Chapter 4

Building a Computer

The Processor

- P&H
  - Sections 4.1 – 4.4

Great Idea #1: Abstraction (Levels of Representation/Interpretation)

High Level Language Program (e.g., C)

Compiler

Assembly Language Program (e.g., RISC-V)

Assembler

Machine Language Program (RISC-V)

Hardware Architecture Description (e.g., block diagrams)

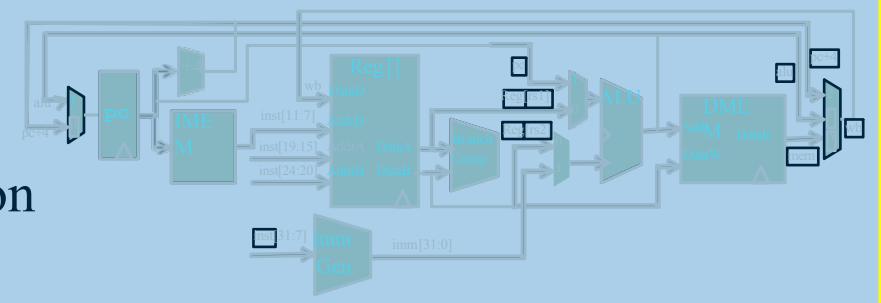
Architecture Implementation

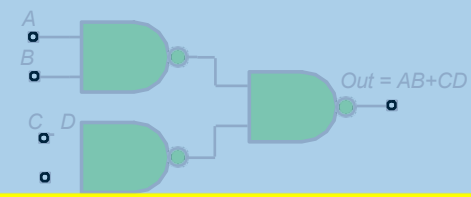
Logic Circuit Description (Circuit Schematic Diagrams)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw    x3, 0(x10)
lw    x4, 4(x10)
sw    x4, 0(x10)
sw    x3, 4(x10)
```

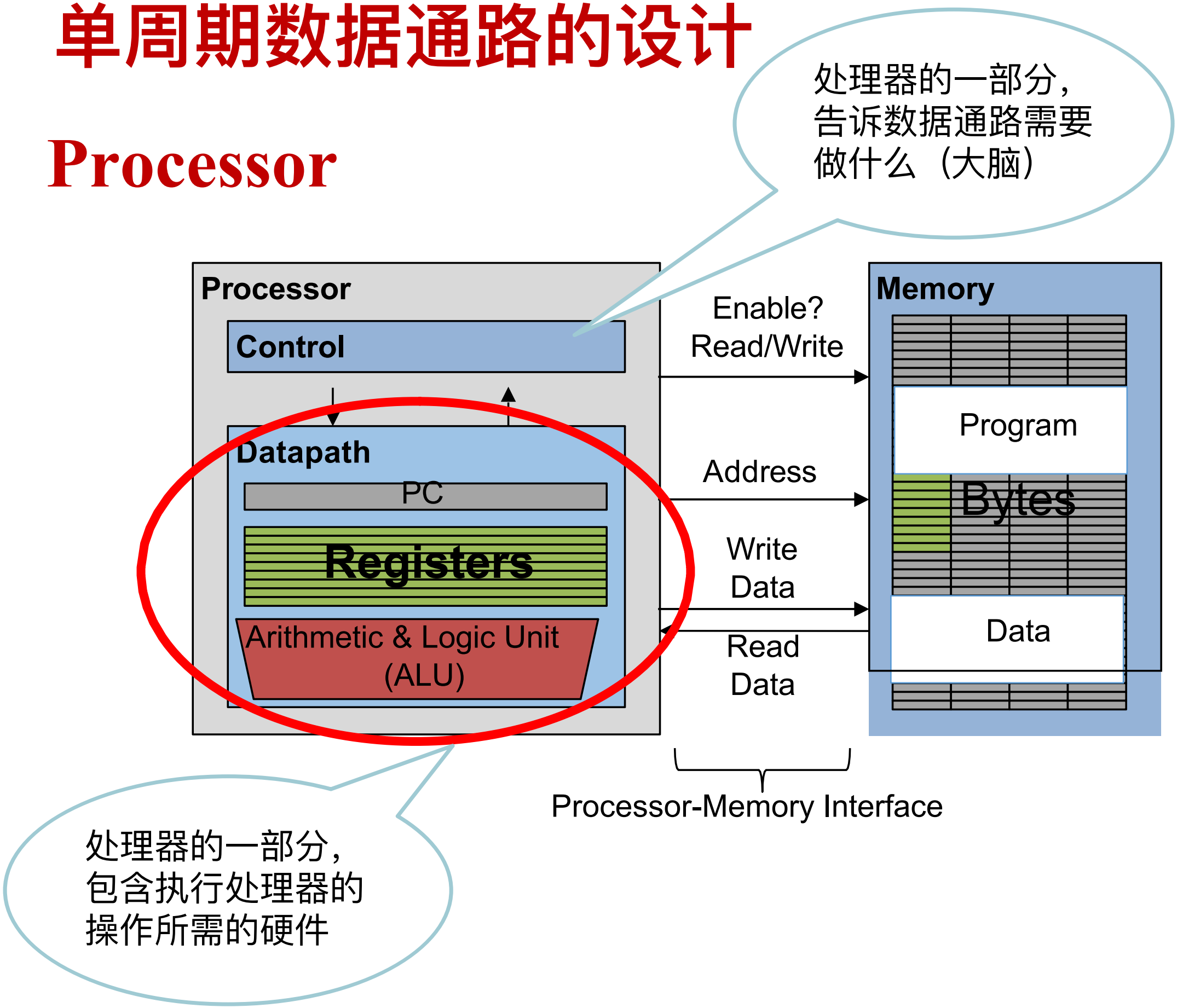
```
1000 1101 1110 0010 0000 0000 0000 0000
1000 1110 0001 0000 0000 0000 0000 0100
1010 1110 0001 0010 0000 0000 0000 0000
1010 1101 1110 0010 0000 0000 0000 0100
```





单周期数据通路的设计

Processor



Datapath

Datapath

执行算术运算的处理器部件，把状态部件、计算部件互连起来，共同为执行期间处理器中的数据流和转换提供通道。

Datapath Elements

ALU 是数据通路的一个部件，其他部件还有什么？

计算部件

组合电路输出随输入变化，比如 ALU

状态部件

- 时序逻辑电路
- 输出随时钟边沿变化
- 比如寄存器 Register

指令处理

5个通用步骤

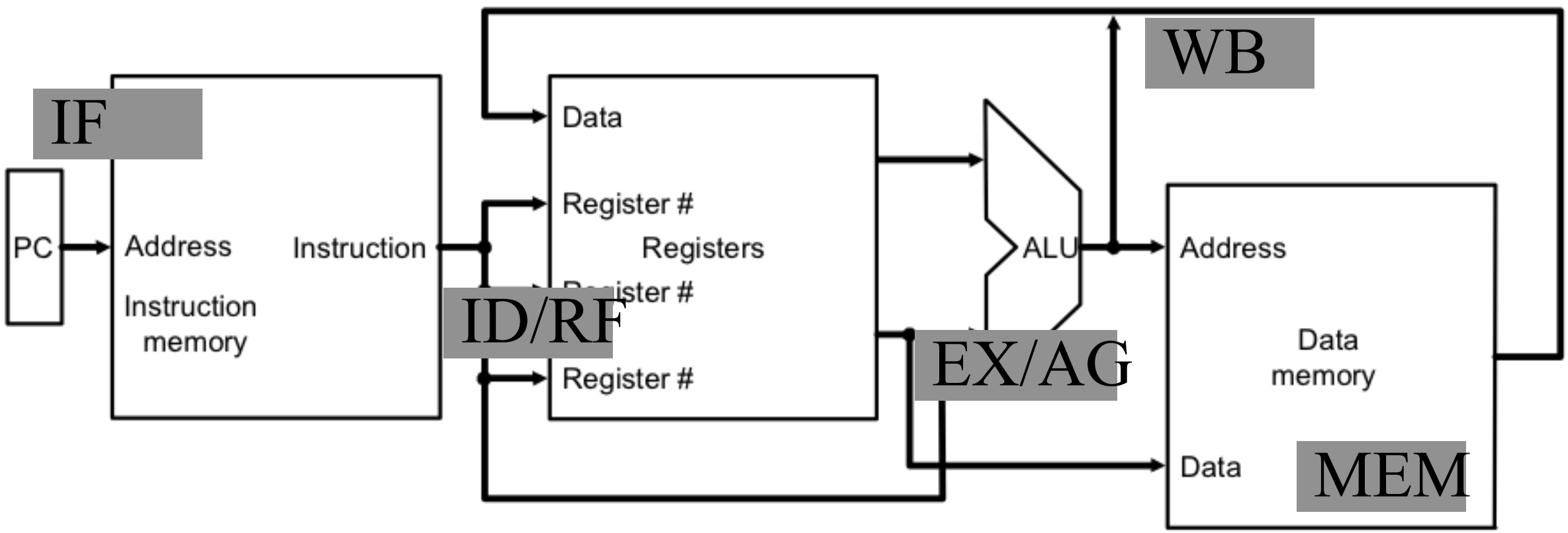
指令提取（IF）

指令解码和寄存器操作数提取（ID/RF）

执行/评估内存地址（EX/AG）

内存操作数提取 (MEM)

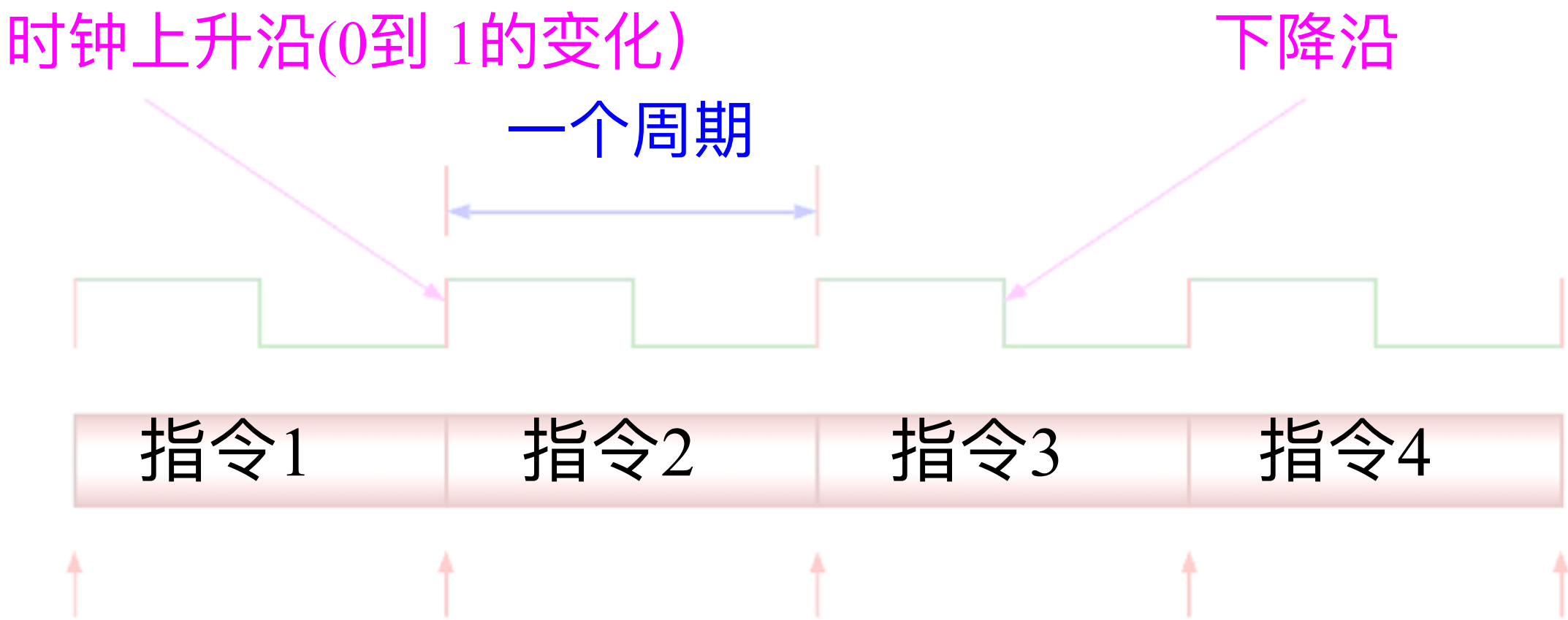
存储/回写结果（WB）



单周期CPU

单周期(Single-Cycle) CPU执行一条指令用一个时钟周期

(最简单的执行方式　　)



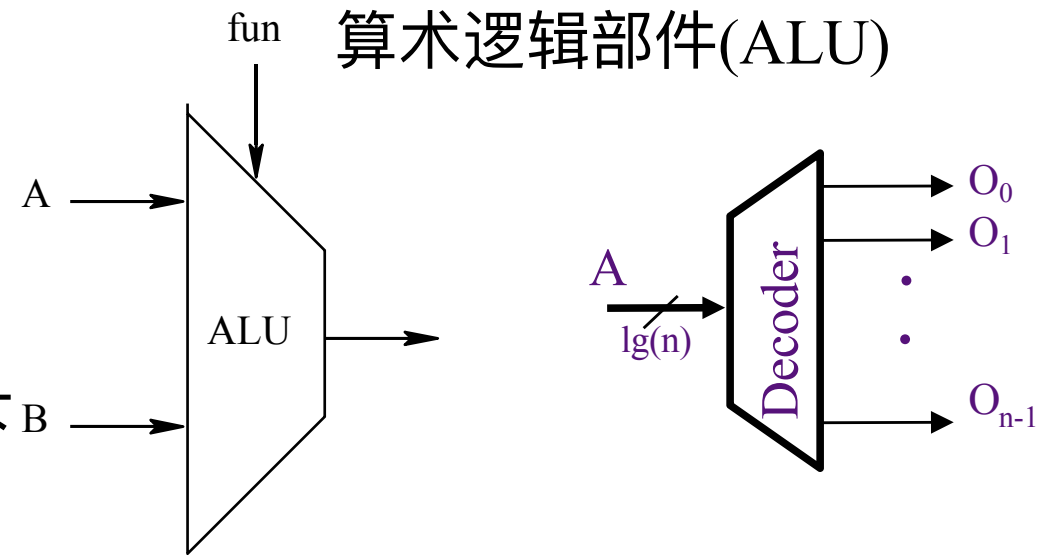
在时钟上升沿保存指令的结果和下一条指令的地址

=> Data path 上的资源每指令周期最多用一次

数据通路部件

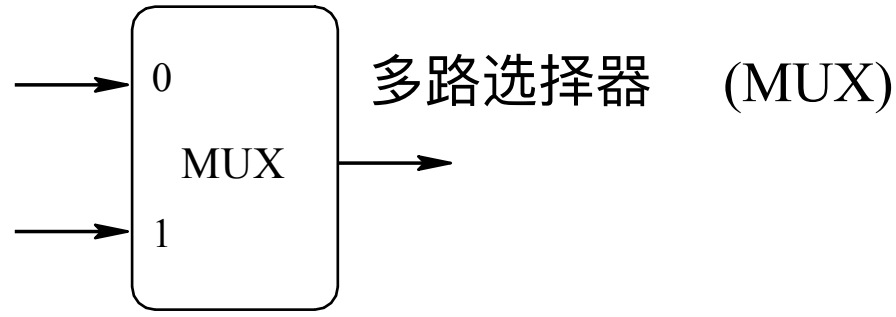
组合逻辑元件的特点

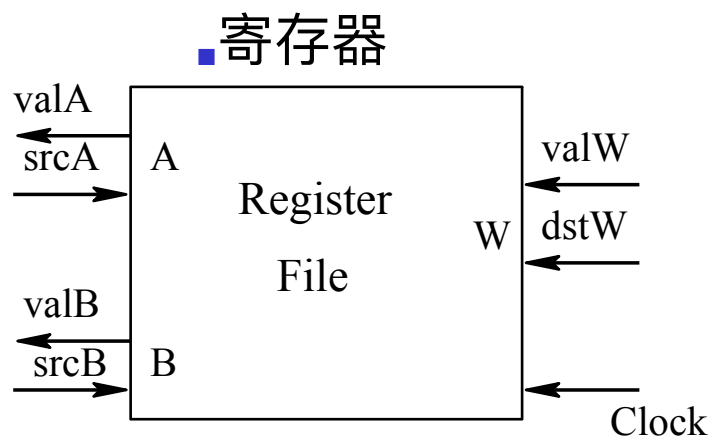
- 其输出只取决于当前的输入
- 定时：所有输入到达后，经过一定的逻辑门延时，输出端改变，并保持到下次改变，不需要时钟信号来定时

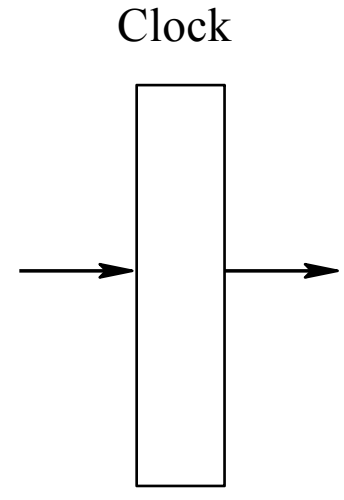


时序逻辑部件

具有存储功能，在时钟控制下输入状态被写到电路中，直到下一个时钟到达，输入端状态由时钟决定何时写入，输出端状态随时可读出

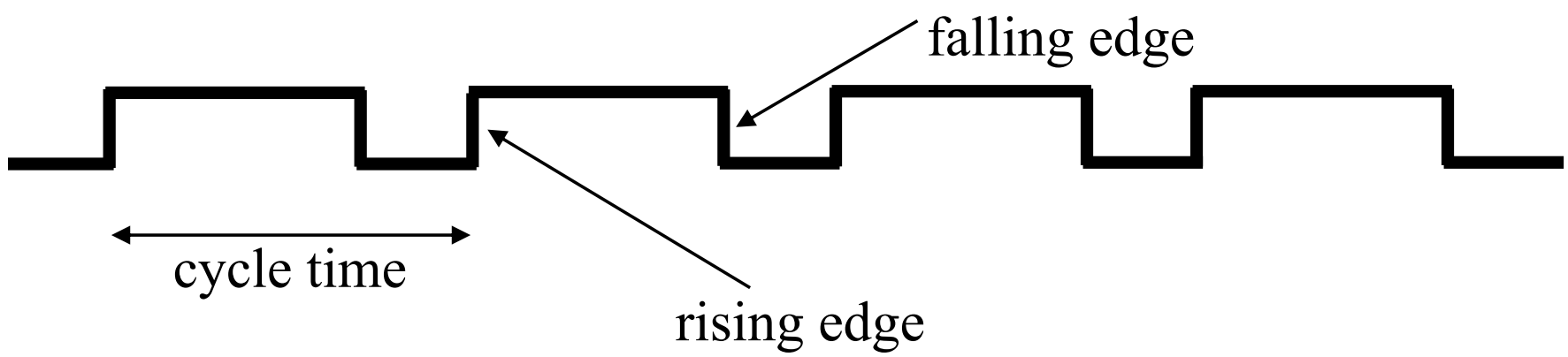




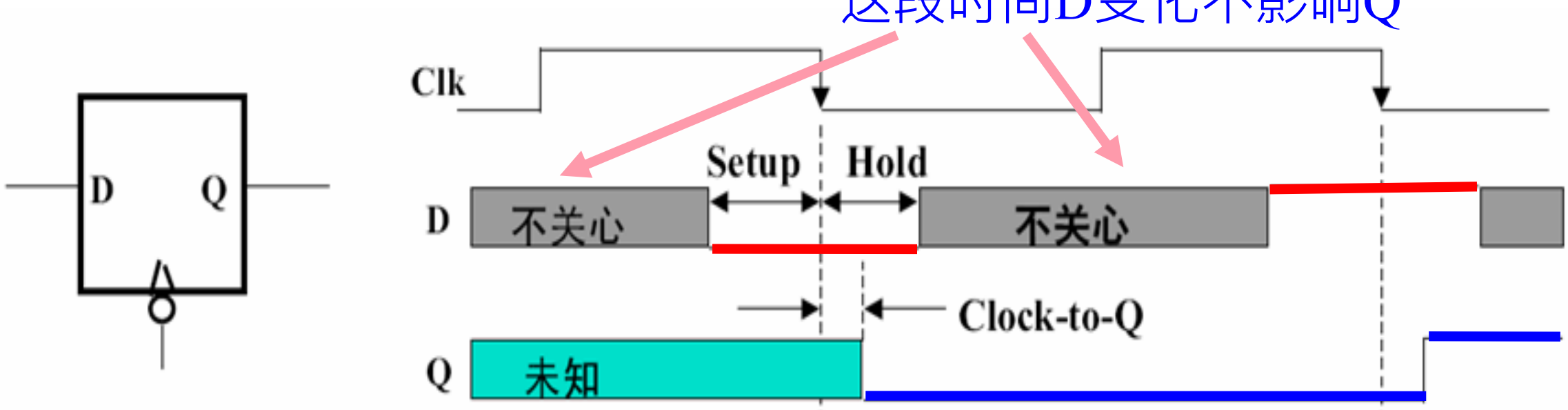


时序逻辑电路 State (sequential) elements

- 定时方式:规定信号何时写入状态元件或何时从状态元件读出
  - 边沿触发(edge-triggered)方式
    - 状态单元中的值只在时钟边沿改变。每个时钟周期改变一次
- 最简单的状态单元
  - D触发器：一个时钟输入、一个状态输入、一个状态输出



回顾——D触发器



Q总是在clk-Q后跟着D变化

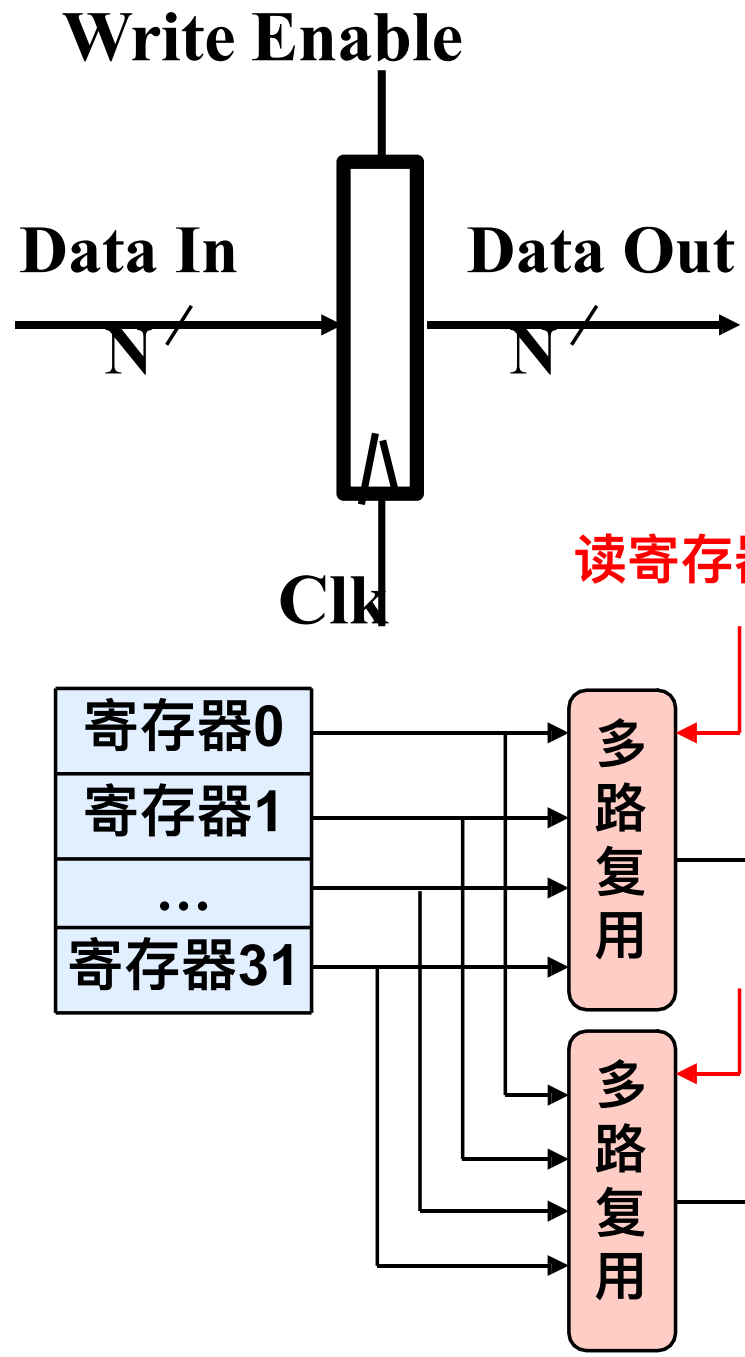
- 建立时间(Set Time):在触发时钟边沿之前输入必须稳定
- 保持时间(Hold Time):在触发时钟边沿之后输入必须保持
- Clock-to-Q-time:在触发时钟边沿，输出并不能立即变化

切记：状态单元的输入信息总是在一个时钟边沿到达后的“Clk-to-Q”时才被写入到单元中，此时的输出才反映新的状态值

数据通路中的状态元件有两种：寄存器(组) + 存储器

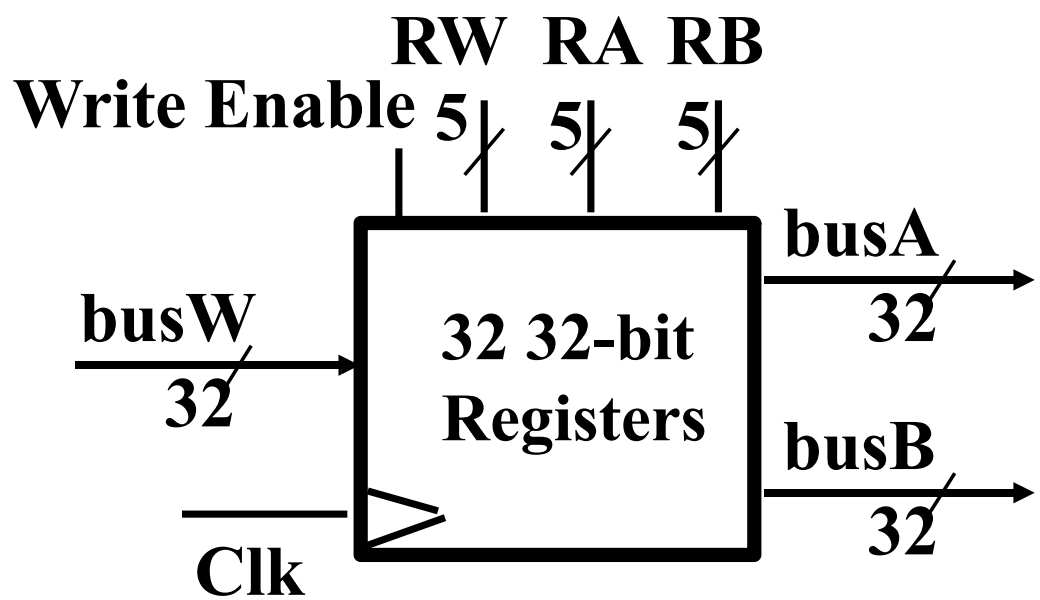
存储单元:
寄存器(Basic Building Block)

- 寄存器
  - 类似D触发器，有以下不同
    - N-bit 输入输出
    - 可以同时从两个口读取,
    - 有写入使能端
- 写入使能:
  - 禁止时(0): 寄存器内容不变
  - 使能时(1): 改变

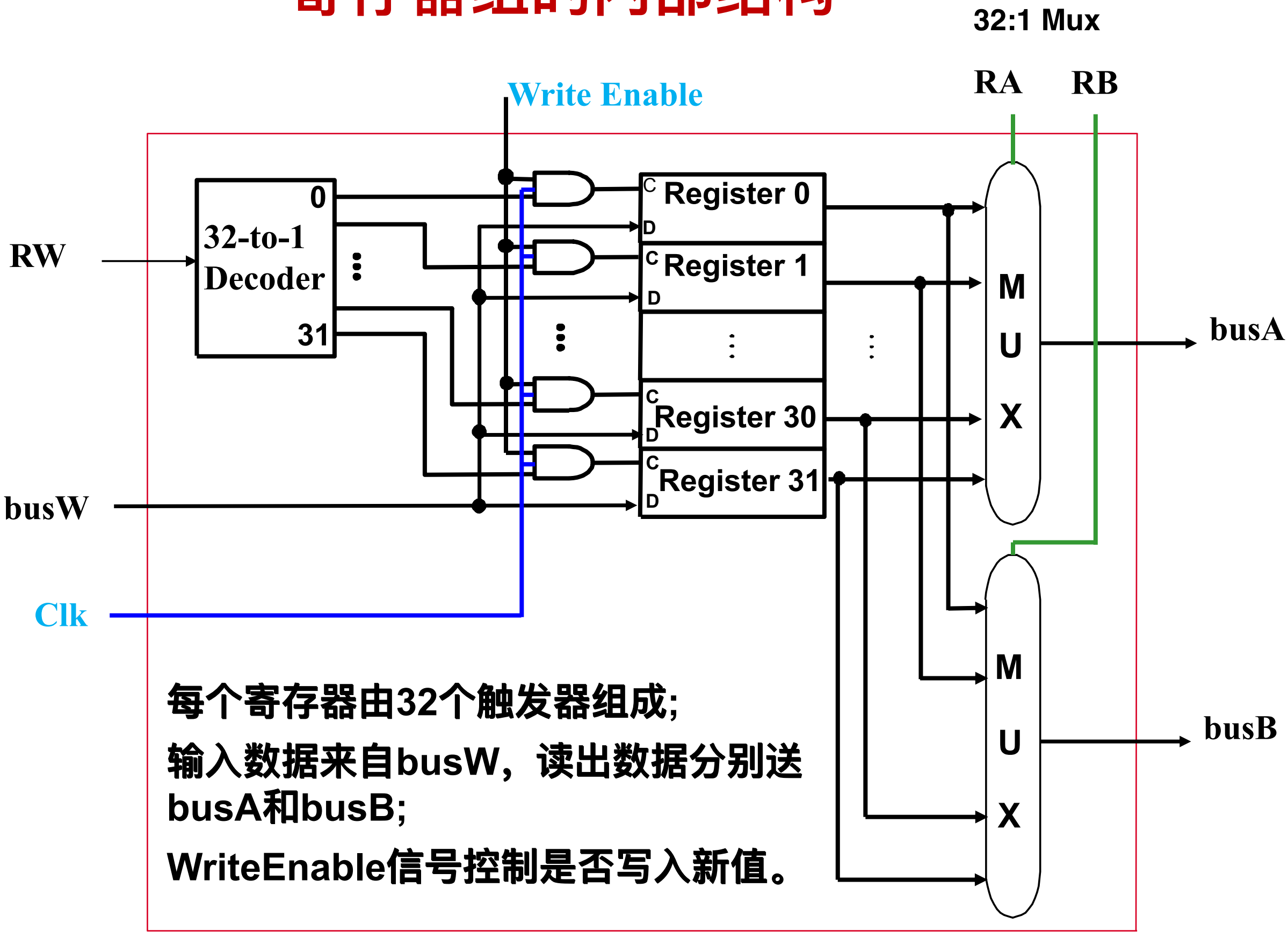


存储单元: 寄存器组

- 寄存器组**包含** 32 registers:
  - 两个 32-bit 输出总线: busA and busB
  - 一个32-bit 输入总线: busW
  - 寄存器组中的寄存器选择:
    - RA (number) 选出哪个32位寄存器的数据放在busA (data)
    - RB (number)选出哪个32位寄存器的数据放在busB (data)
    - RW (number)选出哪个32位寄存器的数据将通过busW被写入(当WE=1时)
  - 时钟 (CLK)
  - CLK 只在写人时发挥作用
  - 读取电路总能读到值
    - RA or RB valid => busA or busB valid after “access time.”



寄存器组的内部结构



每个寄存器由32个触发器组成;

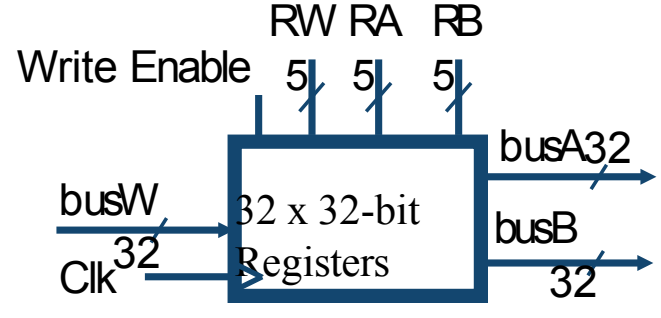
输入数据来自busW，读出数据分别送busA和busB;

WriteEnable信号控制是否写入新值。

Datapath Elements: State and Sequencing

- **Reg file in Verilog**

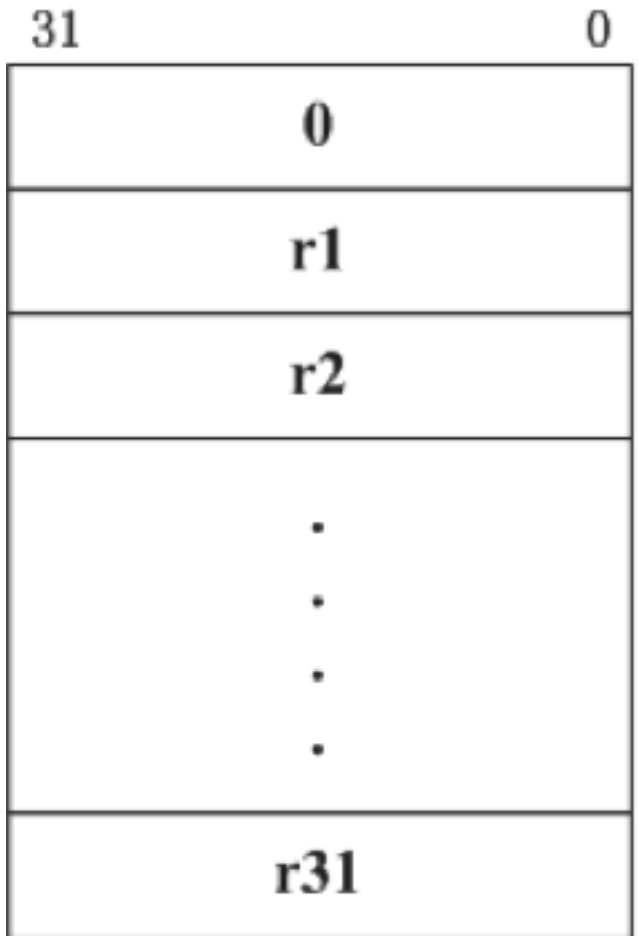
```
module rv32i_regs (
    input clk, wen,
    input [4:0] rw,
input [4:0] ra,
input [4:0] rb,
input [31:0] busw,
output [31:0] busa,
output [31:0] busb
);
    reg [31:0] regs [0:30];
    always @(posedge clk)
    if (wen)
    regs[rw] <= busw;
    assign busa = (ra == 5'd0) ? 32'd0: regs[ra];
    assign busb = (rb == 5'd0) ? 32'd0: regs[rb];
    endmodule
```



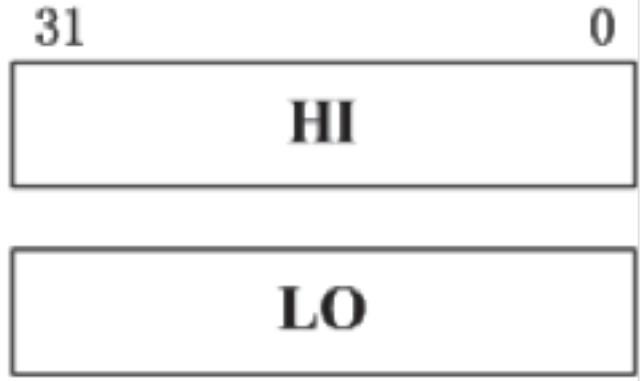
Mips寄存器结构

- 32个通用寄存器
- 程序计数器PC
