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，它是一系列二进制代码
- **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
- **指令集系统架构（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和结构，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，基本系统架构相同的计算机
  - 解决软件兼容的问题。**给定一个ISA，可以有不同的实现方式**；例如：AMD/Intel CPU 都是X86-64指令集。ARM 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

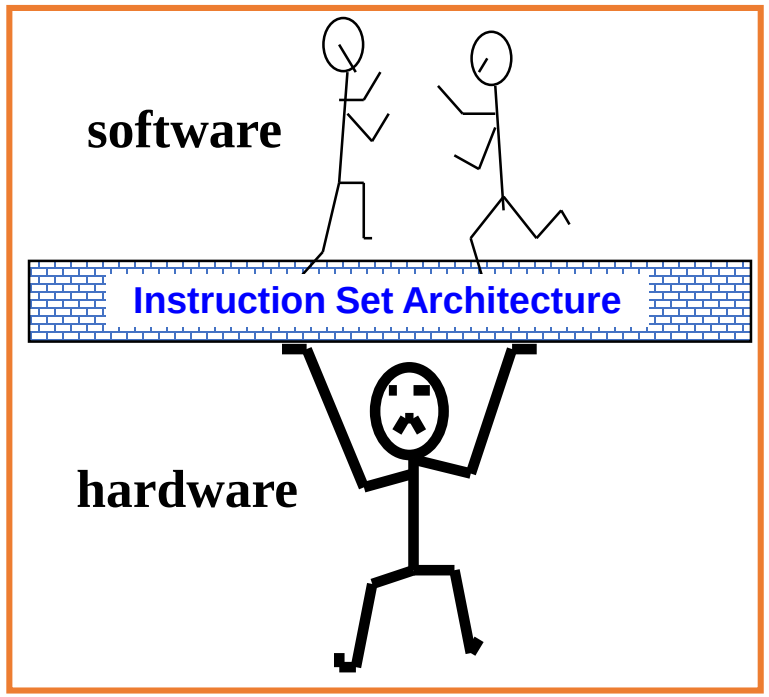
系统模型

- 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

成本和资源占用低

简洁性：指令规整简洁；

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(桌面、服务器、 嵌入式应用)

易于编程/编译/链接

- 为高层软件的设计与开发提供方便的功能

性能好

- 方便低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
- ISA位宽和指令编码长度不一定相等：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(通用寄存器)　　**指令编码长度(指令寄存器)**

