SIMD,RISC-V V Extension and Zvabd

寄存器模型、配置层级与执行范式

关天扬

2026 年 1 月 13 日

x86 SIMD 演进：从 MMX 到 SSE

x86 的 SIMD 发展史，是寄存器宽度不断扩展的“军备竞赛”。

- **MMX (MultiMedia Extensions) - 1997**
  - **背景**: 90 年代中期多媒体需求爆发（VCD、早期 3D 游戏）。
  - **宽度**: 64-bit。
  - **缺陷**: 借用浮点单元 (FPU) 寄存器，导致 CPU 无法同时处理浮点和 MMX 指令（上下文切换开销大）。
- **SSE (Streaming SIMD Extensions) - 1999**
  - **背景**: Pentium III 时代，针对 3D 性能加强。
  - **宽度**: 128-bit (XMM 寄存器)。
  - **改进**: 引入独立寄存器文件，不再占用 FPU。
  - **演进**: 最初专注单精度浮点 (4x `float`)，后续 SSE2-SSE4 补全了整数和双精度支持。

x86 SIMD 演进：AVX 时代的困境

随着高清视频编解码和科学计算需求，128-bit 捉襟见肘。

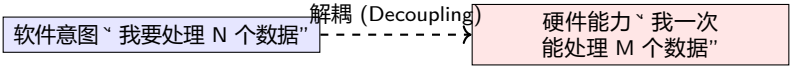
- **AVX (Advanced Vector Extensions) - 2011**
  - **宽度**: 256-bit (YMM 寄存器)，8x `float` 或 4x `double`。
  - **改进**: 引入三操作数指令 ($C = A + B$)，非破坏性操作减少了寄存器搬运开销。
- **AVX-512 - 2013 至今**
  - **宽度**: 512-bit (ZMM 寄存器)。
  - **问题 1 (功耗)**: 电路过密，运行 AVX-512 时 CPU 需大幅降频防止过热，有时总性能甚至不如 AVX2。
  - **问题 2 (碎片化)**: 衍生出太多子集 (F, CD, ER, PF, VL, BW, DQ...)，软件适配极为痛苦。
  - **设计哲学**: 依然是“定长寄存器”思维。

ARM SIMD：从 NEON 到 SVE 的范式转变

- **NEON - 2004**
  - 类似于 x86 SSE/AVX，固定 128-bit 宽度。
  - 注重能效比，紧密集成于处理器流水线。
- **SVE (Scalable Vector Extension) - 2016**
  - **背景:** 日本富士通“富岳”超算的需求。NEON 无法满足高性能计算。
  - **革命性创新:** 变长向量 (Vector Length Agnostic, VLA)。
  - **特点:** 硬件支持 128 到 2048 位任意 128 倍数位宽。
  - **SVE2 (ARMv9):** 将 SVE 技术下放到通用计算领域（手机、PC）。

RISC-V Vector (RVV): 回归与重塑

传统 SIMD 面临指令集碎片化、二进制不兼容、硬件升级成本高等问题。RVV 1.0 采用 **VLA (Vector Length Agnostic)** 设计范式：



编译器和程序员不再硬编码寄存器宽度（如 512 位），而是编写适应任意宽度的代码。

物理参数的二元性：VLEN 与 ELEN

RVV 硬件实现由两个不可变参数定义：

VLEN (Vector Register Length)

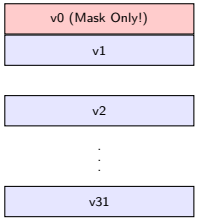
- 物理向量寄存器的位宽 (bits)。
- 必须是 2 的幂。
- 范围：嵌入式 (Zve) 64/128-bit $\leftrightarrow$ HPC 512+ bit。
- **意义**: 对软件透明。代码不假设 VLEN 具体值。

ELEN (Maximum Element Length)

- 硬件支持的最大元素位宽 (如 64-bit)。
- 定义了 ALU 数据通路的上限。
- **解耦**: 允许 “宽寄存器 (大 VLEN) + 窄计算单元 (小 ELEN)” 的设计，通过时间切片 (Time-slicing) 换取面积/成本优势。

