RISC-V 调研报告

简介

RISC-V（发音为“risk-five”）是基于已创建的精简指令集（RISC）原则的一个开源指令集架构（ISA）。

与大多数指令集相比，RISC-V 指令集可以自由地用于任何目的，允许任何人设计、制造和销售 RISC-V 芯片和软件。虽然这不是第一个开源指令集，但它具有重要意义，因为其设计使其适用于现代计算设备（如仓库规模云计算机、高端移动电话和微小嵌入式系统）。设计者考虑到了这些用途中的性能与功率效率。该指令集还具有众多支持的软件，这解决了新指令集通常的弱点。

该项目 2010 年始于加州大学柏克莱分校，但许多贡献者是该大学以外的志愿者和行业工作者。

RISC-V 指令集的设计考虑了小型、快速、低功耗的现实世界实现，但没有对特定微架构风格的过度架构。[1][2]

截至2017年5月，用户空间的指令集版本2.2已固定，特权指令集已处在草案版本1.10。

意义

RISC-V 的作者们旨在以 BSD 许可证自由地提供数种 CPU 设计。该许可证使 RISC-V 芯片设计等派生作品可以像 RISC-V 本身一样被公开且自由发行，或者被闭源或专有。

相比而言，ARM 控股和 MIPS 科技等商业芯片供应商会对使用其专利收取许可费用。[3]在接收其描述设计优点和指令集的文档前，还需要与其签署保密协议。许多设计优点为完全专有，从不会披露给客户。这种保密制度阻碍了公共教育用途和安全审计，以及开发公共、低成本的自由及开放源代码软件编译器和操作系统。

开发一个 CPU 需要多种专业的设计知识，包括电子逻辑、编译器和操作系统。这种资源在专业工程团队之外很难见到。其结果是，现代的高质量通用计算机指令集近年来未在任何地方被广泛使用，甚至没有被阐述，除了学术环境。正因如此，许多 RISC-V 贡献者将此视为一项一致的社区方向，这也是 RISC-V 被设计为适应很多种用途的一项原因。

RISC-V 的作者们还有大量研究和用户体验验证了他们在硅片和仿真中的设计。RISC-V 指令集从一系列学术计算机设计项目直接发展而来。它起源于帮助这些项目。

特色

1．完全开源

对指令集使用，RISC-V 基金会不收取高额的授权费。开源采用宽松的 BSD 协议，企业完全自有免费使用，同时也容许企业添加自有指令集拓展而不必开放共享以实现差异化发展。

2.架构报告

RISC-V 架构秉承简单的设计哲学。体现为：

在处理器领域，主流的架构为 x86 与 ARM 架构。x86 与 ARM 架构的发展的过程也伴随了现代处理器架构技术的不断发展成熟，但作为商用的架构，为了能够保持架构的向后兼容性，其不得不保留许多过时的定义，导致其指令数目多，指令冗余严重，文档数量庞大，所以要在这些架构上开发新的操作系统或者直接开发应用门槛很高。而 RISC-V 架构则能完全抛弃包袱，借助计算机体系结构经过多年的发展已经成为比较成熟的技术的优势，从轻上路。RISC-V 基础指令集则只有 40 多条，加上其他的模块化扩展指令总共几十条指令。 RISC-V 的规范文档仅有 145 页，而“特权架构文档”的篇幅也仅为 91 页

3.易于移植

现代操作系统都做了特权级指令和用户级指令的分离，特权指令只能操作系统调用，而用户级指令才能在用户模式调用，保障操作系统的稳定。RISC-V 提供了特权级指令和用户级指令，同时提供了详细的 RISC-V 特权级指令规范 [3] 和 RISC-V 用户级指令规范 [2] 的详细信息，使开发者能非常方便的移植 linux 和 unix 系统到 RISC-V 平台。

4.模块化设计

RISC-V 架构不仅短小精悍，而且其不同的部分还能以模块化的方式组织在一起，从而试图通过一套统一的架构满足各种不同的应用场景。用户能够灵活选择不同的模块组合，来实现自己定制化设备的需要，比如针对于小面积低功耗嵌入式场景，用户可以选择 RV32IC 组合的指令集，仅使用 Machine Mode（机器模式）；而高性能应用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。

