中国科学技术大学
University of Science and Technology of China

3. MIPS处理器设计

王 超

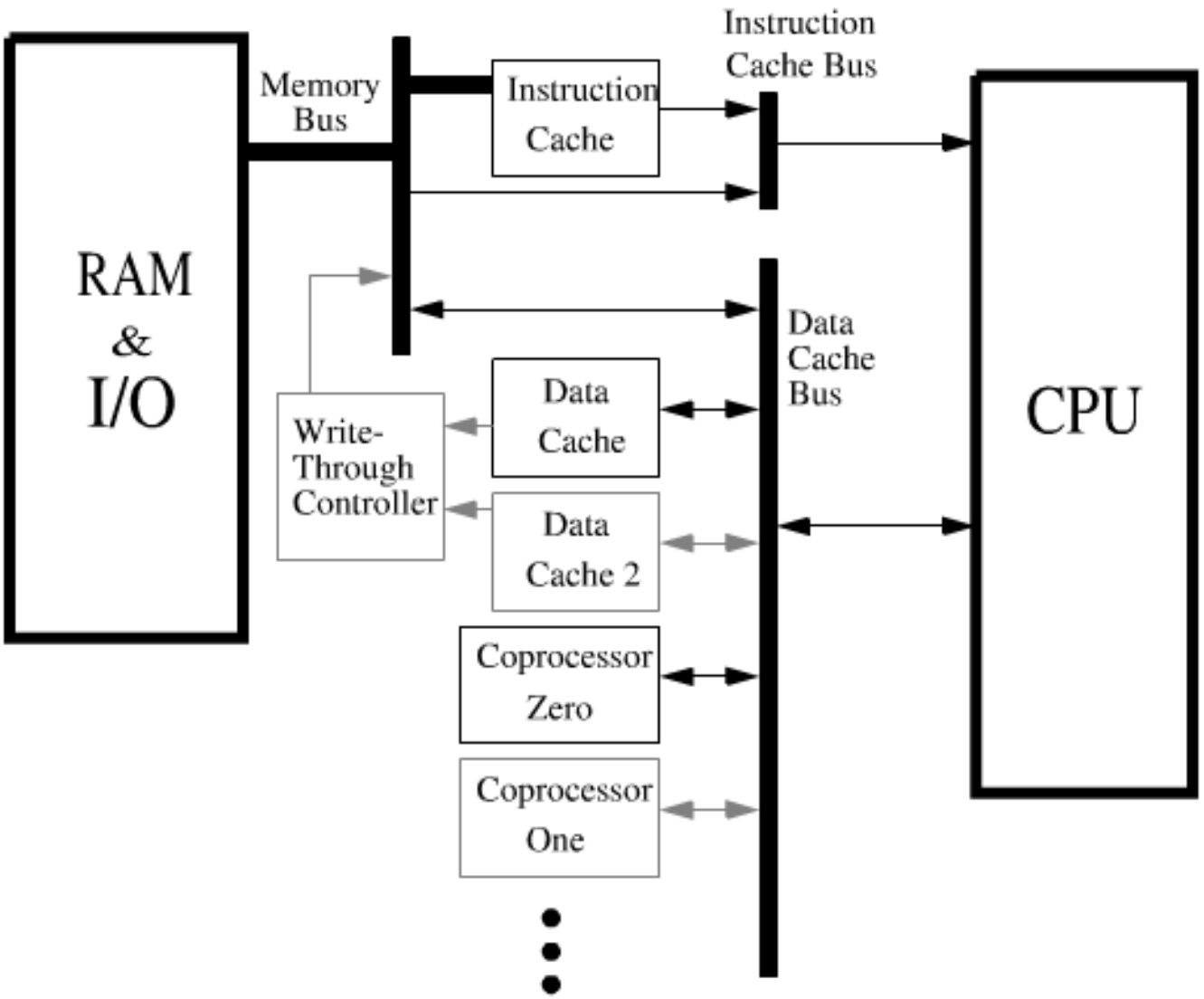
中国科学技术大学计算机学院
嵌入式系统实验室

2019年春

主要内容

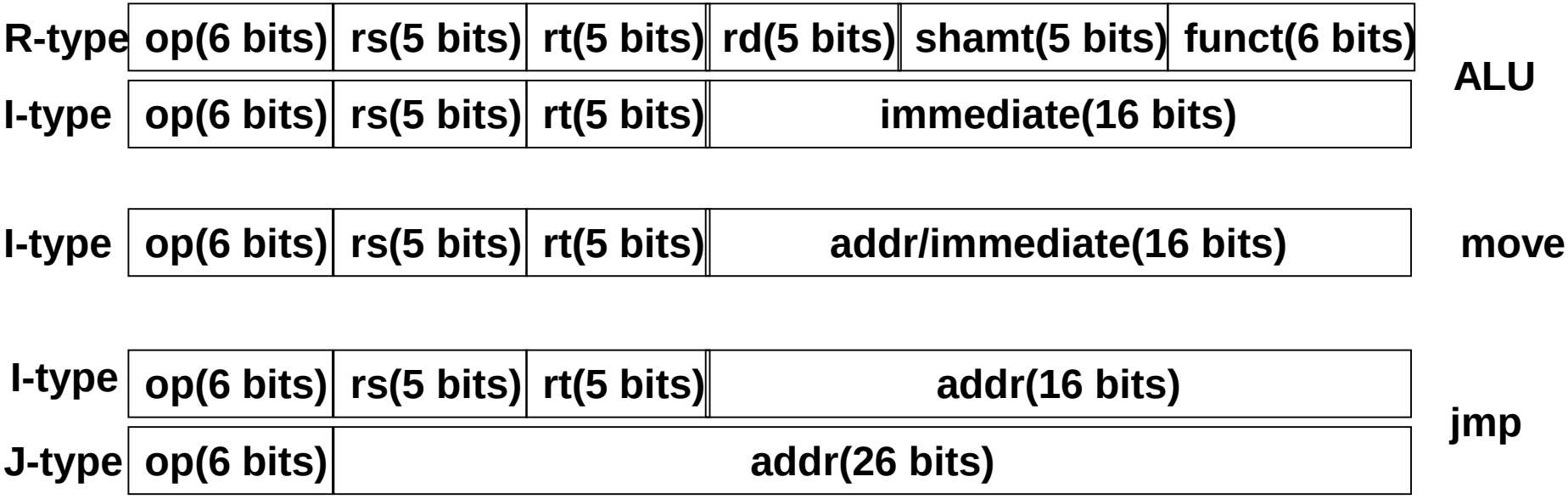
- MIPS处理器结构
- MIPS指令集结构
- 单周期CPU设计
  - ✓功能部件
  - ✓数据通路
- 多周期MIPS处理器设计

MIPS R3000


RAM
&
I/O
Memory
Bus
Instruction
Cache
Instruction
Cache Bus
Write-
Through
Controller
Data
Cache
Data
Cache 2
Coprocessor
Zero
Coprocessor
One
Data
Cache
Bus
CPU

MIPS指令字格式

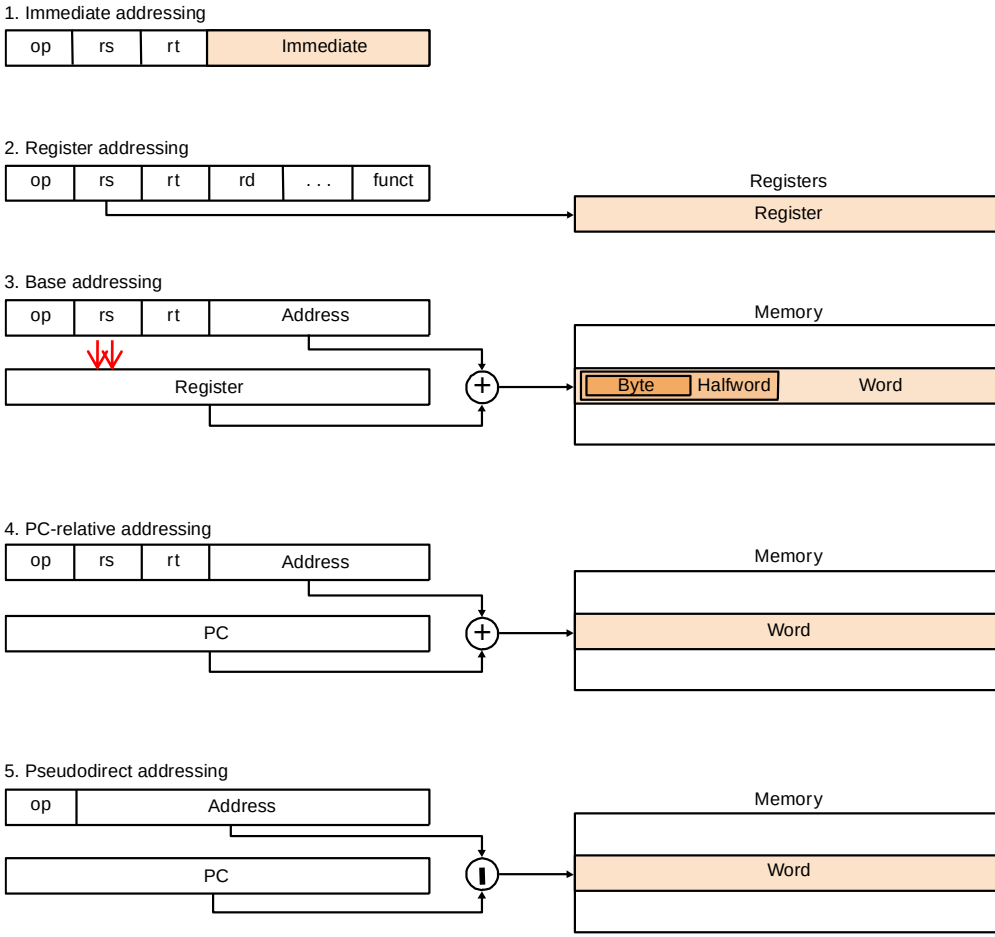
- 100余条指令（P&H 33条），32个通用寄存器
- 指令格式：定长32位
  - ✓R-type：arithmetic instruction
  - ✓I-type：data transfer, arithmetic instruction（如addi）
  - ✓J-type：branch instruction(conditional & unconditional)



新版BEQ指令 J类->I类

MIPS寻址模式

- 立即寻址： I-type
- 寄存器寻址：R-type
- 基址寻址：I-type
  - ✓ rs = BaseReg
- 相对寻址：J(I)-type
- 伪直接寻址：J-type
  - ✓ pseudodirect addressing
  - ✓ 26位形式地址左移2位，与PC的高4位拼接



寄存器使用约定

REGISTER	NAME	USAGE
$0	$zero	常量0(constant value 0)
$1	$at	保留给汇编器(Reserved for assembler)
$2-$3	$v0-$v1	函数调用返回值(values for results and expression evaluation)
$4-$7	$a0-$a3	函数调用参数(arguments)
$8-$15	$t0-$t7	暂时的(或随便用的)
$16-$23	$s0-$s7	保存的(或如果用，需要SAVE/RESTORE的)(saved)
$24-$25	$t8-$t9	暂时的(或随便用的)
$28	$gp	全局指针(Global Pointer)
$29	$sp	堆栈指针(Stack Pointer)
$30	$fp	帧指针(Frame Pointer)
$31	$ra	返回地址(return address)

寄存器使用约定

- $0:即$zero,该寄存器总是返回零，为0这个有用常数提供了一个简洁的编码形式。（初始化、BEQ和0比较）
  move $t0,$t1　　实际为　　add $t0,$0,$t1
  使用伪指令可以简化任务，汇编程序提供了比硬件更丰富的指令集。
- $1:即$at，该寄存器为汇编保留，由于I型指令的立即数字段只有16位，在加载大常数时，编译器或汇编程序需要把大常数拆开，然后重新组合到寄存器里。比如加载一个32位立即数需要 lui（装入高位立即数）和addi两条指令。像MIPS程序拆散和重装大常数由汇编程序来完成，汇编程序必需一个临时寄存器来重组大常数，这也是为汇编 保留$at的原因之一。
- $2..$3:($v0-$v1)用于子程序的非浮点结果或返回值，对于子程序如何传递参数及如何返回，MIPS范围有一套约定，堆栈中少数几个位置处的内容装入CPU寄存器，其相应内存位置保留未做定义，当这两个寄存器不够存放返回值时，编译器通过内存来完成。
- $4..$7:($a0-$a3)用来传递前四个参数给子程序，**不够的用堆栈**。a0-a3和v0-v1以及ra一起来支持子程序 / 过程 调用，分别用以传递参数，返回结果和存放返回地址。当需要使用更多的寄存器时，就需要堆栈（stack)了,MIPS编译器总是为参数在堆栈中留有空间以防有参数需要存储。
- $8..$15:($t0-$t7)临时寄存器，子程序可以使用它们而不用保留。
- $16..$23:($s0-$s7)保存寄存器，在过程调用过程中需要保存的（被调用者保存和恢复，还包括$fp和$ra），MIPS提供了临时寄存器和保存寄存器，这样就减少了**寄存器溢出**（**spilling**,即将不常用的变量放到存储器的过程),编译器在编译一个叶（leaf)过程（不调用其它过程的过程）的时候，总是在临时寄存器分配完了才使用需要保存的寄存器。
- $24..$25:($t8-$t9)同($t0-$t7)
- $26..$27:($k0,$k1)为操作系统 / 异常处理保留，至少要预留一个。 异常（或中断）是一种不需要在程序中显式调用的过程。MIPS有个叫异常程序计数器（exception program counter,EPC)的寄存器，属于CP0寄存器，用于保存造成异常的那条指令的地址。查看控制寄存器的唯一方法是把它复制到通用寄存器里，指令mfc0 (move from system control)可以将EPC中的地址复制到某个通用寄存器中，通过跳转语句（jr)，程序可以返回到造成异常的那条指令处继续执行。MIPS程序员都必须保留两个寄存器$k0和$k1，供操作系统使用。发生异常时，这两个寄存器的值不会被恢复，编译器也不使用k0和k1,异常处理函数可以将返回地址放到这两个中的任何一个，然后使用jr跳转到造成异常的指令处继续执行。
- $28:($gp)为了简化静态数据的访问，MIPS软件保留了一个寄存器：全局指针gp(global pointer,$gp)，全局指针**指向静态数据区**中的运行时决定的地址，在存取位于gp值上下32KB范围内的数据时，只需要一条以gp为基指针的指令即可。在编译时，数据须在以gp为基指针的64KB范围内。
- $29:($sp)MIPS硬件并不直接支持堆栈，你可以把它用于别的目的，但为了使用别人的程序或让别人使用你的程序， 还是要遵守这个约定的，但这和硬件没有关系。
- $30:($fp)GNU MIPS C编译器使用了帧指针(frame pointer),而SGI的C编译器没有使用，而把这个寄存器当作保存寄存器使用（$s8),这节省了调用和返回开销，但增加了代码生成的复杂性。
- $31:($ra)存放返回地址，MIPS有个jal(jump-and-link,跳转并 链接)指令，在跳转到某个地址时，把下一条指令的地址放到$ra中。用于支持子程序，例如调用程序把参数放到$a0~$a3,然后jal X跳到X过程，被调过程完成后把结果放到$v0,$v1,然后使用jr $ra返回。

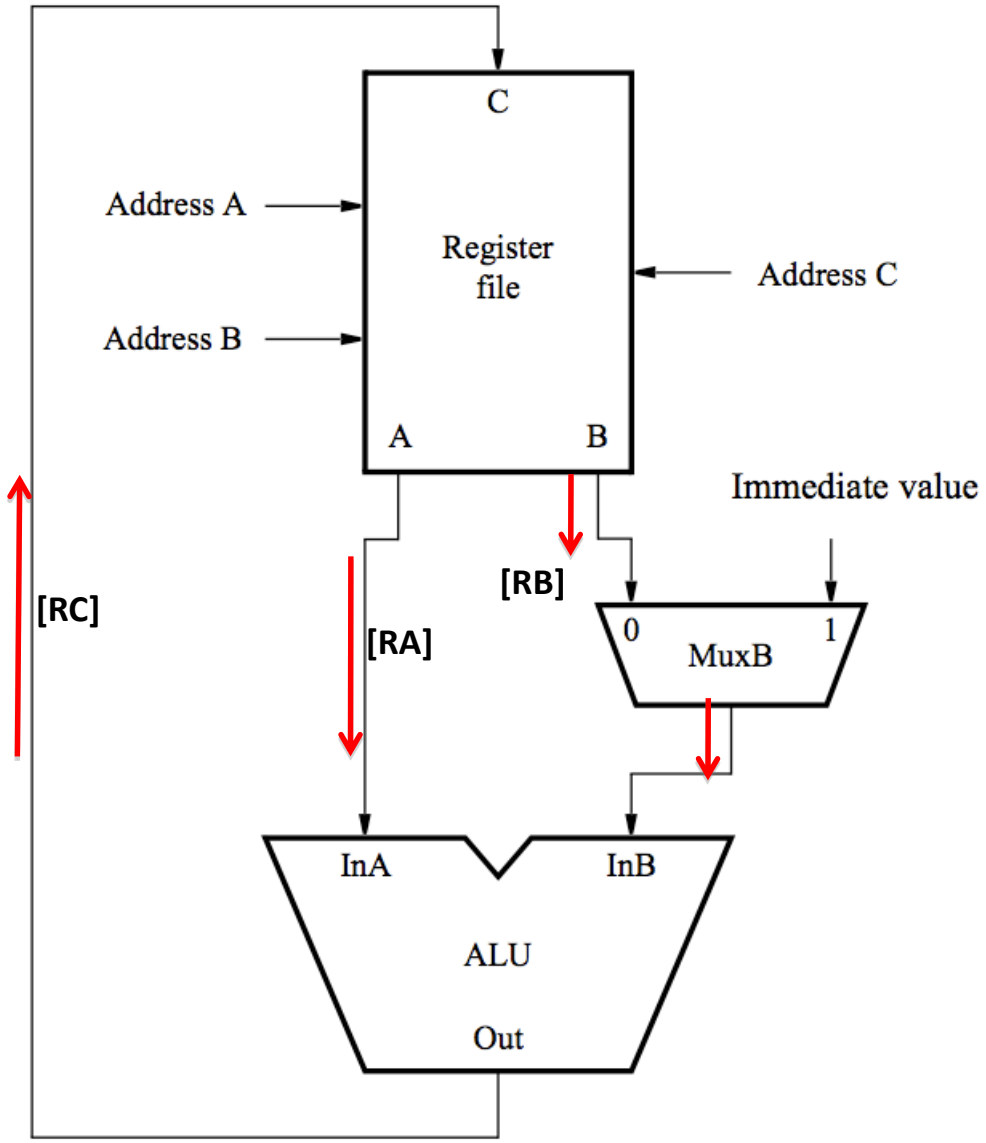
MIPS指令数据通路Overview

- 实现不同指令的多数工作都是相同的，与指令类型无关
  - ✓ 取指：将PC送往MEM
  - ✓ 取数：根据指令字中的地址域读寄存器
- 执行操作各个指令不同，但同类指令非常类似
  - ✓ 算术逻辑指令
    - R类ALU指令 **add $t1, $t2, $t3**
    - I类ALU指令 **addi $s3,$s3,4**
  - ✓ 不同类型指令也有相同之处，如都要使用ALU
    - 访存指令使用ALU计算地址 **lw $s1,100($s2)**
    - 算逻指令使用ALU完成计算 **add $t1, $t2, $t3**
    - 分支指令使用ALU进行条件比较 **beq $t1, $t2, name**
  - ✓ 其后，各个指令的工作就不同了
    - 访存指令对存储器进行读写 **lw $s1,100($s2)**
    - 算逻指令将ALU结果写回寄存器 **add $t1, $t2, $t3**
    - 分支指令将基于比较结果修改下一条指令的地址 **beq $t1, $t2, name**

寄存器计算指令的数据流

- 源操作数和目的操作数都在寄存器中

add $t1, $t2, $t3; $t2+$t3->$t1

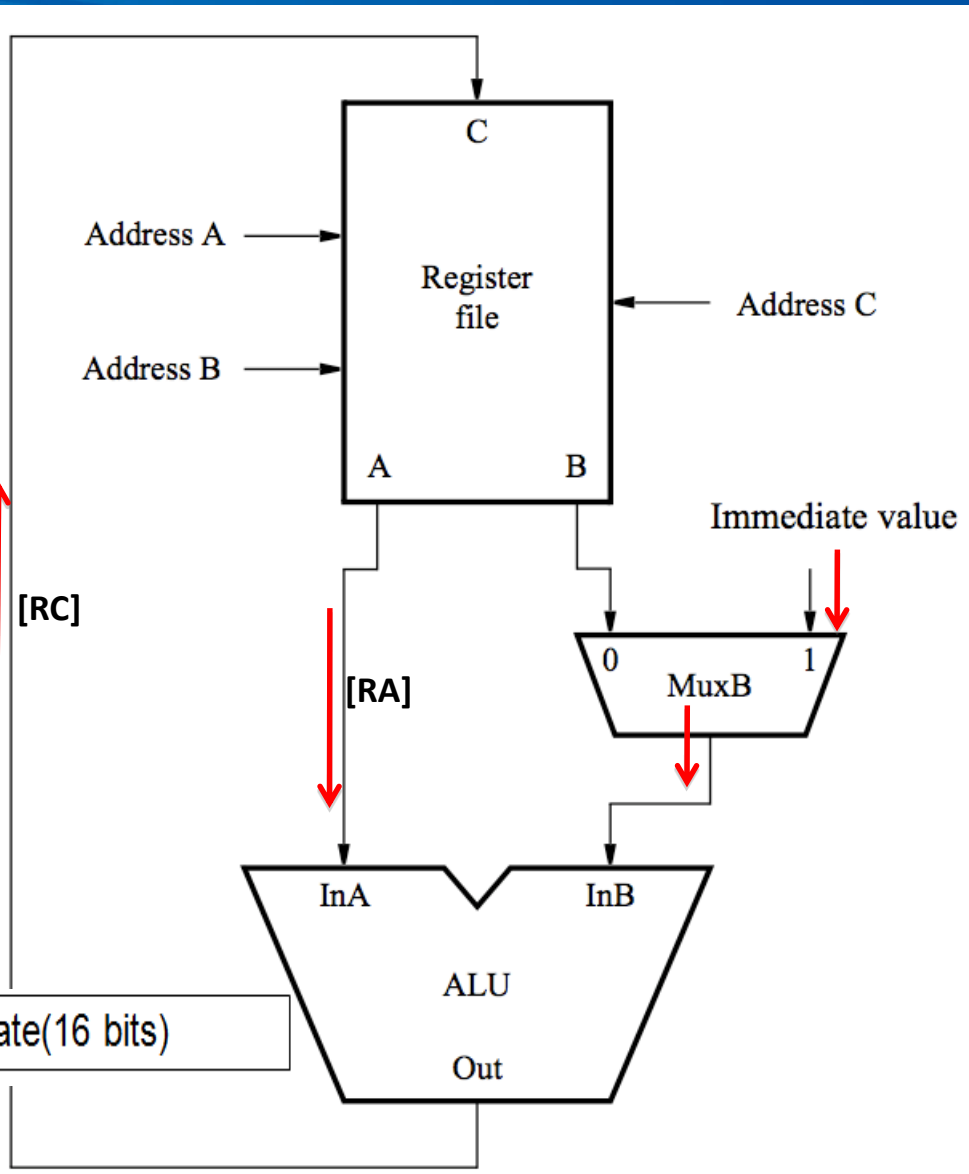


R-type

op(6 bits)	rs(5 bits)	rt(5 bits)	rd(5 bits)	shamt(5 bits)	funct(6 bits)

