南京大学
NANJING UNIVERSITY

中央处理器

殷亚凤
智能软件与工程学院
苏州校区南雍楼东区225
yafeng@nju.edu.cn，https://yafengnju.github.io/

中央处理器

- CPU概述
- 单周期处理器设计
- **多周期处理器设计**
- 带异常处理的处理器设计

多周期处理器的设计

单周期处理器的缺陷

- **单周期处理器的CPI为1，所有指令的执行时间都以最长的load指令为准**
- **最长指令时间为**:
  - 锁存时间+
  - 取指令时间+
  - 寄存器取数时间+
  - ALU延迟+
  - 存储器取数时间+
  - 建立时间+
  - 时钟偏移
- **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，不需要读内存，也不需要读/写寄存器

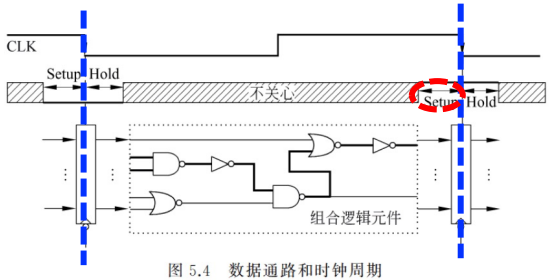


图 5.4 数据通路和时钟周期

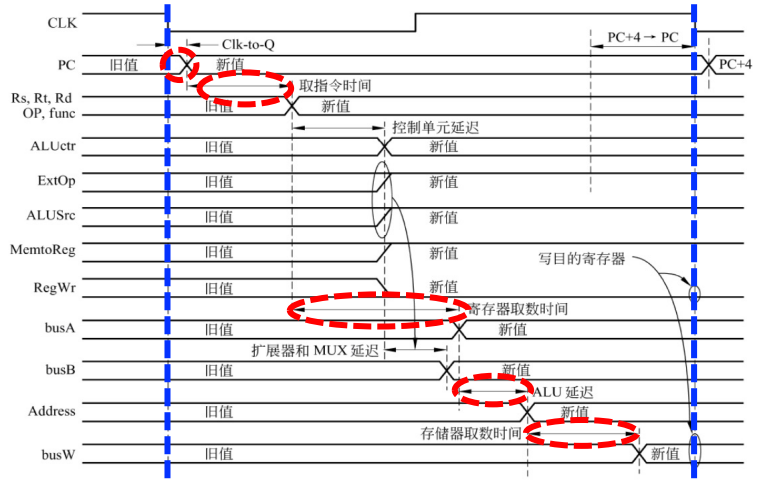
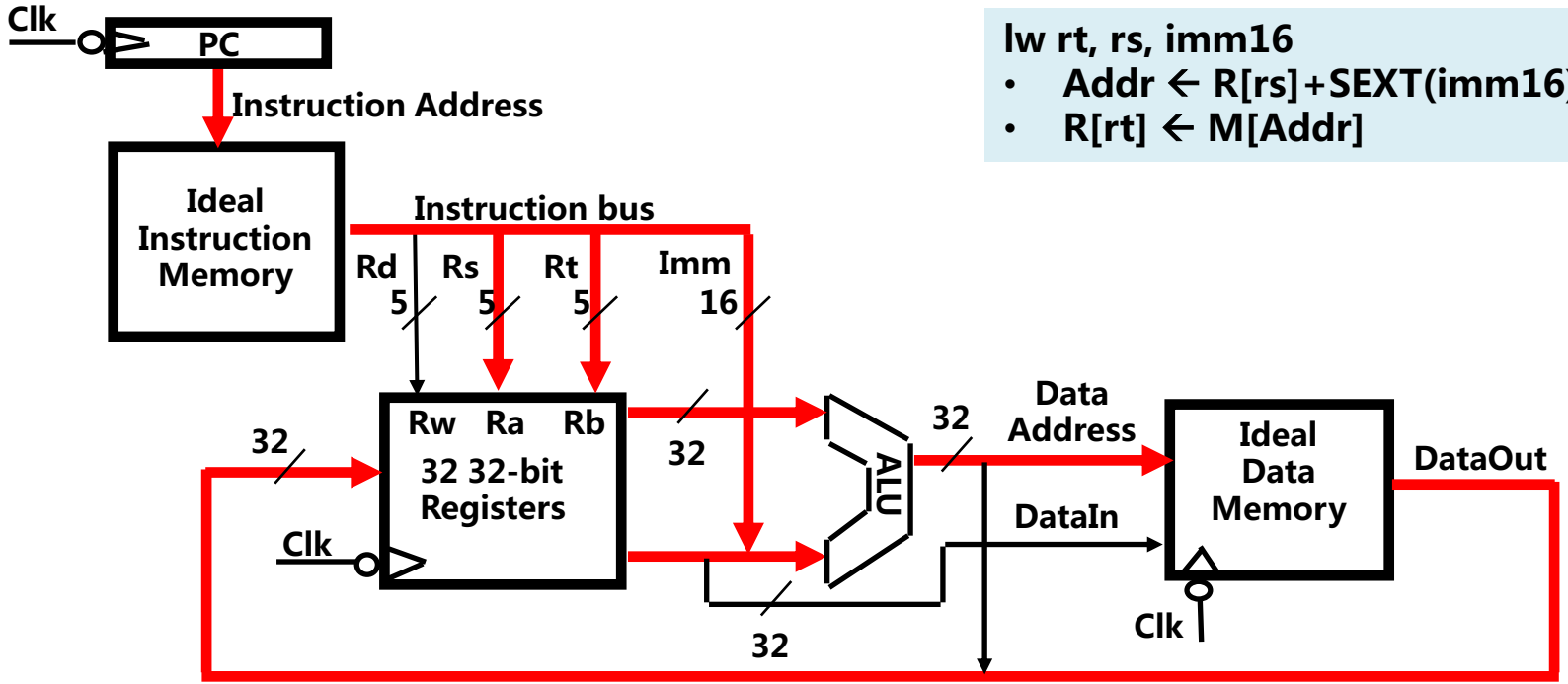


图 5.24 load 指令执行定时

多周期处理器的设计

Load指令执行过程



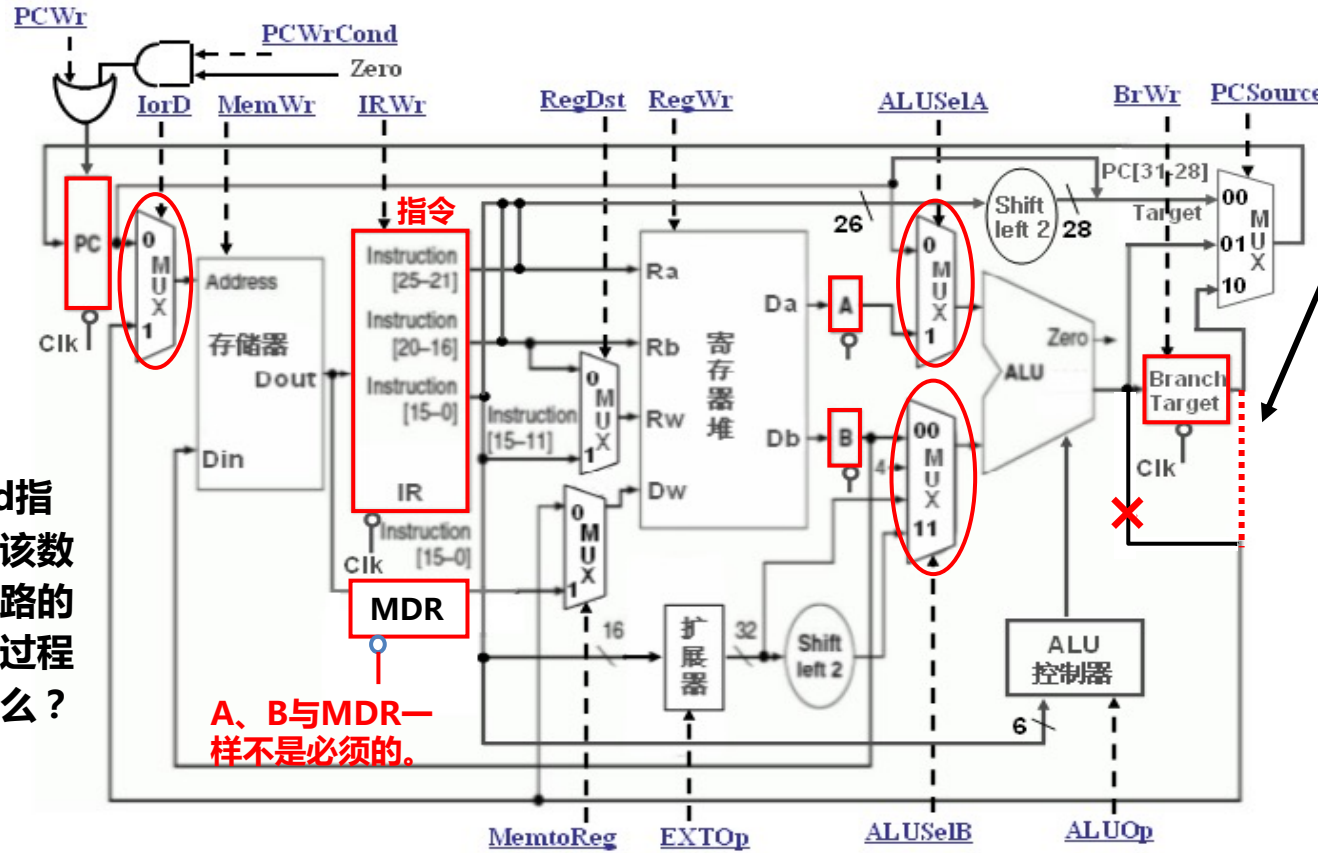
lw rt, rs, imm16

- Addr ← R[rs]+SEXT(imm16)
- R[rt] ← M[Addr]

多周期处理器设计思路

- **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**:
  - 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，每部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，如：
    - Load： 5个时钟周期
    - Jump：3个时钟周期（前两个都一样）
  - 允许功能部件在一条指令执行过程中被重复使用。如：
    - 单周期有Adder和ALU（多周期时只用一个ALU，在不同周期可重复使用）
    - 单周期分开指令和数据存储器（多周期时合用，不同周期中重复使用）

多周期数据通路设计



能否对P.151图5.25作左侧所述调整？

不行！因为取数时所需的BT会被随后结果冲掉！

Load指令在该数据通路的执行过程是什么？

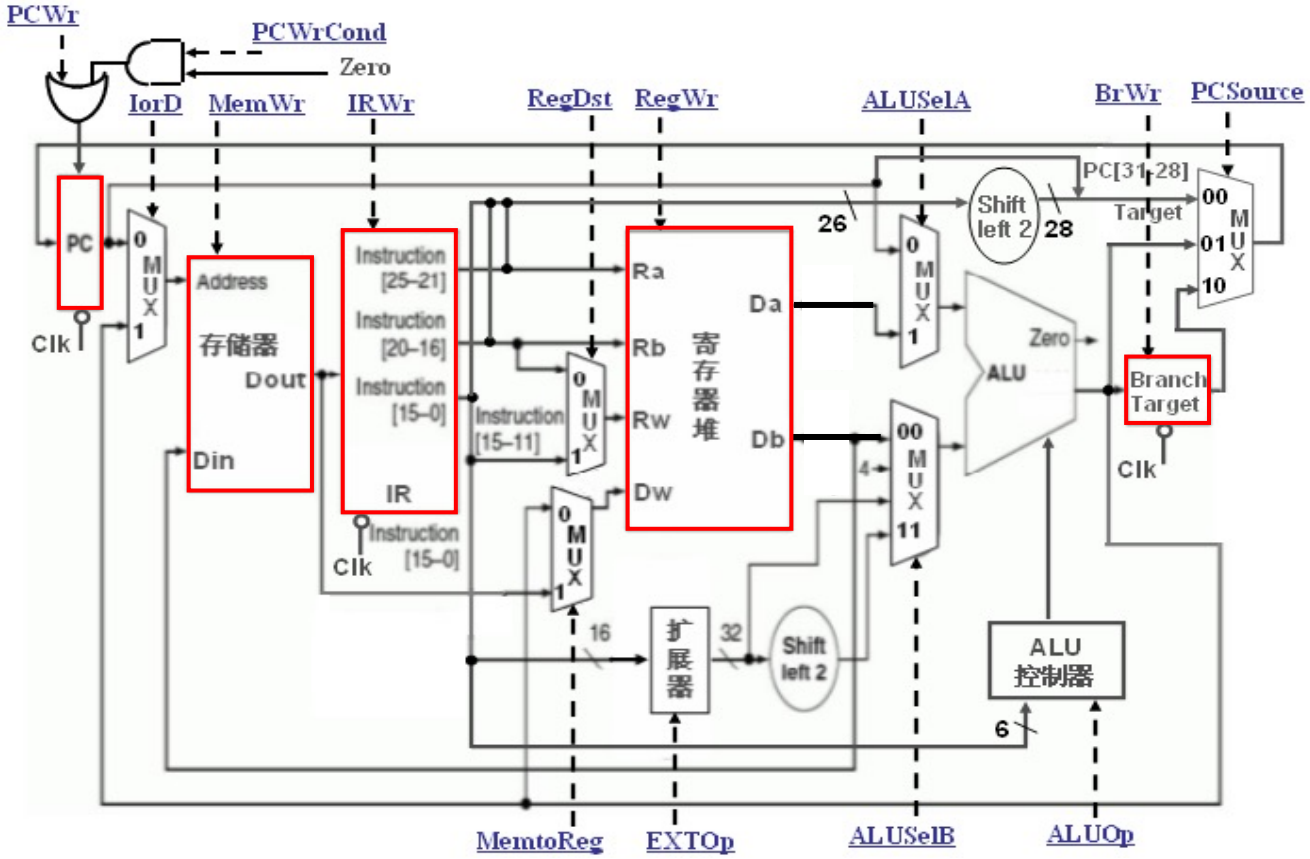
A、B与MDR一样不是必须的。

只有一个ALU、一个存储器，并在多处增加了MUX和临时寄存器（加红框的）

多周期数据通路设计——Load指令各阶段分析

- **取指令**
  - 根据PC读出指令保存到寄存器IR（指令寄存器）,并执行PC+4
  - IR的内容不是每个时钟都更新，所以IR必须加一个“写使能”控制
  - 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都更新PC，所以PC也要有“写使能”控制
- **译码/读寄存器**
  - 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
  - 读出的内容（操作数）保存到临时寄存器A和B中
  - 每个时钟到来时，A和B中的值都要更新，所以不需“写使能”控制
  - 计算分支目标地址：PC+4+4*SignExt(imm16), 保存到临时寄存器BranchTarget中
- **地址生成（ALU运算）**
  - 选择A寄存器中的内容和扩展器的输出分别送A口和B口，控制ALU进行addu运算，结果在ALU的输出端
- **读存储器**
  - 由ALU输出作为地址访问存储器，读出数据，保存在临时寄存器MDR中
- **写结果到寄存器**
  - 把MDR中的内容写到寄存器堆中

多周期数据通路设计——多周期数据通路



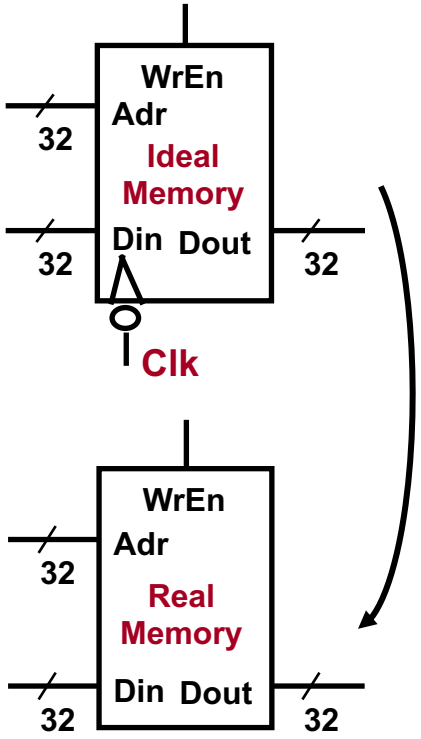
存储元件有PC、存储器、IR、寄存器堆和BranchTarget

多周期数据通路设计——寄存器堆和存储器的写定时

- **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！



多周期数据通路设计——竞争问题

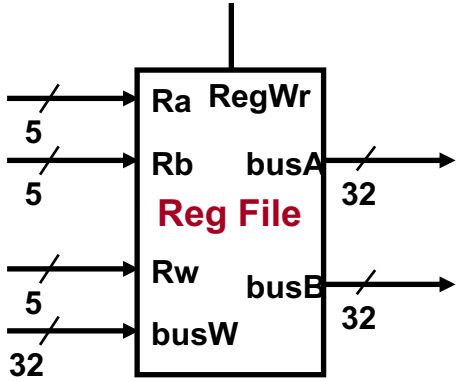
- **Register File(寄存器组)：**

  实际寄存器组在单周期通路中不能可靠工作

  这是因为：

  – 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”



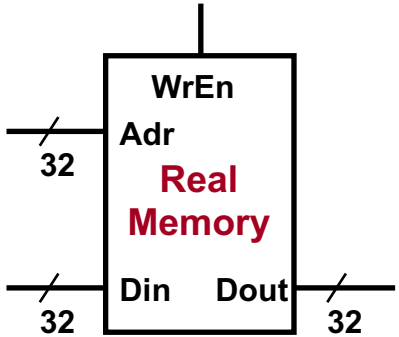
- **Memory(存储器)：**

  实际存储器在单周期通路中也不能可靠工作

  这是因为：

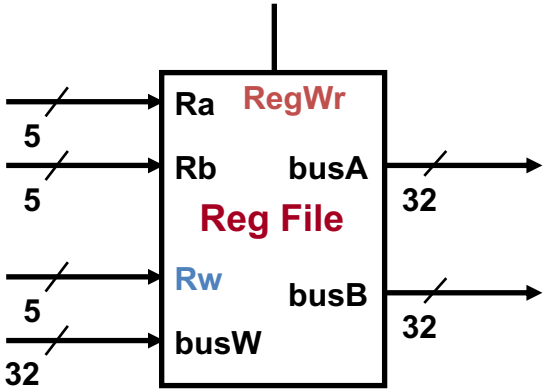
  – 不能保证 Adr和Din在WrEn=1之前稳定

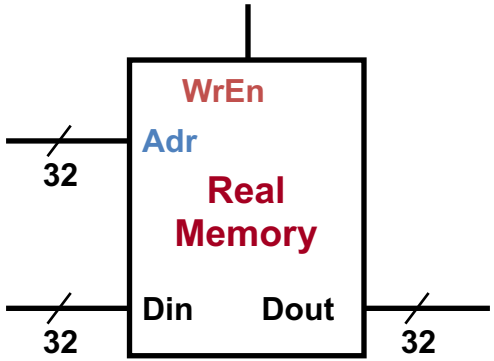
  即：在 Adr 和Din与 WrEn之间存在“race”



