Risc-v 开发

Risc-v 基础知识

北京飞利信科技股份有限公司

2018 年 5 月

RISC-V 相关技术文档

1 RV32I 基本整数指令集

1.1 基本整数子集的程序员模型

有 31 个通用寄存器 x1~x31，它们保存了整数数值。寄存器 x0 是硬件连线的常数 0。没有硬件连线的子程序返回地址连接寄存器，但是在一个过程调用中，标准软件调用约定使用寄存器 x1 来保存返回地址。对于 RV32，其 x 寄存器是 32 位宽度的，对于 RV64，它们是 64 位宽度的。XLEN 指明当前 x 寄存器的宽度（不是 32 就是 64）。

还有一个额外的用户可见寄存器：程序计数器 pc 保存了当前指令的地址。

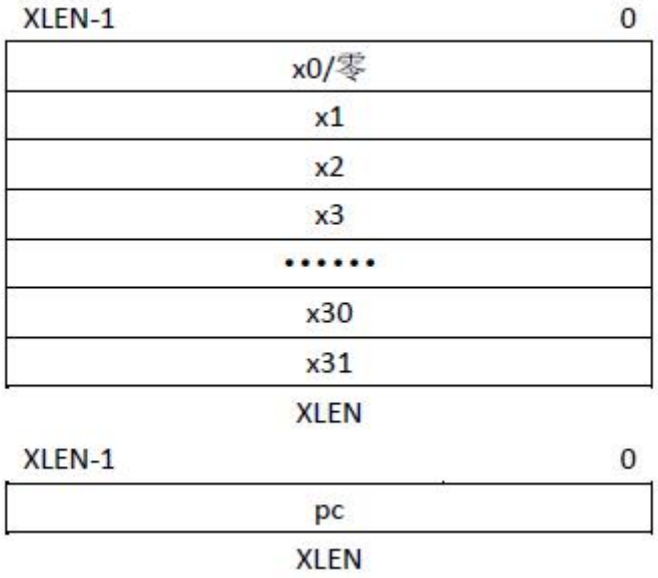


图 1.3 RISC-V 用户级基本整数寄存器状态

1.2 基本指令格式

在基本 ISA 中，有四种核心指令格式（R/I/S/U），如下图所示。所有的指令都是固定 32 位长度的，并且在存储器中必须在 4 字节边界对齐。当发生一个条件分支或者无条件转移而且目标地址不是对齐到 4 字节时，将会产生一个指令地址不对齐的异常。如果条件分支没有发生（not taken），那么将不会产生一个指令不对齐的异常。

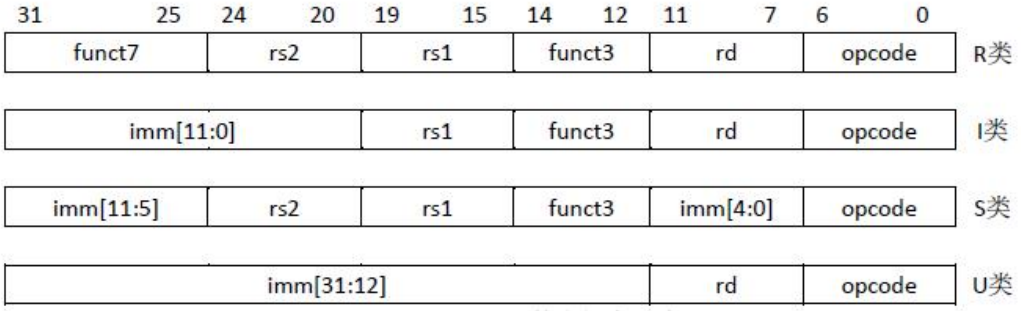


图 1.4 RISC-V 基本指令格式

在所有格式中，RISC-V ISA 将源寄存器（rs1 和 rs2）和目标寄存器（rd）固定在同样的位置，以简化指令译码。在指令中，立即数被打包，朝着最左边可用位的方向，并且是分配好的，以减少硬件复杂度。特别地，所有立即数的符号位总是在指令的第 31 位，以加速符号扩展电路。

