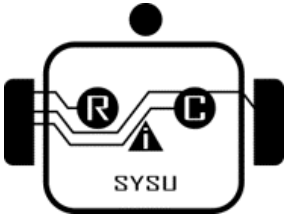
中山大學
SUN YAT-SEN UNIVERSITY



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

本节概要

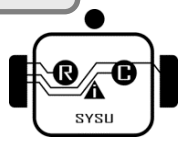
重点内容

第四章 处理器

- 多周期控制器的实现

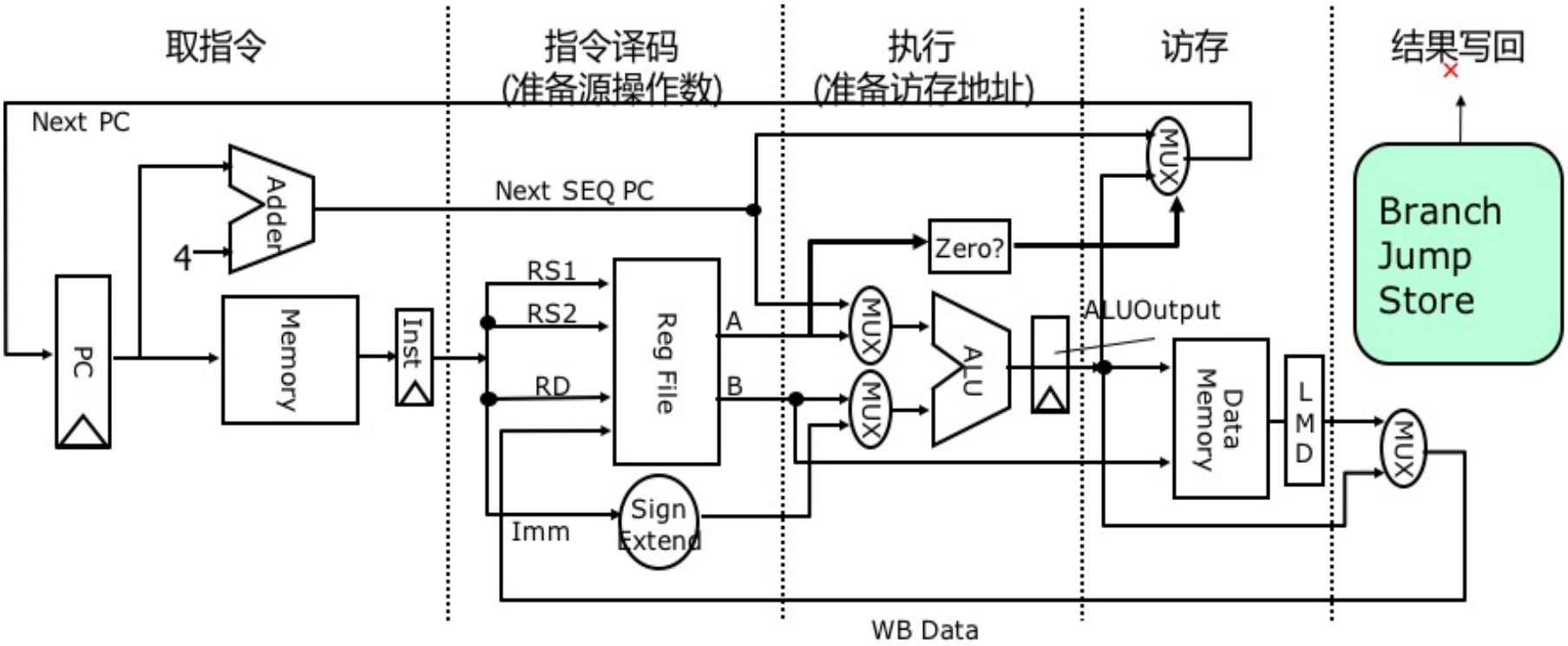
基本要求

- 理解多周期控制器的设计



1 多周期处理器的实现思想

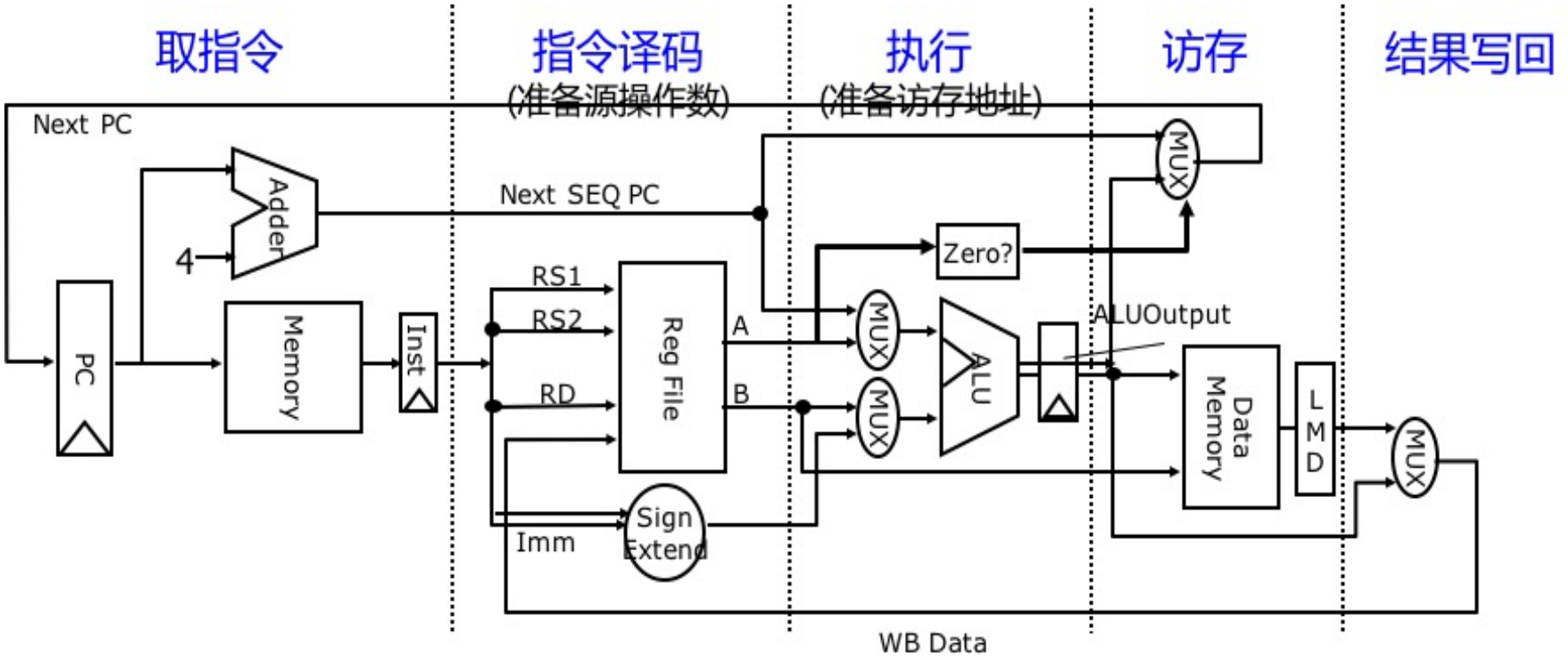
- 单周期处理器的问题根源
  - 时钟周期以最复杂指令所需时间为准，太长！



1 多周期处理器的实现思想

- 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！
- 解决思路

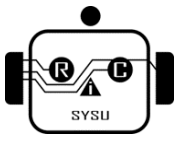
  a）把指令执行分成多个阶段，各阶段在一个时钟周期内完成



1 多周期处理器的实现思想

- 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！
- 解决思路

  a）把指令执行分成多个阶段，各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
  - 每个阶段内最多只能完成：1次访存 或 寄存器堆读/写 或 ALU运算



1 多周期处理器的实现思想

- 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！
- 解决思路

  a）把指令执行分成多个阶段，各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
  - 每个阶段内最多只能完成：1次访存 或寄存器堆读/写 或 ALU

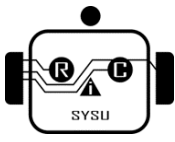
  b）每步都设置相应的存储元件，执行结果都保存到相应单元中

1 多周期处理器的实现思想

- 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
    - Load： 5 cycles
    - Jump：3 cycles
  - 允许功能部件在一条指令执行过程中被重复使用
    - Adder + ALU(多周期控制时只用一个ALU，在不同周期可重复使用)
    - 指令 / 数据存储器（多周期控制时只需一个存储器，不同周期重复使用）

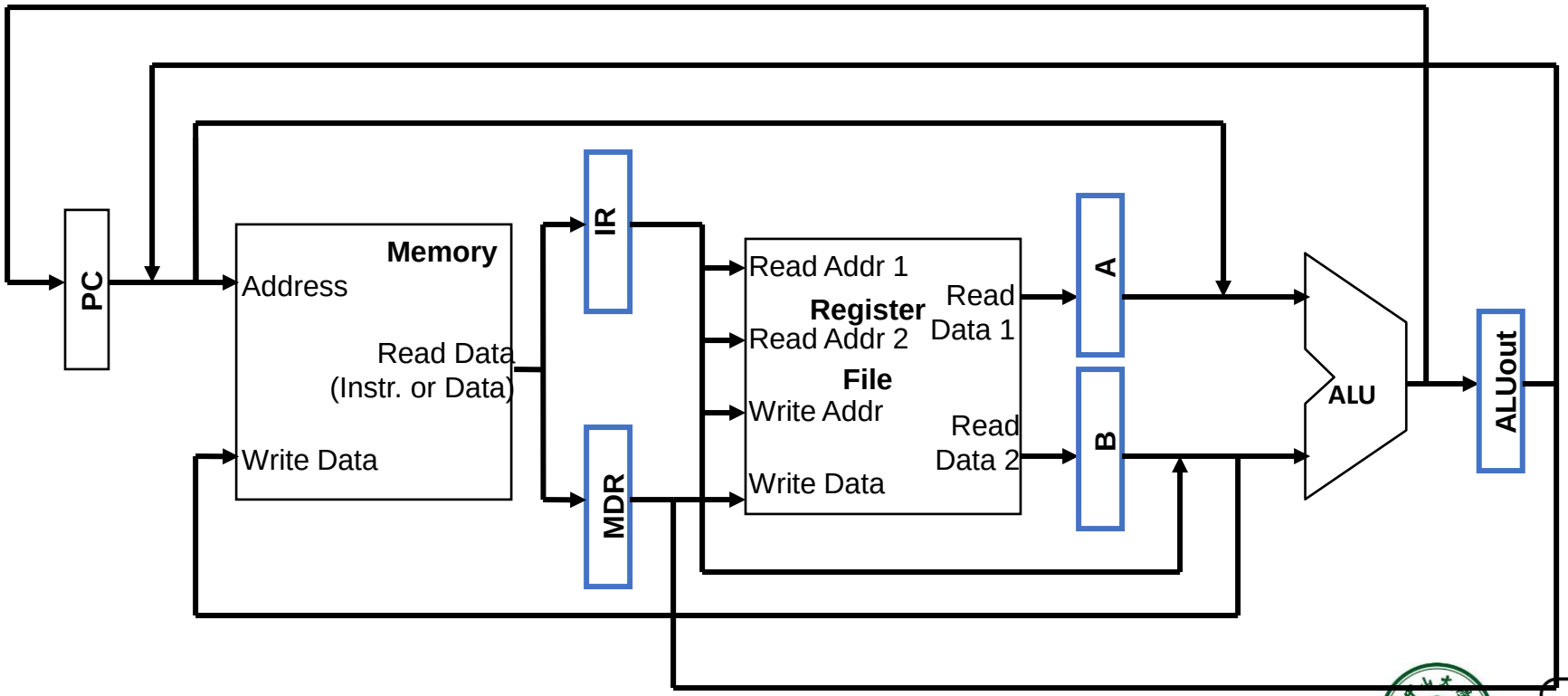
Multicycle Implementation Overview

- Each instruction **step** takes 1 clock cycle
  - Therefore, an instruction takes **more** than 1 clock cycle to complete
- Not every instruction takes the **same** number of clock cycles to complete
- Multicycle implementations allow
  - 更快的时钟频率
  - 不同的指令用不同的周期数
  - 单个ALU用于所有计算
  - 单个存储器用于指令和数据的存储
  - 需要增加寄存器暂存数据用于跨时钟周期

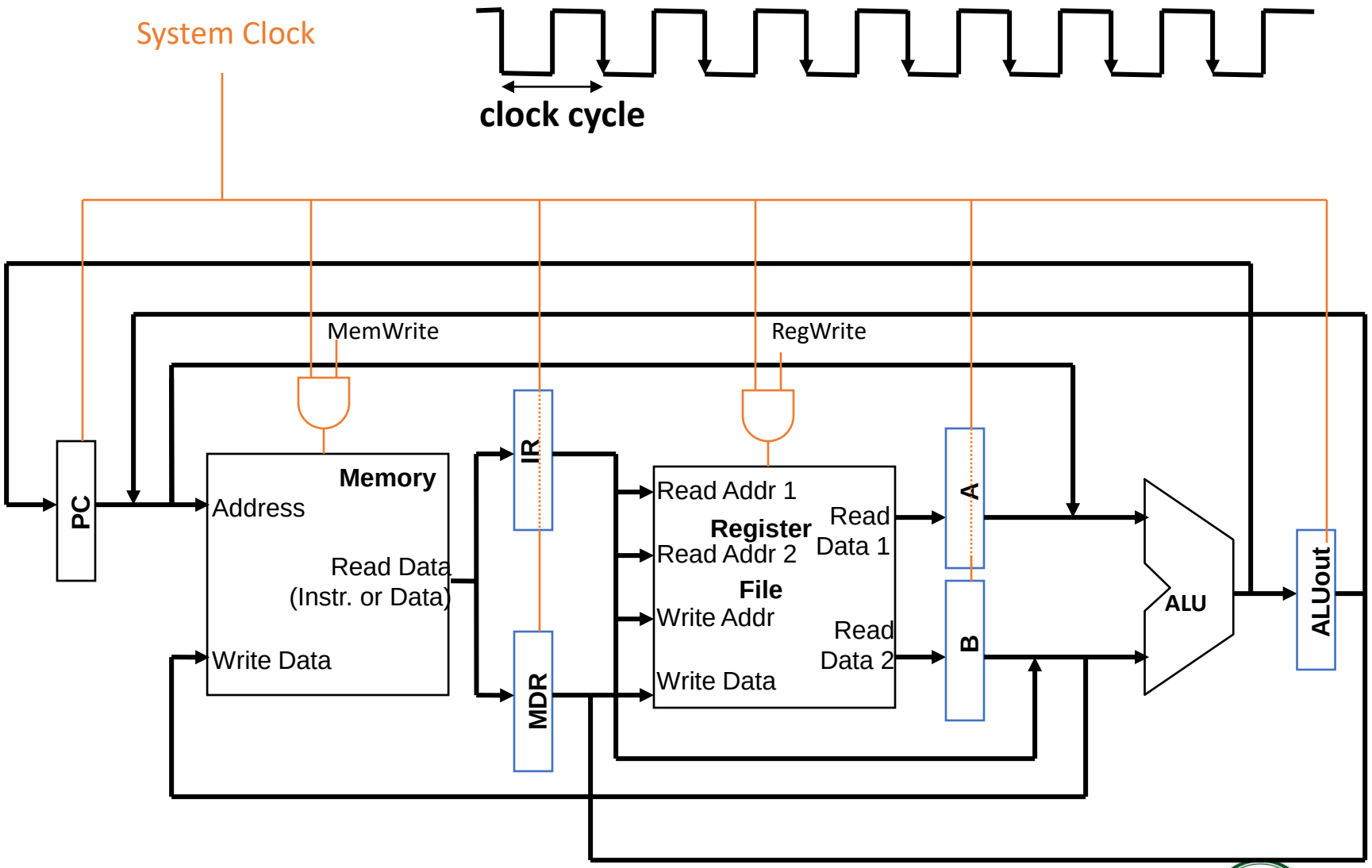


The Multicycle Datapath - A High Level View

- Registers have to be added after every major functional unit to hold the output value until it is used in a subsequent clock cycle



Clocking the Multicycle Datapath



Our Multicycle Approach

- Break up the instructions into steps where each step takes a clock cycle while trying to
  - balance the amount of work to be done in each step
  - use only one major functional unit per clock cycle
- At the end of a clock cycle
  - Store values needed in a later clock cycle by the current instruction in a state element (internal register not visible to the programmer)

