第八章 向量

我追求简洁。我理解不了那些复杂的东西。——Seymour Cray

Seymour Cray（1925–1996）是 1976 年第一个采用向量架构的，并且在商业上取得成功的超级计算机 Cray-1 的架构师。Cray-1 是一颗明珠，即使没有使用向量指令，它也是世界上最快的计算机。

1997 年的英特尔多媒体扩展（MMX）使 SIMD 流行起来。它们通过 1999 年的流媒体 SIMD 扩展（SSE, Streaming SIMD Extensions）和 2010 年的高级向量扩展（AVX）得到了接受和扩展。MMX 的名声在英特尔的一则广告中得到了宣扬，该广告的内容是一种采用彩色干净套装的半导体产品线的光纤工作者在跳迪斯科(https://www.youtube.com/watch?v=paU16B-bZEA)。

8.1 导言

本章重点介绍数据并行，当存在大量数据可供应用程序同时计算时，我们称之为数据级并行性。数组是一个常见的例子。虽然它是科学应用的基础，但它也被多媒体程序使用。前者使用单精度和双精度浮点数据，后者通常使用 8 位和 16 位整数数据。

最著名的数据级并行架构是单指令多数据(SIMD，Single Instruction Multiple Data)。SIMD 最初的流行是因为它将 64 位寄存器的数据分成许多个 8 位、16 位或 32 位的部分，然后并行地计算它们。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。

把现有的 64 位寄存器进行拆分的做法由于其简单性而显得十分诱人。为了使 SIMD 更快，架构师随后加宽寄存器以同时计算更多部分。由于 SIMD ISA 属于增量设计阵营的一员，并且操作码指定了数据宽度，因此扩展 SIMD 寄存器也就意味着要同时扩展 SIMD 指令集。将 SIMD 寄存器宽度和 SIMD 指令数量翻倍的后续演进步骤都让 ISA 走上了复杂度逐渐提升的道路，这一后果由处理器设计者、编译器编写者和汇编语言程序员共同承担。

一个更老的，并且在我们看来更优雅的，利用数据级并行性的方案是向量架构。本章解释了我们在 RISC-V 中使用向量而不是 SIMD 的理由。

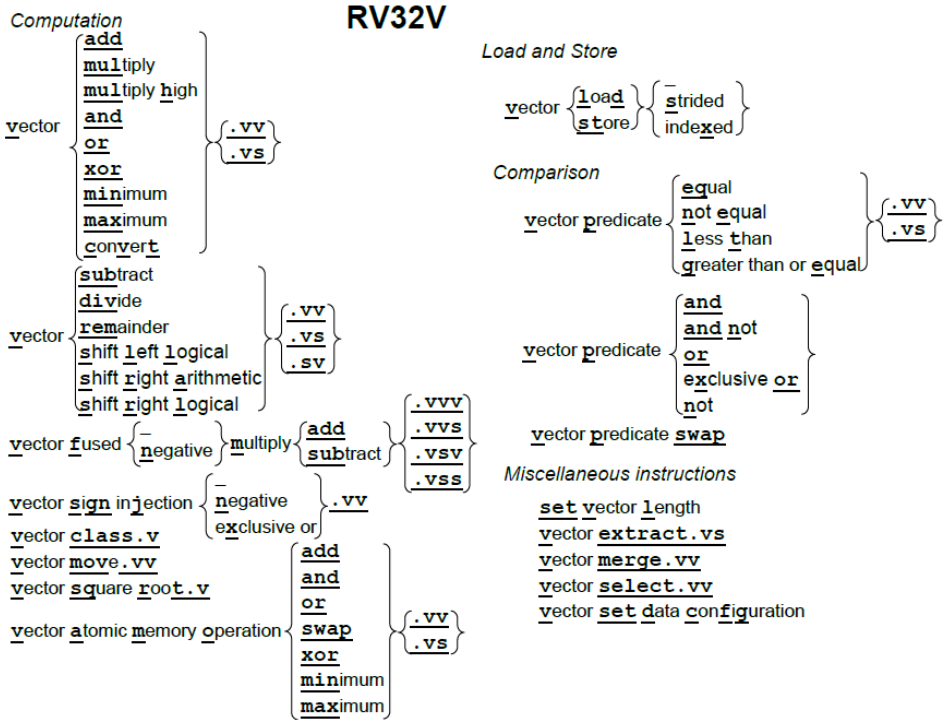


图 8.1：RV32V 的指令图示。由于采用了动态寄存器类型，这个指令图示也可以不加改变地用于第九章的

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。我们将会看到，*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，是向量体系结构的关键所在*：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。而且，与 SIMD 不同，向量架构有着完善的编译器技术。

