# 数字逻辑与计算机组织结构实验指导书（下册）

# RISC-V架构及指令系统

## RISC-V架构简介

芯片是内含集成电路的硅片，是半导体产业的核心。处理器芯片因设计难度大、复杂度高而被公认为芯片产业中最复杂的一类芯片。我国是集成电路进口大国，集成电路已超过石油、铁矿等，成为我国的第一大进口产品，同时，国产芯片的市场占有率不足5%。以2021年为例，我国共进口6355亿个集成电路，进口额高达4326亿美元，其中处理器与控制器芯片进口额超过2034亿美元，占比47%。

从人才培养角度看，我国优秀处理器芯片人才储备严重不足。科教兴国，加快处理器芯片人才培养规模与速度已是迫在眉睫。

中国科学院大学发起的“一生一芯”项目是基于开源RISC-V架构的一种贯通课程的实践型开放式大规模人才培养计划，致力于为我国培养处理器芯片设计人才。本书与“一生一芯”项目组合作，基于开源的RISC-V架构和“龙芯杯”全国大学生系统能力大赛团队赛二等奖获奖作品，设计RISC-V架构流水线实验项目。通过这些实验项目，培养学生设计简单处理器的能力和科技报国的情怀。

计算机软硬件从底层到顶层分为数字逻辑电路、处理器微架构、指令集架构、操作系统、编译系统、应用系统等若干层次。指令集架构处于计算机软硬件的分界线，它定义了指令集的功能，向上支持各层软件的执行。RISC-V是由加州大学伯克利分校的David A. Patterson教授团队研制的开源精简指令集架构，其中V表示第5代。RISC-V是一个模块化的指令集架构，包含若干个面向不同应用场景的功能指令模块，功能指令模块也称为扩展指令集。RISC-V被《MIT科技评论》评选为2023 年全球十大突破性技术之一，评价中写道“芯片设计正走向开放，灵活、开源的RISC-V有望成为改变一切的芯片设计”。发布RISC-V指令集手册和架构文档的官网网址是：[www.riscv.org](http://www.riscv.org)。

David A. Patterson教授团队提出了ISA （Instruction Set Architecture，指令集架构）设计的7个评价指标：成本、简洁、性能、架构和实现分离、提升空间、代码大小、易于编程/编译/链接。RISC-V架构的设计很好地满足了上述7个评价指标。RISC-V架构的设计目标是成为一款通用的ISA：

1. 适合设计各种规模的处理器，从最小的嵌入式控制器到最快的高性能计算机。
2. 兼容各种流行的软件栈和编程语言。
3. 适用于所有实现技术，包括FPGA（Field-Programmable Gate Array，现场可编程逻辑门阵列）、ASIC（Application-Specific Integrated Circuit，专用集成电路）、全定制芯片，甚至未来的制造元件技术。
4. 能用于高效实现所有微体系结构，包括微程序或硬布线控制、顺序、解耦或乱序流水线、单发射或超标量等。
5. 支持高度定制化，成为定制加速器的基础，以应对摩尔定律的放缓。
6. 具有稳定性，基础指令集不改变。

RISC-V的常用扩展指令集如表1-1所示。

* + - * 1. RISC-V的扩展指令集

|  |  |  |
| --- | --- | --- |
| **模块符号** | **扩展指令集（功能指令模块）名称** | **指令字长** |
| I | 基础整数指令集 | 32 |
| M | 乘法和除法指令集 | 32 |
| A | 原子指令集 | 32 |
| F | 单精度浮点指令集 | 32 |
| D | 双精度浮点指令集（与F模块一起使用） | 32 |
| Zicsr | 控制状态寄存器指令集 | 32 |
| Zifencei | 指令流屏障指令 | 32 |
| C | 压缩指令集 | 16 |

设计任何一个RISC-V计算机都必须包含**基础整数指令集**（I），在此基础上添加或不添加其他扩展指令集。如果在I指令集基础上，还添加了M、A、F、D、Zicsr、Zifencei指令集，那么可记为通用指令集G。例如，某64位的RISC-V架构计算机，其指令集除RV64I外，还包含M、A、F、D、Zicsr和Zifenci指令集，则可记为RV64IMAFDZicsr\_Zifencei，亦可记为RV64G。

无论机器字长多少位，RISC-V指令字长都是32位（C指令集除外）。存储器按字节编址，可选采用小端模式或大端模式。指令码皆以小端模式存储，**本书所述RV64IMA****Zicsr\_Zifencei模型机的存储遵循小端模式**。

## 通用寄存器堆GPRs

如表1-2所示，RV64I含32个64位整数通用寄存器，此外还有一个64位的程序计数器PC，用以指示当前指令地址。RISC-V规范的ABI接口（Application Binary Interface，应用程序二进制接口）规定了表1-2中通用整数寄存器和浮点数寄存器的别名，以及子程序调用时每个寄存器的保存使用约定。为支持F和D扩展指令集，还应该配有32个64位的浮点寄存器f0~f31。本书设计的RV64IMAZicsr\_Zifencei CPU不包含浮点数运算功能，因此不介绍F和D扩展指令集，也不详解浮点寄存器f0~f31。

* + - * 1. RV64I的通用寄存器与调用约定

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **寄存器** | **ABI别名** | **用途描述** | **由被调用者保留？** | **由谁保存** |
| x0 | zero | 硬编码零 | — | — |
| x1 | ra | 返回地址 | 否 | 调用者 |
| x2 | sp | 栈指针 | 是 | 被调用者 |
| x3 | gp | 全局指针 | — | — |
| x4 | tp | 线程指针 | — | — |
| x5 | t0 | 临时寄存器/备用链接寄存器 | 否 | 调用者 |
| x6~x7 | t1~t2 | 临时寄存器 | 否 | 调用者 |
| x8 | s0/fp | 保存寄存器/帧指针 | 是 | 被调用者 |
| x9 | s1 | 保存寄存器 | 是 | 被调用者 |
| x10~x11 | a0~a1 | 函数参数/返回值 | 否 | 调用者 |
| x12~x17 | a2~a7 | 函数参数 | 否 | 调用者 |
| x18~x27 | s2~s11 | 保存寄存器 | 是 | 被调用者 |
| x28~x31 | t3~t6 | 临时寄存器 | 否 | 调用者 |
| f0~f7 | ft0~ft7 | 浮点临时寄存器 | 否 | 调用者 |
| f8~f9 | fs0~fs1 | 浮点保存寄存器 | 是 | 被调用者 |
| f10~f11 | fa0~fa1 | 浮点函数参数/返回值 | 否 | 调用者 |
| f12~f17 | fa2~fa7 | 浮点函数参数 | 否 | 调用者 |
| f18~f27 | fs2~fs11 | 浮点保存寄存器 | 是 | 被调用者 |
| f28~f31 | ft8~ft11 | 浮点临时寄存器 | 否 | 调用者 |

