

# 计算机组成原理

文杰

计算机科学与技术学院

wenjie@hit.edu.cn

个人主页: <http://faculty.hitsz.edu.cn/wenjie>

# 第三章 RISC-V汇编及其指令系统

---

- **RISC-V概述**
- RISC-V汇编语言
- RISC-V指令表示
- 案例分析



# 第三章 RISC-V汇编及其指令系统

---

- RISC-V概述

- 指令系统的基本概念
- 主流指令集及发展方向
- RISC-V指令集



# 指令系统基本概念

- 机器指令（指令）
  - 计算机能直接识别、执行的某种操作命令，它是一堆二进制代码
- 指令系统（指令集，**IS: Instruction Set**）
  - 一台计算机中所有机器指令的集合
- 指令集架构（**ISA: Instruction Set Architecture**）
  - 简称“架构”，也可称为：处理器架构、指令集体系结构
  - 包含了程序员正确编写二进制机器语言程序所需的全部信息。
  - 例如：如何使用硬件、指令格式，操作种类、操作数所能存放的寄存器组 and 结构，包括每个寄存器名称、编号、长度和用途等。
- 系列机
  - 基本指令系统相同、基本系统结构相同的计算机。
  - 软件兼容。给定一个**ISA**，可以有不同的处理器硬件实现方案；例如AMD/Intel CPU 都是X86-64指令集。ARM的ISA也有不同的实现方式
  - IBM 360 是**第一个**将ISA与其实现分离的系列机

# 指令集架构

## 功能

数据类型

存储模型

软件可见的处理器状态

- 通用寄存器、PC
- 处理器状态

指令集

- 指令类型与编码
- 寻址模式
- 数据结构

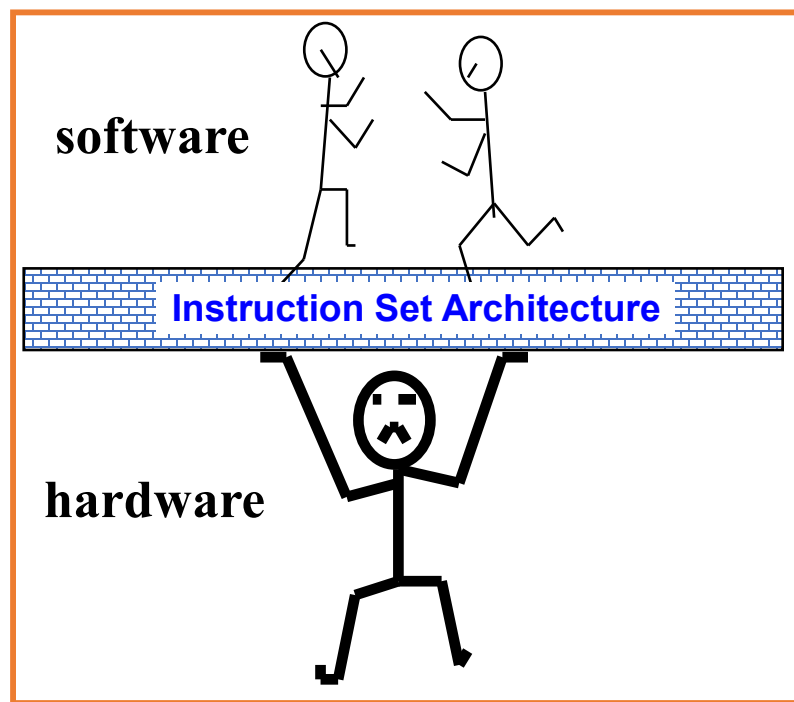
系统模型

- 状态、特权级别
- 中断和异常

外部接口

- 输入/输出接口
- 管理

**ISA: 抽象层，软件子系统与硬件子系统的桥梁和接口**



## 特性

成本和资源低

简洁性：指令规整简洁；存储器访问/运算指令、子程序调用简洁

架构和具体实现分离

- 可持续多代，向后兼容

可扩展空间

- 用户可根据应用领域灵活扩展(desktops, servers, embedded applications)

易于编程/编译/链接

- 为高层软件的设计与开发提供便利

性能好：方便底层硬件子系统高效实现

# 指令集架构(ISA)位宽

- **ISA位宽**：指通用寄存器的宽度，决定了寻址范围的大小、数据运算能力的强弱
- **注意**：ISA位宽和指令编码长度无任何关系，不要误以为64位架构的指令长度是64位，即便在64位架构中，也大量存在16位编码的指令，且基本上很少出现过64位长的指令编码。

