Computer Science and Application 计算机科学与应用, 2019, 9(9), 1667-1682
Published Online September 2019 in Hans. http://www.hanspub.org/journal/csa
https://doi.org/10.12677/csa.2019.99186

Hans汉斯

Research on Instruction Set Architecture of 40-Bit Processor

Mingyi Xu

School of Water Resources and Hydropower Engineering, Wuhan University, Wuhan Hubei
Email: myxu@whu.edu.cn

Received: Aug. 19th, 2019; accepted: Sep. 3rd, 2019; published: Sep. 10th, 2019

Abstract

The preliminary design of a free and open source RISC architecture is proposed for mobile phones and personal computers. This ISA has high availability and scalability with various length in Bytes and simple decode rules convenient for hardware implementation. The total 127 instructions are defined including 93 basic instructions and 34 extended instructions.

Keywords

ISA, 40-Bit Processor, RISC, Mobile Phone, Personal Computer

40位处理器指令集架构研究

徐明毅

武汉大学水利水电学院，湖北 武汉
Email: myxu@whu.edu.cn

收稿日期：2019年8月19日；录用日期：2019年9月3日；发布日期：2019年9月10日

摘　要

初步设计了适用于手机和个人电脑的40位处理器的免费开源精简指令集，具备以字节为单位的可变长度，解码规则简单，便于硬件实现，可用性和扩展性好，已定义共127条指令，包含基本指令93条和扩展指令34条。

关键词

指令集架构，40位处理器，精简指令集，手机，个人电脑

文章引用: 徐明毅. 40 位处理器指令集架构研究[J]. 计算机科学与应用, 2019, 9(9): 1667-1682.
DOI: 10.12677/csa.2019.99186

Copyright © 2019 by author(s) and Hans Publishers Inc.
This work is licensed under the Creative Commons Attribution International License (CC BY).
http://creativecommons.org/licenses/by/4.0/
Open Access

1. 引言

当前中国在信息产业上的重大短板是“缺心少魂”，即缺少市场占有率显著的自主处理器芯片和操作系统，当然情况也在逐步改善，如海思麒麟芯片、龙芯和一些国产操作系统等正在努力扩大市场份额。芯片应用涉及到生态链问题，芯片之上是操作系统，操作系统之上是应用软件，芯片和操作系统是基础，应用软件是对用户产生核心价值的部分，如同冰山浮出水面上的部分。没有应用软件支持，芯片和操作系统等底层部分也会自然消失。从资源投入来说，若芯片开发代价为 1，操作系统开发代价为 10，则各种应用软件的开发代价在 100 以上，投入极大。因此某类芯片要成功占领一部分市场，只有在长期积累下，在芯片及操作系统成熟可靠后，各种应用开发者主动适配该平台，逐渐形成软件生态，才能在市场中站稳脚跟。

发展芯片的途径之一是做兼容芯片，即搭上便车，不用考虑操作系统和应用环境的问题，但必须付出可观的专利授权费用才能使用，一旦不能获得授权，就面临卡脖子的问题。因此做兼容芯片能获得一部分市场，但难以成为市场的领头羊。众所周知，指令集是底层硬件和上层软件的接口，如果不用市场主流但费用高昂的指令集，而是采用免费或费用低廉的新指令集，则发展空间更大，但生态链的发展过程较为缓慢。要使新的指令集尽快应用起来，可以利用已有计算环境，先开发基于新指令集的虚拟机，绕过底层硬件开发上层软件，在验证可靠后，再反向完成底层硬件开发，这样初期投入较小，成本较低。只要指令集是稳定的，硬件和软件可独立迭代升级，为生态链的打造赢得更多时间。

在现有市场中，手机处理器和个人电脑处理器都已从 32 位进化到 64 位。32 位寻址有 4G 内存限制，不能满足大型应用软件的需要，但 64 位寻址空间对个人计算设备来说稍嫌太大，实际应用的处理器并未采用，大多只支持 40 位或 48 位的寻址空间。而采用 40 位寻址，管理内存空间能达到 1024 G = 1 T，从现有的大多数应用软件看，基本够用。因此，针对手机、平板、笔记本电脑以及一部分个人电脑，采用 40 位处理器芯片能够满足需要，很大程度上又可能比 64 位处理器更节省计算资源，如果开发出来，可望在市场中实际采用。

本文参考已有的开源处理器指令架构，如 RISC-V 指令集等[1] [2]，对 40 位处理器的指令集进行了初步设计，以期提供思路，抛砖引玉，并逐步完善，力争在该指令集上开发的芯片能够在市场中占领一席之地。没有历史包袱，使得新的指令集容易做到精简易用，便于简化硬件设计，提高运行速度，同时考虑发展需要，也保留了较好的扩展性，可根据市场要求进行扩充。该指令集架构使芯片设计从底层开始就与现有产品不同，虽然初期发展困难，但可以避开原有的专利限制，省却昂贵的授权费用，从而打造一条自主可控的产业链，为中国的信息产业发展提供坚实基础。

2. 主流的指令集架构

处理器应用领域可简单分为服务器领域、个人电脑领域和嵌入式领域，随着进一步发展，嵌入式领域也发展成几个不同的子领域。随着智能手机和手持设备的发展，移动领域发展为规模甚至超过个人电脑领域的独立领域。现今服务器领域和个人电脑领域由 x86 架构占据主导地位，移动领域主要由 ARM 架构所垄断，其次是实时嵌入式领域，仍然以 ARM 架构占据大多数市场份额，最后是深嵌入式领域，注重低功耗、低成本和高能效比，对软件生态的依赖性相对较低，目前仍然以 ARM 架构为主流[1]。

指令集架构可以理解为一个抽象层，该抽象层构成处理器底层硬件与运行于其上的软件之间的桥梁和接口，指令集架构是区分不同的 CPU 的主要标准。指令集架构主要分为复杂指令集 CISC 和精简指令集 RISC，在早期，CISC 曾经是主流，因为可以用较少的指令完成更多的操作，但是大量的特殊指令让 CPU 设计变得极为复杂，因此现代的指令集架构基本选择 RISC 架构[3]。

除了 CISC 与 RISC 之分，处理器指令集架构的位数也是一个重要区分。譬如 32 位架构的处理器，其通用寄存器的宽度为 32 位，能够寻址的范围为 2^{32} B，即 4 GB 的寻址空间。64 位架构的处理器，其通用寄存器的宽度为 64 位，理论上能够寻址的范围为 2^{64} B，但目前使用的 64 位处理器大多只支持 40 位或 48 位的寻址空间。

x86 架构是由 Intel 公司于 1978 年推出的 8086 处理器使用的复杂指令集，从最初的 16 位架构发展到如今的 64 位架构，逐渐成为个人电脑的标准处理器架构。Intel 公司通过内部“微码化”方法，即先用解码器将指令翻译成为内部的简单指令，然后送给流水线执行，处理器内核已变成 RISC 架构。硬件解码器也带来额外的复杂度和面积开销，这是 x86 架构作为一种 CISC 架构不得不付出的代价。x86 架构不仅在个人计算机领域取得统治性地位，迄今为止还在服务器市场取得主导地位。

