计算机组织与系统结构

设计单周期控制

Designing Single Cycle Control

第八讲

程旭

2020.11.26

如何设计处理器： 循序渐进

1. 分析指令系统 => 数据通路 需求

   通过寄存器传输 描述 每条指令的意图

   针对ISA寄存器，数据通路必须具备必要的存储元件

   可能需要多个

   数据通路必须支持每种寄存器传输

2. 选择一组数据通路部件，建立时钟同步方法

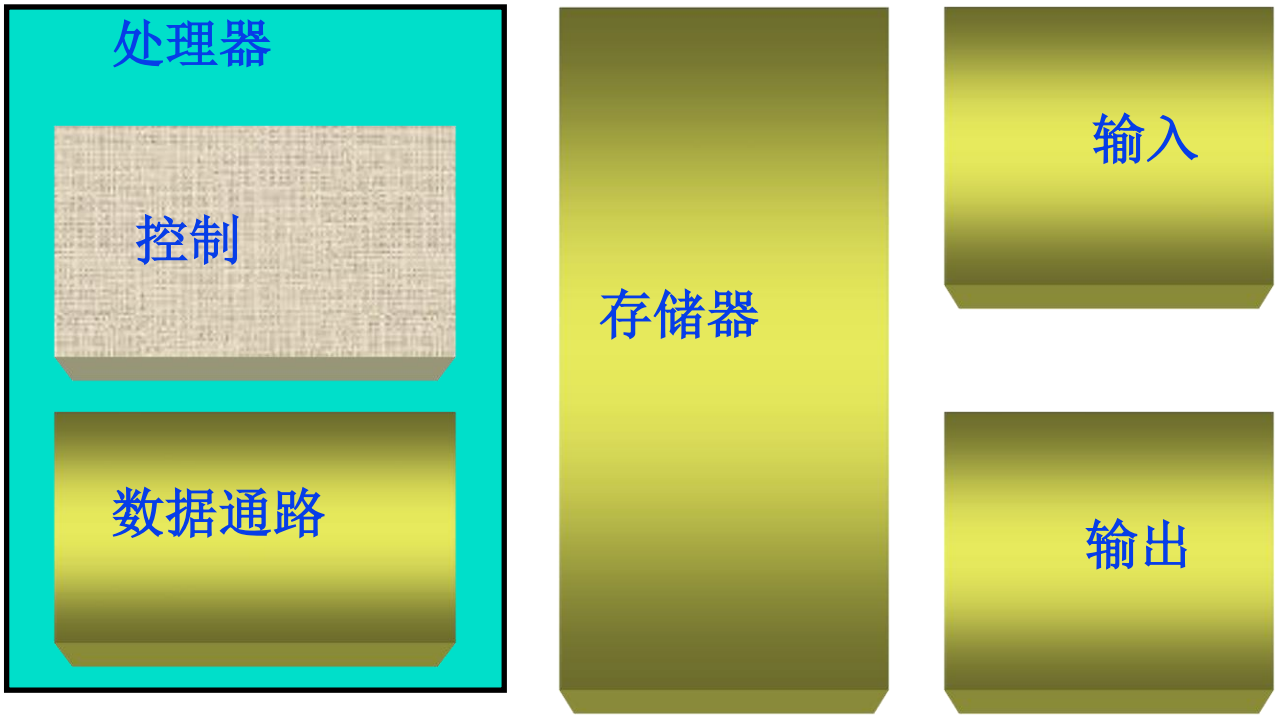
3. 根据需求， 组装 数据通路

4. 分析每条指令的实现，以确定如何设置影响寄存器传输的控制点

5. 装配 控制逻辑

教学目标：已经掌握的内容

- 计算机的五个基本部件

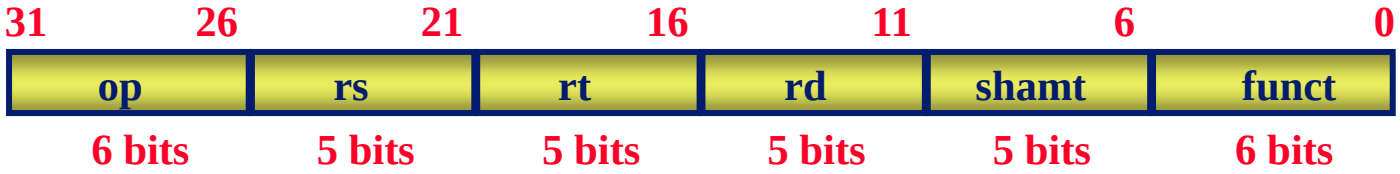


- 本讲主题:为单周期数据通路设计控制

本讲提纲

- 复习和介绍
- 寄存器-寄存器 & 或立即数指令的控制
- 装入、存储、转移和跳转的控制信号
- 建造一个局部控制器: ALU控制
- 主控制器
- 总结

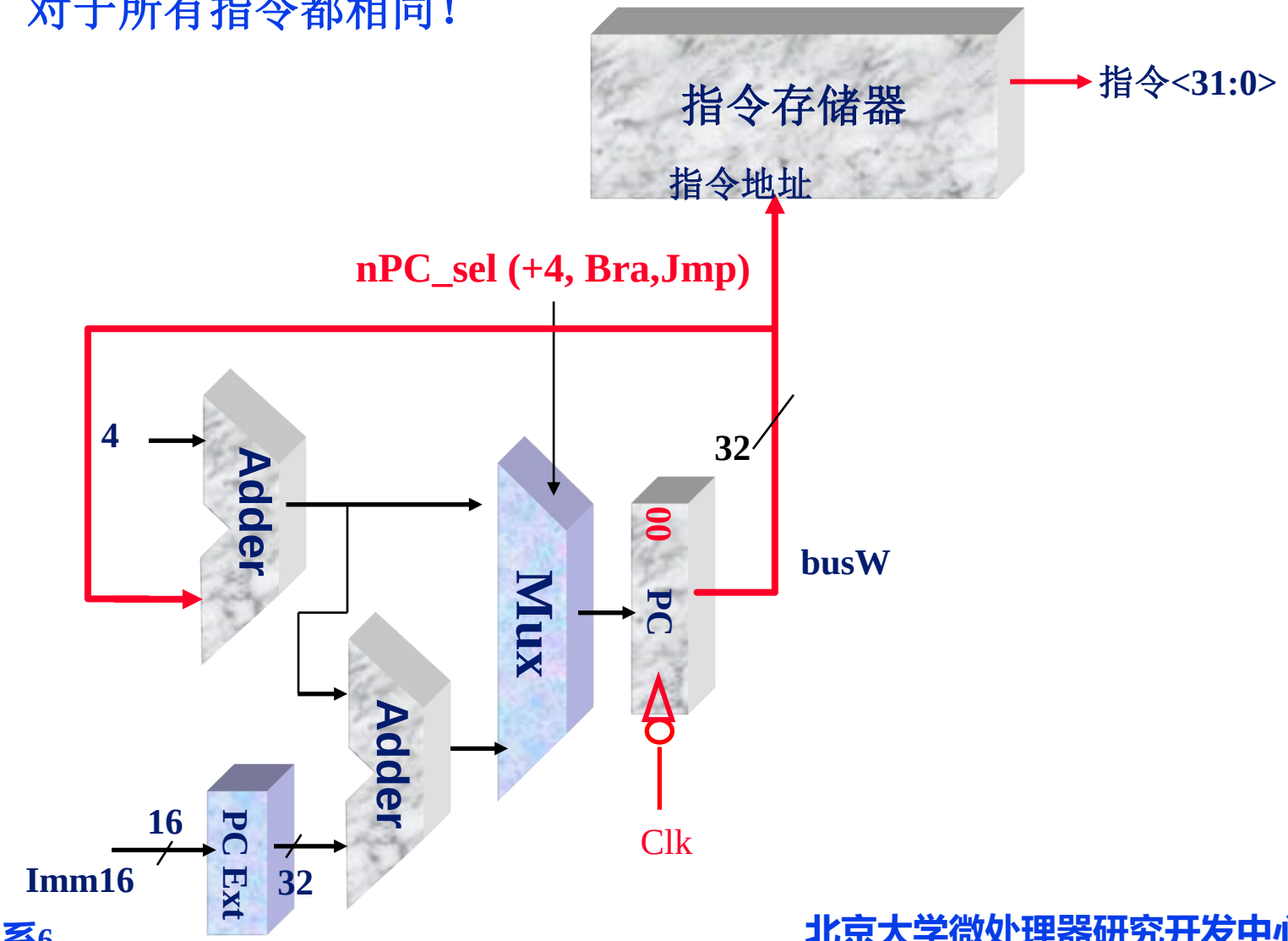
寄存器传输语言： 加法指令



- add rd, rs, rt
  - mem[PC] 从存储器中读取指令
  - R[rd] ⇐ R[rs] + R[rt] 实际操作
  - PC ⇐ PC + 4 计算下一条指令地址

