

# Computer Organization and Design (Fall 2013)

## **The Processor**

**Jiang zhong**

[zhongjiang@cqu.edu.cn](mailto:zhongjiang@cqu.edu.cn)

**QQ: 376917902**

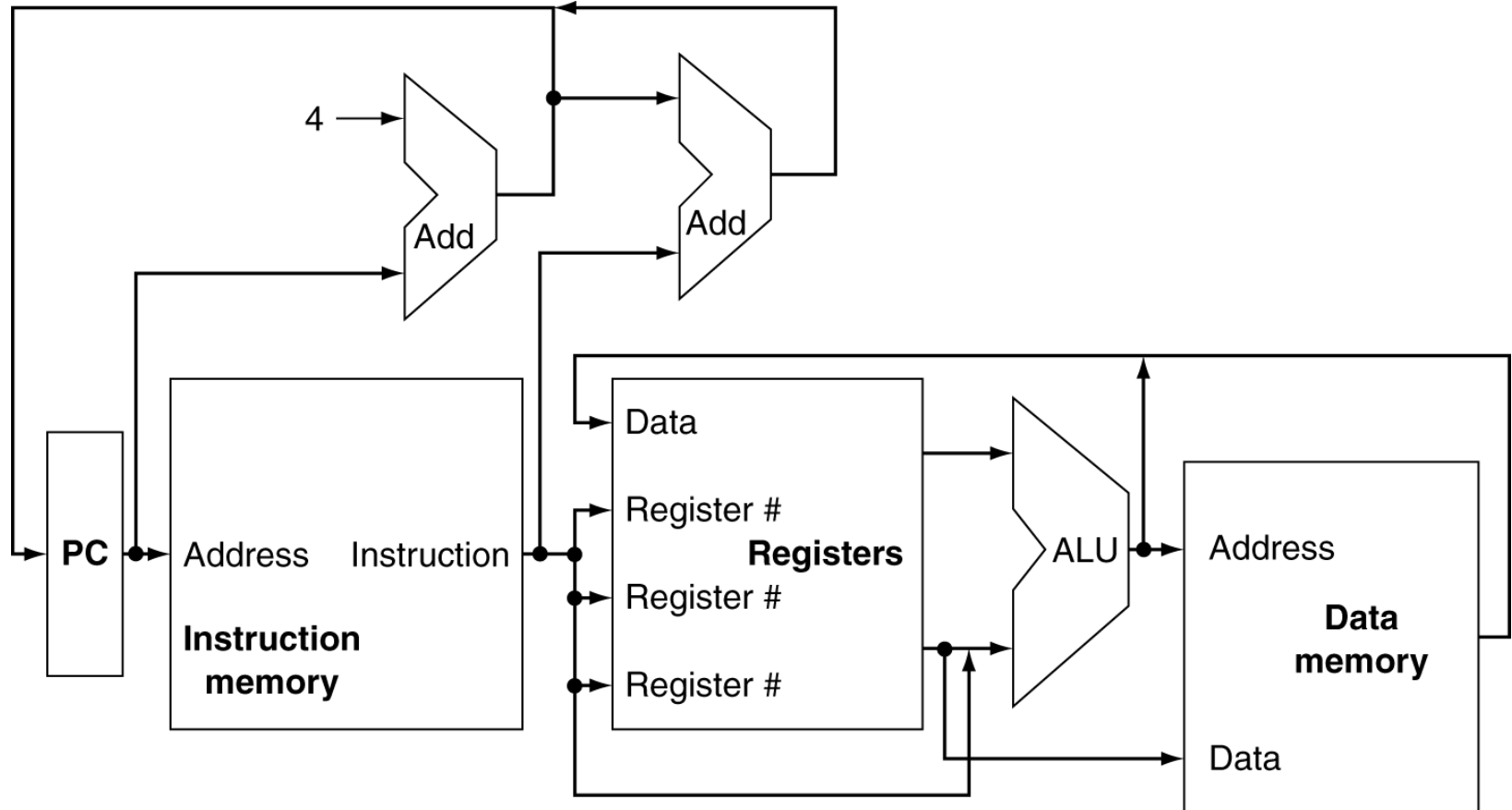
# Introduction

- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Clock cycle time
    - Determined by CPU hardware
- We will examine MIPS CPU implementations
  - A simplified version
- Simple instruction subset, shows most aspects
  - Memory reference: `lw`, `sw`
  - Arithmetic/logical: `add`, `sub`, `and`, `or`, `sll`, `t`
  - Control transfer: `beq`, `j`
- We will introduce additional materials about CPU

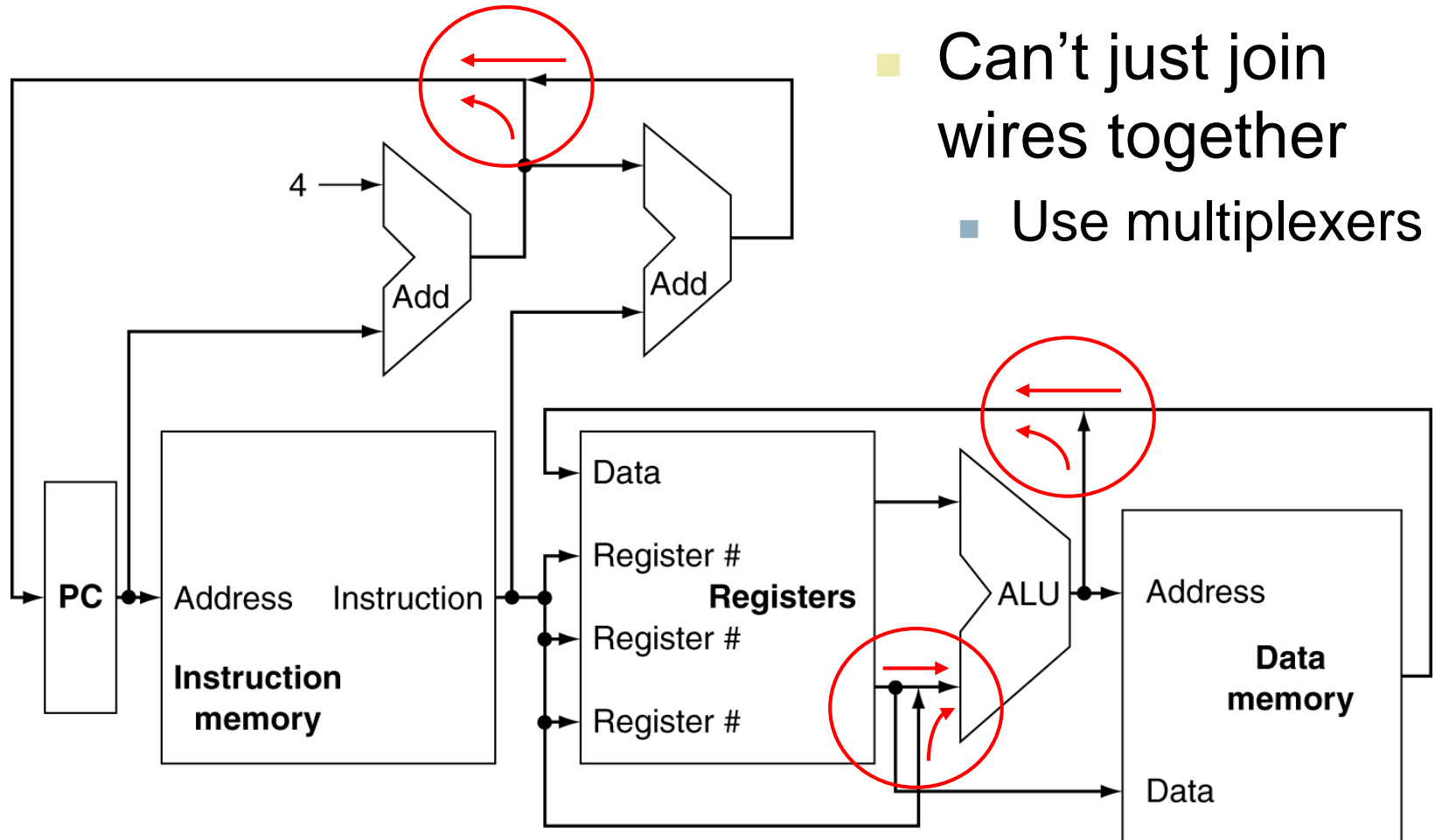
# Instruction Execution: Main Steps

- PC  $\rightarrow$  instruction memory, fetch instruction
- Register numbers  $\rightarrow$  register file, read registers
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch target address
  - Write result to register file (optional)
  - Access data memory for load/store (optional)
  - PC  $\leftarrow$  target address or PC + 4

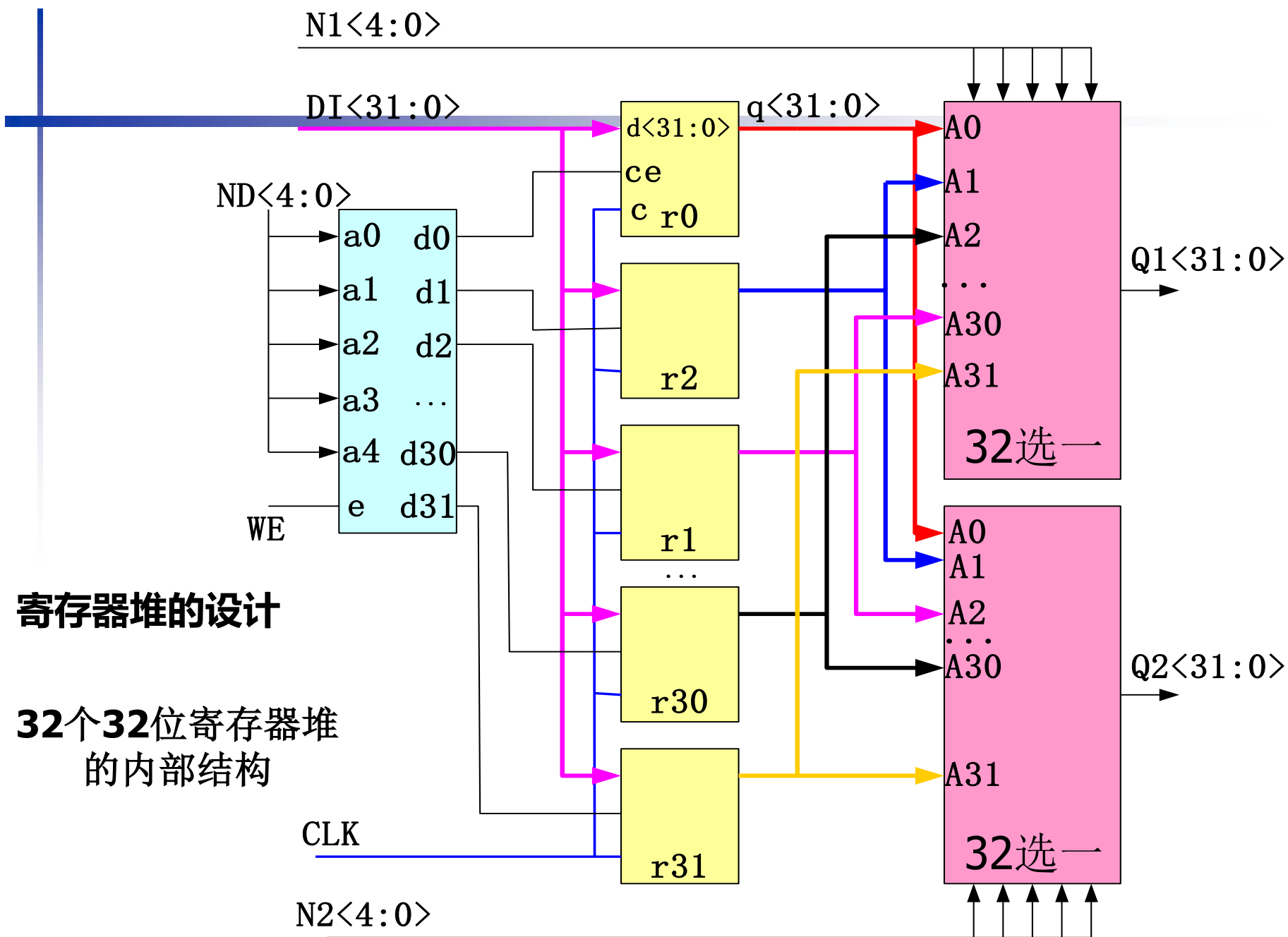
# CPU (the main frame) Overview



# Multiplexers



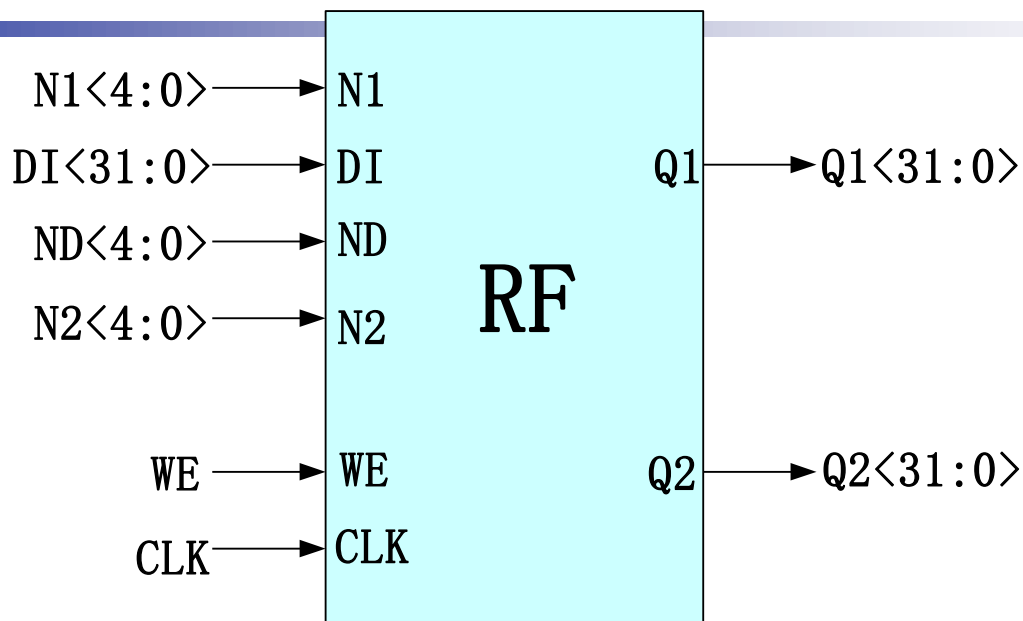
Reference: Appendix C



寄存器堆的设计

32个32位寄存器堆  
的内部结构

## 封装后

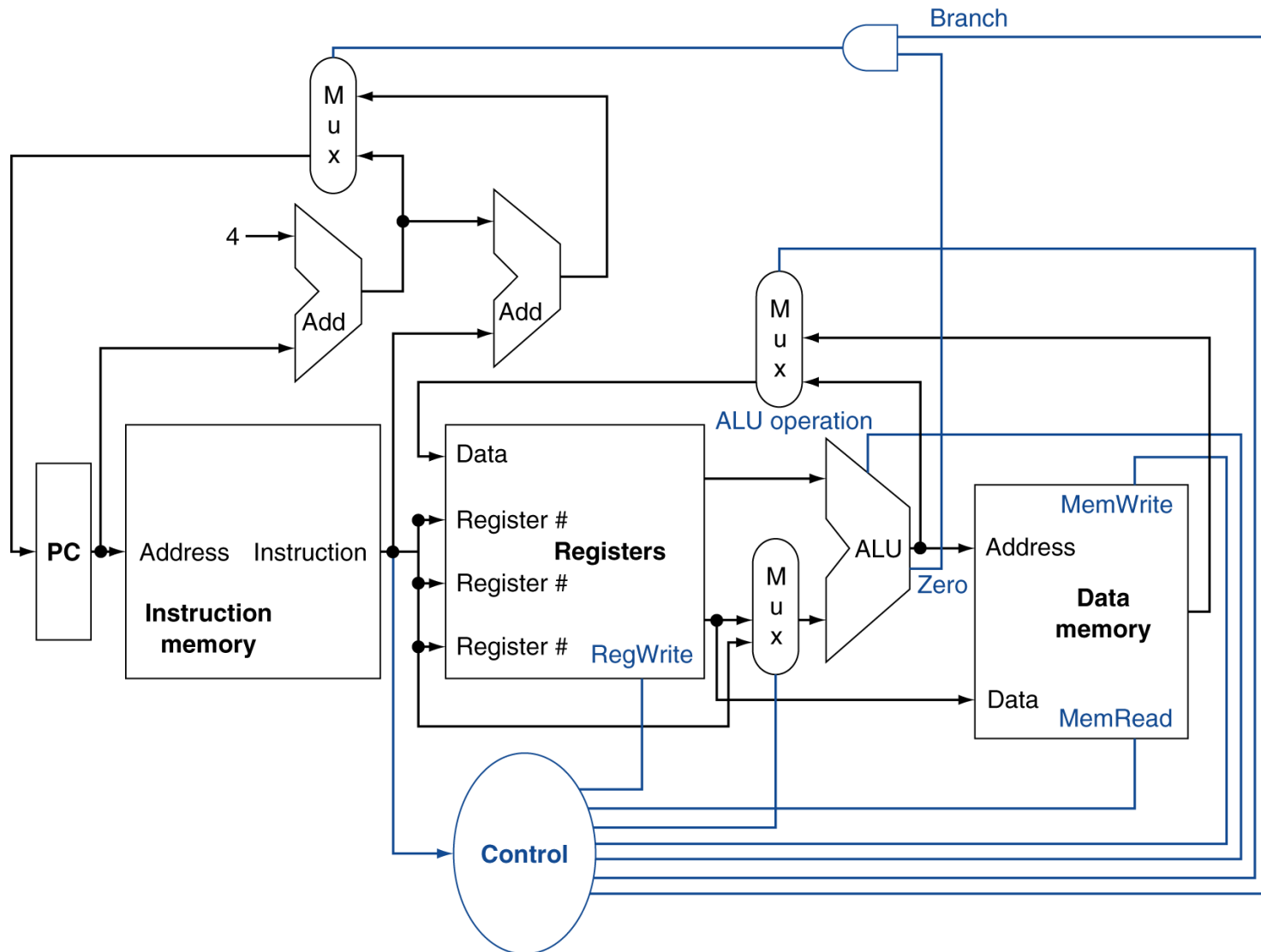


两个读端口和一个写端口的寄存器堆

$N1, N2$ —两个读端口的地址输入端，对应输出端为  $Q1$  和  $Q2$

$ND$ —为写端口的地址输入  
 $DI$ —为32位数据输入端

# Control





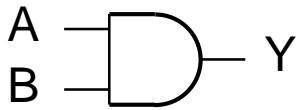
# Logic Design Basics

- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

# Combinational Elements

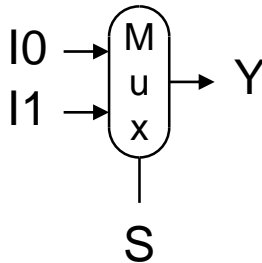
- AND-gate

- $Y = A \& B$



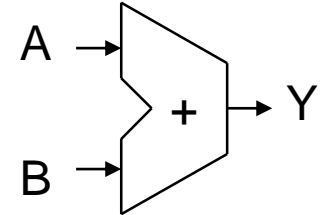
- Multiplexer

- $Y = S ? I1 : I0$



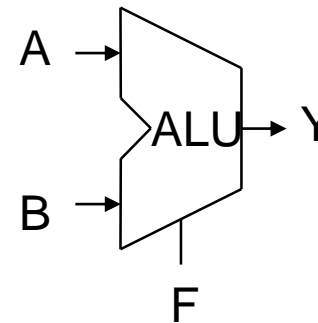
- Adder

- $Y = A + B$



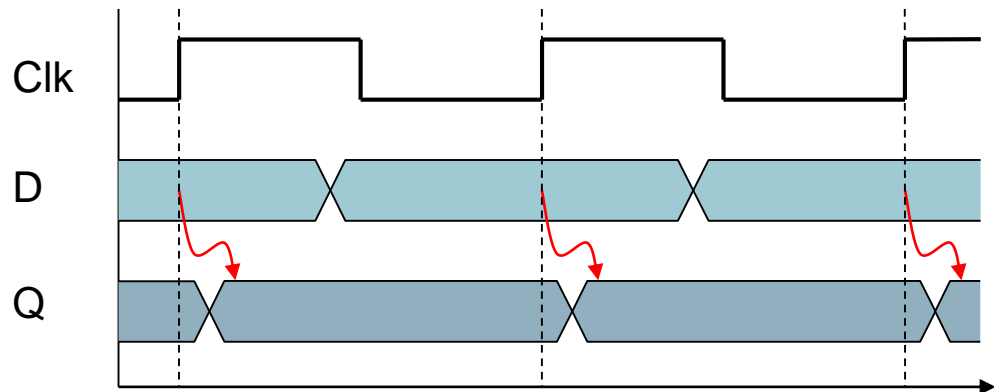
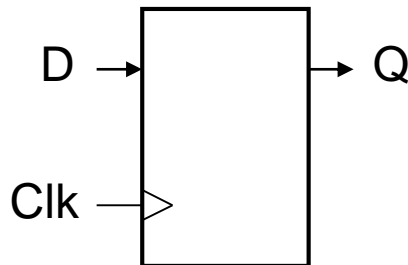
- Arithmetic/Logic Unit

- $Y = F(A, B)$



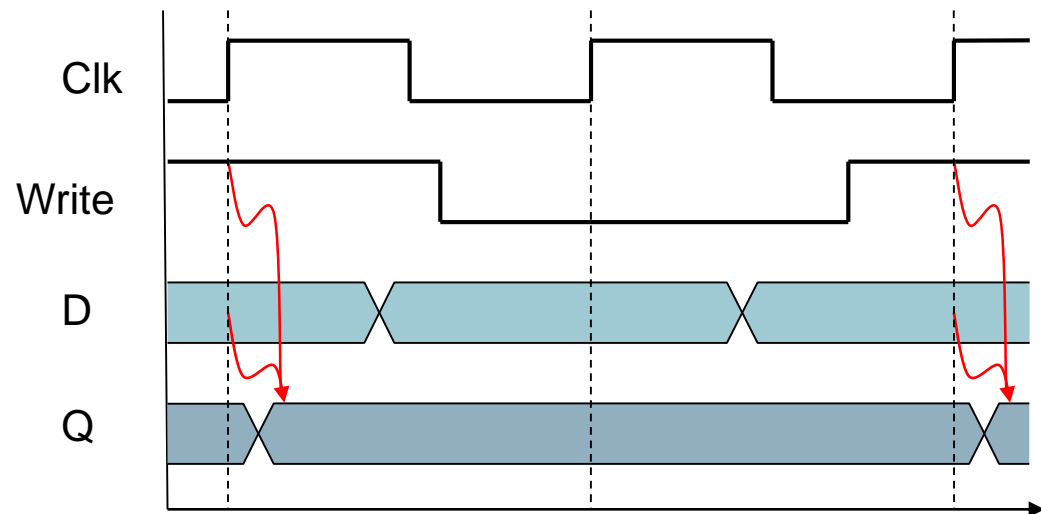
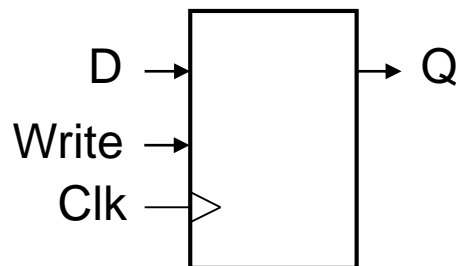
# Sequential Elements

- **Register:** stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1



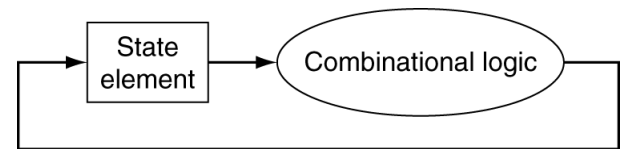
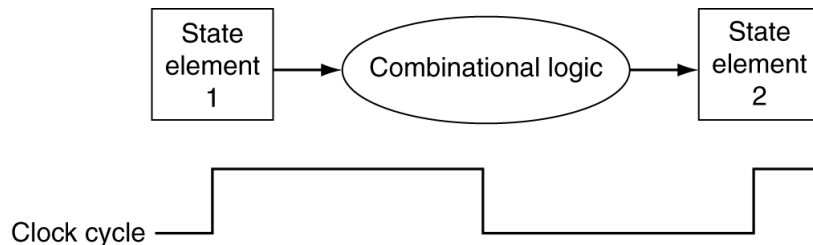
# Sequential Elements

- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later



# Clocking Methodology

- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period
    - Must keep “critical path” as short as possible



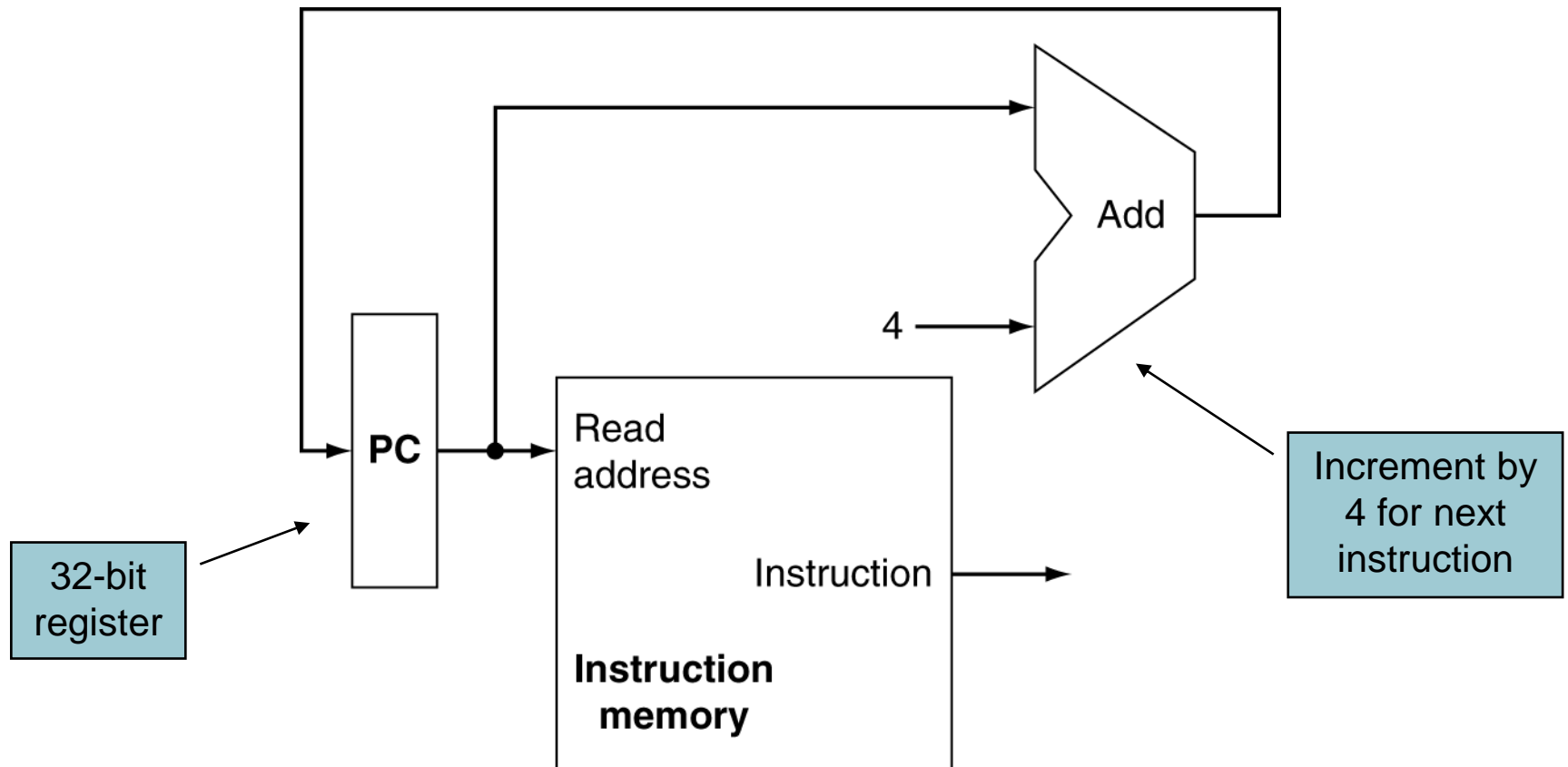
# The Basic MIPS Instruction Types

R-type	<table><tr><td>0</td><td>rs</td><td>rt</td><td>rd</td><td>shamt</td><td>funct</td></tr><tr><td>31:26</td><td>25:21</td><td>20:16</td><td>15:11</td><td>10:6</td><td>5:0</td></tr></table>	0	rs	rt	rd	shamt	funct	31:26	25:21	20:16	15:11	10:6	5:0
0	rs	rt	rd	shamt	funct								
31:26	25:21	20:16	15:11	10:6	5:0								
Load/ Store	<table><tr><td>35 or 43</td><td>rs</td><td>rt</td><td>address</td></tr><tr><td>31:26</td><td>25:21</td><td>20:16</td><td>15:0</td></tr></table>	35 or 43	rs	rt	address	31:26	25:21	20:16	15:0				
35 or 43	rs	rt	address										
31:26	25:21	20:16	15:0										
Branch	<table><tr><td>4</td><td>rs</td><td>rt</td><td>address</td></tr><tr><td>31:26</td><td>25:21</td><td>20:16</td><td>15:0</td></tr></table>	4	rs	rt	address	31:26	25:21	20:16	15:0				
4	rs	rt	address										
31:26	25:21	20:16	15:0										

# Building a Datapath

- **Datapath**
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, ...
- We will build a MIPS datapath incrementally
  - Refining the overview design

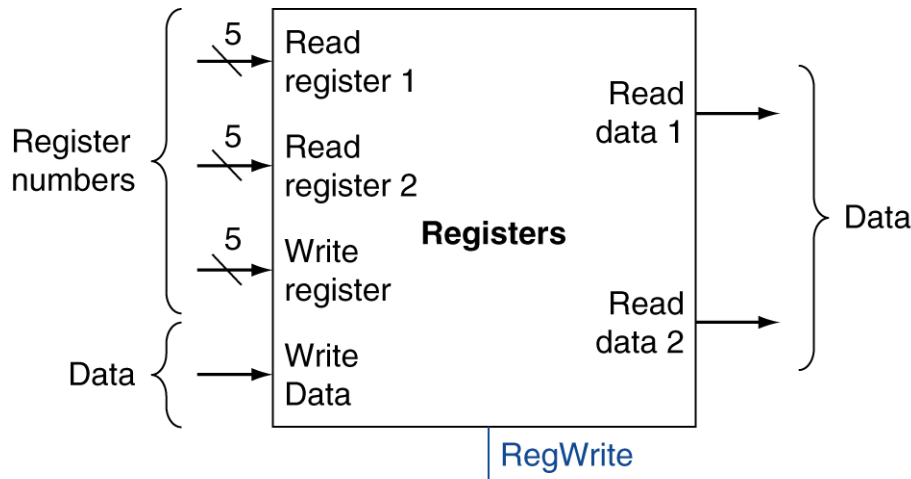
# Instruction Fetch



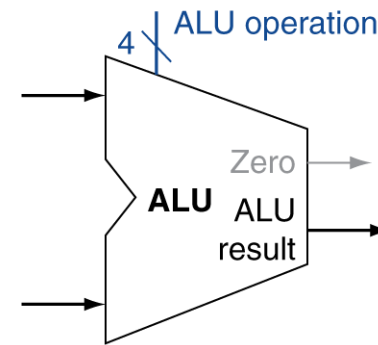


# R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result



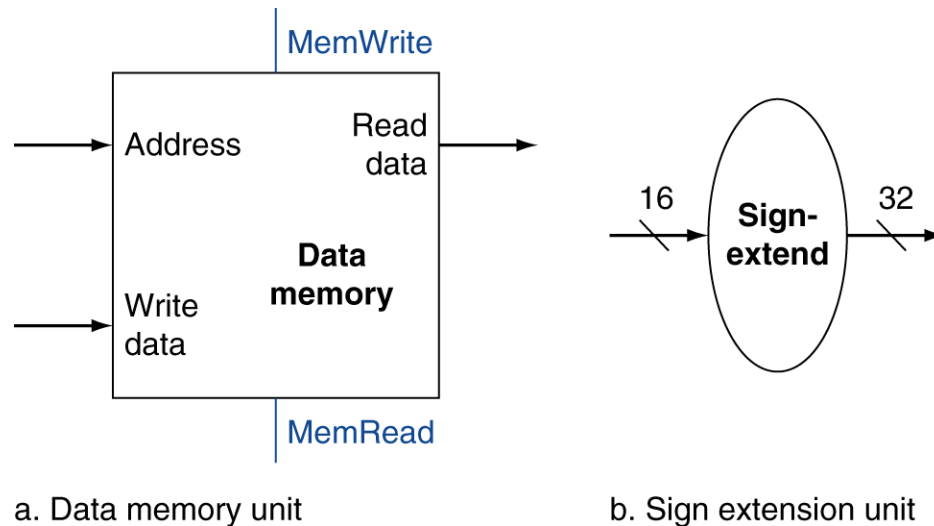
a. Registers



b. ALU

# Load/Store Instructions

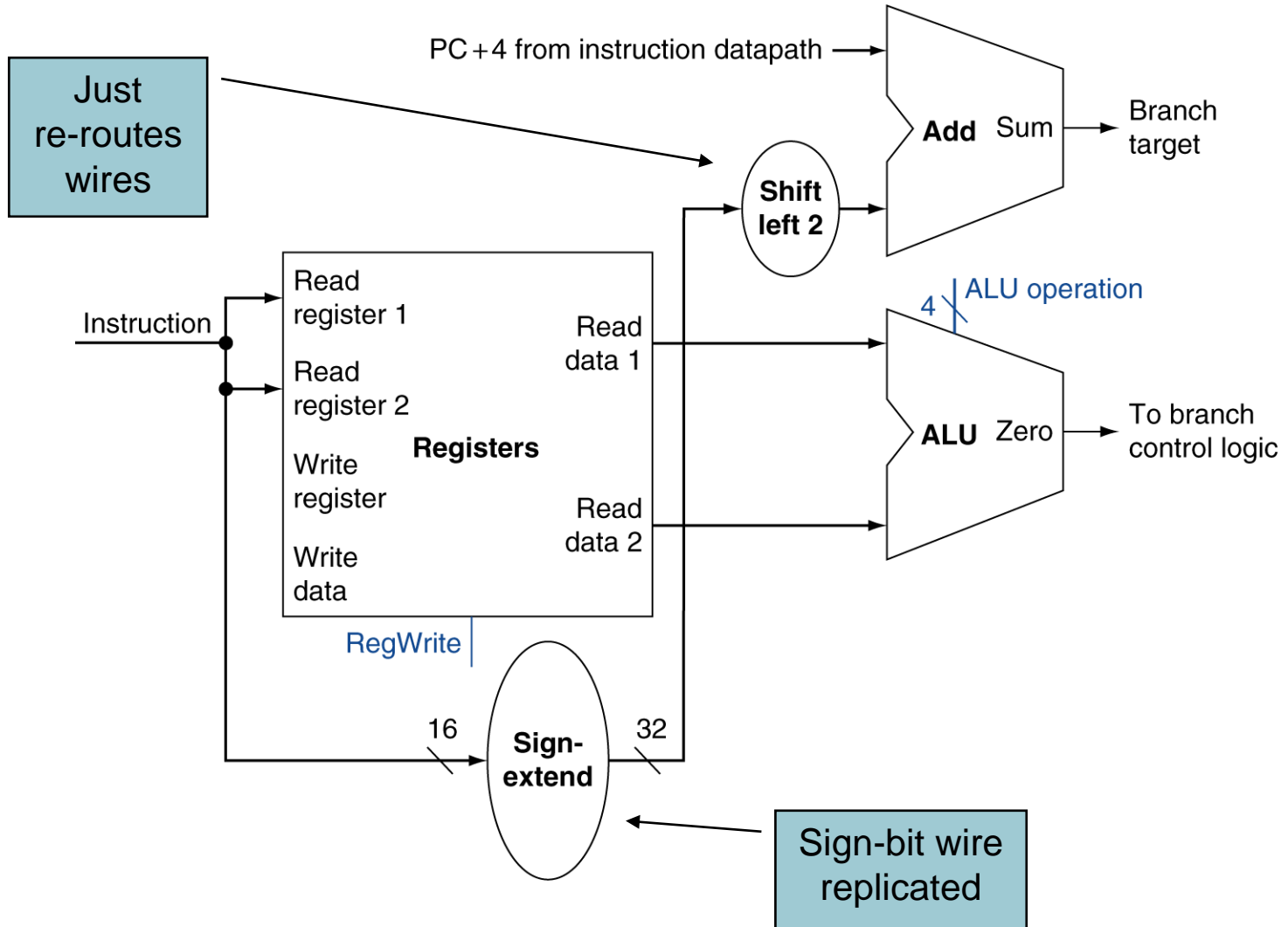
- Read register operands
- Calculate address using 16-bit offset
  - Use ALU, but sign-extend offset
- **Load**: Read memory and update register
- **Store**: Write register value to memory



# Branch Instructions

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to PC + 4
    - Already calculated by instruction fetch

