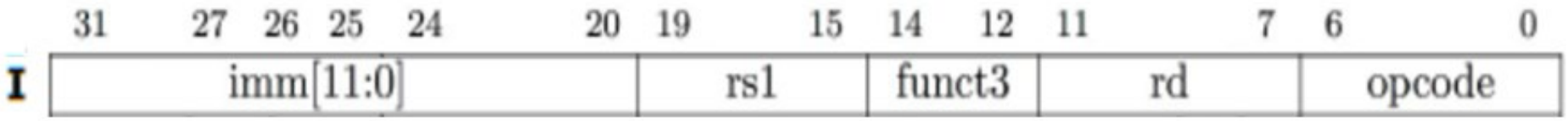


三条R-型指令：add、slt和sltu，分别对应ALU的三种操作，

即ALUctr为add、slt和sltu（比较结果为0（>=）或1（<））。

这三条指令都要写结果，故三条指令对应的控制信号RegWr=1

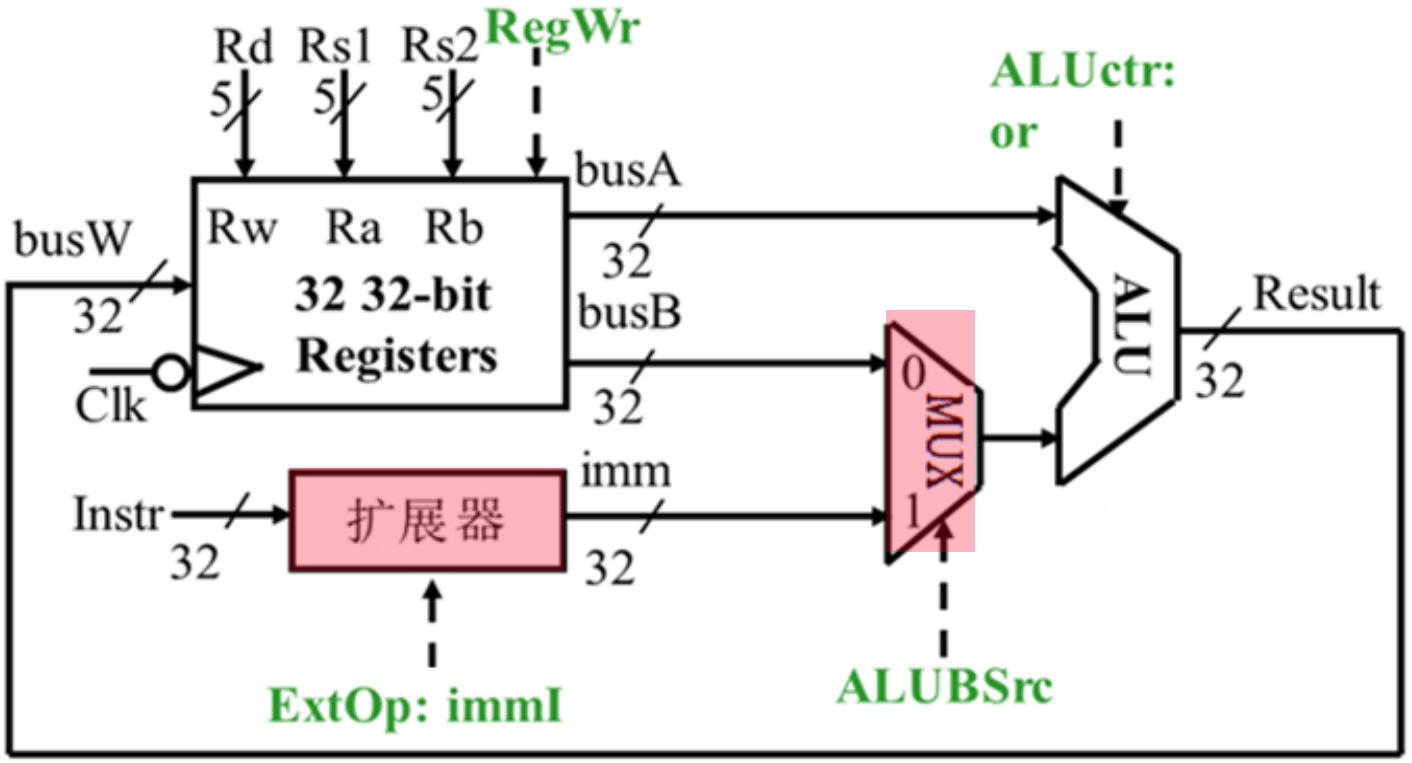
I-型运算指令ori的数据通路



I-型指令 ori 功能：R[rd] ← R[rs1] or SEXT(imm12)

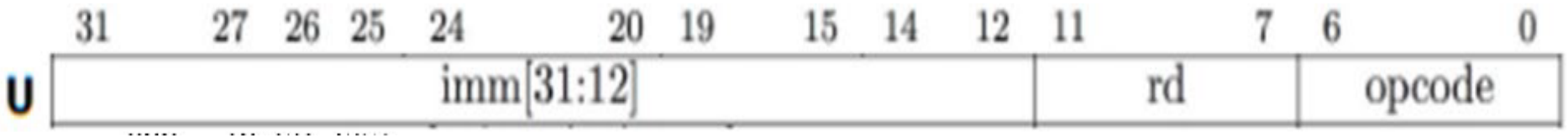
多了一个扩展器和多路选择器MUX

新增的控制信号在执行R型指令时，也需要给出正确的取值！下同

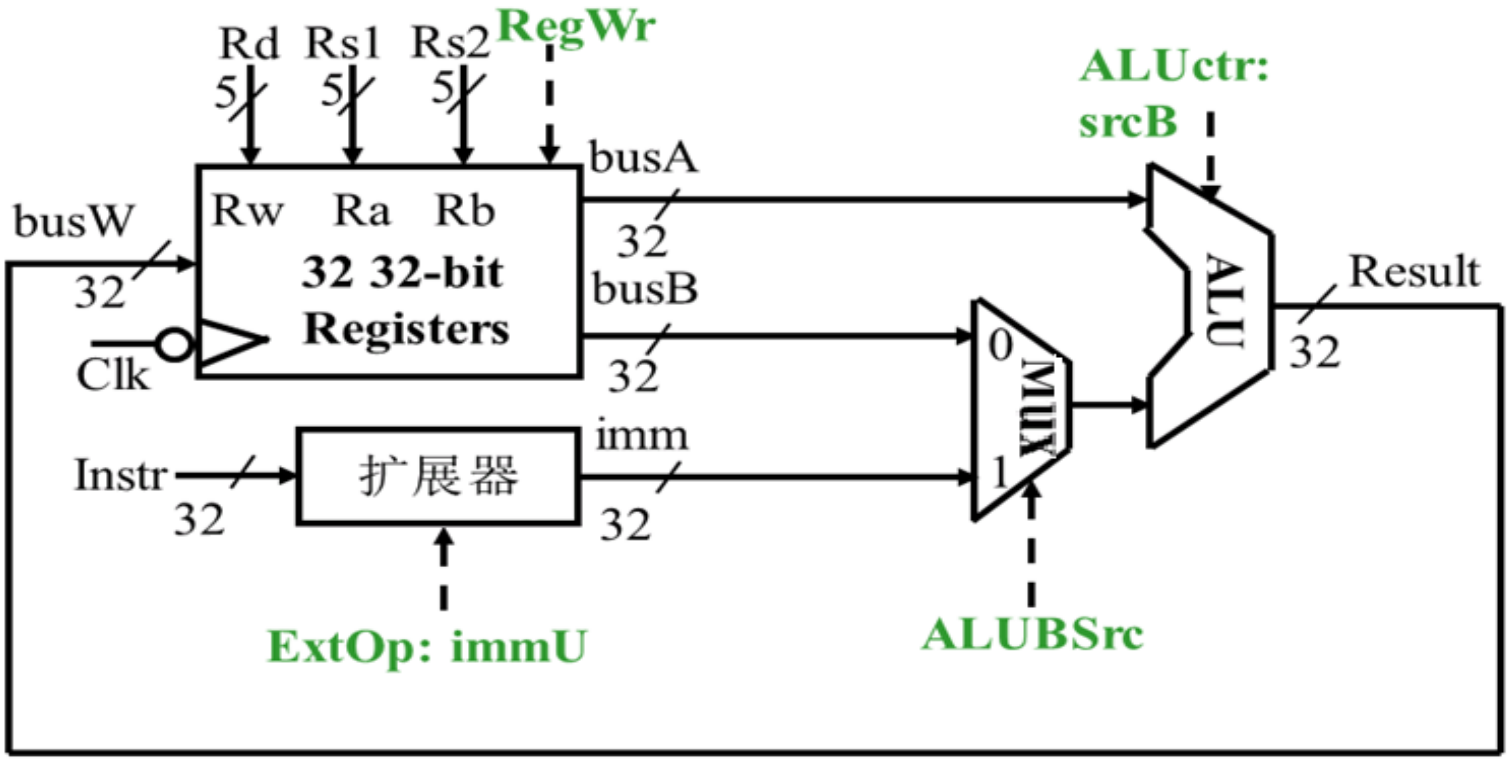


9条目标指令中，ori指令为I-型，其ALUctr为or。第二操作数为I-型立即数 immI ，ExtOp取值为000，ALUBSrc=1，RegWr=1

U-型指令的数据通路



U-型指令lui的功能：R[rd]← imm20||000H 即直接将扩展器结果输出

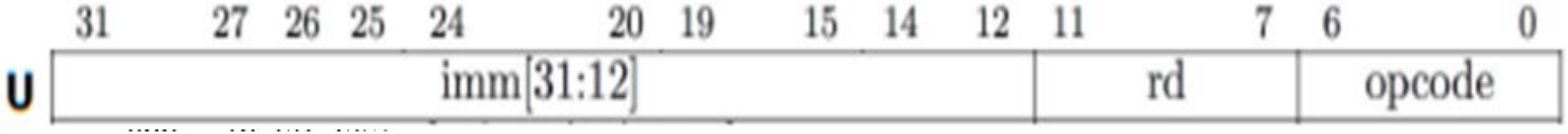


思考：

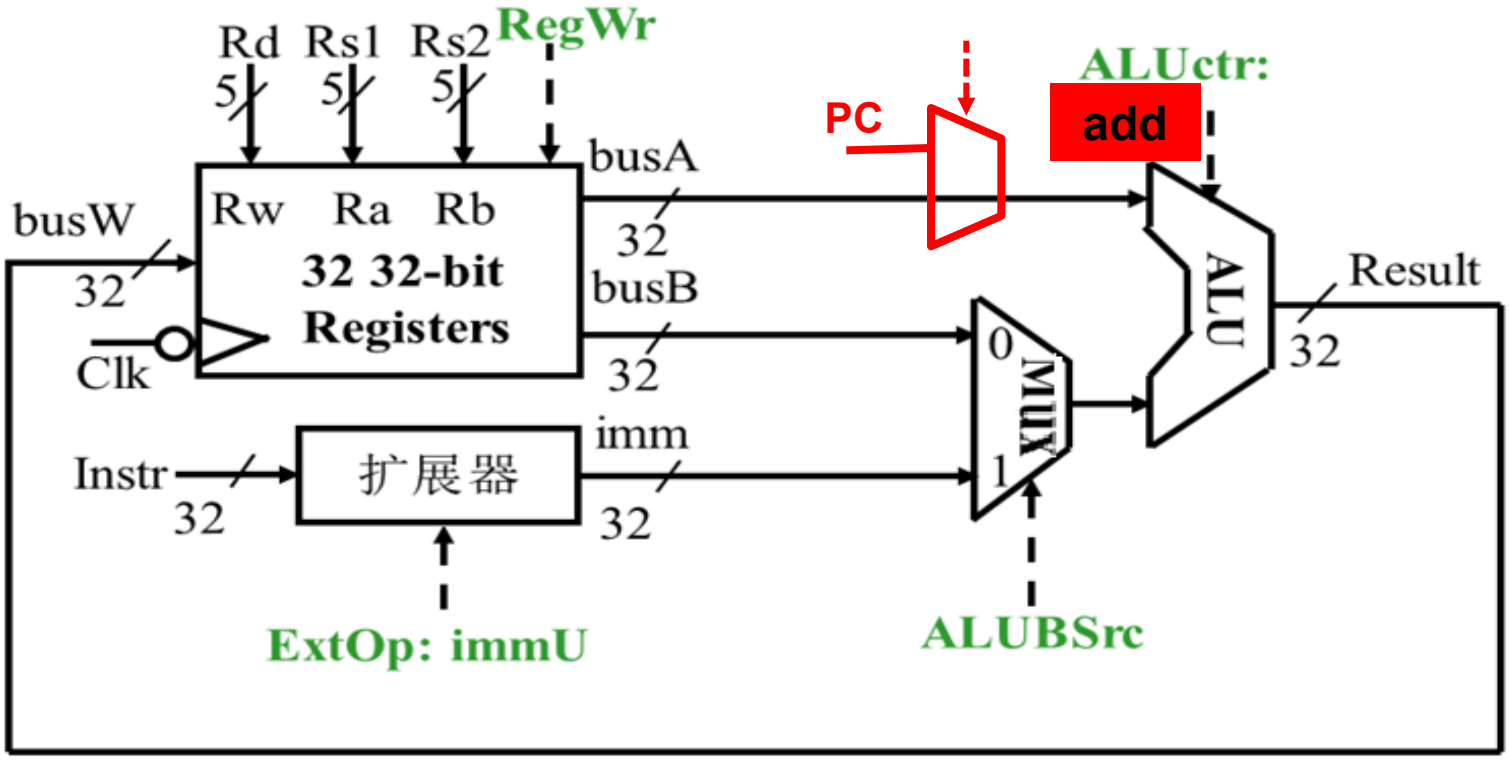
auipc的数据通路?

9条目标指令中，lui指令为U-型，其ALUctr为srcB 。第二操作数为U-型立即数 immU ，ExtOp取值为001，ALUBSrc=1，RegWr=1

U-型指令的数据通路（auipc)



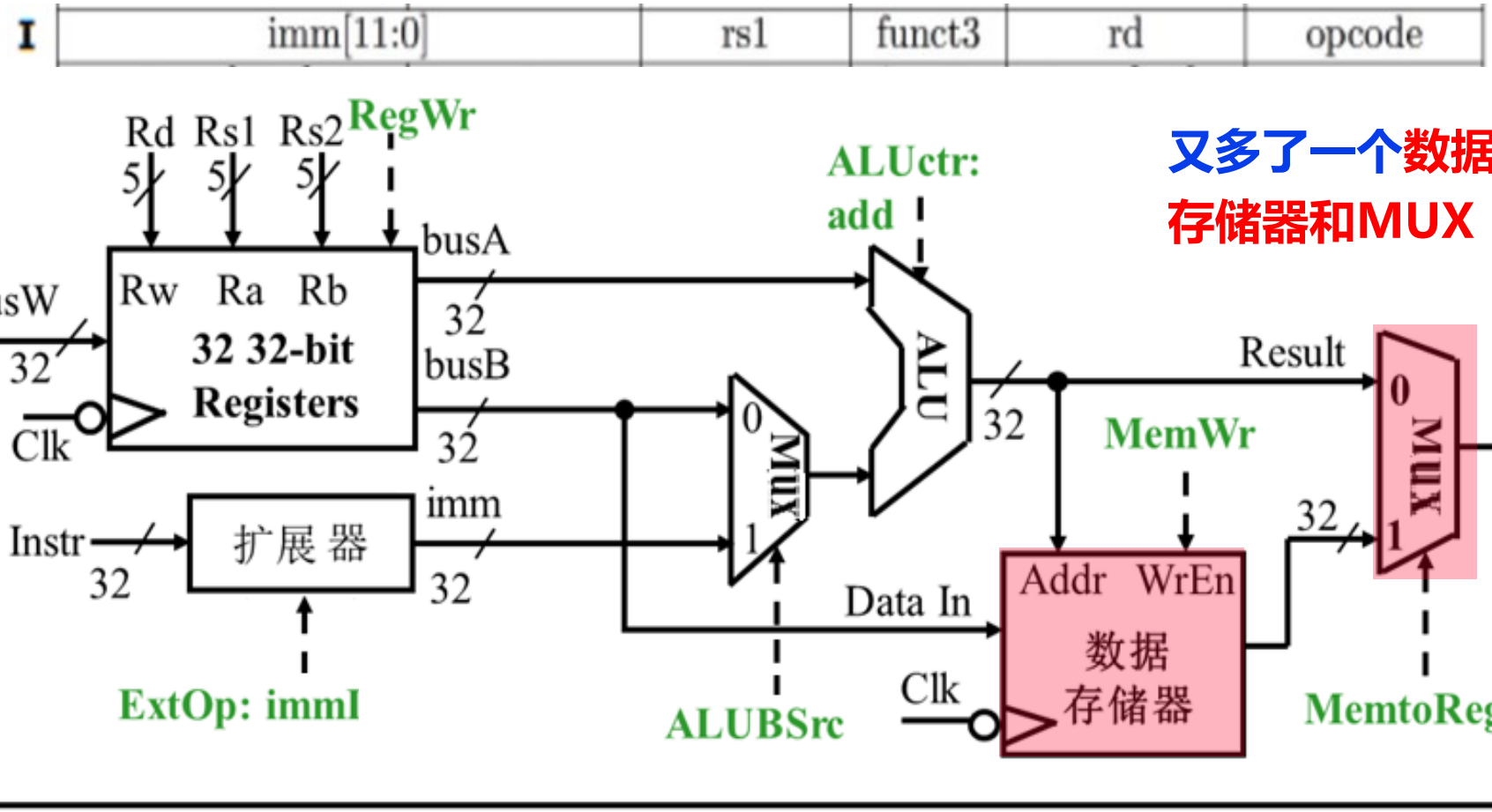
指令auipc的功能：R[rd]← PC+imm20||000H



auipc指令为U-型，其ALUctr为add。第二操作数为U-型立即数 immU，
ExtOp取值为001，ALUBSrc=1，RegWr=1，...

Load指令的数据通路

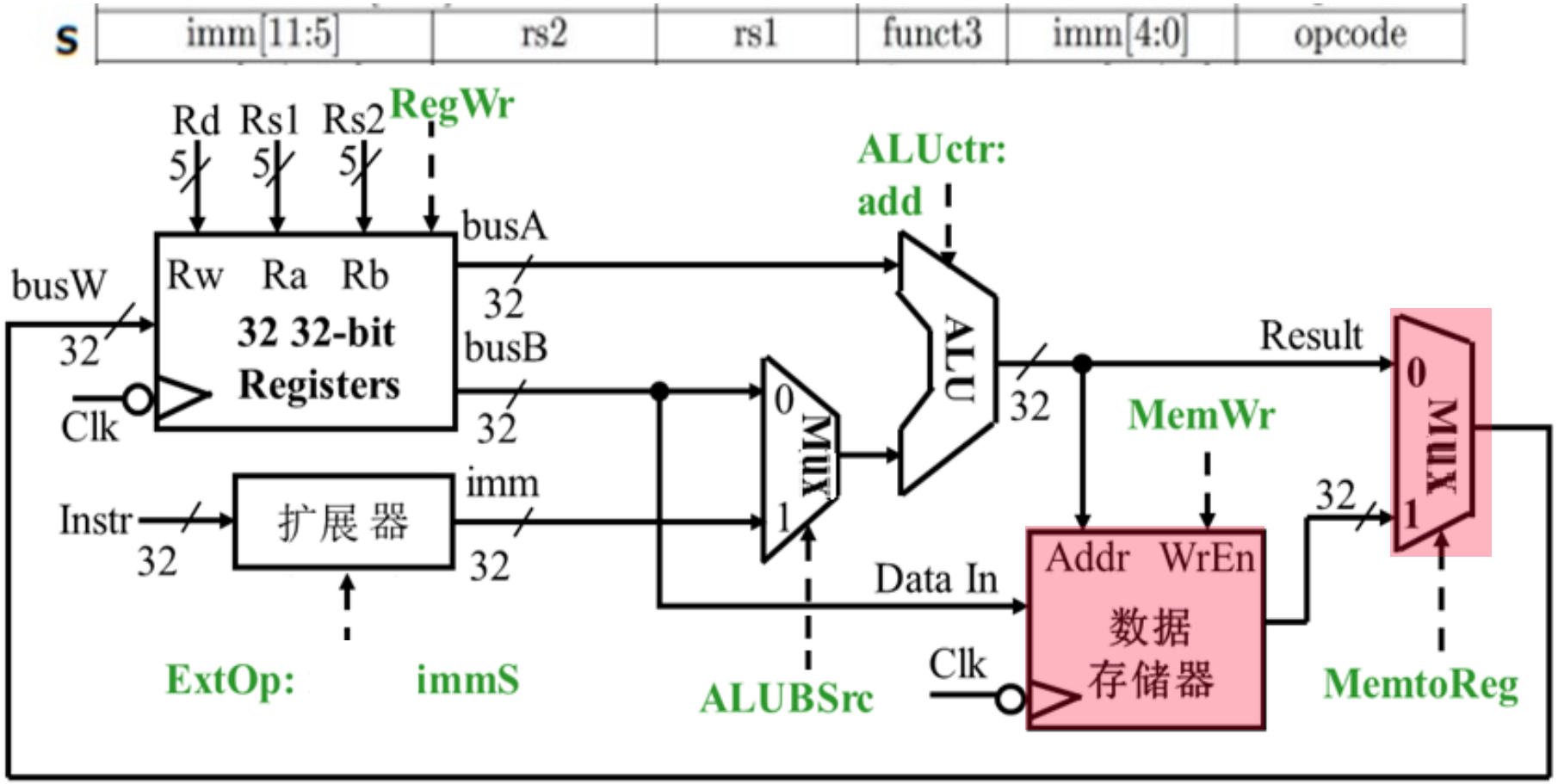
I-型的lw指令的功能：R[rd]← M[R[rs1]+SEXT(imm12)]



ALUctr是add，第二操作数为immI，故ExtOp为000；ALUBSrc=1；MemWr为0；MemtoReg为1；RegWr为1

Store指令的数据通路

S-型的sw指令的功能：M[R[rs1]+SEXT(imm12)]←R[rs2]

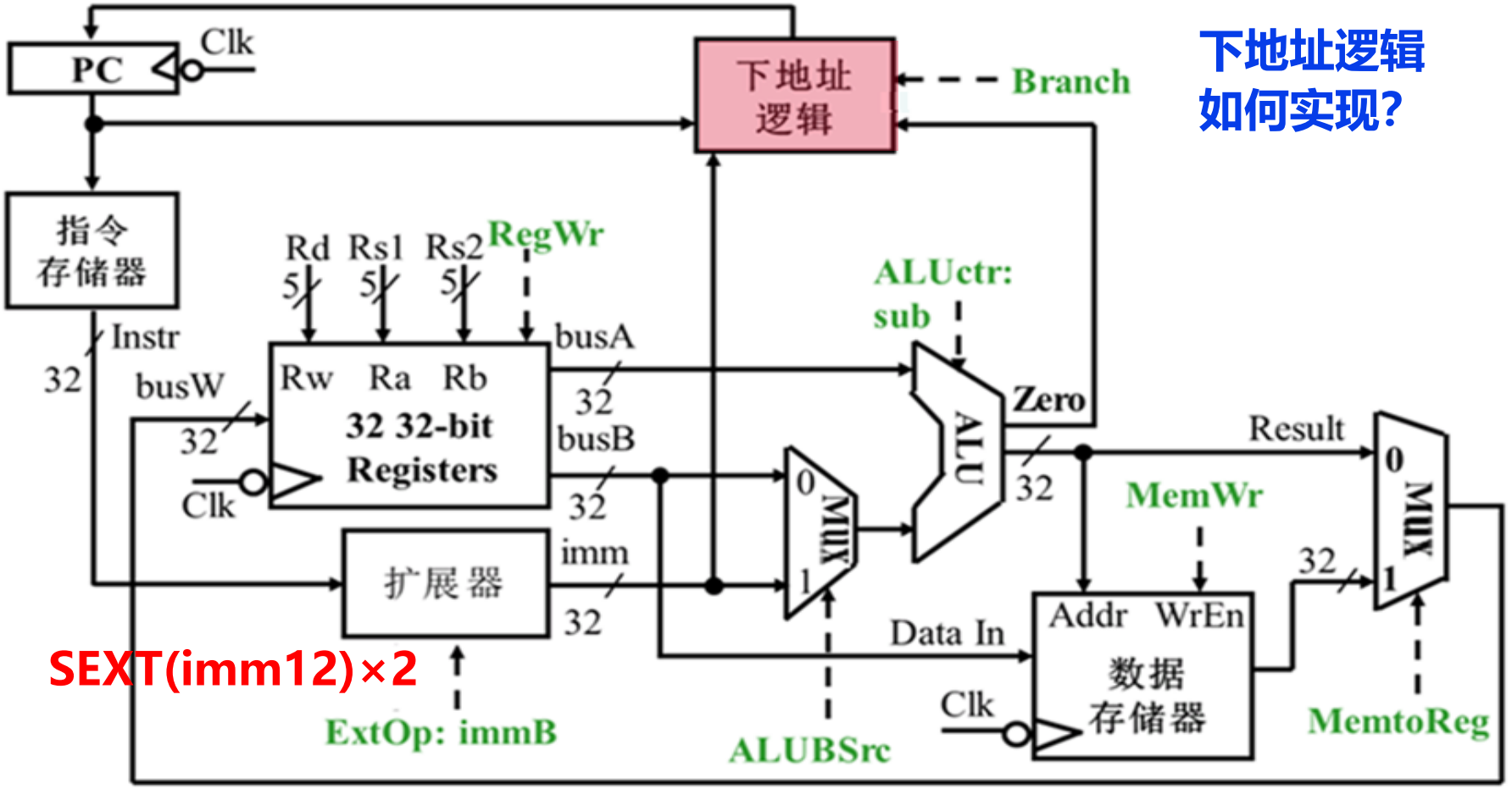


ALUctr是add，第二操作数为immS，故ExtOp为010；ALUBSrc=1；MemWr为1；MemtoReg为x（x表示任意）；RegWr为0

B-型指令的数据通路

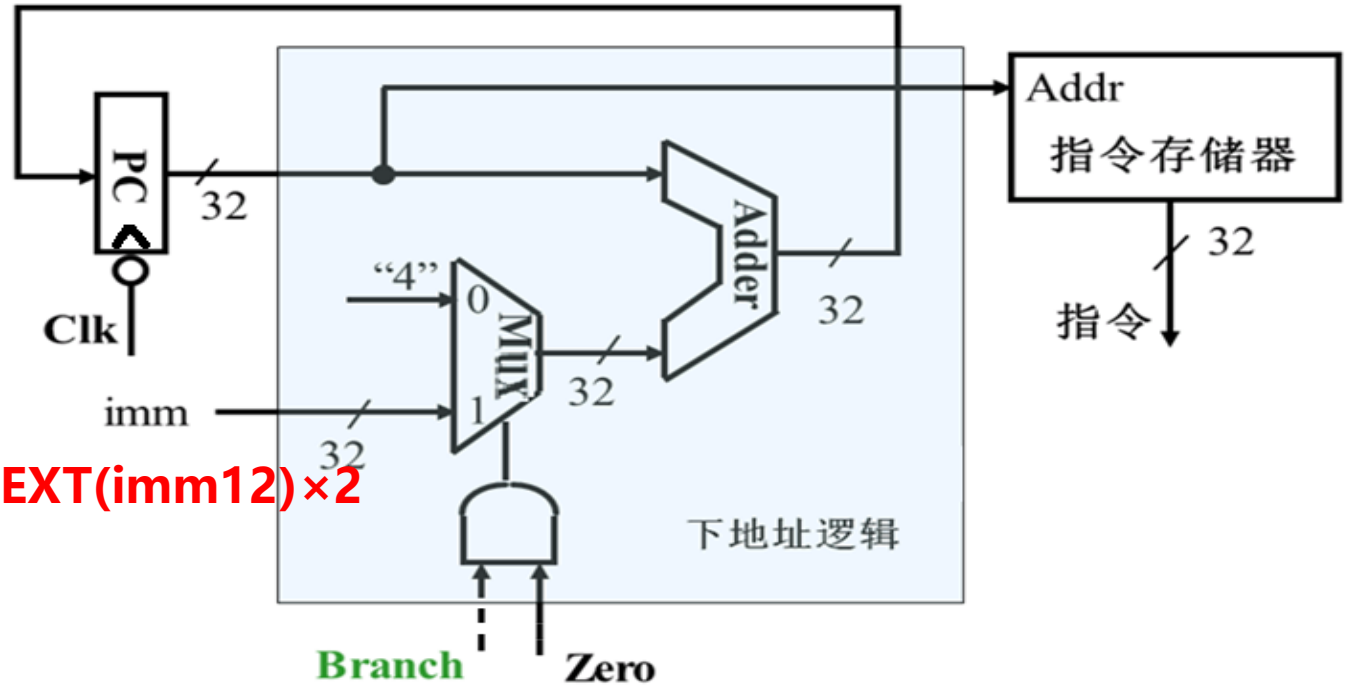
B-型的beq指令的功能是：

if (R[rs1]=R[rs2]) PC←PC+(SEXT(imm12)×2) else PC←PC+4

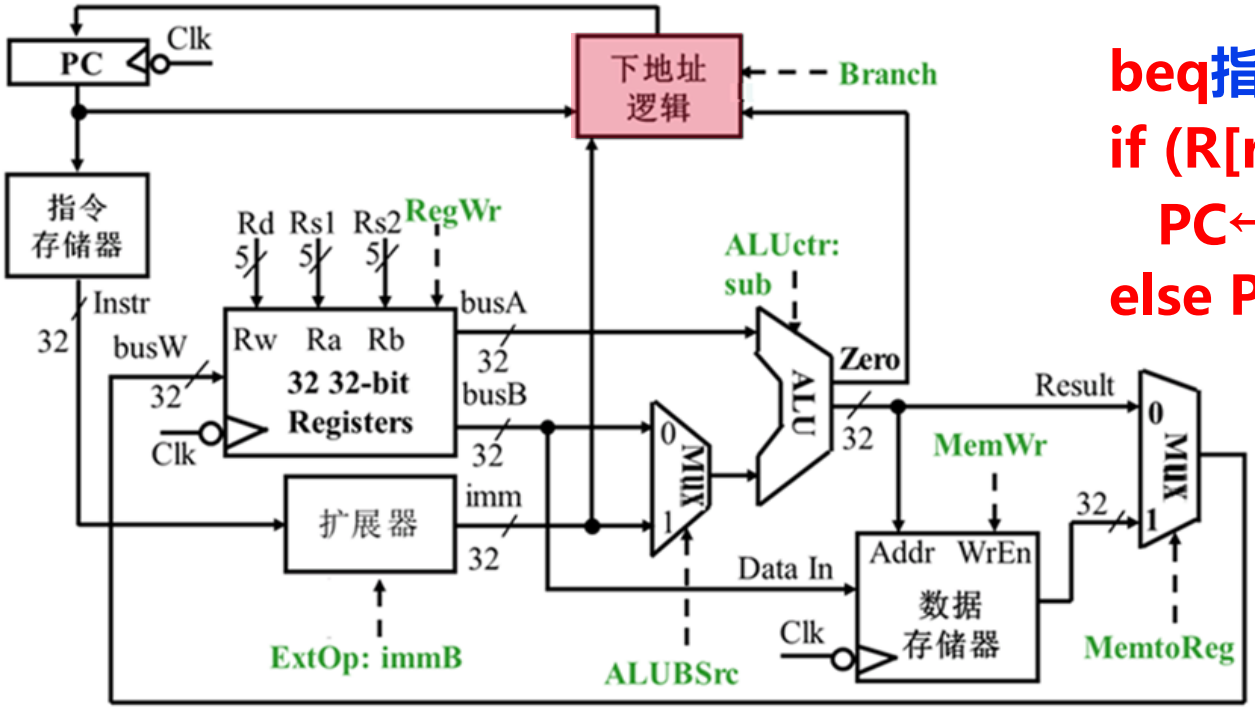


ALUctr是sub，立即数为immB，故ExtOp=011；ALUBSrc=0；
MemWr=0；MemtoReg=x；RegWr=0；Branch=1

B-型指令的数



下地址逻辑如何实现?



beq指令的功能是:

if (R[rs1]=R[rs2])

PC←PC+(SEXT(imm12)×2)

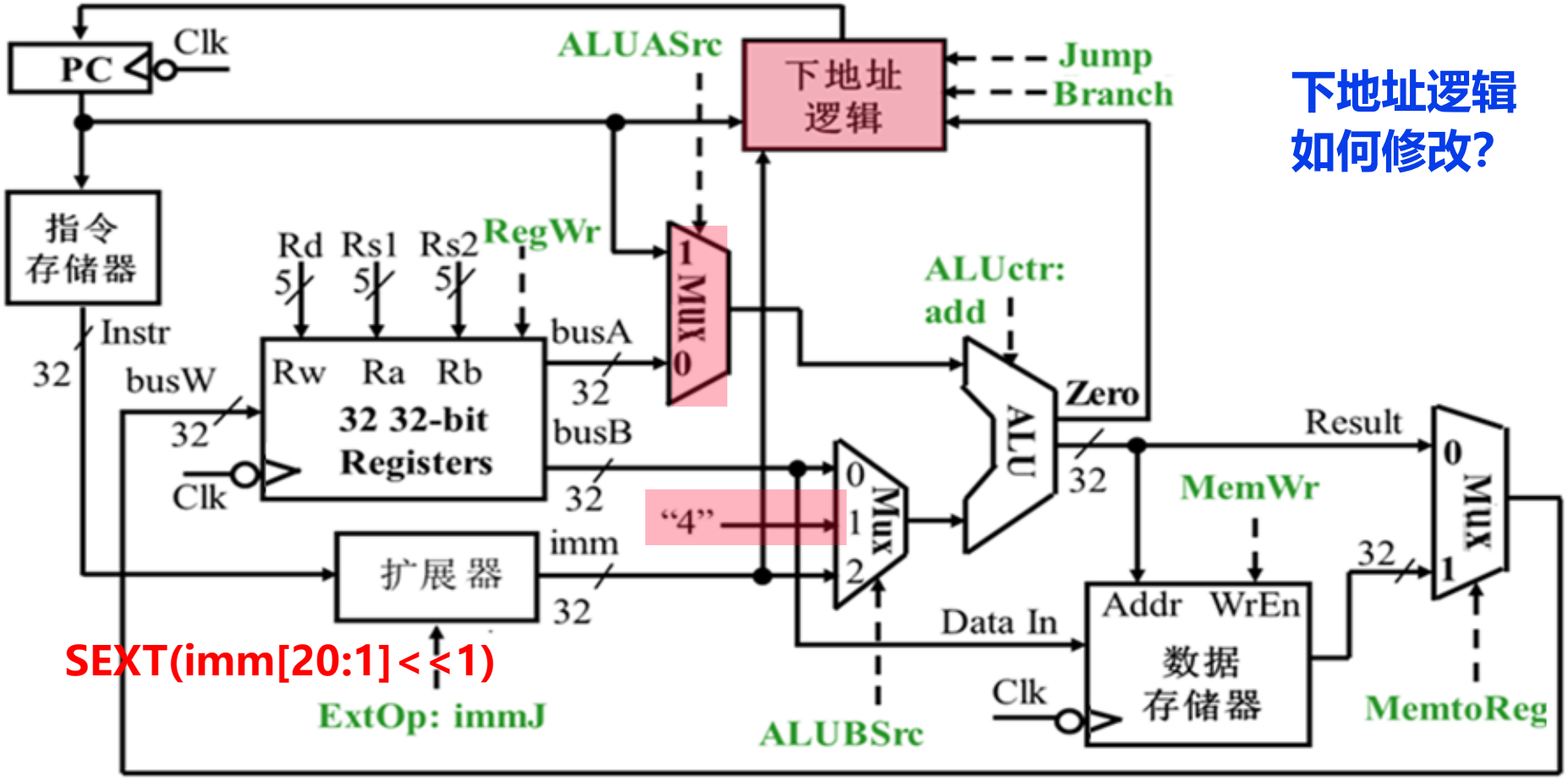
else PC←PC+4

J-型指令的数据通路

比前面的数据通路多了个MUX和“4”

jal是唯一的一条J-型指令，其功能是：

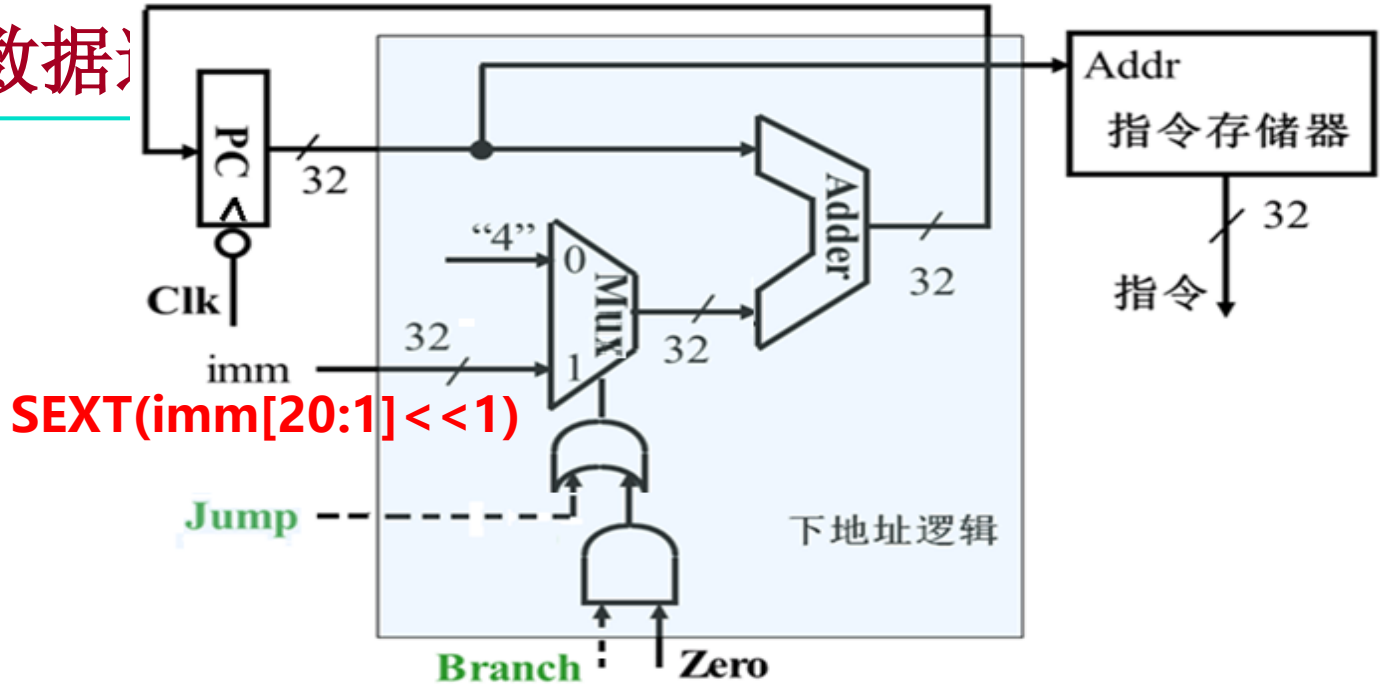
PC←PC+SEXT(imm[20:1]<<1)；R[rd]←PC+4



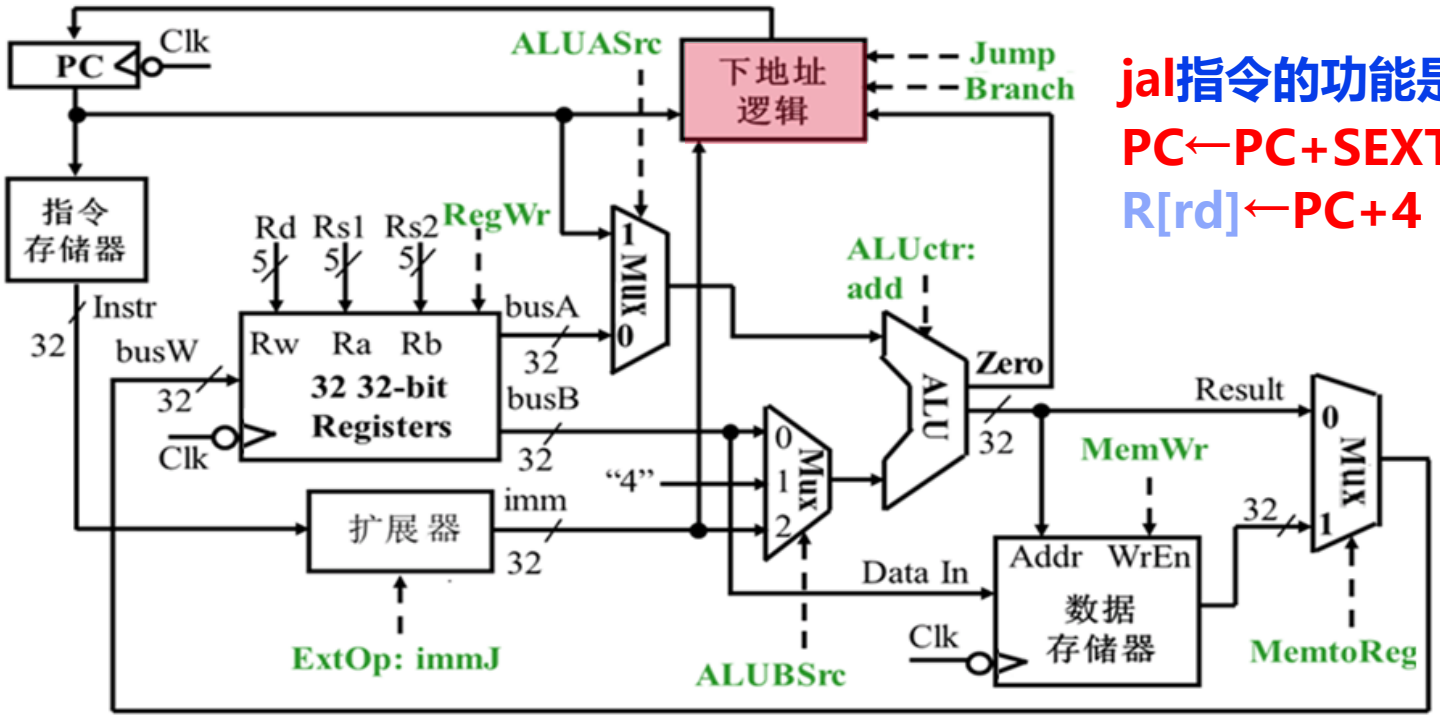
ALUctr是add，立即数为immJ，故ExtOp=100； ALUASrc=1； ALUBSrc=01；

MemWr=0；MemtoReg=0；RegWr=1；Branch=0；Jump=1

J-型指令的数据

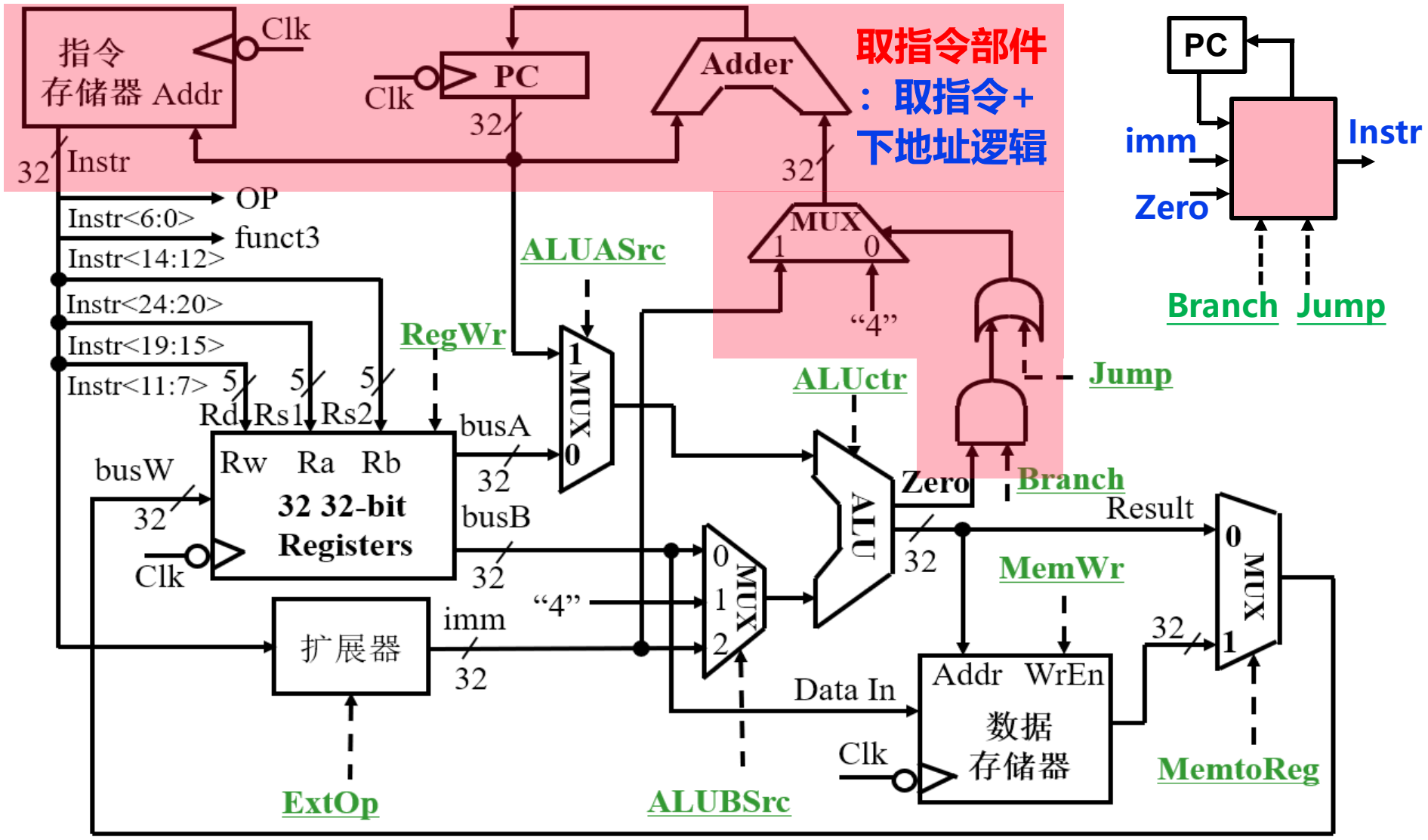


下地址逻辑
如何修改?



jal指令的功能是:
PC←PC+SEXT(imm[20:1]<<1);
R[rd]←PC+4

一个完整的单周期数据通路



指令执行结果总是在下个时钟到来时开始保存在 寄存器 或 存储器 或 PC 中!

下一讲考虑：如何产生控制信号！ （控制器的设计内容）

第二讲小结

- CPU设计直接决定了时钟周期宽度和CPI，所以对计算机性能非常重要!
- CPU主要由数据通路和控制器组成
  - 数据通路：实现指令集中所有指令的操作功能
  - 控制器：控制数据通路中各部件进行正确操作
- 数据通路中包含两种元件
  - 操作元件（组合电路）：ALU、MUX、扩展器、Adder、Reg/Mem Read等
  - 状态 / 存储元件（时序电路）：PC、Reg/Mem Write
- 数据通路的定时
  - 数据通路中的操作元件没有存储功能，其操作结果必须写到存储元件中
  - 在时钟到达后clk-to-Q时存储元件开始更新状态
- RV32I指令集的一个子集作为CPU的实现目标
  - 公共操作：取指令和PC+4
  - 下址计算：PC，三路选择：顺序、Branch（结合标志Zero）、Jump
  - R-型：ALU两个操作数来自rs1和rs2，结果写到rd
  - 访存：符号扩展，数据在rt和主存单元中交换
  - 立即数：不同方式的扩展操作数imm送到ALU的一个输入端

第8章 中央处理器（一）

第一讲 中央处理器概述

第二讲 单周期数据通路的设计

第三讲 单周期控制器的设计

第四讲 多周期处理器的设计

第五讲 流水线处理器设计

第六讲 流水线冒险及其处理

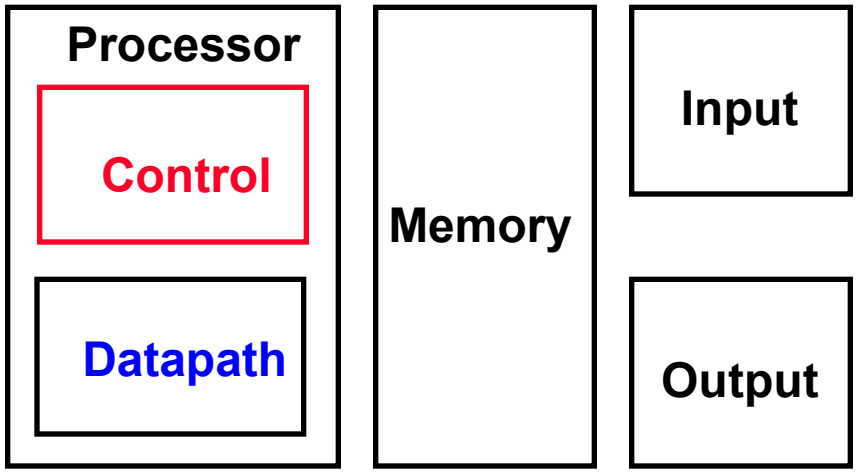
第七讲 高级流水线技术

第三讲 单周期控制器的设计

主 要 内 容

- 考察每条指令在数据通路中的执行过程和涉及到的控制信号的取值
  - 公共操作：取指令和计算下址PC
  - R-型指令（add / slt / sltu）
  - I-型运算类指令 （ori）
  - U-型指令（lui）
  - 访存指令（lw / sw）
  - B-型指令 （beq）
  - J-型指令 （jal）
- 汇总各指令的控制信号取值
- 设计主控制单元

单周期控制器设计



设计方法：

1） 根据每条指令的功能，分析控制信号的取值，并在表中列出。

2） 根据列出的指令和控制信号的关系，写出每个控制信号的逻辑表达式。

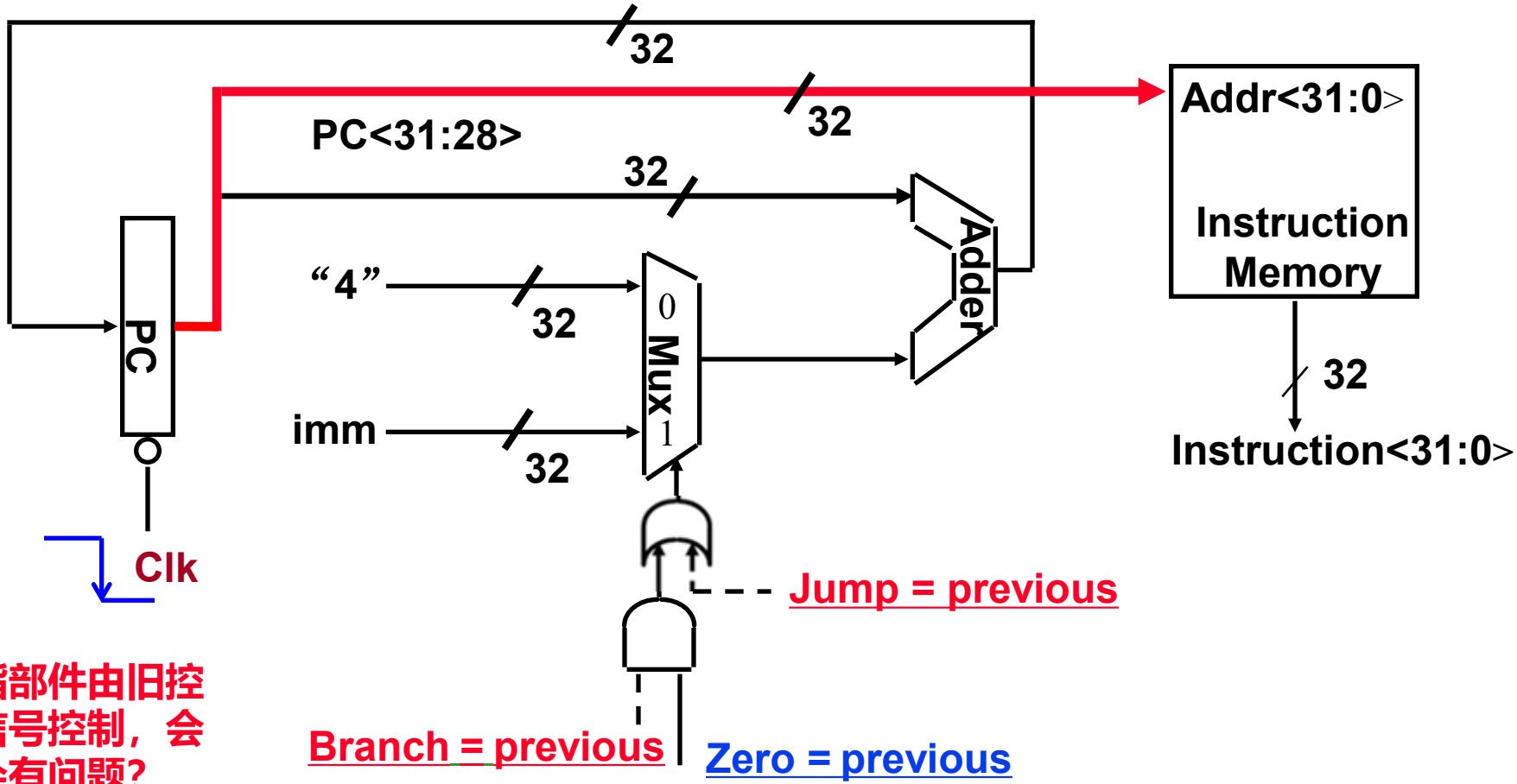
指令开始时取指部件中的动作

取指令: Instruction ← M[PC]

- 所有指令都相同

新指令还没有取出译码，所以控制信号的值还是原来指令的旧值。

新指令还没有执行，所以标志也为旧值。



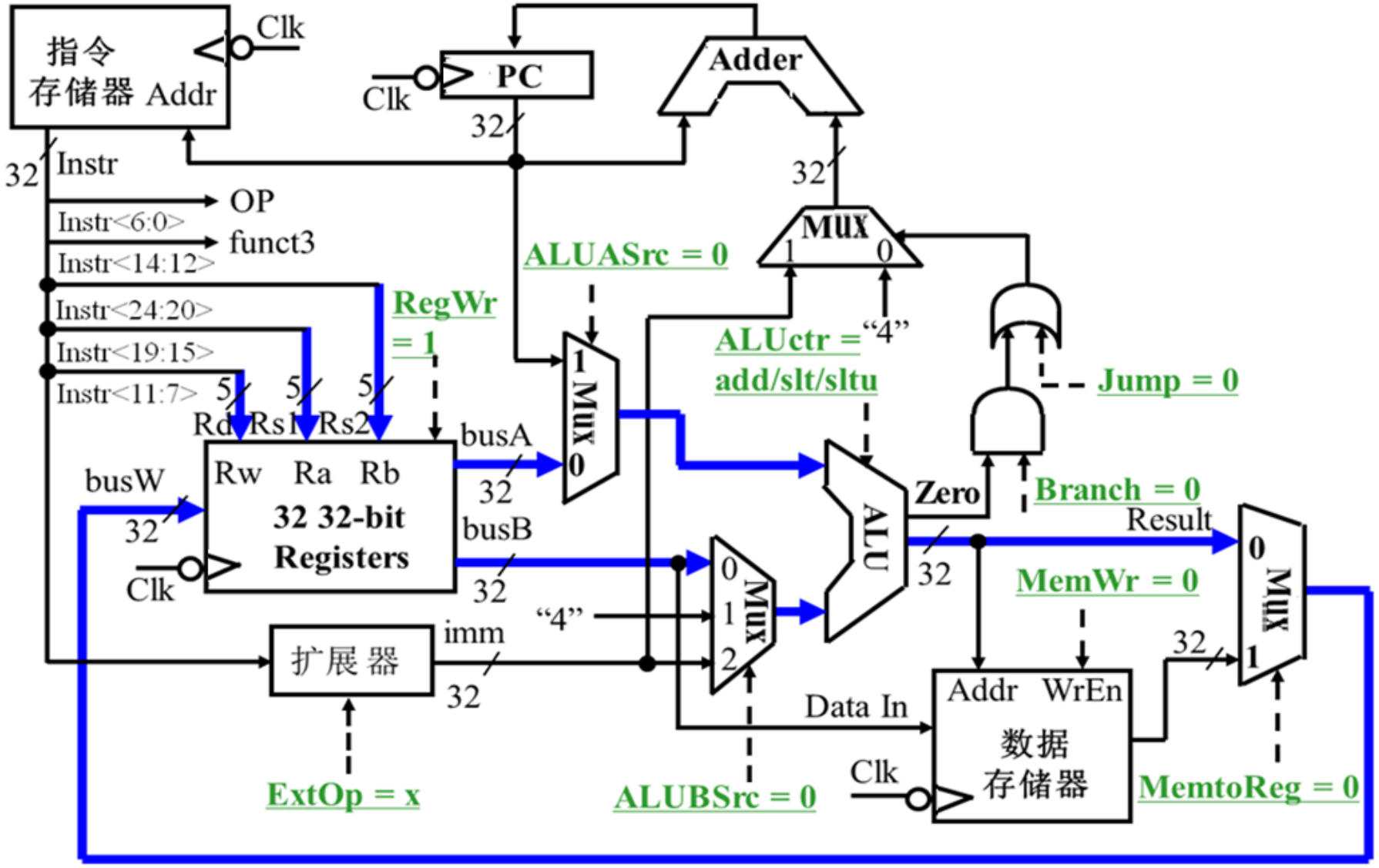
取指部件由旧控制信号控制，会不会有问题?

没有问题！Why?

