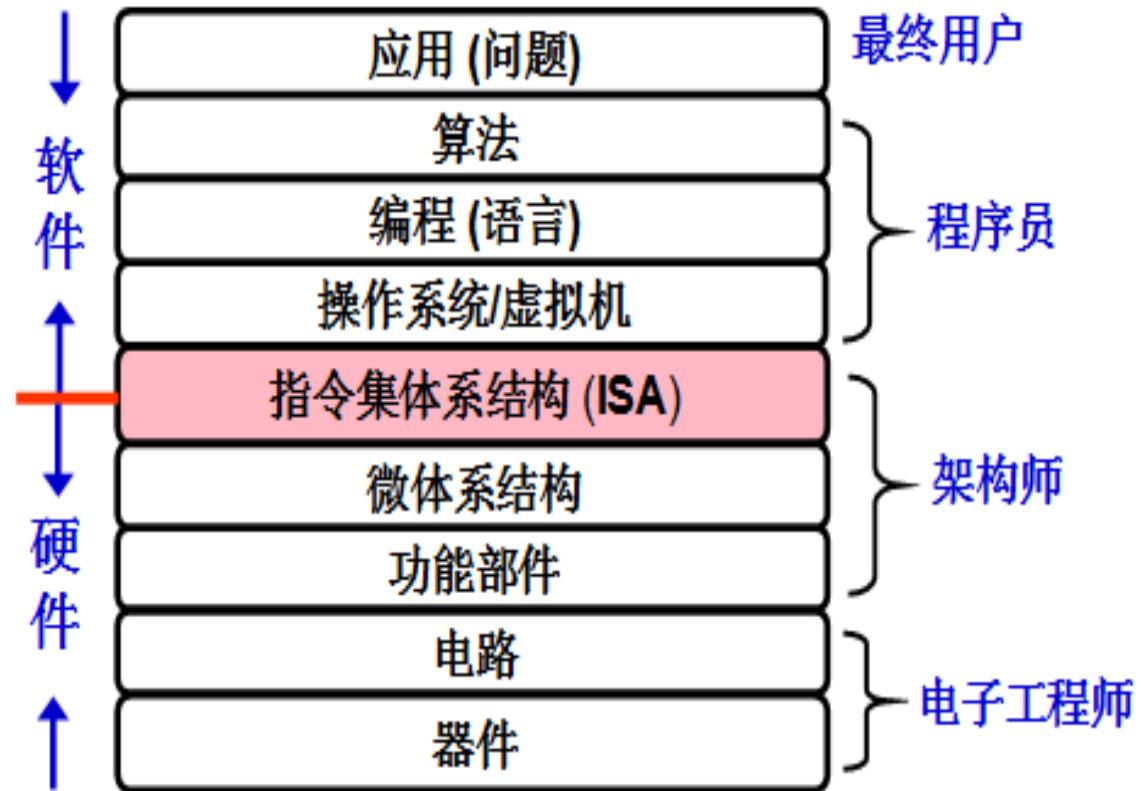


# 指令集在抽象层中的定位



# 第7章 指令系统

---

第一讲 概述与指令系统设计  
第二讲 指令系统实例：RISC-V架构



# 第一讲 概述与指令系统设计

---

## 1. 指令系统概述

## 2. 指令系统设计

1. 操作数及其寻址方式
2. 操作类型和操作码编码
3. 标志信息的生成与使用

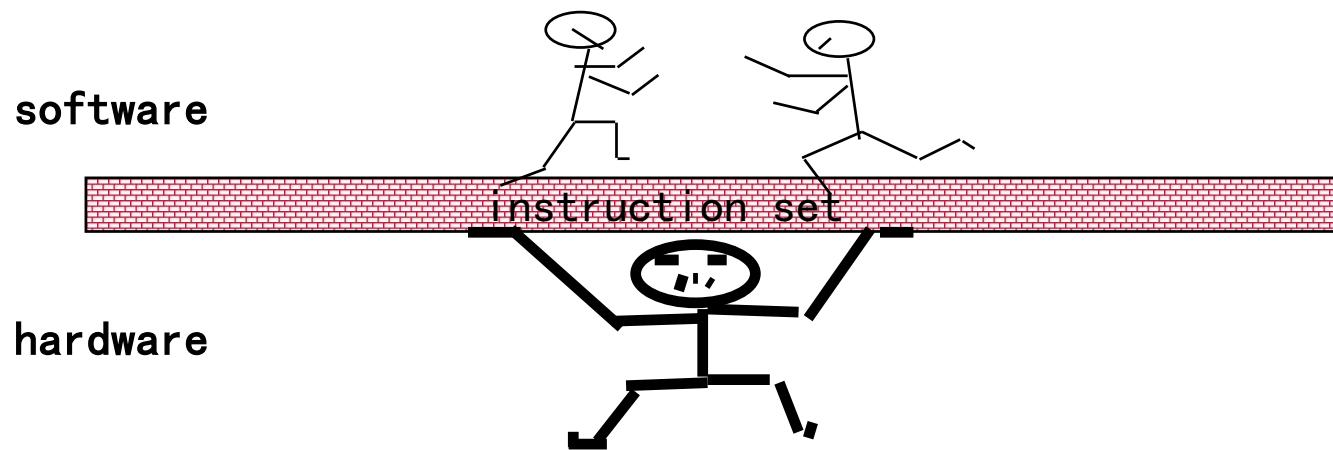
## 3. 指令设计风格

## 4. 异常和中断处理机制



# 指令系统概述

- 指令系统处在软/硬件交界面，同时被硬件设计者和系统程序员看到
- 硬件设计者角度：指令系统为CPU提供功能需求，要求易于硬件设计
- 系统程序员角度：通过指令系统来使用硬件，要求易于编写编译器
- 指令系统设计的好坏还决定了：计算机的性能和成本



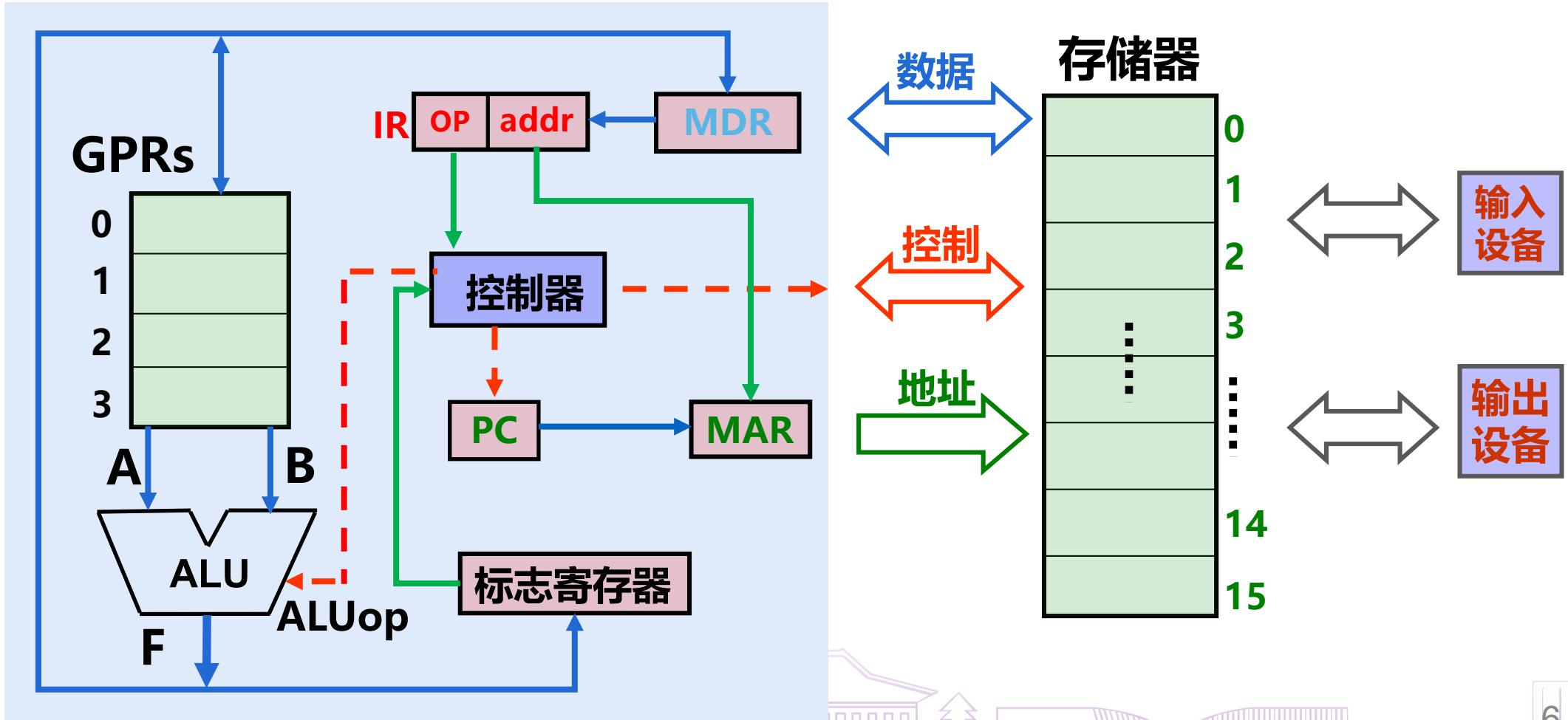
## 冯·诺依曼结构机器对指令规定：

- 用二进制表示，和数据一起存放在主存中
- 由两部分组成：操作码和操作数（或其地址码）

# 指令执行需要的计算机结构

8位定长指令字，4个GPR，16个主存单元

中央处理器 (CPU)



