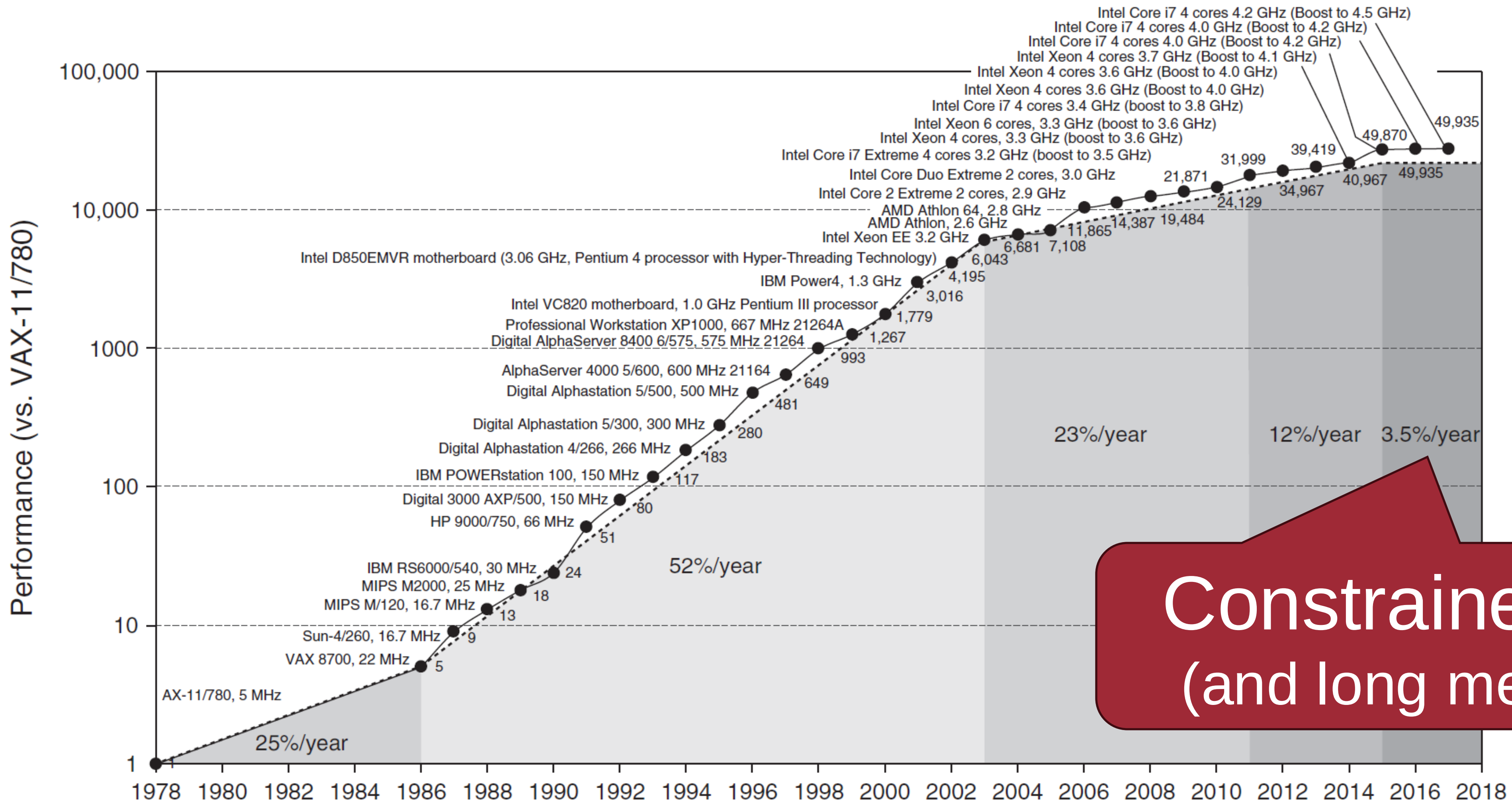
CSE261: Computer Architecture

20. Parallel Architectures

Seongil Wi

Department of Computer Science and Engineering

Uniprocessor Performance



Constrained by power
(and long memory latency)

How we address the power wall?