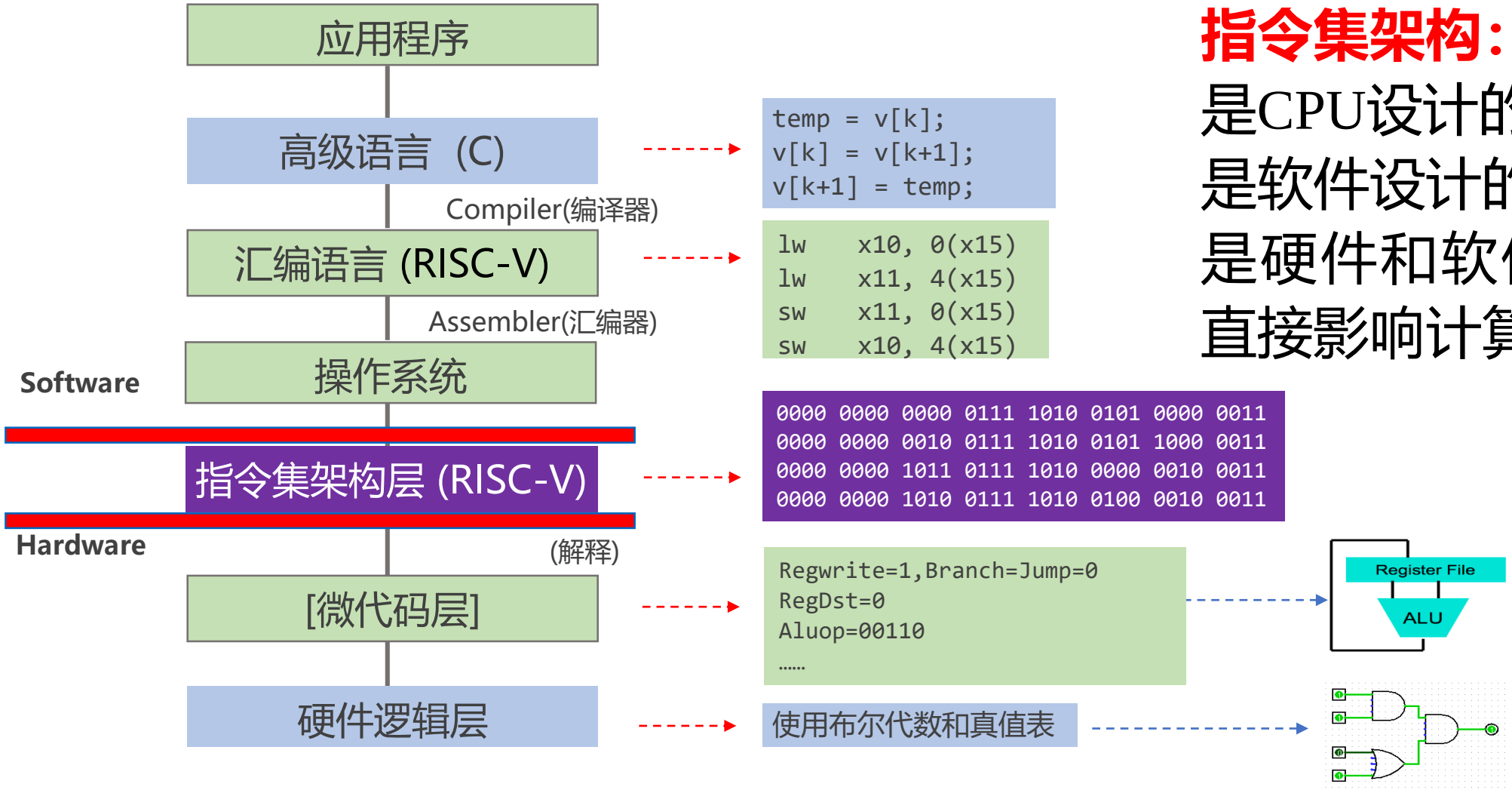
越大越好

寻址范围更大
运算能力更强

越短越好

节省代码存储空间

指令集架构层次



指令集架构：

是CPU设计的依据、

是软件设计的基础、

是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**
  - 简单性来自规整（Simplicity favors regularity）
    - 指令越规整，设计越简单
  - 越小越快 （Smaller is faster）
  - 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）

指令系统的评价——续

- **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
- **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
  - 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
- 操作数的类型与大小
- 所支持的操作
- 控制转移类指令
- 指令格式

存储器寻址

- 1980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）
- 如何读多个字节，例如：32位字
  - 每次一个字节，四次完成；每次一个字，一次完成
- 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**)

多选题 2分

设置

例如：设地址addr存储的字为 0x89ABCDEF，地址addr+1的字节存放的数据是？

A 89

B AB

C CD

D EF

E 以上都不对

提交

尾端问题 （小端little endian **vs** 大端big endian）

在一个（双）字内部的字节顺序问题

高　　低

例如：设地址addr存储的字为 0x89ABCDEF，addr，addr+1，addr+2，addr+3四个字节存放的分别是什么数据？

地址

	addr+3	addr+2	addr+1	addr
大端(字地址为高字节地址)	**EF**	**CD**	**AB**	**89**
小端(字地址为低字节地址)	**89**	**AB**	**CD**	**EF**

大端(MIPS, IBM360...)

小端(Intel 80x86, RISC V...)

口诀：小端地址低对低

大端地址低对高

对齐问题

- 假设对s个字节长的对象访问地址为A，如果A mod 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）
- RISC-V和x86没有对齐要求，但是MIPS有对齐要求
  - 即要求字的起始地址必须是4的倍数，而双字的起始地址必须是8的倍数。**对齐限制会使数据传输更快**
  - **黑书教材指定一个字的长度是32位**

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：**通过指令中的操作数（不同方式）计算出地址
- **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