- HI LO

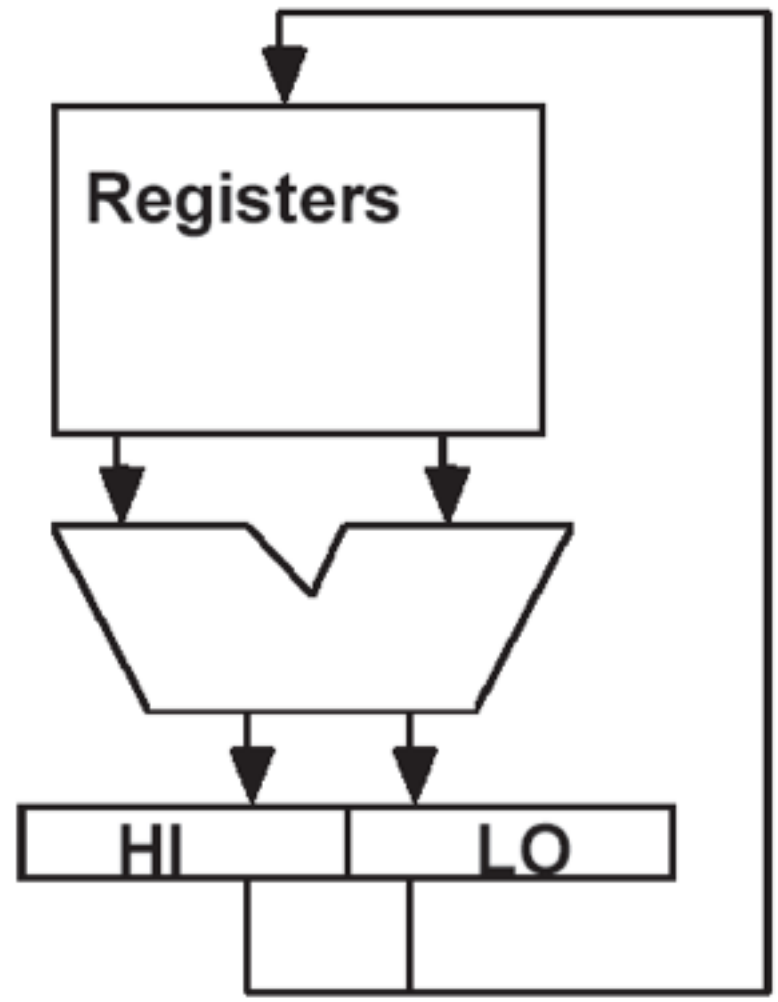
通用寄存器



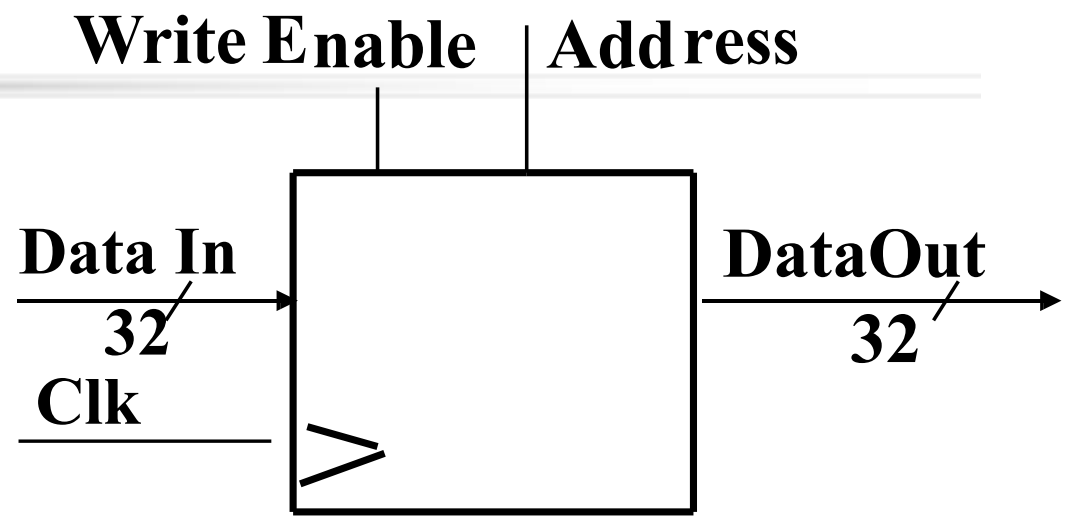
乘/除寄存器



程序计数器



存储单元: 存储器



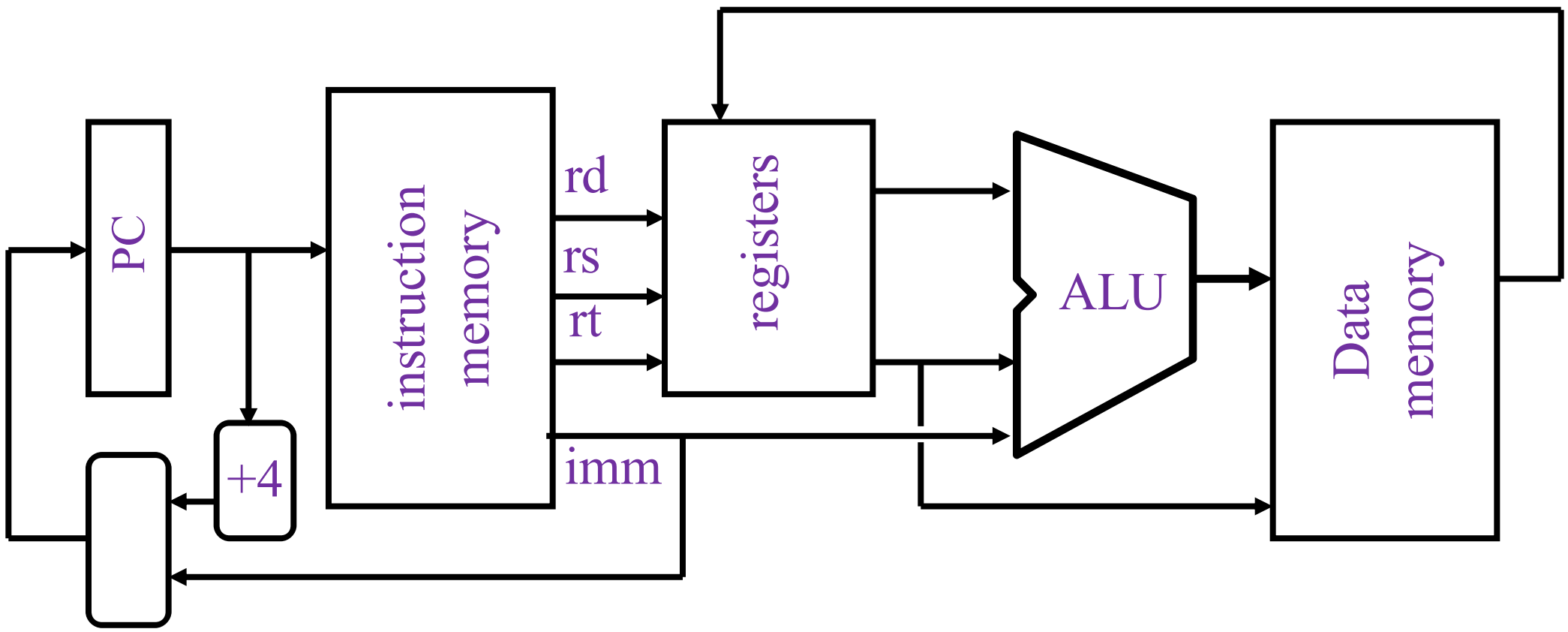
- 存储器(容量大)
  - 一个输入总线: 数据输入
  - 一个输出总线: 数据输出
- 存储器选择:
  - 通过地址线Address 选择哪个32位字被放在数据输出总线上
  - Write Enable = 1: 通过address选择哪个地址的32位字写为输入总线的数据
  - Clock input (CLK)
    - CLK input 只在写入有效
    - 在读出时, 类似于组合逻辑:
      - Address valid => Data Out valid after “access time.”

单周期数据通路设计

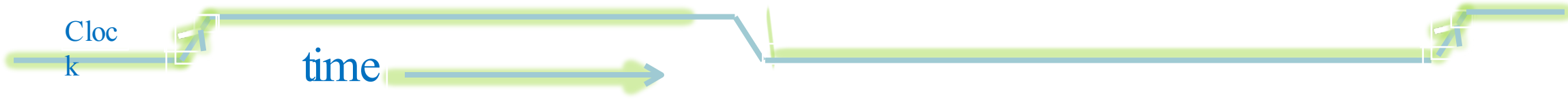
- 问题: 将 “执行整个指令”的块做为一个整体
  - 太大(该块要执行从取指令开始的所有操作)
  - 效率不高
- 解决方案: 将 “执行整个指令” 的操作分解为多个阶段(stage)，然后将所有阶段连接在一起产生整个datapath
  - 每一阶段更小，从而更容易设计
  - 方便优化其中一个阶段，而不必涉及其他阶段

单周期数据通路设计

Generic Steps of Datapath



1. Instruction Fetch
2. Decode/ Register Read
3. Execute
4. Memory
5. Reg. Write



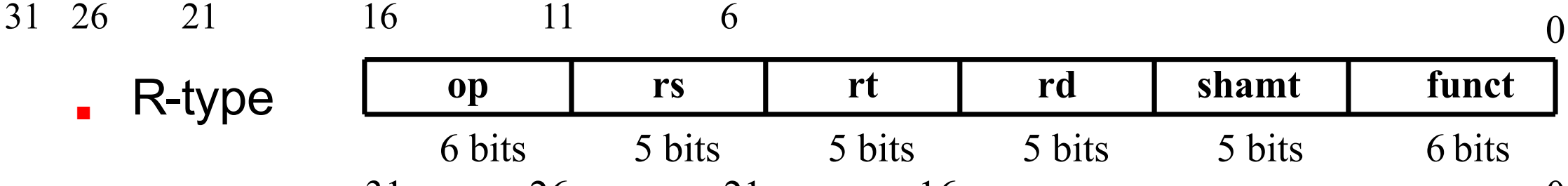
处理器设计: step-by-step

1. 分析指令集 => datapath 需求
   - 每个指令的含义都用寄存器传输级（RTL）给出
   - datapath 必须包含ISA所需要的各个寄存器
     - 为了实现功能还需要一些额外的寄存器
   - datapath 必须支持每一类寄存器转移操作
2. 选择datapath中包含的模块,确定这些模块的功能,时序
3. 组装datapath
4. 根据指令集决定datapath中需要的控制信号
5. 设计控制信号逻辑

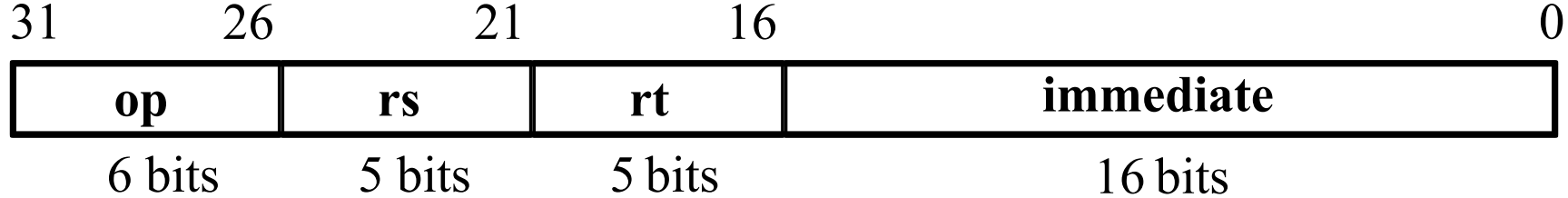
Analyze Instruction Set

- 三类MIPS指令:

- R-type



- I-type



- J-type

- 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

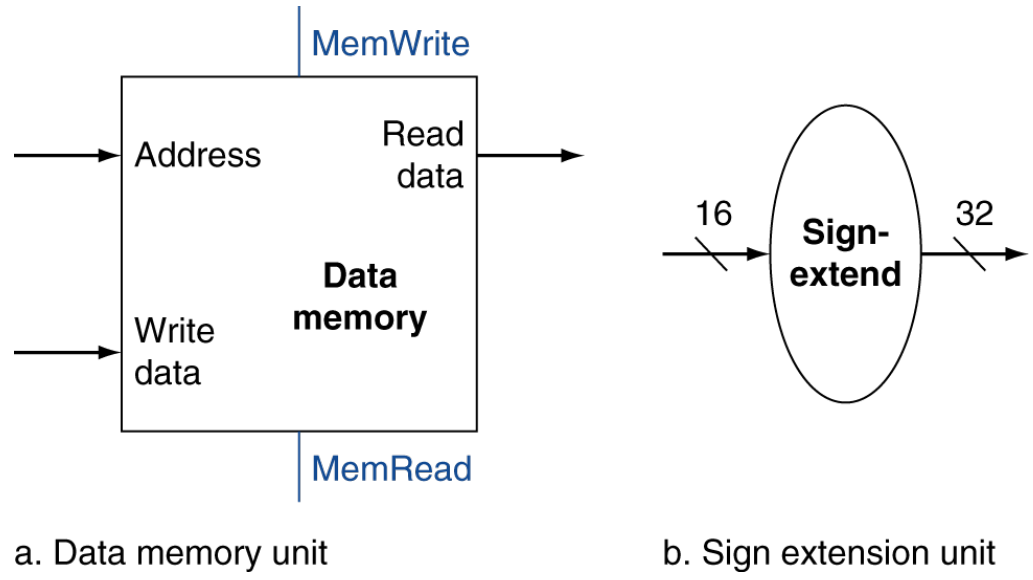
指令系统需求分析

- **Memory**
- 存指令 和 数据
- **Registers (32 x 32)通用寄存器**
  - read RS
  - read RT
  - Write RT or RD
- **PC 程序计数器**

  Extender for zero- or sign-extension

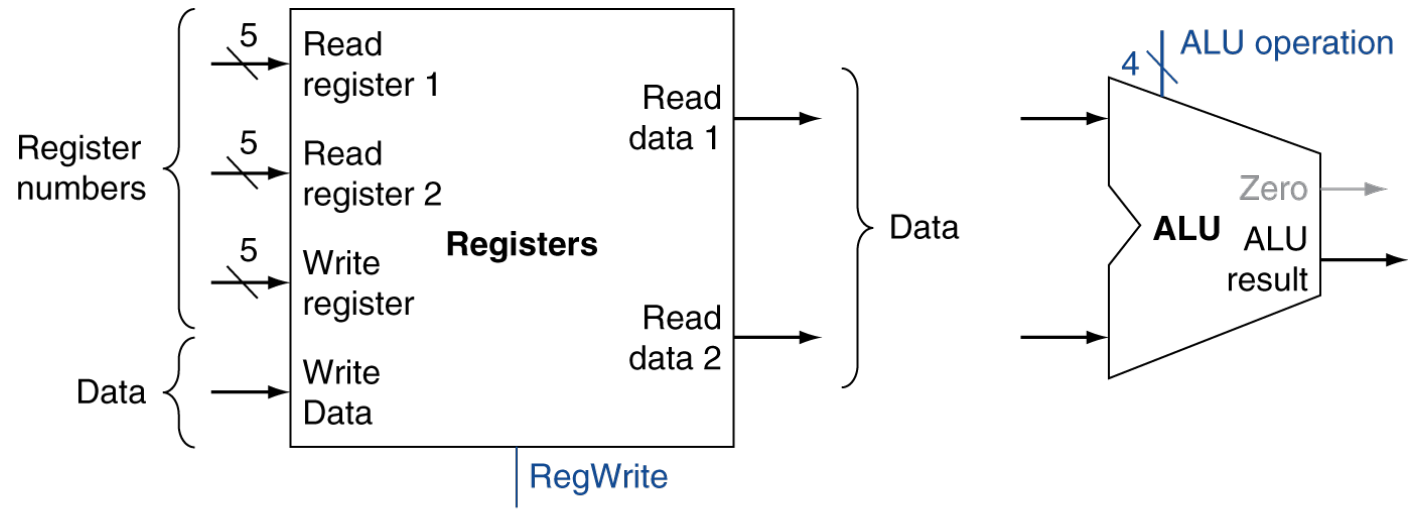
  Add 4 or extended immediate to PC
- **Execute**

  ALU



a. Data memory unit

b. Sign extension unit



a. Registers

b. ALU

Stage 1: 取指令单元IF

所有的指令都以取指开始

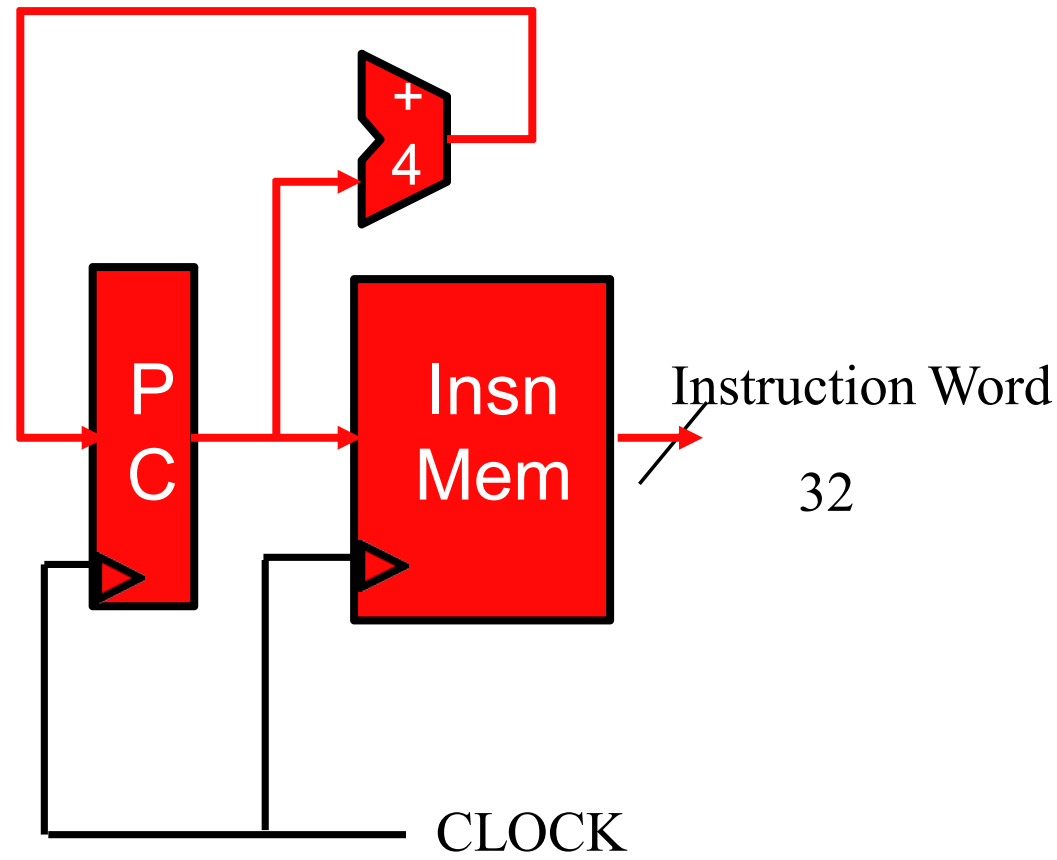
op | rs | rt | rd | shamt | funct = MEM[PC]

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

- 所有指令都公用的单元
  - **Fetch the Instruction: mem[PC]**
  - **更新PC:**
    - 顺序指令时:

      PC <= PC + 4
    - 分支跳转时:

      PC <= “something else”



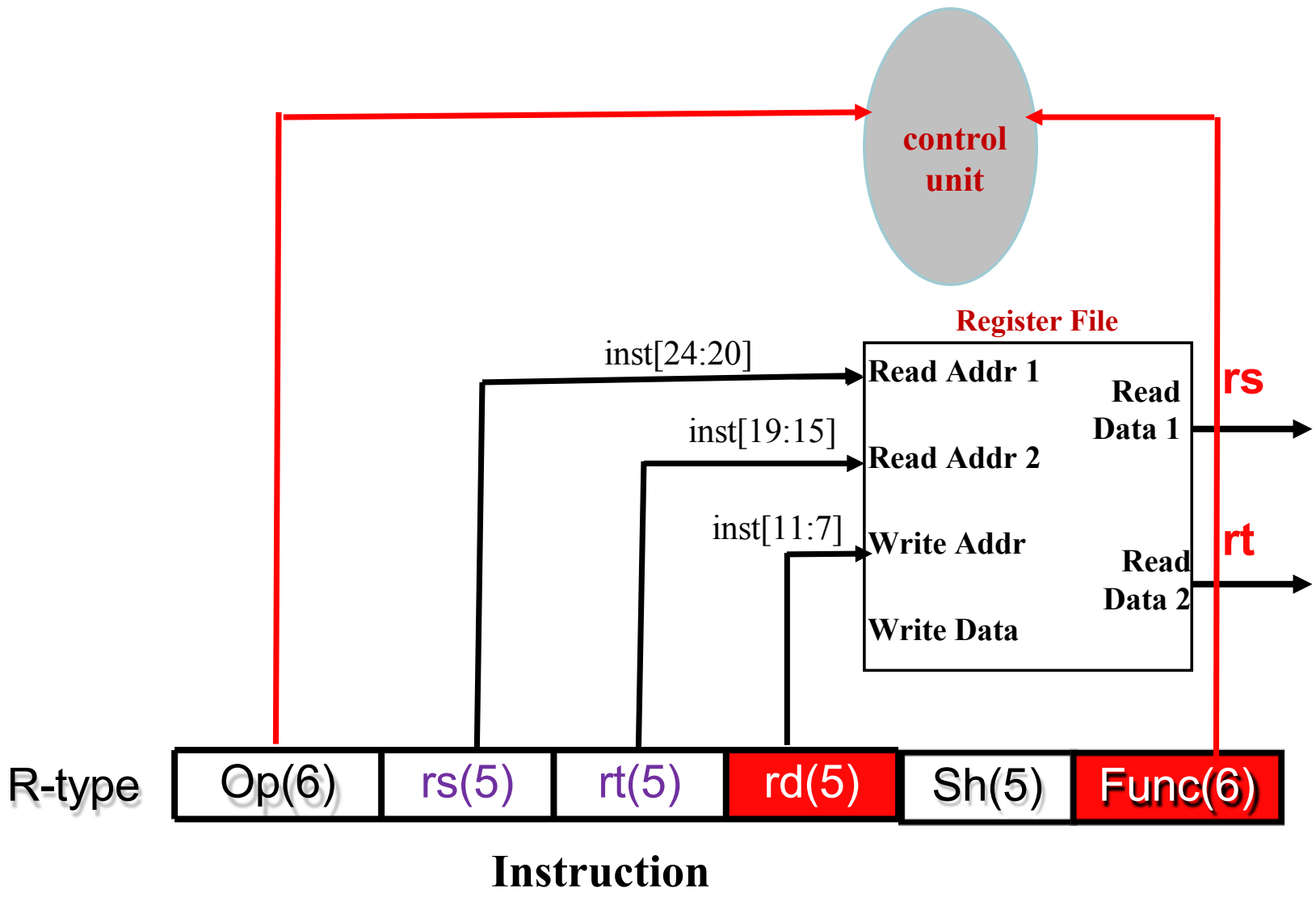
顺序：先取指令，再改变PC的值

(具体实现可以并行)

决不能先改变PC的值，再取指令

Stage 2: 指令译码Instruction Decode

- 在取到指令后, 下一步从各域(fields)中得到数据(对必要的指令数据进行*解码*)
- 首先，读出Opcode，以决定指令类型
- 接下来，从相关部分读出数据



for add, read two registers
for addi, read one register
for jal, no reads necessary

Reading Registers “Just in Case”

- 请注意，在译码阶段，使用 rs 和 rt 指令的字段地址从寄存器堆的读取端口读出寄存器内容，对所有指令都是工作的
  - 因为还没有译码，所以还不知道指令要做什么！
  - 提前读出两个源操作数内容，以防后面用到两个寄存器的内容。
- 几乎所有指令都会用到寄存器内容 (除了 j 指令)

Stage 3: ALU (Arithmetic-Logic Unit)

- 大多数指令的实际工作在此部完成:

  算术指令 (+, -, *, /), shifting, logic (&, |), comparisons (slt)

- what about loads and stores?
  - lw $t0, 40($t1)
  - 要访问的内存地址 = $t1的值 + 40
  - LW/SW 内存地址计算在这阶段完成

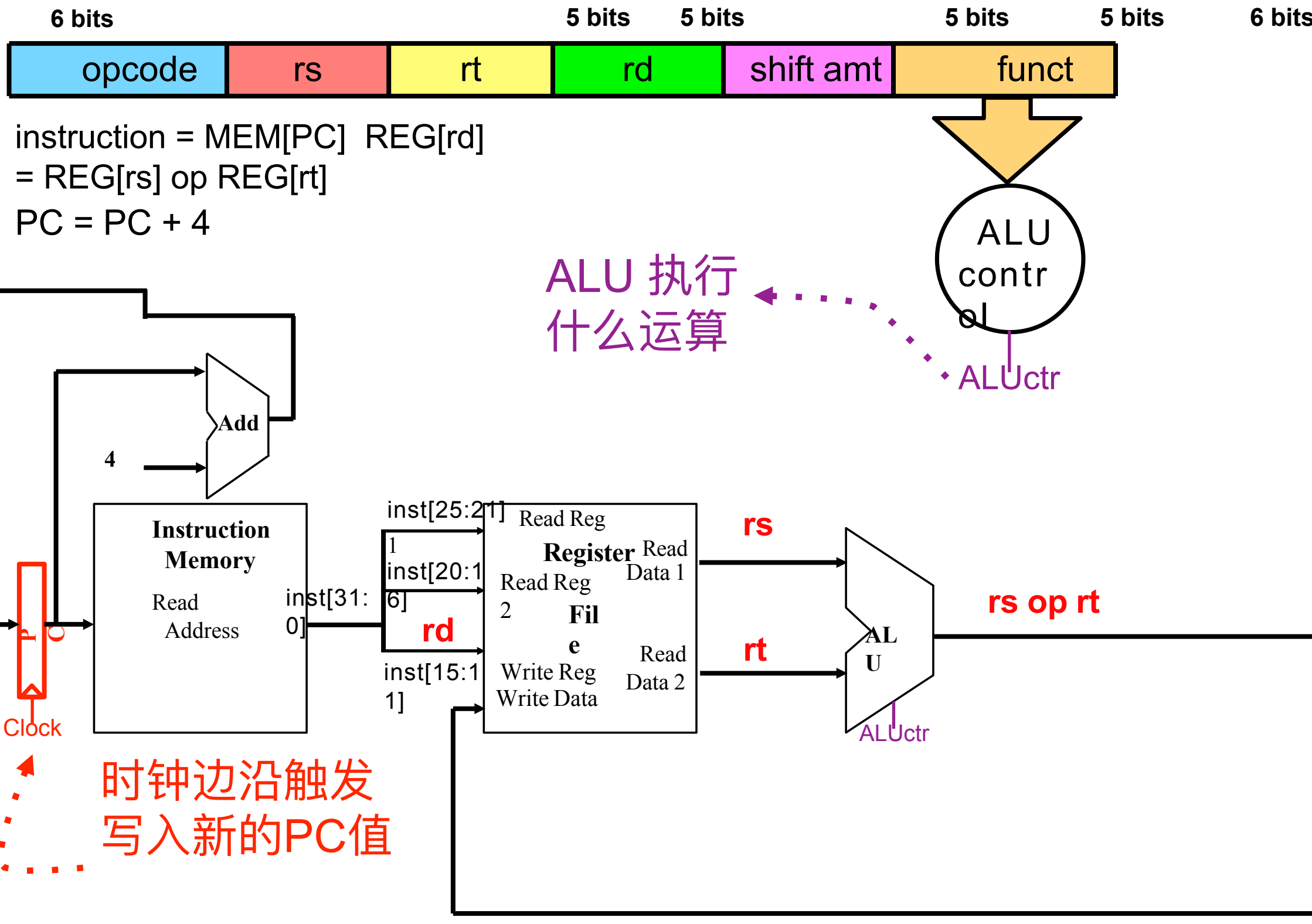
Stage 4: 内存访问Memory Access

- 事实上只有load和store指令在此stage会做事; 其它指令在此阶段空闲idle或者直接跳过本阶段
- 由于load和store需要此步，因此需要一个专门的阶段stage来处理他们
- 由于cache系统的作用，该阶段有望加速
- 如果没有caches，本阶段会很慢

Stage 5: 写寄存器Register Write

- 大多数指令会将计算结果写到寄存器
- 例如: arithmetic, logical, shifts, loads, slt
- what about stores, branches, jumps?
  - 内存写SW、条件分支、绝对跳转指令不需要最后阶段的寄存器写操作。

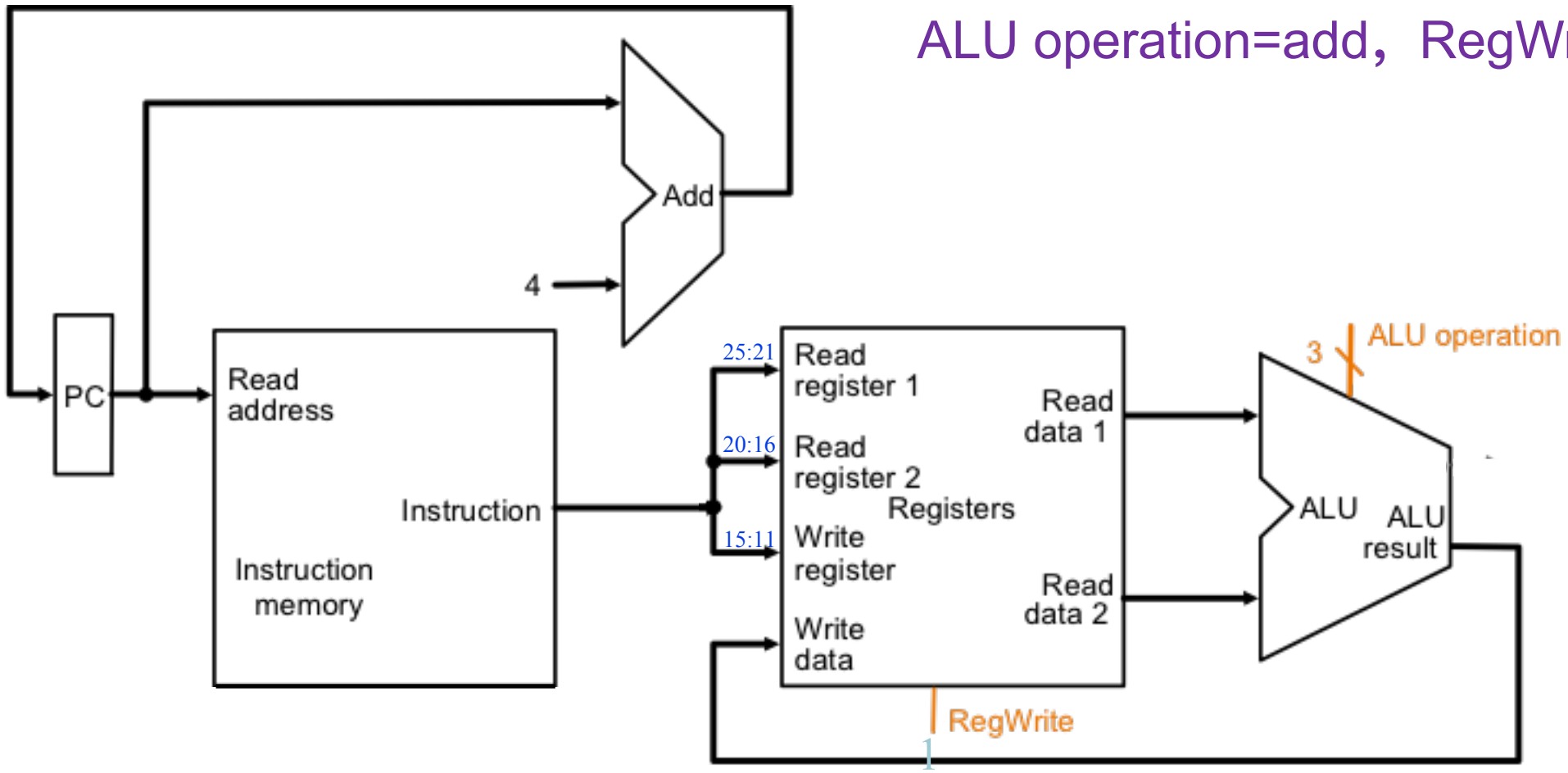
MIPS 指令的硬件实现 R-type 指令



ALU Datapath

指令“add rd, rs, rt” 的控制信号?

ALU operation=add，RegWrite=1



if MEM[PC] == ADD rd rs rt

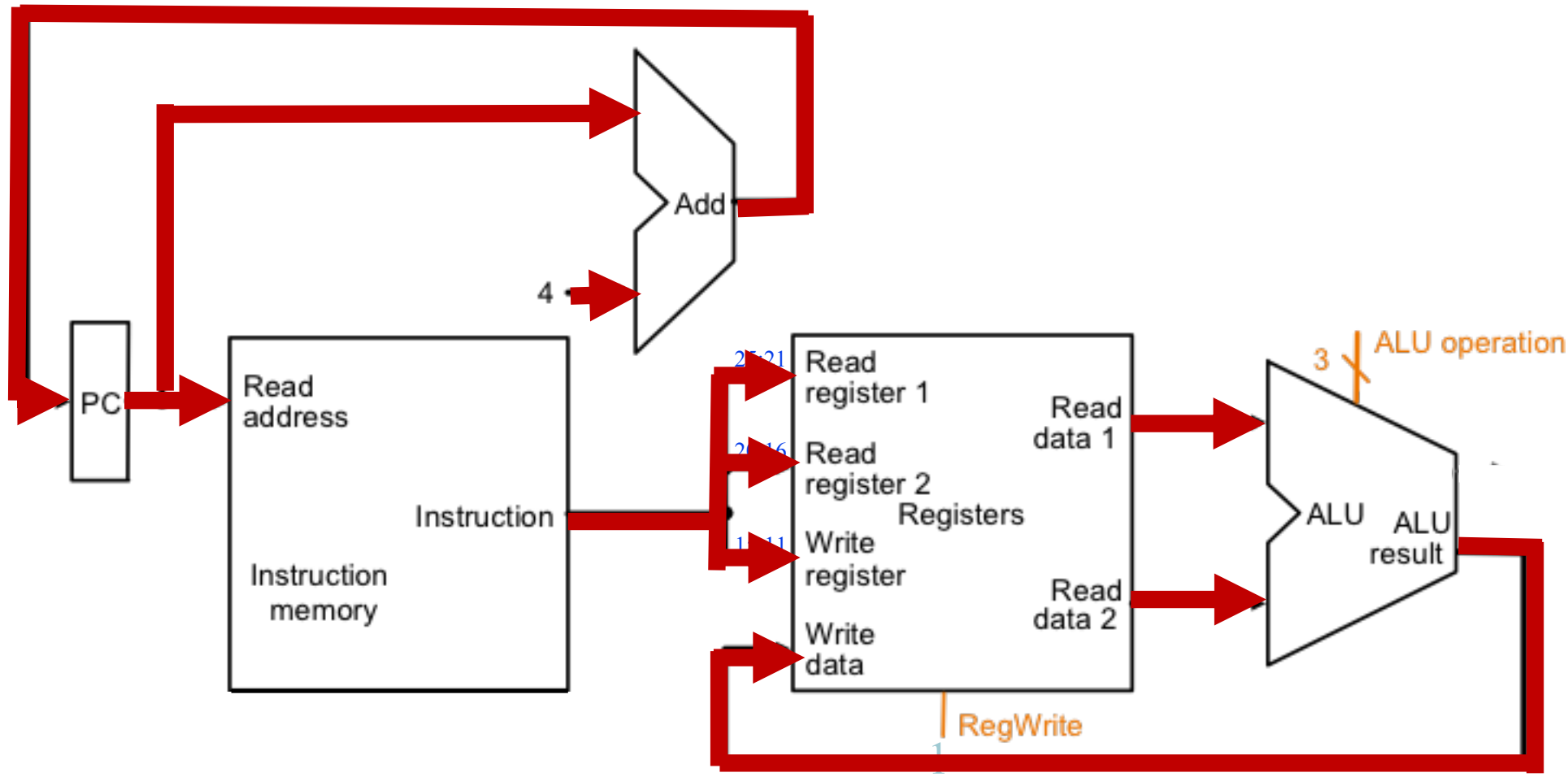
RTL：R[rd] ← R[rs] op R[rt]

PC ← PC + 4

IF	ID	EX	MEM	WB

Combinational state update logic

ALU Datapath



IF	ID	EX	MEM	WB

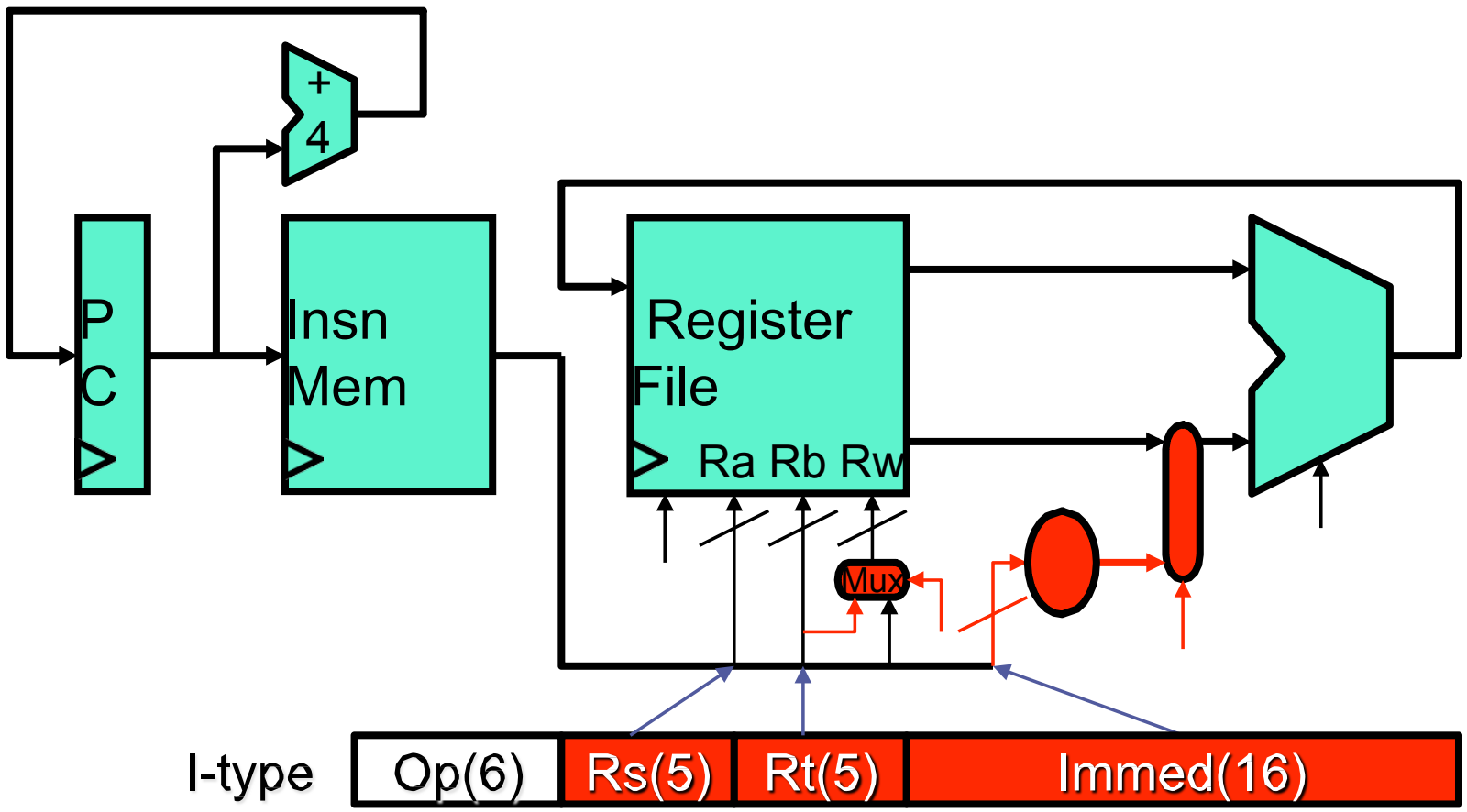
Combinational
state update logic

Verilog 的线选

- 如何获取指令里的各个字段? **Wire select**

