CS302

SIMD and Vector

Spring 2025

Arkaprava Basu & Babak Falsafi

parsa.epfl.ch/course-info/cs302

[A relic CRAY-1 at EPFL]

Adapted from slides originally developed by Prof. Falsafi

Where are We?

M	T	W	T	F
17-Feb	18-Feb	19-Feb	20-Feb	21-Feb
24-Feb	25-Feb	26-Feb	27-Feb	28-Feb
3-Mar	4-Mar		6-Mar	7-Mar
10-Mar	11-Mar	12-Mar	13-Mar	14-Mar
17-Mar	18-Mar	19-Mar	20-Mar	21-Mar
24-Mar	25-Mar	26-Mar	27-Mar	28-Mar
31-Mar	1-Apr	2-Apr	3-Apr	4-Apr
7-Apr	8-Apr	9-Apr	10-Apr	11-Apr
14-Apr	15-Apr	16-Apr	17-Apr	18-Apr
21-Apr	22-Apr	23-Apr	24-Apr	25-Apr
28-Apr	29-Apr	30-Apr	1-May	2-May
5-May	6-May	7-May	8-May	9-May
12-May	13-May	14-May	15-May	16-May
19-May	20-May	21-May	22-May	23-May
26-May	27-May	28-May	29-May	30-May

- SIMD & Vector
  - SIMD execution model
  - Examples in modern processors
- Exercise session
  - Performance debugging
- Next Tuesday:
  - Message Passing

Taxonomy of Computer Architectures

- Michael J. Flynn – 1966/72

		Instruction Streams	
		Single	Multiple
Data Streams	Single	SISD (CPU)	MISD (Systolic Arrays)
	Multiple	SIMD (CPU SIMD/Vector Units, AI Accelerator SIMD/Vector Units, GPUs)	MIMD (Cellphone/laptop chips with CPU/GPU/AI, Multicores, Clusters)

SISD: Most Common Model

- Single processing element, instruction and data streams
  - Uniprocessors
    - You will have covered basic CPU architecture in your previous courses

- Warmup: is it possible to have *concurrency/parallelism* in a SISD processor?

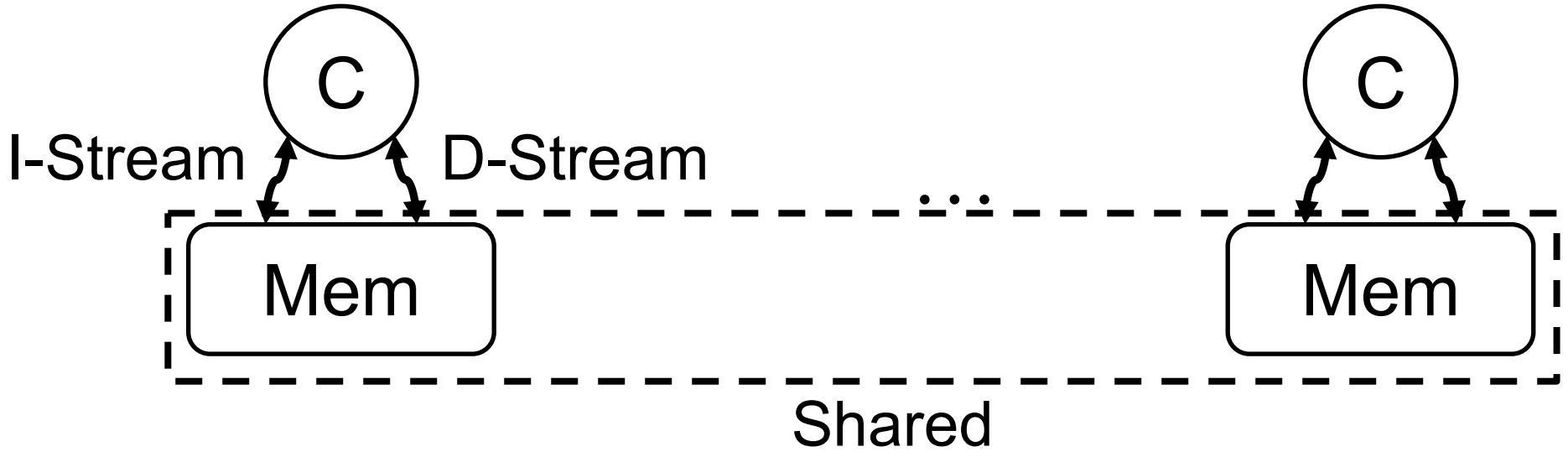
SISD: Most Common Model

- Single processing element, instruction and data streams
  - Uniprocessors
    - You will have covered basic CPU architecture in your previous courses
- Warmup: is it possible to have *concurrency/parallelism* in a SISD processor?
  - Yes!
  - Pipelined, superscalar or out-of-order processors

MIMD: Parallel but Independent

- Multiple processing elements, instruction and data streams
  - Multiprocessors

- Previous lecture introduced shared memory programming
  - Prog. model is a way to specify the instruction & data streams



SIMD: Data Parallelism

- Multiple processing units & data streams, **single** instruction stream
  - Motivation – reduce fetching and writing back operands from/to memory
  - Typical scientific computing - iterate over millions-trillions of elements
  - First commercial machines → *Vector* processors in 1960/70s
    - ILLIAC IV (’66), Cray-1/XMP48 (‘77)

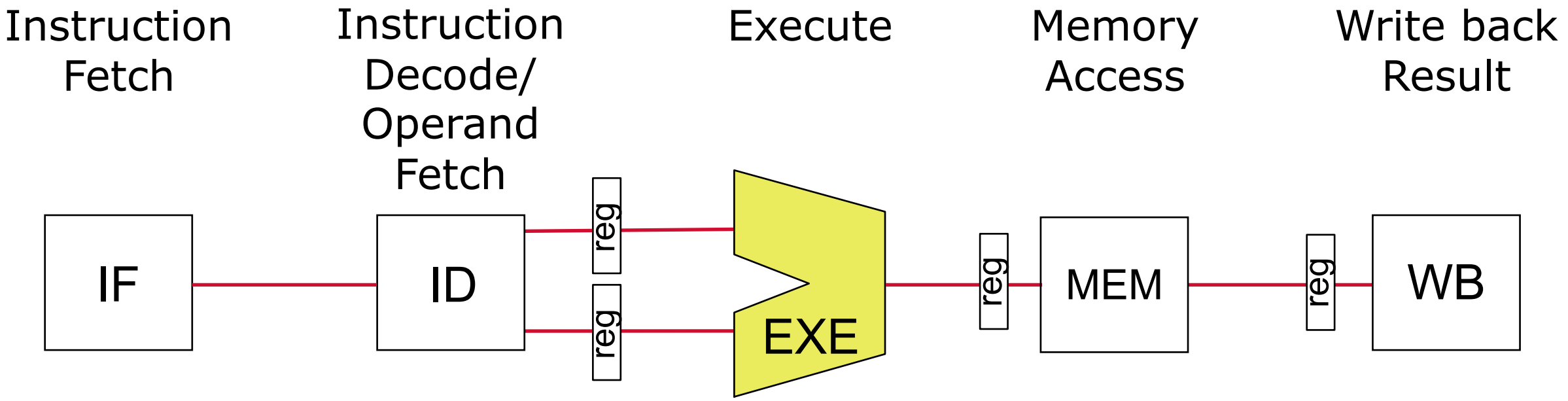
A relic CRAY-1 at EPFL!

Why SIMD?

- Many workloads: simple arithmetic on huge, regular datasets
  - E.g., systems of linear equations (LINPACK/LAPACK for supercomputers)
  - They reduce to loops similar to the following:
    - Double-Precision A.X + Y called DAXPY

```
void daxpy(int n, double a, double *x, double *y)
{
     for(int i = 0; i < n; i++) {
          y[i] = a * x[i] + y[i];
     }
}
```

Recall: Basic Scalar Pipeline



- Instructions fetched, decoded
- Operands from register file, execution in ALUs (arithmetic logic unit)
- Memory access in data cache
- Write results back to register file

Scalar Code Hits von Neumann Bottleneck

- For every loop iteration:
  - Load x[i] and y[i] into registers
  - Calculate a*x[i] + y[i]
  - Write back register into y[i]
- How many instructions for a DAXPY of size 1024?
- How many serialized cache misses? (w/ 64B cache lines)

(Assume a word and a register is 64 bits)

```
; x[] -> r2, y[] -> r3
; a -> r4,
; &x[n] -> r5

loop:
  lw    r1, 0(r2)   ; load x[i]
  lw    r6, 0(r3)   ; load y[i]
  mul   r1, r1, r4  ; a*x[i]
  add   r6, r1, r6  ; … + y[i]
  sw    r6, 0(r3)   ; store y[i]
  add   r2, r2, 8
  add   r3, r3, 8
  bne   r2, r5, loop
```

Scalar Code Hits von Neumann Bottleneck

- For every loop iteration:
  - Load x[i] and y[i] into registers
  - Calculate a*x[i] + y[i]
  - Write back register into y[i]
- How many instructions for a DAXPY of size 1024?
  - 8 * 1024 = 8192
- How many serialized cache misses? (w/ 64B cache lines)

(Assume a word and a register is 64 bits)

```
; x[] -> r2, y[] -> r3
; a -> r4,
; &x[n] -> r5

loop:
  lw    r1, 0(r2)   ; load x[i]
  lw    r6, 0(r3)   ; load y[i]
  mul   r1, r1, r4  ; a*x[i]
  add   r6, r1, r6  ; … + y[i]
  sw    r6, 0(r3)   ; store y[i]
  add   r2, r2, 8
  add   r3, r3, 8
  bne   r2, r5, loop
```