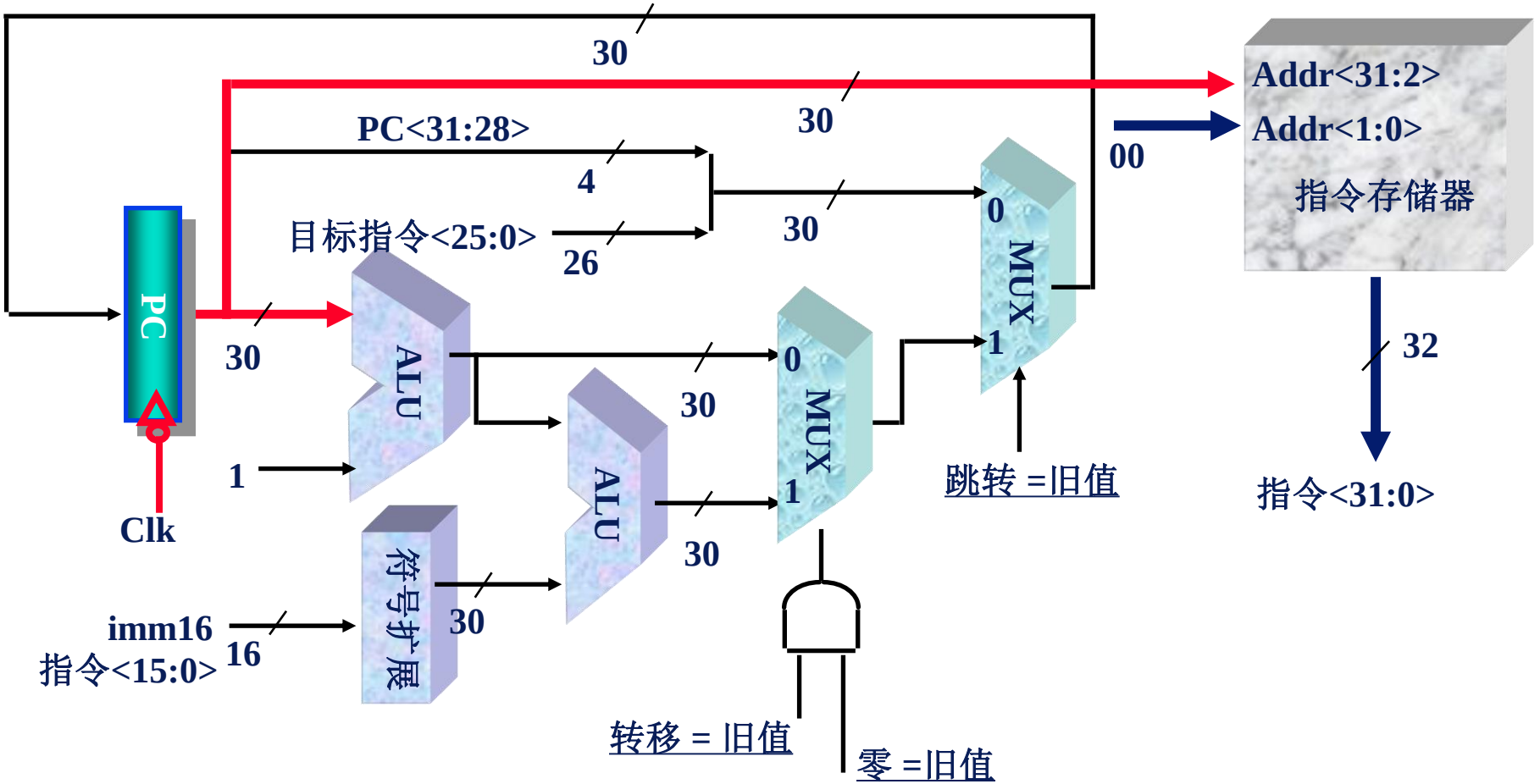
Add开始时的取指部件

- 从指令存储器中取出指令： 指令 ⇐ mem[PC]
  - 这一部分，对于所有指令都相同！

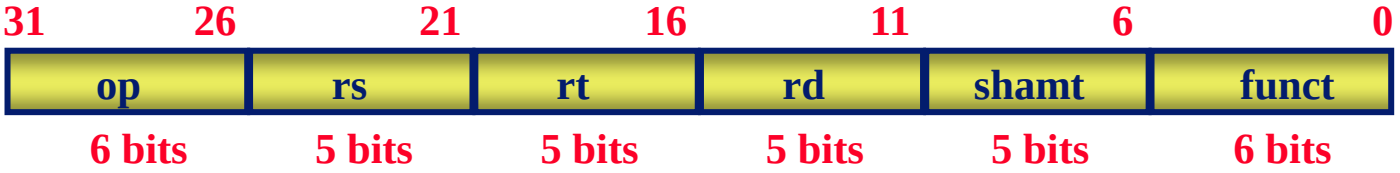


在加法/减法开始时的取指部件

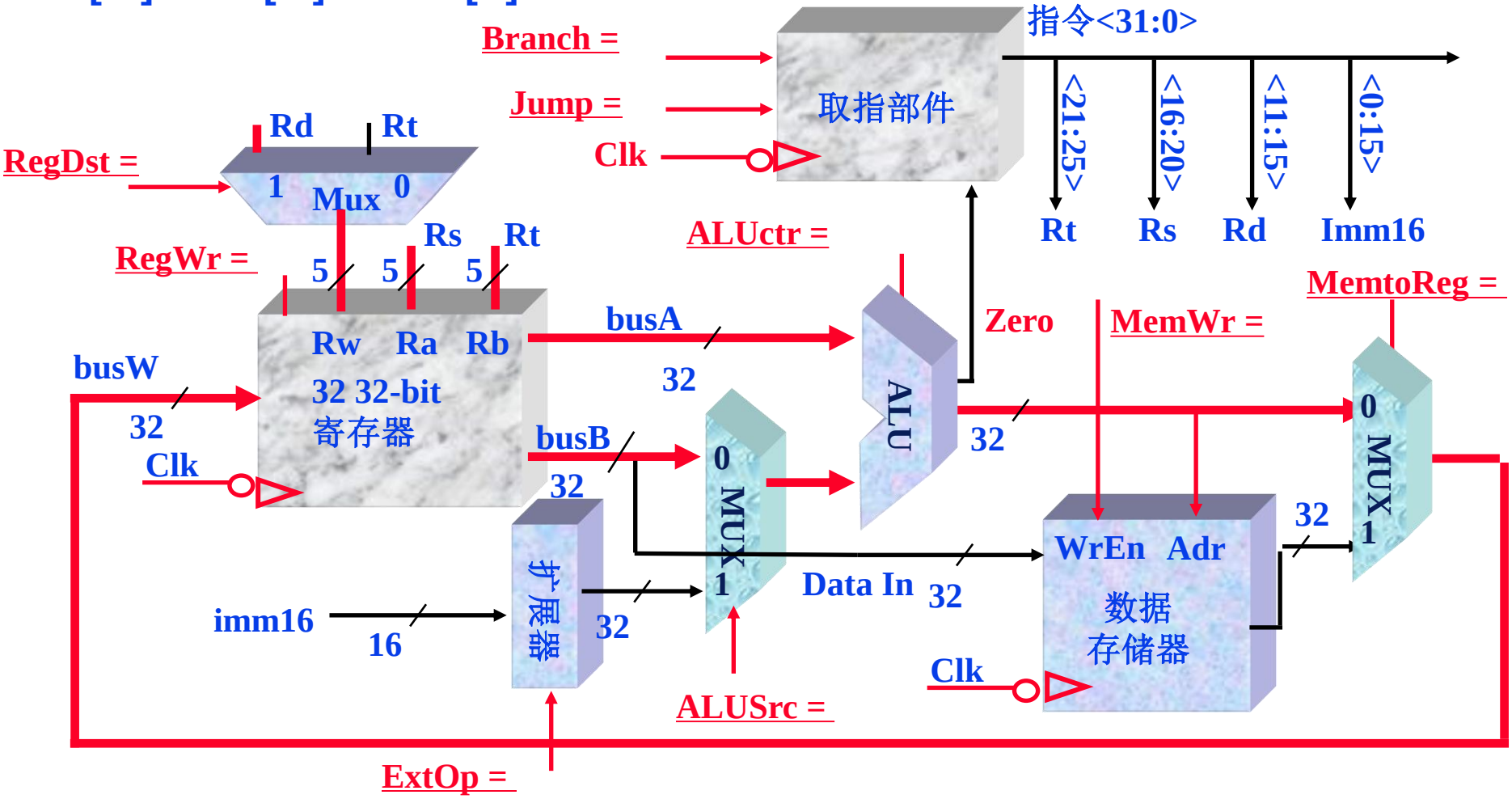
- 从指令存储器中取出指令: 指令 ⇐ mem[PC]
  - 每个指令的取指过程的这一部分都相同



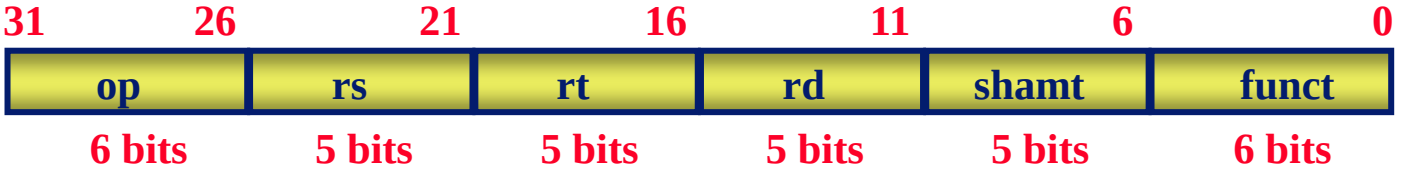
在加法和减法过程中的单周期数据通路



- R[rd] ⇐ R[rs] + / - R[rt]



