处理器设计

处理器设计

- 能实现几条简单指令的处理器

  算术逻辑运算指令: **add, sub, ori**

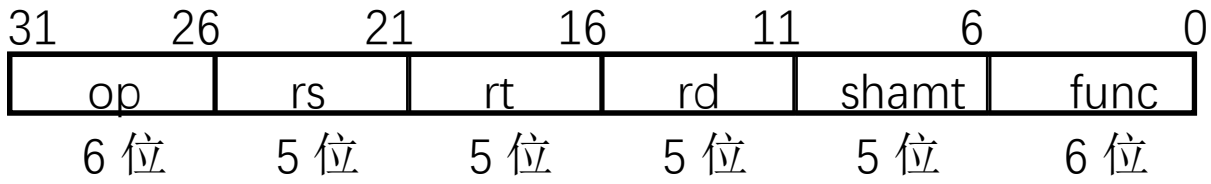
  访问存储器: **lw, sw**

  控制流指令: **beq, j**

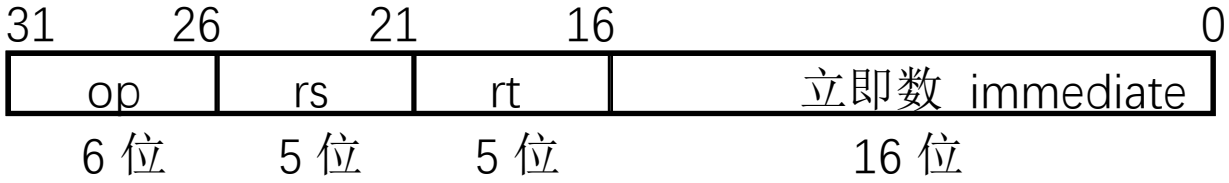
MIPS指令的子集

- 加法与减法
  - add rd, rs, rt
  - sub rd, rs, rt
- OR 立即数:
  - ori rt, rs, imm16
- 读内存 和 写内存
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- 条件转移:
  - beq rs, rt, imm16
- 无条件转移:
  - j target

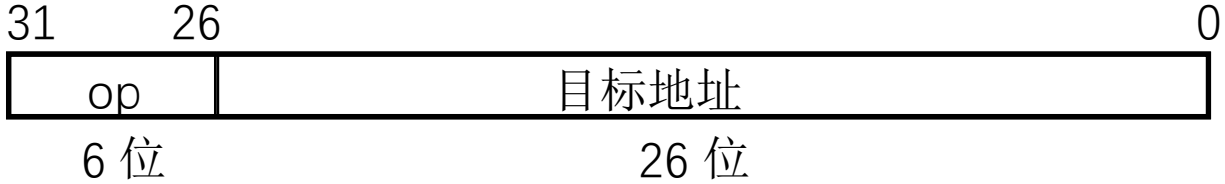
R 型:



I 型:



J 型:



设计处理器的五个步骤

1. 分析指令系统，得出对数据通路的需求
2. 选择数据通路上合适的组件
3. 连接组件构成数据通路
4. 分析每一条指令的实现，以确定控制信号
5. 集成控制信号，完成控制逻辑

分析各条指令的数据通路

指令的执行过程

取指令

指令译码

指令执行

ADD　　R[rd] <– R[rs] + R[rt];　　PC <– PC + 4

ORi　　R[rt] <– R[rs] | zero_ext(Imm16);　　PC <– PC + 4

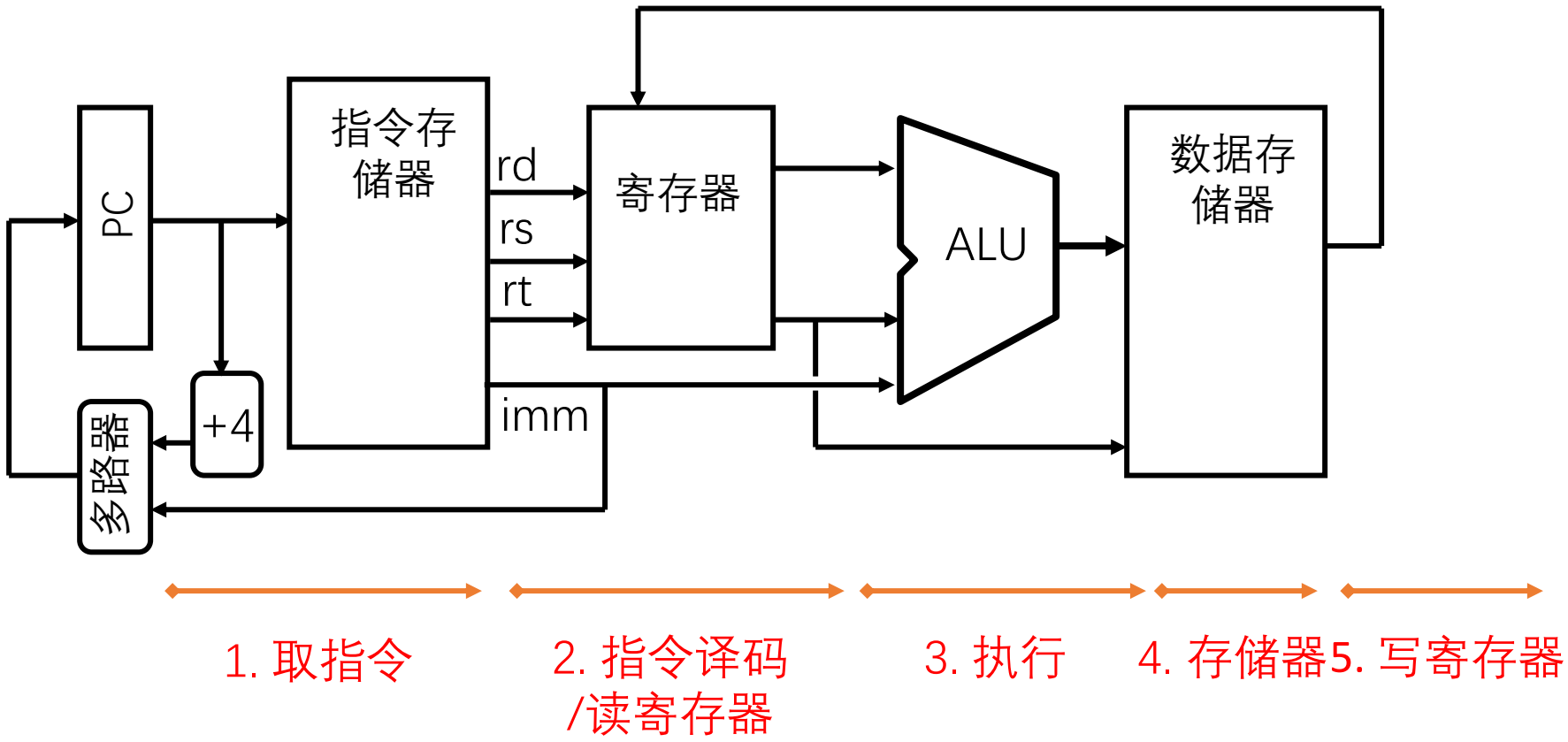
LOAD　　R[rt] <– MEM[R[rs] + sign_ext(Imm16)];　　PC <– PC + 4

STORE　　MEM[R[rs] + sign_ext(Imm16)] <– R[rt];　　PC <– PC + 4

BEQ　　if (R[rs] == R[rt]) then　PC <– PC + 4 +sign_ext(Imm16)] || 00
　　else　PC <– PC + 4

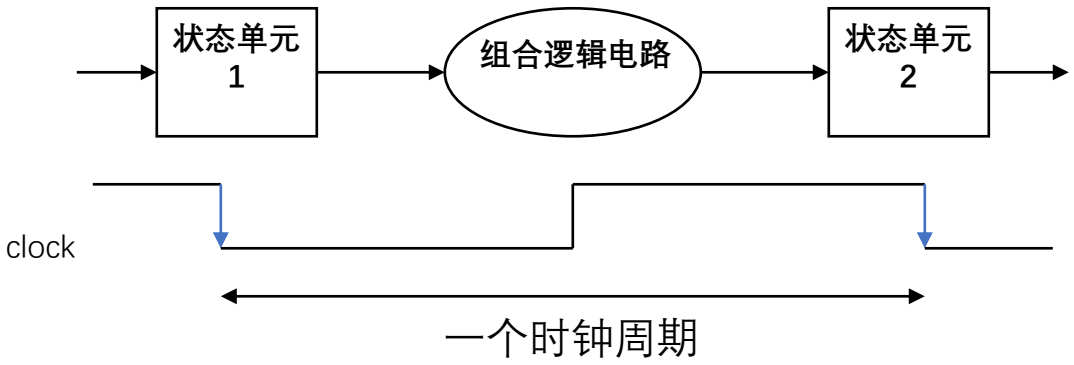
JUMP　　PC <– (PC +4[31-28], I25-0) || 00

数据通路的大致需求


PC
指令存储器
rd
rs
rt
imm
寄存器
ALU
数据存储器
+4
多路器
1. 取指令
2. 指令译码/读寄存器
3. 执行
4. 存储器
5. 写寄存器

单周期处理器

- 单周期处理器：一个时钟周期完成一条指令：



边沿触发

状态转换发生在时钟边沿

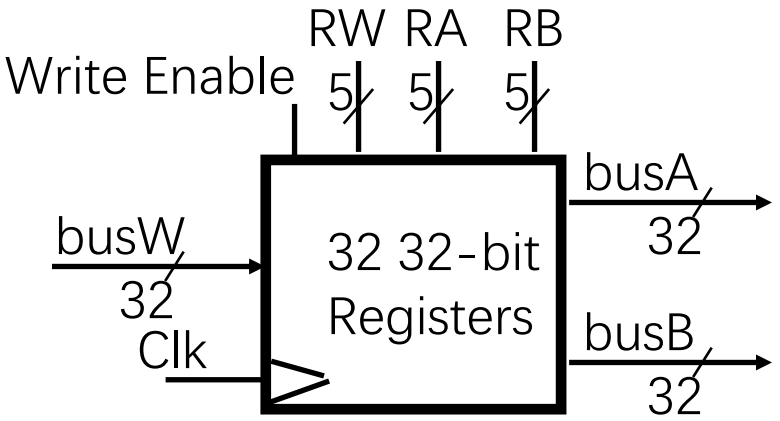
状态单元：

例如：寄存器、存储器

- 读状态单元的内容 -> 通过组合逻辑电路实现指令的功能 -> 将结果写入一个或多个状态单元
- 状态单元每一周期更新一次，是否更新，需要一个显式的控制信号
- 寄存器、存储器：在时钟边沿来到、写允许信号有效时才更新状态

选择数据通路上合适的组件

状态单元：寄存器文件

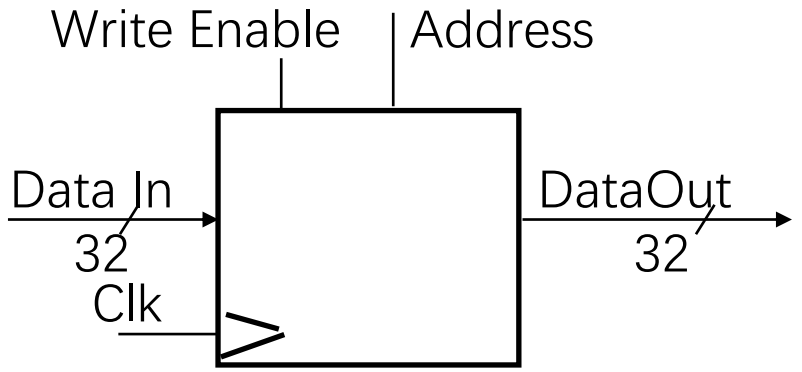


- 时钟输入(CLK)
  - 改变寄存器状态需要时钟边沿触发
- 32个寄存器
  - 两个32位输出：busA 、 busB
  - 一个32位输入： busW
- 寄存器选通
  - RA（5位）：选通RA指定的寄存器
  - RB （5位）：选通RB 指定的寄存器
    - 读操作，看做一个组合电路模块的实现
    - RA 、RB 有效 => busA 、 busB 有效
  - RW（5位）： 选通Rw指定的寄存器
    - 写操作： CLK边沿触发
    - 当Write Enable 为1时，将busW 端口上的数据写入Rw指定的寄存器

选择数据通路上合适的组件

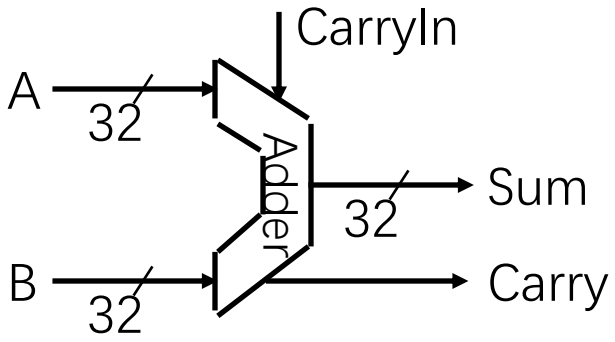
状态单元：存储器

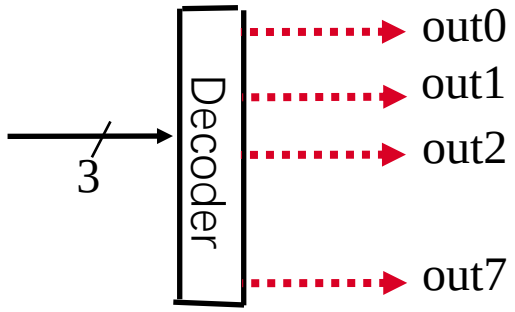
- 时钟输入 (CLK)
  - 改变存储器状态需要时钟边沿触发
- 存储器总线
  - 32位数据输入总线: Data In
  - 32位数据输出总线: Data Out
- 读写操作
  - 读操作，看做一个组合电路模块的实现
    - 一定时间内完成从“地址信号有效” （Address） =>“数据输出” Data Out
  - 写操作：时序电路
    - CLK边沿触发
    - Write Enable = 1: 将Data In的输入写入Address选中的那个字

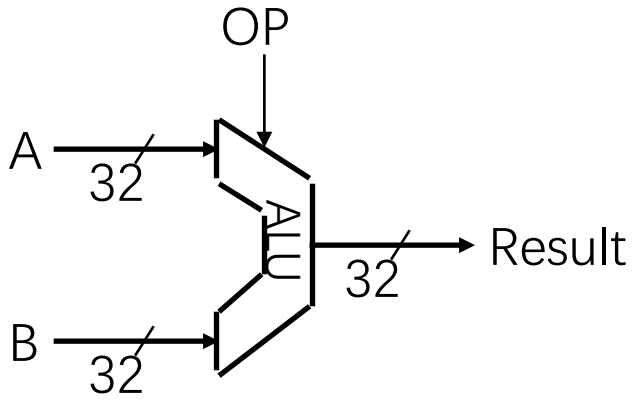


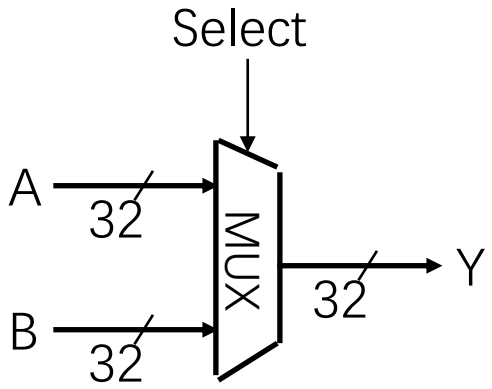
选择数据通路上合适的组件

- 组合逻辑单元









连接组件构成数据通路

（1）取指令

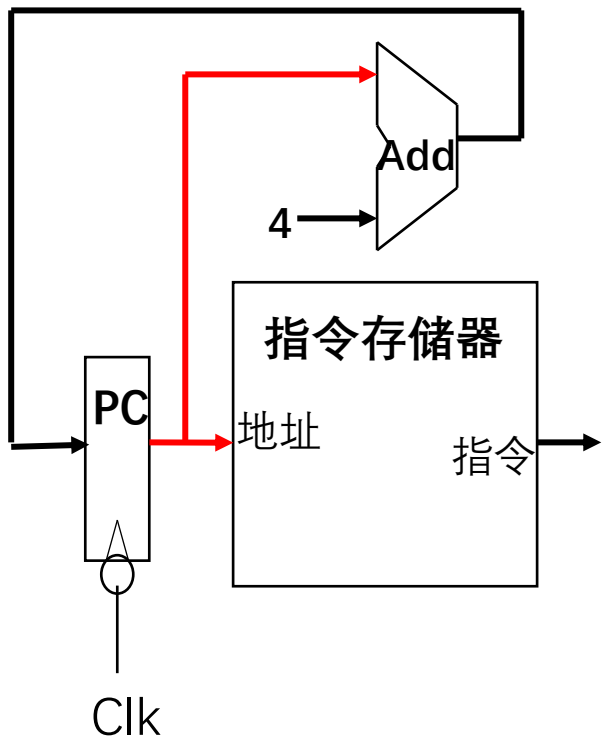
读指令存储器是一个组合电路实现

取指令步骤：

- 从指令存储器读指令
- 将PC值更新为顺序执行的下一条指令的地址
  - PC ← PC + 4

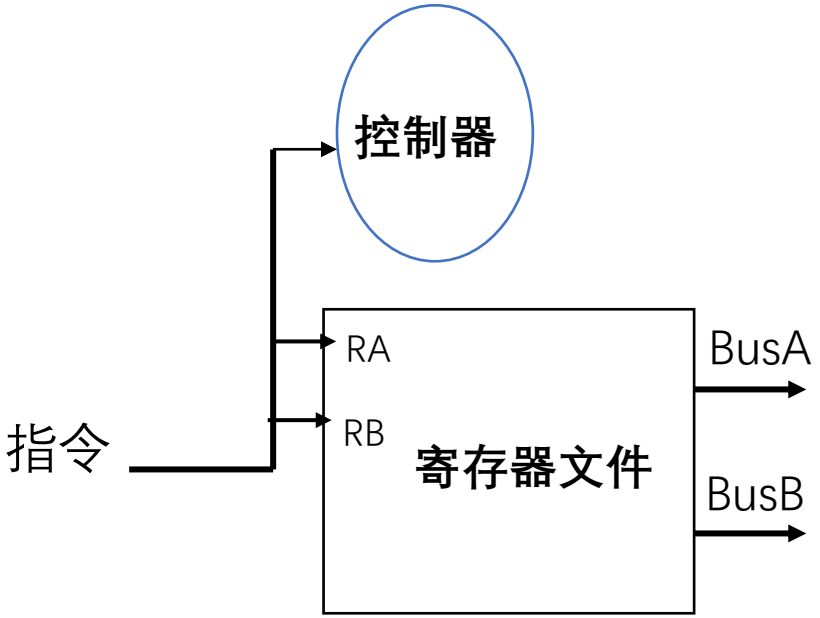
PC的状态更新是时序电路：

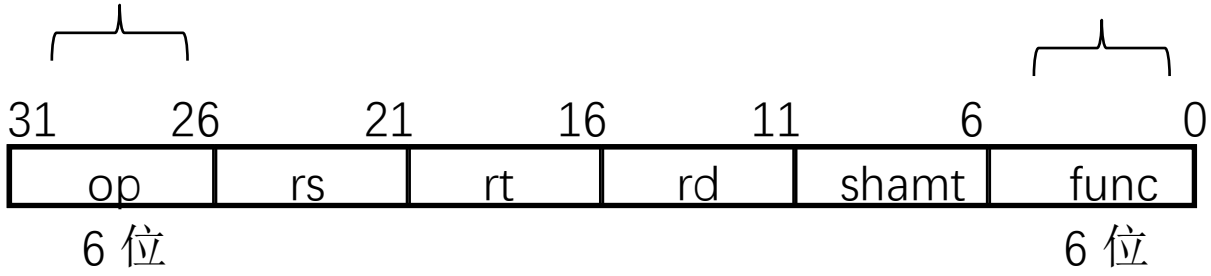
- 需要CLK边沿控制
- 每个时钟周期更新一次，



连接组件构成数据通路

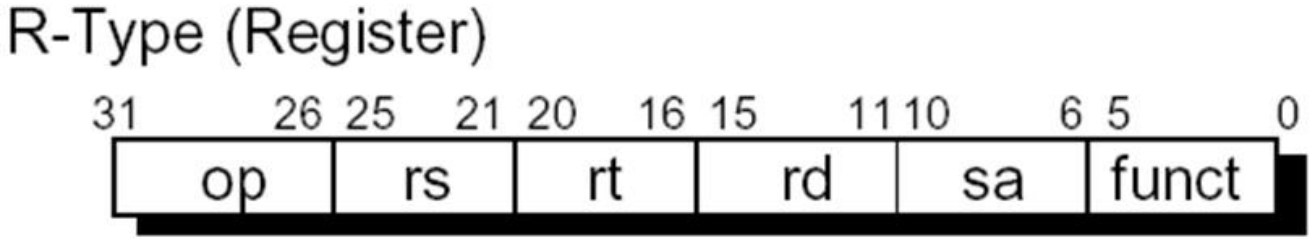
- （2）指令译码
  - 将指令的操作码（op)和功能(func)码作为控制器的输入端
  - 读寄存器
    - 指令中的寄存器地址连接到RA和RB
    - 从寄存器文件读，输出到BusA和BusB

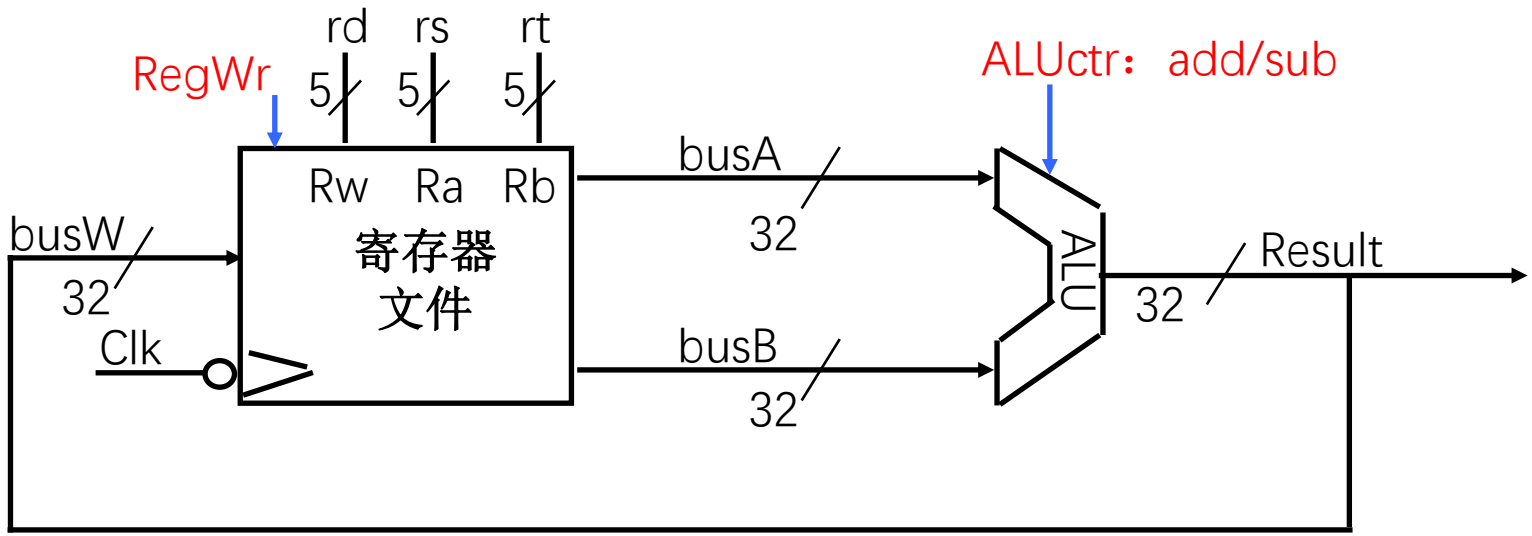




R型指令的数据通路

- add　rd, rs, rt
- R[rd] ← R[rs] + R[rt]



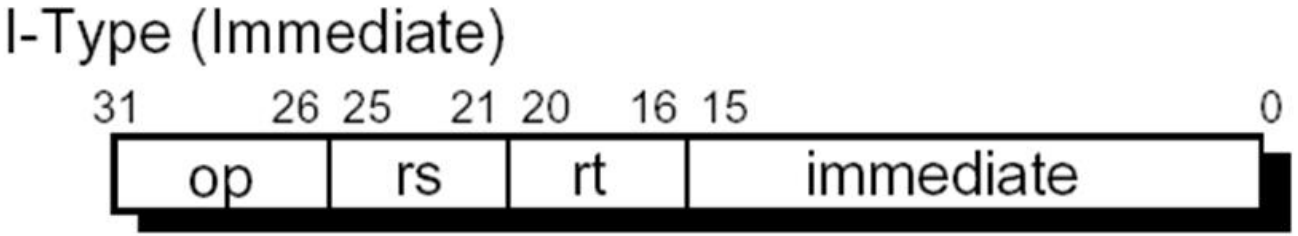


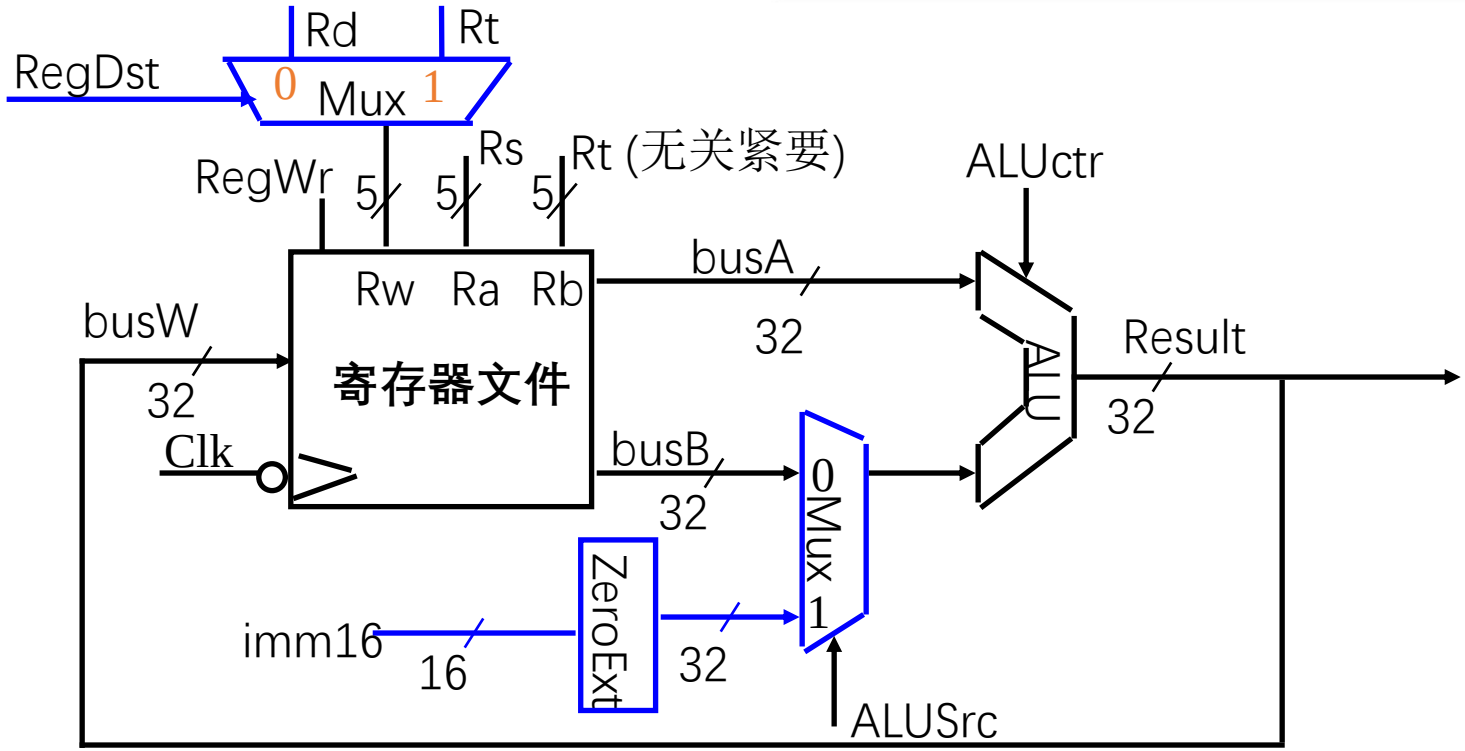
Ra, Rb, Rw 对应 rs, rt, rd

控制信号：ALUctr=add，RegWr=1

Ori 数据通路

- ori　rt, rs, imm16
- R[rt] ← R[rs] or ZeroExt[imm16]]