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，如表 1 所示。

表 1 RISC-V 的模块化指令集

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	59	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
扩展指令集	指令数	描述
M	8	整数乘法与除法指令
A	11	存储器原子（Atomic）操作指令和 Load-Reserved/Store-Conditional 指令
F	26	单精度（32 比特）浮点指令
D	26	双精度（64 比特）浮点指令，必须支持 F 扩展指令
C	46	压缩指令，指令长度为 16 位

为了提高代码密度，RISC-V 架构也提供可选的“压缩”指令子集，由英文字母 C 表示。压缩指令的指令编码长度为 16 比特，而普通的非压缩指令的长度为 32 比特。以上这些模块的一个特定组合“IMAFD”，也被称为“通用”组合，由英文字母 G 表示。因此 RV32G 表示 RV32IMAFD，同理 RV64G 表示 RV64IMAFD。

为了进一步减少面积，RISC-V 架构还提供一种“嵌入式”架构，由英文字母 E 表示。该架构主要用于追求极低面积与功耗的深嵌入式场景。该架构仅需要支持 16 个通用整数寄存器，而非嵌入式的普通架构则需要支持 32 个通用整数寄存器。

通过以上的模块化指令集，能够选择不同的组合来满足不同的应用。譬如，追求小面积低功耗的嵌入式场景可以选择使用 RV32EC 架构；而大型的 64 位架构则可以选择 RV64G。

除了上述的模块，还有若干的模块包括 L、B、P、V 和 T 等。这些扩展目前大多数还在不断完善和定义中，尚未最终确定，因此本文在此不做详细论述。

5．完整的工具链

对于设计 CPU 来说，工具链是软件开发人员和 cpu 交互的窗口，没有工具链，对软件开发人员开发软件要求很高，甚至软件开发者无法让 cpu 工作起来。在 cpu 设计中，工具链的开发是一个需要巨大工作的。如果用 RISC-V 来设计芯片，芯片设计公司不再担心工具链问题，只需专注于芯片设计，RISC-V 社区已经提供了完整的工具链，并且 RISC-V 基金会持续维护该工具链。当前 RISC-V 的支持已经合并到主要的工具中，比如编译工具链 gcc，仿真工具 qemu 等

6．开源实现

……

7.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。

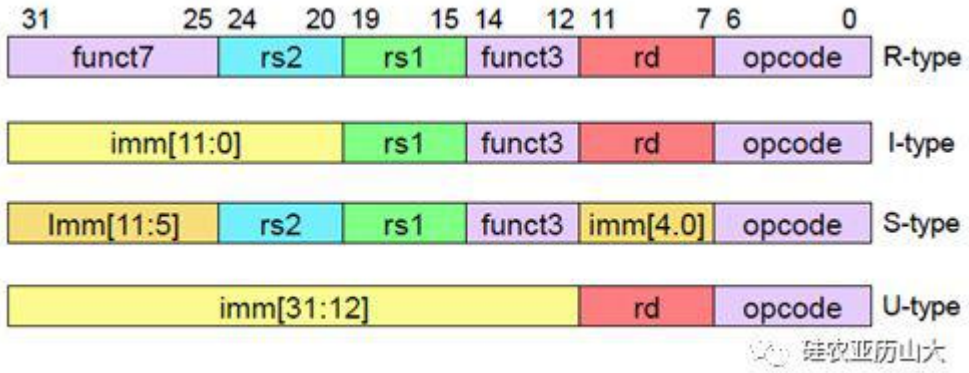
如果使用了浮点模块（F 或者 D），则需要另外一个独立的浮点寄存器组，包含 32 个通用浮点寄存器。如果仅使用 F 模块的浮点指令子集，则每个通用浮点寄存器的宽度为 32 比特；如果使用了 D 模块的浮点指令子集，则每个通用浮点寄存器的宽度为 64 比特。

8.规整的指令编码

在流水线中能够尽早尽快的读取通用寄存器组，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的凌乱，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教训，RISC-V 的指令集编码非常的规整，指令所需的通用寄存器的索引（Index）都被放在固定的位置，如图 2 所示。因此指令译码器（Instruction Decoder）可以非常便捷的译码出寄存器索引然后读取通用寄存器组（Register File，Regfile）。