SPARC 指令集架构由 Sun 公司于 1985 年设计，是一种非常有代表性的高性能 RISC 架构，赢得了当时高端处理器市场的领先地位[4] [5]。其最大的特点是采用寄存器窗口，通过切换不同的寄存器组快速响应函数调用与返回，因此能够带来非常高的性能。后来，Sun 公司被 Oracle 公司收购，在 2017 年 9 月，Oracle 公司宣布放弃硬件业务，至此 SPARC 处理器可以说正式退出了历史舞台。

Power 架构是 IBM 公司开发的一种 RISC 架构指令集，1994 年 IBM 基于此推出 PowerPC 604 处理器，其强大的性能在当时处于全球领先地位。2013 年 IBM 推出 Power8 处理器，处理器核心数量达到 12 个，2017 年推出 Power9 处理器，核心数量为 24 个，并计划在 2020 年推出 Power10 处理器。

MIPS 架构是一种简洁、优化的 RISC 架构，由斯坦福大学开发，曾经是最受欢迎的 RISC 架构，也为我国龙芯处理器所采用。但由于商业运作的原因，MIPS 架构被同属 RISC 阵营的 ARM 架构后来居上，2013 年 MIPS 被英国 Imagination 公司收购，2017 年 Imagination 自身出现危机而寻求整体出售，使 MIPS 架构日渐式微。

Alpha 架构是由 DEC 公司设计开发的一种 64 位的 RISC 架构指令集，我国的申威处理器采用并扩展使用。2001 年，康柏收购 DEC 之后，逐步将 64 位服务器系列产品转移到 intel 的安腾处理器架构上。2004 年，惠普收购康柏，从此 Alpha 架构逐渐淡出了人们的视野。

ARM 架构由位于英国的处理器设计与软件公司 ARM 设计，2004 年推出 ARMv7 内核架构时，将架构分为三类：面向性能密集型系统的 Cortex-A 内核，面向实时应用的 Cortex-R 内核，面向各类嵌入式应用的 Cortex-M 内核，其中，Cortex-A 内核在移动领域占据统治地位，如苹果公司的 iphone 手机的处理器 A7-A12 都是 ARM 架构，华为公司的海思麒麟系列芯片也是属于 ARM 架构。在微控制器领域，主流厂商几乎都有使用 ARM 的 Cortex-M 内核的产品。

RISC-V 架构是处理器架构新秀，由美国加州大学伯克利分校研究人员于 2010 年提出，是一种简单且开放免费的指令集架构[1] [2]，2016 年，成立了 RISC-V 基金会，负责维护标准的 RISC-V 指令集手册与架构文档，并推动 RISC-V 架构的发展。印度将 RISC-V 架构指定为国家标准进行发展，但 RISC-V 架构的生态还不够强，远没有达到威胁 x86 和 ARM 架构的程度，还处于蓬勃发展阶段。

可以看到，处理器芯片竞争的核心是指令集架构的竞争，也是相关生态链的竞争。每一种指令集从提出到成熟，都是在市场竞争中发展壮大的。非市场主导的指令集架构日渐式微，这其中并不完全是技术的原因，而主要与商业运作息息相关。迫于竞争压力，各非主导地位的指令集架构已逐步开源。

3. 指令集设计原则

当前，PC 领域和服务器领域主要由 x86 芯片主导，手机处理器芯片主要由 ARM 芯片主导，对应的操作系统也不同。从发展来看，手机芯片的性能越来越强，手机领域和 PC 领域的界限越来越模糊，两者都希望渗透到对方领域。如果有一种处理器能同时覆盖市场的大部分范围，则有可能在市场的间隙地带取得成功。40 位处理器直接支持 1T 的寻址空间，能够满足现今大部分应用软件的需求，且不存在性能过剩和多余设计，够用且经济，正好可以满足市场需要。

考虑指令集架构的发展趋势，采用精简指令集 RISC，好处是可以简化设计，提高可靠性，还可以节省芯片面积，提高运行速度。指令集的数目尽量减少，一条指令尽量多用，可以使用伪指令的形式来简化使用。对于一条指令语句来说，通常可由操作码和操作数两个域组成，而其中操作数又分无操作数、有一个操作数、有两个操作数等等。指令如果采用定长的形式，格式规整，译码简单，但是存储空间有所浪费，如果采用不定长形式，存储空间节省，但译码复杂。综合考虑，指令格式采用不定长形式，但限定于有限的几种，便于在译码速度和指令大小之间取得平衡。

寄存器的数目选择也很重要。考虑到遵循“越近越快”原则，CPU 处理时应尽量在寄存器中进行，因此应该设计较大的寄存器空间。但寄存器空间过大，又会提高芯片造价，若不能完全利用，运行时则白白增加耗能，故寄存器的数量要适当，与编译器的发展水平有关。x86 的 8086 处理器只有 8 个通用寄存器，进化到 64 位后，增加到 16 个通用寄存器，ARMv8 64 位处理器则为 31 个，RISC-V 处理器一般使用 32 个，SPARC V8 架构处理器则可以拥有更多的寄存器[6] [7]。从发展看，寄存器的数量在逐渐增加，只要编译器能够挖掘出潜力，设计较多数量的寄存器能够使计算局部化，提高程序运行速度的同时减少运行功耗。综合考虑，设置至少 32 个通用寄存器，占用的地址编码为 5 位，考虑到未来需要，采用 1 个字节来表示通用寄存器的地址编码，则最大可支持 256 个通用寄存器。

在程序运行时，对子程序的调用和返回是非常频繁的操作，这就要进行“保存现场”和“恢复现场”的操作，一般通过堆栈的方法来完成，但这涉及到内存操作，速度上有影响，能否将“现场”保留在寄存器中，达到快速调用程序的目的呢？SPARC 处理器巧妙采用寄存器窗口的方法来实现。SPARC 处理器可以有 6~32 个寄存器窗口，每个窗口包括 8 个全局寄存器和 24 个工作寄存器。全局寄存器用来存放整体量，所有窗口共用。工作寄存器逻辑上分为 3 组：8 个输入寄存器，8 个输出寄存器，8 个局部寄存器。执行函数调用时通过改变当前窗口指针分配给该函数一个窗口。由于寄存器窗口的重叠，调用者的输出寄存器和被调用者的输入寄存器是同一组寄存器，因此省掉了寄存器之间的数据传送，极大地提高了速度。当函数调用嵌套深度超出窗口数而发生上溢时，处理器内部产生一个中断，由软件调整若干窗口到主存中去[4]。