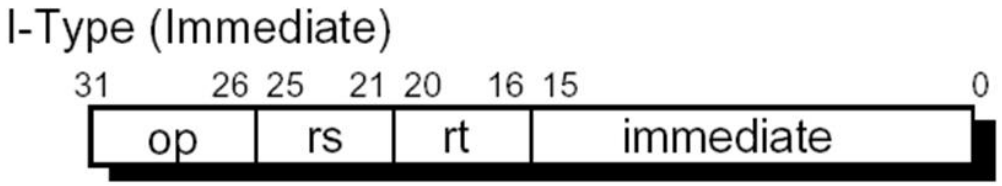


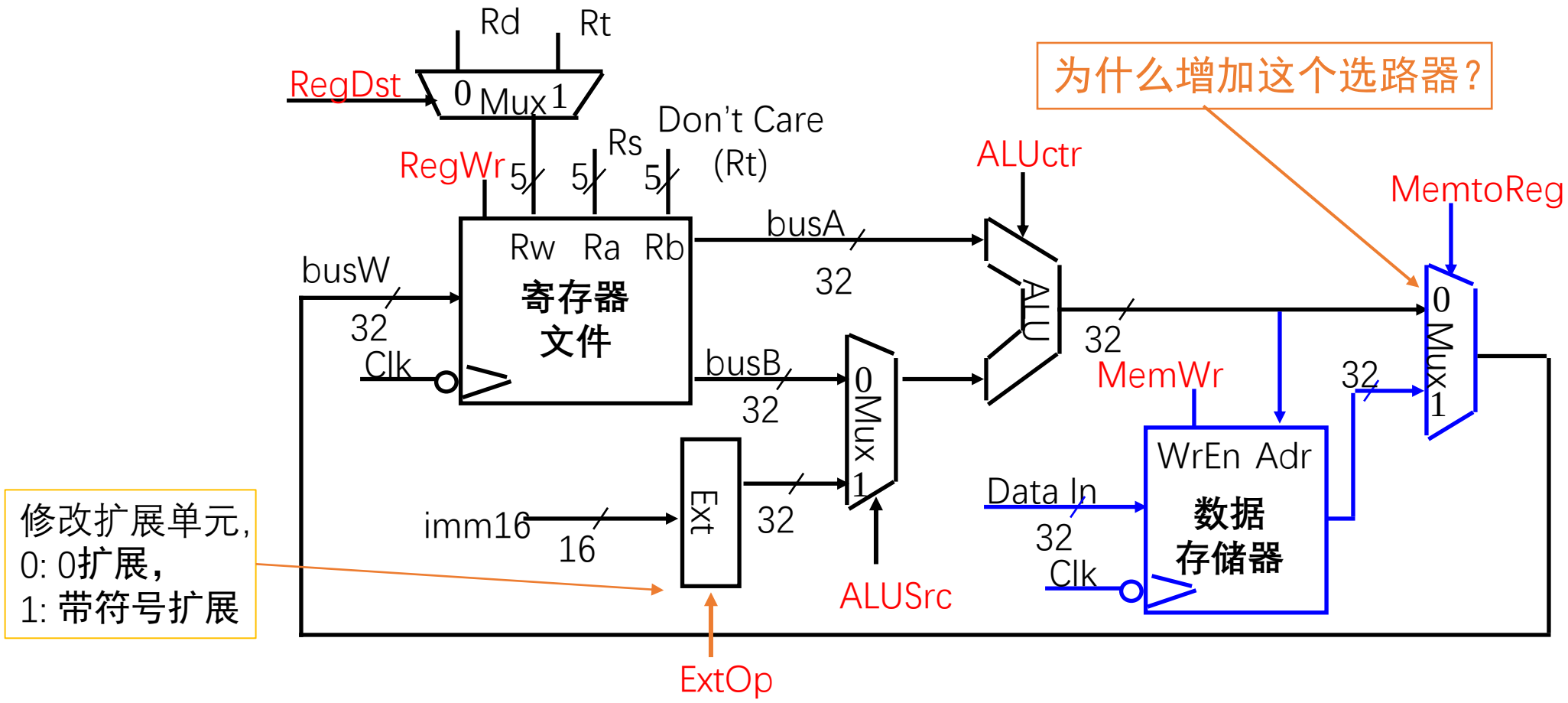


控制信号： ALUSrc=1; ALUctr=or ; RegWr=1 ; RegDst=1

Lw 数据通路

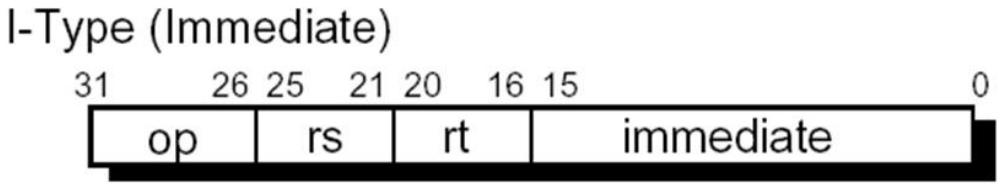
- lw rt, rs, imm16 R[rt] ← M[R[rs] + SignExt[imm16]]



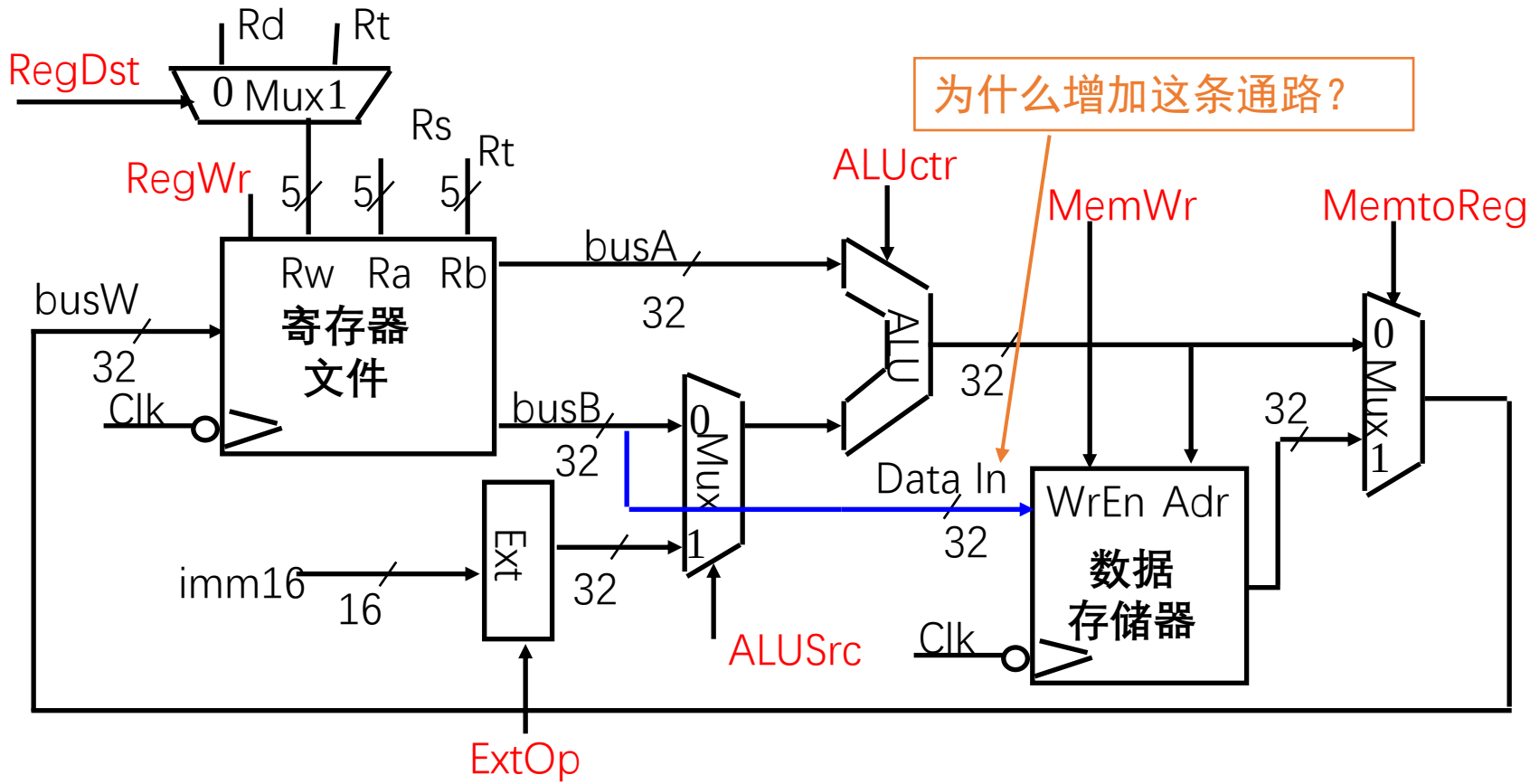


ALUctr=add, ALUSrc=1, ExtOp=1, , MemWr=0, RegDst=1, RegWr=1, MemtoReg=1

Sw 数据通路



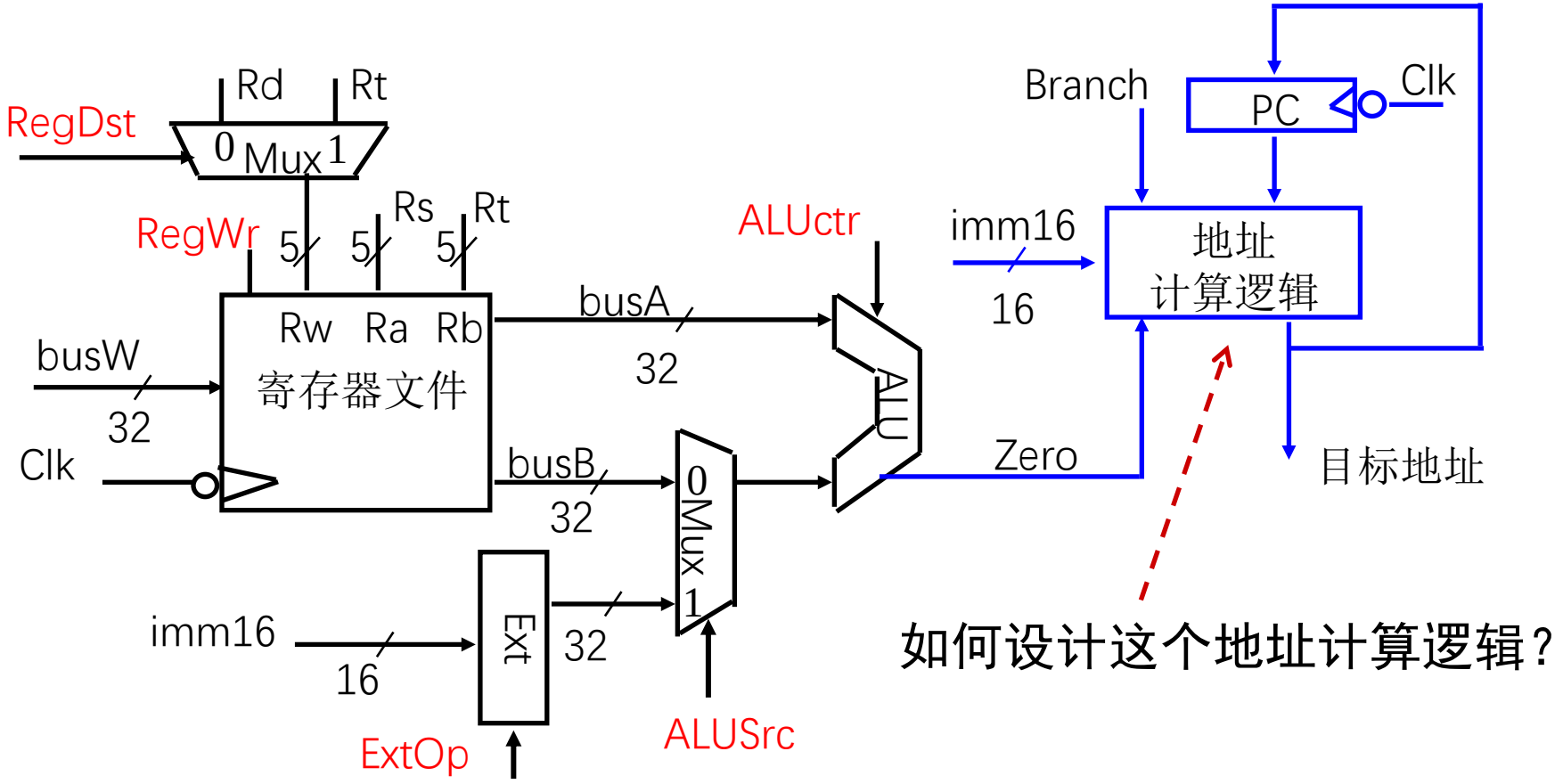
- :sw　rt, rs, imm16 ：　M[R[rs] + SignExt[imm16]] ← R[rt]



ALUctr=add, ALUSrc=1, ExtOp=1, MemWr=1, MemtoReg=x , RegDst=x, RegWr=0

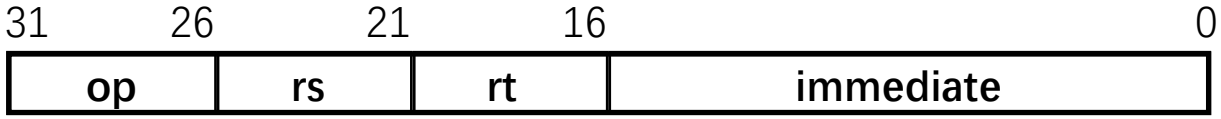
Beq 数据通路

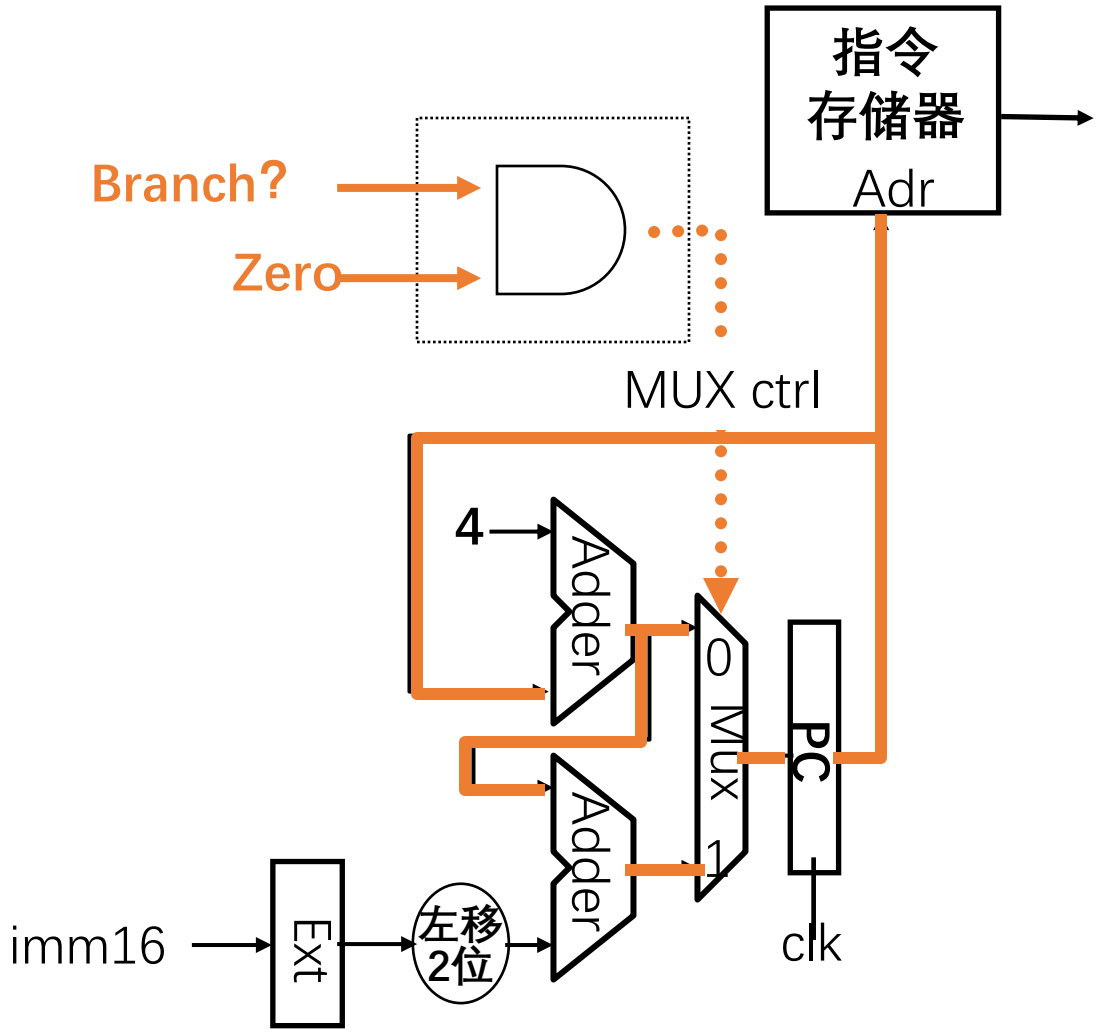
- beq　rs, rt, imm16
- **if (R[rs] == R[rt]) then PC <– PC + sign_ext(Imm16)] || 00 else PC <– PC + 4**



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

Instruction Fetch Unit at the End of Branch





if (Zero == 1) and (Branch ==1)
then PC = PC + 4 + SignExt[imm16]*4 ;
else PC = PC + 4

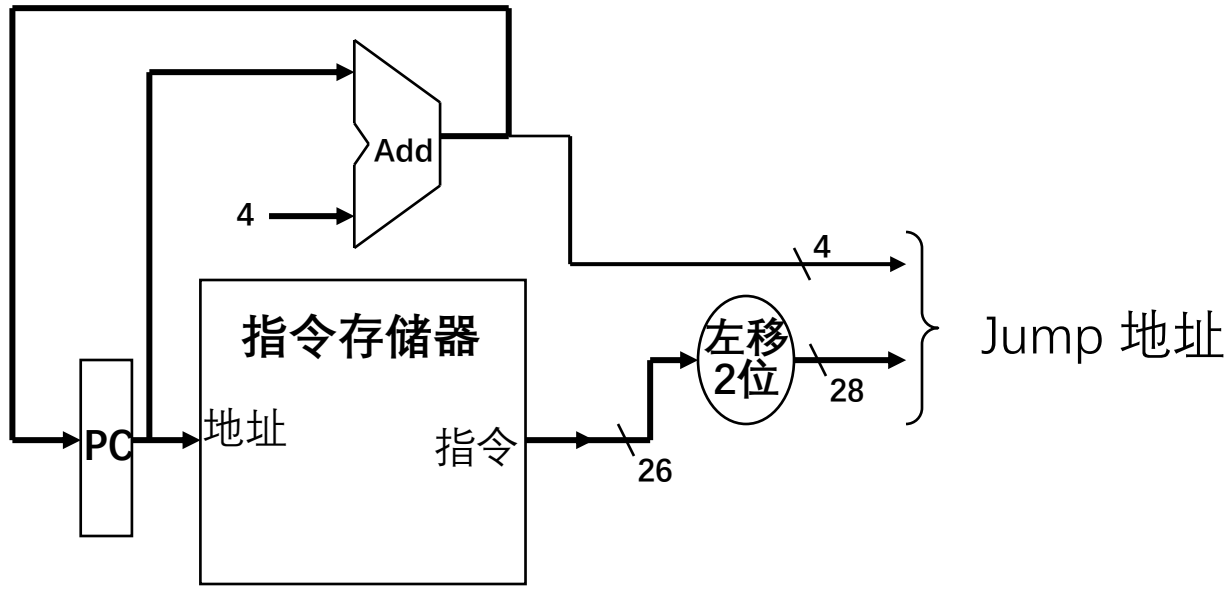
- MUX_ctrl （选路器控制逻辑）

Branch?	zero?	MUX
0	x	0
1	0	0
1	1	1

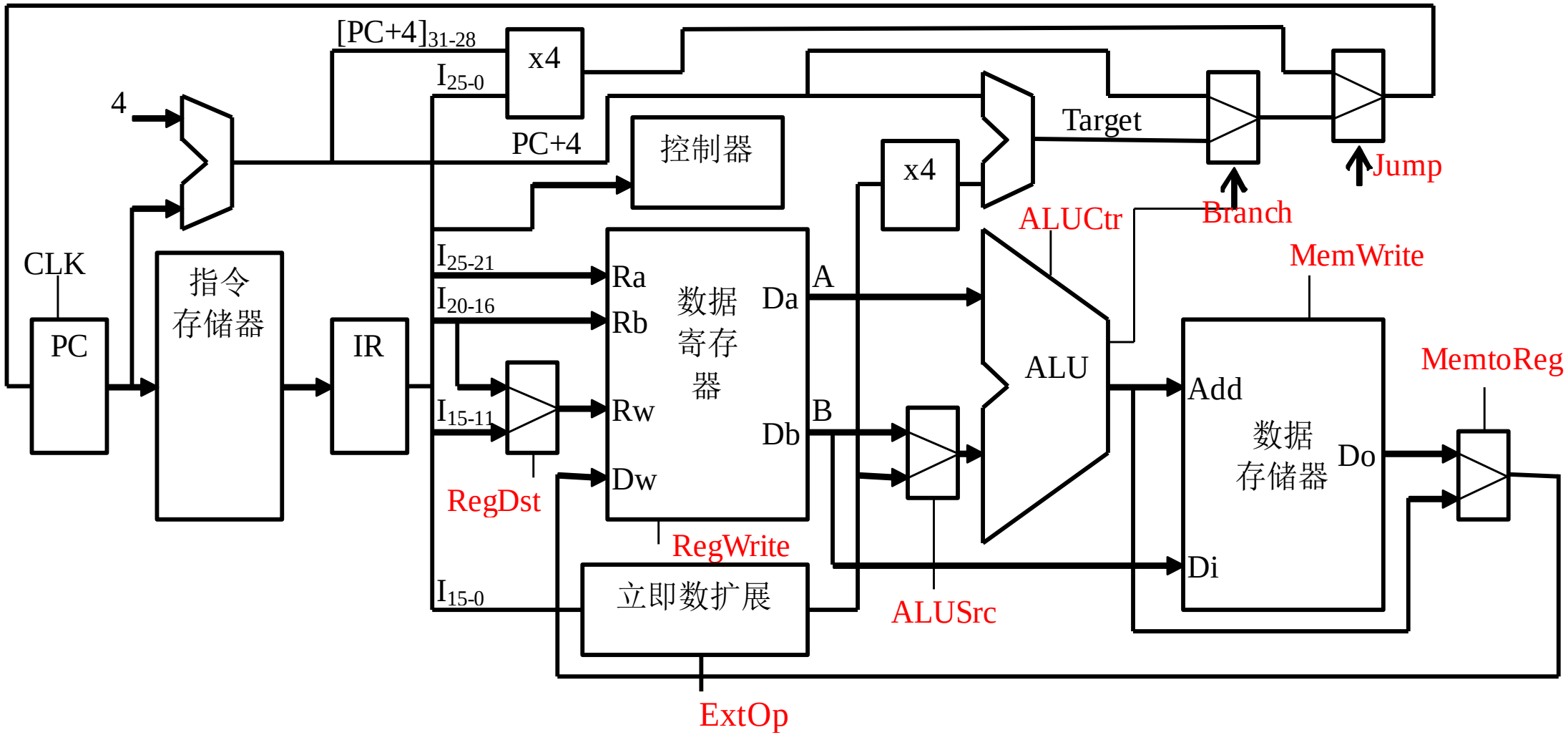
Jump 指令

0x02	26 位地址

- **JUMP： PC <– (PC +4[31-28], $I_{25\text{-}0}$) || 00**
- 指令中的最后26位左移2位后，与PC+4的前4位拼接