不考虑实际成本和实现技术的前提下

ISA位宽

越大越好

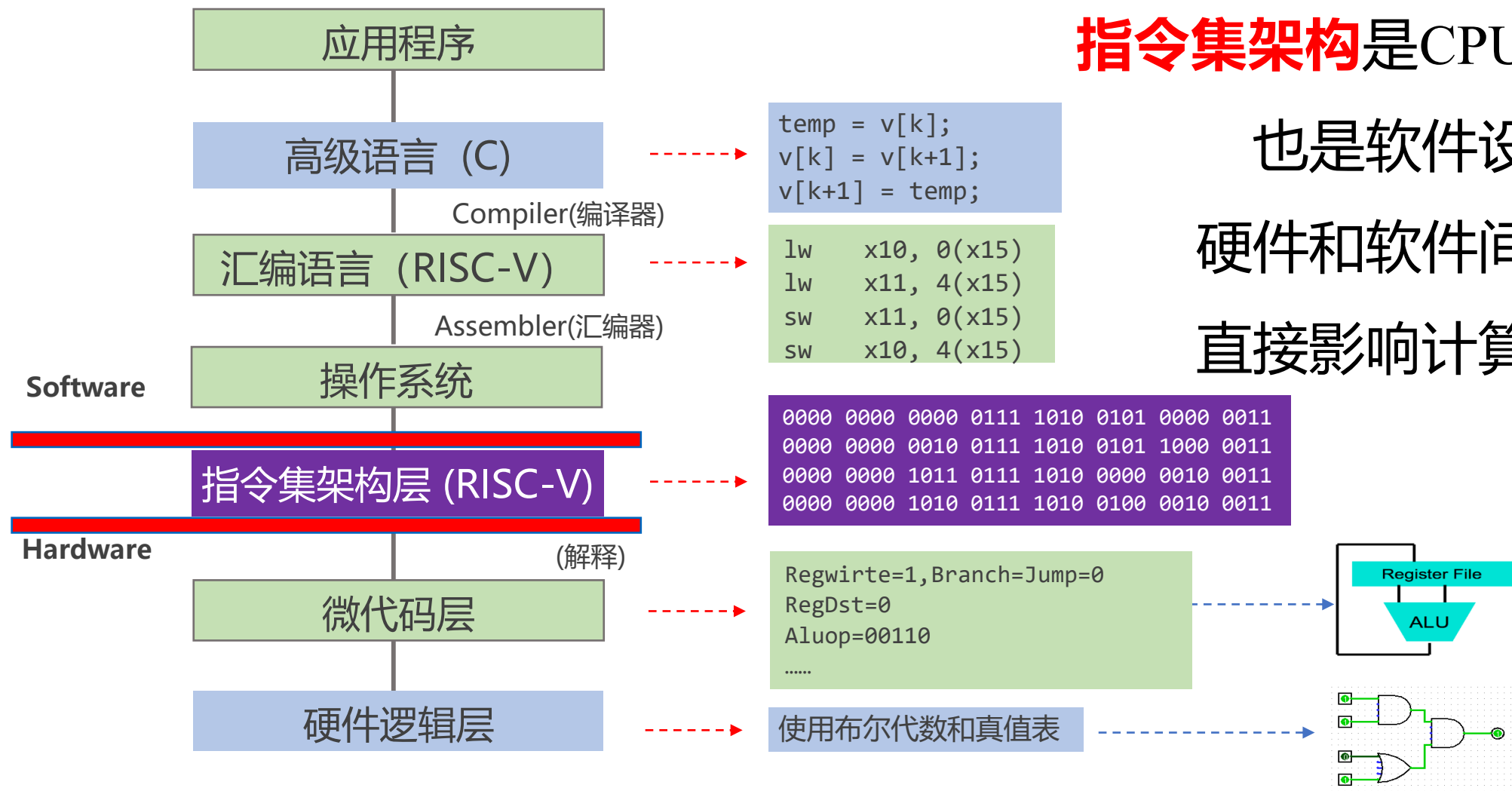
可以带来更大的寻址范围和更强的运算能力

指令编码长度

越短越好

节省代码存储空间

# 计算机指令系统层次



**指令集架构**是CPU设计的依据、  
也是软件设计的基础、  
硬件和软件间的分界面、  
直接影响计算机系统性能

# 指令系统的评价

---

- 指令系统的评价

- 方便**硬件设计**，方便编译器实现，性能更优，成本功耗更低

- **硬件设计四原则**

- 简单源于规整（Simplicity favors regularity）
    - 指令越规整设计越简单
  - 越少越快（Smaller is faster）：如寄存器个数
  - 加快经常性事件（Make the common case fast）
  - 好的设计需要适度的折衷（Good design demands good compromises）
    - 如指令长度和格式



# 指令系统的评价

---

- 性能要求

- 完备性：指令丰富，功能齐全，使用方便
- 高效性：程序占空间小，执行速度快
- 规整性：RISC-V指令长度是32位和16位的压缩指令  
(关于规整性，X86中还包括对称性、匀齐性、一致性的定义)
- 兼容性：系列机软件向上兼容

# 有关ISA的若干问题

---

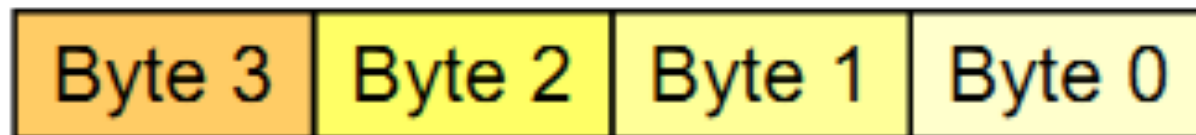
- 存储器寻址
- 操作数的类型
- 所支持的操作
- 控制转移类指令
- 指令格式

# 存储器寻址

- 80年以来几乎所有机器的存储器都是按字节编址
- 一个存储器地址可以访问：
  - 1个字节、2个字节、4个字节、更多字节.....
- 不同体系结构对字的定义是不同的
  - 16位字（Intel X86）、32位字（MIPS、RISC-V）
- 如何读32位字，两种方案
  - 每次一个字节，四次完成；每次一个字，一次完成
- 问题：
  - 如何将字节地址映射到字地址 (尾端问题)
  - 一个字是否可以存放在任何字节边界上(对齐问题)

# 尾端问题（小端 vs 大端）

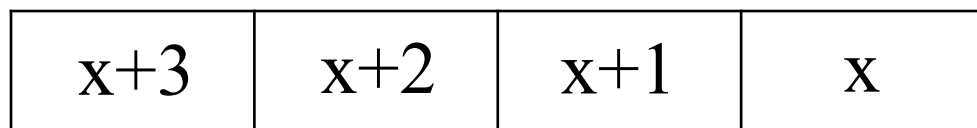
- 小端（little endian），大端（big endian），在一个字内部的字节顺序问题



例：连续存放      fe                  dc                  45                  67

00000000 00000000 00000100 00000001

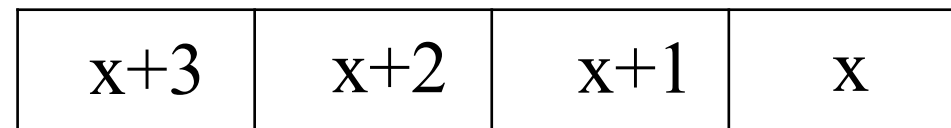
高字节放在低地址      大端



67          45          dc          fe  
00000001 00000100 00000000 00000000

IBM 360/370, Motorola 68k, MIPS, Sparc,  
HP PA

小端      高字节放在高地址



fe          dc          45          67  
00000000 00000000 00000100 00000001

Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

# 对齐问题

---

- 对于s字节的对象，访问地址为A，如果 $A \bmod s = 0$  称为边界对齐。
- 边界对齐的原因是存储器本身读写的要求，存储器本身读写通常就是边界对齐的，对于不是边界对齐的对象的访问可能要导致存储器的两次访问，然后再拼接出所需要的数。（或发生异常）

# 对齐问题

Address mod 8	0	1	2	3	4	5	6	7
Byte	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned
2 Bytes	Aligned		Aligned		Aligned		Aligned	
2 Bytes		Misaligned		Misaligned		Misaligned		Misalign
4 Bytes	Aligned				Aligned			
4 Bytes		Misaligned				Misaligned		
4 Bytes			Misaligned				Misaligned	
4 Bytes				Misaligned				Misalign
8 Bytes	Aligned							
8 Bytes		Misaligned						

# 寻址方式

- 寻址方式：如何说明要访问的对象地址

(R2) 存的主存单元地址号，是有效地址；  
Mem(R2) 是根据这个有效地址对应的内存中的物理地址取数

- 有效地址：由寻址方式说明的某一存储单元的实际存储器地址。有效地址 vs. 物理地址

	Mode	Example	Meaning	When used
寄存器寻址	Register	Add R1, R2	$R1 \leftarrow R1 + R2$	Values in registers
立即数寻址	Immediate	Add R1, 100	$R1 \leftarrow R1 + 100$	For constants
寄存器间接寻址	Register Indirect	Add R1, (R2)	$R1 \leftarrow R1 + \text{Mem}(R2)$	R2 contains address
带有偏移量的间接寻址	Displacement	Add R1, (R2+16)	$R1 \leftarrow R1 + \text{Mem}(R2+16)$	Address local variables
绝对寻址	Absolute	Add R1, (1000)	$R1 \leftarrow R1 + \text{Mem}(1000)$	Address static data
相对基址变址寻址方式	Indexed	Add R1, (R2+R3)	$R1 \leftarrow R1 + \text{Mem}(R2+R3)$	R2=base, R3=index
比例变址寻址	Scaled Index	Add R1, (R2+s*R3)	$R1 \leftarrow R1 + \text{Mem}(R2 + s*R3)$	s = scale factor = 2, 4, or 8
后增寄存器间接寻址	Post-increment	Add R1, (R2)+	$R1 \leftarrow R1 + \text{Mem}(R2)$ $R2 \leftarrow R2 + s$	Stepping through array s = element size
前增寄存器间接寻址	Pre-decrement	Add R1, -(R2)	$R2 \leftarrow R2 - s$ $R1 \leftarrow R1 + \text{Mem}(R2)$	Stepping through array s = element size

