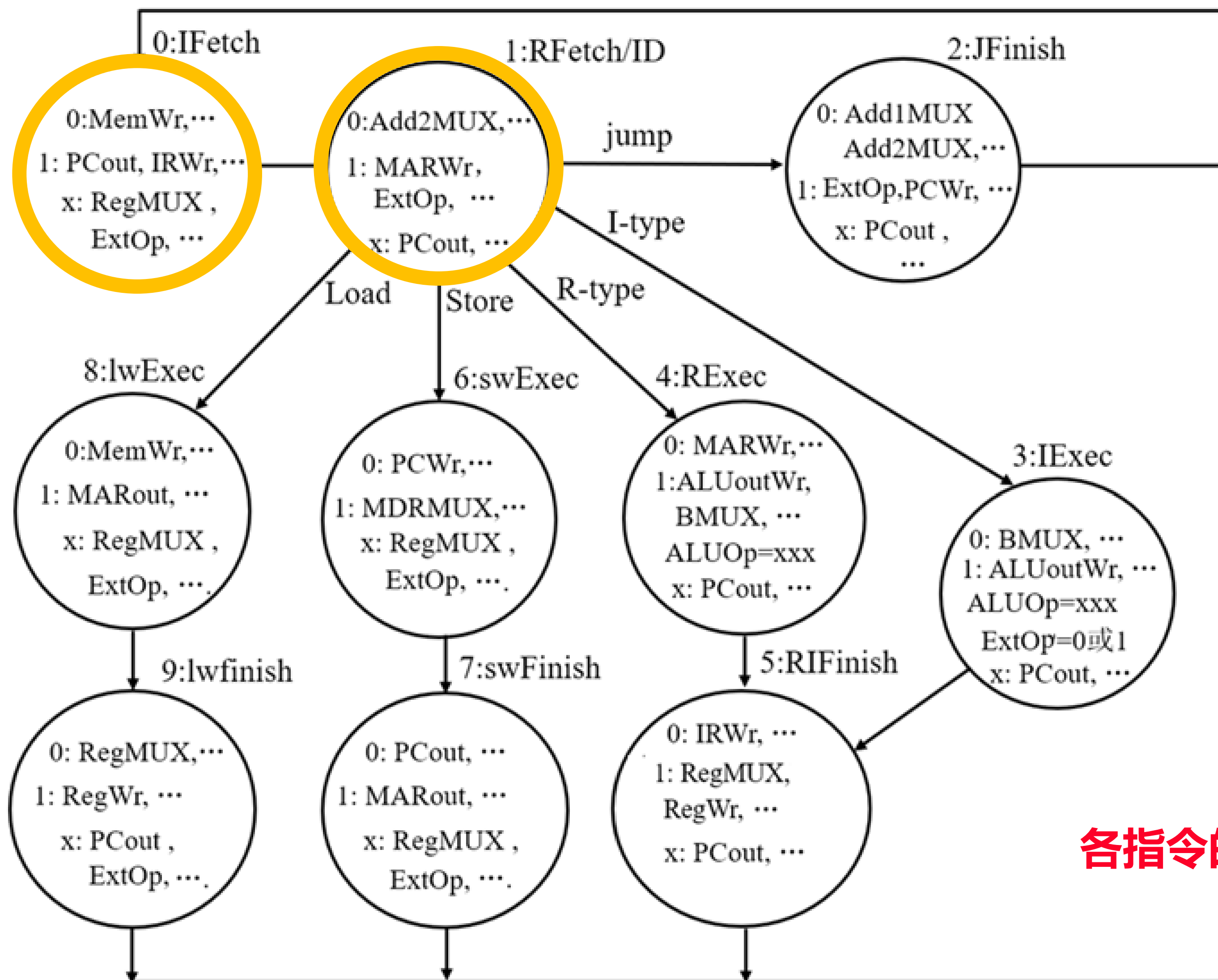
回顾第22次课



每来一个时钟，进入下一个状态

各指令的时钟数多少?

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 $_i$ x CPI $_i$ ）/ 指令数

  = Σ（指令数 $_i$ / 指令数 ）x CPI $_i$

x1=1

这

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。

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

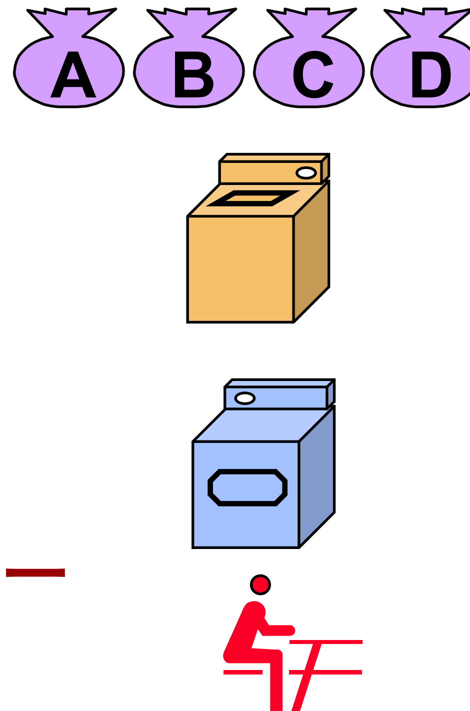
第七讲 高级流水线技术

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

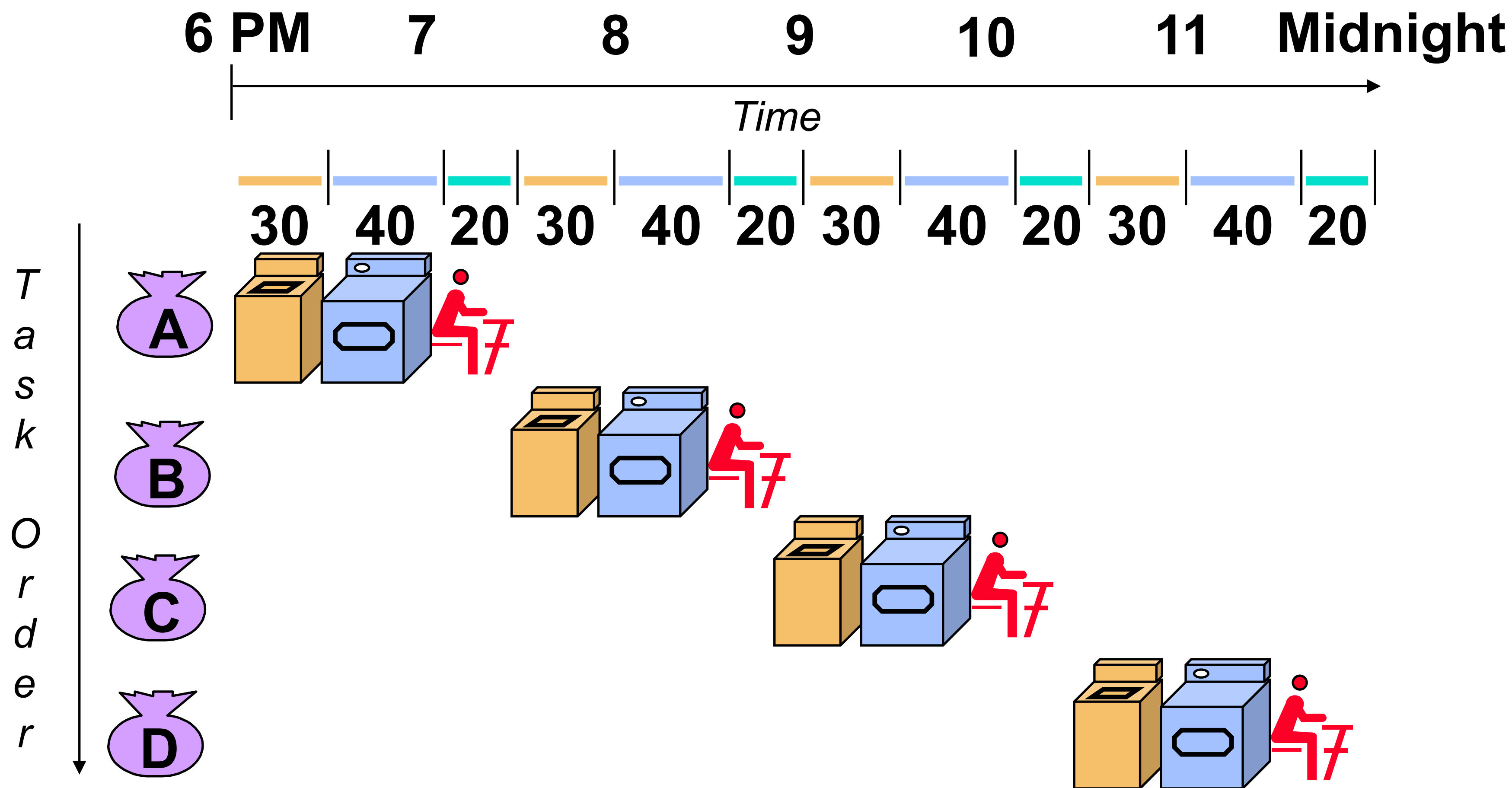


洗衣店只有一台洗衣机，一台干衣机，一个负责folder的员工。

如果让你来管理洗衣店，你会如何安排？

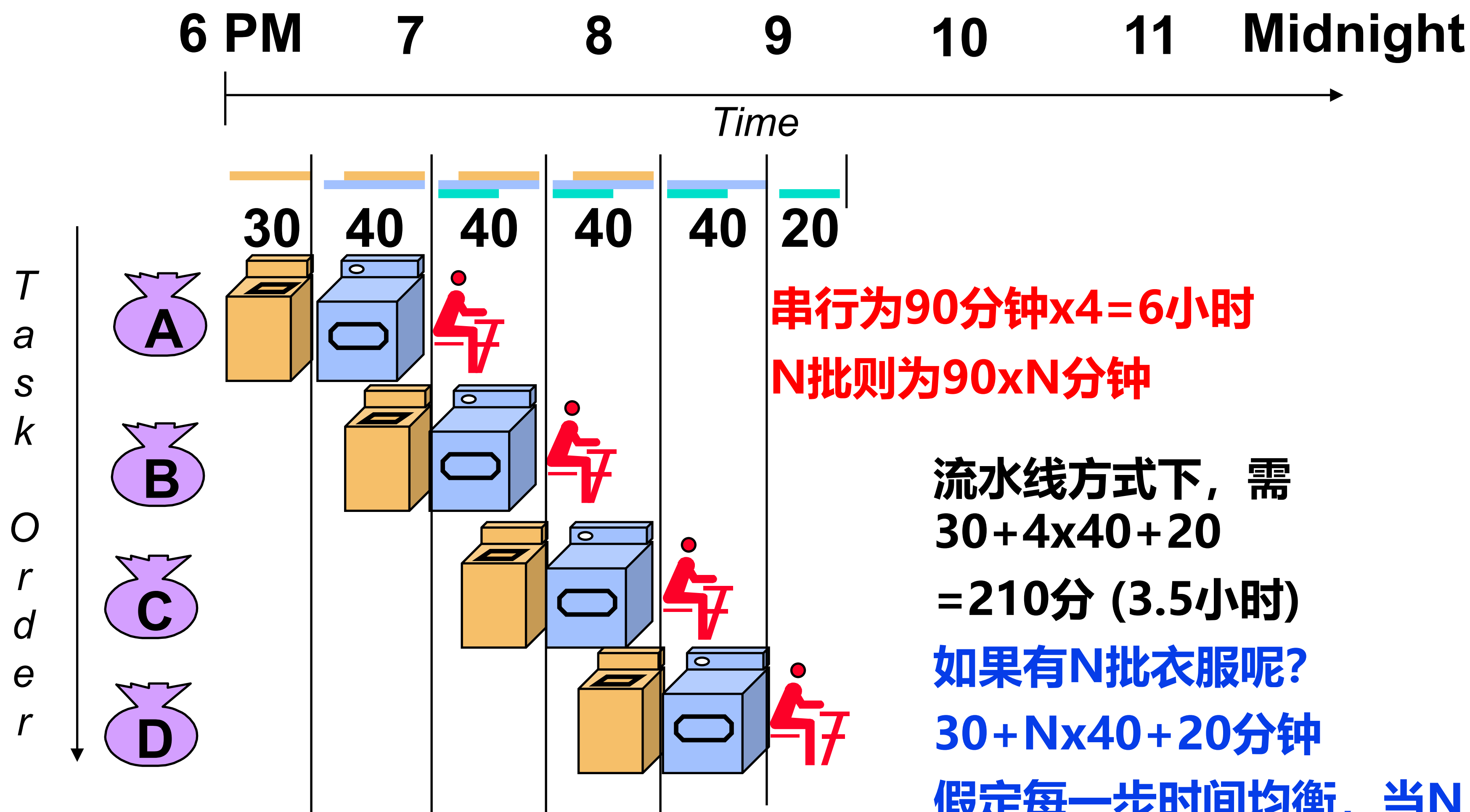
Pipelining: It's Natural !

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

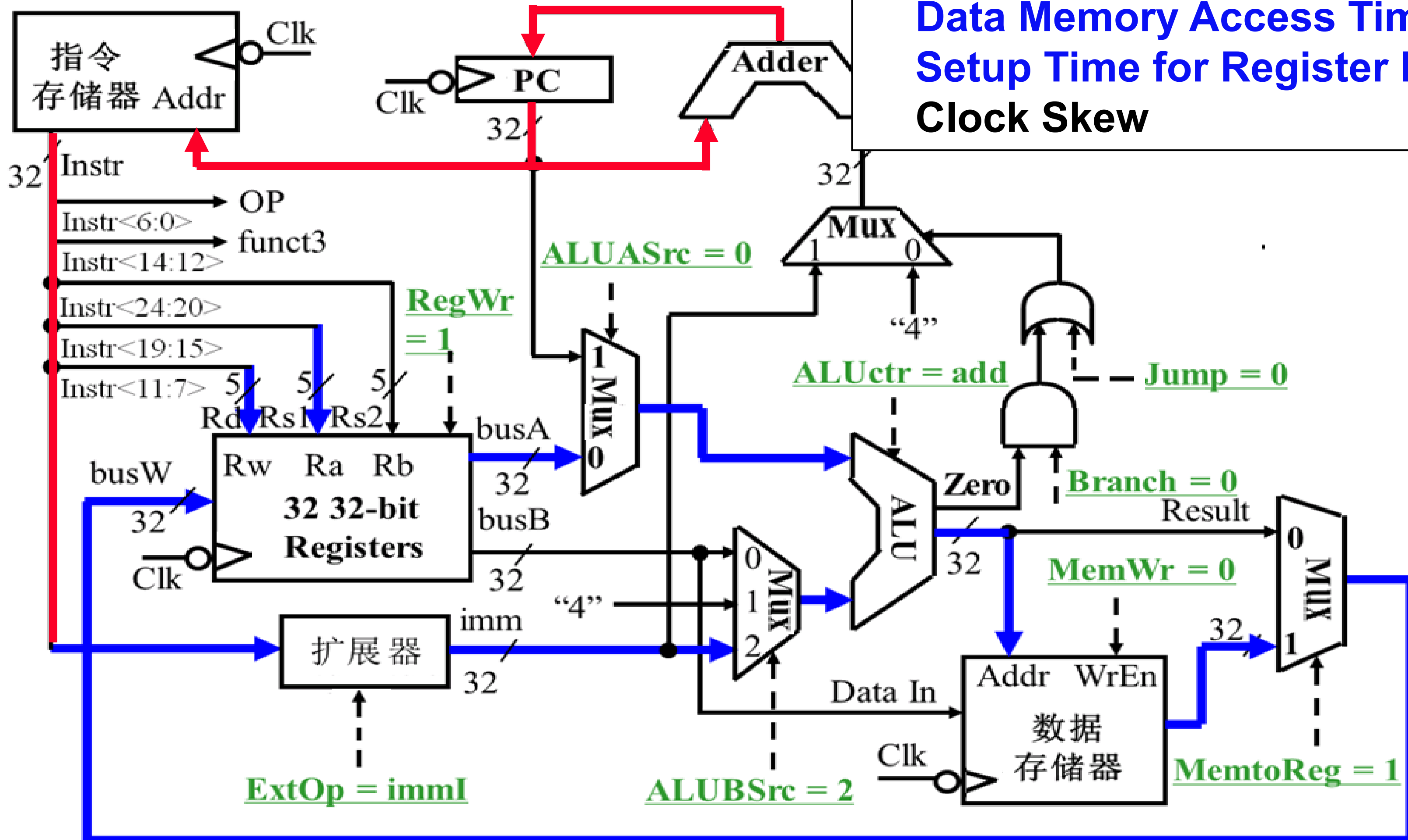
假定每一步时间均衡，当N很大的时候，流水线比串行方式提高约3倍!

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) =
- **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**



分析：Load指令的5个阶段

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

- Ifetch (取指) : 取指令并计算PC+4 (用到哪些部件？)

  指令存储器、Adder

- Reg/Dec (取数和译码) : 取数同时译码 (用到哪些部件？)

  寄存器堆读口、指令译码器

- Exec (执行) : 计算内存单元地址 (用到哪些部件？)

  扩展器、ALU

- Mem (读存储器) : 从数据存储器中读 (用到哪些部件？)

  数据存储器

- Wr(写寄存器): 将数据写到寄存器中 (用到哪些部件？)