1.3 立即数编码变种

基于立即数处理，还有额外两种指令格式变种（SB/UJ），如下图所示。

在图中，每个立即数字段被所生成的立即数值中的位置（imm[x]）标签，而不是在指令的立即数字段中的通常位的位置。每一种基本指令格式生成的立即数，并被标签，以显示哪个指令位（inst[y]）生成了立即数值中的哪个位。

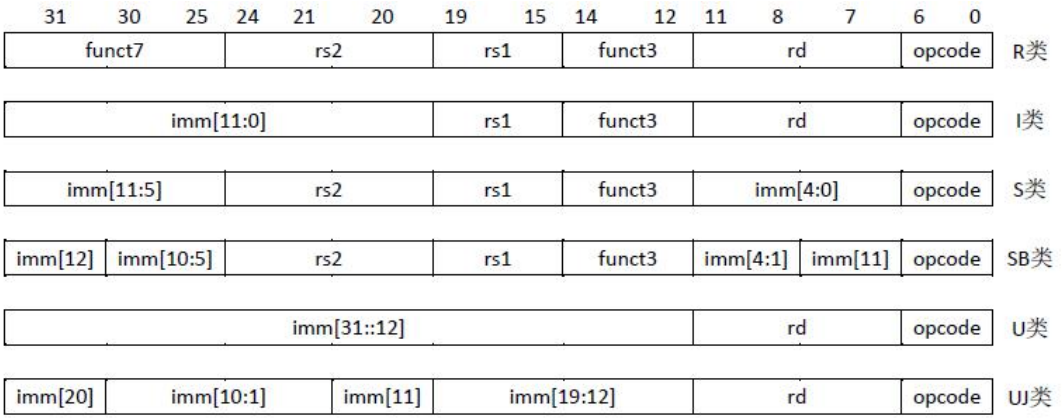


图 1.5 RISC-V 显示了立即数的基本指令格式

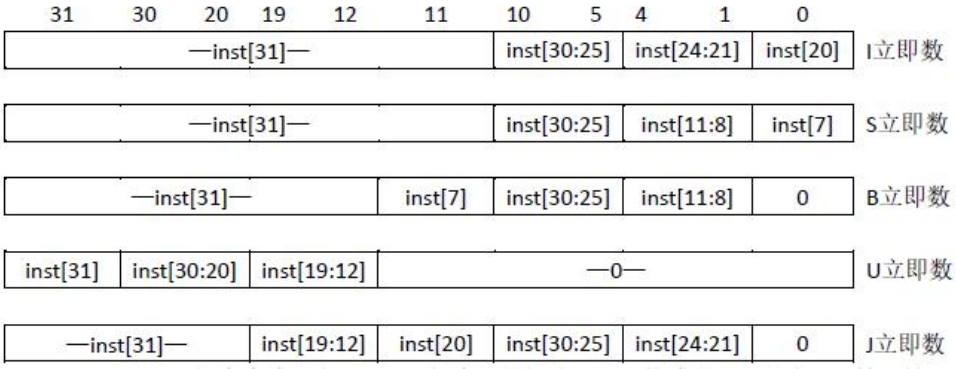


图 1.6 RISC-V 指令生成的立即数。用指令的位标注了用于构成立即数的字段。符号扩展总是使用 inst[31]。

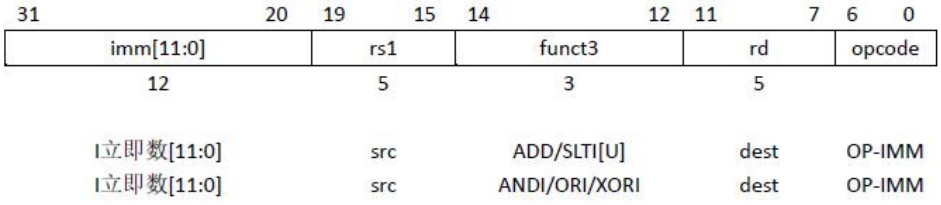
S 和 SB 格式唯一的区别在于，在 SB 格式中，12 位立即数字段用于编码 2 的倍数的分支偏移量。与通常在硬件中将编码在指令中的立即数所有位向左移动 1 位不同，此处中间位（imm[10:1]）和符号位保持在固定的位置，而 S 格式中的最低位（inst[7]）编码为 SB 格式中的高位（imm[11]）。