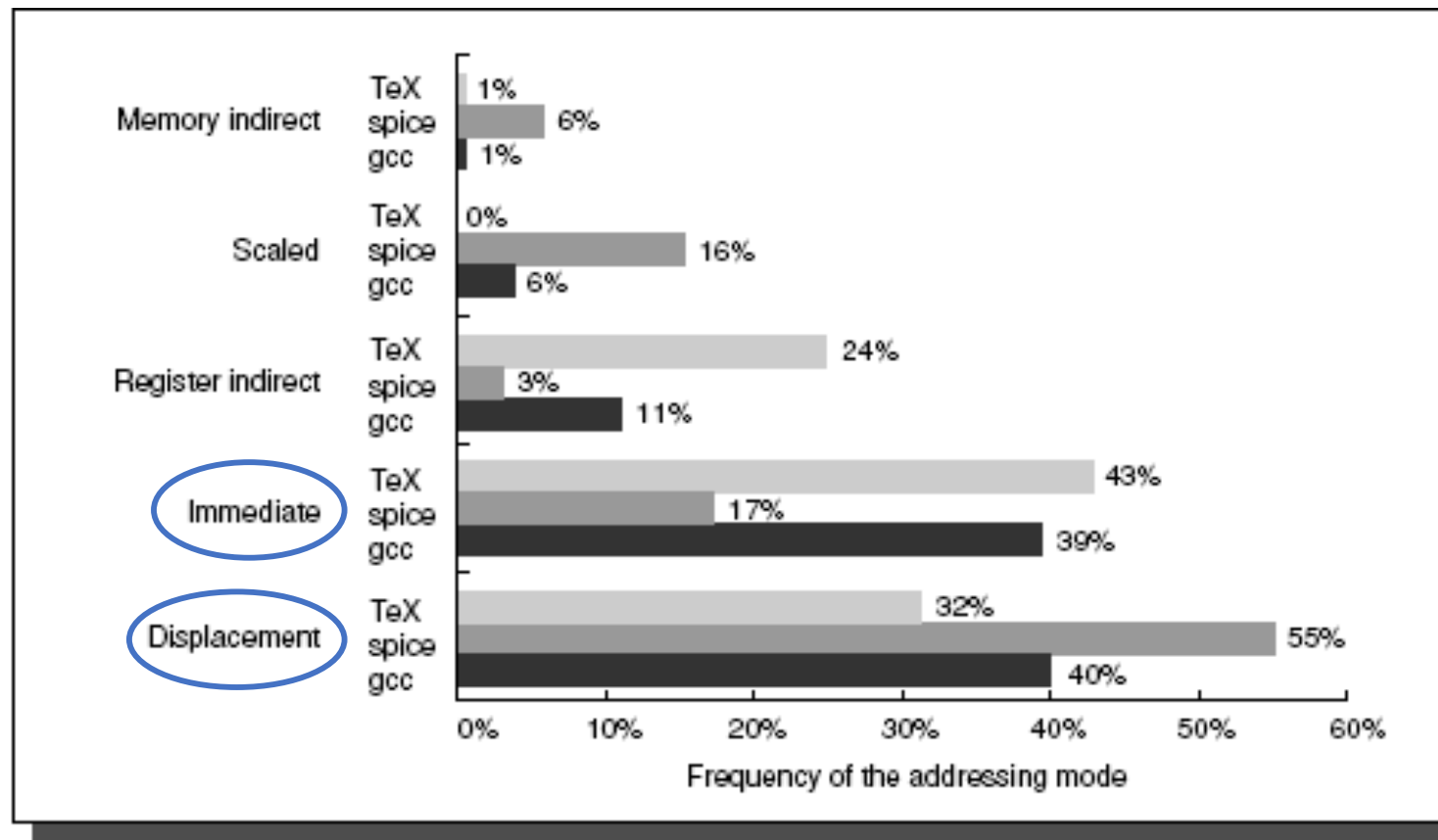
# 各种寻址方式的使用情况？

## SPEC

- (Standard Performance Evaluation Corporation) (标准性能评估协会)
- 1988年由HP、DEC、MIPS、SUN成立

## SPEC89/92/95/2000:

- SPEC建立、维护的基准测试程序标准化集，用于评价计算机性能



三个SPEC89程序在VAX结构上的测试结果：

立即寻址，偏移寻址使用较多



# 操作数的类型

- **操作数类型**：是面向应用、面向软件系统所处理的各种数据类型
  - 整型、浮点型、字符、字符串、向量类型等
  - 类型由操作码确定或数据附加硬件解释的标记，一般采用由操作码确定
  - 数据附加硬件解释的标记，现在已经不采用
- **操作数的表示**：操作数在机器中的表示，硬件结构能够识别，指令系统可以直接使用的表示格式
  - 整型：原码、反码、补码
  - 浮点：IEEE 754标准
  - 十进制：BCD码、二进制

# 常用操作数类型

---

- ASCII character = 1 byte (64-bit register can store 8 characters)
- Unicode character or Short integer = 2 bytes = 16 bits (half word)
- Integer = 4 bytes = 32 bits (word size on many RISC Processors)
- Single-precision float = 4 bytes = 32 bits (word size)
- **Long long integer = 8 bytes = 64 bits (double word)**
- **Double-precision float = 8 bytes = 64 bits (double word)**
- Extended-precision float = 10 bytes = 80 bits (Intel architecture)
- Quad-precision float = 16 bytes = 128 bits

# 第三章 RISC-V汇编及其指令系统

---

- **RISC-V概述**

- 指令系统的基本概念
- 主流指令集及发展方向
- RISC-V指令集



# 指令集架构 (ISA)


- 不同类型的CPU执行不同指令集，ISA是设计CPU的依据

 1970 DEC PDP-11    1992 ALPHA(64位)

 1978 **x86**, 2001 IA64

 1980 PowerPC

 1981 **MIPS**

 1985 **SPARC**

 1991 **ARM**

 2010 **RISC-V**

其他如ARC、Andex、C-SKY(杭州中天微系统有限公司开发)...

# 典型应用场景



服务器



桌面个人计算机



嵌入式移动设备



嵌入式实时设备



深嵌入式

- Intel公司**X86**架构的高性能CPU占垄断地位
- ARM服务器已经进入该领域（华为鲲鹏处理器）

- Intel 或 AMD 公司 X86架构的高性能CPU占垄断地位
- ARM也将有所作为

**ARM** Cortex-A 架构占垄断地位

ARM 架构占最大份额，其他RISC架构嵌入式CPU也有应用



# x86

- **x86架构（The x86 architecture）**是微处理器执行的计算机语言指令集，指一个**Intel**通用计算机系列的标准编号缩写，是具有代表性的“**可变指令长度**”的**CISC**架构。
- x86架构的复杂指令集（CISC）诞生于1978年——Intel发布16位微处理器“8086”。x86架构：
  - 高性能、扩展能力强、**操作系统兼容性好**
  - 8086、286、386、486、586(Pentium)....等多个版本
- 大多数CPU产商(如Intel、AMD)使用的就是x86指令集架构
- x86架构由于其**封闭性**，相较于其他架构成本更高

# x86

- 在过去的四十年里，英特尔的8086架构已经成为笔记本电脑、台式机和服务器市场上最流行的指令集。
  - 在嵌入式系统领域之外，几乎所有流行的软件都被移植到x86上，或者是为x86开发的
  - 它受欢迎的原因有很多：该架构在IBM PC诞生之初的偶然可用性；英特尔专注于二进制兼容性；卓有成效的微结构实现；前沿制造技术
  - 指令集设计质量并不是它流行的原因之一。
- 主要问题：
  - 1300条指令，许多寻址方式，很多特殊寄存器，多种地址翻译方式，从AMD K5微架构开始，所有的Intel支持乱序执行的微结构，都是动态地将x86指令翻译为内部的RISC-风格的指令集。
  - ISA不利于虚拟化，因为一些特权指令在用户模式下会无声地失败，而不是被捕获。VMware的工程师们用复杂的动态二进制翻译软件解决了这一缺陷
  - ISA的指令长度为任意整数字节数，最多为15个字节，但是数量较少的短操作码已被随意使用