对整数寄存器x0~x31的解释如下：

1. 零寄存器x0/zero：硬连线零，任何时候都读出零，不支持写入。
2. 子程序调用相关寄存器：子程序也称为函数或过程。调用时可能需要参数传递数值和返回结果，也可能无返回值。
3. 返回地址寄存器x1/ra：，在执行子程序调用指令时，将主程序中的返回地址，即调用指令的下一条指令的地址（PC+4）写入寄存器ra。当子程序返回时，将ra内容再写回到程序寄存器PC中，从而返回主程序断点。
4. 函数参数寄存器x10~x17/a0~a7：用于函数调用时的参数传递。若参数的数量超出8个，寄存器不够用时，通过内存堆栈区传参。
5. 函数返回值寄存器x10~x11/a0~a1：用于传递调用后的返回值。
6. 保存寄存器x8~x9、x18~x27/ s0~s11：该寄存器在被子程序使用前后须保持原值不变。子程序在使用s0~s11寄存器前，先将其压入堆栈保存；在使用完毕后，返回主程序之前，再从堆栈中恢复这些寄存器的值。
7. 临时寄存器x5~x7、x28~x31/ t0~t6：临时寄存器在子程序调用前已被主程序保存，因此可以被子程序任意使用，而无需保存与恢复。
8. 堆栈指针x2/sp：保存内存中堆栈的栈顶地址。

## 特权模式和CSR寄存器

如果计算机系统中只有1种特权模式，所有程序都可以自由访问硬件平台，那么将无法防止含有错误的应用程序运行带来的后果。为了保护系统免受不可信代码带来的危害，计算机系统应该提供隔离机制来隔离地运行不受信任的进程，譬如，禁止不可信代码越界访问操作系统内核代码的内存空间等。采用权限模式的方式可以容易地实现隔离机制。

对计算机系统而言，应用程序代码是不可信代码，运行在用户模式下，除用户模式以外的其他模式称为**特权模式**。特权模式的权限均高于用户模式。高特权模式能访问低特权模式的所有功能，同时还具备若干低特权模式下不可用的额外功能，如中断处理和I/O 操作。处理器通常在最低特权模式下运行，当发生中断和异常时，则将控制权转移到更高的特权模式。

RISC-V支持的特权模式如表1-3所示。M模式（机器模式）是RISC-V中hart（hardware thread，硬件线程）可以执行的最高权限模式，具有访问所有资源的权限，用于运行最可信的代码。例如，M模式下hart对内存和I/O拥有完全的使用权。M模式是所有RISC-V处理器都必须实现的权限模式。处理器是否支持其他模式则是可选的。

S模式（监管模式）为Linux、Windows 等支持虚拟存储器的操作系统提供支持。M和S这两种模式的特权级均高于用户模式U。

例如，某RV32I处理器仅支持M模式。另一个RV64IMAZicsr\_Zifencei处理器支持M和U两种权限模式，其上能够运行FreeRTOS实时操作系统，实时操作系统无虚拟存储器，也不支持虚拟地址，仅使用物理地址。再有一个RV64G处理器支持M、U和S三种权限模式，能够运行Linux操作系统，支持虚拟存储器，虚拟地址由MMU（Memory Management Unit，内存管理单元）转换成物理地址。

* + - * 1. 特权模式及编码

|  |  |  |  |
| --- | --- | --- | --- |
| **编码** | **模式名称** | **简写** | **说明** |
| 00 | User 用户模式 | U | 应用程序代码运行 |
| 01 | Supervisor 监管模式 | S | 为支持虚拟存储器的操作系统提供支持 |
| 10 | — | — | 保留 |
| 11 | Machine 机器模式 | M | 运行最可信的代码 |

RISC-V处理器的每个hart拥有若干个与机器字长等宽的CSR寄存器（Control and Status Register，控制和状态寄存器），CSR寄存器地址的长度是12位。RISC-V处理器通过CSR寄存器来配合实现特权模式和中断/异常机制。CSR寄存器用于配置或者记录处理器硬件的运行状态。通过使用控制状态寄存器指令来访问CSR寄存器。

RISC-V手册版本更新迭代频繁，不同版本手册中CSR寄存器的格式和位定义不尽相同，以下给出常用的CSR寄存器，它们的具体格式和使用方法将在第6章里介绍。

#### 异常/中断处理必须用到的CSR寄存器

* 中断挂起寄存器mip：用于记录当前的中断请求。
* 中断使能寄存器mie：用于维护处理器的中断使能状态。
* 原因寄存器mcause/scause：用于指示M模式/S模式下发生异常和中断的原因。 mcause/scause的最高位在发生中断时置1；发生同步异常时置0。低位部分标识中断或异常的具体原因。
* 状态寄存器mstatus/sstatus：主要用于M模式/S模式下模式状态的保存和设置。
* 自陷向量基址寄存器mtvec /stvec ：用于存放M模式/S模式下发生异常时处理器跳转的基址。
* 异常值寄存器mtval /stval：用于在M模式/S模式下存放当前自陷相关的额外信息，如地址异常的故障地址、非法指令、异常的指令，发生其他异常时其值为0。
* 异常PC寄存器mepc /sepc：用于M模式/S模式下指向发生异常的指令对应的PC。
* 草稿寄存器mscratch 和sscratch：用于M模式/S模式下向异常处理程序提供一个字的临时存储，即一个空闲可用的寄存器。

#### 标识处理器特性的CSR寄存器

* 机器指令集架构寄存器misa：指示处理器的机器字长（32/64 /128 位），指示处理器支持的扩展。
* 厂商识别码寄存器mvendorid：指示处理器核供应商的JEDEC 标准制造商识别码。
* 机器架构识别码寄存器marchid：指示基础微架构。将mvendorid 和marchid组合在一起，可唯一识别所实现的微架构。
* 机器实现识别码寄存器mimpid：指示marchid 所示基础微架构的实现版本。
* 硬件线程识别码寄存器mhartid：给出正在运行的硬件线程编号。

#### 用于性能测量的CSR寄存器

* 机器时间寄存器mtime：它是一个64 位实时计数器。
* 机器时间比较寄存器mtimecmp：当mtime ≥mtimecmp时触发中断。
* 计数器使能寄存器mcounteren /scounteren ：32 位，用于M模式/S模式下控制硬件性能监视器CSR 是否在更低特权级下可用。
* 硬件性能监视器寄存器mcycle，minstret，mhpmcounter3⋯⋯mhpmcounter31：共31 个，对时钟周期、已执行指令以及软件指定的最多29 个事件（通过mhpmevent3 到mhpmevent31 指定）进行计数。

## 指令集和指令格式

RISC-V处理器支持6种机器指令格式，分别是R型、I型、S型、B型、U型和J型，如表1-4所示。未实现C拓展的RISC-V处理器**指令字长固定32位**，且存储必须**按字（32位）对齐**。因此，本书设计的64位RISC-V处理器的I、M、A、Zicsr、Zifencei指令集中的所有指令长度也都是32位，在指令存储器中按照字（4个字节）边界对齐。表1-4中，opcode表示指令的7位操作码；rs1、rs2表示两个源寄存器，其寄存器地址（即寄存器号）占5位；rd表示目的寄存器，其寄存器号也是5位；func3表示此段功能码长度是3位；func7表示此段功能码长度是7位；imm表示立即数，imm[11:0]表示12位的立即数，也可写成imm12；imm[11:5]表示7位的立即数，也可写成imm7；imm[12,10:5]和imm[4:1,11]表示两段分别为7位和5位的立即数，它们拼接成一个12位的立即数imm[12:1]，该立即数里每个二进制位是按imm[12,10:5]和imm[4:1,11]所示序号来排列，请参考1.5小节的立即数寻址方式。实际上B型格式的立即数是S型格式立即数的旋转。同理，J型格式的立即数是U型格式立即数的旋转。