寄存器寻址

立即数寻址

寄存器间接寻址

带偏移量的间接寻址

绝对寻址

相对基址变址寻址

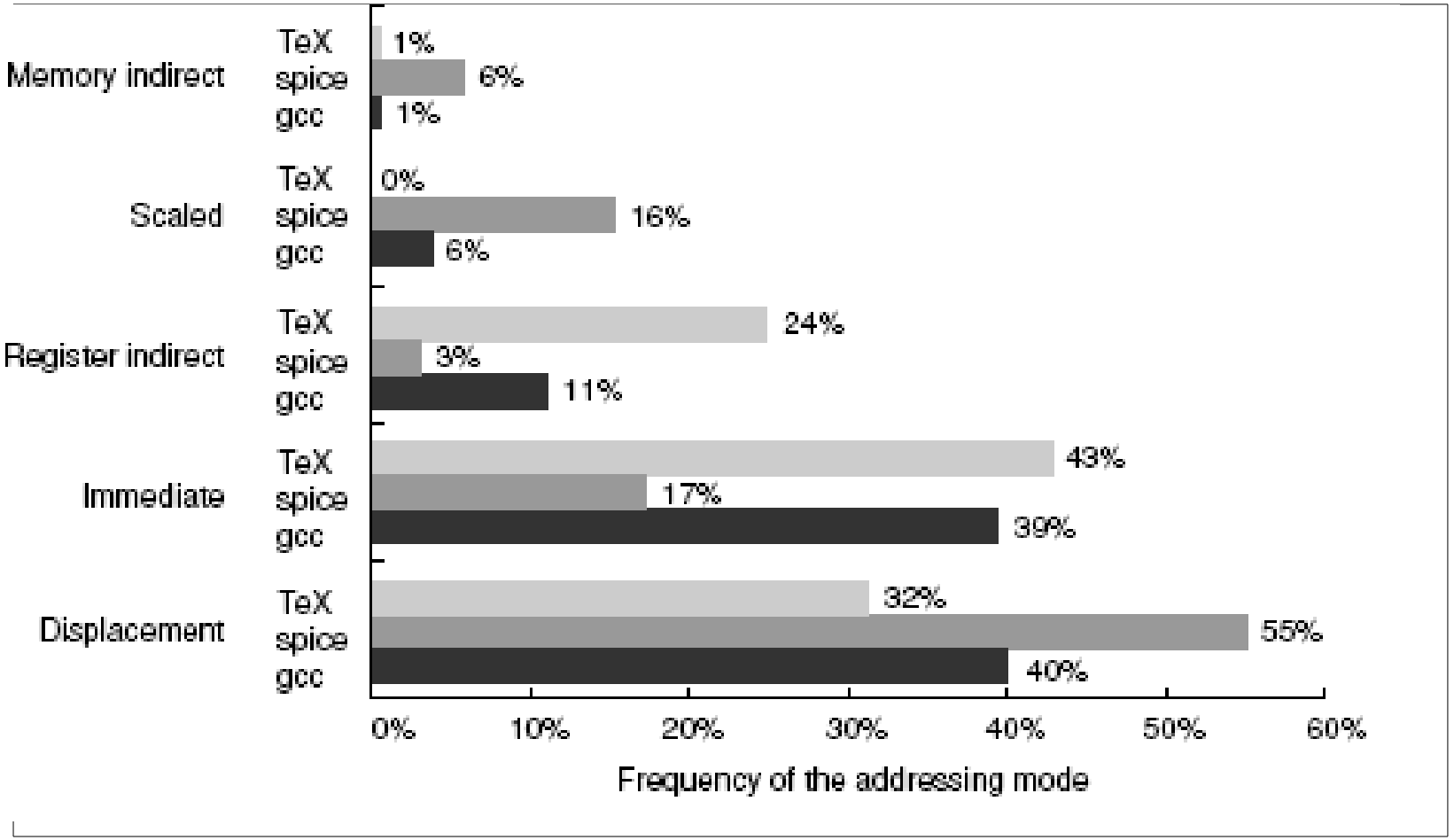
比例变址寻址

后增寄存器间接寻址

前增寄存器间接寻址

Mode	Example	Meaning	When used
Register	Add R1, R2	R1 ← R1 + R2	Values in registers
Immediate	Add R1, 100	R1 ← R1 + 100	For constants
Register Indirect	Add R1, (R2)	R1 ← R1 + Mem(R2)	R2 contains address
Displacement	Add R1, (R2+16)	R1 ← R1 + Mem(R2+16)	Address local variables
Absolute	Add R1, (1000)	R1 ← R1 + Mem(1000)	Address static data
Indexed	Add R1, (R2+R3)	R1 ← R1 + Mem(R2+R3)	R2=base, R3=index
Scaled Index	Add R1, (R2+s*R3)	R1 ← R1 + Mem(R2 + s*R3)	s = scale factor = 2, 4, or 8
Post-increment	Add R1, (R2)+	R1 ← R1 + Mem(R2) R2 ← R2 + s	Stepping through array s = element size
Pre-decrement	Add R1, -(R2)	R2 ← R2 – s R1 ← R1 + Mem(R2)	Stepping through array s = element size

各种寻址方式的使用频率



三个SPEC89程序(详见黑书1.9)在VAX结构上的测试结果：

立即寻址，偏移寻址（带偏移量的间接寻址）使用较多

有关ISA的若干问题

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

操作数的类型、表示

- **操作数类型**：面向应用、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（现已不用）**
- 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、移码
  - 浮点：IEEE 754标准
  - 十进制：一般用二进制描述

常用操作数类型

- ASCII character = 1 byte (64位寄存器能存8个ASCII字符）
- Unicode character or Short integer = 2 bytes = 16 bits （half word)
- **32位=4字节（字）**
  - Integer (很多RISC处理器上字的大小)、Single-precision float(单精度浮点)、**long int(Windows)**
- **64位=8字节（双字）**
  - Long long int、Double-precision float(双精度浮点)、pointer(指针)、**long int(linux)**
- Extended-precision float = 10 bytes = 80 bits (Intel architecture)
- Quad-precision float(四精度浮点) = 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：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

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(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**

国内主流指令集

- MIPS：龙芯等
  - Microprocessor without Interlocked Piped Stages
  - 无内部互锁流水级的微处理器
- X86：兆芯（VIA），海光（AMD）等
- ARM：华为，飞腾，海思，展讯，松果等
  - 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**A**dvanced **R**ISC **M**achine。
- 自主指令：申威 (Alpha指令集扩展)、龙芯LoongArch等
- RISC-V：阿里-平头哥，华米科技等