```
wire   [31:0] inst;
wire   [5:0]  op = inst[31:26];
wire   [4:0]  rs = inst[25:21];
wire   [4:0]  rt = inst[20:16];
wire   [4:0]  rd = inst[15:11];
wire   [4:0]  sh = inst[10:6];
wire   [5:0]  func= inst[5:0];
```

I-type instruction addi



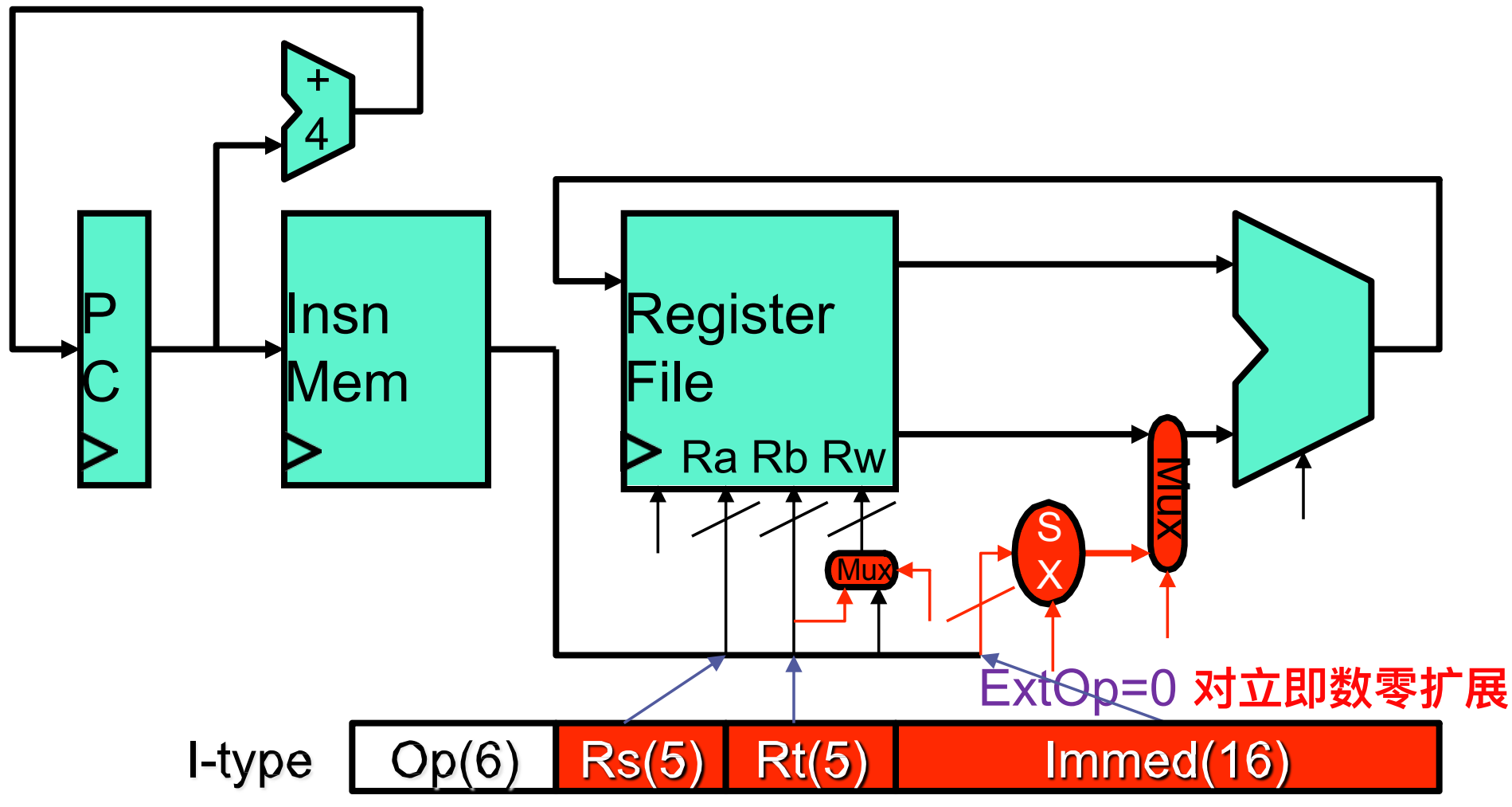
- 目的寄存器I-type 写入的是Rt, R-type 写入的是Rd 寄存器，增加一个2选一选择器。
- 增加符号扩展单元，ALU B端输入增加二选一选择器。

```
wire [31:0] insn;
wire [15:0] imm16 = insn[15:0];
wire [31:0] sximm16 = {{16{imm16[15]}}, imm16};
```

I-type instruction（ori）

- OR Immediate:
  - ori rt, rs, imm16

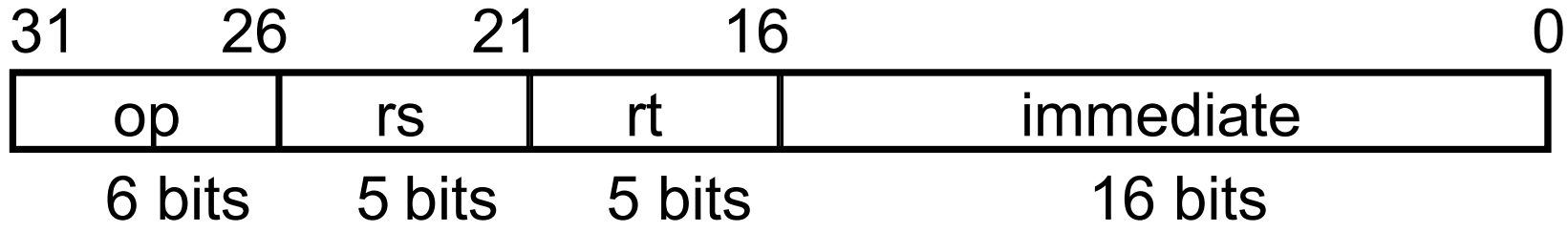
与addi 类似，不同的是对立即数进行零扩展，ALU操作或逻辑