# 程序和指令执行过程举例

## 8位定长指令字，4个GPR，16个主存单元

假设模型机M中8位指令，格式有两种：R型、M型

格式	4位	2位	2位	功能说明
Register R型	Op	rt	rs	$R[rt] \leftarrow R[rt] \text{ op } R[rs]$ 或 $R[rt] \leftarrow R[rs]$
Memory M型	Op	Addr		$R[0] \leftarrow M[\text{addr}]$ 或 $M[\text{addr}] \leftarrow R[0]$

rs和rt为通用寄存器编号；addr为主存单元地址

R型：op=0000，寄存器间传送（mov）；op=0001，加（add）

M型：op=1110，取数（load）；op=1111，存数（store）



# 第一讲 概述与指令系统设计

---

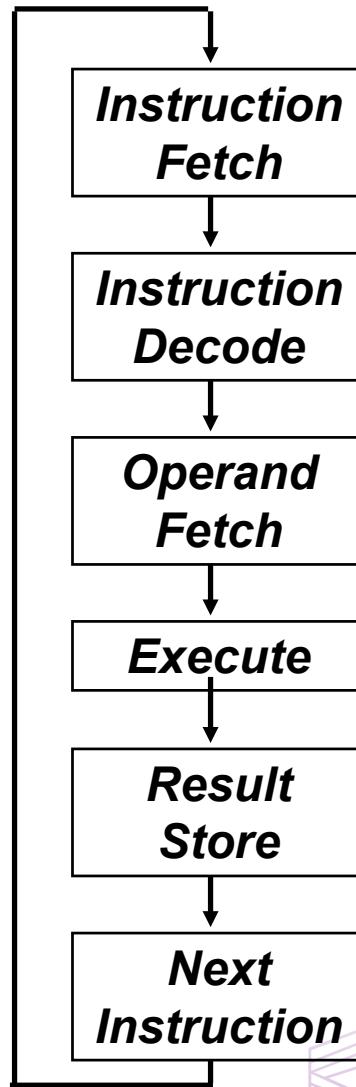
1. 指令系统概述

2. 指令系统设计

1. 操作数及其寻址方式
2. 操作类型和操作码编码
3. 标志信息的生成与使用
3. 指令设计风格
4. 异常和中断处理机制



# 从指令执行周期看指令设计涉及的问题



从存储器取指令  
指令地址、指令长度（定长/变长）

对指令译码，以确定将要做什么操作  
指令格式、操作码编码、操作数类型

计算操作数地址并取操作数  
地址码、寻址方式、操作数格式和存放

进行相应计算，并得到标志位  
操作类型、标志或条件码

将计算结果保存到目的地  
结果数据位置（目的操作数）

计算下条指令地址（通常和取指令同时进行）  
下条指令地址（顺序 / 转移）

指令执行的每一步都可能发生异常或中断，因此，指令集系统架构（ISA）还需要考虑异常和中断机制



# 一条指令包含的信息

---

问题：一条指令必须明显或隐含包含的信息有哪些？

操作功能：用操作码表示，指定操作类型

(操作码长度：固定 / 可变)

源操作数参照：一个或多个源操作数所在的地址

(操作数来源：主(虚)存/寄存器/I/O端口/指令本身)

结果值参照：产生的结果存放何处（目的操作数）

(结果地址：主(虚)存/寄存器/I/O端口)

下一条指令地址：下条指令存放何处

(下条指令地址：主(虚)存)

(正常情况隐含在PC中，改变顺序时由指令给出)



# 一条指令中应该有几个地址码字段？

## 零地址指令

- (1) 无需操作数 如：空操作 / 停机等
- (2) 所需操作数为默认的 如：堆栈 / 累加器等

形式：

OP
----

## 一地址指令

其地址既是操作数的地址，也是结果的地址

- (1) 单目运算：如：取反 / 取负等
- (2) 双目运算：另一操作数为默认的 如：累加器等

形式：

OP	A1
----	----

## 二地址指令（最常用）

分别存放双目运算中两个操作数，并将其中一个地址作为结果的地址。

形式：

OP	A1	A2
----	----	----

## 三地址指令

分别作为双目运算中两个源操作数的地址和一个结果的地址。

形式：

OP	A1	A2	A3
----	----	----	----

## 多地址指令

用于成批数据处理的指令，如：向量 / 矩阵等运算的SIMD指令。

# 指令格式的设计

## 指令格式的选择应遵循的几条基本原则

- 应尽量短
- 要有足够的操作码位数
- 指令编码必须有唯一的解释，否则是不合法的指令
- 指令字长应是字节的整数倍
- 合理地选择地址字段的个数
- 指令尽量规整

一般通过对操作码进行不同的编码来定义不同的含义，操作码相同时，再由功能码定义不同的含义！



# 指令格式的设计

---

## 与指令集设计相关的重要方面

- 操作码的全部组成：操作码个数 / 种类 / 复杂度  
LD/ST/INC/BRN 四种指令已足够编制任何可计算程序，但程序会很长
- 数据类型：对哪几种数据类型完成操作
- 指令格式：指令长度 / 地址码个数 / 各字段长度
- 通用寄存器：个数 / 功能 / 长度
- 寻址方式：操作数地址的指定方式
- 下条指令的地址如何确定：顺序，PC+1；条件转移；无条件转移；……
- 异常和中断机制，包括存储保护方式等



# 第一讲 概述与指令系统设计

---

1. 指令系统概述

2. 指令系统设计

1. 操作数及其寻址方式

2. 操作类型和操作码编码

3. 标志信息的生成与使用

3. 指令设计风格

4. 异常和中断处理机制



# 操作数类型和存储方式

操作数是指令处理的对象，与高级语言数据类型对应，基本类型有哪些？

## 地址（指针）

被看成无符号整数，用来参加运算以确定主（虚）存地址

## 数值数据

定点数（整数）：一般用二进制补码表示

浮点数（实数）：大多数机器采用 IEEE754 标准

十进制数：用 NCD 码表示，压缩/非压缩（汇编程序设计时用）

## 位、位串、字符和字符串

用来表示文本、声音和图像等

- 4 bits is a nibble (一个十六进制数字)
- 8 bits is a byte
- 16 bits is a half-word
- 32 bits is a word

## 逻辑（布尔）数据

按位操作（0-假 / 1-真）

操作数存放在寄存器或内存单元中，也可以立即数的方式直接出现在指令中



# IA-32 & RISC-V Data Type

---

## ➤ IA-32

- 基本类型：字节、字(16位)、双字(32位)、四字(64位)
- 整数：
  - 16位、32位、64位三种2-补码表示的整数
  - 18位压缩8421 BCD码表示的十进制整数
- 无符号整数 (8、16或32位)
- 近指针：32位段内偏移（有效地址）
- 浮点数：IEEE 754 (80位扩展精度浮点数寄存器)

## ➤ RISC-V

- 基本类型：字节、半字(16位)、字(32位)、四字(64位)
- 整数： 16位、32位、64位三种2-补码表示的整数
- 无符号整数： 16位、32位、64位整数
- 浮点数： IEEE 754 (32位/64位浮点数寄存器)



# Addressing Modes (寻址方式)

## ➤ 什么是“寻址方式”？

指令或操作数地址的指定方式。即：根据地址找到指令或操作数的方法。

## ➤ 地址码编码由操作数的寻址方式决定

## ➤ 地址码编码原则：

指令地址码尽量短

为什么？

目标代码短，省空间

操作数存放位置灵活，空间应尽量大

利于编译器优化产生高效代码

地址计算过程尽量简单

指令执行快

## ➤ 指令的寻址----简单

正常：PC增值

跳转（jump / branch / call / return）：同操作数的寻址

## ➤ 操作数的寻址----复杂（想象一下高级语言程序中操作数情况多复杂）

操作数来源：寄存器 / 外设端口 / 主(虚)存 / 栈顶

操作数结构：位 / 字节 / 半字 / 字 / 双字 / 一维表 / 二维表 / ...



通常寻址方式指“操作数的寻址方式”

# Addressing Modes (寻址方式)

---

## ➤ 寻址方式的确定

### (1) 没有专门的寻址方式位 (由操作码确定寻址方式)

如：MIPS指令，一条指令中最多仅有一个主(虚)存地址，且仅有一到两种寻址方式，Load/store型机器指令属于这种情况。

### (2) 有专门的寻址方式位

如：X86指令，一条指令中有多个操作数，且寻址方式各不相同，需要各自说明寻址方式，因此每个操作数有专门的寻址方式位。

## ➤ 有效地址的含义

操作数所在存储单元的地址（可能是逻辑地址或物理地址），可通过指令的寻址方式和地址码计算得到

## ➤ 基本寻址方式

立即 / 直接 / 间接 / 寄存器(直接) / 寄存器间接 / 偏移 / 栈

## ➤ 基本寻址方式的算法及优缺点

(见下页)



# 基本寻址方式的优缺点

假设: A=地址字段值, R=寄存器编号, EA=有效地址, (X)=X中的内容



方式	算法	主要优点	主要缺点
立即数	操作数=A	指令执行速度快	操作数幅值有限
直接	EA=A	有效地址计算简单	地址范围有限
间接	EA=(A)	有效地址范围大	多次存储器访问
寄存器(直接)	操作数=(R)	指令执行快, 指令短	地址范围有限
寄存器间接	EA=(R)	地址范围大	额外存储器访问
偏移	EA=A+(R)	灵活	复杂
栈	EA=栈顶	指令短	应用有限

偏移方式: 将直接方式和寄存器间接方式结合起来。

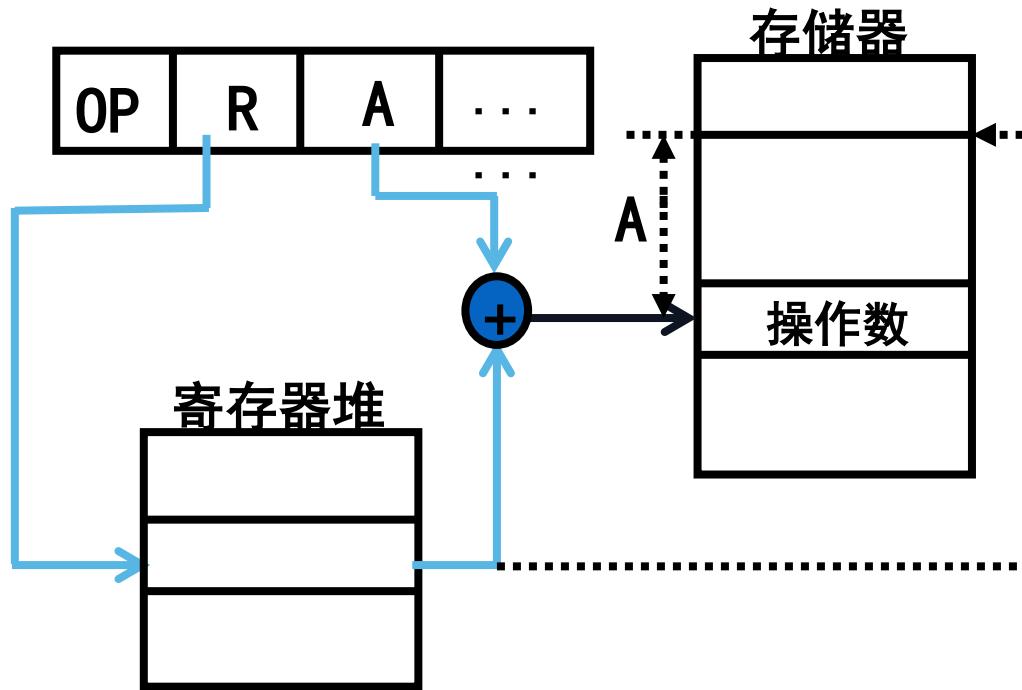
有: 相对 / 基址 / 变址三种 (见后面几页! )

问题: 以上各种寻址方式下, 操作数在寄存器中还是在存储器中?