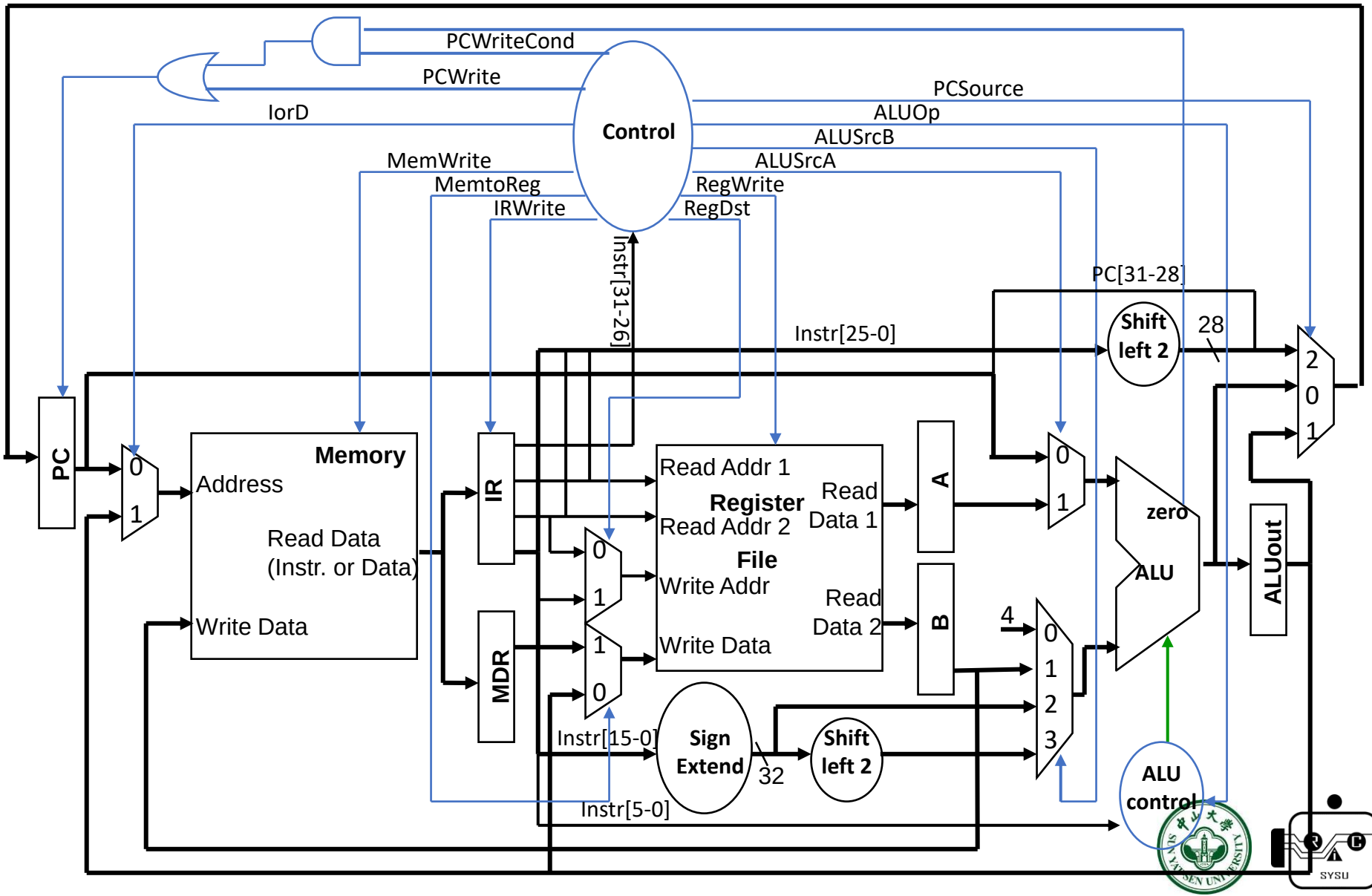
    IR – Instruction Register

    MDR – Memory Data Register

    A and B – Register File read data registers

    ALUout – ALU output register
    - All (except IR) hold data only between a pair of adjacent clock cycles (so they don't need a write control signal)
  - Data used by subsequent instructions are stored in programmer visible state elements (i.e., Register File, PC, or Memory)

The Complete Multicycle Data with Control



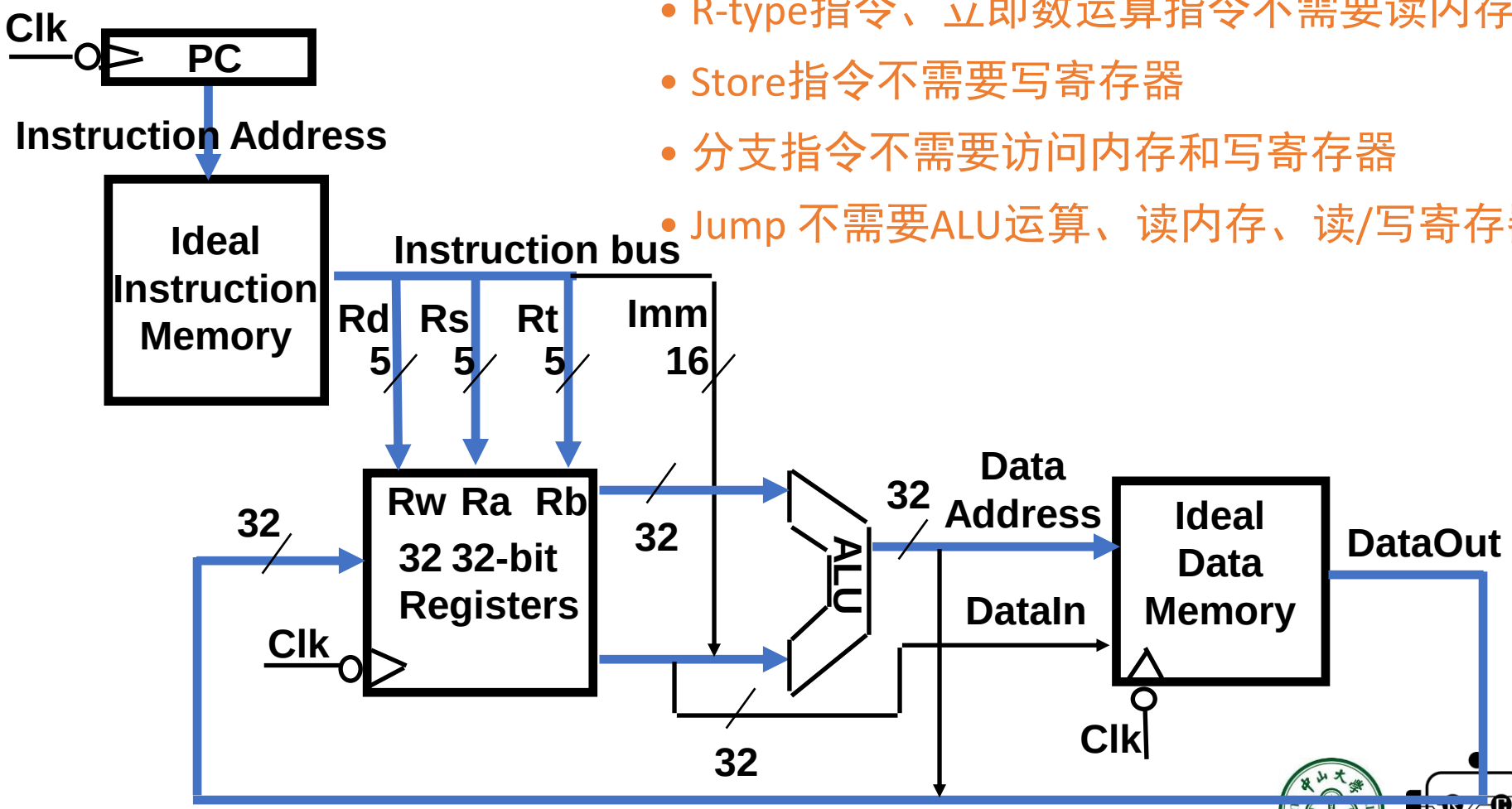
考察每条指令在单周期数据通路中的执行过程

- 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+4+signExt[imm16]*4**
    - **jump=1： PC<31:28> concat target<25:0> concat “00”**
  - 指令取出后被译码，产生指令对应的控制信号
    - R-type指令：rd为目的寄存器，无访存操作，……
    - ori指令：rt为目的寄存器，0扩展，无访存操作，……
    - lw指令： rt为目的寄存器，计算地址、符号扩展，读内存，……
    - sw指令： rt为源寄存器，计算地址、符号扩展，写内存，……
- 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，设计主控制逻辑和ALU局部控制逻辑

单周期数据通路的缺点：回顾 Load 指令执行

单周期处理器的CPI为1，时钟周期以最长的load指令为准，时钟周期远远大于其他指令实际所需的执行时间，效率极低

- R-type指令、立即数运算指令不需要读内存
- Store指令不需要写寄存器
- 分支指令不需要访问内存和写寄存器
- Jump 不需要ALU运算、读内存、读/写寄存器

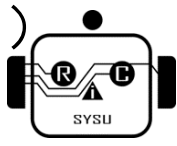


多周期处理器的实现思想

- 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
  - 每步都设置存储元件，每步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，可参考：

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

  - 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）

指令各阶段分析

- 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
- 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”分支地址
- 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
  - 如果是分支branch指令，该阶段需决定是否将分支地址写入PC
- 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

第一个周期（取值周期）

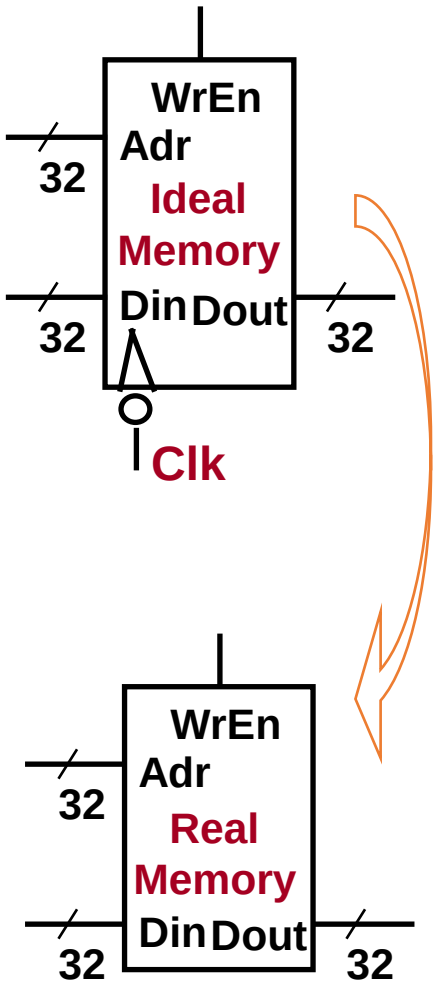
第二个周期（译码取数周期）

第三、四、五个周期（各指令不同）

寄存器堆和存储器的写定时（ Ideal vs. Reality）

- 单周期机器中，寄存器组和存储器被简化为理想的：
  - 时钟边沿到来之前，地址、数据和写使能都已经稳定
  - 时钟边沿到来时，才进行写

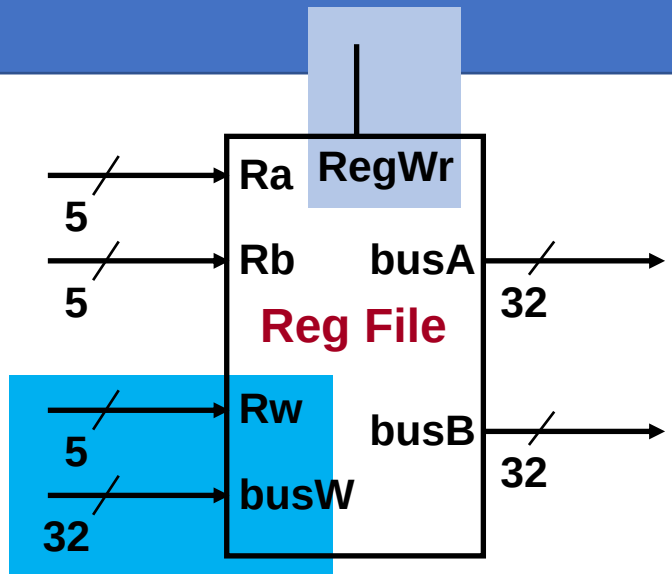
- 实际机器中，寄存器组和存储器的情况为：
  - 写操作不是由时钟边沿触发，是组合电路，其过程为：
    - 写使能(WE)为 1，并且 Din信号已稳定的前提下，经过Write Access时间，Din信号被写入 Adr 处
  - 重要之处： 地址和数据必须在写使能为1前稳定



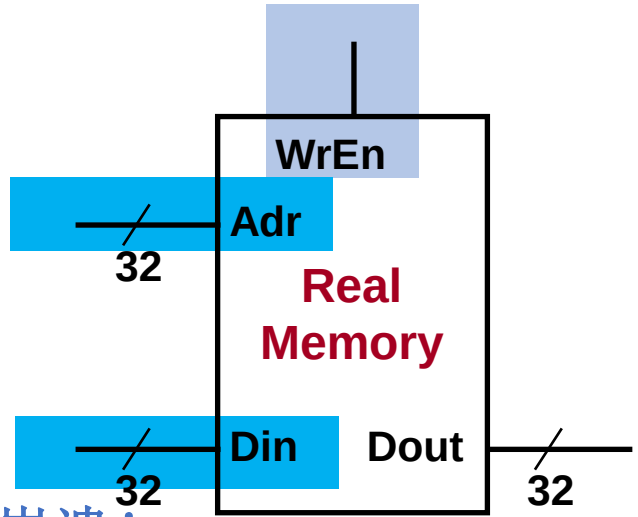
因此，存在地址Adr、数据Din和写使能WrEn信号的“竞争”问题！

竞争（race）问题

- Register File(寄存器组)：
  - 不能保证 Rw 和busW在RegWr = 1之前稳定
    即在 Rw和busW 与RegWr 之间存在“race”



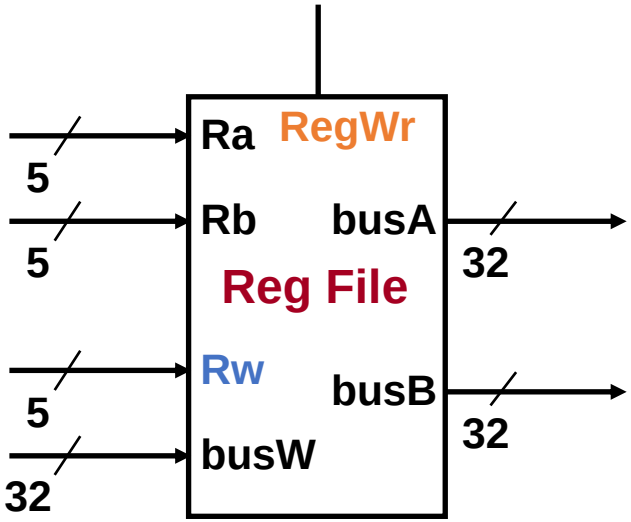
- Memory(存储器)：
  - 不能保证 Adr和Din在WrEn = 1之前稳定
    即：在 Adr 和Din与 WrEn之间存在“race”

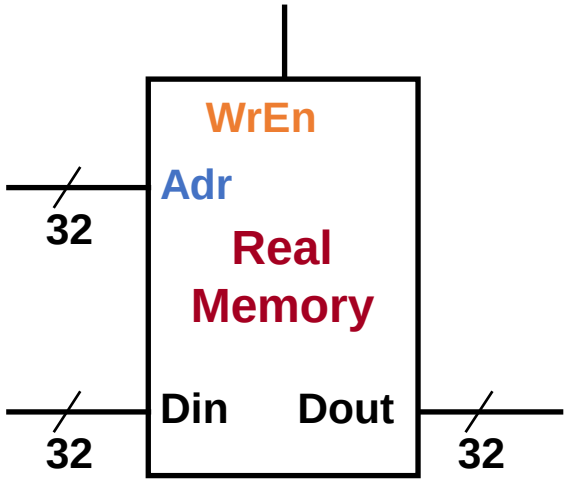


“Race”问题有时会导致机器神秘出错，甚至崩溃！

如何在多周期通路中避免“race”问题

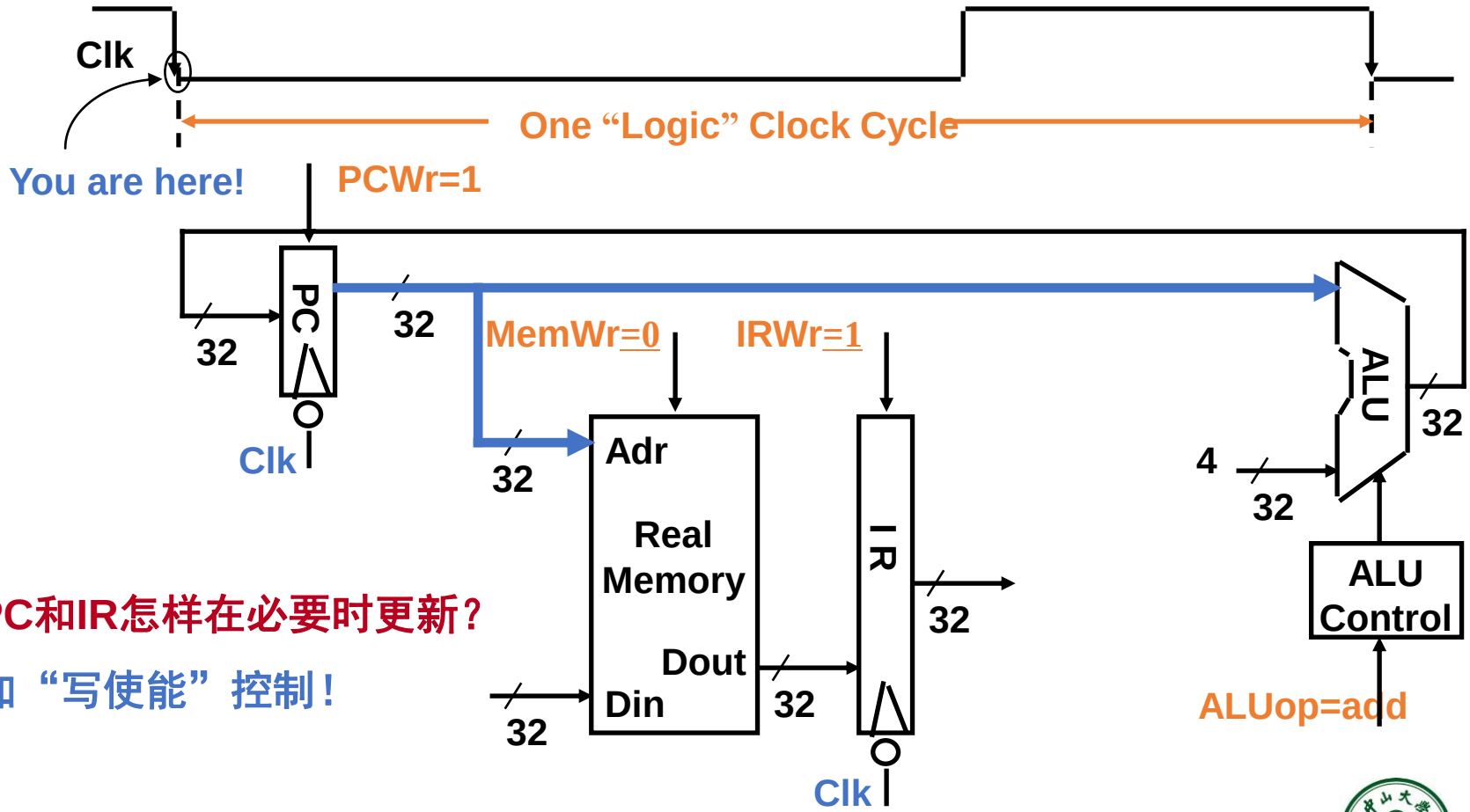
- 多时钟周期中解决“竞争”问题的方案
  - 确认地址和数据在第N周期结束时已稳定
  - 使写使能信号在一个周期后(即：第N+1周期)有效
  - 在写使能信号无效前地址和数据不改变





