计算机组织与系统结构

设计单周期数据通路

Designing a Single Cycle Datapath

第七讲

程旭

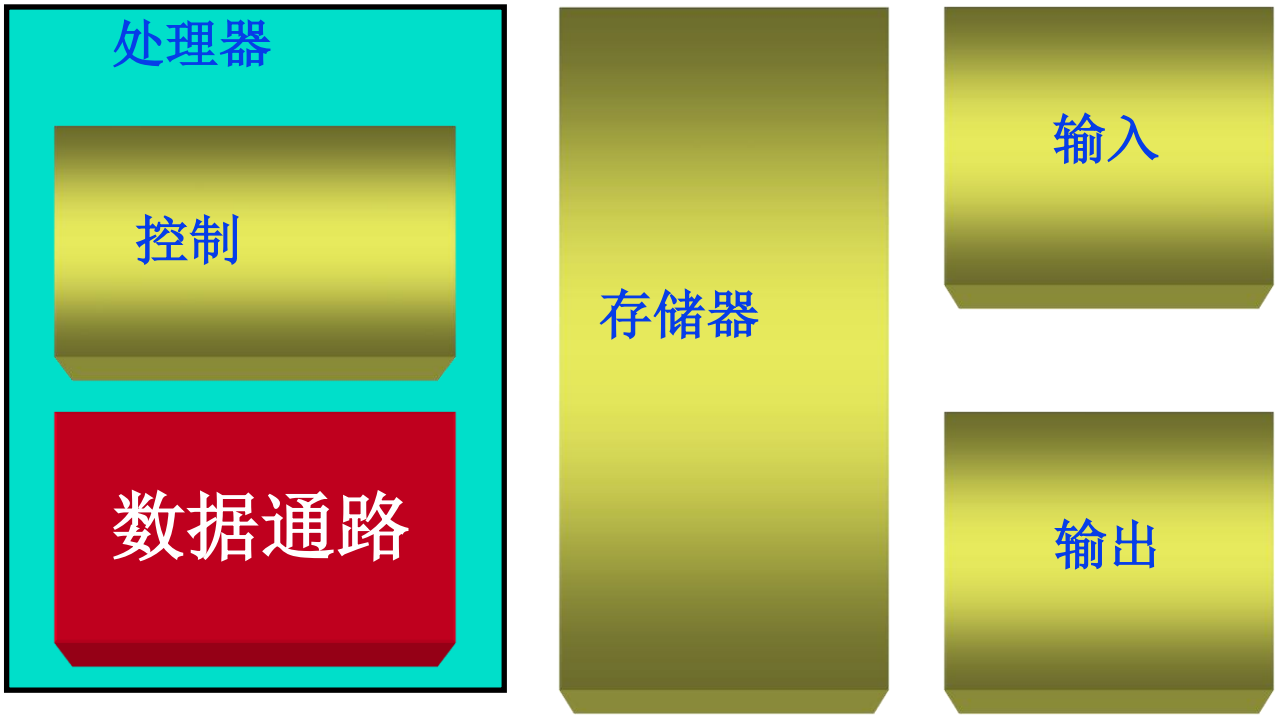
2020.11.26

本讲提纲

- 介绍
- 设计处理器的步骤
- 数据通路、寄存器-寄存器操作的定时
- 立即数逻辑操作的数据通路
- 装入和存储操作的数据通路
- 转移和跳转操作的数据通路

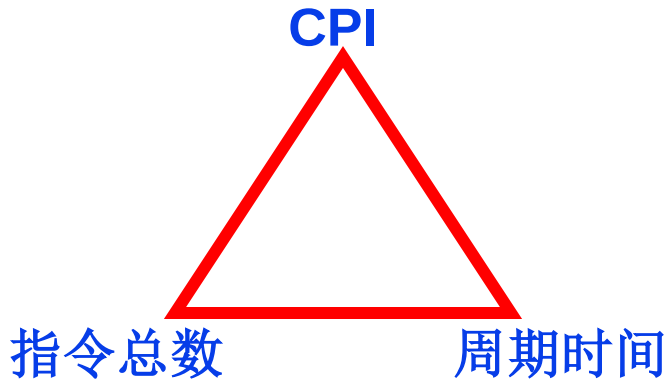
教学目标：已经掌握的内容

- 计算机的五个基本部件



- 本讲主题：数据通路设计

处理器性能



- 计算机的性能决定于：
  - 指令总数
  - 时钟周期时间
  - 每条指令的时钟周期数目
- 处理器设计（数据通路和控制）将决定：
  - 时钟周期时间
  - 每条指令的时钟周期数目
- 单周期处理器：
  - 优点：每条指令一个时钟周期
  - 缺点：时钟周期时间太长

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

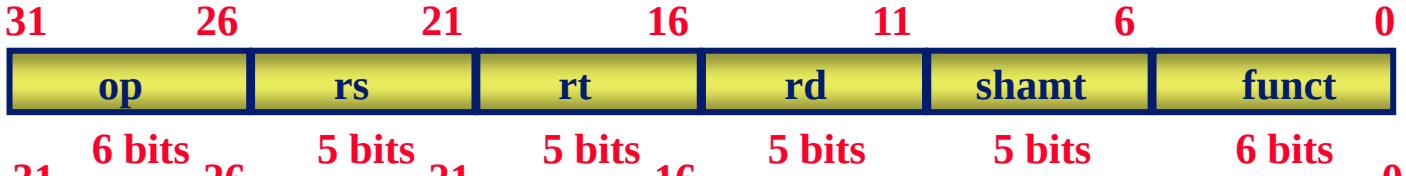
4. 分析每条指令的实现，以确定如何设置影响寄存器传输的控制点

5. 装配 控制逻辑

MIPS指令格式

- 所有的 MIPS 指令都是32位长。具有如下三种格式：

  - R型



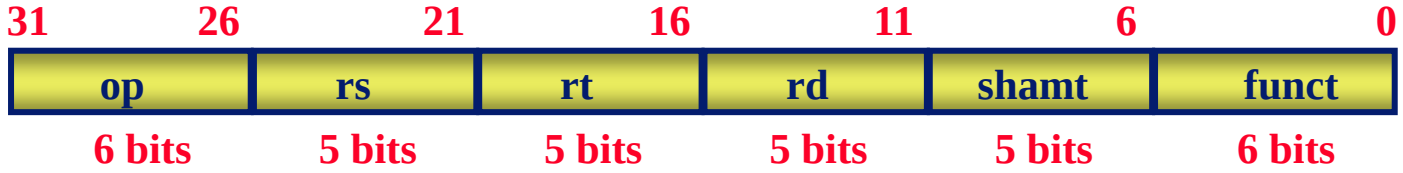
  - I型

  - J型

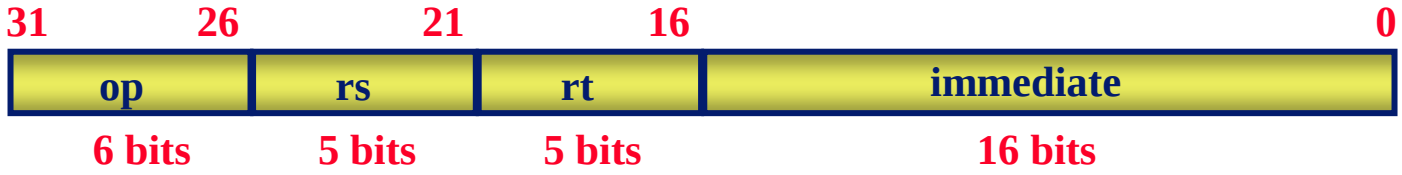
- 不同的场位为：
  - **op:** 指令的操作
  - **rs, rt, rd:** 源和目的寄存器描述符
  - **shamt:** 移位量
  - **funct:** 选择Op场位指定的不同操作
  - **address / immediate:** 地址偏移量或者立即数数值
  - **target address:** 跳转指令的目标地址

本讲涉及的MIPS指令系统子集

- 加法和减法
  - add rd, rs, rt
  - sub rd, rs, rt



- 或立即数:
  - ori rt, rs, imm16



- 装入和存储
  - lw rt, rs, imm16
  - sw rt, rs, imm16

- 转移:
  - beq rs, rt, imm16

- 跳转:
  - j target

逻辑寄存器传输

- 寄存器传输语言（RTL） 描述每条指令的意图
- 所有的指令都以取指开始

op | rs | rt | rd | shamt | funct = MEM[PC]

op | rs | rt | Imm16 = MEM[PC]

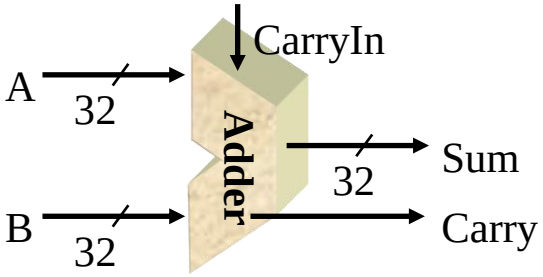
inst	Register Transfers	
ADDU	R[rd] ⇐ R[rs] + R[rt];	PC ⇐ PC + 4
SUBU	R[rd] ⇐ R[rs] - R[rt];	PC ⇐ PC + 4
ORi	R[rt] ⇐ R[rs] ∪ zero_ext(Imm16);	PC ⇐ PC + 4
LOAD	R[rt] ⇐ MEM[R[rs] + sign_ext(Imm16)];	PC ⇐ PC + 4
STORE	MEM[R[rs] + sign_ext(Imm16)] ⇐ R[rt];	PC ⇐ PC + 4
BEQ	if (R[rs] == R[rt]) then PC ⇐ PC + sign_ext(Imm16)] || 00 else PC ⇐ PC + 4	

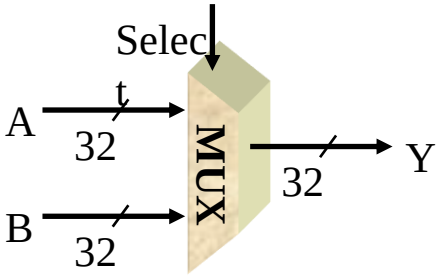
第一步：指令系统需求分析

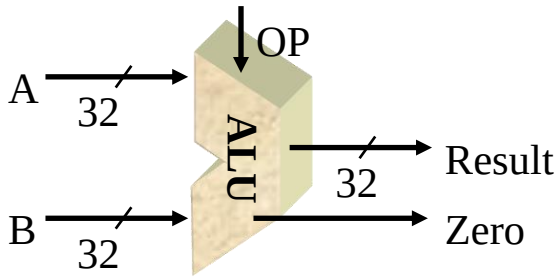
- 存储器
  - 指令 和 数据
- 通用寄存器 (32 x 32)
  - 读 RS
  - 读 RT
  - 写 RT 或 RD
- 程序计数器（PC）
- 扩展器（Extender）
- Add 和 Sub 寄存器或扩展后的立即数
- PC加上 4或扩展后的立即数