Scalar Code Hits von Neumann Bottleneck

- For every loop iteration:
  - Load x[i] and y[i] into registers
  - Calculate a*x[i] + y[i]
  - Write back register into y[i]
- How many instructions for a DAXPY of size 1024?
- How many serialized cache misses? (w/ 64B cache lines)
  - Size of double = 8B
  - 64B, 8 elements per block, 2 blocks → 2 * 1024 / 8 = 256

(Assume a word and a register is 64 bits)

```
; x[] -> r2, y[] -> r3
; a -> r4,
; &x[n] -> r5

loop:
  lw    r1, 0(r2)   ; load x[i]
  lw    r6, 0(r3)   ; load y[i]
  mul   r1, r1, r4  ; a*x[i]
  add   r6, r1, r6  ; … + y[i]
  sw    r6, 0(r3)   ; store y[i]
  add   r2, r2, 8
  add   r3, r3, 8
  bne   r2, r5, loop
```

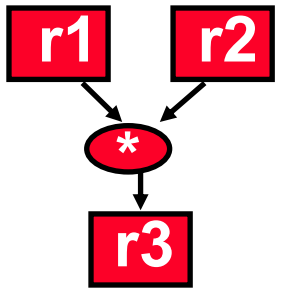
Hardware Limits ILP for Scalar Loops

- Each iteration of the DAXPY loop is completely independent
- Out-of-order (OoO) processors unroll loop and begin new iterations
  - DAXPY loop is memory bound
  - Rough estimate of re-order buffer (ROB) size is ~500
  - Loop contains 8192 instructions, only ~6% in ROB at a time
- ROB cannot hold all instructions even with a very small vector (1k)
- Silicon area & power scales quadratically for bigger OoO windows
  - Problem: our ISA is only letting us specify a single operation at a time
  - No need to duplicate control for such simple programs

Enter Vector Supercomputers (1970s)

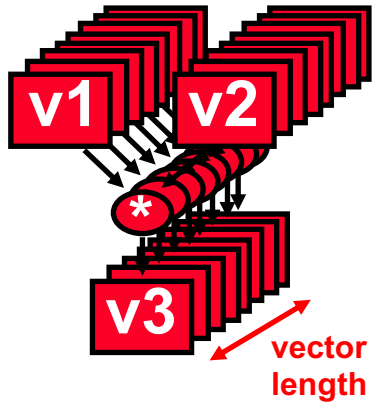
- First generation of commercial SIMD machines
- Vector ISAs define operations on arrays of numbers: "vectors"

SCALAR
(1 operation)



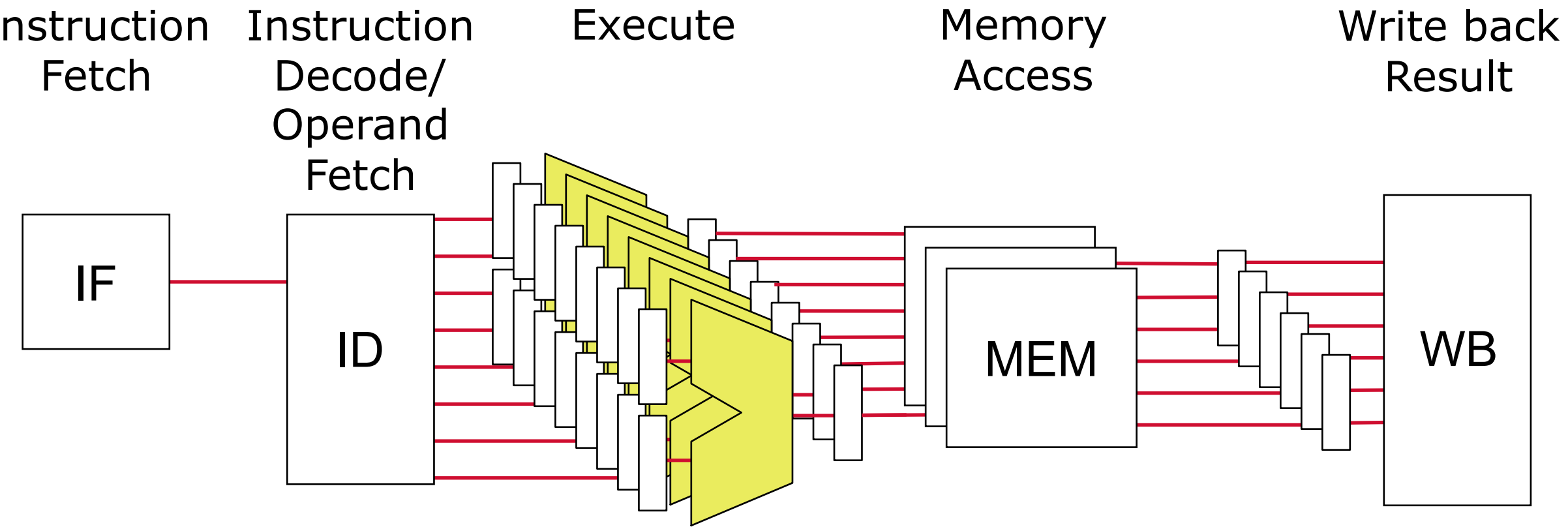
mul r3, r1, r2

VECTOR
(N operations)

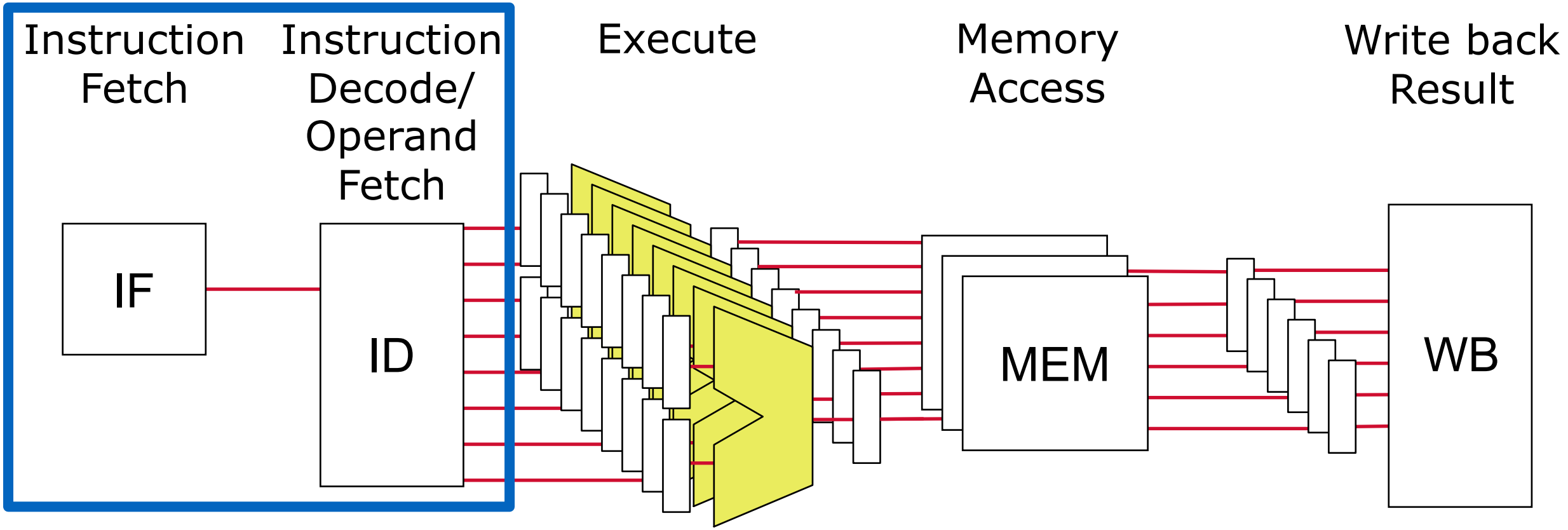


mul.v v3, v1, v2

Basic Vector Pipeline

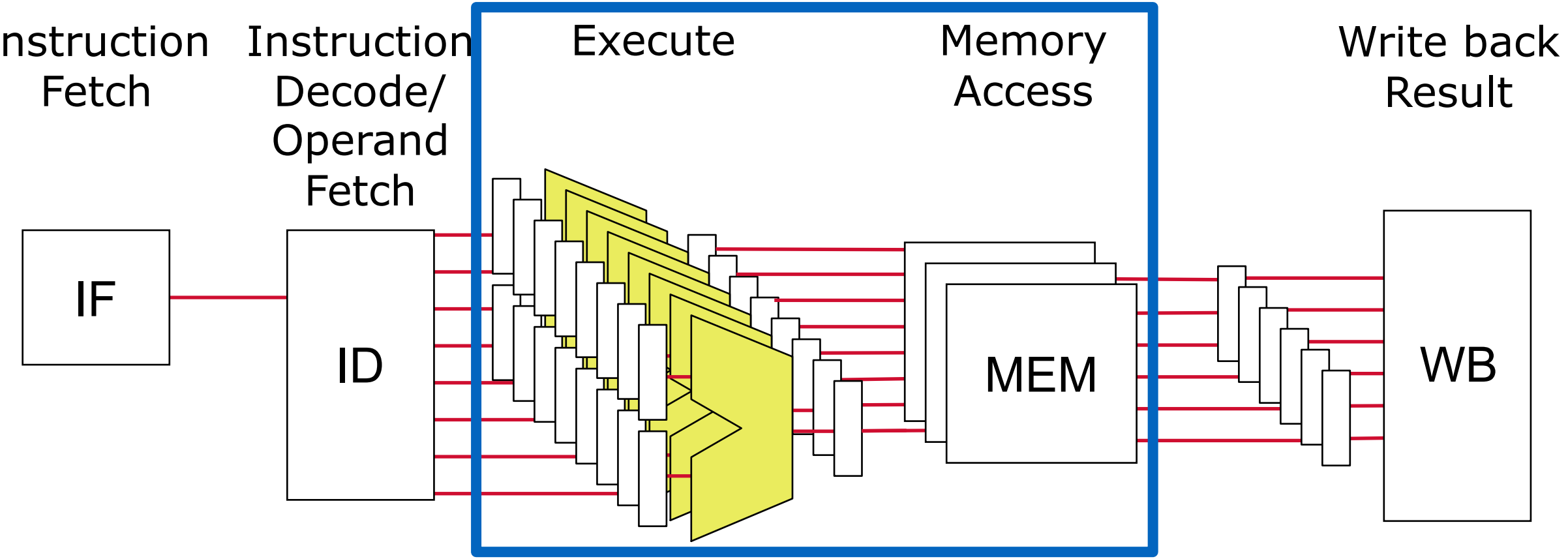


Basic Vector Pipeline



- One instruction, for an array of register operands
  - E.g., 8 vector registers each with 64x64-bit elements in Cray-1