类似的，U 和 UJ 格式唯一的区别在于，20 位立即数被左移 12 位以生成 U 立即数，而被左移 1 位以生成 J 立即数。在 U 和 UJ 格式立即数，其在指令中的位置的选择，以最大化与其它指令的相互覆盖，以及最大化 U 和 UJ 格式立即数的相互覆盖。

1.4 整数计算指令

绝大多数整数计算指令对保存在整数寄存器中的 XLEN 位值进行操作。整数计算指令要么使用 I 类格式编码为寄存器-立即数操作，要么使用 R 类格式编码为寄存器-寄存器操作。对于寄存器-立即数指令和寄存器-寄存器指令，其目标都是寄存器 rd。没有整数计算指令产生算术异常。

整数寄存器-立即数指令

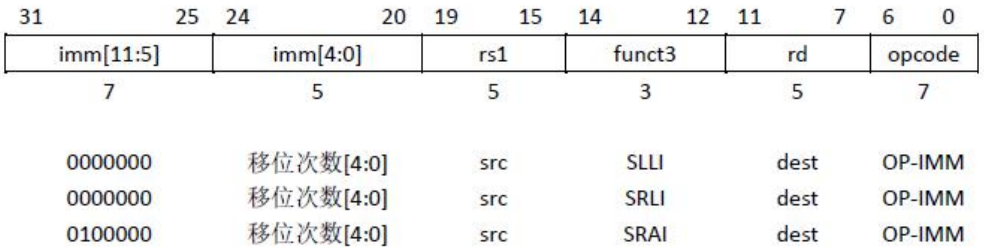


ADDI 将符号扩展的 12 位立即数加到寄存器 rs1 上。算术溢出被忽略，而结

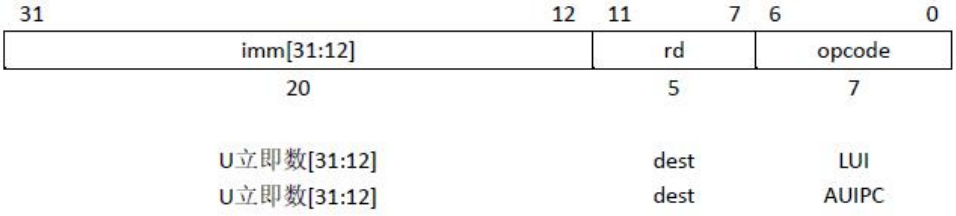
果就是运算结果的低 XLEN 位。ADDI rd,rs1,0 用于实现 MV rd,rs1 汇编语言伪指令。

SLTI（set less than immediate）将数值 1 放到寄存器 rd 中，如果寄存器 rs1 小于符号扩展的立即数(比较时,两者都作为有符号数),否则将 0 写入 rd。SLTIU 与之相似，但是将两者作为无符号数进行比较（也就是说，立即数被首先符号扩展为 XLEN 位，然后被作为一个无符号数)。注意，SLTIU rd,rs1,1 将设置 rd 为 1，如果 rs1 等于 0，否则将 rd 设置为 0（汇编语言伪指令 SEQZ rd,rs)。

ANDI、ORI、XORI 是逻辑操作，在寄存器 rs1 和符号扩展的 12 位立即数上执行按位 AND、OR、XOR 操作，并把结果写入 rd。注意，XORI rd,rs1,-1 在 rs1 上执行一个按位取反操作（汇编语言伪指令 NOT rd,rs)。



被移位常数次，被编码为 I 类格式的特例。被移位的操作数放在 rs1 中，移位的次数被编码到 I 立即数字段的低 5 位。右移类型被编码到 I 立即数的一位高位。SLLI 是逻辑左移（0 被移入低位)；SRLI 是逻辑右移（0 被移入高位)；SRAI 是算术右移（原来的符号位被复制到空出的高位中)。



LUI（load upper immediate）用于构建 32 位常数，并使用 U 类格式。LUI 将 U 立即数放到目标寄存器 rd 的高 20 位，将 rd 的低 12 位填 0。

AUIPC（add upper immediate to pc）用于构建 pc 相对地址，并使用 U 类格式。AUIPC 从 20 位 U 立即数构建一个 32 位偏移量，将其低 12 位填 0，然后将

