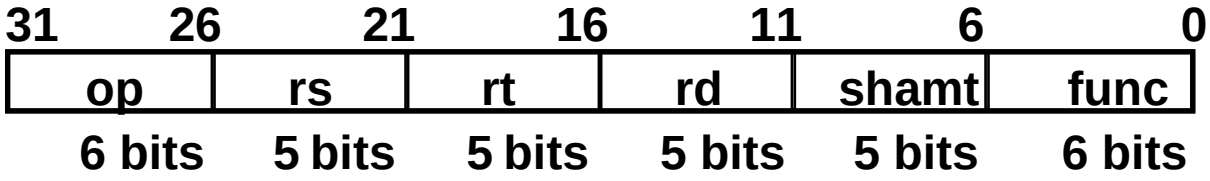
Lecture 22: CPU II

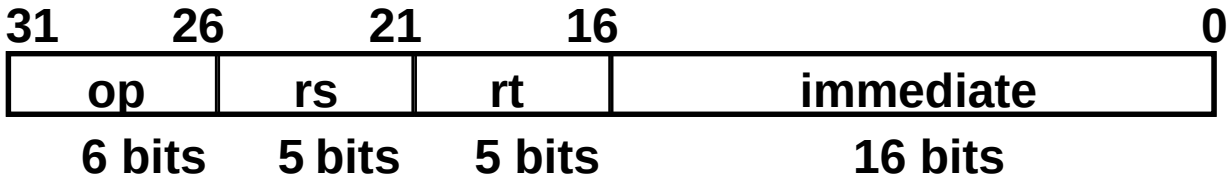
(Single Cycle)

复习：MIPS的三种指令类型

大家记得是哪三种类型？ R-Type、I-Type、J-Type

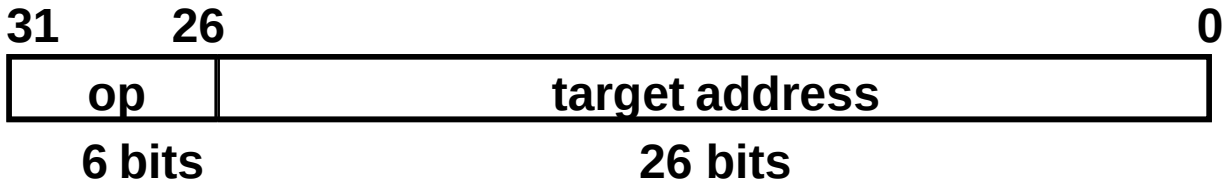
- ADD and SUBSTRACT
  - add rd, rs, rt
  - sub rd, rs, rt
- OR Immediate:
  - ori rt, rs, imm16
- LOAD and STORE
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- BRANCH:
  - beq rs, rt, imm16
- JUMP:
  - j target





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

本讲目标：实现以上7条指令对应的数据通路！
教材中实现了11条指令，可将7条指令和11条指令的数据通路进行对比，以深入理解设计原理。



设计处理器的步骤

第一步：分析每条指令的功能，并用RTL(Register Transfer Language)来表示。

第二步：根据指令的功能给出所需的元件，并考虑如何将他们互连。

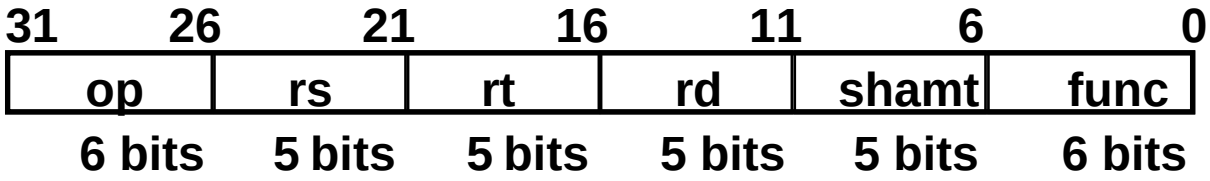
第三步：确定每个元件所需控制信号的取值。

第四步：汇总所有指令所涉及到的控制信号，生成一张反映指令与控制信号之间关系的表。

第五步：根据表得到每个控制信号的逻辑表达式，据此设计控制器电路。

- 处理器设计涉及到数据通路的设计和控制器的设计
- 数据通路中有两种元件
  - 操作元件：由组合逻辑电路实现
  - 存储（状态）元件：由时序逻辑电路实现

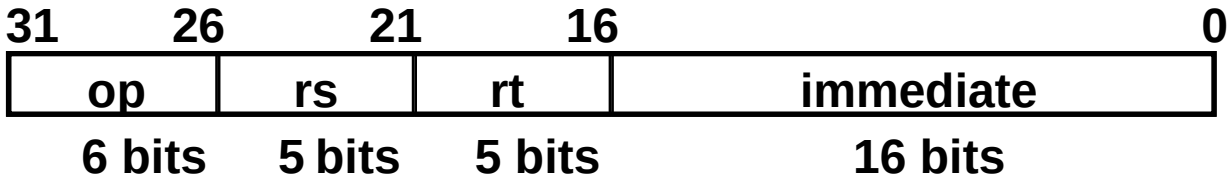
RTL: The ADD Instruction（加法指令）



- **add rd, rs, rt**

  - **M[PC]** 从**PC**所指的内存单元中取指令
  - **R[rd] ← R[rs] + R[rt]** 从**rs**、**rt** 所指的寄存器中取数后相加，结果送**rd** 所指的寄存器中
  - **PC ← PC + 4** **PC**加**4**，使**PC**指向下一条指令

RTL: The Load Instruction（装入指令）



- **lw rt, rs, imm16**
  - **M[PC]** **(同加法指令)**
  - **Addr ← R[rs] + SignExt(imm16)**

    计算数据地址 **(**立即数要进行符号扩展**)**
  - **R[rt] ← M[Addr]**

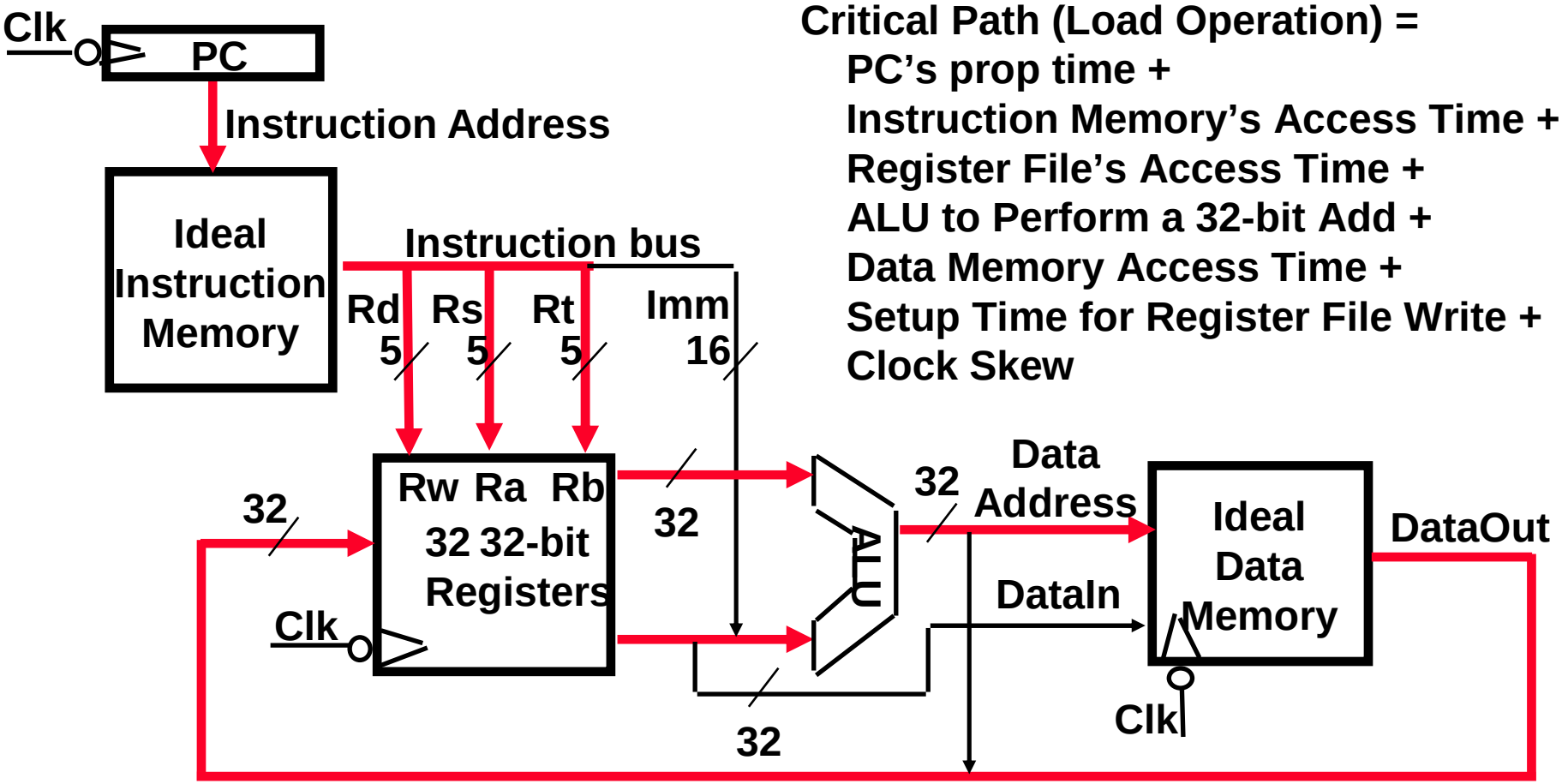
    从存储器中取出数据，装入到寄存器中
  - **PC ← PC + 4** **(同加法指令)**

与**R-type**加法指令相比，更复杂！

数据通路中的关键路径(Load操作)

Load操作：
R[Rt] ← M[R[Rs]+Imm16]

- 记住：寄存器组和理想存储器的定时方式
  - 写操作时，作为时序逻辑电路。即：
    - 时钟到达前，输入需**setup**；到达后经**“Clk to Q”**，写入数据到达输出端
  - 读操作时，作为组合逻辑电路。即：
    - 地址有效后经过 **“access time”**，输出开始有效

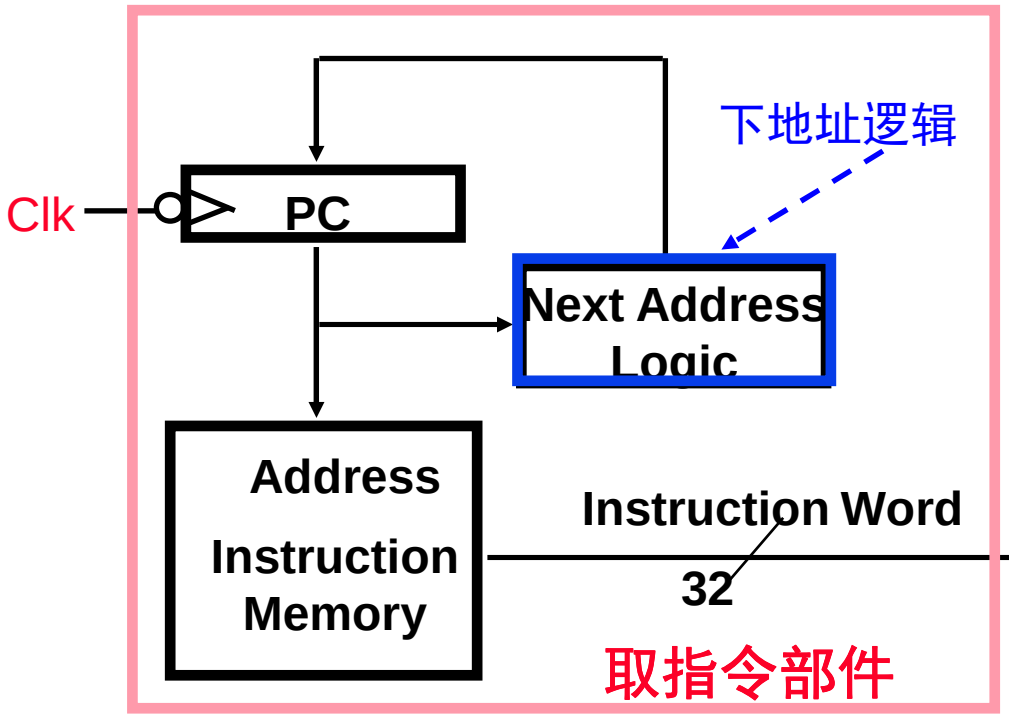


Critical Path (Load Operation) =
PC’s prop time +
Instruction Memory’s Access Time +
Register File’s Access Time +
ALU to Perform a 32-bit Add +
Data Memory Access Time +
Setup Time for Register File Write +
Clock Skew

取指令部件(Instruction Fetch Unit)

- 每条指令都有的公共操作：
  - **取指令： M[PC]**
  - **更新PC：PC ← PC + 4**

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



顺序：先取指令，再改PC的值
（具体实现时，可以并行）

绝不能先改PC的值，再取指令

取指后，各指令功能不同，数据通路中信息流动过程也不同

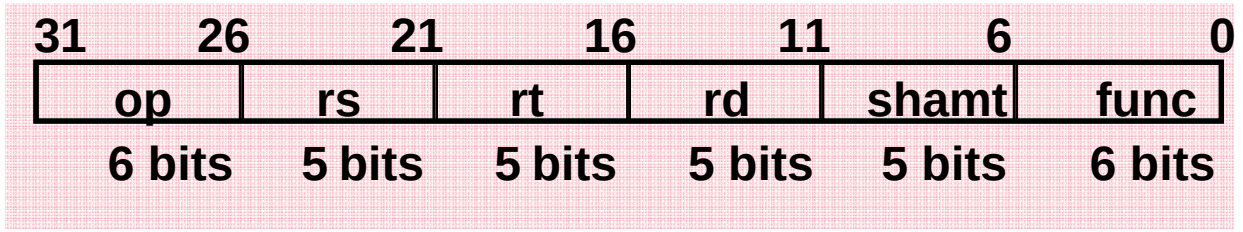
下面分别对每条指令进行相应数据通路的设计

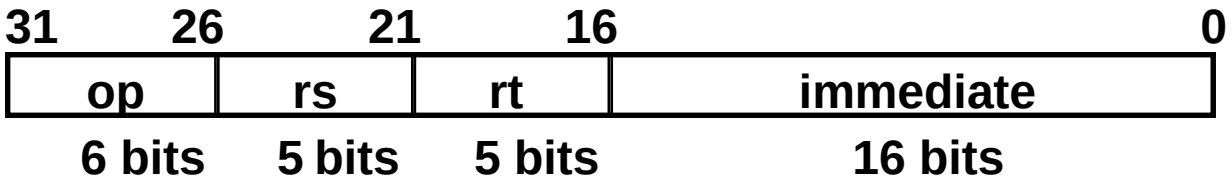
加法和减法指令(R-type类型）

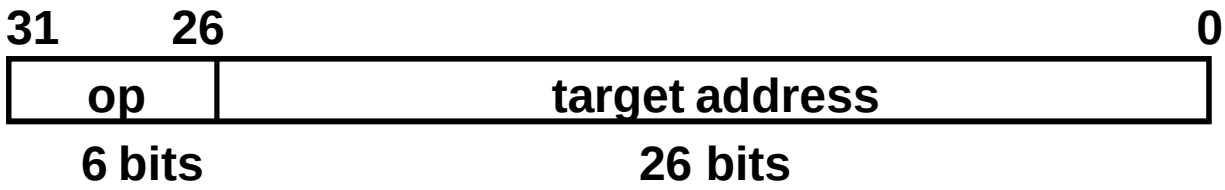
首先考虑add和sub指令（R-Type指令的代表）

实现目标（7条指令）：

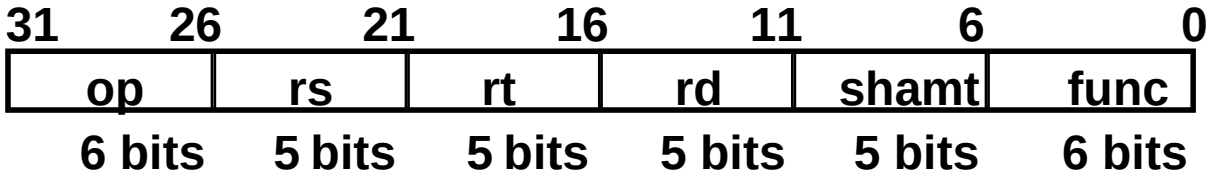
- ADD and subtract
  - add rd, rs, rt
  - sub rd, rs, rt
- OR Immediate:
  - ori rt, rs, imm16
- LOAD and STORE
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- BRANCH:
  - beq rs, rt, imm16
- JUMP:
  - j target