Basic Vector Pipeline

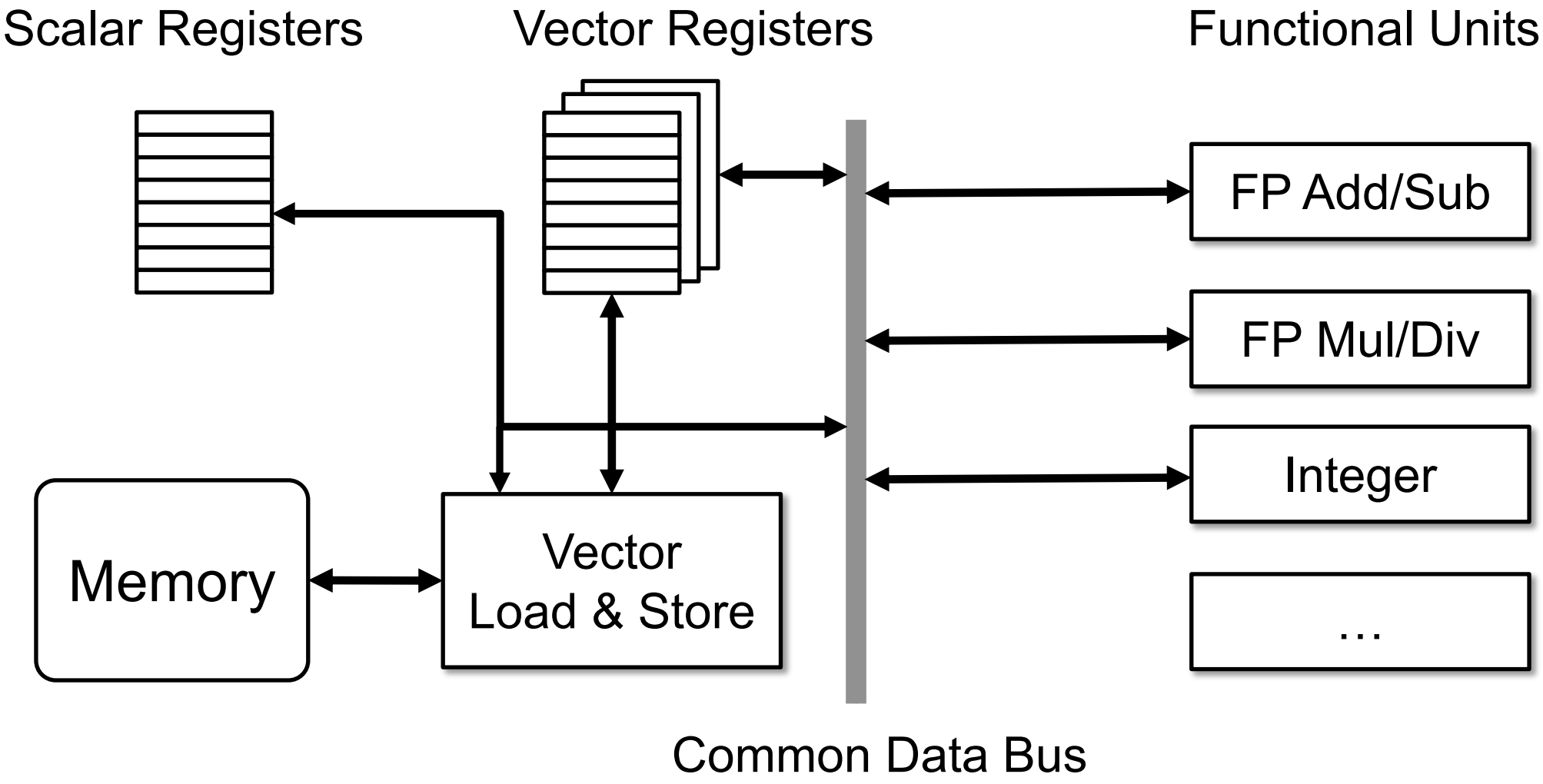


- Multiple execution units
- Wider memory to read/write multiple data

Vector Processors Address CPU Limitations

- Single instruction represents many computations
  - Greatly reduces instruction bloat
- Simplifies dependency checking
  - Only check data hazards for vector operands, not each sub-element
  - No more complex than scalar code
- Known access patterns potentially reduce cache misses
  - e.g., vector load of 64x 8B entries brings 8 cache blocks simultaneously
- Reduced control hazards due to fewer branches

A Basic Vector Processing Architecture



Vector Functional Unit Design

- Choice 1 - each functional unit is pipelined
  - Begins processing a new vector element every clock cycle
- Choice 2 - multiple *lanes* in a single functional unit
  - Multiple hardware units, each executing independent elements of the vector

- How is the design of vector register file & common buses affected?
  - Multi-lane design increases num. ports required
  - Pipelined designs consume 2 elements/cycle BW in total
  - Multi-lane designs multiply BW by the lane-width

Example Vector Instructions

Each vector register contains many scalar values

- e.g., a vector register V has 64 scalars

So,

mul.v	v1, v2, v1	Vector dot product v1*v2
mul.sv	v1, r1, v1	multiplies scalar r1 to all elements of v1
lw.v	v1, 0(r1)	loads vector v1 from address r1
sw.v	v1, 0(r1)	stores vector v1 at address r1

Vectorized DAXPY Loop

- Assumption:
  - Compute 64 elements at a time
- How many instructions for N = 1024?
- How many serialized cache misses? (w/ 64B cache lines)

```
; x[] -> r2, y[] -> r3
; a -> r4,
; &x[n] -> r5

loop:
  lw.v     v1, 0(r2)    ; load x[i]
  lw.v     v2, 0(r3)    ; load y[i]
  mul.sv   v1, r4, v1   ; a*x[i]
  add.v    v1, v1, v2   ; … + y[i]
  sw.v     v1, 0(r3)    ; store y[i]
  add      r2, r2, 512  ; 64*8
  add      r3, r3, 512
  bne      r2, r5, loop
```

Vectorized DAXPY Loop

- Assumption:
  - Compute 64 elements at a time
- How many instructions for N = 1024?
  - 8 * (1024 / 64) = 128
  - 64x reduction over scalar
- How many serialized cache misses? (w/ 64B cache lines)
  - Vector load brings 64 elements at a time, equivalent to 8 cache lines
  - 2 * (1024 / 64) = 32

```
; x[] -> r2, y[] -> r3
; a -> r4,
; &x[n] -> r5

loop:
  lw.v     v1, 0(r2)    ; load x[i]
  lw.v     v2, 0(r3)    ; load y[i]
  mul.sv   v1, r4, v1   ; a*x[i]
  add.v    v1, v1, v2   ; … + y[i]
  sw.v     v1, 0(r3)    ; store y[i]
  add      r2, r2, 512  ; 64*8
  add      r3, r3, 512
  bne      r2, r5, loop
```

