Lecture 24: Instruction Pipeline

指令流水线

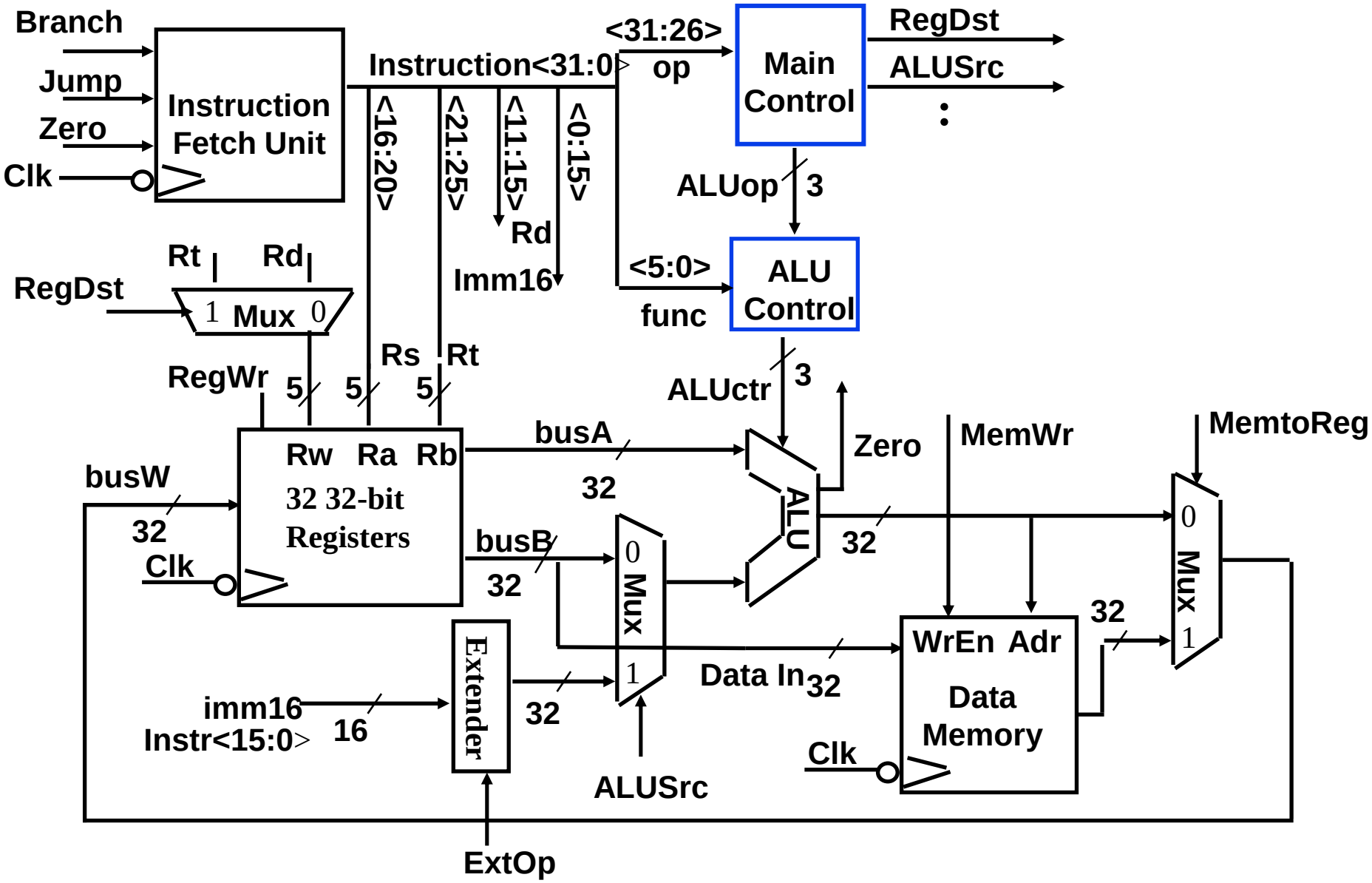
流水线数据通路和控制逻辑

流水线数据通路和控制

主 要 内 容

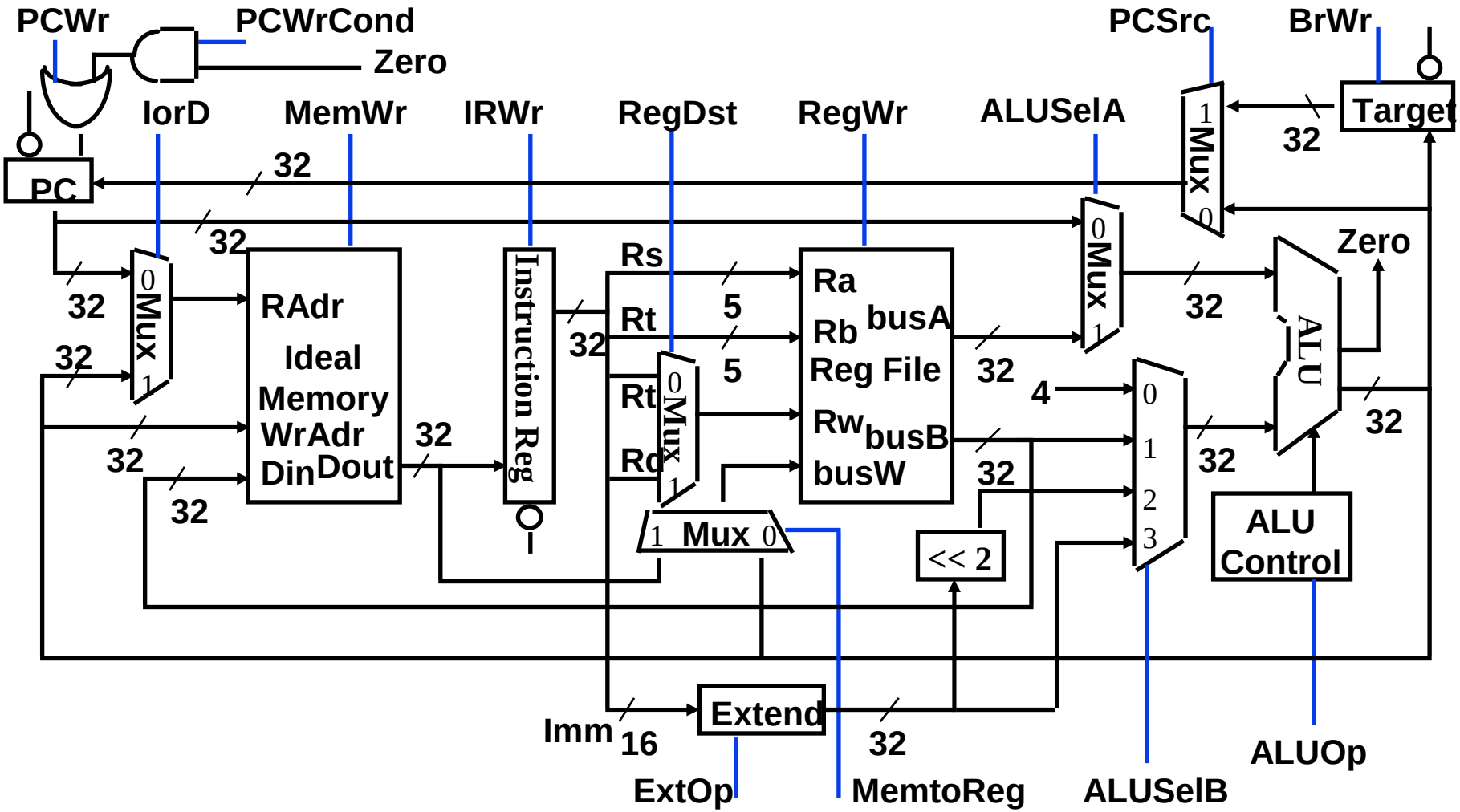
- 日常生活中的流水线处理例子：洗衣服
- 单周期处理器模型和流水线性能比较
- 什么样的指令集适合于流水线方式执行
- 如何设计流水线数据通路
  - 以MIPS指令子集来说明
  - 详细设计取指令部件
  - 详细设计执行部件
  - 分析每条指令在流水线中的执行过程，遇到各种问题：
    - 资源冲突
    - 寄存器和存储器的信号竞争
    - 分支指令的延迟
    - 指令间数据相关
- 如何设计流水线控制逻辑
  - 分析每条指令执行过程中的控制信号
  - 给出控制器设计过程
- 流水线冒险的概念

复习：A Single Cycle Processor

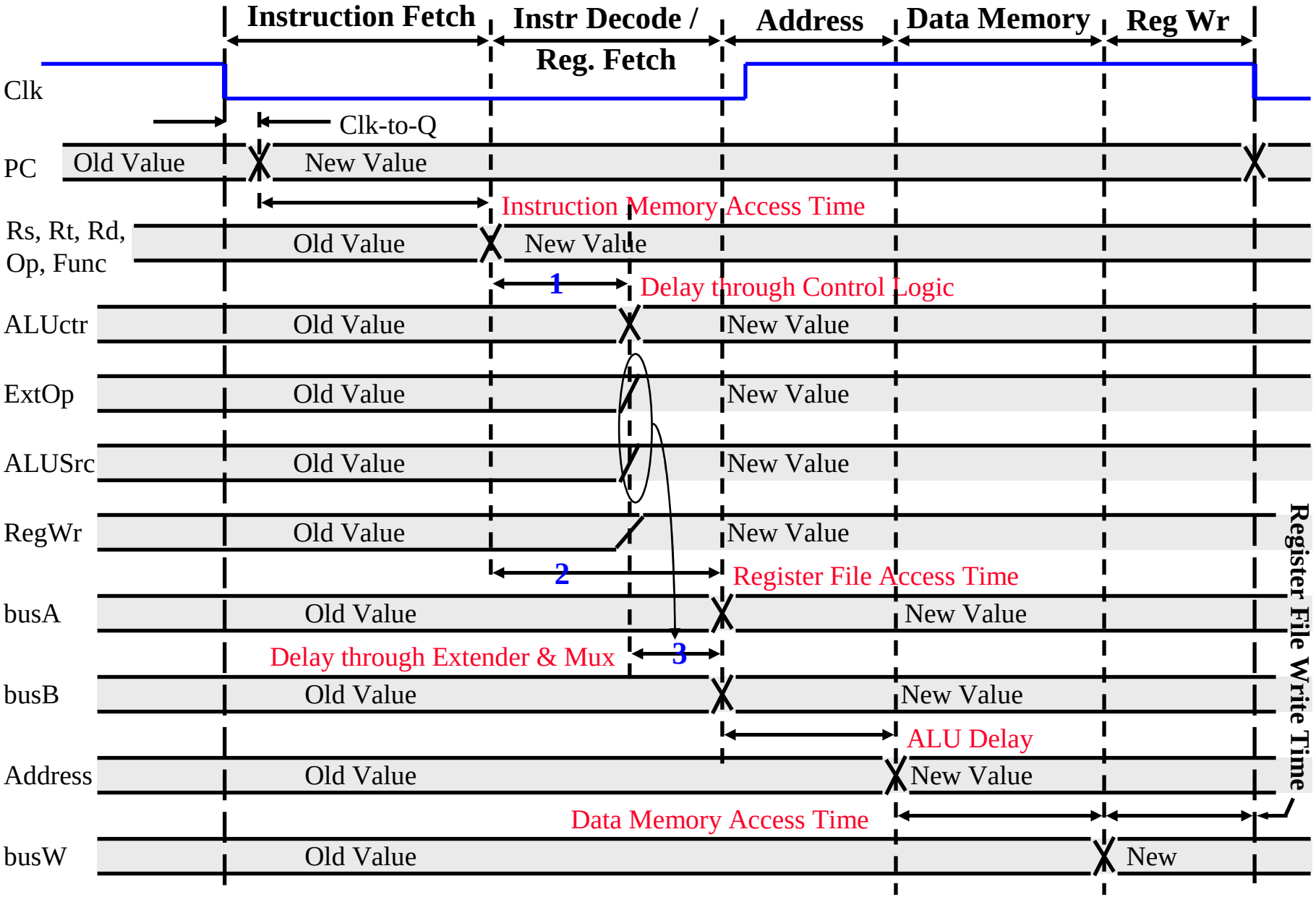


复习：Multiple Cycle Processor

- **MCP:** 一个功能部件在一个指令周期中可以被使用多次。

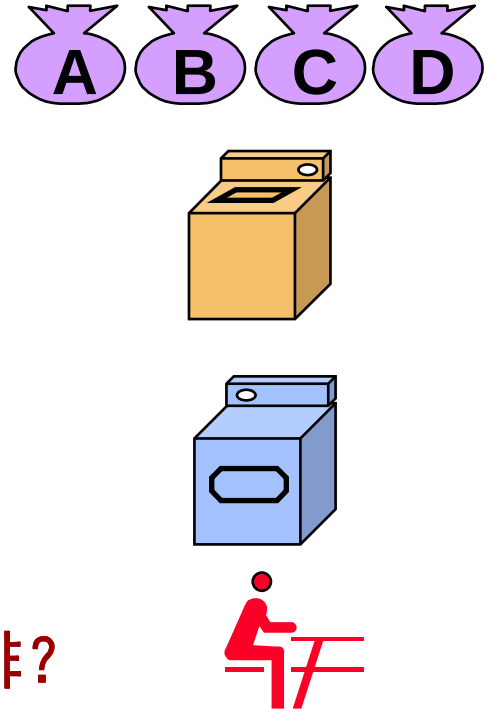


复习：Timing Diagram of a Load Instruction



一个日常生活中的例子—洗衣服

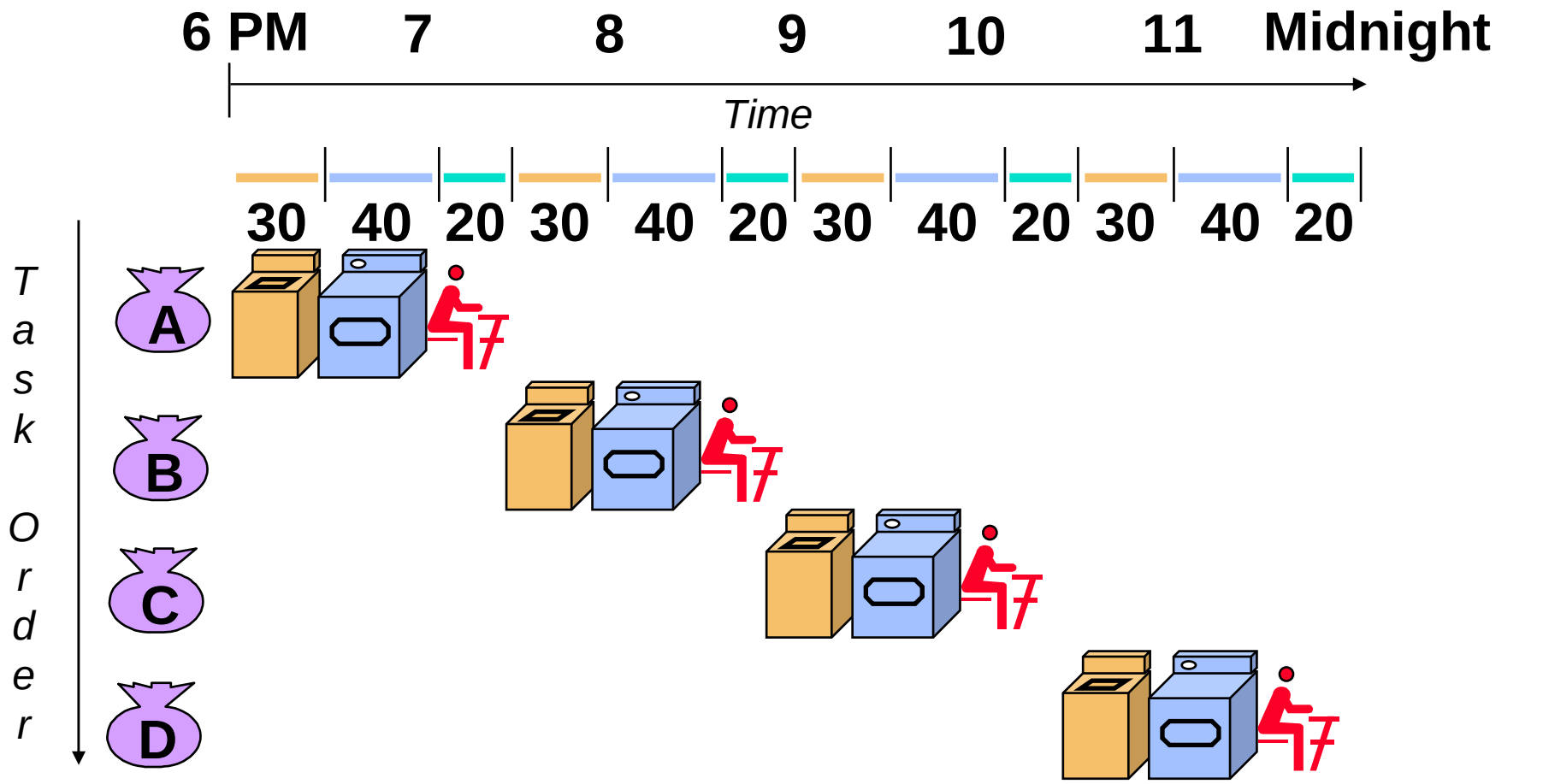
- **Laundry Example**
  - **Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold**
  - **Washer takes 30 minutes**
  - **Dryer takes 40 minutes**
  - **“Folder” takes 20 minutes**



