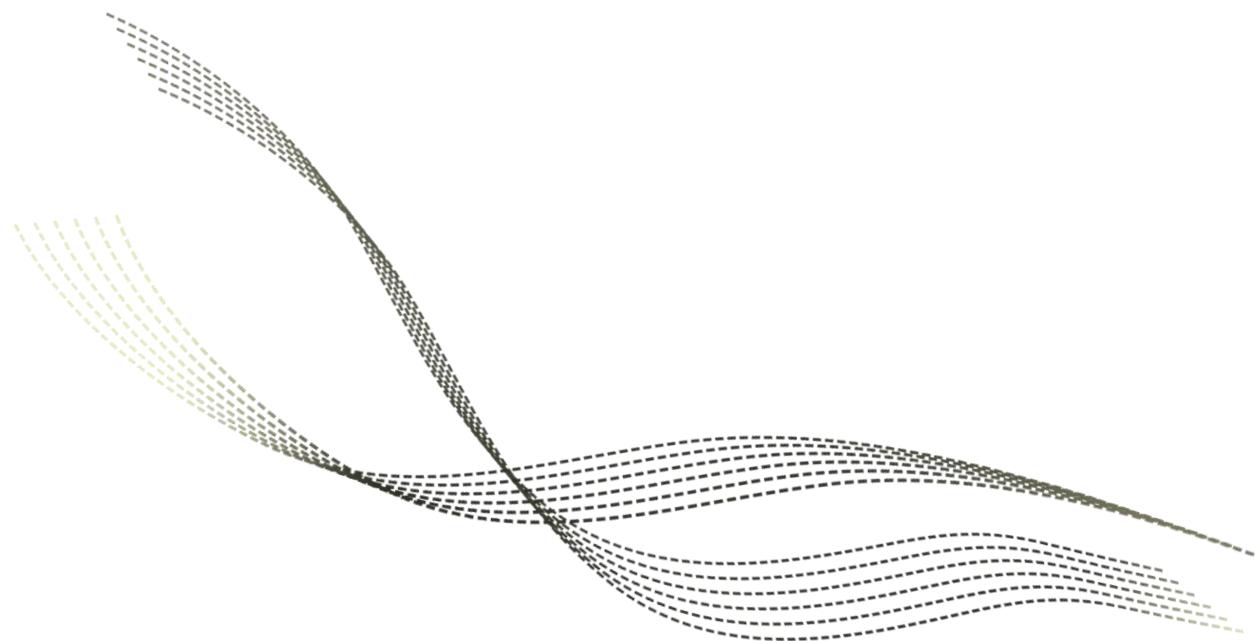


# 高级计算机体系结构

Advanced Computer Architecture

指令集架构

沈明华



# 目录

## CONTENTS

- 01 指令集架构**
- 02 主流指令集**
- 03 用户 / 系统ISA**
- 04 基础微架构设计**

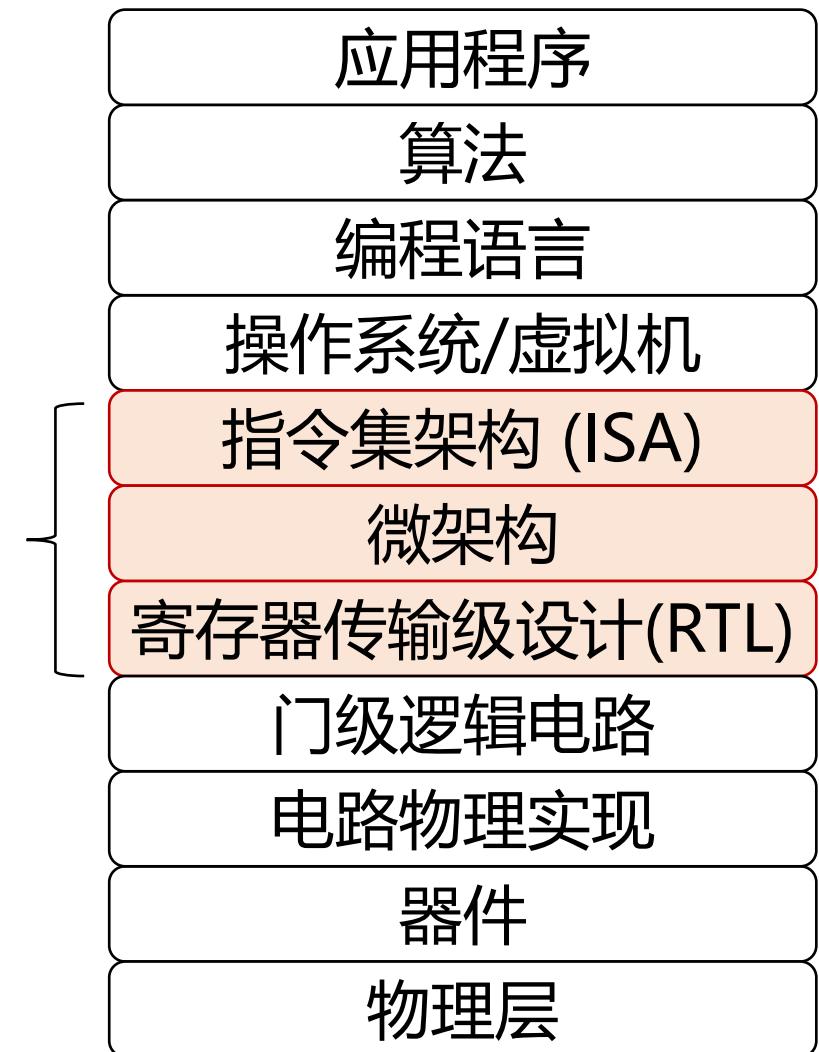
# PART 01

## 指令集架构

# ■ 指令集架构

## □ 不同视角下体系架构设计

- 指令集架构(ISA, instruction set architecture)
  - 在编译器设计者看来：硬件系统通过ISA等接口层向编译器设计者暴露的“状态交互规则”（指令编码格式、I/O端口映射...）
- 微架构(micro-architecture)
  - 在CPU设计者看来：微架构是实现特定ISA的逻辑结构和组织方式
- 物理设计(physical design)
  - 在IC设计者看来：通过设计优化后，承载微架构或芯片设计方案的实体



# ■ 指令集架构

## □ 什么是指令集架构

### - 指令集架构

- 指令集架构是下层硬件与上层程序应用之间交互的规范接口
- 规定了计算机能够执行的所有指令的集合，指令的功能，可访问内存空间等

### - 指令集架构的基本功能：定义数据与控制流

- 明确硬件中可用于存放数据的载体及其访问规则(内存及其寻址方式)
- 定义指令对数据的处理方式(算术/逻辑指令、浮点数指令)



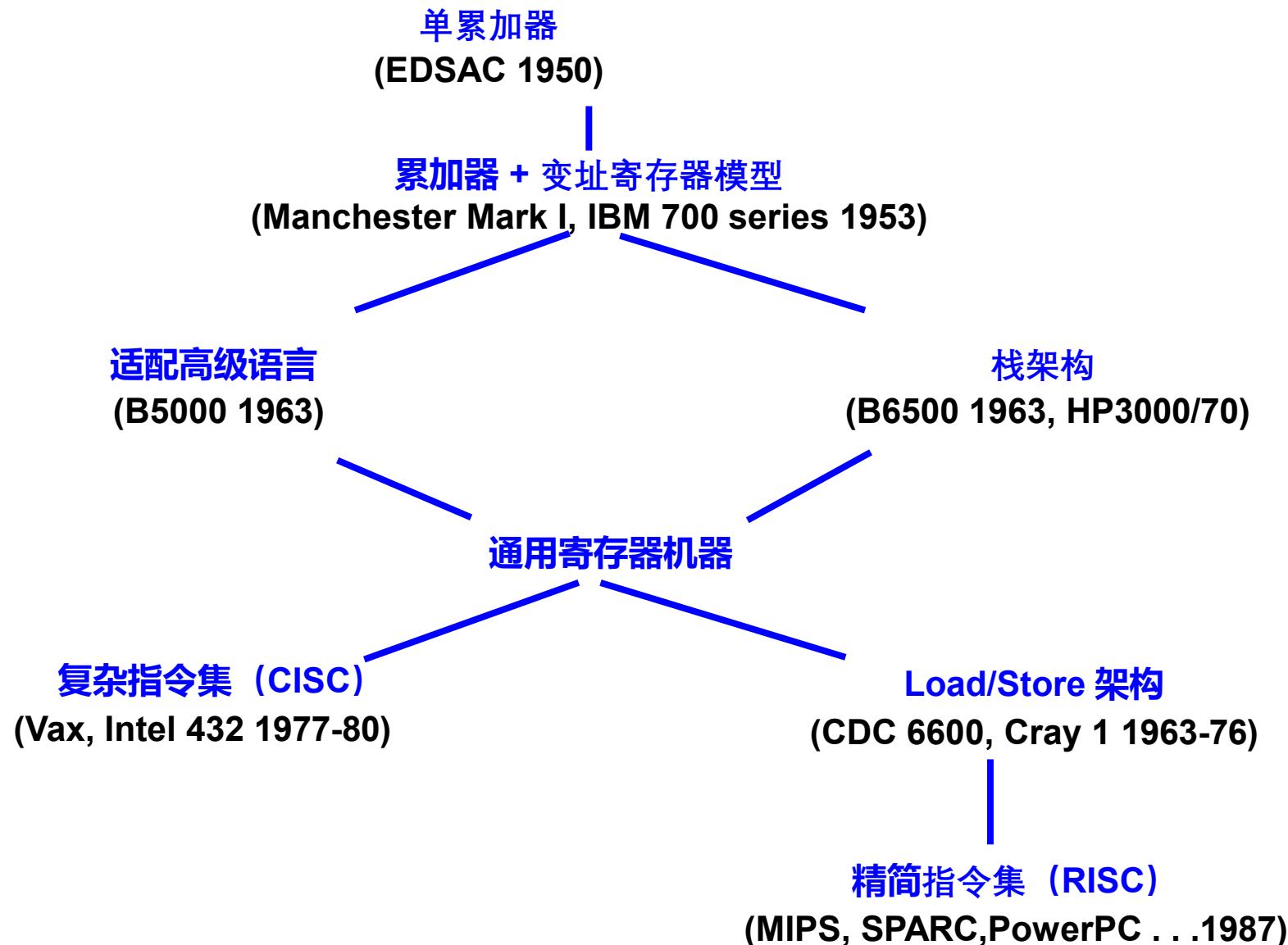
# ■ 指令集架构

---

## □ 其他指令集

- 指令集不止一种，历史上出现过许多种不同的指令集
- IBM 360系列：model 30(1964)、z900(2001)
- x86系列：8086(1978)、80186、286、386
- MIPS系列：R2000、R4000、R10000
- ARM系列：Armv7(32bit)、Armv8(64bit)
- 开源系列：RISC-V

# ■ 架构演变过程



## ■ 四种经典架构

---

- $i = i + 1$ 生成的指令

### Stack

```
PUSH &i ;  
PUSH 0x01;  
ADD;
```

### Accumulator

```
LDA &i ;  
ADDA 0x01;  
STA &i;
```

### Register-Memory

```
MOVE d0, &i ;  
INC d0;  
MOVE &i, d0;
```

### Register-Oriented

```
LD r7, &i ;  
ADD r7, 0x01, r7;  
ST r7, &i;
```

# PART **02**

## 主流指令集

## ■ RISC V.S. CISC

---

□ 当前CPU指令集主要分为两类

- 精简指令集(RISC): 指令简单、统一、功耗低、硬件设计简洁(ARM、RISC-V)
- 复杂指令集(CISC): 指令种类多、功能强大、硬件设计复杂(x86)

