# C910矢量指令集总结

一 玄铁C910矢量指令集介绍

1.1 矢量寄存器

C910中有32个矢量寄存器，它们在Machine/Supervisor/User模式下都能和通用寄存器，浮点寄存器进行数据交换，且矢量寄存器拥有足够大的位宽--128位。

矢量寄存器使用矢量整型标量传送指令VMV.V.X，矢量整型提取元素指令VEXT.X.V 和通用寄存器传递数据，使用矢量浮点标量传送指令VFMV.V.F，矢量首元素浮点传送指令VFMV.F.S 和浮点寄存器传递数据。

1.2 RVV矢量指令集

本节介绍了矢量指令集，玄铁C910指令集手册中描述了全部363条矢量指令的具体定义和编码格式，详细见手册中附录A-7 V指令术语。玄铁C910中矢量指令的操作码遵循RISCV的spec-v-Opcode Inst(6-0)=0b0000111矢量指令集按功能可以分为矢量整型指令，矢量浮点指令和矢量加载存储指令，表1介绍了矢量指令的具体分类，执行延时和类型个数。

表1：C910的矢量（RVV）指令列表

|  |  |  |
| --- | --- | --- |
| 矢量类型 | 执行延时 | 个数 |
| 矢量控制指令 | 阻塞执行 | 2 |
| 矢量 MISC 指令 | 3 | 27 |
|  | 拆分执行 3 + 1 | 3 |
| 矢量缩减指令 | 4 | 7 |
|  | 3 | 3 |
| 矢量乘法乘累加指令 | sew=8 或 16 时：3 | 2 |
|  | sew>16 时：4 | 6 |
|  |  | 12 |
|  | 拆分执行 sew=8 或 16 时： 3+1 | 3 |
|  | 拆分执行 sew>16 时：4+1 | 17 |
| 矢量移位指令 | 3 | 18 |
|  | 3+1 | 9 |
| 矢量整型加减法指令 | 3 | 14 |
|  | 拆分执行 3+1 | 19 |
| 矢量整型比较指令 | 拆分执行：3+1 | 14 |
|  | 3 | 6 |
| 矢量整型最大值/最小值指令 | 3 | 4 |
|  | 拆分执行：3+1 | 4 |
| 矢量整型除法/取余指令 | VV:6~23 VX:7~24 | 4 |
|  | 6~23 + 1 | 4 |
| 矢量元素排序操作指令 | 2 | 1 |
|  | 3 | 3 |
|  | 拆分执行 3+1 | 8 |
| 矢量整型定点加减法指令 | 拆分执行：3+1 | 7 |
|  | 3 | 5 |
| 浮点运算指令 |  |  |
| 矢量浮点比较指令 |  |  |
| 浮点传送指令 |  |  |
| 浮点数据类型转换指令 |  |  |
| 浮点缩减指令 |  |  |
| 浮点分类指令 |  |  |
| 矢量加载存储指令 |  |  |
| 矢量原子指令 |  |  |
|  |  |  |

术语解释; Standard element width (SEW) setting 标准元素宽度

1.3 RISCV Vector指令和SIMD对比

SIMD是单指令多数据（Single Instruction Multiple Data），它可以对64位寄存器的数据拆分，然后执行并行计算。操作码提供了数据宽度和操作类型。数据传输只用单个（宽）SIMD 寄存器的 load 和 store 进行。通过对64位寄存器进行拆分的做法可以使 SIMD 更快，为了满足更大容量的并行计算，增加寄存器的宽度width成为了首选目标。考虑到扩展 SIMD 寄存器需要同时扩展 SIMD 指令集。将 SIMD 寄存器宽度和 SIMD 指令数量翻倍使得SIMD的应用前景不是很明朗。

向量架构也可以满足数据级并行性，向量计算机从内存中中收集数据并将它们放入长的，顺序的向量寄存器中。在这些向量寄存器上，流水线执行单元可以高效地执行运算。然后，向量架构将结果从向量寄存器中取出，并将其并分散地存回主存。向量寄存器的大小由实现决定，而不是像 SIMD 中那样嵌入操作码中。向量体系结构的关键所在是将向量的长度和每个时钟周期可以进行的最大操作数分离。向量微架构可以灵活地设计数据并行硬件而不会影响到程序员，程序员可以不用重写代码就享受到长向量带来的好处。此外，向量架构比 SIMD 架构拥有更少的指令数量。