* + - * 1. RISC-V的6种指令格式

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |  |
| funct7（7位） | rs2（5位） | rs1（5位） | funct3（3位） | rd（5位） | opcode（7位） | R型 |
| imm[11:0] （12位） | | rs1（5位） | funct3（3位） | rd（5位） | opcode（7位） | I型 |
| imm[11:5] | rs2（5位） | rs1（5位） | funct3（3位） | imm[4:0（5位）] | opcode（7位） | S型 |
| imm[12|10:5] | rs2（5位） | rs1（5位） | funct3（3位） | imm[4:1|11] （5位） | opcode（7位） | B型 |
| imm[31:12] （20位） | | | | rd（5位） | opcode（7位） | U型 |
| imm[20|10:1|11|19:12] （20位） | | | | rd（5位） | opcode（7位） | J型 |

以下简要介绍本书将要实现的64位RISC-V处理器的I、M、A、Zicsr、Zifencei指令集中的指令格式和功能。在第5、6章实验项目中详细介绍怎样实现这些指令。

### RV64I指令集

RV64I指令集里包含整数运算类指令、访存指令、转移指令和控制状态类指令。其中，整数运算类指令包含有移位指令、算术运算指令、逻辑运算指令和比较指令。转移指令包含有（条件）分支跳转指令、（无条件）跳转和链接指令。控制状态类指令包括同步指令和环境调试/断点指令。

#### 整数运算类指令

RV64I整数运算类指令分别有R型和I型两种格式。表1-5和表1-6分别给出了R型和I型格式下的移位指令、算术运算指令、逻辑运算指令和比较指令。表1-7给出了U型格式下的算术运算指令。在表1-5～表1-7中，

①SEXT64（n）表示将数据n按符号扩展到64位；

②x<<y表示对x左移y位，以0填充低位；

③x>>y表示对x右移y位，以0填充高位；

④x>> S y表示对x算术右移y位，以符号位填充高位；

⑤{imm20,12{0}}表示一个拼接得到的32位数，其高20位是立即数imm20，低12位都是0。

* + - * 1. R型格式的整数运算指令

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **移位指令** | | | | | | | |
| **位数**  **指令** | **7位** | **5位** | **5位** | **3位** | **5位** | **7位** | **R型格式移位指令的功能** |
| **funct7** | **rs2** | **rs1** | **funct3** | **rd** | **opcode** |
| sll rd,rs1,rs2 | 0000000 | rs2 | rs1 | 001 | rd | 0110011 | 左移，低位补0：rs1<<（rs2的低6位）→rd |
| sllw rd,rs1,rs2 | 0000000 | rs2 | rs1 | 001 | rd | 0111011 | 左移字，低位补0：SEXT64(（rs1的低32位）<<（rs2的低5位）)→rd |
| srl rd,rs1,rs2 | 0000000 | rs2 | rs1 | 101 | rd | 0110011 | 逻辑右移，高位补0：rs1>>(rs2的低6位）→rd |
| srlw rd,rs1,rs2 | 0000000 | rs2 | rs1 | 101 | rd | 0111011 | 逻辑右移字，高位补0：SEXT64(（rs1的低32位）>>(rs2的低5位））→rd |
| sra rd,rs1,rs2 | 0100000 | rs2 | rs1 | 101 | rd | 0110011 | 算术右移，高位补符号位: rs1>>S（rs2的低6位）→rd |
| sraw rd,rs1,rs2 | 0100000 | rs2 | rs1 | 101 | rd | 0111011 | 算术右移字，高位补rs1[31]: SEXT64(（rs1的低32位)>> S（rs2的低5位）)→rd |
| **算术运算指令** | | | | | | | |
| **位数**  **指令** | **7位** | **5位** | **5位** | **3位** | **5位** | **7位** | **R型格式算术运算指令的功能** |
| **funct7** | **rs2** | **rs1** | **funct3** | **rd** | **opcode** |
| add rd,rs1,rs2 | 0000000 | rs2 | rs1 | 000 | rd | 0110011 | 加： rs1+rs2→rd |
| addw rd,rs1,rs2 | 0000000 | rs2 | rs1 | 000 | rd | 0111011 | 先加，截取低32位后再符号扩展：  SEXT64(（rs1+rs2）[31:0])→rd |
| sub rd,rs1,rs2 | 0100000 | rs2 | rs1 | 000 | rd | 0110011 | 减： rs1- rs2→rd |
| subw rd,rs1,rs2 | 0100000 | rs2 | rs1 | 000 | rd | 0111011 | 先加，截取低32位后再符号扩展：  SEXT64(（rs1-rs2）[31:0])→rd |
| **逻辑运算指令** | | | | | | | |
| **位数**  **指令** | **7位** | **5位** | **5位** | **3位** | **5位** | **7位** | **R型格式逻辑运算指令的功能** |
| **funct7** | **rs2** | **rs1** | **funct3** | **rd** | **opcode** |
| xor rd,rs1,rs2 | 0000000 | rs2 | rs1 | 100 | rd | 0110011 | 异或运算：rs1⊕rs2→rd |
| or rd,rs1,rs2 | 0000000 | rs2 | rs1 | 110 | rd | 0110011 | 或运算: rs1 | rs2→rd |
| and rd,rs1,rs2 | 0000000 | rs2 | rs1 | 111 | rd | 0110011 | 与运算：rs1 & rs2→rd |
| **比较指令** | | | | | | | |
| **位数**  **指令** | **7位** | **5位** | **5位** | **3位** | **5位** | **7位** | **R型格式比较指令的功能** |
| **funct7** | **rs2** | **rs1** | **funct3** | **rd** | **opcode** |
| slt rd,rs1,rs2 | 0000000 | rs2 | rs1 | 010 | rd | 0110011 | 有符号数比较，小于则置数：  if(rs1<rs2) then 1→rd else 0→rd |
| sltu rd,rs1,rs2 | 0000000 | rs2 | rs1 | 011 | rd | 0110011 | 无符号数比较，小于则置数：if(rs1<rs2) then 1→rd else 0→rd |

