# Calcul haute performance pour les systèmes embarqués (HPEC)
# Lab 3 - SIMD instruction sets

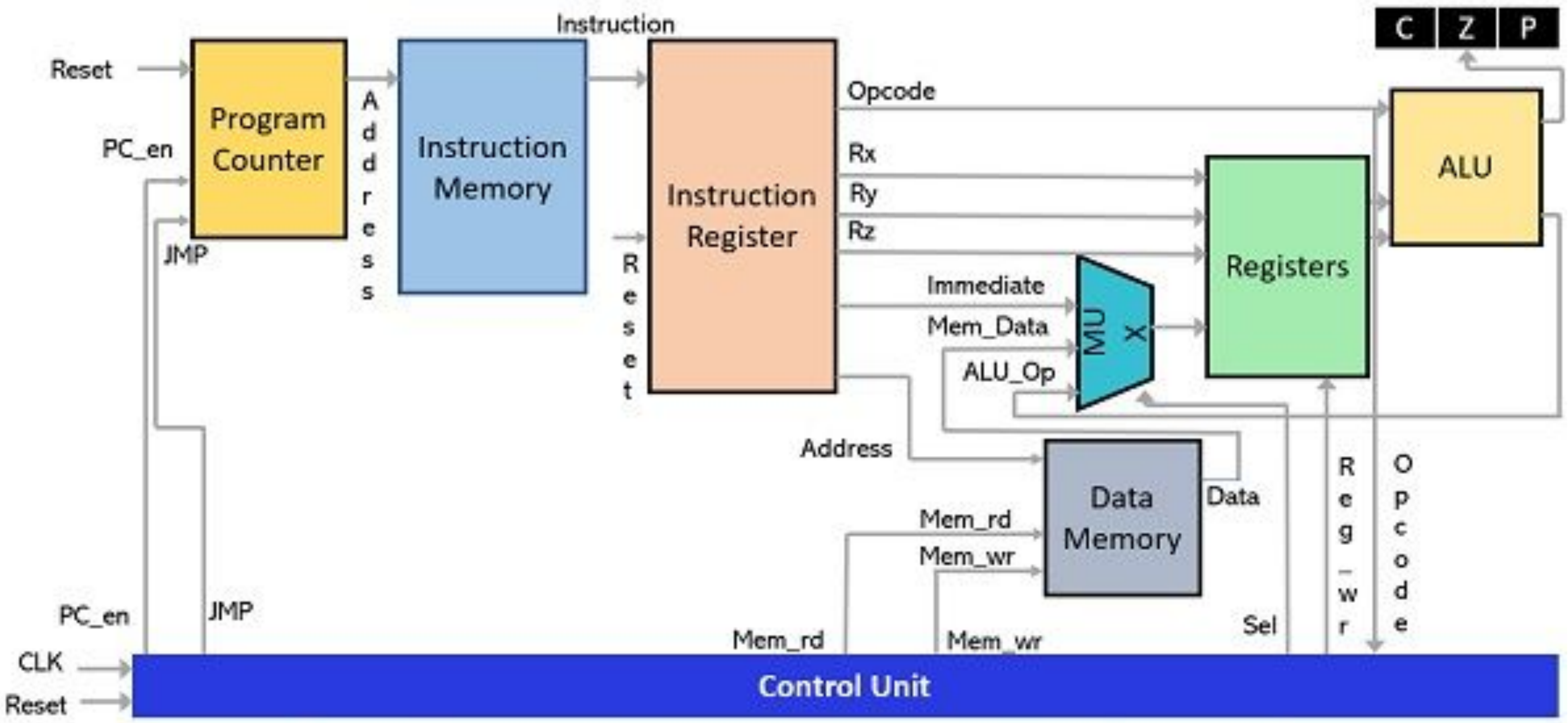
Bertrand LE GAL

IMS laboratory, CNRS UMR 5218
Digital Circuits and Systems team
Bordeaux-INP, University of Bordeaux
France