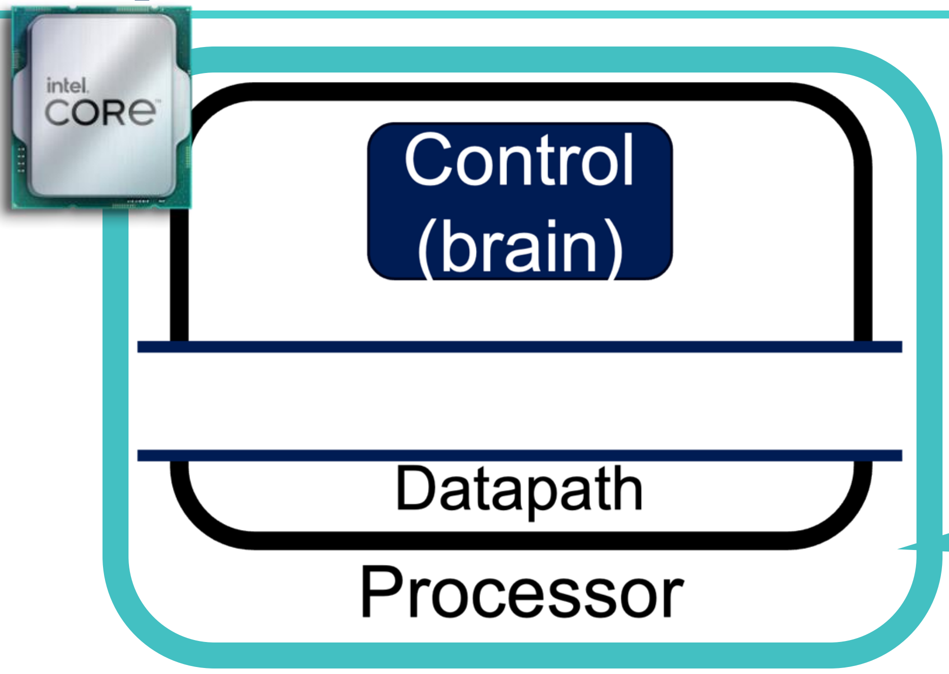
⇒ Parallel architectures

Parallel Processors

- Goal: connecting multiple computers to get higher performance

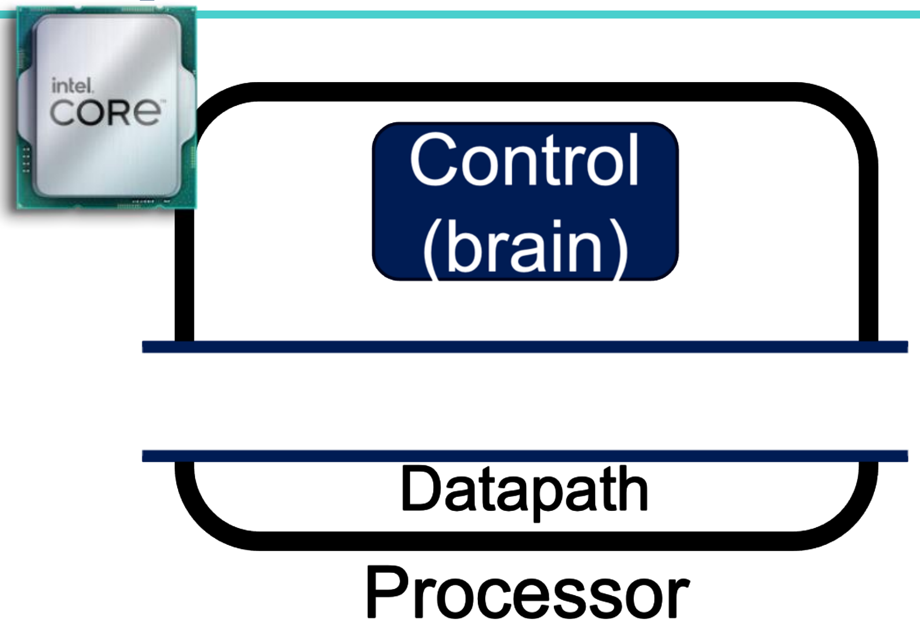
  + availability, power efficiency…

Uniprocessor to Multiprocessors

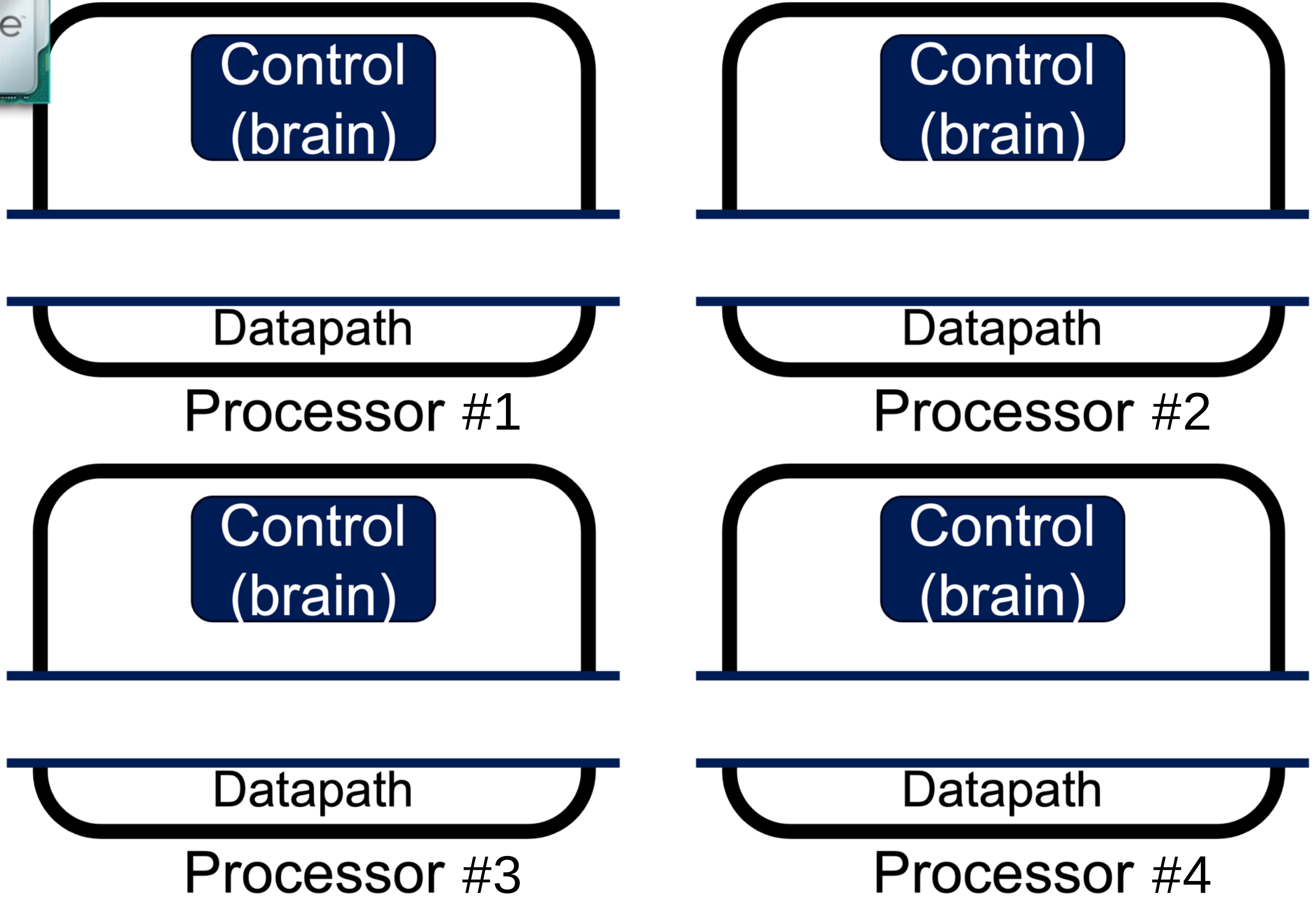


Uniprocessor: one processor (core) per chip

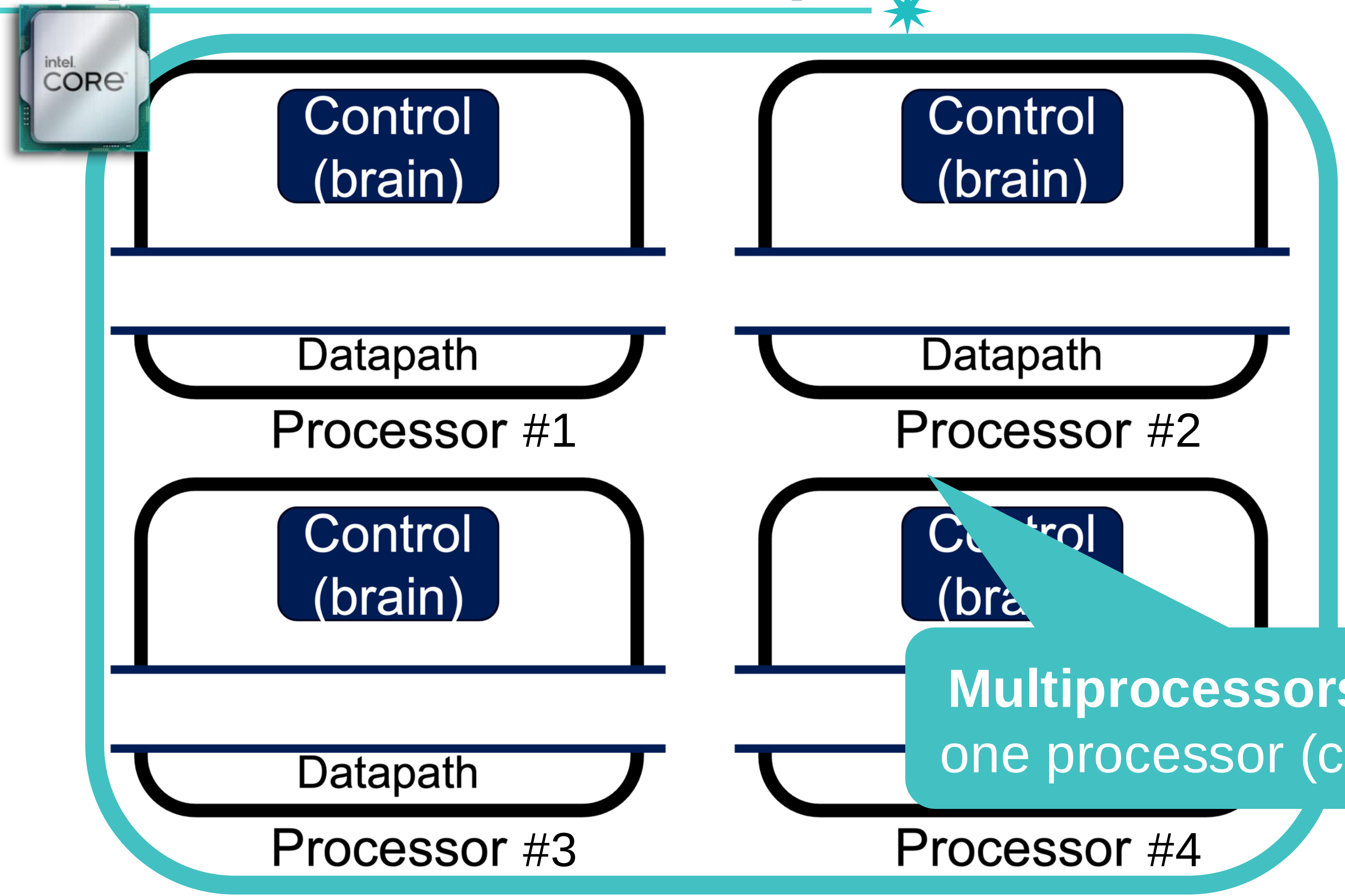
Uniprocessor to Multiprocessors



Uniprocessor to Multiprocessors



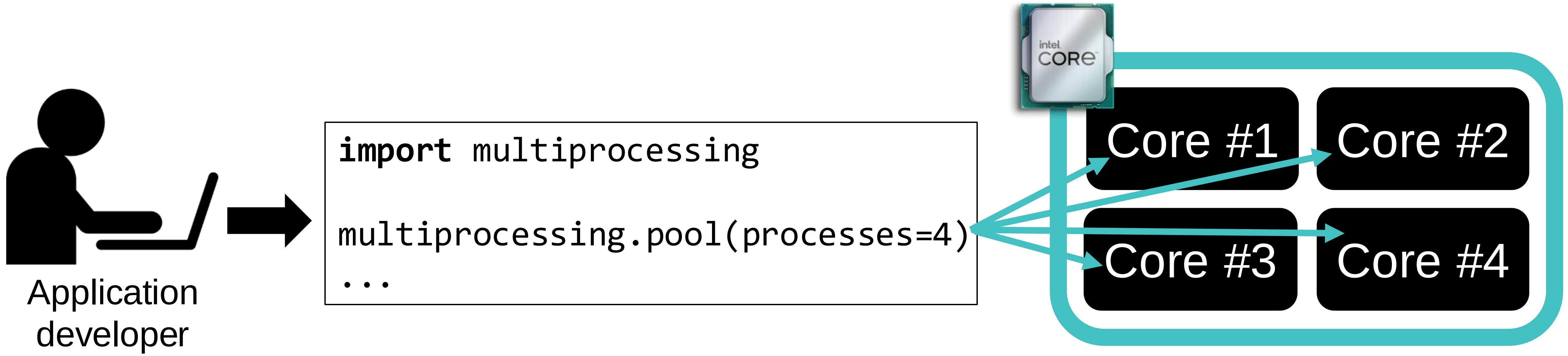
Uniprocessor to Multiprocessors



Multiprocessors: more than one processor (core) per chip

Multiprocessors

- Multicore microprocessors
  - More than one processor (core) per chip
- Requires explicitly **parallel programming**
  - Programming for performance
  - Load balancing
  - Optimizing communication and synchronization



```
import multiprocessing

multiprocessing.pool(processes=4)
...
```

The Challenge with Parallelism

- The challenge is not the hardware, but the **software**
- Why is it difficult to write parallel processing programs (that are fast)?
  - **Partitioning**: the task must be broken into eight equal-sized pieces
  - **Coordination**: synchronize between tasks
  - **Communication overhead**: spend time to communicate each other
  - **Amdahl's law**: sequential part can limit speed up!

Application developer

```
import multiprocessing

multiprocessing.pool(processes=4)
...
```

The challenge is getting worse with more processors

Pitfall: Amdahl's Law (Amdahl's Argument)

"The overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used"

The **speedup** of parallel computations can be estimated:

$$\text{Speedup} = \frac{1}{(1-P) + \frac{P}{n}}$$

P: parallel portion

n: # of processor cores

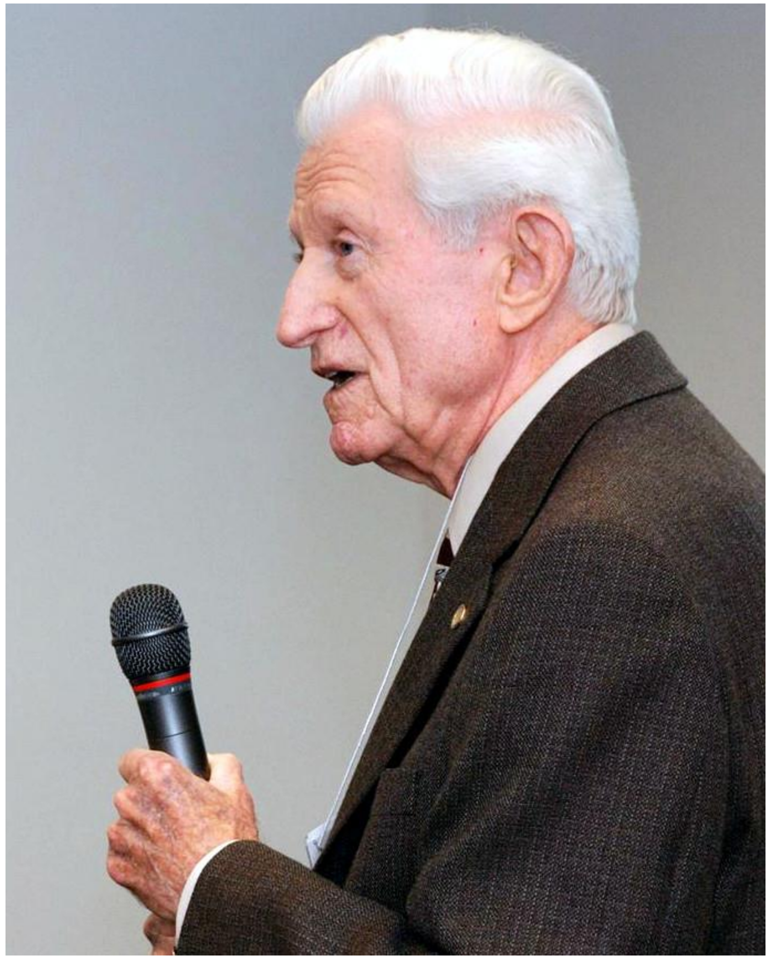

Gene Myron Amdahl (1922-2015)

Image from https://en.wikipedia.org/wiki/Gene_Amdahl

Pitfall: Amdahl's Law (Amdahl's Argument)

"The overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used"

The **speedup** of parallel computations can be estimated:

$$\text{Speedup} = \frac{1}{(1-P) + \frac{P}{n}}$$

1-P: time spent executing the sequential portion

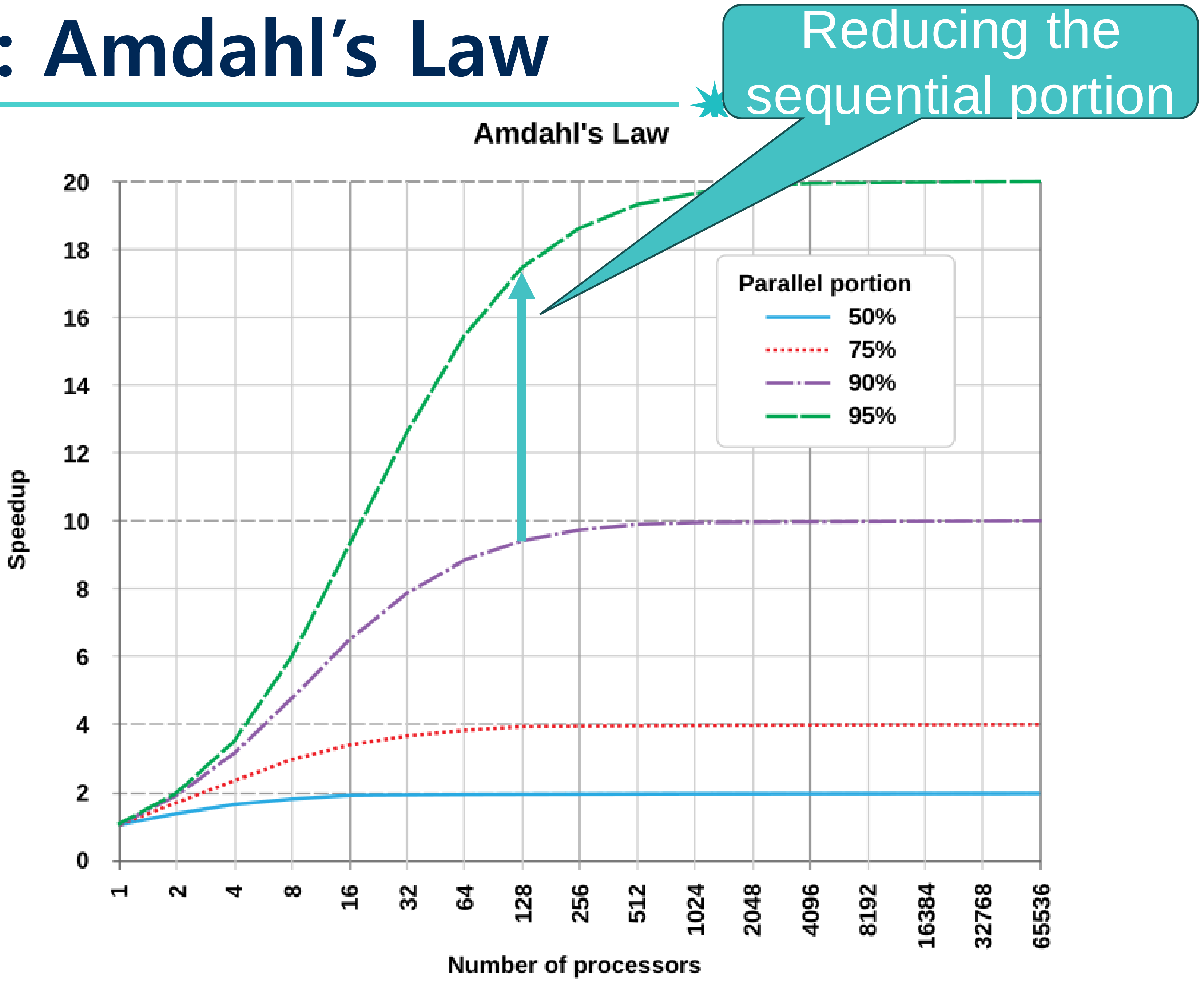
P/n: time spent executing the parallel portion

P: parallel portion

n: # of processor cores

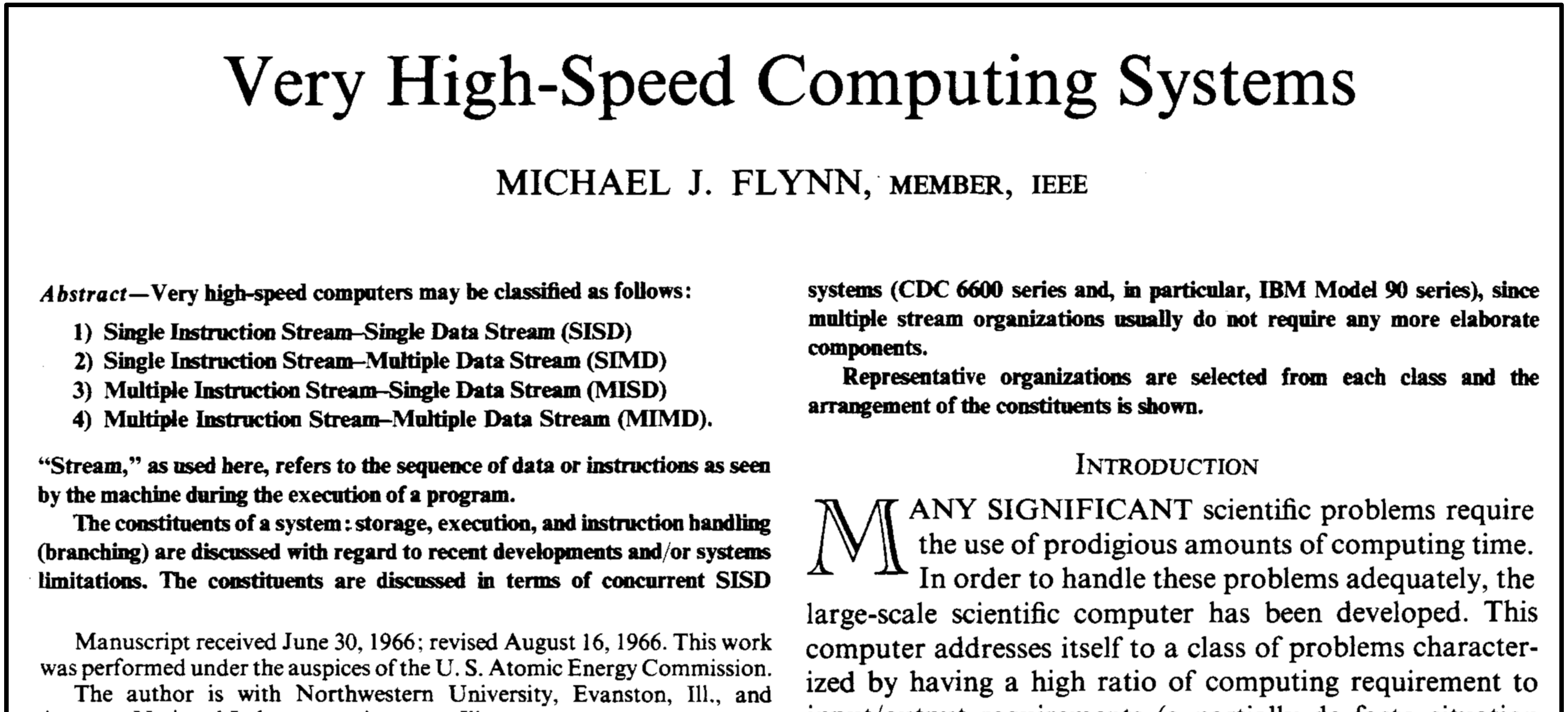
Image from https://en.wikipedia.org/wiki/Gene_Amdahl

Pitfall: Amdahl’s Law



Instruction and Data Streams

- According to the Flynn's taxonomy of computers, …
  - Mike Flynn, "Very High-Speed Computing Systems", Proc. Of IEEE, 1966



Very High-Speed Computing Systems

MICHAEL J. FLYNN, MEMBER, IEEE

Abstract—Very high-speed computers may be classified as follows:

1) Single Instruction Stream–Single Data Stream (SISD)
2) Single Instruction Stream–Multiple Data Stream (SIMD)
3) Multiple Instruction Stream–Single Data Stream (MISD)
4) Multiple Instruction Stream–Multiple Data Stream (MIMD).

"Stream," as used here, refers to the sequence of data or instructions as seen by the machine during the execution of a program.

The constituents of a system: storage, execution, and instruction handling (branching) are discussed with regard to recent developments and/or systems limitations. The constituents are discussed in terms of concurrent SISD systems (CDC 6600 series and, in particular, IBM Model 90 series), since multiple stream organizations usually do not require any more elaborate components.