立即数计算指令的数据流

- 有一个源寄存器在立即数寄存器中



addi $s3,$s3,4；$s3 = $s3 + 4

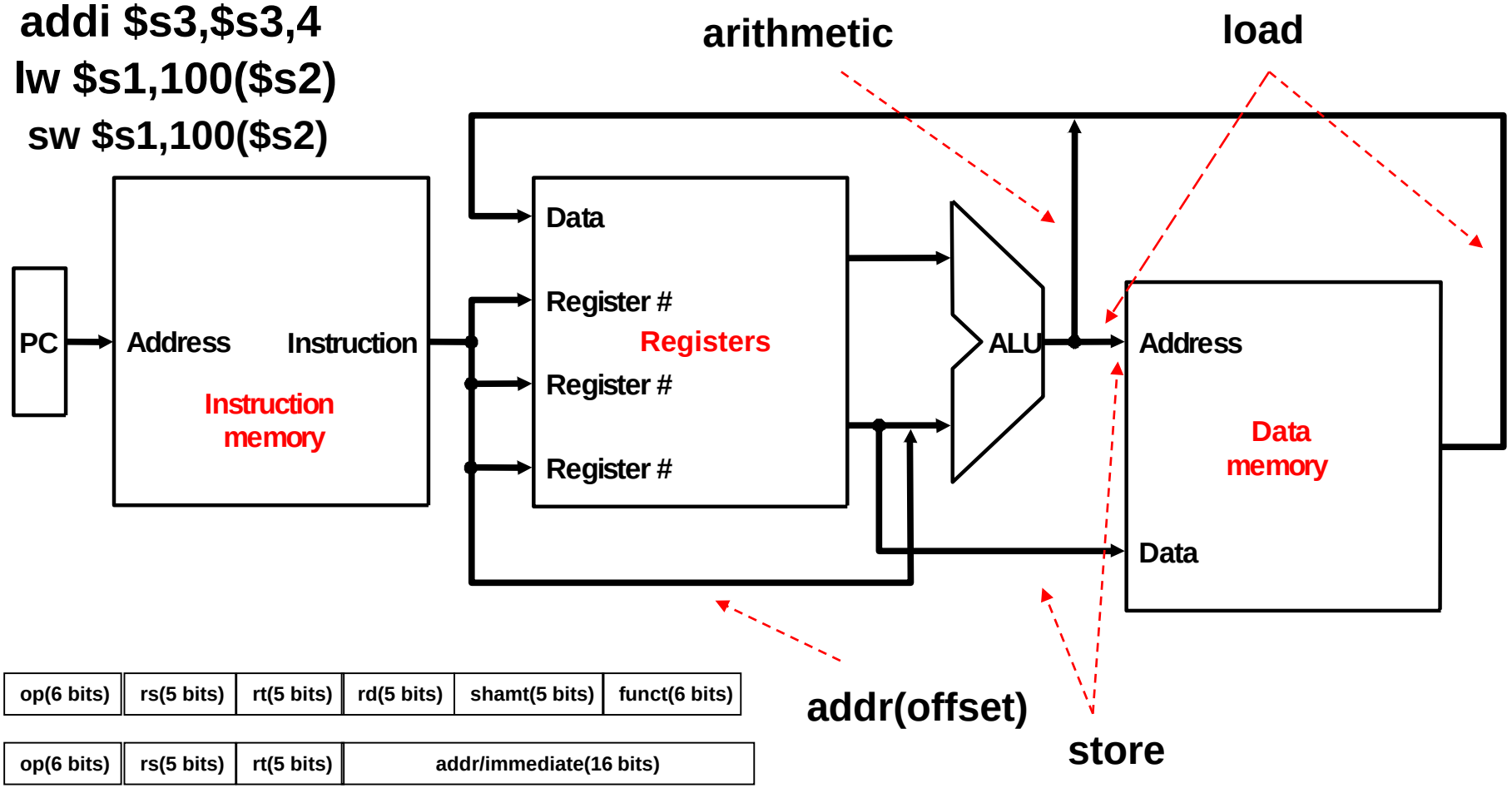
type	op(6 bits)	rs(5 bits)	rt(5 bits)	immediate(16 bits)

MIPS指令数据通路Overview

- 实现不同指令的多数工作都是相同的，与指令类型无关
  - ✓取指：将PC送往MEM
  - ✓取数（译码）：根据指令字中的地址域读寄存器
- 执行操作各个指令不同，但同类指令非常类似
  - ✓算术逻辑指令
    - R类ALU指令 **add $t1, $t2, $t3**
    - I类ALU指令 **addi $s3,$s3,4**
  - ✓不同类型指令也有相同之处，如都要使用ALU
    - 访存指令使用ALU计算地址 **lw $s1,100($s2)**
    - 算逻指令使用ALU完成计算 **add $t1, $t2, $t3**
    - 分支指令使用ALU进行条件比较 **beq $t1, $t2, name**
- 其后，各个指令的工作就不同了
    - 访存指令对存储器进行读写 **lw $s1,100($s2)**
    - 算逻指令将ALU结果写回寄存器 **add $t1, $t2, $t3**
    - 分支指令将基于比较结果修改下一条指令的地址 **beq $t1, $t2, name**

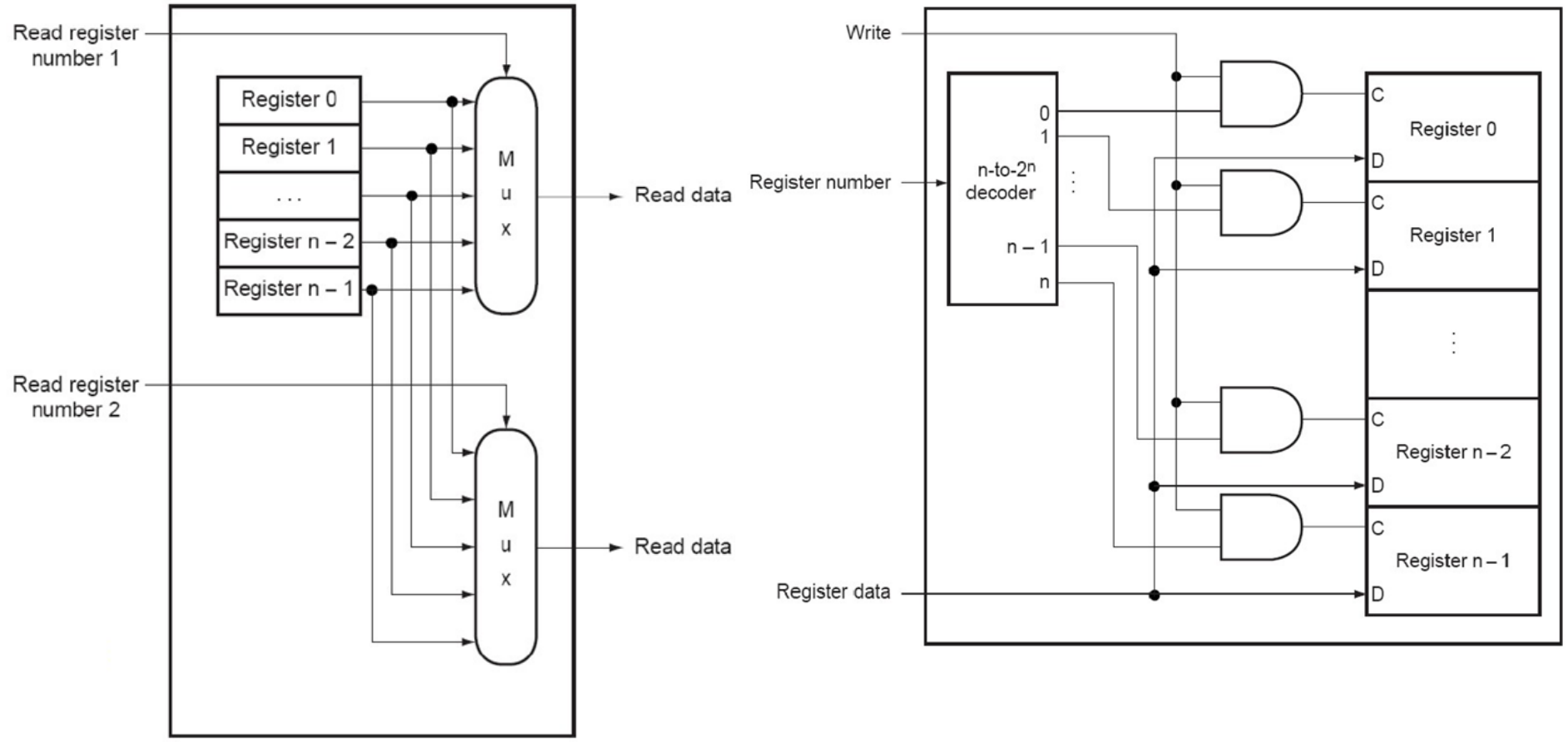
MIPS指令数据通路总图

add $t1, $t2, $t3
addi $s3,$s3,4
lw $s1,100($s2)
sw $s1,100($s2)



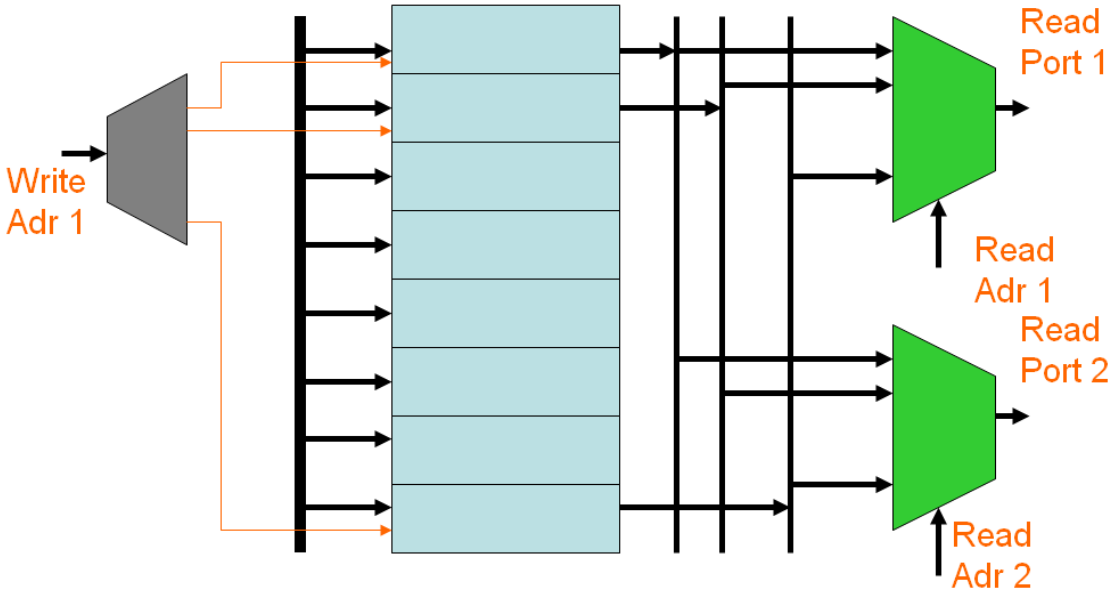
op(6 bits)	rs(5 bits)	rt(5 bits)	rd(5 bits)	shamt(5 bits)	funct(6 bits)
op(6 bits)	rs(5 bits)	rt(5 bits)	addr/immediate(16 bits)		
op(6 bits)	addr(26 bits)				

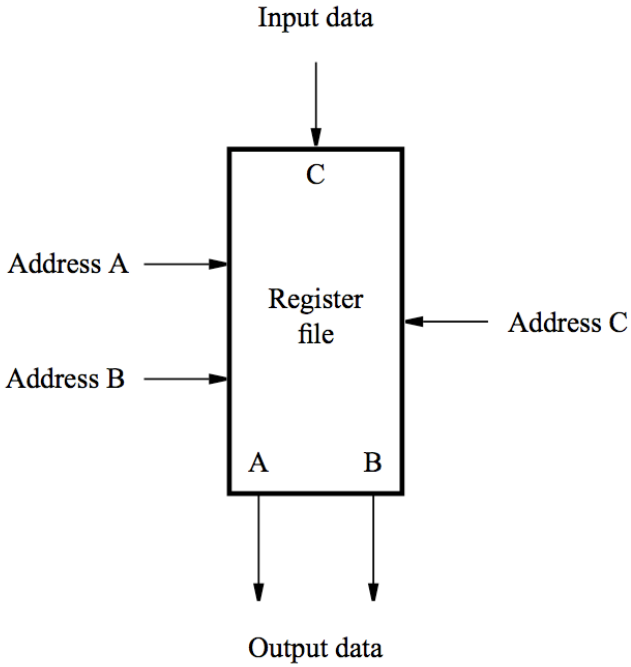
通路中的存储-RegisterFile



RegFile读写操作

- 寄存器文件在一个周期中有两个读和一个写操作。
  - ✓ 在一个周期内，某个REG可以同时完成读写操作，但读出的是上一个周期写入的值
- 寄存器文件不能同时进行读和写操作。两种设计方法：
  - ✓ 后写（late write）：在前半周期读数据，在后半周期写数据。
  - ✓ 先写（early write）：与后写相反。
  - ✓ 两种方式各有什么优缺点？（思考）





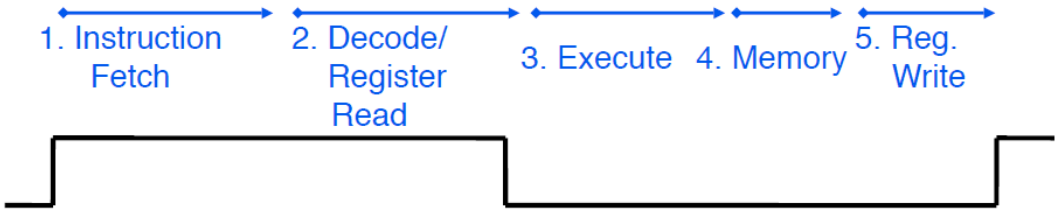
数据通路设计➔控制部件设计

控制信号的作用？

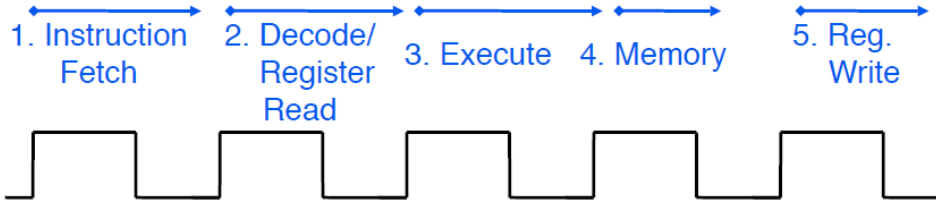
控制信号的来源？

实现 单周期->多周期

- 指令周期 = m 机器周期。机器周期 = n 时钟周期
- 单周期实现：指令周期 = 1 机器周期 = 1 时钟周期
  - ✓ All stages of an instruction are completed within one ***long*** clock cycle.
  - ✓ 所需控制信号同时生成

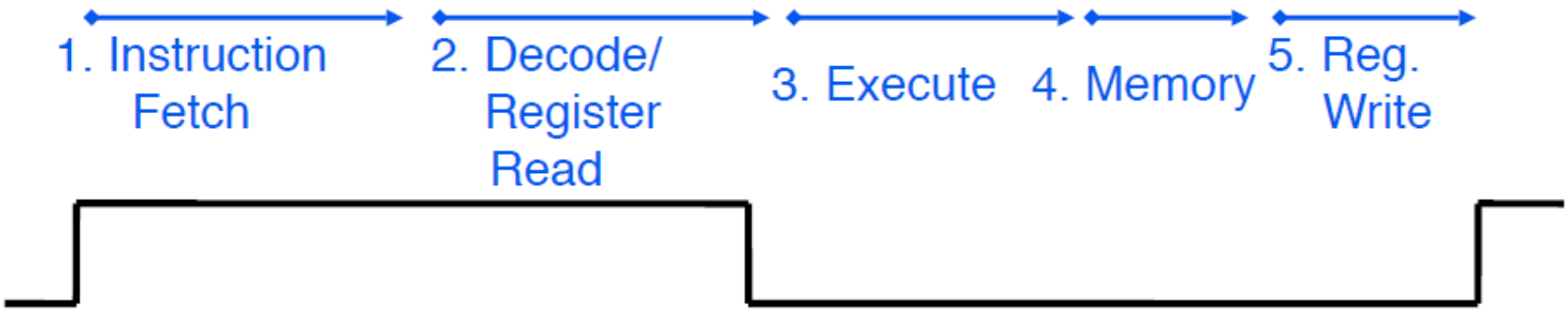


- 多周期实现：指令周期 = n 机器周期 = n 时钟周期
  - ✓ Only one stage of instruction per clock cycle
  - ✓ 按时钟周期（= 机器周期）生成当前周期所需控制信号

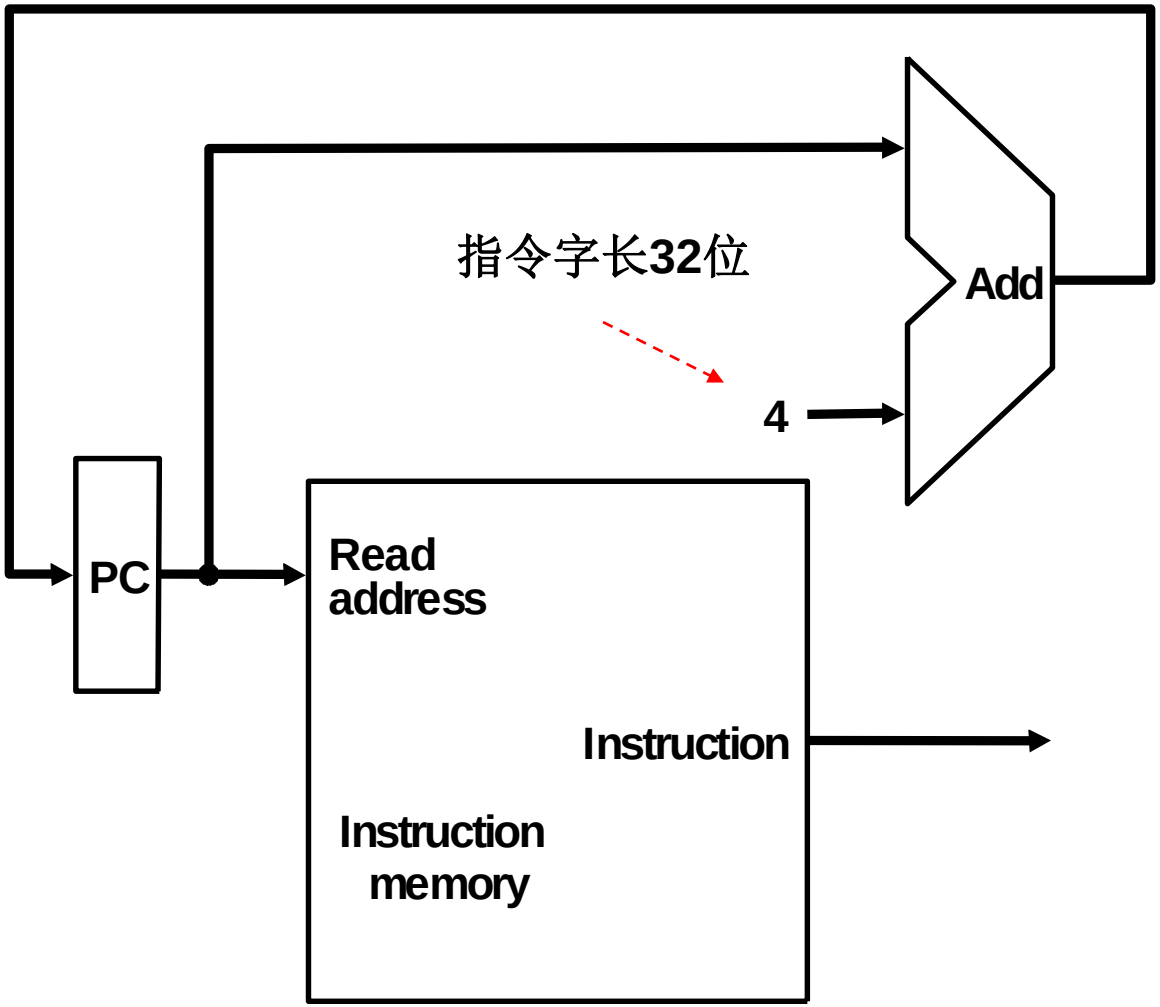


单周期

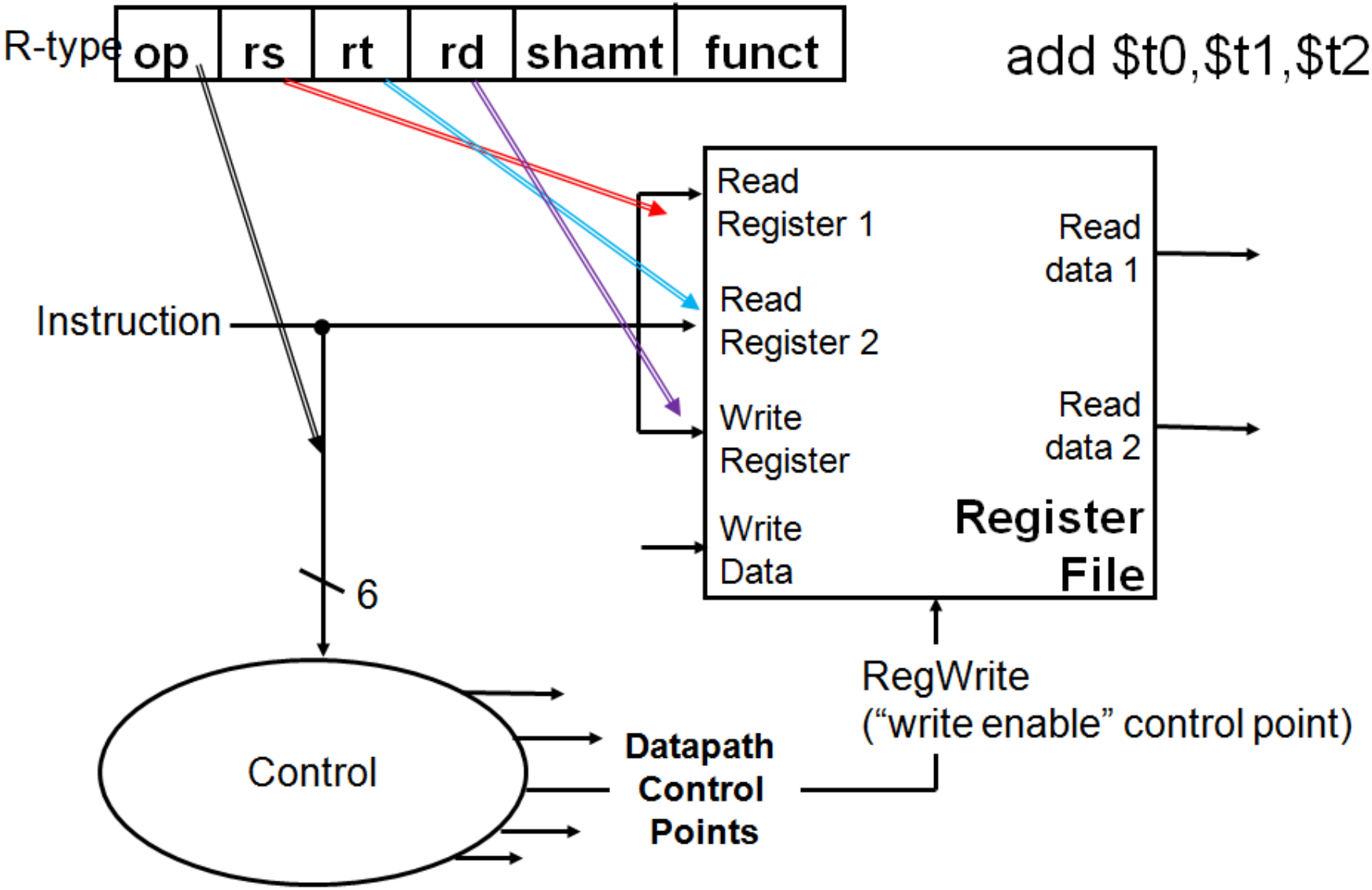
- 定长指令周期
  - ✓All stages of an instruction are completed within one ***long*** clock cycle.
  - ✓采用时钟边沿触发方式
    - 所有指令在时钟的一个边开始执行，在下一个边结束