# x86

---

- ISA有数量极少的寄存器组（通用寄存器组）
- 大多数整数寄存器在ISA中执行特殊功能，这加剧了体系结构寄存器的不足
- 大多数x86指令只有一种**破坏性**的指令格式，它会**覆盖**其中一个源操作数
- 一些ISA特性，包括隐式条件代码和带有谓词的移动操作，在微架构中实现复杂
- x86通常比RISC体系结构使用更少的动态指令完成相同的功能，因为x86指令可以编码多个基本操作。
- **x86是一个专有指令集（不开源）**





# MIPS

- MIPS是最典型的RISC指令集架构。MIPS的意思是“无内部互锁流水级的微处理器”(Microprocessor without interlocked piped stages)，其机制是尽量利用软件办法避免流水线中的数据相关问题。
  - Stanford, 1980年提出，第一个商业实现是R2000处理器（1986）
  - 最初设计的整数指令集**仅有58条指令**，直接实现单发射、顺序流水线
  - 40年来，逐步增加到**约400条指令**。
- 主要特征：
  - Load/Store型结构，专门的指令完成存储器与寄存器之间的传送
  - ALU类指令的操作数来源于寄存器或立即数（指令中的特定区域）
  - 降低了指令集和硬件的复杂性，依赖于优化编译技术，方便了简单流水线的实现；**寻址方式、指令操作非常简单**
- 广泛用于嵌入式系统，在PC机、服务器中也有应用，如SONY和任天堂游戏机、Cisco路由器。相比x86更加简洁雅致，适于教学

# MIPS的问题

- 针对特定的微体系架构的实现方式（5级流水、单发射、顺序流水线）进行**过度的优化设计**
  - 延迟转移问题导致超标量等复杂流水线的实现难度，当无法有效填充延迟槽时会导致代码尺寸变大
  - MIPS-I中暴露出其他流水线冲突（load、乘除引起的冲突）采用简单的Interlocking 简单又高效，但为了**保持兼容性，仍然保留了**延迟转移
- ISA对位置无关代码（position-independent code, PIC)支持不足
  - 直接跳转没有提供PC相对寻址，需要通过间接跳转方式实现PIC，增加了代码尺寸，降低了性能
  - 2014年MIPS的修订，改进了PC-相对寻址(针对数据)，但仍然要多条指令才能完成

# MIPS

- 乘除指令使用了特殊的寄存器（HI, LO），导致上下文切换内容、指令条数、代码尺寸增加，微架构实现复杂
- 在标准的应用程序二进制接口（Application Binary Interface, ABI）中，保留两个整型寄存器用于内核程序，减少了用户程序可用的寄存器数
- 使用特殊指令处理未对齐的load和store会消耗大量的操作码空间，并使除了最简单的实现之外的其他实现复杂化
- 时钟速率/CPI 的**权衡**使得架构师**省略了**整数大小比较和分支指令。随着分支预测和静态CMOS逻辑的出现，这种权衡在今天已经不太合适了
- 除了技术方面，MIPS是**非开放**的专属指令集，不能自由使用

# MIPS 与 X86 差异

	X86	MIPS
1	变长（1-15bytes）	定长指令
2	指令数多 CISC	指令数少 RISC
3	8个通用寄存器	32个通用寄存器
4	寻址方式复杂	寻址方式简单
5	有标志寄存器	无标志寄存器
6	最多两地址指令	三地址指令
7	无限制	只有Load/store能访问存储器
8	有堆栈指令 push, pop	无堆栈指令（访存指令代替）
9	有I/O指令	无I/O指令(设备统一编址)
10	参数传递：栈帧	参数传递（4寄存器+栈帧）



# SPARC

- 1985年，SUN公司设计，全称为“可扩充处理器架构”，设计出发点是服务于工作站（大型服务器），拥有一个大型的寄存器窗口，有72-640个之多的64位通用寄存器。后来SUN被Oracle收购，2017年9月，Oracle放弃硬件业务，SPARC处理器退出历史舞台。**功耗面积太大，不适于PC或嵌入式领域。**
- Sun Microsystems的专属指令集
  - 可追溯到Berkeley RISC-I和RISC-II项目；
  - 最近的32位版本的ISA SPARC V8：用户级 整型**ISA 90条指令**；硬件支持IEEE 754-1985标准的浮点数(50条)；特权级指令 **20条**

## 主要问题

- **SPARC使用了寄存器窗口来加速函数调用**：当函数调用所需的栈空间超过了窗口的寄存器数，性能会急剧下降。对于所有的实现来说，寄存器窗口都消耗很大的面积和功耗
- **分支使用条件码**：这些条件码由于在一些指令之间创建了额外的依赖关系，增加了体系结构状态并使实现复杂化
- **load和store相邻寄存器对的指令**：可以在很少增加硬件复杂性的情况下提高吞吐量；但是使用寄存器重命名使实现复杂化，因为在寄存器文件中数据在物理上可能不再相邻

# SPARC

- 浮点寄存器文件和整数寄存器文件之间的移动必须使用内存系统作为中介，限制了系统性能
  - ISA通过体系结构公开的延迟陷阱队列支持非精确浮点异常，该队列向系统监控程序提供信息，以恢复此类异常上的处理器状态
  - 唯一的原子内存操作是fetch-and-store，这对于实现许多无等待的数据结构是不够的
- SPARC有许多其他80年代RISC结构相似的缺陷特性：
  - ISA设计面向单发射、顺序、五级流水线的微体系架构；
  - SPARC具有分支延迟插槽和许多显式的数据和控制冲突，这些冲突使代码生成复杂化，无助于更积极的实现；
  - 缺乏位置无关的寻址方式（相对寻址）；
  - 由于SPARC缺乏足够的自由编码空间，因此不能方便地对其进行改进以支持压缩ISA扩展。

# Alpha (DEC)



- DEC公司的架构师在20世纪90年代初定义了他们的RISC ISA——Alpha
  - 也称为Alpha AXP，创建了64位寻址空间、设计简洁、实现简单、高性能的ISA
  - Alpha处理器**最早**跨过1GHZ的处理器，**最早**计划采用双核、甚至多核架构的处理器
  - 摒弃了当时非常吸引人的特性，如分支延迟、条件码、寄存器窗口等
  - 将特权体系结构和硬件平台的大部分细节隔离在抽象接口(特权体系结构库)后面(PALcode)
- **主要问题**：DEC对顺序微架构的Alpha进行了**过度优化**，并添加了一些不太适合现代实现的特性
  - 为了追求高时钟频率，ISA的原始版本避免了8位和16位的加载和存储，实际上创建了一个字寻址的内存系统。此外，添加了特殊的未对齐的加载和存储指令以及一些整数指令，以加速重新组合过程。
  - 为了方便长延迟浮点指令的乱序完成，Alpha 有一个非精确的浮点陷阱模型。ISA还定义了异常标记和默认值(如果需要的话)必须由软件例程提供。
  - Alpha缺少整数除法指令，建议使用软件牛顿迭代法实现，导致浮点除法速度高于整数除法。



# Alpha (DEC)

- 与它的前辈RISC一样，没有预先考虑可能的压缩指令集扩展，因此没有足够的操作码空间来进行更新
- ISA包含有条件的移动，这使得微架构与寄存器重命名复杂化
  - 如果移动条件不满足，指令仍然必须将旧值复制到新的物理目标寄存器中。这实际上使条件移动成为ISA中惟一读取三个源操作数的指令。
  - DEC的第一个乱序执行的实现使用了一些技巧来避免该指令的额外数据路径。Alpha 21264通过将条件移动指令分解为两个微操作来执行，第一个微操作评估移动条件，第二个微操作执行移动。这种方法还要求物理寄存器文件加宽一位以保存中间结果。
- 使用商业Alpha ISAs的一个重要风险:它们可能会被摒弃。
- 康柏在上世纪90年代末收购了摇摇欲坠的DEC后不久，逐步将其全部64位服务器系列产品转移到英特尔的安腾架构上。2004年，惠普收购了康柏，从此Alpha架构逐渐淡出人们视野。