在加法和减法过程中的单周期数据通路

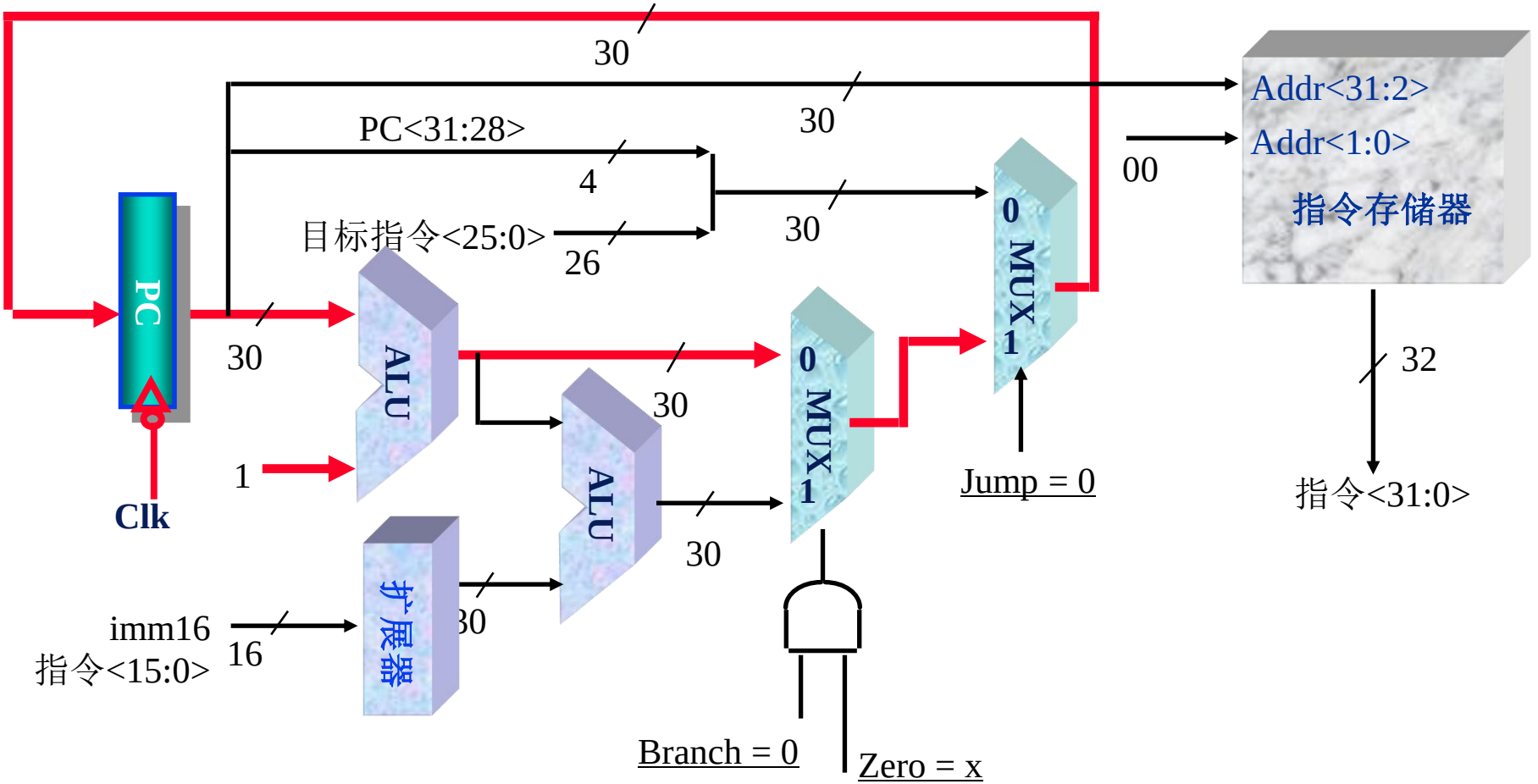


- R[rd] ⇐ R[rs] + / - R[rt]

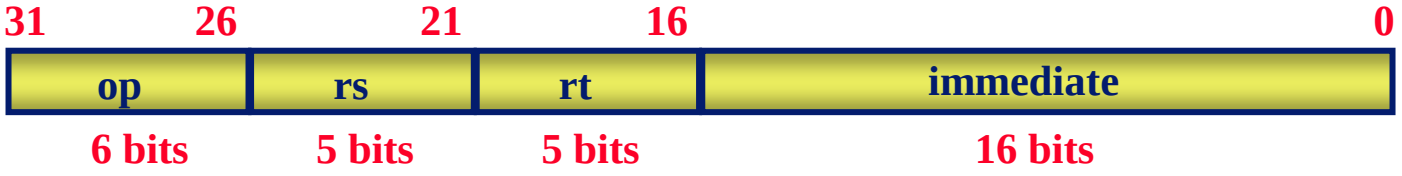
Branch = 0
Jump = 0
Clk
取指部件
指令<31:0>
<21:25>
<16:20>
<11:15>
<0:15>
Rt
Rs
Rd
Imm16
RegDst = 1
Rd
Rt
1
Mux
0
RegWr = 1
5
5
5
Rs
Rt
Rw
Ra
Rb
32 32-bit
寄存器
busW
32
Clk
busA
32
busB
32
ALUctr = add/sub
ALU
Zero
32
MemWr = 0
MemtoReg = 0
0
MUX
1
WrEn
Adr
数据
存储器
Data In
32
32
Clk
imm16
16
扩展器
32
ALUSrc = 0
ExtOp = x

在加法和减法结束时的取指部件

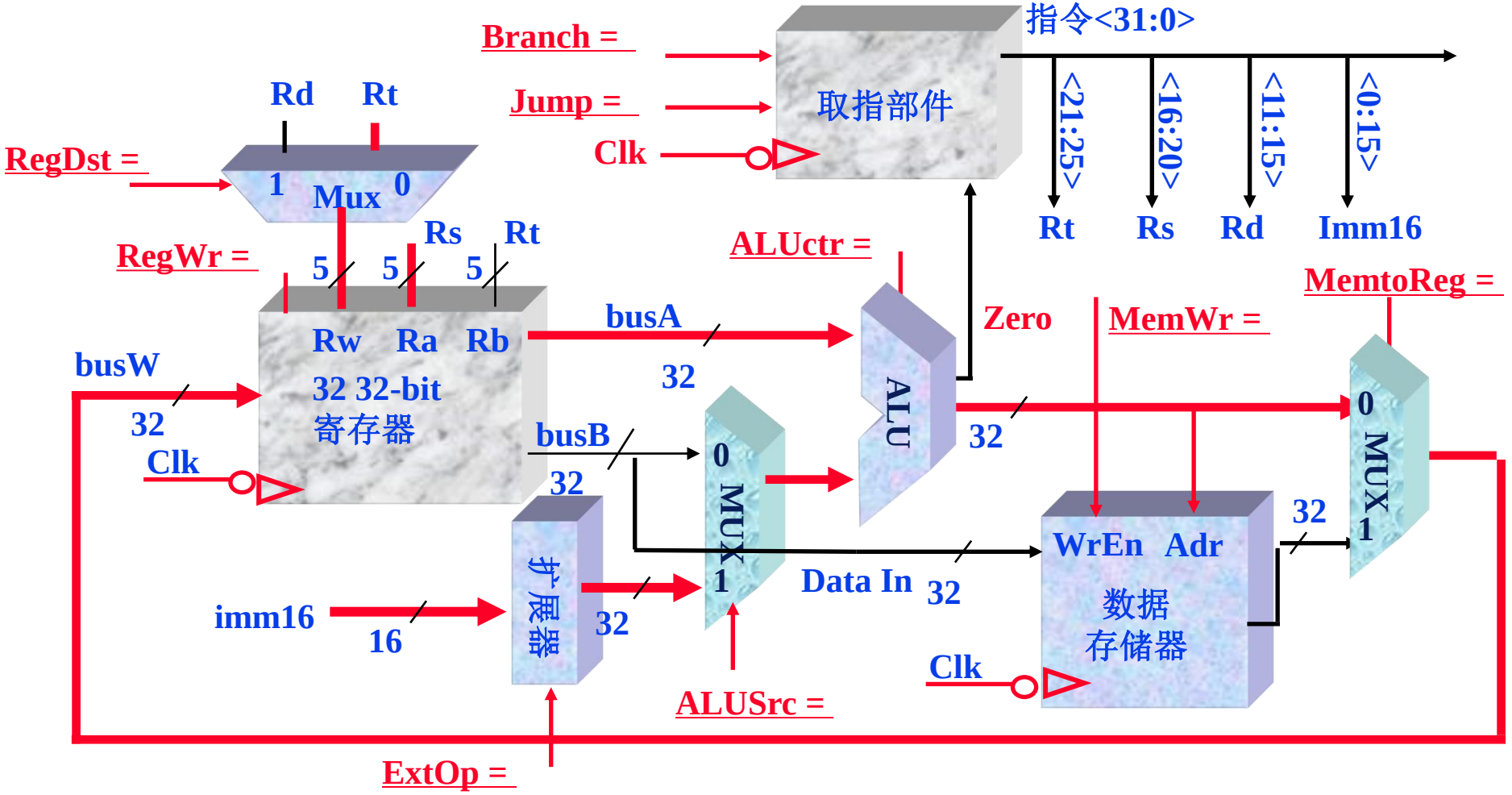
- PC ⇐ PC + 4
  - 除了转移和跳转之外，每条指令的这一部分相同



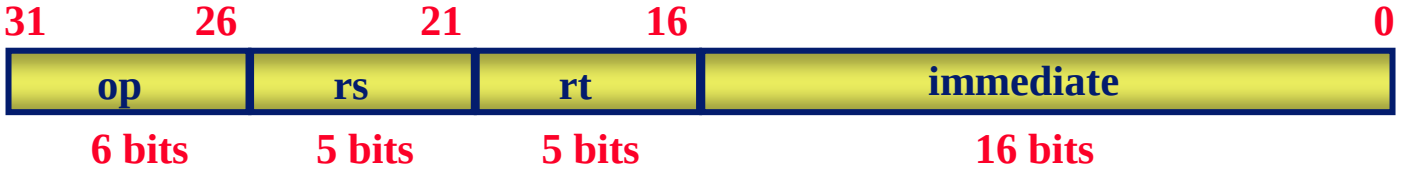
或立即数过程中的单周期数据通路



- R[rt] ⇐ R[rs] or ZeroExt[Imm16]

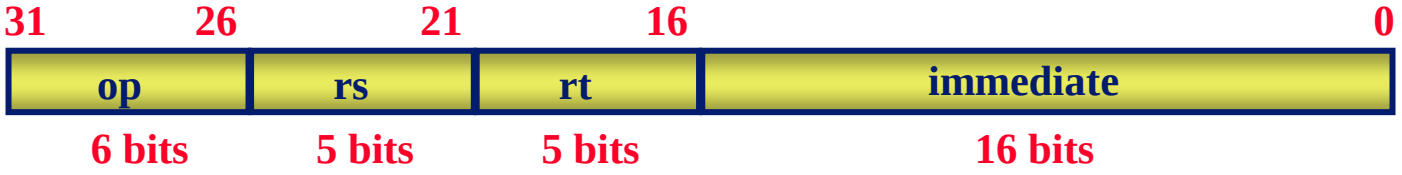


或立即数过程中的单周期数据通路



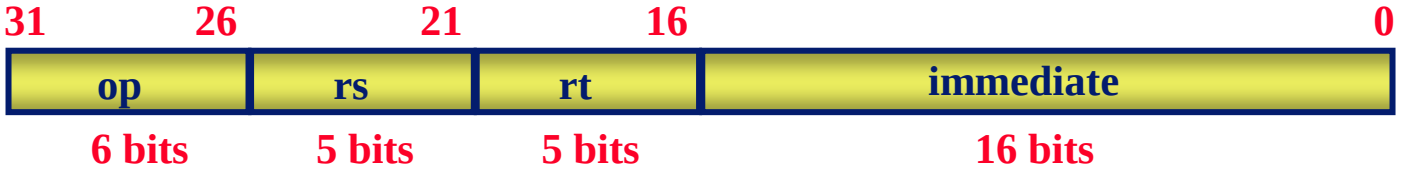
- R[rt] ⇐ R[rs] or ZeroExt[Imm16]