# ■ 复杂指令集(CISC)

---

## □ 复杂指令集(Complex Instruction Set Computer)

- 设计理念可追溯至20世纪80年代中
  - CISC对编译器友好
    - 可以生成字节数更少的代码
    - 因为指令功能更复杂，所以更依赖复杂的硬件设计
- CISC是基于栈的指令集
  - 利用栈传递参数、保存返回地址、保存程序计数器等
  - 指令中具有明确操作栈的指令(push、pop)
- 混合Register memory架构
  - 算术指令可以直接访问内存(无需先加载到寄存器)
- 条件码(condition codes)
  - 作为算术指令的附带结果被设置，后续分支指令可以依据条件码决定程序执行流程

## ■ 复杂指令集

---

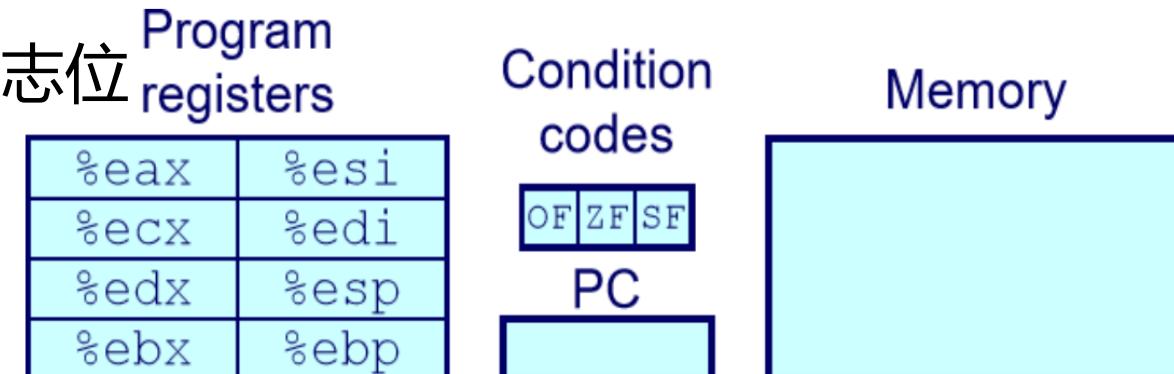
### □ 复杂指令集的代表作：IA32

- 英特尔兼容处理器(俗称x86)
  - IA32指"Intel Architecture, 32-bit"，是全球计算机领域极具影响力的处理器架构家族
  - 始于1978年发行的8086处理器，时至今日仍然在个人电脑与服务器市场表现强势
    - 指令集支持的特性仍然在不断扩充
    - 但在个人移动终端领域相比ARM架构缺乏竞争力
- 注意区分两个概念
  - x86\_64是x86架构的64bit版本
  - IA64则是intel为高性能领域提出的全新指令集，与IA32无关

# ■ 复杂指令集

## □ IA-32寄存器种类

- 包含两种寄存器：通用寄存器(General-purpose register, **GPR**)、专用寄存器(Special-purpose register, **SPR**)
  - GPR能用于存放操作数，例如整型、浮点数、部分指针
  - SPR只能用于特定功能，比如条件码、处理器状态、PC指针、栈指针
- 80386 CPU包含8个GPR，每个都是32bit
- 例如intel的程序计数器(PC指针)是一种SPR，存放下一条指令的地址
- 条件码存放最近执行的算数指令的标志位
  - 例如算术结果为负数、为零
  - 例如有溢出异常
  - 条件跳转指令直接根据标志位执行



# ■ 复杂指令集

---

## □ 处理器状态(Processor State)

- 定义：
  - 指令执行结束时，处理器内部存储的、为下一条指令执行提供上下文信息的集合
  - 例如程序计数器、通用寄存器值、专用寄存器等数据
- 处理器哪些状态对程序员可见十分关键
  - 例如栈指针、PC指针当前的位置应对程序员可见，方便调试
  - 又例如中断，硬件必须以程序员无感的形式自动完成状态的保存与恢复
- 容易与处理器性能状态(Processor Power State)混淆
  - 处理器状态：指令执行的上下文信息集合
  - 处理器性能状态：计算机硬件为平衡性能需求与能耗控制，定义的处理器工作模式集合（空闲状态、性能状态...）

## ■ 精简指令集(RISC)

---

### □ 精简指令集(Reduced Instruction Set Computer)

- 设计理念可追溯至20世纪80年代
  - 仅保留最基础且常用的指令，提供更少且更简单的指令
    - 完成单个任务可能需要执行更多指令
    - 指令功能单一、执行逻辑更简单
    - 可以在更小以及时钟频率更快的硬件上执行
- Register-oriented的指令集
  - RISC指令的操作数几乎都来自于寄存器。需要大量寄存器(通常32个)
  - 比如返回地址存放在专用寄存器，不像x86存放在栈顶
- 基于Load-store架构
  - 只有load/store指令可以直接访问内存
  - 无条件码(condition codes)设计
- 通过显式分支指令实现条件控制逻辑

## ■ 精简指令集

---

### □ 精简指令集的代表作1：RISC-V



- 对比ARM的不可控、x86的不自主
- 开源的RISC-V指令集对于国内CPU的自主可控至关重要
  - 也因为开源的特性，基于RISC-V的CPU支持的指令集不完全相同
  - 设计者可以根据自身需要选配相应的指令集
    - 比如提供32位整数计算的RV32I、RV64I指令包
    - 提供功耗敏感的精简32位整数计算RV32E指令包
    - 提供整数乘法/除法的M扩展指令包
    - 提供浮点数运算的F扩展指令包、D扩展指令包
    - 加速矩阵计算的Q扩展指令包

## ■ 精简指令集

---

### □ 精简指令集的代表作2：MIPS系列

- 这个名称最初是 “**Microprocessor without Interlocked Pipeline Stage**” 的缩写
  - 意为无锁流水线微处理器
- 是20世纪80年代ISA作品中比较优雅的样例
- 现在广泛用于计算机体系结构的课程，商用领域相对较少
  - MIPS架构设计简洁、逻辑清晰，指令集设计规整
  - 五级流水线(取指、译码、执行、访存、写回)是流水线架构标准模型
  - 利用编译器优化代替硬件锁，规避流水线冒险

# ■ MIPS寄存器

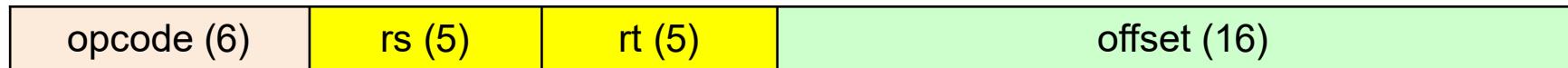
\$0	\$0		常量 0	\$16	\$s0		
\$1	\$at		保留给汇编器	\$17	\$s1		
\$2	\$v0			\$18	\$s2		
\$3	\$v1		函数返回值	\$19	\$s3		
\$4	\$a0			\$20	\$s4		
\$5	\$a1			\$21	\$s5		
\$6	\$a2		函数参数	\$22	\$s6		
\$7	\$a3			\$23	\$s7		
\$8	\$t0			\$24	\$t8		同\$t0~\$t7
\$9	\$t1			\$25	\$t9		
\$10	\$t2			\$26	\$k0		操作系统保留
\$11	\$t3			\$27	\$k1		
\$12	\$t4		Caller Save Temporaries 临时寄存器，被 调用的子函数可 以任意使用 \$t0~\$t7	\$28	\$gp		全局指针
\$13	\$t5			\$29	\$sp		栈指针
\$14	\$t6			\$30	\$s8		帧指针
\$15	\$t7		调用者调用前需 要保存数据	\$31	\$ra		返回地址

# ■ MIPS指令概览



**Register-Register:** rd  $\leftarrow$  (rs) funct (rt)

add \$s1, \$s2, \$s3      # register add: \$s1 = \$s2 + \$s3



**Load/Store:** rt  $\leftarrow$  Mem[(rs) + offset]

lw \$s1, 10(\$s2)      # load word: \$s1 = Mem[\$s2 + 20]



**Branch:** go to offset $\times$ 4 if (rs) equal to (rt)

beq \$s1, \$s2, 25      # if (\$s1==\$s2) then go to [PC+4+100]



**Jump:** go to target $\times$ 4 (append 10000 to PC <31:28> to get new address)

j 2500      # 无条件直接跳转到新地址

# PART **03**

## 用户 / 系统ISA

# ■ 用户 / 系统ISA

## □ 高级语言到机器码的编译过程

### - 编译器(Compiler)

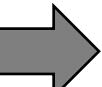
- 编译器将高级语言代码翻译为汇编语言
  - 把变量x、a分配到寄存器\$S1, \$S2，数组A的基地址储存在\$S0
  - 得到右图所示的汇编代码，其实就是利用ISA实现高级语言的功能

### - 汇编器(Assembler)

- 将汇编语言转换为硬件可识别的机器码

源代码

x = a + A[2]