Representative organizations are selected from each class and the arrangement of the constituents is shown.

Manuscript received June 30, 1966; revised August 16, 1966. This work was performed under the auspices of the U. S. Atomic Energy Commission.

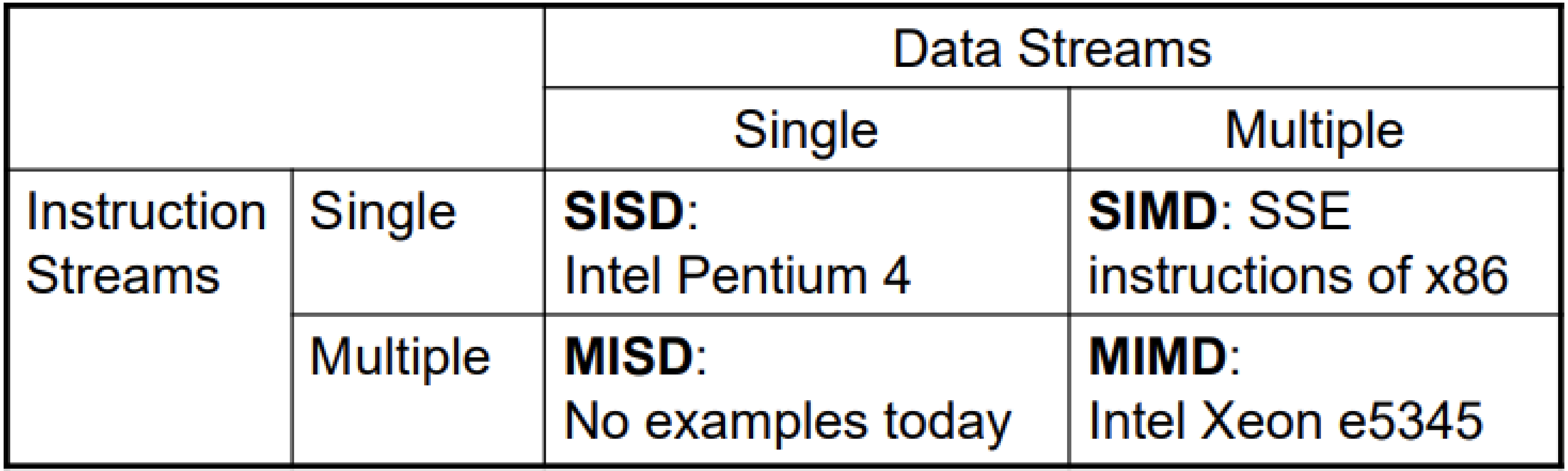
The author is with Northwestern University, Evanston, Ill., and

Introduction

MANY SIGNIFICANT scientific problems require the use of prodigious amounts of computing time. In order to handle these problems adequately, the large-scale scientific computer has been developed. This computer addresses itself to a class of problems characterized by having a high ratio of computing requirement to

Instruction and Data Streams

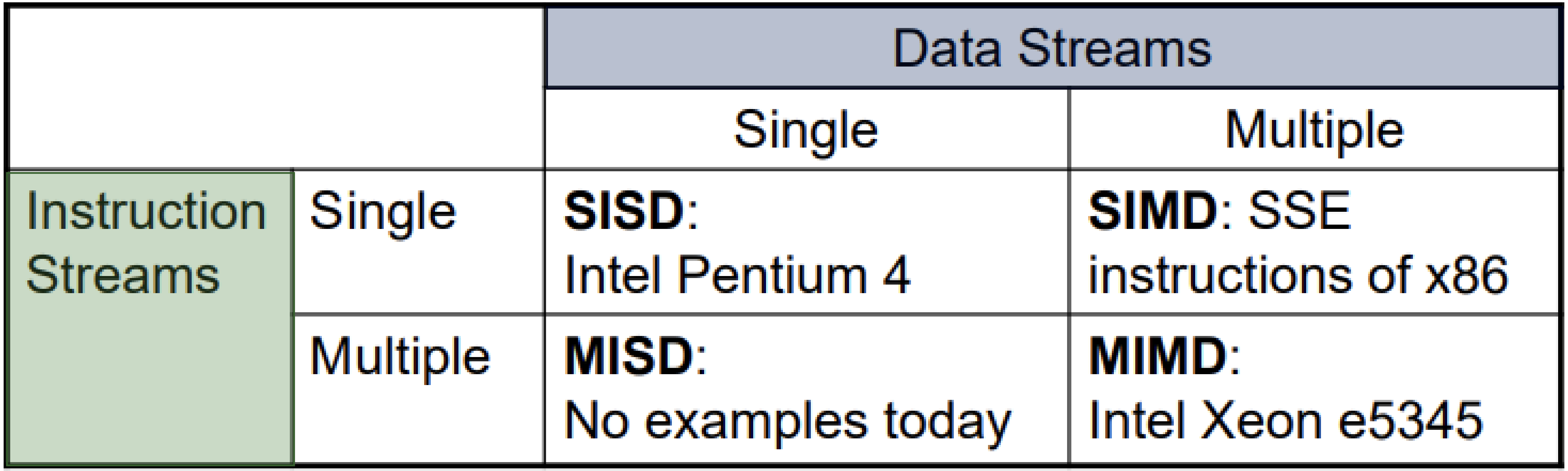
- According to the Flynn’s taxonomy of computers, …



		Data Streams	
		Single	Multiple
Instruction Streams	Single	**SISD**: Intel Pentium 4	**SIMD**: SSE instructions of x86
	Multiple	**MISD**: No examples today	**MIMD**: Intel Xeon e5345

Instruction and Data Streams

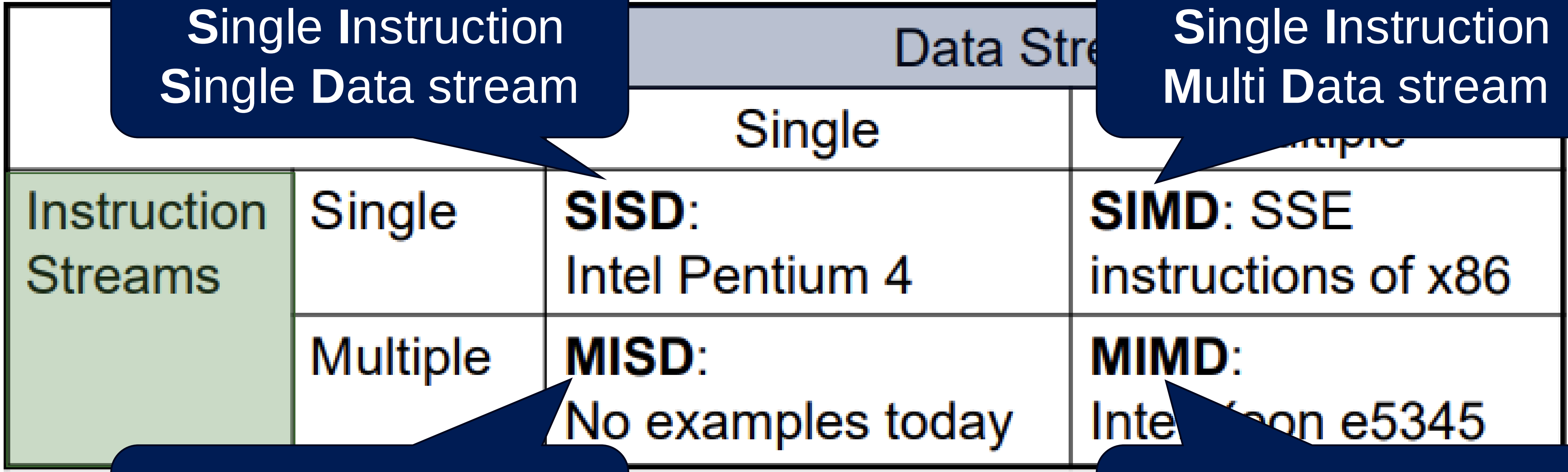
- According to the Flynn's taxonomy of computers, …



		Data Streams	
		Single	Multiple
Instruction Streams	Single	**SISD**: Intel Pentium 4	**SIMD**: SSE instructions of x86
	Multiple	**MISD**: No examples today	**MIMD**: Intel Xeon e5345

Instruction and Data Streams

- According to the Flynn’s taxonomy of computers, …



		Data Stre	
		Single	ltiple
Instruction Streams	Single	**SISD:** Intel Pentium 4	**SIMD:** SSE instructions of x86
	Multiple	**MISD:** No examples today	**MIMD:** Inte eon e5345

Single Instruction Single Data stream

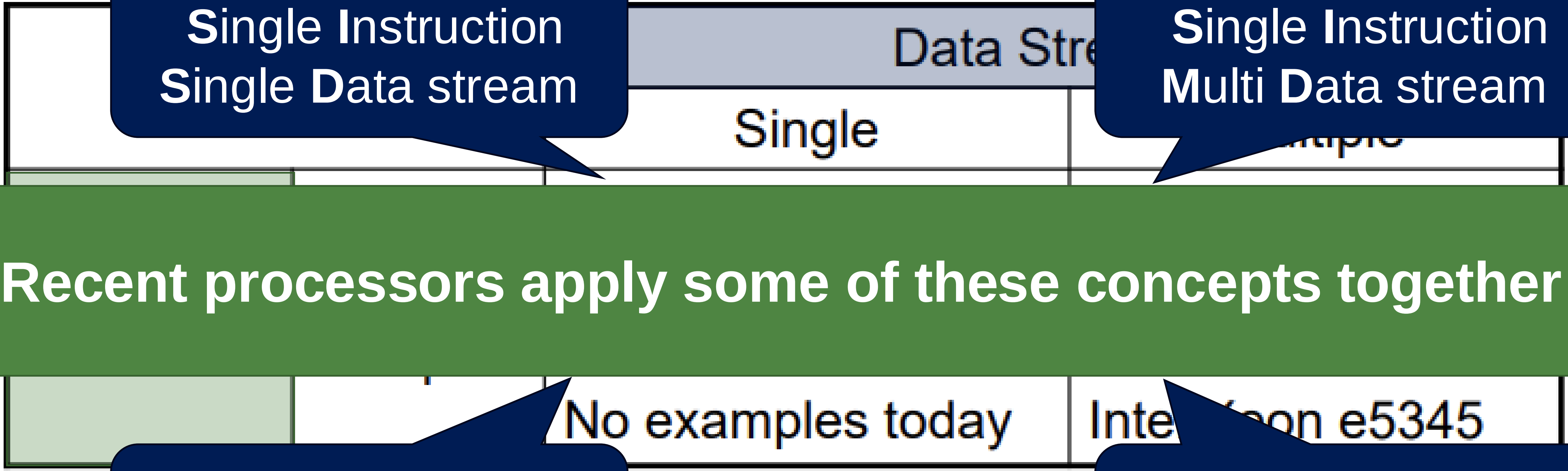
Single Instruction Multi Data stream

Multi Instruction Single Data stream

Multi Instruction Multi Data stream

Instruction and Data Streams

- According to the Flynn's taxonomy of computers, …



Data Stre

Single

No examples today

Inte

on e5345

Single Instruction
Single Data stream

Single Instruction
Multi Data stream

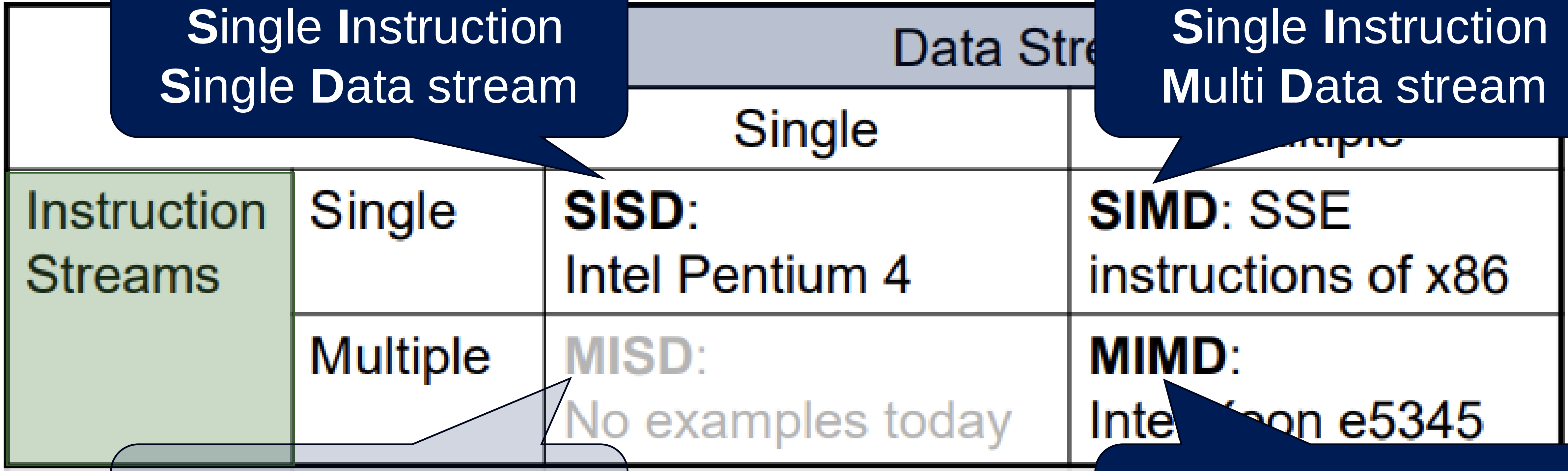
Recent processors apply some of these concepts together

Multi Instruction
Single Data stream

Multi Instruction
Multi Data stream

Instruction and Data Streams

- According to the Flynn's taxonomy of computers, …



		Data Stre	
		Single	ltiple
Instruction Streams	Single	**SISD**: Intel Pentium 4	**SIMD**: SSE instructions of x86
	Multiple	**MISD**: No examples today	**MIMD**: Inte on e5345