对于主要考虑性能的处理器设计，可以参考 SPARC 处理器，实现寄存器窗口，这实际上扩展了通用寄存器的数量，也可以高效地向子程序传递参数，节省环境切换时的开销，免掉现场保护与恢复等操作。窗口大小仍然可以设置为 32 个通用寄存器，将高于此地址编号的设置为全局寄存器，便于在程序间传递更多的参数，或者为子程序增加可用的通用寄存器数量。可考虑采用堆栈形式，将所有窗口公用的全局寄存器的第一个位置设置为栈顶指针，每个子程序使用时，改变该指针，划分出独立的不受干扰的空间，返回时，恢复栈顶指针，释放空间即可。这样实际上在寄存器中实现了堆栈，比在内存中实现应该能够提高运行速度。如果全局寄存器也实现窗口特性，则更可以扩展寄存器空间，但一般情况下似无必要。对于不使用寄存器窗口的设计，也即是寄存器窗口只有一个的特殊情况下，子程序调用时只需要将当前寄存器内容保存到内存堆栈中去，返回时恢复即可。因此寄存器窗口采用与否，与底层的硬件实现有关，从指令上看是无缝兼容的，提高了指令集的灵活性和扩展性。

由以上讨论，可定义指令的基本格式为：针对 40 位处理器，通用寄存器的宽度为 40 位，操作码为 8 位，理论上容纳 256 条指令，通用指令在 128 编号以内，而 128 号及以上编号为可选的扩展指令和用户扩展指令。看起来操作码空间十分有限，但由于采用精简指令集，仍然能够满足要求，在特定的情况下，还可采用灵活的扩展方式。操作数的编码也为 8 位，最大可支持 256 个通用寄存器，要求实现的最少通用寄存器数目为32个。如果实现寄存器窗口，则编码的低地址(如32以内)为程序使用的局部寄存器，编码的高地址为全局寄存器。

为减少指令占用空间，采用变长指令，但以字节为单位，因此有单字节指令、双字节指令、3 字节指令、4 字节指令、5 字节指令五种类型，但保留有更长字节指令的可能性。为减少转换，采用数据排列顺序与网络数据发送顺序一致，即采用大端模式。典型的指令顺序为 1 字节操作码，1 字节目的寄存器地址，1 字节源寄存器 1 地址，1 字节源寄存器 2 地址，1 字节可选操作(源寄存器 3 地址或长度指示等)。若无特别指定，操作码和每个操作数都为 1 个字节，但对于特殊情况，操作数可以占多个字节。

4. 基本整数指令

4.1. 单字节指令

该类指令没有操作数，只有操作码，因此只有一个字节，如表 1 所示。其中全 0 的操作码用于检查非法指令，这在错误使用数据区作为指令区时就容易出现。WFI 指令为休眠指令，此时处理器主动休眠，等待外部中断唤醒，这便于降低功耗。NOP 指令为无操作状态，相当于暂停，不进行任何操作，可用于等待，此时功耗较休眠状态高，好处是可立即恢复到工作状态。ECALL 指令和 ERET 指令用于系统中断的调用和返回，所需的参数由系统的一些特殊寄存器给出。EBRK 指令用于程序调试，便于主动产生一个调试异常。在正常的指令执行序列下，用中断和异常管理外部突发的事件。中断可用 4 位定义中断级别，4 位定义中断类型，合成 1 个字节。异常也可称为软中断，也可用 1 个字节来管理级别和类型。

Table 1. Single byte instructions
表 1. 单字节指令

编号	名称	格式	长	说明
0	错误	ERR	1	为非法指令，便于检查
1	休眠	WFI	1	主动休眠，等待中断唤醒
2	空闲状态	NOP	1	无操作，暂停状态
3	系统调用	ECALL	1	发起系统中断或异常调用
4	系统返回	ERET	1	系统中断和异常调用返回
5	产生断点异常	EBRK	1	发起断点异常调用，进入调试模式
6	获得寄存器窗口	NXTW	1	获得一个新的寄存器窗口
7	回退寄存器窗口	BCKW	1	放弃当前寄存器窗口，回退到上一窗口

NXTW 指令和 BCKW 指令用于寄存器窗口调度，窗口的范围用当前位置和末尾位置来标识。如果当前位置增加后与末尾位置相同，则发生上溢，而当前位置减少后与末尾位置相同，则发生下溢，之间的部分即为空闲窗口。通常情况下，NXTW 指令将当前寄存器窗口的指针移动到下一个位置即可，如果产生上溢，则需要将已被占用的窗口内的寄存器值保存到内存后，才得到可用的寄存器窗口。BCKW 指令将窗口的当前位置回退一个位置，如果发生下溢，则需要将内存中的寄存器值读回，才是正确的当前

窗口。由于有内存作为后备，因此寄存器窗口并不需太多，用 4 位来表示即可，即最多实现 16 个寄存器窗口。每个窗口实现 8 个输入寄存器，8 个输出寄存器，16 个局部寄存器，相邻窗口的输入寄存器和输出寄存器重叠，因此实际每个窗口只占用 24 个寄存器。在特殊情况下，即只有一个寄存器窗口时，NXTW 指令和 BCKW 指令相当于“保存现场”和“恢复现场”的操作，使用该指令可以简化子程序调用，当然也可自行实现寄存器值的保存和恢复操作，便于更灵活控制需要保存的寄存器数量。局部寄存器窗口的调度方法可同样用于全局寄存器，使得通用寄存器的地址编码虽只有 1 个字节，但能突破范围限制，这对于一些应用来说也许是必要的。

4.2. 双字节指令

该部分指令只需要两个字节，用来读取一些重要的系统状态，如表 2 所示。在处理器内部设置一个主频节拍计数器，用于高精度计时，处理器一个频率周期计数一次。用一个 40 位特殊寄存器来保存节拍计数，在计数一定间隔时产生一个中断，这时可以检查系统中的等待任务，RDCYC 指令可读取该计数。另外为准确计时的需要，设置时钟计数器，频率比主频低，用一个 40 位寄存器来保存计数，每隔一定计数产生一次中断，可处理一些低优先级的中断或异常，RDTIME 指令读取该时钟计数。为节省硬件成本，特殊情况下可由主频计数器中断来设置时钟计数器。另外设置程序执行指令计数器，为 40 位特殊寄存器，可以用 RDPC 读取当前的指令计数器值。

Table 2. Double byte instructions
表 2. 双字节指令

编号	名称	格式	长	说明
8	读取主频计数	RDCYC:rd	2	读取频率计数器值，放置到 rd 中
9	读取时钟计数	RDTIME:rd	2	读取时钟计数器值，放置到 rd 中
10	读取指令计数	RDPC:rd	2	读取当前指令的程序计数器值到 rd 中
11	读取堆栈栈顶	RDSTK:rd	2	读取堆栈栈顶指针到 rd 中
12	设置堆栈栈顶	WRSTK:rs	2	将寄存器 rs 中内容设置为栈顶指针
13	增减堆栈栈顶	INCSTK:imm	2	将堆栈的栈顶指针增加立即数 imm
14	计数加 1	INC:rd	2	计算[rd]+=1
15	计数减 1	DEC:rd	2	计算[rd]-=1