```
wire [31:0] inst;
wire [15:0] imm16 = inst[15:0];
wire [31:0] sximm16 = {{16{1'b0}, imm16};
```

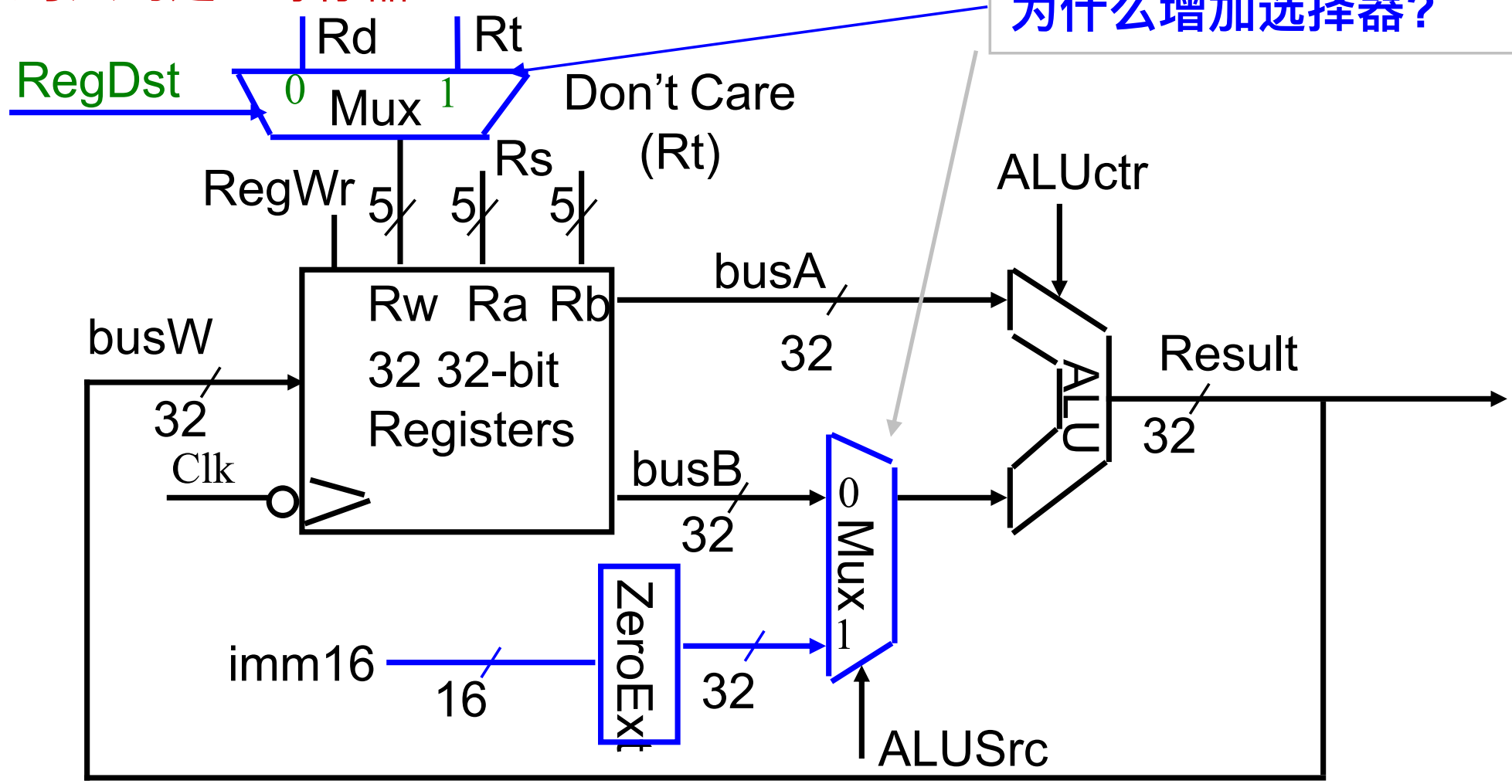
Datapath of OR Immediate Instruction

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



R-Type 指令的结果写入 Rd 寄存器，I-Type 指令结果写入的是Rt寄存器

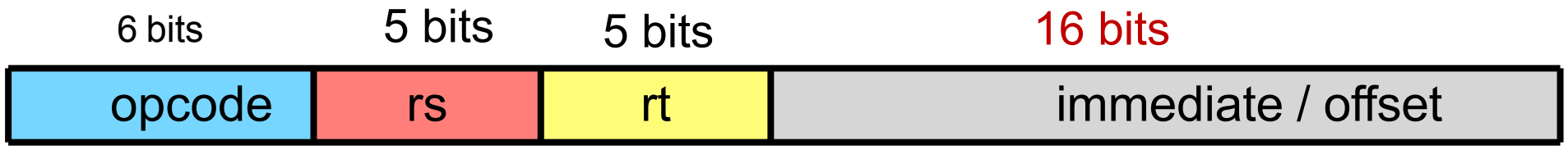
为什么增加选择器?



Ori control signals：RegDst=?；RegWr=1；ALUSrc=1；ALUctr=or

实现 load 指令

load 从数据存储器读取，写入寄存器堆

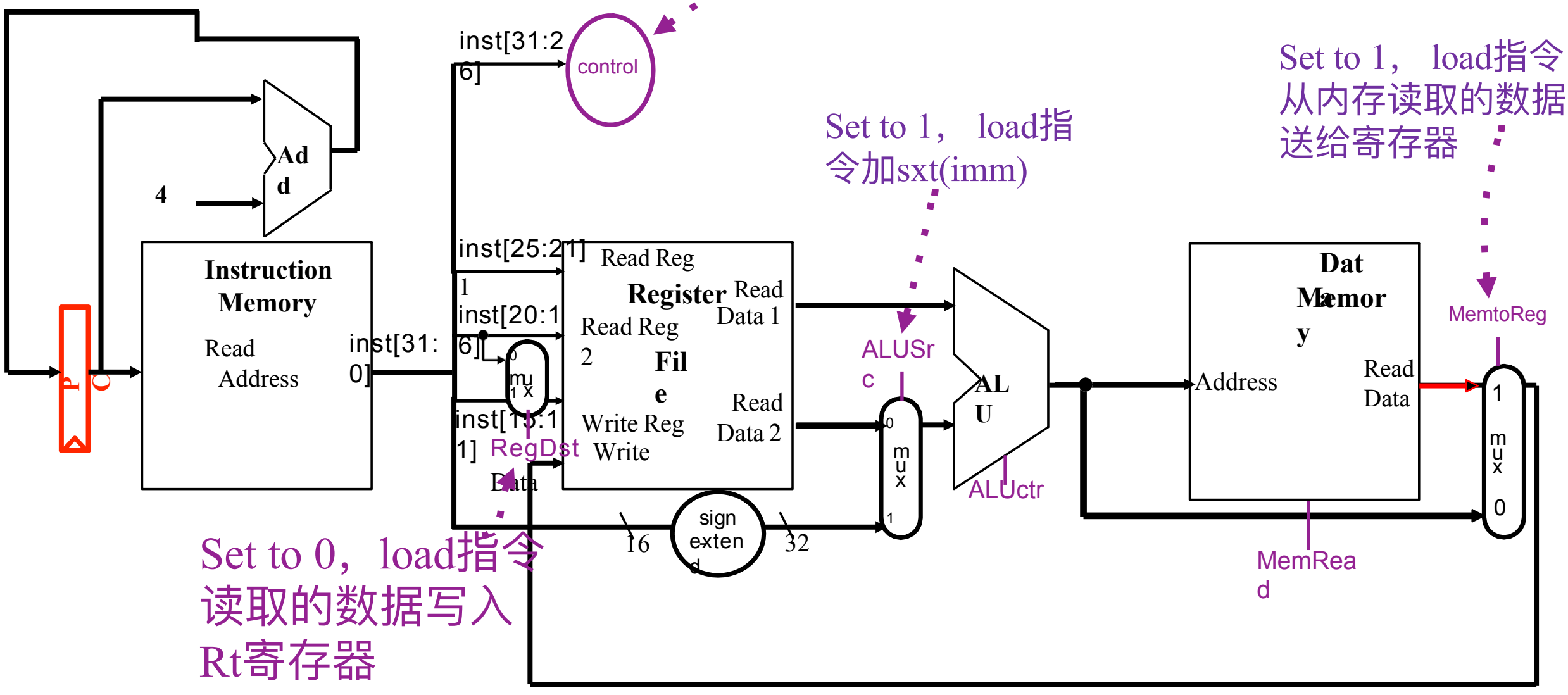


instruction = MEM[PC]

REG[rt] = MEM[signext(immediate) + REG[rs]]

PC = PC + 4

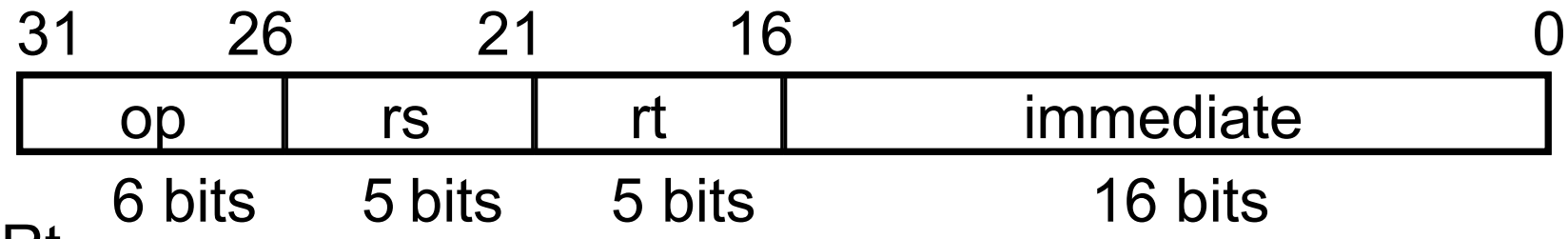
为不同类型的指令设置不同的控制信号

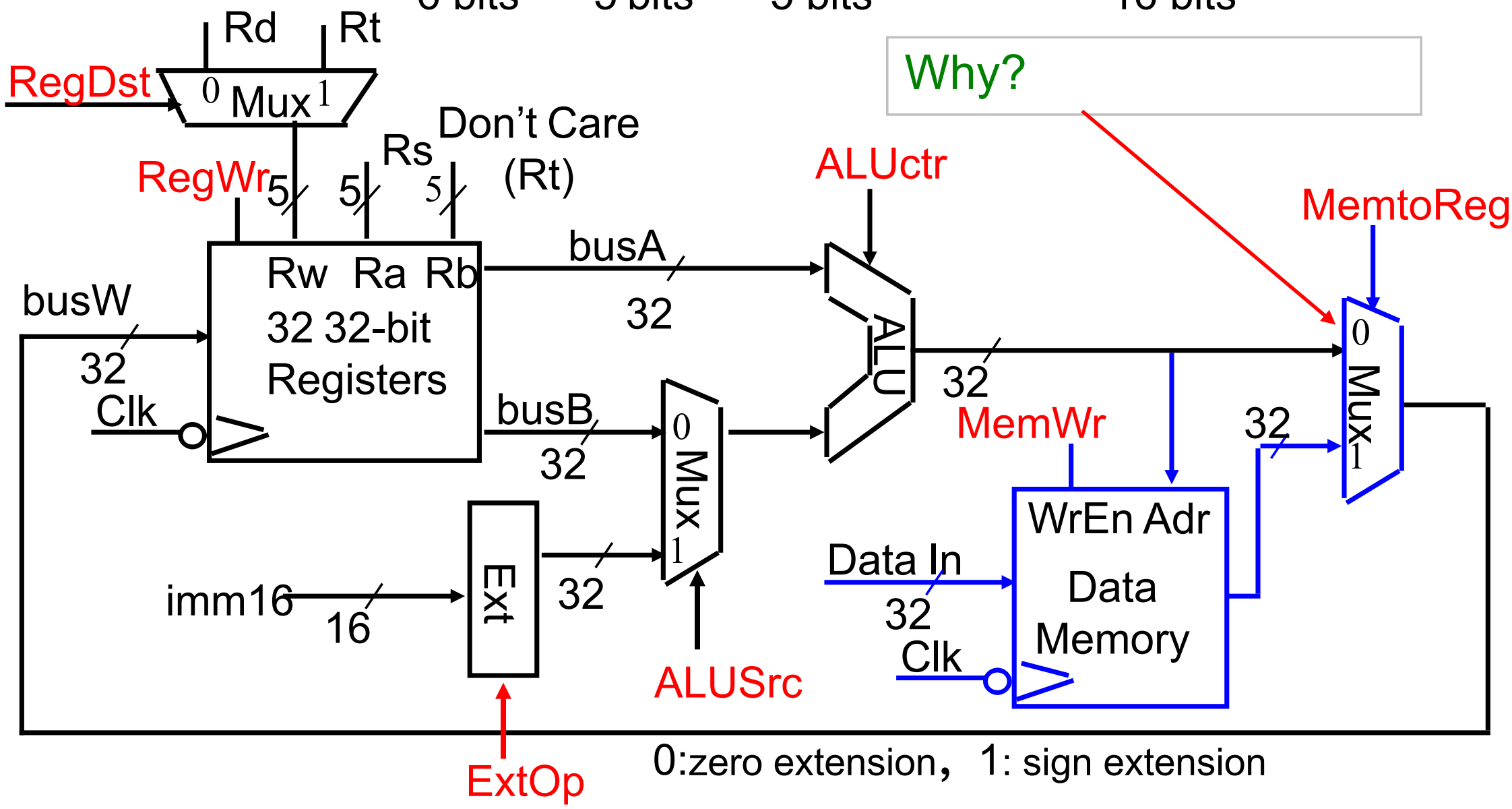


Set to 0，load指令读取的数据写入Rt寄存器

Data path for Load Instruction

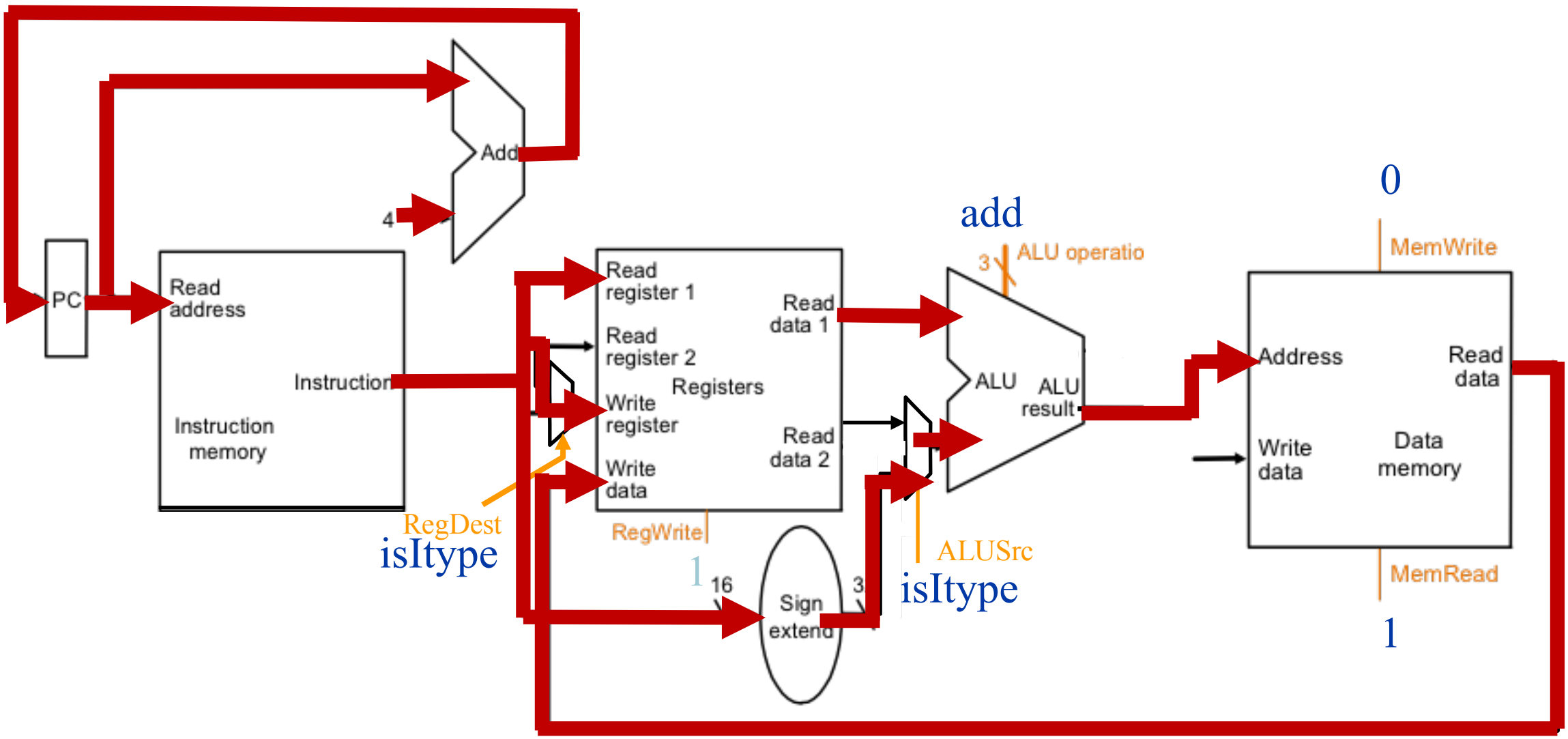
- R[rt] ← M[R[rs] + SignExt[imm16]] Example: lw rt, rs, imm16





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

LW Datapath



if MEM[PC]==LW rt offset$_{16}$ (base)

EA = sign-extend(offset) + GPR[base]

R[rt] ← M[R[rs] + SignExt[imm16]]

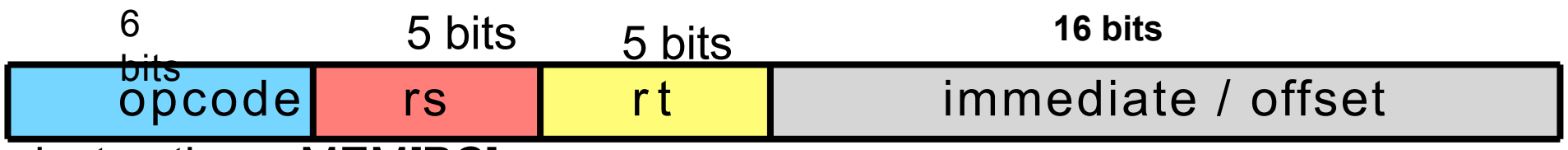
PC ← PC + 4

IF	ID	EX	MEM	WB

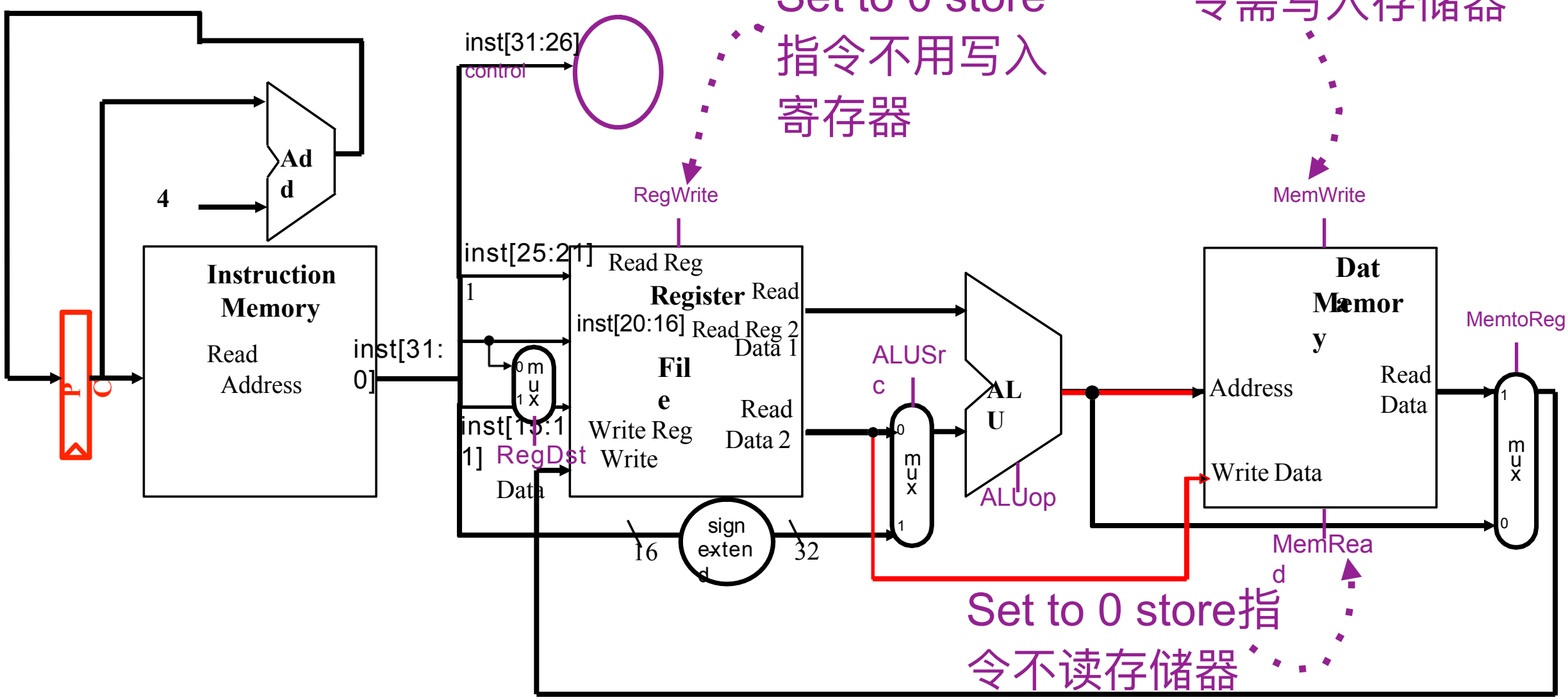
Combinational state update logic

实现 store 指令

Store 把在译码阶段从寄存器文件中读取的数据写入数据存储器

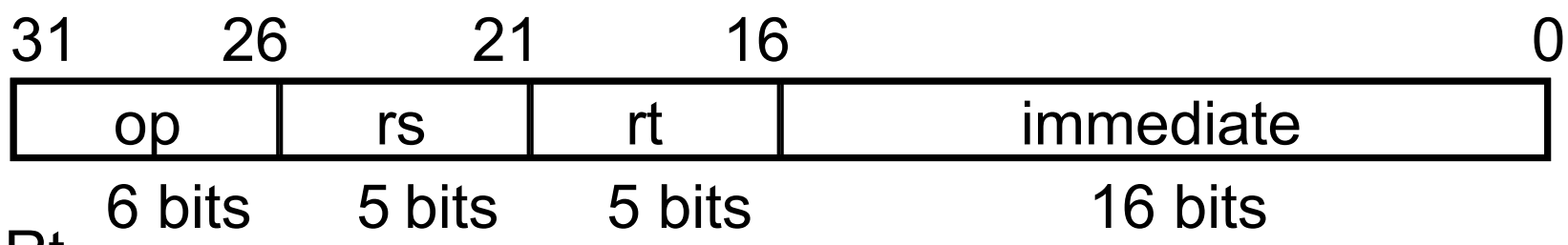


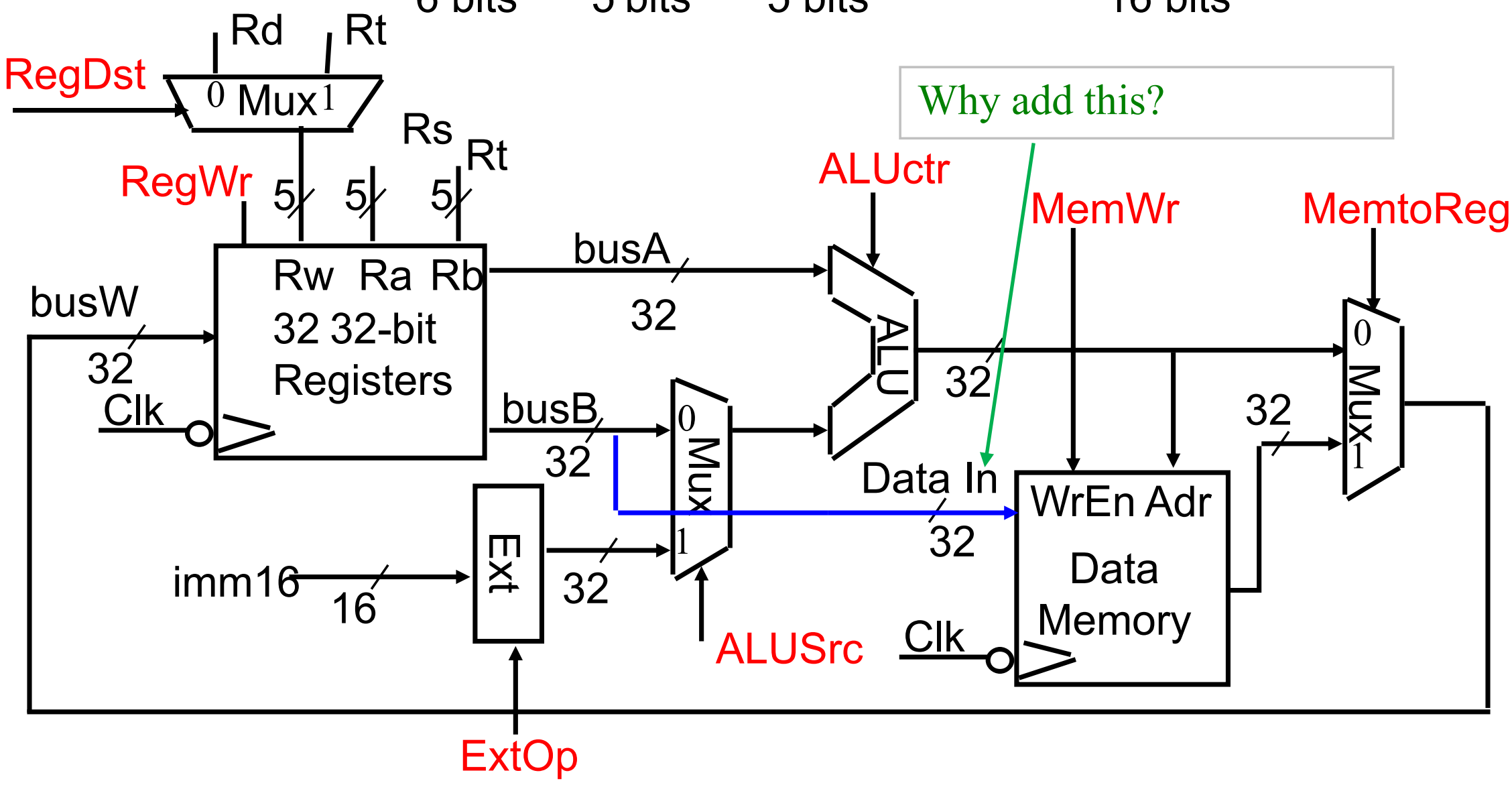
instruction = MEM[PC]
MEM[signext(immediate) + REG[rs]] =
REG[rt] PC = PC + 4



Data path for SW

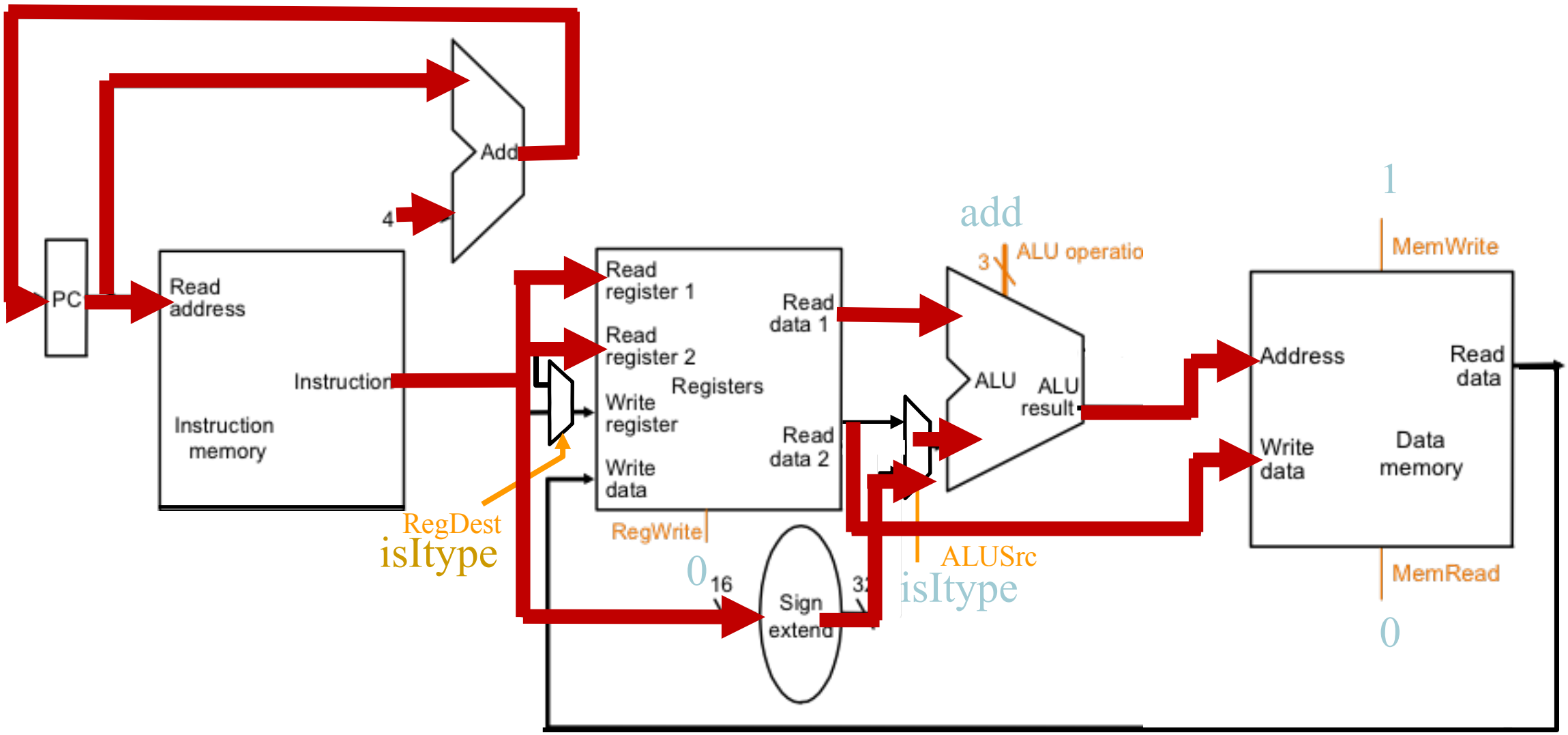
- M[R[rs] + SignExt[imm16] ← R[rt]] Example: sw rt, rs, imm16





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

SW Datapath



if MEM[PC]==SW rt offset$_{16}$ (base)

EA = sign-extend(offset) + GPR[base]

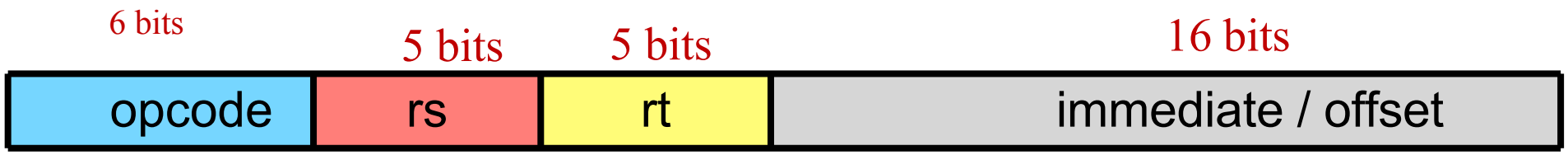
M[R[rs] + SignExt[imm16] ← R[rt]]

PC ← PC + 4

IF	ID	EX	MEM	WB

Combinational state update logic

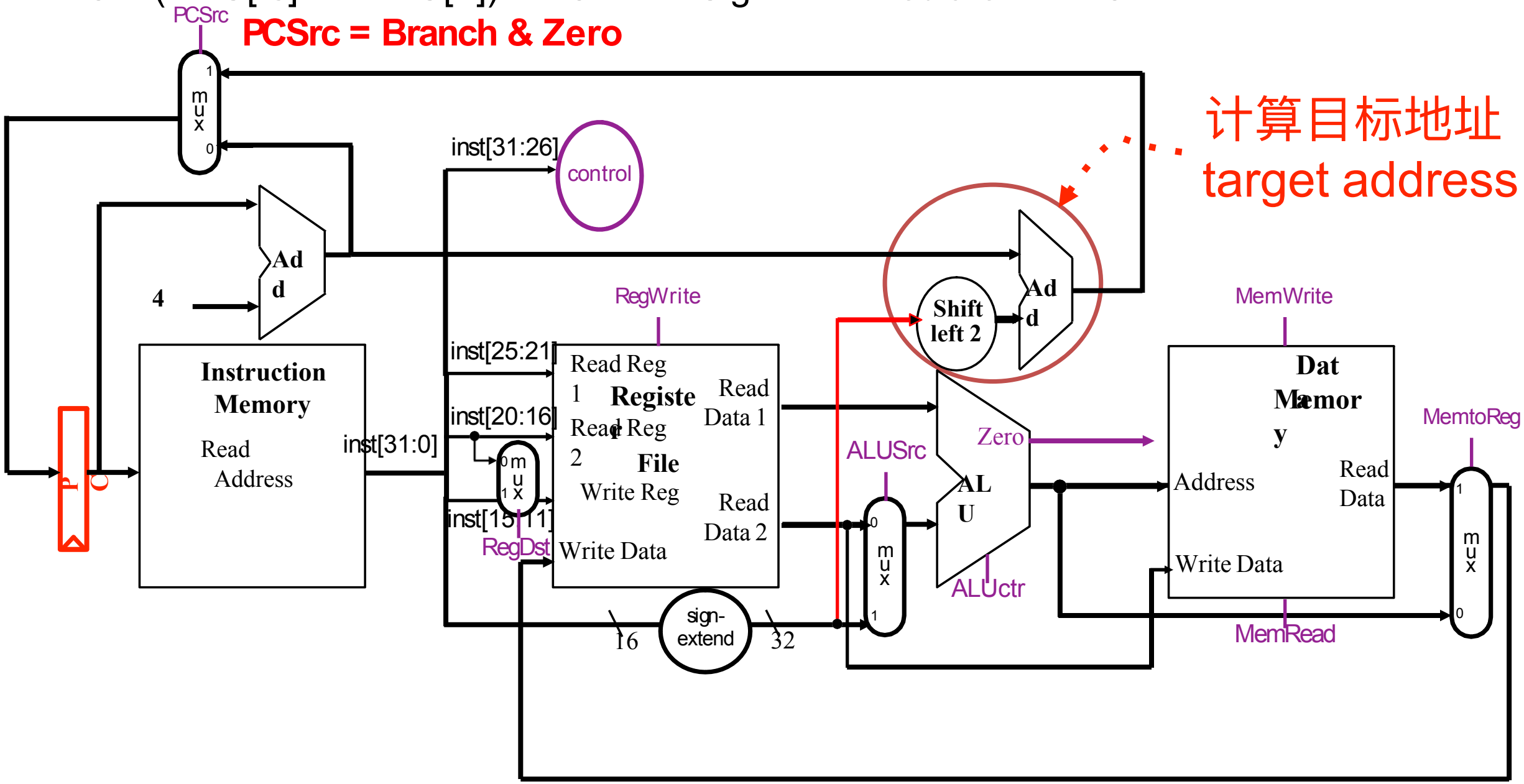
实现 branch 指令



instruction = MEM[PC]

PC = (REG[rs] == REG[rt]) ? PC + 4 + SignExtImmediate *4 : PC + 4

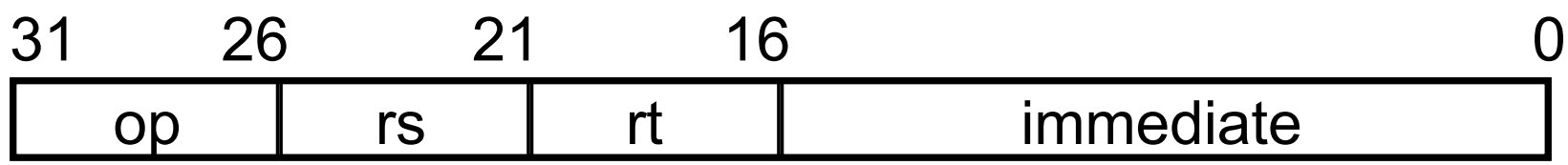
PCSrc = Branch & Zero

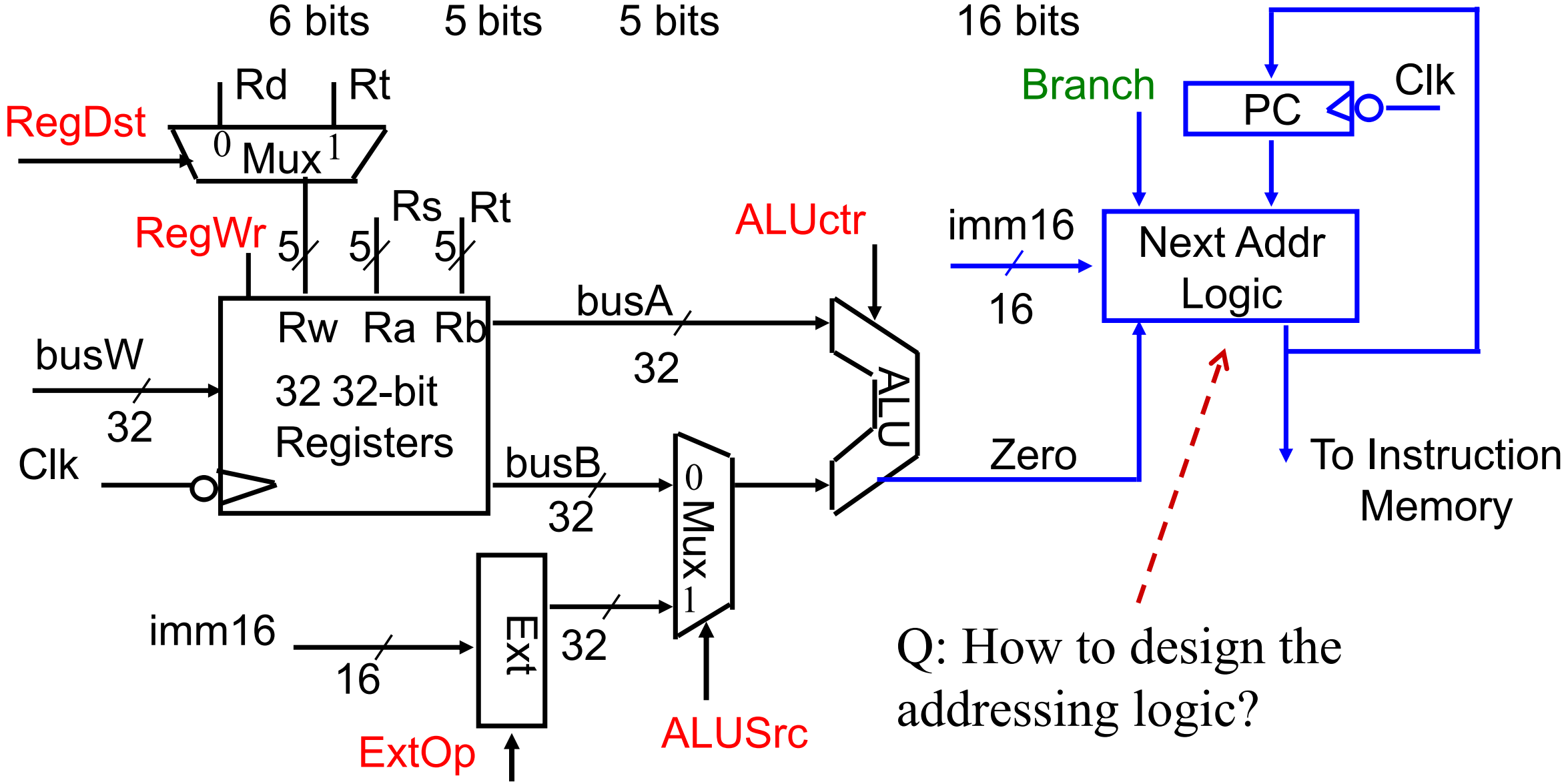


Data path for beq

- beq rs, rt, imm16

需要判断Rs-Rt=0？！

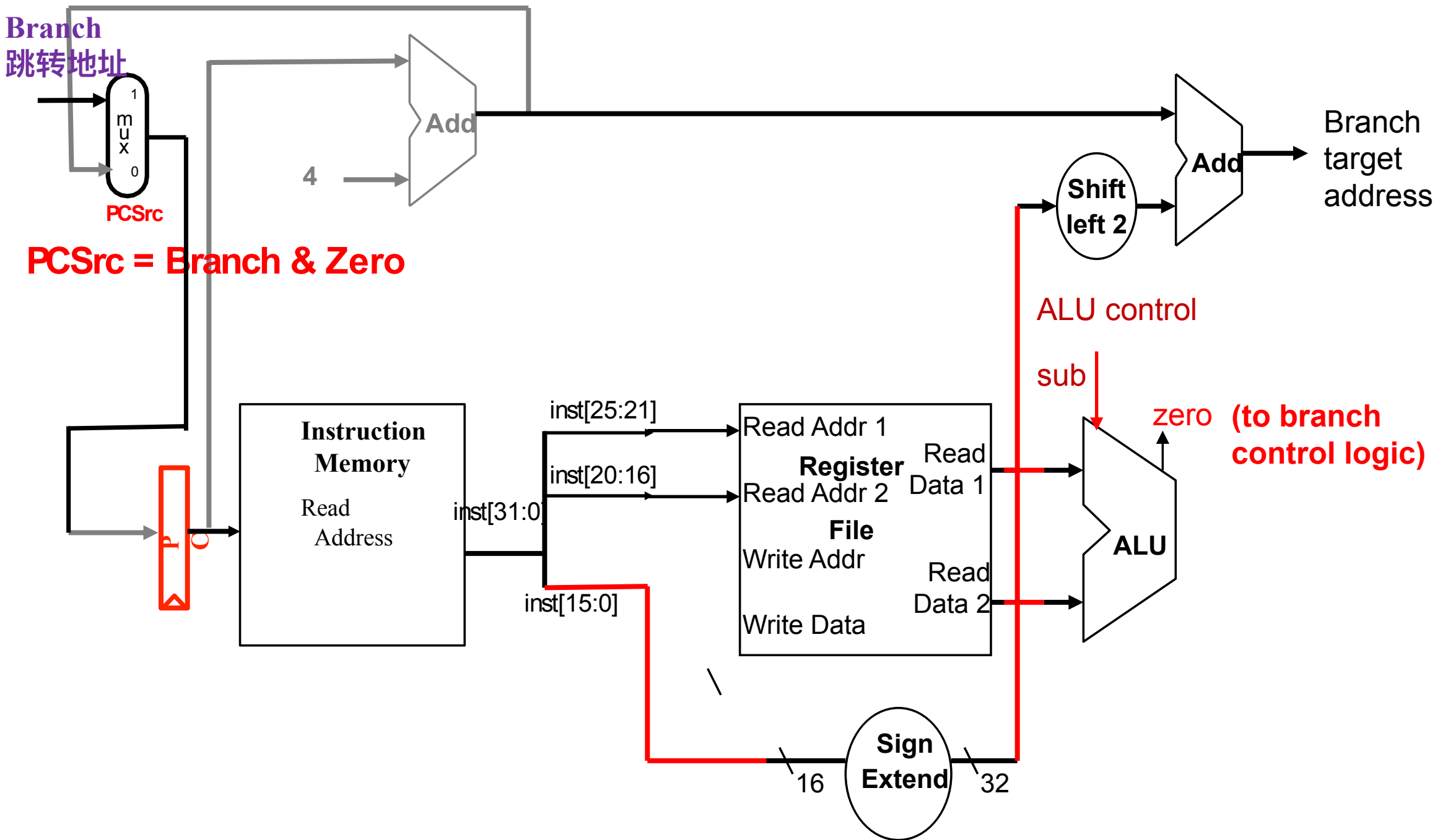




RegDst=x, RegWr=0, ALUctr=sub, ExtOp=x, ALUSrc=0, MemWr=0, MemtoReg=x, Branch=1

Executing Branch Operations

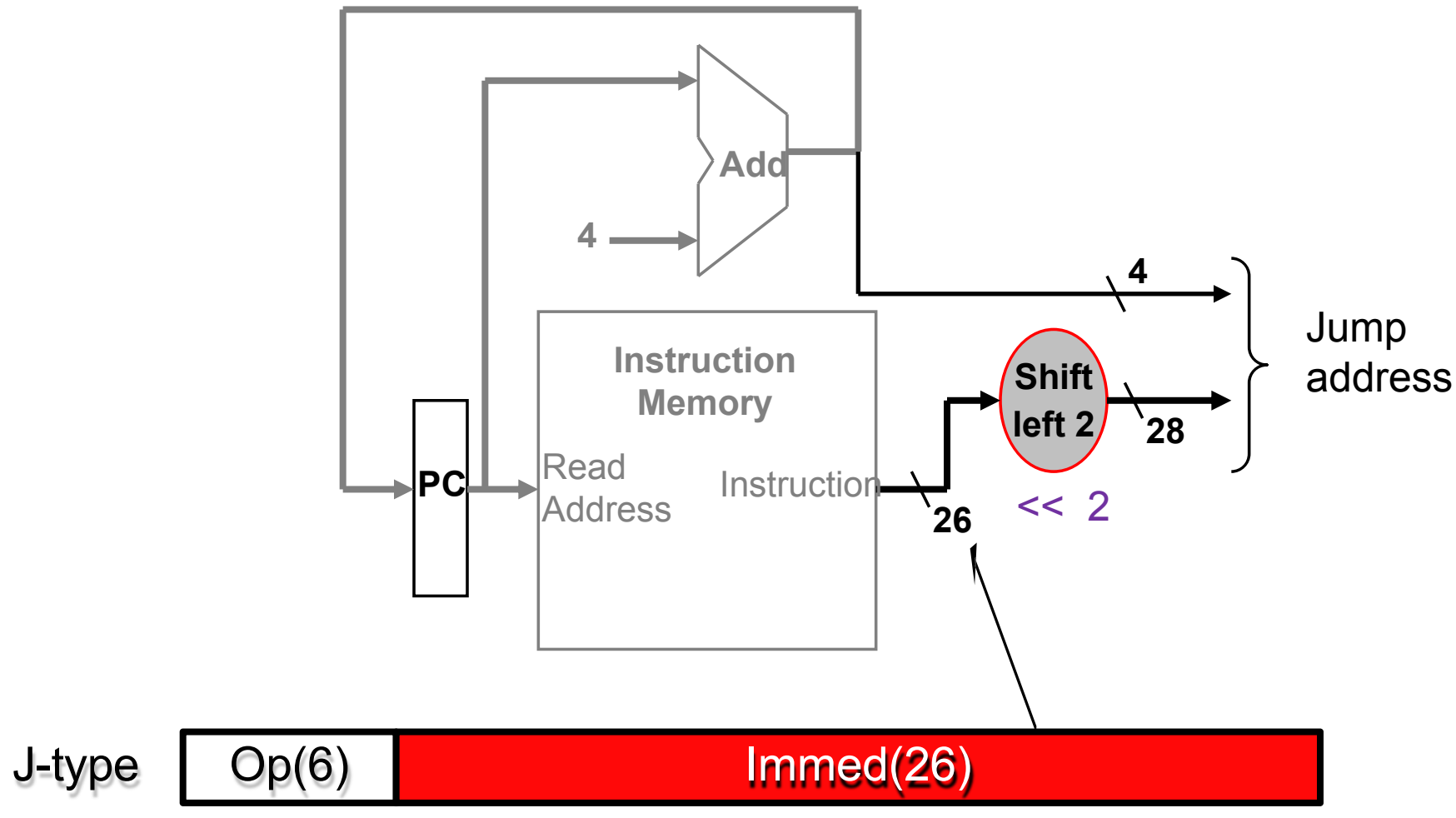
- Branch 操作：
  - 比较译码阶段从寄存器堆读取的两个操作数是否相等（ALU零输出）
  - 计算分支目标地址，(PC)+4+SignEXT(imm16<<2)



Jump operation

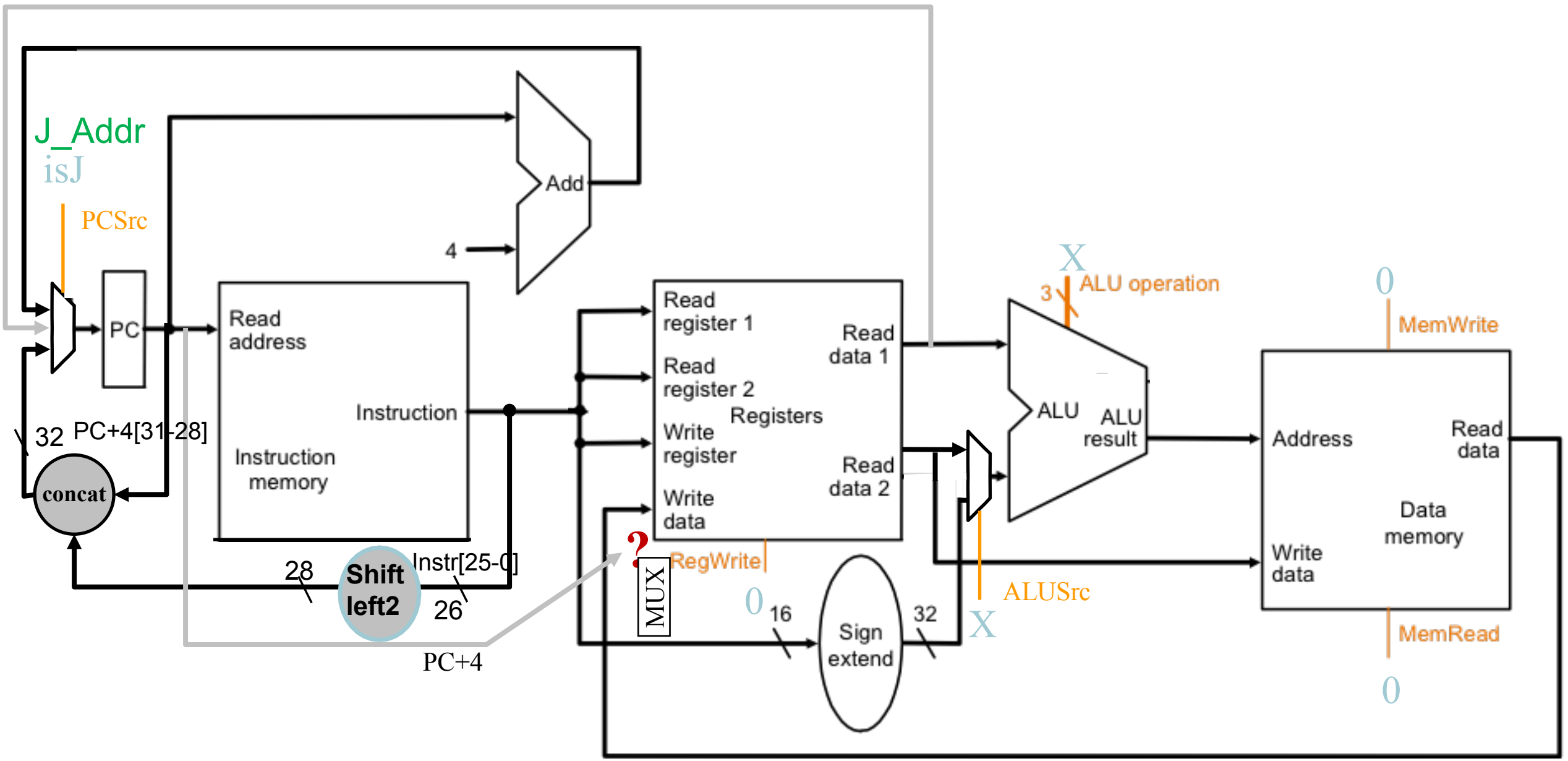
- JUMP:
  - j target

把取出指令的低 26 位左移 2 位后替换 PC 的低 28 位