  寄存器堆写口

这里寄存器堆的读口和写口可看成两个不同的部件。

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

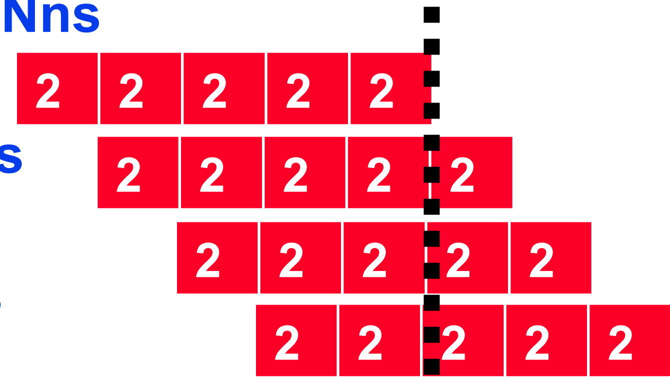
指令的执行过程是否和“洗衣”过程类似？
是否可以采用类似方式来执行指令呢？

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

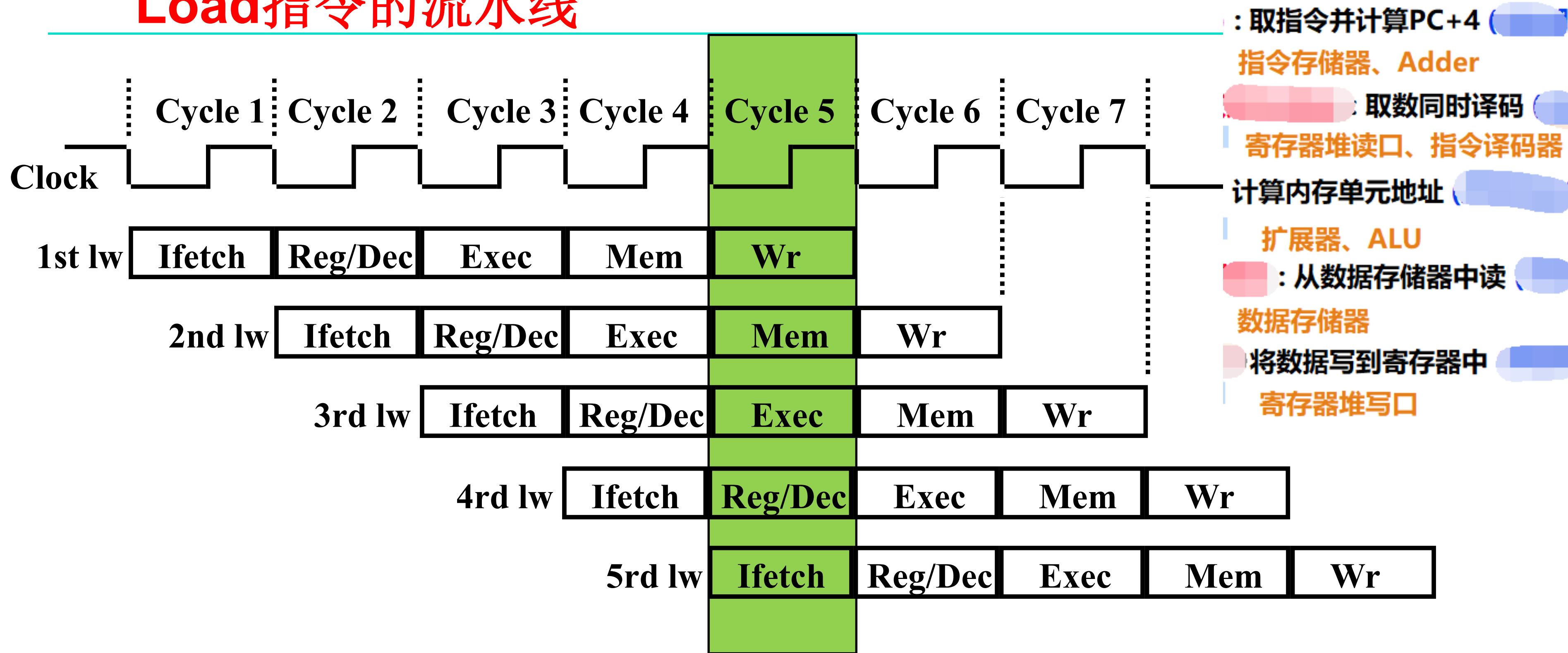
  Load指令执行时间总计为：8ns

  (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



- 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

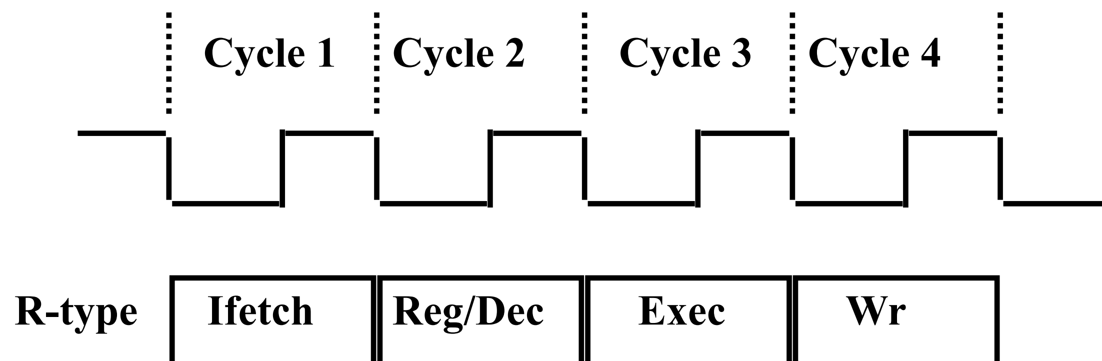
若位置随指令不同而不同，则需先确定指令类型才能取寄存器编号

  - 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

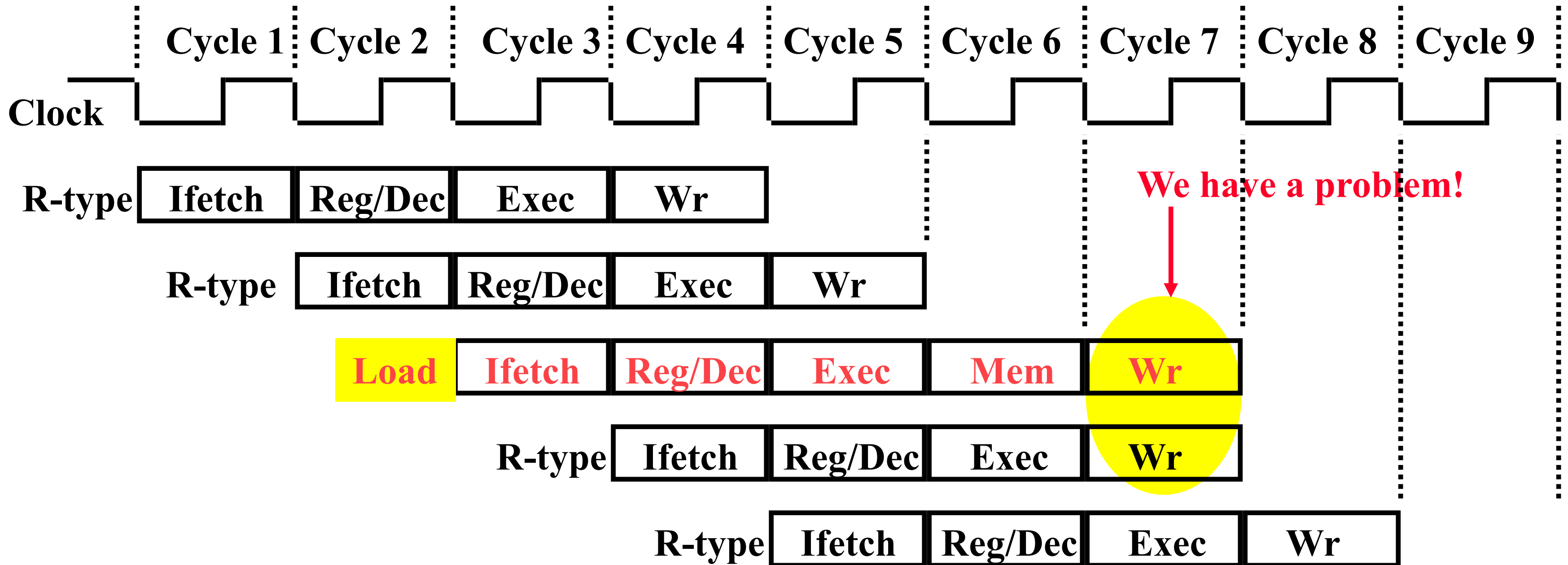
R-type指令的4个阶段

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

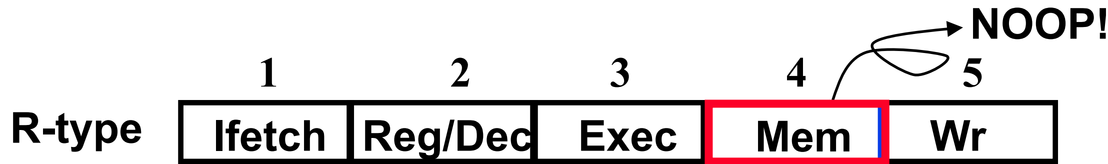


- 上述流水线有个问题: 两条指令试图同时写寄存器，因为
  - Load在第5阶段用寄存器写口
  - R-type在第4 阶段用寄存器写口

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



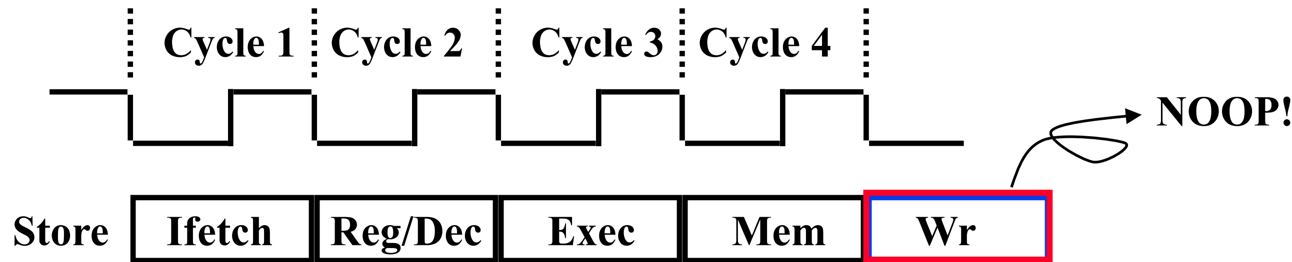
- 加一个NOP阶段以延迟“写”操作:
  - 把“写”操作安排在第5阶段, 这样使R-Type的Mem阶段为空NOP

Clock	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9
R-type	Ifetch	Reg/Dec	Exec	Mem	Wr				
R-type		Ifetch	Reg/Dec	Exec	Mem	Wr			
Load			Ifetch	Reg/Dec	Exec	Mem	Wr		
R-type				Ifetch	Reg/Dec	Exec	Mem	Wr	
R-type					Ifetch	Reg/Dec	Exec	Mem	Wr

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

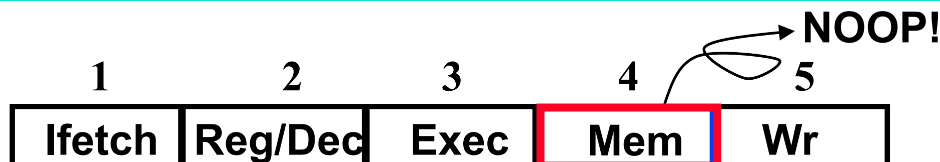
例：sw rs2, rs1(imm12)



- **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

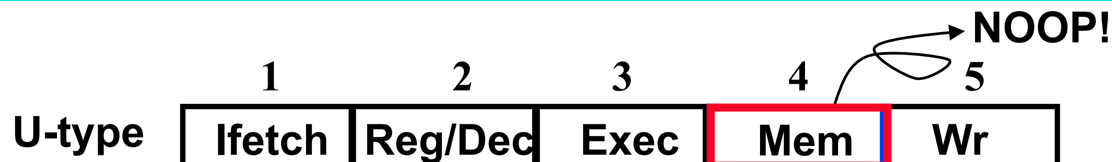
例：ori rd, rs1,imm12



- 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

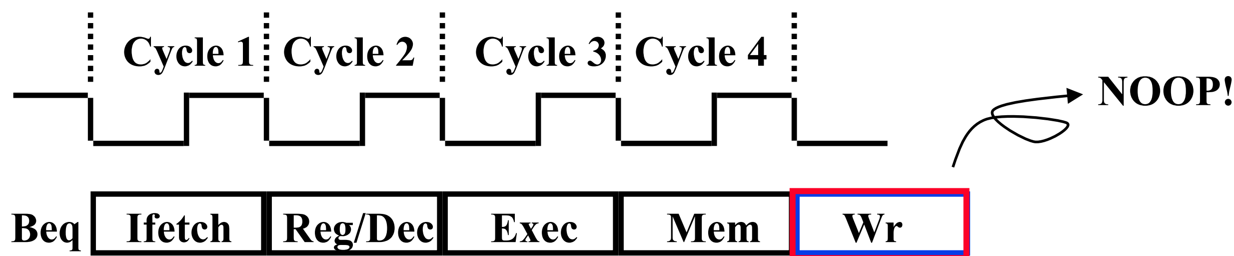
例：lui rd, imm20



- 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

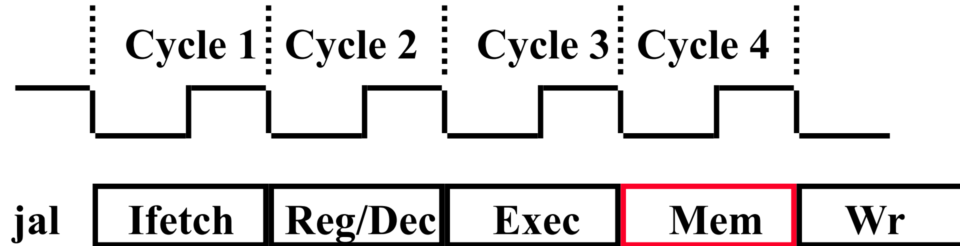
例：beq rs1, rs2, imm12



- 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！

J型的四个阶段

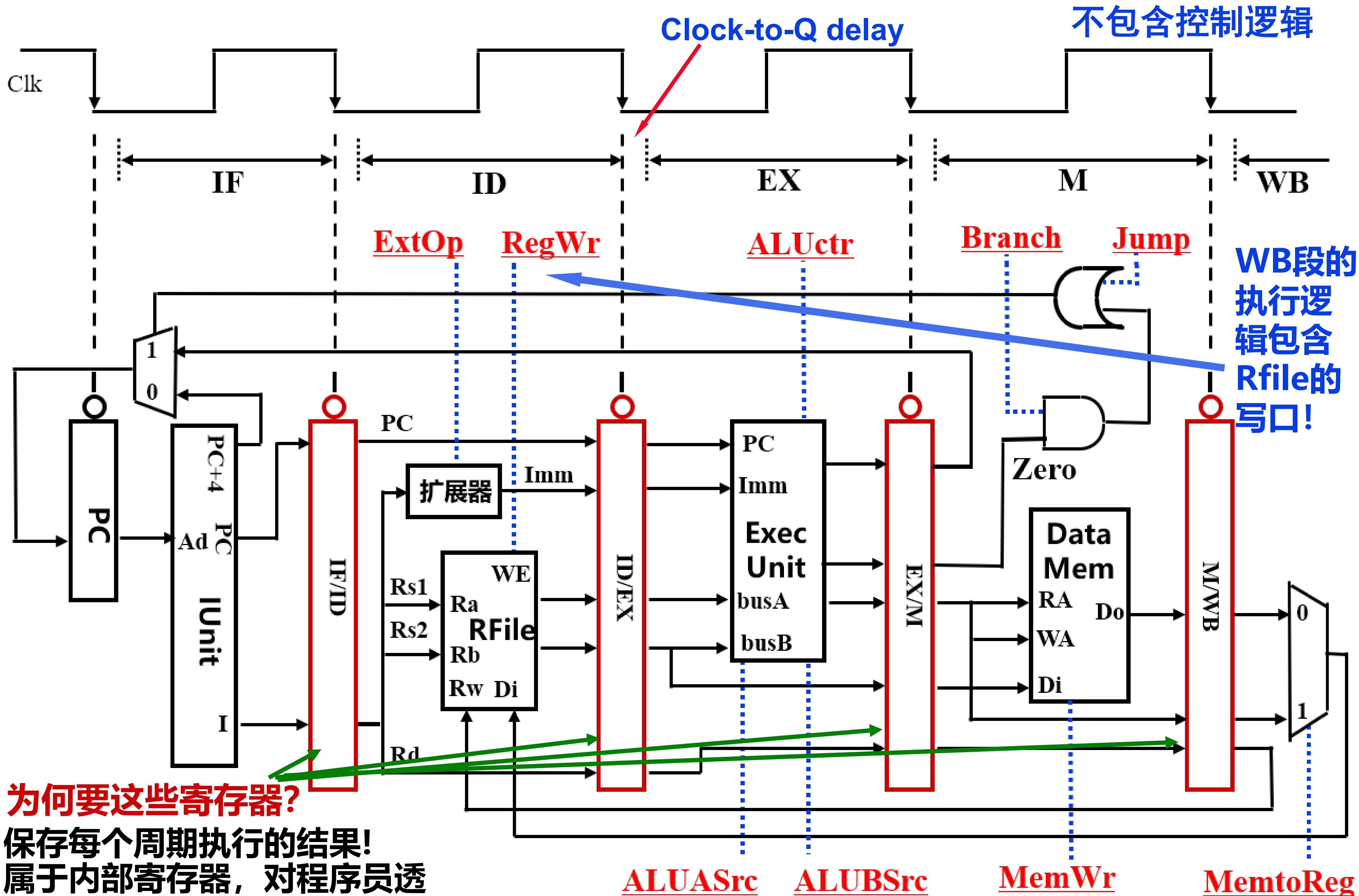
例：jal rd, imm20



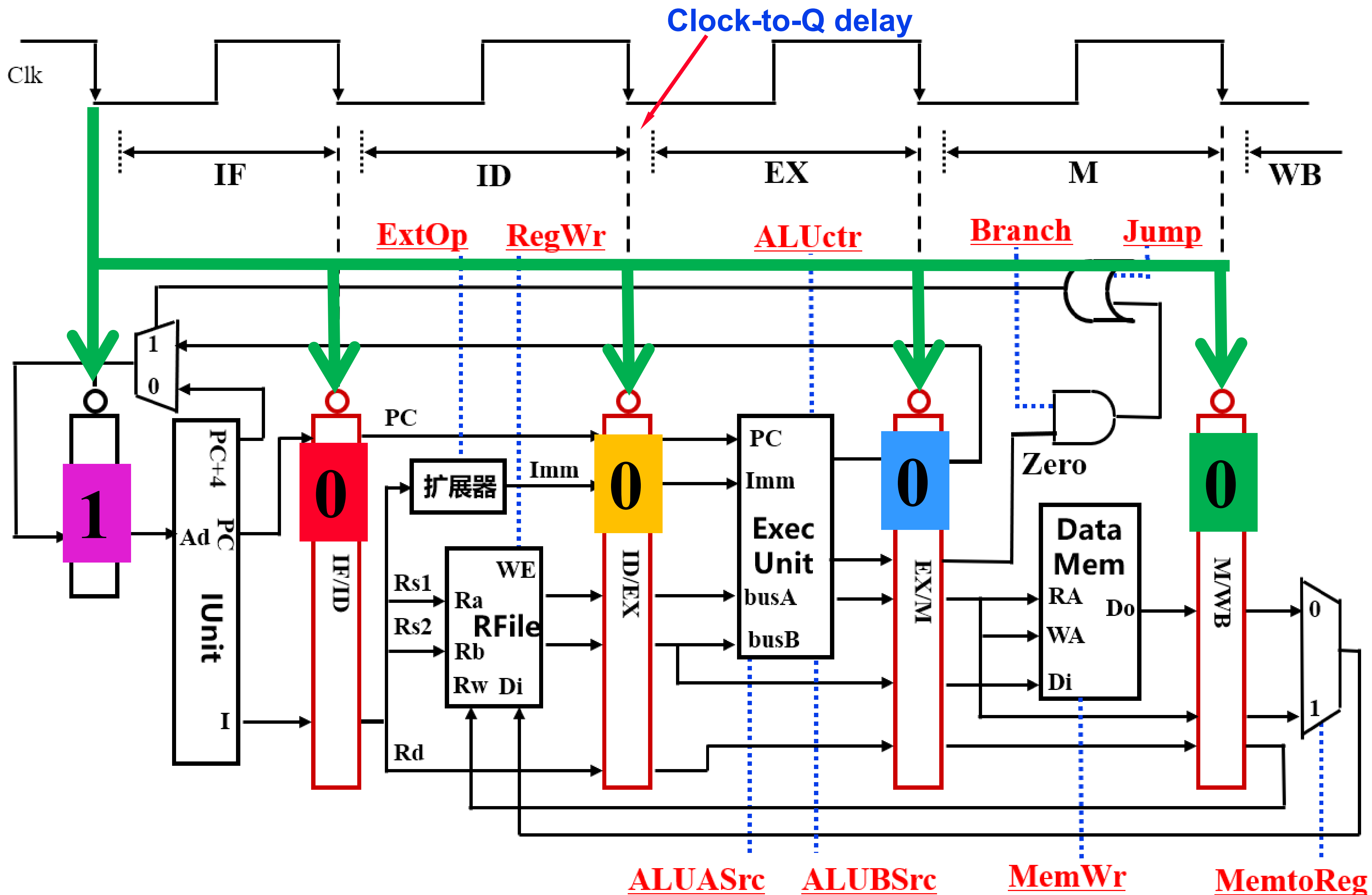
- **Ifetch:** 取指令并计算PC+4 （写入PC， 但后续**肯定**需要修改PC ）
- **Reg/Dec:**从寄存器取数，同时指令在译码器进行译码
- **Exec:** 执行阶段
  - ALU中计算PC+4（准备写入rd）
  - Adder中计算转移地址（PC+SEXT（imm20）<<1）
- **Mem:**把转移地址写入PC
- **Wr:** 把ALU运算结果（PC+4）写入rd,