需要计算有效地址EA的操作数都是在存储器中

# 偏移寻址方式

指令中给出的地址码  
A称为**形式地址**



偏移寻址:  $EA = A + (R)$      $R$ 可以明显给出, 也可以隐含给出

$R$ 可以为**PC**、**基址寄存器B**、**变址寄存器 I**

- 相对寻址:  $EA = A + (PC)$     相对于**当前指令**处位移量为A的单元 (实现公共子程序)

- 基址寻址:  $EA = A + (B)$     相对于**基址 (B)**处位移量为A的单元 (多程序段转移)

- 变址寻址:  $EA = A + (I)$     相对于**首址A**处位移量为(I)的单元 (实现数组)

# 变址寻址实现线性表元素的存取

## 自动变址

指令中的地址码A给定数组首址，变址器I每次自动加/减数组元素的长度x。

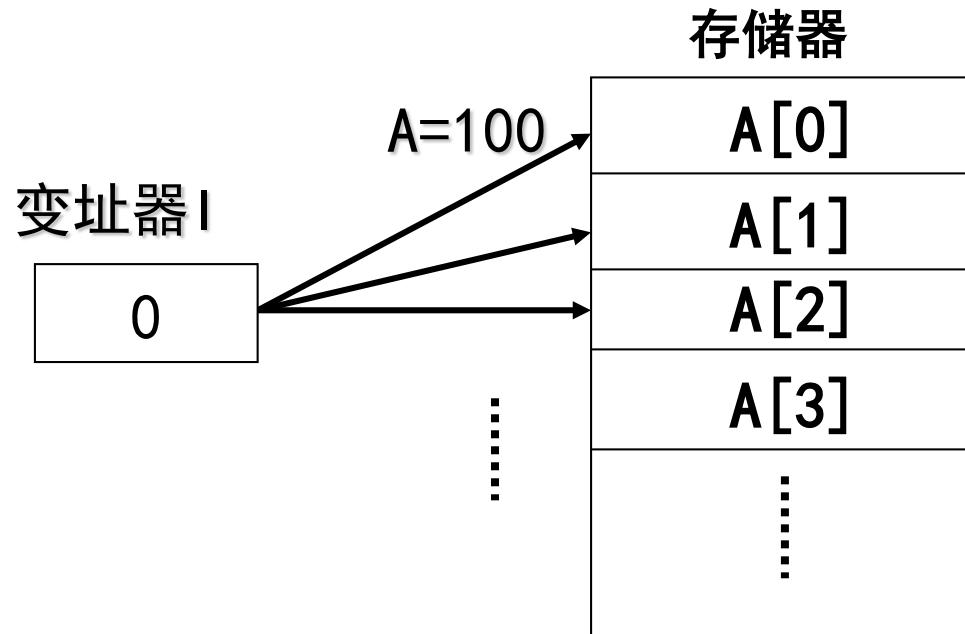
$$EA = A + (I)$$

$$I = (I) \pm x$$

例如，X86中的串操作指令

- 对于“`for (i=0; i<N; i++) ...`”，  
即地址从低→高变化：加
  - 对于“`for (i=N-1; i>=0; i--) ...`”，  
即地址从高→低变化：减
- 可提供对线性表的方便访问

假定一维数组A从100号单元开始



假设按字节编址，则：

每个元素为一个字节时，  $I = (I) \pm 1$

每个元素为4个字节时，  $I = (I) \pm 4$

一般RISC机器不提供自动变址寻址，并将变址和基址寻址统一成一种偏移寻址方式

# 存储器操作数的寻址方式

```
int x;  
float a[100];  
short b[4][4];  
char c;  
double d[10];
```

a[i] 的地址如何计算?

$$104 + i \times 4$$

i=99时,  $104 + 99 \times 4 = 500$

b[i][j] 的地址如何计算?

$$504 + i \times 8 + j \times 2$$

i=3、j=2时,  $504 + 24 + 4 = 532$

d[i] 的地址如何计算?

$$544 + i \times 8$$

i=9时,  $544 + 9 \times 8 = 616$

b31		
b0		
	d[9]	616
	⋮	
	d[0]	544
	⋮	
	c	536
b[3][3]	b[3][2]	532
⋮		
b[0][1]	b[0][0]	504
⋮		
a[99]		500
⋮		
a[0]		104
x		100
⋮		

# 第一讲 概述与指令系统设计

---

1. 指令系统概述
2. 指令系统设计
  1. 操作数及其寻址方式
  2. 操作类型和操作码编码
  3. 标志信息的生成与使用
3. 指令设计风格
4. 异常和中断处理机制



# 操作类型与操作编码

- 算术和逻辑运算指令
  - 加、减、乘、除、比较、与、或、非、取反、取负、异或等
  - 加算术运算包含整数和浮点数操作
- 移位指令
  - 算术移位、逻辑移位、循环移位、半字交换
- 数据传送指令
  - 寄存器间、内存-寄存器、内存内换
- 顺序控制指令
  - 条件转移、无条件转移
  - 调用、返回
- 系统控制指令
  - 停机、开中断、关中断、系统模式切换等
- 输入/输出指令

# Instruction Format (指令格式)

---

- 操作码的编码有两种方式
  - Fixed Length Opcodes (定长操作码法)
  - Expanding Opcodes (扩展操作码编法)
- instructions size
  - 代码长度更重要时：采用变长指令字、变长操作码      为什么？
  - 性能更重要时：采用定长指令字、定长操作码

变长指令字和变长操作码使机器代码更紧凑；定长指令字和定长操作码便于快速访问和译码。学了CPU设计就更明白了。

问题：是否可以有定长指令字、变长操作码？定长操作码、变长指令字呢？

指令长度是否可变与操作码长度是否可变没有绝对关系，但通常是“定长操作码不一定是定长指令字”、“变长操作码一般是变长指令字”。



# 定长操作码编码 Fixed Length Opcodes

## 基本思想

指令的操作码部分采用固定长度的编码

如：假设操作码固定为6位，则系统最多可表示64种指令

## 特点

译码方便，但有信息冗余

## 举例

IBM360/370采用：

8位定长操作码，最多可有256条指令

只提供了183条指令，有73种编码为冗余信息

机器字长32位，按字节编址

有16个32位通用寄存器，基址器B和变址器X可用其中任意一个

问题：通用寄存器编号有几位？B和X的编号占几位？都是4位！

# IBM 370 指令格式

RR型



格式：定长操作码、变长指令字

RX型



R<sub>i</sub>：寄存器

X：变址器

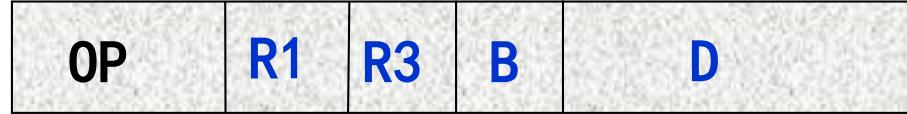
B<sub>i</sub>：基址器

D<sub>i</sub>：位移量

I：立即数

L：数的长度

RS型



SI型



SS型



RR：寄存器 - 寄存器

RX：寄存器 - 变址存储器

SS：基址存储器 - 基址存储器

SI：基址存储器 - 立即数

RS：寄存器 - 基址存储器

# 扩展（变长）操作码编码 Expanding Opcodes

## 基本思想

将操作码的编码长度分成几种固定长的格式。被大多数指令集采用。PDP-11是典型的变长操作码机器。

## 种类

等长扩展法：4-8-12；3-6-9；…… / 不等长扩展法

## 举例说明如何扩展

设某指令系统指令字是16位，每个地址码为6位。若二地址指令15条，一地址指令34条，则剩下零地址指令最多有多少条？

解：操作码按短到长进行扩展编码

二地址指令：(0000 ~ 1110)

一地址指令：11110 (00000 ~ 11111)；11111 (00000 ~ 00001)

零地址指令：11111 (00010 ~ 11111) (000000 ~ 111111)

故零地址指令最多有  $30 \times 2^6 = 15 \times 2^7$  种

# 第一讲 概述与指令系统设计

---

1. 指令系统概述
2. 指令系统设计
  1. 操作数及其寻址方式
  2. 操作类型和操作码编码
  3. 标志信息的生成与使用
3. 指令设计风格
4. 异常和中断处理机制



# 条件测试方式

条件转移指令通常根据Condition Codes (条件码 CC/ 状态位 / 标志位) 转移  
通过执行算术指令或显式地由比较和测试指令来设置CC

ex: sub r1, r2, r3 ; r2和r3相减, 结果在r1中, 并生成标志位ZF、CF等  
bz label ; 标志位ZF=1时转到label处执行; 否则顺序执行

$z=x-y;$   
`if (z==0) goto label;`

常用的标志 (条件码) 有四种

SF - negative      OF - overflow  
CF - 进位/借位      ZF - zero

带符号和无符号整数运算, 标志生成  
方式完全一样!

标志可存于标志寄存器/条件码寄存器/状态寄存器/程序状态字寄存器

bgt的条件?

也可由指定的通用寄存器来存放状态位

无符号数: (ZF=0) & (CF=0)

Ex: cmp r1, r2, r3 ; 比较r2和r3, 标志位存储在r1中

带符号整数: (ZF=0) & (SF=OF)

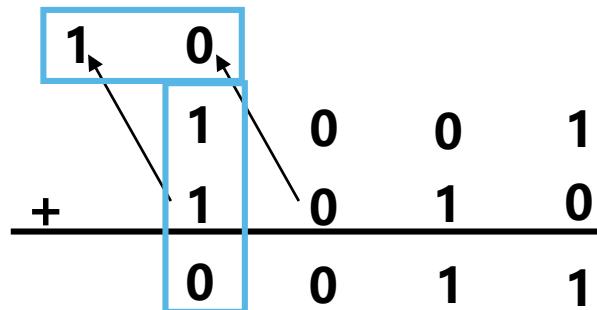
bgt r1, label ; 根据r1判断是否大于0, 是则转移到label处

可以将两条指令合成一条指令, 即: 计算并转移

Ex: bgt r1, r2, label ; 如果r1>r2, 则转移到label处执行; 否则顺序执行

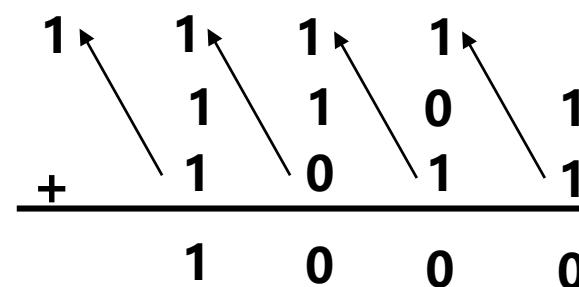
# 回顾：标志信息是干什么的？

$$\begin{array}{r} -7 - 6 = -7 + (-6) = +3 \quad \text{X} \\ 9 - 6 = 3 \quad \checkmark \end{array}$$