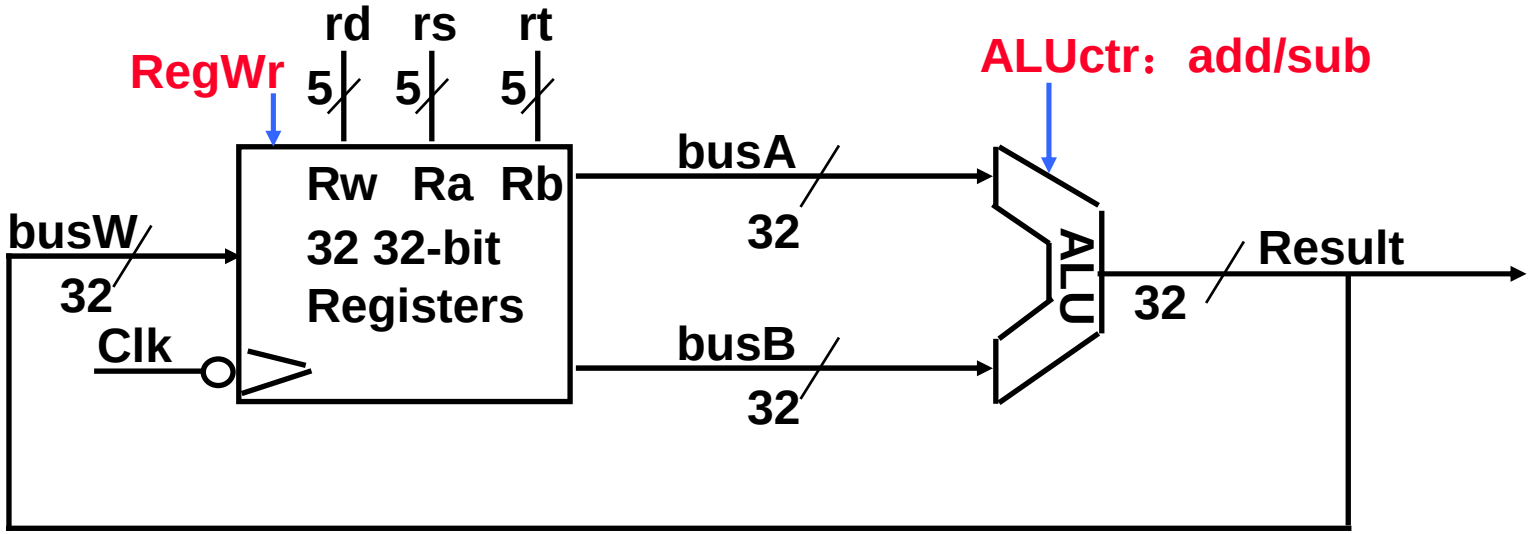


RR（R-type）型指令的数据通路



° 功能：R[rd] ← R[rs] op R[rt]　　Example: add　rd, rs, rt

不考虑公共操作，仅R-Type指令执行阶段的数据通路如下：



Ra, Rb, Rw 分别对应指令的rs, rt, rd

ALUctr,RegWr：指令译码后产生的控制信号

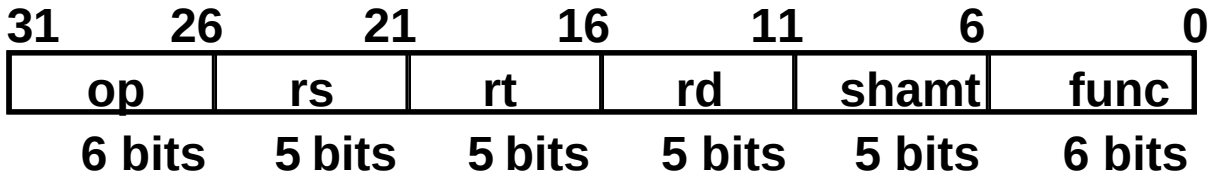
“add rd, rs, rt”控制信号为？

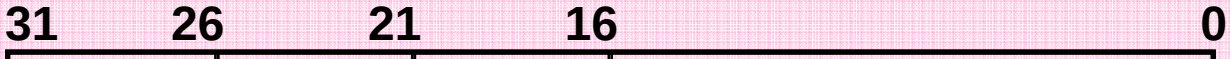
ALUctr=add，RegWr=1

带立即数的逻辑指令（ori指令）

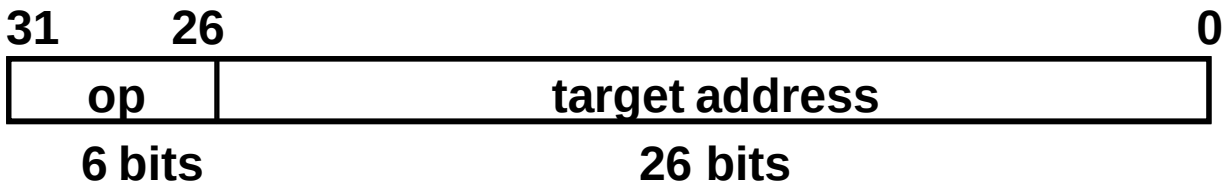
实现目标（7条指令）：

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

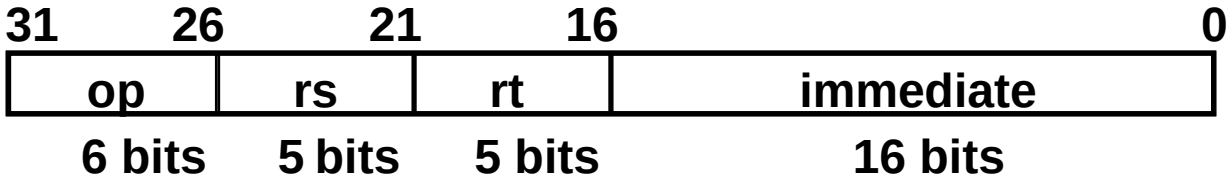




2. 考虑ori 指令（I-Type指令和逻辑运算指令的代表）



RTL: The OR Immediate Instruction

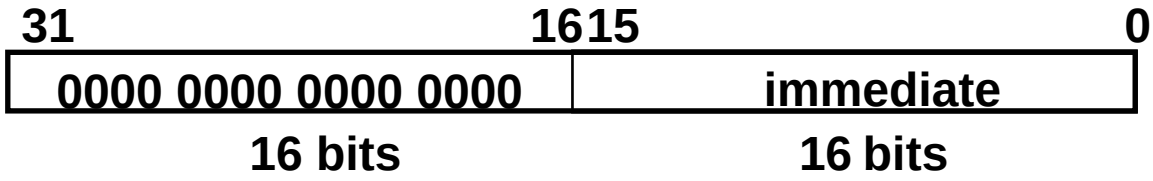


逻辑运算，立即数为逻辑数

- ori rt, rs, imm16
  - M[PC] 取指令（公共操作，取指部件完成）
  - R[rt] ← R[rs] or ZeroExt(imm16)

    立即数零扩展，并与rs内容做“或”运算
  - PC ← PC + 4 计算下地址（公共操作，取指部件完成）

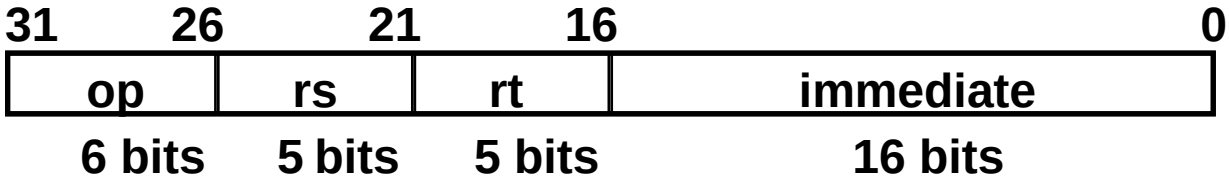
零扩展 ZeroExt(imm16)



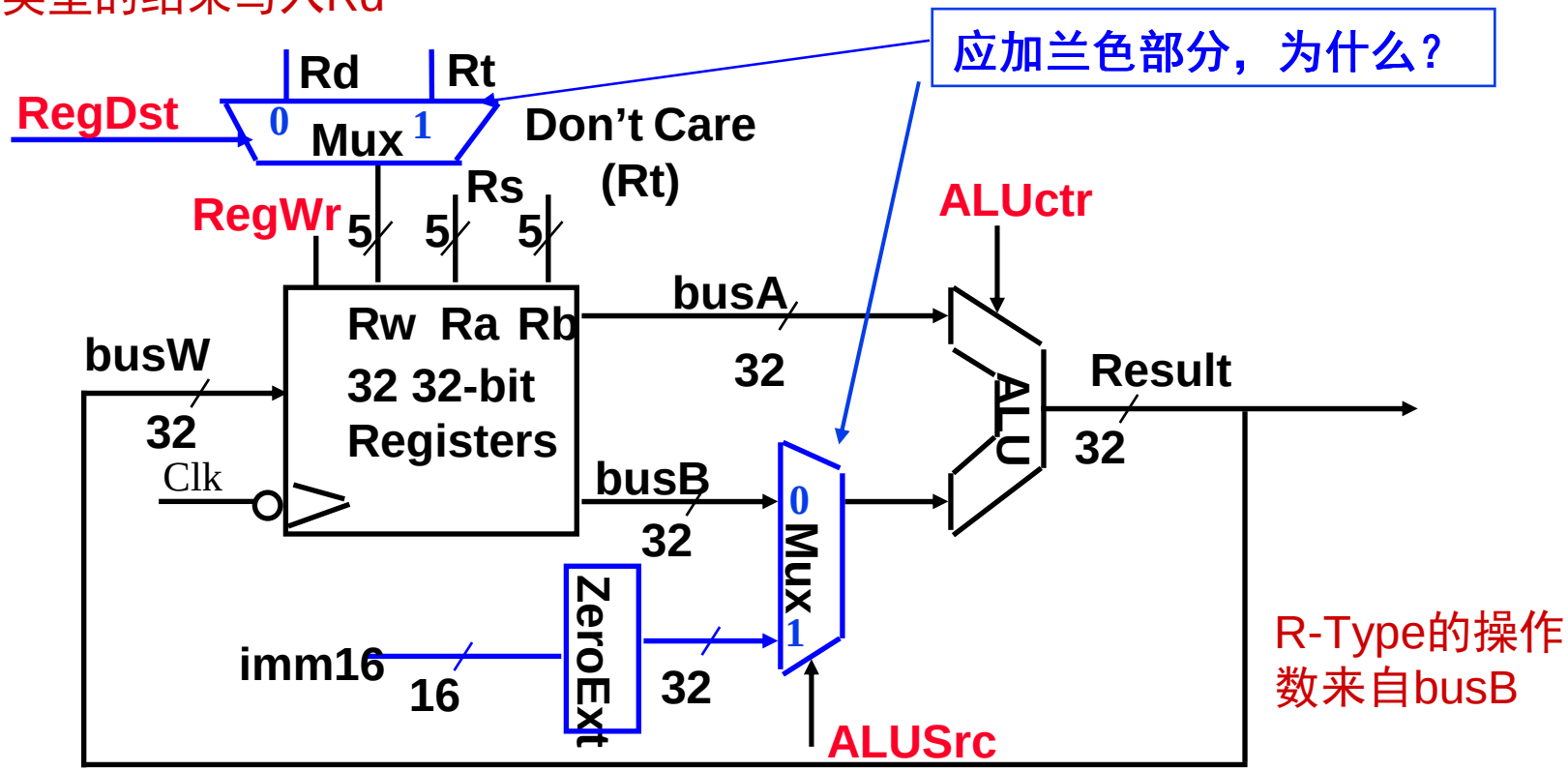
思考：应在前面数据通路上加哪些元件和连线？用何控制信号？

带立即数的逻辑指令的数据通路

- R[rt] ← R[rs] op ZeroExt[imm16]] Example: ori rt, rs, imm16



R-Type类型的结果写入Rd



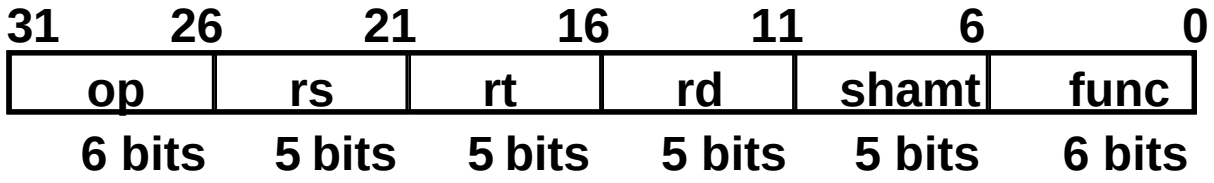
Ori指令的控制信号: RegDst=?, RegWr=?, ALUctr=?, ALUSrc=?

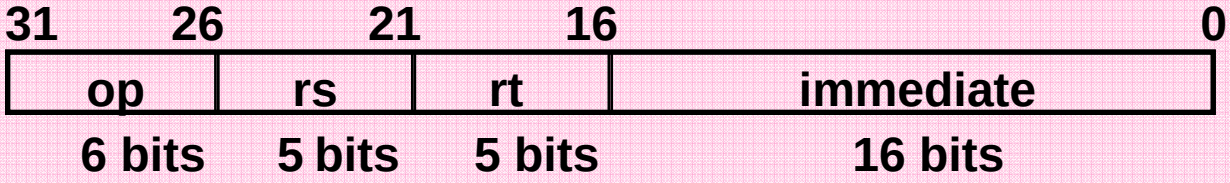
Ori指令的控制信号: RegDst=1, RegWr=1, ALUSrc=1, ALUctr=or

访存指令中的数据装入指令 (lw)

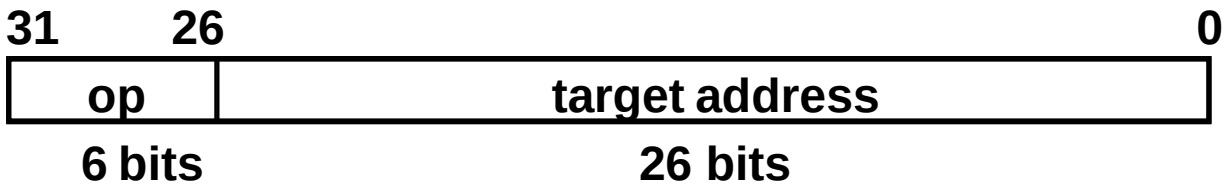
实现目标（7条指令）：

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



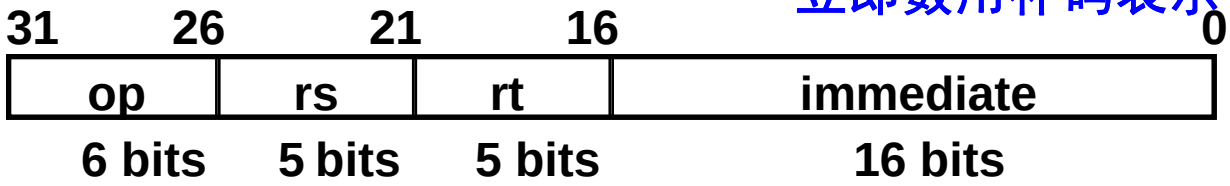


3. 考虑lw 指令（访存指令的代表）



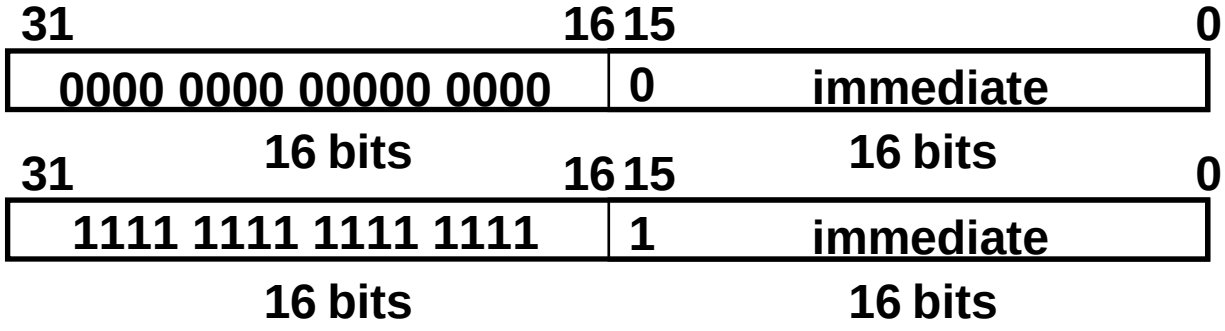
RTL: The Load Instruction

立即数用补码表示



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

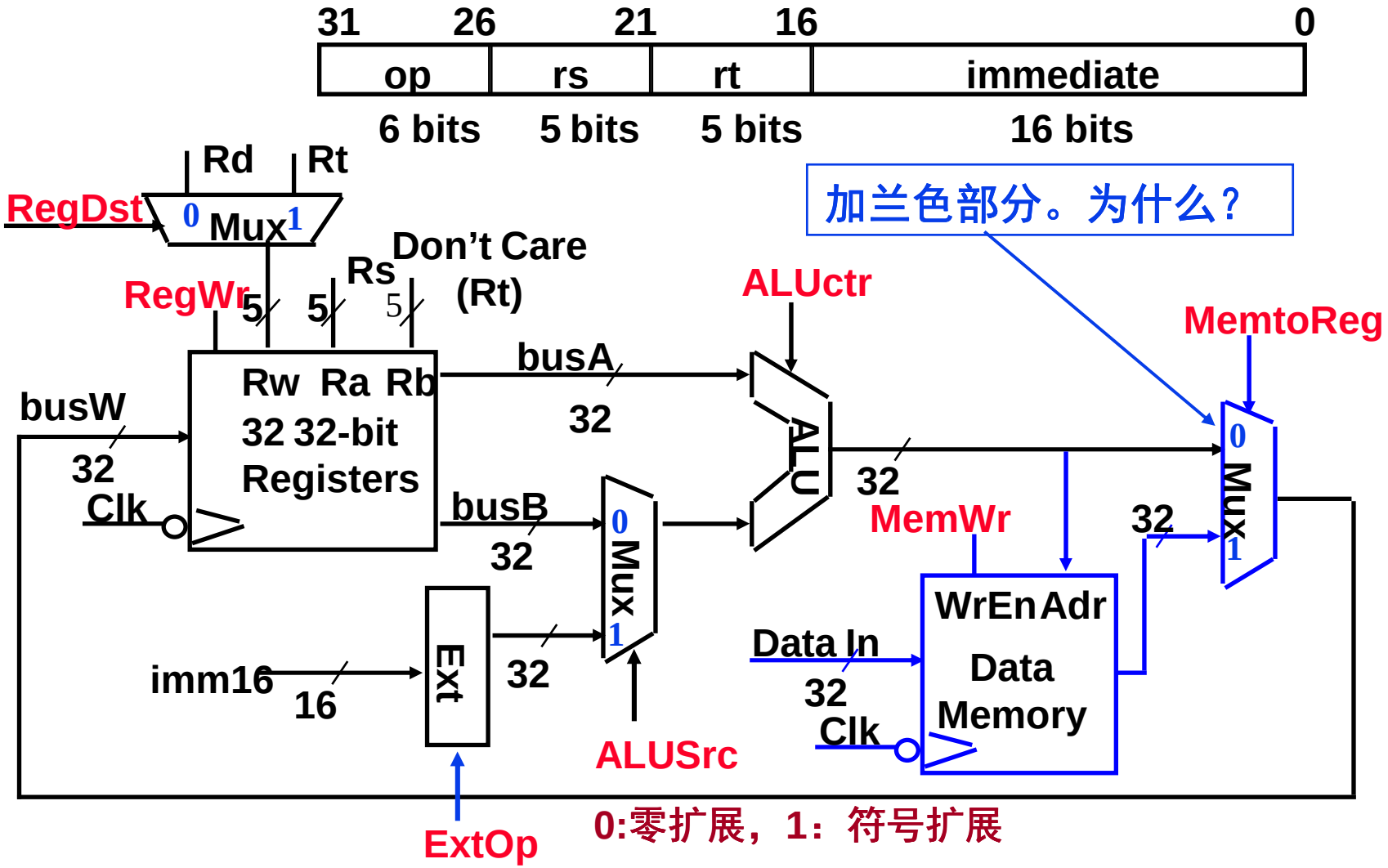
符号扩展(为什么不是零扩展?) :



思考：应在原数据通路上加哪些元件和连线？用何控制信号？

装入(lw)指令的数据通路

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

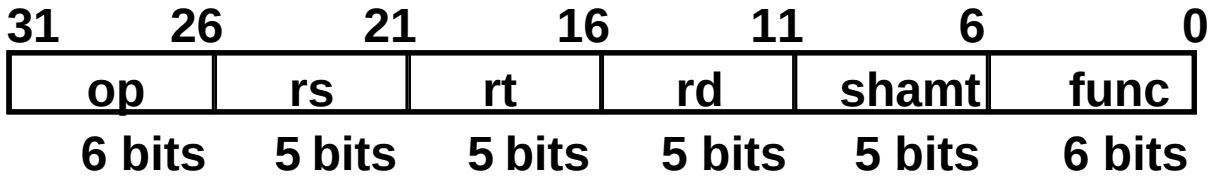


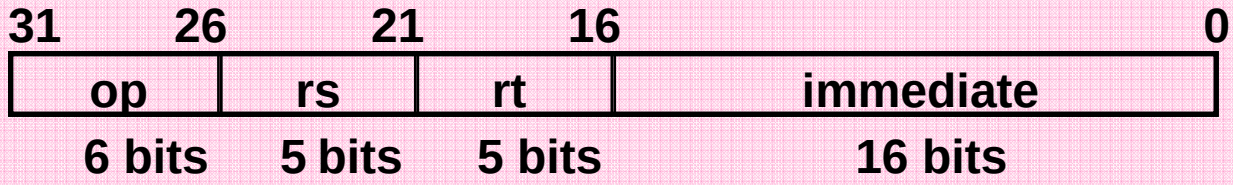
RegDst=1, RegWr=1, ALUctr=add, ExtOp=1, ALUSrc=1, MemWr=0, MemtoReg=1