Single Instruction Single Data stream

Single Instruction Multi Data stream

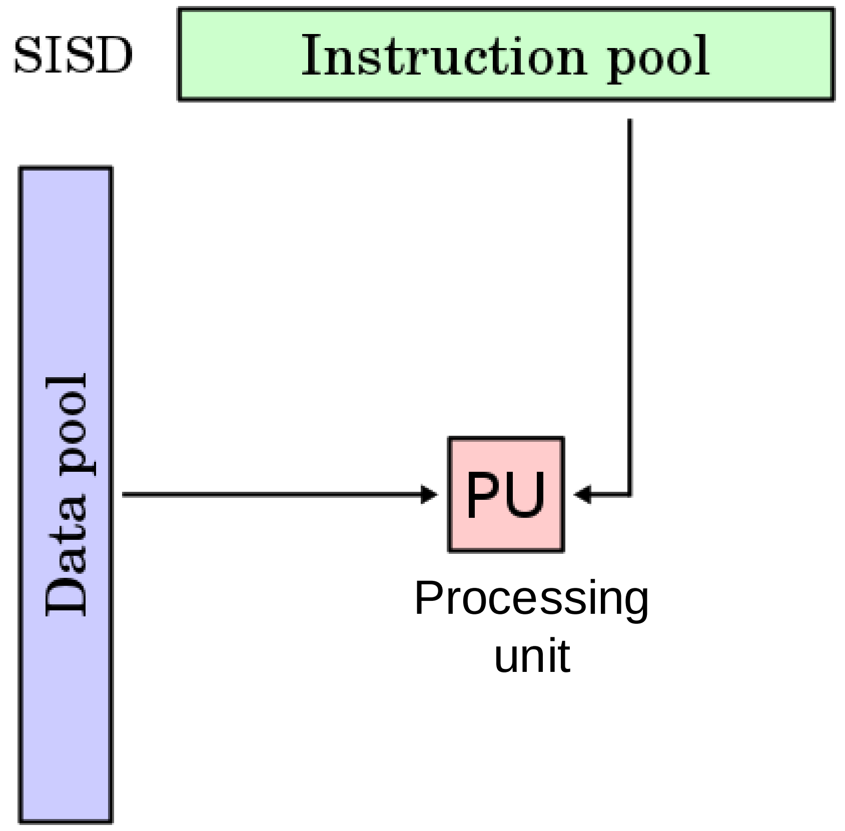
Multi Instruction Single Data stream

Multi Instruction Multi Data stream

Single Instruction Single Data stream (SISD)

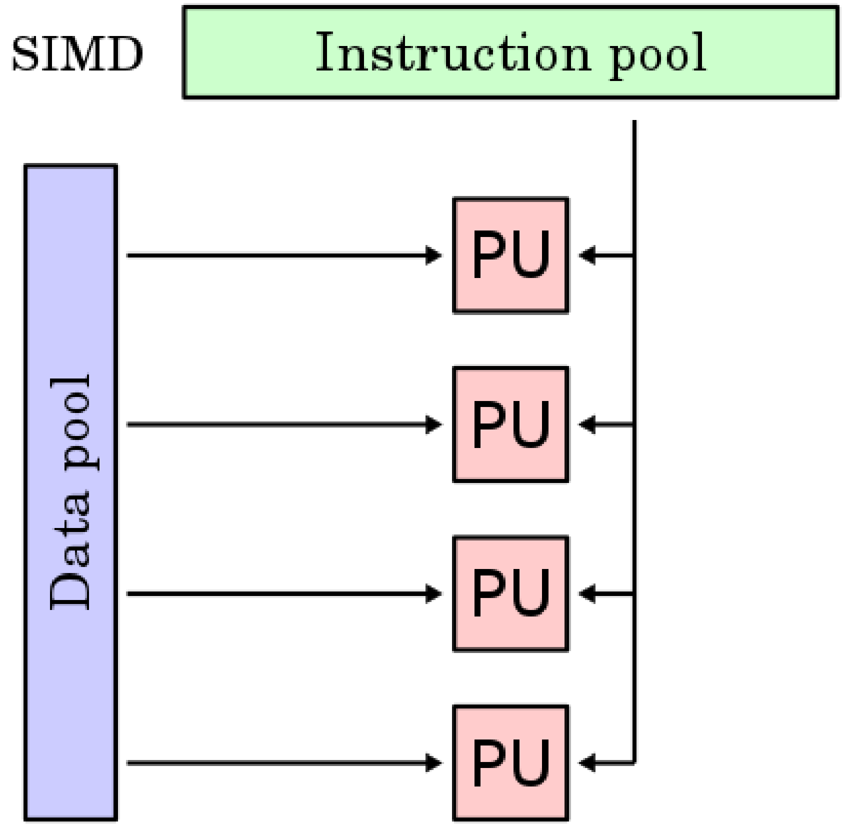
- Uniprocessor

- Disadvantages
  - Limited speed due to being a single core
  - Pipelining can be implemented, but only one instruction will be executed at a time



Single Instruction Multi Data stream (SIMD)

- A single instruction is executed on multiple different pieces of data

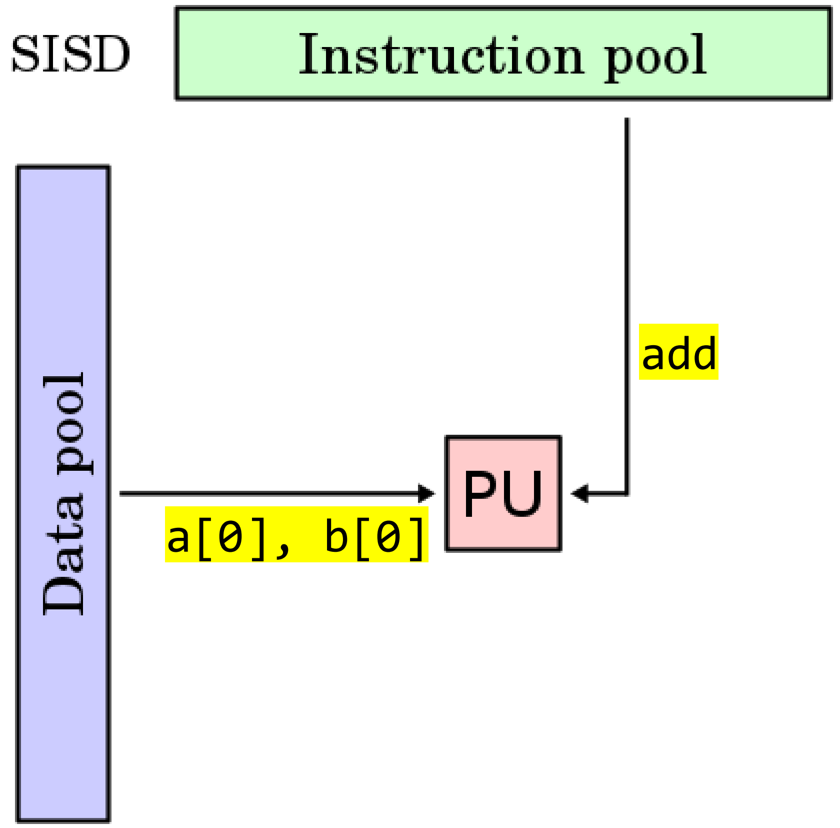


SIMD: Motivating Example

```
for (int i = 0; i < 4; i++) {
    c[i] = a[i] + b[i];
}
```

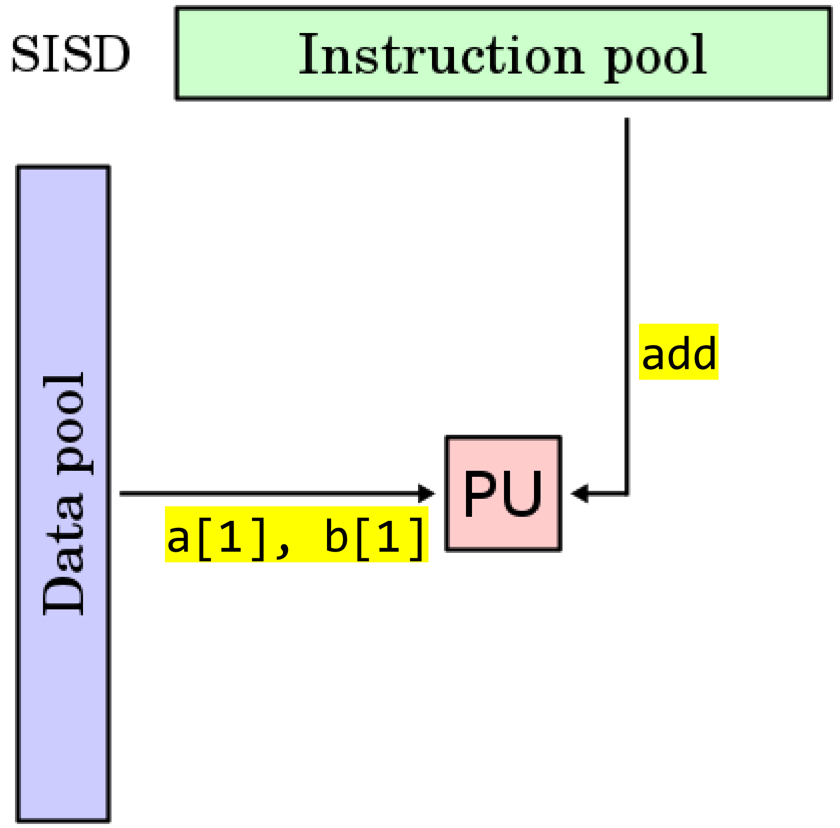
SIMD: Motivating Example

```
for (int i = 0; i < 4; i++) {
        c[i] = a[i] + b[i];
}
```



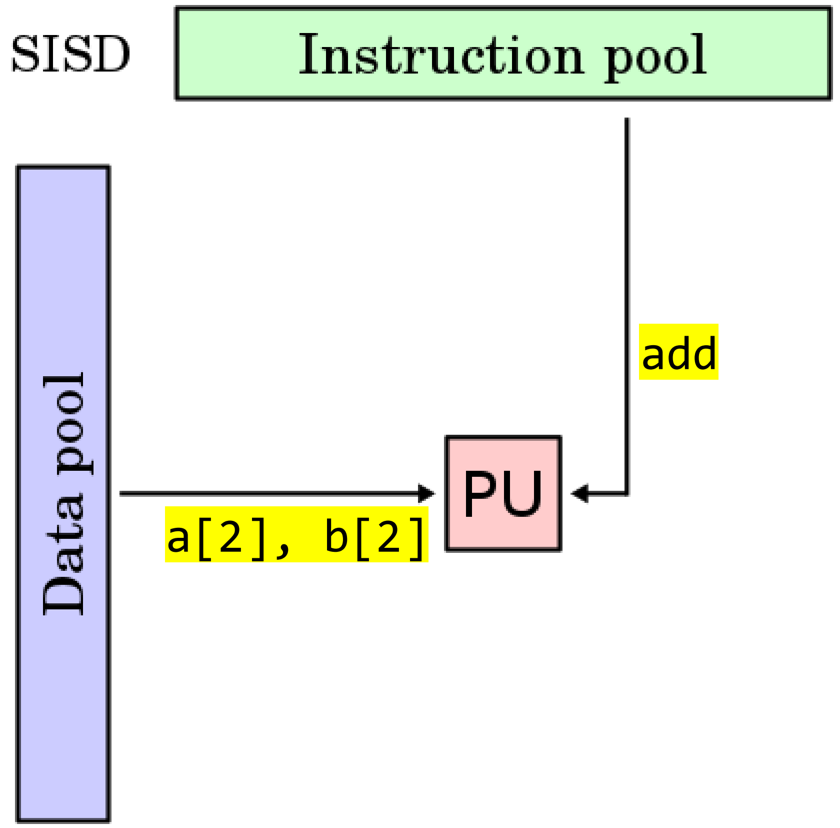
SIMD: Motivating Example

```
for (int i = 0; i < 4; i++) {
    c[i] = a[i] + b[i];
}
```



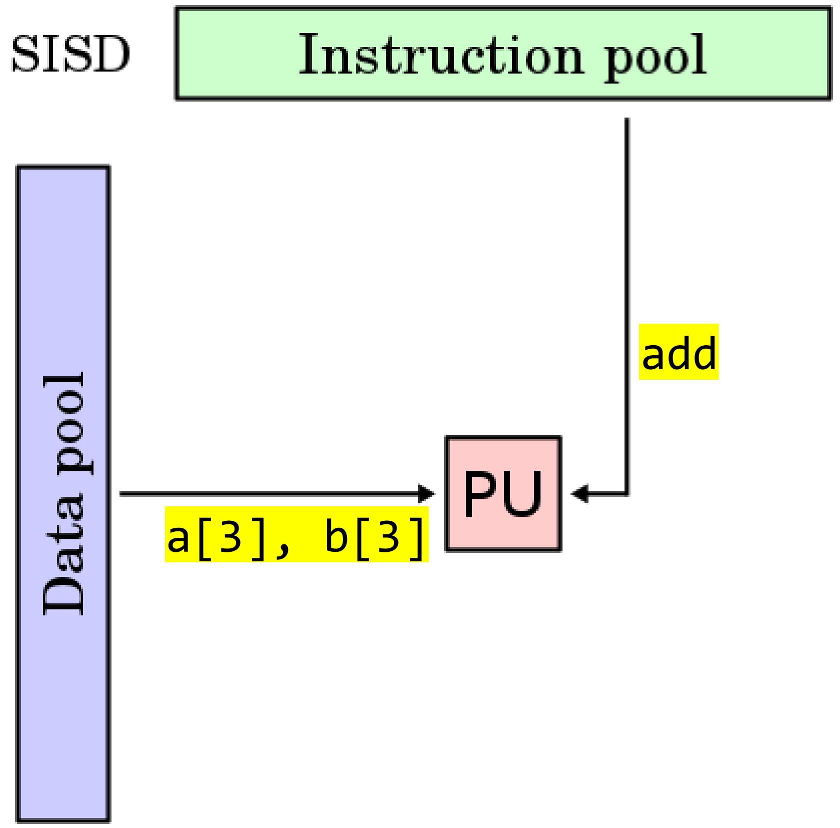
SIMD: Motivating Example

```
for (int i = 0; i < 4; i++) {
    c[i] = a[i] + b[i];
}
```