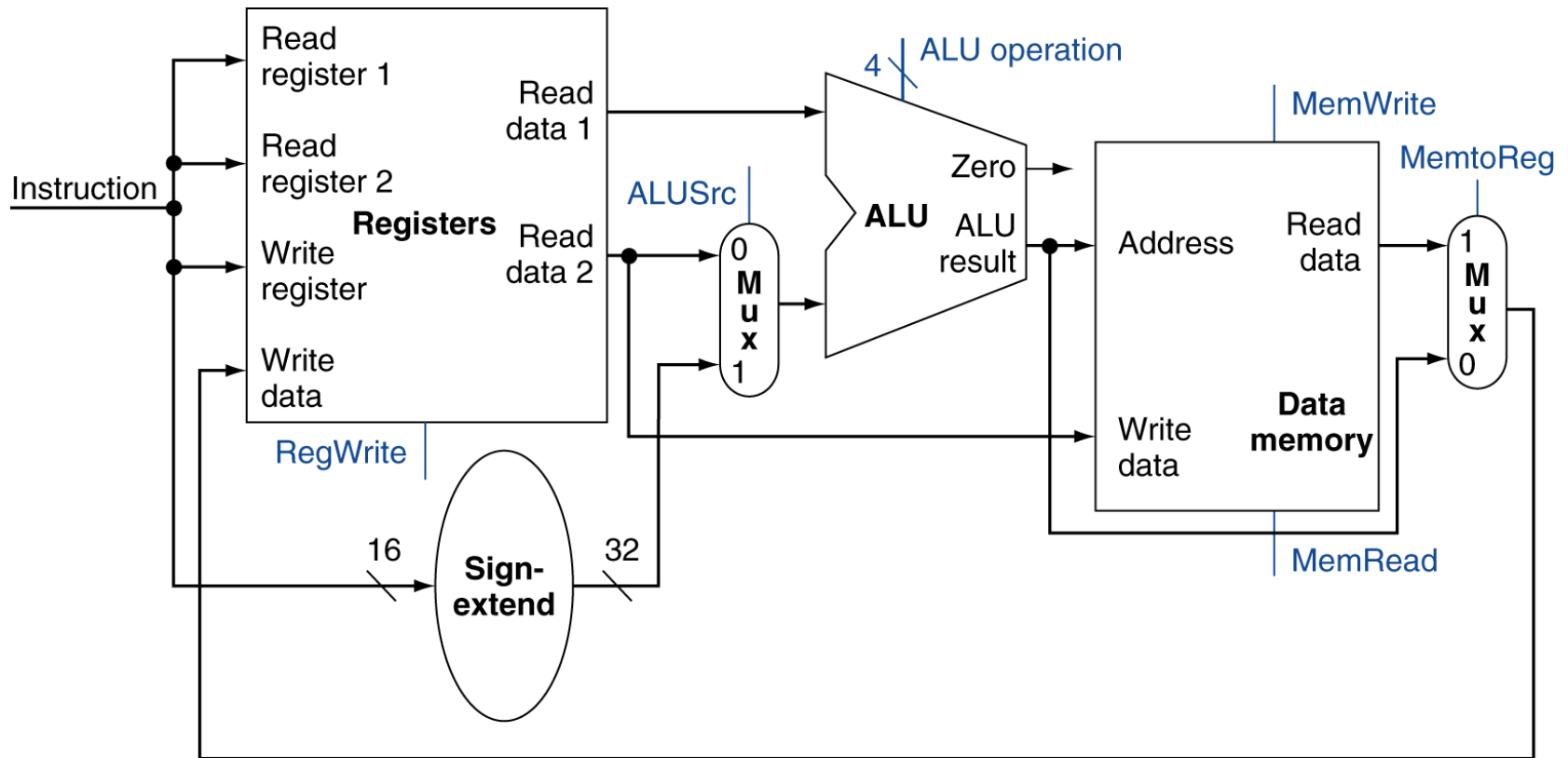
# Branch Instructions



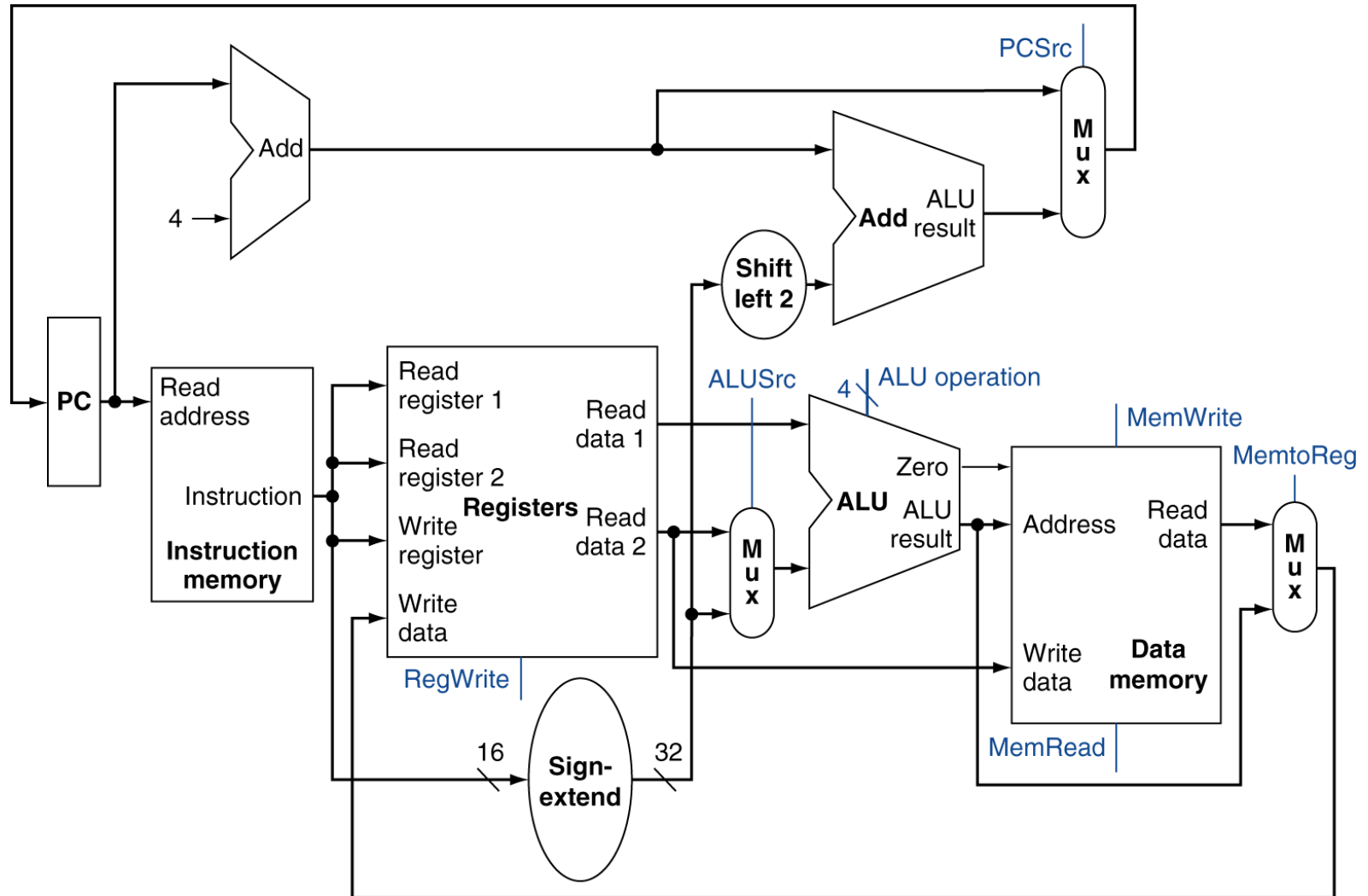
# Composing the Elements

- First-cut data path does an instruction in one clock cycle
  - No datapath resource can be used more than once per instruction
    - Any element needed more than once must be duplicated
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

# R-Type/Load/Store Datapath



# Full Datapath



# ALU Control

- ALU used for
  - **Load/Store**:  $F = ?$
  - **Branch**:  $F = ?$
  - **R-type**:  $F$  depends on funct field

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR




# ALU Control

- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

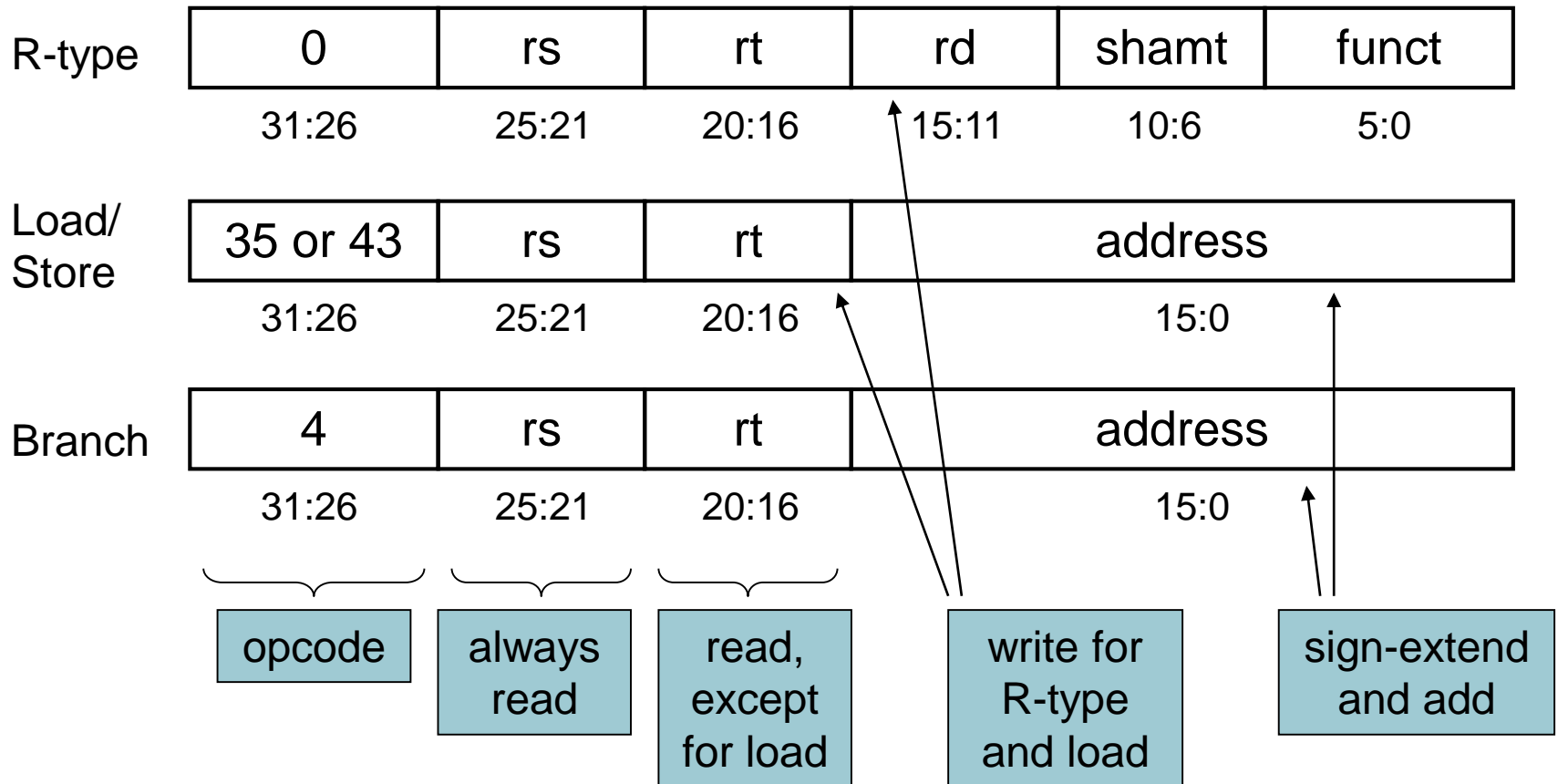
opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

Generated  
output

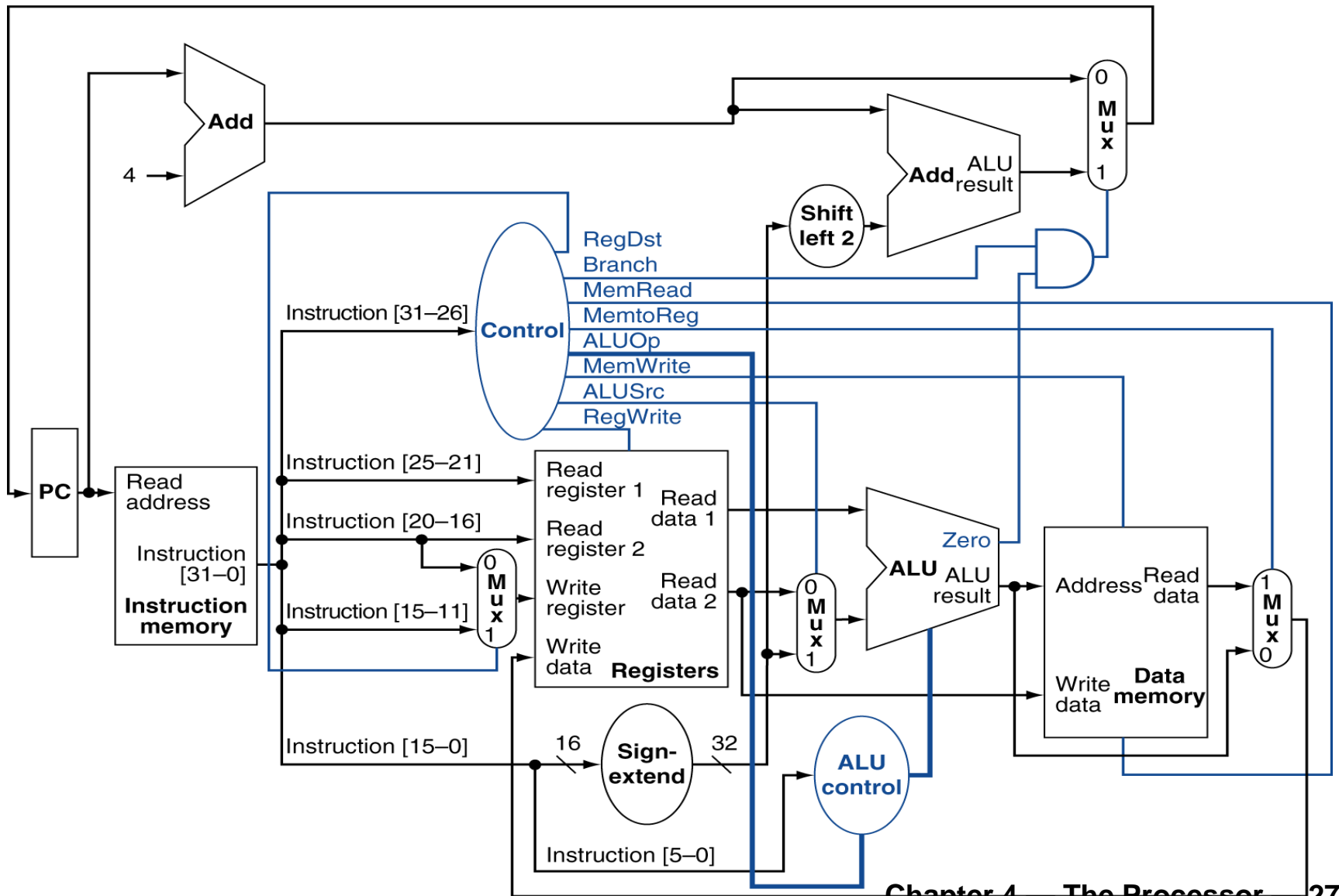


# The Main Control Unit

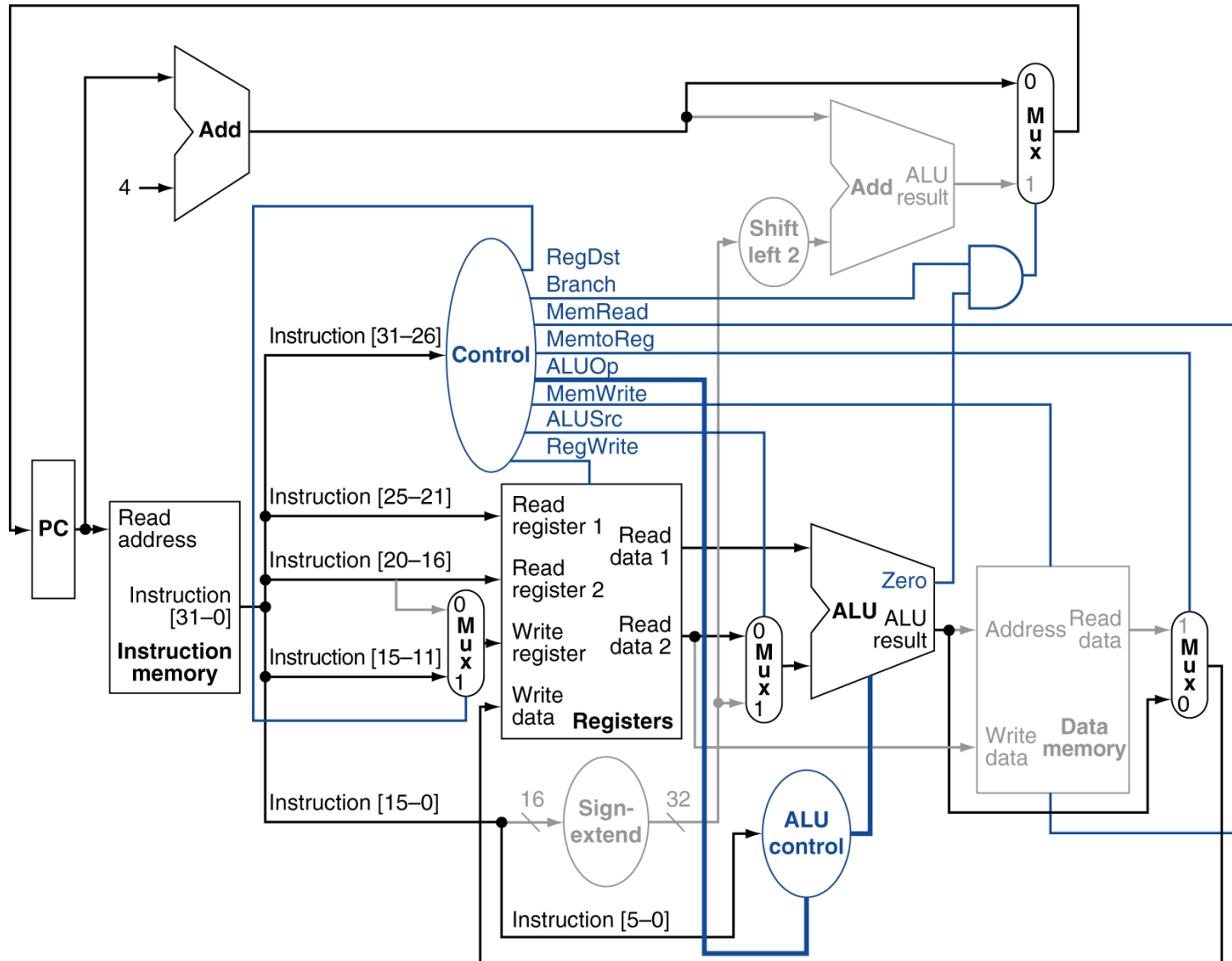
- Control signals derived from instruction



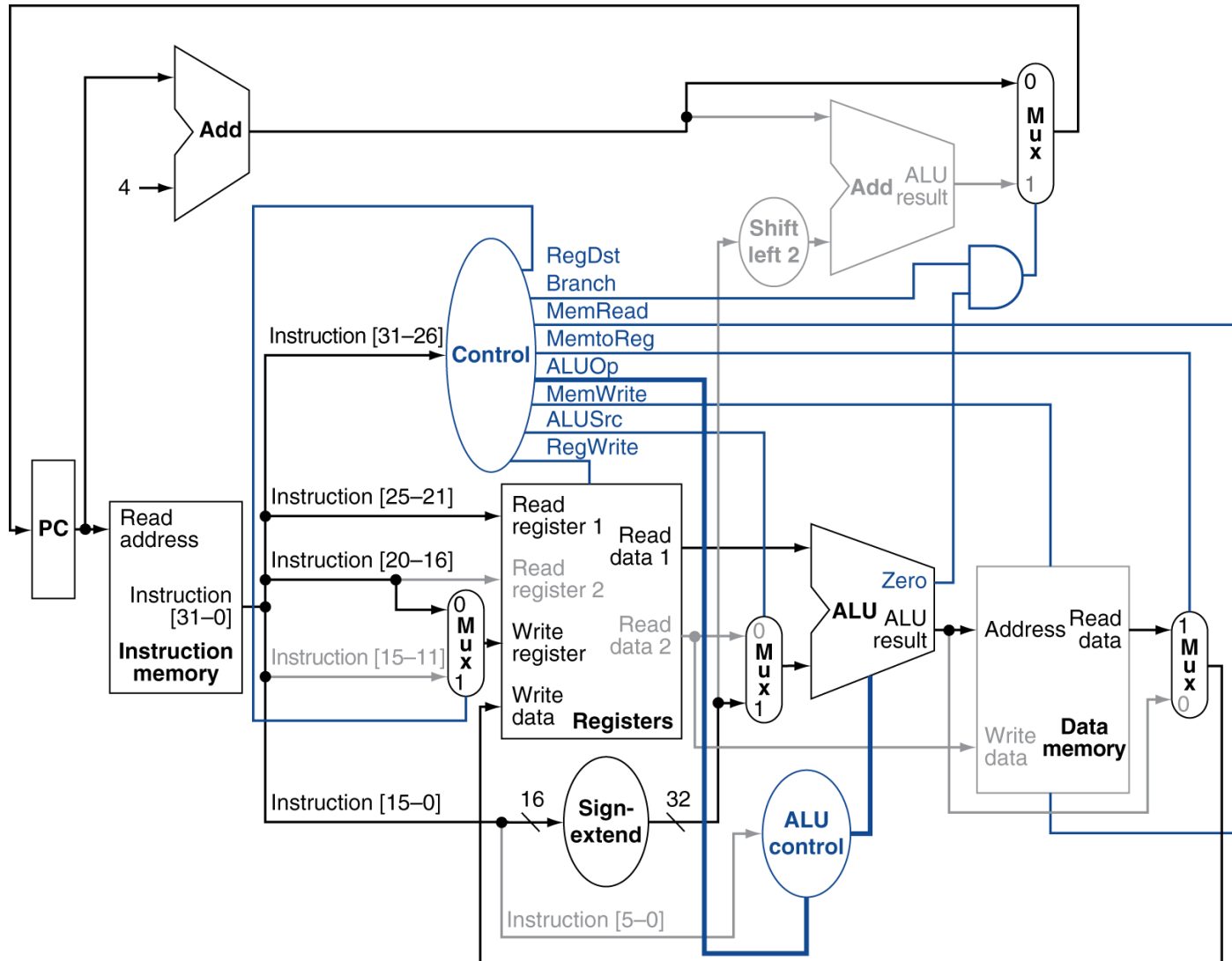
# Datapath With Control



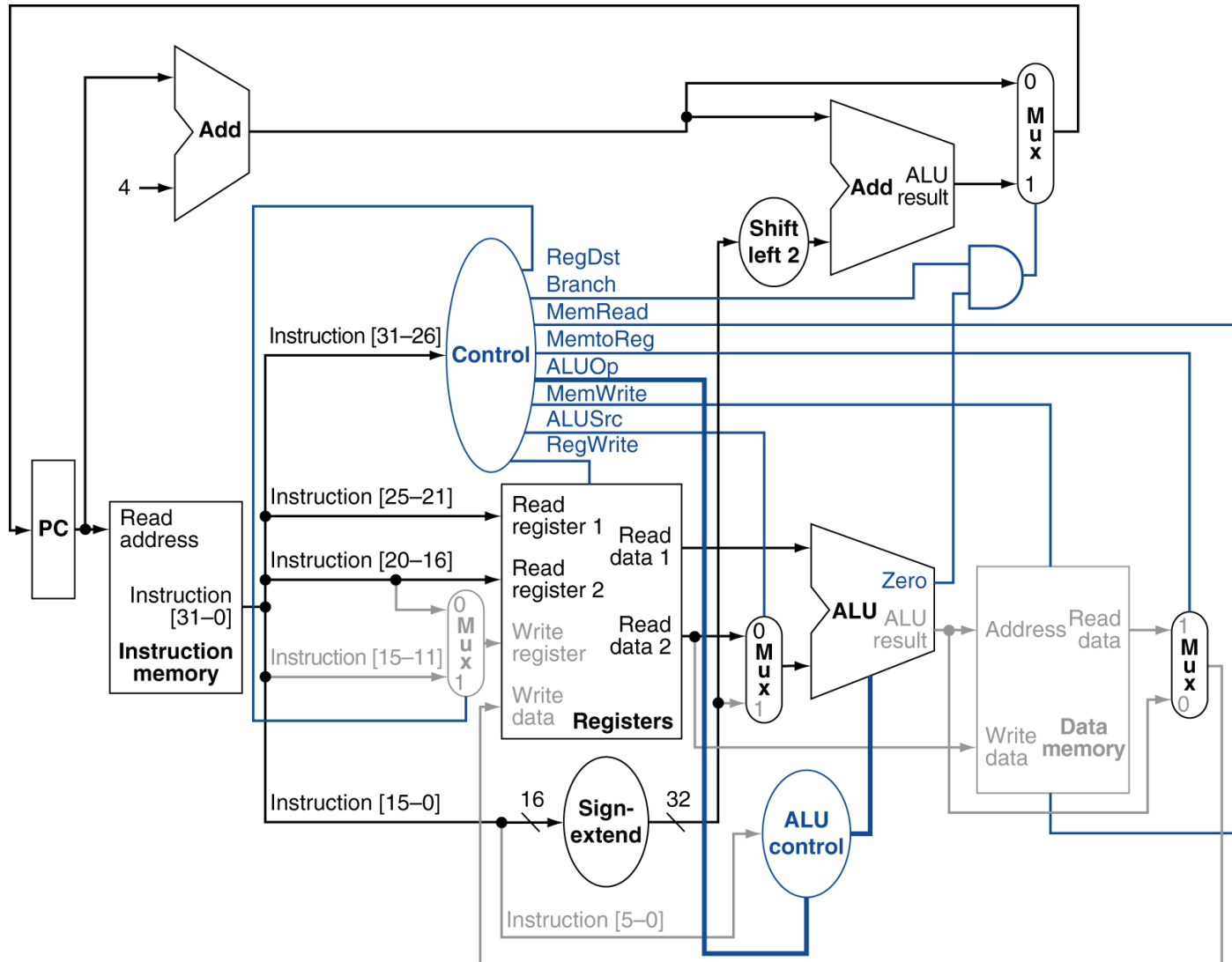
# R-Type Instruction



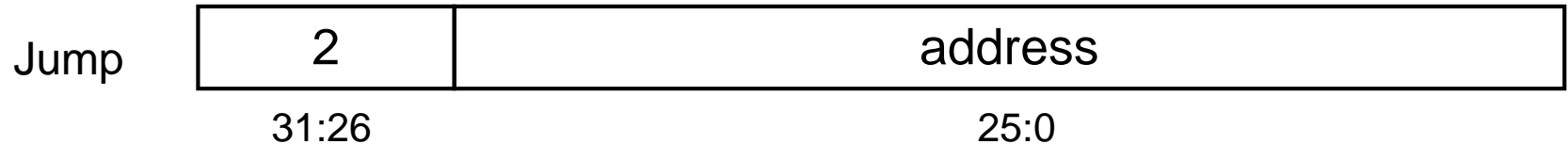
# Load Instruction



# Branch-on-Equal Instruction

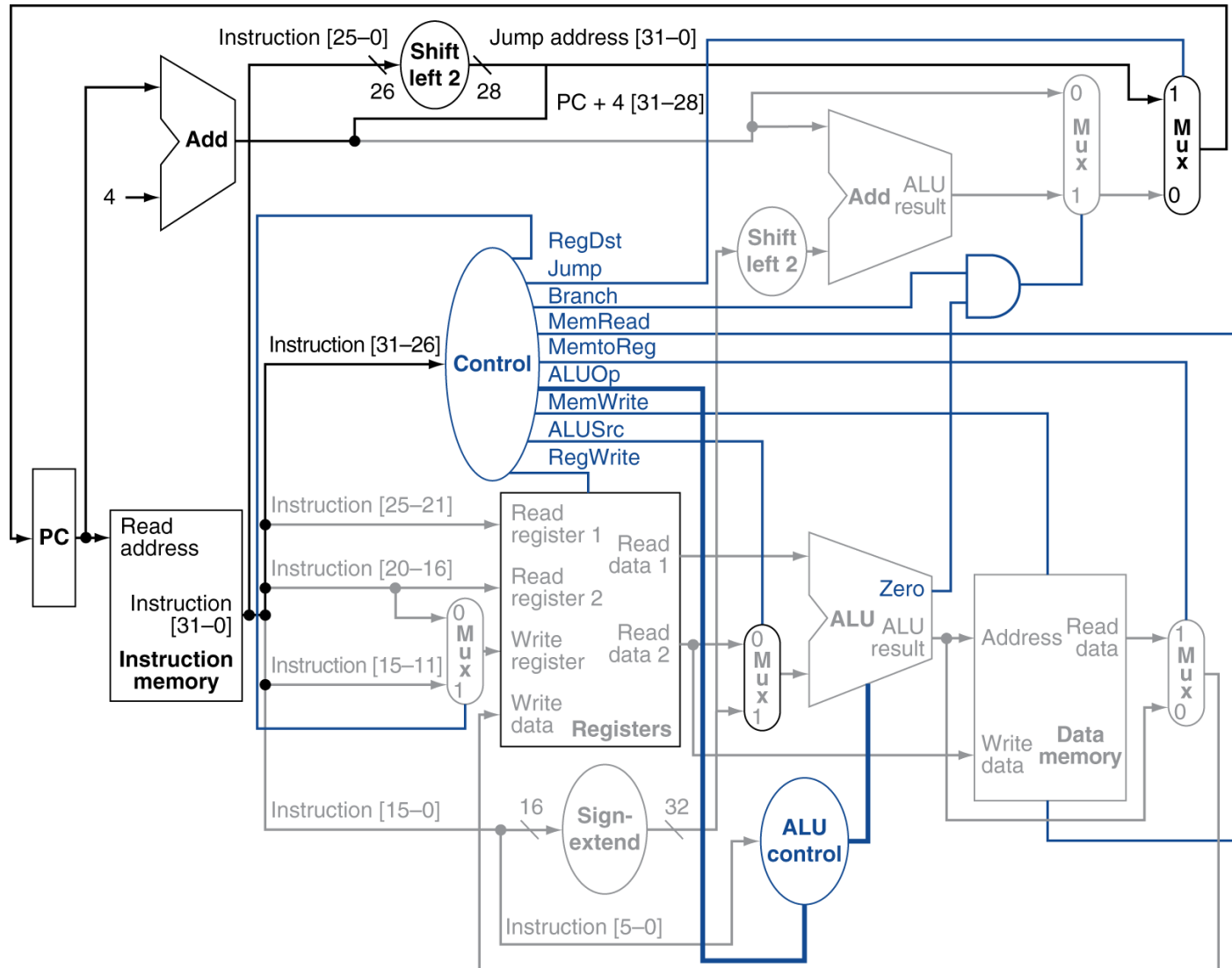


# Implementing Jumps



- Jump uses word address
- Update PC with concatenation of
  - Top 4 bits of old PC
  - 26-bit jump address
  - 00
- Need an extra control signal decoded from opcode

# Datapath With Jumps Added





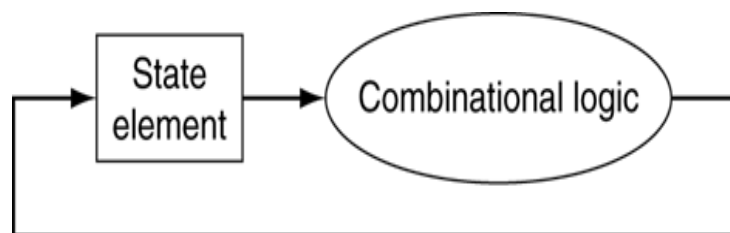
# 简化的MIPS指令 处理机设计

## 单周期处理机的缺陷——性能差

**单周期：**处理机执行每条指令都需要一个时钟周期。而确定时钟周期的长度时，要以**最复杂的指令执行时需要多长时间**为基准

• 对于**装入（load）指令**，周期时间必须足够长，指令的执行过程包括几个步骤：

- 1、从指令存储器取指令
- 2、从寄存器堆读出数据
- 3、用ALU计算地址
- 4、从数据存储器取出数据
- 5、最后把数据写入寄存器堆



## 处理机执行指令步骤:



## RR型/RI型

- 1、从指令存储器取指令
- 2、从寄存器堆读出数据
- 3、ALU执行算术逻辑操作
- ~~4、从数据存储器取出数据~~
- 5、最后把数据写入寄存器堆

## Store指令

- 1、从指令存储器取指令
- 2、从寄存器堆读出数据
- 3、ALU计算地址
- 4、把数据写入数据存储器
- ~~5、最后把数据写入寄存器堆~~

根据统计各类指令的使用频度大约为：

ALU指令占 50%

load指令占 20%

Store指令占 10%

Branch指令占 20%

# 多周期处理机实现概述

## 单周期处理器的问题根源：

- 所有指令必须以最慢的指令使用的时间为准

## 解决方案：

- 提高时钟频率，即缩短时钟周期
- 使用不同数量的时钟周期完成不同类型的指令
- 将指令处理分为若干个周期
- 每个周期执行一步（而不是整个指令）
  - 周期时间： 执行最长步所需的时间
  - 使所有的步骤尽量具有相同的长度

## 多周期处理器的优点：

- 周期时间非常短
- 不同的指令可以使用不同的周期数来完成
  - 装入指令 LW 需要5个周期
  - 跳转指令仅仅需要3个周期
- 允许每条指令多次使用同一个功能部件

指令将完成的操作:

**ALU 指令: RR型和RI型**

**RR型**  $rd \leftarrow (rs1) \text{ op } (rs2)$  或  $R(rd) \leftarrow R(rs1) \text{ op } R(rs2)$

**RI型**  $rd \leftarrow (rs1) \text{ op } \text{imme}$  或  $R(rd) \leftarrow R(rs1) \text{ op } \text{imme}$

**存储器访问指令: RI型**

**load:**  $rd \leftarrow ((rs1) + \text{imme})$  或  $R(rd) \leftarrow \text{mem}[(rs1) + \text{imme}]$

**store**  $rd \rightarrow ((rs1) + \text{imme})$  或  $R(rd) \rightarrow \text{mem}[(rs1) + \text{imme}]$

**转移/跳转: BR型**

**条件转移** **beq:** if  $z=1$  then  $pc=pc+disp$ , else  $pc=pc+1$

**bne:** if  $z=0$  then  $pc=pc+disp$ , else  $pc=pc+1$

**无条件转移: Branch :**  $pc=pc + disp$

# 假定多周期机器涉及的具体指令如下

## ALU指令

- add rd, rs1, rs2      addi rd, rs1, imme
- sub rd, rs1, rs2      subi rd, rs1, imme
- and rd, rs1, rs2      andi rd, rs1, imme
- or rd, rs1, rs2      ori rd, rs1, imme

## 存储器访问

- load rd, rs1, imme
- store rd, rs1, imme

条件转移:    beq    disp    bne    disp

无条件跳转: Branch    disp



所有的指令都是32位长      具有如下三种格式:

RR型

31 26	25 21	20 16	15 5	4 0
opcode	rd	rs1		rs2

RI型

31 26	25 21	20 16	15 5	4 0
opcode	rd	rs1	Immediate	

BR型

31 26	25 21	20 16	15 5	4 0
opcode	Displacement			

**opcode:** 指令的操作码

**rs1, rs2, rd:** 源1和源2以及目的寄存器描述符

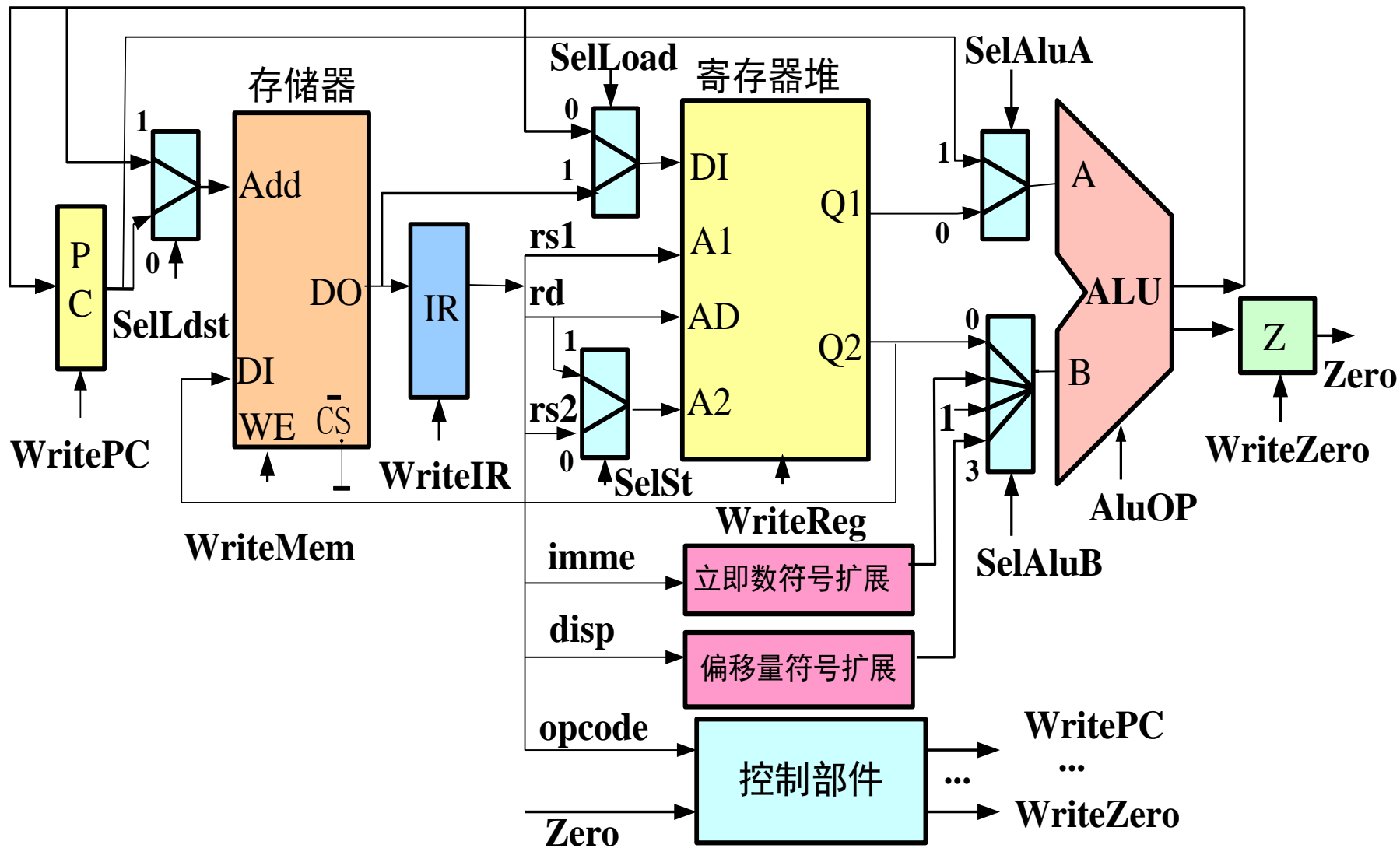
**displacement:** 偏移量

**immediate:** 立即数数值

## 指令系统和指令格式

31    26	25   21	20   16	15   5	4   0	指令助记符	意   义
00 0000	rd	rs1		rs2	and rd, rs1, rs2	rd ←(rs1) and (rs2)
00 0001	rd	rs1	imme		andi rd, rs1, imme	rd←(rs1) 和 imme
00 0010	rd	rs1		rs2	or   rd, rs1, rs2	rd ←(rs1) or   (rs2)
00 0011	rd	rs1	imme		ori   rd, rs1, imme	rd ←(rs1) or imme
00 0100	rd	rs1		rs2	add rd, rs1, rs2	rd ←(rs1) add (rs2)
00 0101	rd	rs1	imme		addi rd, rs1, imme	rd ←(rs1) add imme
00 0110	rd	rs1		rs2	sub rd, rs1, rs2	rd ←(rs1) sub (rs2)
00 0111	rd	rs1	imme		subi rd, rs1, imme	rd ←(rs1) sub imme
00 1000	rd	rs1	imme		load rd, rs1, imme	rd ←((rs1) + imme)
00 1001	rd	rs1	imme		store rd, rs1, imme	(rd)→((rs1) + imme)
00 1010	disp				bne disp	If z=0, pc←(pc)+disp
00 1011	disp				bnq disp	If z=1, pc←(pc)+disp
00 1100	disp				branch disp	pc←(pc) + disp