访存指令中的存数指令 (sw)

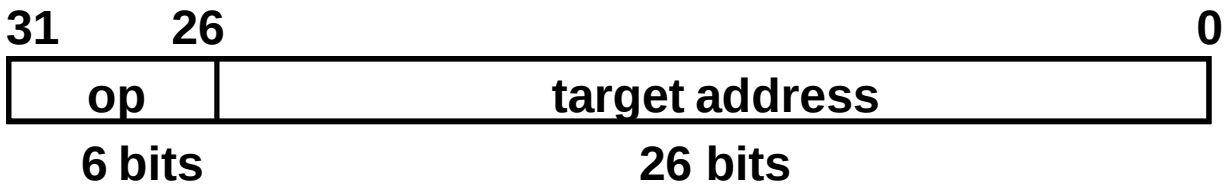
实现目标（7条指令）：

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

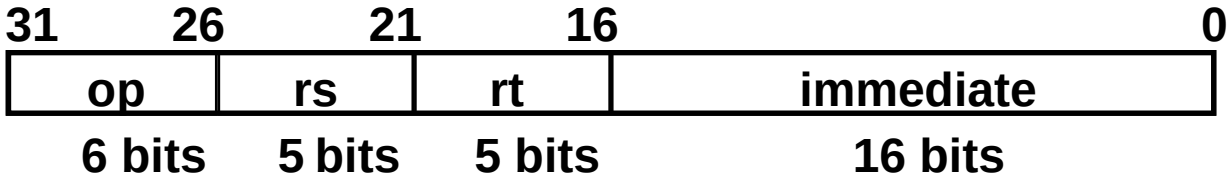




4. 考虑sw 指令（访存指令的代表）



RTL: The Store Instruction

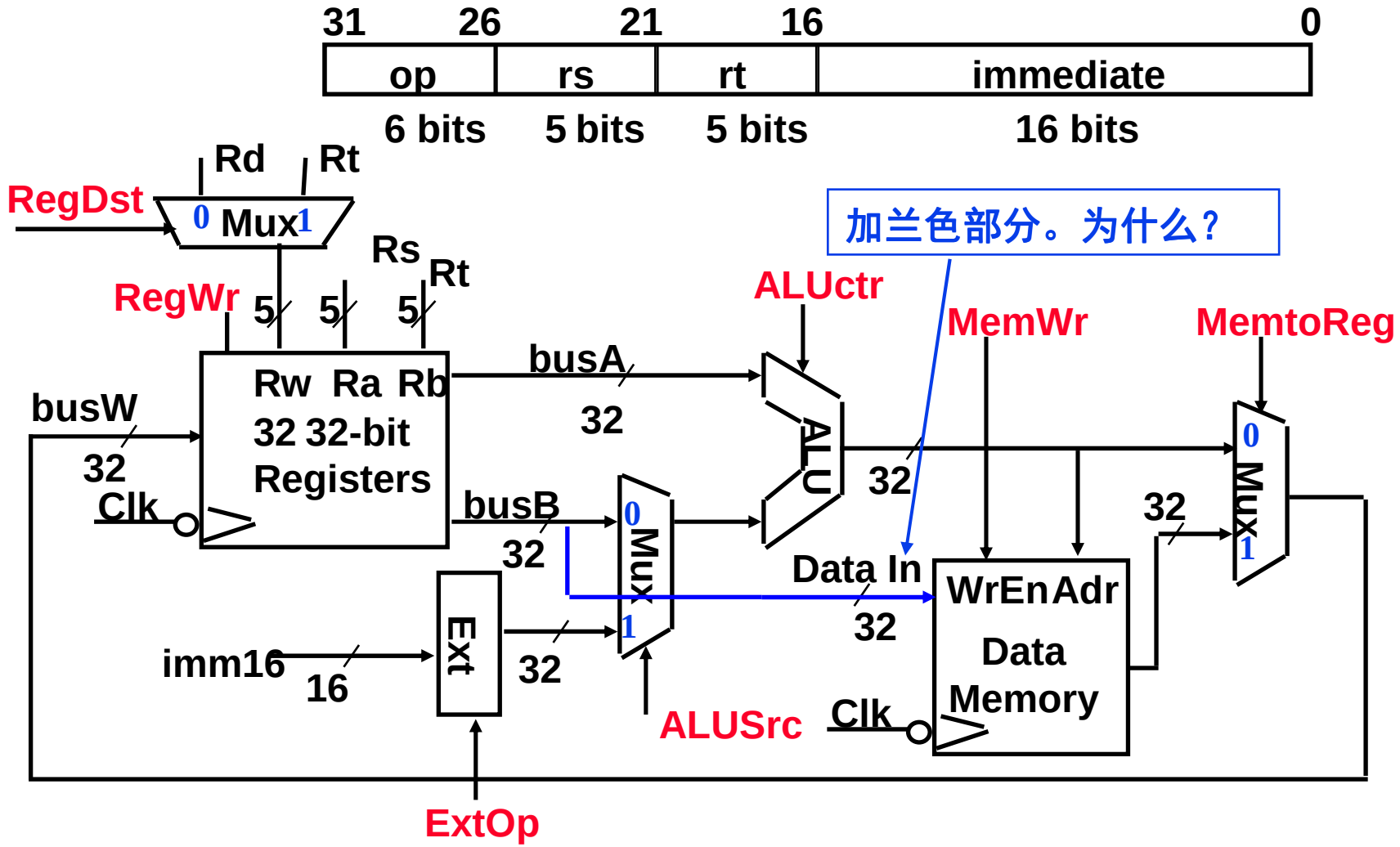


- sw rt, rs, imm16
  - M[PC] 取指令（公共操作，取指部件完成）
  - Addr ← R[rs] + SignExt(imm16) 计算存储单元地址（符号扩展！）
  - Mem[Addr] ← R[rt] 寄存器rt中的内容存到内存单元中
  - PC ← PC + 4 计算下地址（公共操作，取指部件完成）

思考：应在原数据通路上加哪些元件和连线？用何控制信号？

存数(sw)指令的数据通路

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

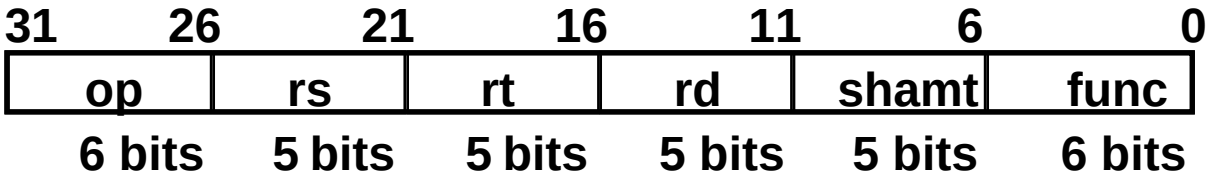


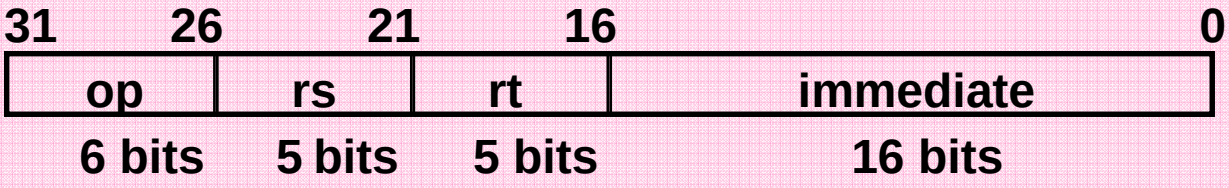
RegDst=x, RegWr=0, ALUctr=add, ExtOp=1, ALUSrc=1, MemWr=1, MemtoReg=x

分支（条件转移）指令（相等转移：beq）

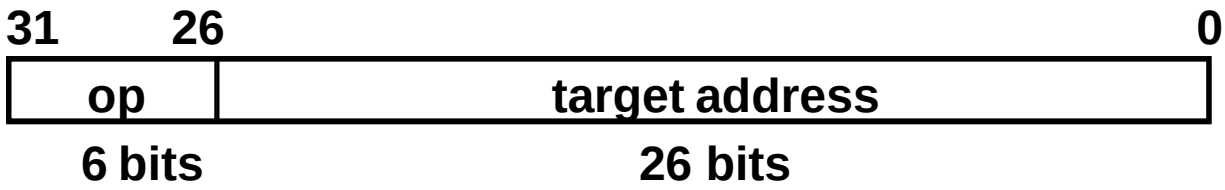
实现目标（7条指令）：

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



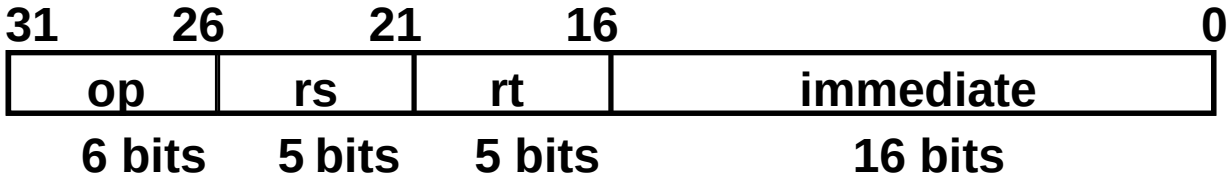


5. 考虑beq指令（条件转移指令的代表）



RTL: The Branch Instruction

立即数用补码表示



- beq rs, rt, imm16

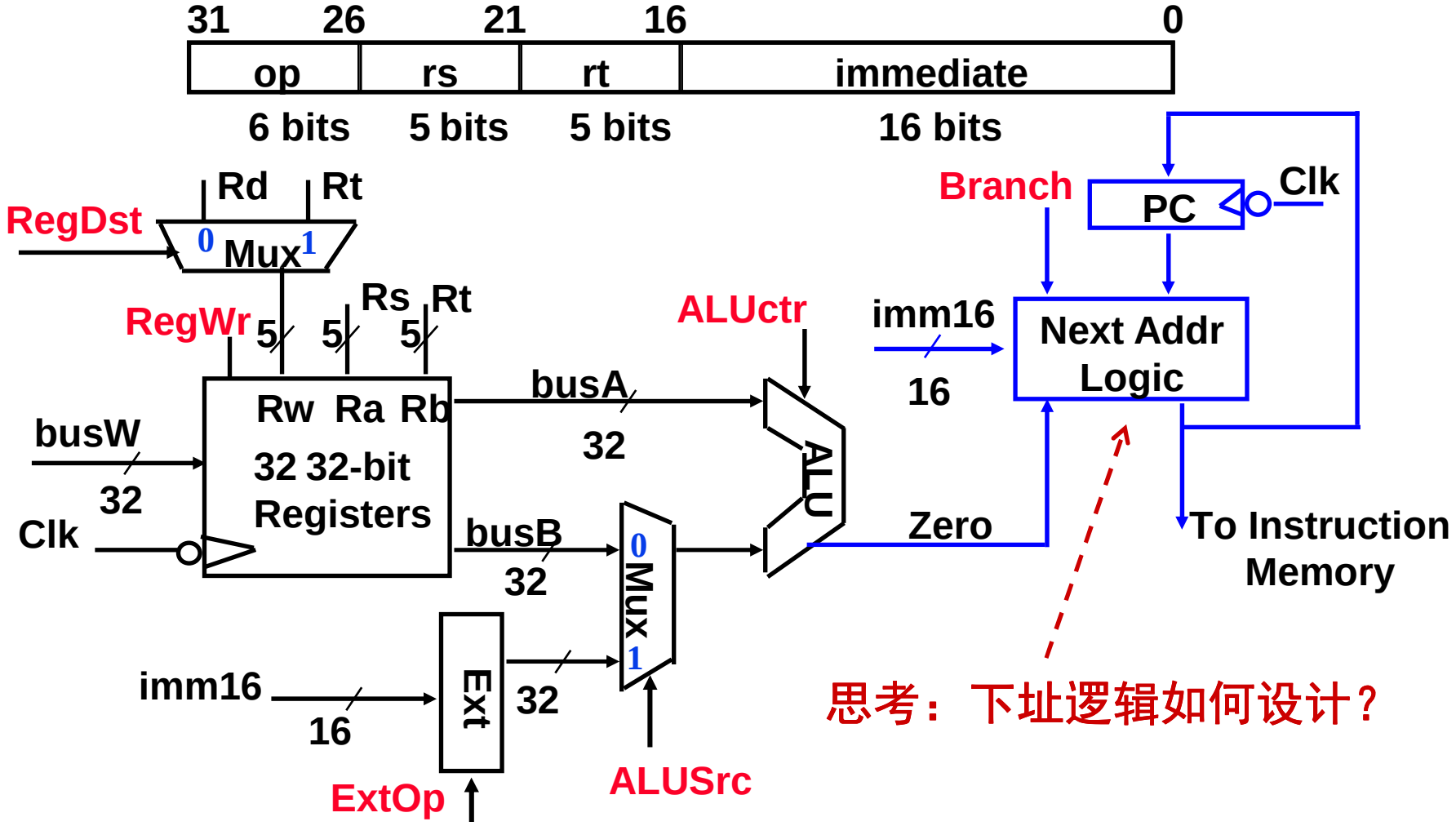
  - M[PC] 取指令（公共操作，取指部件完成）
  - Cond ← R[rs] - R[rt] 做减法比较rs和rt中的内容
  - if (COND eq 0) 计算下地址（根据比较结果，修改PC）
    - PC ← PC + 4 + (SignExt(imm16) x 4)
  - else
    - PC ← PC + 4

思考：立即数的含义是什么？是相对指令数还是相对单元数？

应在原数据通路上加哪些元件和连线？用什么控制信号来控制？

条件转移指令的数据通路

- beq rs, rt, imm16 We need to compare Rs and Rt !



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

下地址计算逻辑的设计

PC是一个32位地址:

顺序执行时: PC<31:0> = PC<31:0> + 4

转移执行时: PC<31:0> = PC<31:0> + 4 + SignExt[Imm16] X 4

采用32位PC时，可用左移2位实现“X4”操作，计算转移地址用2个加法器！用更简便的方式实现如下：

MIPS按字节编址，每条指令为32位，占4个字节，故PC的值总是4的倍数，即后两位为00，因此，PC只需要30位即可。

PC采用30位后，其转移地址计算逻辑变得更加简单。

下地址计算逻辑简化为：

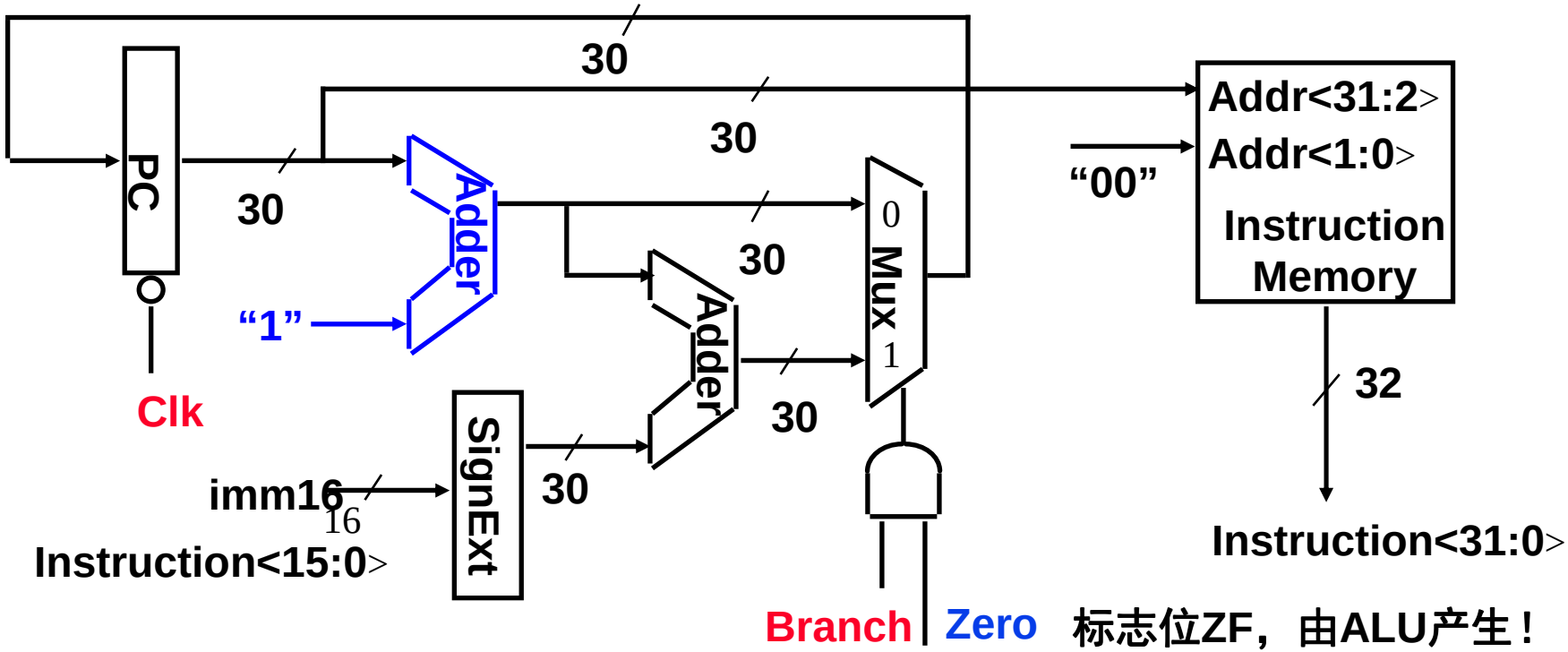
顺序执行时: PC<31:2> = PC<31:2> + 1

转移执行时: PC<31:2> = PC<31:2> + 1 + SignExt[Imm16]

取指令时: 指令地址 = PC<31:2> 串接 “00”

下址逻辑设计方案1: 快速但昂贵

- Using a 30-bit PC:
  - 顺序执行时: PC<31:2> = PC<31:2> + 1
  - 转移执行时: PC<31:2> = PC<31:2> + 1 + SignExt[Imm16]
  - 取指令时: 指令地址 = PC<31:2> concat “00”