取指周期（取指令、计算下地址）开始时

- 在一个时钟到来的下降沿开始取指令周期的任务：
  - M[PC] ； PC←PC + 4

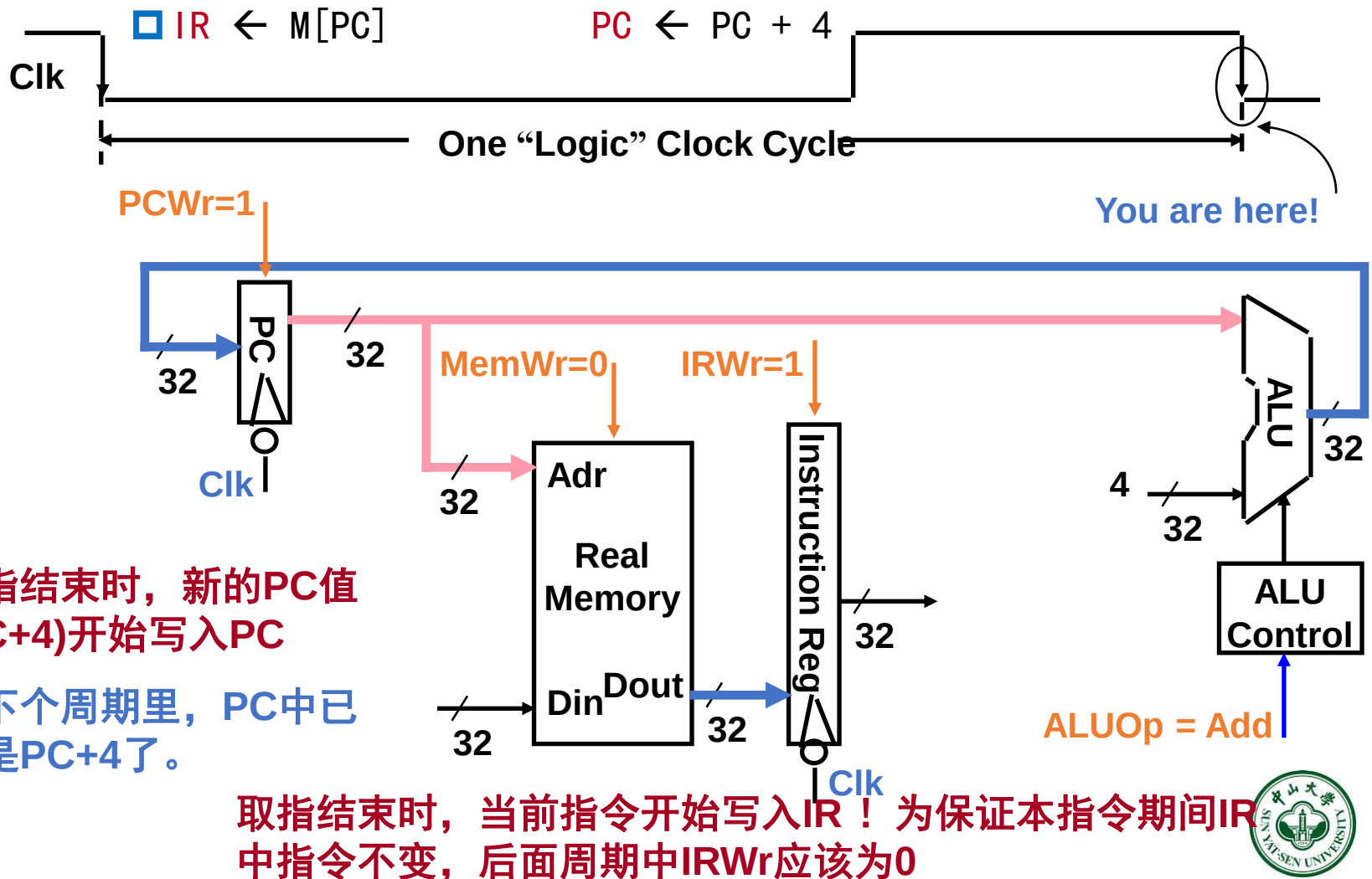


PC和IR怎样在必要时更新？

加“写使能”控制！

取指周期结束时

- 每一个周期都在下一个时钟到来时结束（此时，受时钟控制的存储元件被更新。不受时钟约束的存储原件则没有此特点）：



取指结束时，新的PC值(PC+4)开始写入PC

即下个周期里，PC中已经是PC+4了。

取指结束时，当前指令开始写入IR！为保证本指令期间IR中指令不变，后面周期中IRWr应该为0

考察整个取指周期（第一个周期）

分析：取指周期中各控制信号的取值应为？

想想看，和单周期有哪些不同？

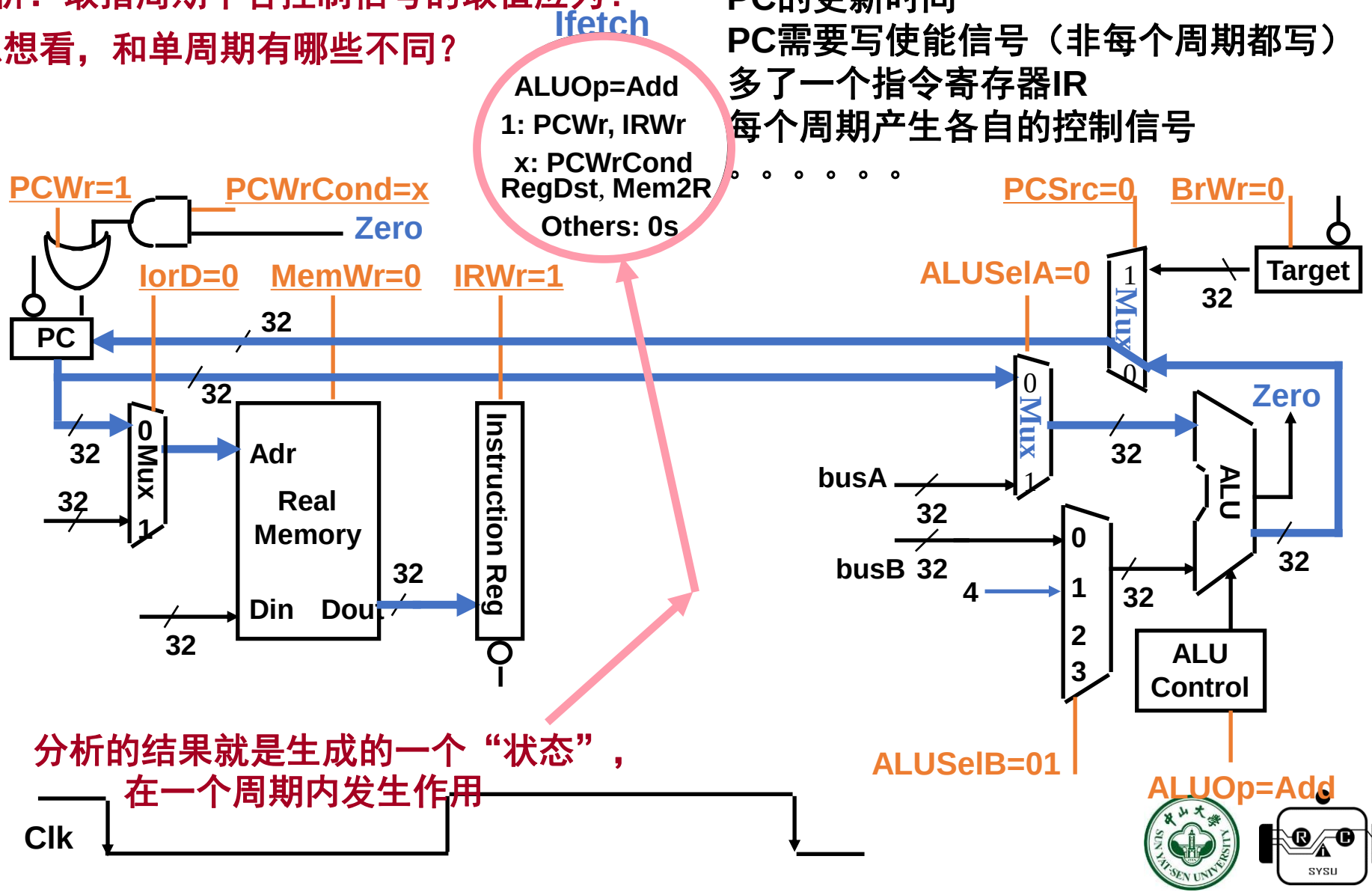
PC的更新时间

PC需要写使能信号（非每个周期都写）

多了一个指令寄存器IR

每个周期产生各自的控制信号

。。。。。。



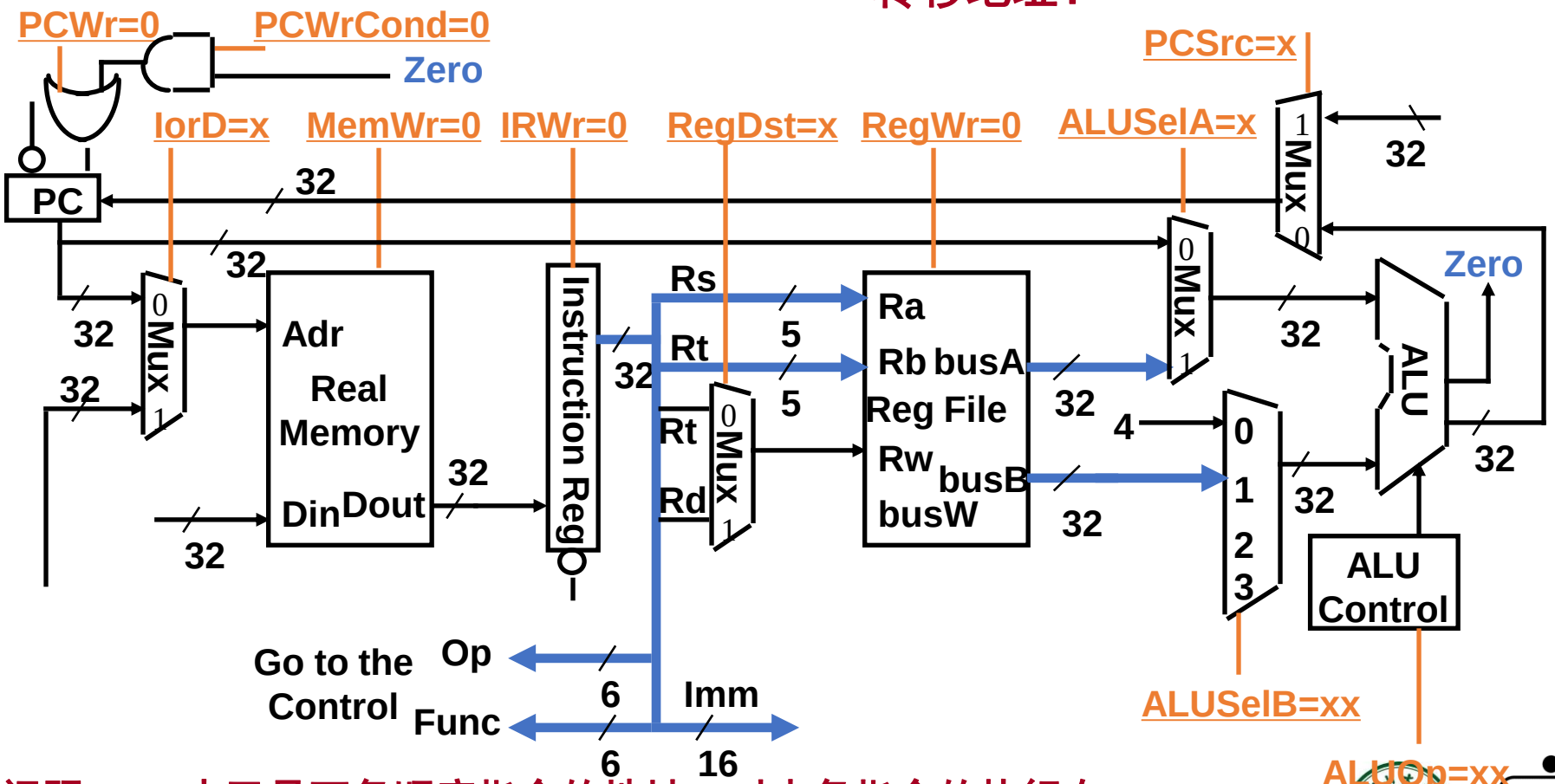
分析的结果就是生成的一个“状态”，在一个周期内发生作用

寄存器取 / 指令译码周期（第二个周期）

- busA ← RegFile[rs] ; busB ← RegFile[rt] ;
- Decoder ← Op and Func;
- ALU 本周期内暂未使用: ALUctr = xx