向量架构比 SIMD 架构更少出现，因此知晓向量 ISA 的读者也更少。因此，本章会比前几章更加具有教程的风格。如果你想深入了解向量架构，请阅读[Hennessy and Patterson 2011]的第 4 章和附录 G。RV32V 还有一些简化了 ISA 的新颖功能。即使你已经熟悉了向量架构，也可能需要阅读我们的进一步解释。

8.2 向量计算指令

图 8.1 是 RV32V 扩展指令集的图形表示。RV32V 的编码尚未最终确定，所以本版不包含通常的指令布局图。

前面章节提到的每一个整数和浮点计算指令基本都有对应的向量版本：图 8.1 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）。比方说，我们的 DAXPY 程序（见第 55 页第五章图 5.7）计算$Y = a \times X + Y$。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。

读者可能会注意到，图 8.1 忽略了向量运算的数据类型和宽度。下一节解释了这么做的原因。

8.3 向量寄存器和动态类型

RV32V 添加了 32 个向量寄存器，它们的名称以 v 开头，但每个向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。比方说，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 个向量寄存器中有 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。

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。我们将在下面的小节中更详细地演示 mvl，vl 和 8 个谓词寄存器（vpi）的应用。

RV32V 采用了一种新颖的方法，即将数据类型和长度与向量寄存器而不是与指令操作码相关联。程序在执行向量计算指令之前用它们的数据类型和宽度标记向量寄存器。使用动态寄存器类型会减少向量指令的数量。这一点很重要，因为每个向量指令通常有六个整数版本和三个浮点版本，如图 8.1 所示。我们将在第 8.9 节看到，当我们面对众多的 SIMD 指令时，使用动态寄存器类型的向量架构减少了汇编语言程序员的认知负担以及编译器生成代码的难度。

动态类型的另一个优点是程序可以禁用未使用的向量寄存器。此功能可以将所有的向量存储器分配给已启用的向量寄存器。比如，假设只启用了两个 64 位浮点类型的向量寄存器，处理器有 1024 字节的向量寄存器空间。处理器将这些空间对半分，给每个向量寄存器 512 字节（512/8=64 个元素），因此将 mvl 设置位 64。因此我们可以看到，mvl 是动态的，但它的值由处理器设置，不能由软件直接改变。

源寄存器和目标寄存器决定了操作的类型和大小以及结果，因此动态类型隐含了转换。例如，处理器可以将双精度浮点数的向量乘以单精度标量，而无需先将操作数转换为相同的精度。这个额外的好处减少了向量指令的总数和实际执行的指令的数量。

可以用 vsetdcfg 指令来设置向量寄存器的类型。图 8.2 显示了 RV32V 可用的向量寄存器类型以及 RV64V 的更多类型（见第九章）。RV32V 要求向量浮点运算也有标量版本。因此，要使用 F32 类型，你也必须用到 RV32FV；要使用 F64 类型，你也必须用到 RV32FDV。RV32V 引入了 16 位浮点类型 F16。如果一个实现同时支持 RV32V 和 RV32F，则它必须支持 F16 和 F32 类型。

Type	Floating Point		Signed Integer		Unsigned Integer	
Width	Name	vetype	Name	vetype	Name	vetype
8 bits	–	–	X8	10 100	X8U	11 100
16 bits	F16	01 101	X16	10 101	X16U	11 101
32 bits	F32	01 110	X32	10 110	X32U	11 110
64 bits	F64	01 111	X64	10 111	X64U	11 111

图 8.2：RV32V 向量寄存器类型的编码。字段的最右边三位指示了数据的位宽，左边两位给出其类型。X64 和 U64 仅适用于 RV64V。F16 和 F32 需要 RV32F 扩展，F64 需要 RV32F 和 RV32D。F16 是 IEEE 754-2008 16 位浮点格式（binary16）。将 vetype 设置为 00000 会禁用向量寄存器。（本图基于[Waterman and Asanovic 2017]的表 17.4。）

为了避免上下文切换时间过慢，英特尔尽量避免在原始 MMX SIMD 扩展中添加寄存器。它只是重用现有的浮点寄存器，这意味着没有额外的上下文切换，但程序无法同时出现浮点和多媒体指令。

