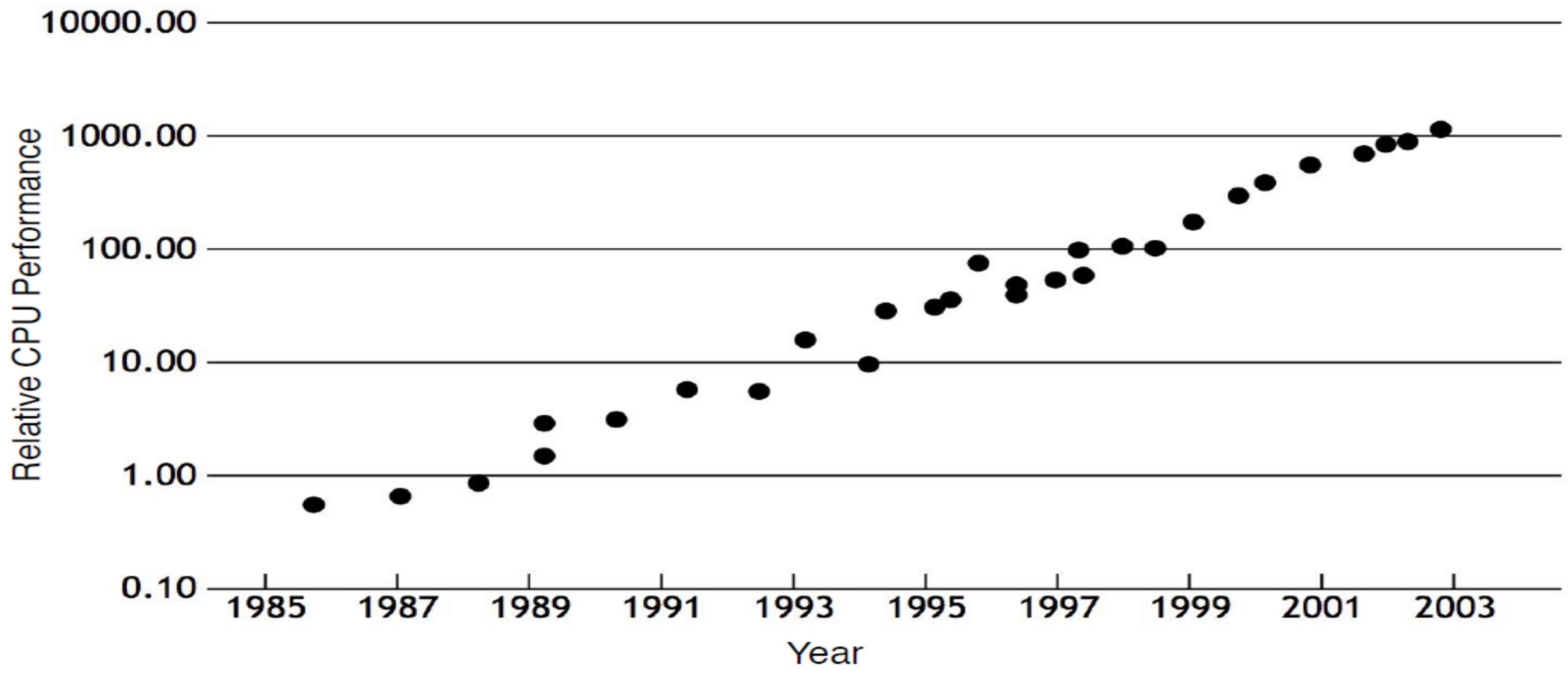
Parallel Computing Fundamentals & Decomposition

Some historical context: why avoid parallel processing?

- **Single-threaded CPU performance doubling ~ every 18 months**
- **Implication: working to parallelize your code was often not worth the time**
  - **Software developer does nothing, code gets faster next year. Woot!**



Until ~15 years ago: two significant reasons for processor performance improvement

1. Exploiting instruction-level parallelism (superscalar execution)

2. Increasing CPU clock frequency

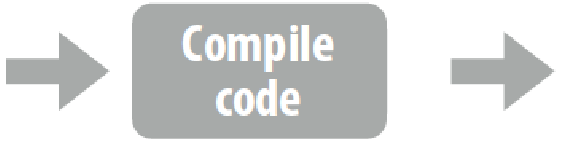
Here is a program written in C

```
int main(int argc, char** argv) {

    int x = 1;

    for (int i=0; i<10; i++) {
       x = x + x;
    }

    printf(“%d\n”, x);

    return 0;
```

What is a program? (from a processor's perspective)

A program is just a list of processor instructions!