因为在下个Clk到来之前PC输入端的值不会写入，只要保证下个Clk来之前能产生正确的PC即可！

指令译码后R-型指令操作过程

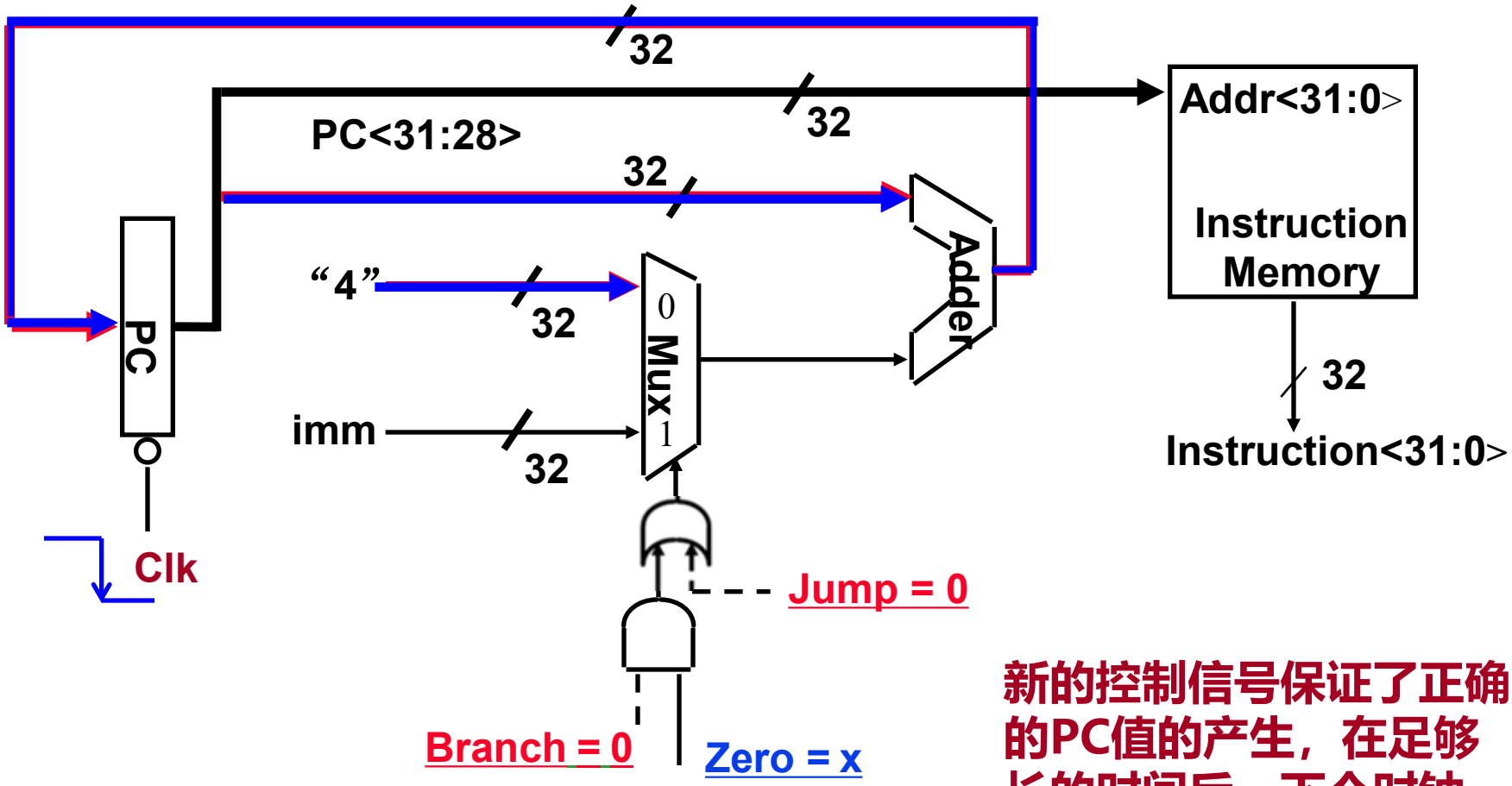
R-型指令功能：R[rd] ← R[rs1] op R[rs2]



R-型指令最后阶段取指部件中的动作

° PC ← PC + 4

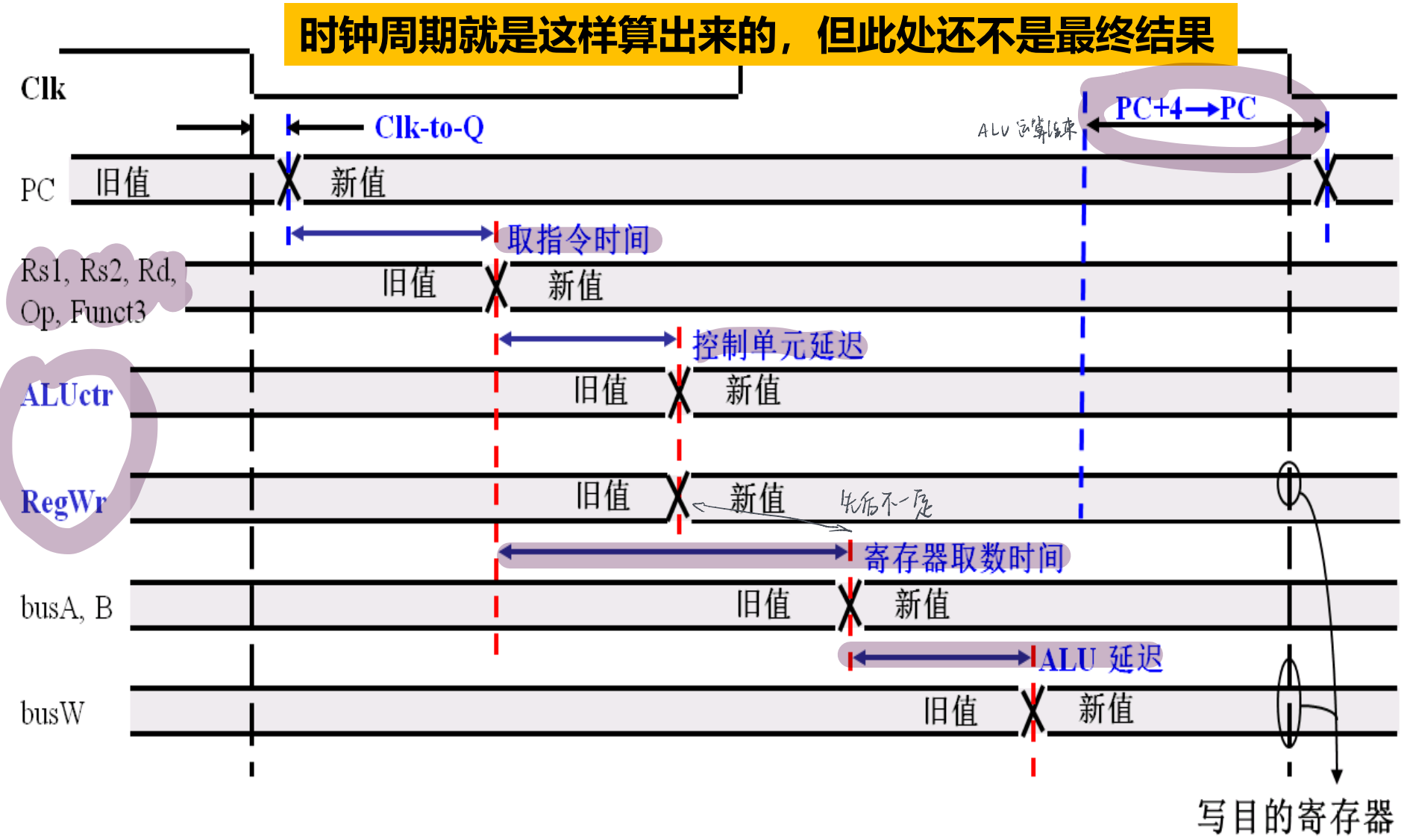
- 除 Branch and Jump以外的指令都相同



新的控制信号保证了正确的PC值的产生，在足够长的时间后，下个时钟Clk到来！

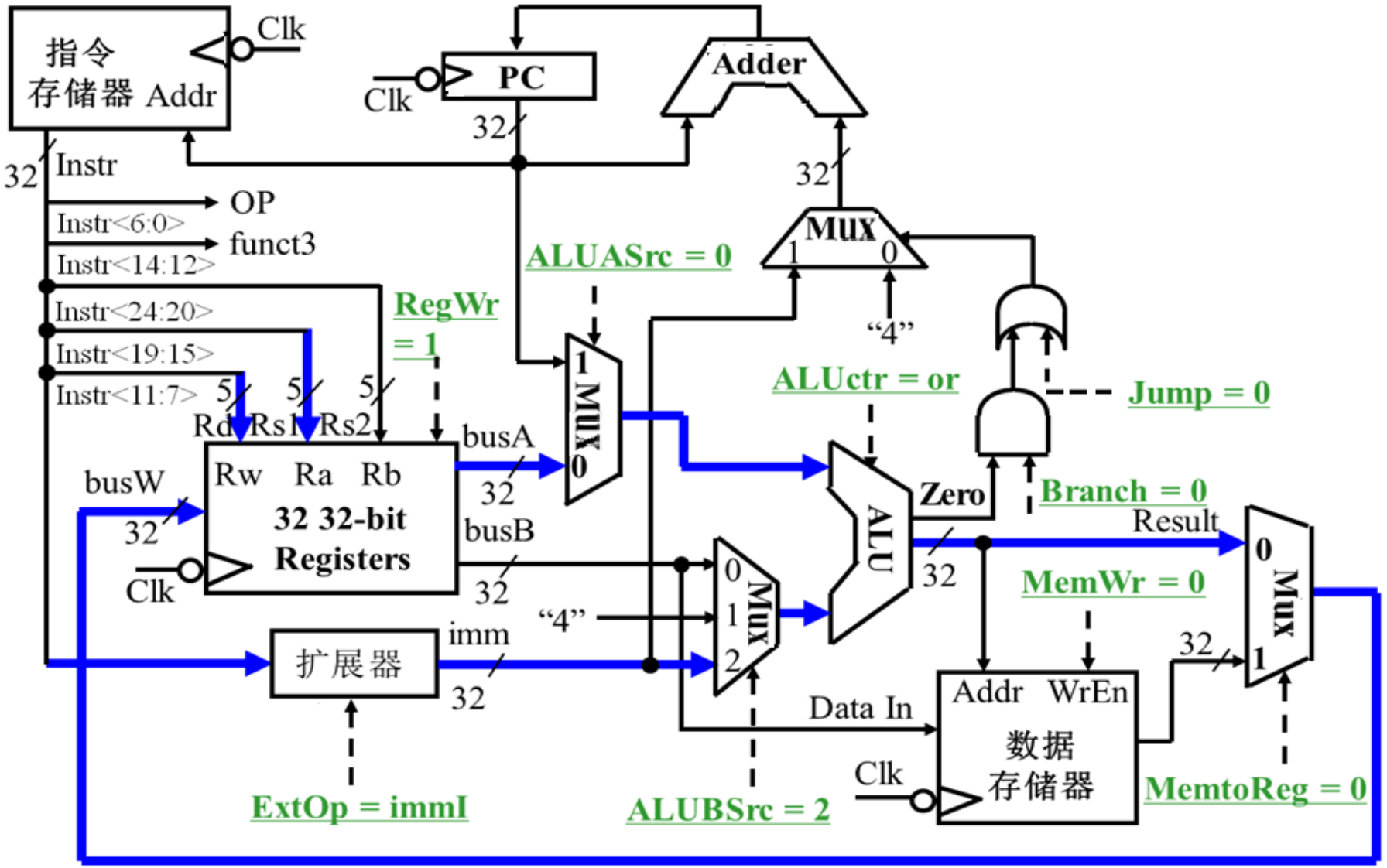
R-型指令的操作定时

R-型指令功能：R[rd] ← R[rs1] op R[rs2]



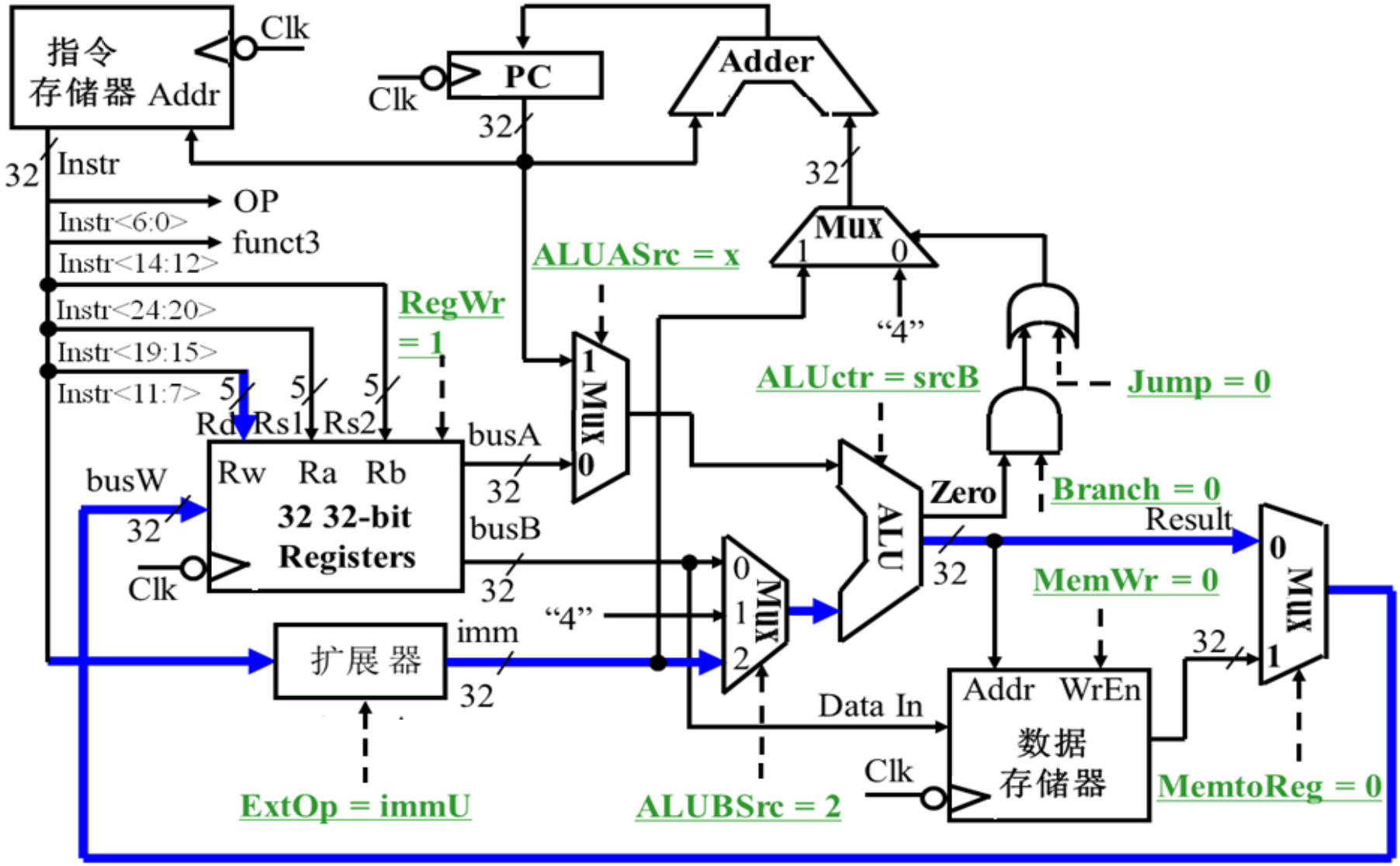
指令译码后I-型运算指令操作过程

I-型指令 ori 功能：R[rd] ← R[rs1] or SEXT(imm12)



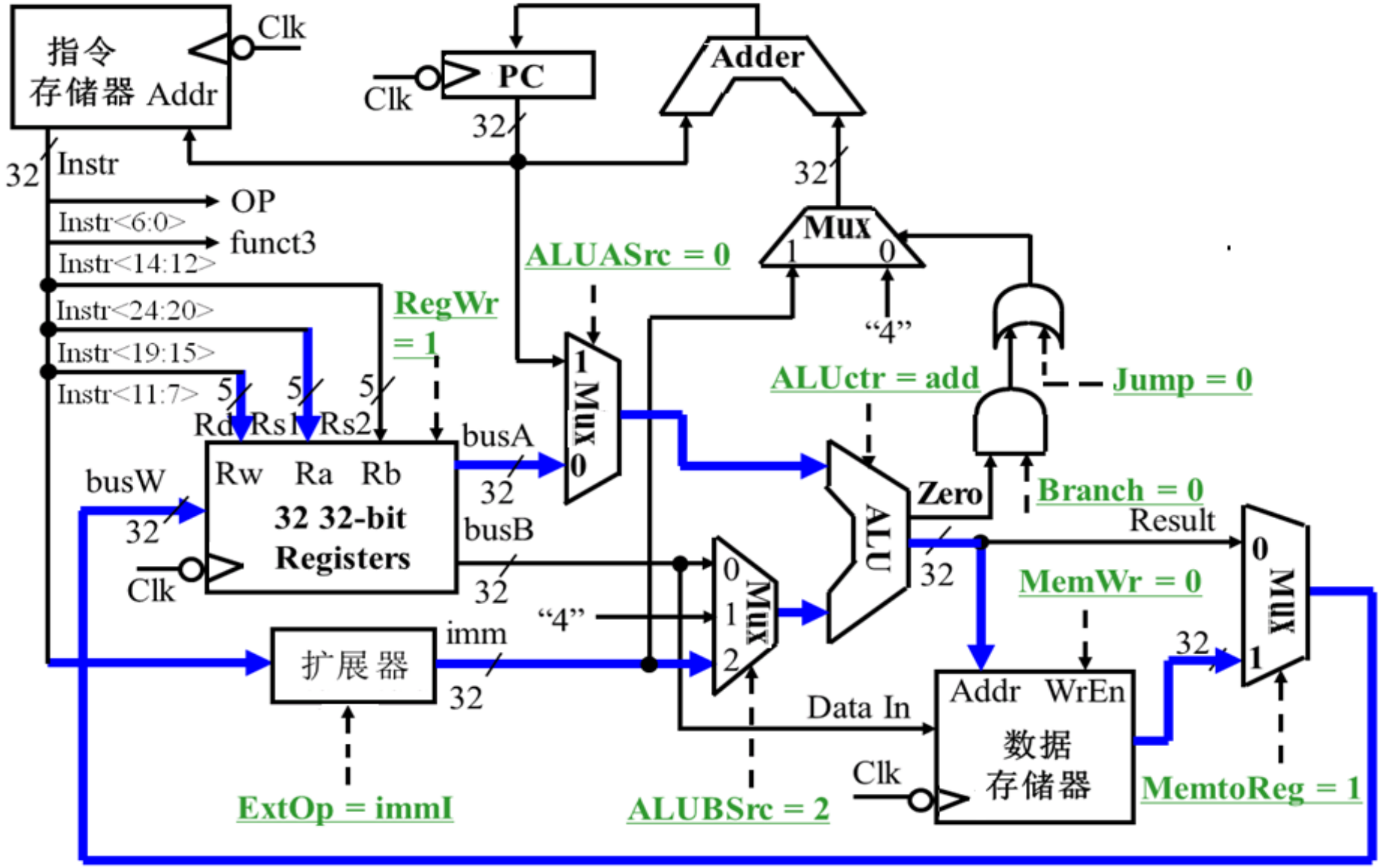
指令译码后U-型指令操作过程

U-型指令lui的功能：R[rd]← imm20||000H 即直接将扩展器结果输出



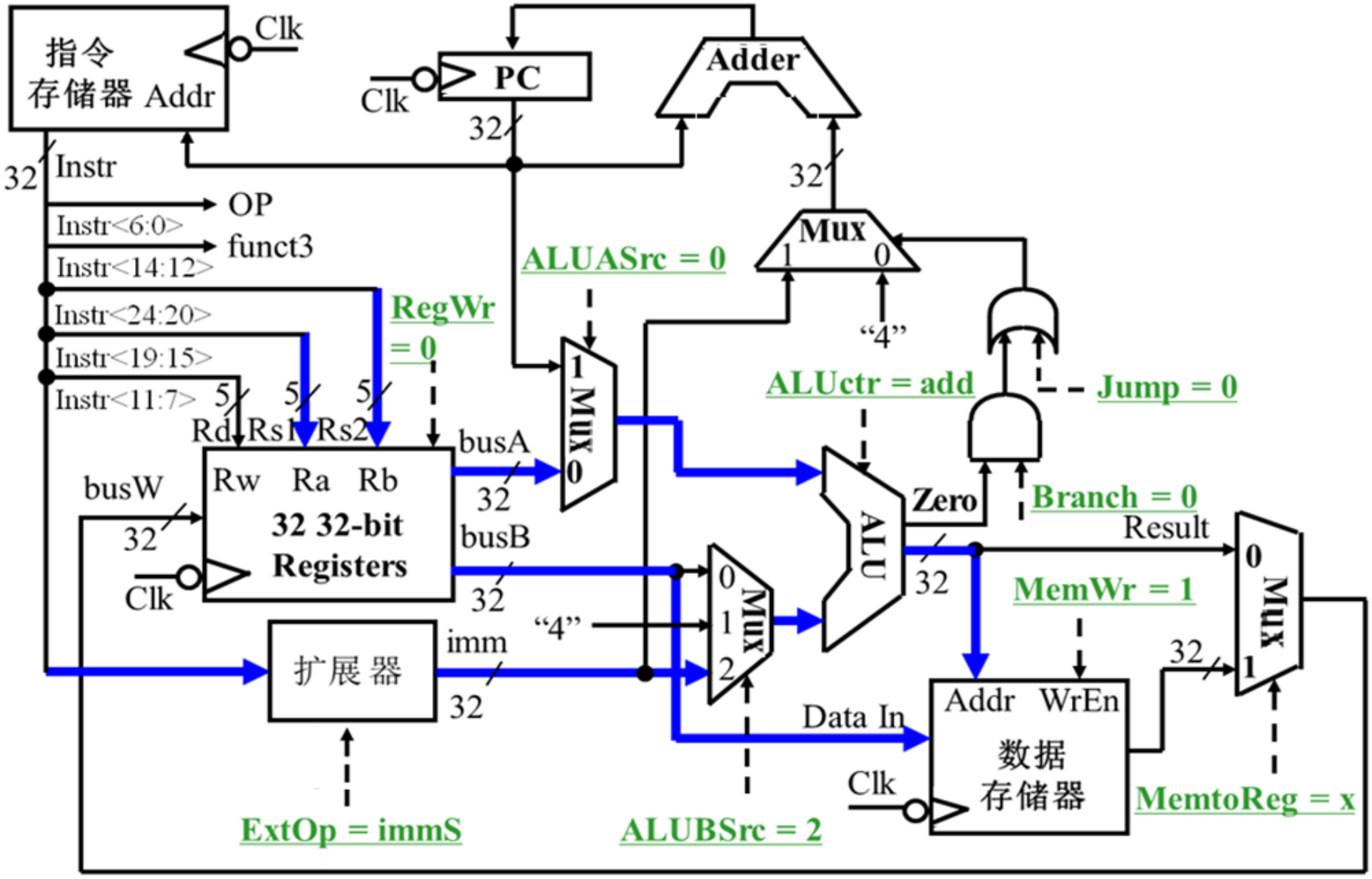
指令译码后Load指令操作过程

I-型的lw指令的功能：R[rd]← M[R[rs1]+SEXT(imm12)]



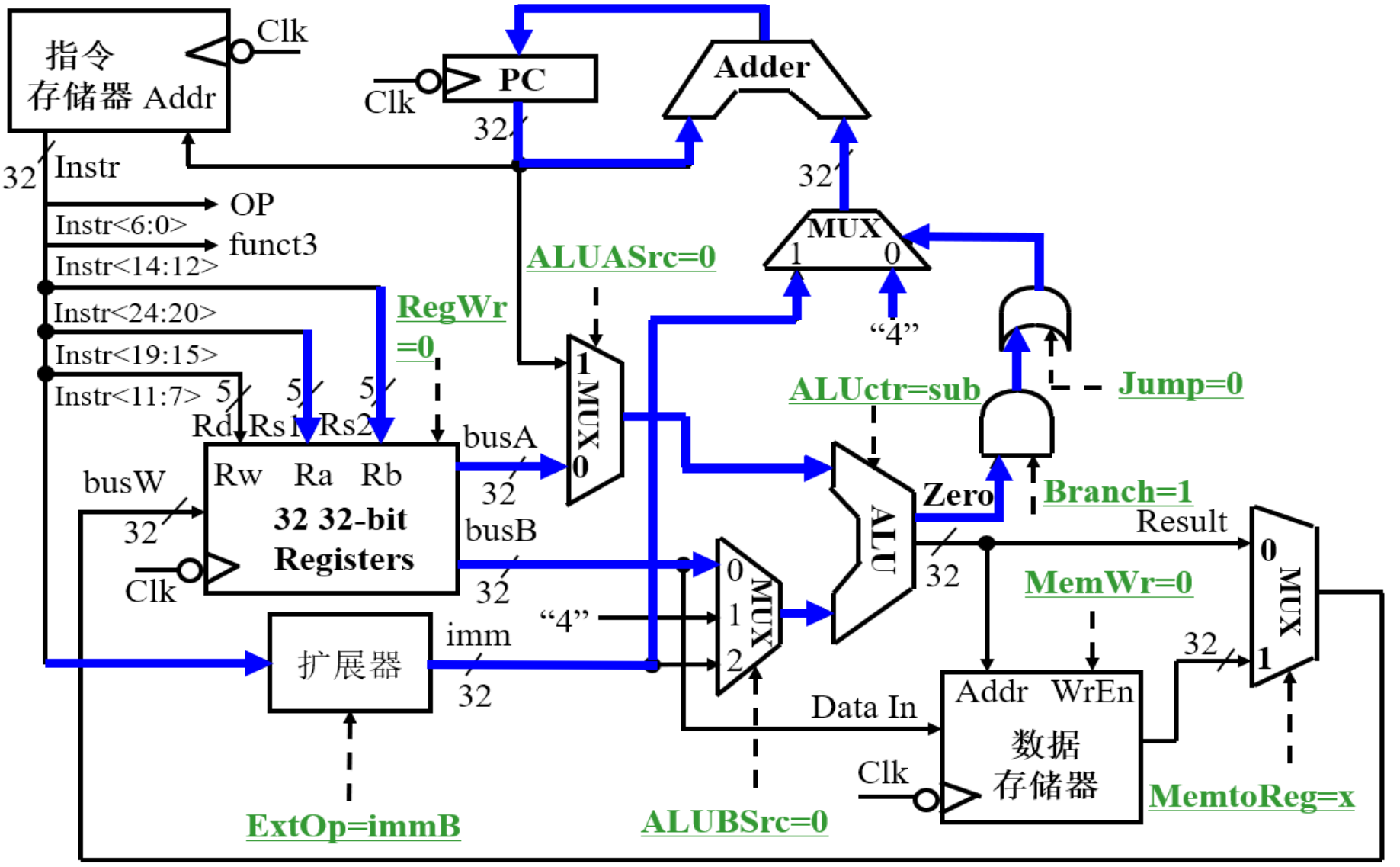
指令译码后Store指令操作过程

S-型的sw指令的功能：M[R[rs1]+SEXT(imm12)]←R[rs2]



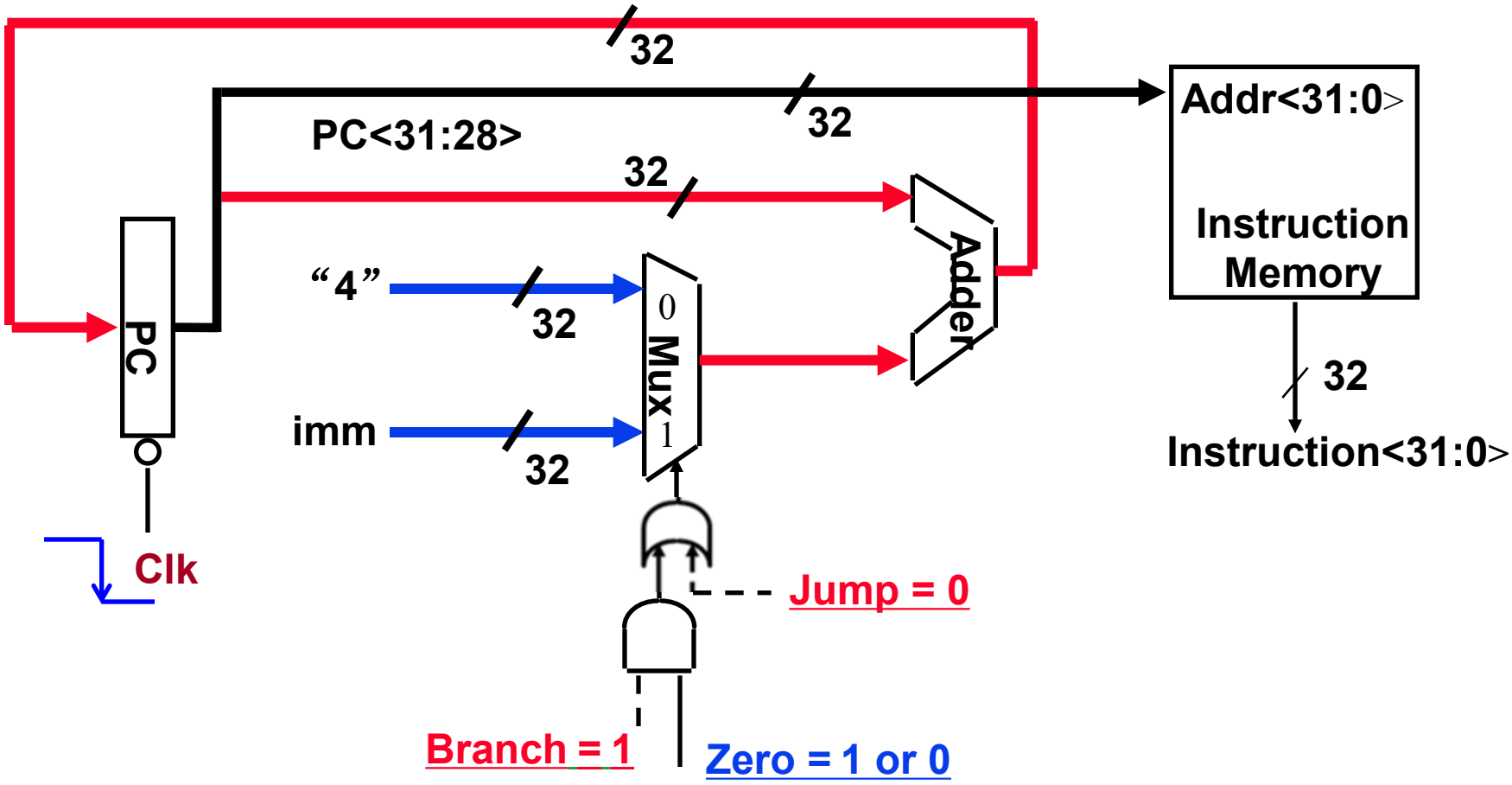
指令译码后B-型指令操作过程

beq功能：if (R[rs1]=R[rs2]) PC←PC+(SEXT(imm12)×2) else PC←PC+4



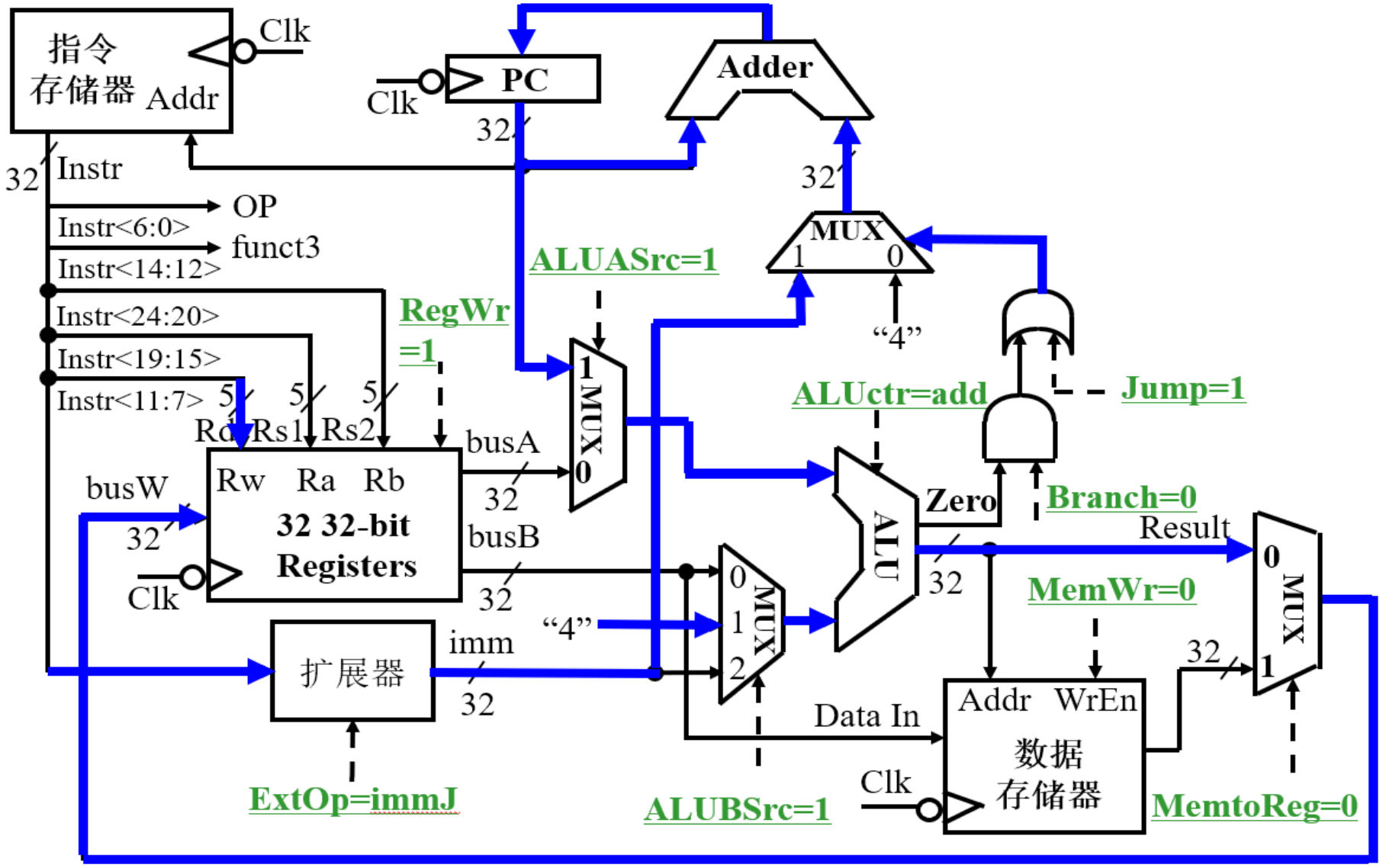
beq指令最后阶段取指部件中的动作

° if (Zero == 1) then PC = PC + imm else PC = PC + 4



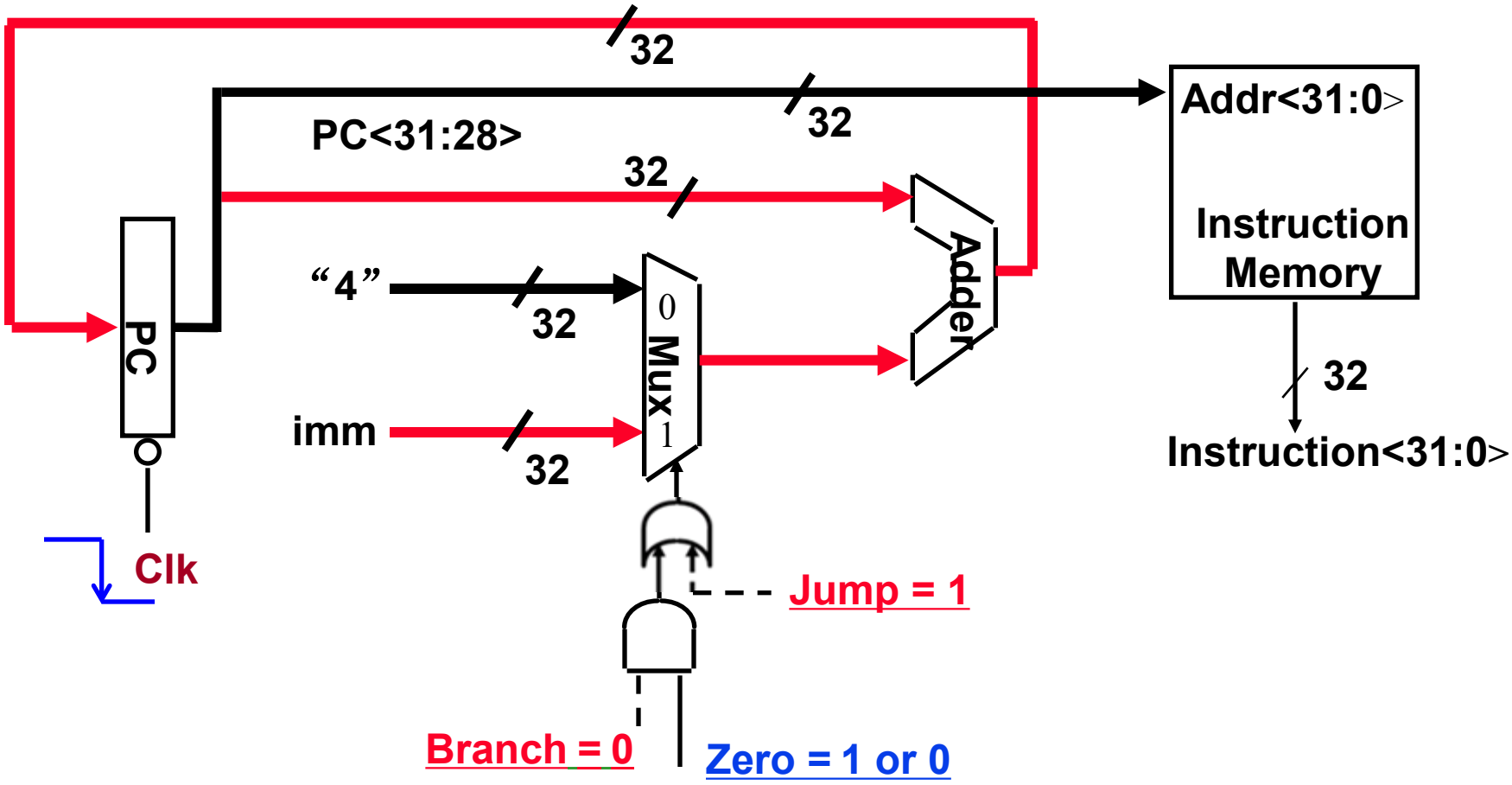
指令译码后J-型指令操作过程

J-型指令jal的功能：PC←PC+SEXT(imm[20:1]<<1)；R[rd]←PC+4



Jal指令最后阶段取指部件中的动作

° PC = PC + imm



综合分析结果，得到如下指令与控制信号的关系表

funct3	000	010	011	110	无关	010	010	000	无关
op	0110011	0110011	0110011	0010011	0110111	0000011	0100011	1100011	1101111
控制信号	add	slt	sltu	ori	lui	lw	sw	beq	jal
Branch	0	0	0	0	0	0	0	1	0
Jump	0	0	0	0	0	0	0	0	1
ALUASrc	0	0	0	0	×	0	0	0	1
ALUBSrc<1:0>	00	00	00	10	10	10	10	00	01
ALUctr<3:0>	0000 (add)	0010 (slt)	0011 (sltu)	0110 (or)	1111 (srcB)	0000 (add)	0000 (add)	1000 (sub)	0000 (add)
MemtoReg	0	0	0	0	0	1	×	×	0
RegWr	1	1	1	1	1	1	0	0	1
MemWr	0	0	0	0	0	0	1	0	0
ExtOp<2:0>	×	×	×	000 immI	001 immU	000 immI	010 immS	011 immB	100 immJ

单值控制信号的逻辑表达式较易

有3个多值控制信号：ALUBSrc、ALUctr、ExtOp，每一位都有逻辑表达式

单值控制信号逻辑表达式的生成例子

funct3	000	010	011	110	无关	010	010	000	无关
op	0110011	0110011	0110011	0010011	0110111	0000011	0100011	1100011	1101111
控制信号	add	slt	sltu	ori	lui	lw	sw	beq	jal
Branch	0	0	0	0	0	0	0	1	0
Jump	0	0	0	0	0	0	0	0	1

Branch=op<6>&op<5>&~op<4>&~op<3>&~op<2>&op<1>&op<0> (B-type)

注意：本例中beq的func3为000，但其它B型指令也一样
会让branch信号为1，所以此处就不用判断func3了

Jump=op<6>&op<5>&~op<4>&op<3>&op<2>&op<1>&op<0>
(J-type)

单值控制信号逻辑表达式的生成例子

funct3	000	010	011	110	无关	010	010	000	无关
op	0110011	0110011	0110011	0010011	0110111	0000011	0100011	1100011	1101111
控制信号	add	slt	sltu	ori	lui	lw	sw	beq	jal
RegWr	1	1	1	1	1	1	0	0	1

RegWr=(~op<6>& op<5>&op<4>&~op<3>&~op<2>&op<1>&op<0>) (R-type)
| (~op<6>&~op<5>&op<4>&~op<3>&~op<2>&op<1>&op<0>) (I-type-ALU)
| (~op<6>&op<5>&op<4>&~op<3>&op<2>&op<1>&op<0>) (lui)
| (~op<6>&~op<5>&~op<4>&~op<3>&~op<2>&op<1>& op<0>) (Load)
| (op<6>&op<5>&~op<4>&op<3>&op<2>&op<1>&op<0>) (J-type)

注意：类似于上页ppt所述，所以此处也都不用判断func3

多值控制信号逻辑表达式的生成

funct3 / op / 控制信号	000	010	011	110	无关	010	010	000	无关
op	0110011	0110011	0110011	0010011	0110111	0000011	0100011	1100011	1101111
控制信号	add	slt	sltu	ori	lui	lw	sw	beq	jal
ALUBSrc<1:0>	00	00	00	10	10	10	10	00	01
ALUctr<3:0>	0000 (add)	0010 (slt)	0011 (sltu)	0110 (or)	1111 (srcB)	0000 (add)	0000 (add)	1000 (sub)	0000 (add)
ExtOp<2:0>	×	×	×	000 immI	001 immU	000 immI	010 immS	011 immB	100 immJ

以ALUctr为例介绍多值控制信号逻辑表达式生成。

从上表中观察到：R-型和I-型运算类指令的funct3与ALUctr后3位编码相同，而第1位为0，即ALUctr=0||funct3，lw、sw和jal都是对应add操作，即ALUctr=0000，综上得到ALUctr编码分配表如下：

	R-type	I-type-ALU	lui	Load/Store	B-type	J-type
ALUctr<3:0>	0 || funct3 (||为串接操作)		1111	0000	1000	0000

多值控制信号ALUctr逻辑表达式的生成

	R-type	I-type-ALU	Lui 0110111	Load/Store	B-type 1100011	J-type
ALUctr<3:0>	0 || funct3 (||为串接操作)		1111	0000	1000	0000

设funct3各位分别表示为fn<2> fn<1> fn<0>，ALUctr各位逻辑表达式如下：

ALUctr<3> =

(~op<6> & op<5> & op<4> & ~op<3> & op<2> & op<1> & op<0>)

(lui)

| (op<6> & op<5> & ~op<4> & ~op<3> & ~op<2> & op<1> & op<0>)

(B-type)

多值控制信号ALUctr逻辑表达式的生成

	R-type	I-type-ALU	Lui 0110111	Load/Store 0000011 0100011	B-type 1100011	J-type 1101111
ALUctr <3:0>	0 || funct3		1111	0000	1000	0000

设funct3各位分别表示为fn<2> fn<1> fn<0>，ALUctr各位逻辑表达式如下：

ALUctr<2> =

((~op<6> & op<5> & op<4> & ~op<3> & ~op<2> & op<1> & op<0>)
| (~op<6> & ~op<5> & op<4> & ~op<3> & ~op<2> & op<1> & op<0>)) & fn<2>
(R-type + I-type-ALU)
| (~op<6> & op<5> & op<4> & ~op<3> & op<2> & op<1> & op<0>)
(lui)

多值控制信号ALUctr逻辑表达式的生成

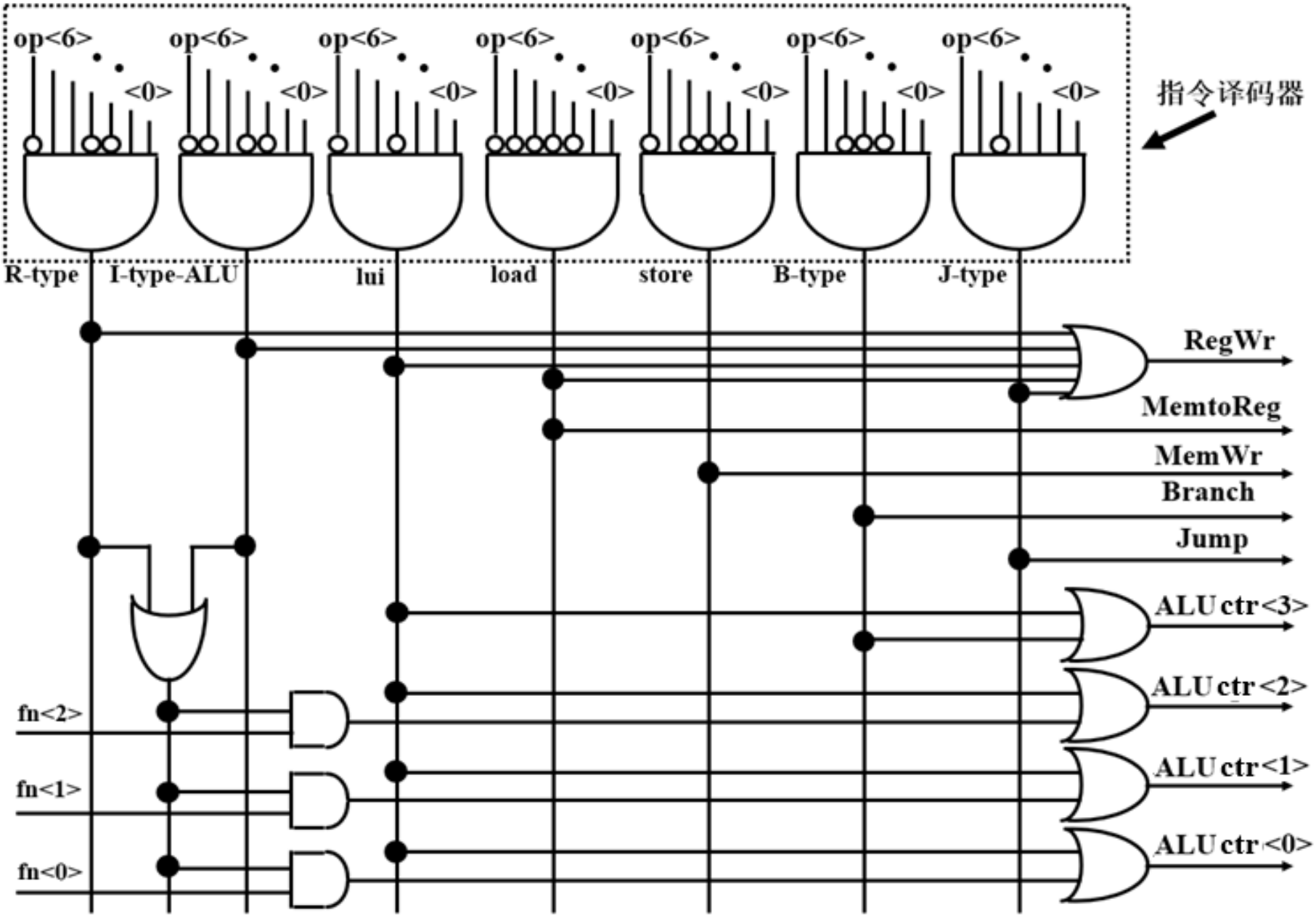
	R-type	I-type-ALU	lui	Load/Store	B-type	J-type
ALUctr<3:0>	0 || funct3（||为串接操作)		1111	0000	1000	0000

设funct3各位分别表示为fn<2> fn<1> fn<0>，ALUctr各位逻辑表达式如下：

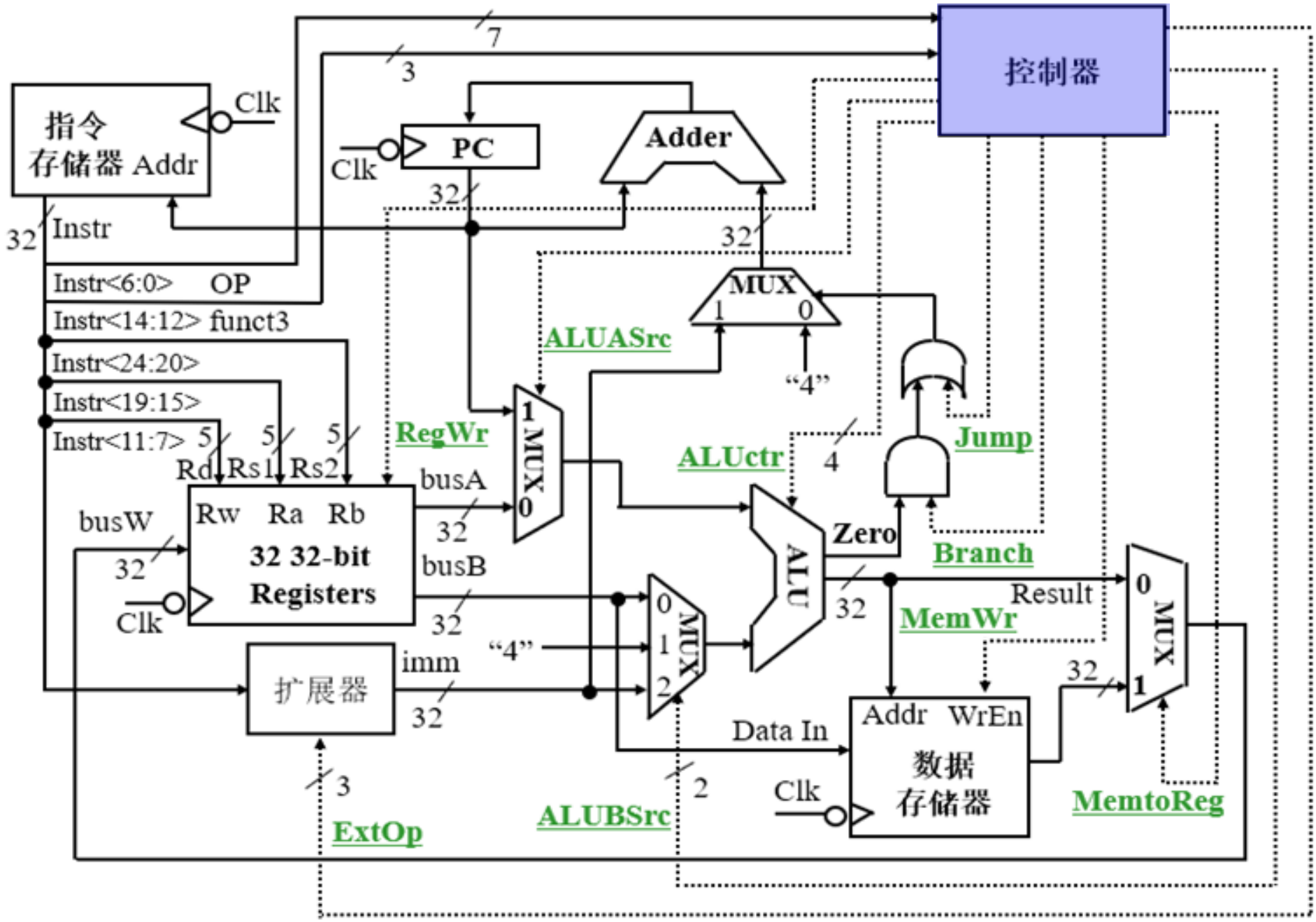
ALUctr<1> = ((~op<6> & op<5> & op<4> & ~op<3> & ~op<2> & op<1> & op<0>)
| (~op<6> & ~op<5> & op<4> & ~op<3> & ~op<2> & op<1> & op<0>)) & fn<1>
| (~op<6> & op<5> & op<4> & ~op<3> & op<2> & op<1> & op<0>)
(R-type + I-type-ALU) &fn<1> +(lui)

ALUctr<0> = ((~op<6> & op<5> & op<4> & ~op<3> & ~op<2> & op<1> & op<0>)
| (~op<6> & ~op<5> & op<4> & ~op<3> & ~op<2> & op<1> & op<0>)) & fn<0>
| (~op<6> & op<5> & op<4> & ~op<3> & op<2> & op<1> & op<0>)
(R-type + I-type-ALU) &fn<0> +(lui)

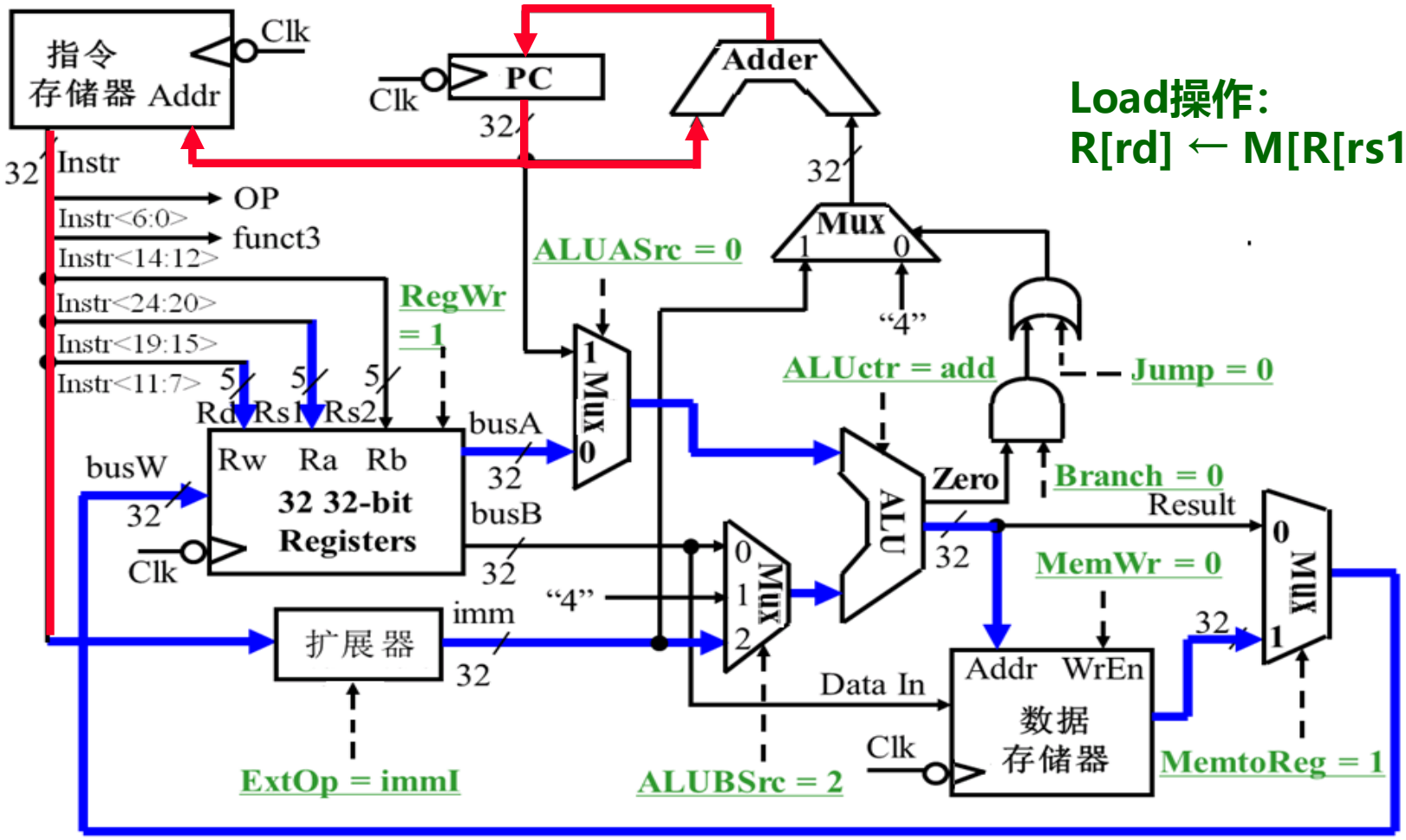
根据逻辑表达式实现控制器的PLA电路



执行前述9条指令完整的单周期处理器



时钟周期的确定：找数据通路中的关键路径

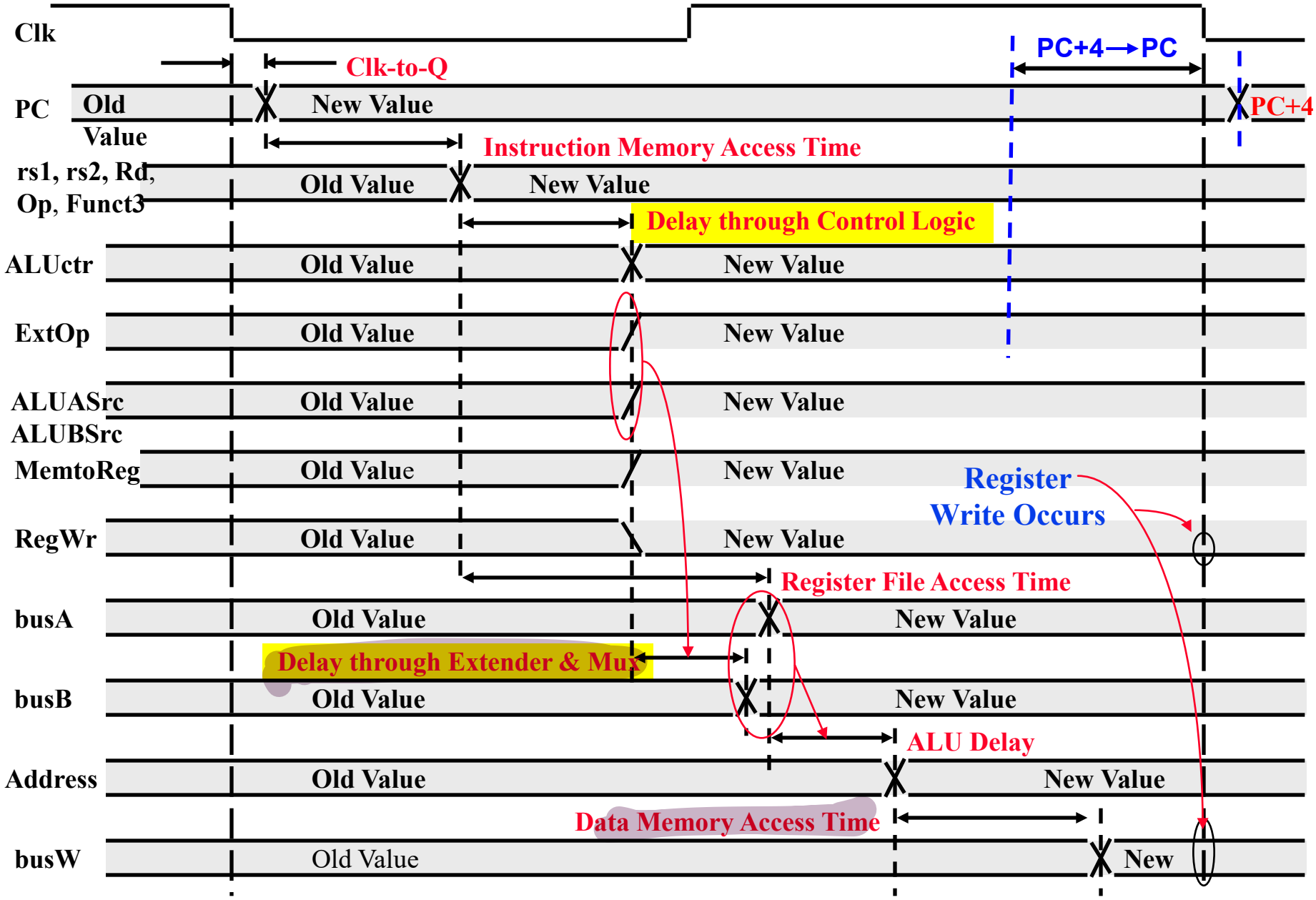


Load操作：
R[rd] ← M[R[rs1]+imm]

- 寄存器组和理想存储器的定时方式
  - 写操作时，作为时序逻辑电路。即：
    - 时钟到达前，输入需setup；时钟到达后经Clk-to-Q，写入数据到达输出端
  - 读操作时，作为组合逻辑电路。即：
    - 地址有效后经过 “access time”，输出开始有效

Critical Path (Load Operation)（时钟周期）=
PC’s prop time (Clk-to-Q) +
Instruction Memory’s Access Time +
Register File’s Access Time +
ALU to Perform a 32-bit Add +
Data Memory Access Time +
Setup Time for Register File Write +
Clock Skew

lw指令的执行时间最长，它所花时间作为时钟周期



单周期计算机的性能

- 单周期处理器的CPI为多少？ CPI=1!
  - 其他条件一定的情况下，CPI越小，则性能越好！
  - CPI=1，不是很好吗？
- 单周期处理器的性能会不会很好？为什么？
  - 计算机的性能除CPI外，还取决于时钟周期的宽度
  - 单周期处理器的时钟宽度为最复杂指令的执行时间
  - 很多指令可以在更短的时间内完成
    - 非load/store指令无需访问DM
    - J-指令无需读通用寄存器

单周期计算机的性能

假设在单周期处理器中，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，则单周期实现方式（每条指令在一个固定长度的时钟周期内完成）中，CPU执行时间如何计算?

各类指令要求的时间长度为：

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	0	550 ps
Branch	200	50	100	0	0	350 ps
Jump	200	0	100	0	50	350 ps

解：CPU执行时间=指令条数 x CPI x 时钟周期

单周期CPI都为1，时钟周期由最长指令来决定，应该是load指令，为600ps

所以：CPU执行时间 = 600x程序指令条数

第三讲 小结

- 考察每条指令在单周期数据通路中的执行过程
  - 每条指令在一个时钟周期内完成
  - 每个时钟到来时，都开始进入取指令操作
    - 经过clk-to-Q，PC得到新值，经过access time后得到当前指令
    - 按三种方式分别计算下条指令地址，在branch / zero / jump的控制下，选择其中之一送到PC输入端，但不会马上写到PC中，一直到下个时钟到达时，才会更新PC。三种下址方式为：
      - branch=jump=0：PC+4
      - branch=zero=1： PC+SEXT(imm12)*2 (ExtOP为ImmB)
      - jump=1： PC+SEXT(imm20)*2 (ExtOP为ImmJ)

第三讲 小结

本章作业：习题3、4、5、6、11、13、14、17、18

- 考察每条指令在单周期数据通路中的执行过程
  - 指令取出后被译码，产生指令对应的控制信号
- 3条R-型指令：add rd, rs1, rs2、slt rd, rs1, rs2、sltu rd, rs1, rs2
  - rd为目的寄存器，无访存操作，……
- I-型指令：ori rd, rs1, imm12、
  - rd为目的寄存器，符号扩展，无访存操作，……
- I-型指令： lw rd, imm12(rs1)
  - rd为目的寄存器，符号扩展，计算地址、读内存，……
- 1条S-型指令：sw rs2, imm12(rs1)
  - rs2为源寄存器，符号扩展，计算地址、写内存，……
- 1条B-型指令：beq rs1, rs2, imm12
  - rs1和rs2为源寄存器，符号扩展，无访存操作，PC有条件更新……
- 1条U-型指令：lui rd, imm20
  - rd为目的寄存器，立即数末尾补0，无访存操作，……
- 1条J-型指令：jal rd, imm20
  - rd为目的寄存器（PC+4），符号扩展，无访存操作， PC无条件更新……
- 汇总每条指令控制信号的取值，生成真值表，写出逻辑表达式，设计控制器逻辑

第8章 中央处理器（一）

第一讲 中央处理器概述

第二讲 单周期数据通路的设计

第三讲 单周期控制器的设计

第四讲 多周期处理器的设计

第五讲 流水线处理器设计

第六讲 流水线冒险及其处理

第七讲 高级流水线技术

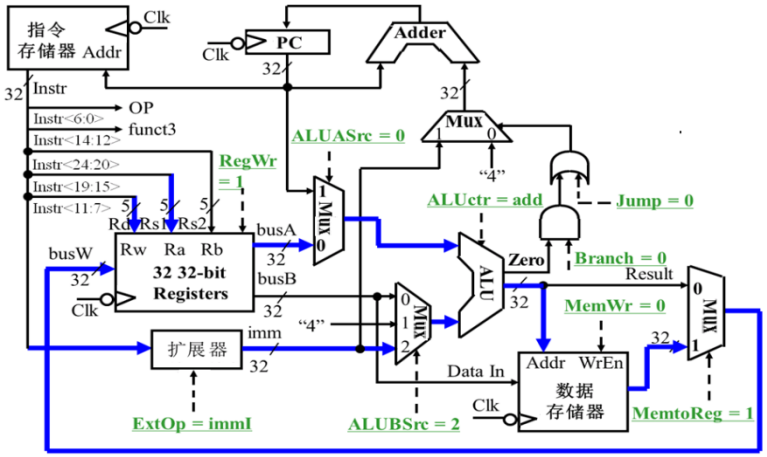
第四讲 多周期处理器的设计

主 要 内 容

- 多周期数据通路实现思想
- 单周期数据通路和多周期数据通路的差别
  - 通过简要分析LOAD指令分阶段执行过程，以加深理解单周期和多周期数据通路的差别
- 实现目标：一个简单指令系统
- 多周期数据通路设计
- 硬连线控制器和微程序控制器
- 带异常处理的处理器设计

单周期处理器时钟周期的特点

- 单周期处理器的CPI为1
- 所有指令执行时间都是1个时钟周期
- 但是，时钟周期必须以最长的load指令为准



- 因此，时钟周期远远大于其他指令实际所需的执行时间，效率低

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	0	550 ps
Branch	200	50	100	0	0	350 ps
Jump	200	0	100	0	50	350 ps

- 单周期处理器的问题根源:
  - 时钟周期以最复杂指令所需时间为准，太长!

