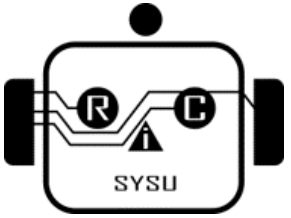
中山大學
SUN YAT-SEN UNIVERSITY



计算机组成原理

第四章：MIPS CPU逻辑设计

中山大学计算机学院
陈刚

2022年秋季

上节回顾

回顾内容

第四章 处理器

- 4.5 多周期控制器的实现
  - 多周期控制器的实现思路
  - 单周期数据通路和多周期数据通路的差别
  - 详细分析7条指令在多周期通路中的执行过程，分析每个周期内控制信号的取值，生成相应的状态
  - 综合生成所有指令的状态转换图
  - 根据状态转换图，生成控制器输出的逻辑表达式

本节概要

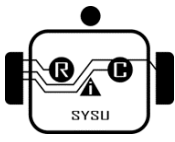
重点内容

第四章 处理器

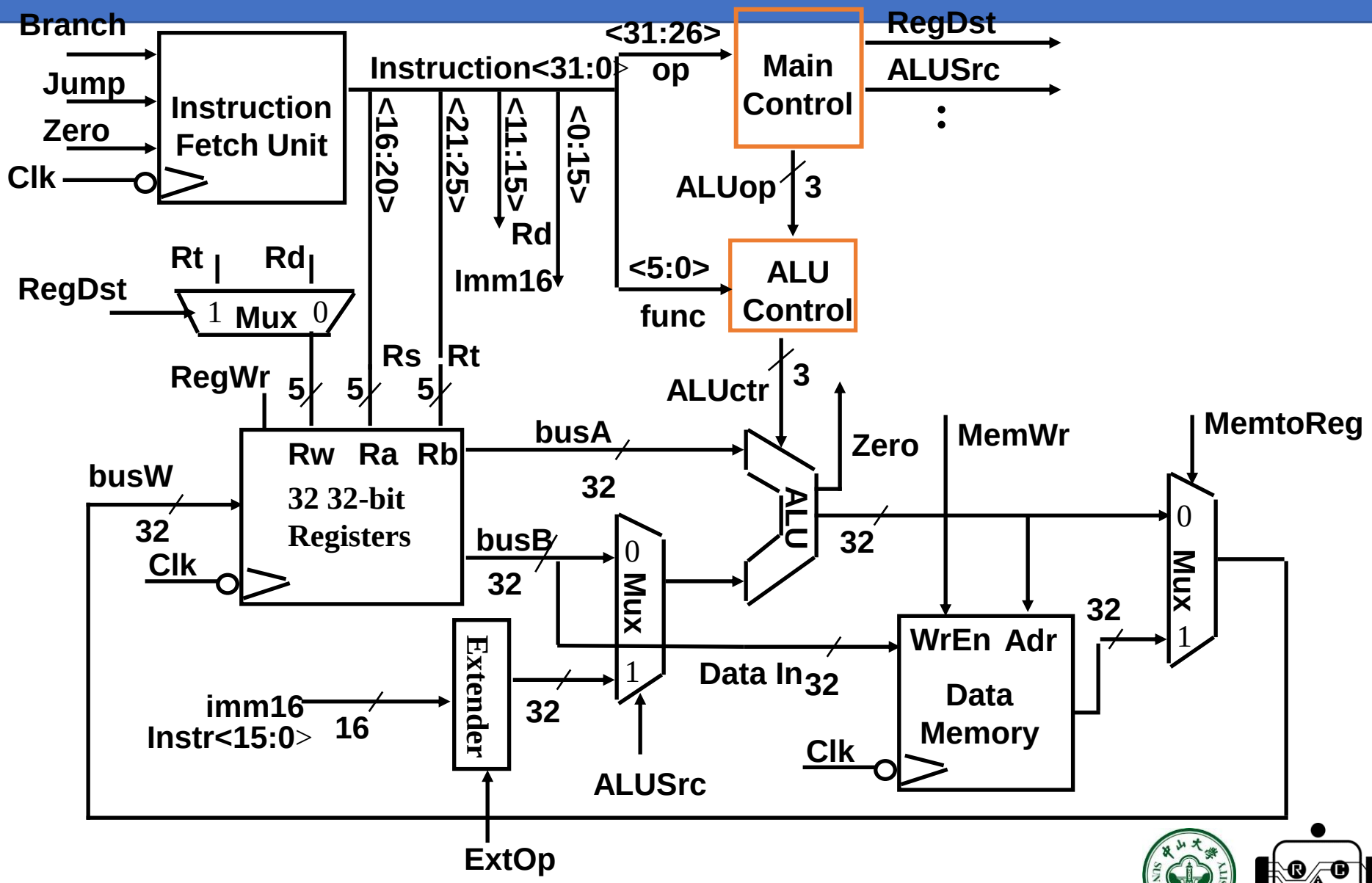
- 4.6 流水线数据通路和控制
- 4.7 数据冒险：转发与阻塞
- 4.8 控制冒险

基本要求

- 理解流水线的基本概念
- 了解三种冒险及其冒险处理方法

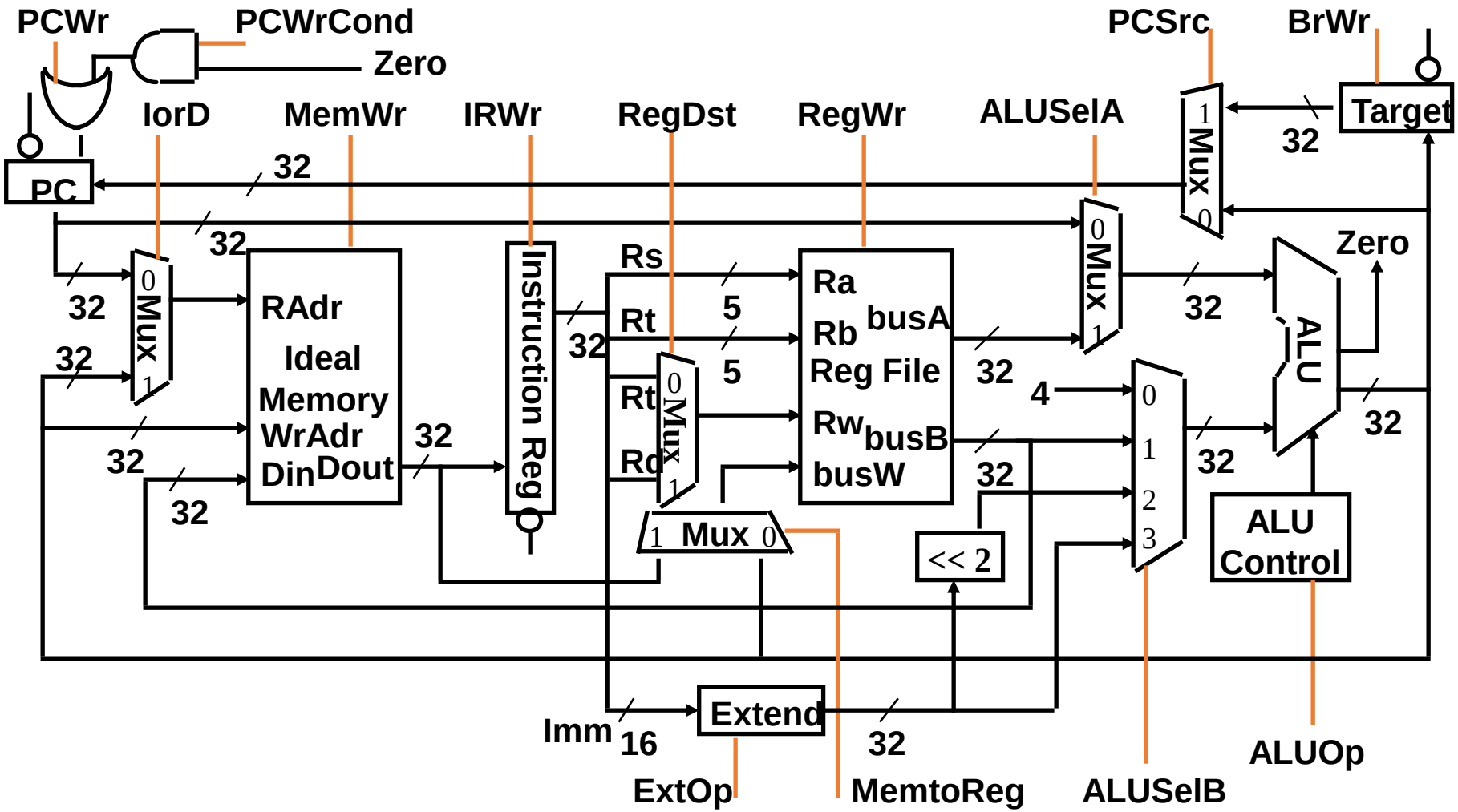


复习：A Single Cycle Processor



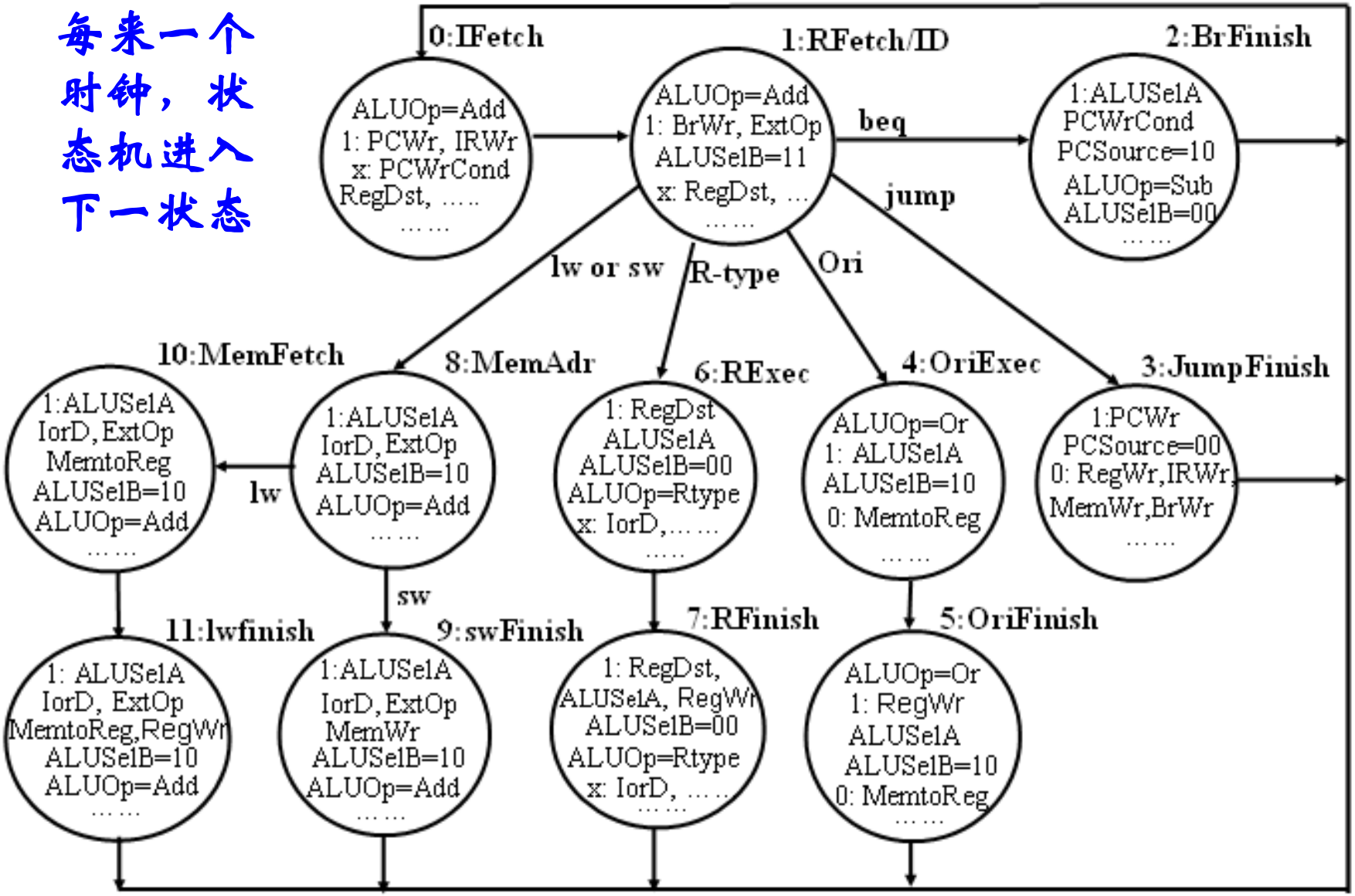
复习：Multiple Cycle Processor

- MCP：一个功能部件在一个指令周期中可以被使用多次。



回顾——多周期处理器的状态转换图

每来一个时钟，状态机进入下一状态

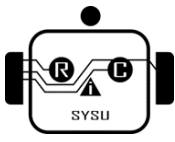


各指令的时钟周期数：R-4, ori-4, beq-3, Jump-3, lw-5, sw-4

多周期处理器的缺点

- 多周期处理器的问题根源
  - 每条指令的执行需要不同个数的时钟周期，下条指令必须等到本条指令完成才能开始执行，性能太慢！
- 解决思路
  - 仍然像多周期控制器那样，把指令执行分成多个阶段，各阶段在一个时钟周期内完成
  - 每个阶段都设置相应的存储元件，其执行结果都在下个时钟开始时保存到相应阶段的存储单元内
  - 多条指令以流水线方式控制并行执行，每个时钟周期能够得到一条指令的执行结果

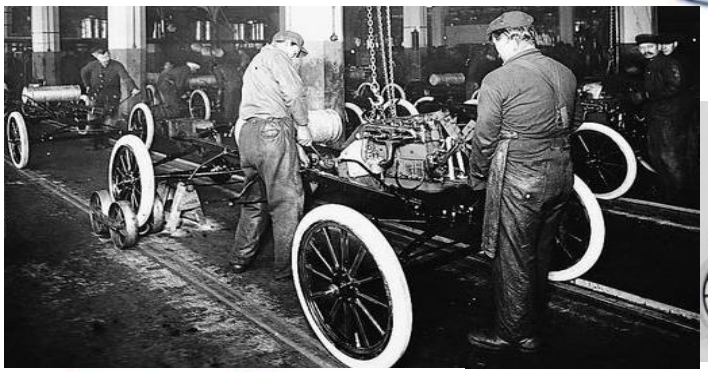
什么是流水线技术？



流水线概述

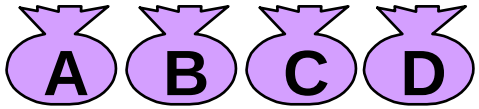
20世纪初美国工程师泰勒发明——工业流水线，是管理学界最伟大的发明

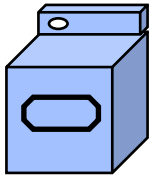
1913年10月7日，福特汽车创立了汽车装配流水线，使速度提高了8倍，第一次实现每10秒钟诞生一部汽车的神话



一个日常生活中的例子—洗衣服

- Laundry Example
  - A, B, C, D四个人，每人都有一批衣服需要 wash, dry, fold
  - Wash阶段：30 minutes
  - Dry阶段：40 minutes
  - Fold阶段： 20 minutes

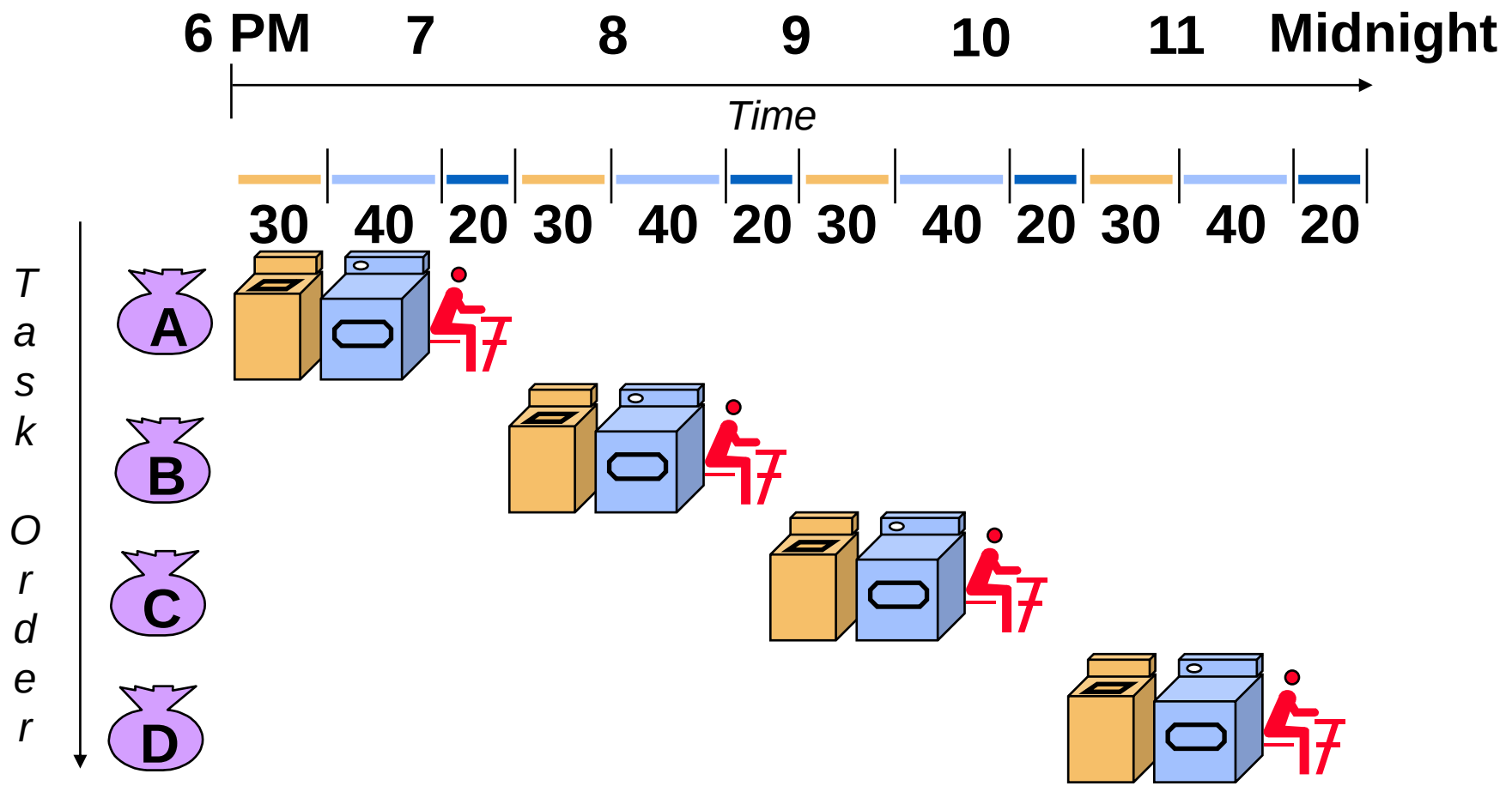




如果让你来管理洗衣店，你会如何安排？

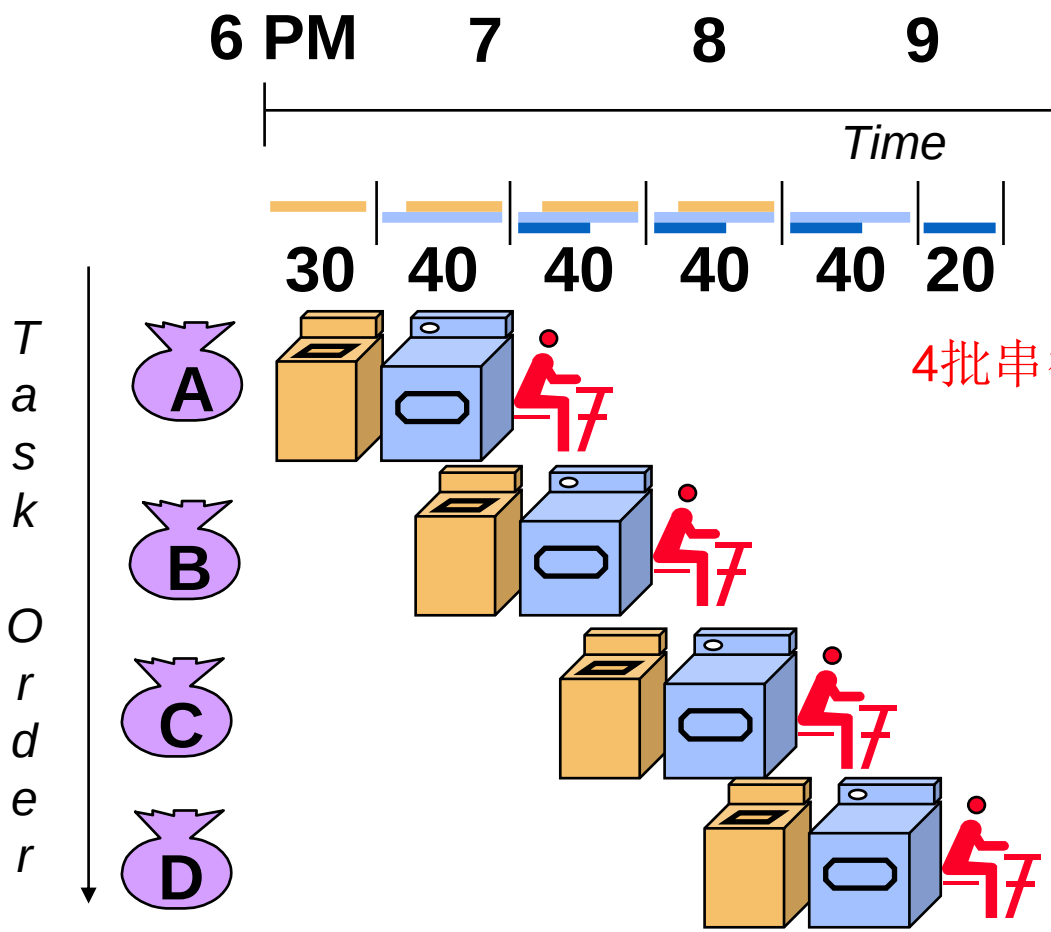
Pipelining: It's Natural !