Evolution of Vector Processors

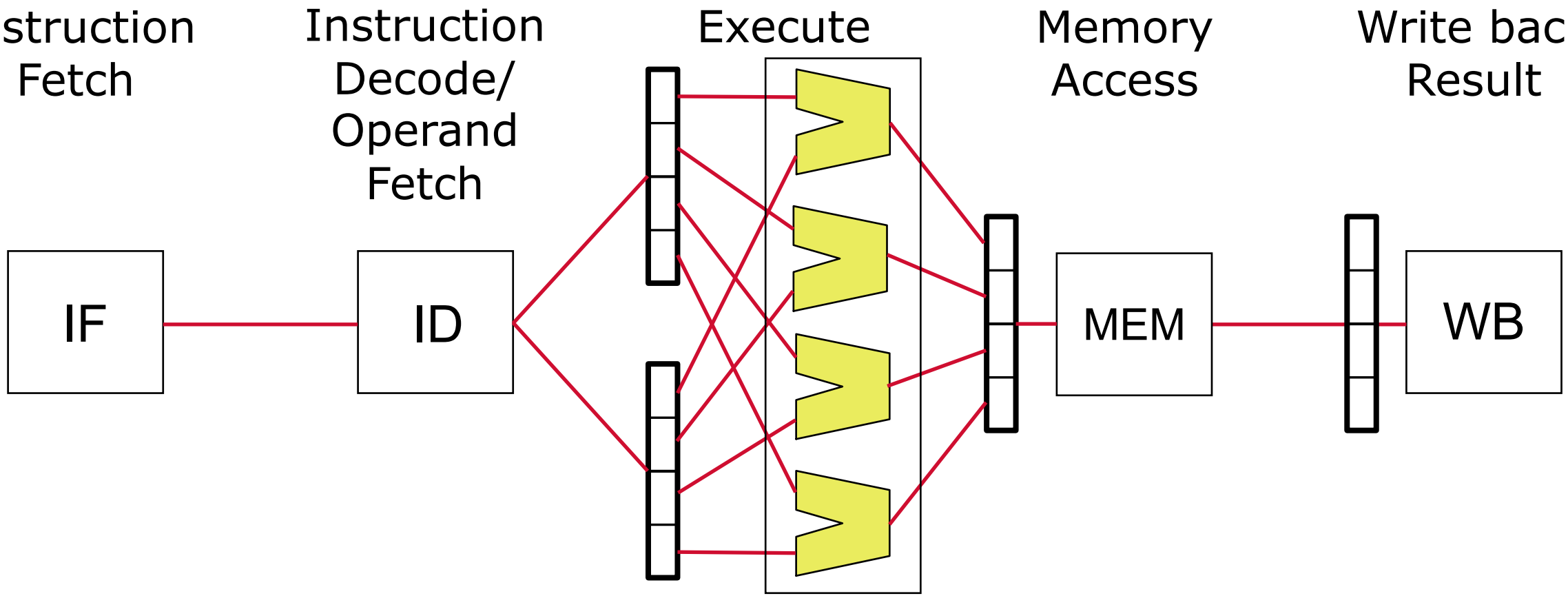
- Vector processors for supercomputers died in 1980s
  - Require a completely different architecture/hardware
  - Compilers can help vectorize code
  - But not all code is vectorizable
- Vector execution in CPUs
  - use SIMD principles with regular registers
    - E.g., a 64-bit wide scalar register becomes a 16-elem. 4-bit vector register
  - Historical perspective: ”SIMD within a register” proposed in 1950s
    - Intel MMX extensions released 1997
- Custom Vector processors on chip
  - Tensor cores in GPUs (covered later)
  - AI accelerator units in CPUs (e.g., Intel Sapphire Rapid)

CPU Vector vs. SIMD Extensions?

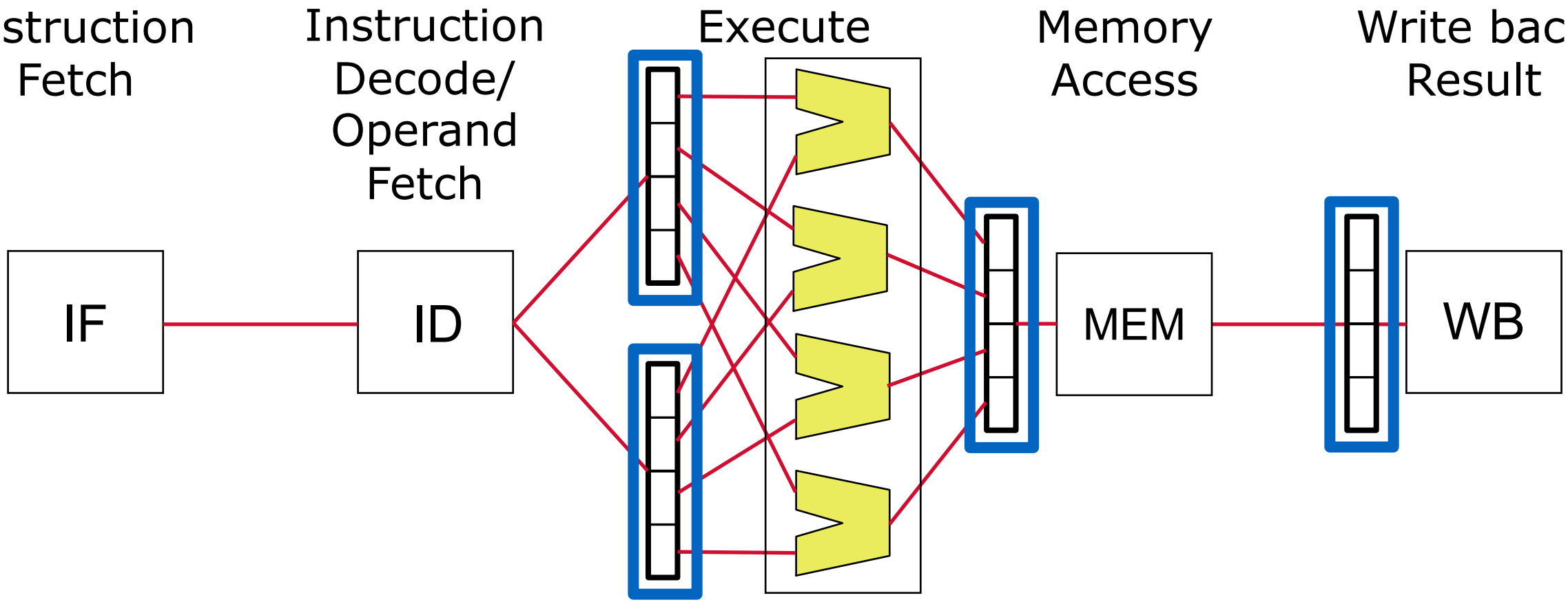
- Vector and SIMD extensions in CPU ISAs
  - ”Vector” and “SIMD” as words are used interchangeably
  - Marketing/branding (similar to CPU vs. GPU cores) except for RISC-V

- Idea: use SIMD principles with regular registers
  - E.g., a 64-bit wide scalar register becomes a 16-elem. 4-bit vector register
    - Limited changes required to existing processors
    - Allow vector computation on these short-width vectors
  - Regular registers referred to as “vector” registers

Basic SIMD Pipeline: ”Poor man’s” Vector processor

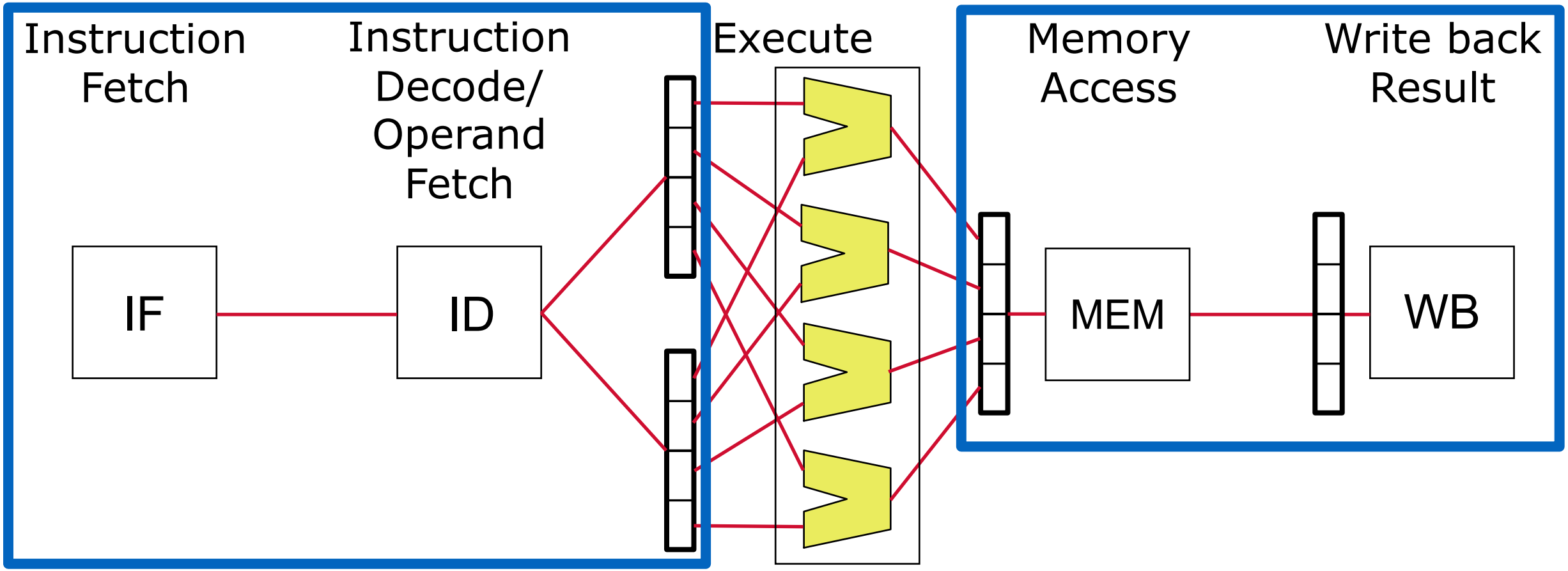


Basic SIMD Pipeline: ”Poor man’s” Vector processor



- Use existing register operands, divide them into smaller operands
  - E.g., one 64-bit registers → four 16-bit registers

Basic SIMD Pipeline: ”Poor man’s” Vector processor



- Use existing register file, memory, data path
  - Execution units are “bit-sliced”: designed to operate on smaller operands

SIMD in Modern CPUs

x86:

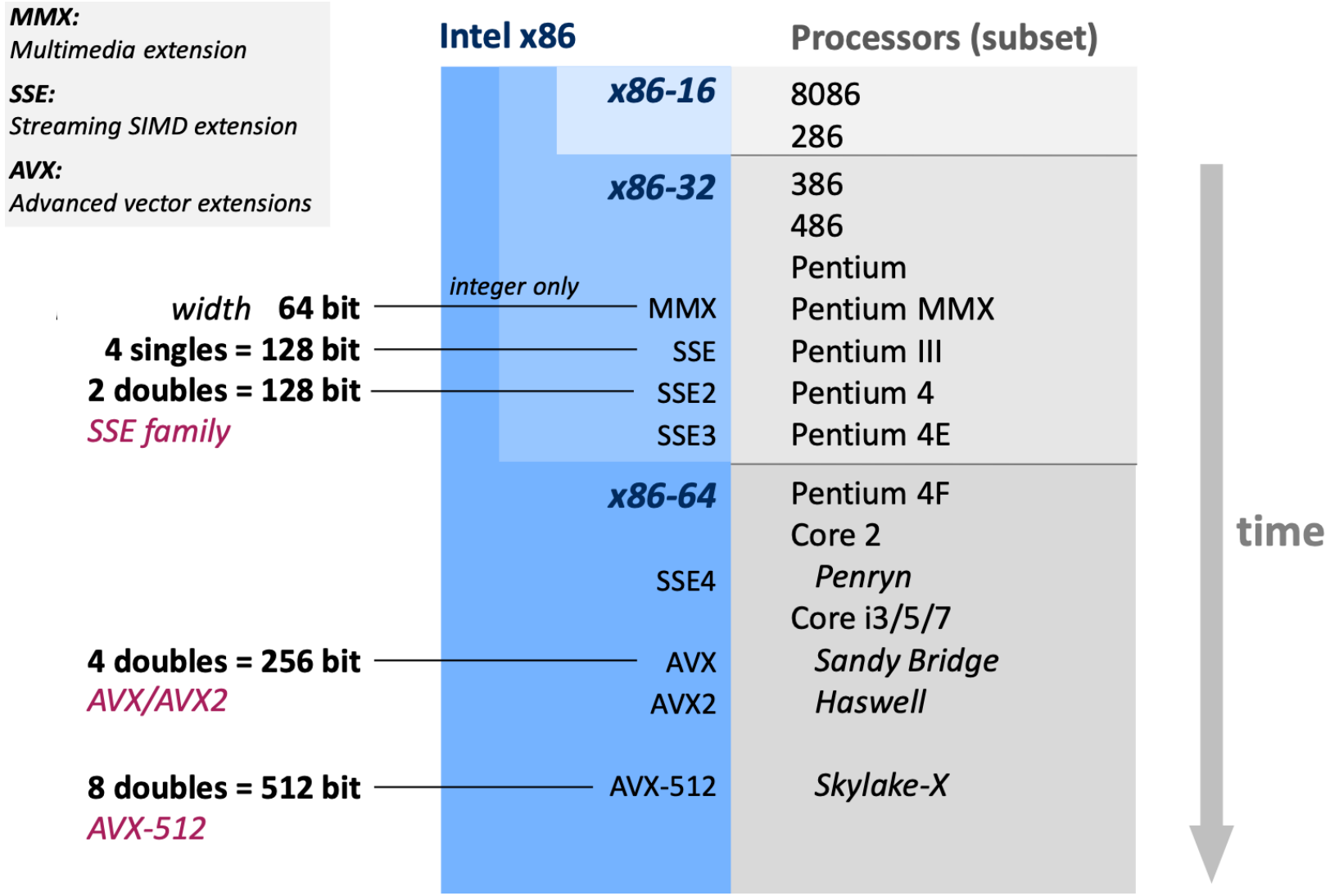
- SSE2 instructions: 128-bit operations (2x64, 4x32, 8x16, 16x8)
- AVX instructions: 256-bit operations (8x32 or 4x64 bits)
- AVX512 instructions: 512-bit operations (16x32 or 8x64 bits)

ARM: Neon instructions: 64-bit operations (2x32, 4x16, 8x8)

RISC-V: Vector extensions: implementation specific

- LMUL * VLEN bit operations, each element SEW bits
- VLEN is a hardware parameter and LMUL and SEW can be configured by software
- Designed this way to prevent ISA bloat

History of x86 SIMD Extensions



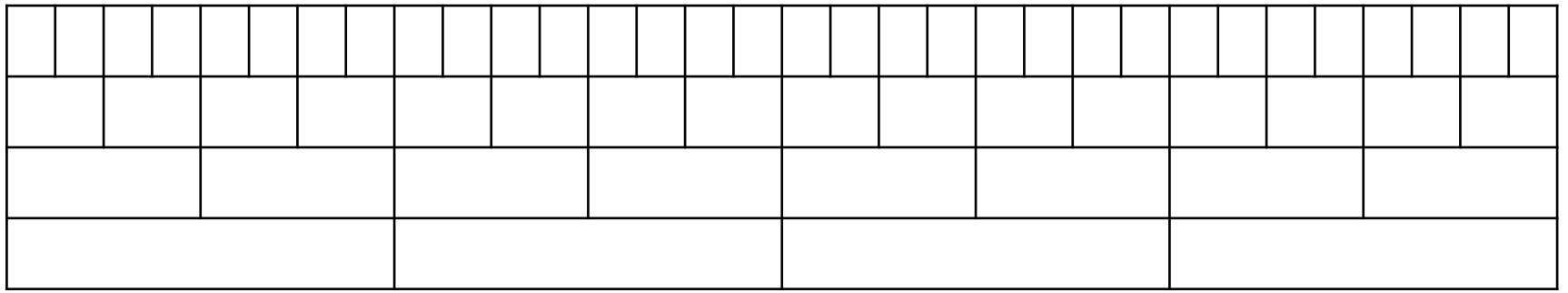
[Credit: Markus Püschel, ETHZ]

What do we need from the hardware?

- Vector Registers
  - To store multiple elements of a vector in a single register
- Replicated functional units inside the pipeline of a core
  - To operate on the elements of a vector register in parallel
- Support for decoding new instructions
- Memory system changes to load and store variable bitwidth elements

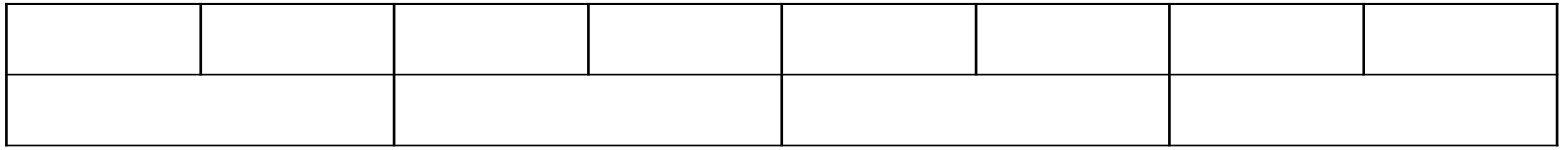
AVX Registers: 16 256-bit registers

Integer vectors



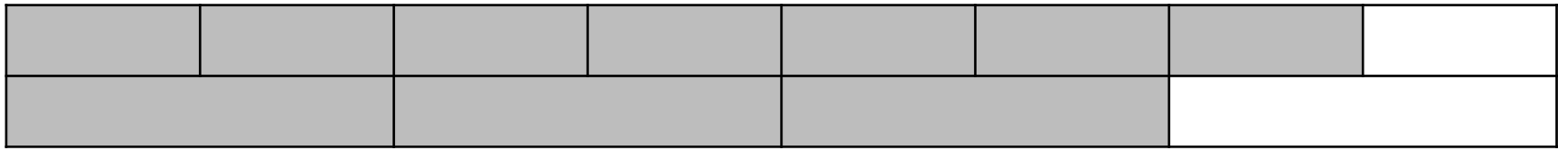
32-way byte
16-way 2 bytes
8-way 4 bytes
4-way 8 bytes