第二步：数据通路的部件

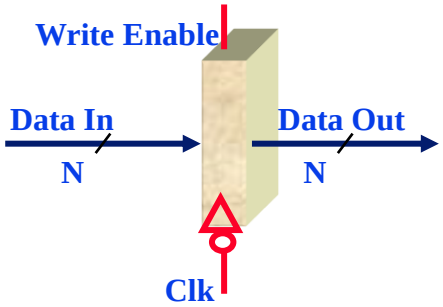
- 组合元件

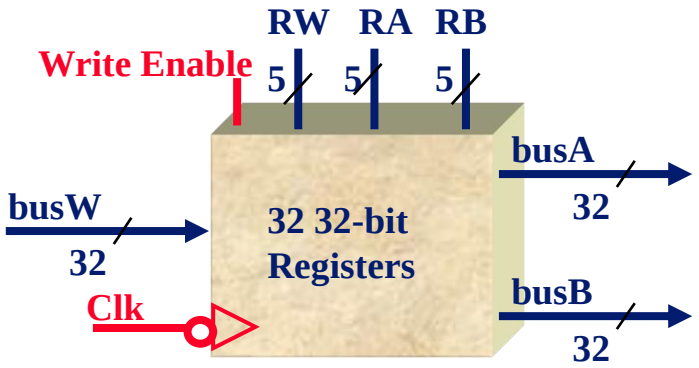


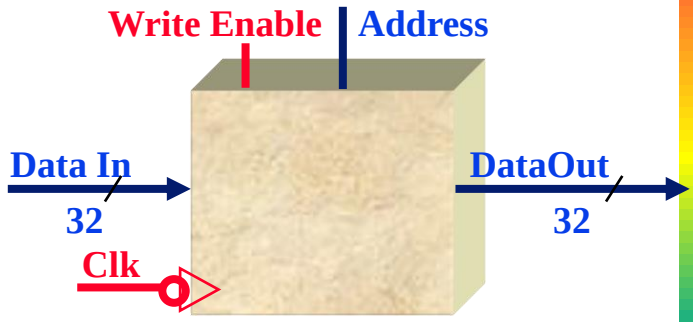




- 存储元件
  - 时钟同步方法

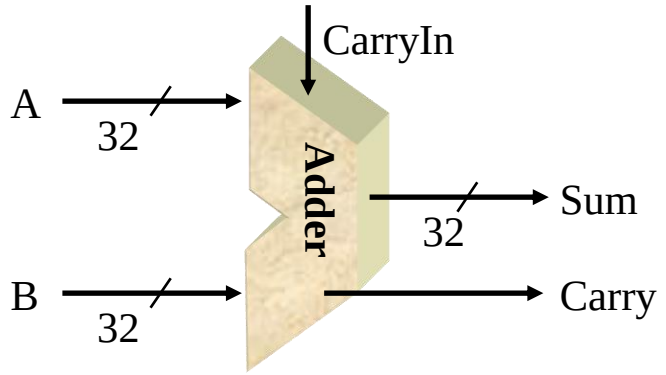




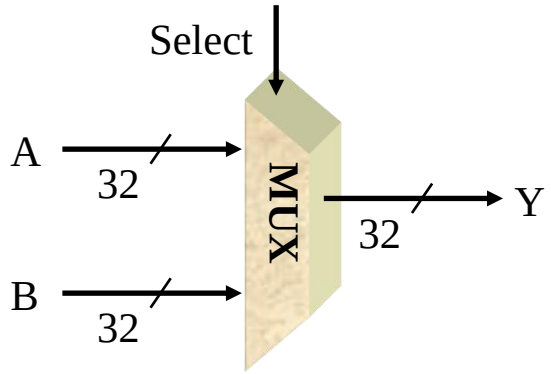


组合逻辑部件

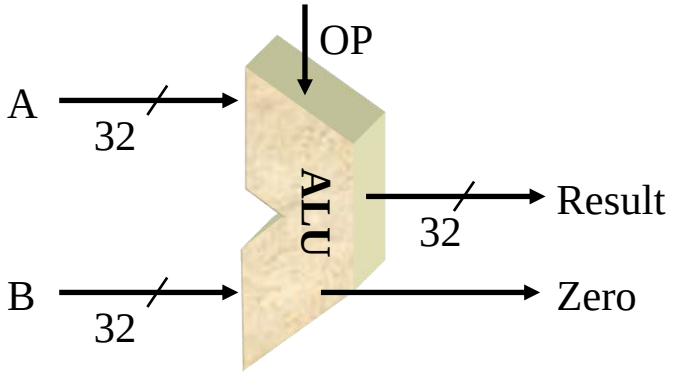
- 加法器



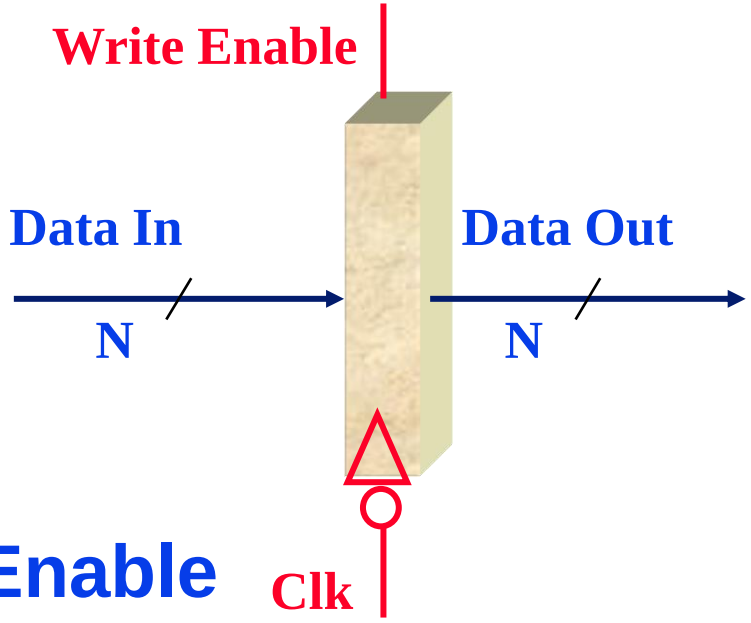
- 多路选择器



- **ALU**



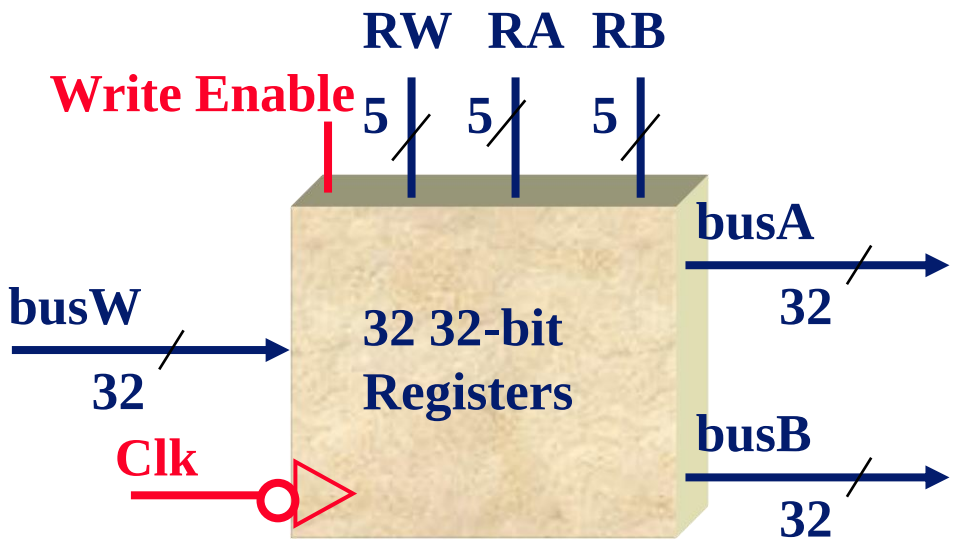
存储部件：寄存器



- 寄存器
  - 类似于D触发器，只是
    - N位输入和输出
    - 写操作使能输入（Write Enable input）
  - 写使能（Write Enable:）
    - 0: 数据输出将不改变
    - 1: 数据输出将变成数据输入的值

存储部件：寄存器堆

- 寄存器堆包含 32个寄存器：
  - 两条32位输出总线：
    busA 和 busB
  - 一条 32位输入总线：busW

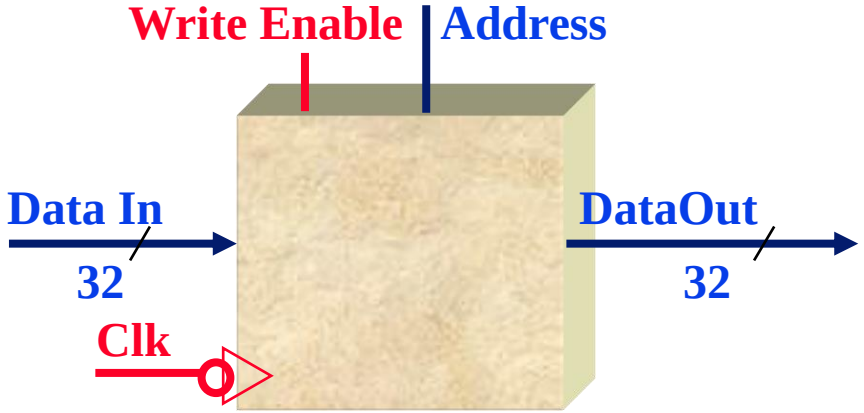


- 选择寄存器：
  - RA 选择将数据放到 busA上的寄存器
  - RB 选择将数据放到 busB上的寄存器
  - RW 选择 在写使能为1时，由 busW 将写的寄存器

- 时钟输入 (CLK)
  - 只有在写操作中，CLK输入才有作用
  - 在读操作中，寄存器的行为与组合逻辑电路一样：
    - RA 或 RB 有效 => 在访问时间之后，busA 或 busB有效。

存储部件：理想化的存储器

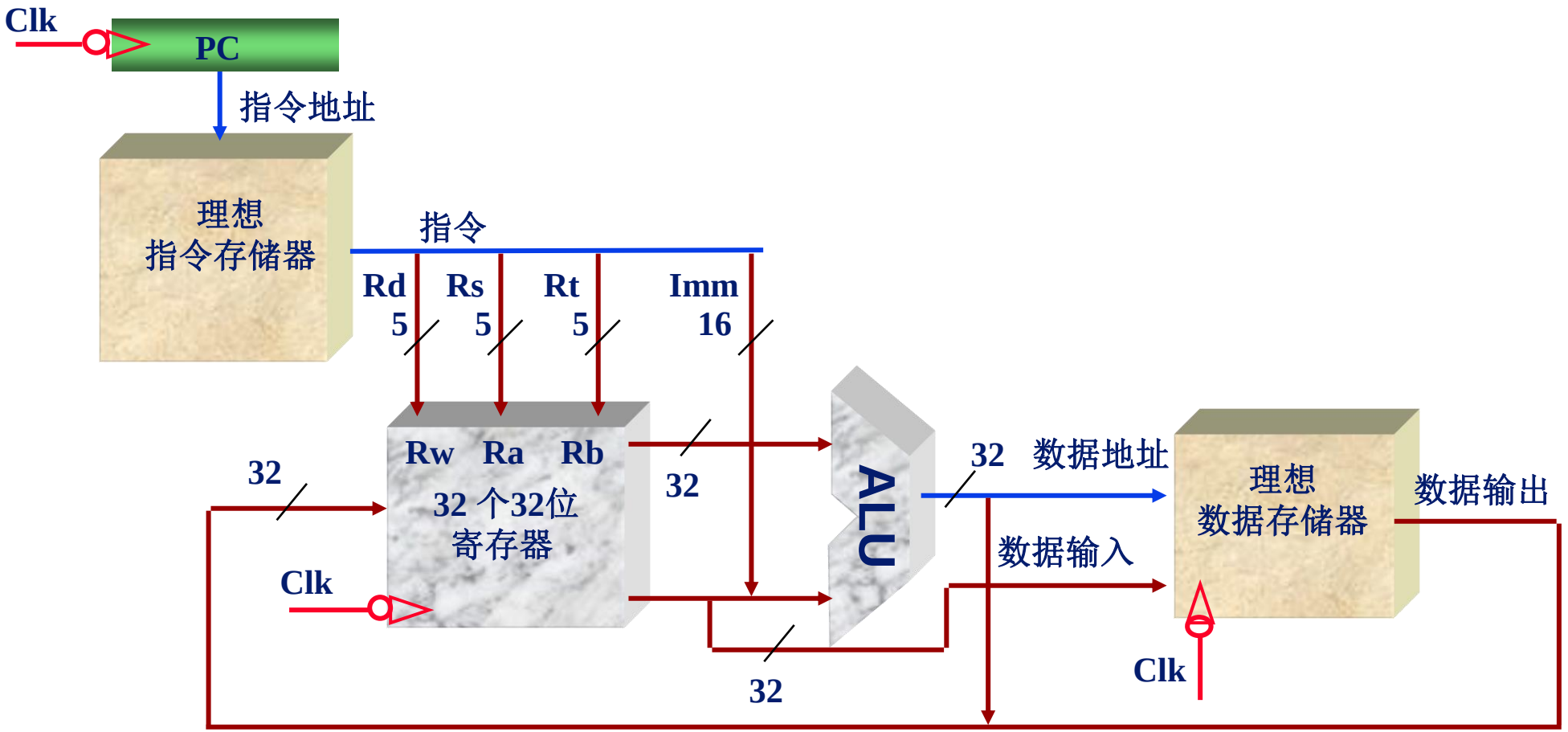
- 理想化的存储器
  - 单输入总线: 数据输入（Data In）
  - 单输出总线: 数据输出（Data Out）



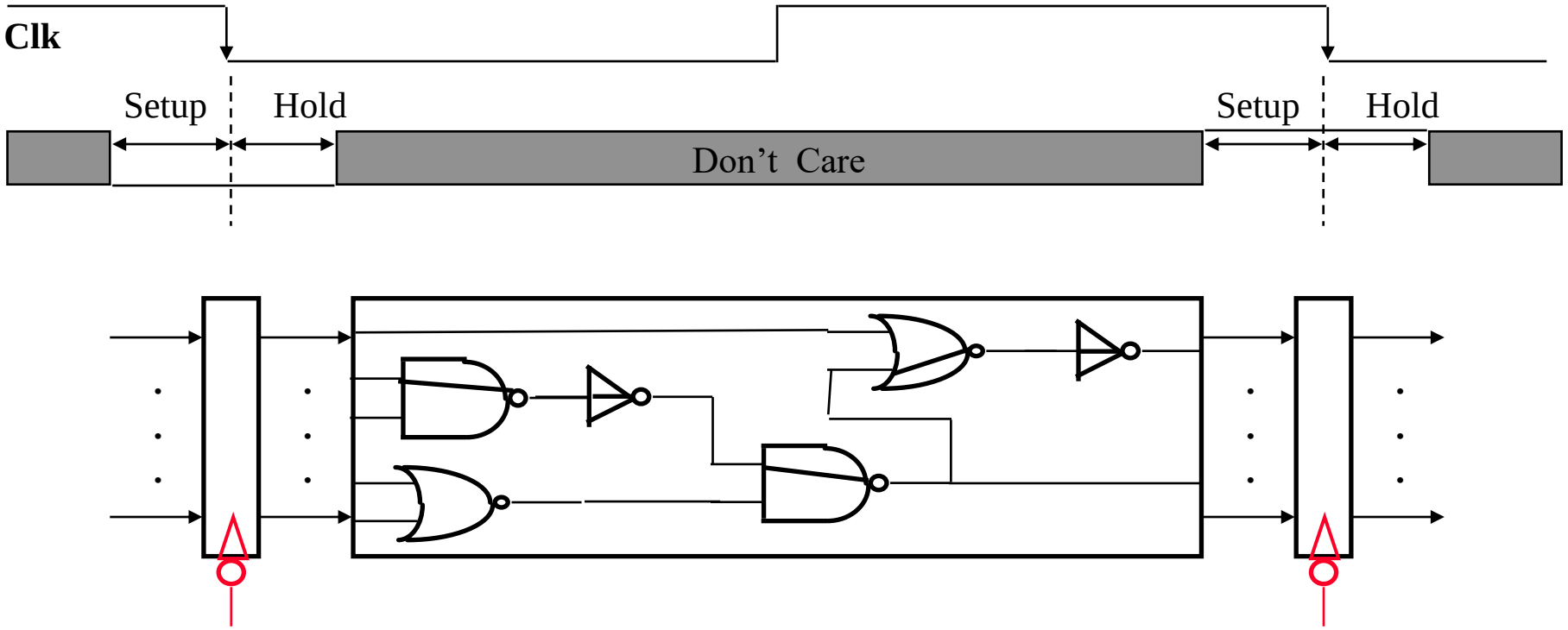
- 选择存储字:
  - 地址（Address）选择的存储字被放在 Data Out上
  - Write Enable = 1: Data In总线上的数据被写入地址选择的存储单元中

- 时钟输入 (CLK)
  - 只有在写操作中，CLK输入才有作用
  - 在读操作中，寄存器的行为与组合逻辑电路一样:
    - RA 或 RB 有效 => 在访问时间之后，busA 或 busB有效。

实现抽象


Clk
PC
指令地址
理想
指令存储器
指令
Rd
5
Rs
5
Rt
5
Imm
16
Rw Ra Rb
32 个32位
寄存器
32
32
Clk
32
ALU
32 数据地址
数据输入
理想
数据存储器
数据输出
Clk

时钟同步方法



- 所有的存储元件都受相同的时钟边沿来驱动
- 周期时间>= CLK-to-Q时间 + 最长延迟路径时间+ 建立时间+ 时钟纽斜
- (CLK-to-Q时间+ 最短延迟路径时间- 时钟纽斜) > 保持时间

寄存器传输语言：加法指令

- add rd, rs, rt
  - mem[PC] 从存储器中取出指令
  - R[rd] ⇐ R[rs] + R[rt] 加法操作
  - PC ⇐ PC + 4 计算下一条指令地址

寄存器传输语言：装入指令

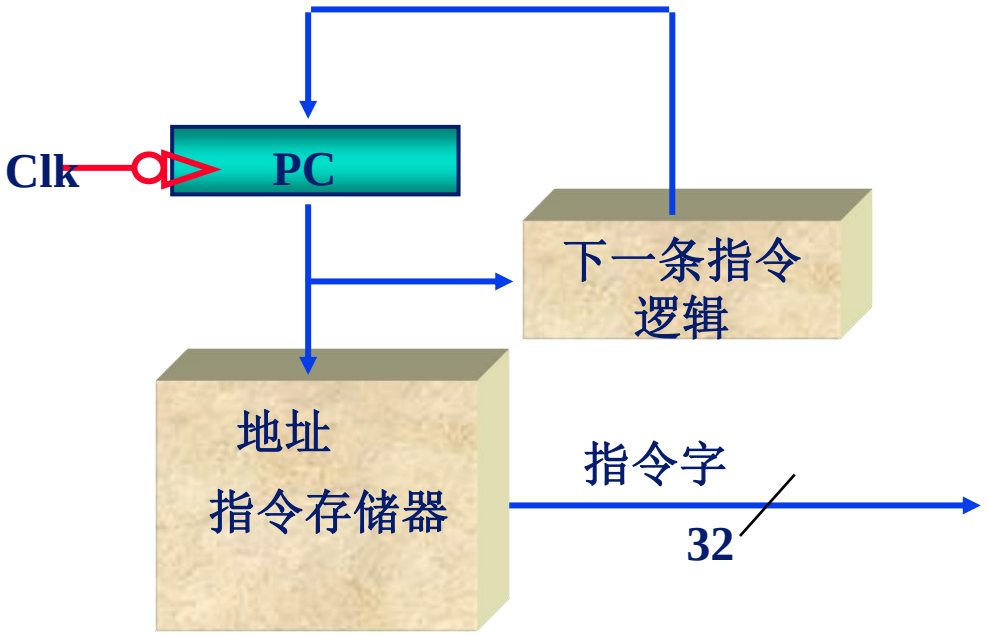
- lw　rt, rs, imm16
  - mem[PC]　　从存储器中取出指令
  - Addr ⇐ R[rs] + SignExt(imm16)　　计算存储器地址
  - R[rt] ⇐ Mem[Addr]　　将数据装入寄存器
  - PC ⇐ PC + 4　　计算下一条指令的地址