装入过程中的单周期数据通路



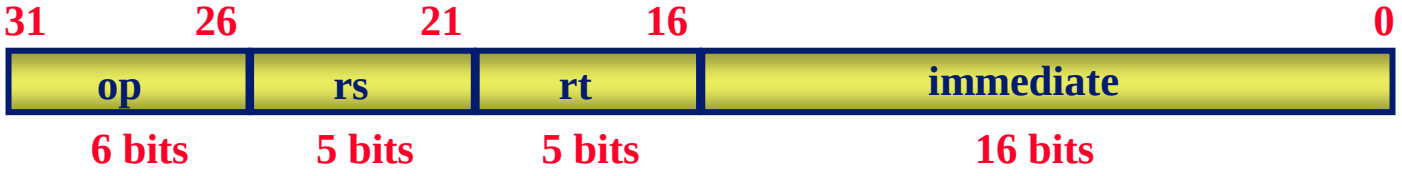
- R[rt] ⇐ Data Memory {R[rs] + SignExt[imm16]}

装入过程中的单周期数据通路



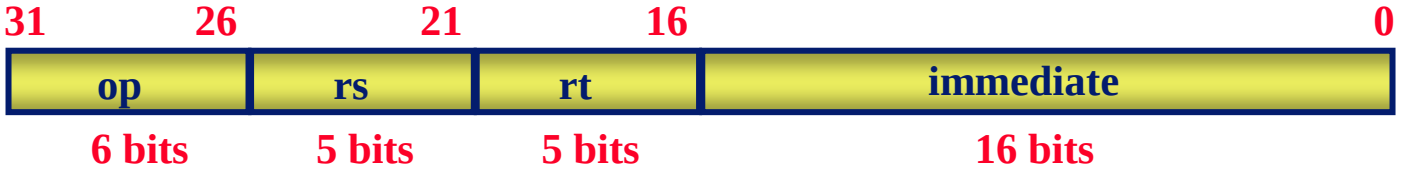
- R[rt] ⇐ Data Memory {R[rs] + SignExt[imm16]}

存储过程中的单周期数据通路



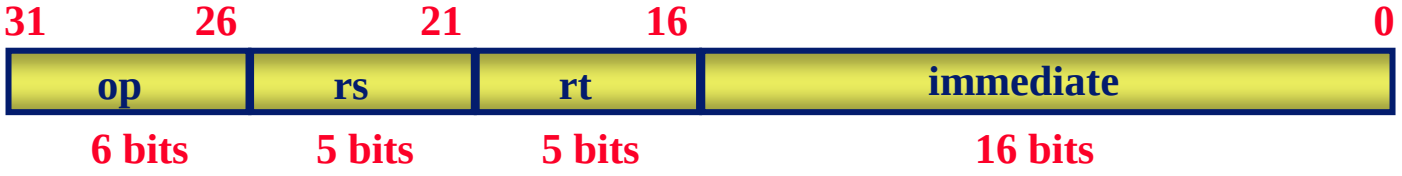
- Data Memory {R[rs] + SignExt[imm16]} ⇐ R[rt]

存储过程中的单周期数据通路

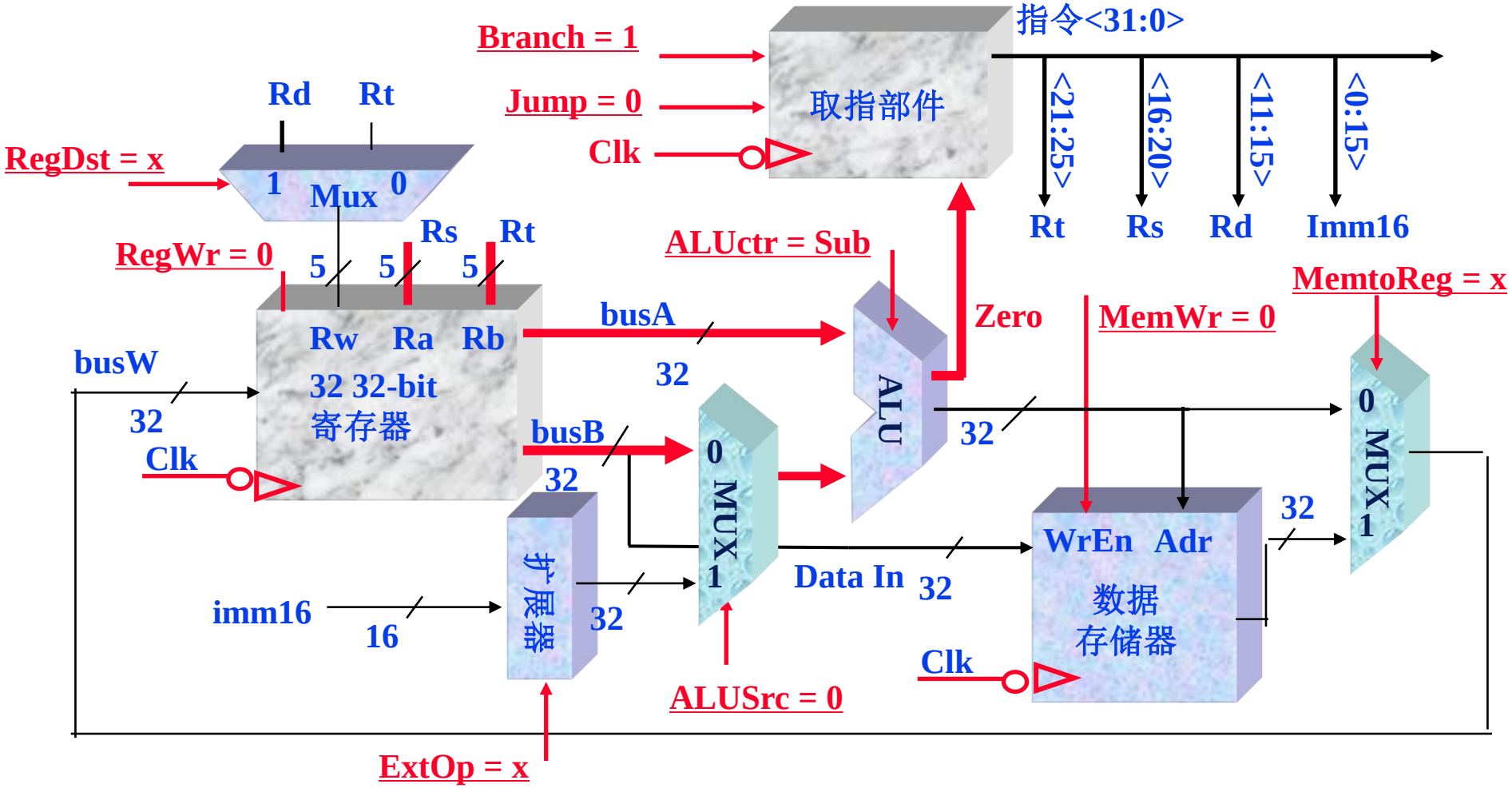


- Data Memory {R[rs] + SignExt[imm16]} ⇐ R[rt]

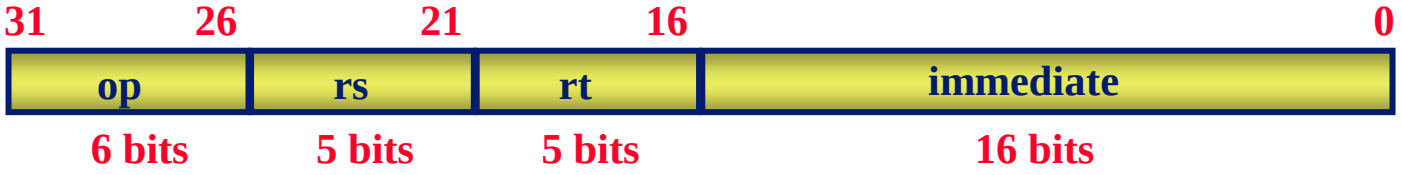
转移过程中的单周期数据通路



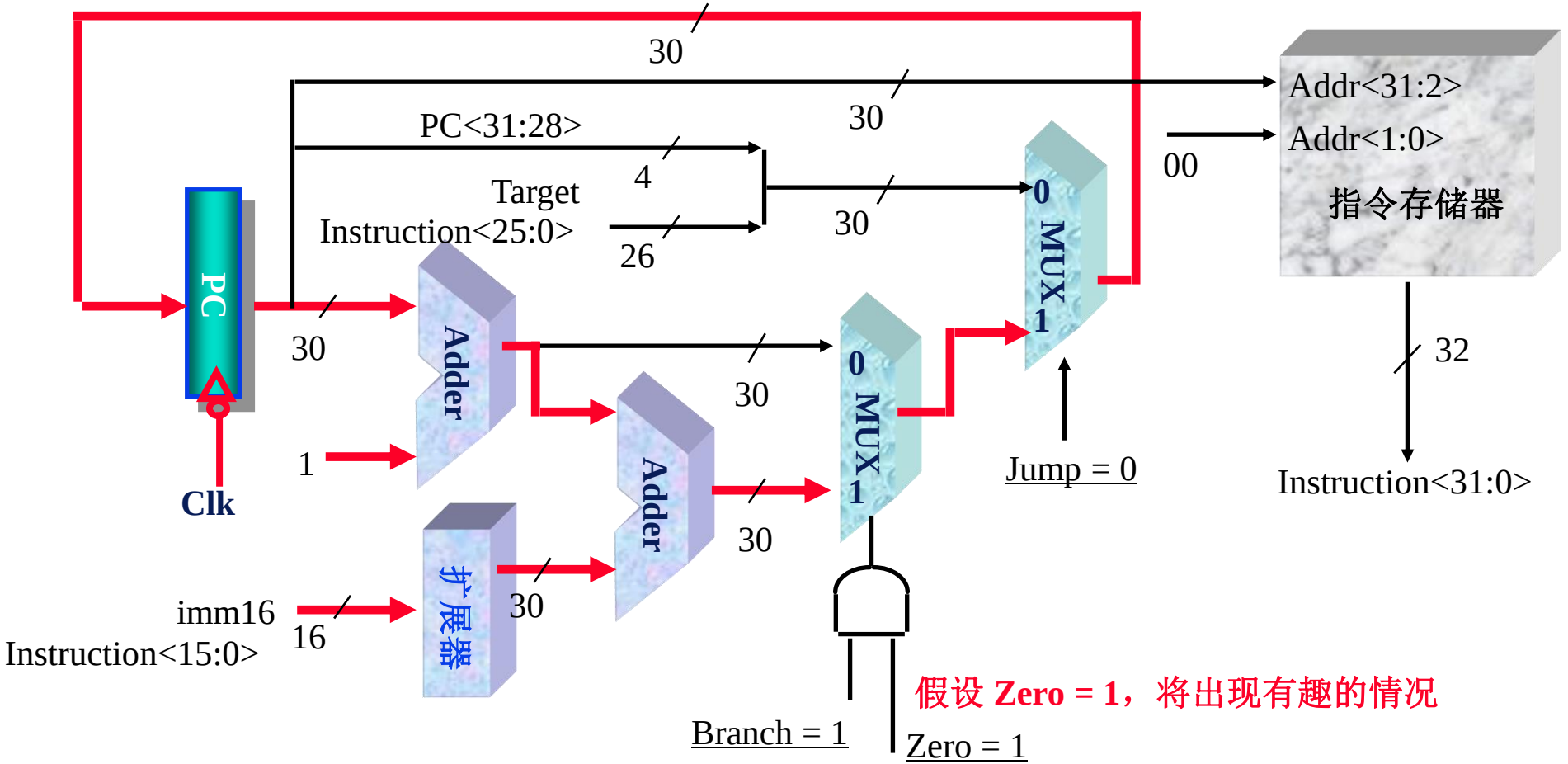
° if (R[rs] - R[rt] == 0) then Zero ⇐ 1 ; else Zero ⇐ 0