设置 40 位的堆栈栈顶指针特殊寄存器，该指针指向内存地址，读取时用指令 RDSTK，设置时用指令 WRSTK，便于切换不同的堆栈空间。移动栈顶指针用指令 INCSTK，一次移动的量为 1 个字节，为有符号立即数，因此分配或释放的最大值为 127 字节。使用时可以灵活采用向下堆栈或向上堆栈，分配空间后，采用内存读写指令在内存堆栈和寄存器之间移动数据。

由于递增和递减操作经常使用，故定义了加 1 指令 INC 和减 1 指令 DEC。

4.3. 载入存储指令

载入存储指令负责寄存器与内存之间的数据交换，有单字(五字节)、双字(十字节)和多字节的读写指令，如表 3 所示。要求对某个处理核心，读写顺序严格按照指令的先后顺序执行，即使用按序一致性模型，这样无需使用存储器屏障指令。多核心的并行调度采用特定的存储器读写原子指令来完成。

Table 3. Load-Save instructions
表 3. 载入存储指令

编号	名称	格式	长	说明
80	载入五字节	LF: rd, rs1,imm[15:0]	5	以[rs1]+imm 为地址，载入五个字节到 rd 寄存器中
81	载入十字节	LT: rd, rs1,imm[15:0]	5	以[rs1]+imm 为地址，将十个字节载入 rd 中，若 rd 寄存器已满，则按顺序移到下一位置
24	载入多字节	LB: rd, rs1, len	4	以[rs1]为初始地址，连续载入 len 个字节到 rd 寄存器中，若 rd 寄存器已满，则按顺序移到下一位置
82	存储五字节	SF: rs2, rs1,imm[15:0]	5	以[rs1]+imm 为地址，将 rs2 中五个字节存入存储器
83	存储十字节	ST: rs2, rs1,imm[15:0]	5	以[rs1]+imm 为地址，将 rs2 中十个字节存入存储器
25	存储多字节	SB: rs2, rs1, len	4	以[rs1]为初始地址，连续将 rs2 中 len 个字节存入存储器，若 rs2 已读完，则按顺序移到下一位置

4.4. 整数加减指令

基本整数指令包含40位整数的加减和复制操作，如表4所示。对40位无符号数，表示范围为0~2^{40}−1，即 0~1099511627775，对 40 位有符号数，最高位为符号位，表示范围为−549755813887~549755813887。实现时，根据整数的最高位来判断正负，负整数用补码的形式表示，因此无符号整数和有符号整数的加法形式一样，不需要区分。结合填充立即数指令 LUI 和立即数加法指令 ADDI，可以对 40 位的整数立即数进行处理。对于程序中循环变量一次增减大于 1 的情况，用自加操作 INCB 处理，立即数 imm 为 8 位，可正可负，范围为−127~127。利用补码，立即数减法可转化为加法进行。

Table 4. Integer addition and subtraction instructions
表 4. 整数加减指令

编号	名称	格式	长	说明
26	加法	ADD: rd, rs1, rs2	4	计算[rd]=[rs1]+[rs2]
27	减法	SUB: rd, rs1, rs2	4	计算[rd]=[rs1]-[rs2]
84	立即数加法	ADDI: rd, rs1, imm[15:0]	5	计算[rd]=[rs1]+imm，imm 为 16 位立即数
85	填充立即数	LUI: rd, imm[23:0]	5	将 24 位立即数 imm 左移 16 位(低位补 0)，放置到 rd 寄存器中
16	自加操作	INCB: rd, imm	3	计算[rd]+=imm，imm 为 8 位立即数
17	复制内容	MOV: rd, rs1	3	将 rs1 寄存器内容复制到 rd 寄存器中

4.5. 整数乘除指令

整数的乘除运算指令包含了相乘、相除、求余数等功能，如表 5 所示。如果希望得到两个 40 位整数相乘的完整 80 位结果，可以融合两条指令连续执行“MUL:rd, rs1, rs2；MULH[U]: rd+1, rs1, rs2”，此处将 80 位结果的小端部分放在前，大端部分放在后，便于 40 位整数和 80 位整数的相互转换，无需特别的转换指令。40 位整数转换为 80 位整数时，将后续的 40 位扩充为大端部分，无符号数和有符号数的正数填充全零，有符号数的负数填充全 1 (补码表示)。80 位整数转换为 40 位整数时，必须满足相反条件，抛弃大端部分即可。

Table 5. Integer multiplication and division instructions
表 5. 整数乘除指令

编号	名称	格式	长	说明
28	整数相乘低位	MUL: rd, rs1, rs2	4	rd 保存[rs1]*[rs2]的低 40 位
29	整数相乘高位	MULH: rd, rs1, rs2	4	rd 保存 rs1]*[rs2]的高 40 位
30	无符号整数相乘高位	MULHU: rd, rs1, rs2	4	rd 保存 40 位无符号整数相乘[rs1]*[rs2]的高 40 位
31	整数相除	DIV: rd, rs1, rs2	4	整数相除[rd]=[rs1]/[rs2]
32	无符号整数相除	DIVU: rd, rs1, rs2	4	无符号整数相除[rd]=[rs1]/[rs2]
33	整数求余数	REM: rd, rs1, rs2	4	整数求余[rd]=[rs1]%[rs2]
34	无符号整数求余数	REMU: rd, rs1, rs2	4	无符号整数求余[rd]=[rs1]%[rs2]

4.6. 逻辑操作指令

逻辑操作指令包含了与、或、异或操作，如表 6 所示，可以是寄存器与寄存器操作，也可以是寄存器和立即数操作，立即数为 2 个字节长度。

Table 6. Logic operation instructions
表 6. 逻辑操作指令

编号	名称	格式	长	说明
35	异或	XOR: rd, rs1, rs2	4	[rs1]与[rs2]进行异或，结果放置到 rd
86	立即数异或	XORI: rd, rs1, imm[15:0]	5	[rs1]与 imm 进行异或，结果放置到 rd
36	或	OR: rd, rs1, rs2	4	[rs1]与[rs2]进行或操作，结果放置到 rd
87	立即数或	ORI: rd, rs1, imm[15:0]	5	[rs1]与 imm 进行或操作，结果放置到 rd
37	与	AND: rd, rs1, rs2	4	[rs1]与[rs2]进行与操作，结果放置到 rd
88	立即数与	ANDI: rd, rs1, imm[15:0]	5	[rs1]与 imm 进行与操作，结果放置到 rd