寄存器文件布局与 v0 的特殊地位

- **通用寄存器:** 32 个 (v0 - v31)。
- **v0 的特殊角色:**
  - 唯一的**掩码操作数 (Mask Operand)** 寄存器。
  - 只有 v0 可以作为指令中的 `vm` (mask) 源。
- **设计权衡:**
  - 指令编码中掩码位仅占 1 bit (`vm`)。
  - 若允许任意寄存器做掩码，需额外 5 bits 编码空间，这在 32-bit 指令集中极其昂贵。
- **规则:** 凡是需要掩码的指令，掩码必须预先加载到 v0。



vtype：RVV 的控制论核心

RVV 指令是**多态 (Polymorphic)** 的。同一条 `vadd.vv` 的行为取决于 `vtype` CSR。

位域	名称	描述
Bit [XLEN-1]	**vill**	**Illegal**. 非法配置熔断器。若请求硬件不支持的配置 (如 ELEN=32 请求 64 位运算)，置 1。后续向量指令触发异常。
Bit 7	**vma**	**Mask Agnostic**. 掩码无关策略 (1=Agnostic, 0=Undisturbed)。
Bit 6	**vta**	**Tail Agnostic**. 尾部无关策略 (1=Agnostic, 0=Undisturbed)。
Bits [5:3]	**vsew**	**Selected Element Width**. 选中元素宽度。
Bits [2:0]	**vlmul**	**Vector Length Multiplier**. 向量长度倍增器。

SEW 与形式化变量 SEW_pow

SEW (Selected Element Width)

- 定义寄存器被“切分”的粒度 (8, 16, 32, 64 bits)。
- 直接影响计算精度和 SIMD 并行度。

SEW_pow (LLVM/Sail 中的派生变量)

- 代表 SEW 以 2 为底的幂次。
- $SEW = 8 \rightarrow SEW\_pow = 3 \quad (2^3 = 8)$
- $SEW = 64 \rightarrow SEW\_pow = 6 \quad (2^6 = 64)$
- **硬件意义:** 简化电路逻辑。计算元素偏移量时，硬件执行移位 (`index << SEW_pow`) 而非乘法。

LMUL：吞吐与寄存器压力的权衡

倍增模式 (LMUL > 1)

- 捆绑 2, 4, 8 个物理寄存器为一个逻辑组。
- **优势**: 极大提升单指令吞吐量 (VLMAX 翻倍)。摊薄取指译码开销。
- **代价**: 可用逻辑寄存器减少 (LMUL=8 时只剩 4 个逻辑寄存器)。

分数模式 (LMUL < 1)

- 使用寄存器的 $1/2$, $1/4$, $1/8$。
- **优势**: 节省寄存器资源，辅助混合宽度计算。

形式化变量 LMUL_pow

LMUL 的指数形式。$LMUL = 8 \to 3$; $LMUL = 1 \to 0$; $LMUL = 1/8 \to -3$。

VLMAX 计算公式

VLMAX 是当前配置下，单条指令能处理的最大元素个数。

$$VLMAX = \frac{VLEN}{SEW} \times LMUL$$

或者使用形式化变量：

$$VLMAX = \frac{VLEN}{SEW} \times 2^{\text{LMUL\_pow}}$$

核心平衡术

通过牺牲寄存器数量 (LMUL 增大)，换取单条指令的计算量 (VLMAX 增大)。

严格的对齐约束 (Alignment Constraints)

为了简化硬件重命名 (Renaming) 逻辑，RVV 强制要求：

- **LMUL = 2:** 索引必须是偶数 (v0, v2, v4...)。访问 v1 触发非法指令异常。
- **LMUL = 4:** 索引必须是 4 的倍数 (v0, v4, v8...)。
- **LMUL = 8:** 索引必须是 8 的倍数 (v0, v8, v16, v24)。

硬件视角

硬件在解码阶段只需检查索引低几位是否为 0，即可快速判断合法性。这保证了物理寄存器堆 (PRF) 的块状分配，避免碎片化。

分数 LMUL 的战略意义：混合宽度运算

为何需要 LMUL < 1? **为了维持 SEW/LMUL 比率恒定。**

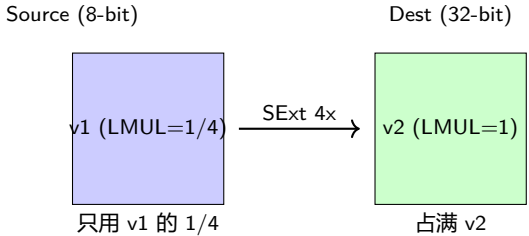
$$\frac{SEW_{src}}{LMUL_{src}} = \frac{SEW_{dst}}{LMUL_{dst}} \implies VLMAX_{src} = VLMAX_{dst}$$

场景： 将 8-bit 向量扩展为 32-bit (扩 4 倍)。

- 若源 $LMUL = 1$ (占 1 个寄存器)，目的需 $LMUL = 4$ (占 4 个)。
- 若源本来就是 $LMUL = 8$，目的需要 $LMUL = 32 \rightarrow$ **爆表!** (最大只支持 8)。

