



南京大學

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





# CPU概述——CPU功能及其与计算机性能的关系

## • CPU执行指令的过程

- 取指令
  - $PC + 1$  送PC
  - 指令译码
  - 进行主存地址运算
  - 取操作数
  - 进行算术 / 逻辑运算
  - 存结果
  - 以上每步都需检测“异常”
  - 若有异常，则自动切换到异常处理程序
  - 检测是否有“中断”请求，有则转中断处理
- 取指令阶段
- 译码和执行阶段
- 指令执行过程

## • CPU的实现与计算机性能的关系

- 计算机性能(程序执行快慢)由三个关键因素决定 (回顾)
  - 指令数目、CPI、时钟周期
    - 指令数目由编译器和ISA决定
    - 时钟周期和CPI主要由CPU的设计与实现决定

因此，CPU的设计与实现非常重要！它直接影响计算机的性能。

问题：

- “取指令”一定在最开始做吗？  
**一定！**
- “ $PC + 1$ ”一定在译码前做吗？  
**不一定！**
- “译码”须在指令执行前做吗？  
**是！**
- 你能说出哪几种“异常”事件？  
**缺页、越界**
- 异常和中断的差别是什么？
  - 异常是在CPU内部发生的
  - 中断是由外部事件引起的





# CPU概述——组成指令功能的四种基本操作

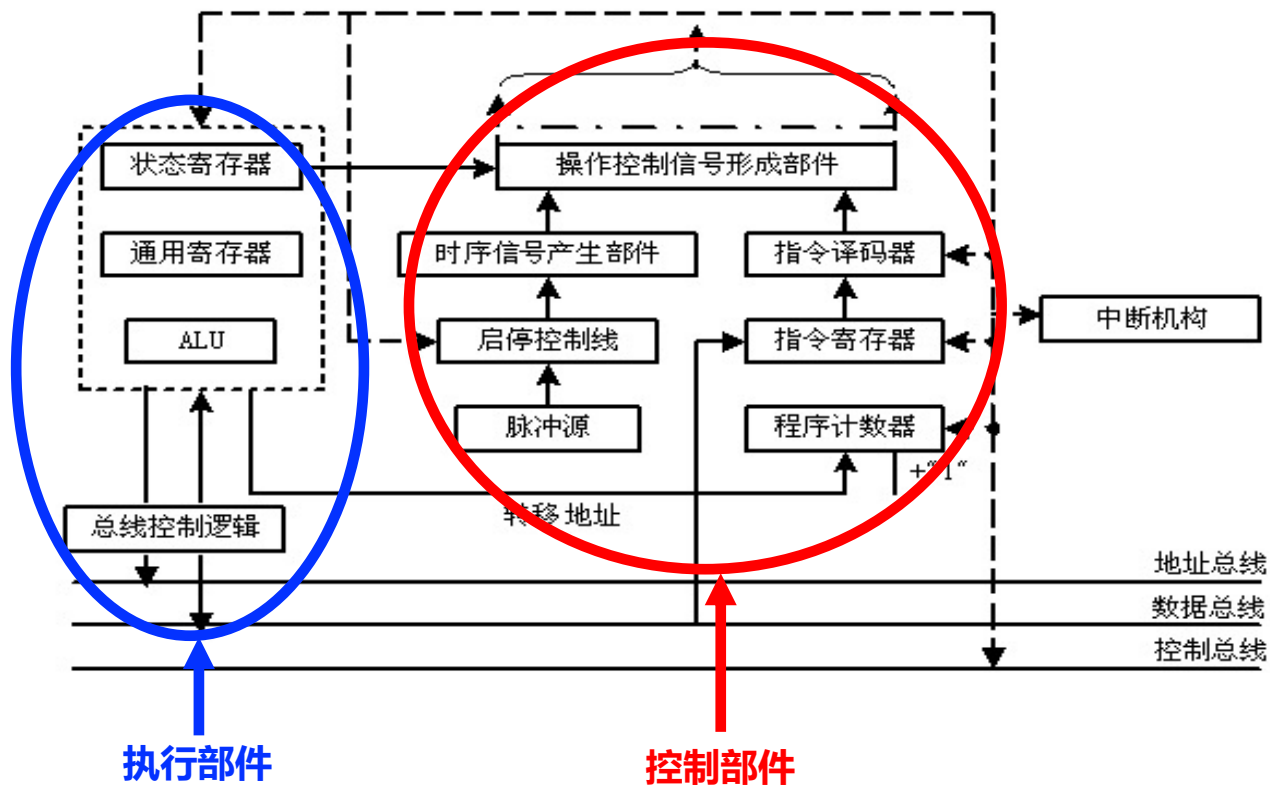
- 每条指令的功能总是由以下四种基本操作的组合来实现：
  - 读取某一主存单元的内容，并将其装入某个寄存器（取指，取数）
  - 把一个数据从某个寄存器存入给定的主存单元中（存结果）
  - 把一个数据从某个寄存器送到另一个寄存器或者ALU（取数，存结果）
  - 进行算术或逻辑运算（ $PC+1$ ，计算地址，运算）
- 操作功能可形式化描述，描述语言称为寄存器传送语言RTL (Register Transfer Language)
  - (1) 用 $R[r]$ 表示寄存器 $r$ 的内容；
  - (2) 用 $M[addr]$ 表示主存单元 $addr$ 的内容；
  - (3) 用 $M[R[r]]$ 表示寄存器 $r$ 的内容所指存储单元的内容；
  - (4) 程序计数器PC直接用PC表示其内容。
  - (5)  $M[PC]$ 表示Pc所指存储单元的内容；
  - (6)  $SEXT[imm]$ 表示对 $imm$ 将进行符号扩展；
  - (7)  $ZEXT[imm]$ 表示对 $imm$ 进行零扩展；
  - (8) 传送方向用“ $\leftarrow$ ”表示，传送源在右，传送目的在左；

例如， $R[\$8] \leftarrow M[R[\$9]+4]$ 的含义是：  
将寄存器\$9的内容加4得到的内存地址中的内容送寄存器\$8中。





# CPU的基本组成——组成原理图



指令寄存器----IR  
程序计数器----PC

**CPU包含：**

- 数据通路(执行部件)
- 控制器(控制部件)

**控制器包含：**

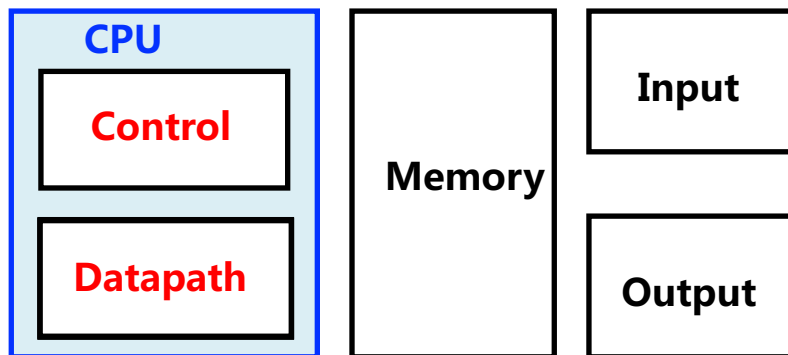
- 指令译码器
- 控制信号形成部件





# CPU的基本组成——数据通路的位置

- 计算机的五大组成部分：



- 什么是数据通路 (DataPath) ?
  - 指令执行过程中，数据所经过的路径，包括路径中的部件。它是指令的执行部件。
- 控制器 (Control) 的功能是什么？
  - 对指令进行译码，生成指令对应的控制信号，控制数据通路的动作。能对执行部件发出控制信号，是指令的控制部件。





# CPU的基本组成——数据通路的基本结构

- **数据通路由两类元件组成**
  - 组合逻辑元件（也称操作元件）
  - 时序逻辑元件（也称状态元件，存储元件）
- **元件间的连接方式**
  - 总线连接方式
  - 分散连接方式
- **数据通路如何构成？**
  - 由“操作元件”和“存储元件”通过总线方式或分散方式连接而成
- **数据通路的功能是什么？**
  - 进行数据存储、处理、传送

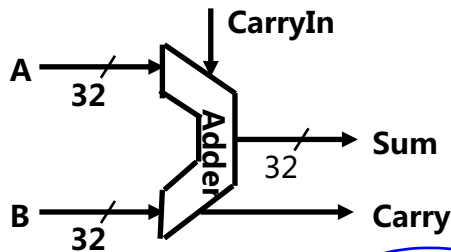


数据通路是由**操作元件**和**存储元件**通过总线方式或分散方式连接而成的进行数据存储、处理、传送的路径。

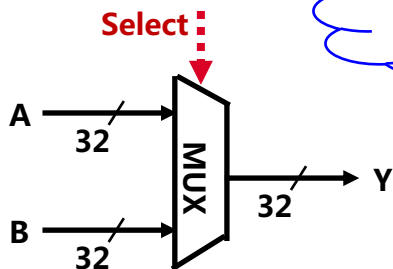


# CPU的基本组成——操作元件

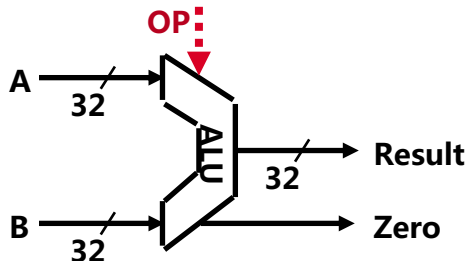
加法器(Adder)



多路选择器(MUX)  
(二选一也可以  
多选一)

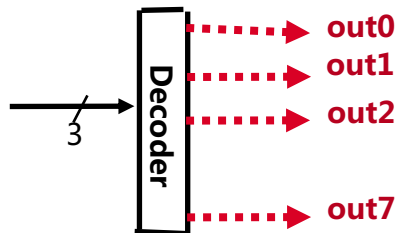


算逻部件(ALU)



译码器(Decoder)

.....► 控制信号



加法器需要什么控制信号?**不需要!**

何时要用到adder, ALU, MUX or Decoder?

## 组合逻辑元件的特点：

- 其输出只取决于当前的输入。即：若输入一样，则其输出也一样
- 定时：所有输入到达后，经过一定的逻辑门延时，输出端改变，并保持到下次改变，**不需要时钟信号**来定时





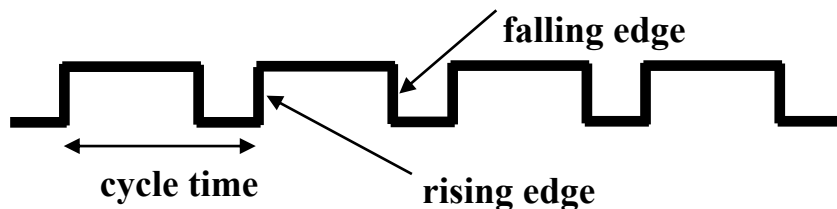
# CPU的基本组成——状态元件

- **状态（存储）元件的特点：**

- 具有存储功能，在**时钟控制**下输入被写到电路中，直到下个时钟到达
- 输入端状态由时钟决定何时被写入，输出端状态随时可以读出

- **定时方式：规定信号何时写入状态元件或何时从状态元件读出**

- 边沿触发（edge-triggered）方式：
  - 状态单元中的值只在时钟边沿改变。每个时钟周期改变一次。
    - 上升沿（rising edge）触发：在时钟正跳变时进行读/写。
    - 下降沿（falling edge）触发：在时钟负跳变时进行读/写。



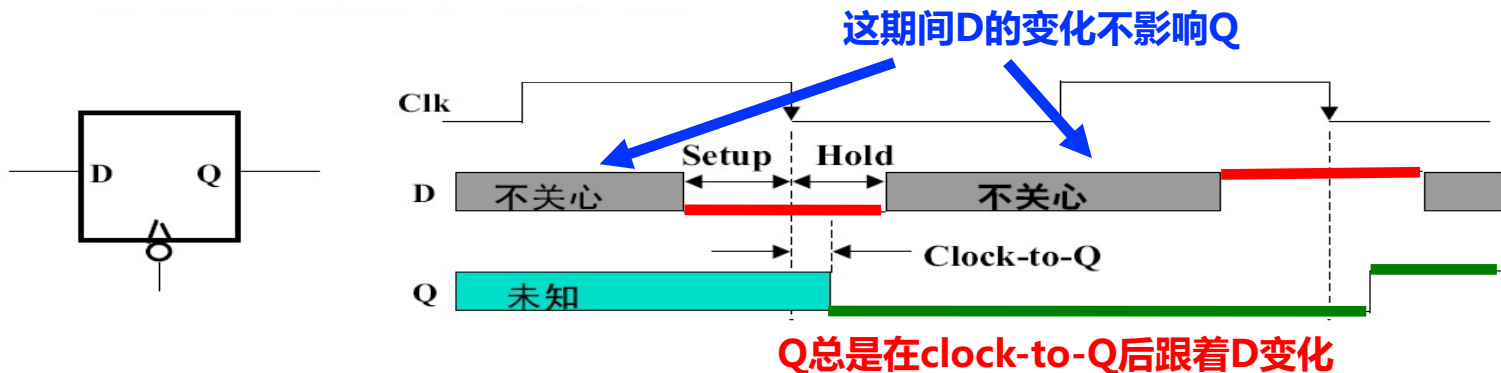
- **最简单的状态单元：**

- D触发器：一个时钟输入、一个状态输入、一个状态输出





# CPU的基本组成——状态元件



**建立时间(Setup):** 在触发时钟边沿**之前**，输入必须稳定

**保持时间(Hold):** 在触发时钟边沿**之后**，输入必须稳定

**锁存延迟(Clk-to-Q):** 在触发时钟边沿，输出并**不能立即变化**。

切记：状态单元的输入信息总是在一个时钟边沿到达后的“Clk-to-Q”时才被写入到单元中，此时的输出才反映新的状态值。

数据通路中的状态元件有两种：寄存器(组) + 存储器





# CPU的基本组成——状态元件

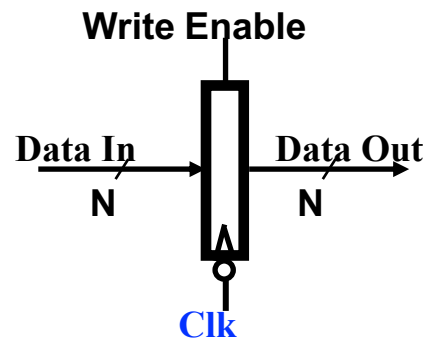
## • 寄存器 ( Register )

- 有一个写使能 ( Write Enable-WE ) 信号

0: 时钟边沿到来时, 输出不变

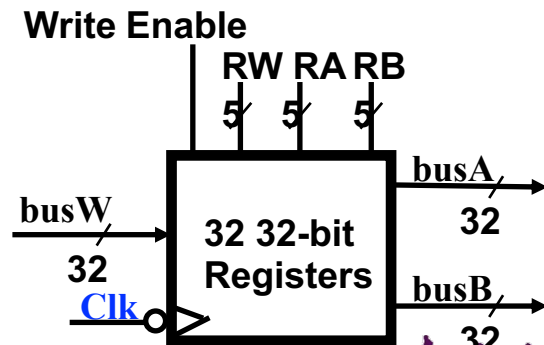
1: 时钟边沿到来时, 输出开始变为输入

- 若每个时钟边沿都写入, 则不需WE信号



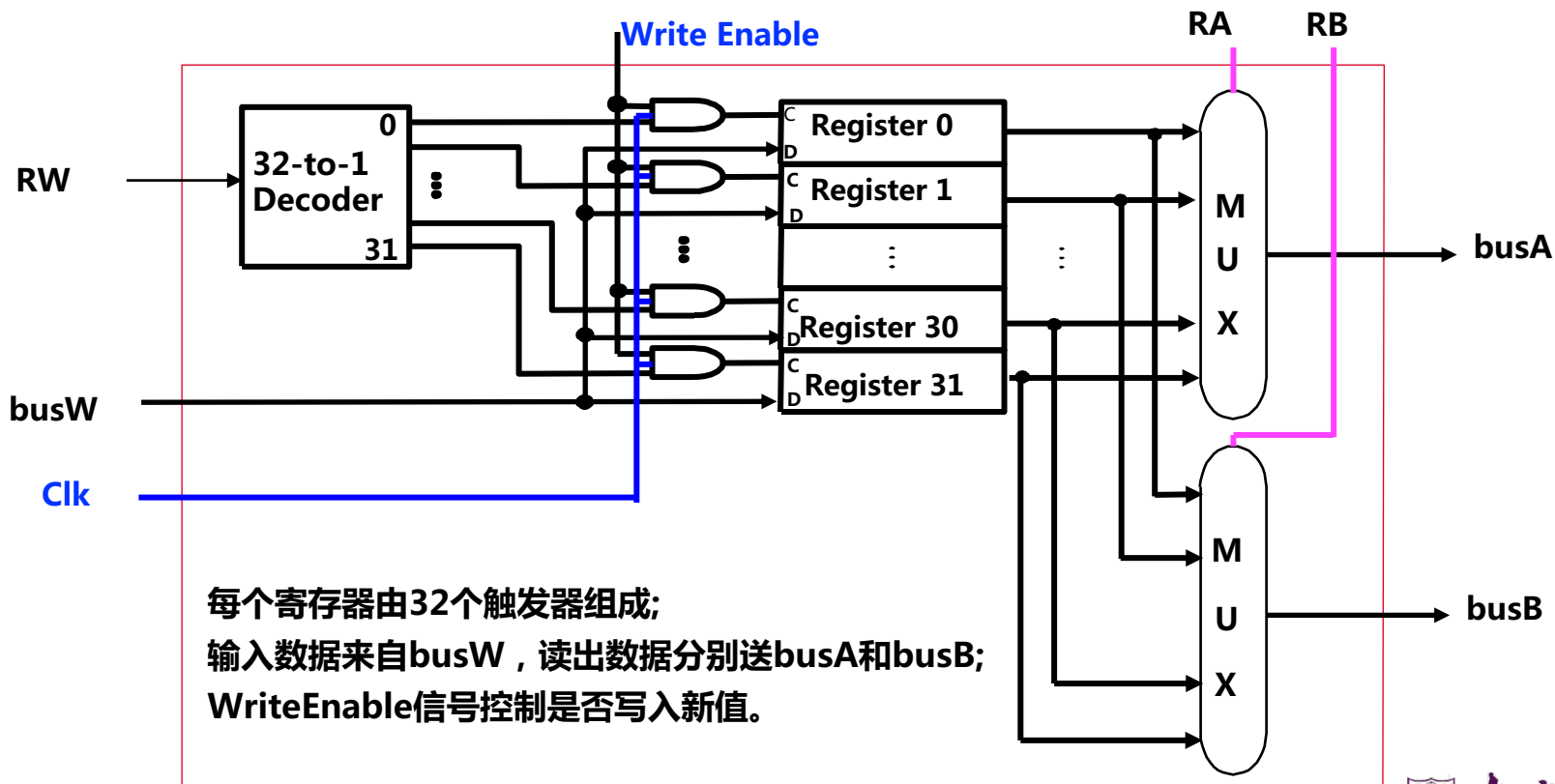
## • 寄存器组 ( Register File )