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)



4批串行为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

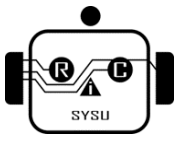
假定每一步时间均衡，则

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！

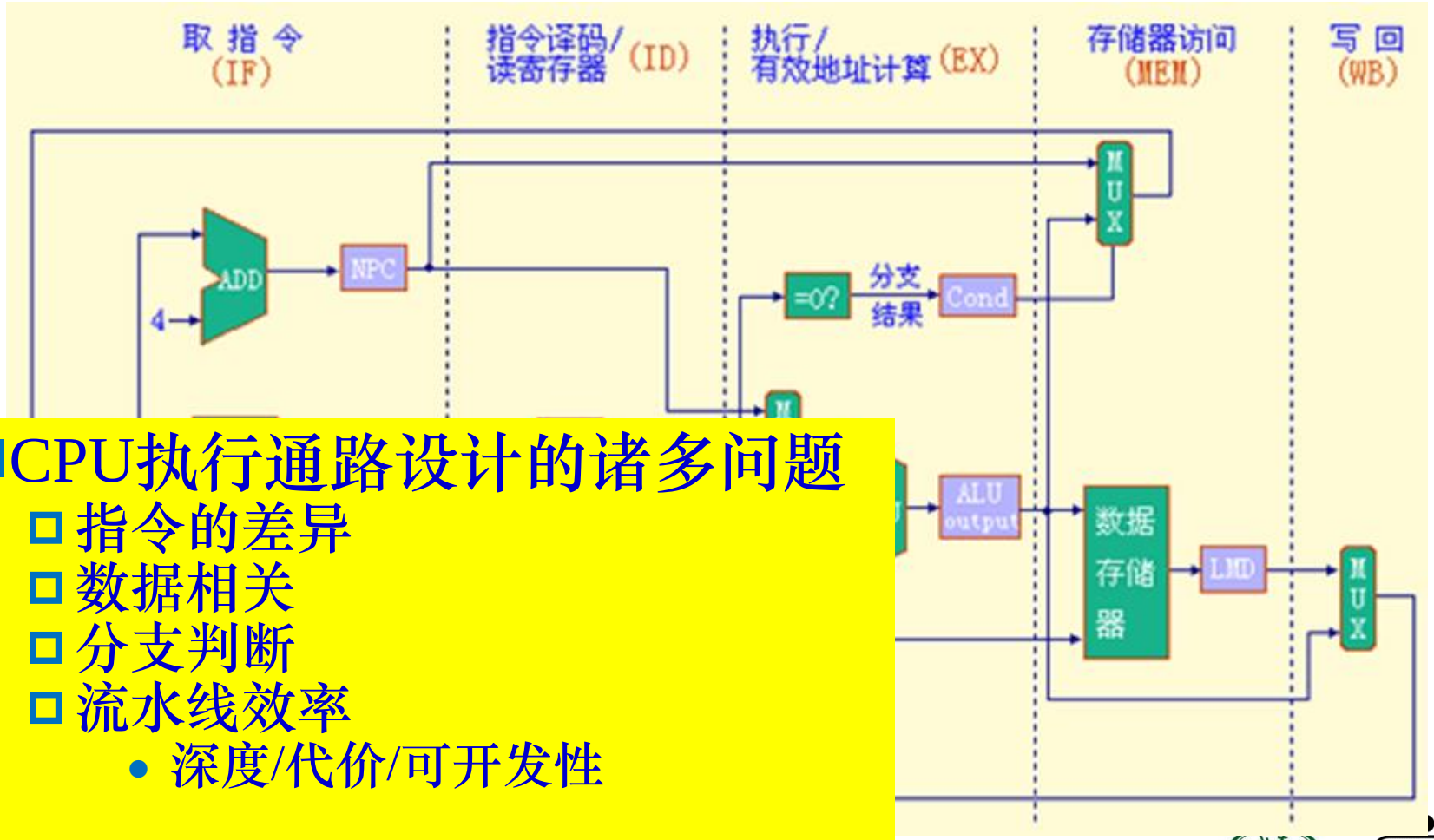
流水线概述

- 流水线(pipeline)技术是指在程序执行时多条指令重叠进行操作的一种准并行处理实现技术
  - 首次在Intel 486中使用，可使CPU并行执行多条指令，从而提高CPU完成指令执行的吞吐率(时间维度的并行)



提高处理器的性能

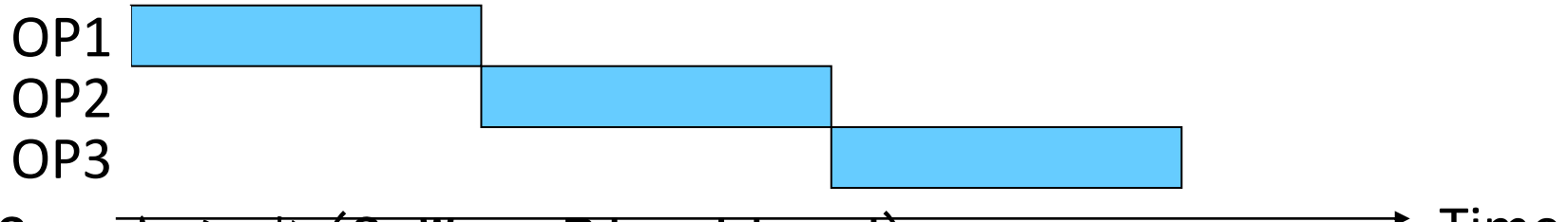
- CPU的执行通路也是一条流水线！



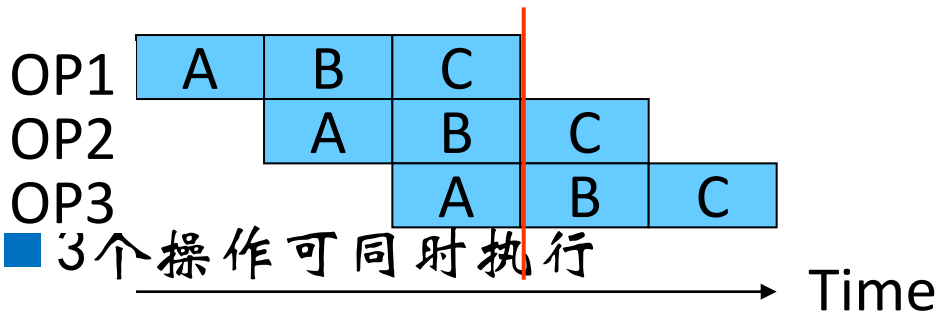
- CPU执行通路设计的诸多问题
  - 指令的差异
  - 数据相关
  - 分支判断
  - 流水线效率
    - 深度/代价/可开发性

流水线图(Pipeline Diagrams)

- 无流水线(Unpipelined)
  - 上一条指令完成之前不能启动新操作



- 3级流水线(3-Way Pipelined)



  - 3个操作可同时执行

提高处理器的性能——提升工作主频

- 如何提升工作主频？

要提高主频→减少每个流水级的执行周期→减小每个流水级的任务量→将任务再分解→增加流水线级数

Intel IA-32的流水线级数

型号	Pentium	P6架构	Pentium 4	Core	Nehalem
流水级数	5	12-13	20~31	14	16
主频MHz	66	200	2000~3600	2667	3200

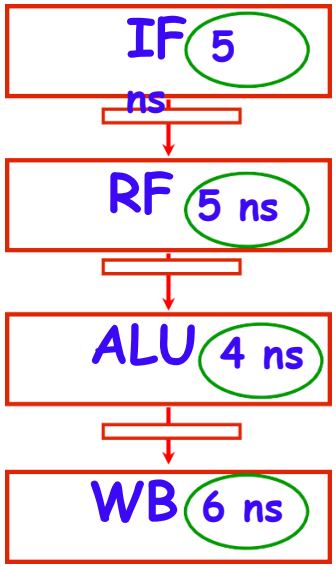
思考：流水线级数变多后，有没有副作用？

数据通路流水线化

目标: 保证(几乎) 1.0 的CPI, 同时提高时钟速率

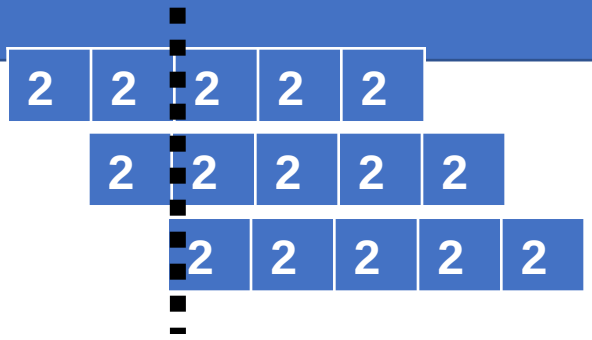
方法: 将处理器转变为一个多级流水线

- Instruction Fetch IF: 维护 PC. 每周期取一个指令.
- Register File(RF/ID): 从寄存器组RF中读取操作数.
- ALU: 执行指定的运算.
- Data Access:存储器访问(I type load store 指令需要)
- Reg Write-Back(WB): 将结果写回到寄存器中.



R型指令的 4 级流水线可以如图所示

Pipelining



流水线性能

时钟周期等于最长阶段所花时间为：2ns

每条lw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很大时约为2Nns，比串行方式提高约 5 倍,若各阶段操作均衡(例如，各阶段都是2ns，则串行需10Nns，流水线仍为2Nns)，加速为5倍。

Pipeline Speedup

$$\text{Time to execute an instruction}_{\text{pipeline}} = \frac{\text{Time to execute an instruction}_{\text{sequential}}}{\text{Number of stages}}$$

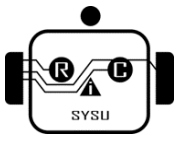
- If not balanced, speedup is less
- Speedup comes from increased **throughput**
- the **latency** of instruction does not decrease

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！

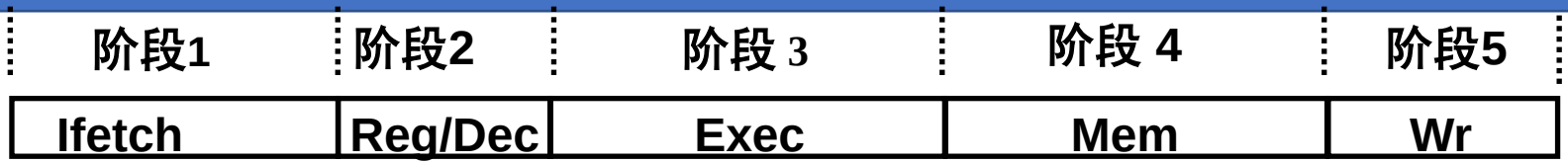
MIPS-流水线方式执行的指令集

	MIPS	X86
指令长度	定长，IF/ID易实现	1-17字节
指令格式	很少，源寄存器位置相同，ID能读寄存器堆	多，非对称，ID段需要一分为二
存储器操作	只有存取指令，可EX算地址，MEM访问	可直接对存储器数操作，Ex+Mem→Adr+Mem+Ex
数据传输	一个指令只访问一次存储器	一条指令可能两次访问存储器

规整、简单和一致等特性有利于指令的流水线执行



复习：Load指令的5个阶段

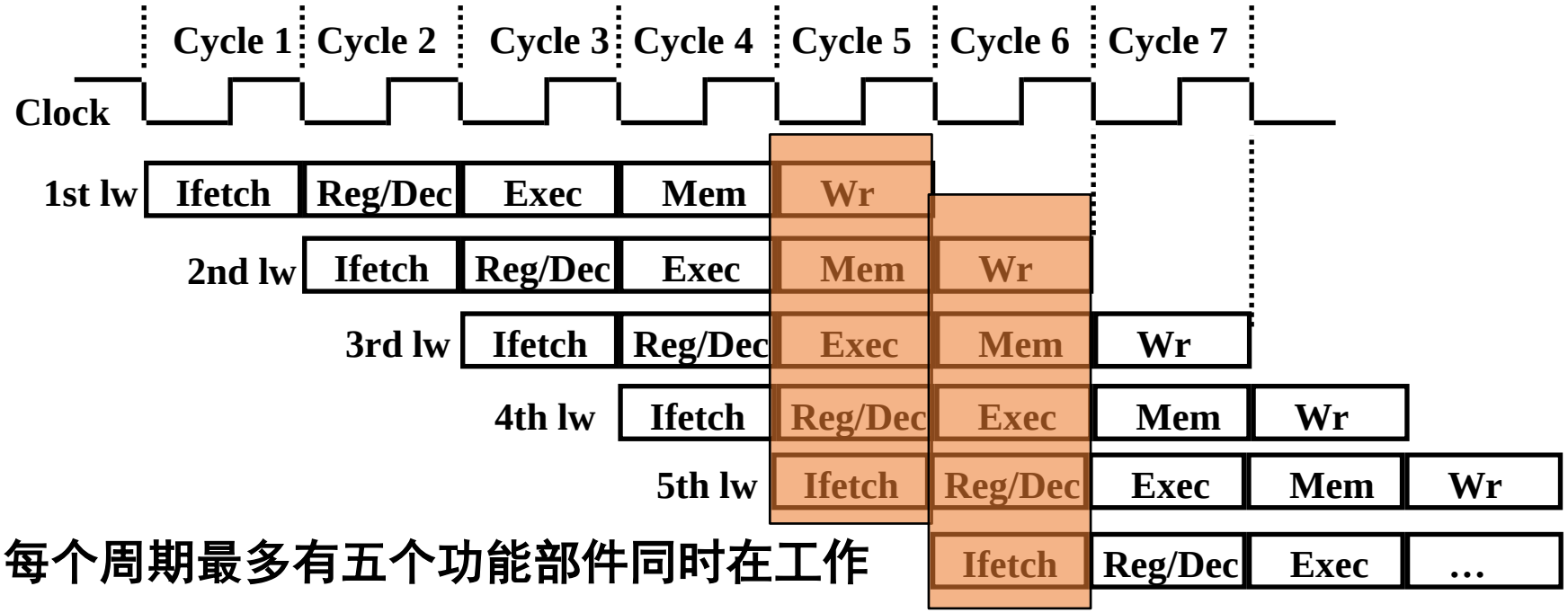


- Ifetch（取指）：取指令并计算PC+4（用到哪些部件？）
  指令存储器、Adder
- Reg/Dec（取数和译码）：取数同时译码（用到哪些部件？）
  寄存器堆读口、指令译码器
- Exec（执行）：计算内存单元地址（用到哪些部件？）
  扩展器、ALU
- Mem（读存储器）：从数据存储器中读（用到哪些部件？）
  数据存储器
- Wr（写寄存器）：将数据写到寄存器中（用到哪些部件？）
  寄存器堆写口

这里寄存器堆的读口和写口可看成两个不同的部件。

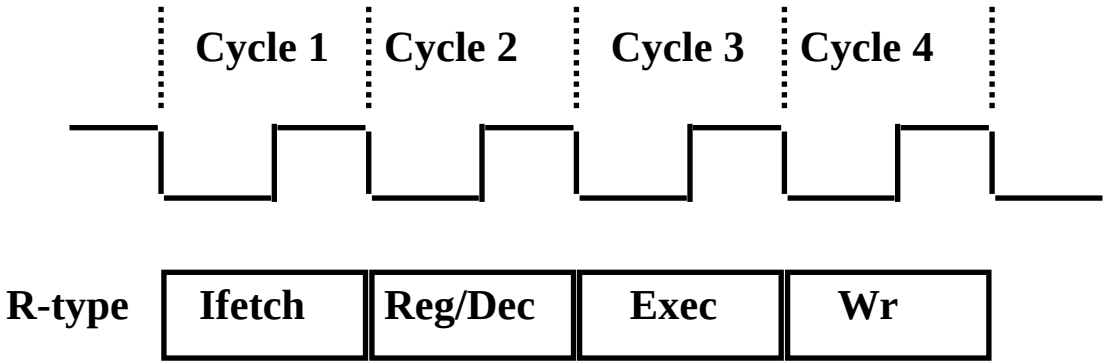
指令的执行过程中，每个阶段使用不同的部件。
是否和“洗衣”过程类似？是否可以采用类似方式来执行指令呢？

Load指令的流水线



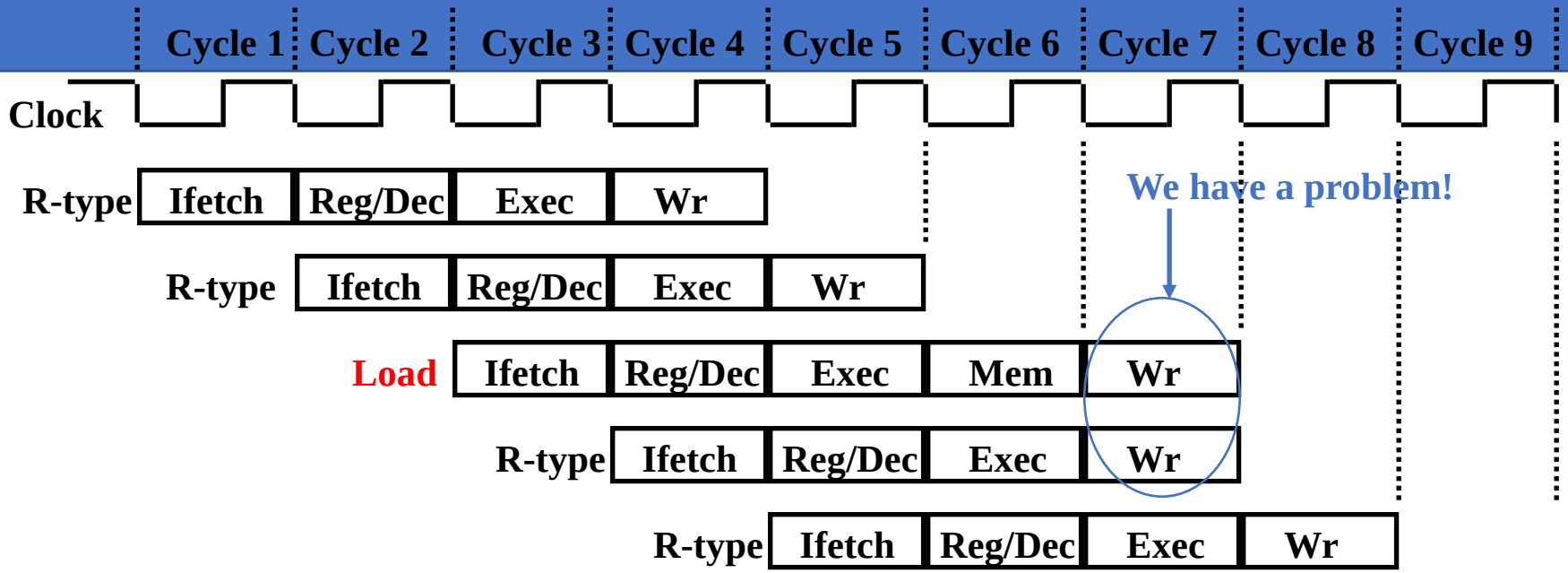
- 每个周期最多有五个功能部件同时在工作
- 后面指令在前面完成取指后马上开始
- 每个load指令仍然需要五个周期完成
- 但是吞吐率(throughput)提高许多，理想情况下，有：
  - 每个周期有一条指令进入流水线
  - 每个周期都有一条指令完成
  - 每条指令的有效周期(CPI)为1

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

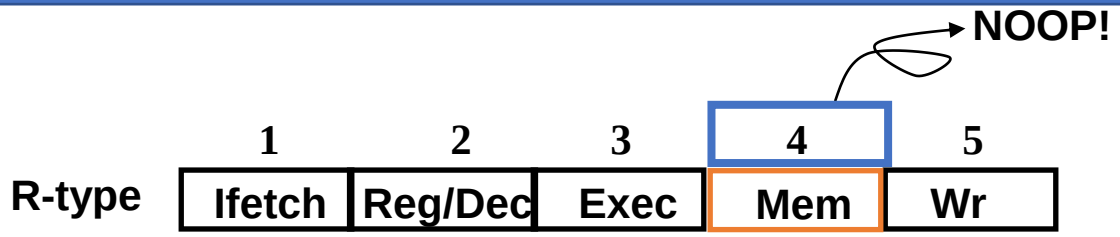


- 上述流水线有个问题：两条指令试图同时写寄存器，因为
  - Load在第5阶段用寄存器写口
  - R-type在第4 阶段用寄存器写口

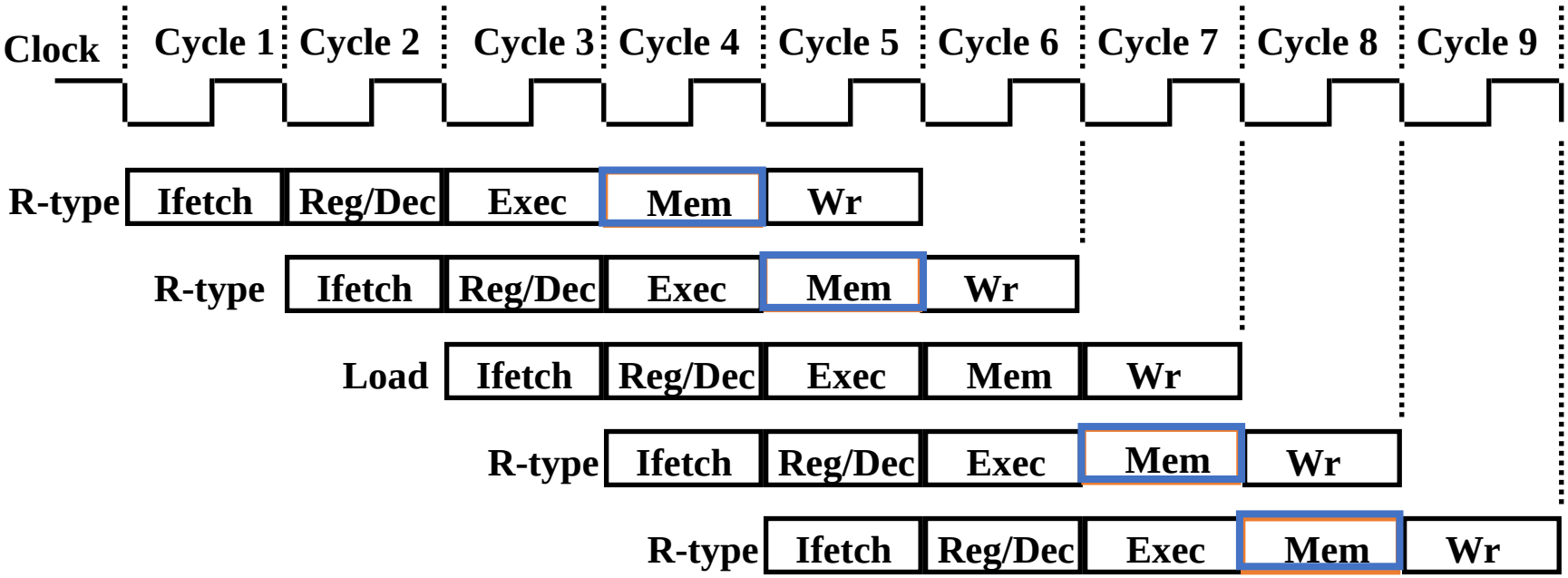
或称为资源冲突！

- 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）

解决方案：R-type的Wr操作延后一个周期执行

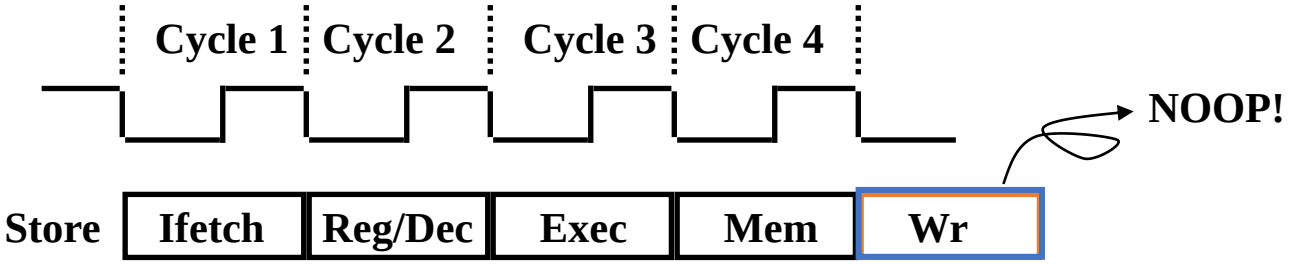


- 加一个NOP阶段以延迟“写”操作：
  - 把“写”操作安排在第5阶段，这样使R-Type的Mem阶段为空NOP



这样使流水线中的每条指令都有相同多个阶段!