典型应用场景

服务器

- 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）

桌面个人计算机

- Intel或AMD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

嵌入式移动设备

ARM Cortex-A架构占垄断地位

嵌入式实时设备

深嵌入式

ARM 架构占最大份额，其他RISC架构嵌入式CPU也有应用

x86

- 1978年Intel发布了16位微处理器“8086”，x86架构诞生
  - 高性能
  - 扩展能力强
  - 操作系统的兼容性
  - 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
- 有着更高的性能、更快的速度和兼容性

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**; 它们积极的卓有成效的微结构实现; 以及他们的**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（因为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。
- 80x86是一个专有指令集（不开源）

MIPS

- MIPS是最典型的RISC 指令集架构
  - Stanford， 1980年提出，主要受到IBM801 小型机的影响
  - 第一个商业实现是R2000（1986）
  - 最初的设计中，其整数指令集仅有**58条指令**，直接实现单发射、顺序流水线
  - 30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
- **更适合于教学，相比X86更加简洁雅致，不会陷入繁琐的细节**

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对位置无关的代码（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

X86与MIPS的差异

	X86	MIPS
1	变长指令（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，实现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
- SPARC V8 主要特征
  - 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与其他80年代RISC结构的许多有缺陷的特性
  - 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
  - Alpha架构师仔细地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。为了在广泛使用这些操作的应用程序上性能，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。这个决定可能是可以单独接受的，但是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，他们选择逐步淘汰Alpha，转而采用英特尔的安腾架构。康柏将Alpha的知识产权出售给了英特尔，此后不久，惠普收购了康柏，并在2004年完成了Alpha的最终实现

ARMv7

- 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所称的架构许可的人
  - 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附带一个压缩的具有固定宽度16位的指令集，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，增加了能耗、延迟和设计成本。
- 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，几乎任何指令都可以改变控制流。
  - 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的缺点

- 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
- 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。

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（CPI）是RISC思想的精华**

精减指令系统(RISC)

- 指令条数少，保留使用频率最高的简单指令，指令定长
  - 便于硬件实现，用软件实现复杂指令功能
- 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（而CISC采用微程序控制）
- 对于单周期模型来说，一个时钟周期完成一条机器指令

硬布线和微程序控制器对比

- **硬布线控制器的特点**：优点是由于控制器的速度取决于电路延迟所以速度快，缺点是由于将控制部件看作专门产生固定时序控制信号的逻辑电路，所以把用最少元件和取得最高速度作为目标，一旦设计完成，不可能通过其他额外修改添加新功能。
- **微程序控制器的特点**：具有规整性、灵活性、可维护性等优点；但由于采用了存储程序原理，所以每条指令都要从控制存储器中取一次，影响了速度。

类别 / 对比项目	微程序控制器	硬布线控制器
工作原理	微操作控制信号以微程序的形式存放在控制存储器中，执行指令时读出即可	微操作控制信号由组合逻辑电路根据当前的指令码、状态和时序，即时产生
执行速度	慢	快
规整性	较规整	烦琐、不规整
应用场合	CISC CPU	RISC CPU
易扩充性	易扩充修改	困难

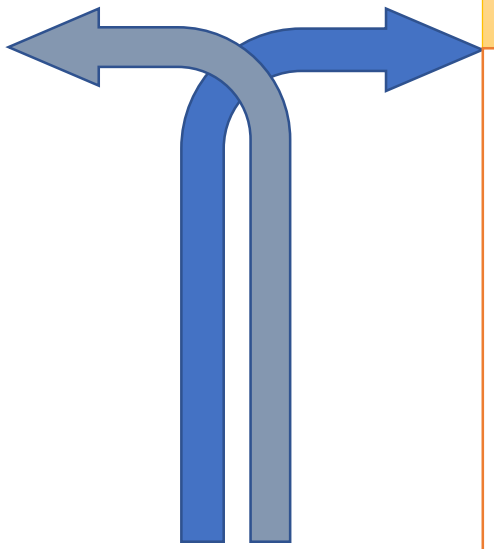
OpenRISC

- 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。

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



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

采用RISC体系结构的微处理器

- SUN Microsystem: SPARC, SuperSPARC, Ulta 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

国产芯片采用的指令集

- MIPS阵营
  - 龙芯

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。

- X86阵营
  - 兆芯（VIA），海光（AMD）

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，也标识一套通用的计算机指令集合。

- 自主指令
  - 申威 (Alpha指令集扩展)
- ARM阵营
  - 飞腾，海思，展讯，松果

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。

- RISC-V阵营
  - 阿里-平头哥，华米科技

RISC-V(读作“RISC-FIVE”)是基于精简指令集计算(RISC)原理建立的开放指令集架构(ISA)，V表示为第五代RISC(精简指令集计算机),表示此前已经四代RISC处理器原型芯片。每一代RISC处理器都是在同一人带领下完成，那就是加州大学伯克利分校的David A. Patterson教授。

问题

RISC的指令系统精简了，CISC中的一条指令实现的功能可能由多条RISC指令才能完成，那么为什么RISC执行程序的速度比CISC还要**快**？（执行时间=IC*CPI*t）