- 两个读口 ( 组合逻辑操作 ) : busA和busB分别由RA和RB给出地址。地址RA或RB有效后, 经一个“取数时间 (AccessTime)”, busA和busB有效。
- 一个写口 ( 时序逻辑操作 ) : 写使能为1的情况下, 时钟边沿到来时, busW传来的值开始被写入RW指定的寄存器中。





# CPU的基本组成——寄存器组的内部结构

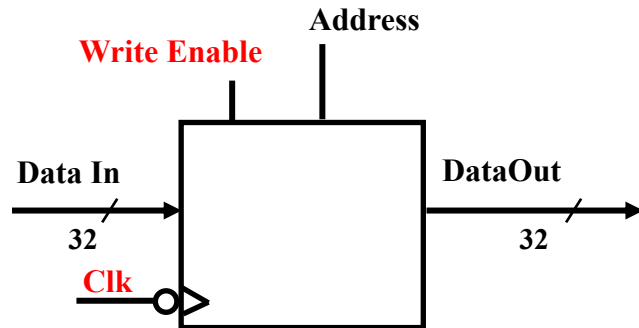




# CPU的基本组成——理想存储器

- 理想存储器 ( idealized memory )

- Data Out : 32位读出数据
- Data In : 32位写入数据
- Address : 读写公用一个32位地址



- 读操作 ( 组合逻辑操作 ) : 地址Address有效后, 经一个 “取数时间 AccessTime” , Data Out上数据有效。
- 写操作 ( 时序逻辑操作 ) : 写使能为1的情况下, 时钟Clk边沿到来时, Data In传来的值开始被写入Address指定的存储单元中。

为简化数据通路操作说明, 把存储器简化为带时钟信号Clk的理想模型。



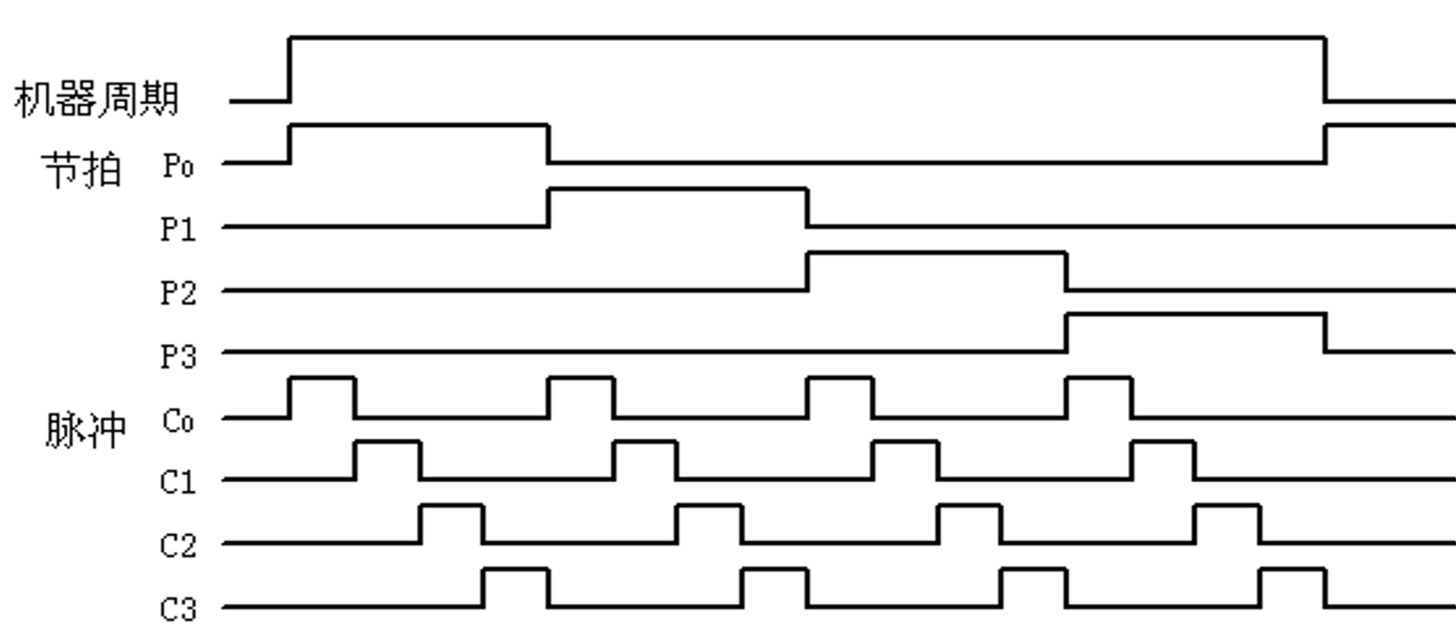


# 数据通路与时序控制

- **同步系统(Synchronous system)**
  - 所有动作有专门时序信号来定时
  - 由时序信号规定何时发出什么动作  
例如，指令执行过程每一步都有控制信号控制，由定时信号确定控制信号何时发出、作用时间多长。
- **什么是时序信号？**
  - 同步系统中用于进行同步控制的定时信号，如时钟信号
- **什么叫指令周期？**
  - 取并执行一条指令的时间
  - 每条指令的指令周期肯定一样吗？(不一样！)
- **早期计算机的三级时序系统**
  - 机器周期 - 节拍 - 脉冲
  - 指令周期可分为取指令、读操作数、执行并写结果等多个基本工作周期，称为机器周期。
  - 机器周期有取指令、存储器读、存储器写、中断响应等不同类型



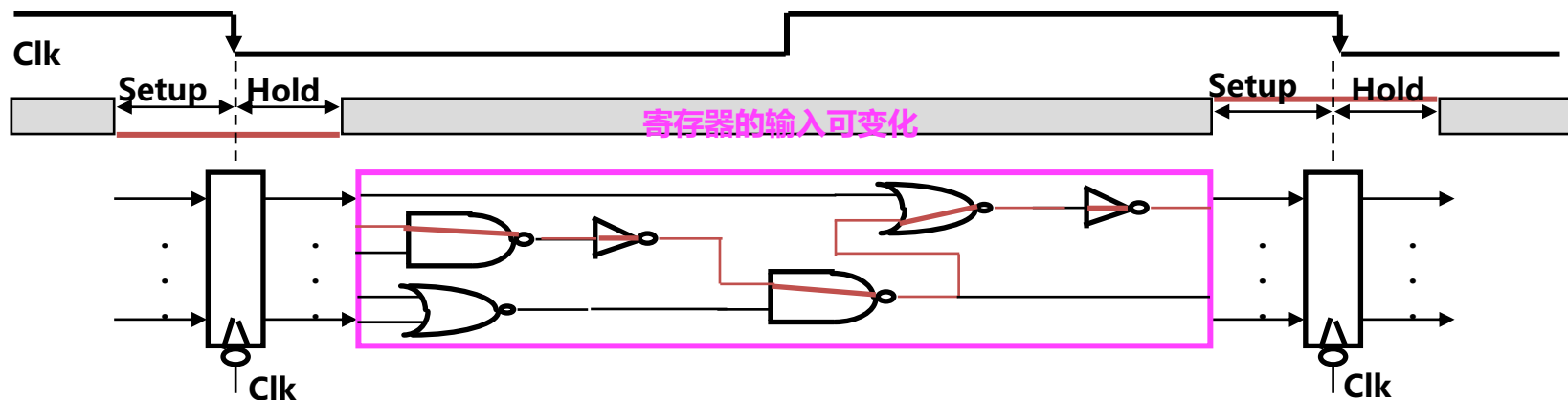
## 数据通路与时序控制——早期计算机的三级时序



现代计算机已不再采用三级时序系统，机器周期的概念已逐渐消失。整个数据通路中的定时信号就是时钟，一个时钟周期就是一个节拍。



# 数据通路与时序控制——现代计算机的时钟周期



数据通路由 “... + 状态元件 + 操作元件(组合电路) + 状态元件 + ...” 组成

- **只有状态元件能存储信息**，所有操作元件都须从状态单元接收输入，并将输出写入状态单元中。  
其输入为前一时钟生成的数据，输出为当前时钟所用的数据
- **假定采用下降沿触发（负跳变）方式（也可以是上升沿方式）**
  - 所有状态单元在下降沿写入信息，经过锁存延迟 Latch Prop (clk-to-Q) 后输出有效
  - **时钟周期 = 锁存延迟 + 最长传输延迟 + 建立时间 + 时钟偏移；**
- **约束条件：(锁存延迟 + 最短传输延迟 - 时钟偏移) > 保持时间**





# 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





# 单周期处理器设计

- **ISA确定后，进行处理器设计的大致步骤：**
  - 第一步：分析每条指令的功能，并用RTL(Register Transfer Language)来表示。
  - 第二步：根据指令的功能给出所需的元件，并考虑如何将他们互连。
  - 第三步：确定每个元件所需控制信号的取值。
  - 第四步：汇总所有指令所涉及到的控制信号，生成反映指令与控制信号之间关系的表。
  - 第五步：根据表得到每个控制信号的逻辑表达式，据此设计控制器电路。
- ◆ **处理器设计涉及到数据通路的设计和控制器的设计**
- ◆ **数据通路中有两种元件**
  - **操作元件**：由组合逻辑电路实现
  - **存储（状态）元件**：由时序逻辑电路实现

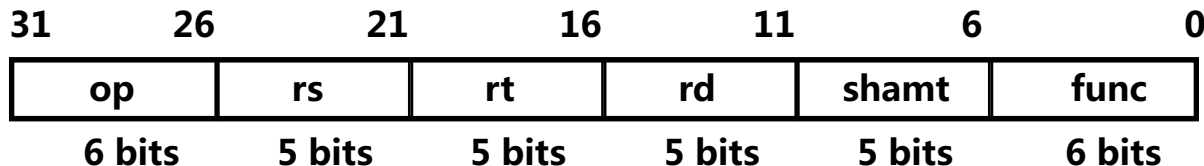




# 单周期处理器设计——MIPS的三种指令格式

- ADD and SUBTRACT**

- add rd, rs, rt
- sub rd, rs, rt



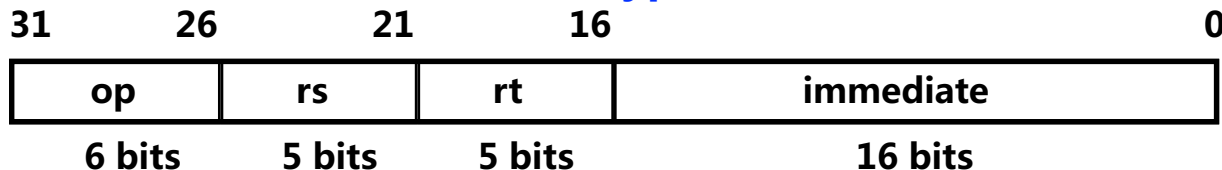
- OR Immediate:**

- ori rt, rs, imm16

R-Type

- LOAD and STORE**

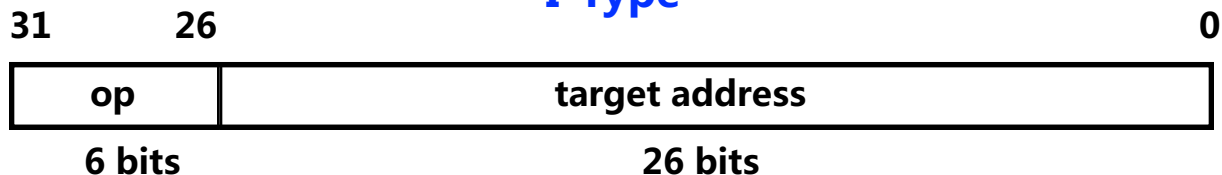
- lw rt, rs, imm16
- sw rt, rs, imm16



I-Type

- BRANCH:**

- beq rs, rt, imm16



J-Type

- JUMP:**

- j target

这些指令具有代表性：有算术运算、逻辑运算；有RR型、RI型；有访存指令；有条件转移、无条件转移。

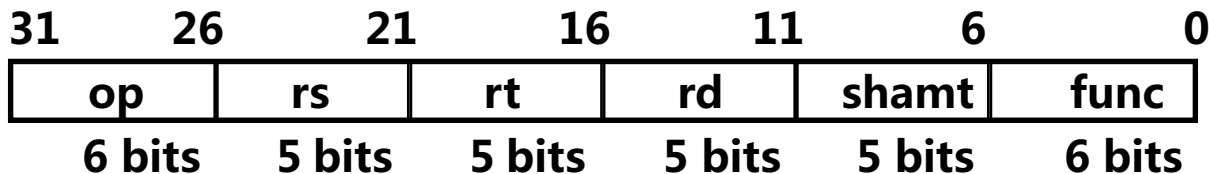
本讲目标：实现以上7条指令对应数据通路！

教材中实现了11条指令，可将7条指令和11条指令的数据通路进行对比，以深入理解设计原理。



# 指令功能的描述——寄存器传送级语言RTL

## 加法指令



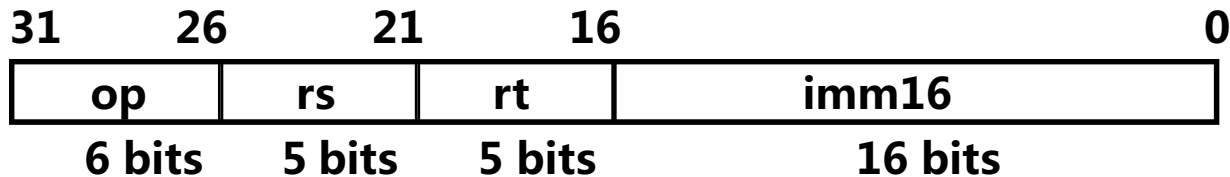
- **add rd, rs, rt**

- $M[PC]$                       #从PC所指的内存单元中取指令
- $R[rd] \leftarrow R[rs] + R[rt]$       #从rs、rt 所指的寄存器中取数后相加。  
        若结果不溢出，则将结果送rd 所指的寄存器中；  
        若结果溢出，则不送结果，  
        并转到“溢出处理程序”执行。
- $PC \leftarrow PC + 4$               #PC加4，使PC指向下一条指令



# 指令功能的描述——寄存器传送级语言RTL

## 装入指令



- **lw rt, rs, imm16**

- $M[PC]$  # (同加法指令)
- $Addr \leftarrow R[rs] + \text{SignExt}(imm16)$  # 计算数据地址 (立即数要进行符号扩展)
- $R[rt] \leftarrow M[Addr]$  # 从存储器中取出数据, 装入到寄存器中
- $PC \leftarrow PC + 4$  # (同加法指令)

与R-type加法指令相比, 更复杂!