在 SIMD 中，ISA 架构师在设计过程中决定了每个时钟周期可以并行操作的最大数据数和每个寄存器的元素个数。相比之下，RV32V 处理器设计人员无需更改 ISA 或编译器就可以选择它们的值，而对于 SIMD，寄存器每增加一倍都会使 SIMD 指令的数量翻倍，并且需要修改 SIMD 编译器。这种隐藏的灵活性意味着相同的 RV32V 程序不用改变，就可以 在最简单或最复杂的向量处理器上运行。

二 The RISC-V Vector ISA

2.1 Vector ISA介绍

RISCV的矢量指令集有有32个vector registers，它们的名称以 v 开头，分别是(v0，…，v31)，矢量扩展还增加了5个CSRs (vstart，vxsat，vxrm，vtype，vsew)到标准RISC-V ISA。如果标准RISC-V ISA不支持浮点，则额外增加一个fcsr的寄存器，用于保持vxsat和vxrm CSRs的镜像值。每一个寄存器都含有一个标量（scalar），矢量（vector）或矩阵（matrix / shape），每个向量寄存器都有一个关联的类型（多态编码）。每个矢量寄存器的元素个数不同。该数量取决于操作的宽度和专用于向量寄存器的存储大小，而这取决于处理器的设计者。比方，如果处理器为向量寄存器分配了 4096 个字节，则这足以让这些 32 个矢量寄存器中有 16 个 64 位元素，或者 32 个 32 位元素，或者 64 个 16 位元素，或 128 个 8 位元素。

支持矢量扩展的每个hart都有三个参数的定义。单个矢量元素的最大位数(ELEN)，必须是2的幂，即2^n。矢量寄存器的位数(VLEN)要大于或等于ELEN，同时也是必须是2的幂，即2^n。位的分条间距(SLEN)要符合VLEN ≥ SLEN ≥ 32，同时也是必须是2的幂，即2^n。平台配置文件可以对这些参数进行进一步的约束，例如，要求ELEN≥max(XLEN,FLEN)，或要求最小VLEN值，或固定一个SLEN值。XLEN为系统的操作位数，如32位或64位，FLEN位浮点操作的位数。

为了在向量 ISA 中保持元素数量的灵活性，向量处理器会计算会最大向量长度 （mvl），即在给定的容量限制下，向量程序使用这个向量寄存器可以运算的最大向量长度。向量长度寄存器（vl）为特定操作设定了向量中含有的元素数量，这有助于数组维度不是 mvl 的整数倍时的编程。VL（Vector Length register）控制了所有矢量指令，所有矢量指令都可以在掩码（mask）下被执行，Vector ISA拥有直观的内存排序模型和精确的异常处理支持。向量存储指令支持线性，跨步和聚集/分散访问模式。