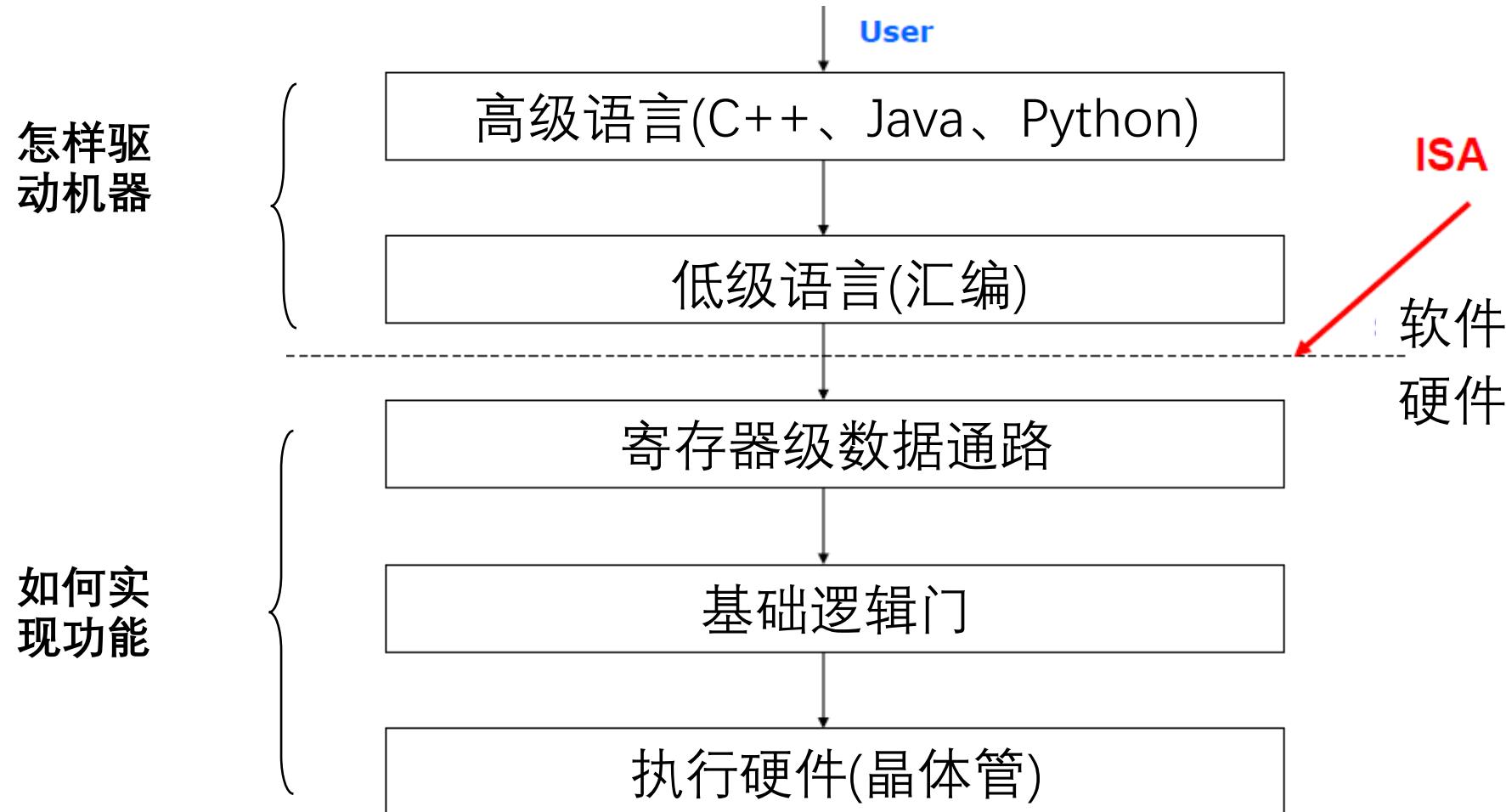


汇编代码

```
lw    $t0, 8($s0)  
add  $s1, $s2, $t0
```

# ■ 用户 / 系统ISA

## □ 指令集架构定义软硬件间接口



# ■ 用户 / 系统ISA

---

□ ISA可以分为用户ISA和系统ISA两类

– 用户ISA，目的是完成工作

- 编译器将高级语言映射为机器码时，仅需关注用户ISA
- 无法直接访问系统级资源(内核态内存...)
- 用户ISA包括算术运算指令，访存指令，控制流指令...

– 系统ISA，目的是管理共享资源

- 系统ISA指令由操作系统底层通过汇编语言编写，实现调度器、虚拟内存、驱动硬件等功能
- 系统ISA包含权限切换指令，资源控制指令，异常与中断处理指令...

# ■ 用户ISA

---

## □ 用户态可见的指令集合

– 通常指代那些可以被用于程序使用的指令集合，例如

- 数据流操作(Load/store)
- ALU操作(算术逻辑运算)
- 控制流操作(分支指令)

整数运算	内存操作	控制流转移	浮点运算
Add	Load byte	Jump	Add single
Sub	Load word	Jump equal	Mult. double
And	Store multiple	Call	Sqrt double
Compare	Push	Return	...
...	...	...	

# ■ 系统ISA

---

## □ 系统ISA

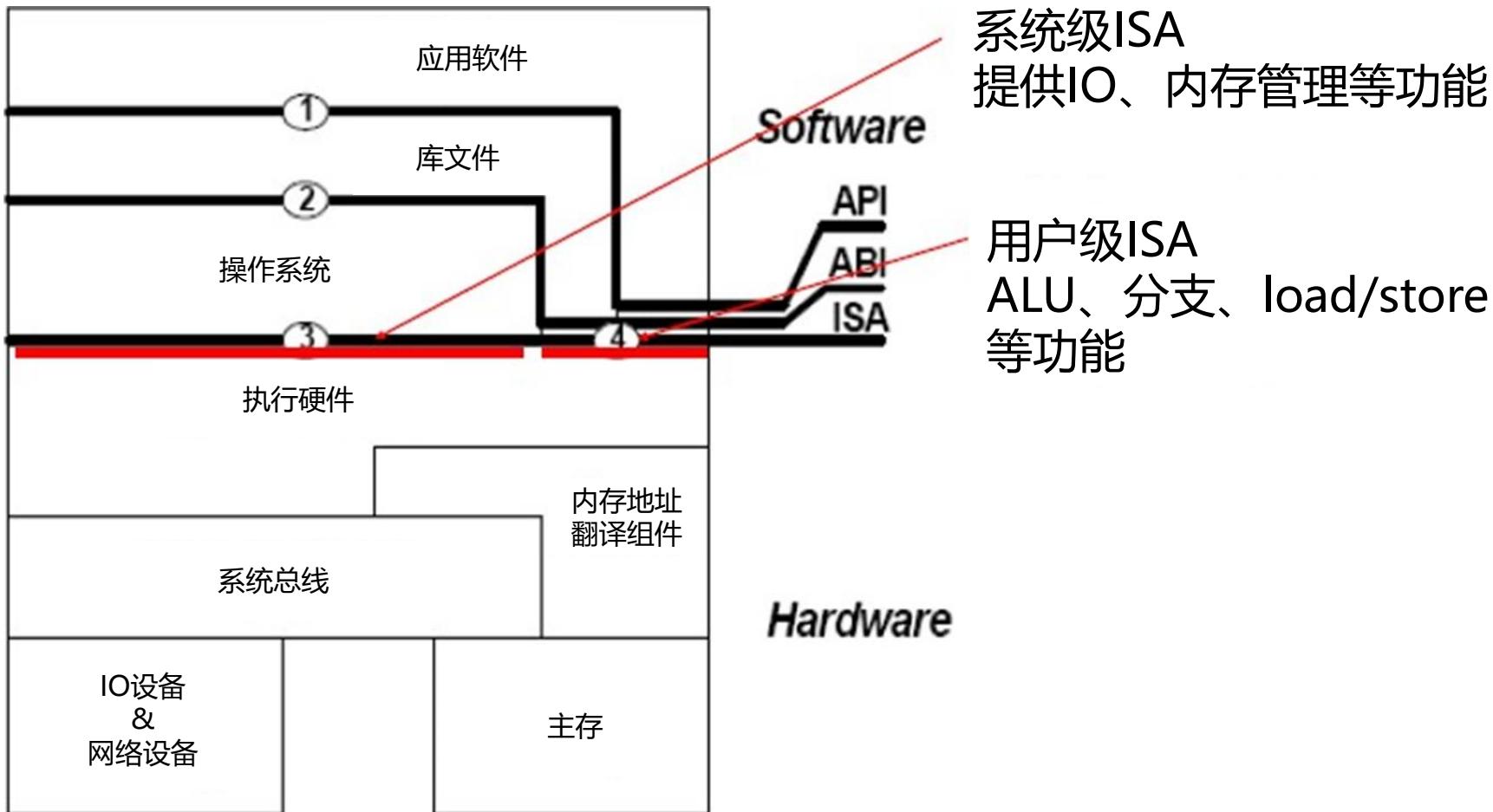
- 内核态可见的指令集，负责管理硬件资源

## □ 系统ISA概览

- 特权级别
  - 区分用户态和内核态，通过硬件层面划分权限等级
- 控制寄存器(系统ISA专用寄存器集合)
- 管理关键资源的指令
  - CPU的上下文切换、时钟控制...
  - 内存的地址隔离、TLB操作...
  - 存储设备的I/O端读写、内存映射I/O...

# ■ 用户 / 系统ISA

## □ 不同ISA的系统位置示意



# ■ 用户 / 系统ISA

---

## □ 总结

- 理想中的ISA应该
  - 适配多种不同的硬件实现(兼容性)
  - 被广泛使用(通用性)
  - 面向高层语言提供方便好用的接口
  - 面向底层设计要方便实现
- 同时ISA一直在变
  - 例如新的技术支持人们制造更大、功耗更高或频率更高的CPU
  - 编译器能力增强
  - 编程风格的改变，从汇编到面向对象
  - 应用的改变，从多媒体转向深度学习

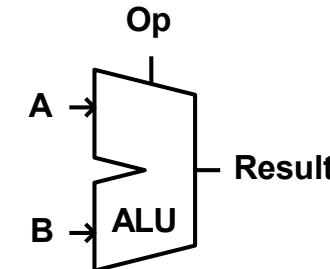
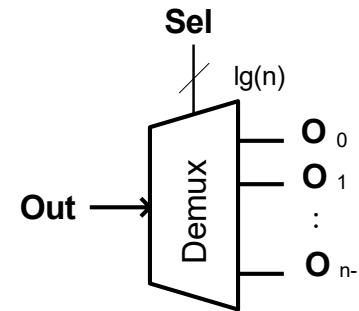
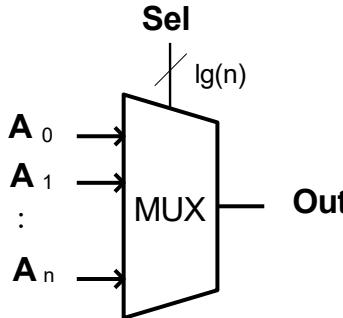
# PART **04**