# ARMv7

- ARM处理器是英国Acorn有限公司设计的低功耗成本的**第一款RISC**微处理器。全称为Advanced RISC Machine。
- 32位 RISC ISA
  - **目前世界上使用最广的体系结构**。当我们权衡是否要设计自己的指令集时，ARMv7是一个自然的选择，大量的软件已经被移植到该ISA上，而且它在嵌入式和移动设备中无处不在。
  - 是一个封闭的标准，剪裁或扩充是不允许的，即使是微架构的创新也仅限于那些能够获得ARM架构许可的人
  - ARMv7十分庞大复杂。**整型类指令600+条**
- 即使知识产权不是问题，它仍然存在一些技术缺陷
  - 不支持64位地址，ISA缺乏硬件支持IEEE754-2008标准（ARMv8纠正了这些缺陷）
  - 特权体系结构的细节渗透到用户级体系结构的定义中

# ARMv7

- ARMv7附带一个压缩ISA，具有固定宽度的16位指令，称为Thumb。
  - Thumb虽然提供了有竞争力的代码尺寸，但性能较差
  - Thumb-2 虽然提供了较高的性能，但32位的Thumb-2编码方式与基本的ISA编码方式不同,16位的Thumb-2的编码方式与基本的16位编码方式也不同。导致译码器需要理解三种编码格式，使得能耗、延迟以及设计成本增加
- ISA中包含了许多实现复杂的特性：
  - 程序计数器是可寻址寄存器之一，这意味着几乎任何指令都可以改变控制流。
  - 更糟糕的是，程序计数器的最低有效位反映ISA当前正在执行(ARM或Thumb)哪个ISA——简单的ADD指令可以更改ISA当前在处理器上执行的指令！
  - 分支使用条件码以及谓词指令进一步使高性能实现复杂化。

# ARMv8

---

- 2011年，ARM发布新的ISA ARMv8
  - 64位地址; 扩展了整型寄存器组。
  - 摒弃了ARMv7中实现复杂的一些特性
    - PC不再是整形寄存器组的成员;
    - 不再有谓词指令
    - 删除了load-multiple和store-multiple 指令
    - 指令编码归一化
- 主要问题
  - 使用条件码
  - 存在许多特殊的寄存器

# ARMv8

---

增加了一些缺陷，

- 包括大量的subword-SIMD架构
  - 指令集更加厚重：**1070条指令，53种格式，8种寻址方式。说明文档达到了5778页**
  - 与其他大多数ISA一样，通常以暴露底层实现的方式将用户和特权架构紧密地结合在一起
  - 此外，随着ARMv8的引入，ARM不再支持压缩指令编码
- 和它的以前版本一样，ARMv8也是一个封闭的标准

# ARM架构处理器发布时间和特点

表 1-4 Cortex-A 系列处理器的发布时间和特点

型 号	发布时间	位数	架构	流水线深度	指令发射类型	乱序执行还是顺序执行	核数
Cortex-A8	2005 年	32	ARMv7-A	13 级	双发射	乱序执行	1
Cortex-A9	2007 年	32	ARMv7-A	8 级	双发射	乱序执行	1~4
Cortex-A5	2009 年	32	ARMv7-A	8 级	单发射	顺序执行	1~4
Cortex-A15	2010 年	32	ARMv7-A	15 级	三发射	乱序执行	1~4
Cortex-A7	2011 年	32	ARMv7-A	8 级	部分双发射	顺序执行	1~8
Cortex-A53	2011 年	64	ARMv8-A	可以理解为 Cortex-A7 的 64 位版			
Cortex-A57	2010 年	64	ARMv8-A	可以理解为 Cortex-A15 的 64 位版			
Cortex-A12	2013 年	32	ARMv7-A	可以理解为 Cortex-A9 的性能提升优化版本			
Cortex-A17	2014 年	32	ARMv7-A	可以理解为 Cortex-A12 性能提升后的优化版本			
Cortex-A35	2015 年	64	ARMv8-A	8 级	部分双发射	顺序执行	1~8
Cortex-A72	2015 年	64	ARMv8-A	可以理解为 Cortex-A57 的性能提升后的优化版本			
Cortex-A73	2015 年	64	ARMv8-A	可以理解为 Cortex-A72 的性能进一步提升后的优化版本			
Cortex-A32	2016 年	32	ARMv8-A	可以理解为 Cortex-A35 的 32 位版本			
Cortex-A55	2017 年	64	ARMv8.2-A	可以理解为 Cortex-A53 的功耗进一步降低后的优化版本			
Cortex-A75	2017 年	64	ARMv8.2-A	可以理解为 Cortex-A73 的性能进一步提升后的优化版本			

# 国内主流指令集

## • MIPS阵营



- **龙芯**：中国科学院计算技术所龙芯课题组研制，授权北京神州龙芯集成电路设计公司研发。龙芯3B系列国产商用8核处理器，主频超过1GHZ，主要用于高性能计算机/服务器、数字信号处理领域。



- **君正**：北京君正公司生产，主要针对可穿戴设备、物联网IC设计。国内第一批上市的电子手表（如果壳一代、土曼一代/二代）采用了君正方案

## • X86阵营

- 众志：北京北大众志微系统科技有限责任公司生产。2005年，北大微电子中心获得了AMD Geode-2处理器的技术授权
- 兆芯（VIA，台湾威盛）：上海兆芯集成电路有限公司生产。
- 海光（AMD）：2016年，AMD宣布将x86技术授权给天津海光公司。

# 国内主流指令集

- 自主指令
  - 申威 (Alpha指令集扩展): 面向高性能计算、服务器领域, 代表产品“神威·太湖之光”
- ARM阵营
  - 飞腾: 天津飞腾公司由国防科技大学高性能处理研究团队建立。
  - 海思: 前身为华为集成电路设计中心, 已购买ARM架构授权, 研发自有处理器核, 主攻服务器市场, 代表性产品“麒麟芯片”、泰山服务器。
  - 展讯、松果
- RISC-V阵营
  - 阿里-平头哥: 平头哥半导体有限公司于2018年10月31日成立, 基于RISC-V的处理器内核IP玄铁910应用于5G、人工智能及自动驾驶。
  - 华米科技、芯来科技

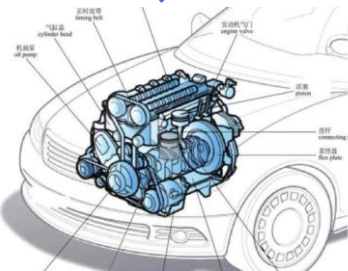


# 以ARM为例——芯片 vs 架构

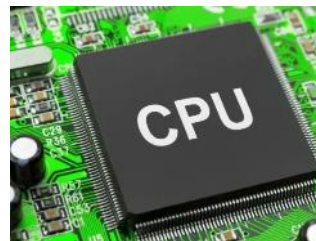


跑车  
轿车  
货车  
...

生产商：奔驰、宝马、大众、丰田...

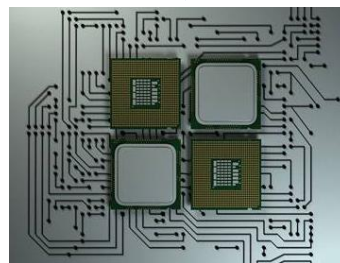


采购/设计发动机



处理器功能：处理器**芯片**  
辅助功能：普通**芯片/芯片**

生产商：高通、联发科、三星、德州仪器...