第三步 根据需求组装数据通路

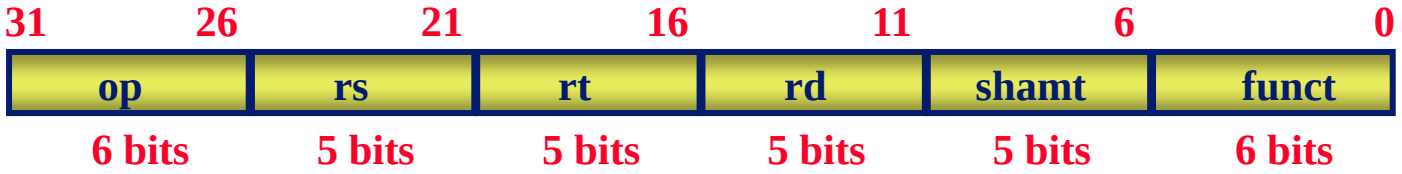
- 寄存器传输 需求分析 和 装配数据通路
- 取指
- 读取操作数 和 执行操作

取指部件（Instruction Fetch Unit）概况

- 基本寄存器传输语言操作
  - 取指: mem[PC]
  - 修改程序计数器:
    - 串行执行代码： PC⇐ PC + 4
    - 转移或跳转： PC ⇐ 其他数值

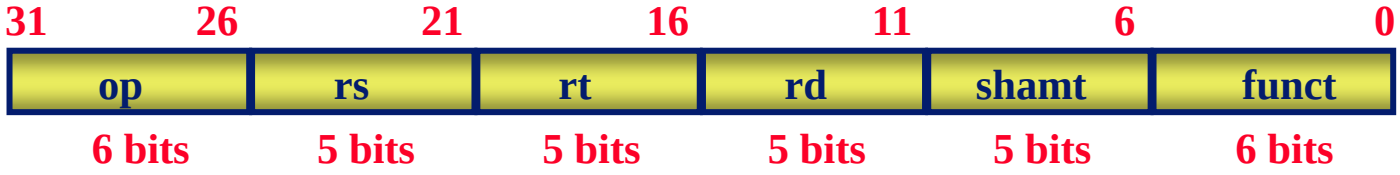


寄存器传输语言： 加法指令



- 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

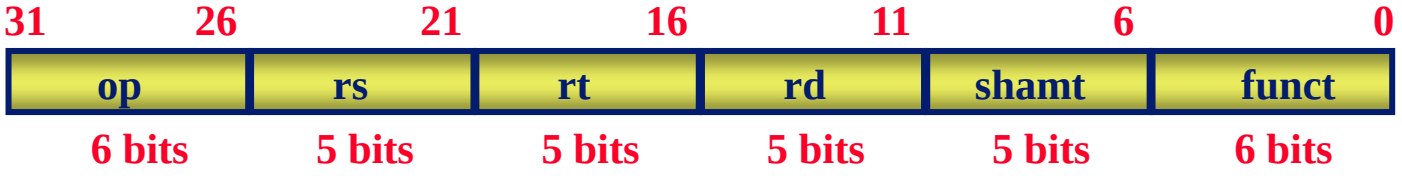
寄存器传输语言： 减法指令

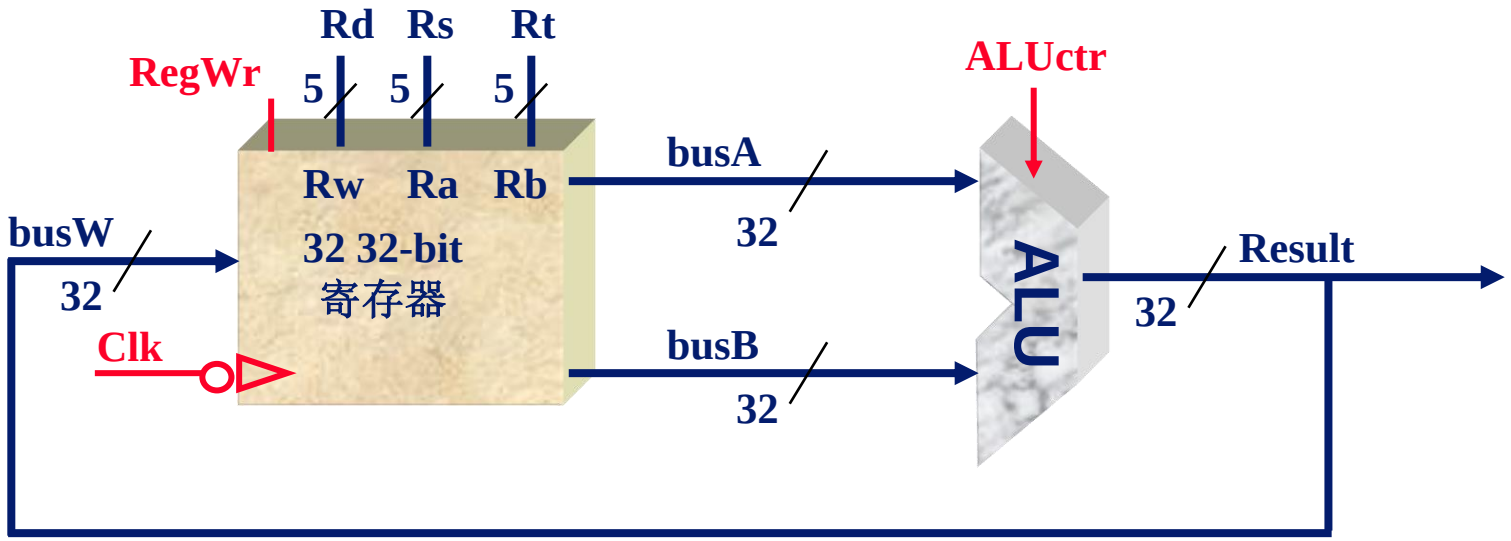


- sub rd, rs, rt
  - mem[PC] 从存储器中读取指令
  - R[rd] ⇐ R[rs] - R[rt] 实际操作
  - PC ⇐ PC + 4 计算下一条指令地址

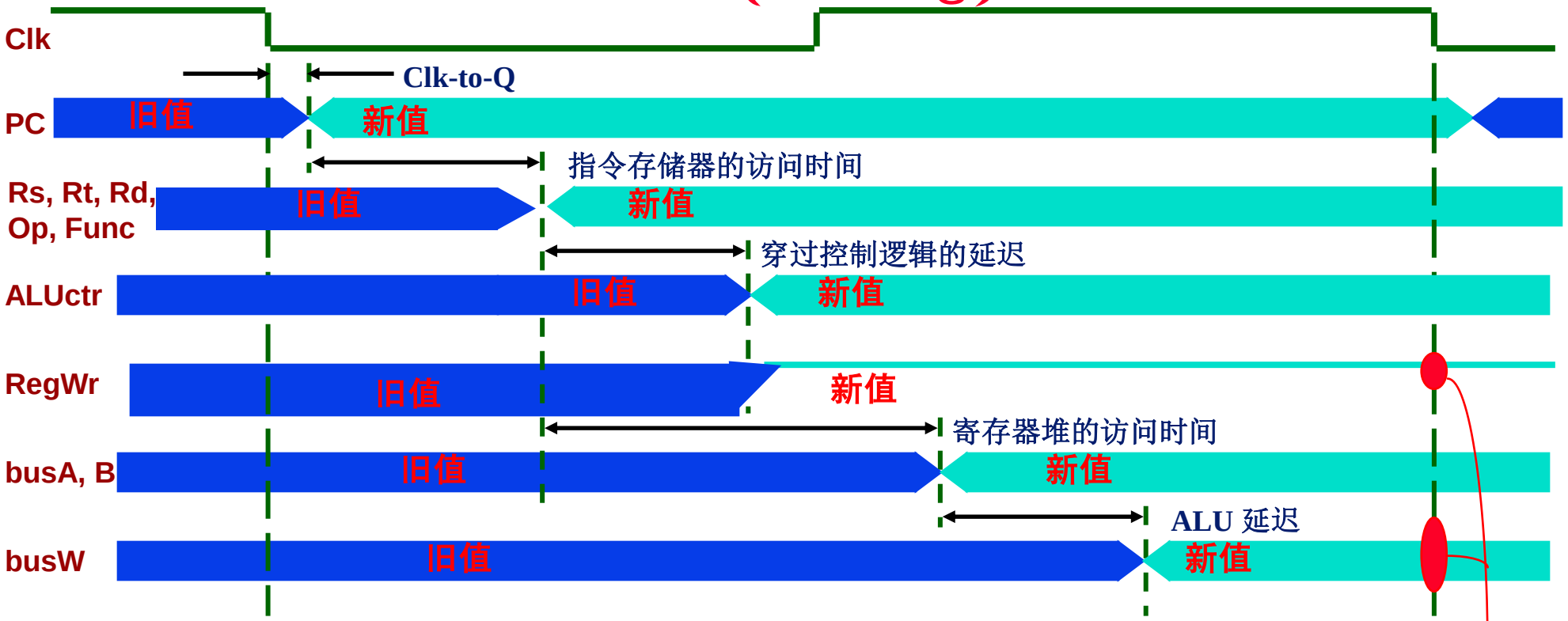
寄存器-寄存器操作的数据通路

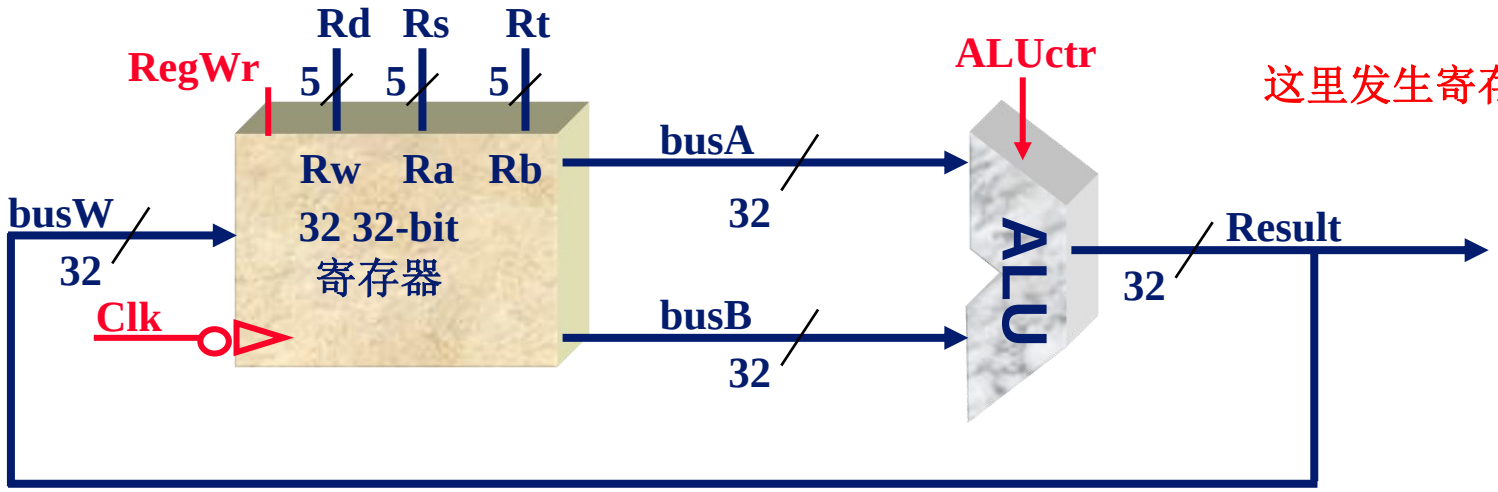
- R[rd] ⇐ R[rs] op R[rt]　　例如: add　rd, rs, rt
  - Ra, Rb和 Rw来自指令的rs, rt和 rd场位
  - ALUctr 和 RegWr: 在对指令进行译码之后的控制逻辑



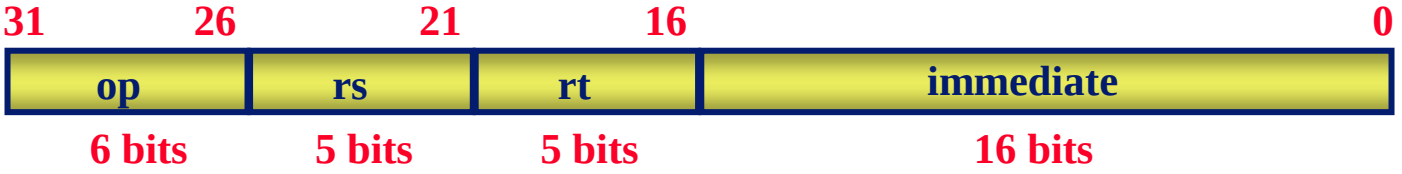


寄存器-寄存器定时 (Timing)



