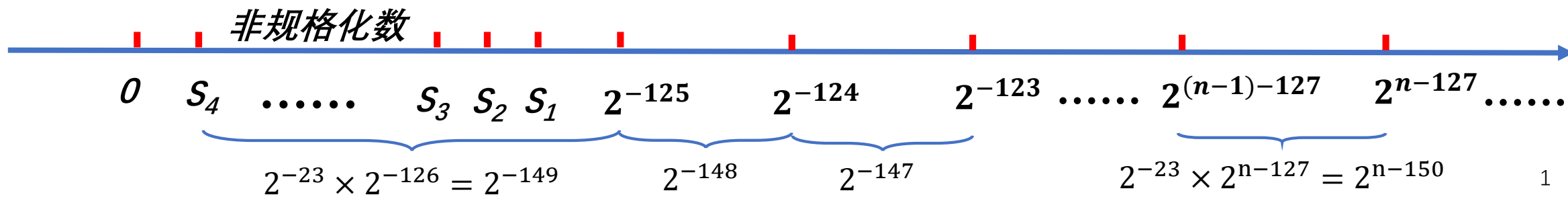


对IEEE 754非规格化数的进一步说明

- IEEE 754定义非规格化数的指数为-126，主要是考虑到保持 $[2^{-149}, 2^{-125})$ 区间（非规格化数和指数为1的规格化数）内两浮点数之间具有统一的最小分度值 2^{-149} ，该区间内表示的数值较小，使用IEEE 754单精度浮点数能表示的最小分度值 2^{-149} ，有利于提高数的表示精度。
- 对于不同的浮点数区间，IEEE 754采用不同的分度值，主要是基于分度值对数值的影响程度。例如：当数值在 $[0, 0.1]$ 之间时，0.01对数值影响很大，需要使用小分度值；当数值在 $[10000, 10001]$ 之间时，0.01对数值几乎没有影响，可以使用更大的分度值，提高数的表示范围。
- 下面的数轴直观地说明了IEEE 754不同区间的分度值，其中 n 为指数，23为尾数的位数。



计算机组成原理



群名称:2022春-计组4班
群 号:139369553

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>

浮点表示

$N = S \times r^j$ 浮点数的一般形式

S 尾数 j 阶码 r 基数（基值）

计算机中 r 取 2、4、8、16 等

当 $r = 2$ $N = 11.0101$ 二进制表示
 $\checkmark = 0.110101 \times 2^{10}$ 规格化数
 $= 1.10101 \times 2^1$
 $= 1101.01 \times 2^{-10}$
 $\checkmark = 0.00110101 \times 2^{100}$

计算机中 S 小数、可正可负
 j 整数、可正可负

规格化

(1) 规格化数的定义

$$r = 2^{-\frac{1}{2}} \leq |S| < 1$$

(2) 规格化数的判断

$S > 0$	规格化形式	$S < 0$	规格化形式
真值	$0.1 \times \times \cdots \times$	真值	$-0.1 \times \times \cdots \times$
原码	0.1 $\times \times \cdots \times$	原码	1.1 $\times \times \cdots \times$
补码	0.1 $\times \times \cdots \times$	补码	1.0 $\times \times \cdots \times$
反码	$0.1 \times \times \cdots \times$	反码	$1.0 \times \times \cdots \times$