如果让你来管理洗衣店，你会如何安排？

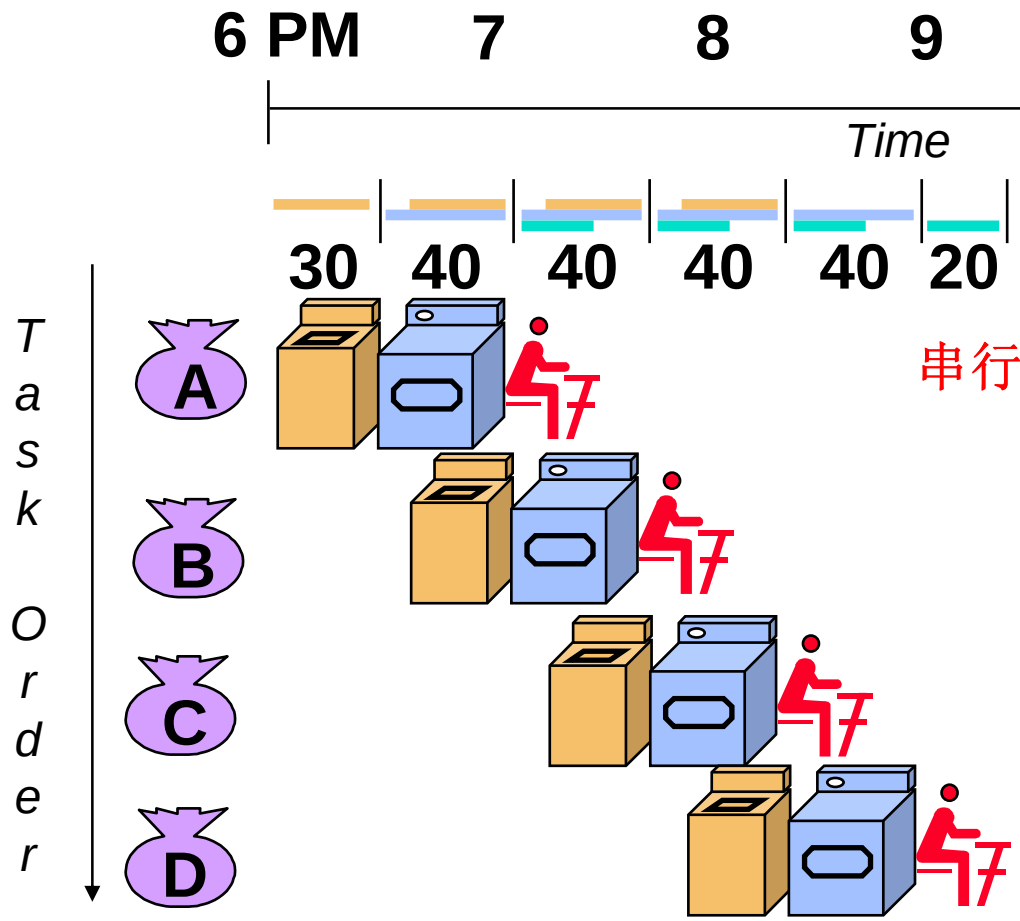
Pipelining: It’s Natural !

Sequential Laundry（串行方式）



- 串行方式下， 4 批衣服需要花费 6 小时（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6小时，N批则为90N分钟

流水线方式下，需

30+4x40+20

=210分 (3.5小时)

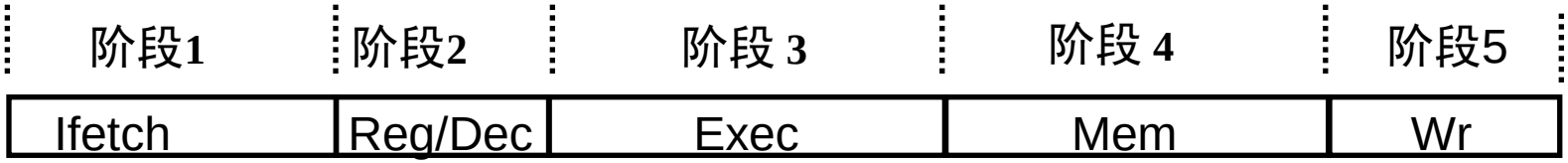
如果有N批衣服呢？

30+Nx40+20分钟

假定每一步时间均衡，则比串行方式提高约**3**倍！

流水方式下，所用时间主要与最长阶段的时间有关！

复习：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

这里寄存器堆的读口和写口可看成两个不同的部件。

指令的执行过程是否和“洗衣”过程类似？是否可以采用类似方式来执行指令呢？

单周期指令模型与流水线性能

- 假定以下每步操作所花时间为：
  - 取指：**2ns**
  - 寄存器读：**1ns**
  - **ALU**操作：**2ns**
  - 存储器读：**2ns**
  - 寄存器写：**1ns**

  Load指令执行时间总计为：**8ns**

  (假定控制单元、**PC**访问、信号传递等没有延迟**)**
- 单周期模型
  - 每条指令在一个时钟周期内完成
  - 时钟周期等于最长的**lw**指令的执行时间，即：**8ns**
  - 串行执行时，**N**条指令的执行时间为：**8Nns**
- 流水线性能
  - 时钟周期等于最长阶段所花时间为：**2ns**
  - 每条指令的执行时间为： **2nsx5=10ns**
  - **N**条指令的执行时间为：**(4+N)x2ns**
  - 在**N**很大时，比串行方式提高约 **4** 倍
  - 若各阶段操作均衡**(**例如，各阶段都是**2ns)**，则提高倍数为**5**倍。

流水线方式下，单条指令执行时间不能缩短，但能大大提高指令吞吐率！

流水线指令集的设计

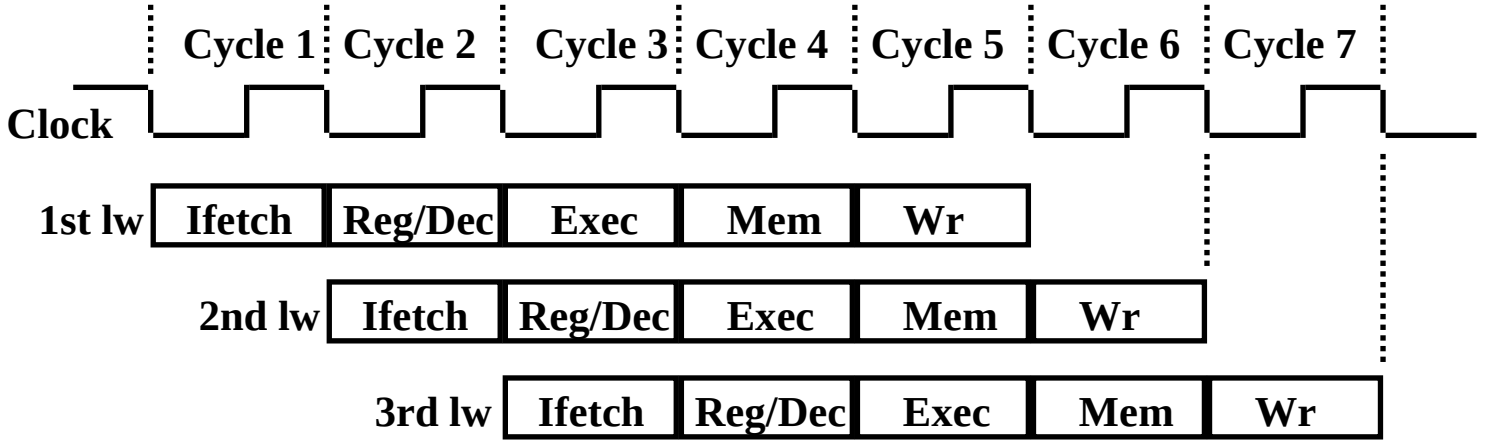
- 具有什么特征的指令集有利于流水线执行呢？
  - 长度尽量一致，有利于简化取指令和指令译码操作
    - **MIPS指令32位，下址计算方便: PC+4**
    - **X86指令从1字节到17字节不等，使取指部件及其复杂**
  - 格式少，且源寄存器位置相同，有利于在指令未知时就可取操作数
    - **MIPS指令的Rs和Rt位置一定，在指令译码时就可读Rs和Rt的值**

    （若位置随指令不同而不同，则需先确定指令后才能取寄存器编号）
  - **load / Store指令才能访问存储器，有利于减少操作步骤，规整流水线**
    - **lw/sw指令的地址计算和运算指令的执行步骤规整在同一个周期**
    - **X86运算类指令操作数可为内存数据，需计算地址、访存、执行**
  - 内存中”对齐”存放，有利于减少访存次数和流水线的规整

总之，规整、简单和一致等特性有利于指令的流水线执行

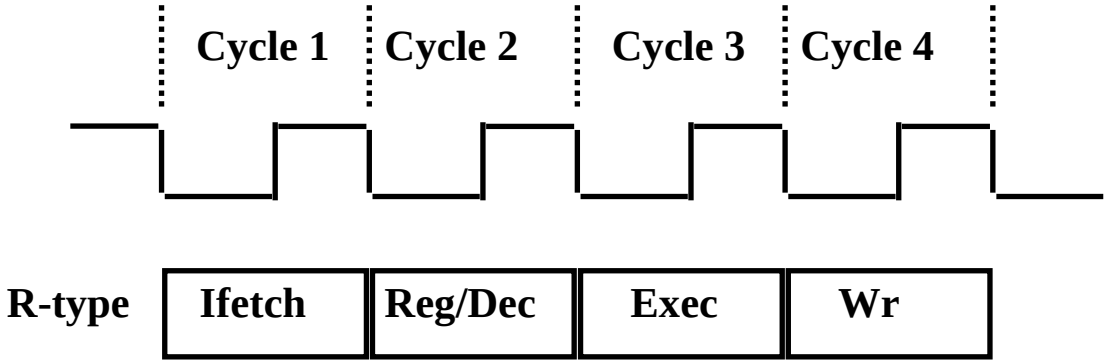
流水线执行方式能大大提高指令吞吐率，现代计算机都采用流水线方式！

Load指令的流水线



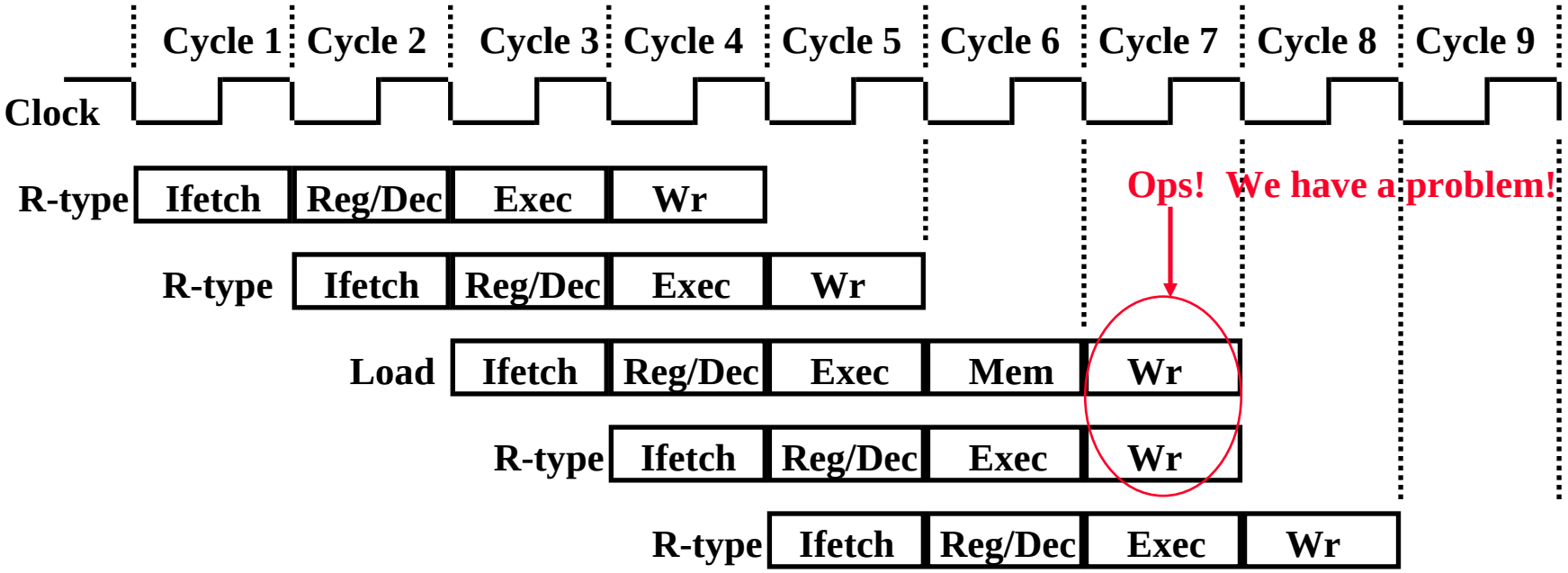
- 每个周期有五个功能部件同时在工作
- 后面指令在前面完成取指后马上开始
- 每个**load**指令仍然需要五个周期完成
- 但是吞吐率**(throughput)**提高许多，理想情况下，有：
  - 每个周期有一条指令进入流水线
  - 每个周期都有一条指令完成
  - 每条指令的有效周期**(CPI)**为**1**

R-type指令的4个阶段



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

含R-type和 Load 指令的流水线

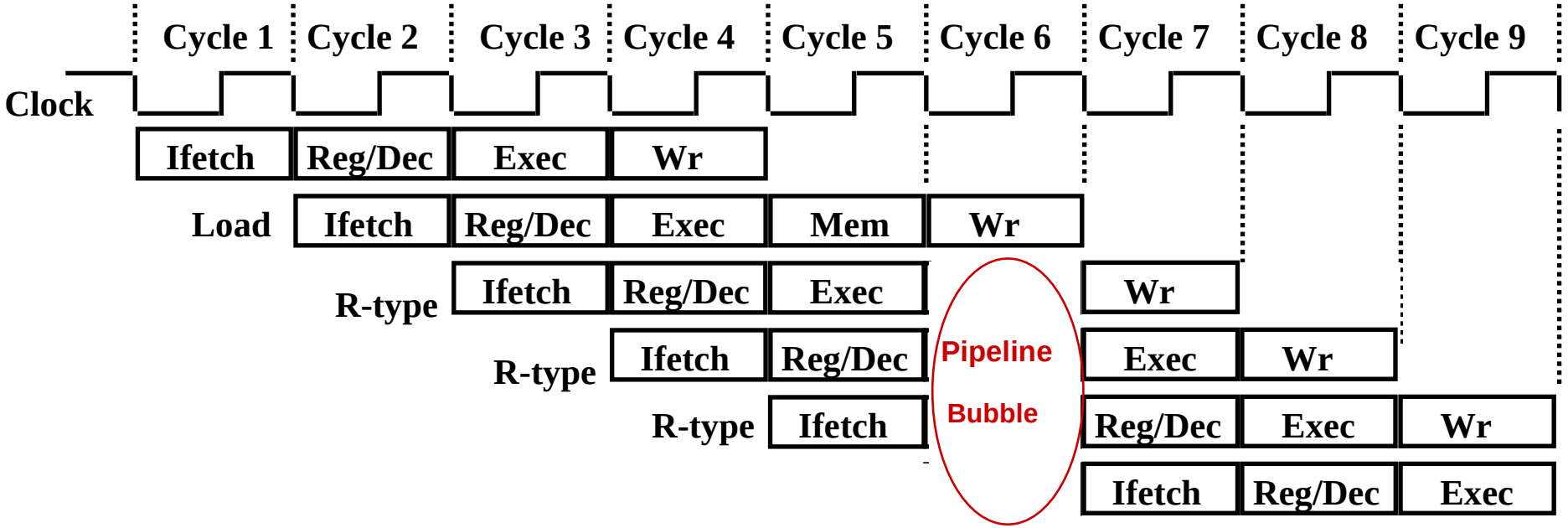


- 上述流水线有个问题: 两条指令试图同时写寄存器，因为
  - Load在第5阶段用寄存器写口
  - R-type在第4 阶段用寄存器写口

  或称为资源冲突!
- 把一个功能部件同时被多条指令使用的现象称为结构冒险(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）