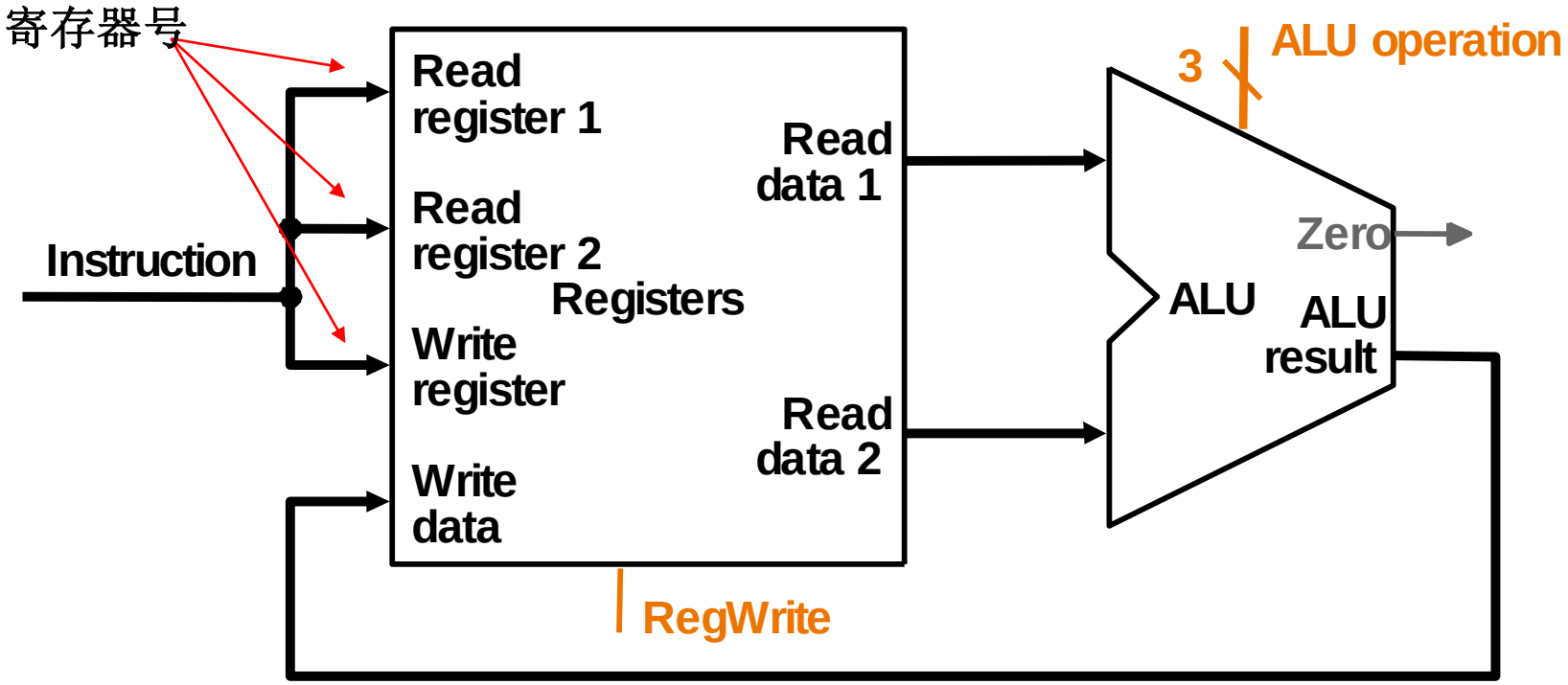
取指 Instruction Fetch



译码 Decode


R-type
op
rs
rt
rd
shamt
funct
add $t0,$t1,$t2
Read Register 1
Read Register 2
Write Register
Write Data
Read data 1
Read data 2
Register File
Instruction
6
Control
Datapath Control Points
RegWrite
("write enable" control point)

R-type指令的执行



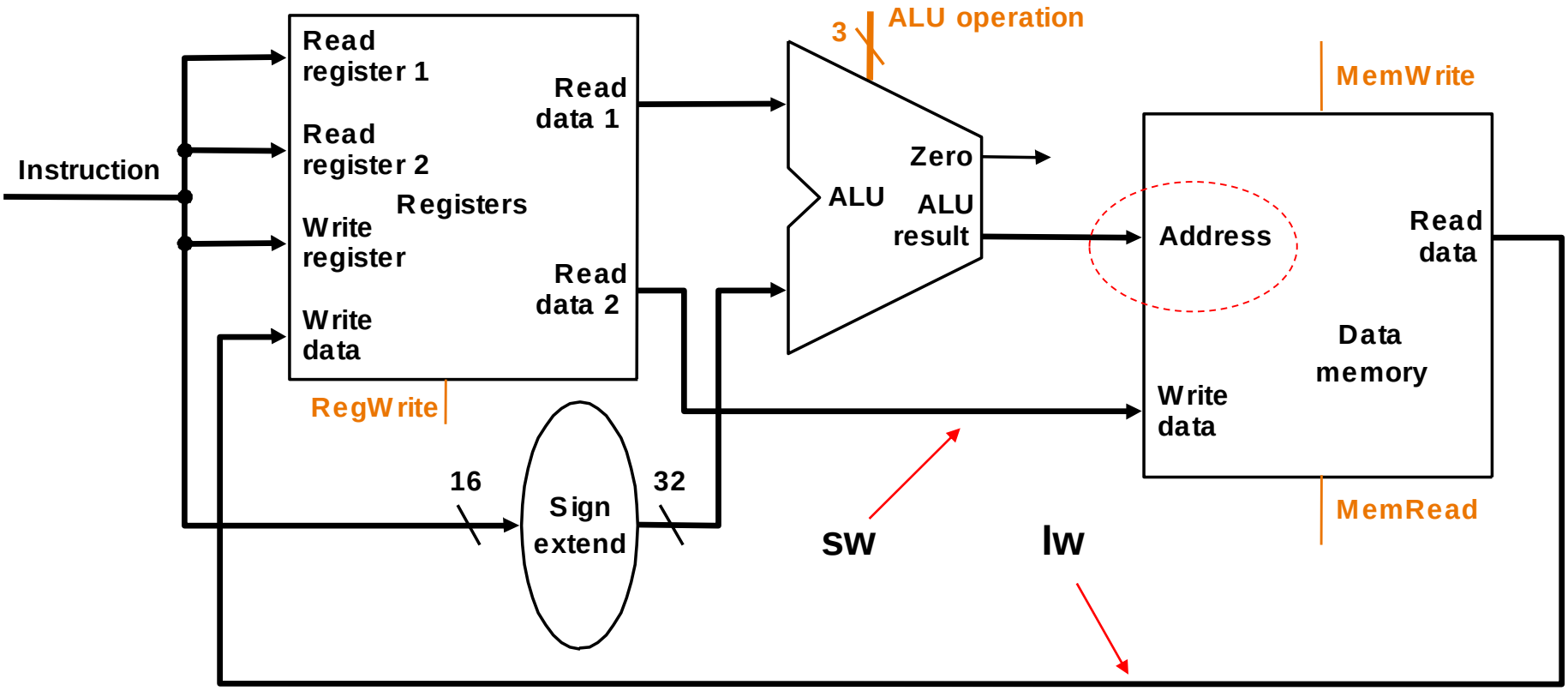
R-type	op(6 bits)	rs(5 bits)	rt(5 bits)	rd(5 bits)	shamt(5 bits)	funct(6 bits)

- 寄存器堆操作
  - 读：给出寄存器编号，则寄存器的值自动送到输出端口
  - 写：需要寄存器编号和控制信号RegWrite，时钟边沿触发
  - 在一个周期内，REG可以同时完成读写操作，但读出的是上一个周期写入的值(读后写)

访存指令的执行

I-type	op(6 bits)	rs(5 bits)	rt(5 bits)	addr (16 bits)

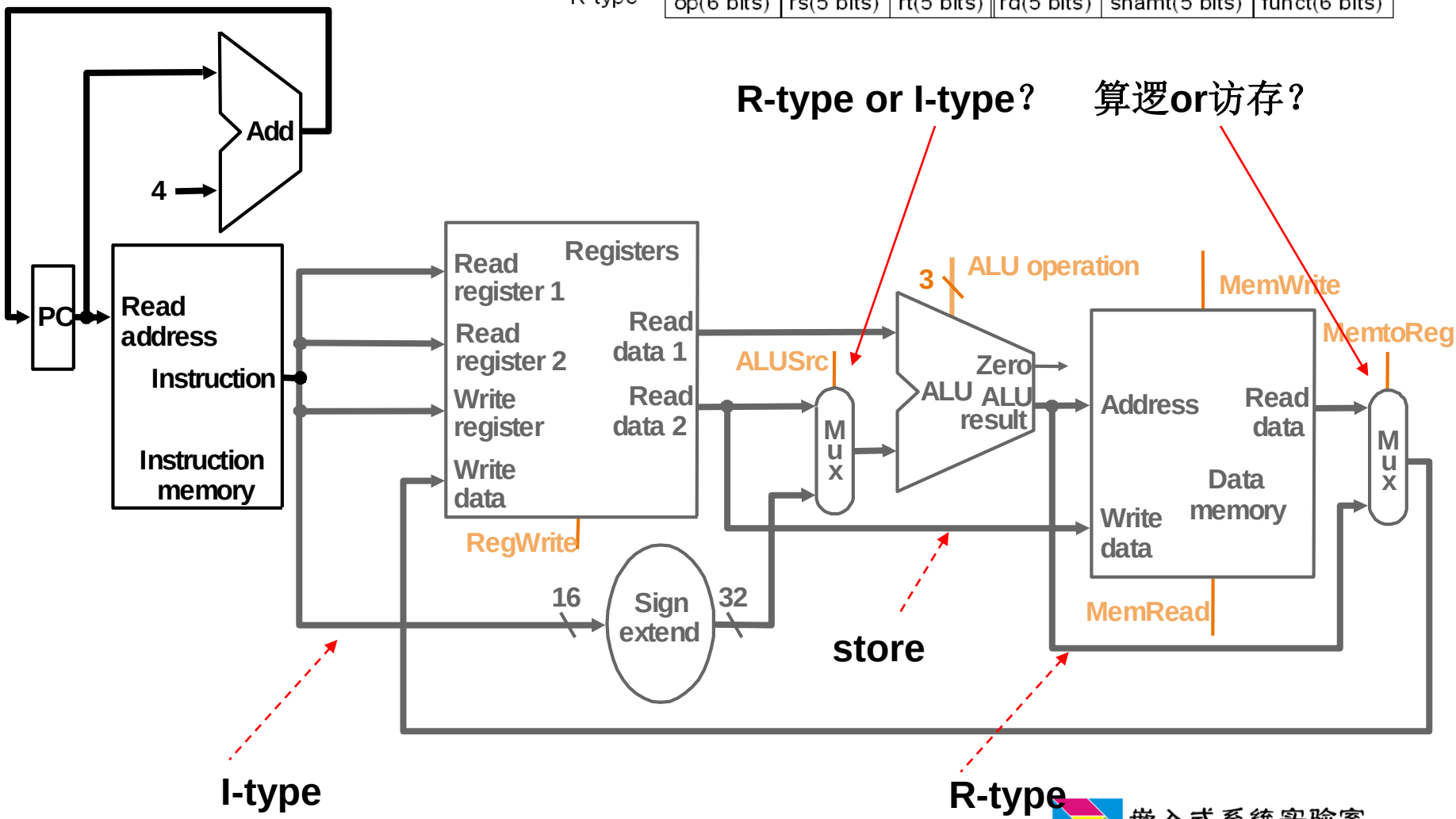
rs：基址
rt：ld的目的/sw的源



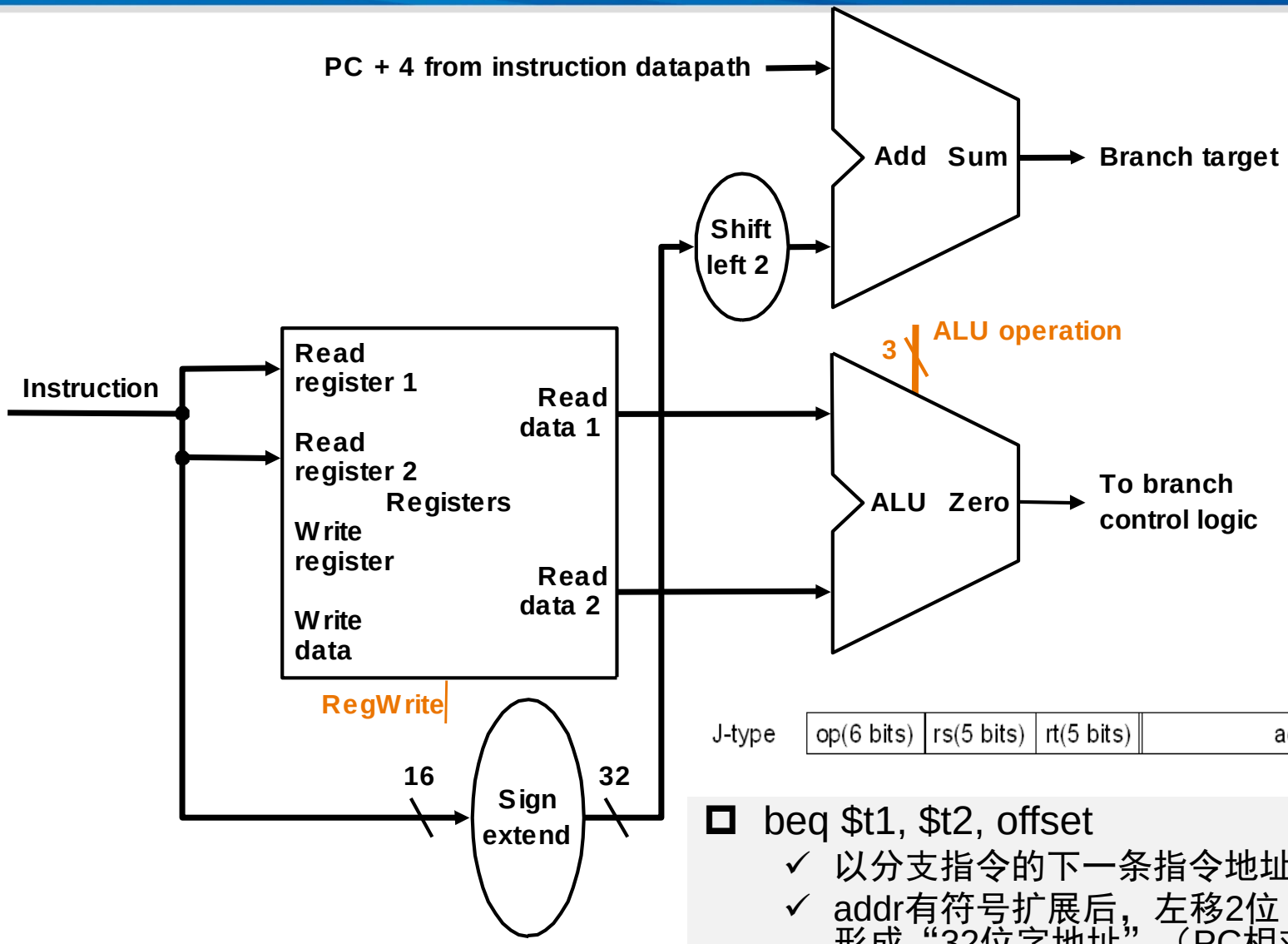
- lw $t1, offset($t2)； M($t2+offset) -> $t1
- sw $t1, offset($t2)； $t1 -> M($t2+offset)
- 需要对指令字中的16位偏移进行32位带符号扩展

访存指令和算逻指令的数据通路综合

I-type	op(6 bits)	rs(5 bits)	rt(5 bits)	addr (16 bits)		
R-type	op(6 bits)	rs(5 bits)	rt(5 bits)	rd(5 bits)	shamt(5 bits)	funct(6 bits)



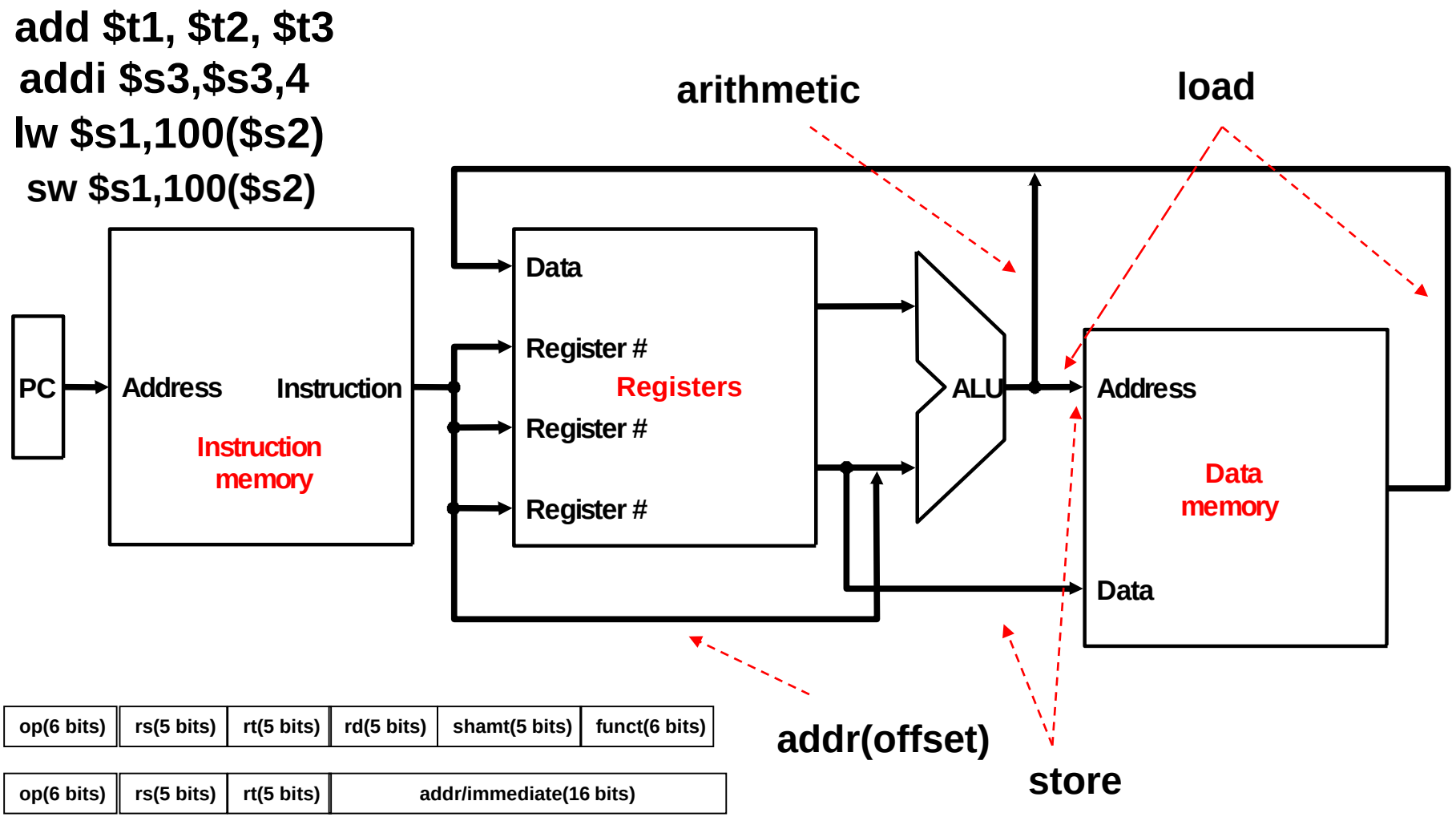
条件转移beq



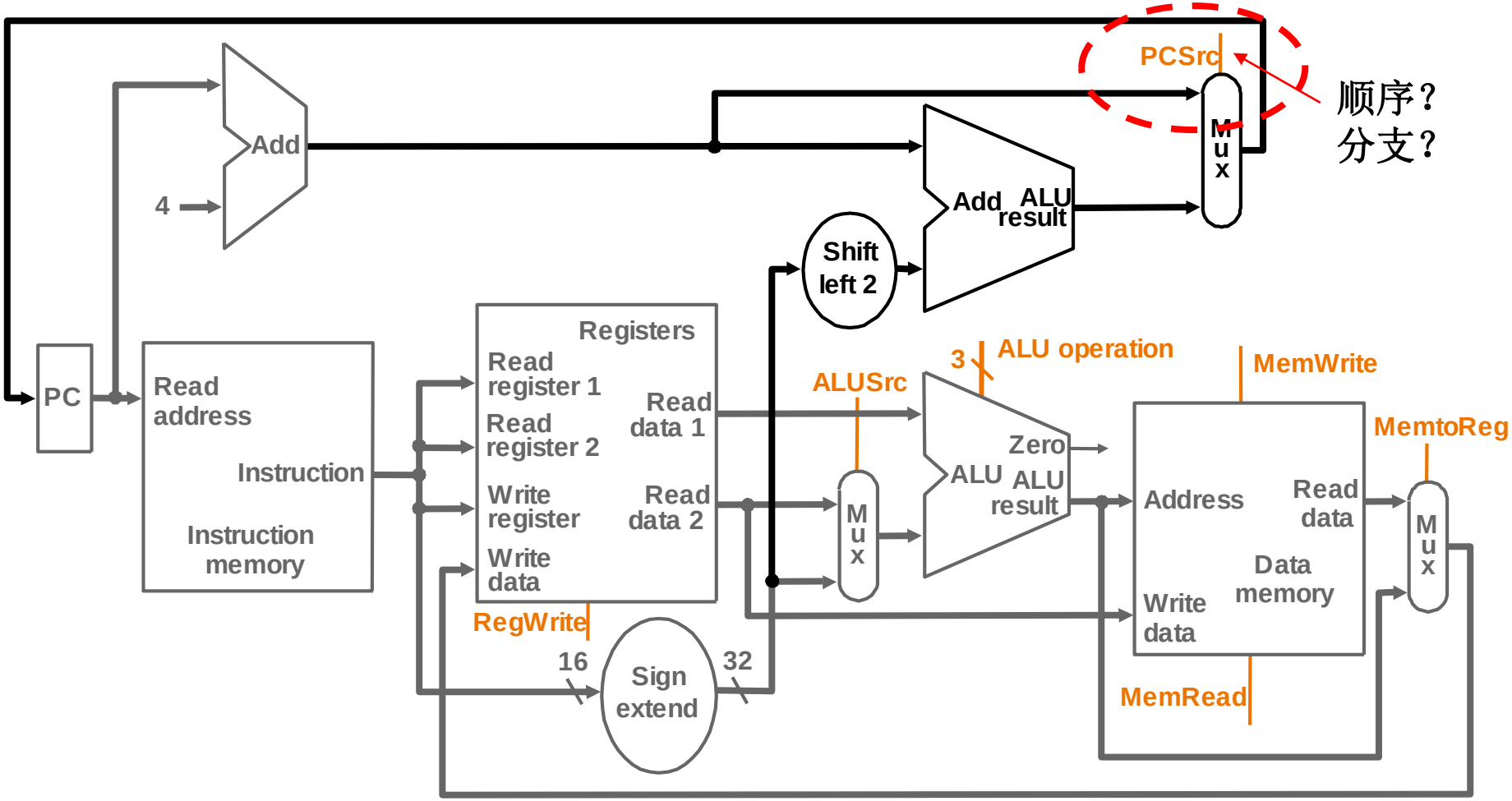
J-type	op(6 bits)	rs(5 bits)	rt(5 bits)	addr (16 bits)

- beq $t1, $t2, offset
  - ✓ 以分支指令的下一条指令地址为基址
  - ✓ addr有符号扩展后，左移2位（字对齐），形成“32位字地址”（PC相对寻址）