* + - * 1. I型格式的整数运算指令

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 移位指令 | | | | | | | | | | | | | | | | | | |
| 位数  指令 | **12位** | | | | | **5位** | | | **3位** | | | **5位** | | | **7位** | | | **I 型格式的移位指令功能** |
| **imm12** | | | | | **rs1** | | | **funct3** | | | **rd** | | | **opcode** | | |
| slli rd,rs1,shamt | 000000 | | shamt | | | rs1 | | | 001 | | | rd | | | 0010011 | | | 逻辑左移，低位补0: rs1<<shamt→rd |
| slliw rd,rs1,shamt | 000000 | | shamt | | | rs1 | | | 001 | | | rd | | | 0011011 | | | 逻辑左移低位补0后截取低32位，再进行符号扩展，仅当shamt[5]=0时才有效:  SEXT64(（ rs1<<shamt）[31:0])→rd |
| srli rd,rs1,shamt | 000000 | | shamt | | | rs1 | | | 101 | | | rd | | | 0010011 | | | 逻辑右移，高位补0: rs1>>U shamt→rd |
| srliw rd,rs1,shamt | 000000 | | shamt | | | rs1 | | | 101 | | | rd | | | 0011011 | | | 逻辑右移高位补0后截取低32位，再进行符号扩展，仅当shamt[5]=0时才有效:  SEXT64(（ rs1>>U shamt）[31:0]）→rd |
| srai rd,rs1,shamt | 010000 | | shamt | | | rs1 | | | 101 | | | rd | | | 0010011 | | | 算术右移，高位补符号位: rs1>> S shamt→rd |
| sraiw rd,rs1,shamt | 010000 | | shamt | | | rs1 | | | 101 | | | rd | | | 0011011 | | | rs1低32位算术右移，高位补rs1[31]，再符号扩展，仅当shamt[5]=0时才有效:  SEXT64（（ rs1[31:0]>>S shamt）→rd |
| 算术运算指令 | | | | | | | | | | | | | | | | | | |
| 位数  指令 | | **12位** | | | **5位** | | | **3位** | | | **5位** | | | **7位** | | | **I 型格式的算术运算指令功能** | |
| **imm12** | | | **rs1** | | | **funct3** | | | **rd** | | | **opcode** | | |
| addi rd,rs1,imm12 | | imm[11:0] | | | rs1 | | | 000 | | | rd | | | 0010011 | | | 立即数加法: rs1+SEXT64(imm12)→rd | |
| addiw rd,rs1,imm12 | | imm[11:0] | | | rs1 | | | 000 | | | rd | | | 0011011 | | | 先符号扩展加，截取低32位后再符号扩展:  SEXT64 ((rs1+SEXT64(imm12))[31:0])→rd | |
| 逻辑运算指令 | | | | | | | | | | | | | | | | | | |
| 位数  指令 | | **12位** | | | **5位** | | | **3位** | | | **5位** | | **7位** | | | **I 型格式的逻辑运算指令功能** | | |
| **imm12** | | | **rs1** | | | **funct3** | | | **rd** | | **opcode** | | |
| xori rd,rs1,imm12 | | imm[11:0] | | | rs1 | | | 100 | | | rd | | 0010011 | | | 立即数异或运算: rs1⊕SEXT64(imm12)→rd | | |
| ori rd,rs1,imm12 | | imm[11:0] | | | rs1 | | | 110 | | | rd | | 0010011 | | | 立即数或运算: rs1 | SEXT64(imm12)→rd | | |
| andi rd,rs1,imm12 | | imm[11:0] | | | rs1 | | | 111 | | | rd | | 0010011 | | | 立即数与运算:rs1 & SEXT64(imm12)→rd | | |
| 比较指令 | | | | | | | | | | | | | | | | | | |
| 位数  指令 | | **12位** | | **5位** | | | **3位** | | | **5位** | | | **7位** | | | **I 型格式的比较指令功能** | | |
| **imm12** | | **rs1** | | | **funct3** | | | **rd** | | | **opcode** | | |
| slti rd,rs1,imm12 | | imm[11:0] | | rs1 | | | 010 | | | rd | | | 0010011 | | | 有符号数比较，小于则置数:  if(rs1<SEXT64(imm12)) then 1→rd else 0→rd | | |
| sltiu rd,rs1,imm12 | | imm[11:0] | | rs1 | | | 011 | | | rd | | | 0010011 | | | 无符号数比较，小于则置数：  if(rs1<SEXT64(imm12)) then 1→rd else 0→rd | | |

U型格式的算术运算指令完成对立即数移位并传送的功能，因此也可称为立即数传送指令。

* + - * 1. U型格式的算术运算指令

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **算术运算指令** | | | | |
| **位数**  **指令** | **20位** | **5位** | **7位** | **U 型格式的算术运算指令功能** |
| **imm20** | **rd** | **opcode** |
| lui rd,imm20 | imm[31:12] | rd | 0110111 | 高位立即数加载:  SEXT64{imm20,12{0}}→rd  //先对20位立即数左移12位，将低12位清零，然后进行符号扩展 |
| auipc rd,imm20 | imm[31:12] | rd | 0010111 | PC加偏移量:  PC+SEXT64{imm20,12{0}}→rd  //先对20位立即数左移12位，将低12位清零，然后进行符号扩展后，与PC相加 |

#### 访存指令

RV64I的访存指令采用I型和S型格式。CPU对存储器有读和写两种操作，表1-8给出了I型格式的访存读数指令和S型格式的访存写数指令。在表1-8中，

①UEXT64（n）表示对数据n进行零扩展到64位，即高位填充0；

②offset代表存储器地址的偏移量。

访存读数指令的偏移量是12位offset12；访存写数指令的偏移量是7位的offset7。无论读还是写，计算存储器地址的时候，都以rs1寄存器的内容为基地址，加上offset偏移量得到目标访存地址，即目标访存地址EA= rs1+SEXT64(offset)。在支持虚拟存储器的系统中，EA就是CPU给出的访存虚拟地址；在不支持虚拟存储器的系统中，EA是CPU给出的访存物理地址。主存储器按字节编址，lb/lh/lw/ld指令分别按地址EA从存储器中读出1/2/4/8个字节的数据，对取出的数据进行符号扩展到64位后写入目的寄存器rd。ld指令读出8个字节，因此读出的数据不需要符号扩展，直接存入rd。lbu/lhu/lwu指令与lb/lh/lw指令的区别在于：对于从内存读到的数据，lbu/lhu/lwu指令采用零扩展，而lb/lh/lw指令采用符号扩展。

* + - * 1. I型格式的访存指令

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **访存读数指令** | | | | | | | | | | | |
| **位数**  **指令** | **12位** | **5位** | | **3位** | | **5位** | | **7位** | | **I型格式访存读数指令的功能** | |
| **offset12** | **rs1** | | **funct3** | | **rd** | | **opcode** | |
| lb rd, imm12(rs1) | offset[11:0] | rs1 | | 000 | | rd | | 0000011 | | 取字节数据：  SEXT64(Mem[rs1+SEXT64(offset)][7:0])→rd | |
| lh rd, imm12(rs1) | offset[11:0] | rs1 | | 001 | | rd | | 0000011 | | 取半字数据：  SEXT64(Mem[rs1+SEXT64(offset)][15:0])→rd | |
| lw rd, imm12(rs1) | offset[11:0] | rs1 | | 010 | | rd | | 0000011 | | 取字数据：  SEXT64(Mem[rs1+SEXT64(offset)][31:0])→rd | |
| ld rd, imm12(rs1) | offset[11:0] | rs1 | | 011 | | rd | | 0000011 | | 取双字数据：  Mem[rs1+SEXT64(offset)]→rd | |
| lbu rd, imm12(rs1) | offset[11:0] | rs1 | | 100 | | rd | | 0000011 | | 取无符号的字节数据：  UEXT64(Mem[rs1+SEXT64(offset)][7:0])→rd | |
| lhu rd, imm12(rs1) | offset[11:0] | rs1 | | 101 | | rd | | 0000011 | | 取无符号的半字数据：  UEXT64(Mem[rs1+SEXT64(offset)][15:0])→rd | |
| lwu rd, imm12(rs1) | offset[11:0] | rs1 | | 110 | | rd | | 0000011 | | 取无符号的字数据：  UEXT64(Mem[rs1+SEXT64(offset)][31:0])→rd | |
| **访存写数指令** | | | | | | | | | | | |
| **位数**  **指令** | **7位** | **5位** | **5位** | | **3位** | | **5位** | | **7位** | | **S型格式访存写数指令的功能** |
| **offset7** | **rs2** | **rs1** | | **funct3** | | **offset5** | | **opcode** | |
| sb rs2, offset12(rs1) | offset[11:5] | rs2 | rs1 | | 000 | | offset[4:0] | | 0100011 | | 存字节数据，将rs2的低8位存入存储器:  rs2[7:0]→Mem[rs1+SEXT64(offset[11:0])] |
| sh rs2, offset 12(rs1) | offset[11:5] | rs2 | rs1 | | 001 | | offset[4:0] | | 0100011 | | 存半字数据，将rs2的低16位存入存储器:  rs2[15:0]→Mem[rs1+SEXT64(offset[11:0])] |
| sw rs2,  offset 12(rs1) | offset[11:5] | rs2 | rs1 | | 010 | | offset[4:0] | | 0100011 | | 存字数据，将rs2的低32位存入存储器:  rs2[31:0]→Mem[rs1+SEXT64(offset[11:0])] |
| sd rs2, offset 12(rs1) | offset[11:5] | rs2 | rs1 | | 011 | | offset[4:0] | | 0100011 | | 存双字数据，将rs2存入存储器:  rs2→Mem[rs1+SEXT64(offset[11:0])] |