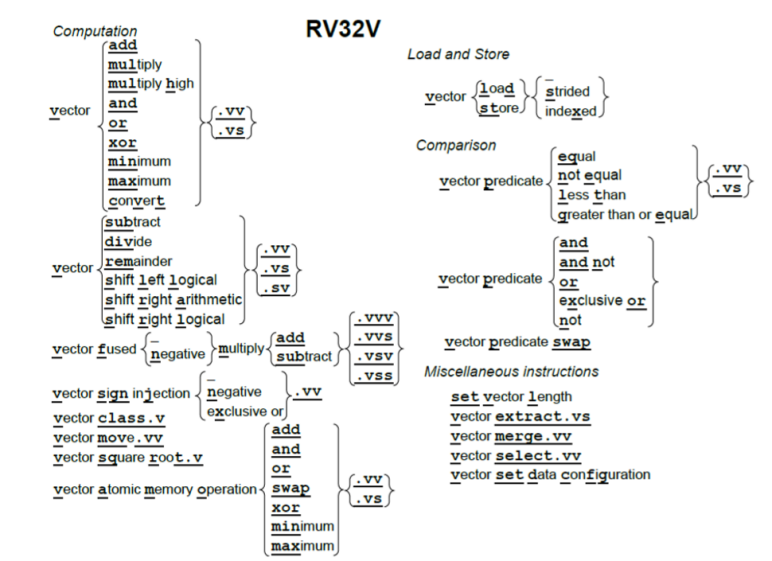


图1：RV32V和RV64V 的指令图示

图1是RV32V 和RV64V的指令图示。由于采用了动态寄存器类型，因此RV32V和RV64V的指令图示是一样的。图1 中的指令继承了来自 RV32I、RV32M、RV32F、RV32D 和 RV32A 的操作。每个向量指令都有几种类型，具体取决于源操作数是否都是向量（.vv 后缀），或者源操作数包含一个向量和一个标量（.vs 后缀）。一个标量后缀意味着有一个操作数来自 x 或 f 寄存器，另一个来自向量寄存器（v）。例如 计算Y = a × X + Y。其中X和Y是向量，a是标量。对于向量-标量操作，rs1 域指定了要访问的标量寄存器。

对诸如减法和除法之类的非对称运算，他们还会使用向量指令的第三种变体。其中第一个操作数是标量，第二个是向量（.sv 后缀）。像Y = a − X这样的操作就会使用这种变体。这种变体对于加法和乘法等对称运算来说是多余的，因此这些指令没有.sv 的版本。融合的（fused）乘法-加法指令有三个操作数，因此它们有着最多的向量和标量选项的组 合：.vvv、.vvs，.vsv 和.vss。

2.2 矢量CSR寄存器介绍

**vstart**是矢量起始索引CSR寄存器，一个可读可写的CSR寄存器，指定矢量指令要执行的第一个元素的索引。通常，vstart只有在矢量指令陷阱时被硬件写入，同时vstart的值能反应陷阱时的元素(要不是同步异常就是异步中断)，并且在处理可恢复陷阱后，应在该元素上恢复执行。所有向量指令都定义为从vstart CSR中给定的元素号开始执行，使目标向量中的早期元素不受干扰，并在执行结束时将vstart CSR重置为零。如果vstart寄存器中的值大于或等于向量长度vl，则不执行任何元素操作，虽然目标向量结束时，vl上的元素被归零，而vstart寄存器被重置为零。vstart CSR被定义为只有足够的可写位来保存最大元素索引(比最大vlmax少一个或是lg2(VLEN)位)，vstart CSR的高位硬连线到零（读取零，写入被忽略）。

**vxrm**是矢量定点舍入模式寄存器，它保存一个两位可读可写的字段，用于表示舍入模式。矢量定点取整模式被赋予一个单独的CSR地址，以允许独立访问，但也作为fcsr高位字段的一个反映。系统如果没有包含浮点操作，那在使用矢量扩展时必须增加fcsr CSR寄存器。

**vxsat**是矢量定点饱和标志，vxsat CSR寄存器包含一个读写位，该位指示定点指令是否必须使输出值饱和，以此适应目标格式。

**vtype**是矢量型寄存器，矢量类型CSR寄存器是一个只读、位宽为XLEN的CSR寄存器，只能被vsetvl{i}指令更新，并提供用于解释向量寄存器文件内容的默认类型。矢量类型决定每个矢量寄存器中元素的安排和多个矢量寄存器的分组方式。早期的草案允许采用常规CSR写入方式来写入vtype。只允许vsetvl{i}指令进行vtype的更新，简化了vtype CSR寄存器的状态维护。vtype最小的基本实现只需要四位存储空间，vsew[1:0]两位,vlmul[1:0]两位。可以使用vsew[1:0]中的非法64位组合对非法值进行编码，而不需要额外的存储位。