Floating-point vectors



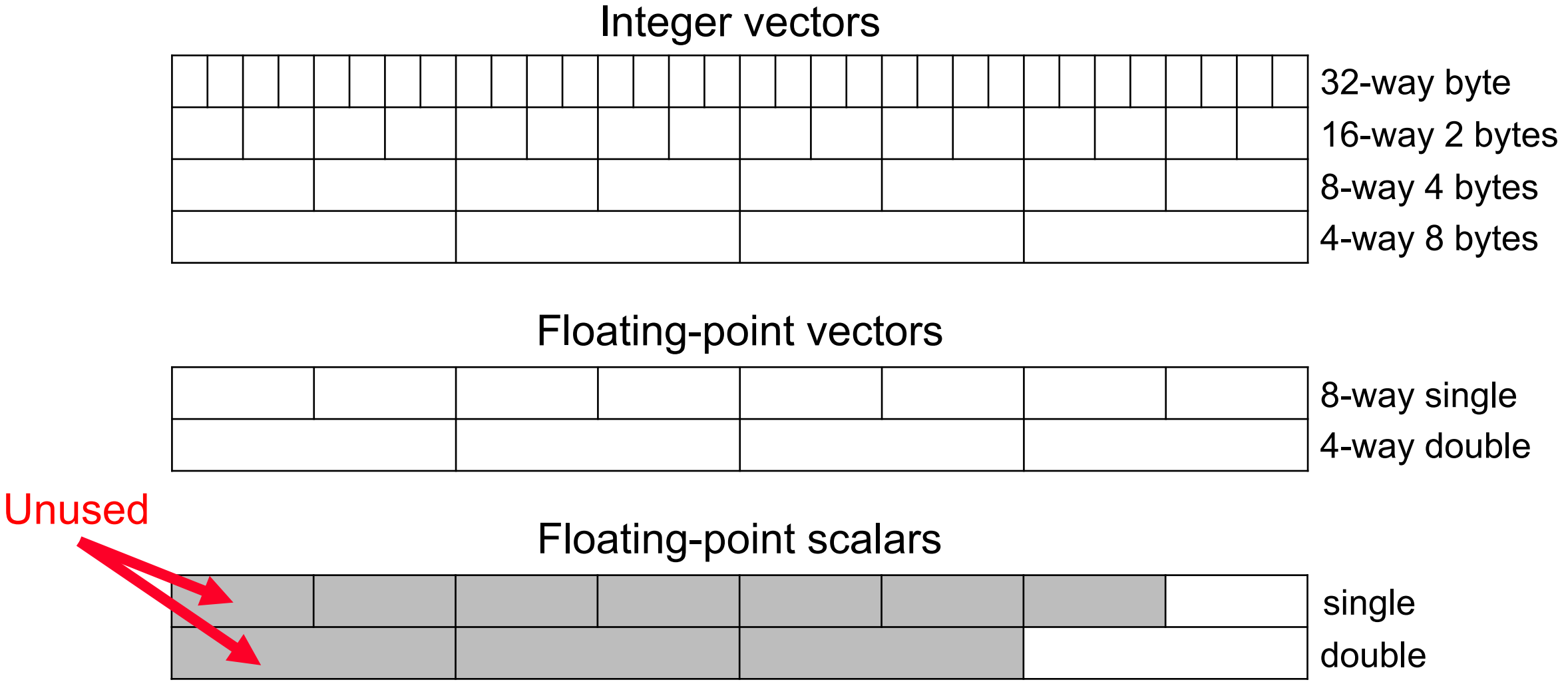
8-way single
4-way double

Floating-point scalars



single
double

AVX Registers: 16 256-bit registers

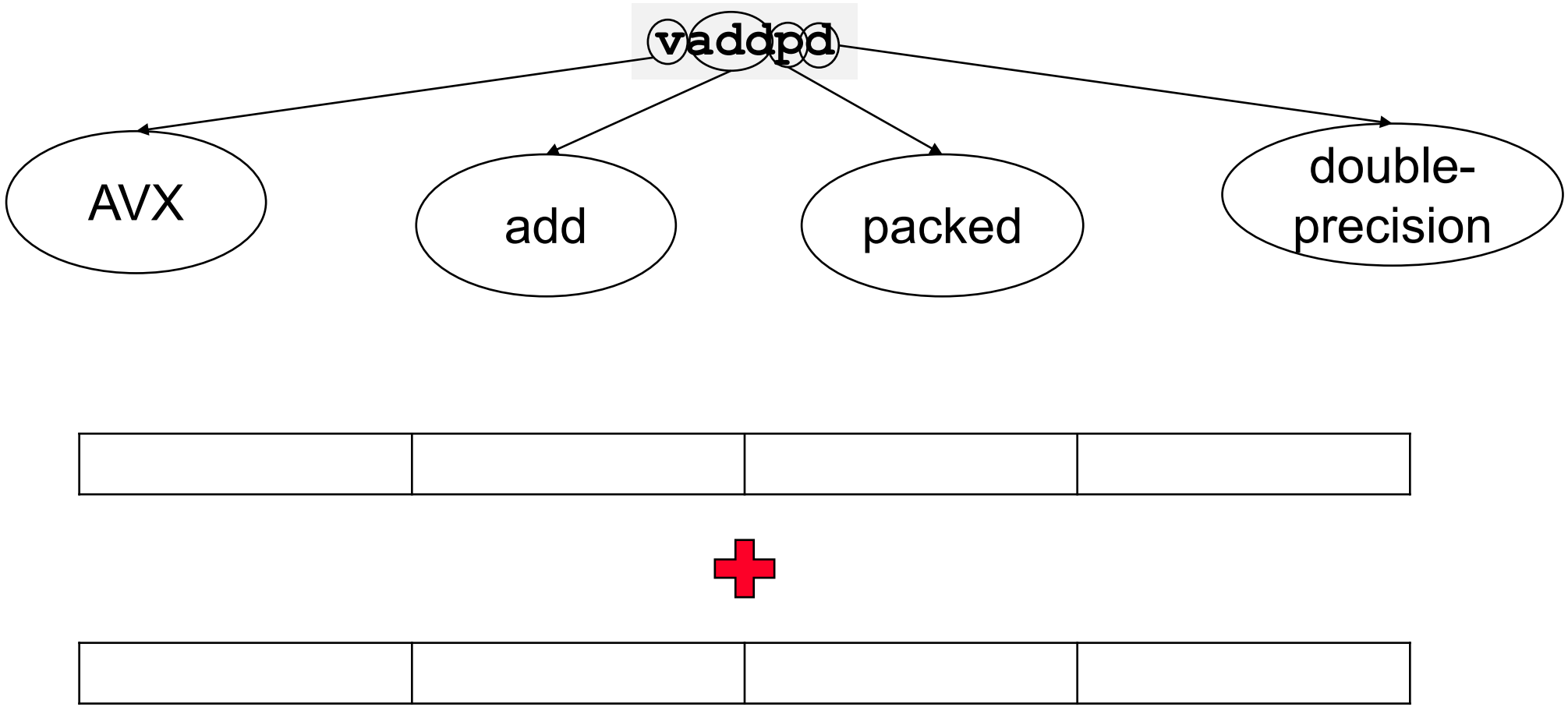


AVX Instructions

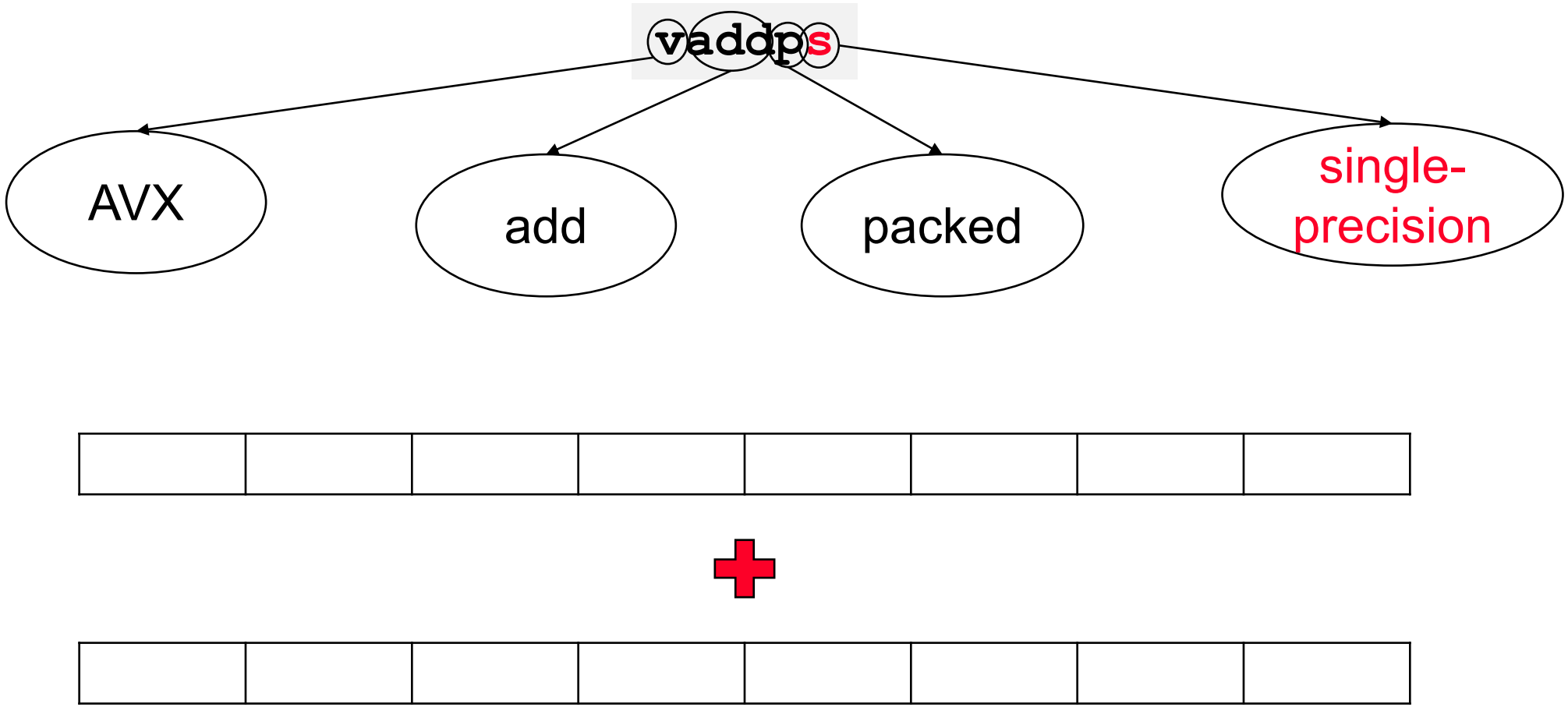
- AVX instructions operate on the vector registers
- Instructions define how the hardware interprets register contents

- Six categories of instructions:
  - Arithmetic
  - Logical/Bitwise
  - Data Movement
  - Shuffle/Permute
  - Blend/Mask
  - Scatter/Gather