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

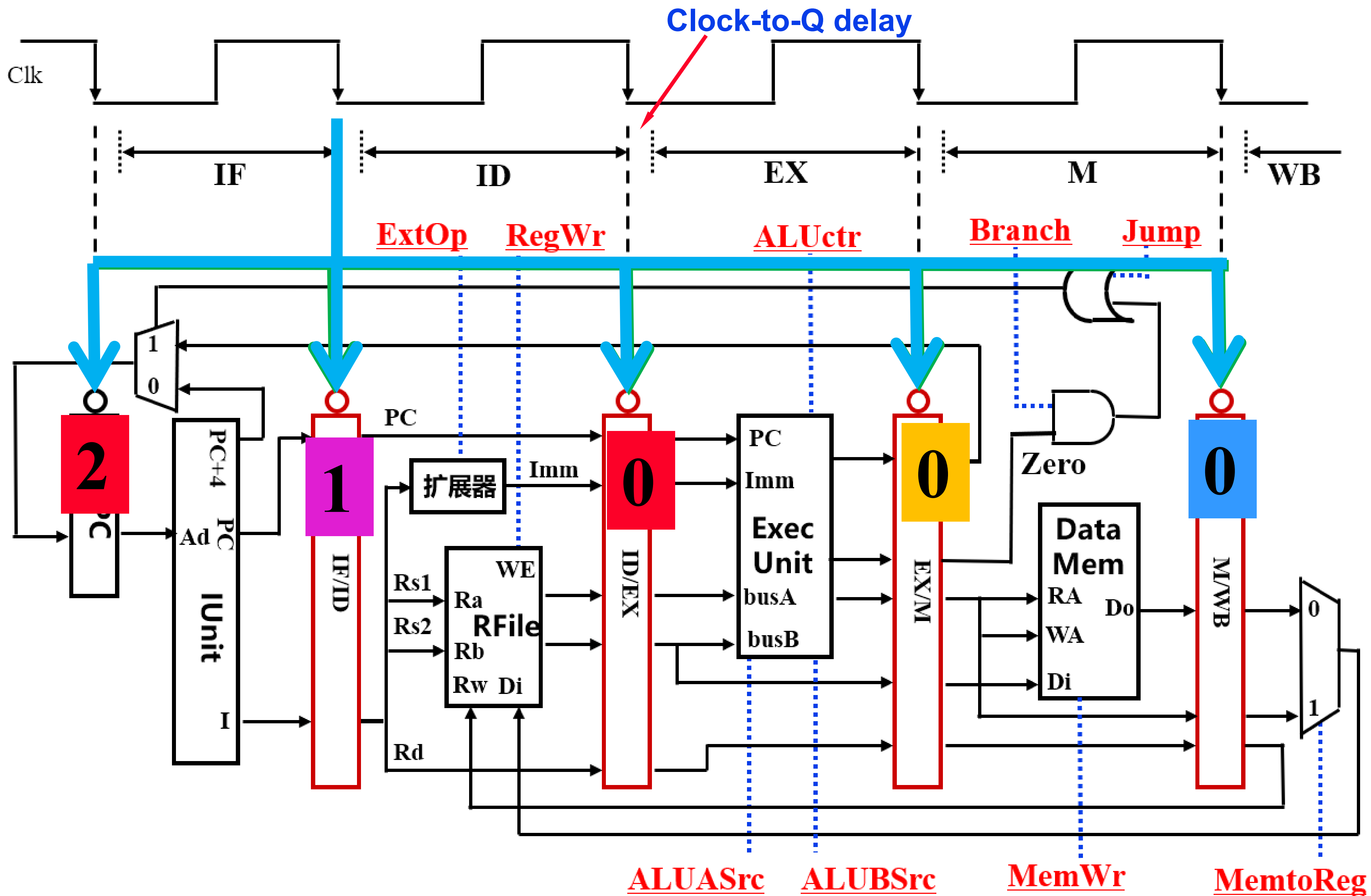
A Pipelined Datapath（五阶段流水线数据通路）



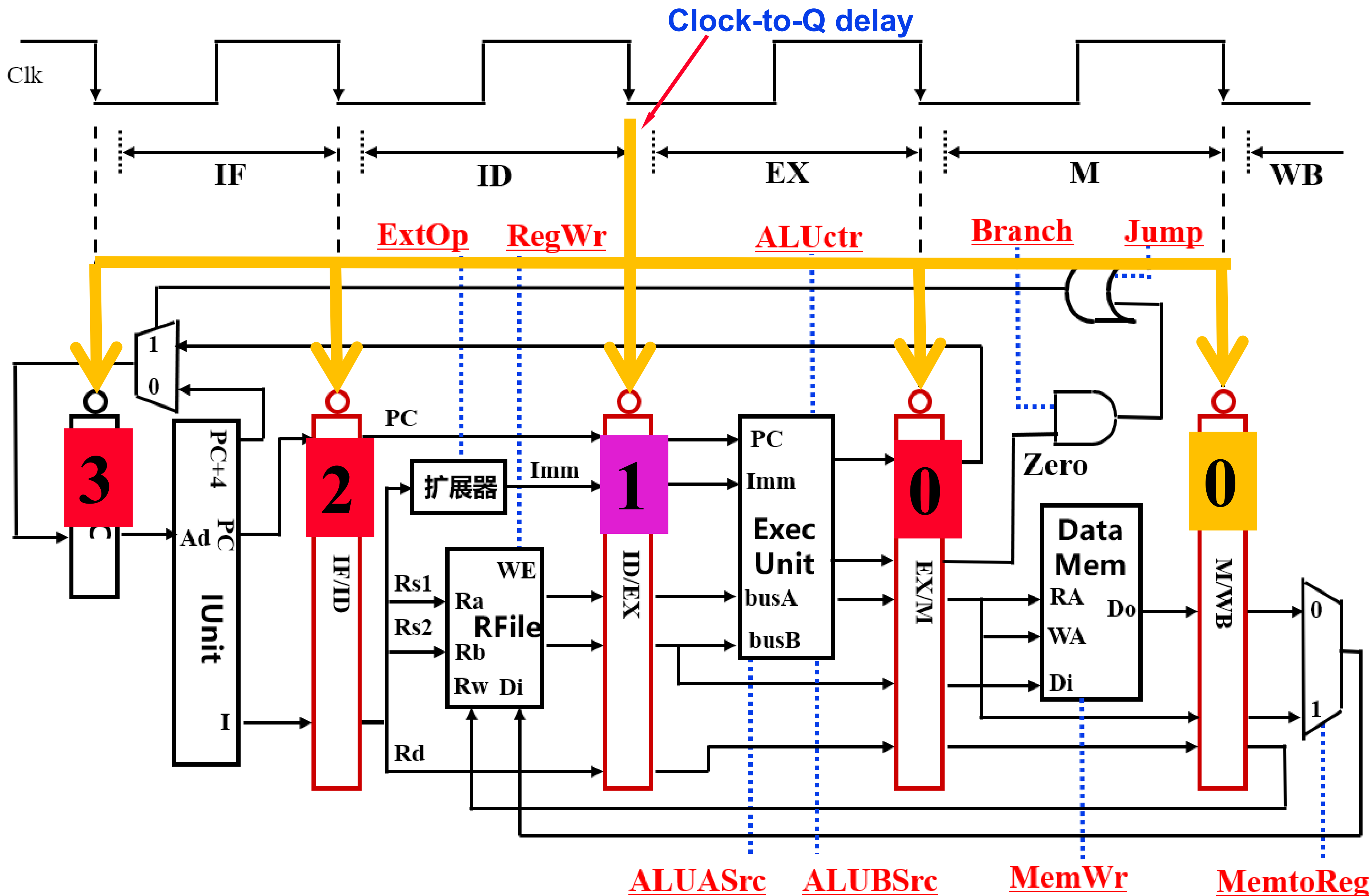
补充说明一下。。。。（请注意“1”）



补充说明一下。。。。（请注意“1”）



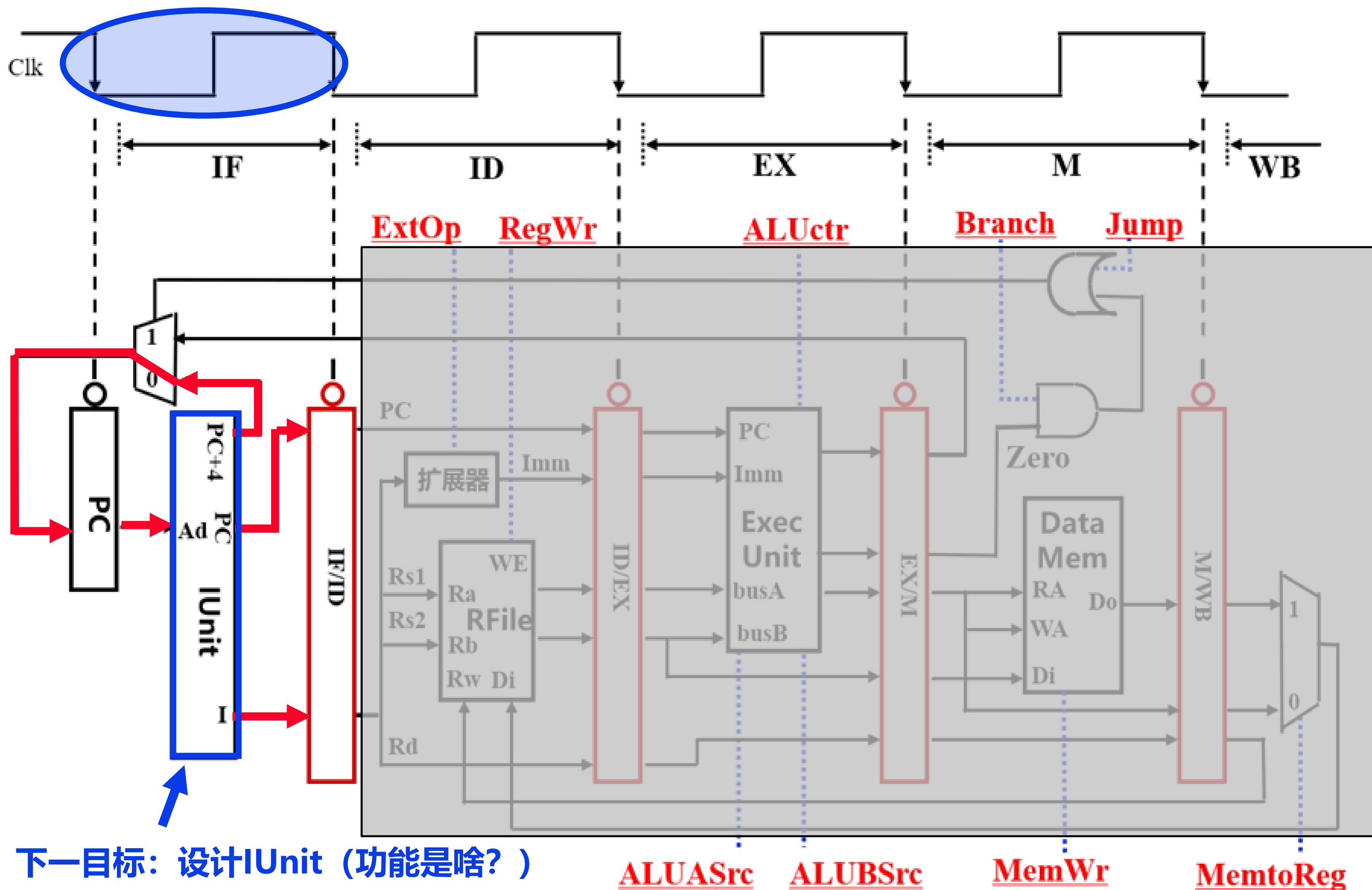
补充说明一下。。。。（请注意“1”）



下面介绍1条指令在流水线通路中的执行过程

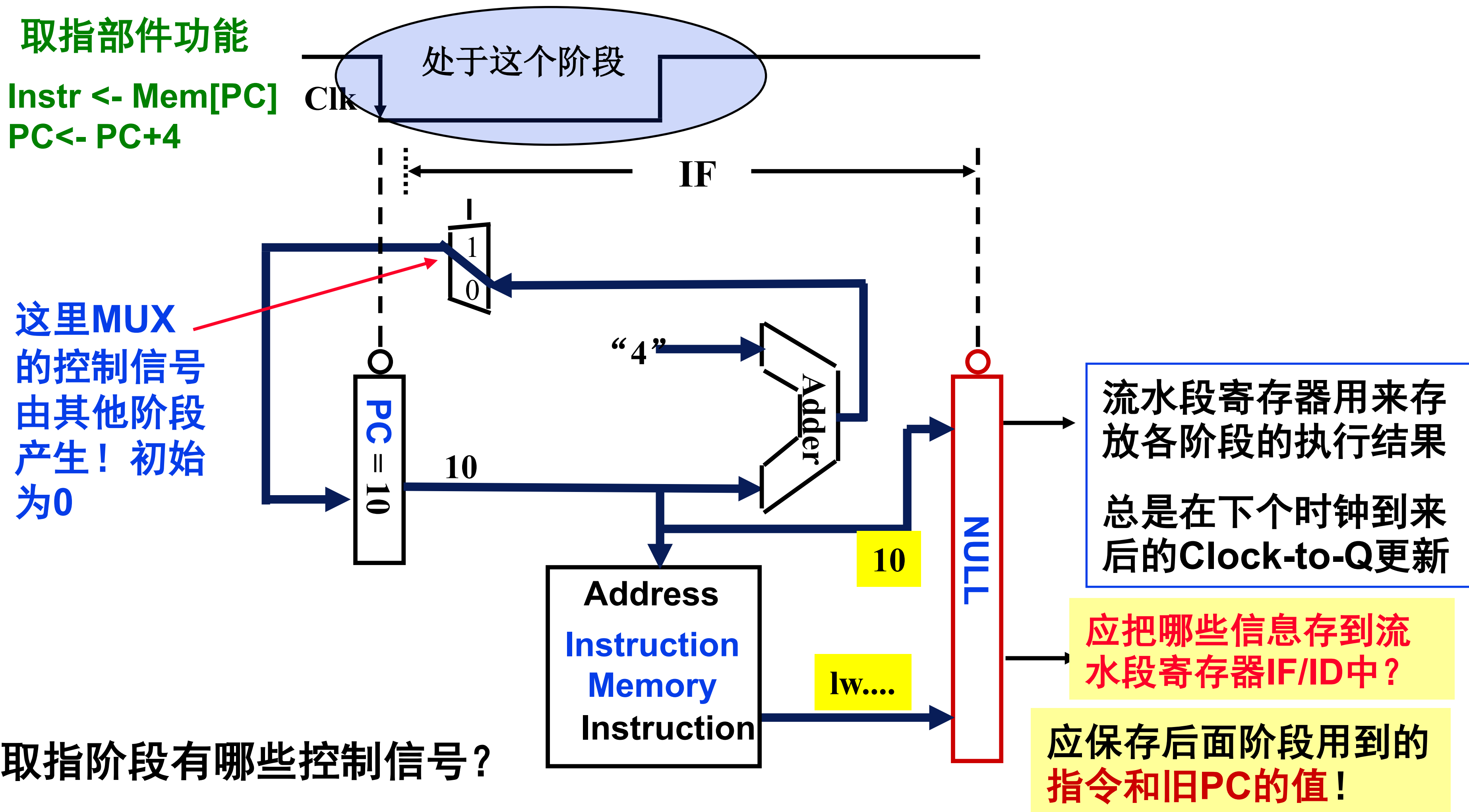
取指令（IF）阶段

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



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)

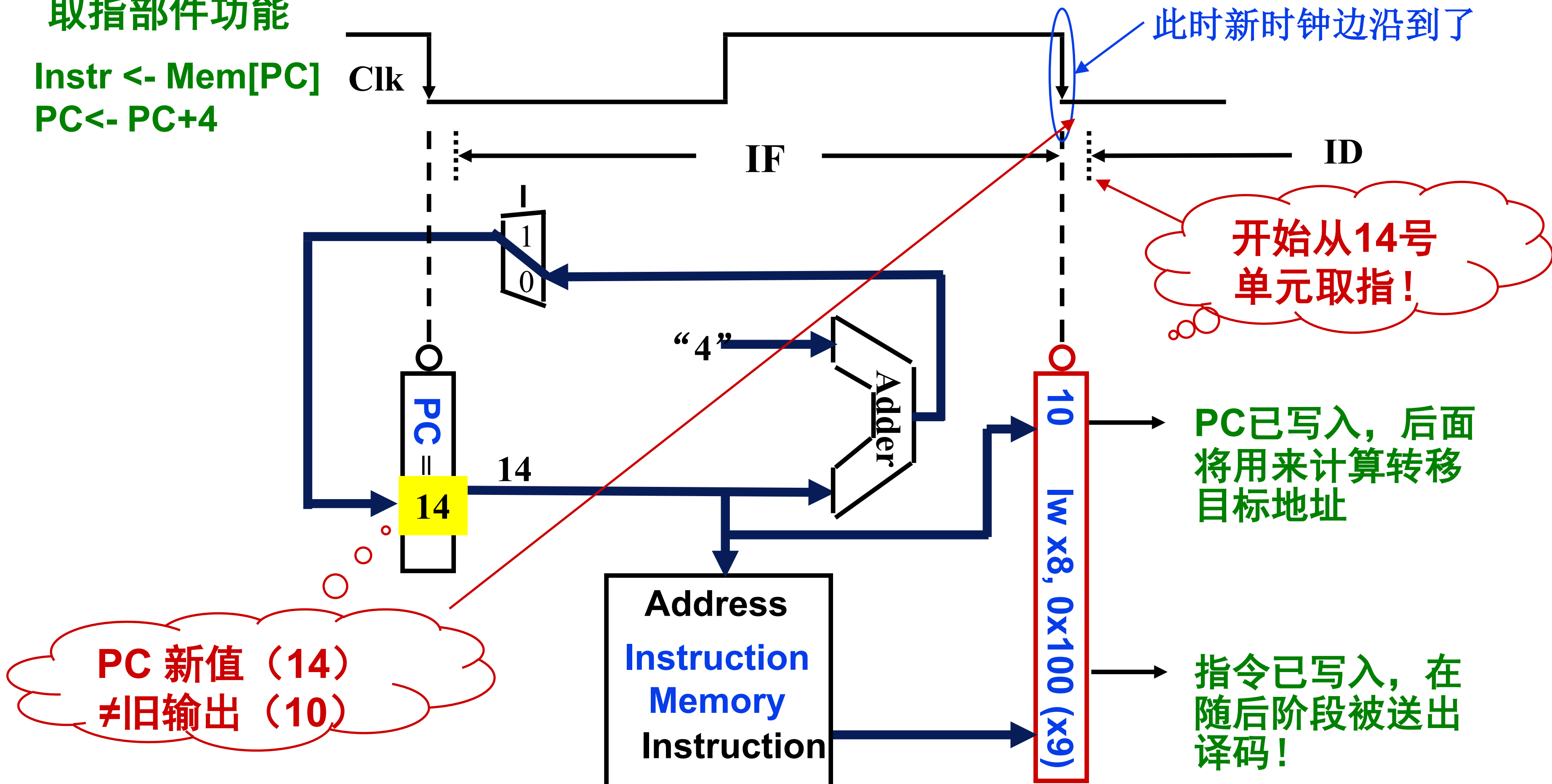


取指阶段有哪些控制信号？

不需控制信号，因为每条指令执行功能一样，是确定的，无需根据指令的不同来控制执行不同的操作！

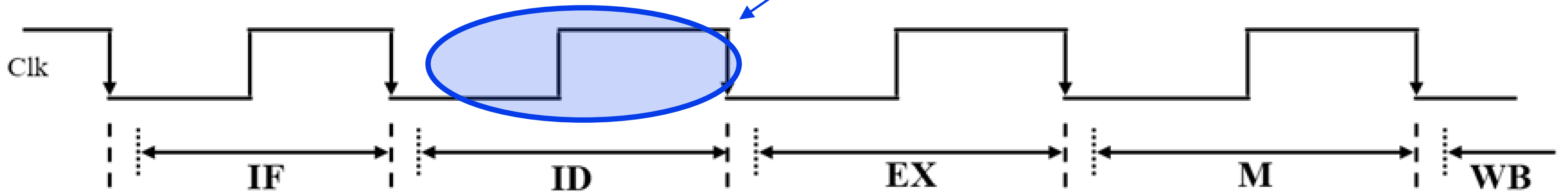
取指令部件 IUnit的设计——结束时

◦ 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)]



指令还要存ID/EX中吗？

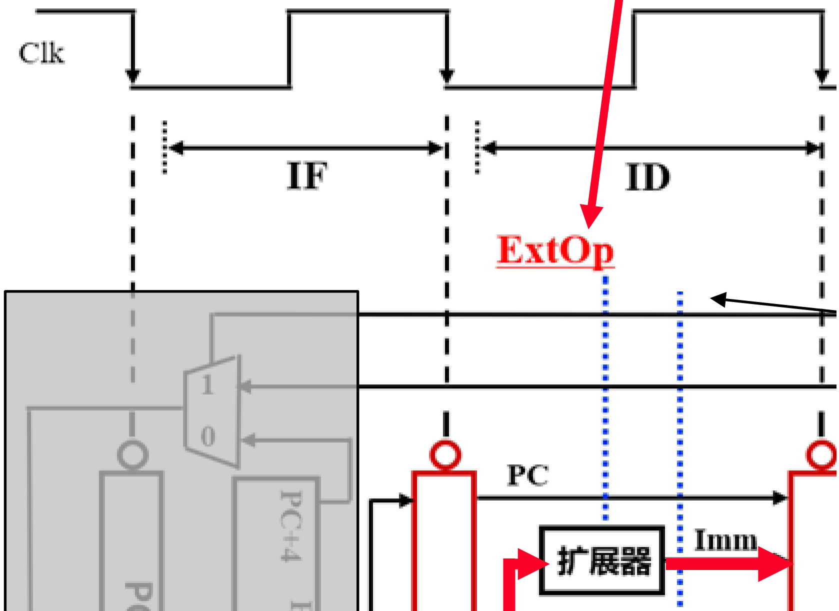
不要，只要存以下信息：

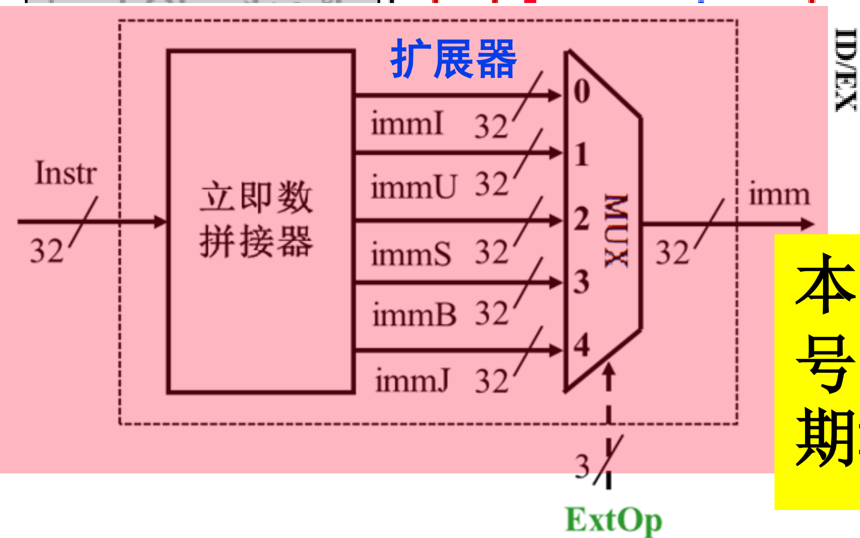
R[Rs1], R[Rs2], Rd, Imm, PC

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



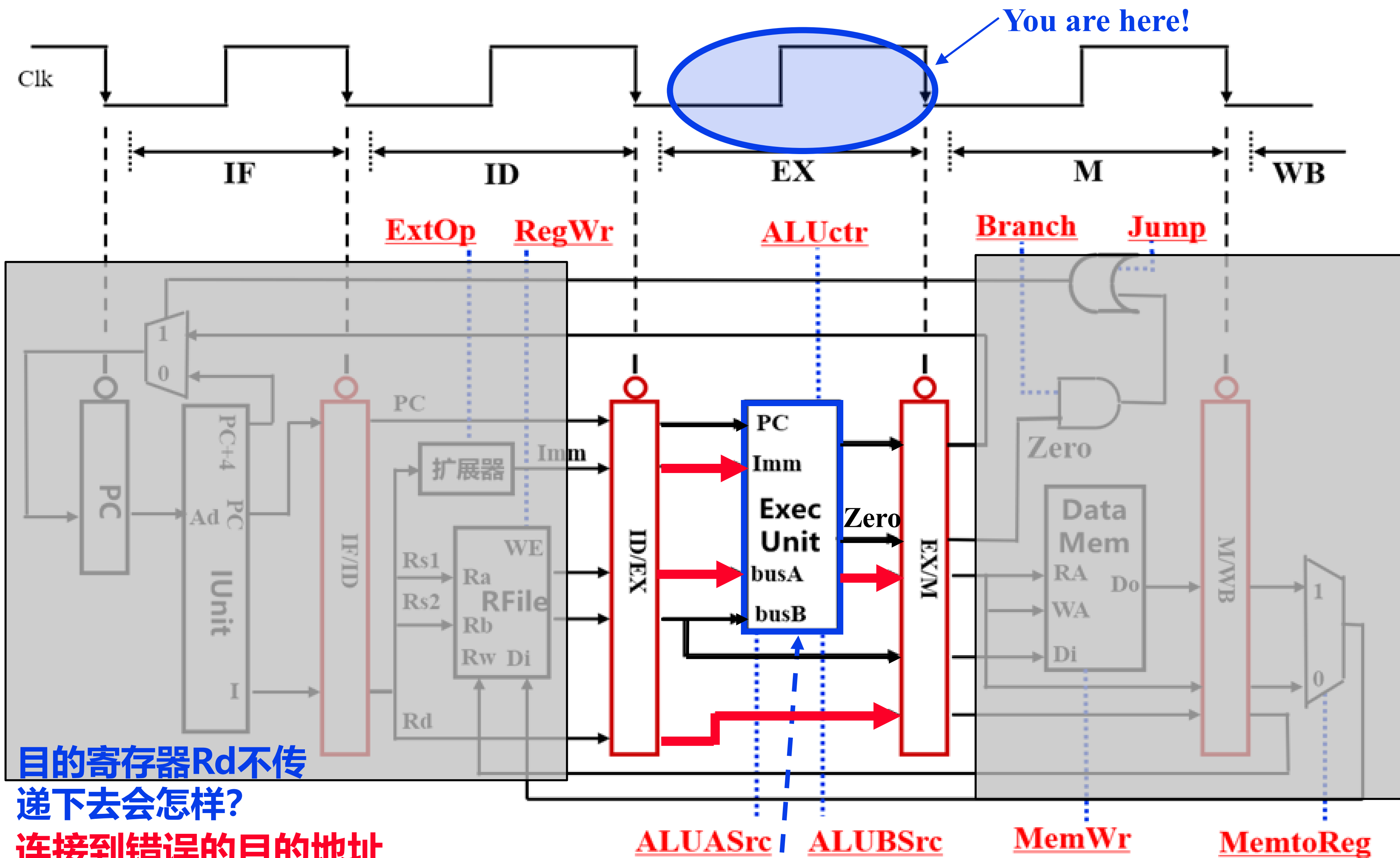


指　令	立即数编码类型	ExtOp<2:0>
add rd, rs1, rs2	无立即数	×××
slt rd, rs1, rs2		
sltu rd, rs1, rs2		
ori rd, rs1, imm12	I-型立即数（immI）	0 0 0
lui rd, imm20	U-型立即数（immU）	0 0 1
lw rd, rs1, imm12	I-型立即数（immI）	0 0 0
sw rs1, rs2, imm12	S-型立即数（immS）	0 1 0
beq rs1, rs2, imm12	B-型立即数（immB）	0 1 1
jal rd, imm20	J-型立即数（immJ）	1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