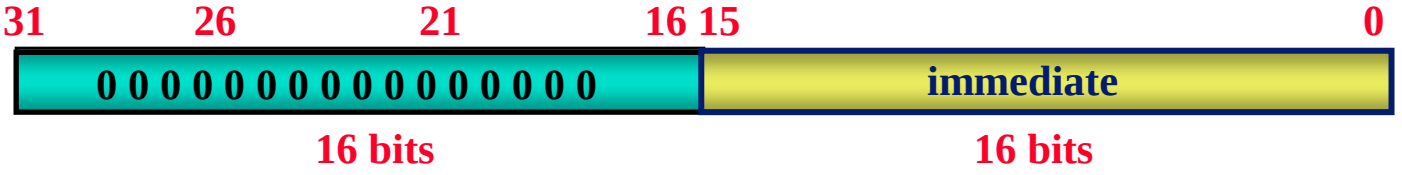


寄存器传输语言： 或立即数指令



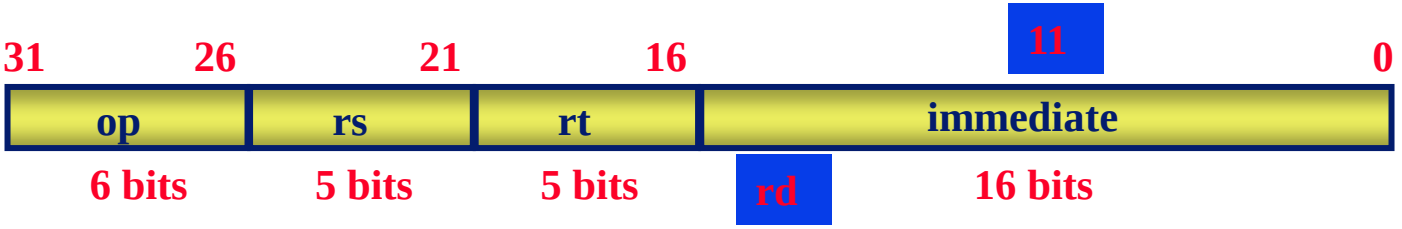
- ori rt, rs, imm16

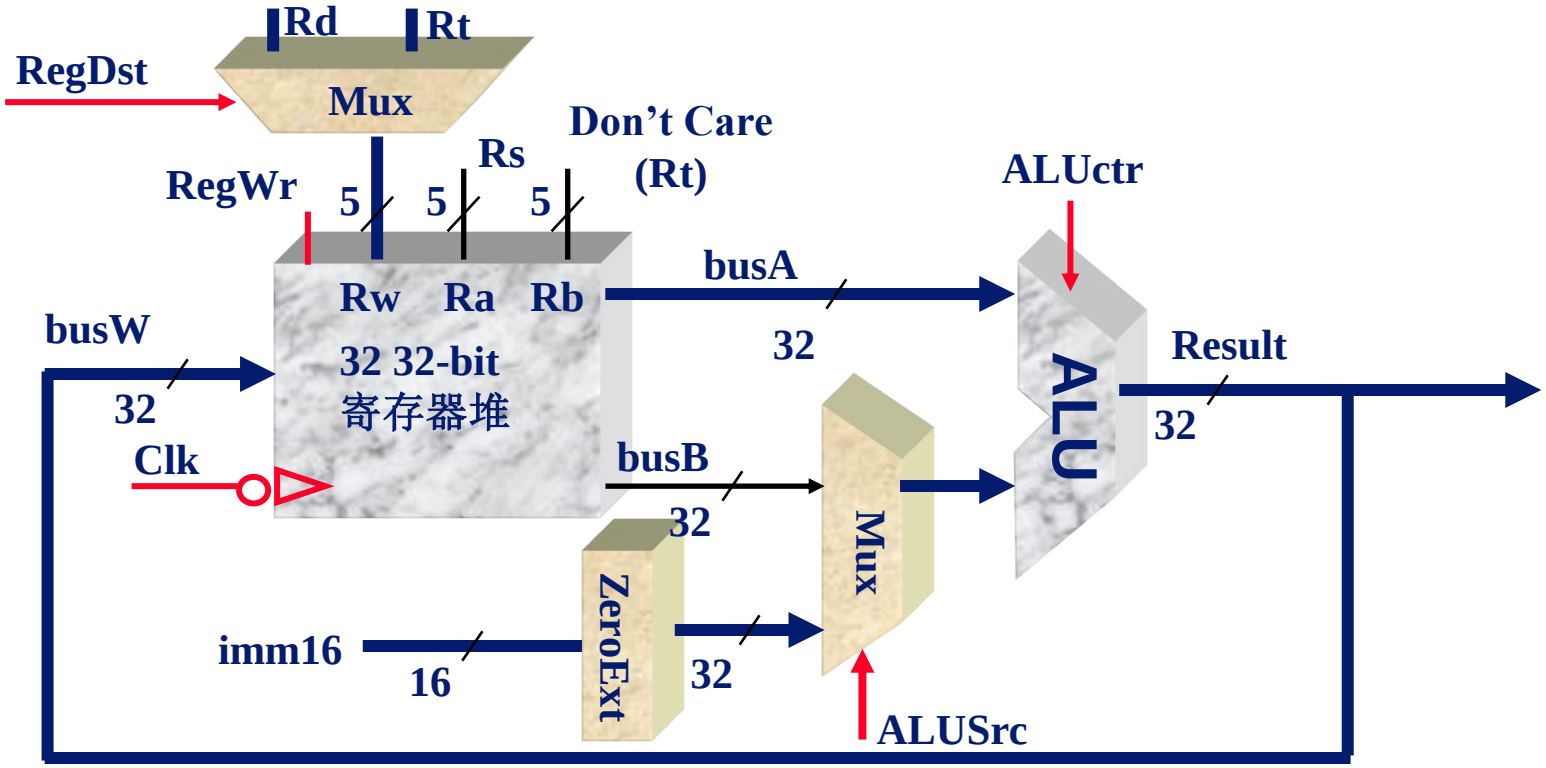
  - mem[PC] 从存储器中读取指令
  - R[rt] ⇐ R[rs] or ZeroExt(imm16)
    或操作
  - PC ⇐ PC + 4 计算下一条指令的地址



有立即数的逻辑操作的数据通路

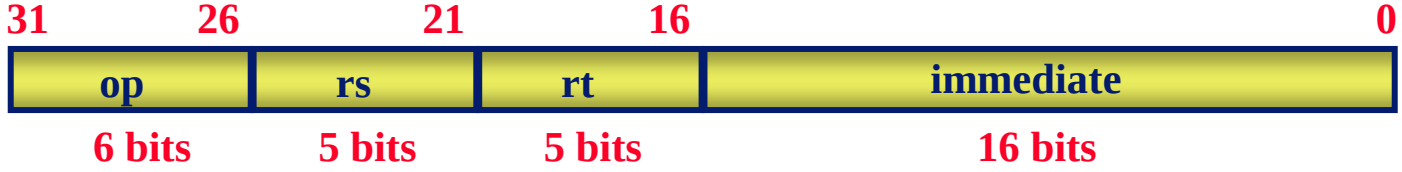
- R[rt] ⇐ R[rs] op ZeroExt[imm16]]　示例: ori　rt, rs, imm16





寄存器传输语言： 装入指令

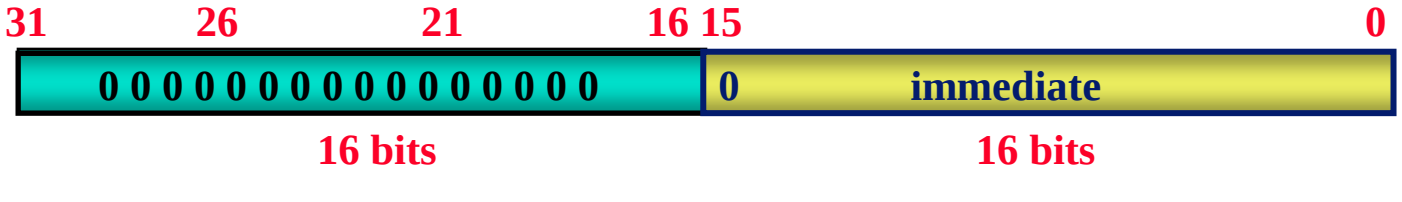
- lw rt, rs, imm16

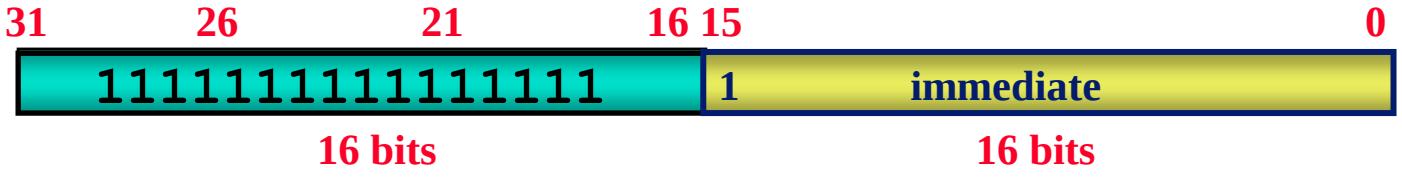


- mem[PC] 从存储器中读取指令
- Addr ⇐ R[rs] + SignExt(imm16)

  计算存储器的地址

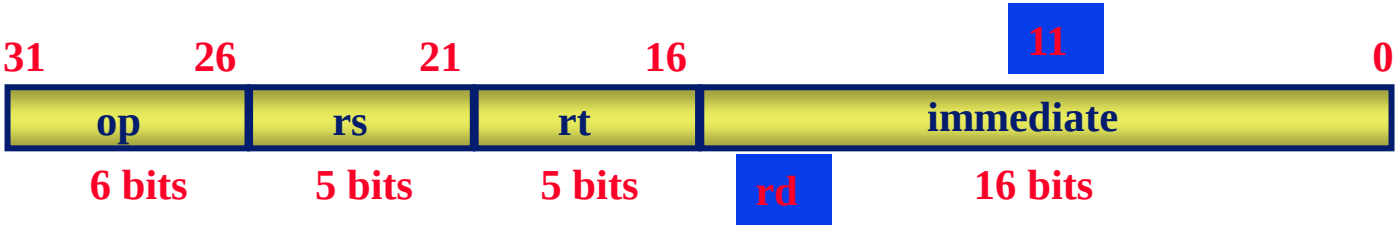
  R[rt] ⇐ Mem[Addr] 从寄存器中装入数据
- PC ⇐ PC + 4 计算下一条指令地址