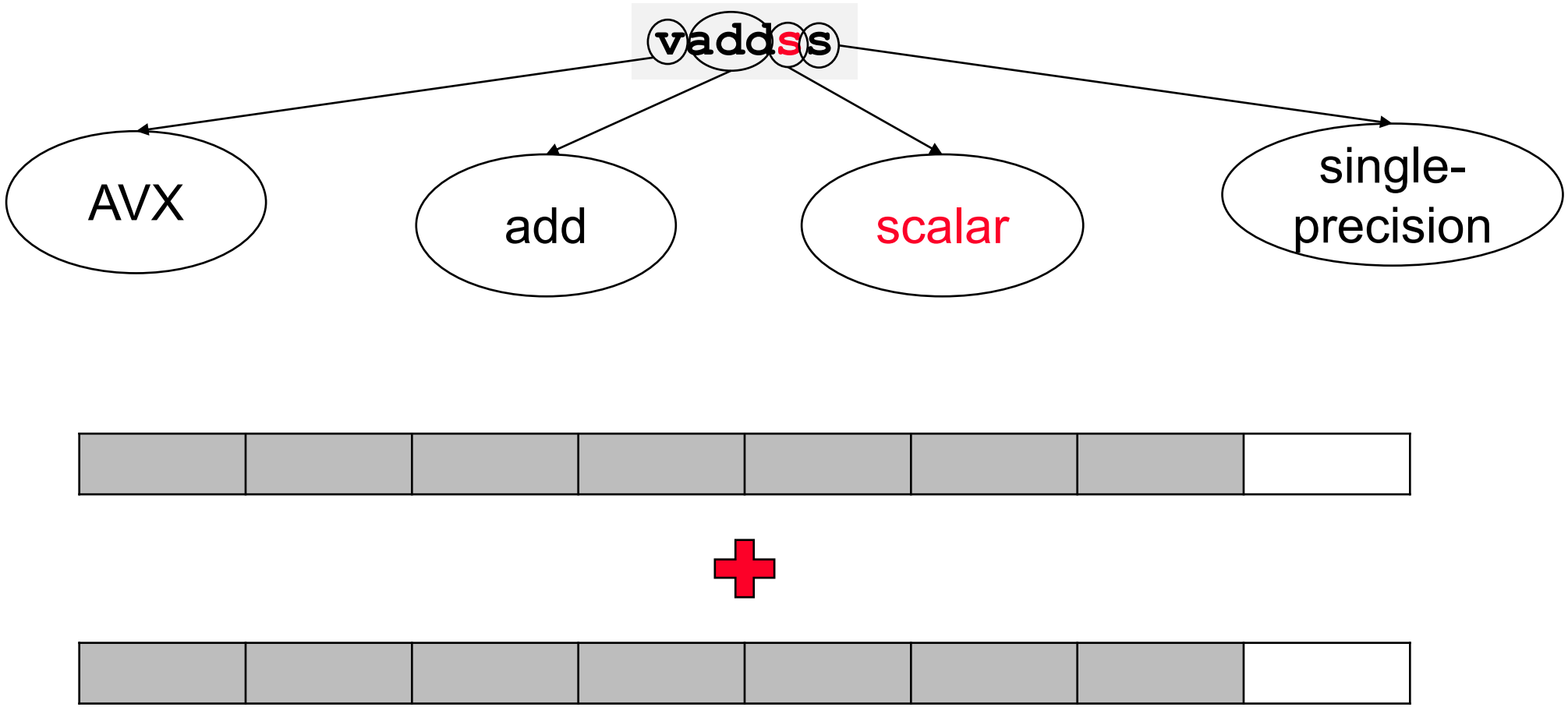
Interpreting AVX Instructions



Interpreting AVX Instructions



Interpreting AVX Instructions



How to use AVX instructions?

- Use libraries with support for AVX instructions
  - E.g., Intel MKL, OpenBLAS, etc
- Auto-compile code with AVX support
  - E.g., using -O3 when compiling C code
- Manually use intrinsics in code
  - Function-like macros that directly map to AVX instructions
- Write assembly code ☺

Auto-Compilation Challenge 1: Aliasing

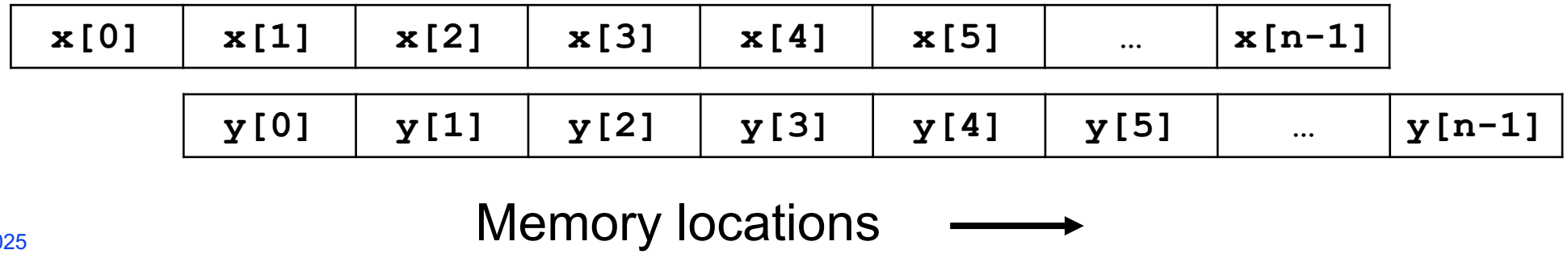
```
void daxpy(int n, double a, double *x, double *y)
{
     for(int i = 0; i < n; i++) {
          y[i] = a * x[i] + y[i];
     }
}
```

- To allow easy detection and optimization by the compiler, loop iterations need to be independent of each other
- Does the compiler know if pointers **`x`** and **`y`** point to different memory locations?

Auto-Compilation Challenge 1: Aliasing

```
void daxpy(int n, double a, double *x, double *y)
{
     for(int i = 0; i < n; i++) {
          y[i] = a * x[i] + y[i];
     }
}
```

- Any overlap between pointers **x** and **y** can result in the loop iterations not being independent of each other.
- For example:



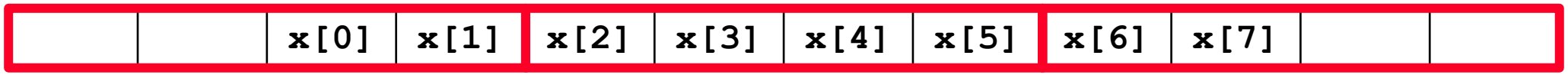
Solutions to Overcome Aliasing

- Solution 1: Use compiler flags
  - Flags like `-fstrict-aliasing` reinforce the C99 aliasing rules
- Solution 2: Declare pointers with the `restrict` keyword
  - Tells the compiler that these pointers do not overlap in memory
- Programmer's responsibility to ensure no aliasing can happen

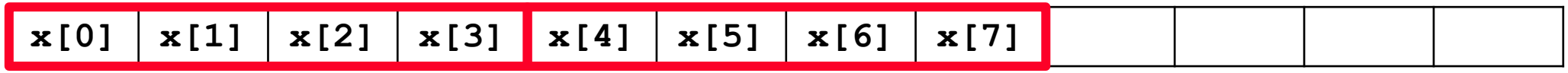
```
void daxpy(int n, double a, double *restrict x, double *restrict y)
{
      for(int i = 0; i < n; i++) {
            y[i] = a * x[i] + y[i];
      }
}
```

Auto-Compilation Challenge 2: Alignment

- Modern DRAM accesses are per 32-bit words (four bytes)
- For best performance, accesses to vector elements should be aligned
- For example, to load an 8-element 32-bit vector:
  - Case 1: Need three loads to load an misaligned vector



  - Case 2: Need two loads to load an aligned vector



- This is a performance issue that cannot be optimized by compiler

Solutions to Overcome Misalignment

- In C use **`aligned_alloc`** when allocating arrays
- Use **`__attribute__((aligned(32)))`** for static arrays

- If arrays are not aligned, compiler will try to “peel” the loop:

```
double *x = (double *) malloc(n * sizeof(double));
peel = (unsigned long) x & 0x1f;          /* x mod 32 */
if (peel != 0) {
    peel = (32 - peel)/sizeof(double);
    for (int i = 0; i < peel; i++) {      /* initial segment */
        <code>
    }
}
for (int i = peel; i < n; i++) {          /* 32-byte aligned access */
        <code>
}
```

Miscellaneous Issues to Take Care Of

- Loops with number of iterations known at runtime
  - Compiler may be able to generate code to deal with corner cases
- No if-else statements or diverging conditions inside loops
  - However, can be cleverly implemented using bitmasks
- Loop iterations should be independent of each other
- Avoid function calls from within loop body
  - Compiler may skip vectorization depending on the function
- Ideally, use unit stride in innermost loop

AVX Intrinsics

- In some cases, auto-compilation with AVX support not enough
- Developers must manually optimize code with “intrinsics”