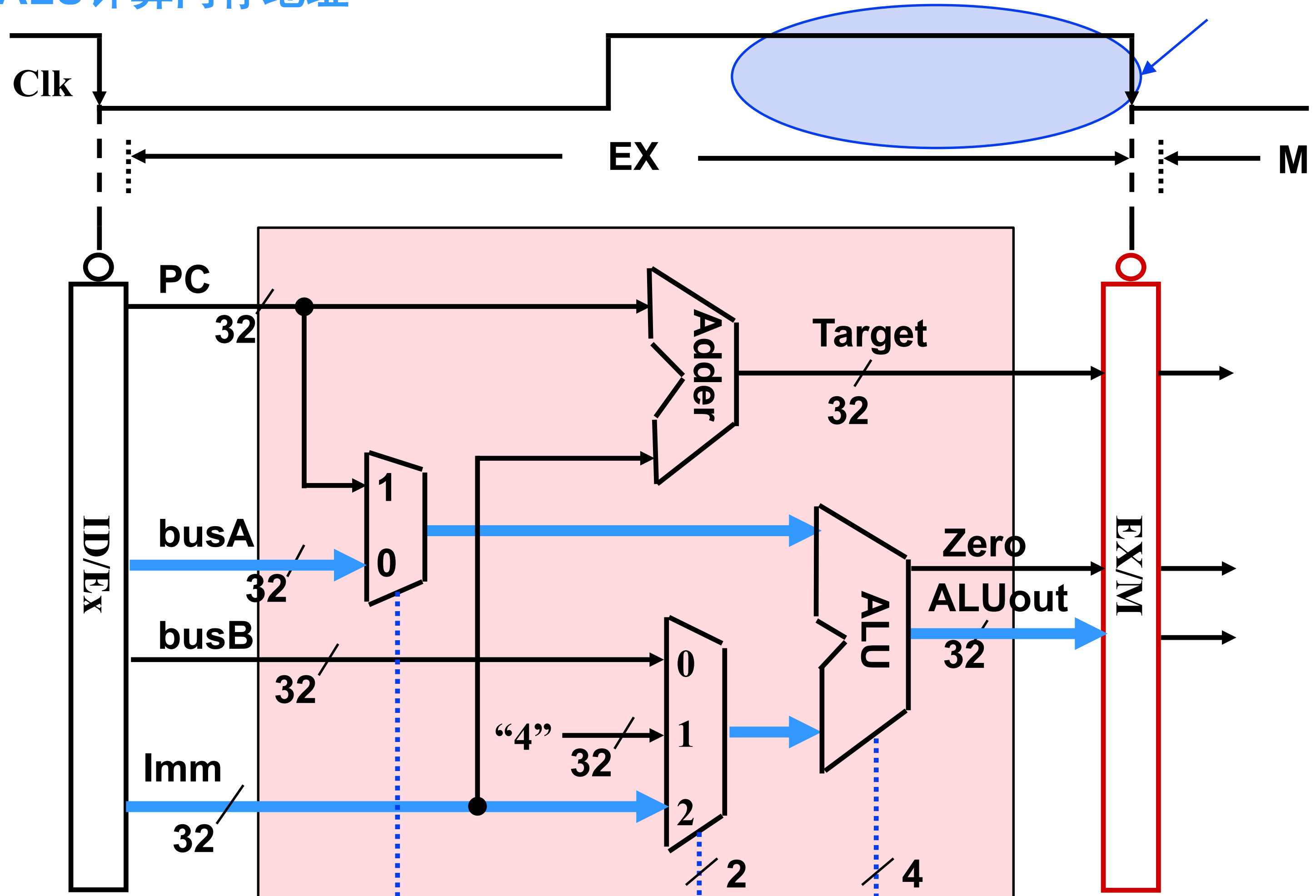
Load指令的地址计算阶段

- 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

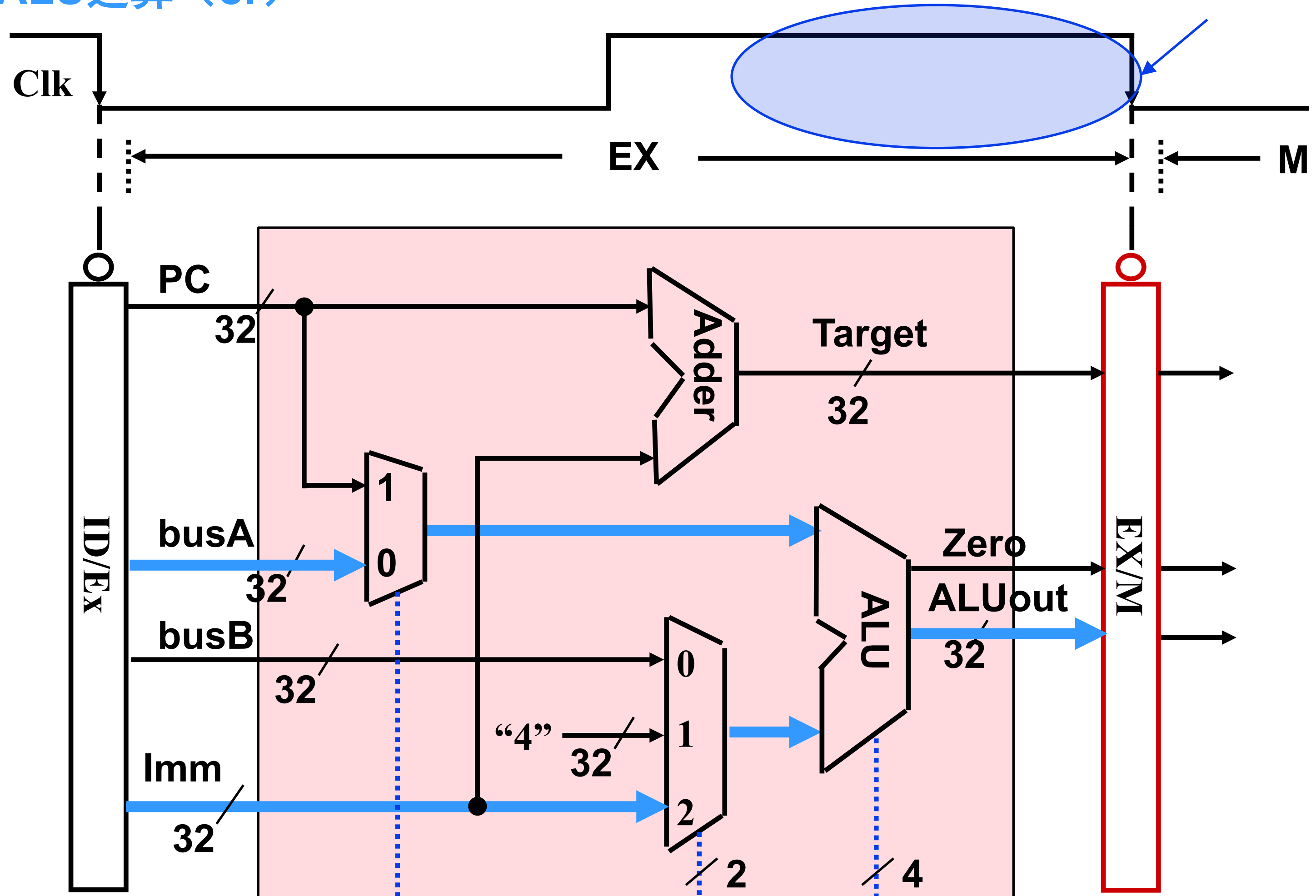
执行部件功能是：ALU计算内存地址



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

执行部件（Exec Unit）的设计：ori指令（I型）

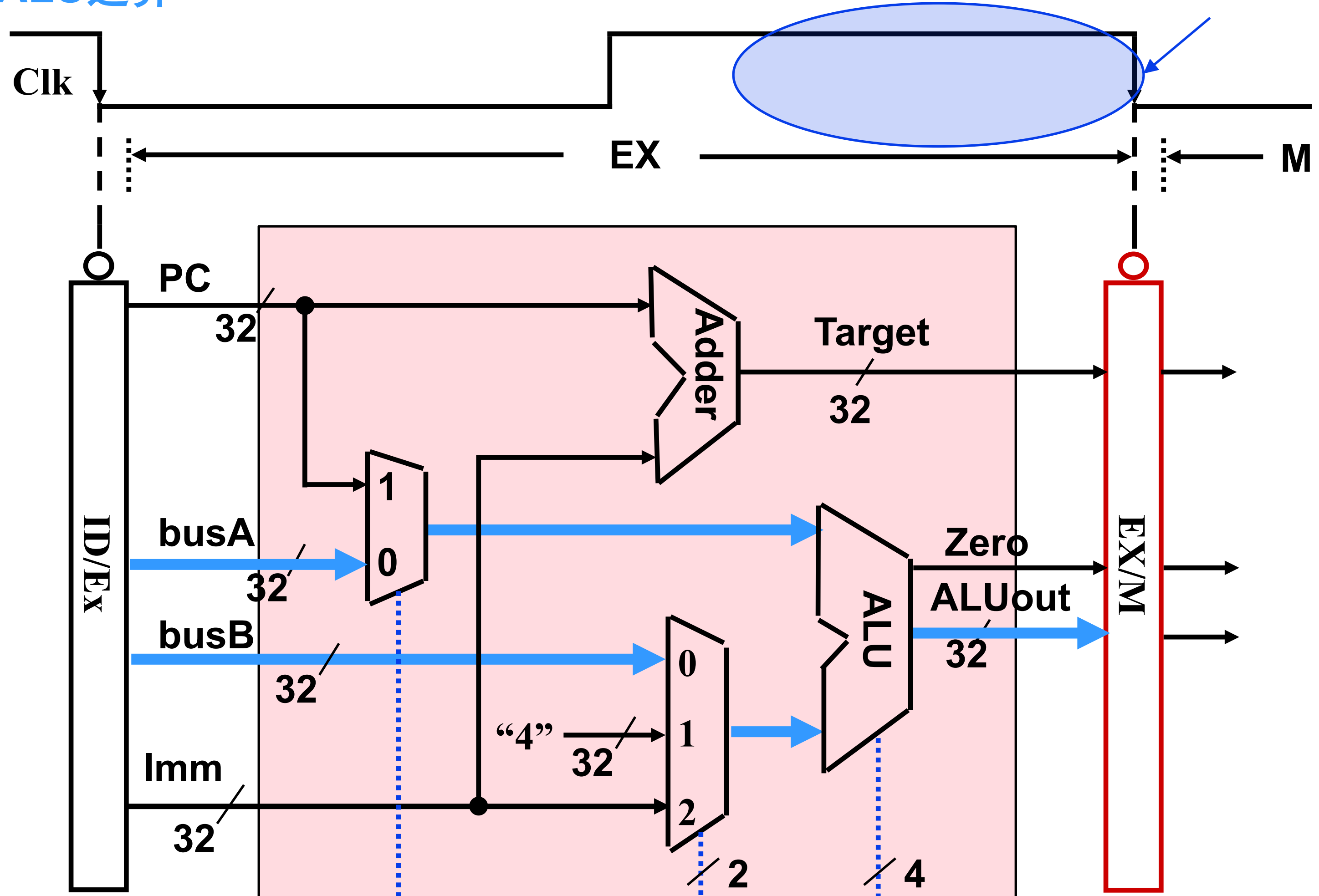
执行部件功能是：ALU运算（or）



ori rd, rs1,imm12

执行部件（Exec Unit）的设计：R型指令

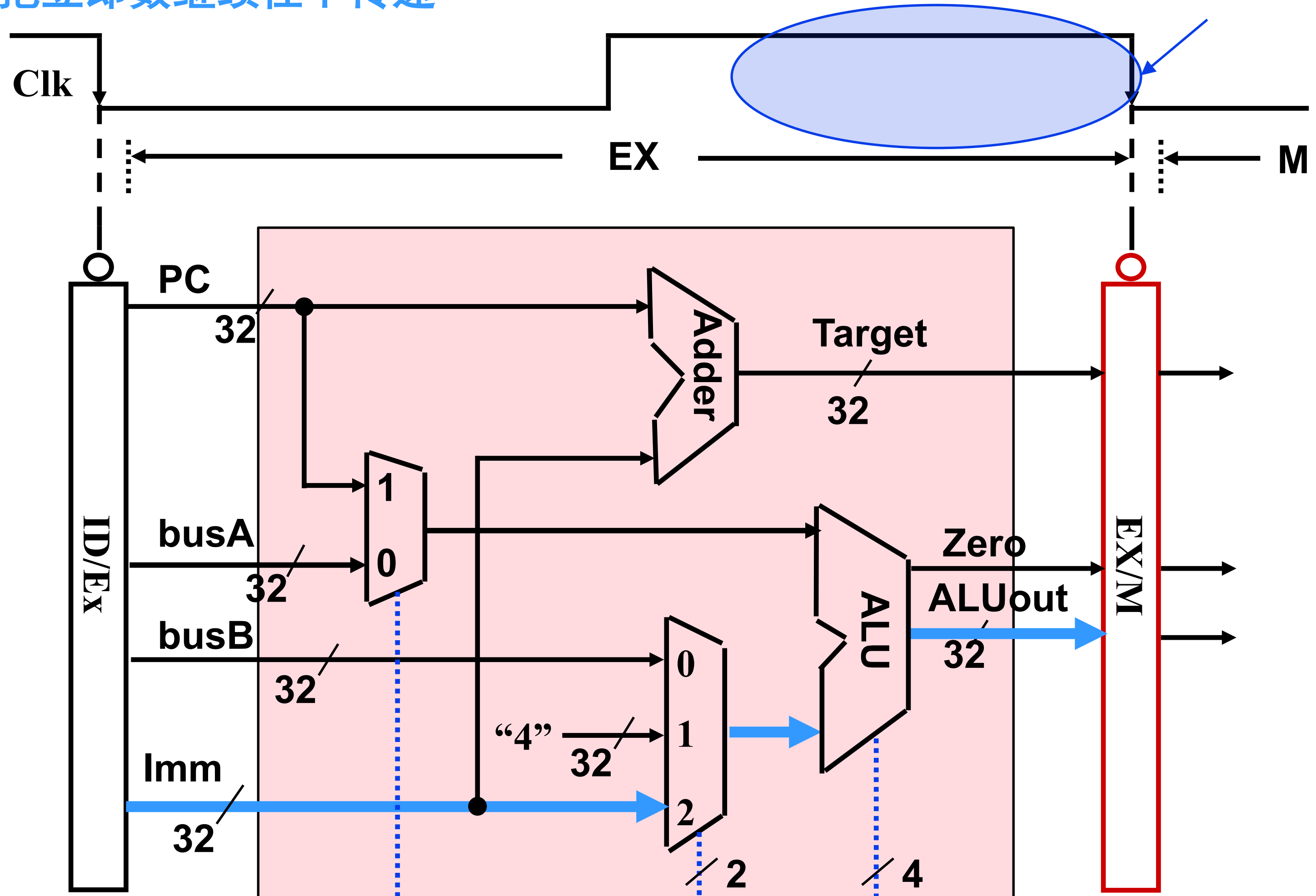
执行部件功能是：ALU运算



add(sub..) rd, rs1, rs2

执行部件（Exec Unit）的设计：U型指令

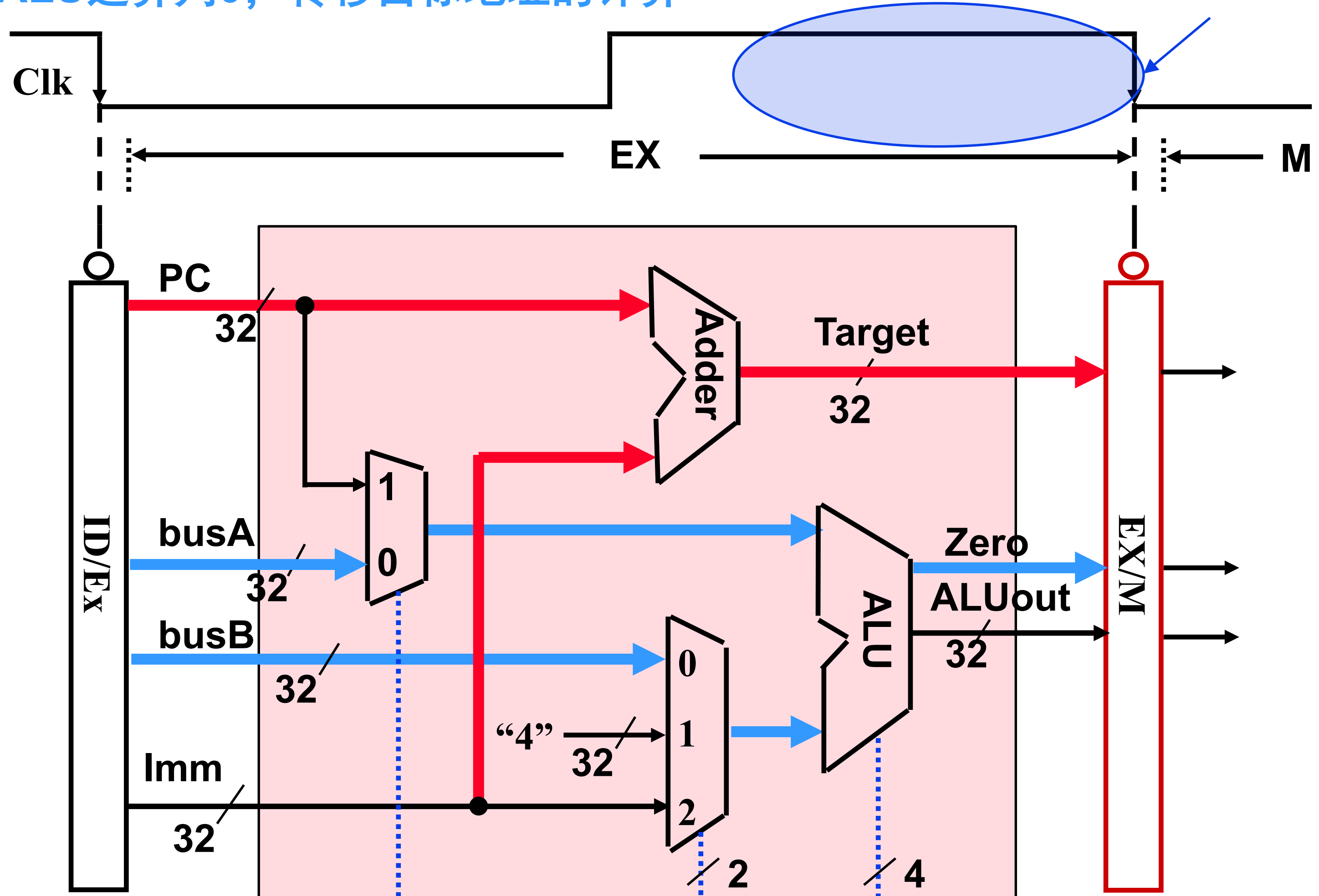
执行部件功能是：把立即数继续往下传递



lui rd, imm20

执行部件（Exec Unit）的设计：B型指令

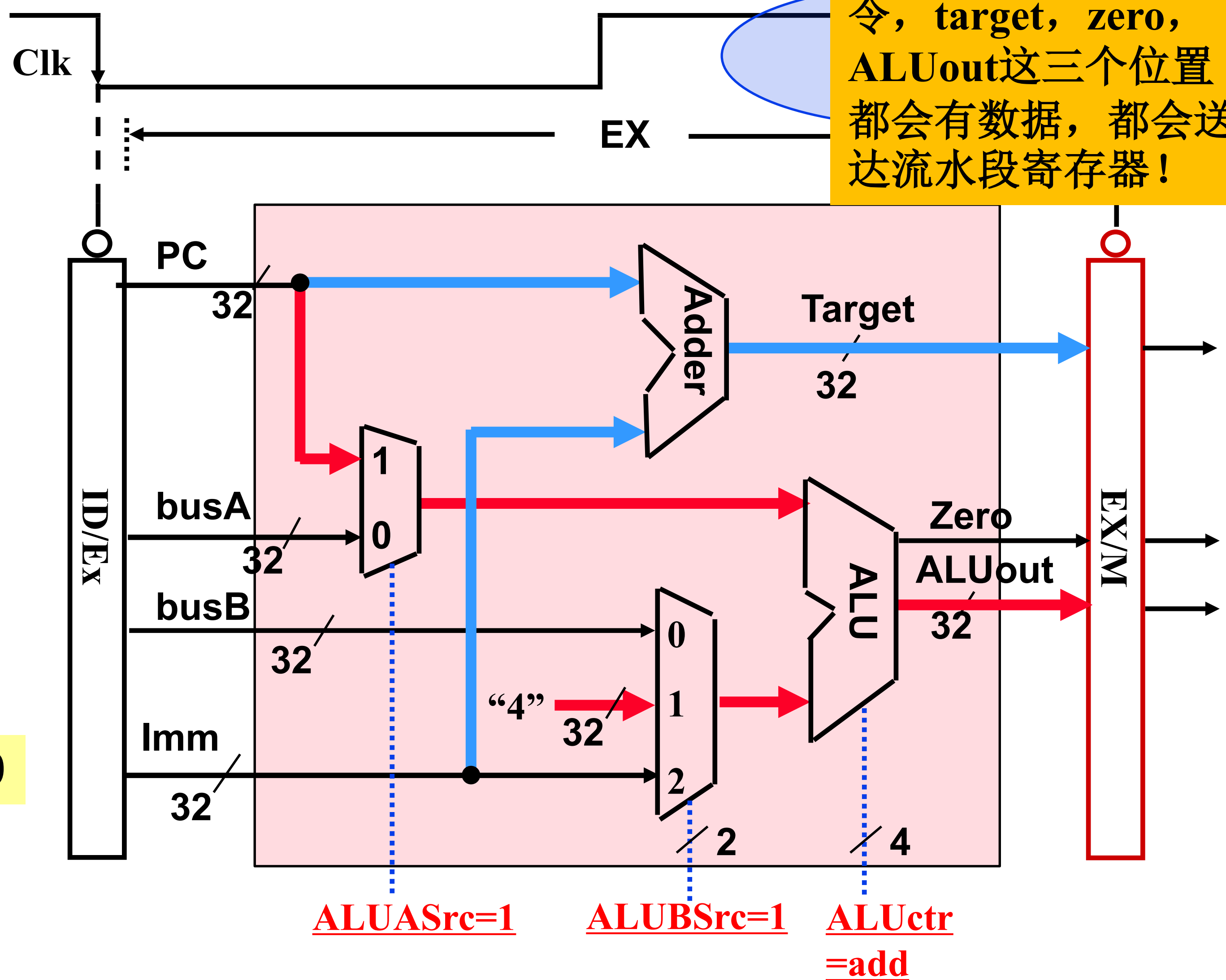
执行部件功能是：ALU运算判0，转移目标地址的计算



beq rs1, rs2, imm12

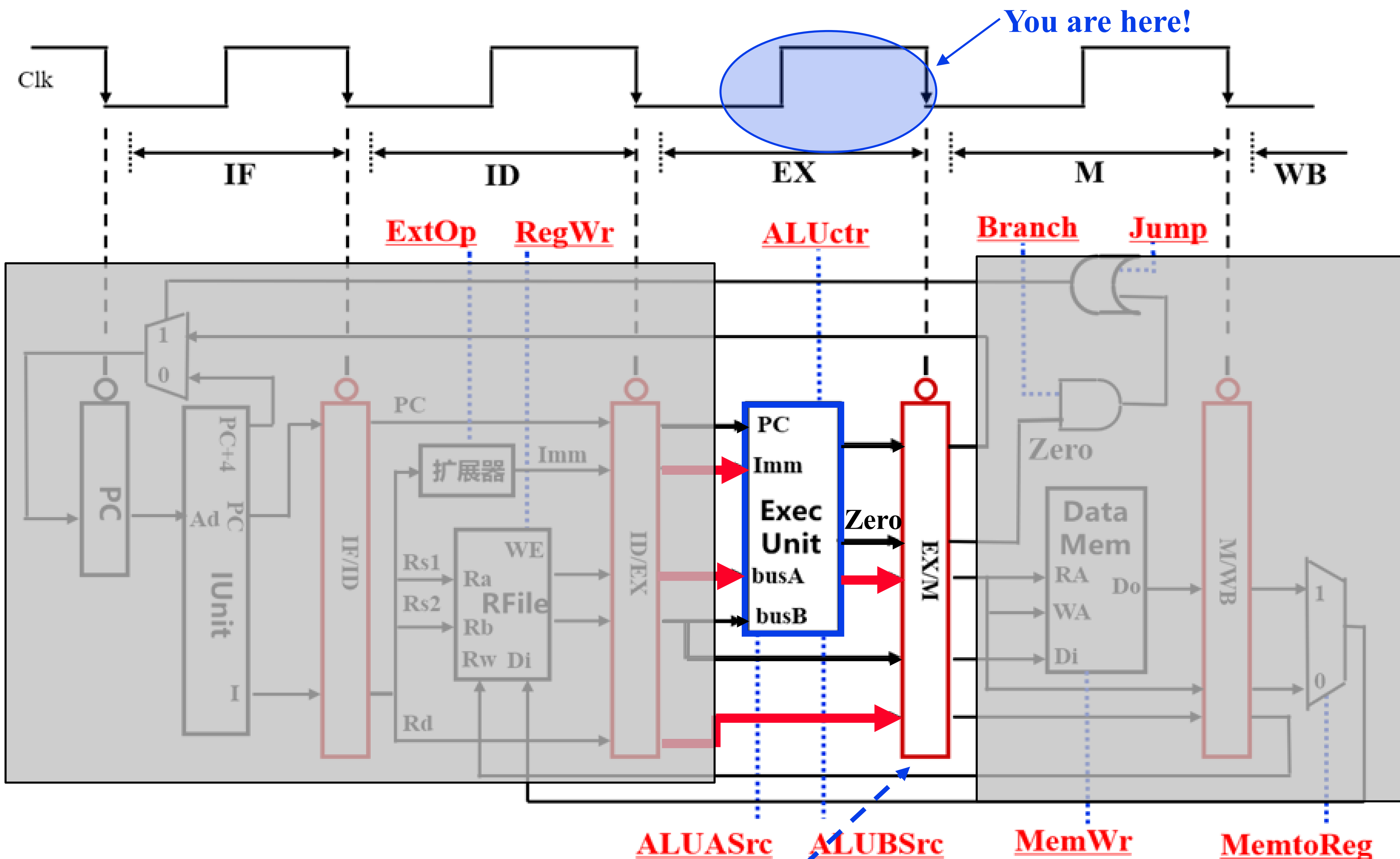