这个偏移量加到 pc 上，最后将结果写入寄存器 rd。

整数寄存器-寄存器操作

RV32I 定义了几种算术 R 类操作。所有操作都是读取 rs1 和 rs2 寄存器作为源操作数，并把结果写入到寄存器 rd 中。funct7 和 funct3 字段选择了操作的类型。

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

ADD 和 SUB 分别执行加法和减法。溢出被忽略，并且结果的低 XLEN 位被写入目标寄存器 rd。SLT 和 SLTU 分别执行符号数和无符号数的比较，如果 rs1<rs2，则将 1 写入 rd，否则写入 0。注意，SLTU rd,x0,rs2，如果 rs2 不等于 0（译者注：在 RISC-V 中，x0 寄存器永远是 0），则把 1 写入 rd，否则将 0 写入 rd（汇编语言伪指令 SNEZrd,rs）。AND、OR、XOR 执行按位逻辑操作。

SLL、SRL、SRA 分别执行逻辑左移、逻辑右移、算术右移，被移位的操作数是寄存器 rs1，移位次数是寄存器 rs2 的低 5 位。

NOP 指令

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	
0	0	ADDI	0	OP-IMM

NOP 指令并不改变任何用户可见的状态，除了使得 pc 向前推进。NOP 被编码为 ADDI x0,x0,0。

1.5 控制转移指令

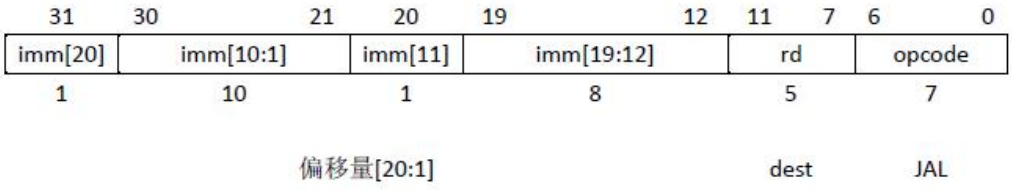
RV32I 提供了两类控制转移指令：无条件跳转和条件分支。RV32I 中的控制

转移指令，并没有体系结构可见的分支延迟槽。

无条件跳转

跳转并连接（JAL）指令使用了 UJ 类格式，此处 J 立即数编码了一个 2 的倍数的有符号偏移量。这个偏移量被符号扩展，加到 pc 上，形成跳转目标地址，跳转范围因此达到±1MB。JAL 将跳转指令后面指令的地址（pc+4）保存到寄存器 rd 中。标准软件调用约定使用 x1 来作为返回地址寄存器。

普通的无条件跳转指令（汇编语言伪指令 J）被编码为 rd=x0 的 JAL 指令。



间接跳转指令 JALR（jump and link register）使用 I 类编码。通过将 12 位有符号 I 类立即数加上 rs1，然后将结果的最低位设置为 0，作为目标地址。跳转指令后面指令的地址（pc+4）保存到寄存器 rd 中。如果不需要结果，则可以把 x0 作为目标寄存器。

JAL 指令和 JALR 指令会产生一个非对齐指令取指异常，如果目标地址没有对齐到 4 字节边界。

条件分支

所有分支指令使用 SB 类指令格式。12 位 B 立即数编码了以 2 字节倍数的有符号偏移量，并被加到当前 pc 上，生成目标地址。条件分支范围是±4KB。

31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
偏移量[12,10:5]		src2	src1	BEQ/BNE	偏移量[11,4:1]		BRANCH
偏移量[12,10:5]		src2	src1	BLT[U]	偏移量[11,4:1]		BRANCH
偏移量[12,10:5]		src2	src1	BGE[U]	偏移量[11,4:1]		BRANCH