```
wire    [31:0]    insn;
wire    [25:0]    imm26 = insn[25:0]
wire    [31:0]    imm26_shifted_by_2 = {4'b0000, imm26, 2'b00};
```

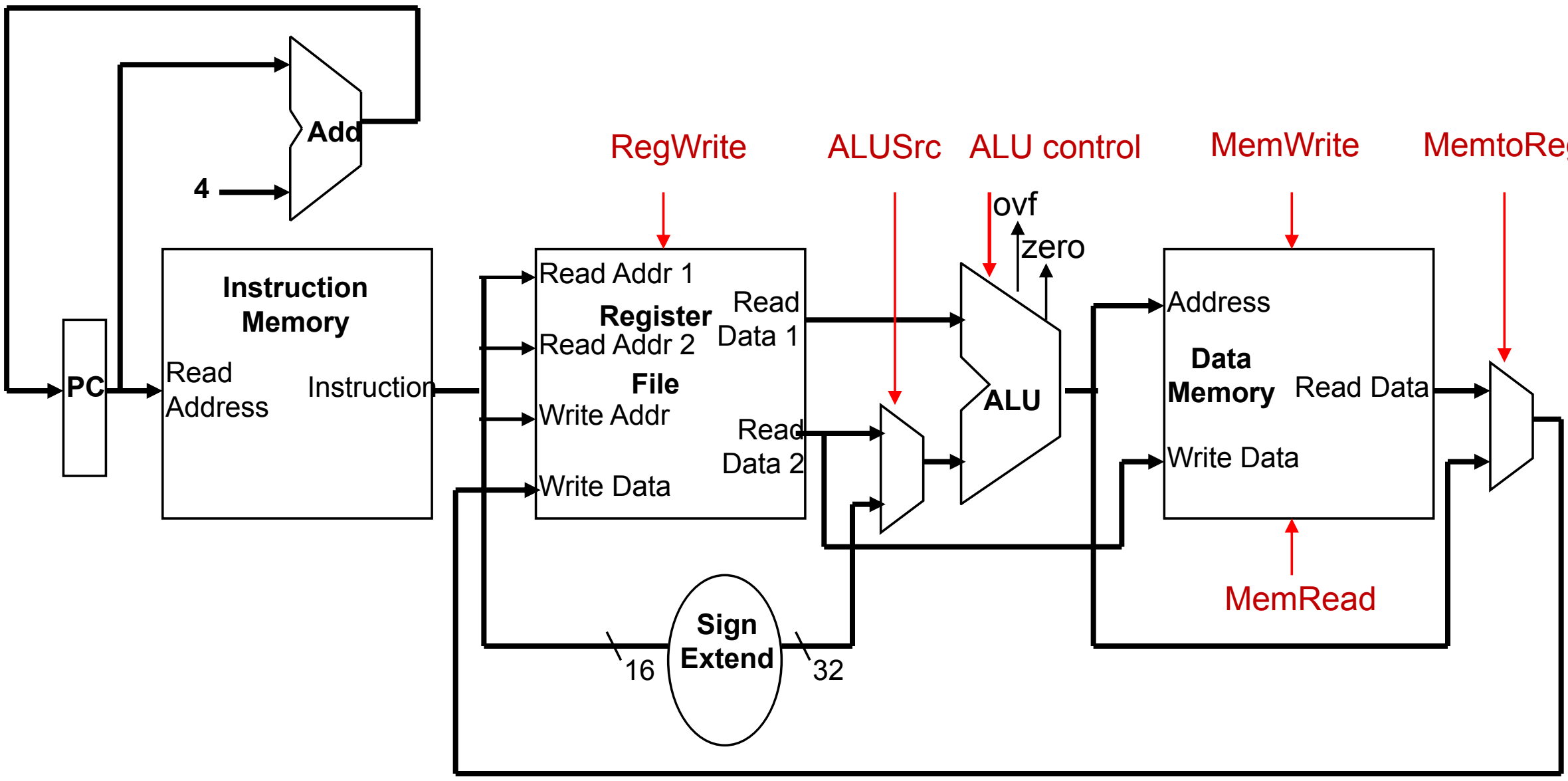
Unconditional Jump Datapath



if MEM[PC]==J immediate26

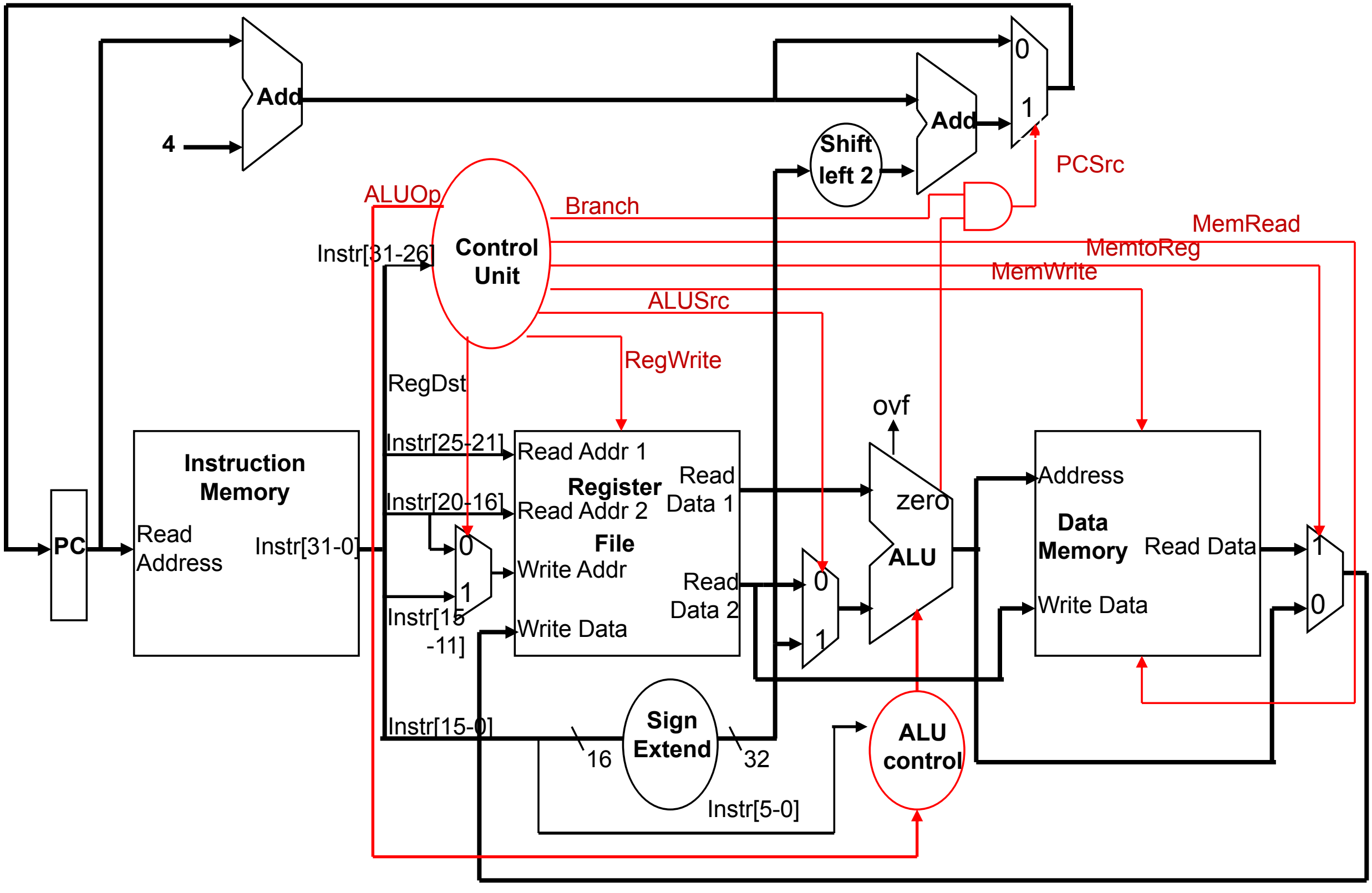
PC = { PC[31:28], immediate26, 2'b00 }

Fetch, R, and Memory Access Portions

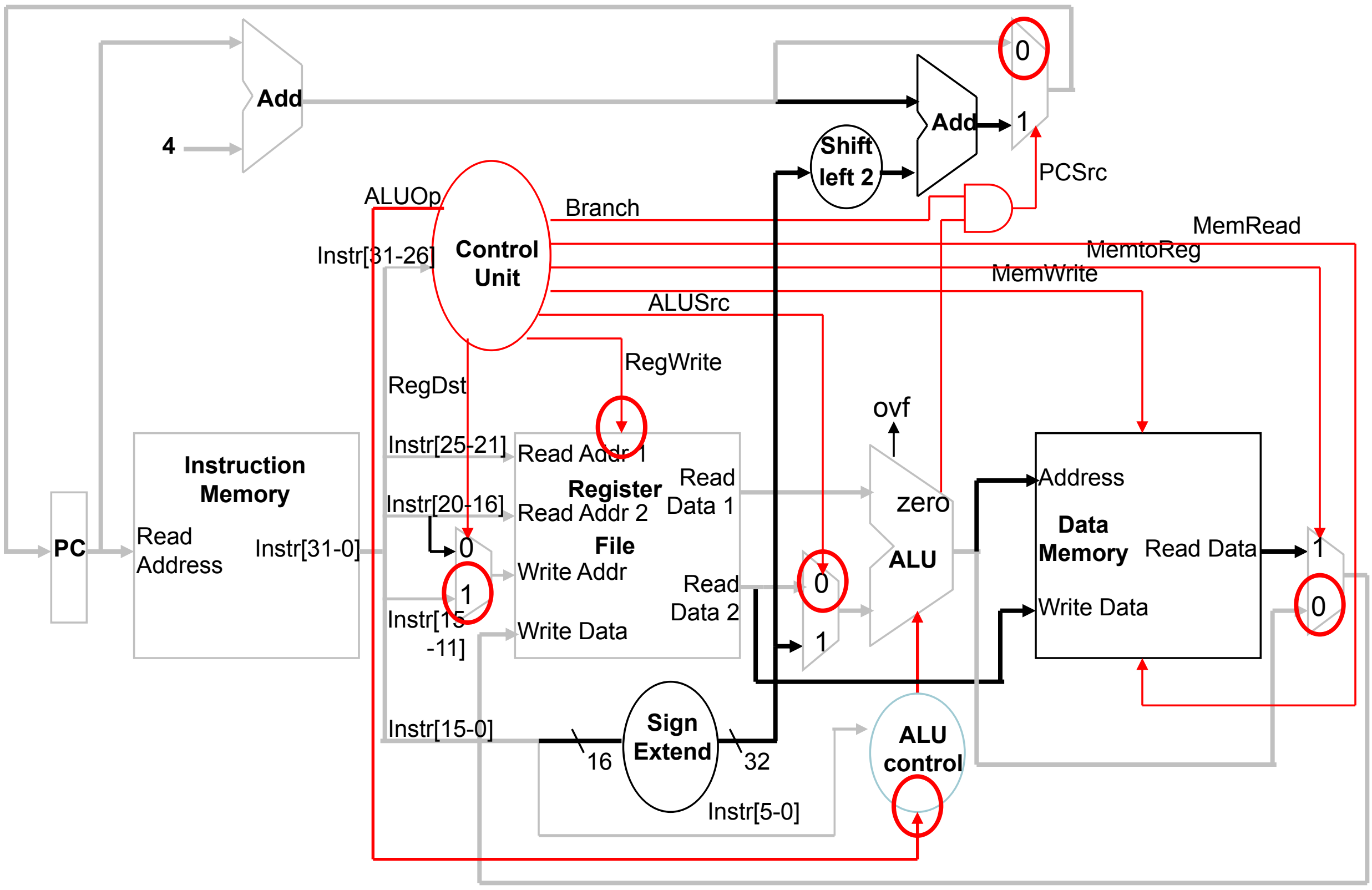


Now, let's put them together!

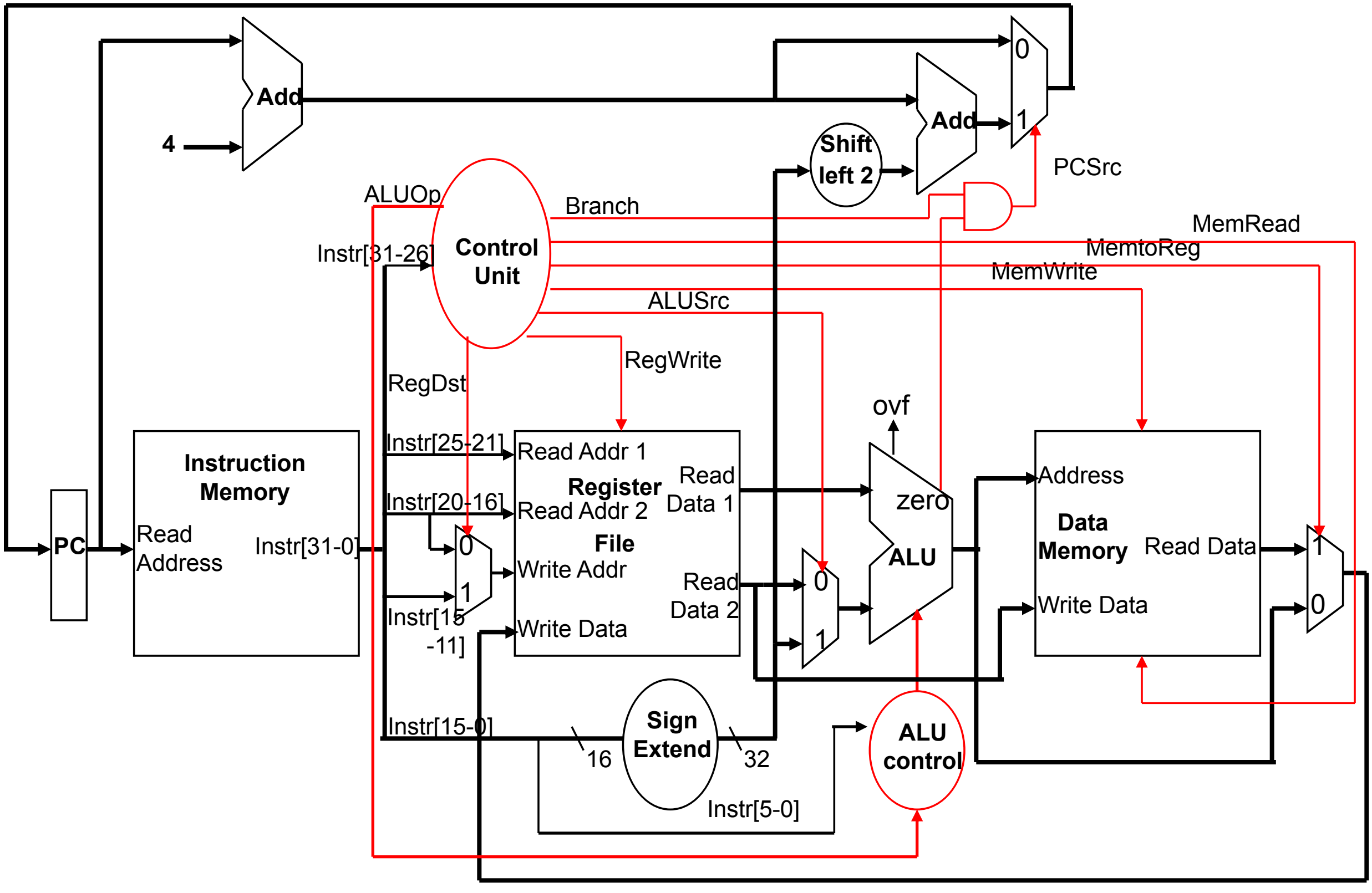
Single Cycle Datapath with Control Unit



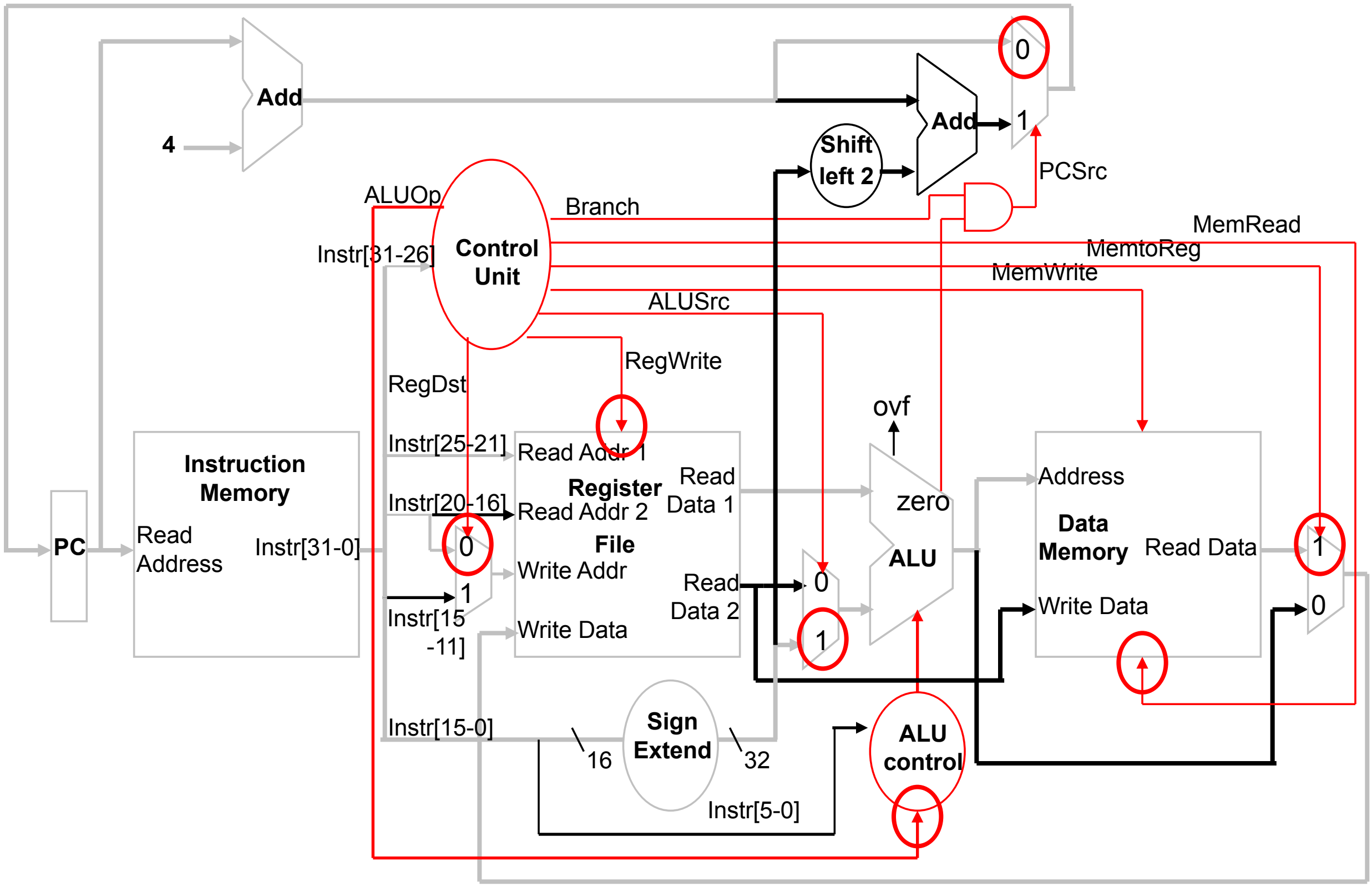
R-type Instruction Data/Control Flow



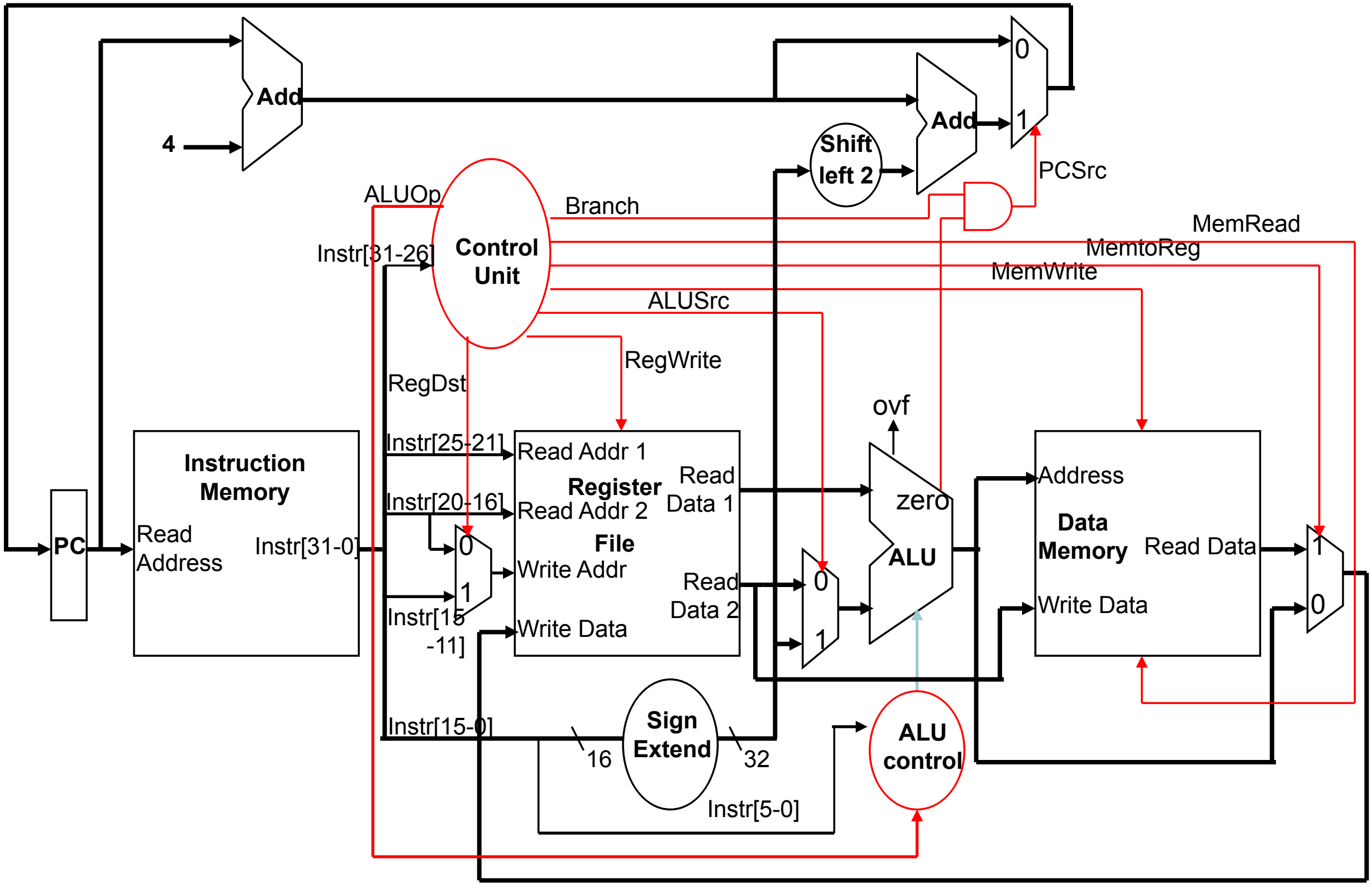
Load Word Instruction Data/Control Flow



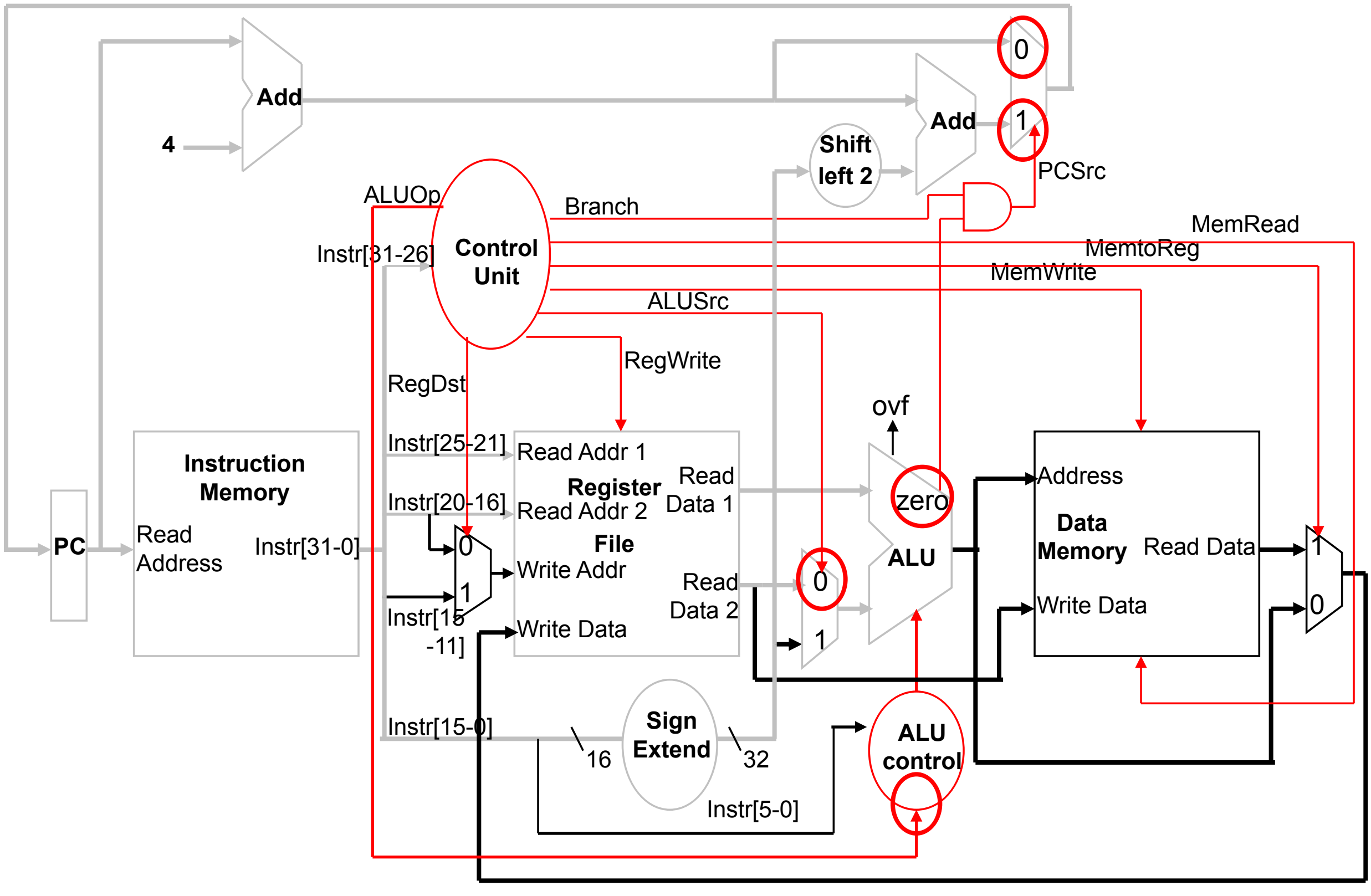
Load Word Instruction Data/Control Flow



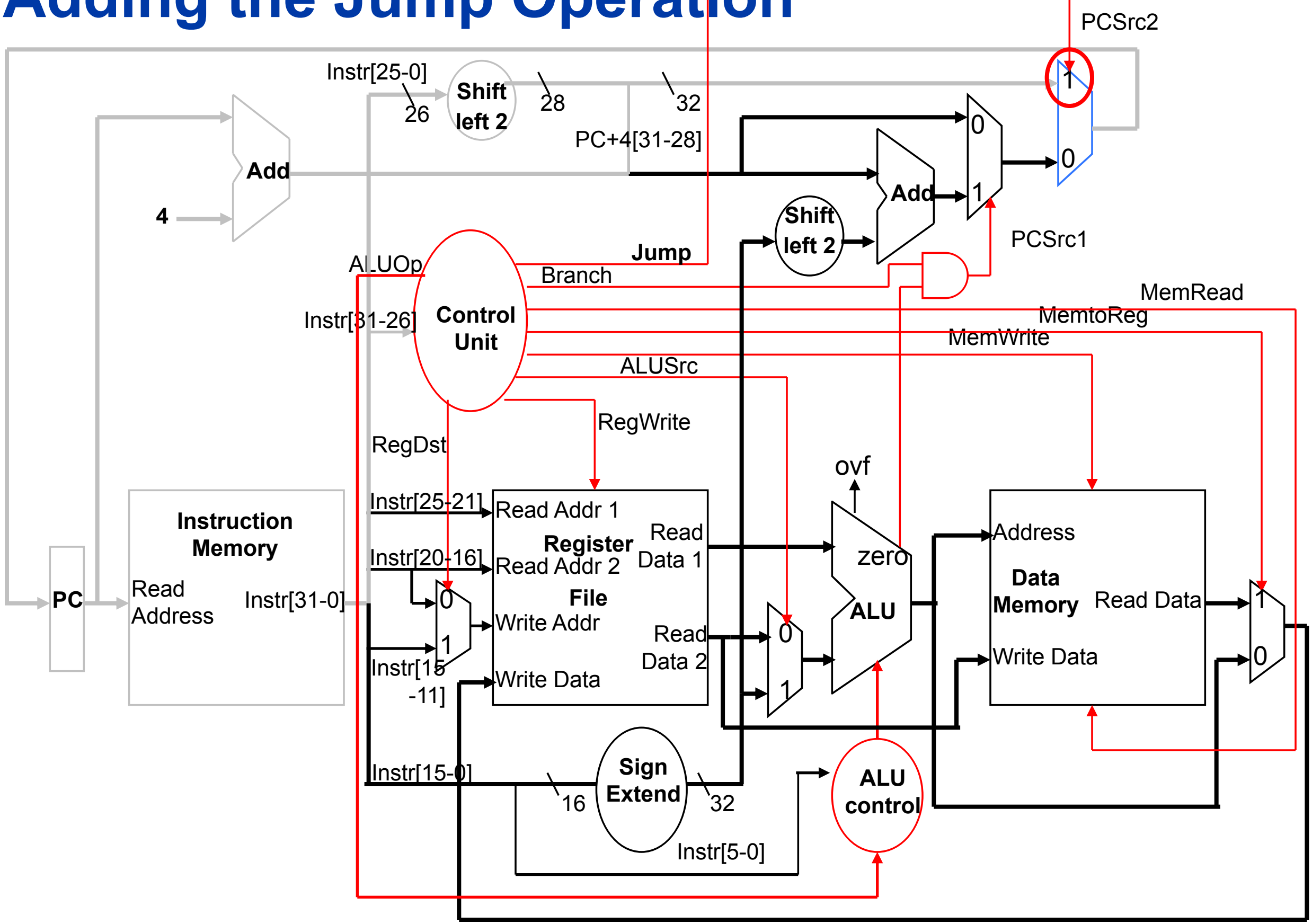
Branch Instruction Data/Control Flow



Branch Instruction Data/Control Flow



Adding the Jump Operation



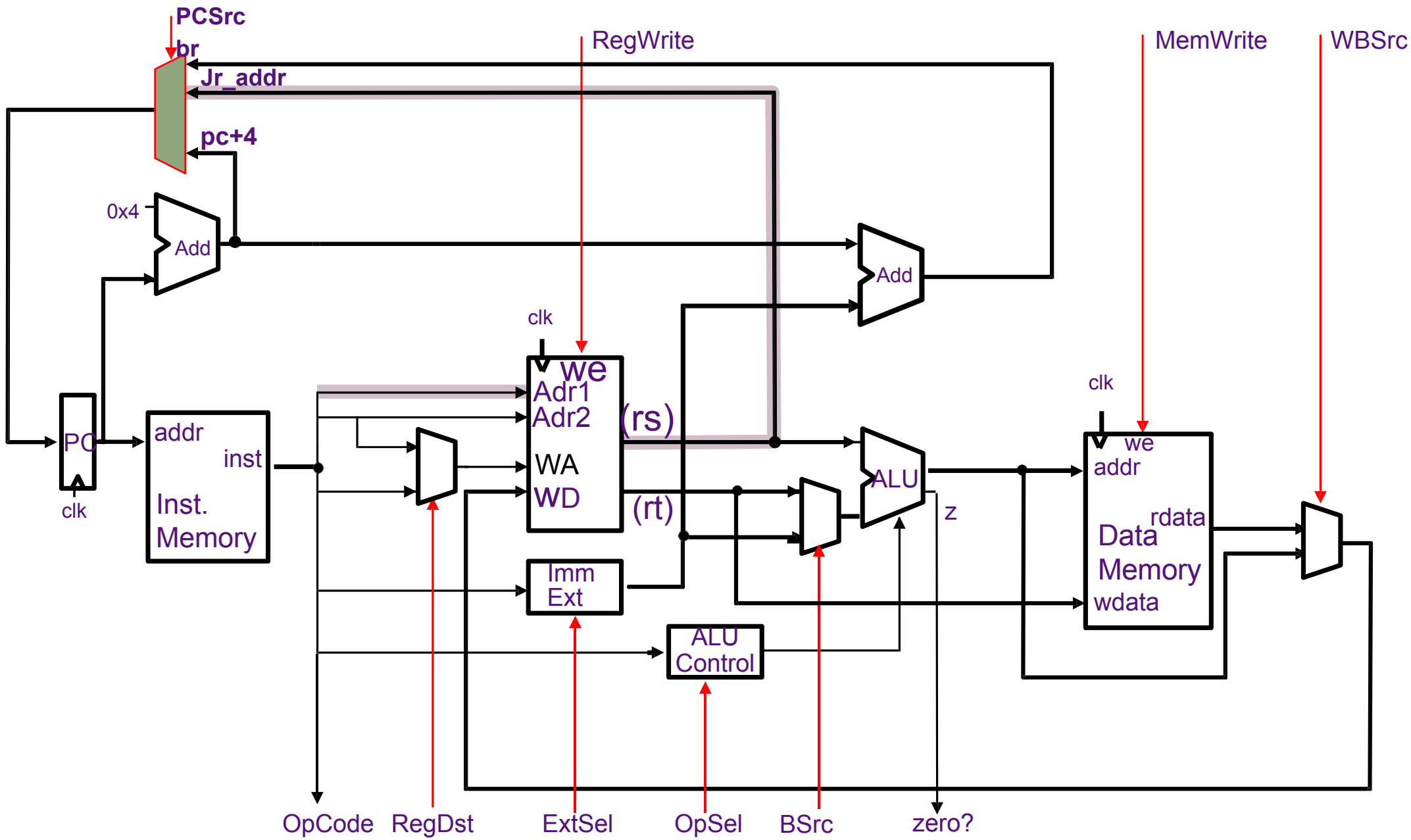
MIPS R2000 instruction set.

- **Shift Instructions**
- **J/JAL**
- **JR/JRAL**
- **slti/sltiu**
- **LUI**
- **Reset, Interrupts, and Exceptions**

前面的设计没有包括 JR, JAL, JALR, Shift , slti/sltiu …这些指令?

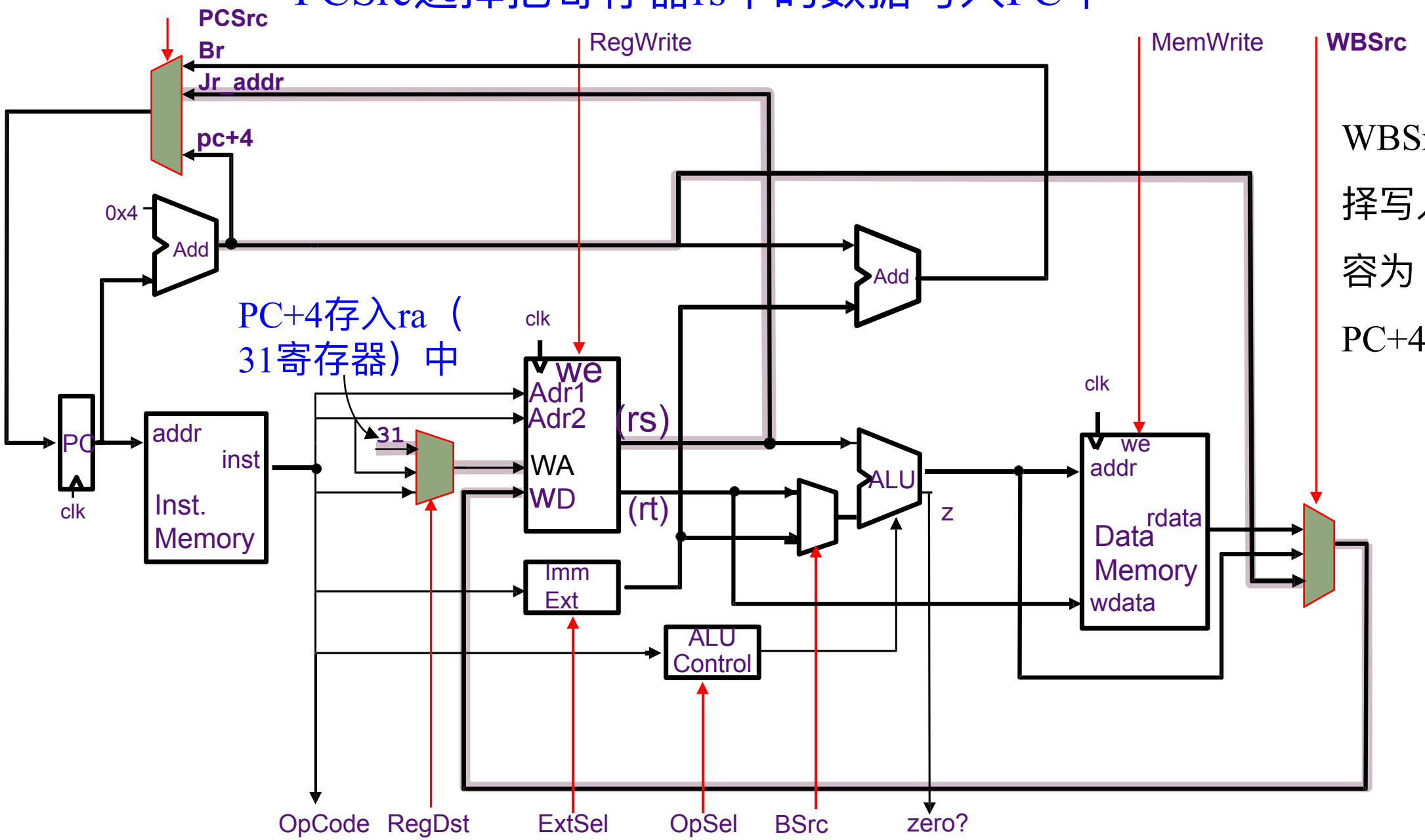
寄存器间接跳转指令 JR
（Register-Indirect Jumps）

jr rs　PCSrc选择把寄存器rs中的数据写入PC中



Register-Indirect Jump-&-Link (JALR)

PCSrc选择把寄存器rs中的数据写入PC中

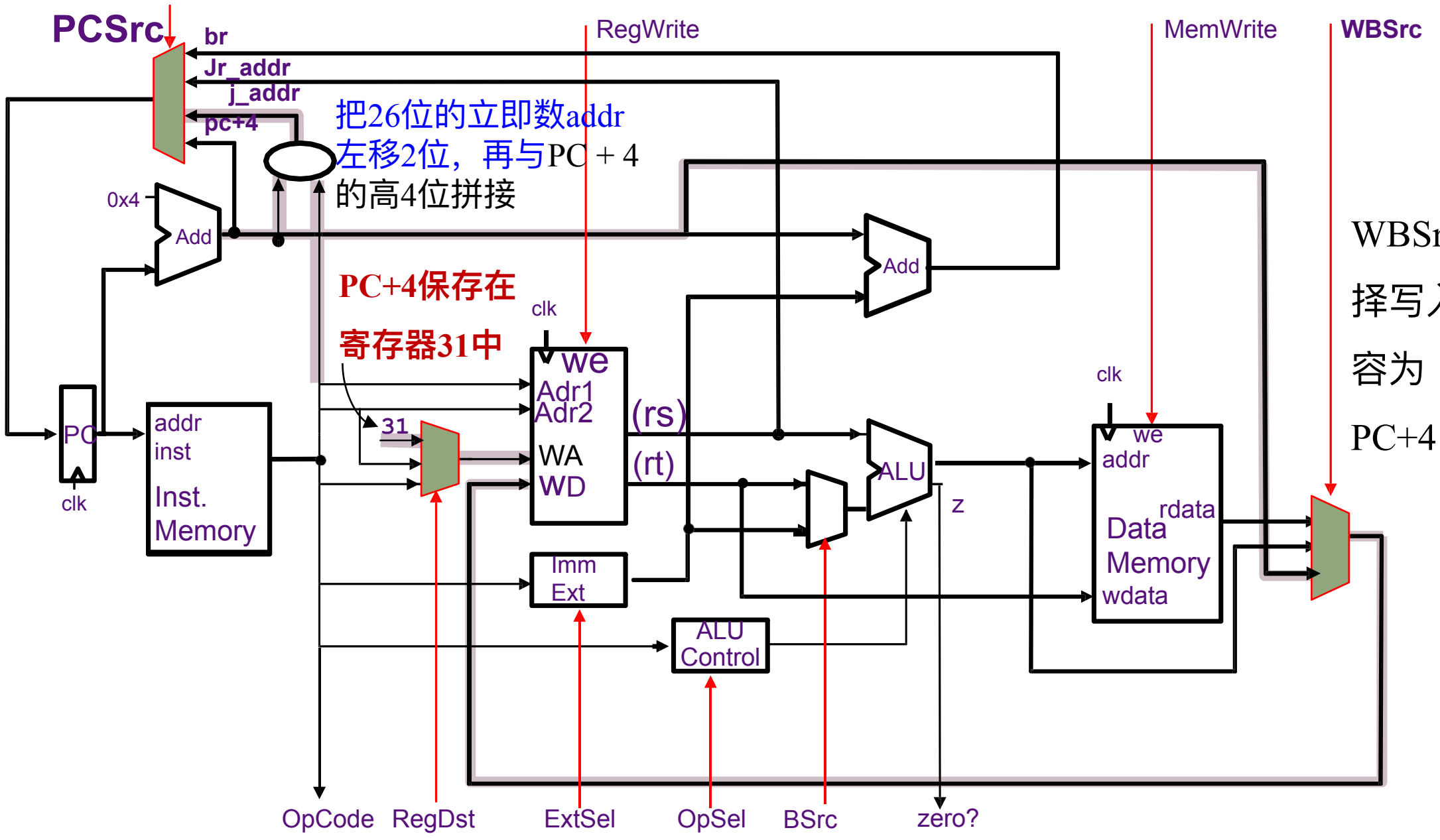


WBSrc选择写入内容为PC+4

RegDst 选择写入地址31

Absolute Jumps (J, JAL)

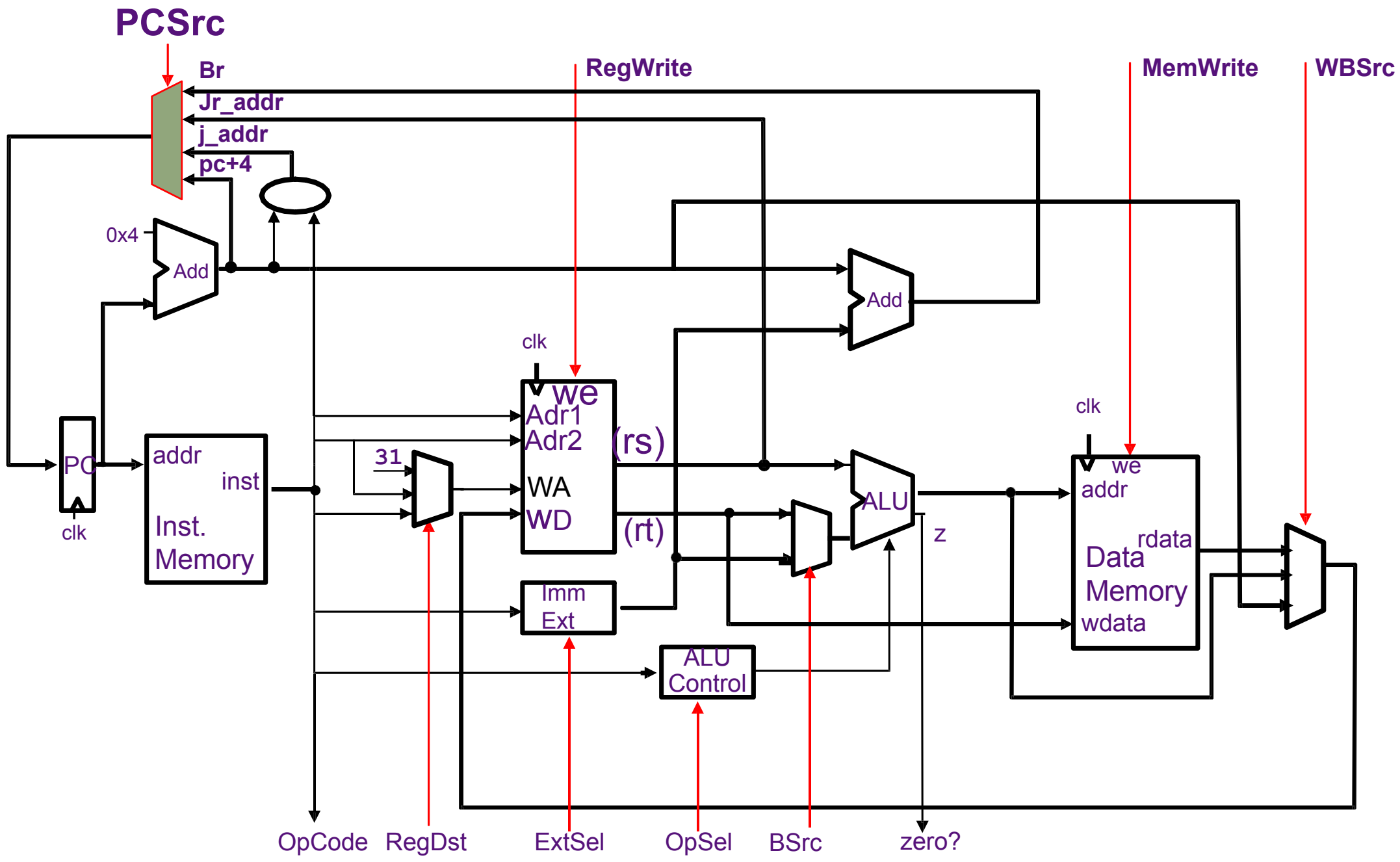
PCSrc选择把拼接后的地址写入PC中



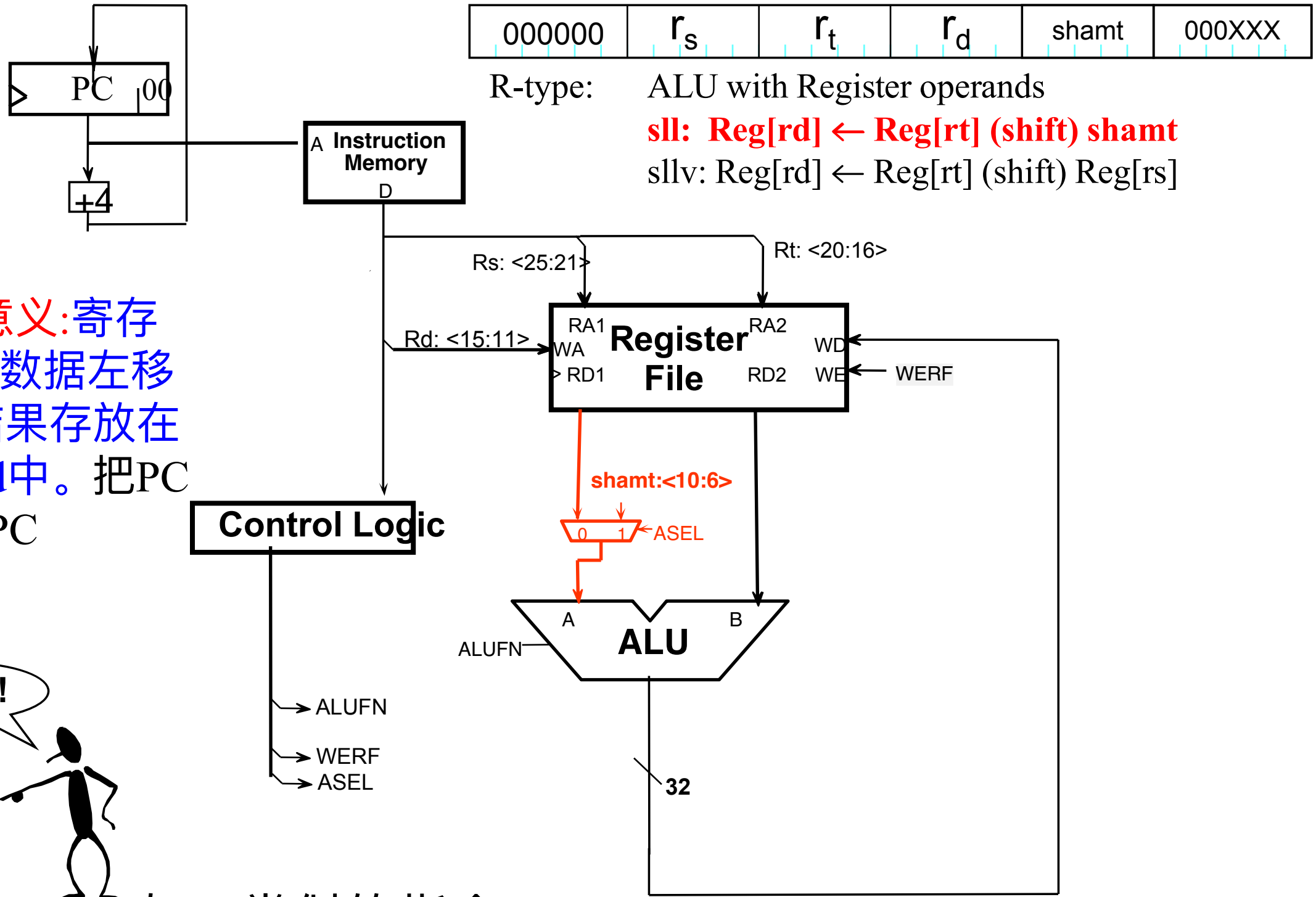
WBSrc选择写入内容为PC+4

RegDst 选择写入地址31

Harvard-Style Datapath for MIPS



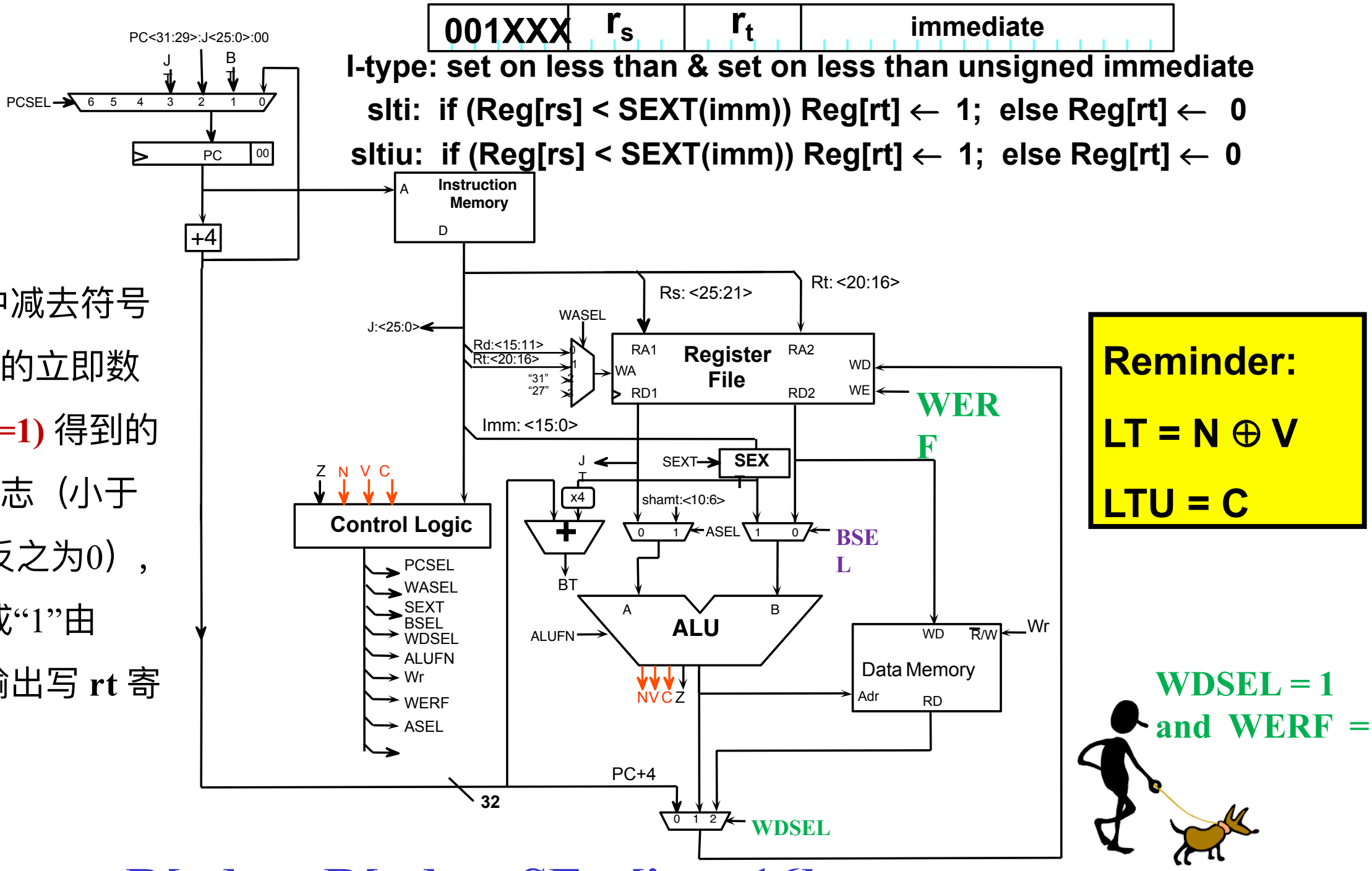
Shift Instructions



指令的意义:寄存器rt中的数据左移sa位，结果存放在寄存器rd中。把PC+4写入PC

与sll类似的指令：srl, sra

Loose Ends 有立即数的操作slti/sltiu



从 rs 中减去符号扩展后的立即数 **(BSEL=1)** 得到的条件标志（小于为1，反之为0），将“0”或“1”由 ALU 输出写 **rt** 寄存器

rd?

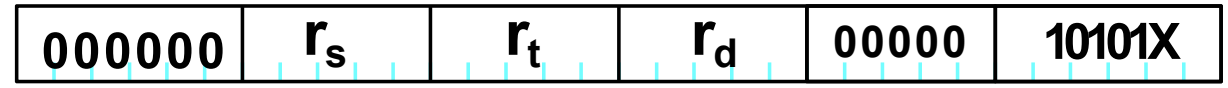
Reminder:

LT = N ⊕ V

LTU = C

- **R[rt] <= R[rs] op SExt[imm16]**

More Loose Ends

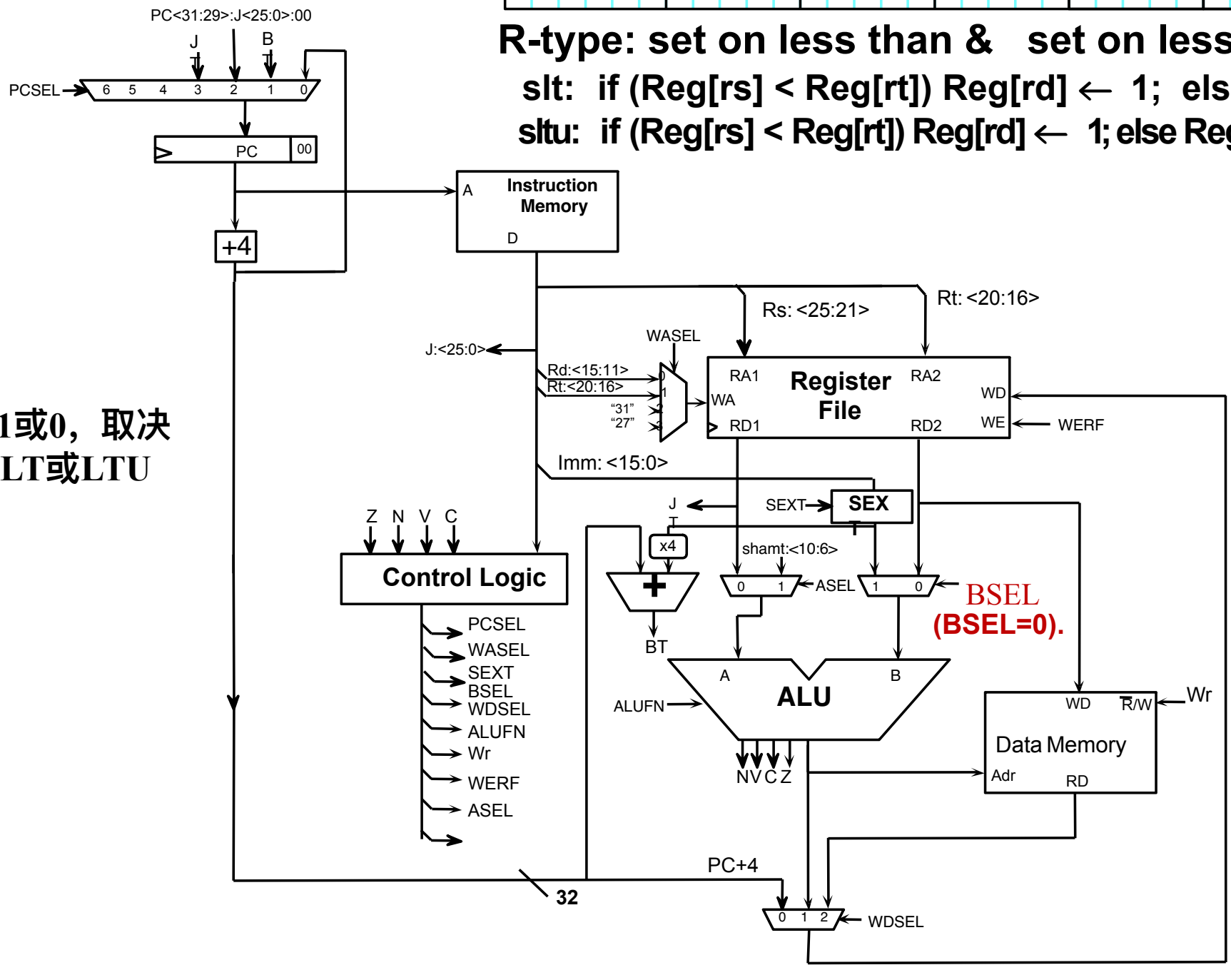


R-type: set on less than & set on less than unsigned

slt: if (Reg[rs] < Reg[rt]) Reg[rd] ← 1; else Reg[rd] ← 0

sltu: if (Reg[rs] < Reg[rt]) Reg[rd] ← 1; else Reg[rd] ← 0

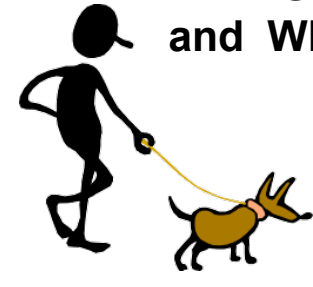
ALU输出1或0，取决于标志位LT或LTU



Reminder:

LT = N ⊕ V

LTU = C

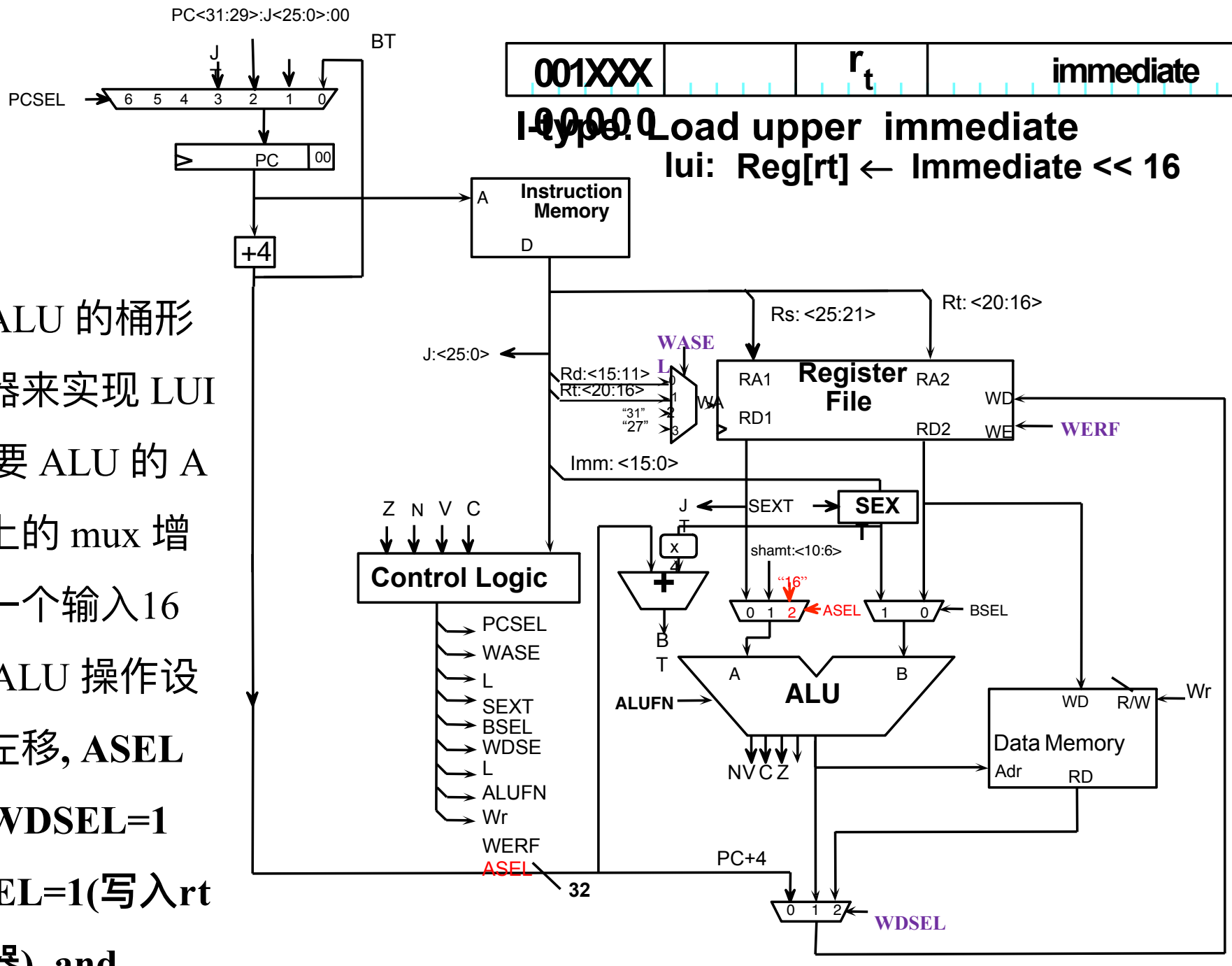


WDSEL = 1 and WERF = 1

LUI Ends

指令的意义:把立即数imm左移16位，结果存放在寄存器rt中。把PC + 4写入PC

使用 ALU 的桶形移位器来实现 LUI 。需要 ALU 的 A 输入上的 mux 增加另一个输入16，将 ALU 操作设置为左移, **ASEL = 2, WDSEL=1 WASEL=1(写入rt寄存器), and WERF=1.**



Reset, Interrupts, and Exceptions

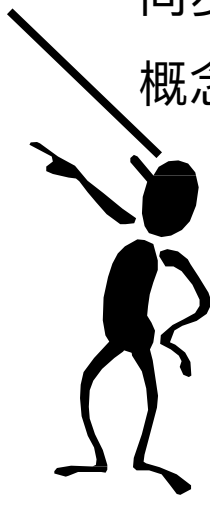
- 首先，我们需要一些方法让我们的机器进入一个已知的初始状态。这并不意味着所有寄存器都会被初始化，只是我们会知道从哪里获取第一条指令。我们将此控制输入称为 RESET
- 我们还希望可恢复性中断，用于故障处理（例如，非法指令）

CPU or SYSTEM 引起的中断 [同步]

陷阱和系统调用（例如，读取字符） CPU产生[同步，由指令引起]

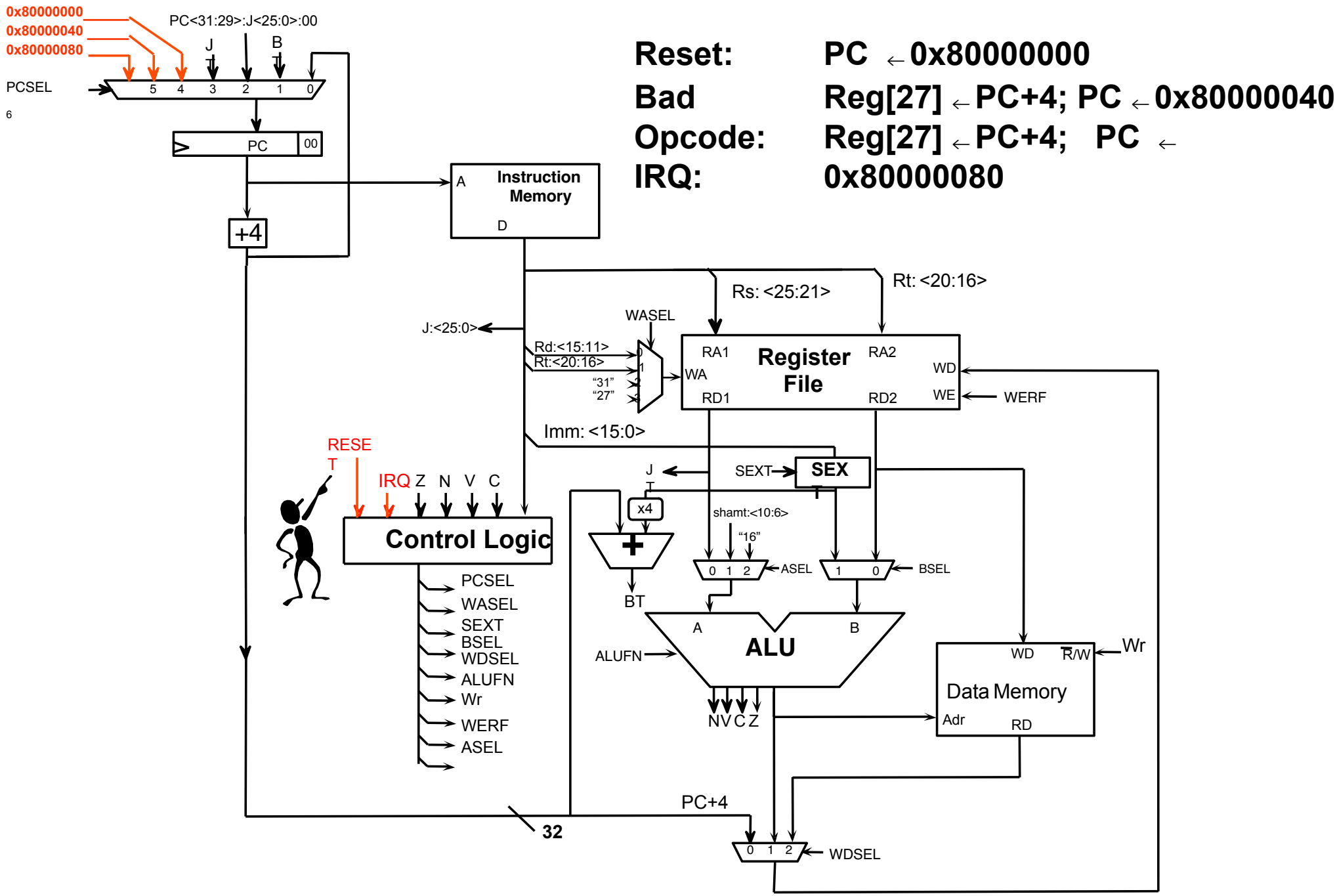
I/O events (eg, key press) 外设引起的中断

外设产生 [异步]



异常目标：中断正在运行的程序，调用异常处理程序，返回继续执行。

Exceptions



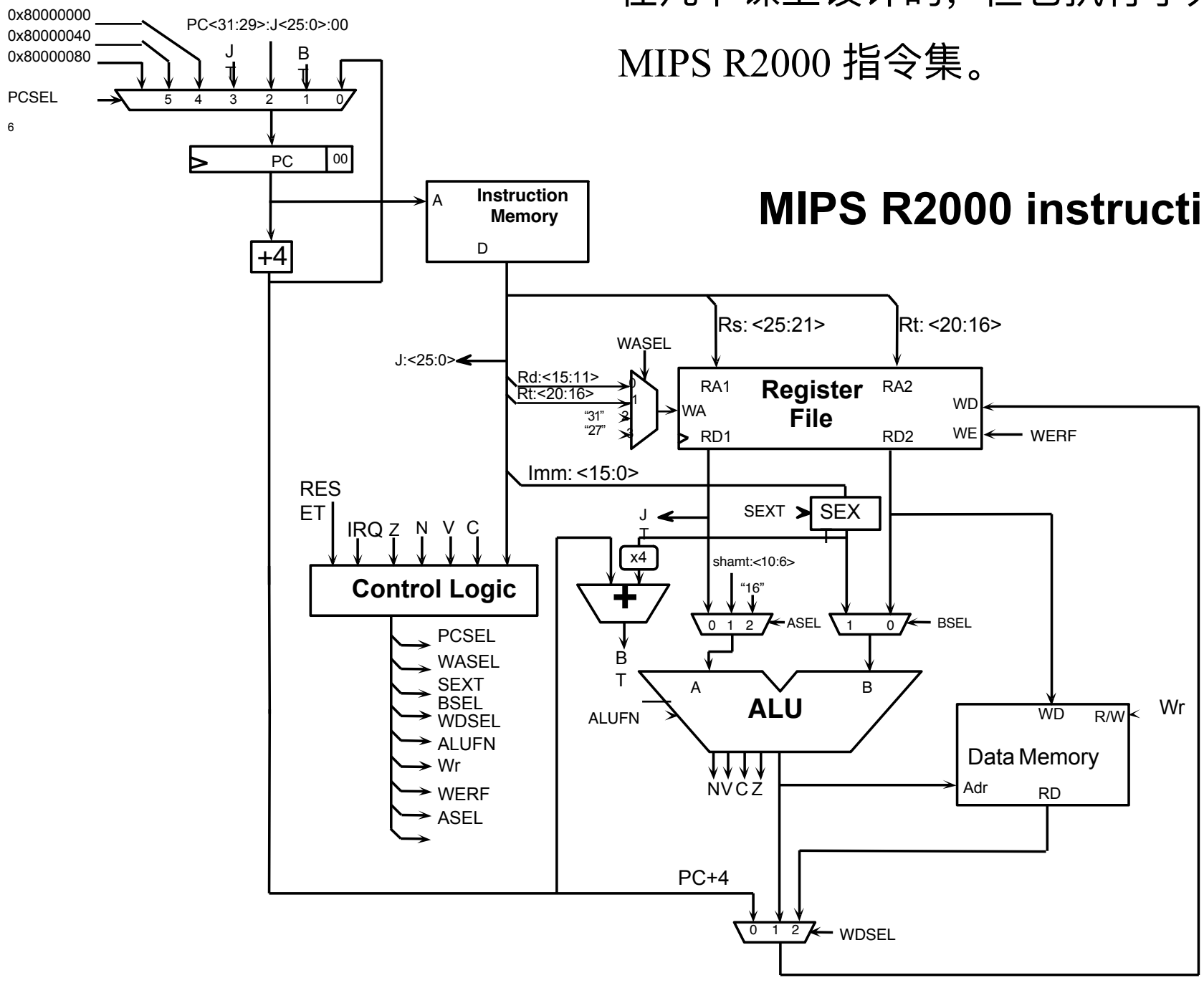
Reset: PC ← 0x80000000

Bad Opcode: Reg[27] ← PC+4; PC ← 0x80000040

IRQ: Reg[27] ← PC+4; PC ← 0x80000080

