第四讲
指令与单周期CPU
（第3部分：单周期CPU原理与设计）

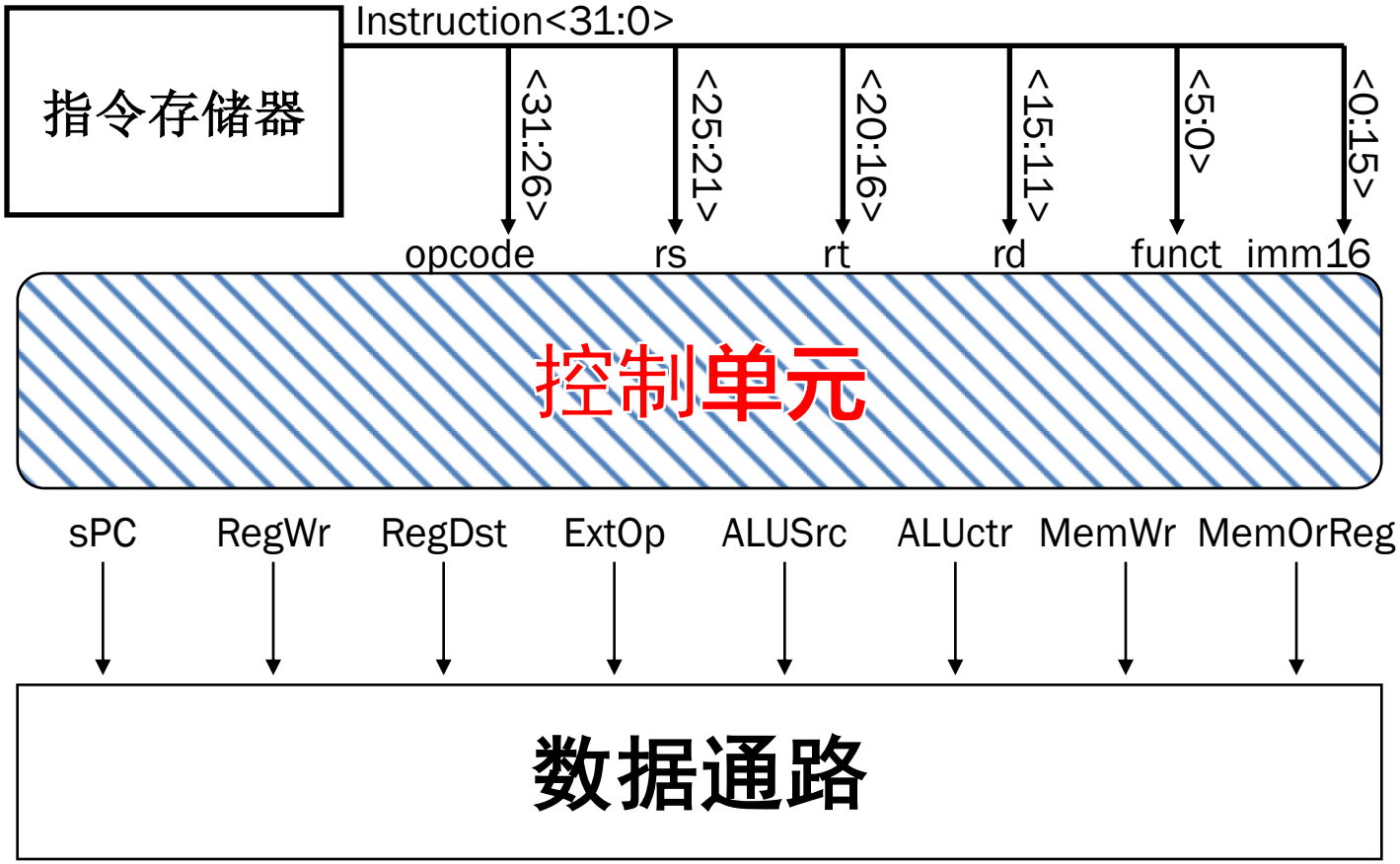
CPU简述

- **CPU的基本功能**
  - 周而复始地执行指令
  - 发现和处理异常情况和中断请求
- **不管CPU多复杂，其基本组成为：**
  - 数据路径（datapath）
  - 控制单元（control unit）

CPU简述

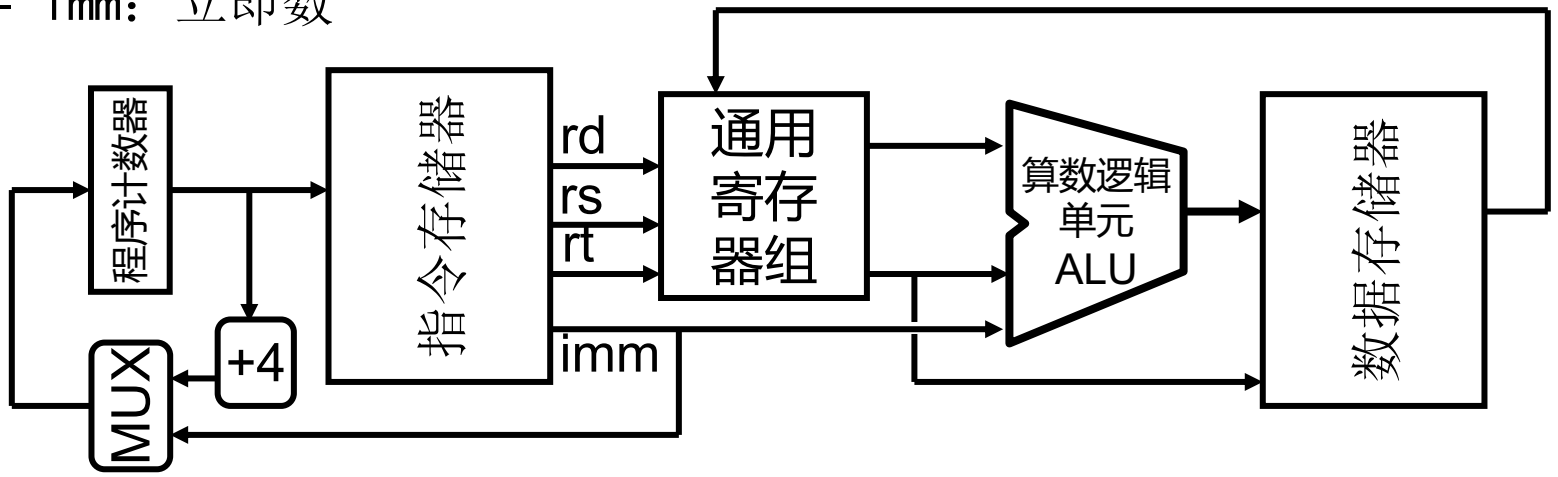
- **通常将指令执行过程中数据所经过的路径及路径上的模块**
  - 包括：ALU、通用寄存器、状态寄存器、cache、MMU、浮点运算模块、异常和中断处理模块
  - 数据路径中专门进行数据运算的模块称为执行单元（execution unit）或者功能单元（function unit）
- **数据通路由控制单元进行控制，控制单元根据每条指令功能的不同生成对数据通路的控制信号，并正确控制指令的执行流程**

CPU简述


指令存储器
Instruction<31:0>
<31:26>
<25:21>
<20:16>
<15:11>
<5:0>
<0:15>
opcode
rs
rt
rd
funct
imm16
控制单元
sPC
RegWr
RegDst
ExtOp
ALUSrc
ALUctr
MemWr
MemOrReg
数据通路

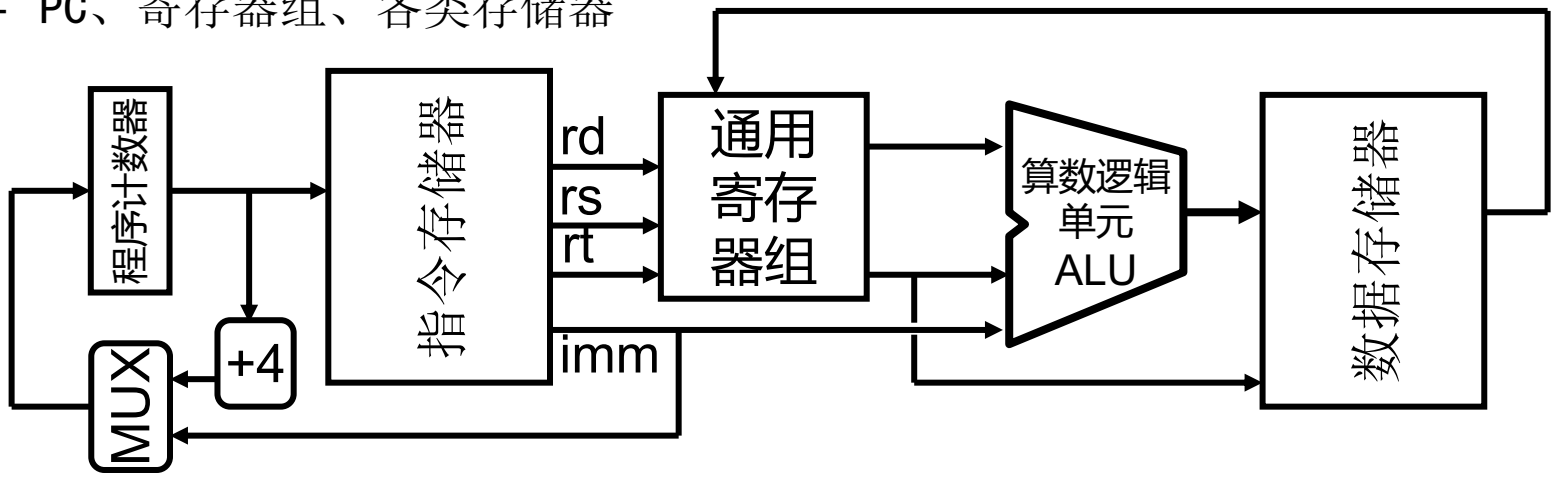
数据通路

- **模块**
  - 程序计数器、指令存储器、通用寄存器组、算数逻辑单元、数据存储器
- **指令**
  - rs：第一个源寄存器的地址码
  - rt：第二个源寄存器的地址码
  - rd：存放结果之目的寄存器的地址码
  - imm：立即数



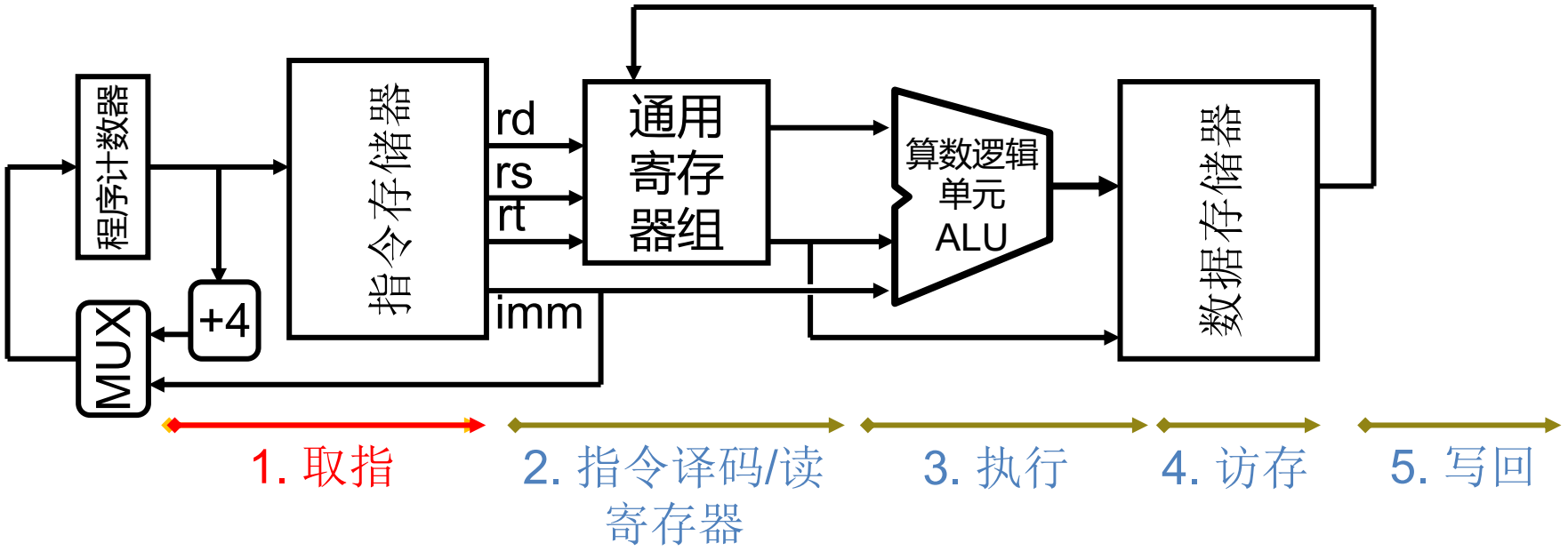
数据通路

- **指令的执行步骤**
  - 取指（IF，*Instruction Fetch*）：从指令存储器中读指令（地址：PC）
  - 指令译码（ID，*Instruction Decode*）：读出一或两个源寄存器的值（通用寄存器组）
  - 执行（EX，*Excute*）：进行指令规定的运算（ALU）
  - 访存（MEM，*Memory Access*）：读/写数据存储器
  - 写回（*WB，Register Write*/Write Back）：将结果写入目的寄存器
- **需要保存的值**
  - PC、寄存器组、各类存储器



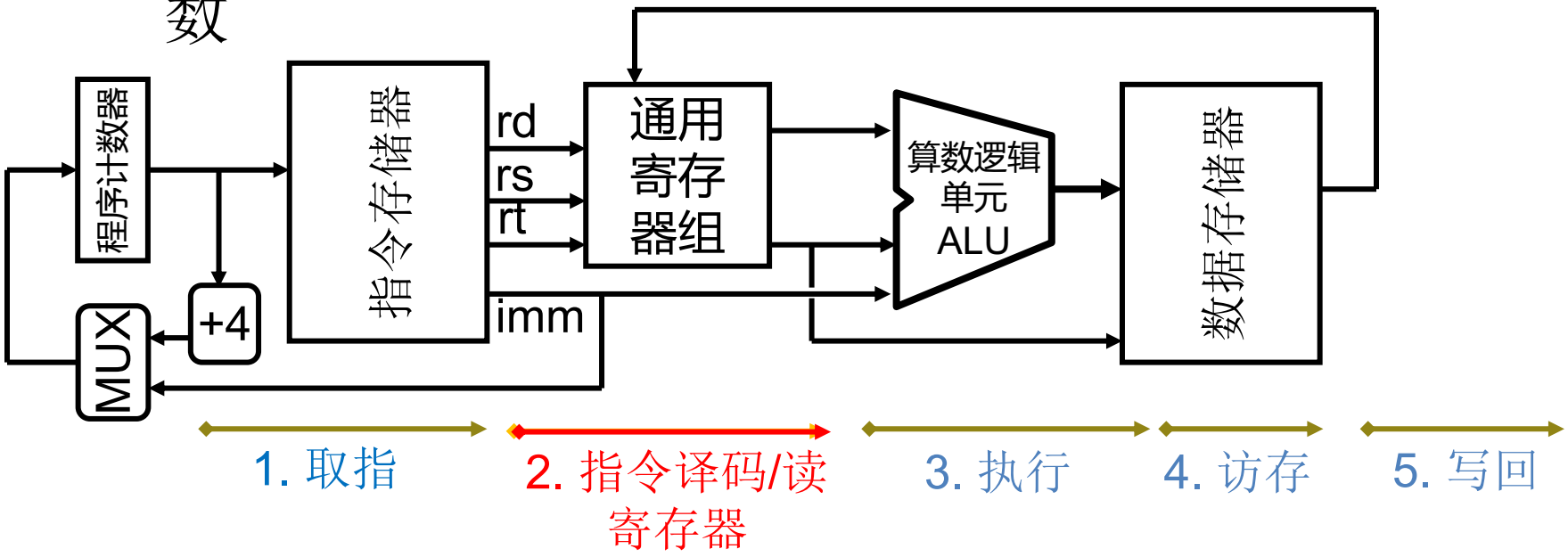
指令的执行步骤——取指

- 阶段1：取指（*IF，Instruction Fetch*）
  - 从内存（暂时不考虑高速缓存cache）中取得（32位）指令
  - 程序计数器PC增加（PC=PC+4）



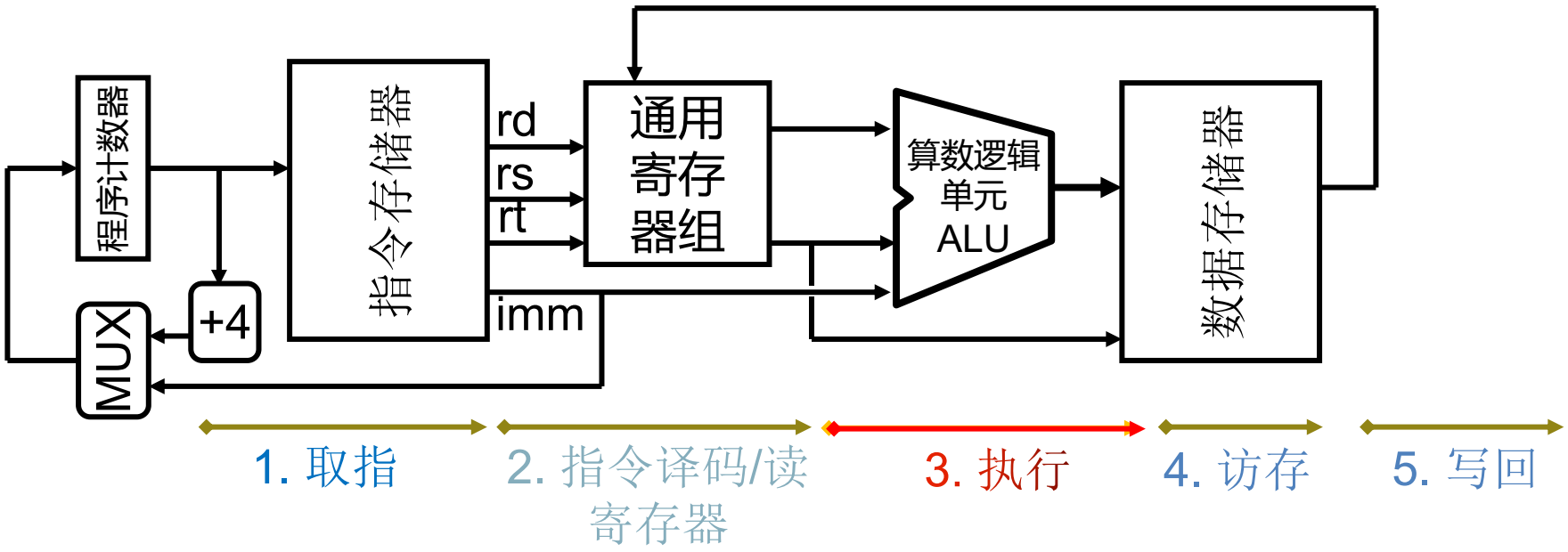
指令的执行步骤——指令译码

- **阶段2：指令译码（*ID，Instruction Decode*）**
  - 从指令中读取指令码和地址码
  - 根据指令码生成指令
  - 根据地址码对应的通用寄存器编号，读出操作数



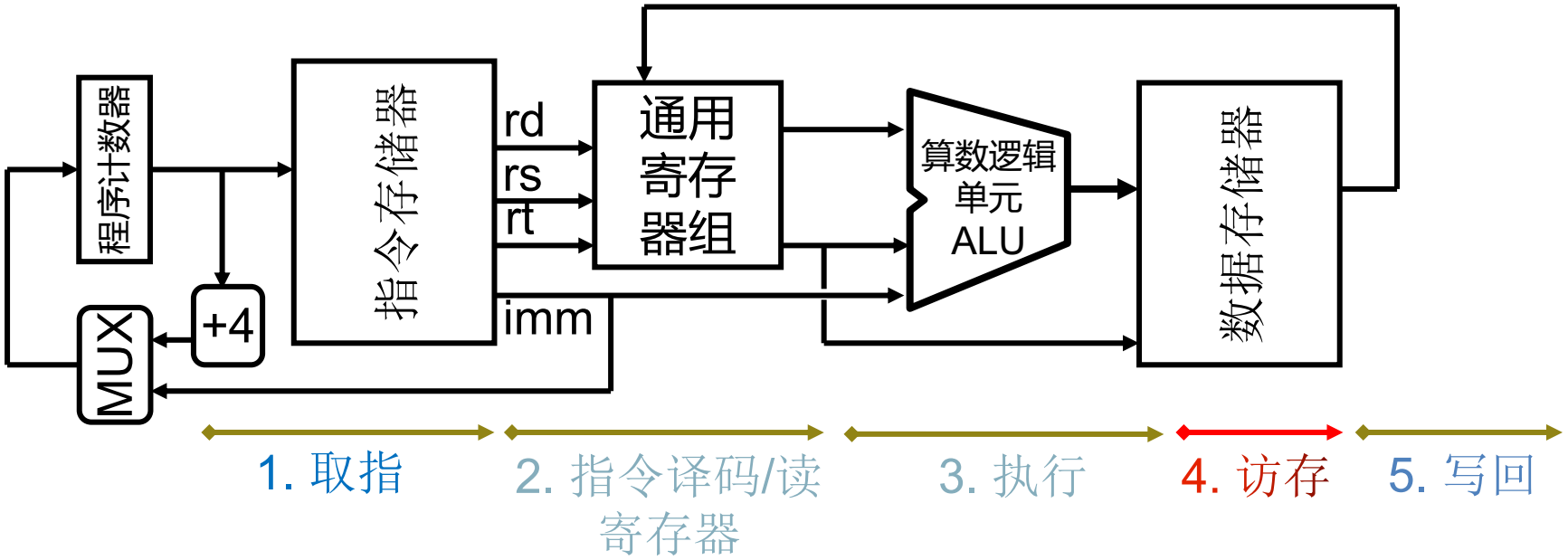
指令的执行步骤——执行

- **阶段3：执行（*EX, Execute*）**
  - 算数逻辑单元（ALU）执行运算，包括算数运算(**+, -, *, /**)、移位、逻辑运算(**&, |**)、比较运算(**slt,==**)
  - 也为load和store计算地址