**vsew**是矢量标准元素宽度，vsew中的值设置动态标准元素宽度（SEW）。默认情况下，向量寄存器被划分为VLEN/SEW个标准宽度元素。在基础向量扩展中，SEW只需要支持(XLEN, FLEN)中的最大值。

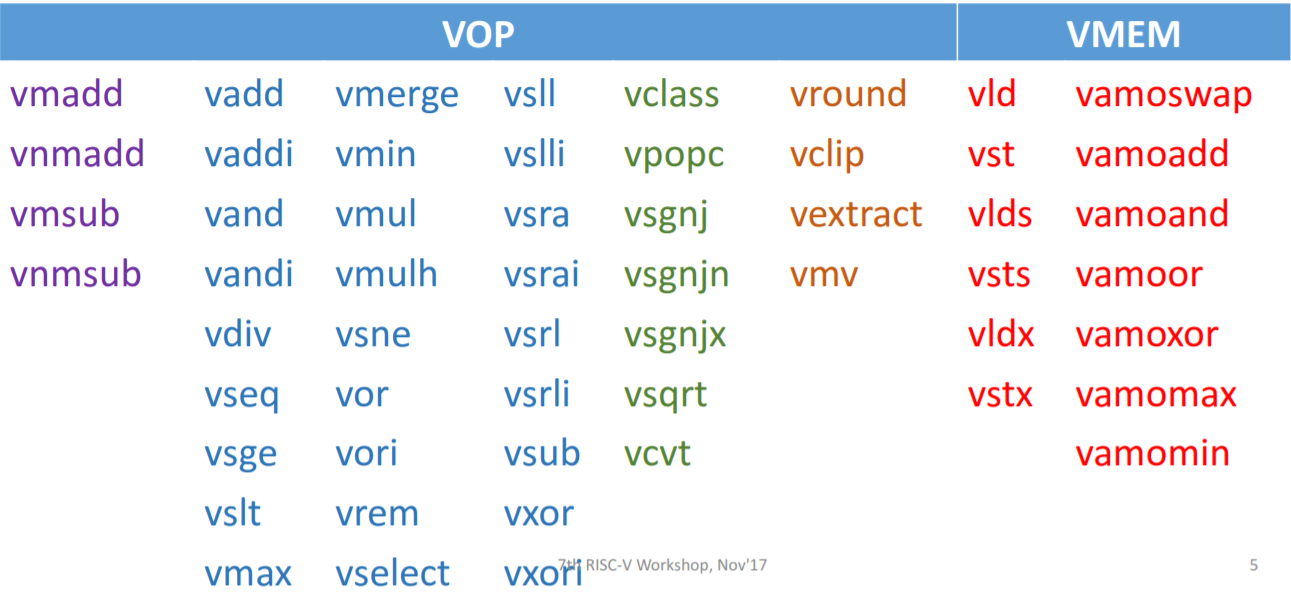


图2：Complete Vector Instruction List

图2是Roger Espasa在2017年11月17日的RISCV Workshop上分享的一个完整的矢量指令集分类表，我们暂时拿来做一个分类参照，随后会根据vector spec 0.8 和 0.9 draft中的最新指令情况进行重新分类总结。

2.3 两个矢量寄存器相加

本节我们介绍两个矢量相加（adding two vector registers），可以表示为vadd v1, v2 -> v0。当VL是0的时候，目的寄存器会被完全清空，然后对应的矢量寄存器进行逐位相加，我们以C++的如下代码来做解释，第一个for循环中，i的约束条件是初始值为0，同时要小于VL(Vector Length)，第二个for循环中，i的初始值为 vl，同时要小于MVL（Maximum Vector Length）,图3更加直观地描述了两个向量相加的过程。图4是SIMD并行计算，在这个例子中可以达到和矢量寄存器相同的效果。