## 基础微架构设计

# ■ 元件

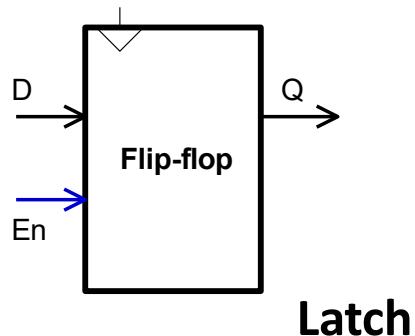
## □ 组合逻辑

- 输出只与当前时刻的输入有关

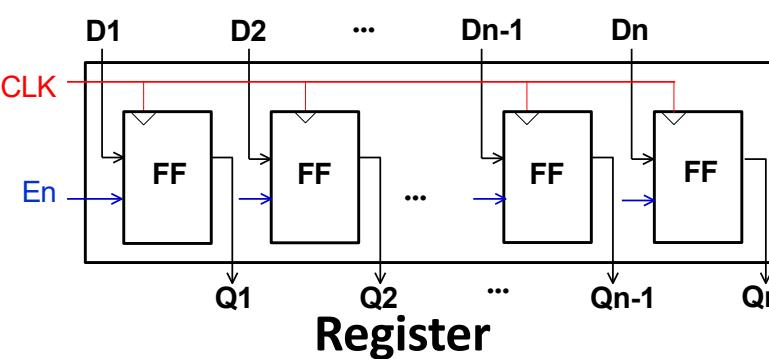


## □ 同步时序逻辑

- 边沿触发(Edge-triggered): 数据在时钟边沿采样

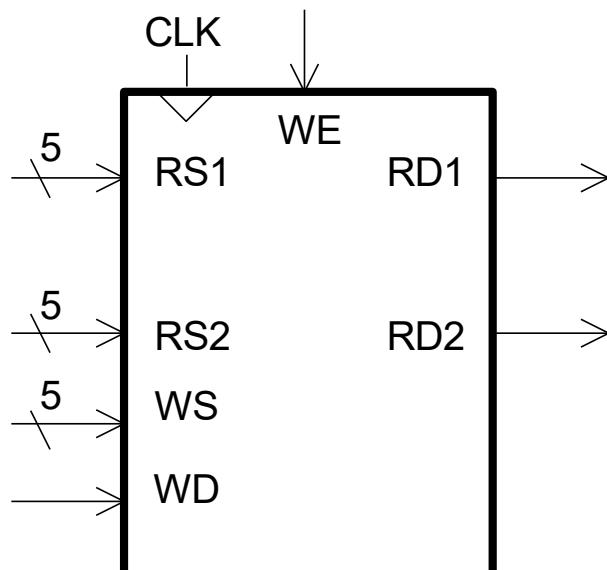


D	CLK	Q
0	Rising	0
1	Rising	1
-	0	Last Q
-	1	Last Q

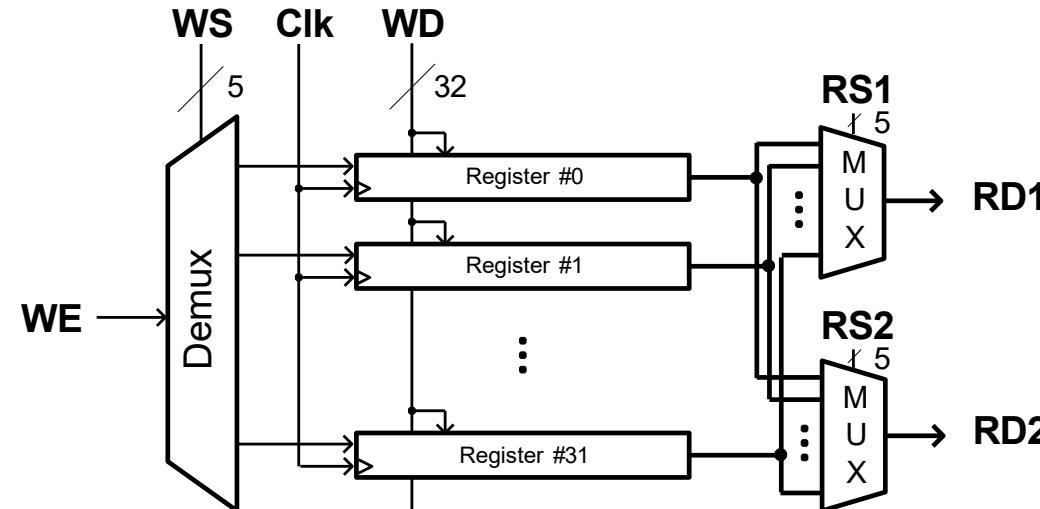


# ■ 寄存器堆

□ 处理器的通用寄存器保存在一个叫做寄存器堆(register file)的结构



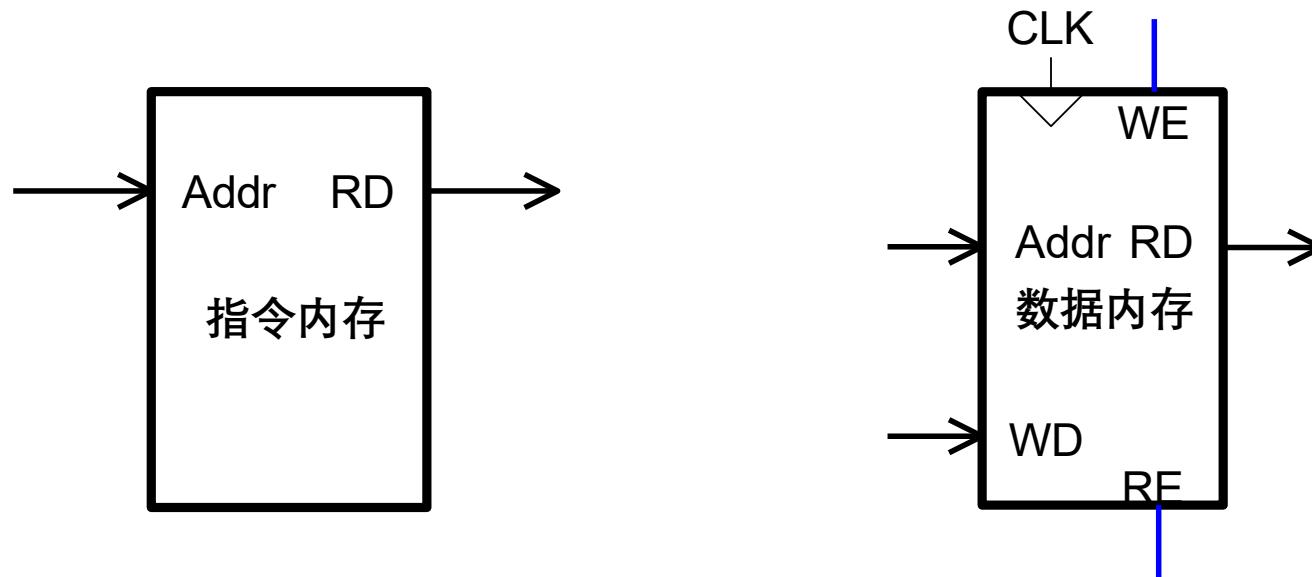
寄存器堆的逻辑结构



寄存器堆的物理结构

## ■ 简单的内存模型

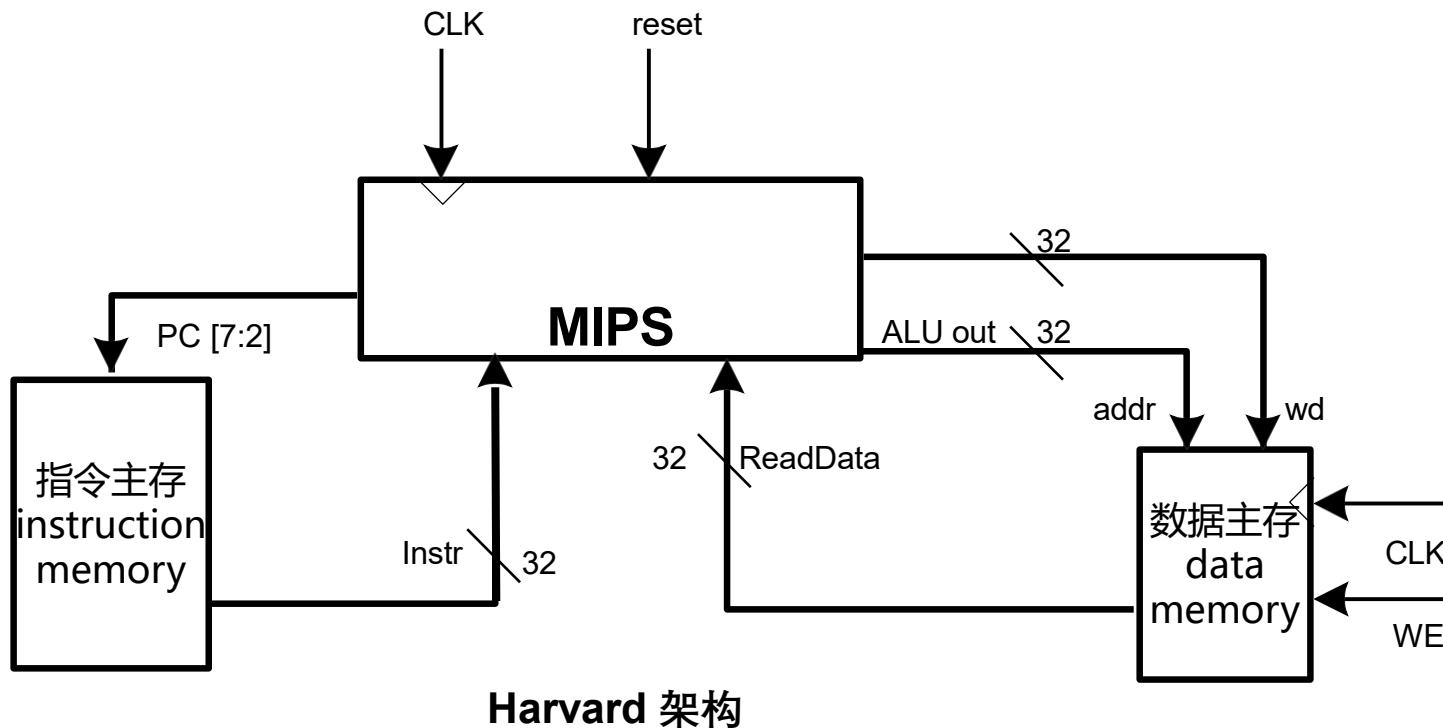
- 可以在任意时刻进行读操作(组合逻辑)
- 只能在时钟上升沿进行写操作



# ■ 两种微架构

## □ 两种微架构

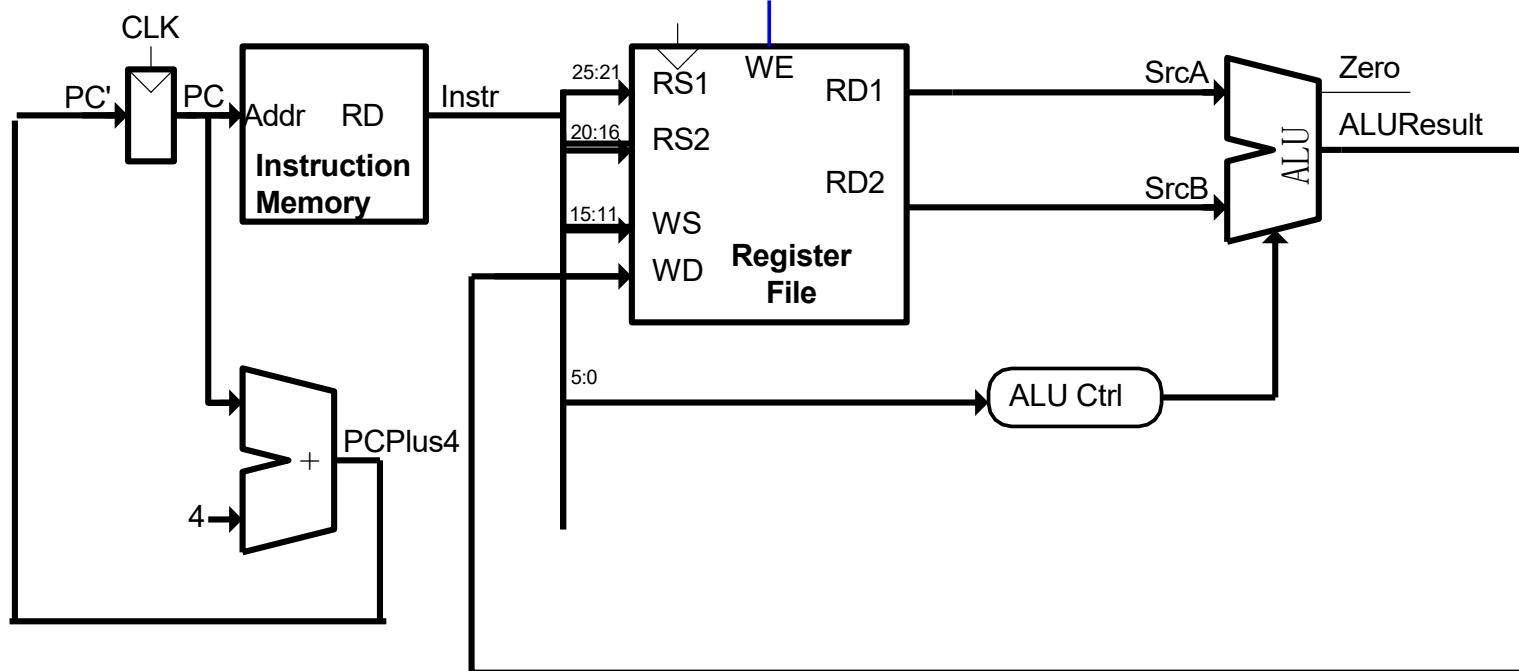
- Harvard架构：为指令和数据设计完全独立的物理存储单元以及信号传输路径
- Princeton架构：共用存储单元以及信号传输路径（冯·诺伊曼模型）



# ■ 数据通路：R型指令

□ MIPS指令可以分为三种类型

- R型指令(右图所示)
  - 仅通过寄存器完成数据运算或操作 (add、sub...)
- J型指令
  - 跳转指令，含opcode/目标地址字段，无寄存器字段
- I型指令
  - 立即数型指令，含立即数字段 (lw、sw、addi...)