补充说明：RV32V 可以快速切换上下文

向量架构不如 SIMD 架构受欢迎的一个原因是：大家担心增加大型向量寄存器会延长中断时保存和恢复程序（上下文切换）的时间。动态寄存器类型对此很有帮助。程序员必须告诉处理器正在使用哪些向量寄存器，这意味着处理器需要在上下文切换中仅保存和恢复那些寄存器。RV32V 约定在不使用向量指令的时候禁用所有向量寄存器，这意味着处理器既可以具有向量寄存器的性能优势，又仅会在向量指令执行过程中发生中断时才会带来额外的上下文切换开销。早期的向量架构在发生中断时，不得不忍受保存和恢复全部向量寄存器的最大的上下文切换开销。

每个 load 和 store 指令都有一个 7 位的无符号立即数偏移量。它对于 load 按照目标寄存器的元素类型进行缩放，而对于 store 则按源寄存器缩放。

8.4 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 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：$1024, 1088(1024 + 1 \times 64), 1152(1024 + 2 \times 64), 1216(1024 + 3 \times 64)$，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。

带索引的 load 也称为收集(*gather*)；带索引的 store 通常称为分散(*scatter*)。

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。

8.5 向量操作期间的并行性

虽然简单的向量处理器一次操作一个向量元素，但由于元素操作根据定义是独立的，所以理论上处理器可以同时计算所有这些元素。RV32G 的数据位宽最大位 64 位，而如今的向量处理器通常在每个时钟周期内操作两个、四个或八个 64 位元素。当向量长度不是每个时钟周期执行的元素数量的倍数时，由硬件处理处理这些边缘情况。

与 SIMD 一样，对于较小数据的操作数量是较窄数据的位宽和较宽数据的位宽之比。因此，每个时钟周期计算 4 个 64 位操作的向量处理器通常每个时钟周期可以做 8 个 32 位，16 个 16 位或 32 个 8 位操作。

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。

8.6 向量运算的条件执行

当一个程序中的绝大部分操作都是用向量指令来实现的，那么这个程序被称为可向量化。Gather，scatter，以及谓词指令让更多的程序变得可向量化了。

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 8.1 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。

8.7 其他向量指令

除了之前提到过的设置向量寄存器数据类型的指令（vsetdcfg），其他指令还有 setvl，它将向量长度寄存器（vl）设置为源操作数和最大向量长度（mvl）中的较小值。选择较小值的原因是，在循环中我们需要判断这些向量代码到底是可以按最大向量长度（mvl）运行，还是要以一个较小值运行，从而能处理循环尾部剩下的元素。因此，为了处理循环尾部的元素，每次循环迭代都执行 setvl。

RV32V 中还有三条指令可以操作向量寄存器中的元素。

向量选择（vselect）按第二个源向量（索引向量）指定的元素位置，从第一个源数据向量中取得元素，从而生成一个新的结果向量：

```
# vindices 存有 0 到 mvl-1 的值，它们用来从 vsrc 中选取元素
vselect vedst, vsrc, vindices
```

因此，如果 v2 的前四个元素是 8、0、4、2，那么 vselect v0, v1, v2 将用 v1 的第 8 个元素替换 v0 的第 0 个元素；v1 的第 0 个元素替换 v0 的第 1 个元素；v1 的第 4 个元素替换 v0 的第 2 个元素；v1 的第 2 个元素替换 v0 的第 3 个元素。

向量合并（vmerge）类似于向量选择，但它用向量谓词寄存器来选择源向量中要用到元素。新的结果向量由根据谓词寄存器从两个源寄存器之一取得元素产生。若谓词向量寄

存器元素为 0，则新元素来自 vsrc1；如果为 1，则来自 vsrc2。

```
# vpo 的第 i 位决定 vdest 中新元素 i 来自 vsrc1（若第 i 位是 0）
# 还是 vsrc2（第 i 位为 1）
vmerge,vpo vdest, vsrc1, vsrc2
```

因此，如果 vpo 的前四个元素是 1、0、0、1，v1 的前四个元素是 1、2、3、4，v2 的前四个元素是 10、20、30、40，那么 vmerge,vpo v0, v1, v2 将把 v0 的前四个元素变为 10、2、3、40。

向量提取指令从一个向量的中间开始取元素，并将它们放在第二个向量寄存器的开头：

```
# start 是一个标量寄存器，其中存储着从 vsrc 中取元素的起始位置
vextract vdest, vsrc, start
```