其他指令的分析与R-type和Load指令类似

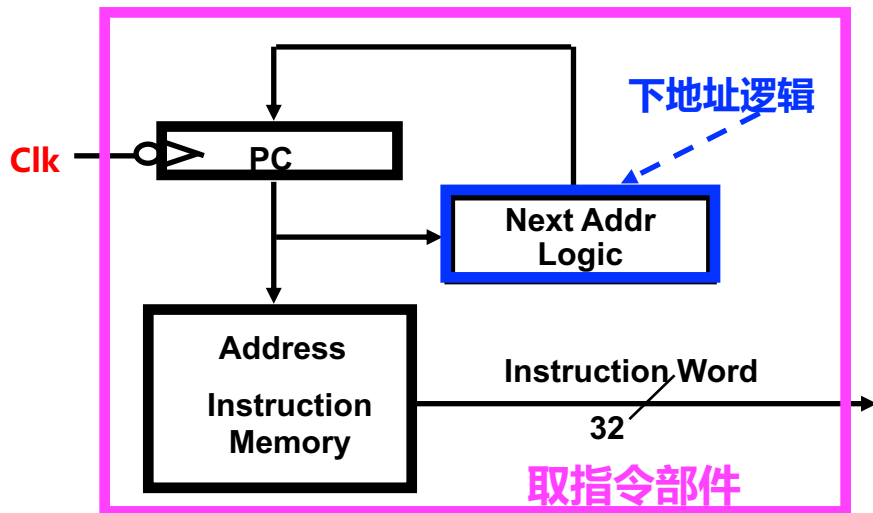


# 数据通路的设计——取指令部件

- 每条指令都有的公共操作：

- 取指令：M[PC]
- 更新PC：PC  $\leftarrow$  PC + 4

转移（Branch and Jump）时，PC内容再次被更新为“转移目标地址”



## 顺序：

- 先取指令，再改PC的值（具体实现时，可以并行）
- 绝不能先改PC的值，再取指令

取指后，各指令功能不同，数据通路中信息流动过程也不同。

下面分别对每条指令进行相应数据通路的设计，

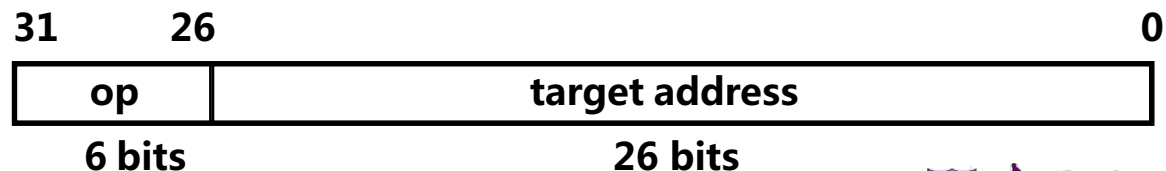
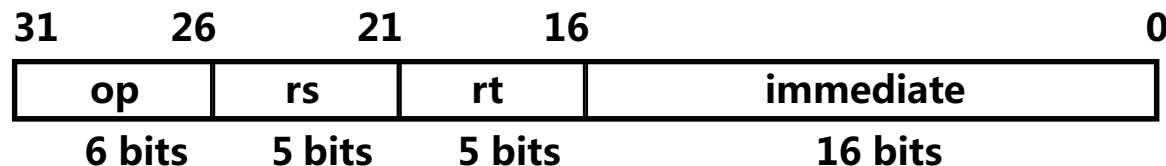
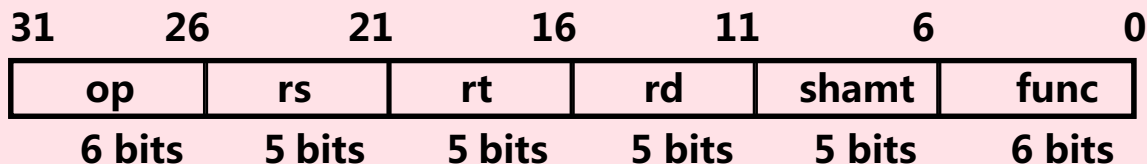


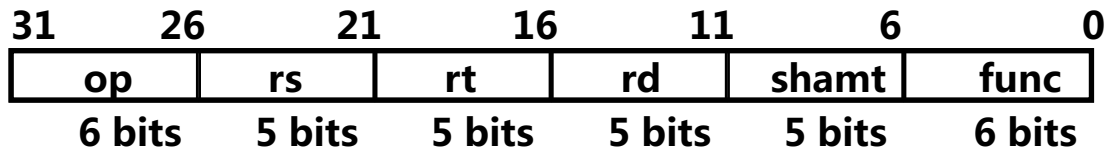
# 数据通路的设计——加法和减法指令(R-型)

实现目标（7条指令）：

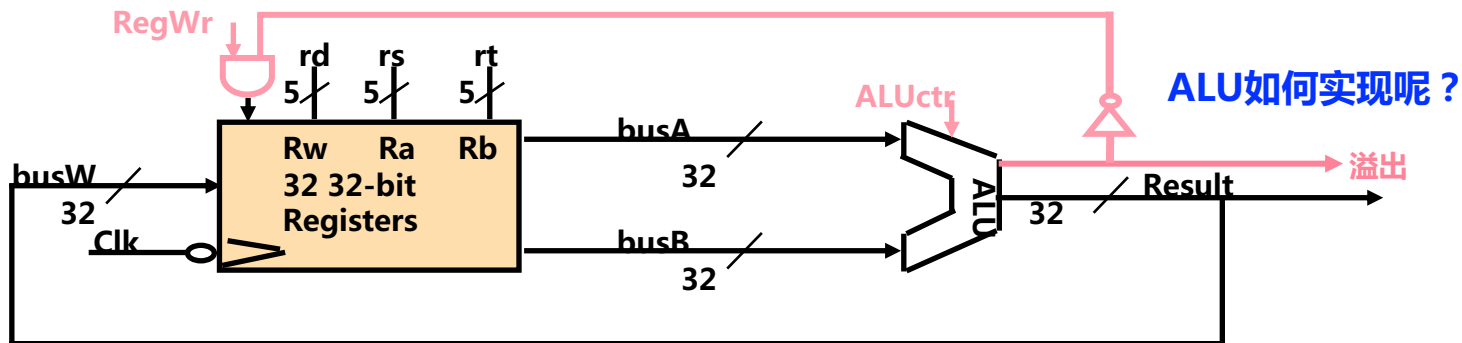
- **ADD and subtract**
  - add rd, rs, rt
  - sub rd, rs, rt
- **OR Immediate:**
  - ori rt, rs, imm16
- **LOAD and STORE**
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- **BRANCH:**
  - beq rs, rt, imm16
- **JUMP:**
  - j target

## 1. 首先考虑add和sub指令（R-Type指令的代表）





- 功能： $R[rd] \leftarrow R[rs] \text{ op } R[rt]$ ，如：`add rd, rs, rt` **溢出时，不写结果并需转异常处理程序**  
不考虑公共操作，仅R-Type指令执行阶段的数据通路如下：



**ALUctr, RegWr:** 指令译码后产生的控制信号

## “add rd, rs, rt” 控制信号？

**ALUctr=add , RegWr=1**



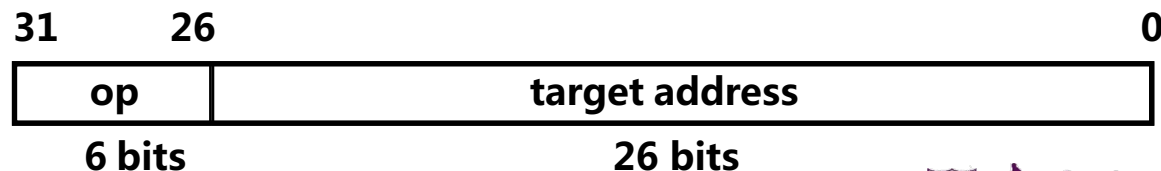
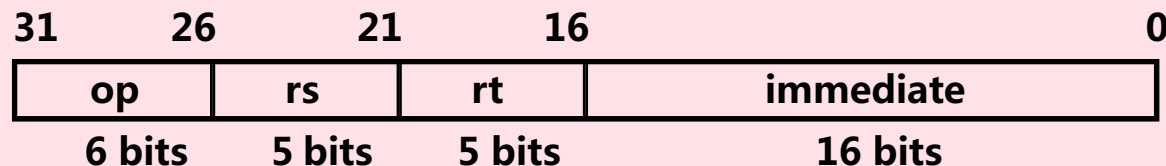
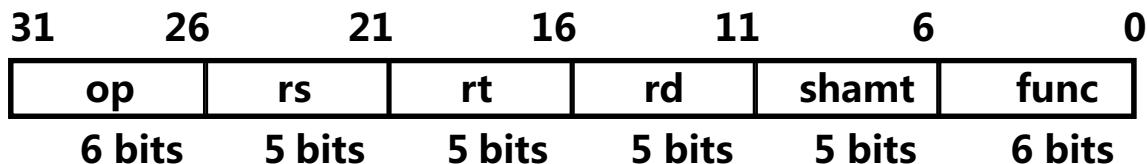


# 数据通路的设计——带立即数的逻辑指令（ori指令）

## 实现目标（7条指令）：

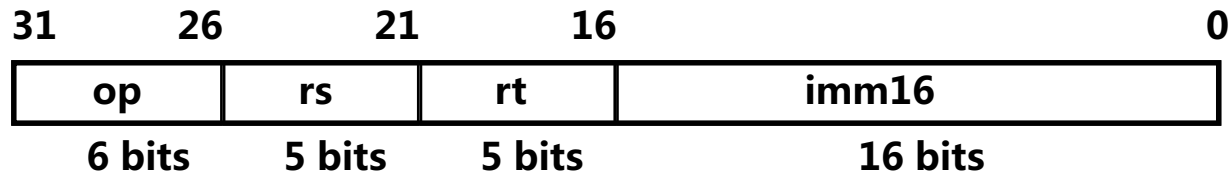
- **ADD and subtract**
  - add rd, rs, rt
  - sub rd, rs, rt
- **OR Immediate:**
  - ori rt, rs, imm16
- **LOAD and STORE**
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- **BRANCH:**
  - beq rs, rt, imm16
- **JUMP:**
  - j target

## 2. 考虑ori 指令（I-Type指令和逻辑运算指令的代表）





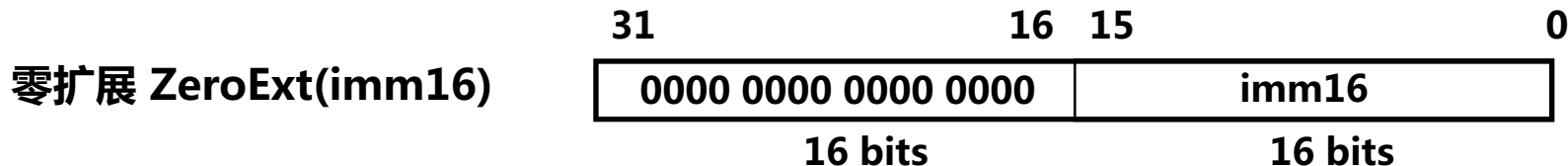
# 数据通路的设计——带立即数的逻辑指令（ori指令）



逻辑运算，  
立即数为逻辑数

## • ori rt, rs, imm16

- M[PC] #取指令（公共操作，取指部件完成）
- $R[rt] \leftarrow R[rs] \text{ or } \text{ZeroExt}(\text{imm16})$  #立即数零扩展，并与rs内容做“或”运算
- $PC \leftarrow PC + 4$  #计算下地址（公共操作，取指部件完成）



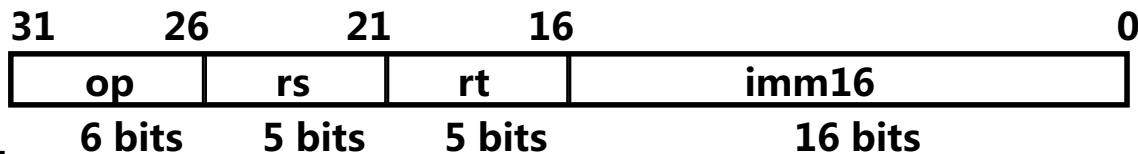
思考：应在前面数据通路上加哪些元件和连线？用何控制信号？



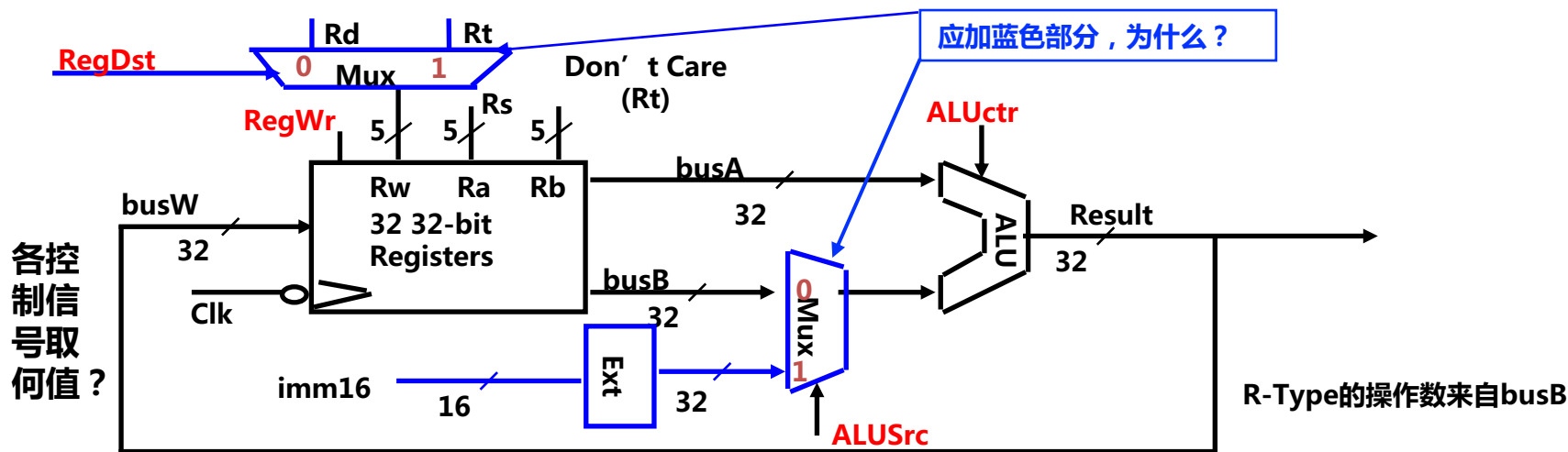
# 数据通路的设计——带立即数的逻辑指令的数据通路

- $R[rt] \leftarrow R[rs] \text{ op ZeroExt}[imm16]$

Example: ori rt, rs, imm16



R-Type类型的结果写入Rd



Ori指令的控制信号：RegDst=1；RegWr=1；ALUSrc=1；ALUctr=or

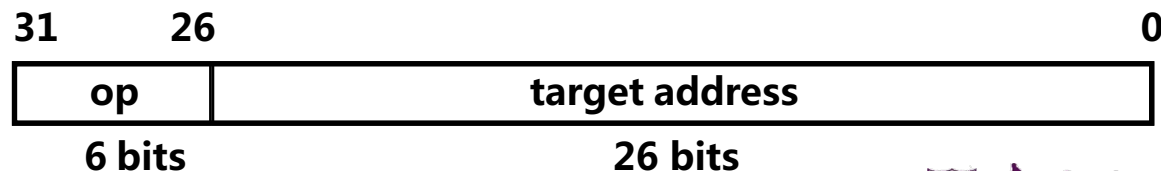
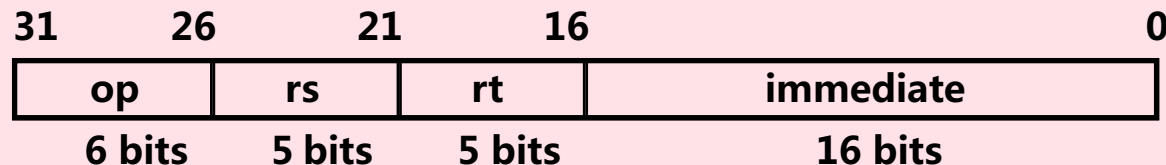
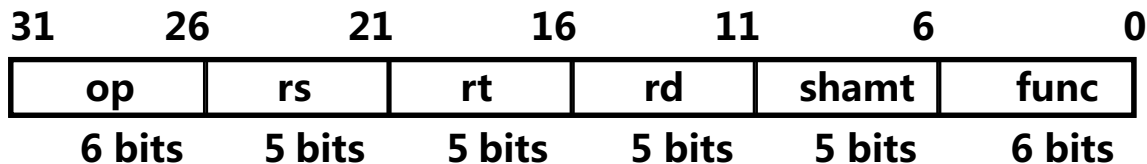


# 数据通路的设计——访存指令中的数据装入指令 (lw)

## 实现目标（7条指令）：

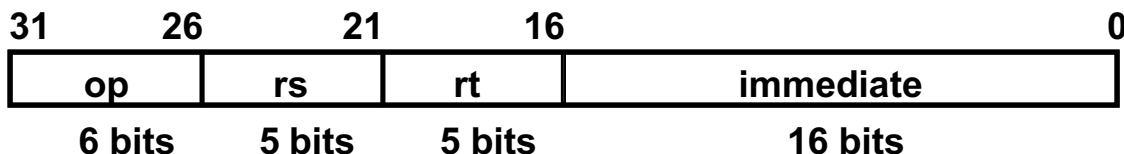
- **ADD and subtract**
  - add rd, rs, rt
  - sub rd, rs, rt
- **OR Immediate:**
  - ori rt, rs, imm16
- **LOAD and STORE**
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- **BRANCH:**
  - beq rs, rt, imm16
- **JUMP:**
  - j target