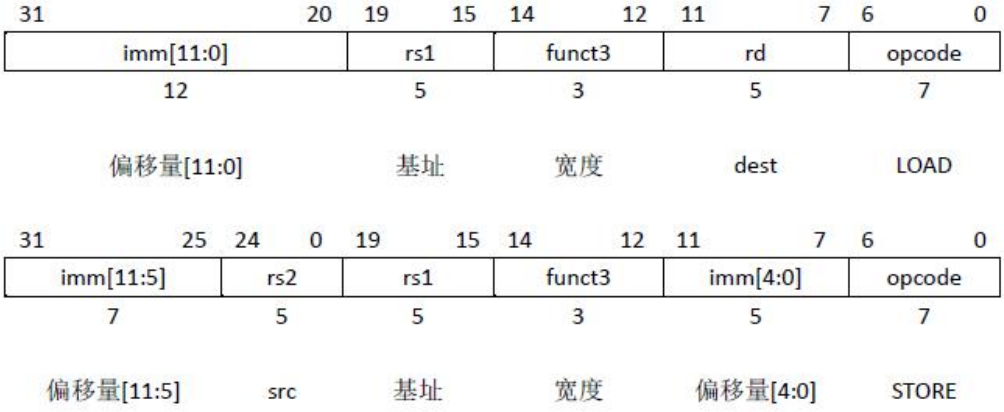
分支指令比较两个寄存器。BEQ 和 BNE 将跳转，如果 rs1 和 rs2 相等或者不相等。BLT 和 BLTU 将跳转，如果 rs1 小于 rs2，分别使用有符号数和无符号数进行比较。BGE 和 BGEU 将跳转，如果 rs1 大于等于 rs2，分别使用有符号数和无符号数进行比较。注意，BGT、BGTU、BLE 和 BLEU 可以通过将 BLT、BLTU、BGE、BGEU 的操作数对调来实现。

软件应当优化，使得顺序代码路径是最常见执行路径，而频率较少的跳转执行代码则放到直线路径之外。软件同时也应当假设向回（向后）跳转总是被预测跳转的，而向前（向下）跳转总是被预测不跳转的，至少第一次碰到分支指令的时候，是这样的。动态分支预测器将很快学会任何可以预测的分支行为。

与其它某些体系结构不同，无条件跳转应当总是使用 RISC-V 的跳转（rd=x0 的 JAL）指令，而不是一条条件永远为真的条件分支指令。RISC-V 跳转总是 pc 相对寻址的，并且比分支指令支持大得多的偏移量范围，而且还不会对条件分支预测表造成压力。（现代处理器都有条件分支预测器，对每一条碰到的条件分支指令，都会记录其结果，以便后面对其进行预测）。

1.6 Load 和 store 指令

RV32I 是一个 load-store 体系结构，也就是说，只有 load 和 store 指令可以访问存储器，而算术指令只在 CPU 寄存器上进行操作运算。RV32I 提供了一个 32 位用户地址空间，它是字节寻址并且是小端的。执行环境将定义这个地址空间的哪些部分是可以合法访问的（涉及到存储保护等）。



Load 和 store 指令在寄存器和存储器之间传输数值。Load 指令编码为 I 类格式，而 store 指令编码为 S 类格式。有效字节地址是通过将寄存器 rs1 与符号扩展的 12 位偏移量相加而获得的。Load 指令将存储器中的一个值复制到寄存器 rd 中。Store 指令将寄存器 rs2 中的值复制到存储器中。

LW 指令将一个 32 位数值从存储器复制到 rd 中。LH 指令从存储器中读取一个 16 位数值，然后将其进行符号扩展到 32 位，再保存到 rd 中。LHU 指令存储器中读取一个 16 位数值，然后将其进行零扩展到 32 位，再保存到 rd 中。对于 8 位数值，LB 和 LBU 指令的定义与前面类似。SW、SH、SB 指令分别将从 rs2 低位开始的 32 位、16 位、8 位数值保存到存储器中。

为了获得最高的性能，所有 load 和 store 指令的有效地址，应该与该指令对应的数据类型相对齐（也就是说，32 位访问应该在 4 字节边界对齐，16 位访问应该在 2 字节边界对齐）。基本 ISA 支持非对齐的访问，但是根据实现的不同，这可能会运行得非常慢。更进一步的，对齐的 load 和 store 访问执行时，可以确保是原子性的，而非对齐的 load 和 store 可能不能原子性的完成，因此需要额外的同步来确保原子性（具体实现非对齐访问时，可能一次访问会被分解为两次存储器访问，这就不是不可分割的原子性操作，有潜在的危险）。