OF=1、ZF=0  
SF=0、借位CF=0

$$\begin{array}{r} -3 - 5 = -3 + (-5) = -8 \quad \checkmark \\ 13 - 5 = 8 \quad \checkmark \end{array}$$



OF=0、ZF=0、  
SF=1、借位CF=0

做减法以比较大小，规则：  
**Unsigned:** CF=0时，大于  
**Signed:** OF=SF时，大于

验证： $9 > 6$ , 故CF=0;  $13 > 5$ , 故CF=0

验证： $-7 < 6$ , 故OF≠SF  
 $-3 < 5$ , 故OF≠SF

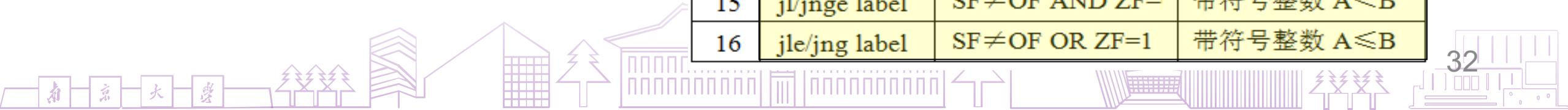


# IA-32中的条件转移指令

序号	指令	转移条件	说明
1	jc label	CF=1	有进位/借位
2	jnc label	CF=0	无进位/借位
3	je/jz label	ZF=1	相等/等于零
4	jne/jnz label	ZF=0	不相等/不等于零
5	js label	SF=1	是负数
6	jns label	SF=0	是非负数
7	jo label	OF=1	有溢出
8	jno label	OF=0	无溢出
9	ja/jnbe label	CF=0 AND ZF=0	无符号整数 A>B
10	jae/jnb label	CF=0 OR ZF=1	无符号整数 A≥B
11	jb/jnae label	CF=1 AND ZF=0	无符号整数 A<B
12	jbe/jna label	CF=1 OR ZF=1	无符号整数 A≤B
13	jg/jnle label	SF=OF AND ZF=0	带符号整数 A>B
14	jge/jnl label	SF=OF OR ZF=1	带符号整数 A≥B
15	jl/jnge label	SF≠OF AND ZF=	带符号整数 A<B
16	jle/jng label	SF≠OF OR ZF=1	带符号整数 A≤B

分三类：

- (1) 根据单个标志的值转移
- (2) 按无符号整数比较转移
- (3) 按带符号整数比较转移



# 第一讲 概述与指令系统设计

---

1. 指令系统概述
2. 指令系统设计
  1. 操作数及其寻址方式
  2. 操作类型和操作码编码
  3. 标志信息的生成与使用
3. 指令设计风格
4. 异常和中断处理机制



# 指令设计风格

## Accumulator: (earliest machines) 累加器型

特点：其中一个操作数（源操作数1）和目的操作数总在累加器中

1 address                  add A                   $acc \leftarrow acc + mem[A]$

1 (+x) address            add x A             $acc \leftarrow acc + mem[A + x]$

## Stack: (e.g. HP calculator, Java virtual machines) 栈型

特点：总是将栈顶两个操作数进行运算，指令无需指定操作数地址

0 address                  add                   $tos \leftarrow tos + next$

## General Purpose Register: (e.g. IA-32, Motorola 68xxx) 通用寄存器型

特点：操作数可以是寄存器或存储器数据（即A、B和C可以是寄存器或存储单元）

2 address                  add A B             $EA(A) \leftarrow EA(A) + EA(B)$

3 address                  add A B C         $EA(A) \leftarrow EA(B) + EA(C)$

## Load/Store: (e.g. SPARC, MIPS, RISC-V) 装入/存储型

特点：运算操作数只能是寄存器数据，只有load/store能访问存储器

3 address    add Ra Rb Rc             $Ra \leftarrow Rb + Rc$

load Ra Rb               $Ra \leftarrow mem[Rb]$

store Ra Rb              $mem[Rb] \leftarrow Ra$

# 指令风格比较

对比不同风格： 指令字长？ 所需指令数量？ 指令执行周期？

关于  $C = A + B$  四种指令集的执行伪代码：

栈型	累加器型	寄存器型 (寄存器存储器通用型)	寄存器型 (装入存储型)
Push A	Load A	Load R1, A	Load R1, A
Push B	Add B	Add R1, B	Load R2, B
Add	Store C	Store C, R1	Add R3, R1, R2
Pop C			Store C, R3

指令条数较少

复杂表达式时，累加器型风格指令条数变多，因为所有运算都要用累加器，使得程序中多出许多移入 / 移出累加器的指令！

想象一下 “ $C=a*x+b*y+x*y$ ” 用累加器型风格实现的情况！

75年开始，寄存器型占主导地位，原因：

- 寄存器速度快，使用大量通用寄存器可减少访存操作
- 表达式编译时与顺序无关（相对于Stack）

# 寄存器使用方式的举例

每条典型ALU指令中的存储器地址个数

每条典型ALU指令中的最多操作数个数

Examples

0	3	SPARC, MIPS, Precision Architecture, Power PC, RISC-V
1	2	Intel 80x86, Motorola 68000
2	2	VAX (also has 3-operand formats)
3	3	VAX (also has 2-operand formats)

In VAX(CISC) : ADDL (R9), (R10), (R11) ;  $\text{mem}[R9] \leftarrow \text{mem}[R10] + \text{mem}[R11]$  一条指令!

In MIPS(RISC) :

哪种所需的时钟周期数(CPI)小?

MIPS!

Iw R1, (R10) :  $R1 \leftarrow \text{mem}[R10]$   
Iw R2, (R11) :  $R2 \leftarrow \text{mem}[R11]$   
add R3, R1, R2 :  $R3 \leftarrow R1 + R2$   
sw R3, (R9) :  $\text{mem}[R9] \leftarrow R3$

四条指令!



# 指令设计风格-按指令格式的复杂度来分

按指令格式的复杂度来分，有两种类型计算机：

复杂指令集计算机CISC (Complex Instruction Set Computer)

精简指令集计算机RISC (Reduced Instruction Set Computer)

早期CISC设计风格的主要特点

(1) 指令系统复杂

变长操作码 / 变长指令字 / 指令多 / 寻址方式多 / 指令格式多

(2) 指令周期长

绝大多数指令需要多个时钟周期才能完成

(3) 各种指令都能访问存储器

除了专门的存储器读写指令外，运算指令也能访问存储器

(4) 采用微程序控制

(5) 有专用寄存器

(6) 难以进行编译优化来生成高效目标代码

例如，VAX-11/780小型机

16种寻址方式；9种数据格式；303条指令；

一条指令包括1~2个字节的操作码和下续N个操作数说明符。一个说明符的长度达1~10个字节。37

# 复杂指令集计算机CISC

## ◆ CISC的缺陷

- 日趋庞大的指令系统不但使计算机的研制周期变长，而且难以保证设计的正确性，难以调试和维护，并且因指令操作复杂而增加机器周期，从而降低了系统性能。
- ◆ 1975年IBM公司开始研究指令系统的合理性问题，John Cocks提出精简指令系统计算机 RISC ( Reduce Instruction Set Computer )。

## • 对CISC进行测试，发现一个事实：

- 在程序中各种指令出现的频率悬殊很大，最常使用的一些简单指令，这些指令占程序的80%，但只占指令系统的20%。而且在微程序控制的计算机中，占指令总数20%的复杂指令占用了控制存储器容量的80%。
- 1982年美国加州伯克利大学的RISC-I，斯坦福大学的MIPS，IBM公司的IBM801相继宣告完成，这些机器被称为第一代RISC机。

# Top 10 80x86 Instructions

---

Rank	instruction	Integer Average Percent total executed
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	Total	96%

◦ Simple instructions dominate instruction frequency

( 简单指令占主要部分，使用频率高！ )

# RISC 设计风格的主要特点

## (1) 简化的指令系统

指令少 / 寻址方式少 / 指令格式少 / 指令长度一致

## (2) 以RR方式工作

除Load/Store指令可访问存储器外，其余指令都只访问寄存器。

## (3) 指令周期短

以流水线方式工作，因而除Load/Store指令外，其他简单指令都只需一个或一个不到的时钟周期就可完成。

## (4) 采用大量通用寄存器，以减少访存次数

## (5) 采用组合逻辑电路控制，不用或少用微程序控制

## (6) 采用优化的编译系统，力求有效地支持高级语言程序

MIPS、RISC(RISC-I到RISC-V) 系列架构是典型的RISC处理器，82年以来新的指令集大多采用 RISC体系结构

x86因为“兼容”的需要，保留了CISC的风格，同时也借鉴了RISC思想

# 第一讲 概述与指令系统设计

---

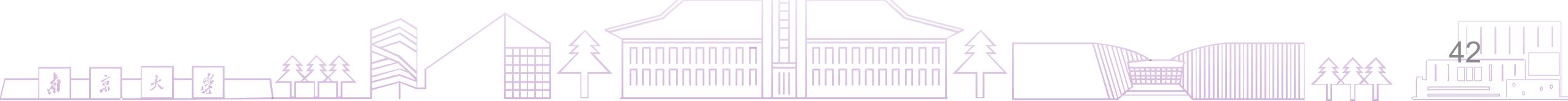
1. 指令系统概述
2. 指令系统设计
  1. 操作数及其寻址方式
  2. 操作类型和操作码编码
  3. 标志信息的生成与使用
3. 指令设计风格
4. 异常和中断处理机制