#### 转移指令

RV64I的转移指令有B型的（条件）分支跳转指令、J型和I型格式的（无条件）跳转和链接指令。表1-9、出了B型、J型和I型格式的转移指令。在表1-9中，B型格式的12位长偏移量offset的拼接方式是：offset={I31|I7|I[30:25]|I[11:8]}，其中In代表指令码的第n位。同理，J型格式的20位offset拼接方式是：offset={I31|I[19:12]|I[20] I[30:21]}。如此编排的原因是硬件实现微架构时能够节约数据选择器的数量，从而符合RISC-V架构的硬件极简设计原则。C指令集里的压缩指令长度为16位，是RISC-V所有指令集里最短的指令。压缩指令按照半字（16位）地址对齐，因此系统里PC的最低位总是0。表1-9里offset<<1表示将offset左移1位使得最低位置0，从而保证地址按照2字节对齐。I型格式的jalr rd,offset12(rs1)并不执行offset<<1，而是将转移地址通过&～1运算来置最低位为0，从而保证地址按照2字节对齐。

从表1-9可知，B型格式的转移指令都是条件转移指令；I型和J型格式的转移指令都是无条件转移指令，其中，jal指令用于子程序调用；jalr指令用于子程序返回。

* + - * 1. 转移指令

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **分支跳转指令** | | | | | | | |
| **位数**  **指令** | **7位** | **5位** | **5位** | **3位** | **5位** | **7位** | **B型格式分支跳转指令的功能** |
| **offset[12|10:5]** | **rs2** | **rs1** | **funct3** | **offset[4:1|11]** | **opcode** |
| beq rs1,rs2,offset12 | offset[12|10:5] | rs2 | rs1 | 000 | offset[4:1|11] | 1100011 | 相等时跳转：if(rs1=rs2) then  PC+SEXT64({offset<<1})→PC |
| bne rs1,rs2,offset12 | offset[12|10:5] | rs2 | rs1 | 001 | offset[4:1|11] | 1100011 | 不相等时跳转：if(rs1≠rs2) then  PC+SEXT64 ({offset<<1})→PC |
| blt rs1,rs2,offset12 | offset[12|10:5] | rs2 | rs1 | 100 | offset[4:1|11] | 1100011 | 小于时跳转：if(rs1＜rs2) then  PC+SEXT64 ({offset<<1})→PC |
| bge rs1,rs2,offset12 | offset[12|10:5] | rs2 | rs1 | 101 | offset[4:1|11] | 1100011 | 大于等于时跳转：if(rs1≥rs2) then  PC+SEXT64 ({offset<<1})→PC |
| bltu rs1,rs2,offset12 | offset[12|10:5] | rs2 | rs1 | 110 | offset[4:1|11] | 1100011 | 无符号比较小于时跳转：if(rs1＜rs2) then PC+SEXT64 ({offse<<1})→PC |
| bgeu rs1,rs2,offset12 | offset[12|10:5] | rs2 | rs1 | 111 | offset[4:1|11] | 1100011 | 无符号比较大于等于时跳转：if(rs1≥rs2) then PC+SEXT64 ({offset<<1})→PC |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **J 型跳转和链接指令** | | | | |
| **位数**  **指令** | **20位** | **5位** | **7位** | **J 型格式的跳转和链接指令功能** |
| **offset** | **rd** | **opcode** |
| jal rd,offset20 | offset[20|10:1|11|19:12] | rd | 1101111 | 相对跳转并链接：PC+4→rd， PC+SEXT64(offset<<1)→PC |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| I 型跳转和链接指令 | | | | | | |
| 位数  指令 | **12位** | **5位** | **3位** | **5位** | **7位** | **I 型格式的跳转和链接指令功能** |
| **offset12** | **rs1** | **funct3** | **rd** | **opcode** |
| jalr rd,offset12(rs1) | offset[11:0] | rs1 | 000 | rd | 1100111 | 寄存器相对跳转并链接，12位偏移量经符号扩展后与rs1相加，将得到的地址最低位置0后置入PC：PC+4→rd,  (rs1+SEXT64(offset12))&~1→PC |

#### 控制状态类指令

RISC-V架构的控制状态类指令包括同步指令和环境调试/断点指令。

##### 同步指令

同步指令也称为栅栏指令或存储器屏障指令。RISC-V架构支持多核硬件多线程技术，一个CPU芯片中有多个RISC-V处理器核core，每个core中可设计多个硬件线程hart，每个hart有独立的寄存器组等现场上下文资源。一个处理器核core内的所有hart共享该core提供的运算资源，所有处理器核core共享主存储器。

对于共享存储器的硬件多线程处理器，存在与多核系统和多处理器系统相同的存储器一致性模型问题，因此需要有支持同步等机制的指令。fence指令用于约束数据访问指令的执行顺序。fence指令保证了在fence指令之前的指令发出的数据访问比fence指令之后所有指令的数据访问先完成。

在表1-10给出了基于I型指令格式的同步指令fence的格式。其中，pred字段由PI、PO、PR和PW这4个二进制位组成，pred字段给出了fence指令之前的处理器操作；succ字段由SI、SO、SR和SW这4个二进制位组成，succ字段给出了fence指令之后的处理器操作。

* + - * 1. 控制状态类指令

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **同步指令** | | | | | | | | | | | | | | |
| **位数**  **指令** | **31 28** | **27 24** | | | | **23 20** | | | | **19 15** | **14 12** | **11 7** | **6 0** | **I型格式同步指令功能** |
| **fm** | **pred** | | | | **succ** | | | | **rs1** | **funct3** | **rd** | **opcode** |
| fence | 0000 | PI | PO | PR | PW | SI | SO | SR | SW | 00000 | 000 | 00000 | 0001111 | 同步线程 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 环境调试/断点指令 | | | | | | |
| 位数  指令 | **12位** | **5位** | **3位** | **5位** | **7位** | **I 型格式的环境调试/断点指令功能** |
| **offset12** | **rs1** | **funct3** | **rd** | **opcode** |
| ecall | 0000 0000 0000 | 00000 | 000 | 00000 | 1110011 | 通过引发环境调用异常来请求执行系统调用 |
| ebreak | 0000 0000 0001 | 00000 | 000 | 00000 | 1110011 | 通过断点异常的方式来请求调试器 |