# 多周期处理机的数据路径总体图



## 数据路径有如下改动

- 1、存储器模块只用一个，指令和数据都使用该存储器
- 2、不使用专门的加法器，而是借用ALU

## 由此而产生如下影响：

- 1、取指令操作和访存指令执行时地址来源不同，**增加多路选择器**
- 2、访存指令执行时，正在执行的指令将丢失。**增加指令寄存器IR**
- 3、ALU前**增加二选一和四选一多路选择器**。解决PC+1和转移地址计算

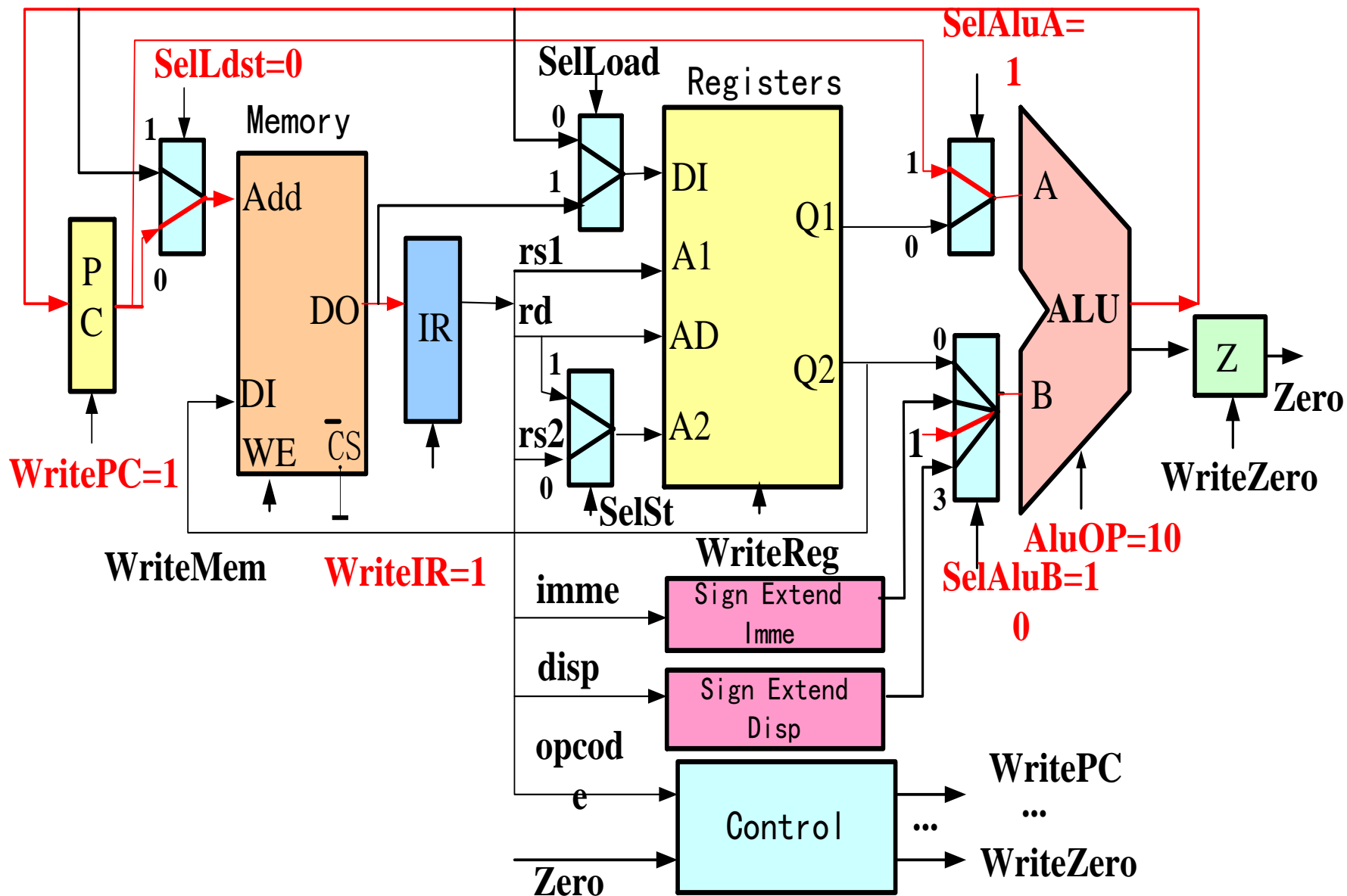
# 多周期处理机执行指令的5个周期

## 1、取指令及PC加1周期

实现以下操作： $IR = \text{Memory}[PC]$   
 $PC = PC + 1$

### 该周期用到的控制信号：

$\text{SelLdst} = 0$  （选择PC的内容做存储器地址）  
 $\text{SelAluA} = 1$  （选择PC的内容做ALU A端的输入）  
 $\text{SelAluB} = 10$  （选择1做ALU B端的输入）  
 $\text{AluOP} = 10$  （ALU 做加运算）  
 $\text{WriteIR} = 1$  （周期结束时写指令寄存器IR）  
 $\text{WritePC} = 1$  （周期结束时写程序计数器PC）



## 2、指令译码，读寄存器及转移周期

实现以下操作：  
A=Register[rs1]  
B=Register[rs2]  
IM=SignExtend[imme]  
If (BranchTaken)  
PC=PC+SignExtend[disp]

该周期用到的控制信号：

SelSt=0 (非store指令)

SelSt=1 (store指令)

如果该指令是转移指令：

SelAluA=1 (选择PC的内容做ALU A端的输入)

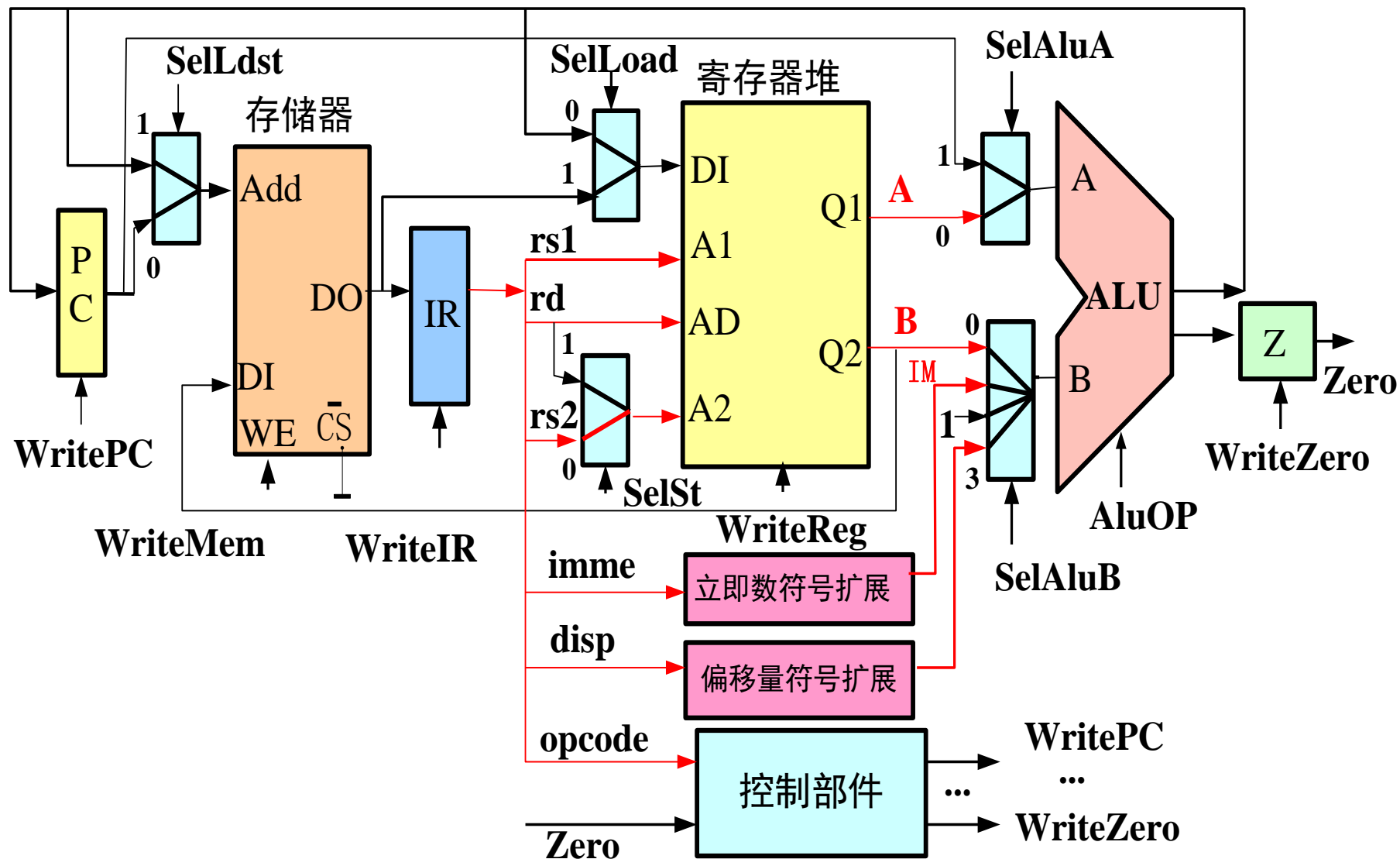
SelAluB=11 (选择扩展的偏移量做ALU B端的输入)

AluOP=10 (ALU 做加运算)

WritePC=Branch+bneZero+beqZero (若转移条件满足时写PC)

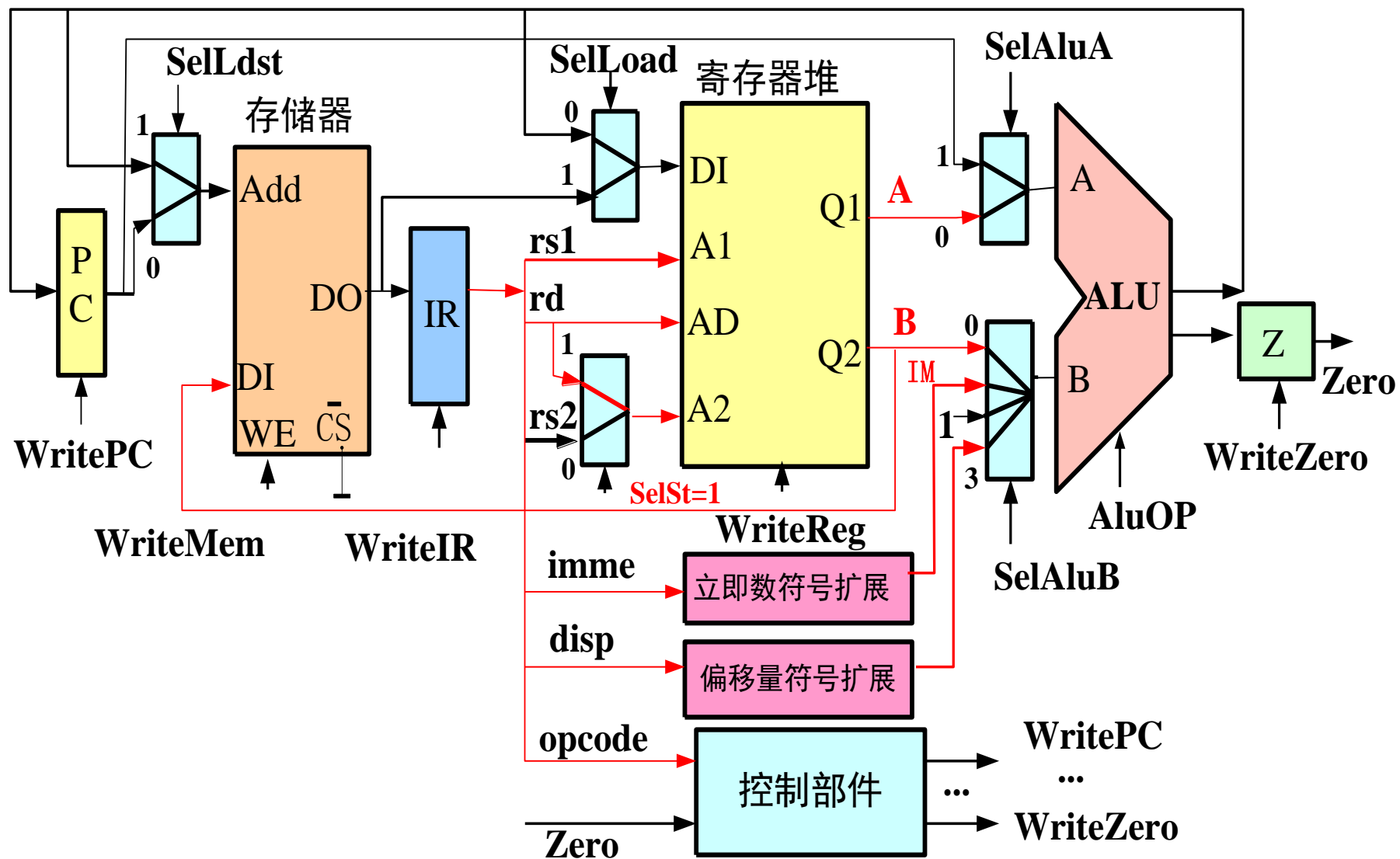
在这个周期结束时，转移指令结束

# 指令译码，读寄存器

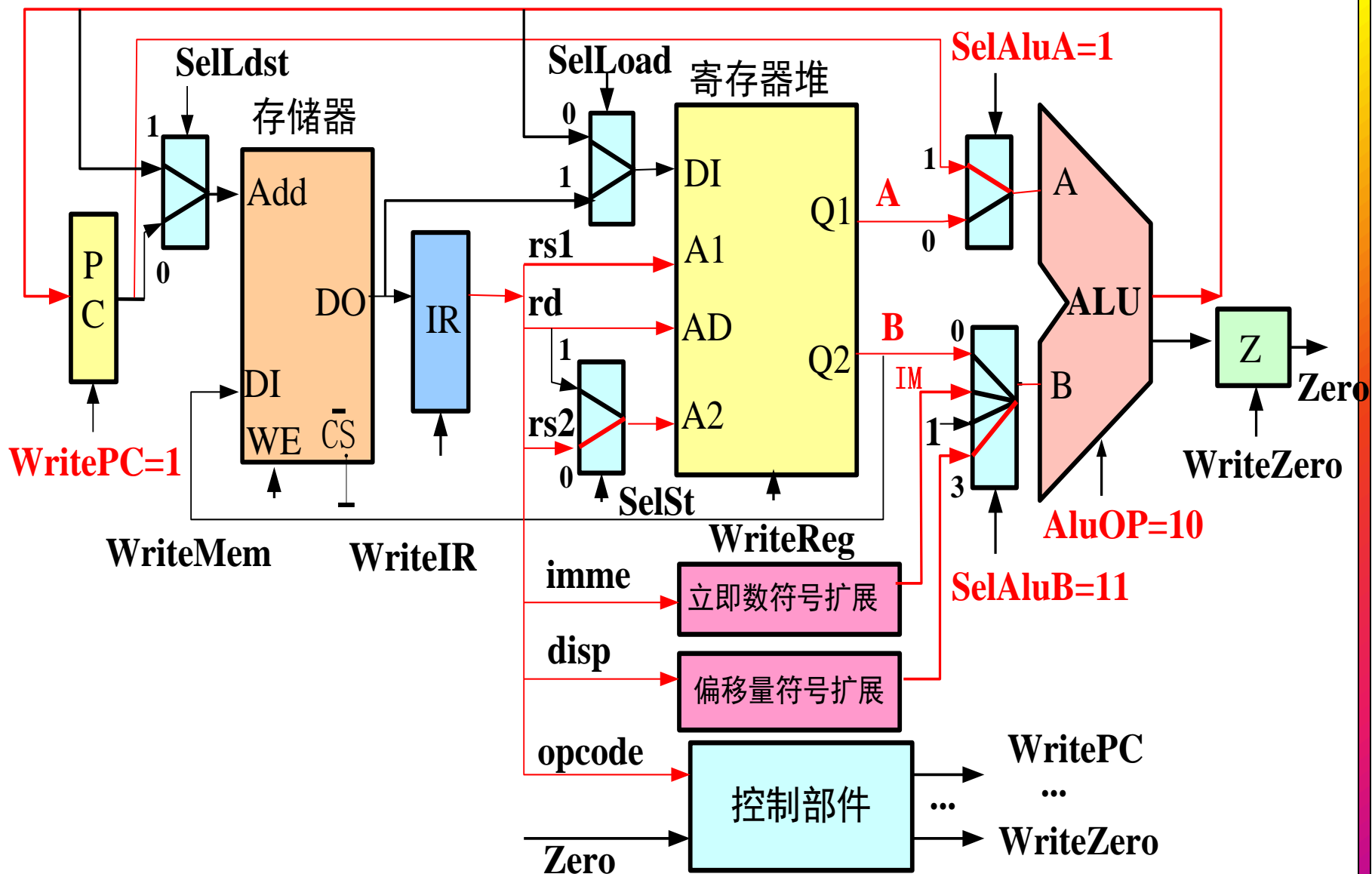




# 指令译码， store指令读寄存器



如果该指令是转移指令，并且条件满足时，把转移地址写入PC



### 3、ALU执行或者存储器地址计算周期

实现以下操作：

RR型指令：  $AluOutput = A \text{ OP } B$

RI型指令：  $AluOutput = A \text{ OP } IM$

存储器访问指令load/stroe:  $AluOutput = A + IM$

该周期用到的控制信号：

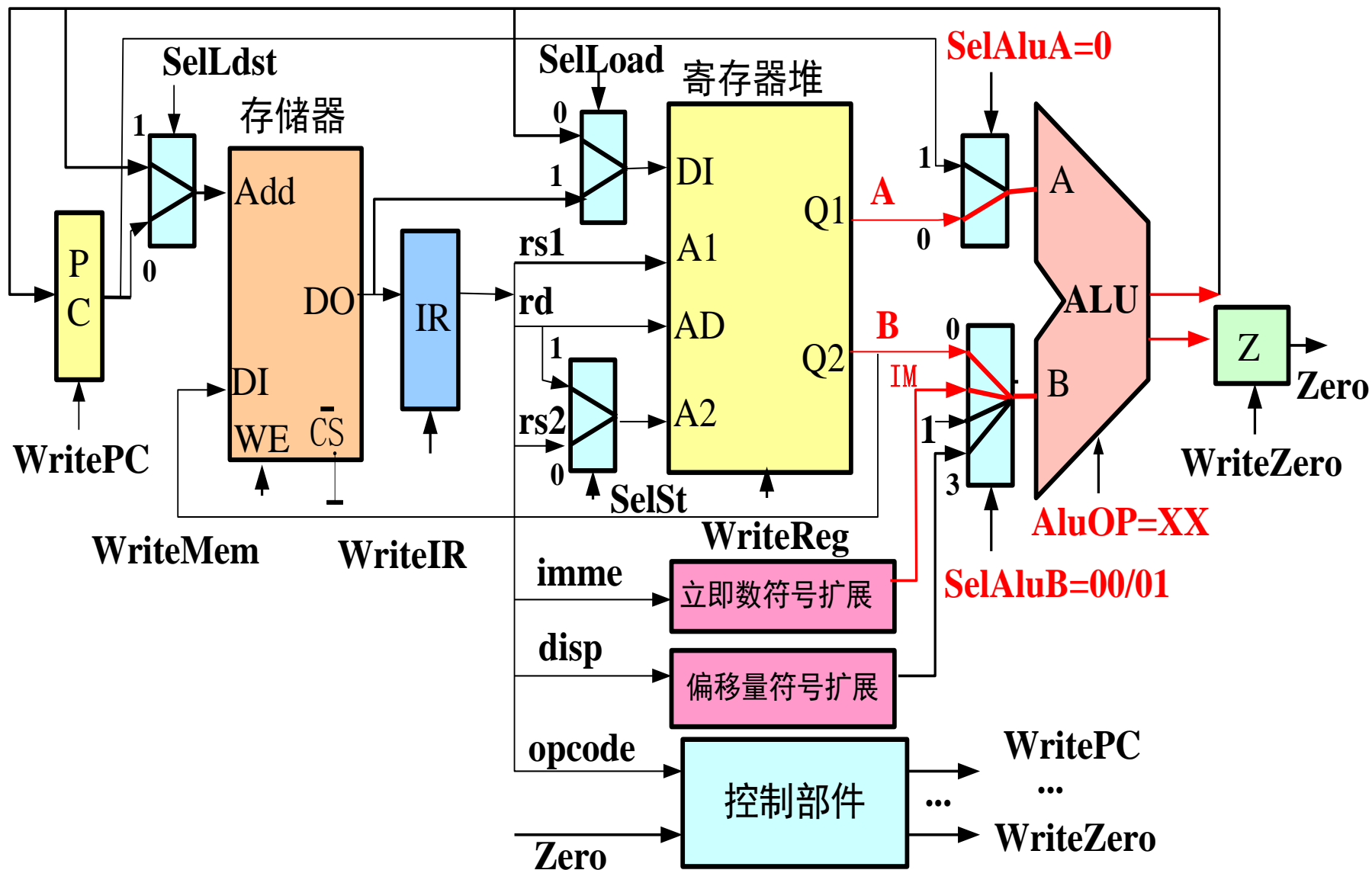
$SelAluA=0$  （选择A做ALU A端的输入）

RR型指令  $SelAluB=00$  （选择B做ALU B端的输入）

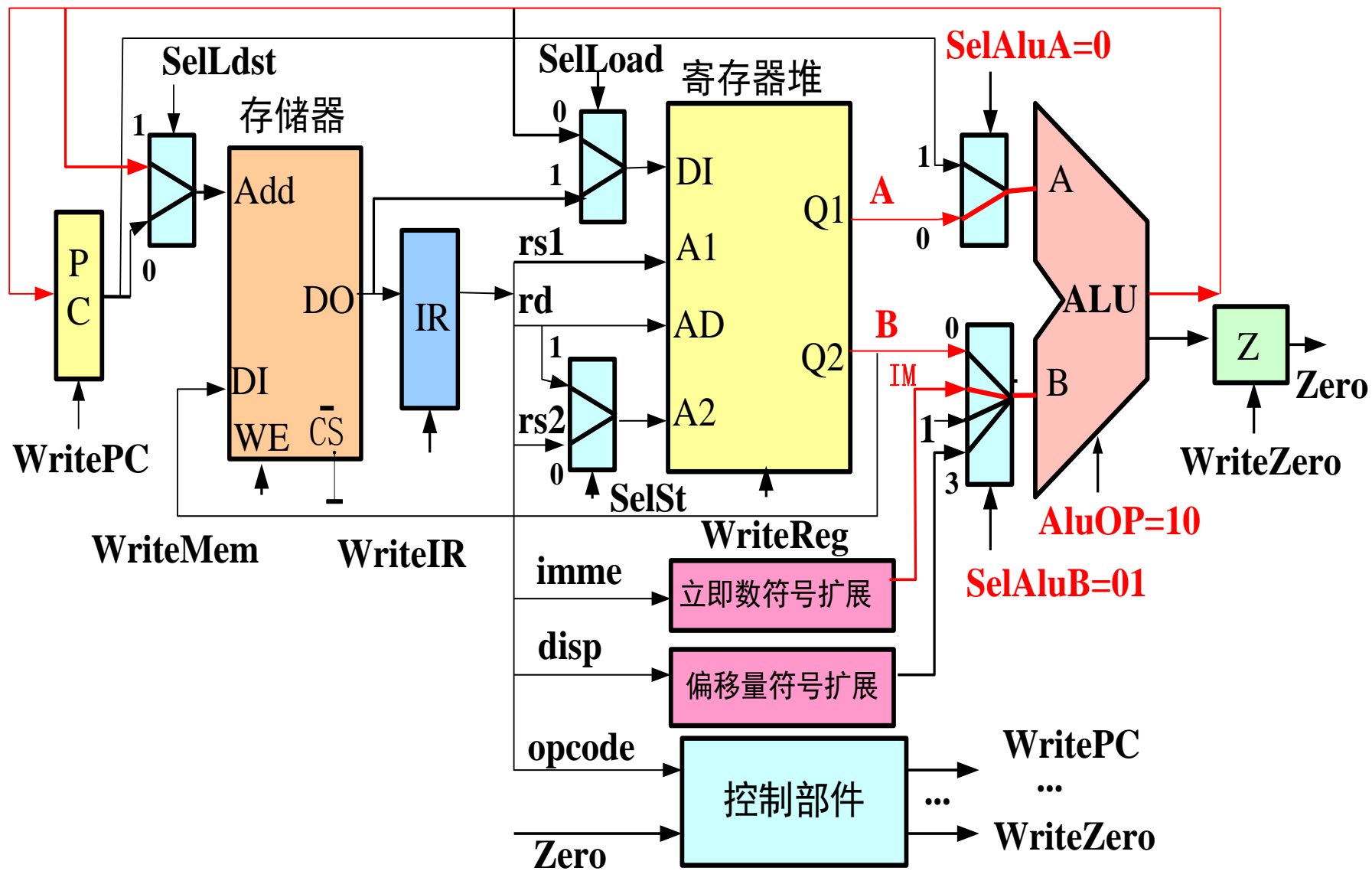
RI型指令  $SelAluB=01$  （选择IM做ALU B端的输入）

$AluOP=XX$  （ALU 完成指令指定的运算）

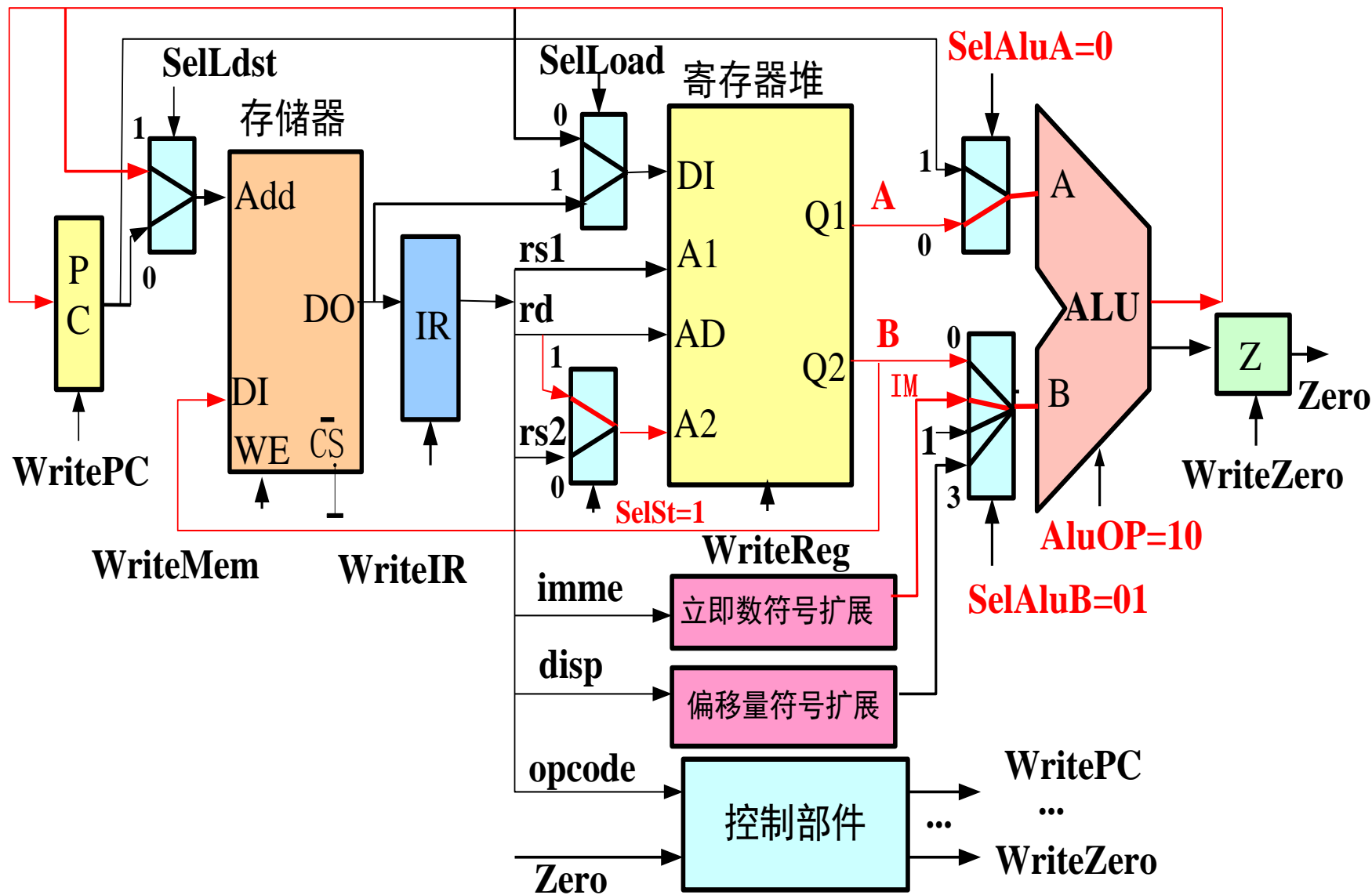
# ALU指令（RR型RI型）执行



## Load 指令计算存储器地址



# Store指令计算存储器地址



#### 4、ALU指令结束周期或者存储器访问周期

##### 1) ALU指令结束周期实现以下操作:

Register[rd]=AluOutput

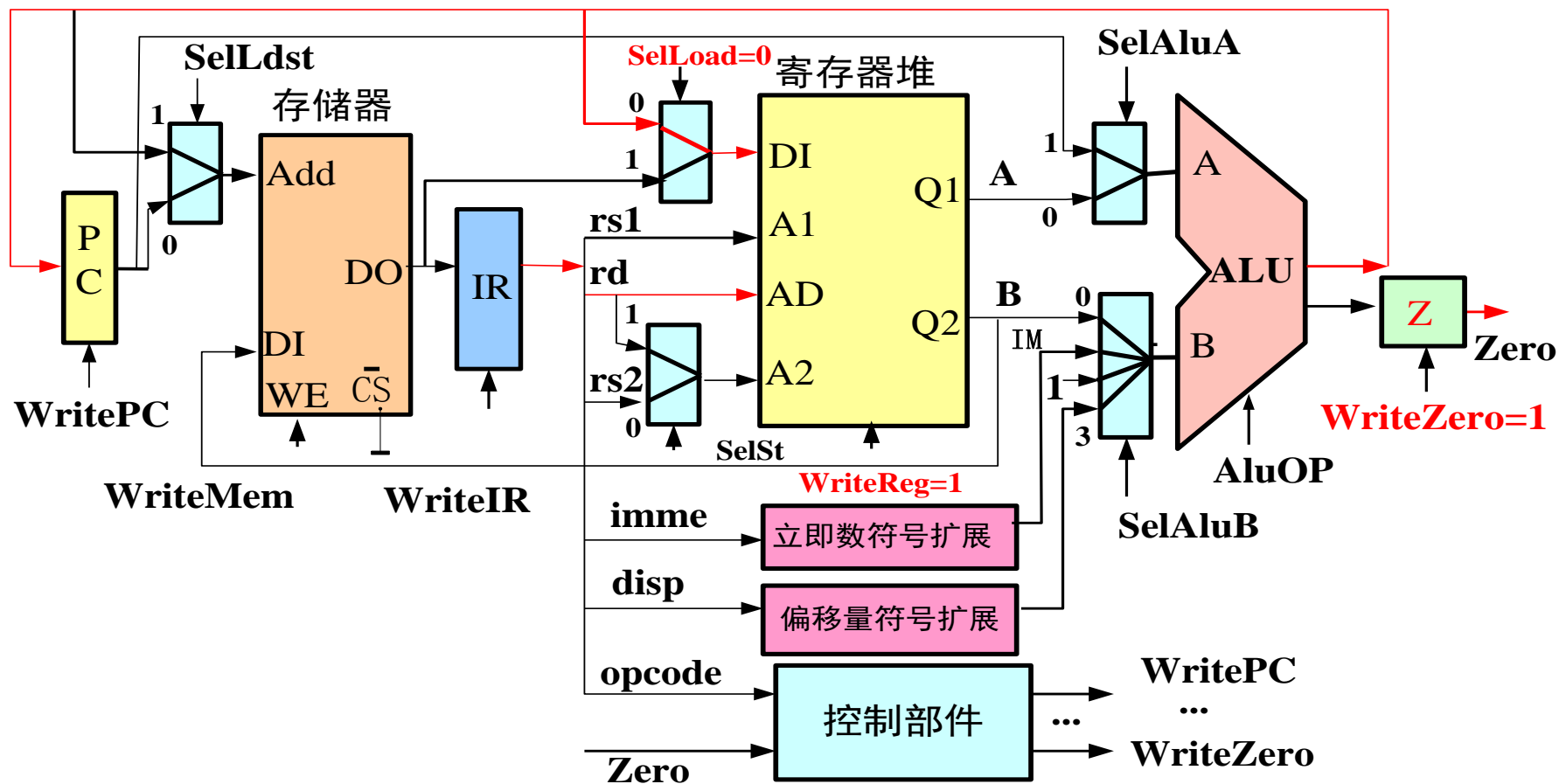
Zero=AluZero

用到的控制信号:

SelLoad=0

WriteZero=1

WriteReg=1

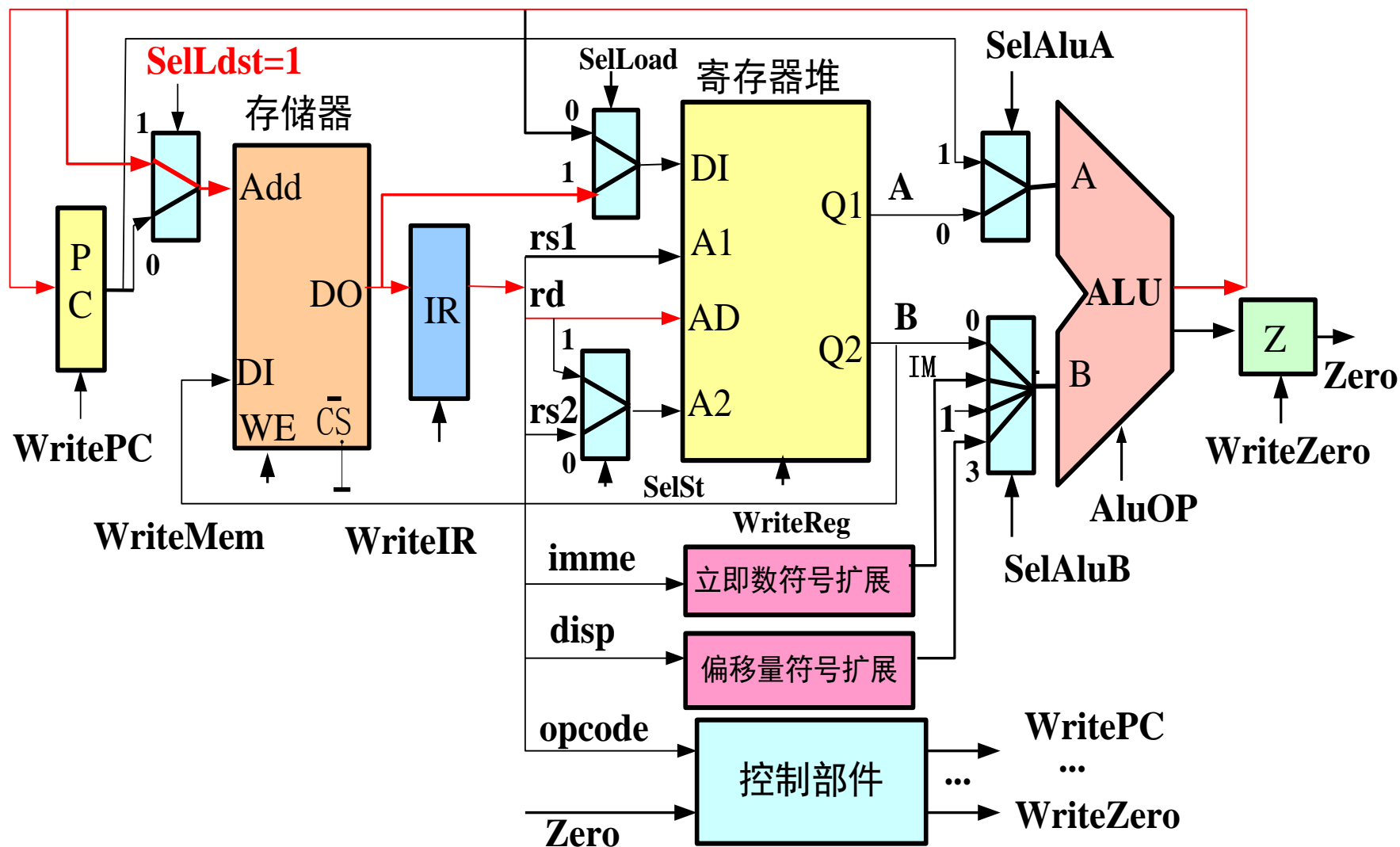


2) 如果是load指令实现以下操作:

$\text{MemOutput} = \text{Memory}[\text{AluOutput}]$

用到的控制信号:

$\text{SelLdst}=1$

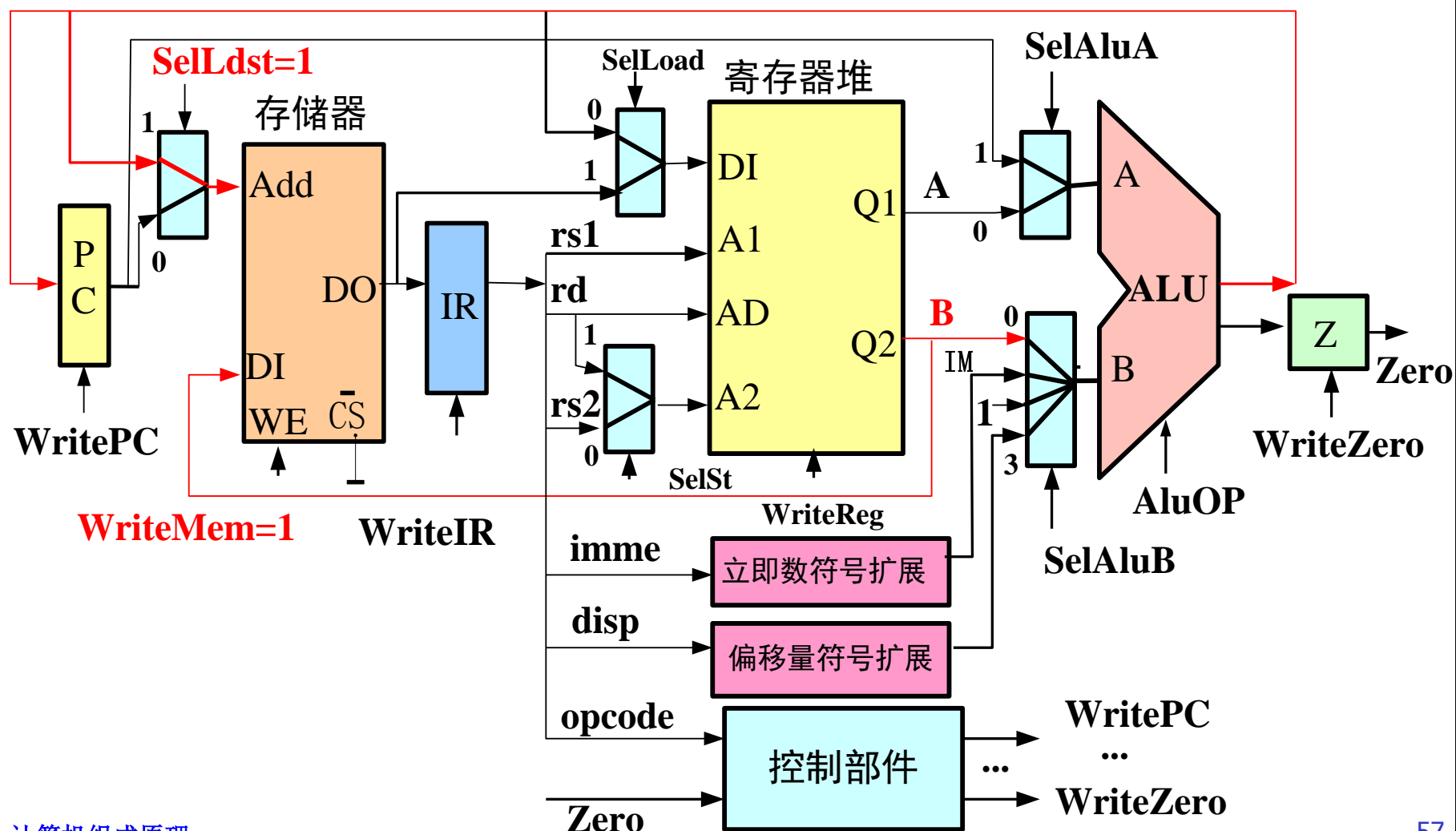




3) 如果是store指令数据写入存储器，指令结束

实现以下操作：  $\text{Memory}[\text{AluOutput}] = B$

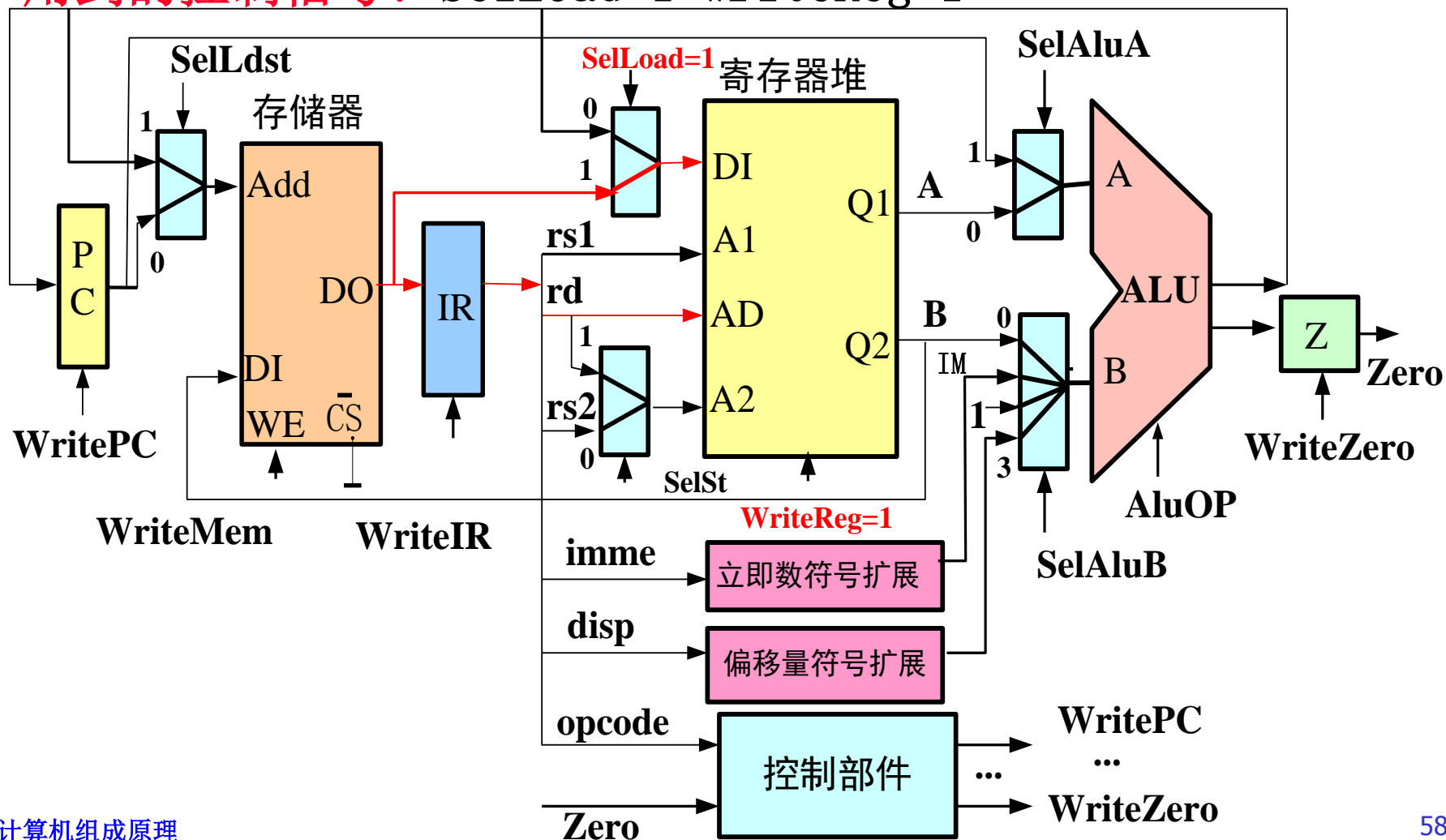
用到的控制信号： $\text{SelLdst}=1$   $\text{WriteMem}=1$



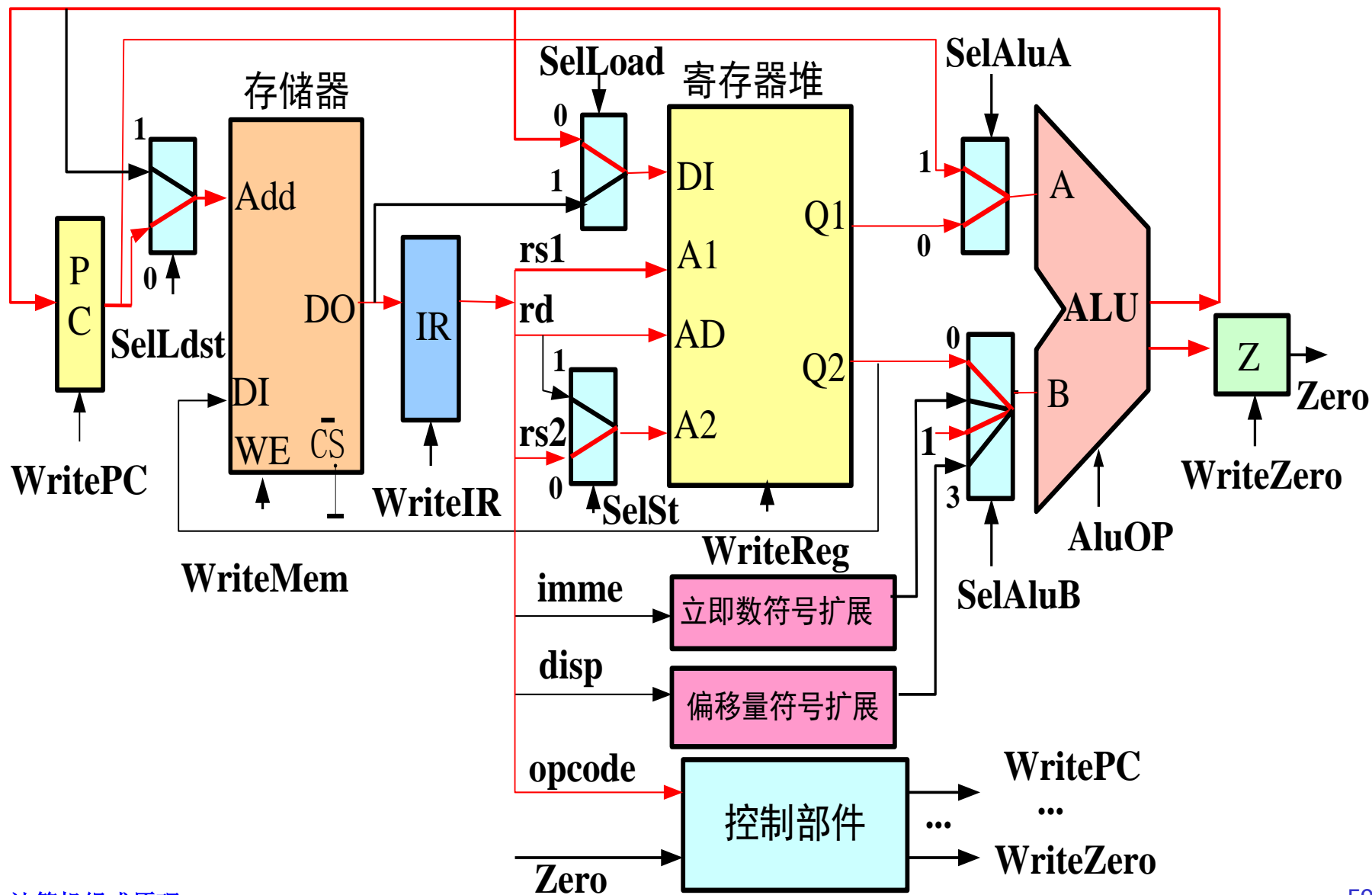
## 5、写回周期—只有load指令才能达到这个周期

实现以下操作: Register[rd]=MemOutput

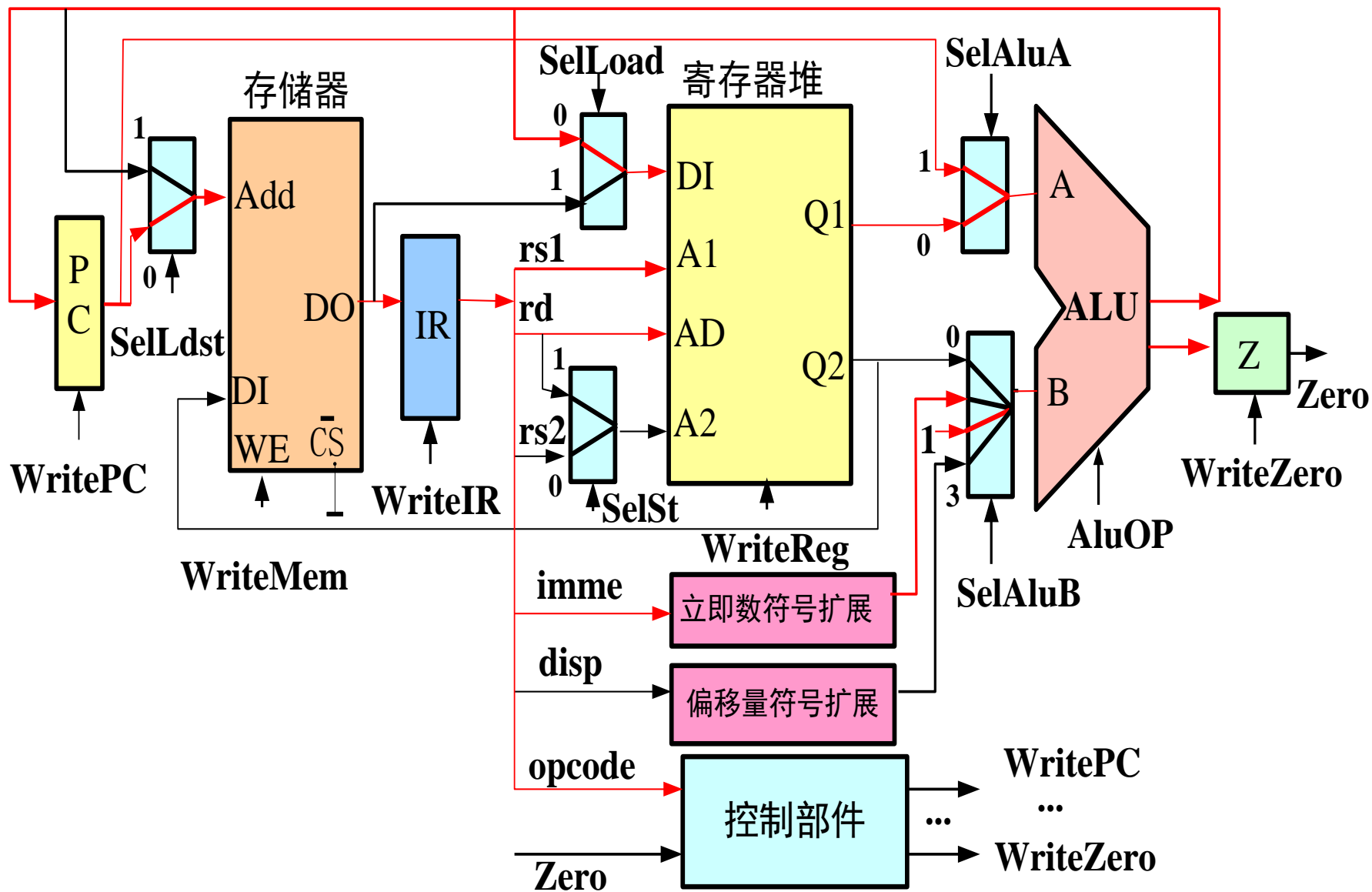
用到的控制信号: SelLoad=1 WriteReg=1



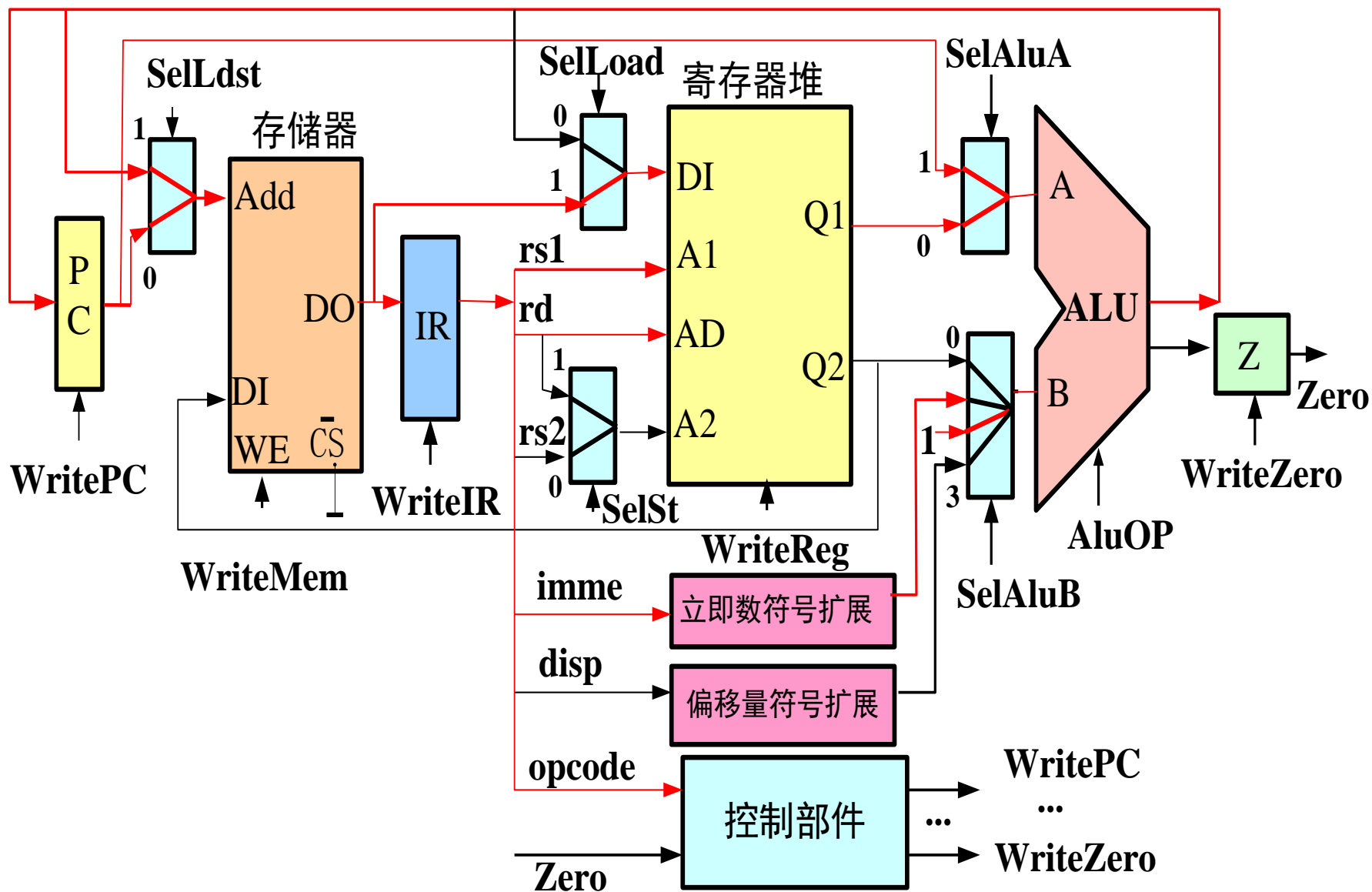
# ALU指令RR型完整数据通路



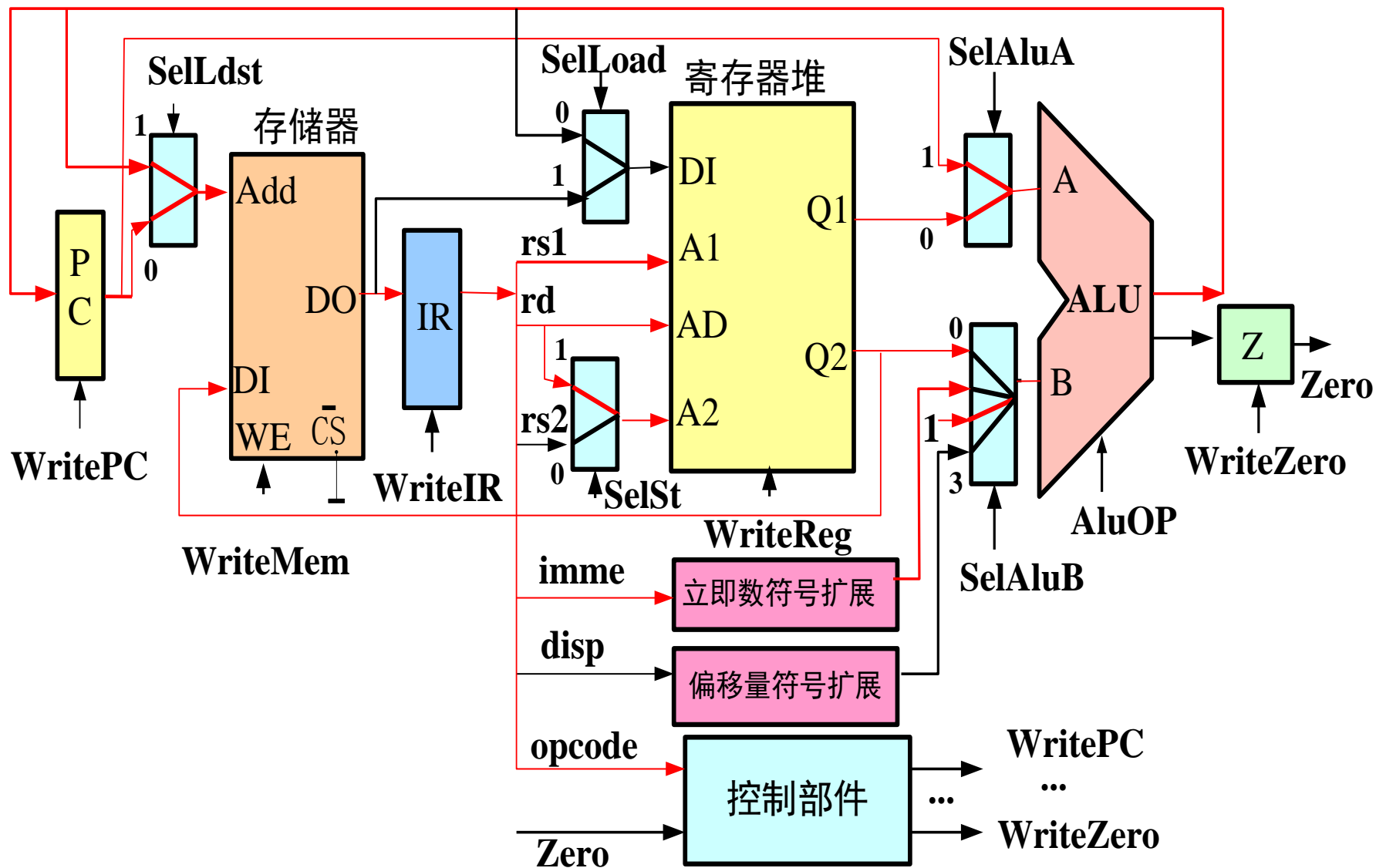
# ALU指令RI型完整数据通路



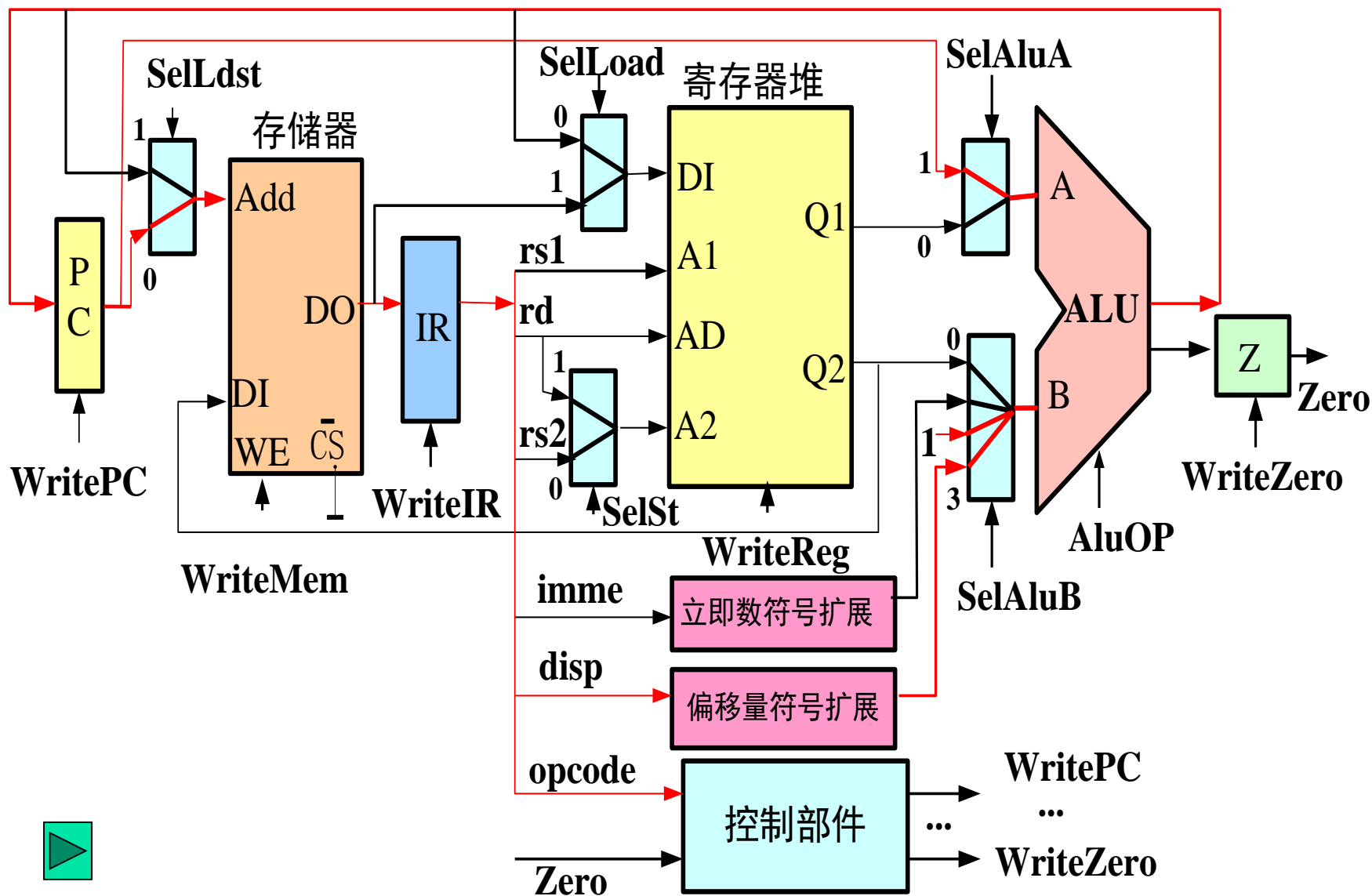
# Load指令完整数据通路



# Store指令完整数据通路



# 转移指令完整数据通路



# 控制器设计

## 有限状态机理论

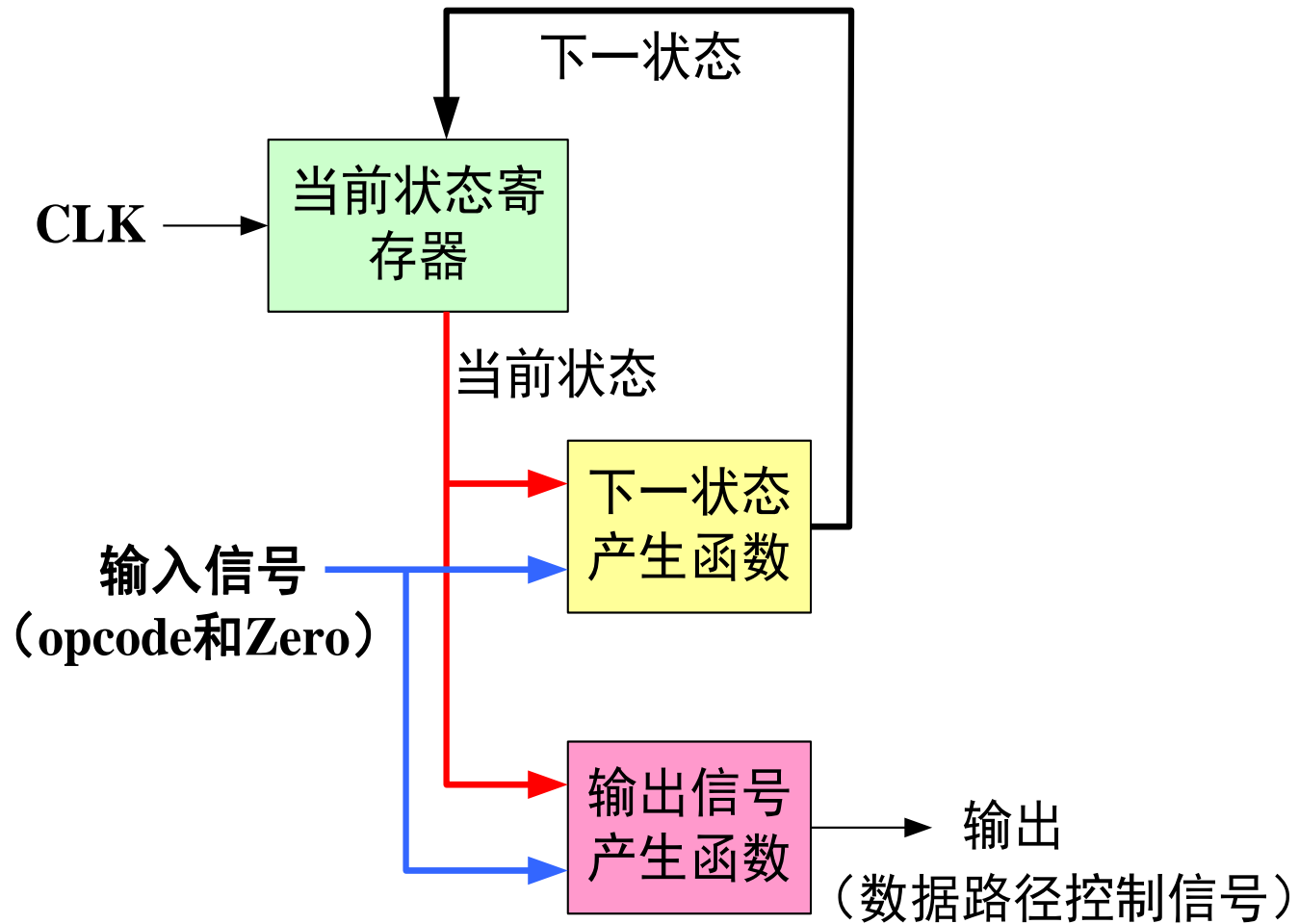
若时序电路有N个状态，则至少需要 $n = \log_2 N$ 个触发器

设计一个有限状态机的步骤一般是：

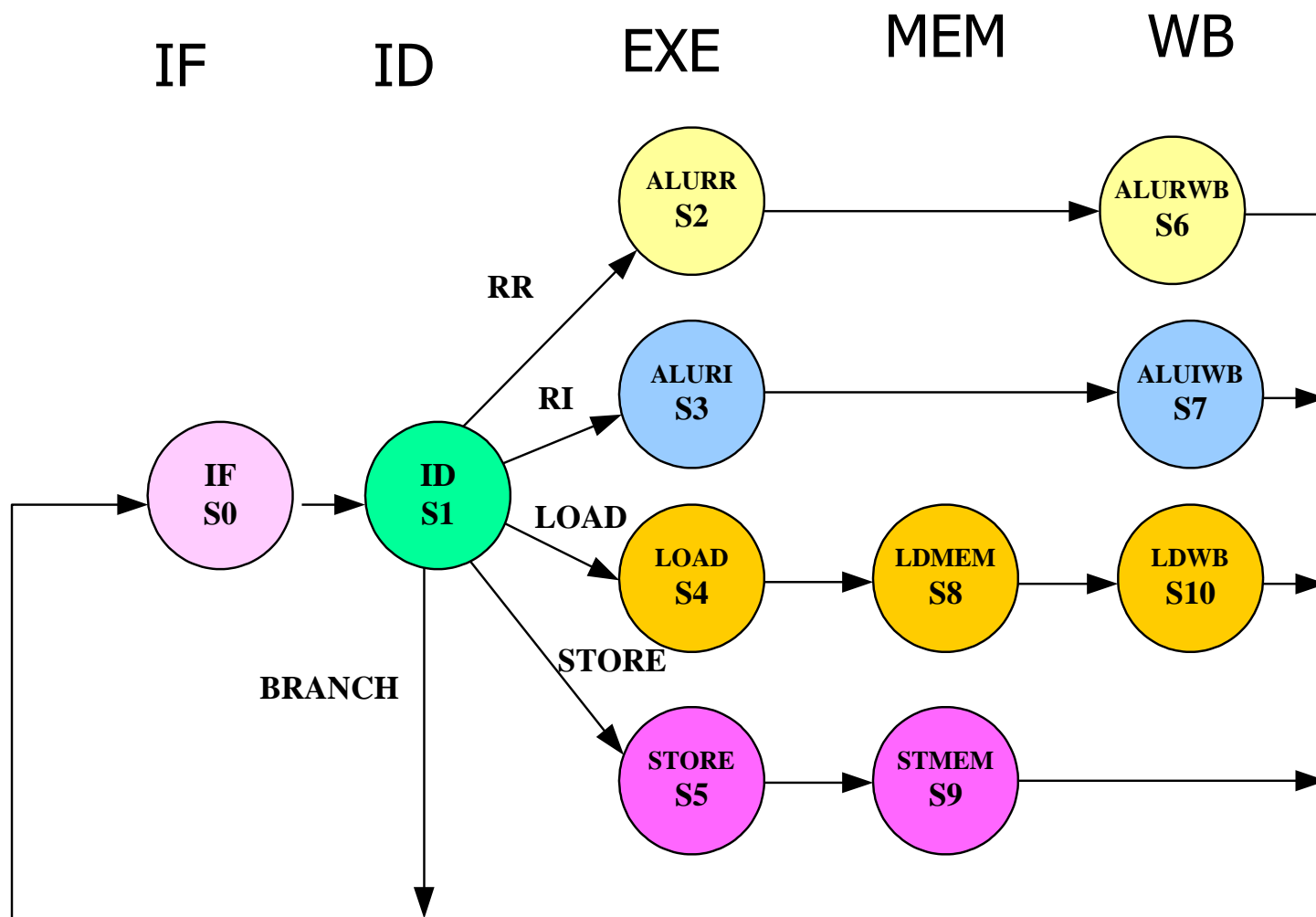
- 1、画出状态转移图。
- 2、写出状态转移表。
- 3、写出下一状态的布尔表达式，并尽可能化简。
- 4、写出输出信号的真值表。
- 5、写出输出信号的布尔表达式，并尽可能化简。
- 6、根据下一状态和输出信号的布尔表达式，画出逻辑图。



# 有限状态机电路模型



# 1、画出状态转移图。



## 2、写出状态转移表。

当前状态		输入	下一状态	
状态	Q3Q2Q1Q0	条件	状态	D3D2D1D0
S0	0 0 0 0	X	S1	0 0 0 1
S1	0 0 0 1	BR	S0	0 0 0 0
	0 0 0 1	RR	S2	0 0 1 0
	0 0 0 1	RI	S3	0 0 1 1
	0 0 0 1	LOAD	S4	0 1 0 0
	0 0 0 1	STROE	S5	0 1 0 1
S2	0 0 1 0	X	S6	0 1 1 0
S3	0 0 1 1	X	S7	0 1 1 1
S4	0 1 0 0	X	S8	1 0 0 0
S5	0 1 0 1	X	S9	1 0 0 1
S6	0 1 1 0	X	S0	0 0 0 0
S7	0 1 1 1	X	S0	0 0 0 0
S8	1 0 0 0	X	S10	1 0 1 0
S9	1 0 0 1	X	S0	0 0 0 0
S10	1 0 1 0	X	S0	0 0 0 0

3、写出下一状态的布尔表达式，并尽可能化简。

$$D0 = S0 + S1 \text{ RI} + S1 \text{ store} + S3 + S5$$

$$D1 = S1 \text{ RI} + S1 \text{ RR} + S2 + S5 + S8$$

$$D2 = S1 \text{ load} + S1 \text{ store} + S2 + S3$$

$$D3 = S4 + S5 + S8$$

## 4、写出输出信号的真值表。

	S0	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10
WritePC	1	BT	0	0	0	0	0	0	0	0	0
SelLdst	0	X	X	X	1	1	X	X	1	1	1
WriteMem	0	0	0	0	0	0	0	0	0	1	0
WriteIR	1	0	0	0	0	0	0	0	0	0	0
SelLoad	X	X	X	X	1	X	0	0	1	X	1
SelSt	X	0	0	X	X	1	0	X	X	1	X
WriteReg	0	0	0	0	0	0	1	1	0	0	1
SelAluA	1	1	0	0	0	0	0	0	0	0	0
SelAluB1	1	1	0	0	0	0	0	0	0	0	0
SelAluB0	0	1	0	1	1	1	0	1	1	1	1
WriteZero	0	0	0	0	0	0	1	1	0	0	0
AluOP1	1	1	OP1	OP1	1	1	OP1	OP1	1	1	1
AluOP0	0	0	OP0	OP0	0	0	OP0	OP0	0	0	0

BT=branch+bne  $\overline{\text{Zreo}}$   
+beq Zreo

表中红色“1”信号与  
前面讨论提前一个周

指令	OP1	OP0
and/andi	0	0
or/ori	0	1
add/addi	1	0
sub/subi	1	1
OP0=or+ori+sub+subi		OP1=add+addi+sub+subi