例如，如果向量长度 vl 是 64，而 a0 的值是 32，那么 vextract v0,v1,a0 会把 v1 中的后 32 个元素复制到 v0 的前三十二个元素。

对于任意的二元结合运算符，可以利用 vextract 指令以递归减半的方法进行缩减运算。例如，要对向量寄存器的所有元素求和，可以用 vextract 将向量的后半部分复制到另一个向量寄存器的前半部分，这就将向量长度缩短了一半。接下来，将这两个向量寄存器加到一起，并将它们的和作为新一轮递归的操作数，直到向量长度减少到 1。此时第零个元素中的结果就是原向量寄存器中所有元素的和。

```
# a0 is n, a1 is pointer to x[0], a2 is pointer to y[0], fa0 is a
  0:  li t0, 2<<25
  4:  vsetdcfg t0              # enable 2 64b Fl.Pt. registers
loop:
  8:  setvl  t0, a0            # vl = t0 = min(mvl, n)
  c:  vld    v0, a1            # load vector x
 10:  slli   t1, t0, 3         # t1 = vl * 8 (in bytes)
 14:  vld    v1, a2            # load vector y
 18:  add    a1, a1, t1        # increment C pointer to x by vl*8
 1c:  vfmadd v1, v0, fa0, v1   # v1 += v0 * fa0 (y = a * x + y)
 20:  sub    a0, a0, t0        # n -= vl (t0)
 24:  vst    v1, a2            # store Y
 28:  add    a2, a2, t1        # increment C pointer to y by vl*8
 2c:  bnez   a0, loop          # repeat if n != 0
 30:  ret                      # return
```

图 8.3：图 5.7 中 DAXPY 程序的 RV32V 代码。没有出现机器语言是因为 RV32V 的操作码还未定义。

8.8 例子：用 RV32V 写成的 DAXPY 程序

RISC-V 中的 V 也代表向量。RISC-V 架构师在向量架构方面拥有丰富的经验，并且对 SIMD 在微处理器中的主导地位感到沮丧。因此，V 不仅代表这是第五个伯克利 RISC 项目，也意味着这个 ISA 会突出向量。

图 8.3 显示了用 RV32V 汇编写成的 DAXPY 程序（见第五章第 55 页图 5.7），我们一次解释一个步骤。

RV32V DAXPY 程序做的第一件事是启用这个函数需要的向量寄存器。它只需要两个向量寄存器保存 x 和 y 的部分元素，这些元素一个个都是 8 字节宽的双精度浮点数。第一条指令生成一个常量，第二条指令将它写入配置向量寄存器的控制状态寄存器（vcfgd），

从而获得两个 F64 类型的寄存器（见图 8.2）。根据定义，硬件按数字顺序分配配置好的寄存器，这样便有了 vo 和 v1。

假设我们的 RV32V 处理器由 1024 字节的空间专门用于向量寄存器。硬件平均地给这两个双精度浮点型（8 字节）的向量寄存器分配空间。每个向量寄存器有$512/8 = 64$个元素，因此处理器将此函数的最大向量长度（mvl）设置为 64。

循环中的第一条指令为接下来的向量指令设置向量长度。setvl 指令把 mvl 和 n 中的小值写入 vl 和 to。其中的深刻原因是，如果循环的迭代次数大于 n，那么这段代码最快可以一次处理 64 个值，所以把 mvl 的值写入 vl。如果 n 比 mvl 小，那么我们的读写不能超出 x 和 y 的范围，所以我们应该在循环最后一次迭代中只计算最后剩下的 n 个元素。setvl 还写入 to，用于保存 vl 的值，在地址为 10 的循环控制变量中会用到。

没有 setvl 的向量架构具有额外的类似*露天采矿*（意为降低效率）的代码，用于将 vl 设置为循环的最后 n 个元素，并检查 n 是否为

地址 c 处的指令 vld 是一个向量 load 操作，按照标量寄存器 a1 中存储的变量 x 的地址从 x 中取值。它把 x 的 vl 个元素从内存传输到 vo。下面的移位指令 slli 将向量长度乘以数据的宽度（8 字节），以便稍后用于递增指向 x 和 y 的指针。