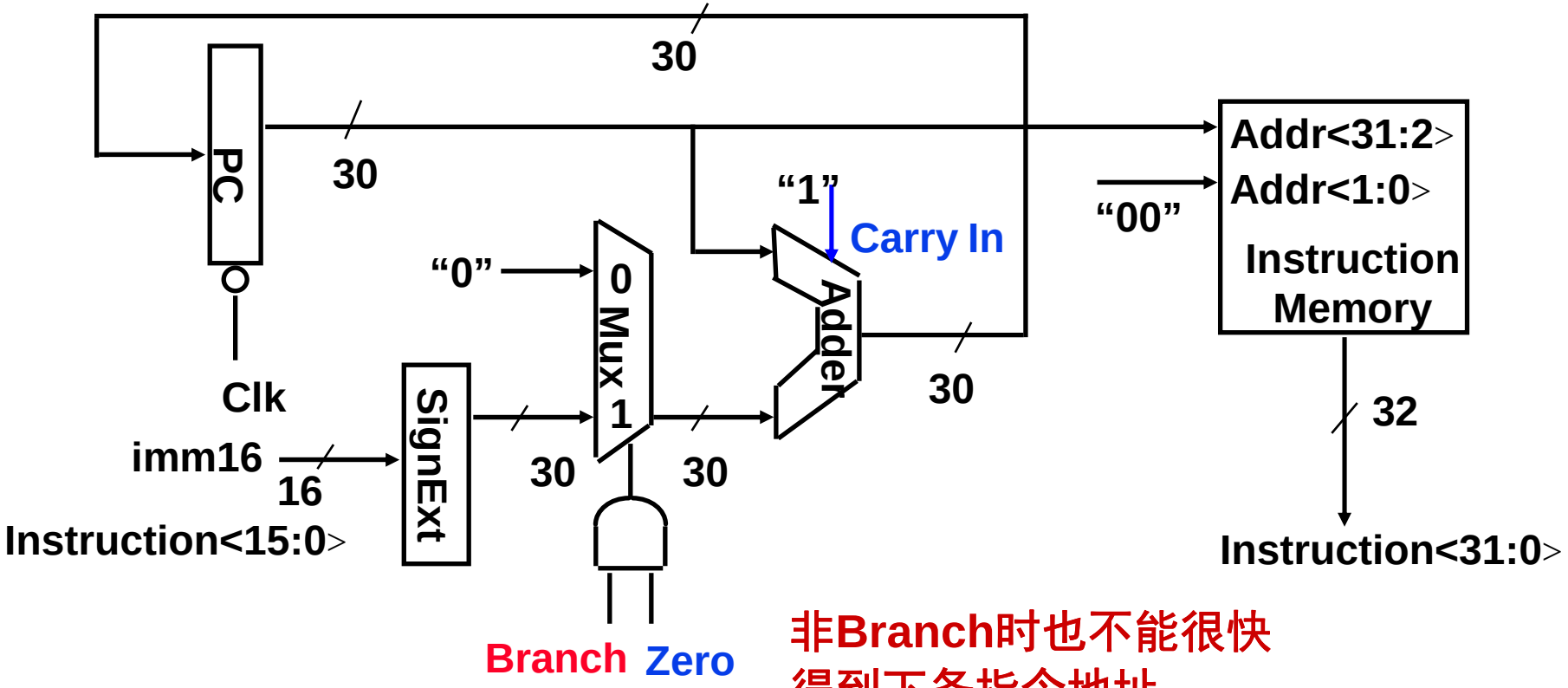
先根据当前PC取指令， 计算的下条指令地址在下一个时钟到来后才能写入PC

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

下址逻辑设计方案2:：慢但便宜

- 为什么慢?
  - 只能等到“Zero”有值后才能进行地址计算
- 对性能有没有影响?
  - 没有，因为Load指令更慢。
- 为什么便宜?
  - “+1”操作用“进位”来实现，节省一个“Adder”

如果是 **bgt**，则如何？

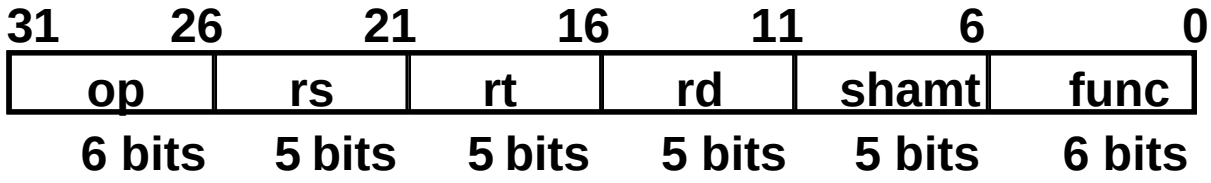


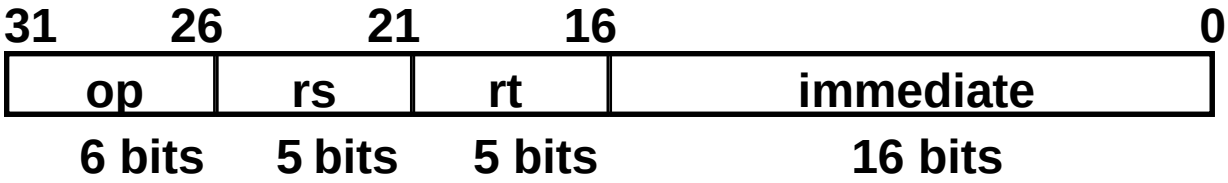
非Branch时也不能很快
得到下条指令地址

无条件转移指令

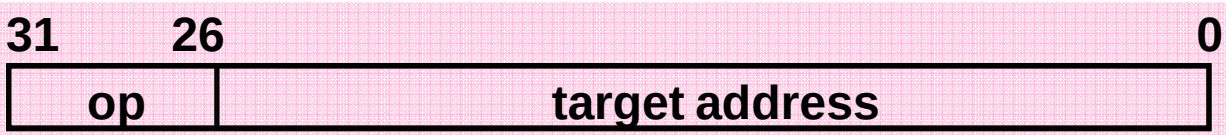
实现目标（7条指令）：

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

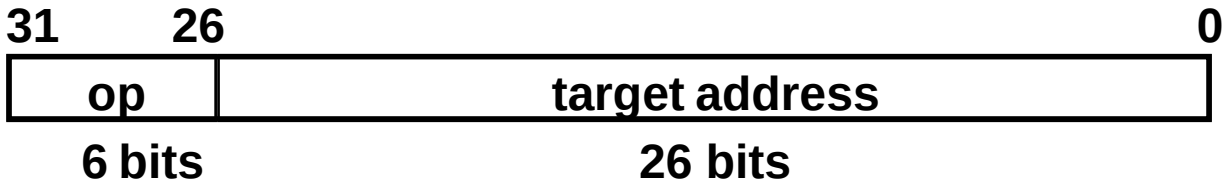




6. 考虑Jump指令（无条件转移指令的代表）



RTL: The Jump Instruction

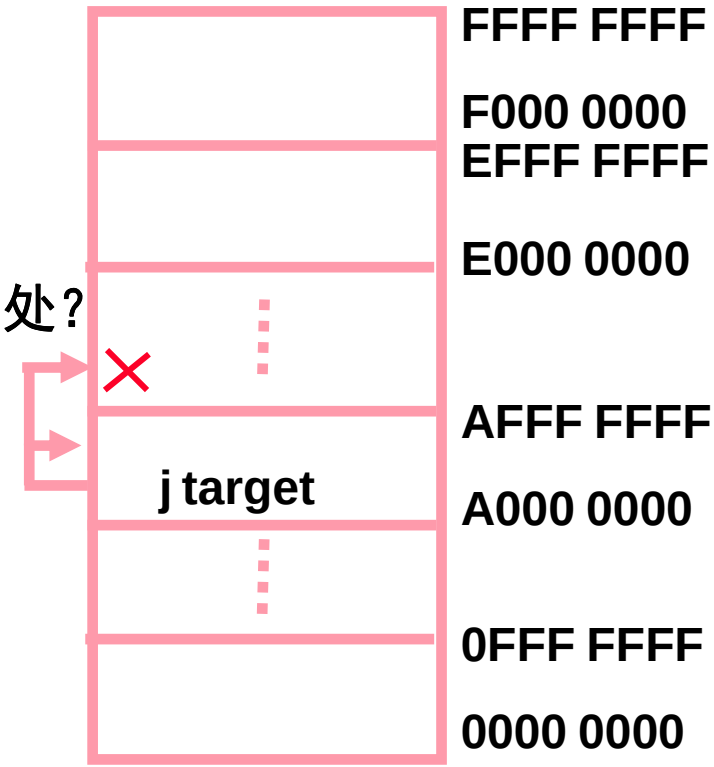


- j target
  - M[PC] 取指令（公共操作，取指部件完成）
  - PC<31:2> ← PC<31:28> 串接 target<25:0> 计算目标地址

想一想：跳转指令的转移范围有多大？

是当前指令后面的0x000 0000~0xFFF FFFC 处？

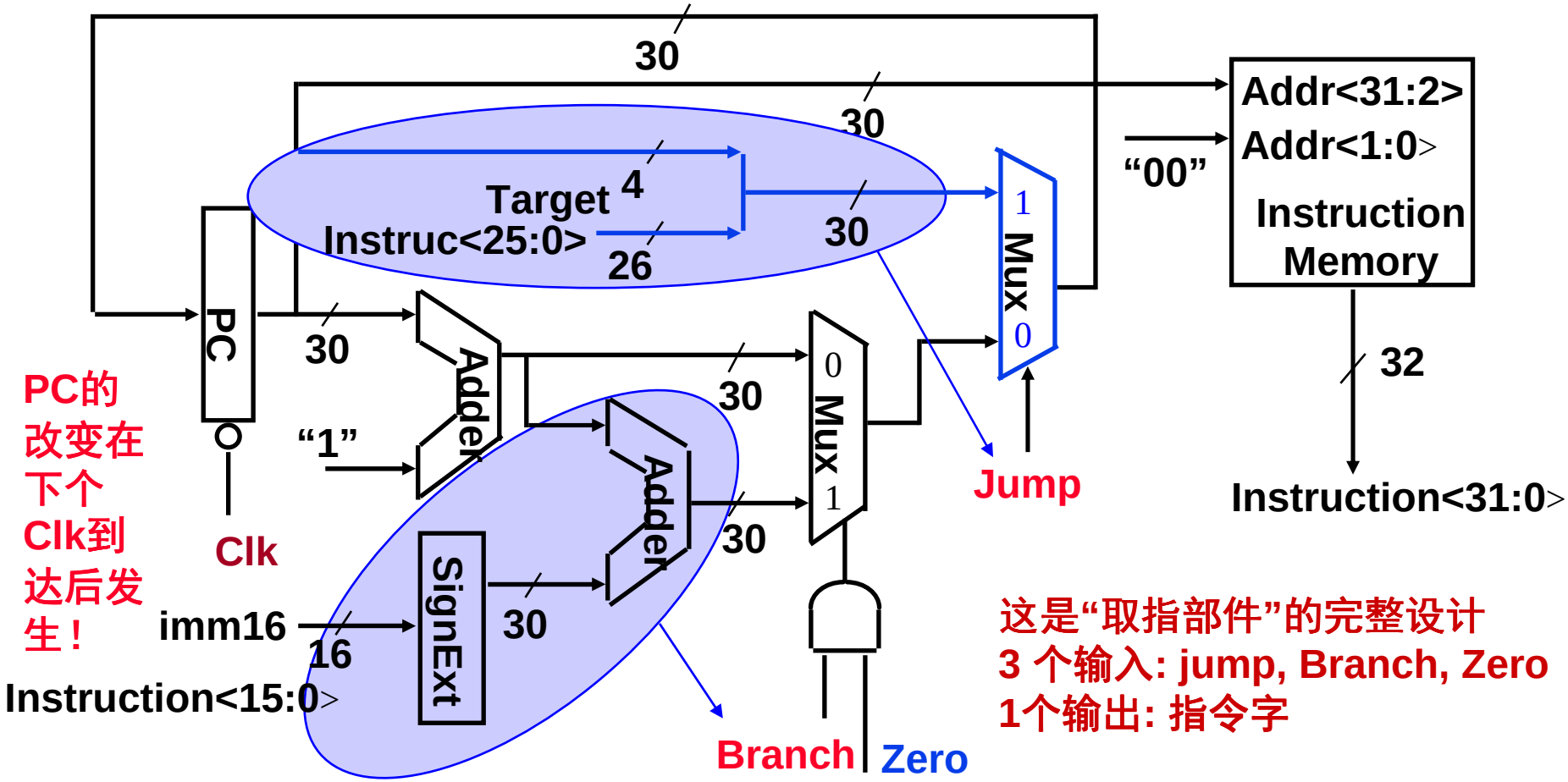
不对！它不是相对寻址，而是绝对寻址



思考：应在原数据通路上加哪些元件和连线？用什么控制信号来控制？

Instruction Fetch Unit: 取指令部件

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

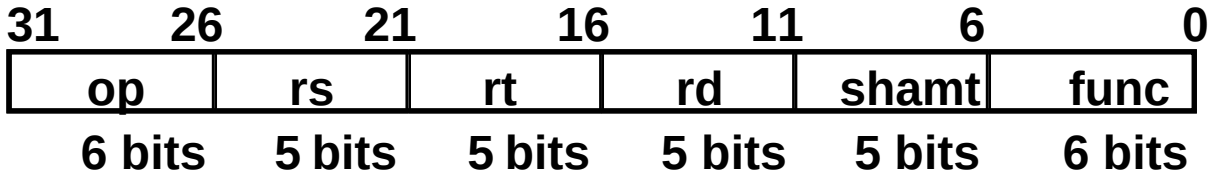


RegDst, RegWr, ALUctr, ExtOp, ALUSrc, MemWr, MemtoReg, Branch, Jump 各取何值?

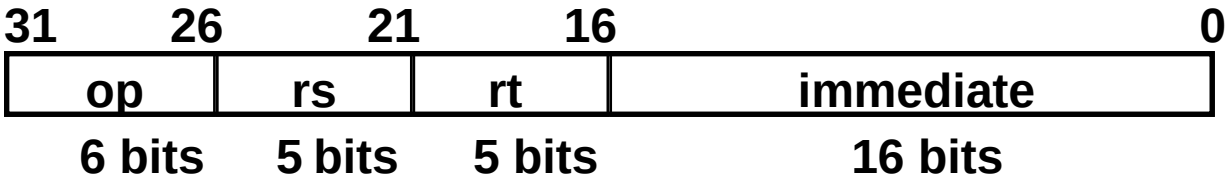
RegDst=ExtOp=ALUSrc=MemtoReg=ALUctr=x, RegWr=0, MemWr=0, Branch=0, Jump=1

The MIPS Subset(考察实现以下指令的数据通路)

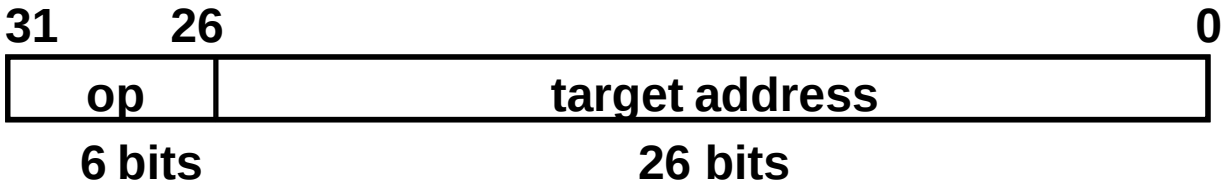
- ADD and subtract
  - add rd, rs, rt
  - sub rd, rs, rt



- OR Immediate:
  - ori rt, rs, imm16

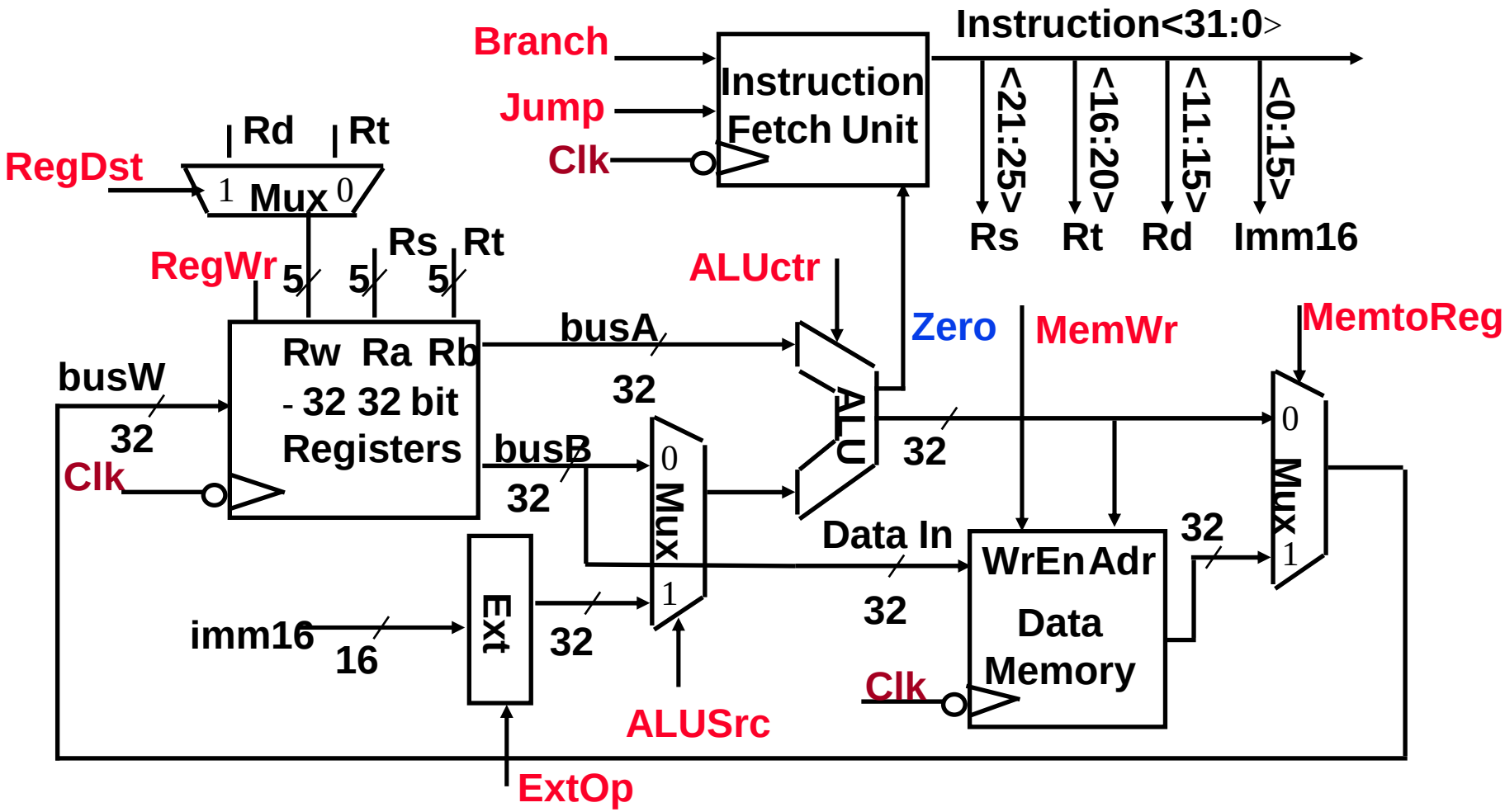


- LOAD and STORE
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- BRANCH:
  - beq rs, rt, imm16
- JUMP:
  - j target



所有指令的数据通路都已设计好，合起来的数据通路是什么样的？

Putting it All Together: A Single Cycle Datapath



指令执行结果总是在下个时钟到来时开始保存在 寄存器 或 存储器 或 PC 中！

下一讲考虑：如何产生控制信号！（这就是控制器的设计内容）

第一讲小结

- CPU设计直接决定了时钟周期宽度和CPI，所以对计算机性能非常重要！
- CPU主要由数据通路和控制器组成
  - 数据通路：实现指令集中所有指令的操作功能
  - 控制器：控制数据通路中各部件进行正确操作
- 数据通路中包含两种元件
  - 操作元件（组合电路）：ALU、MUX、Ext.、Adder、Reg/Mem Read等
  - 状态 / 存储元件（时序电路）：PC、Reg/Mem Write
- 数据通路的定时
  - 数据通路中的操作元件没有存储功能，其操作结果必须写到存储元件中
  - 在时钟到达后clk-to-Q时存储元件开始更新状态
- MIPS指令集的一个子集作为CPU的实现目标
  - 公共操作：取指令和PC+4
  - 下址计算：30位PC，三路选择：顺序、Branch（结合标志Zero）、Jump
  - R型：ALU两个操作数来自rs和rt，结果写到rd
  - 访存：符号扩展，数据在rt和主存单元中交换
  - 立即数：0扩展后的操作数送到ALU的一个输入端

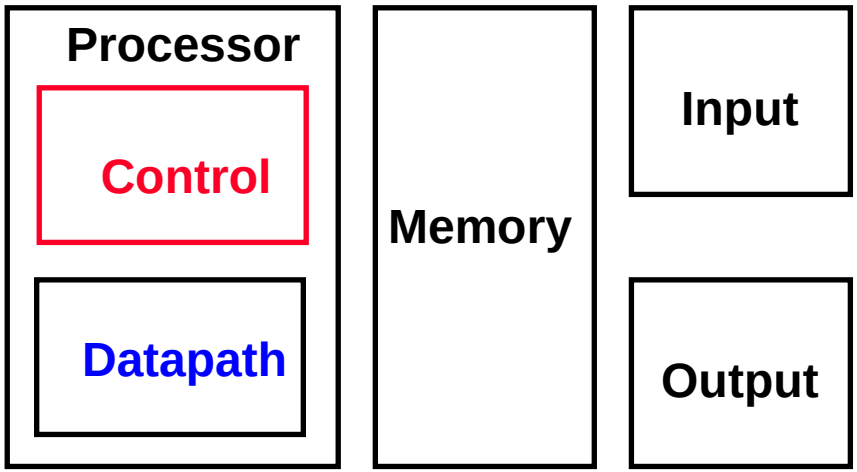
第二讲 单周期控制器的设计

主 要 内 容

- 考察每条指令在数据通路中的执行过程和涉及到的控制信号的取值
  - 公共操作：取指令和计算下址**PC**
  - **R-Type**指令（**add / sub**）
  - 立即数指令 （**ori**）
  - 访存指令（**lw / sw**）
  - 分支指令 （**beq**）
  - 跳转指令 （**j**）
- 汇总各指令的控制信号取值
  - 分两类控制信号：直接送往数据通路 / 送往局部控制单元
- 分析**ALU**操作对应的控制信号与**func**字段之间的关系
- 设计**ALU**局部控制单元
- 设计主控制单元

The Big Picture: Where are We Now?