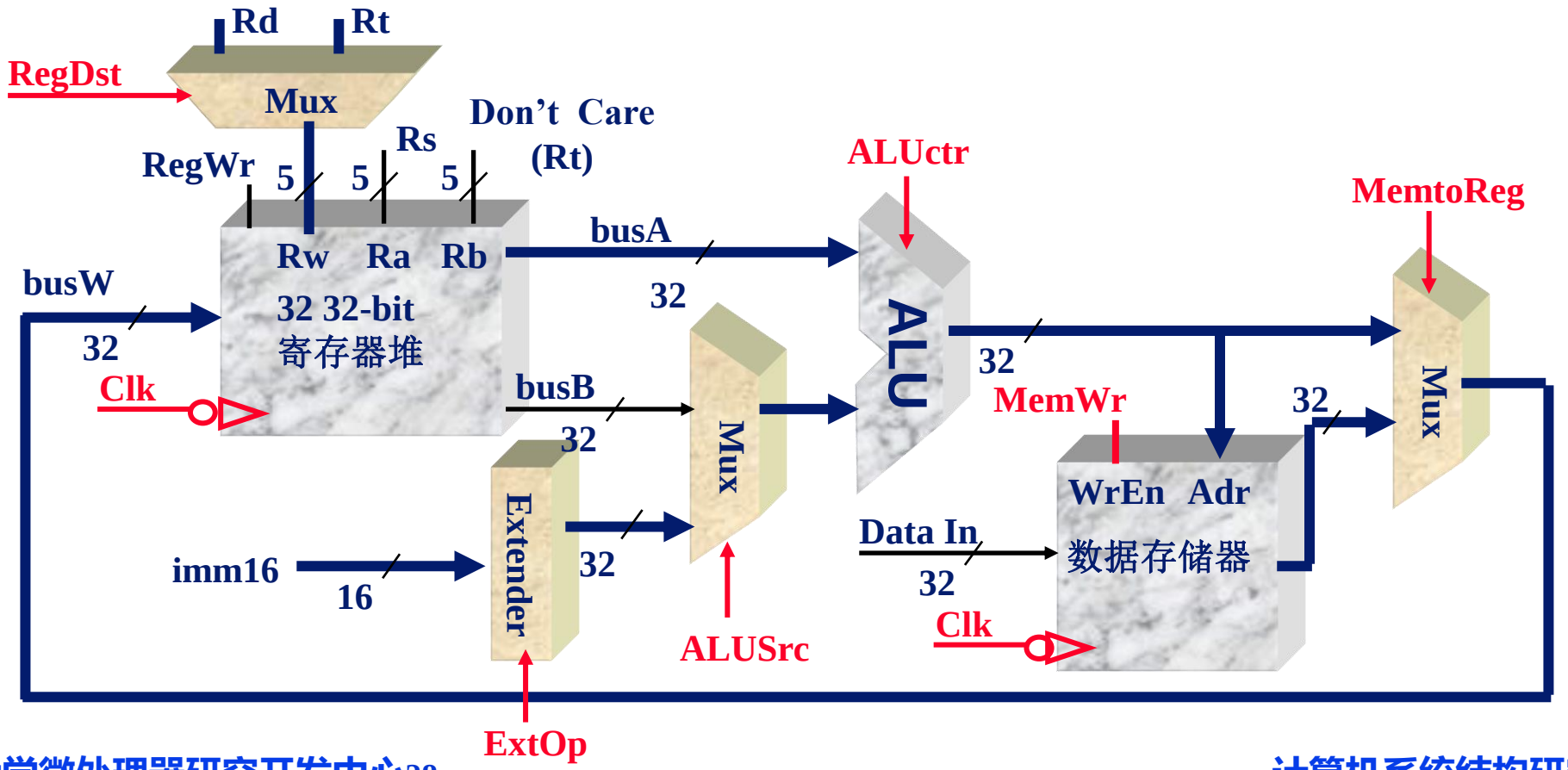




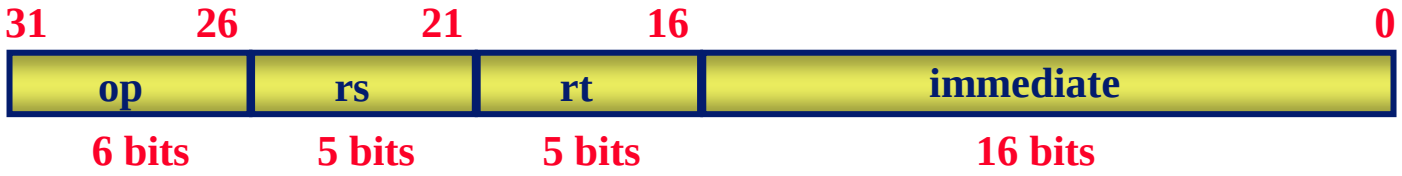
装入操作的数据通路

- R[rt] ⇐ Mem[R[rs] + SignExt[imm16]]　示例: lw　rt, rs, imm16





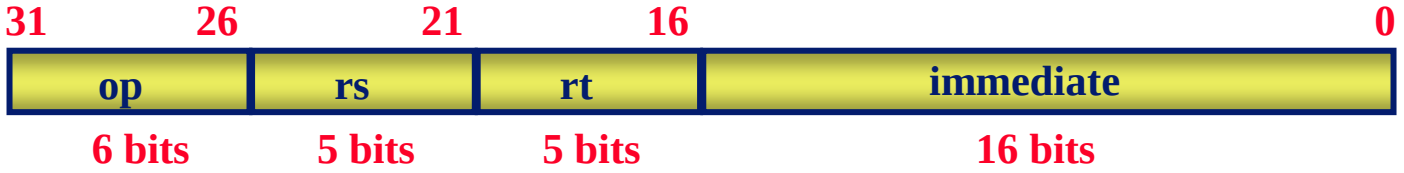
寄存器传输语言： 存储指令

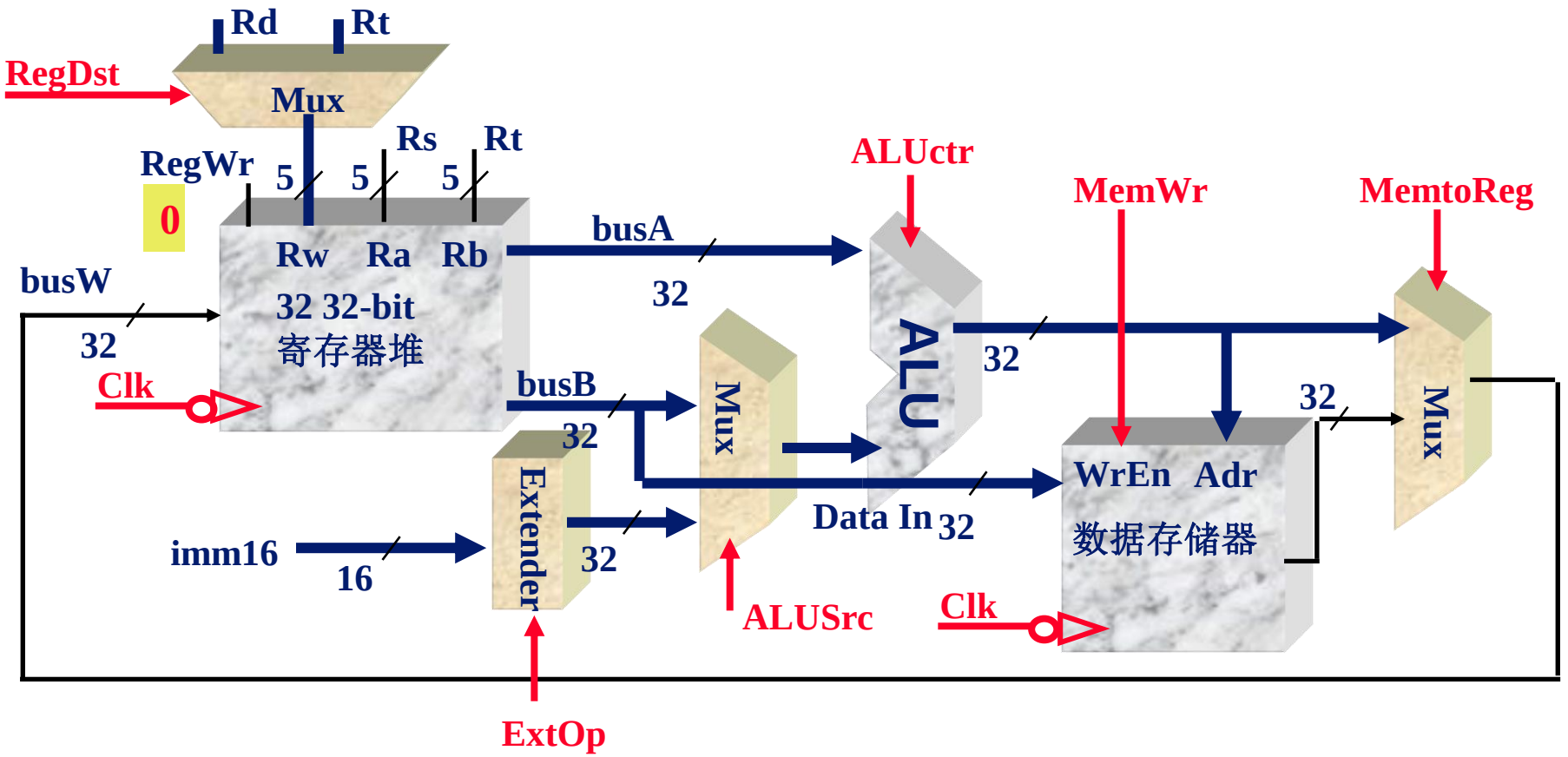


- sw rt, rs, imm16
  - mem[PC] 从存储器中读取指令
  - Addr ⇐ R[rs] + SignExt(imm16) 计算存储器地址
  - Mem[Addr] ⇐ R[rt] 将寄存器的内容存储到存储器中
  - PC ⇐ PC + 4 计算下一条指令地址

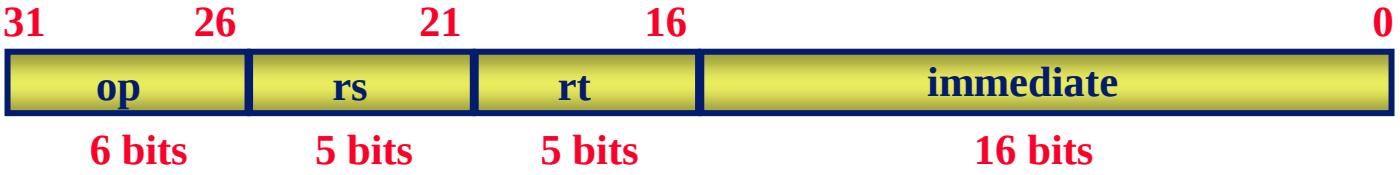
存储指令的数据通路

- Mem[R[rs] + SignExt[imm16] ⇐ R[rt]]　示例: sw　rt, rs, imm16





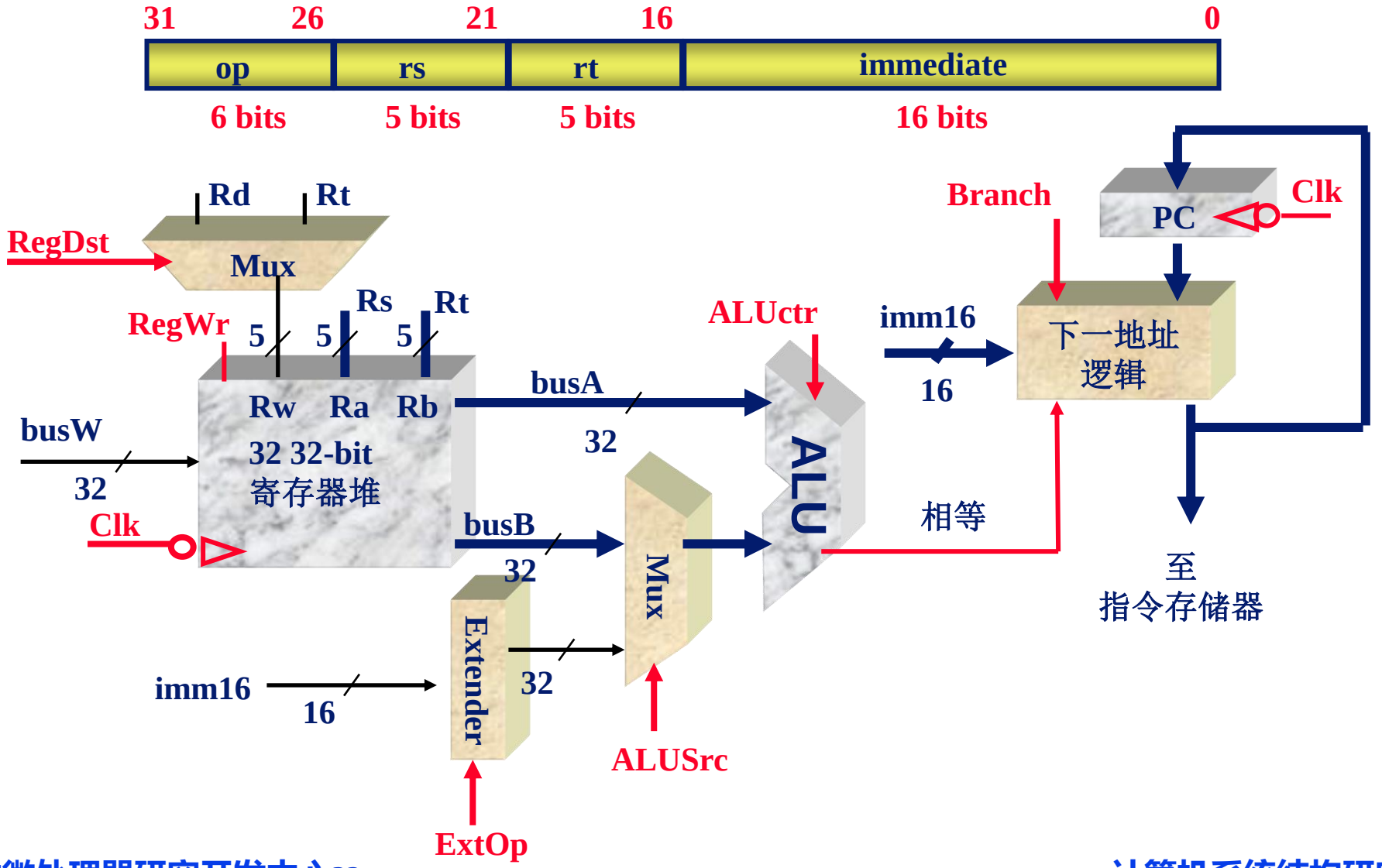
寄存器传输语言： 转移指令



- beq rs, rt, imm16
  - mem[PC] 从存储器中读取指令
  - Cond ⇐ R[rs] - R[rt] 计算转移地址
  - if (COND eq 0) 计算下一条指令地址
    - PC ⇐ PC + 4 + (SignExt(imm16) x 4)
  - else
    - PC ⇐ PC + 4

转移操作的数据通路

- beq rs, rt, imm16　　需要比较 Rs 和 Rt!



下一地址的二进制运算

- 从理论上讲，PC是一个输入到指令存储器的32位的字节地址:
  - 串行操作: PC<31:0> = PC<31:0> + 4
  - 转移操作: PC<31:0> = PC<31:0> + 4 + SignExt[Imm16] × 4

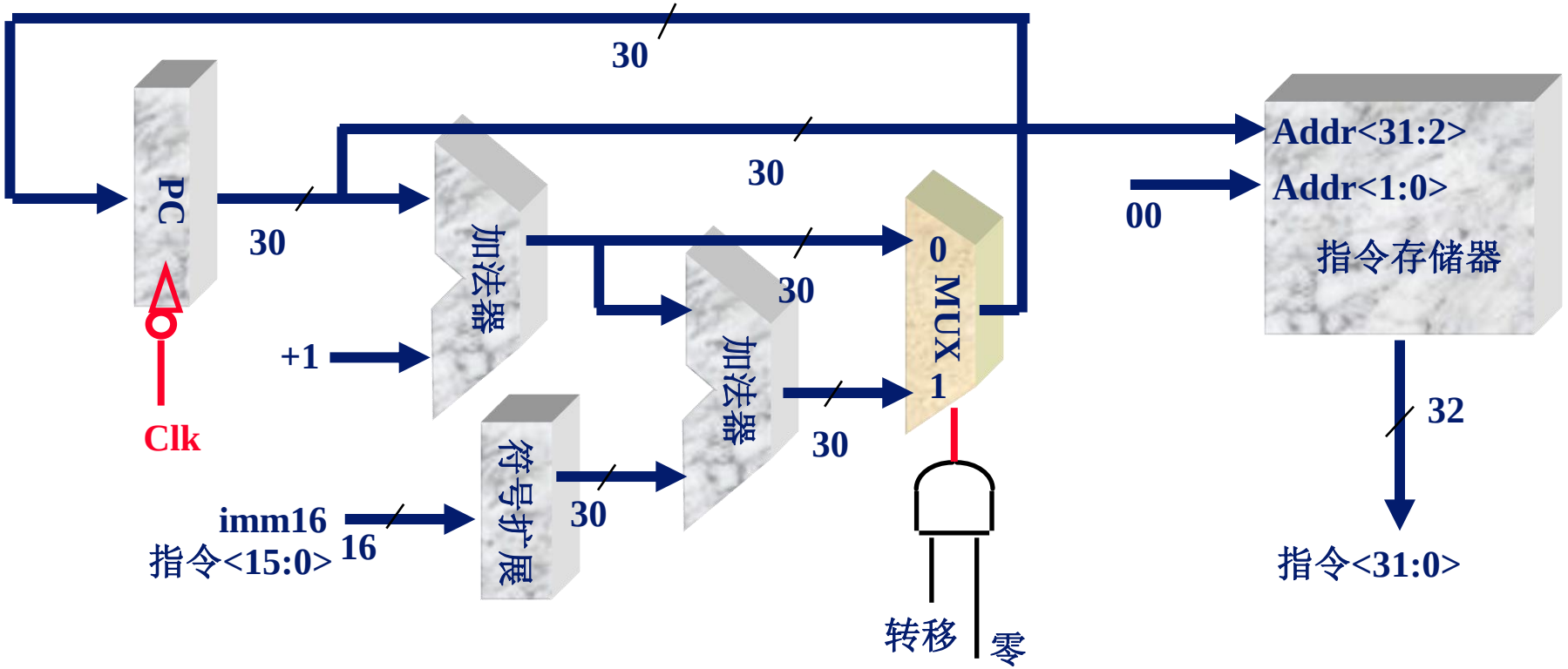
- 总是产生**奇特的数 ??**，这是因为:
  - 32位PC是一个字节地址
  - 并且，所有的指令都是 4字节 (32位) 长

- 也就是说:
  - 32位PC的最小两位(LSB) 总是 0 :-)
  - 因而，没有必要用硬件保存这两位

- 实际上, 可以只使用 30位 PC<31:2>:
  - 串行操作: PC<31:2> = PC<31:2> + 1
  - 转移操作: PC<31:2> = PC<31:2> + 1 + SignExt[Imm16]
  - 其他情况: 指令存储器地址 = PC<31:2> || 00

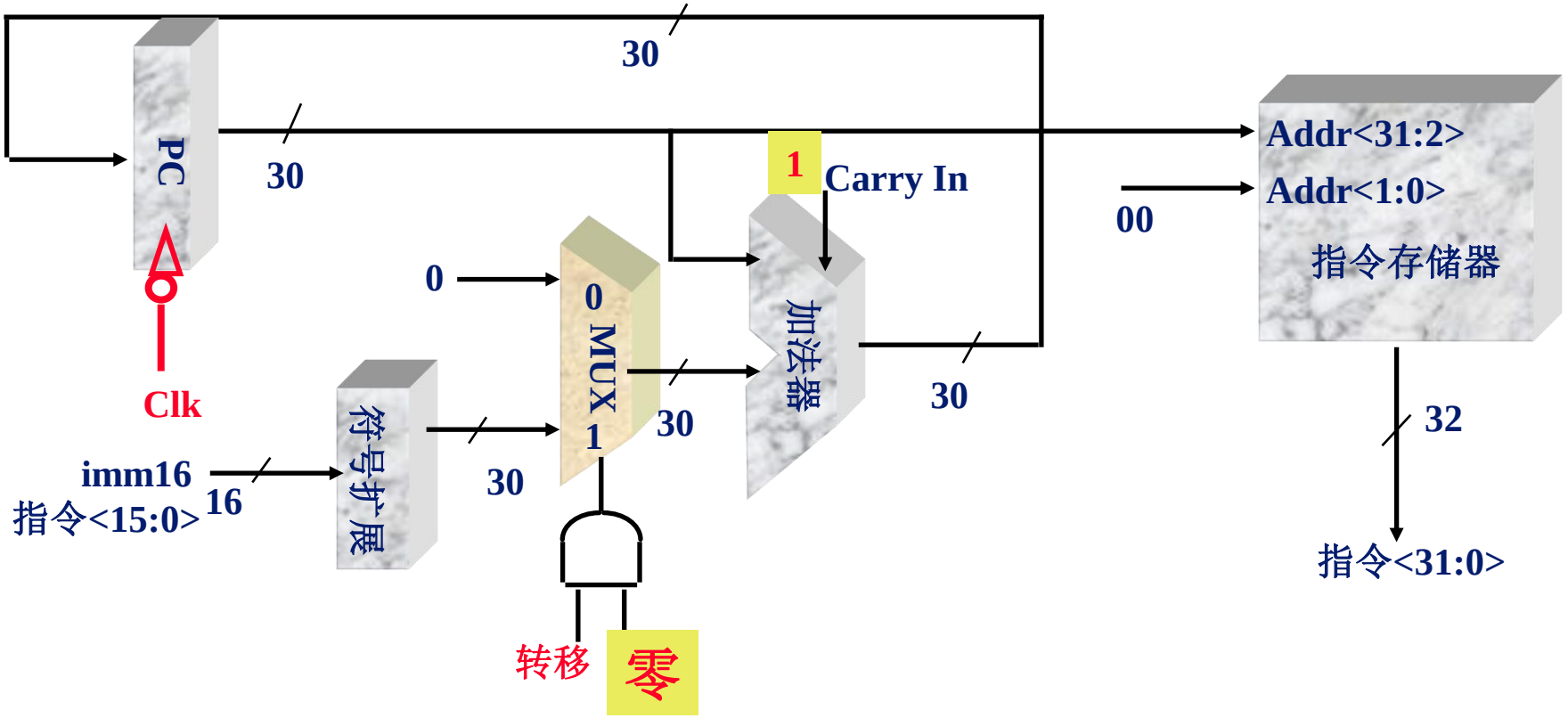
下一地址逻辑：昂贵、快速的解决方案

- 使用 30位PC:
  - 串行操作: PC<31:2> = PC<31:2> + 1
  - 转移操作: PC<31:2> = PC<31:2> + 1 + SignExt[Imm16]
  - 其他情况: 指令存储器地址 = PC<31:2> || 00