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

IC：指令条数，实际统计：RISC的IC只比CISC多30%~40%;

CPI: CISC中一般在为4~6，RISC中一般为1，Load/Store 为2;

t: 时钟周期

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

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。
  - "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
  - 32bit, 64bit, 128bit
- 规范由RISC-V非营利性基金会维护
  - RISC-V基金会负责维护RSIC-V指令集标准手册与架构文档

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。
- 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、VAX、Hewlett Packard PA-RISC、Intel i860、Motorola 88000、以及Zilog Z8000。
- **完全开源**

RISC-V架构的特点

- 指令集架构简单
  - 指令集有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。

RISC-V架构的特点——续

- 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

- 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
- 最基本也是唯一强制要求实现的指令集部分是由字母I表示的基本整数指令子集，使用该子集，便能够实现完整的软件编译器
- 其他子集均为可选模块，代表性模块有M/A/F/D/C等
- 预留了大量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，用户可以基于此直接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
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

可配置的寄存器组

- 通用寄存器（GPR：General Purpose Registers）
  - **32位**架构(RV**32**I)：32个**32位**的通用寄存器；
  - **64位**架构(RV**64**I)：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（重要）

- 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

注意：部分指令类型的立即数存在拼接与隐藏最低位0的特点。

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**

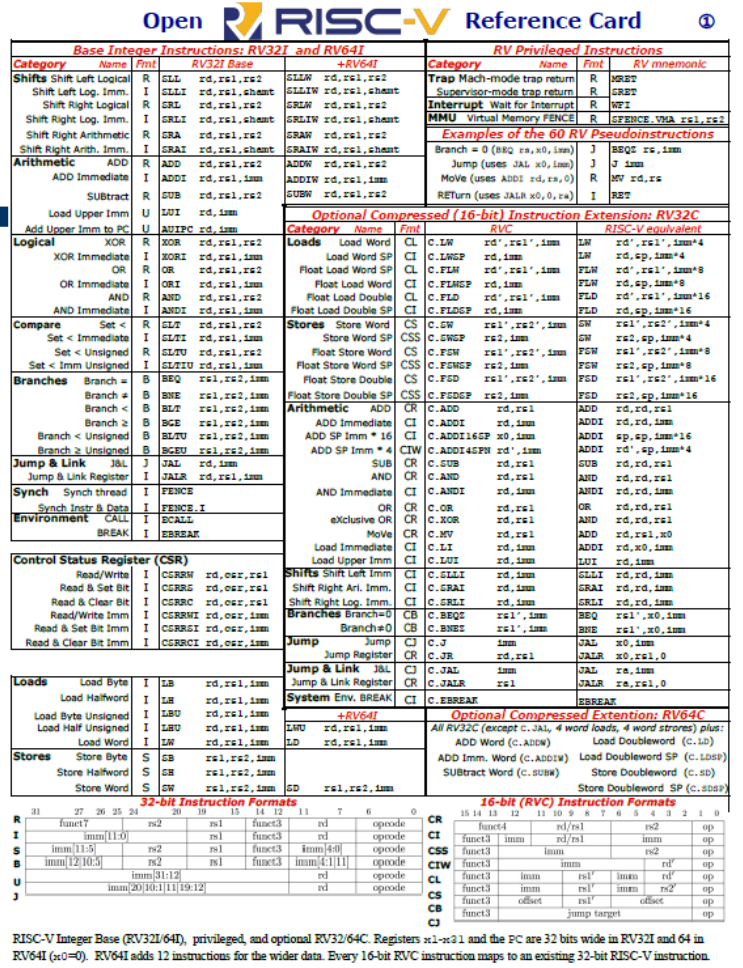
RISC-V 的指令集

- RISC-V官方指令集手册

https://riscv.org/specifications/isa-spec-pdf/

- 中文简化版

http://riscvbook.com/chinese/RISC-V-Reader-Chinese-v2p1.pdf


Open RISC-V Reference Card

Base Integer Instructions: RV32I and RV64I

Category	Name	Fmt	RV32I Base	+RV64I
Shifts	Shift Left Logical	R	SLL rd,rs1,rs2	SLLW rd,rs1,rs2
	Shift Left Log. Imm.	I	SLLI rd,rs1,shamt	SLLIW rd,rs1,shamt
	Shift Right Logical	R	SRL rd,rs1,rs2	SRLW rd,rs1,rs2
	Shift Right Log. Imm.	I	SRLI rd,rs1,shamt	SRLIW rd,rs1,shamt
	Shift Right Arithmetic	R	SRA rd,rs1,rs2	SRAW rd,rs1,rs2
	Shift Right Arith. Imm.	I	SRAI rd,rs1,shamt	SRAIW rd,rs1,shamt
Arithmetic	ADD	R	ADD rd,rs1,rs2	ADDW rd,rs1,rs2
	ADD Immediate	I	ADDI rd,rs1,imm	ADDIW rd,rs1,imm
	SUBtract	R	SUB rd,rs1,rs2	SUBW rd,rs1,rs2
	Load Upper Imm	U	LUI rd,imm	
	Add Upper Imm to PC	U	AUIPC rd,imm	
Logical	XOR	R	XOR rd,rs1,rs2	
	XOR Immediate	I	XORI rd,rs1,imm	
	OR	R	OR rd,rs1,rs2	
	OR Immediate	I	ORI rd,rs1,imm	
	AND	R	AND rd,rs1,rs2	
	AND Immediate	I	ANDI rd,rs1,imm	
Compare	Set <	R	SLT rd,rs1,rs2	
	Set < Immediate	I	SLTI rd,rs1,imm	
	Set < Unsigned	R	SLTU rd,rs1,rs2	
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm	
Branches	Branch =	B	BEQ rs1,rs2,imm	
	Branch ≠	B	BNE rs1,rs2,imm	
	Branch <	B	BLT rs1,rs2,imm	
	Branch ≥	B	BGE rs1,rs2,imm	
	Branch < Unsigned	B	BLTU rs1,rs2,imm	
	Branch ≥ Unsigned	B	BGEU rs1,rs2,imm	
Jump & Link	J&L	J	JAL rd,imm	