BORDEAUX INP Enseirb-Matmeca

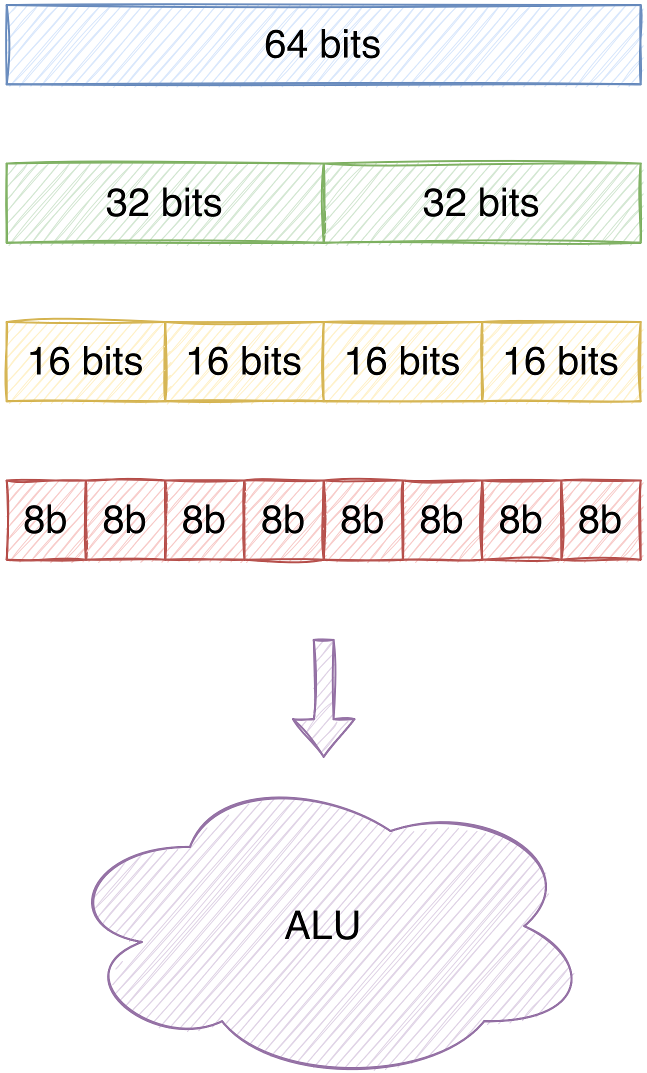
bertrand.legal@enseirb-matmeca.fr

# Processor architecture from 90's



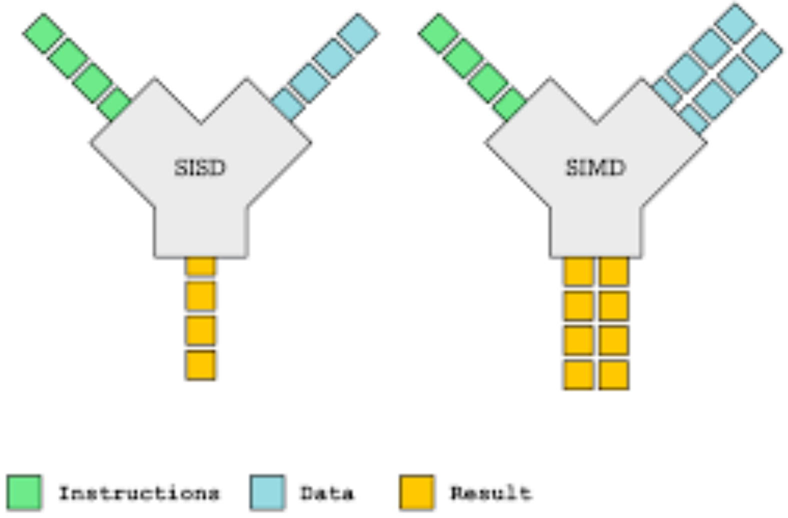
# Increasing the hardware usage rate

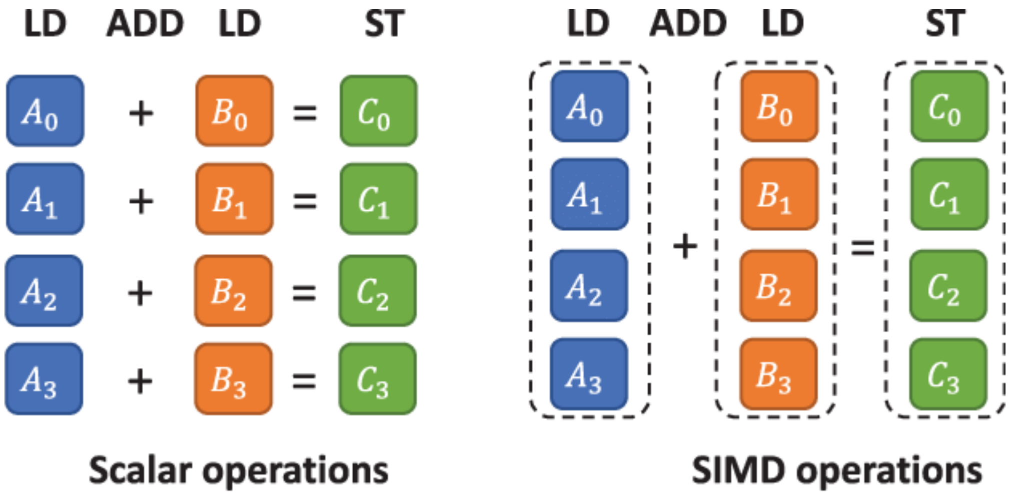
- Current processors are:
  - 64 bits word length,
  - $\geq$ 18 pipeline stages,
- It means that
  - Registers are 64 bits wide,
  - ALU ressources are 64 bits wide,
- In real life
  - Most data are $\leq$ 64 bits wide (pixels = 8b),
  - Reusing hardware should be possible,
- It is easy for logical operation, more complex for others.



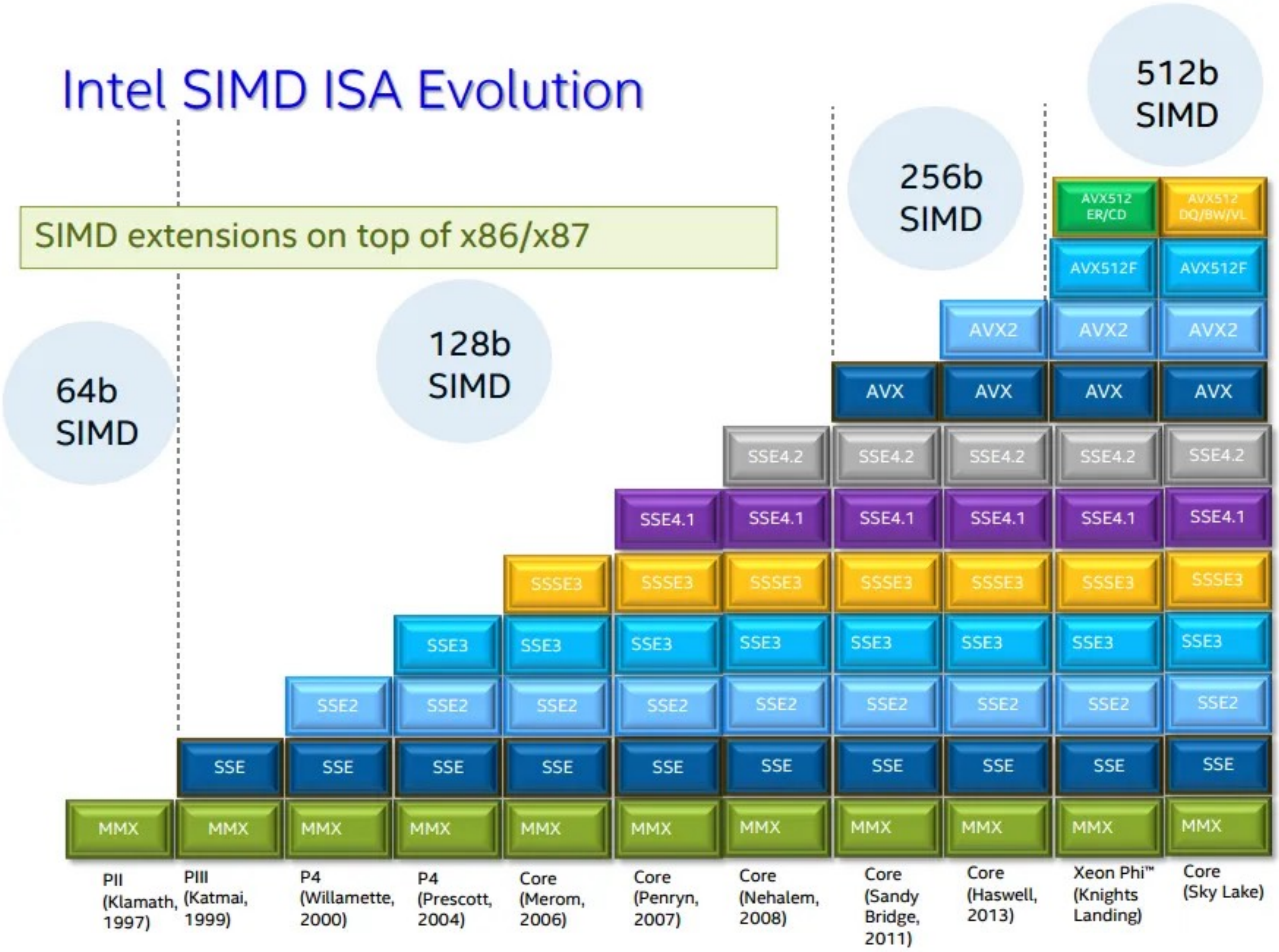
# SIMD feature for parallel computing

- Single Instruction Multiple Data (SIMD) paradigm
  - Appears at the beginning of 20’s,
  - Needed for video processing,
- INTEL MMX ALU
  - 32 bits processor
  - 64 bits integer ALU
  - Dedicated register bank,
- ALU operations
  - Data width ∈ {8, 16, 32, 64} bits.