这是一个完整的 32 位处理器。 虽然是在几节课上设计的，但它执行了大部分 MIPS R2000 指令集。

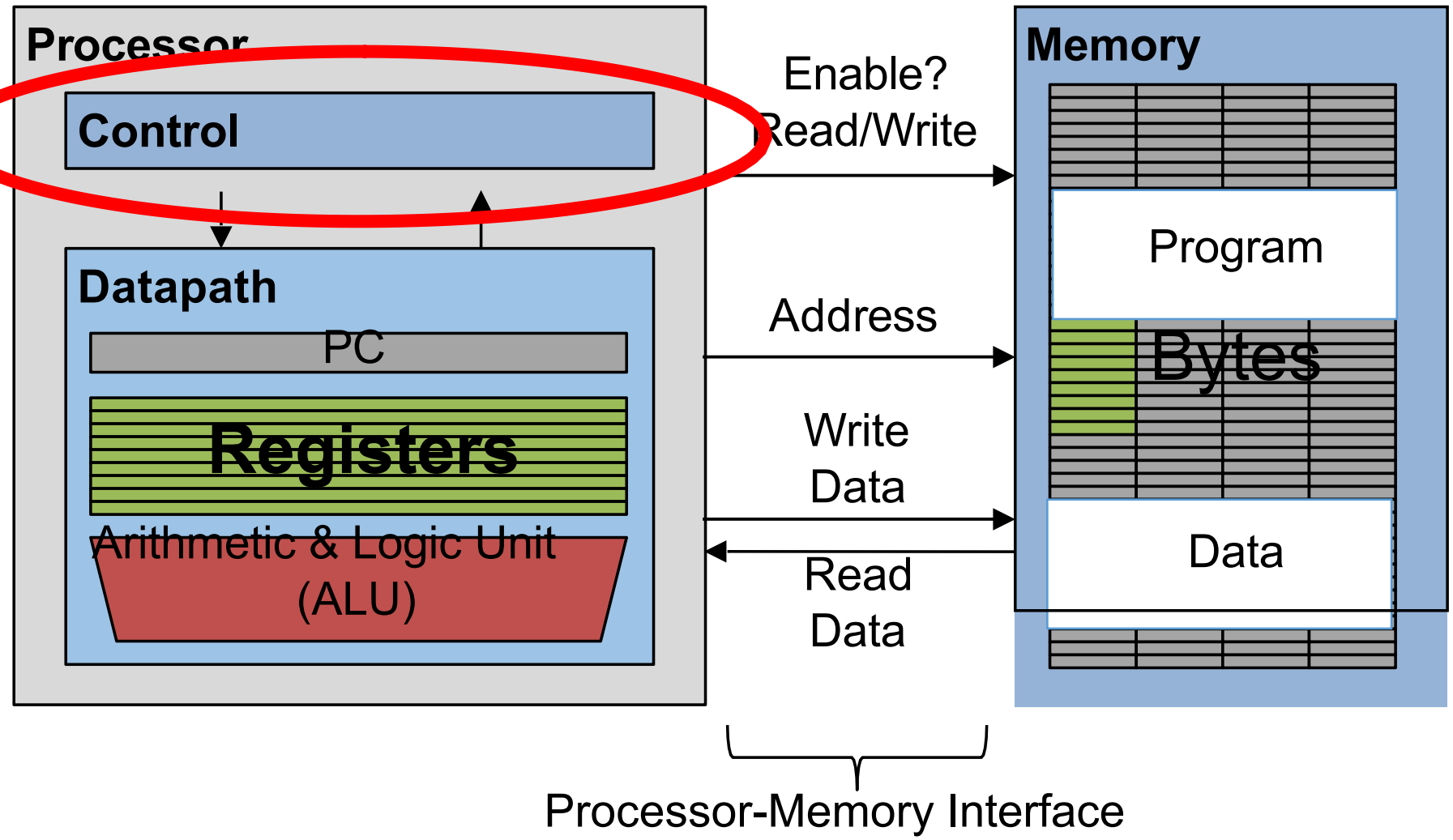
MIPS R2000 instruction set.



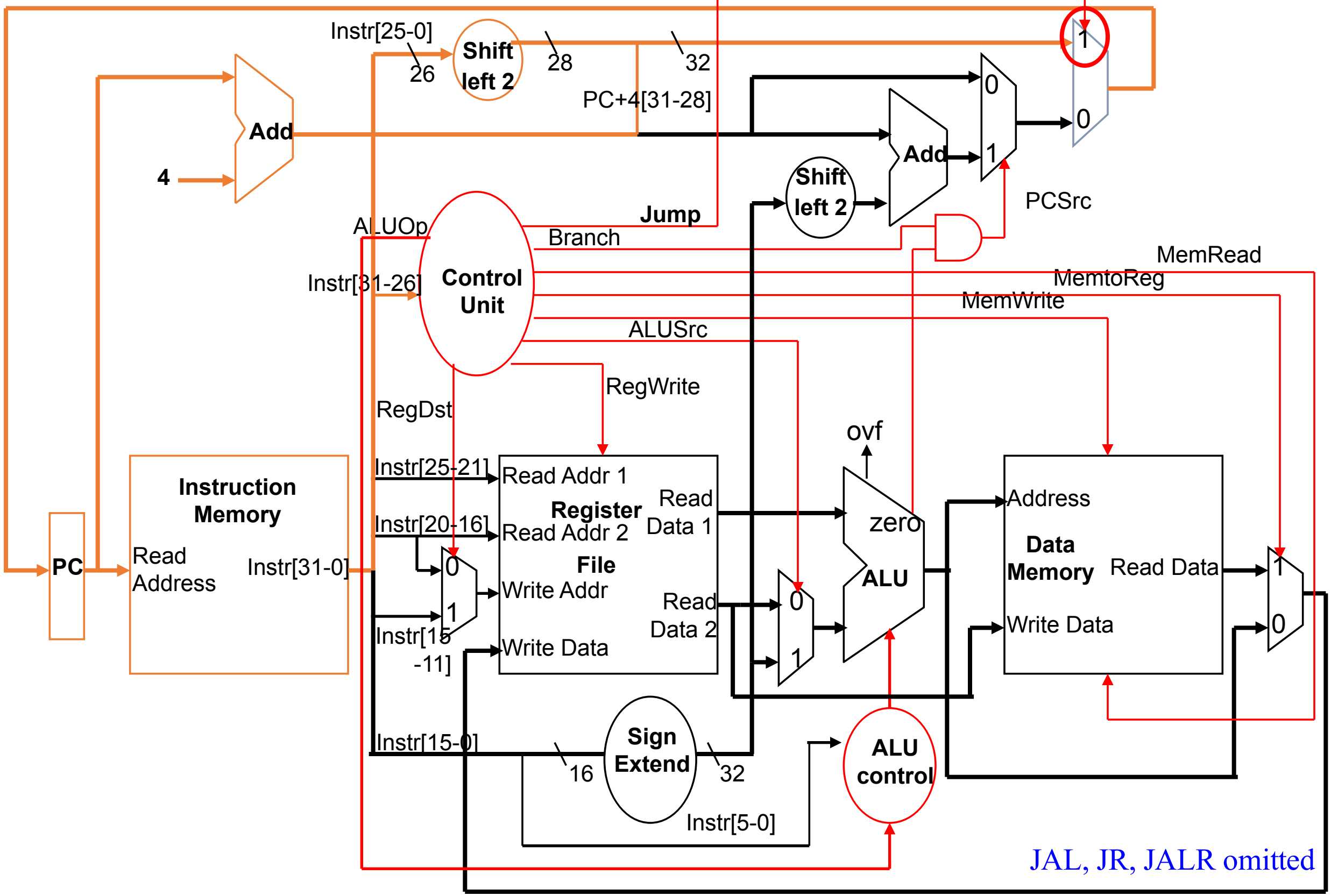
每个时钟执行一条指令，剩下的就是控制逻辑设计了。

单周期控制器的设计

Processor



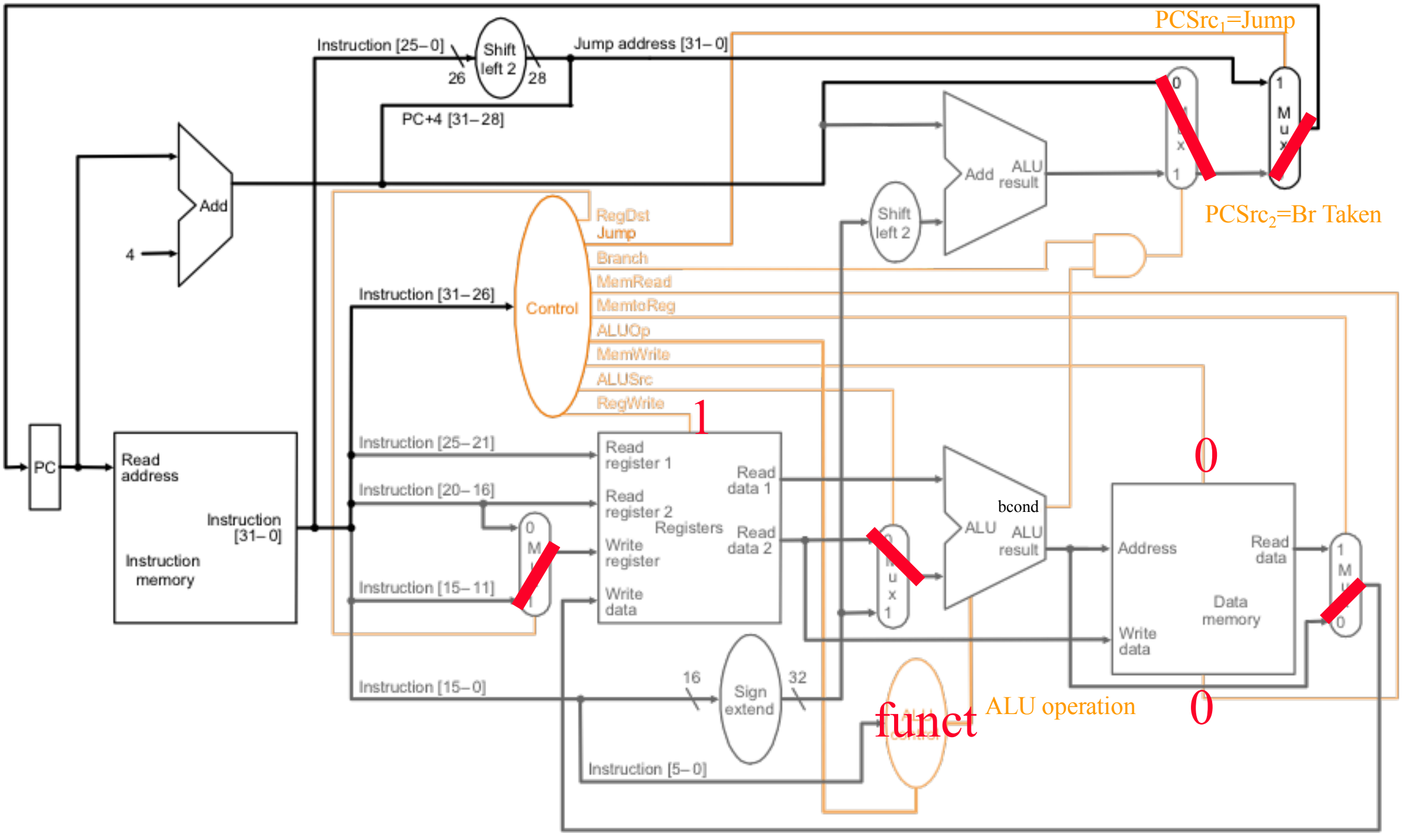
Mips CPU


Instr[25-0]
26
Shift left 2
28
32
PC+4[31-28]
Add
4
Add
Shift left 2
Jump
Branch
PCSrc
MemRead
MemtoReg
MemWrite
ALUSrc
RegWrite
RegDst
ALUOp
Instr[31-26]
Control Unit
Instruction Memory
Read Address
Instr[31-0]
PC
Instr[25-21]
Instr[20-16]
Instr[15-11]
Instr[15-0]
Instr[5-0]
Read Addr 1
Read Addr 2
Register File
Write Addr
Write Data
Read Data 1
Read Data 2
ovf
zero
ALU
Address
Data Memory
Read Data
Write Data
Sign Extend
16
32
ALU control
0
1
JAL, JR, JALR omitted

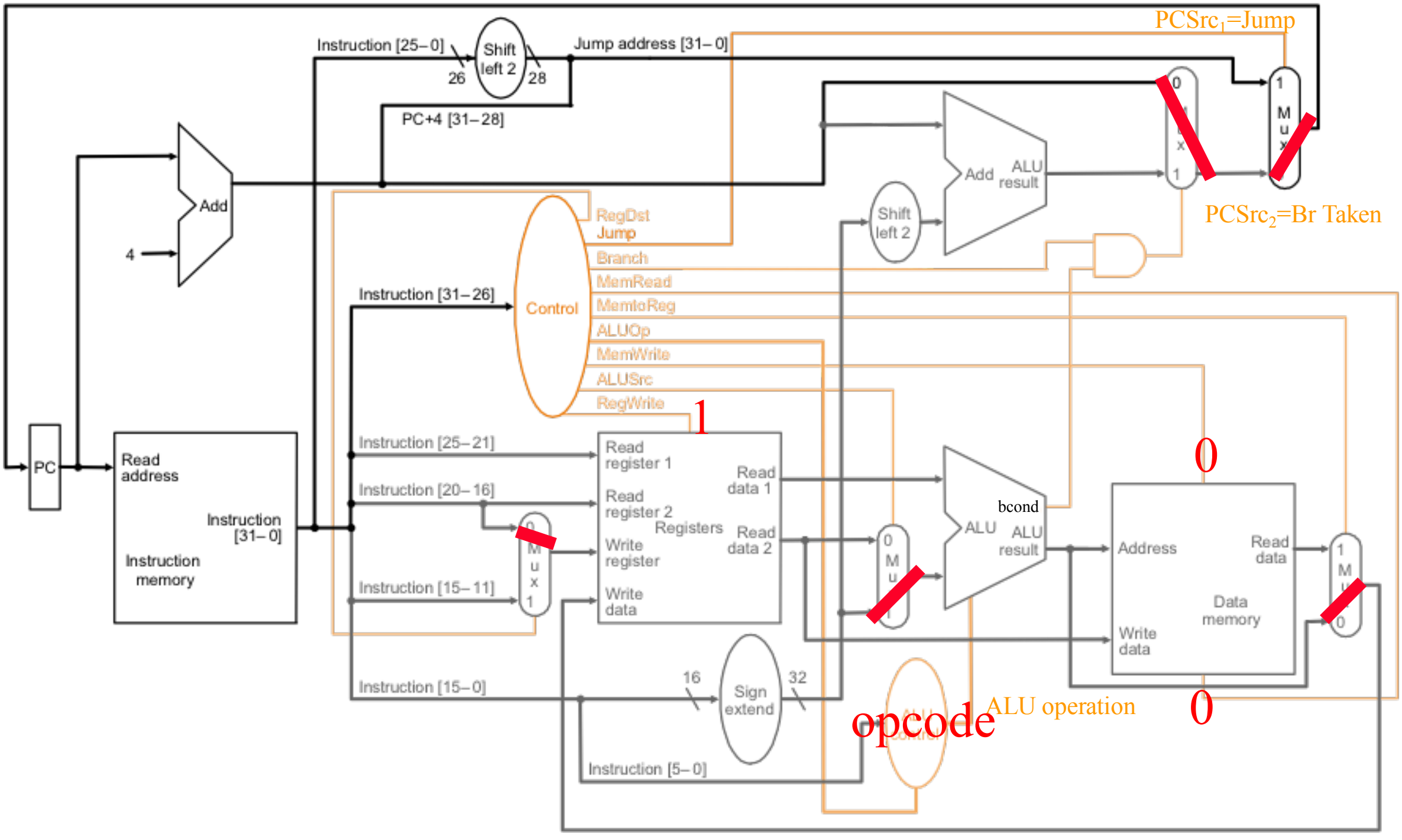
Recap: A Summary of Control Signals

inst	Register Transfer	
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 = RegB, ALUctr = “sub”, RegDst = rd, RegWr, nPC_sel = “+4”	
ORi	R[rt] <– R[rs] + zero_ext(Imm16);	PC <– PC + 4
	ALUsrc = Im, Extop = “Z”, ALUctr = “or”, RegDst = rt, RegWr, nPC_sel = “+4”	
LOAD	R[rt] <– MEM[R[rs] + sign_ext(Imm16)];	PC <– PC + 4
	ALUsrc = Im, Extop = “Sn”, ALUctr = “add”, MemtoReg, RegDst = rt, RegWr, nPC_sel = “+4”	
STORE	MEM[R[rs] + sign_ext(Imm16)] <– R[rs];	PC <– PC + 4
	ALUsrc = Im, Extop = “Sn”, ALUctr = “add”, MemWr, nPC_sel = “+4”	
BEQ	if (R[rs] == R[rt]) then PC <– PC + sign_ext(Imm16)] || 00 else PC <– PC + 4	
	nPC_sel = “Br”, ALUctr = “sub”	

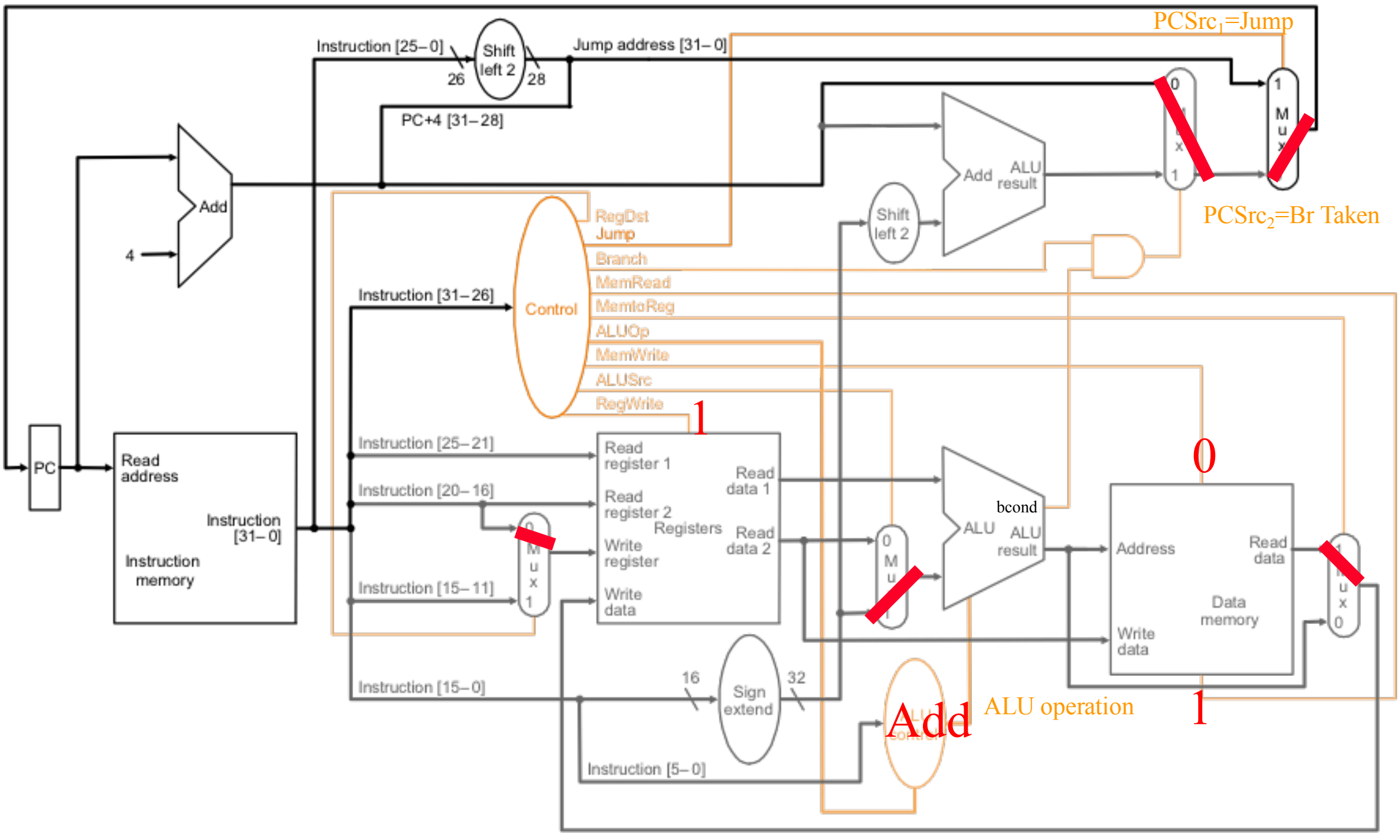
R-Type ALU


PCSrc1=Jump
Instruction [25–0]
Shift left 2
26
28
Jump address [31–0]
PC+4 [31–28]
Add
4
PC
Read address
Instruction [31–0]
Instruction memory
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
1
0
M u x
Shift left 2
Add
ALU result
ALU
bcond
16
Sign extend
32
funct
ALU operation
Address
Read data
Data memory
Write data
PCSrc2=Br Taken

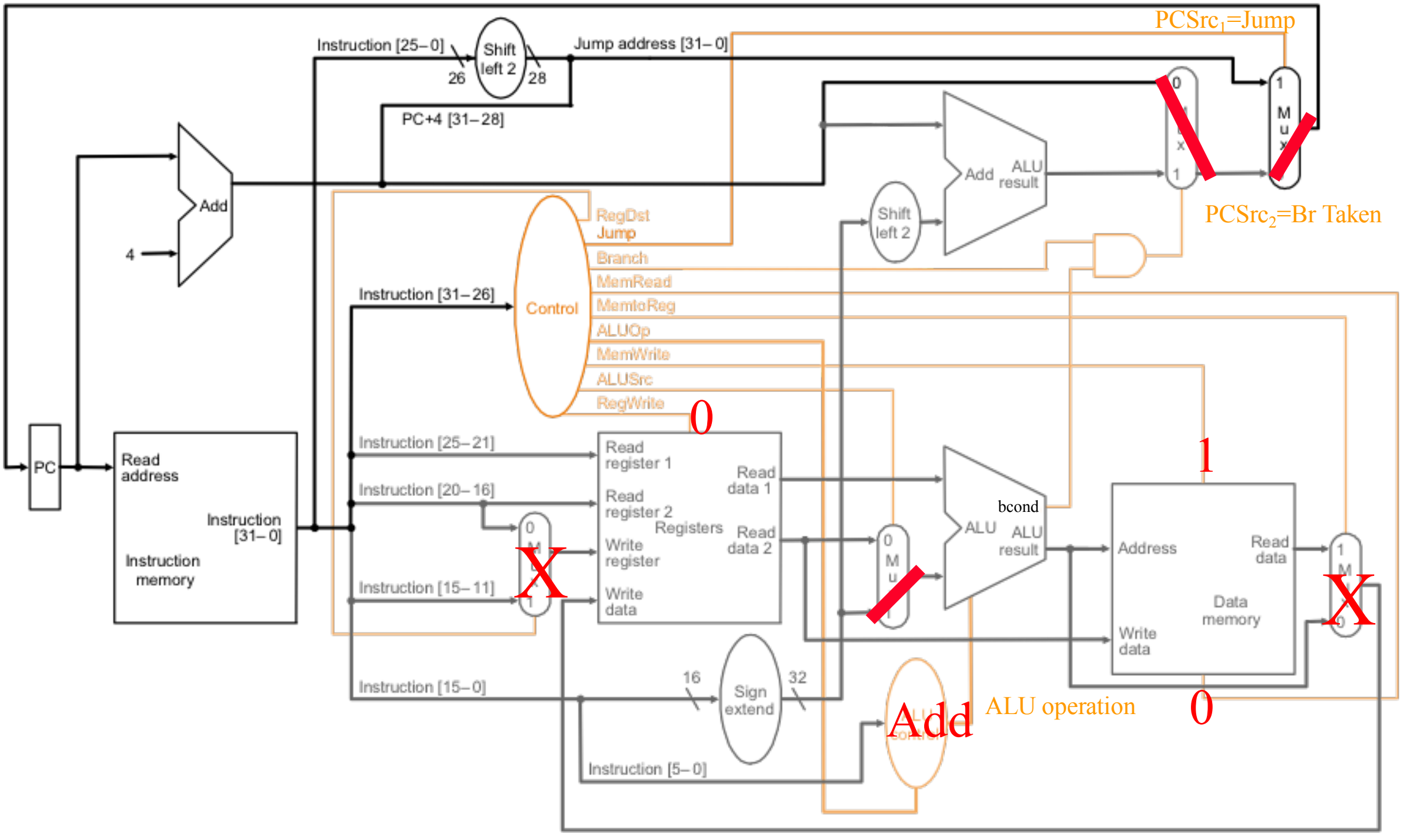
I-Type ALU


PCSrc1=Jump
Instruction [25–0]
Shift left 2
26
28
Jump address [31–0]
PC+4 [31–28]
Add
4
PC
Read address
Instruction [31–0]
Instruction memory
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
1
Shift left 2
Add
ALU result
0
M u x
1
PCSrc2=Br Taken
bcond
ALU
ALU result
Address
Read data
Data memory
Write data
0
16
Sign extend
32
opcode
ALU operation
0
1
M u x
0

LW

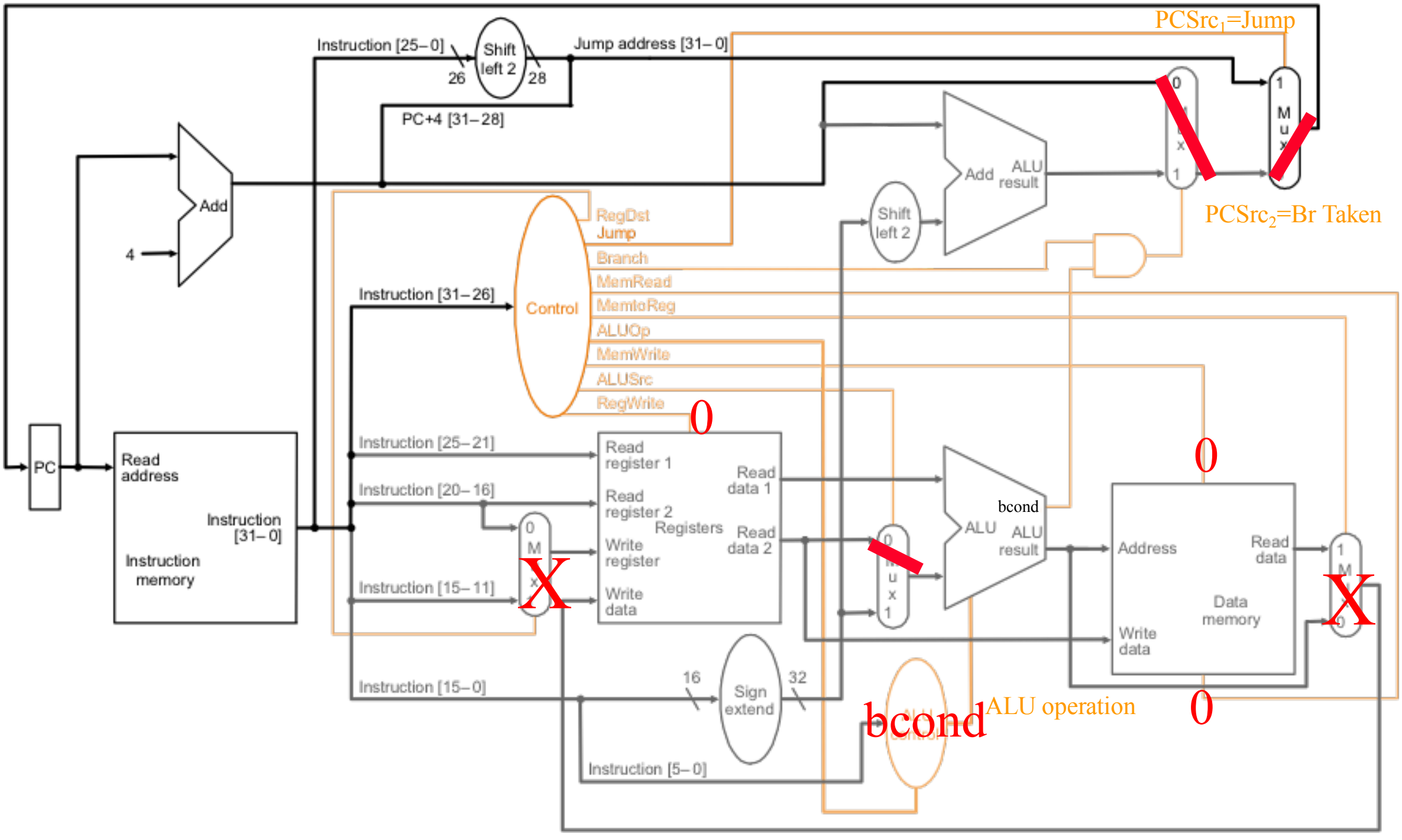

PCSrc1=Jump
Instruction [25–0]
Shift left 2
26
28
Jump address [31–0]
PC+4 [31–28]
Add
4
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
Shift left 2
Add
ALU result
0
M u x
1
PCSrc2=Br Taken
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign extend
32
1
0
bcond
ALU
ALU result
Address
Read data
Data memory
Write data
Add
ALU operation

SW


PCSrc1=Jump
Instruction [25–0]
Shift left 2
26
28
Jump address [31–0]
PC+4 [31–28]
Add
4
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
Shift left 2
Add
ALU result
0
M u x
1
PCSrc2=Br Taken
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign extend
32
ALU
ALU result
bcond
Address
Read data
Data memory
Write data
Add
ALU operation
0
1
X
X
0
1

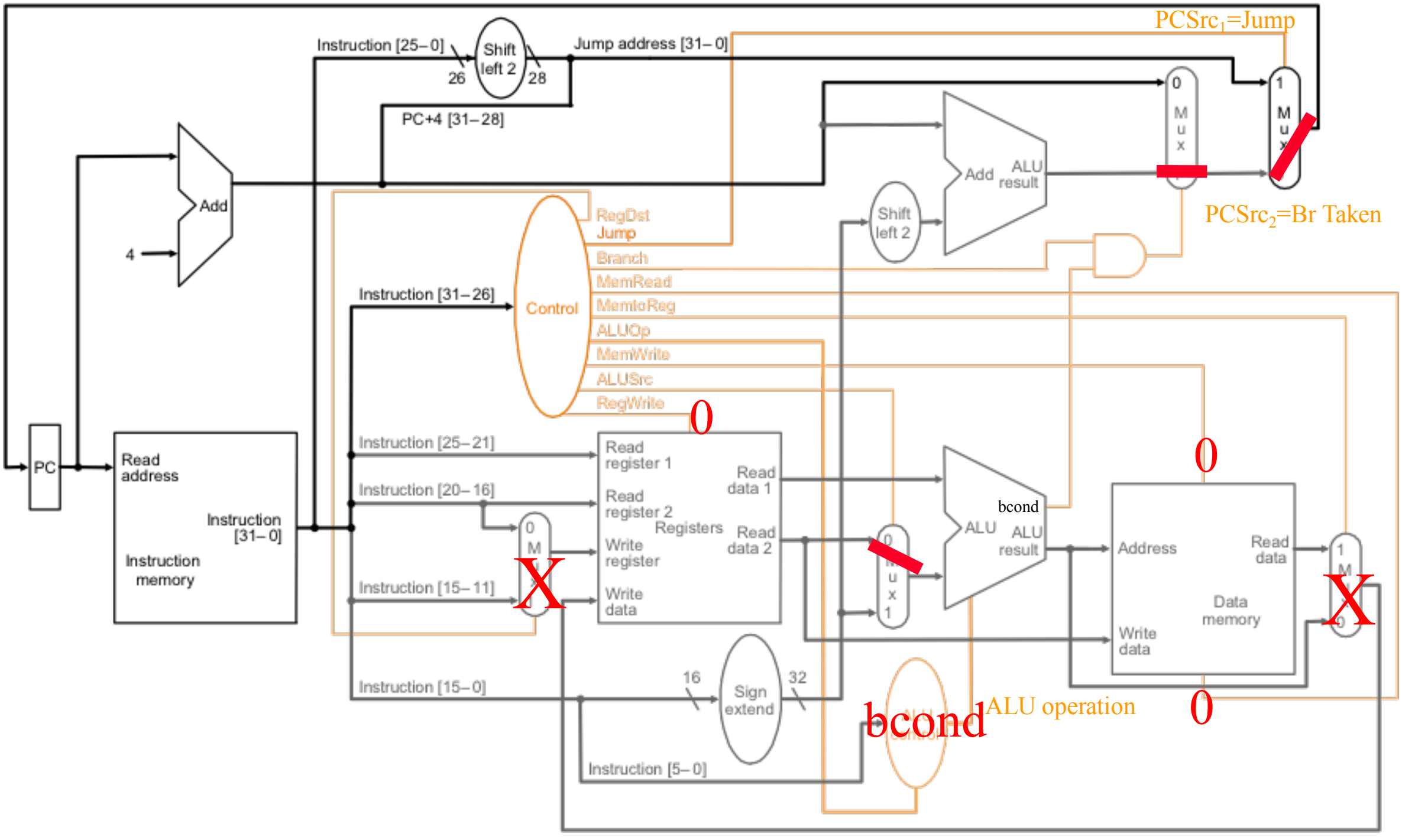