4.7. 移位操作指令

移位操作指令包含了左移、右移、算术右移操作，如表 7 所示，可以是寄存器与寄存器操作，也可以是寄存器和立即数操作，立即数为 1 个字节长度。

Table 7. Shift operation instructions
表 7. 移位操作指令

编号	名称	格式	长	说明
38	左移	SLL: rd, rs1, rs2	4	[rd]=[rs1]<<[rs2]，低位补入 0
39	立即数左移	SLLI: rd, rs1, imm	4	[rd]=[rs1]<< imm，低位补入 0
40	右移	SRL: rd, rs1, rs2	4	[rd]=[rs1]>>[rs2]，高位补入 0
41	立即数右移	SRLI: rd, rs1, imm	4	[rd]=[rs1]>> imm，高位补入 0
42	算术右移	SRA: rd, rs1, rs2	4	[rd]=[rs1]>>[rs2]，最高位保持符号位
43	立即数算术右移	SRAI: rd, rs1, imm	4	[rd]=[rs1]>> imm，最高位保持符号位

4.8. 比较操作指令

比较操作指令包含小于、小于等于、等于比较操作，如表 8 所示，可以是寄存器与寄存器比较，也可以是寄存器与立即数比较，立即数为 2 个字节长度。

Table 8. Comparison operation instructions
表 8. 比较操作指令

编号	名称	格式	长	说明
44	小于	SLT: rd, rs1, rs2	4	如果[rs1]<[rs2]，[rd]=1，否则为 0
89	立即数小于	SLTI: rd, rs1, imm[15:0]	5	如果[rs1]< imm，[rd]=1，否则为 0
45	小于等于	SLE: rd, rs1, rs2	4	如果[rs1]<=[rs2]，[rd]=1，否则为 0
90	立即数小于等于	SLEI: rd, rs1, imm[15:0]	5	如果[rs1]<= imm，[rd]=1，否则为 0
46	等于	SEQ: rd, rs1, rs2	4	如果[rs1]==[rs2]，[rd]=1，否则为 0
91	立即数等于	SEQI: rd, rs1, imm[15:0]	5	如果[rs1]==imm，[rd]=1，否则为 0

4.9. 控制转移指令

控制转移指令用于程序的跳转操作，如表 9 所示。JAL 指令用于近跳转，跳转到与当前位置偏移 24 位立即数地址，该立即数为有符号整数，因此能跳转到前后 8M 的地址区间。更远的跳转，可使用 JALR 指令，跳转偏移为寄存器与立即数的和。在近跳转和远跳转时，将当前指令地址保存到寄存器，这样在返回时，跳转到该地址递增 5 的位置，即下一指令地址，就结束子程序调用，继续执行原来的指令流。

在程序的内部跳转可以使用 BLT、BLE、BEQ、BNE 指令，在进行比较操作后进行跳转，通常无需返回，故不用保存当前指令地址，跳转偏移立即数为 16 位有符号数，因此偏移量范围为±32 KB。

Table 9. Control transfer instructions
表 9. 控制转移指令

编号	名称	格式	长	说明
92	近跳转	JAL: rd, imm[23:0]	5	跳转到当前指令偏移 imm 的地址，rd 保存当前指令位置
93	远跳转	JALR: rd, rs1, imm[15:0]	5	跳转到偏移[rs1]+imm 的指令地址，rd 保存当前指令位置
94	小于	BLT: rs1, rs2,imm[15:0]	5	如果[rs1]<[rs2]，跳转到偏移 imm 地址
95	小于等于	BLE: rs1, rs2,imm[15:0]	5	如果[rs1]<=[rs2]，跳转到偏移 imm 地址
96	等于	BEQ: rs1, rs2,imm[15:0]	5	如果[rs1]==[rs2]，跳转到偏移 imm 地址
97	不等于	BNE: rs1, rs2,imm[15:0]	5	如果[rs1]!=[rs2]，跳转到偏移 imm 地址

4.10. 控制状态设置指令

在处理器架构中，需要定义一些控制和状态寄存器，用于配置或记录一些运行的状态，称之为 CSR 寄存器，使用专有的地址编码空间，便于统一管理。如架构的版本信息、寄存器窗口的控制信息、整数和浮点数的运算状态、中断的控制状态等。CSR 寄存器的访问采用 CSR 指令统一管理，如表 10 所示，对于一些常用的操作指令还可用伪指令来简化使用。CSR 寄存器的编码仍然采用 8 位，即一个字节，它们与通用寄存器的编码空间互相独立，互不干扰。每个 CSR 寄存器编码对应 1 个字节宽度，因此最大管理空间是 256 个字节，并不充裕，需要节省使用。用其它指令能读取的内容，如前述的时钟计数等可不

必纳入编码空间，以保留足够的扩充能力。CSR 寄存器可以统一设置(CSRRW 指令)，也可以按位设置(CSRRS 指令)或者按位清除(CSRRC)。该三条指令允许从初始地址开始，进行完整的 40 位值读取和设置。

Table 10. Control state setting instructions
表 10. 控制状态设置指令

编号	名称	格式	长	说明
47	全部设置	CSRRW: rd, csr, rs1	4	读取 csr 内容到 rd 中，然后将 rs1 内容写入 csr 中
48	比特位设置	CSRRS: rd, csr, rs1	4	读取 csr 内容到 rd 中，以 rs1 中的值逐位作为参考，若某比特位为 1，则将 csr 中对应的比特位置 1
49	比特位清除	CSRRC: rd, csr, rs1	4	读取 csr 内容到 rd 中，以 rs1 中的值逐位作为参考，若某比特位为 1，则将 csr 中对应的比特位置 0
50	立即数设置	CSRRWI: rd, csr, imm	4	读取 csr 内容到 rd 中，然后将 8 位立即数写入 csr 中
51	立即数比特位设置	CSRRSI: rd, csr, imm	4	读取 csr 内容到 rd 中，以 8 位立即数中的值逐位参考，若某比特位为 1，则将 csr 中对应的比特位置 1
52	立即数比特位清除	CSRRCI: rd, csr, imm	4	读取 csr 内容到 rd 中，以 8 位立即数中的值逐位参考，若某比特位为 1，则将 csr 中对应的比特位置 0

CSR 寄存器的设置还可以直接利用立即数来处理，立即数为 8 位，一个字节，可以统一设置(CSRRWI 指令)，也可以按位设置(CSRRSI 指令)或者按位清除(CSRRCI)。此时只使用通用寄存器的低 8 位和 CSR 寄存器进行数据交换，高位默认扩展补 0。如果立即数的值为 0，则不会进行写操作，只进行读取操作。如果结果寄存器的索引值为 0，则不会进行读操作。