# 异常和中断

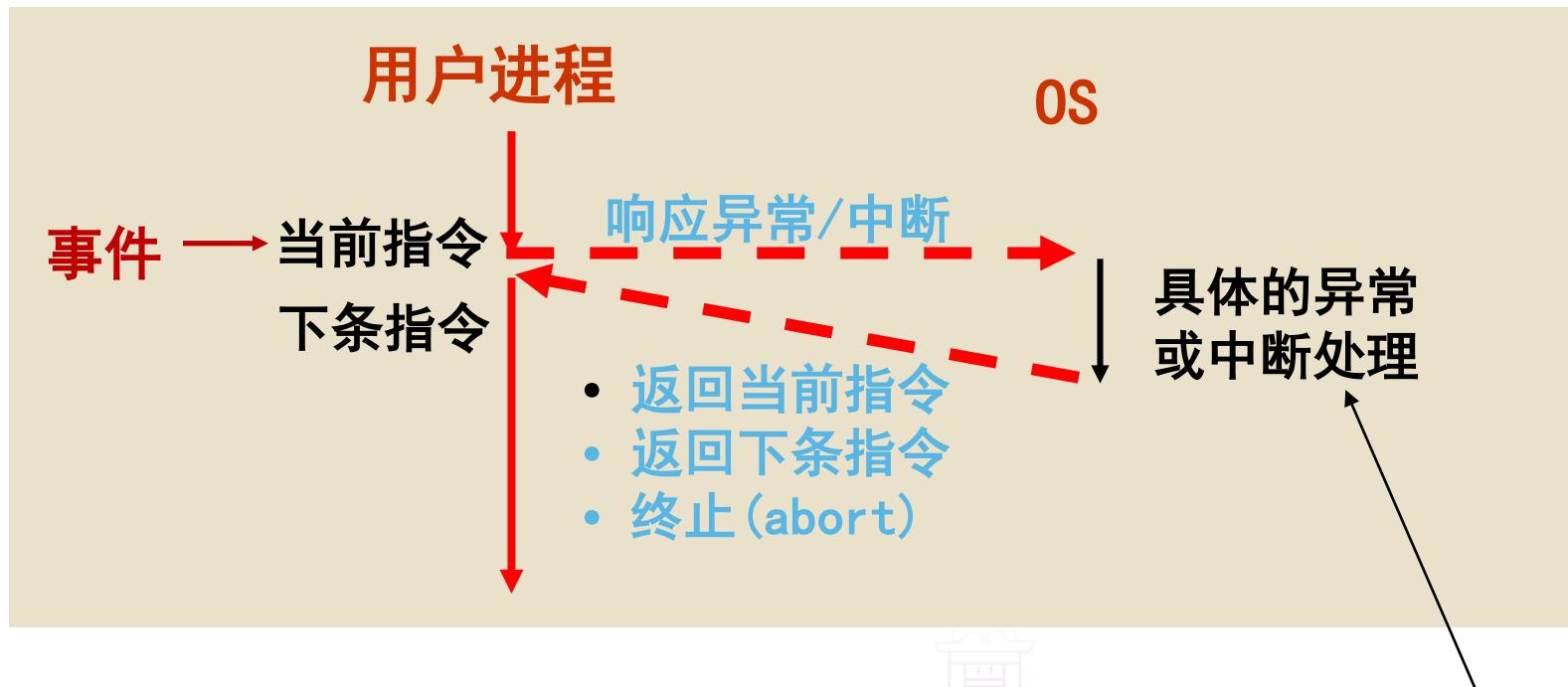
---

- 程序执行过程中CPU会遇到一些特殊情况，使正在执行的程序被“中断”
  - CPU中止原来正在执行的程序，转到处理异常情况或特殊事件的程序去执行，结束后再返回到原被中止的程序处（断点）继续执行。
- 程序执行被“中断”的事件有两类
  - 内部“异常”：在CPU内部发生的意外事件或特殊事件按发生原因分为硬故障中断和程序性中断两类
    - 硬故障中断：如电源掉电、硬件线路故障等
    - 程序性中断：执行某条指令时发生的“例外(Exception)”事件，如溢出、缺页、越界、越权、越级、非法指令、除数为0、堆/栈溢出、访问超时、断点设置、单步、系统调用等
  - 外部“中断”：在CPU外部发生的特殊事件，通过“中断请求”信号向CPU请求处理。如实时钟、控制台、打印机缺纸、外设准备好、采样计时到、DMA传输结束等。



# 异常和中断的处理

发生异常(exception)和中断(interrupt)事件后，系统将进入OS内核态对相应事件进行处理，即改变处理器状态（用户态→内核态）



用户进程的正常控制流中插入了一段内核控制路径

# 第二讲 指令系统实例：RISC-V

1. RISC-V指令系统概述
2. RISC-V指令参考卡和指令格式
3. RISC-V基础整数指令集
4. RISC-V可选的扩展指令集



# RISC-V指令系统概述

## ➤ 设计目标

- 广泛的适应性：从最袖珍的嵌入式微控制器，到最快的高性能计算机
- 支持各种异构处理架构，成为定制加速器的基础
- 稳定的基础指令集架构，并能灵活扩展，且扩展时不影响基础部分

## ➤ 开源理念和设计原则

- 本着“指令集应自由（Instruction Set Want to be Free）”的理念，指令集完全公开，且无需为指令集付费
- 由一个非盈利性质的基金会管理，以保持指令集的稳定性，并加快生态建设
- 与以前的增量ISA不同，遵循“大道至简”的设计哲学，采用模块化设计，既保持基础指令集的稳定，也保证扩展指令集的灵活配置
- 特点：**具有模块化结构、稳定性和可扩展性好，在简洁性、实现成本、功耗、性能和程序代码量等各方面具有显著优势**

## ➤ RISC-V的模块化结构

- 核心：**RV32I + 标准扩展集：RV32M、RV32F、RV32D、RV32A = RV32G**
- 32位架构RV32G = RV32IMAFD，其压缩指令集RV32C（指令长度16位）
- 64位架构RV64G = RV64IMAFD，其压缩指令集RV64C（指令长度16位）
- 向量计算RV32V和RV64V；嵌入式RV32E（RV32I的子集，16个通用寄存器）

# 第二讲 指令系统实例：RISC-V

---

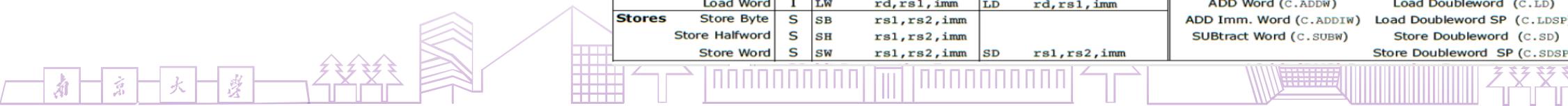
1. RISC-V指令系统概述
2. RISC-V指令参考卡和指令格式
3. RISC-V基础整数指令集
4. RISC-V可选的扩展指令集



# 指令参考卡1

- 核心指令集：基础整数指令集 RV32I 和 RV64I
- 特权指令：陷阱指令对应的返回指令、wfi 等待中断指令、sfence.vma 虚拟存储器的同步操作
- 伪指令举例
- 压缩指令集：RV32C 和 RV64C

Base Integer Instructions: RV32I and RV64I				RV Privileged Instructions				
Category	Name	Fmt	RV32I Base	+RV64I	Category	Name	Fmt	RV mnemonic
<b>Shifts</b>	Shift Left Logical Shift Left Log. Imm. Shift Right Logical Shift Right Log. Imm. Shift Right Arithmetic Shift Right Arith. Imm.	R I R I R I	SLL rd,rs1,rs2 SLLI rd,rs1,shamt SRL rd,rs1,rs2 SRLI rd,rs1,shamt SRA rd,rs1,rs2 SRAI rd,rs1,shamt	SLLW rd,rs1,rs2 SLLIW rd,rs1,shamt SRLW rd,rs1,rs2 SRLIW rd,rs1,shamt SRAW rd,rs1,rs2 SRAIW rd,rs1,shamt		<b>Trap</b> Mach-mode trap return Supervisor-mode trap return	R R	MRET SRET
<b>Arithmetic</b>	ADD ADD Immediate SUBtract Load Upper Imm Add Upper Imm to PC	R I R U U	ADD rd,rs1,rs2 ADDI rd,rs1,imm SUB rd,rs1,rs2 LUI rd,imm AUIPC rd,imm	ADDDW rd,rs1,rs2 ADDIW rd,rs1,imm SUBW rd,rs1,rs2	<b>Interrupt</b> Wait for Interrupt	R	WFI	
<b>Logical</b>	XOR XOR Immediate OR OR Immediate AND AND Immediate	R I R I R I	XOR rd,rs1,rs2 XORI rd,rs1,imm OR rd,rs1,rs2 ORI rd,rs1,imm AND rd,rs1,rs2 ANDI rd,rs1,imm		<b>MMU</b> Virtual Memory FENCE	R	SFENCE.VMA rs1,rs2	
<b>Compare</b>	Set < Set < Immediate Set < Unsigned Set < Imm Unsigned	R I R I	SLT rd,rs1,rs2 SLTI rd,rs1,imm SLTU rd,rs1,rs2 SLTIU rd,rs1,imm		<b>Examples of the 60 RV Pseudoinstructions</b>			
<b>Branches</b>	Branch = Branch ≠ Branch < Branch ≥ Branch < Unsigned Branch ≥ Unsigned	B B B B B	BEQ rs1,rs2,imm BNE rs1,rs2,imm BLT rs1,rs2,imm BGE rs1,rs2,imm BLTU rs1,rs2,imm BGEU rs1,rs2,imm		Branch = 0 (BEQ rs,x0,imm) Jump (uses JAL x0,imm) MoVe (uses ADDI rd,rs,0) RETurn (uses JALR x0,0,ra)	B J R I	BEQZ rs,imm J imm MV rd,rs RET	
<b>Jump &amp; Link</b>	J&L Jump & Link Register	J I	JAL rd,imm JALR rd,rs1,imm		<b>Optional Compressed (16-bit) Instruction Extension: RV32C</b>			
<b>Synch</b>	Synch thread Synch Instr & Data	I I	FENCE FENCE.I		<b>RISC-V equivalent</b>			
<b>Environment</b>	CALL BREAK	I I	ECALL EBREAK		<b>Loads</b>	Load Word	CL	LW rd',rs1',imm*4
					Load Word SP	CI	C.LWSP rd,imm	LW rd,sp,imm*4
					Float Load Word SP	CL	C.FLW rd',rs1',imm*8	FLW rd',rs1',imm*8
					Float Load Word	CI	C.FLWSP rd,imm	FLW rd,sp,imm*8
					Float Load Double	CL	C.FLD rd',rs1',imm*16	FLD rd',rs1',imm*16
					Float Load Double SP	CI	C.FLDSP rd,imm	FLD rd,sp,imm*16
					<b>Stores</b>	Store Word	CS	SW rs1',rs2',imm*4
					Store Word SP	CSS	C.SWSP rs2,imm	SW rs2,sp,imm*4
					Float Store Word	CS	C.FSW rs1',rs2',imm*8	FSW rs1',rs2',imm*8
					Float Store Word SP	CSS	C.FSWSP rs2,imm	FSW rs2,sp,imm*8
					Float Store Double	CS	C.FSD rs1',rs2',imm	FSD rs1',rs2',imm*16
					Float Store Double SP	CSS	C.FSDSP rs2,imm	FSD rs2,sp,imm*16
					<b>Arithmetic</b>	ADD	CR	ADD rd,rs1
					ADD Immediate	CI	C.ADDI rd,imm	ADDI rd,rd,imm
					ADD SP Imm * 16	CI	C.ADDI16SP x0,imm	ADDI sp,sp,imm*16
					ADD SP Imm * 4	CIW	C.ADDI4SPN rd',imm	ADDI rd',sp,imm*4
					SUB	CR	C.SUB rd,rs1	SUB rd,rd,rs1
					AND	CR	C.AND rd,rs1	AND rd,rd,rs1
					AND Immediate	CI	C.ANDI rd,imm	ANDI rd,rd,imm
					OR	CR	C.OR rd,rs1	OR rd,rd,rs1
					eXclusive OR	CR	C.XOR rd,rs1	AND rd,rd,rs1
					MoVe	CR	C.MV rd,rs1	ADD rd,rs1,x0
					Load Immediate	CI	C.LI rd,imm	ADDI rd,x0,imm
					Load Upper Imm	CI	C.LUI rd,imm	LUI rd,imm
					<b>Shifts</b>	Shift Left Imm	CI	SLLI rd,imm
					Shift Right Ari. Imm.	CI	C.SRAI rd,imm	SRAI rd,rd,imm
					Shift Right Log. Imm.	CI	C.SRLI rd,imm	SRLI rd,rd,imm
					<b>Branches</b>	Branch=0	CB	BEQ rs1',x0,imm
					Branch≠0	CB	C.BNEZ rs1',imm	BNE rs1',x0,imm
					<b>Jump</b>	Jump	CJ	JAL x0,imm
					Jump Register	CR	C.JR rd,rs1	JALR x0,rs1,0
					<b>Jump &amp; Link</b>	J&L	C.JAL imm	JAL ra,imm
					Jump & Link Register	CR	C.JALR rs1	JALR ra,rs1,0
					<b>System Env. BREAK</b>	CI	C.EBREAK	EBREAK
					<b>Optional Compressed Extension: RV64C</b>			
					All RV32C (except C.JAL, 4 word loads, 4 word stores) plus:			
					ADD Word (C.ADDW)		Load Doubleword (C.LD)	
					ADD Imm. Word (C.ADDIW)		Load Doubleword SP (C.LDSP)	
					SUBtract Word (C.SUBW)		Store Doubleword (C.SD)	
							Store Doubleword SP (C.SDSP)	