在转移结束时的取指部件

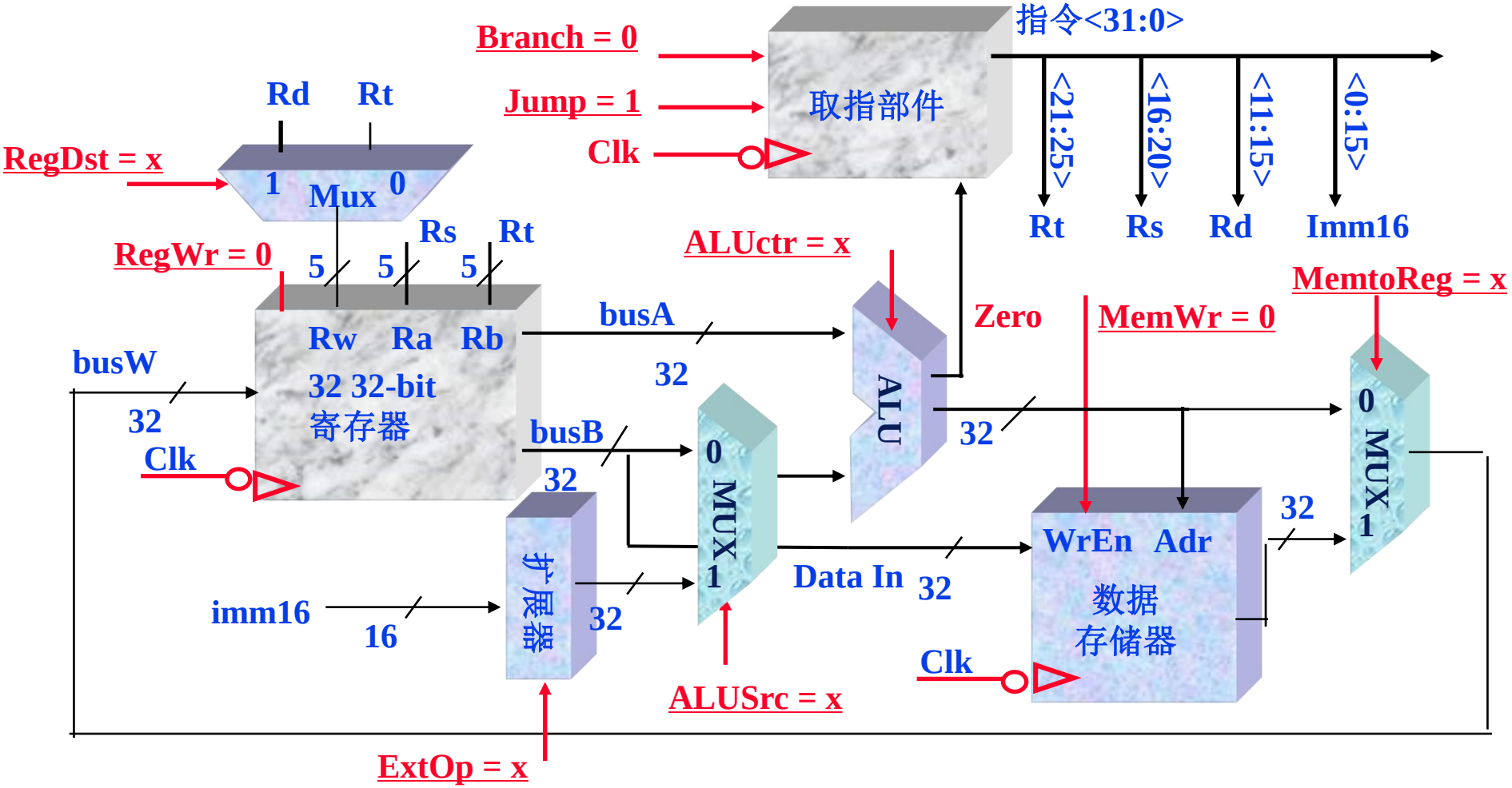


° if (Zero == 1) then PC = PC + 4 + SignExt[imm16]*4 ; else PC = PC + 4



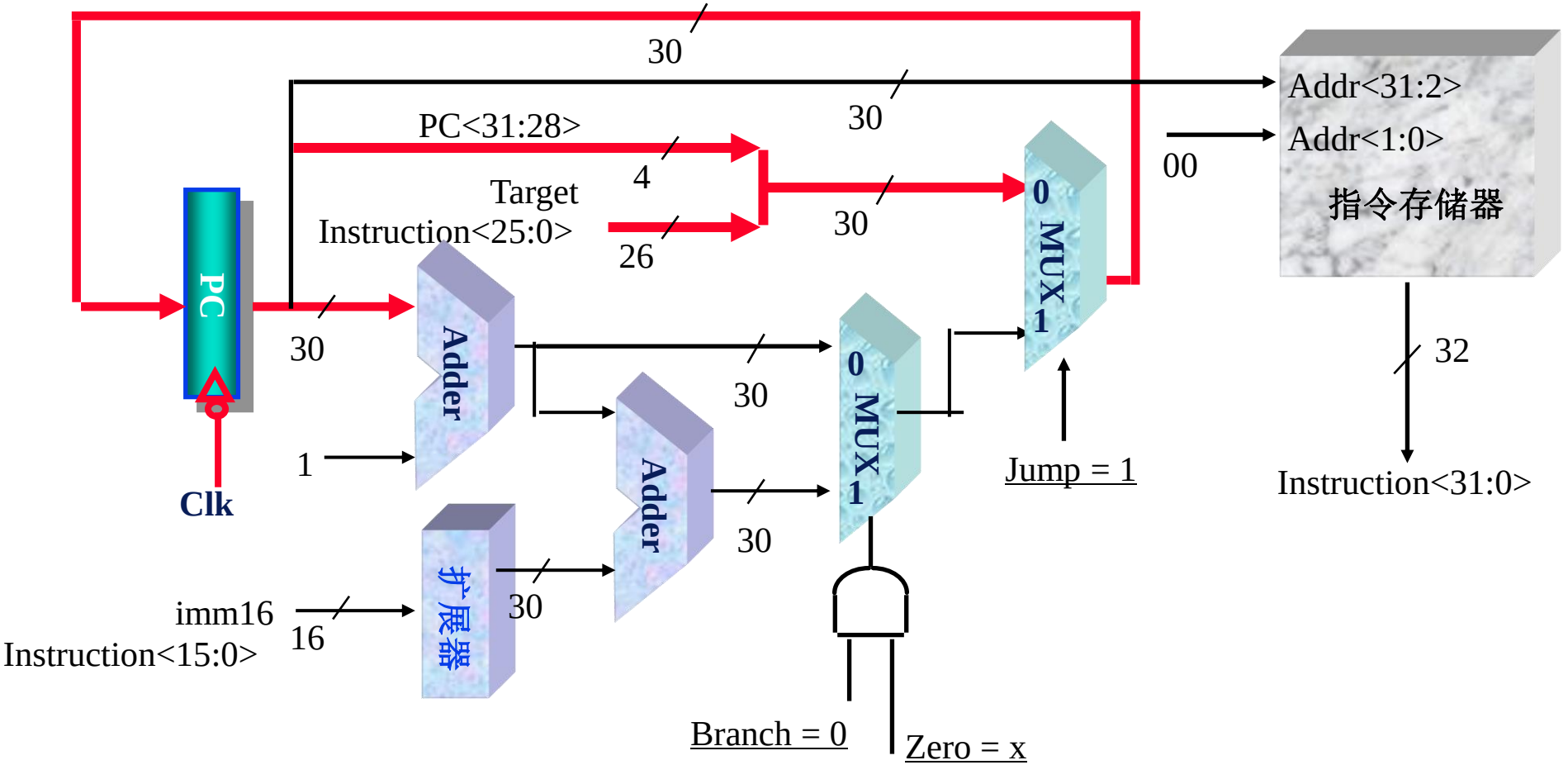
跳转过程中的单周期数据通路

- 什么也不做！ 确认控制信号设置正确！

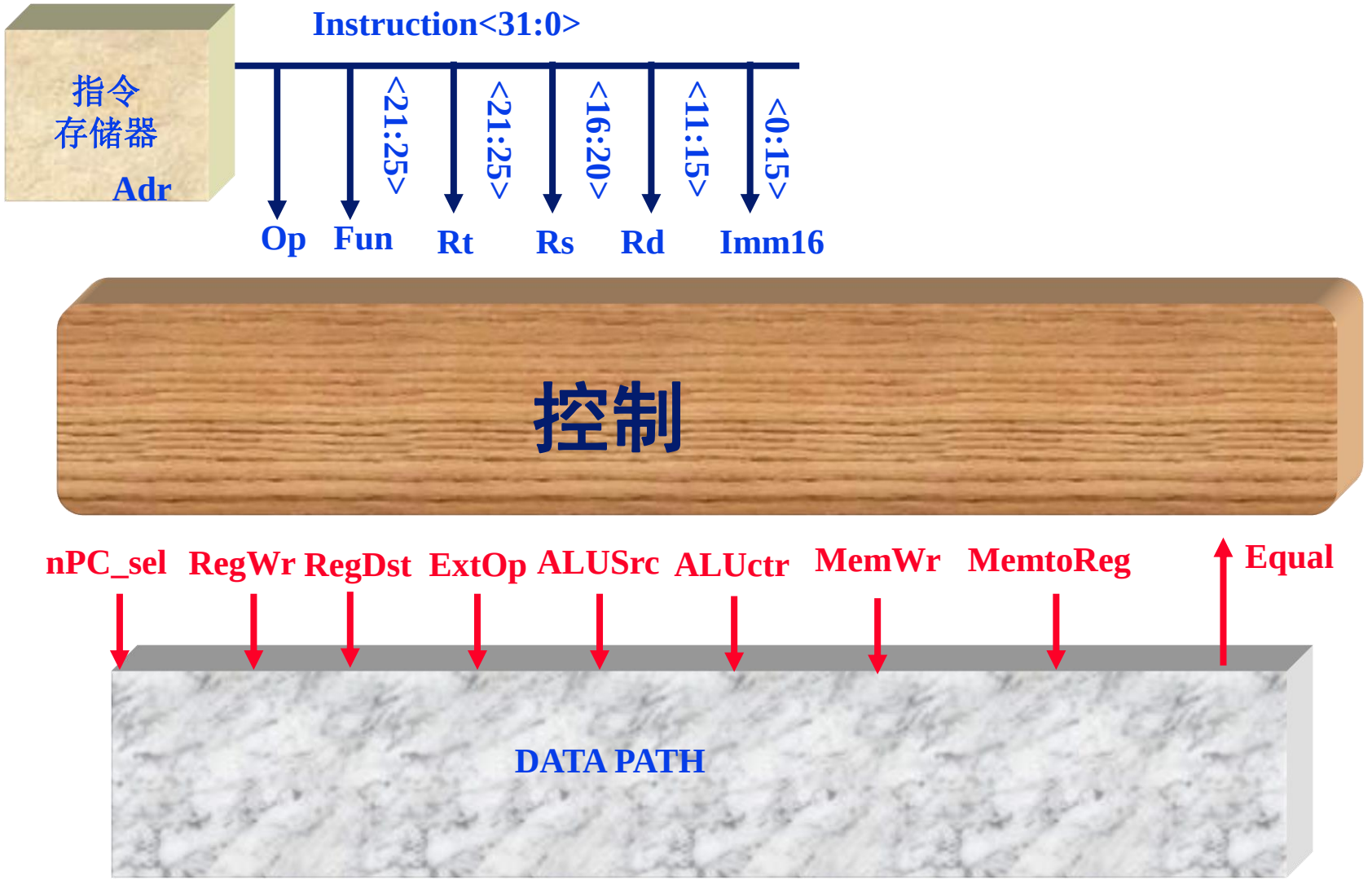


在跳转结束时的取指部件

- PC ⇐ PC<31:29> || target<25:0> || 0

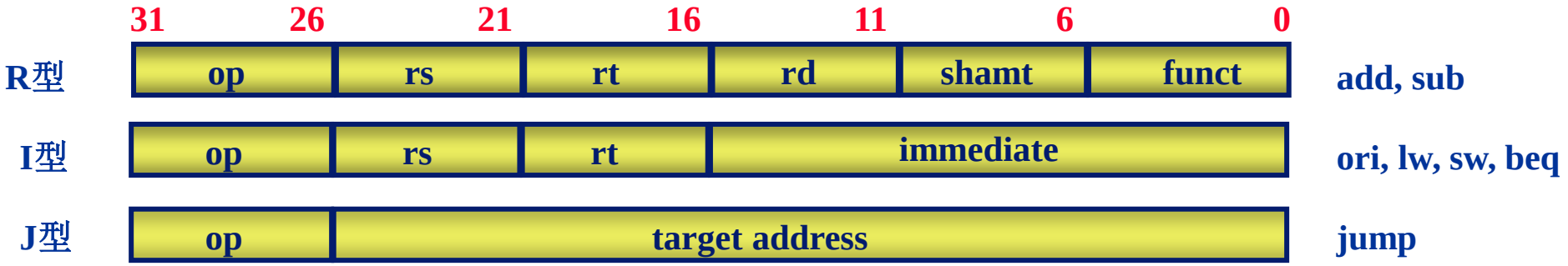


第四步　给定数据通路：RTL ⇒ 控制



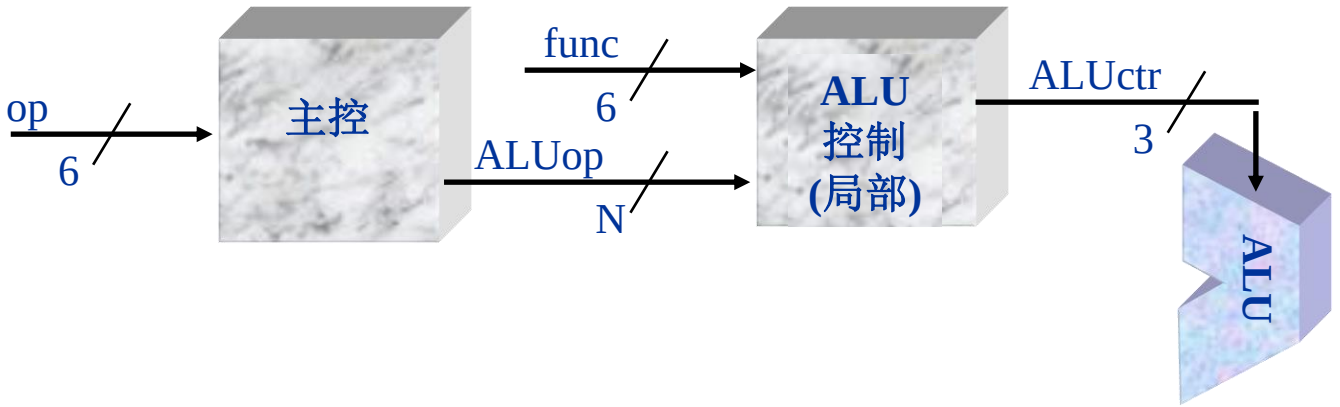
控制信号小结

func	10 0000	10 0010	我们不关心 :-)				
op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	add	sub	ori	lw	sw	beq	jump
RegDst	1	1	0	0	x	x	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	x	x
ALUctr<2:0>	Add	Subtract	Or	Add	Add	Subtract	xxx

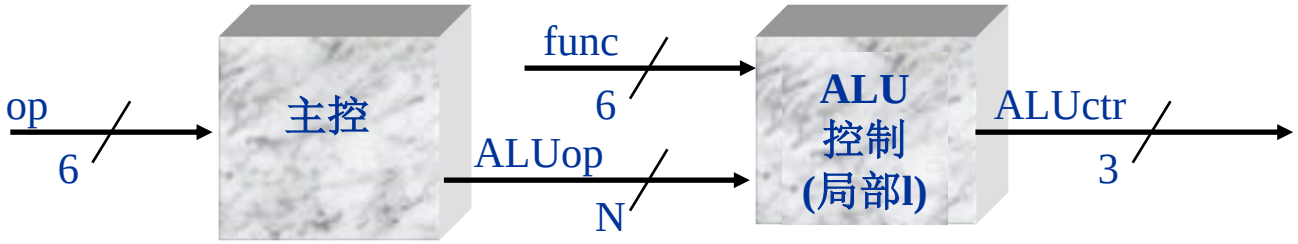


局部译码（Local Decoding）的概念

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<N:0>	R-type	Or	Add	Add	Subtract	xxx



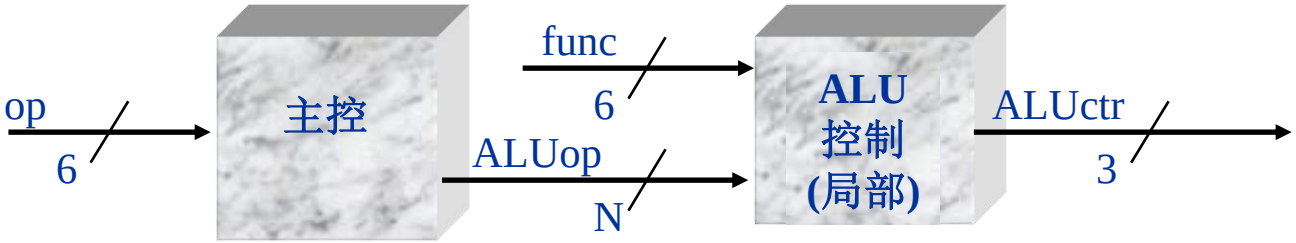
ALUop的编码



- ALUop需要 2位宽度来表示:
  - (1) R型 指令
  - 要求ALU执行下列操作的 I型指令:
    - (2) 或、 (3) 加 和 (4) 减
- 为了完整实现 MIPS ISA, ALUop需要 3位来表示:
  - (1) R型指令
  - 要求ALU执行下列操作的 I型指令:
    - (2) 或、 (3) 加、 (4) 减 和 (5) 与 (例如: andi)

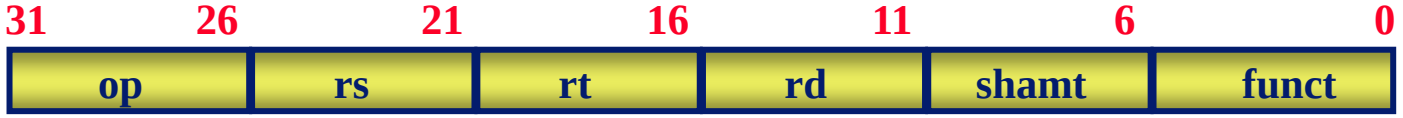
	R-type	ori	lw	sw	beq	jump
ALUop (Symbolic)	R-type	Or	Add	Add	Subtract	xxx
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01	xxx

Funct场位的译码

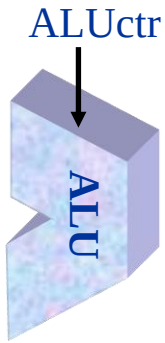


	R-type	ori	lw	sw	beq	jump
ALUop (Symbolic)	R-type	Or	Add	Add	Subtract	xxx
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01	xxx

R-type



funct<5:0>	Instruction Operation
10 0000	add
10 0010	subtract
10 0100	and
10 0101	or
10 1010	set-on-less-than



ALUctr<2:0>	ALU Operation
000	And
110	Subtract
010	Add
001	Or
111	Set-on-less-than

ALUctr的真值表

ALUop (Symbolic)	R-type	ori	lw	sw	beq
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01

funct<3:0>	Instruction Op.
0000	add
0010	subtract
0100	and
0101	or
1010	set-on-less-than