- The Five Classic Components of a Computer



- 下一个目标：设计单周期数据通路的控制器。

设计方法：

1) 根据每条指令的功能，分析控制信号的取值，并在表中列出。
2) 根据列出的指令和控制信号的关系，写出每个控制信号的逻辑表达式。

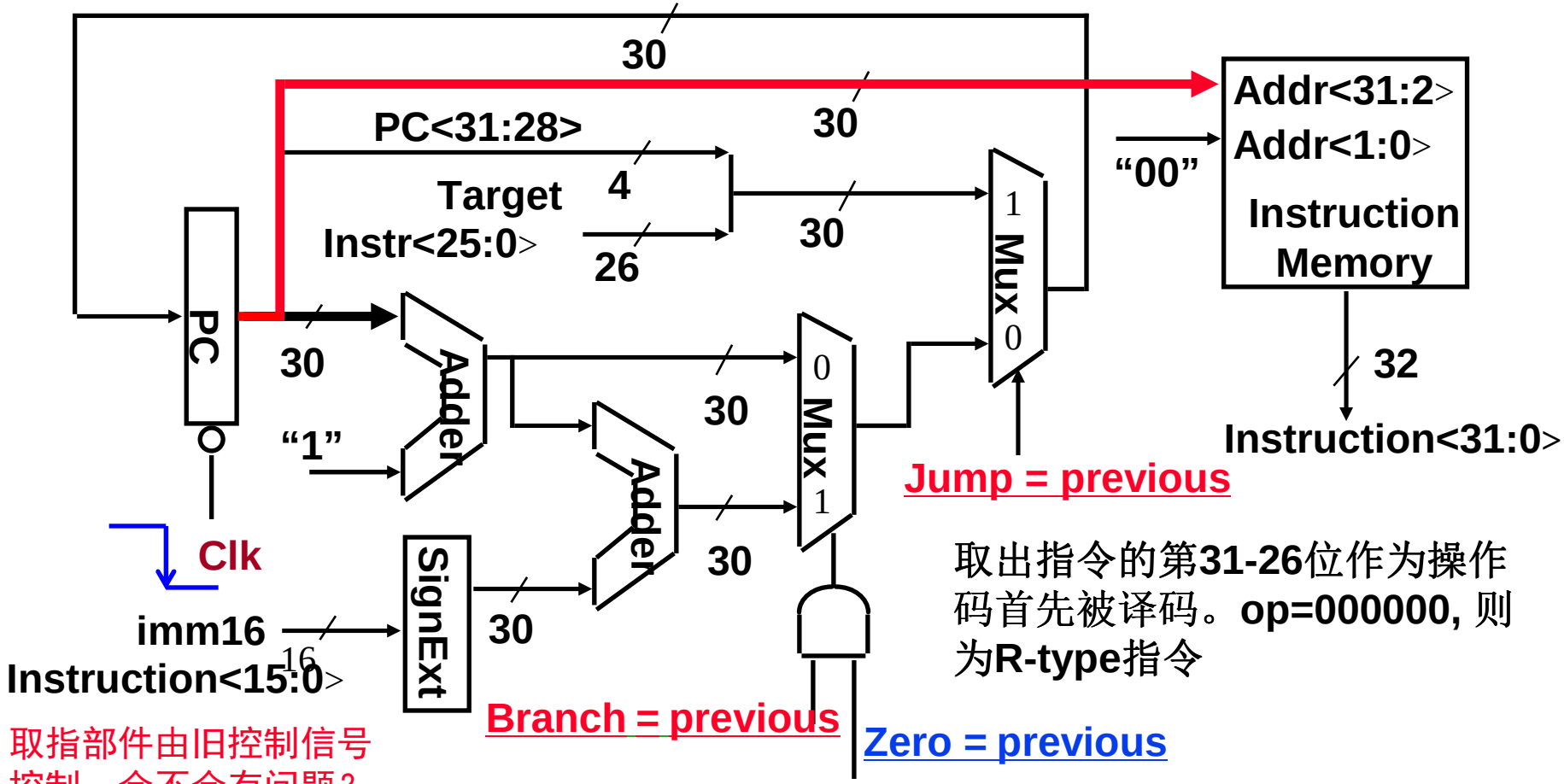
Add / Sub操作开始时取指部件中的动作

取指令: Instruction ← M[PC]

- 所有指令都相同

新指令还没有取出译码，所以控制信号的值还是原来指令的旧值。

新指令还没有执行，所以标志也为旧值。



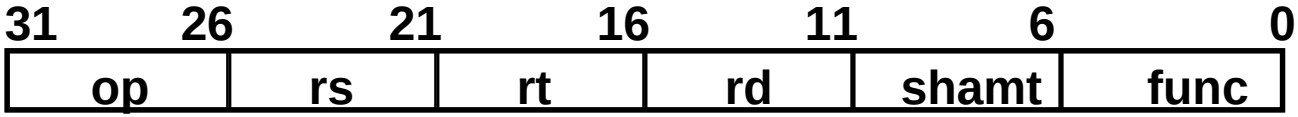
取出指令的第31-26位作为操作码首先被译码。op=000000, 则为R-type指令

取指部件由旧控制信号控制，会不会有问题？

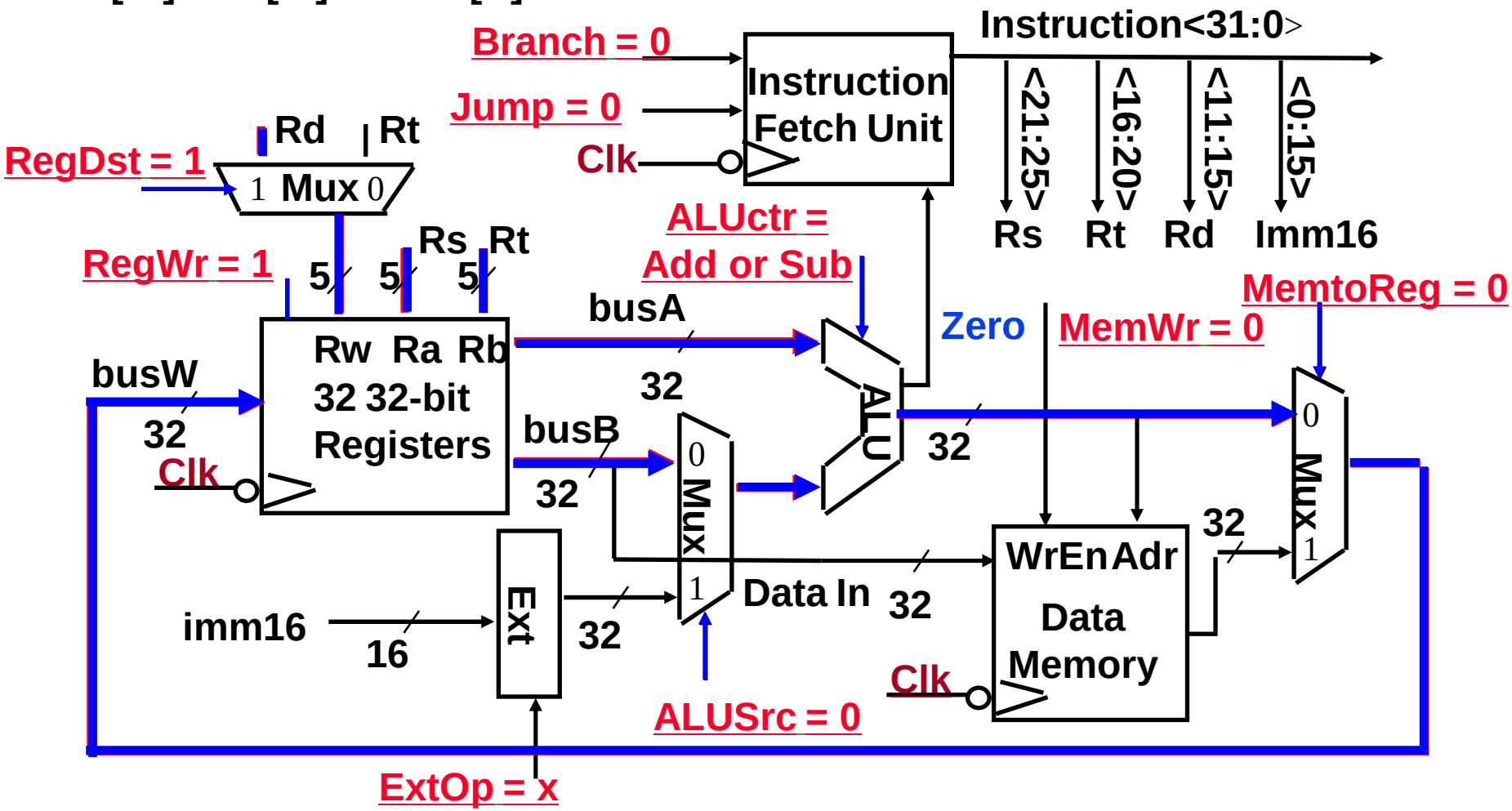
因为在下个Clk到来之前PC输入端的值不会写入只要保证下个Clk来之前能产生正确的PC即可！

sing 没有问题！ Why?

指令译码后R型指令（Add / Sub）操作过程

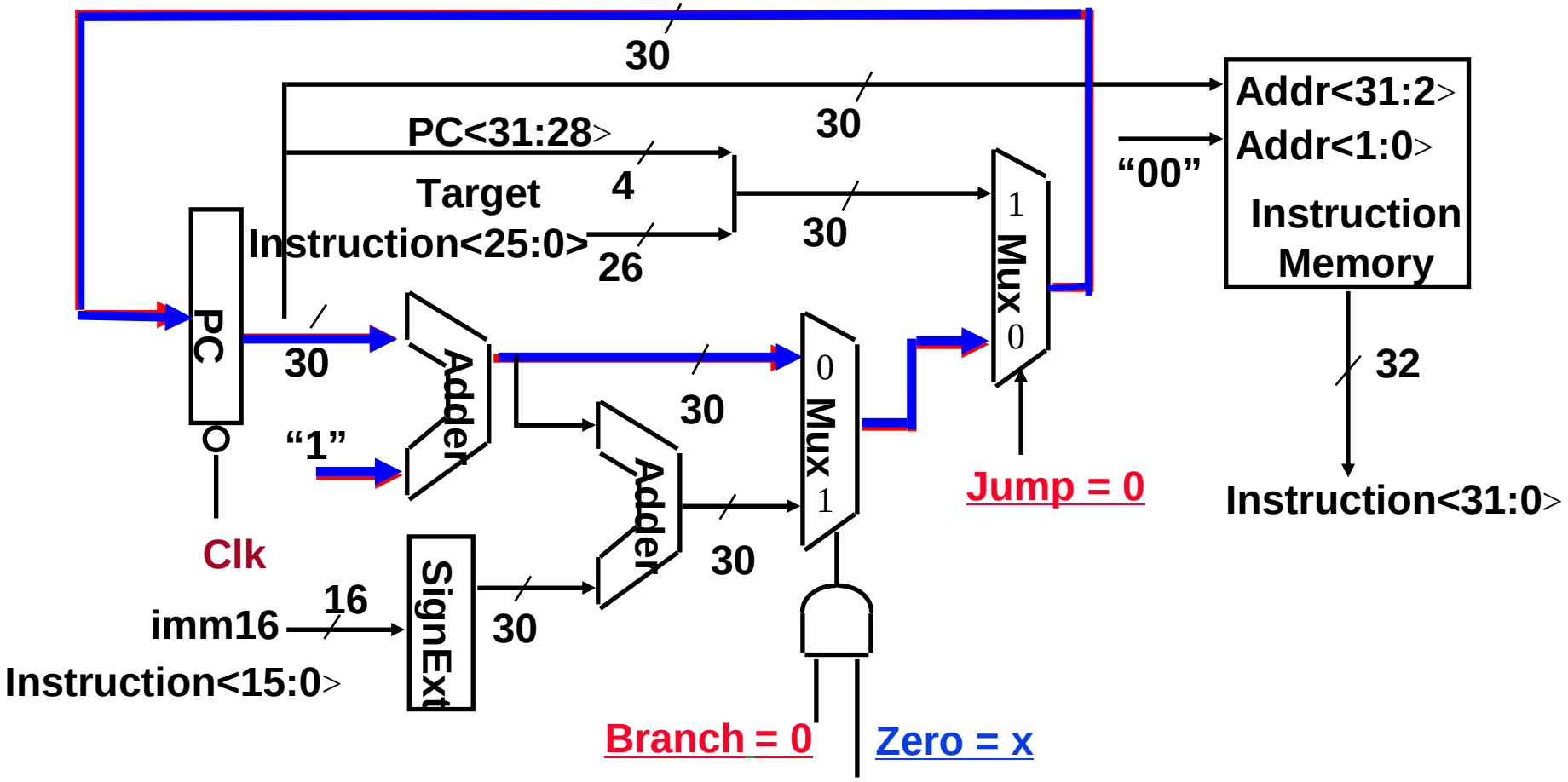


° R[rd] ← R[rs] + / - R[rt]



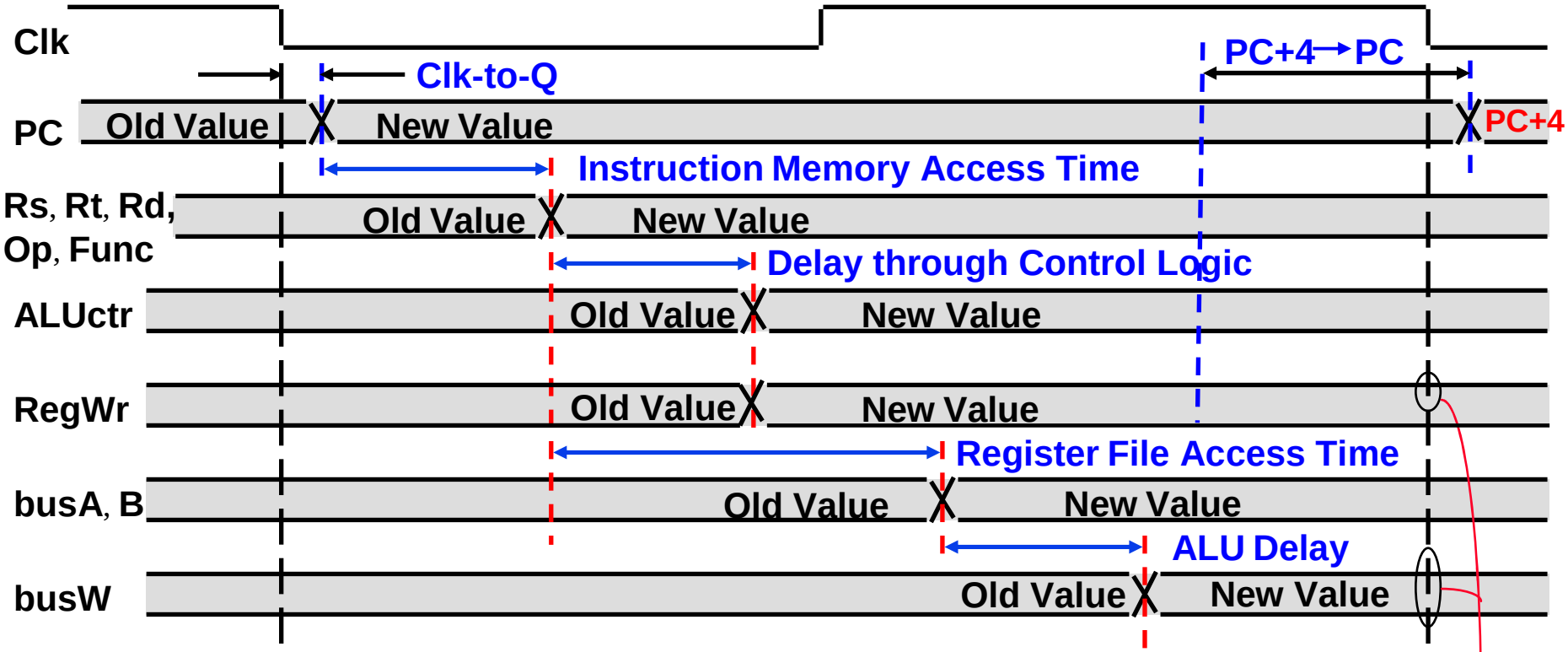
R型指令（Add /Sub）最后阶段取指部件中的动作

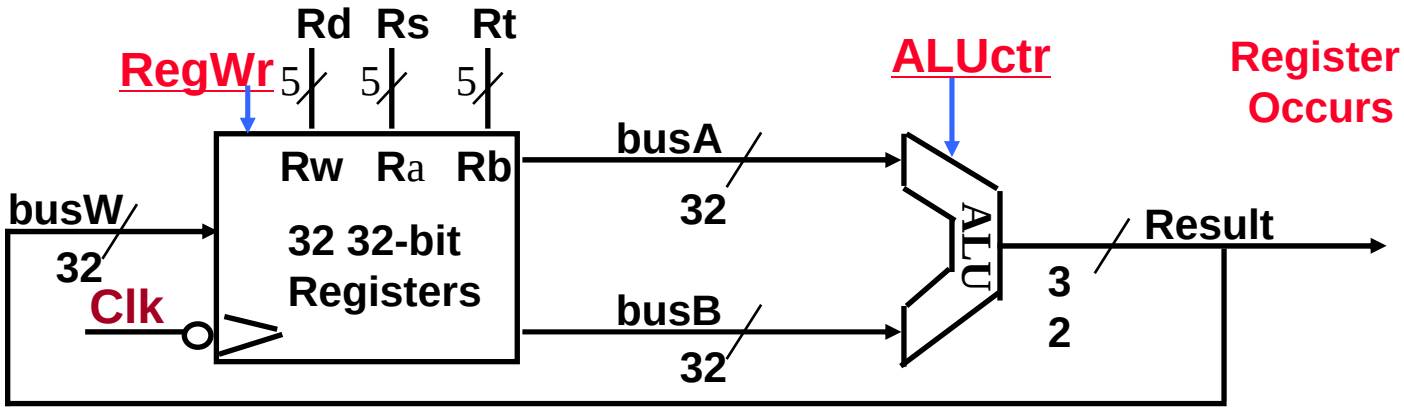
- PC ← PC + 4
  - 除 Branch and Jump以外的指令都相同