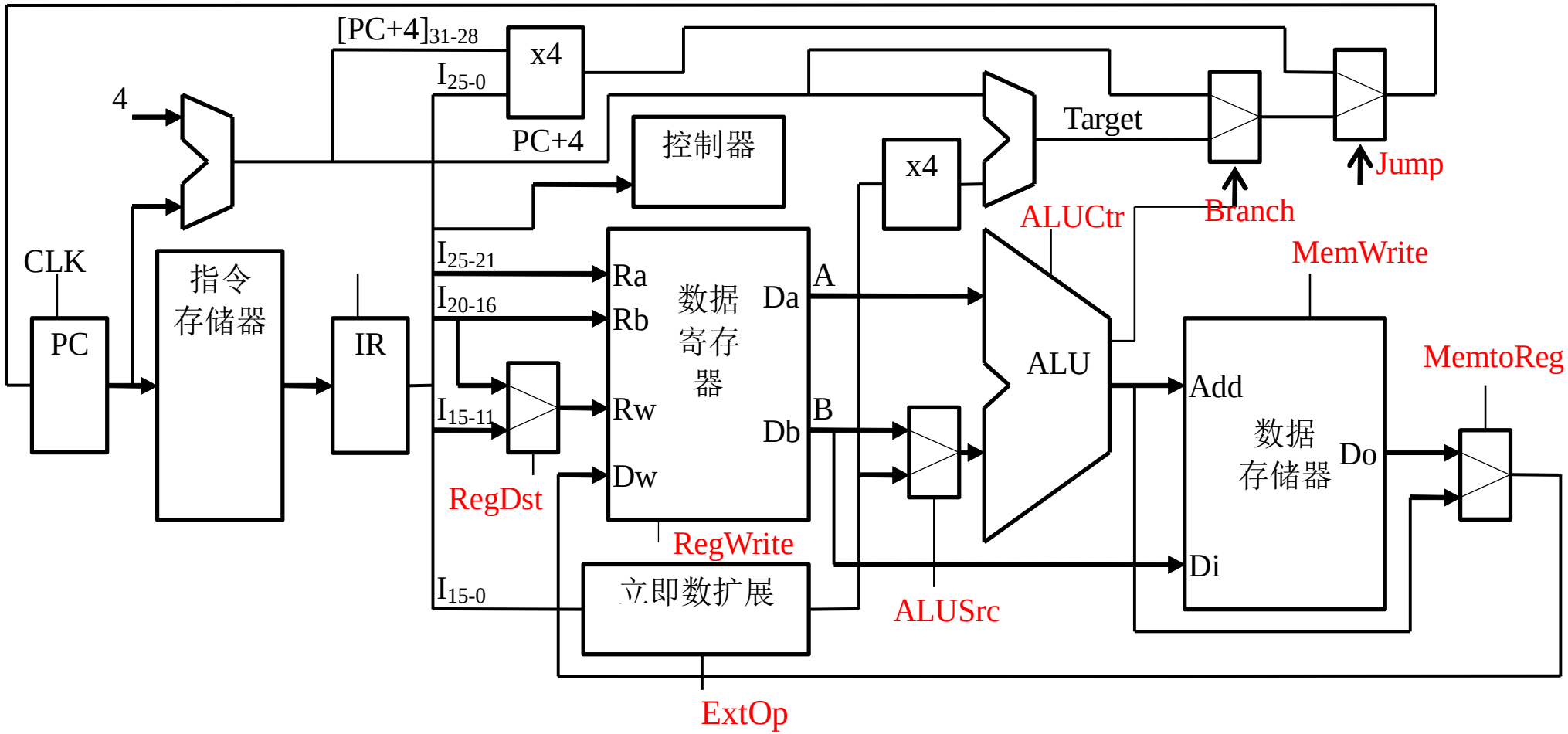
完整的数据通路


[PC+4]31-28
I25-0
x4
4
PC+4
控制器
Target
Jump
Branch
ALUCtr
MemWrite
MemtoReg
CLK
PC
指令
存储器
IR
I25-21
I20-16
I15-11
I15-0
Ra
Rb
Rw
Dw
数据
寄存
器
Da
Db
A
B
ALU
Add
数据
存储器
Do
Di
RegDst
RegWrite
立即数扩展
ALUSrc
ExtOp

处理器设计的五个步骤

1. 分析指令，得出对数据通路的需求
2. 选择数据通路上合适的组件
3. 连接组件构成数据通路
4. **分析每一条指令的实现，以确定控制信号**
5. 集成控制信号，完成控制逻辑

单周期处理器



控制信号:

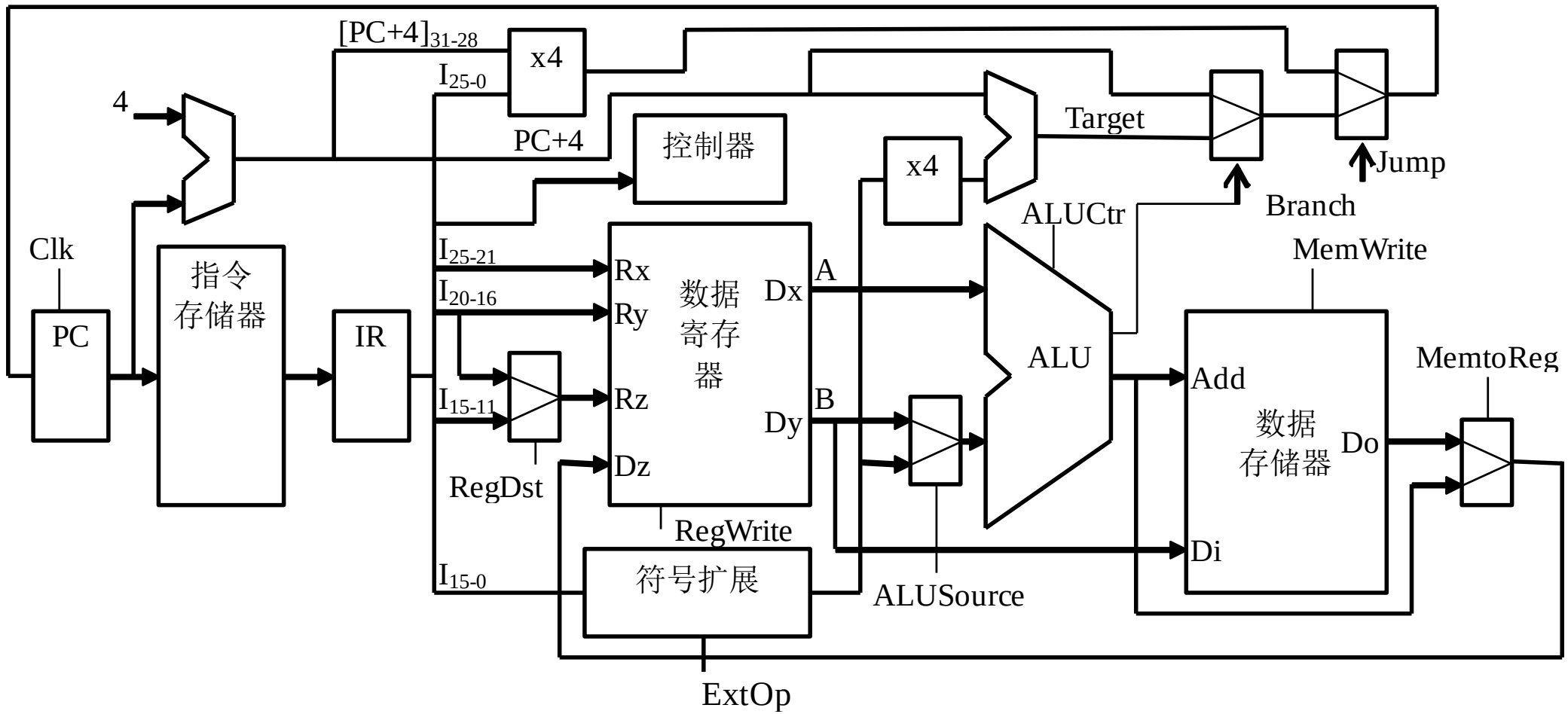
ALUCtr 运算操作码
ALUSrc ALU数据选择
ExtOp 无/带符号扩展

Branch 是否为条件转移指令
Jump 是否为无条件转移指令
MemWrite 存储器写

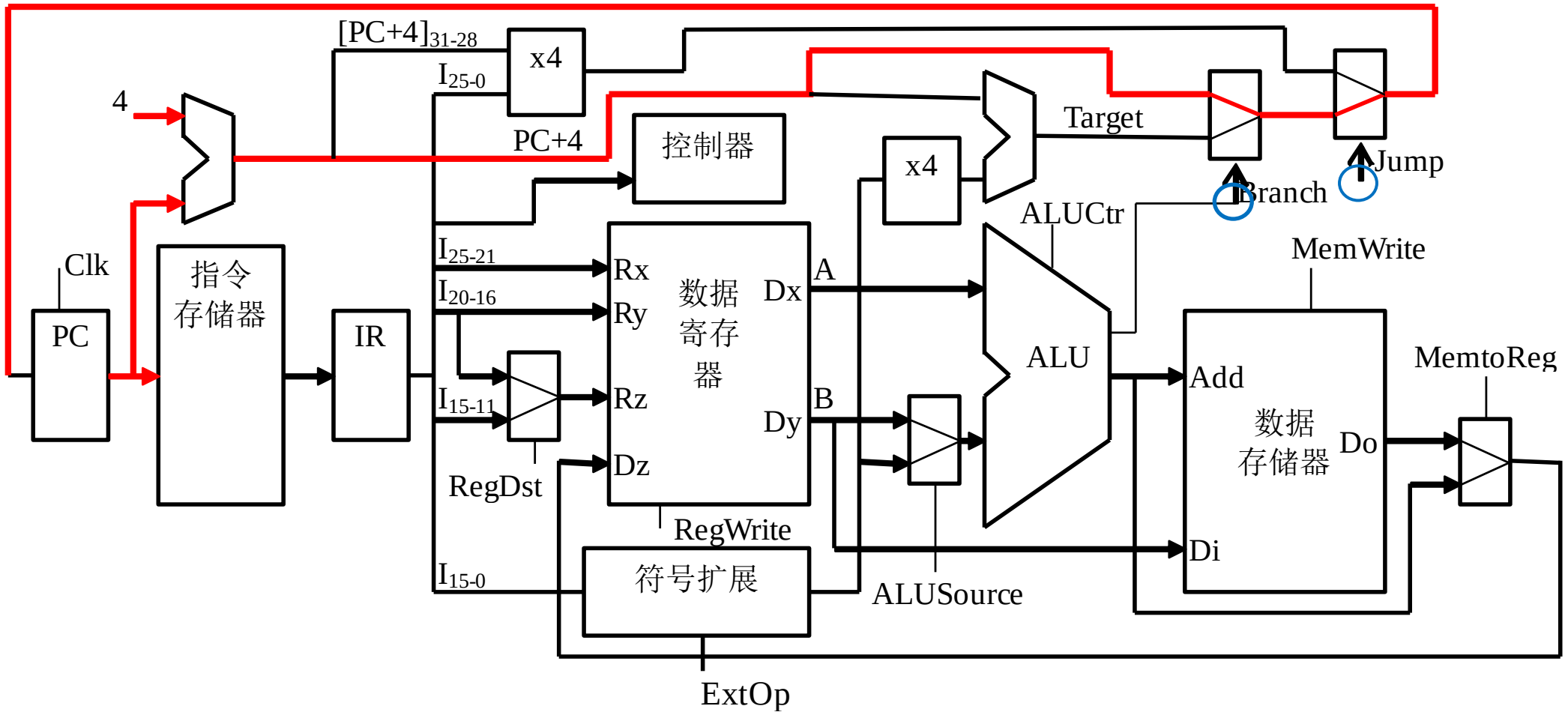
MemtoReg 写数据选择
RegWrite 数据寄存器写
RegDst 写寄存器选择

指令	数据通路	
ADD	**PC <– PC + 4**	**R[rd] <– R[rs] + R[rt];**

控制信号：



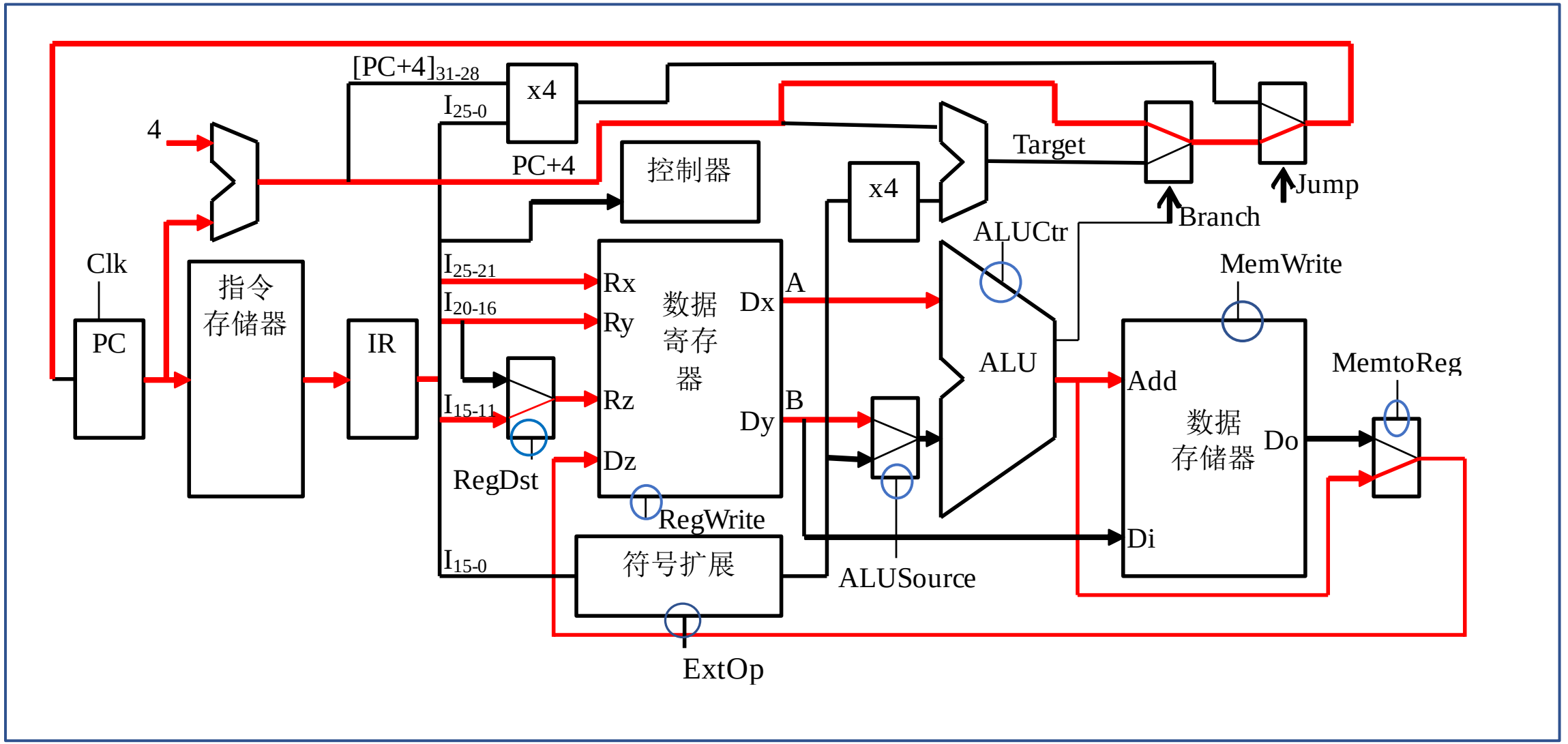
指令	数据通路	
ADD	PC <– PC + 4	R[rd] <– R[rs] + R[rt];
控制信号：	Branch = 0 , Jump=0,	ALUsrc = BusB , ALUctr = “add”, Extop=x, Memwrite=0, MemtoReg=ALU, RegDst = rd, RegWr=1



指令	数据通路
ADD	PC <– PC + 4
控制信号：	Branch = 0 , Jump=0,

R[rd] <– R[rs] + R[rt];

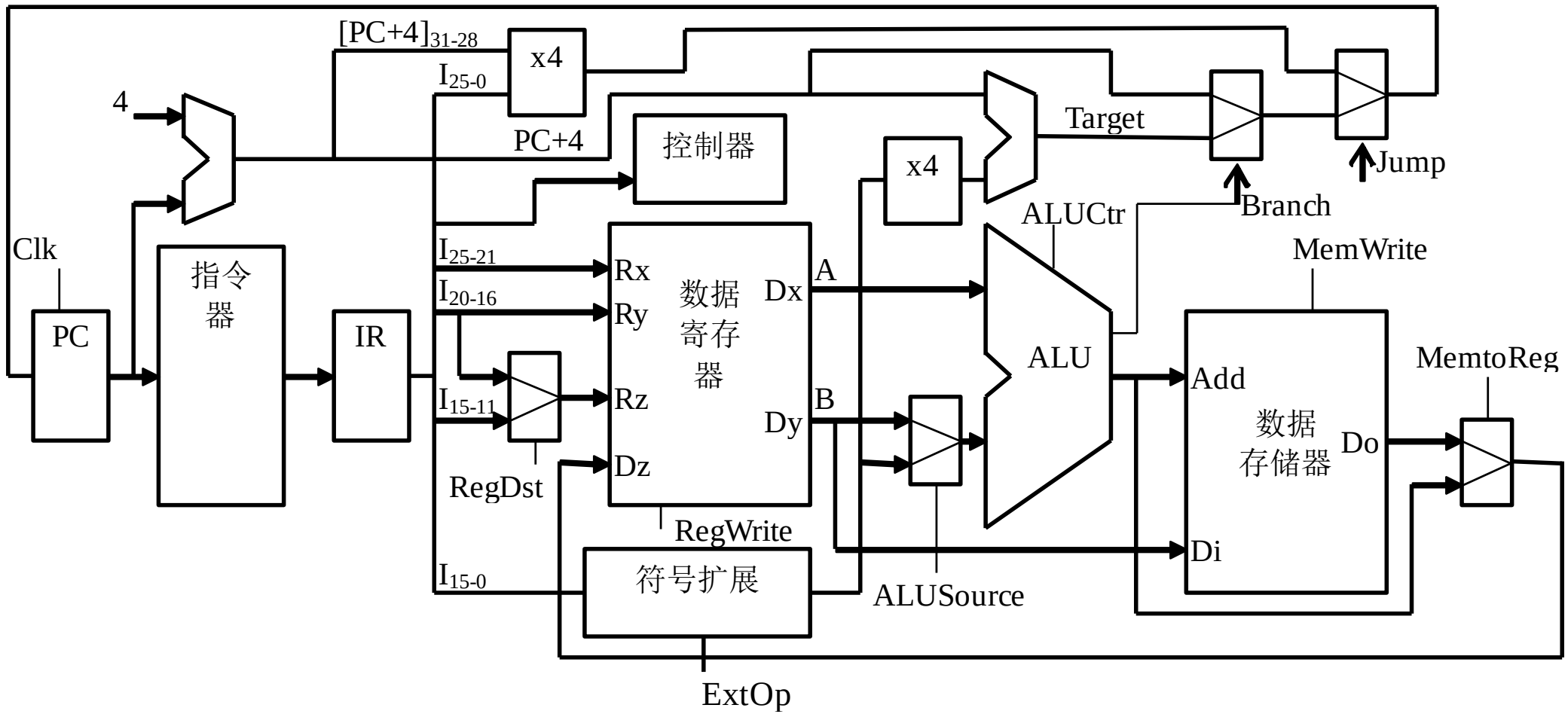
ALUsrc = BusB , ALUctr = “add”, Extop=x, Memwrite=0, MemtoReg=ALU, RegDst = rd, RegWr=1



指令　　数据通路

Ori　　PC <– PC + 4　　R[rt] <– R[rs] or unsign_ext(Imm16)];;

控制信号：



指令　　数据通路

Ori

控制信号：

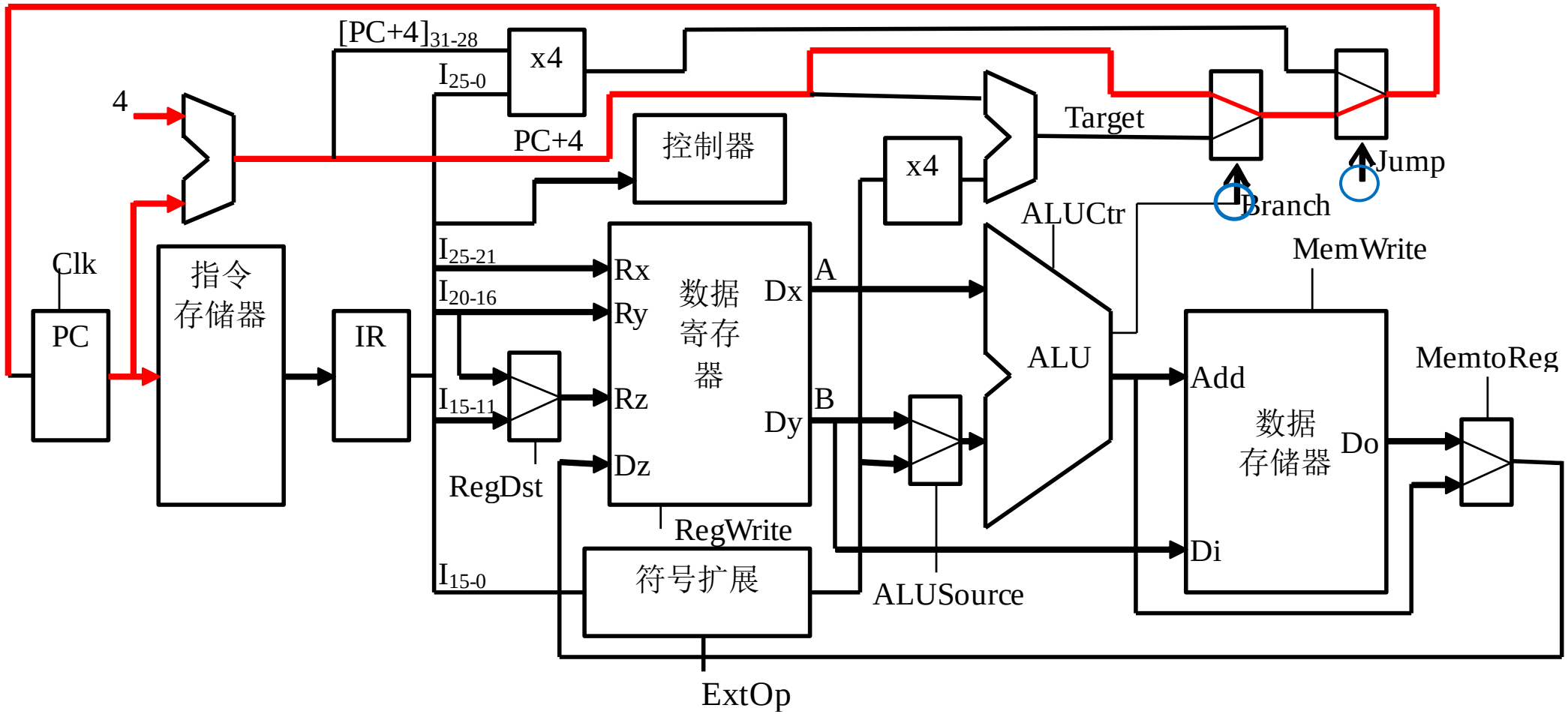
PC <– PC + 4

PC_source = 0, Jump=0,

R[rt] <– R[rs] or unsign_ext(Imm16)];

ALUsrc = Im, ALUCtr = “or”, Extop = “unSn，

Memwite=0, MemtoReg=ALU, RegDst = rt, RegWr=1

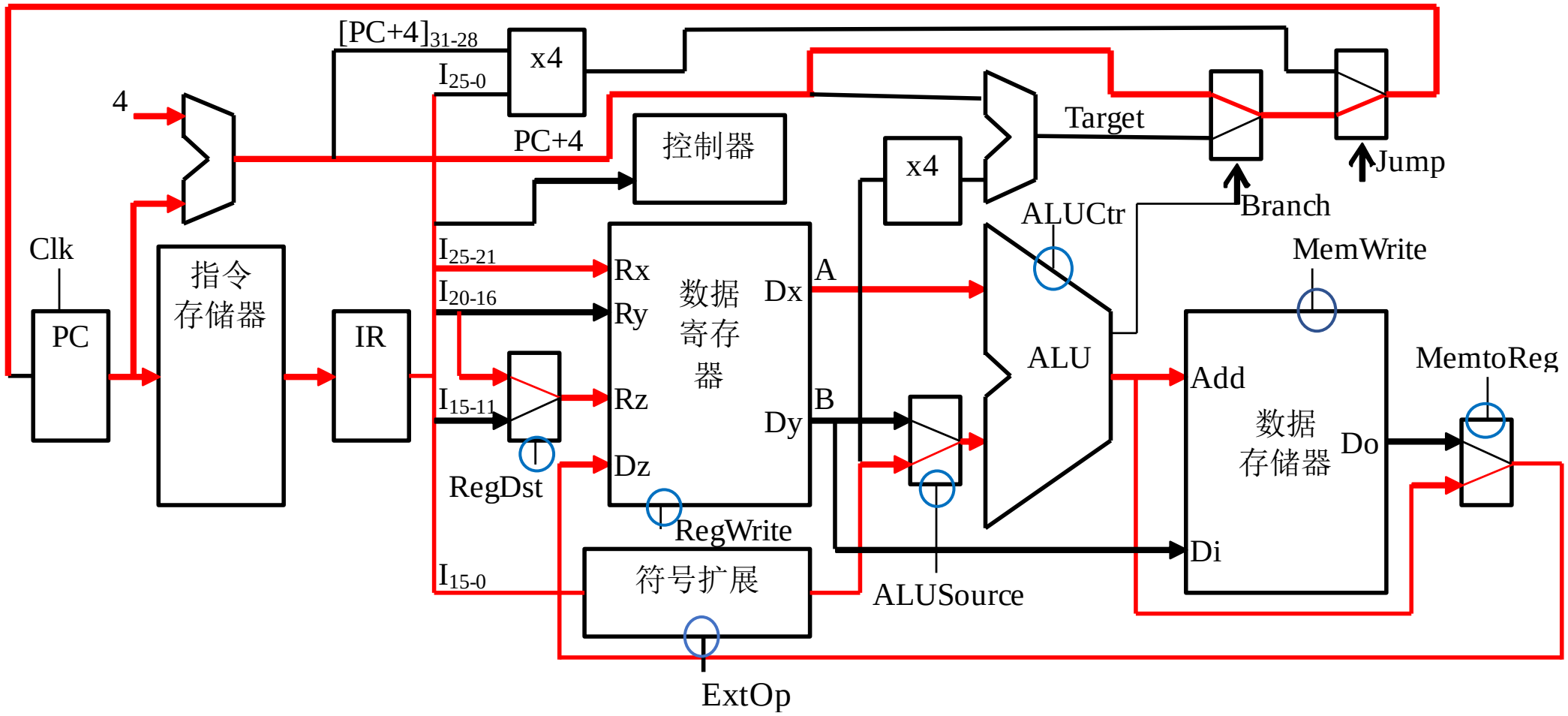


指令　　数据通路

Ori　　PC <– PC + 4

控制信号：　PC_source = 0, Jump=0,

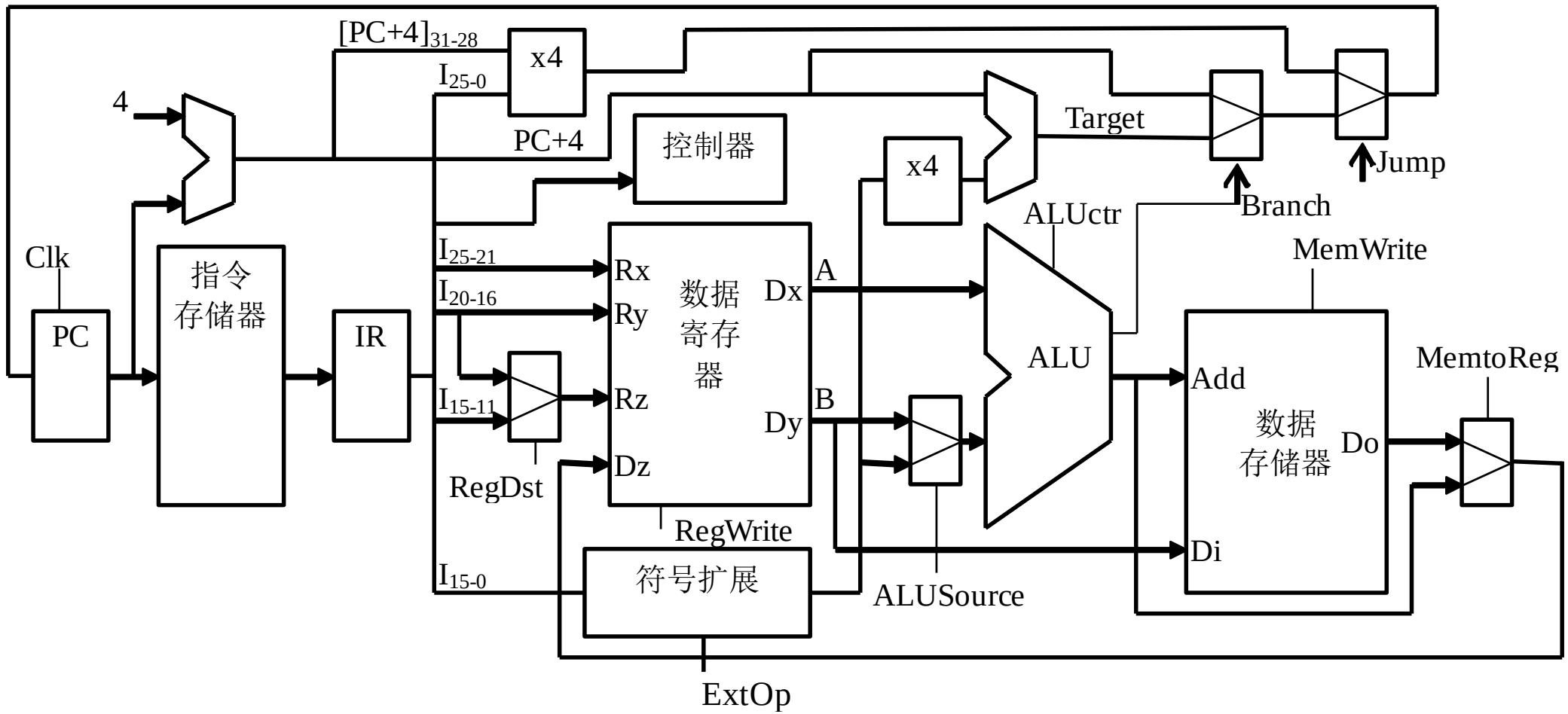
R[rt] <– R[rs] or unsign_ext(Imm16)];
ALUsrc = Im, ALUCtr = “or”, Extop = “unSn，
Memwite=0, MemtoReg=ALU, RegDst = rt, RegWr=1