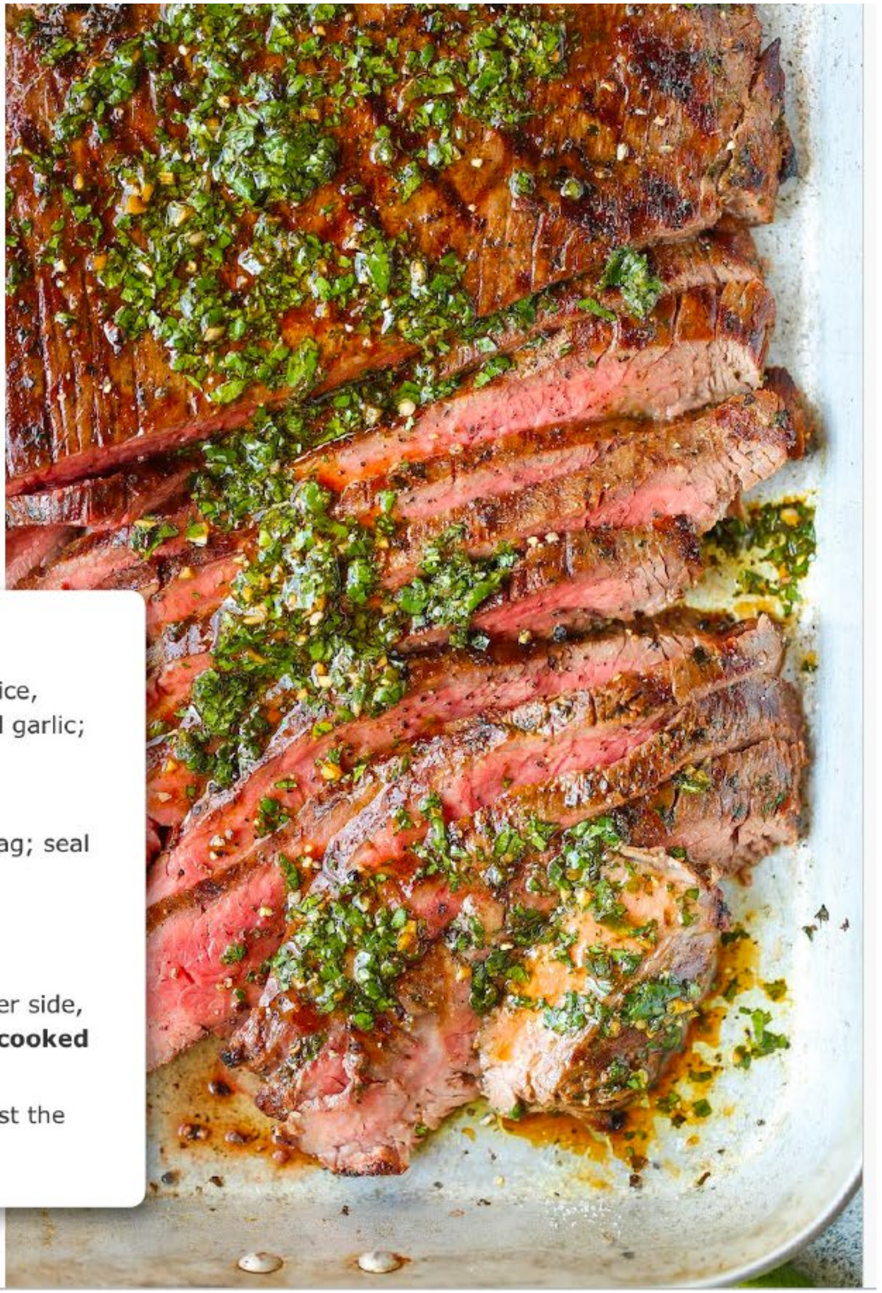
```
int main(int argc, char** argv) {

  int x = 1;

  for (int i=0; i<10; i++) {
    x = x + x;
  }

  printf("%d\n", x);

  return 0;
}
```