图 2 RV32I 规整的指令编码格式



9. 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 架构的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：

（1）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，但是地址非对齐的存储器操作 RISC-V 架构也支持，处理器可以选择用硬件来支持，也可以选择用软件来支持。

（2）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，本文在此不做过多介绍，若对此不甚了解的初学者可以自行查阅学习。

（3）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。

（4）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。

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架构如此定义非常合理，能够达到能屈能伸的效果。譬如：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。

10．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），jal与jalr指令。跳转链接（Jump and Link）指令jal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2个整数操作数，然后对其进行比较，如果比较的条件满足时，则进行跳转。因此，此类指令将比较与跳转两个操作放到了一条指令里完成。

作为比较，很多的其他RISC架构的处理器需要使用两条独立的指令。第一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的架构明确要求其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也能得到不错的性能。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的找到这个固定的位置，并判断其是0还是1来判断其是正数还是负数，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，则可以采用高级的动态分支预测机制来保证性能。

11．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：

（1）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”。

（2）在退出子程序之时，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来完成，使用高层语言（譬如 C 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。

为了加速这个“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。但是此种“Load Multiple”和“Store Multiple”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，笔者在曾经设计此类处理器时便深受其苦。

RISC-V 架构则放弃使用这种“Load Multiple”和“Store Multiple”指令。并解释，如果有的场合比较介意这种“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“Load Multiple”和“Store Multiple”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 能够快速的跳转执行，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，但是同时达到高性能。

12.无条件码执行

很多早期的 RISC 架构发明了带条件码的指令，譬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得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得 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，笔者在曾经设计此类处理器时便深受其苦。

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 能够快速的跳转执行达到高性能。

13.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）”，最具有代表性的便是 MIPS 架构，在很多经典的计算机体系结构教材中，均使用 MIPS 对分支延迟槽进行过介绍。

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因为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，所以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的别扭，CPU 设计人员对此往往苦不堪言。

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 能够准确的预测跳转执行达到高性能。而对于低功耗小面积的 CPU，由于无需支持分支延迟槽，硬件得到极大简化，也能进一步减少功耗和提高时序。

14.无零开销硬件循环

很多 RISC 架构还支持零开销硬件循环（Zero Overhead Hardware Loop）指令，其思想是通过硬件的直接参与，通过设置某些循环次数寄存器（Loop Count），然后可以让程序自动地进行循环，每一次循环则 Loop Count 自动减 1，这样持续循环直到 Loop Count 的值变成 0，则退出循环。

之所以提出发明这种硬件协助的零开销循环是因为在软件代码中的 for 循环（for i=0; i<N; i++）极为常见，而这种软件代码通过编译器编译之后，往往会编译成若干条加法指令和条件分支跳转指令，从而达到循环的效果。一方面这些加法和条件跳转指令占据了指令的条数；另外一方面条件分支跳转如存在着分支预测的性能问题。而硬件协助的零开销循环，则将这些工作由硬件直接完成，省掉了这些加法和条件跳转指令，减少了指令条数且提高了性能。

然有得必有失，此类零开销硬件循环指令大幅地增加了硬件设计的复杂度。因此，零开销循环指令与 RISC-V 架构简化硬件的哲学是完全相反的，在 RISC-V 架构中自然没有使用此类零开销硬件循环指令。

15.简洁的运算指令

在本章第 2.1 节中曾经提到 RISC-V 架构使用模块化的方式组织不同的指令子集，最基本的整数指令子集（I 字母表示）支持的运算包括加法、减法、移位、按位逻辑操作和比较操作。这些基本的运算操作能够通过组合或者函数库的方式完成更多的复杂操作（譬如乘除法和浮点操作），从而能够完成大多数的软件操作。

整数乘除法指令子集（M 字母表示）支持的运算包括，有符号或者无符号的乘法和除法操作。乘法操作能够支持两个 32 位的整数相乘得到一个 64 位的结果；除法操作能够支持两个 32 位的整数相除得到一个 32 位的商与 32 位的余数。

单精度浮点指令子集（F 字母表示）与双精度浮点指令子集（D 字母表示）支持的运算包括浮点加减法，乘除法，乘累加，开平方根和比较等操作，同时提供整数与浮点，单精度与双精度浮点彼此之间的格式转换操作。