# 指令参考卡2

## 扩展指令集

乘除运算指令集RVM、

原子操作指令RVA、

浮点运算指令集RVF和RVD、

向量操作指令RVV

## 通用寄存器的调用约定

32个定点通用寄存器x0~x31；

32个浮点寄存器f0~f31；通用寄存器x0中恒0；x1中返回地址；x2、x3和x4分别为栈指针、全局指针和线程指针

Optional Multiply-Divide Instruction Extension: RVM						Optional Vector Extension: RVV		
Category	Name	Fmt	RV32M (Multiply-Divide)	+RV64M		Name	Fmt	RV32V/R64V
<b>Multiply</b>	MULTiply	R	MUL rd,rs1,rs2		MULW rd,rs1,rs2	SET Vector Len.	R	SETVL rd,rs1
	MULTiply High	R	MULH rd,rs1,rs2			MULTiply High	R	VMULH rd,rs1,rs2
	MULTiply High Sign/Uns	R	MULHSU rd,rs1,rs2			REMainder	R	VREM rd,rs1,rs2
	MULTiply High Uns	R	MULHU rd,rs1,rs2			Shift Left Log.	R	VSLL rd,rs1,rs2
<b>Divide</b>	DIVide	R	DIV rd,rs1,rs2	DIVW	rd,rs1,rs2	Shift Right Log.	R	VSRL rd,rs1,rs2
	DIVide Unsigned	R	DIVU rd,rs1,rs2	rd,rs1,rs2	rd,rs1,rs2	Shift R. Arith.	R	VSRA rd,rs1,rs2
<b>Remainder</b>	REMAinder	R	REM rd,rs1,rs2			LoaD	I	VLD rd,rs1,imm
	REMAinder Unsigned	R	REMU rd,rs1,rs2	REMW	rd,rs1,rs2	LoaD Strided	R	VLDS rd,rs1,rs2
Optional Atomic Instruction Extension: RVA						LoaD indeXed	R	VLDX rd,rs1,rs2
Category	Name	Fmt	RV32A (Atomic)	+RV64A		STore	S	VST rd,rs1,imm
Load	Load Reserved	R	LR.W rd,rs1	LR.D	rd,rs1	STore Strided	R	VSTS rd,rs1,rs2
Store	Store Conditional	R	SC.W rd,rs1,rs2	SC.D	rd,rs1,rs2	STore indeXed	R	VSTX rd,rs1,rs2
Swap	SWAP	R	AMOSWAP.W rd,rs1,rs2	AMOSWAP.D	rd,rs1,rs2	AMO SWAP	R	AMOSWAP rd,rs1,rs2
Add	ADD	R	AMOADD.W rd,rs1,rs2	AMOADD.D	rd,rs1,rs2	AMO ADD	R	AMOADD rd,rs1,rs2
<b>Logical</b>	XOR	R	AMOXOR.W rd,rs1,rs2	AMOXOR.D	rd,rs1,rs2	AMO XOR	R	AMOXOR rd,rs1,rs2
	AND	R	AMOAND.W rd,rs1,rs2	AMOAND.D	rd,rs1,rs2	AMO AND	R	AMOAND rd,rs1,rs2
	OR	R	AMOOR.W rd,rs1,rs2	AMOOR.D	rd,rs1,rs2	AMO OR	R	AMOOR rd,rs1,rs2
<b>Min/Max</b>	MINimum	R	AMOMIN.W rd,rs1,rs2	AMOMIN.D	rd,rs1,rs2	AMO MINimum	R	AMOMIN rd,rs1,rs2
	MAXimum	R	AMOMAX.W rd,rs1,rs2	AMOMAX.D	rd,rs1,rs2	AMO MAXimum	R	AMOMAX rd,rs1,rs2
	MINimum Unsigned	R	AMOMINU.W rd,rs1,rs2	AMOMINU.D	rd,rs1,rs2	Predicate =	R	VPEQ rd,rs1,rs2
	MAXimum Unsigned	R	AMOMAXU.W rd,rs1,rs2	AMOMAXU.D	rd,rs1,rs2	Predicate ≠	R	VPNE rd,rs1,rs2
Two Optional Floating-Point Instruction Extensions: RVF & RVD						Predicate <	R	VPLT rd,rs1,rs2
Category	Name	Fmt	RV32{F D} (SP,DP Fl. Pt.)	+RV64{F D}		Predicate ≥	R	VPGE rd,rs1,rs2
<b>Move</b>	Move from Integer	R	FMV.W.X rd,rs1	FMV.D.X	rd,rs1	Predicate AND	R	VPAND rd,rs1,rs2
	Move to Integer	R	FMV.X.W rd,rs1	FMV.X.D	rd,rs1	Pred. AND NOT	R	VPANDN rd,rs1,rs2
<b>Convert</b>	ConVerT from Int	R	FCVT.{S D}.W rd,rs1	FCVT.{S D}.L	rd,rs1	Predicate OR	R	VPOR rd,rs1,rs2
	ConVerT from Int Unsigned	R	FCVT.{S D}.WU rd,rs1	FCVT.{S D}.LU	rd,rs1	Predicate XOR	R	VPXOR rd,rs1,rs2
	ConVerT to Int	R	FCVT.W.{S D} rd,rs1	FCVT.L.{S D}	rd,rs1	Predicate NOT	R	VPNOT rd,rs1
	ConVerT to Int Unsigned	R	FCVT.WU.{S D} rd,rs1	FCVT.LU.{S D}	rd,rs1	Pred. SWAP	R	VPSWAP rd,rs1
	Load	I	FLOAT{W,D} rd,rs1,imm	Calling Convention		MOVE	R	VMOV rd,rs1
Store	Store	S	FLOAT{W,D} rs1,rs2,imm	Register	ABI Name	ConVerT	R	VCVT rd,rs1
<b>Arithmetic</b>	ADD	R	FADD.{S D} rd,rs1,rs2	x0	zero	ADD	R	VADD rd,rs1,rs2
	SUBtract	R	FSUB.{S D} rd,rs1,rs2	x1	ra	SUBtract	R	VSUB rd,rs1,rs2
	MULTiply	R	FMUL.{S D} rd,rs1,rs2	x2	sp	MULTiply	R	VMUL rd,rs1,rs2
	DIVide	R	FDIV.{S D} rd,rs1,rs2	x3	gp	DIVide	R	VDIV rd,rs1,rs2
	SQuare Root	R	FSQRT.{S D} rd,rs1	x4	tp	SQuare Root	R	VSQRT rd,rs1,rs2
<b>Mul-Add</b>	Multiply-ADD	R	FMADD.{S D} rd,rs1,rs2,rs3	x5-7	t0-2	Multiply-ADD	R	VFMADD rd,rs1,rs2,rs3
	Multiply-SUBtract	R	FMSUB.{S D} rd,rs1,rs2,rs3	x8	s0/fp	Multiply-SUB	R	VFMSUB rd,rs1,rs2,rs3
	Negative Multiply-SUBtract	R	FNMSUB.{S D} rd,rs1,rs2,rs3	x9	s1	Neg. Mul.-SUB	R	VFNMSUB rd,rs1,rs2,rs3
	Negative Multiply-ADD	R	FNMADD.{S D} rd,rs1,rs2,rs3	x10-11	a0-1	Neg. Mul.-ADD	R	VFNMADD rd,rs1,rs2,rs3
<b>Sign Inject</b>	SiGN source	R	FSGNJ.{S D} rd,rs1,rs2	x12-17	a2-7	SiGN inJECT	R	VSGNJ rd,rs1,rs2
	Negative SiGN source	R	FSGNINJ.{S D} rd,rs1,rs2	x18-27	s2-11	Neg SiGN inJECT	R	VSGNINJ rd,rs1,rs2
	Xor SiGN source	R	FSGNJKX.{S D} rd,rs1,rs2	x28-31	t3-t6	Xor SiGN inJECT	R	VSGNJK rd,rs1,rs2
<b>Min/Max</b>	MINimum	R	FMIN.{S D} rd,rs1,rs2	f0-7	ft0-7	MINimum	R	VMIN rd,rs1,rs2
	MAXimum	R	FMAX.{S D} rd,rs1,rs2	f8-9	fs0-1	MAXimum	R	VMAX rd,rs1,rs2
<b>Compare</b>	compare Float =	R	FEQ.{S D} rd,rs1,rs2	f10-11	fa0-1	XOR	R	VXOR rd,rs1,rs2
	compare Float <	R	FLT.{S D} rd,rs1,rs2	f12-17	fa2-7	OR	R	VOR rd,rs1,rs2
	compare Float ≤	R	FLE.{S D} rd,rs1,rs2	f18-27	fs2-11	AND	R	VAND rd,rs1,rs2
<b>Categorize</b>	CLASSify type	R	FCLASS.{S D} rd,rs1	f28-31	ft8-11	CLASS	R	VCLASS rd,rs1
	Configure	R	FRCSR rd	zero		SET Data Conf.	R	VSETDCFG rd,rs1
<b>Configure</b>	Read Status	R	FRRM rd	ra		EXTRACT	R	VEXTRACT rd,rs1,rs2
	Read Rounding Mode	R	FRFLRM rd	sp		MERGE	R	VMERGE rd,rs1,rs2
	Read Flags	R	FRFLAGS rd	gp		SELECT	R	VSELECT rd,rs1,rs2
	Swap Status Reg	R	FCSR rd,rs1	tp				
	Swap Rounding Mode	R	FSRM rd,rs1	t0-0,ft0-7	Temporaries			
<b>Swap</b>	Swap Flags	R	FSFLAGS rd,rs1	s0-11,fs0-11	Saved registers			
	Swap Rounding Mode Imm	I	FSRMI rd,imm	a0-7,fa0-7	Function args			
	Swap Flags Imm	I	FSFLAGSI rd,imm					