ALUop			func				ALU Operation	ALUctr		
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>		bit<2>	bit<1>	bit<0>
0	0	0	x	x	x	x	Add	0	1	0
0	x	1	x	x	x	x	Subtract	1	1	0
0	1	x	x	x	x	x	Or	0	0	1
1	x	x	0	0	0	0	Add	0	1	0
1	x	x	0	0	1	0	Subtract	1	1	0
1	x	x	0	1	0	0	And	0	0	0
1	x	x	0	1	0	1	Or	0	0	1
1	x	x	1	0	1	0	Set on <	1	1	1

ALUctr<2>的逻辑方程式

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

这使得可以不关心 func<3>

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

ALUctr<1>的逻辑方程式

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

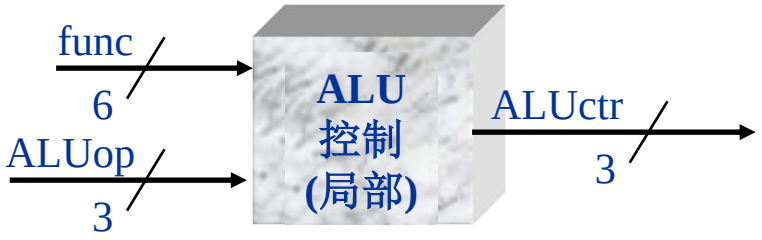
- ALUctr<1> = !ALUop<2> & !ALUop<1> +!ALUop<2> & ALUop<0>+
  ALUop<2> & !func<2> & !func<0>

ALUctr<0>的逻辑方程式

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

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

ALU的控制框图 (Control Block)



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

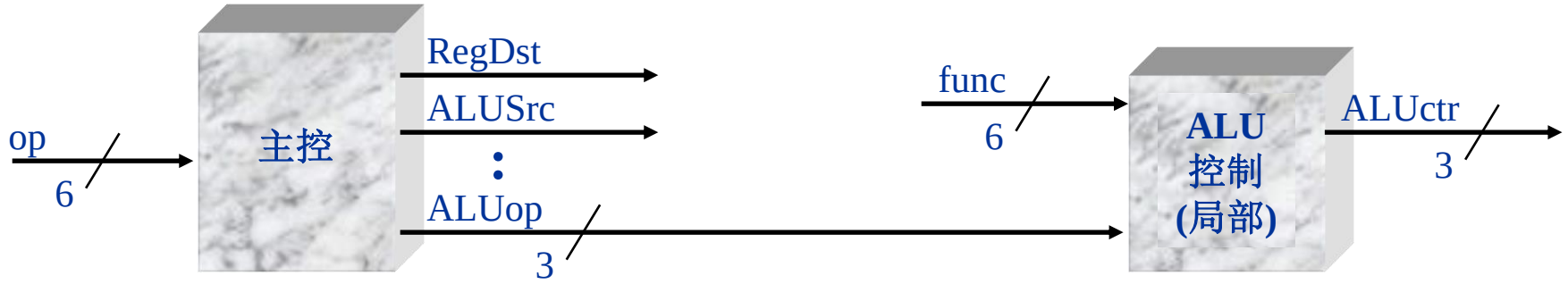
第五步：每个控制信号的逻辑

- Branch <= if (OP == BEQ) then 1 else 0
- Jump <= if (OP == JUMP) then 1 else 0
- ALUsrc <= if (OP == R-type) then RegB else Immed
- ALUctr <= if (OP == R-type) then funct
  elseif (OP == ORi) then OR
  elseif (OP == BEQ) then Sub
  else Add
- ExtOp <= _____
- MemWr <= _____
- MemtoReg <= _____
- RegWr: <=_____
- RegDst: <= _____

第五步：每个控制信号的逻辑

- Branch <= if (OP == BEQ) then 1 else 0
- Jump <= if (OP == JUMP) then 1 else 0
- ALUsrc <= if (OP == R-type) then RegB else Immed
- ALUctr <= if (OP == R-type) then funct
  elseif (OP == ORi) then OR
  elseif (OP == BEQ) then Sub?
  else Add
- ExtOp <= if (OP == ORi) then Zero else Sign
- MemWr <= (OP == Store)
- MemtoReg <= (OP == Load)
- RegWr: <= if ((OP == Store) || (OP == BEQ)) then 0 else 1
- RegDst: <= if ((OP == Load) || (OP == ORi)) then 0 else 1

主控制器（主控）的真值表

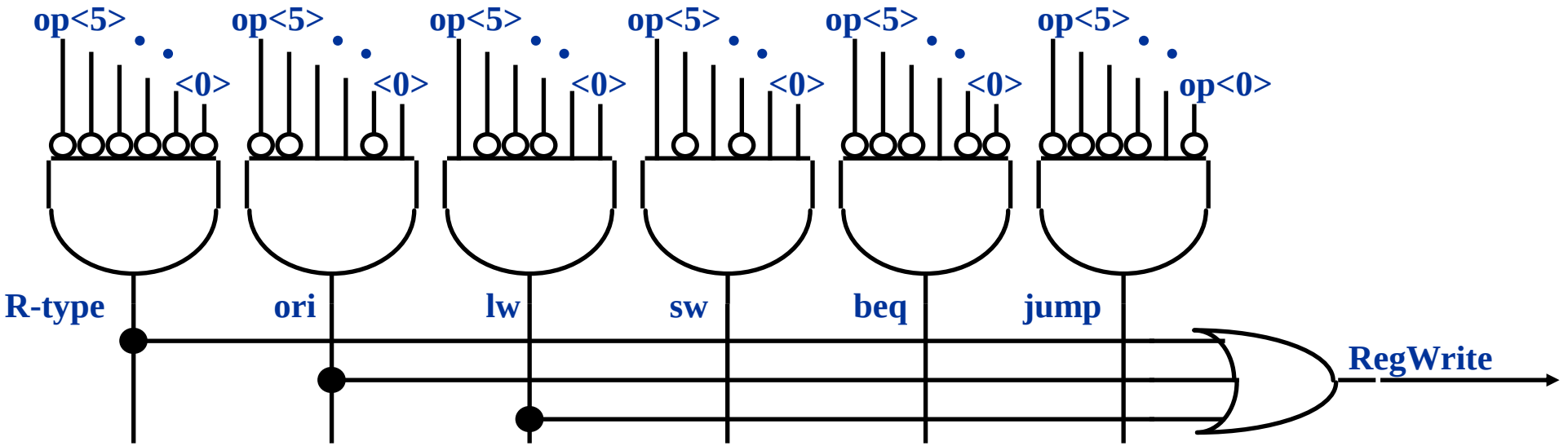


op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	**1**	**1**	**1**	**0**	**0**	**0**
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop (Symbolic)	R-type	Or	Add	Add	Subtract	xxx
ALUop <2>	1	0	0	0	0	x
ALUop <1>	0	1	0	0	0	x
ALUop <0>	0	0	0	0	1	x