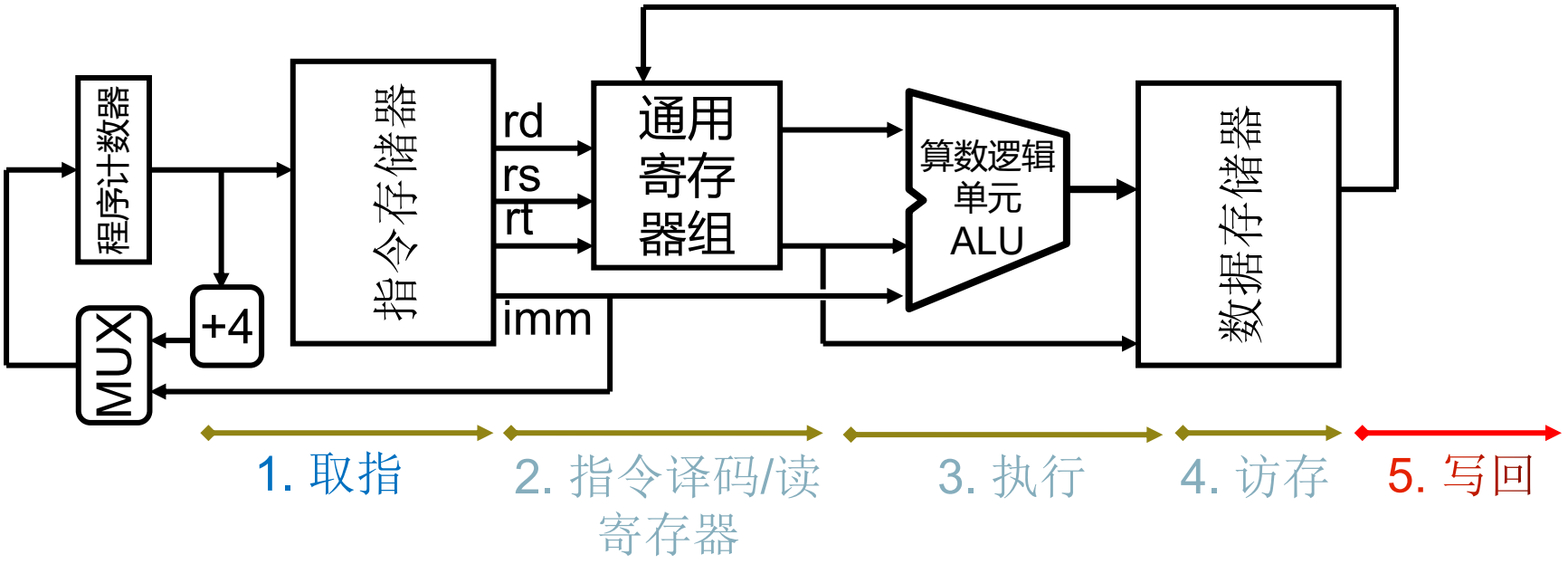
指令的执行步骤——访存

- 阶段4：访存（*MEM，Memory Access*）
  - 在此阶段只执行load/store指令，其他指令都空转（不执行）
  - 可以快速应对cache（高速缓存）操作



指令的执行步骤——写回

- 阶段5：写回（*WB, Register Write*/write back）
  - 将指令结果写入到通用寄存器
  - 有些指令（如e.g. sw，j，beq）将继续空转或者跳过这个阶段



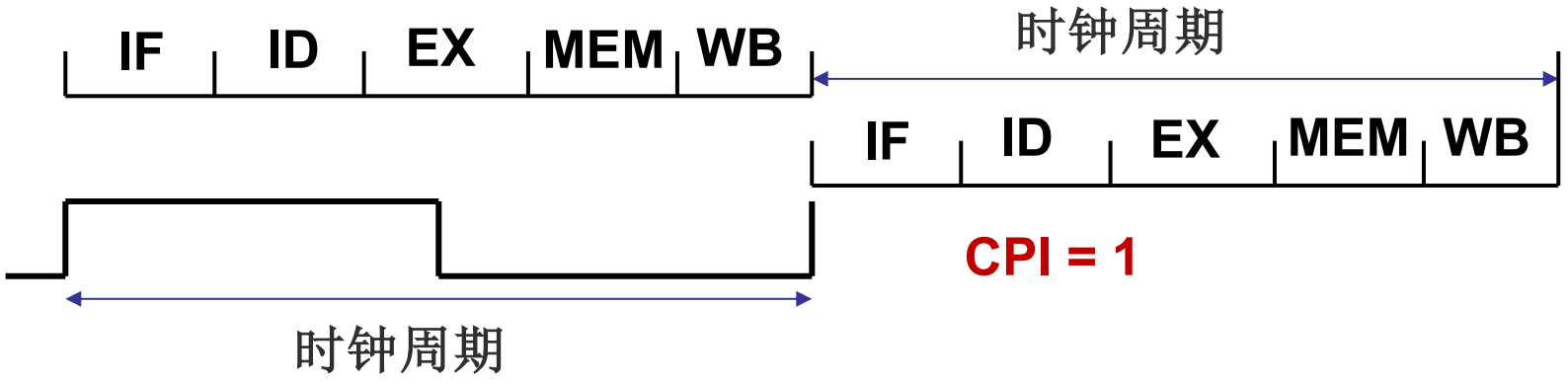
再看CPI

- 指令周期
  - 取指令到指令执行完毕的时间
- CPU周期/机器周期
  - 一条指令的执行过程可从逻辑上划分为若干个阶段，每一阶段完成一项基本操作
  - 完成一个基本操作所需要的时间称为CPU周期/机器周期
  - 通常用内存中读取一个指令的最短时间来规定CPU周期
- 时钟周期
  - 时钟周期也称为振荡周期，定义为时钟频率的倒数
  - 时钟周期是计算机中最基本的、最小的时间单位
- 执行每条指令平均使用的时钟周期个数被称为 CPI（Clock cycle Per Instruction）

一、据通路和指令

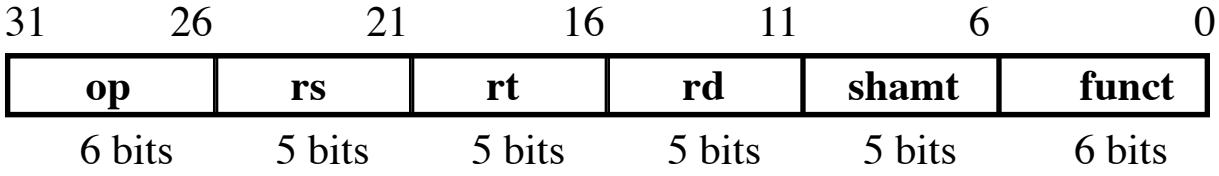
单周期CPU

- 单周期CPU
  - 单个时钟周期内可以完成所有指令
  - 指令串行执行，前一条指令结束后才启动下条指令
  - 每条指令的执行都分为5个阶段
  - 控制各部件运行的信号在整个指令周期不变化
  - 早期计算机采用，系统性能和资源利用率很低，适用于初期开发

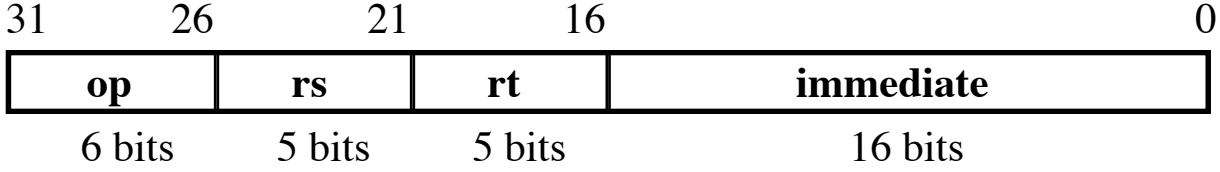


MIPS指令综述

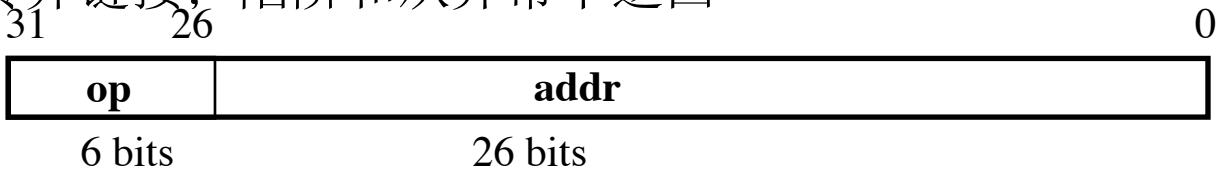
- **寄存器型（R型）指令**
  - 寄存器-寄存器ALU操作，读写专用寄存器



- **立即数型（I型）指令**
  - 加载/存储字节、半字、字、双字，条件分支；跳转，跳转并链接寄存器

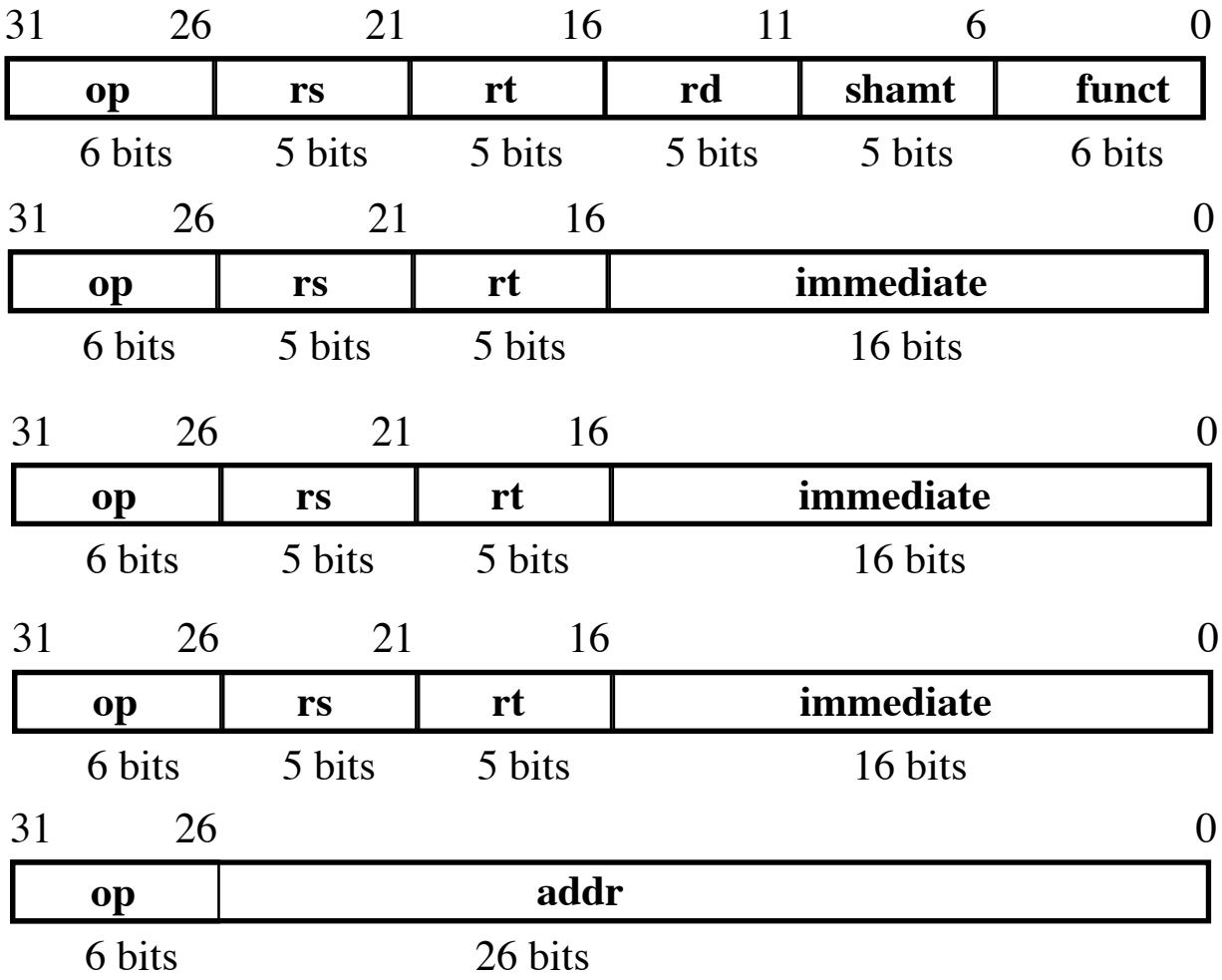


- **跳转型（J型）指令**
  - 跳转，跳转并链接；陷阱和从异常中返回



典型MIPS指令

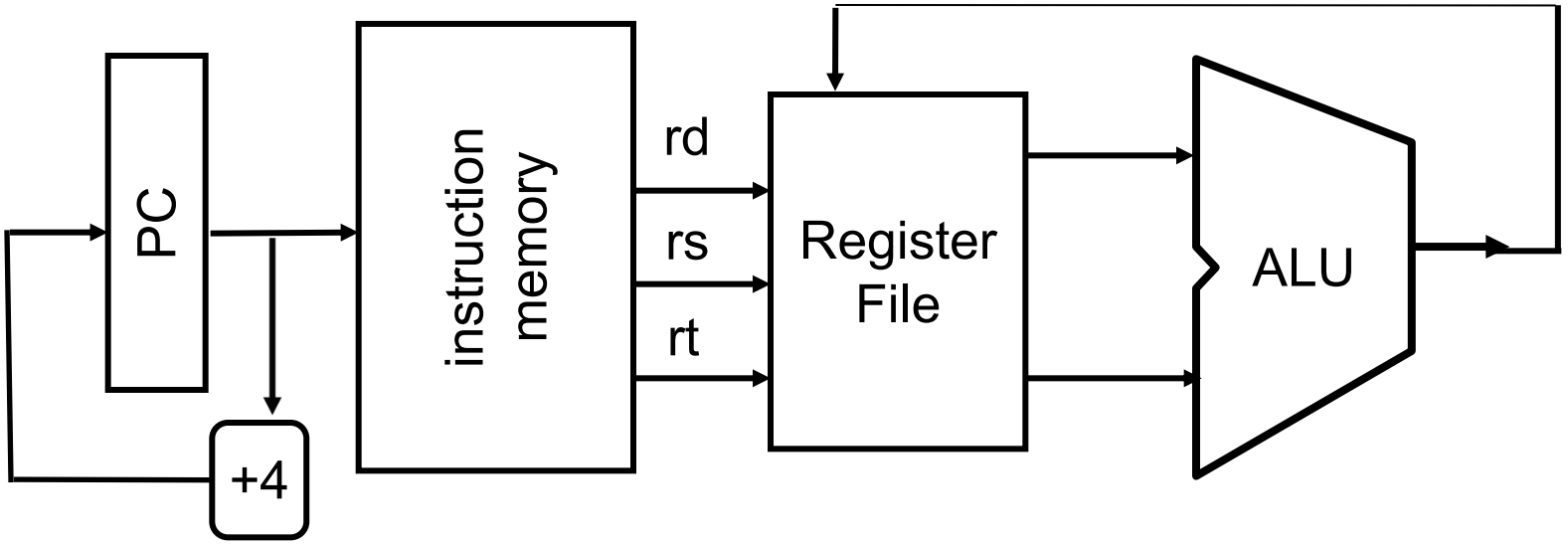
- 加/减
  - ADDU rd,rs,rt
  - SUBU rd,rs,rt
- 逻辑或
  - ORI rt,rs,imm16
- 读/写一个字（word）
  - LW rt,rs,imm16
  - SW rt,rs,imm16
- 比较
  - BEQ rs,rt,imm16
- 跳转
  - J target



addu\subu

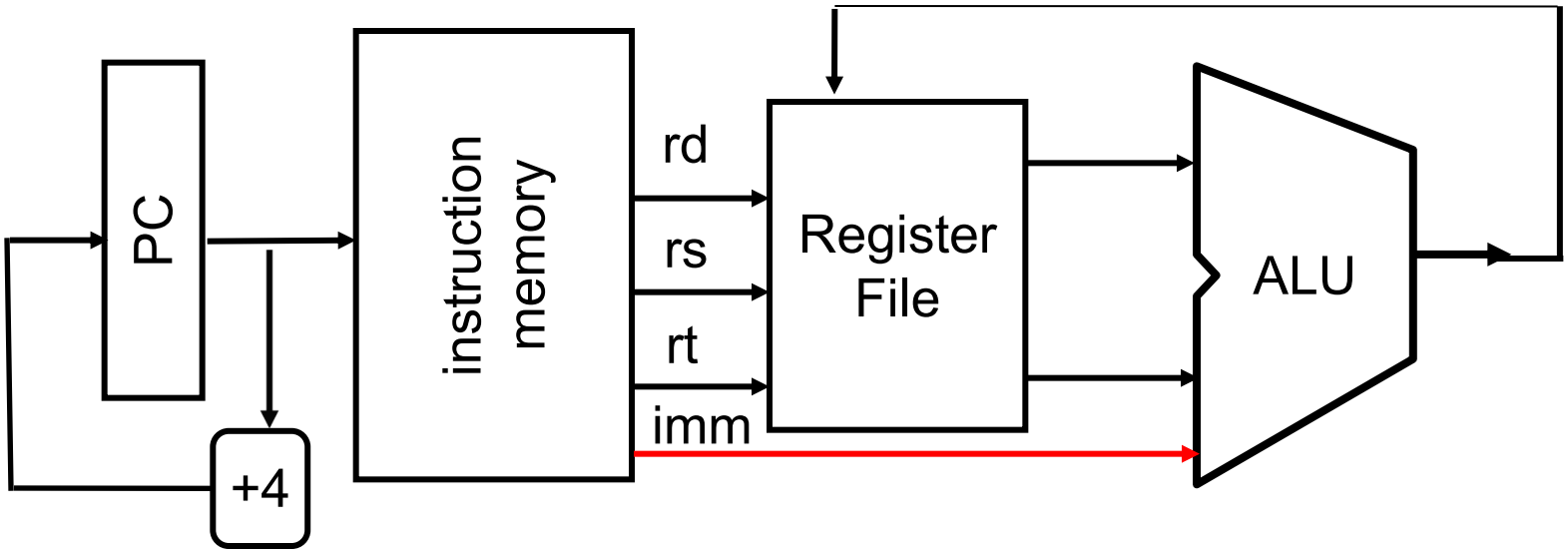
- 算术运算指令ADDU 和 SUBU
  - ADDU rd rs rt
  - SUBU rd rs rt

- 指令功能
  - R[rd] = R[rs] + R[rt]
  - R[rd] = R[rs] - R[rt]



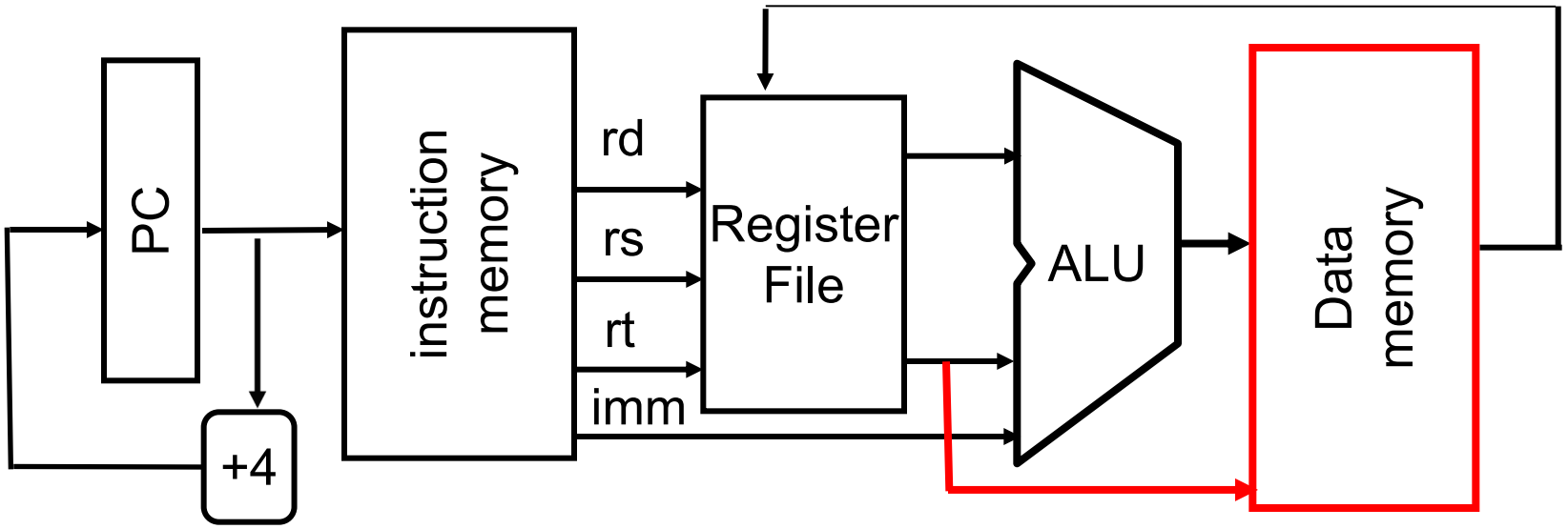
ori

- 逻辑运算ori
  - ORI rt rs imm
- 指令功能
  - R[rt] = R[rs] OR ZeroExt(imm)