指令未译码，故只执行公共操作

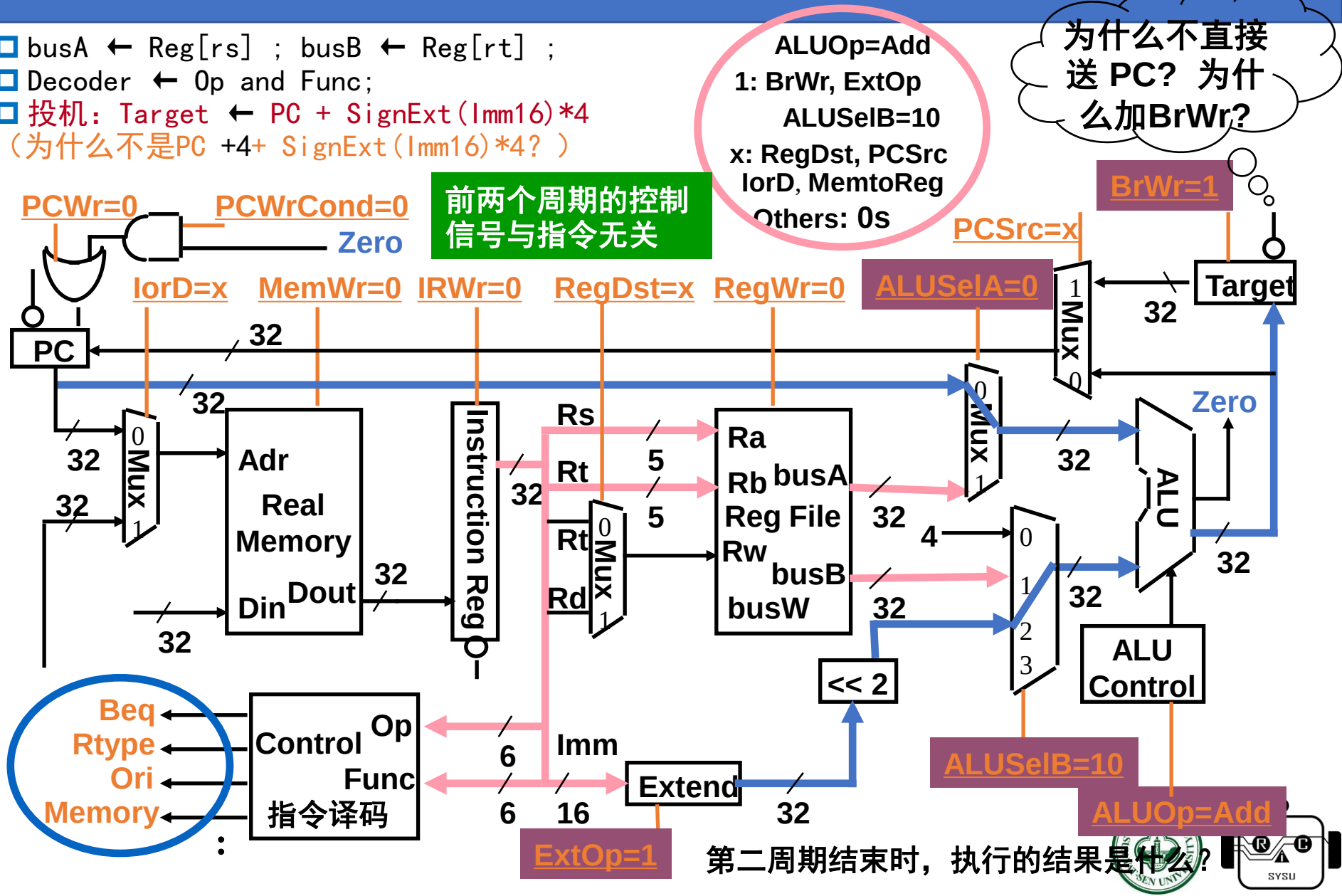
ALU空闲，可用ALU“投机计算”转移地址！



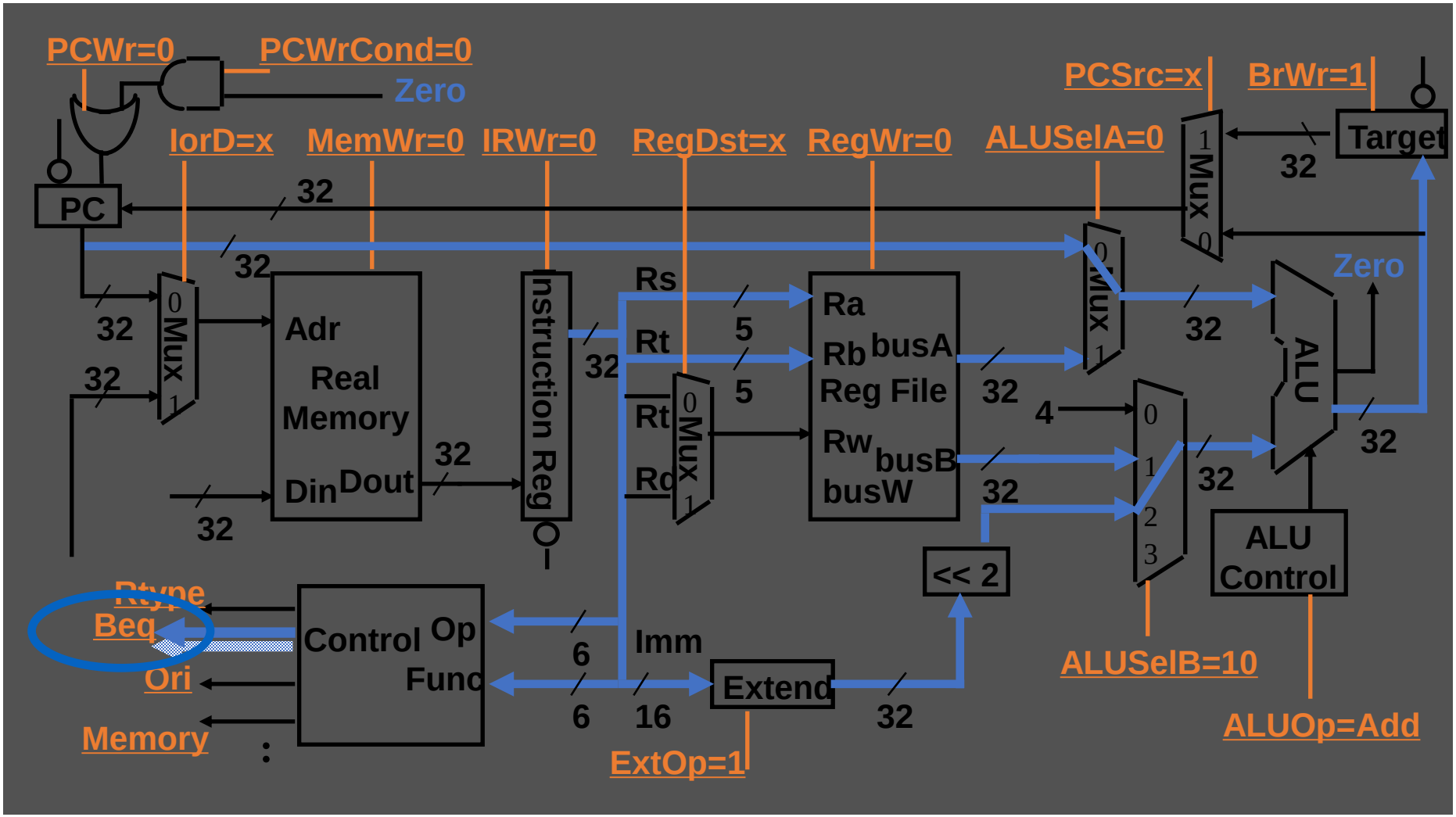
问题：PC中已是下条顺序指令的地址，对本条指令的执行有没有影响？没有影响，因为IRWr=0! 考虑转移地址投机计算的数据通路如何实现？

寄存器取 / 指令译码周期（第二个周期）

- busA ← Reg[rs] ; busB ← Reg[rt] ;
- Decoder ← Op and Func;
- 投机：Target ← PC + SignExt(Imm16)*4
  （为什么不是PC +4+ SignExt(Imm16)*4？）



寄存器取 / 指令译码周期（第二个周期）



如果指令译码输出为：Beq

下面第三个周期就是Beq指令的第一个执行周期！

Branch指令执行并完成周期（第三个周期）

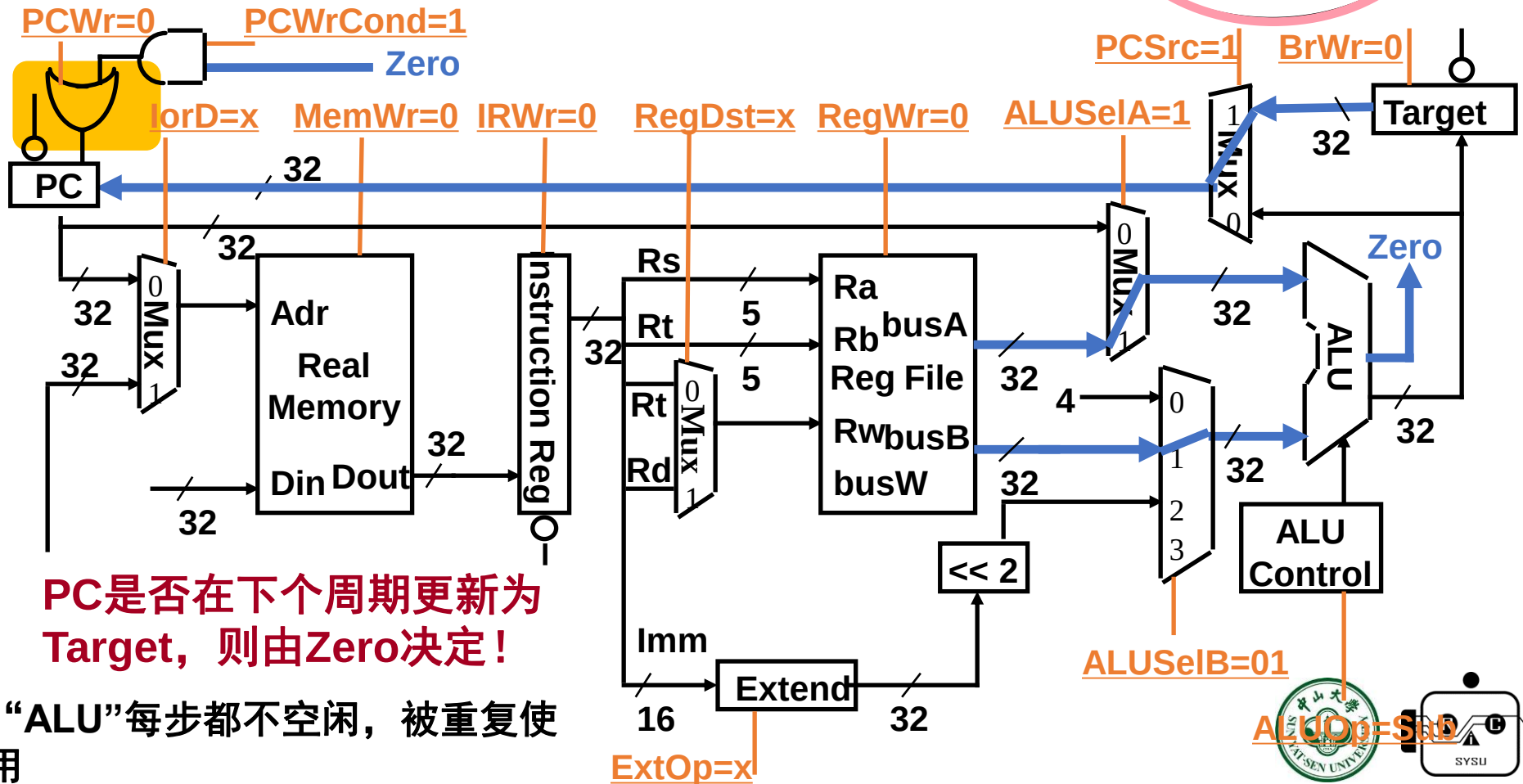
如果指令译码输出为：Branch

- if (busA == busB)
  - PC ← Target

若不“投机”，则在此周期前还要加一个周期，用来计算转移地址后保存到Target中！

控制信号的取值是什么？

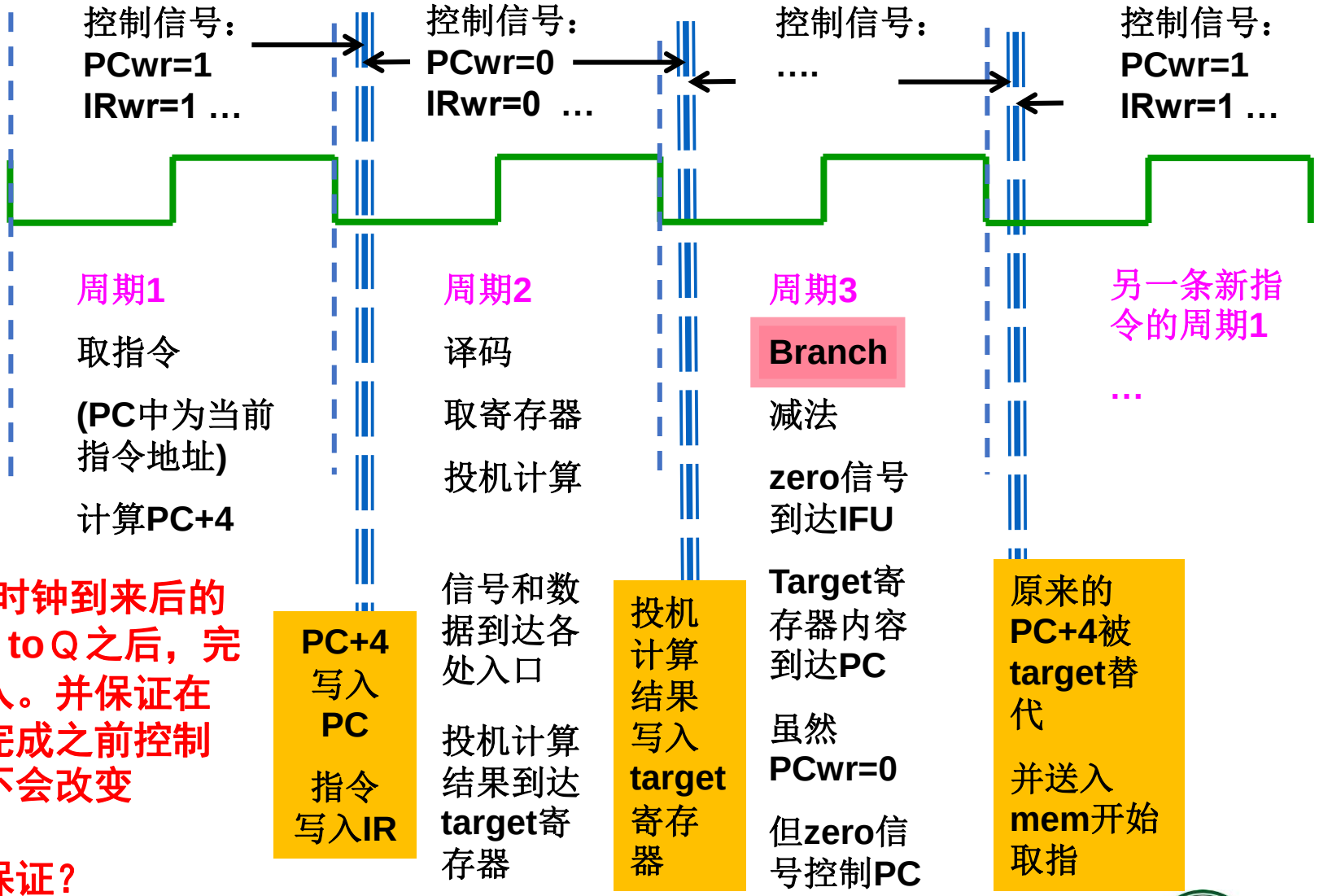
ALUOp=Sub
ALUSelB=01
x: IorD, Mem2Reg
RegDst, ExtOp
1: PCWrCond
PCSrc
ALUSelA