指令　　数据通路

LOAD　　PC <– PC + 4 ，　　R[rt] <– MEM[R[rs] + sign_ext(Imm16)];

控制信号：



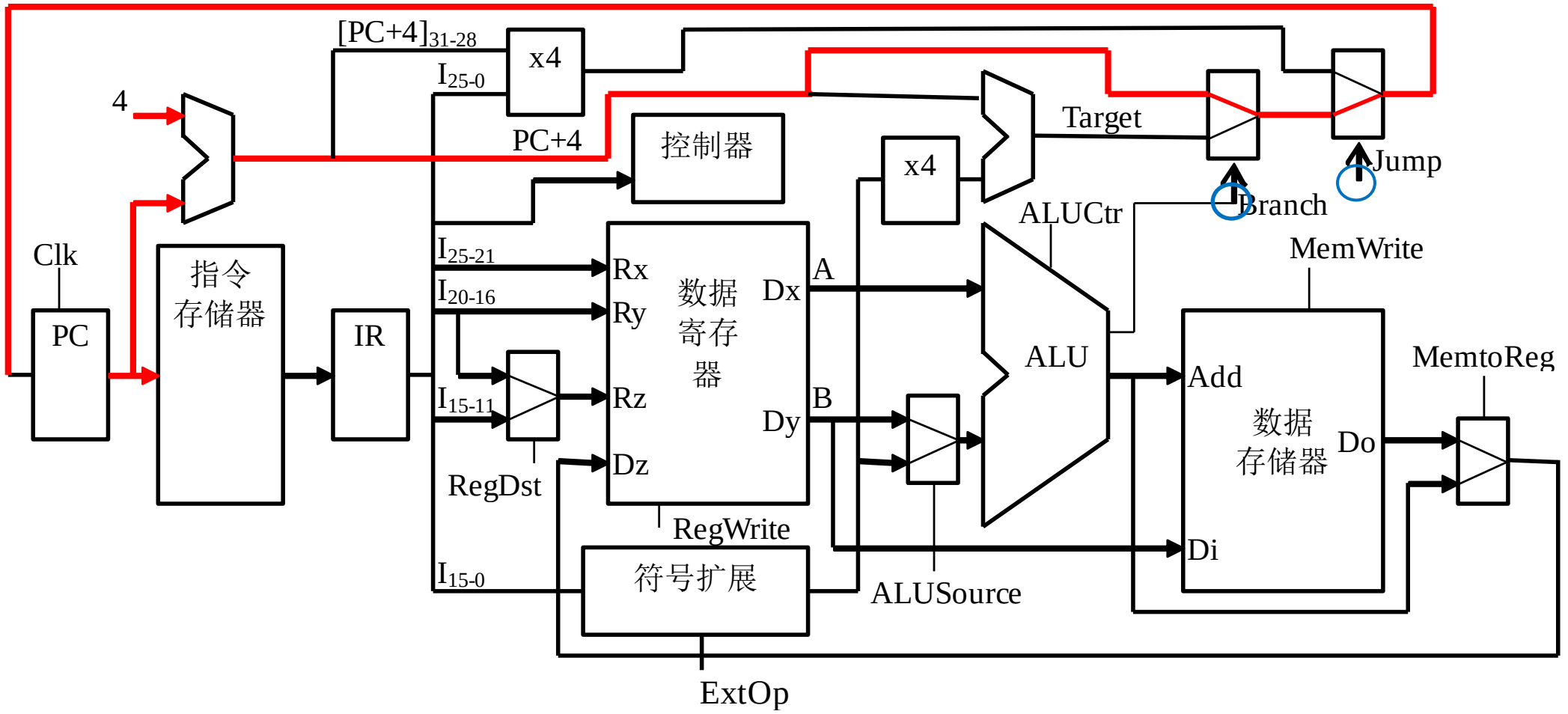
指令　　数据通路

LOAD　　**PC <– PC + 4 ，**

控制信号：　　**Branch = 0 , Jump=0,**

R[rt] <– MEM[R[rs] + sign_ext(Imm16)];

ALUsrc = Im, ALUctr= “add”, Extop = “Sn”,
Memwrite=0, MemtoReg=Mem, RegDst = rt, RegWr=1



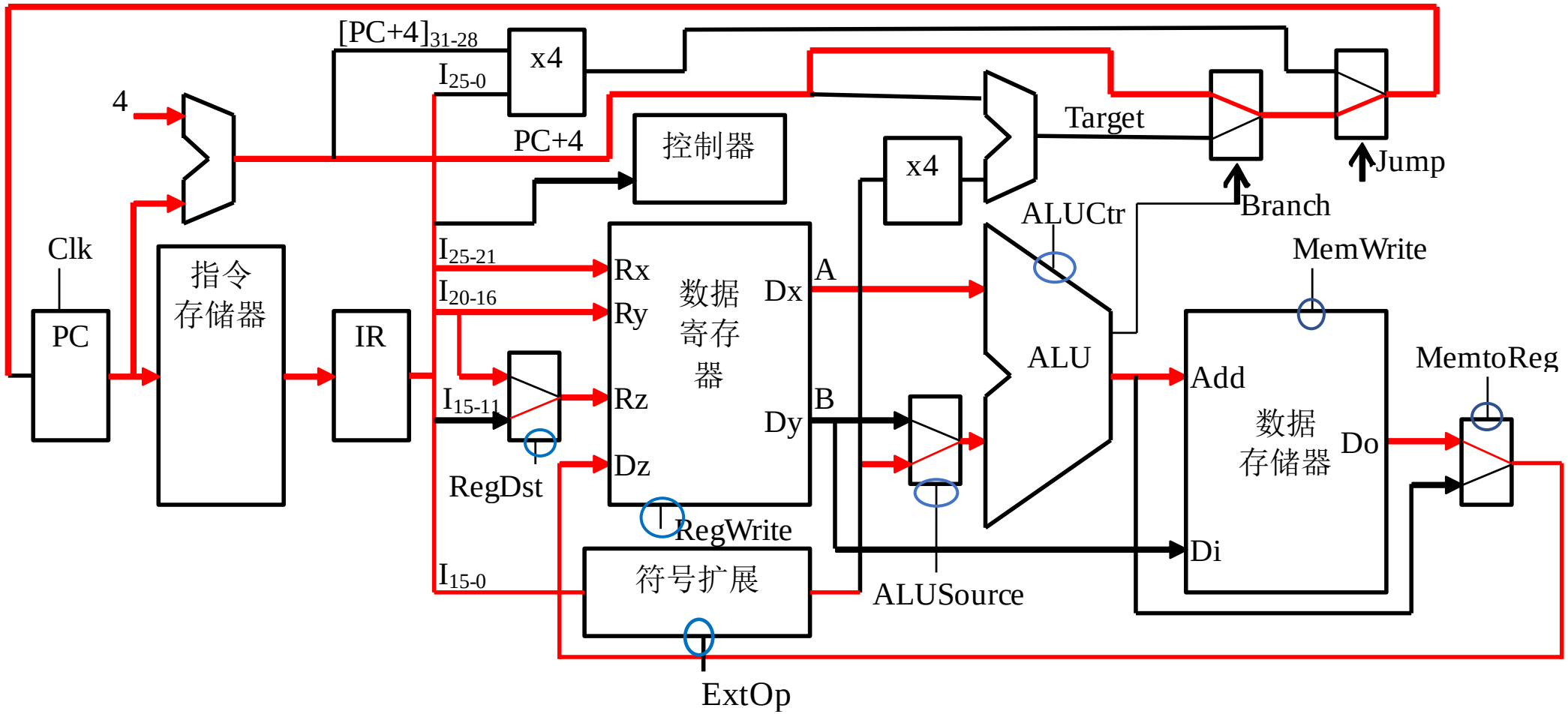
指令　　数据通路

LOAD　　PC <– PC + 4 ，

控制信号：　Branch = 0 , Jump=0,

R[rt] <– MEM[R[rs] + sign_ext(Imm16)];

ALUsrc = Im, ALUctr= “add”, Extop = “Sn”,
Memwrite=0, MemtoReg=Mem, RegDst = rt, RegWr=1

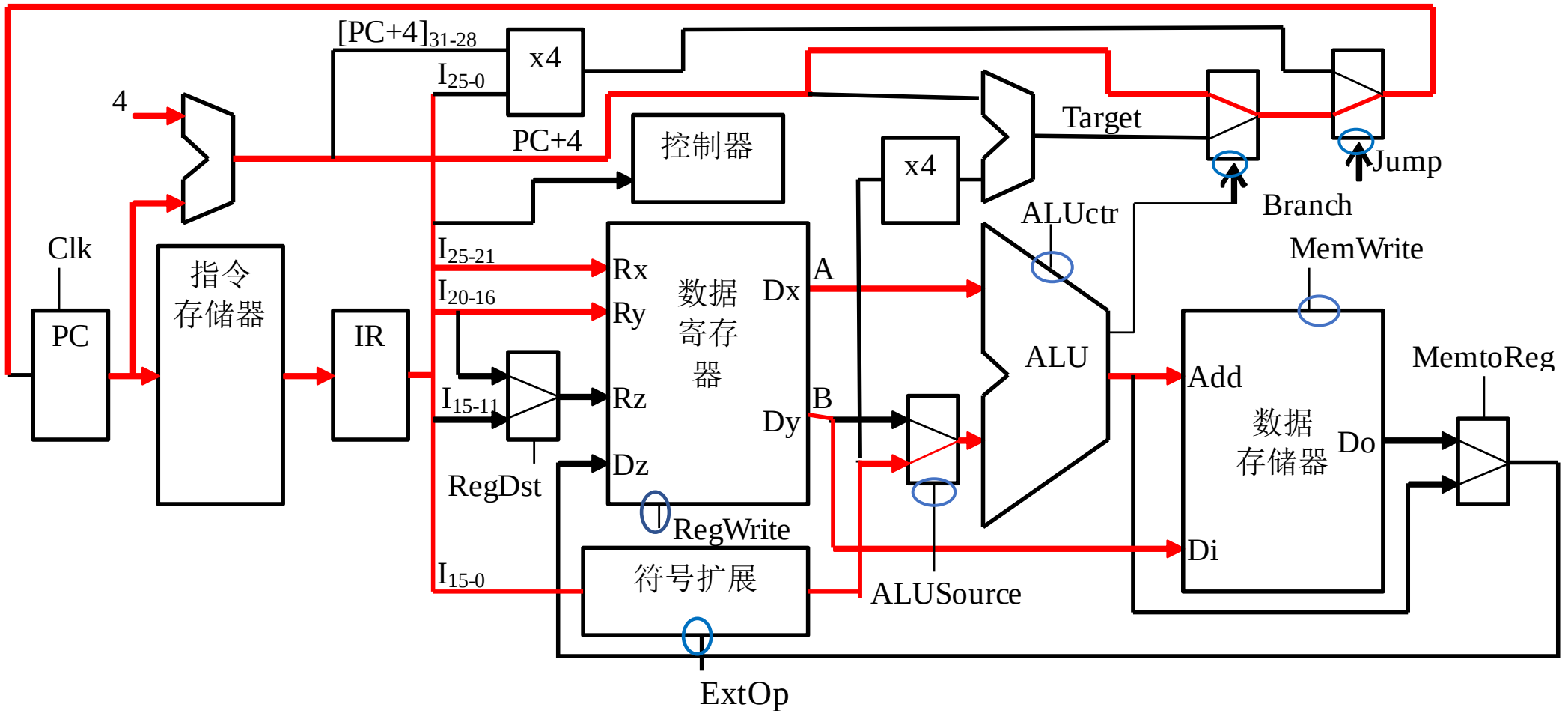


指令　　数据通路

STORE　　PC <– PC + 4 ，　　MEM[R[rs] + sign_ext(Imm16)] <– R[rs];

控制信号：　Branch = 0 , Jump=0,　　ALUsrc = Im, ALUctr = “add”, Extop = “Sn”,

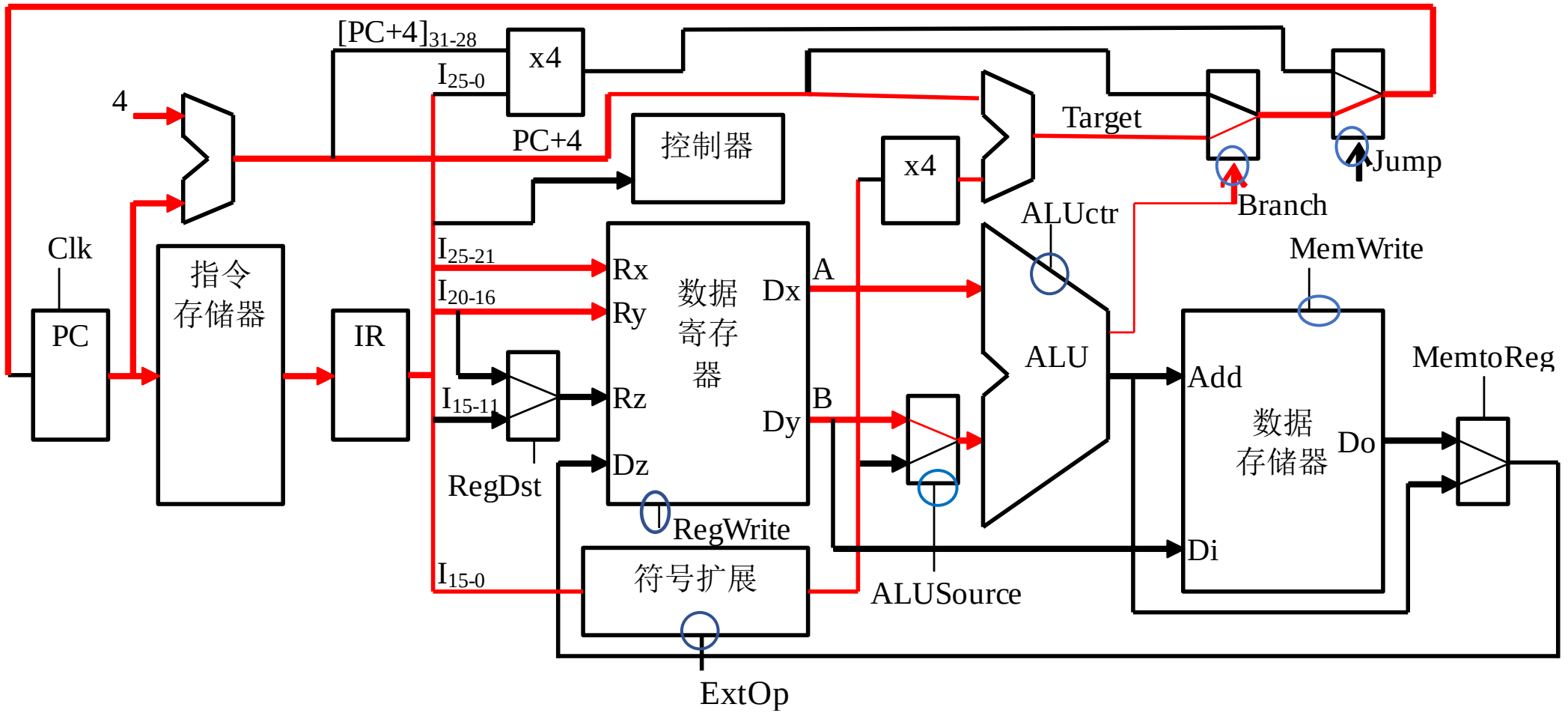
Memwrite=1, MemtoReg=x, RegDst = x, RegWr=0



指令　　数据通路

BEQ　　if (R[rs] == R[rt]) then PC <– PC+4 + sign_ext(Imm16)] || 00 else PC <– PC + 4

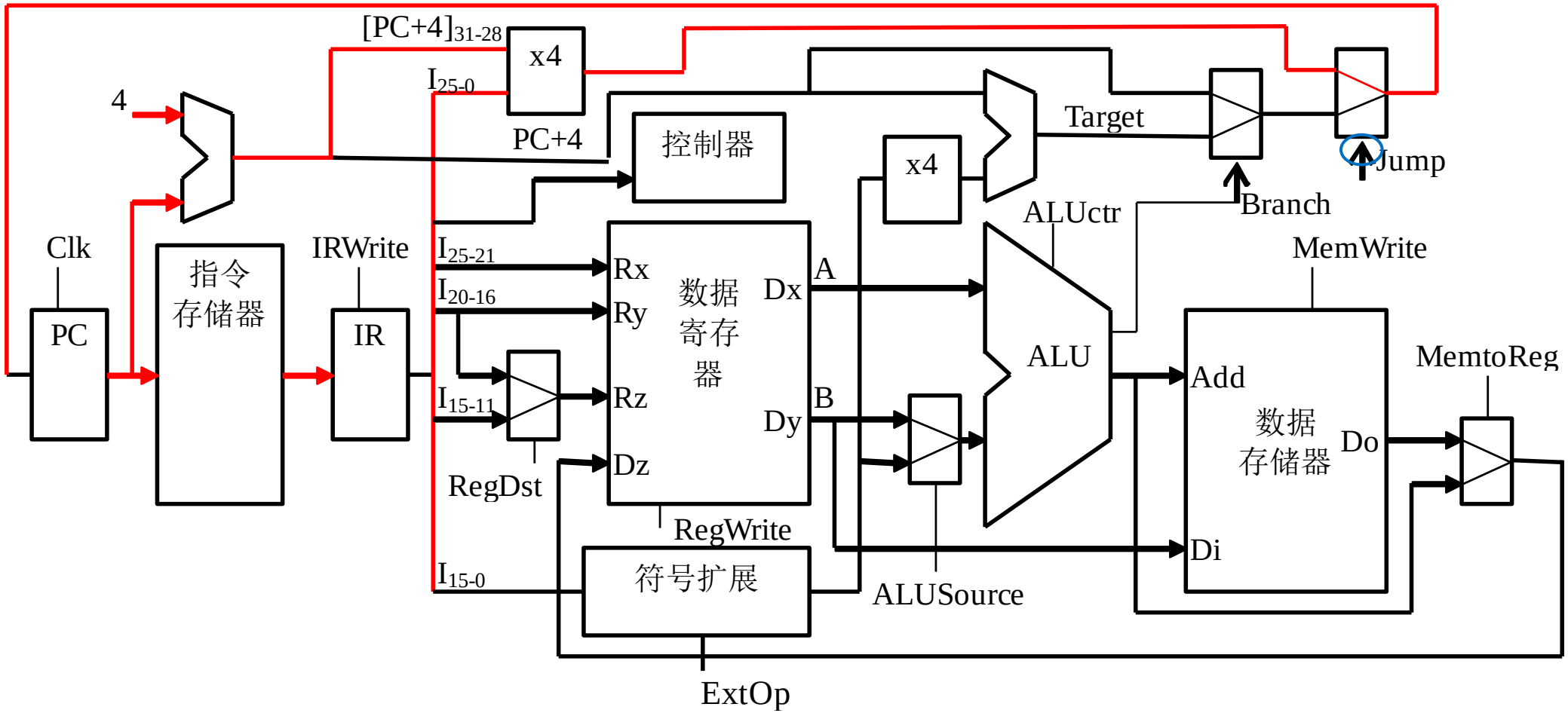
控制信号：　**ALUsrc = BusB , ALUctr = “sub” , Extop = “Sn”, Branch = “Br”, Jump=0, Memwrite=0, Regwrite=0，其余=x**



指令　　数据通路

JUMP　　PC <– (PC +4[31-28], I_{25-0}) || 00

控制信号：　Branch=0, Jump=1, Memwrite=0, Regwrite=0，其余=x



控制信号总结

指令　　数据通路和控制信号：

ADD　　**R[rd] <– R[rs] + R[rt];**　　**PC <– PC + 4**

Branch = 0 , Jump=0, ALUsrc = BusB , Extop=x, ALUctr = “add”,
Memwrite=0, MemtoReg=ALU, RegDst = rd, RegWr=1

Ori　　**R[rd] <– R[rs] or R[rt];**　　**PC <– PC + 4**

PC_source = 0, Jump=0, Extop = “Sn”, ALUsrc = Im, ALUCtr = “or”,
Memwite=0, MemtoReg=ALU, RegDst = rt, RegWr=1

LOAD　　**R[rt] <– MEM[R[rs] + sign_ext(Imm16)];**　　**PC <– PC + 4**

Branch = 0 , Jump=0, Extop = “Sn”, ALUsrc = Im, ALUctr= “add”,
Memwrite=0, MemtoReg=Mem, RegDst = rt, RegWr=1

STORE　　**MEM[R[rs] + sign_ext(Imm16)] <– R[rs];**　　**PC <– PC + 4**

Branch = 0 , Jump=0, Extop = “Sn”, ALUsrc = Im, ALUctr = “add”,
Memwrite=1, MemtoReg=x, RegDst = x, RegWr=0

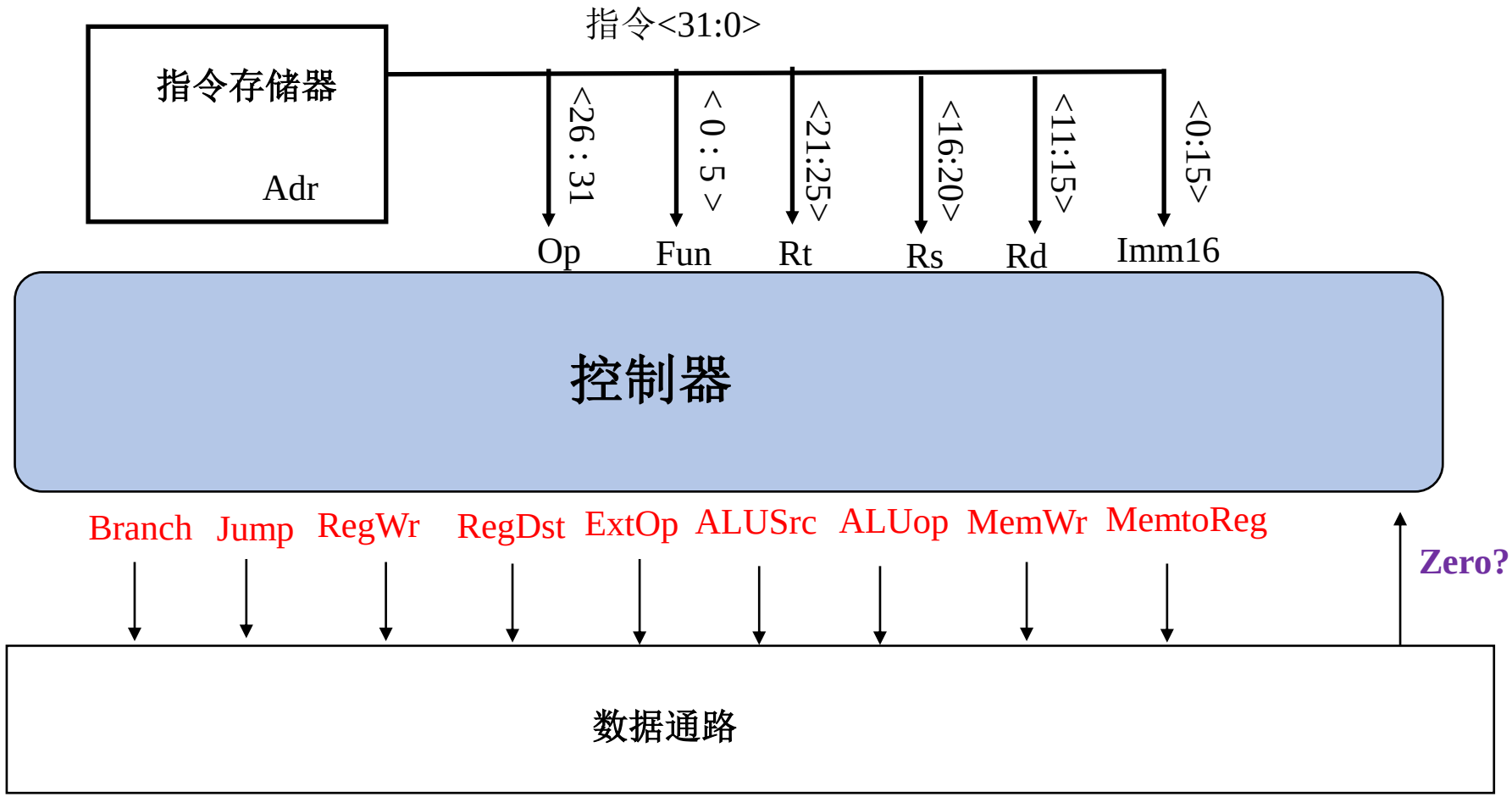
BEQ　　**if (R[rs] == R[rt]) then PC <– PC + sign_ext(Imm16)] || 00 else PC <– PC + 4**