## 3. 考虑lw 指令（访存指令的代表）





# 数据通路的设计——访存指令中的数据装入指令 (lw)

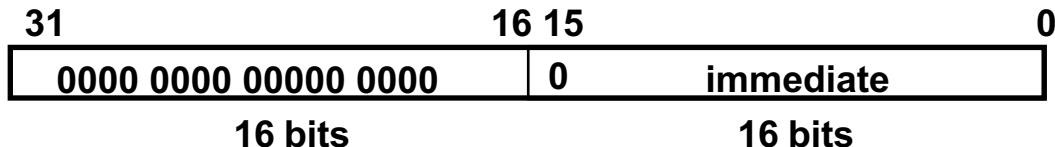


立即数用补码表示

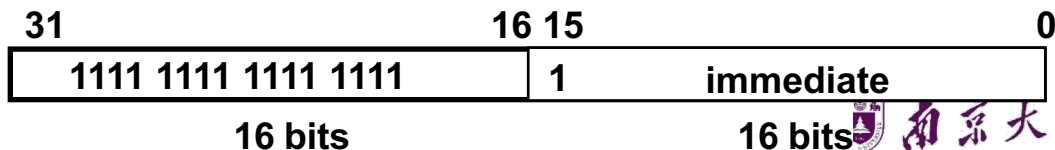
## • lw rt, rs, imm16

- M[PC] #取指令 (公共操作, 取指部件完成)
- $\text{Addr} \leftarrow R[\text{rs}] + \text{SignExt}(\text{imm16})$  #计算存储单元地址 (符号扩展!)
- $R[\text{rt}] \leftarrow M[\text{Addr}]$  #装入数据到寄存器rt中
- $\text{PC} \leftarrow \text{PC} + 4$  #计算下地址 (公共操作, 取指部件完成)

符号扩展(为什么不是零扩展?):



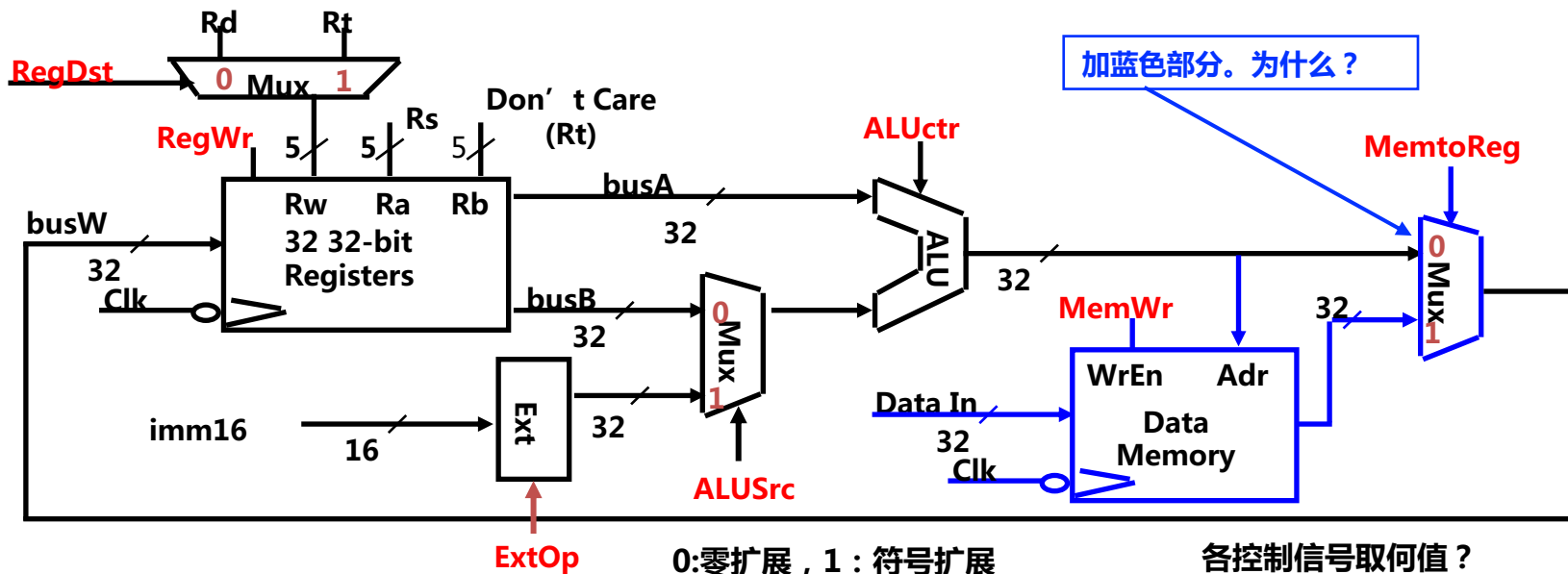
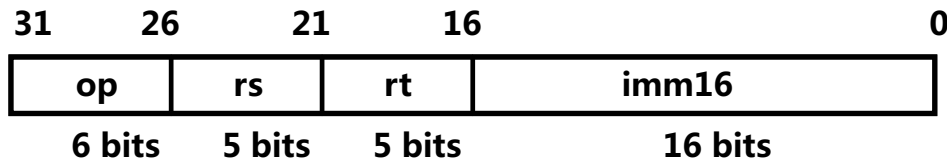
思考: 应在原数据通路上加哪些元件和连线? 用何控制信号?





# 数据通路的设计——装入指令 (lw) 的数据通路

- $R[rt] \leftarrow M[R[rs] + \text{SignExt}[imm16]]$
- lw rt, rs, imm16



RegDst=1, RegWr=1, ALUctr=addu, ExtOp=1, ALUSrc=1, MemWr=0, MemtoReg=1

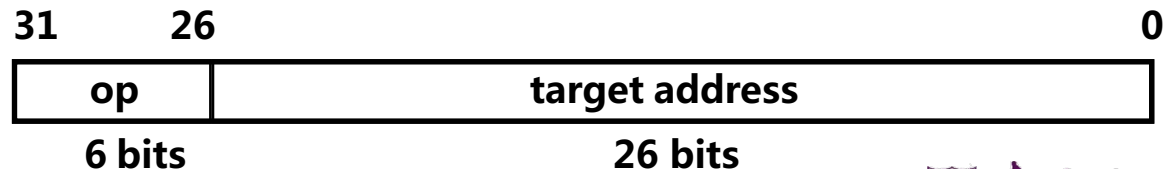
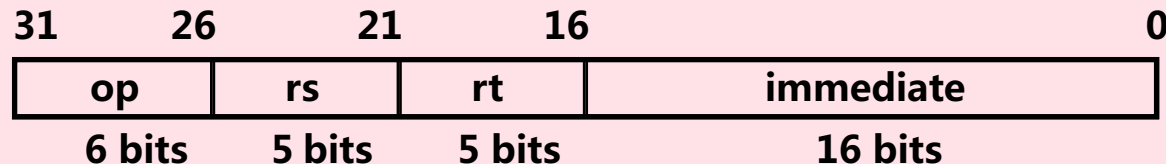
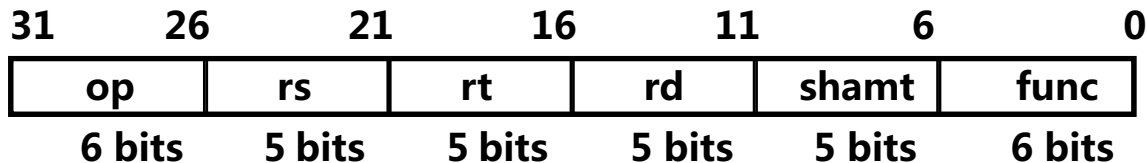


# 数据通路的设计——访存指令中的存数指令 (sw)

## 实现目标（7条指令）：

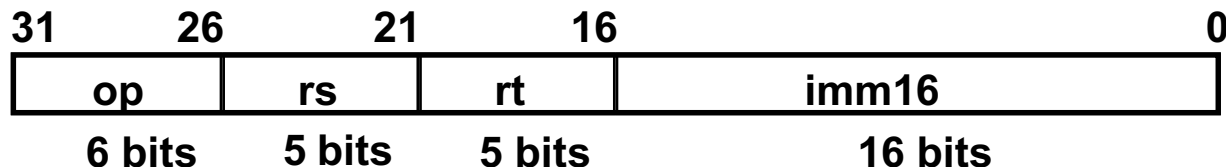
- **ADD and subtract**
  - add rd, rs, rt
  - sub rd, rs, rt
- **OR Immediate:**
  - ori rt, rs, imm16
- **LOAD and STORE**
  - lw rt, rs, imm16
  - **sw rt, rs, imm16**
- **BRANCH:**
  - beq rs, rt, imm16
- **JUMP:**
  - j target

## 4. 考虑sw 指令（访存指令的代表）





## 数据通路的设计——访存指令中的存数指令 (sw)



- **sw rt, rs, imm16**

- $M[PC]$  #取指令（公共操作，取指部件完成）
- $Addr \leftarrow R[rs] + \text{SignExt}(imm16)$  #计算存储单元地址（符号扩展！）
- $Mem[Addr] \leftarrow R[rt]$  #寄存器rt中的内容存到内存单元中
- $PC \leftarrow PC + 4$  #计算下地址（公共操作，取指部件完成）

思考：应在原数据通路上加哪些元件和连线？用何控制信号？







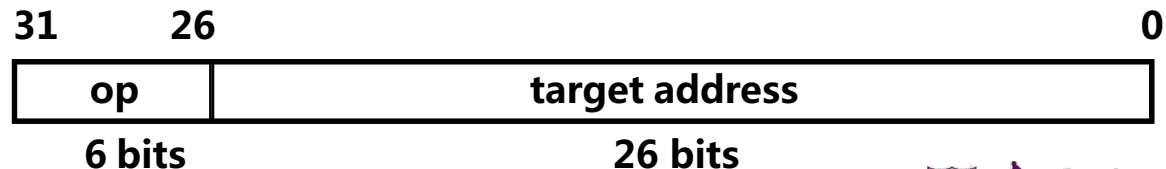
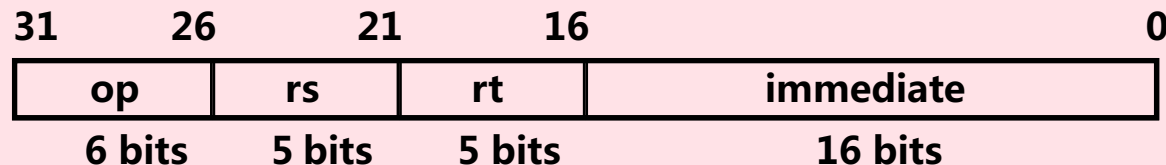
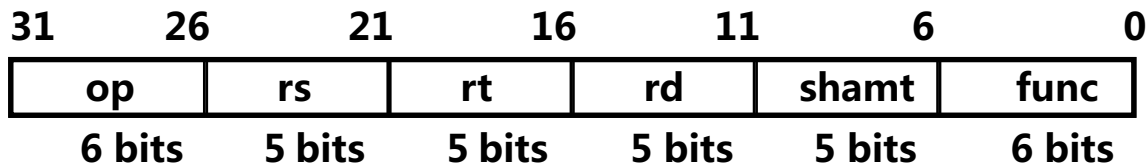


# 数据通路的设计——分支（条件转移）指令（beq）

## 实现目标（7条指令）：

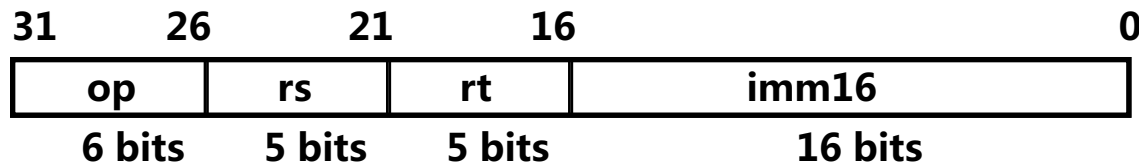
- **ADD and subtract**
  - add rd, rs, rt
  - sub rd, rs, rt
- **OR Immediate:**
  - ori rt, rs, imm16
- **LOAD and STORE**
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- **BRANCH:**
  - beq rs, rt, imm16
- **JUMP:**
  - j target

## 5. 考虑beq 指令（条件转移指令的代表）





# 数据通路的设计——分支（条件转移）指令（beq）



立即数用补码表示

## • beq rs, rt, imm16

- M[PC] #取指令（公共操作，取指部件完成）
- $\text{Cond} \leftarrow R[\text{rs}] - R[\text{rt}]$  #做减法比较rs和rt中的内容
- if (COND eq 0) #计算下地址(根据比较结果，修改PC)
  - $\text{PC} \leftarrow \text{PC} + 4 + (\text{SignExt}(\text{imm16}) \times 4)$
- else
  - $\text{PC} \leftarrow \text{PC} + 4$

思考：立即数的含义是什么？是相对指令数还是相对单元数？(指令数！)

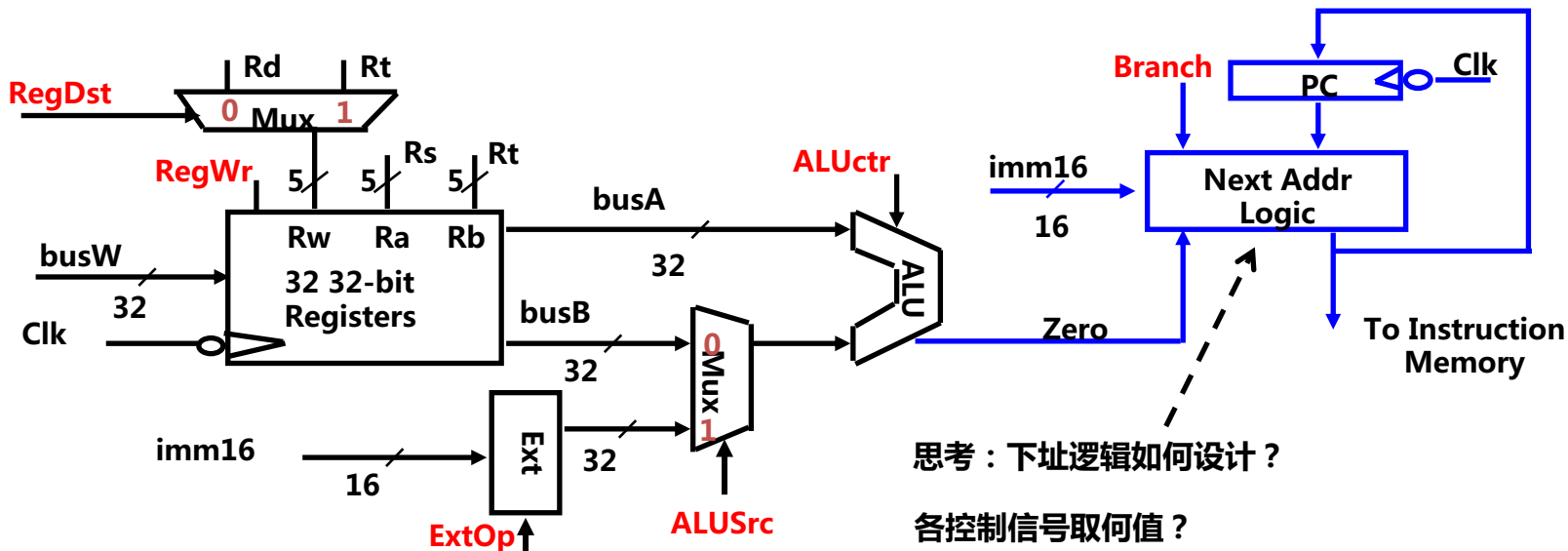
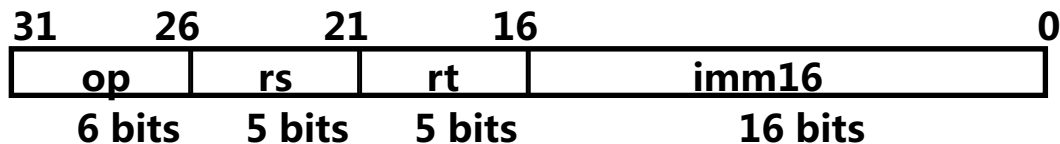
应在原数据通路上加哪些元件和连线？用什么控制信号来控制？





# 数据通路的设计——条件转移指令的数据通路

• **beq** *rs, rt, imm16*



**RegDst=x, RegWr=0, ALUctr=subu, ExtOp=x, ALUSrc=0, MemWr=0, MemtoReg=x, Branch=1**



# 数据通路的设计——下地址计算逻辑的设计

- **PC是一个32位地址:**
  - **顺序执行时:**  $PC\langle 31:0 \rangle = PC\langle 31:0 \rangle + 4$
  - **转移执行时:**  $PC\langle 31:0 \rangle = PC\langle 31:0 \rangle + 4 + \text{SignExt}[\text{Imm16}] \times 4$  (指令条数  $\times 4$ )
- 采用32位PC时, 可用左移2位实现 “ $\times 4$ ” 操作, 计算转移地址用2个加法器! 可以用更简便的方式实现如下:
  - MIPS按字节编址, 每条指令为32位, 占4个字节, 故PC的值总是4的倍数, 即后两位为00, 因此, PC只需要30位即可。
  - PC采用30位后, 其转移地址计算逻辑变得更加简单。
- **下地址计算逻辑简化为:**
  - **顺序执行时:**  $PC\langle 31:2 \rangle = PC\langle 31:2 \rangle + 1$
  - **转移执行时:**  $PC\langle 31:2 \rangle = PC\langle 31:2 \rangle + 1 + \text{SignExt}[\text{Imm16}]$
  - **取指令时:** 指令地址 =  $PC\langle 31:2 \rangle$  串接 “00”