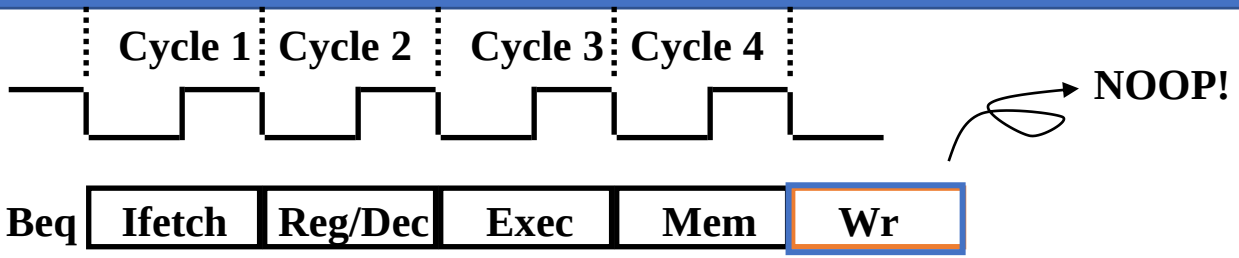
Store指令的四个阶段



- 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

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

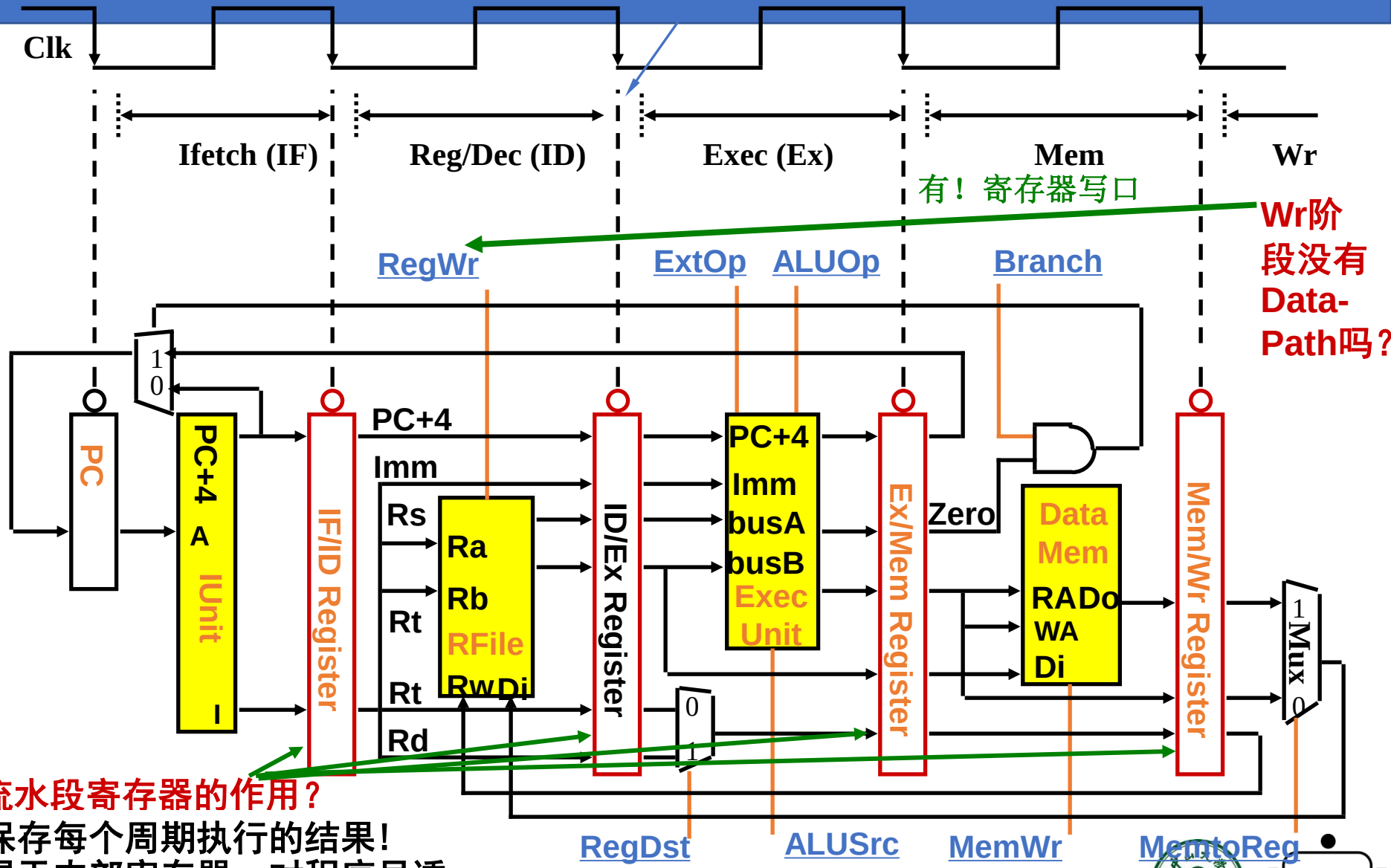
为什么流水线中不进行“投机”计算？

- **Wr:** 加一个空写阶段，使流水线更规整！

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

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

A Pipelined Datapath（五阶段流水线数据通路）



流水段寄存器的作用？

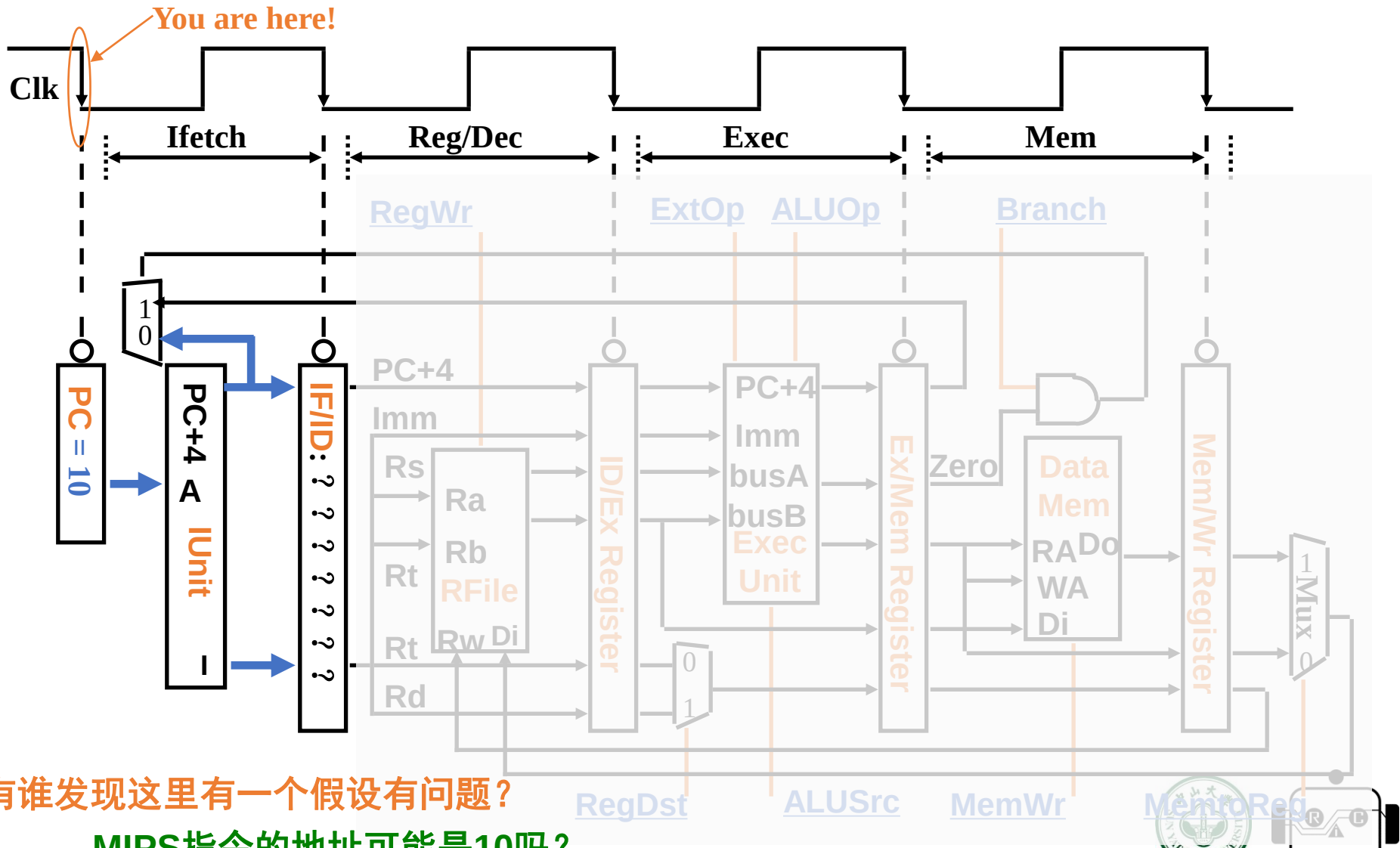
保存每个周期执行的结果！

属于内部寄存器，对程序员透明，不需作为现场保存

下面看每条指令在流水线通路中的执行过程

取指令（Ifetch）阶段——开始时

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

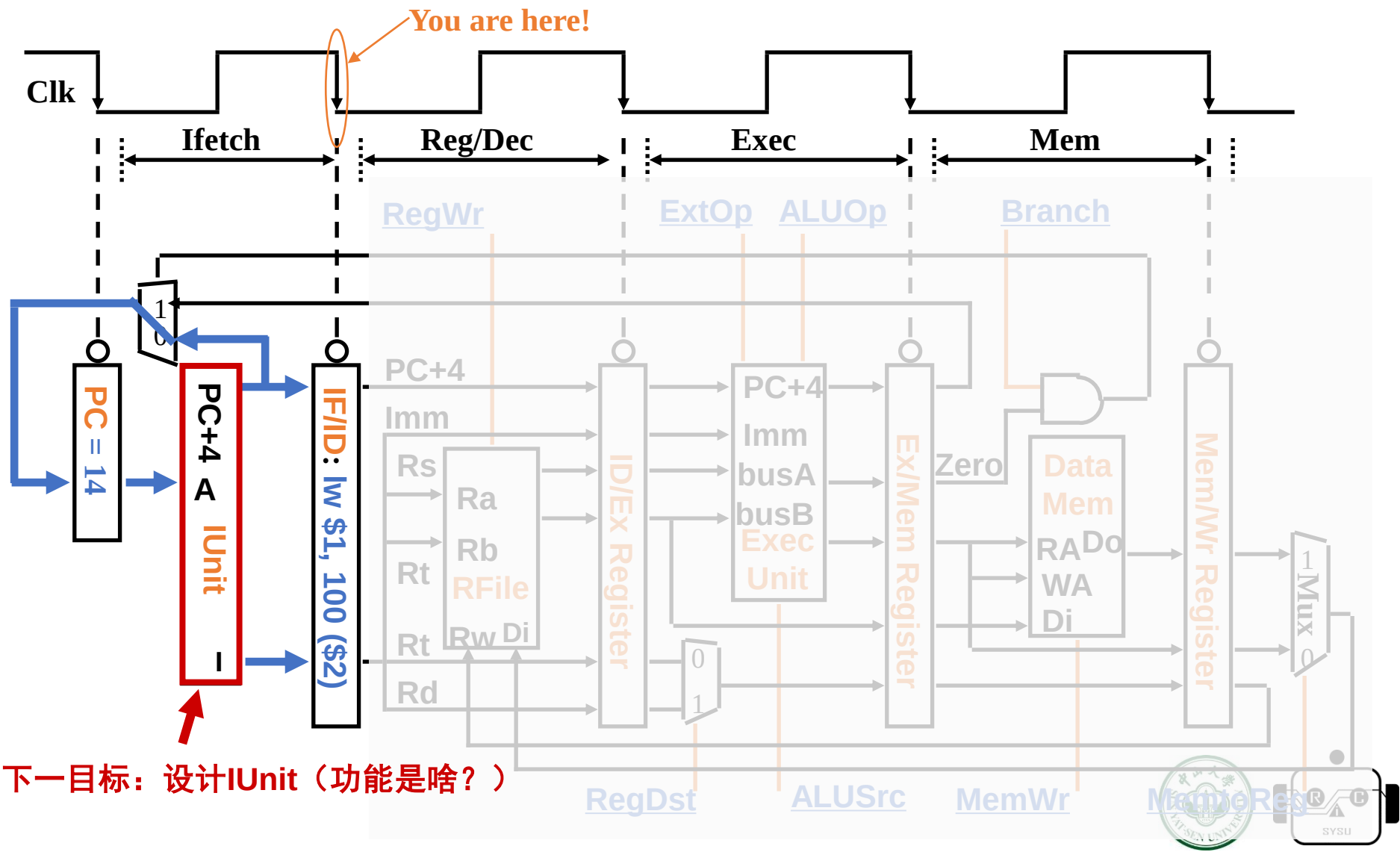


有谁发现这里有一个假设有问题？

MIPS指令的地址可能是10吗？

取指令（Ifetch）阶段——结束后

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



指令部件 IUnit的设计

- 第10单元指令：： lw $1, 0x100($2)

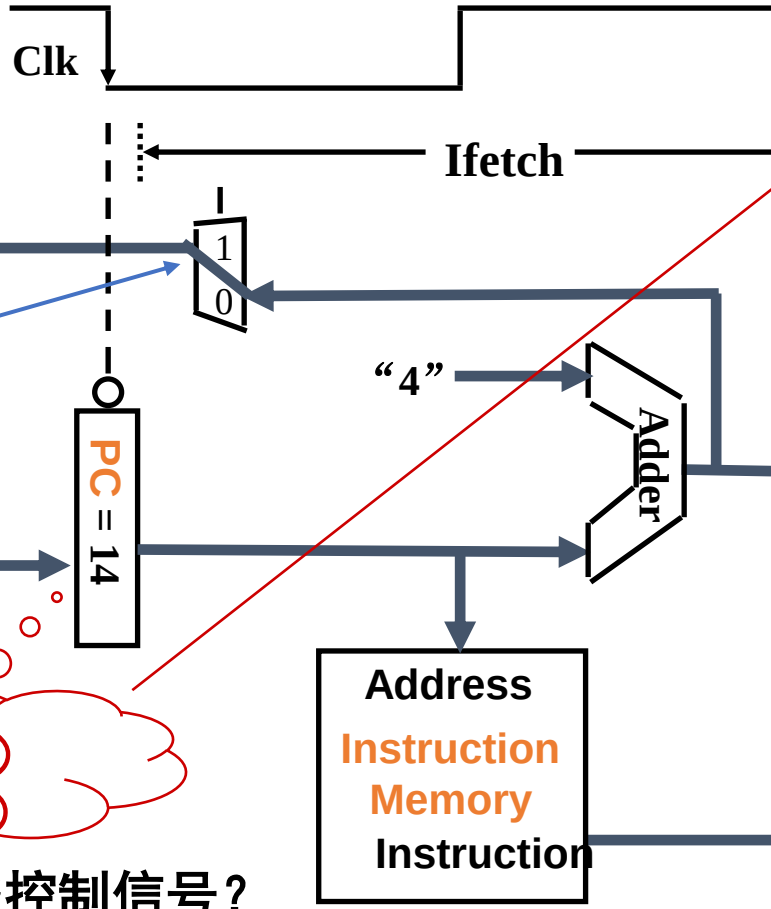
随后的指令在14号单元中！

You are here!

指令部件功能

Instr ← Mem[PC]

PC← PC+4



开始从14号单元取指！

这里MUX的控制信号由其他阶段产生！

流水段寄存器用来存放各阶段的执行结果

总是在下个时钟到来后的Clock-to-Q写入

PC 新值（14）≠旧输出（10）

应把哪些信息存到流水段寄存器IF/ID中？

取指阶段有哪些控制信号？

应保存后面阶段用到的指令和PC+4的值！

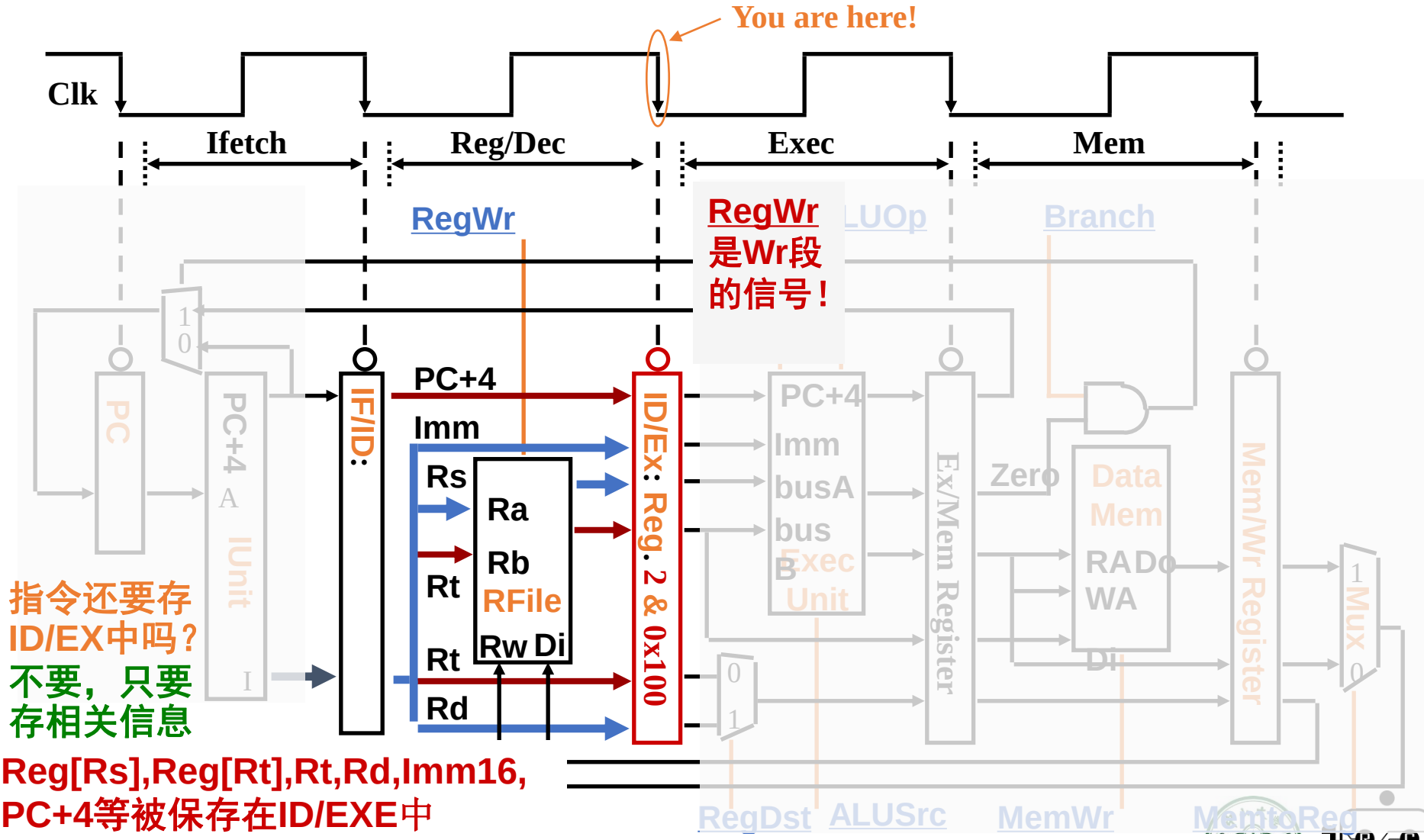
不需控制信号，因为每条指令执行功能一样，是确定的，无需根据指令的不同来控制执行不同的操作！