其中VRF是向量寄存器堆（Vector Register File），每一个数据是单精度32位，即F32。图4中下面每一个执行完之后，都会写回到上面的VRF中。

*for (i = 0; i < vl; i++ )*

*{*

*v0[i] = v1[i] +v2[i]；*

*}*

*for (i = vl; i < MVL; i++ )*

*{*

*v0[i] = 0；*

*}*

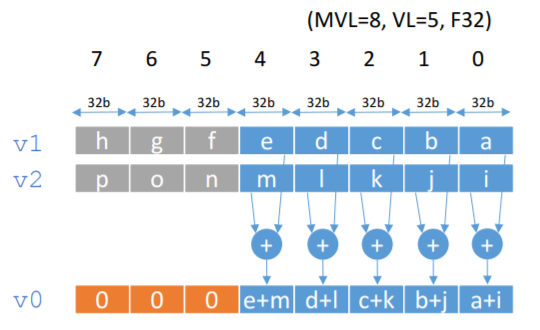


图3 矢量寄存器相加

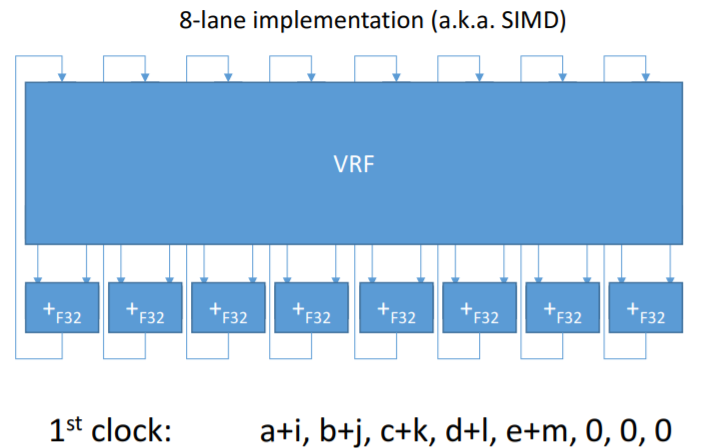


图4 SIMD并行计算

2.4 矢量和标量相加

本节我们介绍矢量和标量相加，可以表示为vadd v1, v2.s -> v0。在矢量规范中没有定义在矢量中的标量，应该具体存储位置，我们可以根据实际情况灵活选择。

for (i = 0; i < vl; i++ )

{

v0[i] = v1[i] +F32 v2[0]

}

for (i = vl; i < MVL; i++ )

{

v0[i] = 0

}

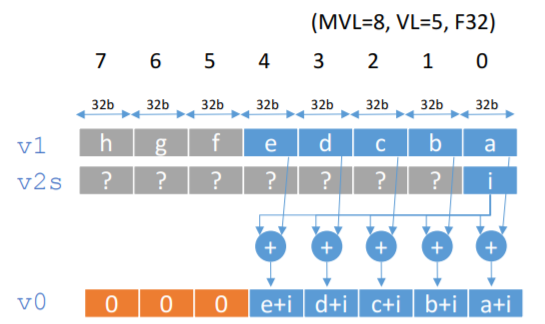


图5 矢量和标量相加

2.5 掩码计算

掩码（Masks）存储在矢量寄存器中，每一个元素的LSB（Least Significant Bit)设置为布尔值“0”或“1”，其他的Bits位可以忽略。掩码用比较操作(vseq，vsne，vslt，vsge)来计算，可以表示为 veq v6, v7 à-> v1，比较结果是整型Integer的“0”或“1”。指令使用编码中的两位（2 bits ）来选择掩码计算，其中

• 00 : 表示没有掩码计算 （假设掩码是0xFFFF…FFFF）

• 01 : 不在掩码计算中定义使用

• 10 : 用v1的 LSB作掩码

• 11 : 用v1的 LSB作掩码

vadd v3, v4, v1.t -> v5

for (i = 0; i < vl; i++ )

{

v5[i] = lsb(v1[i]) ? v3[i] +v4[i] : 0;

}

for (i = vl; i < MVL; i++ )

{

v5[i] = 0

}

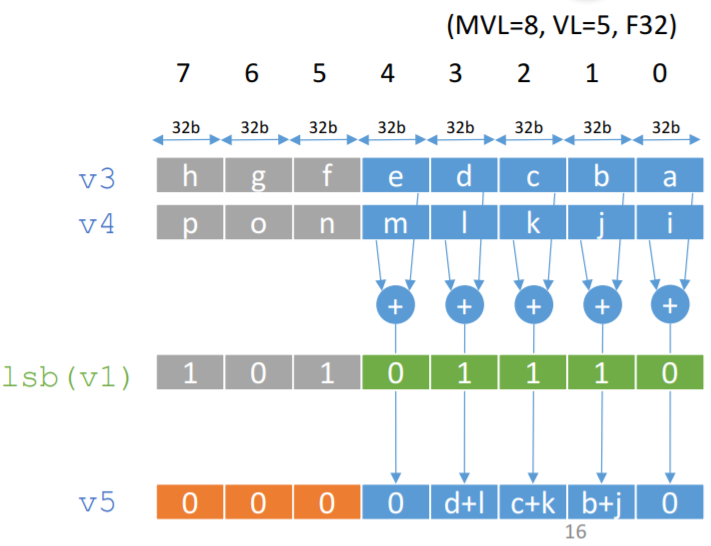


图6 矢量掩码计算

2.6 向量的Load和Store操作

向量 Load 和 Store 操作的最简单情况是处理按顺序存储在内存中的一维数组。向量 Load 用以 vld（vector load） 指令中地址为起始地址的顺序存储的数据来填充向量寄存器。向量寄存器的 数据类型确定数据元素的大小，向量长度寄存器 vl 中设置了要取的元素数量。向量 store 执行 vld 的逆操作。