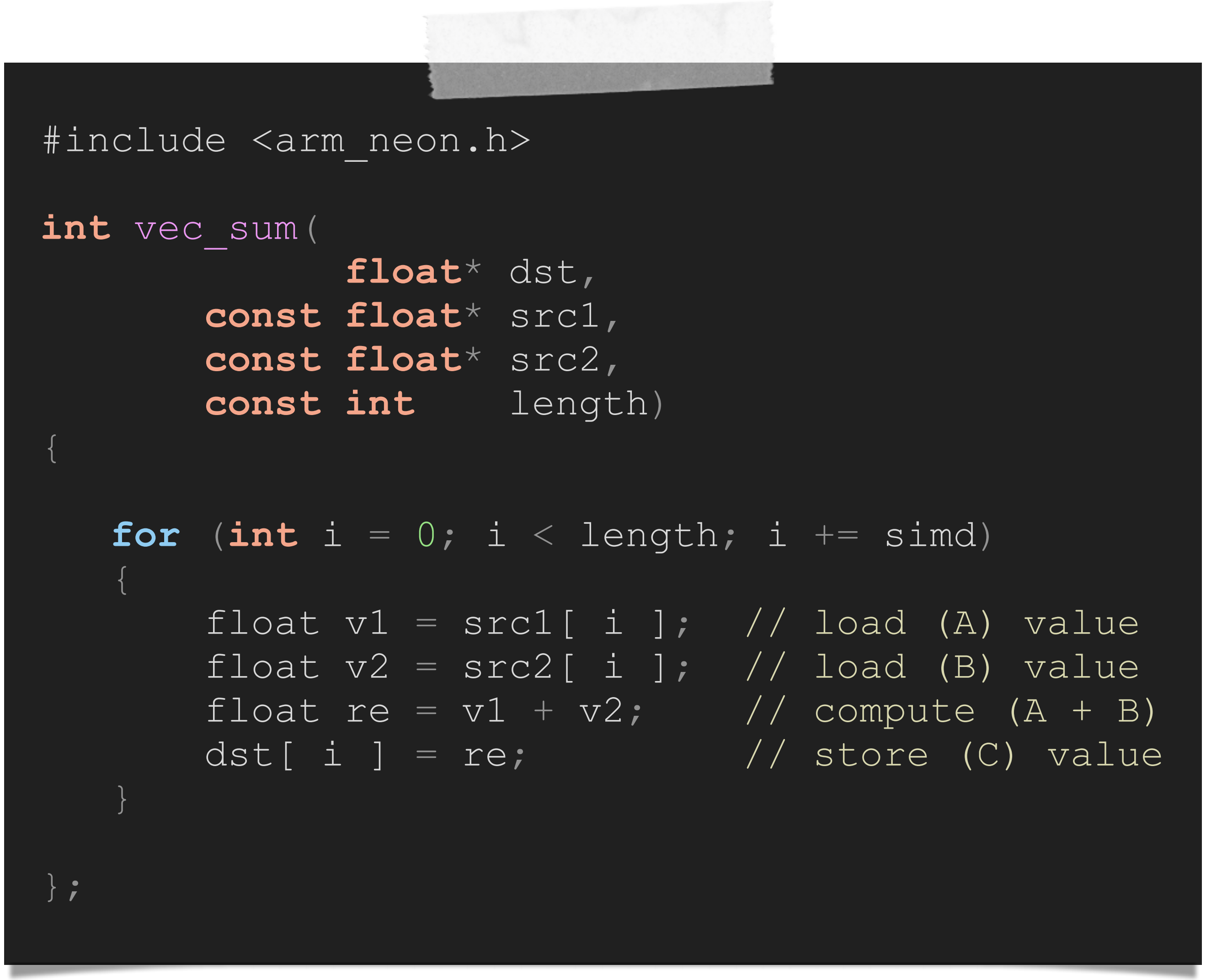




# Evolution des unités SIMD chez INTEL

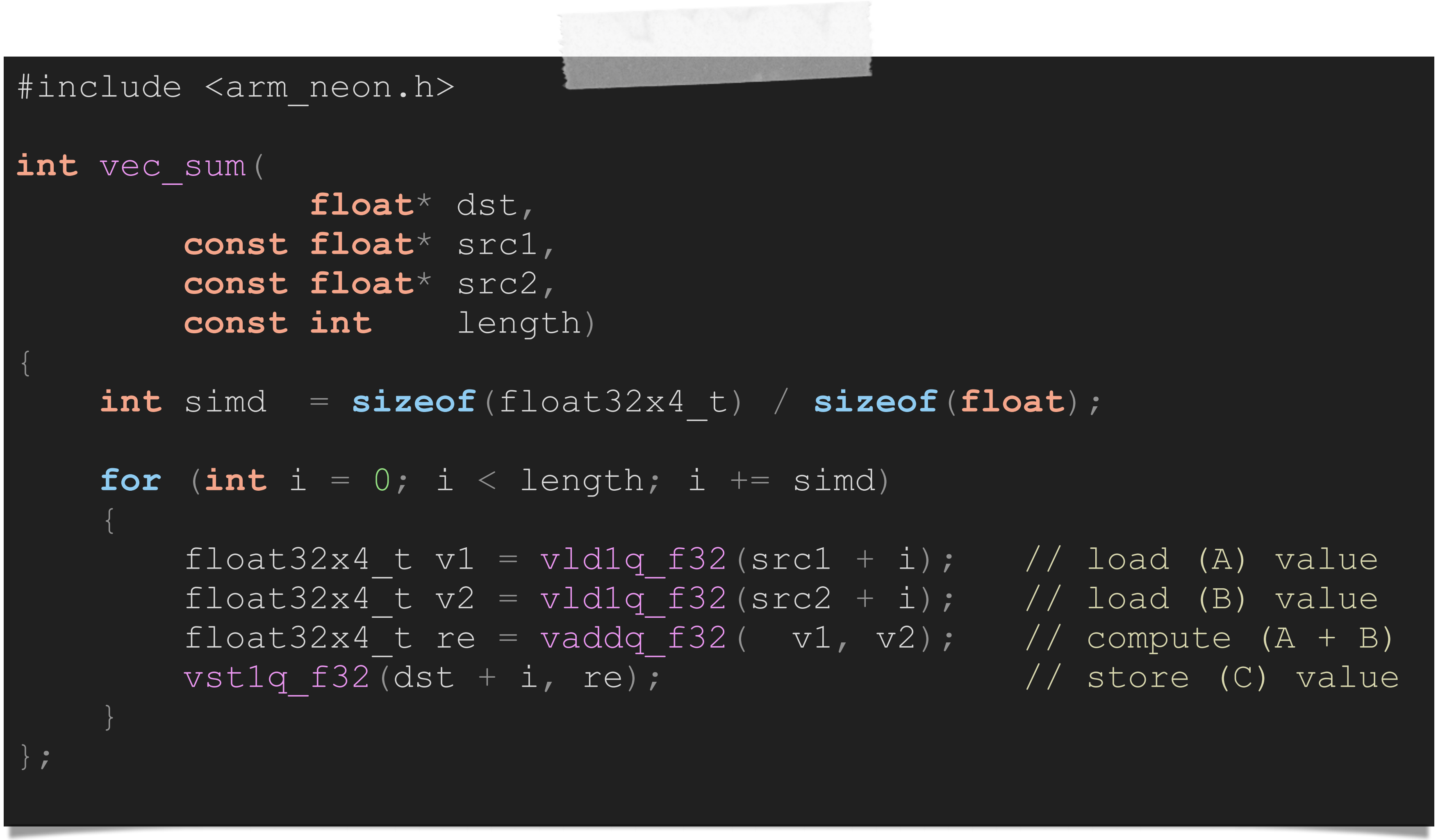


# Toy SIMD example – ARM Neon version



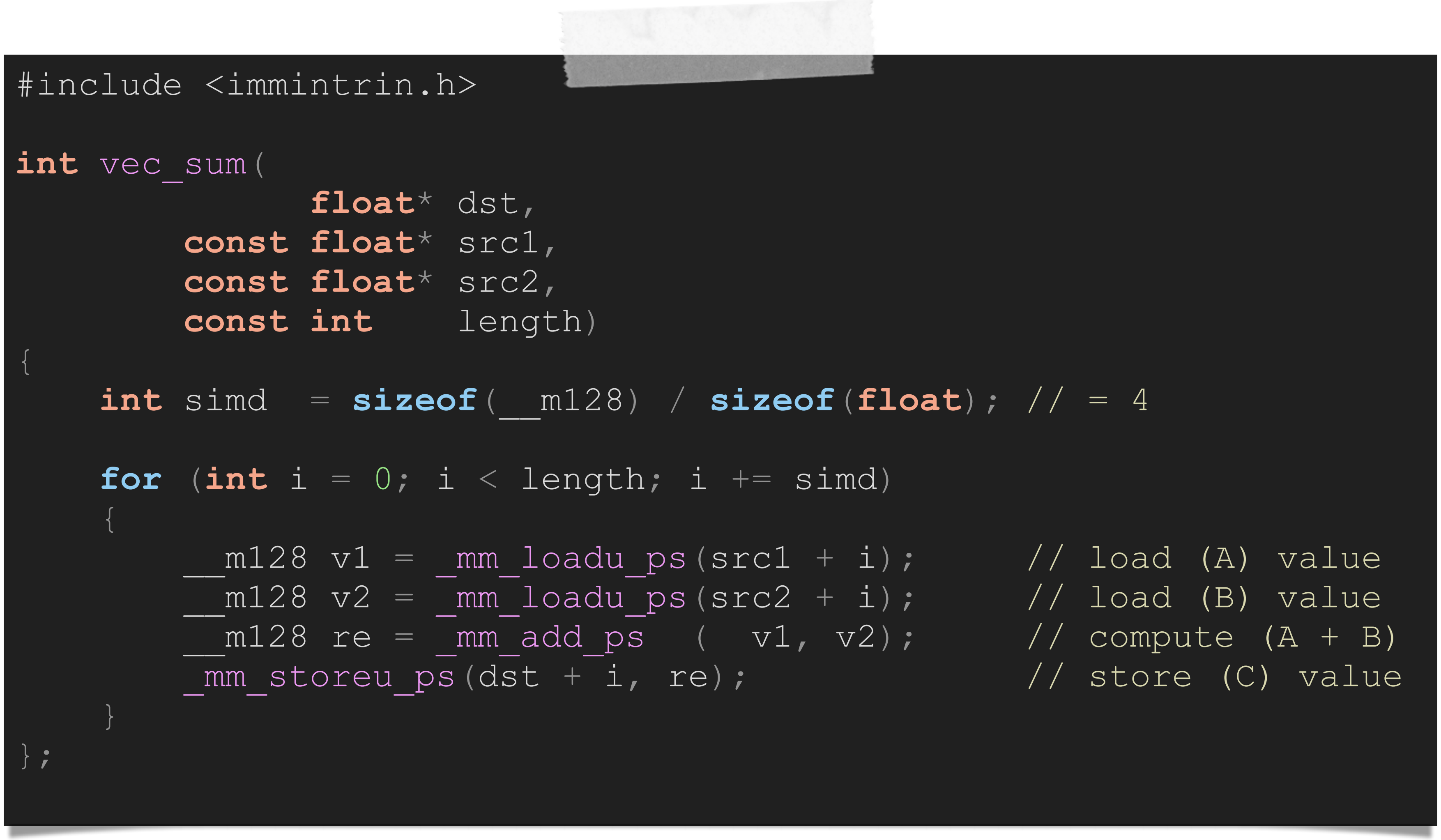
```c
#include <arm_neon.h>

int vec_sum(
             float* dst,
       const float* src1,
       const float* src2,
       const int    length)
{

   for (int i = 0; i < length; i += simd)
   {
       float v1 = src1[ i ];   // load (A) value
       float v2 = src2[ i ];   // load (B) value
       float re = v1 + v2;     // compute (A + B)
       dst[ i ] = re;          // store (C) value
   }

};
```