##### 环境调试/断点指令

环境调试/断点指令包含ecall和ebreak两条陷阱指令，用于从用户程序陷入操作系统内核或者调试器，它们会在执行环境中产生一个精确的请求陷阱。ecall指令用于应用程序向操作系统内核发出系统调用的服务请求；ebreak指令用于产生断点异常，从而将控制权交给调试环境。表1-10给出了I型指令格式的环境调试/断点指令ecall和ebreak的格式。

### RV64M指令集

如表1-11所示，RV64M指令集中包含了乘法运算指令、除法运算指令和除法求余运算指令。表1-11中，

1. s×s表示有符号数乘法，即补码乘法；
2. U×U表示无符号数乘法，即绝对值乘法；
3. s×u表示有符号数与无符号数乘法，即补码与绝对值相乘；
4. >>U表示无符号数右移，即逻辑右移；
5. ÷S表示有符号数除法，即补码除法；
6. ÷U表示无符号数除法，即绝对值除法；
7. %S表示有符号数除法求余数，即补码除法运算后取其余数；
8. %U表示无符号数除法求余数，即绝对值除法运算后取其余数。
   * + - 1. 乘除法和求余数运算指令

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **乘法运算指令** | | | | | | | |
| **位数**  **指令** | **7位** | **5位** | **5位** | **3位** | **5位** | **7位** | **R型格式乘法运算指令的功能** |
| **funct7** | **rs2** | **rs1** | **funct3** | **rd** | **opcode** |
| mul rd,rs1,rs2 | 0000001 | rs2 | rs1 | 000 | rd | 0110011 | 乘法。rs1和rs2视为补码，乘积低64位写入rd：rs1×rs2→rd |
| mulh rd,rs1,rs2 | 0000001 | rs2 | rs1 | 001 | rd | 0110011 | 高位乘,rs1和rs2都视为补码，乘积高位置入rd：(rs1 S×S rs2)>>S64→rd |
| mulhsu rd,rs1,rs2 | 0000001 | rs2 | rs1 | 010 | rd | 0110011 | 高位有符号-无符号乘，rs1视为补码，rs2视为无符号数，乘积高位置入rd：(rs1 S×U rs2)>>s64→rd |
| mulhu rd,rs1,rs2 | 0000001 | rs2 | rs1 | 011 | rd | 0110011 | 高位无符号乘，rs1和rs2都视为无符号数，乘积高位置入rd：(rs1 U×U rs2)>>U 64→rd |
| mulw rd,rs1,rs2 | 0000001 | rs2 | rs1 | 000 | rd | 0111011 | 乘字。乘积截取低32位，然后符号扩展为64位后写入rd：  SEXT64((rs1×rs2)[31:0])→rd |
| **除法运算指令** | | | | | | | |
| **位数**  **指令** | **7位** | **5位** | **5位** | **3位** | **5位** | **7位** | **R型格式除法运算指令的功能** |
| **funct7** | **rs2** | **rs1** | **funct3** | **rd** | **opcode** |
| div rd,rs1,rs2 | 0000001 | rs2 | rs1 | 100 | rd | 0110011 | 除法。rs1和rs2视为补码，向零舍入，商写入rd：rs1÷S rs2→rd |
| divu rd,rs1,rs2 | 0000001 | rs2 | rs1 | 101 | rd | 0110011 | 无符号除法。rs1和rs2视为无符号数，向零舍入，商写入rd：rs1÷U rs2→rd |
| divuw rd,rs1,rs2 | 0000001 | rs2 | rs1 | 101 | rd | 0111011 | 无符号字除法。rs1和rs2视为无符号数，低32位相除，向零舍入，符号扩展后写入rd：SEXT(rs1[31:0]÷U rs2[31:0])→rd |
| divw rd,rs1,rs2 | 0000001 | rs2 | rs1 | 100 | rd | 0111011 | 字除法。rs1和rs2视为补码，低32位相除，向零舍入，符号扩展后写入rd：  SEXT(rs1[31:0]÷S rs2[31:0])→rd |
| **除法求余运算指令** | | | | | | | |
| **位数**  **指令** | **7位** | **5位** | **5位** | **3位** | **5位** | **7位** | **R型格式除法求余运算指令的功能** |
| **funct7** | **rs2** | **rs1** | **funct3** | **rd** | **opcode** |
| rem rd,rs1,rs2 | 0000001 | rs2 | rs1 | 110 | rd | 0110011 | 求余数。rs1和rs2视为补码，向零舍入，余数写入rd：rs1 %S rs2→rd |
| remu rd,rs1,rs2 | 0000001 | rs2 | rs1 | 111 | rd | 0110011 | 求余数。rs1和rs2视为无符号数，向零舍入，余数写入rd：rs1%U rs2→rd |
| remuw rd,rs1,rs2 | 0000001 | rs2 | rs1 | 111 | rd | 0111011 | 求无符号数的余数字。rs1和rs2视为无符号数，低32位相除，向零舍入，余数符号扩展后写入rd：  SEXT64(rs1[31:0] %U rs2[31:0])→rd |
| remw rd,rs1,rs2 | 0000001 | rs2 | rs1 | 110 | rd | 0111011 | 求余数字。rs1和rs2视为补码，低32位相除，向零舍入，余数符号扩展后写入rd：  SEXT64(rs1[31:0]%S rs2[31:0])→rd |

### RV64A指令集

RV64A指令集中用于同步的原子操作有两种： ①原子内存操作，相关指令的助记符用amo开头； ②预订取数和条件存数，即lr指令和sc指令。AMO指令和lr/sc 指令都要求存储器地址对齐。

**“原子” 表示一系列操作连贯而不被打断。**例如，内存读和写操作之间不会因发生中断而被打断，也不会被其他处理器修改内存值。

AMO指令连续执行①访存读数②执行运算③将结果写入目的寄存器这3个步骤的操作。AMO指令在大型多处理器系统中具有较好的可扩展性，可高效实现归约操作。AMO 指令在与I/O 设备通信时可在单次原子总线事务中进行一次读操作和写操作，这种原子性可简化设备驱动并提升I/O 性能。

预订取数指令的功能是：从存储器读出一个字/双字后将其写入目的寄存器，并为该字/双字注册预订信息。

条件存数指令的功能是：以源寄存器rs1为存储器的起始地址，向该地址指向的4/8个字节写入一个字/双字，前提是该目标地址是被预订过的。若写入成功，则将0写入目的寄存器rd；否则将一个非0 的错误码写入rd。

RV64A指令集中的原子指令格式如表1-12所示。每条原子指令都包含一个获取位aq和一个释放位rl。RISC-V 架构处理器采用松耦合内存一致性模型，因此其他线程hart可能看到乱序的内存访问。若原子指令的aq =1则保证其他线程hart看到的原子操作和在其之后的访存操作顺序一致；若rl=1则保证其他线程hart看到的原子操作和在其之前的访存操作顺序一致。