4.11. 原子存储操作指令

处理器对速度的追求是无止境的，超线程技术能尽量挖掘处理器内部执行部件的能力，也就是用同一套执行机构，不同的寄存器来达到程序快速切换的能力，避免了保存现场和恢复现场的费时操作，这与采用寄存器窗口的思路是一致的。在单个处理器核心充分挖掘达到速度瓶颈后，一般通过多个核心来提高处理能力。但核心之间的并行性必须要协调，由于多个核心一般围绕同一份数据进行处理，因此只要保证数据的一致性，就容易达成多核心的并行性，故多核心的并行性一般通过内存变量的同步来处理。在核心数少时，某个核心读写时可以通知其他核心暂停存储器操作，保证读写共享变量的唯一性。在核心数多时，通过总线来控制更简单一些。此时某个核心锁定总线，只有当前核心能进行读写，其它核心则不能操作存储器，保证了共享变量的正确读写。

原子性存储操作指令为多核心执行读－算－写操作时对共享变量进行同步，即在对存储器执行操作时，不被其它核心干扰。在读出和写回的间隙，存储器的该地址不能被其它的进程或线程访问，这保证了共享变量的操作唯一性，然后借此达成多个核心的同步和并行。这些原子计算存储指令原子性地从 rs1 地址读取数据值、将这个值写入寄存器 rd、在这个值和 rs2 的原始值上施加一个二进制操作、然后把结果保存到 rs1 地址的存储器中。支持的操作包括原子比较交换、交换、整数加、逻辑操作等，如表 11 所示。

这些指令在执行时先尝试获取存储总线，获取后进行读取并计算，存储结果后释放存储总线。如果不能获取总线，一般需要等待，这阻塞了指令执行，对计算速度有所影响。其中原子比较交换指令 AMCMPS 无需等待，可用来实现加锁，一般用 1 表示加锁状态，用 0 表示解锁状态。如果发现为 0 的解锁状态，则进行加锁，否则退出，可等待片刻后再尝试加锁。解锁时更为简单，只需将解锁信号传递即可，无需比较，可采用原子交换指令 AMSWAP。为并行计算的简单性，无需加锁解锁的繁琐过程，可采用原子加、与、或、异或、取最小值指令，这些指令能保证核心操作时的唯一性。

Table 11. Atomic storage operation instructions
表 11. 原子存储操作指令

编号	名称	格式	长	说明
53	原子比较交换	AMCMPS: rd, rs2, rs1	4	以[rs1]为地址，读出五字节内容，与 rd 寄存器中内容比较，相等则将[rs2]存储到[rs1]位置，不相等则将 rd 设为读出值，但并不存储[rs2]到[rs1]位置
54	原子交换	AMSWAP: rd, rs2, rs1	4	以[rs1]为地址，读出五字节存放到 rd 寄存器中，将[rs2]存储到[rs1]为地址的位置，然后[rs2]=[rd]
55	原子加	AMADD: rd, rs2, rs1	4	以[rs1]为地址，读出五字节存放到 rd 寄存器中，[rs2]+=[rd]，将[rs2]存储到[rs1]位置
56	原子与	AMAND: rd, rs2, rs1	4	从[rs1]读出五字节存放到 rd 寄存器中，[rs2]&=[rd]，将[rs2]存储到[rs1]位置
57	原子或	AMOR: rd, rs2, rs1	4	从[rs1]读出五字节存放到 rd 寄存器中，[rs2]|=[rd]，将[rs2]存储到[rs1]位置
58	原子异或	AMXOR: rd, rs2, rs1	4	从[rs1]读出五字节存放到 rd 寄存器中，[rs2]|^=[rd]，将[rs2]存储到[rs1]位置
59	原子最小	AMMIN: rd, rs2, rs1	4	从[rs1]读出五字节整数存放到 rd 寄存器中，[rs2]|=min([rd],[rs2])，将[rs2]存储到[rs1]位置
60	原子无符号数最小	AMMINU: rd, rs2, rs1	4	从[rs1]读出五字节无符号整数存放到 rd 寄存器中，[rs2]|=min([rd],[rs2])，将[rs2]存储到[rs1]位置

5. 单精度浮点操作指令

浮点数操作也是通用处理器实现中必不可少的内容，由于默认的字长是 40 位，故此处的单精度不是 32 位的单精度浮点，而是 40 位单精度浮点数。浮点数格式可参考国际标准 ANSI/IEEE-754，为了与双精度浮点数无缝转换，综合考虑，可以用 1 位存符号，11 位存指数，剩下 28 位存尾数。因此指数的范围为−1022~1023，能表示的大小范围为 $\pm 2*2^{1023} \approx \pm 1.798*10^{308}$，尾数的精度可达到 $28 \cdot \log_{10}^{2} \approx 8.43$，即有 8 位有效数字。对常用工程计算来说，假设损失一半精度，最后结果仍有 4 位有效数字，基本能满足需要。

5.1. 单精度浮点数运算指令

单精度浮点数运算指令包含了 40 位浮点数加、减、乘、除、平方根指令，如表 12 所示，由于很多工程计算中有求乘积后立即相加或相减的运算，故设置了浮点乘加指令和浮点乘减指令。

Table 12. Single-precision floating-point number operation instructions
表 12. 单精度浮点数运算指令

编号	名称	格式	长	说明
61	浮点加	FADD: rd, rs1, rs2	4	计算[rd]=[rs1]+[rs2]
62	浮点减	FSUB: rd, rs1, rs2	4	计算[rd]=[rs1]-[rs2]
63	浮点乘	FMUL: rd, rs1, rs2	4	计算[rd]=[rs1]*[rs2]
64	浮点除	FDIV: rd, rs1, rs2	4	计算[rd]=[rs1]/[rs2],
18	平方根	FSQRT: rd, rs1	3	计算[rd]=sqrt([rs1])
98	浮点乘加	FMADD: rd, rs1, rs2, rs3	5	计算[rd]=[rs1]*[rs2]+[rs3]
99	浮点乘减	FMSUB: rd, rs1, rs2, rs3	5	计算[rd]=[rs1]*[rs2]-[rs3]

5.2. 单精度浮点数比较操作指令

40 位浮点数比较操作指令包含判断浮点数类型、小于、小于等于、等于、最小和最大指令，如表 13 所示，其中浮点数类型包含上溢、下溢等类型，与浮点数定义格式有关，可参考 RISC-V 开源指令集中的定义[1]。

Table 13. Single-precision floating-point number comparison operation instructions
表 13. 单精度浮点数比较操作指令

编号	名称	格式	长	说明
19	浮点数类型	FCLS: rd, rs1	3	判断[rs1]单精度浮点数类型，类型码写入 rd
65	浮点小于	FLT: rd, rs1, rs2	4	如果[rs1]<[rs2]，[rd]=1，否则为 0
66	浮点小于等于	FLE: rd, rs1, rs2	4	如果[rs1]<=[rs2]，[rd]=1，否则为 0
67	浮点等于	FEQ: rd, rs1, rs2	4	如果[rs1]==[rs2]，[rd]=1，否则为 0
68	最小值	FMIN: rd, rs1, rs2	4	将[rs1]和[rs2]中最小值放置到 rd 中
69	最大值	FMAX: rd, rs1, rs2	4	将[rs1]和[rs2]中最大值放置到 rd 中