	Jump & Link Register	I	JALR rd,rs1,imm	
Synch	Synch thread	I	FENCE	
	Synch Instr & Data	I	FENCE.I	
Environment	CALL	I	ECALL	
	BREAK	I	EBREAK	

Control Status Register (CSR)

	Read/Write	I	CSRRW rd,csr,rs1	
	Read & Set Bit	I	CSRRS rd,csr,rs1	
	Read & Clear Bit	I	CSRRC rd,csr,rs1	
	Read/Write Imm	I	CSRRWI rd,csr,imm	
	Read & Set Bit Imm	I	CSRRSI rd,csr,imm	
	Read & Clear Bit Imm	I	CSRRCI rd,csr,imm	

Loads	Load Byte	I	LB rd,rs1,imm	
	Load Halfword	I	LH rd,rs1,imm	
	Load Byte Unsigned	I	LBU rd,rs1,imm	
	Load Half Unsigned	I	LHU rd,rs1,imm	
	Load Word	I	LW rd,rs1,imm	
Stores	Store Byte	S	SB rs1,rs2,imm	
	Store Halfword	S	SH rs1,rs2,imm	
	Store Word	S	SW rs1,rs2,imm	SD rs1,rs2,imm

RV Privileged Instructions

Category	Name	Fmt	RV mnemonic
Trap	Mach-mode trap return	R	MRET
	Supervisor-mode trap return	R	SRET
Interrupt	Wait for Interrupt	R	WFI
MMU	Virtual Memory FENCE	R	SFENCE.VMA rs1,rs2

Examples of the 60 RV Pseudoinstructions

Branch = 0 (BEQ rs,x0,imm)		J	BEQZ rs,imm
Jump (uses JAL x0,imm)		J	J imm
MoVe (uses ADDI rd,rs,0)		R	MV rd,rs
RETurn (uses JALR x0,0,ra)		I	RET

Optional Compressed (16-bit) Instruction Extension: RV32C

Category	Name	Fmt	RVC	RISC-V equivalent
Loads	Load Word	CL	C.LW rd',rs1',imm	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	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	FLD rd',rs1',imm*16
	Float Load Double SP	CI	C.FLDSP rd,imm	FLD rd,sp,imm*16
Stores	Store Word	CS	C.SW rs1',rs2',imm	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	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
Arithmetic	ADD	CR	C.ADD rd,rs1	ADD rd,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	XOR 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
Shifts	Shift Left Imm	CI	C.SLLI rd,imm	SLLI rd,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
Branches	Branch=0	CB	C.BEQZ rs1',imm	BEQ rs1',x0,imm
	Branch≠0	CB	C.BNEZ rs1',imm	BNE rs1',x0,imm
Jump	Jump	CJ	C.J imm	JAL x0,imm
	Jump Register	CR	C.JR rd,rs1	JALR x0,rs1,0
Jump & Link	J&L	CJ	C.JAL imm	JAL ra,imm
	Jump & Link Register	CR	C.JALR rs1	JALR ra,rs1,0
System	Env. BREAK	CI	C.EBREAK	EBREAK

+RV64I

LWU	rd,rs1,imm		
LD	rd,rs1,imm		

Optional Compressed Extention: RV64C

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)

32-bit Instruction Formats

16-bit (RVC) Instruction Formats

RISC-V Integer Base (RV32I/64I), privileged, and optional RV32/64C. 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

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
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

小结

- **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

附加解释

- 这部分补充内容等大家学完流水线再去阅读就很清晰

可配置的通用寄存器组

RISC-V架构支持32位或者64位的架构，32位架构由RV32表示，其每个通用寄存器的宽度为32比特；64位架构由RV64表示，其每个通用寄存器的宽度为64比特。RISC-V架构的整数通用寄存器组，包含32个（I架构）或者16个（E架构）通用整数寄存器，其中整数寄存器0被预留为常数0，其他的31个（I架构）或者15个（E架构）为普通的通用整数寄存器。

