# SIMD

# SIMD

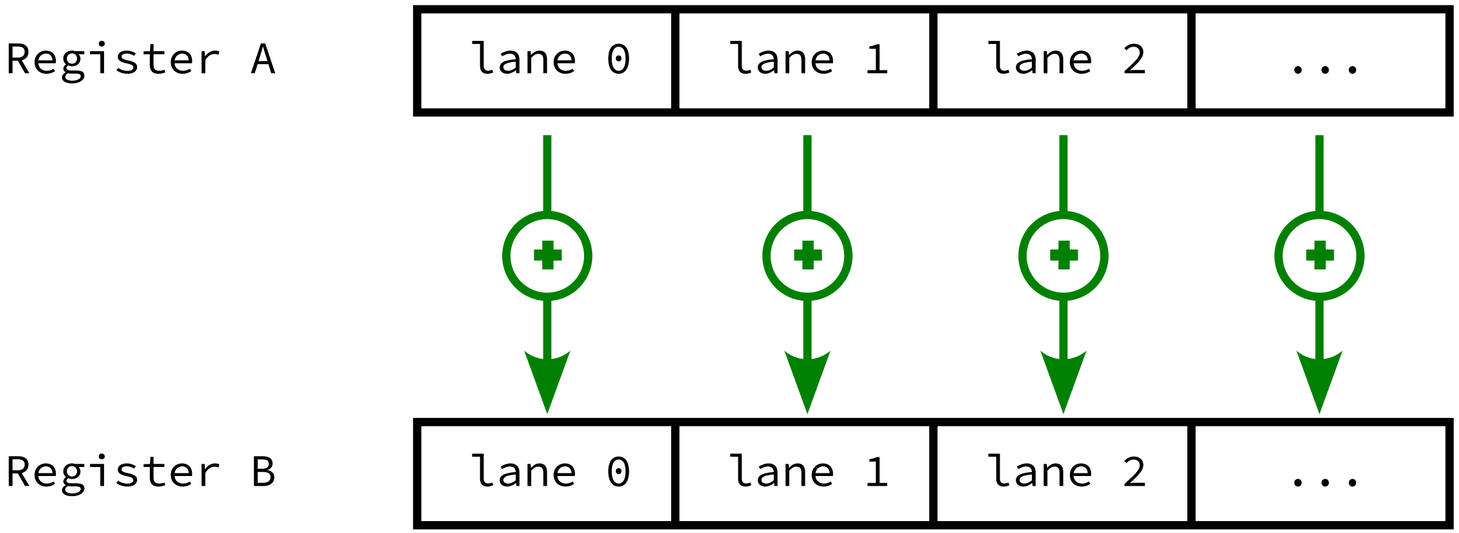
# SIMD

- SIMD stands for Single Instruction Multiple Data
- new, larger registers (in addition to the general purpose ones): “**vector** registers”

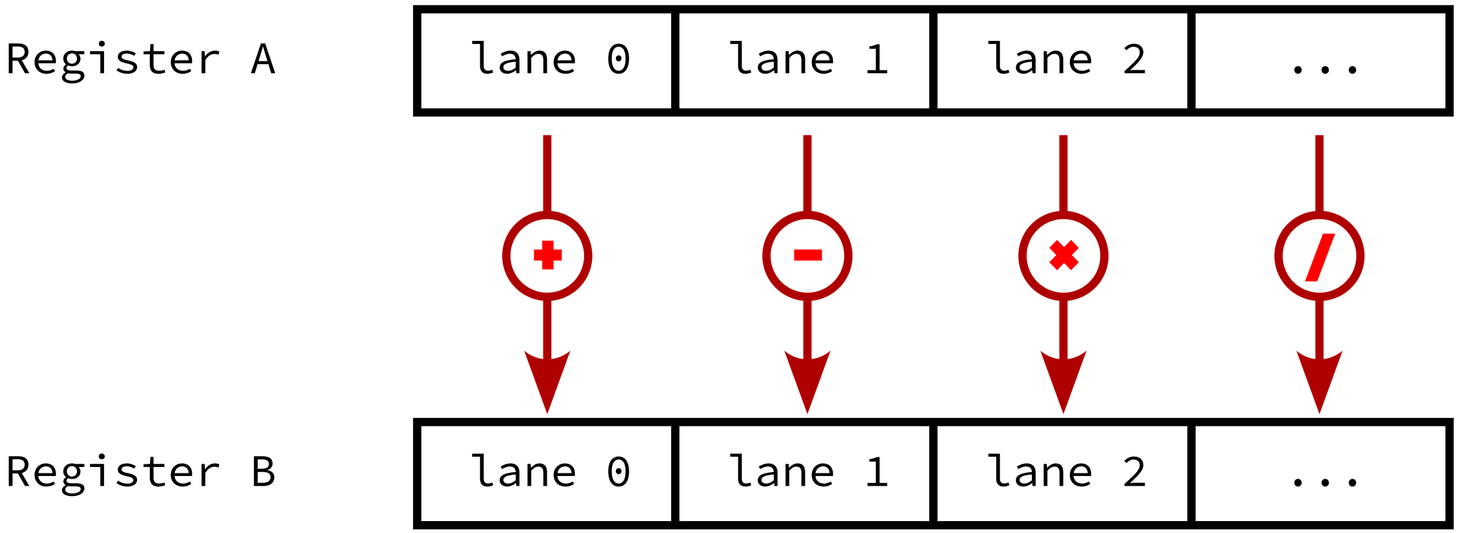
| bits | 255..224 | 223…192 | 191…160 | 159…128 | 127…96 | 95…64 | 63…32 | 31…0 |
|---|---|---|---|---|---|---|---|---|
| 256 | ymm0 | | | | | | | |
| 64 | fp64 #3 | | fp64 #2 | | fp64 #1 | | fp64 #0 | |
| 32 | fp32 #7 | fp32 #6 | fp32 #5 | fp32 #4 | fp32 #3 | fp32 #2 | fp32 #1 | fp32 #0 |
| 16 | | | | | | | | |
| 8 | | | | | | | | |