ALUsrc = BusB , Extop = “Sn”, ALUctr = “sub” , Branch = “Br”, Jump=0,
Memwrite=0, Regwrite=0， MemtoReg=x, RegDst = x,

JUMP　　**PC <– (PC +4[31-28], $I_{25\text{-}0}$) || 00**

Branch=0, Jump=1, Memwrite=0, Regwrite=0，其余=x

集成控制信号


指令<31:0>
指令存储器
Adr
<26:31>
<0:5>
<21:25>
<16:20>
<11:15>
<0:15>
Op
Fun
Rt
Rs
Rd
Imm16
控制器
Branch
Jump
RegWr
RegDst
ExtOp
ALUSrc
ALUop
MemWr
MemtoReg
Zero?
数据通路

处理器设计的五个步骤

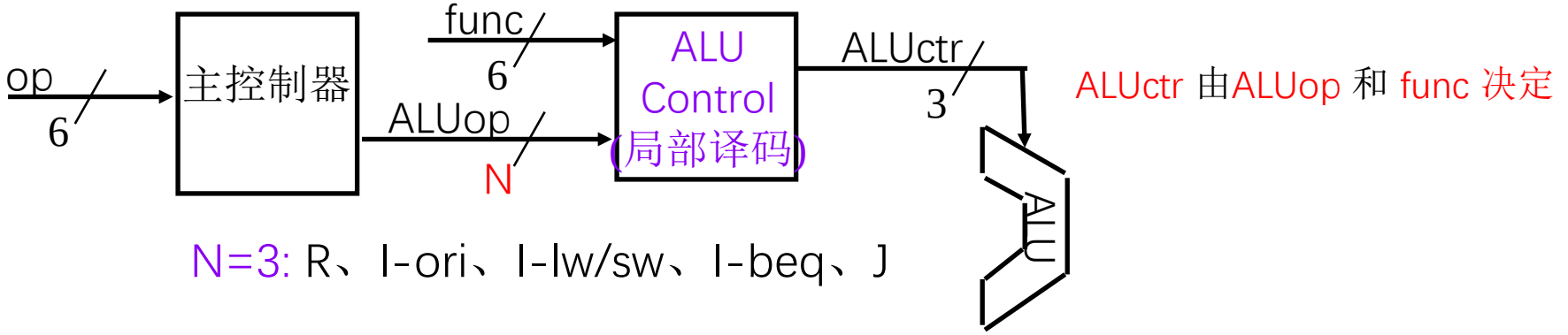
1. 分析指令，得出对数据通路的需求
2. 选择数据通路上合适的组件
3. 连接组件构成数据通路
4. 分析每一条指令的实现，以确定控制信号
5. **集成控制信号，完成控制逻辑**

控制信号总结

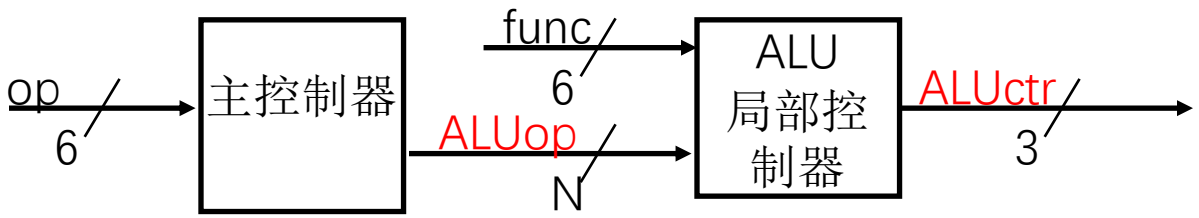
func / op	10 0000	10 0010	无关项				
	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	add	sub	ori	lw	sw	beq	jump
RegDst	1	1	0	0	x	x	x
ALUSrc	0	0	1	1	1	0	x
MemtoReg	0	0	0	1	x	x	x
RegWrite	1	1	1	1	0	0	0
MemWrite	0	0	0	0	1	0	0
Branch	0	0	0	0	0	1	0
Jump	0	0	0	0	0	0	1
ExtOp	x	x	0	1	1	1	x
ALUctr<2:0>	Add	Subtr	Or	Add	Add	Subtr	xxx

ALU Control的局部译码

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/Subtr	Or	Add	Add	Subtr	xxx



ALUop的编码

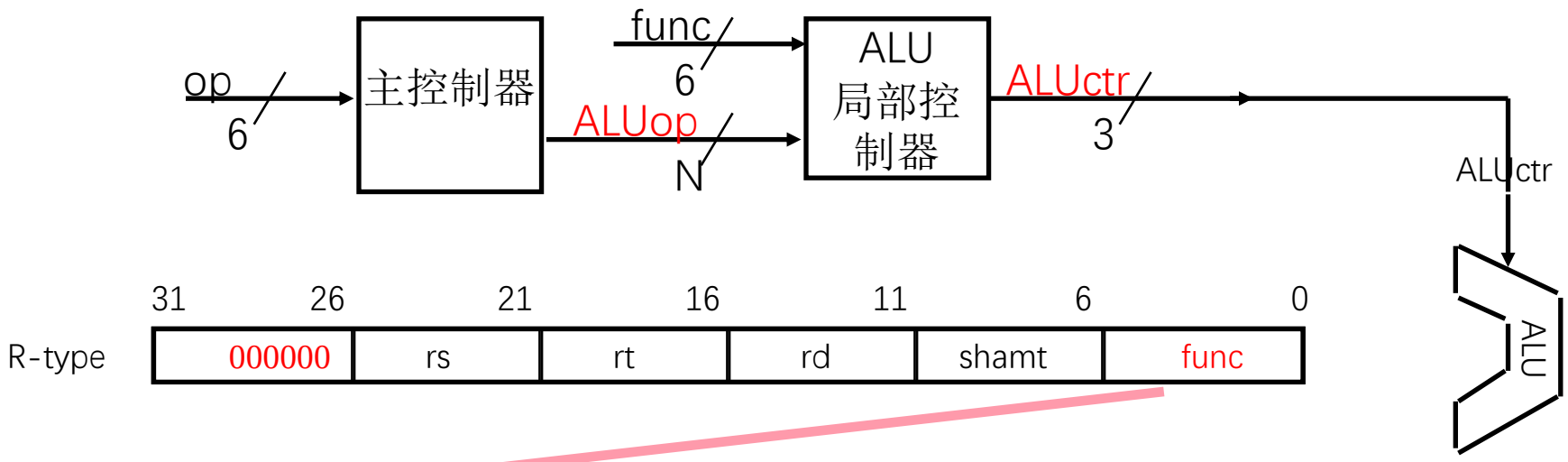


对 ALUop 编码(N=3）

	R-type	ori	lw	sw	beq	jump
ALUop (Symbolic)	“R-type”	Or	Add	Add	Subtr	xxx
ALUop<2:0>	1 xx	0 10	0 00	0 00	0x1	xxx

ALUop 也可以只用两位（N=2）：
J-xx , R:11, ori:10, beq:01, lw/sw:00,

ALUctr的编码



func<5:0>	指令的操作	ALUctr<2:0>	ALU 运算
10 0000	add	000	Add
10 0010	subtract	001	Subtract
10 0100	and	100	And
10 0101	or	101	Or
10 1010	set-on-less-than	010	Subtract

ALUctr 的真值表

R型指令由 func决定ALUctr

非R型指令由 ALUop决定ALUctr

ALUop (符号)	R-型	ori	lw	sw	beq
	“R-type”	Or	Add	Add	Subtr
ALUop<2:0>	1 00	0 10	0 00	0 00	0 x1

funct<3:0>	指令的运算操作
0000	add
0010	subtract
0100	and
0101	or
1010	set-on-less-than

ALUop			func				ALU 运算操作	ALUctr		
bit2	bit1	bit0	bit<3>	bit<2>	bit<1>	bit<0>		bit<2>	bit<1>	bit<0>
0	0	0	x	x	x	x	Add	0	0	0
0	x	1	x	x	x	x	Subtract	0	0	1
0	1	0	x	x	x	x	Or	1	1	0
1	x	x	0	0	0	0	Add	0	0	0
1	x	x	0	0	1	0	Subtract	0	0	1
1	x	x	0	1	0	0	And	0	1	0
1	x	x	0	1	0	1	Or	1	1	0
1	x	x	1	0	1	0	Subtract	0	0	1

ALUctr<0> 的逻辑表达式

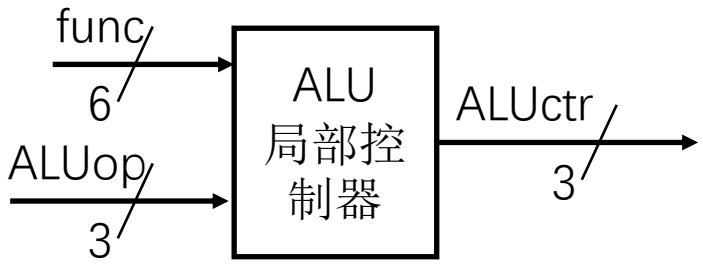
ALUctr[0]=1 所在的行

ALUop			func				
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>	ALUctr<0>
0	x	1	x	x	x	x	1
1	x	x	0	0	1	0	1
1	x	x	1	0	1	0	1

ALUctr<0> = !ALUop<2> & ALUop<0> +

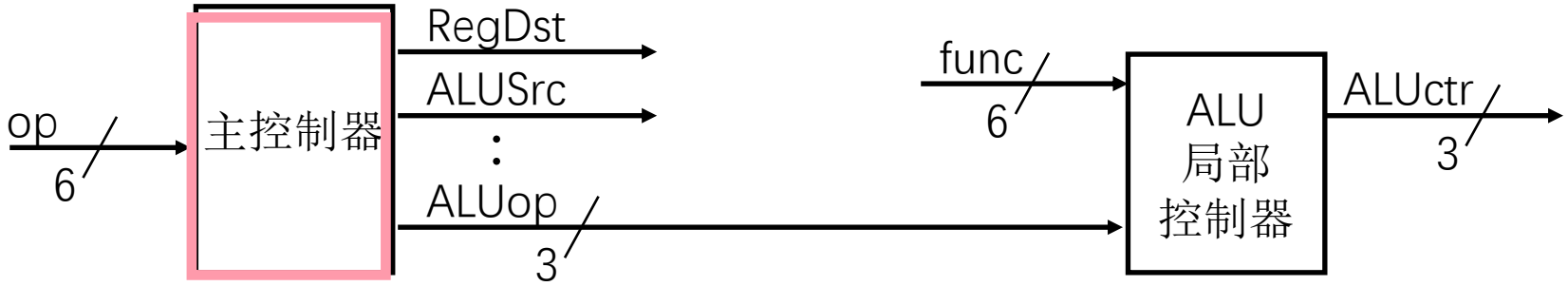
ALUop<2> & !func<2> & func<1> & !func<0>

ALU Control 控制信号汇总



- ALUctr<0> = !ALUop<2> & ALUop<0> +
  ALUop<2> & !func<2> & func<1> & !func<0>
- ALUctr<1> = !ALUop<2> & ALUop<1> & !ALUop<0> +
  ALUop<2> & !func<3> & func<2> & !func<1>
- ALUctr<2> = !ALUop<2> & ALUop<1> & !ALUop<0> +
  ALUop<2> & !func<3> & func<2> & !func<1> & func<0>

Main Control 的真值表



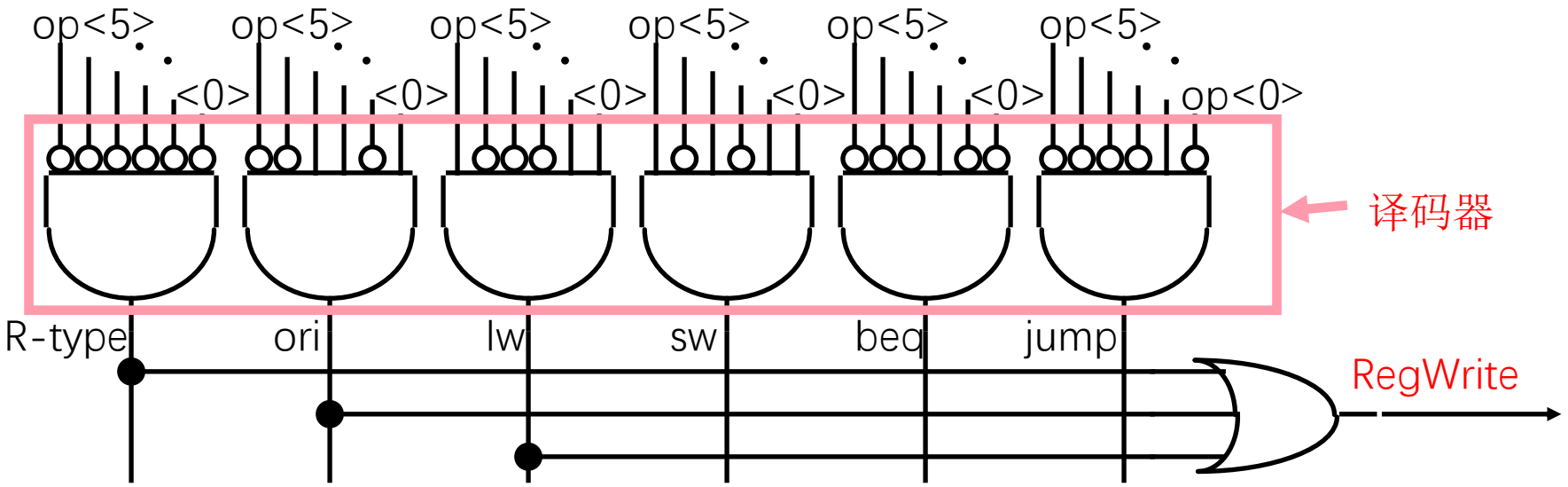
主控制器的输入

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
ALUop (符号)	“R-型”	Or	Add	Add	Subtr	xxx
ALUop <2>	1	0	0	0	0	x
ALUop <1>	x	1	0	0	x	x
ALUop <0>	x	0	0	0	1	x

主控制器的输出

RegWrite 信号的真值表

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



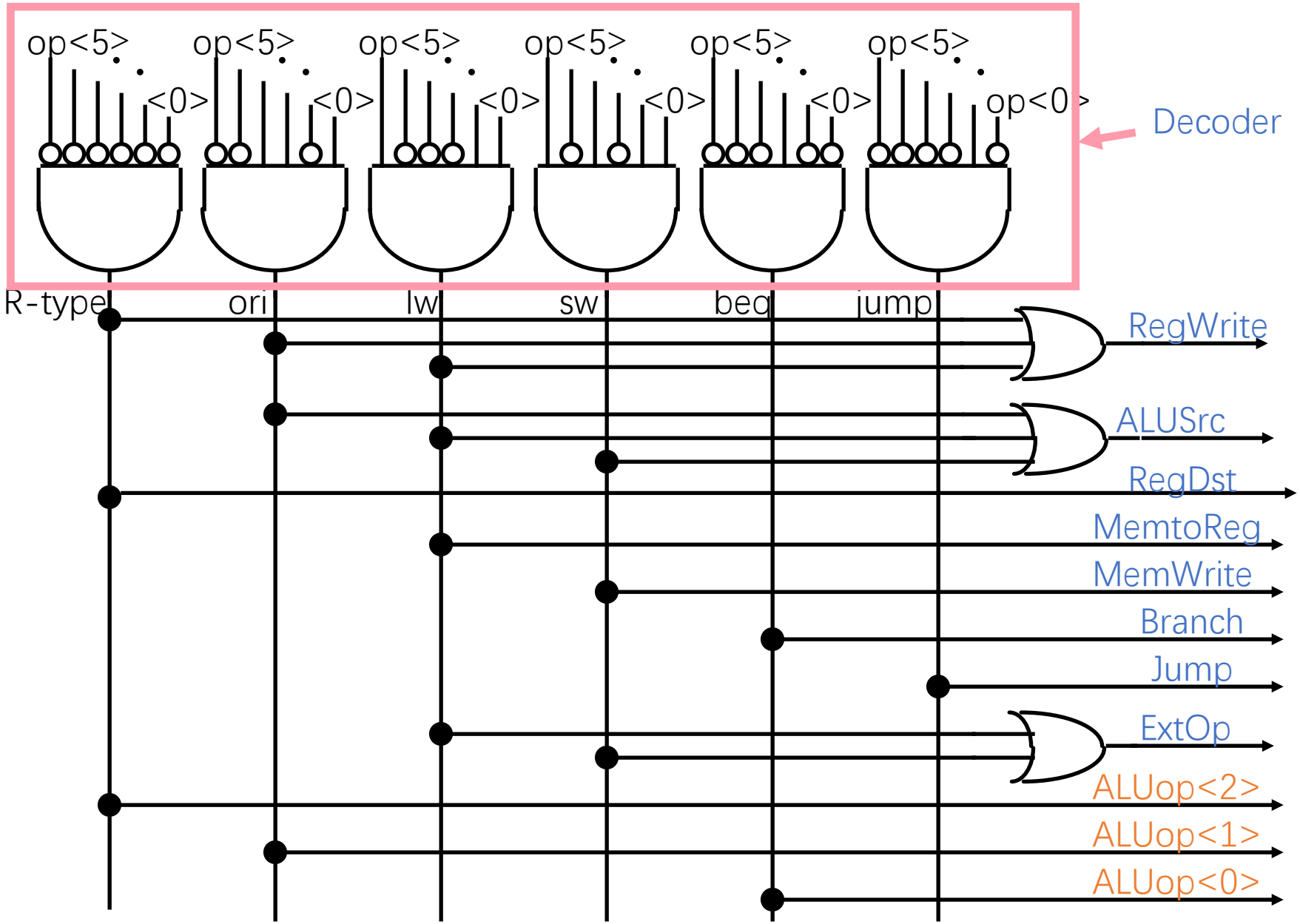
RegWrite = R型 + ori + lw

= !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0> (R型)

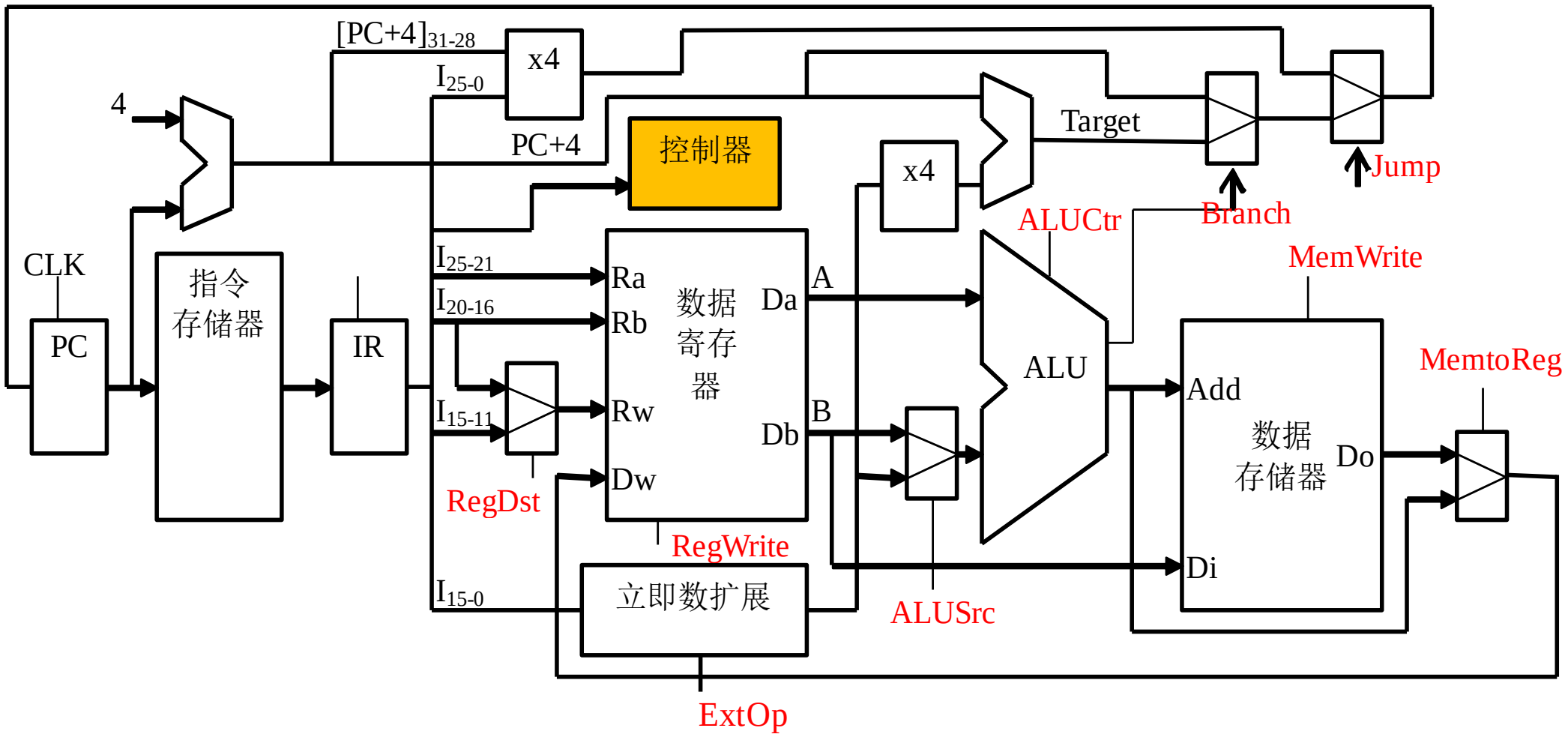
+ !op<5> & !op<4> & op<3> & op<2> & !op<1> & op<0> (ori)

+ op<5> & !op<4> & !op<3> & !op<2> & op<1> & op<0> (lw)

其他控制信号真值表


op<5>
<0>
op<0>
Decoder
R-type
ori
lw
sw
beq
jump
RegWrite
ALUSrc
RegDst
MemtoReg
MemWrite
Branch
Jump
ExtOp
ALUop<2>
ALUop<1>
ALUop<0>

小结


[PC+4]31-28
I25-0
x4
4
PC+4
控制器
Target
Jump
Branch
ALUCtr
MemWrite
MemtoReg
CLK
PC
指令
存储器
IR
I25-21
I20-16
I15-11
I15-0
Ra
Rb
Rw
Dw
数据
寄存
器
Da
Db
A
B
ALU
Add
数据
存储器
Do
Di
RegDst
RegWrite
立即数扩展
ALUSrc
ExtOp

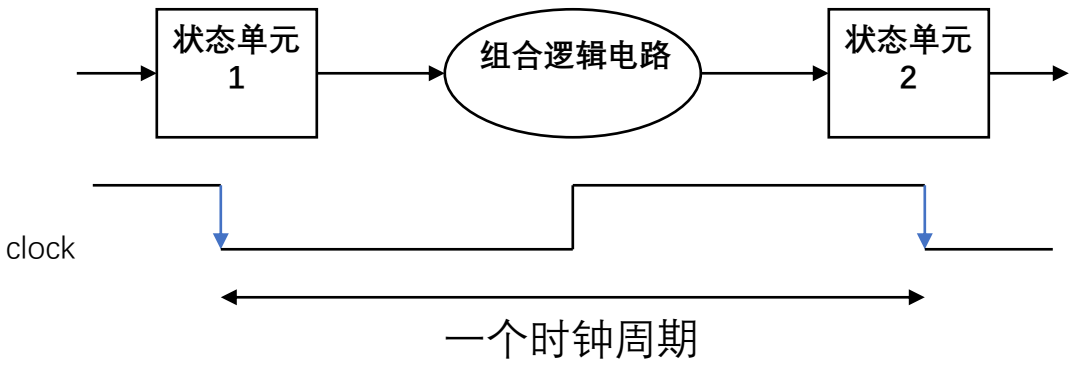
设计处理器的五个步骤

1. 分析指令系统，得出对数据通路的需求
2. 选择数据通路上合适的组件
3. 连接组件构成数据通路
4. 分析每一条指令的实现，以确定控制信号
5. 集成控制信号，完成控制逻辑