可以用以下两种方法解决上述结构冒险问题！

解决方案1: 在流水线中插入“Bubble”（气泡）



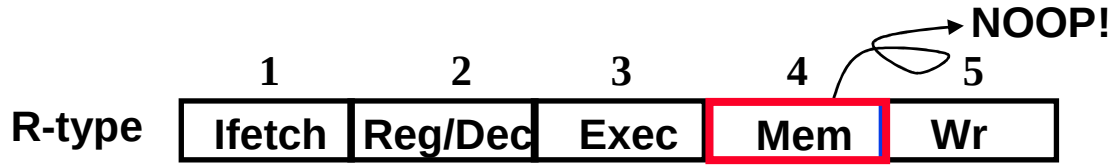
- 插入“Bubble”到流水线中，以禁止同一周期有两次写寄存器。

  缺点：

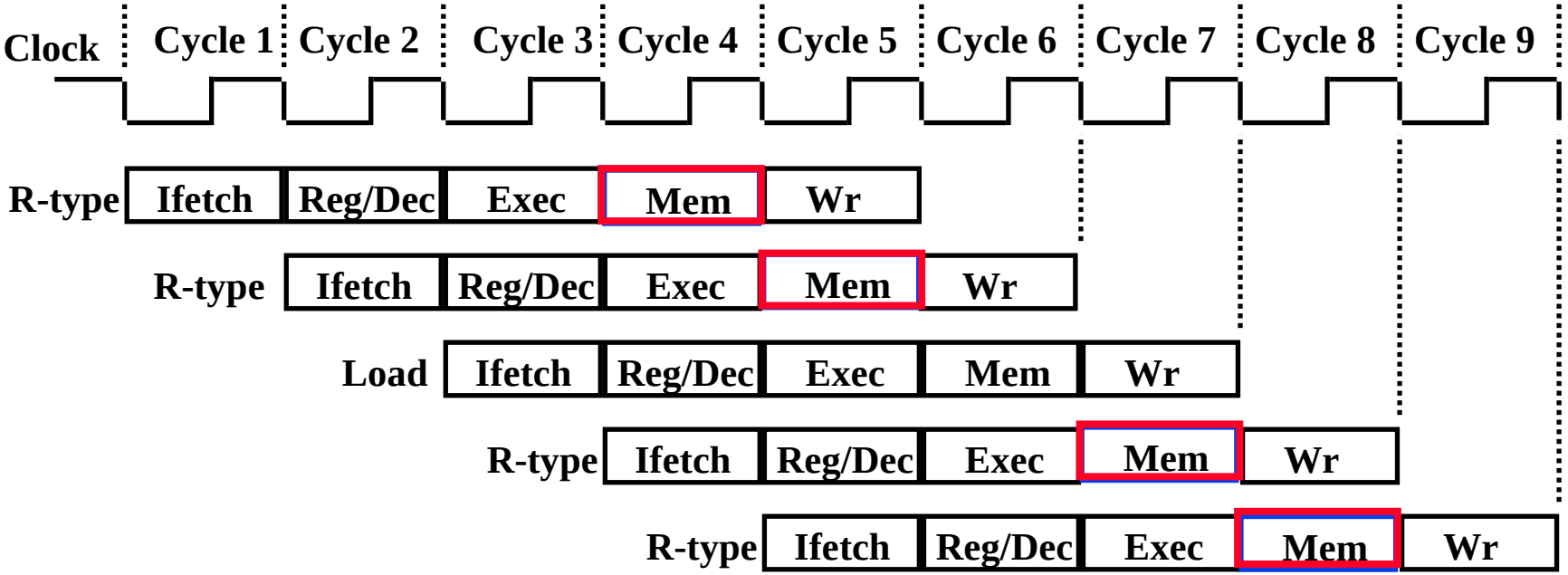
  - 控制逻辑复杂
  - 第5周期没有指令被完成（CPI不是1，而实际上是2）

方案不可行！

解决方案2: R-type的Wr操作延后一个周期执行

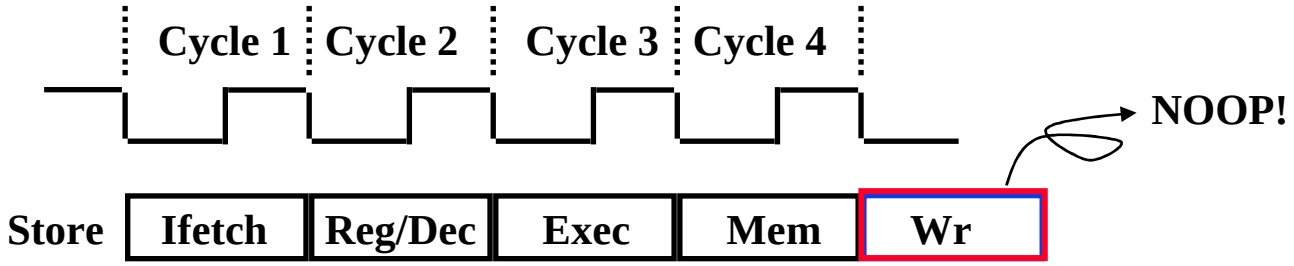


- 加一个NOP阶段以延迟“写”操作:
  - 把“写”操作安排在第5阶段, 这样使R-Type的Mem阶段为空NOP



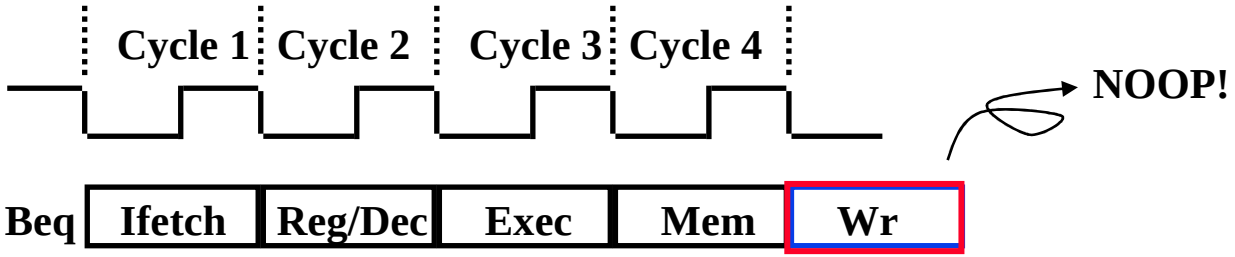
这样使流水线中的每条指令都有相同多个阶段!

Store指令的四个阶段



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

Beq的四个阶段



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

与多周期通路有什么不同？

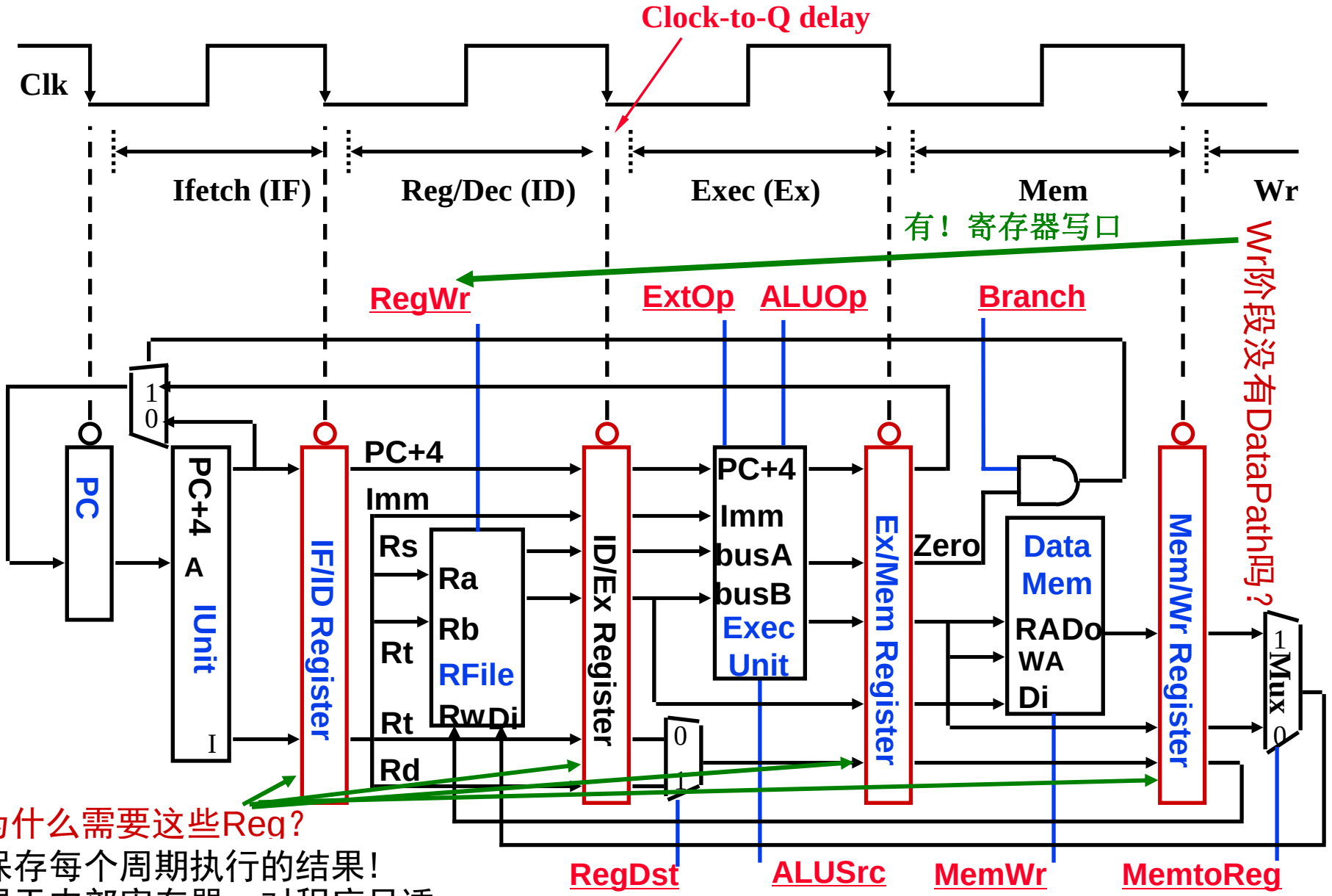
多周期通路中，在Reg/Dec阶段投机进行了转移地址的计算！可以减少Branch指令的时钟数

为什么流水线中不进行“投机”计算？

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

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

A Pipelined Datapath（五阶段流水线数据通路）



为什么需要这些Reg?

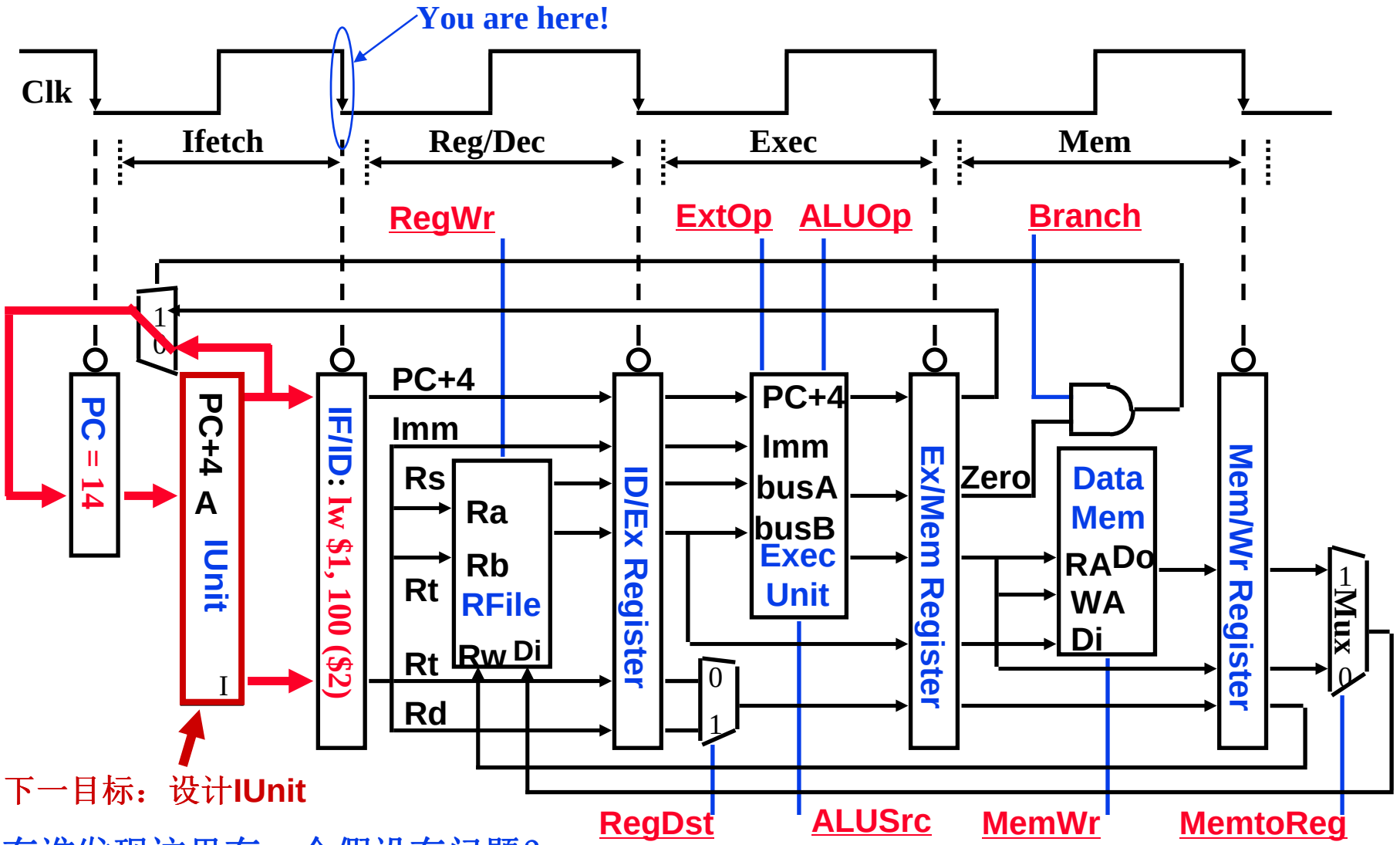
保存每个周期执行的结果！

属于内部寄存器，对程序员透明，不需作为现场保存

下面看一下每条指令在流水线通路中的执行过程

取指令（Ifetch) 阶段

◦ 第10单元指令: lw $1, 0x100($2)　　功能：$1 <- Mem [($2) + 0x100]