- **but**
  - SIMD registers cannot be treated as big integers

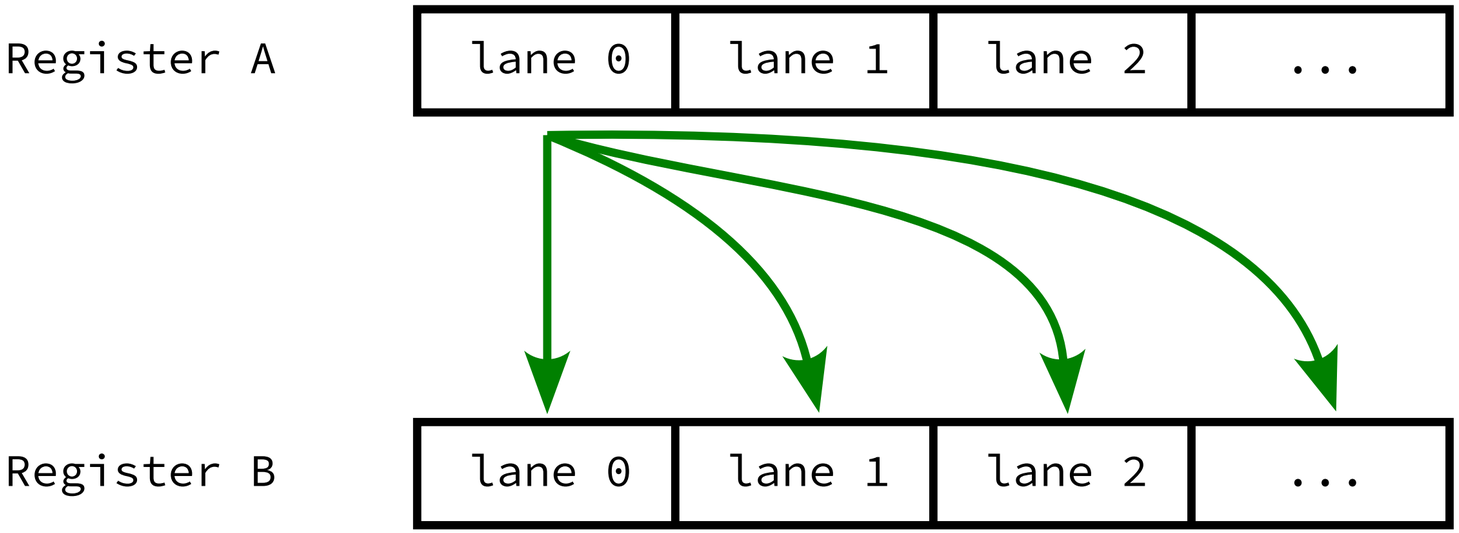
- individual “**lanes**” (8-, 16-, 32- or 64-bit parts) generally cannot be accessed individually