多周期数据通路设计——竞争问题

- **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

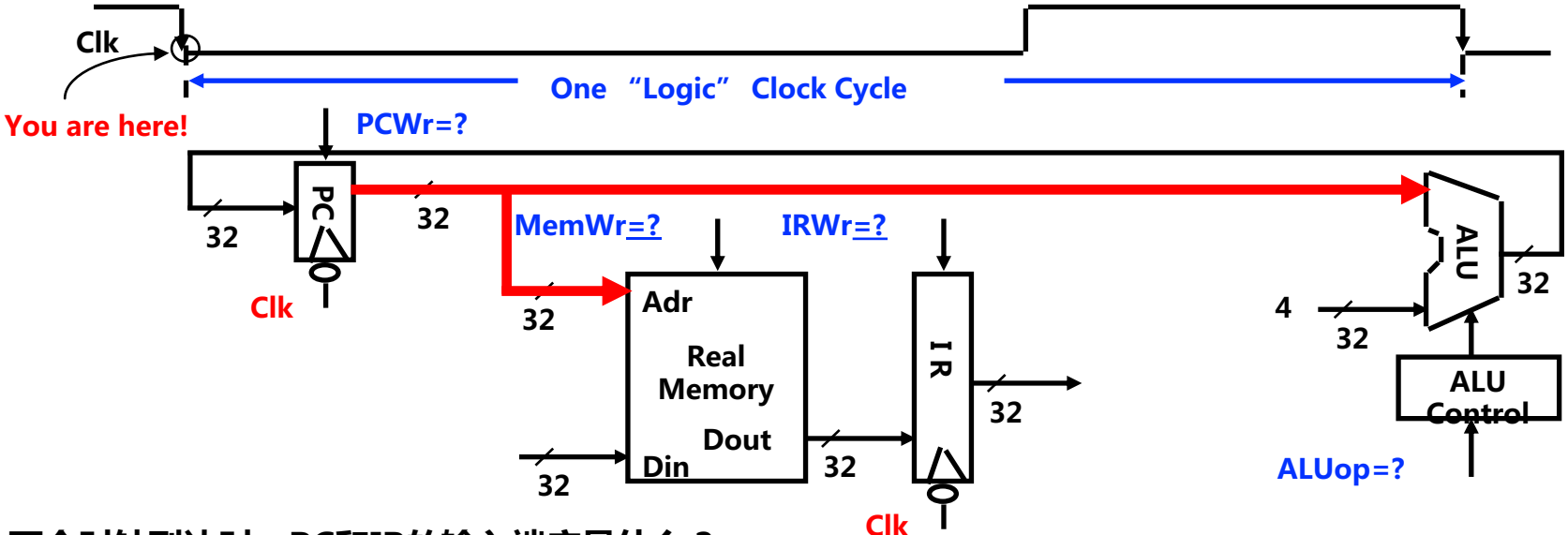




“Race”问题有时会导致机器神秘出错！

多周期数据通路设计——取指周期开始时

- 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

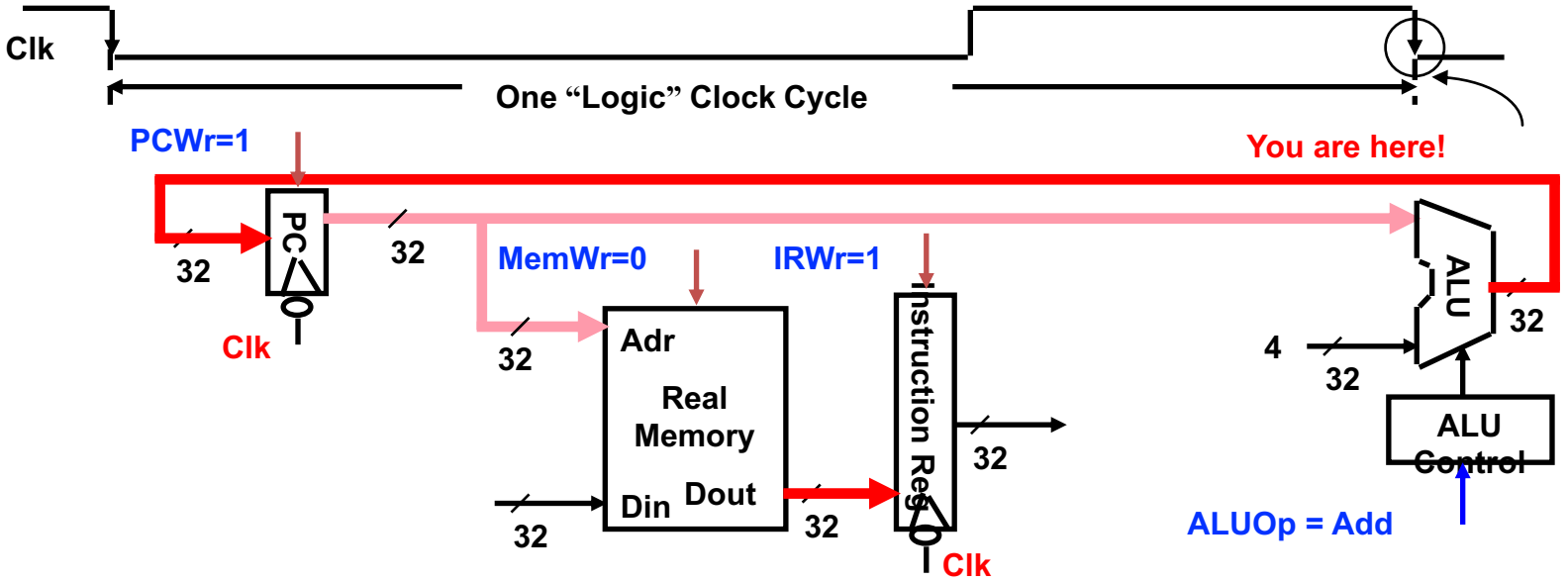


下个时钟到达时，PC和IR的输入端应是什么？
能否每个时钟更新PC和IR? 不能！
PC和IR怎样在必要时更新？加“写使能”控制！

PCWr=?, MemWr=?, IRWr=?, ALUop=?
PCWr=1, MemWr=0, IRWr=1, ALUop=addu

多周期数据通路设计——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，存储元件被更新):
  - IR ← M[PC]　　PC← PC + 4



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

多周期数据通路设计——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？

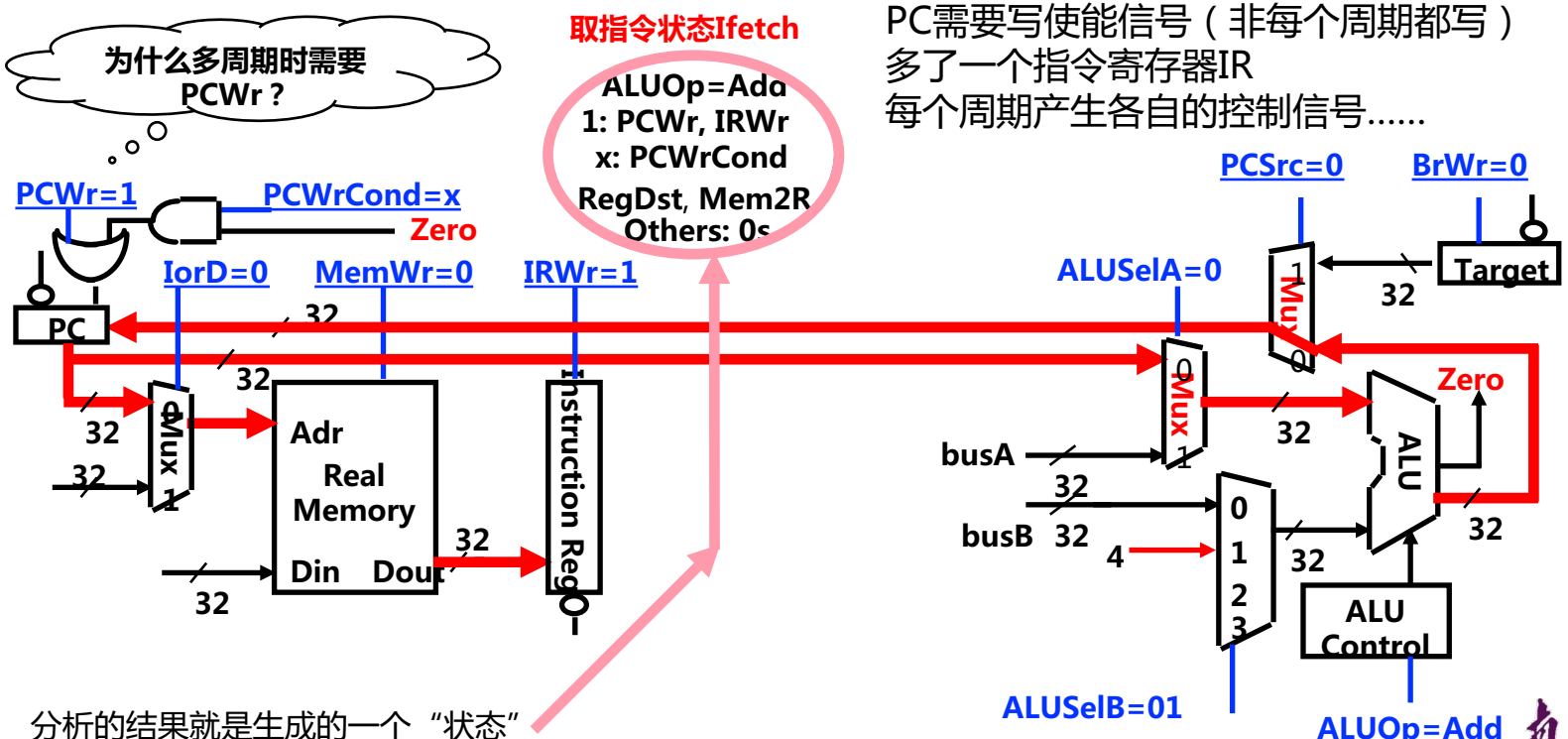
想想看，和单周期有哪些不同？

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……



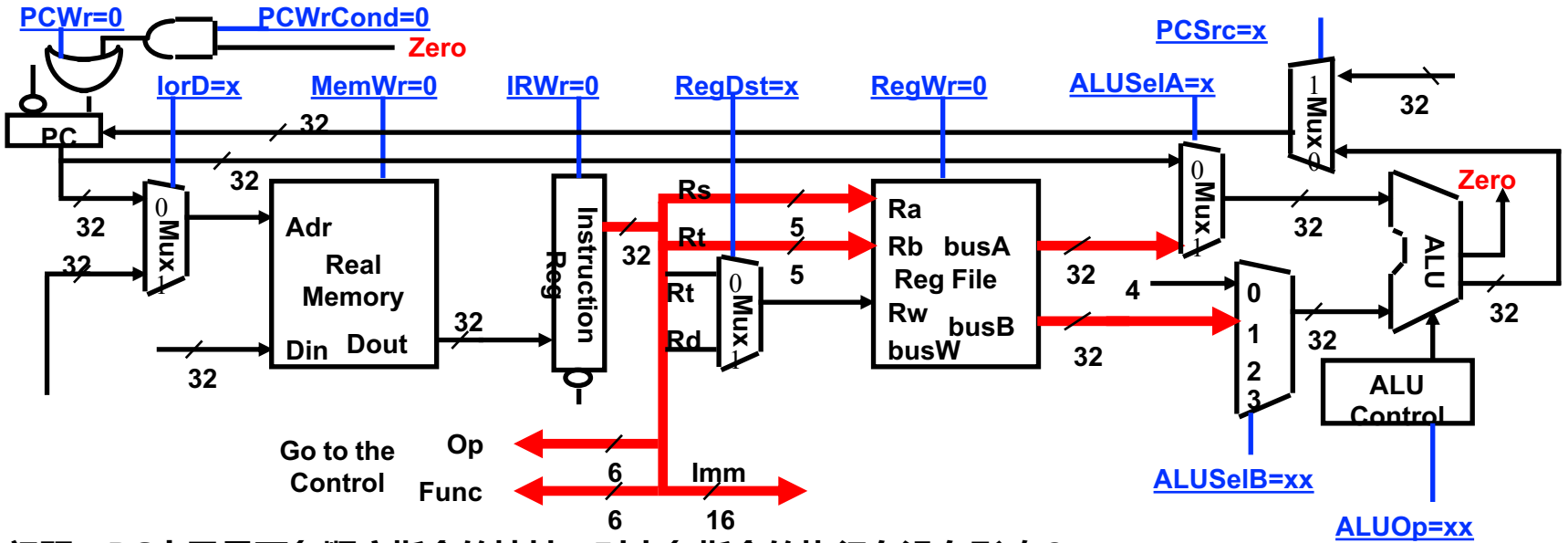
分析的结果就是生成的一个“状态”

多周期数据通路设计——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 is not being used: 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!　　转移地址投机计算的数据通路如何实现？

多周期数据通路设计——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

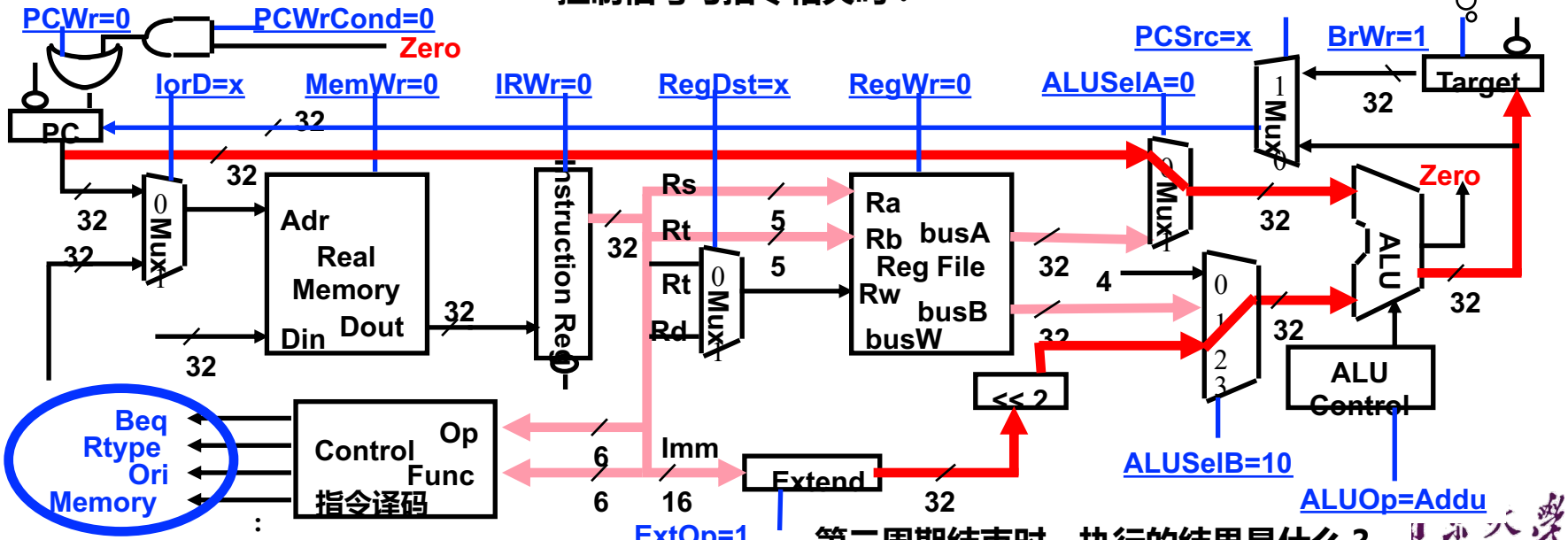
（为什么不是 PC+4+ SignExt(Imm16)*4？）

取并译码状态Rfetch/Decode

ALUOp=Add　1: BrWr, ExtOp
ALUSelB=10　x: RegDst, PCSrc
IorD, MemtoReg　Others: 0s

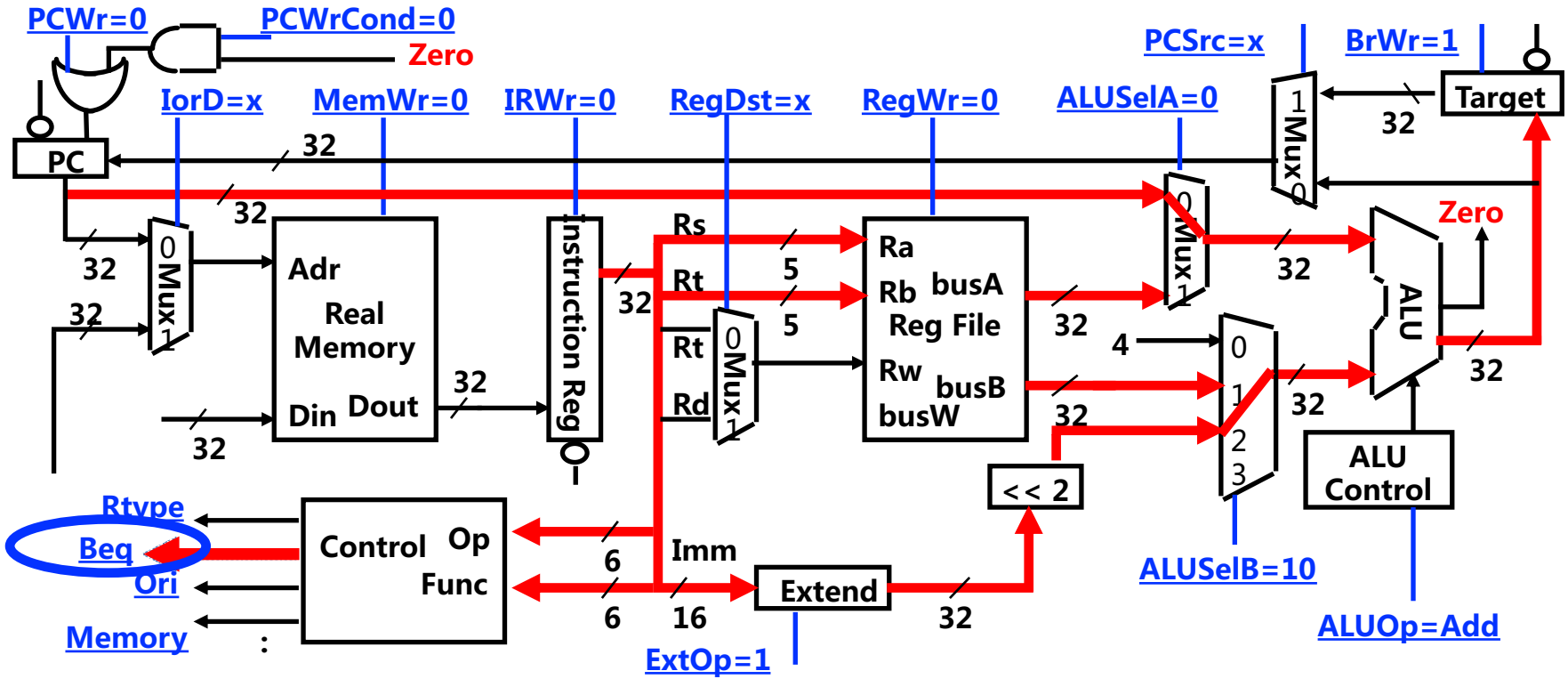
为什么不直接送PC？为什么加BrWr?