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

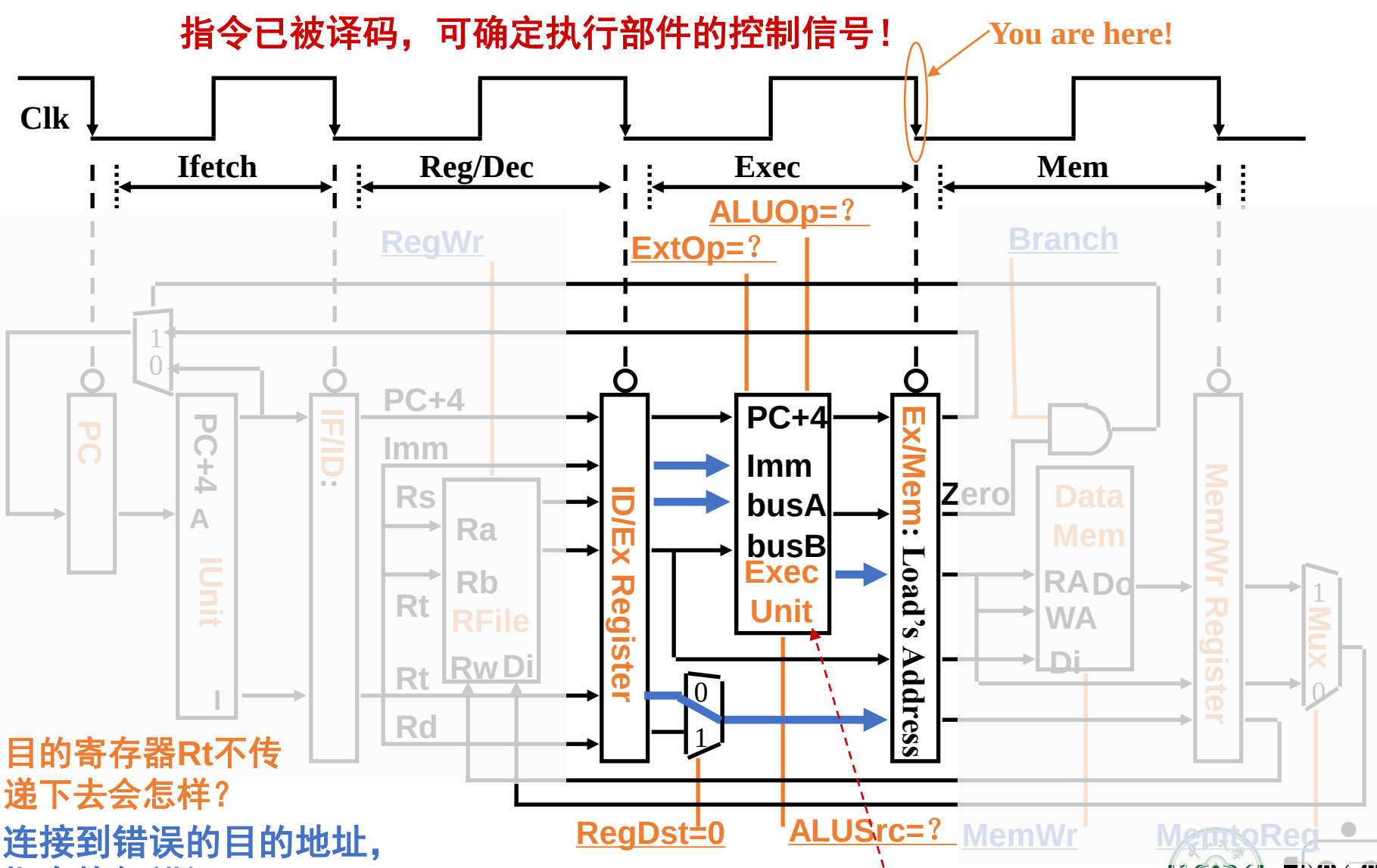
- 第10单元指令：lw $1, 0x100($2)　　功能：$1 ← Mem[($2) + 0x100]



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

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



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=add, Extop=1

Branch指令呢？

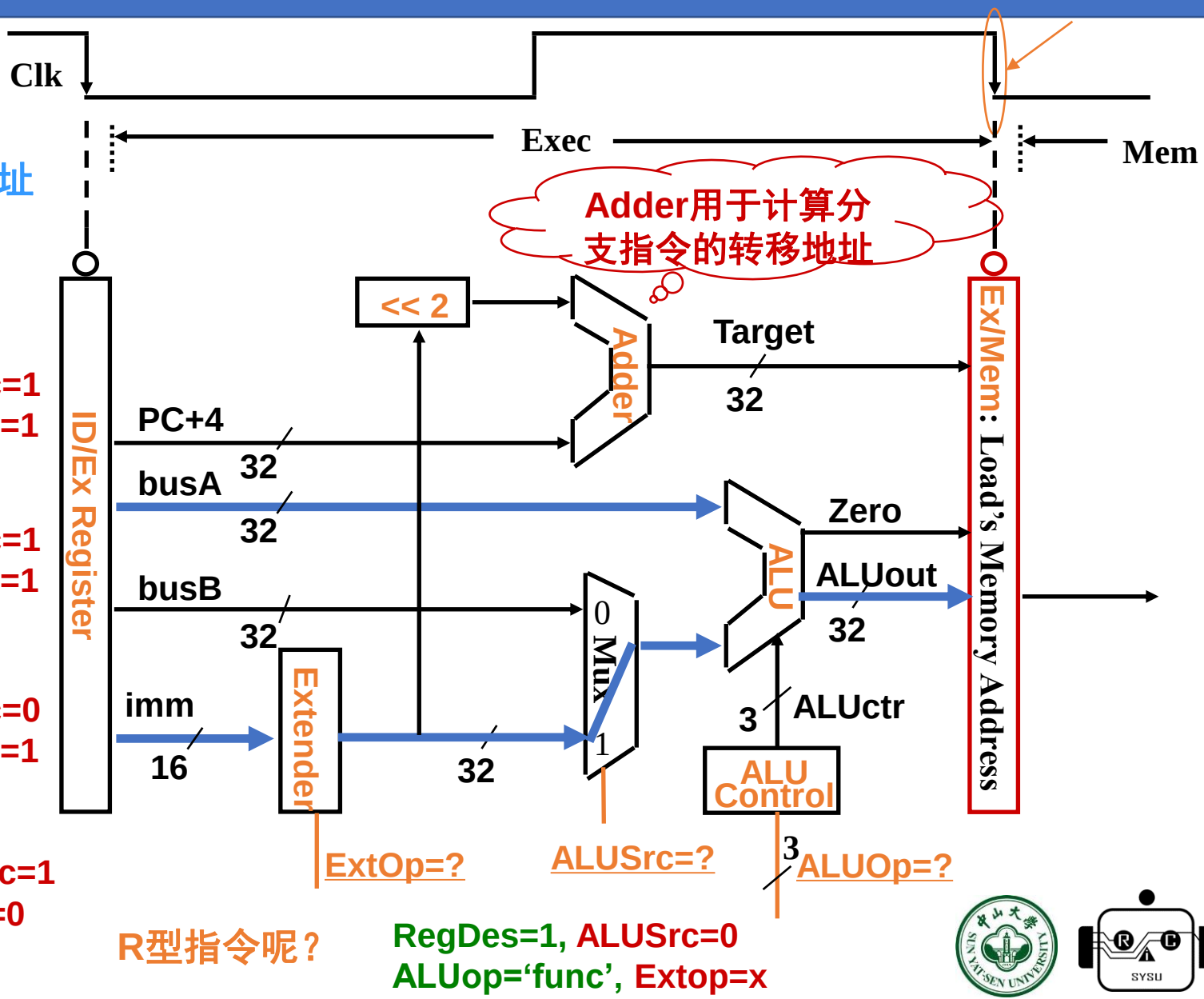
RegDes=x, ALUSrc=0
ALUop=sub, Extop=1

Ori指令呢？

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

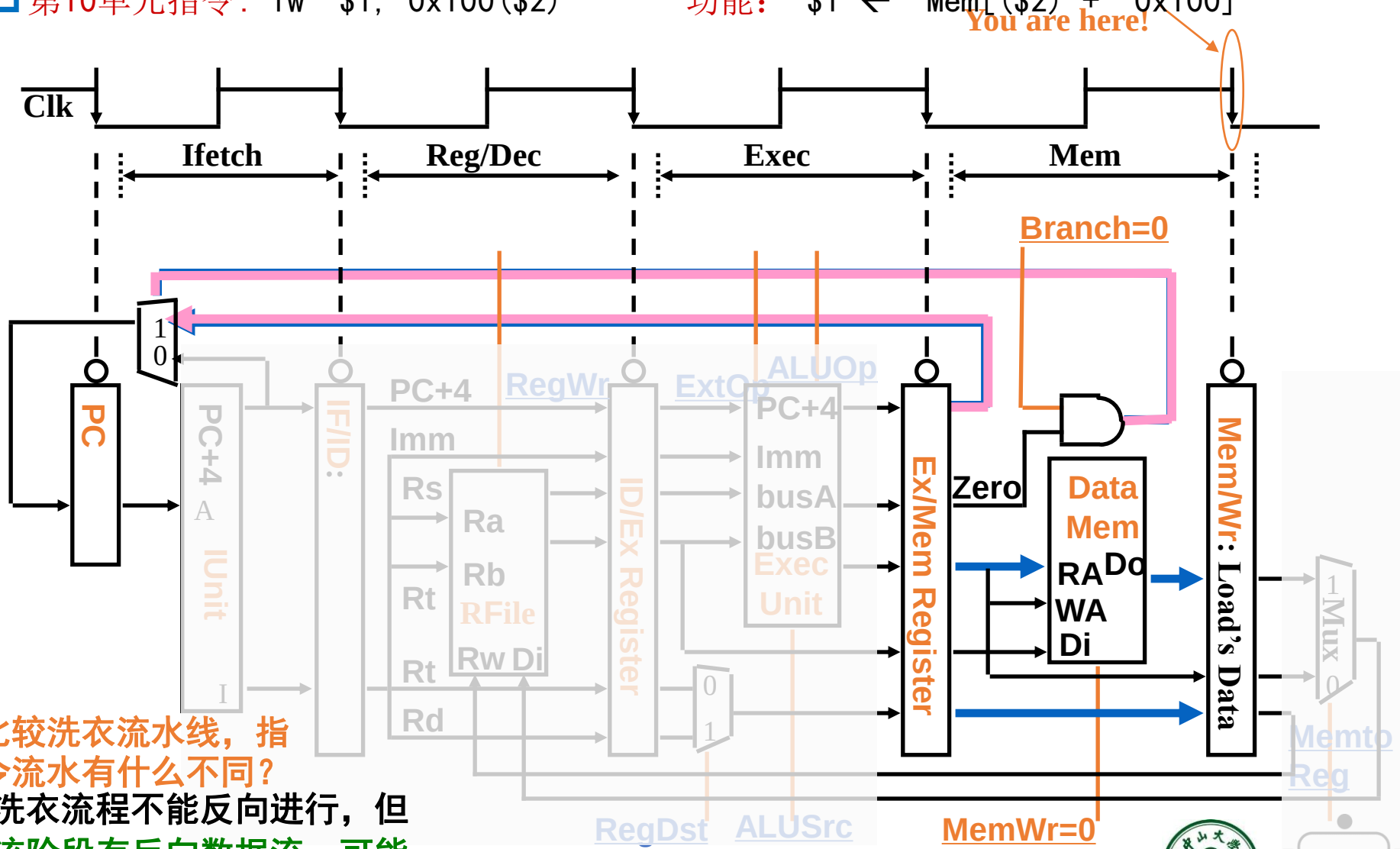
R型指令呢？

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



Load指令的存储器读(Mem)周期

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



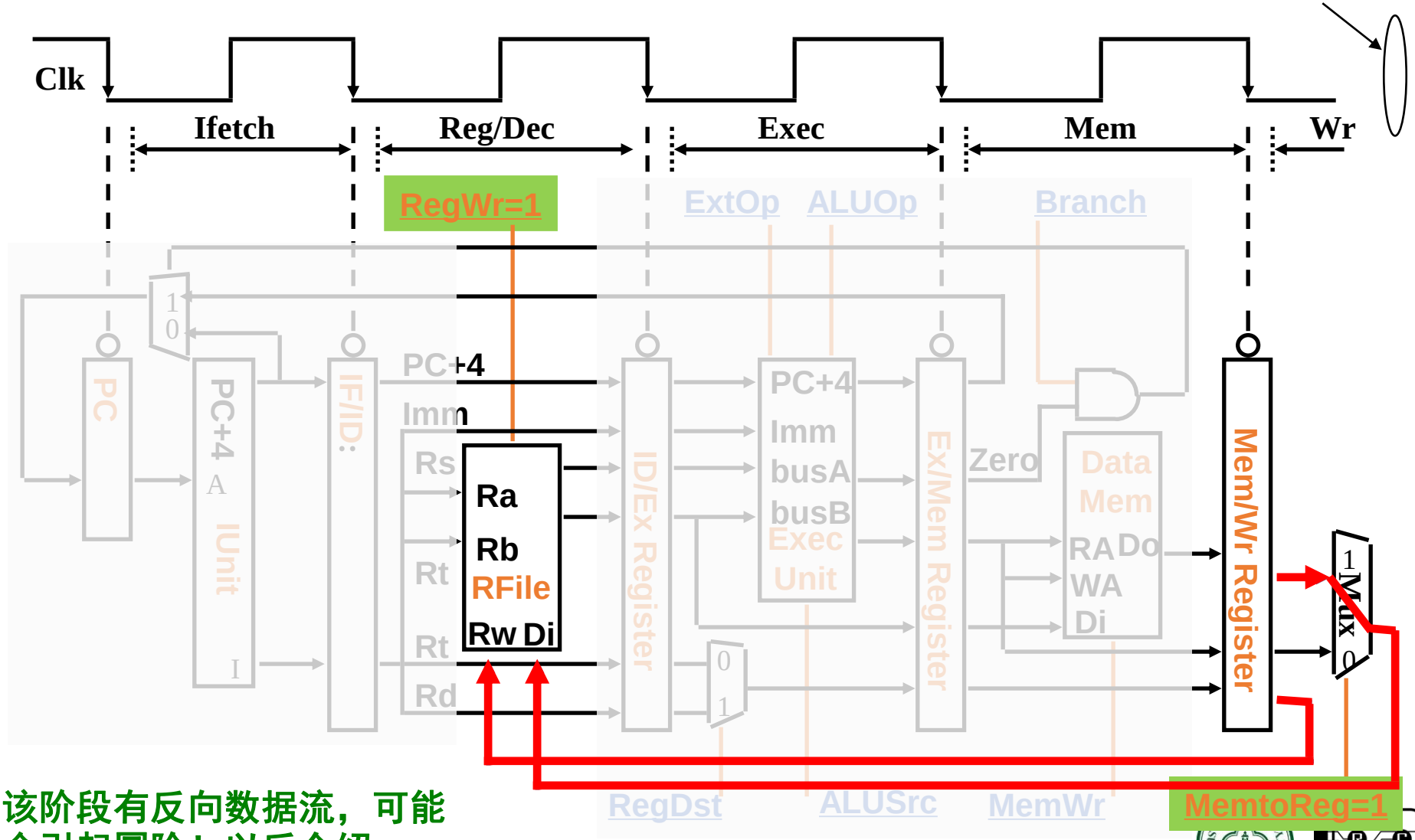
比较洗衣流水线，指令流水有什么不同？

洗衣流程不能反向进行，但该阶段有反向数据流，可能会引起冒险！以后介绍。

周期以最长操作为准设计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]



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？

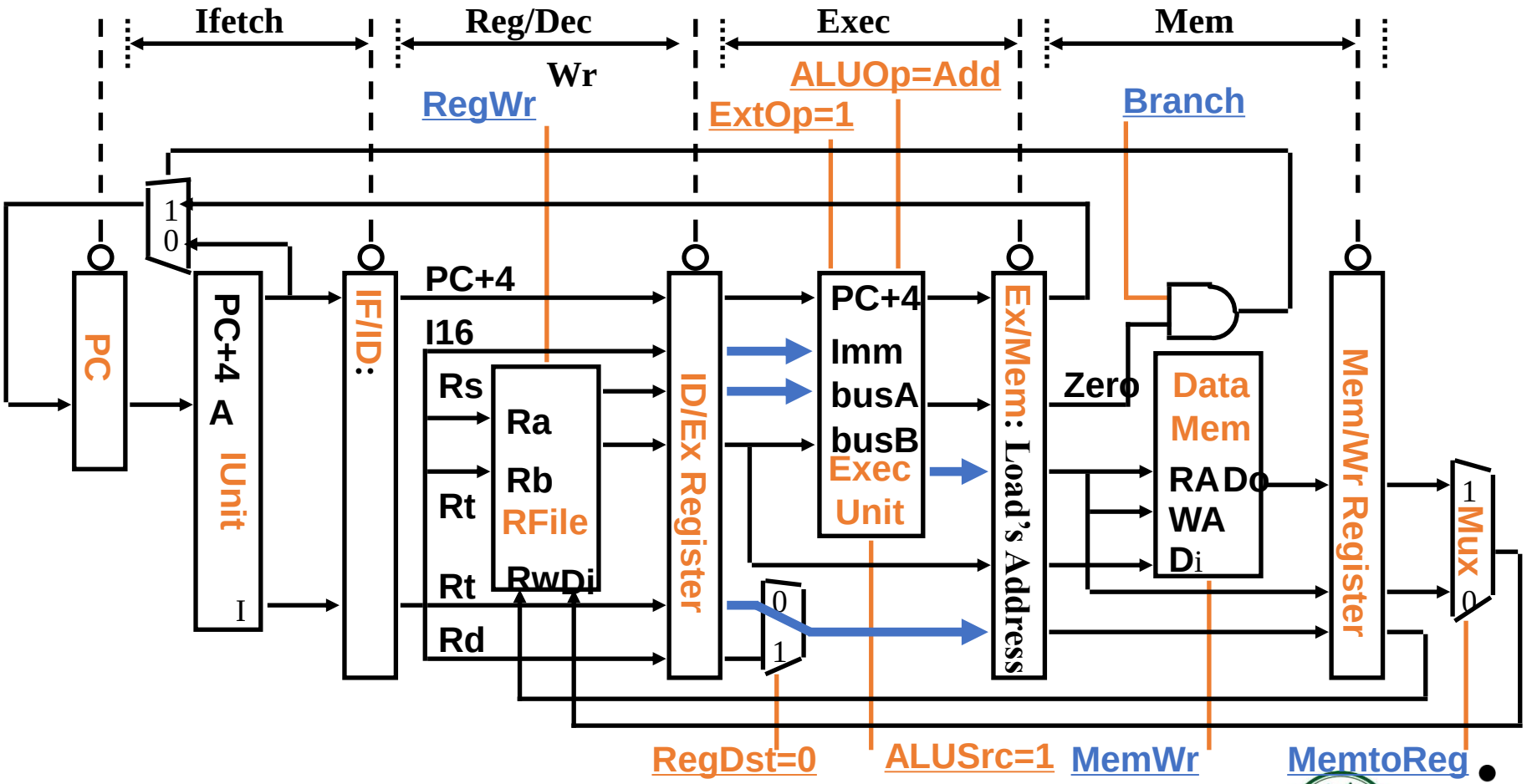
Pipelined datapath:控制信号怎么办?

- 控制信号在**RF/ID**阶段产生—由控制逻辑电路译码生成
- 控制信号也必须流水线化
  - 针对**EX**阶段的控制信号
    - ExtOP, ALUSrc等；
      - 一个时钟周期后用到
    - 针对**Memory**的控制信号
      - 两个周期后用到
    - MemWr, Branch；
  - 针对寄存器回写的控制信号
    - MemToReg, RegWr等；
      - 三个周期后用到

(只有这三个阶段有控制信号）

流水线中的Control Signals如何获得?

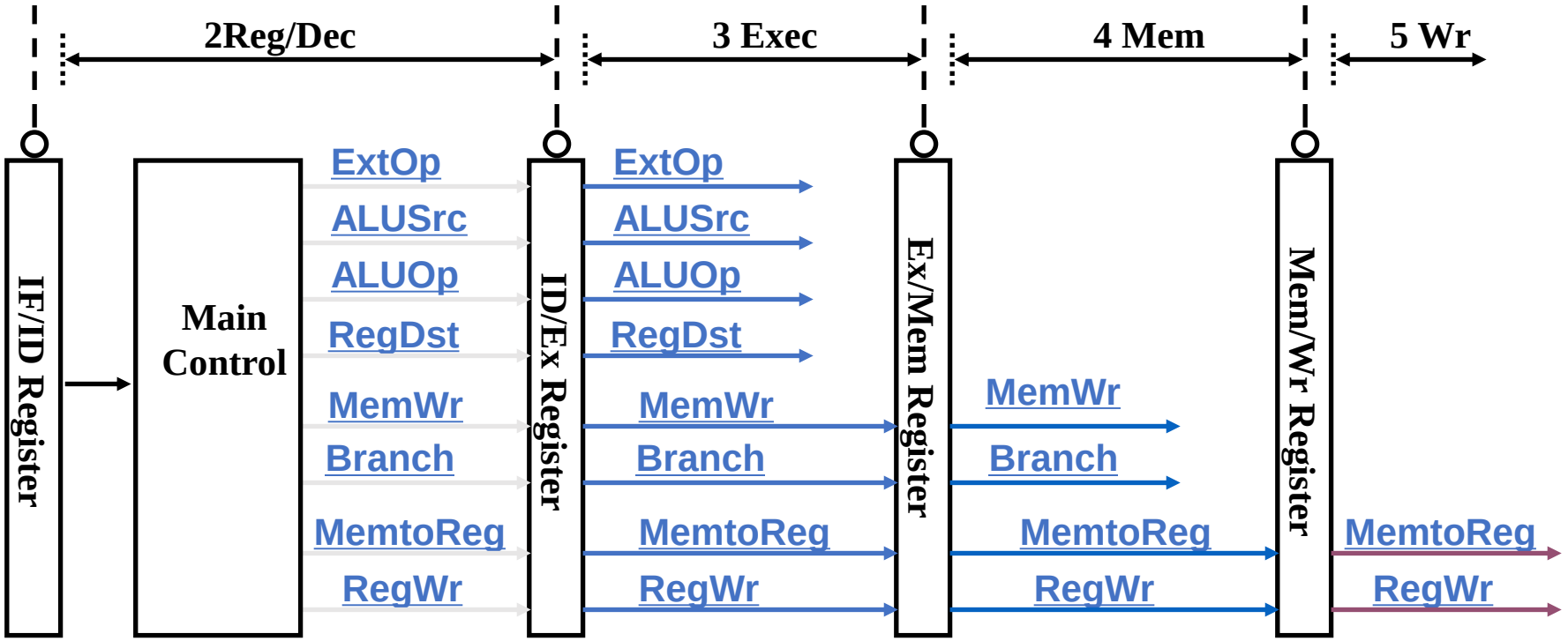
- 例：Load的Exec段的控制信号如图中所示



流水线中的Control Signals如何获得?