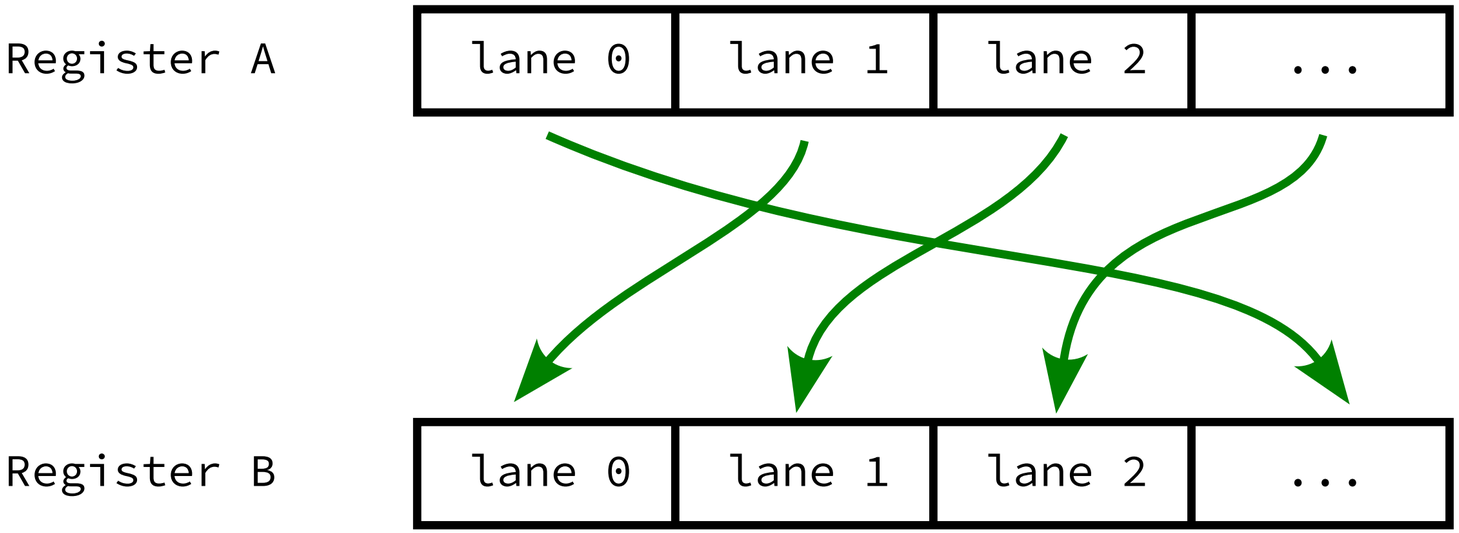

Allowed



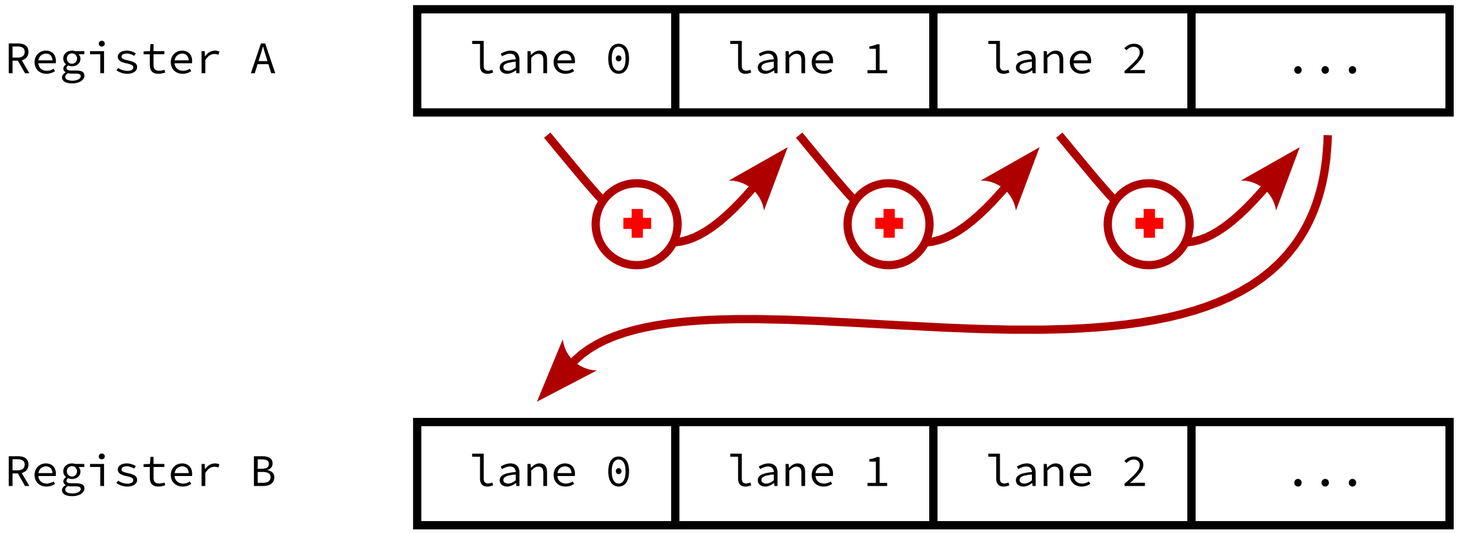

**Not** allowed




Allowed




Allowed




**Not** allowed

# SIMD registers

- On Intel (and AMD) ISAs:
  - SSE (~1999): 8 128-bit registers `xmm0` - `xmm7`
  - AVX (~2011): 16 256-bit registers `ymm0` - `ymm15`
  - AVX-512 (~2016, but not yet universal): 32 512-bit registers `zmm0` - `zmm31`

- On ARM:
  - Neon (~2005): 16 128-bit registers `Q0` - `Q15`
  - AMX (~2020, Apple-specific): 8 + 8 + 64 512-bit registers `x`, `y`, `z`

# Compiler autovectorization

# Example 1

```
void add_1234(float v[4])
{
    v[0] += 1.0;
    v[1] += 2.0;
    v[2] += 3.0;
    v[3] += 4.0;
}
```

```
add_1234:
        movups  xmm0, xmmword ptr [rdi]              ; xmm0 <- v
        addps   xmm0, xmmword ptr [rip + .LCPI0_0]   ; xmm0 <- xmm0 + { 1.0, 2.0, 3.0, 4.0 }   (4x 32-bits)
        movups  xmmword ptr [rdi], xmm0              ; v <- xmm0
        ret
```

The addition is performed with a single SIMD instruction

# Example 2

```
void many_ops(float v[4])
{
    v[0] += 1.0;
    v[1] -= 2.0;
    v[2] *= 3.0;
    v[3] /= 4.0;
}
```

```
many_ops:
        movss   xmm1, DWORD PTR [rdi+4]     ; xmm1 <- v[1]
        movss   xmm3, DWORD PTR .LC3[rip]   ;              xmm3 <- 0.25
        mulss   xmm3, DWORD PTR [rdi+12]    ;              xmm3 <- xmm3 * v[3]
        ("DIV")
        movss   xmm0, DWORD PTR .LC0[rip]   ;                                   xmm0 <- 1.0
        movaps  xmm2, xmm1                  ; xmm2 <- xmm1
        movss   xmm1, DWORD PTR .LC2[rip]   ;                                                 xmm1 <- 3.0
        addss   xmm0, DWORD PTR [rdi]       ;                                   xmm0 <- xmm0 + v[0]                       (ADD)