控制信号与指令相关吗？



第二周期结束时，执行的结果是什么？

多周期数据通路设计——寄存器取/指令译码周期：第二个周期



如果指令译码输出为：Beq

下面第三个周期就是Beq指令的第一个执行周期！

多周期数据通路设计——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！

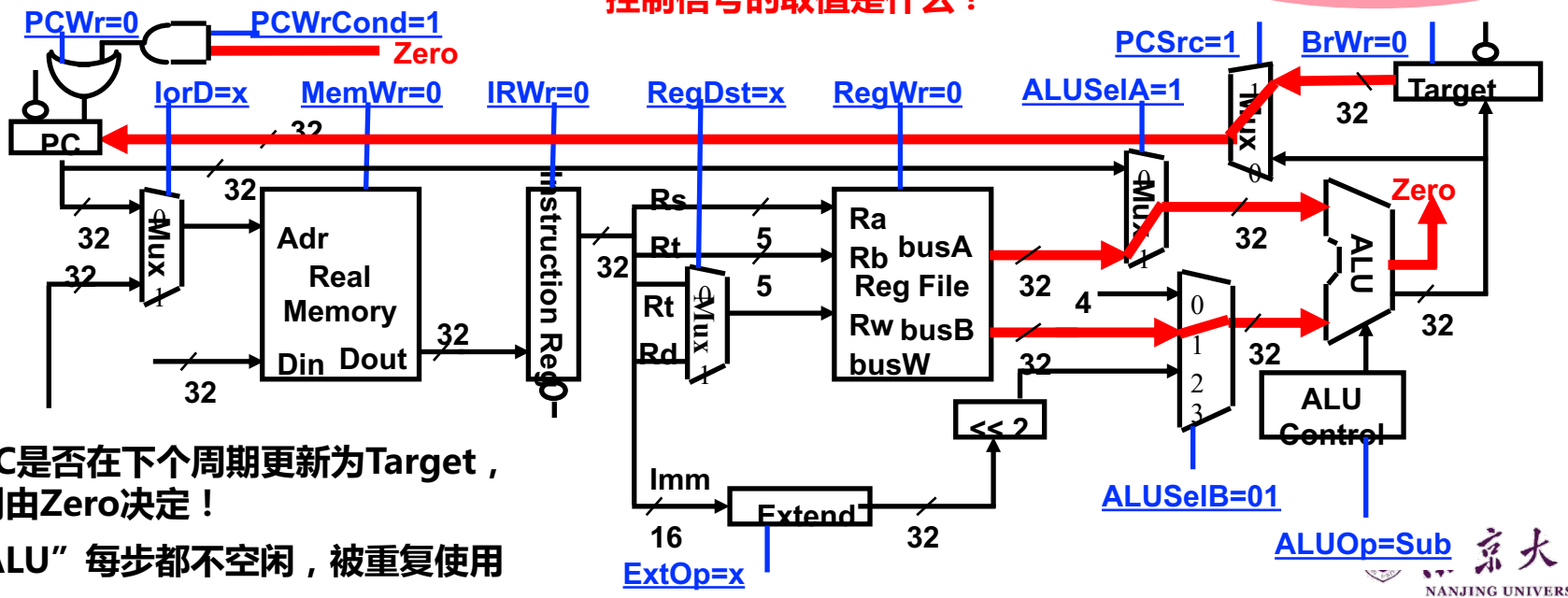
分支执行状态BrFinish

ALUOp=Sub ALUSelB=01

x: IorD, Mem2Reg RegDst, ExtOp

1: PCWrCond ALUSelA PCSrc

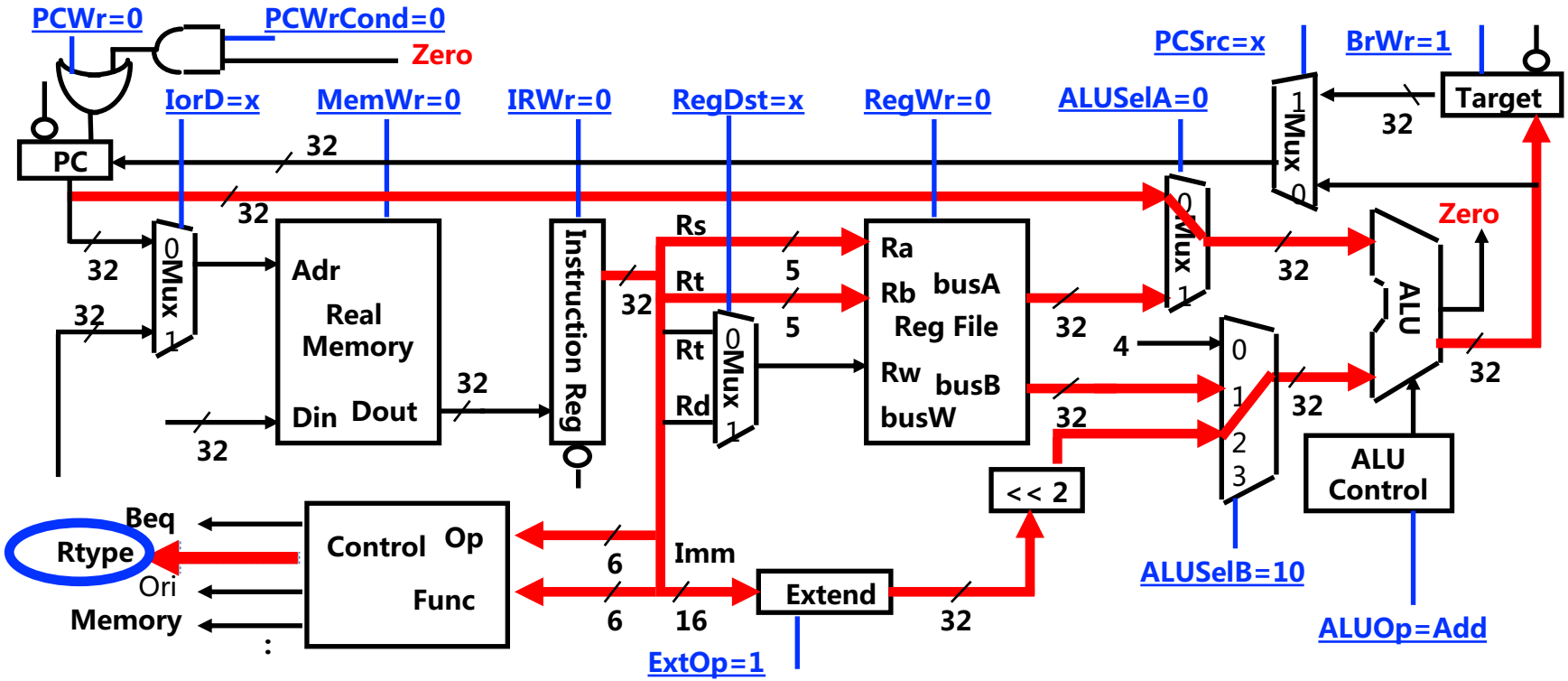
控制信号的取值是什么？



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

多周期数据通路设计——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！

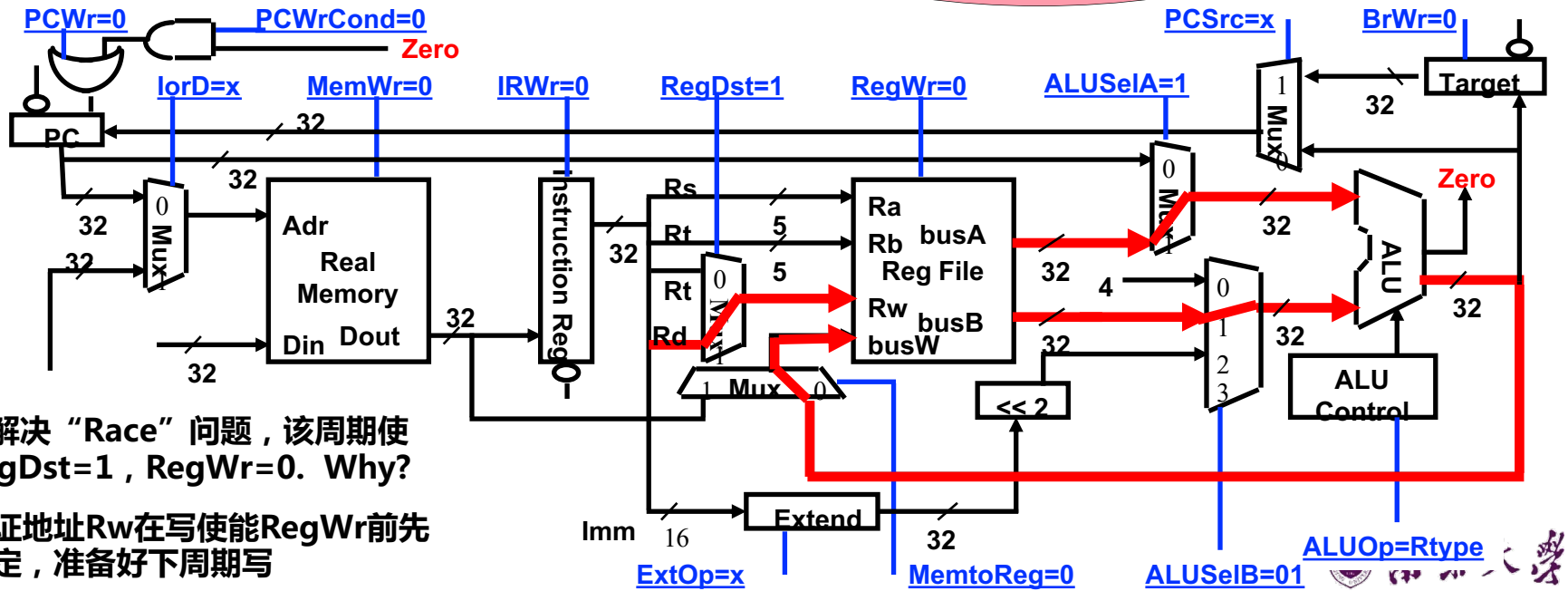
多周期数据通路设计——R-type指令的执行周期：第三个周期

- **ALU Output ← busA op busB**

R-type指令第一个周期控制信号取值？

1: RegDst ALUSelA
ALUSelB=01 ALUOp=Rtype
x: PCSrc, IorD ExtOp 0: MemtoReg

R型指令执行状态RExec



为解决“Race”问题，该周期使RegDst=1，RegWr=0. Why?

保证地址Rw在写使能RegWr前先稳定，准备好下周期写

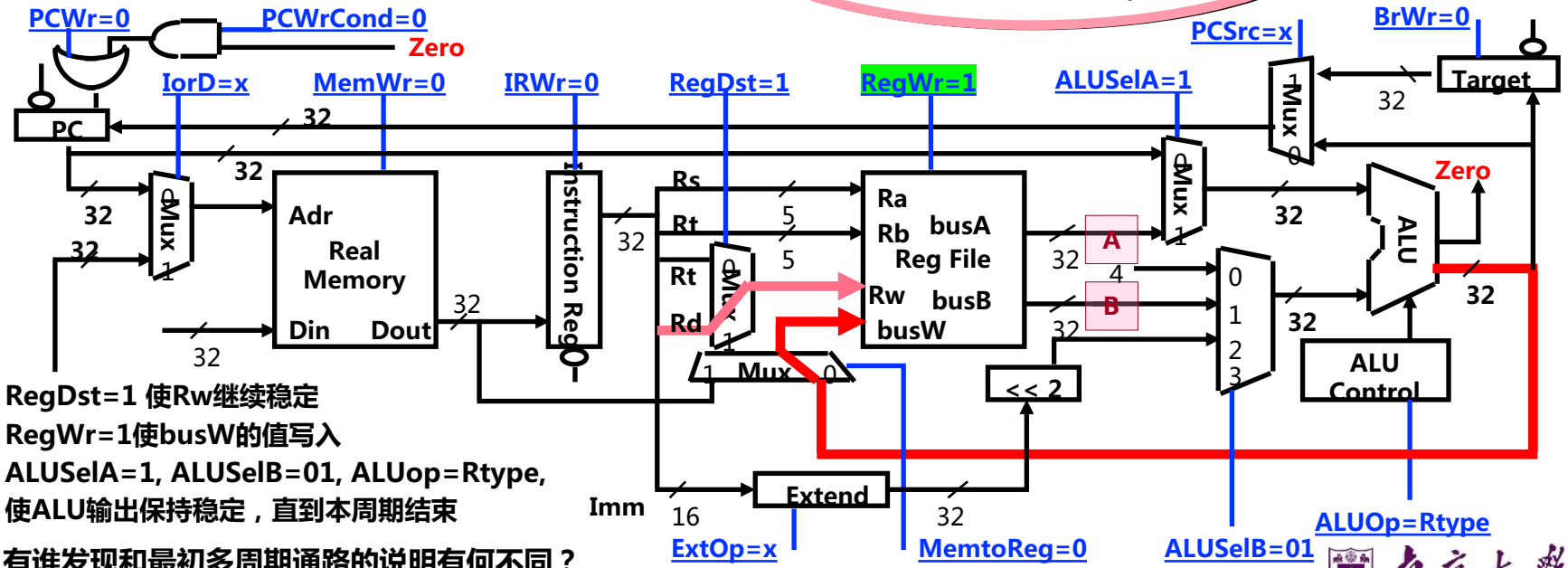
多周期数据通路设计——R-type完成周期：第四个周期

- R[rd] ← ALU Output

R-type指令第二个周期控制信号取值？

ALUOp=Rtype　1: RegDst, RegWr
ALUselA　ALUSelB=01
x: IorD, PCSrc　ExtOp

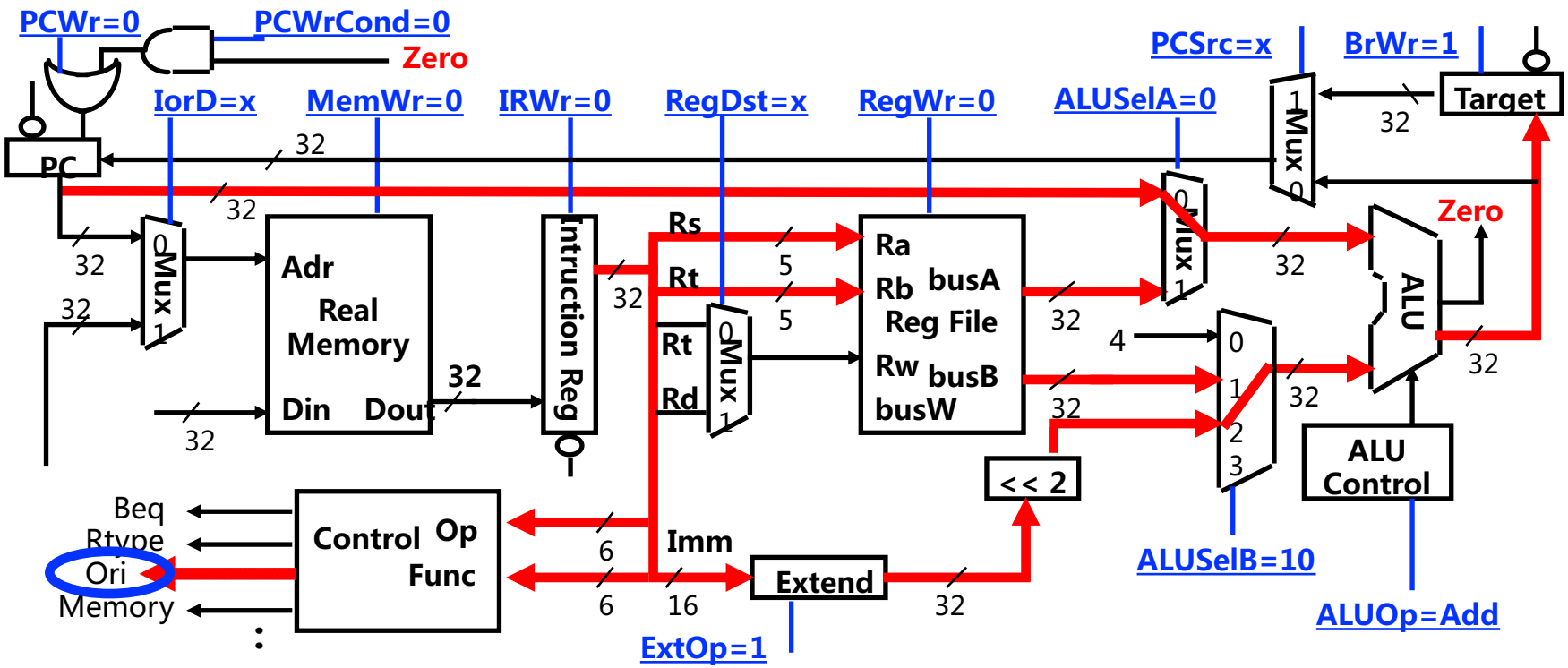
结束状态
Rfinish



RegDst=1 使Rw继续稳定
RegWr=1使busW的值写入
ALUSelA=1, ALUSelB=01, ALUop=Rtype,
使ALU输出保持稳定，直到本周期结束

有谁发现和最初多周期通路的说明有何不同？
这里少了些什么？为什么能少？(A、B可少)