地址 14 处的指令（vld）将来自内存的 vl 个元素 load 到 v1 中，接下来的一条指令（add）将指向 x 的指针进行了递增。

地址 1c 处的指令是最重要的部分。vfmadd 将 x（vo）中的 vl 个元素乘以标量 a（fo）并将每个乘积加上 y（v1）中的 vl 个元素，最后将这 vl 个和存回 y（v1）。

剩下的就是将结果存到内存中以及一些必须的循环开销。在地址 20 处的指令（sub）将 n（ao）的减去 vl，以记录在本次迭代中完成的操作数。接下来的一条指令（vst）将 vl 个结果写入内存中 y 数组中。地址 28 处的指令（add）将指向 y 数组的指针递增。接下来的指令判断 n（ao）是否为 0，若不是则继续循环，反之执行最后一条 ret 指令返回调用点。

向量架构的强大之处在于，这个包含 10 条指令的循环的每次迭代都会进行$3 \times 64 = 192$次访存操作和$2 \times 64 = 128$个浮点乘加（假设 n 至少为 64）。这意味着每条指令平均有 19 次访存和 13 次运算。我们将在下一节看到，SIMD 的这些数据要差一个数量级。

8.9 RV32V，MIPS-32 MSA SIMD 和 x86-32 AVX SIMD 的比较

ARM-32 有一个名为 NEON 的 SIMD 扩展，但它不支持双精度浮点指令，所以它对 DAXPY 没有帮助。

我们即将看到 SIMD 和向量架构执行 DAXPY 程序的对比。如果你换一种角度来看，也可以把 SIMD 视为有着短向量寄存器（8 个 8 位“元素”）的受限向量架构，但它没有向量长度寄存器，也没有跨步或索引数据传输。

MIPS SIMD 第 83 页的图 8.5 显示了 DAXPY 程序的 MIPS SIMD 架构（MSA）版本。由于 MSA 寄存器为 128 位宽，所以每个 MSA SIMD 指令可以操作两个双精度浮点数。

这种簿记代码被认为是向量架构中露天采矿代码的一部分。如图 8.5 的标题所示，向量长度寄存器 vl 使得这样的 SIMD 簿记代码对于 RV32V 没有实际意义。传统的向量架构需要额外的代码来处理 n = 0 的极端情况。RV32V 只是在 n = 0 时使向量指令像 nops 一样。

与 RV32V 不同，由于没有向量长度寄存器，MSA 需要额外的指令来检查 n 的值是否有问题。当 n 为奇数时，计算单个浮点数的乘-加运算需要额外的代码，因为 MSA 必须对成对的操作数进行操作。该代码位于图 8.5 的地址 3c 到 4c 处。尽管概率不大，但 n 也有可能为 0。在这种情况下，地址为 10 处的分支将跳过主计算循环。

如果没有在循环附近执行分支跳转，则地址为 18 处的指令（splati.d）把 a 的值同时放入 SIMD 寄存器 w2 的两半中。在 SIMD 中，要加一个标量数据，我们需要将其复制拓宽到与 SIMD 寄存器等宽。

在循环内部，地址为 1c 处的 ld.d 指令将 y 的两个元素 load 到 SIMD 寄存器 wo 中，然后将指向 y 的指针进行递增。然后它将 x 的两个元素 load 到 SIMD 寄存器 w1 中。接下

来地址 28 处的指令执行将指向 x 的指针进行递增，紧接在后面的是地址为 2c 处的最重要的乘加指令。

循环结束时的分支指令（带延迟槽）判断指向 y 的指针是否已经超出了 y 的范围。如果没有，循环继续。地址 34 处的延迟槽中的 SIMD store 指令将结果写入 y 的两个元素。

主循环终止后，代码检查 n 是否是奇数。若 n 是奇数，用第五章的标量指令执行最后一次乘加操作。最后一条指令返回到调用点。

MIPS MSA DAXPY 代码核心的循环部分包含了 7 条指令，执行了 6 次双精度访存操作和 4 次浮点乘加。平均每个指令大约有 1 个访存和 0.5 个运算操作。

ISA	MIPS-32 MSA	x86-32 AVX2	RV32FDV
Instructions (static)	22	29	13
Bytes (static)	88	92	52
Instructions per Main Loop	7	6	10
Results per Main Loop	2	4	64
Instructions (dynamic, n=1000)	3511	1517	163