下一目标：设计IUnit

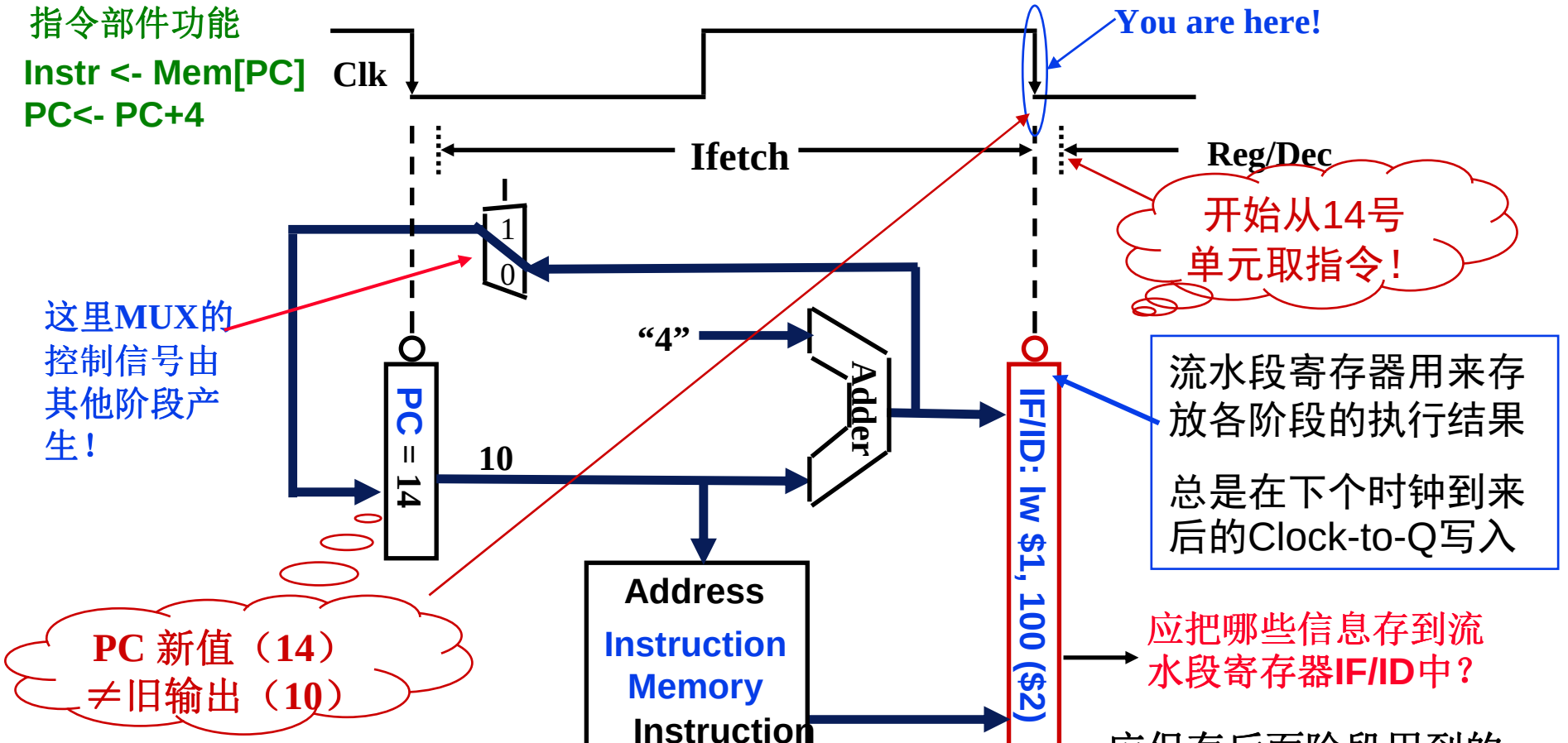
有谁发现这里有一个假设有问题？

MIPS指令的地址可能是10吗？

指令部件 IUnit的设计

- 第10单元指令: : lw $1, 0x100($2)

随后的指令在14号单元中！

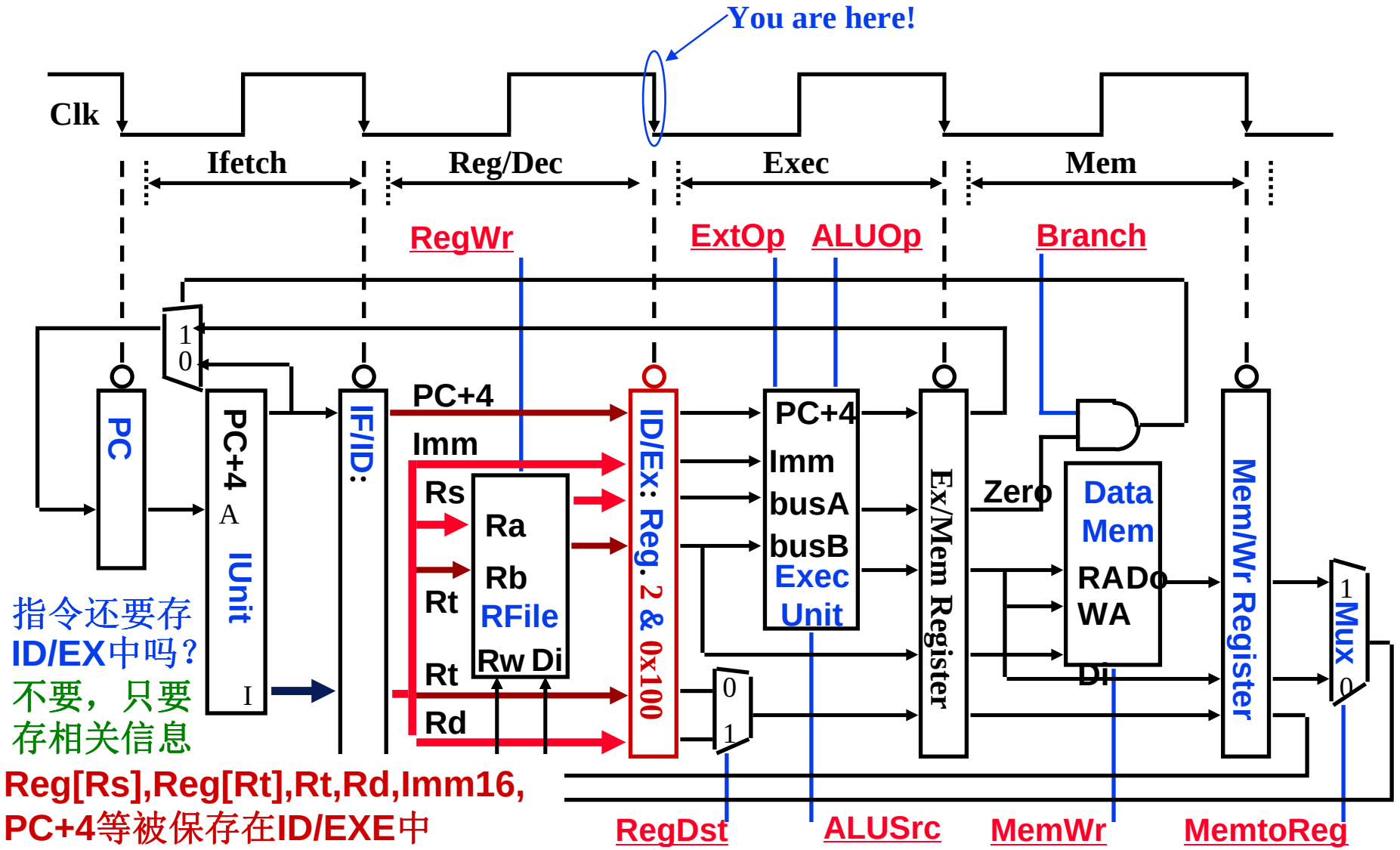


不需控制信号，因为每条指令执行功能一样，是确定的，无需根据指令的不同来控制执行不同的操作！

指令在随后阶段被送出译码！
PC+4用来计算转移目标地址！

译码/取数（Reg/Dec）阶段

° **Location 10**: lw $1, 0x100($2)　　功能：$1 <- Mem[($2) + 0x100]

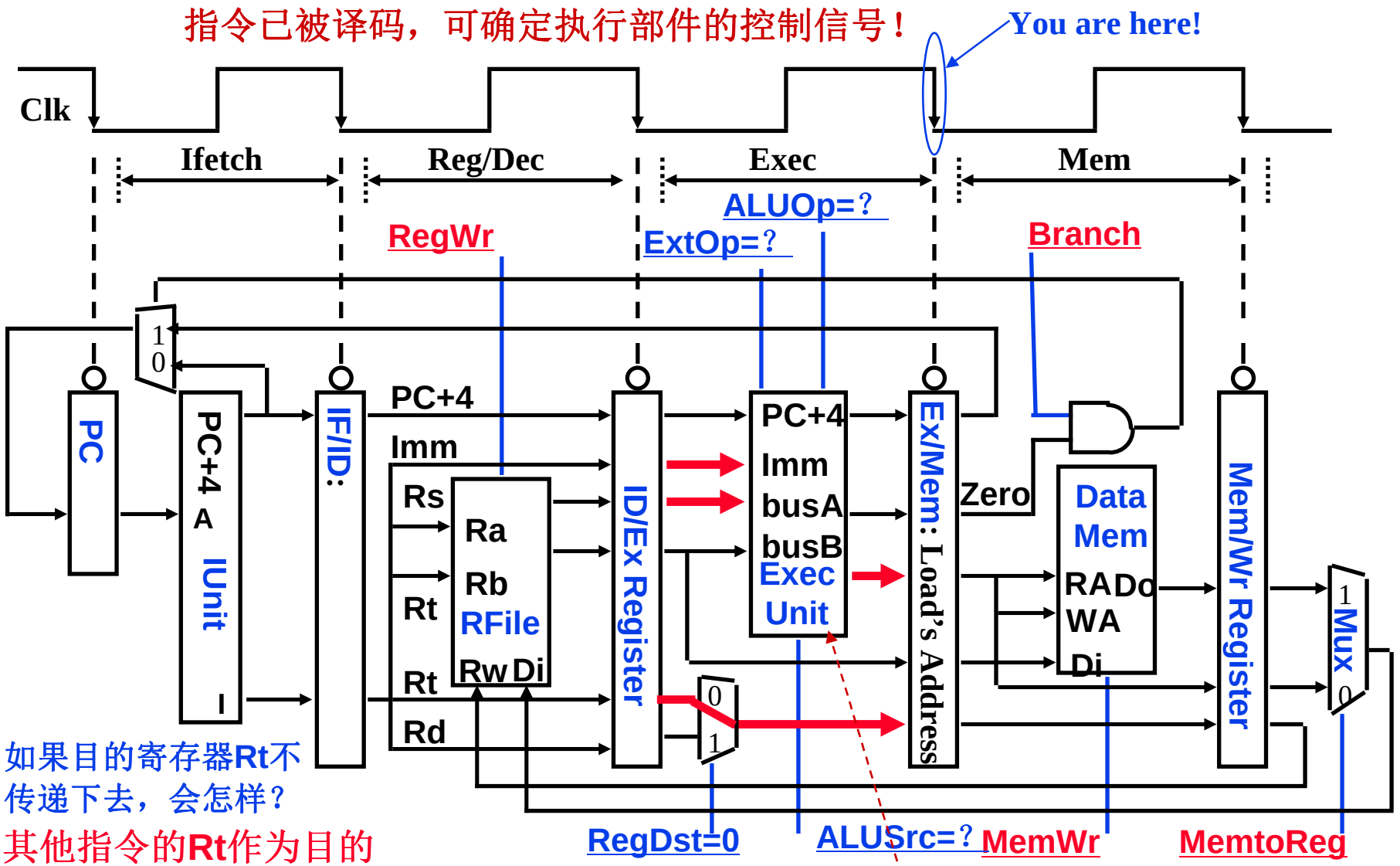


该阶段有哪些控制信号？　没有！因是所有指令的公共操作，故无控制信号！

Load指令的地址计算阶段

° **Location 10:** lw $1, 0x100($2)　　功能：$1 <- Mem[($2) + 0x100]

指令已被译码，可确定执行部件的控制信号！



如果目的寄存器Rt不传递下去，会怎样？

其他指令的Rt作为目的地址，指令执行错误！

下一目标：设计执行部件(Exec Unit)