Load/Store

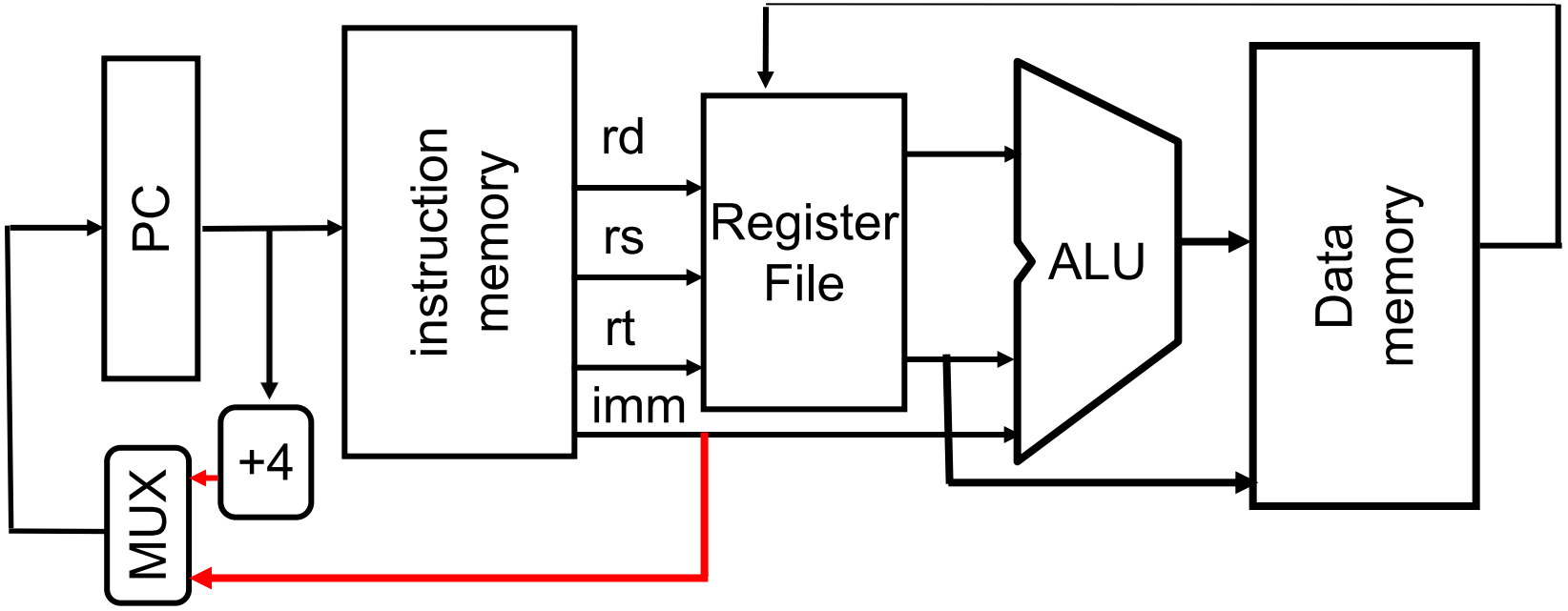
- 读指令 load
  - LW rt rs imm
  - Addr = R[rs] + SignExt(imm)
  - R[rt] = MEM[Addr]
- 写指令store
  - SW rt rs imm
  - Addr = R[rs] + SignExt(imm)
  - MEM[Addr] = R[rt]



BEQ

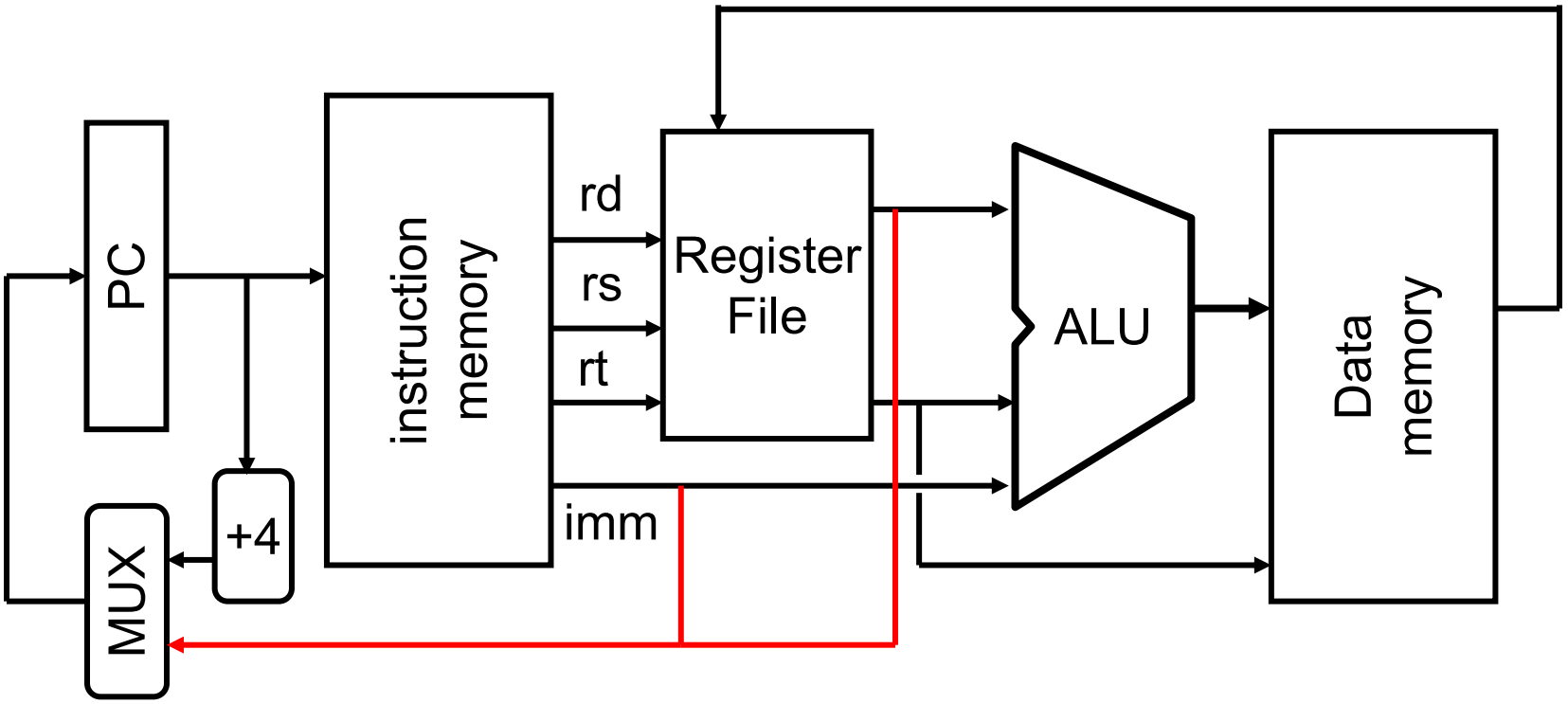
比较指令BEQ

- BEQ rs rt imm
  - if R[rs] = R[rt]
    - then PC ← (PC +4)+ SignExt(imm)
  - Else
    - PC← PC+4



➢ 跳转指令J target

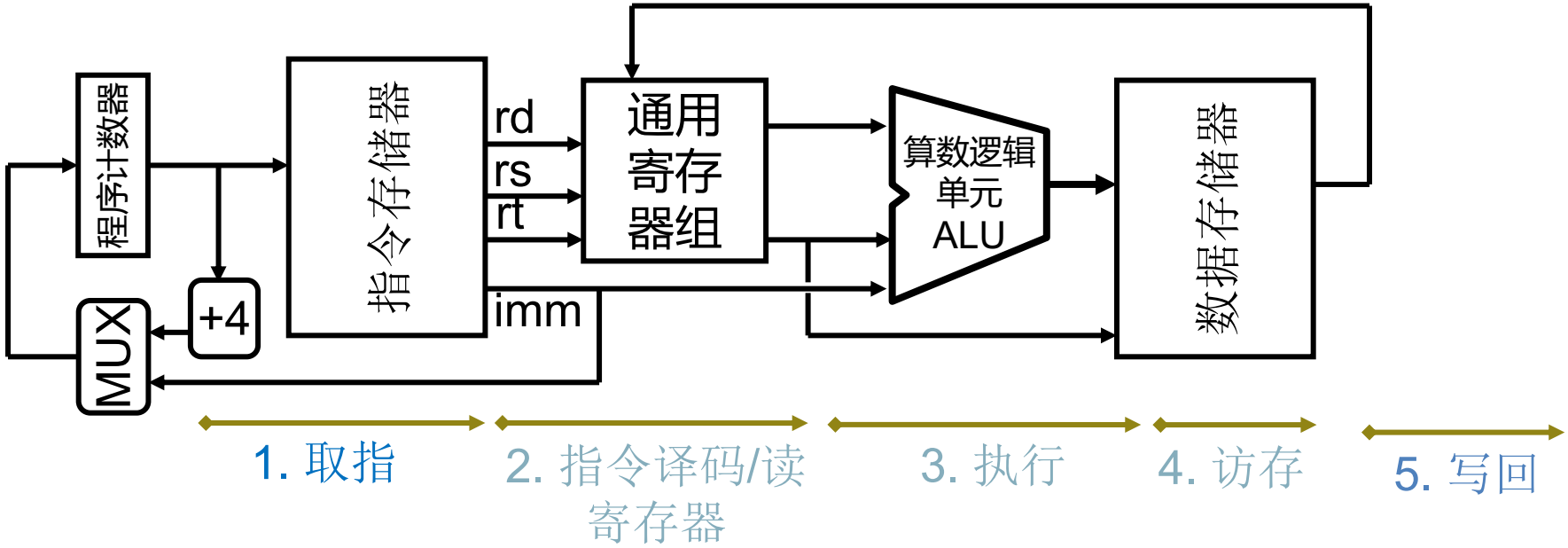
– PC[31:0] ⇐ PC[31:28]～target[25:0]～[00]



二、数据通路的实现

回顾数据通路

- 数据通路的5个阶段

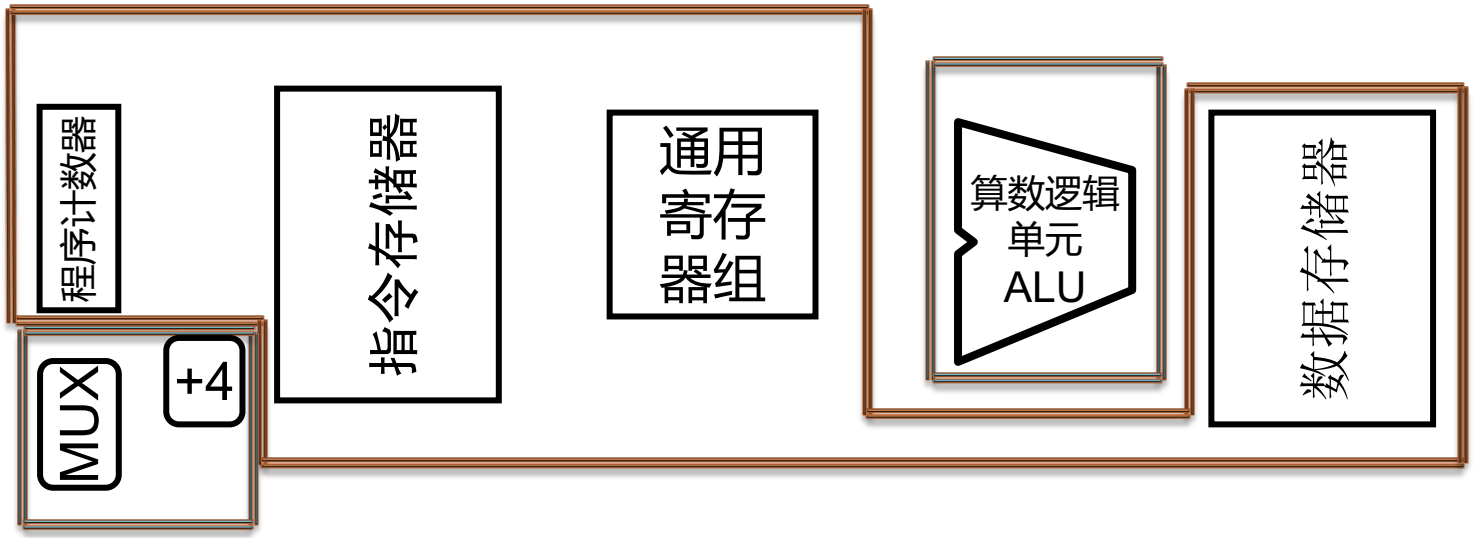


回顾数据通路

- 数据通路的处理模块

回顾数据通路

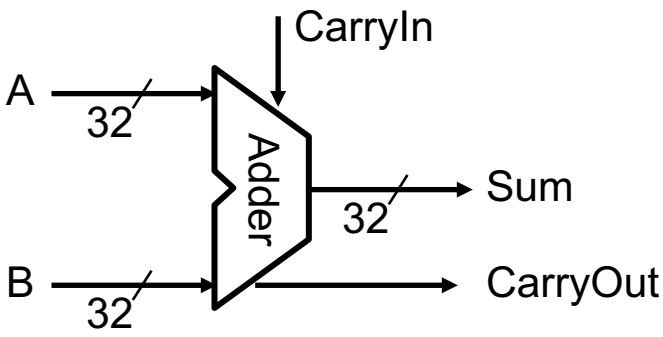
- 数据通路的处理模块



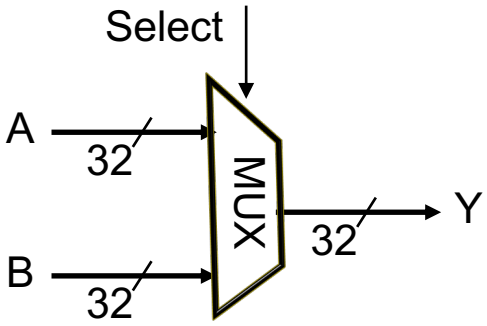
  - 功能模块——组合逻辑
  - 记忆模块——时序逻辑

功能模块

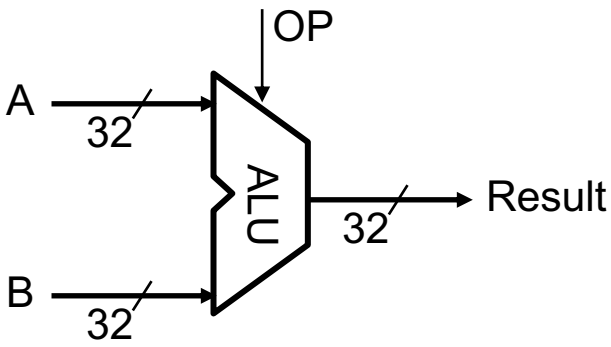
- 加法器
- 乘法器
- 算数逻辑单元ALU



Adder



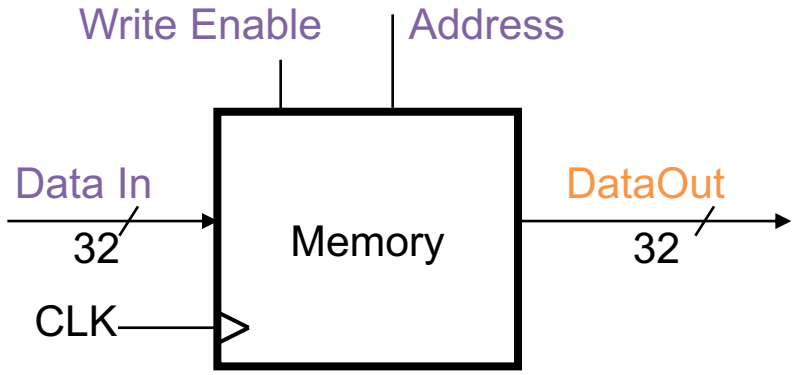
Multiplexer

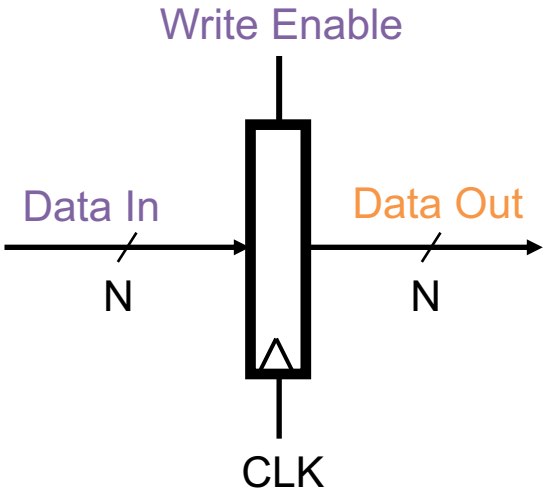


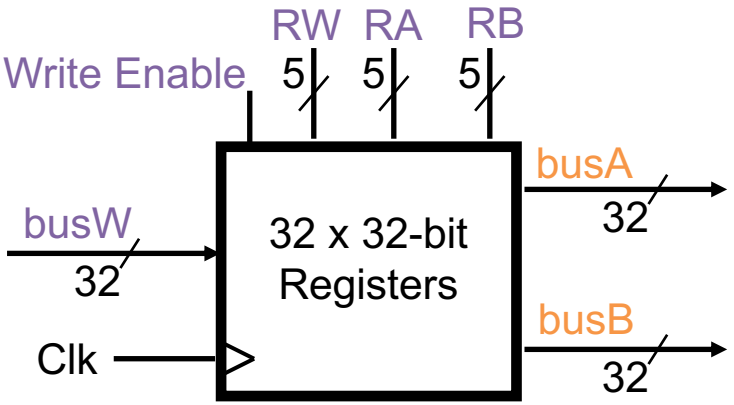
ALU

记忆模块

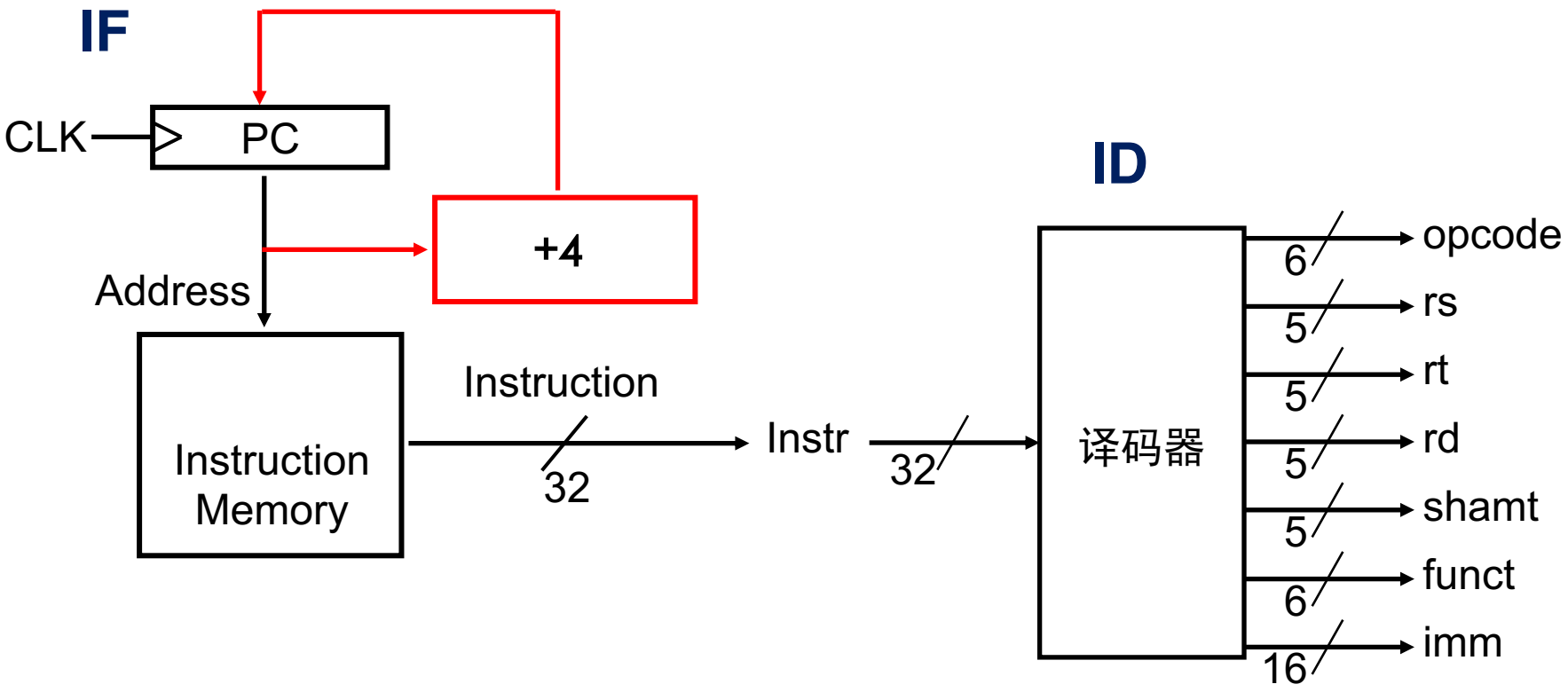
- 主存
- 寄存器
- 寄存器组







取指（IF）和译码（ID）

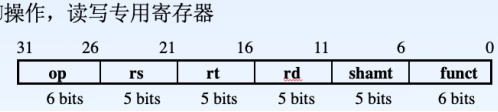

IF
CLK
PC
+4
Address
Instruction Memory
Instruction
32
Instr
32
ID
译码器
6
opcode
5
rs
5
rt
5
rd
5
shamt
6
funct
16
imm