- ❑ 发明一种处理器架构
- ❑ 购买其他商业公司非ARM架构处理器IP
- ❑ 找ARM公司
  - 购买ARM公司**ARM架构授权**，**自己定制**开发基于ARM架构的处理器
  - 购买ARM公司提供的**ARM处理器IP**（需支付前期授权费+大规模生产的芯片版税）



# 指令集架构：CISC & RISC

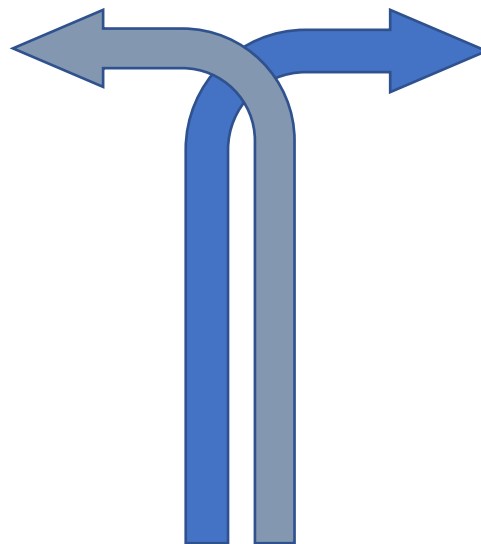


## CISC

- 复杂指令系统计算机 (Complex Instruction Set Computer)
- **指令数目多**：含有处理器常用和不常用的特殊指令
- CPU诞生的早期是主流

## RISC

- 精简指令系统计算机 (Reduced Instruction Set Computer)
- **指令数目少**：仅含有处理器常用指令；对于不常用的指令，通过执行多条常用指令的方式实现



# 指令集架构：CISC & RISC

---

- ISA的功能设计
  - 任务：确定硬件支持哪些操作
  - 方法：统计的方法
- CISC（Complex Instruction Set Computer）
  - 目标：强化指令功能，减少运行的指令条数，提高系统性能
  - 方法：面向目标程序的优化，面向高级语言和编译器的优化
- RISC（Reduced Instruction Set Computer）
  - 目标：通过简化指令系统，用高效的方法实现最常用的指令
  - 方法：充分发挥流水线的效率，降低（优化）CPI

# RISC的定义和特点

---

- **RISC是一种计算机体系结构的设计思想，不是一种产品。**
  - 直到现在，RISC没有一个确切的定义
- 是近代计算机体系结构发展史中的一个里程碑
- 早期对RISC特点的描述：
  - 大多数指令在单周期内完成、采用Load/Store结构、硬布线控制逻辑
  - 减少指令和寻址方式的种类、固定的指令格式
  - 注重代码的优化
- 从目前的发展看，RISC体系结构还应具有如下特点：
  - 面向寄存器结构
  - 十分重视流水线的执行效率—尽量减少断流
  - 重视优化编译技术
- 减少指令平均执行周期数是RISC思想的精华

# 精简指令系统(RISC)

- 指令条数少，保留使用频率最高的简单指令，软件实现复杂指令
  - 便于硬件实现
- Load/Store架构
  - 只有存/取数指令才能访问存储器，其余指令的操作都在寄存器之间进行
  - 便于硬件实现
- 指令长度固定，指令格式简单、寻址方式简单
  - 便于硬件实现
- CPU设置大量寄存器（32~192）
  - 便于编译器实现
- RISC CPU采用硬布线控制，CISC采用微程序
- 一个时钟周期完成一条机器指令（单周期模型）

# OpenRISC

---

- 是一个开放源码处理器设计项目
  - 适用于教学、科研和工业界的实现（Hennessy和Patterson）。
- 主要问题
  - 主要是开源处理器设计项目，而不是开源的ISA 规格说明，**ISA和实现是紧密耦合的**
  - 固定的32位编码与16位立即数阻碍了压缩ISA扩展
  - 硬件不支持IEEE 754-2008标准
  - 用于分支和条件转移的条件码使高性能实现复杂化
  - ISA对位置无关的寻址方式支持较弱
  - OpenRISC不利于虚拟化。从异常返回的指令L.RFE，定义为在用户模式下功能，而不是捕获
  - 值得一提的是：2010年这两个问题都得到了解决:延迟插槽已经成为可选的，64位版本已经定义(但是，据我们所知，从未实现过)。
- 最终，我们(UCB)认为最好从头开始，而不是相应地修改OpenRISC。

# 采用RISC体系结构的微处理器

---

- SUN Microsystem: SPARC, SuperSPARC, Ultra SPARC
- SGI: MIPS R4000, R5000, R10000,
- IBM: Power PC
- Intel: 80860, 80960
- DEC: Alpha
- Motorola 88100
- HP HP300/930系列, 950系列
- ARM, MIPS
- RISC-V

# 问题

**问题：RISC的指令系统精简了，CISC中的一条指令可能由一串指令才能完成，那么为什么RISC执行程序的速度比CISC还要快？**

	IC	CPI	T
CISC	1	2~15	33ns~5ns
RISC	1.3~1.4	1.1~1.4	10ns~2ns

**IC：** 实际统计结果，RISC的IC只比CISC 长30%~40%

**CPI:** CISC CPI一般在4~6之间，RISC 一般CPI =1 , Load/Store 为2

**T:** RISC采用硬布线逻辑，指令要完成的功能比较简单

# RISC为什么会减少CPI

---

- 硬件方面：
  - 硬布线控制逻辑
  - 减少指令和寻址方式的种类
  - 使用固定格式
  - 采用Load/Store
  - 指令执行过程中设置多级流水线。
- 软件方面： 十分强调优化编译的作用



# 指令系统发展方向（CISC-RISC）

---

- CISC—复杂指令集计算机(Complex Instruction Set Computer)
  - 指令数量多，指令功能复杂，几百条指令。
  - 每条指令都有对应的电路设计，CPU电路设计复杂，功耗较大。
  - 对应编译器的设计简单（各种操作都有对应的指令）。
  - Intel x86
- RISC---精简指令集计算机(Reduced Instruction Set Computer)
  - 指令数量少，指令功能单一，通常只有几十条指令。
  - CPU设计相对简单，功耗较小。
  - 编译器的设计比较复杂（许多操作需要一些指令的灵活组合）
  - 1982年后的指令系统基本都是RISC
  - ARM、MIPS、RISC-V
- CISC、RISC互相融合：外围 CISC 架构，内部 RISC 架构

# 第三章 RISC-V汇编及其指令系统

---

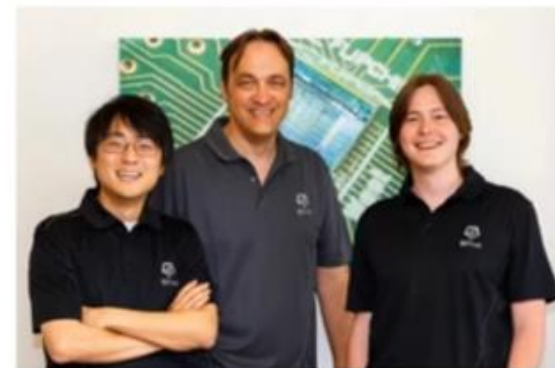
- **RISC-V概述**

- 指令系统的基本概念
- 主流指令集及发展方向
- **RISC-V指令集**



# RISC-V指令集历史

- 加州大学伯克利分校Krste Asanovic教授、Andrew Waterman和Yunsup Lee等开发人员于2010年发明。
  - 其中"RISC"表示精简指令集，而其中"V"表示伯克利分校从RISC I开始设计的第五代指令集。
- 基于BSD协议许可的**免费开放**的指令集架构
- 适合多层次计算机系统
  - 从微控制器到超级计算机
  - 支持大量定制与加速功能；支持32bit、64bit、128bit
- 规范由RISC-V**非营利**性基金会维护
  - 规范包括：指令集标准手册与架构文档
  - 每年举行两次研讨会，第六次研讨会于2017年5月在“上交”举办