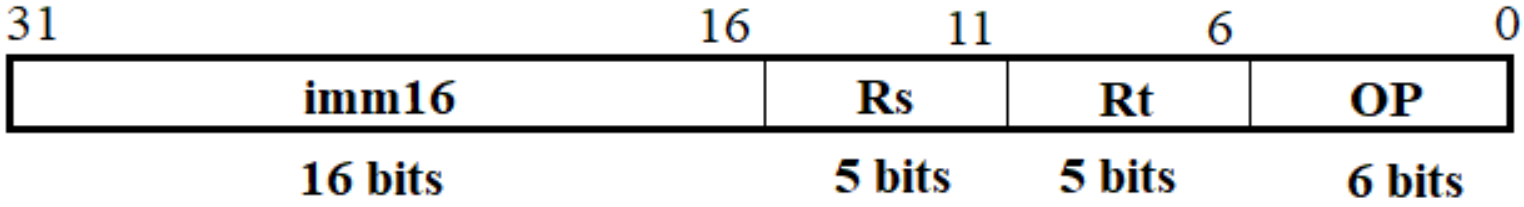
多周期处理器的实现思想

- 解决思路:
  - 把指令的执行分成多个阶段，每个阶段在一个时钟周期内完成
    - 时钟周期以最复杂阶段所花时间为准
    - 尽量分成大致相等的若干阶段
    - 规定每个阶段最多只能完成1次访存 或 寄存器堆读/写 或 ALU
  - 每步都设置存储单元，每步的执行结果都在下个时钟开始保存到相应单元
- 多周期处理器的好处:
  - 时钟周期短
  - 不同指令所用周期数可以不同（前两个周期每条指令都一样），如:
    - Load：4? 5? cycles
    - 其它： 2? 3? 4? cycles
  - 允许功能部件在一条指令执行过程中被重复使用。如:
    - Adder + ALU（多周期时只用一个ALU，在不同周期可重复使用）
    - Inst. / Data mem（多周期时合用，不同周期中重复使用）

以下以一个简单的指令系统为实现目标，来介绍多周期处理的设计

多周期处理器实现目标：一个简单指令系统

- 只有一种指令格式



- 可实现以下几类指令功能

① R-型：R[Rt]←R[Rs] op R[Rt]，两个寄存器内容运算，结果送寄存器。

② I-型运算：R[Rt]←R[Rs] op EXT[imm16]，寄存器内容和立即数运算。

③ Load：R[Rt]←M[R[Rs]+SEXT[imm16]]，地址为寄存器内容加立即数。

④ Store：M[R[Rs]+SEXT[imm16]]←R[Rt]，地址为寄存器内容加立即数。

⑤ Jump指令：PC←PC+SEXT[imm16]，跳转目标地址为PC内容加立即数。

EXT[imm16]：对16位立即数imm16进行扩展，转换为32位操作数。

对于逻辑运算和无符号整数算术运算，采用零扩展；对于带符号整数算术运算，则采用符号扩展。

SEXT[imm16]：符号扩展。

指令各阶段分析

单周期CPU不需要IR

- 取指令阶段
  - 执行一次存储器读操作
  - 读出的内容（指令）保存到寄存器IR（指令寄存器）中
  - IR的内容不是每个时钟都更新，所以IR必须加一个“写使能”控制
  - 在取指令阶段结束时，ALU的输出为PC+4，并送到PC的输入端，但不能在每个时钟到来时就更新PC，所以PC也要有“写使能”控制

第一个周期（取指周期）

- 译码/读寄存器堆阶段
  - 经过控制逻辑延迟后，控制信号更新为新值
  - 执行一次寄存器读操作，并同时进行译码
  - 期间ALU空闲，可以考虑“投机计算”地址

第二个周期
（译码取数周期）

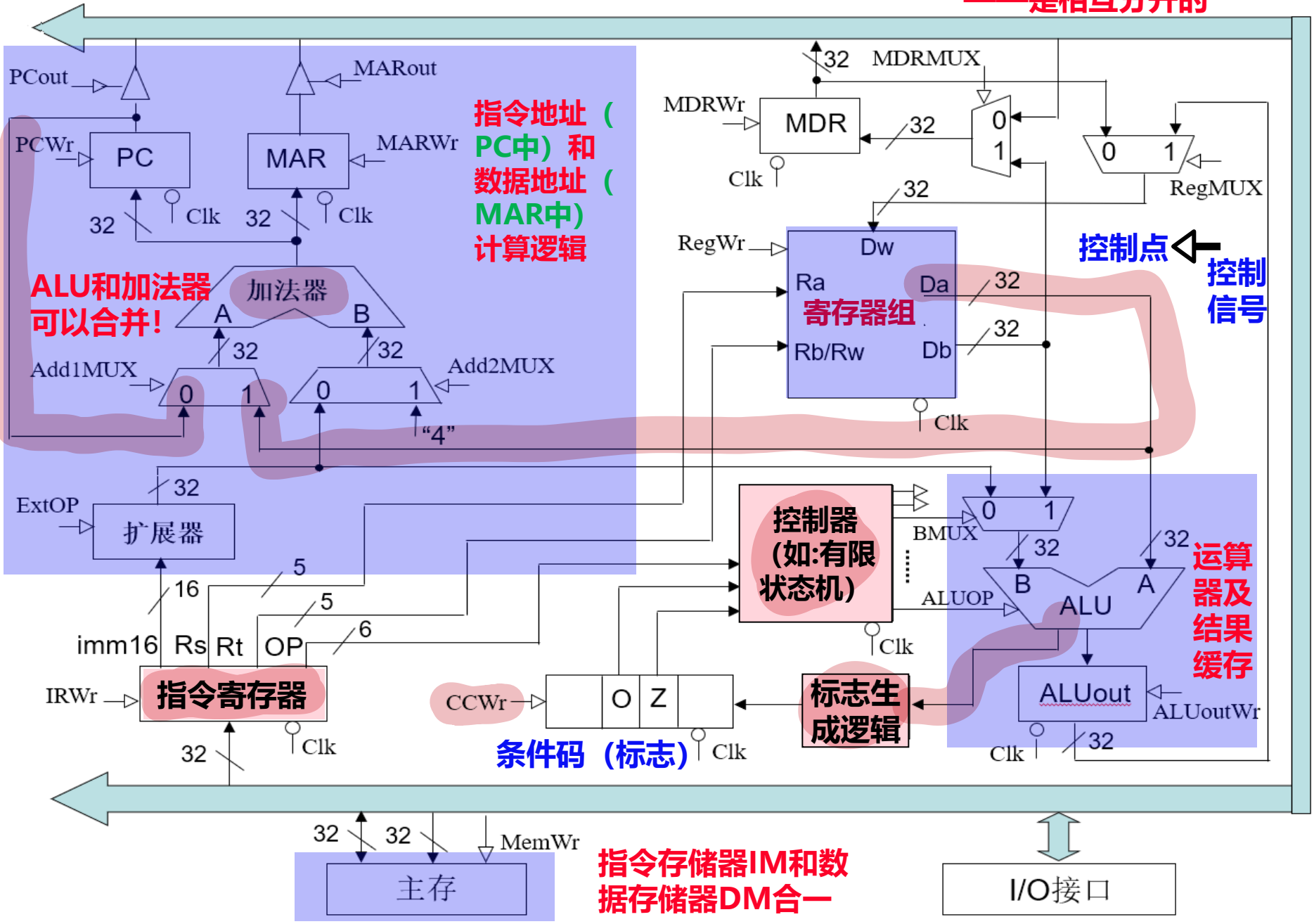
单周期CPU需要在一个时钟周期内完成整条指令的所有功能，ALU不可能空闲

- ALU运算阶段
  - ALU运算，输出结果一定要在下个时钟到达之前稳定
- 读存储器阶段
  - 由ALU运算结果作为地址访问存储器，读出数据
- 写结果到寄存器
  - 把之前的运算结果或读存储器结果写到寄存器堆中

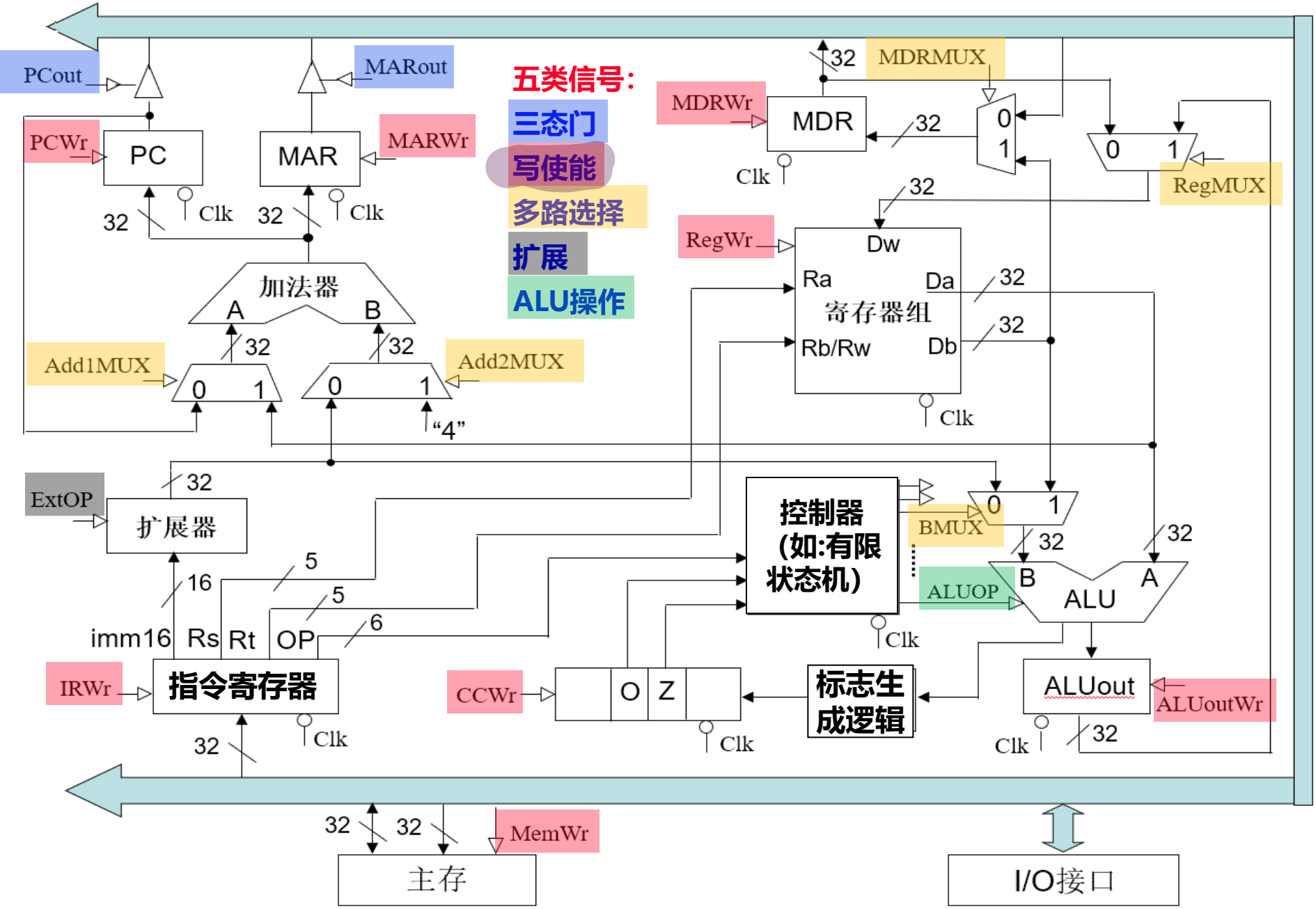
后面的周期（
各指令不同）

简单指令系统对应的多周期CPU

系统总线：AB+DB+CB
——是相互分开的

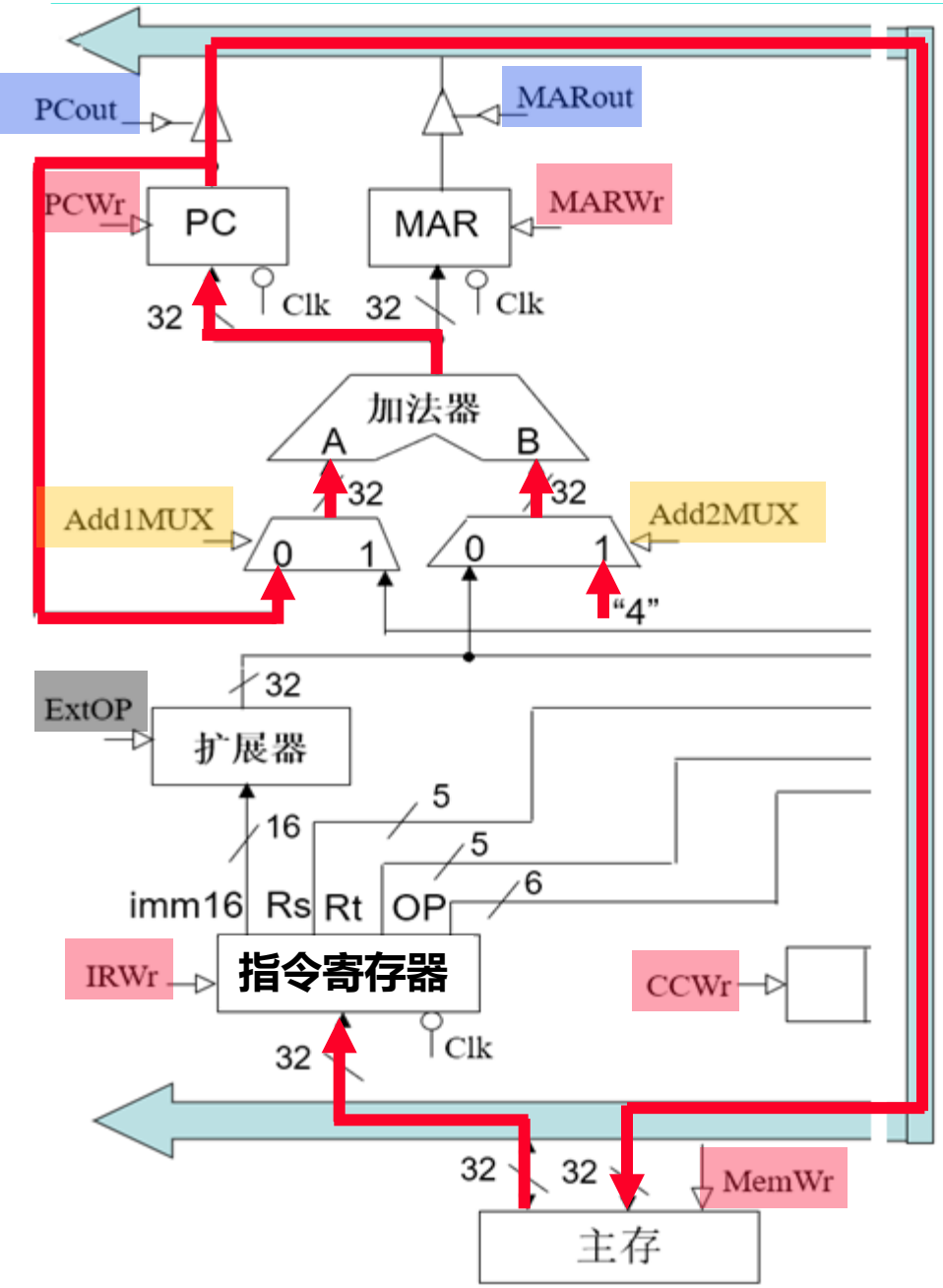


多周期CPU中的控制信号



各类指令执行过程分析

（1）取指令并计算下条指令地址（公共操作），记为 IFetch



°根据PC读指令并保存到IR，PC+4

°IR的内容不是每个时钟都更新，所以IR必须加一个“写使能”控制

°不是每个时钟到来时都更新PC，所以PC也要有“写使能”控制

°有效控制信号及其取值如下：

① R[IR] ← M[PC]：PCout=1、MARout=0、MemWr=0、IRWr=1

② PC ← PC+4：Add1MUX=0、Add2MUX=1、PCWr=1

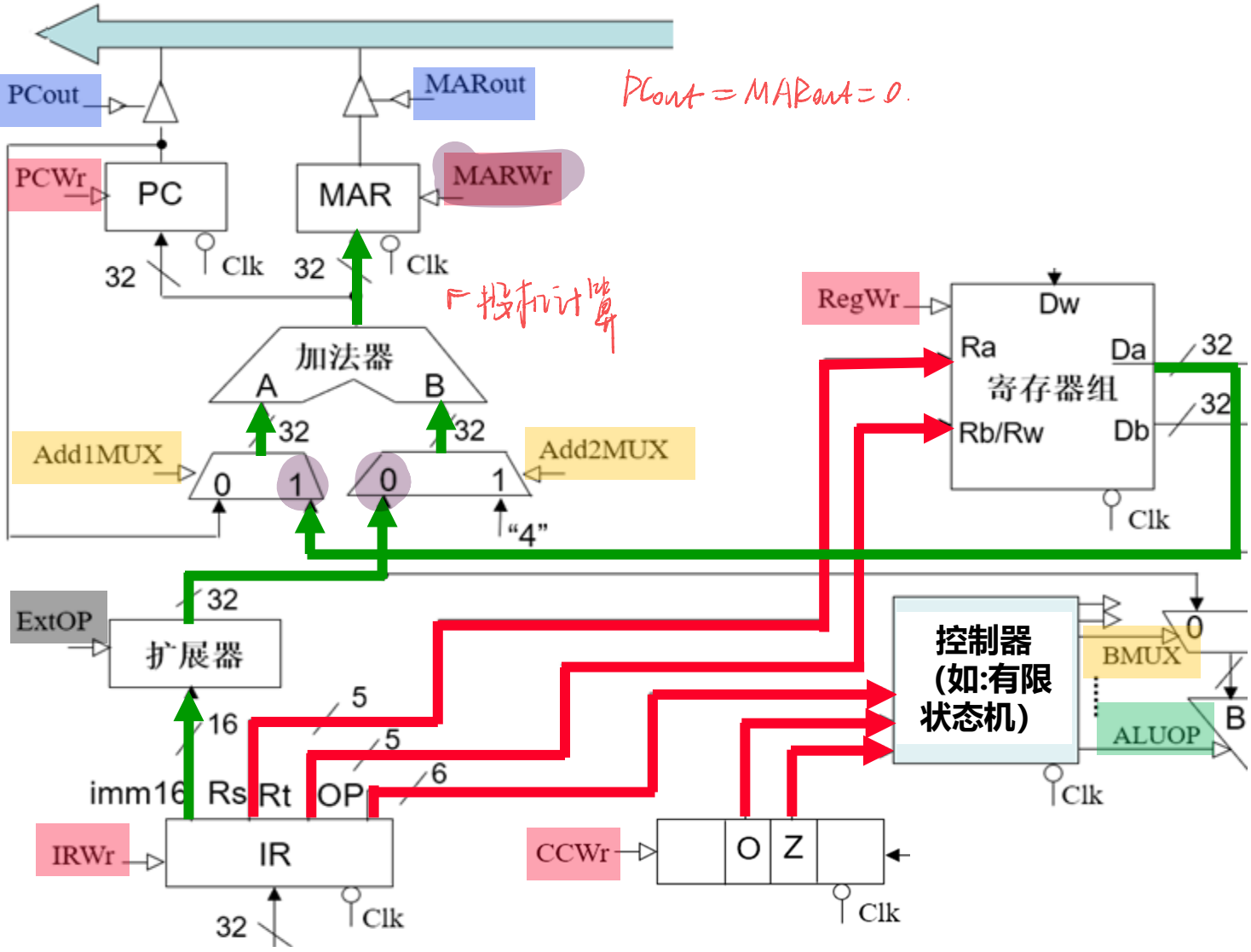
③ 其他写使能信号（如MARWr、CCWr、MDRWr、ALUoutWr、RegWr）全部为0

°当前时钟结束时，在IR和PC的输入端有新值，在IRWr和PCwr的控制下，下个时钟到来后的clk-to-Q时可用！

各类指令执行过程分析 (2) 译码并取数（公共操作）记为 Rfetch/ID

°IR的OP及CC送控制器译码，并根据Rs和Rt读取寄存器中数据

°加法器空闲，投机计算主存地址



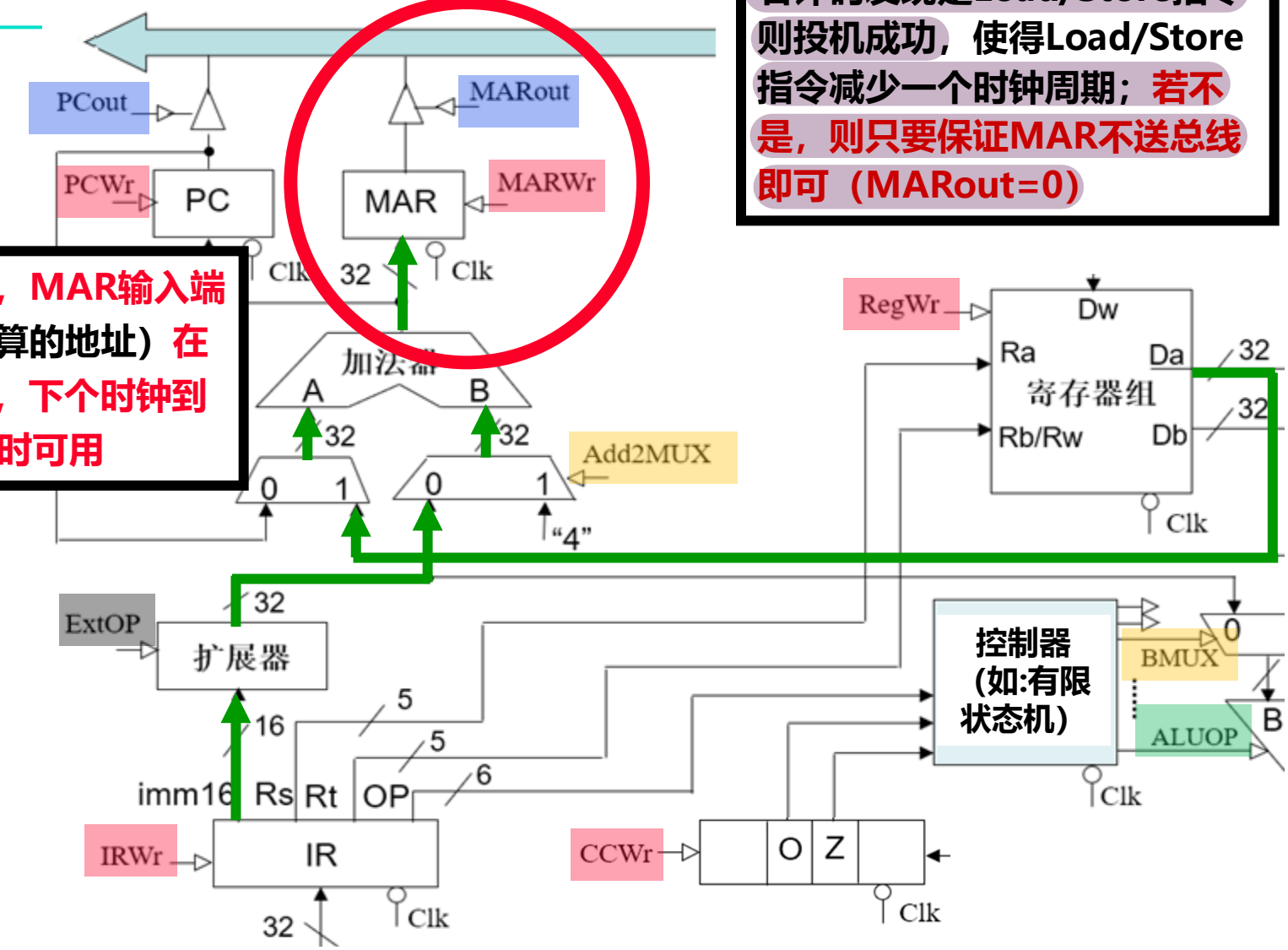
有效控制信号及其取值为：

ExtOP=1(符扩)、Add1MUX=1、Add2MUX=0、MARWr=1、其他写使能信号全部为0

PCout=0、MARout=0

若译码发现是Load/Store指令则投机成功，使得Load/Store指令减少一个时钟周期；若不是，则只要保证MAR不送总线即可（MARout=0）

当前时钟结束时，MAR输入端有新值（投机计算的地址）在MARWr控制下，下个时钟到来后的clk-to-Q时可用



下个时钟开始，在控制器生成的控制信号控制下执行指令。

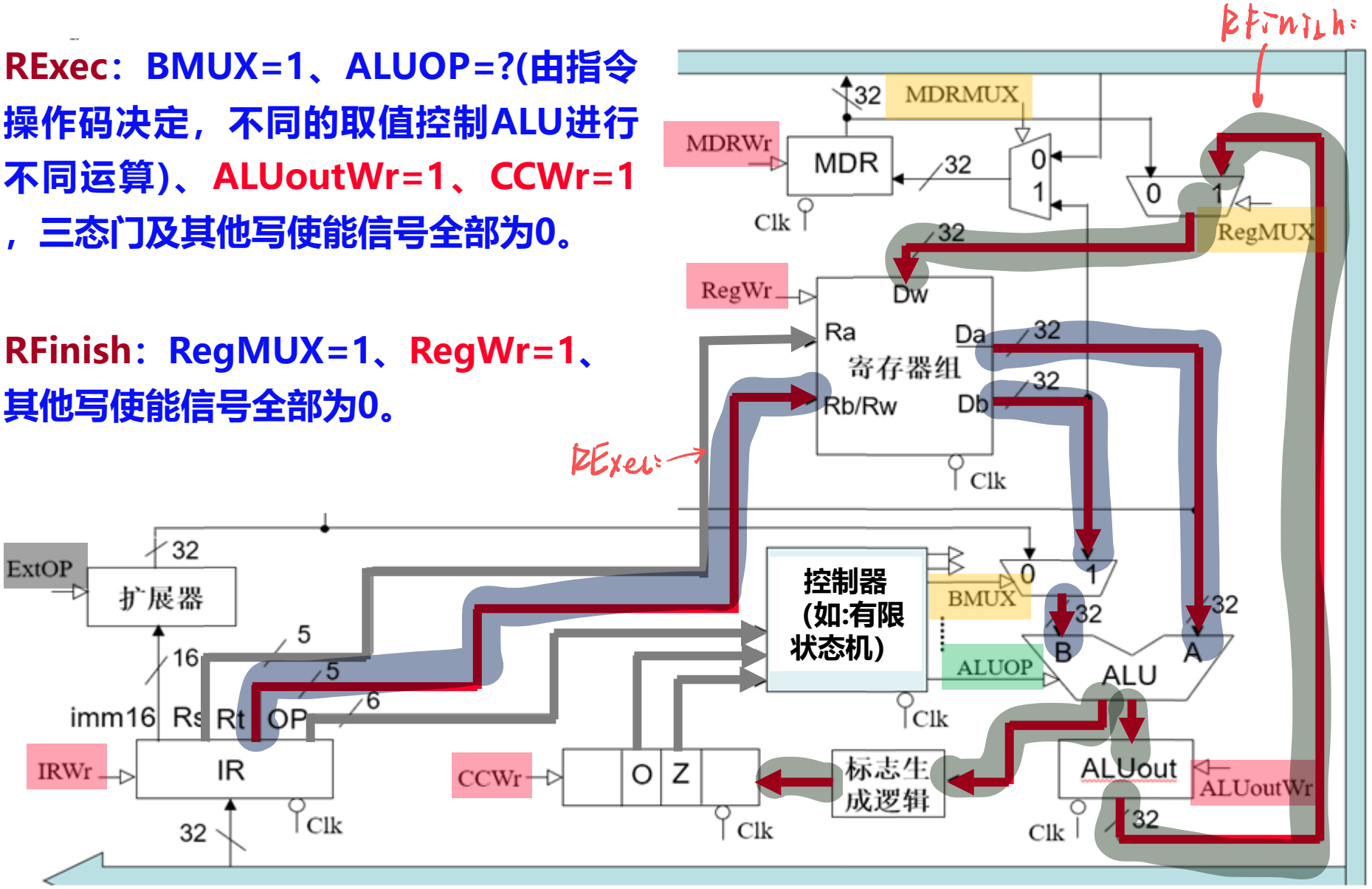
各类指令执行过程分析

（3）R-型指令的执行，需要两个时钟周期，记为 RExec、RFinish状态

进行ALU运算并将结果存入ALUout，再将相应结果分别写入Rt和CC寄存器。

RExec：BMUX=1、ALUOP=?(由指令操作码决定，不同的取值控制ALU进行不同运算)、ALUoutWr=1、CCWr=1，三态门及其他写使能信号全部为0。

RFinish：RegMUX=1、RegWr=1、其他写使能信号全部为0。



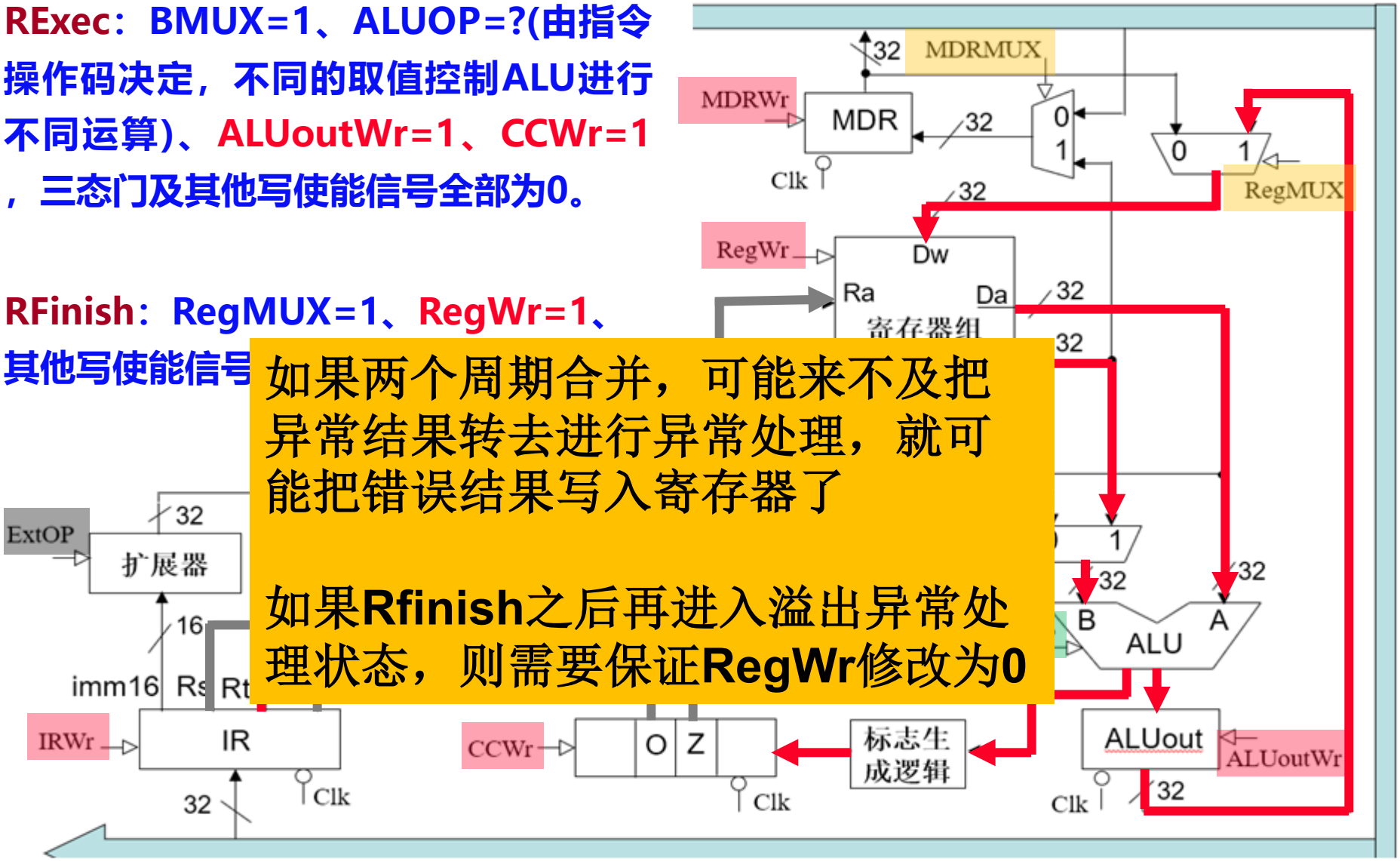
各类指令执行过程分析

（3）R-型指令的执行，需要两个时钟周期，记为 RExec、RFinish状态

进行ALU运算并将结果存入ALUout，再将相应结果分别写入Rt和CC寄存器。

RExec：BMUX=1、ALUOP=?(由指令操作码决定，不同的取值控制ALU进行不同运算)、ALUoutWr=1、CCWr=1，三态门及其他写使能信号全部为0。

RFinish：RegMUX=1、RegWr=1、其他写使能信号



如果两个周期合并，可能来不及把异常结果转去进行异常处理，就可能把错误结果写入寄存器了

如果**Rfinish**之后再进入溢出异常处理状态，则需要保证**RegWr**修改为0

各类指令执行过程分析

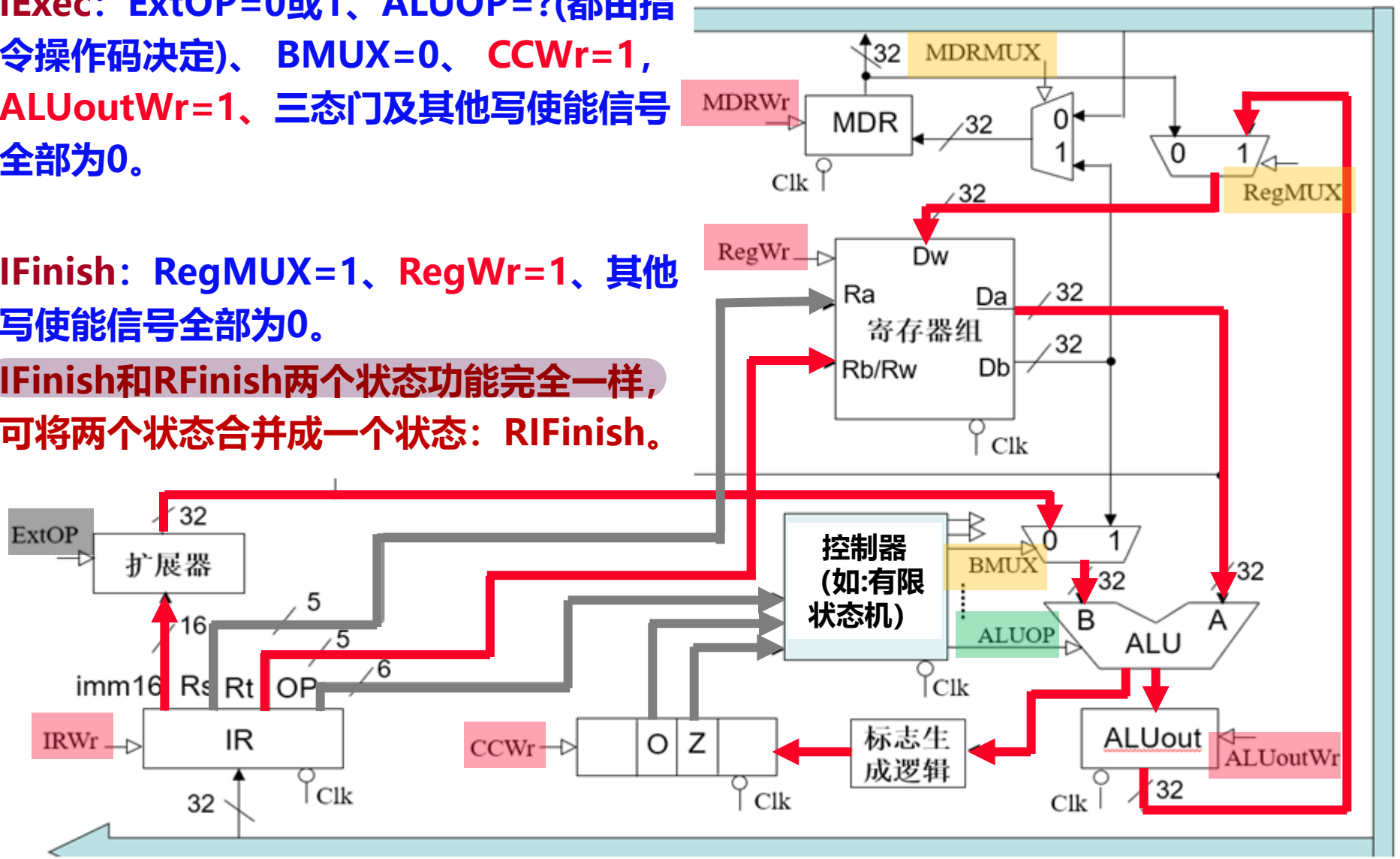
（4）I-型运算指令的执行，需要两个时钟周期，记为 IExec、IFinish状态

进行ALU运算并将结果存入ALUout，再将相应结果分别写入Rt和CC寄存器。

IExec：ExtOP=0或1、ALUOP=?(都由指令操作码决定)、 BMUX=0、 CCWr=1，ALUoutWr=1、三态门及其他写使能信号全部为0。

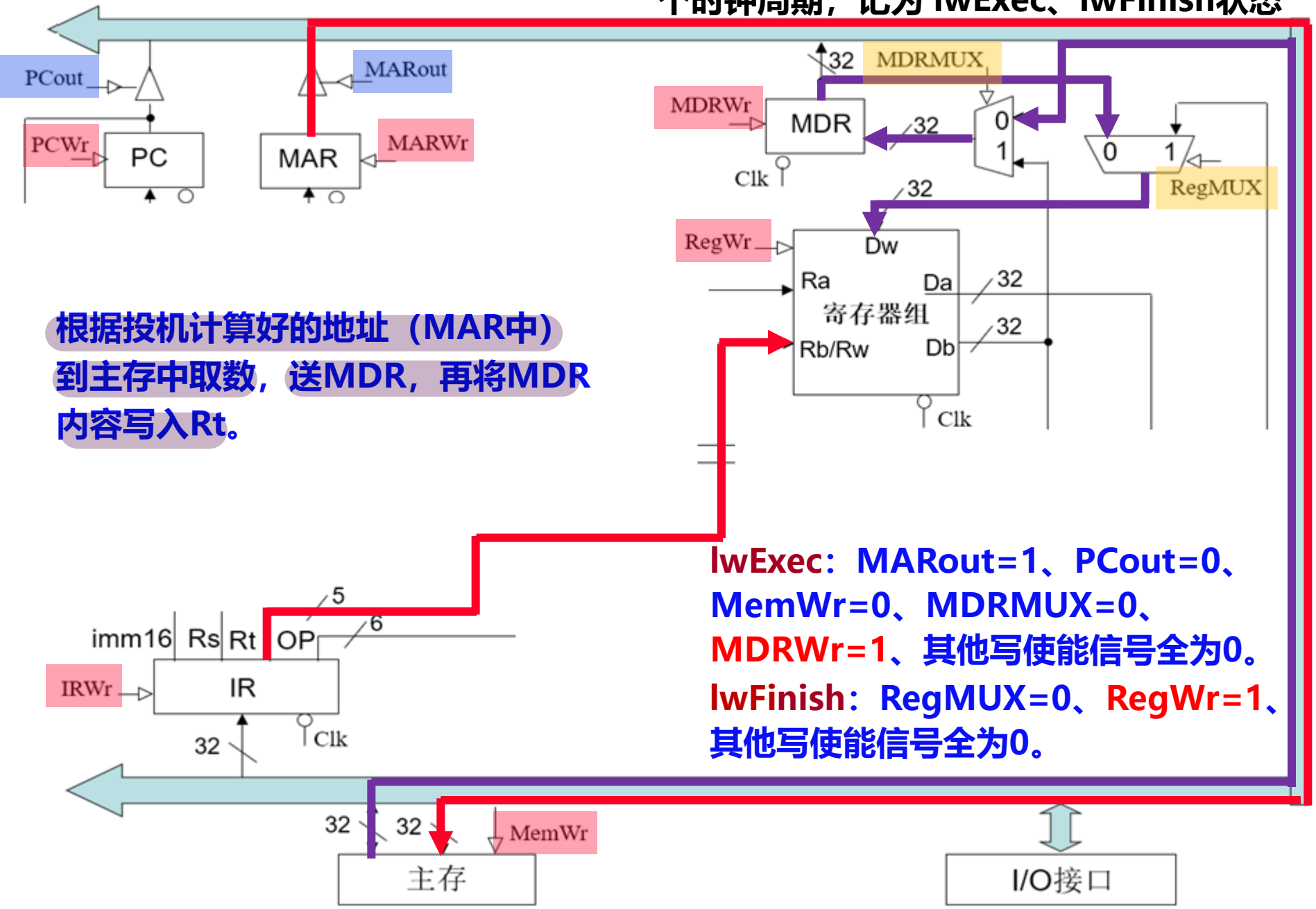
IFinish：RegMUX=1、RegWr=1、其他写使能信号全部为0。

IFinish和RFinish两个状态功能完全一样，可将两个状态合并成一个状态：RIFinish。



各类指令执行过程分析

（5）Load指令：地址已投机计算，还需两个时钟周期，记为 lwExec、lwFinish状态



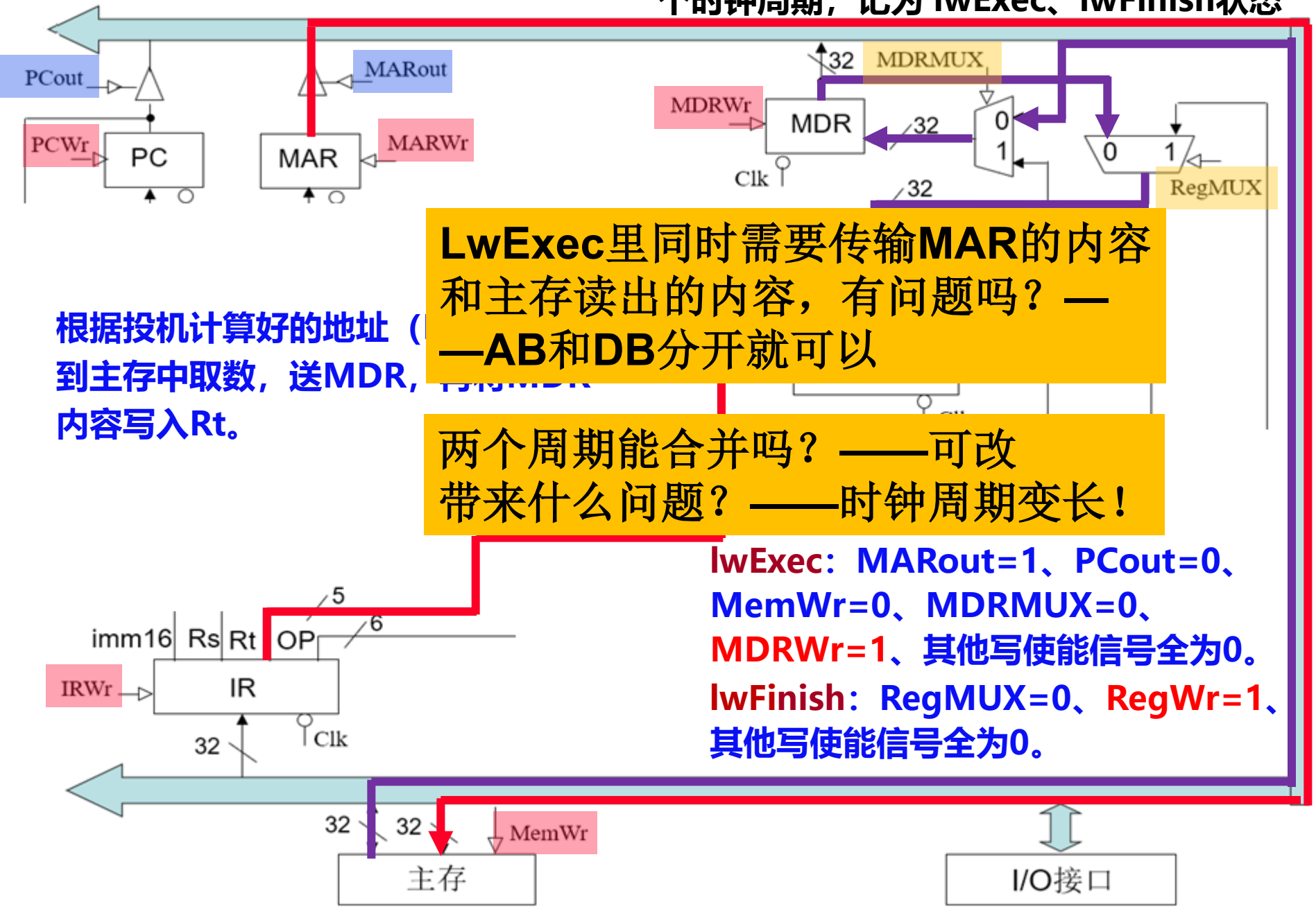
根据投机计算好的地址（MAR中）到主存中取数，送MDR，再将MDR内容写入Rt。

lwExec：MARout=1、PCout=0、MemWr=0、MDRMUX=0、MDRWr=1、其他写使能信号全为0。

lwFinish：RegMUX=0、RegWr=1、其他写使能信号全为0。

各类指令执行过程分析

（5）Load指令：地址已投机计算，还需两个时钟周期，记为 lwExec、lwFinish状态



根据投机计算好的地址（
到主存中取数，送MDR，
内容写入Rt。

LwExec里同时需要传输MAR的内容和主存读出的内容，有问题吗？——AB和DB分开就可以

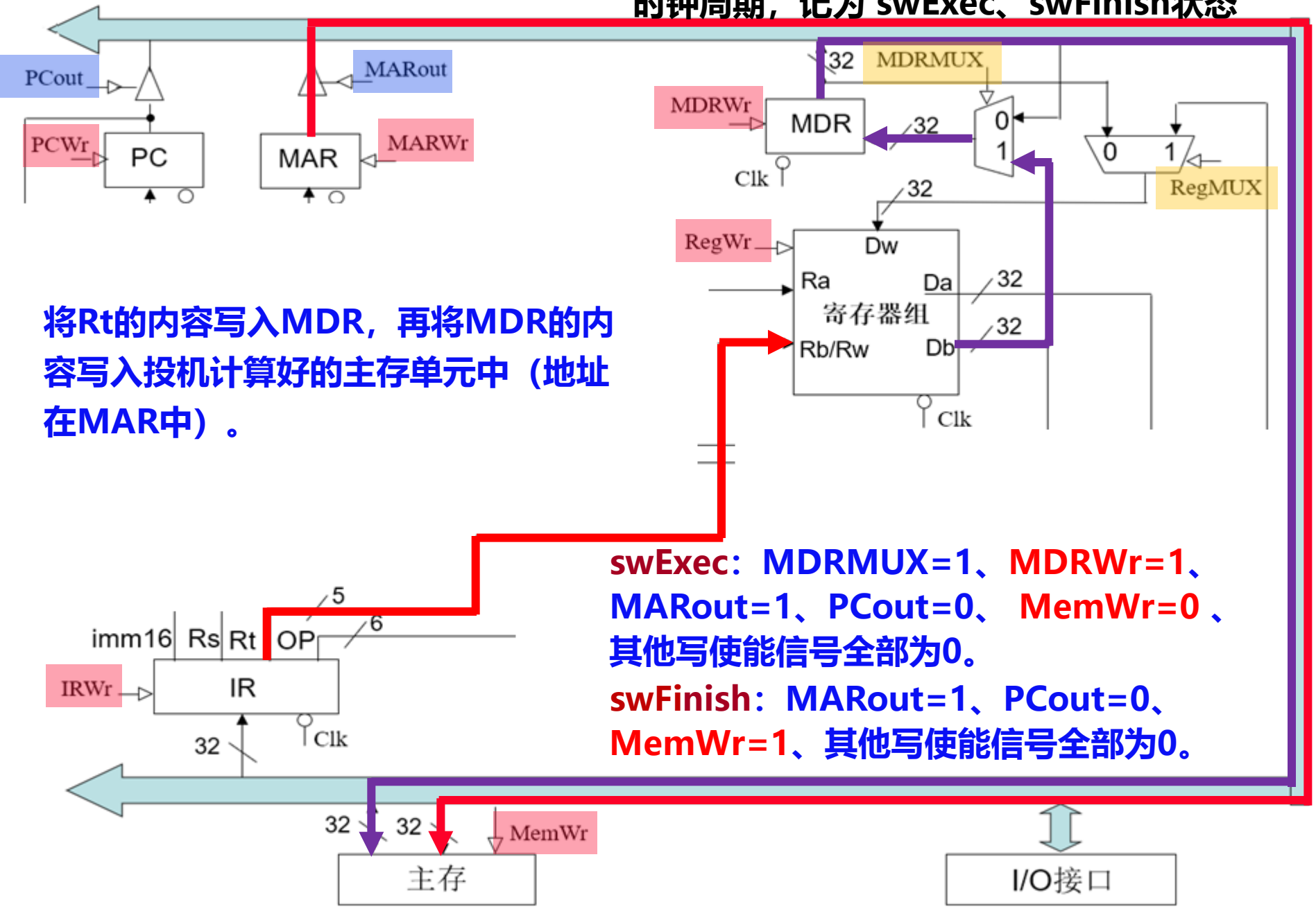
两个周期能合并吗？——可改
带来什么问题？——时钟周期变长！

lwExec：MARout=1、PCout=0、MemWr=0、MDRMUX=0、MDRWr=1、其他写使能信号全为0。

lwFinish：RegMUX=0、RegWr=1、其他写使能信号全为0。

各类指令执行过程分析

（6）Store指令：地址已投机计算，还需两个时钟周期，记为 swExec、swFinish状态



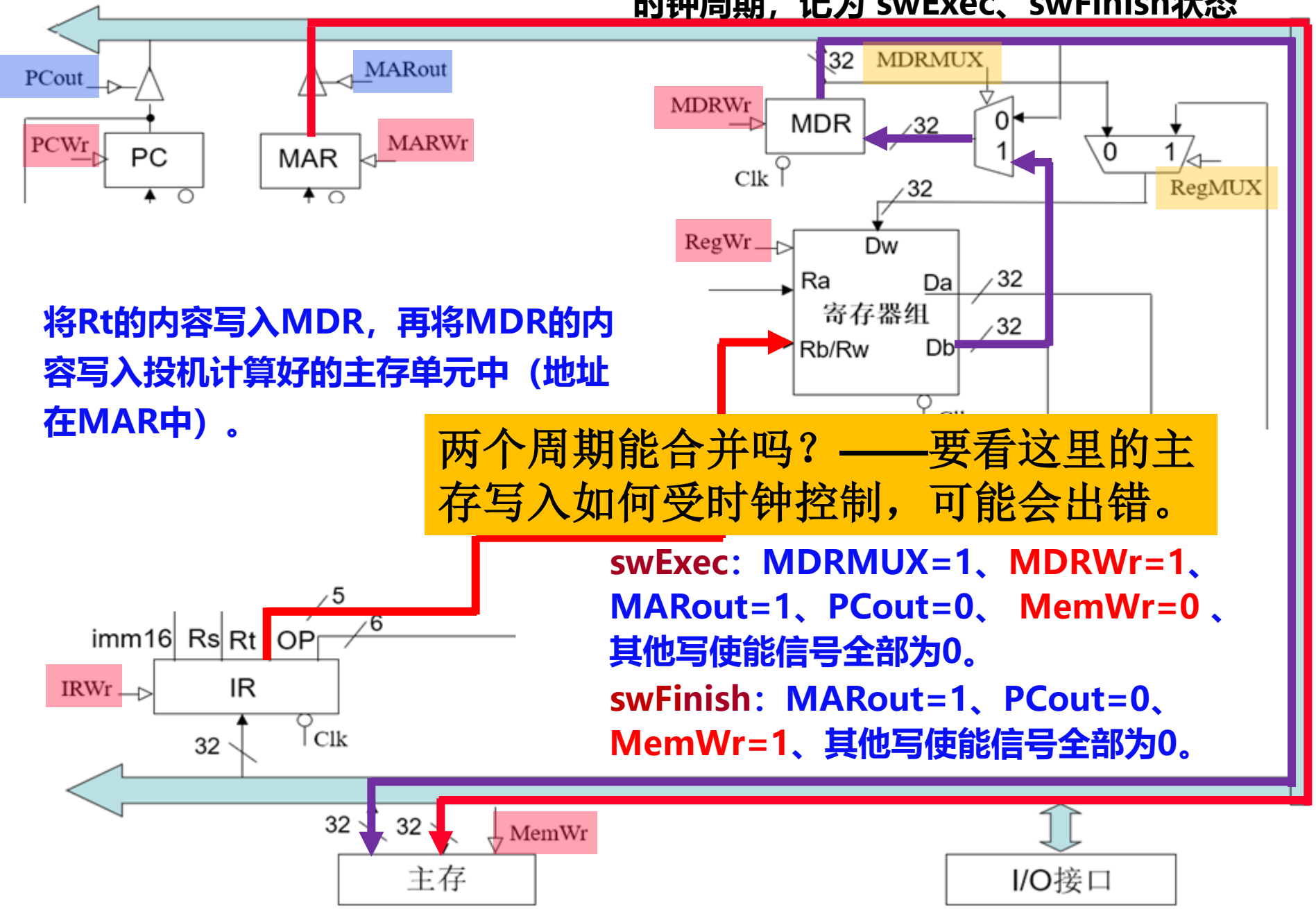
将Rt的内容写入MDR，再将MDR的内容写入投机计算好的主存单元中（地址在MAR中）。

swExec：MDRMUX=1、MDRWr=1、MARout=1、PCout=0、 MemWr=0 、其他写使能信号全部为0。

swFinish：MARout=1、PCout=0、MemWr=1、其他写使能信号全部为0。

各类指令执行过程分析

（6）Store指令：地址已投机计算，还需两个时钟周期，记为 swExec、swFinish状态



将Rt的内容写入MDR，再将MDR的内容写入投机计算好的主存单元中（地址在MAR中）。

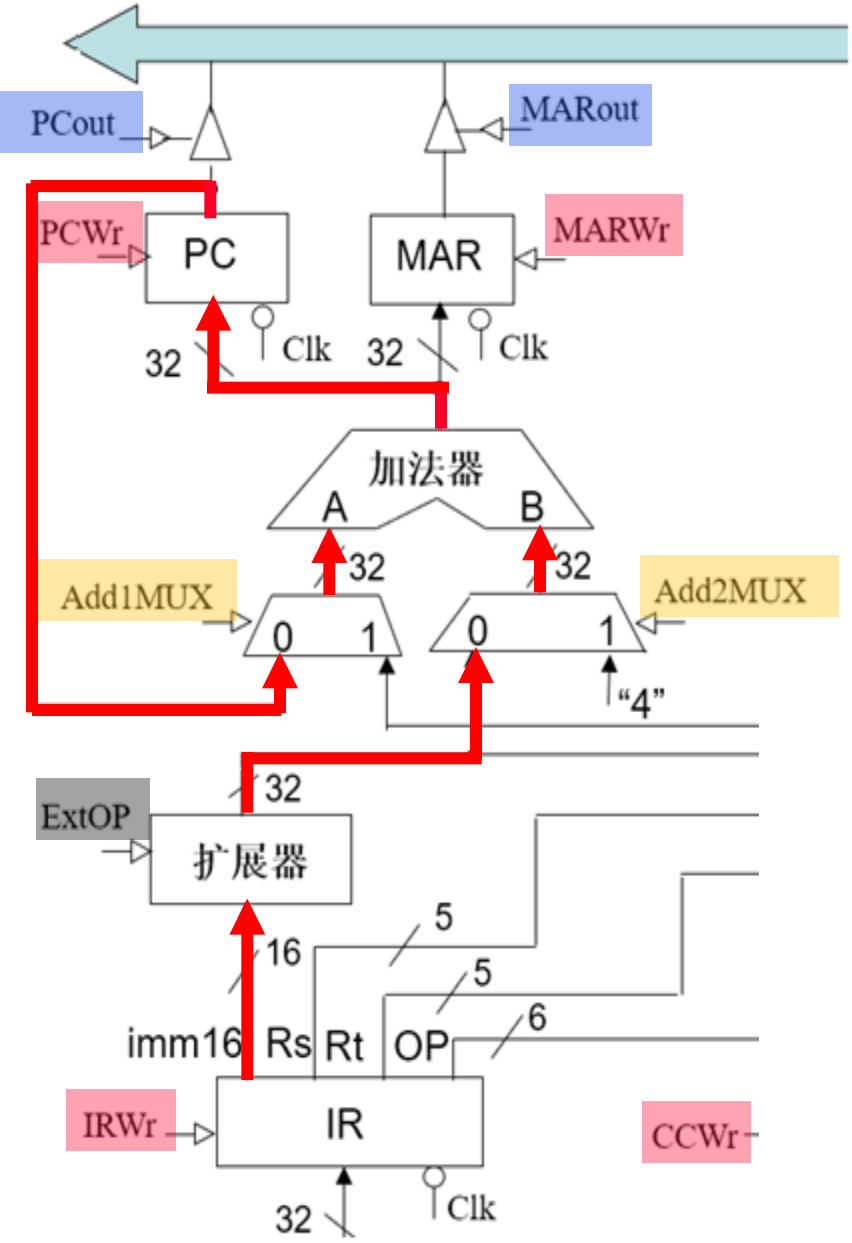
两个周期能合并吗？——要看这里的主存写入如何受时钟控制，可能会出错。

swExec：MDRMUX=1、MDRWr=1、MARout=1、PCout=0、 MemWr=0 、其他写使能信号全部为0。

swFinish：MARout=1、PCout=0、MemWr=1、其他写使能信号全部为0。

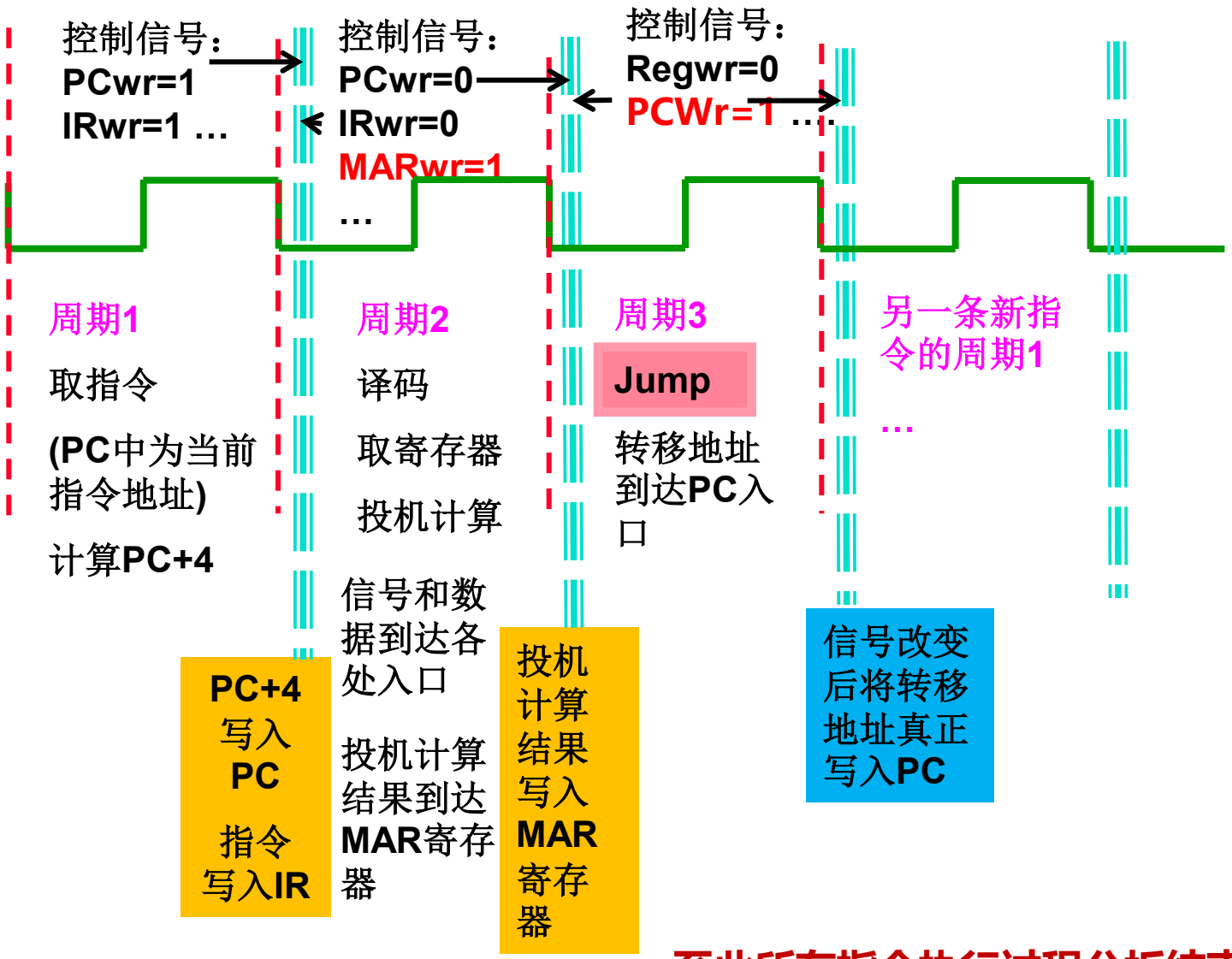
各类指令执行过程分析

(7) Jump指令：计算转移目标地址(PC+SEXT（imm6）)并送PC，需一个时钟周期，记为 JFinish状态



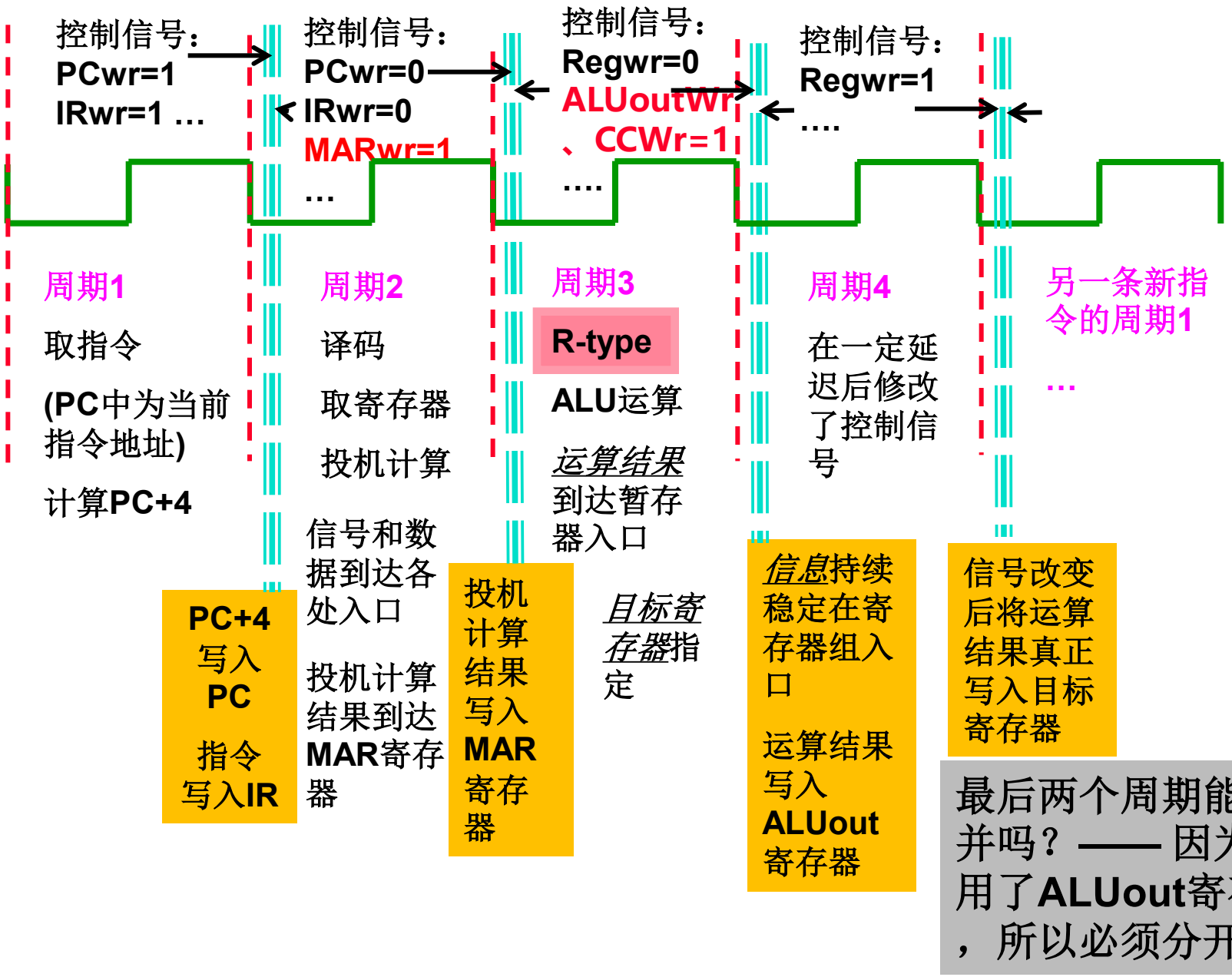
JFinish：ExtOP=1、Add1MUX=0、Add2MUX=0、PCWr=1、其他写使能信号全部为0。

状态元件内容和控制信号取值的改变：在时钟到来后的一定延时后发生



至此所有指令执行过程分析结束，下面设计控制部件，以支持能够完成上述指令功能！

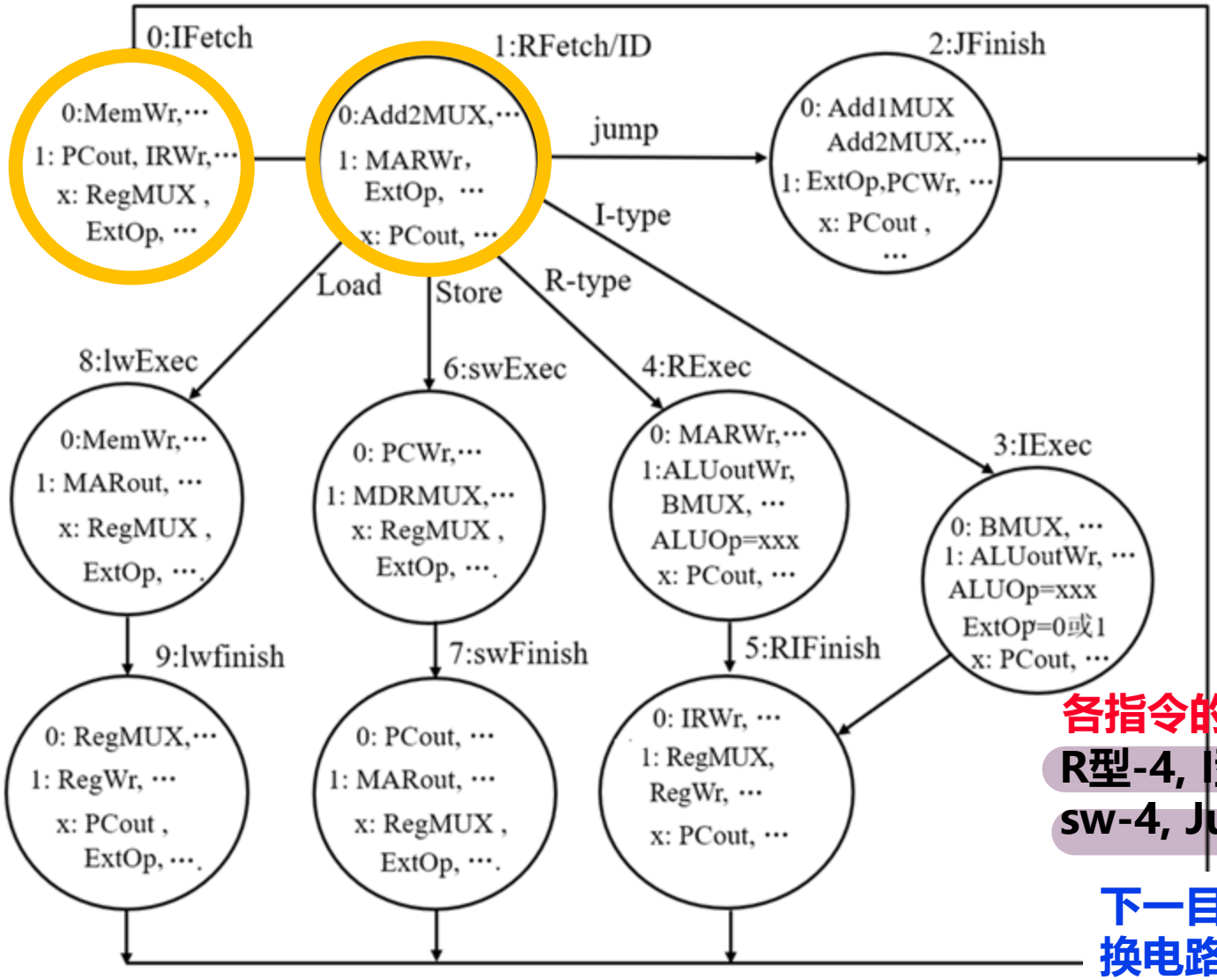
状态元件内容和控制信号取值的改变：在时钟到来后的一定延时后发生（R型指令，自行了解）



状态转换图

综合前面各指令执行过程，得到状态转换图

状态0和1是公共操作，状态1后译码得到不同的指令



每来一个时钟，进入下一个状态

各指令的时钟数多少？

R型-4, I型运算-4, lw-4, sw-4, Jump-3

下一目标：设计状态转换电路，即：控制器

多周期控制器的实现

回忆单周期控制器的实现：控制信号在整个指令执行过程中不变，用真值表能反映指令和控制信号的关系。根据真值表就能实现控制器！

多周期控制器能不能这样做?

——每个指令有多个周期

——每个周期控制信号取值不同！

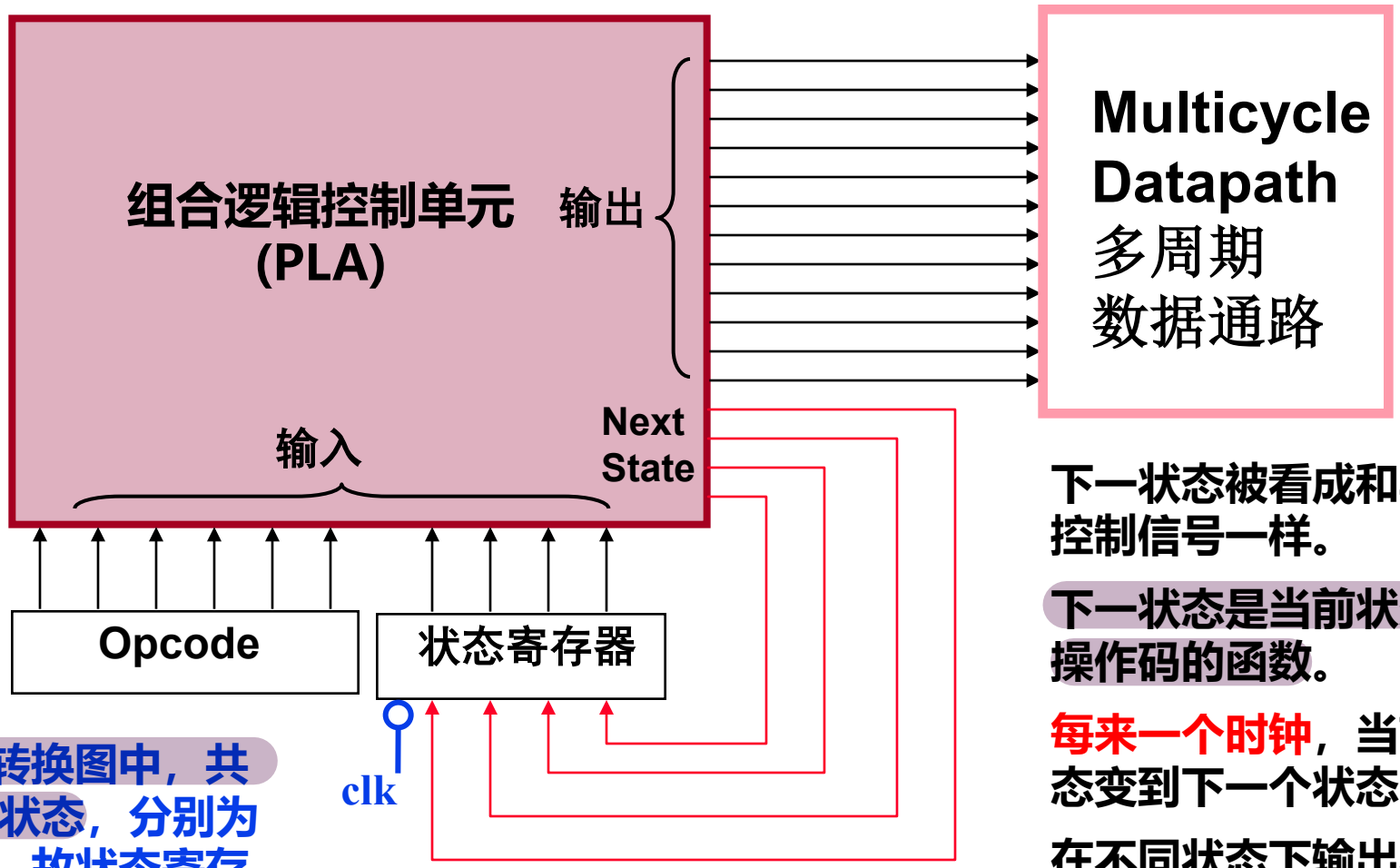
多周期控制器功能描述方式：

- **有限状态机：用硬连线路(PLA)实现**
- **微程序：用ROM存放微程序实现**

有限状态机（PLA）控制方式

由时钟、当前状态和操作码确定下一状态。不同状态输出不同控制信号值

控制逻辑采用“摩尔机”方式，即：输出函数仅依赖于当前状态



下一状态被看成和其他控制信号一样。

下一状态是当前状态和操作码的函数。

每来一个时钟，当前状态变到下一个状态

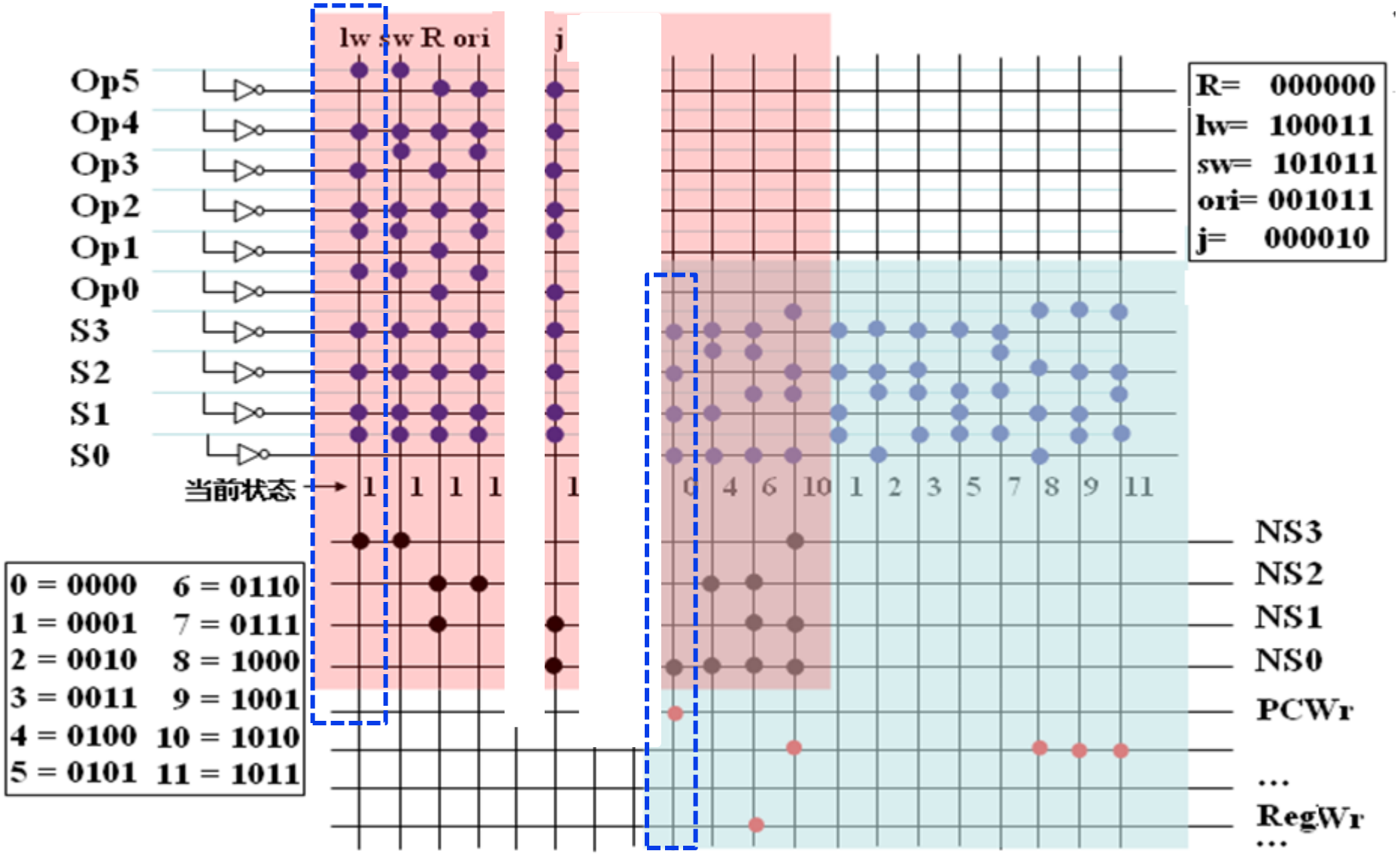
在不同状态下输出不同的控制信号。

状态转换图中，共10个状态，分别为0~9，故状态寄存器至少需4位

状态转换表——最终由PLA电路来实现

当前状态 $S_3S_2S_1S_0$	指令操作码 $OP_5OP_4OP_3OP_2OP_1OP_0$	下一状态 $NS_3NS_2NS_1NS_0$
State2、5、7、9		0 0 0 0
State0 (IFetch)		0 0 0 1
State1 (RFetch/ID)	000010 (jump)	0 0 1 0
	001101 (ori)	0 0 1 1
	000000 (R-type)	0 1 0 0
State3 (IExec)		0 1 0 1
State4 (RExec)		0 1 0 1
State1 (RFetch/ID)	101011 (sw)	0 1 1 0
	100011 (lw)	1 0 0 0
State6 (swExec)		0 1 1 1
State8 (lwExec)		1 0 0 1

*示意：用PLA电路实现控制单元（硬布线方式）



左上角：由操作码和当前状态确定下一状态的电路

右下角：由当前状态确定控制信号的电路

硬连线控制器的特点：

优点：速度快，适合于简单或规整的指令系统

缺点：它是一个多输入/多输出的巨大逻辑网络。对于复杂指令系统来说，结构庞杂，实现困难；修改、维护不易；灵活性差。甚至无法用有限状态机描述！

简化控制器设计的一个方法：微程序设计

微程序控制器设计

微程序控制器的基本思想：

- **仿照程序设计的方法，编制每个指令对应的微程序**
  - **每个微程序由若干条微指令构成，分别和各状态对应**
  - **每条微指令包含若干条微命令，分别和状态中的控制信号对应**

- **所有微程序放在只读存储器中（称为控制存储器Control Storage，简称控存CS ），都是0/1序列**

微程序设计的特点：具有规整性、可维性和灵活性，但速度慢。

微程序\微指令\微命令\微操作的关系

执行某条指令时

- 从CS中取出对应微程序
- 执行微程序，就是执行其中的各条微指令
- 对微指令译码就是产生对应的微命令——控制信号
- 由微命令控制数据通路的执行

控制程序执行要解决什么问题?