图 8.4：向量 ISA 的 DAXPY 指令数和代码大小。他列出了指令总数（静态），代码大小，每个循环的指令数和运算结果数，以及执行的指令数（n = 1000）。带 MSA 的 microMIPS 将代码大小缩减到 64 字节，RV32FDCV 将代码缩减到 40 字节。

x86 SIMD 在 84 页的图 8.6 的代码中我们可以看到，Intel 公司经历了多代 SIMD 扩展。SSE 扩展到了 128 位 SIMD，带来了 xmm 寄存器和可以使用这些寄存器的指令；AVX 的一部分带来了 256 位 SIMD，以及 ymm 寄存器及其指令。

地址 0 到 25 的第一组指令从内存中 load 变量，在 256 位 ymm 寄存器中创建 a 的四个副本，并在进入主循环之前进行测试，以确保 n 至少为 4。这用到了两条 SSE 指令和一条 AVX 指令。（图 8.6 的标题中有更详细的解释）

主循环是 DAXPY 计算的核心。地址 27 处的 AVX 指令 vmovapd 将 x 的 4 个元素 load 到 ymm0 中。地址 2c 处的 AVX 指令 vfmadd213pd 将 a（ymm2）乘以 x（ymm0）的 4 个元素的 4 个副本，加上 y 的四个元素（在内存中地址为 ecx+edx*8 处），并将 4 个和放入 ymm0。接下来地址 32 处的 AVX 指令 vmovapd 将 4 个结果存储到变量 y 中。随后的三条指令执行计数器的递增操作并在需要的时候重复循环。

与 MIPS MSA 的情况一样，地址 3e 和 57 之间的“边缘”代码处理了 n 不是 4 的倍数的情况。它用到了三个 SSE 指令。

x86-32 AVX2 DAXPY 代码中主循环的 6 条指令执行了 12 次双精度访存和 8 次浮点的乘法和加法操作。这样每条指令平均有约 2 次访存和 1 次运算。

补充说明：Illiac IV 最先显现了 SIMD 的编译复杂性

凭借 64 个并行的 64 位浮点单元（FPU），在摩尔定律发布之前，Illiac IV 计划拥有超过 100 万个逻辑门。它的架构师最初预测它每秒可以进行 10 亿次浮点运算（1000MFLOPS），但它的实际最好性能只有 15MFLOPS。它的成本从 1966 年估计的 800 万美元上升到了 1972 年的 3100 万美元（尽管只建造了计划的 256 个 FPU 中的 64 个）。该项目于 1965 年启动，但直到 1976 年（Cray-1 发布的那一年）才开始发挥实际作用。它可能是最臭名昭著的超级计算机，成为了十大工程灾难之一[Falk 1976]。

8.10 结束语

如果代码能向量化，最好的架构就是向量架构。

——Jim Smith 于 1994 年在国际计算机体系结构研讨会上的主旨演讲

图 8.4 总结了 RV32IFDV，MIPS-32 MSA 和 x86-32 AVX2 的 DAXPY 程序中的指令数和字节数。SIMD 架构程序中用于计算的代码比用于循环控制的代码要少不少。MIPS-32 MSA 和 x86-32 AVX2 代码中的三分之二到四分之三是 SIMD 开销：这些额外的代码要么是在为主 SIMD 循环准备数据，要么是在 n 不是 SIMD 寄存器中浮点数个数的倍数时处理那些边缘元素。

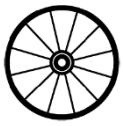


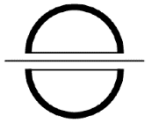
图 8.3 中的 RV32V 代码不需要这样的循环控制代码，因此它的指令数量少了一半。与 SIMD 不同，RV32V 有一个向量长度寄存器，使得不论 n 为何值，向量指令都可以工作。你可能会觉得 n 为 0 时 RV32V 会出现问题。实际上它不会，因为 RV32V 中的向量指令在 vl=0 时不会做出任何改变。

但是，SIMD 和向量处理之间的最为显著的区别不在于代码的长短。SIMD 执行的指令数比 RV32V 多 10 到 20 倍，因为每个 SIMD 循环在向量模式下只操作 2 到 4 个元素，而不是 RV32V 的 64 个元素。额外的取指和译码意味着在执行相同任务时要耗费更多的能量。

将图 8.4 中的结果与第五章中第 29 页的图 5.8 中的 DAXPY 的标量版本进行比较，我们发现 SIMD 大概使得代码的指令数和字节数加倍，但主循环的大小相同。执行的动态指令的数量以 2 或 4 的因子减少，这取决于 SIMD 寄存器的宽度。然而，RV32V 的向量代码大小变为原来的 1.2 倍（主循环 1.4 倍），但动态指令数是原来的 1/43！