复习：MIPS指令数据通路总图



R-/I-/J-type操作数据通路总图



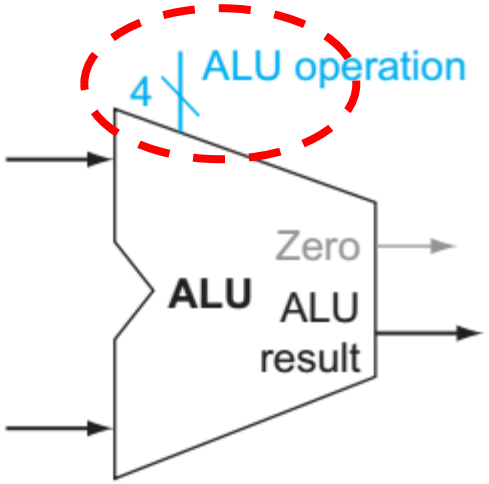
控制信号如何处理？

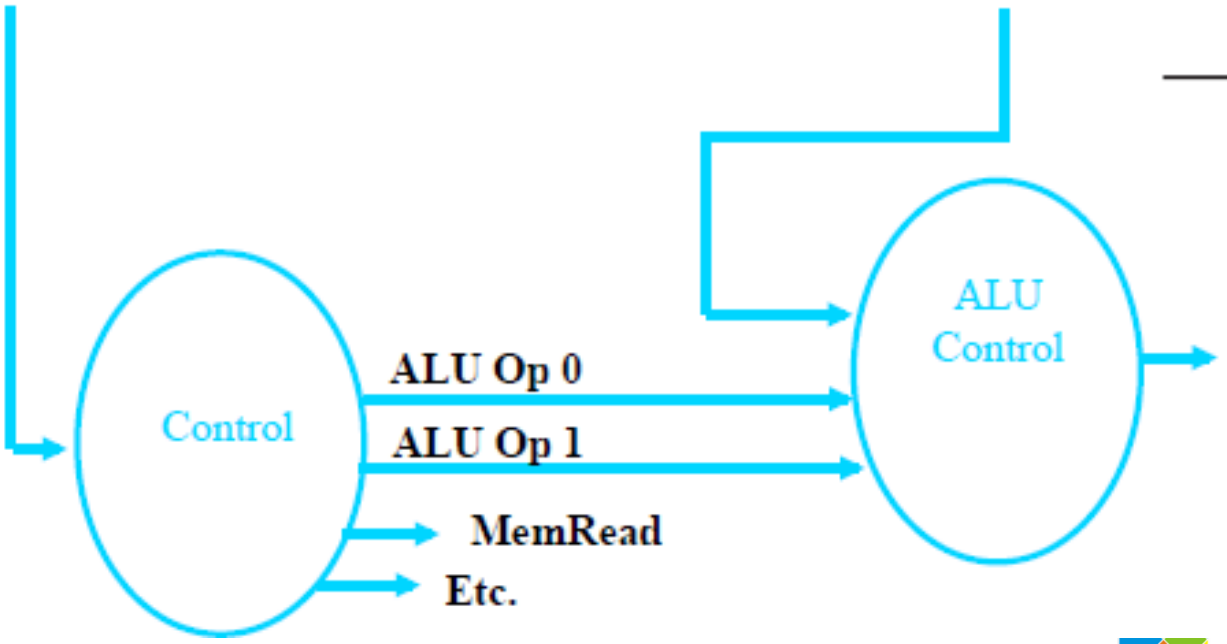
控制器：ALU控制信号

op(6 bits)	rs(5 bits)	rt(5 bits)	rd(5 bits)	shamt(5 bits)	funct(6 bits)
op(6 bits)	rs(5 bits)	rt(5 bits)	addr/immediate(16 bits)		
op(6 bits)	addr(26 bits)				

Example: add $8, $17, $18

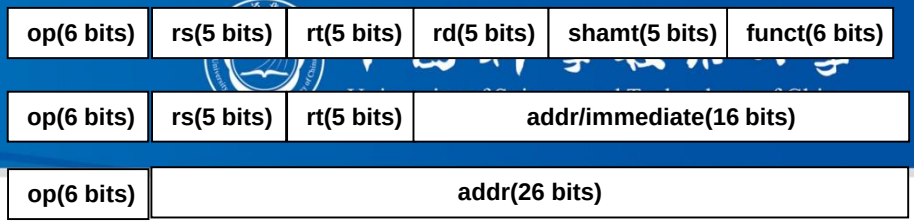
000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct





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

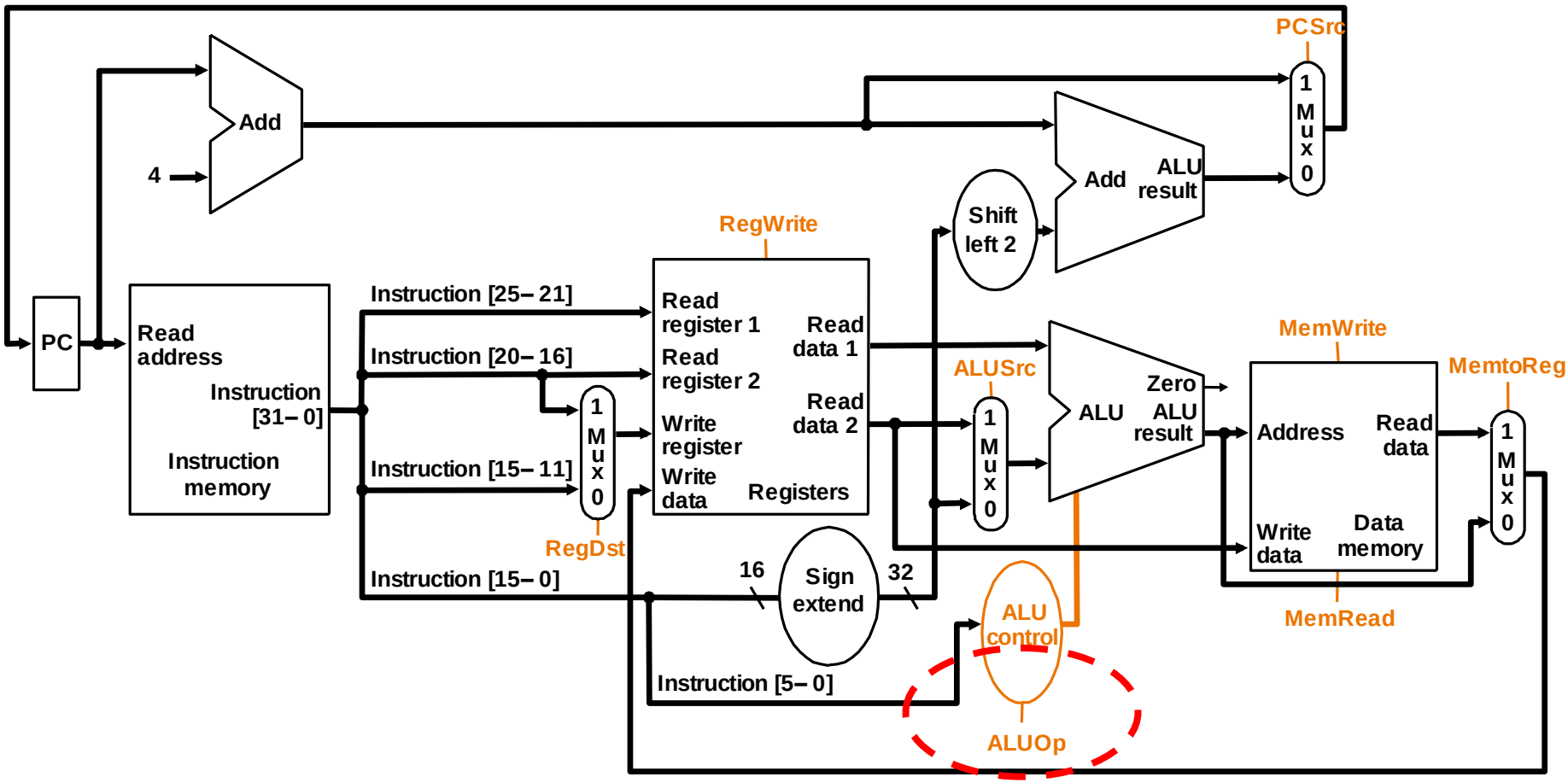
ALU控制信号



Instruction opcode	ALUop	Instruction operation	Funct field	desired ALU action	ALU ctrl input
LW	00	Load word	xxxxxx	add	0010
SW	00	store word	xxxxxx	add	0010
beq	01	Branch eq	xxxxxx	subtract	0110
R-type	10	Add	100000	Add	0010
R-type	10	Substract	100010	Substract	0110
R-type	10	And	100100	And	0000
R-type	10	Or	100101	Or	0001
R-type	10	**Set on less than**	101010	**Set on less than**	0111

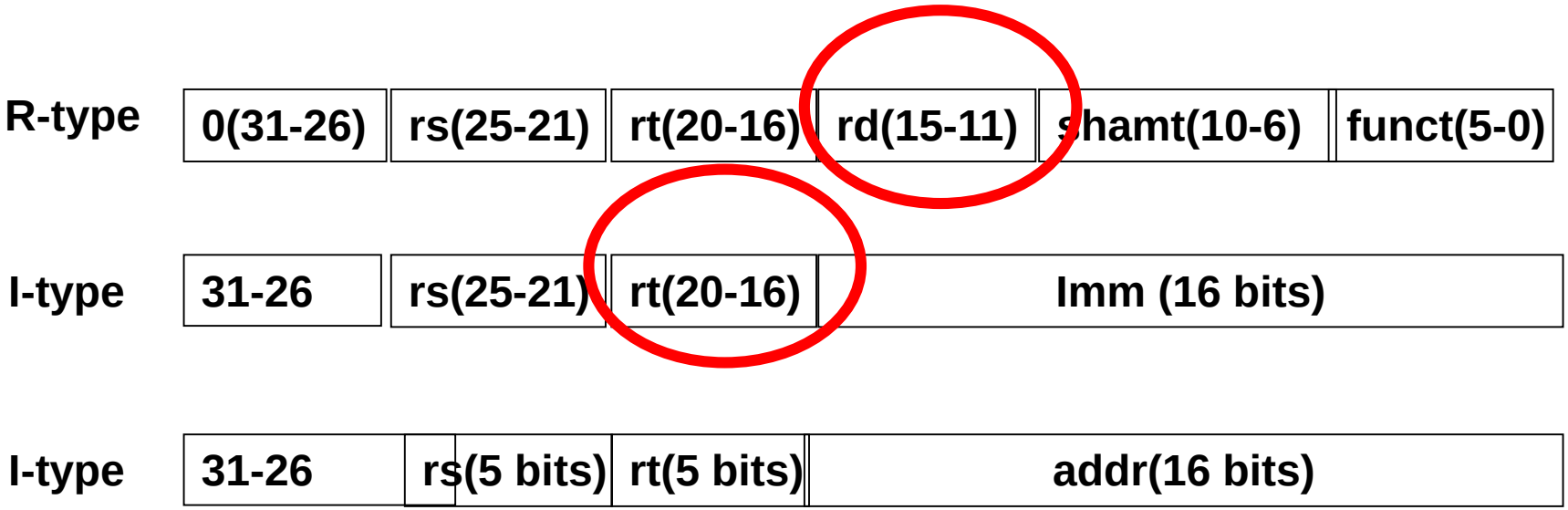
2位ALUop和6位func组合，产生ALU_ctrl_input（即4位ALU operation）

ALU控制选择



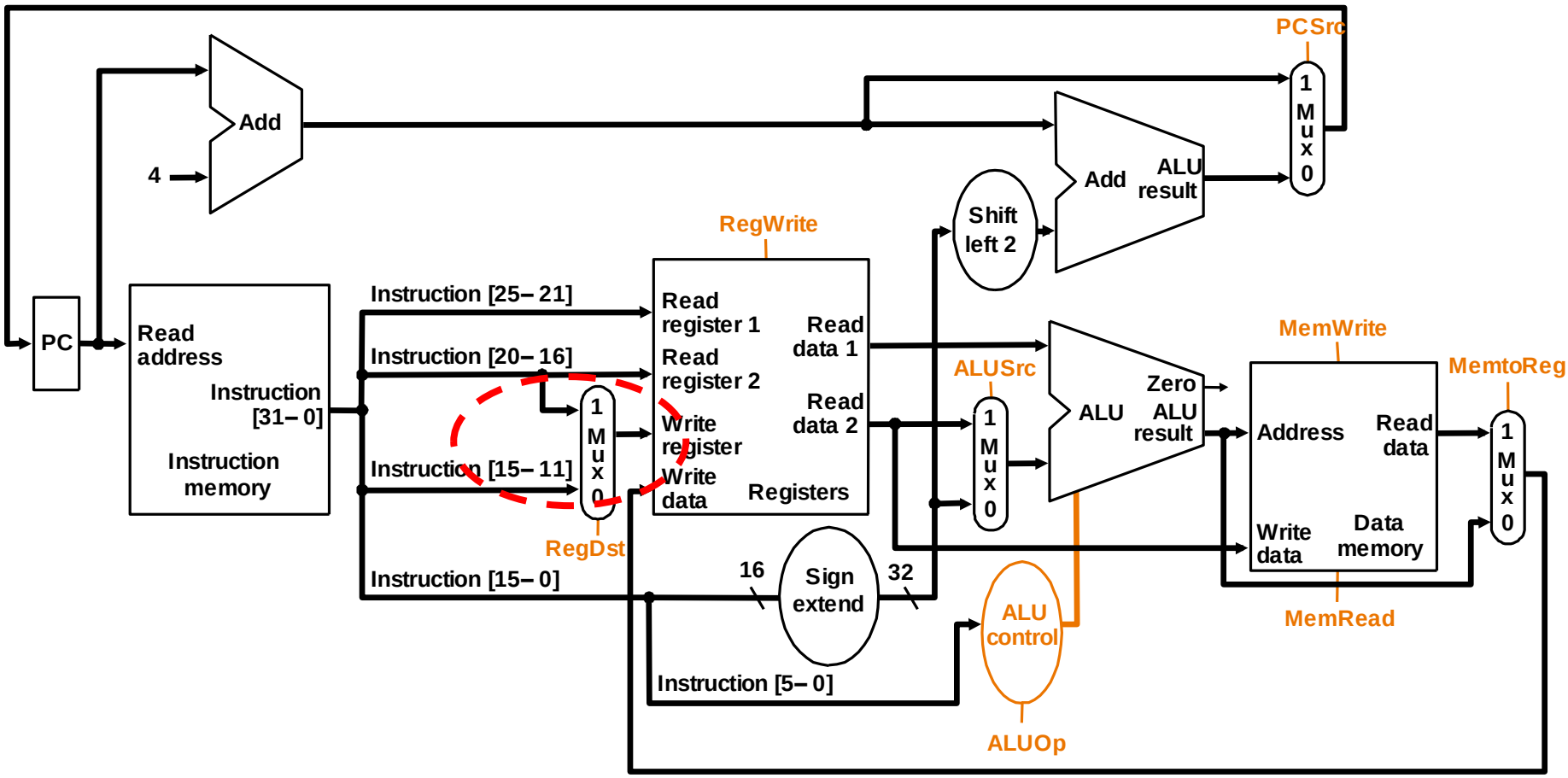
- 两位ALUop和func组合产生ALU控制选择

控制器：目的寄存器选择



- 指令格式分析
  - ✓操作码：在31-26
  - ✓操作类型R-type：需要参考5-0
  - ✓目的地址：需要对目的寄存器进行选择控制
    - 对R-type指令，在rd
    - 对load/I-ALU指令，在rt

目的寄存器选择

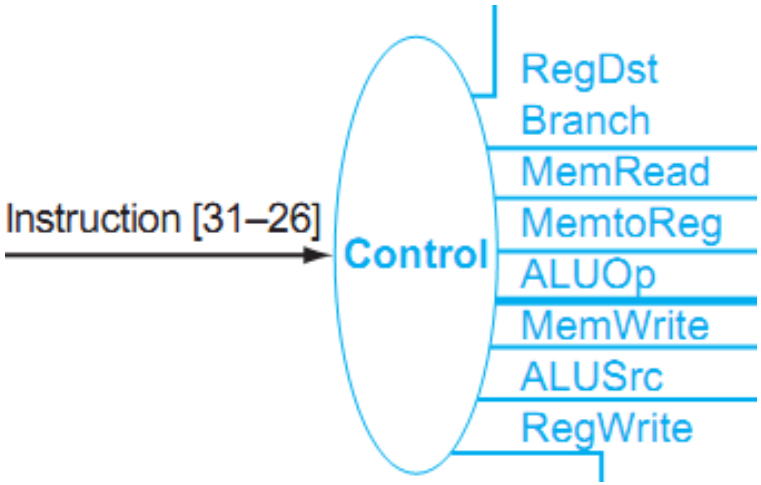


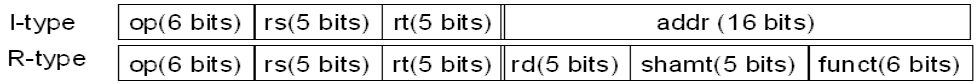
R-type	0(31-26)	rs(25-21)	rt(20-16)	rd(15-11)	shamt(10-6)	funct(5-0)

I-type	31-26	rs(25-21)	rt(20-16)	Imm (16 bits)

控制器：控制信号生成

- op域（6位）译码产生的控制信号：8个
  - ✓ RegDst：选择rt或rd作为写操作的目的寄存器
    - R-type指令（1）与 I-type指令二选一（0）
  - ✓ RegWrite：寄存器写操作控制
  - ✓ ALUSrc：ALU的第二个操作数来源
    - R-type指令（0）与 I-type指令（1）（含branch指令）二选一
  - ✓ ALUOp：R-type指令（2位）
  - ✓ MemRead：存储器读控制，load指令
  - ✓ MemWrite：存储器写控制，store指令
  - ✓ MemtoReg：目的寄存器数据来源
    - R-type（I类ALU）指令与load指令二选一
  - ✓ Branch：是否分支指令（产生PCSrc）





- PCSrc：nPC来源控制，顺序与分支成功二选一
  - ✓ “是否beq指令（Branch）”&“ALU的Zero状态有效”

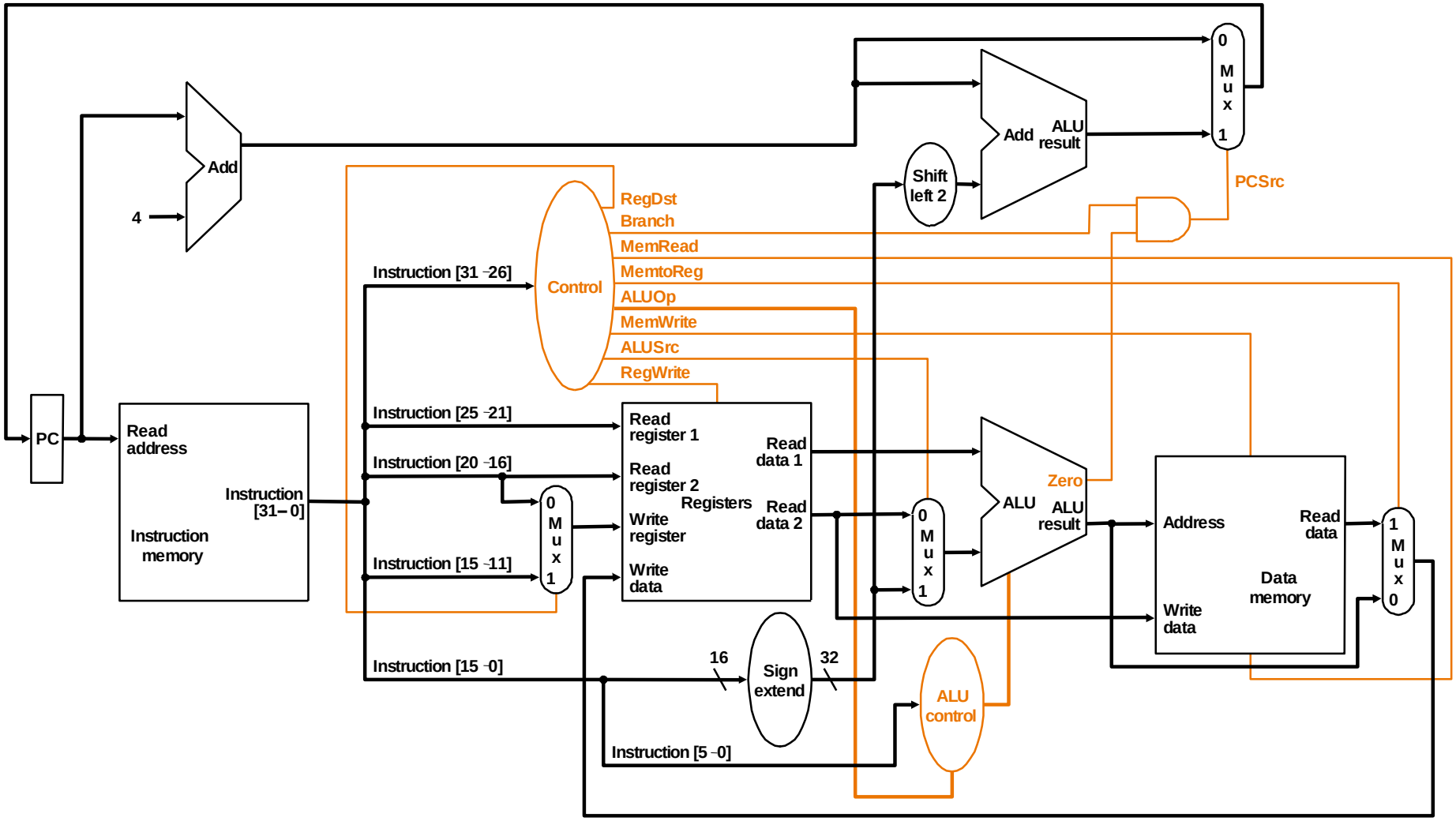
控制信号例子

- 目的寄存器编号Rd/Rt
  - R-type指令（1）
  - load指令（0）
- ALU二操作数来源
  - R-type指令（0）
  - I-type指令（1）
- 目的寄存器来源
  - MEM（1）
  - ALU（0）
- 目的寄存器写
- 存储器读
- 存储器写
- 是否为Branch
- ALUOP 2位 + 6位Func
  ➔4位ALU 控制信号

Instruction opcode	ALUop
LW	00
SW	00
beq	01
R-type	10

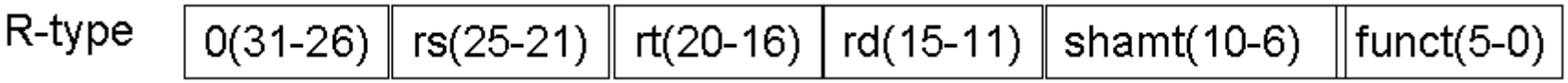
	Signal name	R-type	lw （31）	sw （35）	beq
inputs	op5（26）	0	1	1	0
	op4	0	0	0	0
	op3	0	0	1	0
	op2	0	0	0	1
	op1	0	1	1	0
	op0（31）	0	1	1	0
outputs	RegDst				
	ALUSrc				
	MemtoReg				
	RegWrite				
	MemRead				
	MemWrite				
	Branch				
	ALUop1				
	ALUop0				