执行部件（Exec Unit）的设计

执行部件功能？

- 计算内存地址
- 计算转移目标地址
- 一般ALU运算

Load指令的各控制信号取值？

RegDes=0, ALUSrc=1
ALUop=add, Extop=1

Store指令呢？

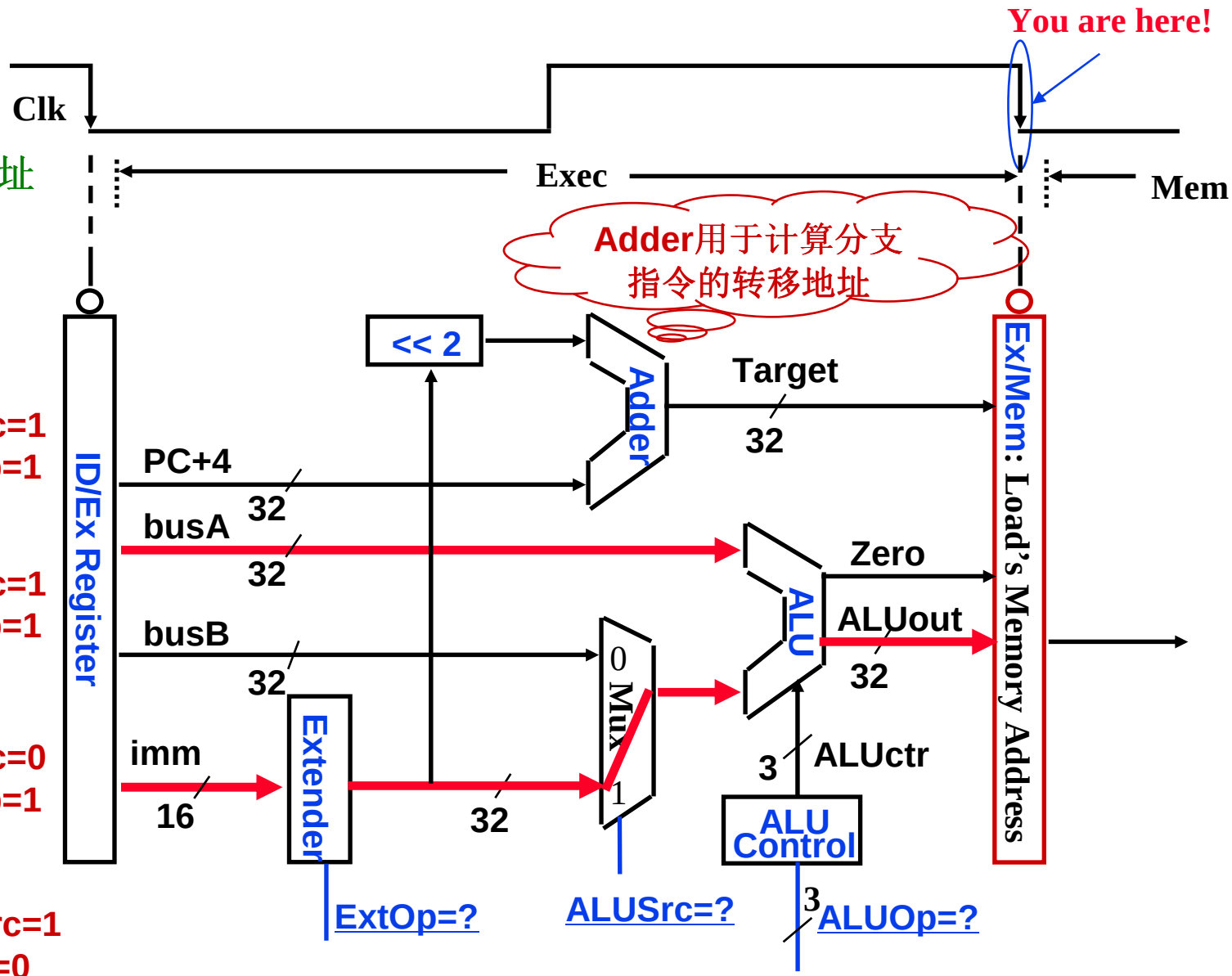
RegDes=x, ALUSrc=1
ALUop=abb, Extop=1

Branch指令呢？

RegDes=x, ALUSrc=0
ALUop=sub, Extop=1

Ori指令呢？

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

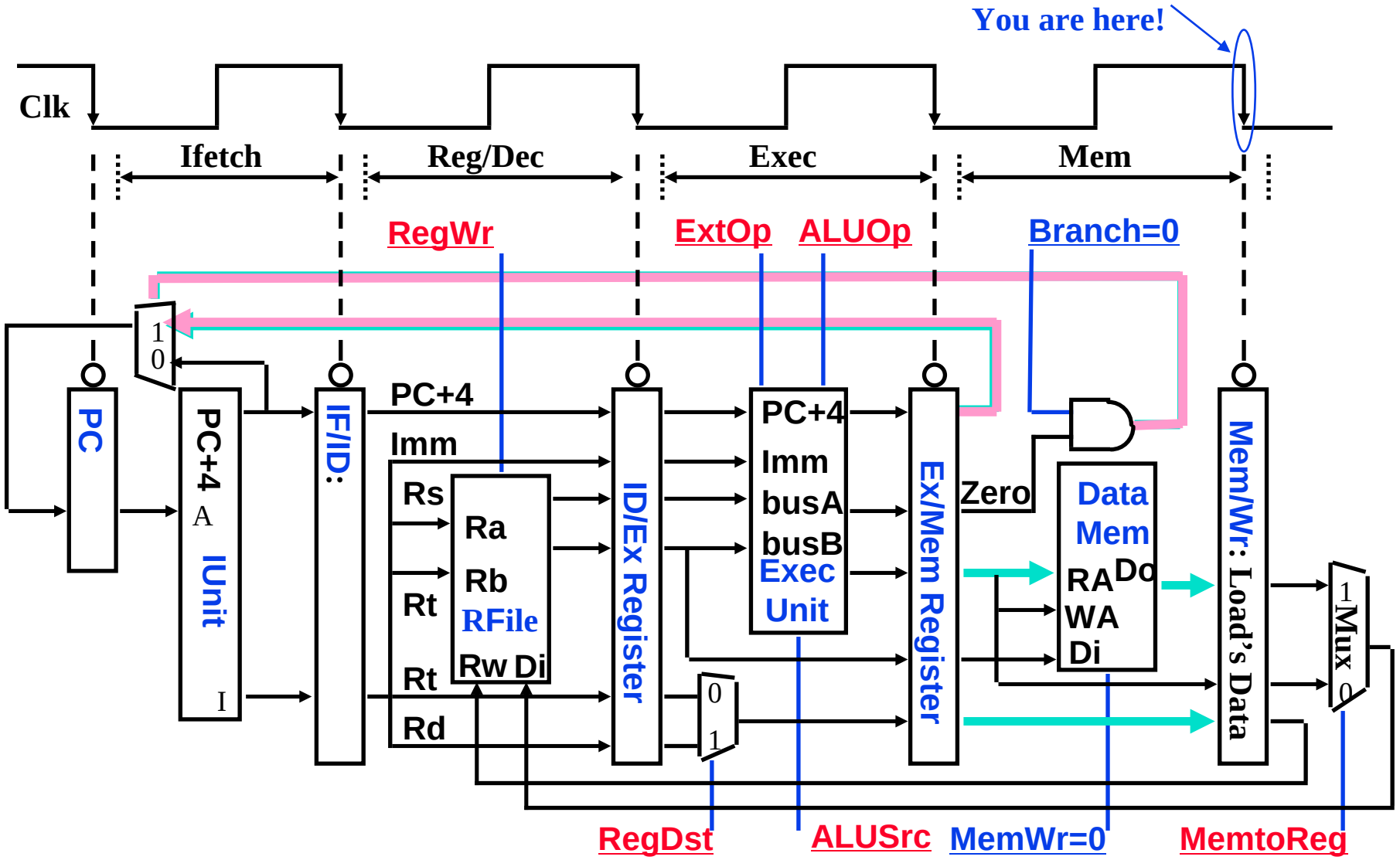


R型指令呢？

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

Load指令的存储器读(Mem)周期

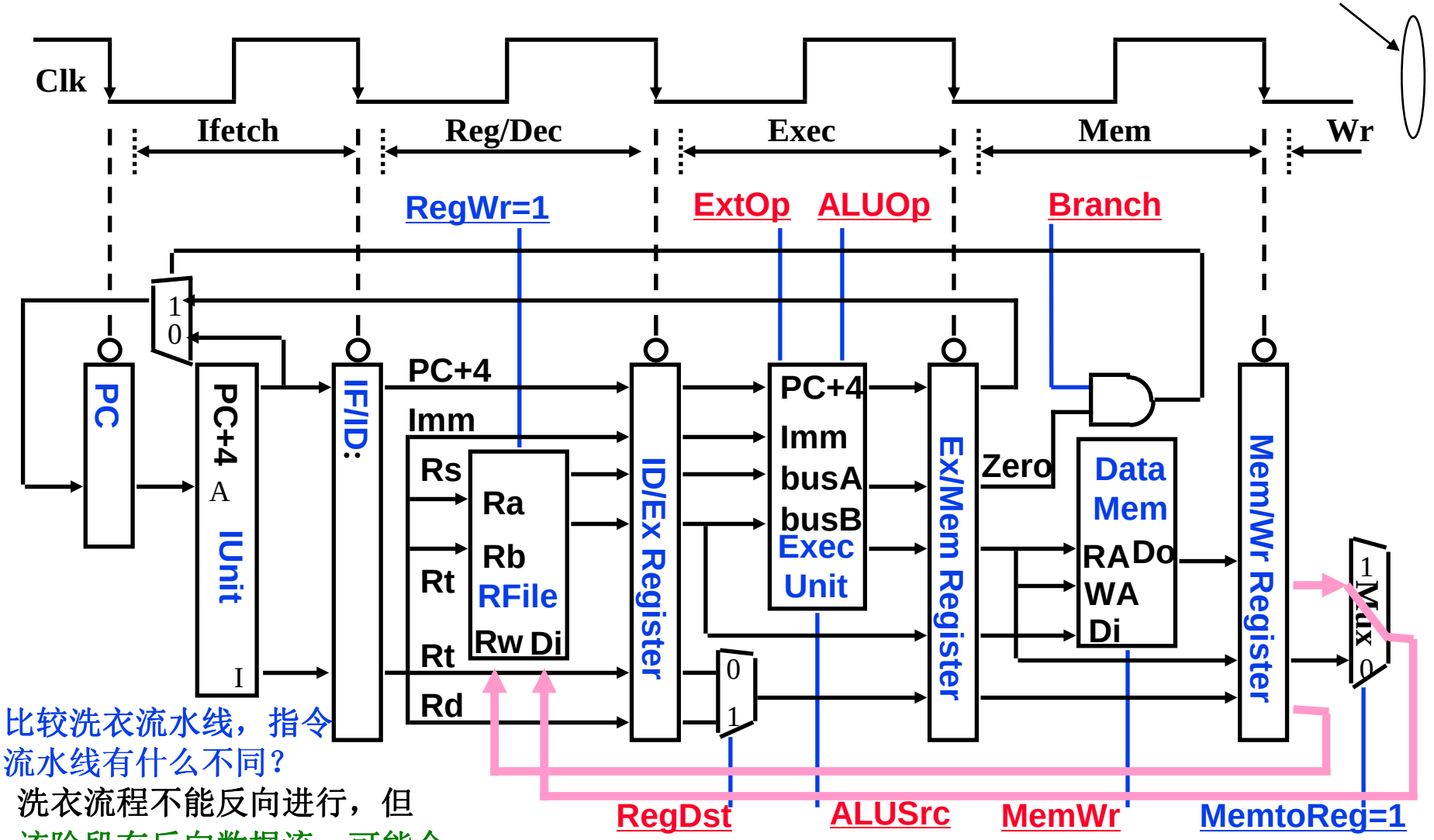
° **Location 10:** lw $1, 0x100($2)　　功能： $1 <- Mem[($2) + 0x100]



周期以最长操作为准设计Cycle > T_{read}

Load指令的回写（Write Back）阶段

° **Location 10:** lw $1, 0x100($2)　　功能：$1 <- Mem[($2) + 0x100]



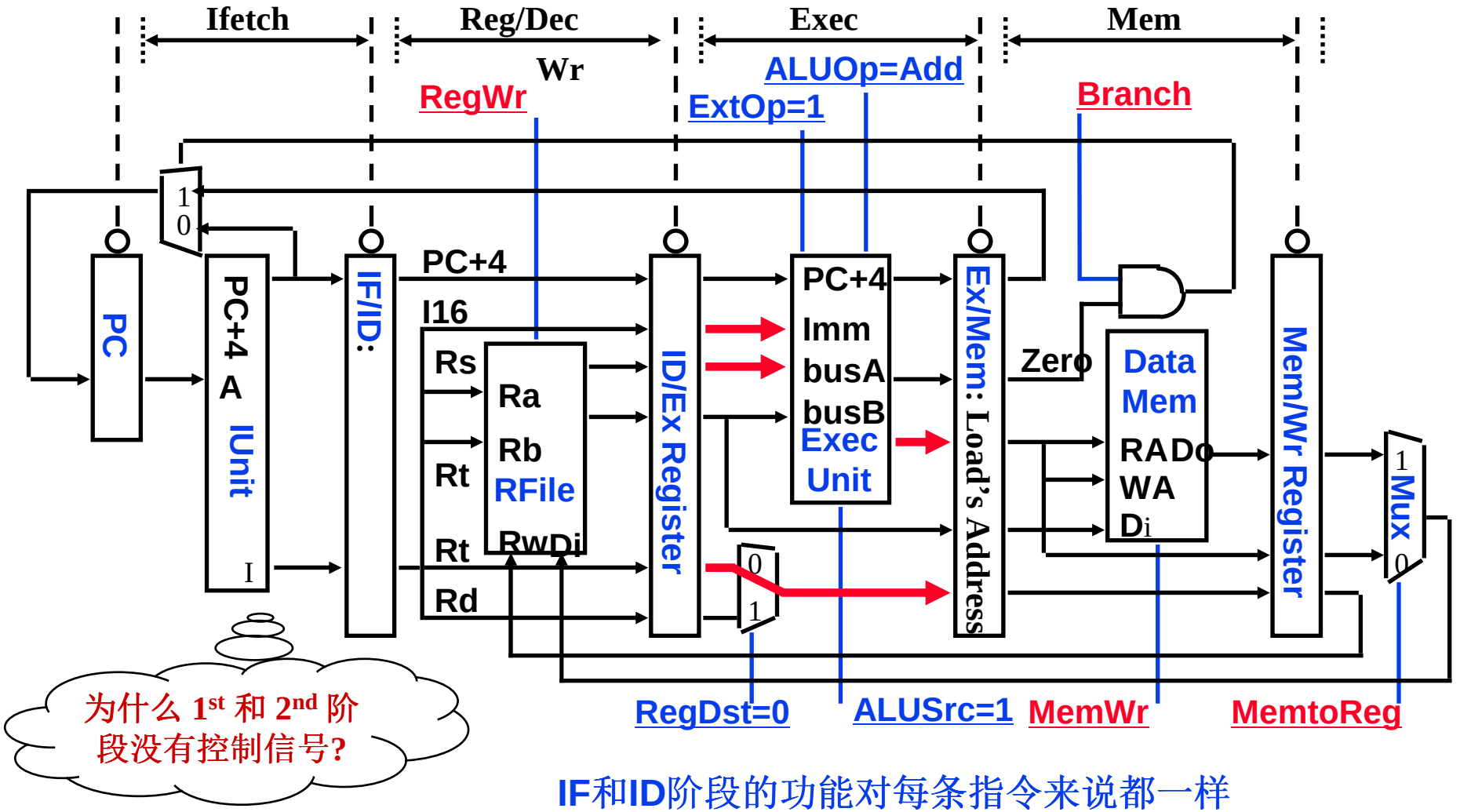
比较洗衣流水线，指令流水线有什么不同？

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。

各阶段所经DataPath已有，控制信号如何得到？

流水线中的Control Signals如何获得?

- 主要考察: 第N阶段的控制信号,它取决于是哪条指令的哪个阶段。
  - N = Exec, Mem, or Wr (只有这三个阶段有控制信号)
  - 例: Load的Exec段的控制信号 = Func (Load's Exec)

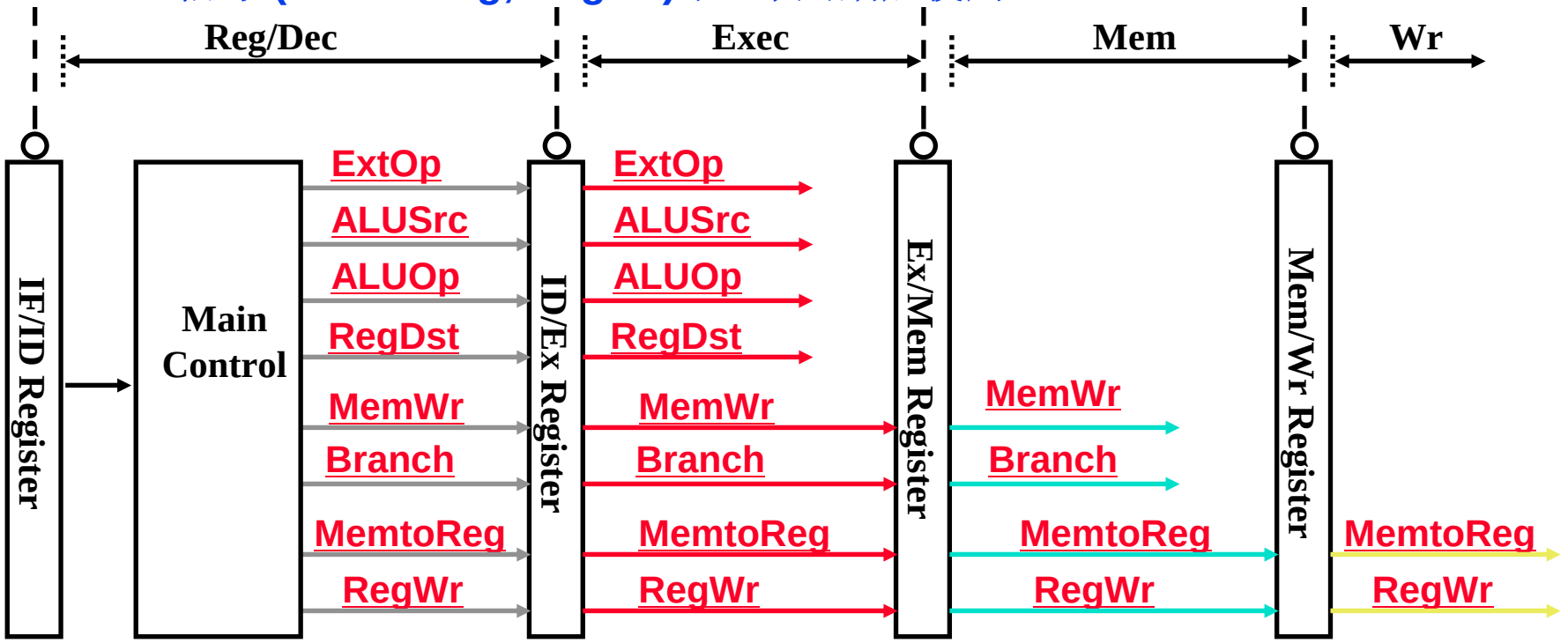


IF和ID阶段的功能对每条指令来说都一样

Load指令:流水线中的控制信号

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

所以，控制信号也要保存在流水段寄存器中！



各流水段部件在一个时钟内完成某条指令的某个阶段的工作！

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

流水线中的Control Signals

- 通过对前面流水线数据通路的分析，得知：
  - 因为每个时钟都会改变PC的值，所以PC不需要写控制信号
  - 流水段寄存器每个时钟都会写入一次，也不需要写控制信号
  - Ifecth阶段和Dec/Reg阶段都没有控制信号
  - Exec阶段的控制信号有四个
    - ExtOp (扩展器操作)：1- 符号扩展；0- 零扩展
    - ALUSrc (ALU的B口来源)：1- 来源于扩展器；0- 来源于BusB
    - ALUOp (主控制器输出，用于辅助局部ALU控制逻辑来决定ALUCtrl)
    - RegDst (指定目的寄存器)：1- Rd；0- Rt
  - Mem阶段的控制信号有两个
    - MemWr (DM的写信号)：Store指令时为1，其他指令为0
    - Branch (是否为分支指令)：分支指令时为1，其他指令为0
  - Wr阶段的控制信号有两个
    - MemtoReg (寄存器的写入源)：1- DM输出；0- ALU输出
    - RegWr (寄存器堆写信号)：结果写寄存器的指令都为1，其他指令为0