# Toy SIMD example - ARM Neon version



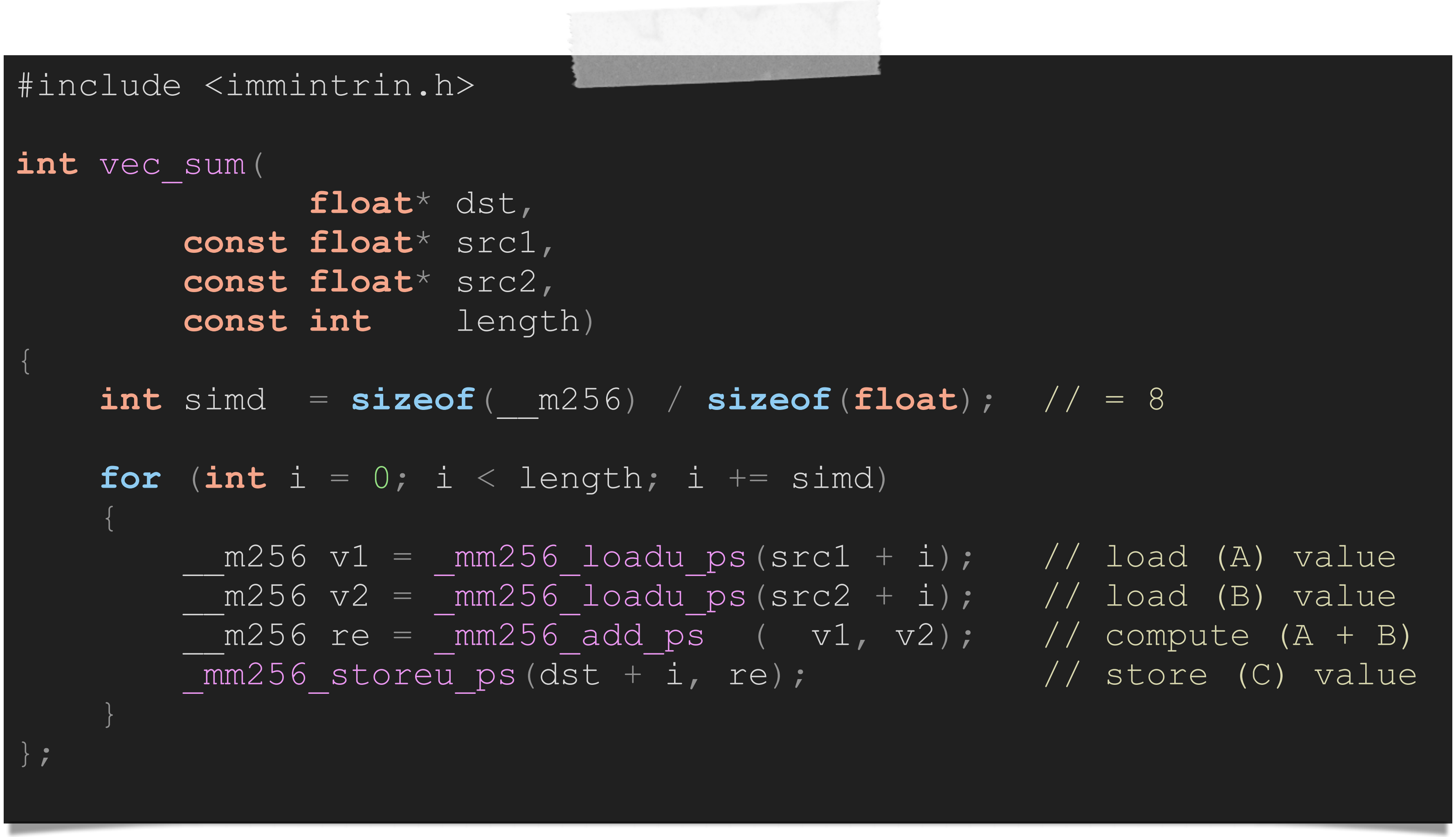
```
#include <arm_neon.h>

int vec_sum(
              float* dst,
        const float* src1,
        const float* src2,
        const int    length)
{
    int simd  = sizeof(float32x4_t) / sizeof(float);

    for (int i = 0; i < length; i += simd)
    {
        float32x4_t v1 = vld1q_f32(src1 + i);   // load (A) value
        float32x4_t v2 = vld1q_f32(src2 + i);   // load (B) value
        float32x4_t re = vaddq_f32(  v1, v2);   // compute (A + B)
        vst1q_f32(dst + i, re);                 // store (C) value
    }
};
```

# Toy SIMD example – INTEL SSE version



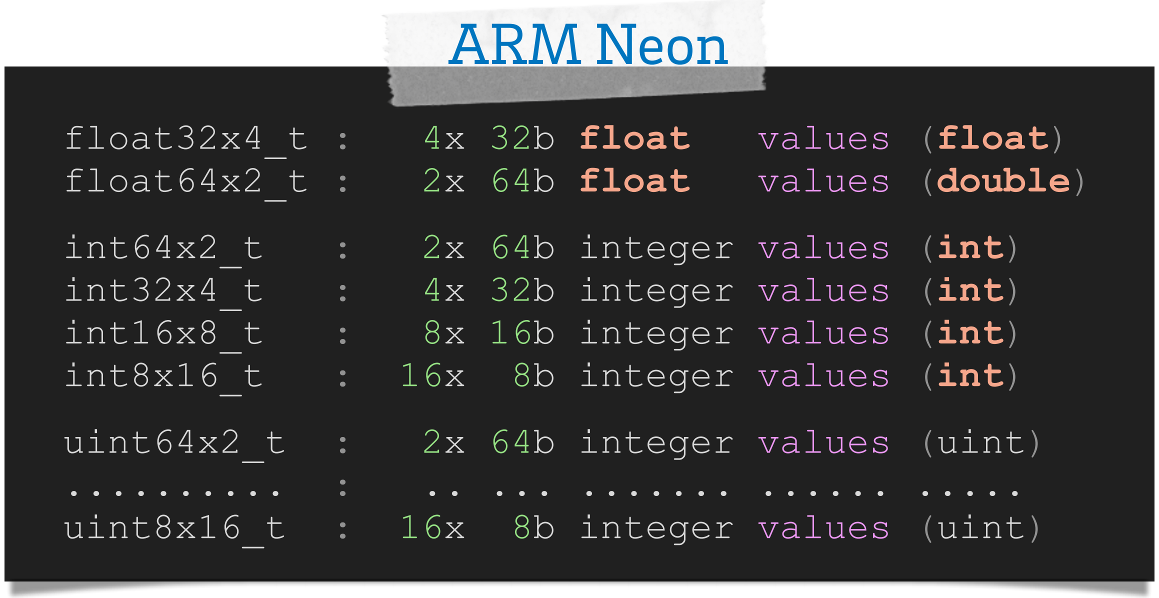
```c
#include <immintrin.h>

int vec_sum(
              float* dst,
        const float* src1,
        const float* src2,
        const int    length)
{
    int simd  = sizeof(__m128) / sizeof(float); // = 4

    for (int i = 0; i < length; i += simd)
    {
        __m128 v1 = _mm_loadu_ps(src1 + i);     // load (A) value
        __m128 v2 = _mm_loadu_ps(src2 + i);     // load (B) value
        __m128 re = _mm_add_ps  (  v1, v2);     // compute (A + B)
        _mm_storeu_ps(dst + i, re);             // store (C) value
    }
};
```