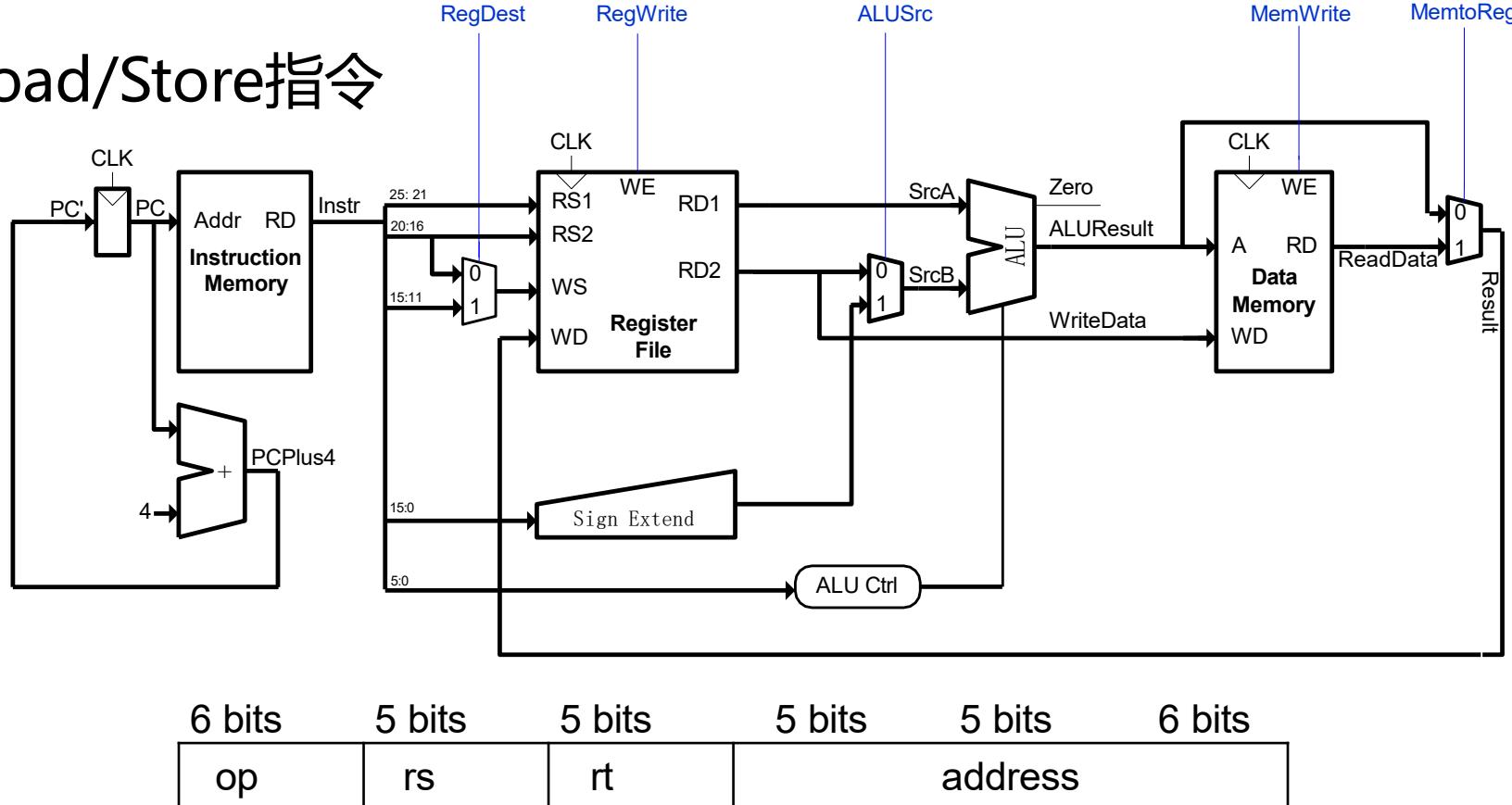
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	rs	rt	rd	shamt	funct