多周期数据通路设计——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！

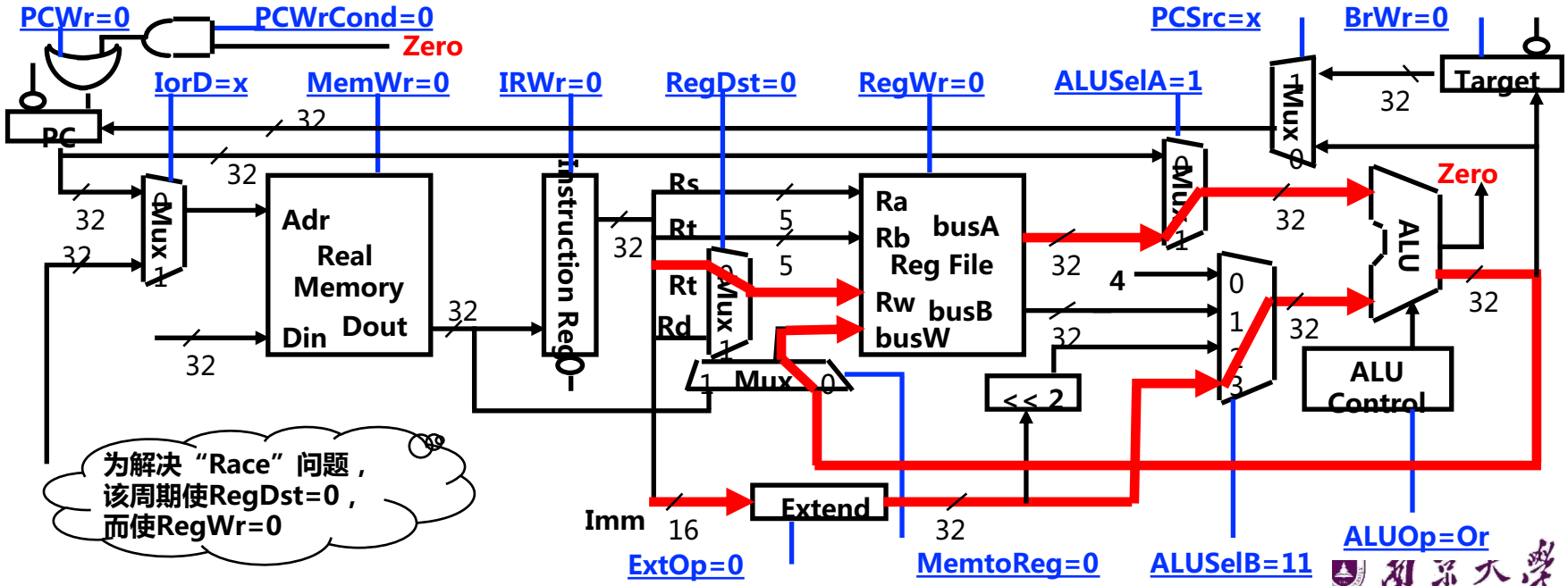
多周期数据通路设计——Ori 指令执行周期：第三个周期

- **ALU output ← busA or ZeroExt[Imm16]**

  ori指令的第一个周期，控制信号取值？

ALUOp=Or　1: ALUSelA
ALUSelB=11　0: MemtoReg
x: IorD, PCSrc

Ori指令执行状态
OriExec



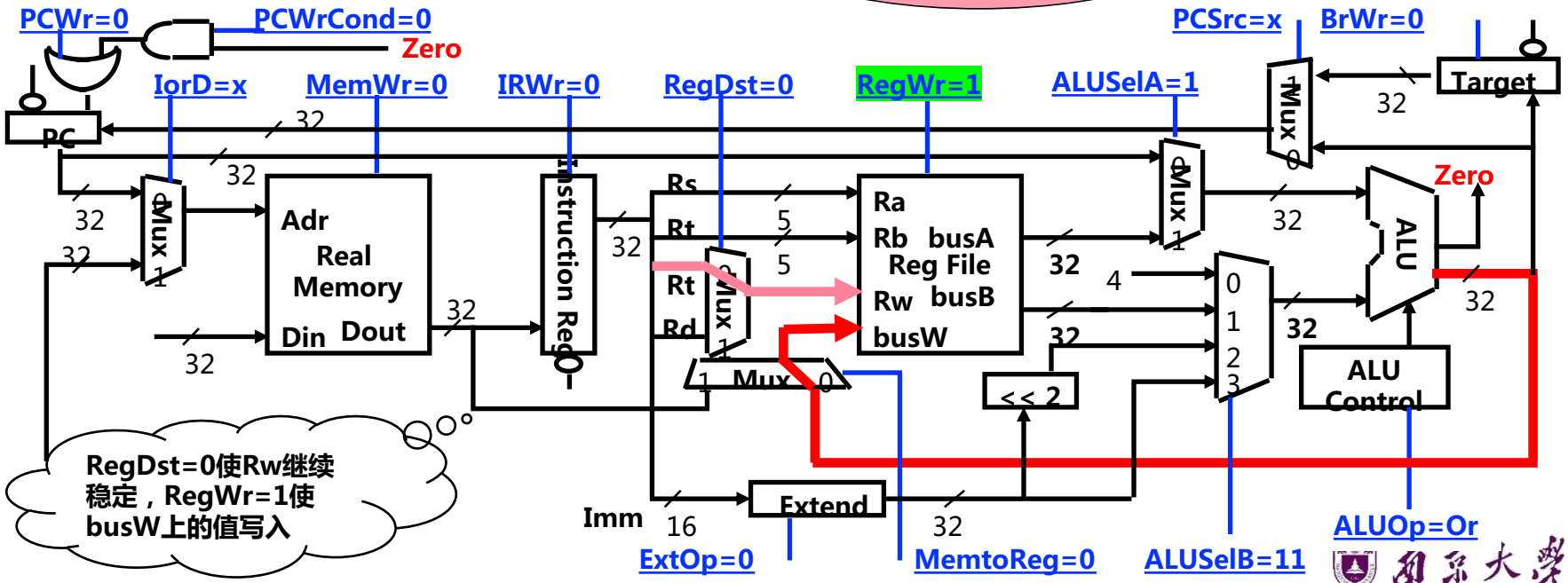
多周期数据通路设计——Ori 指令完成周期：第四个周期

- R [rt] ← ALU output

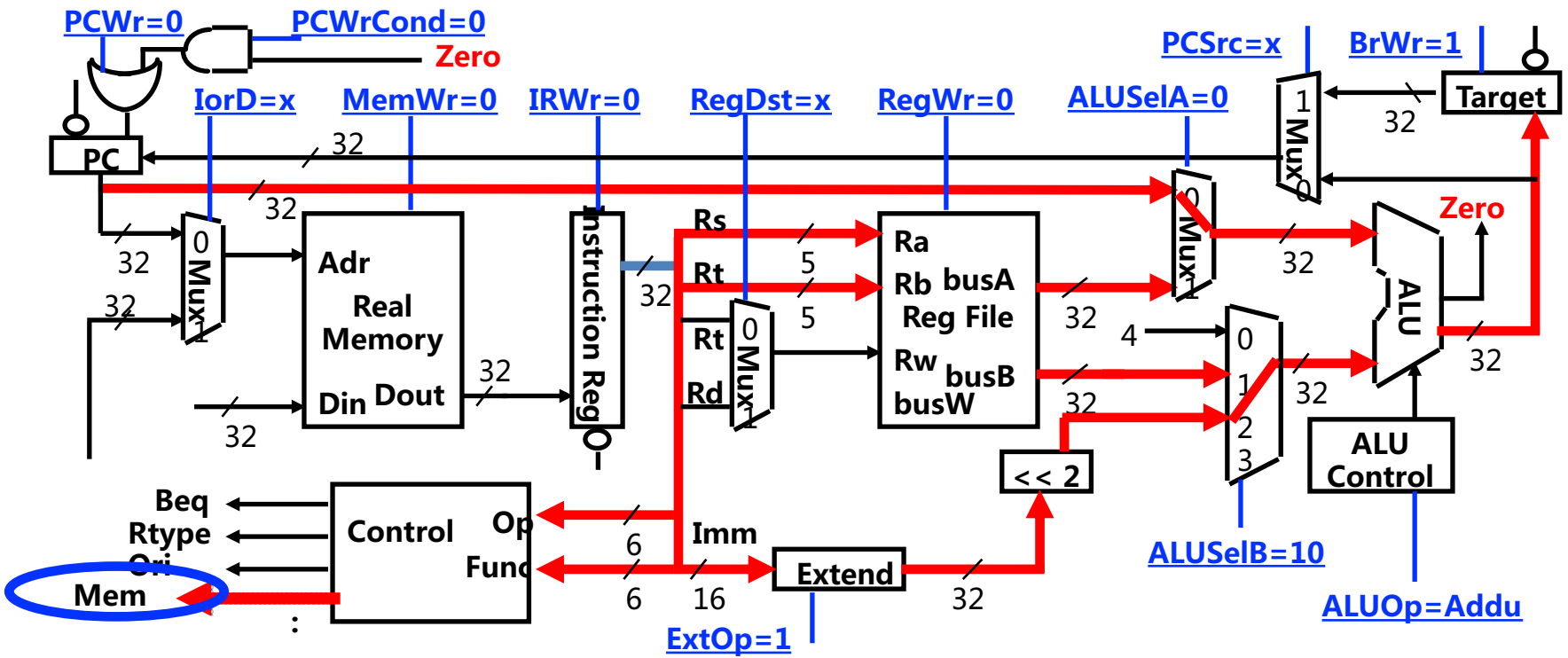
  ori指令的第二个周期控制信号取值？

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

Ori指令结束状态
OriFinish



多周期数据通路设计——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指令的第一个周期！

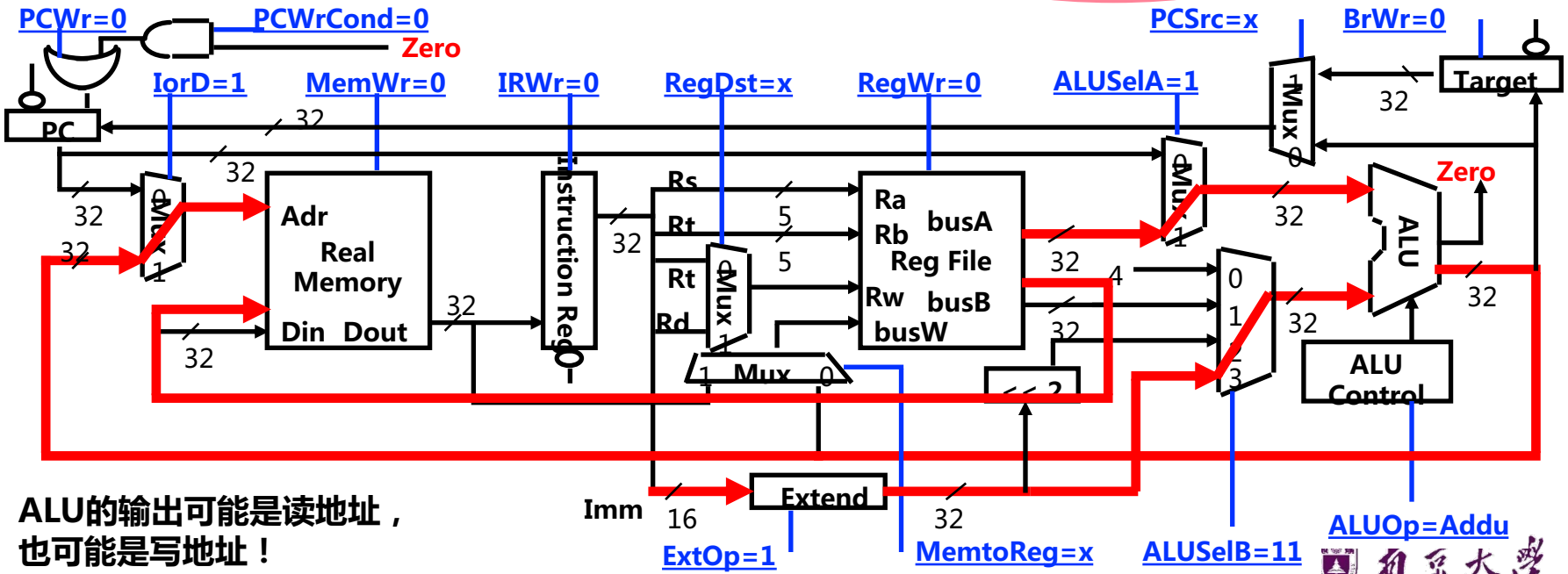
多周期数据通路设计——lw/sw内存地址计算周期：第三个周期

- **ALU output ← busA + SignExt[Imm16]**

  lw/sw指令的第一个周期控制信号取值？

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

MemAdr



ALU的输出可能是读地址，
也可能是写地址！

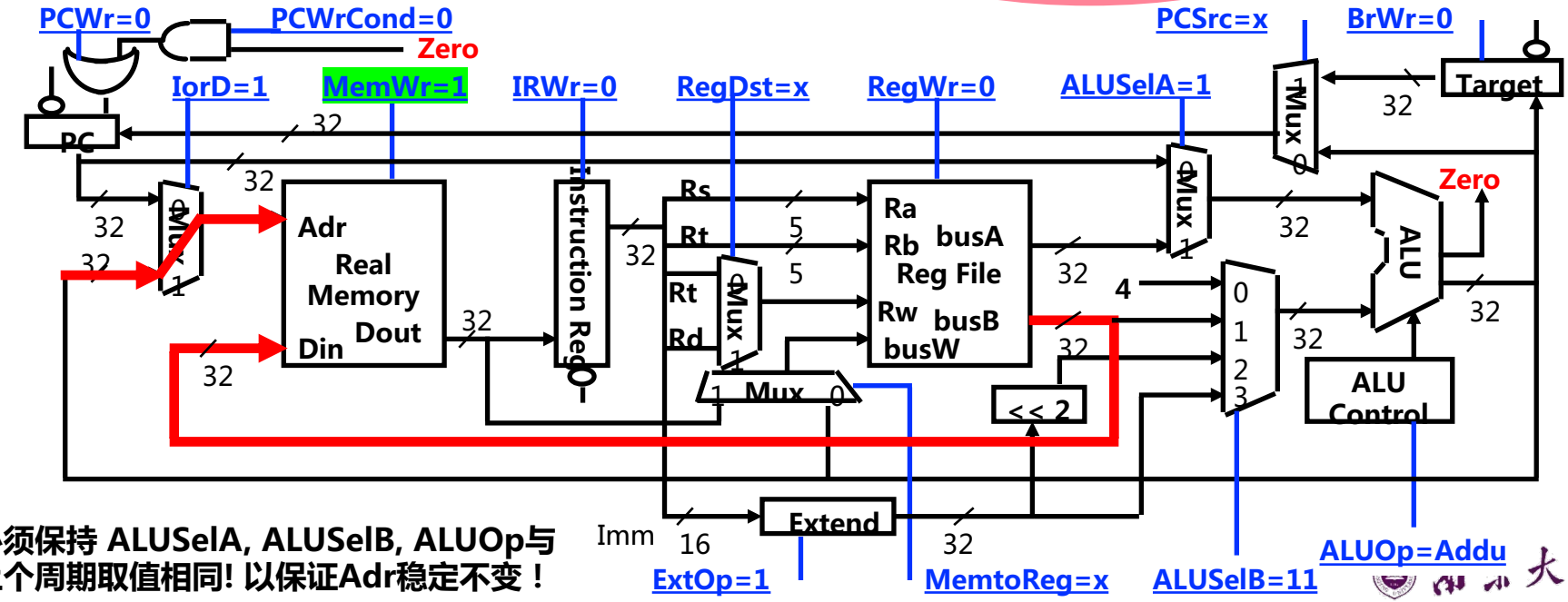
多周期数据通路设计——sw指令存数周期：第四个周期

- **M[ALU output] ← busB**

  sw指令的第二个周期，控制信号取值？

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

swFinish



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

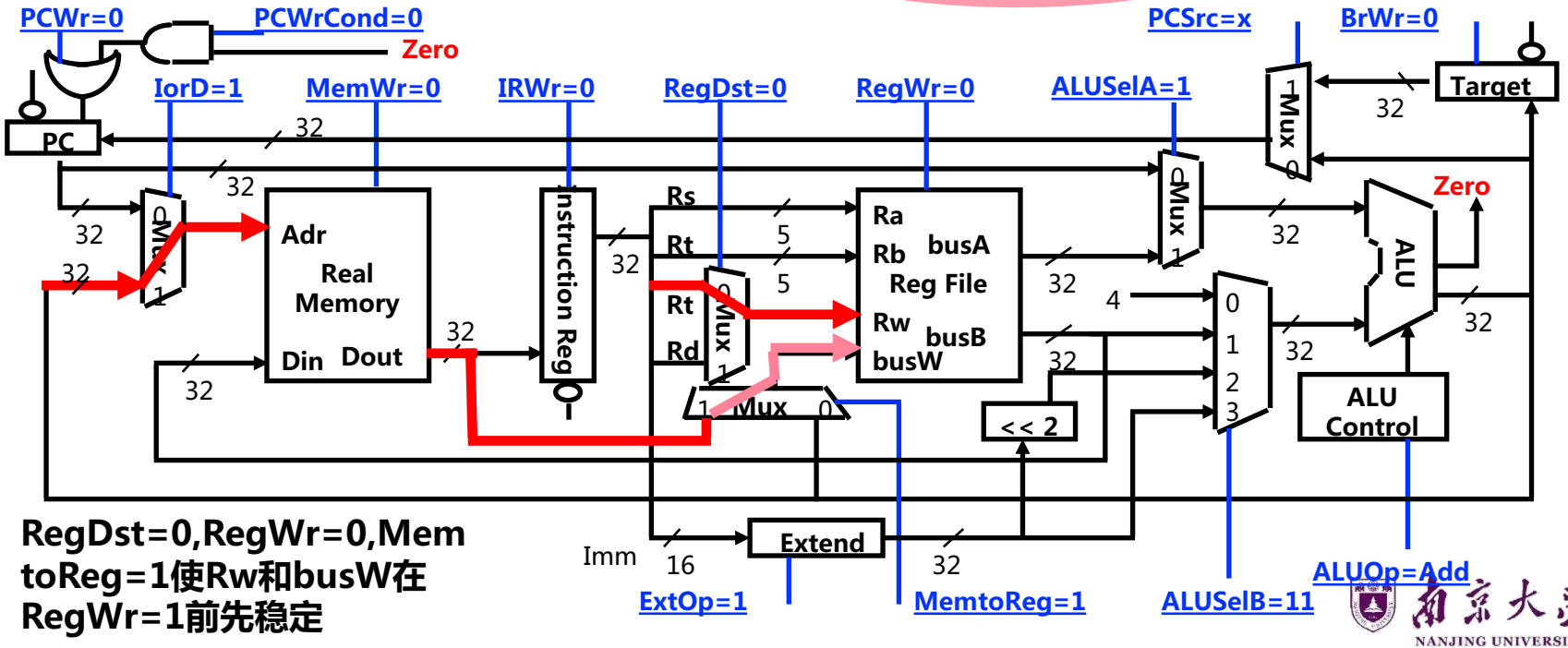
多周期数据通路设计——lw指令取数周期：第四个周期

- **Mem Dout ← M[ALU output]**

lw指令的第二个周期，控制信号取值？