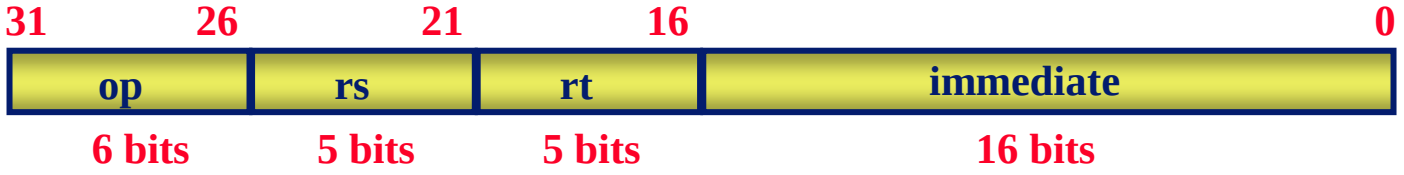
下一地址逻辑：价廉、慢速的解决方案

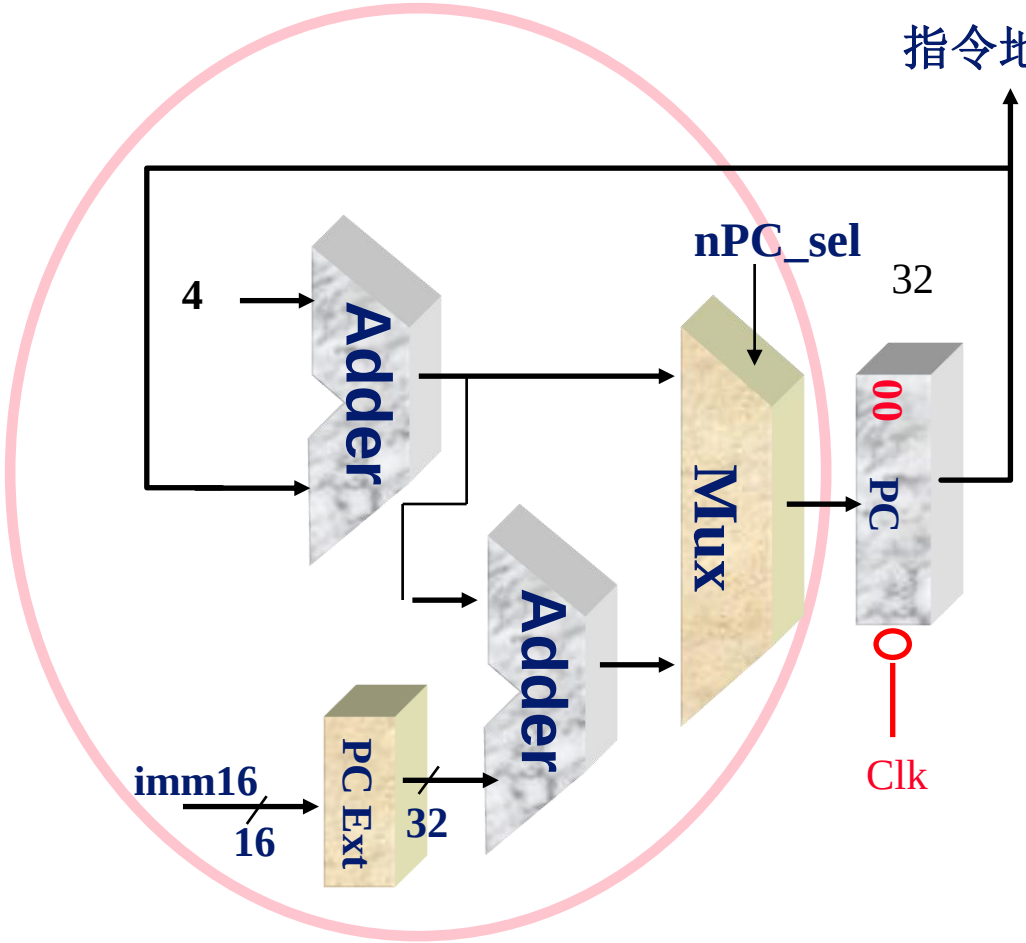
- 为什么慢?
  - 直到 “零”（ALU）的输出有效之后，才能开始地址加法
- 在整个设计中，这是否会影响到整体性能?
  - 这里，不会！ 因为，关键路径是装入操作。

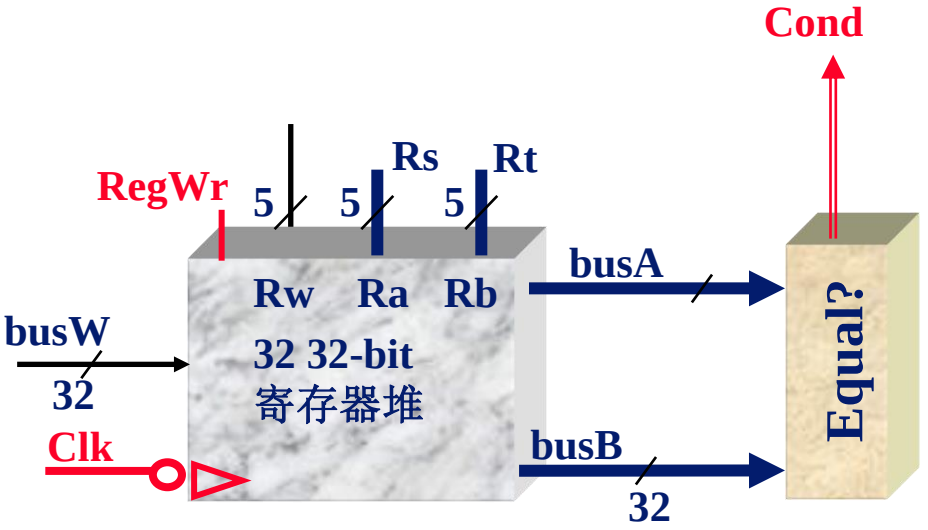


转移操作的数据通路

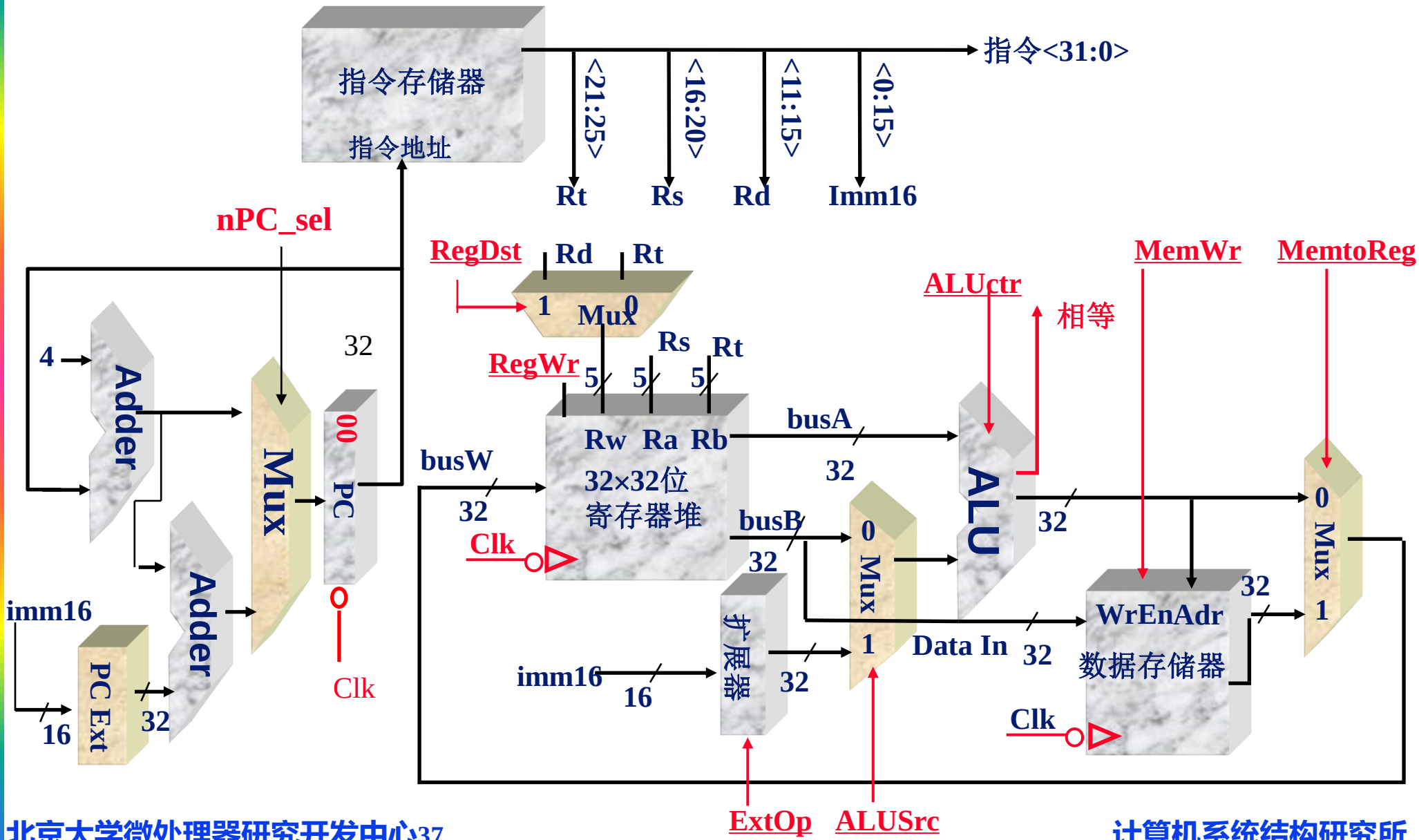
- beq rs, rt, imm16　　数据通路比较 Rs 和 Rt，生成条件（等于）！