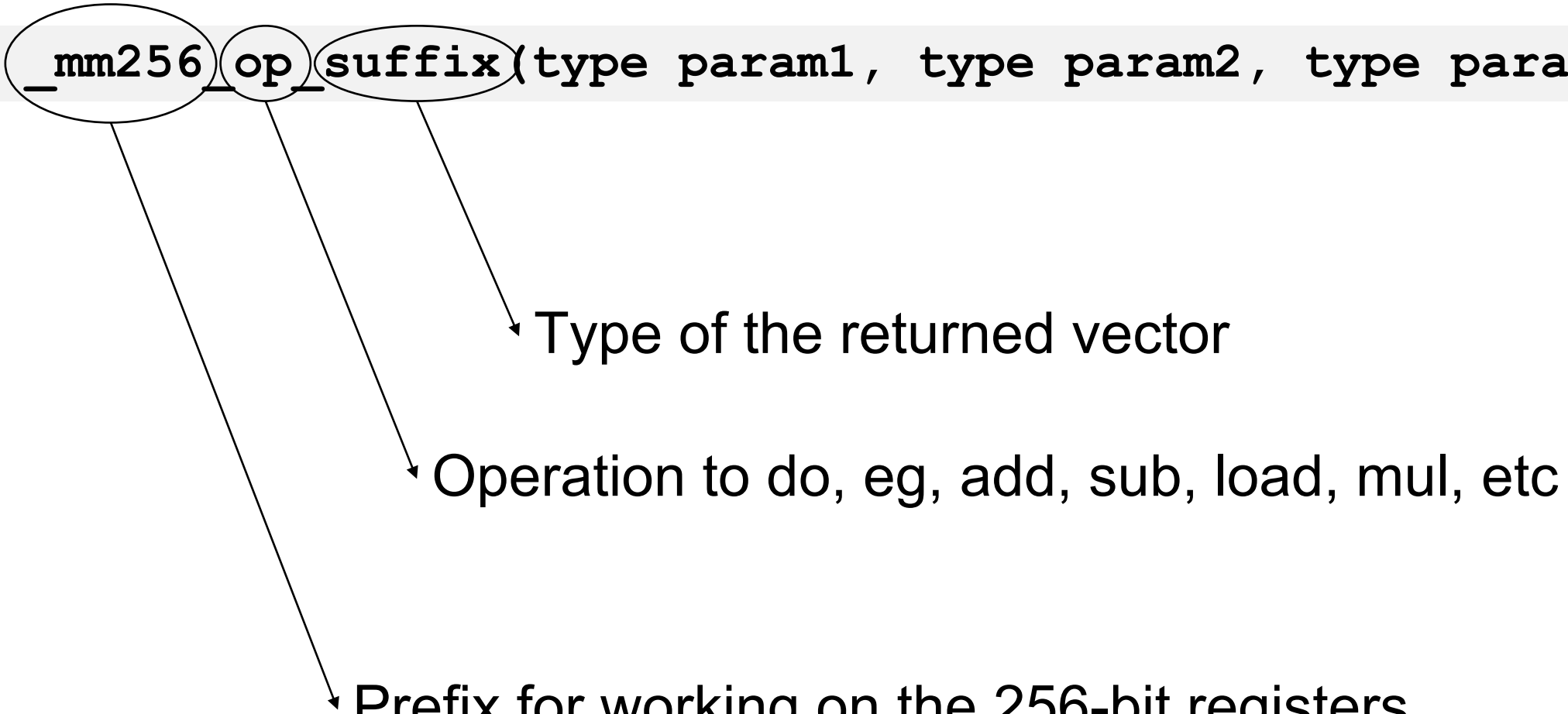
- What are intrinsics?
  - Specialized, function-like calls that directly map to specific CPU instructions
  - Inlined upon compilation: no runtime overhead
  - Include custom datatypes that map to vector registers

AVX Intrinsics Data Types

Type	Meaning
__m256	8x single-precision floats
__m256d	4x double-precision floats
__m256i	32x 8-bit / 16x 16-bit / 8x 32-bit / 4x 64-bit integers
__m128	4x single-precision floats
__m128d	2x double-precision floats

AVX Intrinsics Functions

```
_mm256_op_suffix(type param1, type param2, type param3)
```



AVX Intrinsics Functions

```
_mm256_op_suffix(type param1, type param2, type param3)
```

suffix	Meaning
ps/pd/sd	Packed single / packed double / scalar double
epi{X}$_{\|X = 8, 16, 32, 64}$	Packed 8-bit / 16-bit / 32-bit / 64-bit signed integers
epu{X}$_{\|X = 8, 16, 32 64}$	Packed 8-bit / 16-bit / 32-bit / 64-bit unsigned integers
si256	Scalar 256-bit integer

AVX Intrinsics Functions: Example

```
void daxpy(int n, double a, double *x, double *y)
{
     for(int i = 0; i < n; i++) {
          y[i] = a * x[i] + y[i];
     }
}
```

AVX Intrinsics Functions: Example

```
void daxpy(int n, double a, double *x, double *y)
{
     // Convert a to vector
     double vecA[4] = {a, a, a, a};

     // Process in chunks of 4 doubles
     for(int i = 0; i < n; i+=4) { // Assume n is div by 4
          y[i]   = vecA[i]   * x[i]   + y[i];
          y[i+1] = vecA[i+1] * x[i+1] + y[i+1];
          y[i+2] = vecA[i+2] * x[i+2] + y[i+2];
          y[i+3] = vecA[i+3] * x[i+3] + y[i+3];
     }
}
```

AVX Intrinsics Functions: Example

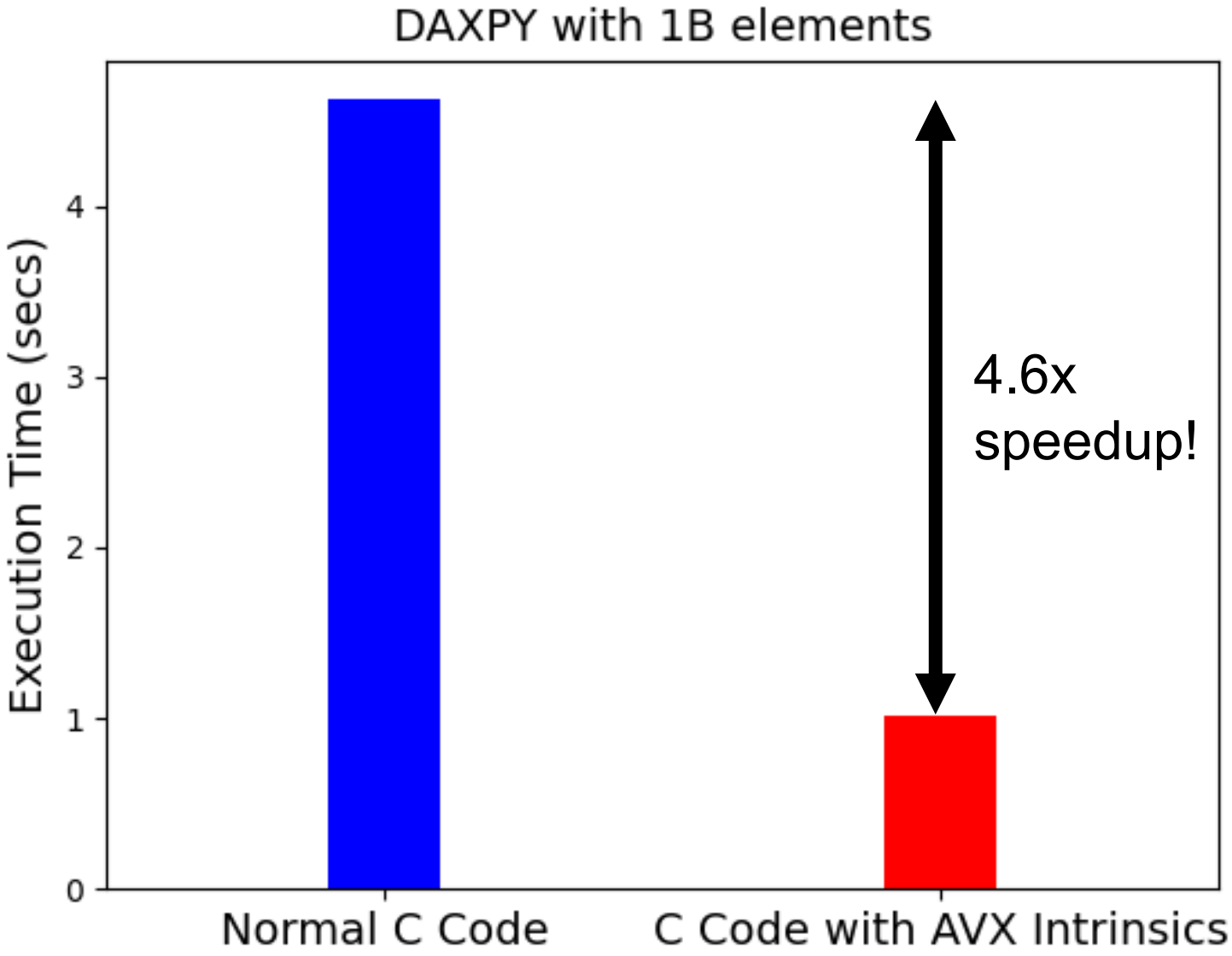
```
__m256d vecA = _mm256_set_pd(a, a, a, a);

for(int i = 0; i < n; i+=4) {
      __m256d vecX = _mm256_load_pd(&x[i]); // Load x[i]…x[i+3]
      __m256d vecY = _mm256_load_pd(&y[i]); // Load y[i]…y[i+3]

      __m256d vecZ = _mm256_mul_pd(vecA, vecX); // Compute a*x
      vecY = _mm256_add_pd(vecZ, vecY);         // … + y

      // Store result back to y
      _mm256_store_pd(&y[i], vecY);
}
```

AVX Intrinsics Functions: Speedup



Summary

- SIMD: Data-level parallelism (one instruction, multiple data)
  - Occurs inside a single core with hardware support
  - Fine-grained parallelism on array/vector elements
  - No synchronization overhead
  - Vector and SIMD in CPU are used interchangeably
- Compiler can help generate code
- Not all code is vectorizable by compilers
  - Developers often directly write their own vector code