# Toy SIMD example – INTEL AVX version



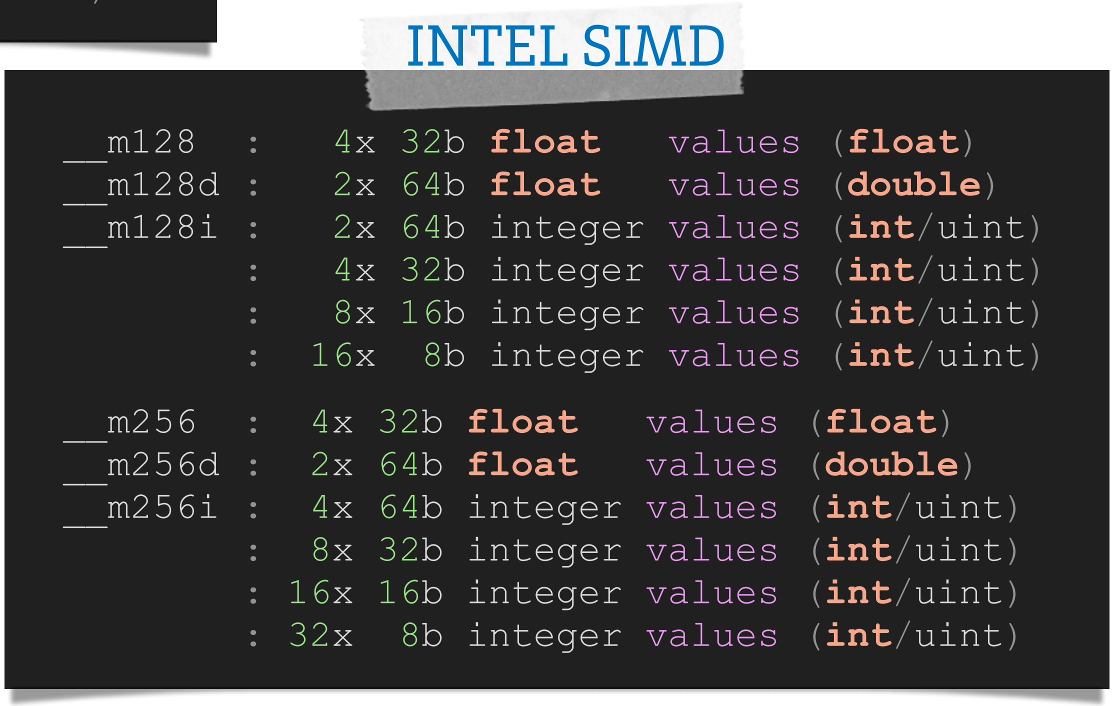
```
#include <immintrin.h>

int vec_sum(
              float* dst,
        const float* src1,
        const float* src2,
        const int    length)
{
    int simd  = sizeof(__m256) / sizeof(float);  // = 8

    for (int i = 0; i < length; i += simd)
    {
        __m256 v1 = _mm256_loadu_ps(src1 + i);   // load (A) value
        __m256 v2 = _mm256_loadu_ps(src2 + i);   // load (B) value
        __m256 re = _mm256_add_ps  (  v1, v2);   // compute (A + B)
        _mm256_storeu_ps(dst + i, re);           // store (C) value
    }
};
```

# SIMD data types available in INTEL/ARM



ARM Neon

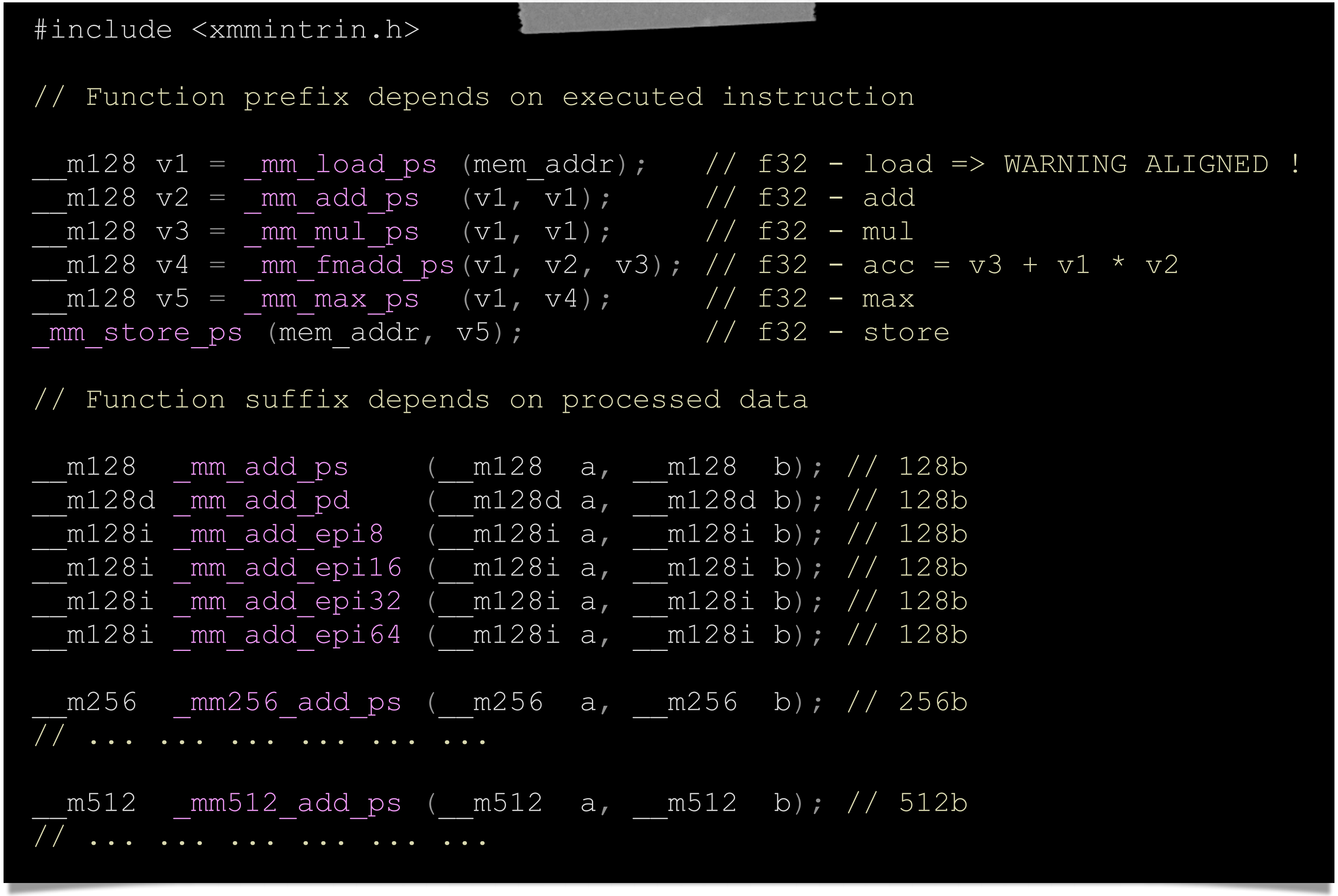
```
float32x4_t :   4x 32b float   values (float)
float64x2_t :   2x 64b float   values (double)

int64x2_t   :   2x 64b integer values (int)
int32x4_t   :   4x 32b integer values (int)
int16x8_t   :   8x 16b integer values (int)
int8x16_t   :  16x  8b integer values (int)

uint64x2_t  :   2x 64b integer values (uint)
..........  :   .. ... ....... ...... .....
uint8x16_t  :  16x  8b integer values (uint)
```