如果使用浮点模块（F或者D），则需要另外一个独立的浮点寄存器组，包含32个通用浮点寄存器。如果仅使用F模块的浮点指令子集，则每个通用浮点寄存器的宽度为32比特；如果使用了D模块的浮点指令子集，则每个通用浮点寄存器的宽度为64比特。

规整的指令编码

在流水线中能够尽快地读取通用寄存器组，往往是处理器流水线设计的期望之一，这样可以提高处理器性能和优化时序。这个看似简单的道理在很多现存的商用RISC架构中都难以实现，因为经过多年反复修改不断添加新指令后，其指令编码中的寄存器索引位置变得非常凌乱，给译码器造成了负担。

得益于后发优势和总结了多年来处理器发展的经验，RISC-V的指令集编码非常规整，指令所需的通用寄存器的索引（Index）都被放在固定的位置。因此指令译码器（Instruction Decoder）可以非常便捷地译码出寄存器索引，然后读取通用寄存器组（Register File，Regfile）。

简洁的存储器访问指令

与所有的RISC处理器架构一样，RISC-V架构使用专用的存储器读（Load）指令和存储器写（Store）指令访问存储器（Memory），其他的普通指令无法访问存储器，这种架构是RISC架构常用的一个基本策略。这种策略使得处理器核的硬件设计变得简单。存储器访问的基本单位是字节（Byte）。RISC-V的存储器读和存储器写指令支持一个字节（8位）、半字（16位）、单字（32位）为单位的存储器读写操作。如果是64位架构还可以支持一个双字（64位）为单位的存储器读写操作。

RISC-V架构的存储器访问指令还有如下显著特点。

为了提高存储器读写的性能，RISC-V架构推荐使用地址对齐的存储器读写操作，但是也支持地址非对齐的存储器操作RISC-V架构。处理器既可以选择用硬件来支持，也可以选择用软件来支持。

由于现在的主流应用是小端格式（Little-Endian），RISC-V架构仅支持小端格式。有关小端格式和大端格式的定义和区别，在此不做过多介绍。若对此不太了解的初学者可以自行查阅学习。

很多的RISC处理器都支持地址自增或者自减模式，这种自增或者自减的模式虽然能够提高处理器访问连续存储器地址区间的性能，但是也增加了设计处理器的难度。RISC-V架构的存储器读和存储器写指令不支持地址自增自减的模式。

RISC-V架构采用松散存储器模型（Relaxed Memory Model），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不作要求，除非使用明确的存储器屏障（Fence）指令加以屏蔽。有关存储器模型（Memory Model）和存储器屏障指令的更多信息，请参见附录A13。

这些选择都清楚地反映了RISC-V架构力图简化基本指令集，从而简化硬件设计的哲学。RISC-V架构如此定义是具有合理性的，能达到能屈能伸的效果。例如，对于低功耗的简单CPU，可以使用非常简单的硬件电路即可完成设计；而对于追求高性能的超标量处理器，则可以通过复杂设计的动态硬件调度能力来提高性能。

高效的分支跳转指令

RISC-V架构有两条无条件跳转指令（Unconditional Jump），即jal指令与jalr指令。跳转链接（Jump and Link）指令——jal指令可用于进行子程序调用，同时将子程序返回地址存在链接寄存器（Link Register，由某一个通用整数寄存器担任）中。跳转链接寄存器（Jump and Link-Register）指令——jalr指令能够用于子程序返回指令，通过将jal指令（跳转进入子程序）保存的链接寄存器用于jalr指令的基地址寄存器，则可以从子程序返回。

RISC-V架构有6条带条件跳转指令（Conditional Branch），这种带条件的跳转指令跟普通的运算指令一样直接使用两个整数操作数，然后对其进行比较。如果比较的条件满足，则进行跳转，因此此类指令将比较与跳转两个操作放在一条指令里完成。作为比较，很多其他的RISC架构的处理器需要使用两条独立的指令。***条指令先使用比较指令，比较的结果被保存到状态寄存器之中；第二条指令使用跳转指令，判断前一条指令保存在状态寄存器当中的比较结果为真时，则进行跳转。相比而言，RISC-V的这种带条件跳转指令不仅减少了指令的条数，同时硬件设计上更加简单。

对于没有配备硬件分支预测器的低端CPU，为了保证其性能，RISC-V的架构明确要求采用默认的静态分支预测机制，即如果是向后跳转的条件跳转指令，则预测为“跳”；如果是向前跳转的条件跳转指令，则预测为“不跳”，并且RISC-V架构要求编译器也按照这种默认的静态分支预测机制来编译生成汇编代码，从而让低端的CPU也得到不错的性能。
在低端的CPU中，为了使硬件设计尽量简单，RISC-V架构特地定义了所有的带条件跳转指令跳转目标的偏移量（相对于当前指令的地址）都是有符号数，并且其符号位被编码在固定的位置。因此这种静态预测机制在硬件上非常容易实现，硬件译码器可以轻松地找到固定的位置，判断该位置的比特值为1，表示负数（反之则为正数）。根据静态分支预测机制，如果是负数，则表示跳转的目标地址为当前地址减去偏移量，也就是向后跳转，则预测为“跳”。当然，对于配备有硬件分支预测器的高端CPU，则还可以采用高级的动态分支预测机制来保证性能。