执行部件（Exec Unit）的设计：J型指令

执行部件功能是：ALU运算(PC+4) 和转移目标地址的计算



jal rd, imm20

再看一下：EX/M这个流水段寄存器

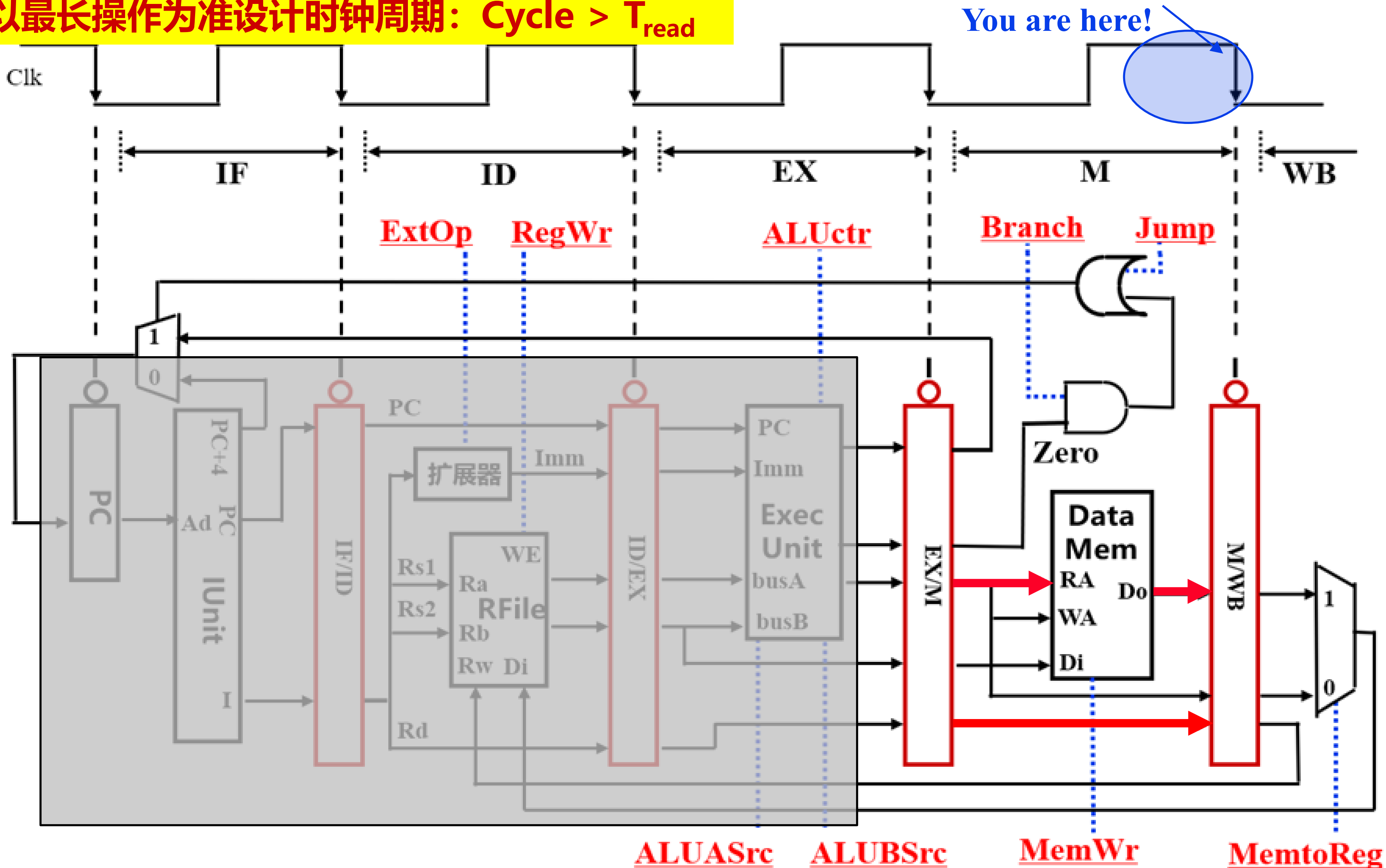


记录了：跳转地址，Zero，ALU运算结果，busB（sw指令后面要用到），rd

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)]

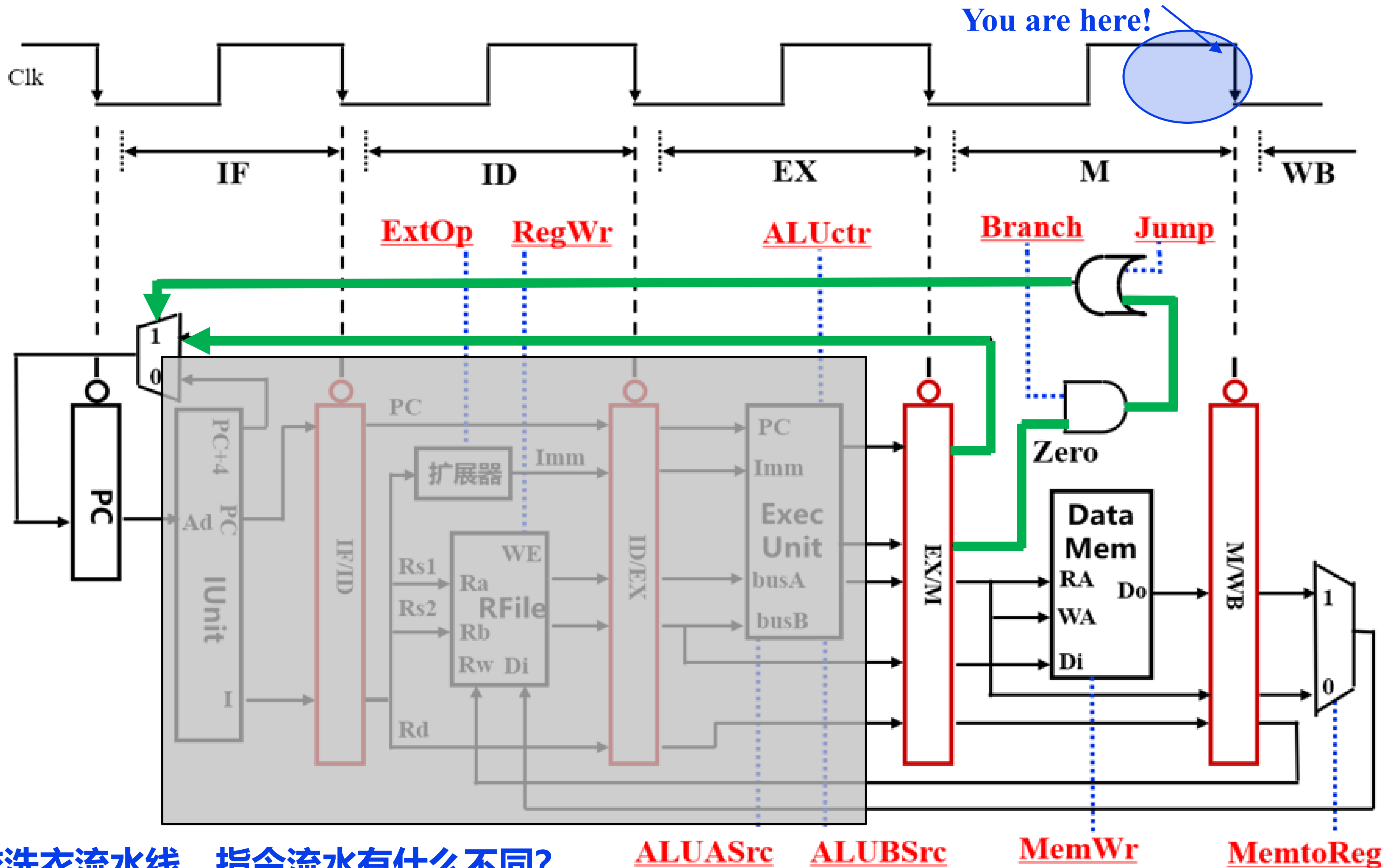
以最长操作为准设计时钟周期：Cycle > T_{read}



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

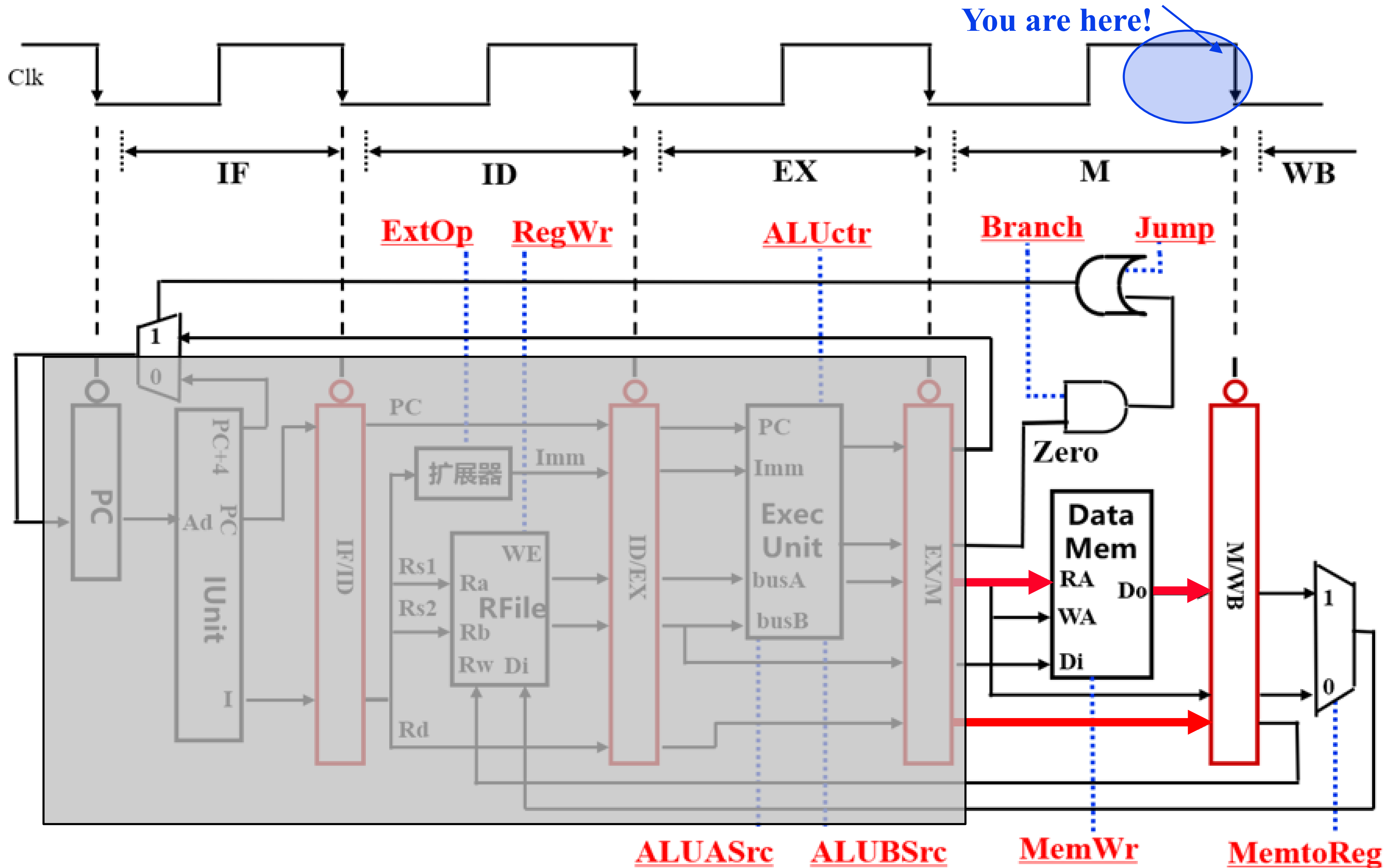
如果是Beq指令呢？



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

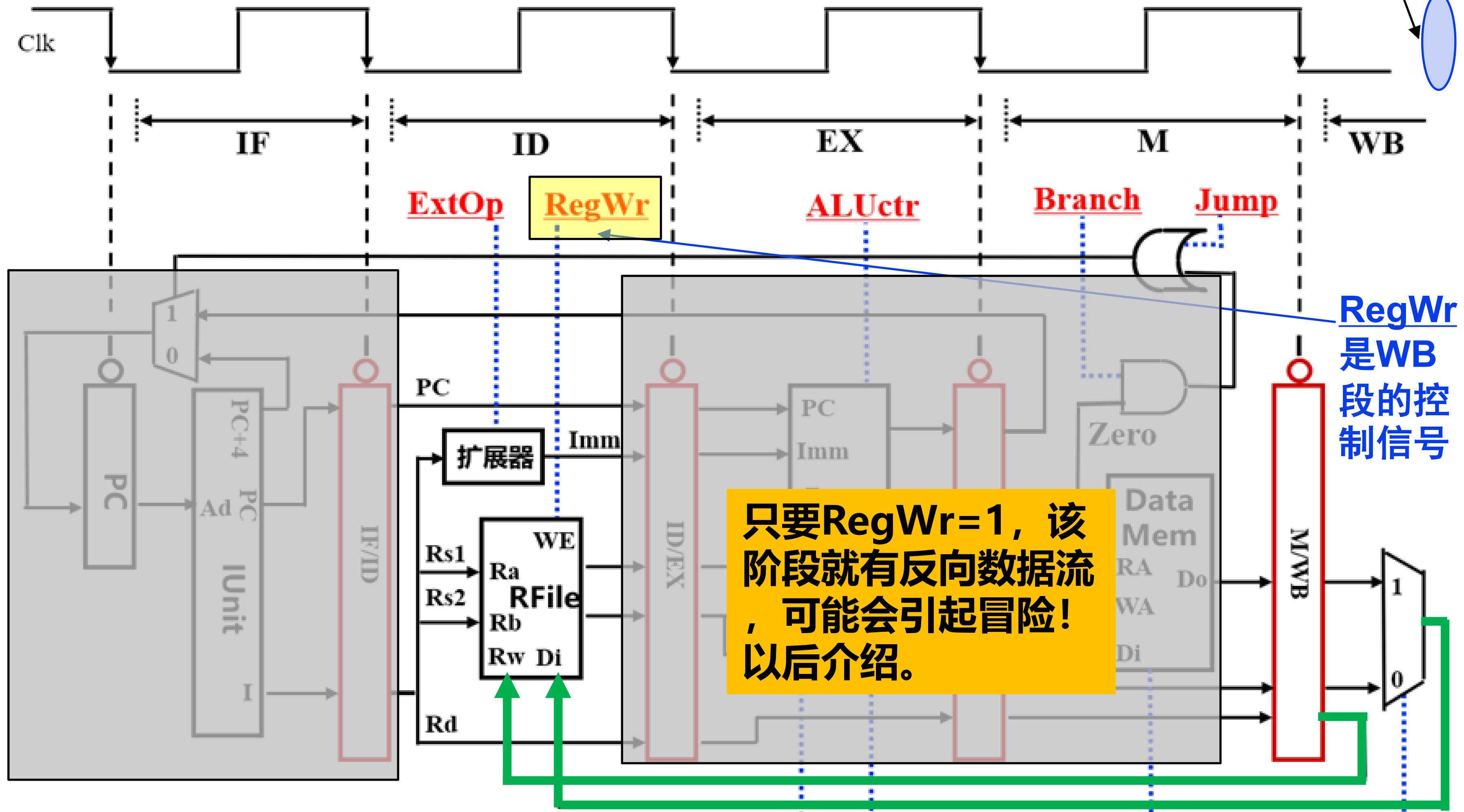


记录了：内存读出的数据，ALU运算结果，rd

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)]

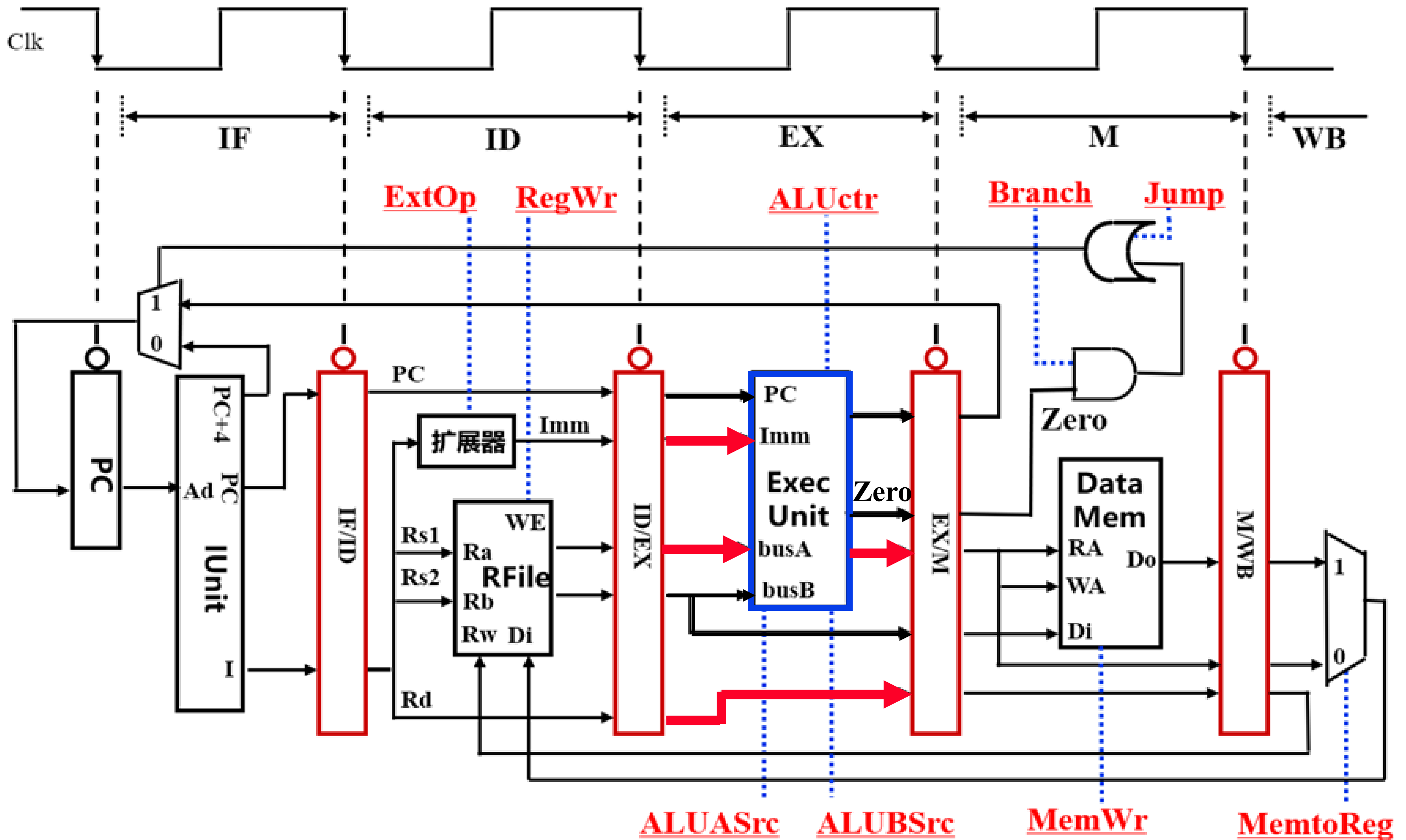


Load控制信号：MemtoReg=1、RegWr=1。

其他指令呢?