INTEL SIMD

```
__m128  :   4x 32b float   values (float)
__m128d :   2x 64b float   values (double)
__m128i :   2x 64b integer values (int/uint)
        :   4x 32b integer values (int/uint)
        :   8x 16b integer values (int/uint)
        :  16x  8b integer values (int/uint)

__m256  :  4x 32b float   values (float)
__m256d :  2x 64b float   values (double)
__m256i :  4x 64b integer values (int/uint)
        :  8x 32b integer values (int/uint)
        : 16x 16b integer values (int/uint)
        : 32x  8b integer values (int/uint)
```

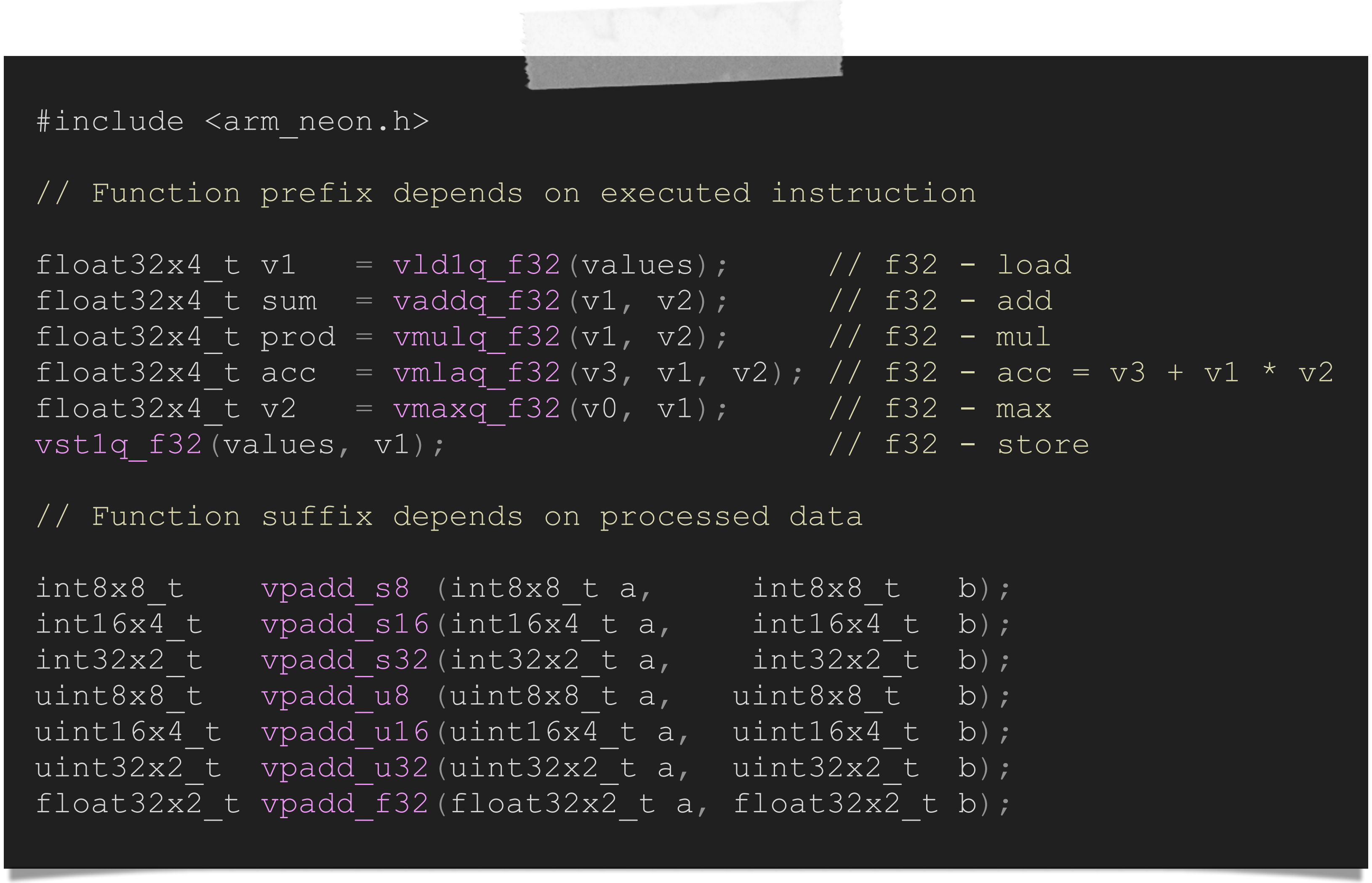
# The INTEL intrinsics for SSE/AVX targets



```c
#include <xmmintrin.h>

// Function prefix depends on executed instruction

__m128 v1 = _mm_load_ps (mem_addr);   // f32 - load => WARNING ALIGNED !
__m128 v2 = _mm_add_ps  (v1, v1);     // f32 - add
__m128 v3 = _mm_mul_ps  (v1, v1);     // f32 - mul
__m128 v4 = _mm_fmadd_ps(v1, v2, v3); // f32 - acc = v3 + v1 * v2
__m128 v5 = _mm_max_ps  (v1, v4);     // f32 - max
_mm_store_ps (mem_addr, v5);          // f32 - store

// Function suffix depends on processed data

__m128  _mm_add_ps    (__m128  a, __m128  b); // 128b
__m128d _mm_add_pd    (__m128d a, __m128d b); // 128b
__m128i _mm_add_epi8  (__m128i a, __m128i b); // 128b
__m128i _mm_add_epi16 (__m128i a, __m128i b); // 128b
__m128i _mm_add_epi32 (__m128i a, __m128i b); // 128b
__m128i _mm_add_epi64 (__m128i a, __m128i b); // 128b

__m256  _mm256_add_ps (__m256  a, __m256  b); // 256b
// ... ... ... ... ... ...

__m512  _mm512_add_ps (__m512  a, __m512  b); // 512b
// ... ... ... ... ... ...
```

https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

# The ARM intrinsics for NEON targets

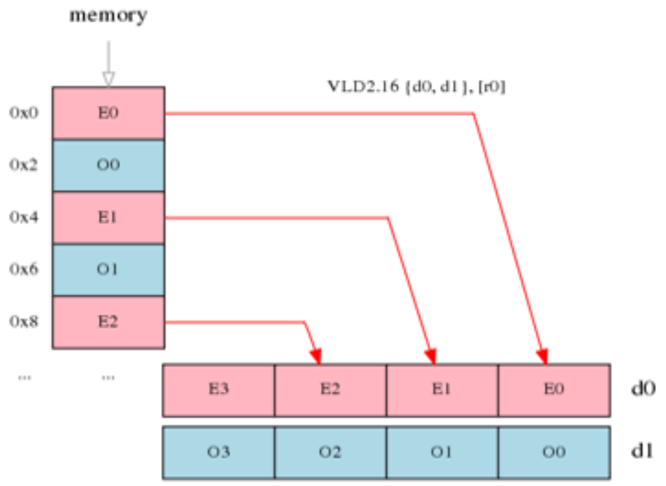


```
#include <arm_neon.h>

// Function prefix depends on executed instruction

float32x4_t v1   = vld1q_f32(values);     // f32 - load
float32x4_t sum  = vaddq_f32(v1, v2);     // f32 - add
float32x4_t prod = vmulq_f32(v1, v2);     // f32 - mul
float32x4_t acc  = vmlaq_f32(v3, v1, v2); // f32 - acc = v3 + v1 * v2
float32x4_t v2   = vmaxq_f32(v0, v1);     // f32 - max
vst1q_f32(values, v1);                    // f32 - store

// Function suffix depends on processed data

int8x8_t    vpadd_s8 (int8x8_t a,     int8x8_t   b);
int16x4_t   vpadd_s16(int16x4_t a,    int16x4_t  b);
int32x2_t   vpadd_s32(int32x2_t a,    int32x2_t  b);
uint8x8_t   vpadd_u8 (uint8x8_t a,   uint8x8_t   b);
uint16x4_t  vpadd_u16(uint16x4_t a,  uint16x4_t  b);
uint32x2_t  vpadd_u32(uint32x2_t a,  uint32x2_t  b);
float32x2_t vpadd_f32(float32x2_t a, float32x2_t b);
```