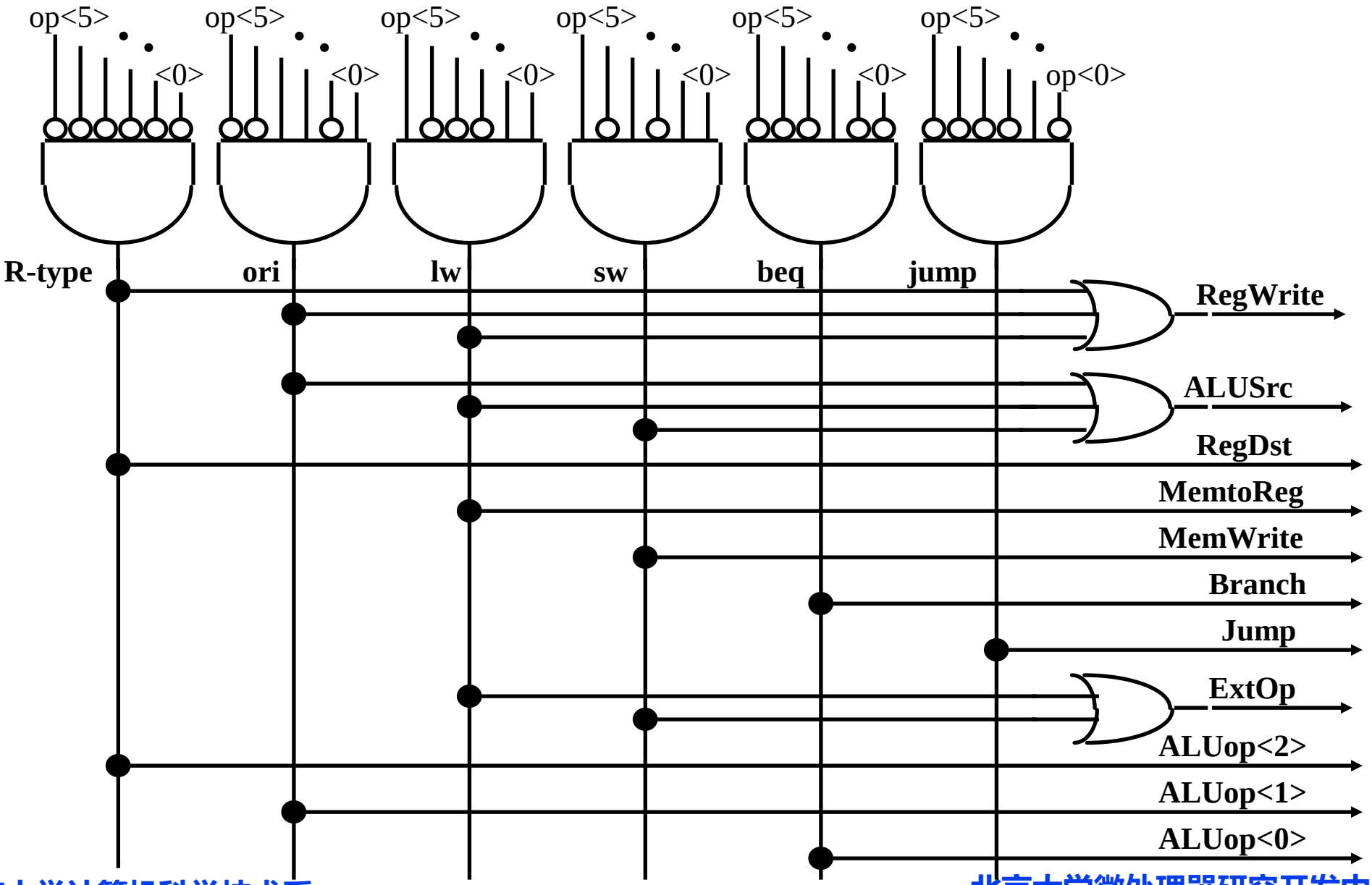
RegWrite的真值表

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

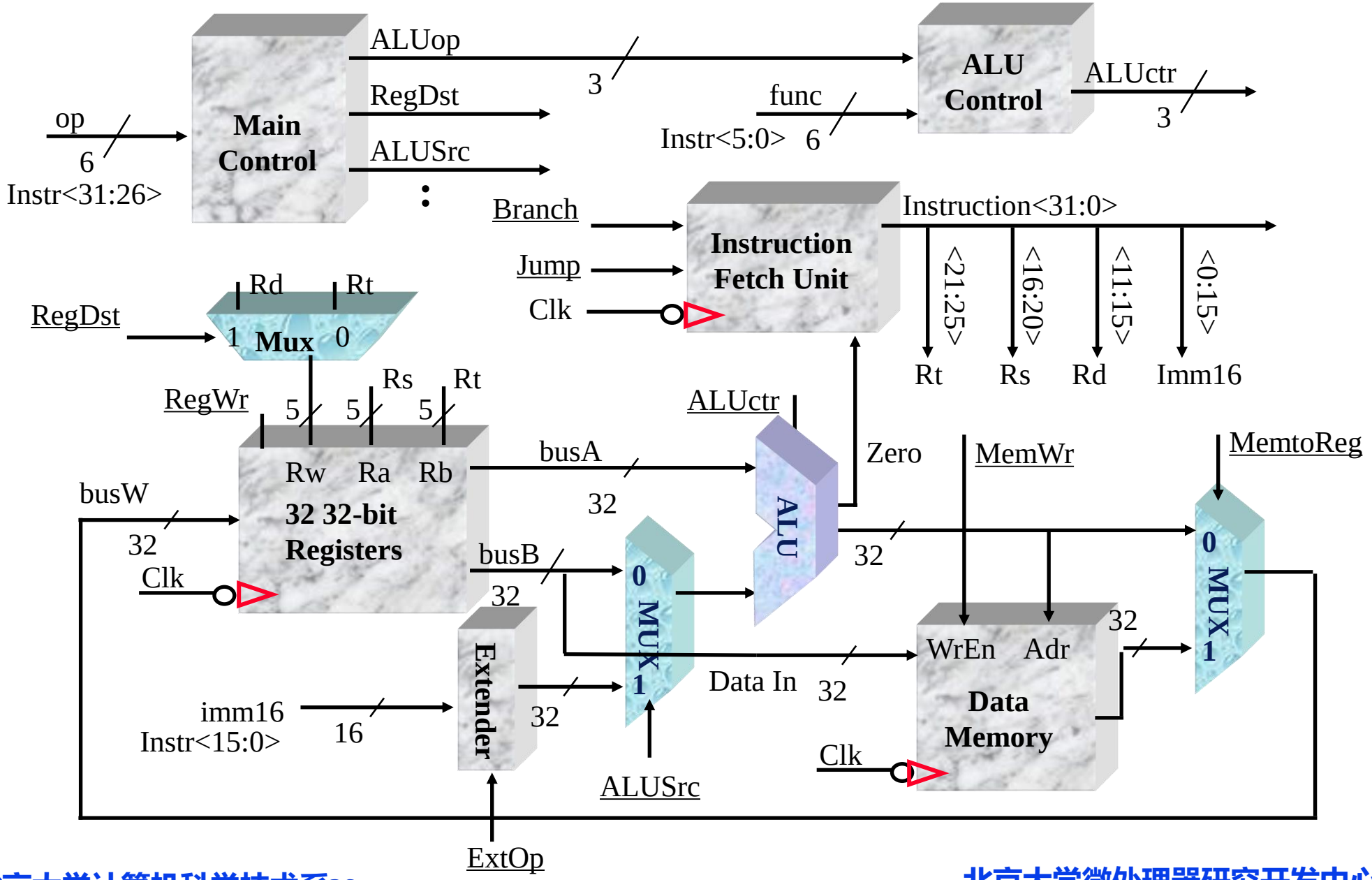
- RegWrite = R-type + ori + lw
  = !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0> (R-type)
  + !op<5> & !op<4> & op<3> & op<2> & !op<1> & op<0> (ori)
  + op<5> & !op<4> & !op<3> & !op<2> & op<1> & op<0> (lw)



主控的PLA实现


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

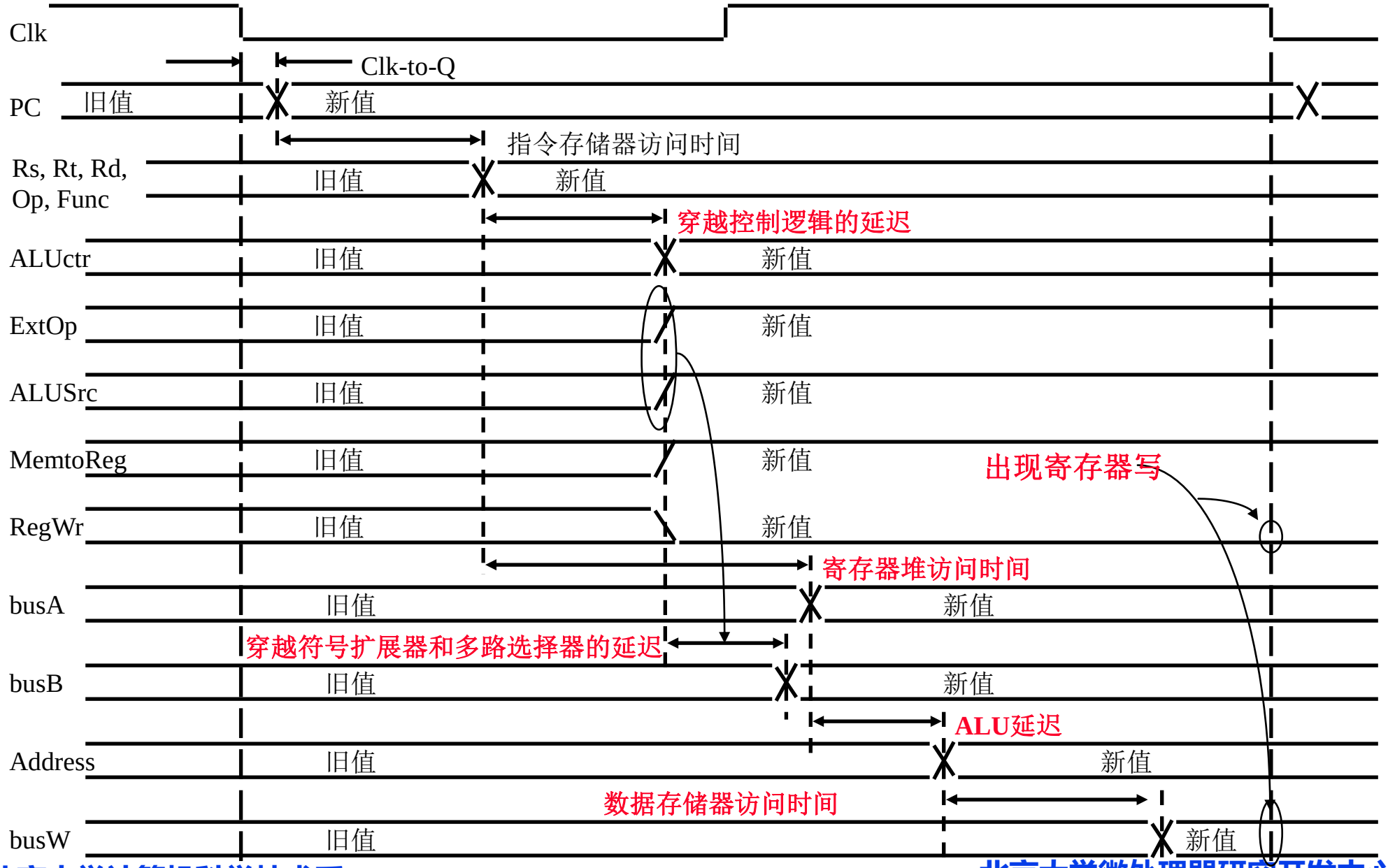
单周期处理器


Main Control
ALUop
RegDst
ALUSrc
op
6
Instr<31:26>
3
ALU Control
func
Instr<5:0>
6
ALUctr
3
Branch
Jump
Clk
Instruction Fetch Unit
Instruction<31:0>
<21:25>
<16:20>
<11:15>
<0:15>
Rt
Rs
Rd
Imm16
RegDst
Rd
Rt
1
Mux
0
RegWr
5
5
5
Rs
Rt
Rw
Ra
Rb
32 32-bit Registers
busW
32
Clk
busA
32
ALUctr
ALU
Zero
MemWr
MemtoReg
busB
32
0
MUX
1
32
Extender
imm16
Instr<15:0>
16
32
ALUSrc
Data In
32
WrEn
Adr
Data Memory
32
Clk
ExtOp

与实际MIPS处理器的区别

- 实际MIPS处理器 采用 延迟装入策略:
  - lw $1, 100 ($2) // 装载寄存器 R1
  - add $3, $1, $0 // 将R1的老数据传输到R3中
  - add $4, $1, $0 // 将R1的新数据传输到R4中
  - 在上述单周期处理器中，装入的影响 没有 被延迟
    - lw $1, 100 ($2) // 装载寄存器 R1
    - add $3, $1, $0 //将R1的新数据传输到R3中
- 实际MIPS处理器对转移和跳转 采用 延迟转移策略:
  - 指令地址: 0x00 j 1000
  - 指令地址: 0x04 add $1, $2, $3
  - 指令地址: 0x1000 sub $1, $2, $3
  - 在上述单周期处理器中，对转移和跳转 没有采用 延迟转移策略：
    - 指令地址: 0x00 j 1000
    - 指令地址: 0x1000 sub $1, $2, $3

最坏情况的定时（Load）



该单周期处理器的缺陷

- 时钟周期时间长:
  - 对于装入指令，周期时间必须足够长:

    PC的Clock -to-Q +

    指令存储器访问时间 +

    寄存器堆访问时间 +

    ALU延迟（地址计算） +

    数据存储器访问时间 +

    寄存器对建立时间 +

    时钟纽斜

- 对于所有其他指令，周期时间都比所需的要长很多!