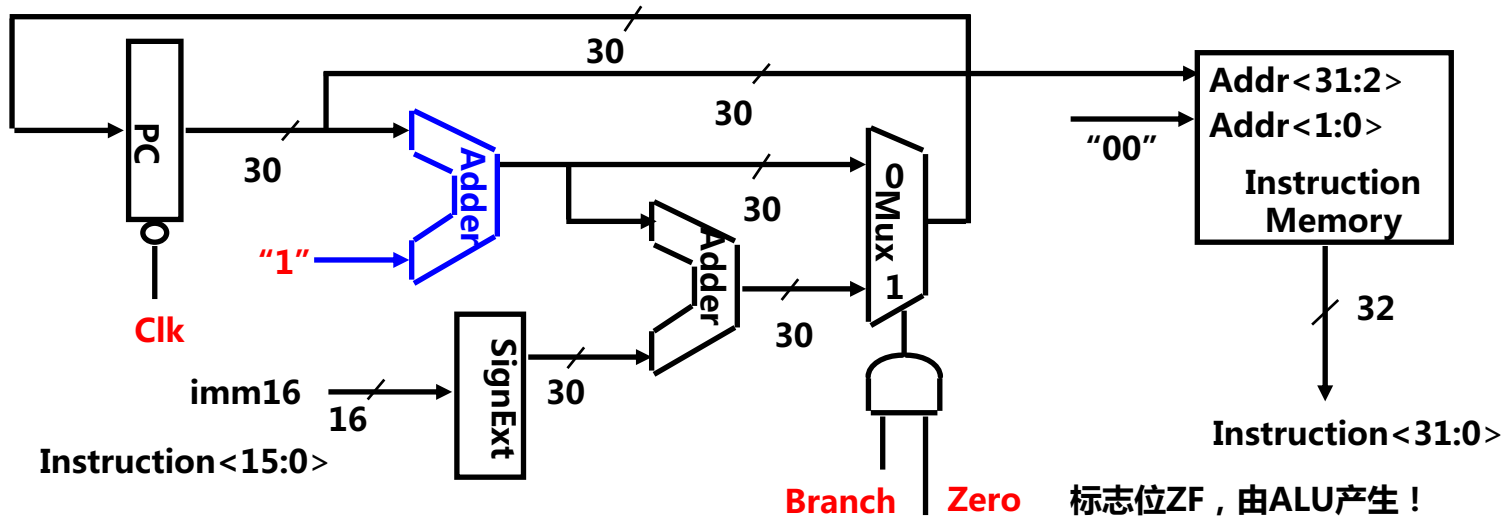




# 数据通路的设计——下地址计算逻辑的设计

- 使用30比特PC:

- 顺序执行时:  $PC\langle 31:2 \rangle = PC\langle 31:2 \rangle + 1$
- 转移执行时:  $PC\langle 31:2 \rangle = PC\langle 31:2 \rangle + 1 + \text{SignExt}[\text{Imm16}]$
- 取指令时: 指令地址 =  $PC\langle 31:2 \rangle$  串接 "00"



先根据当前PC取指令, 计算的下条指令地址在下一个时钟到来后才能写入PC!

为什么这里没有用“ALU”而是用“Adder”? “ALU”和“Adder”有什么差别?



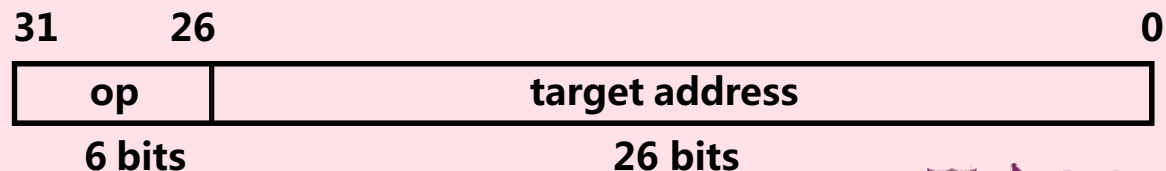
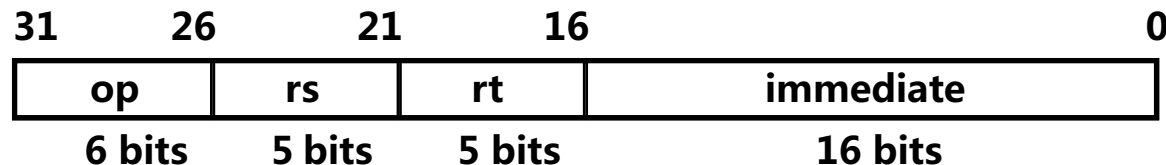
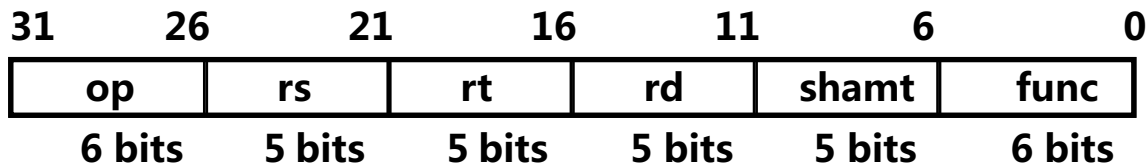


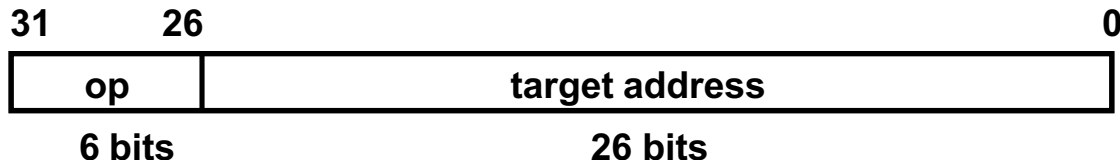
# 数据通路的设计——无条件转移指令

## 实现目标（7条指令）：

- **ADD and subtract**
  - add rd, rs, rt
  - sub rd, rs, rt
- **OR Immediate:**
  - ori rt, rs, imm16
- **LOAD and STORE**
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- **BRANCH:**
  - beq rs, rt, imm16
- **JUMP:**
  - j target

## 6. 考虑Jump指令（无条件转移指令的代表）





- **j target**

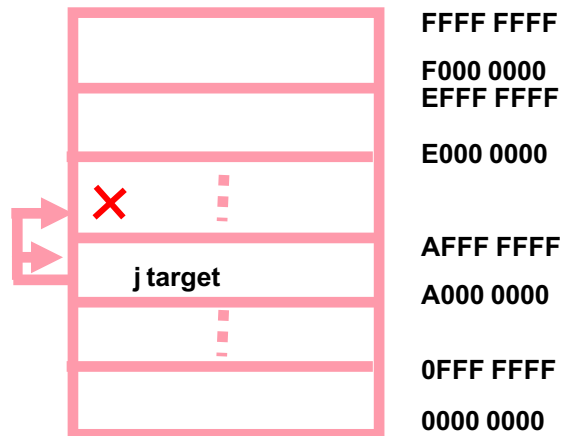
- M[PC] #取指令（公共操作，取指部件完成）
- $PC_{\langle 31:2 \rangle} \leftarrow PC_{\langle 31:28 \rangle} \text{ 串接 } target_{\langle 25:0 \rangle}$  #计算目标地址

### 想一想：跳转指令的转移范围有多大？

是当前指令后面的0x000 0000~0xFFF FFFC 处？

## 不对！它不是相对寻址，而是绝对寻址

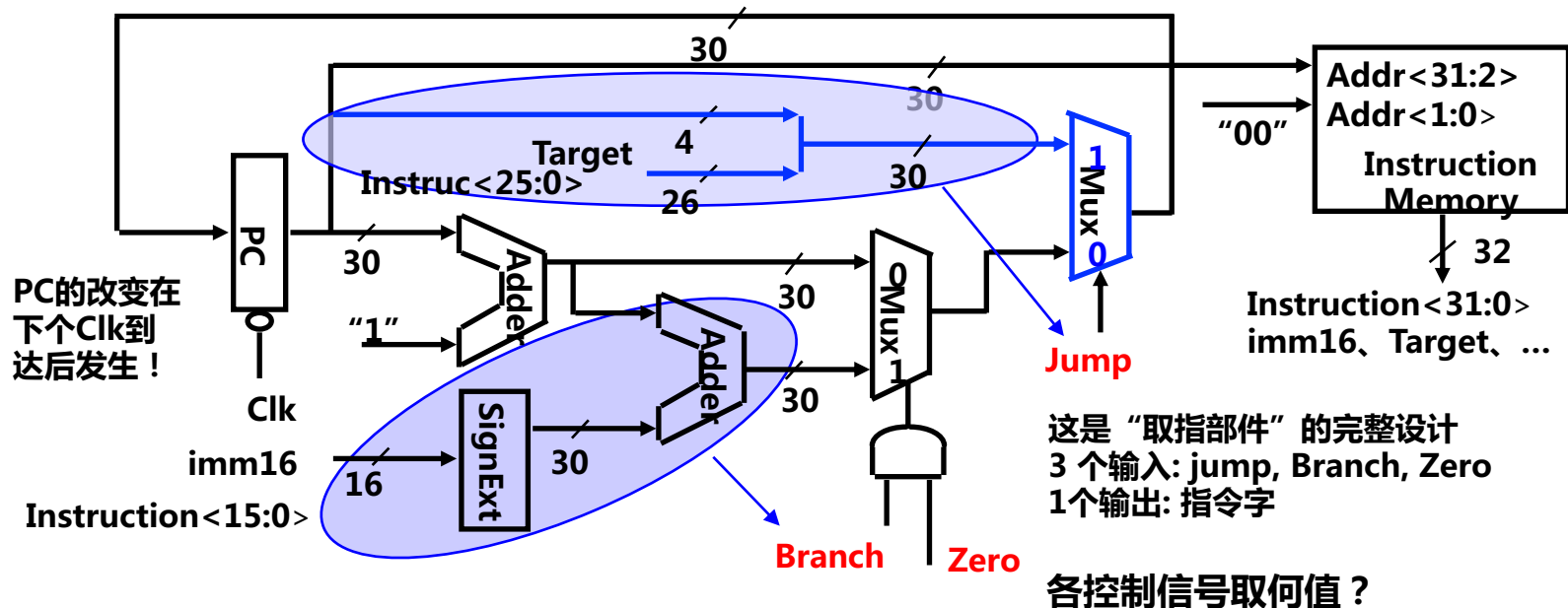
**思考：应在原数据通路上加哪些元件和连线？  
用什么控制信号来控制？**







- **j target**
  - **PC<31:2> ← PC<31:28> concat target<25:0>**



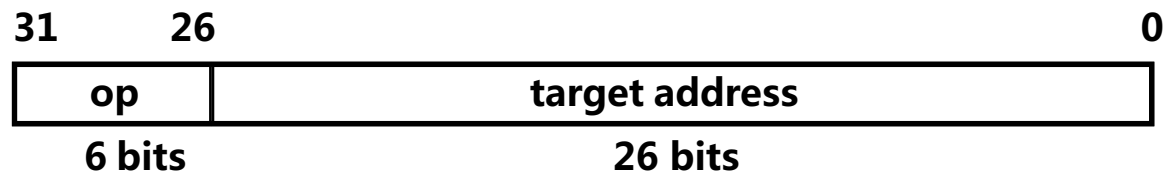
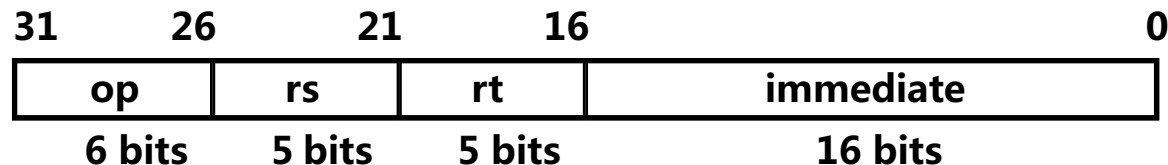
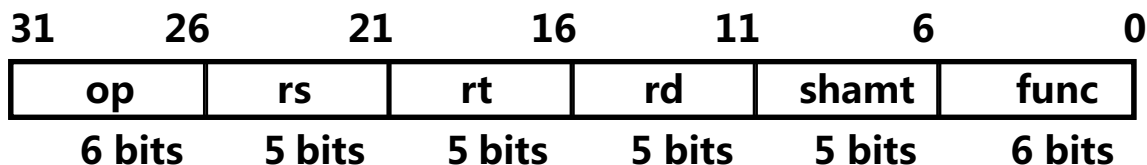
**RegDst=ExtOp=ALUSrc=MemtoReg=ALUctr=x, RegWr=0, MemWr=0, Branch=0, Jump=1**



# 数据通路的设计——7条指令

## 实现目标（7条指令）：

- **ADD and subtract**
  - add rd, rs, rt
  - sub rd, rs, rt
- **OR Immediate:**
  - ori rt, rs, imm16
- **LOAD and STORE**
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- **BRANCH:**
  - beq rs, rt, imm16
- **JUMP:**
  - j target



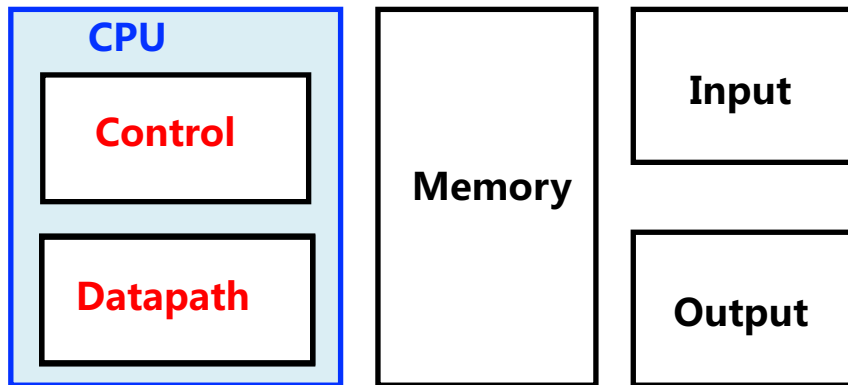
所有指令的数据通路都已设计好，合起来的数据通路是什么样的？





# 控制器的设计

- 计算机的五大组成部分



- 下一个目标：**设计单周期数据通路的控制器。**
- 设计方法：
  - 1) 根据每条指令的功能，分析控制信号的取值，并在表中列出。
  - 2) 根据列出的指令和控制信号的关系，写出每个控制信号的逻辑表达式。