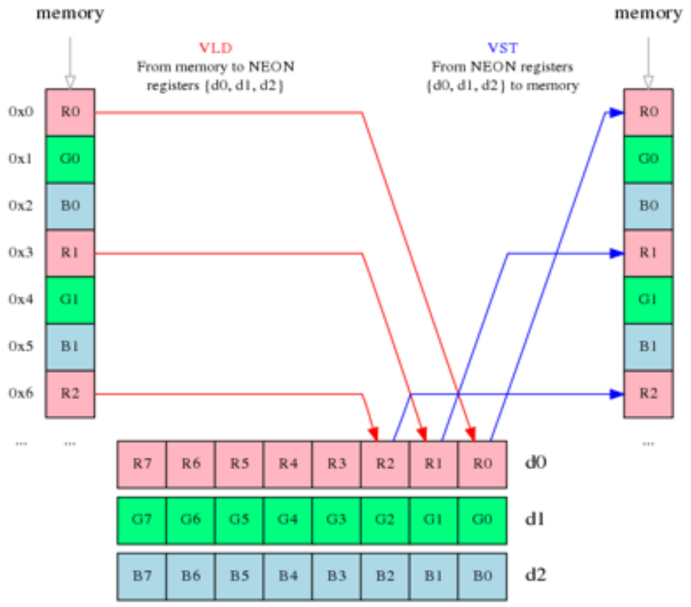
https://github.com/thenifty/neon-guide
https://developer.arm.com/documentation/dui0472/m/Using-NEON-Support

# Missing features in INTEL ISA

- INTEL processors
  - aligned or unaligned two instr.
  - contiguous memory load/store
- ARM processors
  - aligned/unaligned one intr.
  - contiguous memory load/store
  - interleaved load/store
- On intel architecture such feature should be hand implemented



Loading and deinterleaving 16-bit data.

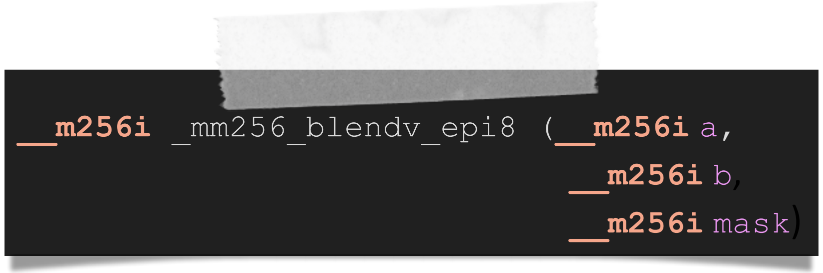


NEON structure loads and stores.

https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/coding-for-neon---part-1-load-and-stores

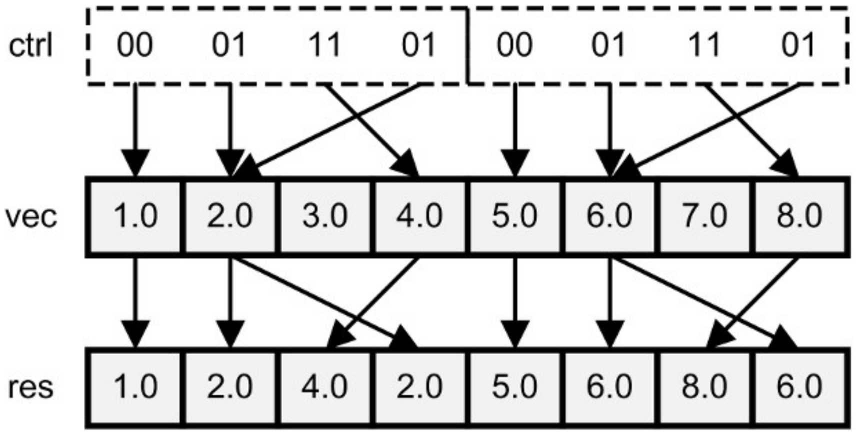
# Missing features in ARM NEON

- Complex algorithms & memory structures needs at runtime
  - Data permutation,
  - Data duplication, etc,
  - Conditional moves.
- Two intrinsic families
  - _mm*_shuffle_ps
  - _mm*_permute_ps
  - Easy to use with one lane, otherwise…

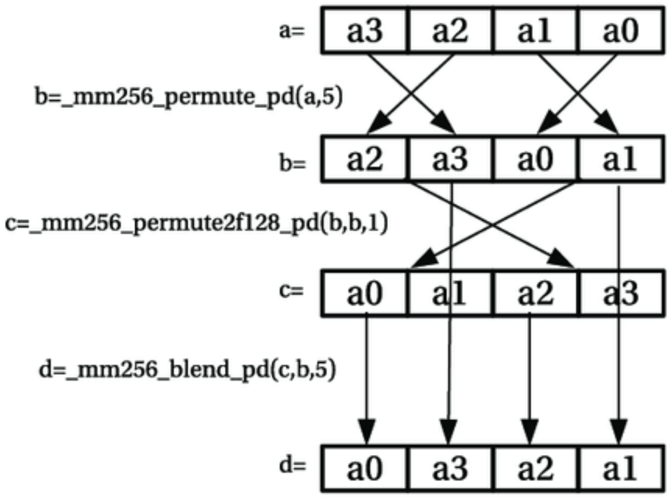


```
__m256i _mm256_blendv_epi8 (__m256i a,
                             __m256i b,
                             __m256i mask)
```

```
res = _mm256_permute_ps(vec, 0b01110100)
```



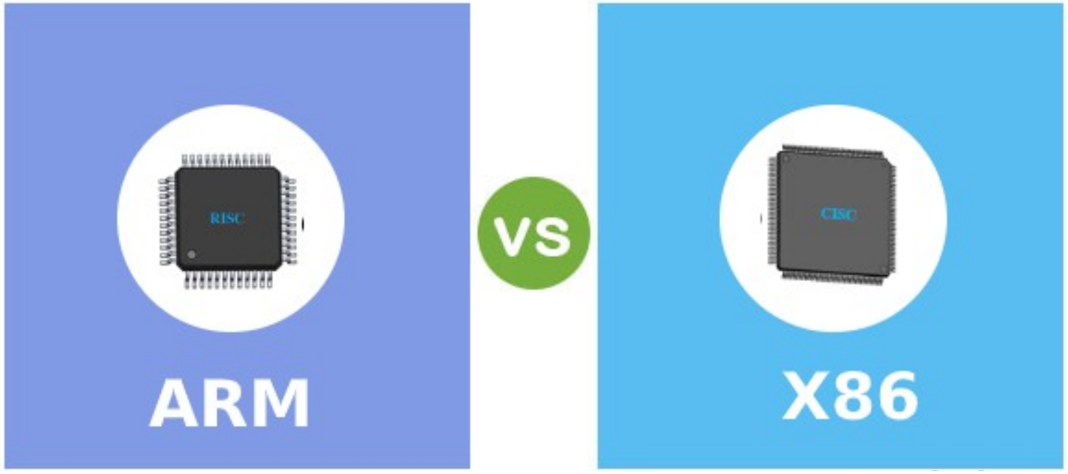
Cyclic double vector rotate



https://www.cs.cmu.edu/afs/cs/academic/class/15213-s19/www/lectures613/04-simd.pdf