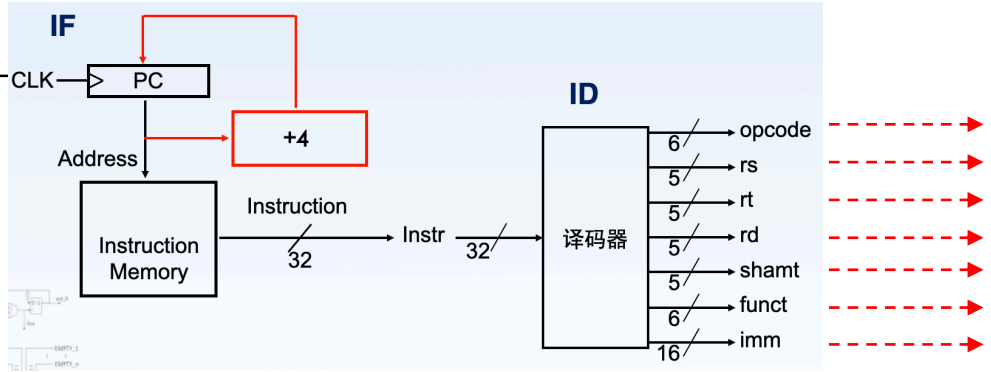
ADDU分析（1）

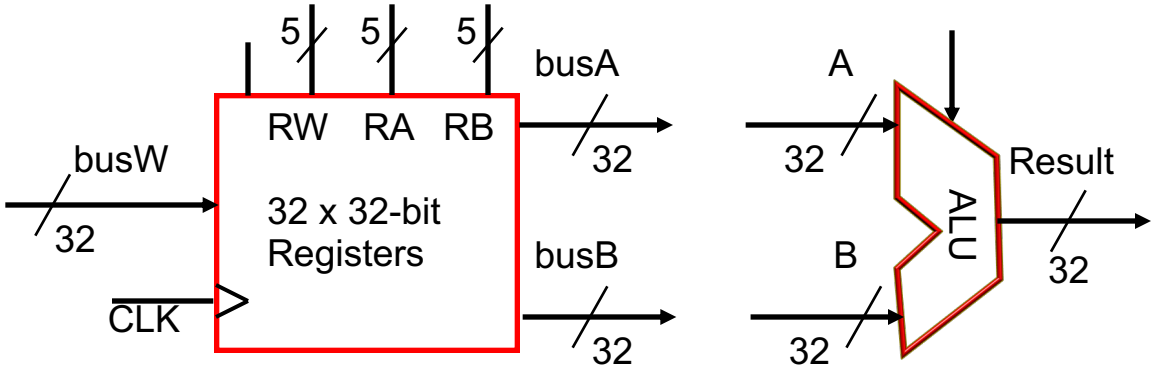
- ADDU
  - 无符号数加法
  - 汇编指令：ADDU R[rd]←
  - 必需的硬件
    - IF和ID
    - 寄存器组
    - ALU

●寄存器型（R型）指令

■寄存器-寄存器ALU操作，读写专用寄存器

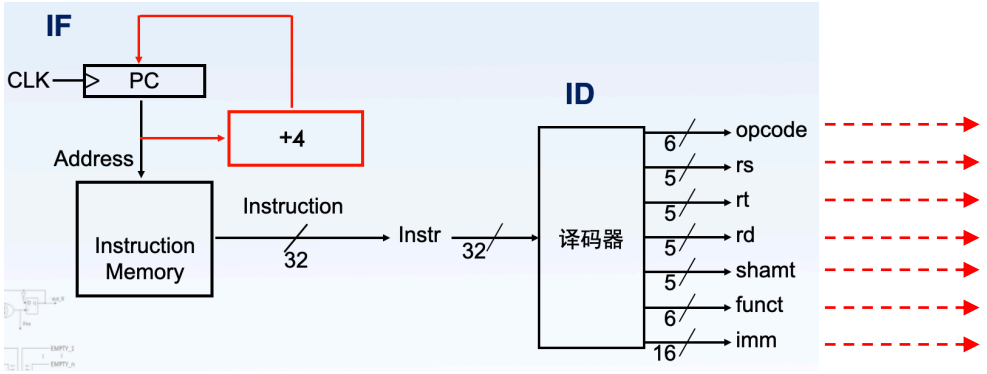




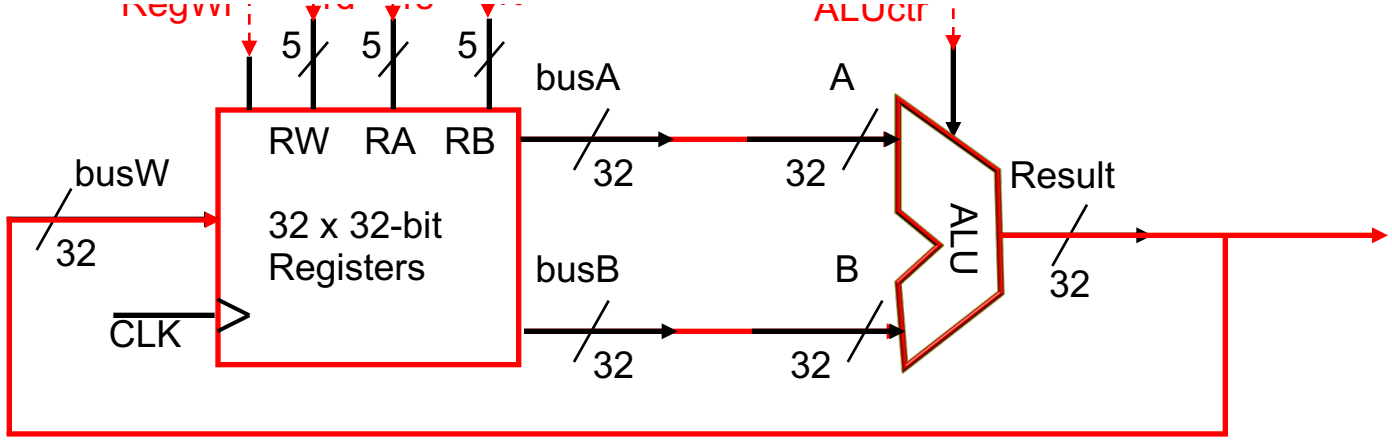


ADDU分析（2）

- 硬件连接：IF/ID与寄存器、IF/ID与ALU、寄存器与ALU
- 需要的模块：寄存器组、ALU
- 控制信号
  - RegWr：写使能
  - ALUctr：加/减/其他
  - rd/rs/rt：寄存器组选择



此结构是否适用于所有指令？



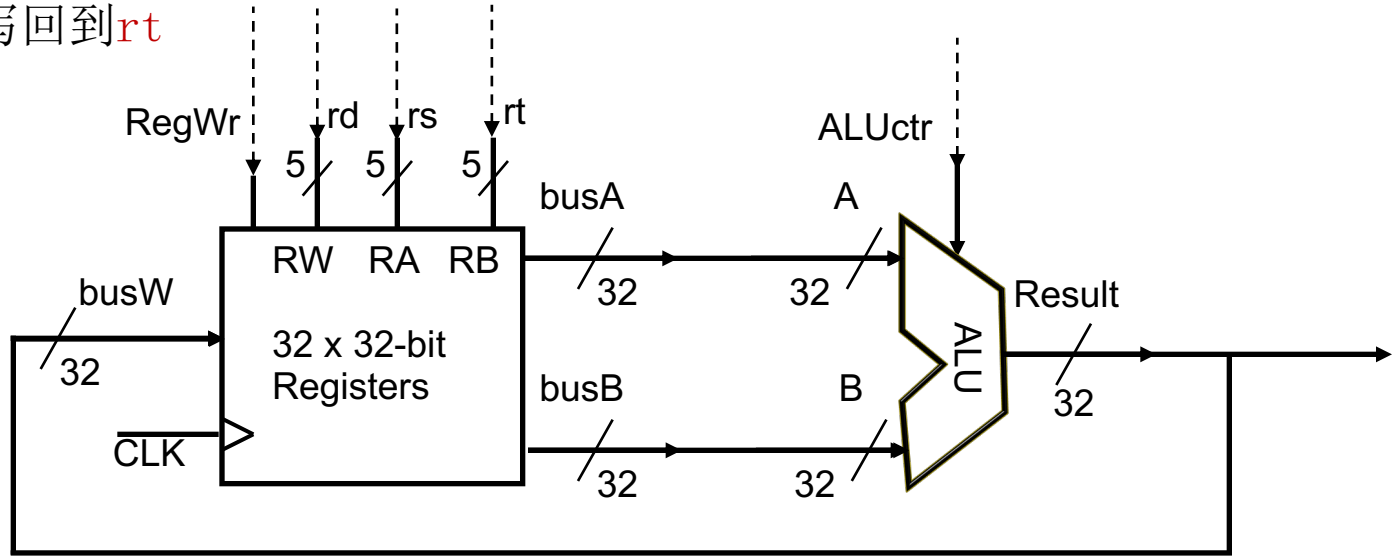
ORI分析（1）

- ORI
  - 按位逻辑或操作
  - 汇编指令：R[rt]←R[rs]|zero_ext(Imm16);
- **下面的结构还适合吗？**
- **缺少：**
  - 16位立即数的补0扩展模块
  - 16位立即数扩展结果到ALU的传输
  - 结果写回到rt

●立即数型（I型）指令

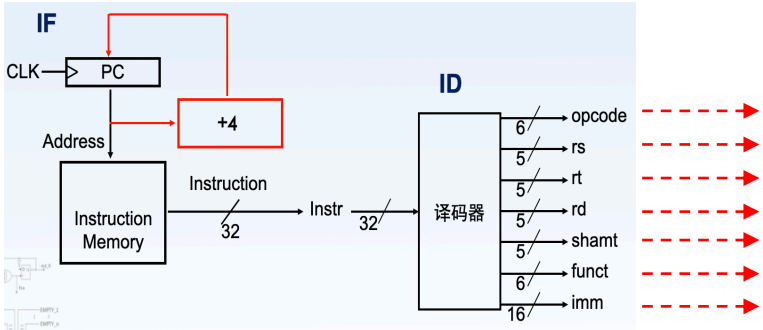
■加载/存储字节、半字、字、双字，条件分支；跳转，跳转并链接寄存器

31	26	21	16	0
op	rs	rt	immediate	
6 bits	5 bits	5 bits	16 bits	

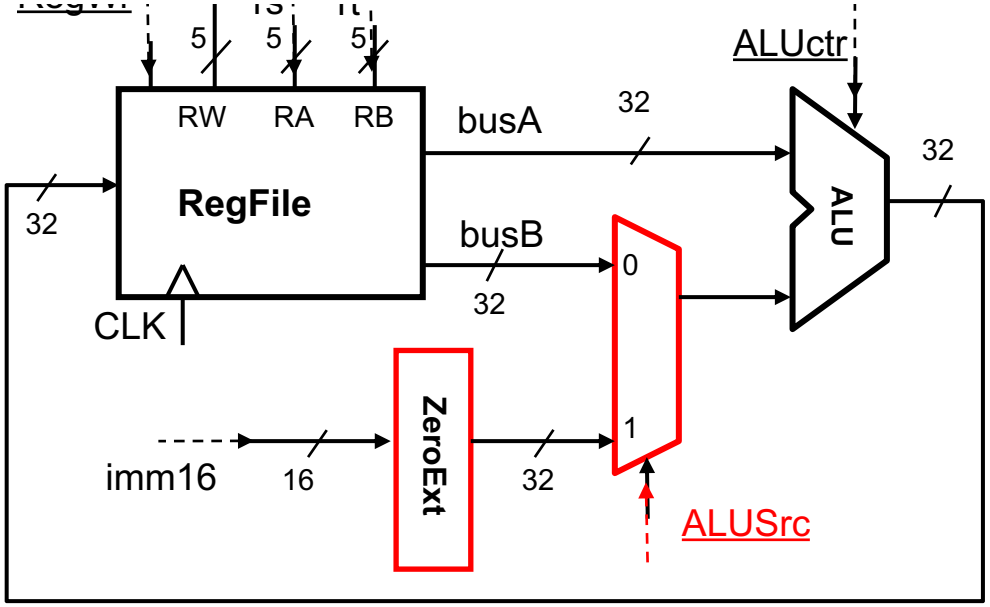


ORI分析（2）

- **新模块：**
  - （无符号）补0扩展
  - 选择器（MUX，多路复用器）
- **新控制（信号）：**
  - RegDst：选择写回结果的寄存器编号
  - ALUSrc：选择ALU输入来源

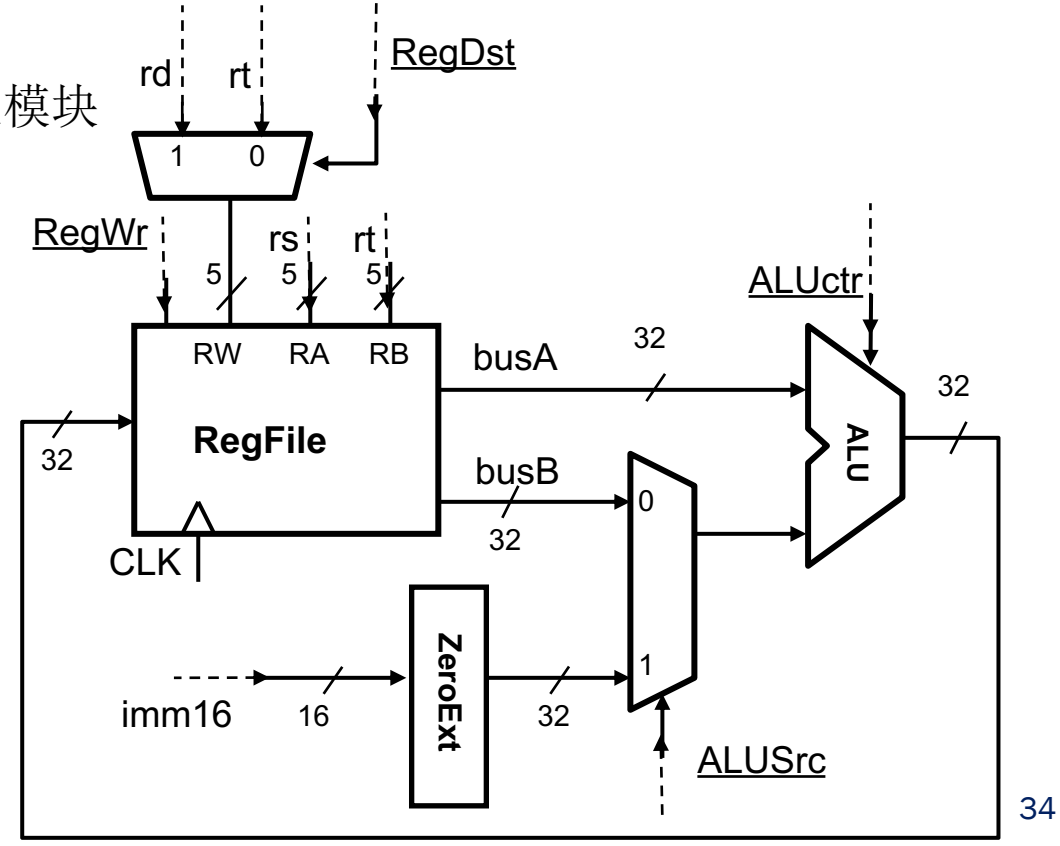


此扩展结构是否适用于所有指令？



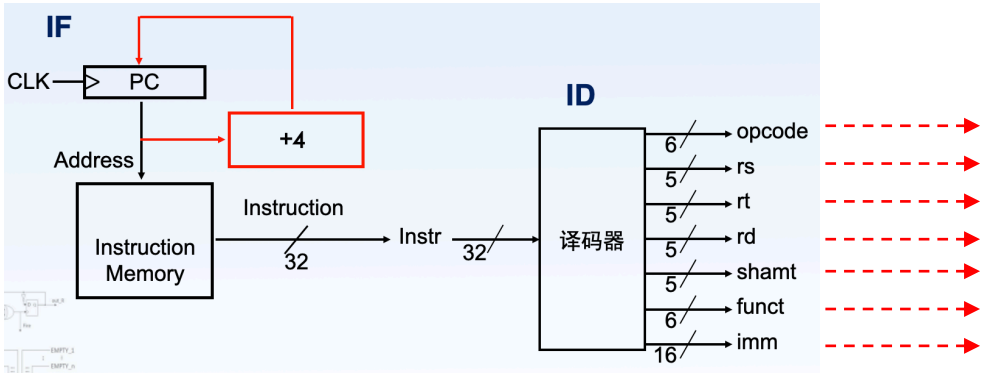
LOAD分析（1）

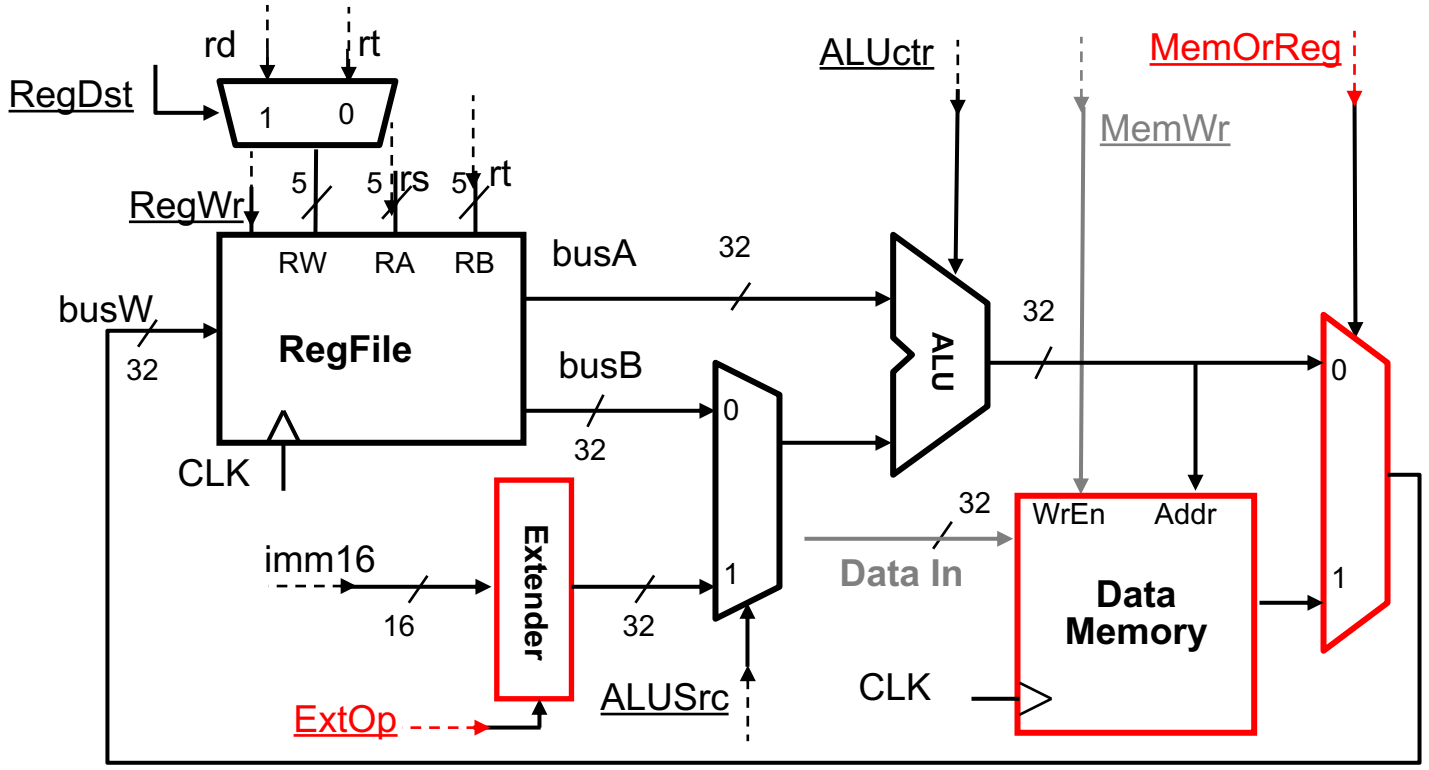
- **读操作数**
  - 从内存中读取操作数
  - 汇编指令：`LOAD R[rt]←MEM[R[rs]+sign_ext(Imm16)];`
- **下面的结构还适合吗？**
- **缺少**
  - 16位立即数的符号扩展模块
  - 主存模块



LOAD分析（2）

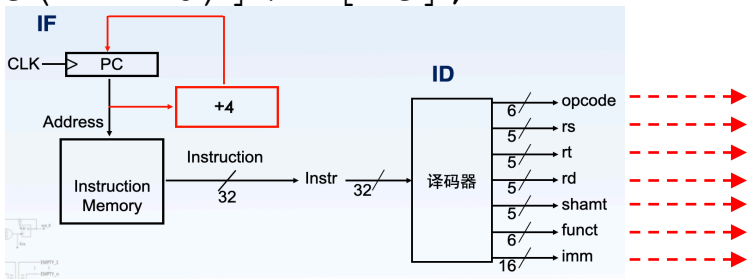
- 新模块：
  - （符号）补0扩展
  - 内存
- 新控制（信号）：
  - MemWr：存储器写使能
  - MemOrReg：操作数源选择信号
    （来自存储器还是ALU结果）
  - ExtOp：扩展方式