多周期处理器

回顾：单周期处理器

- 单周期处理器：一个时钟周期完成一条指令：

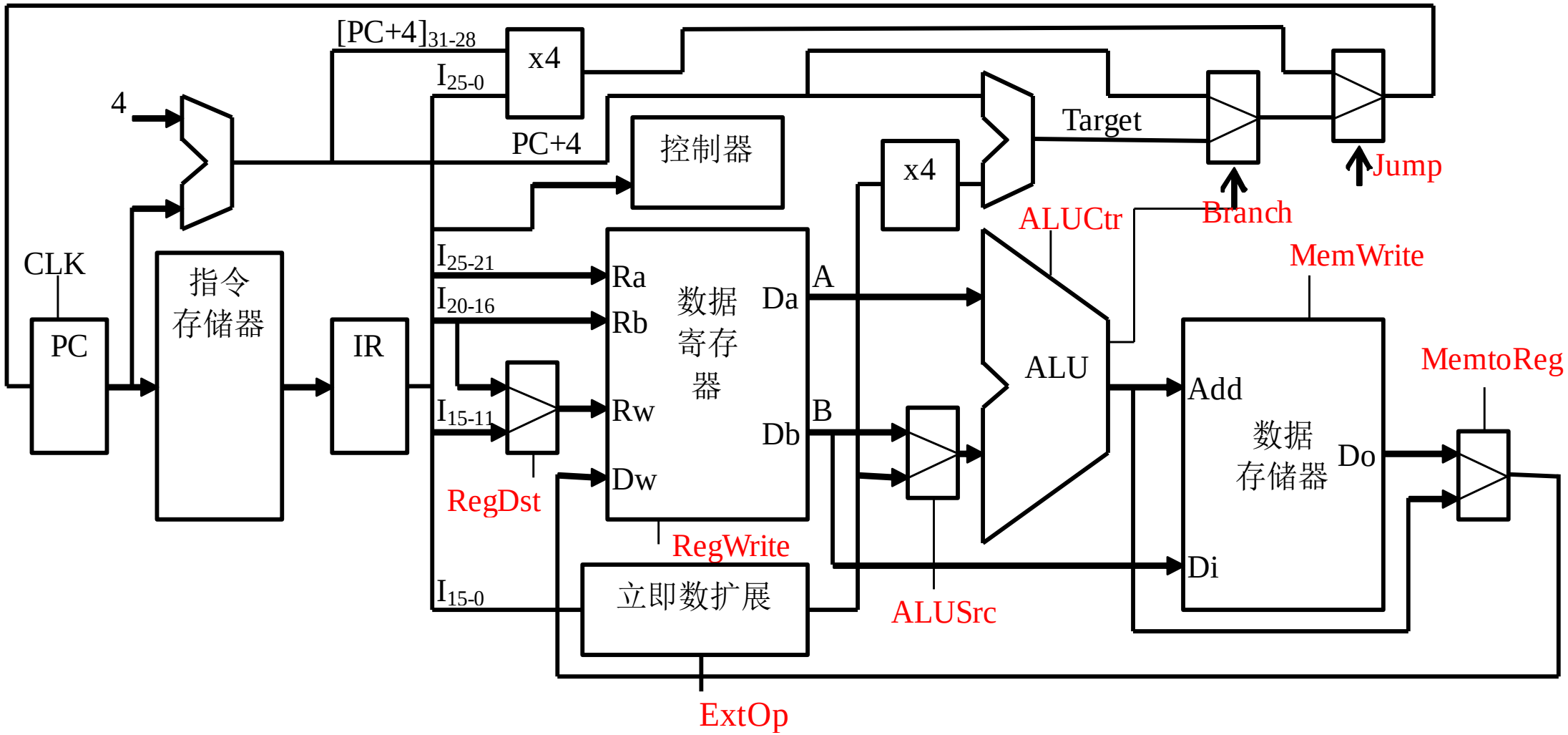


- 读状态单元的内容 -> 通过组合逻辑电路实现指令的功能 -> 将结果写入一个或多个状态单元
  - 状态单元：

    例如：寄存器、存储器
  - 边沿触发

    状态转换发生在时钟边沿

单周期处理器完整的数据通路



单周期处理器的性能分析

假定在具体实现中，以下主要组成部件的工作时间如下：

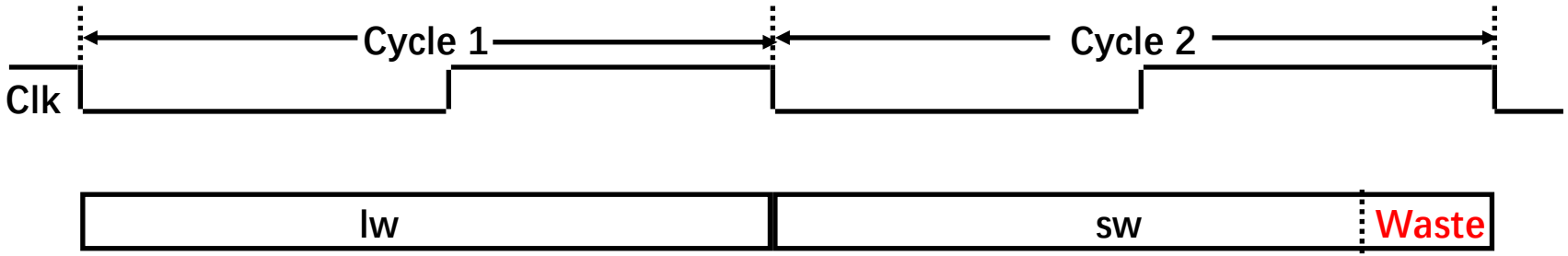
- 存储器: 200 ps； ALU 和加法器: 100 ps； 寄存器文件 (读 或 写): 50 ps
- 多路选择器、控制器、PC访问、符号位扩展没有延迟、连接线路也无延迟
- 不考虑寄存器建立时间和锁存延迟、不考虑时钟偏斜

❑ 计算每条指令的延迟：

指令	I Mem	Reg Rd	ALU Op	D Mem	Reg Wr	合计
R型	200	50	100	0	50	400
load	200	50	100	200	50	600
store	200	50	100	200		550
beq	200	50	100	0		350
jump	200					200

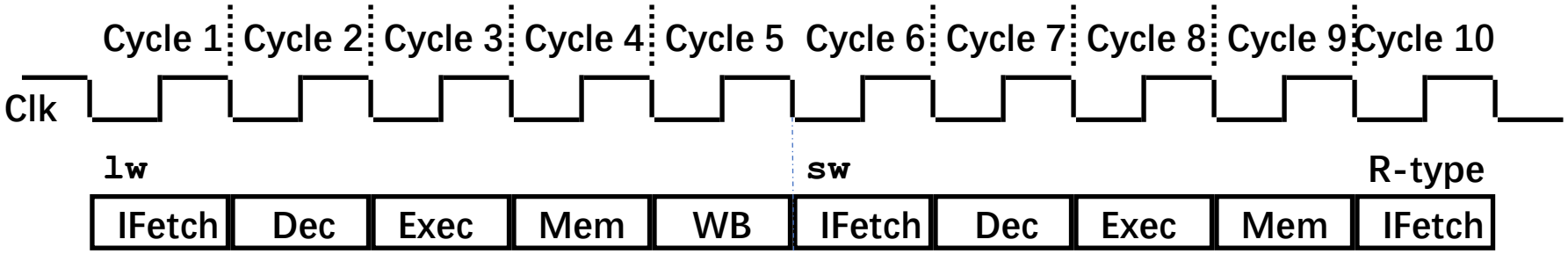
单周期： Disadvantages & Advantages

- 时钟周期受限于最慢的一条指令 （slowest instruction）



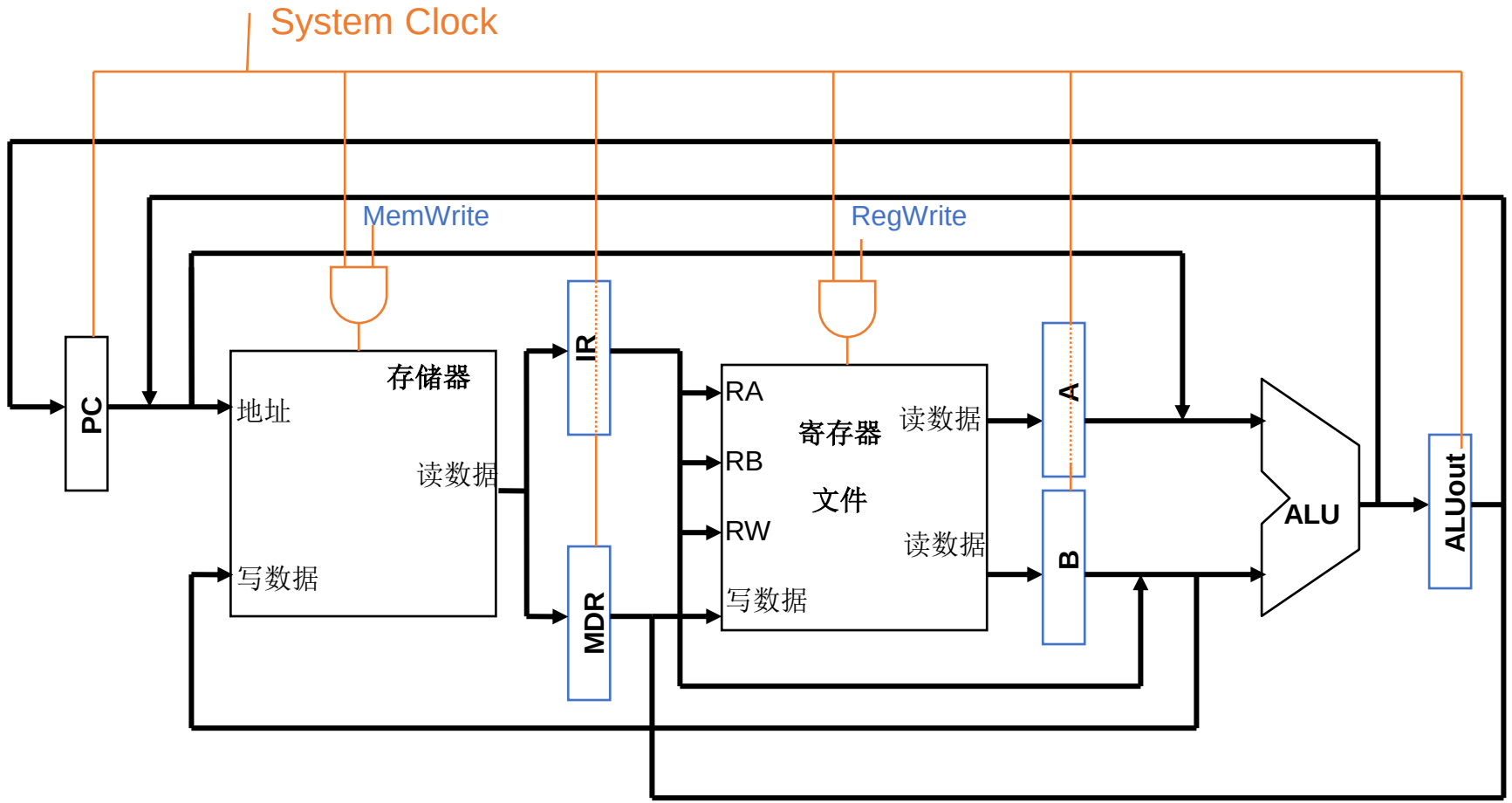
- 缺点： 芯片面积的浪费，
  - 某些功能部件 (例如：加法器) 在一条指令周期内只能使用一次，同一个部件需要多个。
- 优点： 简单、容易理解

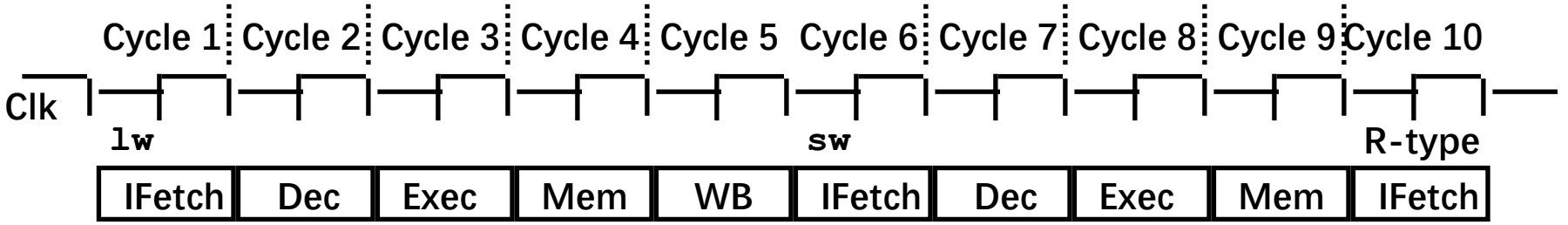
解决方案1：多周期完成一条指令



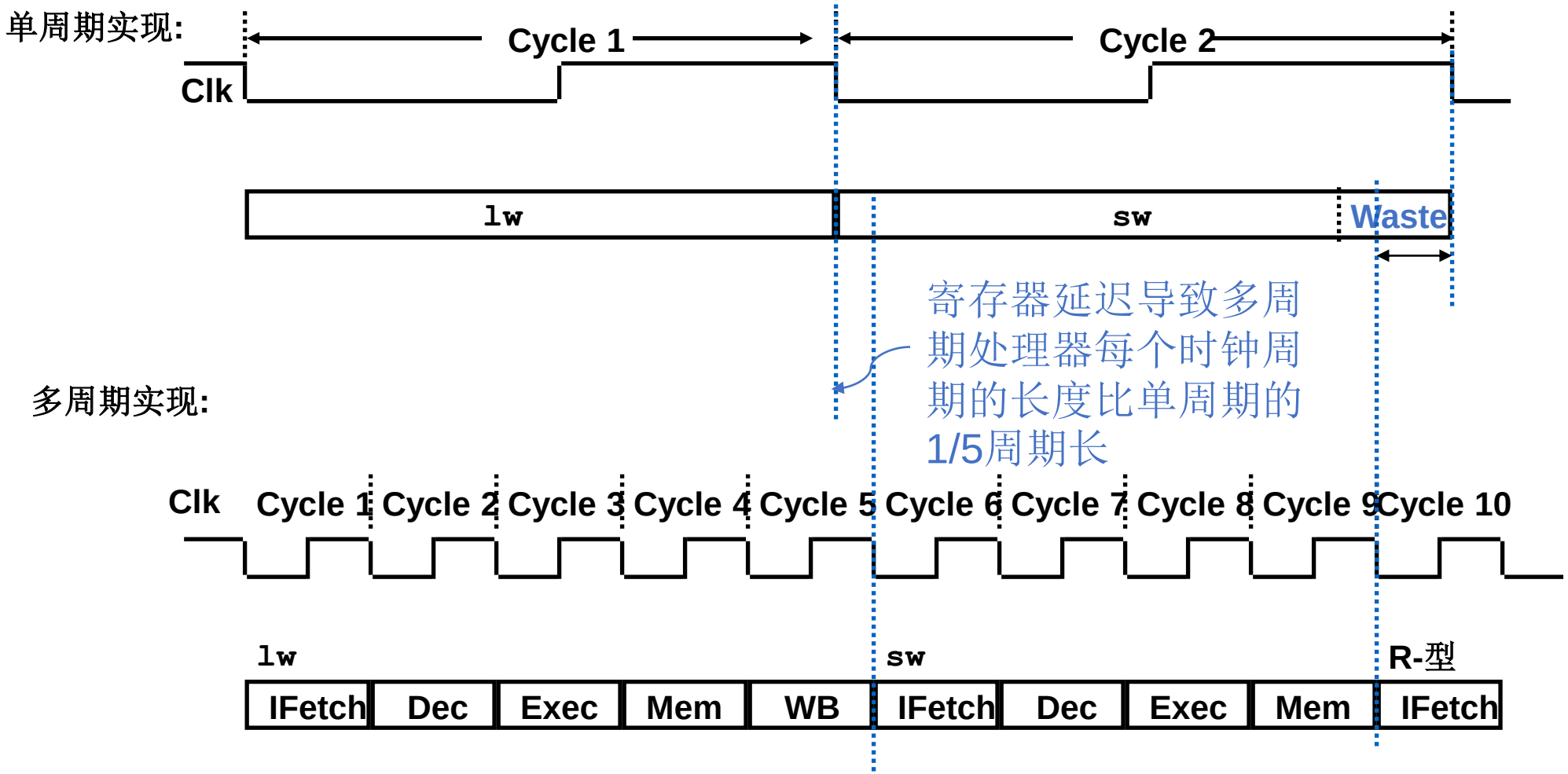
- 每条指令所花的时钟周期不相同
  - 时钟频率变快
  - 不同指令所需的时钟周期不同
- 一个部件在不同的周期可以重复使用
  - 不需要增加新部件

专用通路的多周期处理器

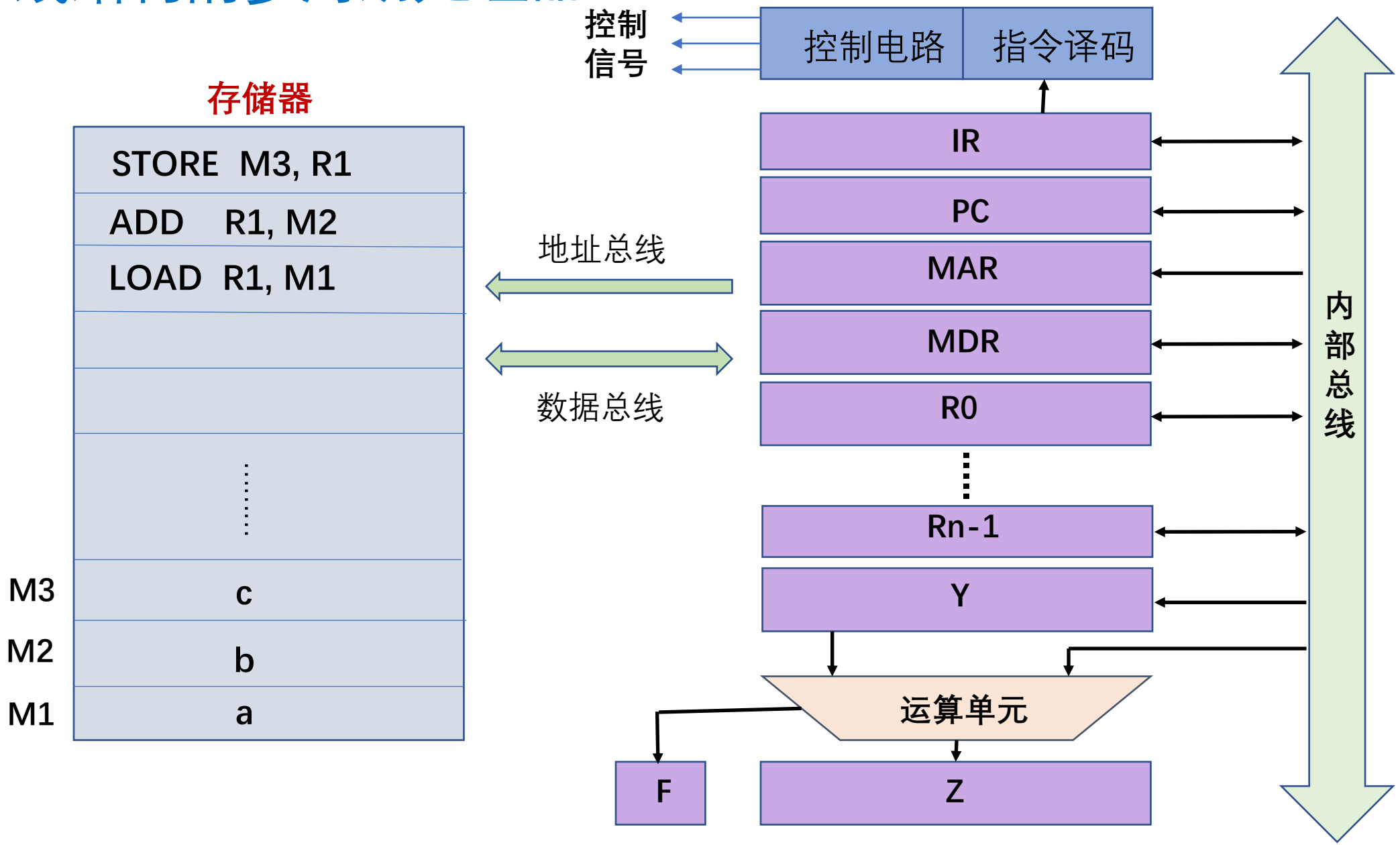




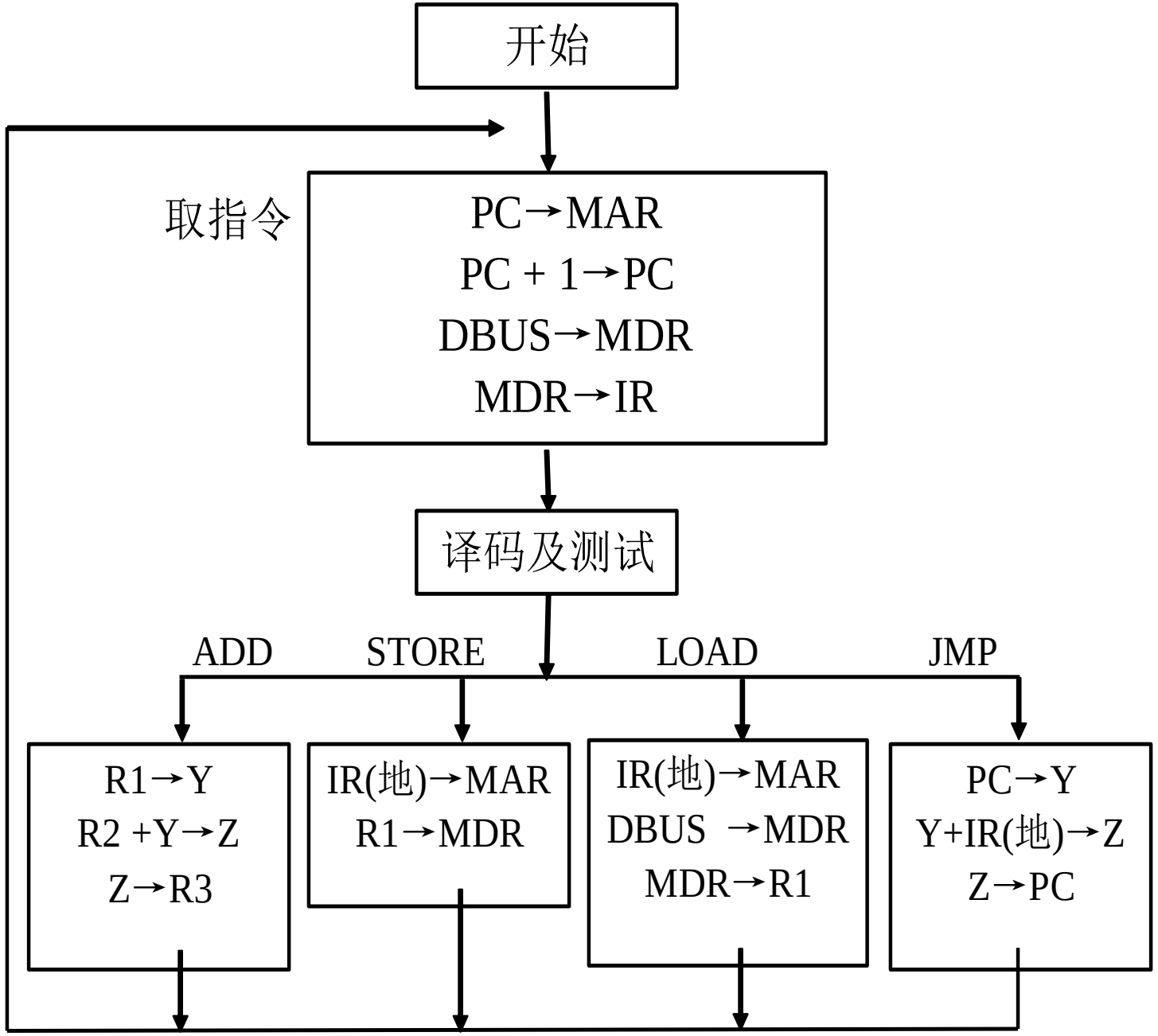
单周期 vs. 多周期 的时序



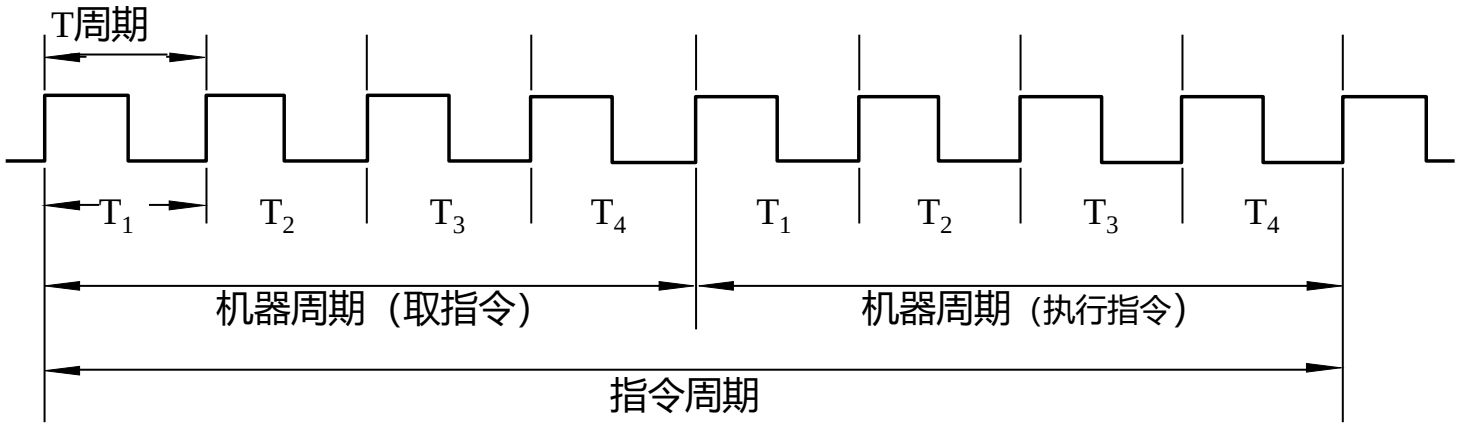
单总线结构的多周期处理器



单总线结构处理器的指令执行**流程**



指令周期的基本概念



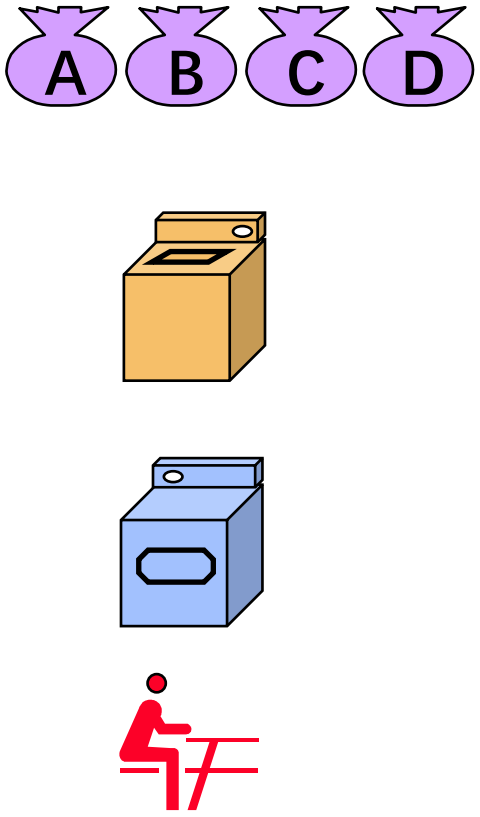
- 时钟周期：T,节拍脉冲
- 机器周期，CPU 周期：从内存读出一条指令的最短时间
- 指令周期：从内存取一条指令并执行该指令所用的时间。由若干个CPU周期组成。一个CPU周期又包含若干个时钟周期（节拍脉冲）