Branch (Not Taken)

Some control signals are dependent on the processing of data

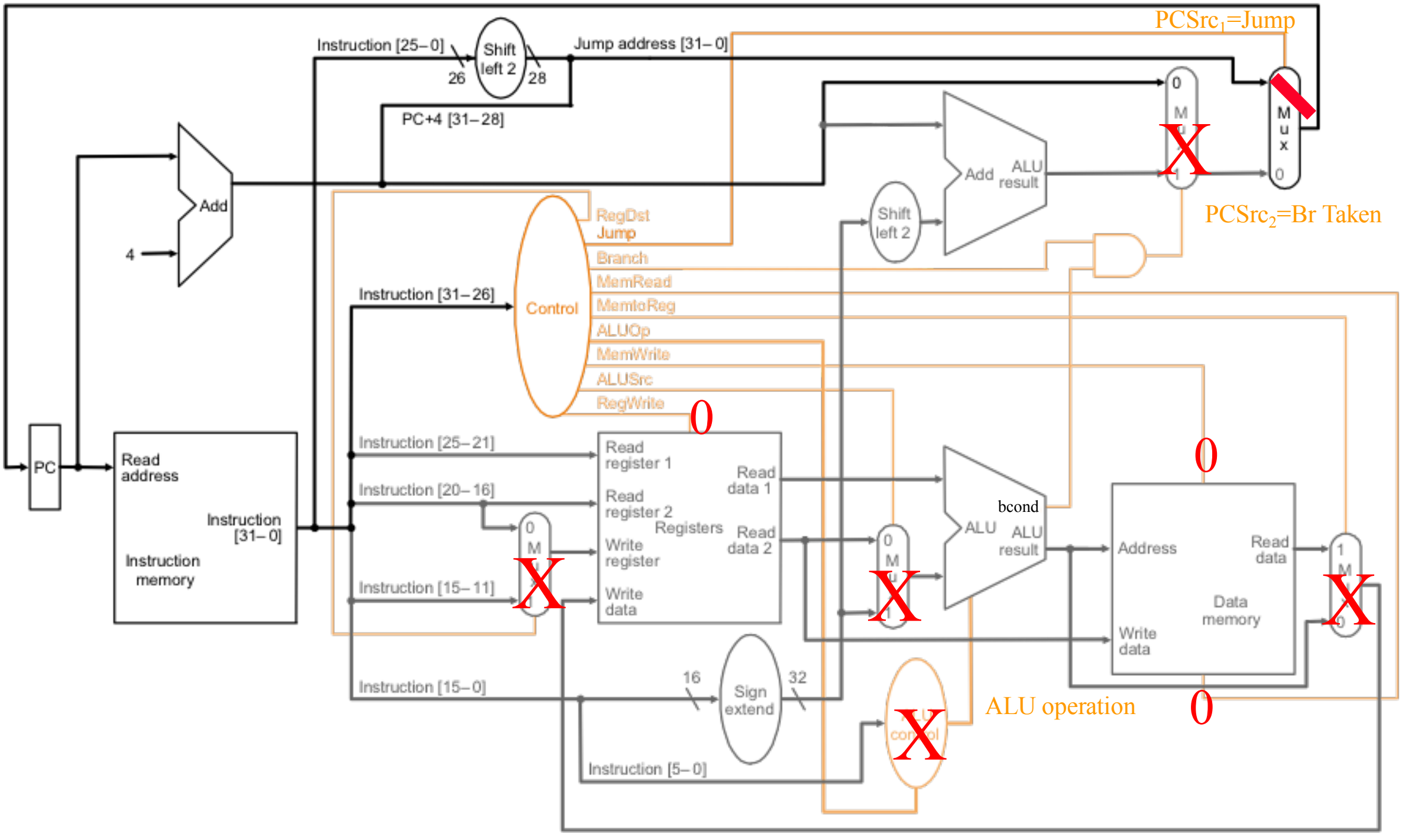


Branch (Taken)

Some control signals are dependent on the processing of data



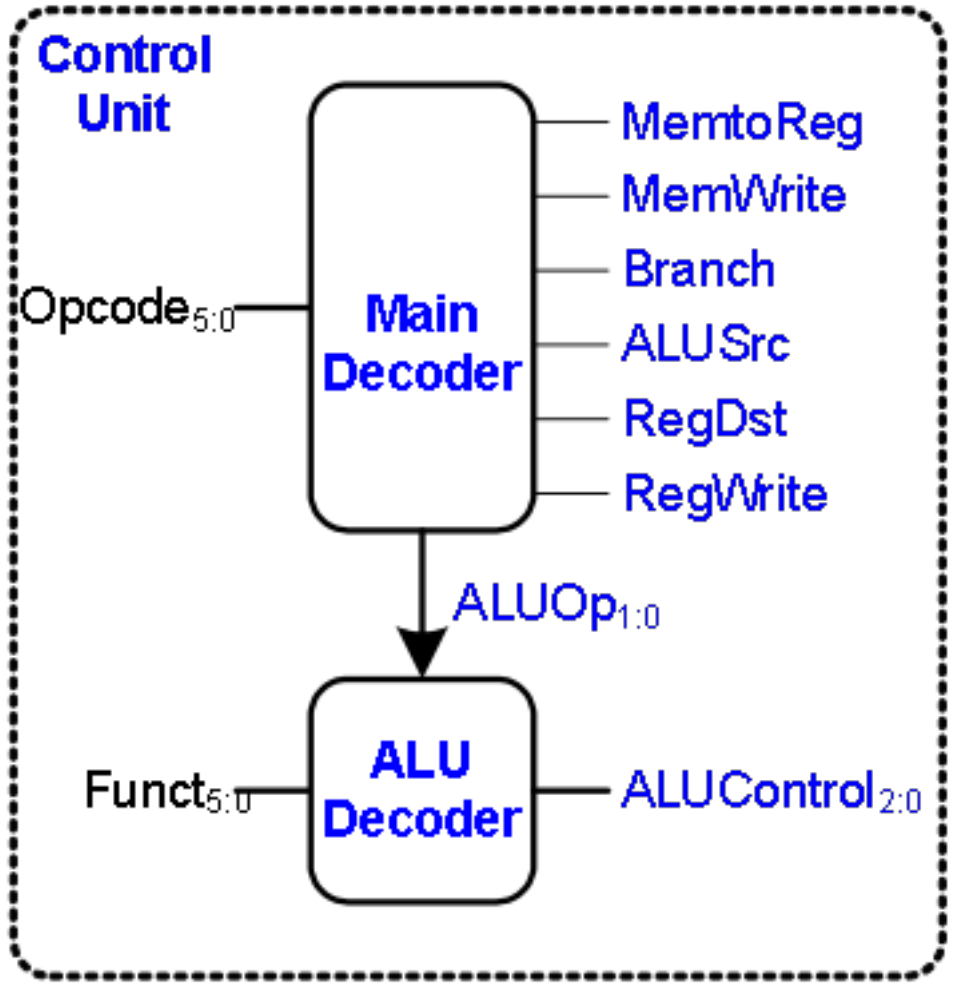
Jump


PCSrc1=Jump
Instruction [25–0]
Shift left 2
26
28
Jump address [31–0]
PC+4 [31–28]
Add
4
M u x
0
1
Add
ALU result
Shift left 2
PCSrc2=Br Taken
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign extend
32
ALU
bcond
ALU result
Address
Read data
Data memory
Write data
ALU operation
0
X

MIPS Datapath 有几个选择

- ALU 输入
  - RT或者立即数 *(MUX)*
- 寄存器堆写入地址
  - RD 或者 RT *(MUX)*
- 寄存器堆写入数据
  - ALU 运算结果或者数据存储器取出的数据 *(MUX)*
- 寄存器堆写使能信号
  - 不是总是有效写入状态 *(MUX)*
- 数据存储器写使能信号
  - 只有sw指令需要写入 *(MUX)*

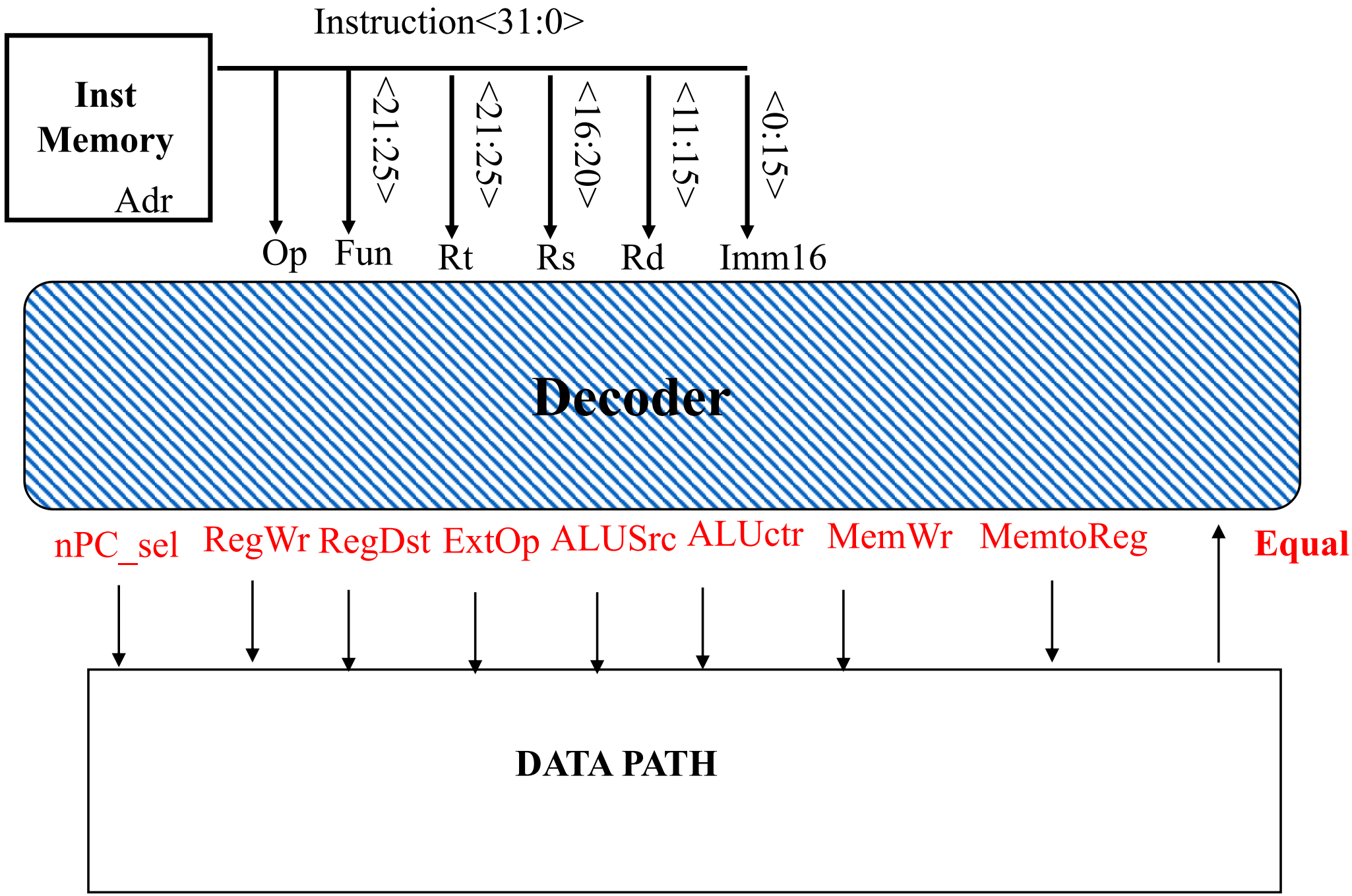
所有这些选择需要控制信号



控制装置
Control Unit

AluOp (3:0) 操作码	Mnemonic助记符	运算结果Result =	Description
0000	add	A + B	Addition 加
0010	sub	A - B	Subtraction 减
0100	and	A and B	Logical and 与
0101	or	A or B	Logical or 或
0110	xor	A xor B	Exclusive or 异或
0111	nor	A nor B	Logical nor 或非
1000	sll	A << B	逻辑左移
1001	srl	A >> B	逻辑右移
1010	slt	(A - B)[31]	Set less than 小于设置
Others	n.a.	Don’t care	

Step 5: Assemble Control logic



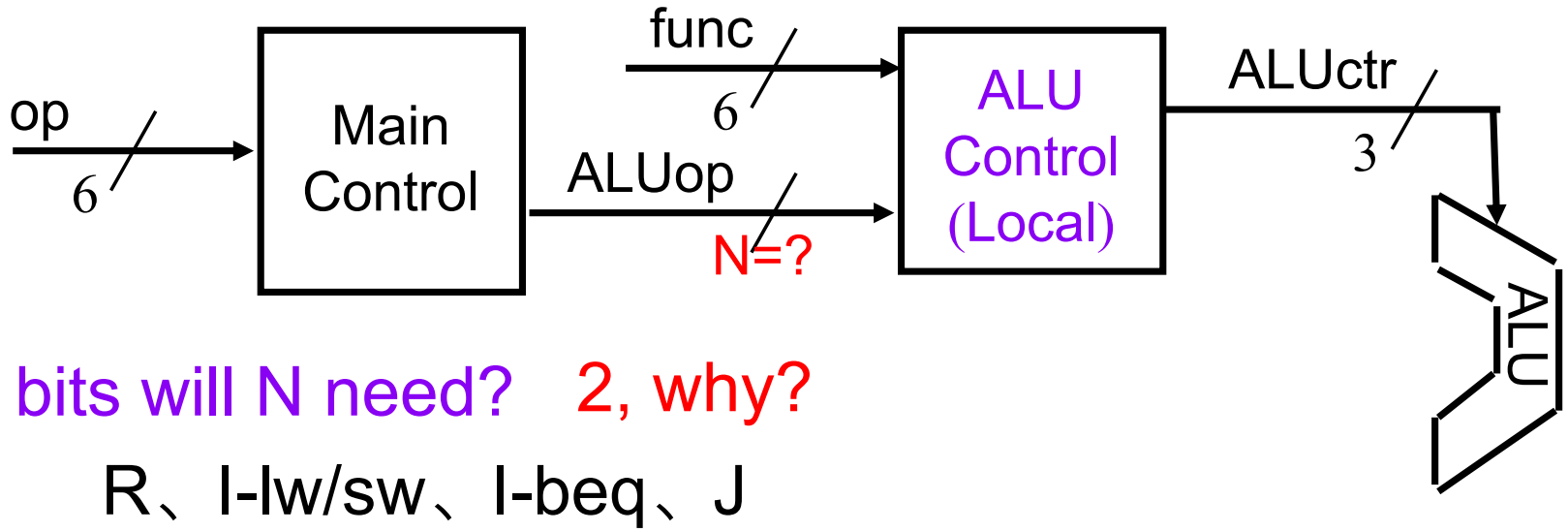
控制器的设计

- 实现方式：组合逻辑电路
  - 输入信号
    - 指令字段的op和func。
    - ALU的zero、V、N状态

- 确定每条指令的控制信号
  - 0
  - 1
  - X(与该指令无关)
- 构建控制信号的真值表

控制器的设计

- 控制器：一个宠大的逻辑电路
- 好的方法：分成多个较小的逻辑电路
- 较小规模的逻辑电路速度更快
- 较小规模的逻辑电路更容易协同工作
- 显然
- func字段只与ALU的operation有关
- 好的方法：建立一个单独的ALU控制电路



How many bits will N need? 2, why?

R、I-lw/sw、I-beq、J

主控单元

输出 **ALUop** 作为ALU控制电路的输入

Inputs								Outputs	
Opcode	op5	op4	op3	op2	op1	op0	Value	ALUOp2	ALUOp1
R-Format	0	0	0	0	0	0	0_{10}	1	X
lw	1	0	0	0	1	1	35_{10}	0	0
sw	1	0	1	0	1	1	43_{10}	0	0
beq	0	0	0	1	0	0	4_{10}	X	1
Jump	0	0	0	0	1	0	2_{10}	X	X

主控输出: 数据通路中的控制信号

Signal	R-Format	lw	sw	beq	Jump
RegDst	1	0	X	X	X
ALUSrc	0	1	1	0	X
MemtoReg	0	1	X	X	X
RegWrite	1	1	1	0	0
MemWrite	0	0	0	0	0
Branch	0	0	0	1	0
Jump	0	0	0	0	1
ExtOp	X	1	1	X	X
ALUctr		add	add	subtract	XXX

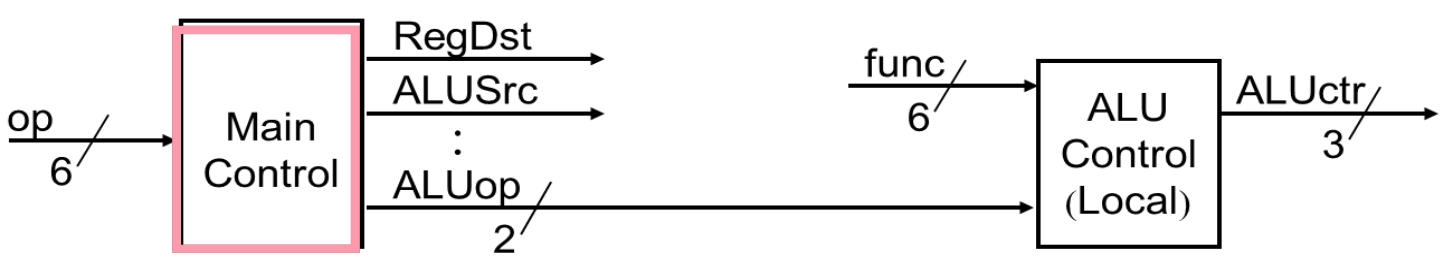
ALU控制电路

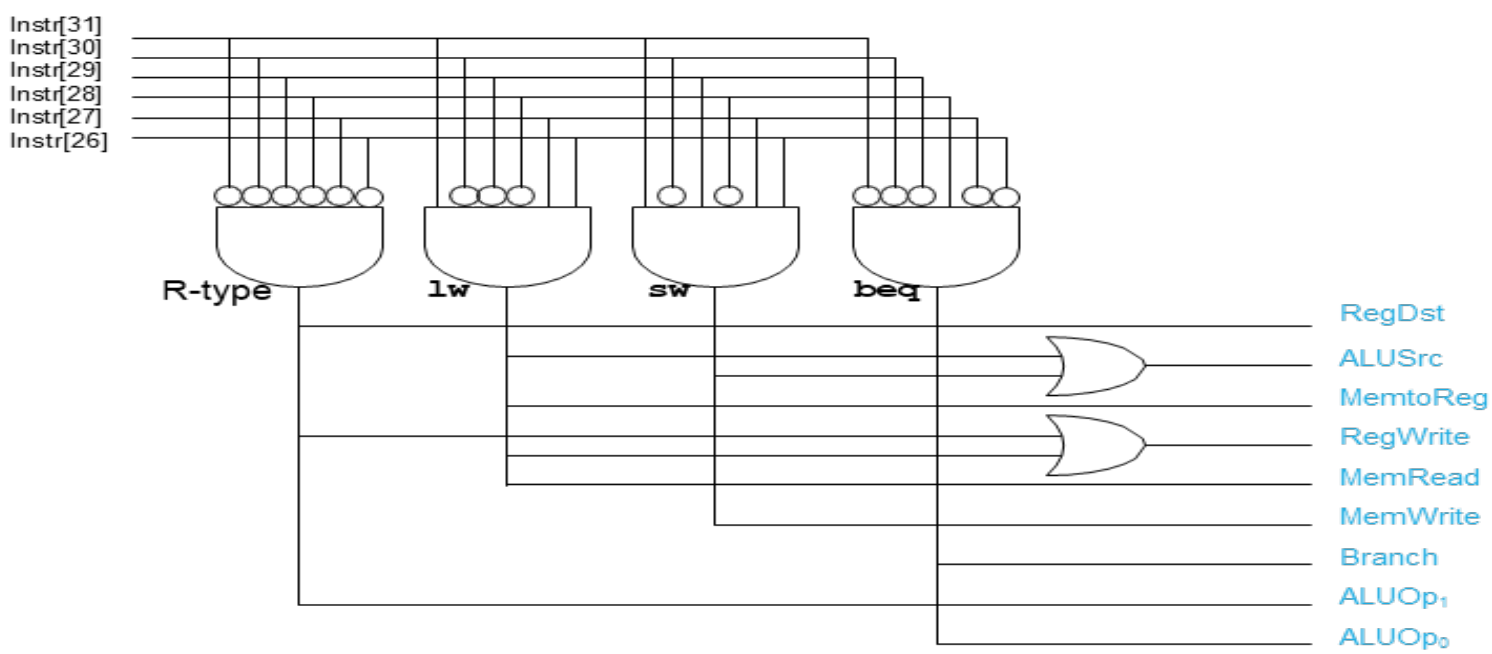
ALUOp		Funct field						Operation	
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0		
0	0	X	X	X	X	X	X	010	lw/sw
X	1	X	X	X	X	X	X	110	beq
1	X	X	X	0	0	0	0	010	R-Type
1	X	X	X	0	0	1	0	110	
1	X	X	X	0	1	0	0	000	
1	X	X	X	0	1	0	1	001	
1	X	X	X	1	0	1	0	111	

ALUControl	Function
000	AND
001	OR
010	add
110	subtract
111	set on less than

主控逻辑

Instr	RegDst	ALUSrc	MemReg	RegWr	MemRd	MemWr	Branch	ALUOp
R-type 000000	1	0	0	1	0	0	0	10
lw 100011	0	1	1	1	1	0	0	00
sw 101011	X	1	X	0	0	1	0	00
beq 000100	X	0	X	0	0	0	1	01





控制器硬件描述语言实现

- 逻辑case 语句实现...

always @*

begin

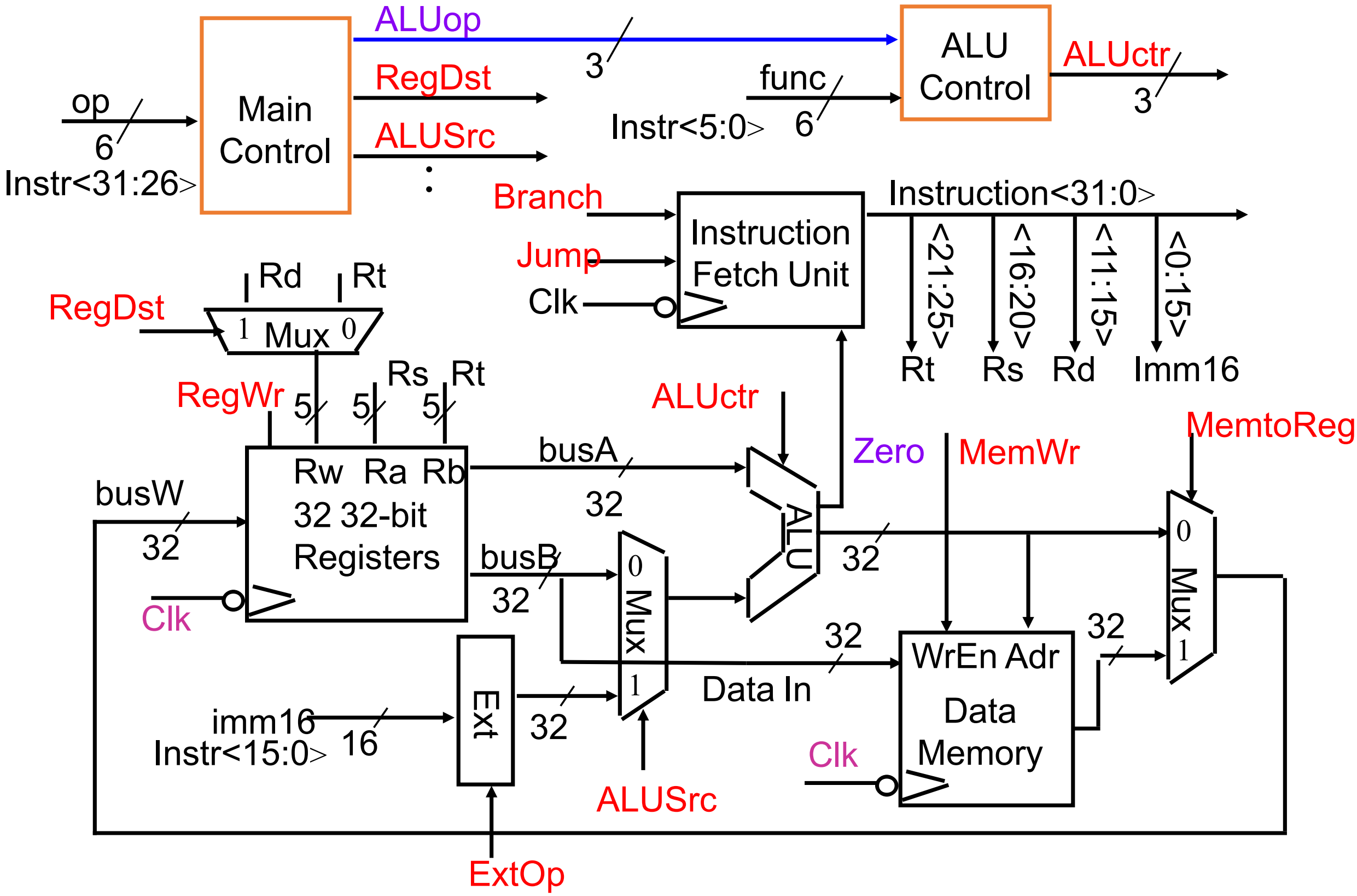
op = instr[26:31]; imm = instr[15:0]; ...

case (op)

6'b000000: begin reg_write = 1; ... end

...

The Complete Single Cycle Data Path with Control

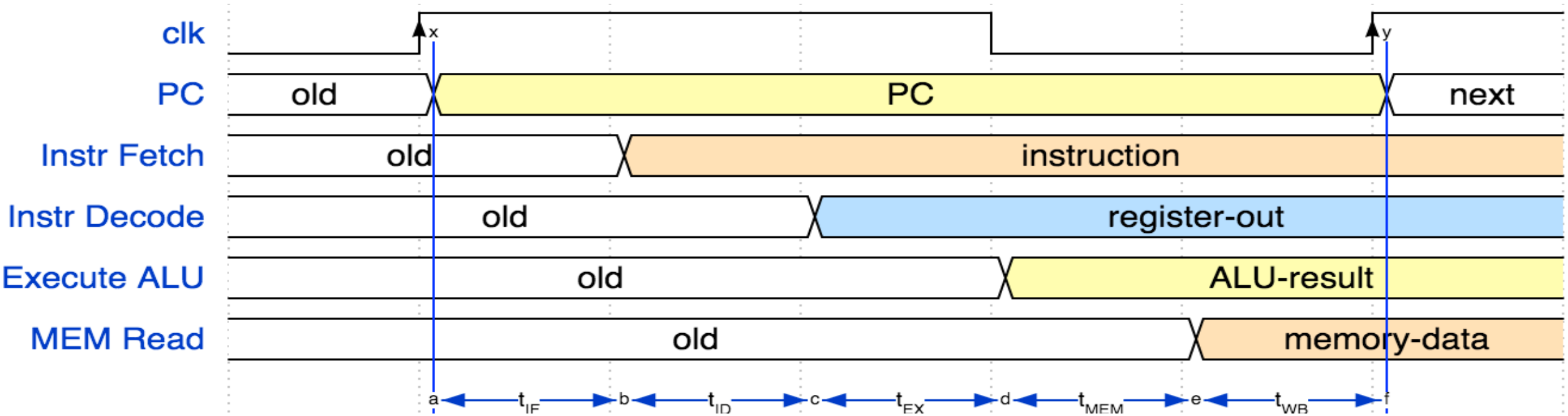


Magic Moment!

此刻, 你应该可以设计一个计算机了!

1. 按照数据通路要求分析指令集
2. 选择数据通路一组部件
3. 组装满足要求的数据通路
4. 分析每条指令的实现以确定控制信号的设置
5. 组装控制逻辑
6. 设置时钟频率

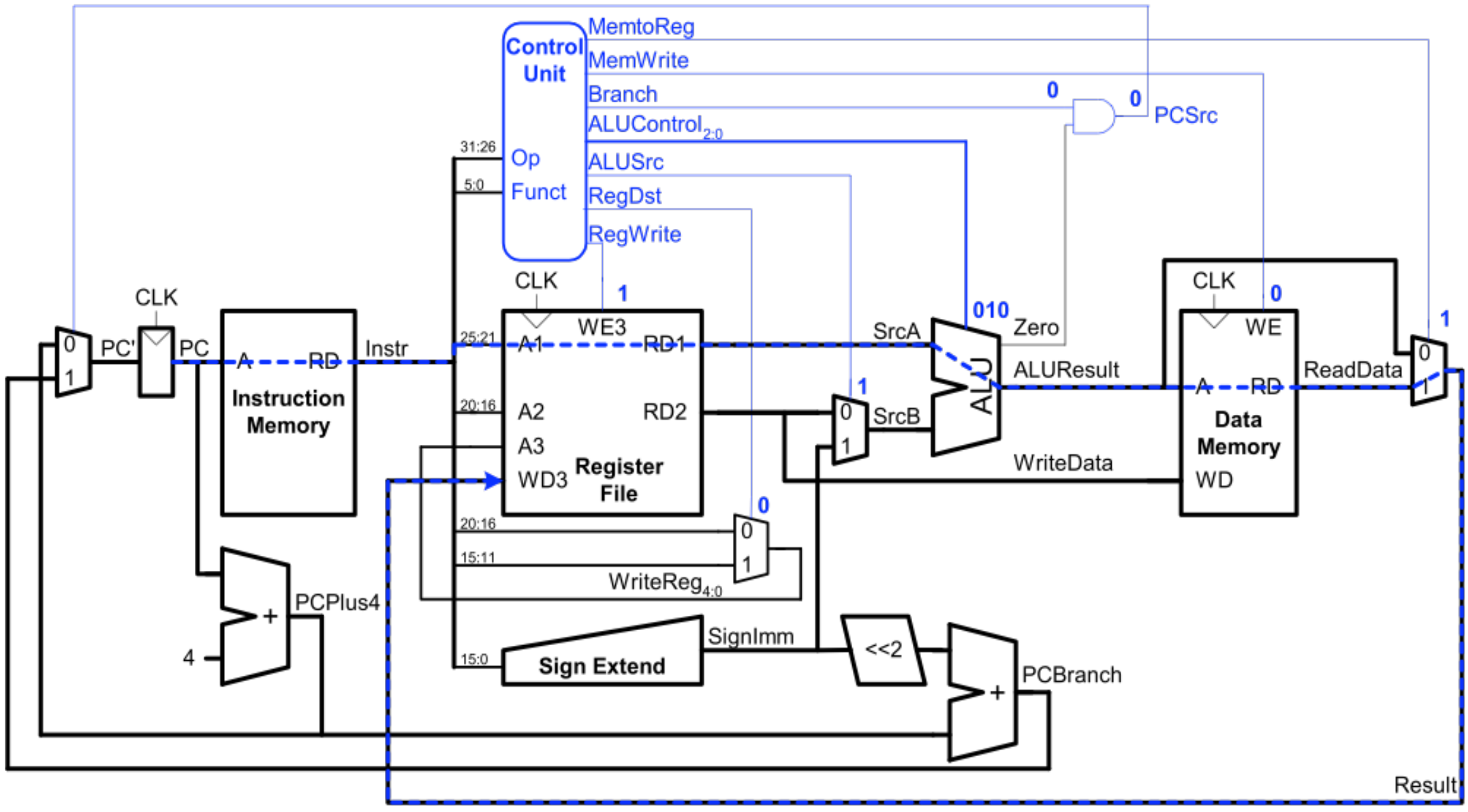
Approximate Instruction Timing