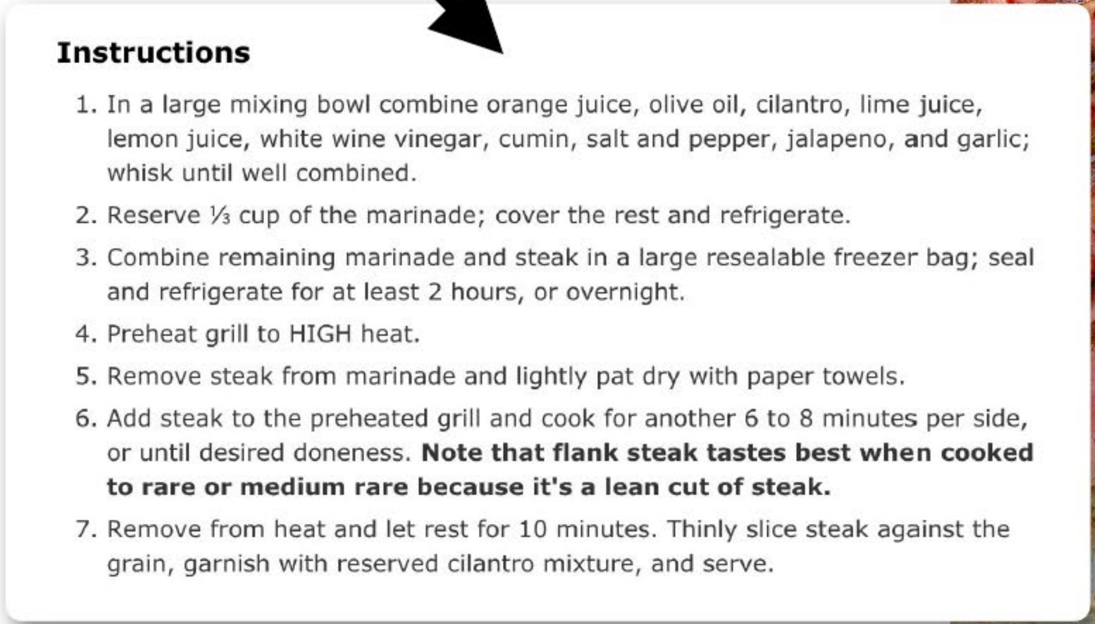


```
_main:
100000f10:    pushq    %rbp
100000f11:    movq %rsp, %rbp
100000f14:    subq $32, %rsp
100000f18:    movl $0, -4(%rbp)
100000f1f:    movl %edi, -8(%rbp)
100000f22:    movq %rsi, -16(%rbp)
100000f26:    movl $1, -20(%rbp)
100000f2d:    movl $0, -24(%rbp)
100000f34:    cmpl $10, -24(%rbp)
100000f38:    jge  23 <_main+0x45>
100000f3e:    movl -20(%rbp), %eax
100000f41:    addl -20(%rbp), %eax
100000f44:    movl %eax, -20(%rbp)
100000f47:    movl -24(%rbp), %eax
100000f4a:    addl $1, %eax
100000f4d:    movl %eax, -24(%rbp)
100000f50:    jmp  -33 <_main+0x24>
100000f55:    leaq 58(%rip), %rdi
100000f5c:    movl -20(%rbp), %esi
100000f5f:    movb $0, %al
100000f61:    callq    14
100000f66:    xorl %esi, %esi
100000f68:    movl %eax, -28(%rbp)
100000f6b:    movl %esi, %eax
```