# How to compile codes with SIMD intrinsics

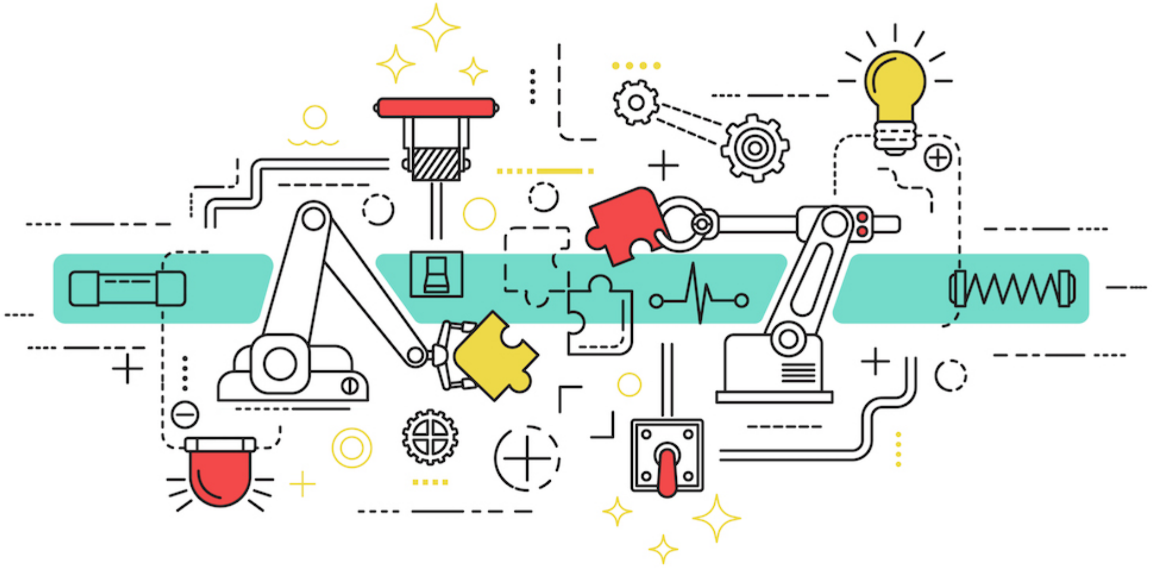
- Depends on the targeted CPU or toolchain
  - g++ -o main main.cpp -O3 *-mtune=native -march=native* # INTEL CPUs
  - g++ -o main main.cpp -O3 *-mtune=native -mcpu=native* # ARM CPUs
- The « native » architecture defines some preprocessor directives
  - __SSE4_2__ __AVX__ __AVX2__ __AVX512F__ # INTEL CPUs
  - __ARM_NEON__ __ARM_NEON # ARM CPUs



https://stackoverflow.com/questions/28939652/how-to-detect-sse-sse2-avx-avx2-avx-512-avx-128-fma-kcvi-availability-at-compile

# Compiler auto-vectorization features

- Compilers can do the job 4u
- Pro
  - Generic (int8_t … int32_t … float),
  - Generic (SSE4 … AVX2 … AVX512),
  - Generic (ARM & INTEL),
  - (Very) efficient thanks to cost models used by the compiler,
  - Compiler provides vectorization tips.
- Con
  - Limited to simple kernels,
  - Learn compiler code style,
  - Needs additional pragma or keywords.



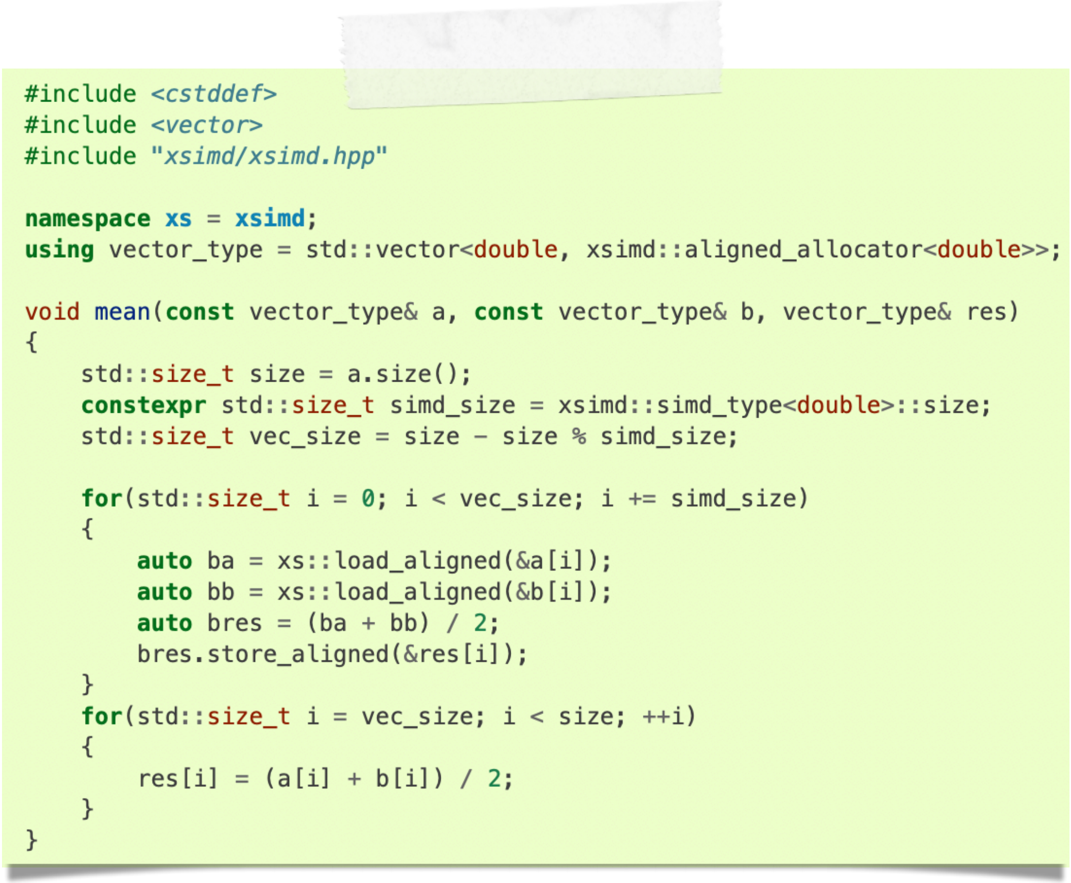
https://gcc.gnu.org/projects/tree-ssa/vectorization.html

https://llvm.org/docs/Vectorizers.html

https://blog.minhazav.dev/guide-compiler-to-auto-vectorise/

# Managing « efficiently » various SIMD ISA

- Writing SIMD codes
  - long and painful,
  - adapted to target features,
- C++ wrappers for SIMD intrinsics
  - XSIMD
  - VectorClass
  - MIPPS
  - nova-simd and …
- Easy software codes
  - Portability & flexibility
  - What is really executed ?



```cpp
#include <cstddef>
#include <vector>
#include "xsimd/xsimd.hpp"

namespace xs = xsimd;
using vector_type = std::vector<double, xsimd::aligned_allocator<double>>;

void mean(const vector_type& a, const vector_type& b, vector_type& res)
{
    std::size_t size = a.size();
    constexpr std::size_t simd_size = xsimd::simd_type<double>::size;
    std::size_t vec_size = size - size % simd_size;

    for(std::size_t i = 0; i < vec_size; i += simd_size)
    {
        auto ba = xs::load_aligned(&a[i]);
        auto bb = xs::load_aligned(&b[i]);
        auto bres = (ba + bb) / 2;
        bres.store_aligned(&res[i]);
    }
    for(std::size_t i = vec_size; i < size; ++i)
    {
        res[i] = (a[i] + b[i]) / 2;
    }
}
```