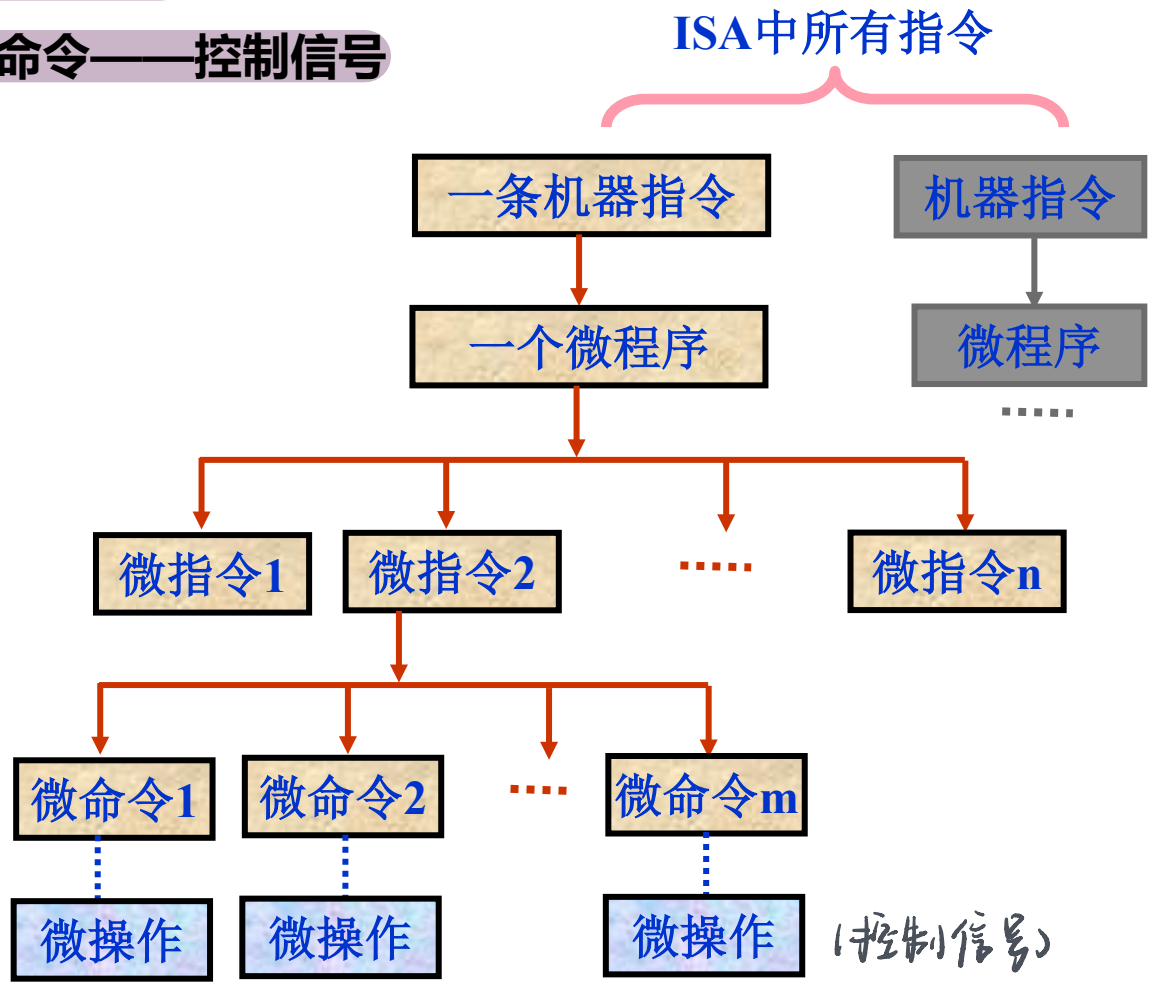
(1) 指令如何译码、执行

(2) 下条指令到哪里去取

微程序执行也要解决两个问题:

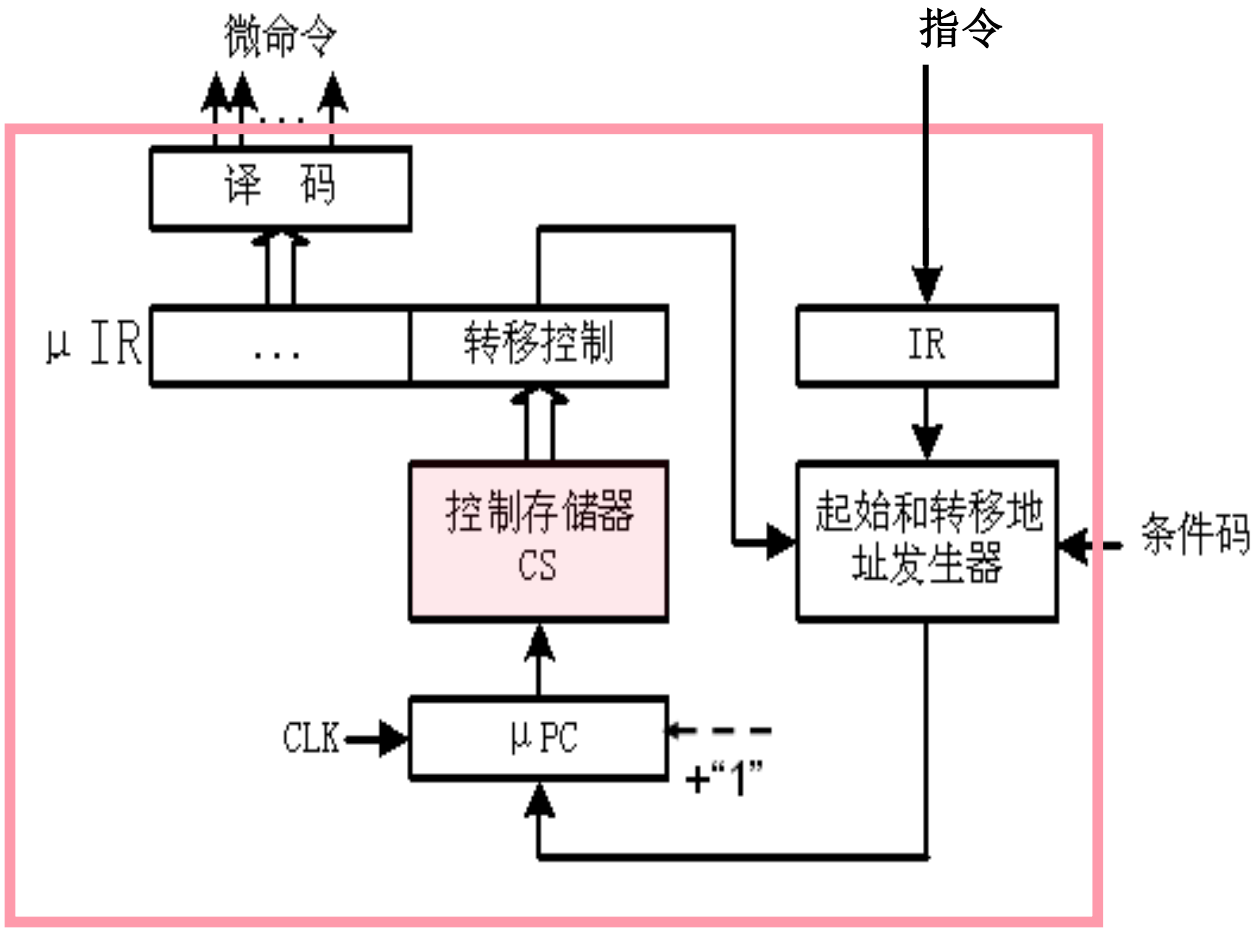
(1)微指令如何对微命令编码

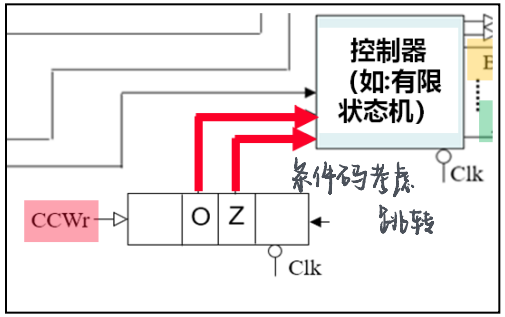
(2)下条微指令在哪里



微程序控制器的基本结构

- 输入：指令、条件码
- 输出：控制信号(微命令)
- 核心：控存CS
- μPC：指出将要执行的微指令在CS中的位置
- μIR：正在执行的微指令
- 每个时钟执行一条微指令
- 微程序第一条微指令地址由起始地址发生器产生
- 顺序执行时，μPC+1
- 转移执行时，由转移控制字段指出对哪些条件码进行测试，转移地址发生器根据条件码修改μPC





第一个问题：微指令格式的设计

微指令中包含了：若干微命令、下条微指令地址（可选）、常数（可选）

微指令格式：

μOP: 微操作码字段，产生微命令（控制信号）；

μAddr（配合常数）：微地址码字段，产生下条微指令地址。

- 微指令格式设计风格取决于微操作码的编码方式
- 微操作码编码方式：

不译法（直接控制法）

字段直接编码（译）法

字段间接编码（译）法

} 水平型微指令风格（微指令长，微程序短）

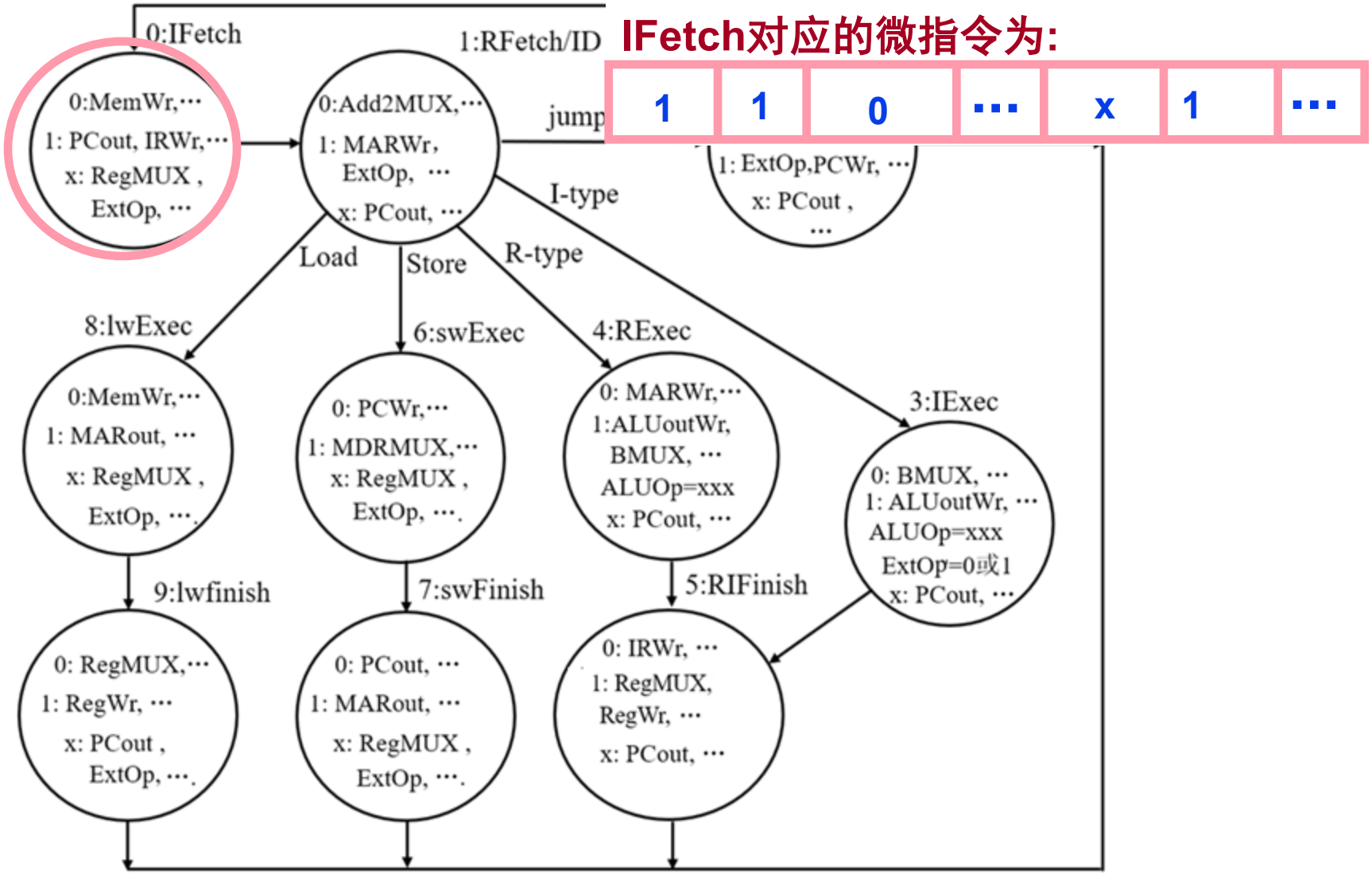
最小（最短、垂直）编码（译）法 —— 垂直型微指令风格（微指令短，微程序长）

采用不译法的微操作码格式为：

PCWr	IRWr	MemWr	……	ExtOp	PCout	……

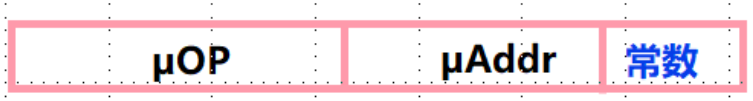
IFetch对应的微指令为:

1	1	0	…	x	1	…



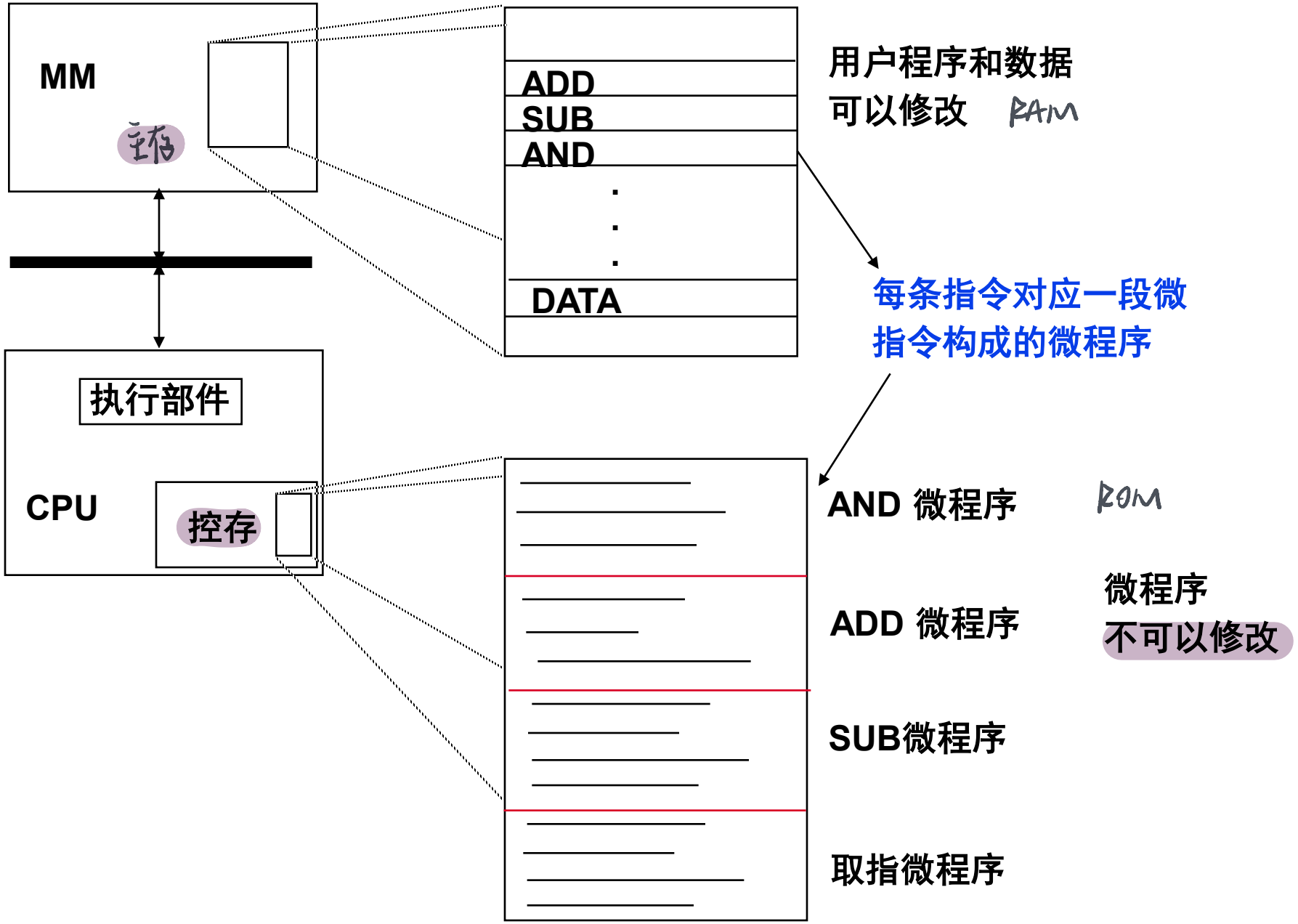
第二个问题：下条微地址的确定方式

- 什么是微程序执行顺序的控制问题?
  - 指在现行微指令执行完毕后，怎样控制产生下一条微指令的地址。
- 怎样控制微程序的执行顺序?



  - 通过在本条微指令中明显或隐含地指定下条微指令在控存中的地址来控制。
- 微指令地址的产生方法有两种：　问题：指令是隐含还是明显指出下条指令地址的?
  - 增量(计数器)法：下条微指令地址隐含在微程序计数器μPC中。
  - 断定(下址字段)法：在本条微指令中明显指定下条微指令地址。
- 选择下条要执行的微指令有以下四种情况：
  - 取指微程序首址：每条指令执行前，CPU先执行取指微程序。(公共部分)
  - 第一条微指令：每条指令取出后，必须转移到该指令对应的第一条微指令执行。
  - 顺序执行时：微程序执行过程中顺序取出下条微指令执行。　↳ 取指后，可能转移
  - 分支执行时：在遇到按条件转移到不同微指令执行时，需要根据控制单元的输入来选择下条微指令。

微指令字的解释执行


MM
主存
ADD
SUB
AND
.
.
.
DATA
用户程序和数据
可以修改
RAM
每条指令对应一段微
指令构成的微程序
执行部件
CPU
控存
AND 微程序
ROM
ADD 微程序
微程序
不可以修改
SUB微程序
取指微程序

回顾：异常和中断的处理

- 程序执行过程中CPU会遇到一些特殊情况，使正在执行的程序被“中断”
  - 此时，CPU中止原来正在执行的程序，转到处理异常情况或特殊事件的程序去执行，执行后再返回到原被中止的程序处（断点）继续执行
- 程序执行被 “中断” 的事件有两类
  - 内部“异常”：在CPU内部发生的意外事件或特殊事件，发现立刻处理

    按发生原因分为硬故障中断和程序性中断两类

    硬故障中断：如电源掉电、硬件线路故障等

    程序性中断：执行某条指令时发生的“例外(Exception)”，如溢出、缺页、越界、越权、非法指令、除数为0、堆栈溢出、访问超时、断点设置、单步、系统调用等
  - 外部“中断”：在CPU外部发生的特殊事件，通过“中断请求”信号向CPU请求处理。如实时钟、控制台、打印机缺纸、外设准备好、采样计时到、DMA传输结束等。——每个指令执行结束，CPU查询有没有中断请求，有则响应中断

处理器中的异常处理机制

检测到异常时，处理器必须进行以下基本处理：

① 关中断（“中断/异常允许”状态位清0）：使处理器处于“禁止中断”状态，以防止新异常(或中断)破坏断点、程序状态和现场（现场指通用寄存器的值）。

② 保护断点和程序状态：将断点和程序状态保存到堆栈或特殊寄存

PC→栈 或 专门存放断点的寄存器。

PSWR →栈 或 EPSWR （专门保存程序状态的寄存器）

③ 识别异常事件：

软件识别

硬件识别（向量中断方式）

识别异常事件：软件识别和硬件识别

(1) 软件识别（RISC-V、MIPS等采用）

设置一个异常原因寄存器（如RISC-V和MIPS中的Cause寄存器），用于记录异常原因。操作系统使用一个统一的异常处理程序，该程序按优先级顺序查询异常状态寄存器，识别出异常事件。例如：

RISC-V中由某个CSR（如M-模式下的mtvec寄存器）所定义的地址作为异常处理程序的首地址；MIPS中位于内核地址0x8000 0180处有一个专门的异常处理程序。

该异常处理程序会检测Cause寄存器以查明异常的具体原因，然后转到内核中相应的异常处理程序段中进行具体的处理。

(2) 硬件识别（向量中断）（80x86采用）

每个异常和中断都有一个异常/中断号，根据此号，到中断向量表（中断描述符表）中读取对应的具体的中断服务程序的入口地址。

RISC-V架构也支持通过硬件识别方式来识别外部中断源。

异常和中断机制是处理器设计中最具挑战性的任务之一

*

- RISC-V架构定义了一些控制和状态寄存器（Control and Status Register，CSR），用于配置或记录一些运行的状态。
- CSR寄存器是处理器核内部的寄存器，使用自己的地址编码空间，和存储器寻址的地址区间完全无关系。
- CSR寄存器的访问采用专用的CSR指令，包括CSRRW、CSRRS、CSRRC、CSRRWI、CSRRSI以及CSRRCI指令。

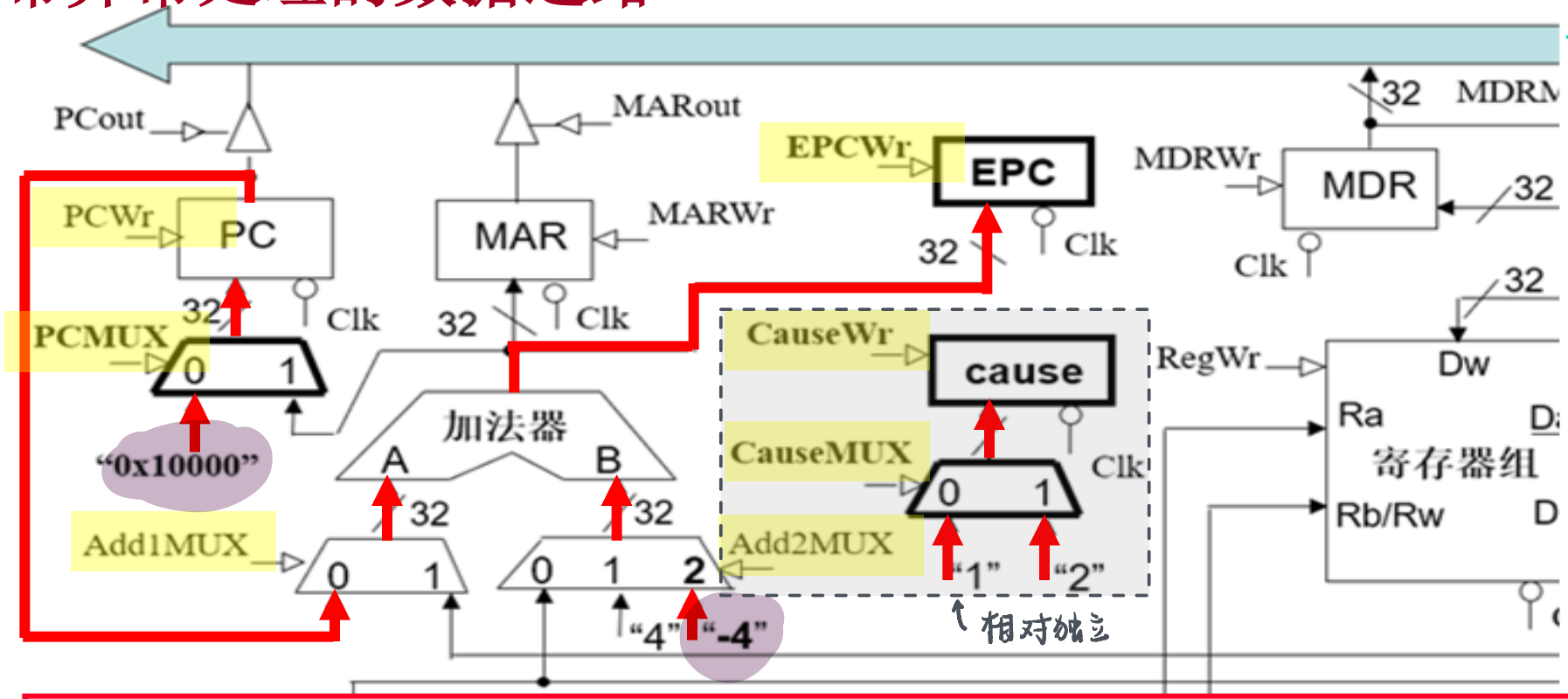
Register	Address
mstatus	0x300
misa	0x301
medeleg	0x302
mideleg	0x303
mie	0x304
mtvec	0x305
mcounterer	0x306
mverndorid	0xf11
marchid	0xf12
mimpid	0xf13
mhartid	0xf14
mscratch	0x340
mepc	0x341
mcause	0x342
mtval	0x343
mip	0x344
mcycle	0xb00
mcycleh	0xb80
minstret	0xb02
minstreth	0xb82

带异常处理的数据通路设计

以前述多周期处理器为例，简要说明带异常处理数据通路设计

- 假定其异常/中断机制中包括以下两个寄存器：
  - EPC：32位，用于存放断点（异常处理后返回到的指令的地址）。
    - 写入EPC的断点可能是正在执行的指令的地址（故障时）
    - 也可能是下条指令的地址（自陷和中断时）
    - 前者需要把PC的值减4后送到EPC，后者则直接送PC到EPC
  - Cause：32位（有些位还没有用到），记录异常原因。
    - 假定处理的异常类型有以下两种：
      未定义指令（Cause=1）、溢出（Cause=2）
- 需要加入两个寄存器的“写使能”控制信号
  - EPCWr：在保存断点时该信号有效，使断点PC写入EPC。
  - CauseWr：在处理器发现异常（如：非法指令、溢出）时，该信号有效，使异常类型被写到Cause寄存器。
- 需要一个控制信号CauseMUX来选择正确的值写入到Cause中
- 需要将异常查询程序的入口地址（假设为0x10000）写入PC，可以在PC输入端增加一个MUX（控制信号PCMUX），其中一个输入为0x10000

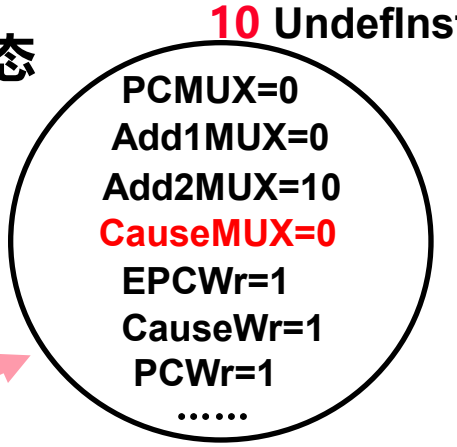
带异常处理的数据通路



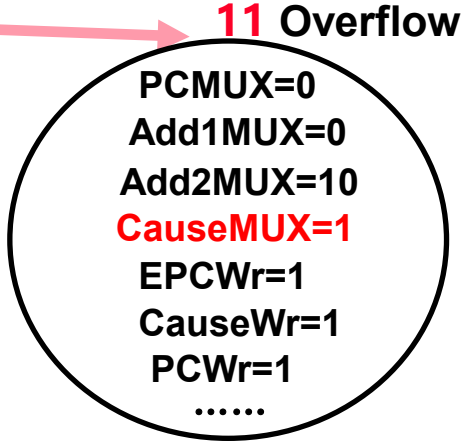
- 假设只要处理两种故障类异常：未定义指令（Cause=1）、溢出（Cause=2）
- 故障类异常的断点为：PC减4。因为状态0（IFetch）时，已执行PC加4
- Cause寄存器的两个输入端，分别是 1（未定义指令时）和 2（溢出时）
- PC中将设置异常处理程序的首地址（假设为0x10000）
- 若发生异常，则Add1MUX=0， Add2MUX=10，EPCWr=1，CauseWr=1，PCMUX=0，PCWr=1，其他写使能信号都为0，
- 发生未定义指令时CauseMUX=0，发生溢出时CauseMUX=1。

带异常处理的控制器设计

- 在有限状态机中增加异常处理的状态，每种异常占一个状态
- 每个异常处理状态中，需考虑以下基本控制
  - Cause寄存器的设置
  - 计算断点处的PC值（PC-4），并送EPC
  - 将异常查询程序的入口地址送PC
- 假设要控制的数据通路中有以下两种异常处理
  - 非法操作码（Cause=0）：状态10
  - 溢出（Cause=1）：状态11
- 在原来状态转换图基础上加入两个异常处理状态
  - 如何检测是否发生了这两种异常?
    - 未定义指令（非法操作码）：当指令译码器发现op字段是一个未定义的编码时
    - 溢出：当R-型或I-型运算类指令在ALU中执行后，在条件码寄存器CC中的标志O为1时

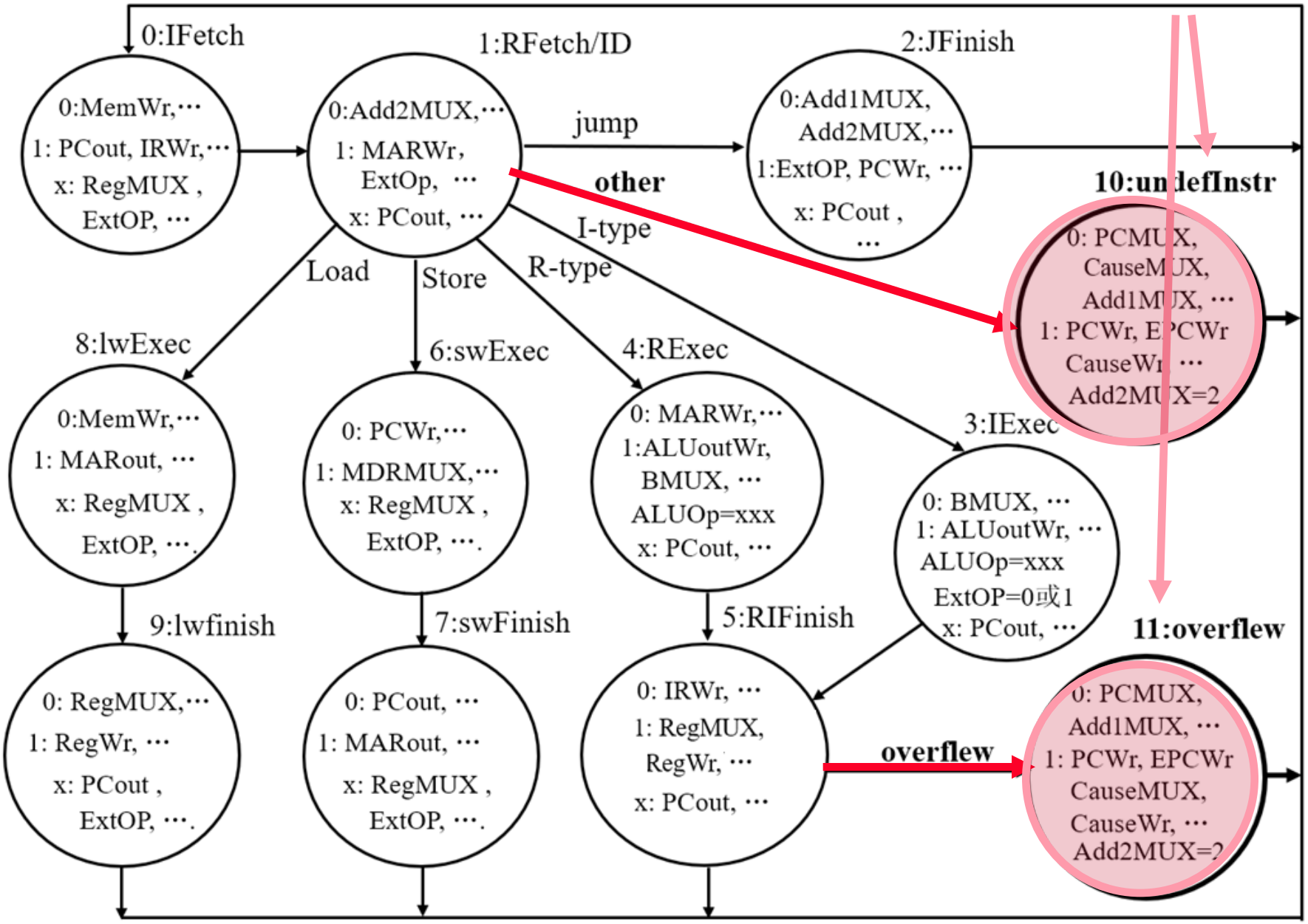


10 未定义指令异常状态

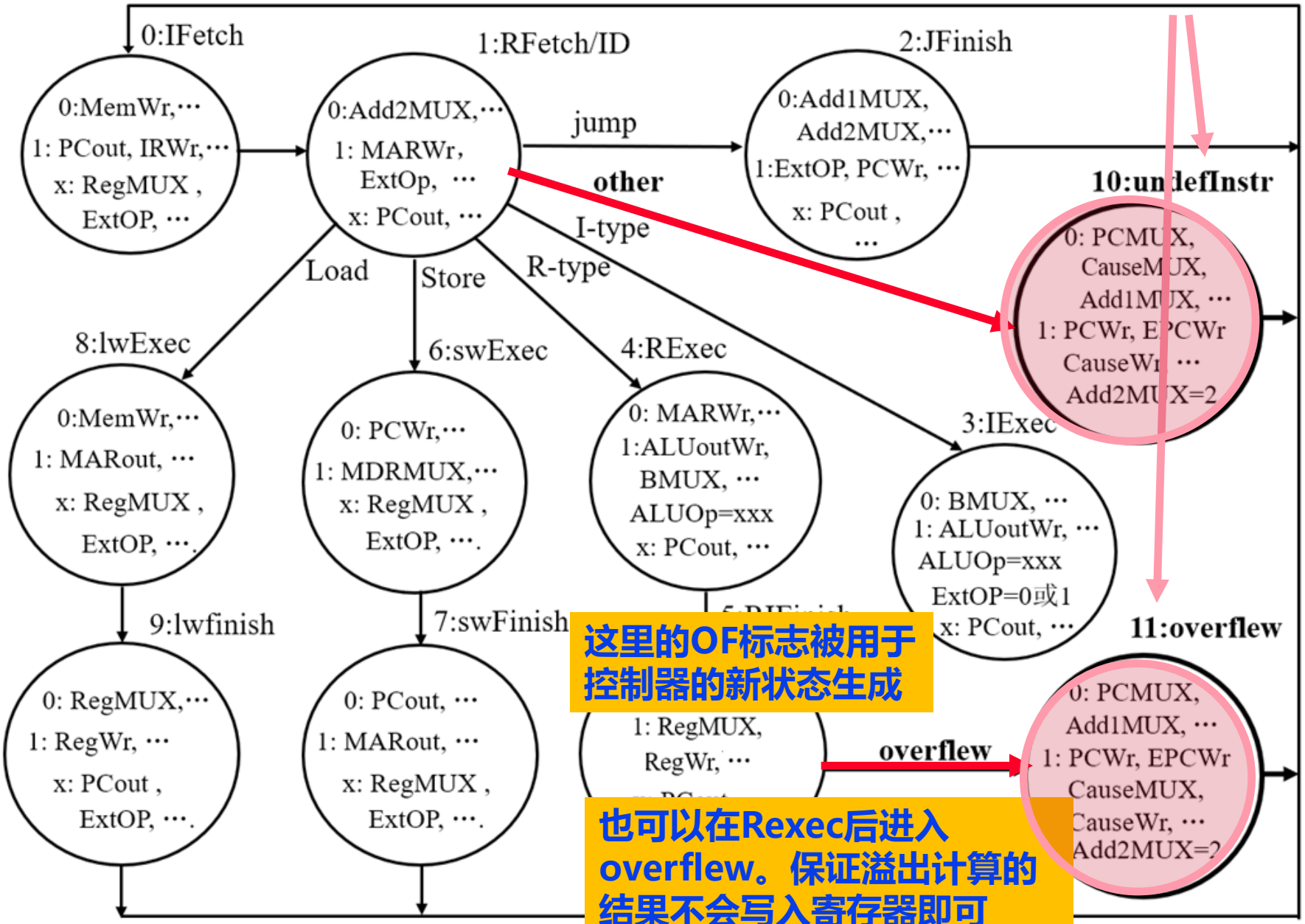


11 溢出异常状态

加入异常处理后的有限状态转换图



加入异常处理后的有限状态转换图



单周期和多周期的CPU比较

- 成本比较：
  - 单周期下功能部件不能重复使用；而多周期下可重复使用，比单周期省
  - 单周期指令执行结果直接保存在PC、Regfile和Memory；而多周期下需加一些临时寄存器保存中间结果，比单周期费
- 性能比较：
  - 单周期CPU的CPI为1，但时钟周期为最长的load指令执行时间
  - 多周期CPU的CPI是多少？时钟周期多长？

    假定程序中22%为Load，11%为Store，49%为R-Type，16%为I-Type，2%为Jump。每个状态需要一个时钟周期，CPI为多少？

    若每种指令所需的时钟周期数为：

    只有jump是3

    Load：4；Store：4；R-Type：4；I-Type：4；Jump：3

    则CPI计算如下：

    CPI=CPU时钟周期数 / 指令数 = Σ（指令数$_i$ x CPI$_i$）/ 指令数

    = Σ（指令数$_i$ / 指令数）x CPI$_i$

    CPI = 0.22x4+0.11x4+0.49x4+0.16x4+0.02x3 = 3.98

    假设单周期时钟宽度为1，且多周期时钟周期约为单周期的1/5，则，

    多周期的总体时间约：3.98x1/5=0.796；而单周期总体时间为：1x1=1

    这种情况下：多周期比单周期效率高！

单周期和多周期的CPU比较

- 成本比较：
  - 单周期下功能部件不能重复使用；而多周期下可重复使用，比单周期省
  - 单周期指令执行结果直接保存在PC、Regfile和Memory；而多周期下需加一些临时寄存器保存中间结果，比单周期费
- 性能比较：
  - 单周期CPU的CPI为1，但时钟周期为最长的load指令执行时间
  - 多周期CPU的CPI是多少？时钟周期多长？

    假定程序中22%为Load，11%为Store，49%为R-Type，16%为I-Type，2%为Jump。每个状态需要一个时钟周期，CPI为多少？

    若每种指令所需的时钟周期数为：

    Load：4；Store：4；R-Type：4；I-Type：4；Jump：3

    则CPI计算如下：

    CPI=CPU时钟周期数/指令数 = Σ（指令数 x CPI）/指令数

    但是，如果多周期的时钟周期是单周期的1/3

    那么多周期总体时间就是3.98/3=1.33 > 1

    所以，多周期的性能不一定比单周期好！

    关键就在于划分阶段是否均匀。

    x1=1

    这

前四讲总结1

- CPU的主要功能
  - 周而复始执行指令
  - 执行指令过程中，若发现异常情况，则转异常处理
  - 每个指令结束，查询有没有中断请求，有则响应中断
- CPU的内部结构
  - 由数据通路(Datapath)和控制单元(Control unit)组成
    - 数据通路中包含组合逻辑单元和存储信息的状态单元
      - 组合逻辑单元用于对数据进行处理，如：加法器、运算器ALU、扩展器（0扩展或符号扩展）、多路选择器、以及状态单元的读操作线路等。
      - 状态单元包括触发器、寄存器、寄存器堆、数据/指令存储器等，用于对指令执行的中间状态或最终结果进行保存。
    - 控制单元对指令进行译码，与指令执行得到的条件码或当前机器的状态、时序信号（时钟）等组合，生成对数据通路进行控制的控制信号

前四讲总结2

- CPU中的寄存器
  - 用户可见寄存器（用户可使用）
    - 通用寄存器：用来存放地址或数据，需在指令中明显给出
    - 专用寄存器：用来存放特定的地址或数据，无需在指令中明显给出
    - 数据寄存器：专用于保存数据，可以是通用或专用寄存器
    - 地址寄存器：专用于保存地址，可以是通用或专用寄存器。如：段指针、变址器、基址器、堆栈指针、栈帧指针等。
    - 标志(条件码)寄存器、程序计数器PC：部分可见。由CPU根据指令执行结果设定，只能以隐含方式读出其中若干位，用户程序（非内核程序）不能改变
  - 控制和状态寄存器（用户不可使用）
    - 指令寄存器IR
    - 存储器地址寄存器MAR
    - 存储器缓冲(数据)寄存器 MBR / MDR
    - 程序状态字寄存器PSWR
    - 临时寄存器：用于存放指令执行过程中的临时信息
    - 其他寄存器：如，进程控制块指针、系统堆栈指针、页表指针等

前四讲总结3

- 指令执行过程
  - 取指、译码、取数、运算、存结果、查中断
  - 指令周期：取出并执行一条指令的时间，由若干个时钟周期组成
  - 时钟周期：CPU中用于信号同步的信号，是CPU最小的时间单位
- 数据通路的定时方式
  - 现代计算机都采用时钟信号进行定时
  - 一旦时钟有效信号到来，数据通路中的状态单元可以开始写入信息
  - 如果状态单元每个周期都更新信息，则无需加“写使能”控制信号，否则，需加“写使能”控制信号，以使必要时控制信息写入寄存器
- 数据通路中信息的流动过程
  - 每条指令在取指令阶段和指令译码阶段都一样
  - 每条指令的功能不同，故在数据通路中所经过的部件和路径可能不同
  - 数据在数据通路中的流动过程由控制信号确定
  - 控制信号由控制器根据指令代码来生成

前四讲总结4

- 单周期处理器的设计
  - 每条指令都在一个时钟周期内完成
  - 时钟周期以最长的Load指令所花时间为准
  - 无需加临时寄存器存放指令执行的中间结果
  - 同一个功能部件不能重复使用
  - 控制信号在整个指令执行过程中不变，所以控制器设计简单，只要写出指令和控制信号之间的真值表，就可以设计出控制器
- 多周期处理器的设计
  - 每条指令分成多个阶段，每个阶段在一个时钟内完成
  - 不同指令包含的时钟个数不同
  - 阶段的划分要均衡，每个阶段只能完成一个独立、简单的功能，如：
    - 一次ALU操作
    - 一次存储器访问
    - 一次寄存器存取
  - 需加临时寄存器存放指令执行的中间结果
  - 同一个功能部件能在不同的时钟中被重复使用
  - 可用有限状态机来表示指令执行流程，并以此设计控制器

前四讲总结5

本章作业：习题3、4、5、6、11、13、14、17、18

- 控制单元实现方式
  - 有限状态机描述方式
    - 每个时钟周期包含的控制信号的值的组合看成一个状态，每来一个时钟，控制信号会有一组新的取值，也就是一个新的状态
    - 所有指令的执行过程可用一个有限状态转换图来描述
    - 用一个组合逻辑电路（一般为PLA电路）来生成控制信号，用一个状态寄存器实现状态之间的转换
    - 实现的控制器称为硬布线控制器
  - 微程序描述方式
    - 每个时钟周期所包含的控制信号的值的组合看成是一个0/1序列，每个控制信号对应一个微命令，控制信号取不同的值，就发出不同的微命令
    - 若干微命令组合成一个微指令，每条指令所包含的动作就由若干条微指令来完成，每来一个时钟，执行一条微指令
    - 每条指令对应一个微程序，执行时，先找到对应的第一条微指令，然后按照特定的顺序取出后续的微指令执行
    - 实现的控制器称为微程序控制器

第8章 中央处理器（2）

第一讲 中央处理器概述

第二讲 单周期数据通路的设计

第三讲 单周期控制器的设计

第四讲 多周期处理器的设计

第五讲 流水线处理器设计

第六讲 流水线冒险及其处理

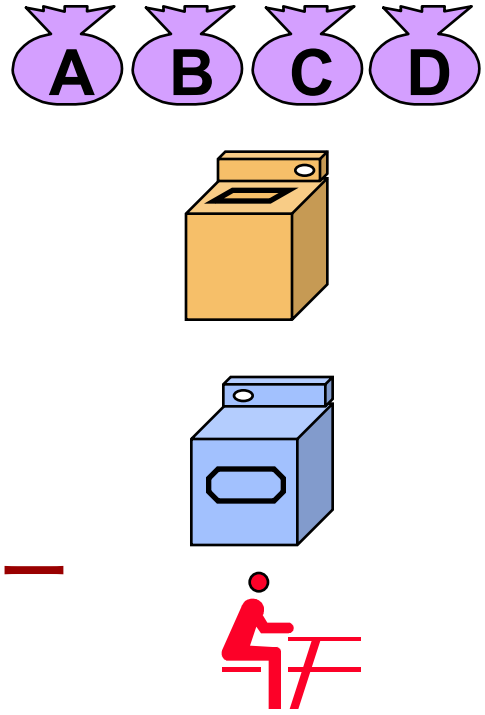
第七讲 高级流水线技术

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RV32I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

一个日常生活中的例子—洗衣服

- Laundry Example
  - A, B, C, D四个人每人有一批衣服需要 wash, dry, fold
  - Washer takes 30 minutes
  - Dryer takes 40 minutes
  - “Folder” takes 20 minutes

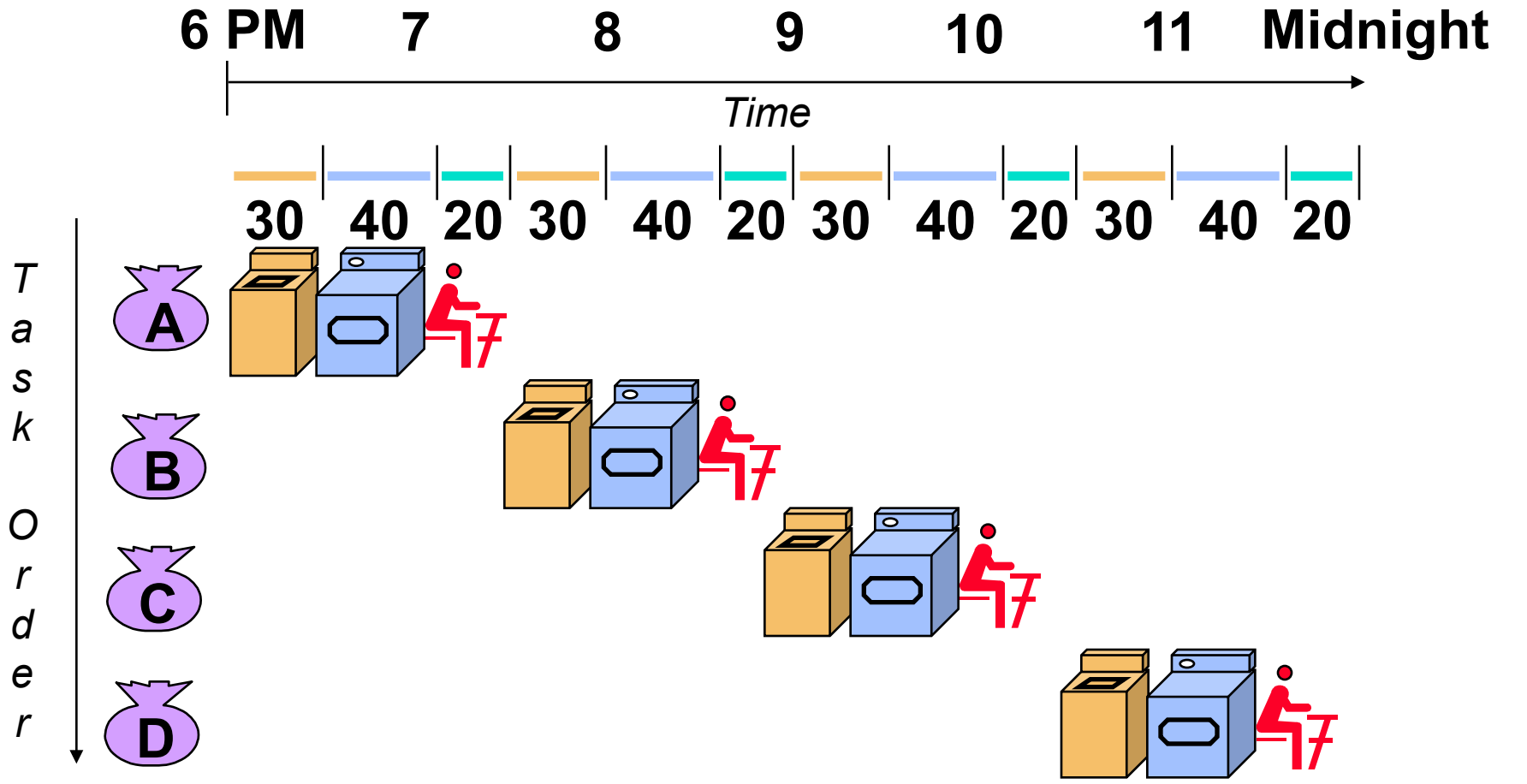


洗衣店只有一台洗衣机，一台干衣机，一个负责folder的员工。

如果让你来管理洗衣店，你会如何安排？

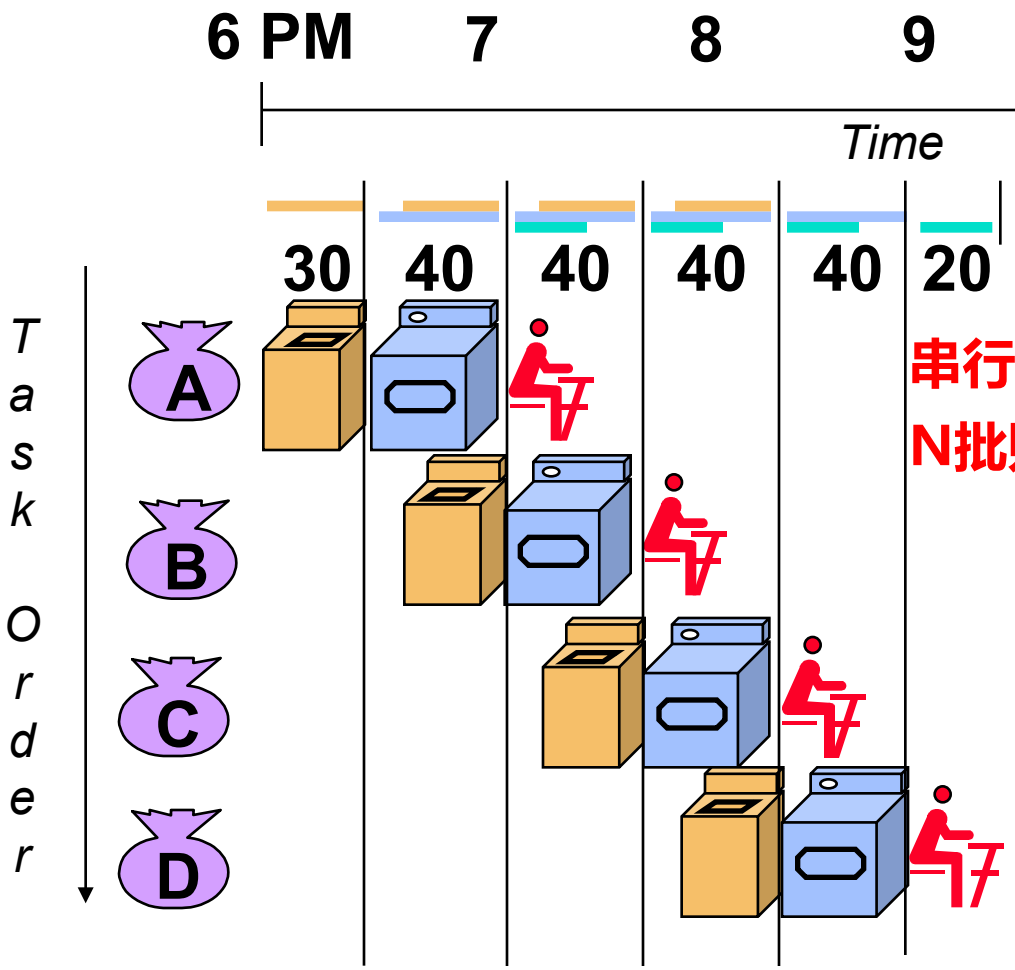
Pipelining: It’s Natural !

Sequential Laundry（串行方式）



- 串行方式下， 4 批衣服需要花费 6 小时（4x(30+40+20)=360分钟）
- N批衣服，需花费的时间为Nx(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)



串行为90分钟x4=6小时

N批则为90xN分钟

流水线方式下，需
30+4x40+20
=210分 (3.5小时)

如果有N批衣服呢?

30+Nx40+20分钟

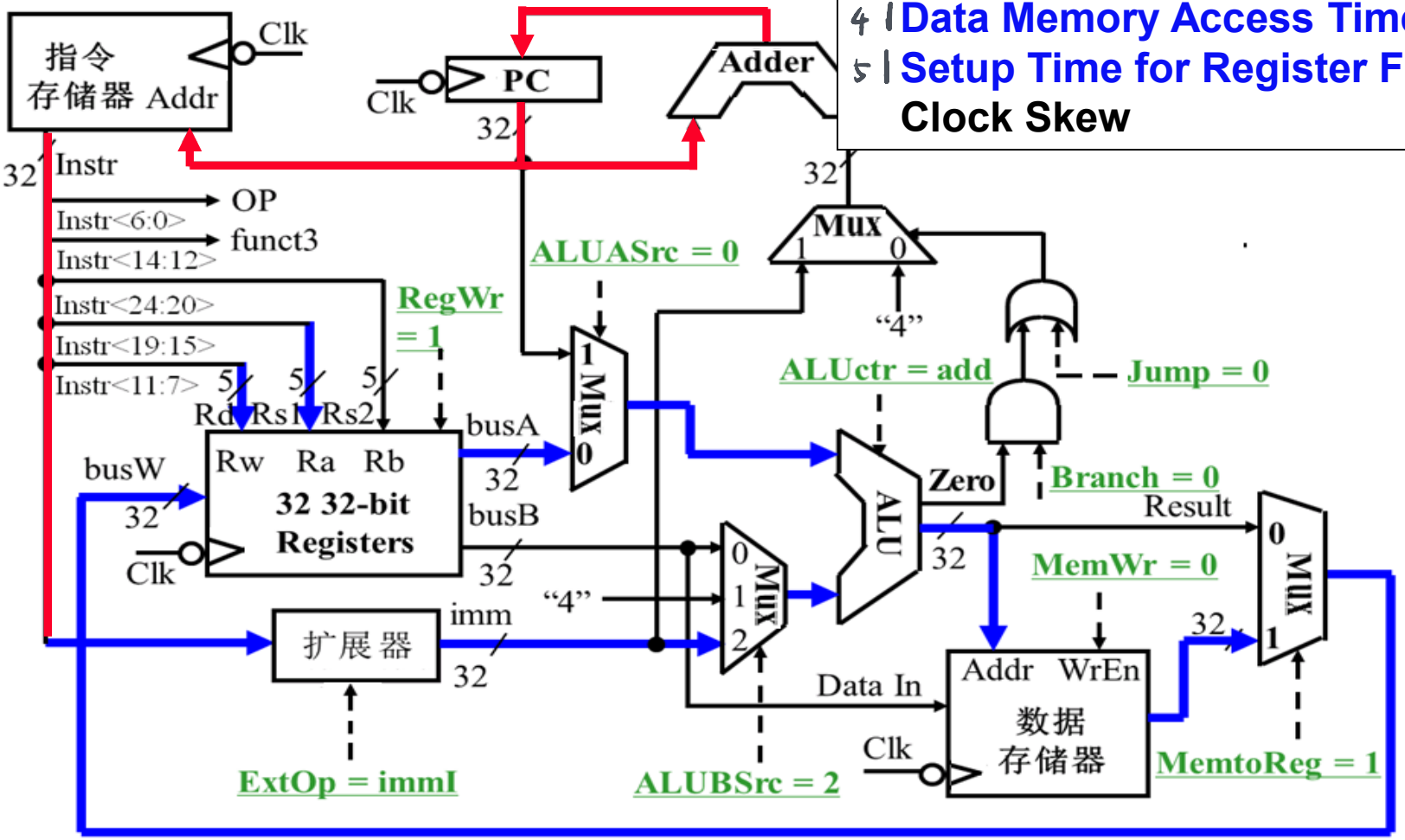
假定每一步时间均衡，当N很大的时候，流水线比串行方式提高约3倍!

流水方式下，所用时间主要与最长阶段的时间有关!