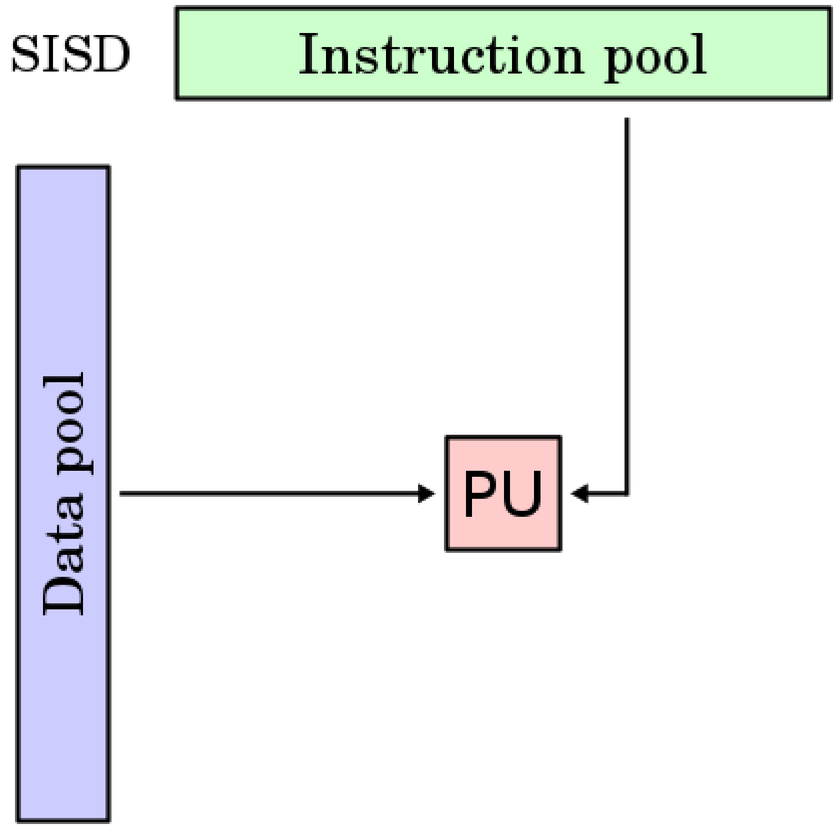
SIMD: Motivating Example

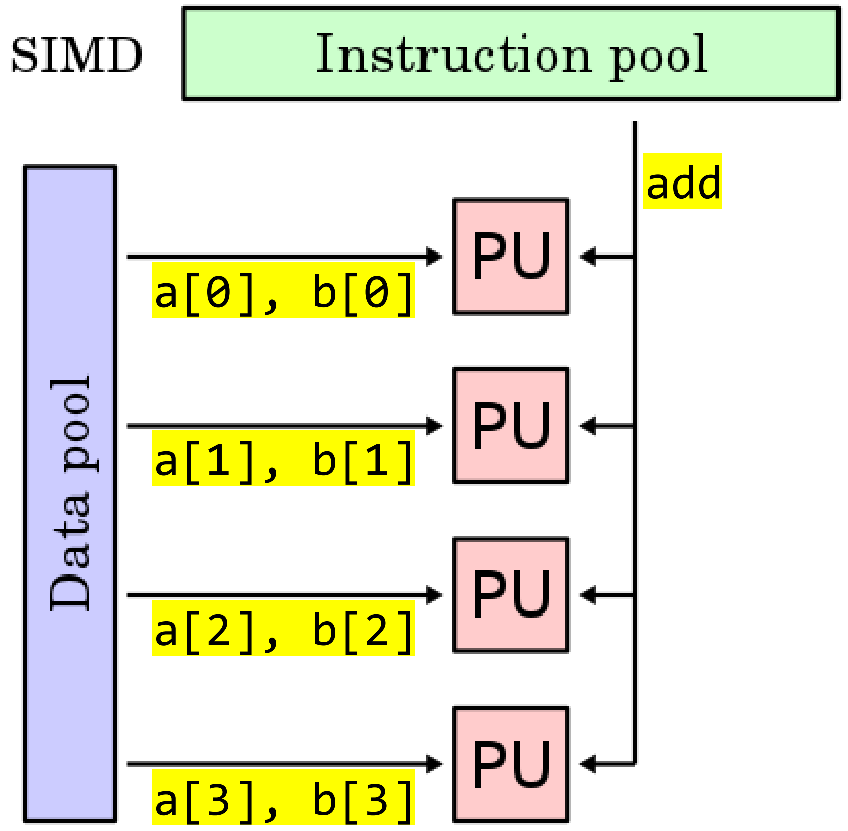
```
for (int i = 0; i < 4; i++) {
        c[i] = a[i] + b[i];
}
```



SIMD: Motivating Example

```
for (int i = 0; i < 4; i++) {
        c[i] = a[i] + b[i];
}
```

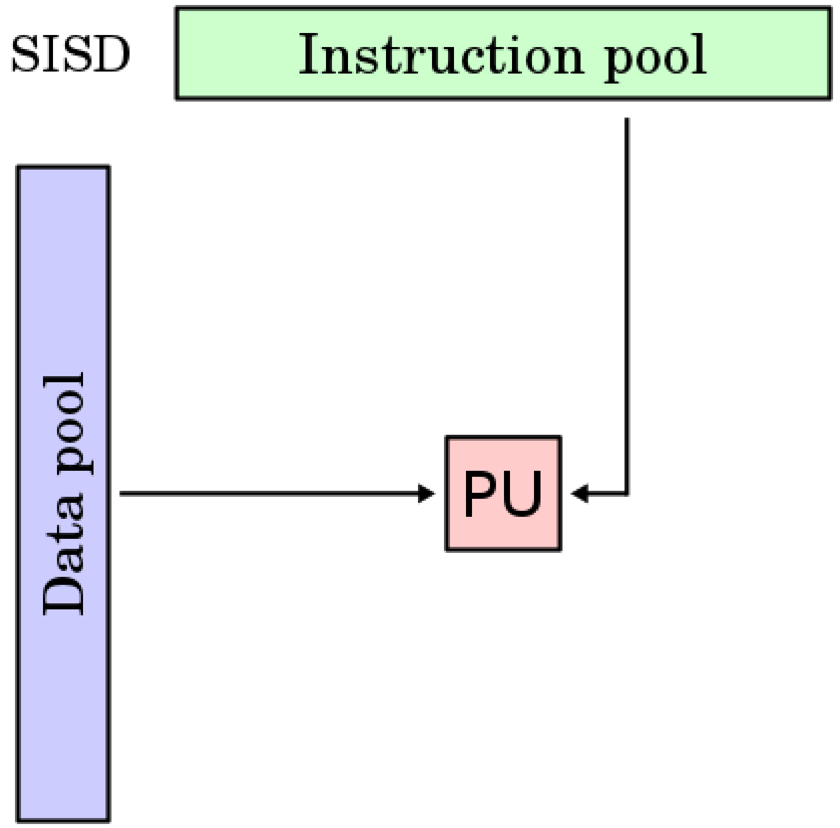


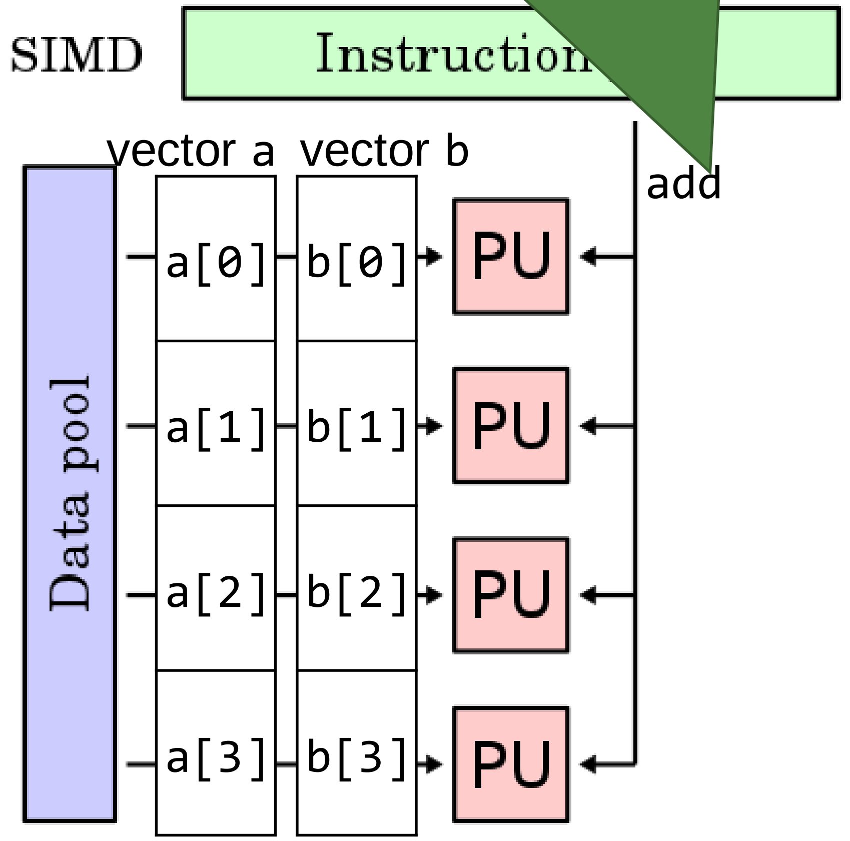


SIMD: Motivating Example

```
for (int i = 0; i
        c[i] = a[i] +
}
```

Single control unit dispatches the same (single) instruction

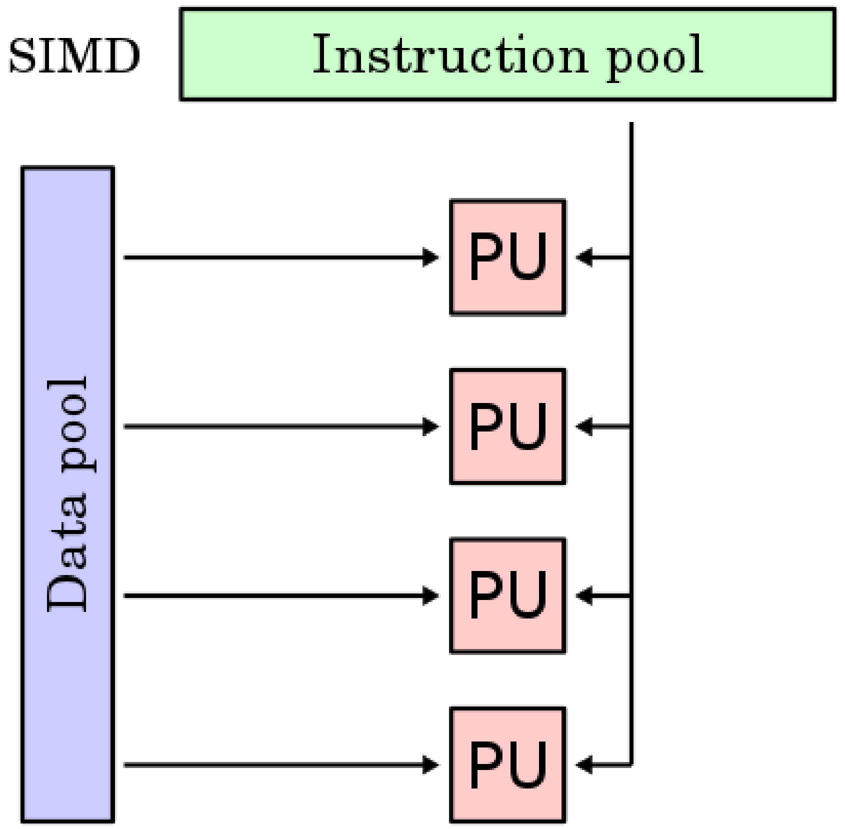




Single Instruction Multi Data stream (SIMD)

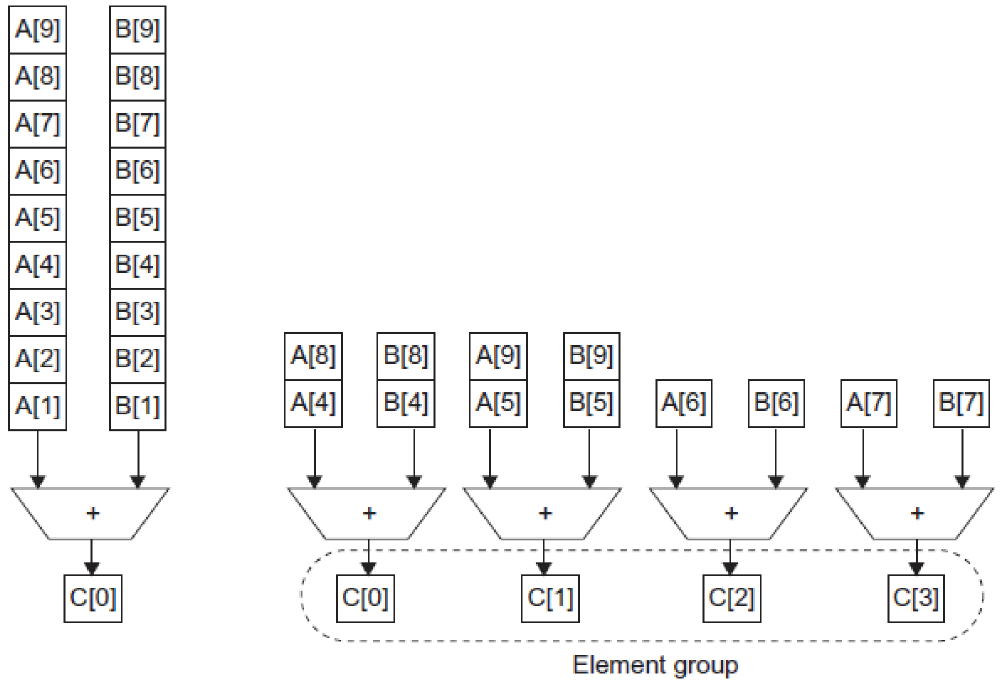
- A <u>single instruction</u> is executed on <u>multiple different pieces of data</u>

- Excels for computations with regular structure (e.g., linear algebra)
- Works best for data-parallel applications
- **Primary application**: vector processing
  - e.g., numpy in Python



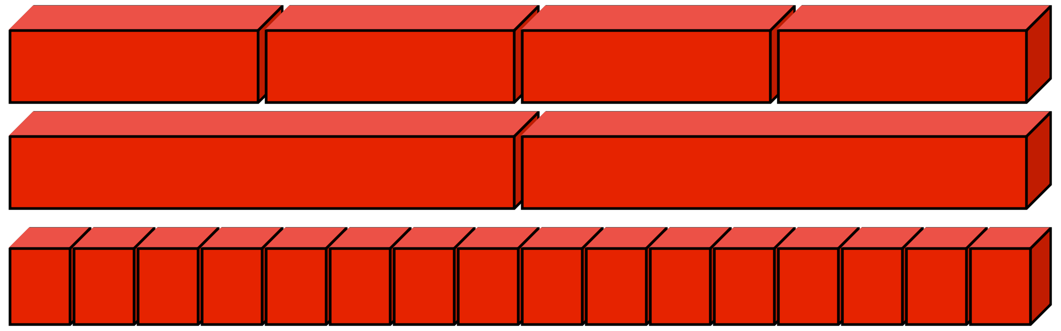
Vector Processing

- Highly pipelined function units
- Stream data from/to vector registers to units
  - Data collected from memory into registers
  - Operate on them sequentially using pipelined execution units
  - Results stored from registers to memory



Example: 128-bit SIMD Vectors

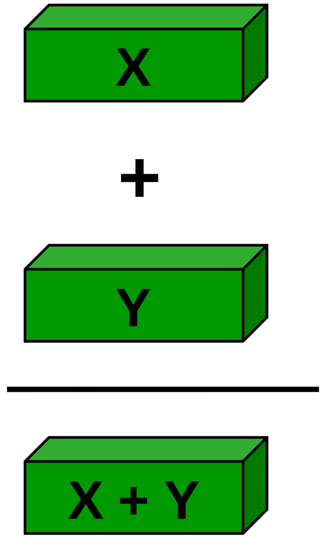
- Data types: anything that fits into 16 bytes, e.g.,