5.3. 浮点数转换和符号操作指令

浮点数转换指令包含 40 位浮点数和 40 位有符号整数和无符号整数之间的转换操作指令，符号注入指令包含符号位的提取、取反和异或操作，如表 14 所示。由于符号位放置在最高位，操作位置相同，故也适用于双精度浮点数操作。

Table 14. Floating-point number conversion operation and sign injection instructions
表 14. 浮点数转换和符号注入指令

编号	名称	格式	长	说明
20	从整数到浮点	FWS: rd, rs	3	rs 中的整数转换为浮点数，放置到 rd
21	从浮点到整数	FSW: rd, rs	3	rs 中的浮点数转换为整数，放置到 rd
22	从无符号整数到浮点	FUWS: rd, rs	3	rs 中的无符号整数转换为浮点数，放置到 rd
23	从浮点到无符号整数	FSUW: rd, rs	3	rs 中的浮点数转换为无符号整数，放置到 rd
70	符号提取	SGJ: rd, rs1, rs2	4	提取[rs2]的符号，融合[rs1]，结果放置到 rd
71	符号取反	SGJN: rd, rs1, rs2	4	取反[rs2]的符号，融合[rs1]，结果放置到 rd
72	符号异或	SGJX: rd, rs1, rs2	4	将[rs1]和[rs2]的符号进行异或(XOR)操作，融合[rs1]，结果放置到 rd

指令编号 73、74、75、76、77、78、79 为预留的 4 字节指令。

6. 扩展指令

对于高精度计算来说，40 位整数和 40 位浮点数可能不足，故可定义 80 位双精度的整数和浮点操作。这部分指令不是所有 40 位芯片都必须实现的，故作为可选的扩展指令。扩展指令从 128 号指令开始，即操作码的最高位为 1，除了此处定义的双精度操作指令，预留的指令空间较多，需要实现特殊功能的芯片可以方便地扩展指令集。

实现双精度计算的简单方法是将通用寄存器的位宽加倍，从 40 位扩展到 80 位，原有的基本指令只

使用其中 40 位，故不受影响，这个方法的缺点是要增加硬件成本。另一个较为经济的做法是通用寄存器的位宽不变，只是将相邻的两个寄存器合并起来达到 80 位宽度，这个方法能降低硬件制造成本，缺点是双精度操作所用的通用寄存器数量减少。综合考虑，如果双精度操作使用不多，可维持原有寄存器设计，以尽量经济，如果双精度操作十分密集，则应将通用寄存器宽度加倍，以获得更好性能。

寄存器和存储器之间的载入和存储操作采用已有指令，即载入十字节的指令 LT: rd, rs1,imm，和存储十字节的指令 ST: rs2, rs1,imm，转移的数据可以是整数，也可以是浮点数，因为占用的空间大小一样，为方便使用，也可改写为伪指令。

80 位双精度浮点数参考前述的 40 位浮点数格式，可以用 1 位存符号，11 位存指数，剩下 68 位存尾数。指数的范围为−1022~1023，能表示的大小范围为 $\pm 2*2^{1023} \approx \pm 1.798*10^{308}$，尾数的精度可达到 $68 \cdot \log_{10}^{2} \approx 20.47$，即有 20 位有效数字，对高精度计算十分有用。

6.1. 双精度浮点数运算指令

双精度浮点数运算指令包含了 80 位浮点数加、减、乘、除、平方根指令，如表 15 所示，由于很多工程计算中有求乘积后立即相加或相减的运算，故设置双精度浮点乘加指令和双精度浮点乘减指令。

Table 15. Double-precision floating-point number operation instructions
表 15. 双精度浮点数运算指令

编号	名称	格式	长	说明
136	浮点加	DADD: rd, rs1, rs2	4	双精度浮点数[rd]=[rs1]+[rs2]
137	浮点减	DSUB: rd, rs1, rs2	4	双精度浮点数[rd]=[rs1]−[rs2]
138	浮点乘	DMUL: rd, rs1, rs2	4	双精度浮点数[rd]=[rs1]*[rs2]
139	浮点除	DDIV: rd, rs1, rs2	4	双精度浮点数[rd]=[rs1]/[rs2],
128	平方根	DSQRT: rd, rs1	3	双精度浮点数[rd]=sqrt([rs1])
160	浮点乘加	DMADD: rd, rs1, rs2, rs3	5	双精度浮点数[rd]=[rs1]*[rs2]+[rs3]
161	浮点乘减	DMSUB: rd, rs1, rs2, rs3	5	双精度浮点数[rd]=[rs1]*[rs2]−[rs3]

6.2. 双精度浮点数比较操作指令

双精度浮点数比较操作指令包含判断双精度浮点数类型、小于、小于等于、等于、最小和最大指令，如表 16 所示，其中双精度浮点数类型与单精度浮点数类型的判断是相似的。

Table 16. Double-precision floating-point number comparison operation instructions
表 16. 双精度浮点数比较操作指令

编号	名称	格式	长	说明
129	双精度浮点数类型	DCLS: rd, rs1	3	判断[rs1]的浮点数类型，类型码写入 rd
140	双精度浮点小于	DLT: rd, rs1, rs2	4	如果[rs1]<[rs2]，[rd]=1，否则为 0
141	双精度浮点小于等于	DLE: rd, rs1, rs2	4	如果[rs1]<=[rs2]，[rd]=1，否则为 0
142	双精度浮点等于	DEQ: rd, rs1, rs2	4	如果[rs1]==[rs2]，[rd]=1，否则为 0
143	双精度浮点最小值	DMIN: rd, rs1, rs2	4	将[rs1]和[rs2]中最小值放置到 rd 中
144	双精度浮点最大值	DMAX: rd, rs1, rs2	4	将[rs1]和[rs2]中最大值放置到 rd 中

6.3. 双精度浮点数转换指令

双精度浮点数转换指令包含80位浮点数和80位有符号长整数和无符号长整数之间的转换操作指令，如表17所示。不提供40位整数和80位双精度浮点数之间的直接转换操作，因为精度损失太大。要实现40位整数和双精度浮点之间的转换，可以先将40位整数扩展为80位整数，再转换为双精度浮点数。由于单精度浮点数和双精度浮点数的指数位宽度一样，所以它们之间的转换是无缝的，即取前40位为单精度浮点数，取完整的80位为双精度浮点数。双精度浮点数的符号操作与单精度浮点数相同，都是对最高位即符号位的处理，故无需另外指令。

Table 17. Double-precision floating-point number conversion operation instructions
表17. 双精度浮点数转换操作指令