PC是否在下个周期更新为Target，则由Zero决定！

“ALU”每步都不空闲，被重复使用

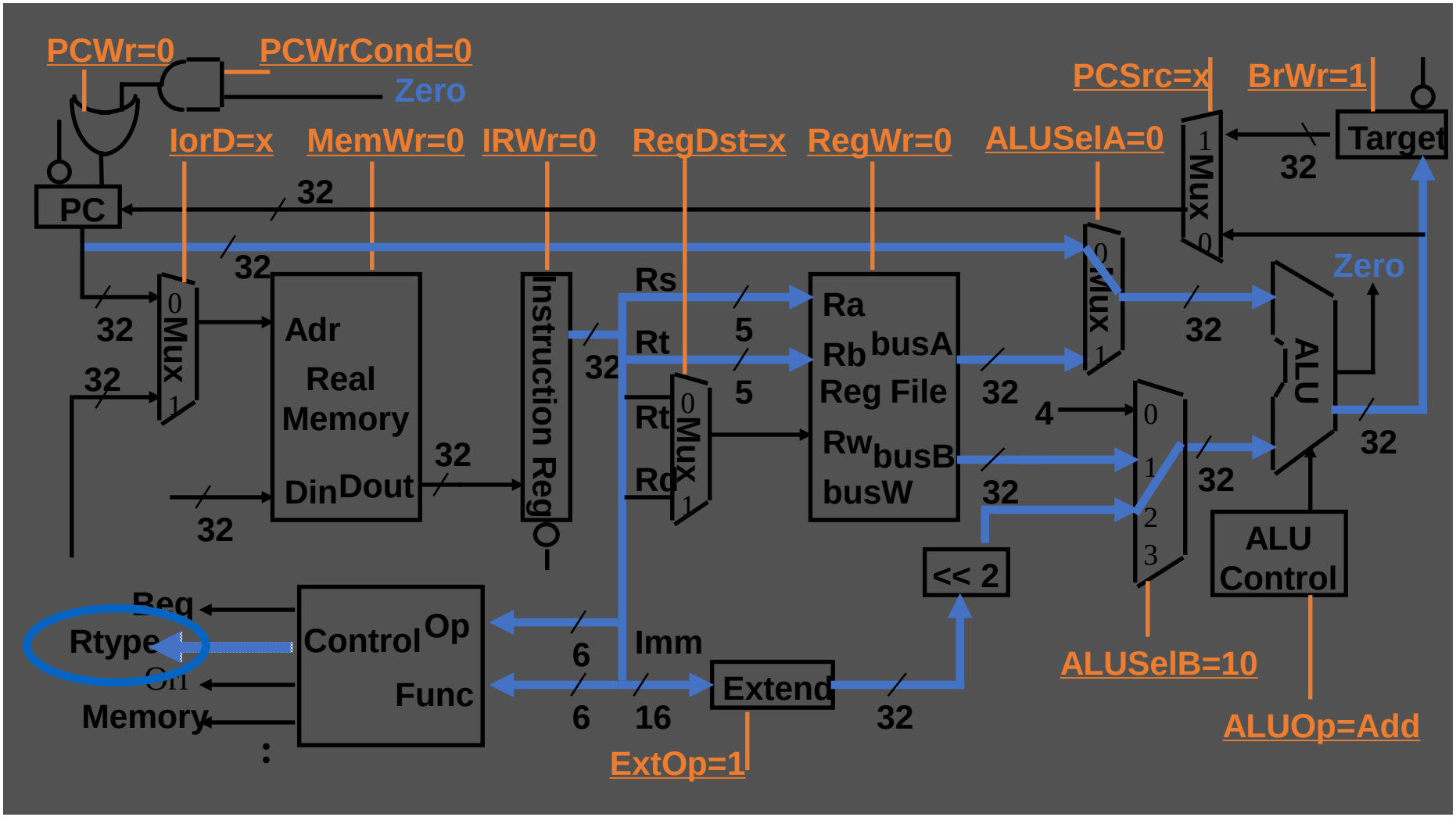
状态元件内容和控制信号取值的改变：在时钟到来后的一定延时后发生



PC在时钟到来后的clock to Q之后，完成写入。并保证在写入完成之前控制信号不会改变

如何保证？看后面的控制器设计！

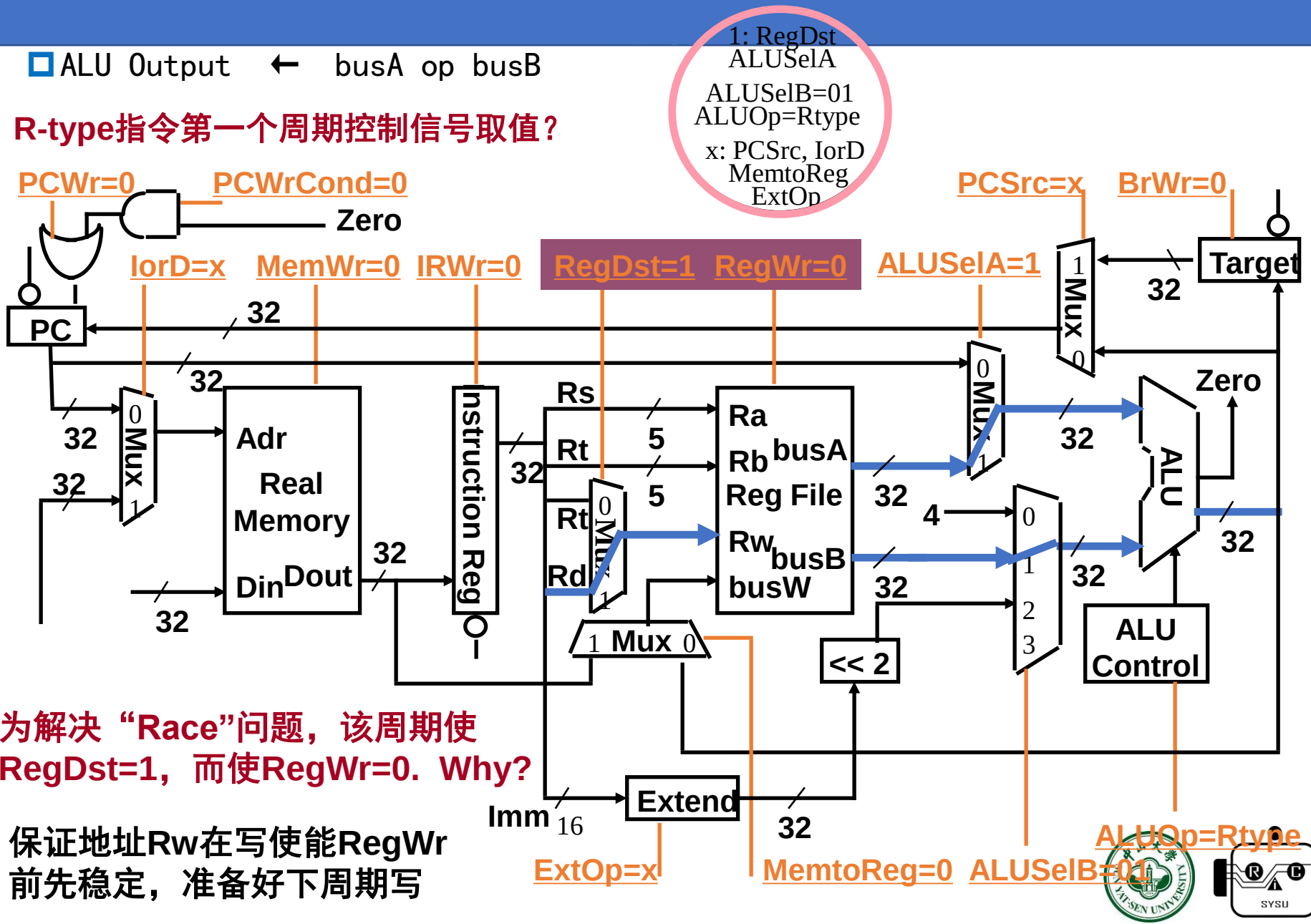
寄存器取 / 指令译码周期（第二个周期）



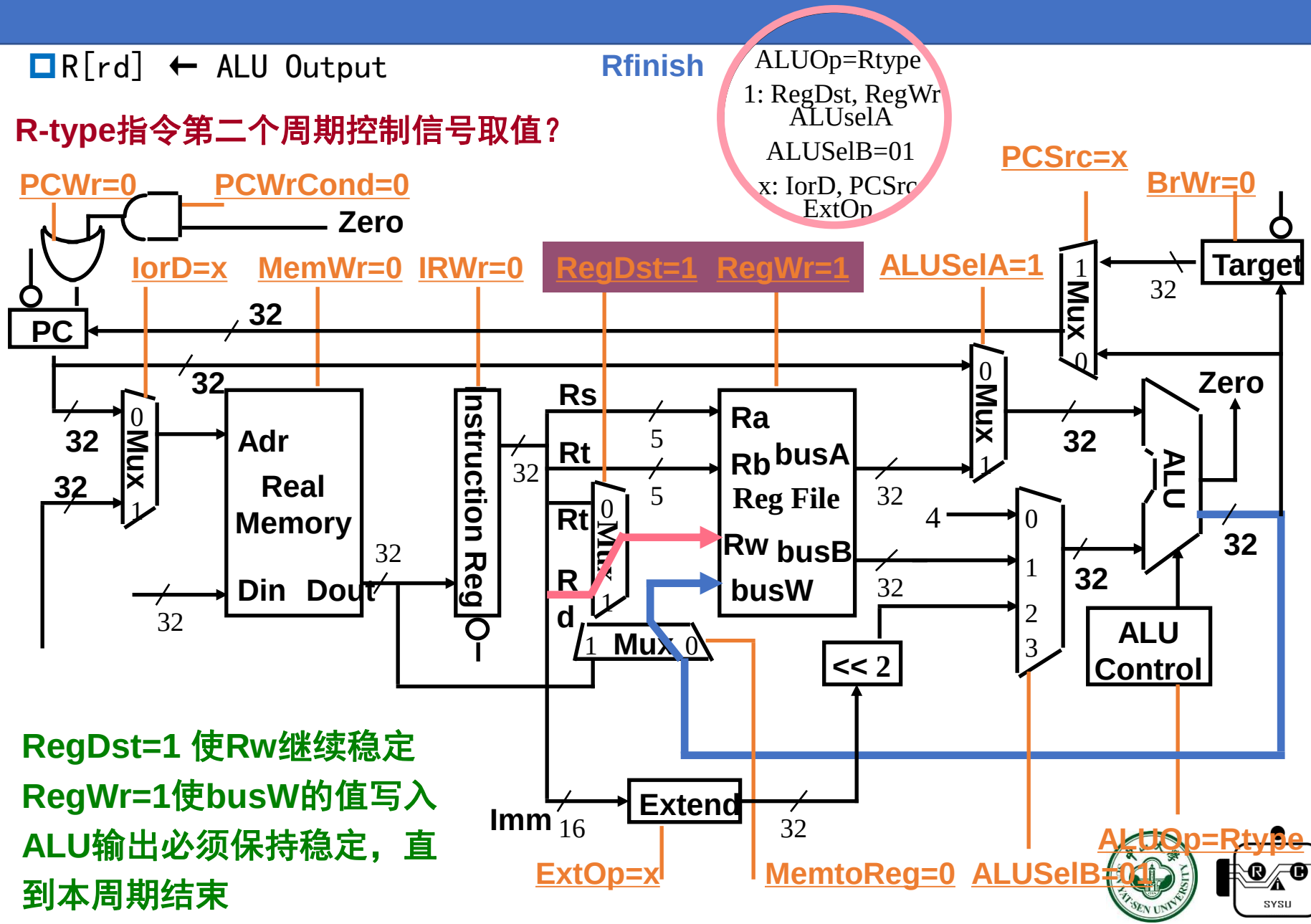
如果指令译码输出为：R-Type

第三个周期就是R-Type指令的第一个执行周期！

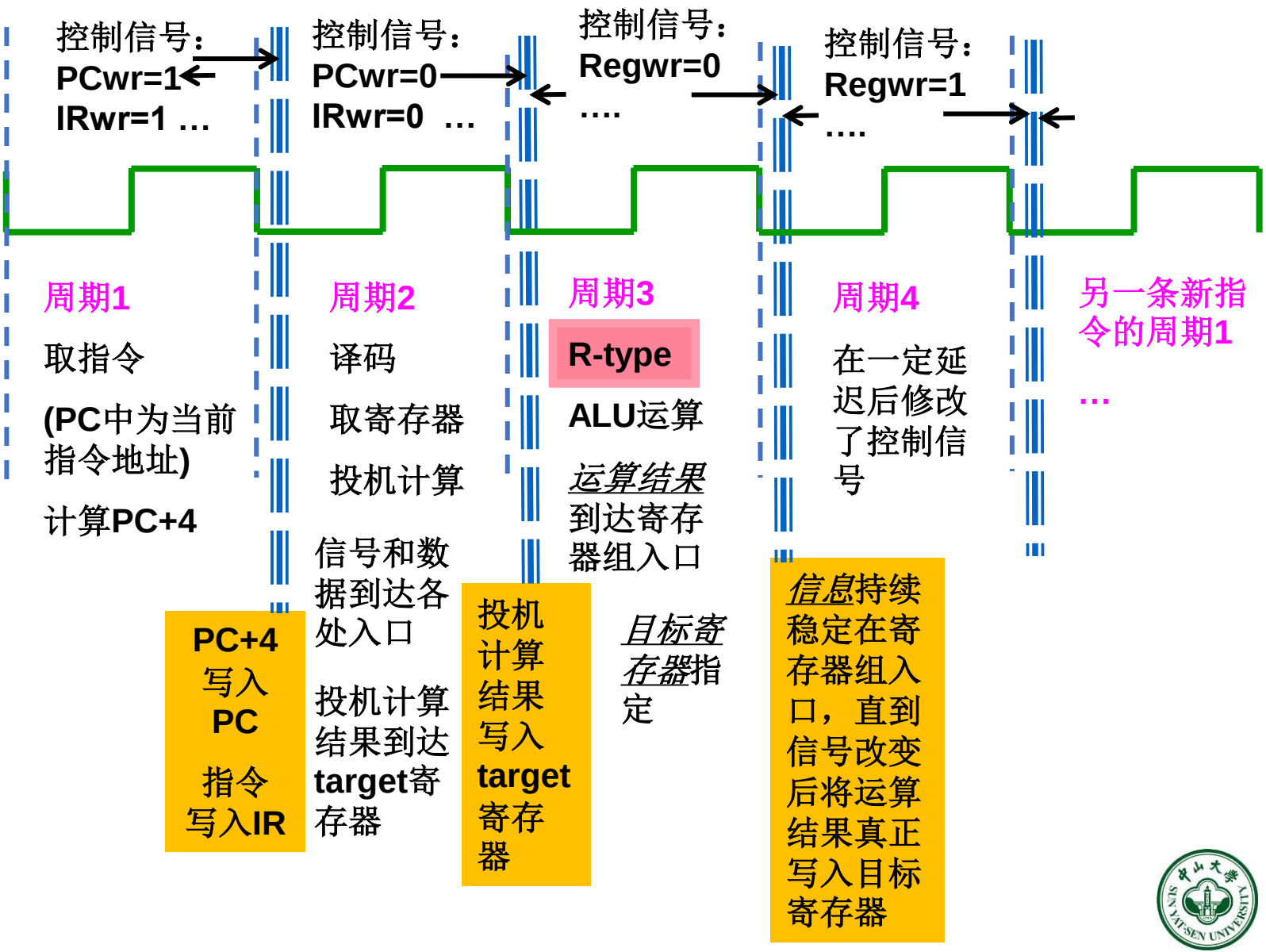
R-type指令的执行周期（第三个周期，ALU运算）



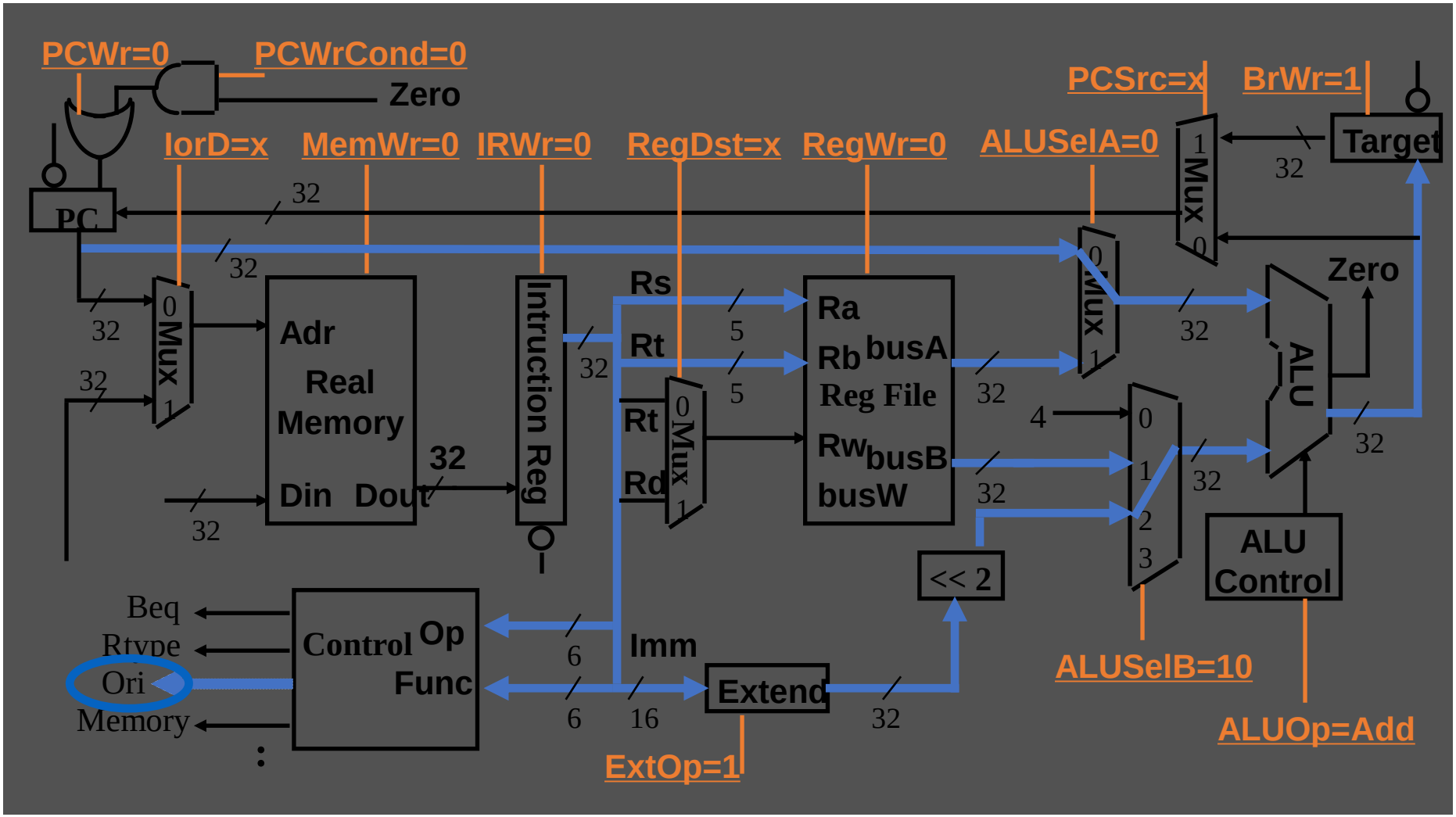
R-type完成周期（第四个周期，写寄存器）



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



寄存器取 / 指令译码周期（第二个周期）



指令译码输出为： ori

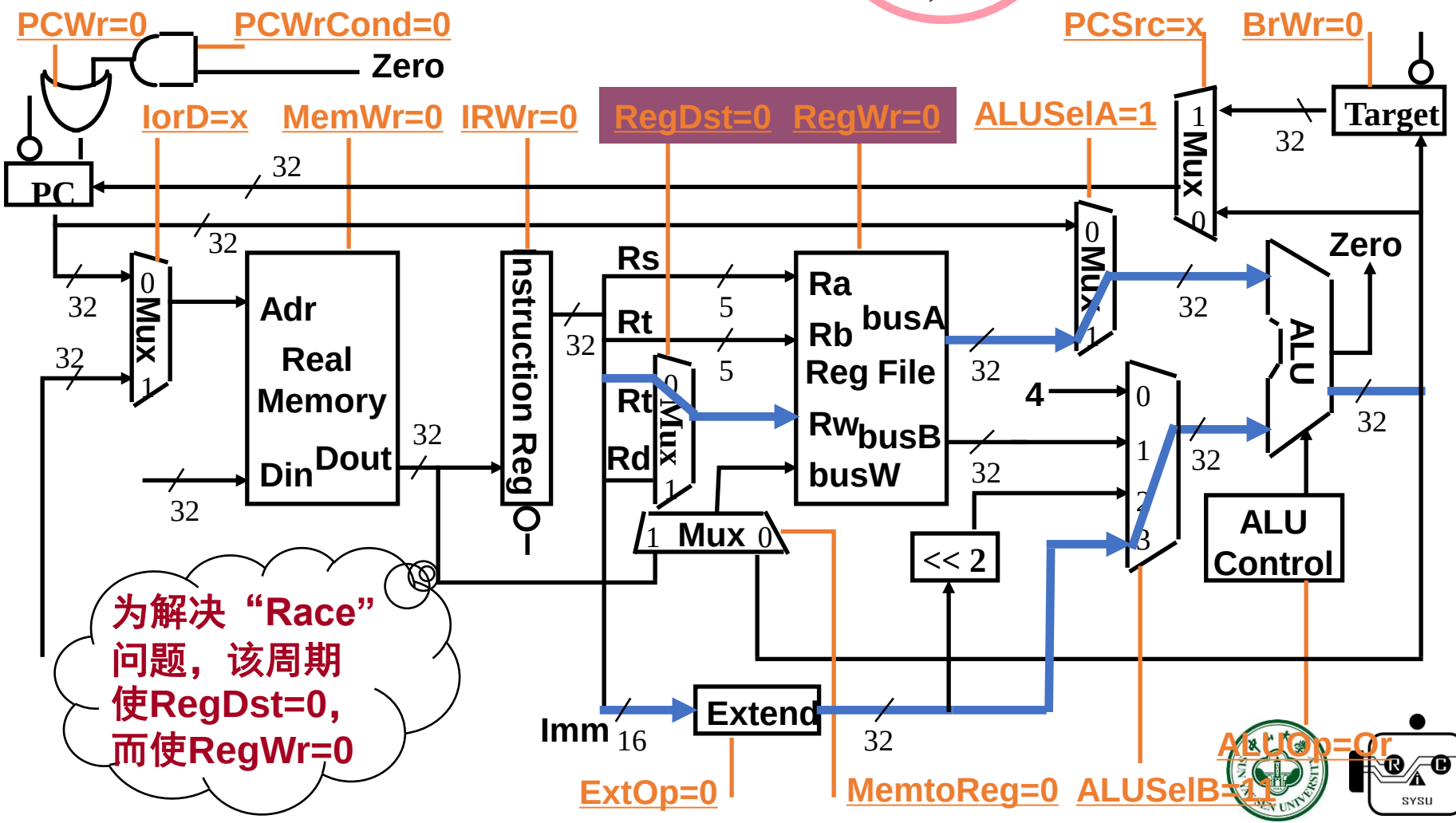
下面第三个周期就是ori指令的第一个执行周期！

Ori 指令执行周期（第三个周期，ALU运算）

- ALU output ← busA or ZeroExt[Imm16]