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

MemFetch



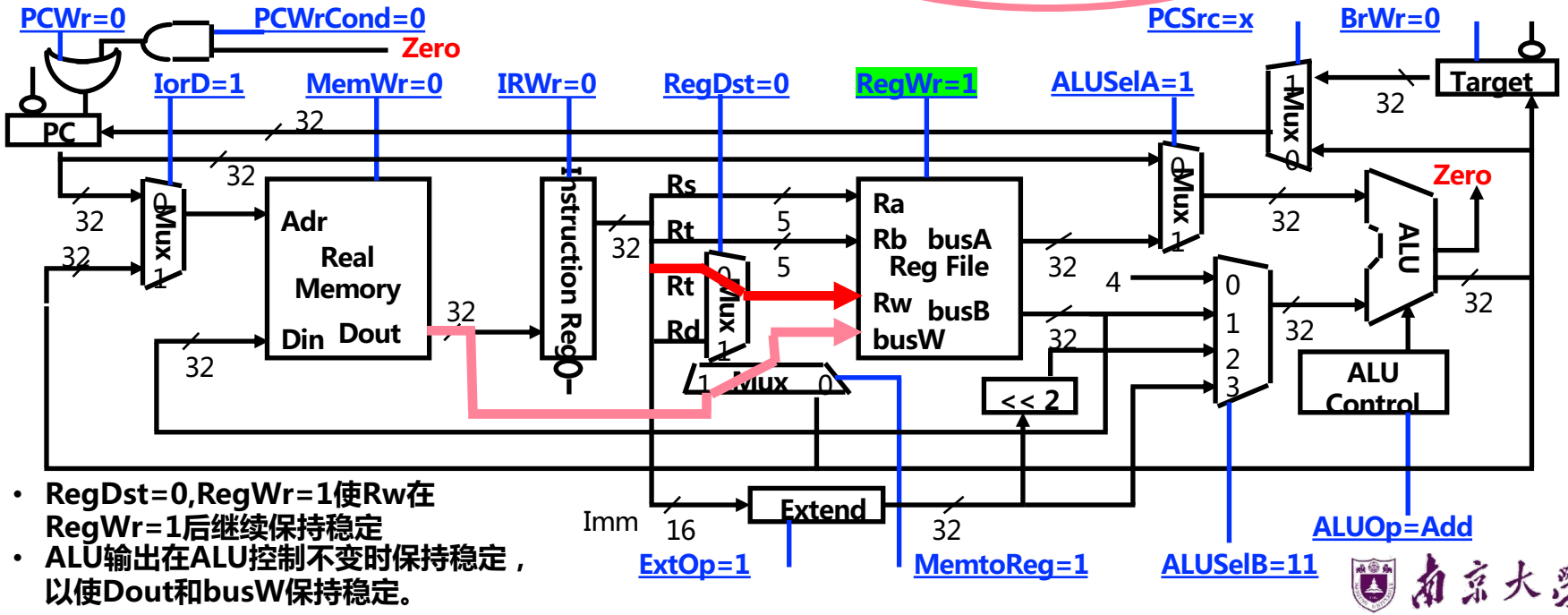
多周期数据通路设计——lw指令回写周期：第五周期

- R[rt] ← Mem Dout

lw指令的第三个周期，控制信号取值？

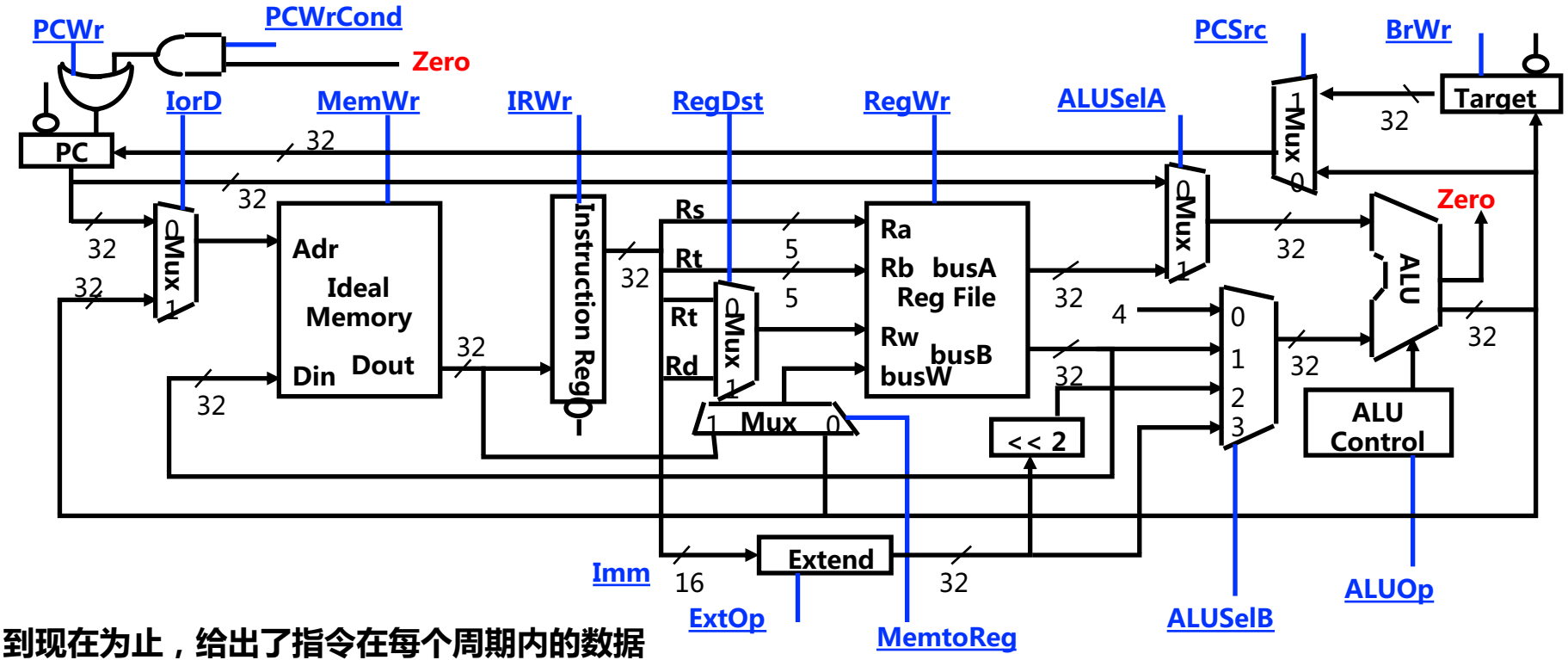
1: ALUSelA　RegWr, ExtOp
MemtoReg　ALUSelB=11
ALUOp=Add　x: PCSrc

lwFinish



- 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。

多周期数据通路设计——完成前述6条指令的完整多周期数据通路



到现在为止，给出了指令在每个周期内的数据流动过程，以及每个周期包含的控制信号取值(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！

多周期数据通路设计——状态转换图

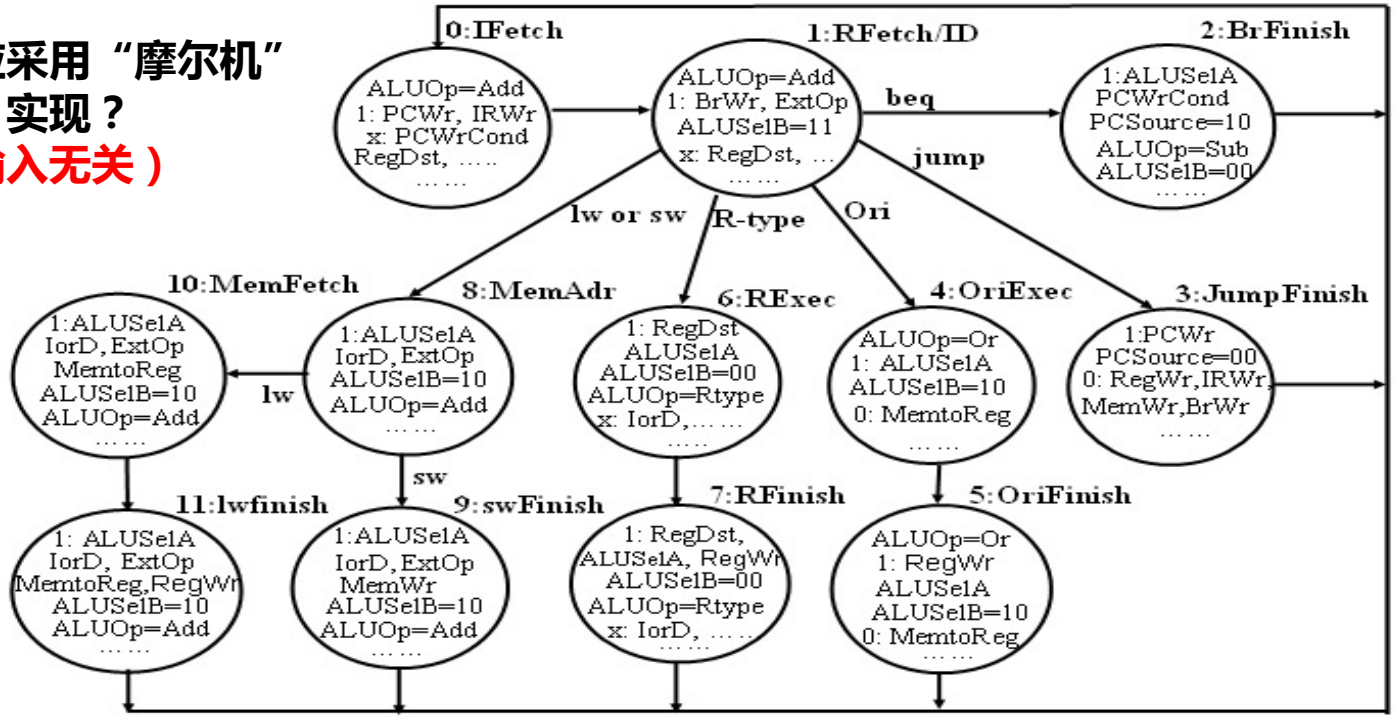
每来一个时钟，进入下一个状态

每个状态下，输出的控制信号有相应的不同取值！

问题：控制器应采用“摩尔机”还是“米利机”实现？
（摩尔机！与输入无关）

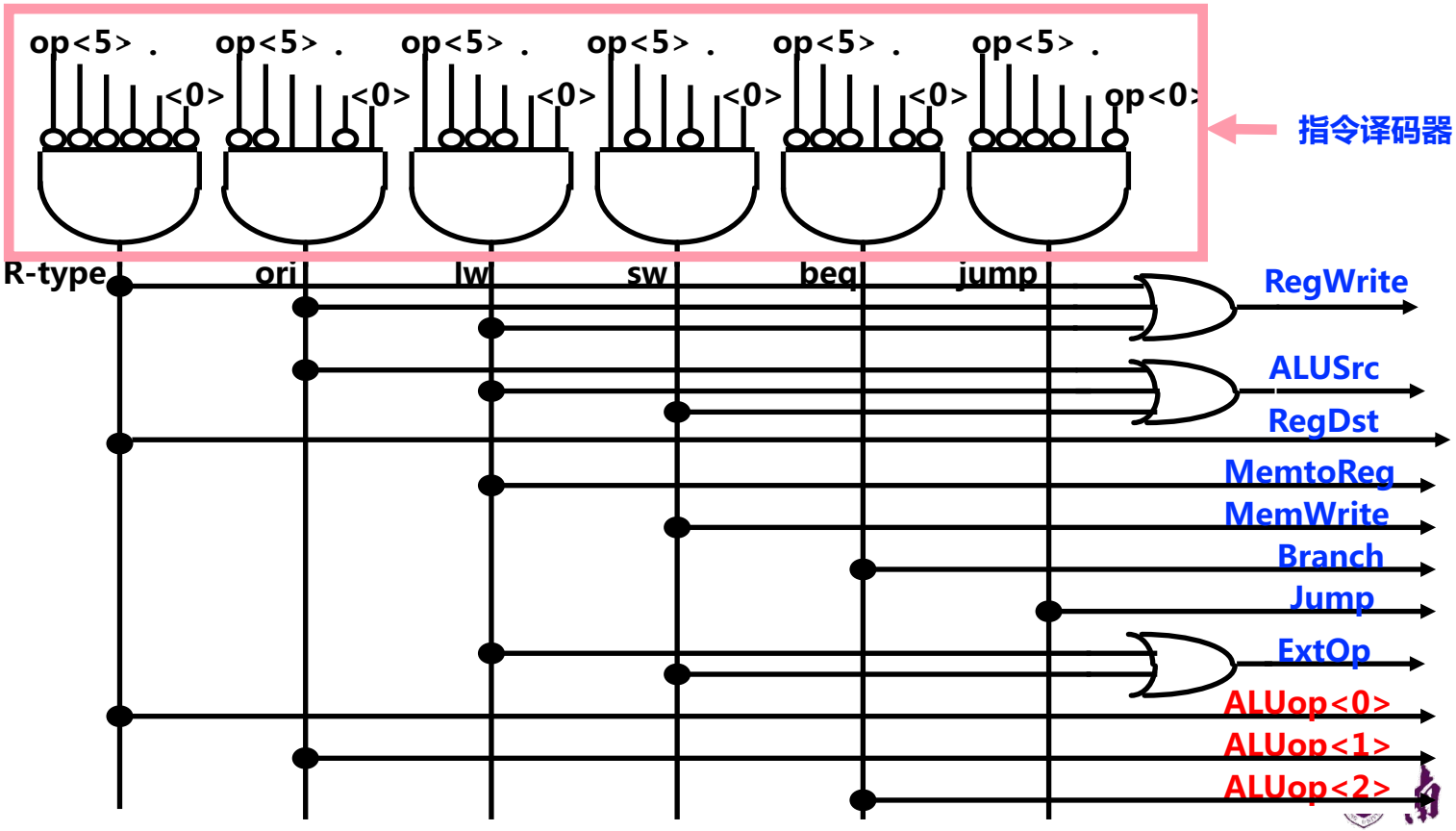
问题：各指令的时钟数多少？

下一步目标：设计“状态转换电路”（控制器）



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

控制器设计——回顾：单周期数据通路

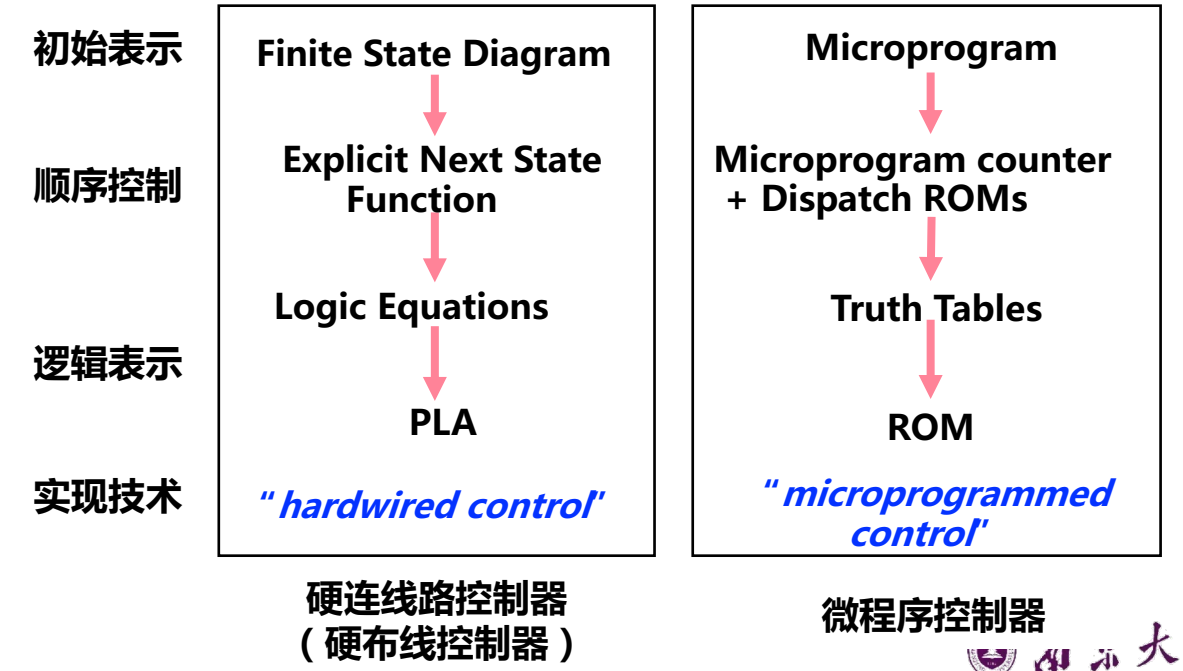


控制器设计——多周期控制器的实现

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？

多周期数据通路的控制更复杂，体现在：每个指令有多个周期，每个周期控制信号取值不同！

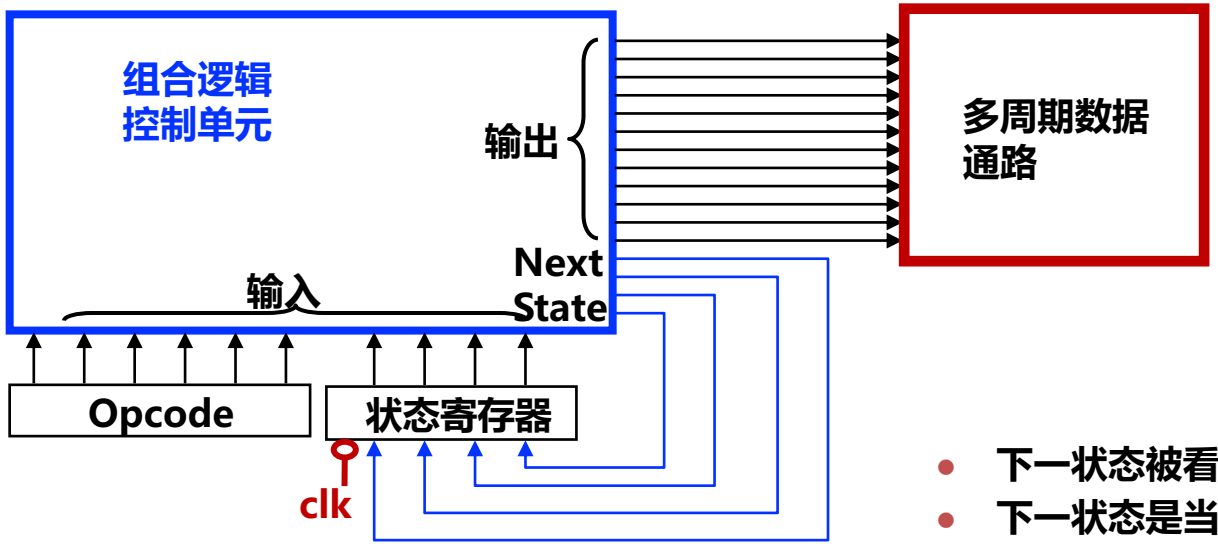
- **多周期控制器功能描述方式：**
  - **有限状态机：用硬连线路(PLA)实现**
  - **微程序：用ROM存放微程序实现**



硬连线控制器设计——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，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。