# RV32I寄存器的功能定义和两种汇编表示

寄存器	ABI 名	功能描述	被调用过程保存?
x0	zero	硬编码 0	—
x1	ra	返回地址	否
x2	sp	栈指针	是
x3	gp	全局指针	—
x4	tp	线程指针	—
x5	t0	临时寄存器	否
x6~x7	t1~t2	临时寄存器	否
x8	s0/fp	保存寄存器/帧指针	是
x9	s1	保存寄存器	是
x10~x11	a0~a1	过程参数/返回值	否
x12~x17	a2~a7	过程参数	否
x18~x27	s2~s11	保存寄存器	是
x28~x31	t3~t6	临时寄存器	否

Registers are referenced either by number—x0, ... x31, or by name —zero, ra, s1... t0.

# 32位RISC-V指令格式

- 共有6种指令格式

R-型为寄存器操作数指令

I-型为短立即数或装入（Load）指令

S-型为存储（Store）指令

B-型为条件跳转指令

U-型为长立即数操作指令

J-型为无条件跳转指令

➤ **opcode**: 操作码字段

➤ **rd、rs1和rs2**: 通用寄存器编号

➤ **imm**: 立即数，其位数在括号[ ]中表示

➤ **funct3**和**funct7**: 分别表示3位功能码和7位功能码，和opcode字段一起定义指令的操作功能

	31	27	26	25	24	20	19	15	14	12	11	7	6	0
<b>R</b>	funct7				rs2		rs1	funct3		rd		opcode		
<b>I</b>		imm[11:0]					rs1	funct3		rd		opcode		
<b>S</b>	imm[11:5]				rs2		rs1	funct3		imm[4:0]		opcode		
<b>B</b>	imm[12 10:5]				rs2		rs1	funct3		imm[4:1 11]		opcode		
<b>U</b>					imm[31:12]					rd		opcode		
<b>J</b>					imm[20 10:1 11 19:12]					rd		opcode		



# 16位RISC-V压缩指令格式

- 共有8种指令格式。与32位指令相比，16位指令中的一部分寄存器编号还是占5位。指令变短了，但还是32位架构，处理的还是32位数据，还是有32个通用寄存器。
- 为了缩短指令长度，操作码op、功能码funct、立即数imm和另一部分寄存器编号的位数都减少了。
- 每条16位指令都有功能完全相同的32位指令，在执行时由硬件先转换为32位指令再执行。  
目的是：缩短程序代码量，用少量时间换空间！

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<b>CR</b>				funct4				rd/rs1				rs2			op	
<b>CI</b>				funct3	imm			rd/rs1				imm			op	
<b>CSS</b>				funct3				imm				rs2			op	
<b>CIW</b>				funct3				imm				rd'			op	
<b>CL</b>				funct3								rd'			op	
<b>CS</b>				funct3								imm			op	
<b>CB</b>				funct3								rs2'			op	
<b>CJ</b>				funct3				offset				offset			op	
									jump target						op	

# 第二讲 指令系统实例：RISC-V

---

1. RISC-V指令系统概述
2. RISC-V指令参考卡和指令格式
3. RISC-V基础整数指令集
4. RISC-V可选的扩展指令集



# RISC-V基础整数指令集 (RV32I)

- 包含：

- 移位 (Shifts)
- 算术运算 (Arithmetic)
- 逻辑运算 (Logical)
- 比较 (Compare)
- 分支 (Branch)
- 跳转链接 (Jump & Link)
- 同步 (Sync)
- 环境 (Environment)
- 控制状态寄存器 (Control Status Register)
- 取数 (Load)
- 存数 (Store)

整数运算类指令

控制转移类指令

系统控制类指令

存储访问类指令

**RTL规定：**

**R[r]**：通用寄存器r的内容

**M[addr]**：存储单元addr的内容

**M[R[r]]**：寄存器r的内容所指存储单元的内容

**PC**：PC的内容

**M[PC]**：PC所指存储单元的内容

**SEXT[imm]**：对imm进行符号扩展

**ZEXT[imm]**：对imm进行零扩展

传送方向用  $\leftarrow$  表示，即传送源在右，传送目的在左

# RISC-V基础整数指令集 (RV32I)

## 整数运算类指令

31	25 24	20 19	15 14	12 11	7 6	0
	imm[31:12]			rd	0110111	U lui
	imm[31:12]			rd	0010111	U auipc
	imm[11:0]	rs1	000	rd	0010011	I addi
	imm[11:0]	rs1	010	rd	0010011	I slti
	imm[11:0]	rs1	011	rd	0010011	I sltiu
	imm[11:0]	rs1	100	rd	0010011	I xor
	imm[11:0]	rs1	110	rd	0010011	I ori
	imm[11:0]	rs1	111	rd	0010011	I andi
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	Rslt
0000000	rs2	rs1	011	rd	0110011	R sltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and

# RISC-V基础整数指令集 (RV32I)

U型指令共2条

31	25 24	20 19	15 14	12 11	7 6	0
		imm[31:12]		rd	0110111	U lui
		imm[31:12]		rd	0010111	U auipc

**lui rd, imm20:** 将立即数imm20存到rd寄存器高20位，低12位为0。该指令和“addi rd, rs1, imm12”结合，可以实现对一个32位变量赋初值。

**举例：**请给出C语句“int x=-8191;”对应的RISC-V机器级代码。

**解：**C语句“int x=-8191;”对应的RISC-V机器指令和汇编指令为：

1111 1111 1111 1111 1110 00101 0110111 lui x5, 1048574 #R[x5]  $\leftarrow$  FFFF E000H

0000 0000 0001 00101 000 00101 0010011 addi x5, x5, 1 #R[x5]  $\leftarrow$  R[x5] + SEXT[001H]

-8191的机器数为：1111 1111 1111 1111 1110 0000 0000 0001

SEXT表示符号扩展

**auipc rd, imm20:** 将立即数imm20加到PC的高20位上，结果存rd。可用指令“auipc x10, 0”获取当前PC的内容，存入寄存器x10中。

# RISC-V基础整数指令集 (RV32I)

I型指令共9条，其中三条为用立即数指定所移位数的移位指令

31	25 24	20 19	15 14	12 11	7 6	0
	imm[11:0]	rs1	000	rd	0010011	I addi
	imm[11:0]	rs1	010	rd	0010011	I slti
	imm[11:0]	rs1	011	rd	0010011	I sltiu
	imm[11:0]	rs1	100	rd	0010011	I xor
	imm[11:0]	rs1	110	rd	0010011	I ori
	imm[11:0]	rs1	111	rd	0010011	I and
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai

**操作码opcode：**都是0010011，其功能由funct3指定，

- 当funct3=101时，再由高7位区分是算术右移 (srai) 还是逻辑右移 (srli)。
- shamt**：指出移位位数，因为最多移31位，故用5位即可。
- imm[11:0]**：12位立即数，**符号扩展为32位**，作为第2个源操作数，和R[rs1]（寄存器rs1中的内容）进行运算，结果存rd。

# RISC-V基础整数指令集 (RV32I)

R型指令  
共10条

31	25 24	20 19	15 14	12 11	7 6	0	
0000000	rs2	rs1	000	rd	0110011		R add
0100000	rs2	rs1	000	rd	0110011		R sub
0000000	rs2	rs1	001	rd	0110011		R sll
0000000	rs2	rs1	010	rd	0110011		R slt
0000000	rs2	rs1	011	rd	0110011		R sltu
0000000	rs2	rs1	100	rd	0110011		R xor
0000000	rs2	rs1	101	rd	0110011		R srl
0100000	rs2	rs1	101	rd	0110011		R sra
0000000	rs2	rs1	110	rd	0110011		R or
0000000	rs2	rs1	111	rd	0110011		R and

**操作码opcode**: 都是0110011，其功能由funct3指定，而当funct3=000、101时，再由funct7区分是加（add）还是减（sub）、逻辑右移（srl）还是算术右移（sra）。

**rs1、rs2、rd**: 5位通用寄存器编号，共32个；两个源操作数分别在rs1和rs2寄存器中，结果存rd。

**sll**: 逻辑左移指令，无算术左移指令。因逻辑左移和算术左移结果完全相同！

# RISC-V基础整数指令集 (RV32I)

4条比较指令：带符号小于 (slt、slti) 、无符号小于 (sltu、sltiu)

例如，“sltiu rd, rs1, imm12” 功能为：将rs1内容与imm12符号扩展结果按无符号整数比较，若小于，则1存入rd中；否则，0存入rd中。

举例：假定变量x、y和z都是long long型，占64位，x的高、低32位分别存放在寄存器x13、x12中；y的高、低32位分别存放在寄存器x15、x14中；z的高、低32位分别存放在寄存器x11、x10中，请写出C语句“z=x+y;”对应的32位字长RISC-V机器级代码。

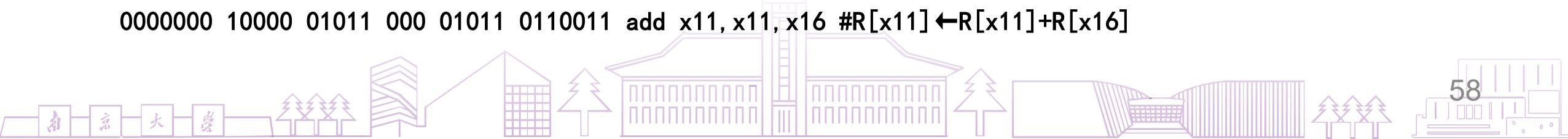
解：可通过sltu指令将低32位的进位加入到高32位中。

0000000 01110 01100 000 01010 0110011 add x10, x12, x14 # $R[x10] \leftarrow R[x12] + R[x14]$