ALUOp=Or
1: ALUSelA
ALUSelB=11
x: MemtoReg
IorD, PCSrc

ori指令的第一个周期，控制信号取值？

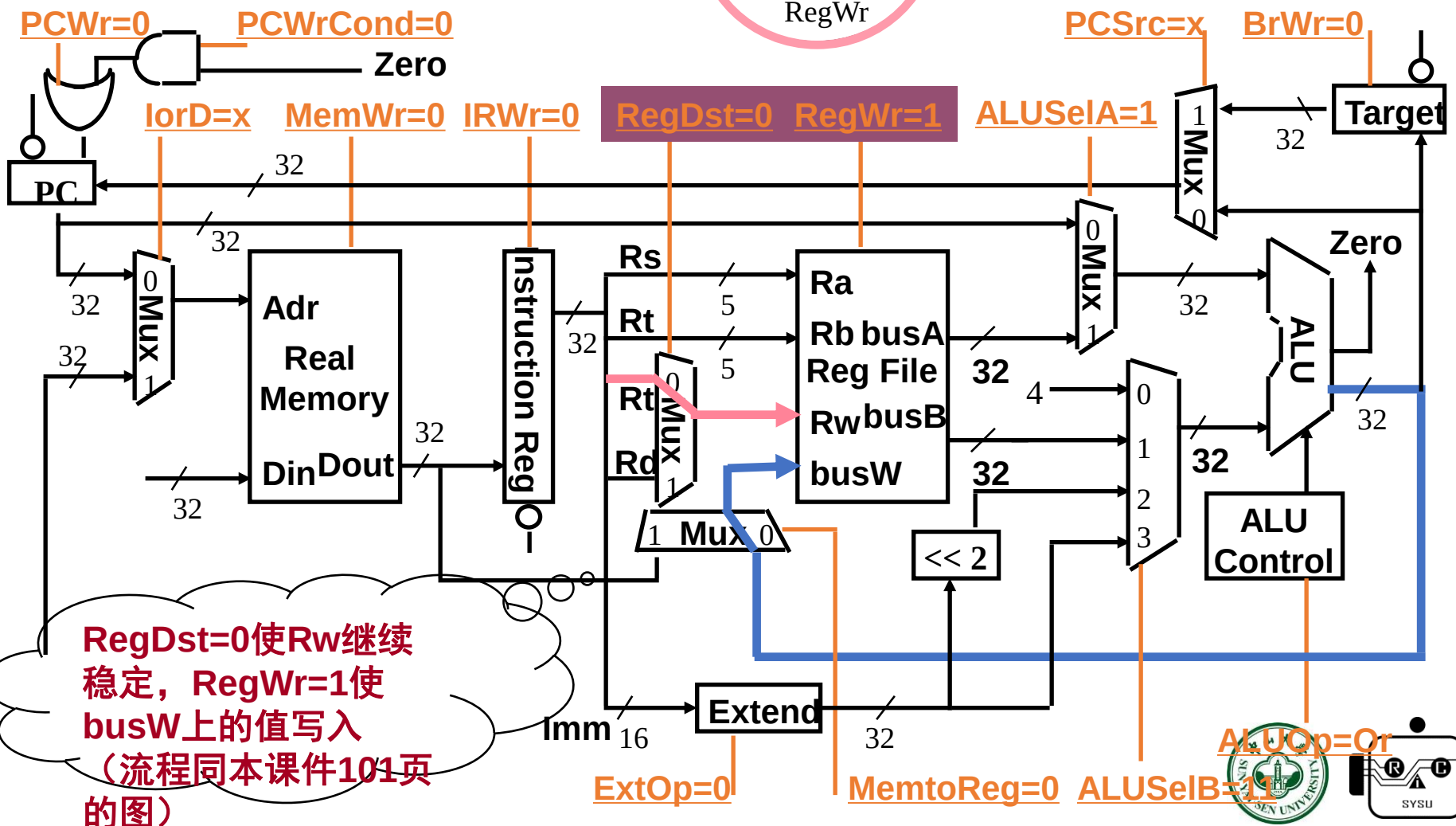


Ori 指令完成周期（第四个周期，写寄存器）

R [rt] ← ALU output

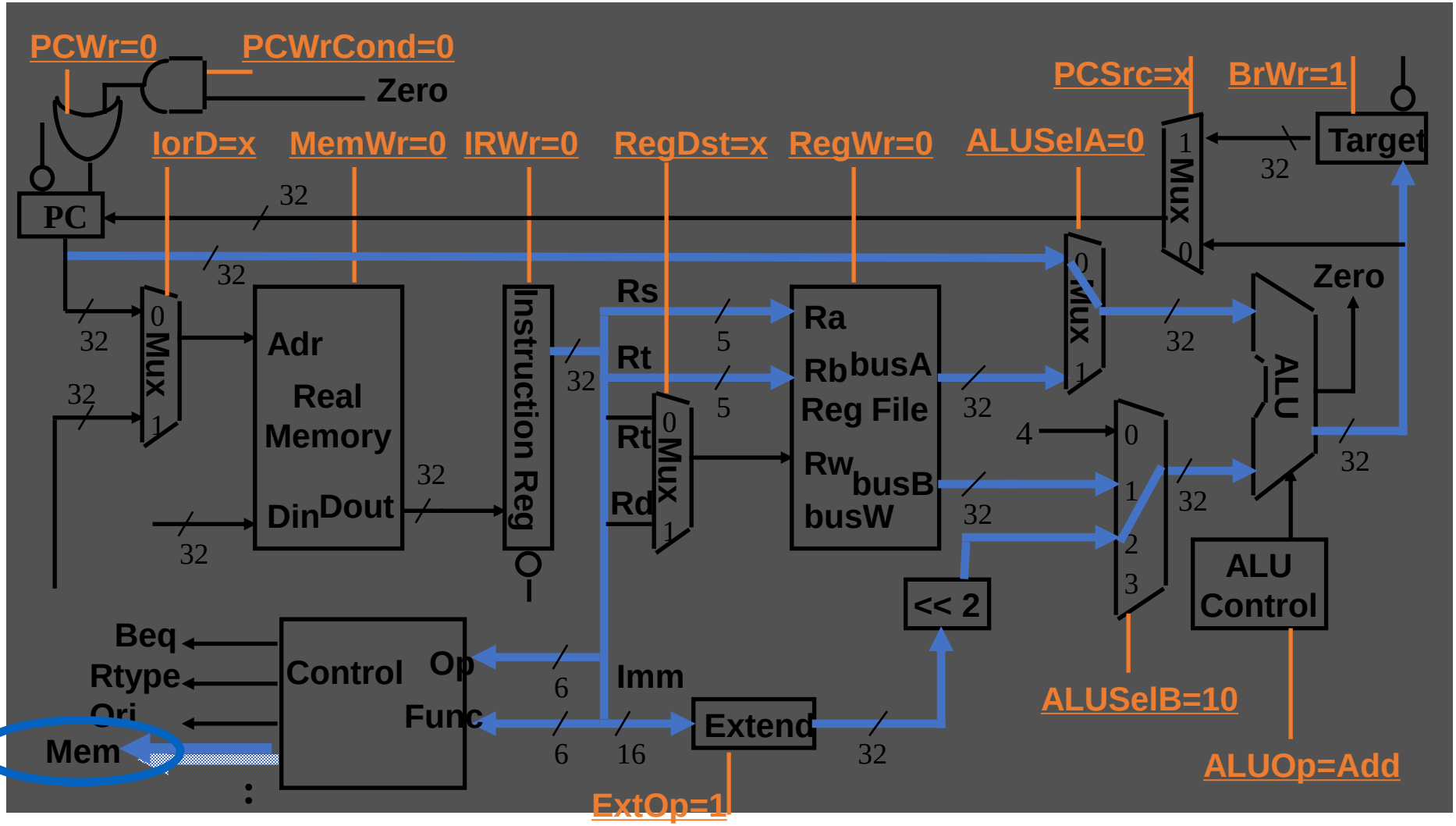
ALUOp=Or
x: IorD, PCSrc
ALUSelB=11
1: ALUSelA
RegWr

ori指令的第二个周期控制信号取值？



RegDst=0使Rw继续稳定，RegWr=1使busW上的值写入（流程同本课件101页的图）

寄存器取 / 指令译码周期（第二个周期）



指令译码输出：访存指令（lw 或 sw）

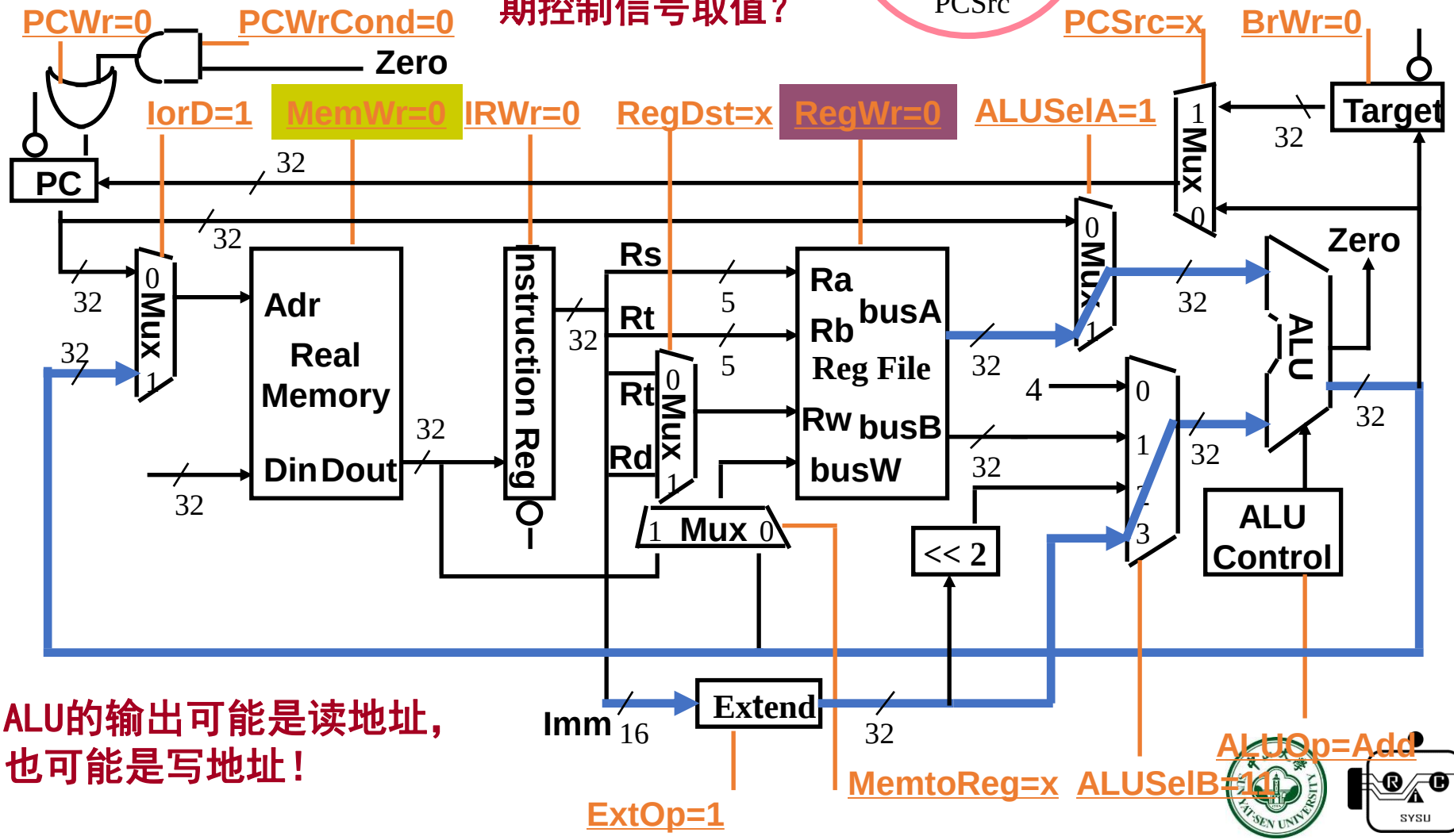
第三个周期就是lw/sw指令的第三个周期！

lw/sw内存地址计算周期（第三个周期，ALU运算）

- ALU output ← busA + SignExt[Imm16]