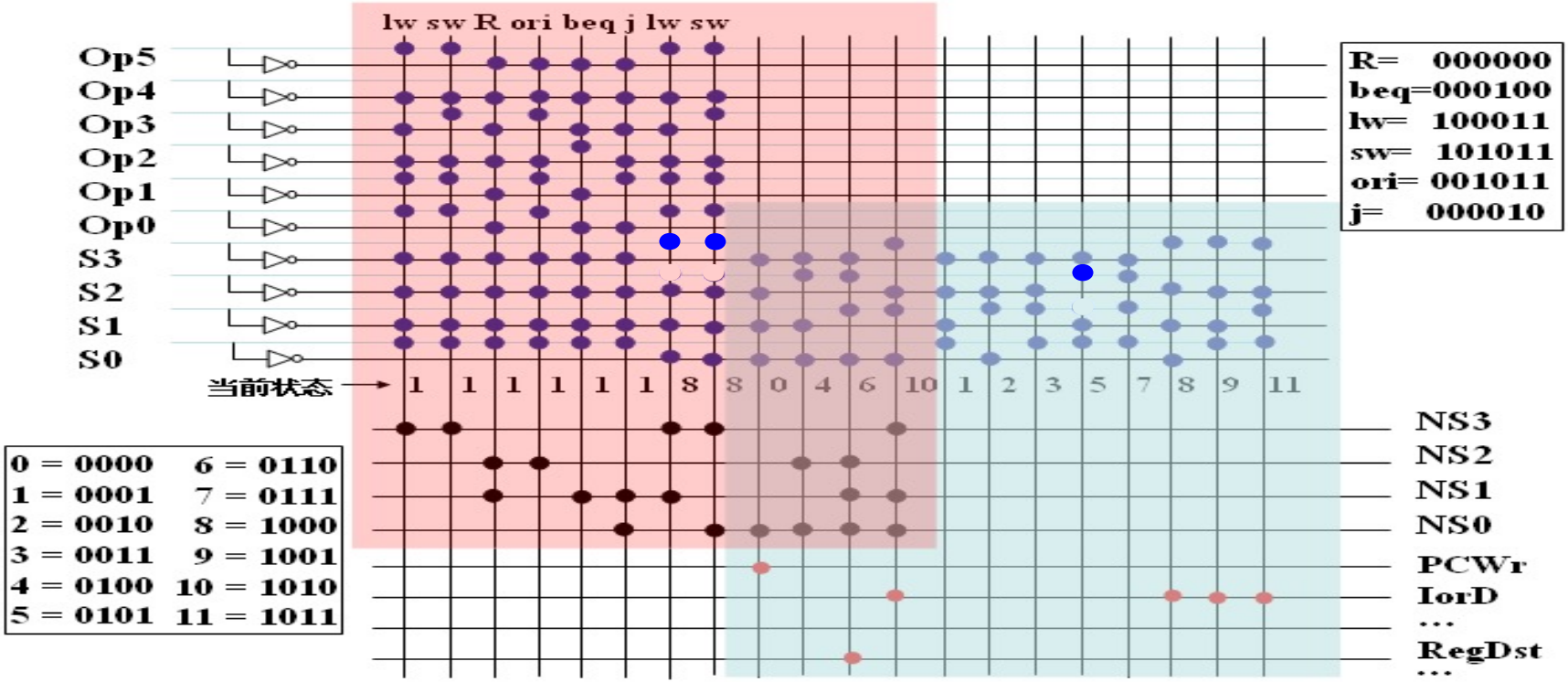
下一步目标：设计控制逻辑（control Logic）

硬连线控制器设计——多周期控制器状态转换表

当前状态 $S_3S_2S_1S_0$	指令操作码$OP_5OP_4OP_3OP_2OP_1OP_0$	下一状态$NS_3NS_2NS_1NS_0$
State2、3、5、7、9、11		0 0 0 0
State0 (IFetch)		0 0 0 1
State1 (ID/RFetch)	000100 (beq)	0 0 1 0
State1 (ID/RFetch)	000010 (jump)	0 0 1 1
State1 (ID/RFetch)	001101 (ori)	0 1 0 0
State4 (OriExec)		0 1 0 1
State1 (ID/RFetch)	000000 (R-type)	0 1 1 0
State6 (RExec)		0 1 1 1
State1 (ID/RFetch)	100011 (lw)	1 0 0 0
State1 (ID/RFetch)	101011 (sw)	1 0 0 0
State8 (MemAdr)	101011 (sw)	1 0 0 1
State8 (MemAdr)	100011 (lw)	1 0 1 0
State10 (MemFetch)		1 0 1 1

以上功能可以由PLA电路来实现！

硬连线控制器设计——PLA组合逻辑控制单元(硬布线方式)



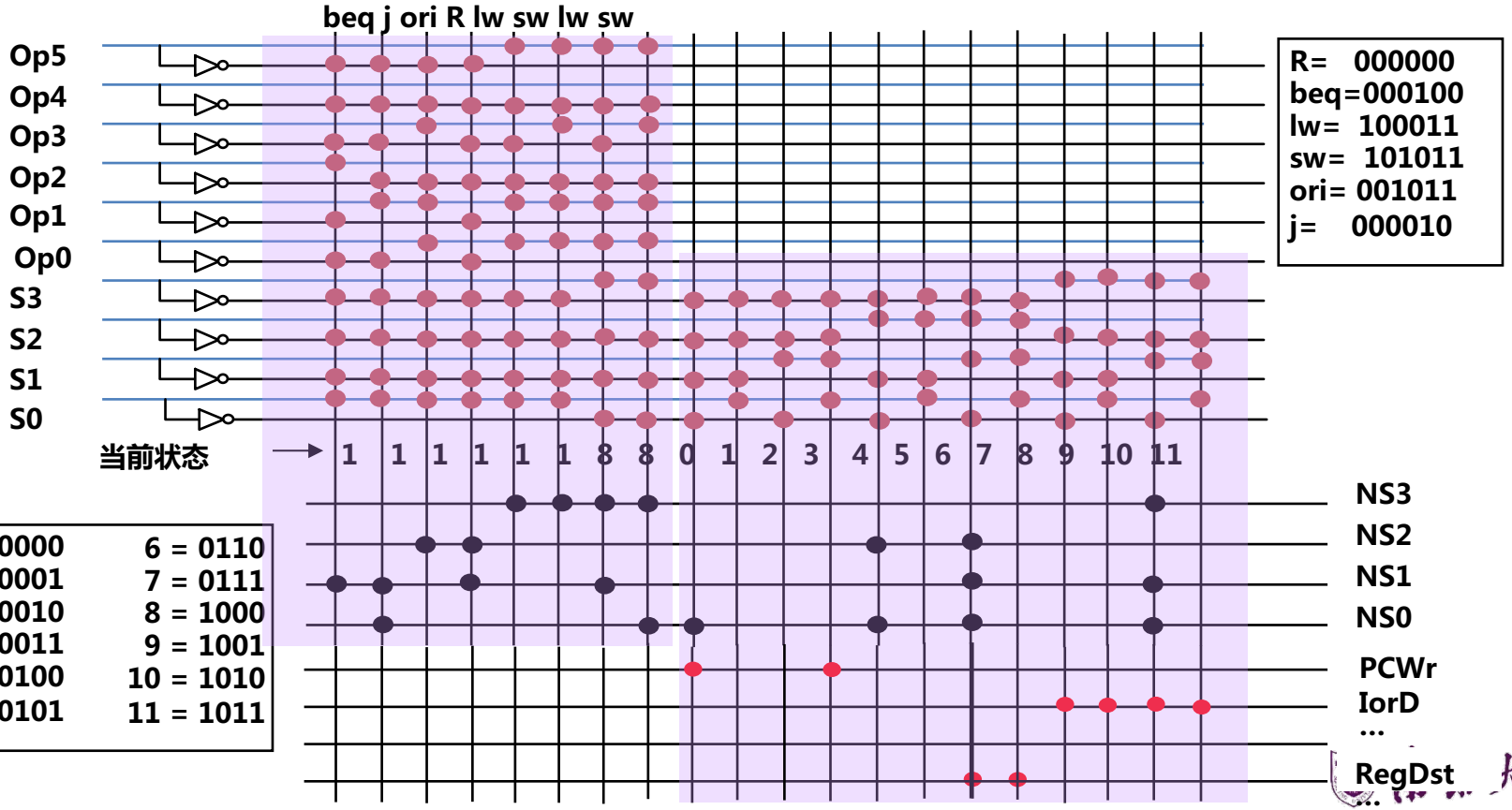
左上角：由操作码和当前状态确定下一状态的电路

右下角：由当前状态确定控制信号的电路

你能找出图中的错误吗？

有三个点的位置不对！已改正！

硬连线控制器设计——PLA组合逻辑控制单元(另一种布局)



微程序控制器设计

硬连线路设计的特点：

- **优点**：速度快，适合于简单或规整的指令系统，例如，MIPS指令集。
- **缺点**：它是一个多输入/多输出的巨大逻辑网络。对于复杂指令系统来说，结构庞杂，实现困难；修改、维护不易；灵活性差。甚至无法用有限状态机描述！

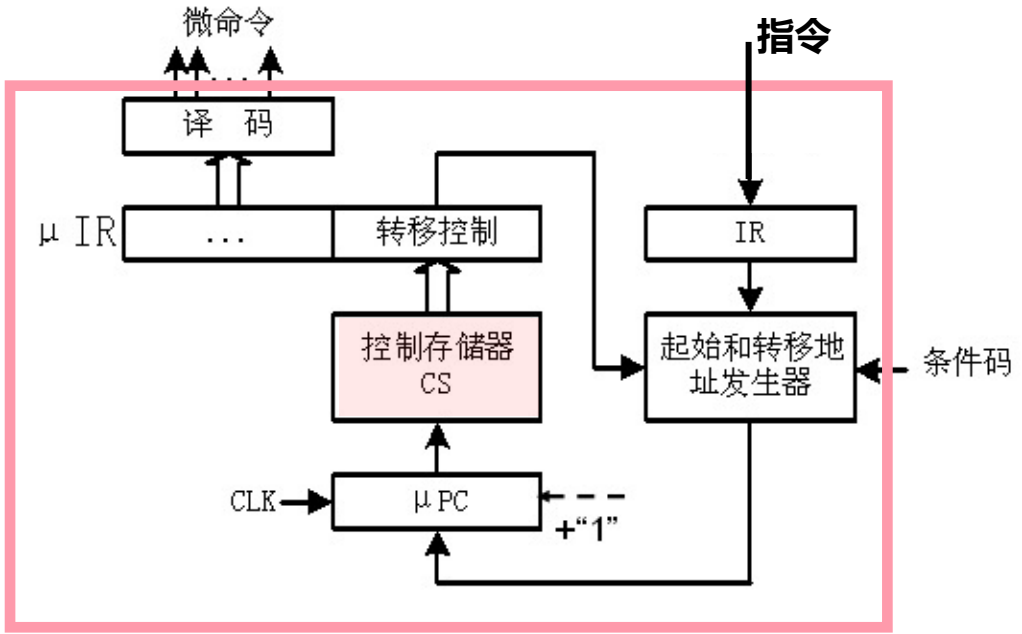
简化控制器设计的一个方法：微程序设计

微程序控制器的基本思想：

- 仿照程序设计的方法，编制每个指令对应的微程序
- **每个微程序由若干条微指令构成，各微指令包含若干条微命令**（一条微指令相当于一个状态，一个微命令就是状态中的控制信号）
- 所有指令对应的微程序放在只读存储器中，执行某条指令时，取出对应微程序中的各条微指令，**对微指令译码产生对应的微命令，这个微命令就是控制信号**。这个只读存储器称为控制存储器（Control Storage），简称控存CS 。

微程序控制器设计——微程序控制器的基本结构

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

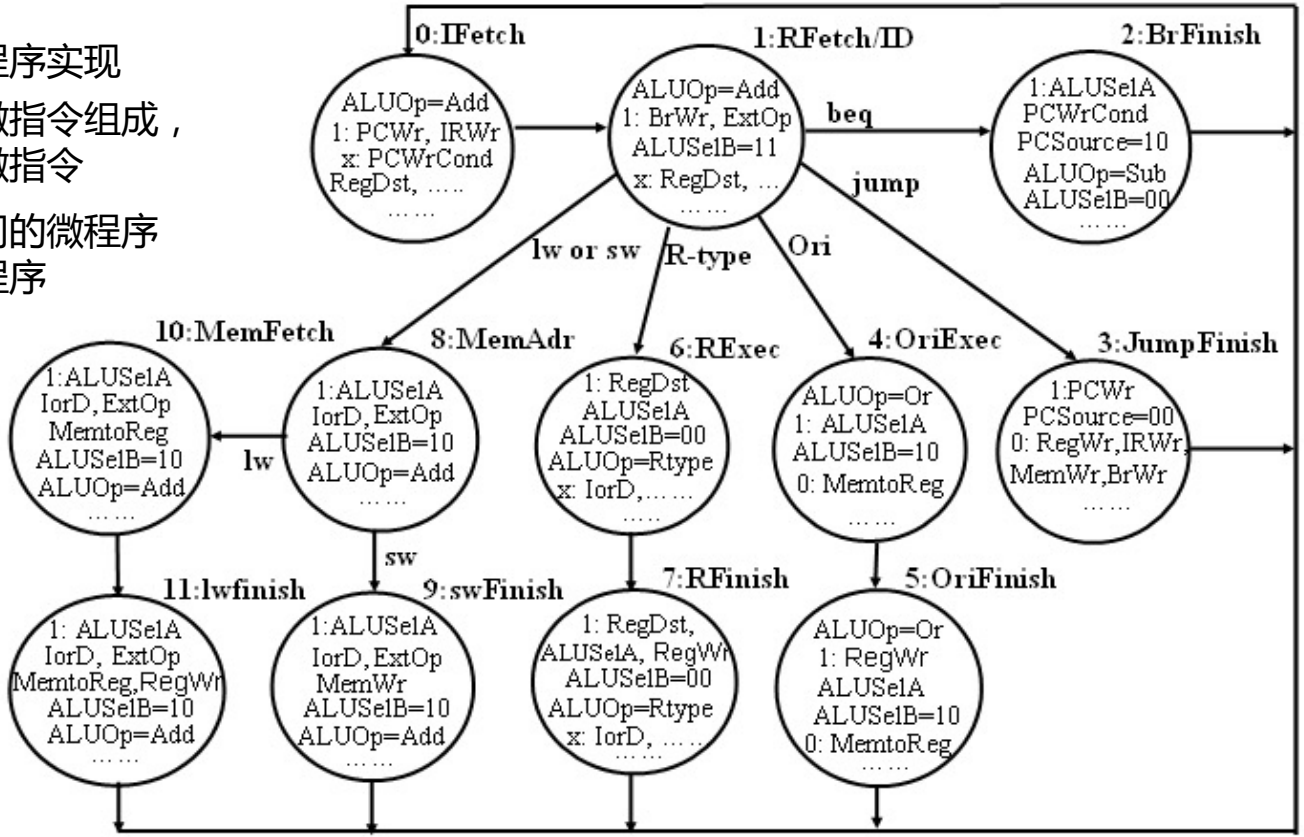


最初把固化在ROM的微程序称固件（Firmware），表示用软件实现的硬部件，现在对固件通俗的理解是在ROM中“固化的软件”。

微程序控制器设计——状态和微程序的对应关系

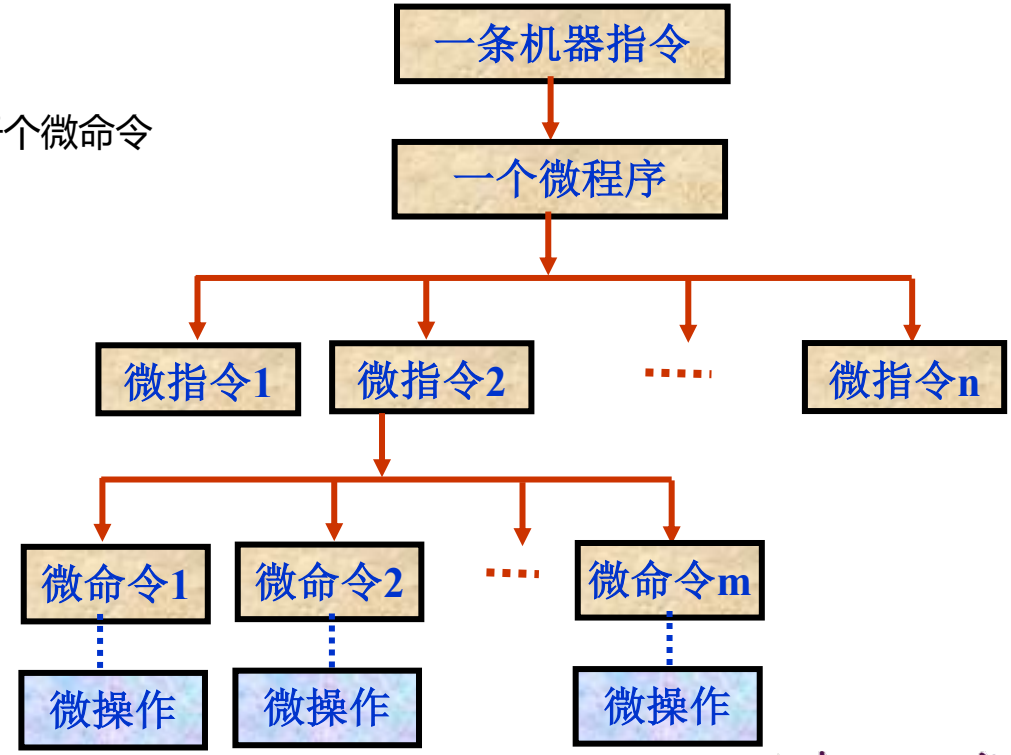
- 每条指令用一个微程序实现
- 每个微程序由若干微指令组成，每个状态对应一条微指令
- 取指令和译码用专门的微程序实现，称为取指微程序

- 上述取指微程序包含几条微指令？（**2条**）
- lw指令有几条微指令？（**3条**）



微程序控制器设计——微程序\微指令\微命令\微操作的关系

- 将指令的执行转换为微程序的执行
- 微程序是一个微指令序列
- 每条微指令是一个0/1序列，其中包含若干个微命令（即：控制信号）
- 微命令控制数据通路的执行



- **控制程序执行要解决什么问题？**

(1) 指令如何译码、执行

(2) 下条指令到哪里去取

- **微程序执行也要解决两个问题：**

(1)微指令如何对微命令编码

(2)下条微指令在哪里

微程序控制器设计——微指令格式的设计

- **水平型微指令**
  - 基本思想：相容微命令尽量多地安排在一条微指令中。
  - 优点：微程序短，并行性高，适合于较高速度的场合。
  - 缺点：微指令长，编码空间利用率较低，并且编制困难。

- **垂直型微指令**
  - 基本思想：一条微指令只控制一、二个微命令。
  - 优点：微指令短，编码效率高，格式与机器指令类似，故编制容易。
  - 缺点：微程序长，一条微指令只能控制一、二个，无并行，速度慢。