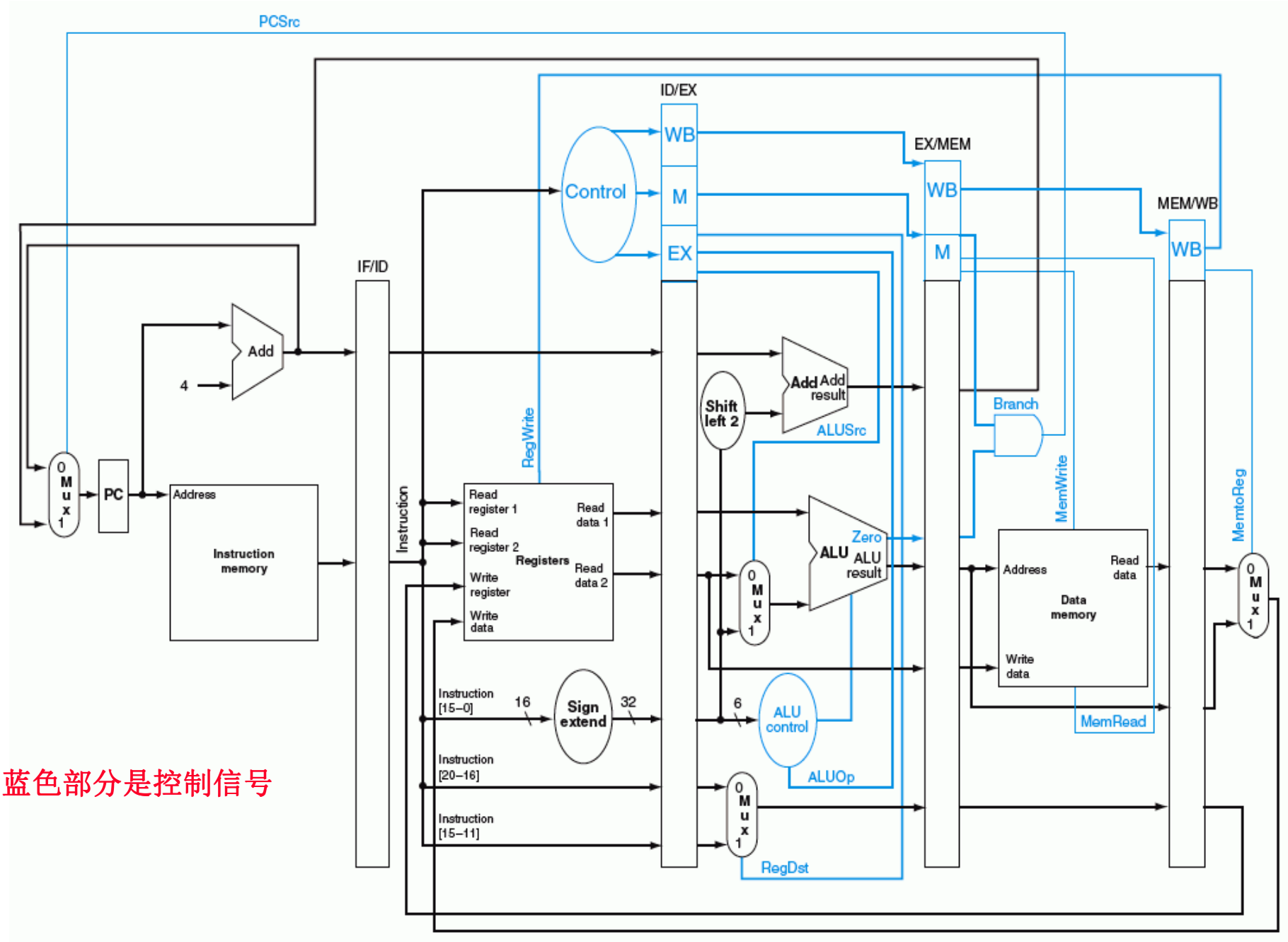
控制逻辑 (Control)的设计

- 流水线控制逻辑的设计
  - 每条指令的控制信号在指令执行期间都不变

    （谁记得单周期和多周期时是怎样的情况？）
  - 与单周期控制逻辑设计类似

    （谁记得单周期和多周期控制逻辑各是怎样设计的？）
  - 设计过程
    - 控制逻辑分成两部分
      - 主控制逻辑：生成ALUop和其他控制信号
      - 局部ALU控制逻辑：根据ALUop和func字段生成ALUCtrl
    - 用真值表建立指令和控制信号之间的关系
    - 写出每个控制信号的逻辑表达式
  - 控制逻辑的输出在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
  - 同一时刻在不同阶段执行不同指令，因而不同阶段的控制信号对应不同的指令

忘记单周期和多周期控制设计的同学，复习一下第五章的内容！


PCSrc
ID/EX
EX/MEM
MEM/WB
WB
M
EX
Control
IF/ID
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALUSrc
Branch
MemWrite
MemtoReg
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
MemRead
Instruction [15–0]
16
Sign extend
32
ALU control
6
ALUOp
Instruction [20–16]
Instruction [15–11]
RegDst

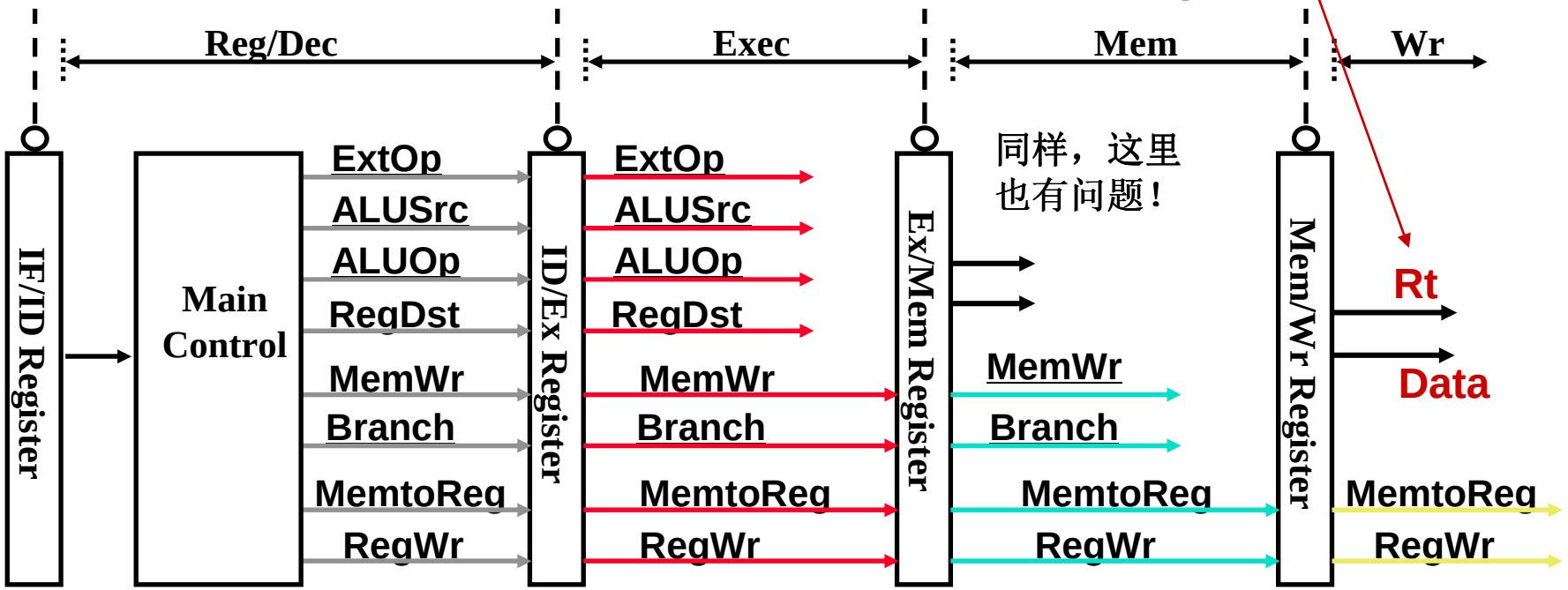
蓝色部分是控制信号

Load指令:流水线中的控制信号

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

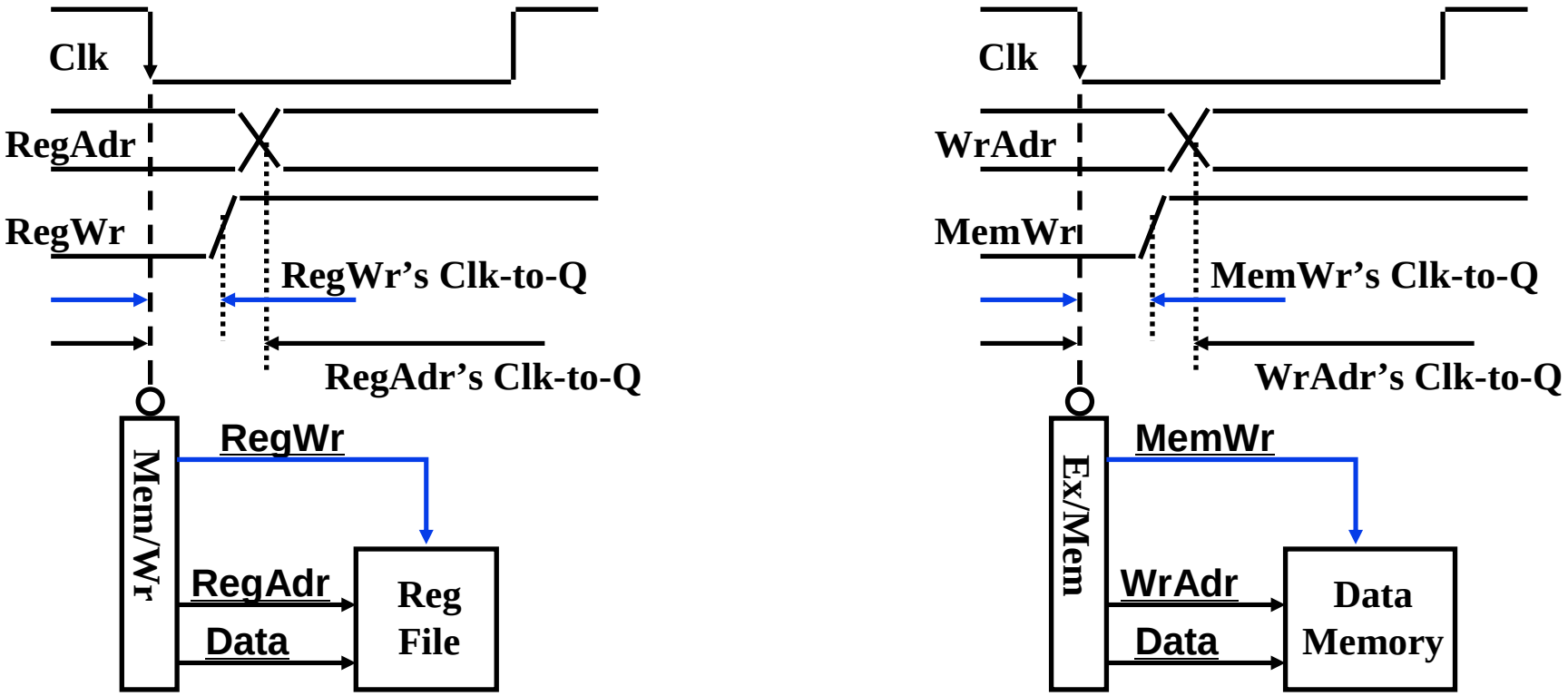
所以，控制信号也要保存在流水段寄存器中！

Rt和Data在RegWr后到达怎么办？



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

Wr阶段的开始: 存在一个实际的问题!



- 在流水线中也存在地址 和 写使能之间的"竞争"问题
  - Wr段开始时若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!
- 不能用多周期中的方法! 为什么?

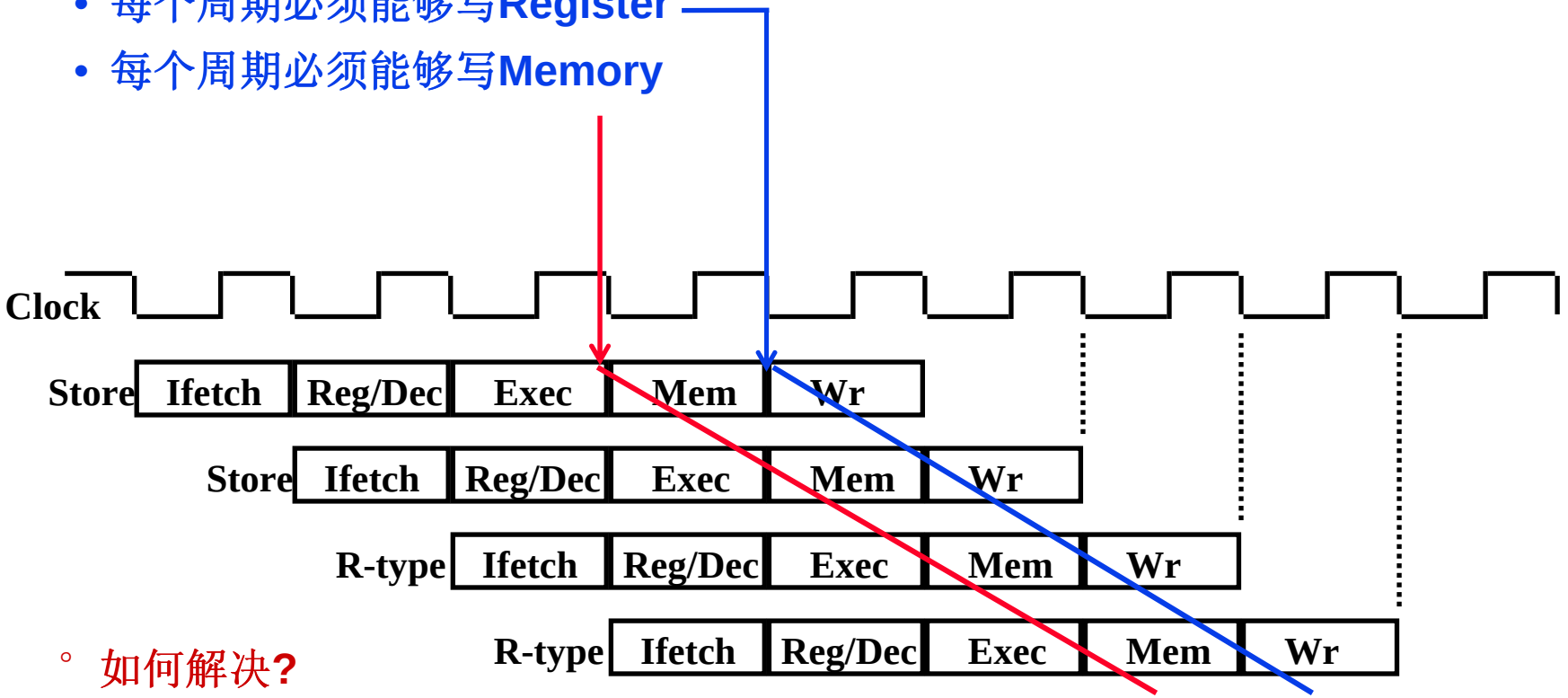
  哪个同学记得多周期中是如何处理"竞争"问题的?