主控制部件

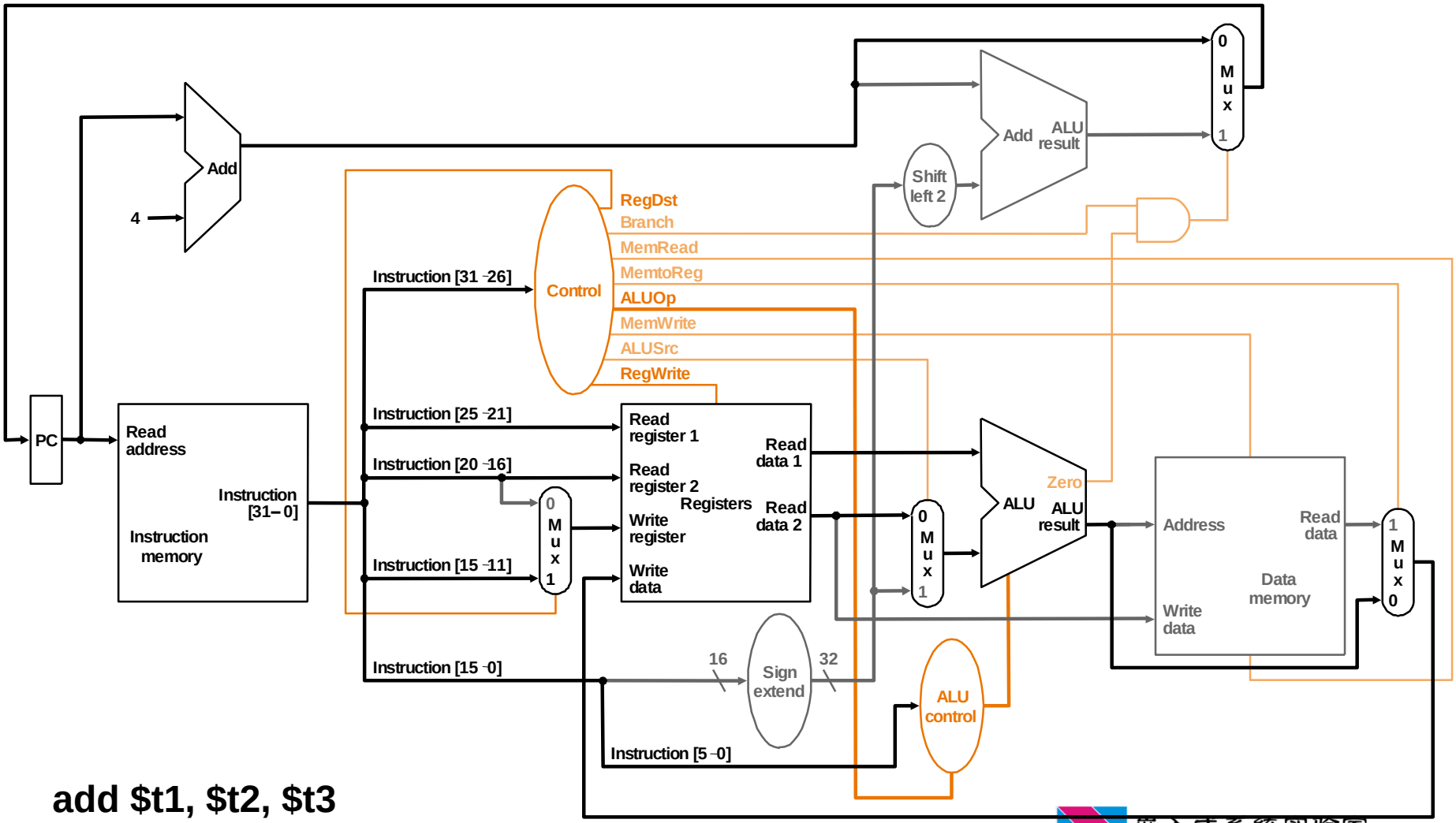

Add
4
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [31–26]
Control
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
0 M u x 1
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign extend
32
Shift left 2
Add
ALU result
0 M u x 1
PCSrc
0 M u x 1
ALU
Zero
ALU result
ALU control
Address
Read data
Data memory
Write data
1 M u x 0

R-type指令的执行过程

- add $t1, $t2, $t3;　　$t2+$t3->$t1
- 在一个周期内完成如下动作
  - ✓第一步：取指和PC＋4
  - ✓第二步：读两个源操作数寄存器$t2和$t3
  - ✓第三步：ALU操作
  - ✓第四步：结果写回目的寄存器$t1



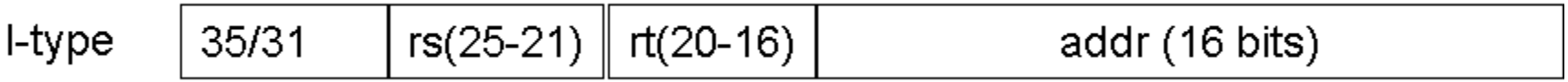
R-type指令的执行路径



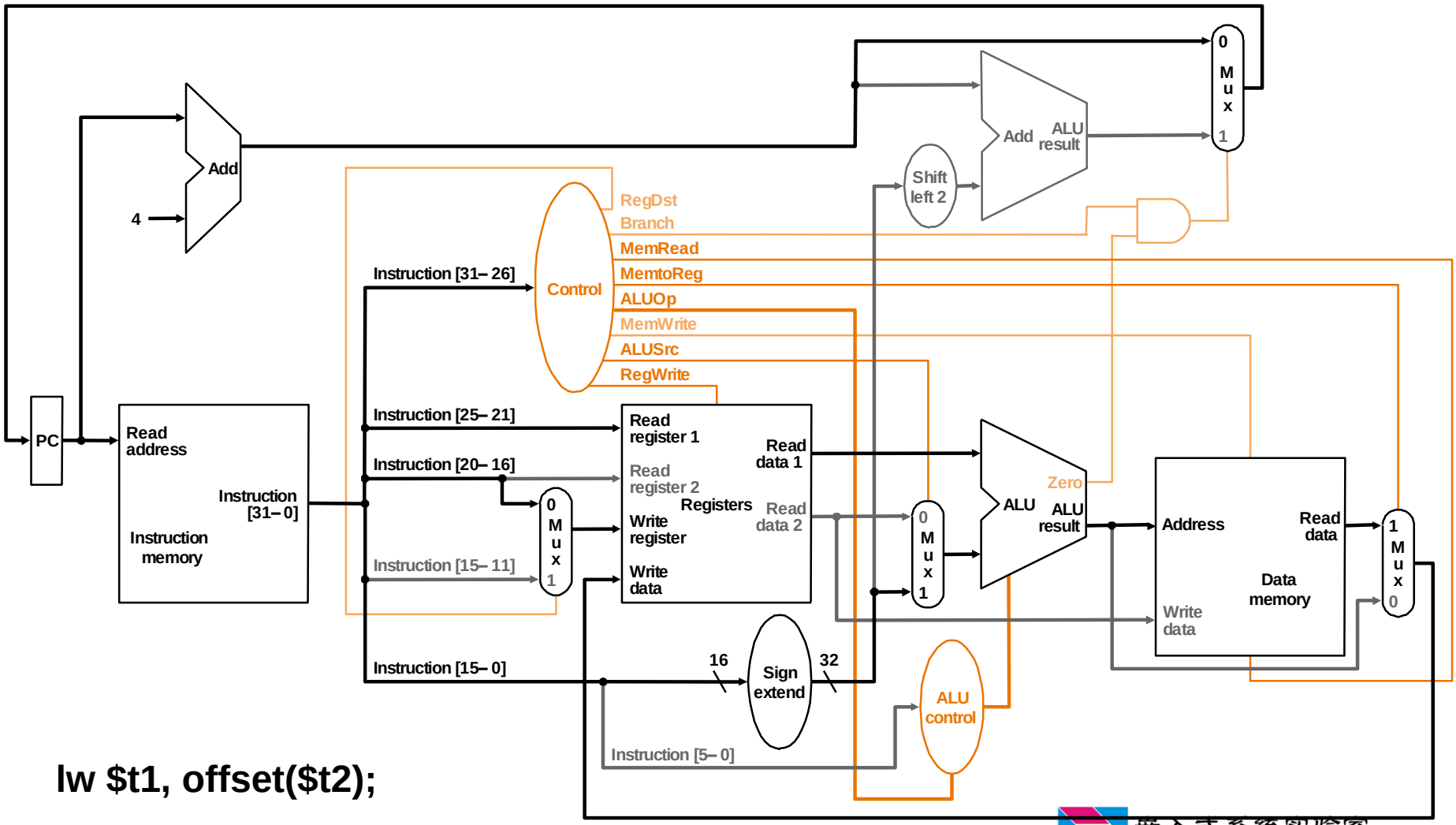
add $t1, $t2, $t3

lw指令的执行过程

- lw $t1, offset($t2);　　M($t2+offset) -> $t1
- 第一步：取指和PC＋4
- 第二步：读寄存器$t2
- 第三步：ALU操作完成$t2与符号扩展后的16位offset加
- 第四步：ALU的结果作为访存地址，送往数据MEM
- 第五步：内存中的数据送往$t1



lw指令的执行路径



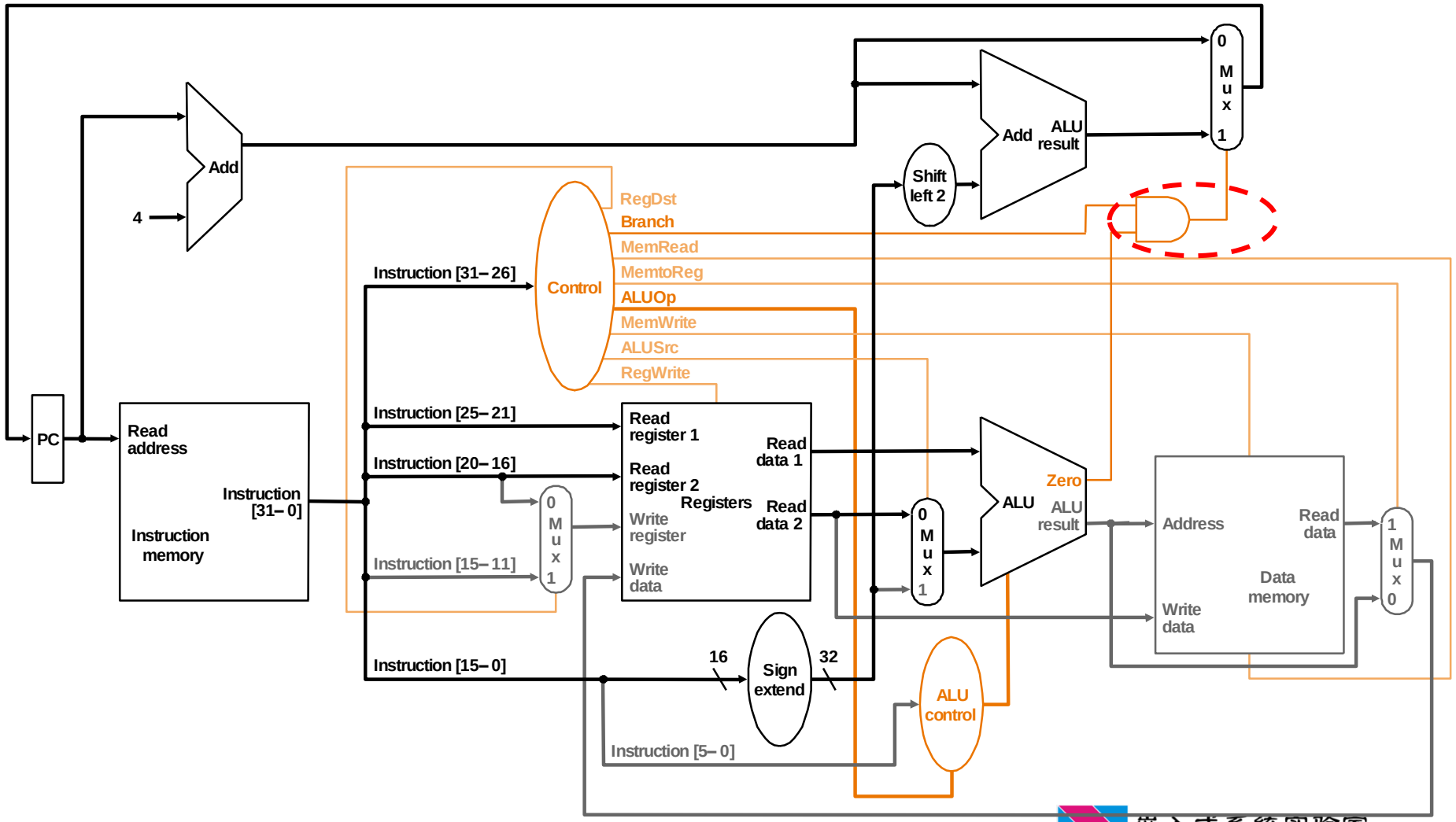
lw $t1, offset($t2);

beq指令的执行过程

- beq $t1, $t2, offset
- 第一步：取指和PC＋4
- 第二步：读寄存器$t1，$t2
- 第三步：ALU将$t1和$t2相减；PC+4与被左移两位并进行符号扩展后的16位offset相加，作为分支目标地址
- 第四步：ALU的Zero确定应送往PC的值

J-type	4	rs(5 bits)	rt(5 bits)	addr(16 bits)

beq的执行路径



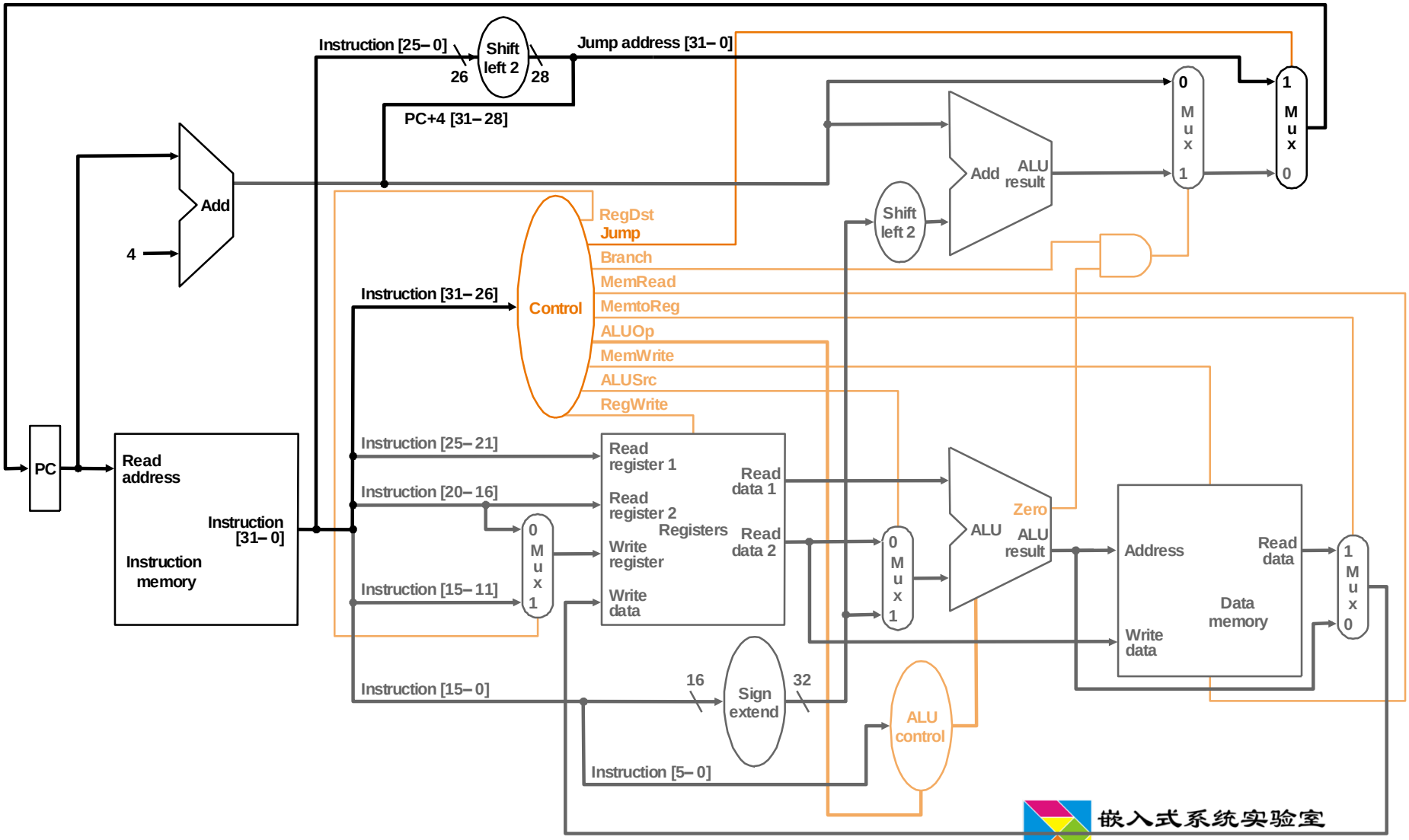
beq $t1, $t2, offset

jump指令的实现

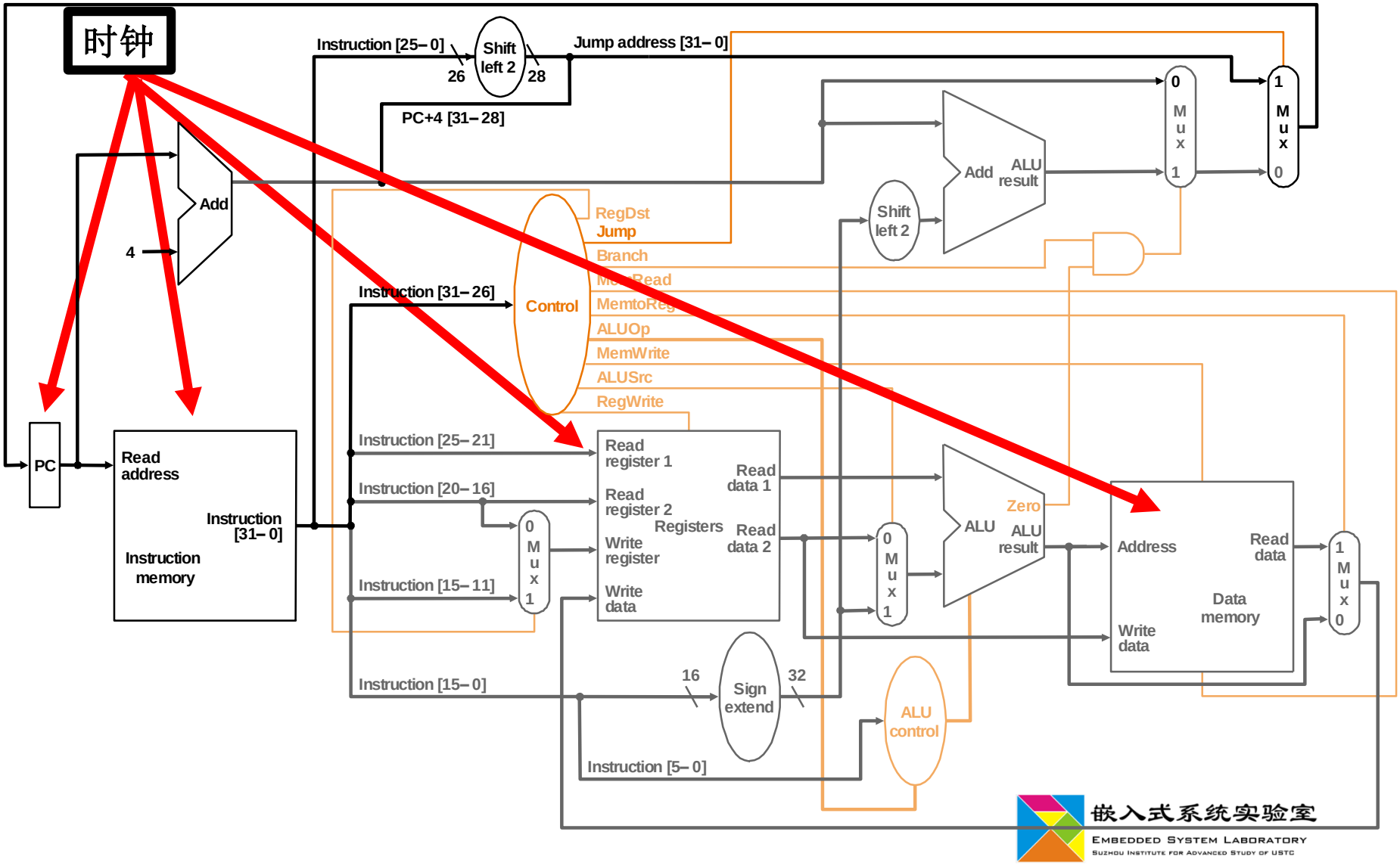
J-type	2(31-26)	addr(25-0)

- 无条件转移，关键在于目标地址的拼装
  - ✓PC+4的最高4位
  - ✓指令字中的26位地址
  - ✓最低两位补00
- “拼装”：只需合并地址总线
- 增加一个jump指令识别控制

jump指令的实现


Instruction [25– 0]
Shift left 2
26
28
Jump address [31– 0]
PC+4 [31– 28]
Add
4
0
M
u
x
1
1
M
u
x
0
Add
ALU result
Shift left 2
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31– 26]
PC
Read address
Instruction [31– 0]
Instruction memory
Instruction [25– 21]
Instruction [20– 16]
Instruction [15– 11]
Instruction [15– 0]
Instruction [5– 0]
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign extend
32
ALU control
Zero
ALU
ALU result
Address
Read data
Data memory
Write data

Single-Cycle Processor: Clocking



时钟周期的确定方法

- 时钟周期由关键路径确定
- 关键路径的选择=哪种指令执行时间最长？
  - ✓PC时钟初始化=30ps　读内存=250ps
  - ✓寄存器读=150ps　寄存器写=20ps
  - ✓ALU计算=200ps　多个多路选择器共25ps（几个？）
- 画出关键路径，并计算主频

$$
\begin{aligned}
T_C &= t_{\text{pcq-PC}} + t_{\text{mem-I}} + t_{\text{RFread}} + t_{\text{ALU}} + t_{\text{mem-D}} + t_{\text{mux}} + t_{\text{RFsetup}} \\
&= (30 + 250 + 150 + 200 + 250 + 25 + 20)\ \text{ps} \\
&= 925\ \text{ps} \\
&= 1.08\ \text{GHz}
\end{aligned}
$$

定长单周期or不定长单周期？

- 设程序中load有24%，store有12%，R-type有44%，beq有18%，jump有2%。假设load, store,R,beq,jump指令的时间分别是8,7,6,5,2ns（如下表）。如果时钟周期固定，单周期的时钟为8ns；如果时钟周期不定长，单周期的时钟可以是2ns～8ns。试比较时钟定长单周期实现和不定长单周期实现的性能。

指令	inst MEM	Reg Read	ALU	Data MEM	Reg Write	Total
R-Type	2	1	2		1	6ns
lw	2	1	2	2	1	8ns
sw	2	1	2	2		7ns
beq	2	1	2			5ns
jump	2					2ns

单周期实现

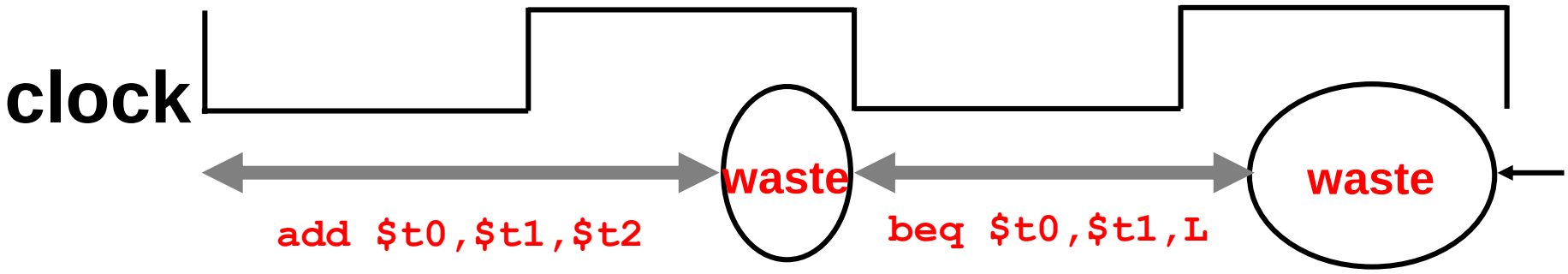
- 程序执行时间＝指令数×CPI×时钟周期时间
- 指令周期＝机器周期＝时钟周期
  - ✓CPI＝1

- 平均指令执行时间＝8×24%＋7×12%＋6×44%＋5×18%＋2×2%＝6.3ns
- 因此，变长单周期实现较定长单周期实现快8/6.3=1.27倍

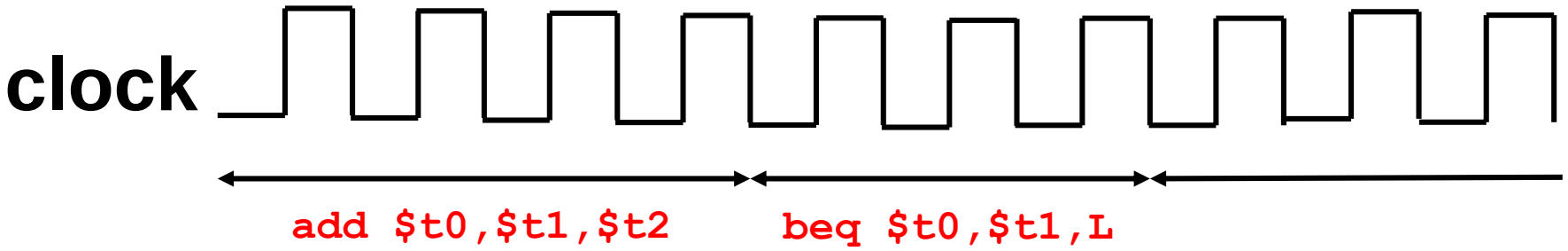
变长单周期实现?