1.7 存储器模型

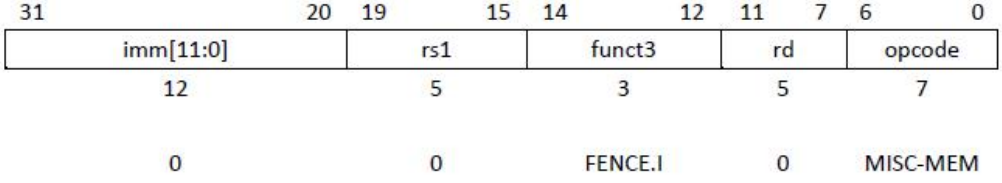
基本 RISC-V ISA 在一个单一的用户地址空间内支持多个同时线程的执行。每个 RISC-V 线程拥有它自己的寄存器和程序计数器，并执行一段不相关的顺序指令流。执行环境将定义 RISC-V 线程是如何创建和管理的。RISC-V 线程可以通过调用执行环境或者直接通过共享存储器系统来在相互之间进行通信和同步，执行环境将在规范的另外文档中描述。RISC-V 线程也可以与 I/O 设备交互，并可通过对

指派给 I/O 的地址空间部分进行 load 和 store，间接地在 I/O 设备间通信。

在基本 RISC-V ISA 中，每个 RISC-V 线程看到它自己的存储器操作，如同它们就是按照程序中的顺序执行一样。RISC-V 在线程间有一个放松的存储器模型（relaxed memory model），在不同的 RISC-V 线程之间的存储器操作，需要一条明确的 FENCE 指令来确保任何特定地顺序。

FENCE 指令用于顺序化其他 RISC-V 线程、外部设备或者协处理器看到的设备 I/O 和存储器访问。任何设备输入（I）、设备输出（O）、存储器读（R）、存储器写（W）的组合，相对于其他一样的组合，可能需要按序的。通俗的说，在所有前续集合（***predecessor set***）执行到 FENCE 指令前的任何操作之前，处在 FENCE 指令后的后续集合（***successor set***）中的任何操作，都不能被任何其他 RISC-V 线程或者外部设备看到（**FENCE** 就像一个栅栏，**FENCE** 之前所有的存储器操作、**I/O** 操作必须完成后，在 **FENCE** 之后的指令才能看到结果）。执行环境将定义什么 I/O 操作是可能的，特别地，哪些 load 或者 store 指令被处理并且作为设备输入或设备输出操作顺序化，而不是被作为存储器读和写来处理。例如，内存映射 I/O（memory-mapped I/O）设备通常被非缓存（uncached）的 load 和 store 指令来访问，并使用（FENCE 指令中的）I 和 O 位，而不是 R 和 W 位。

FENCE 指令中未使用的字段 ***imm[11:8]***、***rs1*** 和 ***rd*** 被保留给未来扩展中更细粒度的栅栏。为了保持前向兼容性，基本实现应当忽略这些字段，而标准软件应当对这些字段写 0。



FENCE.I 指令用于同步指令和数据流。RISC-V 并不能确保在同一个 RISC-V 线

程中，取指看得到前面对指令存储器的 store，直到执行一条 FENCE.I 指令。一条 FENCE.I 指令只是保证在一个 RISC-V 线程中，该指令之后的取指操作，可以看得到这条指令之前的任何数据 store。在多处理器系统中，FENCE.I 指令并不能确保其他 RISC-V 线程的取指看得到本地线程的 store。为了使得一条对指令存储器的 store 对所有 RISC-V 线程可见，写数据的线程必须在要求所有远程 RISC-V 线程执行 FENCE.I 指令之前，执行一条数据 FENCE 指令。

FENCE.I 指令中未使用的字段 imm[11:0]、rs1 和 rd 被保留给未来扩展中更细粒度的栅栏。为了保持前向兼容性，基本实现应当忽略这些字段，而标准软件应当对这些字段写 0。

1.8 控制和状态寄存器指令

系统指令用于访问那些可能需要特权访问的系统功能，以 I 类指令格式编码。这可以分为两类：一类是原子性读-修改-写控制和状态寄存器（CSR）的指令，另一类是其他特权指令。