- **垂直型微指令面向算法描述，水平型微指令面向内部控制逻辑描述**

微程序控制器设计——微指令格式的设计

- **微指令中包含了：若干微命令、下条微指令地址（可选）、常数（可选）**

微指令格式：

- **μOP**: 微操作码字段，产生微命令；
- **μAddr**：微地址码字段，产生下条微指令地址。
- 微指令格式设计风格取决于微操作码的编码方式（微命令：控制信号）
- **微操作码编码方式**：

不译法（直接控制法）

字段直接编码（译）法

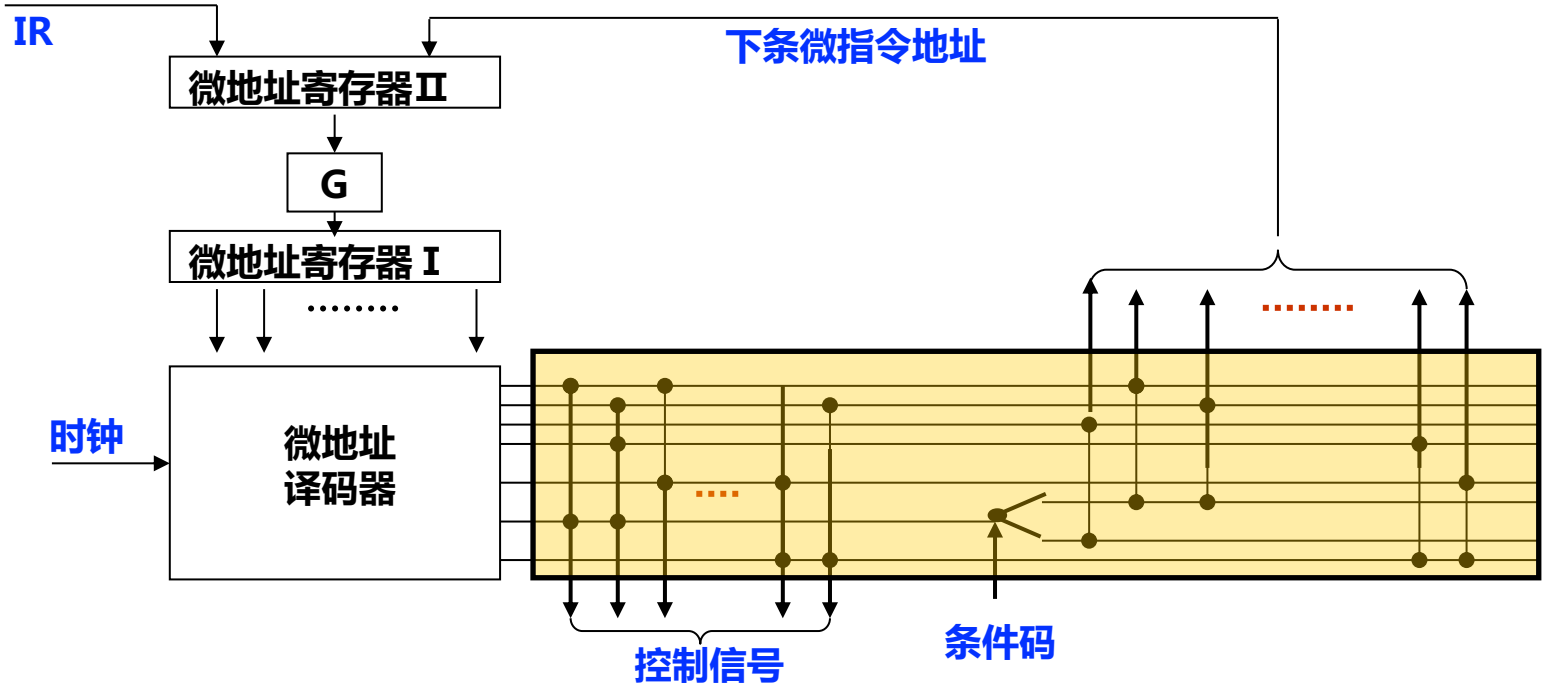
字段间接编码（译）法

} 水平型微指令风格

最小（最短、垂直）编码（译）法 —— 垂直型微指令风格

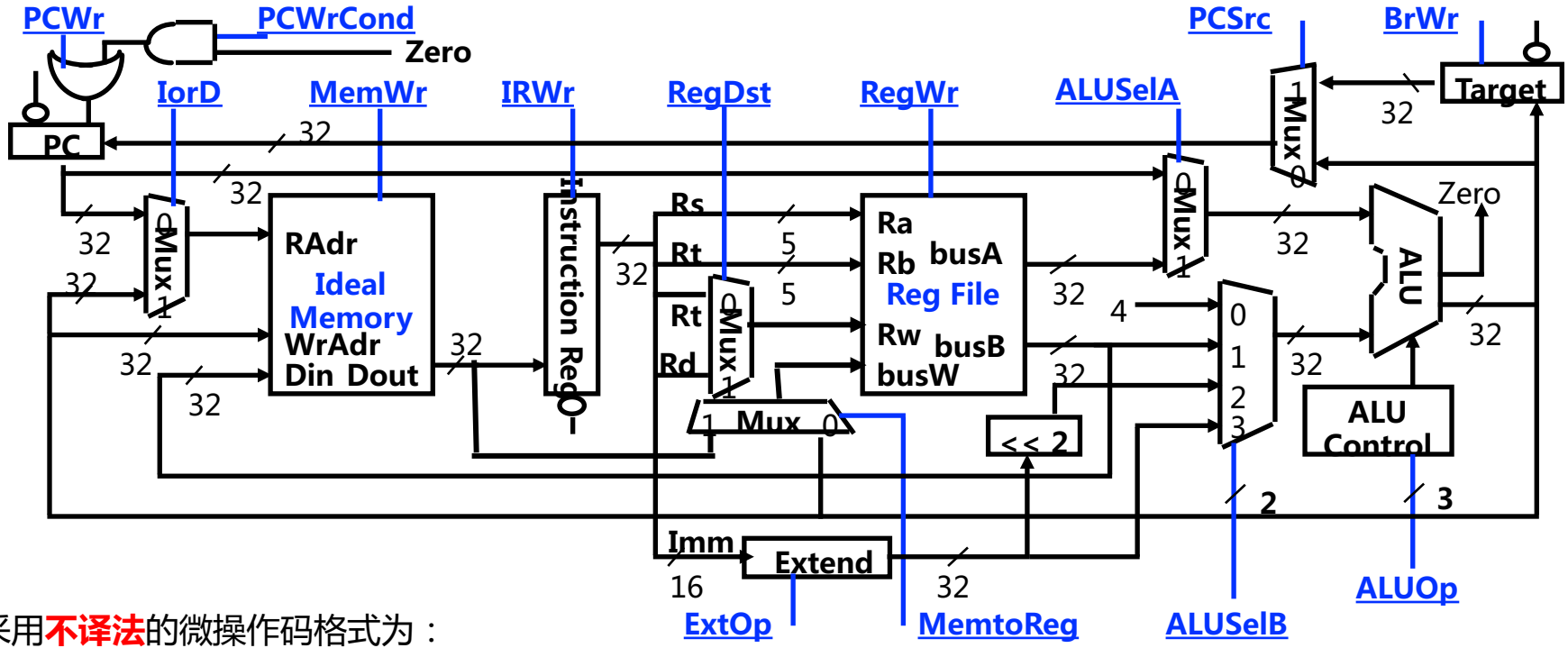
微程序控制器设计——不译法(直接控制法)

- 基本思想： 一位对应一个微命令（控制信号），不需译码。



Wilkes微程序控制器是最早提出的，采用的就是不译法。

微程序控制器设计——多周期数据通路对应的微操作码



采用**不译法**的微操作码格式为：

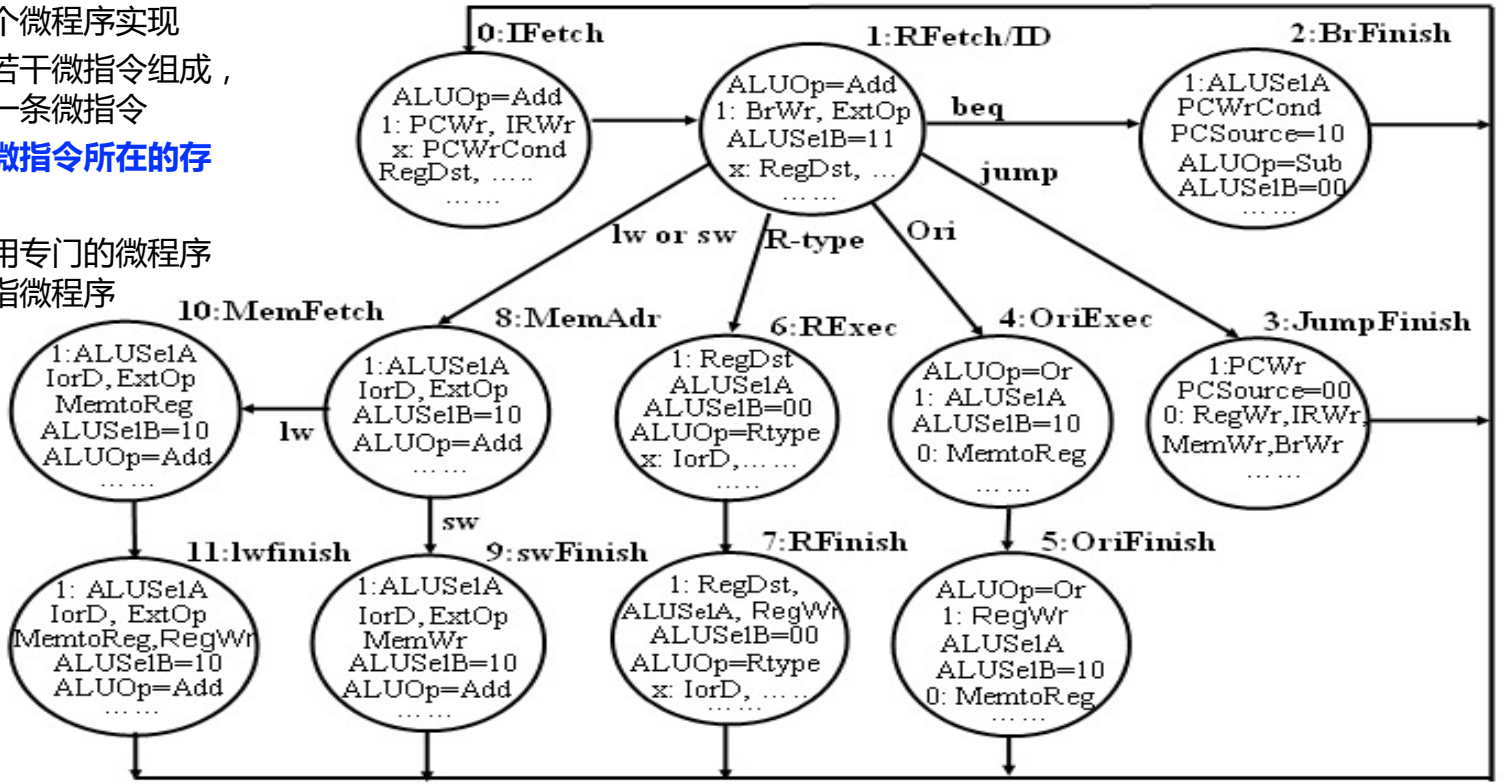
PCWr	IorD	MemWr	.....	PCSrc	BrWr	.....

控制字 (微指令) 长度等于控制信号（微命令）总位数

该图对应控制器的微指令长度为多少位？(17位)

微程序控制器设计——状态和微程序的对应关系

- 每条指令用一个微程序实现
- 每个微程序由若干微指令组成，每个状态对应一条微指令
- **状态号相当于微指令所在的存储单元地址**
- 取指令和译码用专门的微程序实现，称为取指微程序



PCWr	IorD	MemWr	…	PCSrc	BrWr	…

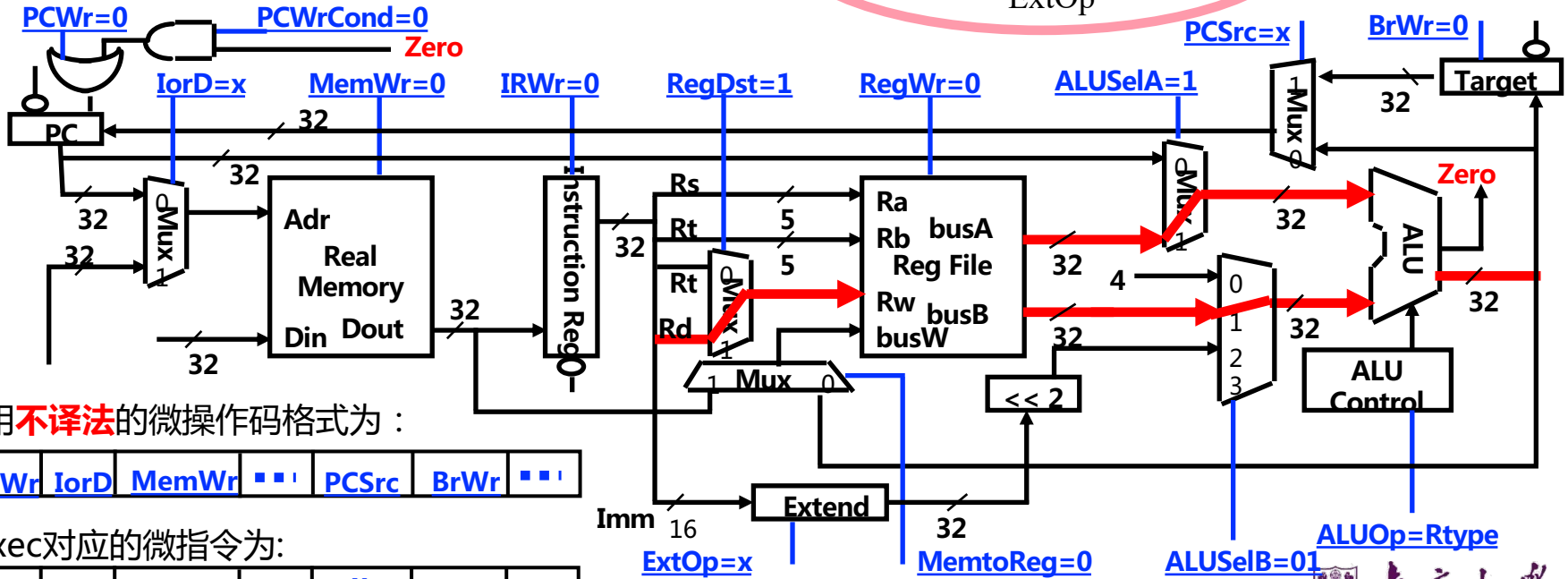
微程序控制器设计——R-type指令的执行周期(第三个周期)

- **ALU Output ← busA op busB**

R-type指令第一个周期控制信号取值？

1: RegDst　　ALUSelA
ALUSelB=01　　ALUOp=Rtype
x: PCSrc, IorD　　MemtoReg
ExtOp

RExec



采用**不译法**的微操作码格式为：

PCWr	IorD	MemWr	▪▪▪	PCSrc	BrWr	▪▪▪

REXec对应的微指令为:

0	x	0	▪▪▪	x	0	▪▪▪

微程序控制器设计——不译法(直接控制法)

- **基本思想**： 一位对应一个微命令（控制信号），不需译码。
- **优点**：
  - 并行控制能力强，且不必译码，故执行速度快。
  - 编制的微程序短。
- **缺点**：
  - 微指令字很长，可能多达几百位。
  - 编码空间利用率低。(几百位中可能只有几位为1)

- **微操作码编码方式**：

不译法（直接控制法）
字段直接编码（译）法
字段间接编码（译）法
} 水平型微指令风格

最小（最短、垂直）编码（译）法 —— 垂直型微指令风格

微程序控制器设计——字段直接编码法

- **基本思想：**
  - 将微指令分成若干字段，每个字段对包含的若干微命令编码
  - 把**互斥微命令**组合在同一字段，**相容微命令**组合在不同字段
  - 一条微指令中最多可同时发出的微命令个数就是字段数
    - **相容微操作：能同时进行的微操作，称为相容的。**
    - **互斥微操作：不能同时进行的微操作，称为互斥的。例如：**
    - **ALU运算(add/sub/or/…)，存储器操作(读指令/读数据/写数据)**
    - **你还能想出哪些互斥微操作？ 从各个寄存器送总线: $R1_{out}$、$R2_{out}$**
- **优点：**
  - 有较高的并行控制能力，速度较快。
  - 微指令短，能压缩到不译法的1/2到1/3，节省控存容量。
- **缺点：**
  - 增加译码线路，并开销一部分时间。但因分段后各字段位数少，所以译码对微指令的执行速度影响不大。