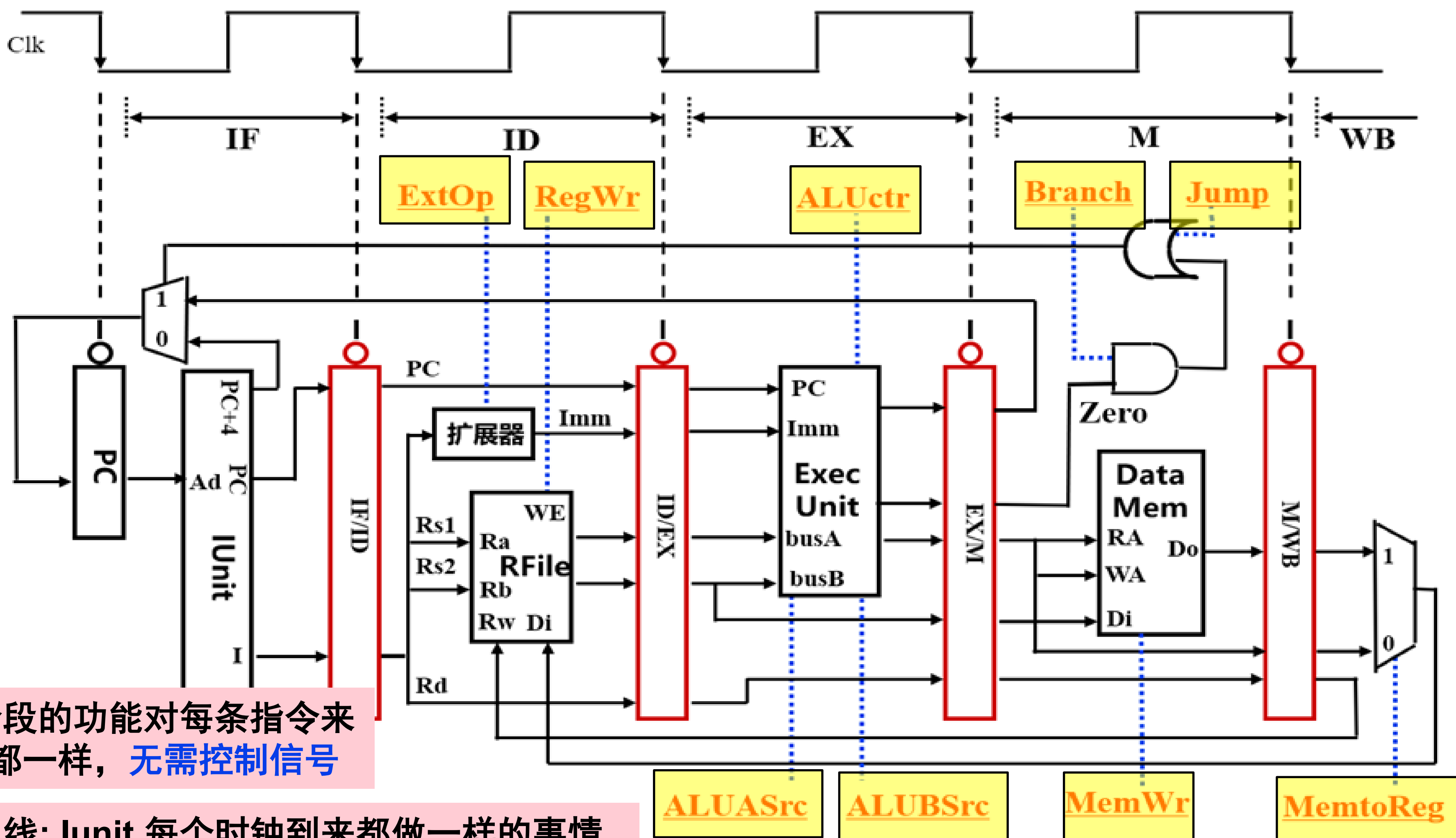
R-型、I-型运算、U-型、J-型: MemtoReg=0、RegWr=1 ; B-型、S-型: MemtoReg=x、RegWr=0

回顾第23次课（一条load指令的流水过程）



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）



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: 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

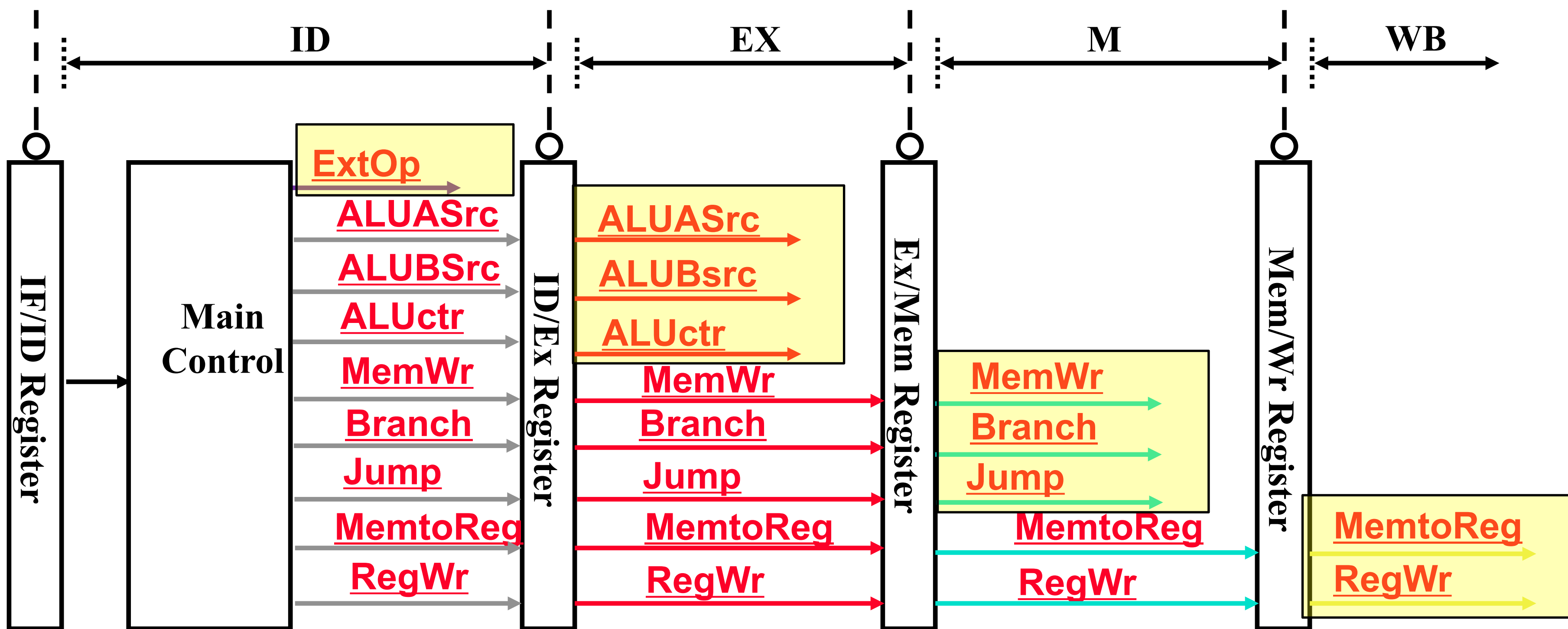
流水线中的Control Signals有哪些？（续）

- 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
    - 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

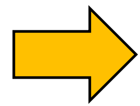
- 在取数/译码（ID）阶段产生本指令所有控制信号
  - ID信号 (ExtOp) 在当前周期中使用
  - EX信号 (ALUAScr, ALUBSrc, ...) 在1个周期后使用
  - M信号 (MemWr, Branch, ...) 在2个周期后使用（改写PC）
  - WB信号 (MemtoReg, RegWr) 在3个周期后使用

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？

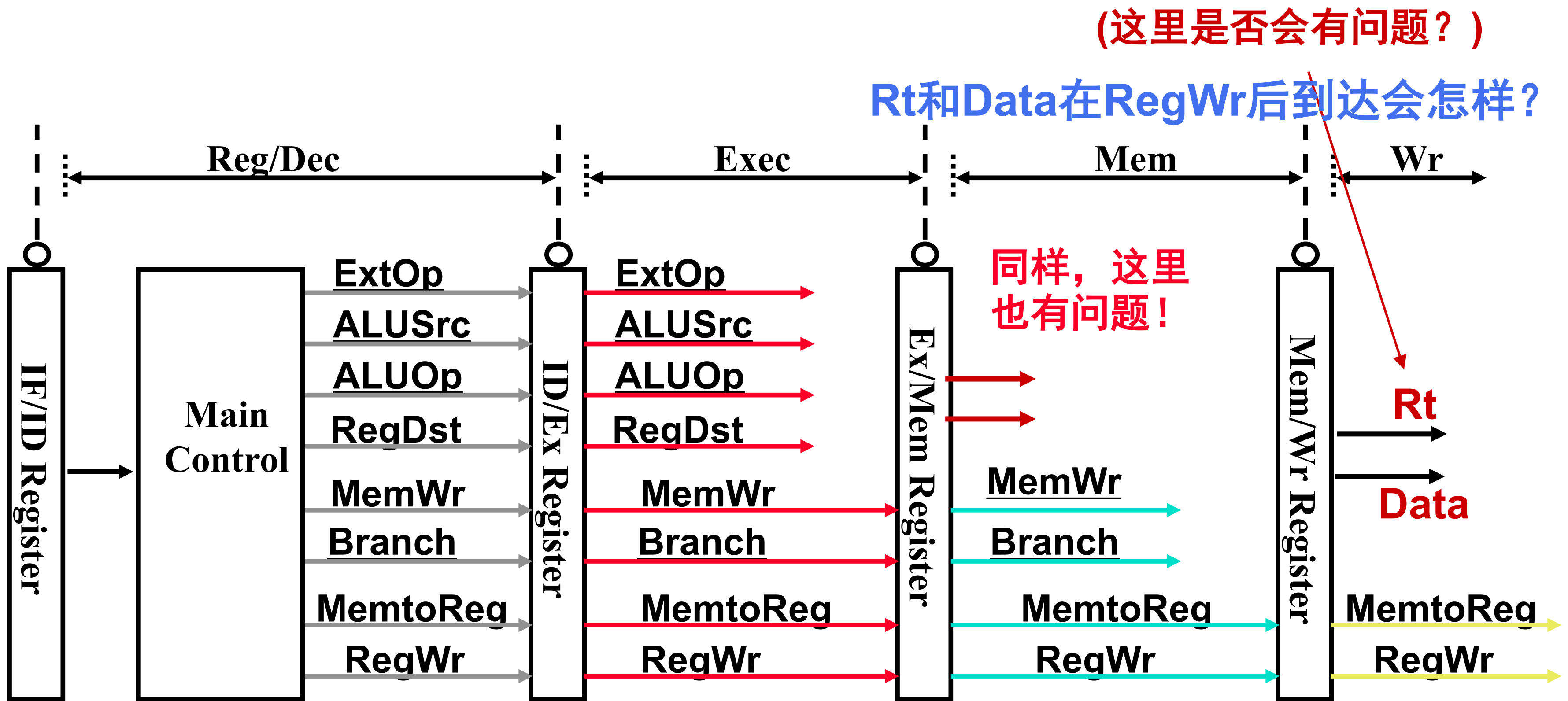
单周期！

（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

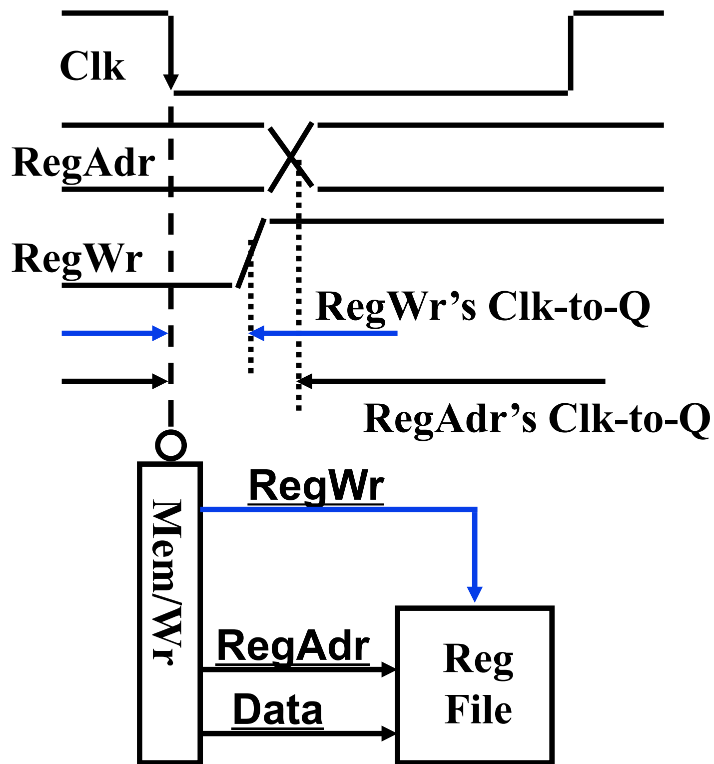
- 某时刻（某个时钟周期内），
- CPU数据通路中的不同阶段同时工作，
- 分别执行着不同的指令，
- 而这些不同的指令都能够从流水段寄存器得到自己所需的控制信号和数据

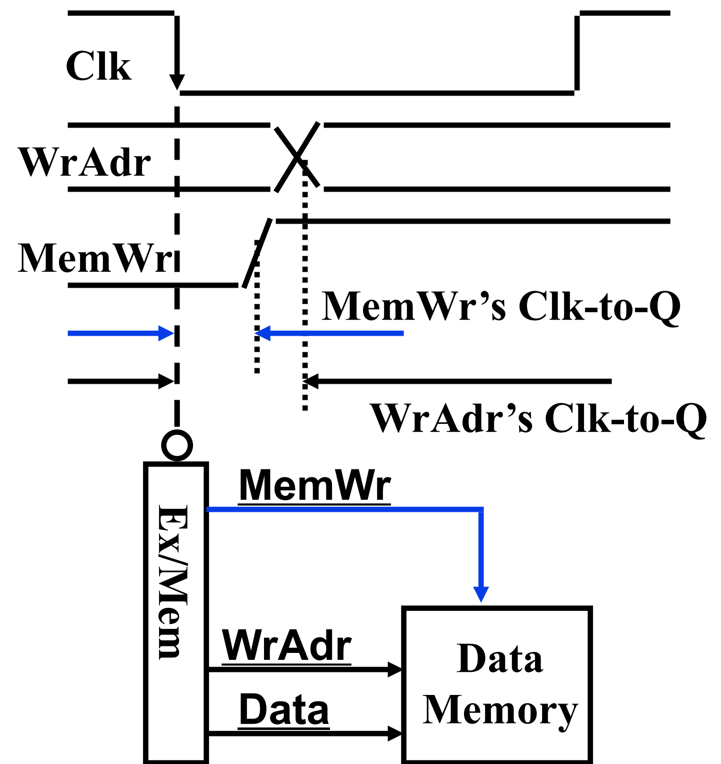
（不要求）流水线中的“竞争”问题



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！

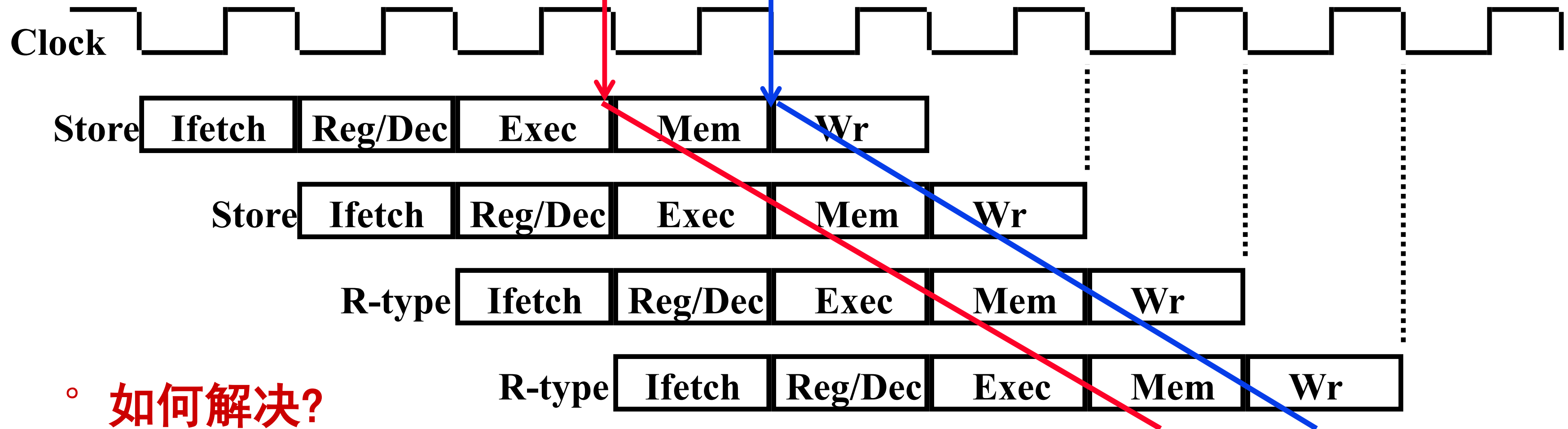




- 在流水线中存在地址 和 写使能之间的“竞争”问题
  - Wr段开始时若RegAdr's(Rd/Rt)Clk-to-Q > RegWr's Clk-to-Q, 则错写寄存器！
  - Mem 阶段开始时若WrAdr's Clk-to-Q > MemWr's Clk-to-Q, 则错写存储器！

（不要求）流水线中的“竞争”问题

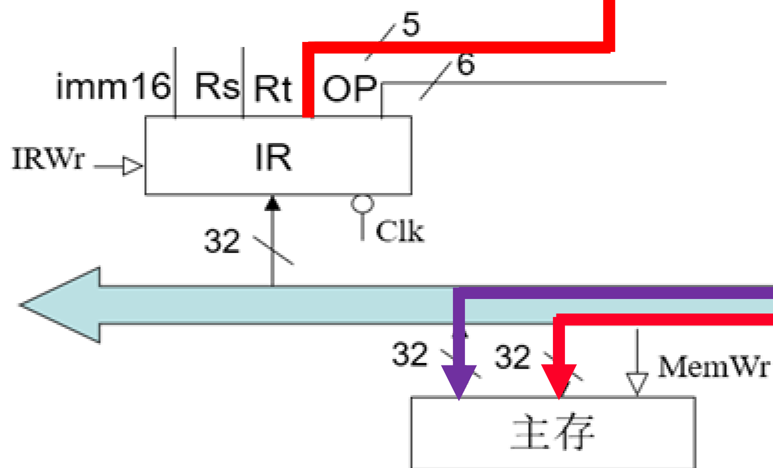
- 多周期中没有出现Addr、data 和 WrEn之间的竞争，因为:
  - 在第 N周期结束时，让Addr和data信号有效
  - 在第 N + 1周期让WrEn有效
- 上述方法在流水线设计中不能用，因为：
  - 每个周期必须能够写Register
  - 每个周期必须能够写Memory



- 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



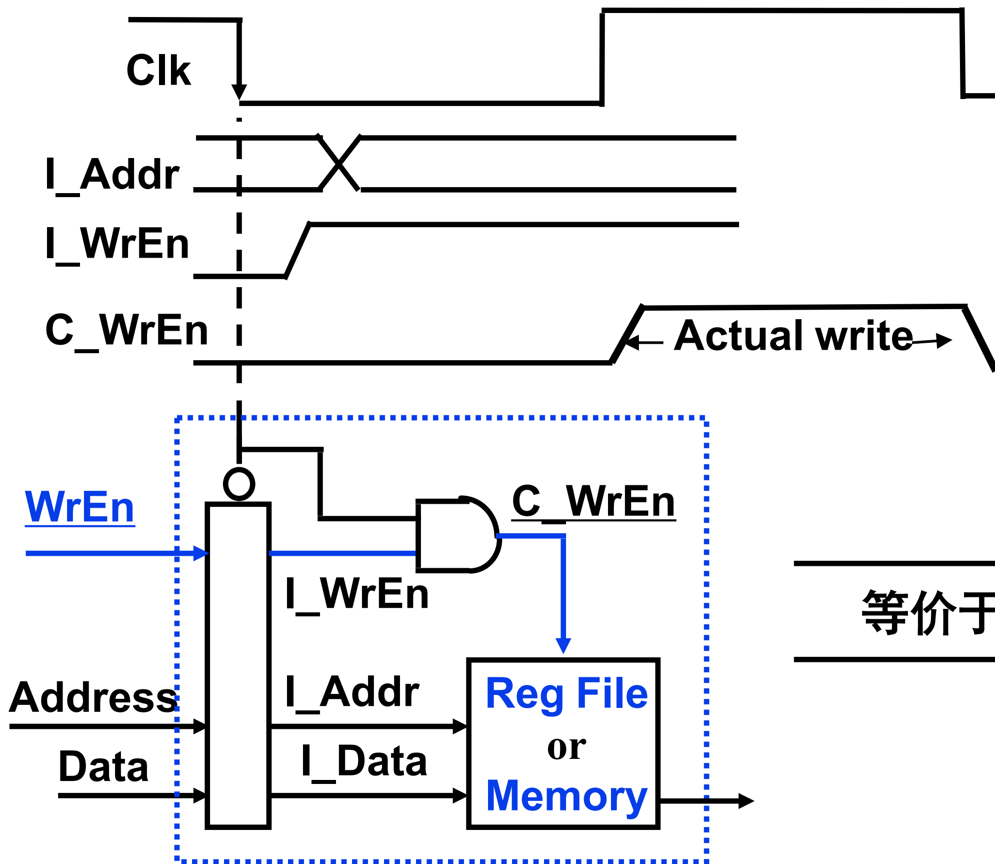
swExec：MDRMUX=1、MDRWr=1、MARout=1、PCout=0、MemWr=0、其他写使能信号全部为0。