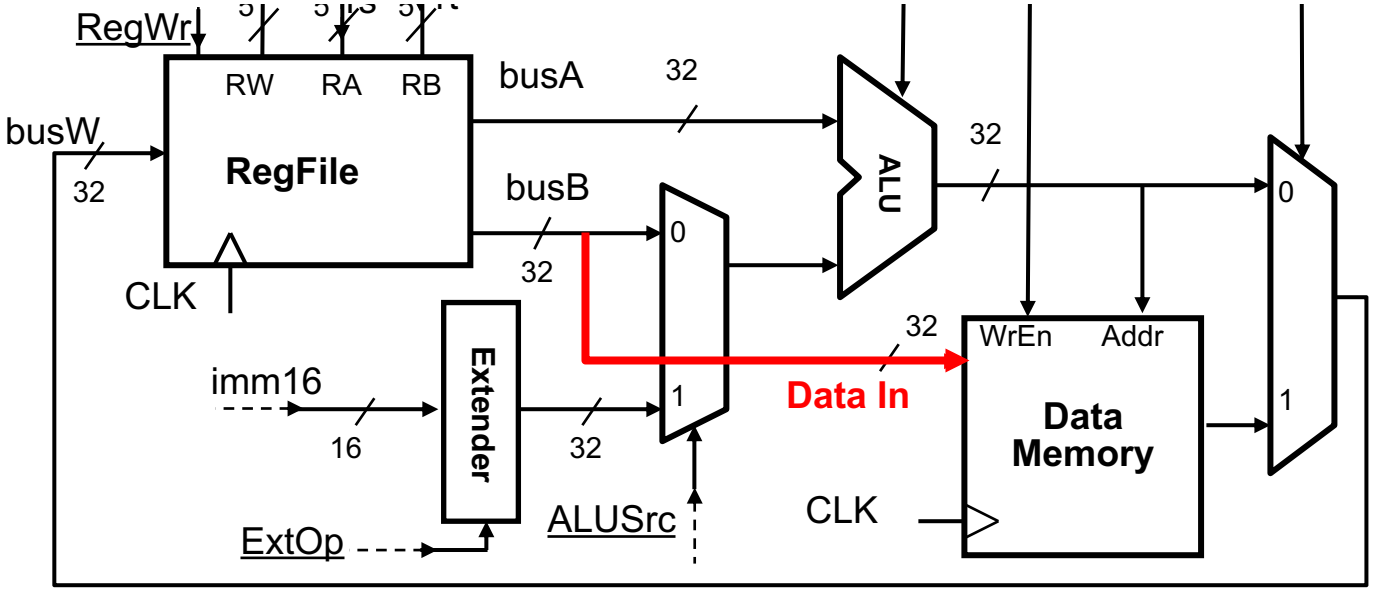
STORE分析

- 写操作数
  - 往内存中写操作数
  - 汇编指令：STORE MEM[R[rs]+sign_ext(Imm16)]←R[rt];
- 新控制（信号）：
  - MemWr：存储器写使能



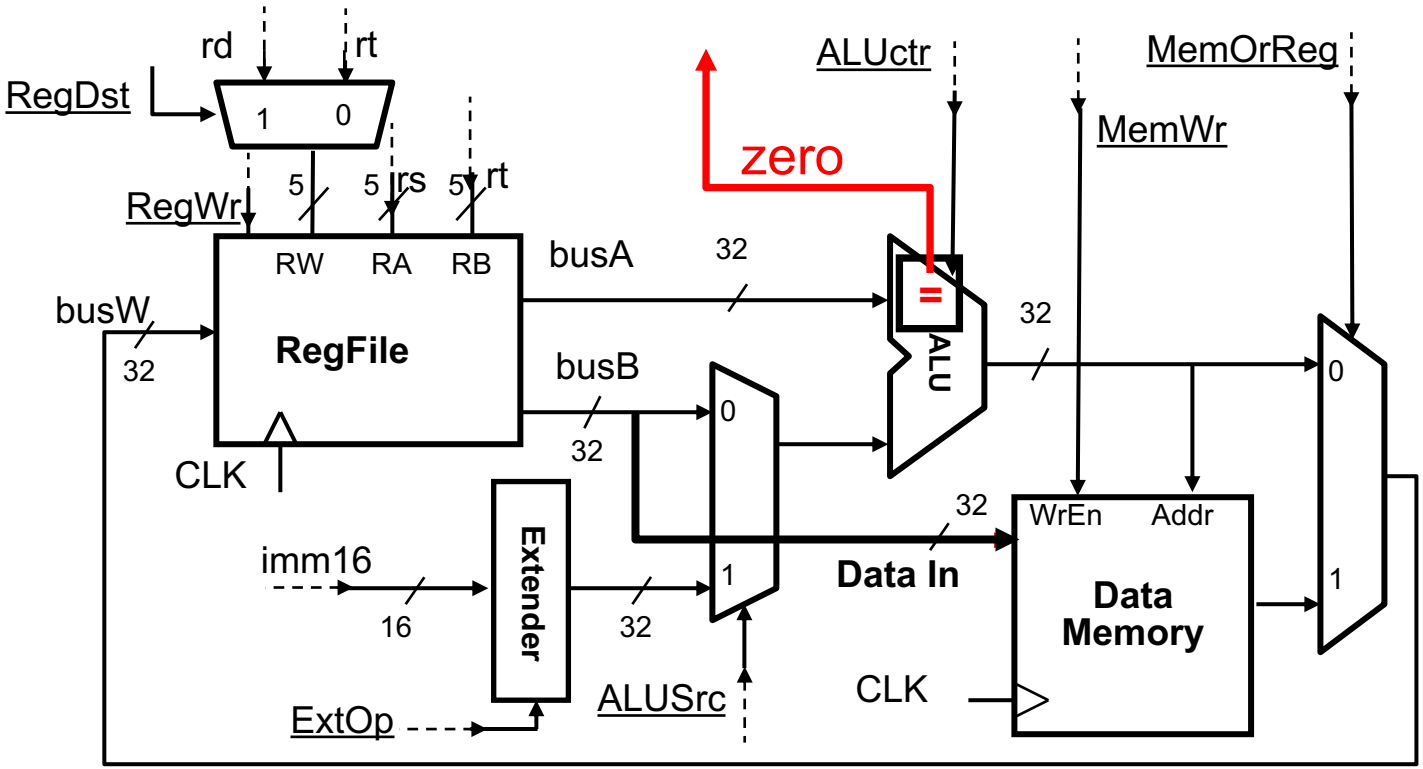
RegDst

此扩展结构是否适用于所有指令？



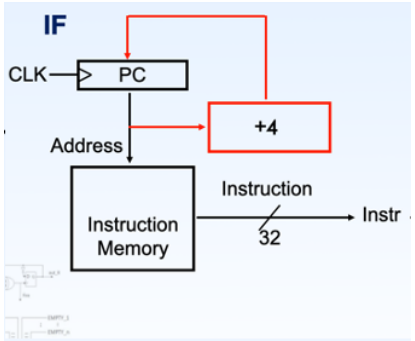
BEQ分析（1）

- **BEQ，比较并跳转**
  - if(R[rs]==R[rt])
    - then PC←PC+4 + (sign_ext(Imm16) || 00)
  - else PC ←PC+4
  - 汇编指令：BEQ rs rt imm
- 下面的结构还适合吗？
- 需要知道ALU的结果才能决定后续PC的操作（取指IF）

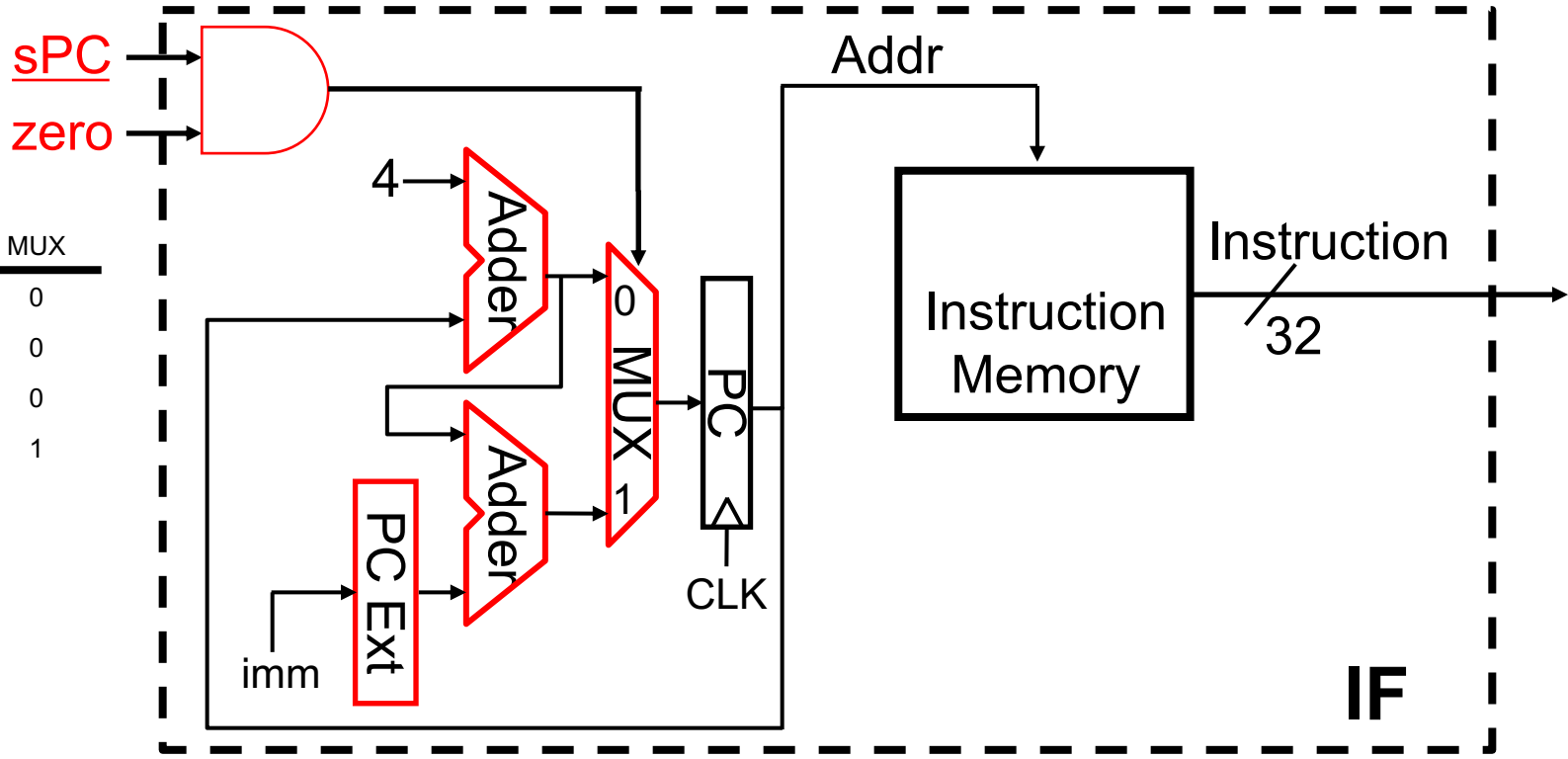


BEQ分析（2）

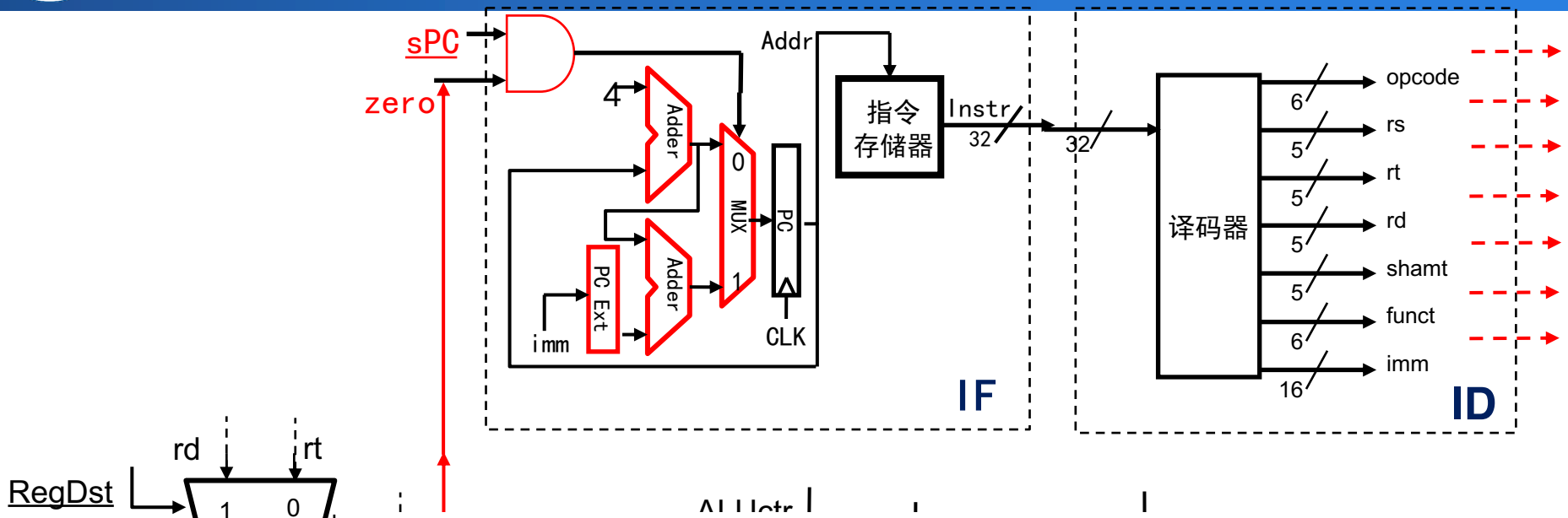
- 精化取指模块（IF）
  - 新模块：加法器、选择器、符号扩展、选择机制
  - 新控制（信号）：
    - sPC：更改PC
    - zero：ALU计算结果为0

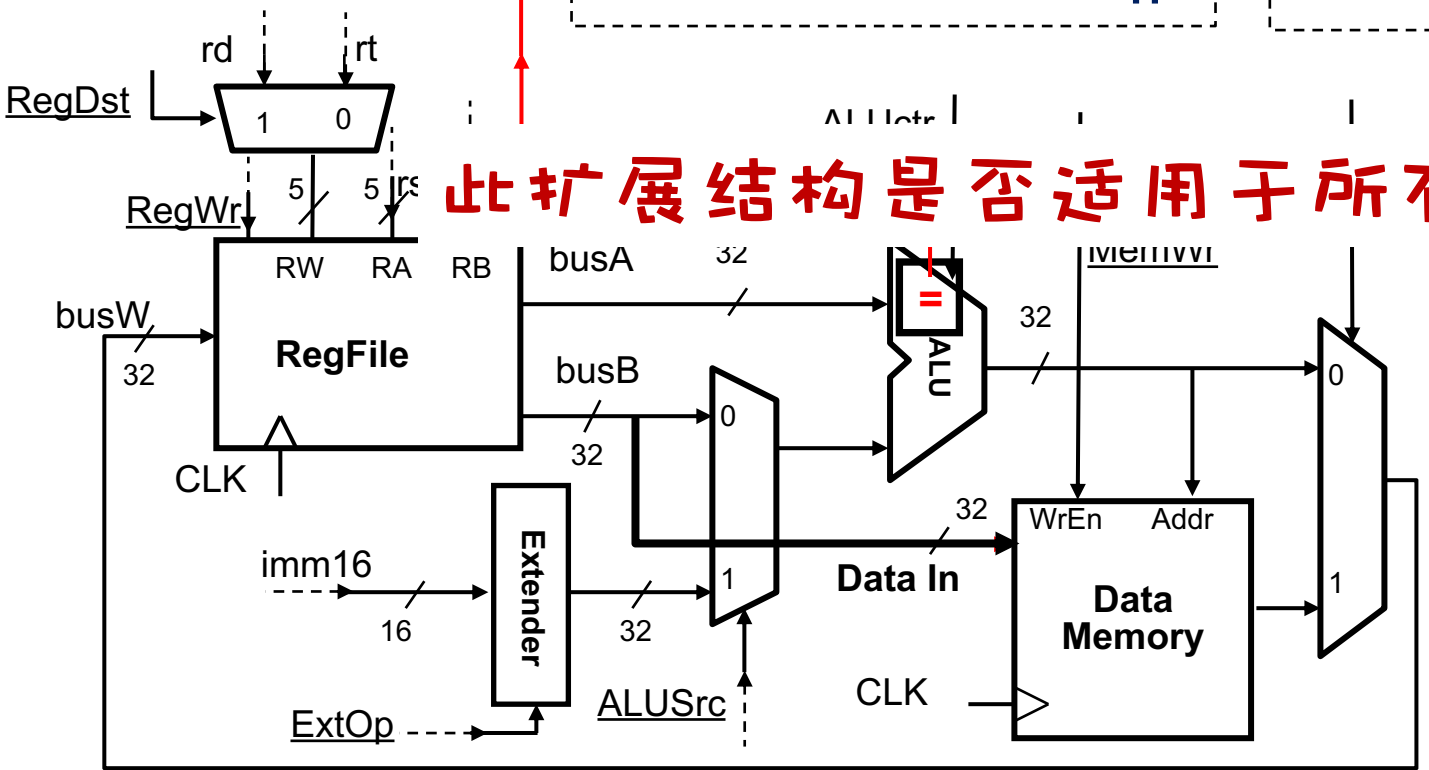


sPC	zero	MUX
0	0	0
0	1	0
1	0	0
1	1	1



BEQ分析（3）





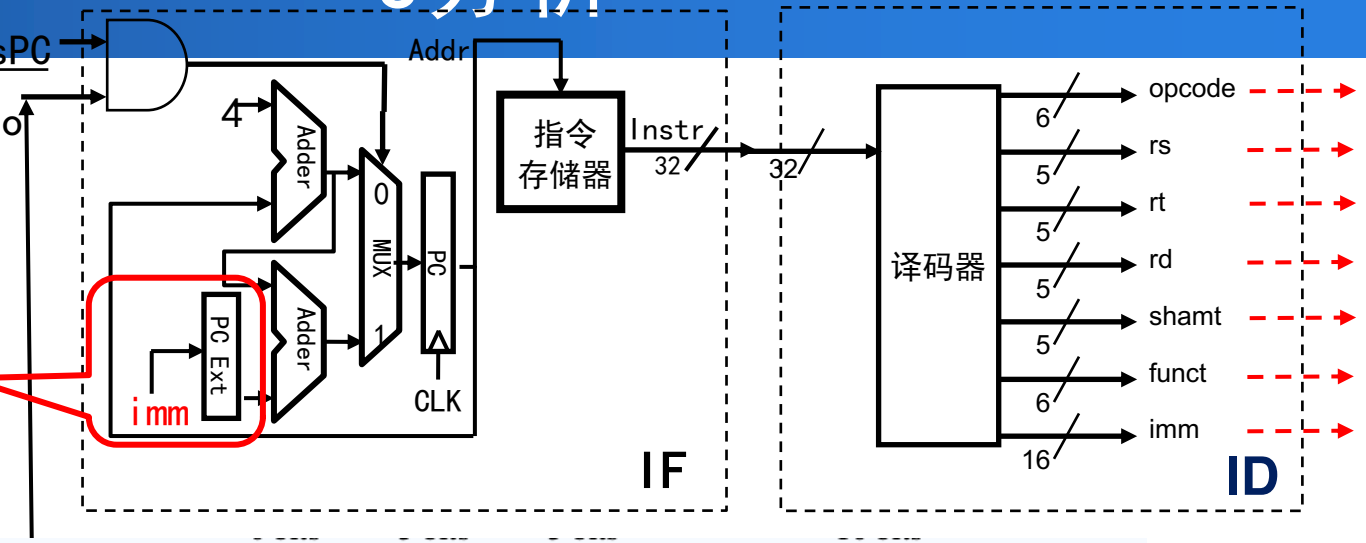
此扩展结构是否适用于所有指令？

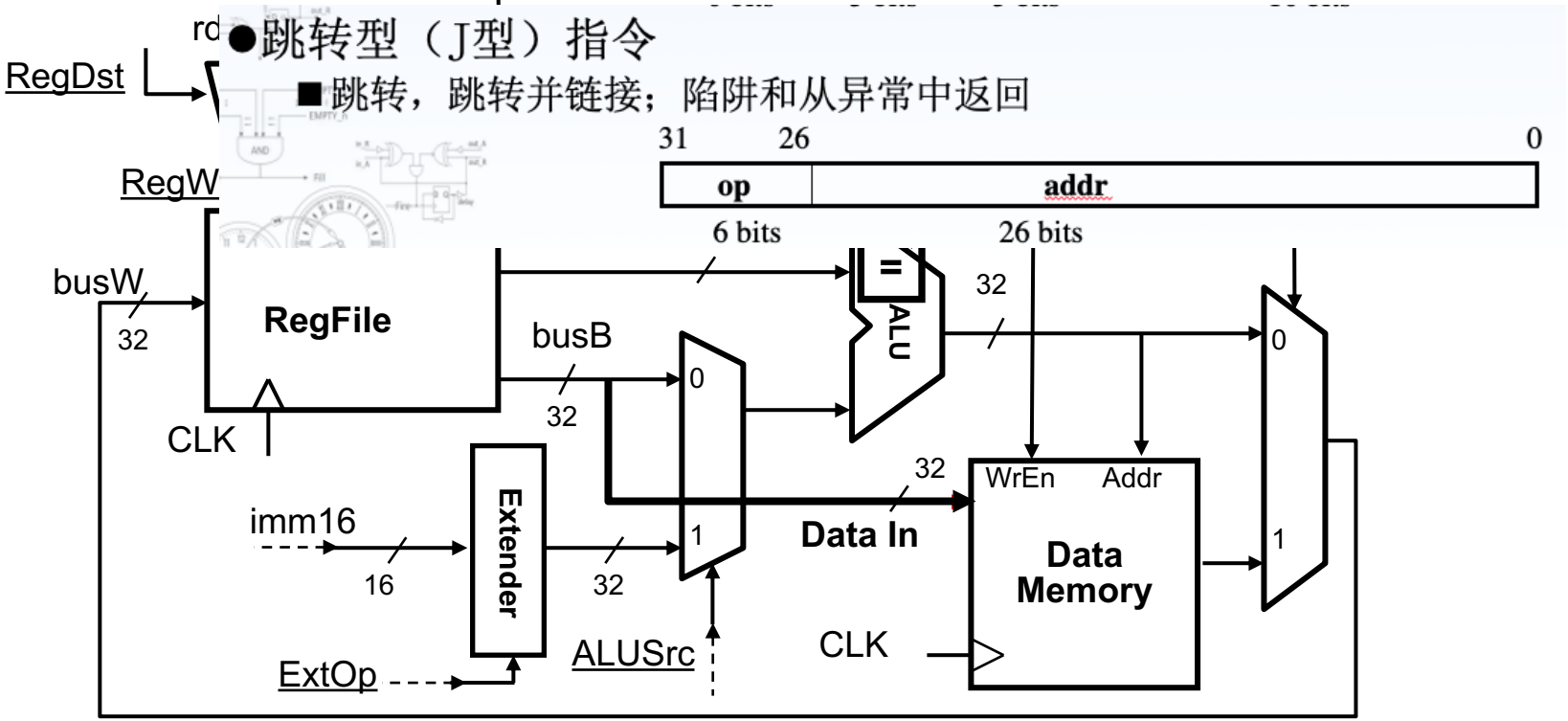
J分析

- J，无条件跳转
  - 26位立即数

PC[31:0]←PC[31:28]||target[25:0]||[00]