## 5、写出输出信号的布尔表达式，并尽可能化简。

$$\text{WritePC} = S0 + S1 \text{ BT}$$

$$\text{SelLdst} = S4 + S5 + S8 + S9 + S10$$

$$\text{WriteMem} = S9$$

$$\text{WriteIR} = S0$$

$$\text{SelLoad} = S4 + S8 + S9$$

$$\text{SelSt} = S5 + S9$$

$$\text{WriteReg} = S6 + S7 + S10$$

$$\text{SelAluA} = S0 + S1$$

$$\text{SelAluB 1} = S0 + S1$$

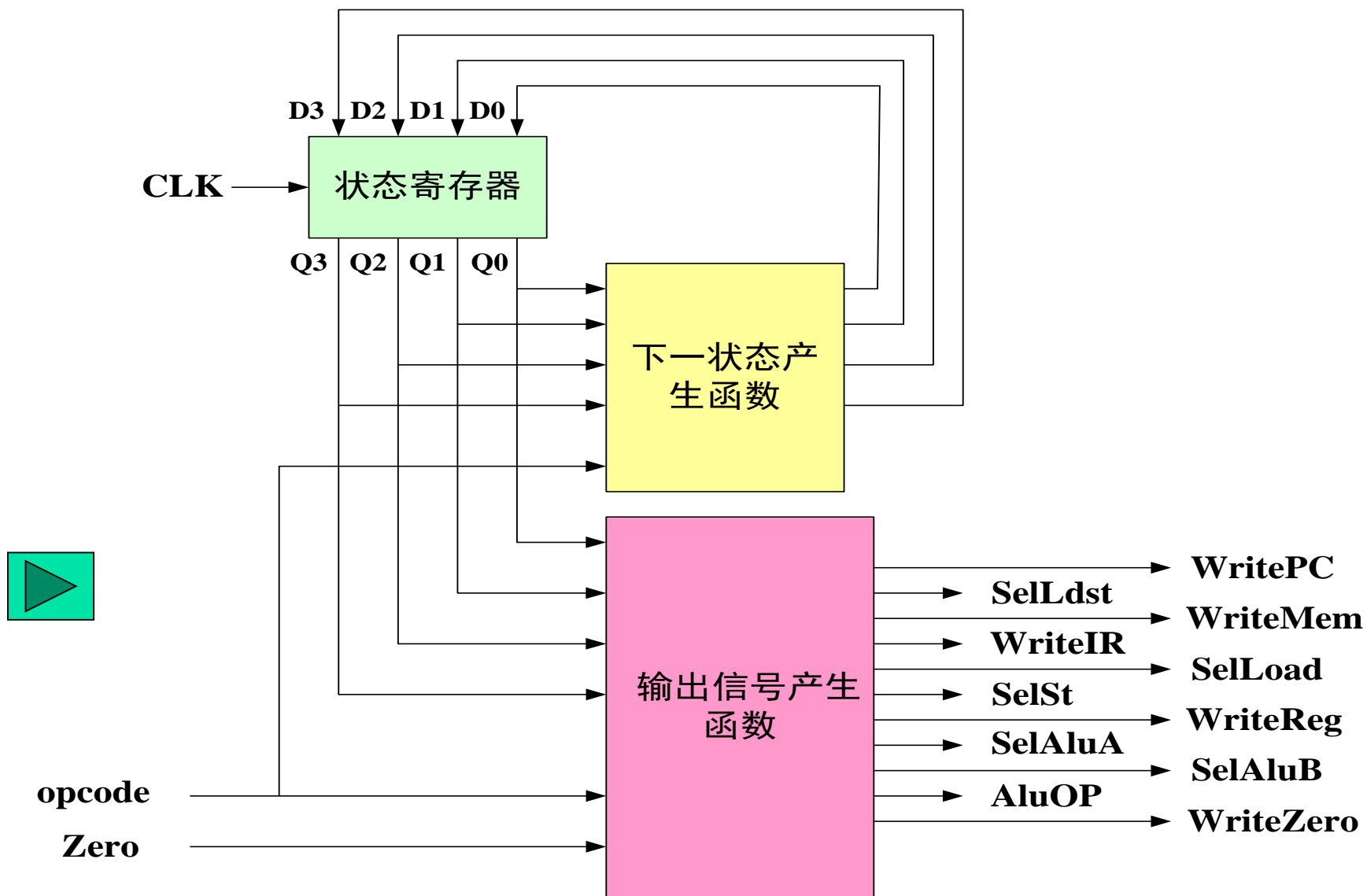
$$\text{SelAluB0} = S1 + S3 + S4 + S5 + S7$$

$$\text{WriteZero} = S6 + S7$$

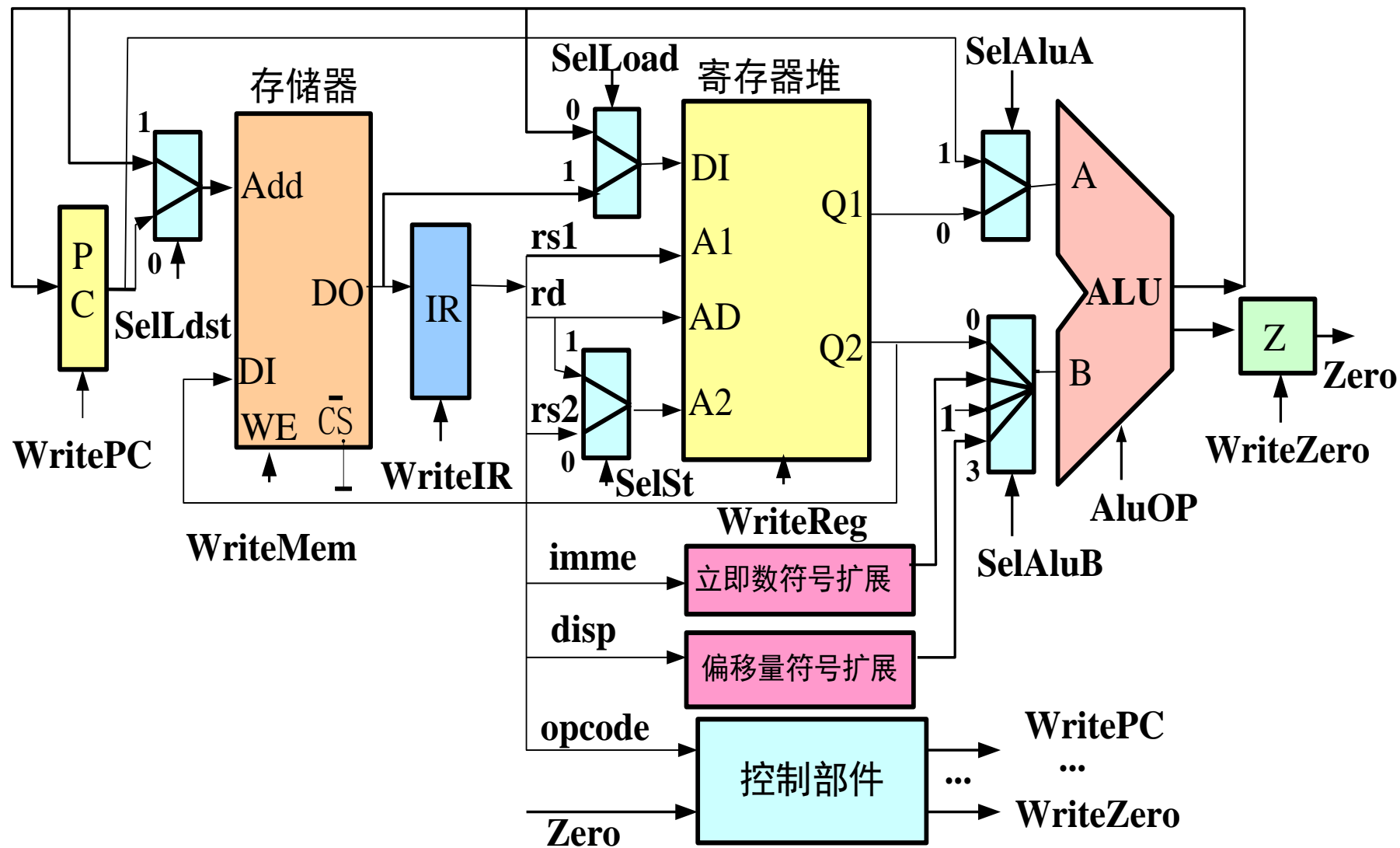
$$\text{AluOP1} = S0 + S1 + S2 \text{ OP1} + S3 \text{ OP1} + S4 + S5 + S6 \text{ OP1} + S7 \text{ OP1} + S8 + S9 + S10$$

$$\text{AluOP0} = S2 \text{ OP1} + S3 \text{ OP1} + S6 \text{ OP1} + S7 \text{ OP1}$$

## 6、根据下一状态和输出信号的布尔表达式，画出逻辑图。



# 多周期处理机的控制部件



数据路径总体图



## 专用总线系统和单总线系统

根据需要建立CPU中各个部件之间的数据通路和控制线路，因此其**扩展性较差**，添加新的功能或者部件时需要重新设计数据通路。

为了简化处理机的设计，提高系统的扩展性，提出了单总线结构，即所有的部件都通过相同的数据总线、地址总线和控制总线连接，添加的设备只要符合总线的接口标准就可以挂接到系统中。

由于采用单总线的结构，只能**有一个部件**向总线写数据，可以有**多个部件**接收数据。

# 单总线CPU结构

## 基本构成:

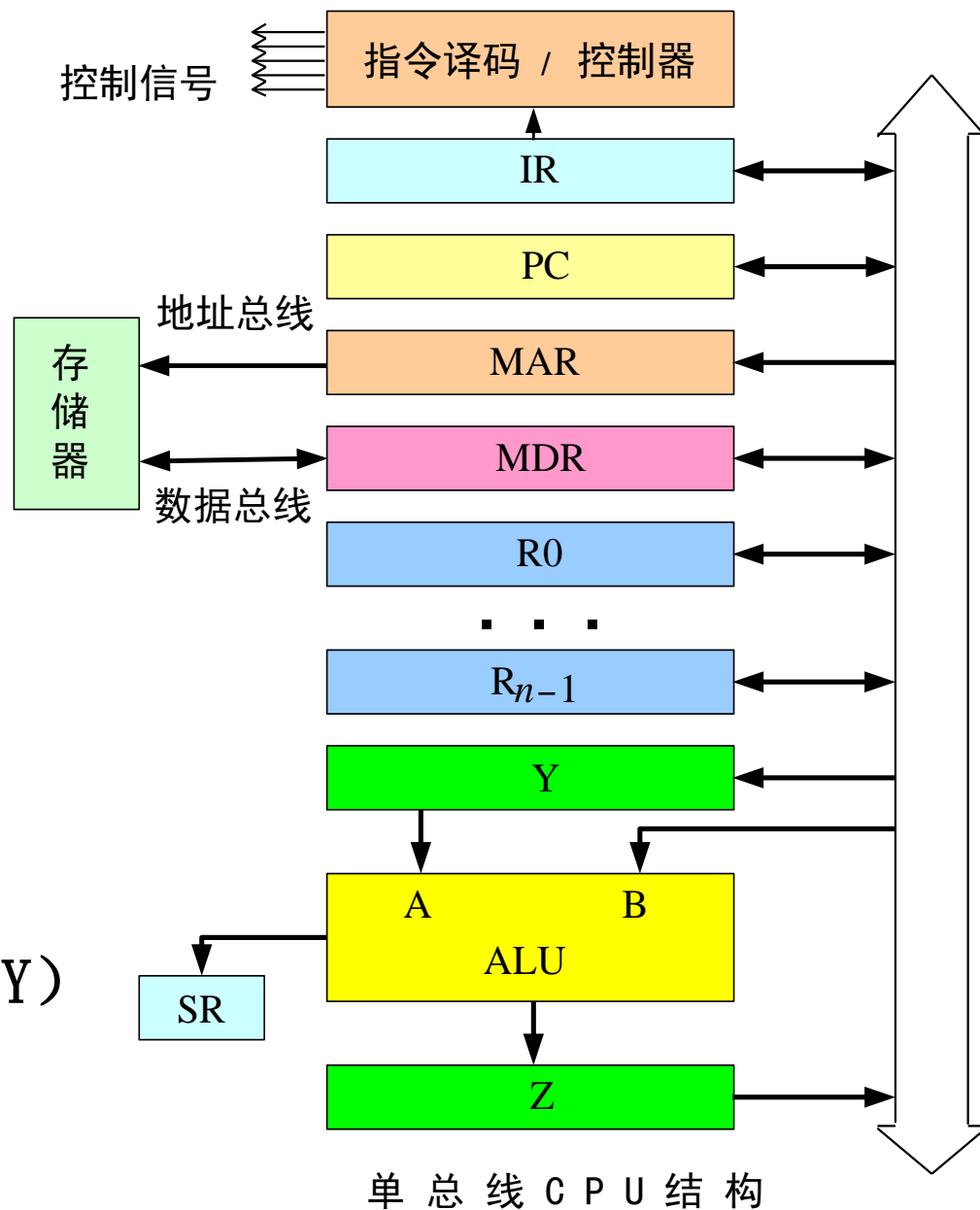
控制器, 运算器,  
寄存器, 数据通路

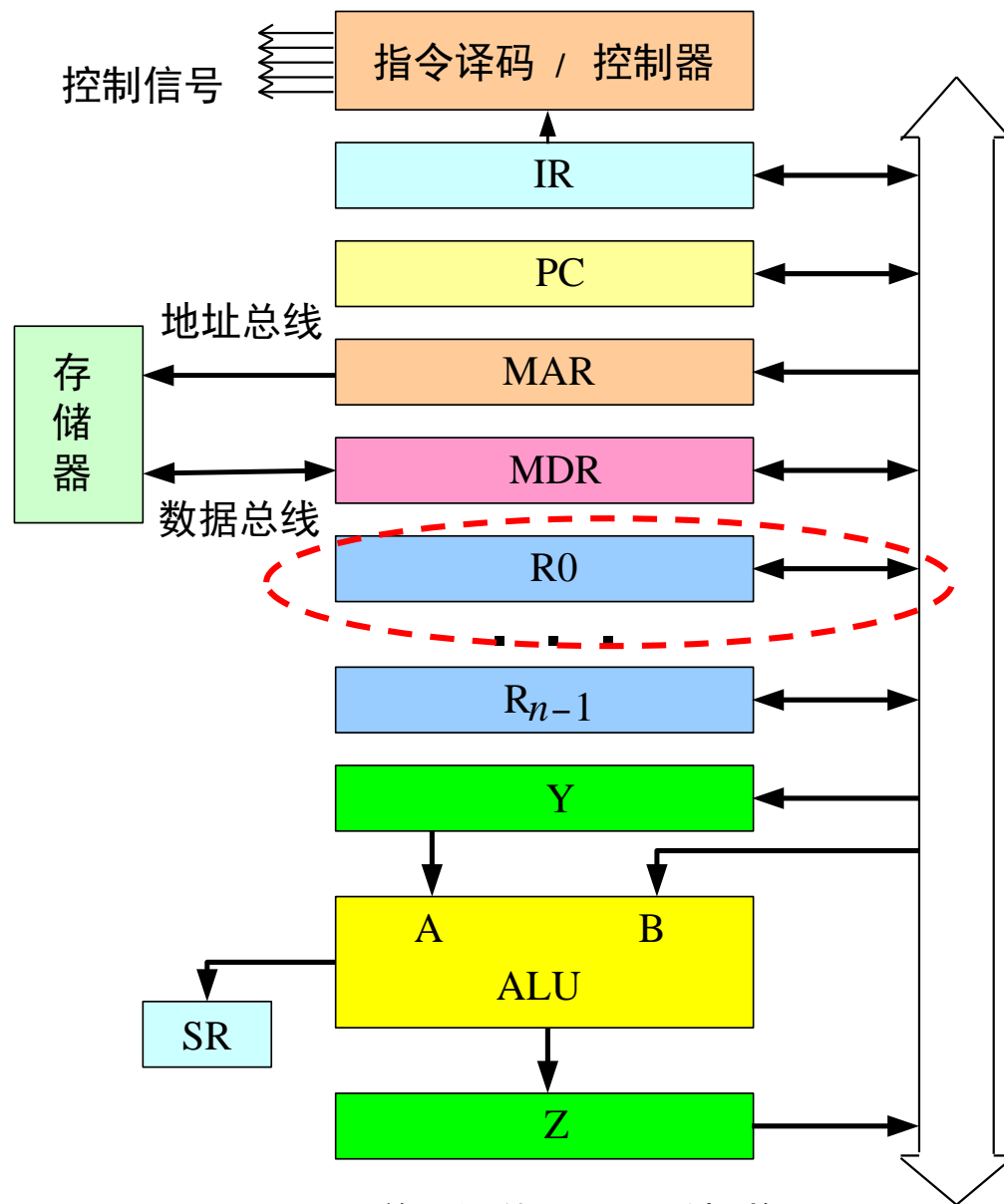
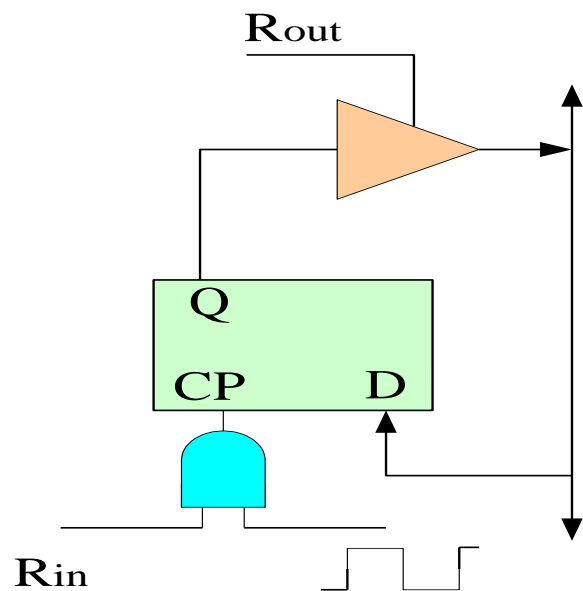
## 寄存器的类型:

指令寄存器 (IR)  
程序计数器 (PC)  
数据寄存器 (MDR)  
地址寄存器 (MAR)  
状态寄存器 (SR)  
通用寄存器 ( $R_i$ )  
用户不可见暂存器 (Z Y)