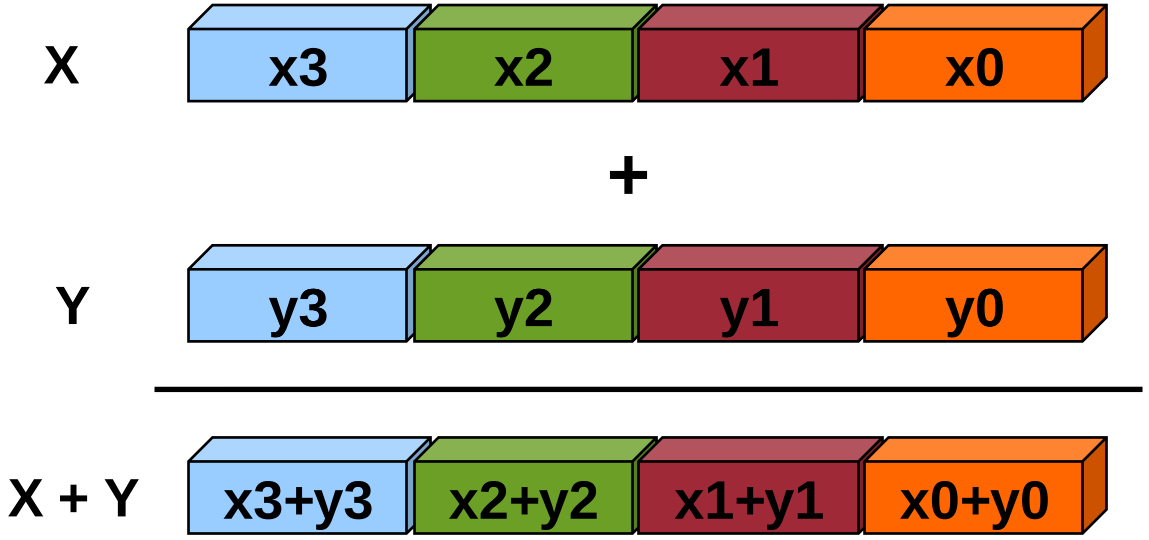


- Data bytes must be contiguous in memory and aligned
- Instructions operate in parallel on data in this 16-byte register
  - add, subtract, multiply, etc.
- Additional instructions needed for
  - Masking data
  - Moving data from one part of a register to another

Computing with SIMD Vector Units

- SISD processing
  - One operation produces one result
- SIMD vector units
  - One operation produces multiple results





Vector Extension to MIPS (VMIPS)

- 32 vector registers: `v0` to `v31` (64 64-bit elements)
- Vector instructions
  - `lv`, `sv`: load/store vector
  - `addv.d`: add vectors of double
  - `addvs.d`: add scalar to each element of vector of double

Example: DAXPY (Y = a*X + Y)

- Address of X in $s0 and Y in $s1

SISD

```
      l.d    $f0,a($sp)      ;load scalar a
      addiu  $t0,$s0,#512    ;upper bound of what to load
loop: l.d    $f2,0($s0)      ;load x(i)
      mul.d  $f2,$f2,$f0     ;a × x(i)
      l.d    $f4,0($s1)      ;load y(i)
      add.d  $f4,$f4,$f2     ;a × x(i) + y(i)
      s.d    $f4,0($s1)      ;store into y(i)
      addiu  $s0,$s0,#8      ;increment index to x
      addiu  $s1,$s1,#8      ;increment index to y
      subu   $t0,r4,$s0      ;compute bound
      bne    $t0,$zero,loop  ;check if done
```

Example: DAXPY (Y = a*X + Y)

0 and Y in $s1

1. Load for one element (X[n])

2. Compute a*X[n]

3. Load for one element (Y[n])

4. Compute a*X[n] + Y[n]

5. Store for one element

```
      l.d   $f0,a($sp)       ;load scalar a
      addiu $t0,$s0,#512     ;upper bound of what to load
loop: l.d   $f2,0($s0)       ;load x(i)
      mul.d $f2,$f2,$f0      ;a × x(i)
      l.d   $f4,0($s1)       ;load y(i)
      add.d $f4,$f4,$f2      ;a × x(i) + y(i)
      s.d   $f4,0($s1)       ;store into y(i)
      addiu $s0,$s0,#8       ;increment index to x
      addiu $s1,$s1,#8       ;increment index to y
      subu  $t0,r4,$s0       ;compute bound
      bne   $t0,$zero,loop   ;check if done
```

Example: DAXPY (Y = a*X + Y)

- Address of X in $s0 and Y in $s1

SISD

```
      l.d    $f0,a($sp)       ;load scalar a
      addiu $t0,$s0,#512      ;upper bound of what to load
loop: l.d    $f2,0($s0)       ;load x(i)
      mul.d $f2,$f2,$f0       ;a × x(i)
      l.d    $f4,0($s1)       ;load y(i)
      add.d $f4,$f4,$f2       ;a × x(i) + y(i)
      s.d    $f4,0($s1)       ;store into y(i)
      addiu $s0,$s0,#8        ;increment index to x
      addiu $s1,$s1,#8        ;increment index to y
      subu   $t0,r4,$s0       ;compute bound
      bne    $t0,$zero,loop   ;check if done
```

SIMD

```
      l.d      $f0,a($sp)    ;load scalar a
      lv       $v1,0($s0)    ;load vector x
      mulvs.d $v2,$v1,$f0    ;vector-scalar multiply
      lv       $v3,0($s1)    ;load vector y
      addv.d  $v4,$v2,$v3    ;add y to product
      sv       $v4,0($s1)    ;store the result
```

Example: DAXPY (Y = a*X + Y)

- Address of X in $s0 and Y in $s1

SISD

```
        l.d    $f0,a($sp)      ;load scalar a
        addiu  $t0,$s0,#512    ;upper bound of what to load
               $f2,0($s0)      ;load x(i)
            d  $f2,$f2,$f0     ;a × x(i)
               $f4,0($s1)      ;load y(i)
        add.d  $f4,$f4,$f2     ;a × x(i) + y(i)
        s.d    $f4,0($s1)      ;store into y(i)
        addiu  $s0,$s0,#8      ;increment index to x
        addiu  $s1,$s1,#8      ;increment index to y
        subu   $t0,r4,$s0      ;compute bound
         ne    $t0,$zero,loop  ;check if done
```

SIMD

```
        l.d      $f0,a($sp)    ;load scalar a
        lv       $v1,0($s0)    ;load vector x
        mulvs.d  $v2,$v1,$f0   ;vector-scalar multiply
        lv       $v3,0($s1)    ;load vector y
        addv.d   $v4,$v2,$v3   ;add y to product
        sv       $v4,0($s1)    ;store the result
```

1. Load for one vector X

2. Compute a*X

3. Load for one vector Y

4. Compute a*X + Y

5. Store for one vector

Example: DAXPY (Y = a*X + Y)

- Address of X in $s0 and Y in $s1

Almost 600 instructions

Vs.

6 instructions

SISD

```
      l.d   $f0,a($sp)      ;load scalar a
      addiu $t0,$s0,#512    ;upper bound of what to load
loop: l.d   $f2,0($s0)      ;load x(i)
      mul.d $f2,$f2,$f0     ;a × x(i)
      l.d   $f4,0($s1)      ;load y(i)
      add.d $f4,$f4,$f2     ;a × x(i) + y(i)
      s.d   $f4,0($s1)      ;store into y(i)
      addiu $s0,$s0,#8      ;increment index to x
      addiu $s1,$s1,#8      ;increment index to y
      subu  $t0,r4,$s0      ;compute bound
      bne   $t0,$zero,loop  ;check if done
```

SIMD

```
      l.d     $f0,a($sp)    ;load scalar a
      lv      $v1,0($s0)    ;load vector x
      mulvs.d $v2,$v1,$f0   ;vector-scalar multiply
      lv      $v3,0($s1)    ;load vector y
      addv.d  $v4,$v2,$v3   ;add y to product
      sv      $v4,0($s1)    ;store the result
```

Example: DAXPY (Y = a*X + Y)

- Address of X in $s0 and Y in $s1

SISD

```
       l.d    $f0,a($sp)      ;load scalar a
       addiu  $t0,$s0,#512    ;upper bound of what to load
loop:  l.d    $f2,0($s0)      ;load x(i)
       mul.d  $f2,$f2,$f0     ;a × x(i)
       l.d    $f4,0($s1)      ;load y(i)
       add.d  $f4,$f4,$f2     ;a × x(i) + y(i)
       s.d    $f4,0($s1)      ;store into y(i)
       addiu  $s0,$s0,#8      ;increment index to x
       addiu  $s1,$s1,#8      ;increment index to y
       subu   $t0,r4,$s0      ;compute bound
       bne    $t0,$zero,loop  ;check if done
```

Stalls per vector element

SIMD

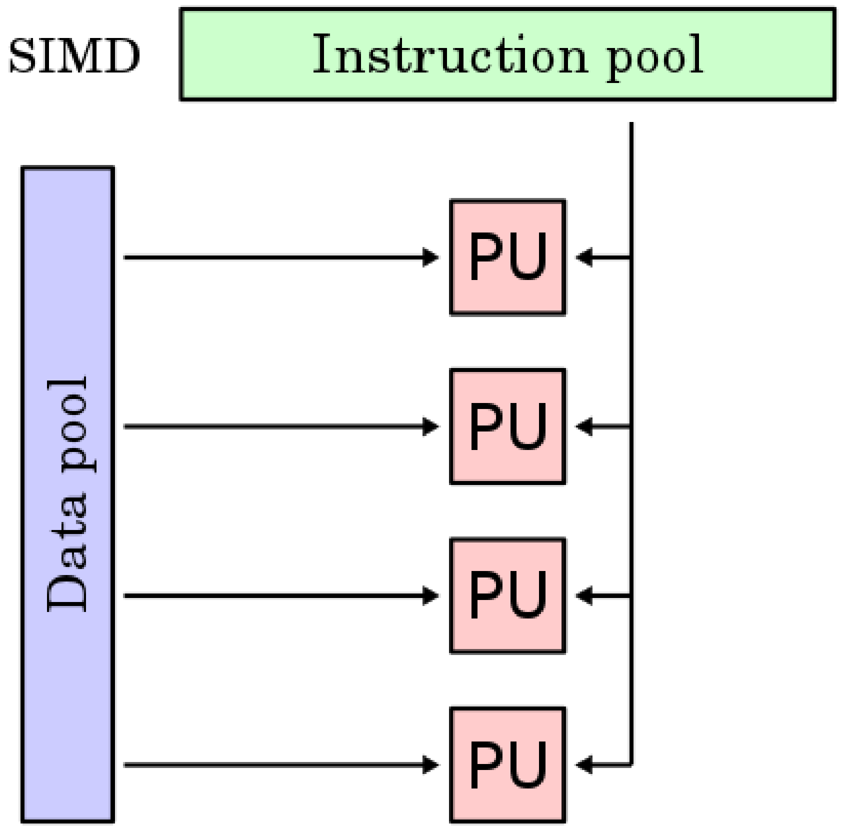
```
       l.d      $f0,a($sp)    ;load scalar a
       lv       $v1,0($s0)    ;load vector x
       mulvs.d  $v2,$v1,$f0   ;vector-scalar multiply
       lv       $v3,0($s1)    ;load vector y
       addv.d   $v4,$v2,$v3   ;add y to product
       sv       $v4,0($s1)    ;store the result
```

Stalls only once per vector operation

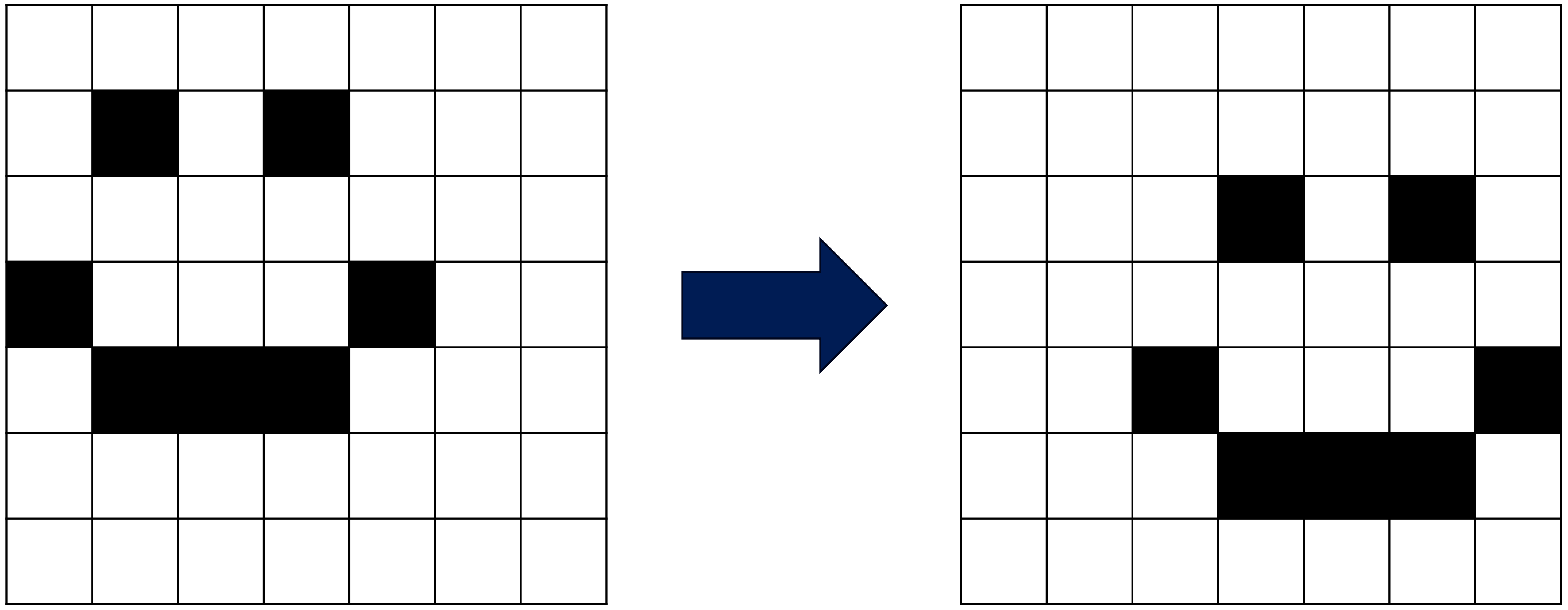
Single Instruction Multi Data stream (SIMD)

- A single instruction is executed on multiple different pieces of data

- Excels for computations with regular structure (e.g., linear algebra)
- Works best for data-parallel applications
- **Primary application**: vector processing
  - e.g., numpy in Python
- Modern GPUs, containing vector processors, are commonly SIMD systems
- Disadvantages
  - Limited to specific applications