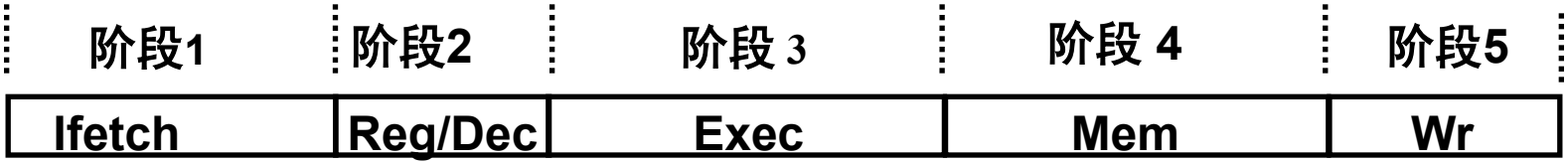
回顾：单周期数据通路中的关键路径

Load操作：
R[rd] ← M[R[rs1]+imm]

Critical Path (Load Operation) =
1. PC's prop time (Clk-to-Q) +
2. Instruction Memory's Access Time +
2. Register File's Access Time +
3. ALU to Perform a 32-bit Add +
4. Data Memory Access Time +
5. Setup Time for Register File Write +
   Clock Skew



分析：Load指令的5个阶段



- 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 实际上为一个部件

这里寄存器堆的读口和写口可看成两个不同的部件。

不考虑投机，因为还有别的指令需要等到译码之后才能进行ALU运算

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？

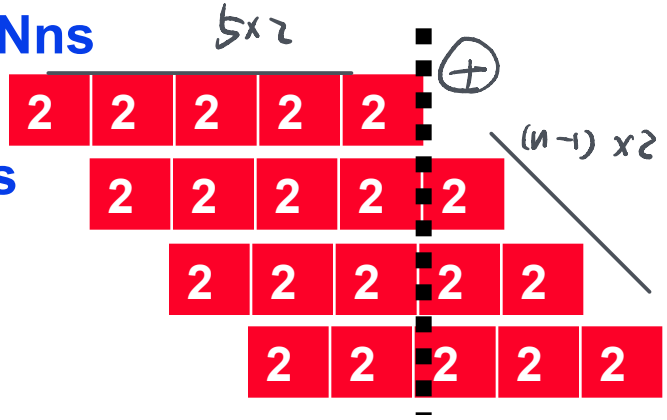
单周期模型与流水线模型的性能比较

- 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

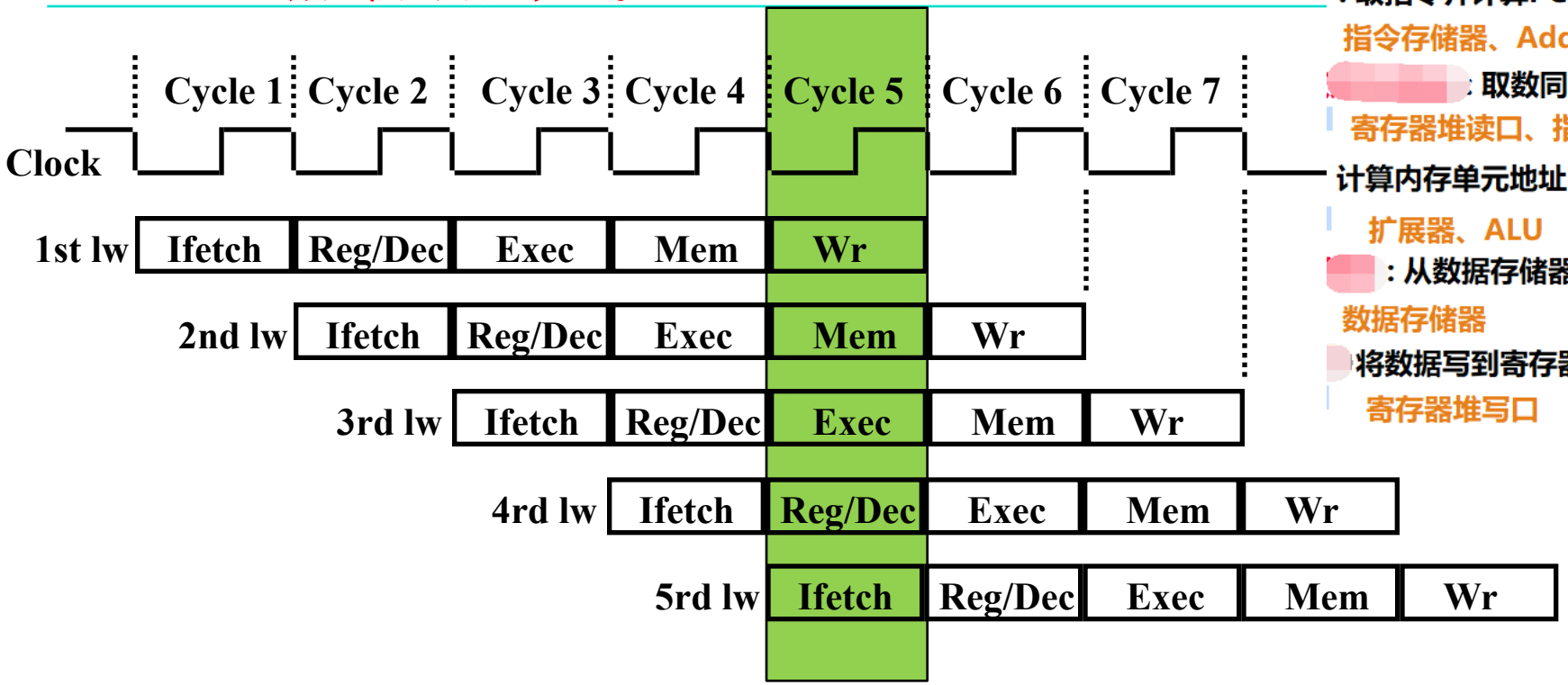
  (假定控制单元、PC访问、信号传递等没有延迟)

- 单周期模型
  - 时钟周期等于最长的lw指令的执行时间，即：8ns
  - 每条指令的执行时间都是一个时钟周期：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： (5+(N-1))x2ns
  - 在N很大时，几乎是串行方式的 4 倍
  - 若各阶段操作均衡(例如，各阶段都是2ns)，则会是串行的5倍。



流水线方式下，单条指令执行时间可能延长，但能大大提高指令吞吐率！

Load指令的流水线



：取指令并计算PC+4
指令存储器、Adder
：取数同时译码
寄存器堆读口、指令译码器
计算内存单元地址
扩展器、ALU
：从数据存储器中读
数据存储器
将数据写到寄存器中
寄存器堆写口

- 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：
  - 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

多周期中第二周期投机只对lw/sw有意义。流水线中可取消投机而不影响CPI

* 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
    - RV32I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
    - RV32I指令的rs1和rs2编码位置固定，在指令译码时就可读其值

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	opcode
I	imm[11:0]			rs1	funct3	rd	opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
B	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode

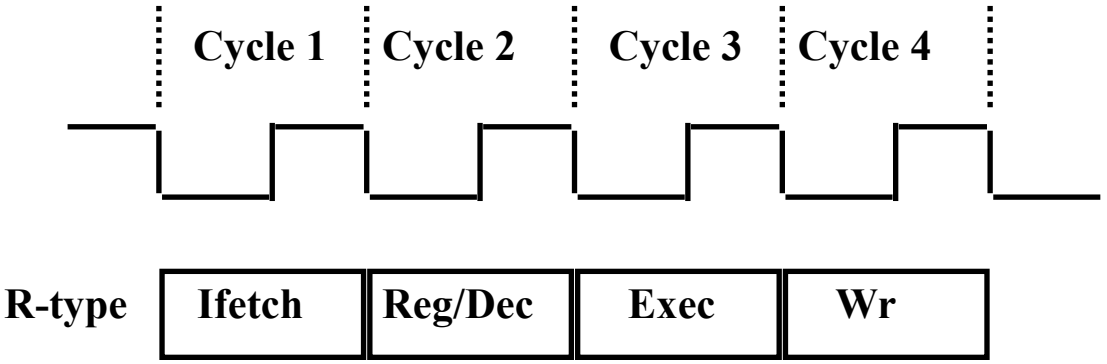
若位置随指令不同而不同，则需先确定指令类型才能取寄存器编号

  - load / Store指令才能访存，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运算类指令的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

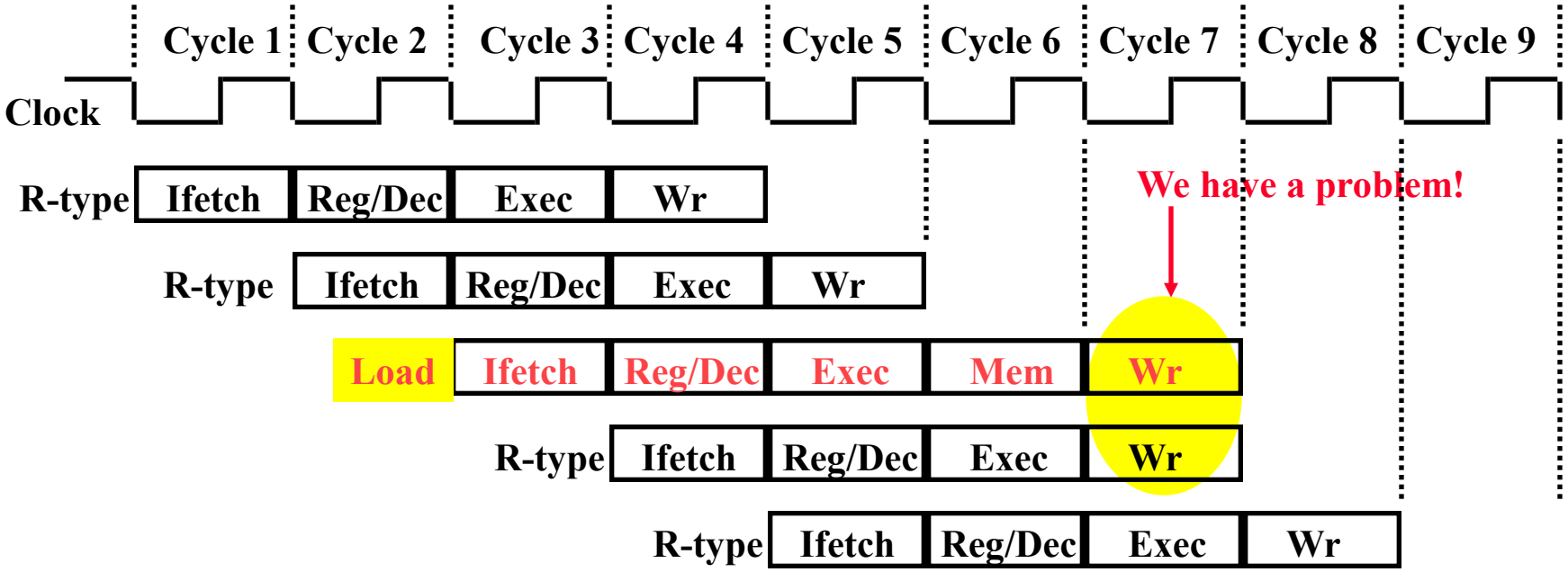
R-type指令的4个阶段

例：add rd, rs1, rs2



- **Ifetch:** 取指令并计算PC+4（写入PC）
- **Reg/Dec:** 从寄存器（rs1和rs2）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（rd）

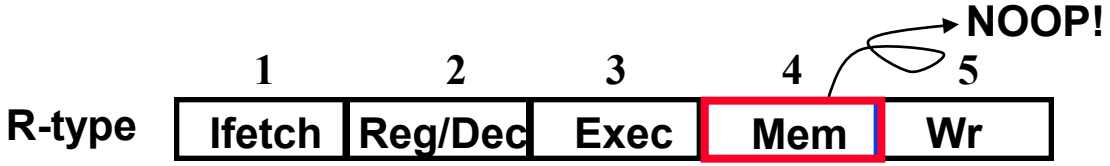
含R-type和 Load 指令的流水线



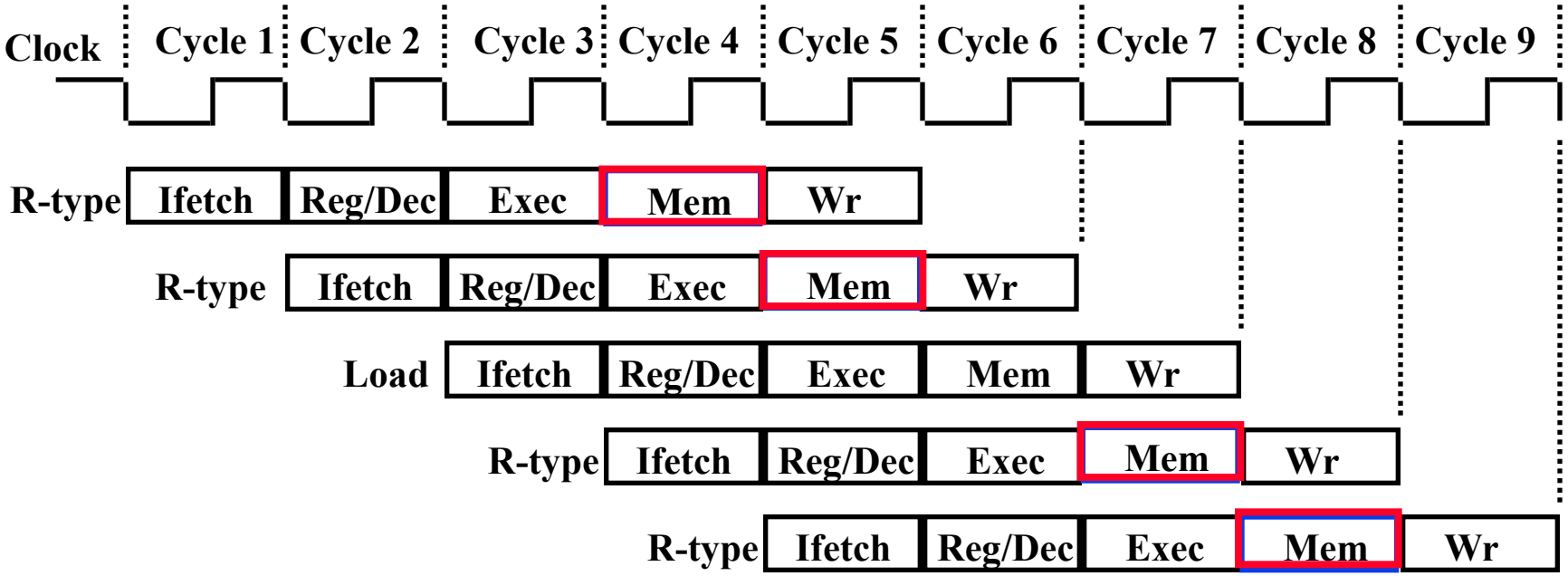
- 上述流水线有个问题: 两条指令试图同时写寄存器，因为
  - Load在第5阶段用寄存器写口
  - R-type在第4 阶段用寄存器写口

  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(Struture 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



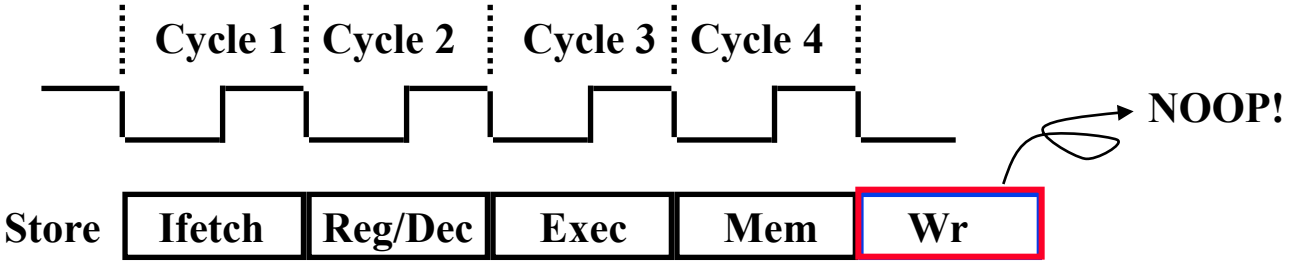
- 加一个NOP阶段以延迟“写”操作:
  - 把“写”操作安排在第5阶段, 这样使R-Type的Mem阶段为空NOP



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

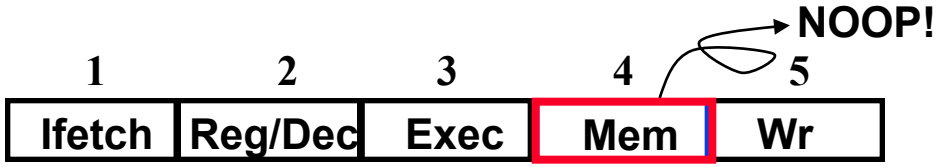
例：sw rs2, rs1(imm12)



- **Ifetch**：取指令并计算PC+4 （写入PC）
- **Reg/Dec**： 从寄存器（rs1）取数，同时指令在译码器进行译码
- **Exec**：12位立即数（imm12）符号扩展后与寄存器值（ rs1 ）相加，计算主存地址
- **Mem**：将寄存器（rs2）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！

I-type的运算类型指令

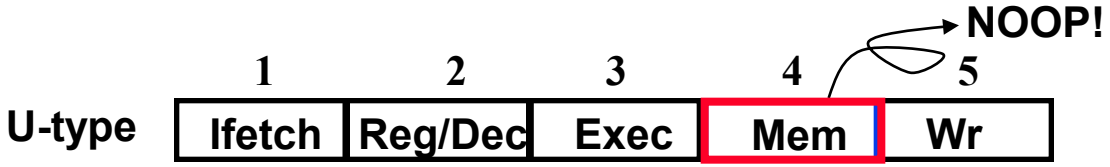
例：ori rd, rs1,imm12



- **Ifetch：取指令并计算PC+4 （写入PC）**
- **Reg/Dec： 从寄存器（rs1）取数，同时指令在译码器进行译码**
- **Exec：使用ALU完成12位立即数（imm12）符号扩展后与寄存器值（ rs1 ）的运算（or）**
- **Mem：空阶段**
- **Wr: ALU计算的结果写到寄存器（rd）**

U-type的运算类型指令

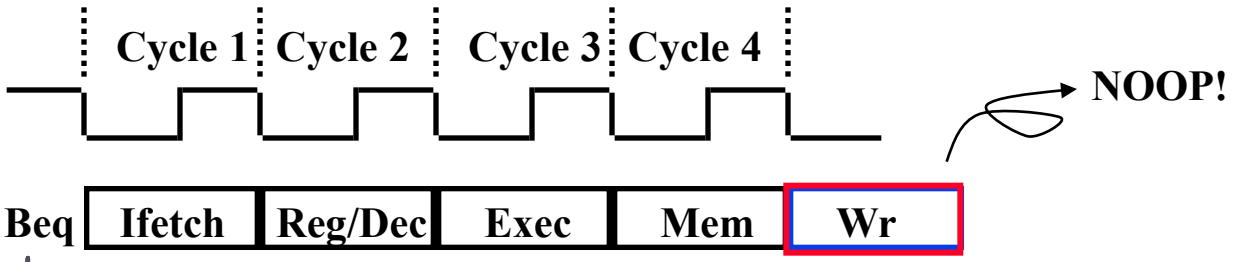
例：lui rd, imm20



- **Ifetch：取指令并计算PC+4 （写入PC）**
- **Reg/Dec： 指令在译码器进行译码**
- **Exec：将20位立即数（imm20）末尾补0后形成32位数据直接送到ALU输出端**
- **Mem：空阶段**
- **Wr: ALU输出端的结果写到寄存器（rd）**

Beq的四个阶段（可能的实现方式）

例：beq rs1, rs2, imm12



使用了Adder

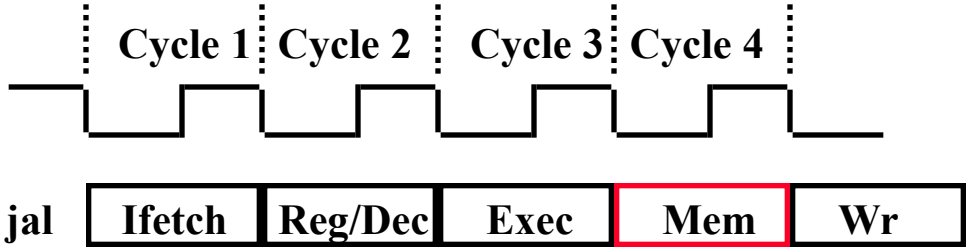
- **Ifetch:** 取指令并计算PC+4 （写入PC，但后续可能需要修改PC）
- **Reg/Dec:**从寄存器（rs1，rs2）取数，同时指令在译码器进行译码
- **Exec:** 执行阶段
  - ALU中比较两个寄存器（rs1，rs2）的大小（做减法）
  - Adder中计算转移地址（PC+SEXT（imm12）<<1）
- **Mem:** 如果比较相等, 则：
  - 转移目标地址写到PC
- **Wr:** 加一个空写阶段，使流水线更规整！

冲突

转移

J型的四个阶段

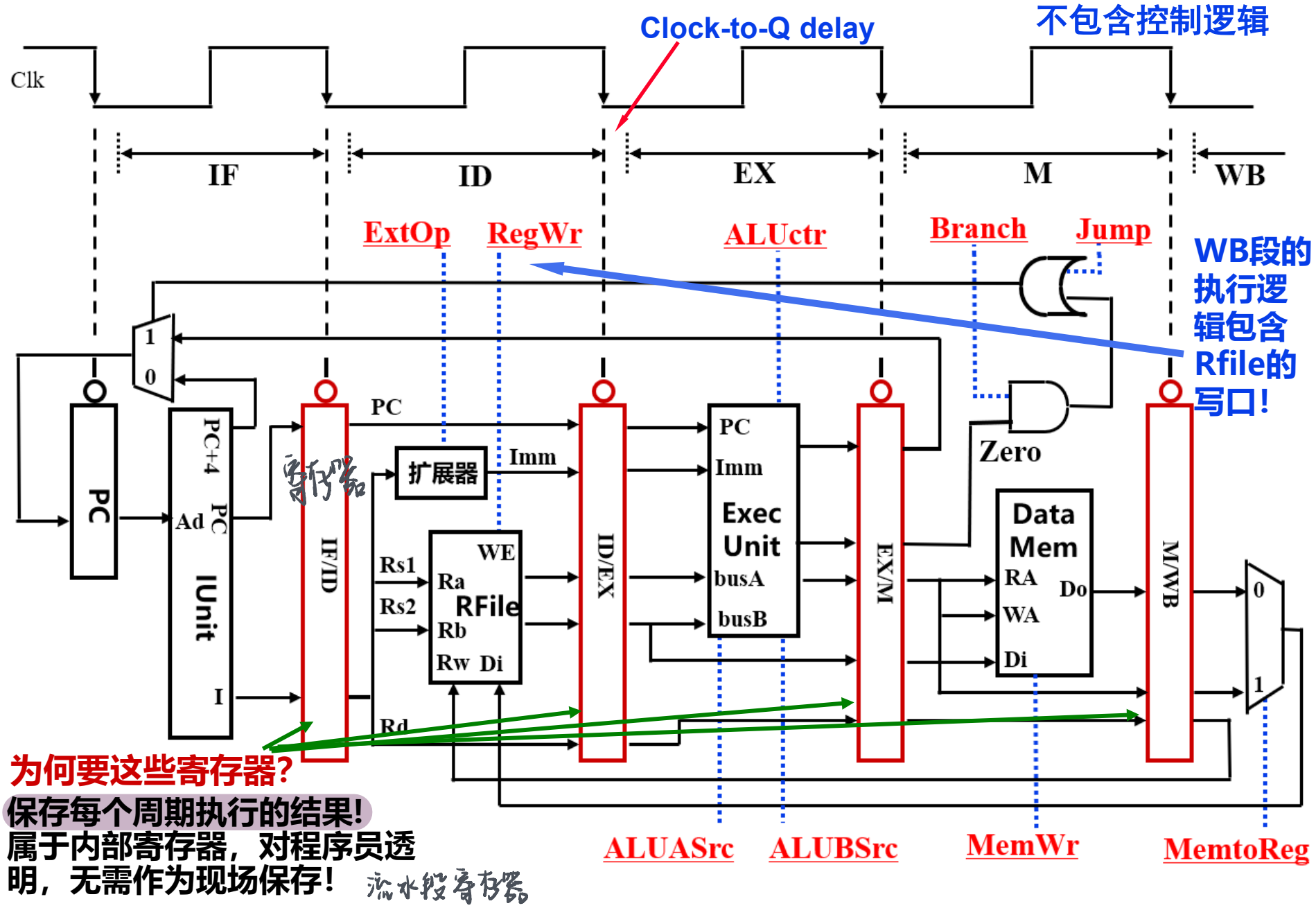
例：jal rd, imm20



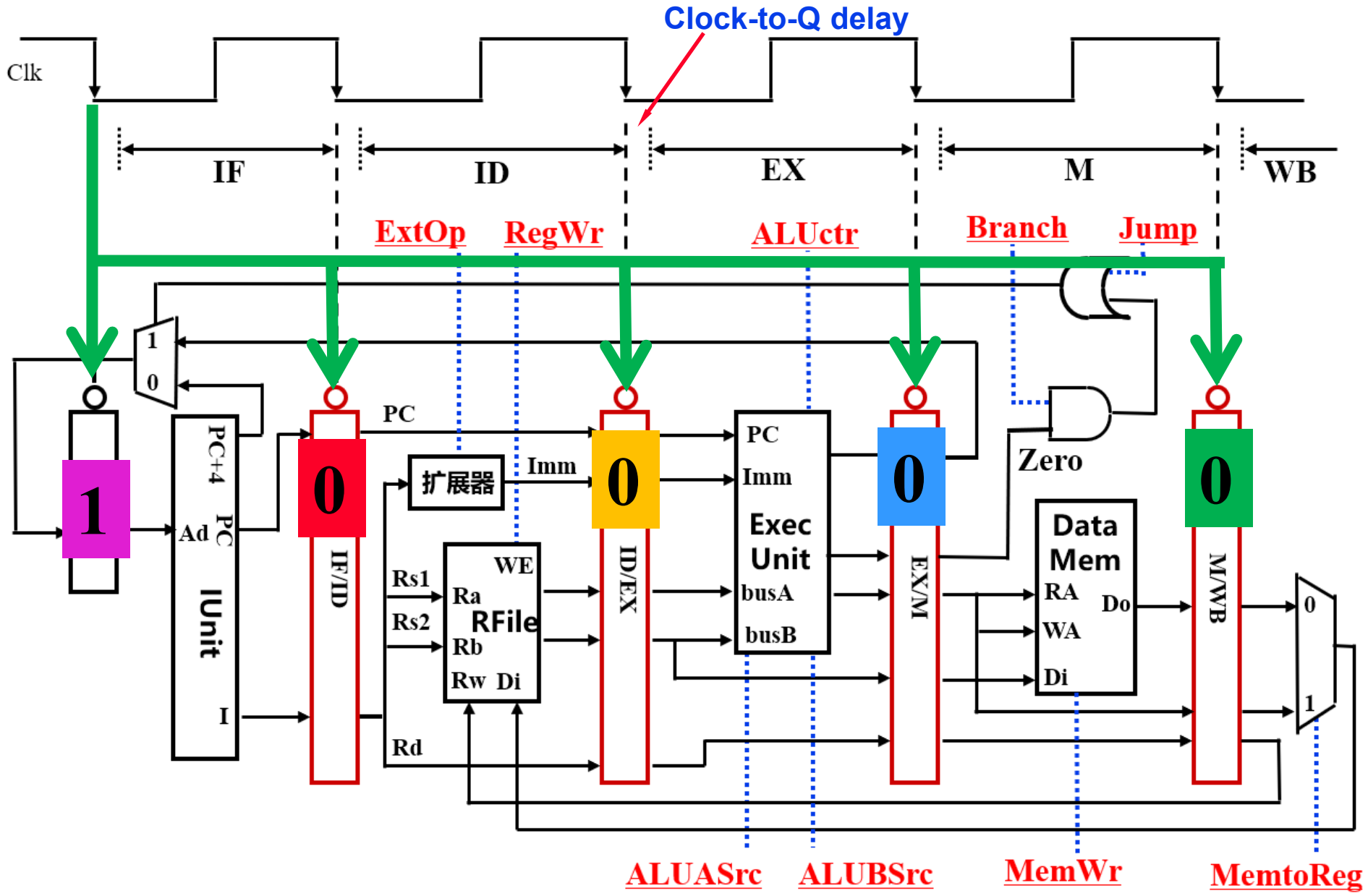
- **Ifetch:** 取指令并计算PC+4 （写入PC，但后续肯定需要修改PC）
- **Reg/Dec:**从寄存器取数，同时指令在译码器进行译码
- **Exec:** 执行阶段
  - ALU中计算PC+4（准备写入rd）
  - Adder中计算转移地址（PC+SEXT（imm20）<<1）
- **Mem:**把转移地址写入PC
- **Wr:** 把ALU运算结果（PC+4）写入rd,

至此，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

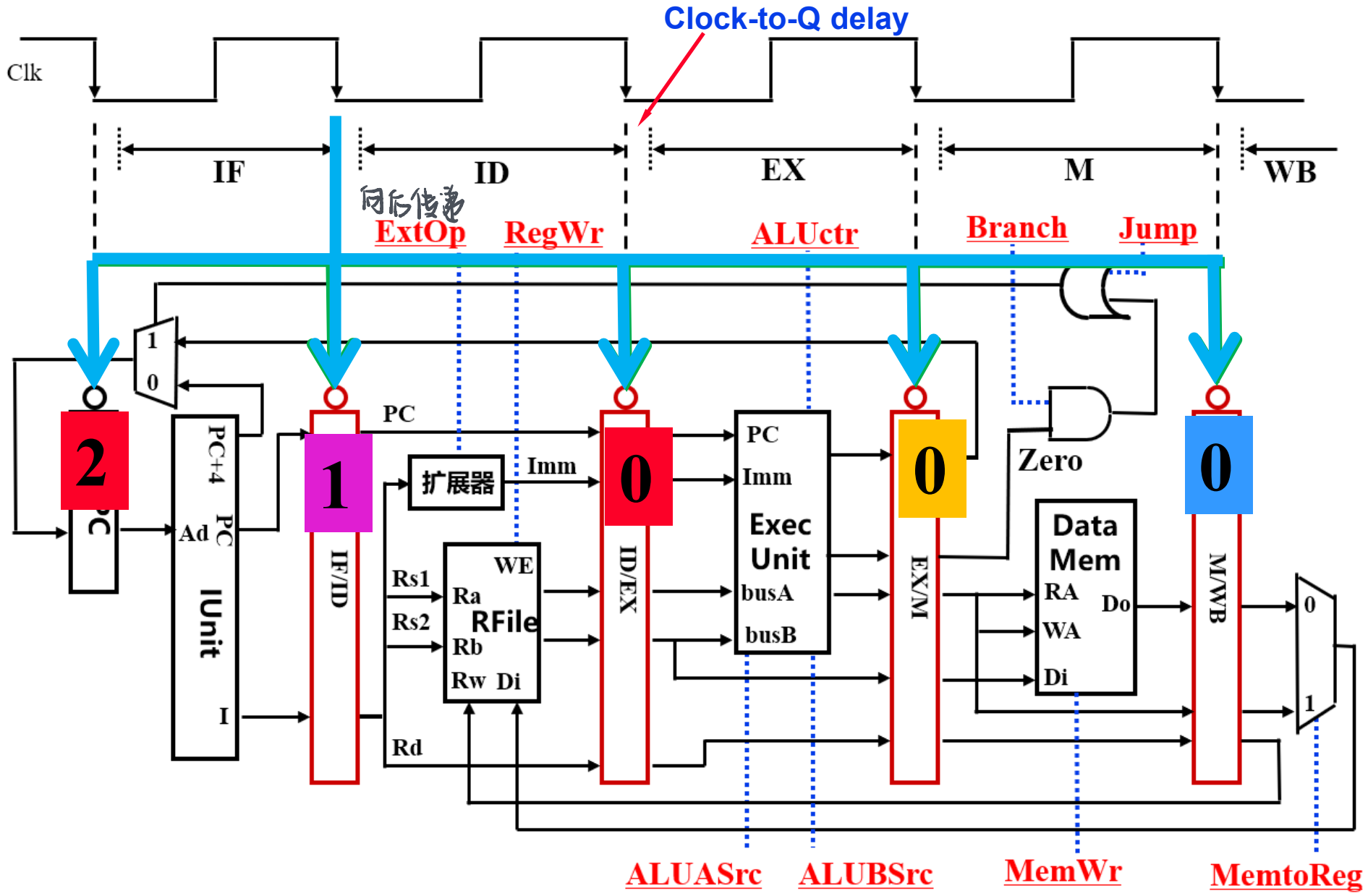
A Pipelined Datapath（五阶段流水线数据通路）



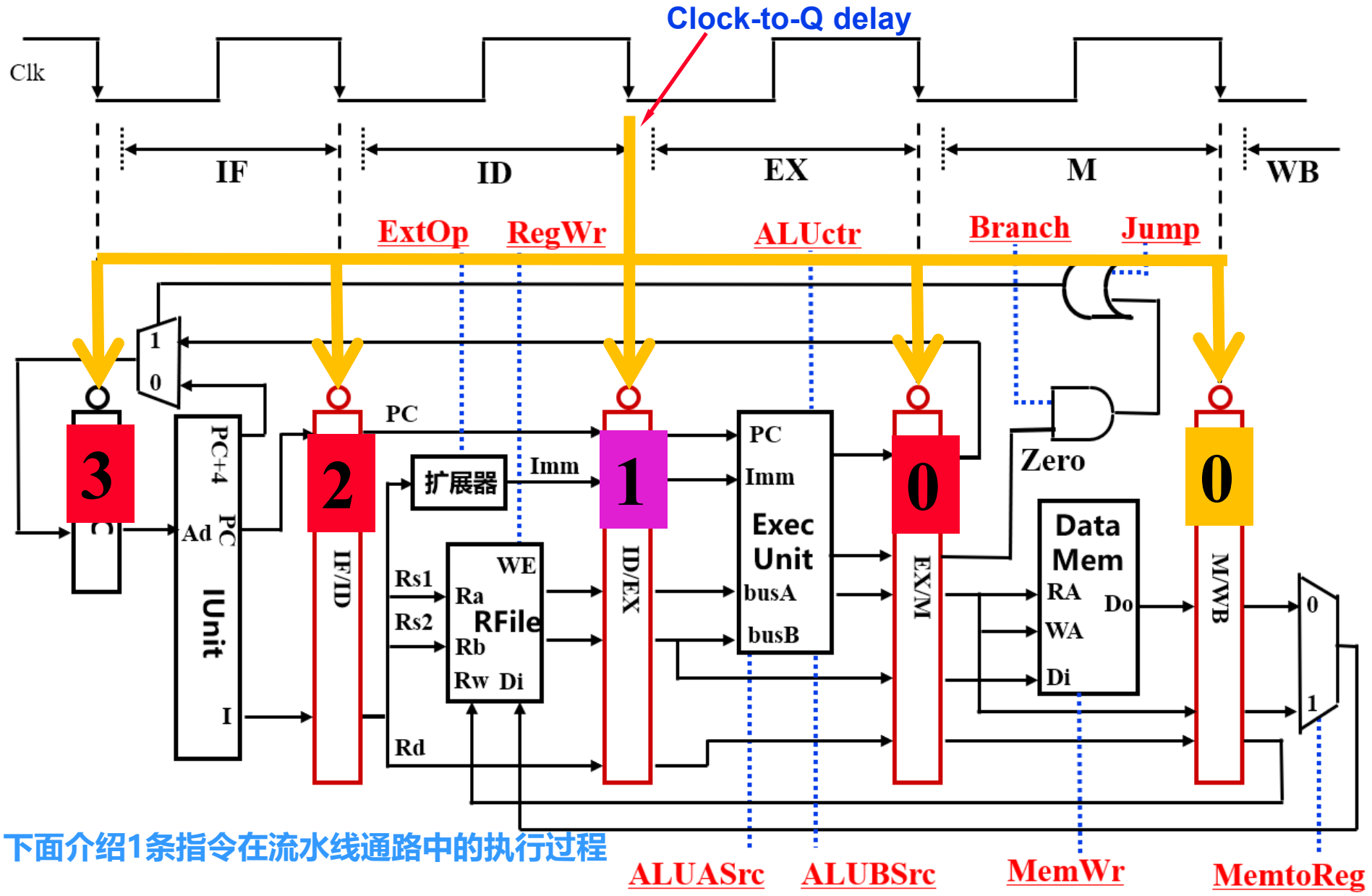
补充说明一下。。。。（请注意“1”）


Clock-to-Q delay
Clk
IF
ID
EX
M
WB
ExtOp
RegWr
ALUctr
Branch
Jump
1
0
PC
PC+4
Ad
IUnit
I
IF/ID
扩展器
Imm
Rs1
Rs2
Rd
WE
Ra
RFile
Rb
Rw
Di
ID/EX
PC
Imm
Exec Unit
busA
busB
EX/M
Zero
Data Mem
RA
WA
Di
Do
M/WB
ALUASrc
ALUBSrc
MemWr
MemtoReg

补充说明一下。。。。（请注意“1”）

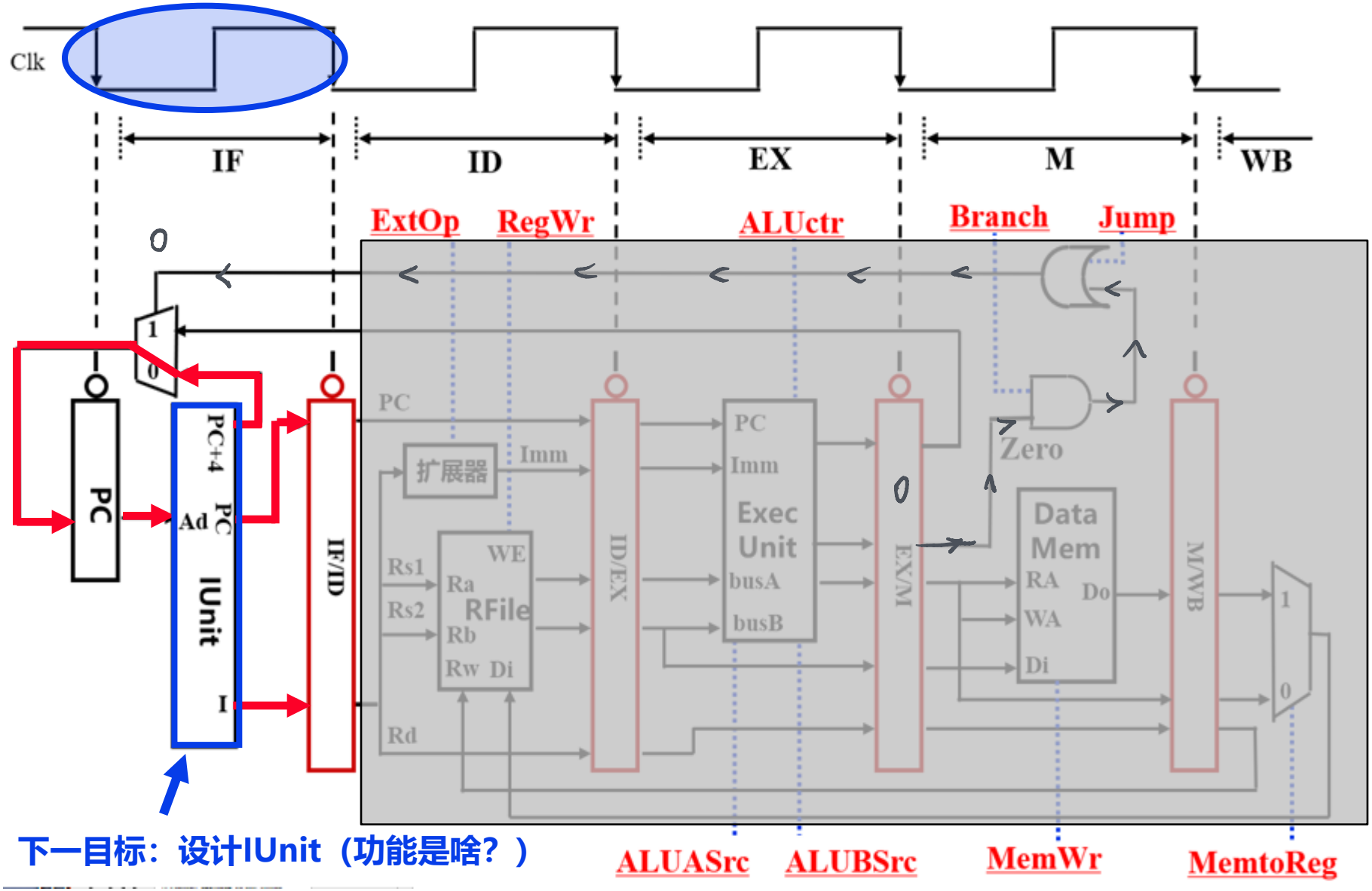

Clock-to-Q delay
Clk
IF
ID
EX
M
WB
同步传递
ExtOp
RegWr
ALUctr
Branch
Jump
2
1
0
PC
PC+4
Ad
IUnit
I
IF/ID
扩展器
Imm
Rs1
Rs2
Rd
WE
Ra
RFile
Rb
Rw
Di
ID/EX
PC
Imm
Exec
Unit
busA
busB
EX/M
Zero
Data
Mem
RA
WA
Di
Do
M/WB
ALUASrc
ALUBSrc
MemWr
MemtoReg

补充说明一下。。。。（请注意“1”）


Clock-to-Q delay
Clk
IF
ID
EX
M
WB
ExtOp
RegWr
ALUctr
Branch
Jump
1
0
3
2
1
0
0
PC
PC+4
Ad
PC
IUnit
I
IF/ID
扩展器
Imm
Rs1
Rs2
Rd
WE
Ra
RFile
Rb
Rw
Di
ID/EX
PC
Imm
Exec
Unit
busA
busB
EX/M
Zero
Data
Mem
RA
WA
Di
Do
M/WB
0
1
下面介绍1条指令在流水线通路中的执行过程
ALUASrc
ALUBSrc
MemWr
MemtoReg

取指令（IF）阶段

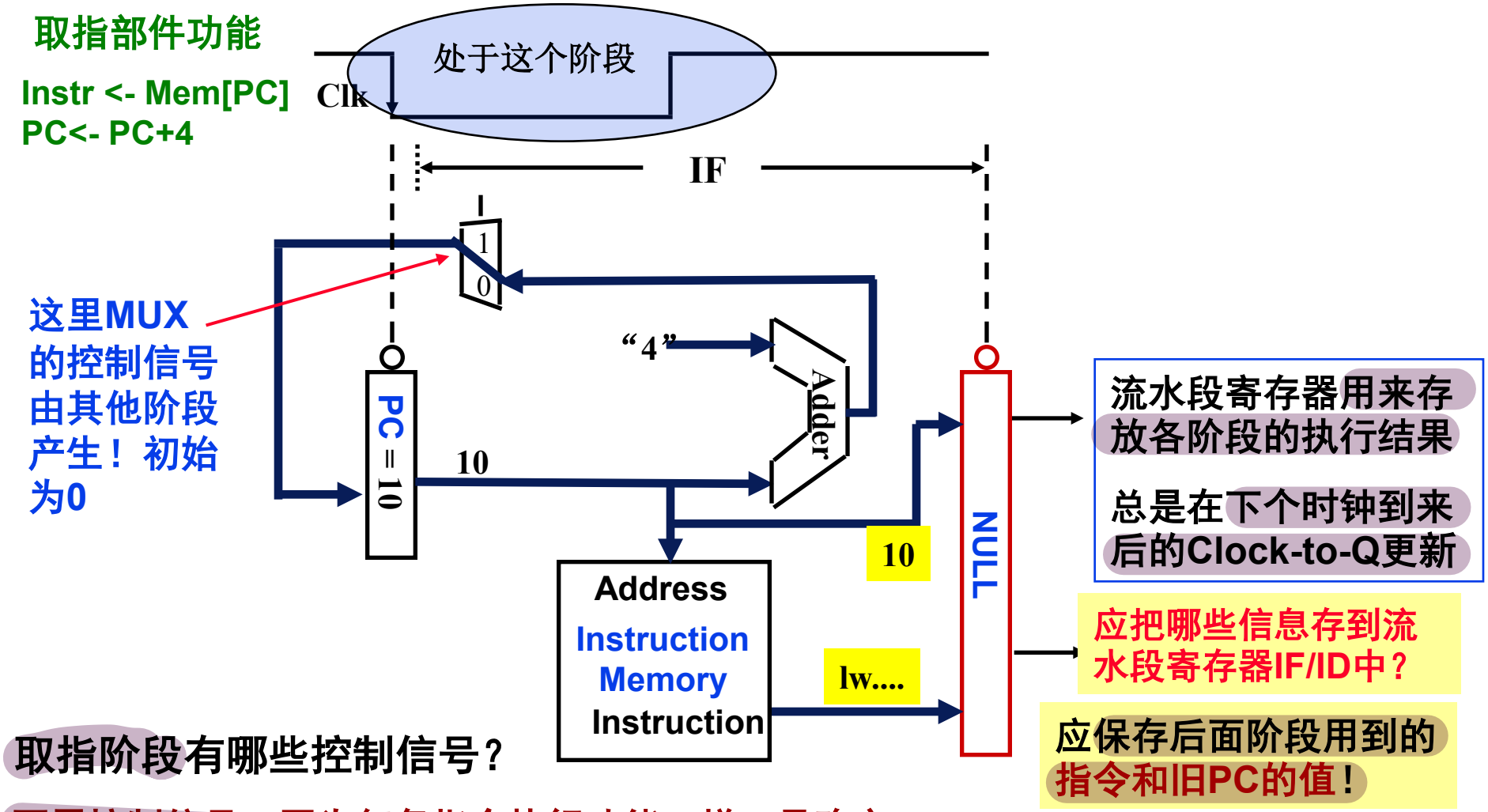
- 第10单元指令： lw x8, 0x100(x9)　　功能：R[x8] <- M[R[x9]+SEXT(0x100)]



下一目标：设计IUnit（功能是啥？）

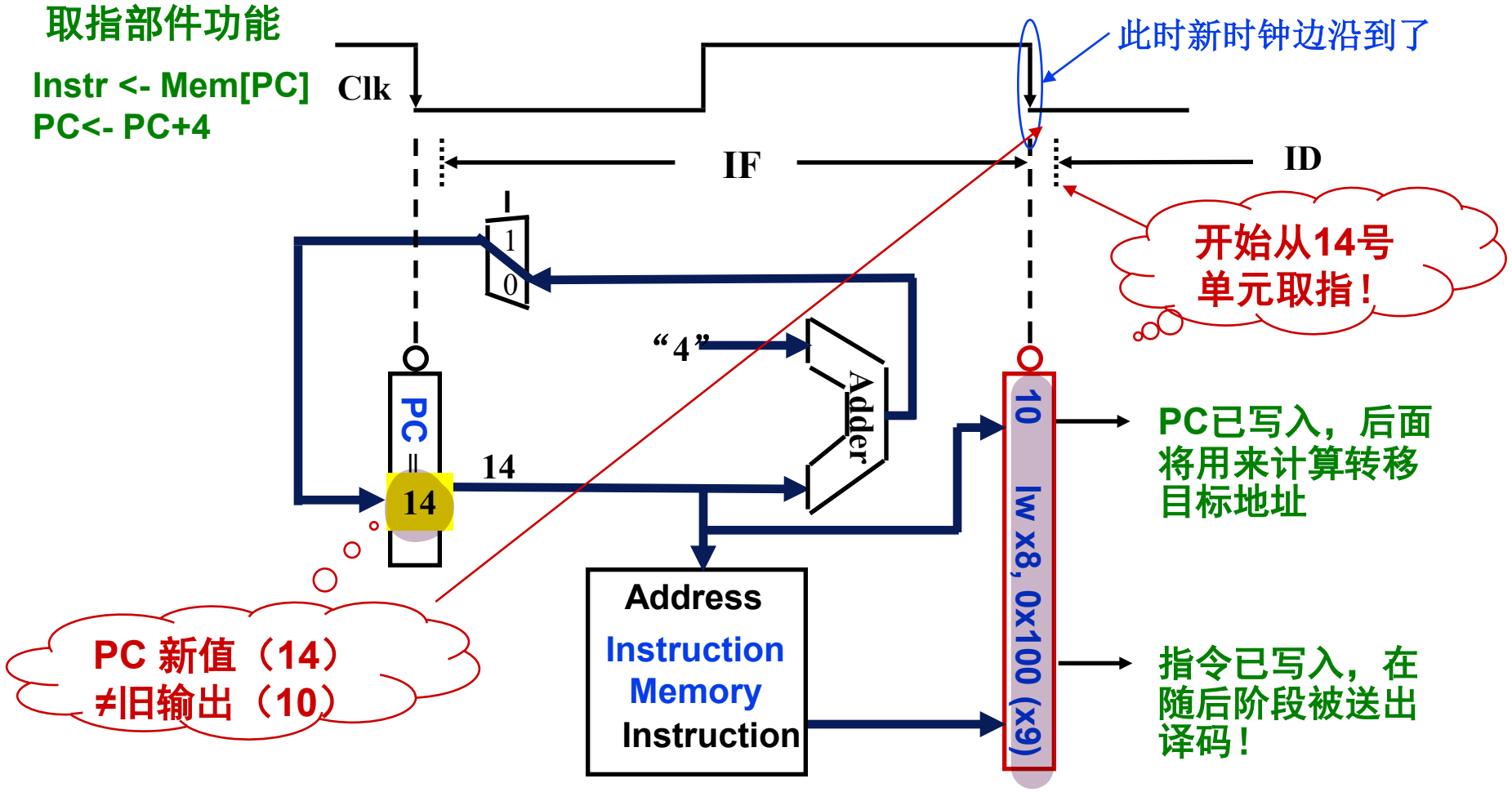
取指令部件 IUnit的设计——开始时和过程中

- 第10单元指令: : lw x8, 0x100(x9)



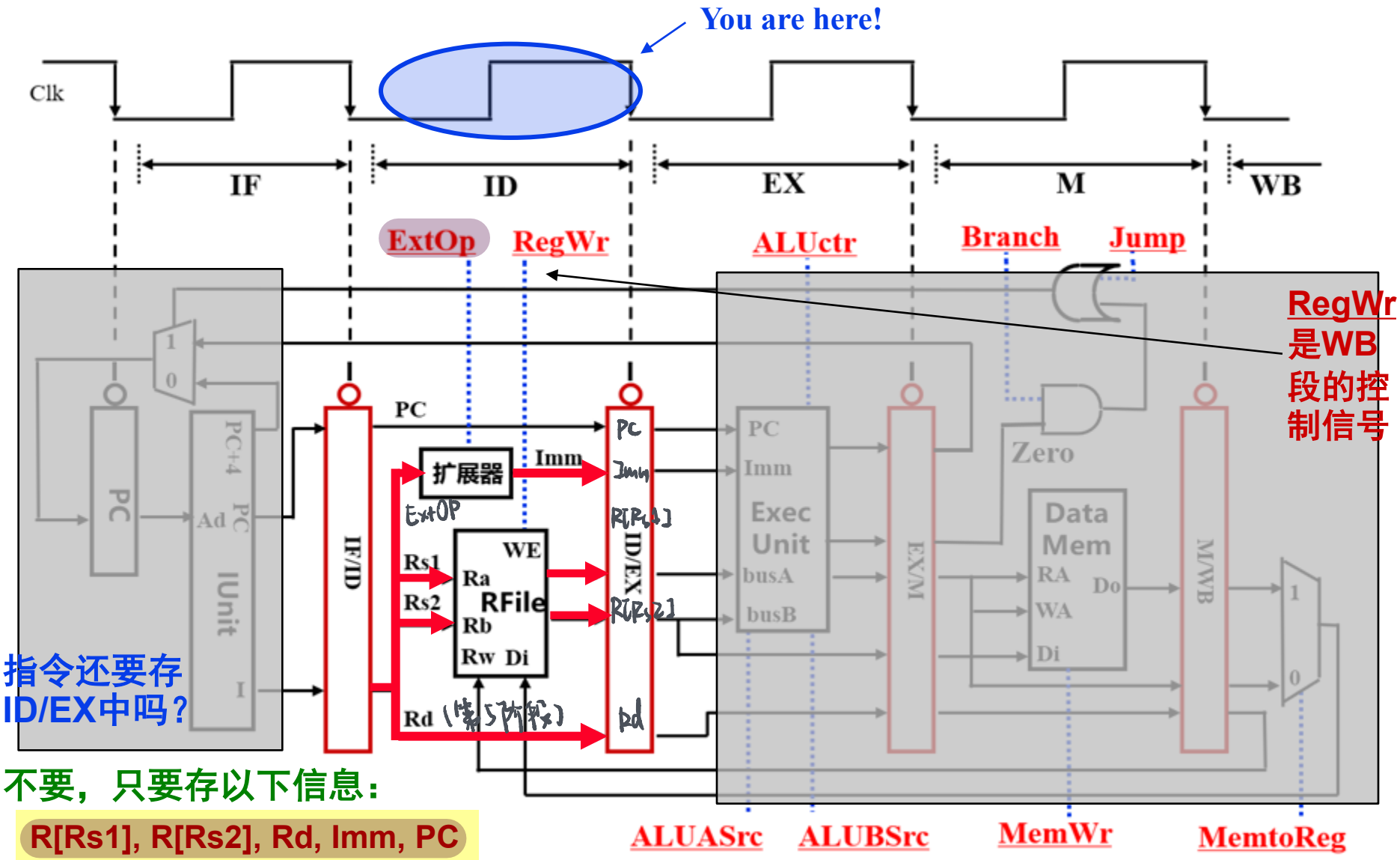
取指令部件 IUnit的设计——结束时

- 第10单元指令: : lw x8, 0x100(x9)　　随后的指令在14号单元中！



译码/取数（Reg/Dec）阶段

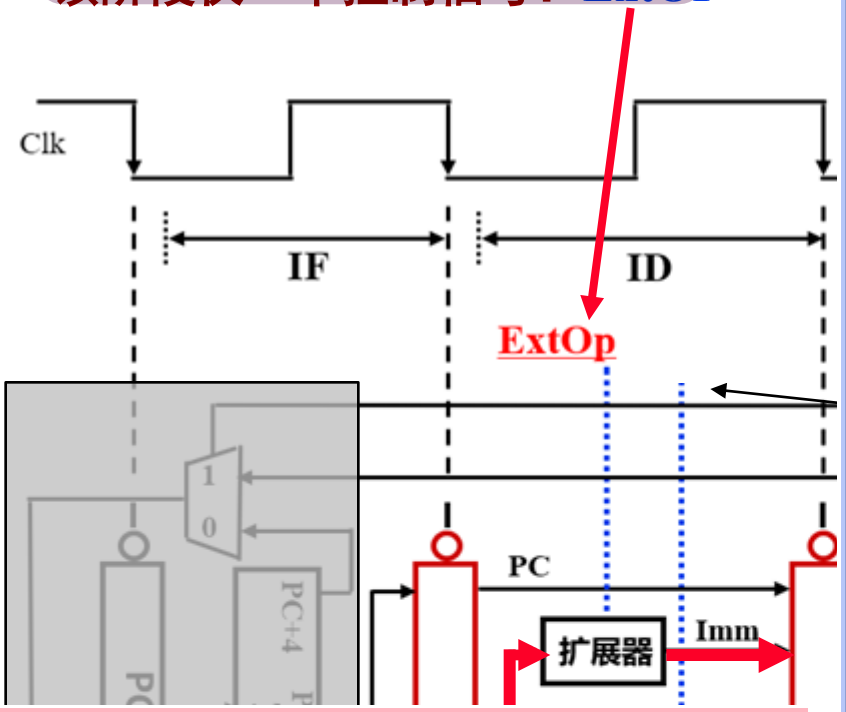
◦ 第10单元指令： lw x8, 0x100(x9)　　功能：R[x8] <- M[R[x9]+SEXT(0x100)]

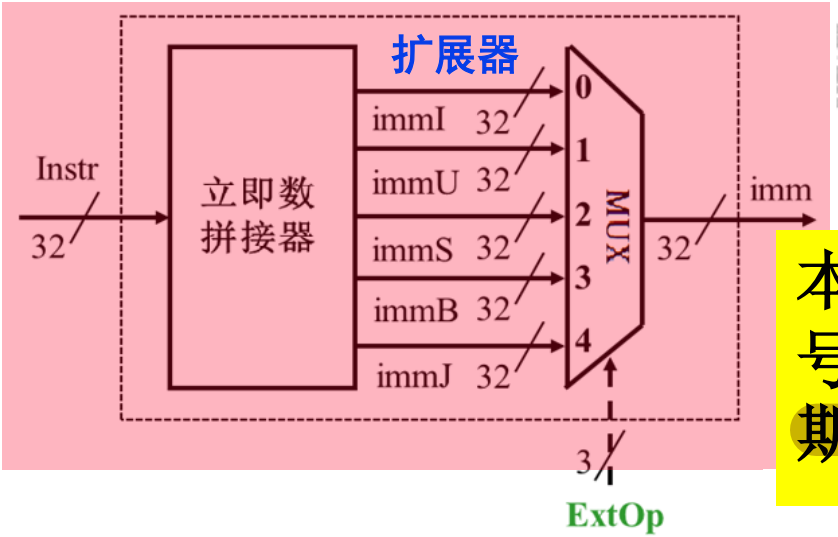


该阶段有哪些控制信号？　ExtOP用于控制扩展器对立即数进行扩展。 ExtOP=？？

译码/取数（Reg/Dec）阶

该阶段仅一个控制信号：ExtOP



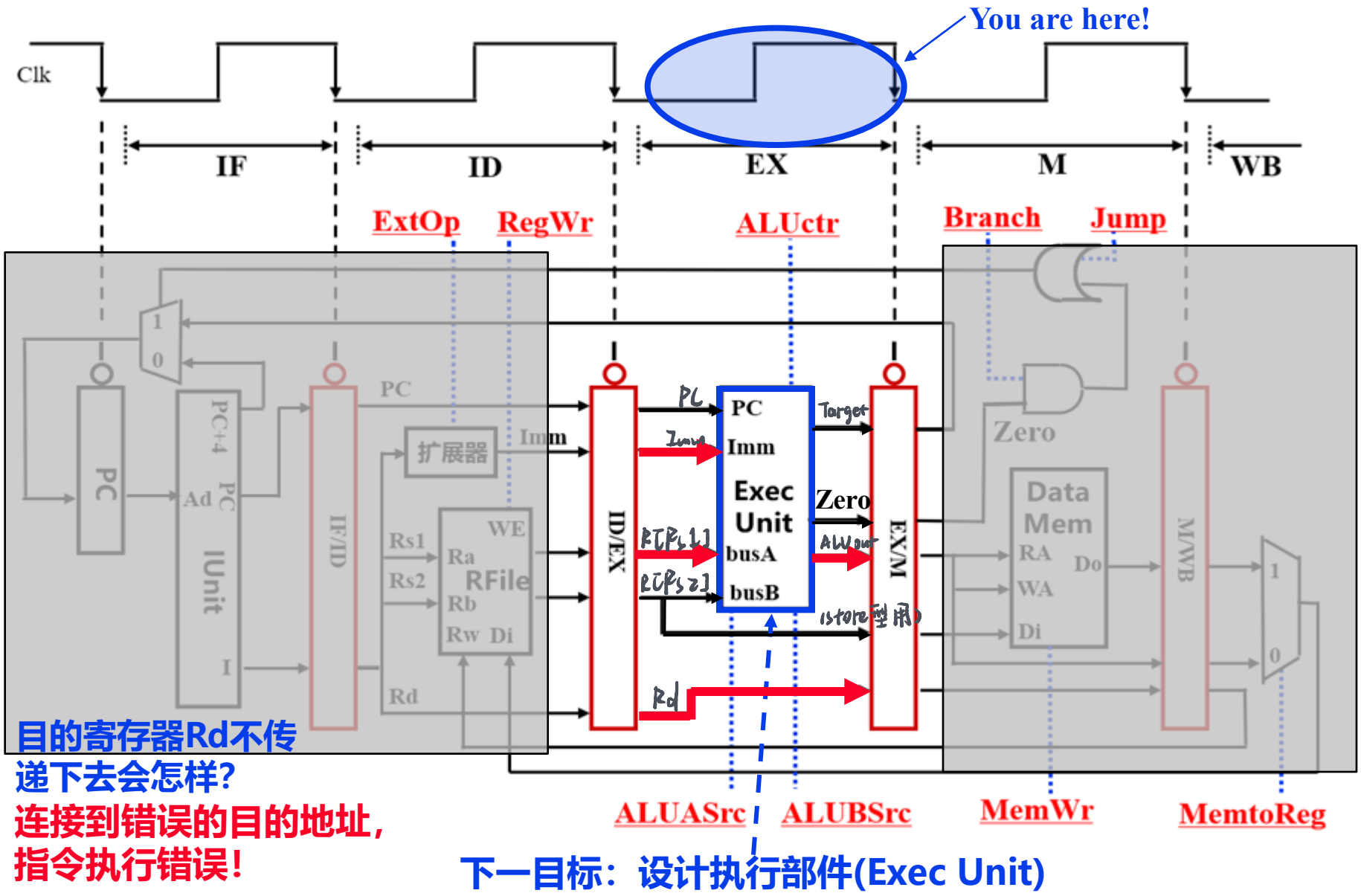


指　令	立即数编码类型	ExtOp<2:0>
add rd, rs1, rs2	无立即数	× × ×
slt rd, rs1, rs2		
sltu rd, rs1, rs2		
ori rd, rs1, imm12	I-型立即数（immI）	0 0 0
lui rd, imm20	U-型立即数（immU）	0 0 1
lw rd, rs1, imm12	I-型立即数（immI）	0 0 0
sw rs1, rs2, imm12	S-型立即数（immS）	0 1 0
beq rs1, rs2, imm12	B-型立即数（immB）	0 1 1
jal rd, imm20	J-型立即数（immJ）	1 0 0

本阶段译码结束后，产生了ExtOP信号，并可在本阶段立刻使用（时钟周期> 译码时延+扩展器时延+…）

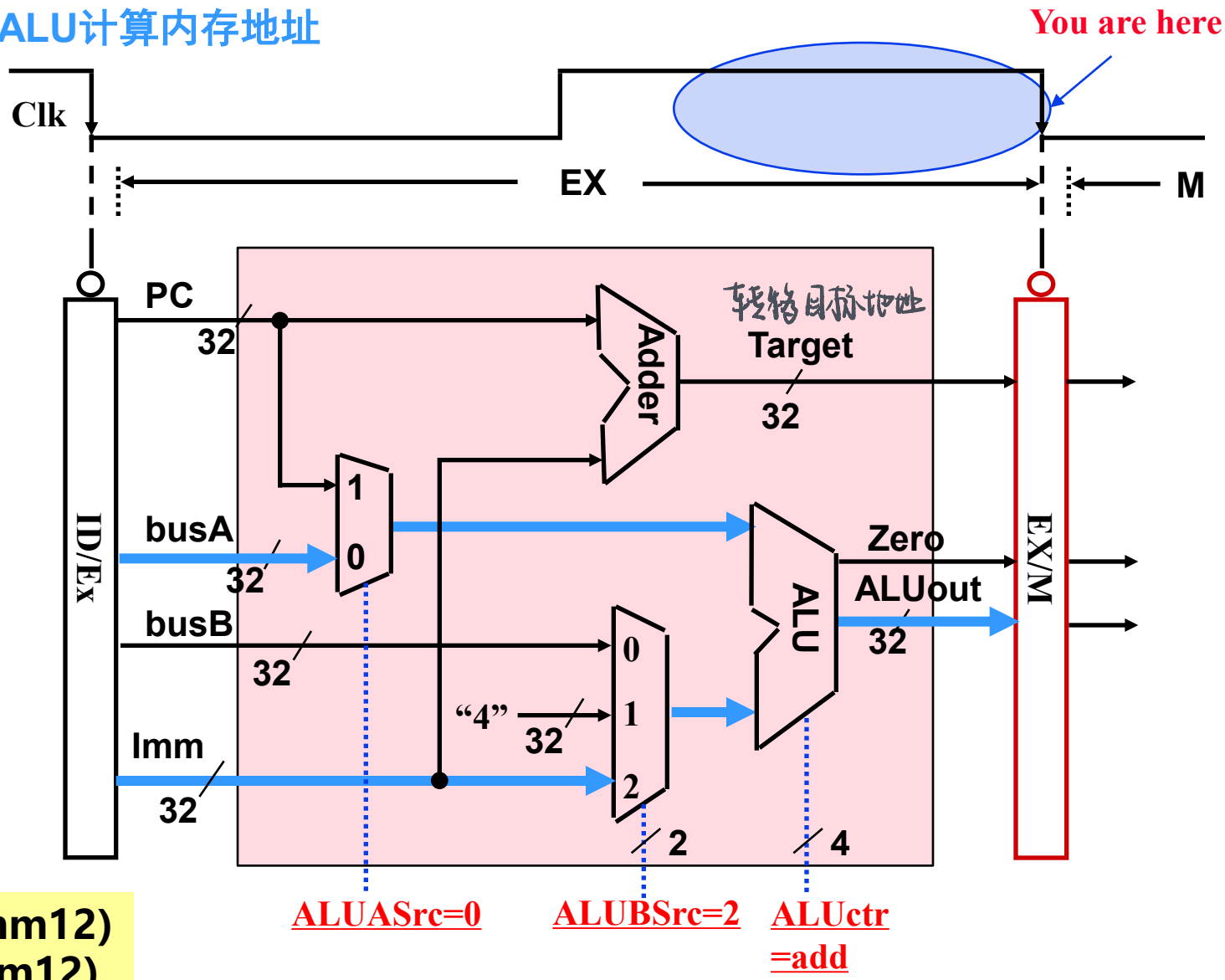
Load指令的地址计算阶段

- 第10单元指令： lw x8, 0x100(x9)　　功能：R[x8] <- M[R[x9]+SEXT(0x100)]



执行部件（Exec Unit）的设计：lw和sw

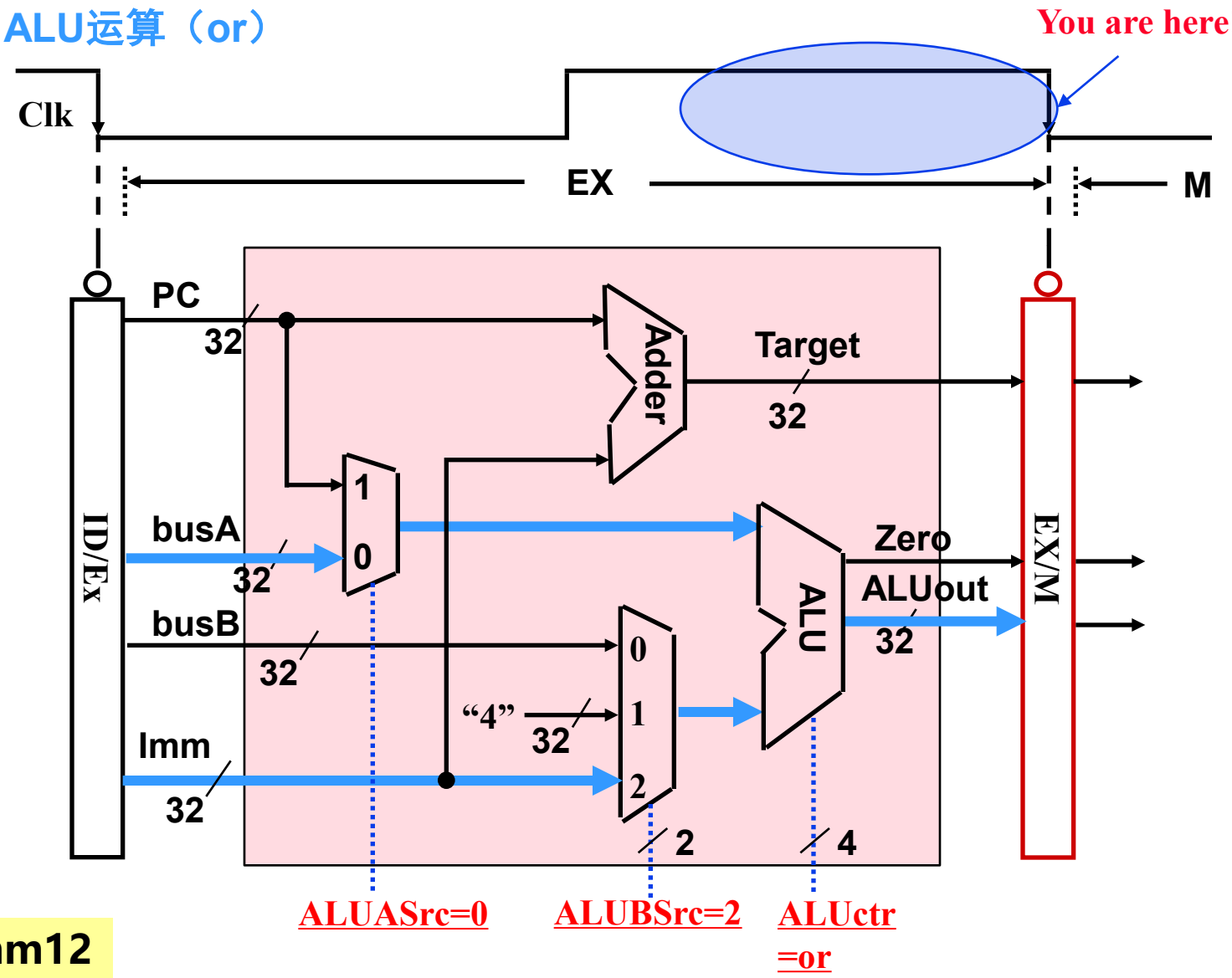
执行部件功能是：ALU计算内存地址



sw rs2, rs1(imm12)
lw rd, rs1(imm12)