## 数据通路:

单总线结构





单总线 CPU 结构

# 单总线CPU结构指令的执行过程

## 一、ALU指令的执行过程

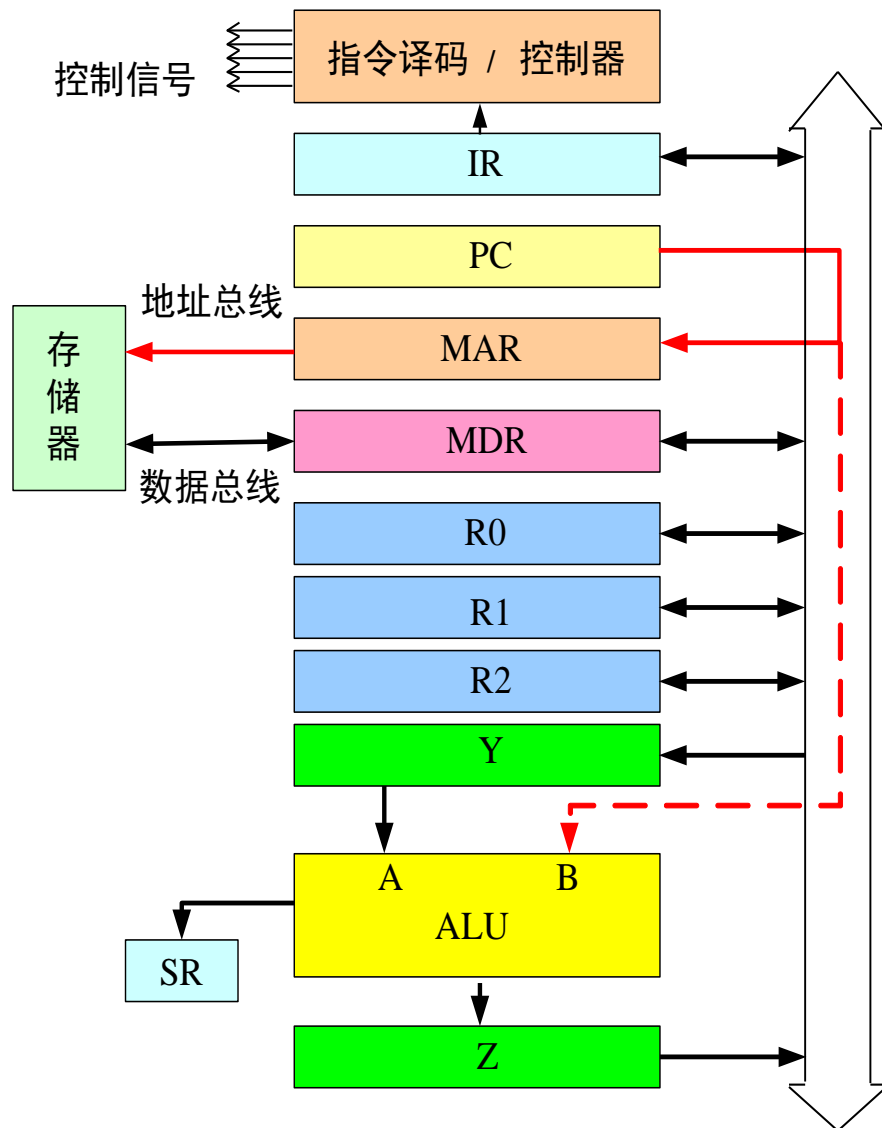
ADD R0, R1, R2

(1) PC → MAR

(2) PC + 1 → PC

a) PC由计数器构成

b) PC由寄存器构成,  
+1操作可由ALU完成。



单总线CPU结构

# 一、ALU指令的执行过程

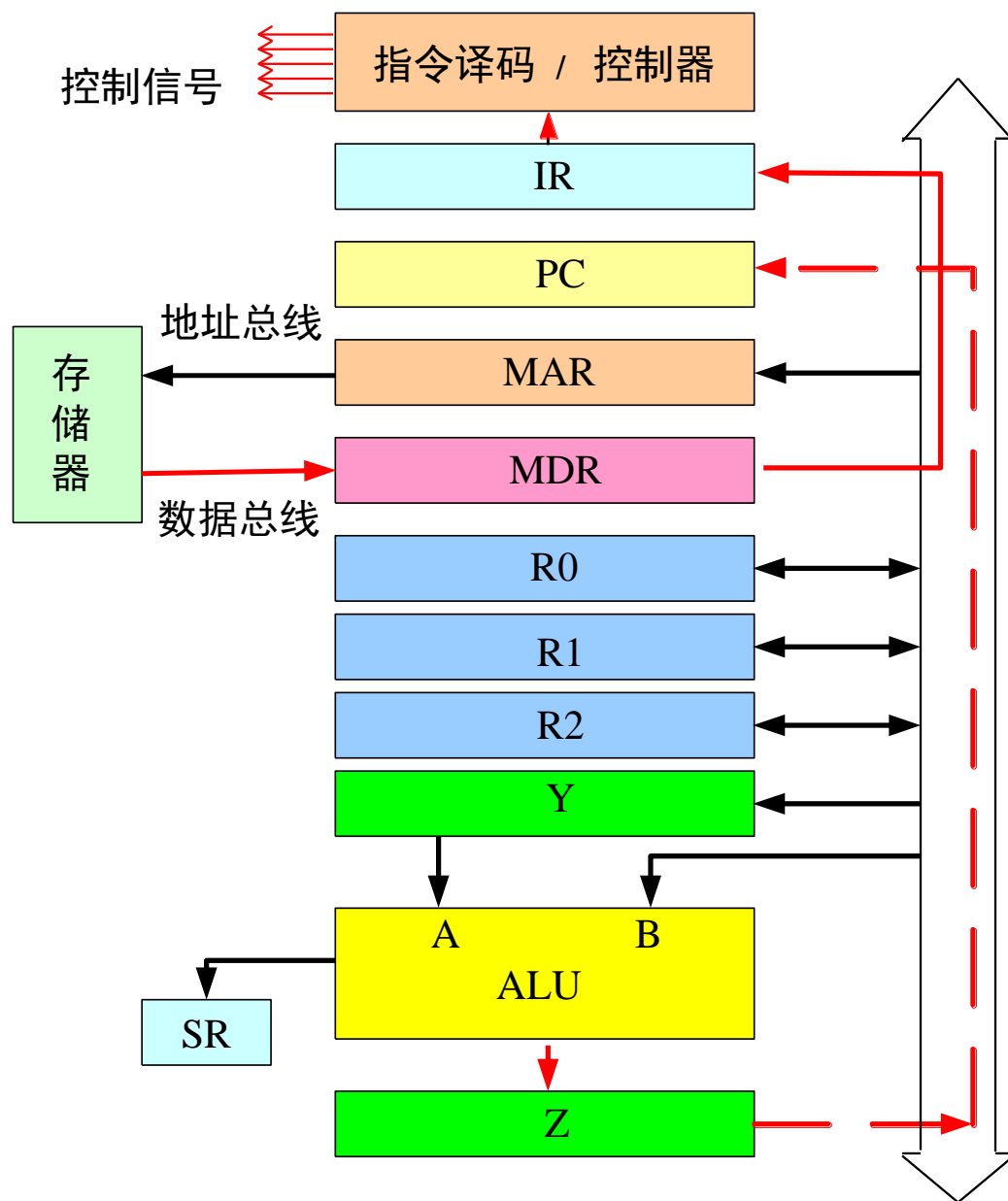
ADD R0, R1, R2

(1) PC→MAR

(2) PC+1→PC

(3) DBUS→MDR

(4) MDR→IR



ADD R0, R1, R2

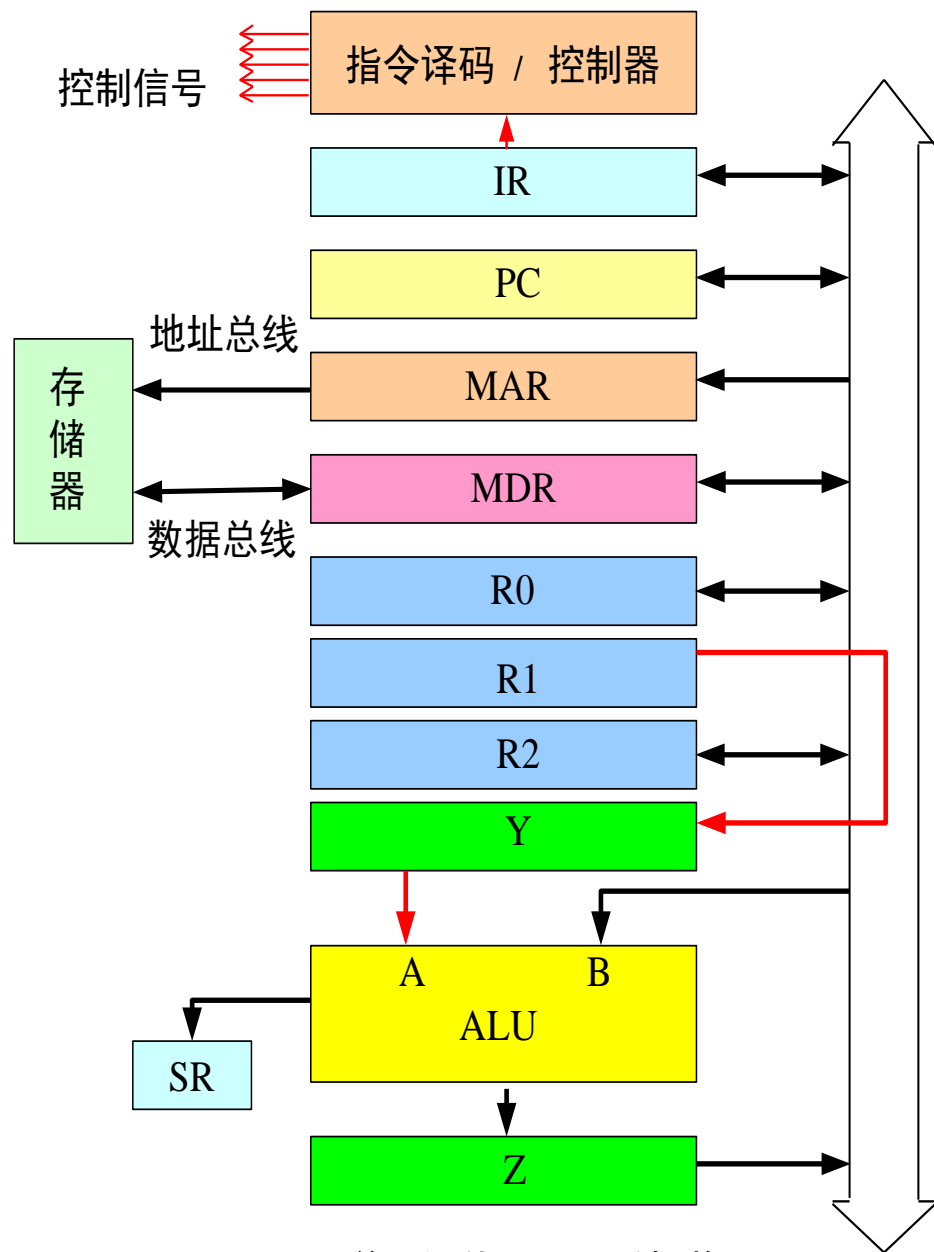
(1)  $PC \rightarrow MAR$

(2)  $PC+1 \rightarrow PC$

(3)  $DBUS \rightarrow MDR$

(4)  $MDR \rightarrow IR$

(5)  $R1 \rightarrow Y$



单总线CPU结构

ADD R0, R1, R2

(1)  $PC \rightarrow MAR$

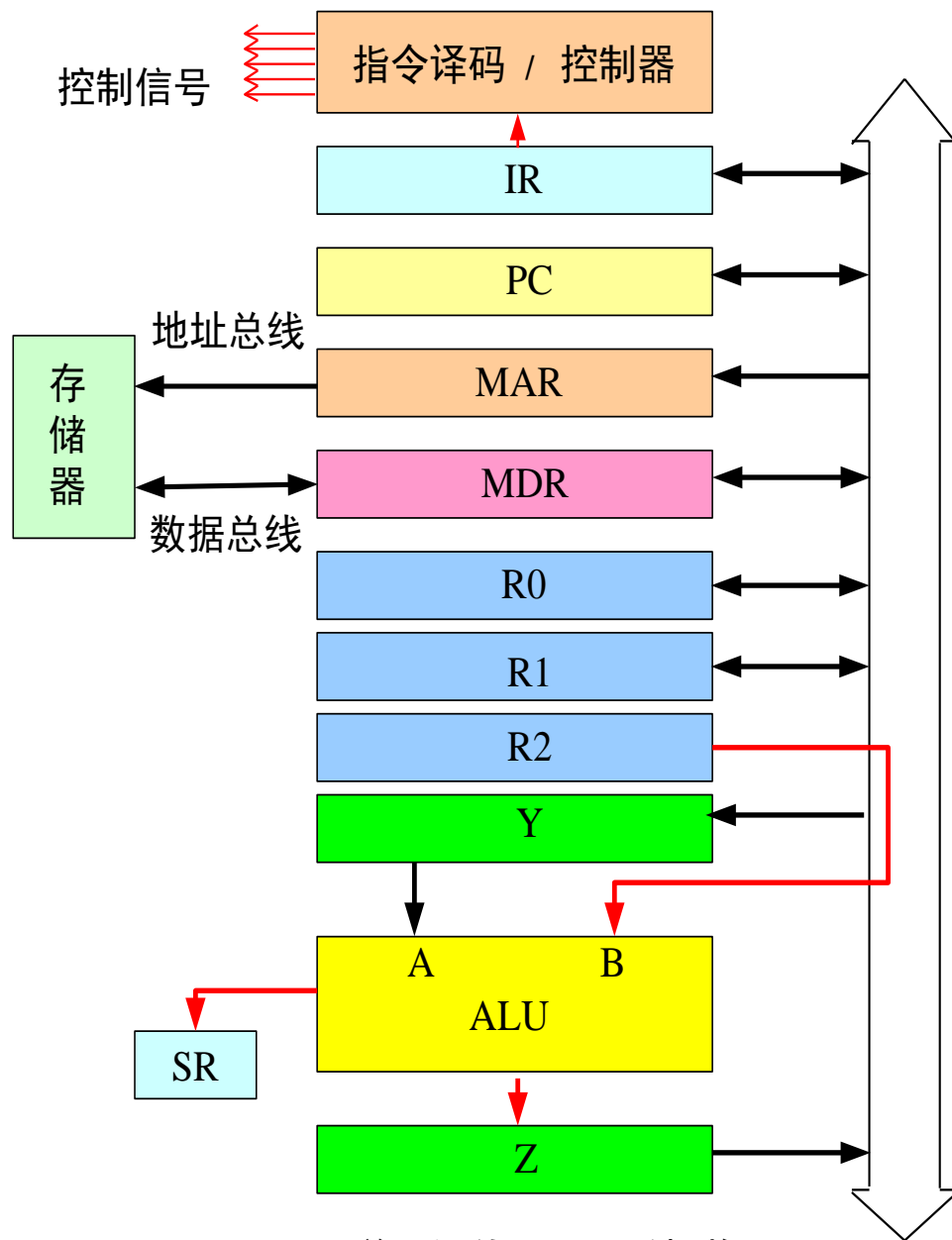
(2)  $PC+1 \rightarrow PC$

(3)  $DBUS \rightarrow MDR$

(4)  $MDR \rightarrow IR$

(5)  $R1 \rightarrow Y$

(6)  $R2 + Y \rightarrow Z$



ADD R0, R1, R2

(1)  $PC \rightarrow MAR$

(2)  $PC+1 \rightarrow PC$

(3)  $DBUS \rightarrow MDR$

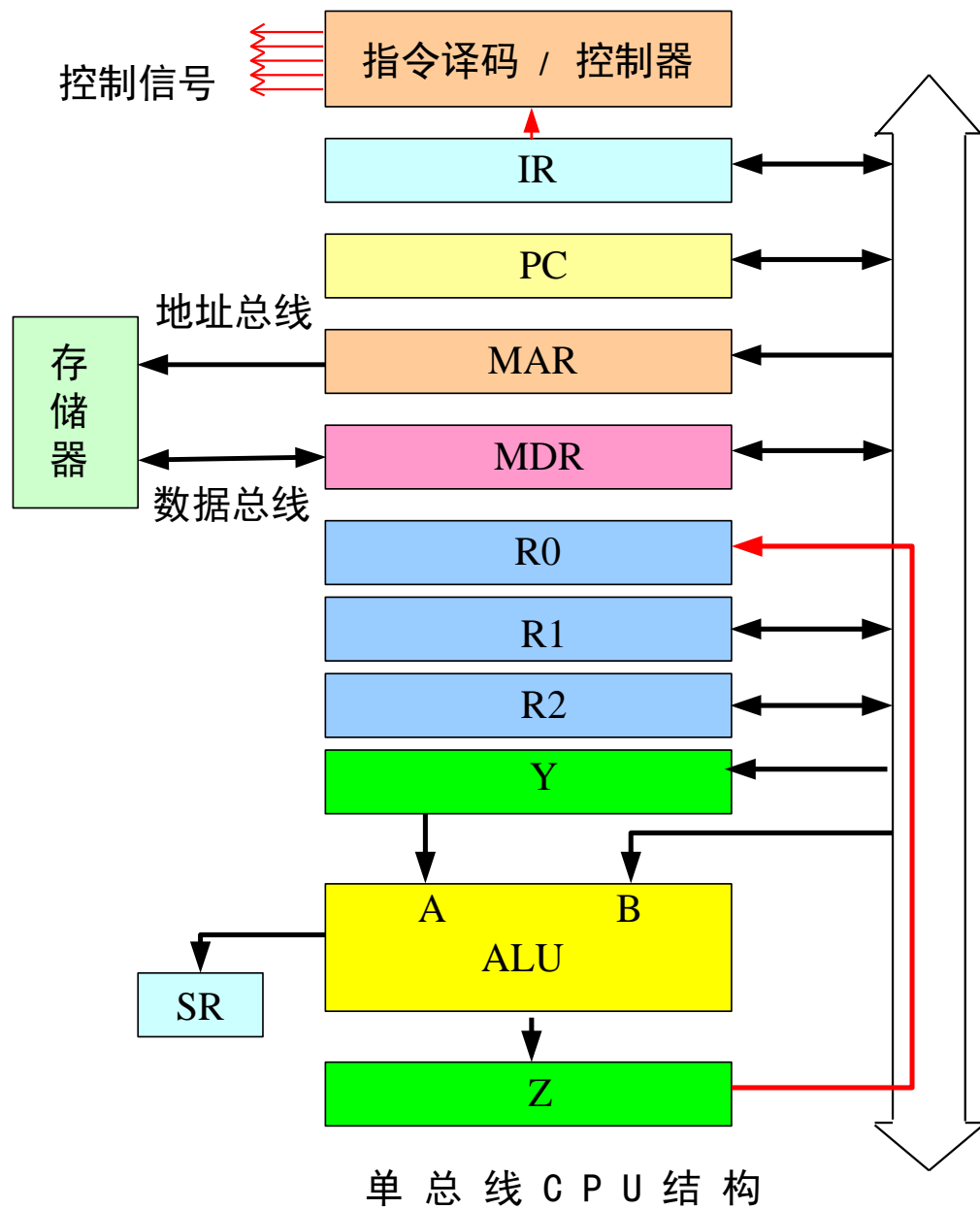
(4)  $MDR \rightarrow IR$

(5)  $R1 \rightarrow Y$

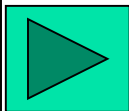
(6)  $R2 + Y \rightarrow Z$

(7)  $Z \rightarrow R0$

ADD指令执行结束



单总线CPU结构



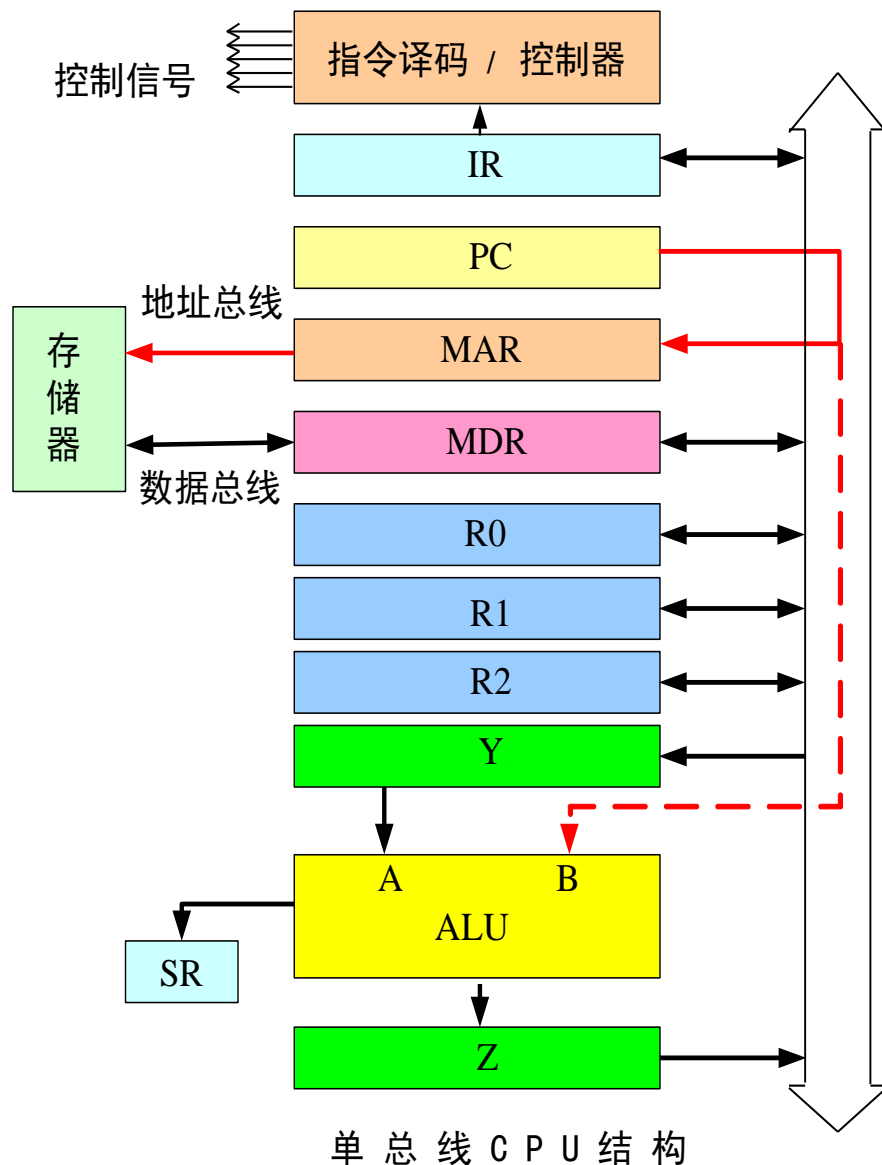


## 二、访存指令的执行过程

装人操作: LOAD R2, mem

(1) PC → MAR

(2) PC + 1 → PC



装人操作: LOAD R2, mem

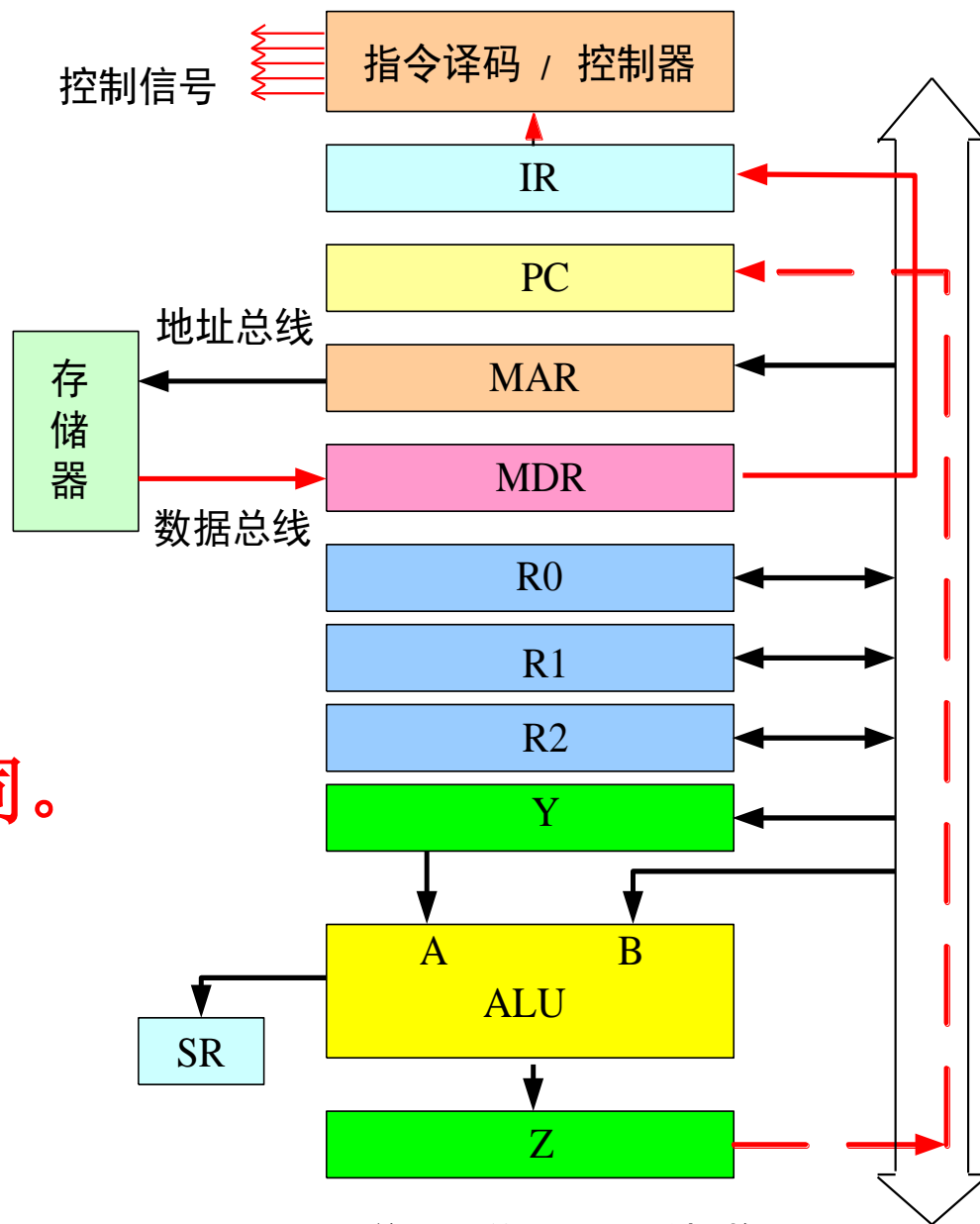
(1) PC → MAR

(2) PC + 1 → PC

(3) DBUS → MDR

(4) MDR → IR

所有指令的开始都相同。  
即取指令阶段



单总线CPU结构

装人操作: LOAD R2, mem

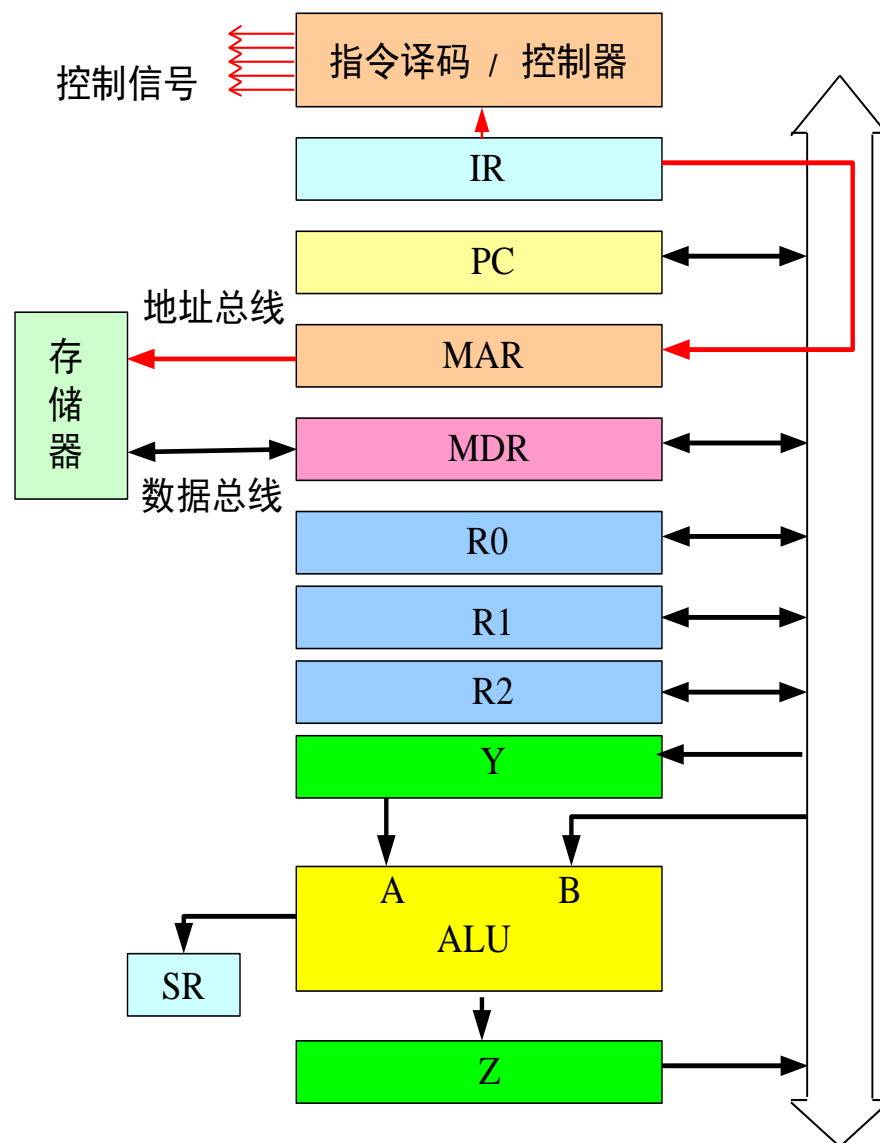
(1) PC → MAR

(2) PC + 1 → PC

(3) DBUS → MDR

(4) MDR → IR

(5) IR(地址段) → MAR, 读  
存储器

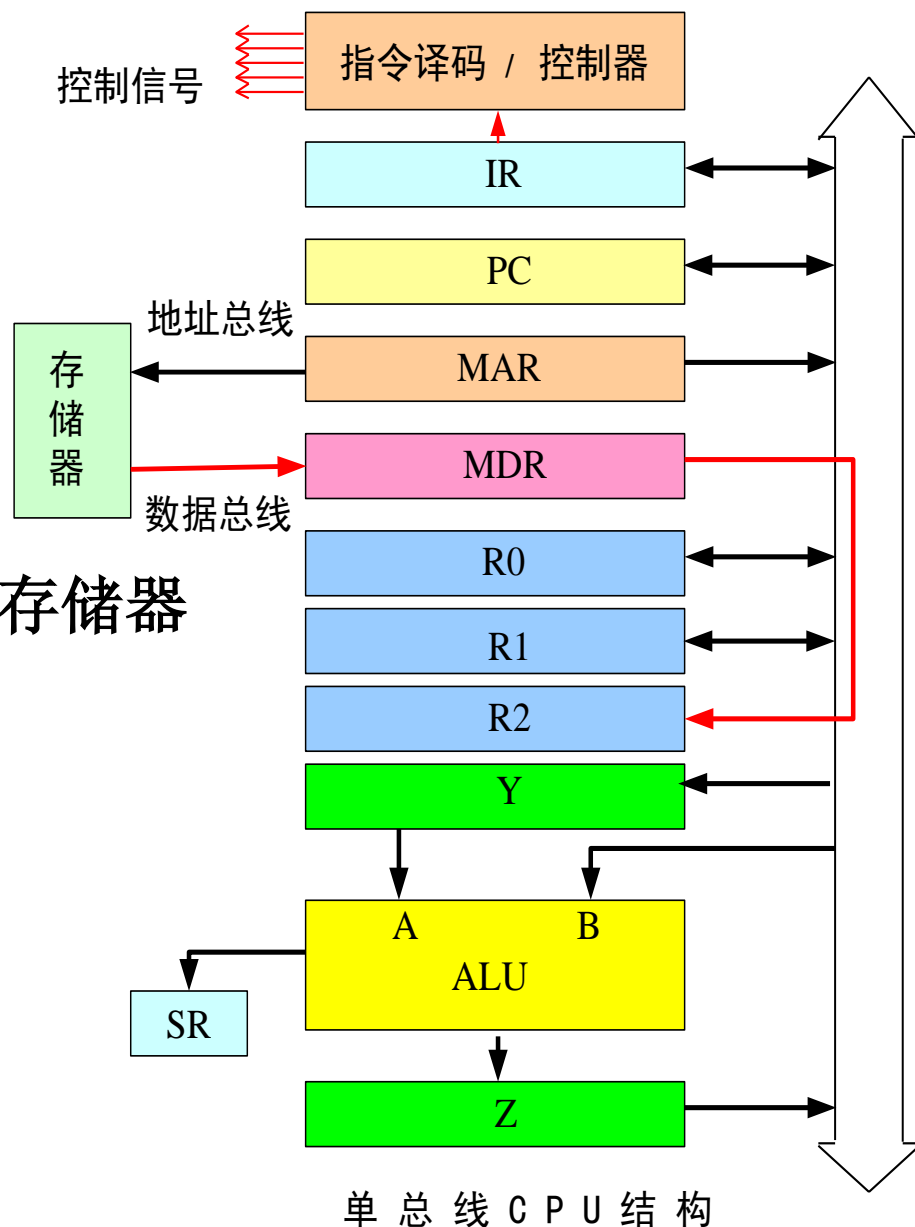


单总线CPU结构

装人操作: LOAD R2, mem

- (1) PC → MAR
- (2) PC + 1 → PC
- (3) DBUS → MDR
- (4) MDR → IR
- (5) IR(地址段) → MAR, 读存储器
- (6) DBUS → MDR
- (7) MDR → R2

LOAD指令执行结束



## 二、访存指令的执行过程

写存储器操作:

STORE R1, mem

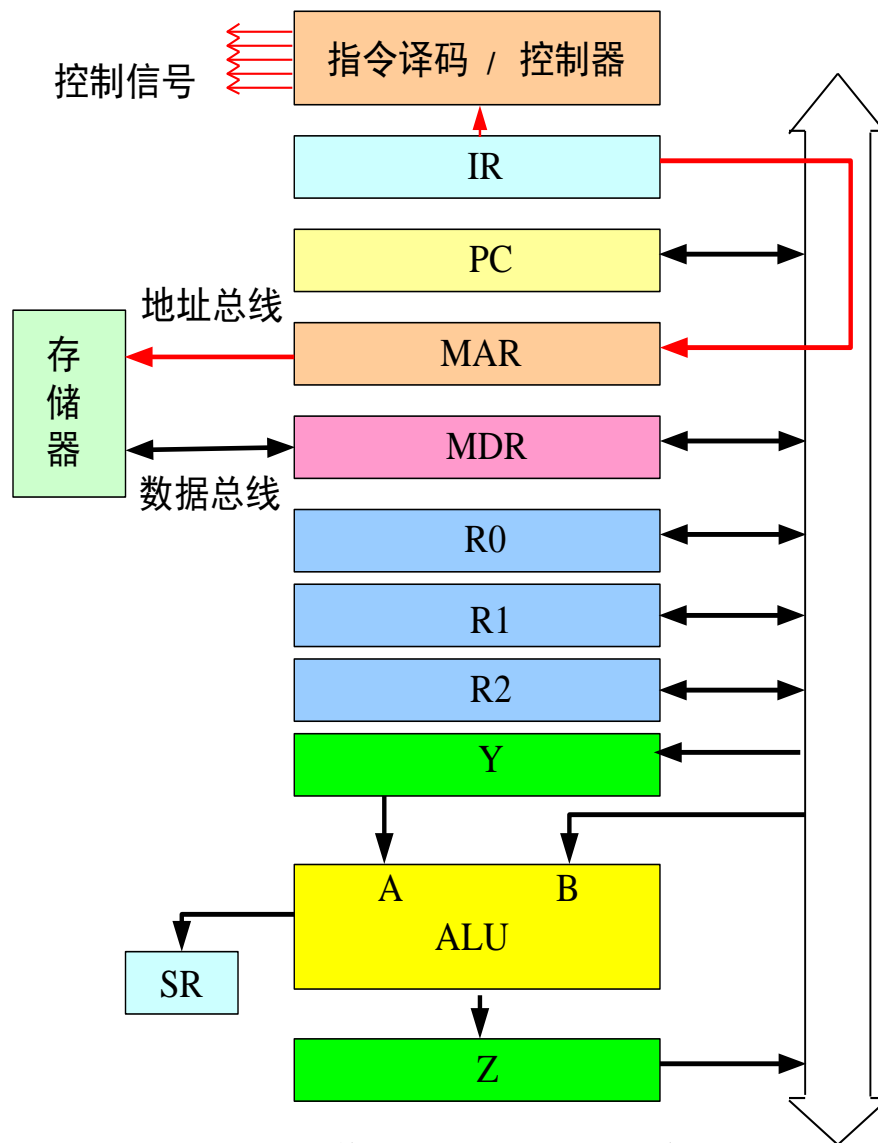
(1) PC → MAR

(2) PC+1 → PC

(3) DBUS → MDR

(4) MDR → IR

(5) IR(地址段) → MAR



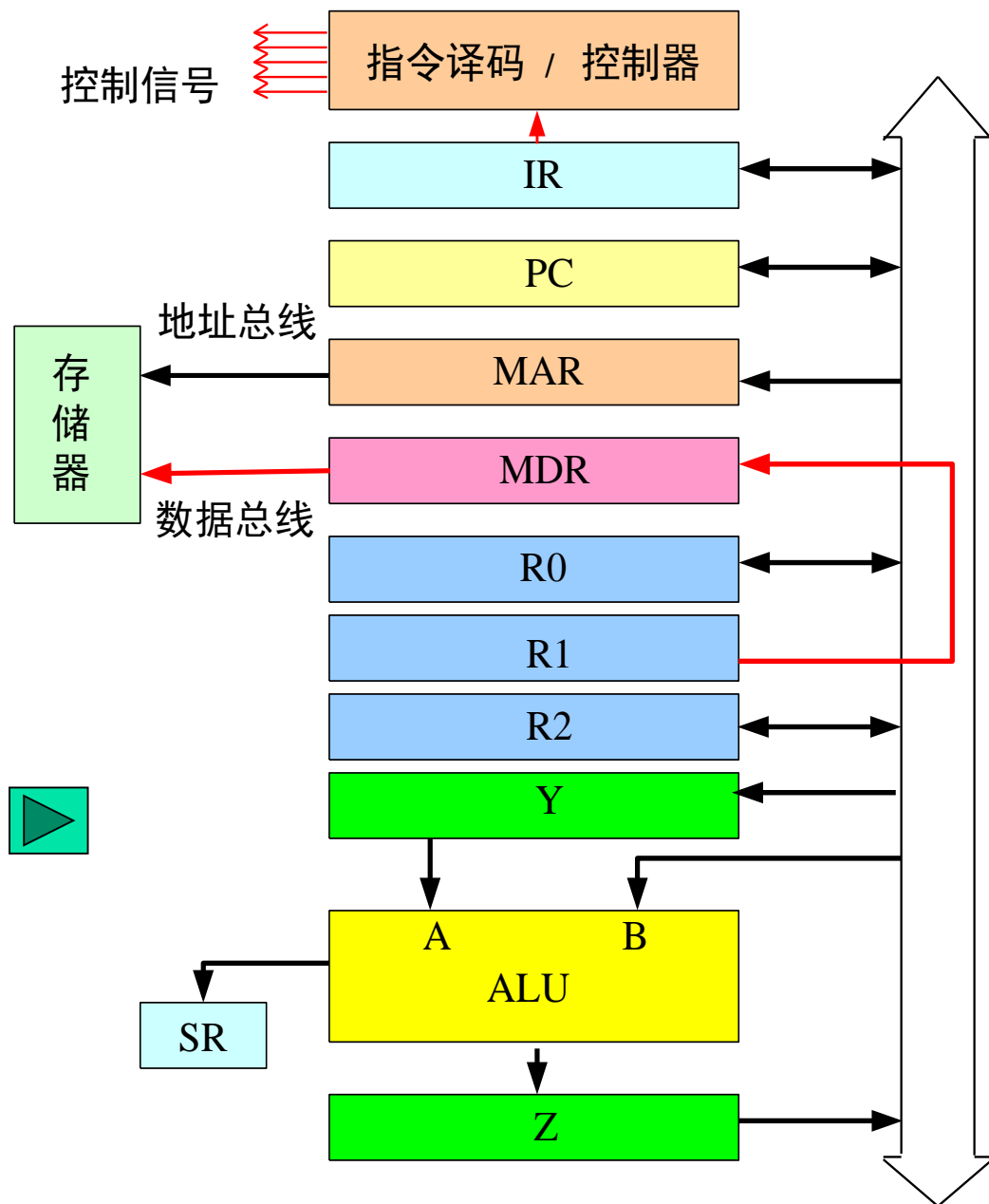
单总线CPU结构

## 写存储器操作:

STORE R1, mem

- (1) PC → MAR
- (2) PC + 1 → PC
- (3) DBUS → MDR
- (4) MDR → IR
- (5) IR(地址段) → MAR
- (6) R1 → MDR, 写存储器

**STORE指令执行结束**

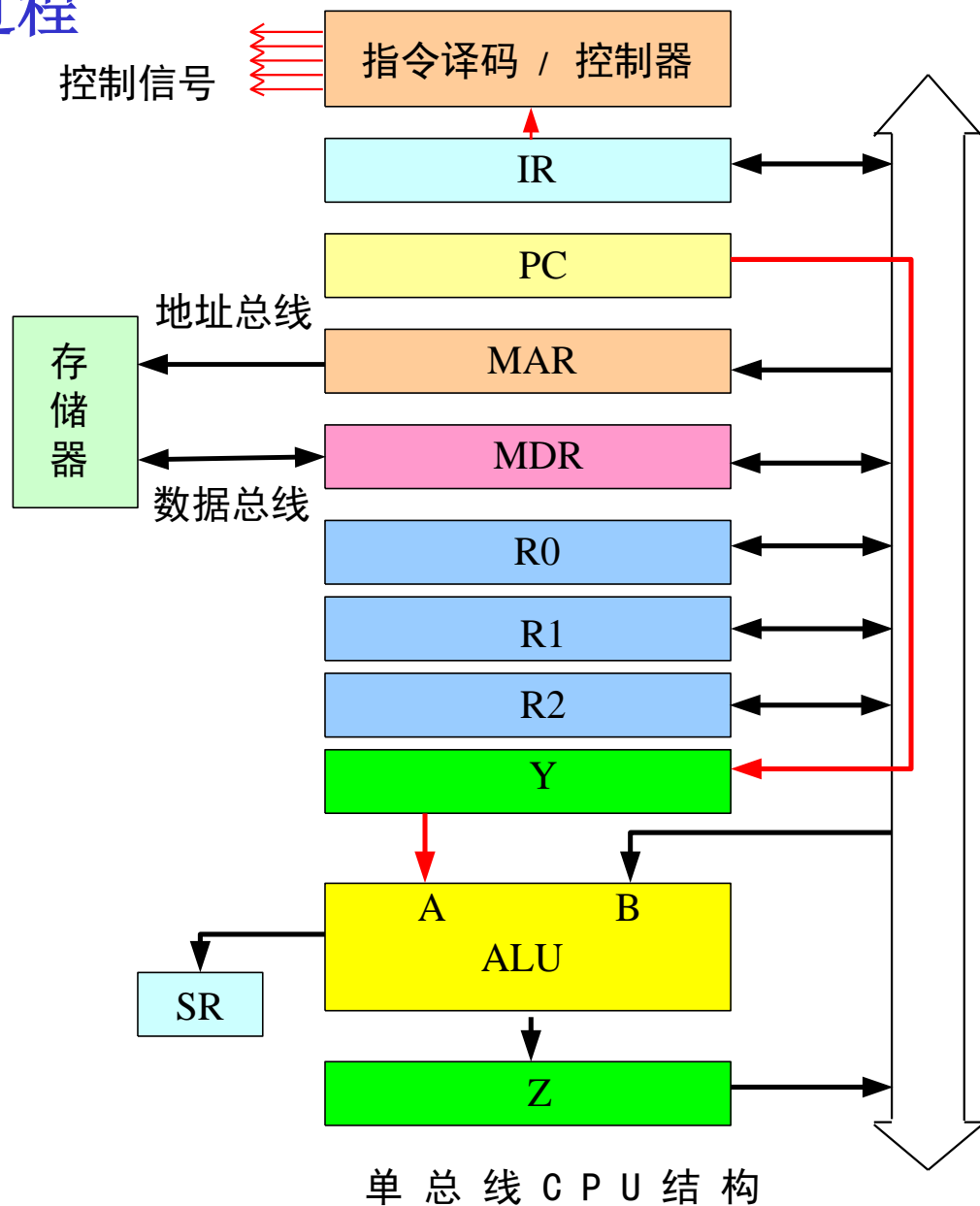


### 三、转移控制指令的执行过程

#### 无条件转移:

Branch disp

- (1)  $PC \rightarrow MAR$
- (2)  $PC+1 \rightarrow PC$
- (3)  $DBUS \rightarrow MDR$
- (4)  $MDR \rightarrow IR$
- (5)  $PC \rightarrow Y$



单总线CPU结构

## 无条件转移:

Branch disp

(1)  $PC \rightarrow MAR$

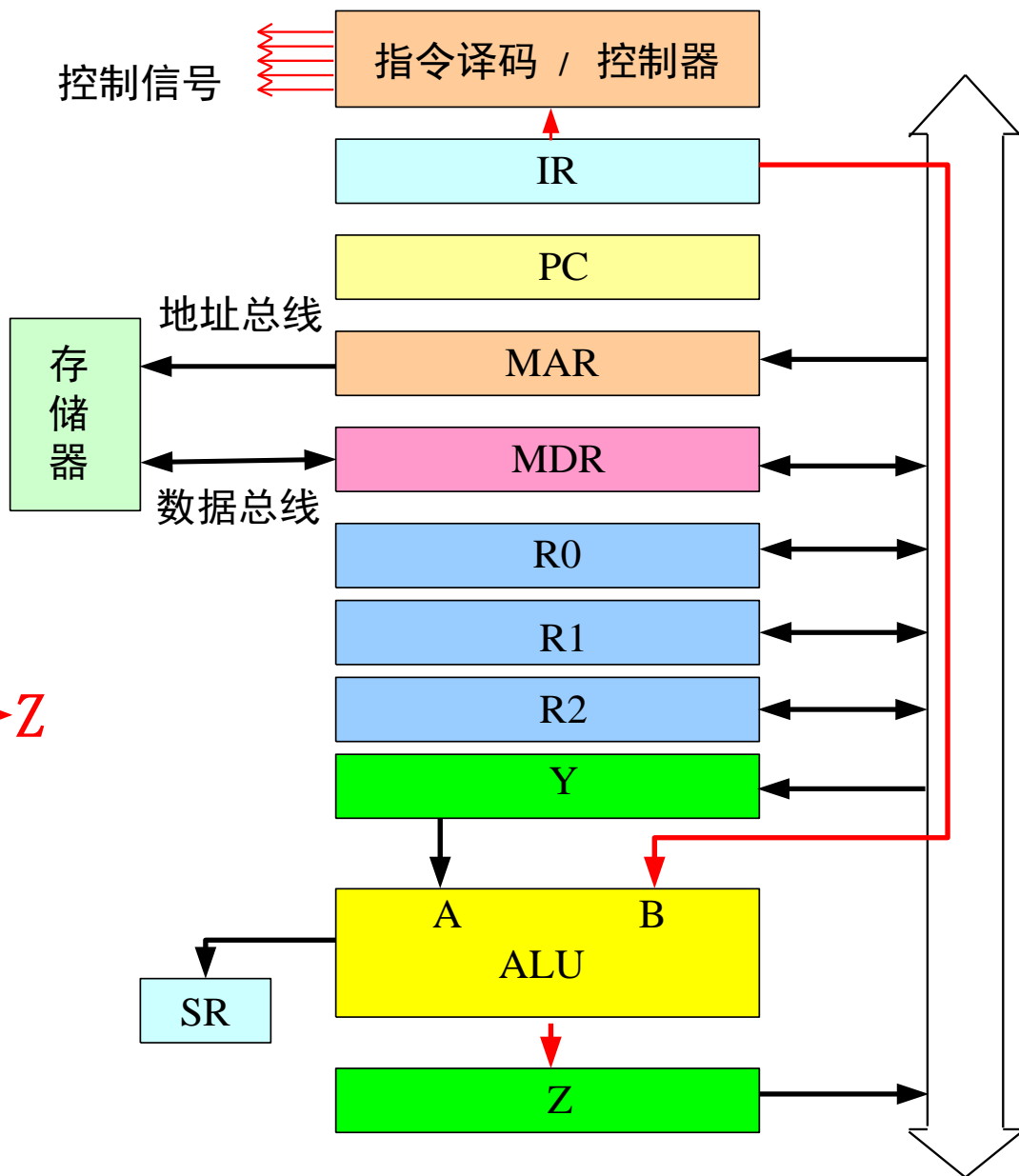
(2)  $PC+1 \rightarrow PC$

(3)  $DBUS \rightarrow MDR$

(4)  $MDR \rightarrow IR$

(5)  $PC \rightarrow Y$

(6)  $Y + IR(\text{地址段}) \rightarrow Z$



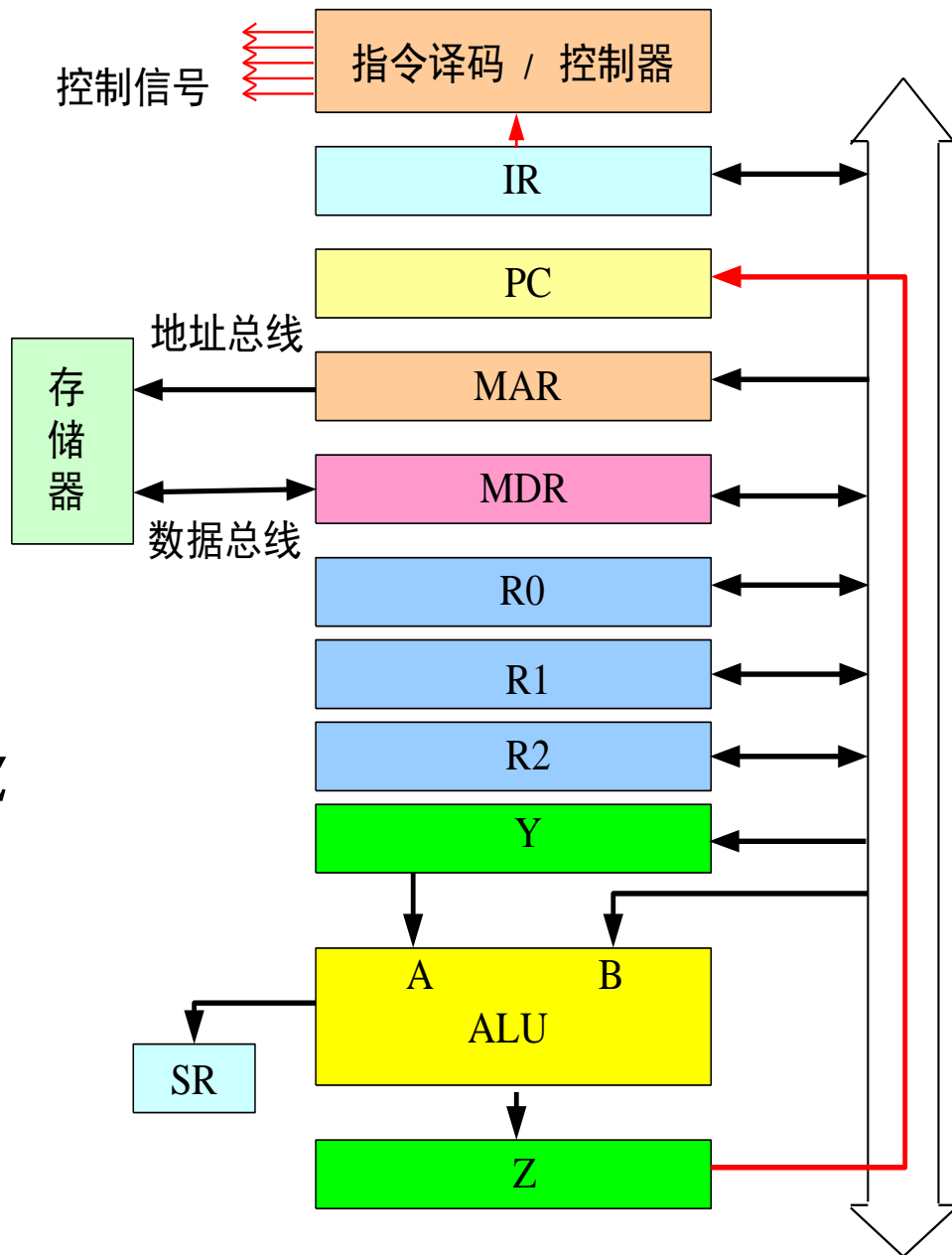


## 无条件转移:

Branch disp

- (1)  $PC \rightarrow MAR$
- (2)  $PC+1 \rightarrow PC$
- (3)  $DBUS \rightarrow MDR$
- (4)  $MDR \rightarrow IR$
- (5)  $PC \rightarrow Y$
- (6)  $Y + IR(\text{地址段}) \rightarrow Z$
- (7)  $Z \rightarrow PC$

BRANCH指令执行结束



单总线CPU结构

## 条件转移:

BNE disp

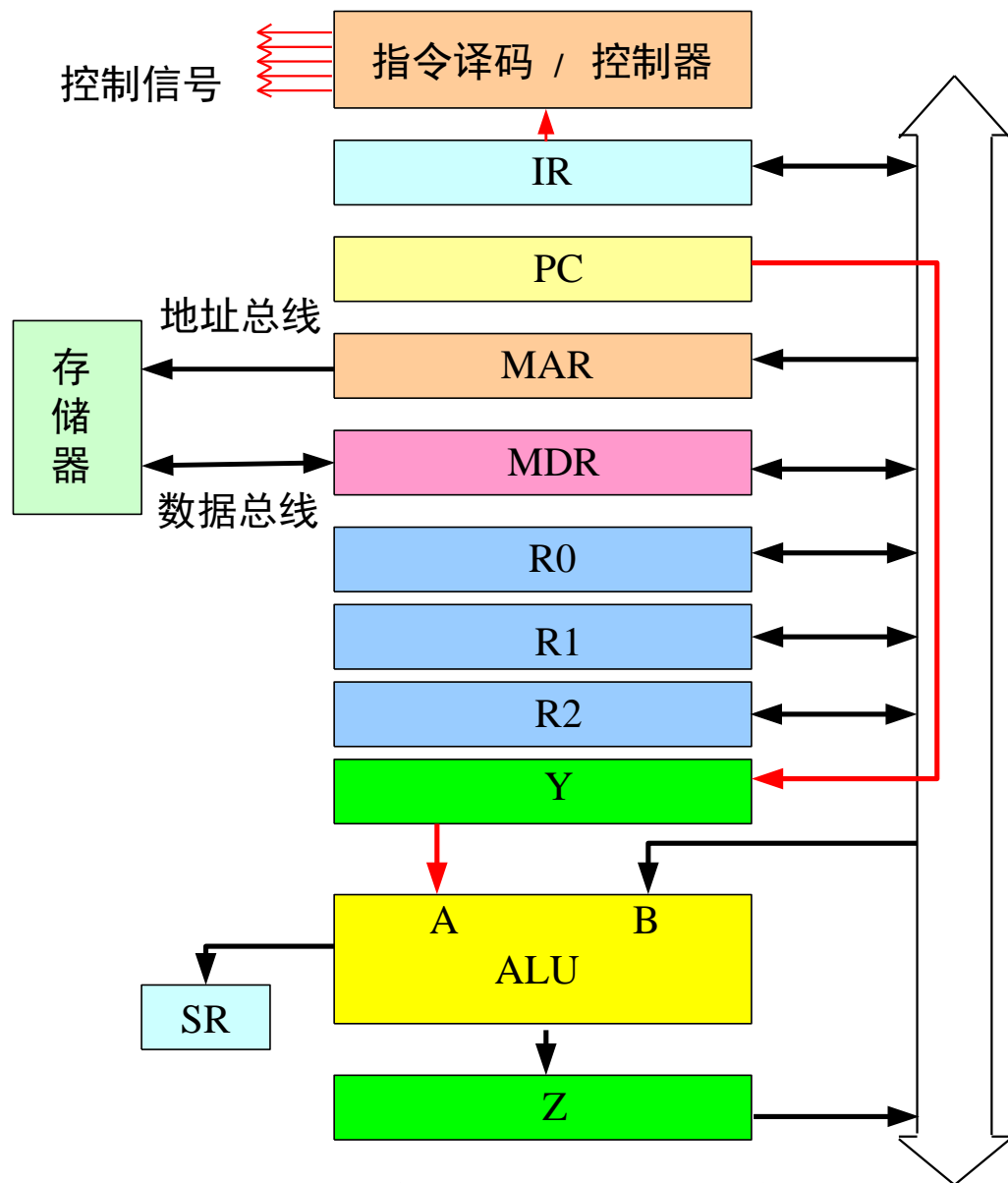
(1) PC→MAR

(2) PC+1→PC

(3) DBUS→MDR

(4) MDR→IR

(5) if(!Z) PC→Y;  
else goto END



单总线CPU结构

## 条件转移:

BNE disp

(1)  $PC \rightarrow MAR$

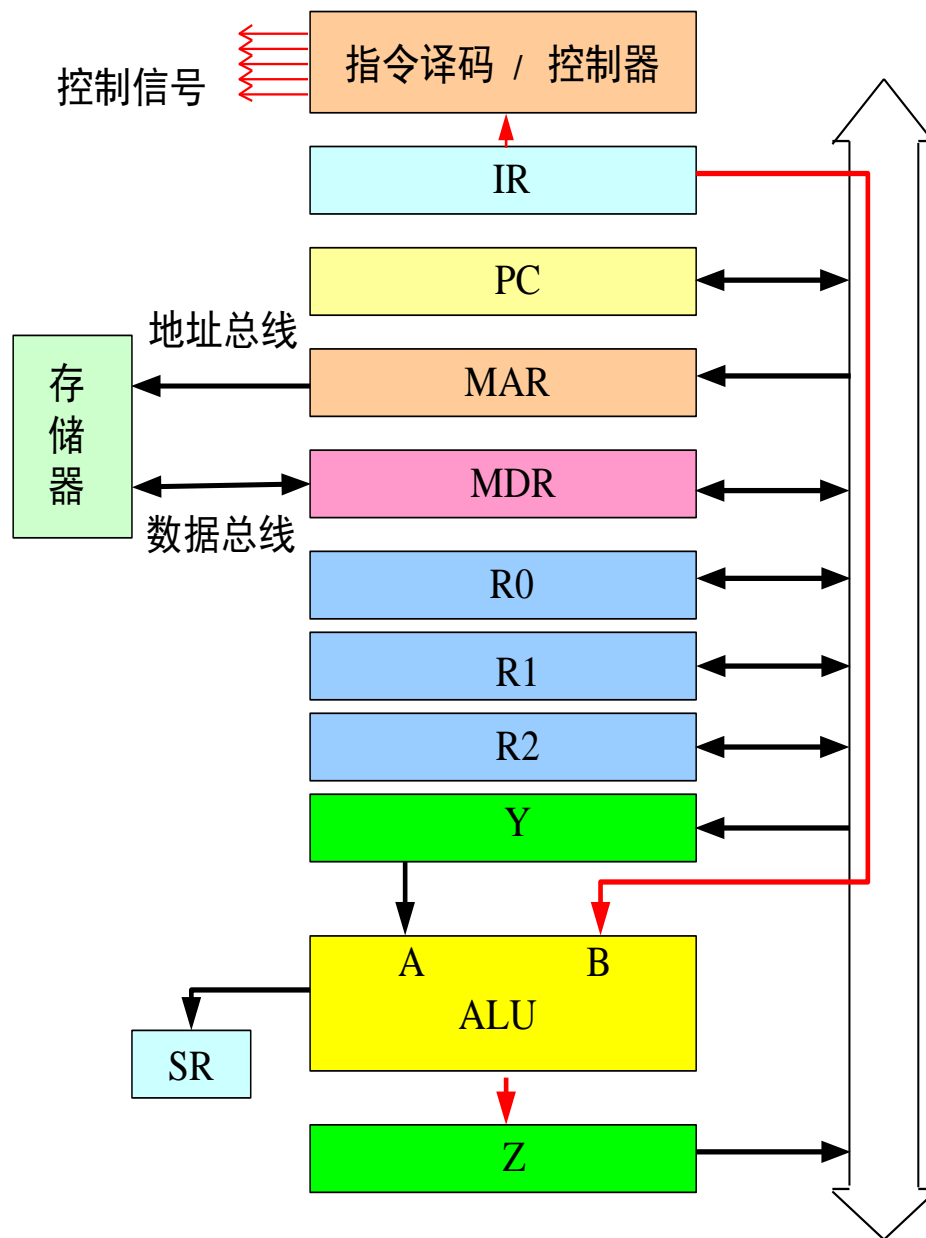
(2)  $PC+1 \rightarrow PC$

(3)  $DBUS \rightarrow MDR$

(4)  $MDR \rightarrow IR$

(5) if (!Z)  $PC \rightarrow Y$ ; else  
goto END

(6)  $Y + IR(\text{地址段}) \rightarrow Z$



单总线CPU结构

## 条件转移:

BNE disp

(1)  $PC \rightarrow MAR$

(2)  $PC+1 \rightarrow PC$

(3)  $DBUS \rightarrow MDR$

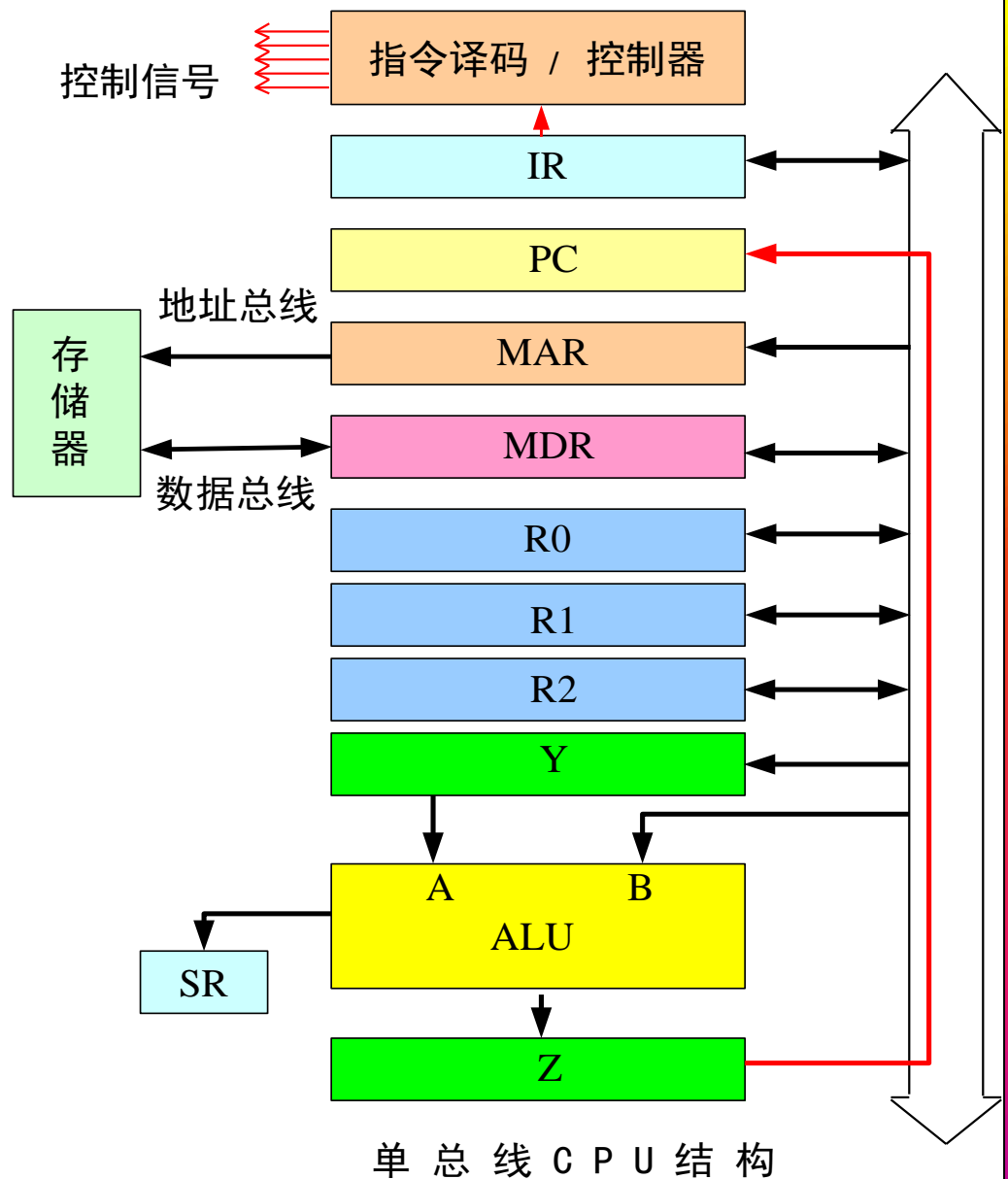
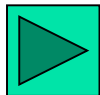
(4)  $MDR \rightarrow IR$