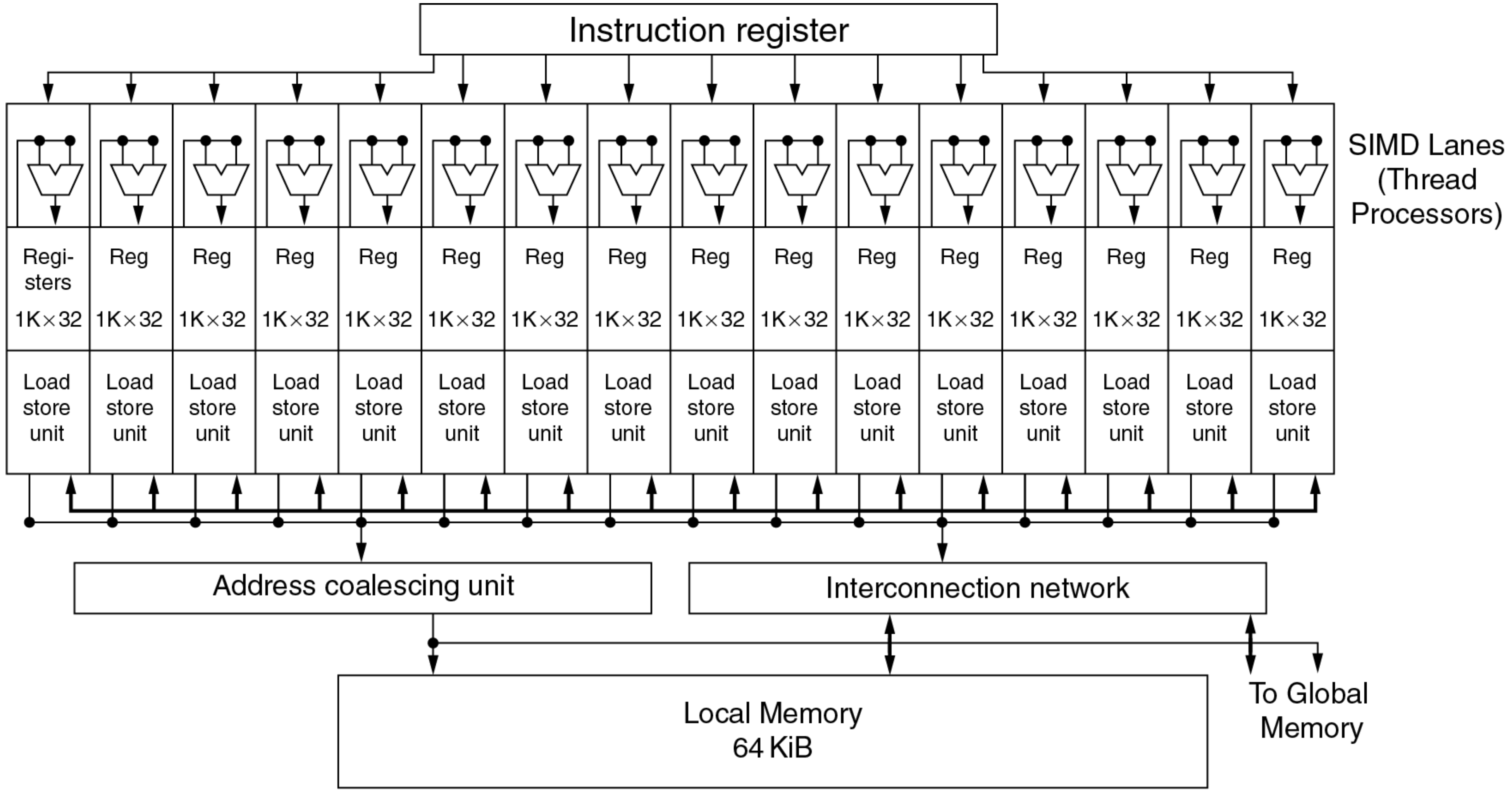
SIMD Example



X+2, Y-1

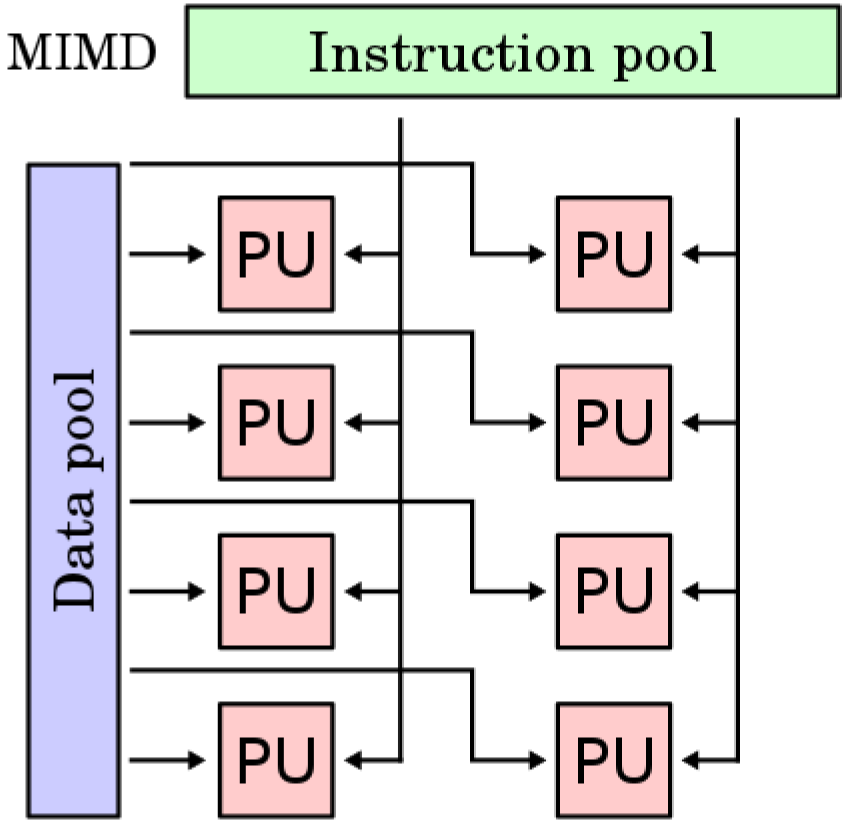
Example: Nvidia Tesla

- Multiple (multithreaded) SIMD processors, each as shown:



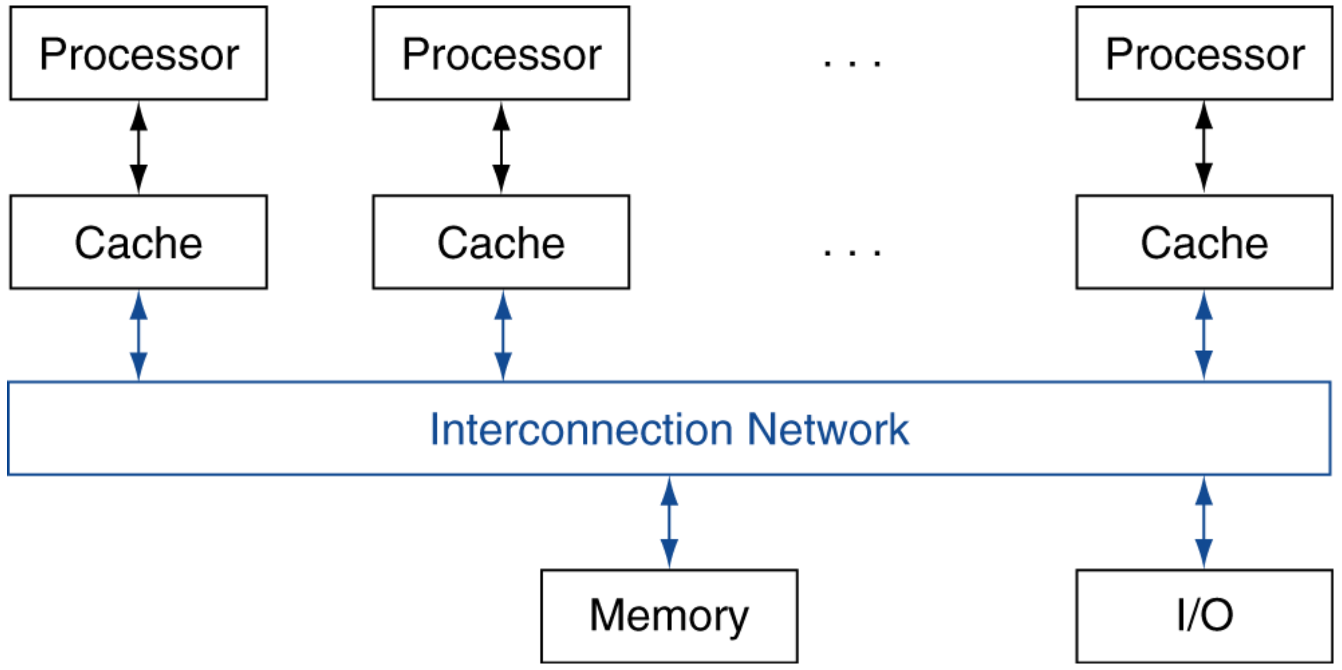
Multi Instruction Multiple Data stream (MIMD)

- Multiple instructions operate on different pieces of data, either *independently* or *as part of shared memory space*
  - Multiprocessor
  - Multithreaded processor
- Uses:
  - Most modern computers
  - Cluster



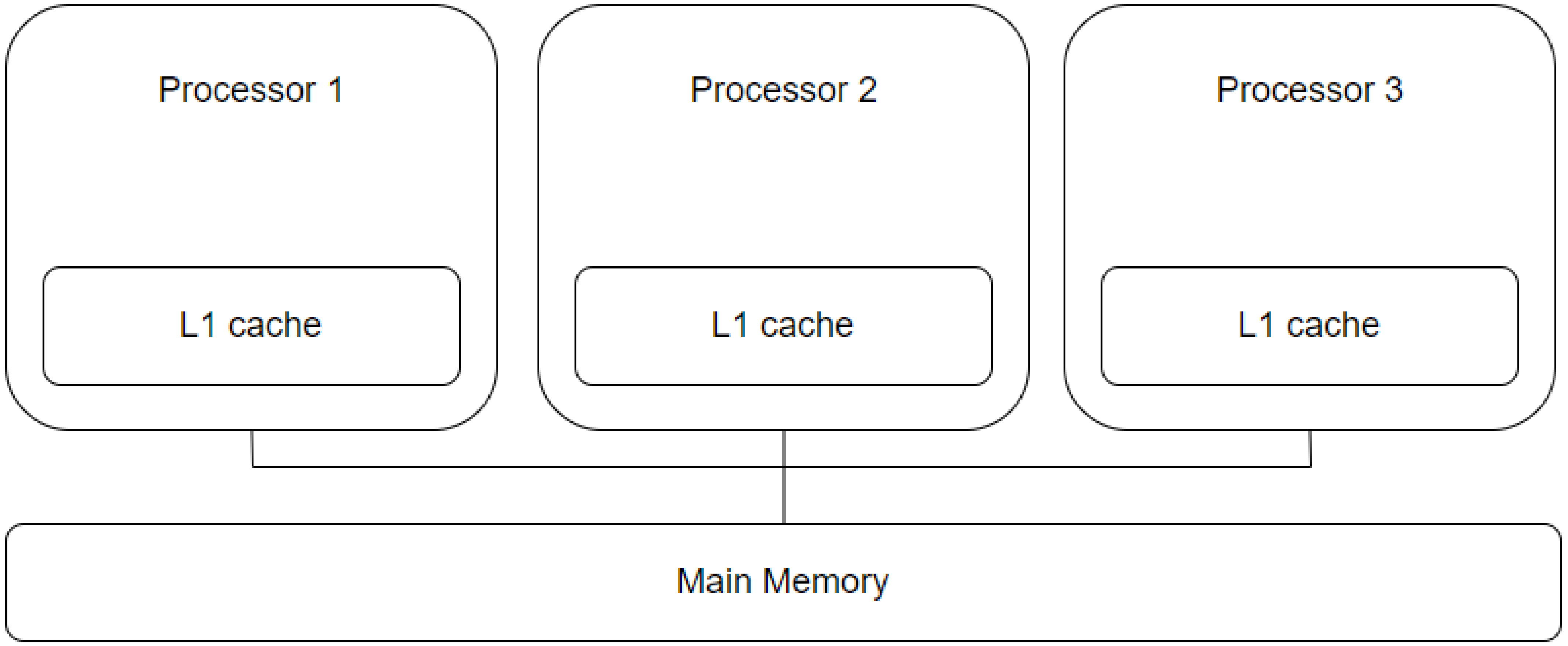
Shared Memory

- **S**hared **M**emory Multi**p**rocessor (SMP)
  - Hardware provides **single physical address space** for all processors
  - Synchronize ***shared variables*** using locks
  - Memory access time
    - UMA (uniform) vs. NUMA (nonuniform)