执行部件（Exec Unit）的设计：ori指令（I型）

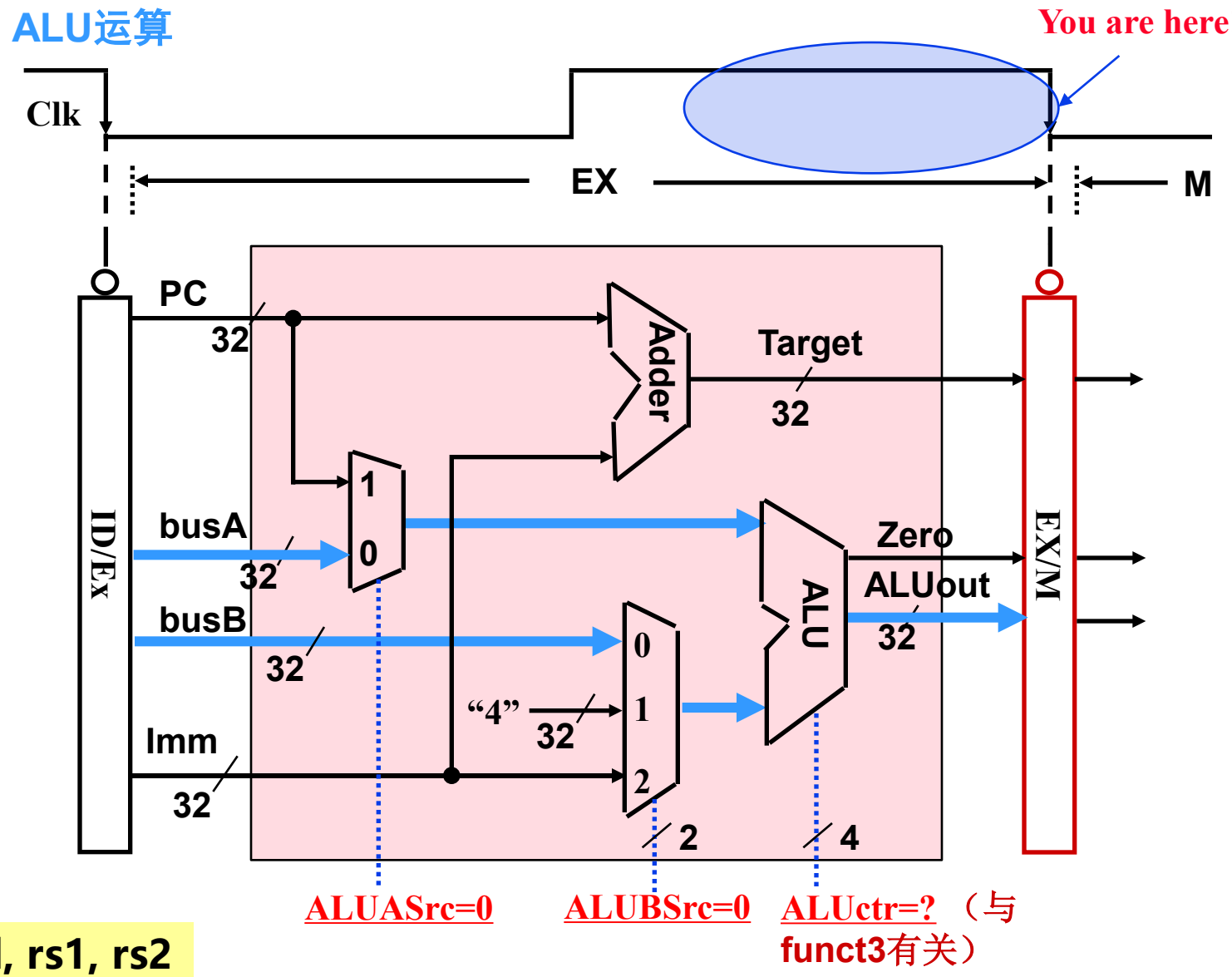
执行部件功能是：ALU运算（or）



ori rd, rs1,imm12

执行部件（Exec Unit）的设计：R型指令

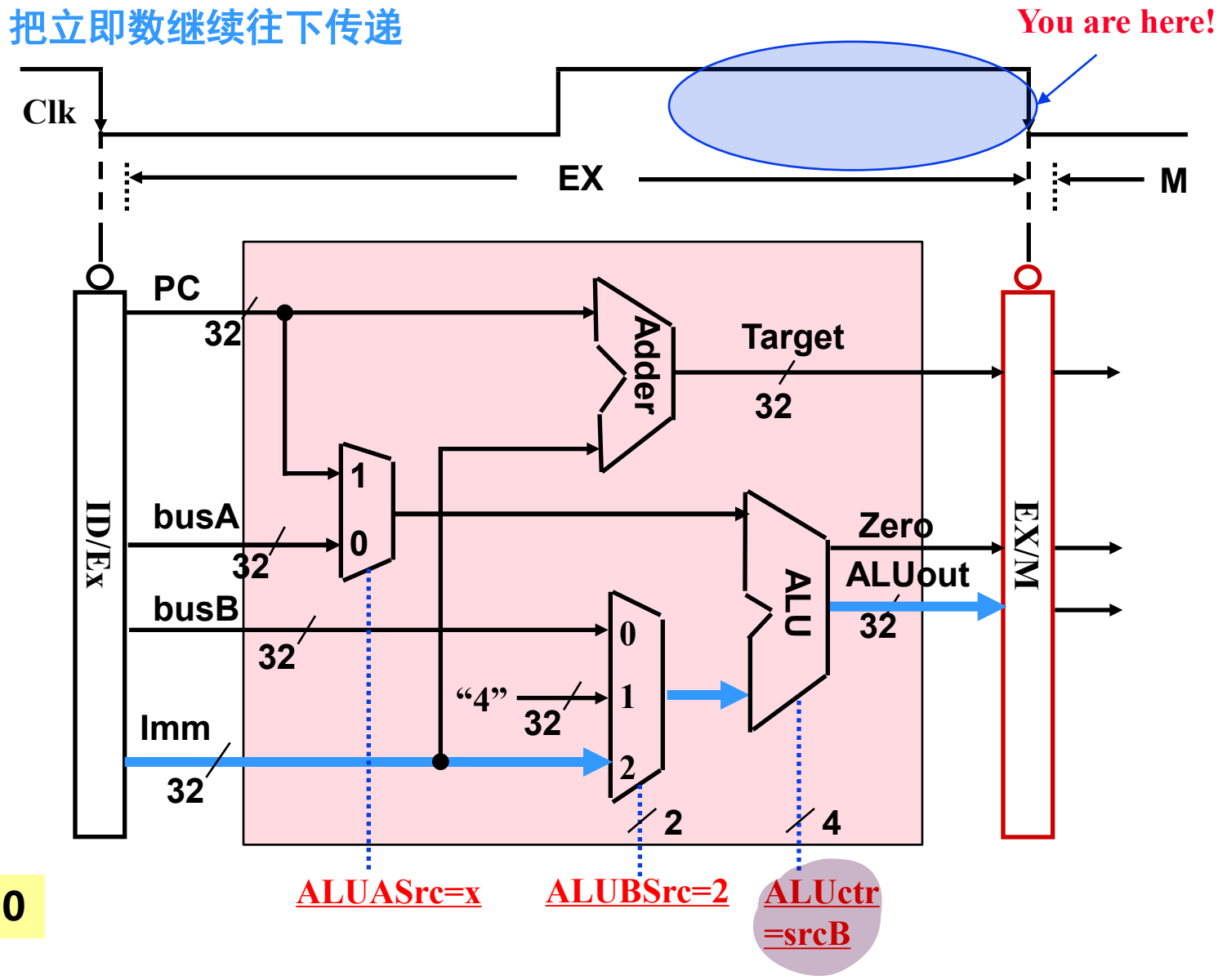
执行部件功能是：ALU运算



add(sub..) rd, rs1, rs2

执行部件（Exec Unit）的设计：U型指令

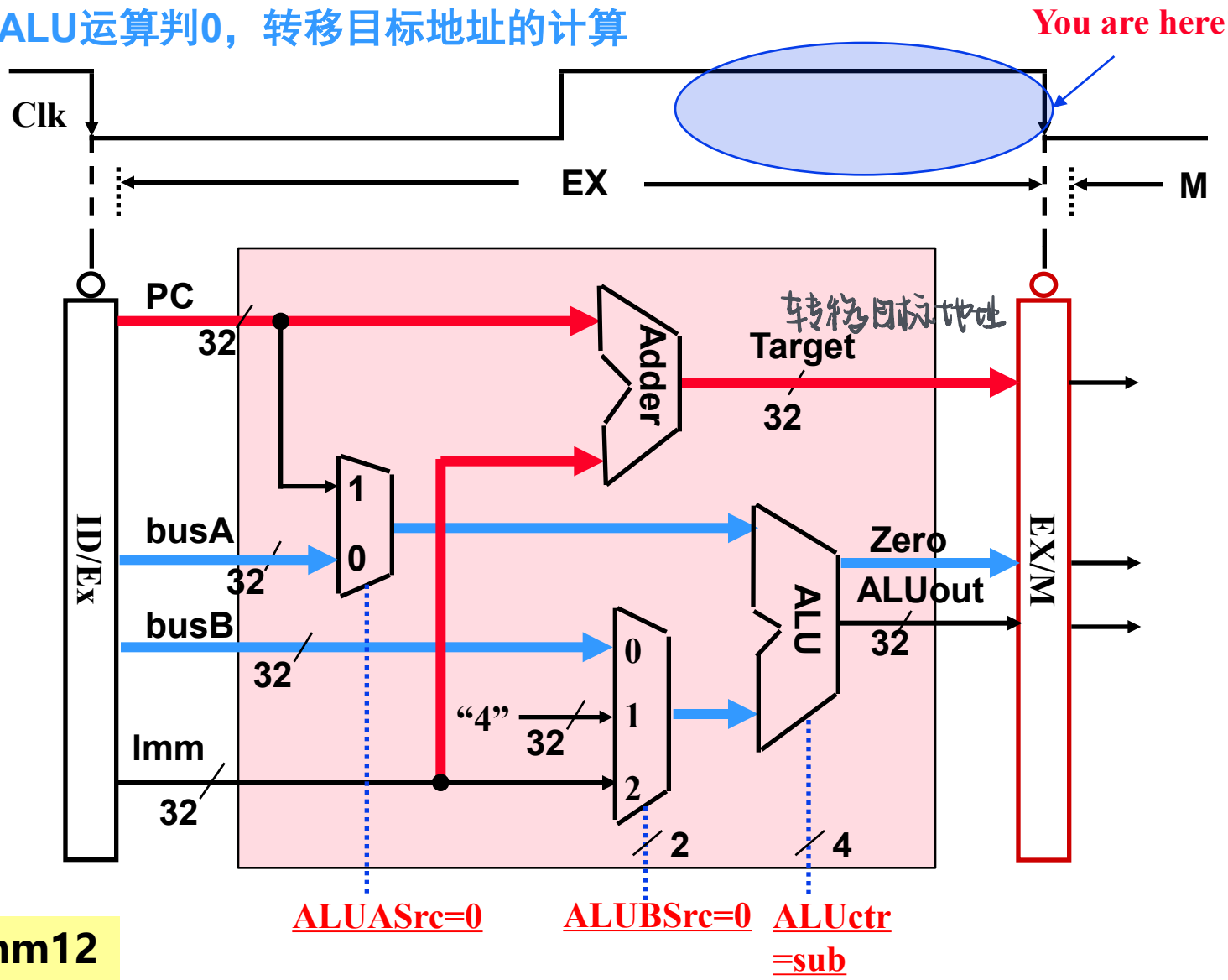
执行部件功能是：把立即数继续往下传递



lui rd, imm20

执行部件（Exec Unit）的设计：B型指令

执行部件功能是：ALU运算判0，转移目标地址的计算

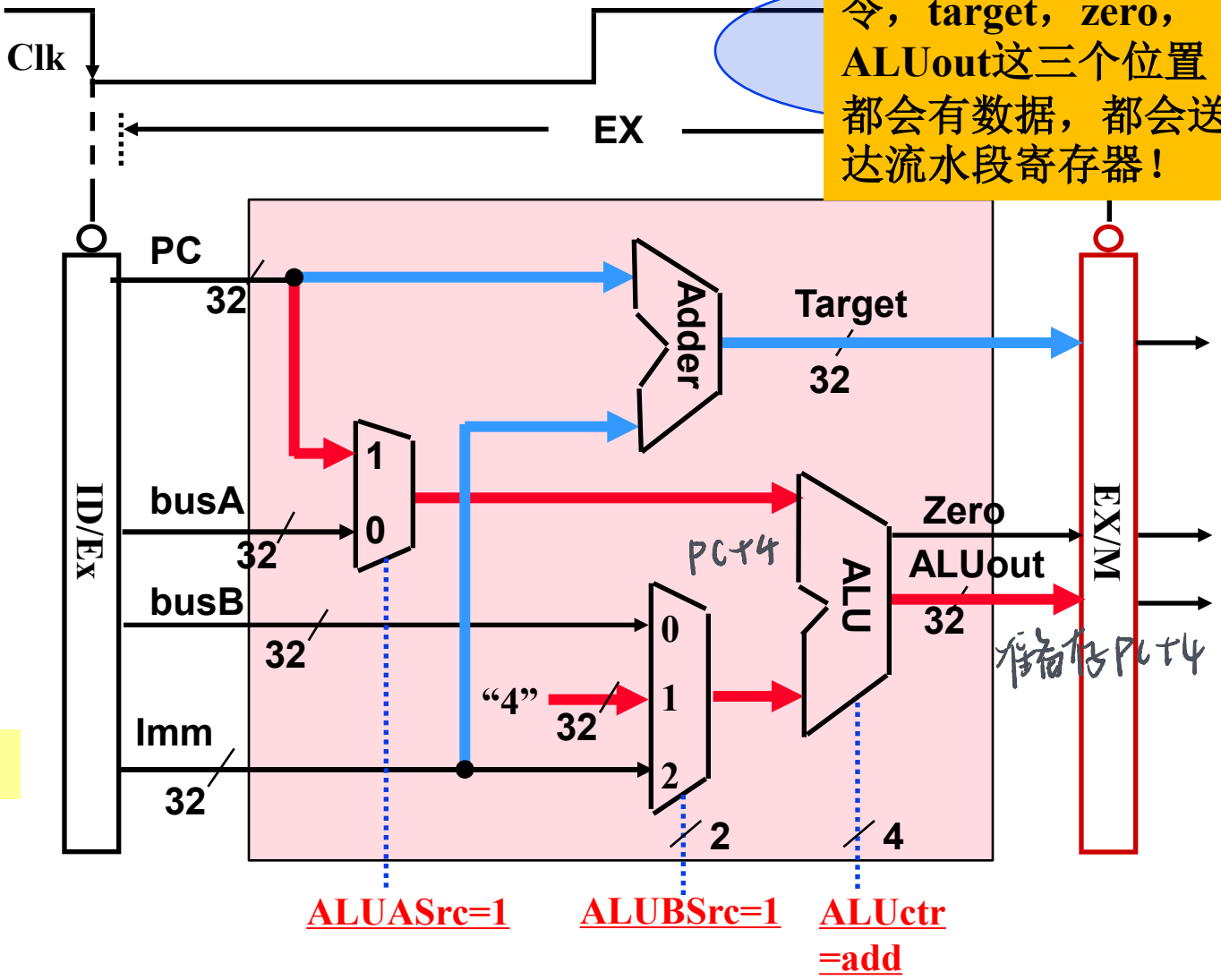


beq rs1, rs2, imm12

执行部件（Exec Unit）的设计：J型指令

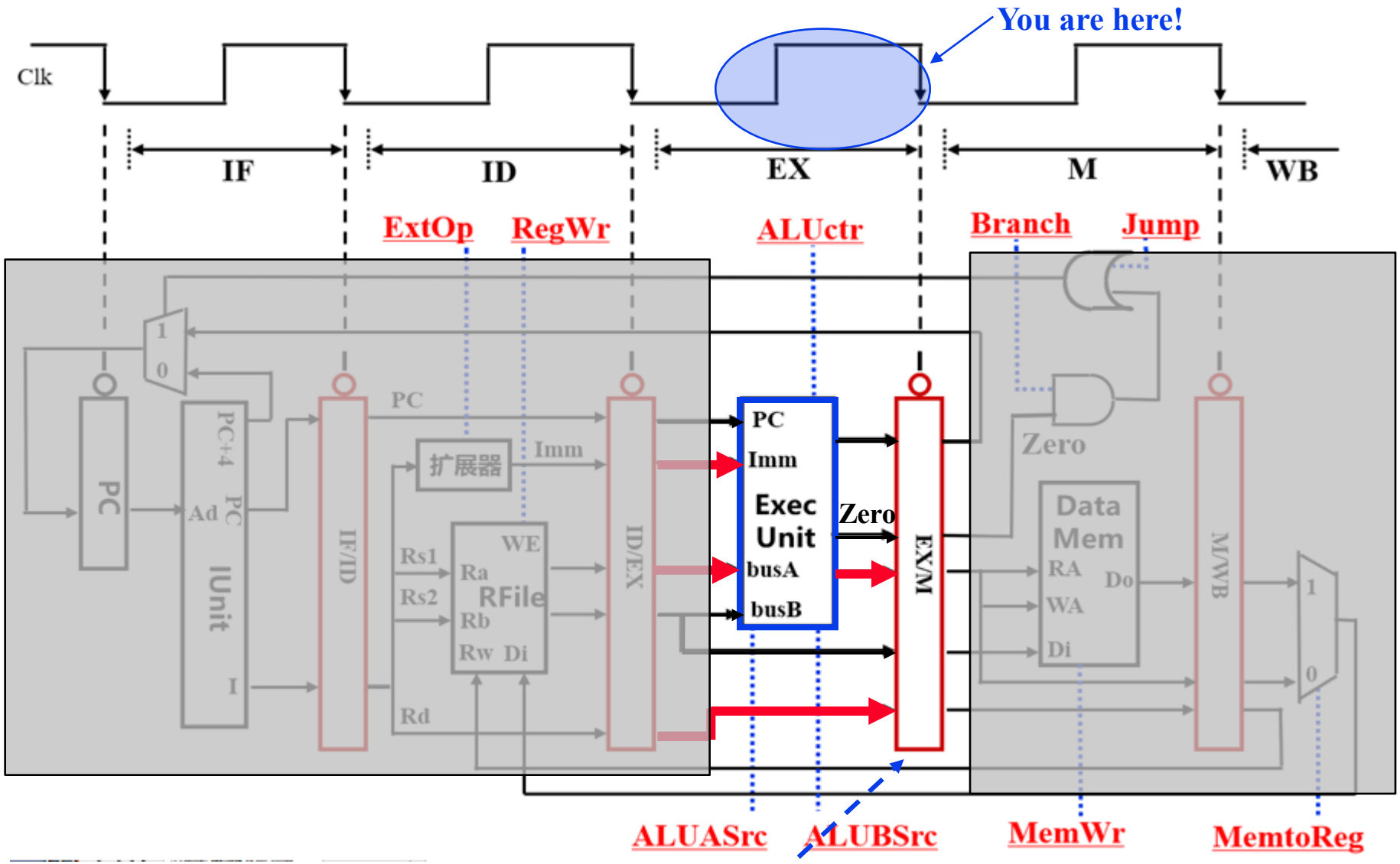
执行部件功能是：ALU运算(PC+4) 和转移目标地址的计算

提醒：不管是哪条指令，target，zero，ALUout这三个位置都会有数据，都会送达流水段寄存器！



jal rd, imm20

再看一下：EX/M这个流水段寄存器

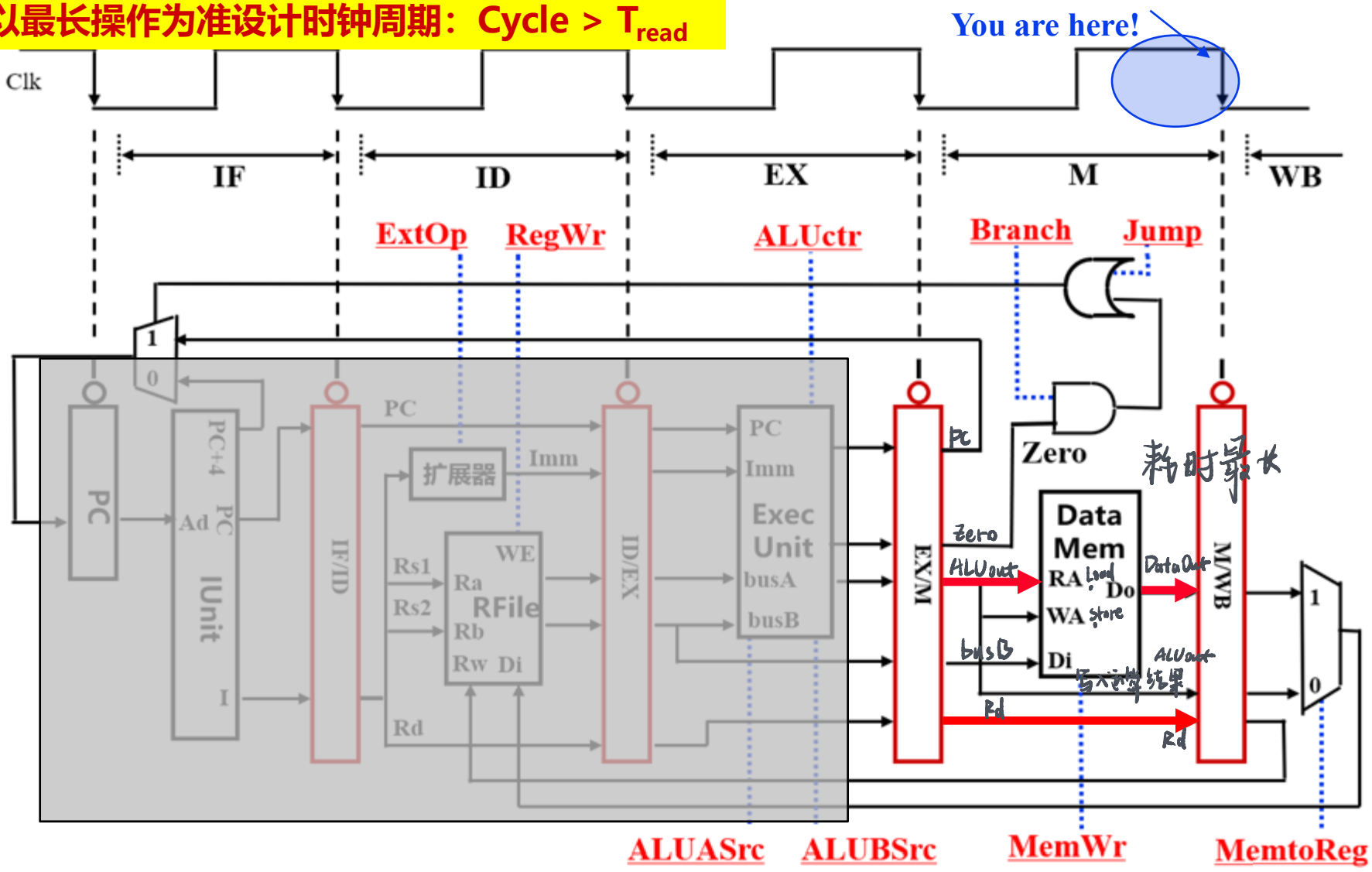


记录了：跳转地址，Zero，ALU运算结果，busB（sw指令后面要用到），rd

Load指令的存储器读(M)周期

- 第10单元指令：lw x8, 0x100(x9)　　功能：R[x8] <- M[R[x9]+SEXT(0x100)]

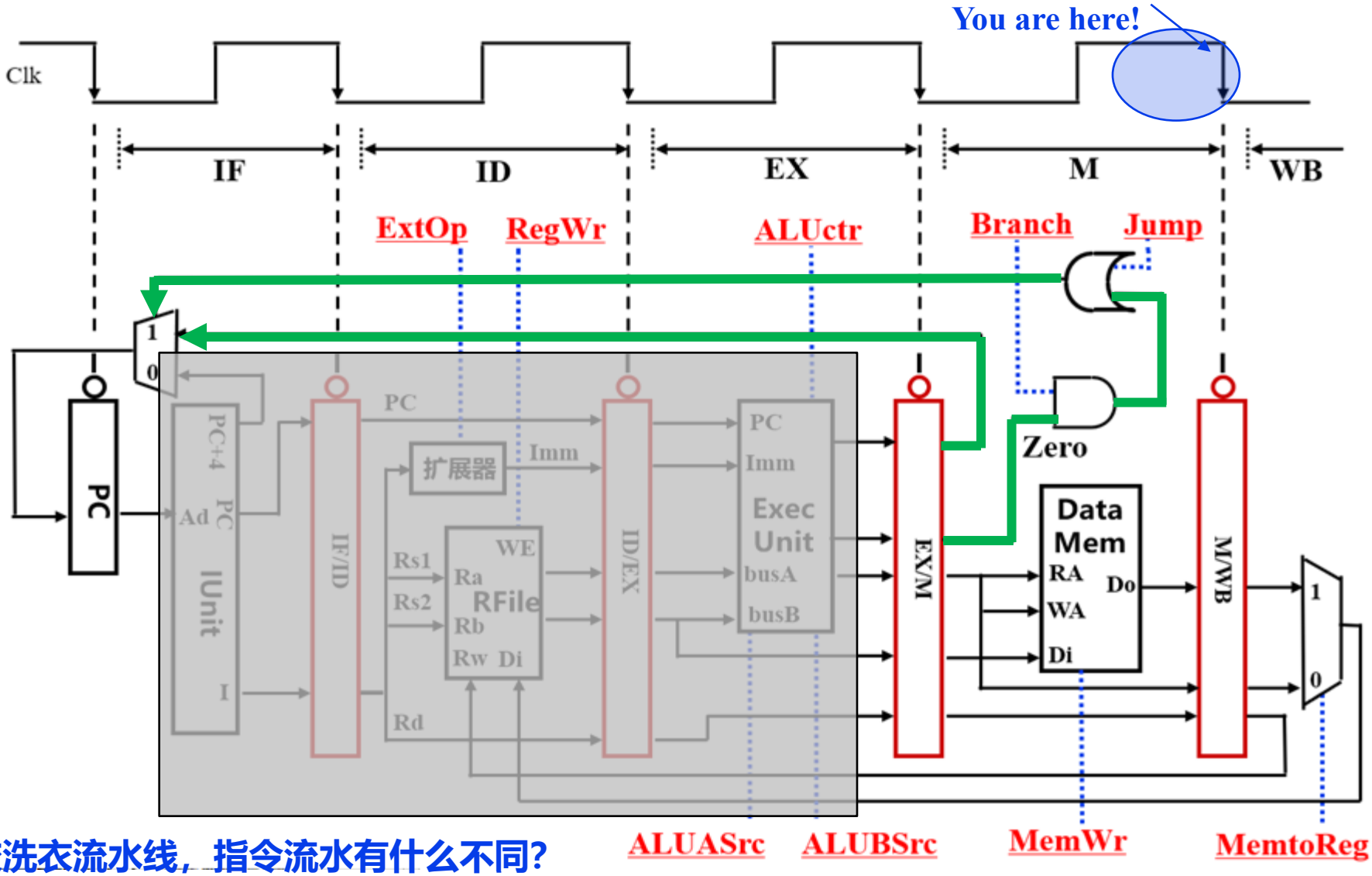
以最长操作为准设计时钟周期：Cycle > T_{read}



Load指令控制信号：Branch=0、Jump=0、MemWr=0

其他指令？B-型: Branch=1，　J-型: Jump=1，　S-型: MemWr=1

如果是Beq指令呢？

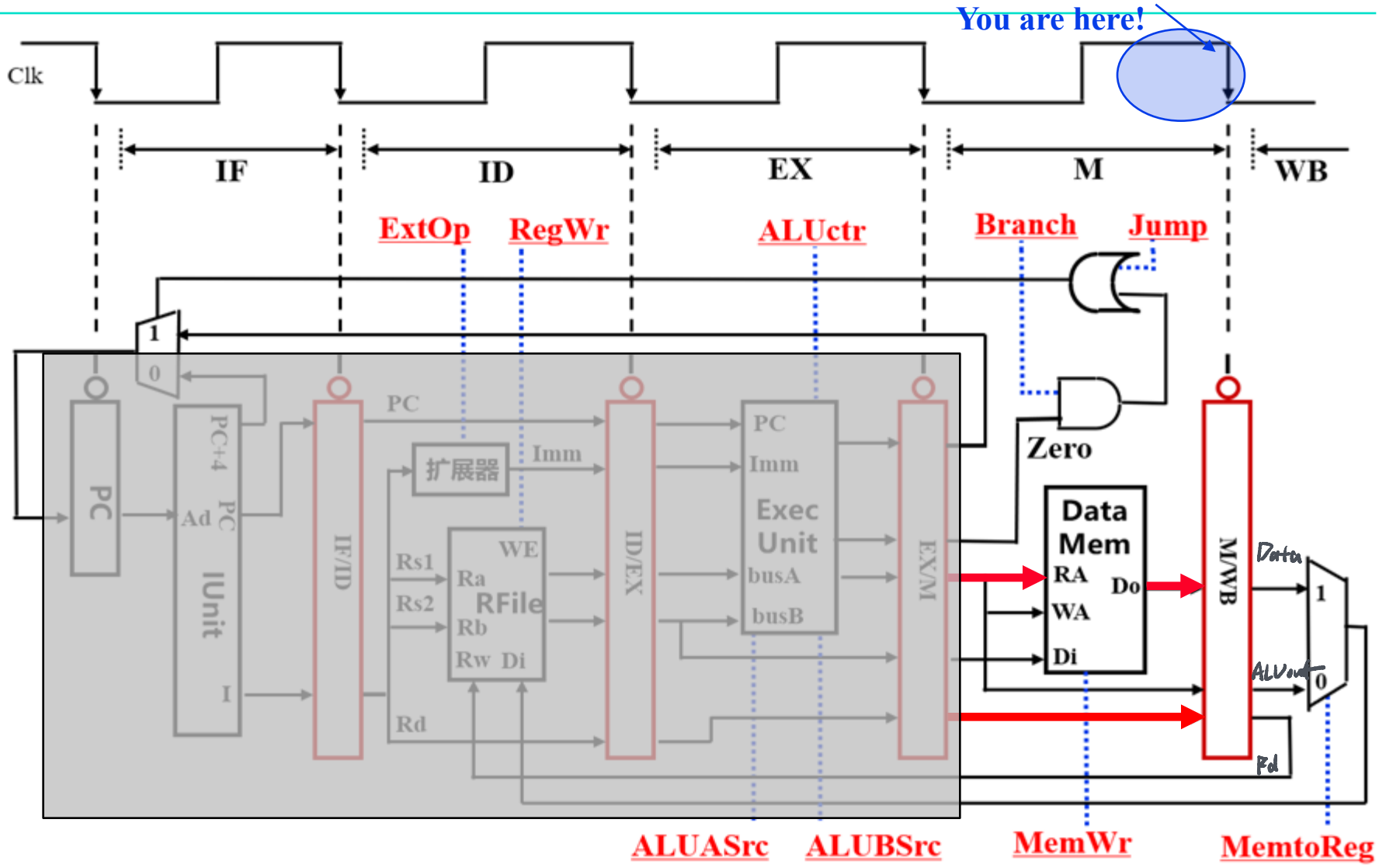


比较洗衣流水线，指令流水有什么不同？

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。

再看一下：M/Wr这个流水段寄存器

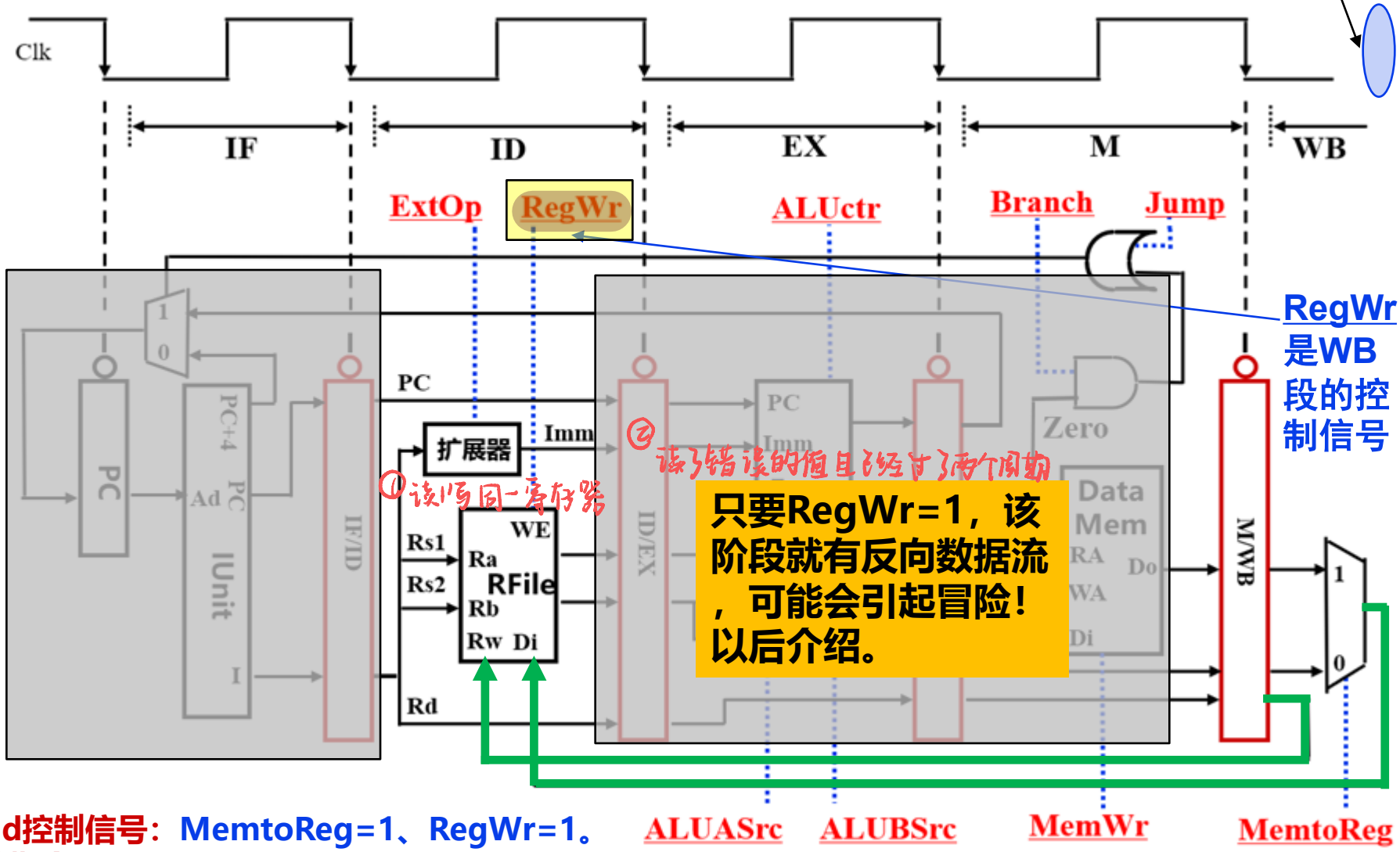


记录了：内存读出的数据，ALU运算结果，rd

Load指令的回写（Write Back）阶段

You are here!

◦ 第10单元指令： lw x8, 0x100(x9)　　功能：R[x8] <- M[R[x9]+SEXT(0x100)]



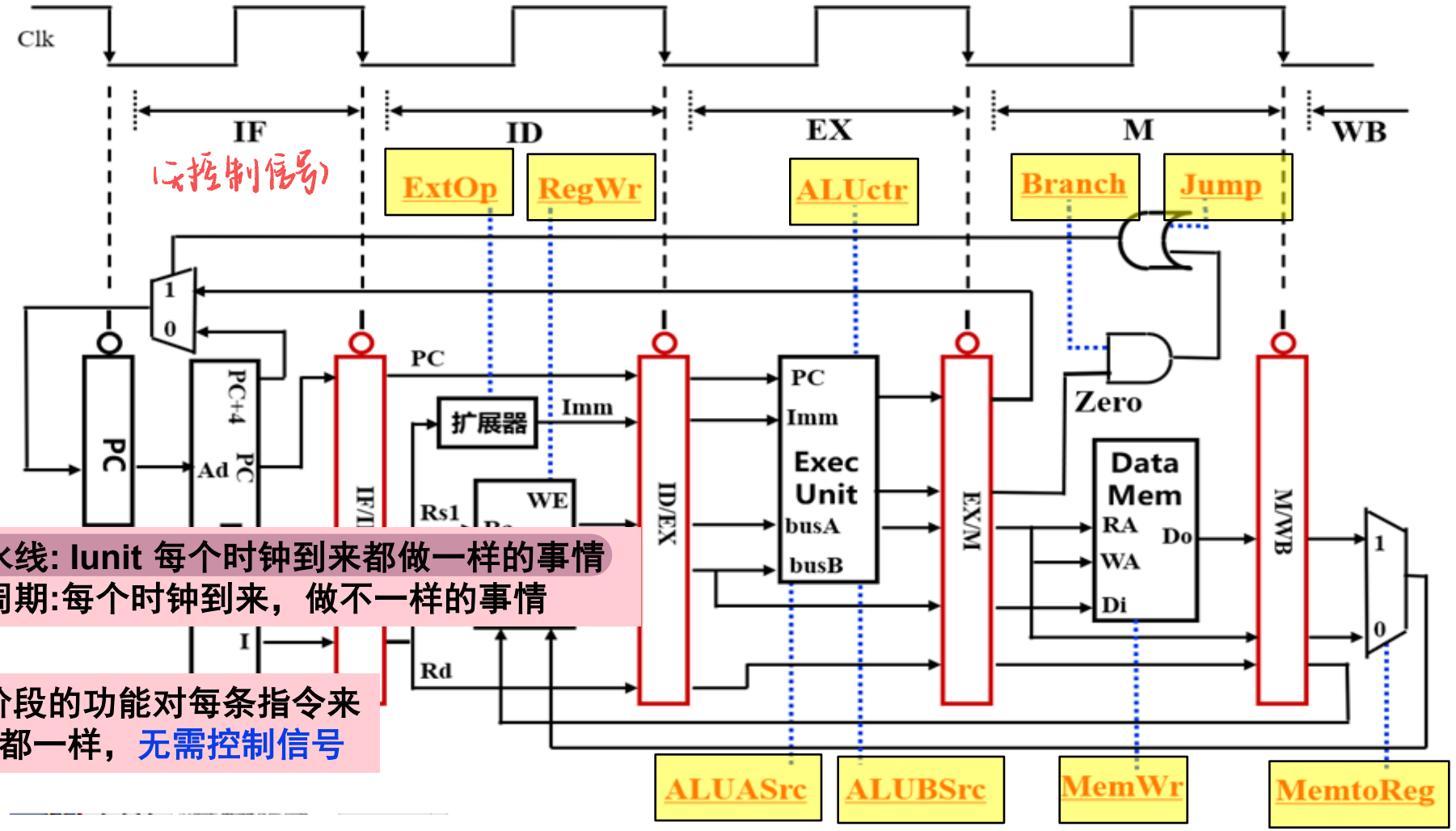
Load控制信号：MemtoReg=1、RegWr=1。

其他指令呢?

R-型、I-型运算、U-型、J-型: MemtoReg=0、RegWr=1；B-型、S-型: MemtoReg=x、RegWr=0

流水线中的Control Signals有哪些？

- 主要考察: 第N阶段的控制信号，它取决于是某条指令的某个阶段。
  - N = ID、EX、M、WB (只有这四个阶段有控制信号）



流水线: Iunit 每个时钟到来都做一样的事情
多周期:每个时钟到来，做不一样的事情

IF阶段的功能对每条指令来说都一样，无需控制信号

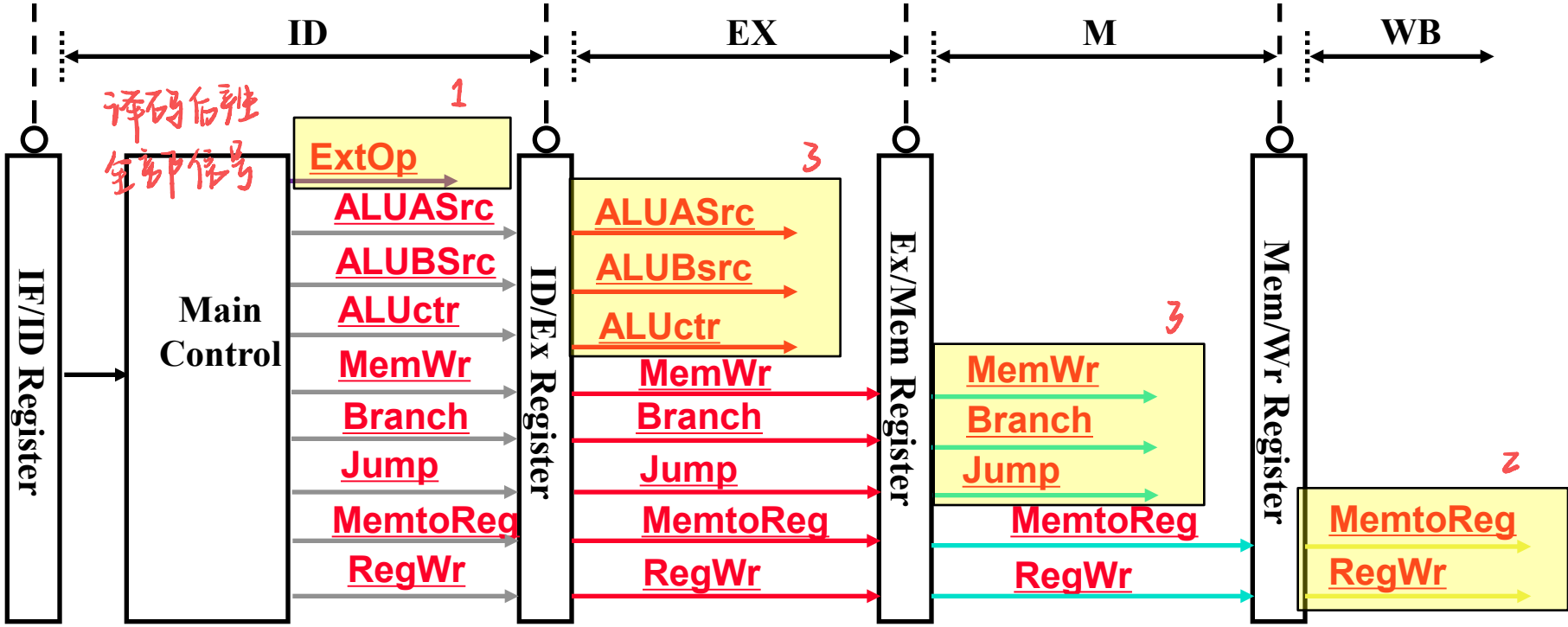
流水线中的Control Signals有哪些？（续）

- 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!
  - IF阶段没有控制信号
  - ID阶段 ExtOp (扩展操作)：除R-型外的所有指令都需要控制
  - EX阶段的控制信号有三个
    - ALUASrc (ALU的A口来源)：1- 来源于PC；0- 来源于BusA
    - ALUBSrc (ALU的B口来源)：0- BusB；1- “4”；2- Imm
    - ALUctr (控制ALU执行不同的操作)： 4位编码
  - M阶段的控制信号有三个
    - MemWr (DM的写信号)：Store指令时为1，其他指令为0
    - Branch (是否为B-型指令)：B-型指令时为1，其他指令为0
    - Jump(是否为J-型指令)：J-型指令时为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- ALU输出；0- DM输出 Data Memory
    - 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

流水线中的控制信号——生成、存储和传递

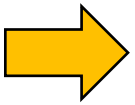
- 在取数/译码（ID）阶段产生本指令所有控制信号
  - ID信号 (ExtOp) 在当前周期中使用
  - EX信号 (ALUAScr, ALUBSrc, ...) 在1个周期后使用
  - M信号 (MemWr, Branch, ...) 在2个周期后使用（改写PC）
  - WB信号 (MemtoReg, RegWr) 在3个周期后使用

所以，控制信号也要保存在流水段寄存器中！



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！

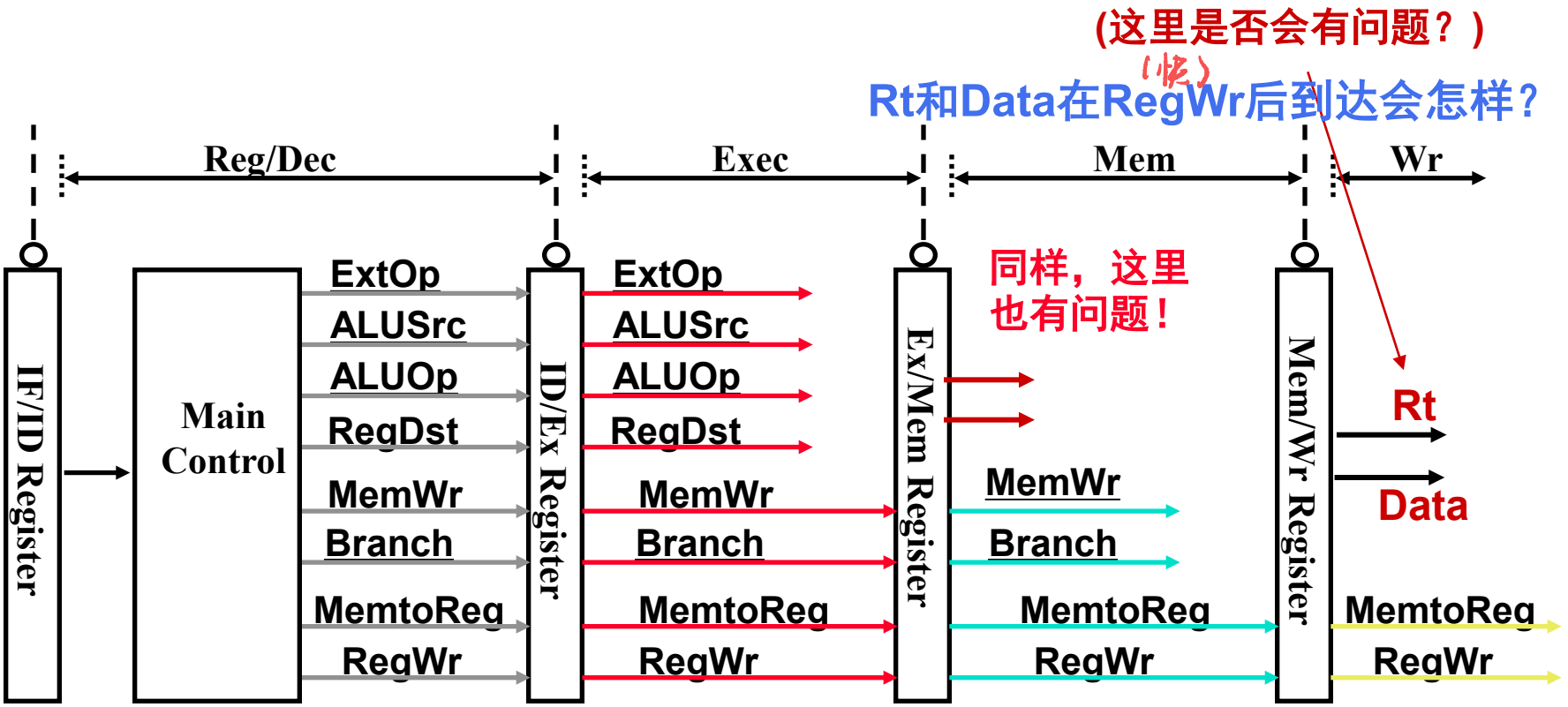
控制逻辑 的设计



- 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！
    （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！
    （单周期和多周期控制逻辑各是怎样设计的？）
  - 设计过程
    - 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

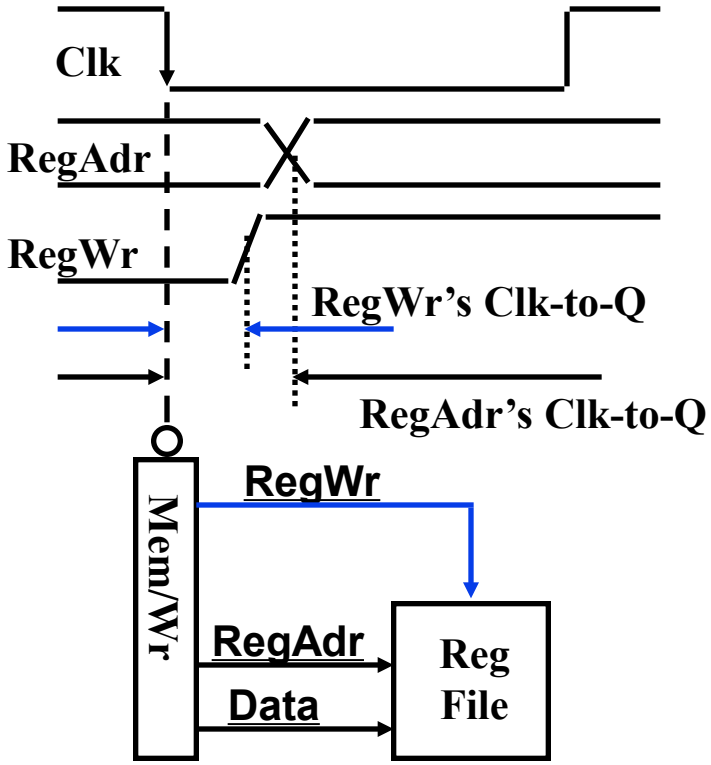
- 某时刻（某个时钟周期内），
- CPU数据通路中的不同阶段同时工作，
- 分别执行着不同的指令，
- 而这些不同的指令都能够从流水段寄存器得到自己所需的控制信号和数据

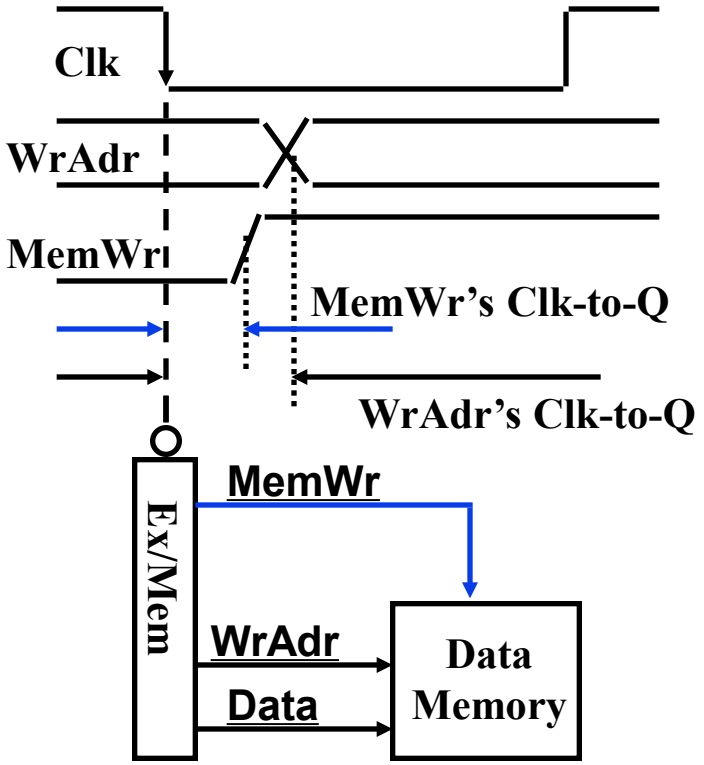
（不要求）流水线中的“竞争”问题



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！

* （不要求）Wr阶段的开始: 存在一个实际的问题！

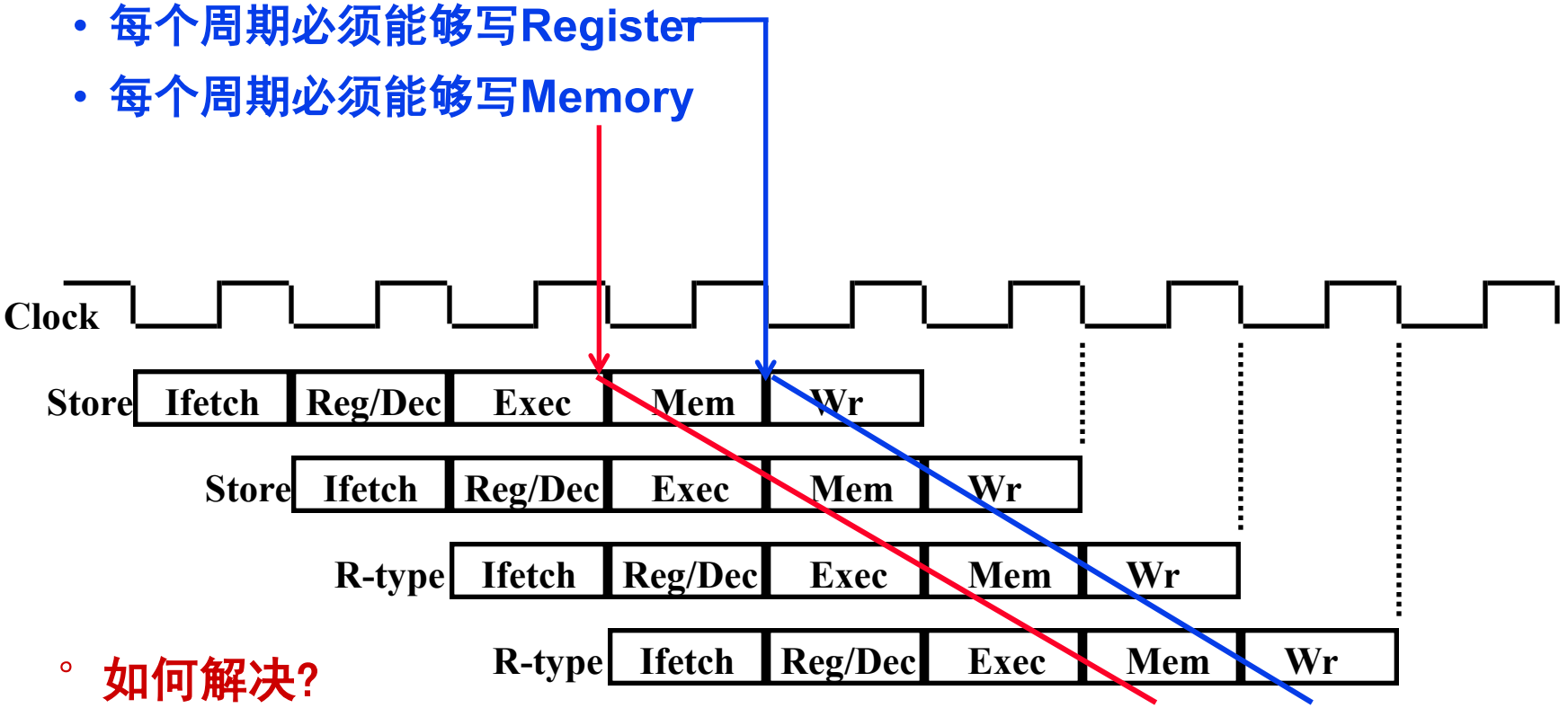




- 在流水线中存在地址 和 写使能之间的“竞争”问题
  - Wr段开始时若RegAdr's(Rd/Rt)Clk-to-Q > RegWr's Clk-to-Q, 则错写寄存器！
  - Mem 阶段开始时若WrAdr's Clk-to-Q > MemWr's Clk-to-Q, 则错写存储器！

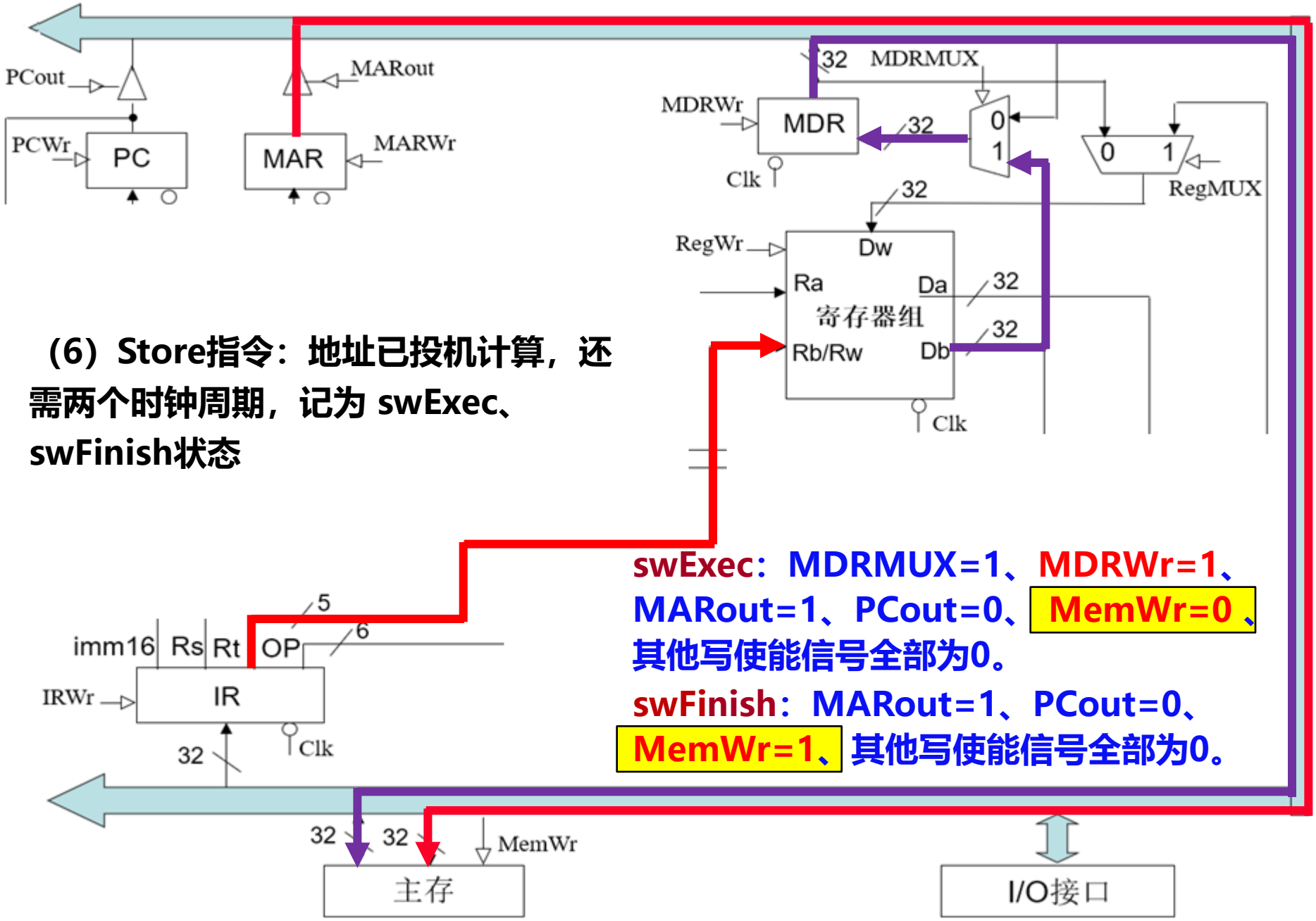
（不要求）流水线中的“竞争”问题

- 多周期中没有出现Addr、data 和 WrEn之间的竞争，因为:
  - 在第 N周期结束时，让Addr和data信号有效
  - 在第 N + 1周期让WrEn有效
- 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



- 如何解决?

✳回顾：多周期处理器中的Store指令



（6）Store指令：地址已投机计算，还需两个时钟周期，记为 swExec、swFinish状态

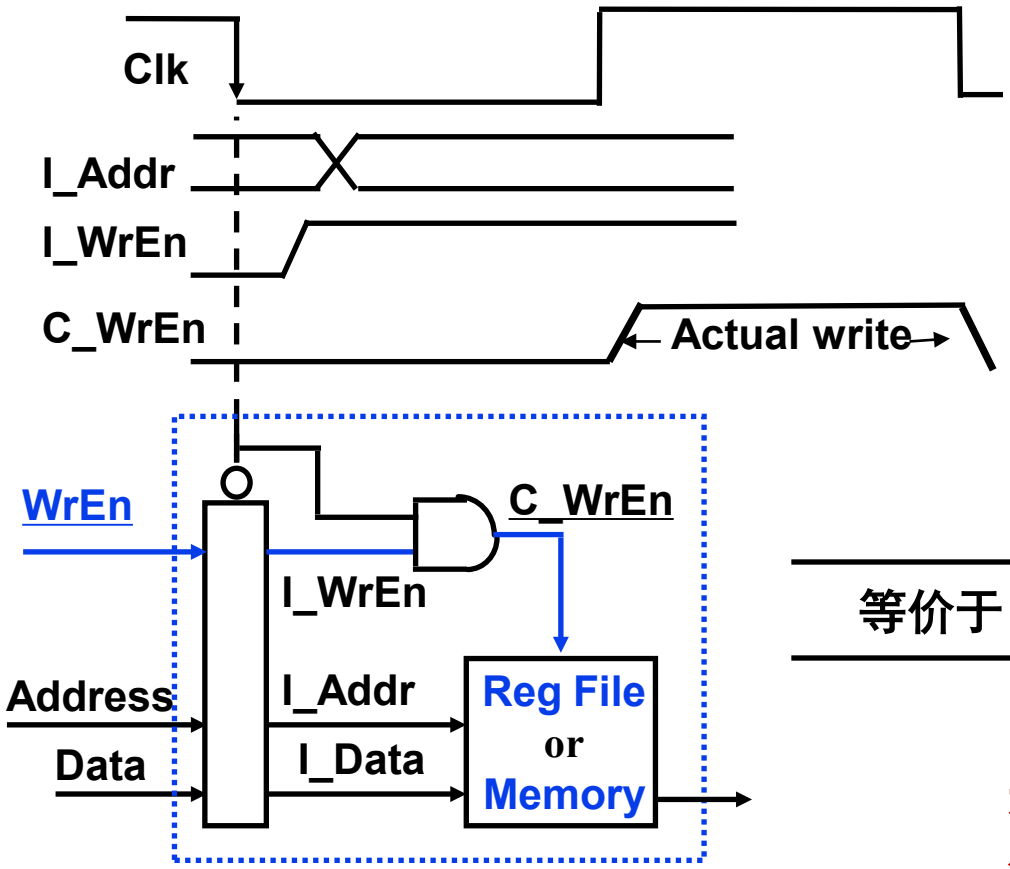
swExec：MDRMUX=1、MDRWr=1、MARout=1、PCout=0、MemWr=0、其他写使能信号全部为0。

swFinish：MARout=1、PCout=0、MemWr=1、其他写使能信号全部为0。

（不要求）寄存器组的同步和存储器的同步

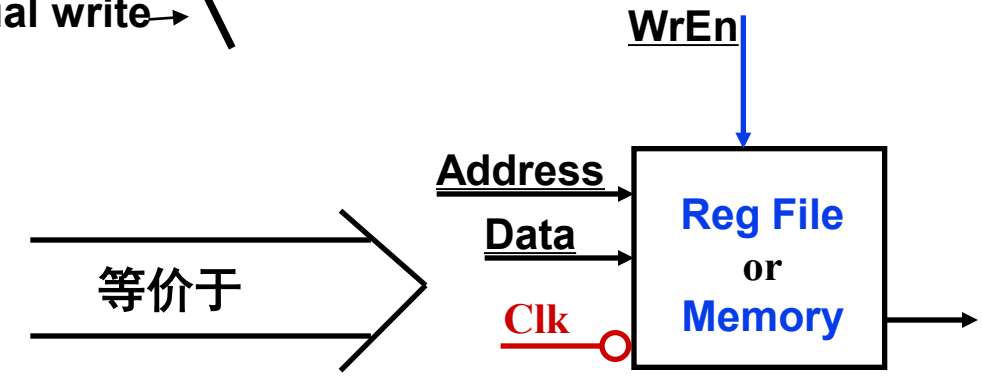
- 解决方案: 将Write Enable和时钟信号相“与”

须由电路专家确保不会发生“定时错误”（即：能合理设计“Clock”!）



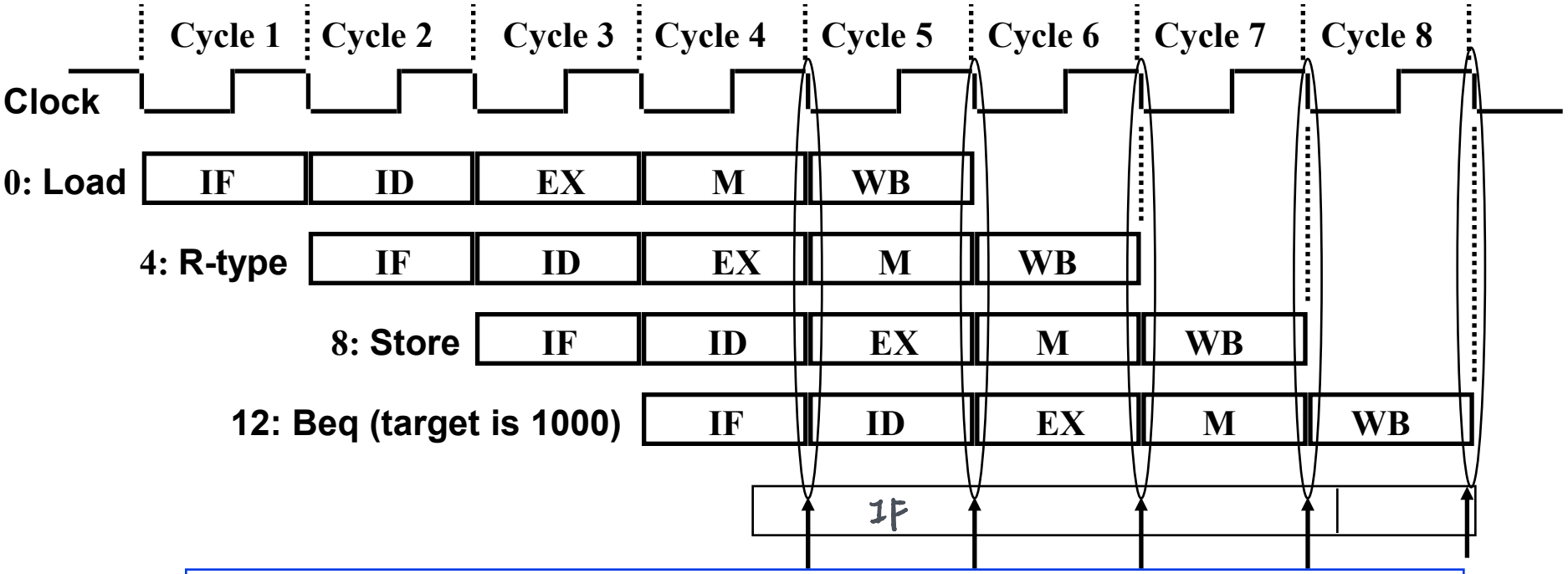
1. Addr, Data和 WrEn 必须在Clk边沿到来后至少稳定一个 set-up时间