很多 RISC 架构的处理器在运算指令产生错误之时，譬如上溢（Overflow）、下溢（Underflow）、非规格化浮点数（Subnormal）和除零（Divide by Zero），都会产生软件异常。RISC-V 架构的一个特殊之处是对任何的运算指令错误（包括整数与浮点指令）均不产生异常，而是产生某个特殊的默认值，同时，设置某些状态寄存器的状态位。RISC-V 架构推荐软件通过其他方法来找到这些错误。再次清楚地反映了 RISC-V 架构力图简化基本的指令集，从而简化硬件设计的哲学。

16.优雅的压缩指令子集

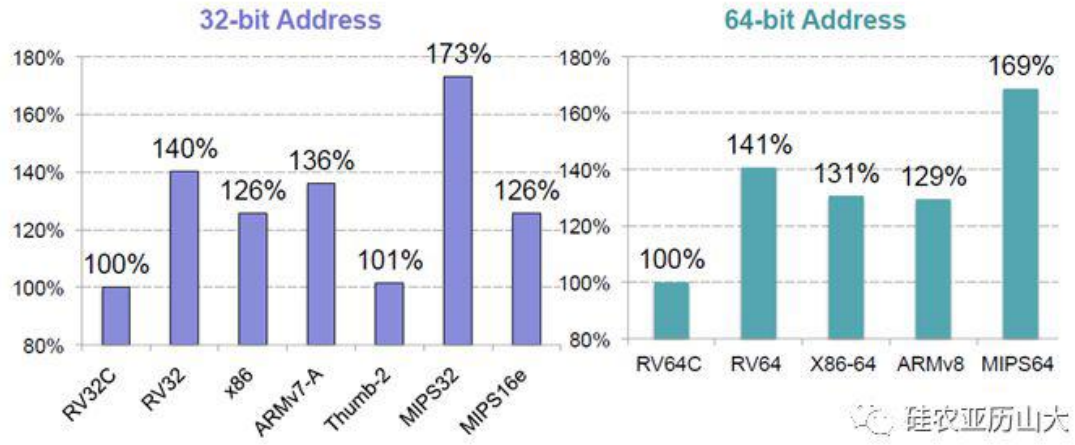
基本的 RISC-V 基本整数指令子集（字母 I 表示 ）规定的指令长度均为等长的 32 位，这种等长指令定义使得仅支持整数指令子集的基本 RISC-V CPU 非常容易设计。但是等长的 32 位编码指令也会造成代码体积（Code Size）相对较大的问题。

为了满足某些对于代码体积要求较高的场景（譬如嵌入式领域），RISC-V 定义了一种可选的压缩（Compressed）指令子集，由字母 C 表示，也可以由 RVC 表示。RISC-V 具有后发优势，从一开始便规划了压缩指令，预留了足够的编码空间，16 位长指令与普通的 32 位长指令可以无缝自由地交织在一起，处理器也没有定义额外的状态。

RISC-V 压缩指令的另外一个特别之处是，16 位指令的压缩策略是将一部分普通最常用的的 32 位指令中的信息进行压缩重排得到（譬如假设一条指令使用了两个同样的操作数索引，则可以省去其中一个索引的编码空间），因此每一条 16 位长的指令都能一一找到其对应的原始 32 位指令。因此，程序编译成为压缩指令仅在汇编器阶段就可以完成，极大的简化了编译器工具链的负担。

RISC-V 架构的研究者进行了详细的代码体积分析，如图 3 所示，通过分析结果可以看出，RV32C 的代码体积相比 RV32 的代码体积减少了百分之四十，并且与 ARM，MIPS 和 x86 等架构相比都有不错的表现。

图 3 各指令集架构的代码密度比较（数据越小越好）



17.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）：

（1）Machine Mode：机器模式，简称 M Mode。

（2）Supervisor Mode：监督模式，简称 S Mode。

（3）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 架构也支持几种不同的存储器地址管理机制，包括对于物理地址和虚拟地址的管理机制，使得 RISC-V 架构能够支持从简单的嵌入式系统（直接操作物理地址）到复杂的操作系统（直接操作虚拟地址）的各种系统。