(5) if (!Z)  $PC \rightarrow Y$ ; else  
goto END

(6)  $Y + IR(\text{地址段}) \rightarrow Z$

(7)  $Z \rightarrow PC$

**BNE指令执行结束**



注意、前面的执行是假设整条指令是一次取出。

如果**不能**一次取出，情况又会是怎样？

例如： `mov r1, add_mem ; R1 ← (add_mem )`

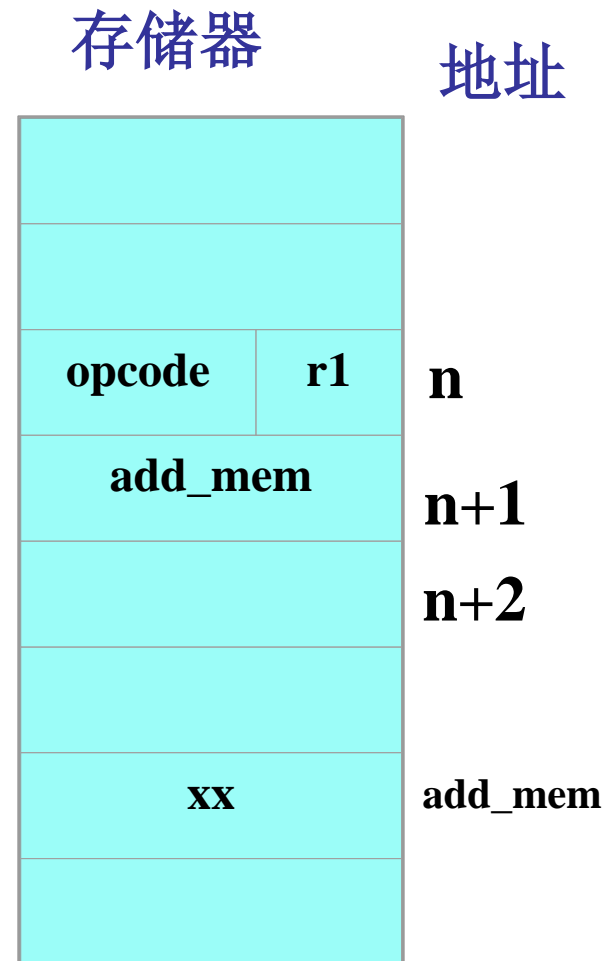
指令格式：

5位	3位	8位
opcode	Rd	mem

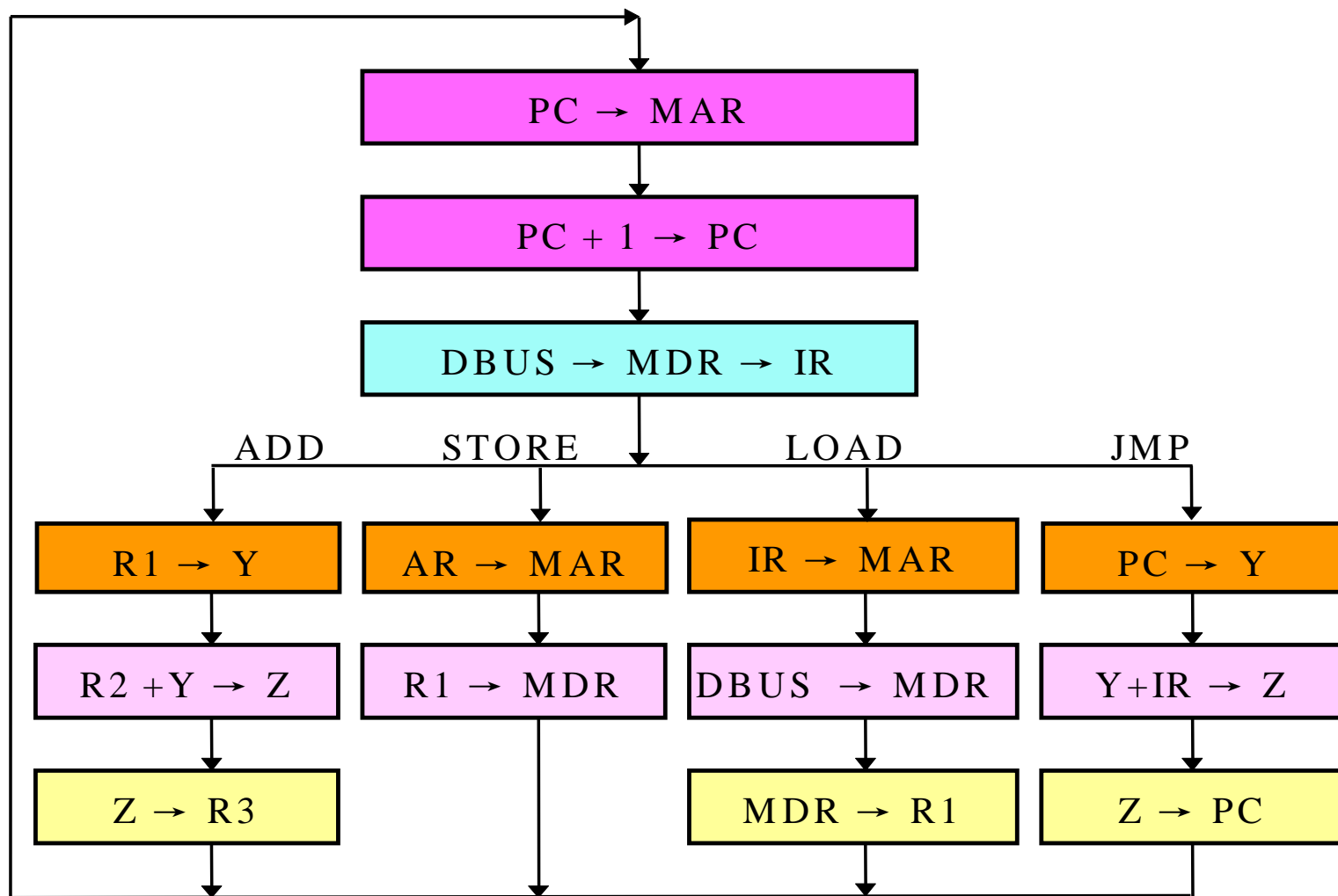
假定处理机字长8位，存储器的读写长度为8为，那么16位指令长度将分两次完成

**mov r1, add\_mem**

- (1) PC→MAR
- (2) PC+1→PC
- (3) DBUS→MDR
- (4) MDR→IR
- (5) PC→MAR ; (PC)=n+1
- (6) PC+1→PC ; (PC)=n+2,  
指向下一条指令
- (7) DBUS→MDR; (n+1)=add\_mem
- (8) MDR →MAR; 以add\_mem为地址,  
访问存储器
- (9) DBUS→MDR; 取出add\_mem单元的内容
- (10) MDR →R1; (add\_mem) →r1



# 指令流程图



一个操作步骤代表在一个机器周期中可完成的操作

# 执行步骤所需的控制信号

分支	操作	时钟周期	控制信号
取指令	PC→MAR, PC+1, read	T1	PCout, MARin, PC+1, Read
	MDR→IR	T2	MDRout, IRin
ADD 操作	R1→Y	T3	R1out, Yin
	R2+Y→Z	T4	R2out, Zin, ADD
	Z→R3	T5	Zout, R3in
LOAD 操作	IR→MAR, read	T3	IRout, MARin, Read
	MDR→R1	T4	MDRout, R1in
STORE 操作	IR→MAR	T3	IRout, MARin
	R1→MDR, write	T4	R1out, MDRin, Write
JMP 操作	PC→Y	T3	PCout, Yin
	IR+Y→Z	T4	IRout, ADD, Zin
	Z→PC	T5	Zout, PCin



# 微操作控制形成电路的逻辑表达形式

$$C = T1*(INS1 + INS2 + ...) + T2*(INS1 + INS2 + ...) + \dots$$

ADD指令每个时钟周期内的控制信号为:

T1: PCout, MARin, PC+1, Read ;PC→MAR, PC+1, read

T2: MDRout, IRin ;MDR→IR

T3: R1out, Yin ;R1→Y

T4: R2out, Zin, Add ;R2+Y→Z

T5: Zout, R3in ;Z→R3

JMP指令中各时钟周期的控制信号为:

T1: PCout, MARin, PC+1, Read ;PC→MAR, PC+1, read

T2: MDRout, IRin ;MDR→IR

T3: PCout, Yin ;PC→Y

T4: IRout, Add, Zin ;IR+Y→Z

T5: Zout, PCin ;Z→PC

# 控制器的逻辑表达式

$$PC+1 = T1$$

$$PCin = T5 * JMP$$

$$PCout = T1 + T3 * JMP$$

$$Yin = T3 * (ADD + JMP)$$

$$Add = T4 * (ADD + JMP)$$

$$Zin = T4 * (ADD + JMP)$$

$$Zout = T5 * (ADD + JMP)$$

$$END = T5 * (ADD + JMP)$$

...

# 组合逻辑控制器的特点

**优点:**速度快, 可用于速度要求较高的机器中.

**缺点:** (1) **缺乏规整性:** 将几百个微操作的执行逻辑组合在一起, 构成的微操作产生部件, 是计算机中最复杂、最不规整的逻辑部件. 不适合于指令复杂的机器.

(2) **缺乏灵活性:** 各微命令的实现是用硬连的逻辑电路完成, 改动不易, 设计困难.

# PLA控制器

PLA控制器的设计步骤与组合逻辑控制器相同, 只是实现方法不同, 它采用**PLA阵列** (Programmed Logic Array). 从设计思想来看是组合逻辑控制器, 从实现方法来看, 是存储逻辑控制器.

**特点:** 可使杂乱无章的组合逻辑规整化、微型化, 而且可以利用PLA的可编程特性, 用存储逻辑部分地取代组合逻辑, 增加了一定的灵活性。

# 微程序控制器

**微程序控制的基本思想**，就是仿照通常的解题程序的方法，把操作控制信号编成所谓的“微指令”，存放到一个只读存储器里。当机器运行时，一条又一条地读出这些微指令，从而产生全机所需要的各种操作控制信号，使相应部件执行所规定的操作。

**组合逻辑**电路一经实现，不能变动其逻辑关系，必要时，必须改变其连线或重新设计。

**微程序控制方法**：把指令执行所需要的所有控制信号存放在控制存储器中，需要时从这个存储器中读取，**存储逻辑**可以修改ROM存放的数据，从而修改逻辑功能，速度略慢，有一个寻址和读数据的过程。

**微程序控制的特点**：灵活性好，速度慢

## 微程序控制方式的历史简介

由莫里斯·威尔克斯（**1967** 图灵奖）。1946年10月，以冯·诺伊曼的EDVAC为蓝本设计建造了EDSAC。它使用了水银延迟线作存储器，穿孔纸带为输入设备和电传打字机为输出设备。EDSAC是第一台诺依曼机器结构的电子计算机。在设计与制造EDSAC和EDSAC2的过程中，威尔克斯创造和发明了许多新的技术概念。诸如“变址”、“宏指令”、微程序、子例程及子例程库、高速缓冲存储器（Cache）等等，这些都对现代计算机的体系结构和程序设计技术产生了深远的影响。

# 微程序控制方式

- 1 基本概念
- 2 基本结构
- 3 实验系统概况
- 4 小结

# 微程序控制的基本概念

## 1. 微命令与微操作

**微命令**: 构成控制信号序列的最小单位。

**微操作**: 控制器中执行部件接受微指令后所进行的操作。

## 2. 微指令和微程序

**微指令**: 在机器的一个节拍中, 一组实现一定操作功能的微命令, 或者说, 控制存储器中每个单元存放的微命令信息组成一条微指令。

**微程序**: 由微指令组成的序列称为微程序, 一个微程序的功能对应一条机器指令的功能。



### 3. 机器指令与微指令

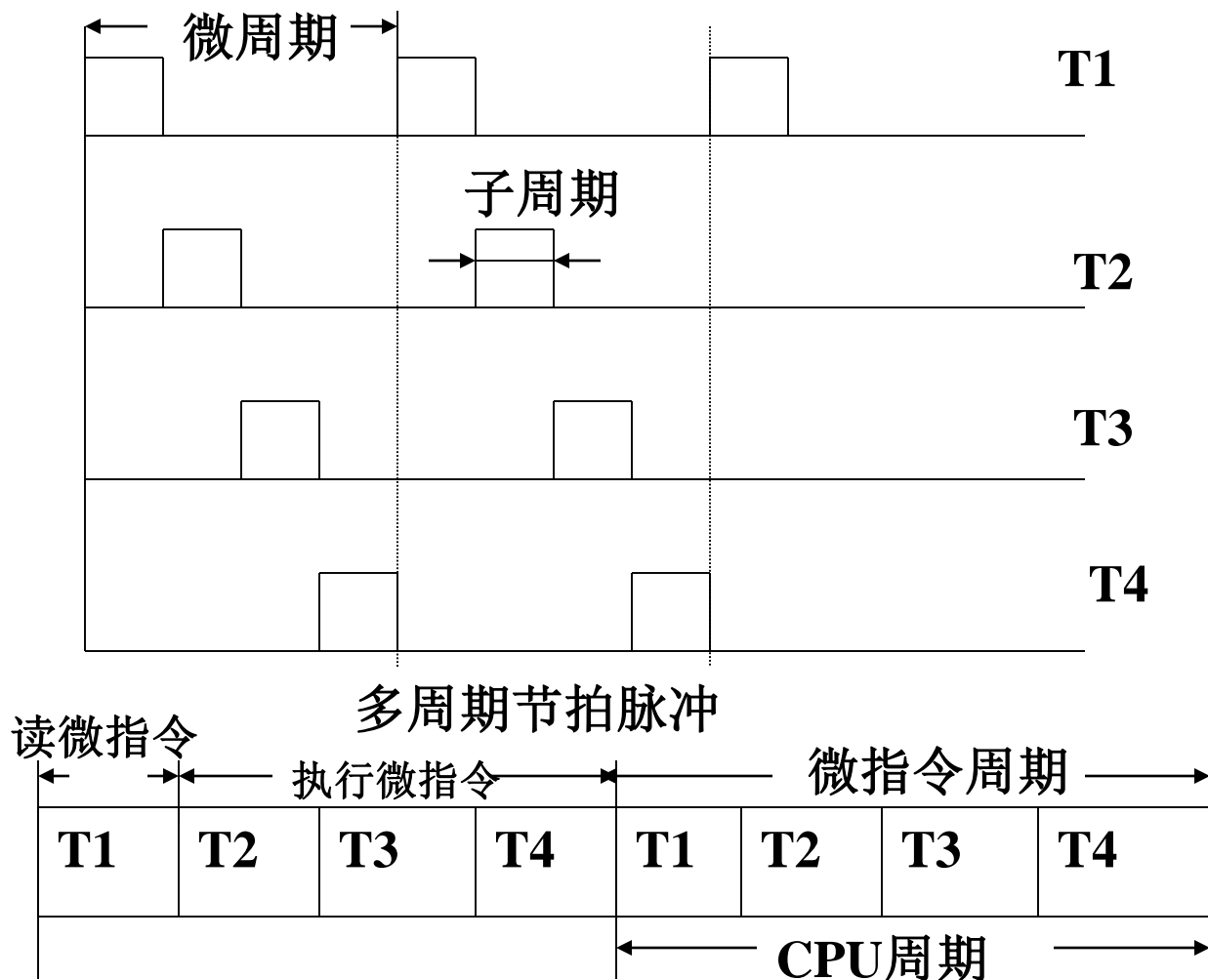
机器指令指提供给使用者编成的基本单位, 每一条指令可以完成一个独立的算术运算或逻辑运算操作.

**一条机器指令对应一组微指令组成的微程序.** 可见, 一条机器指令对应多条微指令, 而一条微指令可为多个机器指令服务.

### 4. 控制存储器CM(Control Memory):

用于存放全部指令的所有微程序, 采用只读存储器结构(固化). 控制存储器的字长等于微指令的长度, 其总容量决定于所有微程序的总长度.

**5. 微指令周期:**从控制存储器中读取一条微指令并执行这条微指令所需的时间,通常一个微指令周期与一个CPU周期的时间相等.微指令中的微命令可以用节拍脉冲来同步定时.



**CPU周期与微指令周期的关系**

# 微程序控制器的基本结构和工作过程

## 一、基本组成

### 1、控制存储器CM

用来存放微程序。

### 2、微指令寄存器 $\mu\text{IR}$

用来存放从控制存储器中取得的微指令。

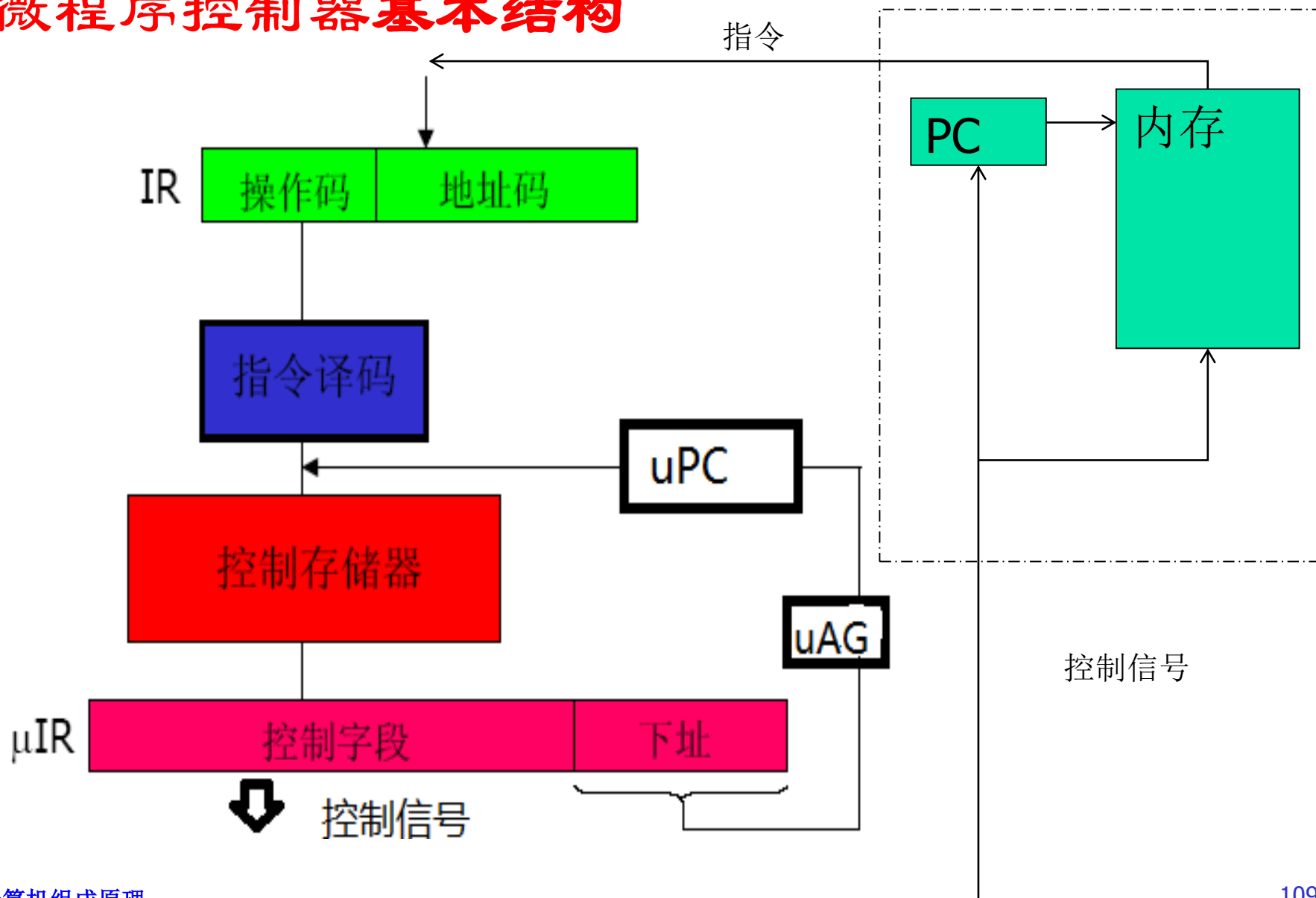
### 3、微地址形成部件 $\mu\text{AG}$

用来产生机器指令的首条微指令地址和后续地址。

### 4、微地址寄存器 $\mu\text{AR}$ （ $\mu\text{PC}$ ）

接收微地址形成部件送来的微地址。

# 微程序控制器基本结构



## 二、工作过程

微程序控制器的工作过程实质上就是在微程序控制器的控制之下，计算机执行机器指令的过程：

- 1、**机器复位后**，从控制存储器中取出第一条微指令
- 2、第一条微指令通常是跳转微指令，转移到取指令微操作
- 3、执行取值令微指令，根据PC成从主存储器中取得机器指令；
- 4、根据机器指令的操作码(译码)得到相应机器指令的微程序入口；
- 5、逐条取出微指令，完成相关微操作控制；
- 6、执行到最后一条微指令，该微指令完成后将跳转到步骤3，即取下一条机器指令的微操作。

# 微程序设计的相关技术

微程序设计需要考虑以下3个问题：

- 1、如何缩短微指令字长
- 2、如何减少微程序长度
- 3、如何提高微程序的执行速度

## (一) 微指令的编码

编码的实质是在微指令中如何组织微操作的问题  
构成：

微操作控制字段

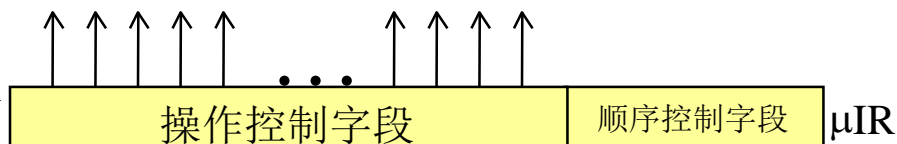
顺序控制字段

### 1、直接控制编码法

控制字段中的每一位表示一个微命令

缺点：

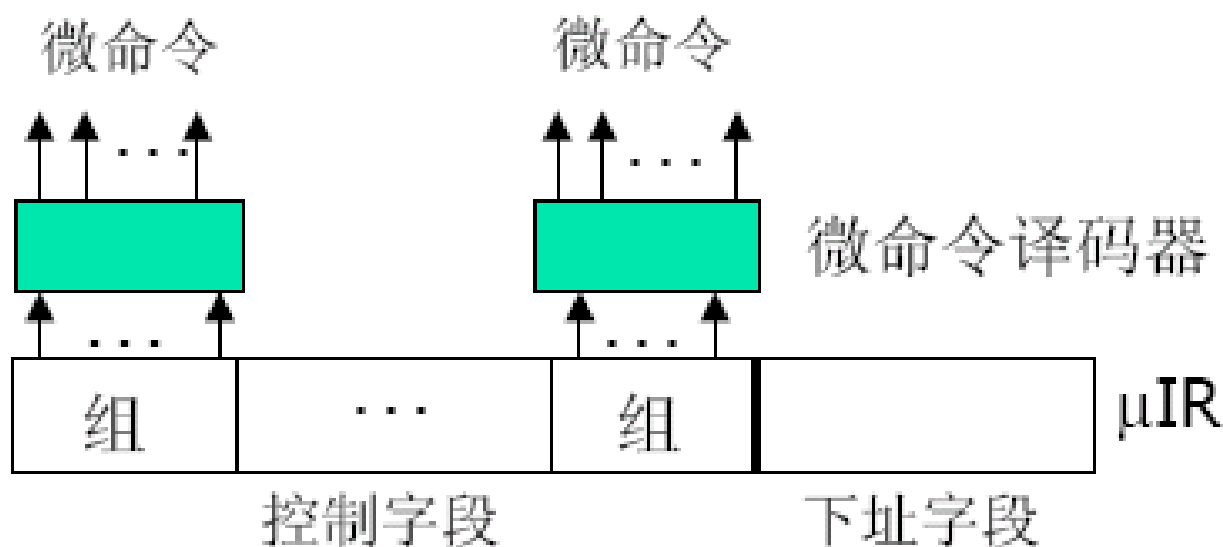
控制字段长，如三，四百位  
-控制存储器容量要大



直接表示法

## 2、字段直接编码法

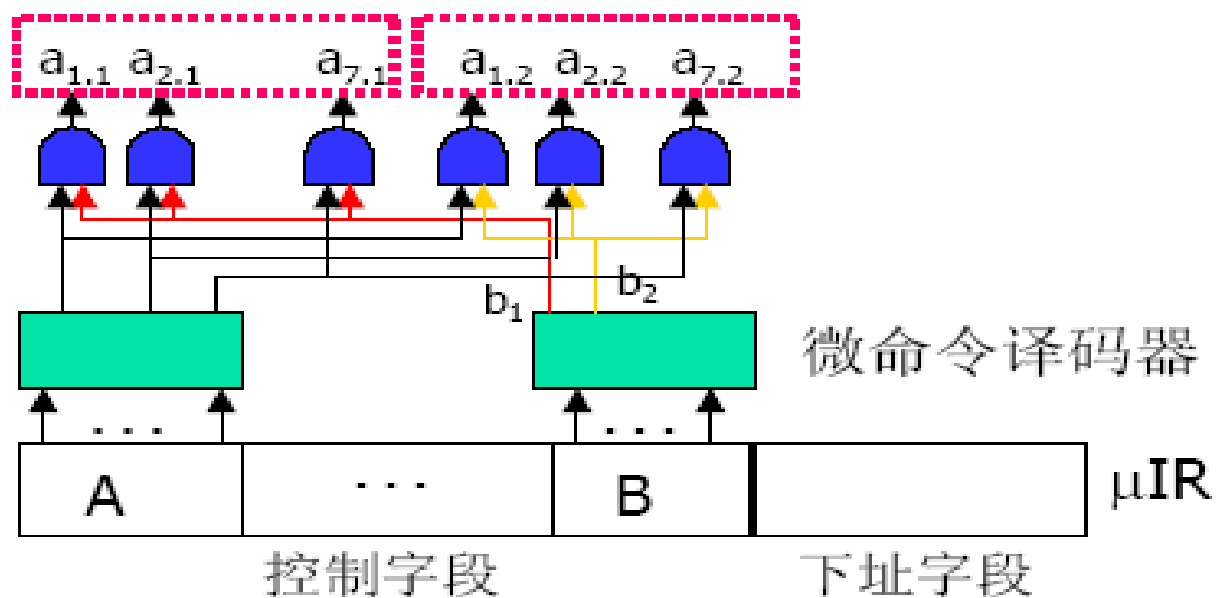
- (1) 把互斥的微命令编为一组
- (2) 对微命令进行编码，留出一个代码表示**本段不发微命令**
- (3) 增设**微命令译码器**



### 3、字段间接编码法

一个字段的某些微命令由另一个字段的某些微命令来解释

如：字段A受字段B的控制





## (二) 后继地址的生成

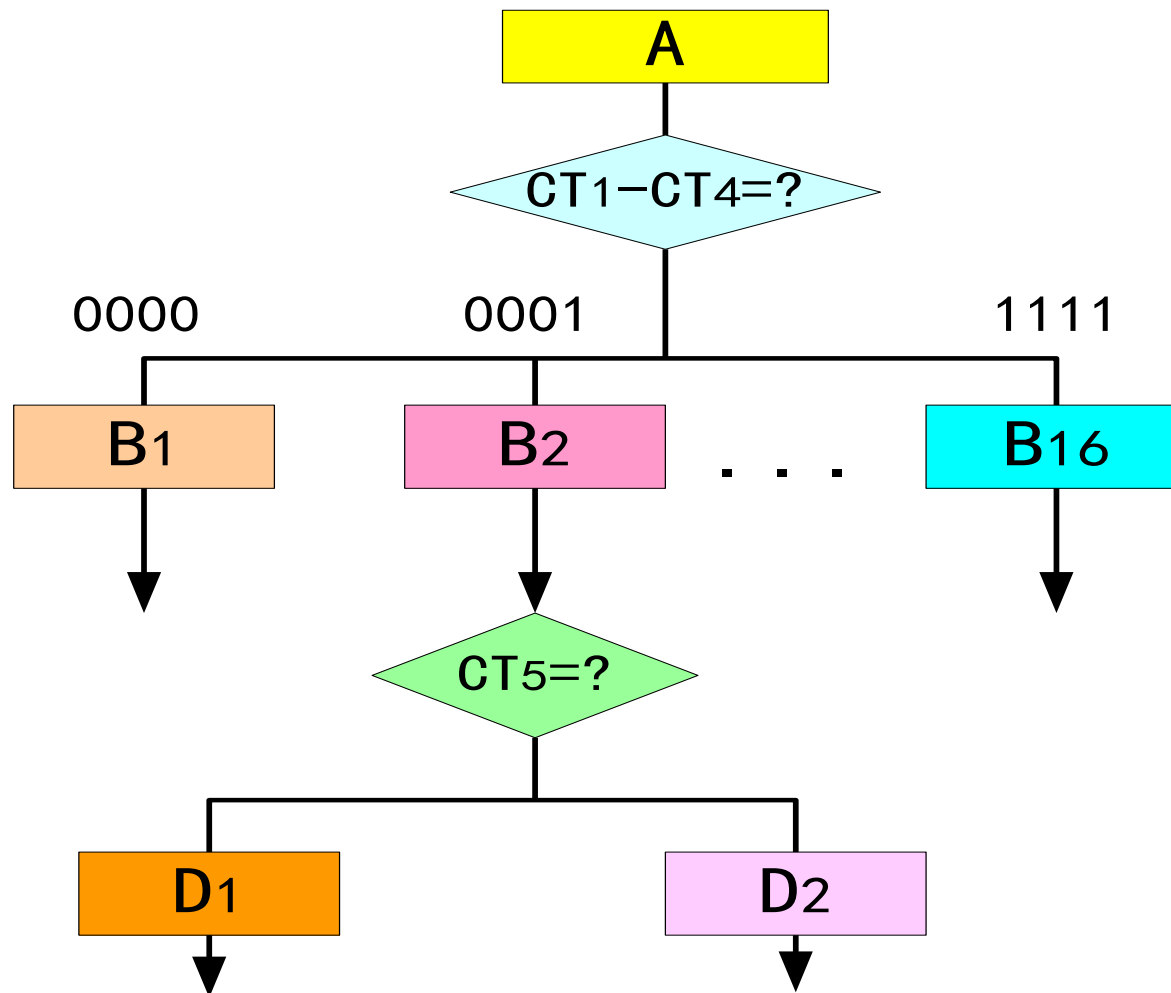
微程序的特征：存在大量的分支

因为：

- 1、转移类指令要有分支
- 2、大多数指令都有几种寻址方式
- 3、每条指令都对应一个独立的微程序

如果为每条机器指令、每种寻址方式都单独设置一段微程序，导致整个必将微程序太长，需要大量的控存容量，因此，应把重复部分作为共享的微子程序库，由各微程序调用，因而会产生大量的寻址分支流程。

# 微程序的特征



## (二) 微指令地址形成方式

### 1、指令对应入口微地址的方法

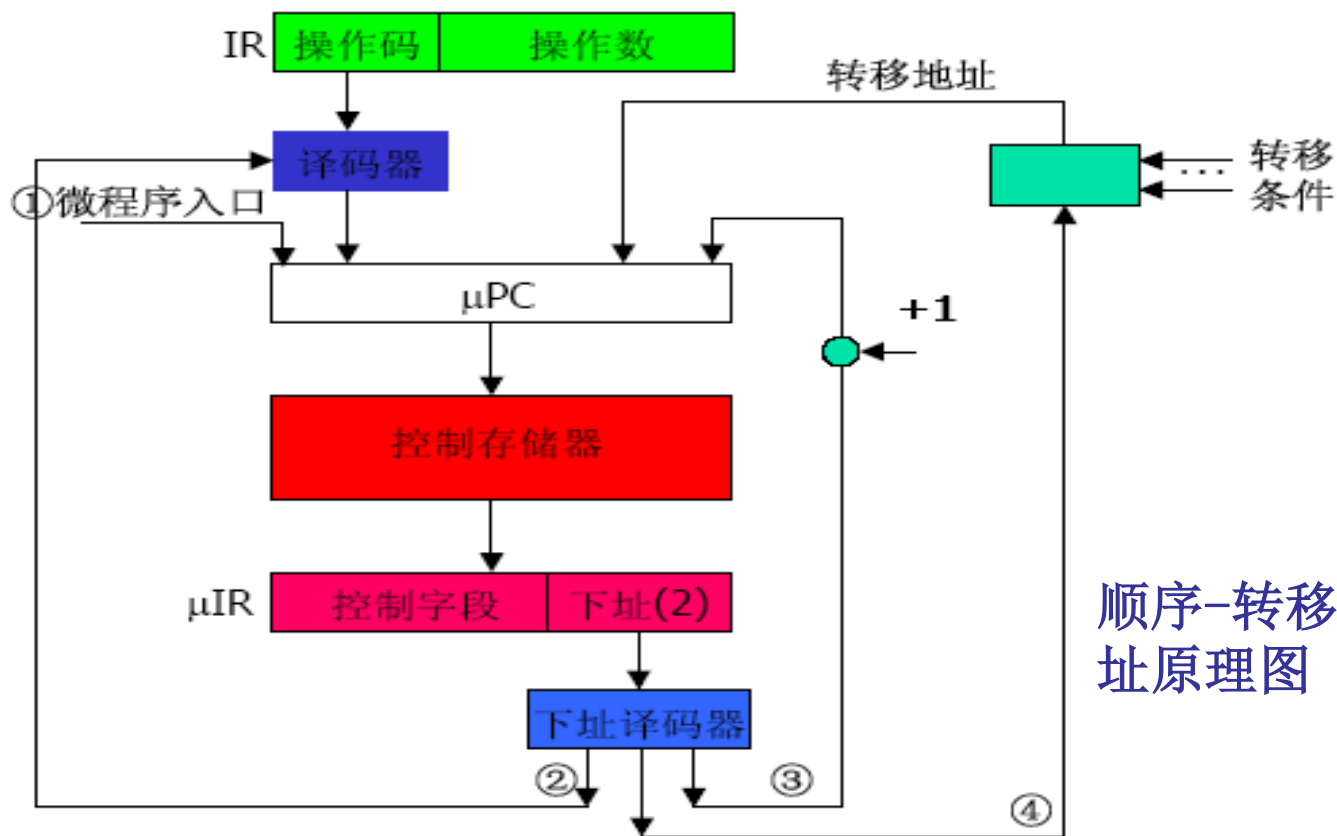
每条机器指令对应一段微程序，当执行公用的取指微程序从主存中取出机器指令后，由机器指令的操作码指出微程序的首地址。这是一种多分支情况，通常有以下几种方式：