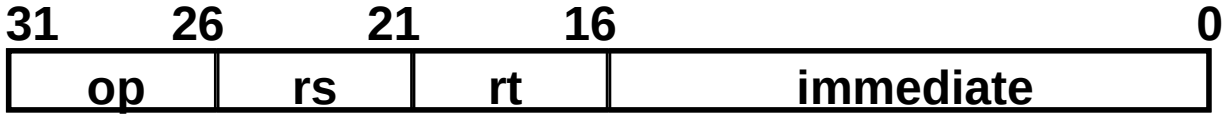
因为新的控制信号保证了正确的PC值的产生，
在足够长的时间后，下个时钟Clk到来！

Register-Register（R型指令） Timing

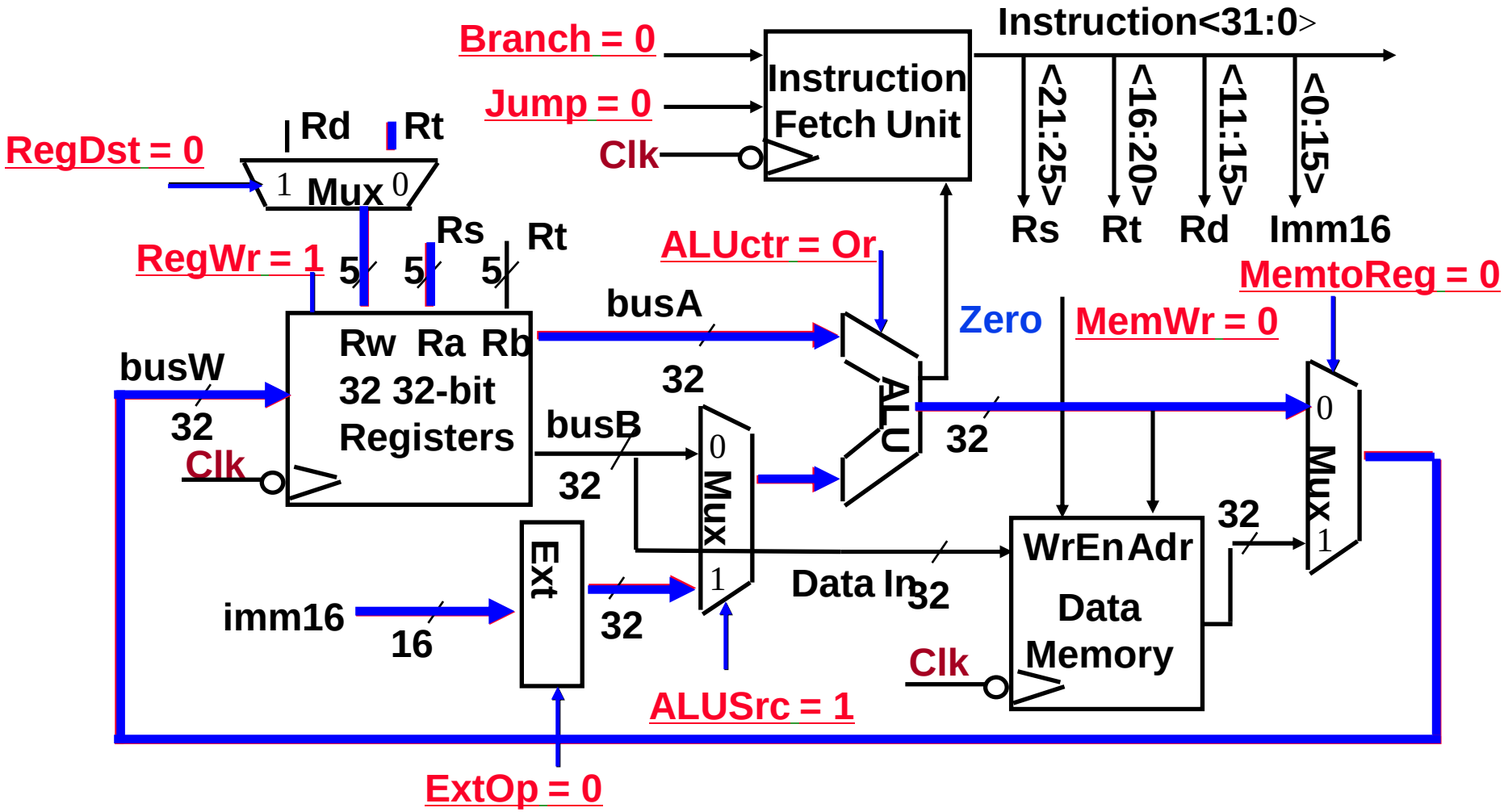




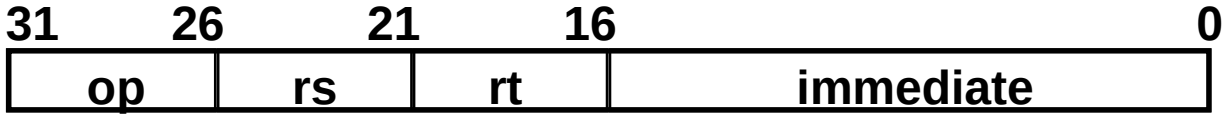
ori 指令译码后的执行过程



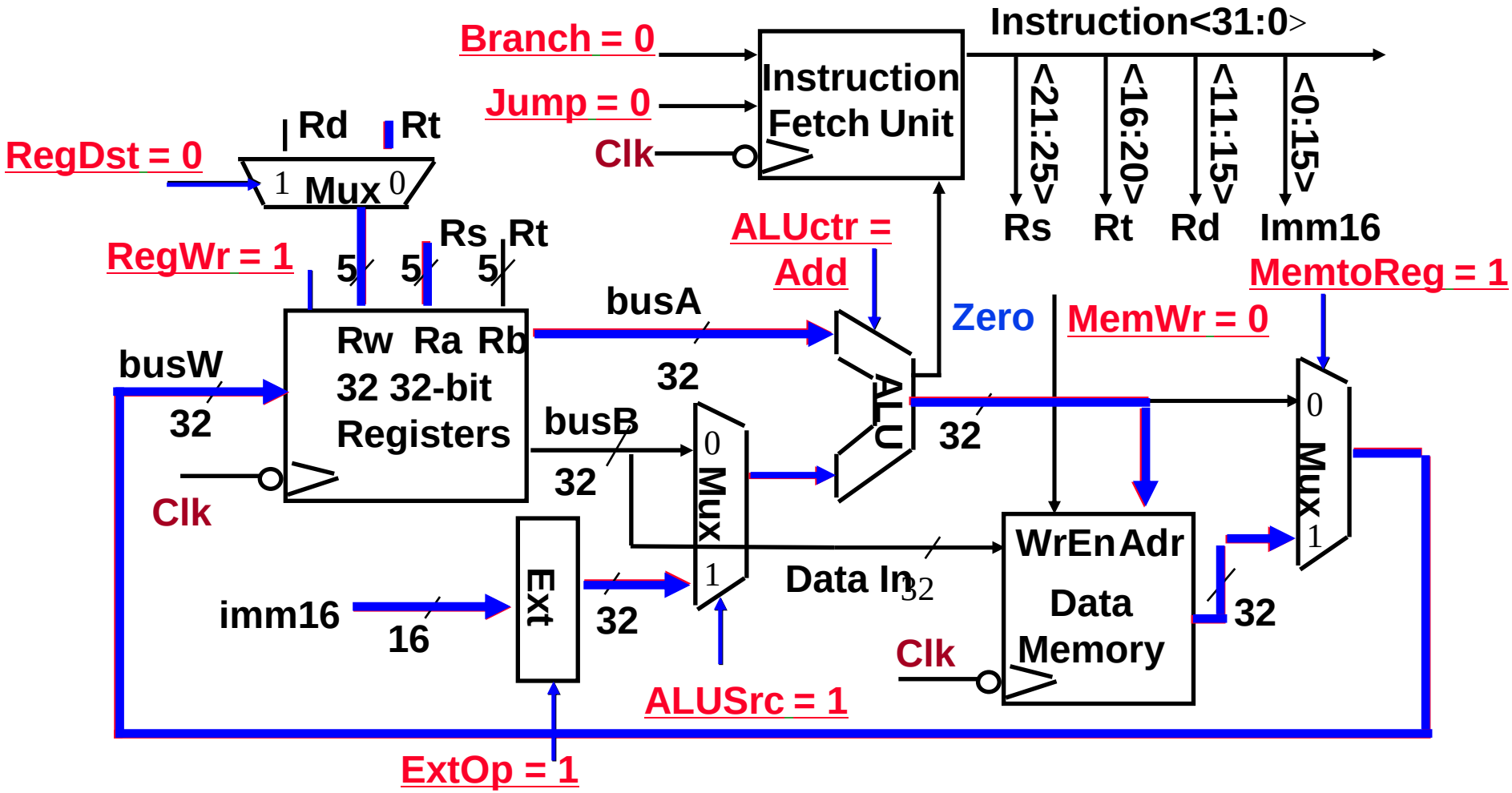
° R[rt] ← R[rs] or ZeroExt[Imm16]



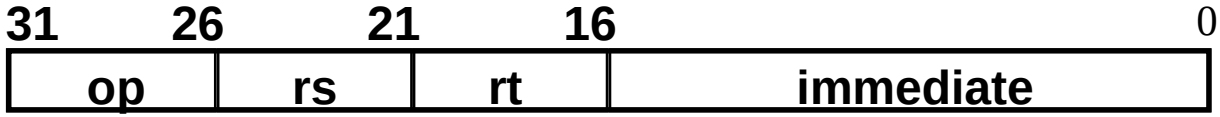
Load指令译码后的执行过程



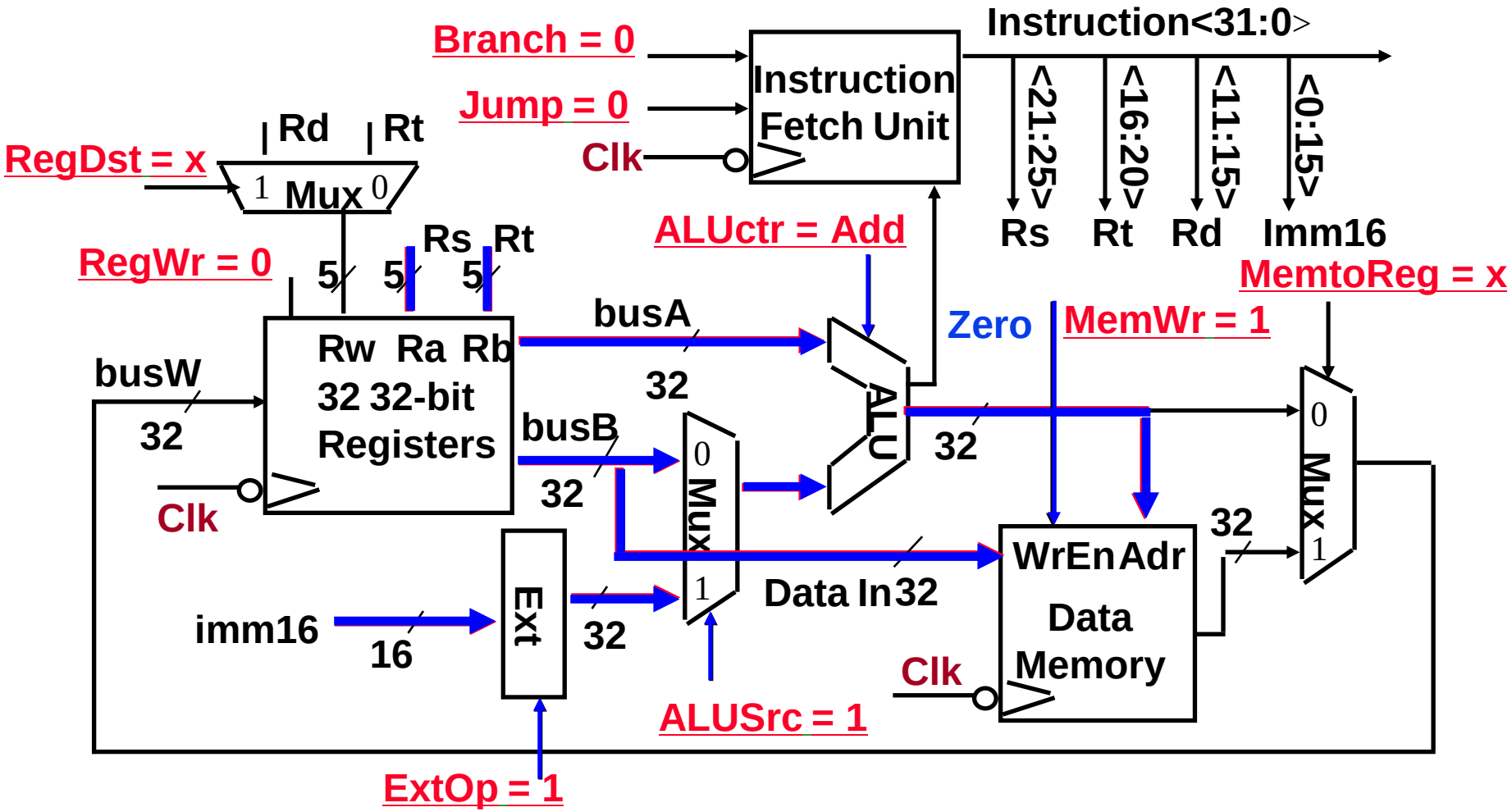
- R[rt] ← Data Memory {R[rs] + SignExt[imm16]}



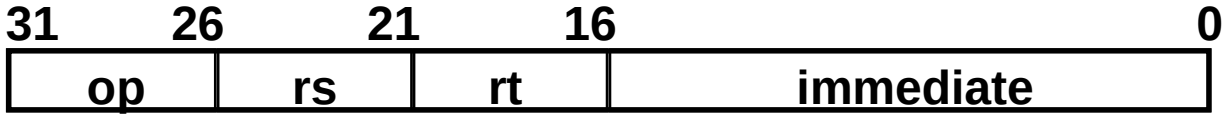
Store指令译码后的执行过程



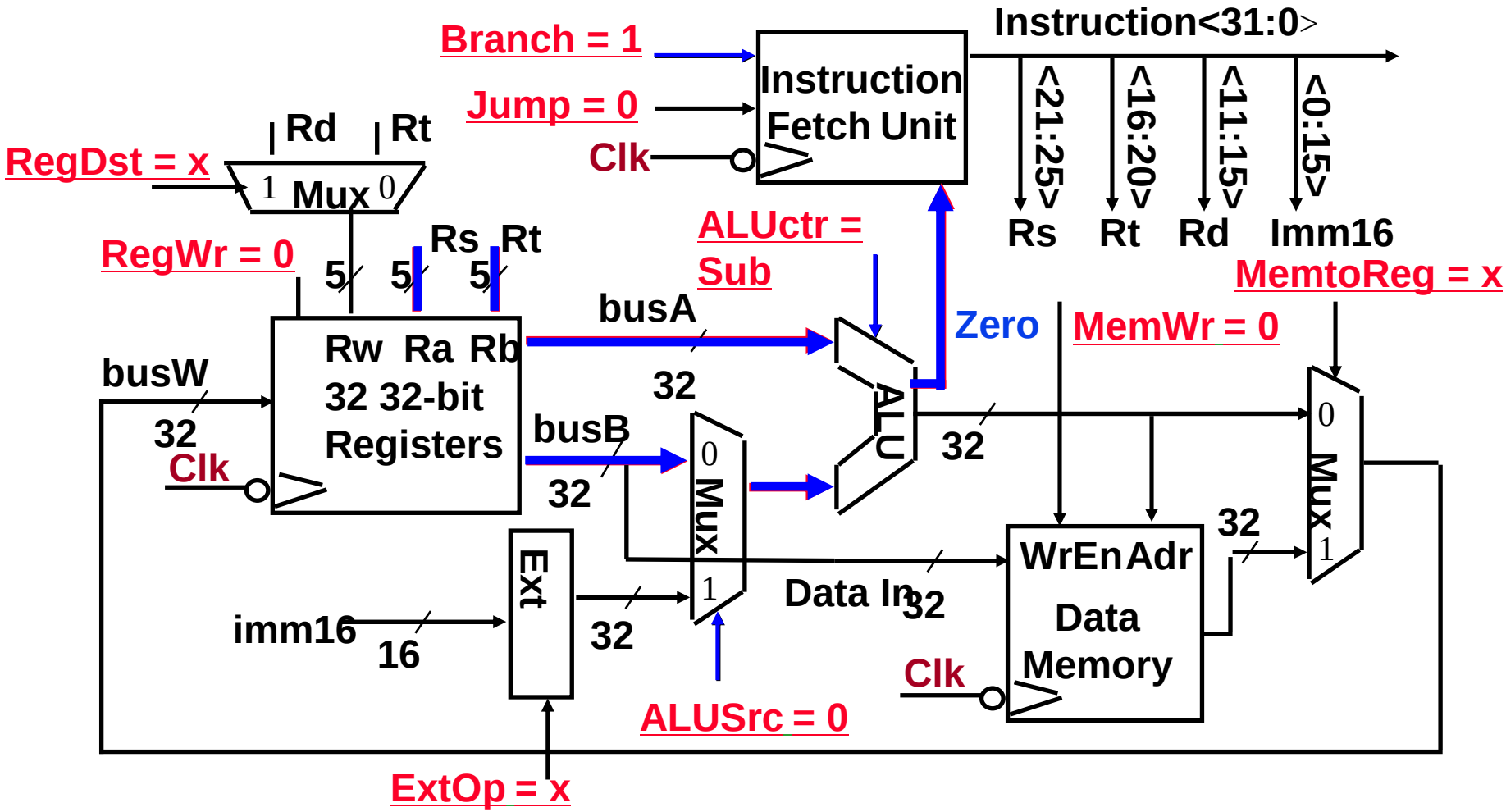
- M{R[rs] + SignExt[imm16]} ← R[rt]