$$rd \leftarrow (rs \text{ funct})(rt)$$

# ■ 数据通路：I型指令

## □ I型指令

– 例如Load/Store指令



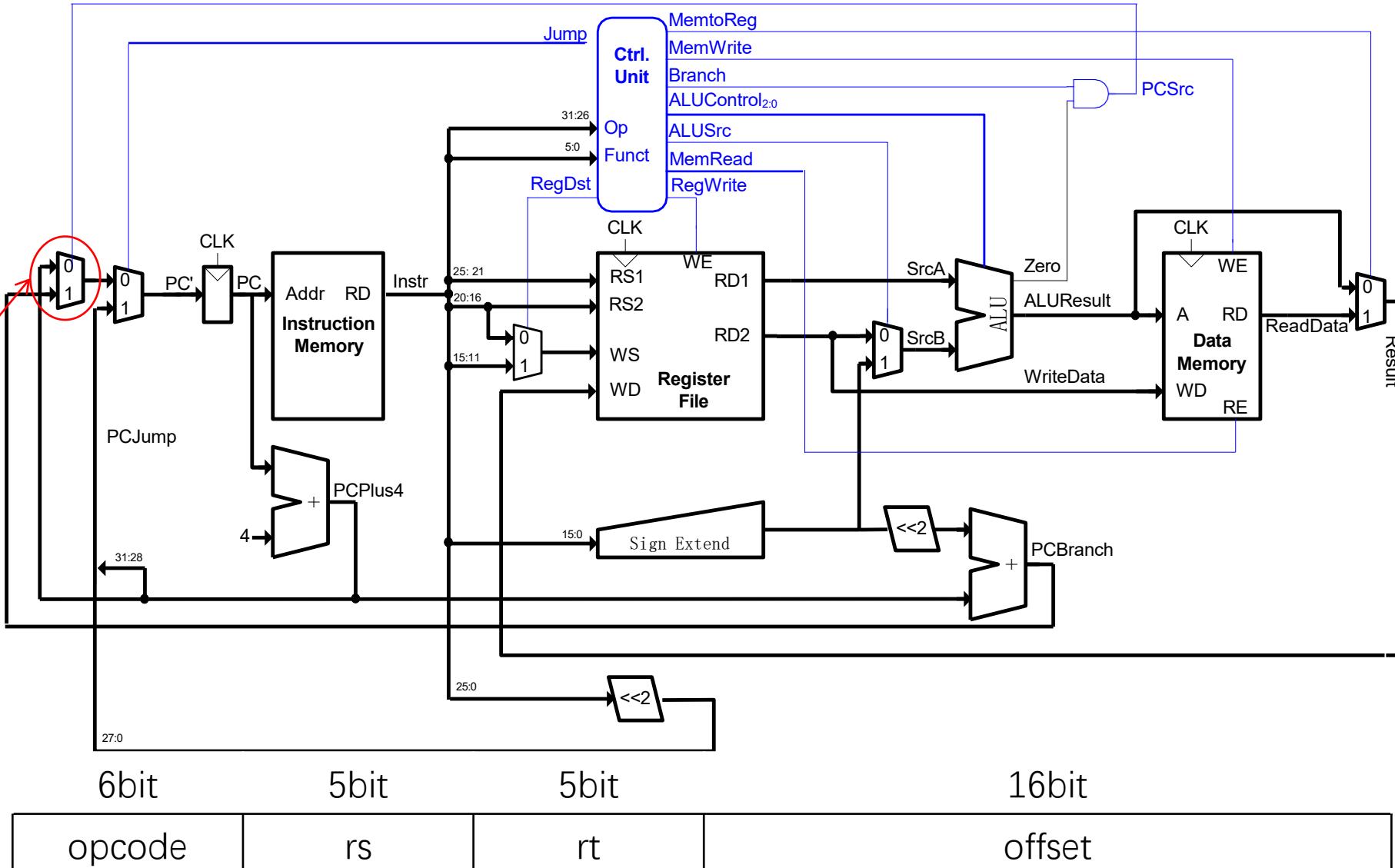
**rs**是基地址寄存器

**rt**是Load指令的目标寄存器 / Store指令的源寄存器

# ■ 数据通路：J型指令

## □ J型指令

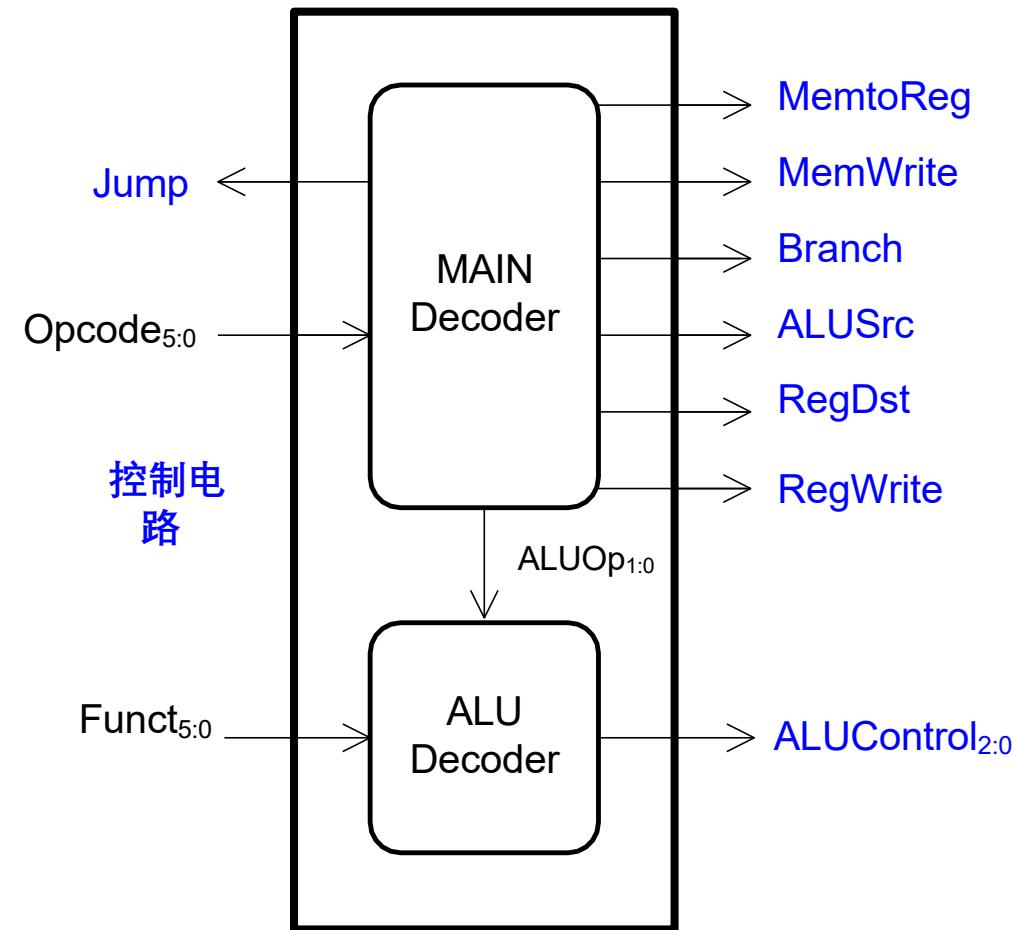
- 例如条件跳转指令
- 在此通过 PCSrc 信号决定下一条指令是否跳转



# ■ 控制电路

## □ 译码器

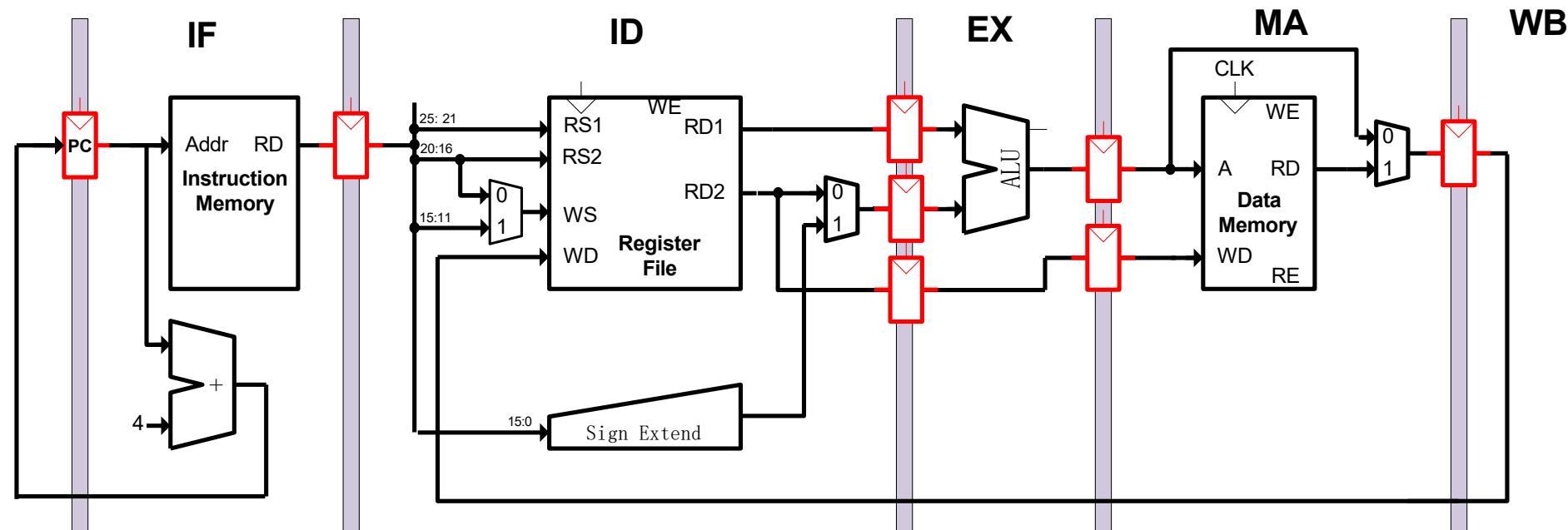
- 专注于解析指令，生成控制信号
- 根据当前状态和指令的opcode / funct，决定下一个状态
- 控制信号的设置取决于指令的操作码(opcode)
  - 例如Load型指令会使能RegDst和RegWrite信号
  - 如果是ALU操作指令，还会解析指令的funct部分生成ALUControl信号



# ■ 标准五级流水线

## □ 流水线

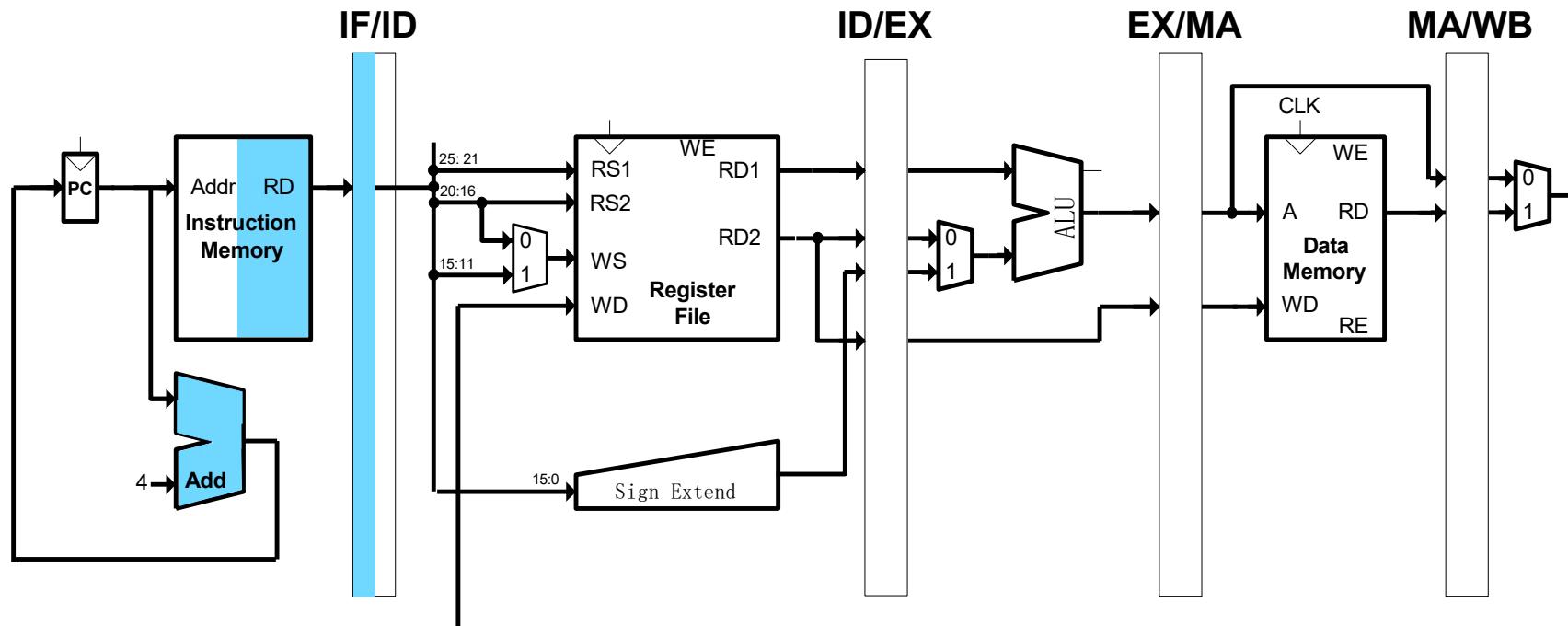
- 是一种能让多条指令重叠执行的技术
  - 属于指令级并行
- 静态并行: 编译器编译阶段分析指令之间的依赖关系, 调度可并行的指令
- 动态并行: CPU运行时动态检测指令依赖, 然后调度指令



# ■ 标准五级流水线

## □ 取指(Instruction Fetch, IF)

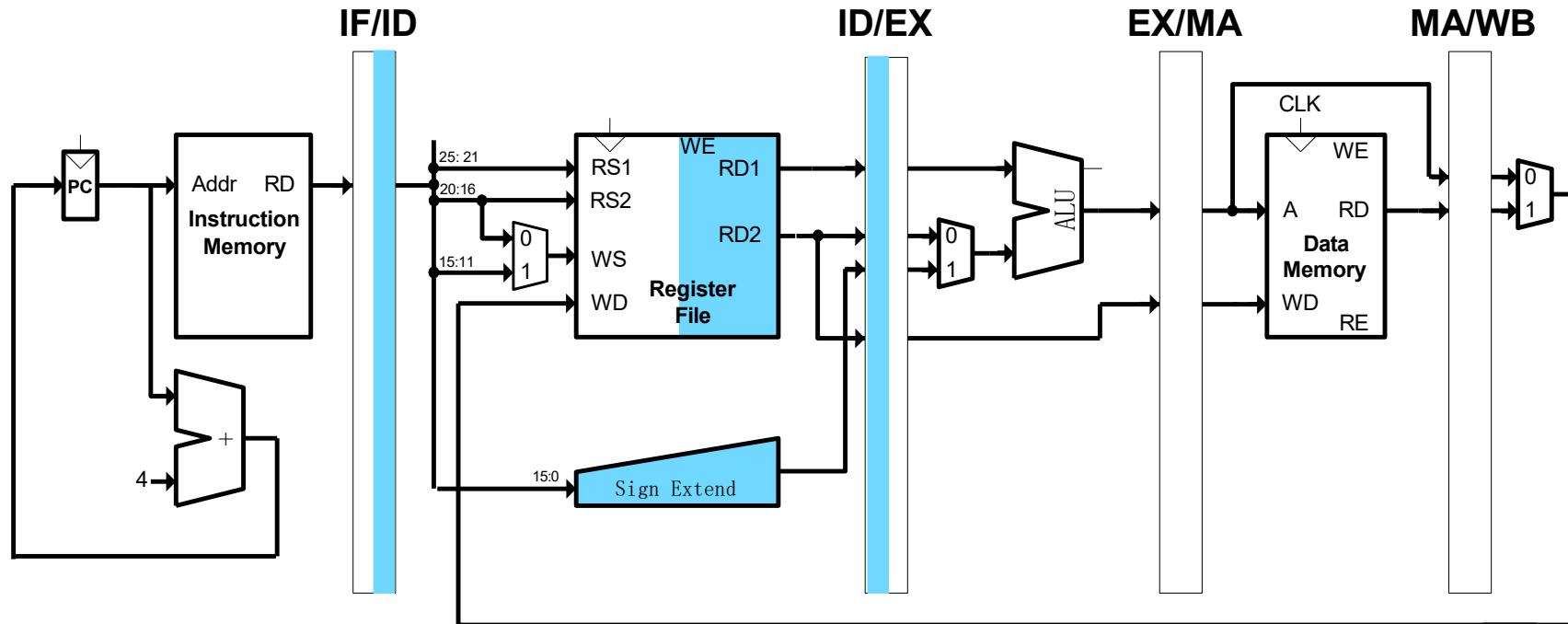
- 从指令存储器取出下一条要执行的指令
  - 蓝色区域所示，同时计算出下一条指令的PC地址