- 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！

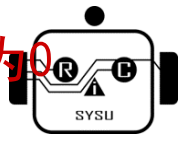


各流水段部件在一个时钟内完成某条指令的某个阶段的工作！

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

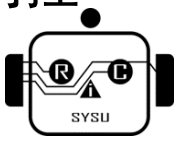
- 通过对前面流水线数据通路的分析，得知：
  - 因为每个时钟都会改变PC的值，所以PC不需要写控制信号
  - 流水段寄存器每个时钟都会写入一次，也不需要写控制信号（不考虑阻塞）
  - Ifecth阶段和Dec/Reg阶段都没有控制信号，因为功能都一样
  - Exec阶段的控制信号有四个
    - ExtOp（扩展器操作）：1- 符号扩展；0- 零扩展
    - ALUSrc（ALU的B口来源）：1- 来源于扩展器；0- 来源于BusB
    - ALUOp（主控制器输出，用于辅助局部ALU控制逻辑来决定ALUCtrl）
    - RegDst（指定目的寄存器）：1- Rd；0- 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

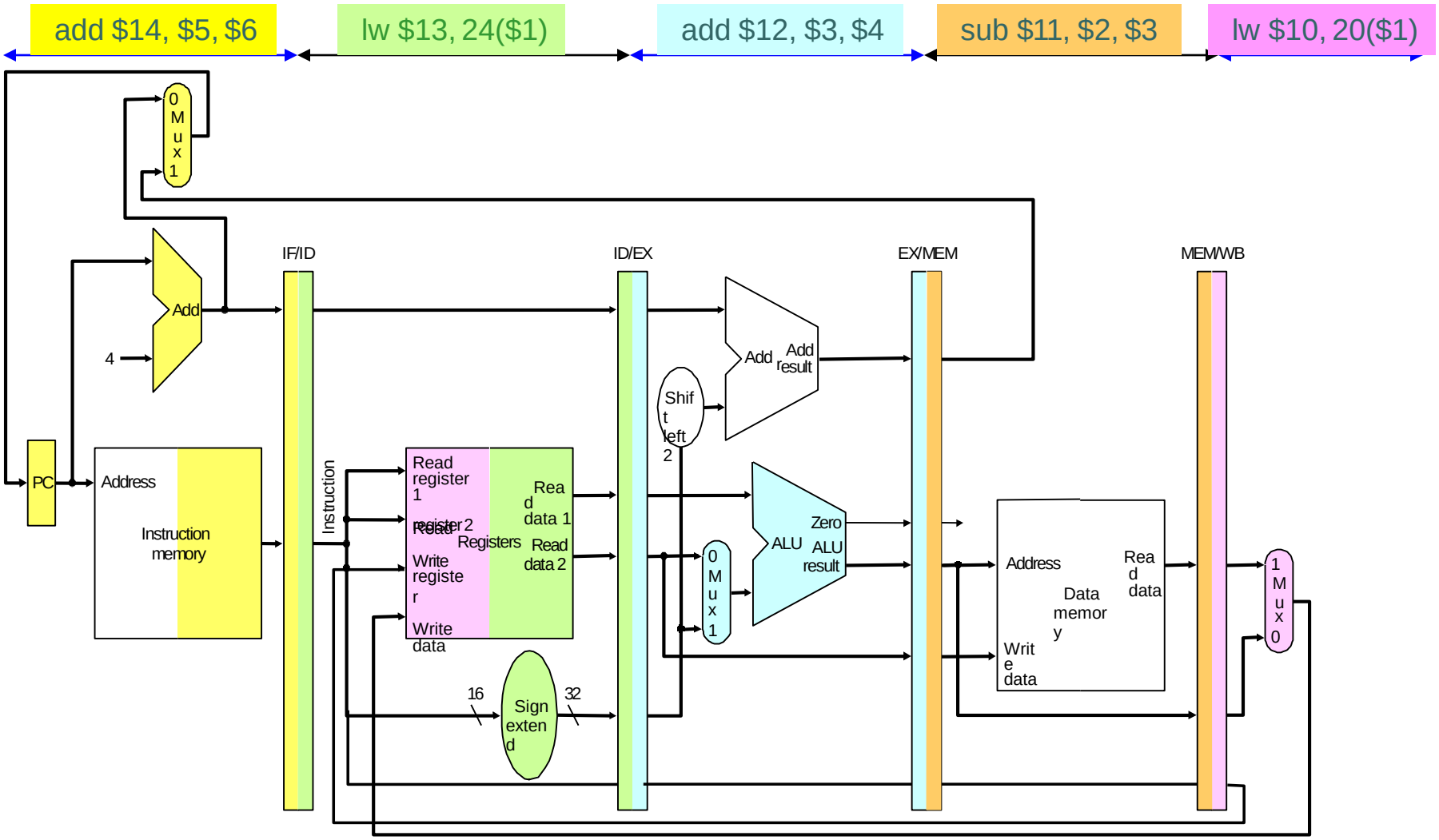
控制逻辑(Control)的设计

- 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
  - 与单周期控制逻辑设计类似
  - 设计过程
    - 控制逻辑分成两部分
      - **主控制逻辑：生成ALUop和其他控制信号**
      - **局部ALU控制逻辑：根据ALUop和func字段生成ALUCtrl**
    - 用真值表建立指令和控制信号之间的关系
    - 写出每个控制信号的逻辑表达式
  - 控制逻辑的输出（控制信号）在ID阶段生成，并存放在ID/EX流水段寄存器中，然后每来一个时钟跟着其它数据和信息传送到下一级流水段寄存器
  - 在同一时刻，不同阶段同时执行不同指令，不同的指令得到不同控制信号

请复习单周期和多周期控制器设计。。。



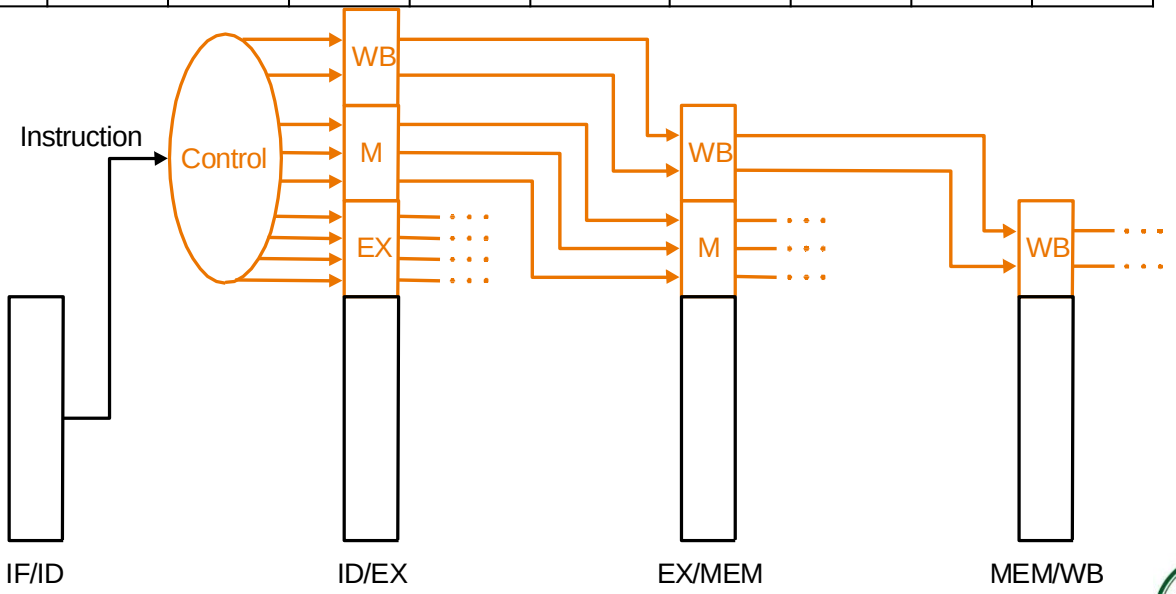
Pipelining Example



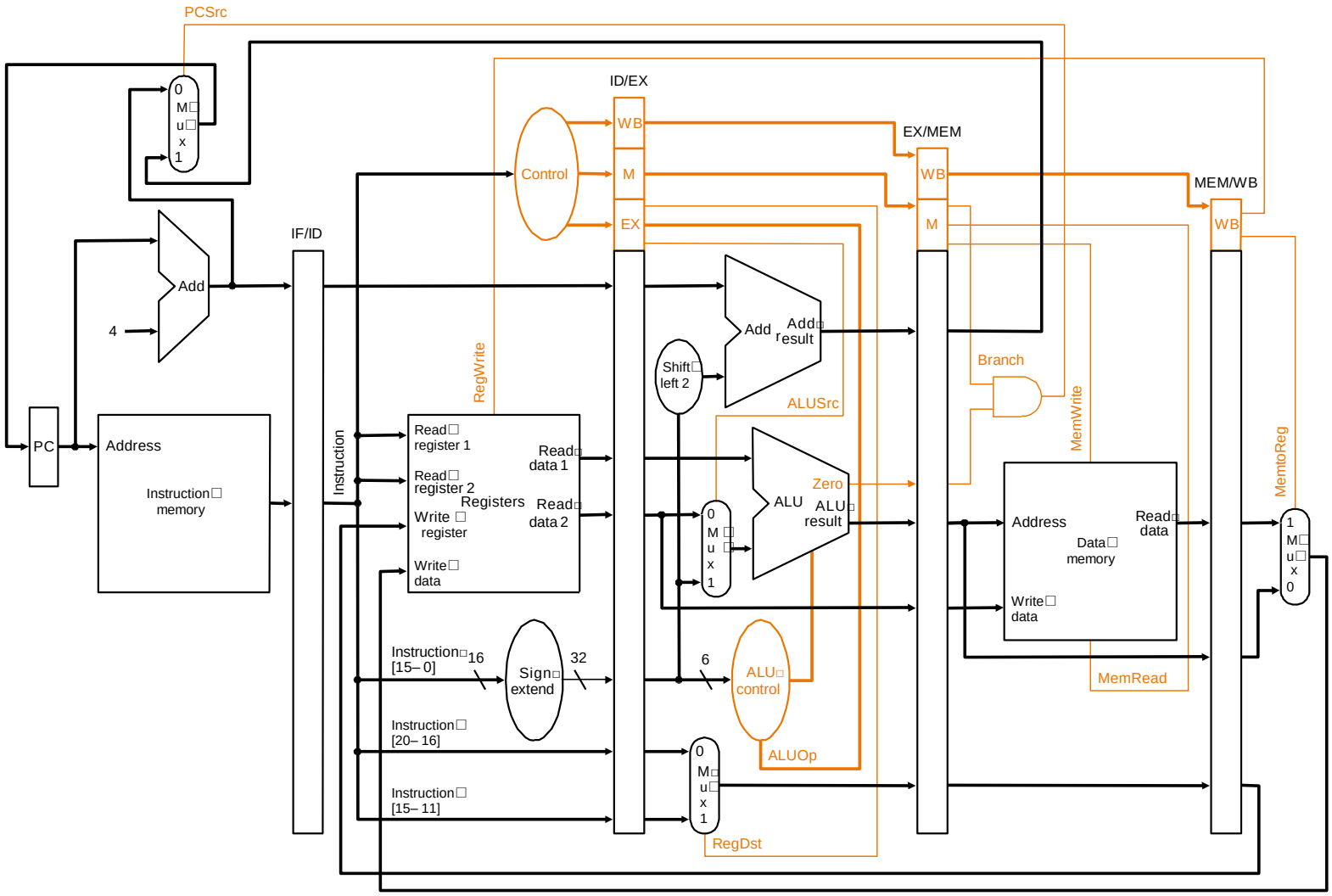
Pipeline Control

- Extend pipeline registers to include control information created in ID stage
- Pass control signals along just like the data

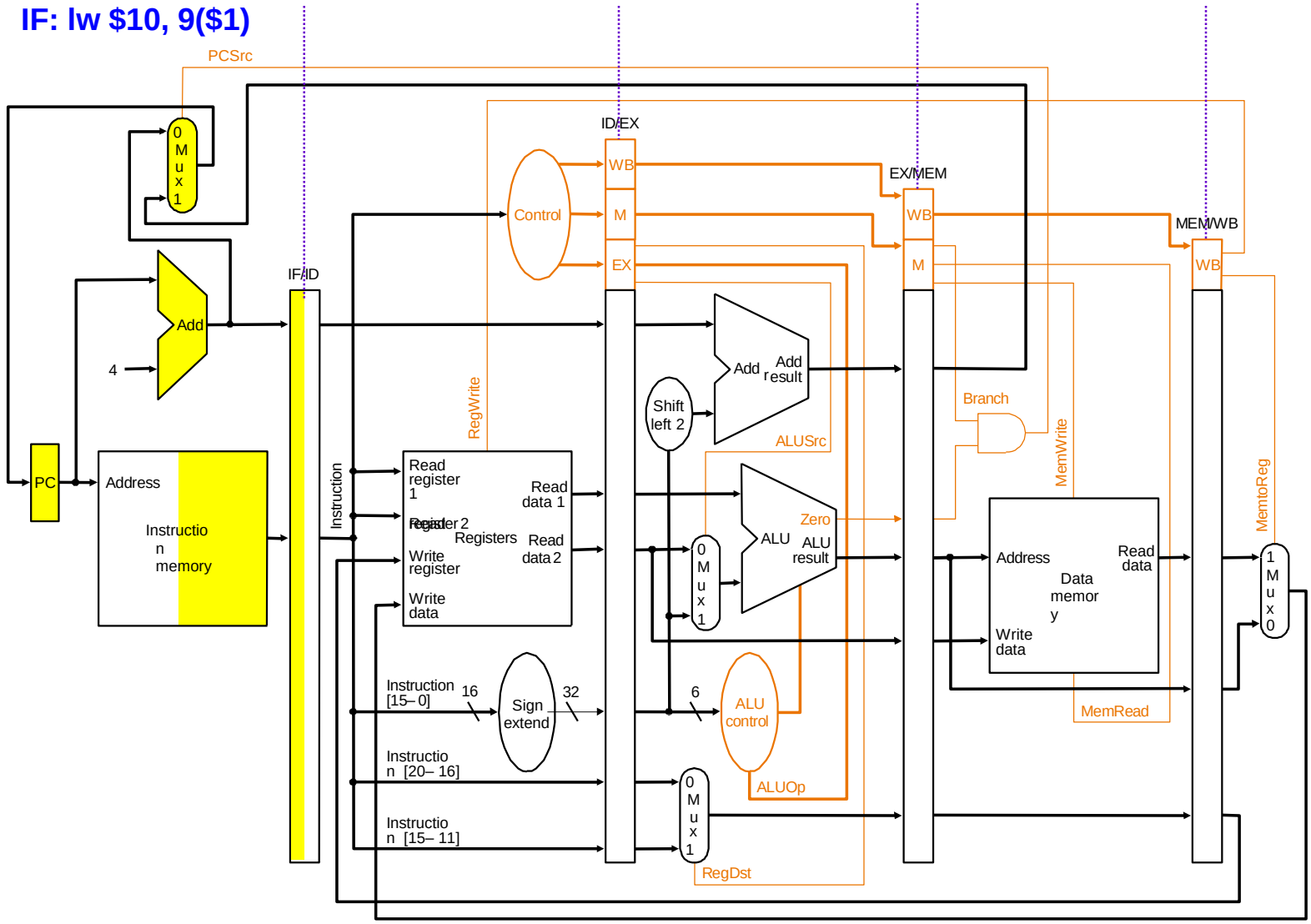
Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	0	1	0	0	0	X