解法： 源使用 $LMUL = 1/4$，目的使用 $LMUL = 1$。

混合宽度运算图解



$\frac{8}{1/4} = 32$ vs $\frac{32}{1} = 32$

两者比率一致 → 元素个数 (vl) 一致

EEW (Effective Element Width)

绝大多数指令遵循全局 `vtype` 中的 SEW。但 Load/Store 和转换指令例外。

- **定义**: 指令显式指定的操作宽度。
- **例子**: `vle8.v` (Vector Load Element 8-bit)。
- **行为**:
  - 此指令执行时，硬件**忽略**全局 SEW。
  - 强制以 8-bit 为粒度访问内存和切分寄存器。
  - 仅影响该指令本身，不改变全局 CSR。

EMUL (Effective LMUL)

当指令 EEW $\neq$ 全局 SEW 时，硬件如何保证处理的元素个数与当前 *vl* 匹配？**自动调整分组系数 (EMUL)。**

$$EMUL = \frac{EEW}{SEW} \times LMUL$$

案例分析

当前配置: $SEW = 32, LMUL = 1$。执行 `vle8.v v1, (a0)`。

- $EEW = 8$
- $EMUL = (8/32) \times 1 = 1/4$

结果: 这条加载指令实际上只填充 v1 寄存器的低 $1/4$ 部分。

vsetvli 指令编码

配置变量如何塞进 32-bit 指令?

`vsetvli rd, rs1, vtypei`

- **vtypei (11-bit 立即数)**: 硬编码了 vtype 的各个域。
  - Bits [2:0] → **vlmul**
  - Bits [5:3] → **vsew**
  - Bits [6] → **vta**
  - Bits [7] → **vma**
- 这意味着大部分配置必须在**编译时**确定。
- 极少数动态配置需使用 `vsetvl` (寄存器版)。

vl (Vector Length) 与 Strip-Mining

vl 是只读 CSR，表示本轮循环处理的元素个数。

$$vl = \min(AVL, VLMAX)$$

- **AVL (Application Vector Length):** 应用程序总共想处理多少数据 (如 n)。
- **机制:** 编译器生成标准循环，硬件根据 VLEN 自动计算 vl。

```
# memcpy example: copy n bytes from a1 to a2
loop:
    vsetvli t0, a0, e8, m8, ta, ma  % 请求 AVL=a0, 使用最大分组 m8
                                    % t0 返回实际处理数量 (vl)
    vle8.v v0, (a1)                 % 加载 t0 个字节
    vse8.v v0, (a2)                 % 存储 t0 个字节
    sub a0, a0, t0                  % 剩余数量 a0 -= vl
    add a1, a1, t0                  % 指针前进
    add a2, a2, t0
    bnez a0, loop                   % 若仍有剩余，继续

```

此代码在 VLEN=128 和 VLEN=512 机器上均可跑出峰值性能。

num_elem：编译器与模拟器的视角

`num_elem` 并非架构 CSR，但在工具链中至关重要。

- **LLVM 视角**：对应类型系统 `<vscale x n x ty>` 中的 n。表示向量化因子。
- **模拟器 (Sail/Spike) 视角**：
  - 往往作为临时变量存储 *vl* 或 *VLMAX*。
  - 例如初始化掩码时，模拟器遍历 0 到 *num_elem* 个比特。
  - 代表了逻辑上“活跃”的元素总数。

vstart：精确异常 (Precise Exception) 的基石

向量指令耗时很长（可能涉及数千次访存）。若执行一半发生 Page Fault 怎么办？

- **没有 vstart:** 必须从头重做。对于 I/O 读取或原子操作，这不可接受（破坏幂等性）。
- **有 vstart:**
  1. 异常发生，硬件记录出错元素的索引到 `vstart`。
  2. OS 处理异常 (如换页)。
  3. 指令重新发射。
  4. 硬件读取 `vstart`，**自动跳过**前 `vstart` 个元素，从断点继续。

vstart 场景模拟

指令：`vle32.v v0, (a0)`，vl=200。

1. 硬件处理了前 99 个元素。
2. **第 100 个元素触发缺页异常。**
3. 硬件写入 `vstart = 100`，跳转到异常处理程序。
4. OS 修复页表，返回用户程序。
5. `vle32.v` 再次执行。
6. 硬件发现 `vstart=100`，直接从内存地址 `a0 + 100*4` 开始读取。
7. 指令完成后，`vstart` 自动清零。