原码 不论正数、负数，第一数位为1

补码 符号位和第1数位不同

特例

(1) 规格化数的定义

$$r = 2 \quad \frac{1}{2} \leq |S| < 1$$

$$S = -\frac{1}{2} = -0.100 \cdots 0$$

$$[S]_{\text{原}} = 1.100 \cdots 0$$

$$[S]_{\text{补}} = \boxed{1.1}00 \cdots 0$$

$\therefore [-\frac{1}{2}]_{\text{补}}$ 不是规格化的数

$$S = -1$$

$$[S]_{\text{补}} = \boxed{1.0}00 \cdots 0$$

$\therefore [-1]_{\text{补}}$ 是规格化的数

第二章 RISC-V汇编及其指令系统

- 计算机中数的表示
- **RISC-V概述**
- RISC-V汇编语言
- RISC-V指令表示
- 案例分析



第二章 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

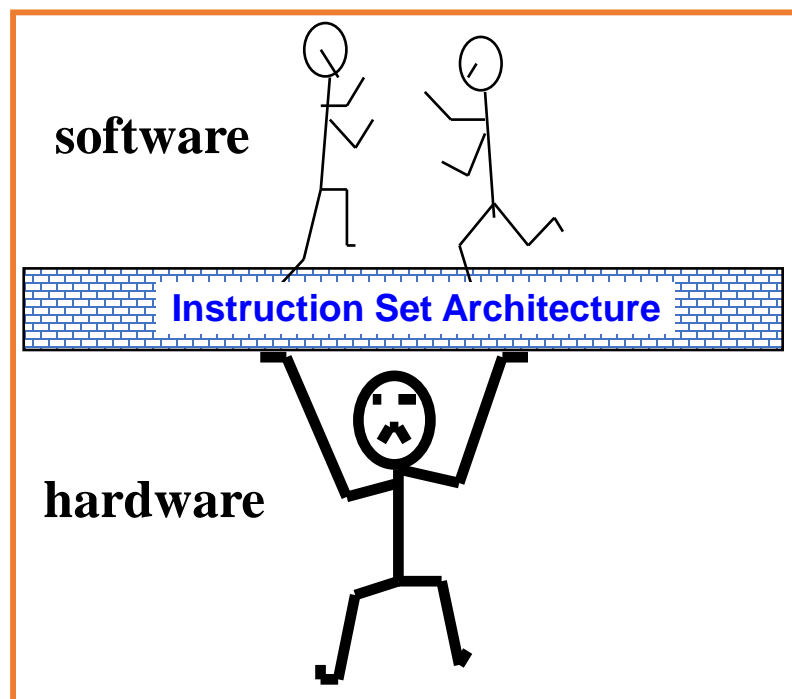