2. Clk高电平时间 大于 写入时间



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

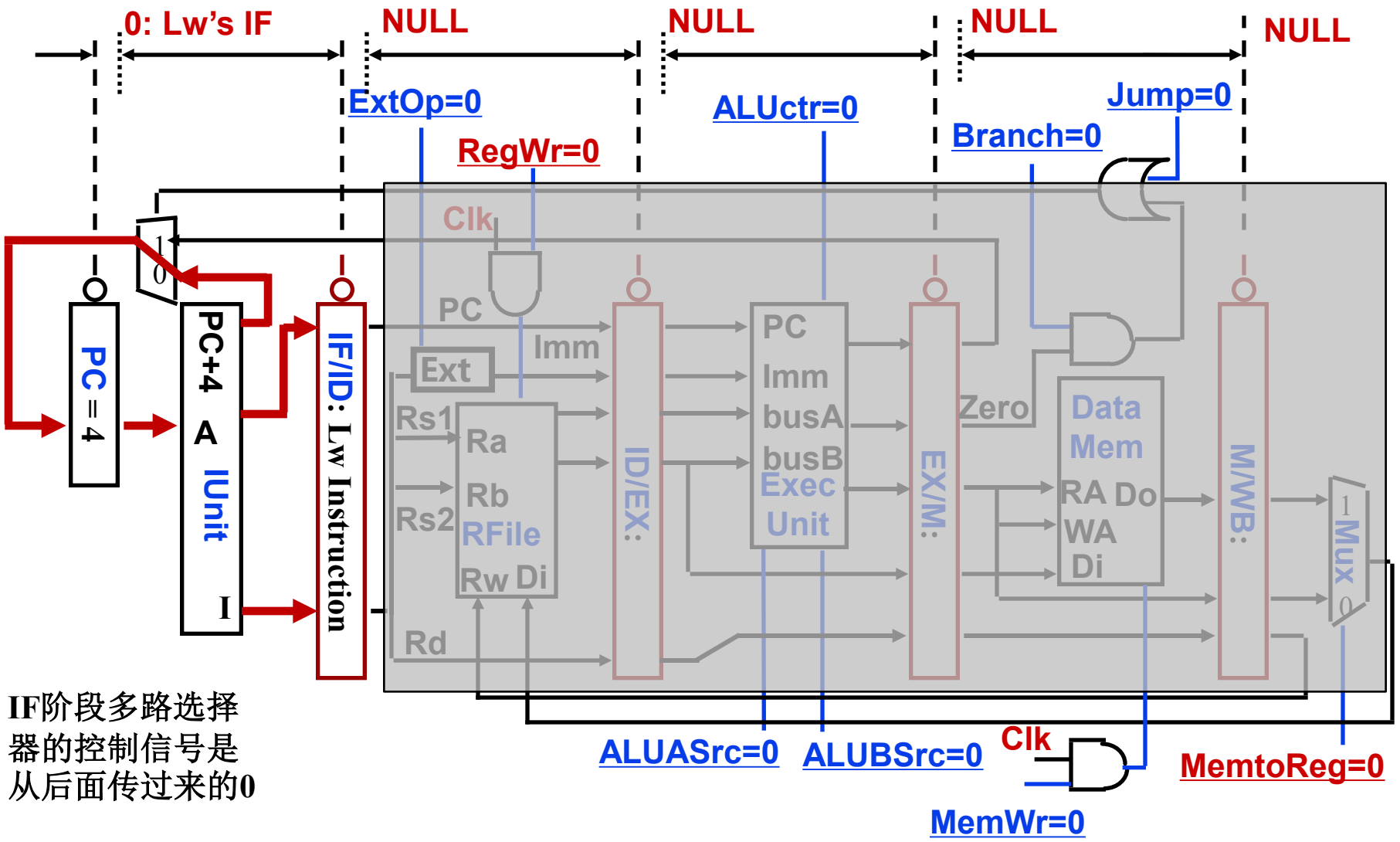


- 注意：最开始的时刻，所有流水段寄存器初始化为0！！！

注意：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。

第一周期结束时的状态：

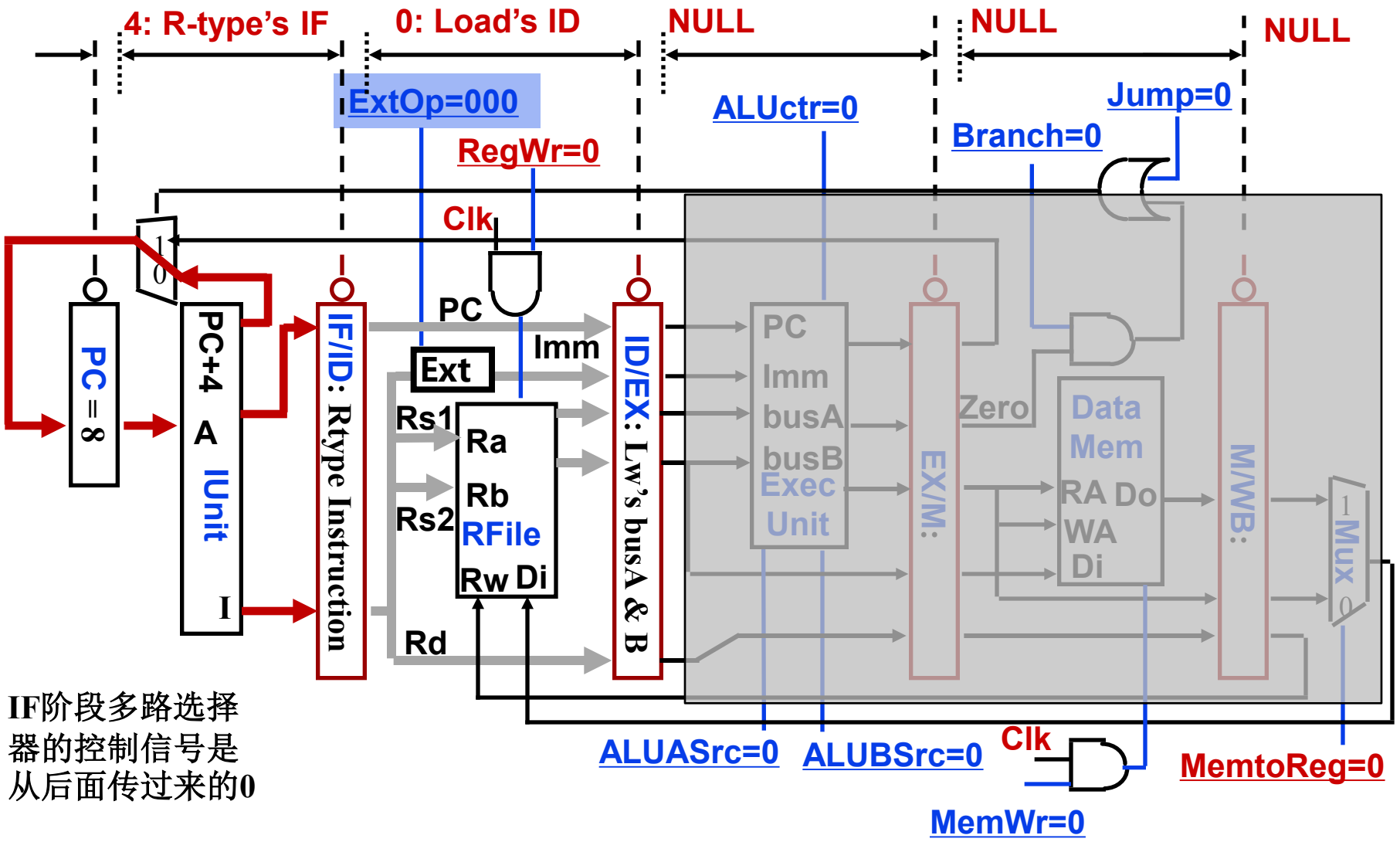
- 0: Load's IF



IF阶段多路选择器的控制信号是从后面传过来的0

第二周期结束时的状态：

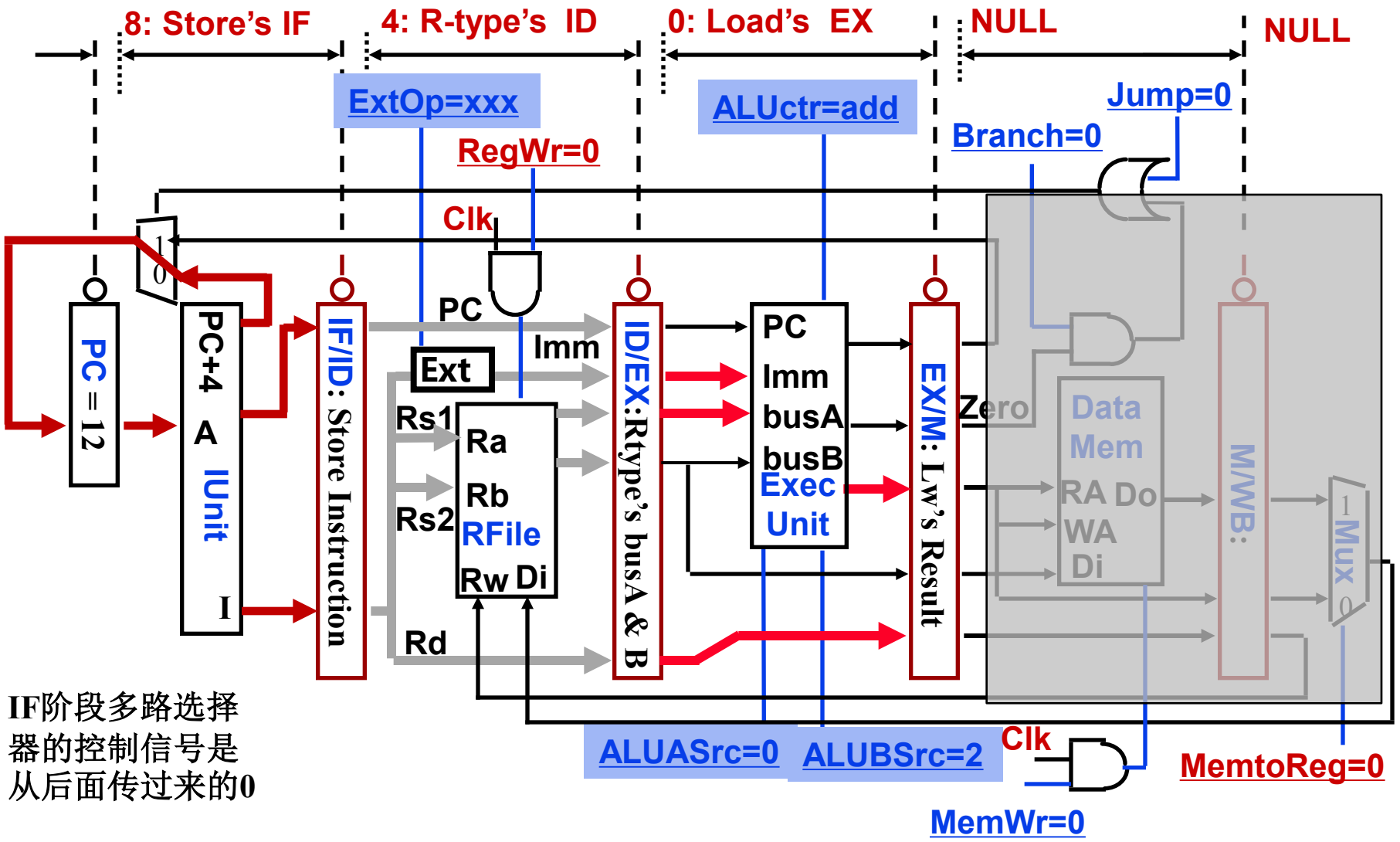
- 4: R-type's IF　0: Load's ID



IF阶段多路选择器的控制信号是从后面传过来的0

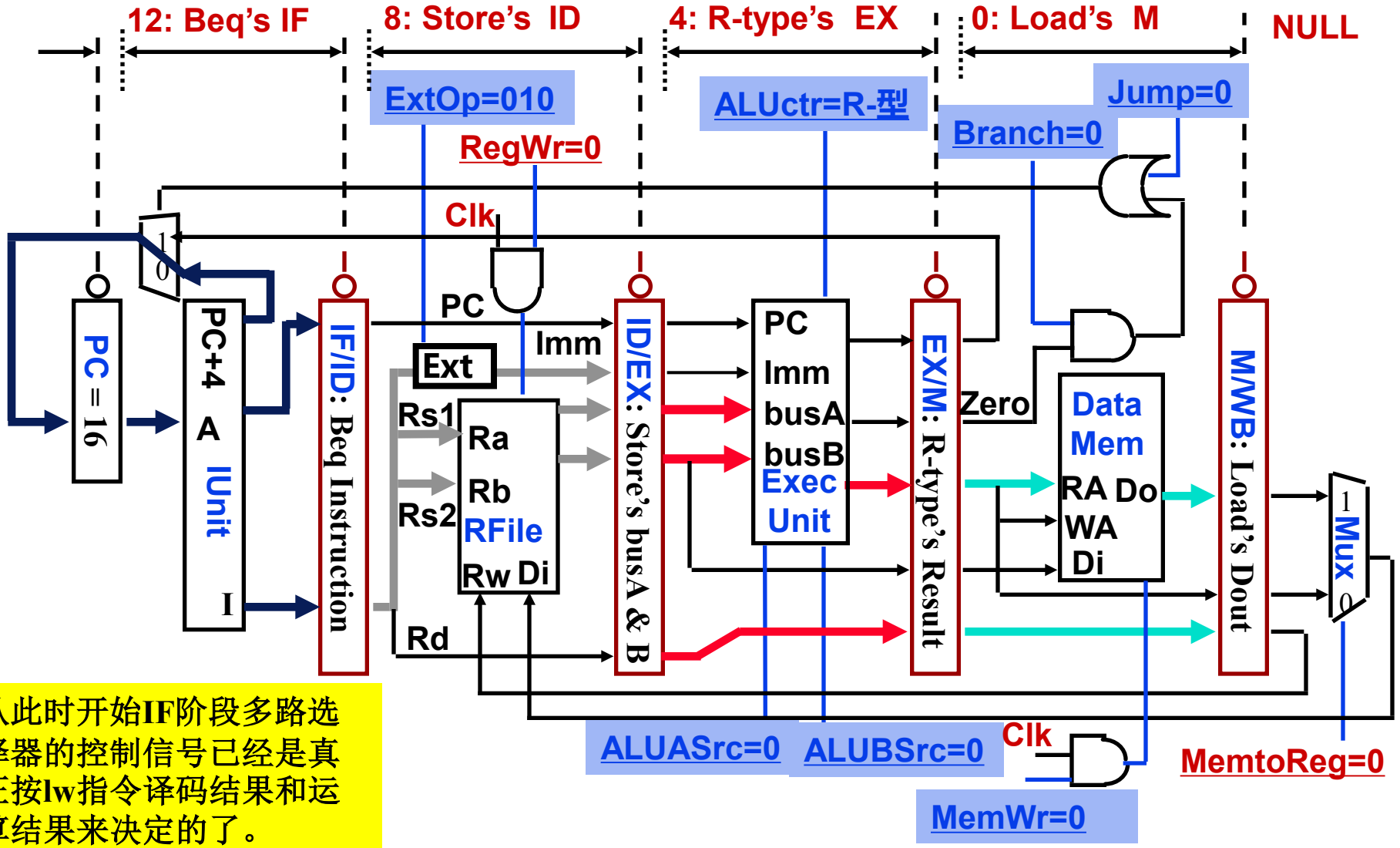
第三周期结束时的状态：

- 8: Store's IF　　4: R-type's ID　　0: Load's EX



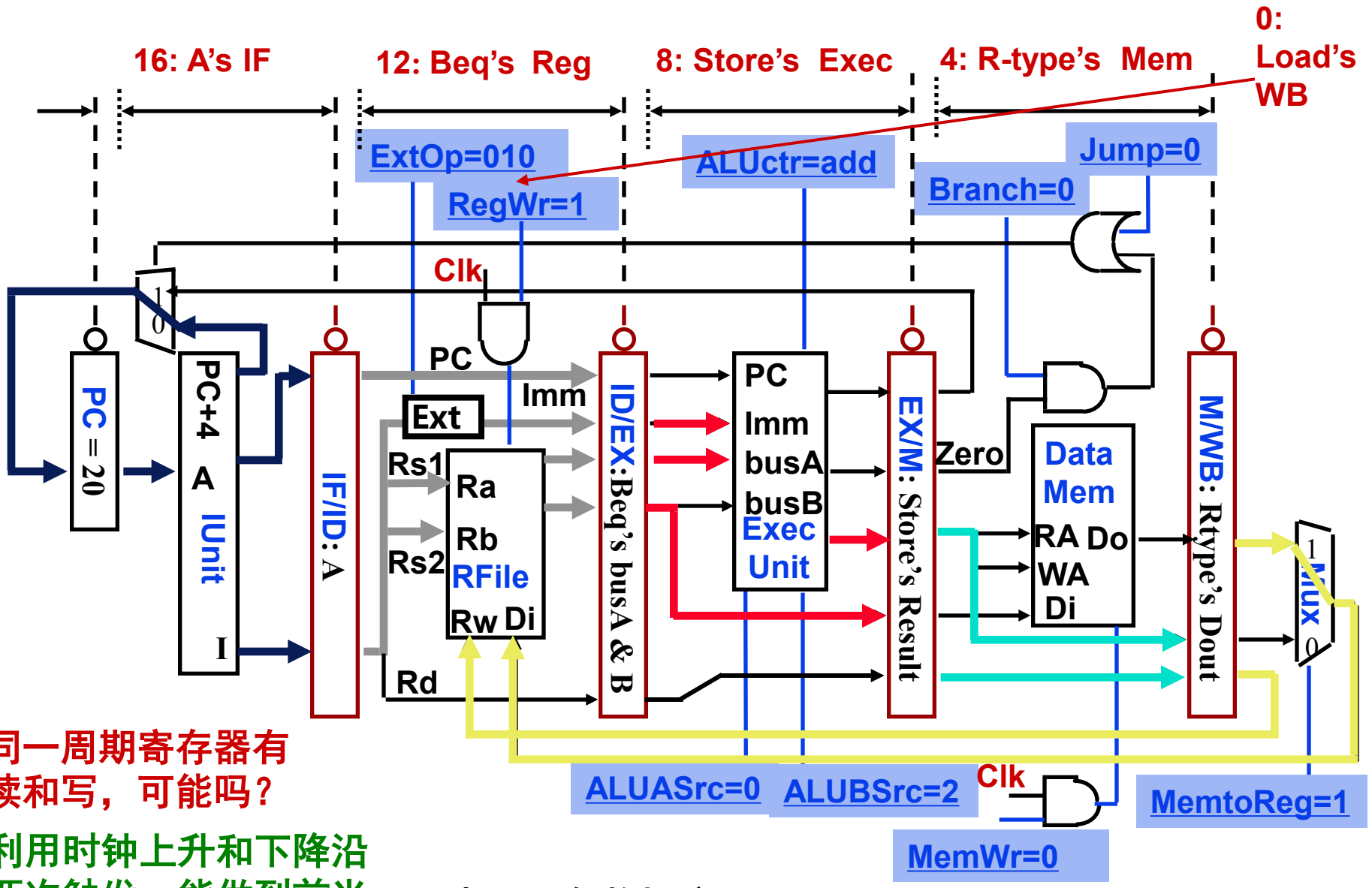
IF阶段多路选择器的控制信号是从后面传过来的0

第四周期结束时的状态：



从此时开始IF阶段多路选择器的控制信号已经是真正按lw指令译码结果和运算结果来决定的了。

第五周期结束时的状态：

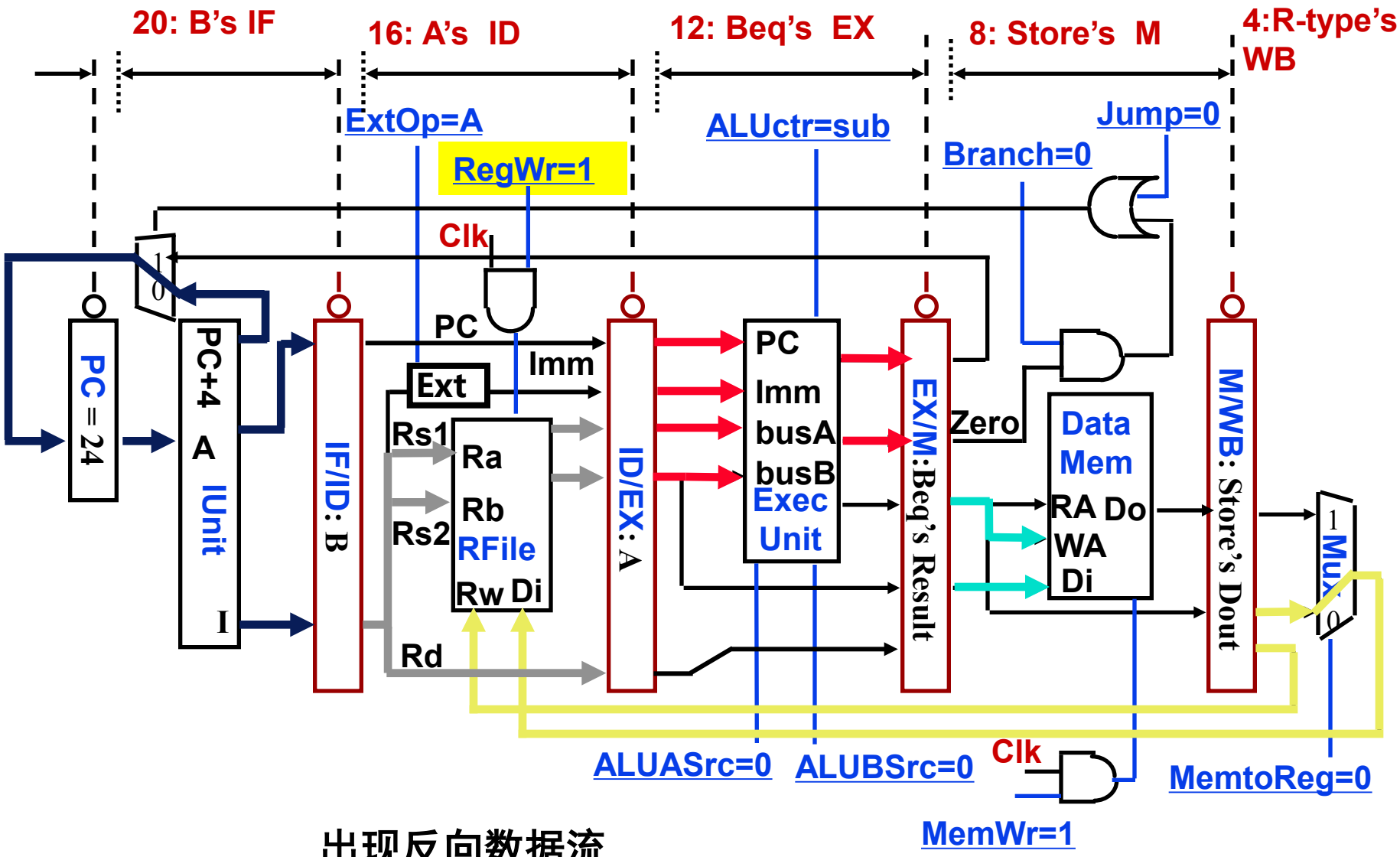


同一周期寄存器有读和写，可能吗？

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

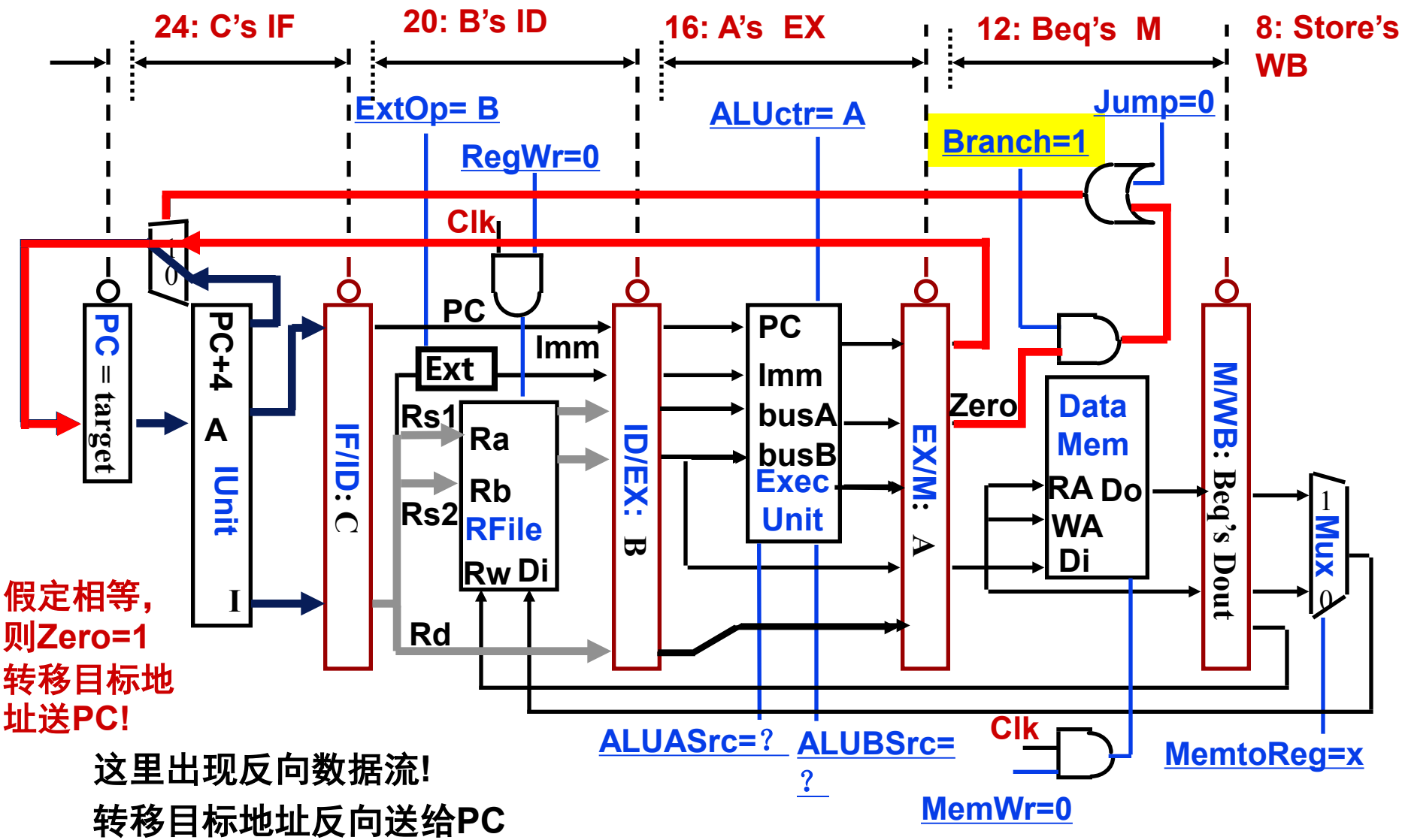
出现反向数据流

第六周期结束时的状态：



出现反向数据流

第七周期结束时的状态：

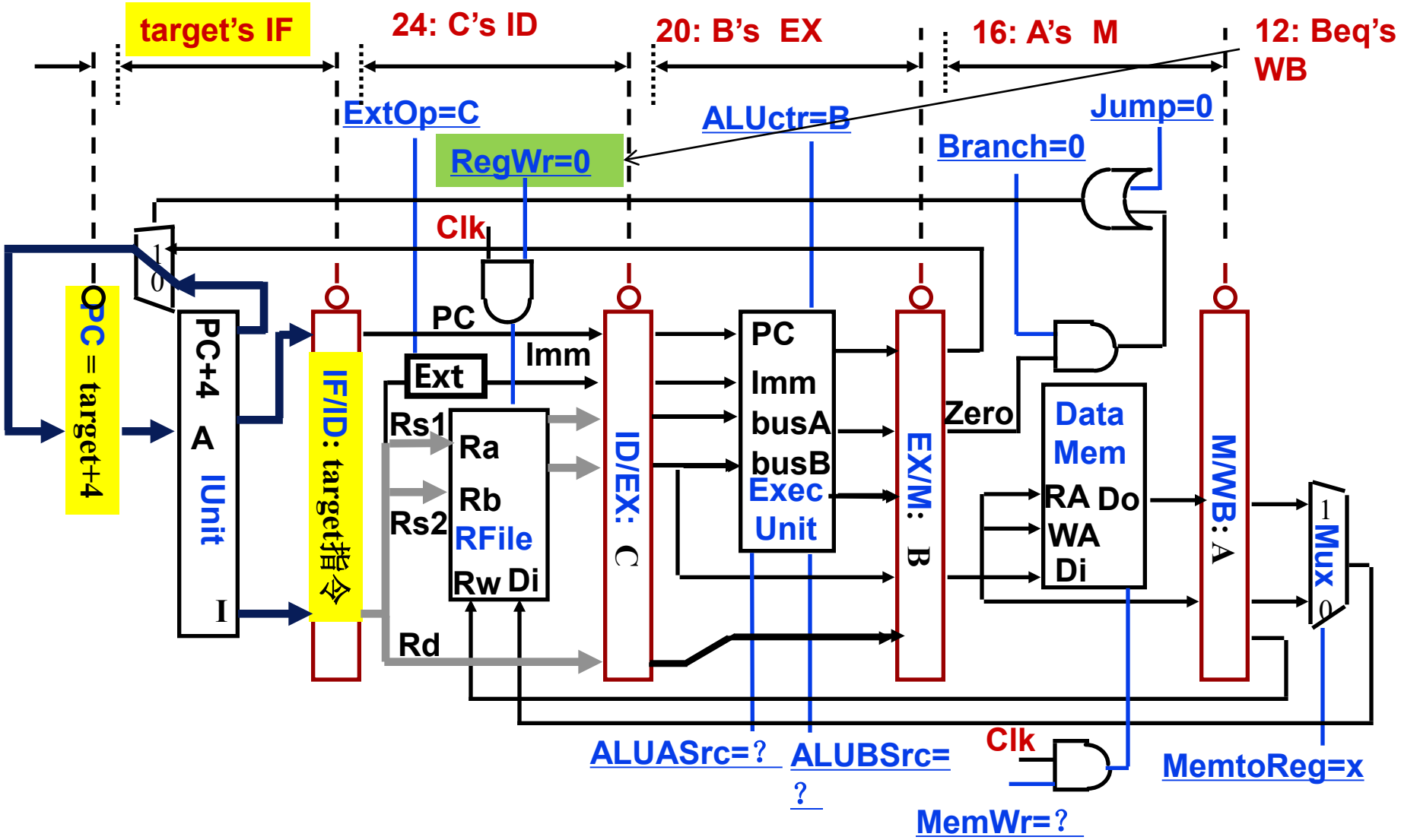


这里出现反向数据流!

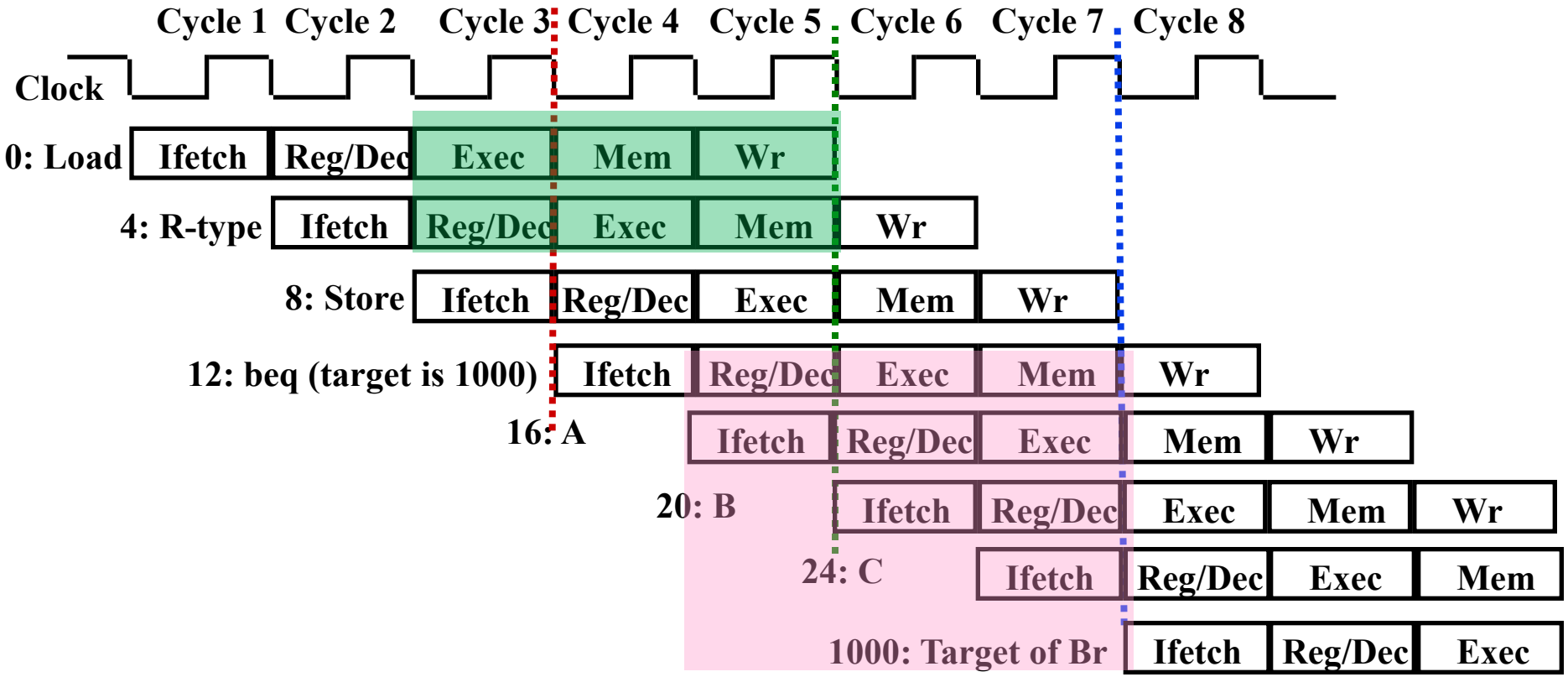
转移目标地址反向送给PC

可能会导致控制冒险！

第8周期结束时的状态：

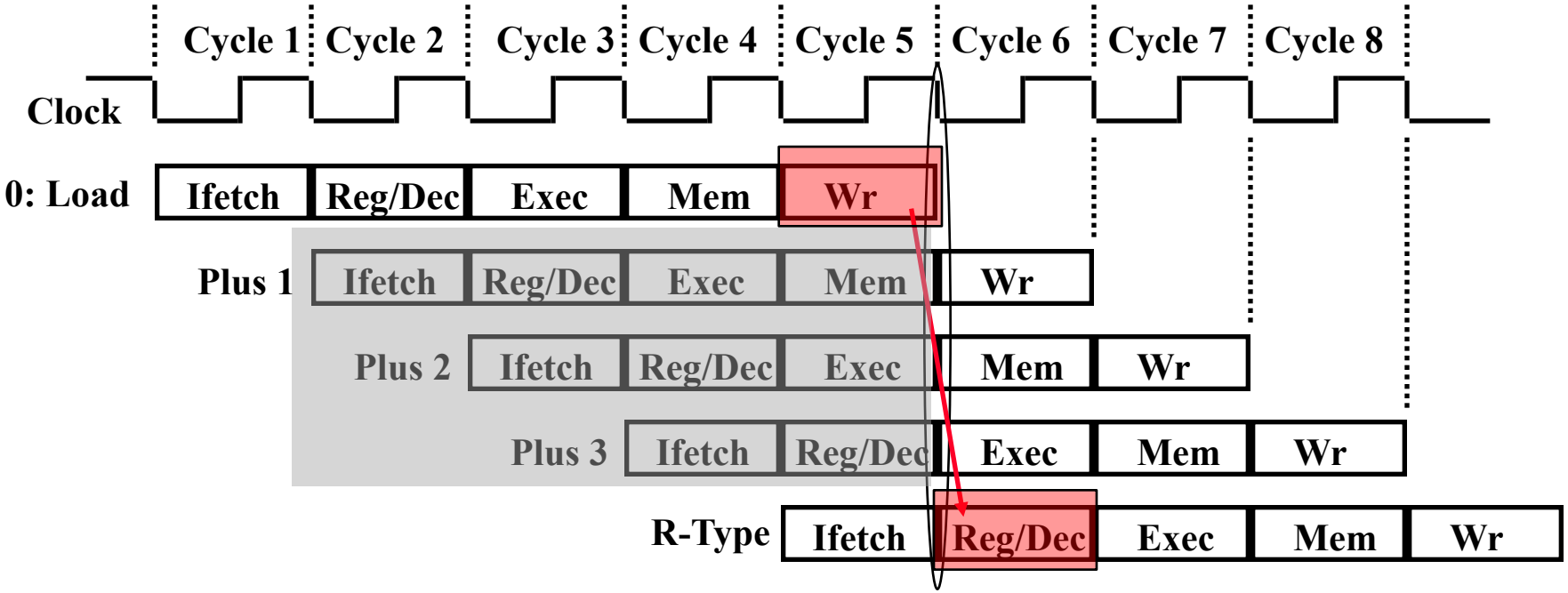

target's IF
24: C's ID
20: B's EX
16: A's M
12: Beq's WB
ExtOp=C
ALUctr=B
Jump=0
Branch=0
RegWr=0
Clk
PC = target+4
PC+4
A
IUnit
I
IF/ID: target指令
PC
Imm
Ext
Rs1
Rs2
Rd
Ra
Rb
RFile
Rw
Di
ID/EX: C
PC
Imm
busA
busB
Exec
Unit
EX/M: B
Zero
Data
Mem
RA
Do
WA
Di
M/WB: A
1
Mux
0
ALUASrc=?
ALUBSrc=
?
Clk
MemWr=?
MemtoReg=x

总结前面的流水线执行过程——反向数据流



- **回忆刚才的过程：**
  - **beq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？**
    - **如果beq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？**
    - **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？**

装入指令(Load)引起的“延迟”现象



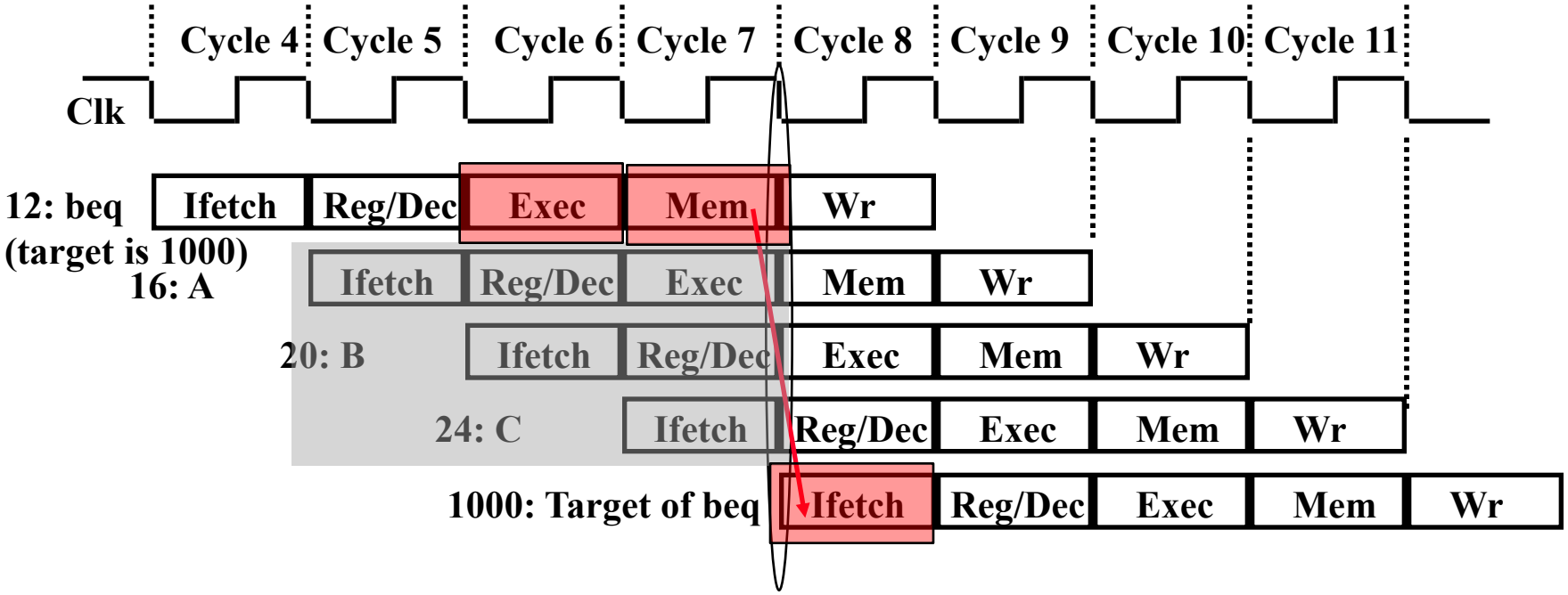
- 尽管Load指令在第一周期就被取出，但:
  - 数据在第五周期结束才被写入寄存器
  - 到第六周期写入的数据才能被用

  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)

寄存器

结构冒险：指令执行不一致

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:
  - 第六周期得到Zero和转移目标地址
  - 转移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 ）转移的

  （注：也称为分支冒险或转移冒险（Branch Hazard） ）

单周期 vs 流水线计算机的性能

假设在单周期处理器中，各主要功能单元的操作时间为：

- 存储单元（取指令、存取存储器里的数据）：200ps
- ALU和加法器：100ps
- 寄存器、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，不考虑任何特殊情况（延迟），则单周期和流水线的实现方式相比，哪个更快？吞吐率呢?

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	0	550 ps
Branch	200	50	100	0	0	350 ps
Jump	200	0	100	0	50	350 ps

- 不再需要关心指令种类占比。CPI都是1。
- 单周期，时钟周期600ps，吞吐率1/600ps = 1.67×10^9指令/秒
- 流水段寄存器延时50ps，最长阶段200ps，所以时钟周期是250ps（如果说忽略流水段寄存器延时，就是200ps），吞吐率1/250ps = 4×10^9指令/秒，是单周期的大约2.4倍

第五讲小结

本章作业：习题3、4、5、6、11、13、14、17、18

- 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)段
    - 指令译码、立即数扩展（Extender）、读Rs和Rt（寄存器读口）
  - 执行(EX)段
    - 计算转移目标地址、ALU运算（ALU、Adder）
  - 存储器(M)段
    - 读或写存储单元（Data Memory）
  - 写回寄存器(WB)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
  - ID段生成所有控制信号，并随指令执行过程信息同步向后续阶段流动
  - 与单周期处理器的控制器的实现方法一样，无需采用有限状态机
- 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8章 中央处理器（2）

第一讲 中央处理器概述

第二讲 单周期数据通路的设计

第三讲 单周期控制器的设计

第四讲 多周期处理器的设计

第五讲 流水线处理器设计

第六讲 流水线冒险及其处理

第七讲 高级流水线技术

第六讲 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

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

- 结构冒险Structural hazards (hardware resource conflicts):

  现象：同一个部件同时被不同指令所使用

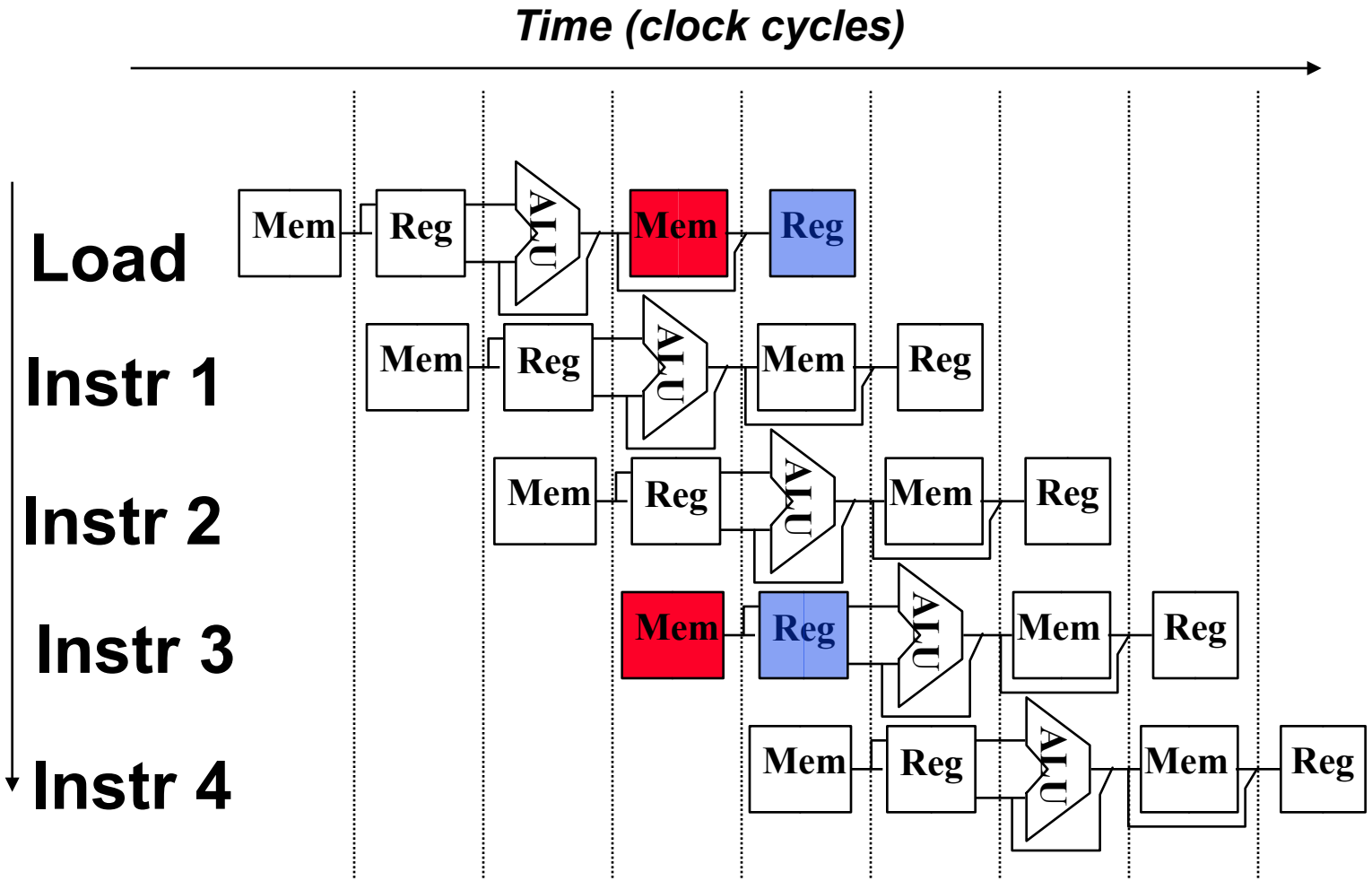
- 数据冒险Data hazards (data dependencies):

  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

- 控制冒险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

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！

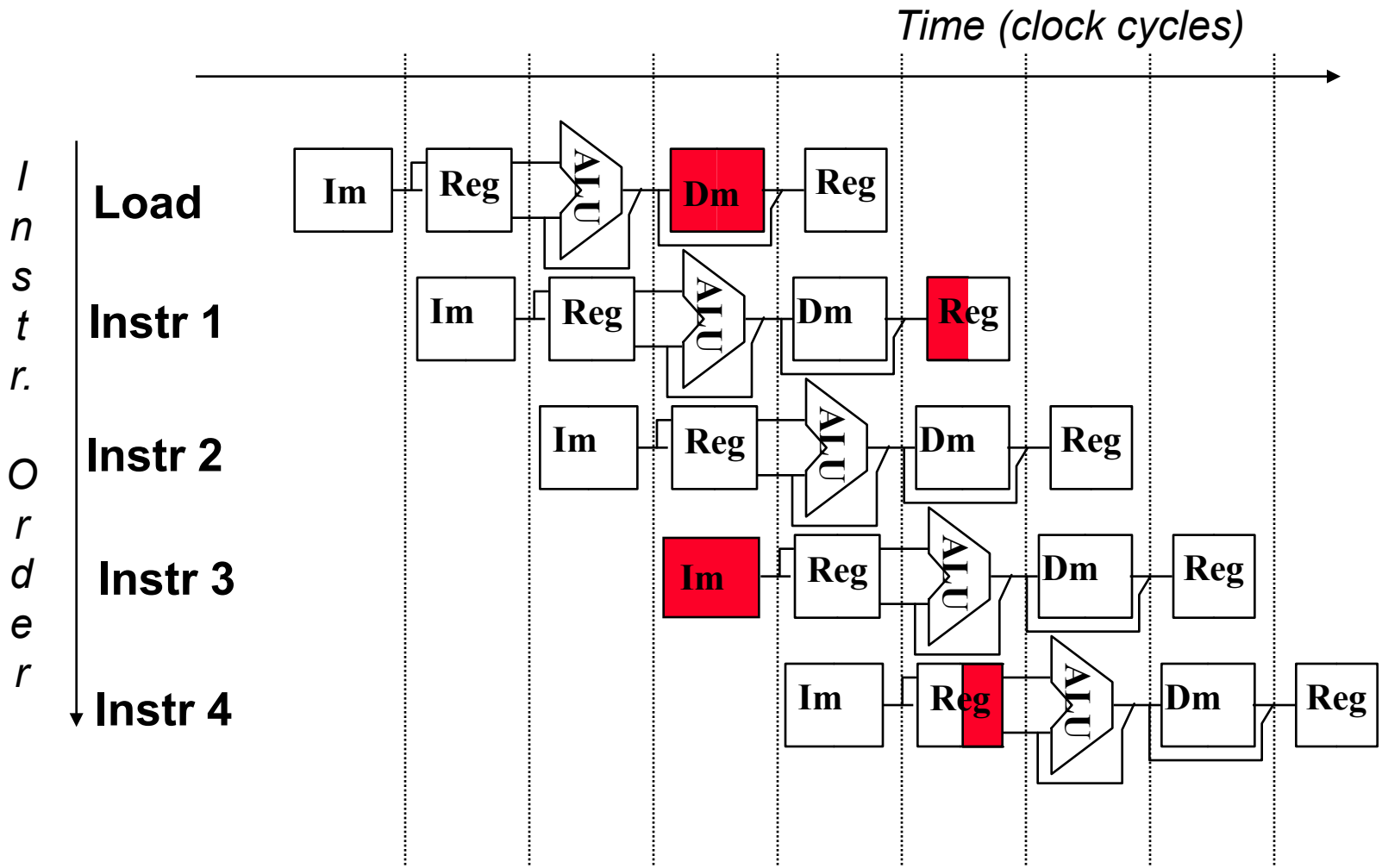
如果不对寄存器堆的写口和读口独立设置的话，则发生冲突！

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。

结构冒险的解决方法

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



数据冒险现象

以下指令序列中，寄存器r1会发生数据冒险

add r1 ,r2,r3

sub r4, r1 ,r3

and r6, r1 ,r7

or r8, r1 ,r9

xor r10, r1 ,r11

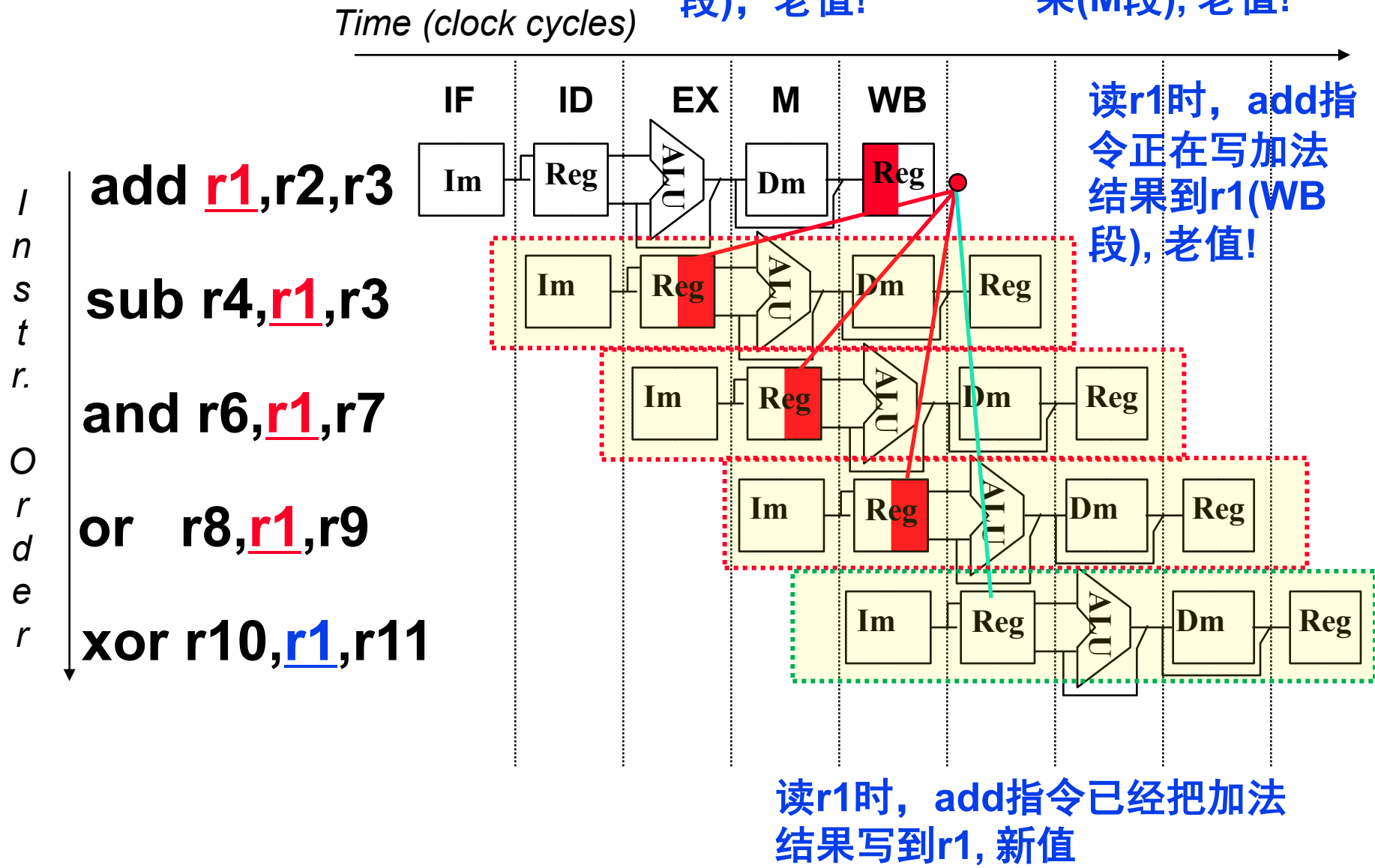
想一下，哪条指令的r1是老的值？
哪条是新的值？

画出流水线图能很清楚理解！

关于r1 的 数据冒险

读r1时，add指令正在执行加法(EX段)，老值!

读r1时，add指令正在传递加法结果(M段), 老值!



读r1时，add指令正在写加法结果到r1(WB段), 老值!

读r1时，add指令已经把加法结果写到r1, 新值

数据冒险现象（小结）

所以——

最后一条指令的r1才是新的值！

如何解决这个问题？

```
add r1 ,r2,r3
sub r4, r1 ,r3
and r6, r1 ,r7
or  r8, r1 ,r9
xor r10, r1 ,r11
```

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

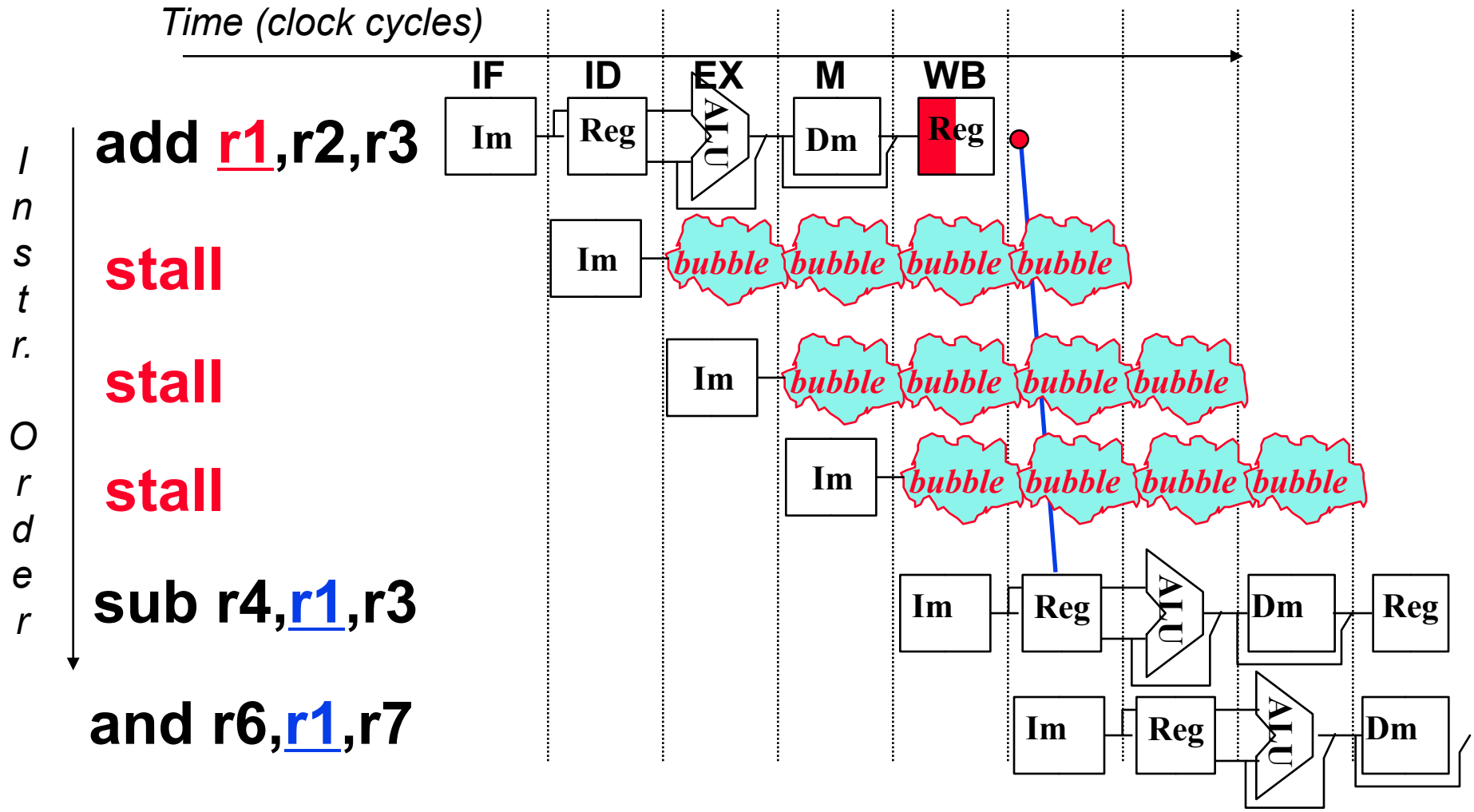
数据冒险的解决方法

- 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）
- 方法4：转发（Forwarding或Bypassing 旁路）技术（不能解决所有数据冒险）
- 方法5：编译优化：调整指令顺序（不能解决所有数据冒险）

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！

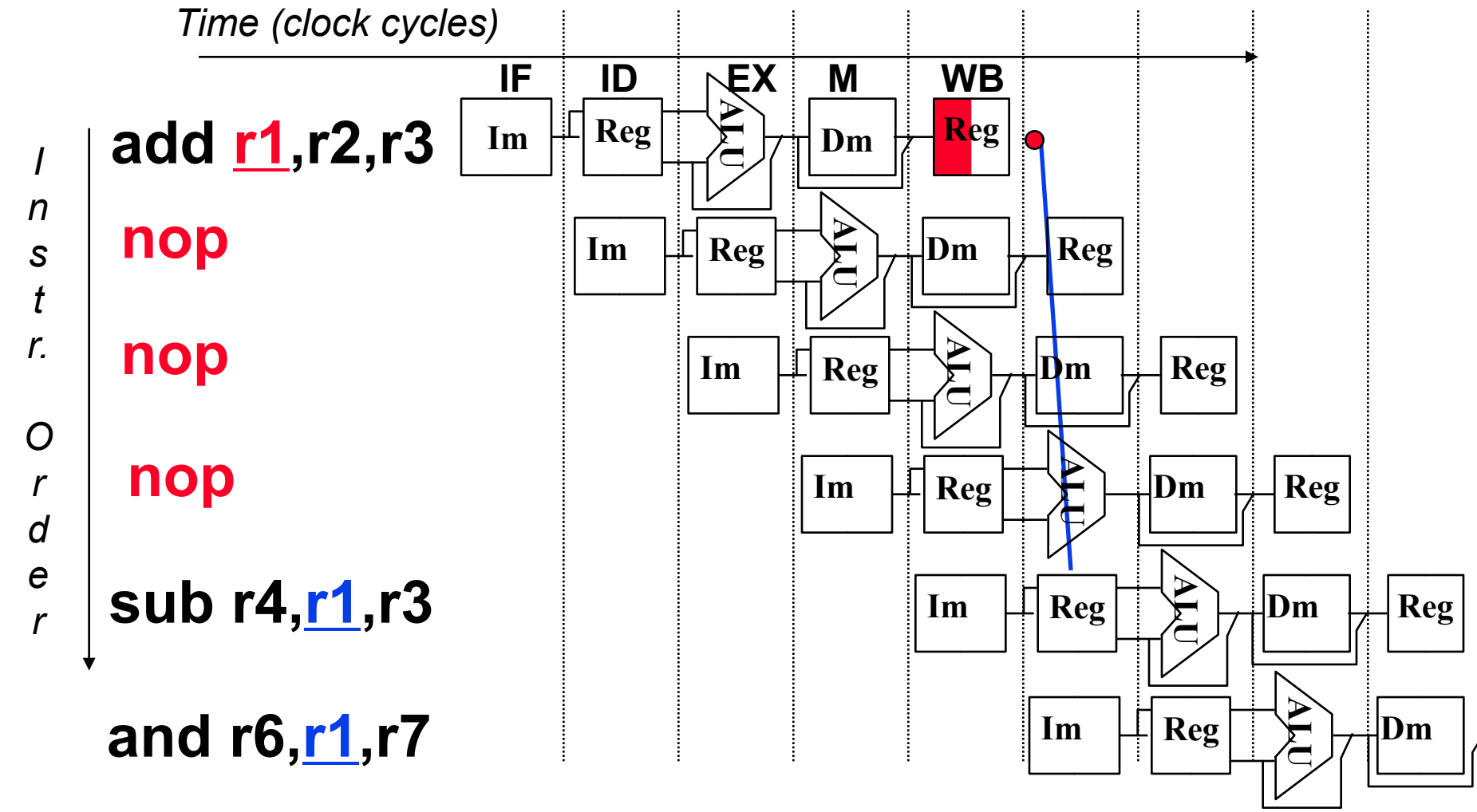
  这种做法称为流水线阻塞，也称为插入“气泡Bubble”



- 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。
  好处：数据通路简单，即无需改数据通路。



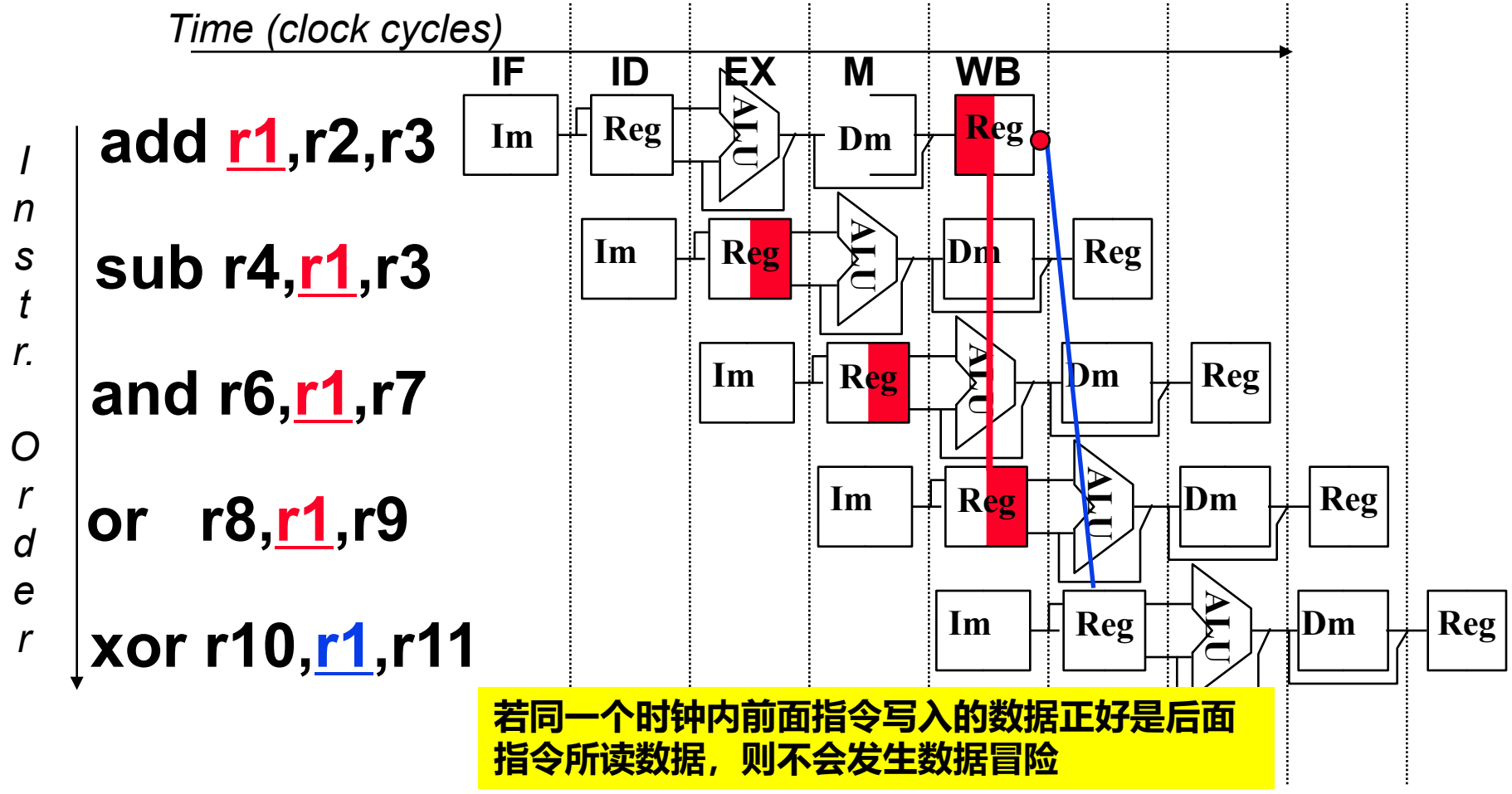
与方案1比，哪个更快?