# ■ 标准五级流水线

## □ 译码(Instruction Decode, ID)

- 译码并从寄存器堆中读取源操作数
- 生成控制信号

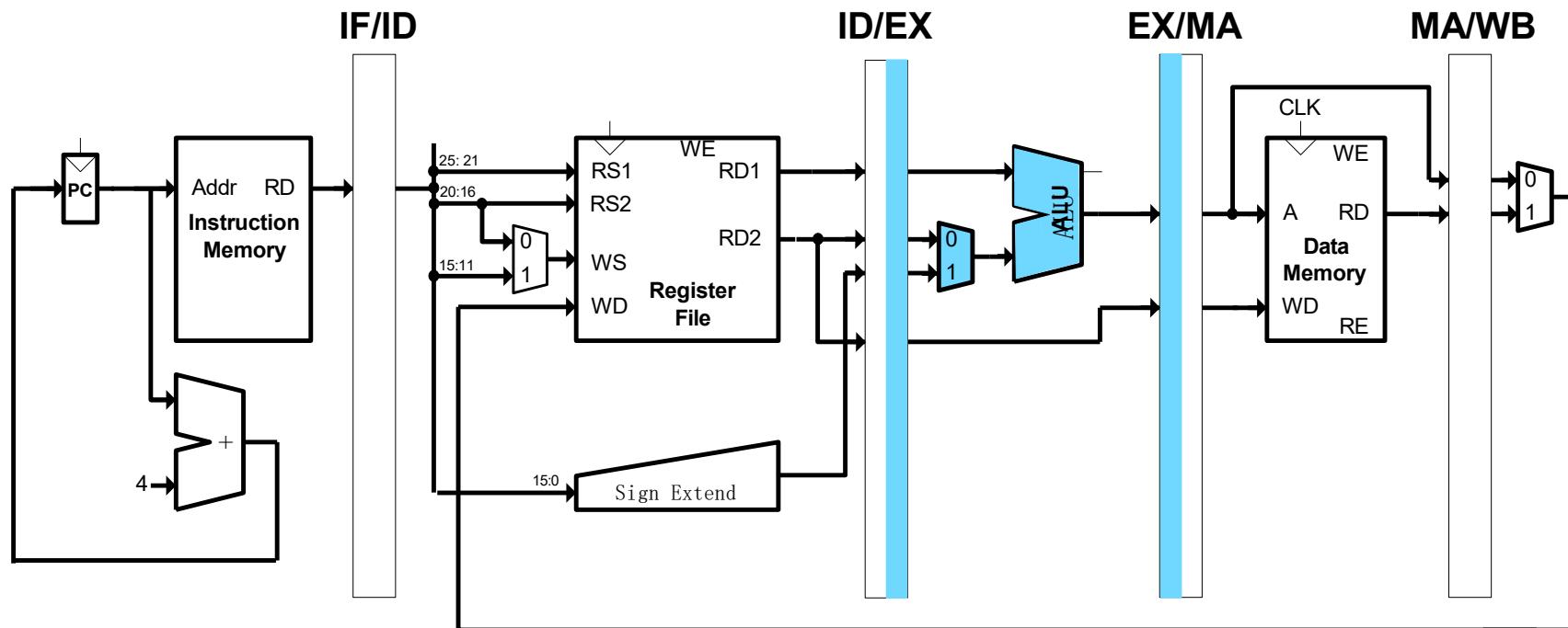


# ■ 标准五级流水线

## □ 执行(Excute, EX)

- 执行指令

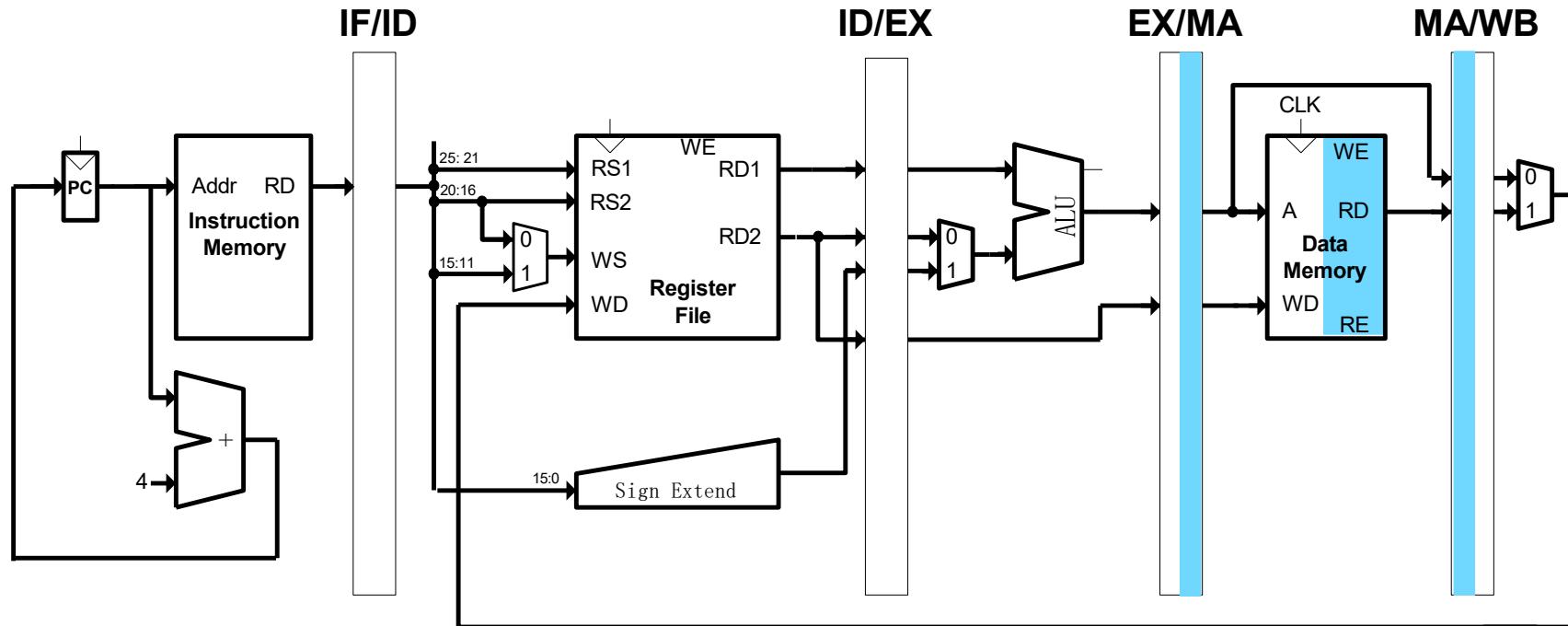
- R型指令就计算结果；J型指令需要计算下一指令的PC地址和是否跳转的判断结果



# ■ 标准五级流水线

## □ 访存(Memory access, MA)

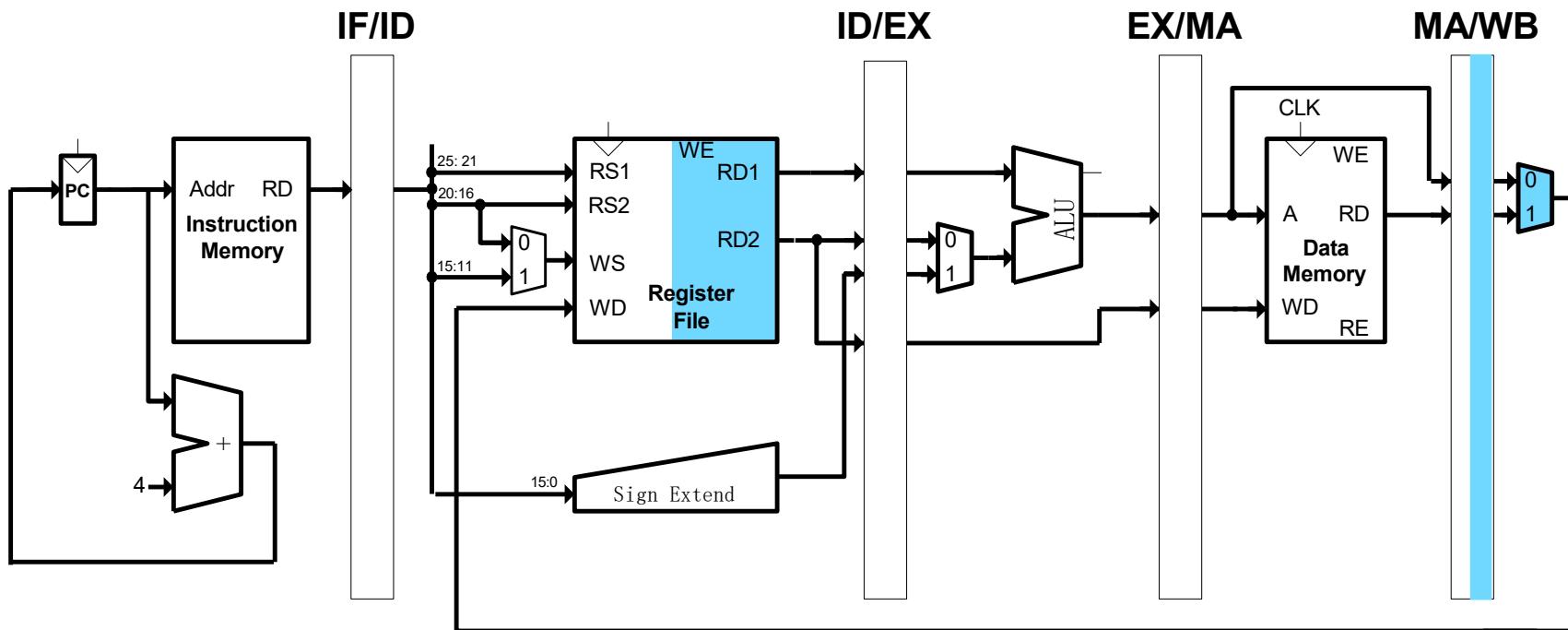
- 如果是Load/Store指令，那么EX阶段计算的是内存地址
- 在这里用计算得到的内存地址访问内存执行读写操作