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”

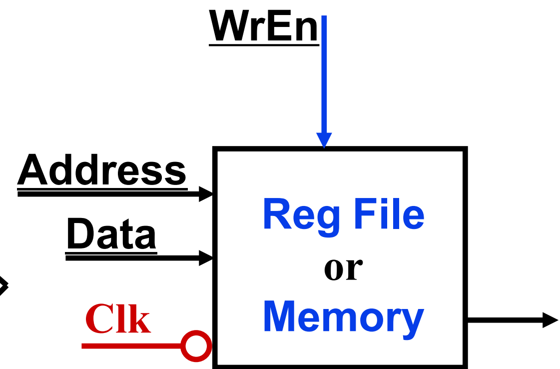
须由电路专家确保不会发生“定时错误”（即：能合理设计“Clock”!）



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

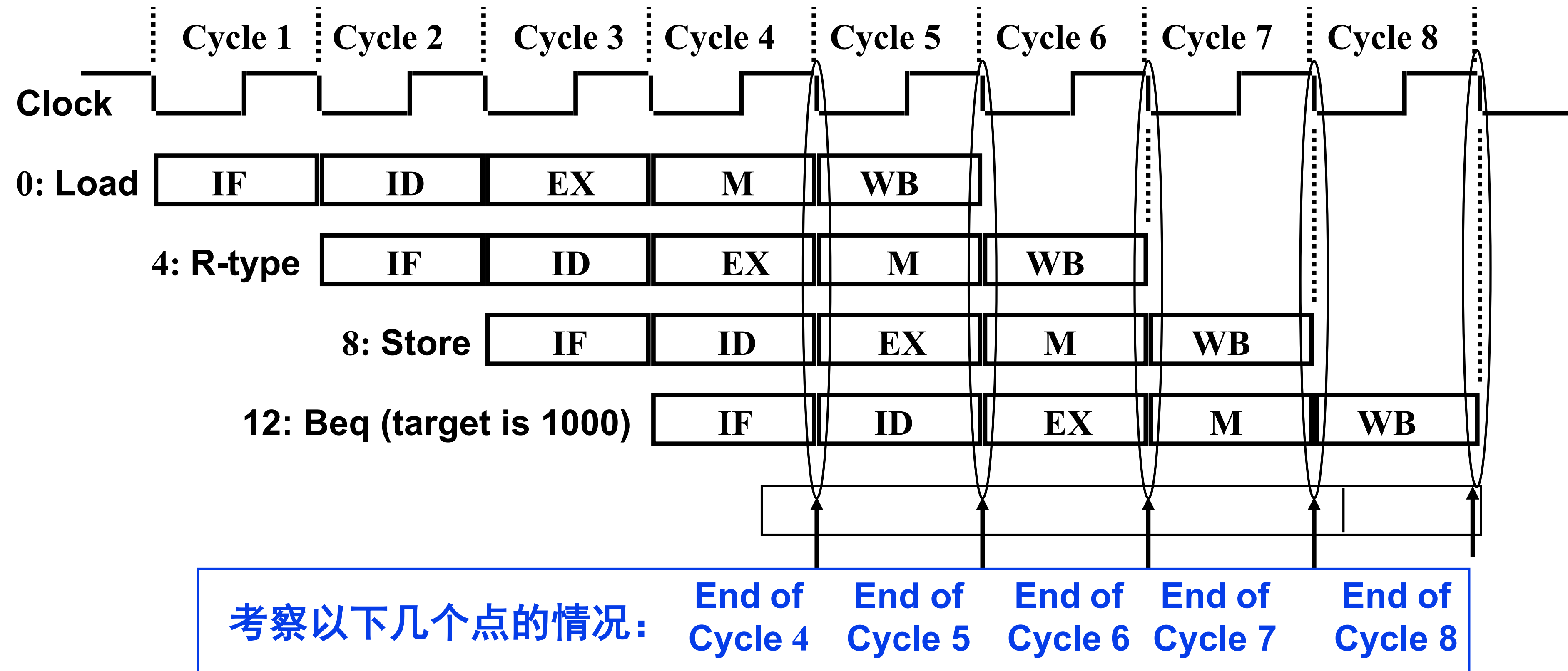
2. Clk高电平时间 大于 写入时间

等价于



相当于单周期通路中的理想寄存器和理想存储器

流水线举例：考察流水线DataPath的数据流动情况

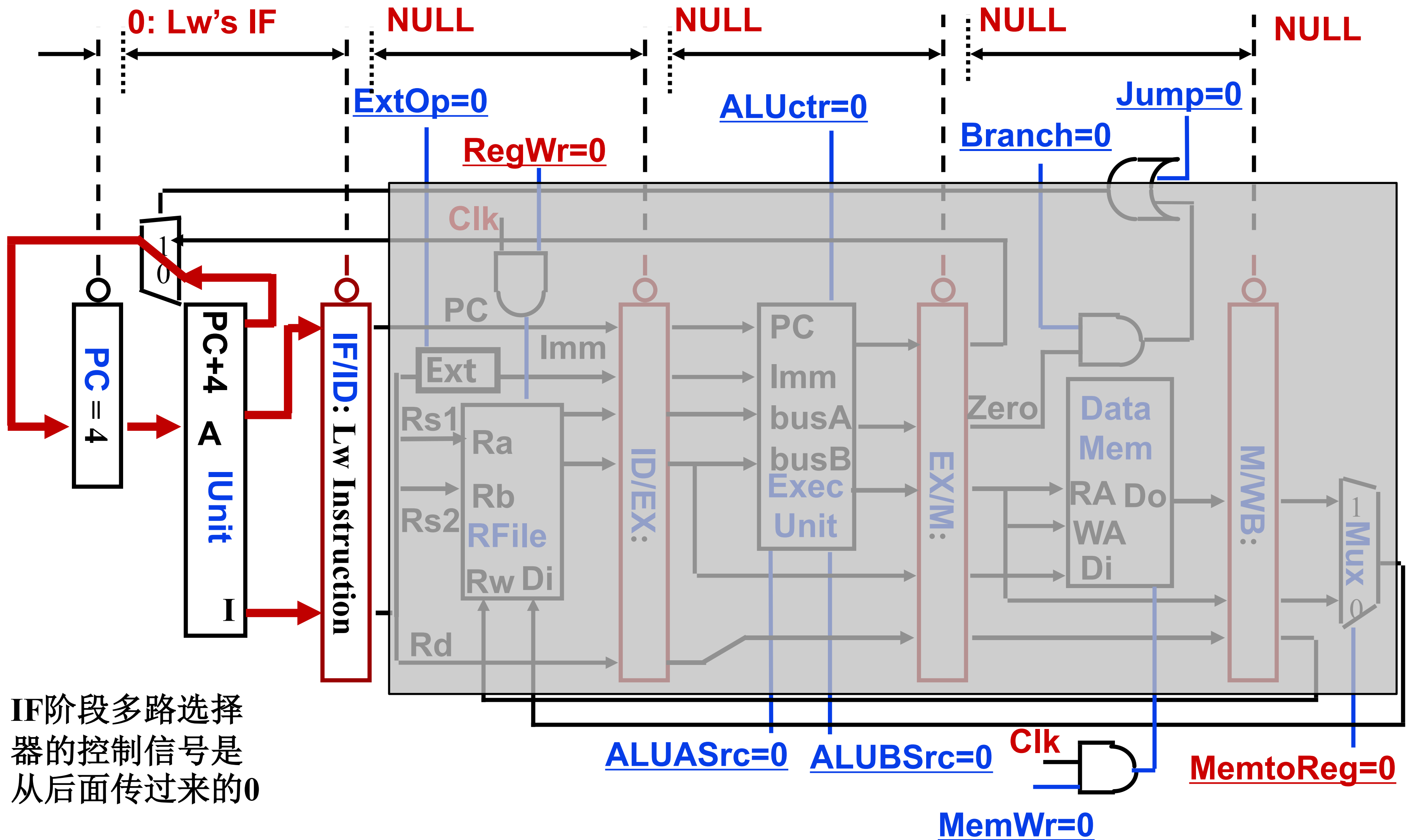


◦ 注意：最开始的时刻，所有流水段寄存器初始化为0！！！

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。

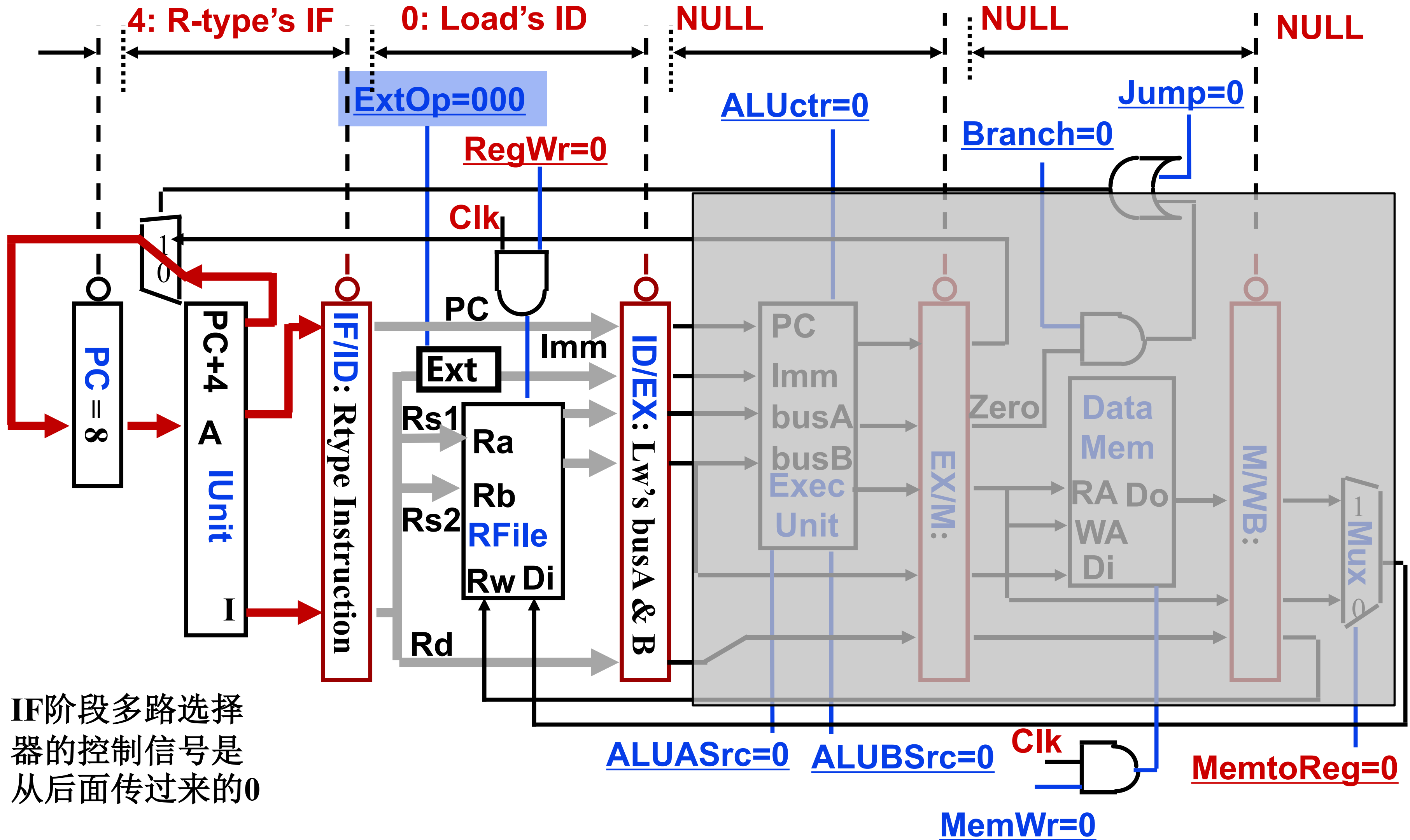
第一周期结束时的状态：

- 0: Load’s IF



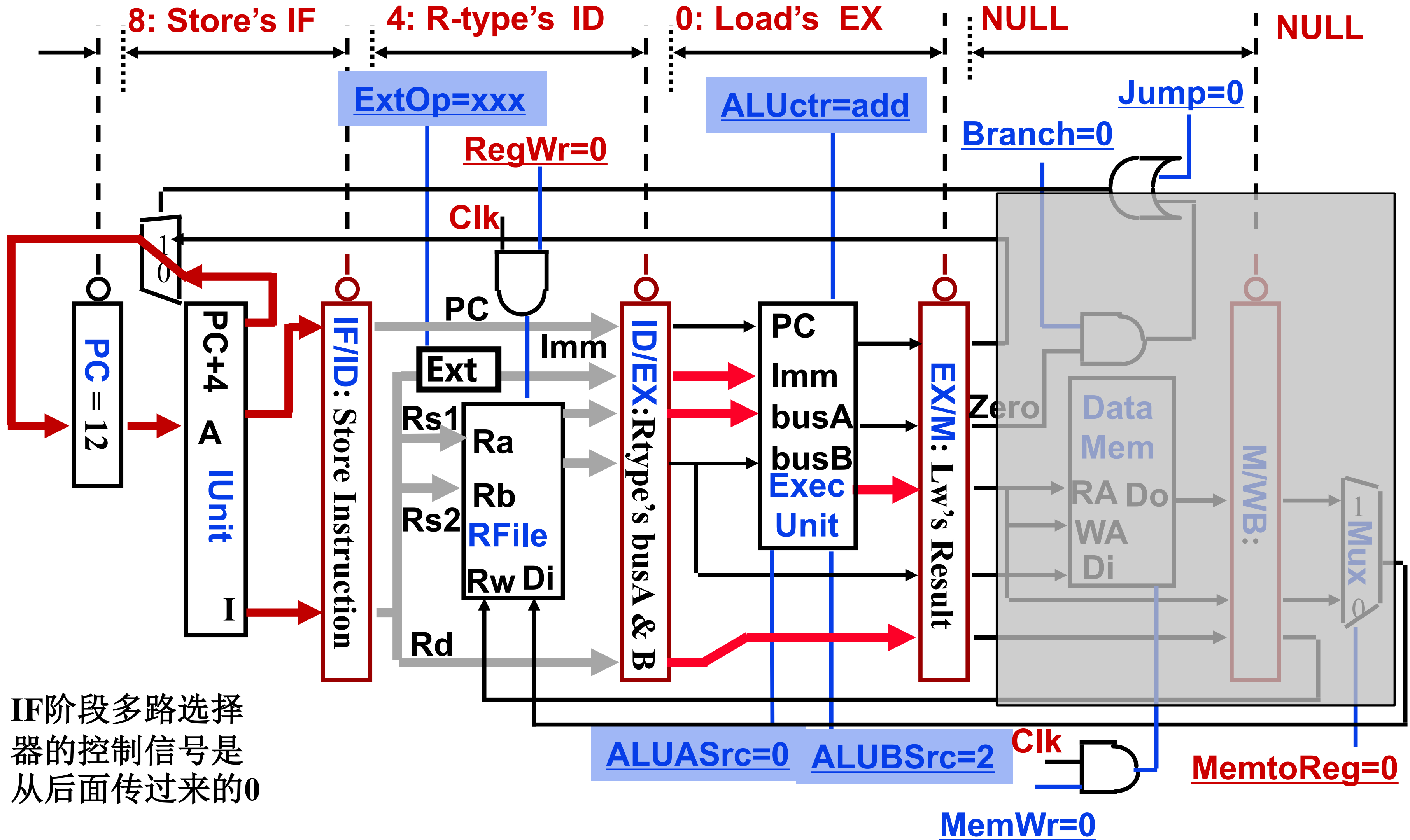
第二周期结束时的状态：

- 4: R-type's IF   0: Load's ID

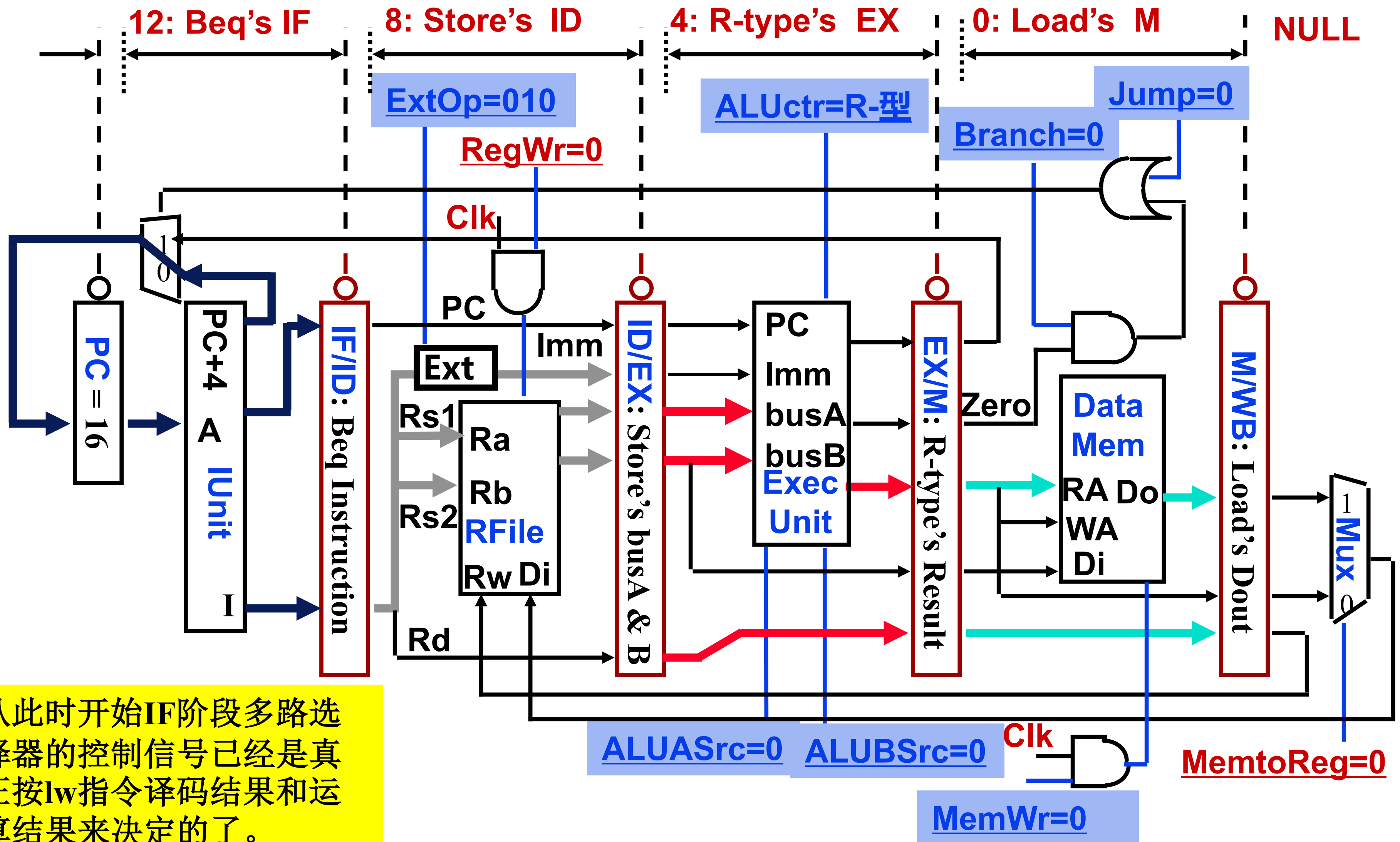


第三周期结束时的状态：

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

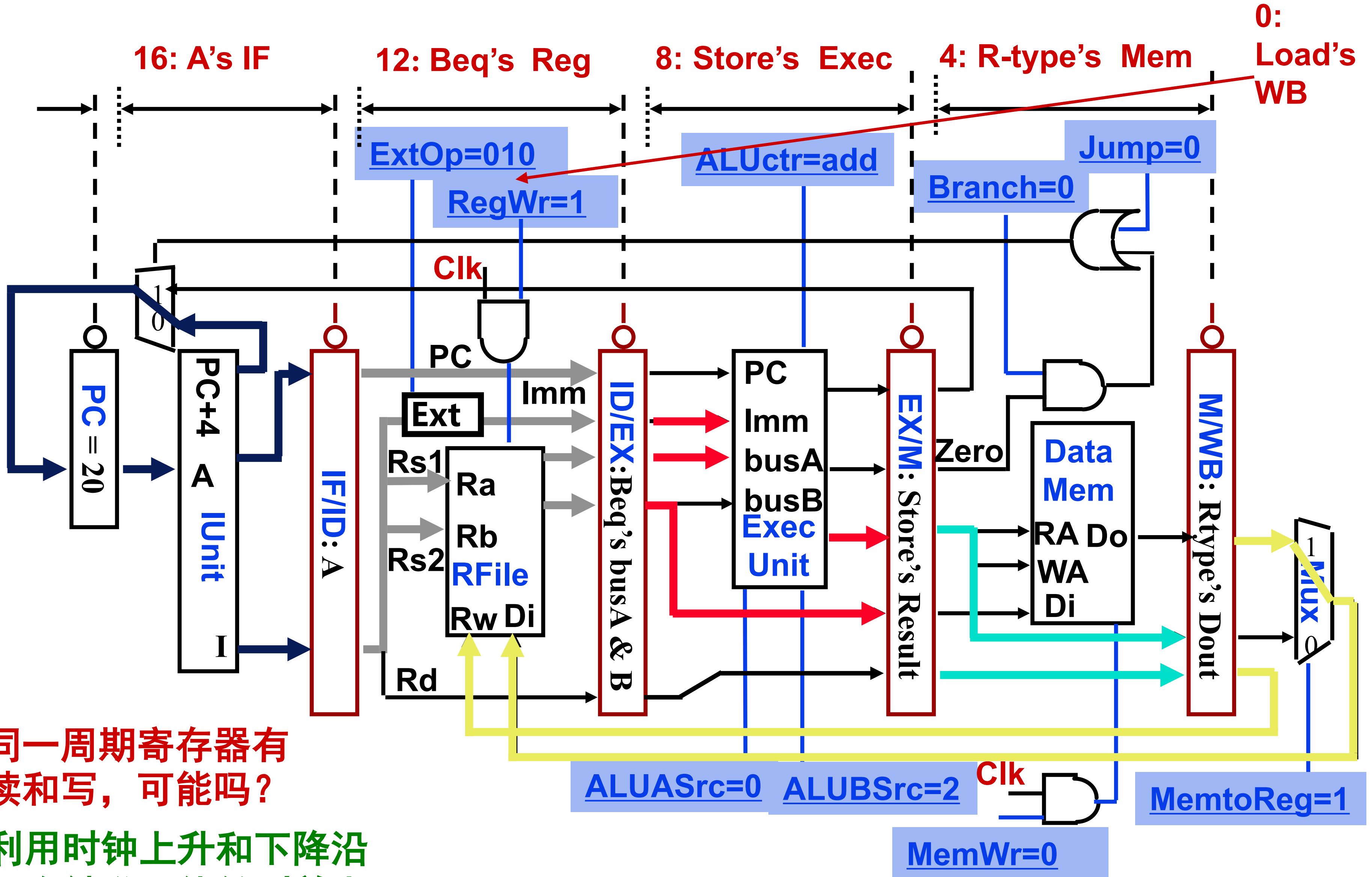


第四周期结束时的状态：