即使动态指令的数量差别很大，但在我们看来，这并不是 SIMD 和向量架构的最主要的差异。没有向量长度寄存器会让指令数和循环控制代码暴增。像 MIPS-32 和 x86-32 这些遵循增量主义的 ISA 必须每次在将 SIMD 寄存器宽度翻倍时，都复制所有那些为较窄的 SIMD 寄存器定义的指令。于是不出意外地，在许多代 SIMD ISA 的传承中一共创造了数百条 MIPS-32 和 x86-32 指令，而且将来还会有数以百计的新指令出现。汇编语言程序员一定因这种粗暴的 ISA 演变方式而承担了难以承受的认知负担。像 vfmadd213pd 这样的指令，谁能记住它的含义并知道什么时候要用它？

相比之下，RV32V 代码不受向量寄存器的可用存储空间的大小影响。如果向量内存变大，不仅 RV32V 不会改变，而且你甚至不用重新编译。处理器提供了最大向量长度 mvl 的值，因此无论处理器将用于向量的存储空间从 1024 字节提升到了 4096 字节，还是将其降低到 256 字节，图 8.3 中的代码都不受影响。



不同于 SIMD 中由 ISA 指示所需的硬件，而且更改 ISA 意味着更改编译器那样，RV32V ISA 允许处理器设计人员为其应用分配合适资源用于数据并行，而不必影响程序员或编译器。可以说 SIMD 违反了第一章中将 ISA 架构和实现分离开来的 ISA 设计原则。

我们认为 RV32V 的模块化向量实现对比 ARM-32、MIPS-32 和 x86-32 的增量式 SIMD 架构在成本-能耗-性能、复杂度和编程简易性等方面的极大优势，可能是选用 RISC-V 的最有说服力的论据。

8.11 扩展阅读

H. Falk. What went wrong V: Reaching for a gigaflop: The fate of the famed Illiac IV was shaped by both research brilliance and real-world disasters. *IEEE spectrum*, 13(10):65–70, 1976.

J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

A. Waterman and K. Asanovi´c, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL https://riscv.org/specifications/.

注记

http://parlab.eecs.berkeley.edu

```
# a0 is n, a2 is pointer to x[0], a3 is pointer to y[0], $w13 is a
00000000 <daxpy>:
  0: 2405fffe  li       a1,-2
  4: 00852824  and      a1,a0,a1        # a1 = floor(n/2)*2 (mask bit 0)
  8: 000540c0  sll      t0,a1,0x3       # t0 = byte address of a1
  c: 00e81821  addu     v1,a3,t0        # v1 = &y[a1]
 10: 10e30009  beq      a3,v1,38        # if y==&y[a1] goto Fringe (t0==0 so n is 0 | 1)
 14: 00c01025  move     v0,a2           # (delay slot) v0 = &x[0]
 18: 78786899  splati.d $w2,$w13[0]     # w2 = fill SIMD register with copies of a
Loop:
 1c: 78003823  ld.d     $w0,0(a3)        # w0 = 2 elements of y
 20: 24e70010  addiu    a3,a3,16         # increment C pointer to y by 2 Fl.Pt. numbers
 24: 78001063  ld.d     $w1,0(v0)        # w1 = 2 elements of x
 28: 24420010  addiu    v0,v0,16         # increment C pointer to x by 2 Fl.Pt. numbers
 2c: 7922081b  fmadd.d  $w0,$w1,$w2      # w0 = w0 + w1 * w2
 30: 1467fffa  bne      v1,a3,1c         # if (end of y != ptr to y) go to Loop
 34: 7bfe3827  st.d     $w0,-16(a3)      # (delay slot) store 2 elts of y
Fringe:
 38: 10a40005  beq      a1,a0,50         # if (n is even) goto Done
 3c: 00c83021  addu     a2,a2,t0         # (delay slot) a2 = &x[n-1]
 40: d4610000  ldc1     $f1,0(v1)        # f1 = y[n-1]
 44: d4c00000  ldc1     $f0,0(a2)        # f0 = x[n-1]
 48: 4c206b61  madd.d   $f13,$f1,$f13,$f0  # f13 = f1 + f0 * f13 (muladd if n is odd)
 4c: f46d0000  sdc1     $f13,0(v1)       # y[n-1] = f13 (store odd result)
Done:
 50: 03e00008  jr       ra               # return
 54: 00000000  nop                       # (delay slot)
```