掩码策略：Agnostic vs Undisturbed

RVV 允许软件通过 `vma` (Mask) 和 `vta` (Tail) 指定“无关元素”的行为。

- **Undisturbed (0):** 保持原值。
  - 语义：$v_d[i] = mask[i] ? result : v_d[i]$
  - **代价:** 引入**读后写 (WAR)** 依赖。指令必须读取目的寄存器的旧值。
  - 用途: 归约 (Reduction) 等需要保留累加器的场景。
- **Agnostic (1):** 覆写为全 1 (通常)。
  - 语义：$v_d[i] = mask[i] ? result : 11\ldots1$
  - **优势:** 硬件无需读取旧值。

Agnostic 对寄存器重命名的影响

在高性能乱序 (Out-of-Order) 处理器 (如玄铁 C910, SiFive X280) 中:

vta=1 (Agnostic) 的黄金法则

- 当 `vta=1` 时，硬件知道整个目的寄存器都将被“新数据”覆盖（不管有没有掩码）。
- **重命名 (Renaming):** 映射单元 (RAT) 可以直接分配一个**全新的物理寄存器**给目的操作数。
- **消除依赖:** 打破了与上一条指令的依赖链，允许指令全速乱序执行。

结论: 除非算法逻辑强制要求 (如累加)，否则永远使用 Agnostic 策略 (ta, ma)。

小结：RVV 的设计哲学

RISC-V 向量扩展不仅仅是一组新指令，更是一套自洽的数据并行计算哲学。

1. **VLEN 不可见**: 软件意图与硬件能力的彻底解耦。
2. **LMUL 分组**: 解决了吞吐量与寄存器压力的矛盾。
3. **SEW/EEW 动态宽度**: 解决了多精度计算的对齐难题。
4. **Agnostic 策略**: 解决了向量化与乱序执行的冲突。
5. **vstart**: 解决了长向量与精确异常的兼容性。

这些机制共同构成了一个既适合嵌入式 (IoT/DSP) 又适合高性能计算 (HPC/AI) 的现代 ISA。

现有 V 扩展局限性

在视频编码标准（如 H.264/AVC, H.265/HEVC, VVC）以及各种计算机视觉算法（如光流法、立体匹配）中，运动估计 (Motion Estimation) 是计算最密集的环节之一。运动估计的核心任务是在参考帧中寻找与当前帧最相似的图块，而衡量“相似度”的最常用指标便是绝对差和 (Sum of Absolute Differences, SAD) 。
数学上，对于两个 $N \times N$ 的像素块 A 和 B，SAD 定义为：

$$SAD(A, B) = \sum_{i=0}^{N^2-1} |A[i] - B[i]|$$

在缺乏专用硬件指令的情况下，在 RVV v1.0 基础指令集中实现整数绝对差 $|A - B|$ 需要极其繁琐的指令序列。

RVV v1.0 实现计算绝对差的方式

路径 A：减法与条件取反

- `vsub.vv`: 计算 $Diff = A - B$。
- `vmslt.vx`: 比较 $Diff < 0$，生成掩码 M。
- `vrsub.vx (masked)`: 在掩码 M 有效的元素位置，利用 $0 - Diff$ 进行取反；在掩码无效位置保持原值。

路径 B：最大值减最小值

- `vmin.vv`: 计算 $Min = \min(A, B)$。
- `vmax.vv`: 计算 $Max = \max(A, B)$。
- `vsub.vv`: 计算 $Result = Max - Min$。

通常需要 3 到 4 条向量指令才能完成一次绝对差计算，且不仅增加了指令发射端的压力，还占用了额外的向量寄存器作为临时存储，导致寄存器压力 (Register Pressure) 增大。考虑到 SAD 计算通常位于视频编码器的最内层循环中，这种指令膨胀直接导致了性能损耗和能效下降。

Zvabd 指令

```
# Signed Integer Absolute
vabs.v    vd, vs2, vm   # vd[i] = abs(vs2[i])
# Signed Integer Absolute Difference
vabd.vv    vd, vs2, vs1, vm   # vd[i] = abs(vs2[i] - vs1[i])
# Unsigned Integer Absolute Difference
vabdu.vv   vd, vs2, vs1, vm   # vd[i] = abs(vs2[i] - vs1[i])
# Widening signed absolute difference accumulate, overwrite addend
vwabdacc.vv vd, vs2, vs1, vm # vd[i] = abs(vs2[i] - vs1[i]) + vd[i]
# Widening unsigned absolute difference accumulate, overwrite addend
vwabdaccu.vv vd, vs2, vs1, vm # vd[i] = abs(vs2[i] - vs1[i]) + vd[i]
```

感谢观看

Questions?