简洁的子程序调用

为了理解此节，需先对一般RISC架构中程序调用子函数的过程予以介绍，其过程如下。

进入子函数之后需要用存储器写（Store）指令来将当前的上下文（通用寄存器等的值）保存到系统存储器的堆栈区内，这个过程通常称为“保存现场”。

在退出子程序时，需要用存储器读（Load）指令来将之前保存的上下文（通用寄存器等的值）从系统存储器的堆栈区读出来，这个过程通常称为“恢复现场”。

“保存现场”和“恢复现场”的过程通常由编译器编译生成的指令完成，使用高层语言（例如C语言或者C++）开发的开发者对此可以不用太关心。高层语言的程序中直接写上一个子函数调用即可，但是这个底层发生的“保存现场”和“恢复现场”的过程却是实实在在地发生着（可以从编译出的汇编语言里面看到那些“保存现场”和“恢复现场”的汇编指令），并且还需要消耗若干的CPU执行时间。

为了加速“保存现场”和“恢复现场”的过程，有的RISC架构发明了一次写多个寄存器到存储器中（Store Multiple），或者一次从存储器中读多个寄存器出来（Load Multiple）的指令。此类指令的好处是一条指令就可以完成很多事情，从而减少汇编指令的代码量，节省代码的空间大小。但是“一次读多个寄存器指令”和“一次写多个寄存器指令”的弊端是会让CPU的硬件设计变得复杂，增加硬件的开销，也可能损伤时序，使得CPU的主频无法提高，作者曾经设计此类处理器时便深受其苦。

RISC-V架构则放弃使用“一次读多个寄存器指令”和“一次写多个寄存器指令”。如果有的场合比较介意“保存现场”和“恢复现场”的指令条数，那么可以使用公用的程序库（专门用于保存和恢复现场）来进行，这样就可以省掉在每个子函数调用的过程中都放置数目不等的“保存现场”和“恢复现场”的指令。此选择再次印证了RISC-V追求硬件简单的哲学，因为放弃“一次读多个寄存器指令”和“一次写多个寄存器指令”可以大幅简化CPU的硬件设计，对于低功耗小面积的CPU可以选择非常简单的电路进行实现；而高性能超标量处理器由于硬件动态调度能力很强，可以有强大的分支预测电路保证CPU能够快速地跳转执行，从而可以选择使用公用的程序库（专门用于保存和恢复现场）的方式减少代码量，同时达到高性能。

无条件码执行

很多早期的RISC架构发明了带条件码的指令，例如在指令编码的头几位表示的是条件码（Conditional Code），只有该条件码对应的条件为真时，该指令才被真正执行。

这种将条件码编码到指令中的形式可以使编译器将短小的循环编译成带条件码的指令，而不用编译成分支跳转指令。这样便减少了分支跳转的出现，一方面减少了指令的数目；另一方面也避免了分支跳转带来的性能损失。然而，这种“条件码”指令的弊端同样会使CPU的硬件设计变得复杂，增加硬件的开销，也可能损伤时序使得CPU的主频无法提高。

RISC-V架构则放弃使用这种带“条件码”指令的方式，对于任何的条件判断都使用普通的带条件分支跳转指令。此选择再次印证了RISC-V追求硬件简单的哲学，因为放弃带“条件码”指令的方式可以大幅简化CPU的硬件设计，对于低功耗小面积的CPU可以选择非常简单的电路进行实现，而高性能超标量处理器由于硬件动态调度能力很强，可以有强大的分支预测电路保证CPU能够快速地跳转执行达到高性能。

无分支延迟槽

很多早期的RISC架构均使用了“分支延迟槽（Delay Slot）”，具有代表性的便是MIPS架构。在很多经典的计算机体系结构教材中，均使用MIPS对分支延迟槽进行介绍。分支延迟槽就是指在每一条分支指令后面紧跟的一条或者若干条指令不受分支跳转的影响，不管分支是否跳转，这后面的几条指令都一定会被执行。

早期的RISC架构很多采用了分支延迟槽诞生的原因主要是当时的处理器流水线比较简单，没有使用高级的硬件动态分支预测器，使用分支延迟槽能够取得可观的性能效果。然而，这种分支延迟槽使得CPU的硬件设计变得极为别扭，CPU设计人员对此苦不堪言。

RISC-V架构则放弃了分支延迟槽，再次印证了RISC-V力图简化硬件的哲学，因为现代的高性能处理器的分支预测算法精度已经非常高，可以有强大的分支预测电路保证CPU能够准确地预测跳转执行达到高性能。而对于低功耗、小面积的CPU，由于无须支持分支延迟槽，硬件得到极大简化，也能进一步减少功耗和提高时序。