系统模型

- 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

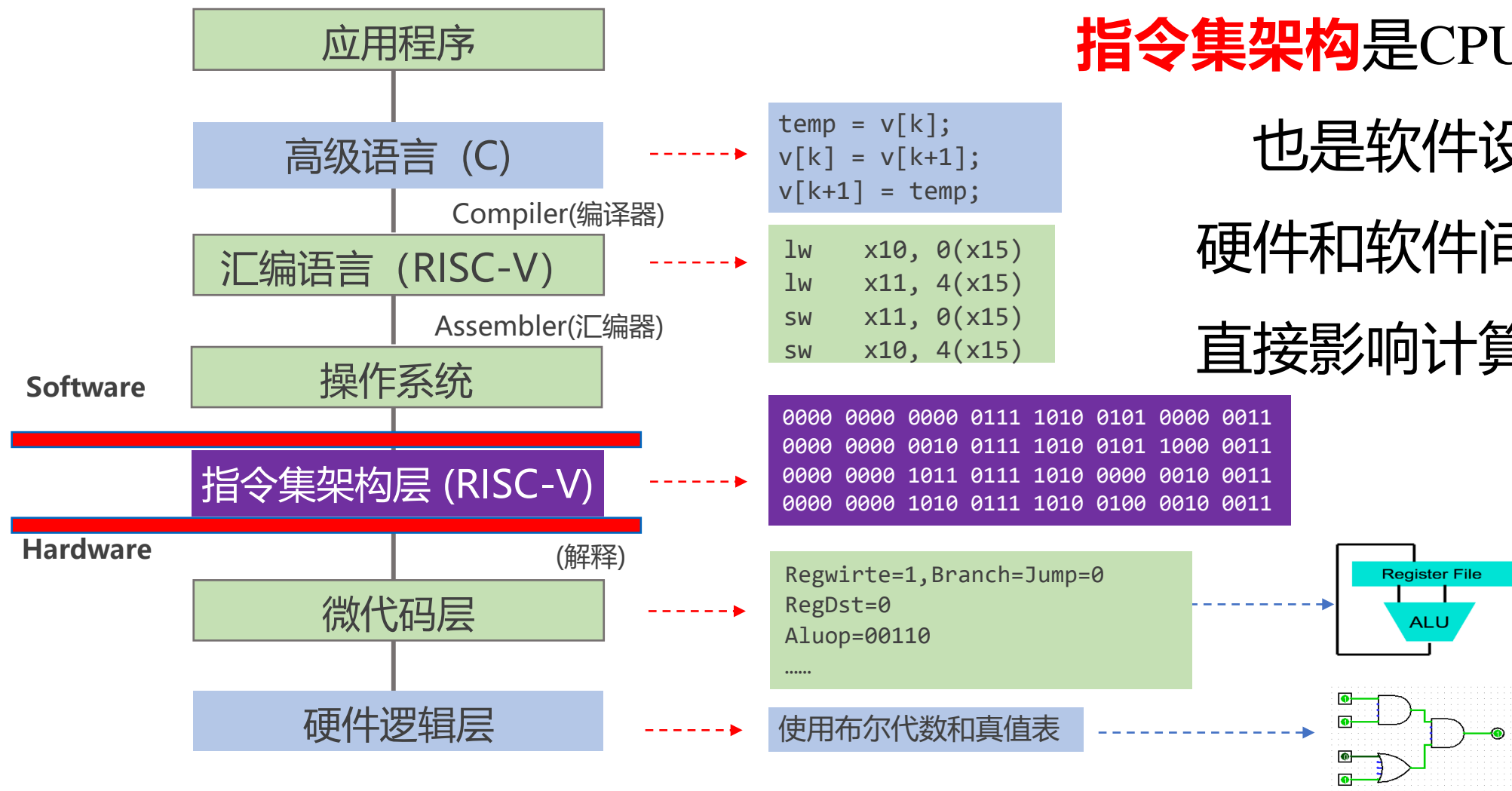
可以带来更大的寻址范围和更强的运算能力

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

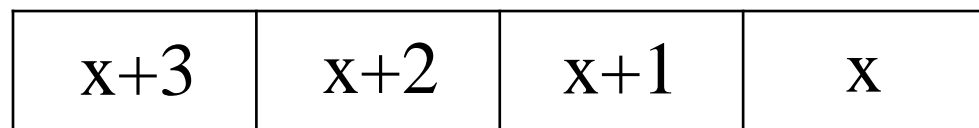


例：连续存放 fe dc 45 67

00000000 00000000 00000100 00000001

高字节放在低地址

大端

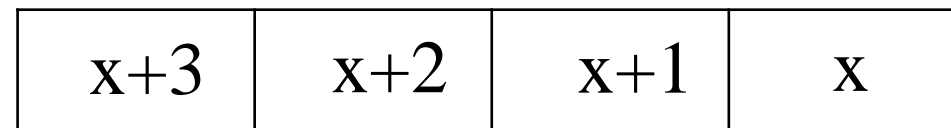


67 45 dc fe
00000001 00000100 00000000 00000000

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
- **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