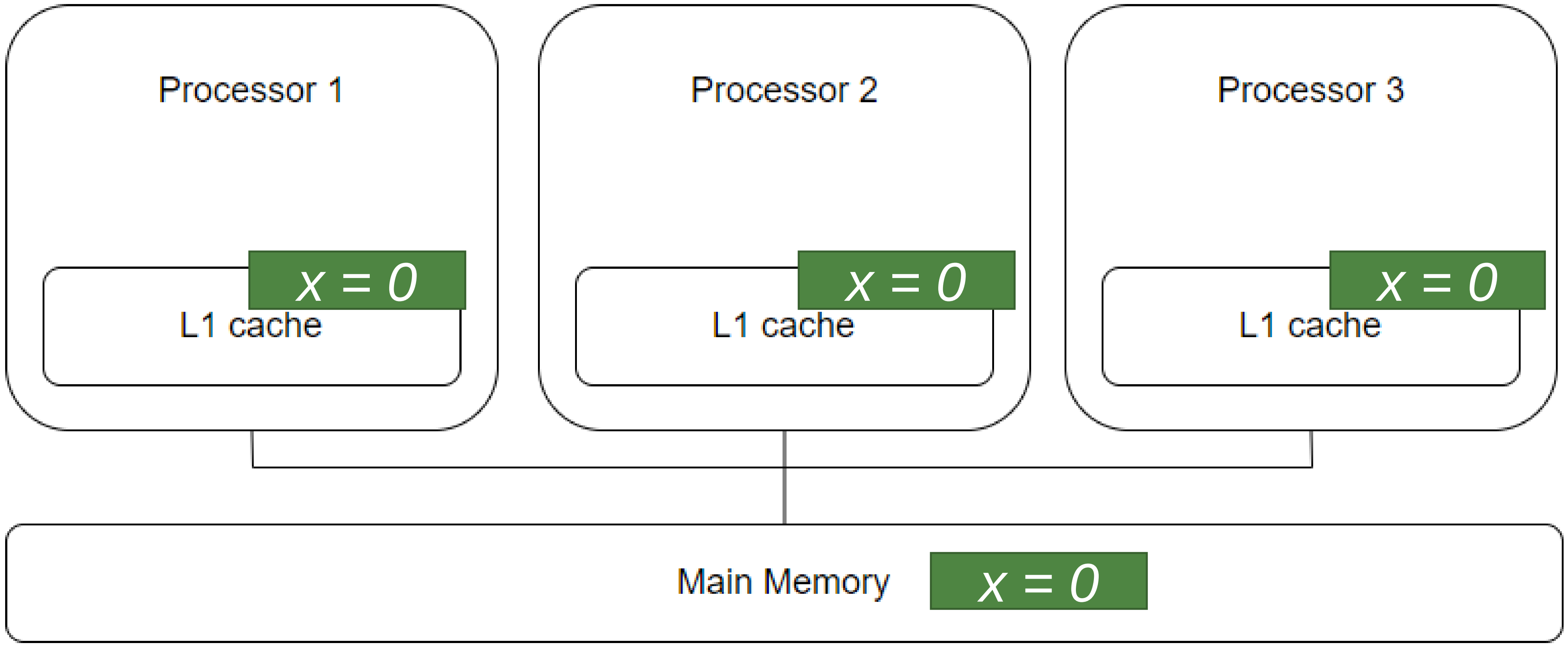
Cache Coherence Problems

- Multiple copies of same data may become inconsistent



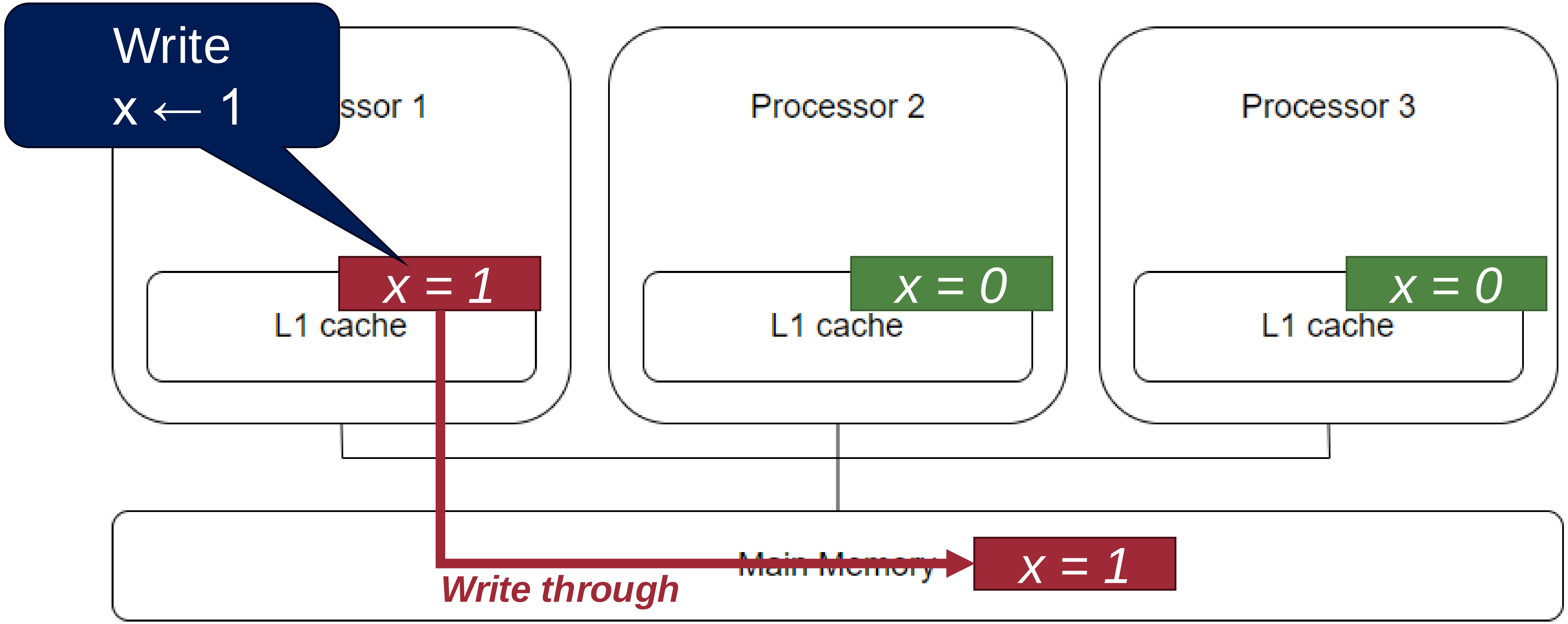
Cache Coherence Problems

- Multiple copies of same data may become inconsistent



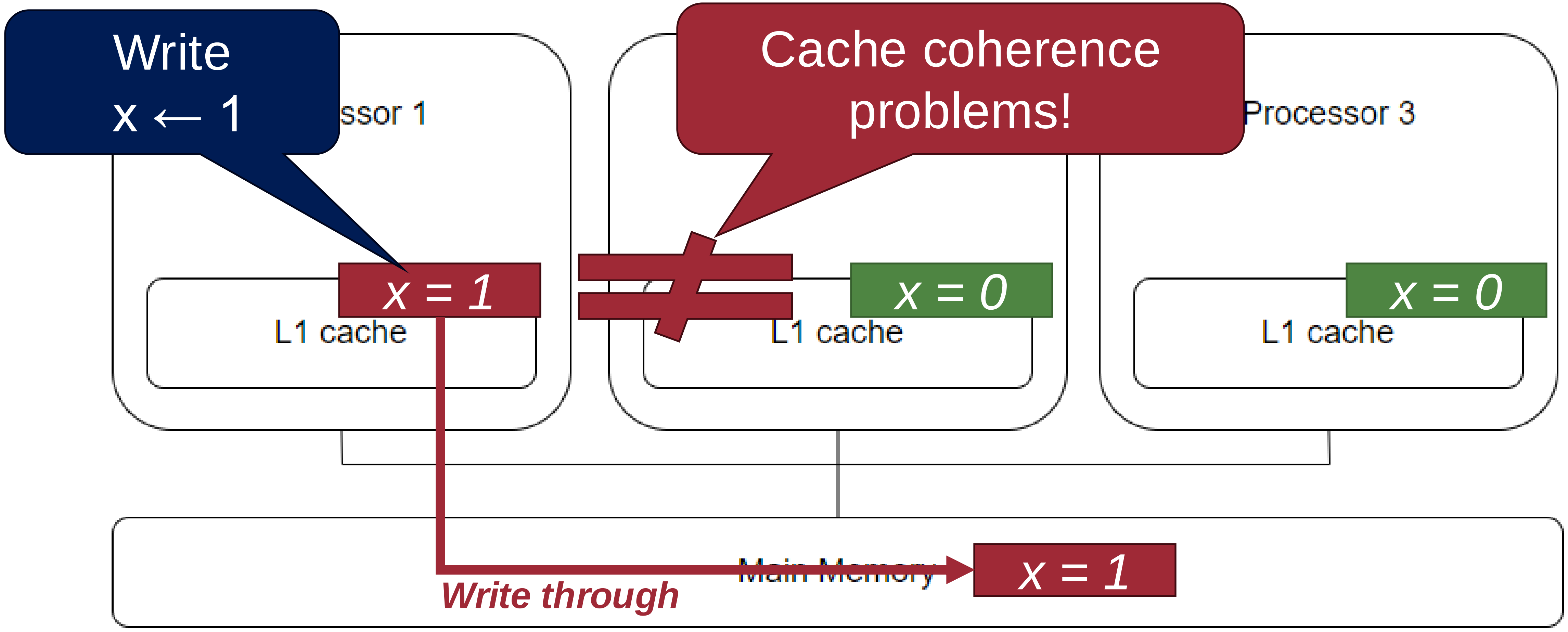
Cache Coherence Problems

- Multiple copies of same data may become inconsistent



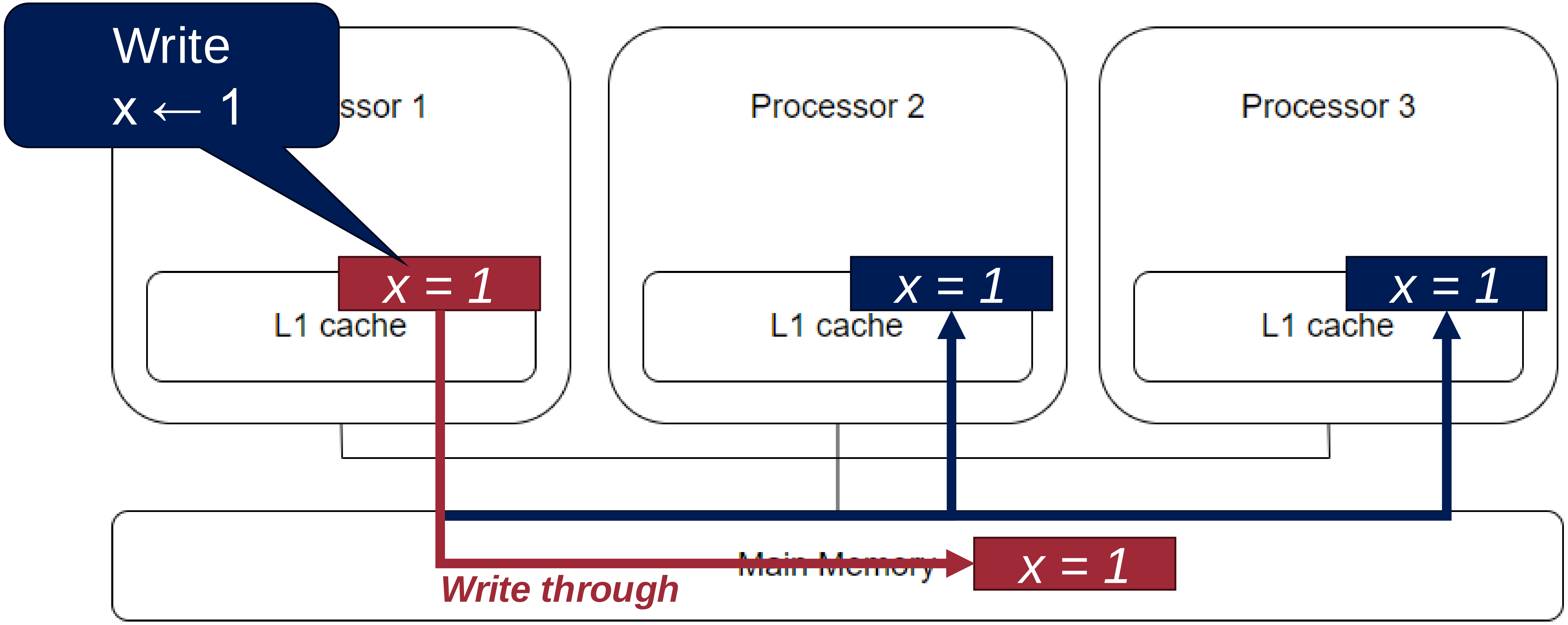
Cache Coherence Problems

- Multiple copies of same data may become inconsistent



Multicore Cache Coherence: Solution

- Use the cache coherence protocol (e.g., MSI, MESI protocol)



Multithreading

- Thread: the unit of execution within a process

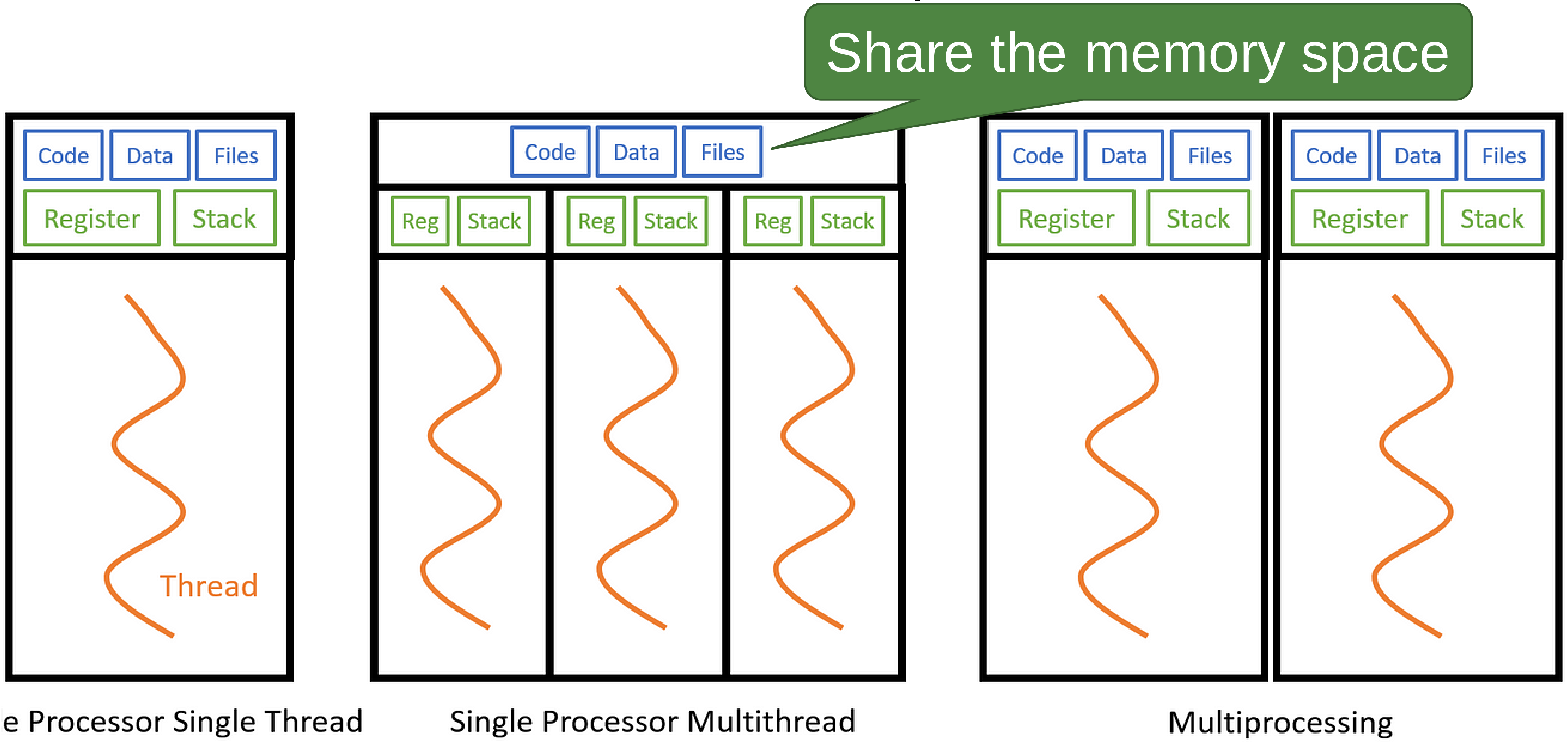


Image from https://towardsdatascience.com/multithreading-and-multiprocessing-in-10-minutes-20d9b3c6a867

Hardware Multithreading

- Run multiple threads of execution in parallel
  - Fast switching between threads
- Fine-grain multithreading
  - Switch threads after each cycle
  - Interleave instruction execution
  - If one thread stalls, others are executed
- Coarse-grain multithreading
  - Only switch on long stall (e.g., L2-cache miss)
  - Simplifies hardware, but doesn't hide short stalls (e.g., data hazards)

Instruction and Data Streams

- According to the Flynn’s taxonomy of computers, …

		Data Streams	
		Single	Multiple
Instruction Streams	Single	**SISD**: Intel Pentium 4	**SIMD**: SSE instructions of x86
	Multiple	**MISD**: No examples today	**MIMD**: Intel Xeon e5345

Question?