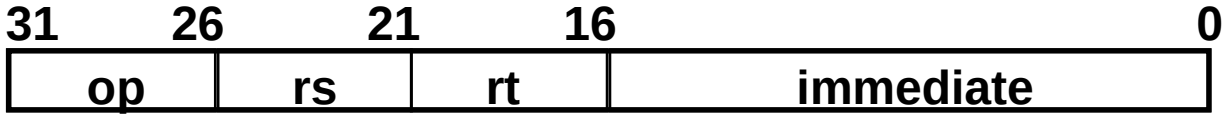
Branch指令译码后的执行过程



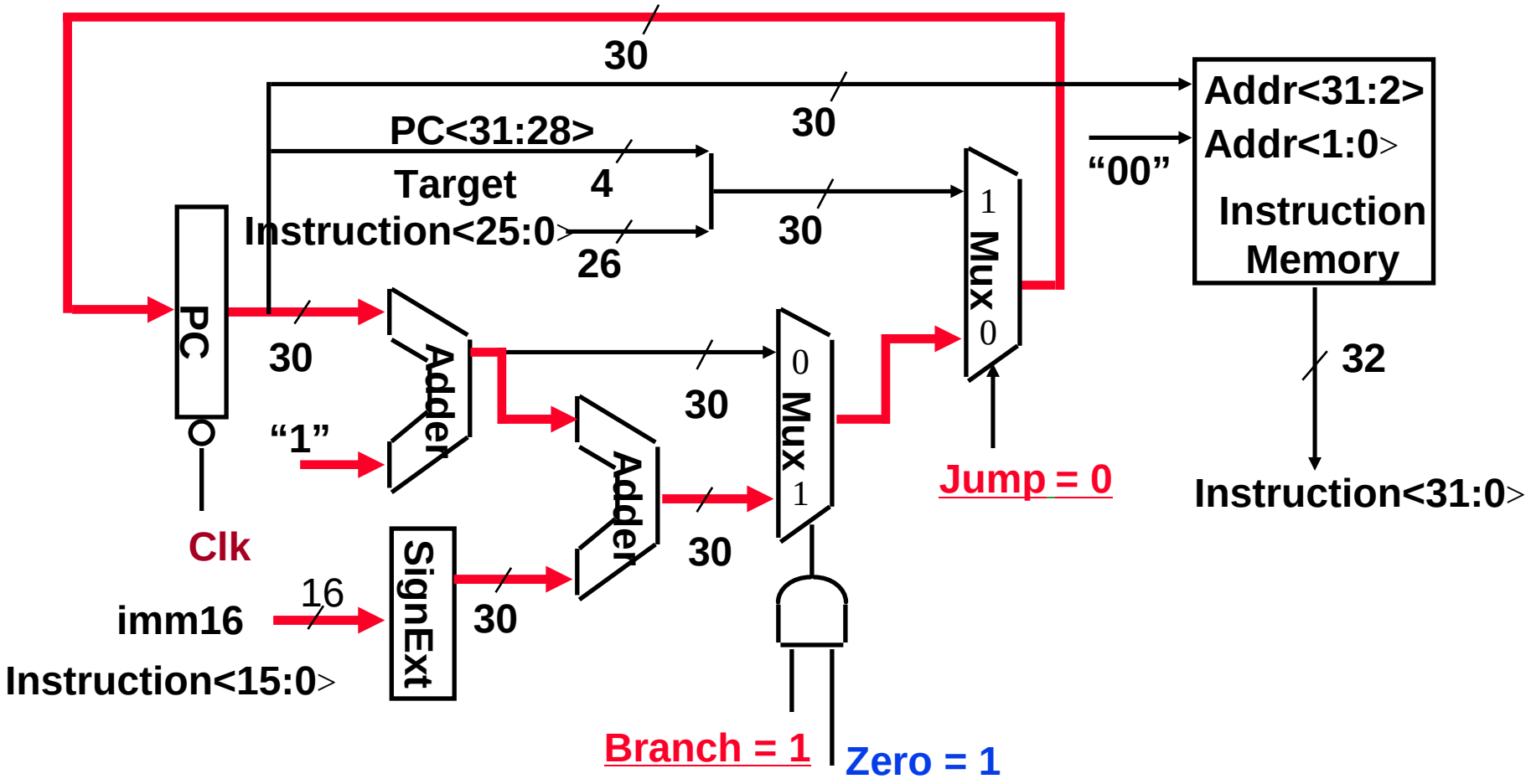
° if (R[rs] - R[rt] == 0) then Zero ← 1 ; else Zero ← 0



Branch指令最后阶段取指部件中的动作



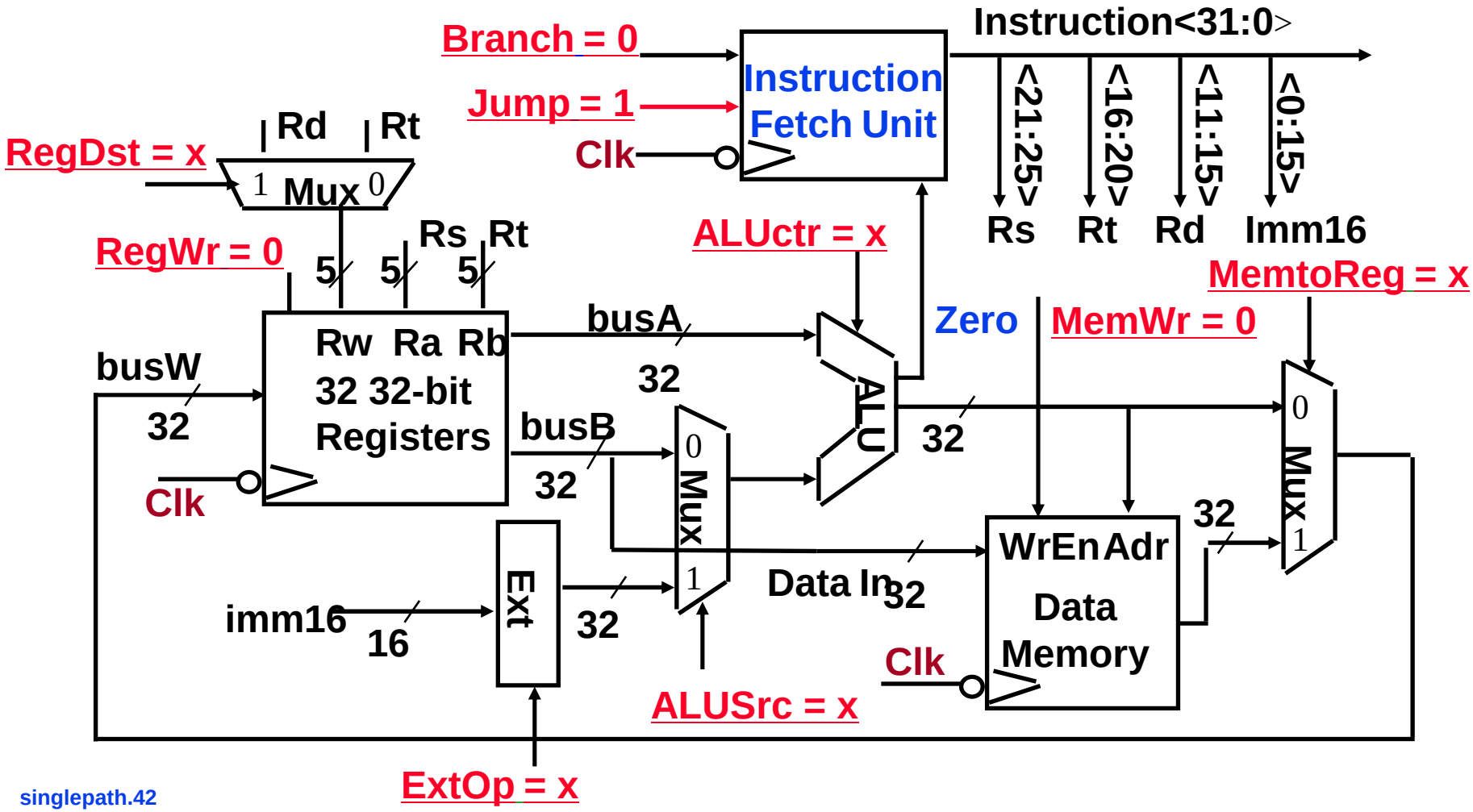
° if (Zero == 1) then PC = PC + 4 + SignExt[imm16]*4 ; else PC = PC + 4



Jump指令译码后的执行过程

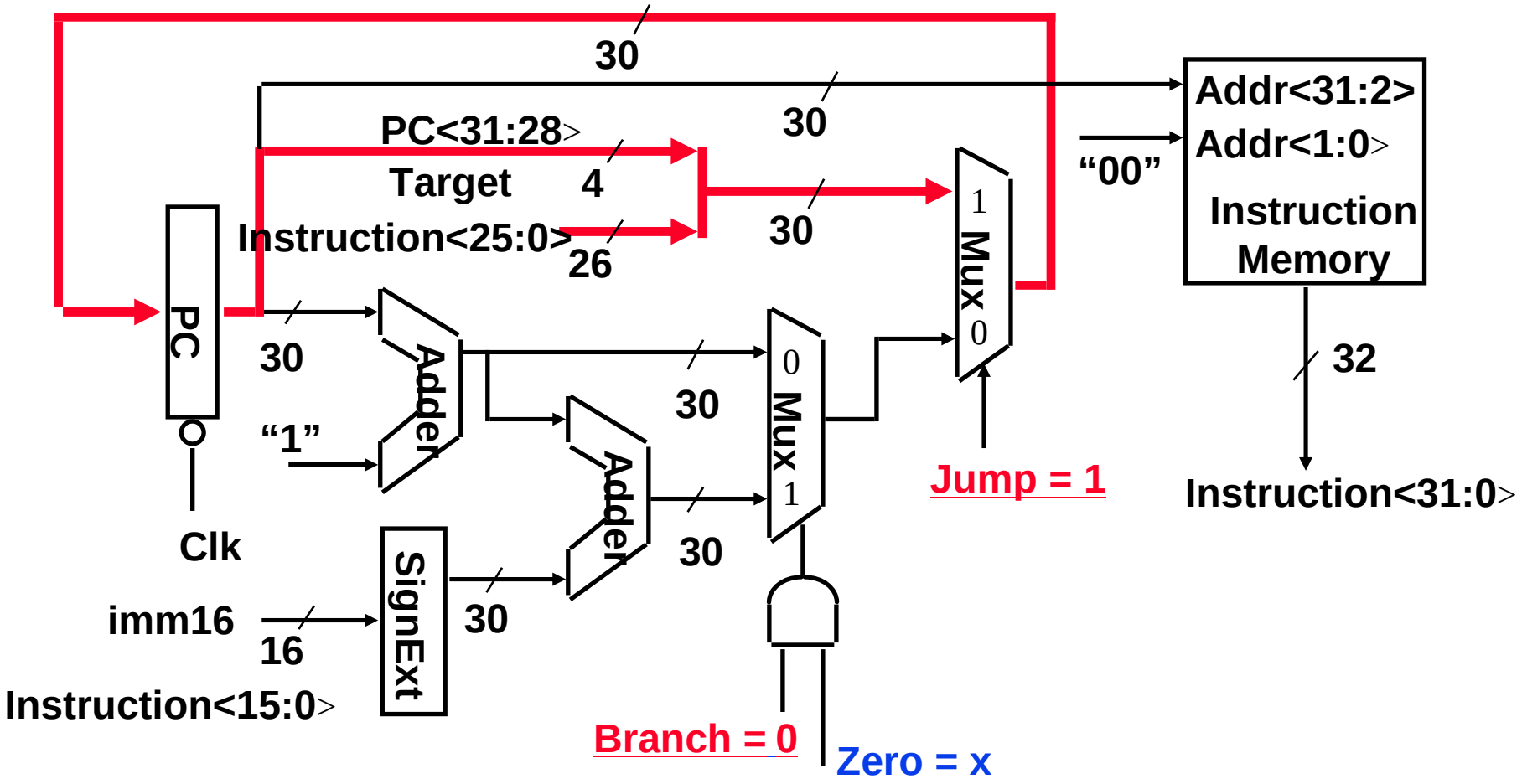
- IFU中目标地址送PC，其他什么都不做（只要保证存储部件不发生写的动作）

  如何保证存储部件不发生写？



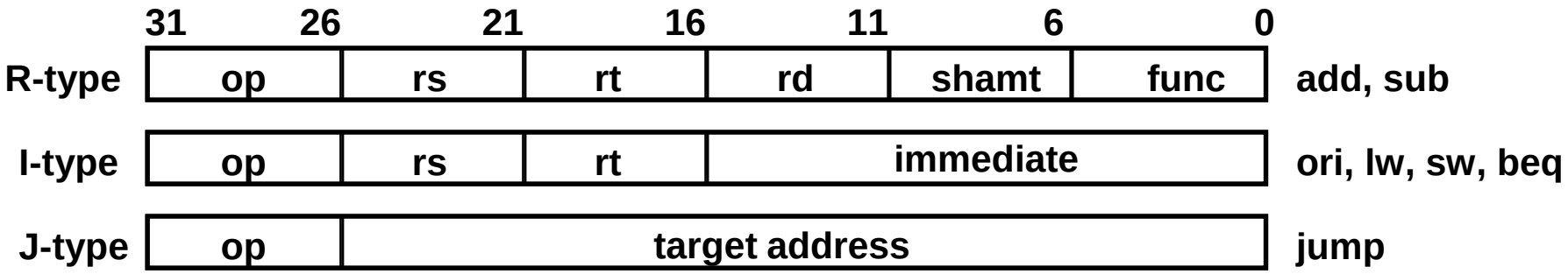
Jump指令结束前IFU中的动作

° PC ← PC<31:29> concat target<25:0> concat “00”



综合分析结果，得到如下指令与控制信号的关系表

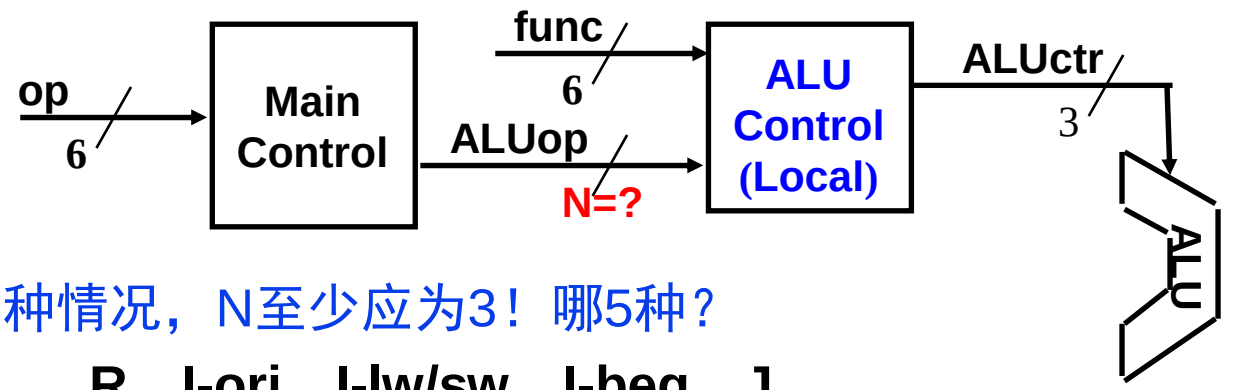
func / op	10 0000	10 0010	We Don't Care :-)				
	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	add	sub	ori	lw	sw	beq	jump
RegDst	1	1	0	0	x	x	x
ALUSrc	0	0	1	1	1	0	x
MemtoReg	0	0	0	1	x	x	x
RegWrite	1	1	1	1	0	0	0
MemWrite	0	0	0	0	1	0	0
Branch	0	0	0	0	0	1	0
Jump	0	0	0	0	0	0	1
ExtOp	x	x	0	1	1	x	x
ALUctr<2:0>	Add	Subtr	Or	Add	Add	Subtr	xxx



主控制单元和ALU局部控制单元

MIPS指令格式中指示操作性质的字段有两个：op(主控) 和 func（ALU局控）。

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

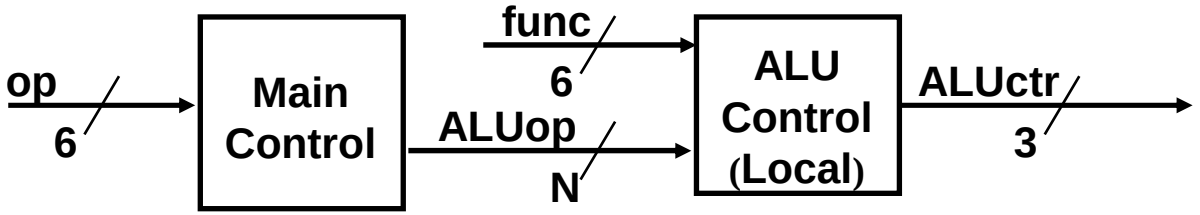


ALUctr的值取决于ALUop和func，其他控制信号仅取决于op。

ALUop有5种情况，N至少应为3！哪5种？

R、I-ori、I-lw/sw、I-beq、J

ALUop和“func”字段的译码

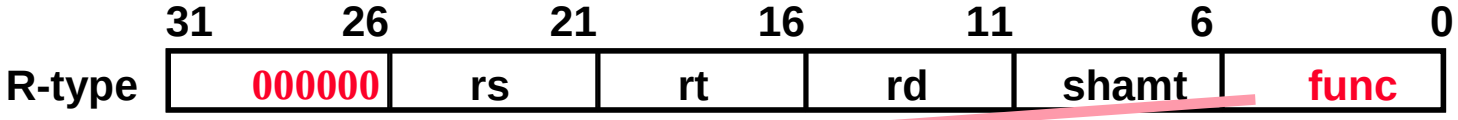


ALUop的编码定义如下：

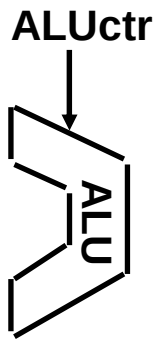
	R-type	ori	lw	sw	beq	jump
ALUop (Symbolic)	“R-type”	Or	Add	Add	Subtr	xxx
ALUop<2:0>	1 xx	0 10	0 00	0 00	0x1	xxx

问题：**ALUop能否仅用2位？** R-Type取1xx，不会发生编码冲突！

能！因为**jump**时任意，故可仅用两位：**R:11, I-ori:10, I-beq:01, I-lw/sw:00, J-xx**



func<5:0>	Instruction Operation	ALUctr<2:0>	ALU Operation
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与func后4位有关，需建立ALUctr 与ALUop和func后四位之间对应关系

ALUctr控制信号的真值表

建立ALUop、func后4位和ALUctr之间的关系表
由关系表可得出ALUctr的逻辑表达式:

ALUctr[i] = f (ALUop[i], func[i])

ALUop (Symbolic)	R-type	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>	Instruction Op.
0000	add
0010	subtract
0100	and
0101	or
1010	set-on-less-than

ALUop			func				ALU Operation	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