* + - * 1. 原子指令

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **位数**  **指令** | **5位** | **1位** | **1位** | **5位** | **5位** | **3位** | **5位** | **7位** | **R型格式的原子指令的功能** |
| **func** | **aq** | **rl** | **rs2** | **rs1** | **funct3** | **rd** | **opcode** |
| lr.w rd,rs1 | 00010 | aq | rl | 00000 | rs1 | 010 | rd | 0101111 | 加载保留字。从内存地址rs1开始加载4个字节，符号扩展后写入rd，并对这个内存值注册保留：  LoadReserved32(M[rs1])→rd |
| lr.d rd,rs1 | 00010 | aq | rl | 00000 | rs1 | 011 | rd | 0101111 | 加载保留双字。从内存地址rs1开始加载8个字节写入rd，并对这个内存值注册保留：  LoadReserved64(M[rs1])→rd |
| sc.w rd,rs2,(rs1) | 00011 | aq | rl | rs2 | rs1 | 010 | rd | 0101111 | 条件存入字。如果内存地址rs1上存在加载保留，将rs2的低4个字节存入该地址，存入成功则将0写入rd，否则写入一个非0的错误码：  (SUCC(rs2[31:0]→M[rs1])?0:~0) →rd |
| sc.d rd,rs2,(rs1) | 00011 | aq | rl | rs2 | rs1 | 011 | rd | 0101111 | 条件存入双字。如果内存地址rs1上存在加载保留，将rs2存入该地址，存入成功则将0写入rd，否则写入一个非0的错误码：  (SUCC(rs2→M[rs1])?0:~0) →rd |
| amoswap.w rd,rs2,(rs1) | 00001 | aq | rl | rs2 | rs1 | 010 | rd | 0101111 | 原子字交换。把内存地址为rs1的双字存入rd后，内存双字换成rs2的值：  AMO32(M[rs1]→rd, M[rs1]↔rs2) |
| amoswap.d rd,rs2,(rs1) | 00001 | aq | rl | rs2 | rs1 | 011 | rd | 0101111 | 原子双字交换。把内存地址为rs1的双字存入rd后，内存双字换成rs2的值：  AMO64(M[rs1]→rd, M[rs1]↔rs2) |
| amoadd.w rd,rs2,(rs1) | 00000 | aq | rl | rs2 | rs1 | 010 | rd | 0101111 | 原子加字。按内存地址rs1取出的字与rs2相加后，符号扩展再写入rd：  AMO64(SEXT64(M[rs1][31:0]+rs2[31:0]))→rd |
| amoadd.d rd,rs2,(rs1) | 00000 | aq | rl | rs2 | rs1 | 011 | rd | 0101111 | 原子加双字。按内存地址rs1取出的双字与rs2相加后写入rd：  AMO64(M[rs1][63:0]+rs2)→rd |
| amoxor.w rd,rs2,(rs1) | 00100 | aq | rl | rs2 | rs1 | 010 | rd | 0101111 | 原子字异或。内存地址rs1取出的字与rs2按位异或后，符号扩展再写入rd：  AMO64(SEXT64(M[rs1][31:0]⊕rs2[31:0]))→rd |
| amoxor.d rd,rs2,(rs1) | 00100 | aq | rl | rs2 | rs1 | 011 | rd | 0101111 | 原子双字异或。按内存地址rs1取出的双字与rs2按位异或后写入rd：  AMO64(M[rs1][63:0]⊕rs2)→rd |
| amoand.w rd,rs2,(rs1) | 01100 | aq | rl | rs2 | rs1 | 010 | rd | 0101111 | 原子字与。按内存地址rs1取出的字与rs2按位与后，符号扩展再写入rd：  AMO64(SEXT64(M[rs1][31:0]&rs2[31:0]))→rd |
| amoand.d rd,rs2,(rs1) | 01100 | aq | rl | rs2 | rs1 | 011 | rd | 0101111 | 原子加双字。按内存地址rs1取出的双字与rs2按位与后写入rd：  AMO64(M[rs1][63:0]&rs2)→rd |
| amoor.w rd,rs2,(rs1) | 01000 | aq | rl | rs2 | rs1 | 010 | rd | 0101111 | 原子字或。按内存地址rs1取出的字与rs2按位或后，符号扩展再写入rd：  AMO64(SEXT64(M[rs1][31:0]|或rs2[31:0]))→rd |
| amoor.d rd,rs2,(rs1) | 01000 | aq | rl | rs2 | rs1 | 011 | rd | 0101111 | 原子双字或。按内存地址rs1取出的双字与rs2按位或后写入rd：  AMO64(M[rs1][63:0]|或rs2)→rd |
| amomin.w rd,rs2,(rs1) | 10000 | aq | rl | rs2 | rs1 | 010 | rd | 0101111 | 原子最小字。按内存地址rs1取出的字与rs2按补码比较，较小的数符号扩展后写入rd：  AMO64(SEXT64(min(M[rs1][31:0]，rs2)))→rd |
| amomin.d rd,rs2,(rs1) | 10000 | aq | rl | rs2 | rs1 | 011 | rd | 0101111 | 原子最小双字。按内存地址rs1取出的双字与rs2按补码比较，较小的写入rd：  AMO64(min(M[rs1][63:0]，rs2))→rd |
| amomax.w rd,rs2,(rs1) | 10100 | aq | rl | rs2 | rs1 | 010 | rd | 0101111 | 原子最大字。按内存地址rs1取出的字与rs2按补码比较，较大的数符号扩展后写入rd：  AMO64(SEXT64(max(M[rs1][31:0]，rs2)))→rd |
| amomax.d rd,rs2,(rs1) | 10100 | aq | rl | rs2 | rs1 | 011 | rd | 0101111 | 原子最大双字。按内存地址rs1取出的双字与rs2按补码比较，较大的写入rd：  AMO64(max(M[rs1][63:0]，rs2))→rd |
| amominu.w rd,rs2,(rs1) | 11000 | aq | rl | rs2 | rs1 | 010 | rd | 0101111 | 原子无符号最小字。按内存地址rs1取出的字与rs2按无符号数比较，较小的数符号扩展后写入rd：  AMO64(SEXT64(minu(M[rs1][31:0]，rs2)))→rd |
| amominu.d rd,rs2,(rs1) | 11000 | aq | rl | rs2 | rs1 | 011 | rd | 0101111 | 原子无符号最小双字。按内存地址rs1取出的双字与rs2按无符号数比较，较小的写入rd：  AMO64(minu(M[rs1][63:0]，rs2))→rd |
| amomaxu.w rd,rs2,(rs1) | 11100 | aq | rl | rs2 | rs1 | 010 | rd | 0101111 | 原子无符号最大字。按内存地址rs1取出的字与rs2按无符号数比较，较大的数符号扩展后写入rd：  AMO64(SEXT64(maxu(M[rs1][31:0]，rs2)))→rd |
| amomaxu.d rd,rs2,(rs1) | 11100 | aq | rl | rs2 | rs1 | 011 | rd | 0101111 | 原子最大双字。按内存地址rs1取出的双字与rs2按无符号数比较，较大的写入rd：  AMO64(maxu(M[rs1][63:0]，rs2))→rd |

### Zicsr指令集

如前所述，CSR寄存器是在处理器核内部用于配置或记录程序性能和状态信息的寄存器。CSR指令用于设置相应的控制信息或读取状态信息。RISC-V将所有CSR指令放入Zicsr指令集中作为拓展。表1-13给出了6条I型格式的CSR指令的格式和功能。