中央处理器

- CPU概述
- 单周期处理器设计
- 多周期处理器设计
- **带异常处理的处理器设计**

带异常处理的处理器设计

- 识别异常事件：软件识别和硬件识别（向量中断）两种不同的方式。

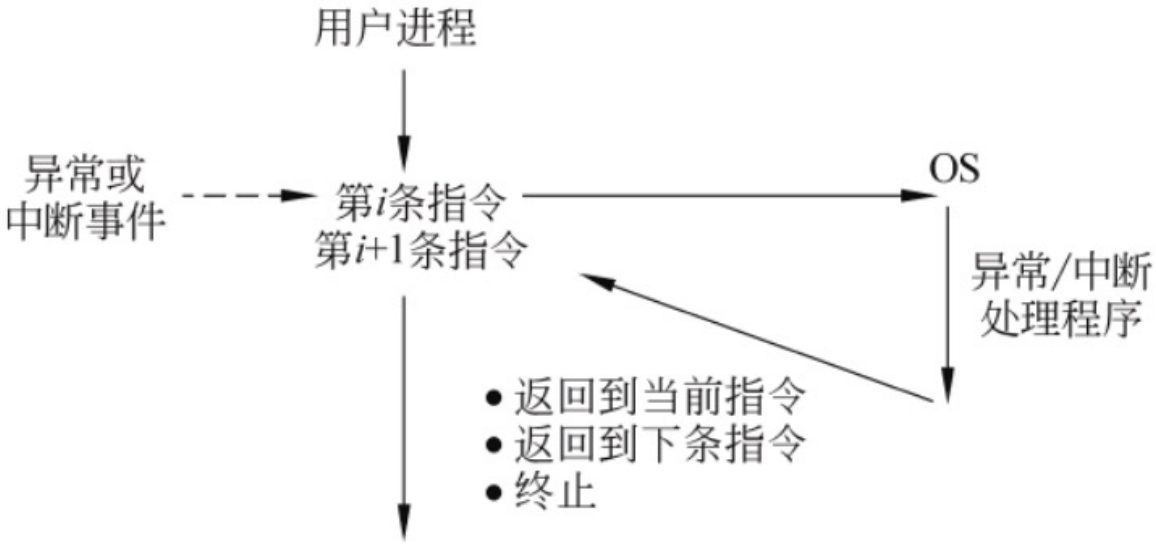


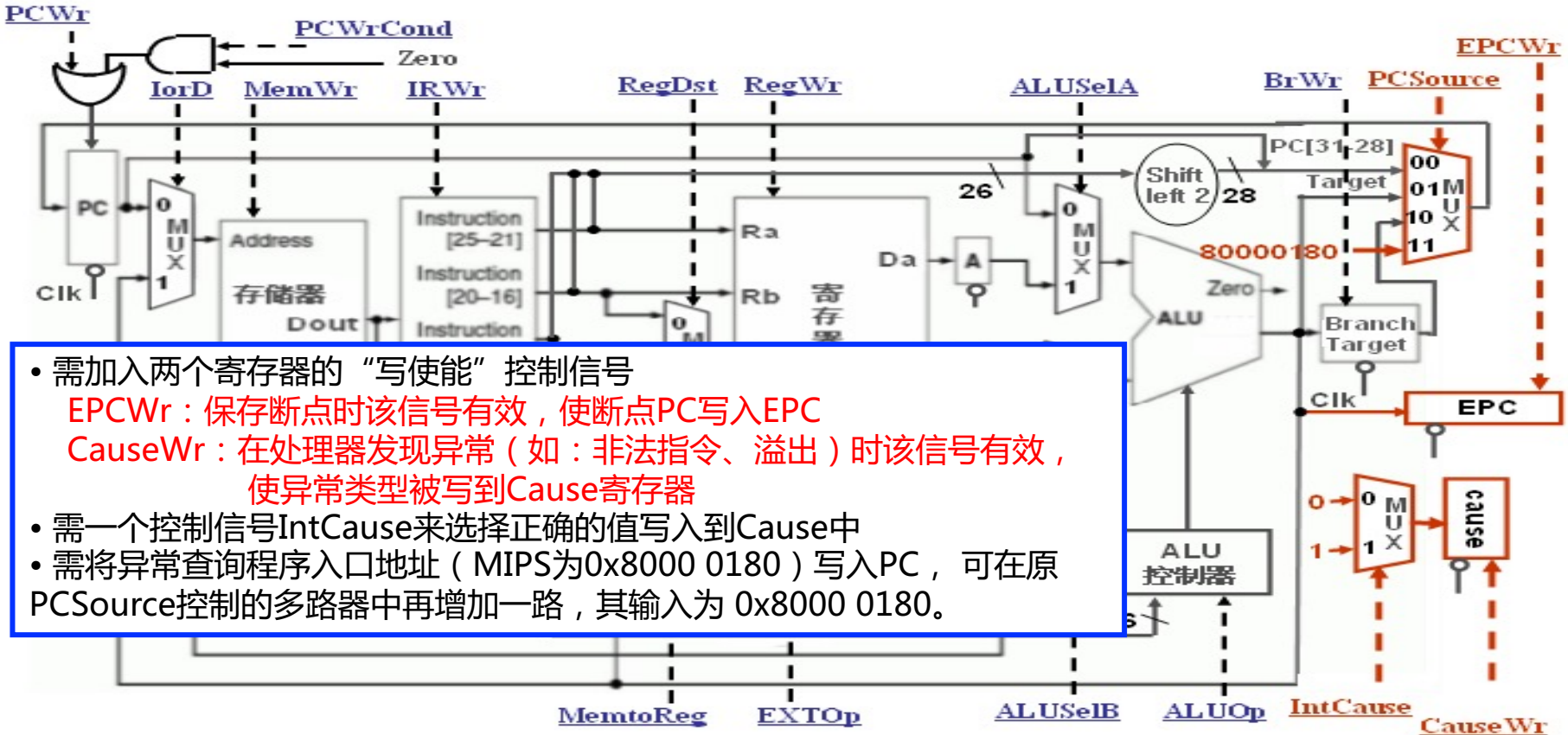
图 5.30　异常和中断处理过程

带异常处理的处理器设计

- MIPS采用软件（操作系统提供的一个特定的异常查询程序）识别中断源
- 数据通路中需增加以下两个寄存器：
  - **EPC**：32位，用于存放断点（异常处理后返回到的指令的地址）。
    - 写入EPC的断点可能是正在执行的指令的地址（故障时），也可能是下条指令的地址（自陷和中断时）。前者需要把PC的值减4后送到EPC，后者则直接送PC到EPC (why?)
  - **Cause**：32位（有些位还没有用到），记录异常原因。
    - 假定处理的异常类型有以下两种：

      未定义指令（Cause=0）、溢出（Cause=1）
- 需要加入两个寄存器的“写使能”控制信号
  - **EPCWr**：在保存断点时该信号有效，使断点PC写入EPC。
  - **CauseWr**：在处理器发现异常（如：非法指令、溢出）时，该信号有效，使异常类型被写到Cause寄存器。
- 需要一个控制信号IntCause来选择正确的值写入到Cause中
- 需要将异常查询程序的入口地址（MIPS为0x8000 0180）写入PC，可以在原来PCSource控制的多路复用器中再增加一路，其输入为0x8000 0180

带异常处理的数据通路

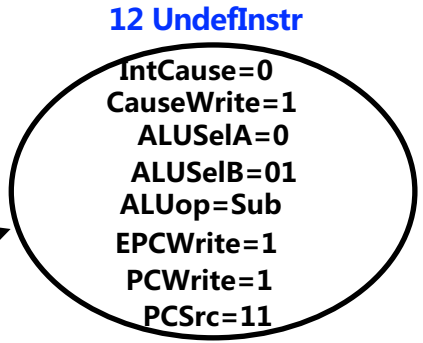


- 需加入两个寄存器的“写使能”控制信号
  EPCWr：保存断点时该信号有效，使断点PC写入EPC
  CauseWr：在处理器发现异常（如：非法指令、溢出）时该信号有效，使异常类型被写到Cause寄存器
- 需一个控制信号IntCause来选择正确的值写入到Cause中
- 需将异常查询程序入口地址（MIPS为0x8000 0180）写入PC， 可在原PCSource控制的多路器中再增加一路，其输入为 0x8000 0180。

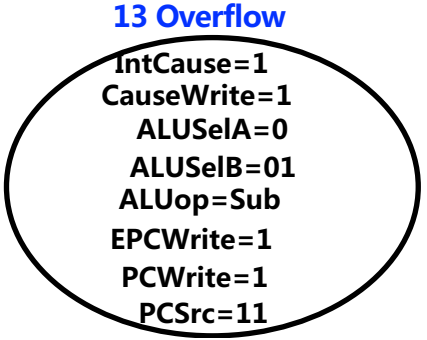
带异常处理的控制器设计

- 在有限状态机中增加异常处理的状态，每种异常占一个状态
- 每个异常处理状态中，需考虑以下基本控制
  - Cause寄存器的设置
  - 计算断点处的PC值（PC-4），并送EPC
  - 将异常查询程序的入口地址送PC
  - 将中断允许位清0（关中断）
- 假设要控制的数据通路中有以下两种异常处理
  - 未定义指令（Cause=0）：状态12
  - 数据溢出（Cause=1）：状态13

  注：7条指令共需12个状态：第0~11状态
- 在原来状态转换图基础上加入两个异常处理状态
  - 如何检测是否发生了这两种异常
    - 未定义指令：当指令译码器发现op字段是一个未定义的编码时
    - 数据溢出：当R-Type指令执行后在ALU输出端的Overflow为1时



12 未定义指令异常状态



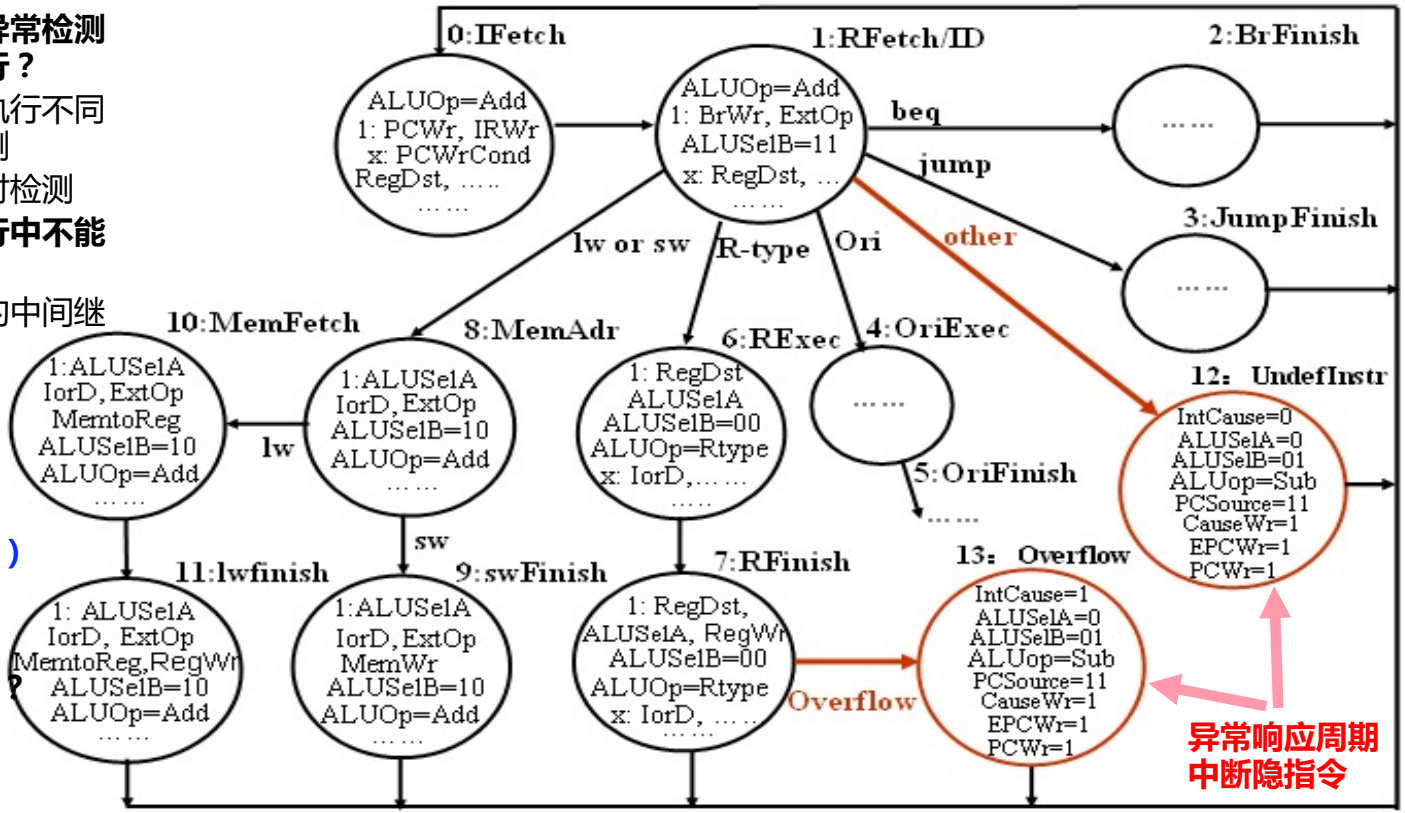
13 数据溢出异常状态

带异常处理的有限状态机

- **问题：中断检测能否和异常检测一样，在指令执行中进行？**
  - 中断随机发生，与指令执行不同步不能在指令执行中检测
  - 总是每条指令执行结束时检测
- **问题：为什么在指令执行中不能响应中断？**
  - 因为无法回到一条指令的中间继续执行

- **“fault”异常的检测在指令执行中。**
- **“trap”异常怎样检测？指令译码（系统调用指令）或条件码检测（单步）**

- **问题：何时检测“缺页”？**
  - MMU中地址转换时！

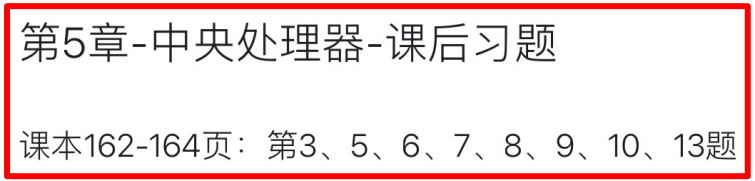


实例：IA-32处理器的实现

- **问题：IA-32处理器适合用单周期还是多周期方式来实现？**
  - **单周期方式**：
    - 每条指令都按最复杂指令时间执行（指令执行效率低！）
    - 功能部件不能重复使用，对于一条具有多个复杂寻址的指令来说，可能要用到相当多个ALU。（成本高！）
  - **多周期方式**：
    - 各指令执行时间可不同，简单指令3-4个时钟，复杂指令几十个时钟（指令执行效率高！）
    - 功能部件可以在一条指令执行过程中重复使用，这对于一条指令中具有多个复杂寻址的指令，非常有好处（成本低！）
- **问题：IA-32处理器适合用硬连线路控制器还是微程序控制器来实现？**
  - **硬连线控制器**：速度快，但无法实现复杂指令
  - **微程序控制器**：容易实现复杂指令，但速度慢
- 从80x486开始，采用了一种**折中的方式**：
  - 简单指令用Hardwired Control
  - 复杂指令用microcoded control，不需为复杂指令构造复杂的控制电路

课程习题（作业）——截止日期：11月6日晚23:59

- **课本162-164页**：第3、5、6、7、8、9、10、13题
- 提交方式：https://selearning.nju.edu.cn/（教学支持系统）



- 命名：学号+姓名+第*章。
- 若提交遇到问题请及时发邮件或在下一次上课时反馈。

课程习题（作业）——截止日期：11月6日晚23:59

3. 右图给出了某 CPU 内部结构的一部分，MAR 和 MDR 直接连到存储器总线(图中省略)。在 CPU 内部总线 A 和 B 之间的所有数据传送都需经过算术逻辑部件 ALU。ALU 的部分控制信号及其功能如下：

MOVa：F=A；　　MOVb：F=B；

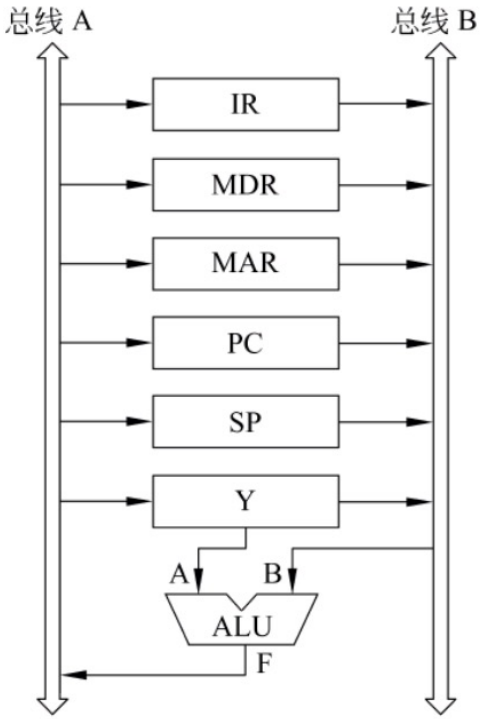
a+1：F=A+1；　　b+1：F=B+1；

a−1：F=A−1；　　b−1：F=B−1。

其中 A 和 B 是 ALU 的输入，F 是 ALU 的输出。假定该 CPU 的指令系统中调用指令 CALL 占两个字，第一个字是操作码，第二个字给出子程序的起始地址，返回地址保存在主存的栈中，用 SP (栈指示器)指向栈顶，存储器按字编址，每次按同步方式从主存读取一个字。要求：

(1) 说明 CALL 指令的功能。

(2) 写出读取并执行 CALL 指令所要求的控制信号序列(提示：当前指令地址已在 PC 中)。



课程习题（作业）——截止日期：11月6日晚23:59

5. 假定图 5.16 所示单周期数据通路对应的控制逻辑发生错误，使得控制信号 RegWr、RegDst、ALUSrc、Branch、MemWr、ExtOp、R-type、MemtoReg 中某一个在任何情况下总是为 0，则该控制信号为 0 时哪些指令不能正确执行？要求分别讨论。

6. 假定图 5.16 所示单周期数据通路对应的控制逻辑发生错误，使得控制信号 RegWr、RegDst、ALUSrc、Branch、MemWr、ExtOp、R-type、MemtoReg 中某一个在任何情况下总是为 1，则该控制信号为 1 时哪些指令不能正确执行？要求分别讨论。

7. 要在 MIPS 指令集中增加一条 swap 指令，可以有两种做法。一种做法是采用伪指令方式（即软件方式），这种情况下，当执行到 swap 指令时，用若干条已有指令构成的指令序列来代替实现；另一种做法是直接改动硬件来实现 swap 指令，这种情况下，当执行到 swap 指令时，则可在 CPU 上直接执行。要求：

（1）写出用伪指令方式实现“swap rs，rt”时的指令序列（提示：伪指令对应的指令序列中不能使用其他额外寄存器，以免破坏这些寄存器的值）。

（2）假定用硬件实现 swap 指令时会使每条指令的执行时间增加 10%，则 swap 指令要在程序中占多大的比例才值得用硬件方式来实现？

课程习题（作业）——截止日期：11月6日晚23:59

8. 假定图 5.25 多周期数据通路对应的控制逻辑发生错误，使得控制信号 PCWr、MemtoReg、IRWr、RegWr、BrWr、MemWr、PCWrCond、R-type 中某一个在任何情况下总是为 0，则该控制信号为 0 时哪些指令不能正确执行？要求分别讨论。

9. 假定图 5.25 多周期数据通路对应的控制逻辑发生错误，使得控制信号 PCWr、MemtoReg、IRWr、RegWr、BrWr、MemWr、PCWrCond、R-type 中某一个在任何情况下总是为 1，则该控制信号为 1 时哪些指令不能正确执行？要求分别讨论。

10. 假定有一条 MIPS 伪指令“bcmp $t1，$t2，$t3”，其功能是实现对两个主存块数据的比较，$t1 和 $t2 中分别存放两个主存块的首地址，$t3 中存放数据块的长度，每个数据占 4 字节，若所有数据都相等，则将 0 置入 $t1；否则，将第一次出现不相等时的地址分别置入 $t1 和 $t2 并结束比较。若 $t4 和 $t5 是两个空闲寄存器，请给出实现该伪指令的指令序列，并说明在类似于图 5.25 所示的多周期数据通路中执行该伪指令时要用多少个时钟周期。

课程习题（作业）——截止日期：11月6日晚23:59

13. 对于多周期 CPU 中的异常和中断处理，回答以下问题：

（1）对于除数为 0、溢出、无效指令操作码、无效指令地址、无效数据地址、缺页、访问越权和外部中断，CPU 在哪些指令的哪个时钟周期能分别检测到这些异常或中断？

（2）在检测到某个异常或中断后，CPU 通常要完成哪些工作？简要说明 CPU 如何完成这些工作。

提问

Q & A

殷亚凤
智能软件与工程学院
苏州校区南雍楼东区225
yafeng@nju.edu.cn，https://yafengnju.github.io/

南京大学 NANJING UNIVERSITY