头三行是非R-Type，操作由ALUop决定，与func无关。R-Type时，操作完全由func决定。

ALUctr可用更多位数，这样便于扩充，例如，可加入异或、移位等操作。

The Logic Equation for ALUctr<0>

从前面的真值表中，抽取出ALUctr[0]为1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

This makes func<3> a don't care

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

The Logic Equation for ALUctr<1>

从前面的真值表中，抽取出ALUctr[1]为1的行

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

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

The Logic Equation for ALUctr<2>

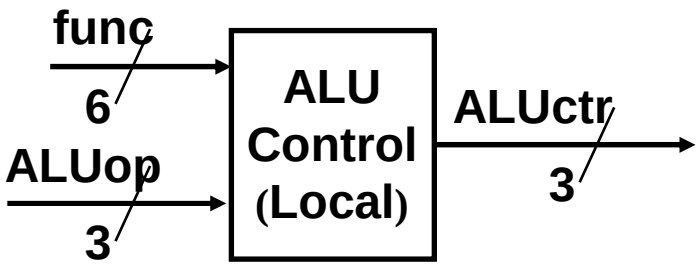
从前面的真值表中，抽取出ALUctr[2]为1的行

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

- **ALUctr<2> = !ALUop<2> & ALUop<1> & !ALUop<0>**

  + ALUop<2> & !func<3> & func<2> & !func<1> & func<0>

局部ALU控制单元逻辑

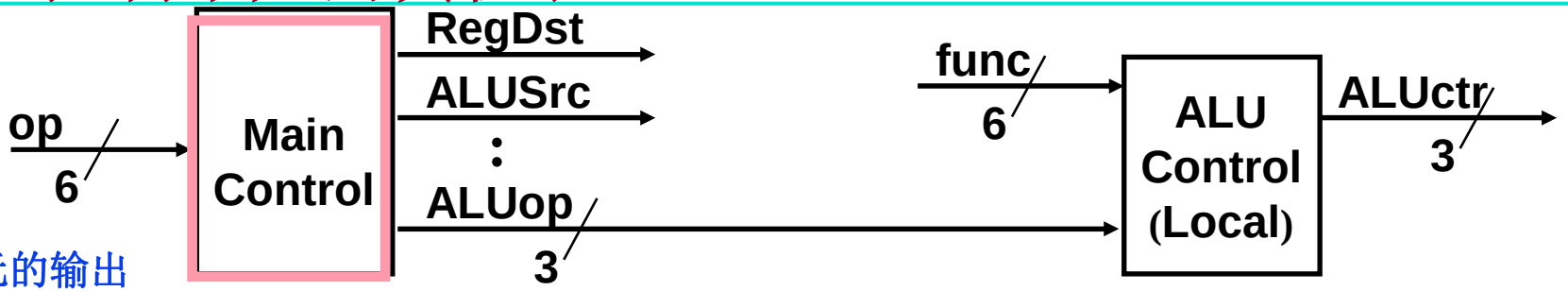


总结前面的结果，得到：

- 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>

根据以上逻辑方程，可实现局部ALU控制单元！

主控制单元的真值表



主控单元的输出

主控单元的输入

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 (Symbolic)	"R-type"	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

考察每个控制信号的逻辑方程（如：RegWrite）

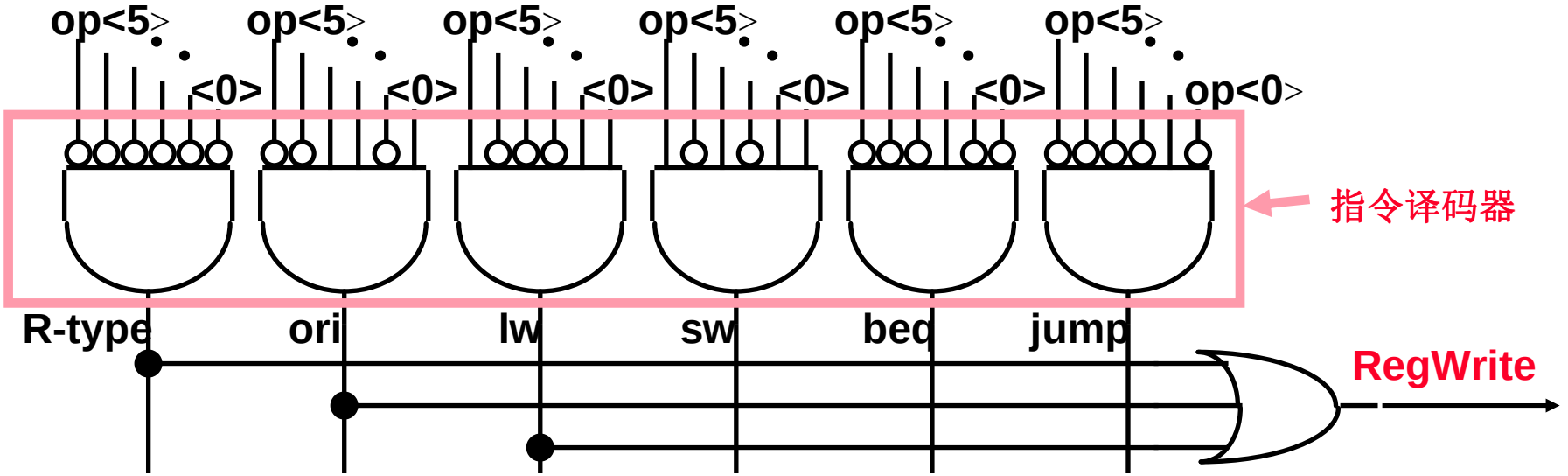
op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegWrite	1	1	1	0	0	0

- RegWrite = R-type + ori + lw

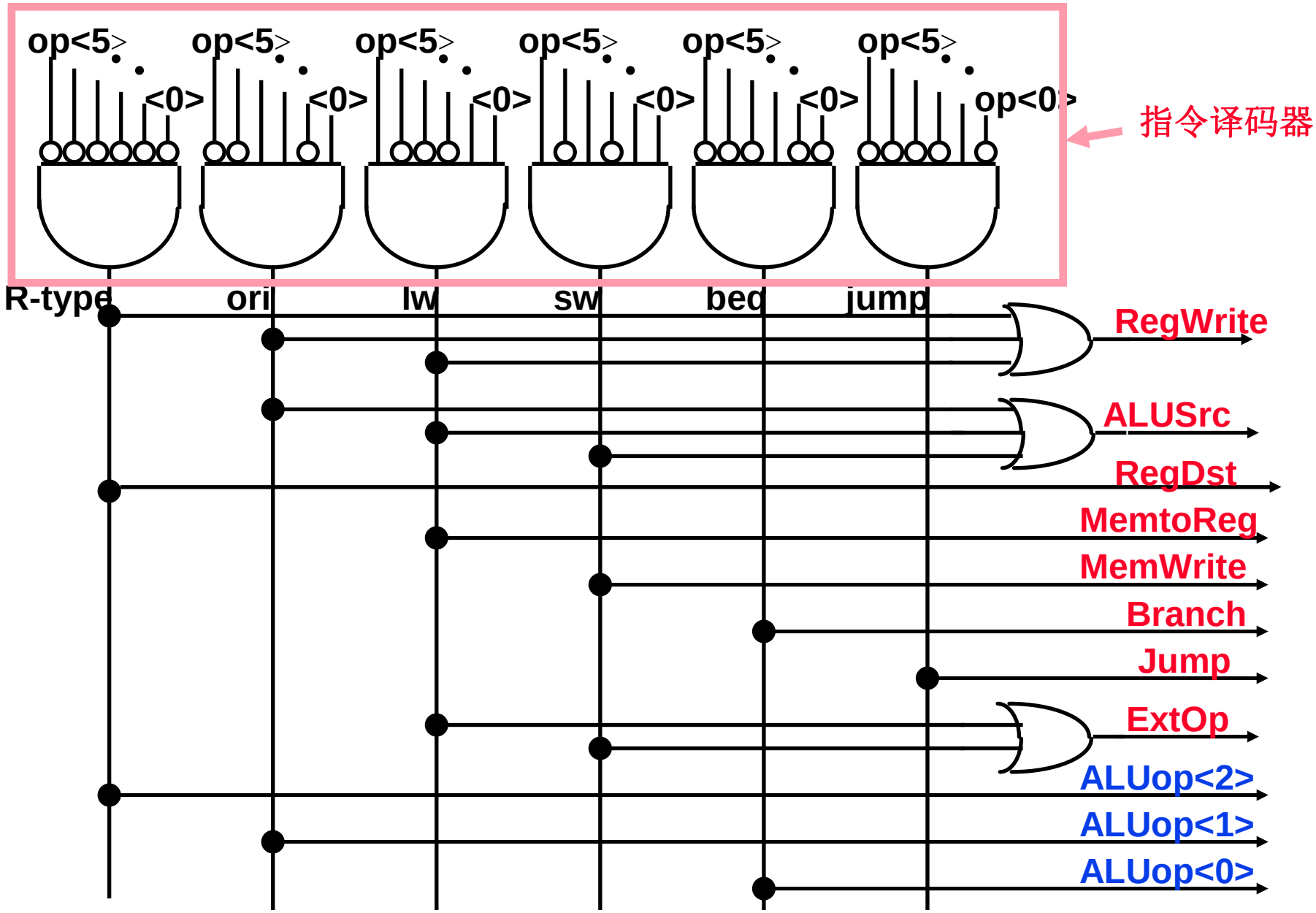
  = !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0> (R-type)

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

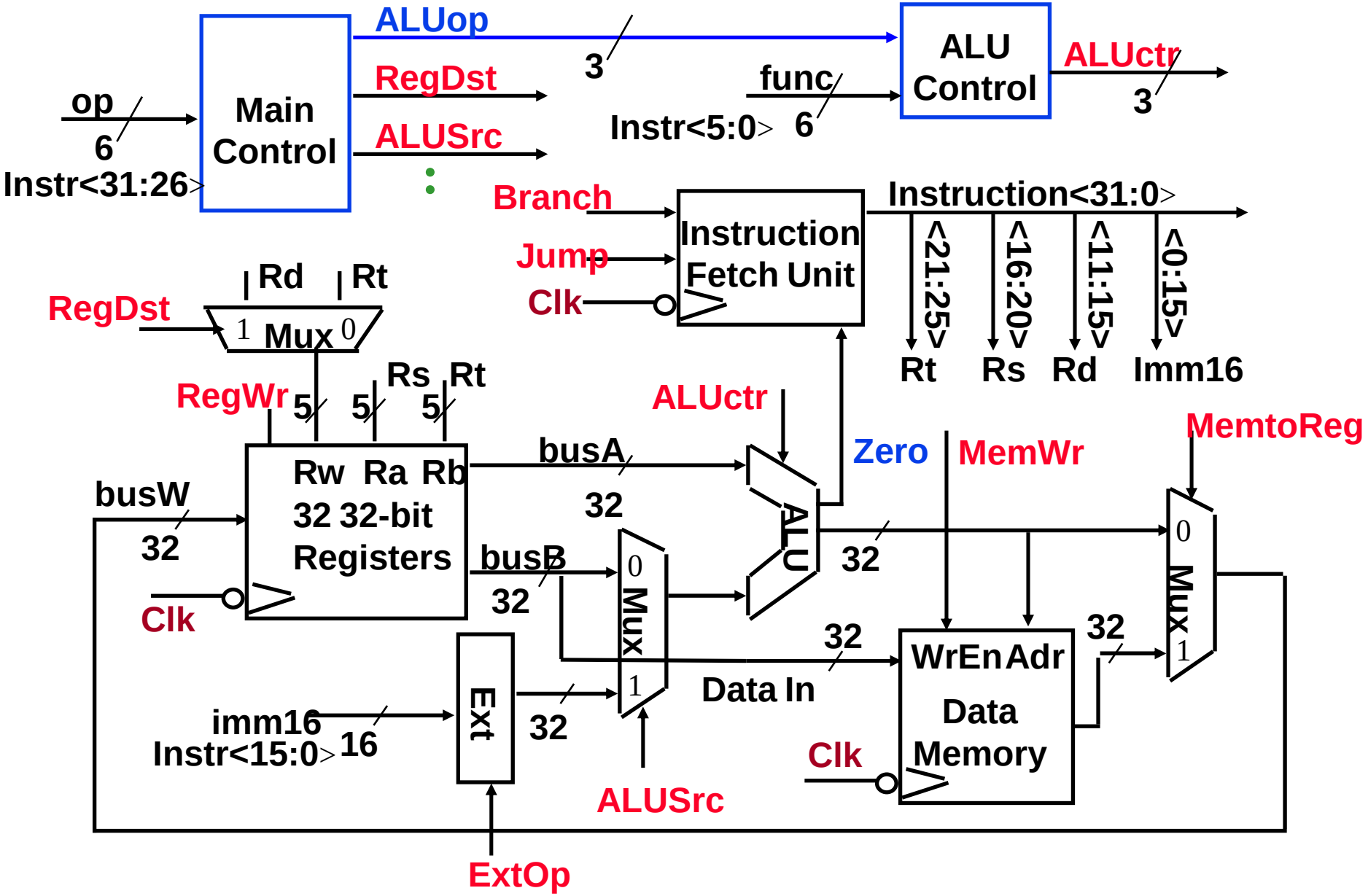
  + op<5> & !op<4> & !op<3> & !op<2> & op<1> & op<0> (lw)



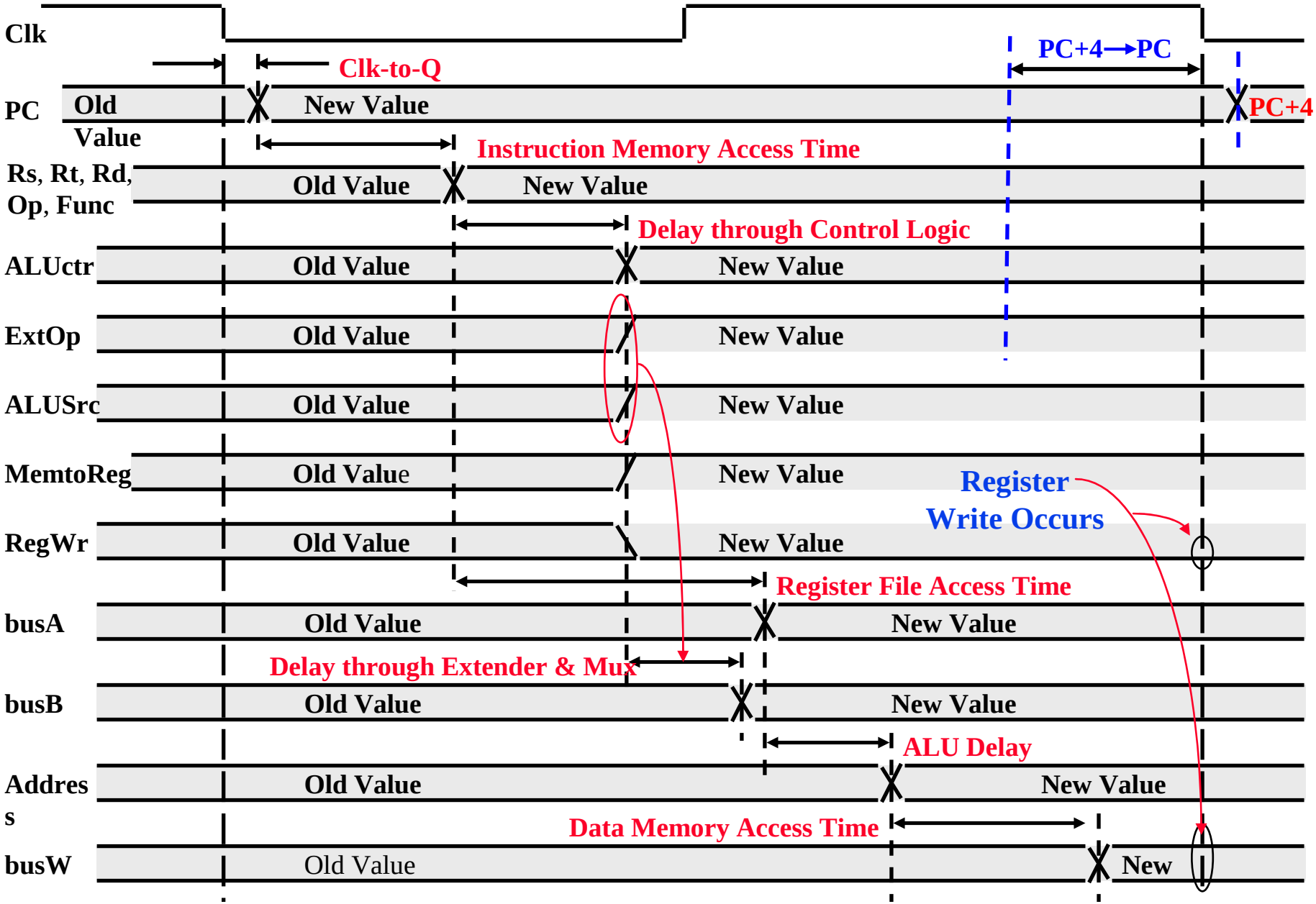
Main Control的PLA实现



执行前述7条指令的完整的单周期处理器



lw指令的执行时间最长，它所花时间作为时钟周期



单周期计算机的性能

- 单周期处理器的CPI为多少？ **CPI=1!**
  - 其他条件一定的情况下，CPI越小，则性能越好！
  - CPI=1，不是很好吗？
- 单周期处理器的性能会不会很好？为什么？
  - 计算机的性能除CPI外，还取决于时钟周期的宽度
  - 单周期处理器的时钟宽度为最复杂指令的执行时间
  - 很多指令可以在更短的时间内完成

单周期计算机的性能

假设在单周期处理器中，各主要功能单元的操作时间为：

- 存储单元：200ps
- ALU和加法器：100ps
- 寄存器堆（读/写）：50ps

假设MUX、控制单元、PC、扩展器和传输线路都没有延迟，则下面实现方式中，哪个更快？快多少？

（1）每条指令在一个固定长度的时钟周期内完成

（2）每条指令在一个时钟周期内完成，但时钟周期仅为指令所需，是可变的（实际不可行，只是为了比较）

假设有25%取数、10%存数、45%ALU、15%分支、5%跳转指令

单周期计算机的性能

解：CPU执行时间=指令条数 x CPI x 时钟周期=指令条数 x 时钟周期

两种方案的指令条数都一样，CPI都为1，所以只要比较时钟周期宽度即可。

Instruction class	Functional units used by the instruction class				
R-type	Instruction fetch	Register access	ALU	Register access	
Load word	Instruction fetch	Register access	ALU	Memory access	Register access
Store word	Instruction fetch	Register access	ALU	Memory access	
Branch	Instruction fetch	Register access	ALU		
Jump	Instruction fetch				

各指令类型要求的时间长度为：

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	50	100	0	50	400 ps
Load word	200	50	100	200	50	600 ps
Store word	200	50	100	200		550 ps
Branch	200	50	100	0		350 ps
Jump	200					200 ps

单周期计算机的性能

对于方式（1），时钟周期由最长指令来决定，应该是load指令，为600ps

对于方式（2），时钟周期取各条指令所需时间，时钟周期从600ps至200ps

根据各类指令的频度，计算出平均时钟周期长度为：

CPU时钟周期=600x25%+550x10%+400x45%+350x15%+200x5%=447.5ps

$$\text{CPU性能比}=\frac{\text{方式(1)的CPU执行时间}}{\text{方式(2)的CPU执行时间}}=\frac{\text{方式(1)的CPU 时钟周期}}{\text{方式(2)的CPU 时钟周期}}=\frac{600}{447.5}=1.34$$

由此可见，可变时钟周期的性能是定长周期的1.34倍！

但是，对每类指令采用可变长时钟周期实现非常困难，而且所带来的额外开销会很大，不合算！

早期的小指令集计算机用过单周期实现技术，但现代计算机都不采用。

下一讲介绍多周期数据通路和控制器，其特点是：

时钟周期固定、时钟周期数可变