流水线中的“竞争”问题

- 多周期中解决 **Addr** 和 **WrEn**之间竞争问题的方法:
  - 在第 **N**周期结束时，让**Addr**信号有效
  - 在第 **N + 1**周期让**WrEn**有效

保证**Addr**信号在**WriteEnable**信号之前到达

- 上述方法在流水线设计中不能用，因为:
  - 每个周期必须能够写**Register**
  - 每个周期必须能够写**Memory**

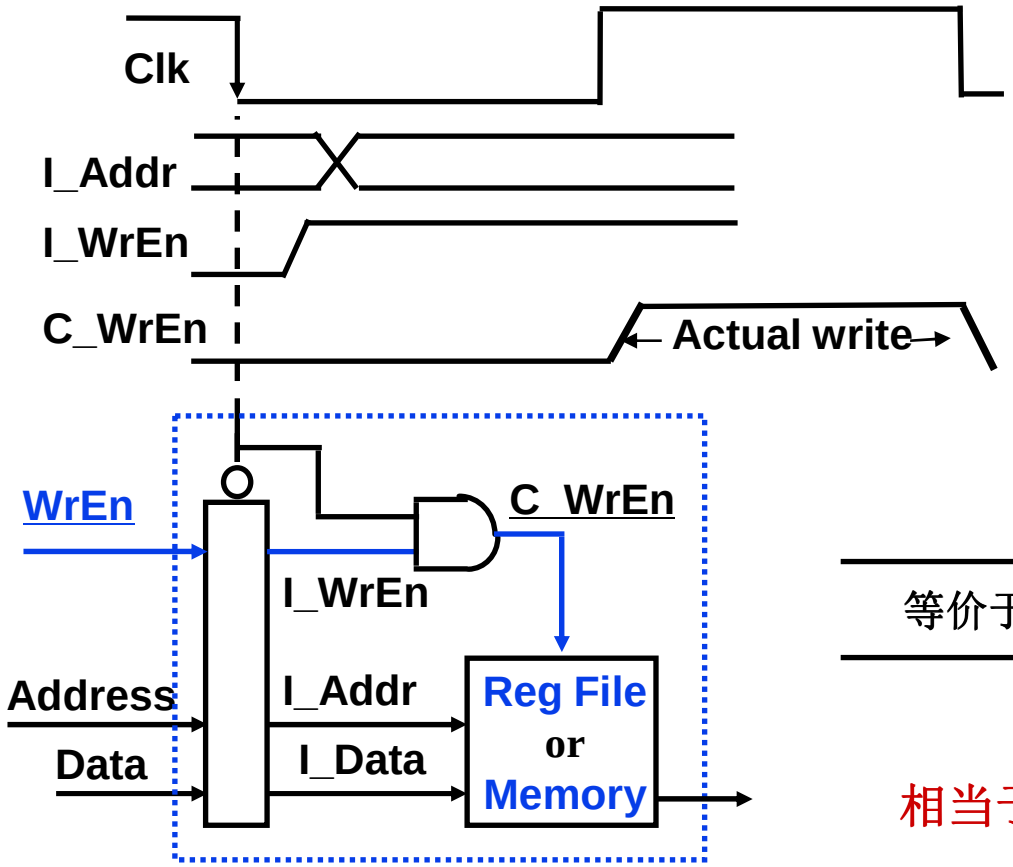


- 如何解决?

寄存器组的同步和存储器的同步

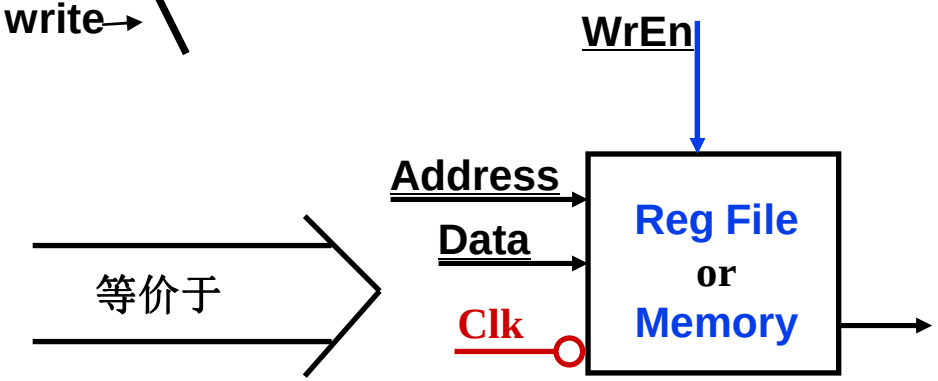
- 解决方案：将**Write Enable**和时钟信号“与”

须由电路专家确保不会发生“定时错误”
（即：能合理设计“Clock”!）



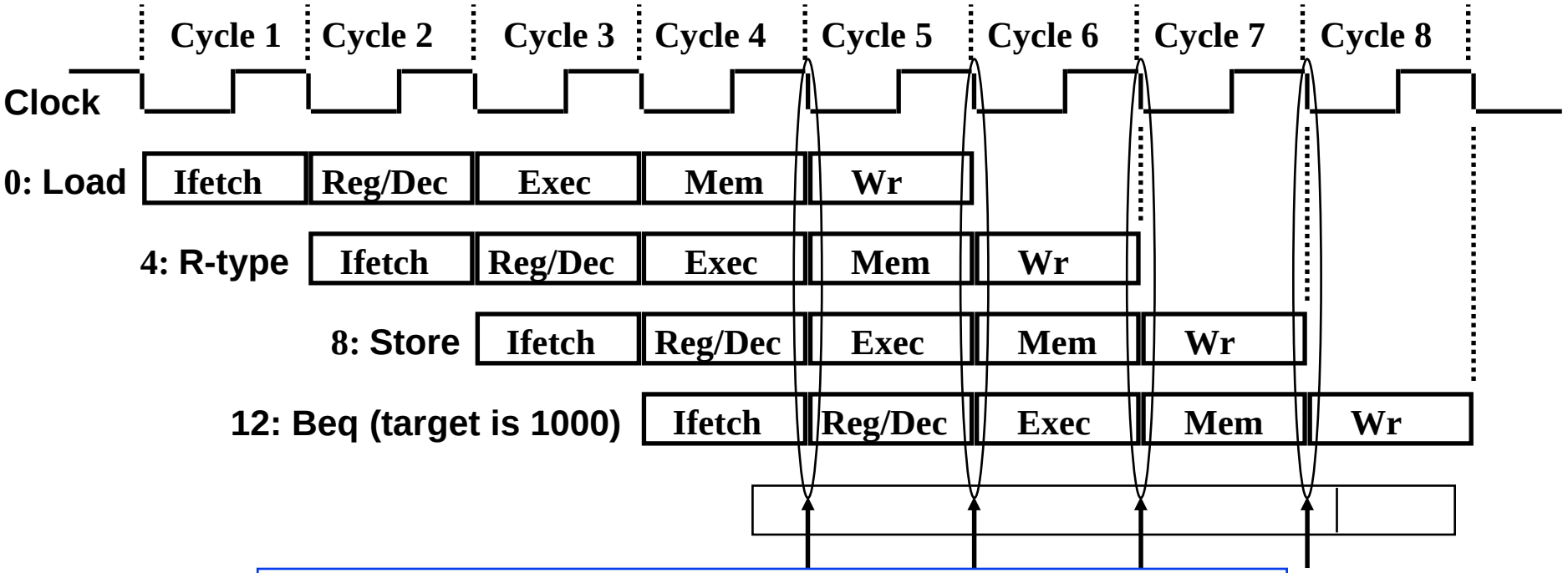
1. Addr, Data, 和 WrEn 必须在Clk边沿到来后至少稳定一个 set-up时间

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存储器

流水线举例：考察流水线DataPath的数据流动情况



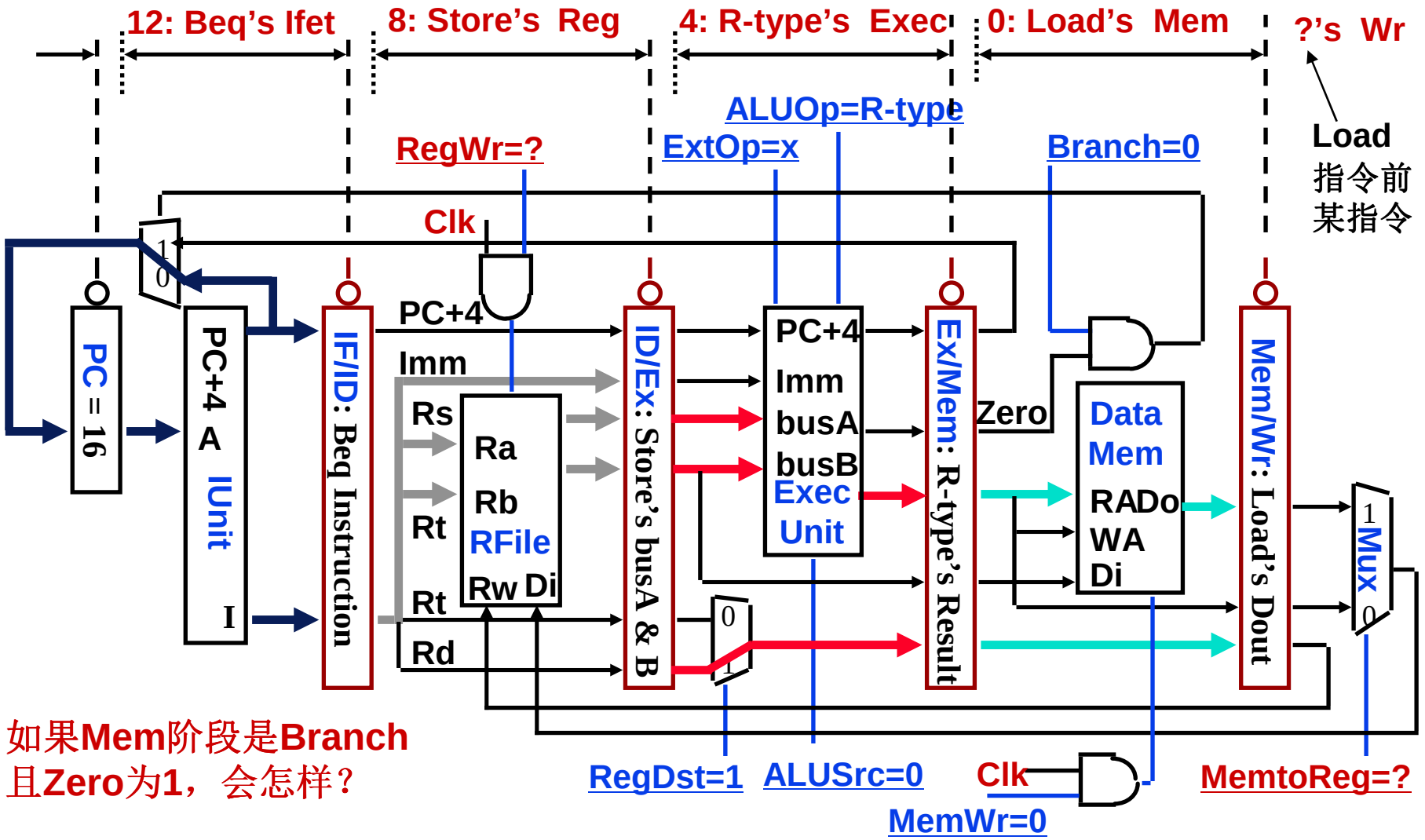
考察以下几个点的情况：End of Cycle 4　End of Cycle 5　End of Cycle 6　End of Cycle 7