编号	名称	格式	长	说明
130	长整数到双精度浮点	FLD: rd, rs	3	rs中的80位整数转换为双精度浮点数
131	双精度浮点到长整数	FDL: rd, rs	3	rs中的双精度浮点数转换为80位整数
132	无符号长整数到双精度浮点	FULD: rd, rs	3	rs中的80位无符号整数转换为双精度浮点数
133	双精度浮点到无符号长整数	FDUL: rd, rs	3	rs中的双精度浮点数转换为80位无符号整数

6.4. 长整数运算指令

长整数为80位二进制整数，分为有符号和无符号两类。对80位无符号整数，表示范围为$0\sim2^{80}-1$，对80位有符号数，最高位区分符号，表示范围为$-2^{79}+1\sim2^{79}-1$。长整数的运算指令包含加减、乘除、求余等功能，如表18所示。如果希望得到两个80位整数相乘的完整160位结果，可以融合两条指令连续执行“LMUL: rd, rs1, rs2; LMULH[U]: rd+2, rs1, rs2”，此处将160位结果的小端部分放在前，大端部分放在后，便于和80位整数无缝转换。另提供复制80位内容的操作指令LMOV和长整数自加操作指令LINCB。

Table 18. Long integer operation instructions
表18. 长整数运算指令

编号	名称	格式	长	说明
145	长整数加法	LADD: rd, rs1, rs2	4	对长整数计算[rd]=[rs1]+[rs2]
146	长整数减法	LSUB: rd, rs1, rs2	4	对长整数计算[rd]=[rs1]-[rs2]
134	长整数自加操作	LINCB: rd, imm	3	对长整数计算[rd]+=imm
135	复制长整数内容	LMOV: rd, rs1	3	将rs1中80位内容复制到rd寄存器
147	长整数相乘低位	LMUL: rd, rs1, rs2	4	rd保存长整数[rs1]*[rs2]的低80位
148	长整数相乘高位	LMULH: rd, rs1, rs2	4	rd保存长整数[rs1]*[rs2]的高80位
149	无符号长整数相乘高位	LMULHU: rd, rs1, rs2	4	rd保存无符号长整数[rs1]*[rs2]的高80位
150	相除	LDIV: rd, rs1, rs2	4	长整数相除[rd]=[rs1]/[rs2]
151	无符号长整数相除	LDIVU: rd, rs1, rs2	4	无符号长整数相除[rd]=[rs1]/[rs2]
152	求余	LREM: rd, rs1, rs2	4	长整数求余[rd]=[rs1]%[rs2]
153	无符号长整数求余	LREMU: rd, rs1, rs2	4	无符号长整数求余[rd]=[rs1]%[rs2]

6.5. 长整数原子存储操作指令

长整数原子存储指令对 80 位的长整数进行原子性操作，便于多个核心并行处理长整数，支持的操作包括原子比较交换、交换、加减和逻辑操作，如表 19 所示。

Table 19. Long integer atomic storage operation instructions
表 19. 长整数原子存储操作指令

编号	名称	格式	长	说明
154	长整数原子比较交换	AMLCMPS: rd, rs2, rs1	4	以[rs1]为地址读出长整数，与[rd]比较，相等则将[rs2]存储到[rs1]位置，不相等则将 rd 设为读出值
155	长整数原子交换	AMLSWAP: rd, rs2, rs1	4	以[rs1]为地址，读出长整数存放到 rd 寄存器中，将[rs2]存储到[rs1]为地址的位置，然后[rs2]=[rd]
156	长整数原子加	AMLADD: rd, rs2, rs1	4	以[rs1]为地址，读出长整数存放到 rd 寄存器中，[rs2]+=[rd]，将[rs2]存储到[rs1]位置
157	长整数原子与	AMLAND: rd, rs2, rs1	4	从[rs1]读出长整数存放到 rd 寄存器中，[rs2]&=[rd]，将[rs2]存储到[rs1]位置
158	长整数原子或	AMLOR: rd, rs2, rs1	4	从[rs1]读出长整数存放到 rd 寄存器中，[rs2]|=[rd]，将[rs2]存储到[rs1]位置
159	长整数原子异或	AMLXOR: rd, rs2, rs1	4	从[rs1]读出长整数存放到 rd 寄存器中，[rs2]^=[rd]，将[rs2]存储到[rs1]位置

7. 结语

在手机、平板、笔记本电脑等应用领域，x86 芯片和 ARM 芯片正在互相渗透，而 40 位处理器能够满足主流应用要求且经济性可能更好，可望在市场缝隙中获得一线生机。借鉴并杂合各类处理器指令集的优点，如 x86、ARM、RISC-V、SPARC 等，本文尝试对 40 位处理器的指令集架构进行了初步设计，以便于抛砖引玉，为中国芯片产业发展提供一些思路。定义的 40 位处理器指令集是以字节为单位的可变长度精简指令集，解码规则简单，便于硬件实现，可用性和扩展性好。该指令集定义了从 0 号到 99 号共 100 条基本指令，其中已定义指令 93 条，预留 7 条四字节指令，另提供为双精度计算服务的扩展指令 34 条，编号从 128 号到 161 号，全部为已定义指令。总的已定义指令共 127 条，还留有较大的指令扩充空间。为推广使用，该指令集免费开源，不妨命名为熊猫处理器精简指令集，以突出中国特色。

指令集作为硬件和软件之间的接口，既是相对稳定的，又不是一成不变的，必须适应软硬件的发展而扩充。本文提出的 40 位处理器的指令集没有历史包袱，便于全新设计，但能否满足信息处理的现代需求和未来发展，还需要实践来回答。该指令集架构设计为了适应硬件体系的实现要求，还有许多细节问题尚待讨论和细化，如寄存器窗口如何设计更为优化，中断系统的定义和控制，甚至浮点数格式的定义和控制等等，都可以有不同实现方案。要使得定义的指令集能够实际运作，还需要各方有识之士共同努力，将这一叶新苗培育成长，扎根于广阔的芯片市场，为中国和世界的信息产业服务。

参考文献

[1] 胡振波. 手把手教你设计 CPU-RISC-V 处理器篇[M]. 北京: 人民邮电出版社, 2018.

[2] 雷思磊. RISC-V 架构的开源处理器及 SOC 研究综述[J]. 单片机与嵌入式系统应用, 2017(2): 56-60.

[3] 寇晓斌, 杨琴, 王亮亮. 主流处理器体系结构与架构发展现状综述[J]. 微型机与应用, 2014, 33(16): 1-5.

[4] 刘文胜, 荣广颐. RISC 技术与 SPARC 结构[J]. 计算机工程与应用, 1988(12): 1-5.
[5] 许俊贤, 张祥, 李童. SPARC 体系结构[J]. 计算机研究与发展, 1990(11): 1-28.
[6] 覃辉, 于立新, 刘鹏. SPARC 微处理器综述[J]. 电子产品世界, 2010, 17(7): 71-72.
[7] 尤念祖. RISC 技术与工程工作站—从 SUN 公司 SPARC 谈起[J]. 小型微型计算机系统, 1988, 9(11): 1-6.