# ■ 标准五级流水线

## □ 写回(Write back, WB)

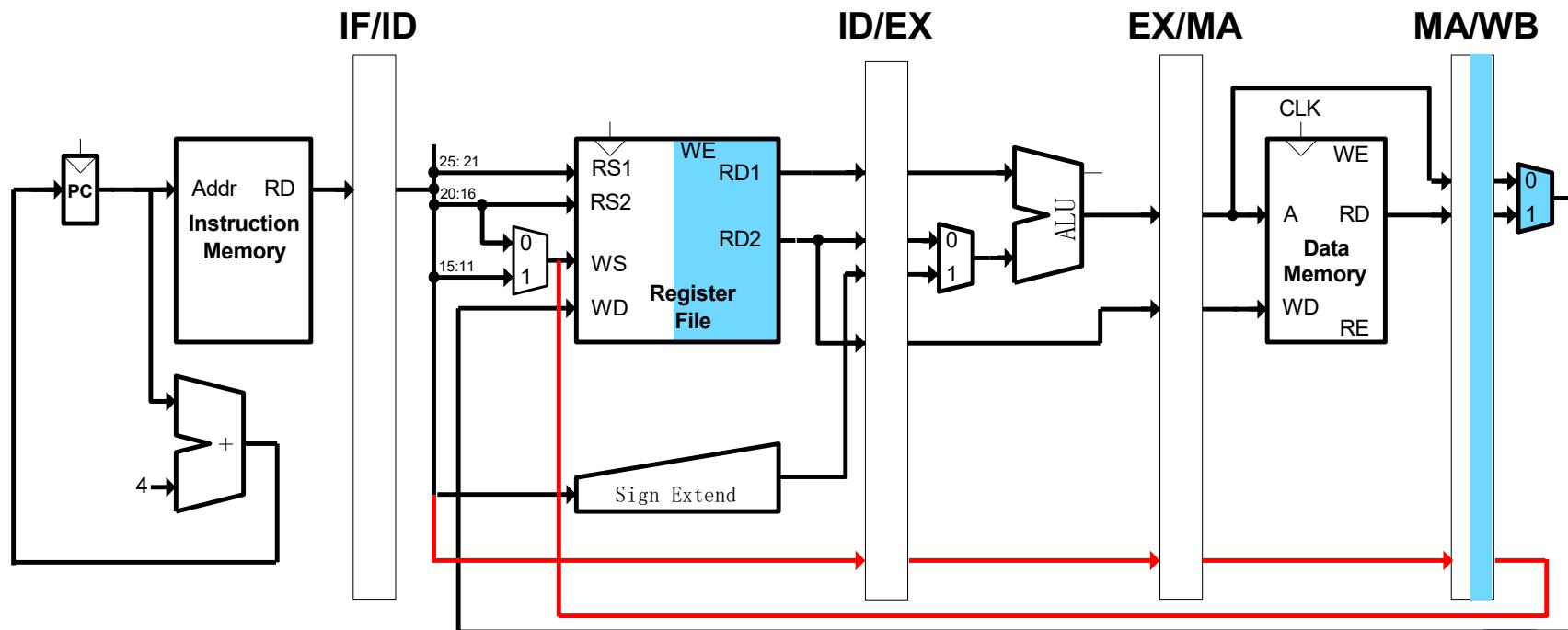
- 把执行结果写回寄存器堆



# ■ 标准五级流水线

## □ 目标寄存器传递

- 要保证WB阶段写入正确的寄存器，需要传递当前指令的目标寄存器信息。  
这条红色的线展示目标寄存器的数据通路



# ■ 流水线加速比

---

## □ 流水线加速比

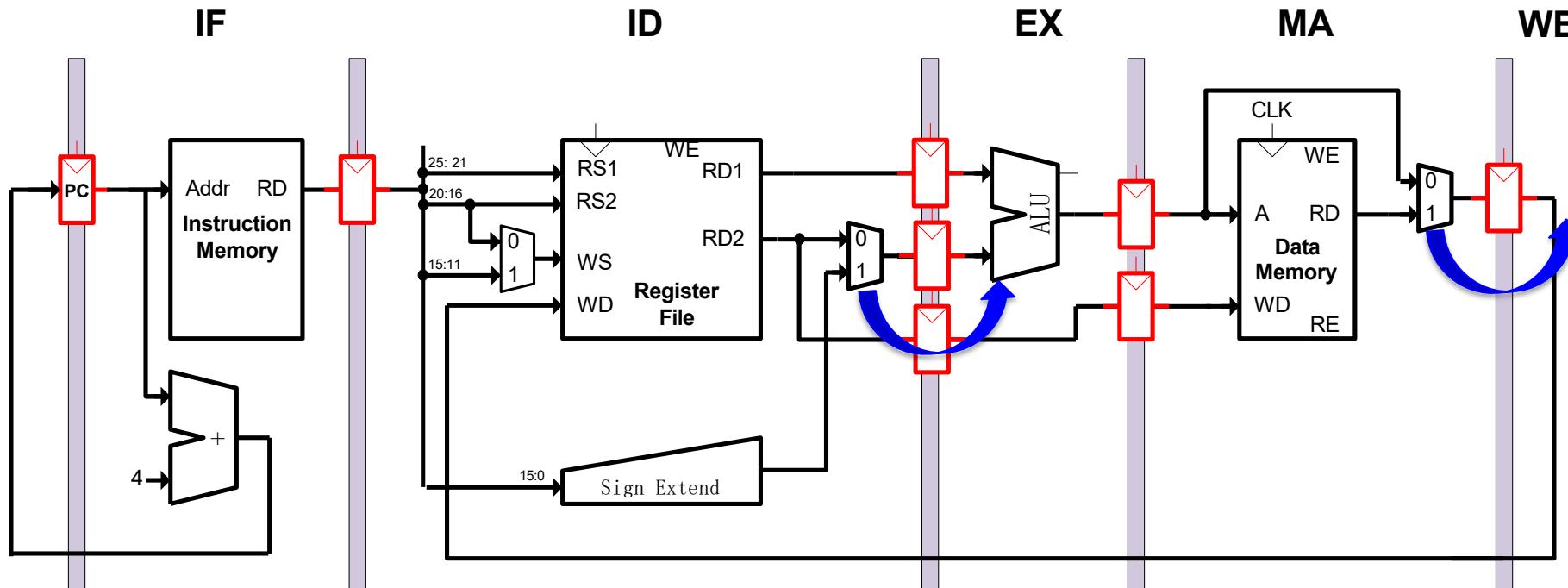
- 流水线的加速比来源于吞吐量的提升，单条指令执行时间没有变化
- 流水线的执行速度受制于最慢的阶段
- 理想情况下，流水线的加速比应等于阶段数
- 计算方法
  - 计算公式 $S = \frac{nmt}{mt+(n-1)t}$ ，其中 $n$ 是指令数量， $m$ 是流水线阶段数， $t$ 是每个阶段执行时间
  - 例如一条流水线有5个阶段，每个阶段都需要2纳秒执行，一共100条指令。不考虑冒险
  - 那么它的加速比是 $S = \frac{100*2*5}{5*2+(100-1)*2} = \frac{1000}{208} \approx 5$

# ■ 理想流水线

## □ 理想流水线具备的特质

- 每个阶段的执行时长一致 => 平衡流水线各部件使用率
- 指令之间完全独立 => 完全避免流水线冒险
- 相同的计算指令 => ?

思考：为什么统一的计算指令也是理想流水线的特点之一？



# ■ 流水段量化 (Stage Quantization)

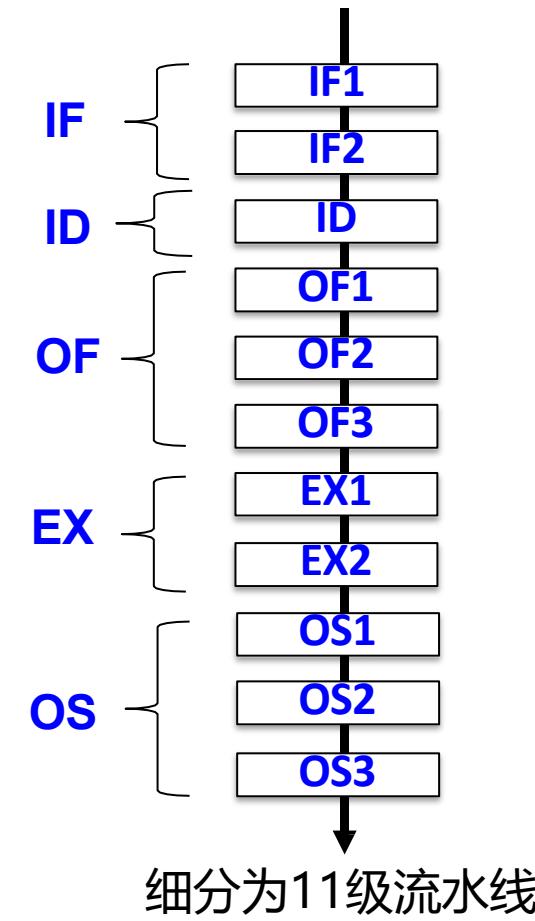
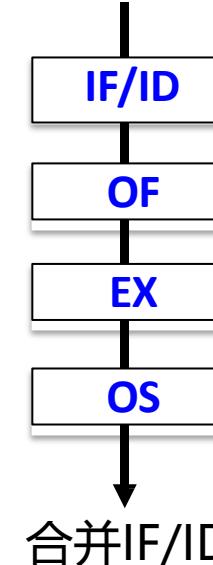
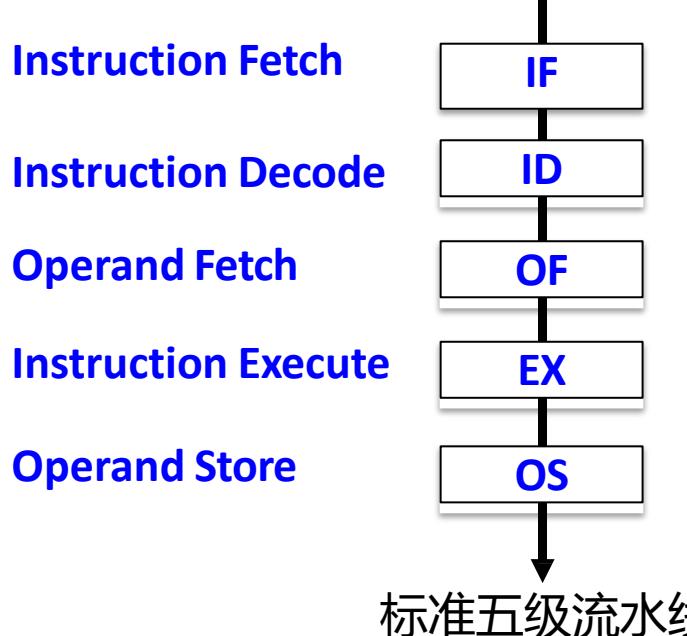
□ 可以修改标准五级流水线架构适应不同需求

- 合并多个子计算

➤ 把多个低延迟的子计算合并为一个流水段

- 细分一个子计算为多个

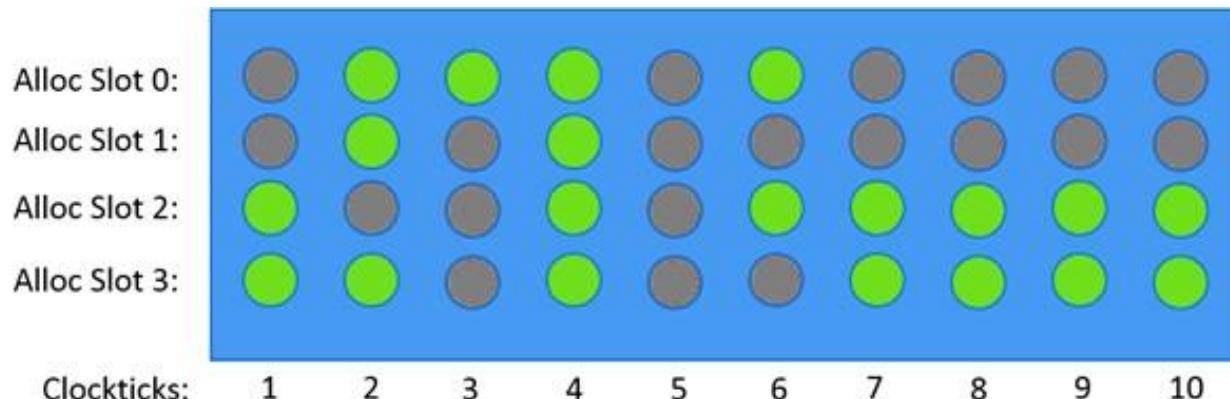
➤ 对子计算进行细粒度划分



# ■ 流水线槽

## □ 流水线槽(Pipeline Slot)

- 现代CPU的流水线已经相当复杂，流水线概念暂时简化为前端和后端两部分
    - 前端负责获取指令代码，将其解码为一个或多个低级硬件操作，称为micro-ops(uOps)，通过分配(Allocation)过程发送到后端
    - 后端监控uOp的操作数是否可用，并在空闲执行单元中执行uOp
  - 流水线槽表示处理一个uOp所需的硬件资源



# ■ 流水线槽

---

## □ 流水线槽停滞(stall)

- 在任意一个周期中，流水线槽可以被uOp填充，也可以是空的。如果流水线槽在这个周期为空，需要确定停滞原因以确定优化方向
  - 如果停滞是前端无法用uOp填充导致的，归类为前端绑定槽(Front-end)
  - 如果停滞是后端耗尽某些资源导致的，归类为后端绑定槽(Back-end)。注意如果同时发生前端和后端的停滞，应归类为后端绑定槽
    - 因为这种情况下修复前端停滞对于程序性能没有帮助，后端的瓶颈是限制性的
  - 如果uOp没有完成它的执行，应划分为Bad Speculation
  - 其余情况归类为Retiring

**感谢！**

---