        mulss   xmm1, DWORD PTR [rdi+8]     ;                                                 xmm1 <- xmm1 * v[2]   (MUL)
        subss   xmm2, DWORD PTR .LC1[rip]   ; xmm2 <- xmm2 - 2.0                                                        (SUB)
        unpcklps        xmm0, xmm2
        unpcklps        xmm1, xmm3
        movlhps xmm0, xmm1
        movups  XMMWORD PTR [rdi], xmm0
        ret
```

(gcc 16.2, released August 2026)

The operations are performed by four non-SIMD instructions

# Example 2 (clang)

```
void many_ops(float v[4])
{
    v[0] += 1.0;
    v[1] -= 2.0;
    v[2] *= 3.0;
    v[3] /= 4.0;
}
```

```
many_ops:
        movups  xmm0, xmmword ptr [rdi]                 ; xmm0 <- { v[0], v[1], v[2], v[3] }
        movsd   xmm1, qword ptr [rip + .LCPI0_2]        ; xmm1 <- {  1.0, -2.0, [0.0, 0.0] }
        addps   xmm1, xmm0                              ; xmm1 <- xmm1 + xmm0                          ("ADD+SUB")
        mulps   xmm0, xmmword ptr [rip + .LCPI0_1]      ; xmm0 <- xmm0 * { 0.0, 0.0, 3.0, 0.25 }       ("MUL+DIV")
        movsd   xmm0, xmm1                              ; xmm0[0..1] <- xmm1[0..1]
        movupd  xmmword ptr [rdi], xmm0
        ret
```

(clang 23.1.0, released August 2026)

The operations are performed by two SIMD instruction

# Example 3

```
float horizontal(float v[4])
{
    return v[0] + v[1] + v[2] + v[3];
}
```

```
horizontal:
        movss   xmm0, DWORD PTR [rdi]       ; xmm0 <- v[0]
        addss   xmm0, DWORD PTR [rdi+4]     ; xmm0 <- xmm0 + v[1]
        addss   xmm0, DWORD PTR [rdi+8]     ; xmm0 <- xmm0 + v[2]
        addss   xmm0, DWORD PTR [rdi+12]    ; xmm0 <- xmm0 + v[3]
        ret
```