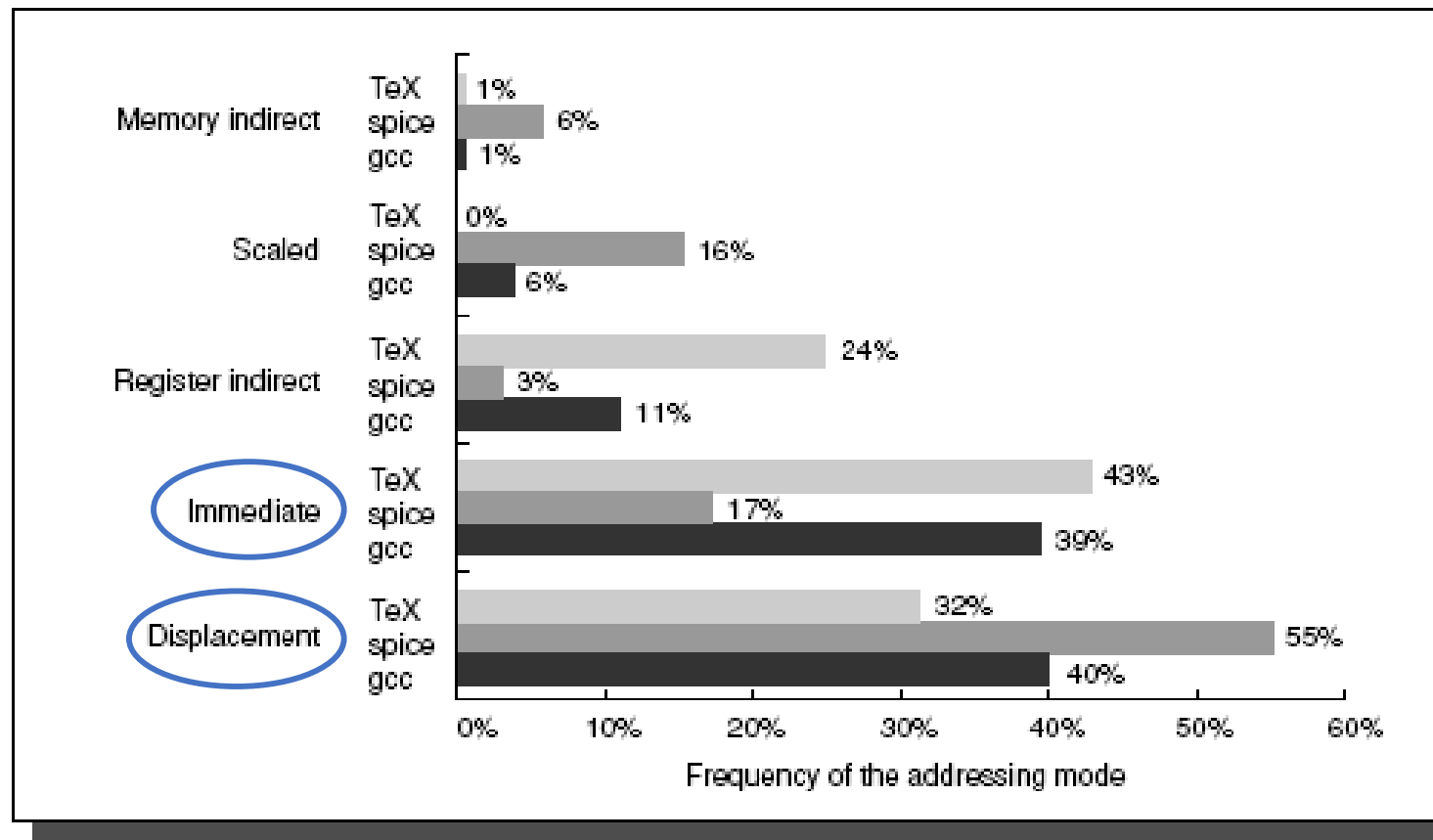
各种寻址方式的使用情况？

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

第二章 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

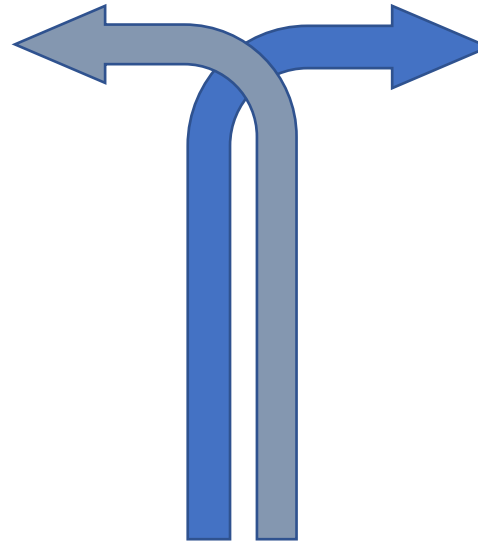


CISC

- 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：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

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)

苹果A14处理器
- 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。

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垄断地位
- ARM服务器已经进入该领域

ARM Cortex-A架构垄断地位

ARM 架构占最大份额，其他RISC架构嵌入式CPU也有应用

x86



- 1978年Intel发布了16位微处理器“8086”，x86架构的复杂指令集（CISC）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

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寄存器组（通用寄存器）的数量极少
- 大多数整数寄存器在ISA中执行特殊功能，加剧了体系结构寄存器的不足
- 更糟糕的是，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特性在微架构中的实现复杂，包括隐式条件代码和带有谓词的移动操作