18．CSR 寄存器

RISC-V 架构定义了一些控制和状态寄存器（Control and Status Register，CSR），用于配置或记录一些运行的状态。CSR 寄存器是处理器核内部的寄存器，使用其自己的地址编码空间和存储器寻址的地址区间完全无关系。

CSR 寄存器的访问采用专用的 CSR 指令，包括 CSRRW、CSRRS、CSRRC、CSRRWI、CSRRSI 以及 CSRRCI 指令。

19.中断和异常

中断和异常机制往往是处理器指令集架构中最为复杂而关键的部分。RISC-V 架构定义了一套相对简单基本的中断和异常机制，但是也允许用户对其进行定制和扩展。

20.矢量指令子集

RISC-V 架构目前虽然还没有定型矢量（Vector）指令子集，但是从目前的草案中已经可以看出，RISC-V 矢量指令子集的设计理念非常的先进，由于后发优势及借助矢量架构多年发展成熟的结论，RISC-V 架构将使用可变长度的矢量，而不是矢量定长的 SIMD 指令集（譬如 ARM 的 NEON 和 Intel 的 MMX），从而能够灵活的支持不同的实现。追求低功耗小面积的 CPU 可以选择使用长度较短的硬件矢量进行实现，而高性能的 CPU 则可以选择较长的硬件矢量进行实现，并且同样的软件代码能够彼此兼容。

21.自定制指令扩展

除了上述阐述的模块化指令子集的可扩展、可选择，RISC-V 架构还有一个非常重要的特性，那就是支持第三方的扩展。用户可以扩展自己的指令子集，RISC-V 预留了大量的指令编码空间用于用户的自定义扩展，同时，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，因此，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。

22.总结与比较

处理器设计技术经过几十年的衍进，随着大规模集成电路设计技术的发展直至今天，呈现出如下特点：

> 由于高性能处理器的硬件调度能力已经非常强劲且主频很高，因此，硬件设计希望指令集尽可能的规整、简单，从而，使得处理器可以设计出更高的主频与更低的面积。
> 以 IoT 应用为主的极低功耗处理器更加苛求低功耗与低面积。
> 存储器的资源也比早期的 RISC 处理器更加丰富。

如上种种这些因素，使得很多早期的 RISC 架构设计理念（依据当时技术背景而诞生），时至今日不仅不能帮助现代处理器设计，反而成了负担桎梏。某些早期 RISC 架构定义的特性，一方面使得高性能处理器的硬件设计束手束脚；另一方面又使得极低功耗的处理器硬件设计背负不必要的复杂度。

得益于后发优势，全新的 RISC-V 架构能够规避所有这些已知的负担，同时，利用其先进的设计哲学，设计出一套“现代”的指令集。本节再次将其特点总结如表 2 所示。

表 2 RISC-V 指令集架构特点总结

特性	x86 或 ARM 架构	RISC-V
架构篇幅	数千页	少于三百页
模块化	不支持	支持模块化可配置的指令子集
可扩展性	不支持	支持可扩展定制指令
指令数目	指令数繁多，不同的架构分支彼此不兼容	一套指令集支持所有架构。基本指令子集仅 40 余条指令，以此为共有基础，加上其他常用模块子集指令总指令数也仅几十条
易实现性	硬件实现的复杂度高	硬件设计与编译器实现非常简单： ➢ 仅支持小端格式 ➢ 存储器访问指令一次只访问一个元素 ➢ 去除存储器访问指令的地址自增自减模式 ➢ 规整的指令编码格式 ➢ 简化的分支跳转指令与静态预测机制 ➢ 不使用分支延迟槽（Delay Slot） ➢ 不使用指令条件码（Conditional Code） ➢ 运算指令的结果不产生异常（Exception） ➢ 16 位的压缩指令有其对应的普通 32 位指令 ➢ 不使用零开销硬件循环

一言以蔽之，RISC-V 的特点在于极简、模块化以及可定制扩展，通过这些指令集的组合或者扩展，你几乎可以构建适用于任何一个领域的微处理器，比如云计算、存储、并行计算、虚拟化/容器、MCU、应用处理器和 DSP 处理器等。

参考资料：

1. 部分来源于百度百科，维基百科。
2. 大道至简单-ＲＩＳＣ－Ｖ架构之魂

The author:an chuanxu
2018/6