This operation is performed with three non-SIMD additions

# Example 4

```
void add(float v[4], float w[4])
{
    v[0] += w[0];
    v[1] += w[1];
    v[2] += w[2];
    v[3] += w[3];
}
```

```
add:
        vmovss  xmm0, dword ptr [rsi]
        vaddss  xmm0, xmm0, dword ptr [rdi]
        vmovss  dword ptr [rdi], xmm0               ; v[0] <- v[0] + w[0]
        vmovss  xmm0, dword ptr [rsi + 4]
        vaddss  xmm0, xmm0, dword ptr [rdi + 4]
        vmovss  dword ptr [rdi + 4], xmm0           ; v[1] <- v[1] + w[2]
        vmovss  xmm0, dword ptr [rsi + 8]
        vaddss  xmm0, xmm0, dword ptr [rdi + 8]
        vmovss  dword ptr [rdi + 8], xmm0           ; v[2] <- v[2] + w[2]
        vmovss  xmm0, dword ptr [rsi + 12]
        vaddss  xmm0, xmm0, dword ptr [rdi + 12]
        vmovss  dword ptr [rdi + 12], xmm0          ; v[3] <- v[3] + w[3]
        ret
```

The operation is performed by four non-SIMD additions

# Example 5

```
void add_r(float *restrict v, float *restrict w)
{
    v[0] += w[0];
    v[1] += w[1];
    v[2] += w[2];
    v[3] += w[3];
}
```

```
add_r:
        vmovups xmm0, xmmword ptr [rsi]
        vaddps  xmm0, xmm0, xmmword ptr [rdi]
        vmovups xmmword ptr [rdi], xmm0
        ret
```

This operation is performed by a single SIMD instruction

# SIMD intrinsics

# How to use SIMD

- Rely on compilers (“autovectorization”)
- Write assembly code
- Use compiler “intrinsics”
  - Intrinsics look like C functions

    but the compiler knows how to translate them to specific assembly code
  - > Intel intrinsics guide
  - > ARM intrinsics

# Intel intrinsics

Include:

```
#include <immintrin.h>
```

Types:

- `__m128`: 128-bit register, 4x fp32
- `__m256`: 256-bit register, 8x fp32
- `__m512`: 512-bit register, 16x fp32

- `__m128d`, `__m256d`, `__m512d`: 2x, 4x, 8x fp64
- `__m128i`, `__m256i`, `__m512i`: 128-, 256-, 512-bits integers (any lane size)

Function prefixes:

- `_mm_`: 128-bit SSE
- `_mm256_`: 256-bit AVX
- `_mm512_`: 512-bit AVX-512

First function suffix:

- `_s`: non-parallel (operate on lowest lane only)
- `_p`: packed / parallel (floating-point)
- `_ep`: packed / parallel (integers)

Second function suffix:

- `s`: single-precision fp32
- `d`: double-precision fp64
- `i8`, …, `i512`: signed integers
- `u8`, …, `u512`: unsigned integers

# Example

Function declaration:

```
__m256d _mm256_max_pd(__m256d a, __m256d b);
```

Operation:

```
dst[255:192] := MAX(a[255:192], b[255:192])
dst[191:128] := MAX(a[191:128], b[191:128])
dst[127: 64] := MAX(a[127: 64], b[127: 64])
dst[ 63:  0] := MAX(a[ 63:  0], b[ 63:  0])
```

> refer to the intrinsics guide

# ARM intrinsics

Include:

```
#include <arm_neon.h>
```

Types: `{lane type} {lane size} x {number of lanes} _t`

Lane type:

- `int`
- `uint`
- `float`

Examples:

- `uint16x8_t`: 8 lanes, each a 16-bit unsigned integers
- `float16x8_t`: 8 lanes, each a fp16

Function prefix: v

First function suffix: q for 128-bit ops

Additional function suffixes: return and argument lane types

- _f16, _f32, _f64: floating-point
- _s8, …, _s64: signed integer
- _u8, …, _u64: unsigned integer

# Example

Function declaration:

```
float64x2_t vmaxq_f64(float64x2_t a, float64x2_t b);
```

Operation:

```
dst[127: 64] := MAX(a[127: 64], b[127: 64])
dst[ 63:  0] := MAX(a[ 63:  0], b[ 63:  0])
```

> refer to the intrinsics guide