1: ExtOp
ALUSelA
ALUSelB=11
ALUOp=Add
x: MemtoReg
PCSrc

lw/sw指令的第一个周期控制信号取值？



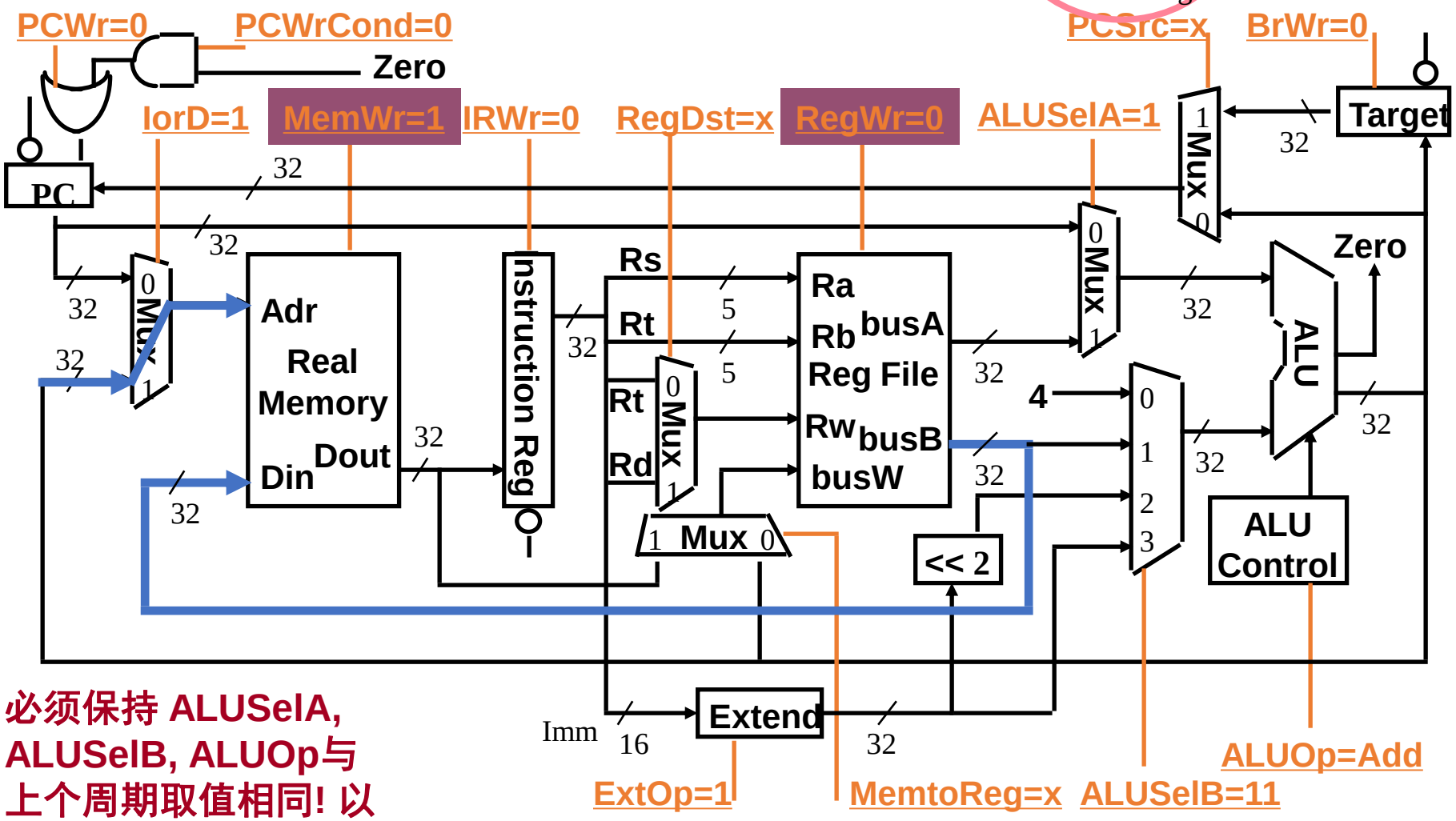
ALU的输出可能是读地址，也可能是写地址！

sw指令存数周期（第四周期，访存写）

- M[ALU output] ← busB

1: ExtOp
MemWr
ALUSelA
ALUSelB=11
ALUOp=Add
x: PCSrc,RegDst
MemtoReg

sw指令的第二个周期，控制信号取值？



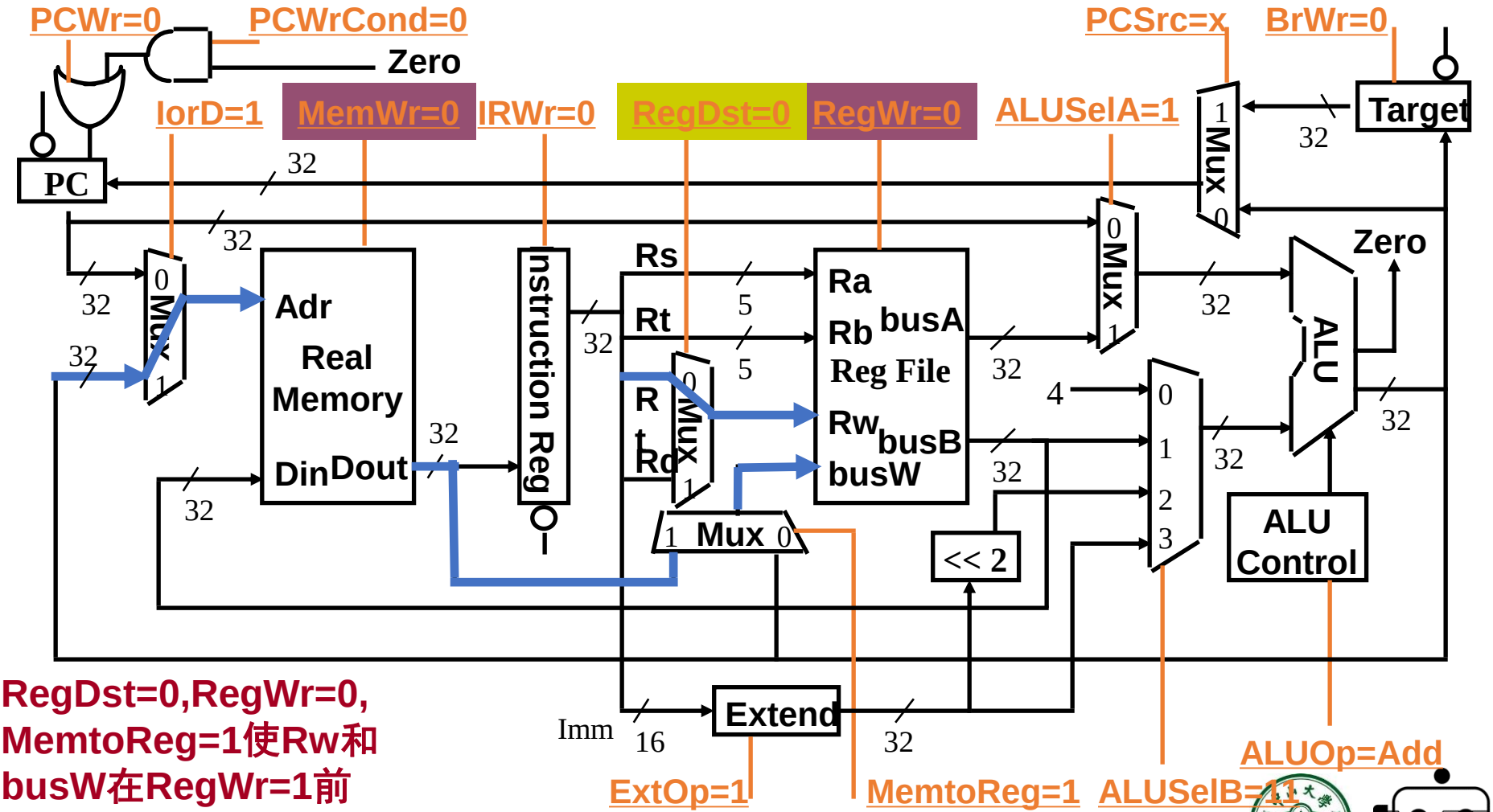
必须保持 ALUSelA, ALUSelB, ALUOp与上个周期取值相同! 以保证Adr稳定不变!

lw指令取数周期（第四周期，访存读）

- Mem Dout ← M[ALU output]