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

说明：后面仅考察数据流动情况，控制信号随数据同步流动因而不再说明。

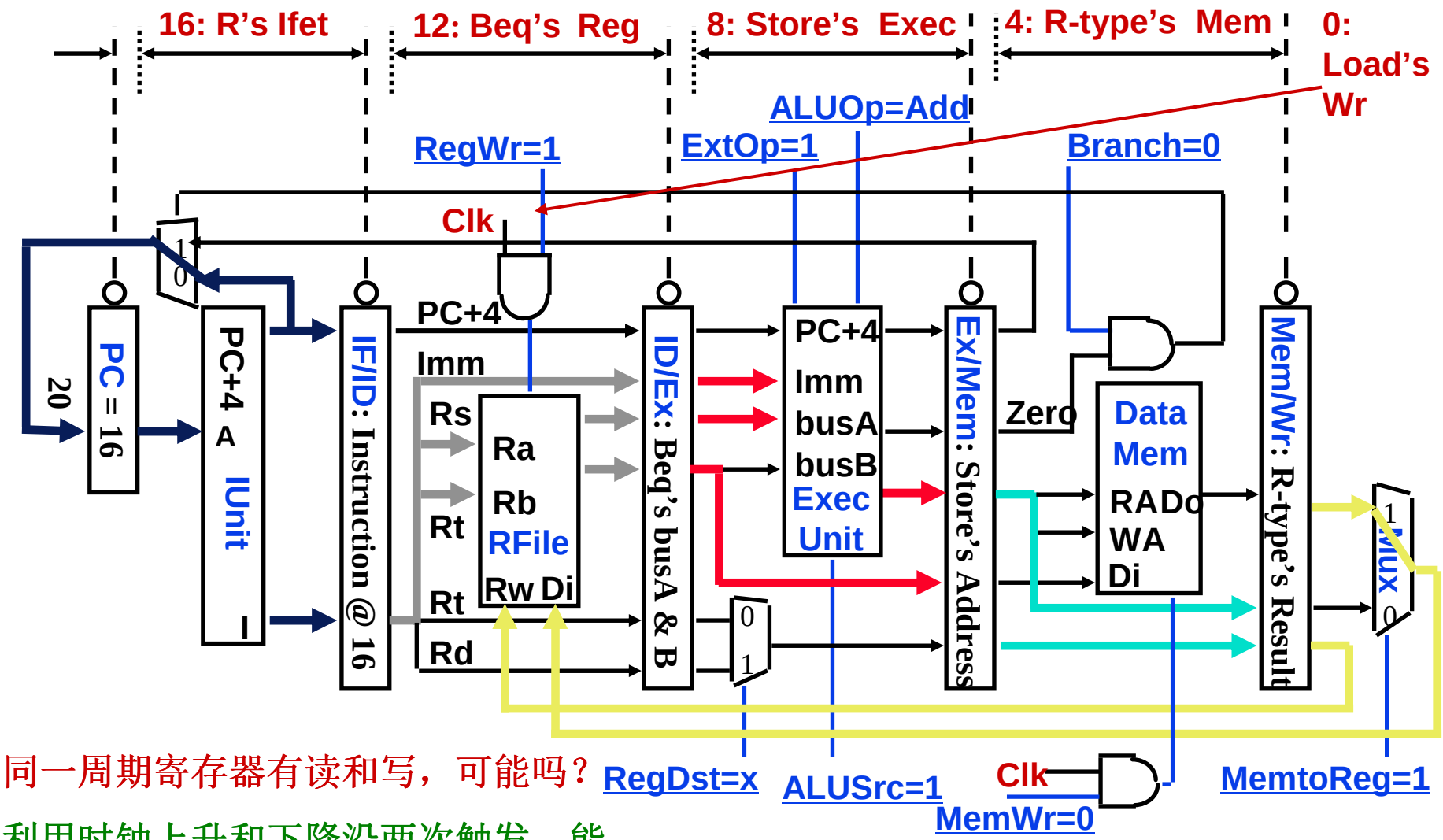
第四周期结束时的状态：

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



第五周期结束时的状态：

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



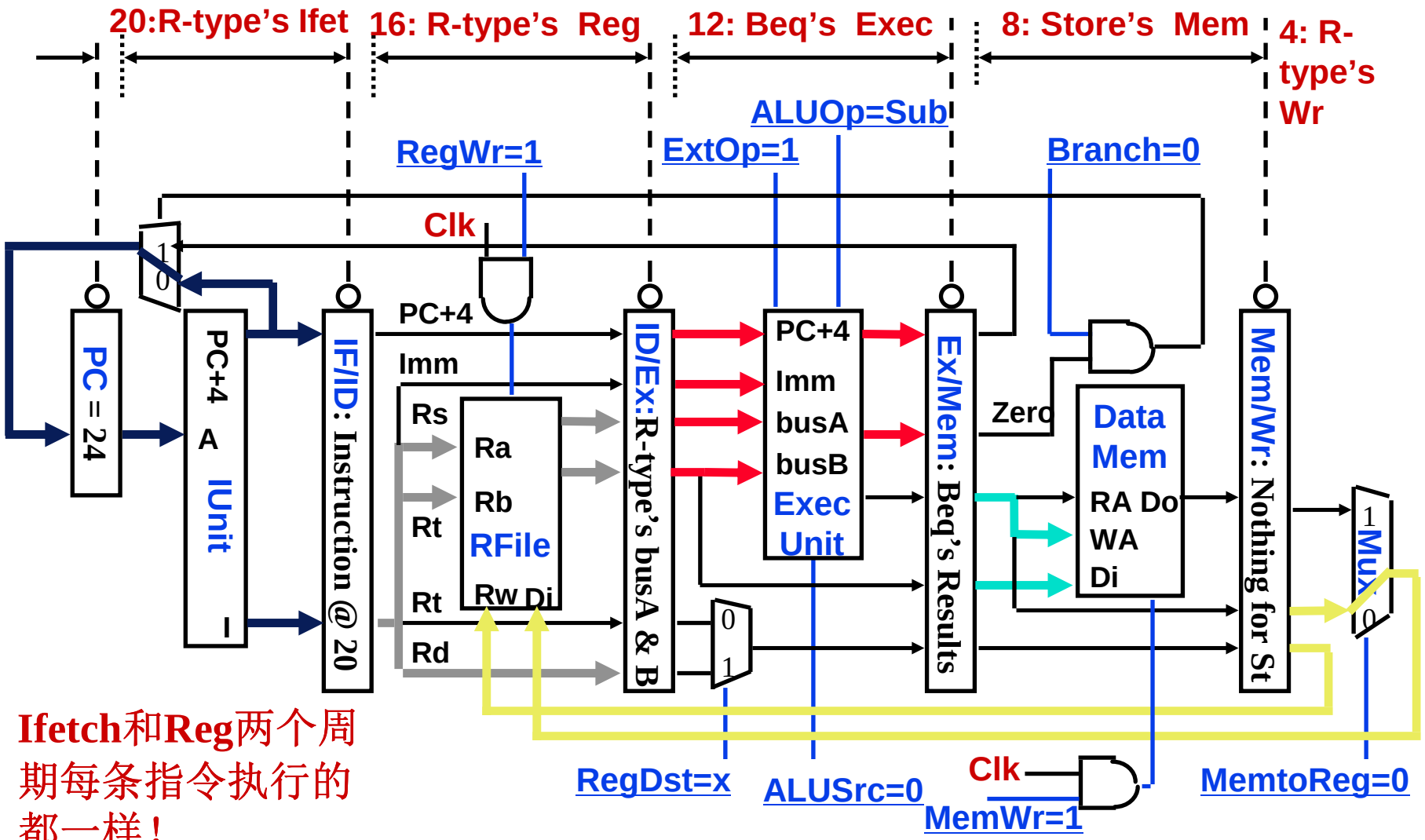
同一周期寄存器有读和写，可能吗？

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

寄存器的写口和读口可看成是独立的两个部件！

第六周期结束时的状态：

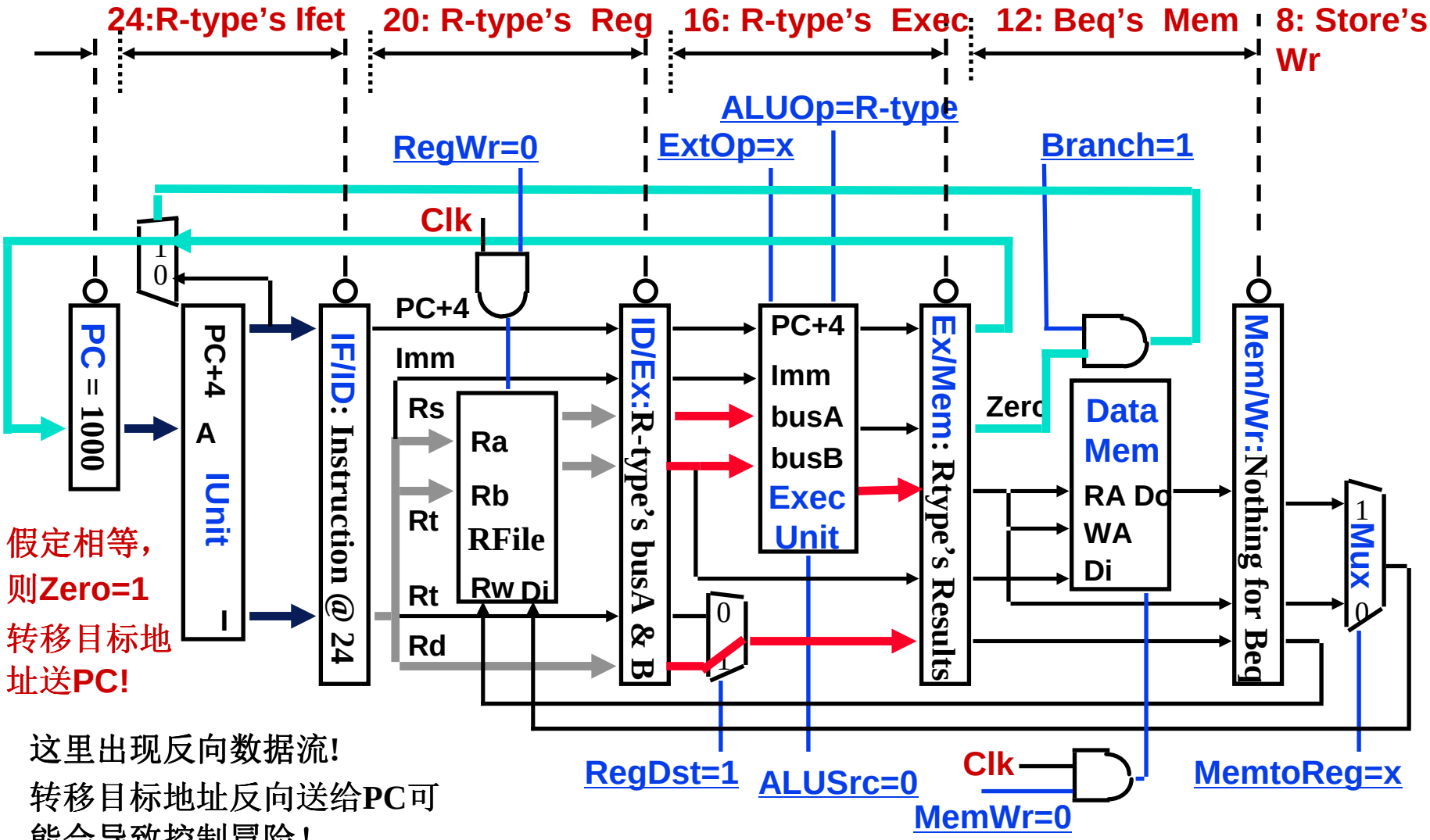
- 4: R's Wr 8: Store's Mem 12: Beq's Exec 16: R's Reg 20: R's Ifet



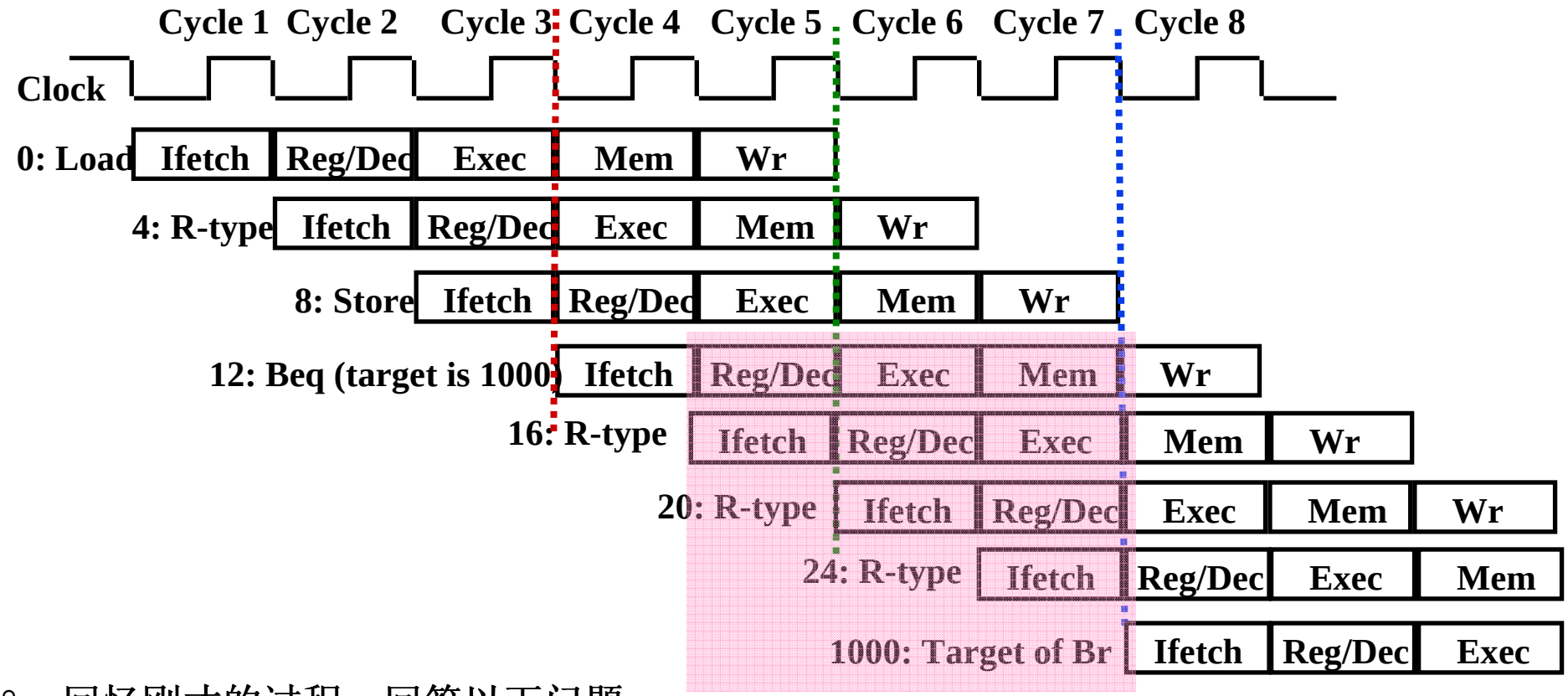
Ifetch和Reg两个周期每条指令执行的都一样！

第七周期结束时的状态：

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

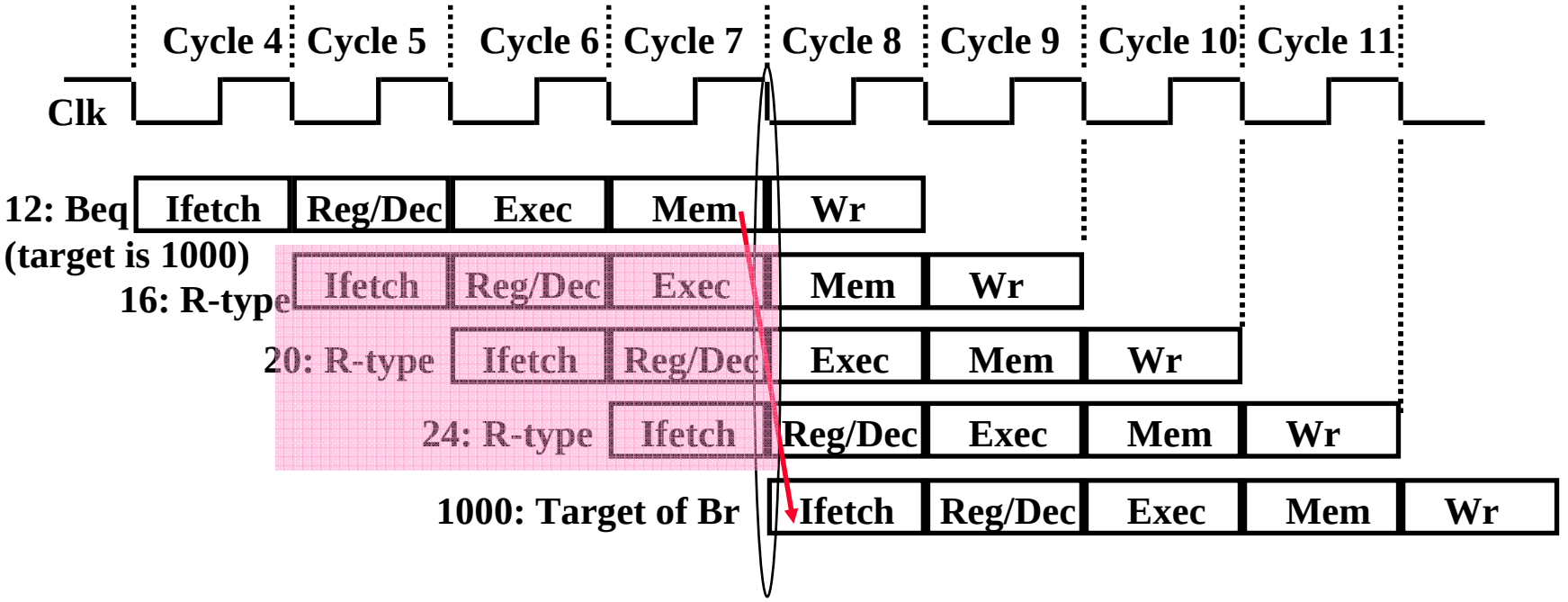


总结前面的流水线执行过程



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

转移分支指令(Branch)引起的“延迟”现象



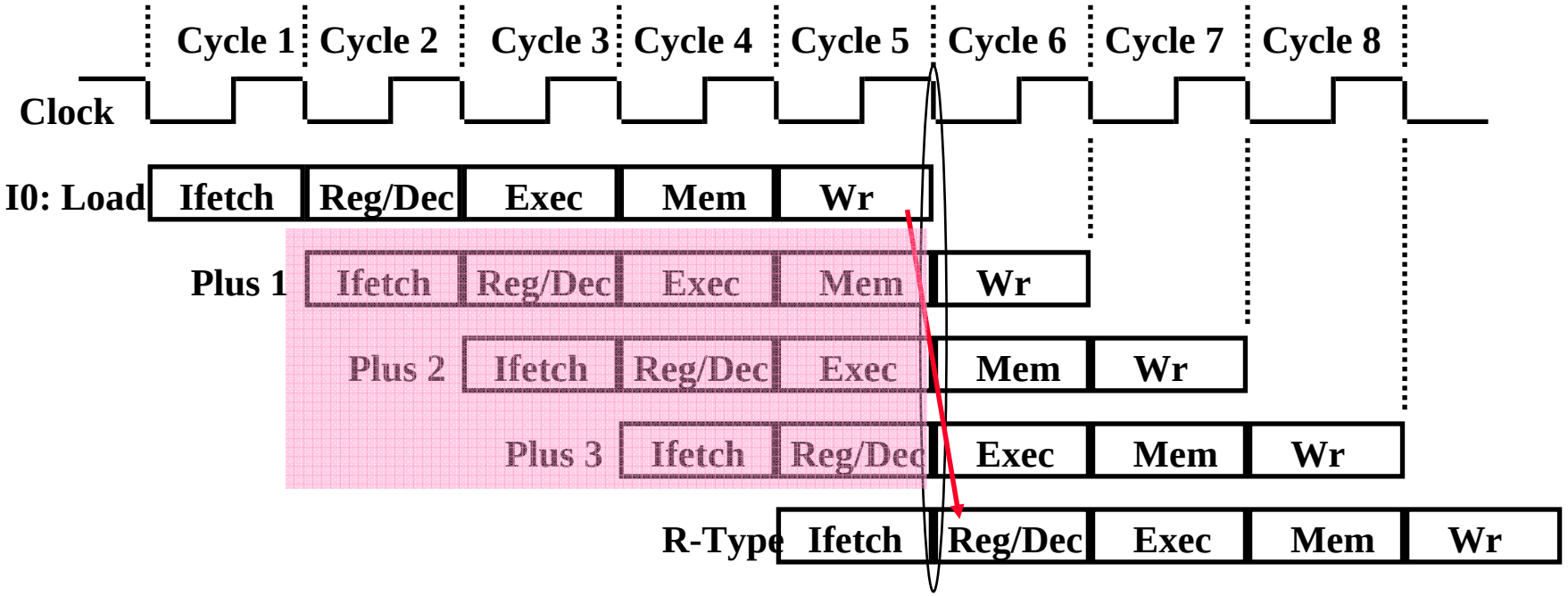
- 虽然Beq指令在第四周期取出，但:
  - 目标地址在第七周期才被送到PC的输入端
  - 第八周期才能取出目标地址处的指令执行

  结果：在取目标指令之前，已有三条指令被取出，取错了三条指令！
- 这种现象称为控制冒险（Control Hazard ）

  （注：也称为分支冒险或转移冒险（Branch Hazard） ）

BACK

装入指令(Load)引起的“延迟”现象



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

  结果：在**Load**指令结果有效前，已经有三条指令被取出

  (如果随后的指令要用到**Load**的数据的话，就需要延迟三条指令才能用！**)**
- 这种现象被称为 数据冒险 **(Data Hazard)** 或数据相关**(Data Dependency)**

内容小结

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

（下一讲主要介绍解决流水线冒险的数据通路如何设计）