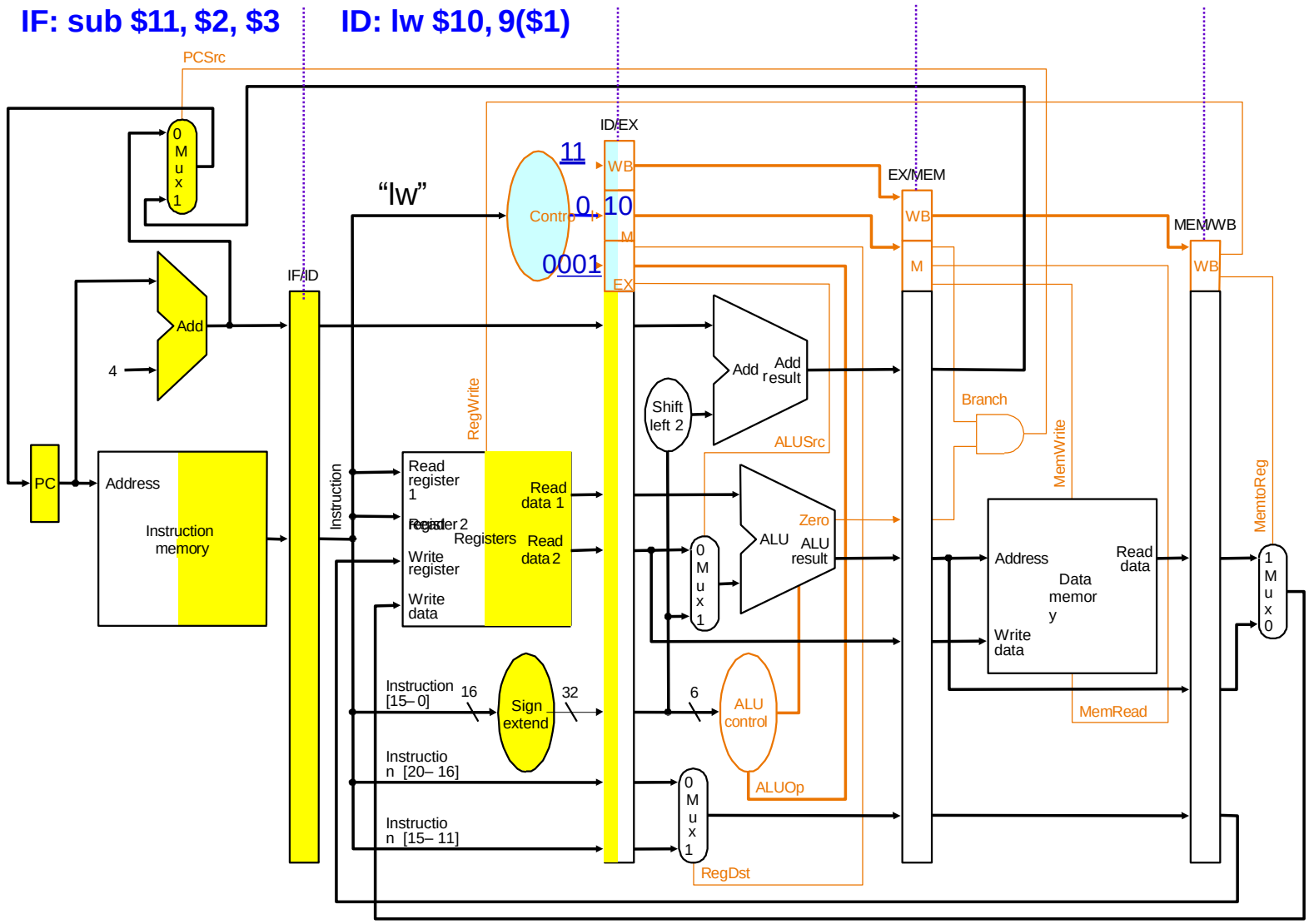
Datapath with Control



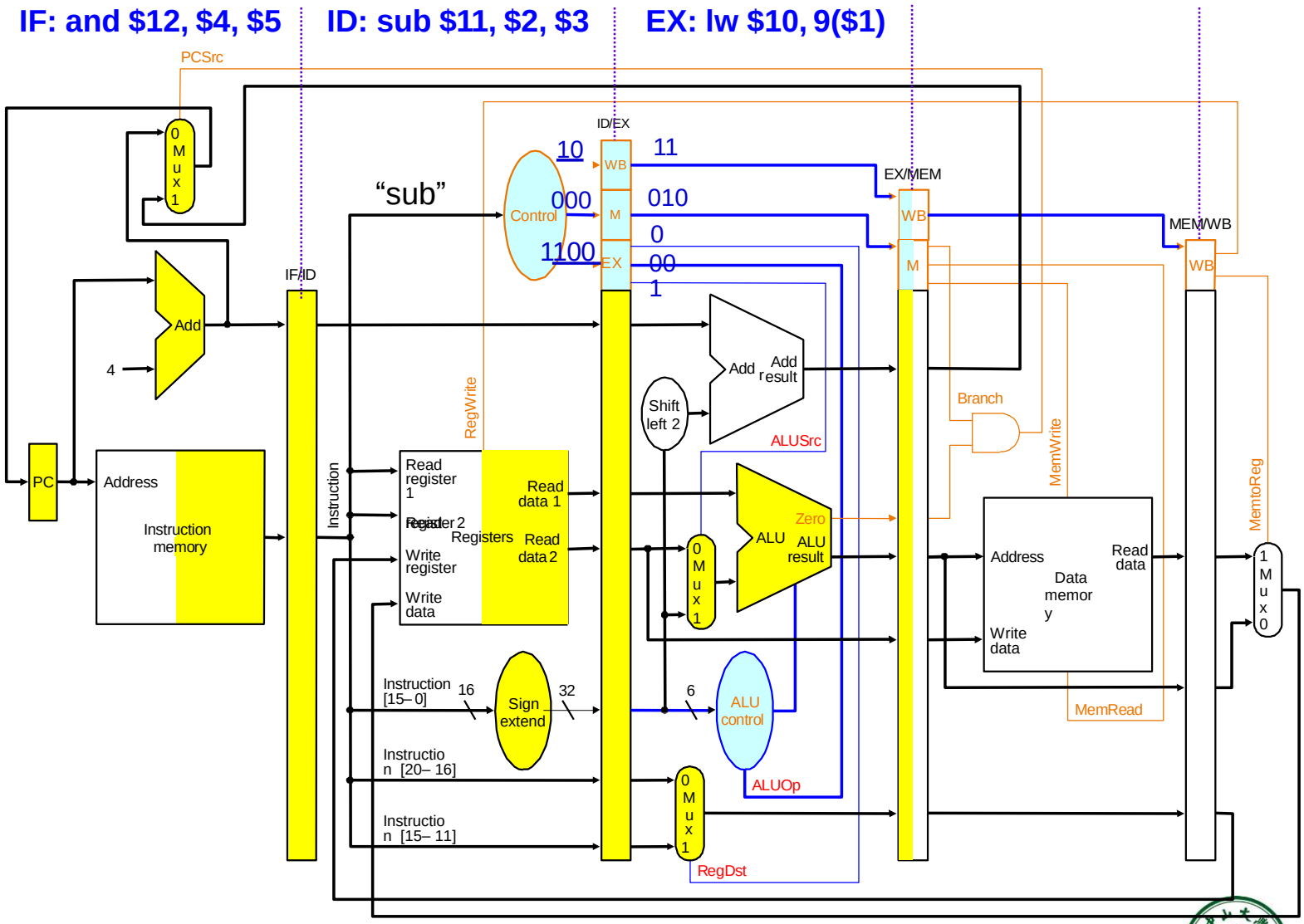
Datapath with Control



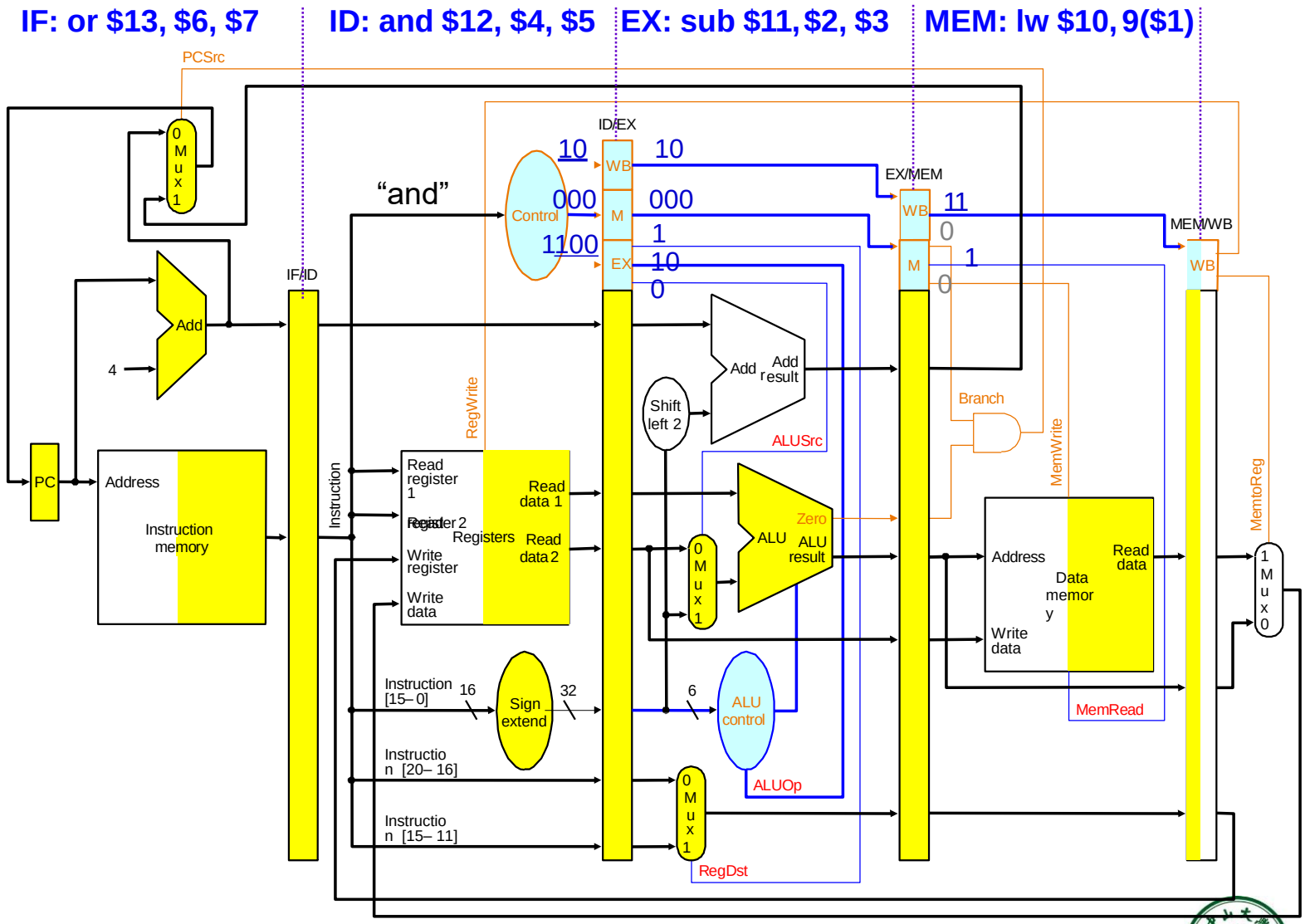
Datapath with Control



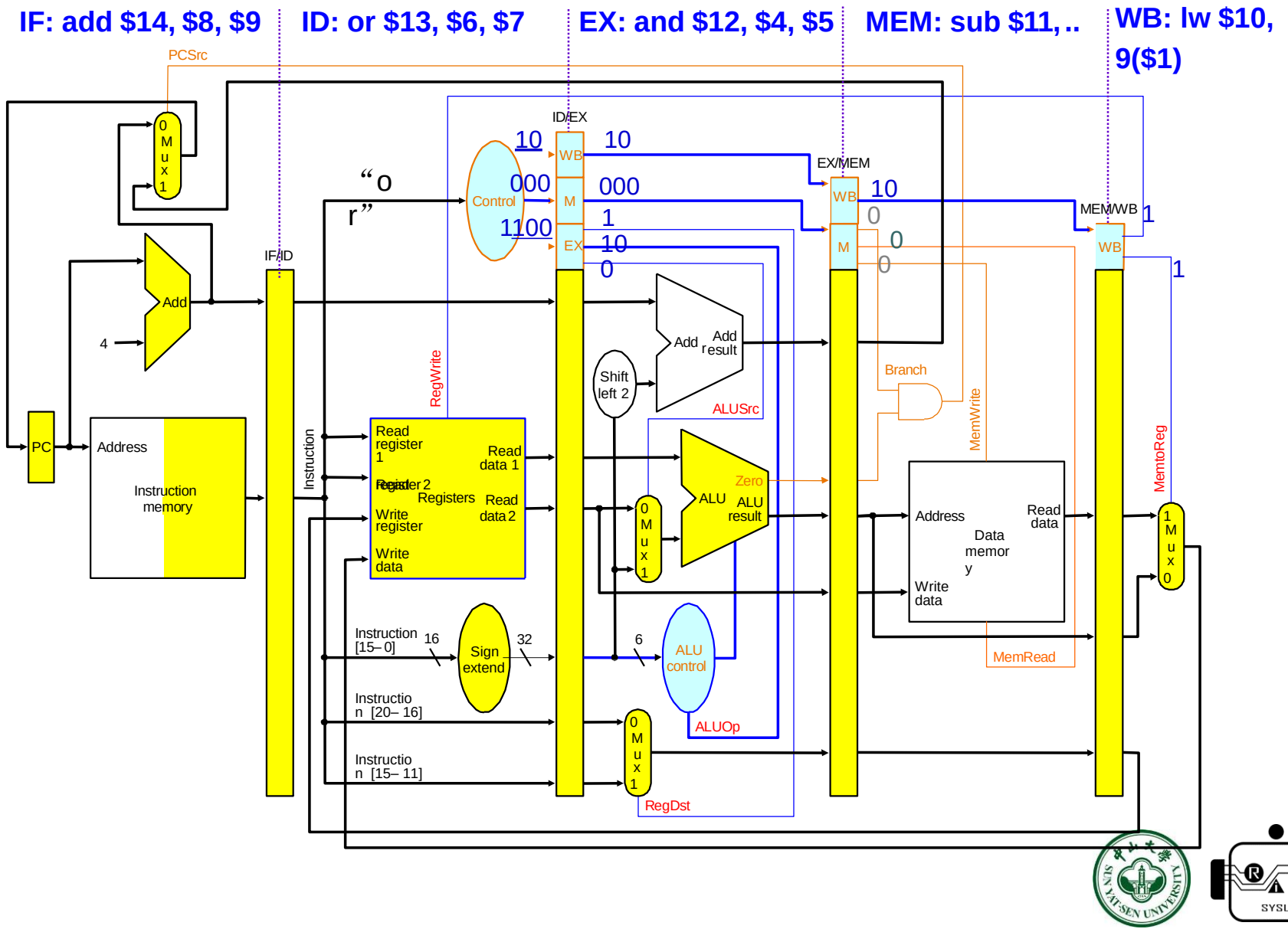
Datapath with Control



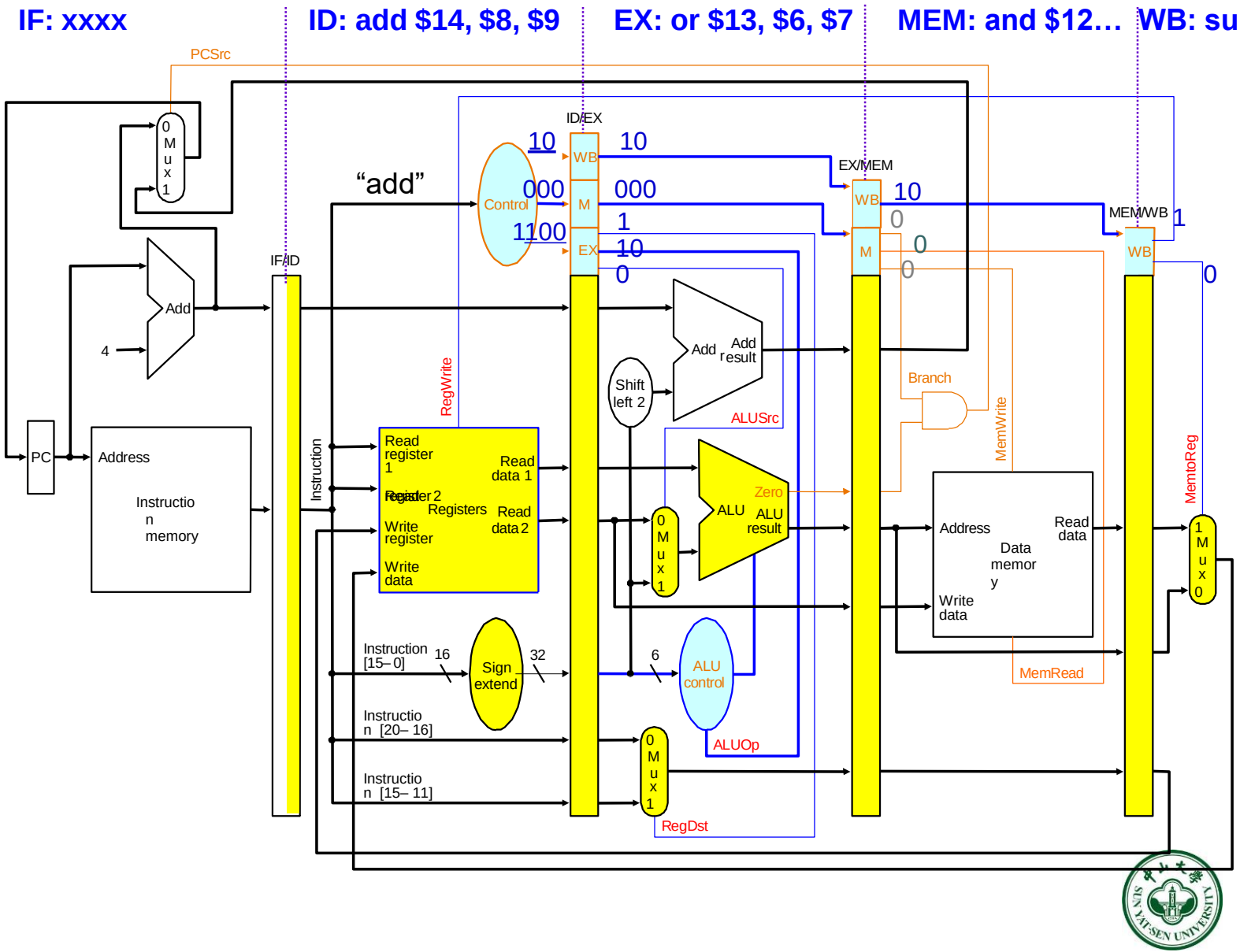
Datapath with Control



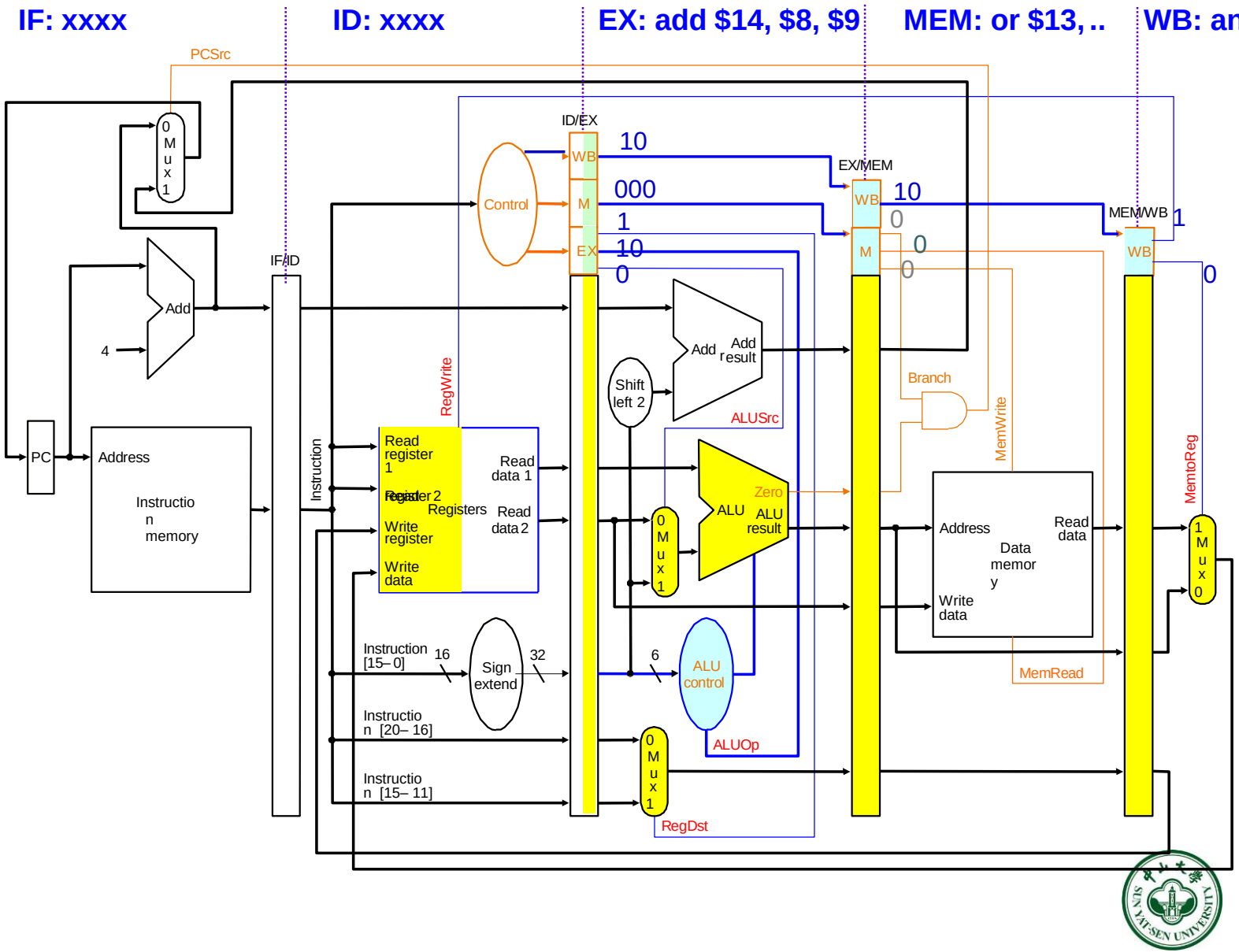
Datapath with Control



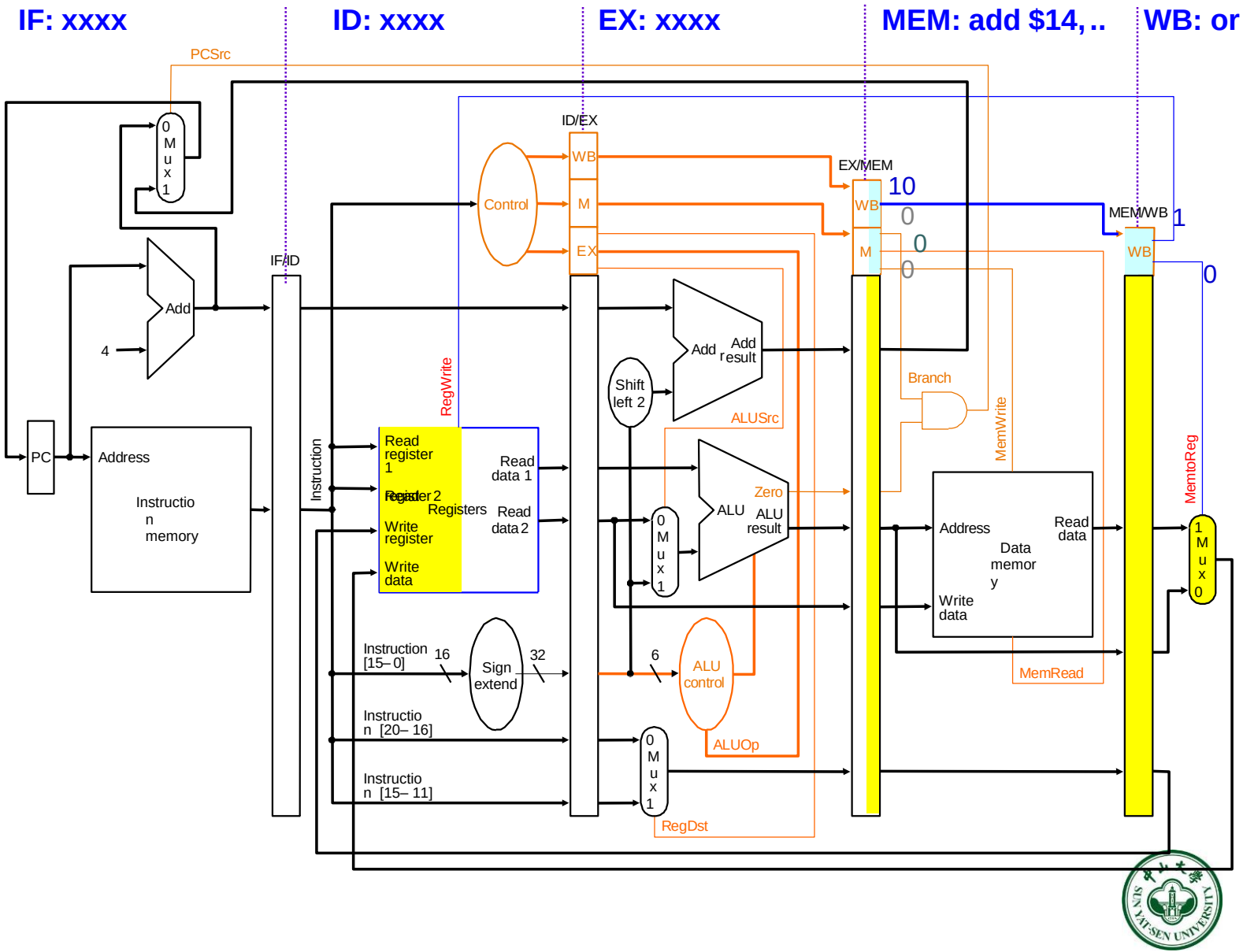
Datapath with Control



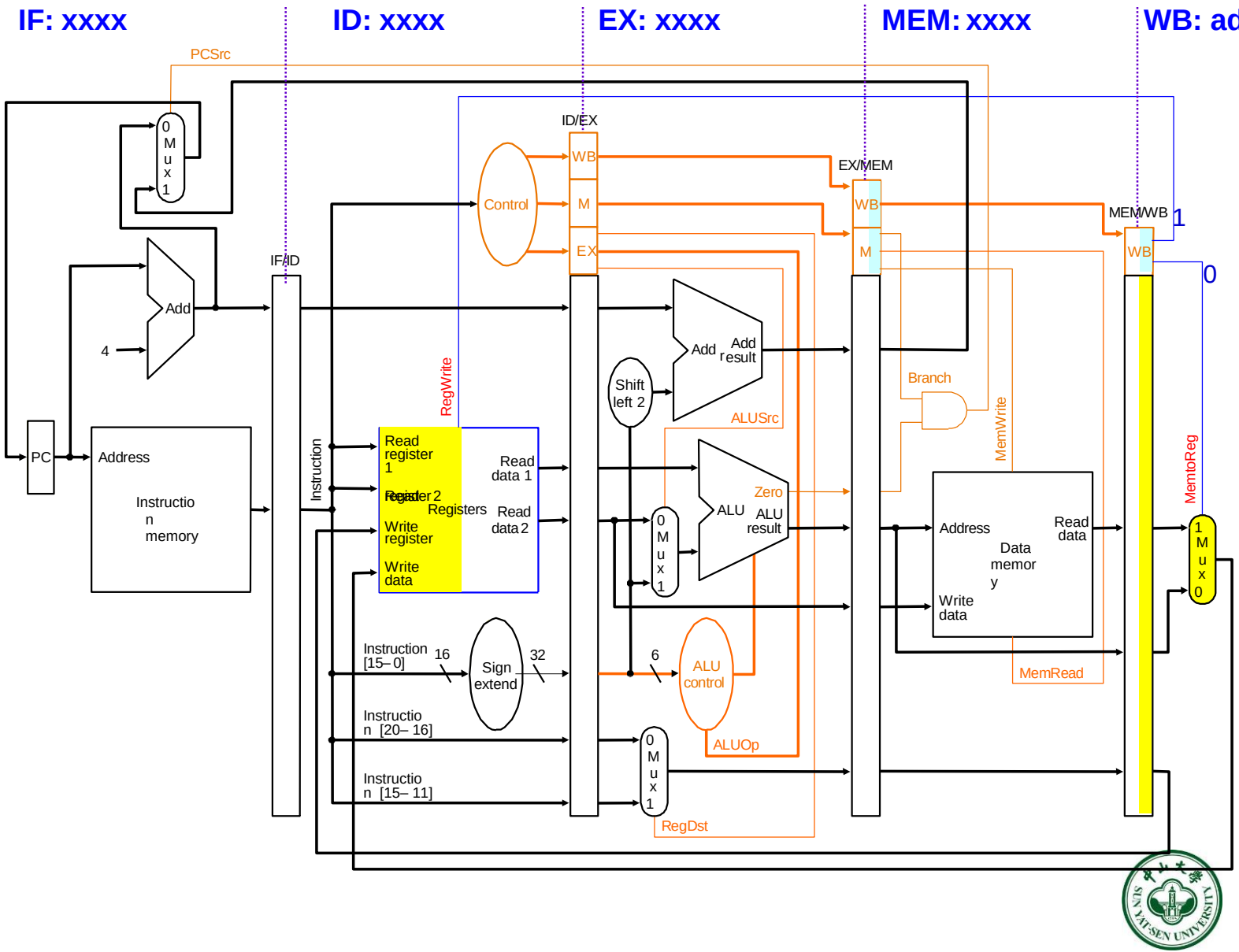
Datapath with Control



Datapath with Control



Datapath with Control



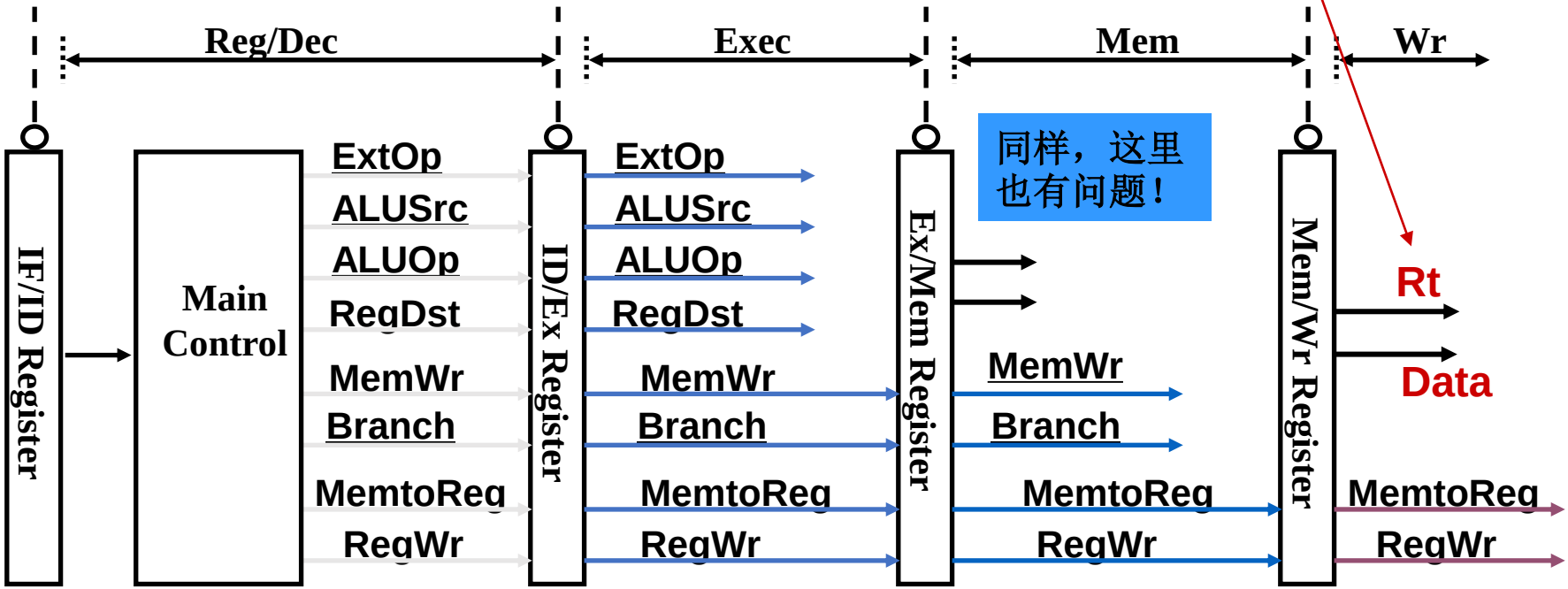
控制信号在流水线中的传递以及带来的问题

- 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？