图 8.5：图 5.7 中 DAXPY 的 MIPS-32 MSA 代码。将此代码与图 8.3 中的 RV32V 代码进行比较时，SIMD 的循环控制开销显而易见。MIPS MSA 代码的第一部分（地址 0 到 18）复制了 SIMD 寄存器中的标量变量 a，并在进入主循环之前执行确保 n 至少为 2 的检查。当 n 不是 2 的倍数时，MIPS MSA 代码的第三部分（地址 38 到 4c）处理了这种边界情况情况。在 RV32V 中不需要这样的代码，因为向量长度寄存器 vl 和 setvl 指令使得该循环的代码适用于 n 的所有值，不论是奇数还是偶数。

```
# eax is i, n is esi, a is xmm1, pointer to x[0] is ebx, pointer to y[0] is ecx
 00000000 <daxpy>:
    0: 56                   push   esi
    1: 53                   push   ebx
    2: 8b 74 24 0c          mov    esi,[esp+0xc]   # esi = n
    6: 8b 5c 24 18          mov    ebx,[esp+0x18]  # ebx = x
    a: c5 fb 10 4c 24 10    vmovsd xmm1,[esp+0x10] # xmm1 = a
   10: 8b 4c 24 1c          mov    ecx,[esp+0x1c]  # ecx = y
   14: c5 fb 12 d1          vmovddup xmm2,xmm1     # xmm2 = {a,a}
   18: 89 f0                mov    eax,esi
   1a: 83 e0 fc             and    eax,0xfffffffc  # eax = floor(n/4)*4
   1d: c4 e3 6d 18 d2 01    vinsertf128 ymm2,ymm2,xmm2,0x1 # ymm2 = {a,a,a,a}
   23: 74 19                je     3e              # if n < 4 goto Fringe
   25: 31 d2                xor    edx,edx         # edx = 0
 Loop:
   27: c5 fd 28 04 d3       vmovapd ymm0,[ebx+edx*8] # load 4 elements of x
   2c: c4 e2 ed a8 04 d1    vfmadd213pd ymm0,ymm2,[ecx+edx*8] # 4 mul adds
   32: c5 fd 29 04 d1       vmovapd [ecx+edx*8],ymm0 # store into 4 elements of y
   37: 83 c2 04             add    edx,0x4
   3a: 39 c2                cmp    edx,eax          # compare to n
   3c: 72 e9                jb     27               # repeat loop if < n
 Fringe:
   3e: 39 c6                cmp    esi,eax          # any fringe elements?
   40: 76 17                jbe    59               # if (n mod 4) == 0 goto Done
 FringeLoop:
   42: c5 fb 10 04 c3       vmovsd xmm0,[ebx+eax*8] # load element of x
   47: c4 e2 f1 a9 04 c1    vfmadd213sd xmm0,xmm1,[ecx+eax*8] # 1 mul add
   4d: c5 fb 11 04 c1       vmovsd [ecx+eax*8],xmm0 # store into element of y
   52: 83 c0 01             add    eax,0x1          # increment Fringe count
   55: 39 c6                cmp    esi,eax          # compare Loop and Fringe counts
   57: 75 e9                jne    42 <daxpy+0x42>  # repeat FringeLoop if != 0
 Done:
   59: 5b                   pop    ebx              # function epilogue
   5a: 5e                   pop    esi
   5b: c3                   ret
```

图 8.6：图 5.7 中 DAXPY 的 x86-32 AVX2 代码。地址 a 处的 SSE 指令 vmovsd **把 a load 到 128 位** xmm1 **寄存器的一半。地址 14 处的 SSE 指令** vmovddup **将 a 复制到** xmm1 **的全部两半，以用于接下来的 SIMD 计算。地址 1d 处的 AVX 指令** vinsertf128 **从** xmm1 **中的 a 的两个副本，在** ymm2 **中生成 a 的四个副本。地址 42 到 4d 的三个 AVX 指令（**vmovsd，vfmadd213sd，vmovsd**）处理mod（n，4）≠ 0 的情况。它们以一次一个元素的方式执行 DAXPY 操作，循环在这个函数正好进行了 n 次乘-加操作的时候停止。再提一次，RV32V 不需要这样的代码，因为向量长度寄存器** vl **和** setvl **指令使得那些循环代码适用于 n 为任意值的情况。**