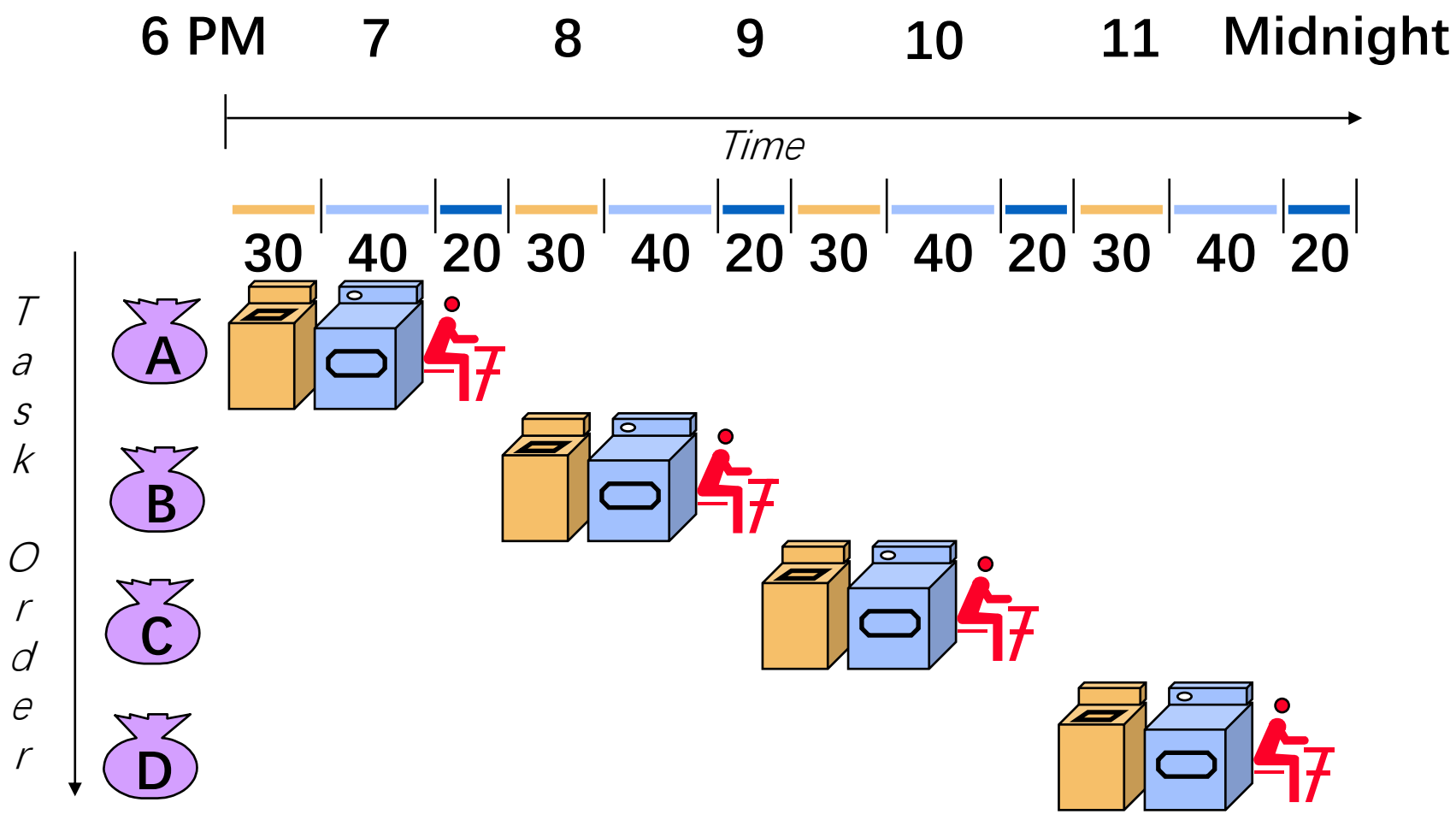
小结

- 分析了单周期处理器的缺点和优点
- 介绍了多周期处理器的特点
- 接下来：
  - 多周期处理器的进化：流水线处理器

举例：洗衣房

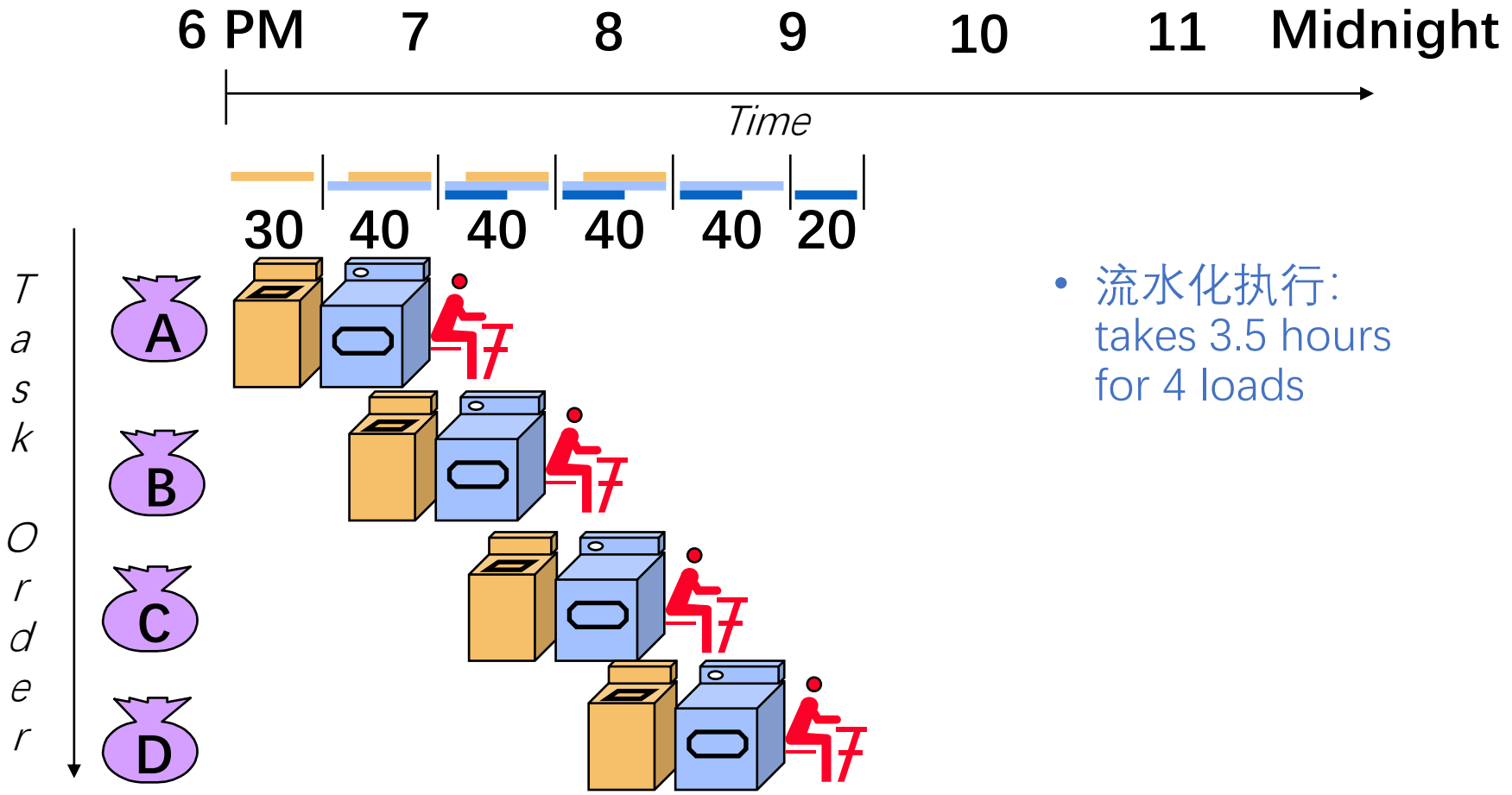
- Washer takes 30 minutes
- Dryer takes 40 minutes
- Folder takes 20 minutes





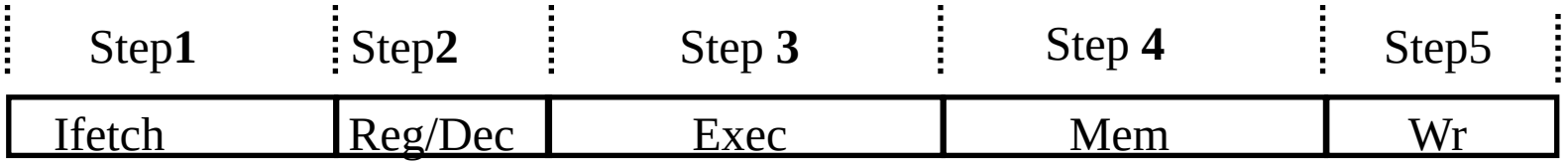
- 顺序执行， 4个任务共花费6小时

流水化的洗衣房: 尽快开始工作



- 流水化执行: takes 3.5 hours for 4 loads

回顾：Load指令的五个阶段　lw rt, rs, imm16



- Ifetch: 取指令 fetch instruction, PC+4

  Instruction MEM、Adder（加法器）

- Reg/Dec : 读寄存器、指令译码 Register Reading, Decoding

  Reg（read)、decoder（译码器）

- Exec: 计算存储器地址 Computer MEM Add

  Extender（立即数扩展）、ALU（运算单元）

- Mem ：读数据存储器 read from Mem

  Data Mem

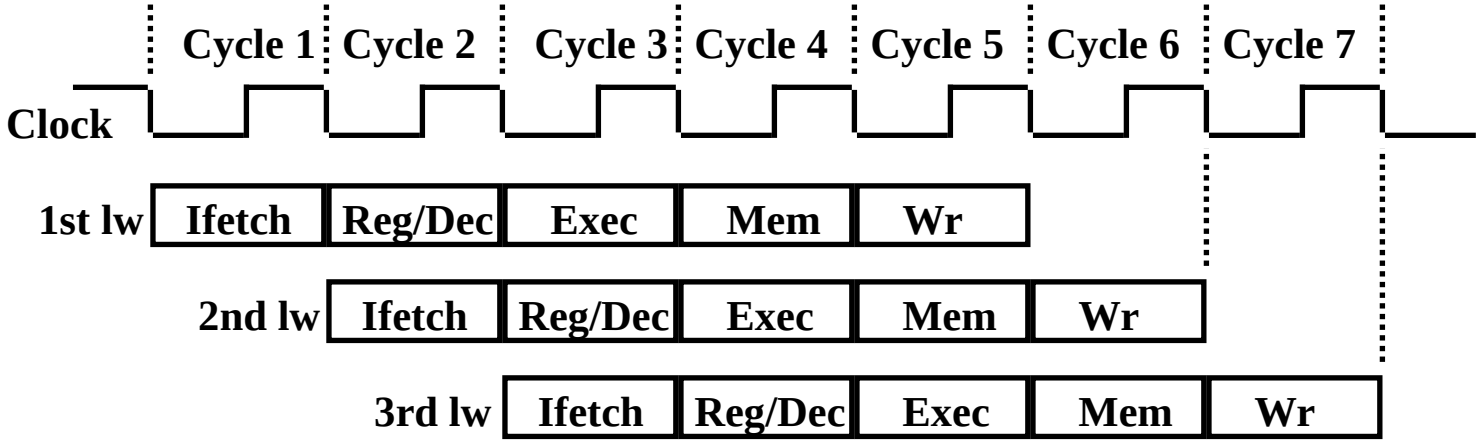
- Wr: 写寄存器 Write to Reg

  Reg (write)

易于流水化：

每一个阶段使用不同的资源（寄存器的读和写一个并行执行，后面再讨论）

将Load指令的执行流水化

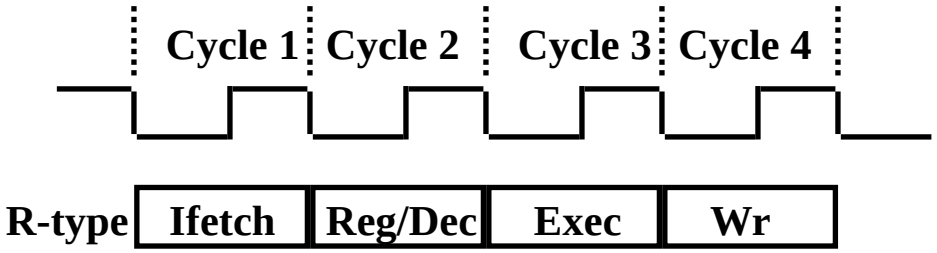


- 在数据通路上有五个功能单元:
  - 指令存储器: Instruction Memory： Ifetch （取指令）阶段
  - 寄存器文件的读端口 (bus A and busB)： Reg/Dec （读寄存器/译码）阶段
  - 算术逻辑运算单元 ALU: Exec（执行）阶段
  - 数据存储器 Data Memory： Mem（访存）阶段
  - 寄存器文件的写端口(bus W)： Wr（写结果） 阶段

MIPS 流水化

- 容易流水化
  - 1. 所有指令等长 (32 bits)
    - 可以在第一阶段内完成取指， 第二阶段完成译码
  - 2. 指令格式少， （三种）
    - 都可以在第二段读寄存器
  - 3. 只有loads and stores指令能访问存储器
    - 都可以在第三阶段计算存储地址
  - 4. 每条指令最多写一个结果
    - 在最后阶段完成，例如在MEM阶段 或 WB阶段写结果

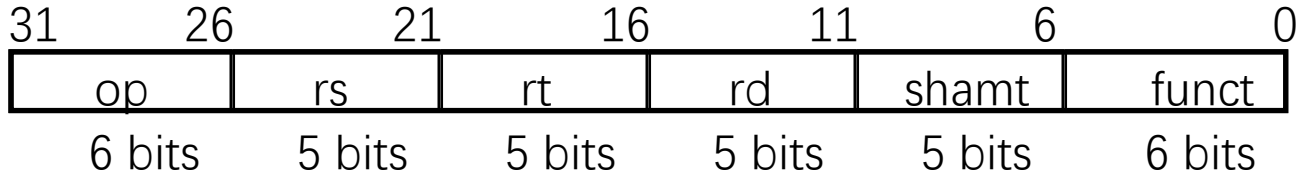
R型指令： 只需要4阶段



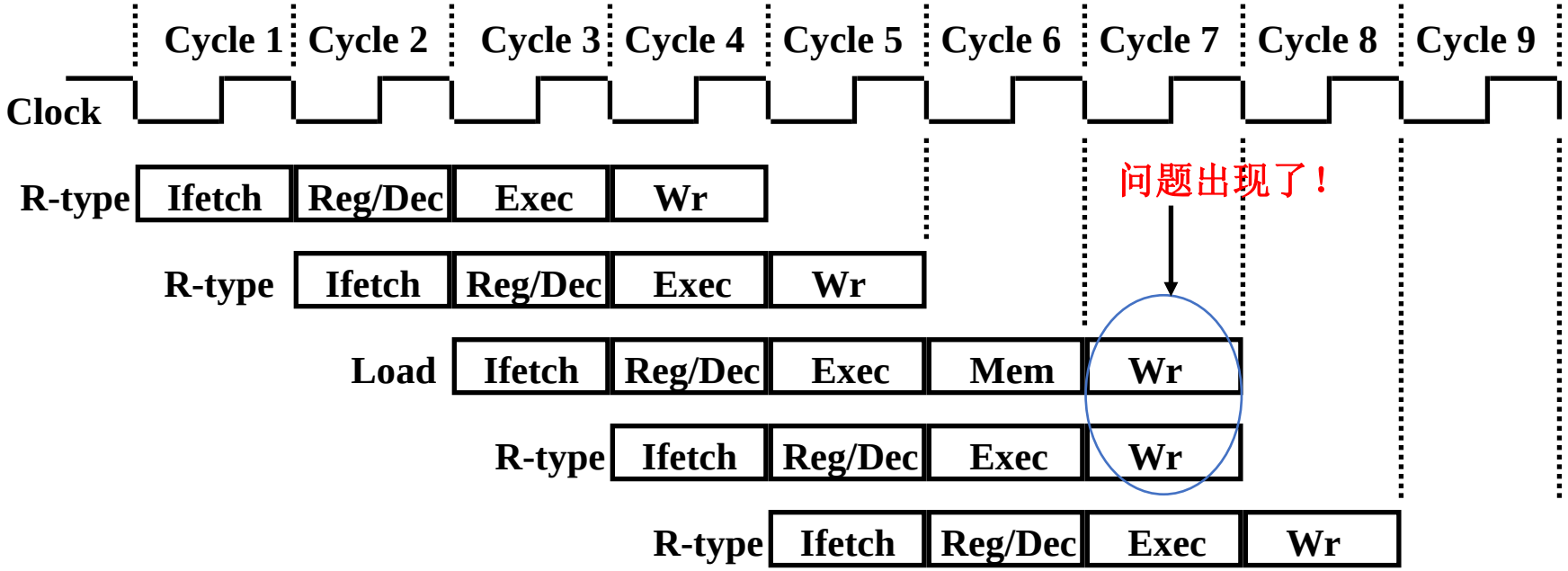
ADD and subtract
add rd, rs, rt
sub rd, rs, rt

- **Ifetch: 从指令存储器中取指令**
- **Reg/Dec: 取寄存器内容 、指令译码**
- **Exec: ： 使用ALU对从两个寄存器中取出的操作数进行运算**
- **Wr: 将 ALU 结果写入寄存器文件**

R型指令格式：



R型指令和 Load指令流水执行



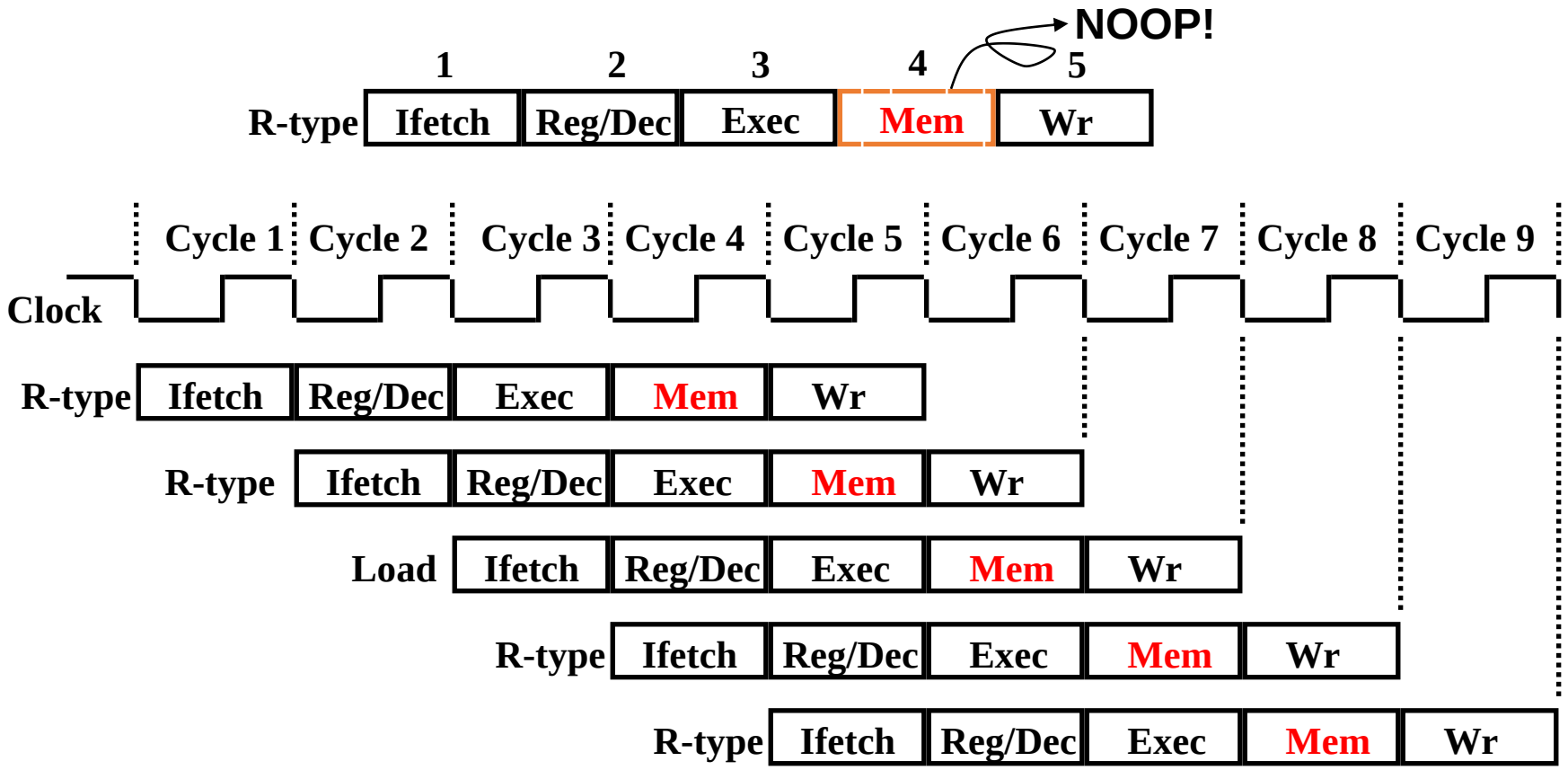
- 出现了资源冲突:

  两条指令要在同一个时间段使用寄存器写端口

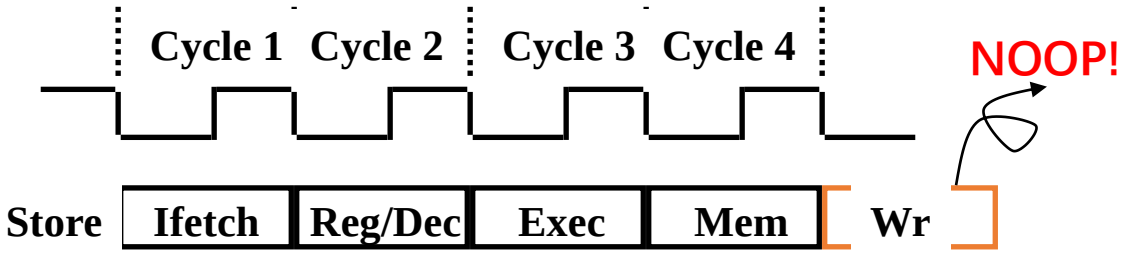
  只有一个寄存器写端口

解决方案：每条指令都有相同的流水段数

- R型指令的 Write 阶段推后一周期:
  - 写寄存器推迟到第五段
  - MEM 段是一个空（NOOP）操作.



Store指令

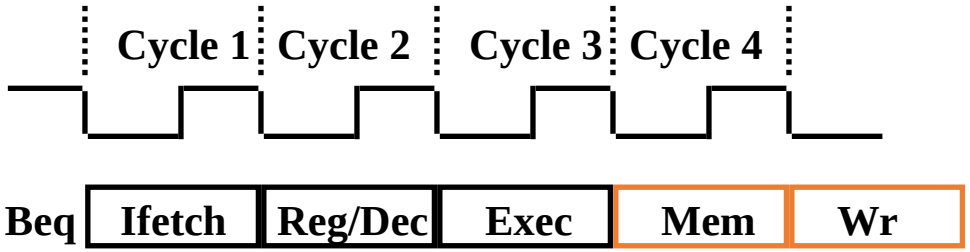


STORE指令：
sw rt, rs, imm16

- Ifetch：取指令
  - Fetch the instruction from the Instruction Memory
- Reg/Dec:：取寄存器内容 、指令译码
- Exec:：计算存储器地址
- Mem：将数据写入数据存储器

加一个空阶段，让所有指令的执行段数保持一致

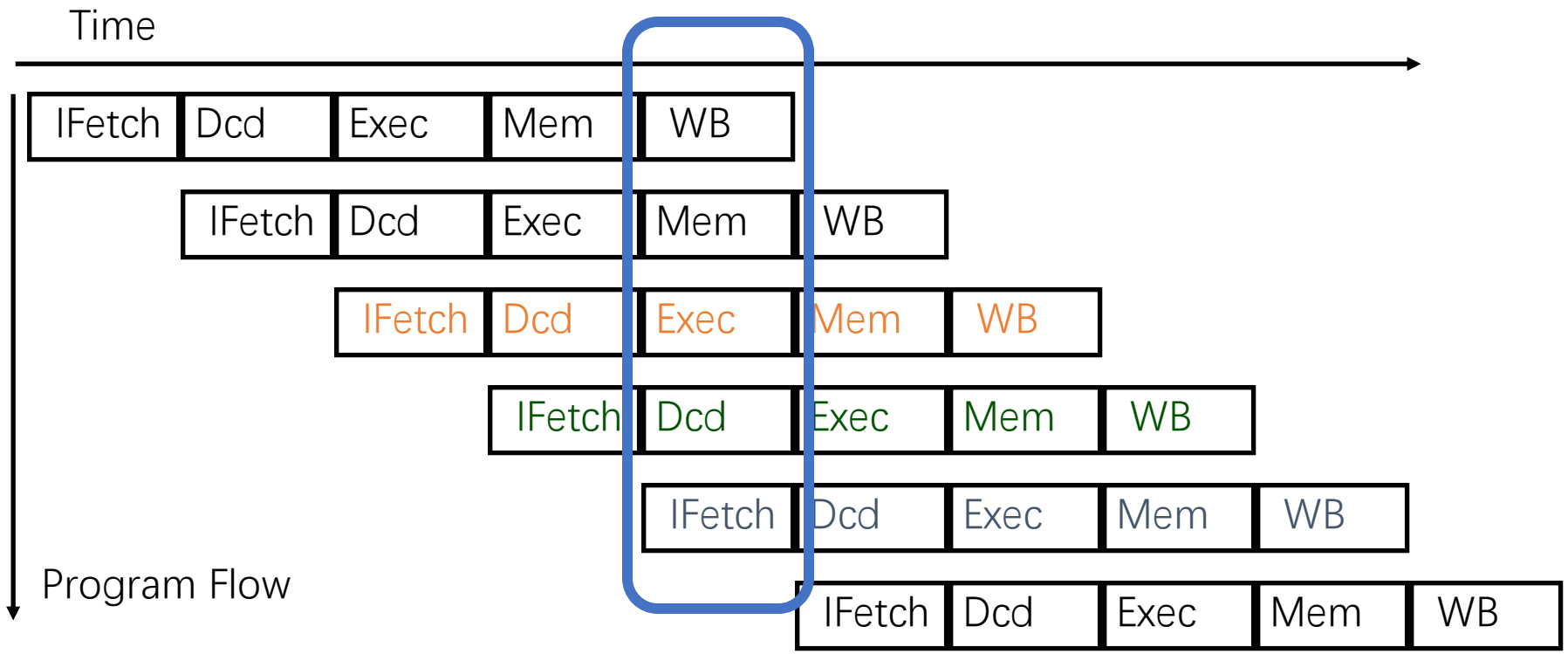
Beq指令



BRANCH
beq rs, rt, imm16

- Ifetch:取指令
- Reg/Dec:取寄存器内容 、指令译码
- Exec:
  - 比较两个操作数,
  - 计算转移目标地址
- Mem:
  - 将 计算好的目标地址送给 PC 输入端
- Wr : 空操作