0000000 01100 01010 011 01011 0110011 sltu x11, x10, x12 #若 $R[x10] < R[x12]$ ，则 $R[x11] \leftarrow 1$ （若和比加数小，则一定有进位）

0000000 01111 01101 000 10000 0110011 add x16, x13, x15 # $R[x16] \leftarrow R[x13] + R[x15]$

0000000 10000 01011 000 01011 0110011 add x11, x11, x16 # $R[x11] \leftarrow R[x11] + R[x16]$



# RISC-V基础整数指令集 (RV32I)

控制转移  
类指令

31	25 24	20 19	15 14	12 11	7 6	0
imm[20 10:1 11 19:12]					rd	1101111
imm[11:0]		rs1	000	rd	1100111	J jal
imm[12 10:5]	rs2	rs1	000	imm[4:1 11]	1100011	I jalr
imm[12 10:5]	rs2	rs1	001	imm[4:1 11]	1100011	B beq
imm[12 10:5]	rs2	rs1	100	imm[4:1 11]	1100011	B bne
imm[12 10:5]	rs2	rs1	101	imm[4:1 11]	1100011	B blt
imm[12 10:5]	rs2	rs1	110	imm[4:1 11]	1100011	B bge
imm[12 10:5]	rs2	rs1	111	imm[4:1 11]	1100011	B bltu
imm[12 10:5]	rs2	rs1	111	imm[4:1 11]	1100011	B bgeu

**J型**: jal 功能为:  $R[rd] \leftarrow PC+4$ ;  $PC \leftarrow PC+SEXT[imm[20:1]\ll1]$

rd指定为x1时可实现**过程调用**; 指定为x0时, 可实现**无条件跳转**。

**I型**: jalr功能为:  $R[rd] \leftarrow PC+4$ ;  $PC \leftarrow R[rs1]+SEXT[imm[12]]$

指令“jalr x0, x1, 0”可实现过程调用的返回。

**B型**: 皆为分支指令, 其中, bltu、bgeu分别为**无符号数比较小于、大于等于转移**。转移目标地址= $PC+SEXT[imm[12:1]\ll1]$

$\ll1$ : 指令地址总是2的倍数 (RV32G、RV32C指令分别为4、2字节长)

# RISC-V基础整数指令集 (RV32I)

**举例：**若int型变量x、y、z分别存放在寄存器x5、x6、x7中，写出C语句“`z=x+y;`”对应的RISC-V机器级代码，要求检测是否溢出。

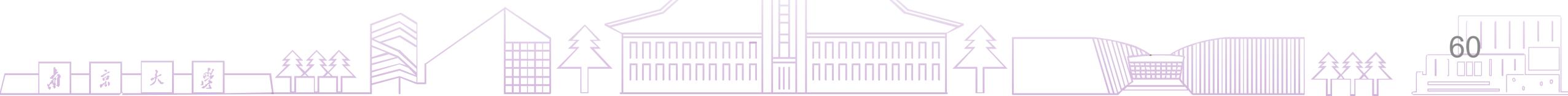
解：当x、y为int类型时，若“ $y < 0$ 且 $x+y \geq x$ ”或者“ $y \geq 0$ 且 $x+y < x$ ”，则 $x+y$ 溢出。可通过slti指令对y与0进行比较。

0000000 00110 00101 000 00111 0110011 add x7, x5, x6	# $R[x7] \leftarrow R[x5] + R[x6]$
0000 0000 0000 00110 010 11100 0010011 slti x28, x6, 0	#若 $R[x6] < 0$ ，则 $R[x28] \leftarrow 1$
0000000 00101 00111 010 11101 0110011 slt x29, x7, x5	#若 $R[x7] < R[x5]$ 则 $R[x29] \leftarrow 1$
0000010 11101 11100 001 10000 1100011 bne x28, x29, overflow	#若 $R[x28] \neq R[x29]$ #则转溢出处理

.....

overflow:	imm[12 10:5]	rs2	rsl	001	imm[4:1 11]	1100011
-----------	--------------	-----	-----	-----	-------------	---------

假定标号为overflow的指令与“`bne x28, x29, overflow`”之间相距20条指令，每条指令4字节，则“`bne x28, x29, overflow`”指令中的偏移量应为80，因此，指令中的立即数为 $40=0000\ 0010\ 1000B$ ，按照B-型格式，该指令的机器码为“`0000010 11101 11100 001 10000 1100011`”。



# RISC-V基础整数指令集 (RV32I)

存储  
访问  
指令

31	25 24	20 19	15 14	12 11	7 6	0
	imm[11:0]	rsl	000	rd	0000011	I lb
	imm[11:0]	rsl	001	rd	0000011	I lh
	imm[11:0]	rsl	010	rd	0000011	I lw
	imm[11:0]	rsl	100	rd	0000011	I lbu
	imm[11:0]	rsl	101	rd	0000011	I lhu
imm[11:5]	rs2	rsl	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rsl	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rsl	010	imm[4:0]	0100011	S sw

**I型**: 5条取数 (Load) 指令。功能:  $R[rd] \leftarrow M[R[rs1]+SEXT[imm[12]]]$ 。**Ibu**、**Ihu**: 分别为无符号字节、半字取，取出数据按0扩展为32位，装入rd

**S型**: 3条存数 (Store) 指令。功能:  $M[R[rs1]+SEXT[imm[12]]] \leftarrow R[rs2]$ 。**sb**、**sh**: 分别将rs2寄存器中低8、16位写入存储单元中。

**汇编形式**: 存储地址可写成 imm12(rs1)。

# RISC-V基础整数指令集 (RV32I)

系统控制类指令

	31	25 24	20 19	15 14	12 11	7 6	0	
	0000	pred	succ	00000	000	00000	0001111	I fence
	0000	0000	0000	00000	001	00000	0001111	I fence.i
	0000000000000000		00000	00	00000	1110011		I ecall
	0000000000000000		00000	000	00000	1110011		I ebreak
	csr		rs1	001	rd	1110011		I csrrw
	csr		rs1	010	rd	1110011		I csrrs
	csr		rs1	011	rd	1110011		I csrrc
	csr		zimm	101	rd	1110011		I csrrwi
	csr		zimm	110	rd	1110011		I csrrsi
	csr		zimm	111	rd	1110011		I csrrci

**fence**: RISC-V架构在不同硬件线程之间使用宽松一致性模型，fence和fence.i两条屏障指令，用于保证一定的存储访问顺序。

**ecall和ebreak**: 陷阱(trap)指令，也称自陷指令，主要用于从用户程序陷入到操作系统内核(ecall)或调试环境(ebreak)执行，因此也称为环境(Environment)类指令。

**csrxxx**: 6条csr指令用于设置和读取相应的**控制状态寄存器(CSR)**。

# 第二讲 指令系统实例：RISC-V

---

1. RISC-V指令系统概述
2. RISC-V指令参考卡和指令格式
3. RISC-V基础整数指令集
4. RISC-V可选的扩展指令集



# RISC-V可选的控制指令集

- ◆ 标准扩展指令集
  - RV32I基础指令集之上，可标准扩展RV32M、RV32F/D、RV32A，以形成32位架构合集**RV32IMAFD**，也称为**RV32G**
  - RV32G基础上，对每个指令集进行调整和添加，可形成64位架构**RV64G**，原先在RV32G中处理的数据将调整为64位。但为了支持32位数据操作，每个64位架构指令集中都会添加少量32位数据处理指令。
- ◆ RISC-V扩展集包括
  - 针对**64位架构**需要，在47条RV32I指令基础上，增加12条整数指令（+RV64I），包括6条32位移位指令、3条32位加减运算指令、两条64位装入（Load）指令和1条64位存储（Store）指令，故RV64I共**59条指令**。
  - 针对乘除运算需要，提供了32位架构乘除运算指令集RV32M中的**8条指令**，并在此基础上增加了4条RV64M专用指令（+RV64M）
  - 针对浮点数运算的需要，提供了32位架构的单精度浮点处理指令集RV32F和双精度浮点处理指令集RV32D，并在此基础上分别增加了RV64F和RV64D专用指令集（+RV64F）和（+RV64D）。
  - 针对事务处理和操作原子性的需要，提供了32位架构原子操作指令集RV32A以及RV64A专用指令集（+RV64A）。
  - 向量处理指令集RVV、未来可选扩展指令集RVB、RVE、RVH、.....



# RISC-V中整数的乘、除运算处理

- 乘法指令：`mul`, `mulh`, `mulhu`, `mulhsu`
  - `mul rd, rs1, rs2`: 将低32位乘积存入结果寄存器rd
  - `mulh`、`mulhu`: 将两个乘数同时按带符号整数（`mulh`）或同时按无符号整数（`mulhu`）相乘，高32位乘积存入rd中
  - `mulhsu`: 将两个乘数分别作为带符号整数和无符号整数相乘后得到的高32位乘积存入rd中
  - 得到64位乘积需要两条连续的指令，其中一定有一条是`mul`指令，实际执行时只有一条指令
  - 两种乘法指令都不检测溢出，而是直接把结果写入结果寄存器。由软件根据结果寄存器的值自行判断和处理溢出
- 问题：如何判断溢出？
- 除法指令：`div`, `divu`, `rem`, `remu`
  - `div / rem`: 按带符号整数做除法，得到商 / 余数
  - `divu / remu`: 按无符号整数做除法，得到商 / 余数
  - RISC-V指令不检测和发出异常，而是由系统软件自行处理

# 整数的乘运算

- 硬件可保留 $2n$ 位乘积，故有些指令的乘积为 $2n$ 位，可供软件使用
- 如果程序不采用防止溢出的措施，且编译器也不生成用于溢出处理的代码，就会发生一些由于整数溢出而带来的问题。
- 指令：**分无符号数乘指令、带符号整数乘指令
- 乘法指令的操作数长度为 $n$ ，而乘积长度为 $2n$ ，例如：
  - IA-32中，若指令只给出一个操作数SRC，则另一个源操作数隐含在累加器AL/AX/EAX中，将SRC和累加器内容相乘，结果存放在AX（16位时）或DX-AX（32位时）或EDX-EAX（64位时）中。
  - MIPS中，mult会将两个32位带符号整数相乘，得到的64位乘积置于两个32位内部寄存器Hi和Lo中，因此，可以根据Hi寄存器中的每一位是否等于Lo寄存器中的第一位来进行溢出判断。
  - RISC-V中，用“mul rd, rs1, rs2”获得低32位乘积并存入结果寄存器rd中；mulh、mulhu指令分别将两个乘数同时按带符号整数、同时按无符号整数相乘后，得到的高32位乘积存入rd中

乘法指令可生成溢出标志，编译器也可使用 $2n$ 位乘积来判断是否溢出！