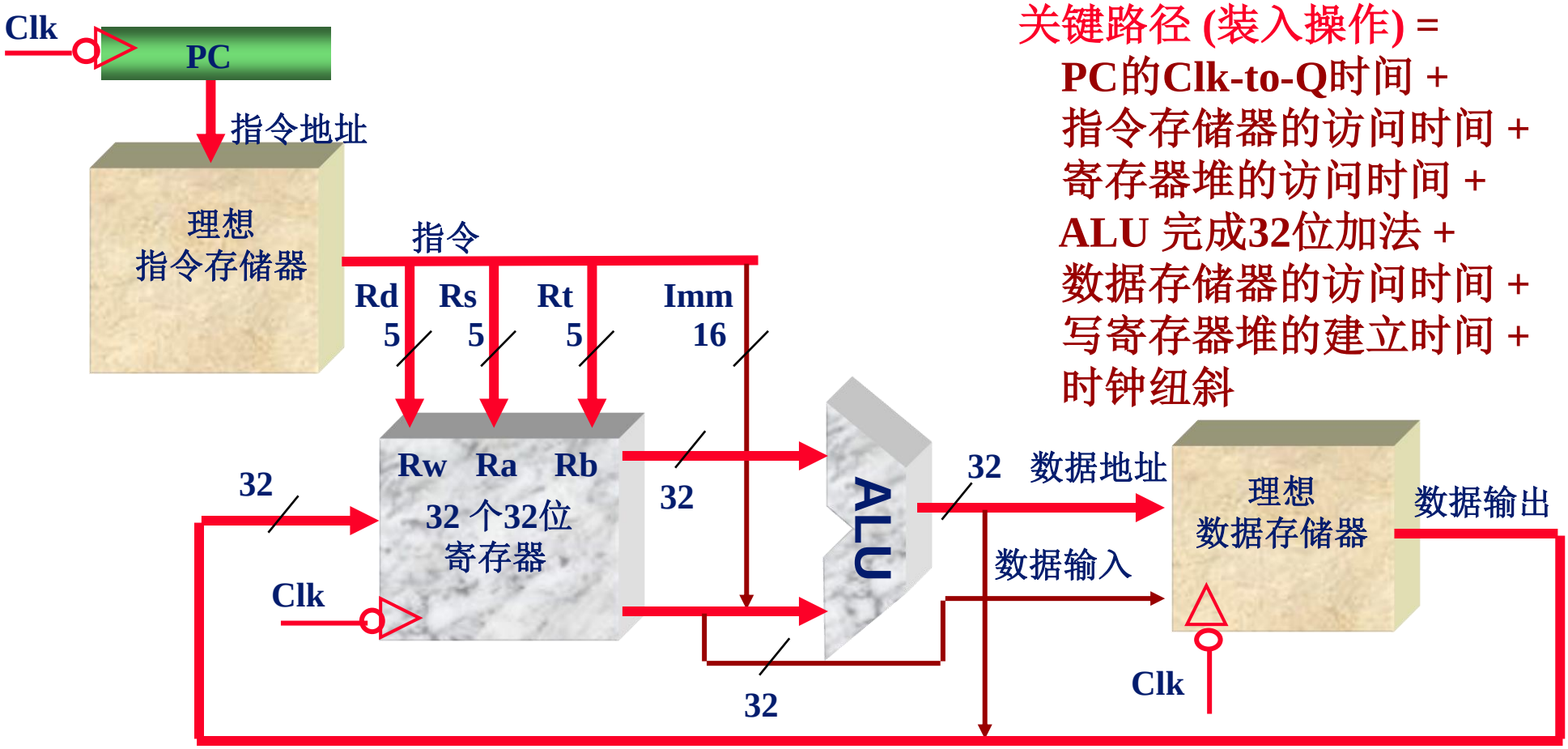


综上所述：单周期数据通路（不包括Jump指令）



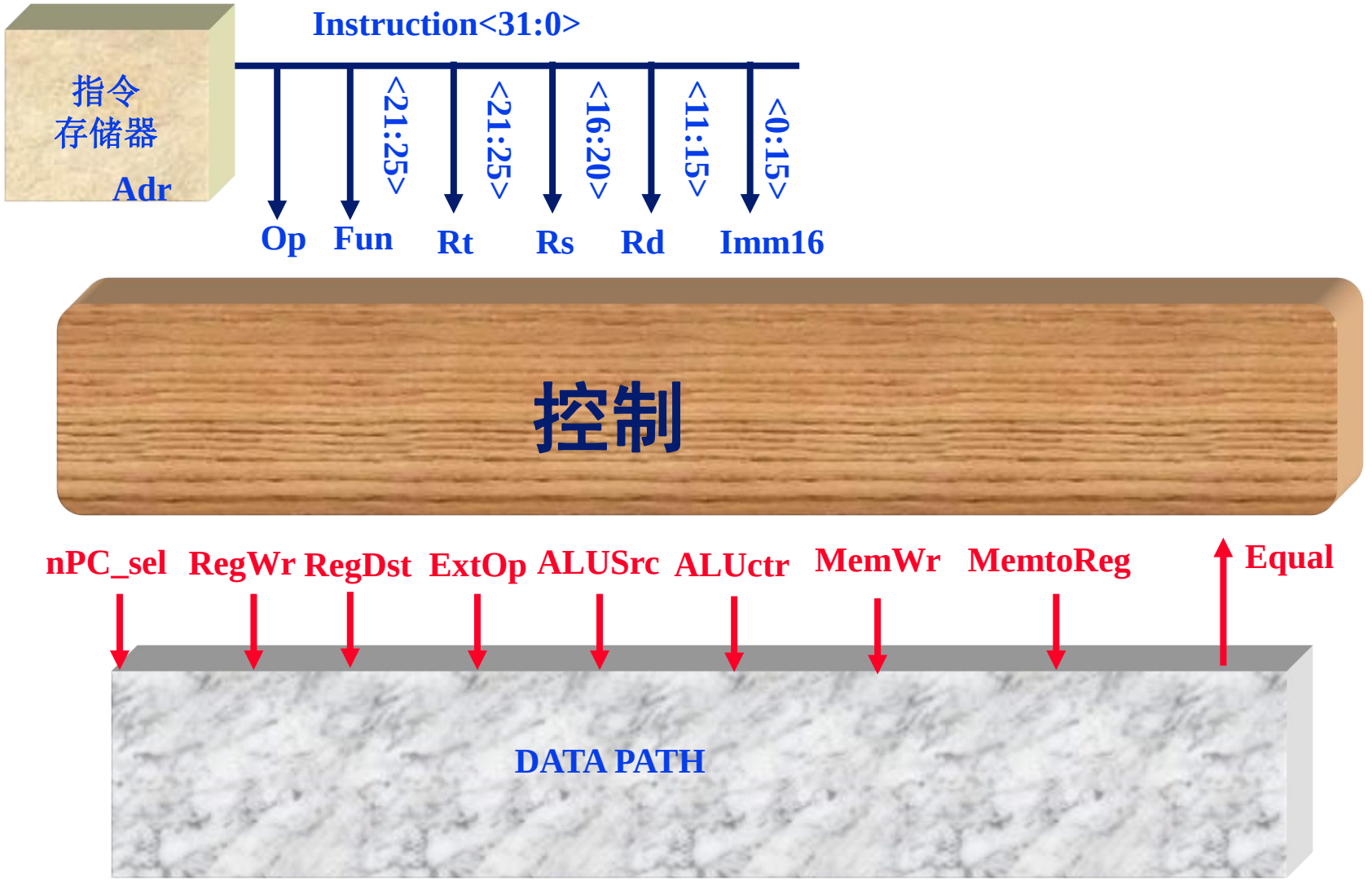
关键路径抽象

- 寄存器堆和理想存储器:
  - 只有在写操作中，CLK输入才能产生影响
  - 在读操作中，其行为与组合逻辑电路一样:
    - 地址 => 在访问时间(Access time)之后，输出有效。



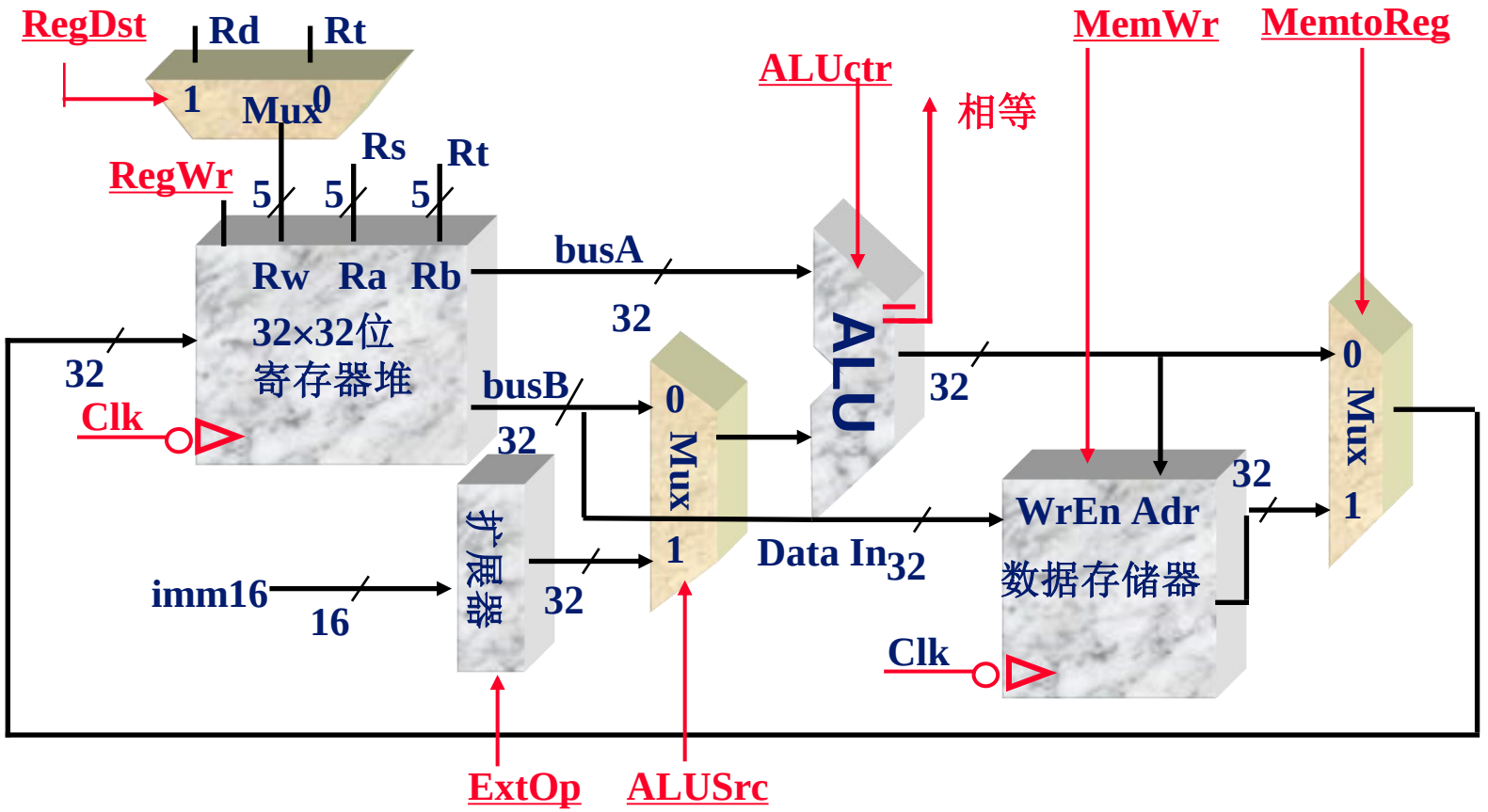
关键路径 (装入操作) =
PC的Clk-to-Q时间 +
指令存储器的访问时间 +
寄存器堆的访问时间 +
ALU 完成32位加法 +
数据存储器的访问时间 +
写寄存器堆的建立时间 +
时钟纽斜

第四步　给定数据通路：RTL ⇒ 控制



控制信号的含义

- ExtOp: Zero / Sign
- ALUsrc: 0 => regB; 1 => immed
- ALUctr: Add / Sub / Or
- MemWr: write memory
- MemtoReg: 1 => Mem
- RegDst: 0 => Rt 1 => Rd
- RegWr: write dest register



控制信号

指令　　寄存器传输

ADD　　R[rd] ⇐ R[rs] + R[rt];　　PC ⇐ PC + 4

ALUsrc = RegB, ALUctr = Add RegDst = rd, RegWr, nPC_sel = +4

SUB　　R[rd] ⇐ R[rs] - R[rt];　　PC ⇐ PC + 4

ALUsrc = ___, Extop = __, ALUctr = ___, RegDst = ___, RegWr(?), MemtoReg(?), MemWr(?), nPC_sel =__

ORi　　R[rt] ⇐ R[rs] + zero_ext(Imm16);　　PC ⇐ PC + 4

ALUsrc = ___, Extop = __, ALUctr = ___, RegDst = ___, RegWr(?), MemtoReg(?), MemWr(?), nPC_sel =__

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

ALUsrc = ___, Extop = __, ALUctr = ___, RegDst = ___, RegWr(?), MemtoReg(?), MemWr(?), nPC_sel =__

STORE　　MEM[R[rs] + sign_ext(Imm16)] ⇐ R[rs];　　PC ⇐ PC + 4

ALUsrc = ___, Extop = __, ALUctr = ___, RegDst = ___, RegWr(?), MemtoReg(?), MemWr(?), nPC_sel =__

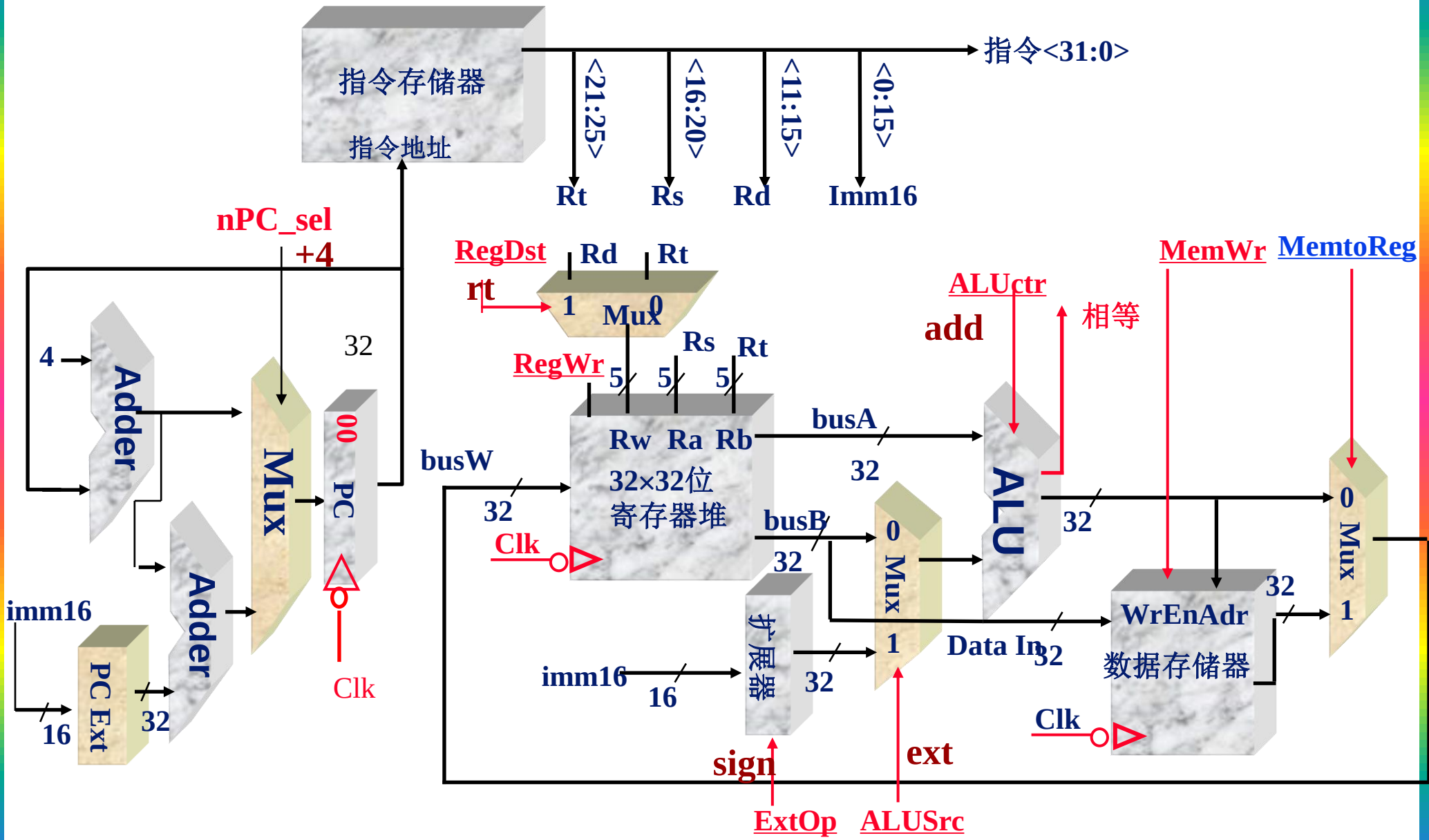
BEQ　　if (R[rs] == R[rt]) then PC ⇐ PC + sign_ext(Imm16)] || 00 else PC ⇐ PC + 4

ALUsrc = ___, Extop = __, ALUctr = ___, RegDst = ___, RegWr(?), MemtoReg(?), MemWr(?), nPC_sel =__

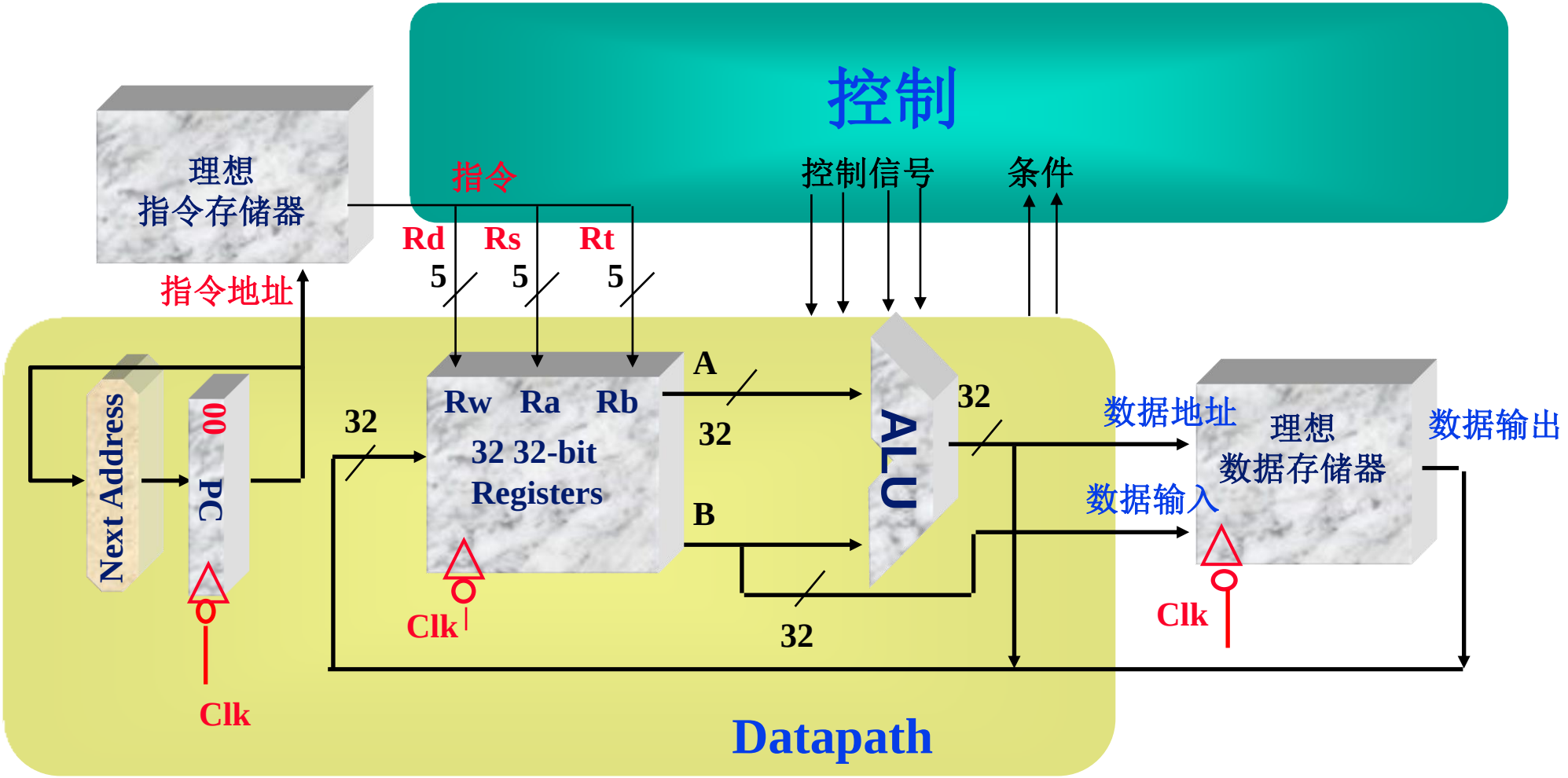
第五步：每个控制信号的逻辑

- nPC_sel ⇐ if (OP == BEQ) then EQUAL else 0
- ALUsrc ⇐ if (OP == 000000) then Reg B else Immed
- ALUctr ⇐ if (OP == 000000) 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