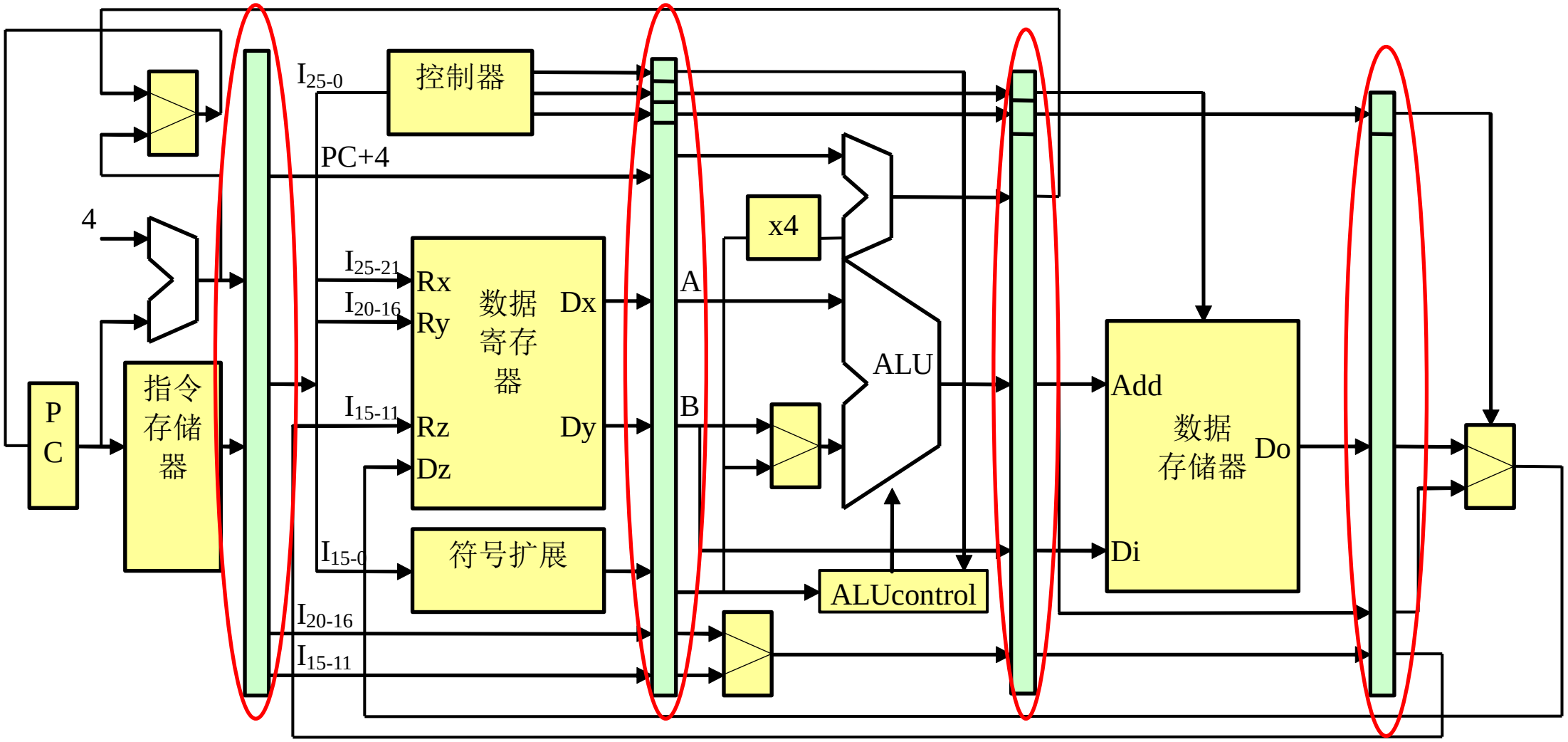
理想的指令流水线



理想情况下，每一个周期
完成一条老的指令；
开始一条新的指令。达到CPI=1；
CPI：完成一条指令所花的周期数

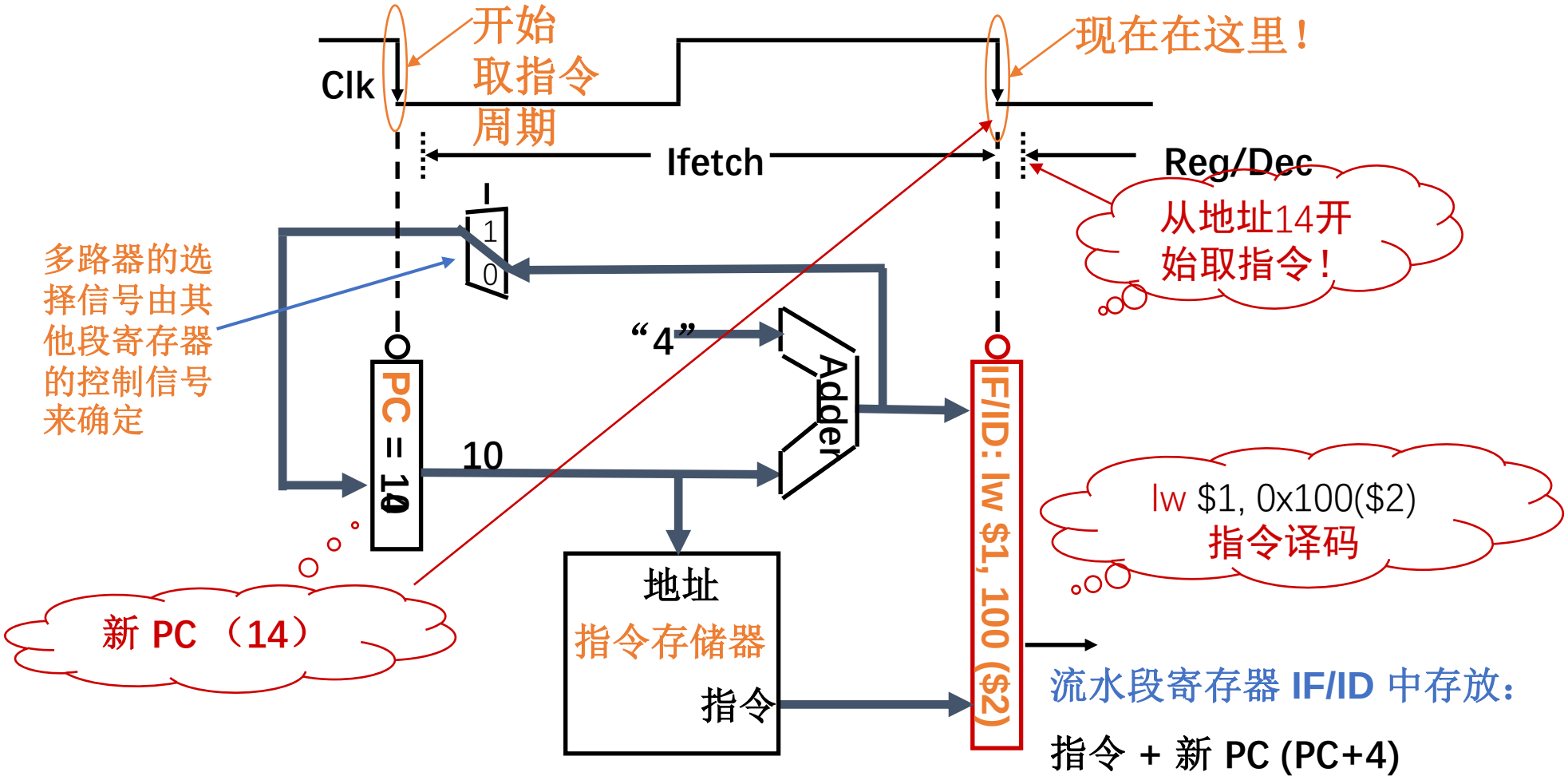
指令流水线的实现

基本指令流水线

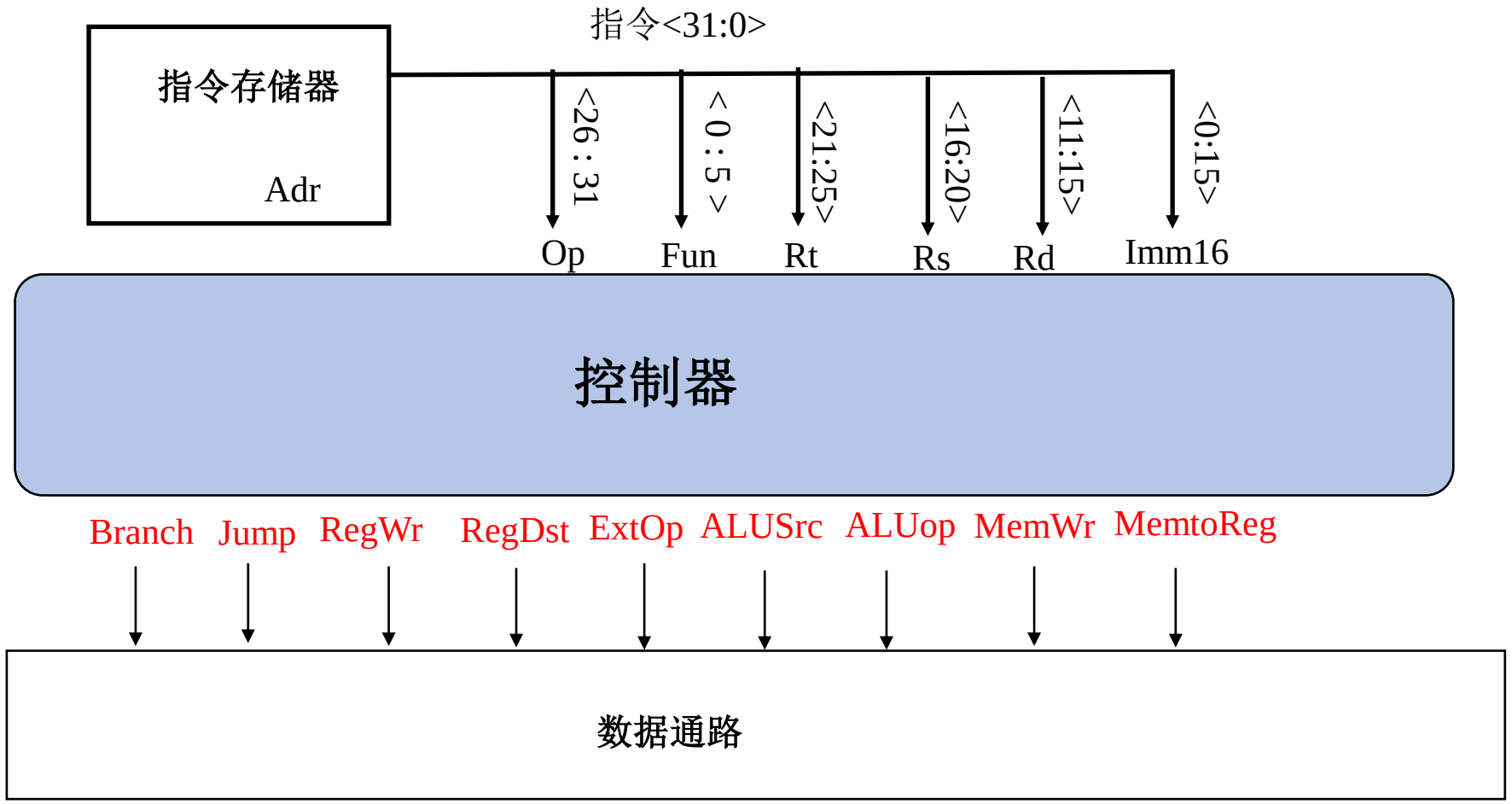

I_{25-0}
控制器
PC+4
4
I_{25-21}
I_{20-16}
I_{15-11}
I_{15-0}
Rx
Ry
Rz
Dz
数据寄存器
Dx
Dy
符号扩展
I_{20-16}
I_{15-11}
P C
指令存储器
A
B
x4
ALU
ALUcontrol
Add
数据存储器
Do
Di

取指令流水段

- 10: : lw $1, 0x100($2)

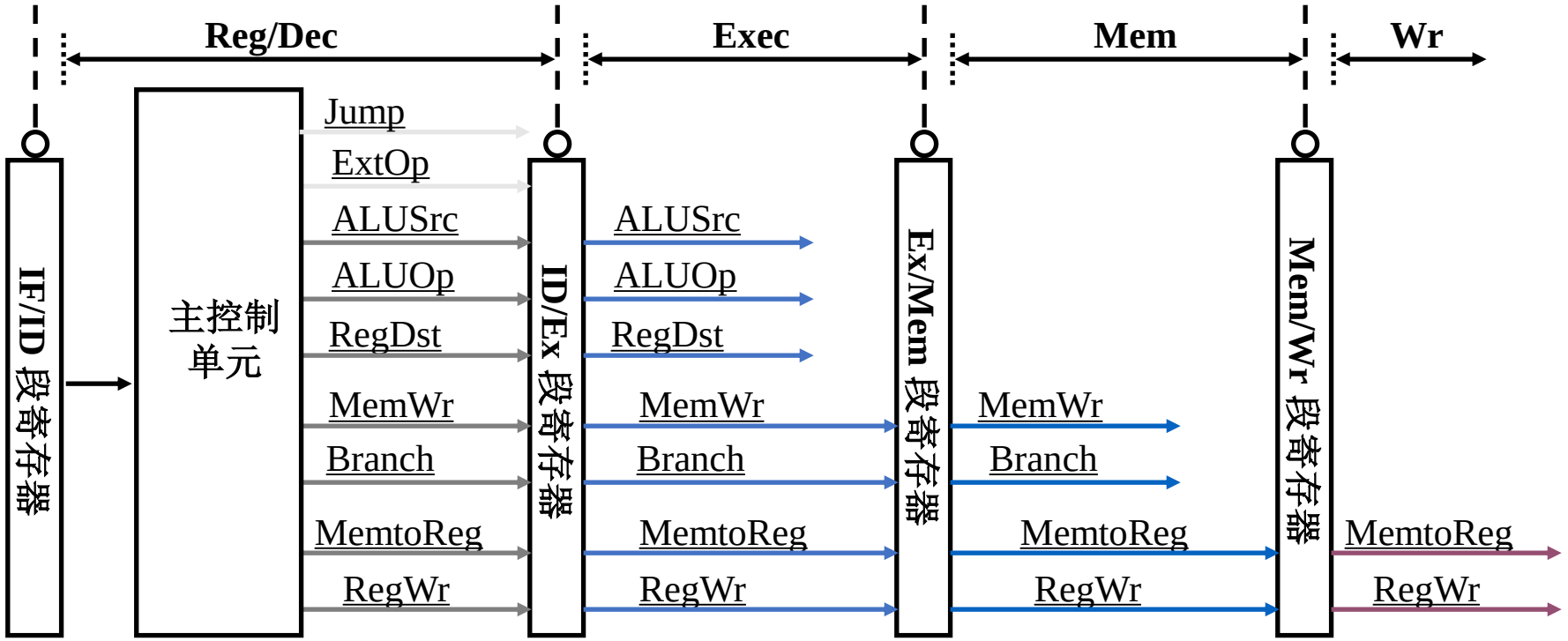


回顾：单周期处理器控制信号的生成



控制信号的传递

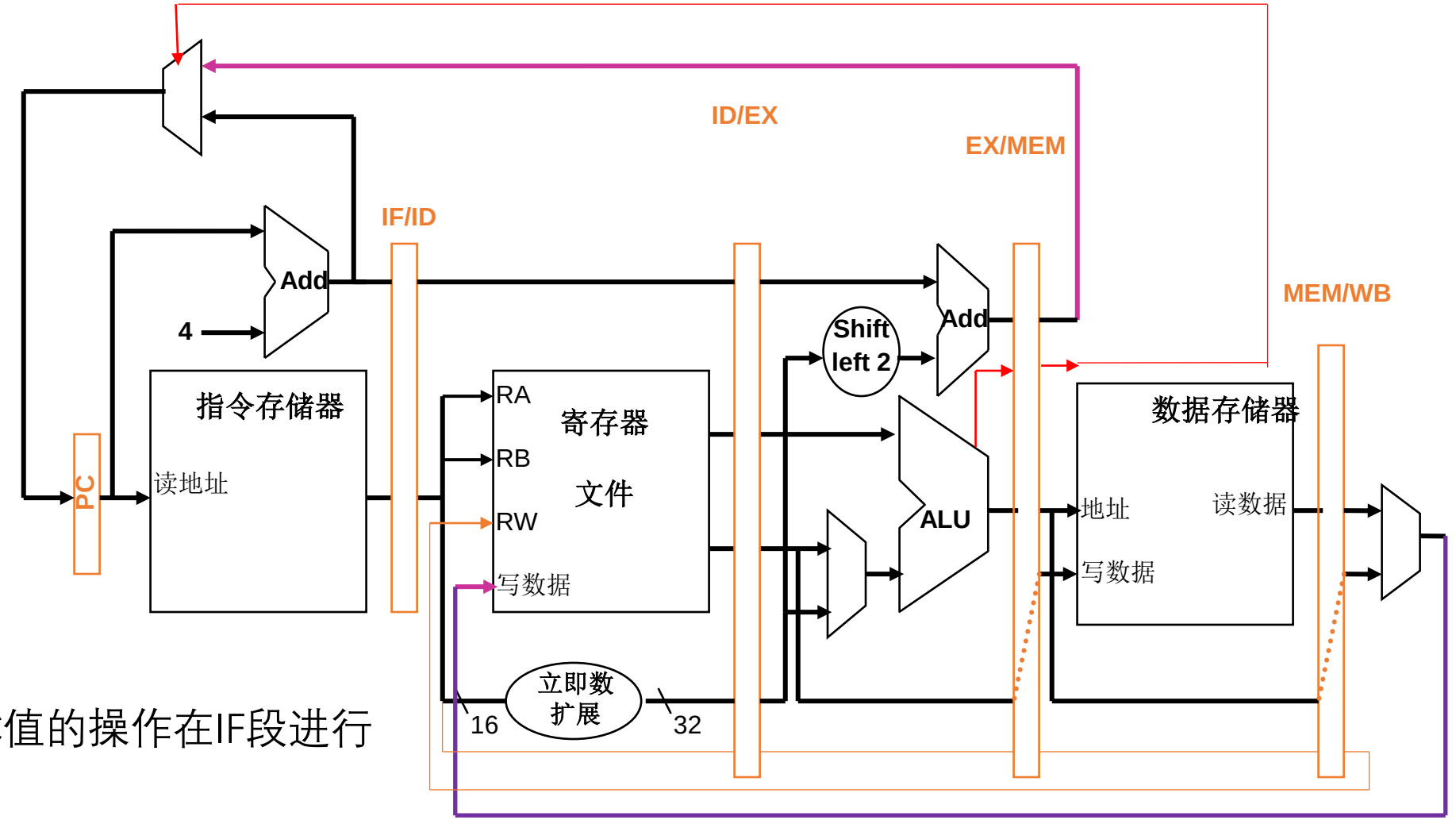
- 主控制单元在译码段（ Reg/Dec ）产生所有控制信号
  - Exec 段需要的控制信号，在一周期后使用
  - Mem 段需要的控制信号，在两周期后使用
  - Wr 段需要的控制信号，在三周期后使用



流水线各阶段所需的控制信号

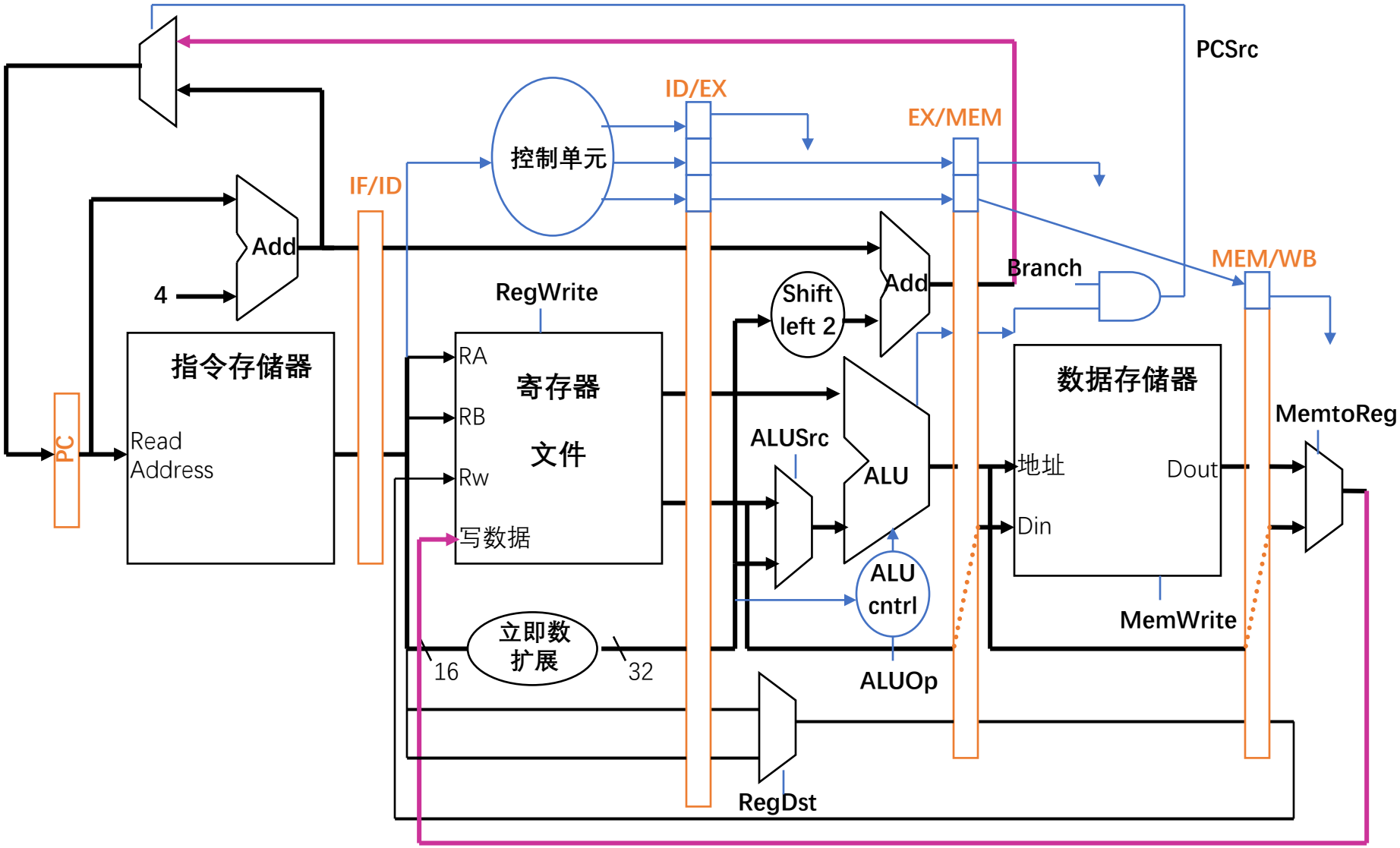
- Ifecth （取指段）：PC 不需要控制信号
- Dec/Reg（译码）段不需要控制信号
  - ExtOp ： 1- 带符号位扩展；0- 无符号数，高位填零扩展
  - Jump: 是否跳转指令
- Exec （执行段）
  - ALUSrc ： 1- 来自于扩展器；0- 来自于 bus B
  - ALUOp: 用于控制ALU完成的功能
  - RegDst ： 1- Rd；0- Rt
- Mem（访存段）
  - MemWr ： 1: 写，0: 其他
  - Branch ： 1: 转移，0: 其他
- Wr （写回段）
  - MemtoReg ： 1- 数据存储器的输出；0- ALU的输出
  - RegWr ： 1: 写寄存器，0:其他

转移指令如何设置PC

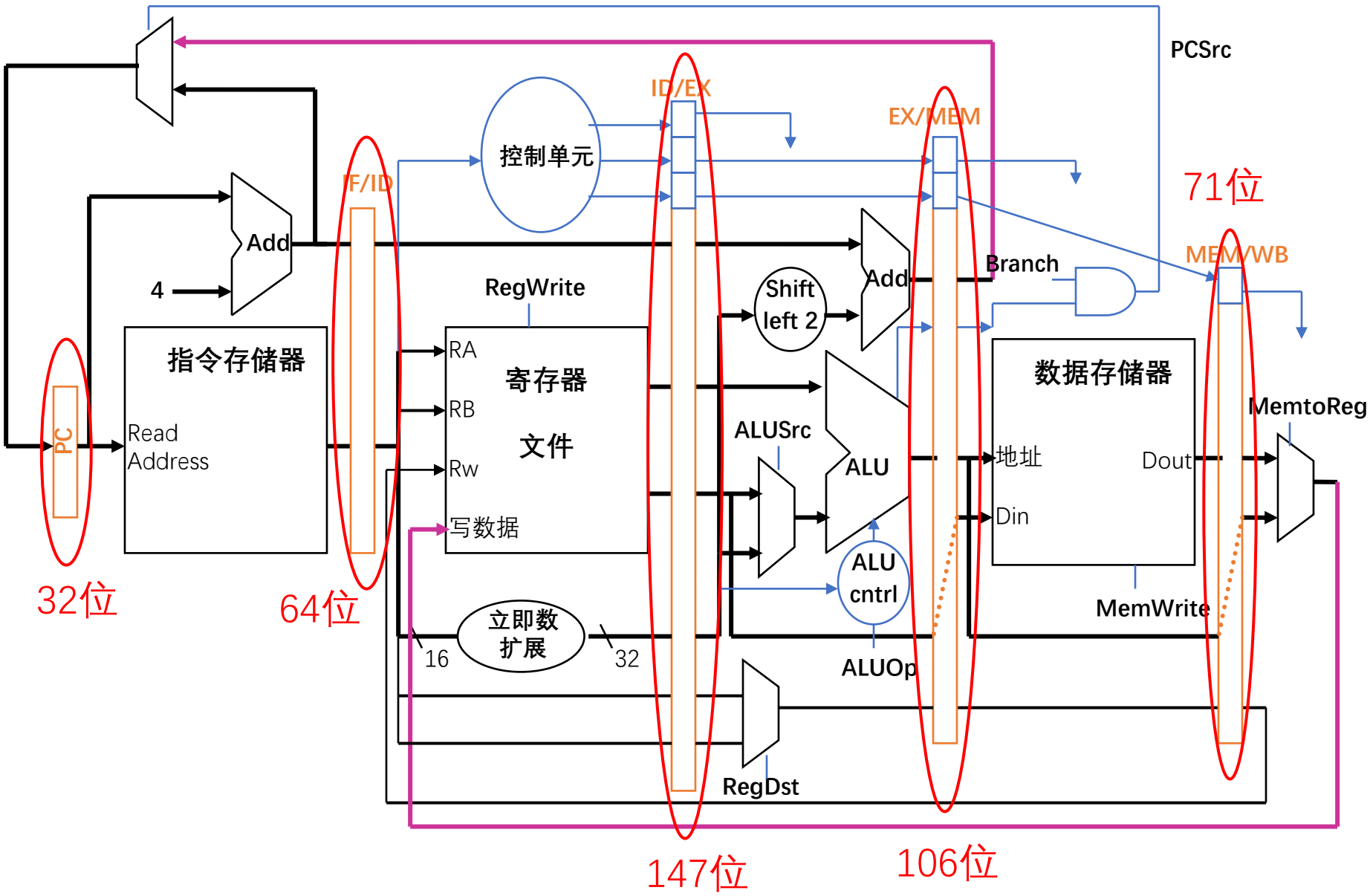


改变PC值的操作在IF段进行

五阶段流水线的实现


PCSrc
ID/EX
EX/MEM
MEM/WB
控制单元
IF/ID
Add
4
RegWrite
Shift
left 2
Add
Branch
指令存储器
PC
Read
Address
RA
RB
Rw
寄存器
文件
写数据
ALUSrc
ALU
数据存储器
地址
Din
Dout
MemtoReg
ALU
cntrl
MemWrite
立即数
扩展
16
32
ALUOp
RegDst

五阶段流水线：段寄存器


PCSrc
ID/EX
EX/MEM
71位
MEM/WB
控制单元
IF/ID
Add
4
RegWrite
Shift
left 2
Add
Branch
指令存储器
Read
Address
PC
RA
RB
Rw
寄存器
文件
写数据
ALUSrc
ALU
数据存储器
地址
Din
Dout
MemtoReg
ALU
cntrl
MemWrite
ALUOp
立即数
扩展
16
32
RegDst
32位
64位
147位
106位

小结

- 五阶段流水线处理器的实现
- 控制信号逐级传递
- 各段寄存器中存储的内容

谢谢！