如何处理？





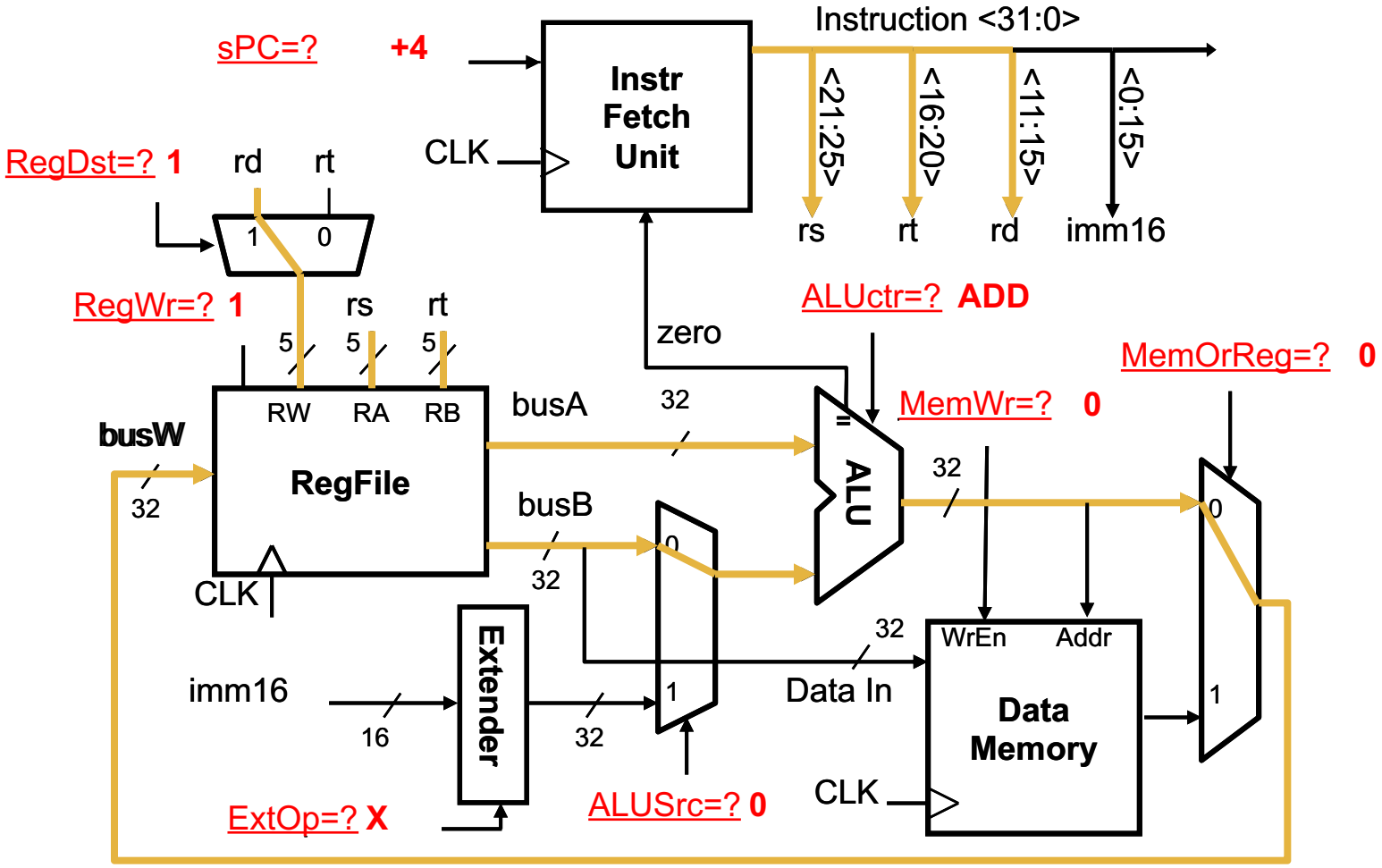
三、控制单元的实现

控制单元的基础假设

- **每条指令占用一个时钟周期**
  - 取指令后分析指令，并给出整个执行期间的全部信号
  - 不需要状态信息，在时钟结束的边沿写入结果
- **控制对象**
  - ALU的运算
  - 寄存器组和存储器的写入
  - 多路选择器
  - 扩展计算等新模块
- **时钟周期开始时读取指令**

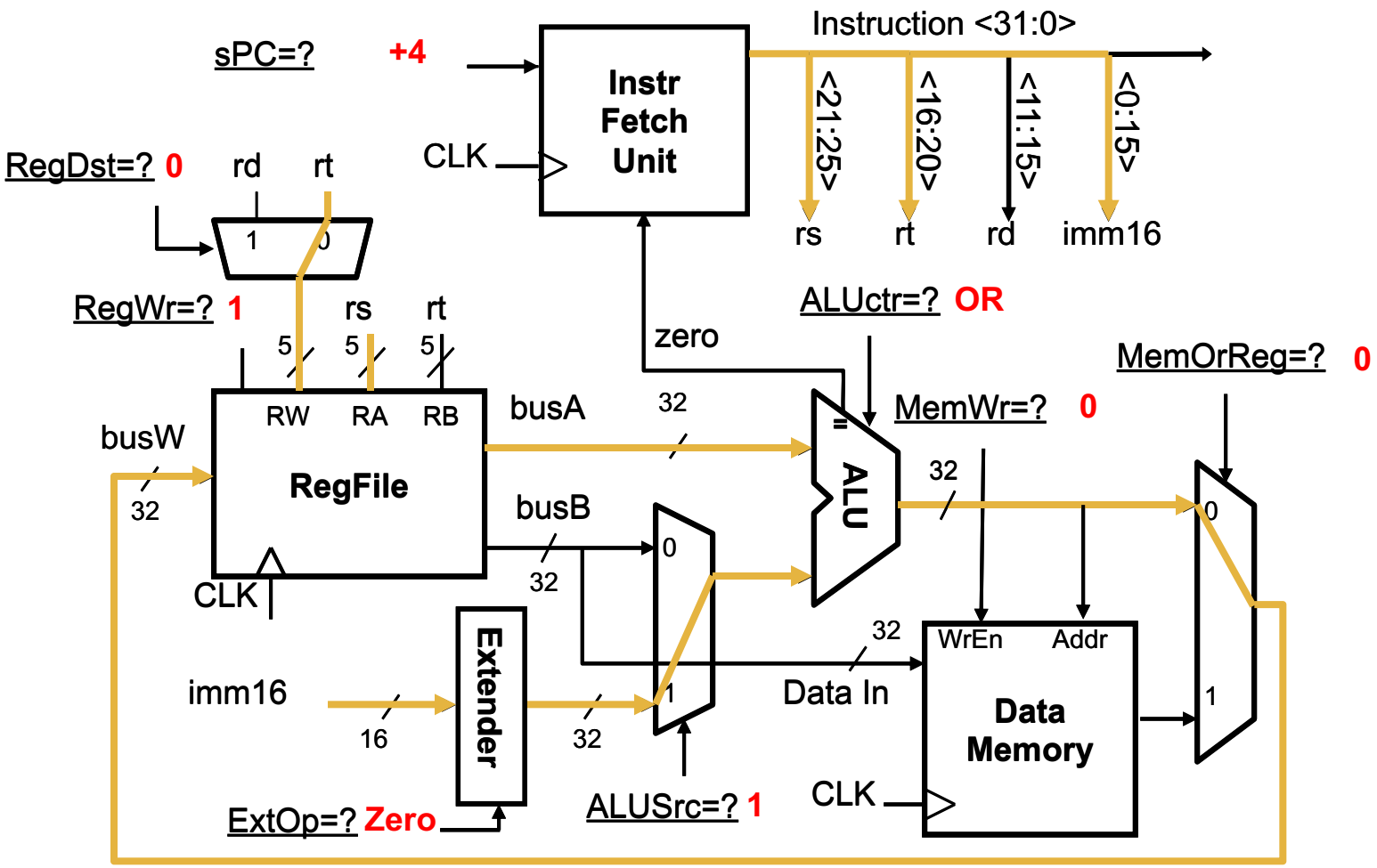
R型指令的控制机制

R[rd]←R[rs]+R[rt];



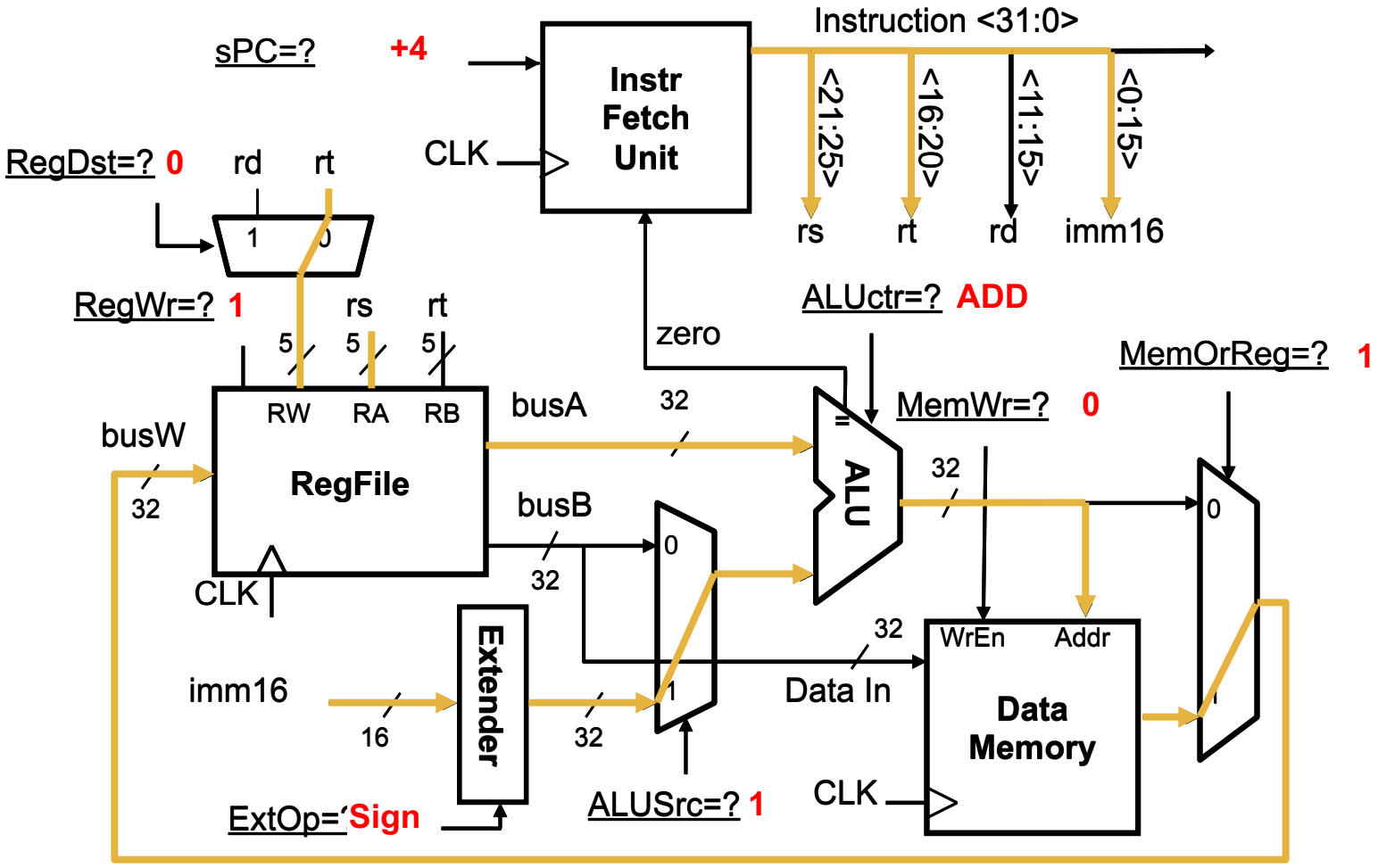
I 型指令的控制机制（1）

R[rt]←R[rs]|ZeroExt(imm16);



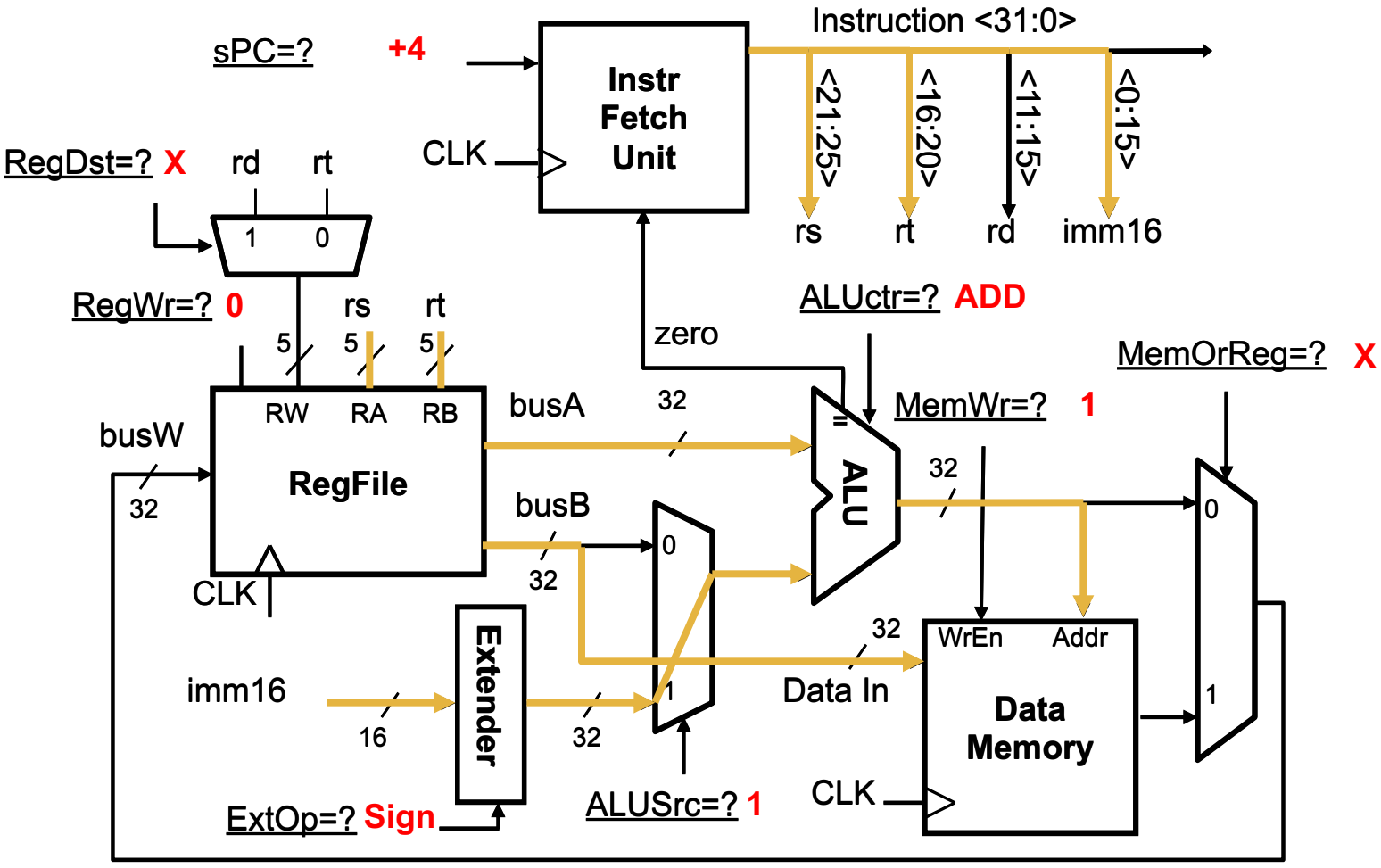
I 型指令的控制机制（2）

R[rt]←MEM{R[rs]+SignExt[imm16]};