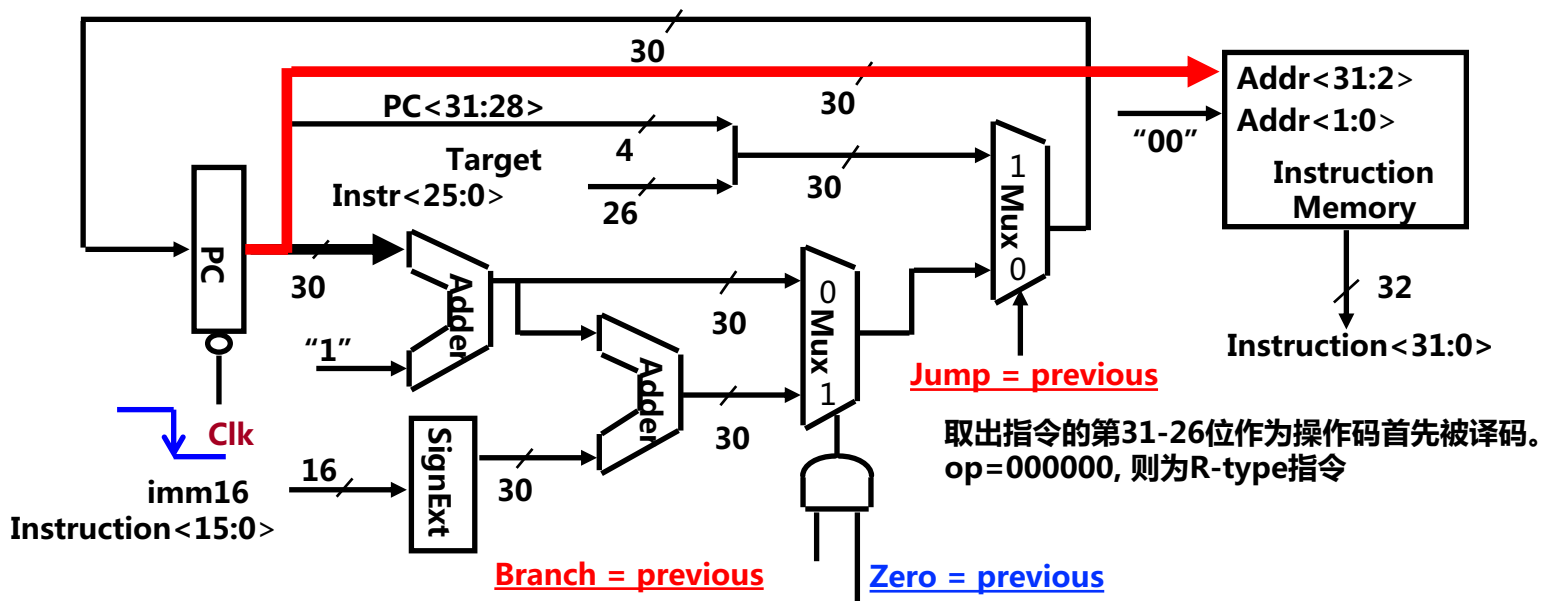




## 一 所有指令都相同

**新指令还没有取出译码，所以控制信号的值还是原来指令的旧值。**

**新指令还没有执行，所以标志也为旧值。**



## 取指部件由旧控制信号控制，会不会有问题？

## 没有问题！为什么？

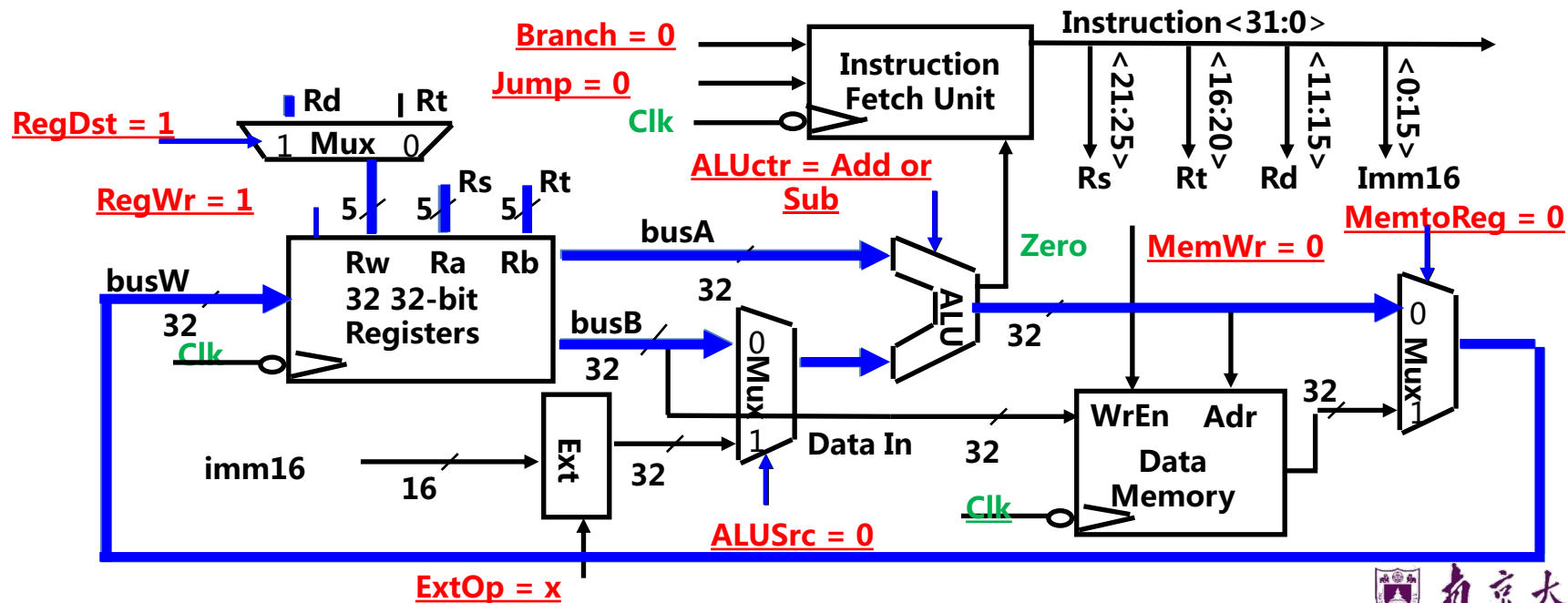
**因为在下个Clk到来之前PC输入端的值不会写入，只要保证下个Clk来之前能产生正确的PC即可！**



# 控制器的设计——指令译码后R型指令(Add/Sub)操作过程

- $R[rd] \leftarrow R[rs] + / - R[rt]$

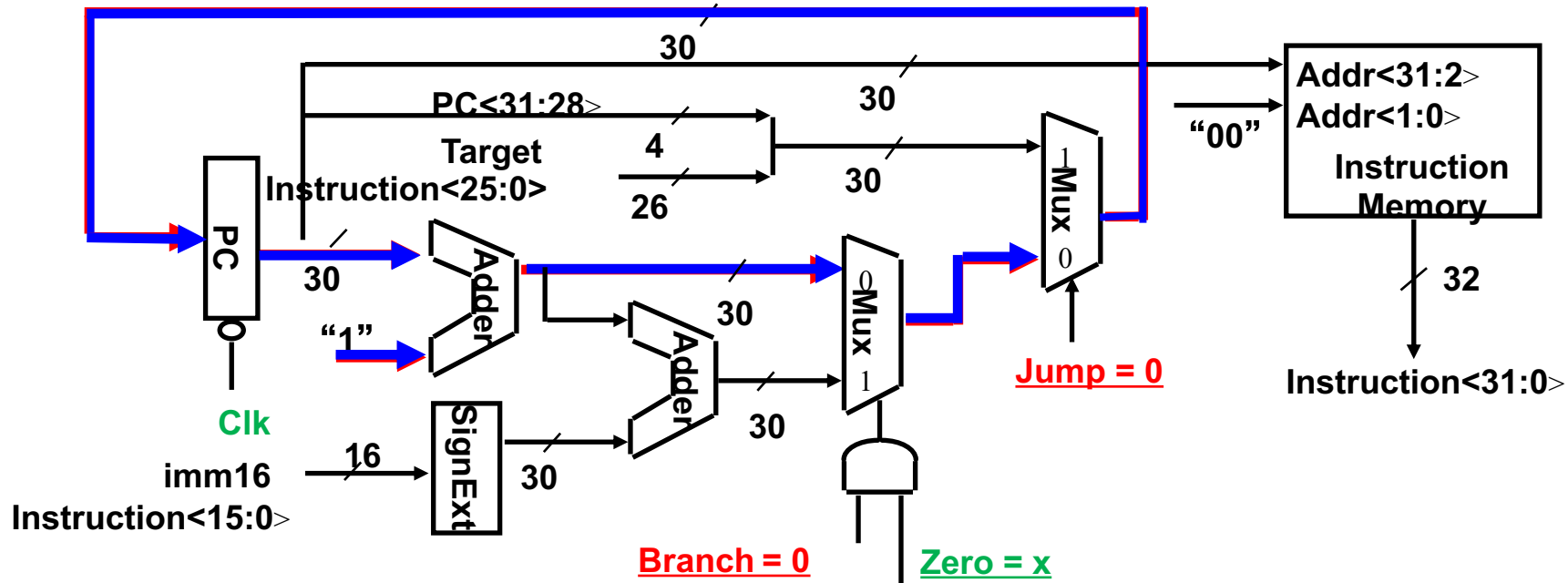
31	26	21	16	11	6	0
op	rs	rt	rd	shamt	func	





# 控制器的设计——R型指令(Add/Sub)最后阶段取指部件中的动作

- $PC \leftarrow PC + 4$ 
  - 除 Branch and Jump以外的指令都相同



因为新的控制信号保证了正确的PC值的产生，在足够长的时间后，下个时钟Clk到来！

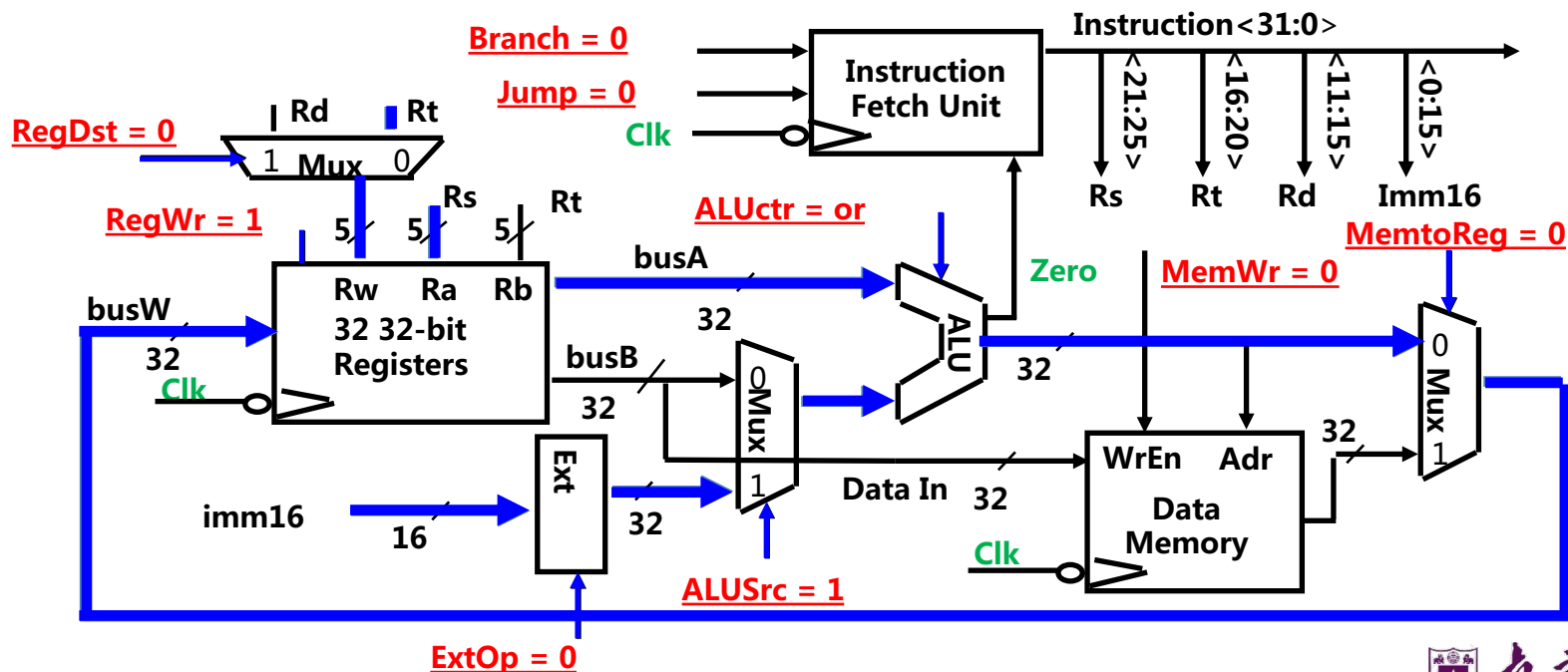
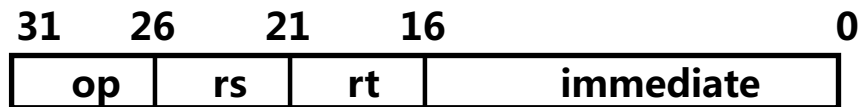






# 控制器的设计——ori 指令译码后的执行过程

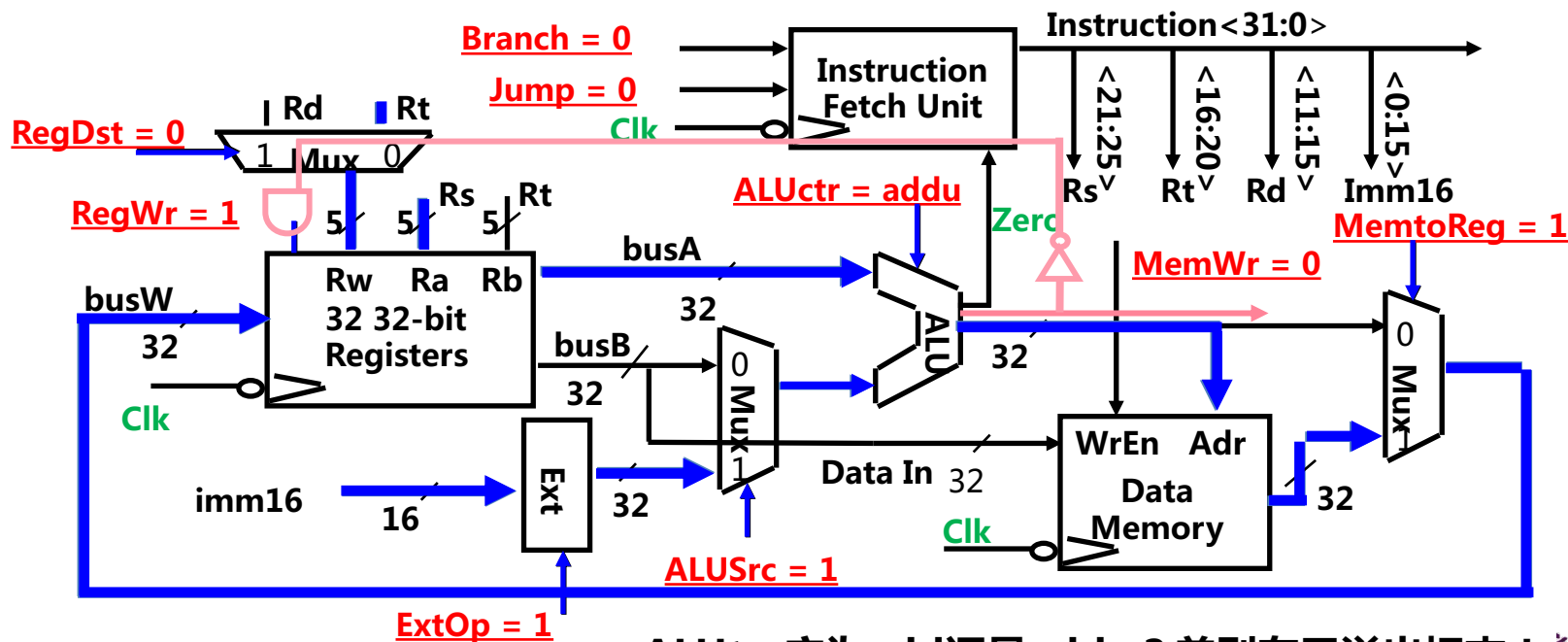
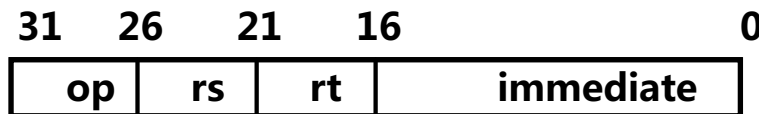
- $R[rt] \leftarrow R[rs] \text{ or } \text{ZeroExt}[Imm16]$





# 控制器的设计——Load指令译码后的执行过程

- $R[rt] \leftarrow \text{Data Memory } \{R[rs] + \text{SignExt}[\text{imm16}]\}$

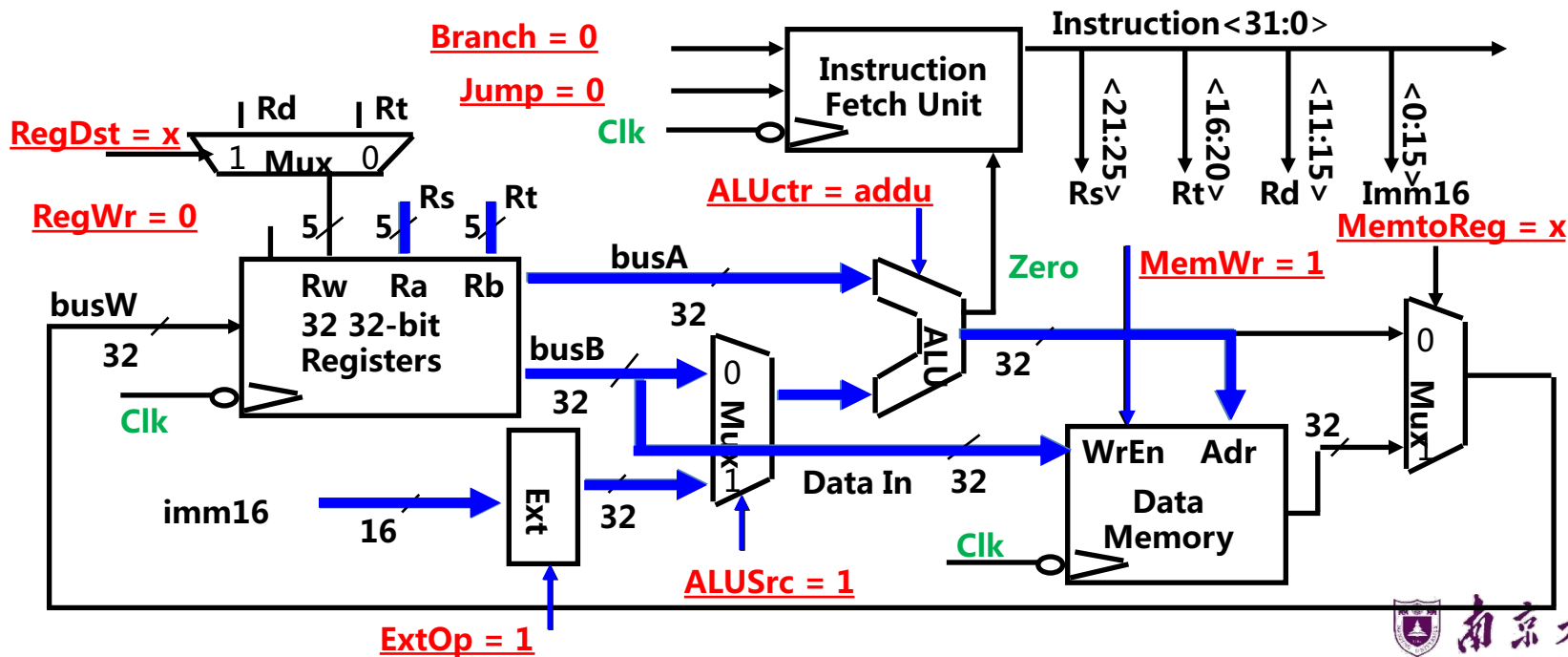
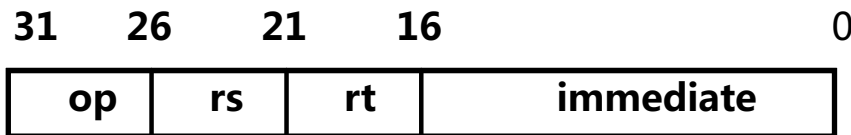