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：

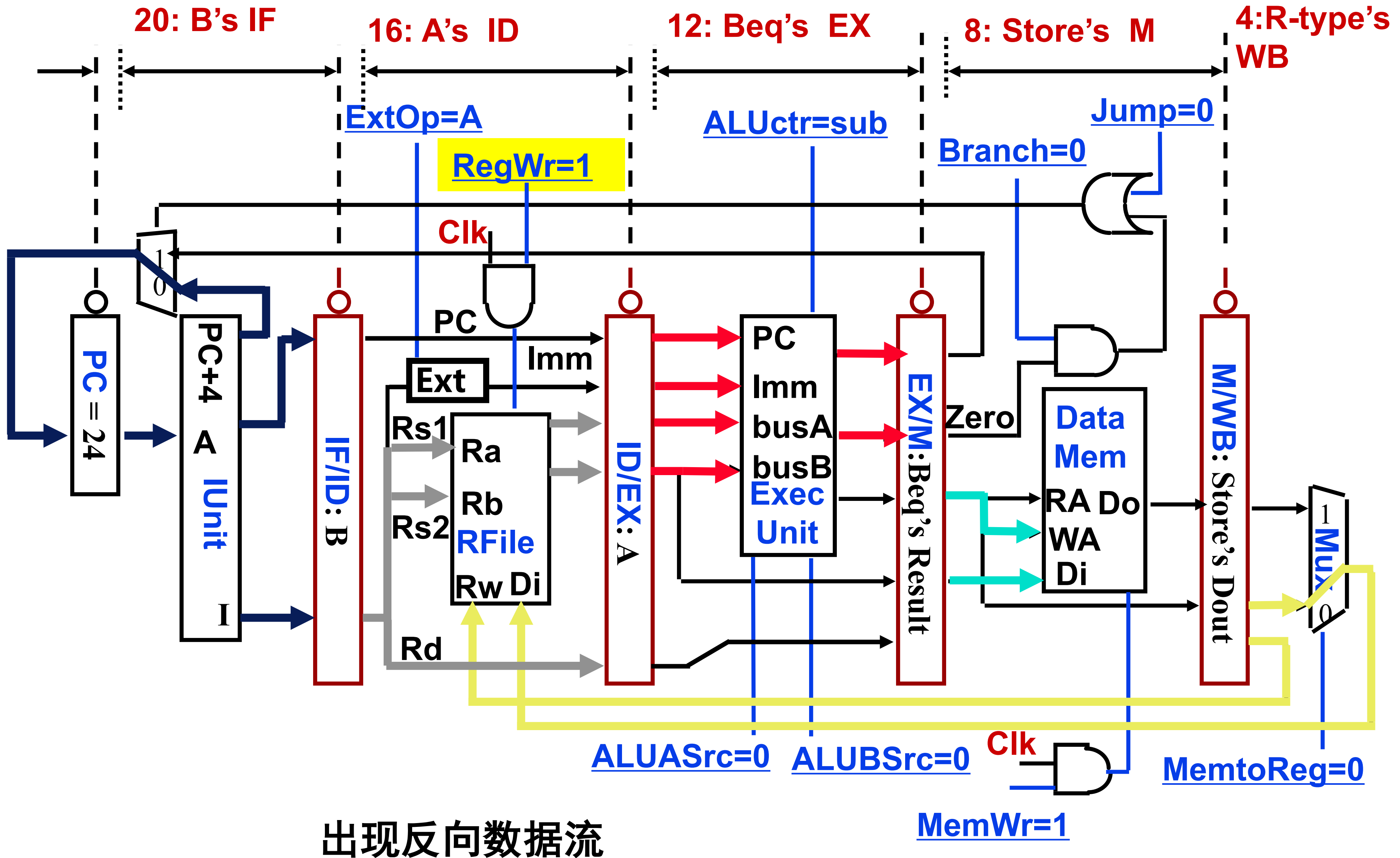


同一周期寄存器有读和写，可能吗？

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

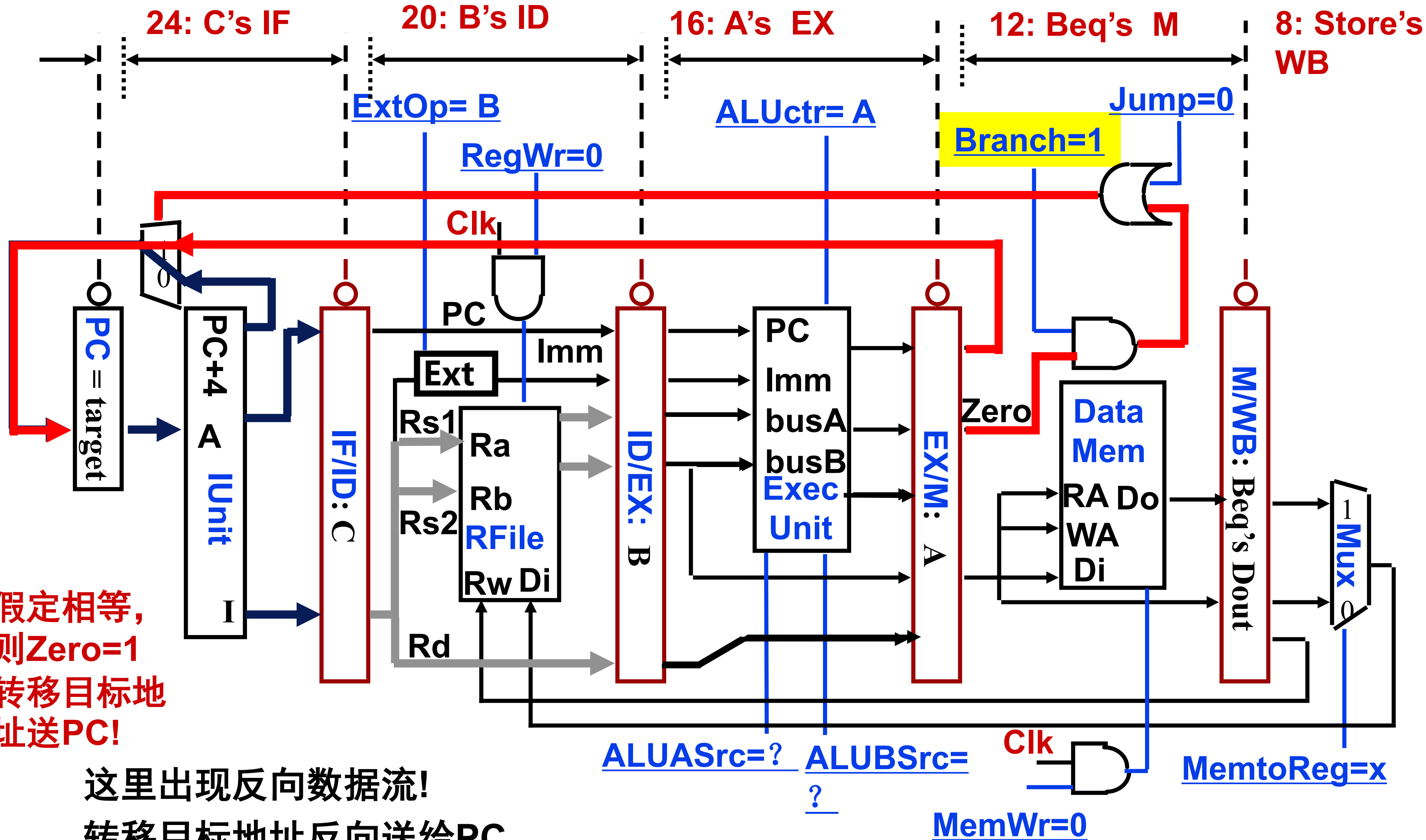
出现反向数据流

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：

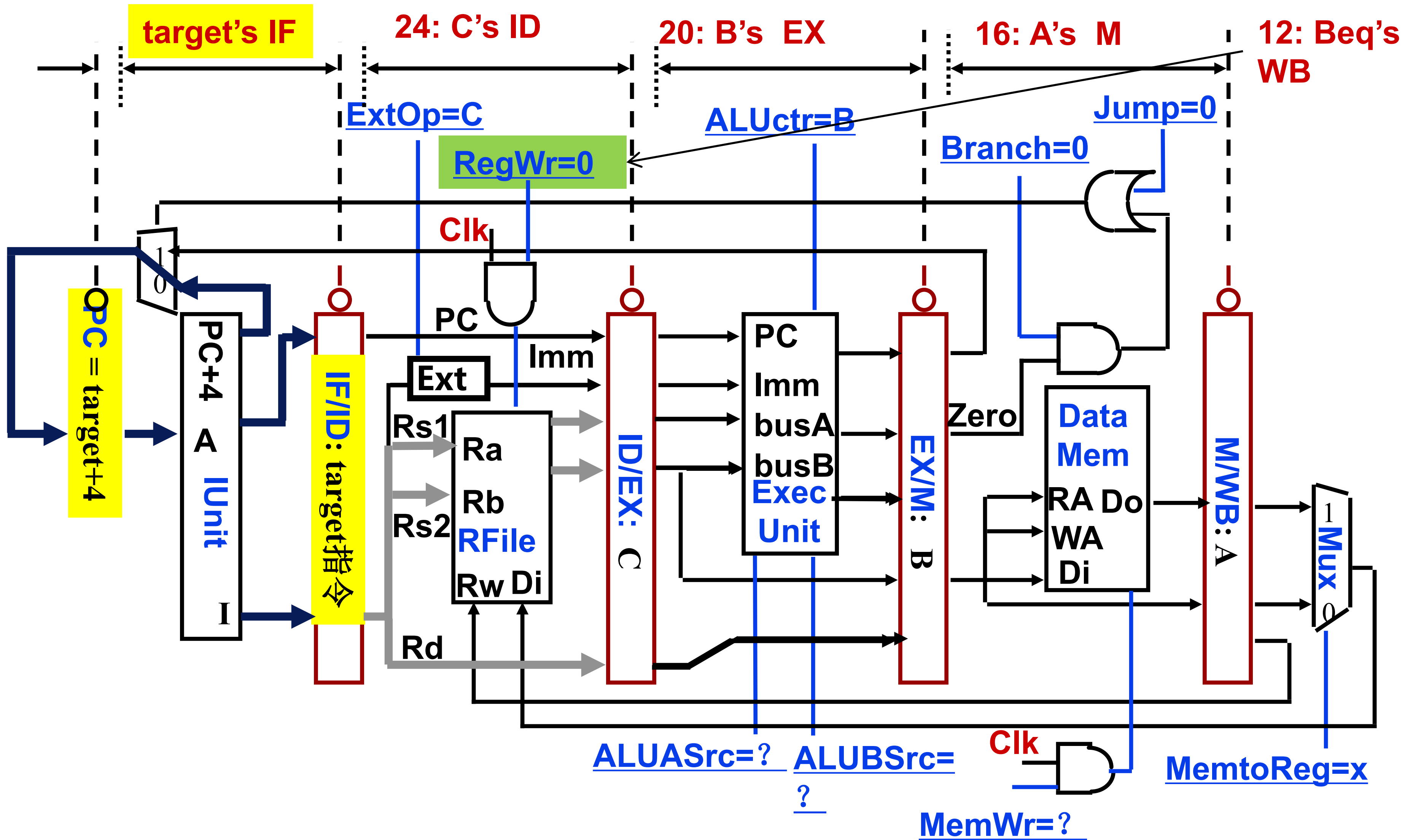


这里出现反向数据流！

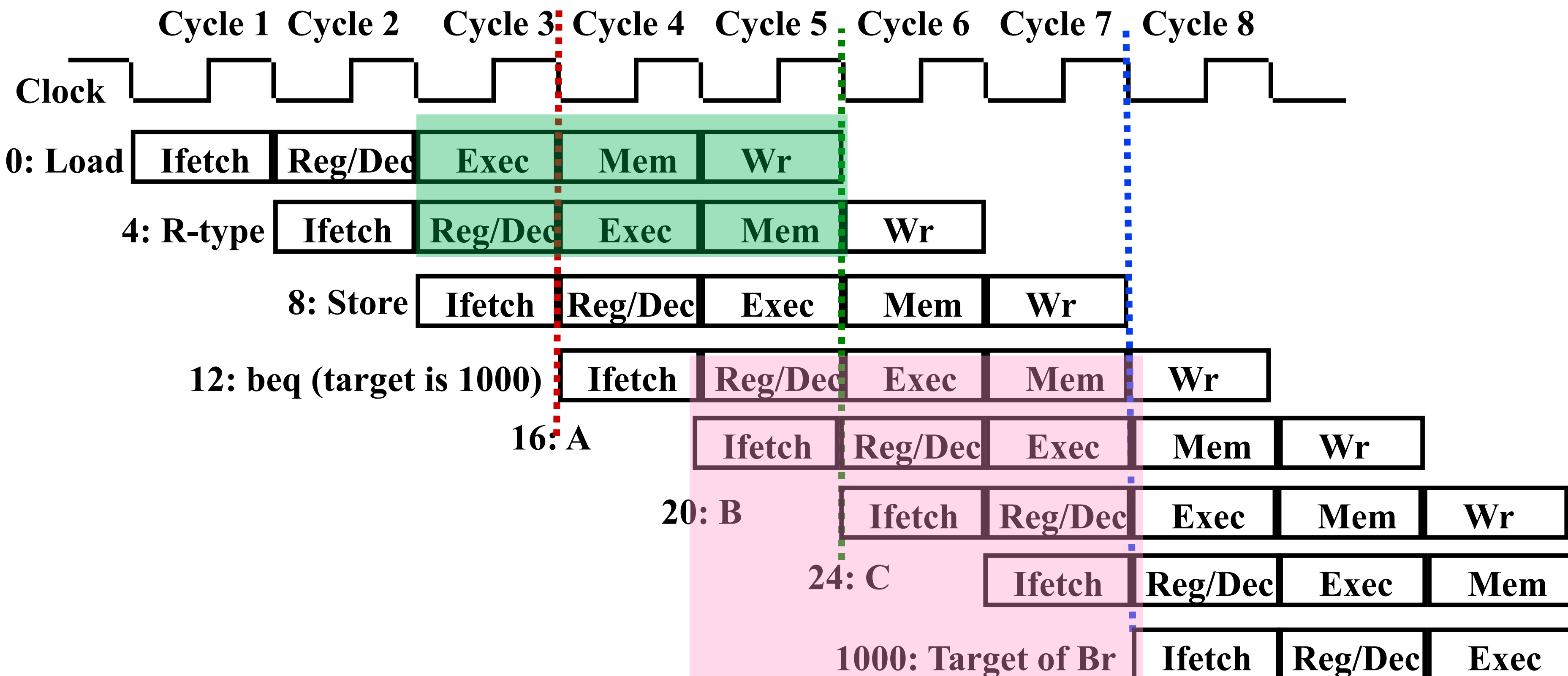
转移目标地址反向送给PC

可能会导致控制冒险！

第8周期结束时的状态：

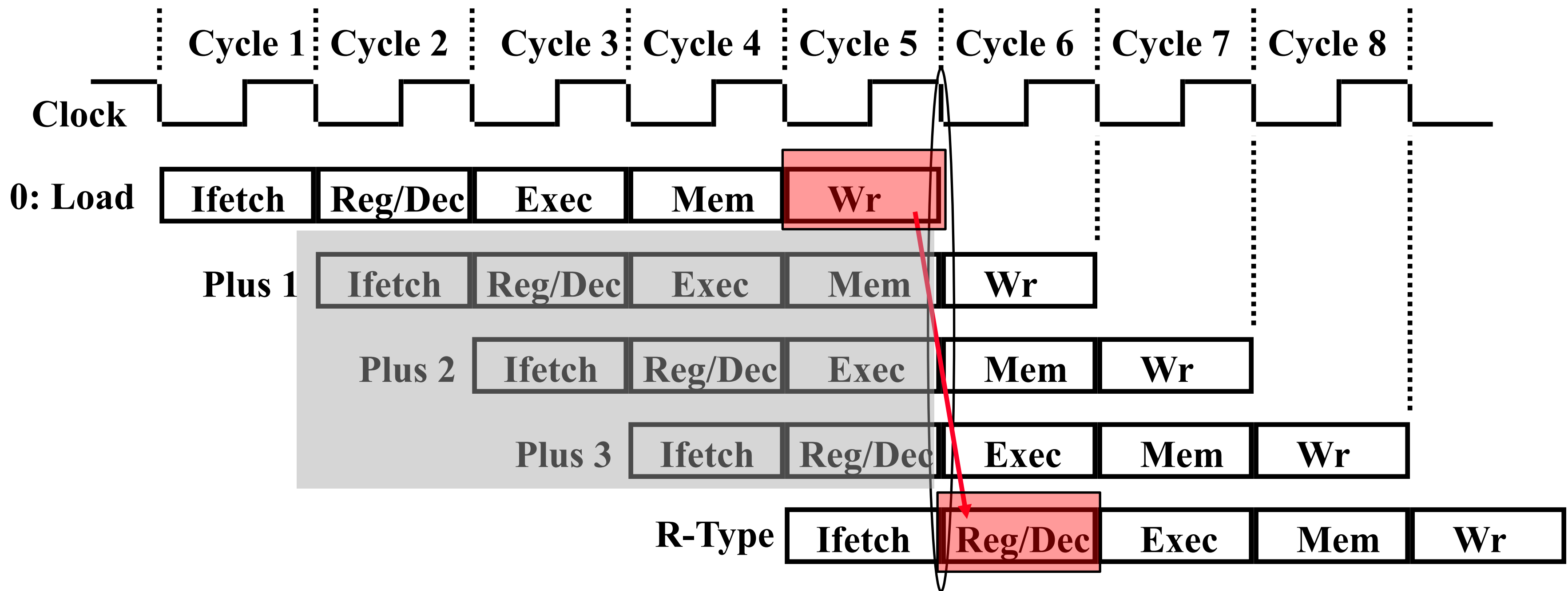


总结前面的流水线执行过程——反向数据流



- **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？

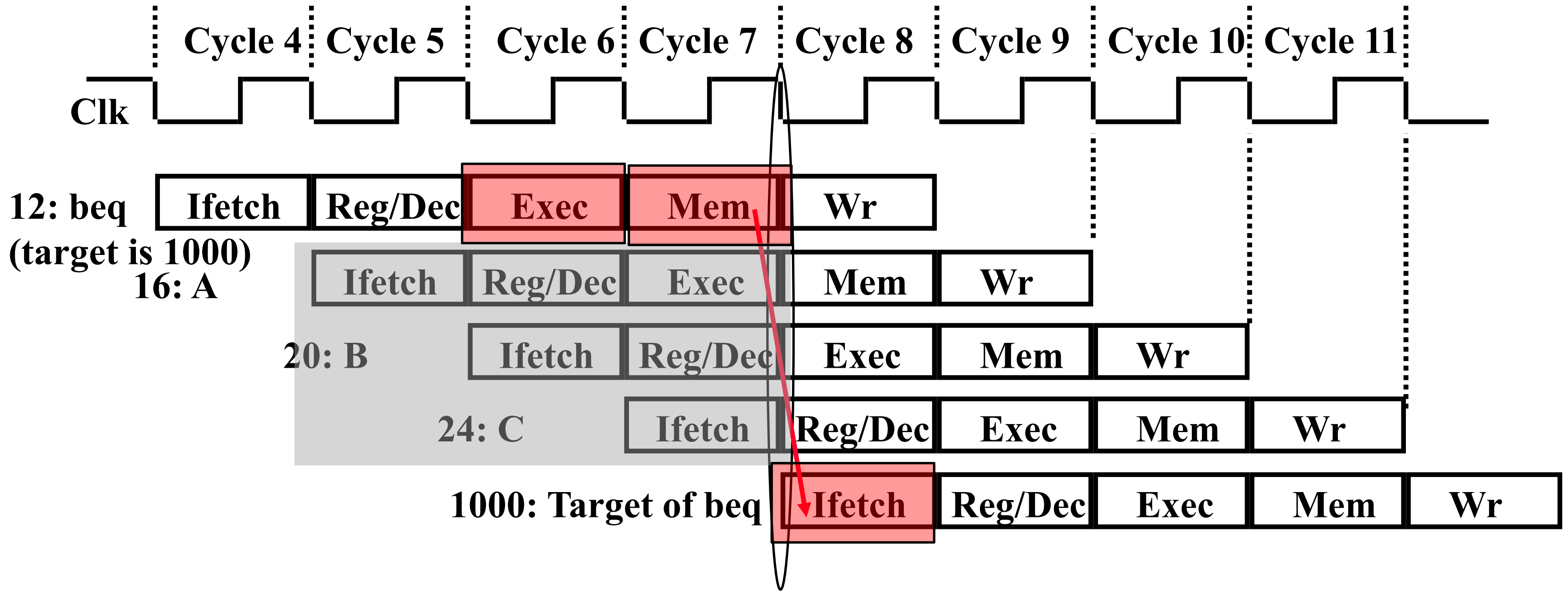
装入指令(Load)引起的“延迟”现象



- **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)**

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 ）

  （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）