1: ExtOp
ALUSelA, IorD
ALUSelB=11
ALUOp=Add
x: MemtoReg
PCSrc

lw指令的第二个周期，控制信号取值？

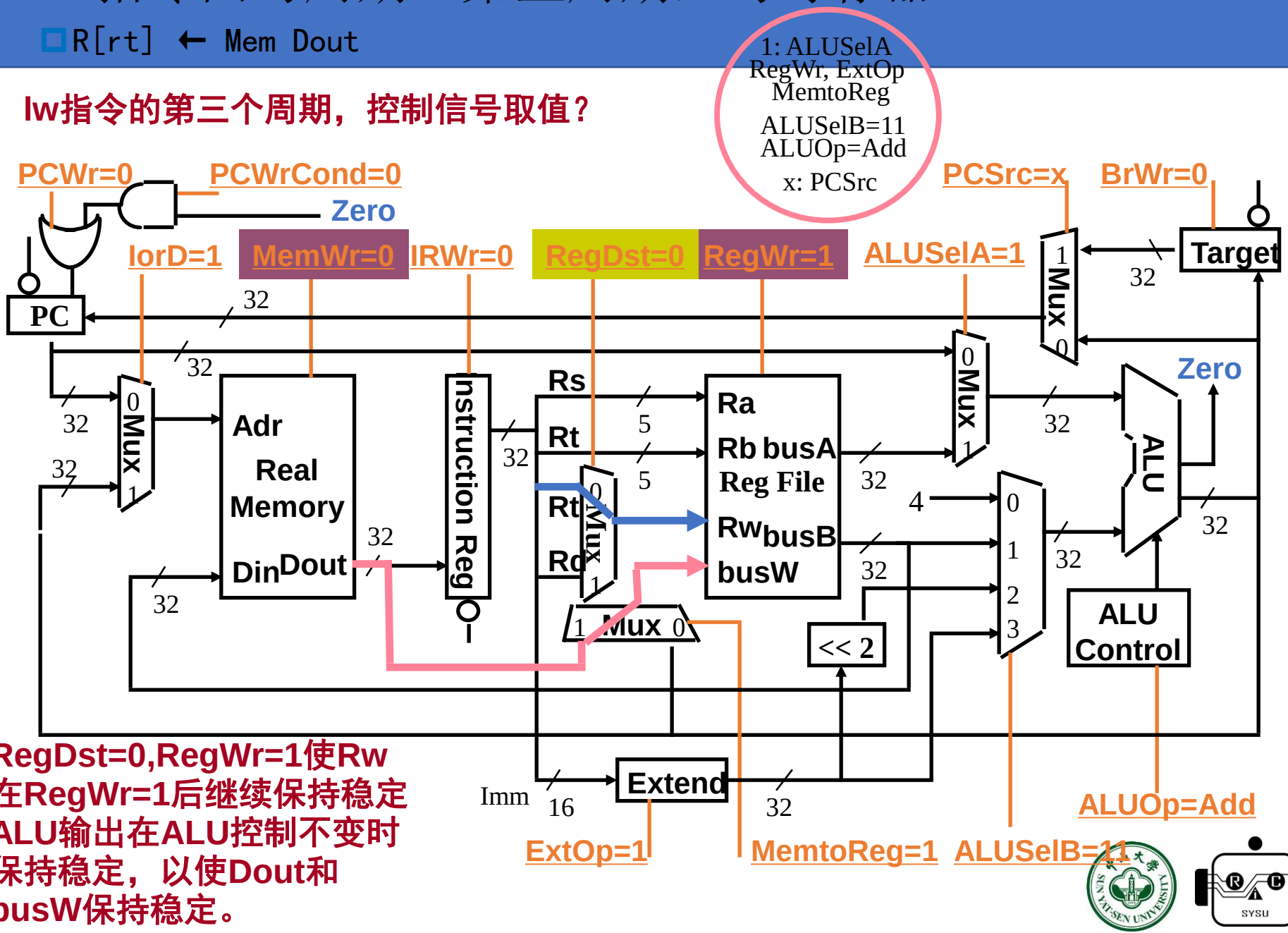


RegDst=0,RegWr=0,
MemtoReg=1使Rw和
busW在RegWr=1前
先稳定

lw指令回写周期（第五周期，写寄存器）

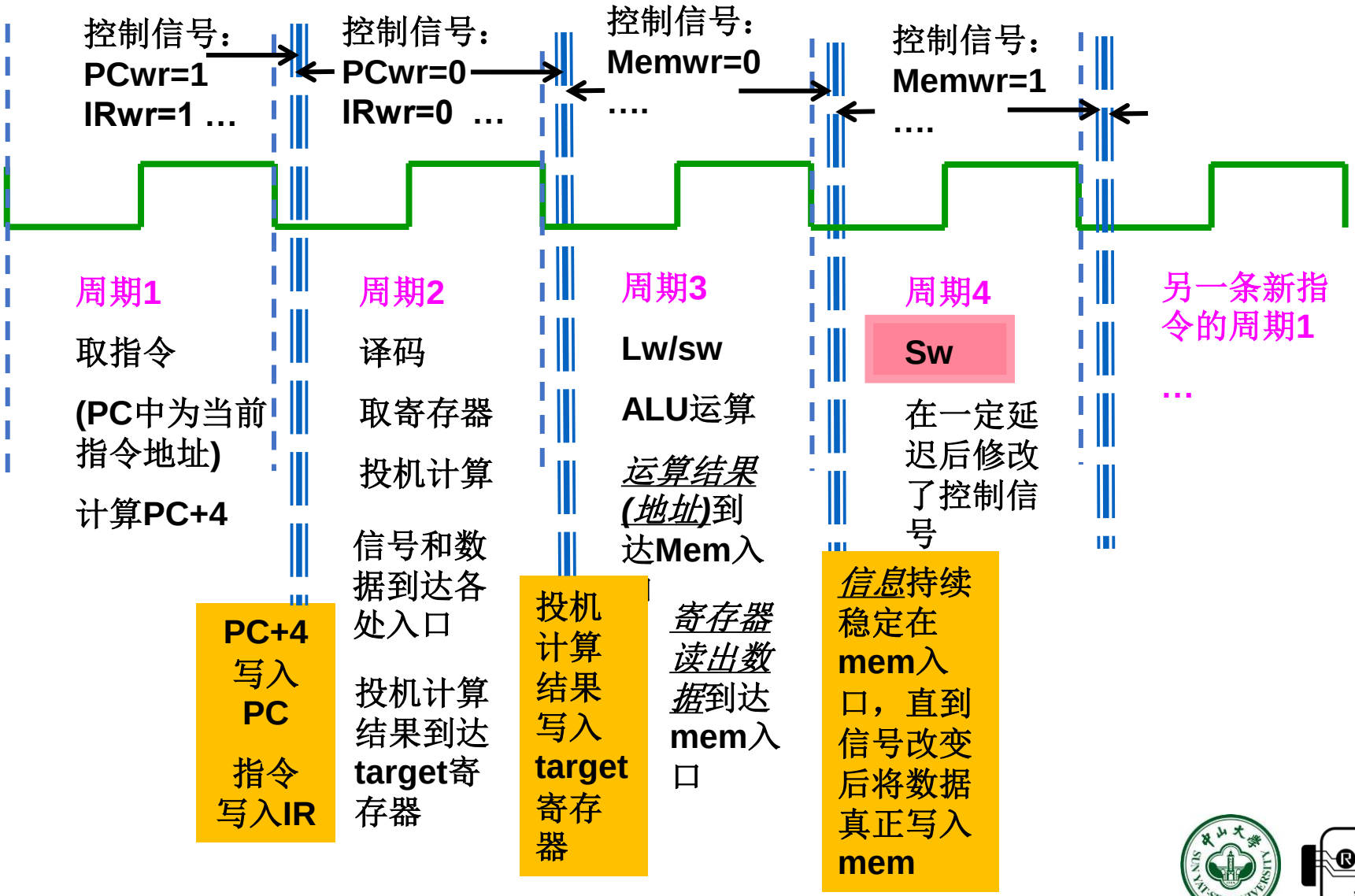
- R[rt] ← Mem Dout

lw指令的第三个周期，控制信号取值？

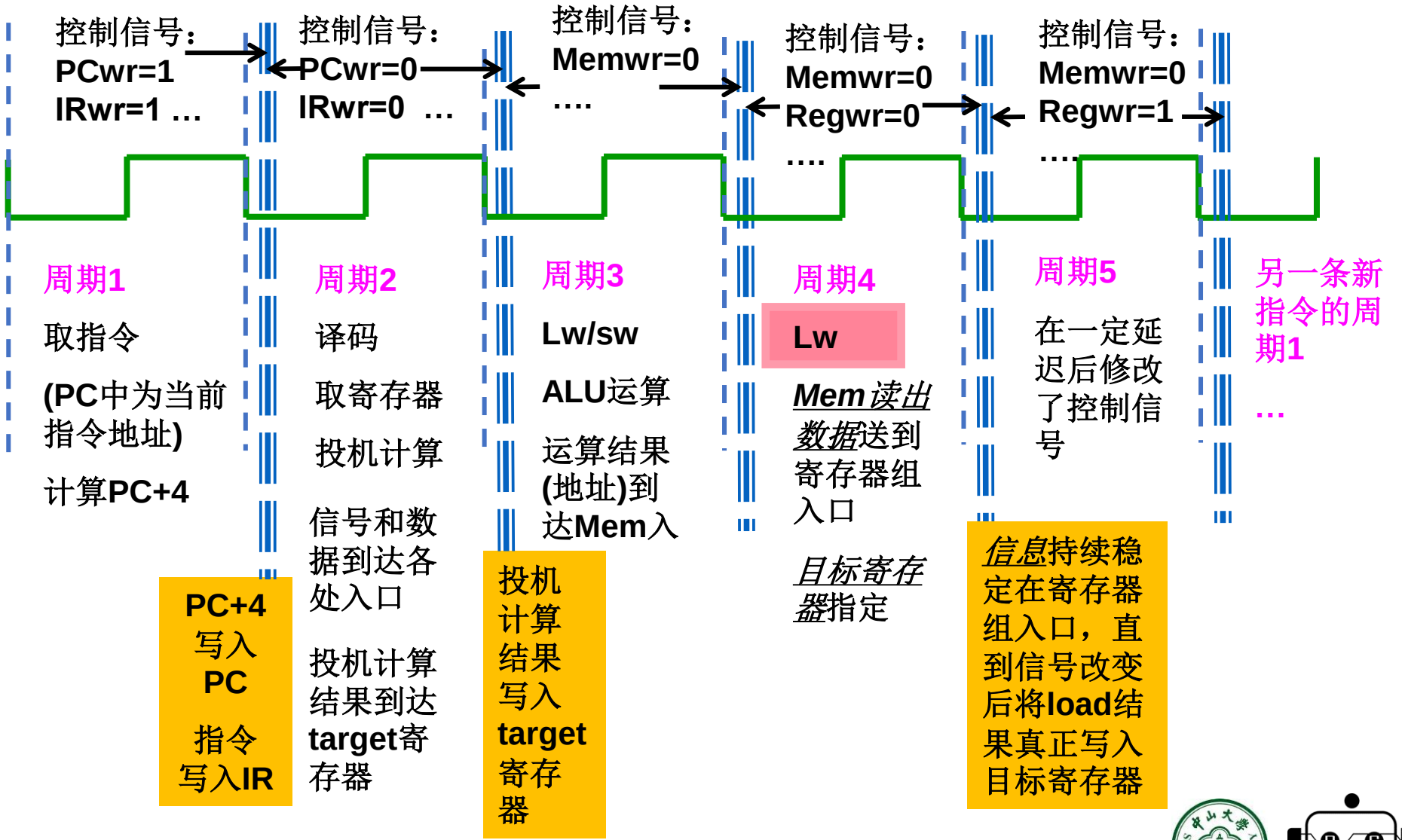


RegDst=0,RegWr=1使Rw在RegWr=1后继续保持稳定
ALU输出在ALU控制不变时保持稳定，以使Dout和busW保持稳定。

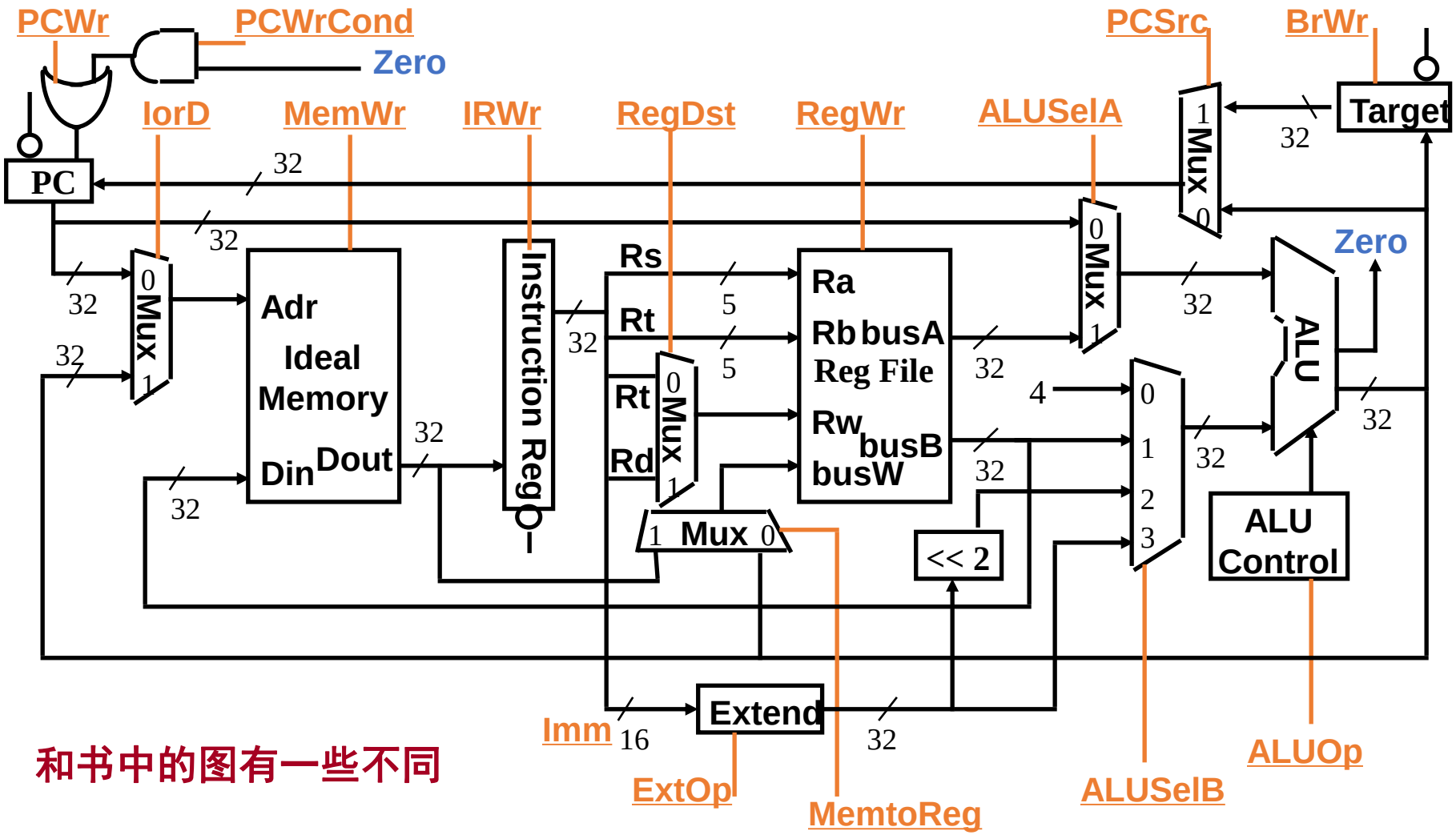
状态元件内容和控制信号取值的改变：在时钟到来后的一定延时后发生



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



完成前述指令的完整多周期数据通路（没有Jump）



和书中的图有一些不同

下面关键是如何控制在不同周期产生不同的控制信号取值！这就是控制器的任务。下面考虑如何设计控制器！

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？

不可以，因为：
每个指令有多个周期，每个周期控制信号取值不同！

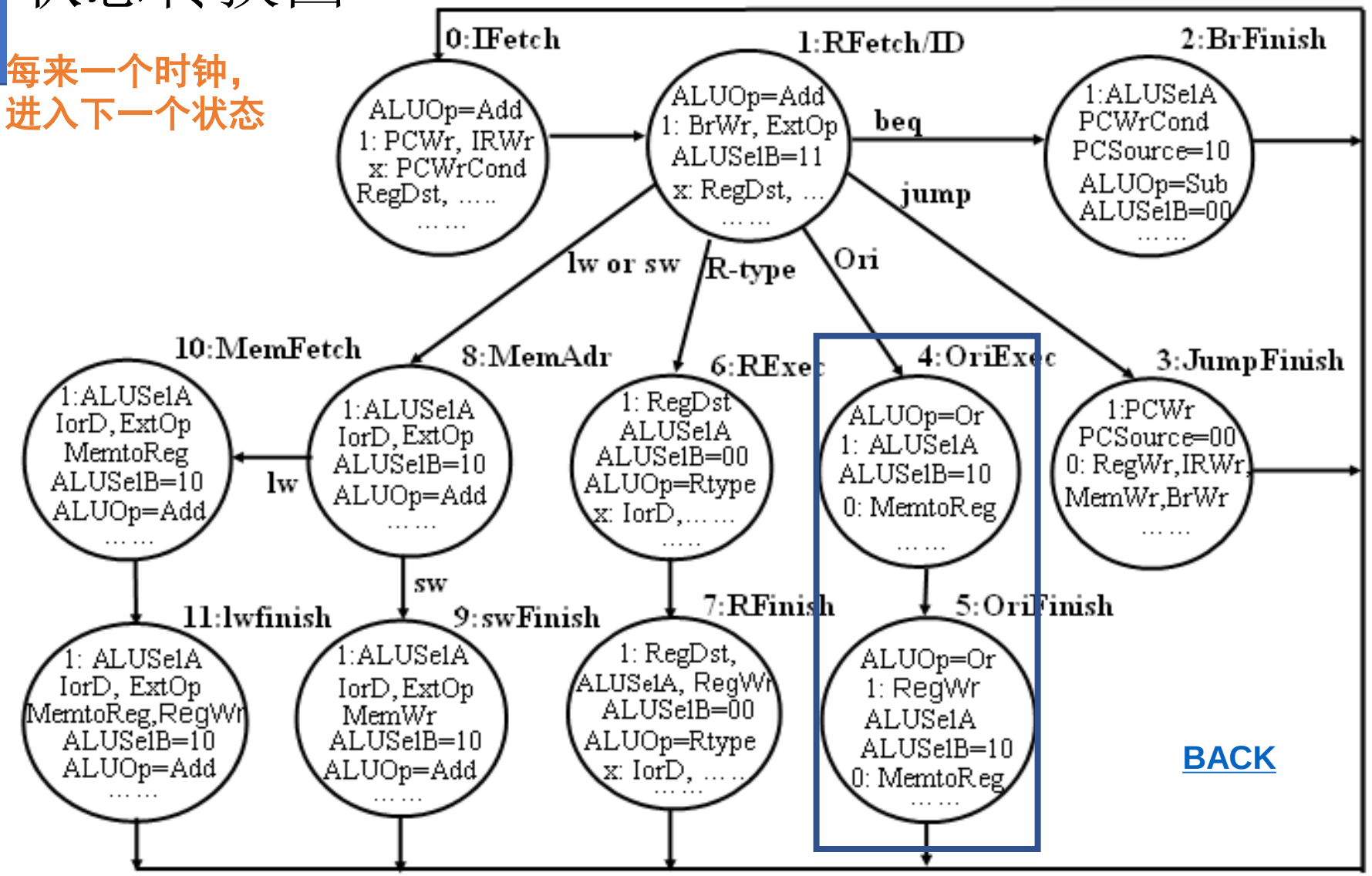
多周期控制器功能描述方式：

- 有限状态机：
  - 采用组合逻辑设计
  - 用硬连线路(PLA)实现
- 微程序：
  - 用ROM存放微程序实现

状态转换图

每个状态下，输出的控制信号有相应的不同取值！

每来一个时钟，
进入下一个状态

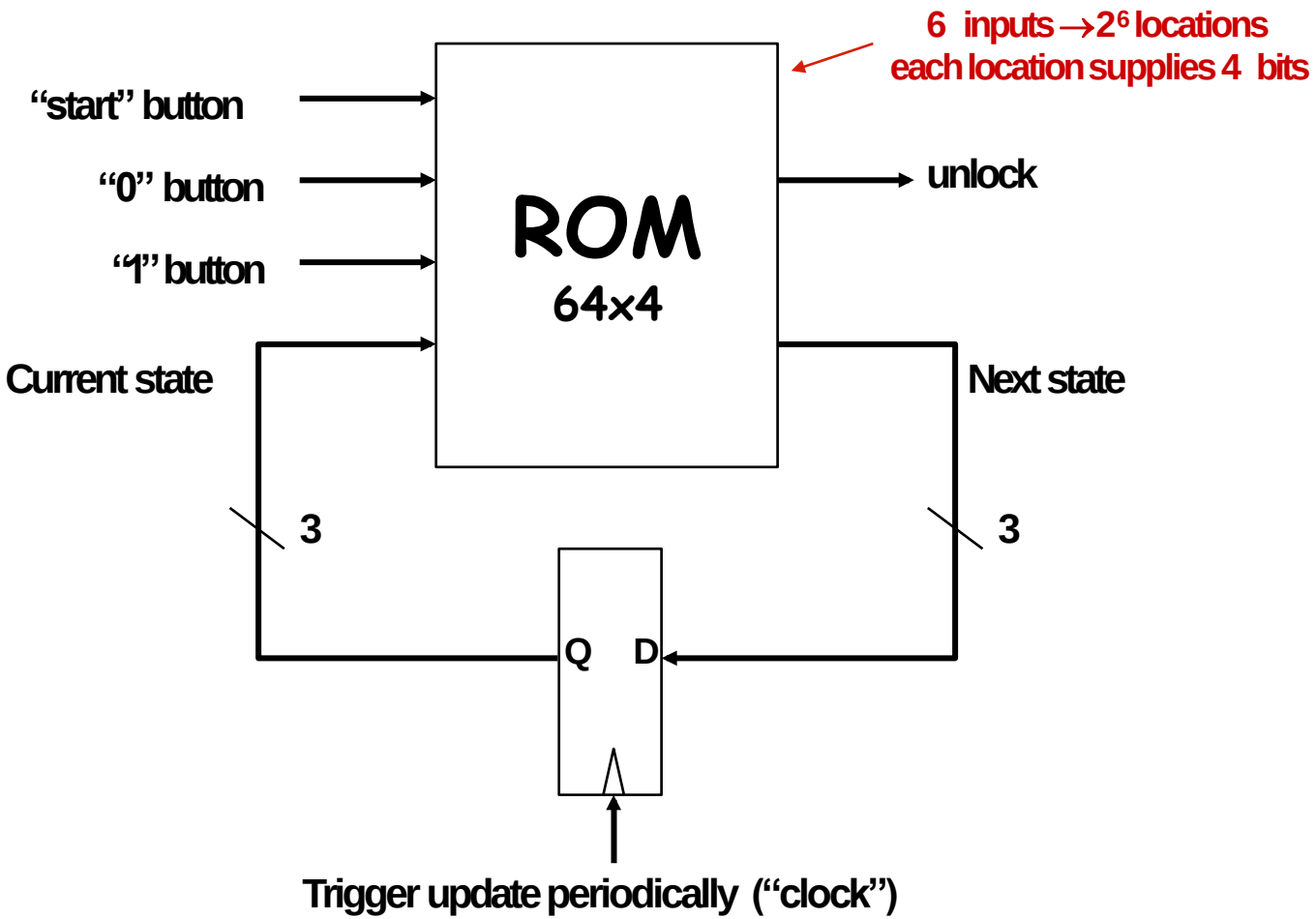


问题：各指令的时钟数多少？

R-4, ori-4, beq-3, Jump-3, lw-5, sw-4

下一步目标：设计“状态转换电路” 即：控制器

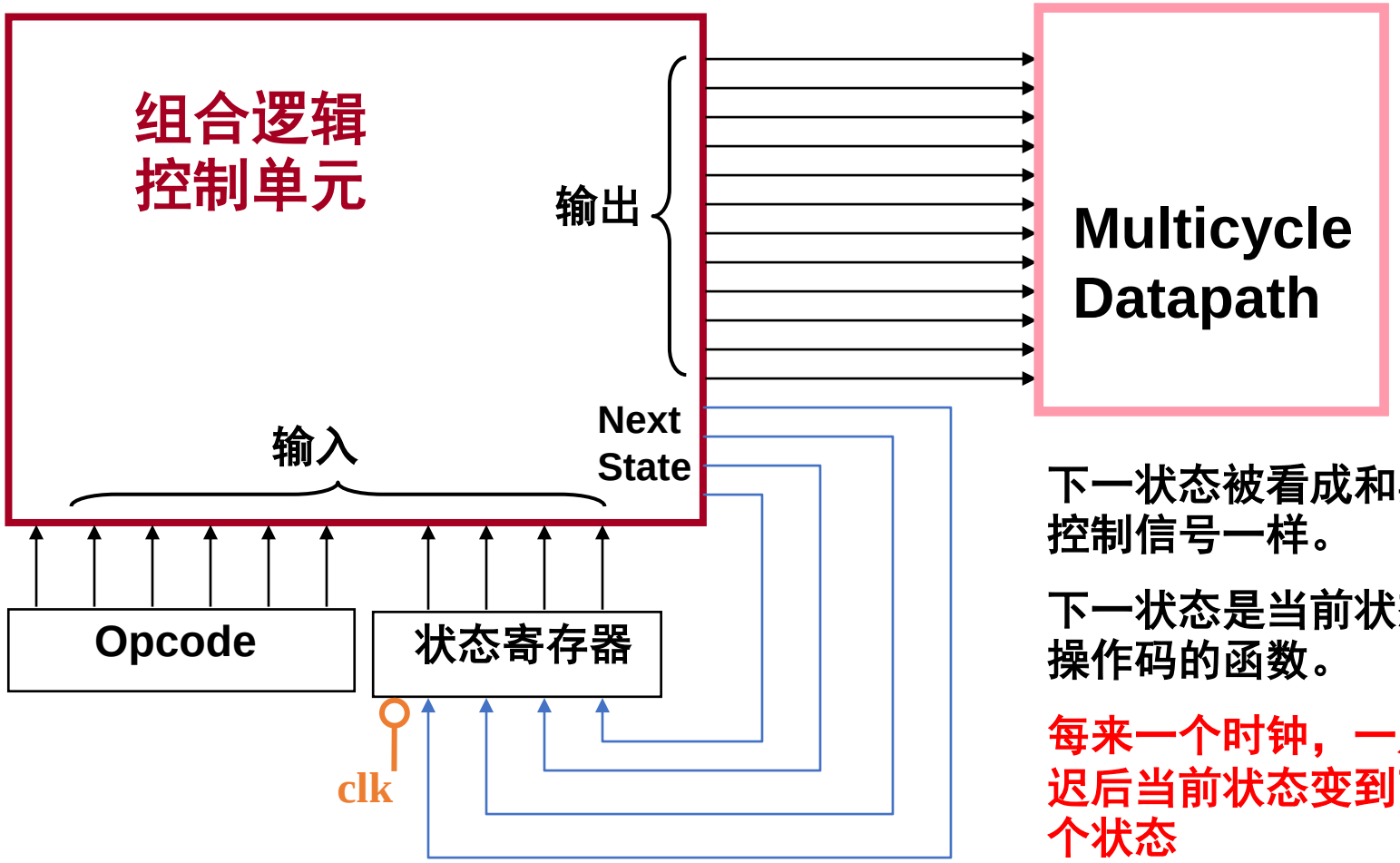
FSM: Implementation with ROM



时序控制的描述

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，一定延迟后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。

下一步目标：设计控制逻辑（control Logic）

```
module maindec(
    input wire[5:0] op,

    output wire memtoreg,memwrite,
    output wire branch,alusrc,
    output wire regdst,regwrite,
    output wire jump,
    output wire[1:0] aluop
    );
    //add your code here according to lab 7
    localparam IFetch=0, RFetch=1, BrFinish=2, JumpFinish=3, AddiExec=4, AddiFinsh=5, REexec=6, RFinish=7, MemAdr=8, swFinish=9, MemFetch=10,

    reg [15:0] outcontrol;

    reg [3:0] state, nextstate;

    always@(negedge clk or posedge rst)
        begin
            if(rst==1)
                state<=IFetch;
            else
                state<=nextstate;
        end

    always@(*)
        begin
            nextstate=state;
            case(state)
                IFetch:
                    begin
                        nextstate=RFetch;
                        outcontrol=XXX;
                    end
                RFetch:
                    begin
                        if(op==beq)
                            begin
                                nextstate=BrFinish;
                                outcontrol=XXX;
                            end
                        //add other operation
                    end
                BrFinish:
```

多周期控制器的实现

❖指令数据通路综合（R型指令、Lw、Sw、Beq）

步骤	R型指令	Lw指令	Sw指令	Beq指令
取指令	IR ← M[PC] PC ← PC + 4 P0			
读寄存器/译码	A ← R[IR[25:21]] B ← R[IR[20:16]] ALUOut ← PC + Signext[IR[15:0]]<<2 P1			
计算	P6 ALUOut ← A op B	ALUOut ← A + Signext(IR[15:0]) P2		P8 If (A-B==0) then PC ← ALUout
R型完成/访问内存	P7 R[IR[15:11]] ← ALUOut	P3 DR ← M[ALUOut]	P5 M[ALUOut]← B	
写寄存器		P4 R[IR[20:16]] ←DR		

小结

- 单周期CPU和多周期CPU的成本比较：
  - 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
- 单周期CPU和多周期CPU的性能比较：
  - 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Branch，2%为Jump。每个状态需要一个时钟周期，CPI为多少？

分析如下：每种指令所需的时钟周期数为：

Load：5；Store：4；R-Type：4；Branch：3；Jump：3

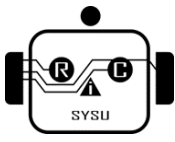
CPI=CPU时钟周期数 / 指令数 = Σ（指令数$_i$ x CPI$_i$）/ 指令数

= Σ（指令数$_i$ / 指令数）x CPI$_i$

CPI = 0.22x5+0.11x4+0.49x4+0.16x3+0.02x3 = 4.04

假设单周期时钟宽度为1，则多周期时钟周期约为单周期的1/5，所以，多周期的总体时间约：4.04x1/5=0.8（其实不可能，也许只是1/3，那么4.04x1/3 > 1）；而单周期总体时间为：1x1=1

由此看出：划分阶段很均匀时，多周期可能比单周期效率高！



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