定长单周期 vs. 不定长多周期

Single-cycle Implementation： 定长指令周期



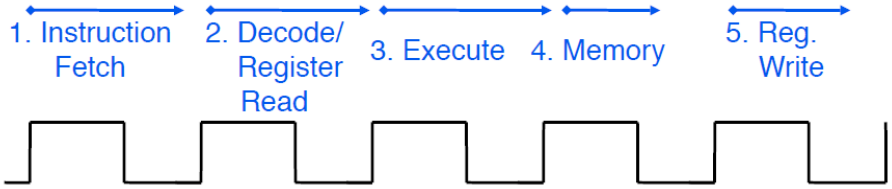
Multicycle Implementation： 不定长指令周期



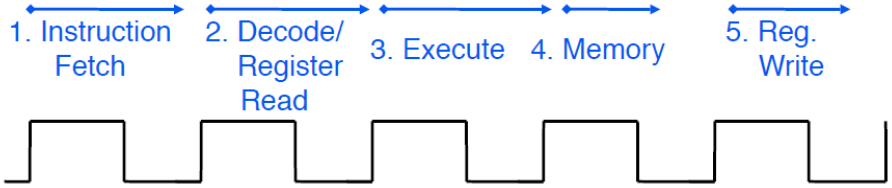
- **Multicycle Implementation:**
  less waste = higher performance

多周期实现

- 根据指令执行所使用的功能部件，将执行过程划分成多个阶段，每个阶段一个周期（机器周期）
  - ✓早结束的指令可以提升性能
- 时钟周期确定
  - ✓每个周期的工作尽量平衡
  - ✓假设一个周期内可以完成
    - 一次MEM访问，or
    - 一次寄存器访问（2 reads or one write），or
    - 一次ALU操作

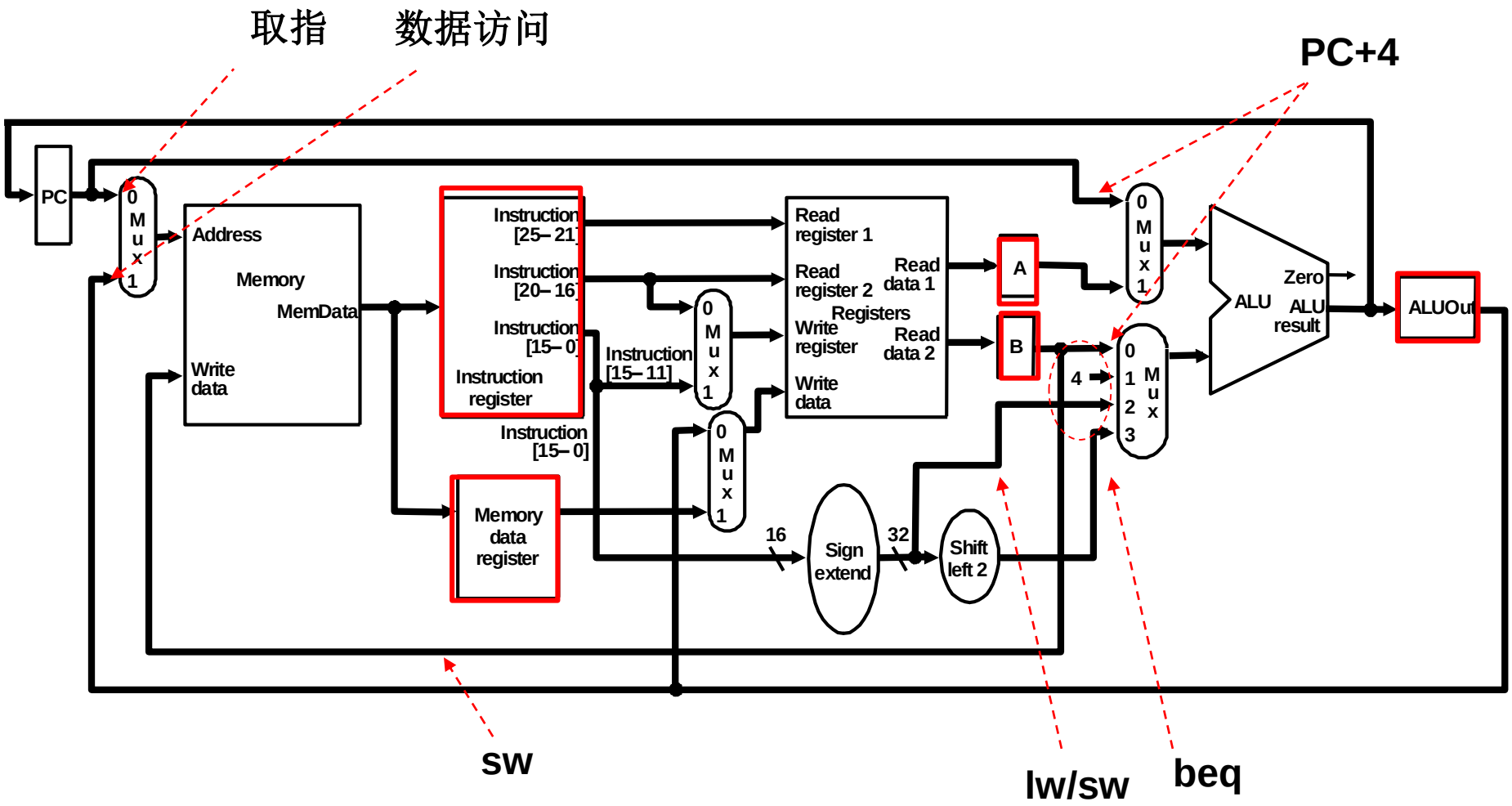


指令执行的阶段划分

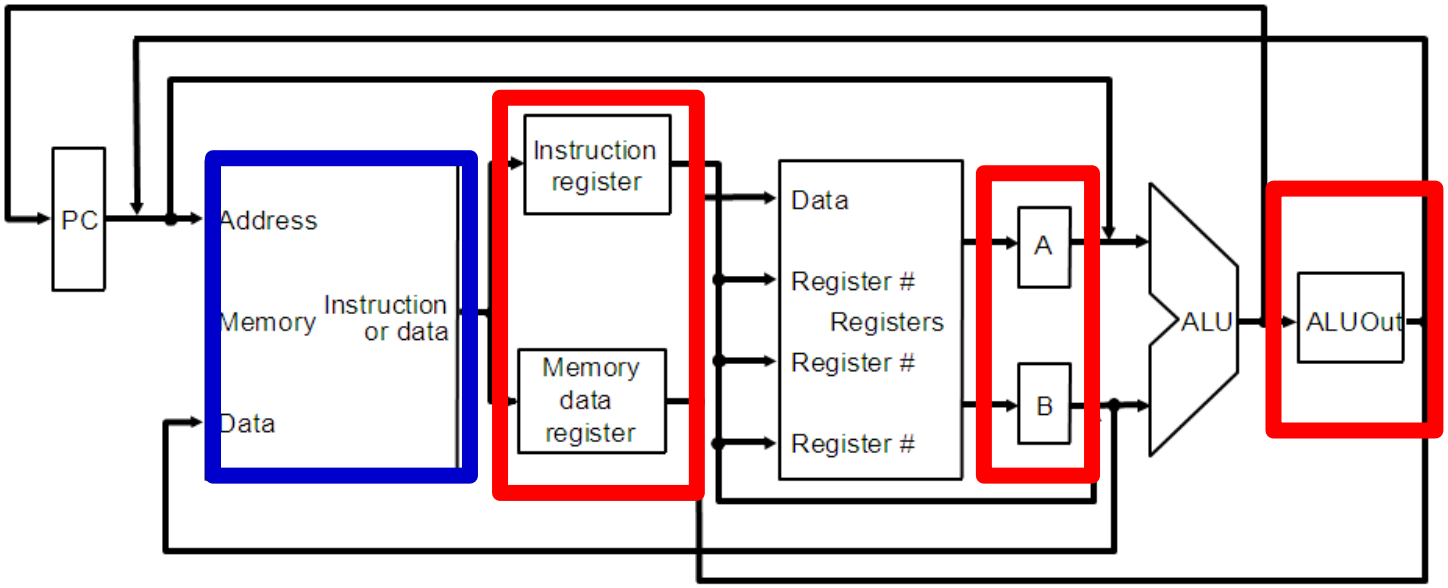


- 共5个阶段
  - ✓取指
  - ✓译码阶段、计算beq目标地址
  - ✓执行：R-type指令执行、访存地址计算，分支完成阶段
  - ✓访存：lw读，store和R-type指令完成阶段
  - ✓写回：lw完成阶段
- 注意
  - ✓定长机器周期：机器周期=时钟周期
  - ✓不定长指令周期：分别为3、4、5个机器周期
- 控制器根据机器周期标识发出控制信号

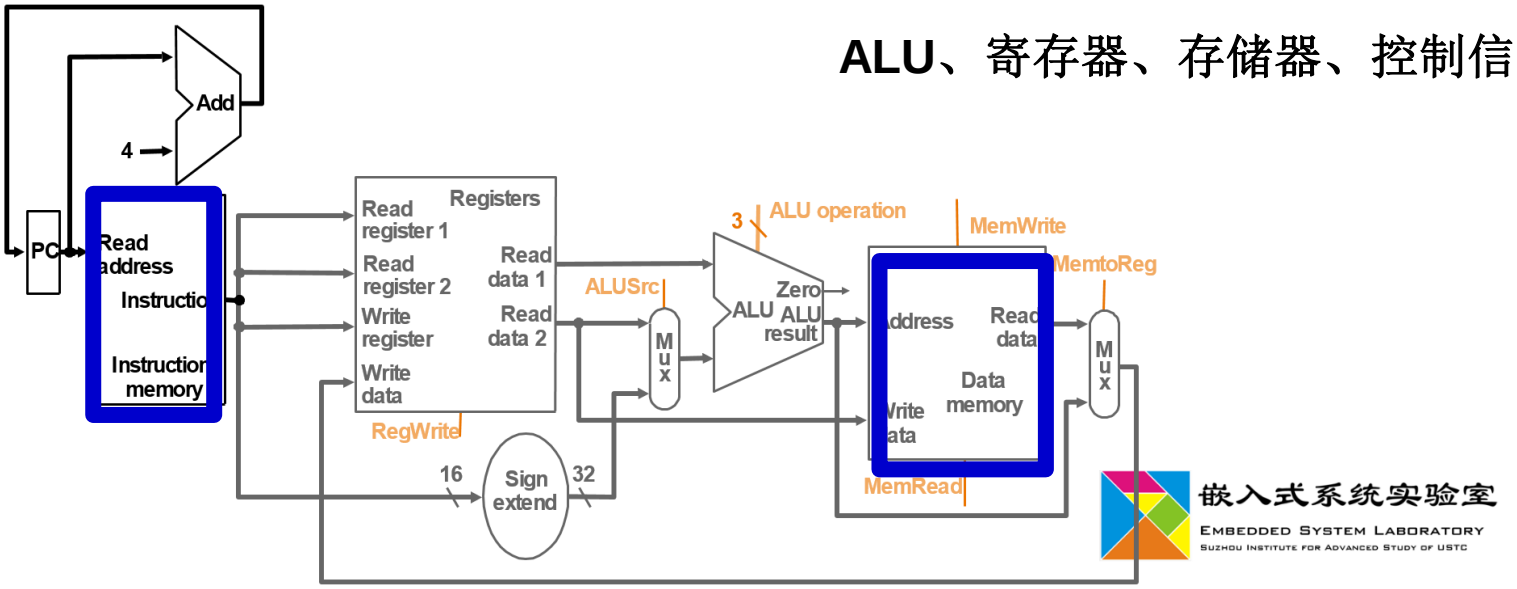
多周期数据通路


取指
数据访问
PC+4
PC
0
M
u
x
1
Address
Memory
MemData
Write
data
Instruction
[25– 21]
Instruction
[20– 16]
Instruction
[15– 0]
Instruction
register
Instruction
[15– 0]
Instruction
[15– 11]
Memory
data
register
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
A
B
4
ALU
Zero
ALU
result
ALUOut
16
Sign
extend
32
Shift
left 2
sw
lw/sw
beq

单周期和多周期数据通路区别

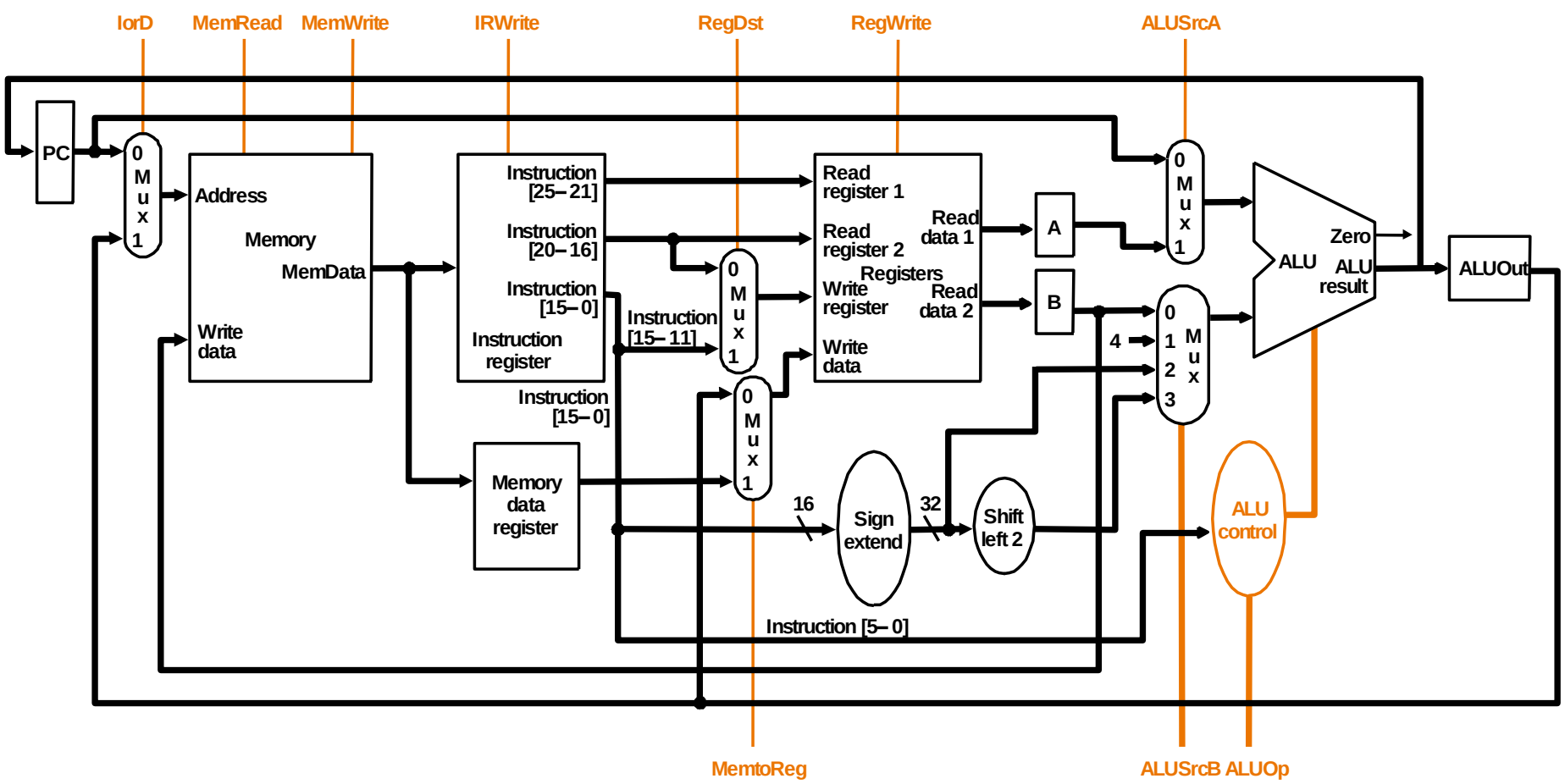


ALU、寄存器、存储器、控制信号



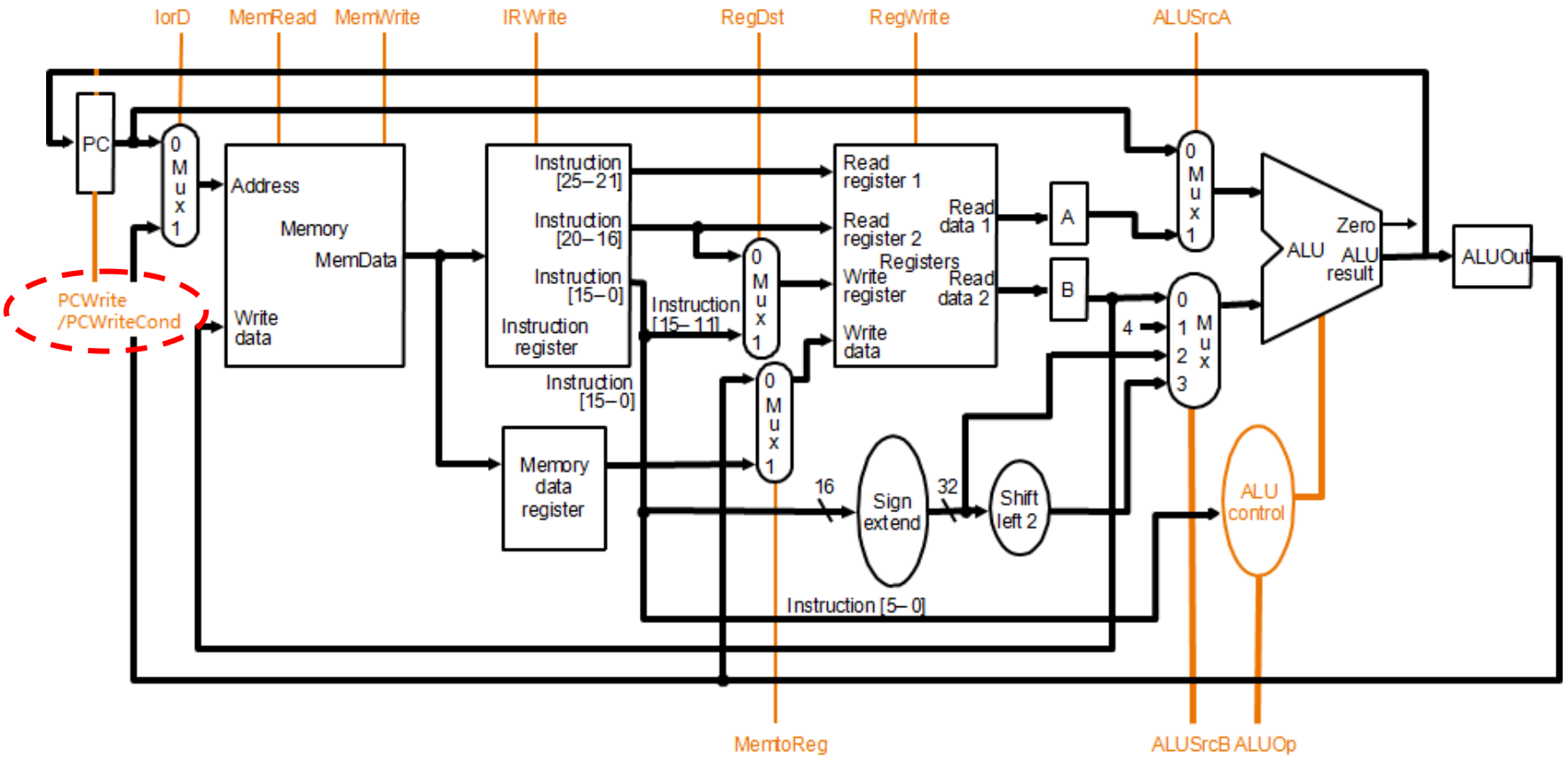
多周期控制信号

op(6 bits)	rs(5 bits)	rt(5 bits)	rd(5 bits)	shamt(5 bits)	funct(6 bits)
op(6 bits)	rs(5 bits)	rt(5 bits)	addr/immediate(16 bits)		
op(6 bits)	addr(26 bits)				



还需要对PC的写控制信号

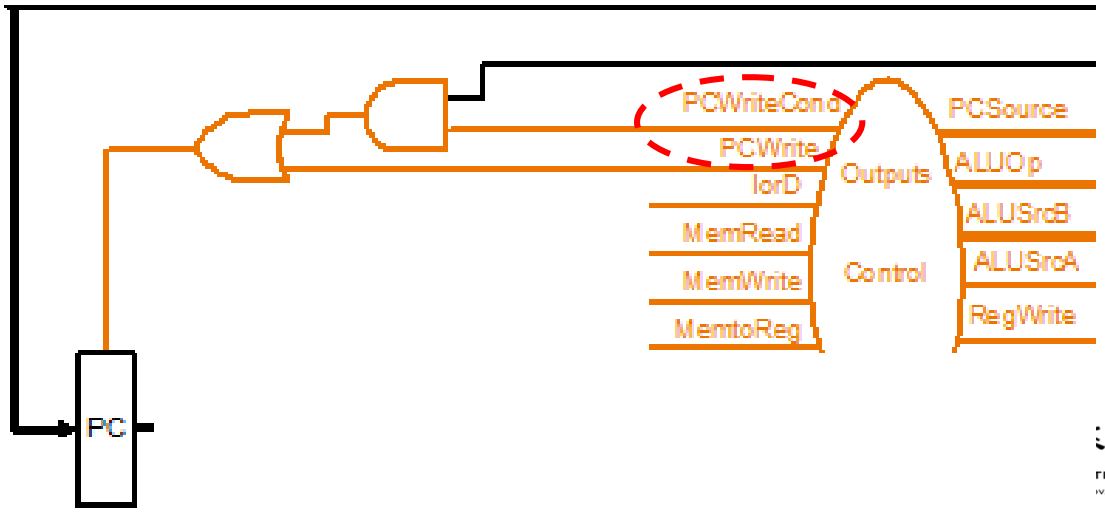
- “在一个指令周期内，PC不能变”