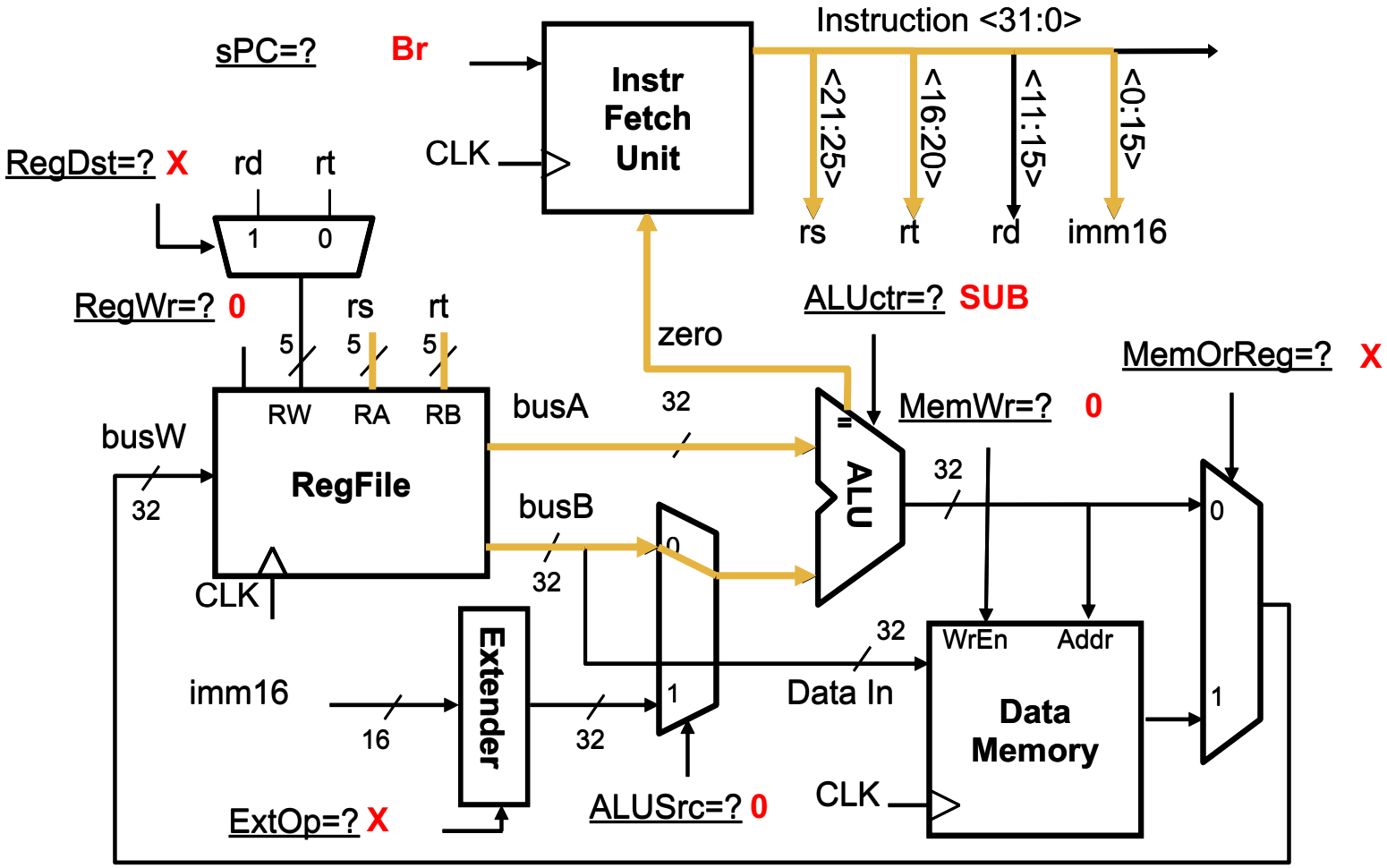
I 型指令的控制机制（3）

- **MEM{R[rs]+SignExt[imm16]}←R[rt];**



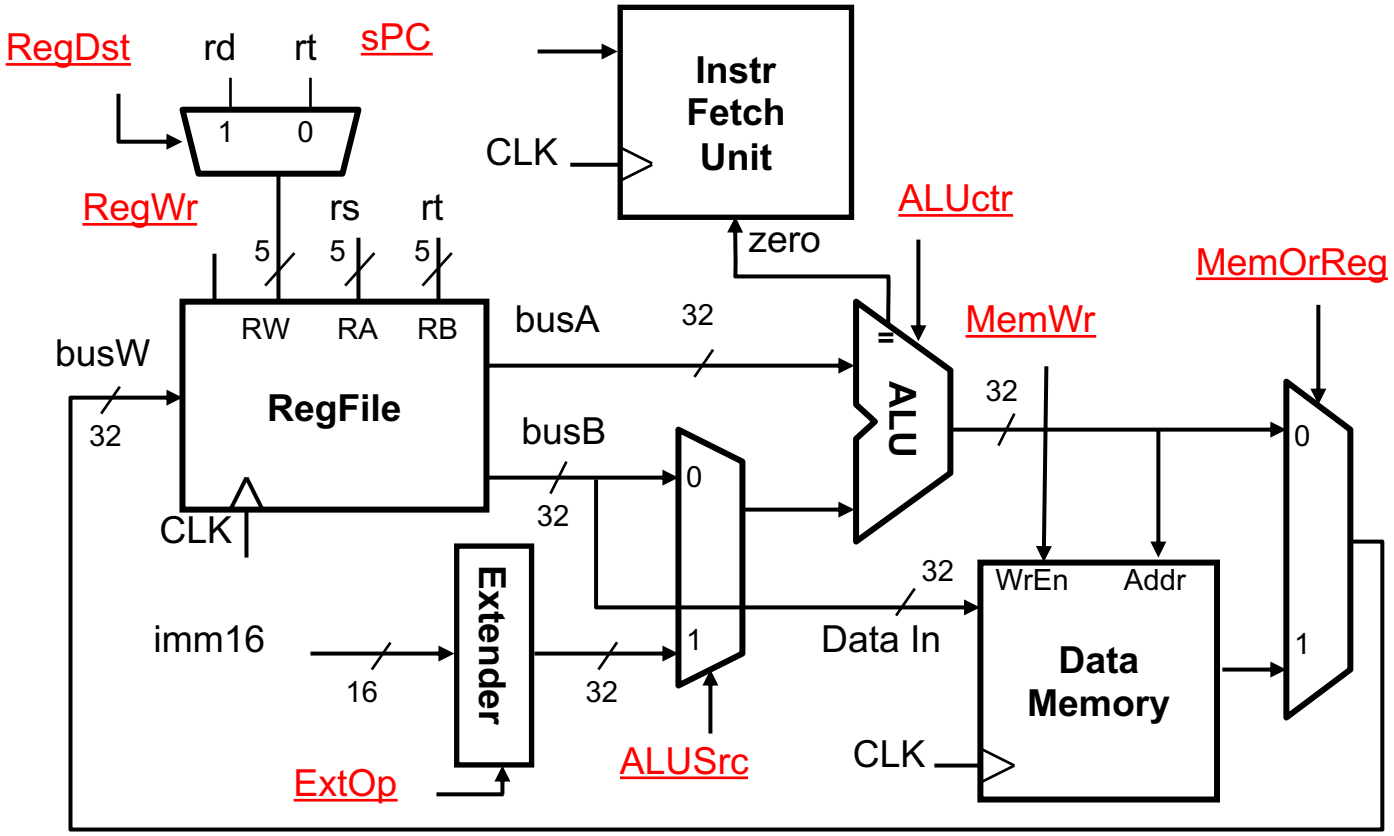
I 型指令的控制机制（4）-1

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



指令的控制

- **ExtOp:** 0 → “zero”;1 → “sign”
- **ALUsrc:** 0 → busB; 1 → imm16
- **ALUctr:** “ADD”, “SUB”, “OR”
- **sPC:** 0 → +4; 1 → branch
- **MemWr:** 1 → write memory
- **MemOrReg:** 0 → ALU; 1 → Mem
- **RegDst:** 0 → “rt”; 1 → “rd”
- **RegWr:** 1 → write register



指令的控制信号

- **ADDU**
  - ALUsrc=RegB, ALUctr="ADD", RegDst=rd, RegWr, sPC="+4"
- **SUBU**
  - ALUsrc=RegB, ALUctr="SUB", RegDst=rd, RegWr, sPC="+4"
- **ORI**
  - ALUsrc=Imm, ALUctr="OR", RegDst=rt, RegWr, ExtOp="Zero", sPC="+4"
- **LW**
  - ALUsrc=Imm, ALUctr="ADD", RegDst=rt, RegWr, ExtOp="Sign", MemOrReg, sPC="+4"
- **SW**
  - ALUsrc=Imm, ALUctr="ADD", MemWr, ExtOp="Sign", sPC="+4"
- **BEQ**
  - ALUsrc=RegB, ALUctr="SUB", sPC="Br" 1

指令的控制信号

参考MIPS指令手册 → func / op

	ADD	SUB	ORI	LW	SW	BEQ
func	10 0000	10 0010	n/a			
op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100
RegDst	1	1	0	0	X	X
ALUSrc	0	0	1	1	1	0
MemOrReg	0	0	0	1	X	X
RegWrite	1	1	1	1	0	0
MemWrite	0	0	0	0	1	0
sPC	0	0	0	0	0	1
ExtOp	X	X	0	1	1	X
ALUctr<1:0>	add	subtract	or	add	add	subtract

控制信号

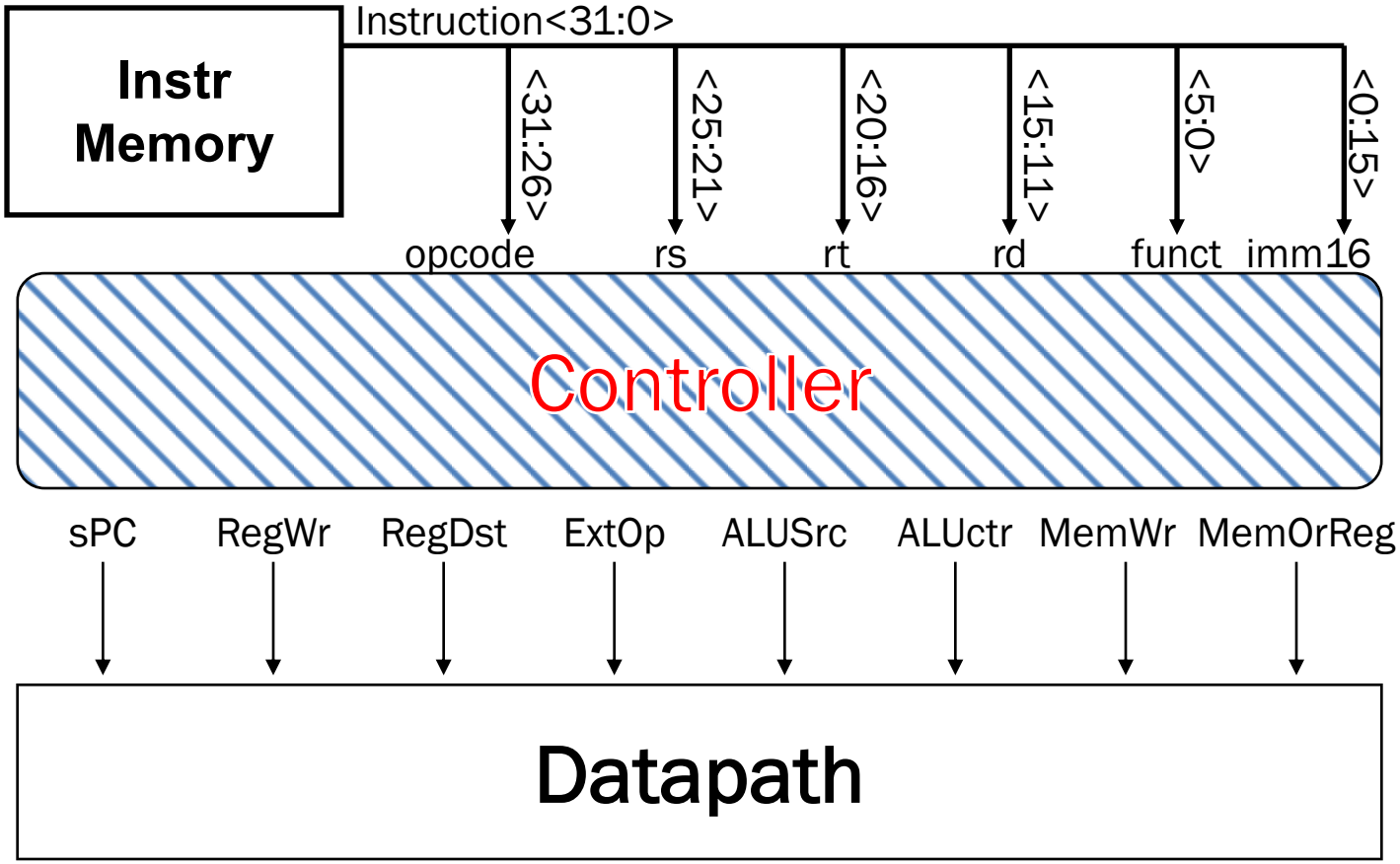
ALU支持的操作）

常用 MIPS 指令集及格式：

MIPS 指令集(共 31 条)

助记符	指令格式						示例	示例含义	操作及其解释
Bit #	31..26	25..21	20..16	15..11	10..6	5..0			
R-type	op	rs	rt	rd	shamt	func			
add	000000	rs	rt	rd	00000	100000	add $1,$2,$3	$1=$2+$3	rd <- rs + rt ；其中 rs=$2，rt=$3，rd=$1
addu	000000	rs	rt	rd	00000	100001	addu $1,$2,$3	$1=$2+$3	rd <- rs + rt ；其中 rs=$2，rt=$3，rd=$1,无符号数
sub	000000	rs	rt	rd	00000	100010	sub $1,$2,$3	$1=$2-$3	rd <- rs - rt ；其中 rs=$2，rt=$3，rd=$1

控制器设计（1）


Instr Memory
Instruction<31:0>
<31:26>
<25:21>
<20:16>
<15:11>
<5:0>
<0:15>
opcode
rs
rt
rd
funct
imm16
Controller
sPC
RegWr
RegDst
ExtOp
ALUSrc
ALUctr
MemWr
MemOrReg
Datapath

控制器设计（2）

- 实现方式多种多样，这里进介绍一种简单直观方式
- 思路
  - 指令的OP和FUNC编码唯一确定指令——译码
  - 指令使能各控制信号

译码（ID）阶段

- 实现方式
  - 指令译码
    - I型和J型指令，只与OP相关
      - $BEQ = \overline{op[5]} \wedge \overline{op[4]} \wedge \overline{op[3]} \wedge op[2] \wedge \overline{op[1]} \wedge \overline{op[0]}$
    - R型指令
      - $Rtype = \overline{op[5]} \wedge \overline{op[4]} \wedge \overline{op[3]} \wedge \overline{op[2]} \wedge$
      - $ADD = Rtype \wedge (op[5] \wedge \overline{op[4]} \wedge \overline{op[3]} \wedge \overline{op}$

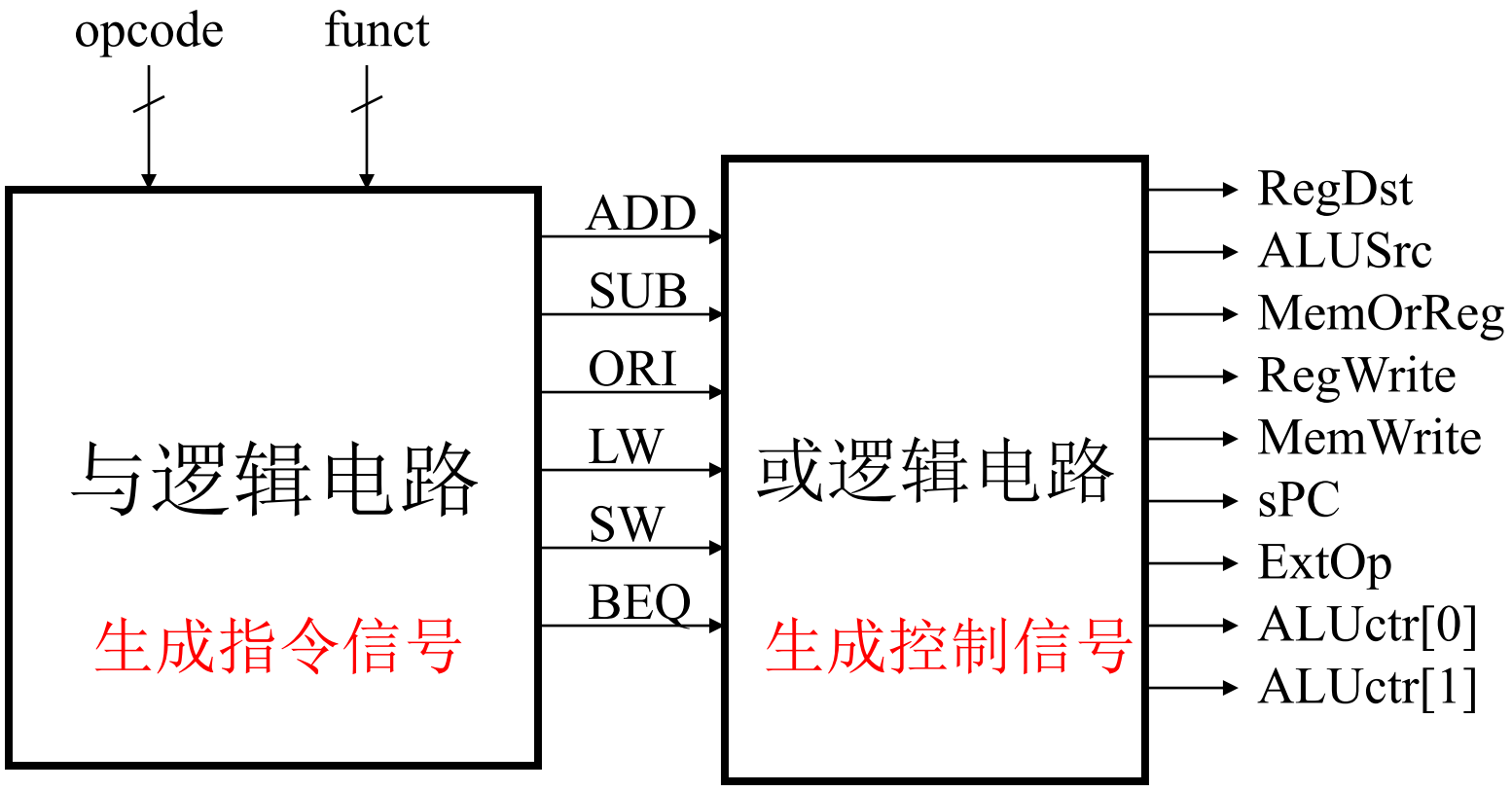
与逻辑

为什么公式中不用考虑其他变量？

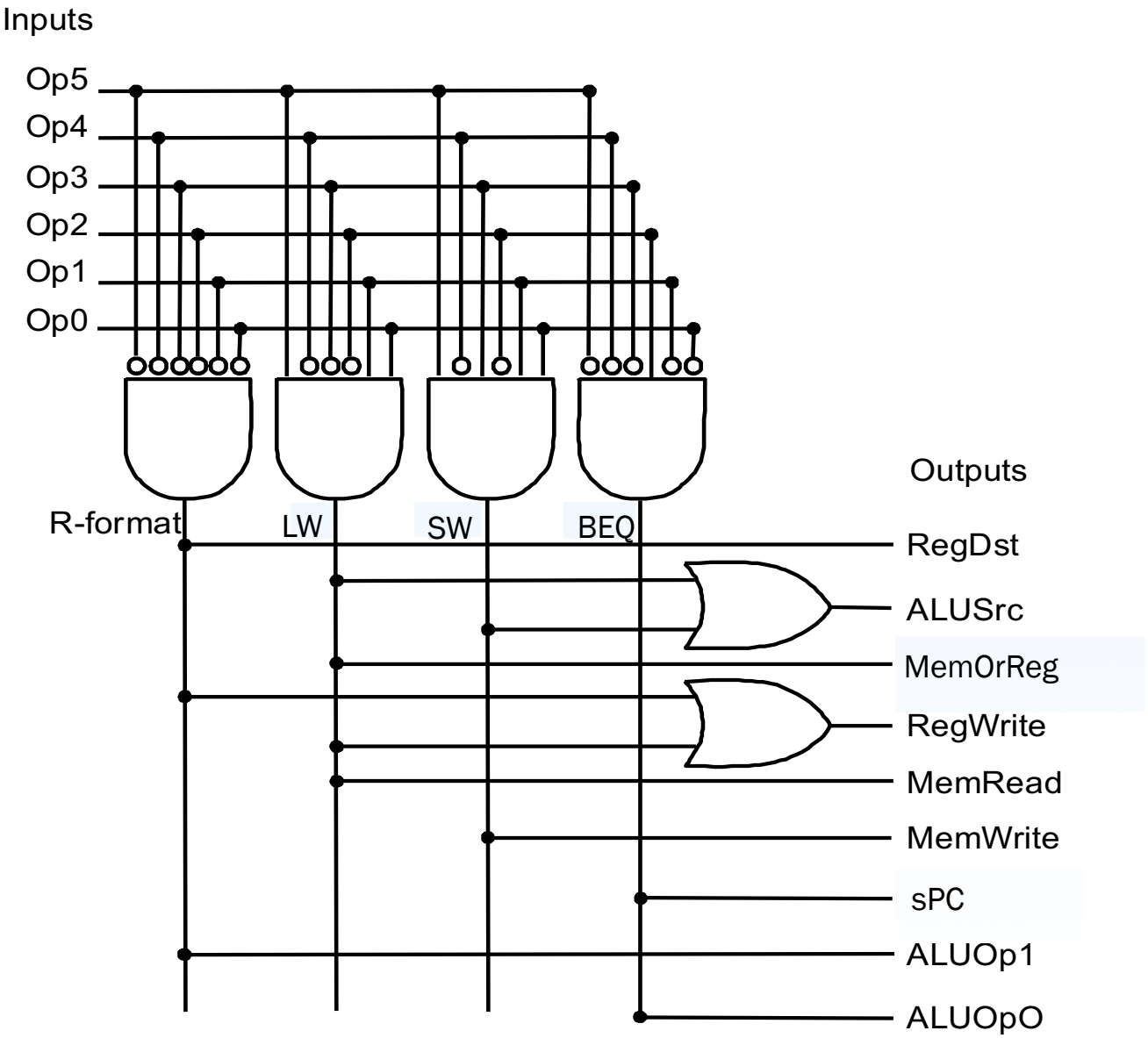
  - 指令控制
    - $MemWrite = SW$
    - $RegWrite = ADD \vee SUB \vee ORI \vee LW$

或逻辑

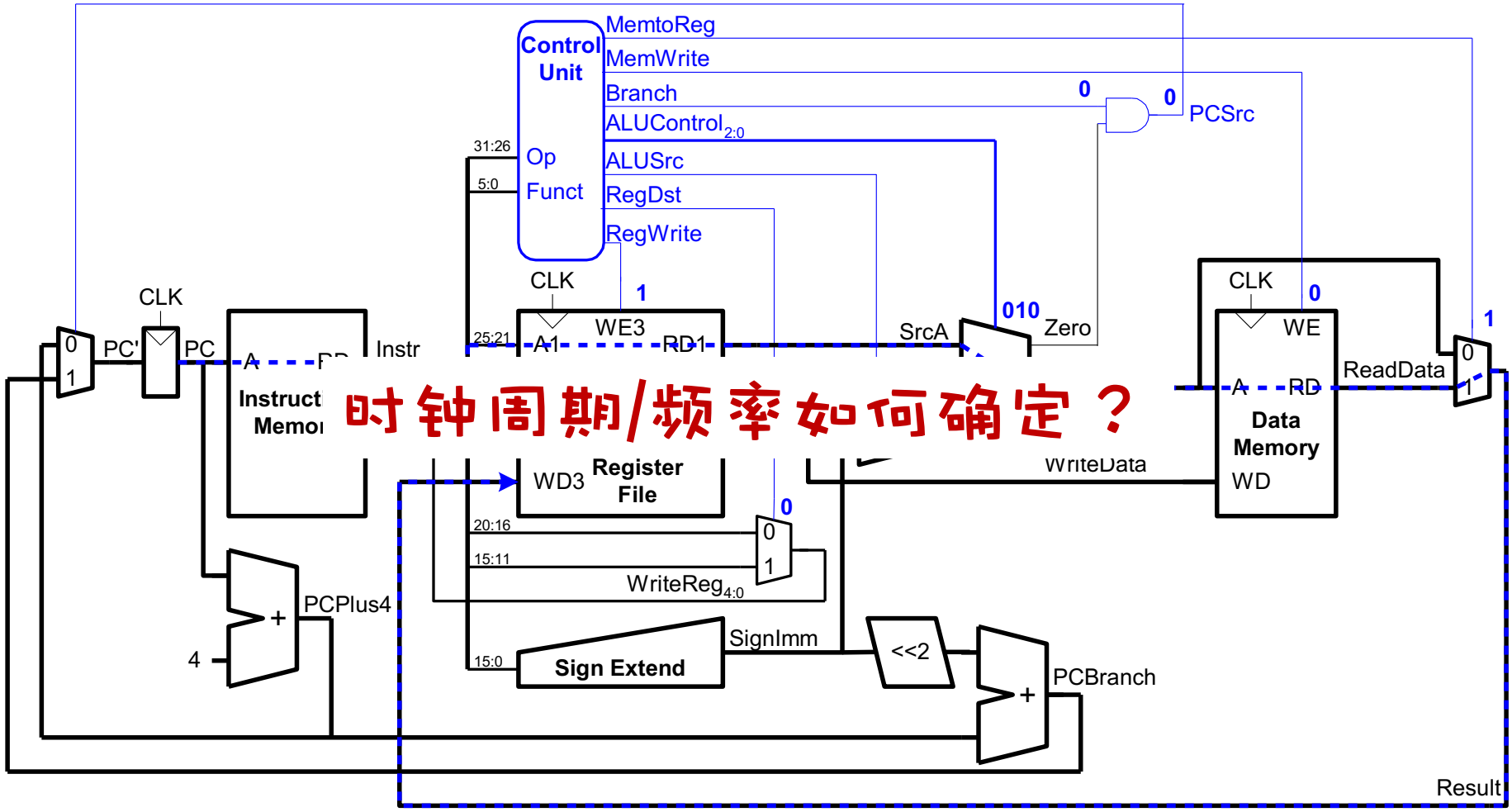
控制器设计（3）


opcode
funct
与逻辑电路
生成指令信号
ADD
SUB
ORI
LW
SW
BEQ
或逻辑电路
生成控制信号
RegDst
ALUSrc
MemOrReg
RegWrite
MemWrite
sPC
ExtOp
ALUctr[0]
ALUctr[1]