ALUctr应为add还是addu？差别在于溢出标志！



# 控制器的设计——Store指令译码后的执行过程

- $M\{R[rs] + \text{SignExt}[imm16]\} \leftarrow R[rt]$





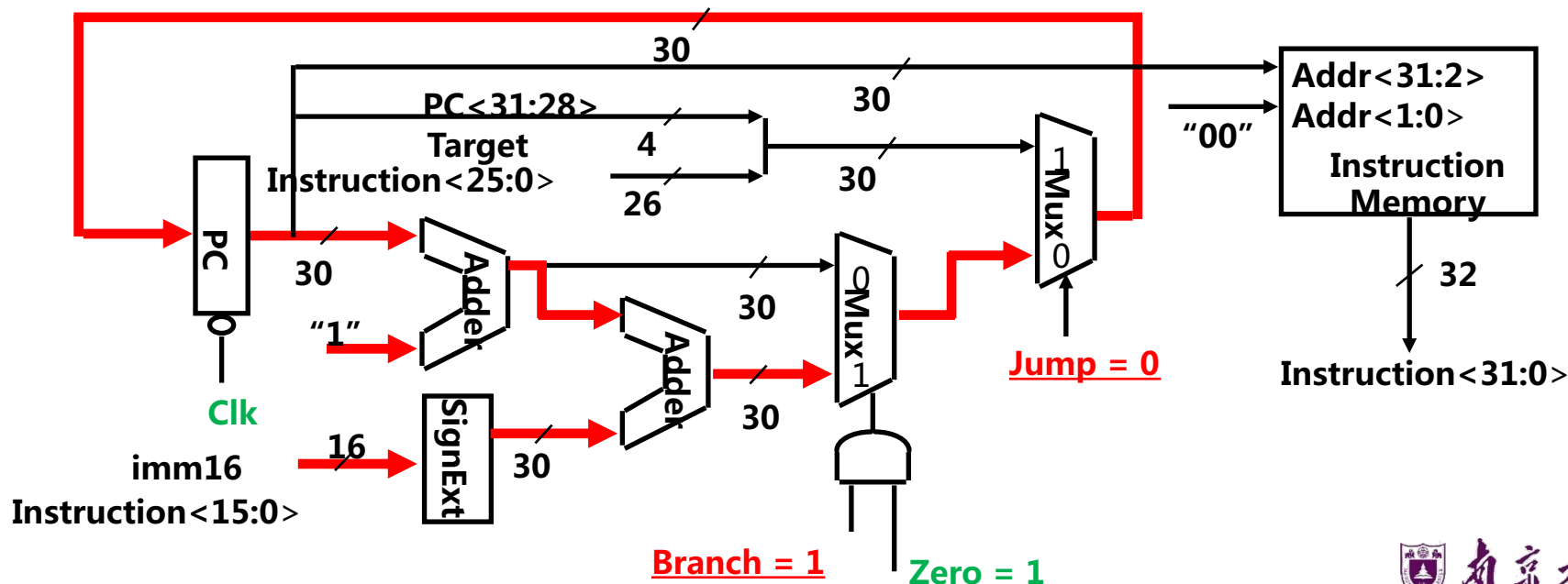
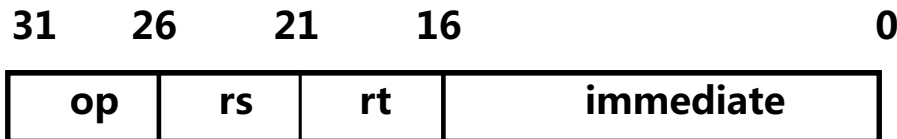
- if ( $R[rs] - R[rt] == 0$ )  
then  $Zero \leftarrow 1$ ; else  $Zero \leftarrow 0$





# 控制器的设计——Branch指令最后阶段取指部件中的动作

if (Zero == 1)  
then  $PC = PC + 4 + \text{SignExt}[\text{imm16}] * 4$  ;  
else  $PC = PC + 4$



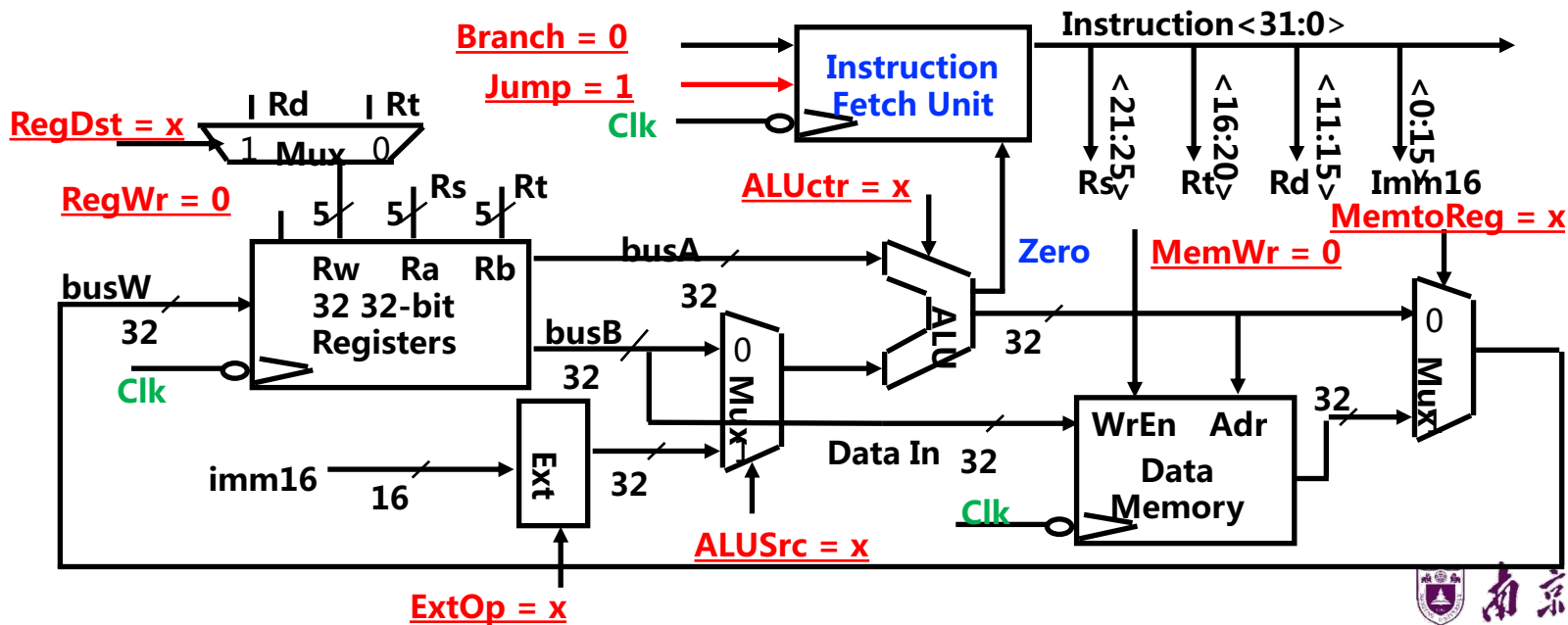


# 控制器的设计——Jump指令译码后的执行过程

- IFU中目标地址送PC，其他什么都不做

( 只要保证存储部件不发生写的动作 )

如何保证存储部件不发生写？ 使相应的写使能信号为0！





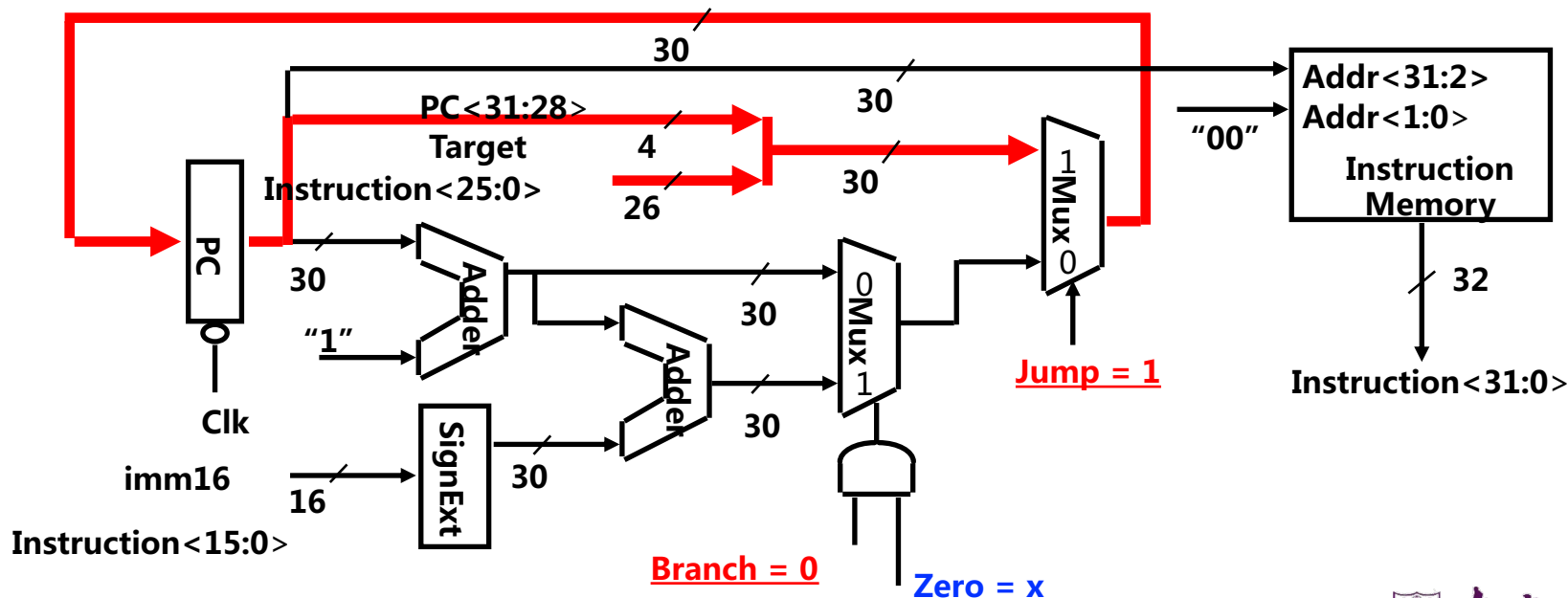
# 控制器的设计——Jump指令结束前取指令部件中的动作

- $PC \leftarrow PC\langle 31:29 \rangle \text{ concat target}\langle 25:0 \rangle$  串接 "00"

31 26

0

op	target address
----	----------------





# 控制器的设计——指令与控制信号的关系表

	func	10 0000	10 0010	与func字段无关!			
	op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100 00 0010
		add	sub	ori	lw	sw	beq jump
RegDst		1	1	0	0	x	x
ALUSrc		0	0	1	1	1	0
MemtoReg		0	0	0	1	x	x
RegWrite		1	1	1	1	0	0
MemWrite		0	0	0	0	1	0
Branch		0	0	0	0	0	1
Jump		0	0	0	0	0	0
ExtOp		x	x	0	1	1	x
ALUctr<2:0>		add	sub	or	addu	addu	subu xxx

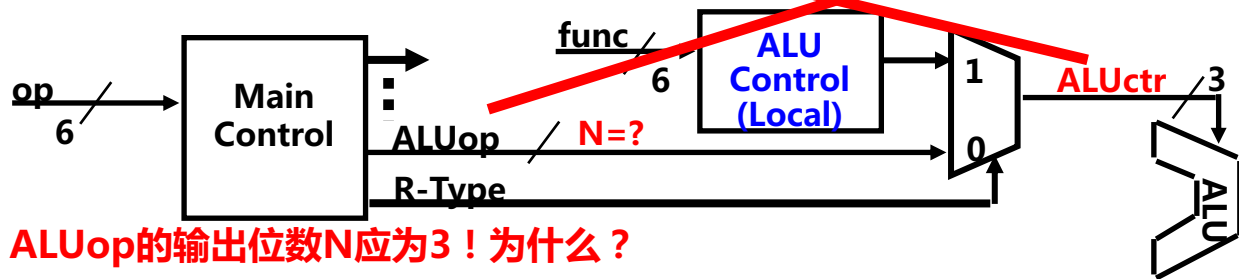
	31	26	21	16	11	6	0	
R-type	op		rs	rt	rd	shamt	func	add, sub
I-type	op		rs	rt	immediate			ori, lw, sw, beq
J-type	op		target address					jump





# 控制器的设计——主控制单元与ALU局部控制单元

op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUctr	Add/Sub	Or	Addu	Addu	Subu	xxx



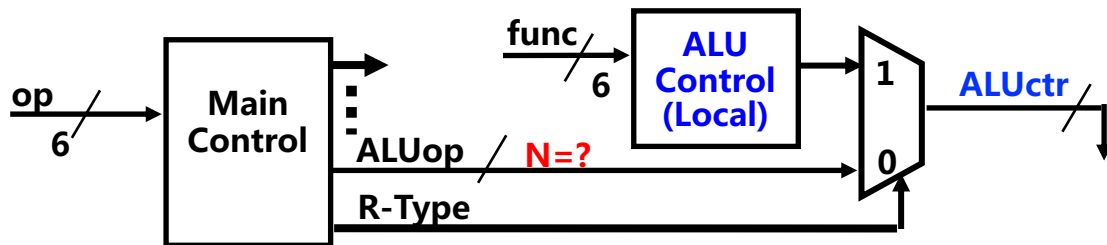
ALUop的输出位数N应为3！为什么？

ALUctr的值：

- 非R型指令时，取决于ALUop；
- R型指令时，取决于func。



# 控制器的设计——ALUop的逻辑表达式



ALUop的编码定义：

	000000	001101	100011	101011	000100	000010
指令	R-type	ori	lw	sw	beq	jump
运算	由func指定	or	addu	addu	subu	xxx
ALUop<2:0>	xxx 或 xx <b>1</b>	<b>0</b> 10	000	000	<b>1</b> 00	xxx

$ALUop<2> = beq = !op<5> \& !op<4> \& !op<3> \& op<2> \& !op<1> \& !op<0>$  (op=000100)

$ALUop<1> = ori = !op<5> \& !op<4> \& op<3> \& op<2> \& !op<1> \& op<0>$  (op=001101)

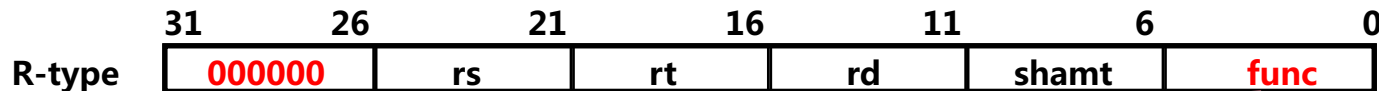
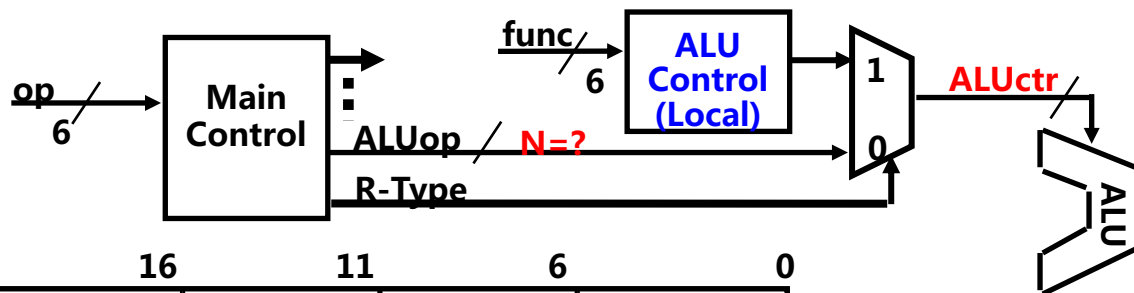
$ALUop<0> = R-type = !op<5> \& !op<4> \& !op<3> \& !op<2> \& !op<1> \& !op<0>$  (op=000000)