- (1) 操作码的位数与位置固定，这时可直接使操作码与微地址码的部分位相对应。例如，若微入口地址=000C，则控制存储器第零页的一些单元被安排为各个微程序入口，再通过无条件微转移指令使这些单元与相应的后续微指令相连接。
- (2) 当每类指令的操作码位数与位置固定，而各类指令之间的操作码位数与位置不固定时，可采用分级转移的方法。先按指令类型转移到某条微指令，区分出是哪一大类，然后进一步按机器指令操作码转移，区分出是哪一种具体的机器指令。
- (3) 当操作码的位数与位置都不固定时，通常的方法是采用PLA可编程逻辑阵列实现。

## 2、后继微指令地址形成

得到微程序入口以后，就开始执行微程序，后继微地址的形成方法对微程序编制的灵活性影响很大。通常采用两种方法形成后继微地址：

### (1) 顺序-转移型微地址



顺序-转移型微地址原理图

## 顺序-转移型微地址

微程序按地址**递增顺序**一条一条地执行微指令，遇到**转移**时，由微指令给出转移微地址，使微程序按新的顺序执行。

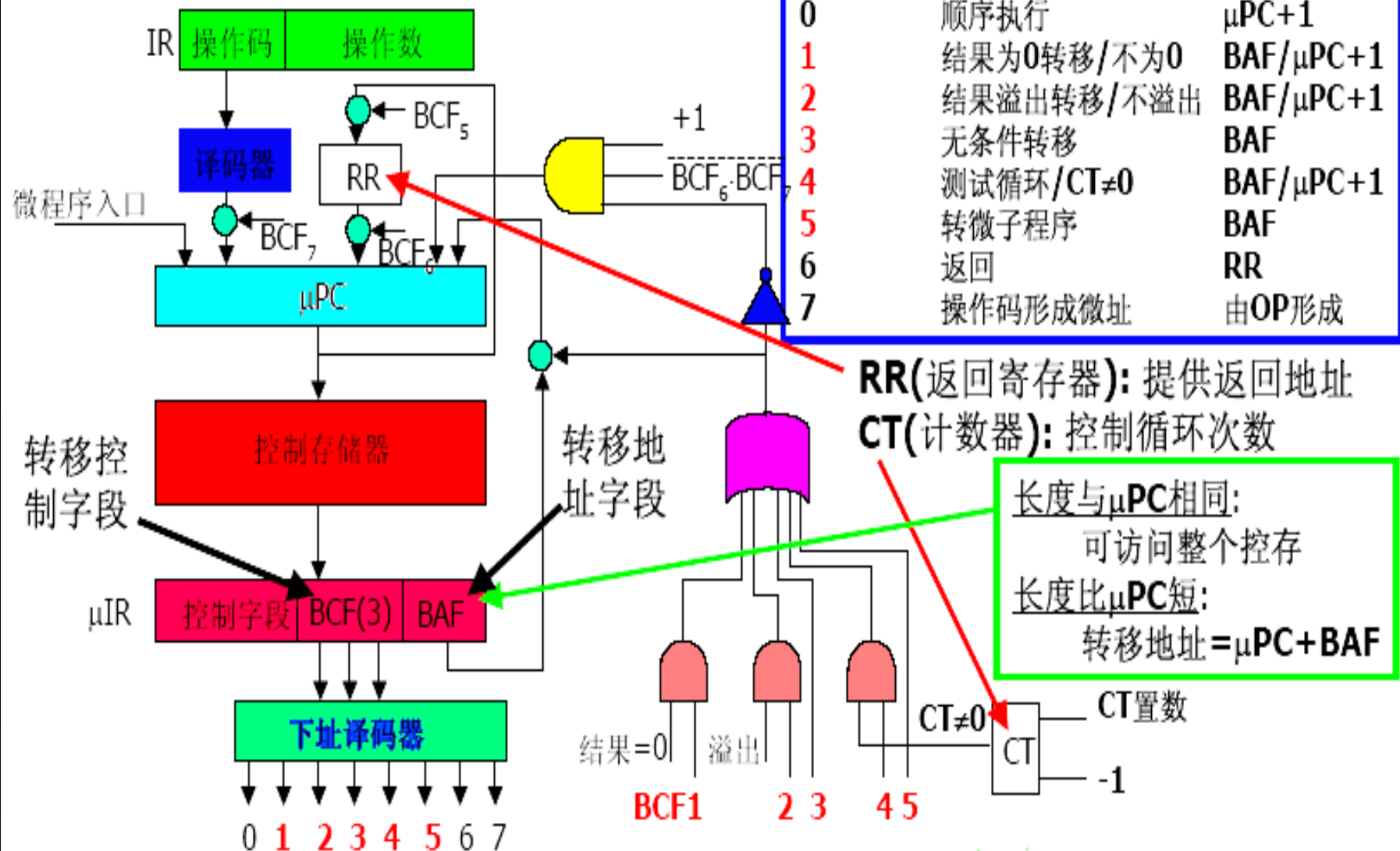
微指令格式：



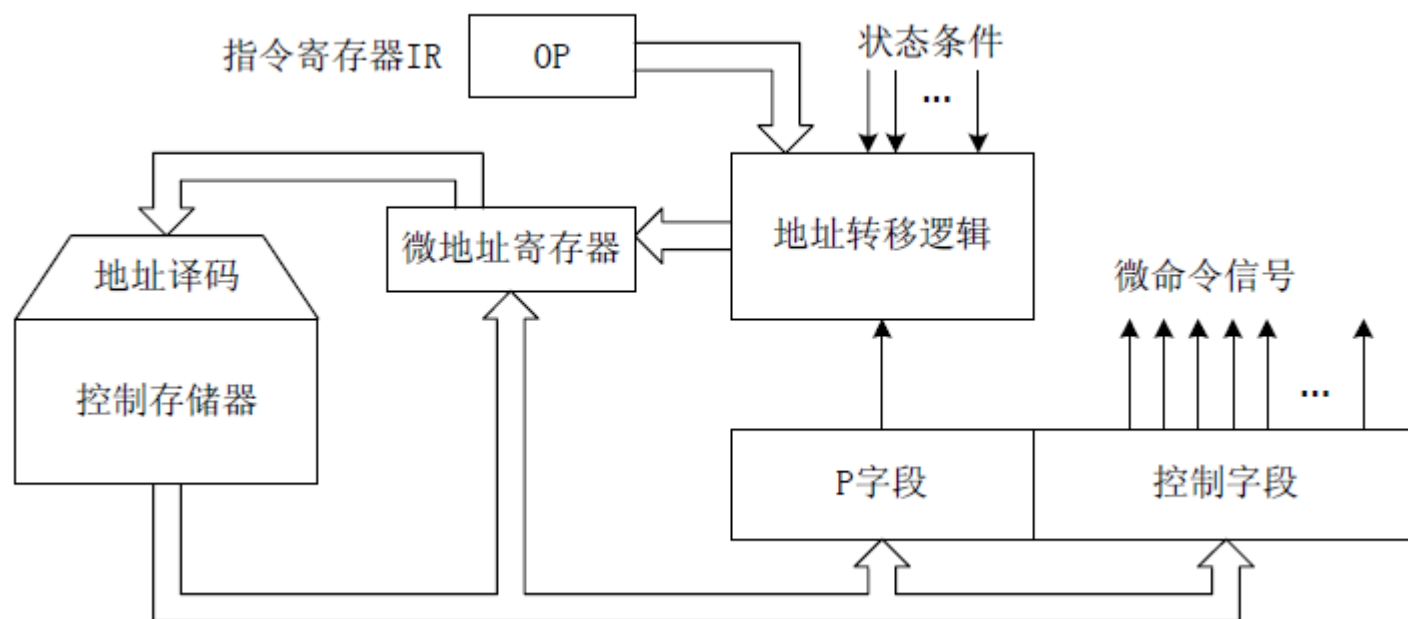
**转移地址字段BAF**：用于给出后继转移微地址

**转移控制字段BCF**：用于规定后继微地址是顺序执行还是按BAF字段的转移地址执行。

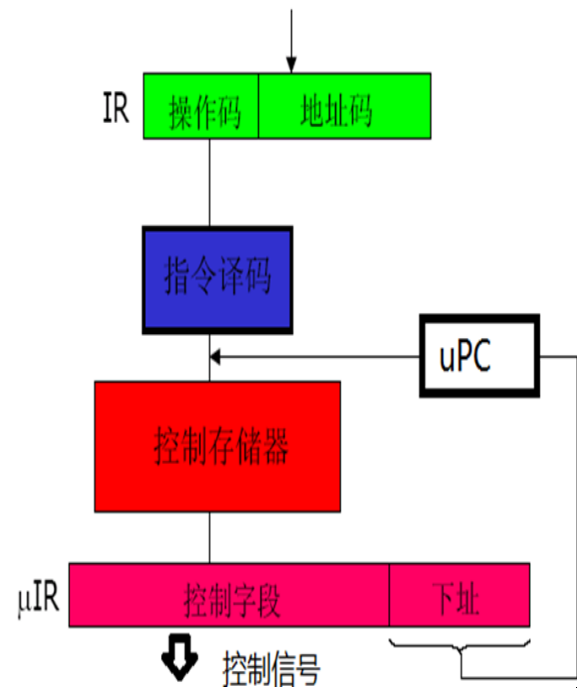
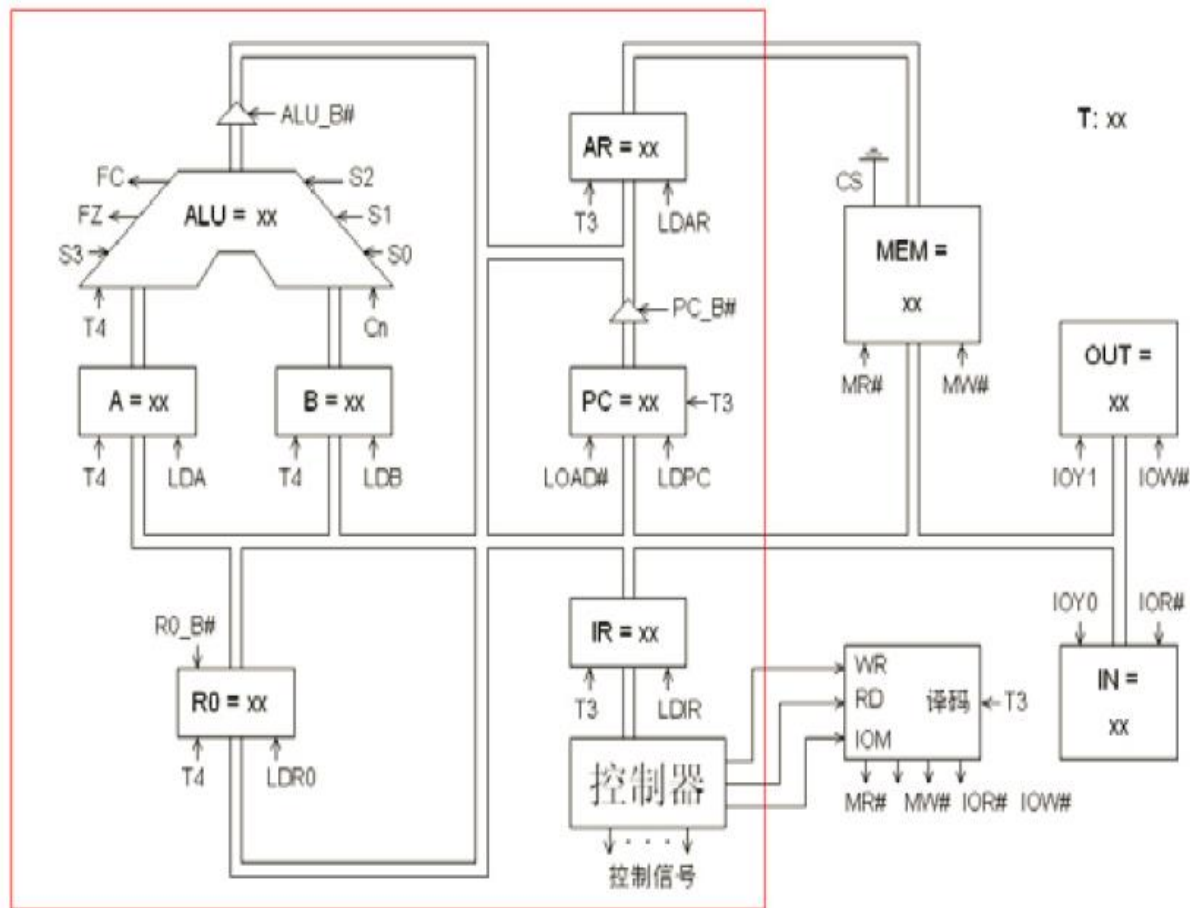
# uAG部件举例



# 实验系统：微程序控制器结构

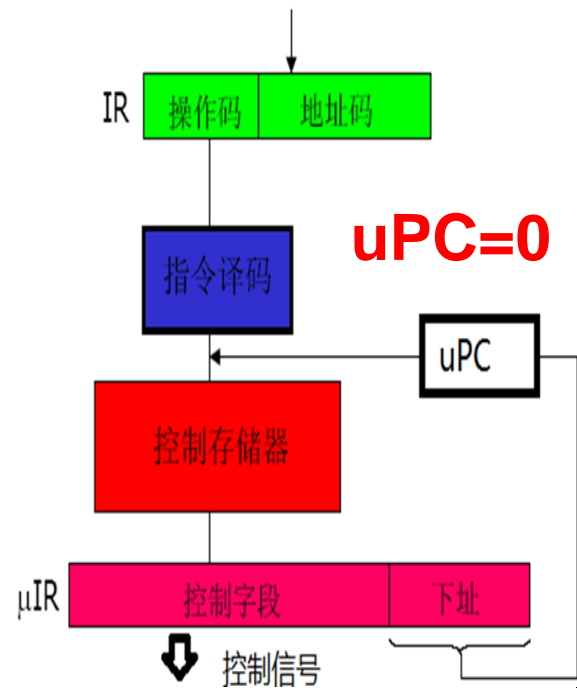
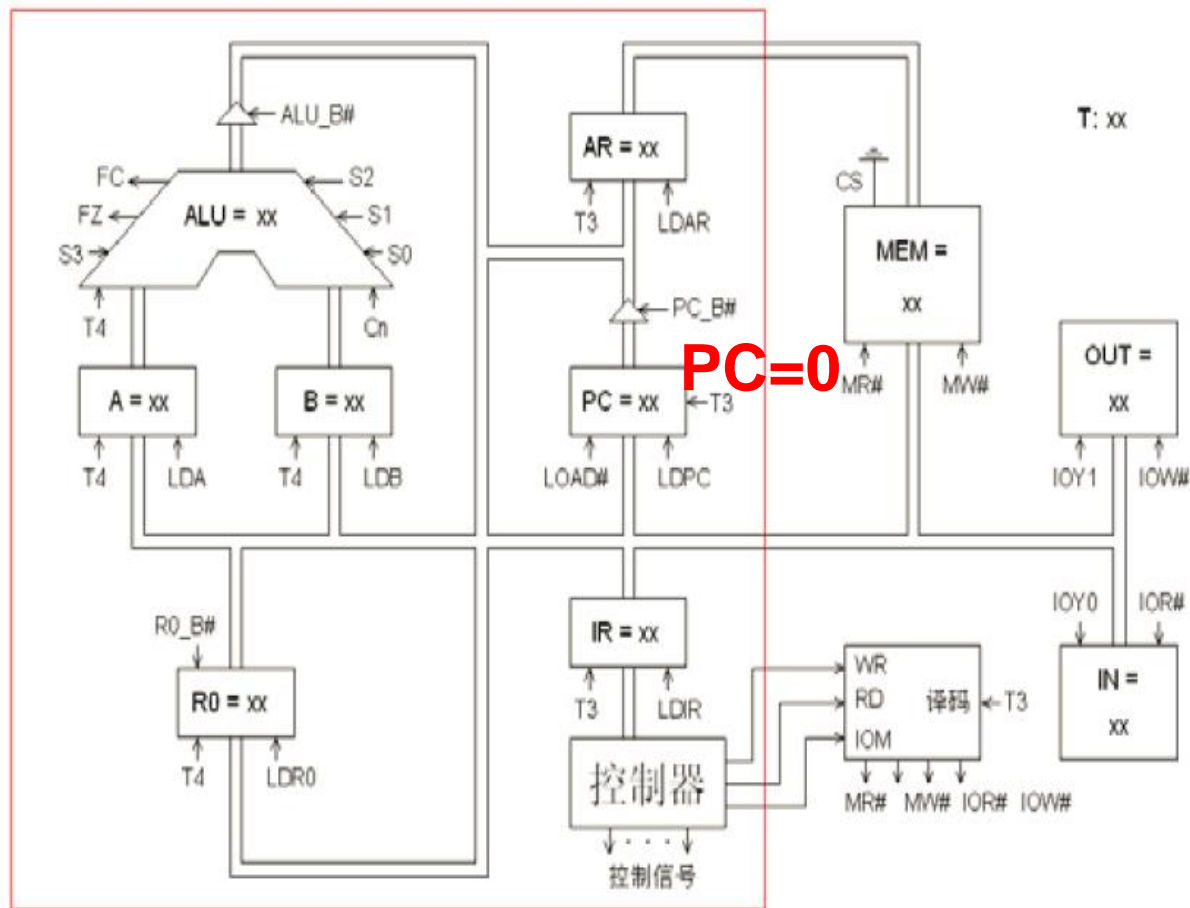


# 实验系统中主机系统的数据通路





# 系统复位后寄存器的情况



# 指令执行的过程

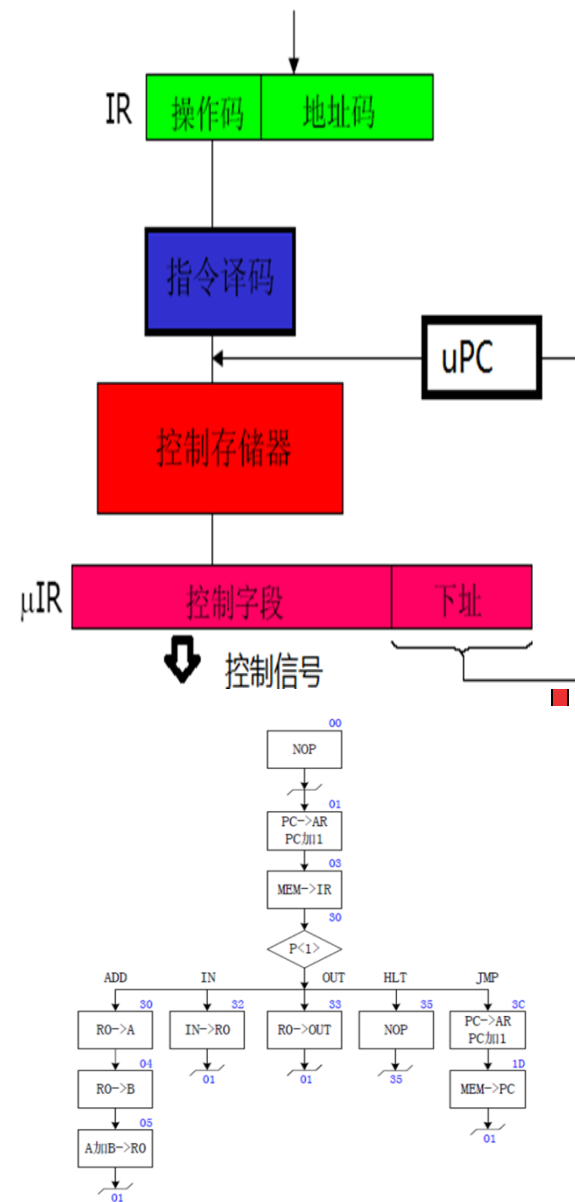
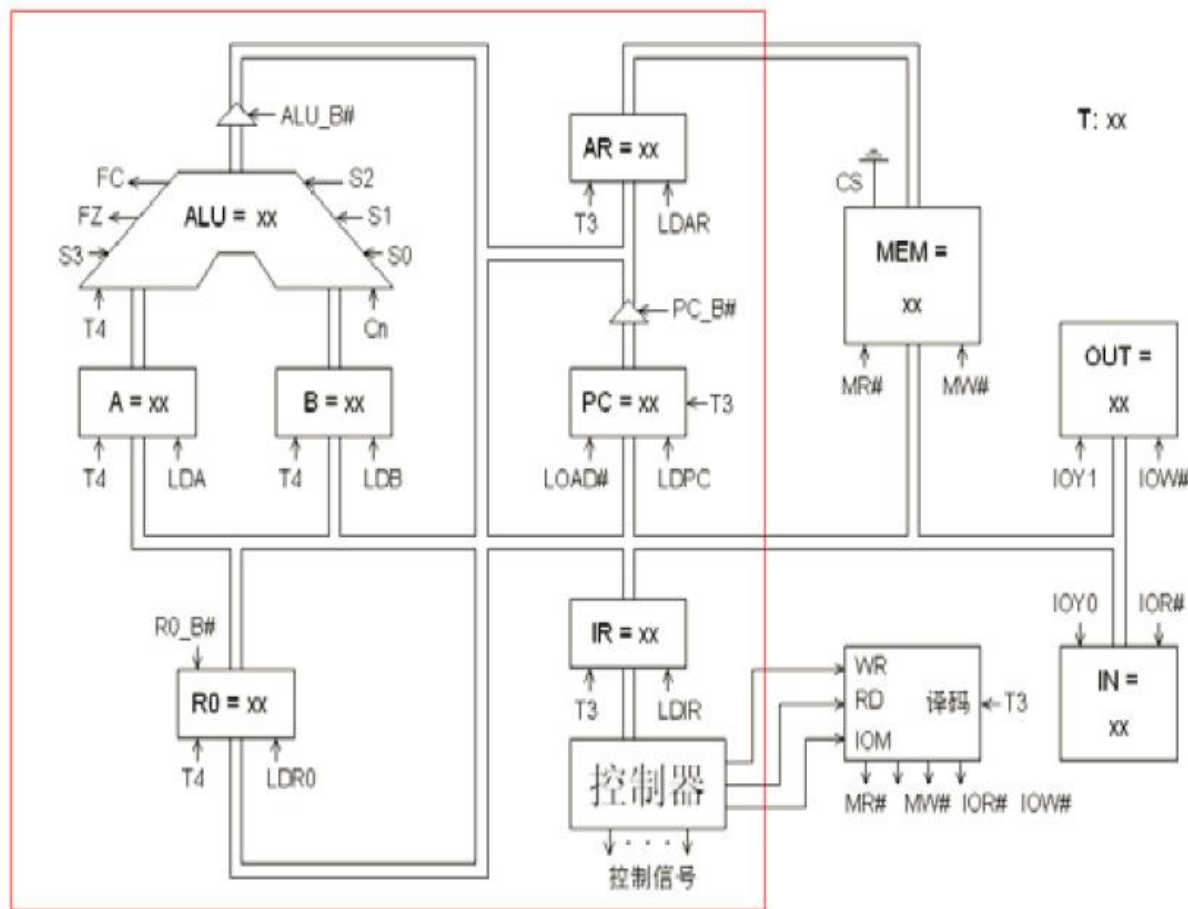


表 5-1-2 二进制微代码表

地址	十六进制	高五位	S3-S0	A 字段	B 字段	C 字段	MA5-MA0
00	00 00 01	00000	0000	000	000	000	000001
01	00 6D 43	00000	0000	110	110	101	000011
03	10 70 70	00010	0000	111	000	001	110000
04	00 24 05	00000	0000	010	010	000	000101
05	04 B2 01	00000	1001	011	001	000	000001
1D	10 51 41	00010	0000	101	000	101	000001
30	00 14 04	00000	0000	001	010	000	000100
32	18 30 01	00011	0000	011	000	000	000001
33	28 04 01	00101	0000	000	010	000	000001
35	00 00 35	00000	0000	000	000	000	110101
3C	00 6D 5D	00000	0000	110	110	101	011101

表 5-1-1 微指令格式

23	22	21	20	19	18-15	14-12	11-9	8-6	5-0
M23	M22	WR	RD	IOM	S3-S0	A 字段	B 字段	C 字段	MA5-MA0

A 字段

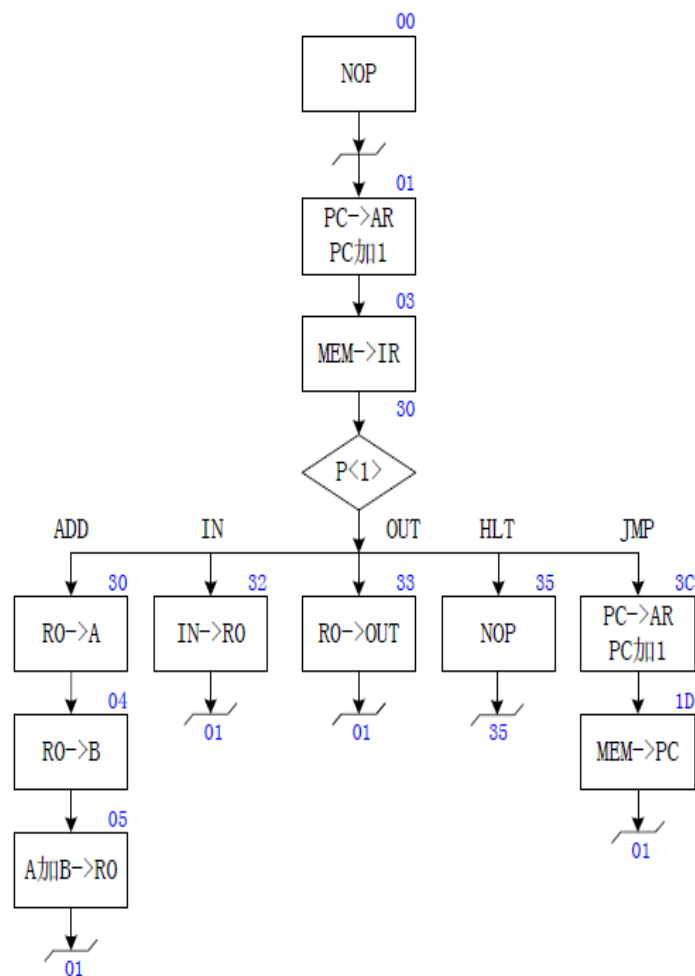
14	13	12	选择
0	0	0	NOP
0	0	1	LDA
0	1	0	LDB
0	1	1	LDR0
1	0	0	保留
1	0	1	LOAD
1	1	0	LDAR
1	1	1	LDIR

B 字段

11	10	9	选择
0	0	0	NOP
0	0	1	ALU_B
0	1	0	RO_B
0	1	1	保留
1	0	0	保留
1	0	1	保留
1	1	0	PC_B
1	1	1	保留

C 字段

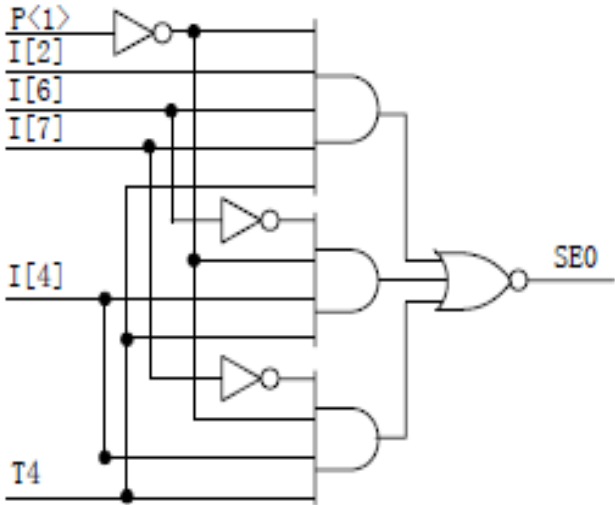
8	7	6	选择
0	0	0	NOP
0	0	1	P<1>
0	1	0	保留
0	1	1	保留
1	0	0	保留
1	0	1	LDPC
1	1	0	保留
1	1	1	保留



# 微程序器中后继地址的生成

表 5-1-1 微指令格式

23	22	21	20	19	18-15	14-12	11-9	8-6	5-0
M23	M22	WR	RD	IOM	S3-S0	A字段	B字段	C字段	MA5-MA0



P<1>  
I[3]  
I[6]  
I[7]

A字段			
14	13	12	选择
0	0	0	NOP
0	0	1	LDA
0	1	0	LDB
0	1	1	LDRO
1	0	0	保留
1	0	1	LOAD
1	1	0	LDAR
1	1	1	LDIR

B字段			
11	10	9	选择
0	0	0	NOP
0	0	1	ALU_B
0	1	0	RO_B
0	1	1	保留
1	0	0	保留
1	0	1	保留
1	1	0	PC_B
1	1	1	保留

C字段			
8	7	6	选择
0	0	0	NOP
0	0	1	P<1>
0	1	0	保留
0	1	1	保留
1	0	0	保留
1	0	1	LDPC
1	1	0	保留
1	1	1	保留

I[5]

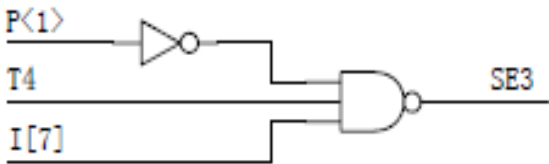
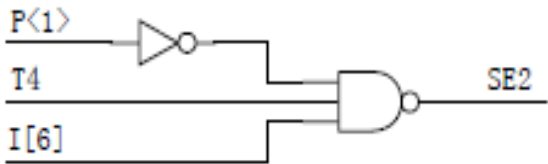
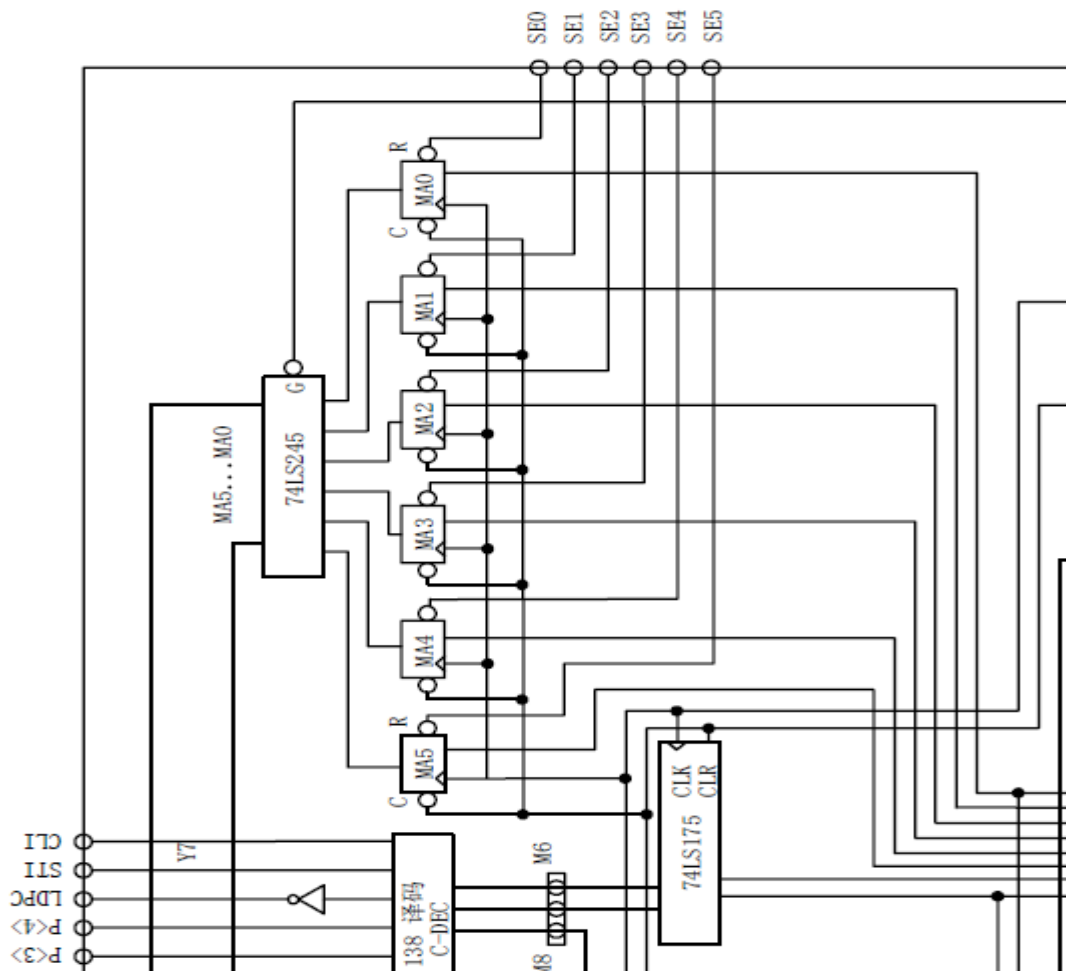


图 3-2-3 指令译码原理图

# 微程序器中后继地址的生成



# 微指令格式

## (1) 水平型微指令——并行性。

微指令中的微操作有高度并行性，灵活性强；执行指令的时间较短；微指令字比较长，但微程序比较短；硬件密切相关，微程序设计比较困难。微指令编码简单

## (2) 垂直型微指令——有微操作码、部件号。

微指令中的并行微操作能力有限，每条微指令只表示一个微操作；需要对微操作码和部件号译码；微指令字短，微程序长。微指令编码比较复杂。

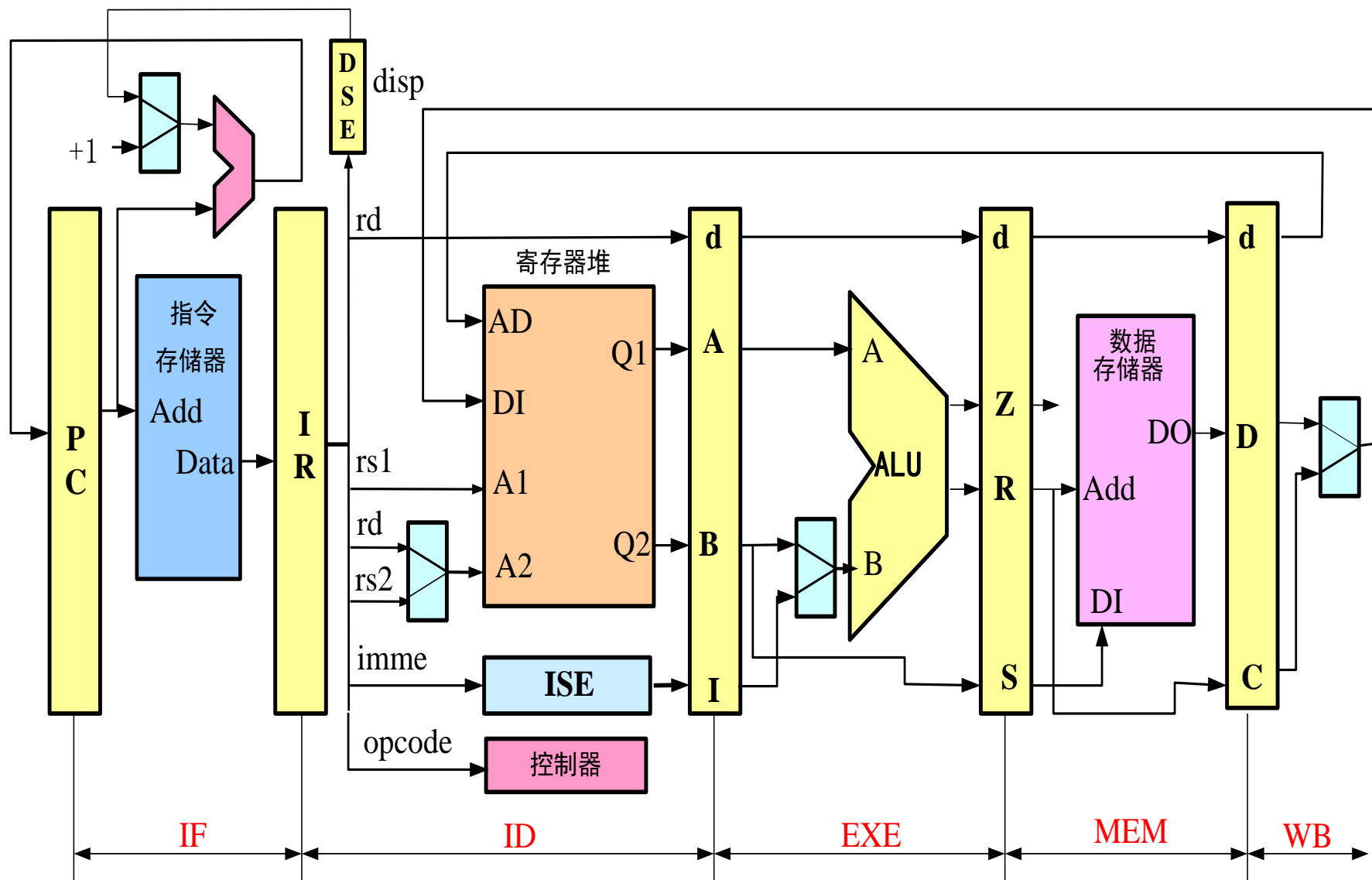
可以用助记符表示，例：

MOV MDR MAR

# Performance Issues

- Longest delay determines clock period
  - Critical path: **load** instruction
  - Instruction memory → register file → ALU → data memory → register file
- Useless to try implementation techniques to reduce delay for different instructions
  - Load instruction still the bottleneck and will still determine clock period
- Violates design principle
  - Making the common case fast!
- We will improve performance by **pipelining**

# DataPath of Pipeline ?





# Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Read Appendix E: A Survey of RISC Architectures for Desktop, Server, and Embedded Computers

# Assignment 4

---

- Homework assignment  
4.1, 4.2 ,4.7,4.9
- To be submitted in the next week class

# 实验安排

实验的内容为

实验(一) 1.1基本运算器实验

实验(二) 2.1静态随机存储器实验

实验(三) 3.2微程序控制器实验

实验(四) 5.1CPU与简单模型机设计实

验