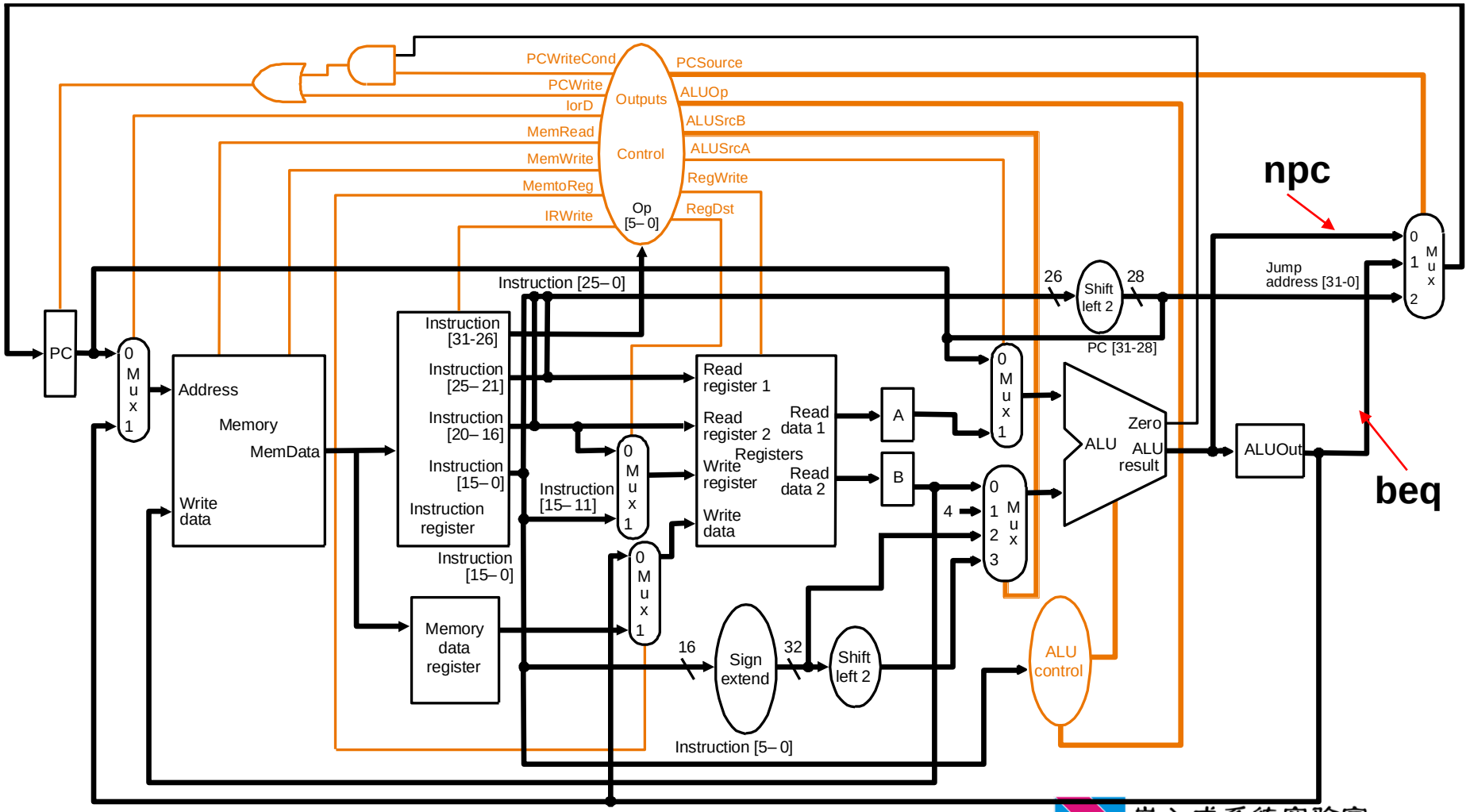
- 思考：为何单周期不需要PC写控制？

PC的写控制

- 3种情况
  - ✓ALU：PC＋4的输出直接存入PC，无条件写（取指阶段）
  - ✓ALU：beq指令的目标地址（译码阶段）
  - ✓ALU：jump指令 J-type（可在译码阶段）
- 需要两个写控制
  - ✓无条件写PCWrite：PC+4, jump
  - ✓有条件写PCWriteCond：beq

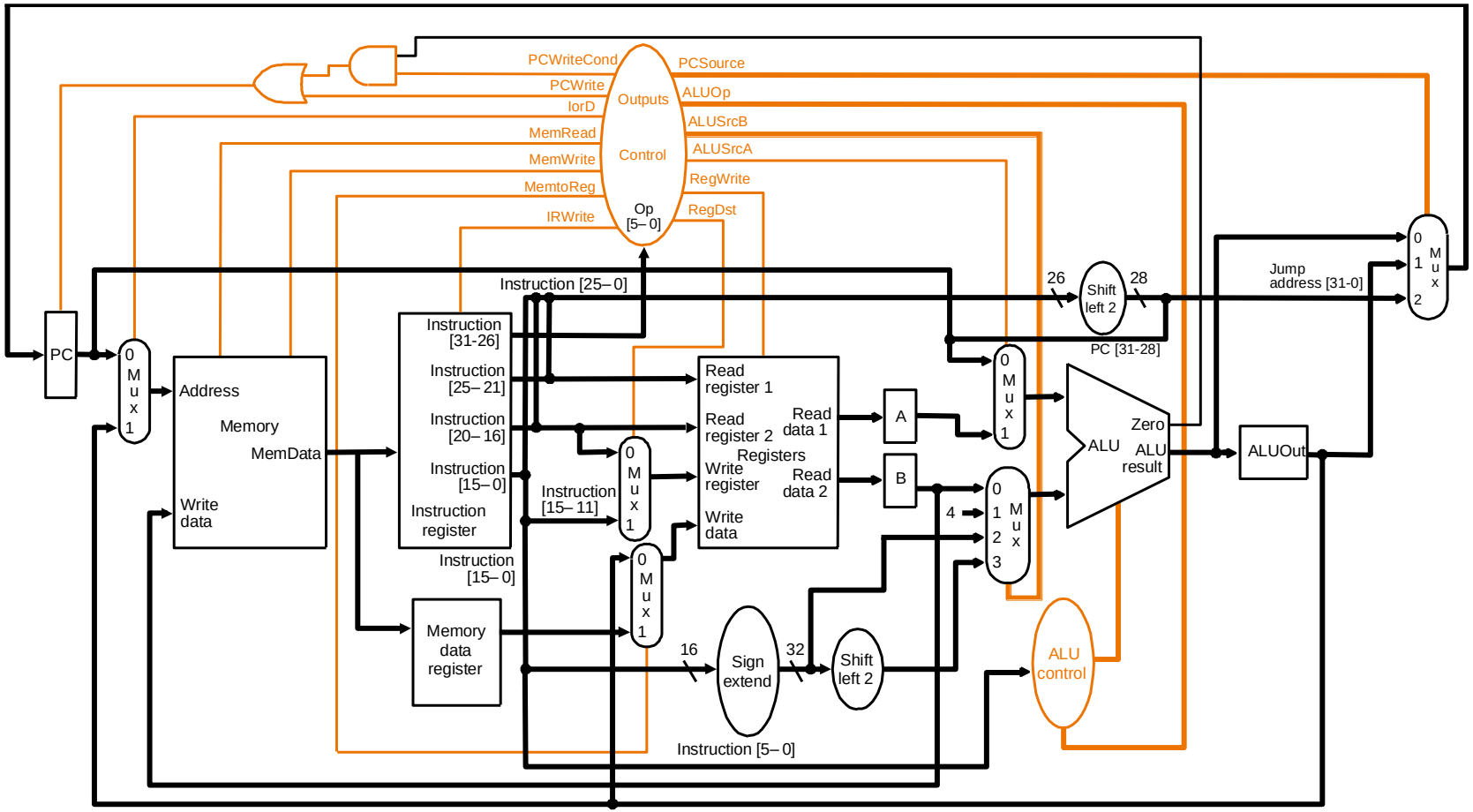


主控制部件


PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op
[5– 0]
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
npc
Jump
address [31-0]
Instruction [25– 0]
26
Shift
left 2
28
PC [31-28]
PC
0
M
u
x
1
Address
Memory
MemData
Write
data
Instruction
[31-26]
Instruction
[25– 21]
Instruction
[20– 16]
Instruction
[15– 0]
Instruction
register
Instruction
[15– 0]
Instruction
[15– 11]
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
A
B
4
ALU
Zero
ALU
result
ALUOut
beq
Memory
data
register
16
Sign
extend
32
Shift
left 2
ALU
control
Instruction [5– 0]

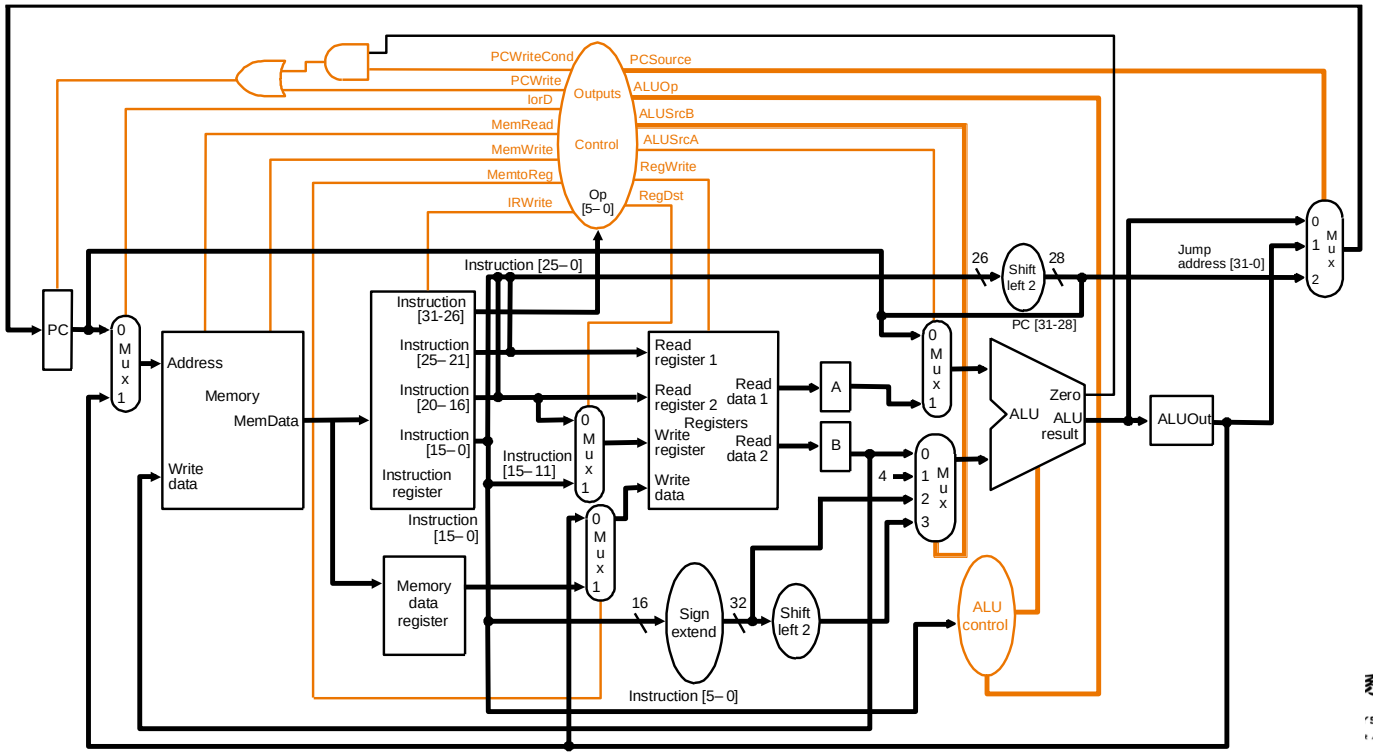
取指阶段

- 取指
  - ✓ 根据PC从MEM中取指，IR=MEM[PC]
  - ✓ 计算NPC，PC=PC+4
  - ✓ 控制信号：MemRead, IRWrite, IorD, ALUSrcA, ALUSrcB, ALUOp, PCWrite,PCSource

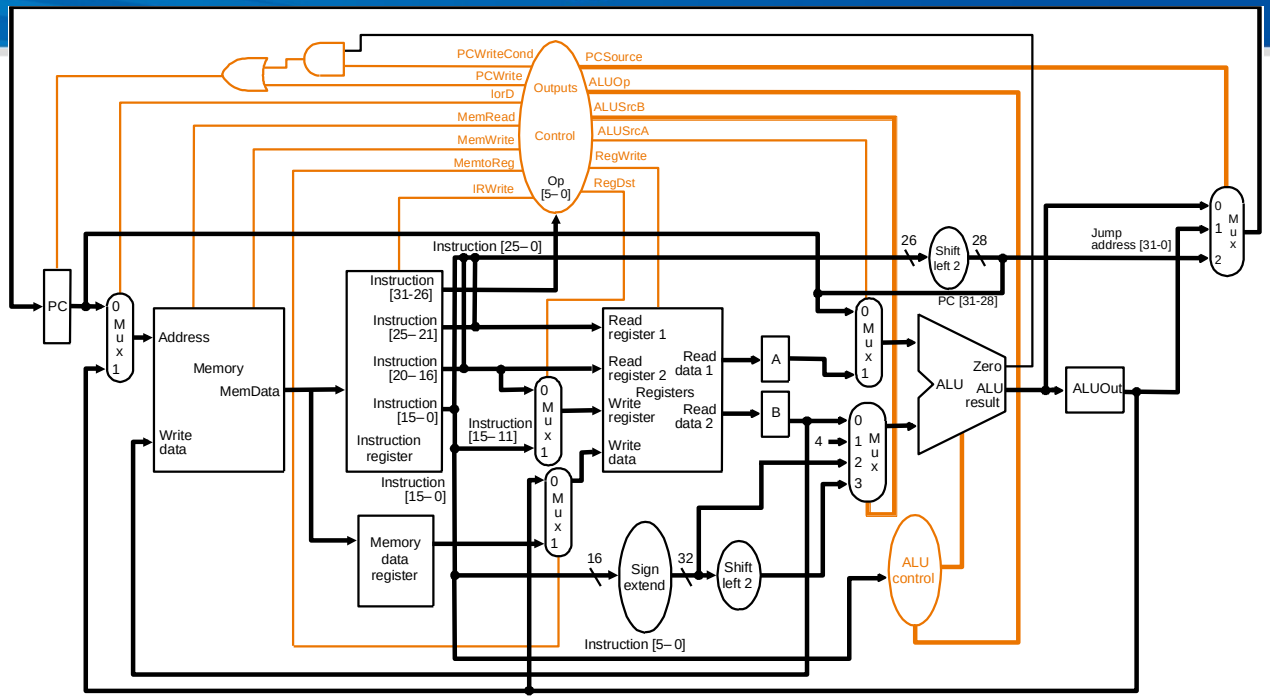


译码阶段

- 指令译码和读寄存器
  - ✓ 将rs和rt送往A和B：A=Reg[IR[25-21]], B=Reg[IR[20-16]]
- 计算beq目标地址
  - ✓ ALUOut=PC+(sign-extend(IR[15-0] <<2)
    - 由于此时尚不知是何指令，所以读寄存器和计算分支地址可能无效
- 控制信号：ALUSrcA, ALUSrcB, ALUOp



R-type执行、访存地址计算、分支完成阶段



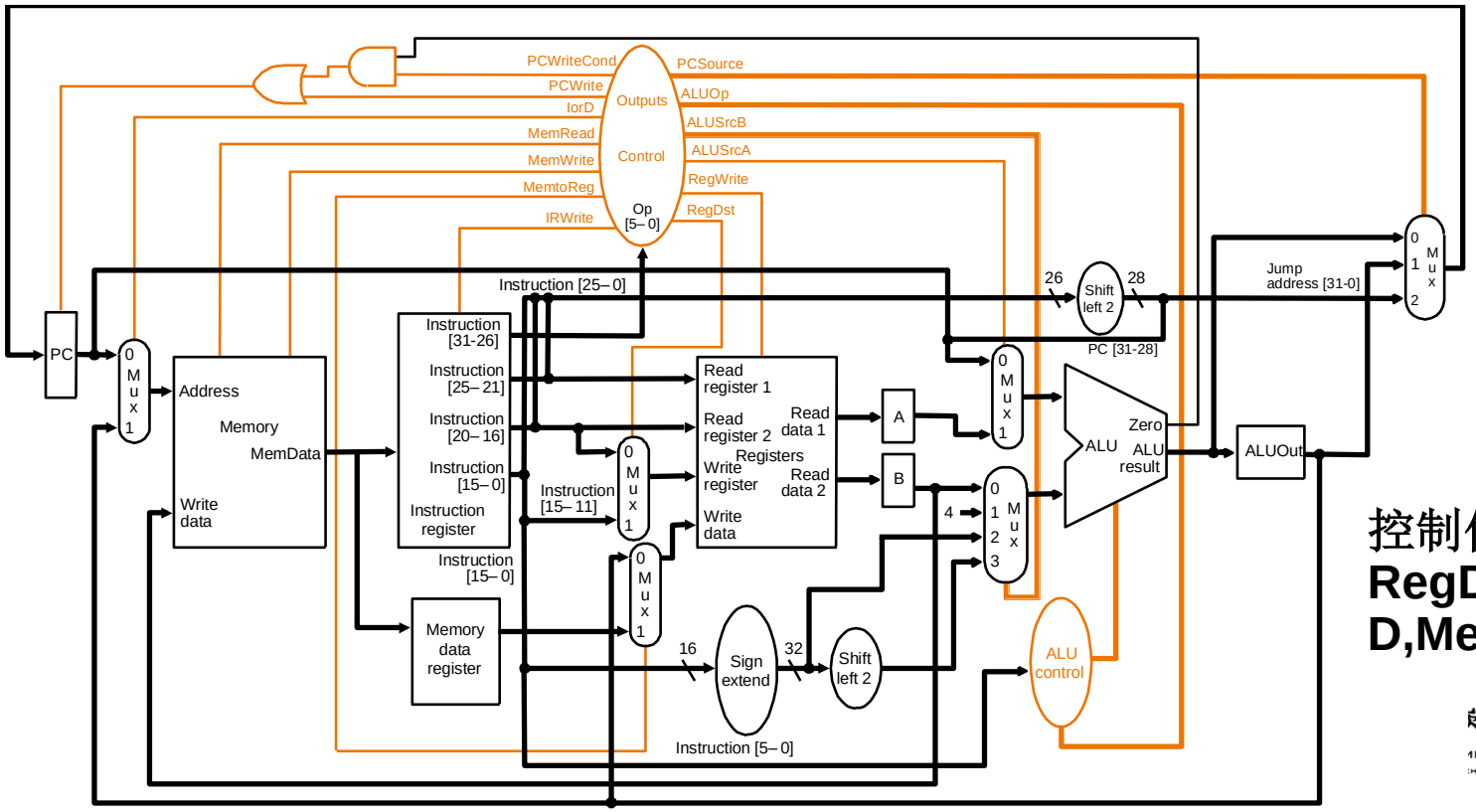
- 依赖于指令类型
  - ✓ R-type指令
    - ALUOut= A op B
  - ✓ 访存指令：计算访存地址
    - ALUOut=A+(sign-extend(IR[15-0])
  - ✓ beq指令
    - if (A == B) PC=ALUOut；

控制信号：**ALUSrcA, ALUSrcB, ALUOp, PCWriteCond,PCSource, PCWrite**

  - ✓ jump指令
    - PC = PC[31-28] || (IR[25-0] << 2)

R-type和sw完成、lw读阶段

- 第四阶段
  - ✓R-type完成：结果写回　Reg[IR[15-11]] = ALUOut
  - ✓sw完成：写入MEM　MEM[ALUOut] = B
  - ✓lw读：　MDR = MEM[ALUOut]

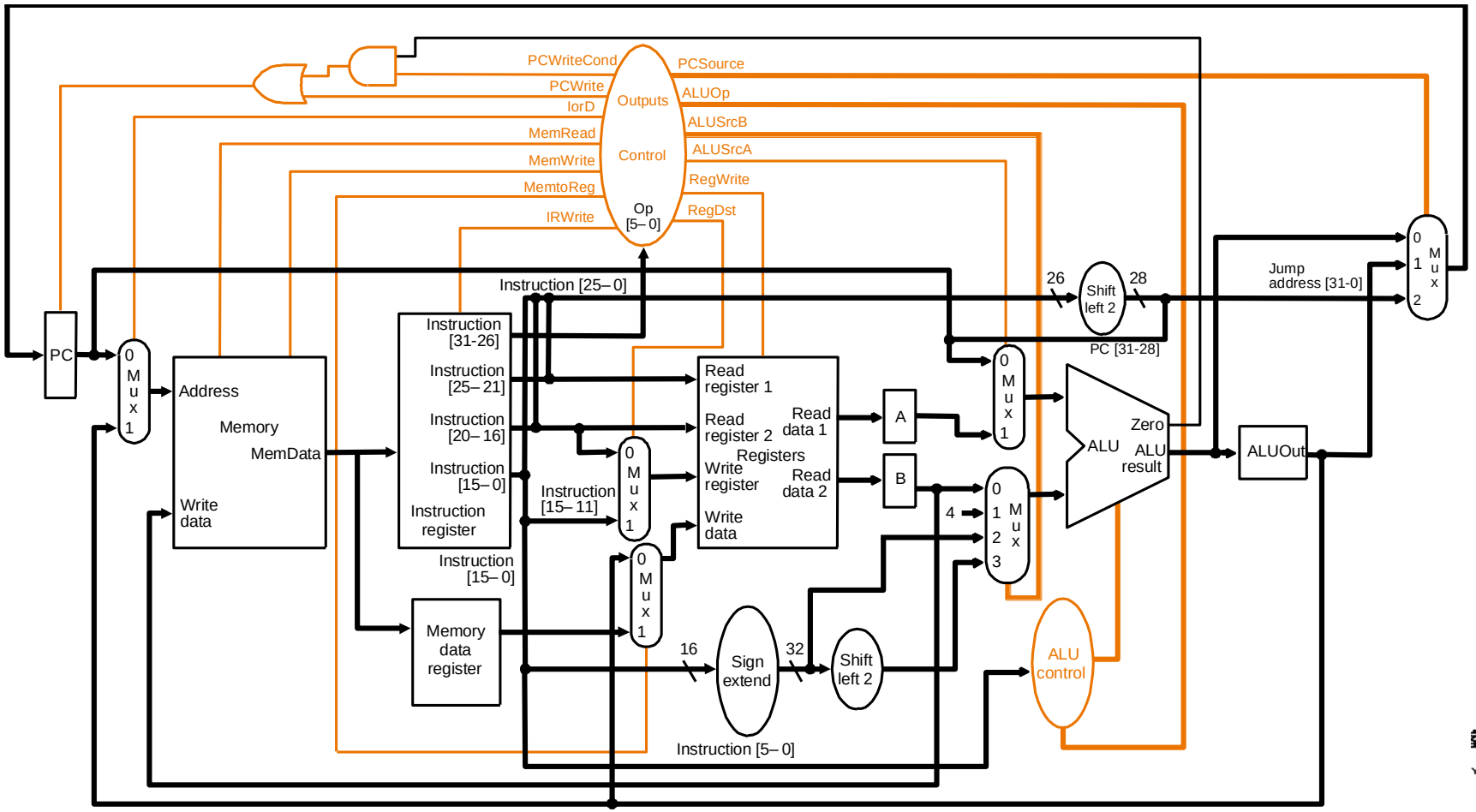


控制信号：RegWrite, RegDst,MemtoReg,IorD,MemRead,MemWrite

lw写阶段

- 第五阶段
  - ✓lw写回：Reg[IR[15-11]] = MDR

控制信号：RegWrite, RegDst,MemtoReg

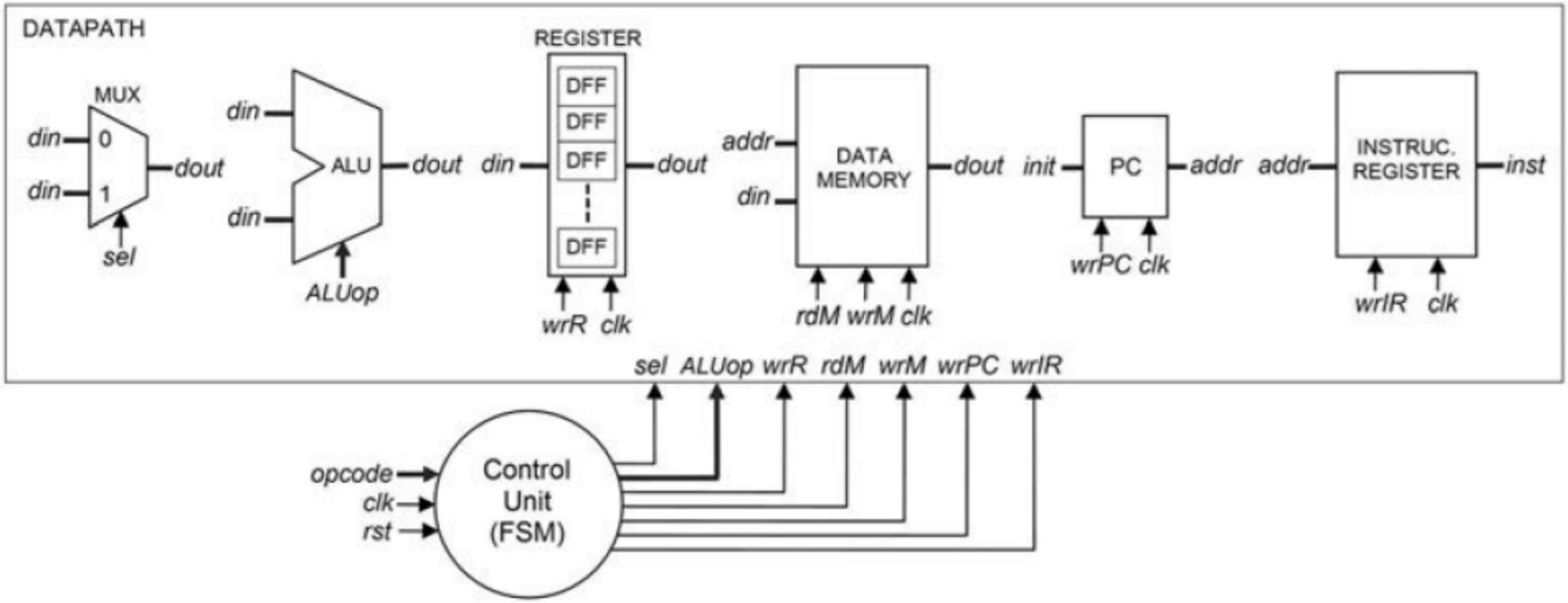


Multicycle RTL

Step	R-Type	lw/sw	beq/bne	j
IF	IR = Mem[PC] PC = PC + 4			
ID	A = Reg[IR[25-21]] B = Reg[IR[20-16]] ALUOut = PC + (SE(IR[15-0]) << 2)			
EX	ALUOut = A op B	ALUOut = A +SE(IR[15-0])	If (A==B) then PC = ALUOut	PC = PC[31-28] || (IR[25-0]<<2
MEM	Reg[IR[15-11]] = ALUOut	MDR=Mem[ALUOut] Mem[ALUOut] = B		
WB		Reg[IR[20-16]] = MDR		