R型指令时，ALUctr与func有关，需建立ALUctr与func之间对应关系





# 控制器的设计——ALU局部控制器逻辑表达式



func<5:0>	Instruction Operation	ALUctr<2:0>	ALU Operation
10 0000	add	001	Add
10 0010	sub	101	Sub
10 0100	and	----	And
10 0101	or	010	Or
10 1010	set-on-less-than	111	Slt

$$ALUctr<0> = \neg func<3> \& \neg func<2> \& \neg func<1> \& \neg func<0> + \neg func<2> \& func<1> \& \neg func<0>$$

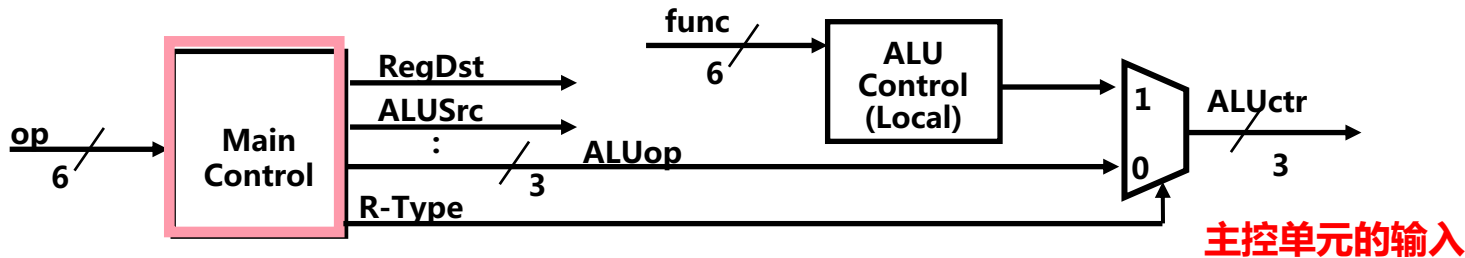
$$ALUctr<1> = \neg func<3> \& func<2> \& \neg func<1> \& func<0> + func<3> \& \neg func<2> \& func<1> \& \neg func<0>$$

$$ALUctr<2> = \neg func<2> \& func<1> \& \neg func<0>$$





# 控制器的设计——主控制单元的真值表



主控制单元的输出

op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALU运算	"R-type"	Or	Addu	Addu	Subu	xxx
ALUop <2>	x	0	0	0	1	x
ALUop <1>	x	1	0	0	0	x
ALUop <0>	1	0	0	0	0	x
R-type	1	0	0	0	0	0

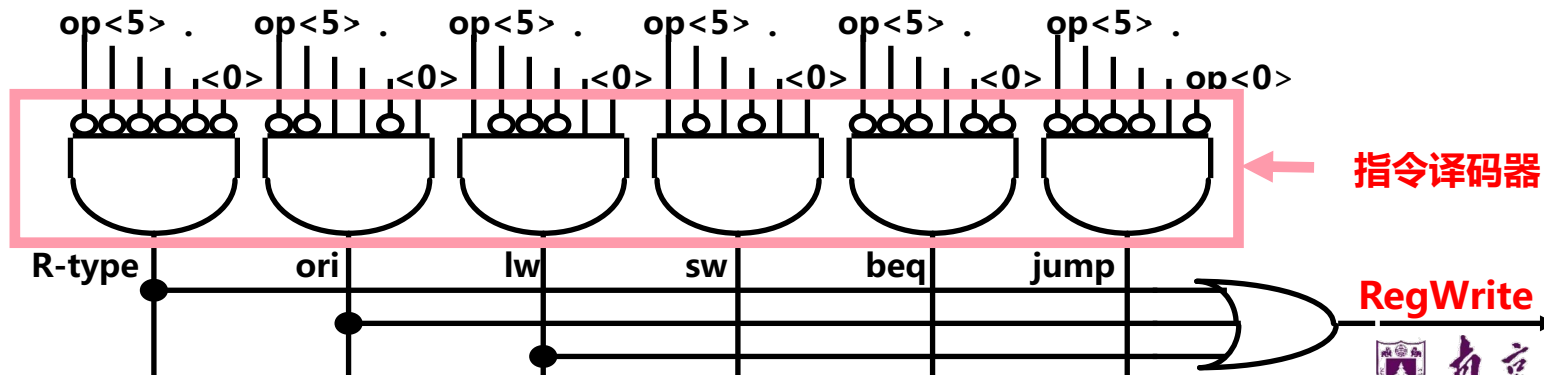




# 控制器的设计——控制信号的逻辑方程(如：RegWrite)

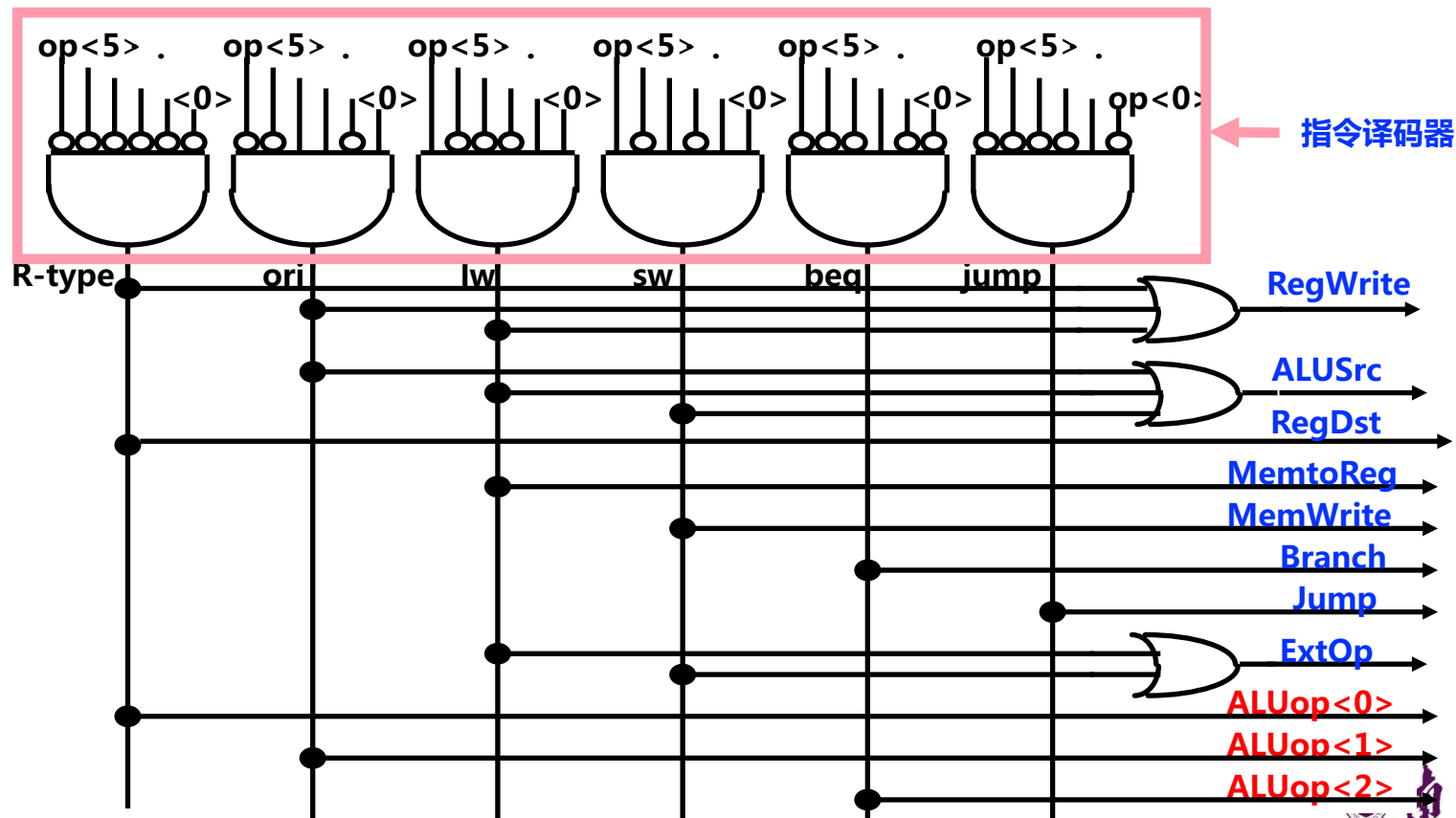
op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegWrite	1	1	1	0	0	0

- RegWrite = R-type + ori + lw  
= !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0> (R-type)  
+ !op<5> & !op<4> & op<3> & op<2> & !op<1> & op<0> (ori)  
+ op<5> & !op<4> & !op<3> & !op<2> & op<1> & op<0> (lw)



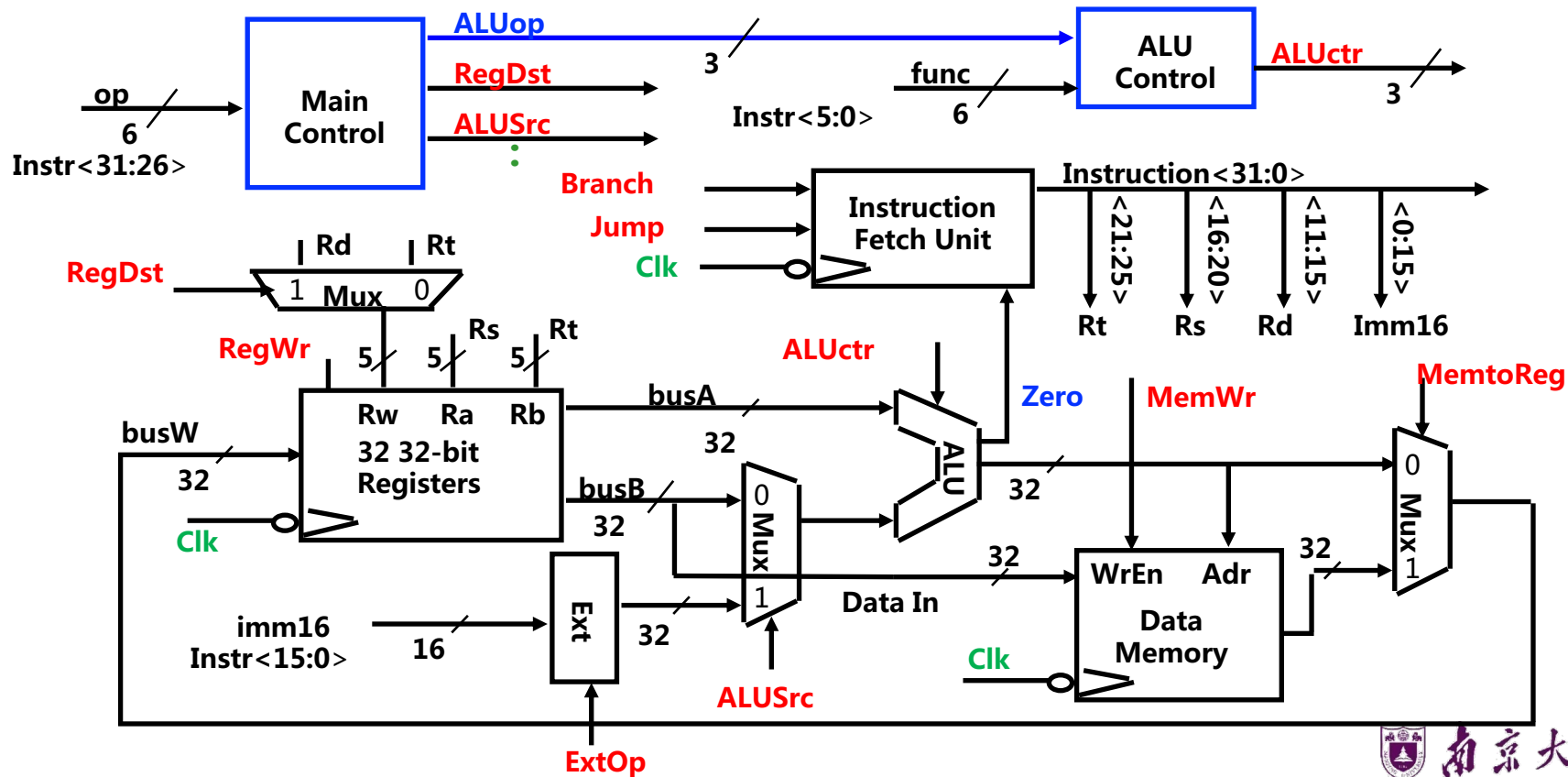


# 控制器的设计——主控制器的PLA实现





# 控制器的设计——执行前述7条指令的完整的单周期处理器





# 时钟周期的确定

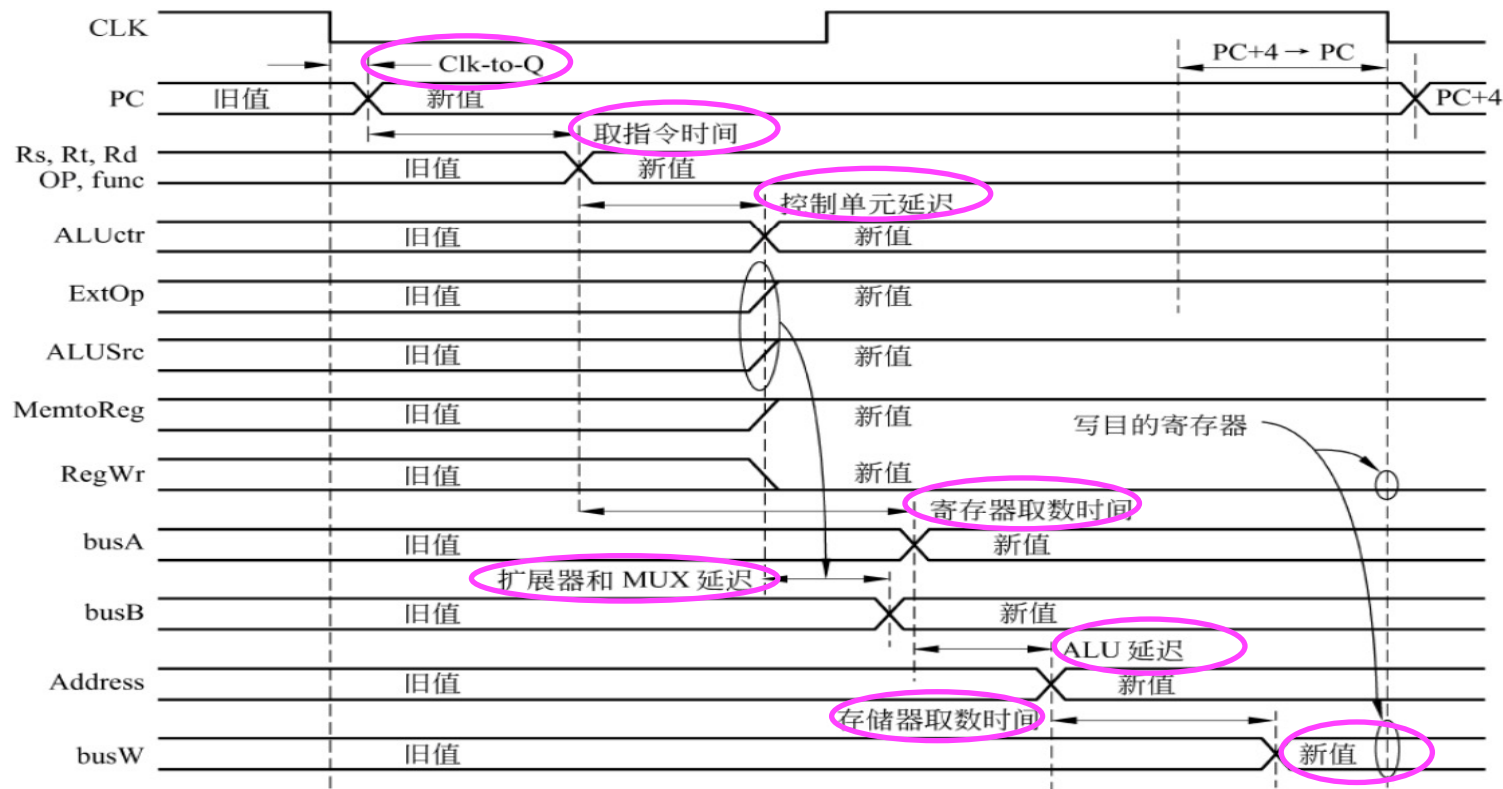


图 5.24 load 指令执行定时





# 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/>



南京大學  
NANJING UNIVERSITY