例如，如果 a0 中存有 1024，且 v0 的类型是 X32，则 vld v0, 0(a0)会生成地址 1024， 1028，1032，1036，……直到达到由 vl 设置的限制。

对于多维数组，某些访问不是顺序的。如果二维数组以行优先序存储，且对列元素进 行顺序访问，则相邻列元素之间的地址差正好是行大小。向量架构通过跨步数据传输来支 持 vlds 和 vsts 数据访问。对于 vlds 与 vsts，虽然可以通过将步长设置为元素大小来达到 与 vld 和 vst 相同的效果，但 vld 和 vst 保证了所有的访问都是顺序的，这可以提供更高 的内存带宽。另一个原因是，对于常见的按单位步长访问，使用 vld 和 vst 可以缩减代码 长度，并减少执行的指令数。毕竟使用 vlds 和 vsts 指令来需要指定两个源寄存器，一个 给出起始地址，另一个给出以字节为单位的步长，而对于单位步长的访问，多花指令来设 置第二个寄存器，无遗是一种浪费。

例如，假设 a0 中的起始地址是地址 1024，且 a1 中行的长度是 64 字节。vlds v0, a0, a1 会将这个地址序列发送到内存：1024,1088(1024 + 1 × 64),1152(1024 + 2 × 64),1216(1024 + 3 × 64)，以此类推，直到向量长度寄存器 vl 告诉它停止。返回的数 据被顺序写入目标向量寄存器的各个元素。

到目前为止，我们都假设该程序在对密集数组进行操作。为了支持稀疏数组，向量架 构用 vldx 和 vstx 提供索引数据传输。这些指令的一个源寄存器是向量寄存器，另一个是 标量寄存器。标量寄存器具有稀疏数组的起始地址，向量寄存器的每个元素包含稀疏数组 的非零元素的字节索引。

假设 a0 中的起始地址是地址 1024，向量寄存器 v1 在前四个元素中有这些字节索 引：16，48，80，160。vldx v0, a0, v1 会将这个地址序列发送到内存：1040（1024+ 16）,1072(1024 + 48),1104(1024 + 80),1184(1024 + 160)。它将返回的数据顺序写入目 标向量寄存器的元素中。

以上我们把稀疏数组访问作为索引 Load 和 Store 操作的主要支持目标，但是还有许多 其他算法通过索引表来间接访问数据。

2.7 向量运算的条件执行

一些向量计算包括 if 语句。向量架构不依赖于条件分支，而是包含了一个掩码，这个 掩码禁止向量操作作用于某些元素。图 1 中的谓词指令在两个向量或向量和标量之间执 行条件测试，如果条件成立则在掩码向量的每一个元素中写入一个 1，反之写入 0。（掩码 向量必须和向量寄存器有相同的元素个数。）任何后续的向量指令都可以使用这个掩码。第 i 位为 1 表示元素 i 会被向量运算更改，为 0 表示该元素不会由向量运算改变。

RV32V 为掩码向量提供了 8 个向量谓词寄存器（vpi）。 vpand，vpandn，vpor， vpxor 和 vpnot 指令在它们之间执行逻辑运算，从而有效处理嵌套条件语句。 RV32V 指定 vp0 或 vp1 作为控制向量操作的掩码。要对所有元素执行一个正常的操 作，必须将这两个谓词寄存器中的一个设置为全1。RV32V 中有一条 vpswap 指令，用于将其他六个谓词寄存器的一个快速交换到 vp0 或 vp1。谓词寄存器也是动态启用的，禁用 它们可以快速清除所有谓词寄存器中的值。

例如，假设向量寄存器 v3 中的所有偶数元素都是负整数，所有奇数元素都是正整数。 考虑如下的代码：

vplt.vs vp0,v3,x0 # 将 v3 < 0 的掩码位置 1

add.vv vp0,v0,v1,v2 # 将 v0 的掩码为 1 的对应元素替换为 v1+v2

这段代码将把 vp0 中所有的偶数位设为 1，奇数位设为 0，并且将把 v0 中所有的偶数元素 替换为 v1 和 v2 中对应元素的和。v0 中的奇数元素不会改变.