# RISC-V ISA设计理念

## • 通用的ISA

- 能适应从最袖珍的嵌入式控制器，到最快的高性能计算机等各种规模的处理器。
- 能兼容各种流行的软件栈和编程语言。
- 适应所有实现技术，包括现场可编程门阵列（FPGA）、专用集成电路（ASIC）、全定制芯片，甚至未来的技术。
- 对所有微体系结构实现方式都有效。例如：
  - 微编码或硬连线控制；顺序或乱序执行流水线；单发射或超标量等等。
- 支持广泛的定制化，成为定制加速器的基础。随着摩尔定律的消退，加速器的重要性日益提高。
- 基础的指令集架构是稳定的。不能像以前的专有指令集架构一样被弃用，例如AMD Am29000、Digital Alpha、Digital VAX、Hewlett Packard PA-RISC、Intel i860、Intel i960、Motorola 88000、以及Zilog Z8000。
- 完全开源

# RISC-V架构的特点

- 指令集架构简单
  - **指令集的238页**，特权级编程手册135页，其中RV32I只有16页
  - 作为对比，Intel的处理器手册有5000多页
  - 新的体系结构设计吸取了经验和最新的研究成果
  - 指令数量少，基本的RISC-V指令数目仅有40多条，加上其他的模块化扩展指令总共几十条指令。
- **模块化的指令集设计**
  - 不同的部分还能以模块化的方式组织在一起
  - ARM的架构分为A、R和M三个系列，分别针对于Application（应用操作系统）、Real-Time（实时）和Embedded（嵌入式）三个领域，彼此之间并不兼容
  - RISC-V嵌入式场景，用户可以选择RV32IC组合的指令集，仅使用Machine Mode（机器模式）；而高性能操作系统场景则可以选择譬如RV32IMFDC的指令集，使用Machine Mode（机器模式）与User Mode（用户模式）两种模式，两种使用方式的**共同部分相互兼容**

# RISC-V的模块化设计

---

- RISC-V指令集使用模块化的方式进行组织，每个模块使用一个英文字母来表示
- RISC-V最基本也是唯一强制要求实现的指令集部分是由I字母表示的基本整数指令子集，使用该整数指令子集，便能够实现完整的软件编译器
- 其他的指令子集部分均为可选的模块，具有代表性的模块包括M/A/F/D/C
- RISC-V预留了大量的指令编码空间用于用户的自定义扩展，还定义了四条Custom指令可供用户直接使用，每条Custom指令都有几个比特位的子编码空间预留，用户可以直接使用四条Custom指令扩展出几十条自定义的指令。

# 模块化的RISC-V 指令子集

基本指令集	指令数	描述
RV32I	47	32位地址空间与整数指令，支持32个通用整数寄存器
RV32E	47	RV32I的子集，仅支持16个通用整数寄存器（嵌入式架构）
RV64I	51	64位地址空间与整数指令及一部分32位的整数指令
RV128I	71	128位地址空间与整数指令及一部分64位和32位的指令
RV64扩展指令集	指令数	描述
I	51	基本体系结构
M	13	整数乘法与除法指令
A	22	原子操作(存储原子操作和load-reserved/store-conditional指令)
F	30	单精度（32bit）浮点指令
D	32	双精度（64bit）浮点指令
C	36	压缩指令

提高代码密度

# 可配置的寄存器组

- 通用寄存器（GPR: General Purpose Registers）
  - **32位**架构(RV**32I**): 32个**32位**的通用寄存器;
  - **64位**架构(RV**64I**): 32个**64位**的通用寄存器
  - 嵌入式架构RV32E有16个32位的通用寄存器
  - 支持单精度浮点数（F），或者双精度浮点数（D），另外增加一组独立的通用浮点寄存器组，f0~f31
- 控制状态寄存器（CSR: Control and Status Registers）
  - 用于配置或记录一些运行的状态（异常和中断处理中常用）
  - 处理器核内部的寄存器，使用专有的12位地址码空间



# 规整的指令编码

- 所有通用寄存器在指令码的位置是一样的，方便译码；
- 所有的指令都是32位字长，有 6 种指令格式：寄存器型、立即数型、存储型、分支指令、跳转指令和大立即数

R 型	funct7	rs2	rs1	funct3	rd	opcode
I 型	imm[11:0]		rs1	funct3	rd	opcode
S 型	Imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB / B 型	Imm[12,10:5]	rs2	rs1	funct3	imm[4:1,11]	opcode
UJ / J 型	Imm[20,10:1,11,19:12]				rd	opcode
U 型	Imm[31:12]				rd	opcode

# RISC-V的数据传输指令

---

- 专用内存到寄存器之间传输数据的指令，其它指令都只能操作寄存器
  - 简化硬件设计
  - 支持字节（8位）、半字（16位）、字（32位）、双字（64位，64位架构）的数据传输
  - 推荐但不强制地址对齐
  - 小端格式
  - 采用松散存储器模型，对于访问不同地址的存储器读写指令的执行顺序没有要求

# RISC-V的特权模式

---

- RISC-V架构定义了三种工作模式，又称特权模式（Privileged Mode）：
  - Machine Mode: 机器模式，简称M Mode
  - Supervisor Mode: 监督模式，简称S Mode
  - User Mode: 用户模式，简称U Mode
- RISC-V架构定义M Mode为必选模式，另外两种为可选模式。通过不同的模式组合可以实现不同的系统

## RISC-V

## • RISC-V官方指令集手册

<https://riscv.org/specification>

## • 中文简化版

<http://riscvbook.com/chinese>

[v2p1.pdf](http://riscvbook.com/chinese-v2p1.pdf)