Kind of like the instructions in a recipe for your favorite meals

Mmm, grilled meat



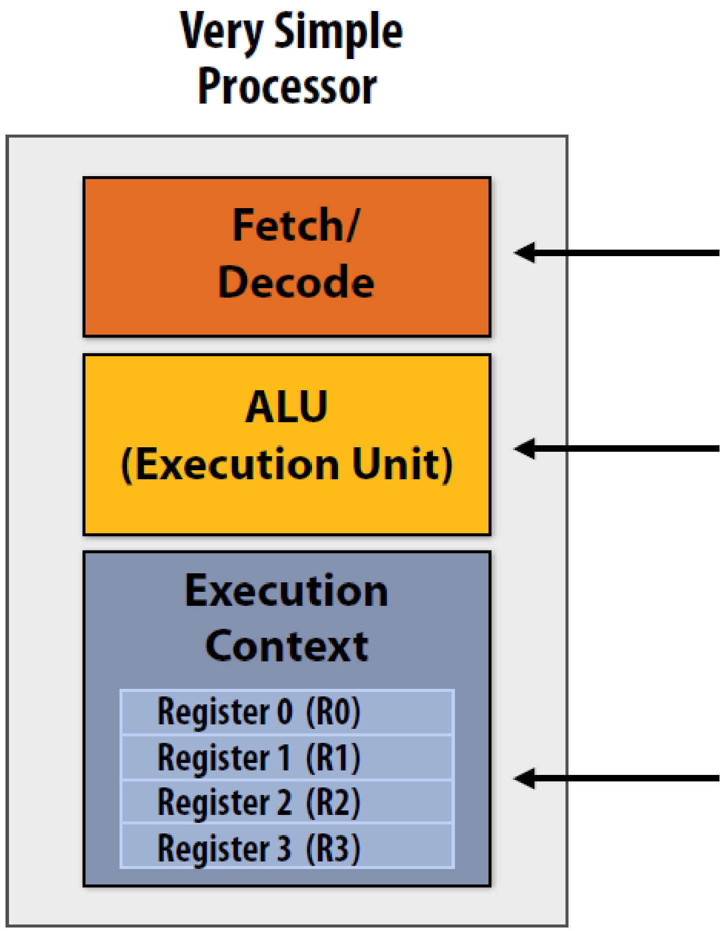


Instructions

1. In a large mixing bowl combine orange juice, olive oil, cilantro, lime juice, lemon juice, white wine vinegar, cumin, salt and pepper, jalapeno, and garlic; whisk until well combined.
2. Reserve ⅓ cup of the marinade; cover the rest and refrigerate.
3. Combine remaining marinade and steak in a large resealable freezer bag; seal and refrigerate for at least 2 hours, or overnight.
4. Preheat grill to HIGH heat.
5. Remove steak from marinade and lightly pat dry with paper towels.
6. Add steak to the preheated grill and cook for another 6 to 8 minutes per side, or until desired doneness. **Note that flank steak tastes best when cooked to rare or medium rare because it's a lean cut of steak.**
7. Remove from heat and let rest for 10 minutes. Thinly slice steak against the grain, garnish with reserved cilantro mixture, and serve.

What does a processor do?