尽管存在所有这些缺陷，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。

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处理器 (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

- 主要问题:

- 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，但保持了兼容性，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
- 16位位宽立即数消耗了大量编码空间，只有少量的编码空间可供扩展指令
 - 2014修订版，保存有1/64的编码空间供扩展
 - 架构师如果想采用压缩指令编码来降低代码空间，就不得不采用新的指令编码

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
- ISA假设浮点操作部件是一个独立的协处理器，使得单芯片实现无法最优
 - 例如，整型数与浮点数的转换结果写到浮点数寄存器，使用结果时，需要额外的mov指令，更糟糕的是浮点数寄存器文件与整型数寄存器文件之间的传输，需要有显式的延迟槽
- 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**

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，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结构相似的缺陷：
 - 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。
 - 实际上，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
- 最后，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，它不是一种产品。**RISC**是近代计算机体系结构发展史中的一个里程碑，直到现在，**RISC**没有一个确切的定义
- 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，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

- OpenRISC项目是一个开放源码处理器设计项目
 - 来源于Hennessy和Patterson的体系结构教科书。
 - 适用于教学、科研和工业界的实现。
- 主要问题
 - OpenRISC项目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。

ISA Summary

	MIPS	SPARC	Alpha	ARMv7	ARMv8	RISC-V	80x86
Free and Open		✓				✓	
64-bit Address	✓	✓	✓		✓	✓	✓
Compressed Instructions	✓			✓			Partial
Separate Privileged ISA			✓				
Position-Indep. Code	Partial			✓	✓		✓
IEEE 754-2008					✓		✓
Classically Virtualizable	✓	✓	✓		✓		

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
 - 两种类型：CISC和RISC
- 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指令集结构的功能设计

- 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

第二章 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
 - 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RISC-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。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的指令集使用模块化的方式进行组织，每一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

模块化的RISC-V 指令子集

- RISC-V的指令集使用模块化的方式进行组织，每一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

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

可配置的通用寄存器组

- 寄存器组主要包括通用寄存器（General Purpose Registers）和控制状态寄存器（Control and Status Registers）
 - 32位架构(RV32I)32个32位的通用寄存器，64位架构(RV64I)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
- CSR寄存器用于配置或记录一些运行的状态（后续异常和中断处理中会详细描述）
 - CSR寄存器是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，方便译码阶段的使用
- 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
 - 小端机结构

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																				
	Add Upper Imm to PC	U	AUIPC	rd,imm									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.ADDI4SPH	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	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															
Shifts	Shift Left Imm	CI	C.SLLI	rd,imm				SLLI	rd,rd,imm															
	Shift Right Arl. 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.BEQ	rs1',imm				BEQ	rs1',x0,imm															
	Branch≠0	CB	C.BNE	rs1',imm				BNE	rs1',x0,imm															
	Jump	CJ	C.J	imm				JAL	x0,imm															
Jump & Link	Jump Register	CR	C.JR	rd,rs1				JALR	x0,rs1,0															
	Jump & Link	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										Optional Compressed Extension: RV64C														
LWU rd,rs1,imm										All RV32C (except C.JAL, 4 word loads, 4 word stores) plus:														
LD rd,rs1,imm										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)														
SD 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	rs1				funct3	rd	opcode												
S	imm[11:0]				rs2				rs1	funct3	rd	opcode												
B	imm[11:5]				rs2	rs1				funct3	imm[4:0]	opcode												
U	imm[31:12]				rs2				rs1	funct3	imm[4:1]	opcode												
J	imm[20:10]				rs2				rs1	funct3	imm[19:12]	rd				opcode								
CR	funct4				rd/rs1				rs2				op											
CI	funct3				imm	rd/rs1				imm				op										
CSS	funct3				imm				rs2				op											
CIW	funct3				imm				rd'				op											
CL	funct3				imm	rs1'				imm	rd'				op									
CS	funct3				imm	rs1'				imm	rs2'				op									
CB	funct3				offset				rs1'				offset				op							
CJ	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.

总结



- 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