* + - * 1. CSR指令

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 位数  指令 | 12位 | 5位 | 3位 | 5位 | 7位 | I 型格式的csr指令功能 |
| **csr** | **rs1** | **funct3** | **rd** | **opcode** |
| csrrw rd,csr,rs1 | csr | rs1 | 001 | rd | 1110011 | 读后写csr: csr→rd, rs1→csr |
| csrrs rd,csr,rs1 | csr | rs1 | 010 | rd | 1110011 | 读后置位csr: csr→rd, rs1 | csr→csr |
| csrrc rd,csr,rs1 | csr | rs1 | 011 | rd | 1110011 | 读后清除csr: csr→rd, & csr→csr |
| csrrwi rd,csr,rs1 | csr | imm5 | 101 | rd | 1110011 | 立即数读后写csr，把5位的零扩展立即数zimm写入csr：  csr→rd, UEXT64 (imm5)→csr |
| csrrsi rd,csr,rs1 | csr | Imm5 | 110 | rd | 1110011 | 立即数读后写csr，把5位的零扩展立即数zimm和csr按位或后写入csr：  csr→rd, UEXT64(imm5) | csr→csr |
| csrrci rd,csr,rs1 | csr | imm5 | 111 | rd | 1110011 | 立即数读后清除csr，把5位的零扩展立即数zimm和csr按位与后写入csr：  csr→rd, & csr→csr |

### Zifencei指令集

为解决存储器一致性模型问题，RISC-V在Zifencei指令集中定义了fence.i指令用于同步指令流和数据流。

表1-14给出了基于I型指令流屏障指令fence.i的格式和功能。fence.i指令保证了“在fence.i指令之前所有指令的数据访问结果”一定能够被“在fence.i指令之后所有指令的取指令操作”访问到。

* + - * 1. 指令流屏障指令

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 指令流屏障指令 | | | | | | |
| 位数  指令 | **12位** | **5位** | **3位** | **5位** | **7位** | **I型格式指令流屏障指令功能** |
| **offset12** | **rs1** | **funct3** | **rd** | **opcode** |
| fence.i | 0000 0000 0000 | 00000 | 001 | 00000 | 0001111 | 同步指令流和数据流 |

### 特权指令

本书模型机实现4条RISC-V的特权指令，如表1-13所示。

* + - * 1. 特权指令

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **位数**  **指令** | **7位** | **5位** | **5位** | **3位** | **5位** | **7位** | **R型格式特权指令的功能** |
| **funct7** | **rs2** | **rs1** | **funct3** | **rd** | **opcode** |
| mret | 0011000 | 00010 | 00000 | 000 | 00000 | 1110011 | 机器模式异常返回 |
| sret | 0001000 | 00010 | 00000 | 000 | 00000 | 1110011 | 监管模式异常返回 |
| wfi | 0001000 | 00101 | 00000 | 000 | 00000 | 1110011 | 等待中断特权指令 |
| sfence.vma rs1,rs2 | 0001001 | rs2 | rs1 | 000 | 00000 | 1110011 | 虚拟内存屏障特权指令 |

## RISC-V的寻址方式

RISC-V架构常用的寻址方式有4种：

### 立即寻址

RISC-V的立即寻址方式支持两种获得立即数的方式：

#### 立即数由二进制指令码直接给出

例如，表1-6中I型格式的指令addi rd,rs1,imm12，其立即数长度为12位，从指令码中获得立即数后按符号扩展到64位，与寄存器rs1的值相加后，结果存入目的寄存器rd。图1-1展示了addi指令中直接引用指令码中给出的立即数的立即寻址方式

#### 由指令码的某些字段拼接而成，拼接方式因指令格式不同而不同

①S型指令的立即数（偏移量）拼接方式为：offset12={I[31:25]|I[11:7]}

②B型指令的立即数（偏移量）拼接方式为：offset12={I31|I7|I[30:25]|I[11:8]}

③U型指令的立即数拼接方式为：imm20={I[31:12]}

④J型指令的立即数（偏移量）拼接方式为：offset20=={I31|I[19:12] |I20|I[30:21]}

在64位系统中，无论哪种方式获得的立即数如果不足64位，那么需要扩展成64位立即数后再参加运算。

①I型指令的立即数拓展方式为：imm64= SEXT64（imm12）

②S型指令的立即数（偏移量）拓展方式为：offset64= SEXT64（offset12）

③B型指令的立即数（偏移量）拓展方式为：offset64= SEXT64（{offset12<<1}）

④U型指令的立即数拓展方式为：imm64= SEXT64（{imm20|12{0}}）

⑤J型指令的立即数（偏移量）拓展方式为：offset64== SEXT64（{offset20<<1}）

譬如，S型格式的sh rs2,offset12(rs1)指令，其偏移量offset12采用拼接的立即数寻址方式，即从二进制指令码中获得字段I[31:25]和I[11:7]后经过拼接构成偏移量offset12，再进行符号拓展得到最终的64位偏移量offset64。图1-2～图1-5分别展示了S、B、U、J型格式的立即数拼接方式。



* + - 1. 直接引用立即数



* + - 1. S型格式下拼接立即数的方式



* + - 1. B型格式下拼接立即数的方式



* + - 1. U型格式下拼接立即数的方式



* + - 1. J型格式下拼接立即数的方式

### 寄存器寻址

图1-1～图1-5也展示了寄存器寻址方式。寄存器寻址方式下，操作数位于通用寄存器堆的某个寄存器里，指令码的rs1、rs2、rd字段分别给出源寄存器和目的寄存器的地址（即寄存器号），RISC-V的所有指令格式都用到了寄存器寻址方式。

### 基址寻址

在图1-2中，offset12(rs1)表示目的操作数采用了基址寻址方式，其起始字节的访存地址EA由基址rs1与符号扩展的偏移量offset12相加得到，即EA = rs1+ SEXT64（offset12）。 在图1-2的举例中，sh rs2,offset12(rs1)指令要求将rs2寄存器的低16位写入存储器。将rs2的低16位中的低8位和高8位分别存入相邻的两个存储器单元，单元地址分别是EA和EA+1。

基址寻址在I型和S型格式的访存指令中有较多使用。

注：RISC-V的基址寻址方式实际上是变址寻址方式，为了和RISC-V资料上的称谓统一，称之为基址寻址方式。

### 相对寻址

图1-3和图1-5分别展示了相对寻址方式。两者的共同点是，偏移量与程序计数器PC的值相加后形成目标转移地址。不同之处主要在于图1-5中的是无条件跳转，即必须跳转；而图1-3中是条件跳转，由比较器产生的比较结果来决定是否跳转。为了避免链接时重新计算转移地址所带来的繁复，**在计算机系统中，所有跳转指令都采用相对寻址方式来生成转移目标地址。**

需要注意的是，为了保证转移目标地址能够按照半字（2字节）对齐，对于所有B型格式的分支跳转指令和J型格式的jal指令，它们的偏移量必须保证末位是0，因此offset先左移1位补0后再与PC相加得到转移目标地址。因为PC的末位始终是0，所以能够保证计算得到的转移目标地址末位为0。而对于I型格式的jalr指令, 它的偏移量offset直接与PC相加后，再通过&～1操作来使得转移目标地址的末位为0。