CSR 指令

CSR 指令，虽然在用户级基本 ISA 中，只能访问少数几个只读计数器。

31 20	19 15	14 12	11 7	6 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
source/dest	source	CSRRW	dest	SYSTEM
source/dest	source	CSRRS	dest	SYSTEM
source/dest	source	CSRRC	dest	SYSTEM
source/dest	zimm[4:0]	CSRRWI	dest	SYSTEM
source/dest	zimm[4:0]	CSRRSI	dest	SYSTEM
source/dest	zimm[4:0]	CSRRCI	dest	SYSTEM

CSRRW（Atomic Read/Write CSR）指令原子性的交换 CSR 和整数寄存器中的值。CSRRW 指令读取在 CSR 中的旧值，将其零扩展到 XLEN 位，然后写入整数寄存器 rd 中。rs1 寄存器中的值将被写入 CSR 中。如果 rd＝x0，那么这条指令将不会读该 CSR，且不会导致任何因为 CSR 读而出现的副作用。

CSRRS（Atomic Read and Set Bitsin CSR）指令读取 CSR 的值，将其零扩展到 XLEN 位，然后写入整数寄存器 rd 中。整数寄存器 rs1 中的初始值被当做按位掩

码指明了哪些 CSR 中的位被置为 1。rs1 中的任何为 1 的位，将导致 CSR 中对应位被置为 1，如果 CSR 中该位是可以写的话。CSR 中的其他位不受影响（虽然当 CSR 被写入时可能有些副作用）。

CSRRC（Atomic Read and Clear Bitsin CSR）指令读取 CSR 的值，将其零扩展到 XLEN 位，然后写入整数寄存器 rd 中。整数寄存器 rs1 中的初始值被当做按位掩码指明了哪些 CSR 中的位被置为 0。rs1 中的任何为 1 的位，将导致 CSR 中对应位被置为 0，如果 CSR 中该位是可以写的话。CSR 中的其他位不受影响。

对于 CSRRS 指令和 CSRRC 指令,如果 rs1＝x0,那么指令将根本不会去写 CSR,因此应该不会产生任何由于写 CSR 产生的副作用(某些特殊 CSR 检测是否有人尝试写入，一旦有写入，则执行某些动作。这和写入什么值没什么关系），例如试图访问一个只读 CSR 时产生一个非法指令异常。注意如果 rs1 寄存器包含的值是 0，而不是 rs1＝x0，那么将会把一个不修改的值写回 CSR。

CSRRWI 指令、CSRRSI 指令、CSRRCI 指令分别于 CSRRW 指令、CSRRS 指令、CSRRC 指令相似，除了它们是使用一个处于 rs1 字段的、零扩展到 XLEN 位的 5 位立即数(zimm[4:0])而不是使用 rs1 整数寄存器的值。对于 CSRRSI 指令和 CSRRCI 指令，如果 zimm[4:0]字段是零，那么这些指令将不会写 CSR，因此应该不会产生任何由于写 CSR 产生的副作用。对于 CSRRWI 指令，如果 rd=x0，则这条指令将不会读 CSR，且不会导致任何因为 CSR 读而出现的副作用。

某些 CSR，例如指令退休计数器，instret，可能由于指令的执行副作用而被修改。在这种情形下，如果一条 CSR 访问指令读取了一个 CSR，它读取到的是该指令执行之前的值。如果一条 CSR 指令写了一个 CSR，这个写的值更新是在该指令执行完之后才发生的。特别地，一条指令写入 instret 一个值，将是该指令后面一条指令读取到的值（也就是说，由于第一条指令退休导致的 instret 的增长，是在写入新值之前发生的）。

用于读取 CSR 的汇编语言伪指令 CSRR rd, csr 被编码为 CSRRS rd, csr, x0。用于写 CSR 的汇编语言伪指令 CSRW csr, rs1 被编码为 CSRRW x0, csr, rs1，而伪指令 CSRWI csr, zimm 被编码为 CSRRWI x0, csr, zimm。

还有汇编语言伪指令被定义在不需要 CSR 旧值时,用来设置和清除 CSR 中的位：CSRS/CSRC csr, rs1；CSRSI/CSRCI csr, zimm。