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

前半后半

- 寄存器堆的读口和写口是相互独立的部件！



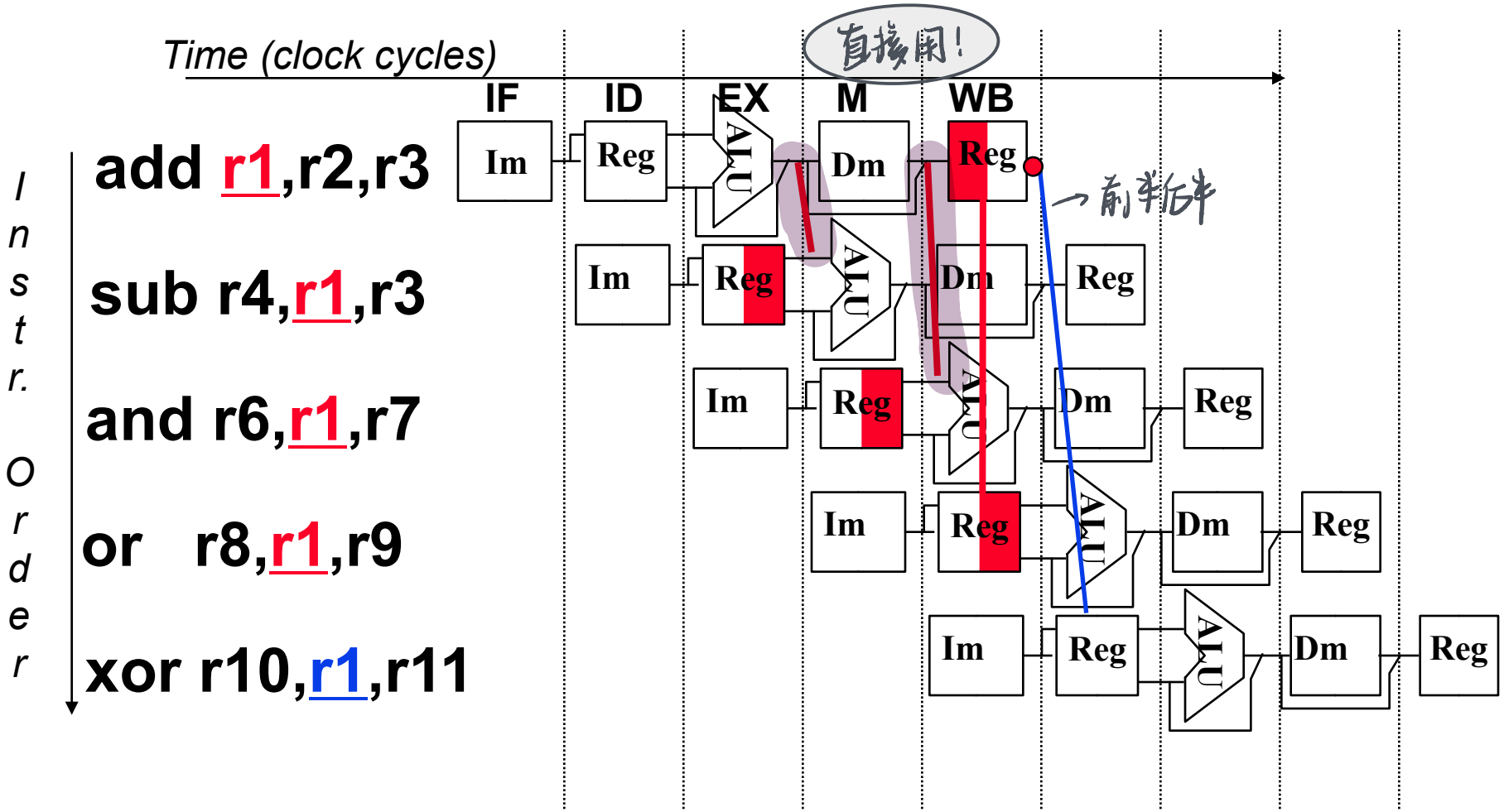
寄存器写口/读口分别在前/后半周期进行操作，使写入数据被直接读出

但是，只能解决部分数据冒险！

练习1.3、2.3 将三个空周期→两个空周期

方案4：利用DataPath中的中间数据：转发

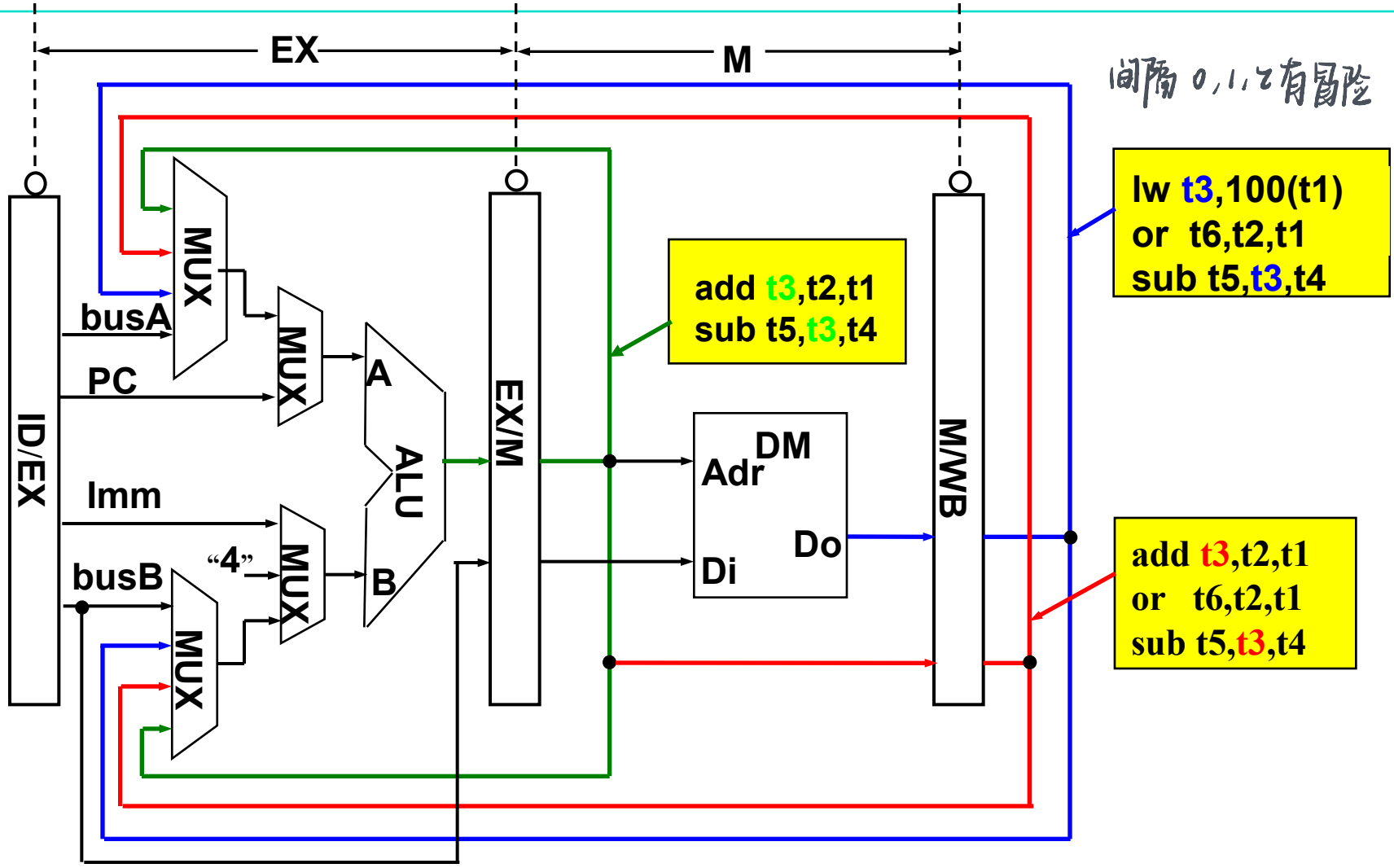
- 仔细观察后发现：流水段寄存器中已有需要的值r1！



把数据从流水段寄存器中直接取到ALU的输入端

称为转发（Forwading）
或旁路（Bypassing）

硬件上的改动以支持“转发”技术——四种情况



如果指令序列为

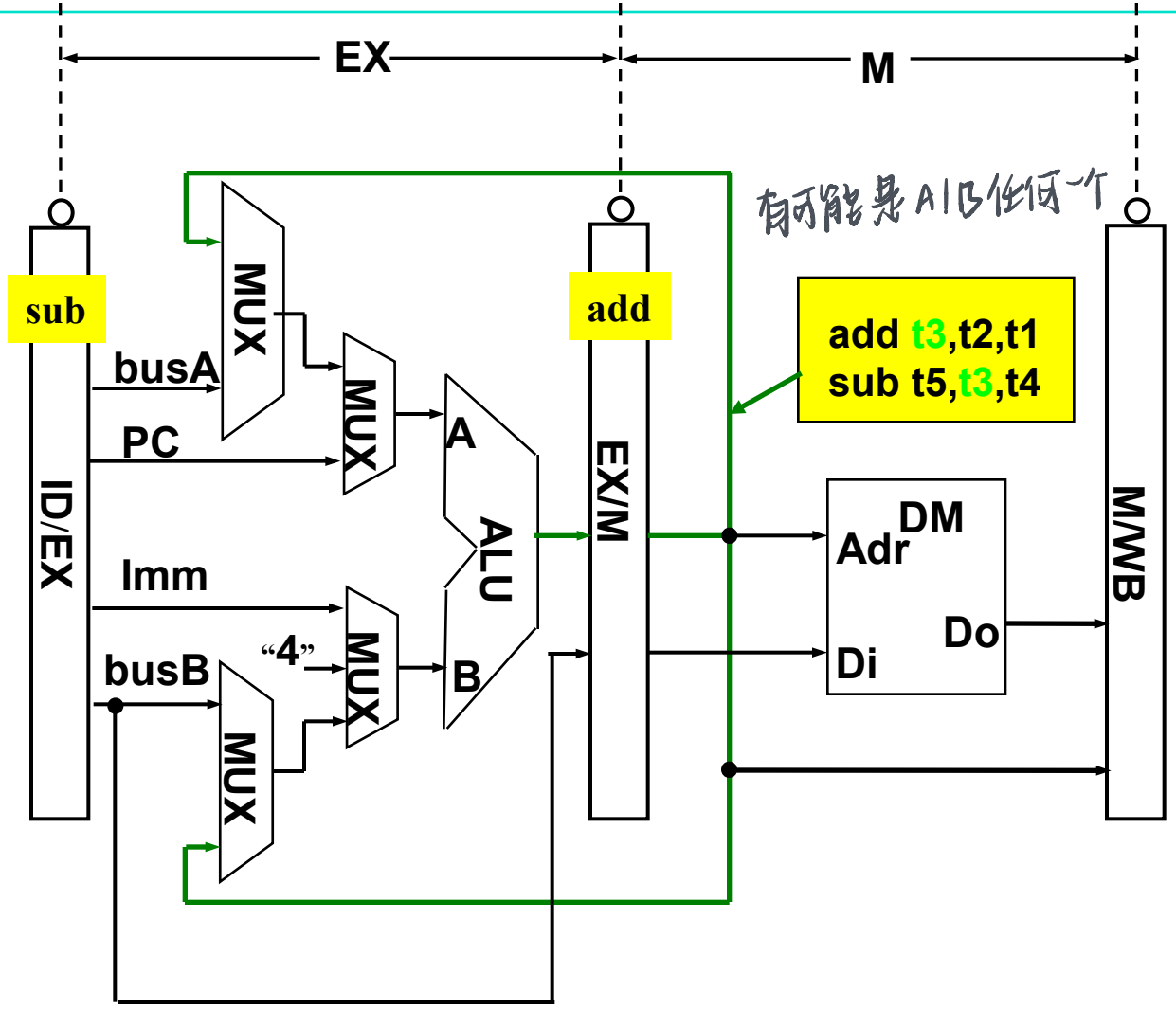
```
lw t3, 100(t1)
or t6, t3, t1
sub t5, t3, t4
```

能用“转发”技术解决第1、2两条指令间的数据冒险吗？

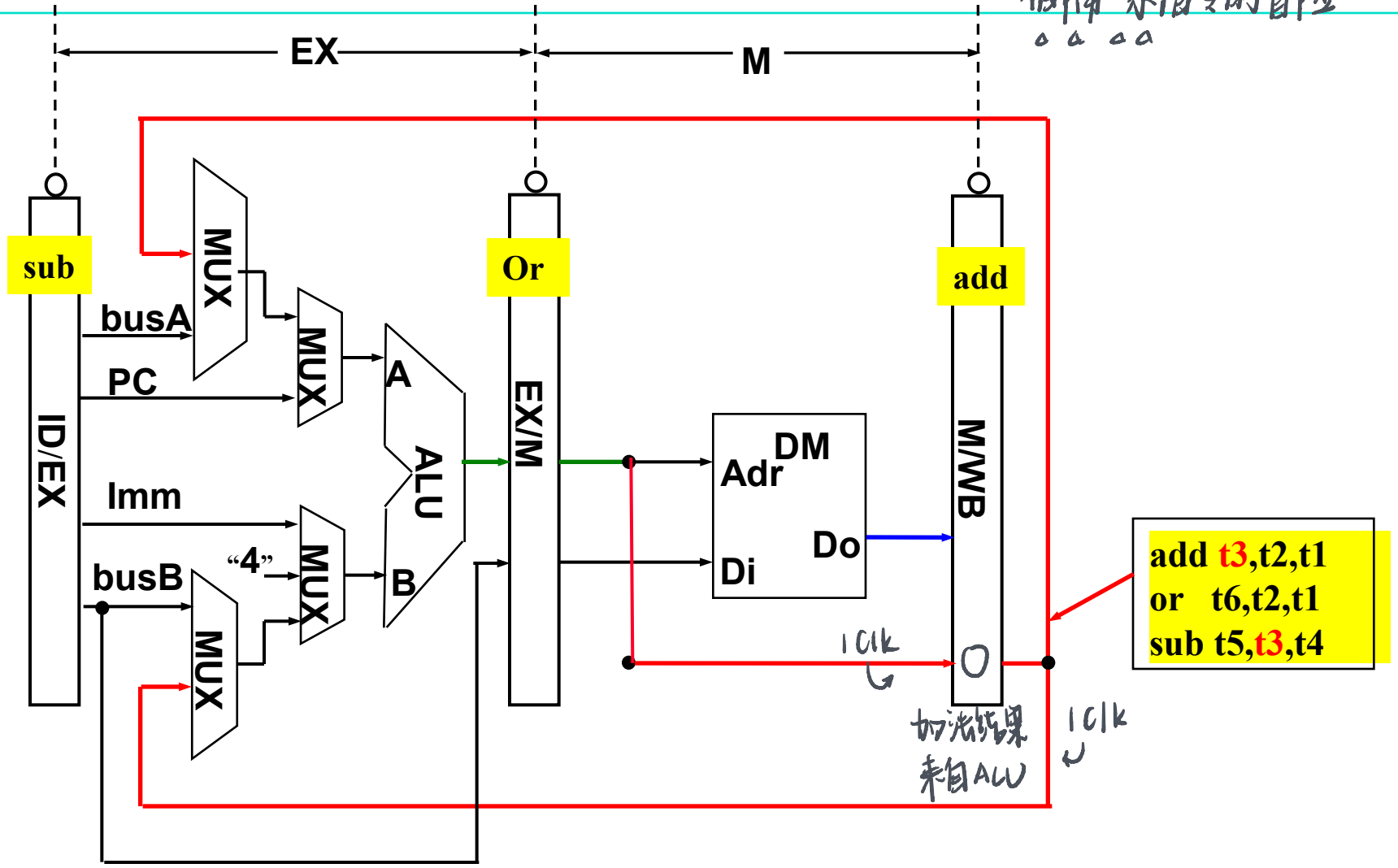
接下来展开分析。。

硬件上的改动以支持“转发”技术（续1）

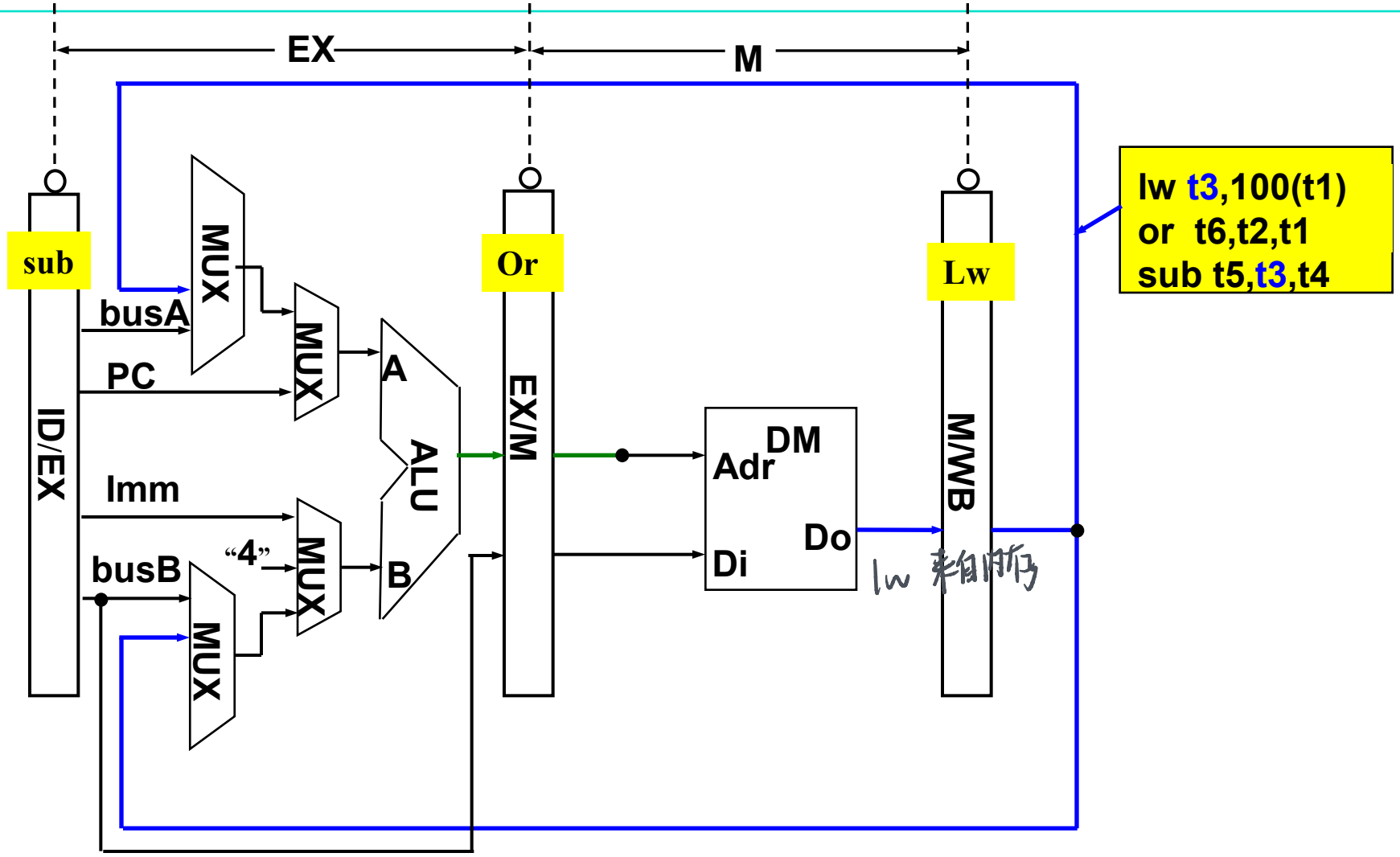
相邻两条指令的冒险



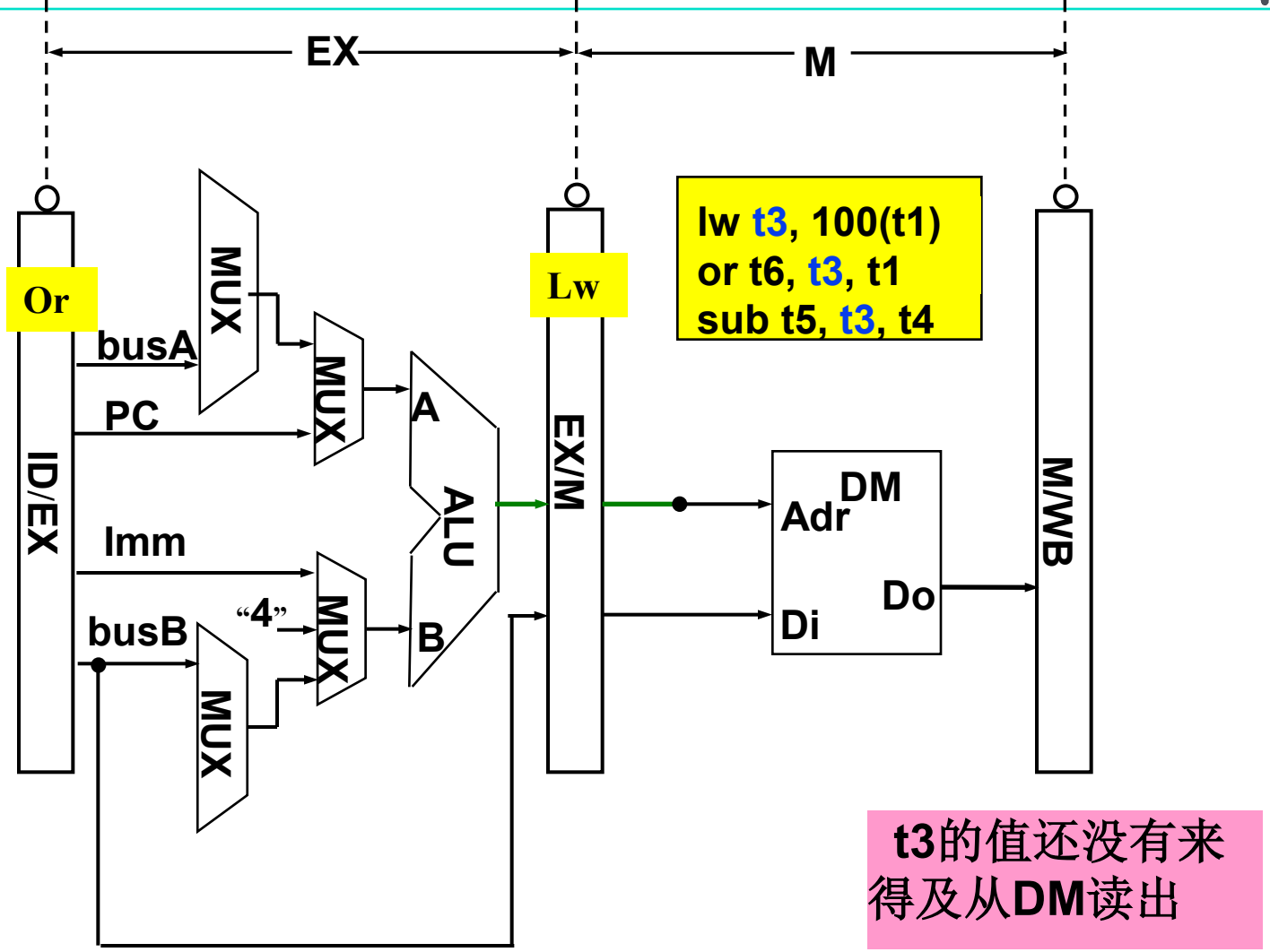
硬件上的改动以支持“转发”技术（续2）



硬件上的改动以支持“转发”技术（续3）

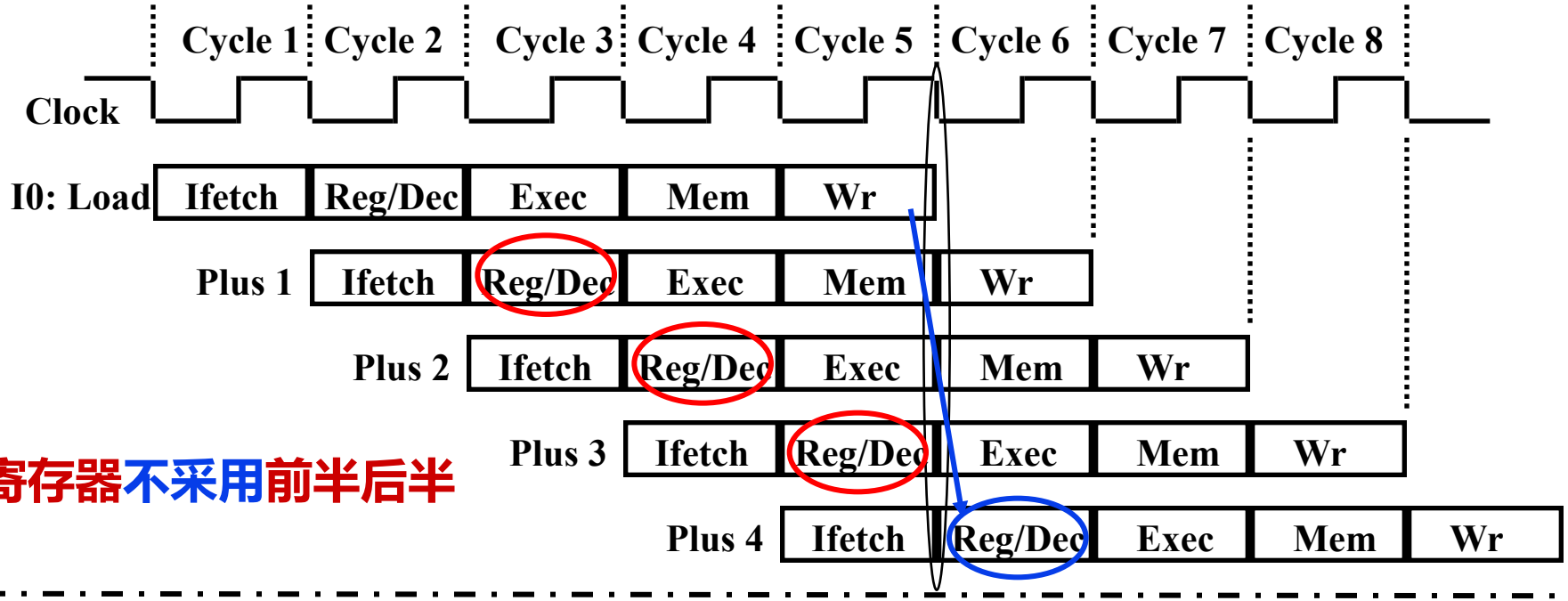


无法“转发”的情况（续4）——只和load有关

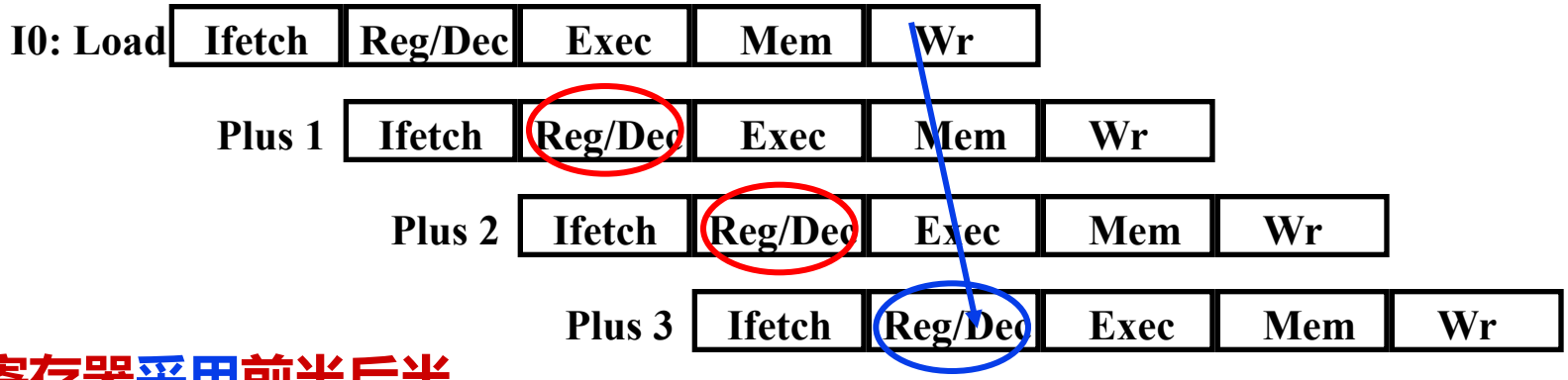


t3的值还没有来得及从DM读出

回顾: Load指令引起的延迟现象



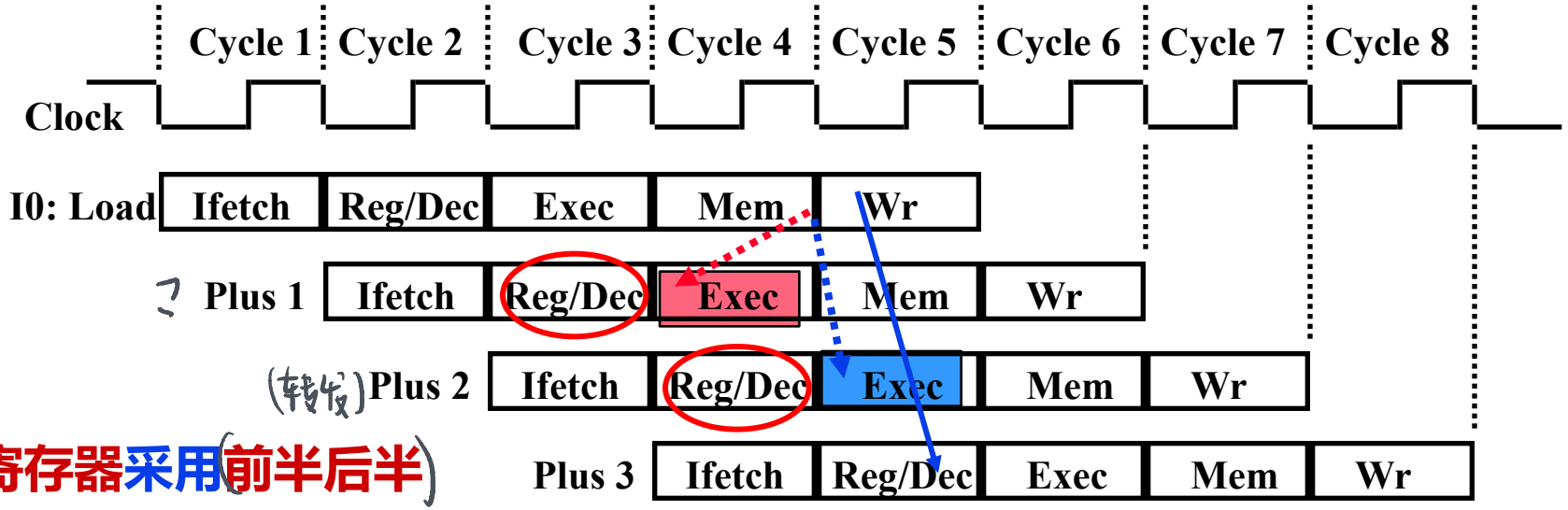
寄存器不采用前半后半



寄存器采用前半后半

若不采用转发，在何时才能使用Load指令的结果?

回顾: Load指令引起的延迟现象（续）



- 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?

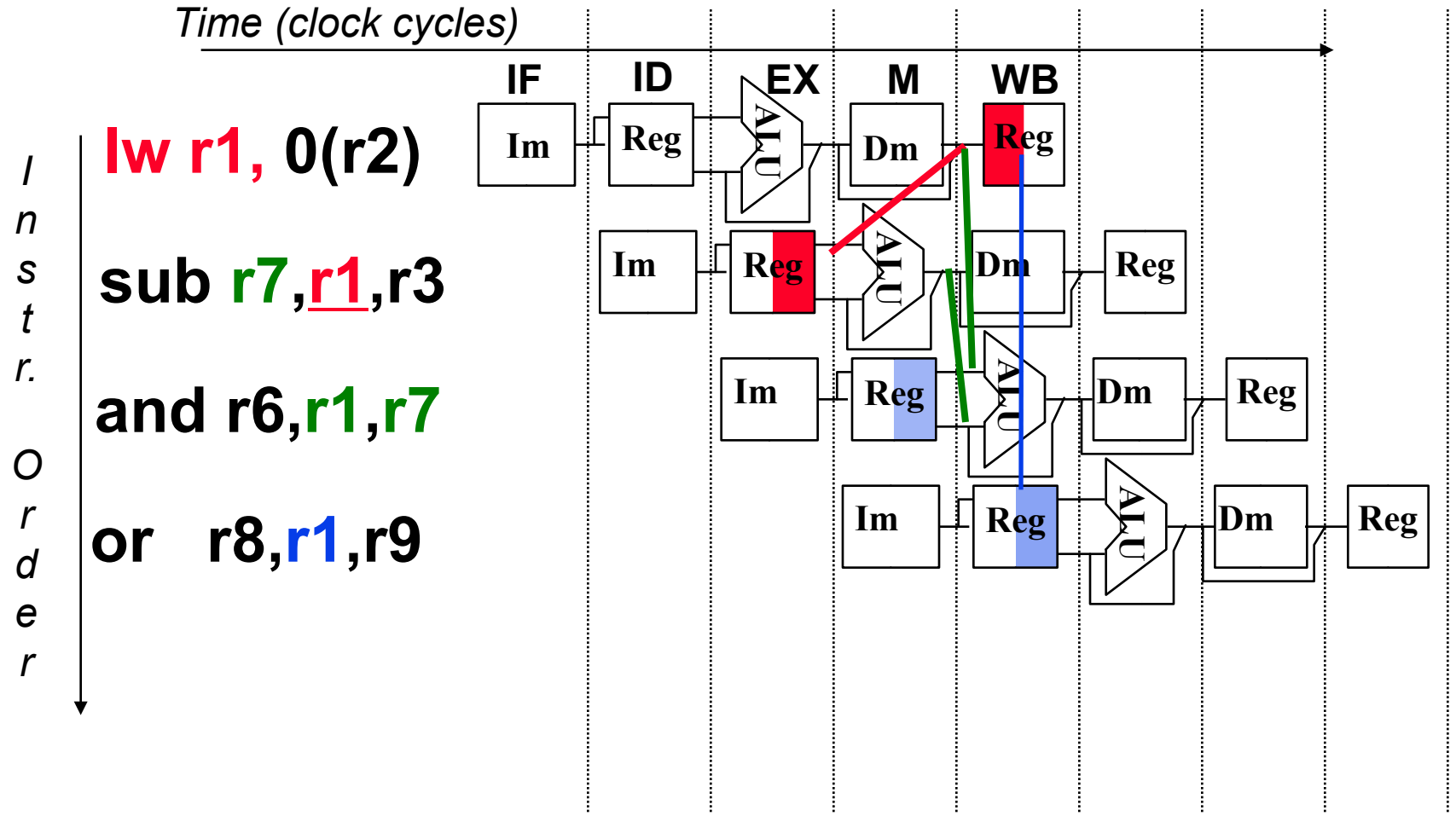
实际上，在第四周期结束时，数据在流水段寄存器中已经有值。

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

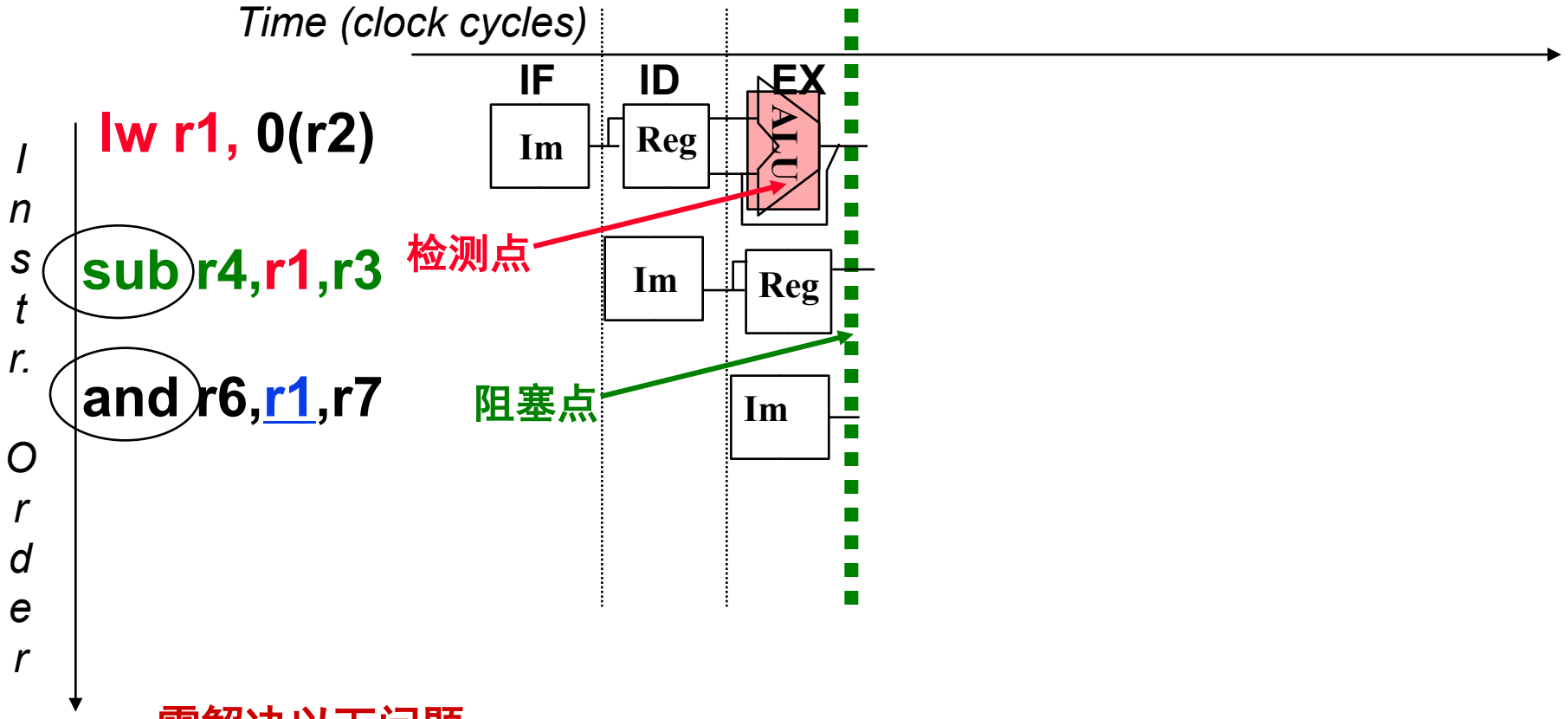


采用“转发”后仅第二条指令 SUB r7,r1,r3 不能按时执行!

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”

包括load-use数据冒险的解决方式——转发+ 阻塞



需解决以下问题：

(1) 判断什么条件下需要阻塞

(2) 修改数据通路来实现阻塞

ID/EX.MemRead

&& (ID/EX.Rd==IF/ID.Rs1

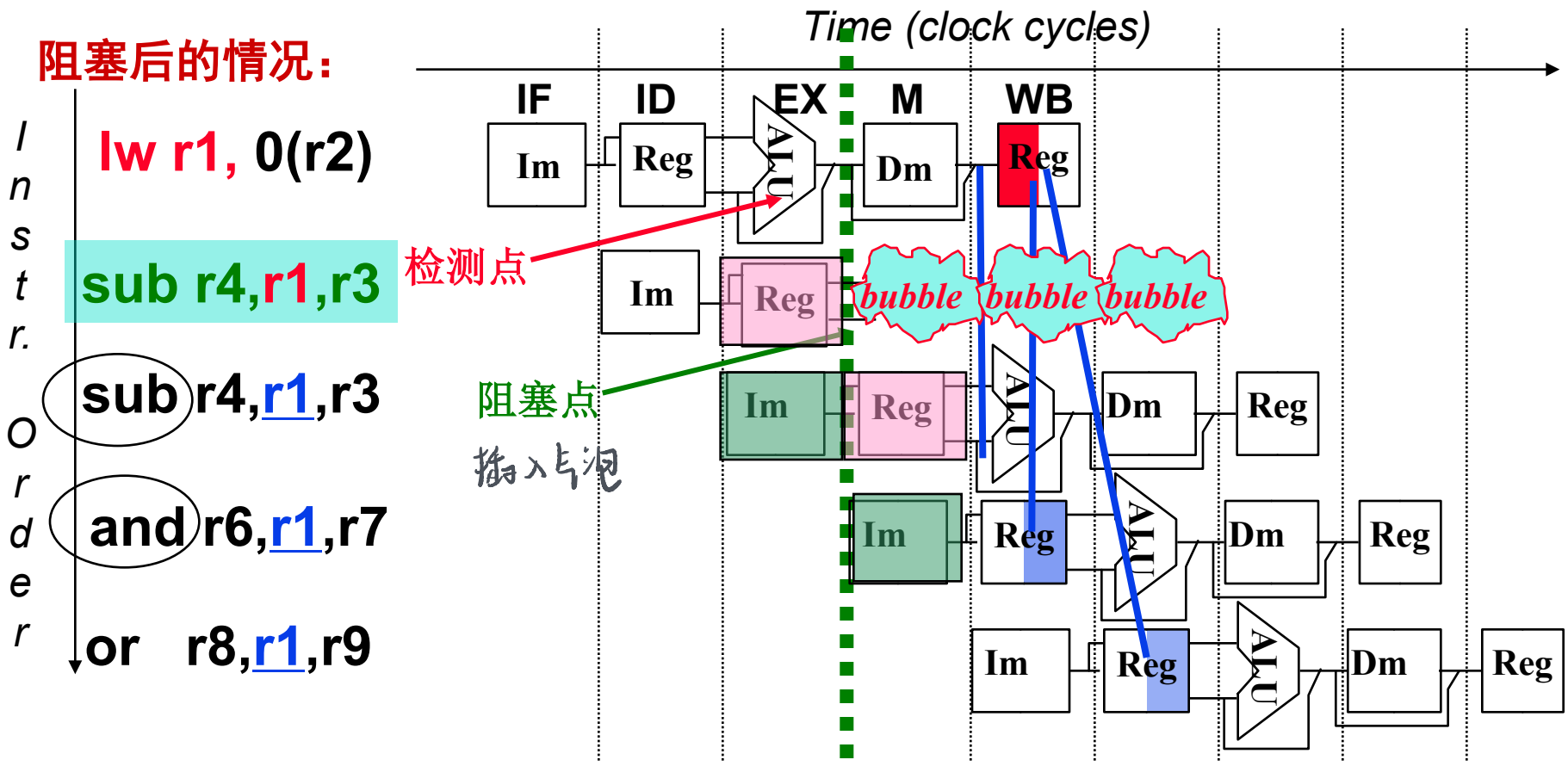
|| ID/EX.Rd==IF/ID.Rs2)

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

包括load-use数据冒险的解决方式——转发+ 阻塞

阻塞后的情况：



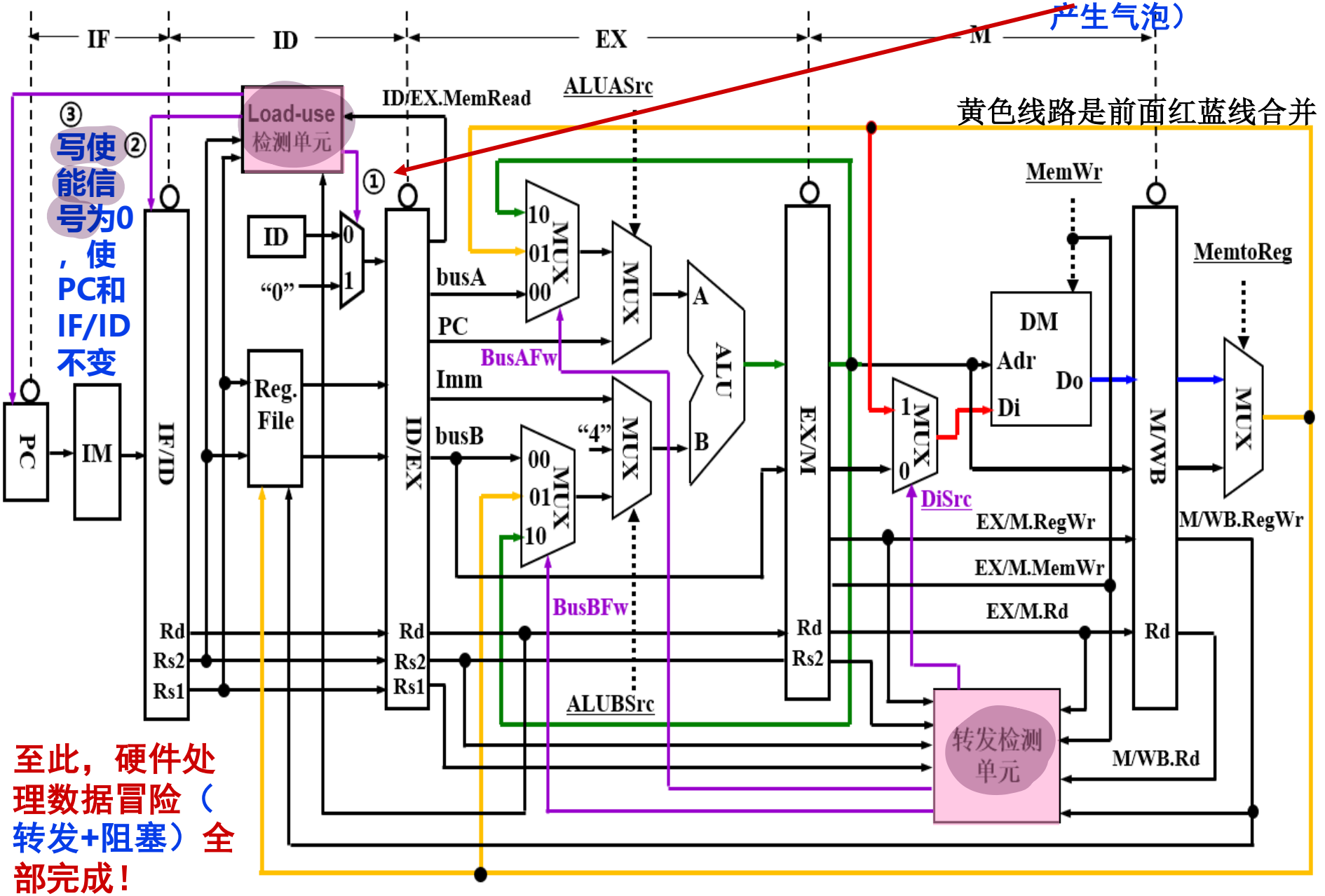
在阻塞点，将load后面两条指令的执行结果清除，并延迟一个周期执行

① 将ID/EX段寄存器中所有控制信号清0，以插入一个“气泡”

② IF/ID寄存器中的信息不变（还是sub指令），sub指令将重新译码执行

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



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！

优化调度后load阻塞现象大约可以降低1/2~1/3

Slow code:（需阻塞2个时钟）

```
lw    t2, b
lw    t3, c
add   t1, t2, t3
sw    t1, a
lw    t5, e
lw    t6, f
sub   t4, t5, t6
sw    t4, d
```

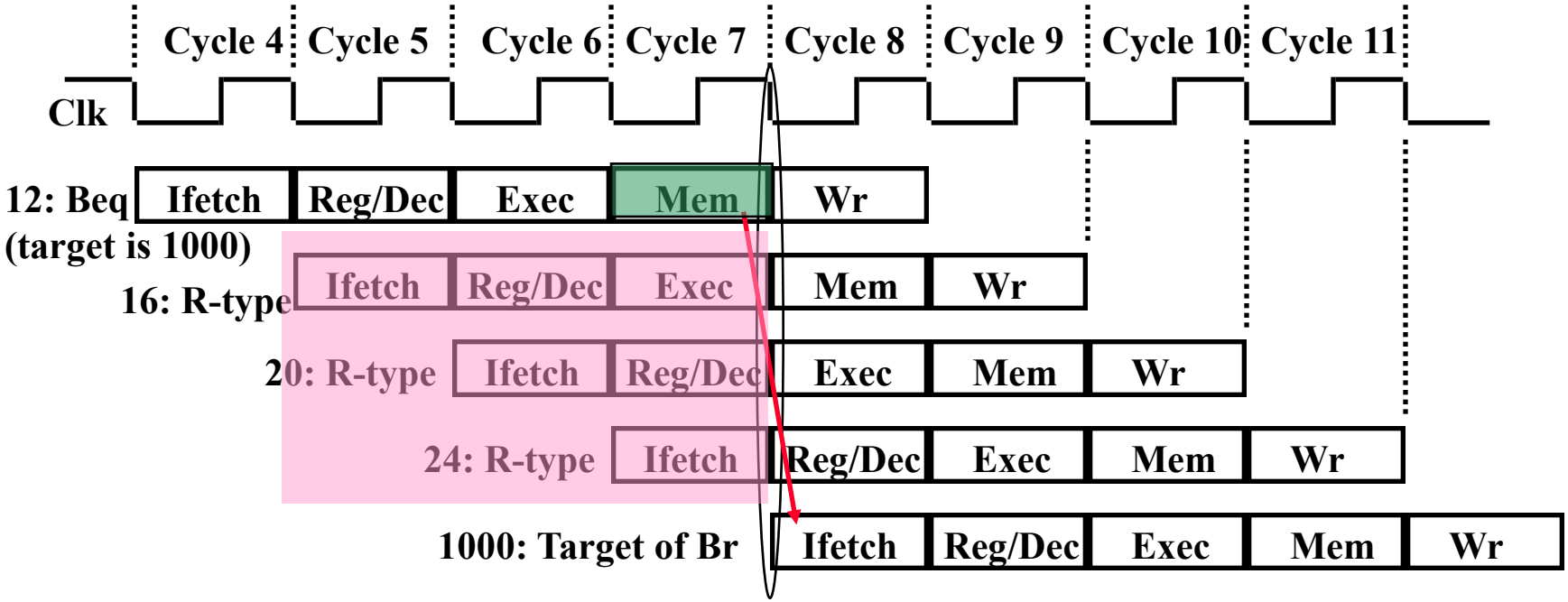
调整后

Fast code:（无需阻塞）

```
lw    t2, b
lw    t3, c
lw    t5, e
add   t1, t2, t3
lw    t6, f
sw    t1, a
sub   t4, t5, t6
sw    t4, 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！

控制冒险 现象——延迟损失时间片C



- 虽然Beq指令在第四周期取出，但:
  - “是否转移”在M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EX 、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

这里 C=3

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
- 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：不跳转
  - 动态预测：一般用于循环
    - 根据程序执行的历史情况进行动态预测调整

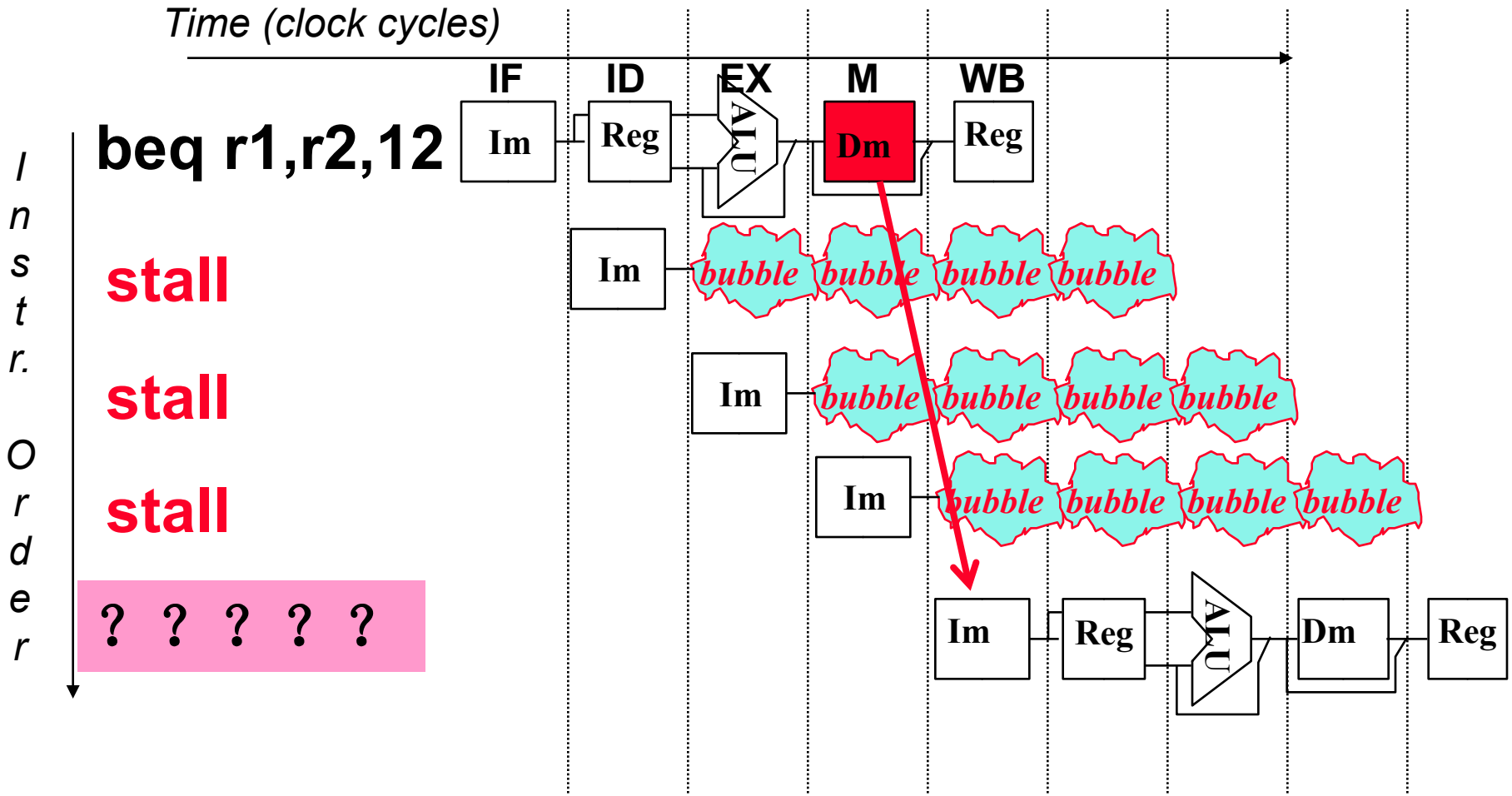
  注：流水线控制必须确保被错误预测指令的执行结果不能生效，而且要能从正确的分支地址处重新启动流水线工作

- 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

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，延迟到分支指令能够确定是否转移以后！——使接下来三条指令清0或 其操作信号清0，以插入气泡。

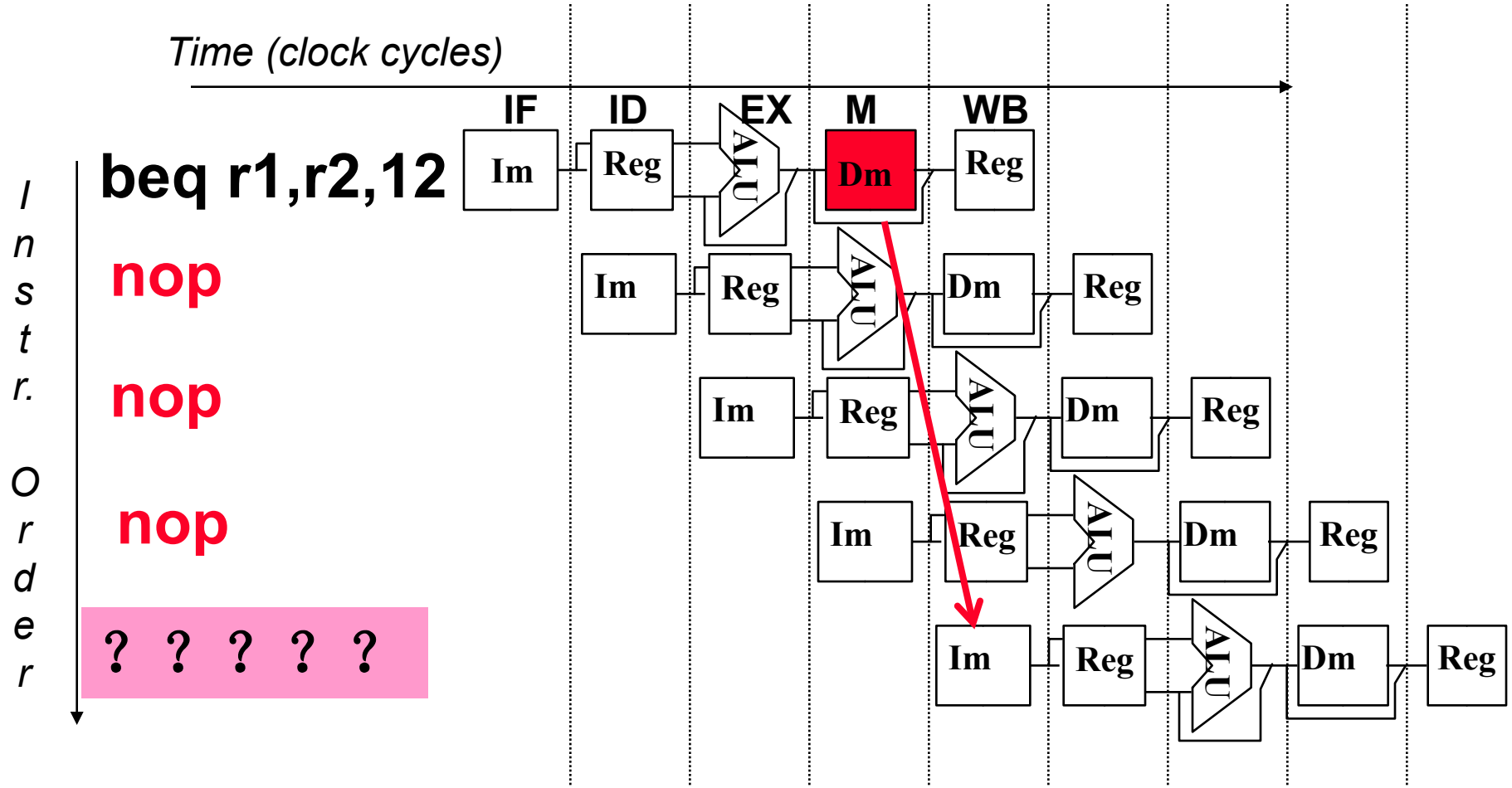


• 缺点：控制比较复杂，需要改数据通路；指令被延迟三个时钟执行。

方案 2: 软件上插入无关指令

- 由编译器插入三条NOP指令，延迟到分支指令能够确定是否转移有新值以后再执行后续有效的指令！浪费三条指令的空间和时间。

  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！

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：不跳转

    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
  - 预测失败时，需把流水线中三条错误预测指令（C=3）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（C=3，2，1？）
  - 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，而不等到MEM阶段才确定?

    可以！ ID

    那最早可以提前到哪个阶段呢?

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M阶段移到ID阶段，可以吗？为什么？

      （可：IF/ID流水段寄存器中已有PC的值，ID段已有扩展后立即数Imm）
    - 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)直接比较Rs1和Rs2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）
  - 如果原来预测不转移
  - 但是发现Branch=1并且Zero=1时，则beq预测失败
  - 此时需要完成以下两件事（延迟损失时间片C=1时）：

    ① 将转移目标地址->PC

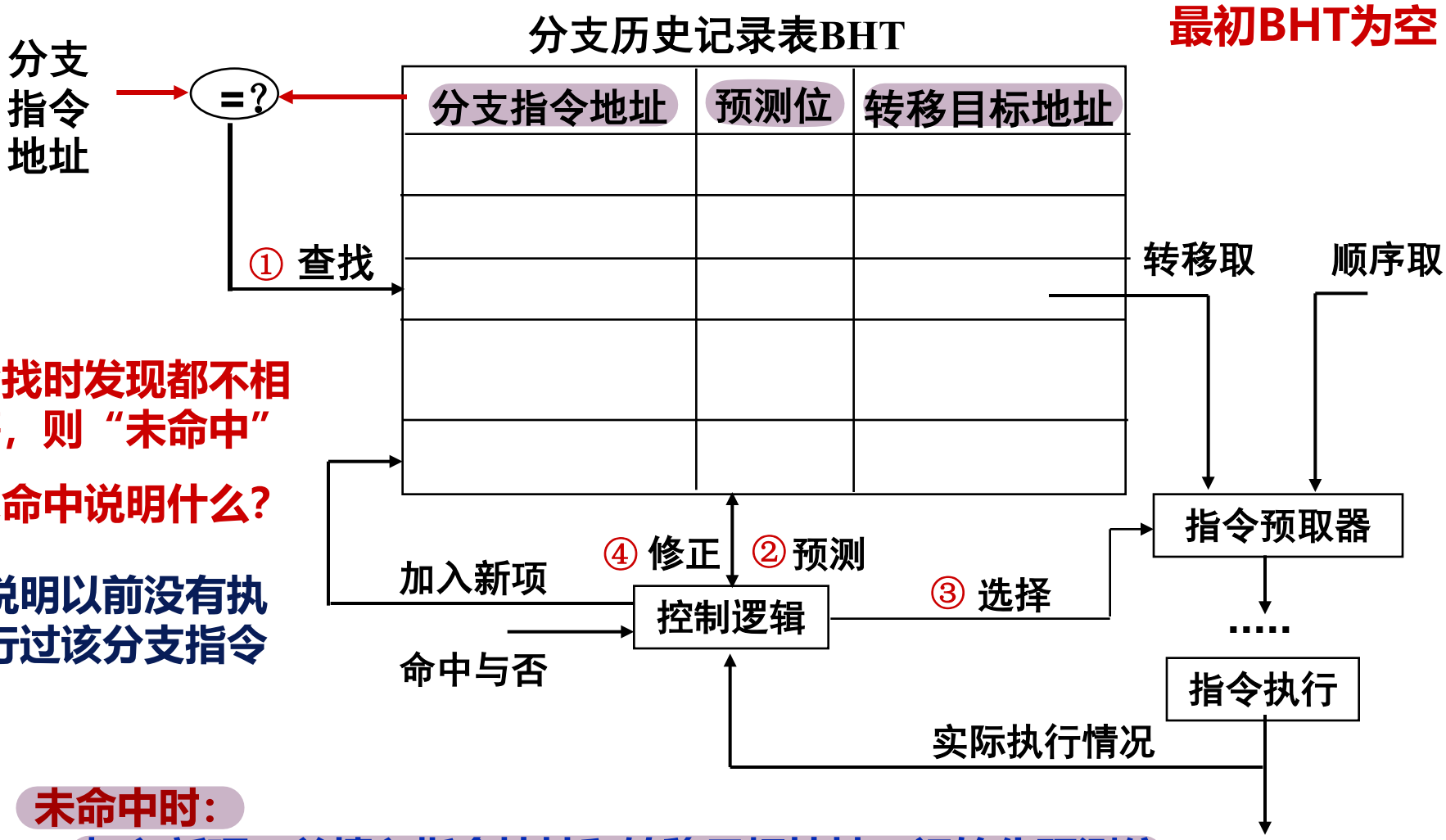
    ② 清除IF段中取出的指令，即：将IF/ID中的指令字清0，转变为nop指令

原来要清除三条指令（C=3），调整后只需要清除一条指令（C=1），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！

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）

分支历史记录表BHT（或BTB、BPB）



未命中时：

加入新项，并填入指令地址和转移目标地址、初始化预测位

命中时：

根据预测位，选择“转移取”还是“顺序取”

动态预测基本方法

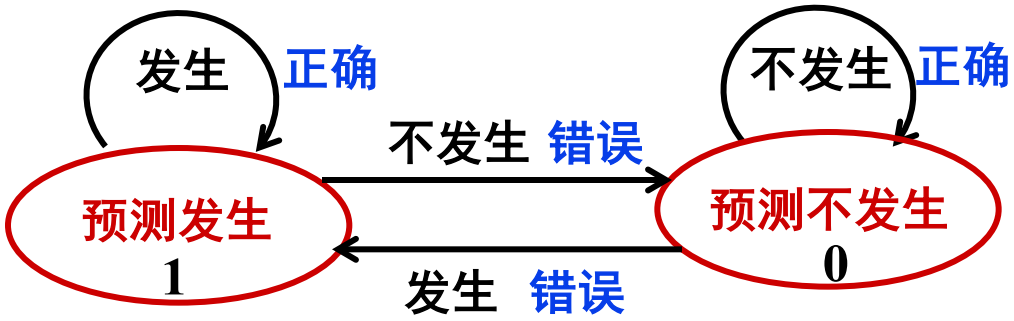
- 按照预测状态图进行预测和调整

- 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）
  - 缺点：当连续两次的分支情况发生改变时，预测错误

- 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
  - 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 ~ t1, t2, t3, t4 and base address of array is in t5

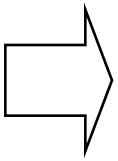
- 指令预取时，按照预测位读取相应分支的指令
  - 预测发生时，选择“转移取”
  - 预测不发生时，选择“顺序取”
- 指令执行时，按实际执行结果修改预测位
  - 对照状态转换图来进行修改
  - 例如：对于一个循环分支
    - 若初始状态为0(再次循环时为0)，则第一次和最后一次都错
    - 若初始状态为1，则只有最后一次会错。（再次循环时又改为0，还是有两次错）

```
Loop: add t7, t3, t3        ; i*2
      add t7, t7, t7        ; i*4
      add t7, t7, t5
      lw t6, 0(t7)          ; t6=A[i]
      add t1, t1, t6        ; g= g+A[i]
      add t3, t3, t4
      bne t3, t2, Loop
        … …
```

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;
}
```



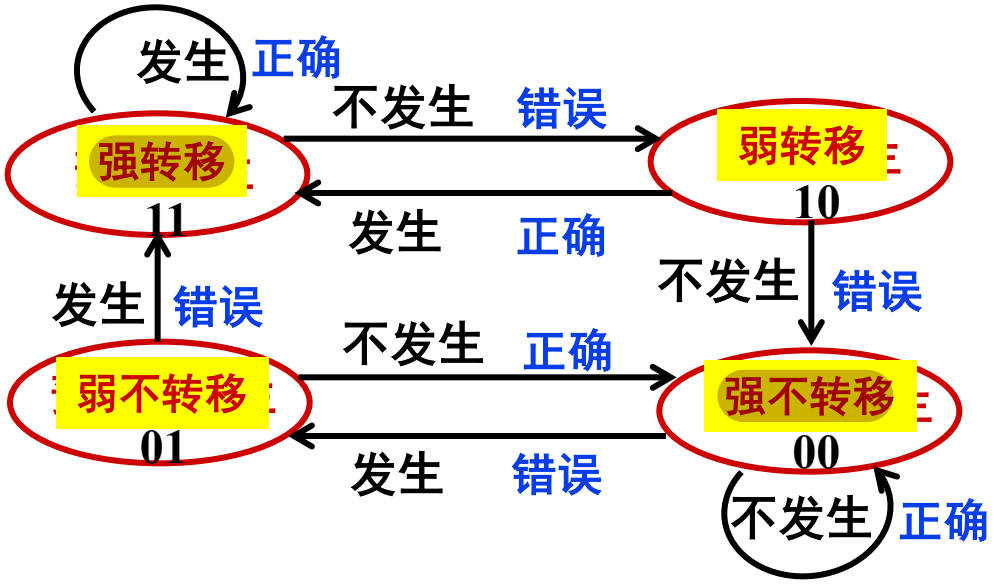
```
… …
Loop-i:  beq t1,a0, exit-i    # 若( i=N)则跳出外循环
         add t2, zero, zero   #j=0
Loop-j:  beq t2, a0, exit-j   # 若(j=N)则跳出内循环
         addi t2, t2, 1       # j=j+1
         addi t0, t0, 1       #sum=sum+1
         j  Loop-j
exit-j:  addi t1, t1, 1       # i=i +1
         j  Loop-i
exit-i:  …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

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

N=10, 准确率分别90.9%和82.7%
N=100,准确率分别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×(N–1)次预测错误（第一次内循环有1次预测错，后面(N-1)次内循环每次有2次预测错）。N越大预测准确率越高！

两位预测状态图

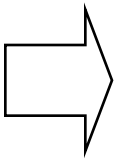


预测发生时，选择“转移取”
预测不发生时，选择“顺序取”

- 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

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1 ×(N+1)次，
内循环中的分支指令共执行N×(N+1)次。

N=10, 准确率分别90.9%和90.9%
N=100,准确率分别99%和99%

若预测位初始为00（强不转移），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（弱不转移），再进入内循环时，第一次预测正确，且预测位再次变为00，如此循环，最终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！

方案4： 延迟分支（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操作填充

举例：如何对以下程序段进行分支延迟调度？

（假定时间片C=2）

```
lw $1, 0($2)
lw $3, 0($2)
add  $6, $4, $2
beq $3, $5, 4
add $3, $3,$2
sw $1, 0($2)
……
```

调度后

```
lw $3, 0($2)
add  $6, $4, $2
beq $3, $5, 8
lw $1, 0($2)
nop
add $3, $3,$2
sw $1, 0($2)
```

若C=1，则可以：

```
lw $3, 0($2)
add  $6, $4, $2
beq $3, $5, 6
lw $1, 0($2)
add $3, $3,$2
sw $1, 0($2)
……
```

调度后可能带来其他问题：产生新的load-use数据冒险

调度后，降低了分支延迟损失

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
  - 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
    - ✓ 将异常/中断处理程序首地址送PC

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%；无条件跳转指令的更新地址工作也在ID段完成。不考虑流水段寄存器延时，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。

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	0	550 ps
Branch	200	50	100	0	0	350 ps
Jump	200	0	100	0	50	350 ps

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(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=5x25%+4x10%+4x52%+3x11%+3x2%=4.12

所以， N条指令的执行时间为4.12x200xN=824N(ps)

三种处理器实现方式的比较

注意：这里的CPI并非是一条指令实际执行总时长了

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：.75x1+.25x2=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：1.5x25%+1x10%+1x52%+1.25x11%+2x2%=1.17

所以， N条指令的执行时间为1.17x200xN=234N(ps)

第六讲小结

- 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
  - 动态分支预测技术
  - 缩短分支延迟技术
- 异常和中断是一种特殊的控制冒险

全课程终

谢谢大家