Base Integer Instructions: RV32I and RV64I						RV Privileged Instructions								
Category	Name	Fmt	RV32I Base			+RV64I			Category	Name	Fmt	RV mnemonic		
Shifts	Shift Left Logical	R	SLL	rd,rs1,rs2		SLLW	rd,rs1,rs2		Trap Mach-mode trap return	R	MRET			
	Shift Left Log. Imm.	I	SLLI	rd,rs1,shamt		SLLIW	rd,rs1,shamt		Supervisor-mode trap return	R	SRET			
	Shift Right Logical	R	SRL	rd,rs1,rs2		SRLW	rd,rs1,rs2		Interrupt Wait for Interrupt	R	WFI			
	Shift Right Log. Imm.	I	SRLI	rd,rs1,shamt		SRLIW	rd,rs1,shamt		MMU Virtual Memory FENCE	R	FENCE.VMA rs1,rs2			
	Shift Right Arithmetic	R	SRA	rd,rs1,rs2		SRAW	rd,rs1,rs2		Examples of the 60 RV Pseudoinstructions					
Arithmetic	Shift Right Arith. Imm.	I	SRAI	rd,rs1,shamt		SRAIW	rd,rs1,shamt		Branch = 0 (BEQ rs,x0,imm)	J	BEQ rs,imm			
	ADD	R	ADD	rd,rs1,rs2		ADDW	rd,rs1,rs2		Jump (uses JAL x0,imm)	J	J imm			
	ADD Immediate	I	ADDI	rd,rs1,imm		ADDIW	rd,rs1,imm		MoVe (uses ADDI rd,rs,0)	R	MV rd,rs			
	SUBtract	R	SUB	rd,rs1,rs2		SUBW	rd,rs1,rs2		RETurn (uses JALR x0,0,rs)	I	RET			
	Load Upper Imm	U	LUI	rd,imm					Optional Compressed (16-bit) Instruction Extension: RV32C					
Add Upper Imm to PC						U	AUIPC	rd,imm	Category Name Fmt RVC RISC-V equivalent					
Logical	XOR	R	XOR	rd,rs1,rs2					Loads	Load Word	CL	C.LW rd',rs1',imm	LW rd',rs1',imm*4	
	XOR Immediate	I	XORI	rd,rs1,imm						Load Word SP	CI	C.LWSP rd,imm	LW rd,sp,imm*4	
	OR	R	OR	rd,rs1,rs2						Float Load Word SP	CL	C.FLW rd',rs1',imm	FLW rd',rs1',imm*8	
	OR Immediate	I	ORI	rd,rs1,imm						Float Load Word	CI	C.FLWSP rd,imm	FLW rd,sp,imm*8	
	AND	R	AND	rd,rs1,rs2						Float Load Double	CL	C.FLD rd',rs1',imm	FLD rd',rs1',imm*16	
	AND Immediate	I	ANDI	rd,rs1,imm						Float Load Double SP	CI	C.FLDSP rd,imm	FLD rd,sp,imm*16	
Compare	Set <	R	SLT	rd,rs1,rs2					Stores	Store Word	CS	C.SW rs1',rs2',imm	SW rs1',rs2',imm*4	
	Set < Immediate	I	SLTI	rd,rs1,imm						Store Word SP	CSS	C.SWSP rs2,imm	SW rs2,sp,imm*4	
	Set < Unsigned	R	SLTU	rd,rs1,rs2						Float Store Word	CS	C.FSW rs1',rs2',imm	FSW rs1',rs2',imm*8	
	Set < Imm Unsigned	I	SLTIU	rd,rs1,imm						Float Store Word SP	CSS	C.FSWSP rs2,imm	FSW rs2,sp,imm*8	
Branches	Branch =	B	BEQ	rs1,rs2,imm						Float Store Double	CS	C.FSD rs1',rs2',imm	FSD rs1',rs2',imm*16	
	Branch ≠	B	BNE	rs1,rs2,imm						Float Store Double SP	CSS	C.FSDSP rs2,imm	FSD rs2,sp,imm*16	
	Branch <	B	BLT	rs1,rs2,imm					Arithmetic	ADD	CR	C.ADD rd,rs1	ADD rd,rd,rs1	
	Branch ≥	B	BGE	rs1,rs2,imm						ADD Immediate	CI	C.ADDI rd,imm	ADDI rd,rd,imm	
	Branch < Unsigned	B	BLTU	rs1,rs2,imm						ADD SP Imm * 16	CI	C.ADDI16SP x0,imm	ADDI sp,sp,imm*16	
	Branch ≥ Unsigned	B	BGEU	rs1,rs2,imm						ADD SP Imm * 4	CIW	C.ADDI4SPH rd',imm	ADDI rd',sp,imm*4	
Jump & Link	J&L	J	JAL	rd,imm						SUB	CR	C.SUB rd,rs1	SUB rd,rd,rs1	
	Jump & Link Register	I	JALR	rd,rs1,imm						AND	CR	C.AND rd,rs1	AND rd,rd,rs1	
Synch	Synch thread	I	FENCE							AND Immediate	CI	C.ANDI rd,imm	ANDI rd,rd,imm	
	Synch Instr & Data	I	FENCE.I							OR	CR	C.OR rd,rs1	OR rd,rd,rs1	
Environment	CALL	I	ECALL							eXclusive OR	CR	C.XOR rd,rs1	XOR rd,rd,rs1	
	BREAK	I	EBREAK							MoVe	CR	C.MV rd,rs1	MV rd,rd,rs1	
Control Status Register (CSR)										Load Immediate	CI	C.LI rd,imm	LI rd,imm	
	Read/Write	I	CSRWR	rd,csr,rs1						Load Upper Imm	CI	C.LUI rd,imm	LUI rd,imm	
	Read & Set Bit	I	CSRRE	rd,csr,rs1						Shifts	Shift Left Imm	CI	C.SLLI rd,imm	SLLI rd,rd,imm
	Read & Clear Bit	I	CSRRC	rd,csr,rs1							Shift Right Arl. Imm.	CI	C.SRAI rd,imm	SRAI rd,rd,imm
	Read/Write Imm	I	CSRRI	rd,csr,imm							Shift Right Log. Imm.	CI	C.SRLI rd,imm	SRLI rd,rd,imm
	Read & Set Bit Imm	I	CSRREI	rd,csr,imm						Branches	Branch=0	CB	C.BEQ rs1',imm	BEQ rs1',x0,imm
	Read & Clear Bit Imm	I	CSRRCI	rd,csr,imm							Branch≠0	CB	C.BNE rs1',imm	BNE rs1',x0,imm
Loads	Load Byte	I	LB	rd,rs1,imm						Jump	Jump	CJ	C.J imm	JAL x0,imm
	Load Halfword	I	LBH	rd,rs1,imm							Jump Register	CR	C.JR rd,rs1	JALR x0,rs1,0
	Load Byte Unsigned	I	LBU	rd,rs1,imm						Jump & Link	J&L	CJAL	imm	JAL rs,imm
	Load Half Unsigned	I	LBUH	rd,rs1,imm							Jump & Link Register	CR	C.JALR rs1	JALR rs,rs1,0
Stores	Load Word	I	LW	rd,rs1,imm						System Env. BREAK	CI	C.EBREAK		EBREAK
	Store Byte	S	SB	rs1,rs2,imm						+RV64I				
	Store Halfword	S	SH	rs1,rs2,imm						LWU	rd,rs1,imm			
	Store Word	S	SW	rs1,rs2,imm						LD	rd,rs1,imm			
		S	SD	rs1,rs2,imm						GD	rs1,rs2,imm			
32-bit Instruction Formats						16-bit (RVC) Instruction Formats								
R	31	27	26	25	24	20	19	15	14	12	11	7	6	0
I	funct7		rs2			funct3			rd			opcode		
S	imm[11:0]		rs1			funct3			rd			opcode		
B	imm[11:5]		rs2			funct3			imm[4:0]			opcode		
U	imm[31:12]		rs1			funct3			imm[4:1]			opcode		
J	imm[20:10]		rs1			funct3			imm[19:12]			rd opcode		
CR	funct4		rd/rs1			funct3			imm			rs2 op		
CI	funct3		imm			funct3			rd/rs1			imm op		
CSS	funct3		imm			funct3			imm			rs2 op		
CIW	funct3		imm			funct3			imm			rd' op		
CL	funct3		imm			funct3			imm			rs1' op		
CS	funct3		imm			funct3			imm			rs2' op		
CB	funct3		offset			funct3			offset			rs1' op		
CJ	funct3		jump target			funct3			jump target			op		

RISC-V Integer Base (RV32I/64I), privileged, and optional RV32C/64C. Registers  $x1-x31$  and the PC are 32 bits wide in RV32I and 64 in RV64I ( $x0=0$ ). RV64I adds 12 instructions for the wider data. Every 16-bit RVC instruction maps to an existing 32-bit RISC-V instruction.

# RISC-V指令集系统架构小结

---



- 完全开放的 ISA
- 精简
  - 包含一个最小的ISA固定核心（可支撑OS，方便教学）
  - 适合硬件实现，而不仅仅是适用于模拟或者二进制翻译
- 后发优势
  - 模块化的可扩展指令集
  - 简化硬件实现，提升性能
    - 更规整的指令编码、更简洁的运算指令、更简洁的访存模式：Load/Store架构
    - 高效分支跳转指令（减少指令数目）、简洁的子程序调用
    - 无条件码执行、无分支延迟槽