定时器和计数器

31 20	19 15	14 12	11 7	6 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
RDCYCLE[H]	0	CSRRS	0	SYSTEM
RDTIME[H]	0	CSRRS	0	SYSTEM
RDINSTRET[H]	0	CSRRS	0	SYSTEM

RV32I 提供了多个用户级只读的 64 位计数器,它们被映射到一个 12 位的 CSR 地址空间中，它们可以使用 CSRRS 指令以 32 位片段的形式进行访问。

RDCYCLE 伪指令读取 cycle CSR 的低 XLEN 位，这个计数值是从硬件线程从过去的任意时刻开始执行以来的时钟周期计数值。RDCYCLEH 指令是一条 RV32I 仅有的指令，它读取同样的计数值的 63-32 位。底层的 64 位计数器在实际使用中应当永远不会溢出。这个周期计数器推进的速率，与实现和操作系统有关。执行环境应当提供一种手段来判定当前的速率（每秒钟多少个时钟周期），周期计数器就是按这个时钟周期速率递增的。

RDTIME 伪指令读取 time CSR 的低 XLEN 位，这个计数值是从过去的任意时刻开始以来的墙钟实时时间计数值。RDTIMEH 指令是一条 RV32I 仅有的指令，它读取同样的实时时钟计数值的 63-32 位。底层的 64 位计数器在实际使用中应当永远不会溢出。执行环境应当提供一种手段来判定实时时钟的周期（每 tick 多少秒），这个周期应当是一个常量。在一个单用户应用程序中，所有硬件线程的实时时钟必须是同步的，而且误差不能超过实时时钟的一个 tick。环境应当提供一种手段来判定时钟的精度。

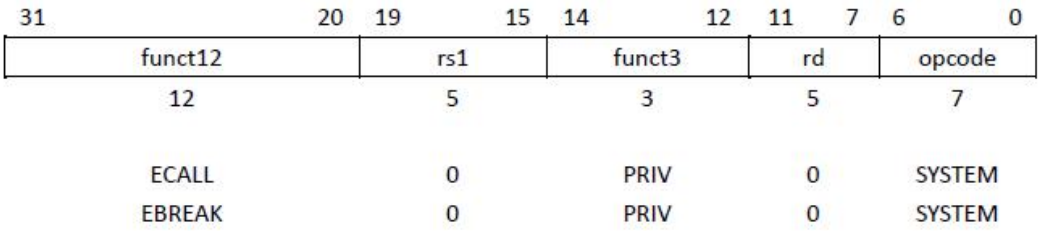
RDINSTRET 伪指令读取 instret CSR 的低 XLEN 位，这个计数值是从硬件线程从过去的任意时刻开始执行以来的本硬件线程退休（retire）指令的计数值。RDINSTRETH 指令是一条 RV32I 仅有的指令，它读取同样的指令计数值的 63-32 位。底层的 64 位计数器在实际使用中应当永远不会溢出。

下面的代码序列可以将一个有效的 64 位周期计数器值写入到 x3:x2 中，即使这个计数器在读取它的高低两部分之间产生溢出。

```
again:
        rdcycleh    x3
        rdcycle     x2
        rdcycleh    x4
        bne         x3, x4, again
```

图 1.7 在 RV32 中读取 64 位周期计数器的示例代码

环境调用和断点



31 20	19 15	14 12	11 7	6 0
funct12	rs1	funct3	rd	opcode
12	5	3	5	7
ECALL	0	PRIV	0	SYSTEM
EBREAK	0	PRIV	0	SYSTEM

ECALL 指令用于向支持的运行环境发出一个请求，这个运行环境通常是一个操作系统。系统的 ABI 将定义环境请求的参数是如何传递的，但通常这些参数应当是保存在整数寄存器中确定的位置。

EBREAK 指令被调试器所使用，用来将控制权传送回给调试环境。

原文详见 **https://riscv.org/specifications/** Chapter 2

参考文献： riscv-spec-v2.2

2 RV32E、RV64I、RV128I 基本整数指令集

由于飞利信参照开源指令集 Rv32IMC 进行 MCU 芯片研发，其他指令集技术文档不作具体翻译，如有需要，详见 **https://riscv.org/specifications/** Chapter 3-5。

参考文献： riscv-spec-v2.2