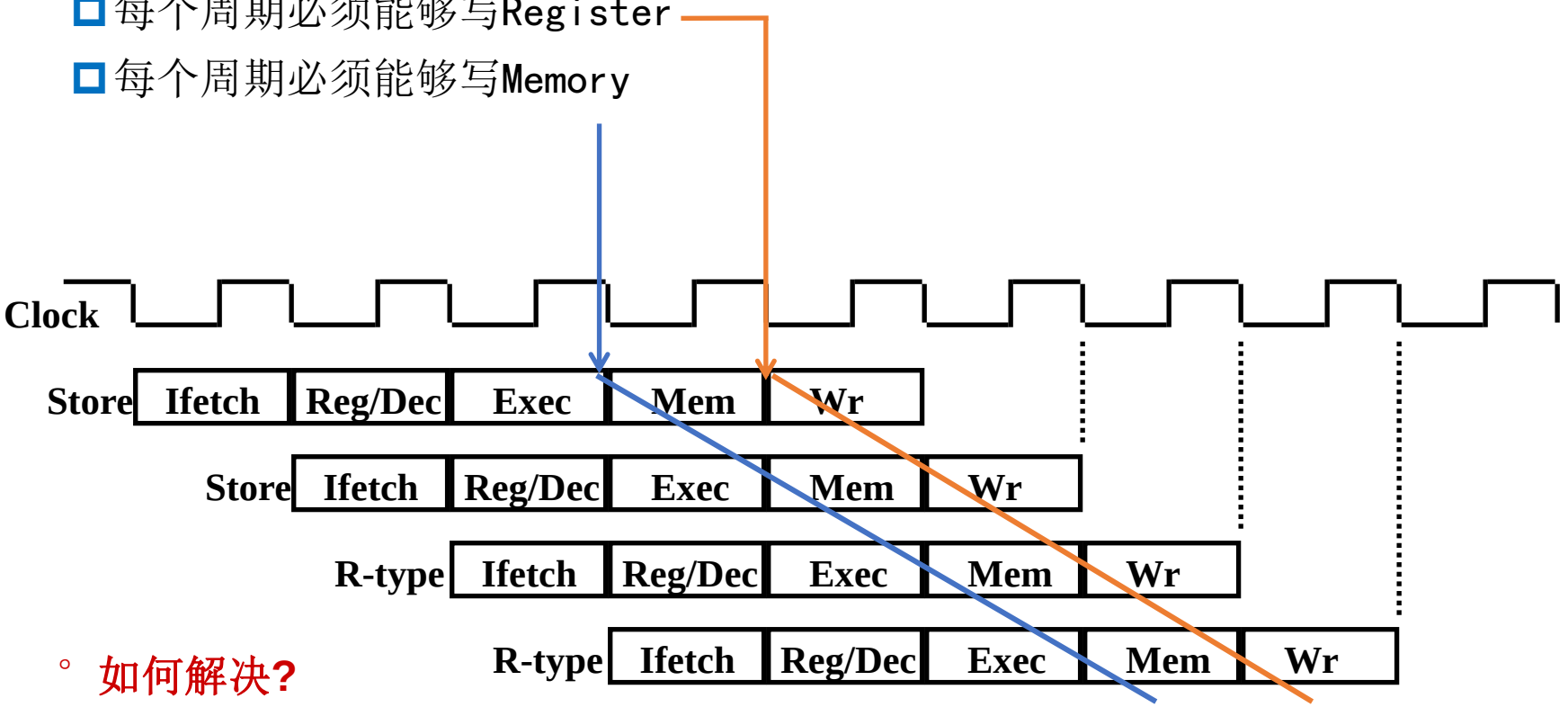
保存在流水段寄存器中的信息（包括前面阶段传递来或执行的结果及控制信号）一起被传递到下一个流水段！

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

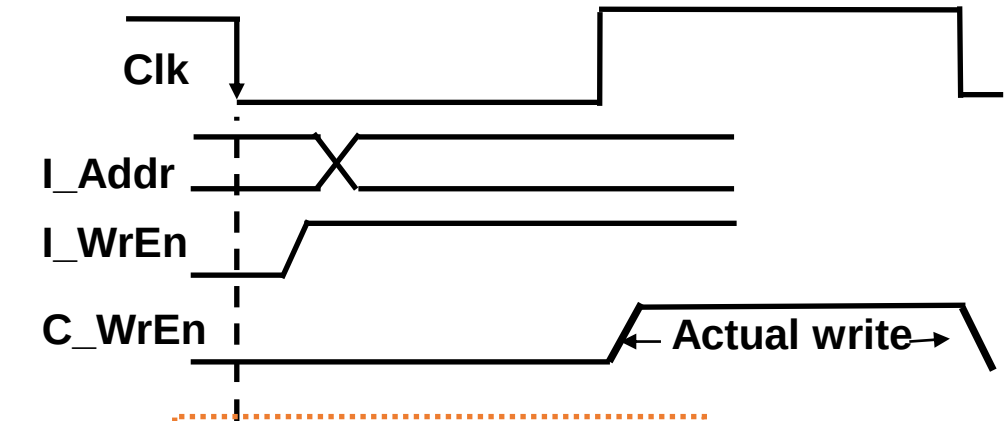


- 如何解决?

寄存器组的同步和存储器的同步

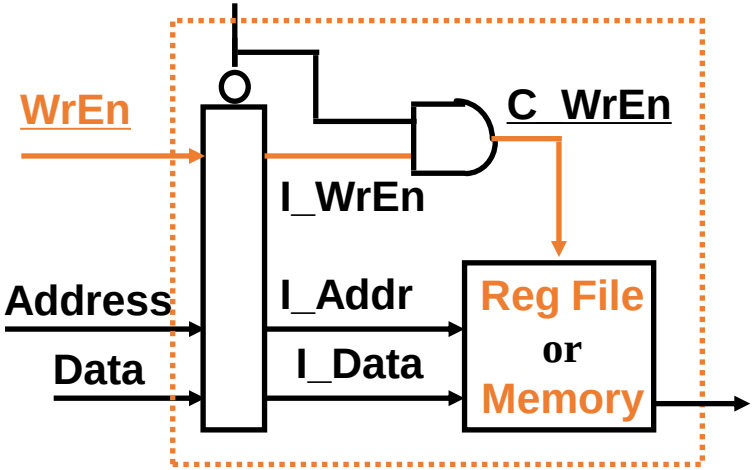
- 解决方案： 将Write Enable和时钟信号“与”

须由电路专家确保不会发生“定时错误”
（即：能合理设计“Clock”!）

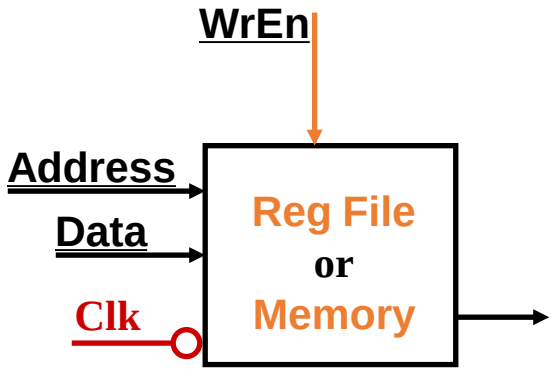


1. Addr, Data, 和 WrEn 必须在Clk边沿到来后至少稳定一个 set-up时间

2. Clk高电平时间 大于 写入时间

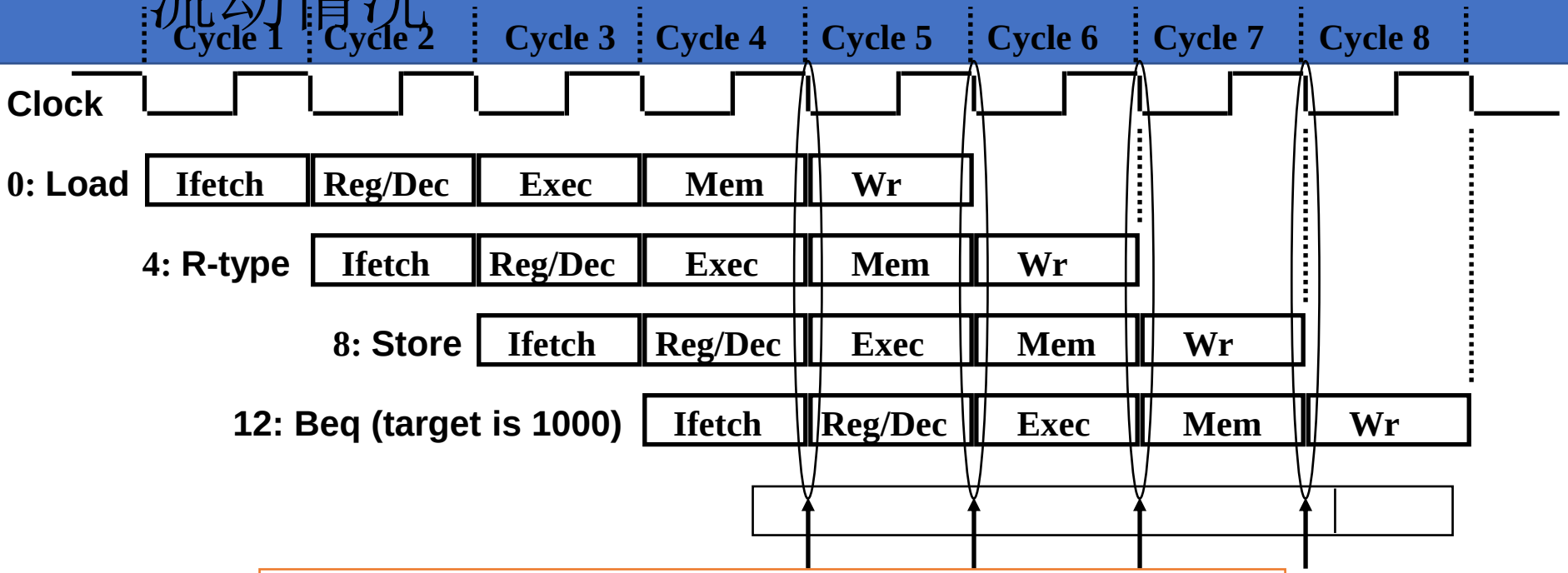


等价于



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



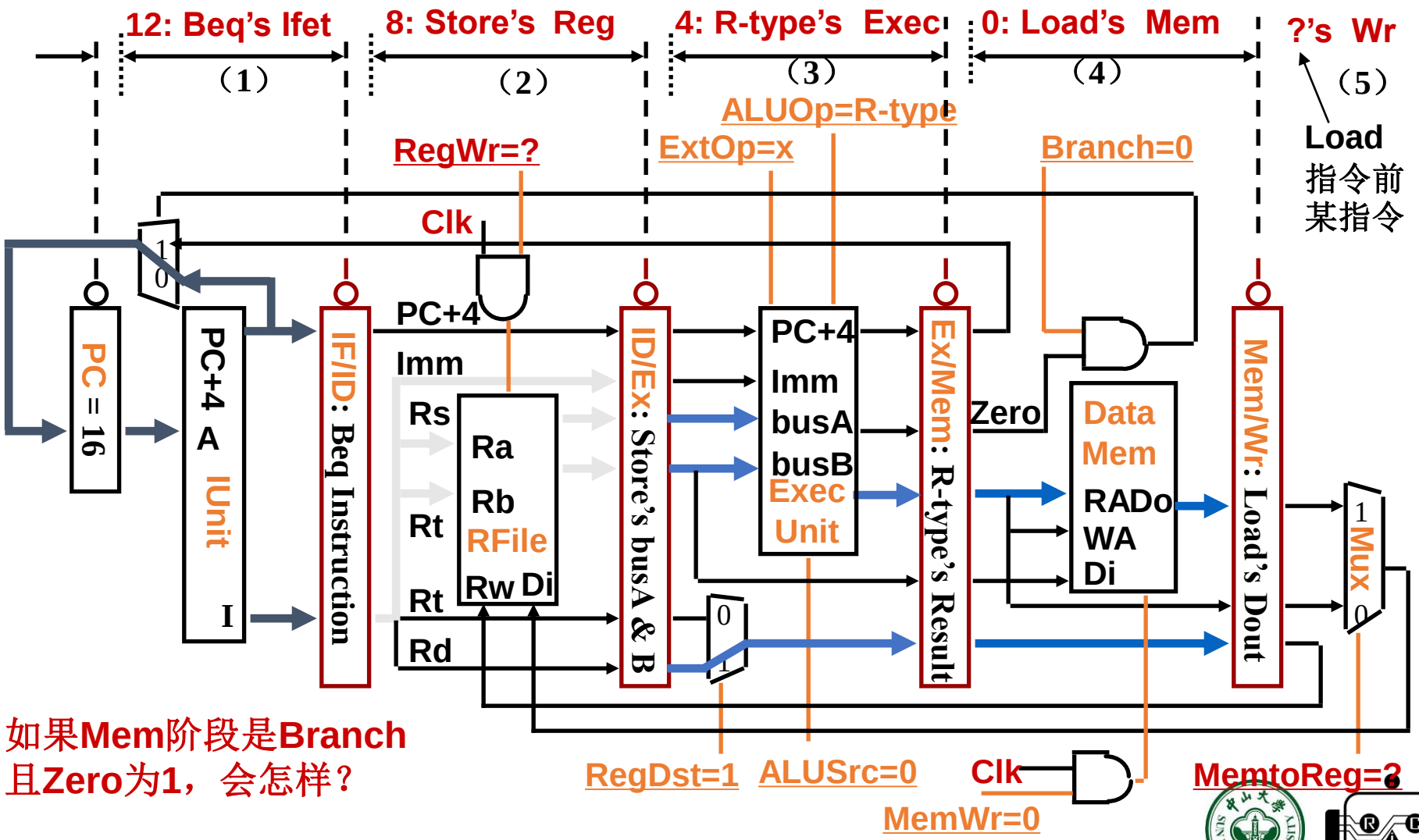
考察以下几个点的情况：End of Cycle 4　End of Cycle 5　End of Cycle 6　End of Cycle 7