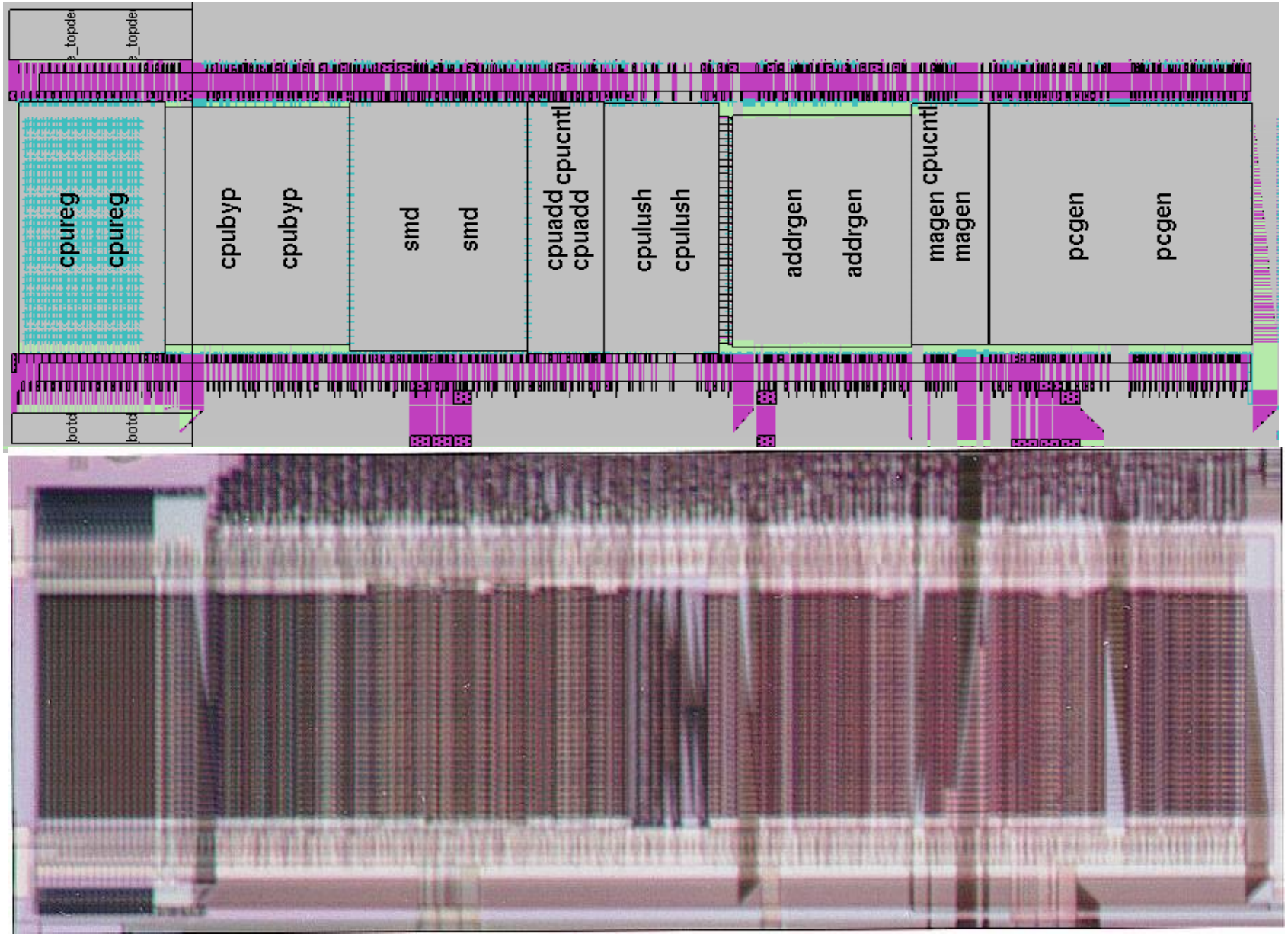
例如： Load指令


指令存储器
指令地址
指令<31:0>
<21:25>
<16:20>
<11:15>
<0:15>
Rt
Rs
Rd
Imm16
nPC_sel
+4
4
Adder
32
Mux
00
PC
Clk
imm16
PC Ext
16
32
Adder
RegDst
rt
Rd
Rt
1
Mux
0
RegWr
5
5
5
Rs
Rt
Rw Ra Rb
32×32位
寄存器堆
busW
32
Clk
busA
32
busB
32
ALUctr
add
相等
ALU
MemWr
MemtoReg
0
Mux
1
32
WrEnAdr
数据存储器
Data In
32
Clk
0
Mux
1
扩展器
imm16
16
32
sign
ext
ExtOp
ALUSrc

实现的抽象图


控制
理想
指令存储器
指令
控制信号
条件
Rd
Rs
Rt
5
5
5
指令地址
Next Address
PC
00
Clk
32
Rw Ra Rb
32 32-bit
Registers
Clk
A
B
32
32
32
ALU
数据地址
数据输入
理想
数据存储器
数据输出
Clk
Datapath

实际MIPS的数据通路

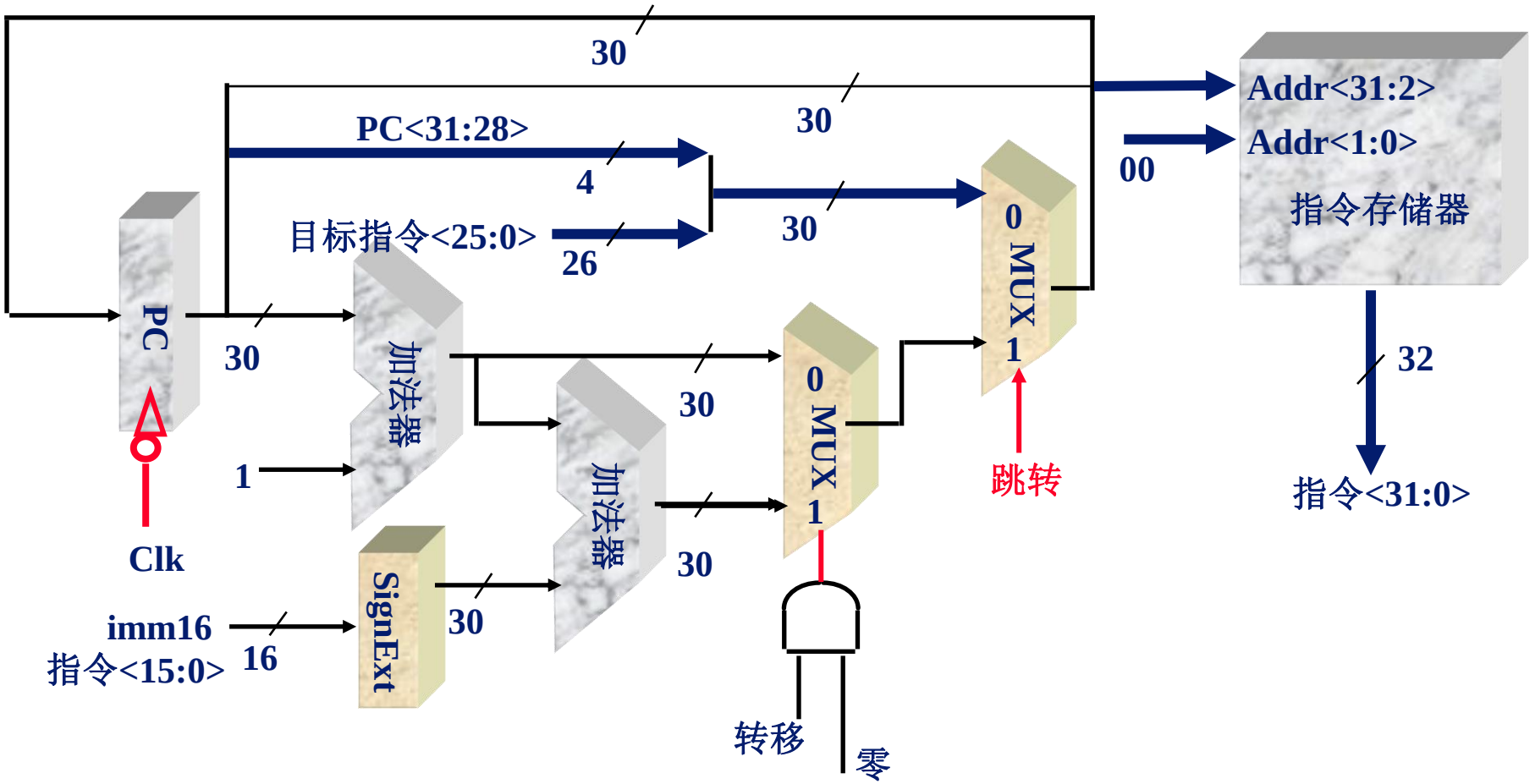


寄存器传输语言： 跳转指令

- j target
  - mem[PC] 从存储器中读取指令
  - PC<31:2> ⇐ PC<31:28> || target<25:0>
    计算下一条指令地址

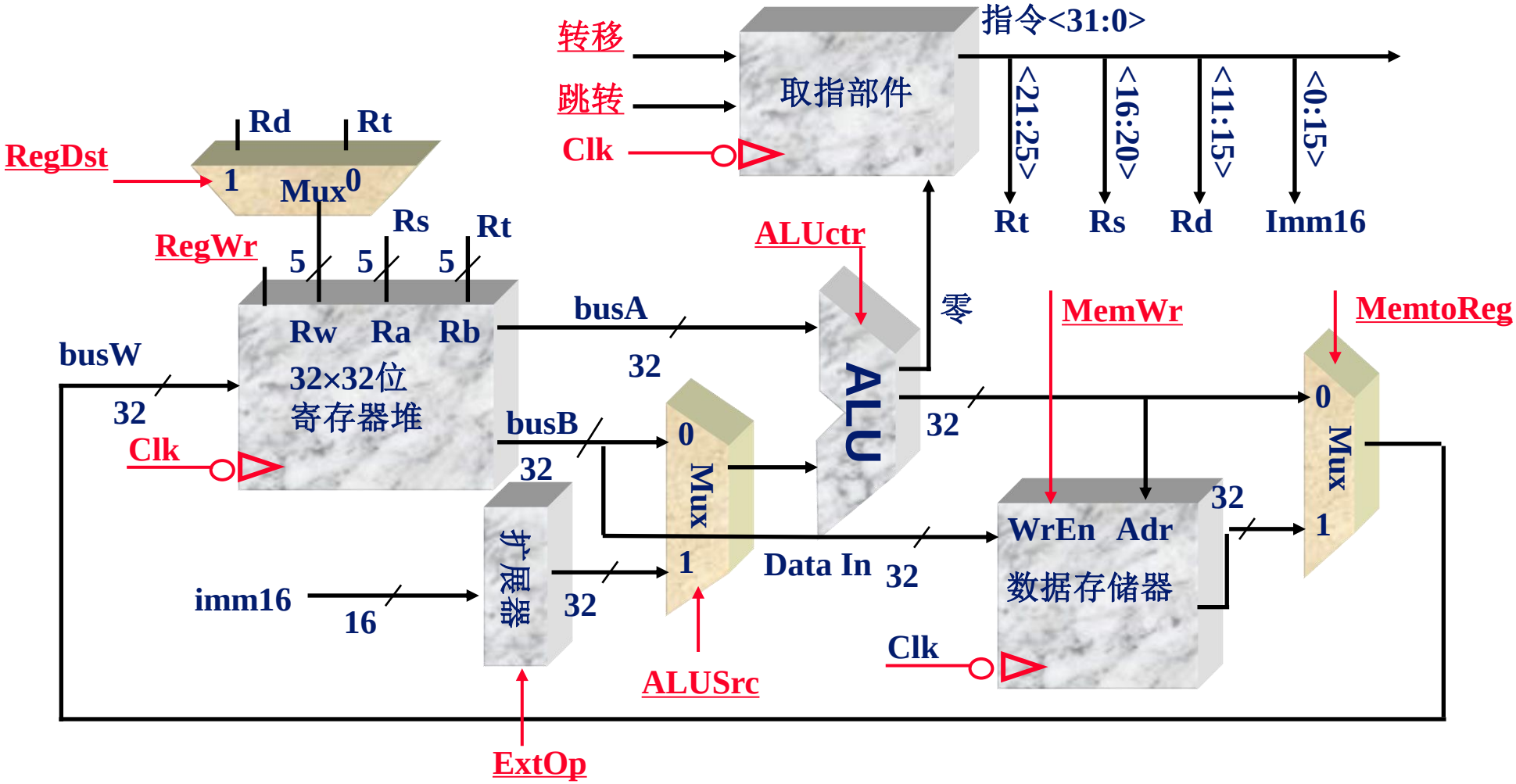
取指部件

- j target
  - PC<31:2> ⇐ PC<31:28> || target<25:0>



综上所述：单周期数据通路（包括Jump指令）

- 除了*控制信号*，我们已经学会了单周期数据通路的设计



本讲小结

- 设计处理器的五个步骤
  1. 分析指令系统 => 数据通路 需求
  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

- MIPS 可以简化上述工作
  - 所有指令具有相同的大小
  - 源寄存器都在相同的位置
  - 立即数的场位大小、位置恒定
  - 操作都作用于　寄存器 / 立即数

- 单周期数据通路⇒ CPI=1, 时钟周期时间 ⇒ 很长

- 下一讲: 实现控制逻辑