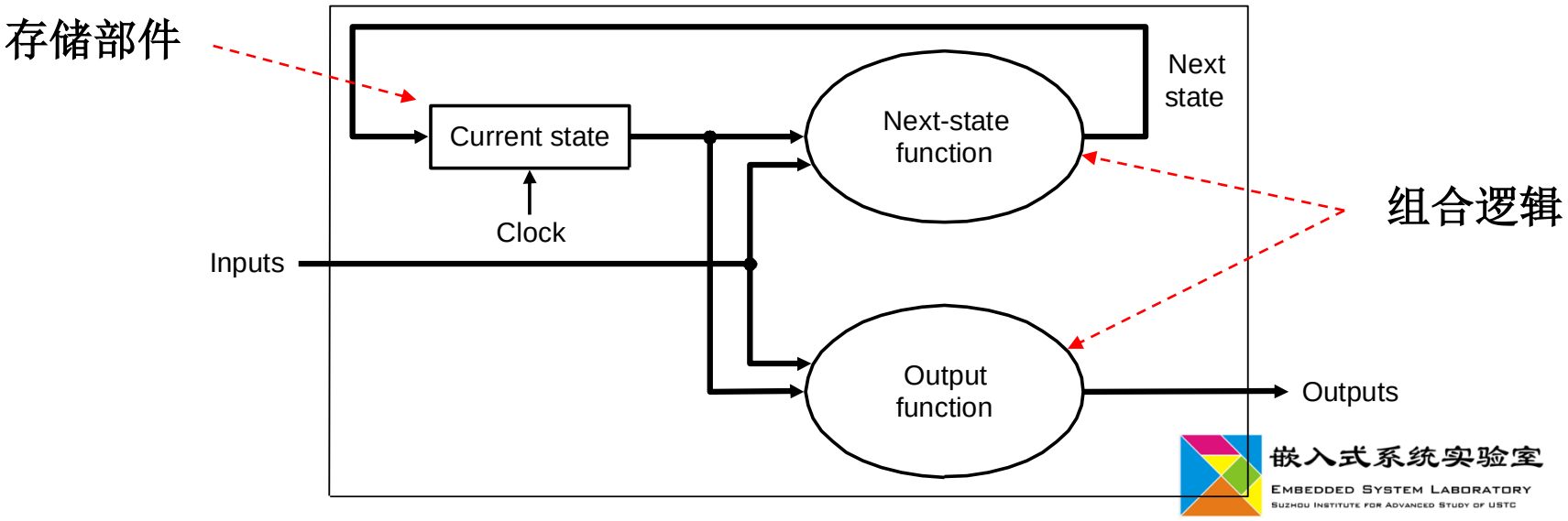
控制信号产生

- FSM: Finite State Machine



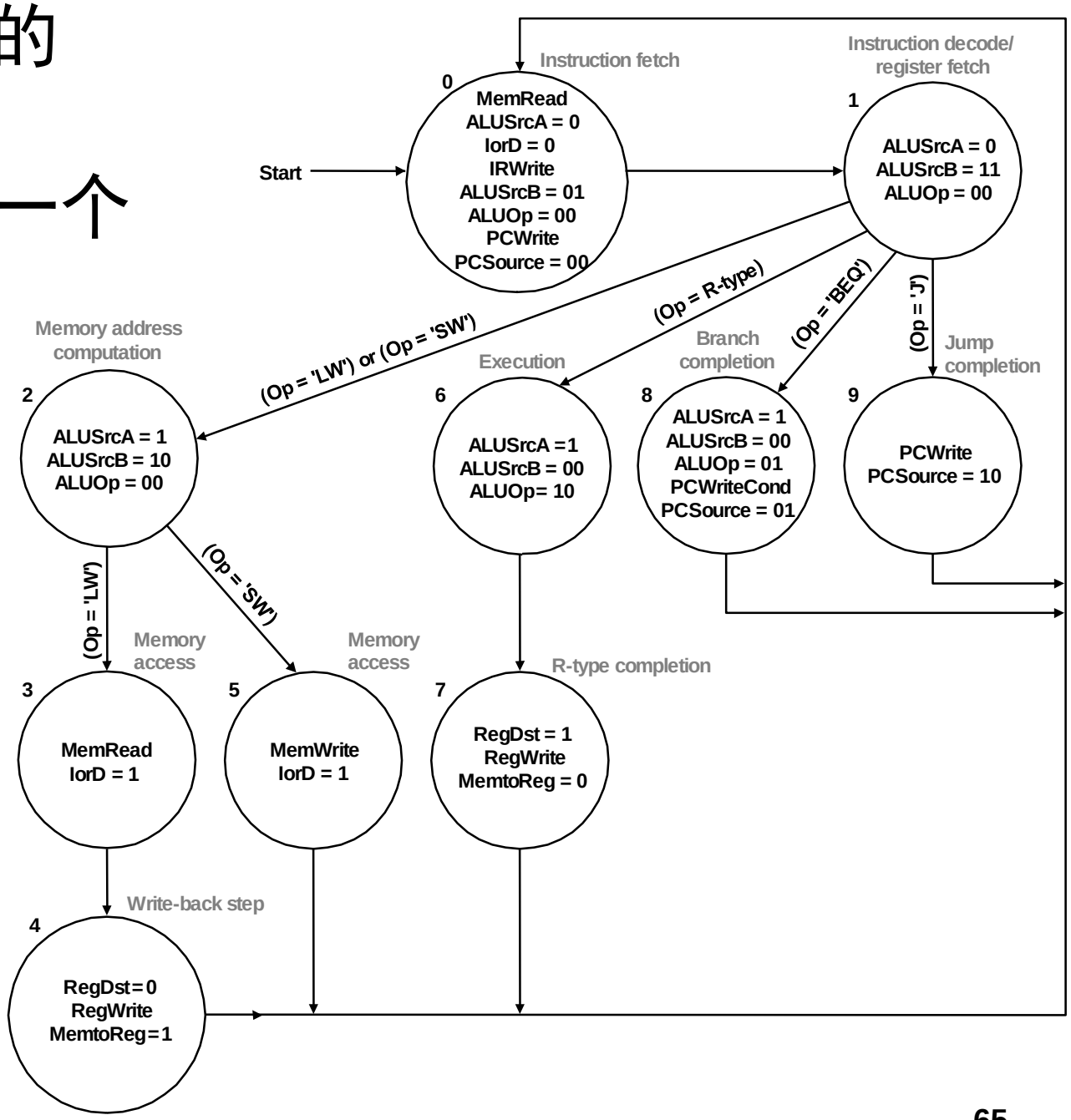
FSM控制部件实现

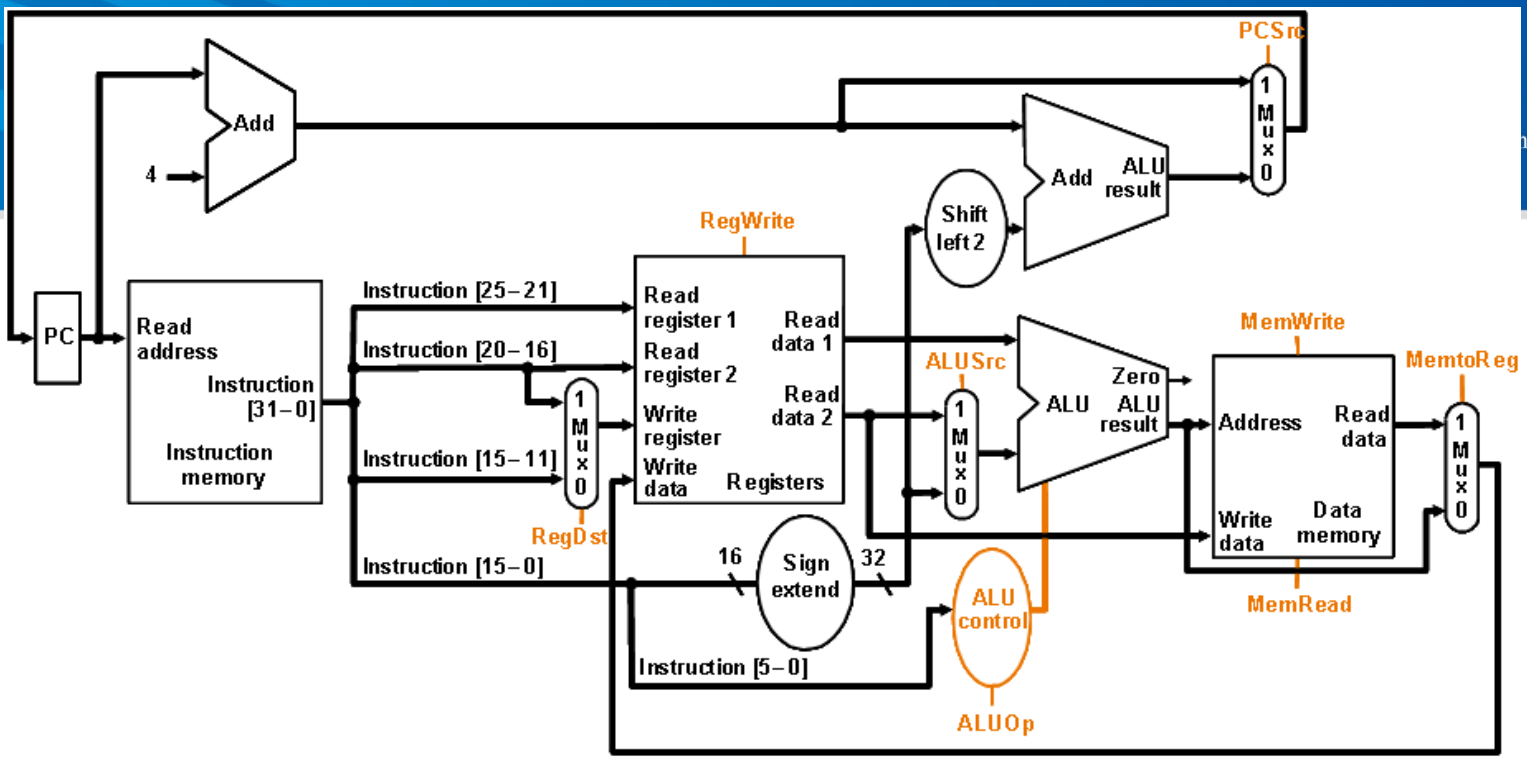
- FSM: Finite State Machine
- Moore型（Edward Moore），Mealy型（George Mealy）
  - ✓ Moore型速度快（输出与输入无关，可以在周期一开始就提供控制信号）,输出与时钟完全同步。
  - ✓ Mealy型电路较小。输出与时钟不完全同步（一个tick内随输入而变）。
  - ✓ 两种状态机可以相互转换。
- EDA工具可以根据FSM自动综合生成控制器

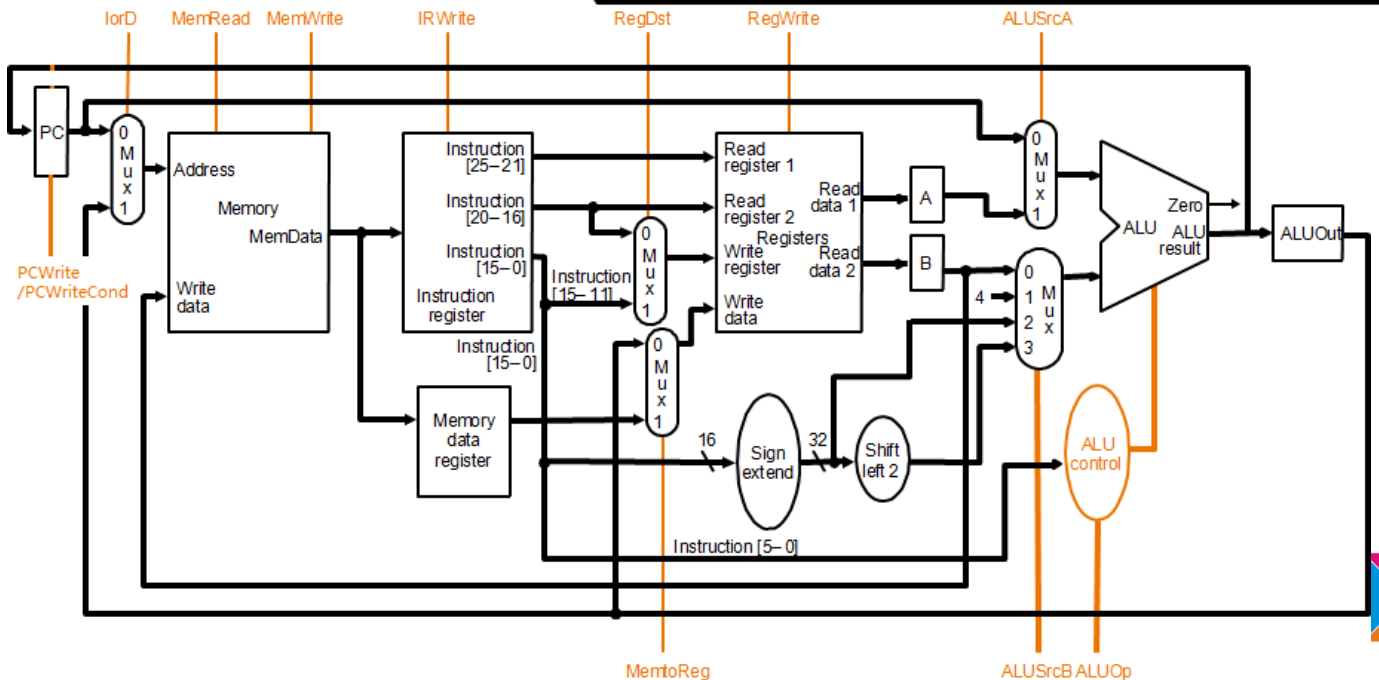


多周期控制器的MooreFSM，每个状态需要一个时钟周期。

- 取指：PC+4
- 译码: BEQ算地址
- 执行
  - ✓ R执行
  - ✓ LW/SW算地址
  - ✓ 分支完成
- 访存
  - ✓ R/SW完成
  - ✓ Load访存
- 写回
  - ✓ Load完成

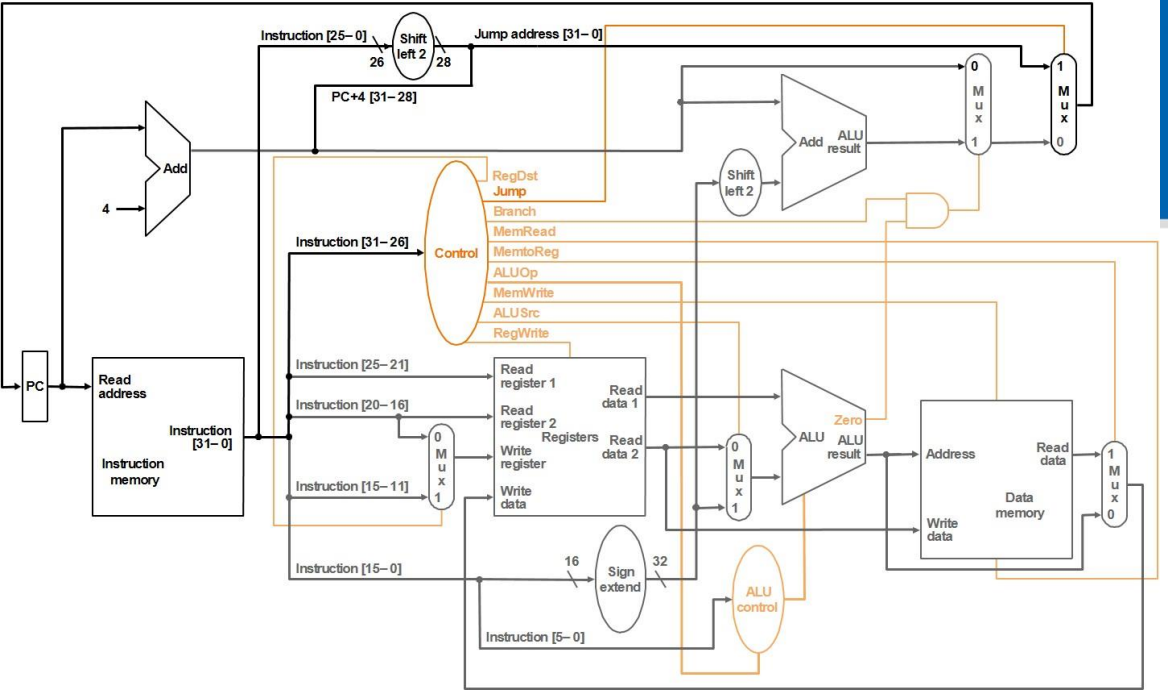






思考：找找两张图中相同及不同之处。

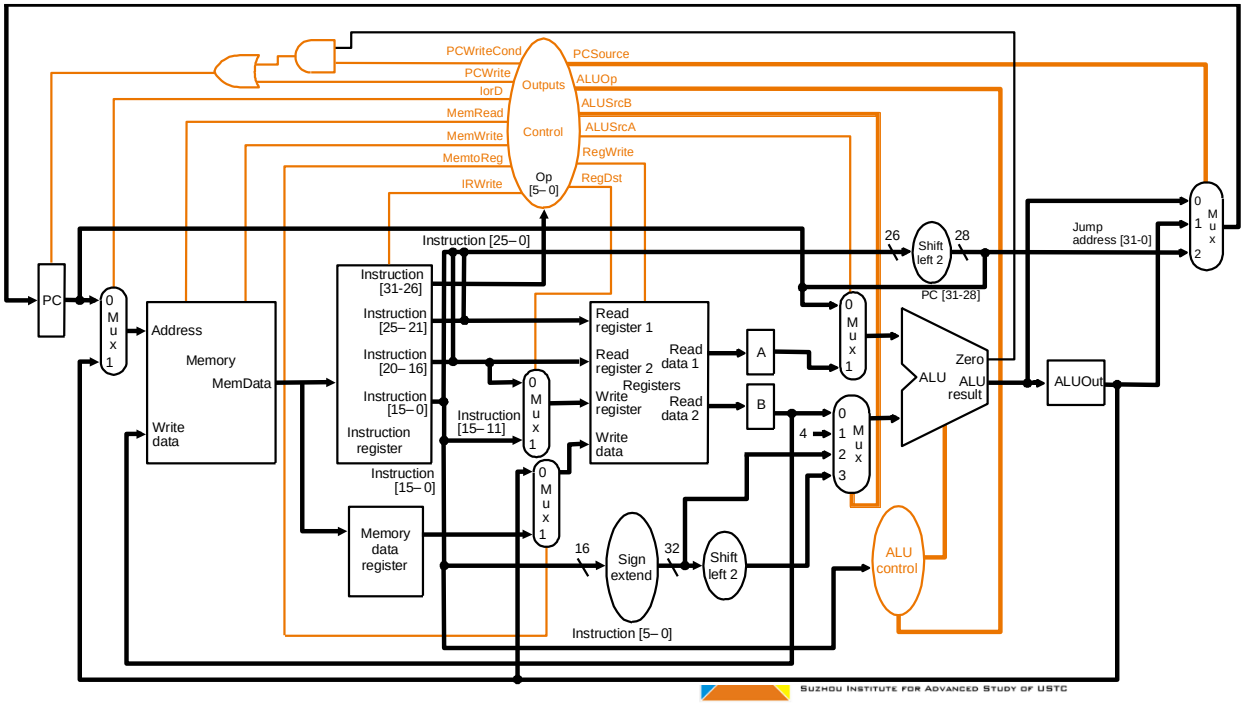
寄存器、存储器、ALU、多选



思考：找找两张图中相同及不同之处。

控制信号

1. 寄存器文件写使能
2. 存储器读写使能
3. PC/IR 寄存器写
4. 多路选择器
5. ALU控制



小结

- 单周期与多周期
  - ✓ 单周期：在一个周期内完成指令的所有操作
    - 周期宽度如何确定,能否在一个clk内完成？
  - ✓ 多周期：一个周期完成指令的一步
    - 与单周期的区别：功能部件复用，中间结果保存
      - 可以实现不定长指令周期，提高性能
    - 按当前周期产生响应的控制信号
      - 何时刷新PC
- 作业：
  1. COD4 4.1，4.9
  2. 分析MIPS三种类型指令的多周期设计方案中每个周期所用到的功能部件。

联系方式

中国科学技术大学计算机学院
嵌入式系统实验室（西活科住物业对面）
高效能智能计算实验室 （中科大苏州研究院）

主要研究方向：
- 基于分布式系统, GPU，FPGA的神经网络、图计算加速
- 人工智能和深度学习（寒武纪）芯片及智能计算机

课程问题、研究方向问题、实验室问题 欢迎大家邮件

cswang@ustc.edu.cn
http://staff.ustc.edu.cn/~cswang