- 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。

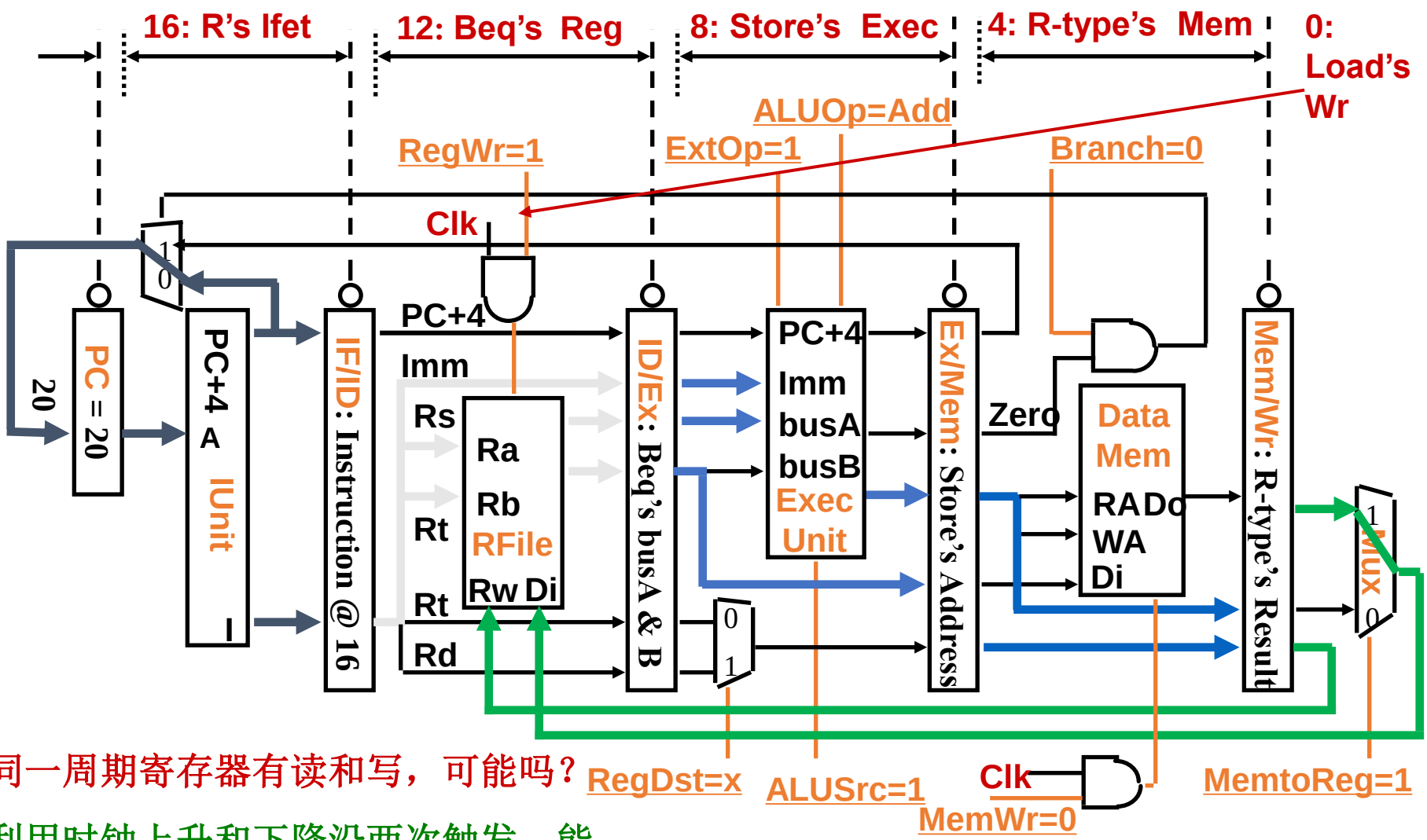
第四周期结束时的状态：

- 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



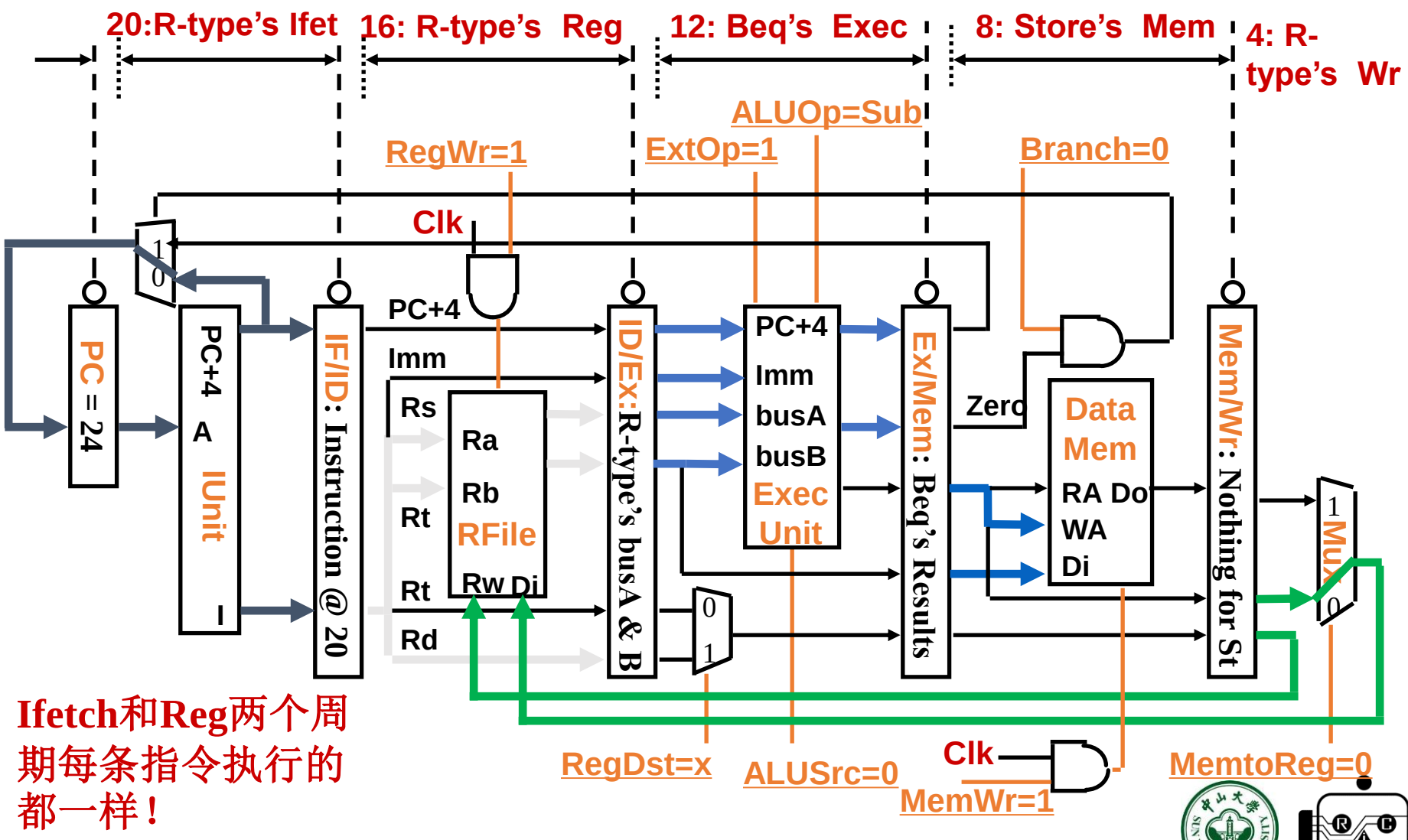
同一周期寄存器有读和写，可能吗？

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：

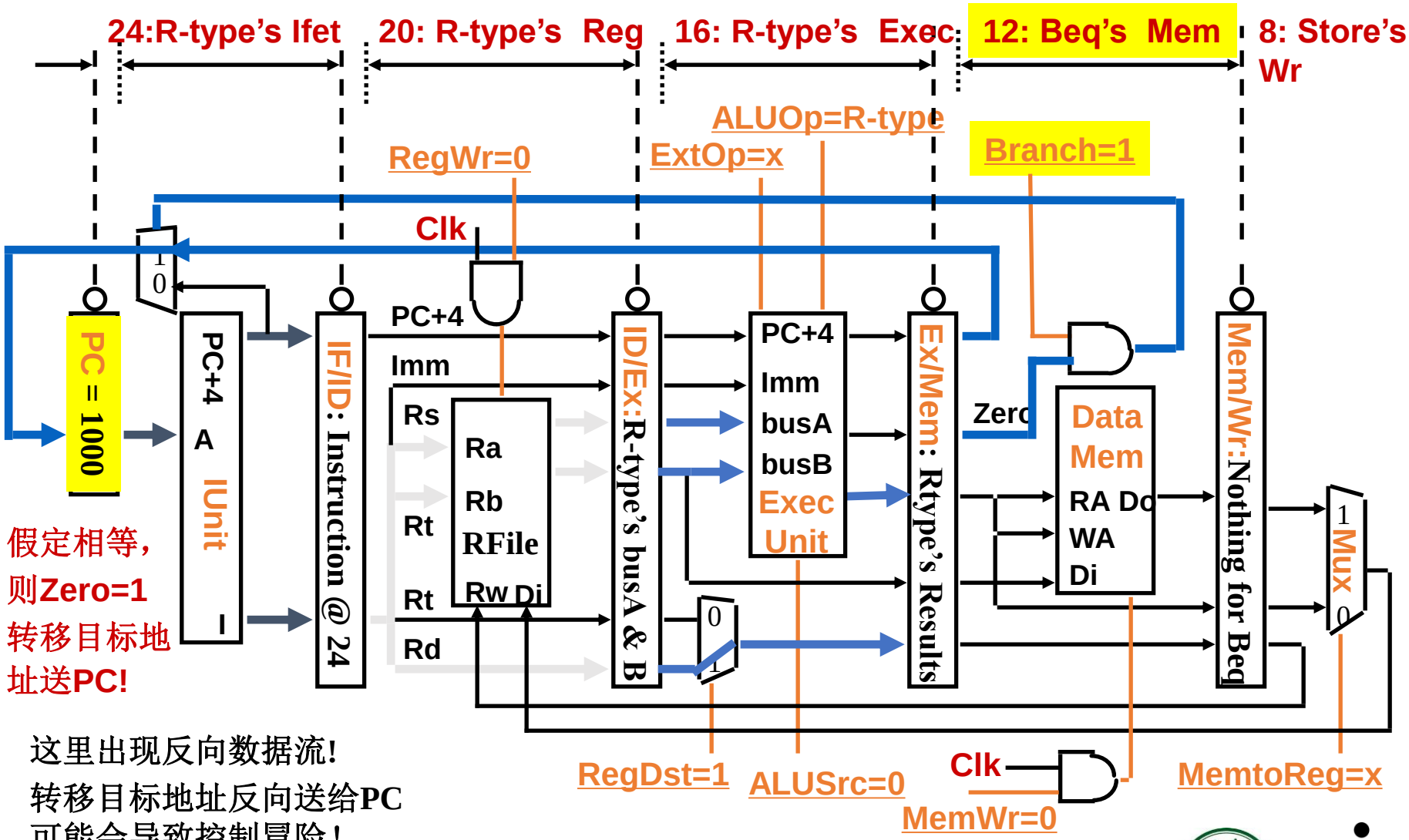
- 4: R’s Wr 8: Store’s Mem 12: Beq’s Exec 16: R’s Reg 20: R’s Ifet



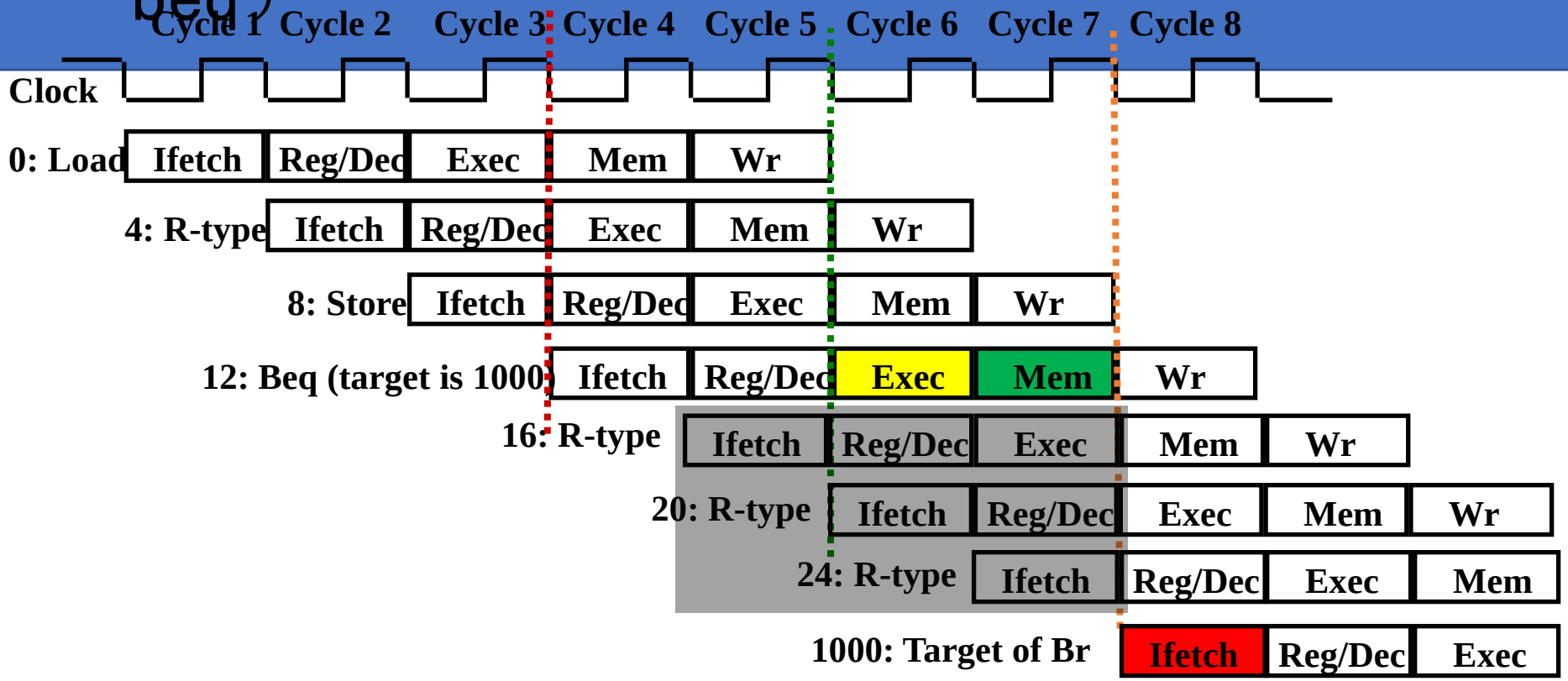
Ifetch和Reg两个周期每条指令执行的都一样！

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

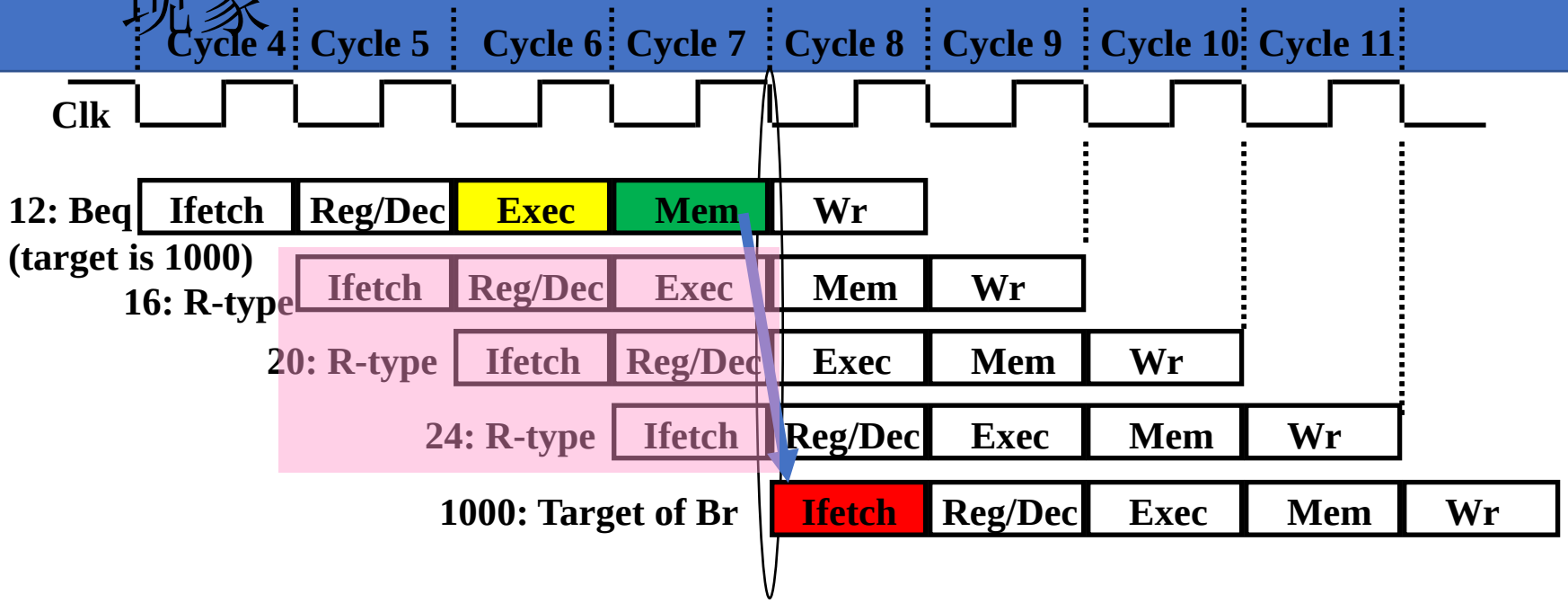


总结前面的流水线执行过程（有关beq）



- 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？

转移分支指令(Branch)引起的“延迟”现象

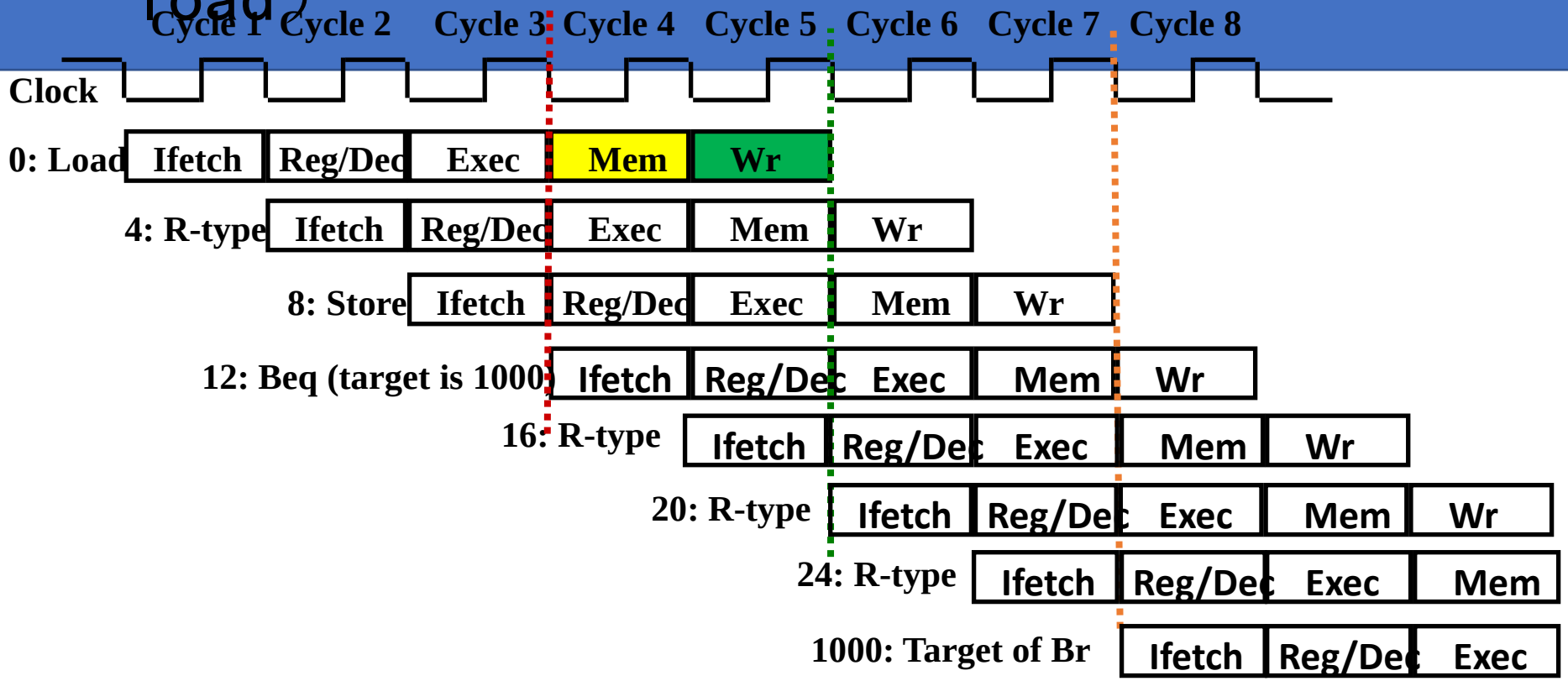


- 虽然Beq指令在第四周期取出，但:
  - 目标地址在第七周期才被送到PC的输入端
  - 第八周期才能取出目标地址处的指令执行

  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 ）

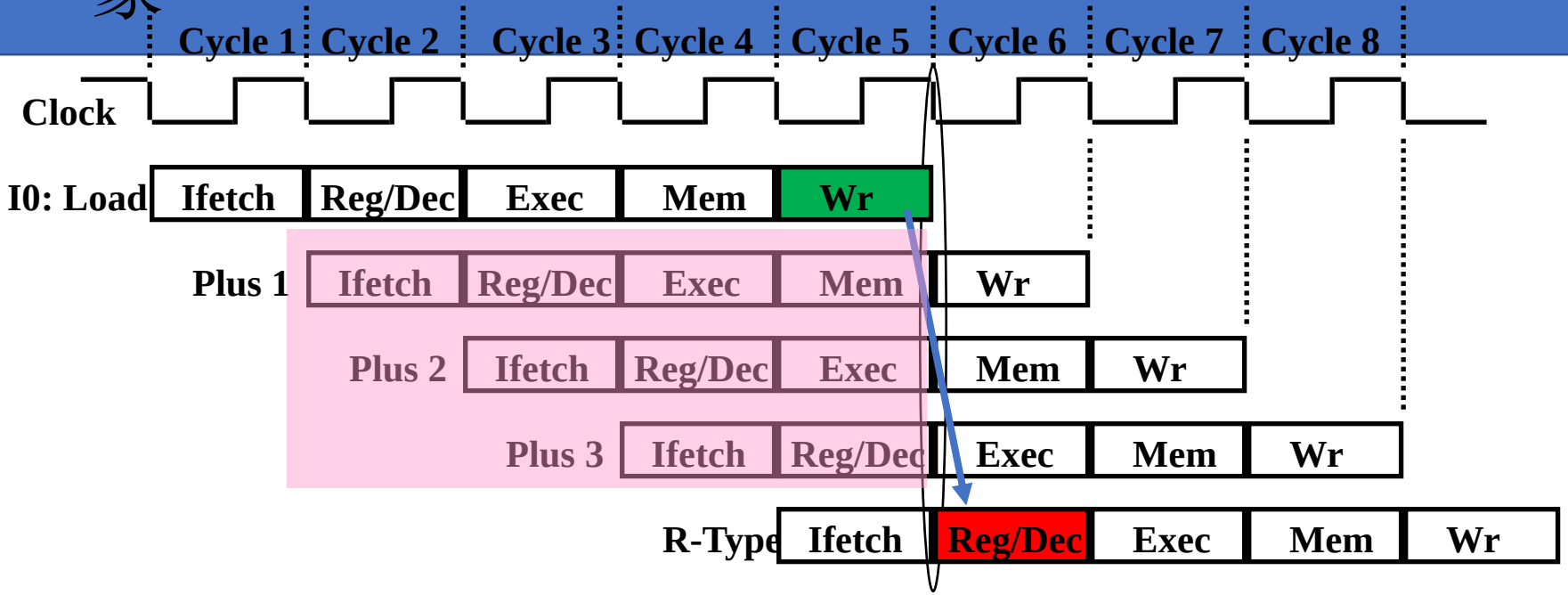
  （注：也称为分支冒险或转移冒险（Branch Hazard） ）

总结前面的流水线执行过程（有关Load）



- 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？

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

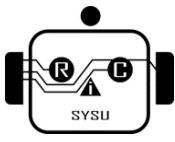
  结果：在Load指令结果有效前，已经有三条指令被取出

  (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，分为N个步骤，并用流水线方式进行
  - 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/Reg)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/Reg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）



联系方式

- Acknowledgements:
- This slides contains materials from following lectures:
  - Computer Architecture (ETH，NUDT，USTC)

- Research Area：
  - 计算机视觉与机器人应用计算加速，
  - 人工智能和深度学习芯片及智能计算机

- Contact：
  - 中山大学计算机学院
  - 管理学院D101（图书馆右侧）
  - 机器人与智能计算实验室
  - cheng83@mail.sysu.edu.cn