控制器设计（4）


Inputs
Op5
Op4
Op3
Op2
Op1
Op0
R-format
LW
SW
BEQ
Outputs
RegDst
ALUSrc
MemOrReg
RegWrite
MemRead
MemWrite
sPC
ALUOp1
ALUOpO

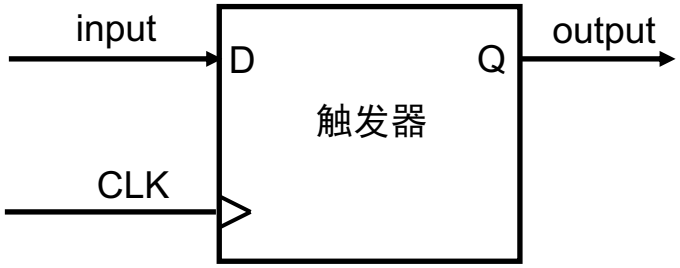
完整的CPU


Control Unit
MemtoReg
MemWrite
Branch
ALUControl2:0
ALUSrc
RegDst
RegWrite
Op
Funct
31:26
5:0
PCSrc
CLK
PC'
PC
Instruct Memor
Instr
25:21
A1
WE3
RD1
WD3
Register File
SrcA
010
Zero
WE
A
RD
Data Memory
WD
ReadData
WriteData
20:16
15:11
WriteReg4:0
PCPlus4
4
SignImm
15:0
Sign Extend
<<2
PCBranch
Result
时钟周期/频率如何确定？

四、性能分析

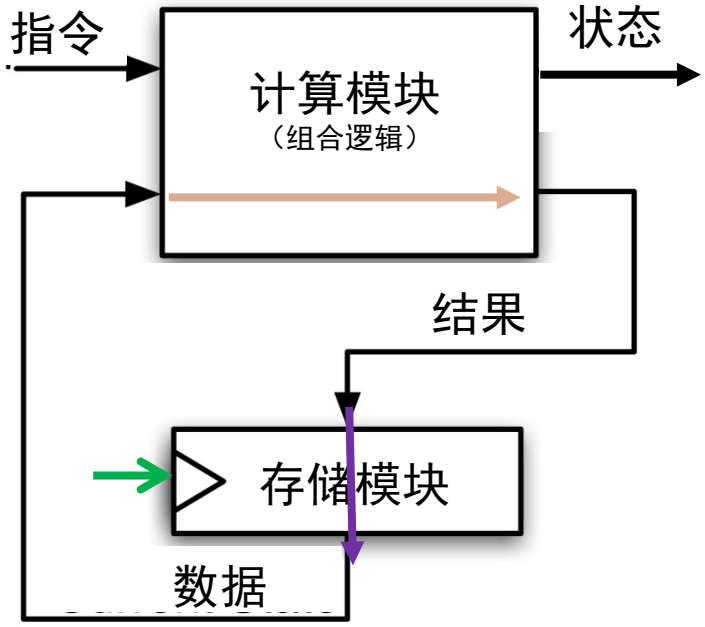
触发器的时间因素

- **配置时间（Setup Time）：**
  - 从CLK触发开始到数据传输开始，input需要保持稳定的时间
- **保持时间（Hold Time）**
  - CLK触发结束，input需要保持的时间
- **数据传输延迟（“CLK-to-Q” Delay）**
  - input开始传输到完成的时间（包含了保持时间）



电路的最大延迟

- **数据角度上的CPU模型**
  - 速度受限于得到下一次数据的时间



最大延迟 = 配置时间

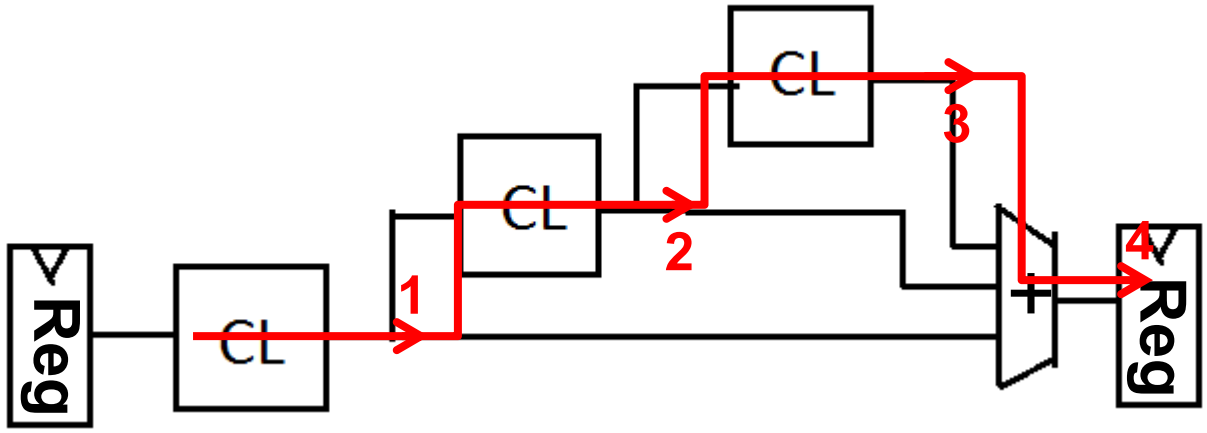
\+ 数据传输延迟

\+ 组合逻辑延迟

最大频率 = 1/最大延迟

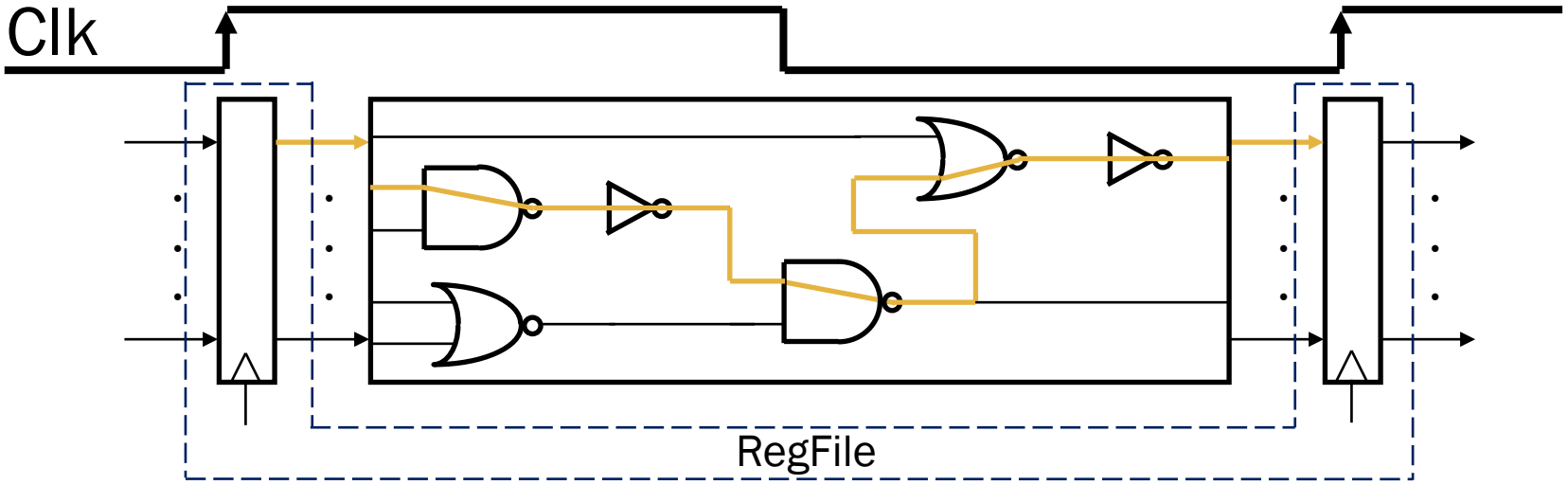
关键路径

- 关键路径
  - 两个寄存器之间最长的延迟
- 时钟周期必须大于关键路径延迟，否则信号就不会稳定和正确地传输到下一个寄存器



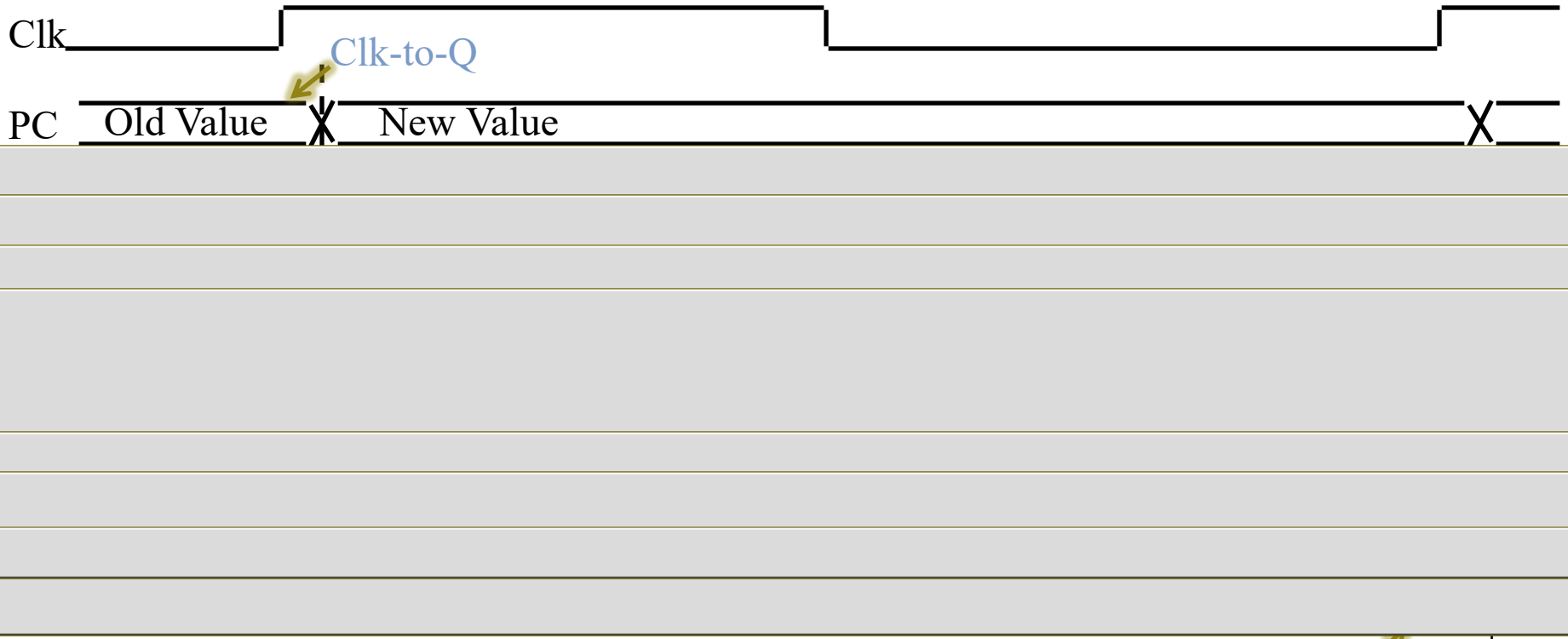
关键路径 =
组合电路1延迟
+ 组合电路2延迟
+ 组合电路3延迟
+ 加法器延迟

单周期CPU时序分析（1）

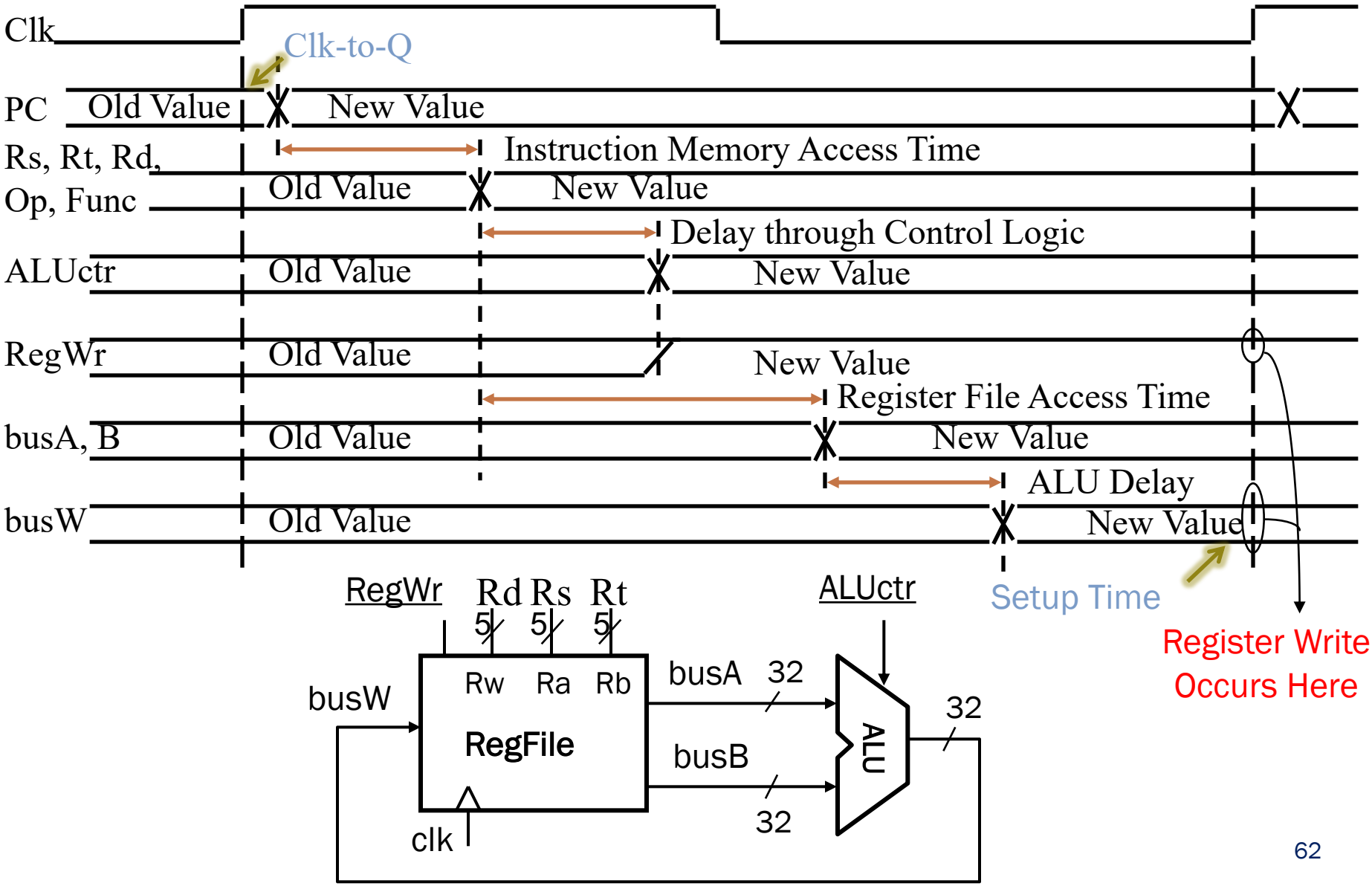


- 存储单元（RegFile, Mem, PC）由同一个系统时钟触发
- 关键路径确定时钟周期
  - 包含寄存器的配置时间和数据传输时间，以及组合电路
- 所有指令都在1个时钟周期内完成

单周期CPU时序分析（2）



单周期CPU时序分析（3）



性能估算

- 假设寄存器读和写的时间假设为100ps，功能模块时延为200ps
- 最小的时钟周期估算

指令	取指令	寄存器读	ALU操作	访问内存	寄存器写	总时间
LW						
SW						
R型指令						
BEQ	200ps	100 ps	200ps			500ps

如何提升时钟频率？

提高时钟频率是否能有效提升性能？

提高时钟频率就能提高指令速度吗？

单周期CPU的问题（1）

- 假定某单周期CPU各主要部件的延迟为：
  - 存储器（Memory）：2ns
  - 运算器（ALU/Adder）：2ns
  - 寄存器组（Register File）：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

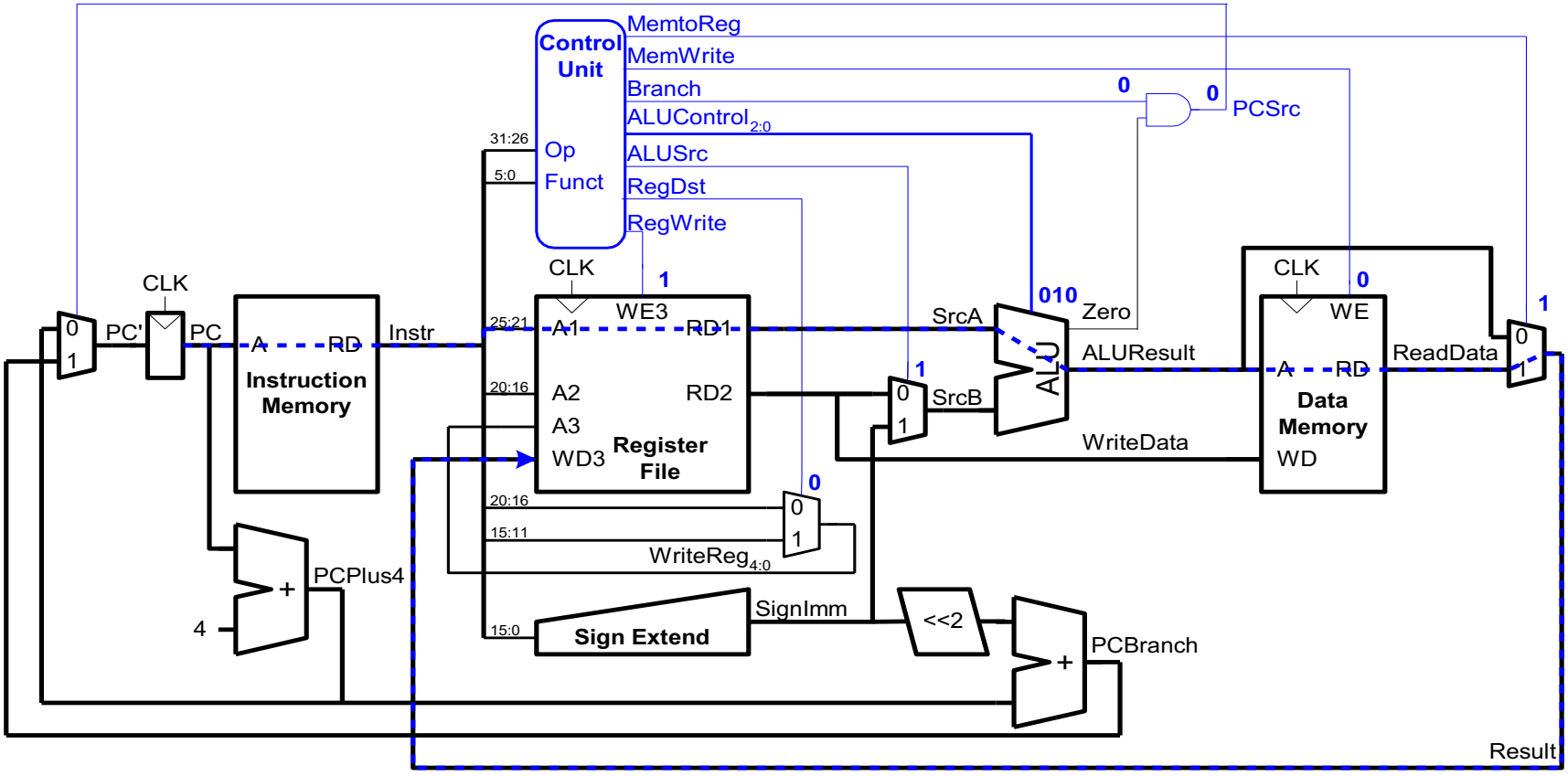
- 指令周期比较长
- 所有指令都必须使用最长的周期

单周期CPU的问题（2）

- 假设某单周期CPU，执行100条指令：
  - 25%的Load指令
  - 10%的Store指令
  - 45%的算逻指令
  - 20%的跳转指令
- 单周期的执行时间
  - 100*8 = 800ns
- 可能的优化
  - 25*8+10*7+45*6+20*5=640ns
  - Speedup=800/640=1.25

单周期CPU的问题（3）

- 事实上，指令和数据都保存在同一个存储器中；
- 许多部件保持数据的时间过长，无法复用。
  - 例如， Adder 是否可以利用ALU?



单周期CPU特点

- 优点
  - 每条指令占用一个时钟周期
  - 逻辑设计简单，易于设计控制器
- 缺点
  - 各组成部件的电路模块复用！
    - 各部件大部分时间在等待
  - 时钟周期应满足执行时间最长指令的要求
    - Load指令
  - CPI =1

如何进一步改进？

问题和讨论