IF	ID	EX	MEM	WB	Total
I-MEM	Reg Read	ALU	D-MEM	Reg W	
2 ns	1 ns	2 ns	2 ns	1 ns	8 ns

Single-Cycle Performance

- T_c is limited by the critical path (`lw`)



- *Single-cycle critical path:*

$T_c = t_{q\_PC} + t_{mem} + \max(t_{RFread}, t_{sext} + t_{mux}) + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$

Instruction Timing

Instr	IF = 200ps	ID = 100ps	ALU = 200ps	MEM=200ps	WB = 100ps	Total
add	X	X	X		X	600ps
beq	X	X	X			500ps
jal	X	X	X			500ps
lw	X	X	X	X	X	800ps
sw	X	X	X	X		700ps

Time Delay for LW: Critical Path

我们的单周期数据通路的最坏情况发生在LOAD指令。

- Maximum clock frequency
- f_{max} = 1/800ps = 1.25 GHz

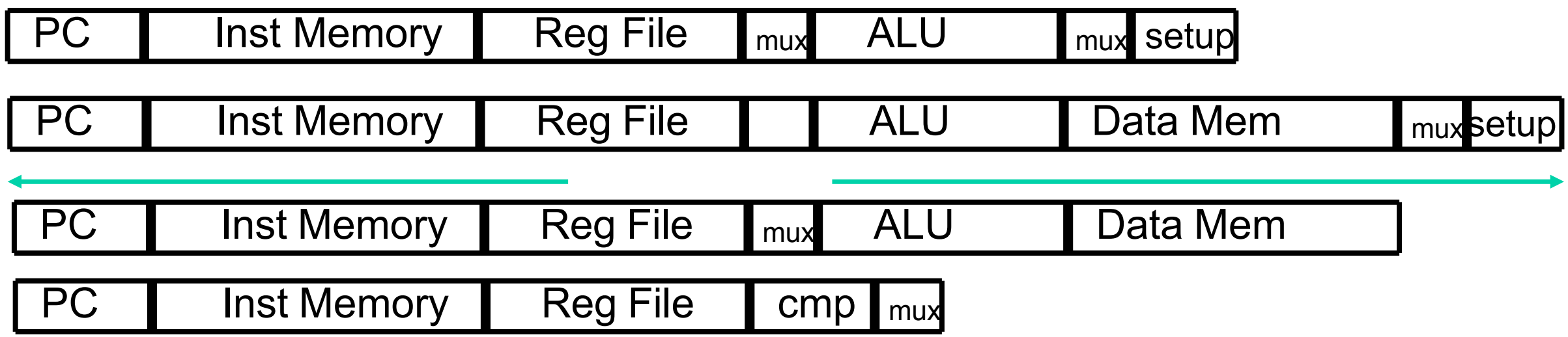
■ 切记：状态单元的输入信息总是在一个时钟边沿到达后的“Clk-to-Q”时才被写入到单元中，此时的输出才反映新的状态值

$$\text{Critical path} = t_{clk\text{-}q} + \max\{t_{Add}+t_{mux},\ t_{IMEM}+t_{Imm}+t_{mux}+t_{ALU}+t_{DMEM}+t_{mux},\ t_{IMEM}+t_{Reg}+t_{ALU}+t_{DMEM}+t_{mux}\}+t_{setup}$$

What's Wrong with Single Cycle?

memory (2ns), ALU/adders (2ns), register file access (1ns)

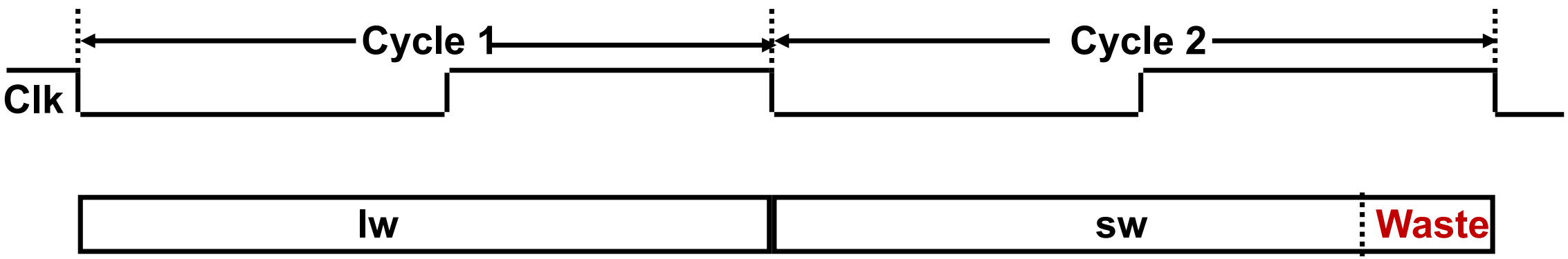
Inst.	Inst. Mem	Reg. Read	ALU	Data Mem	Reg. Write	Total
ALU	2	1	2		1	6
lw	2	1	2	2	1	8
sw	2	1	2	2		7
br	2	1	2			5



时钟周期很长，所有指令花费的时间与最慢的一样多，实际内存不如我们理想化的内存好 （不能总是在固定的时间内完成工作）

Single Cycle Disadvantages & Advantages

- 使用时钟周期效率低 - 时钟周期必须定时以适应最慢的指令，对于浮点乘法等更复杂的指令尤其成问题



- 可能会浪费空间，因为某些功能单元（例如，加法器）必须复制，因为它们不能在时钟周期内共享。
- 优点是简单容易理解

  - **单周期设计中时钟周期对所有的指令等长，因此要由计算机中可能的最长路径决定：**

    ——取指令　违反了设计原则　——　加速完成常用操作

其它可实现的方式

- 多周期处理器
  - 缩短指令周期
  - 一条指令多个周期
    - 不同类型指令所需的周期数不同
    - 硬件代价小
- 流水线处理
  - 指令重叠执行
  - 尽可缩短时钟周期数和CPI
  - 硬件代价大，但性能更好

单周期CPU的性能

- 各类指令的数据路径长度
  - R型指令　　200 + 50 + 100+0+50　　400ps
  - **Load word　　200 + 50 + 100 + 200 + 50　　600ps**
  - Store word　　200 + 50 + 100 + 200　　550ps
    - 分支　　200 + 50 + 100　　350ps
    - 转移　　200　　200ps
- 性能受最慢指令的限制

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

假设程序中各类指令占比：25%取数、10%存数、45%ALU、15%分支、5%跳转

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，也即为可变的（实际不可行，只是为了比较）

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				

各类指令要求的时间长度为：

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不等
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。

单周期计算机的性能

- 单周期计算机的性能练习：

  假设各主要功能单元的操作时间为：

  - 读存储器：10ns，写存储器：5ns
  - ALU和加法器：10ns
  - 寄存器堆（读/写）：5ns

  而MUX、控制单元、PC、扩展器和传输线路没有延迟，若各类指令的执行次数占总数的比例为：20%取数、10%存数、50%ALU、15%分支、5%跳转，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，但时钟周期是可以根据指令类型动态变化的。

单周期计算机的性能

解：方式(1)：时钟周期由最长指令来决定，应是load指令，为40ns——Load指令的执行如下：

取指令10ns,读寄存器堆5ns，ALU计算地址10ns,读存储器10ns,写寄存器堆5ns,总的时间是40 ns。

方式(2)：时钟周期取各条指令所需时间，根据各类指令的频度，计算出平均时钟周期为：

30×50% ＋ 40×20% ＋ 35×10% ＋25×15% +10 ×5% = 30.75ns

加速比=可变周期处理机的性能/固定周期处理机的性能

$= (I \times 1 \times 40) / (I \times 1 \times 30.75) = 1.3$

可变周期处理机的性能是固定周期处理机性能的1.3倍！

单周期计算机的性能

解：方式(1)：时钟周期由最长指令来决定，应是load指令，为40ns——Load指令的执行如下：

取指令10ns,读寄存器堆5ns，ALU计算地址10ns,读存储器10ns,写寄存器堆5ns,总的时间是40 ns。

方式(2)：时钟周期取各条指令所需时间，根据各类指令的频度，计算出平均时钟周期为：

30×50% ＋ 40×20% ＋ 35×10% ＋25×15% +10 ×5% = 30.75ns

加速比=可变周期处理机的性能/固定周期处理机的性能

$= (I \times 1 \times 40) / (I \times 1 \times 30.75) = 1.3$

可变周期处理机的性能是固定周期处理机性能的1.3倍！