A processor executes instructions



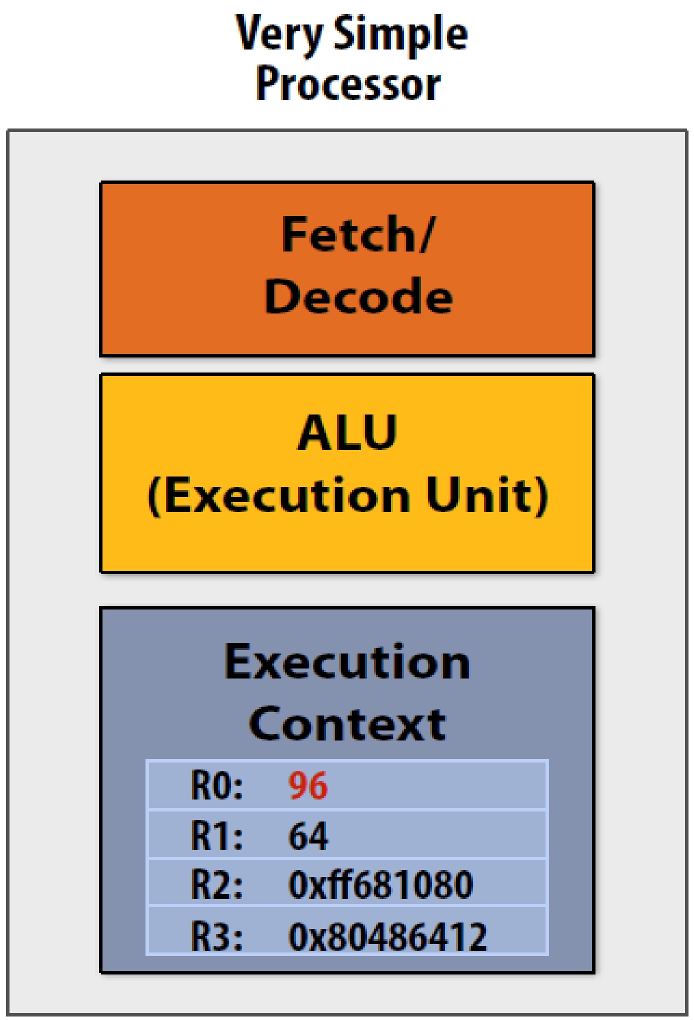
Determine what instruction to run next

Execution unit: performs the operation described by an instruction, which may modify values in the processor's registers or the computer's memory

Registers: maintain program state: store value of variables used as inputs and outputs to operations

TAYLOR'S UNIVERSITY
Wisdom · Integrity · Excellence

One example instruction: add two numbers



Step 1:
Processor gets next program instruction from memory (figure out what the processor should do next)

`add R0 ← R0, R1`

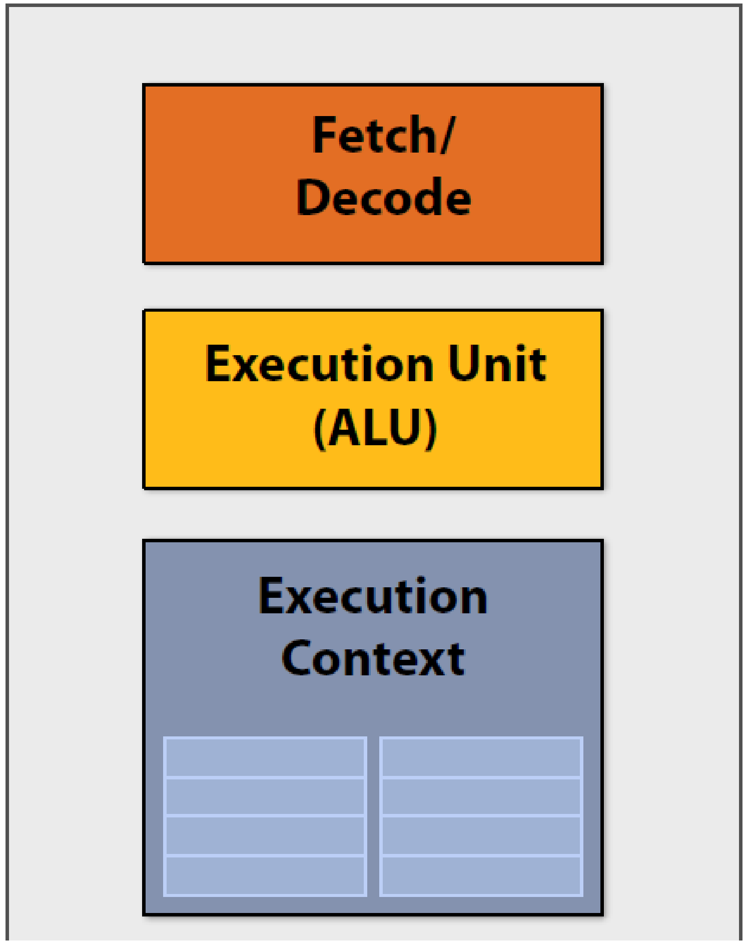
"Please add the contents of register R0 to the contents of register R1 and put the result of the addition into register R0"

Step 2:
Get operation inputs from registers
Contents of R0 input to execution unit: 32
Contents of R1 input to execution unit: 64

Step 3:
Perform addition operation:
Execution unit performs arithmetic, the result is: 96

Execute program

My very simple processor: executes one instruction per clock



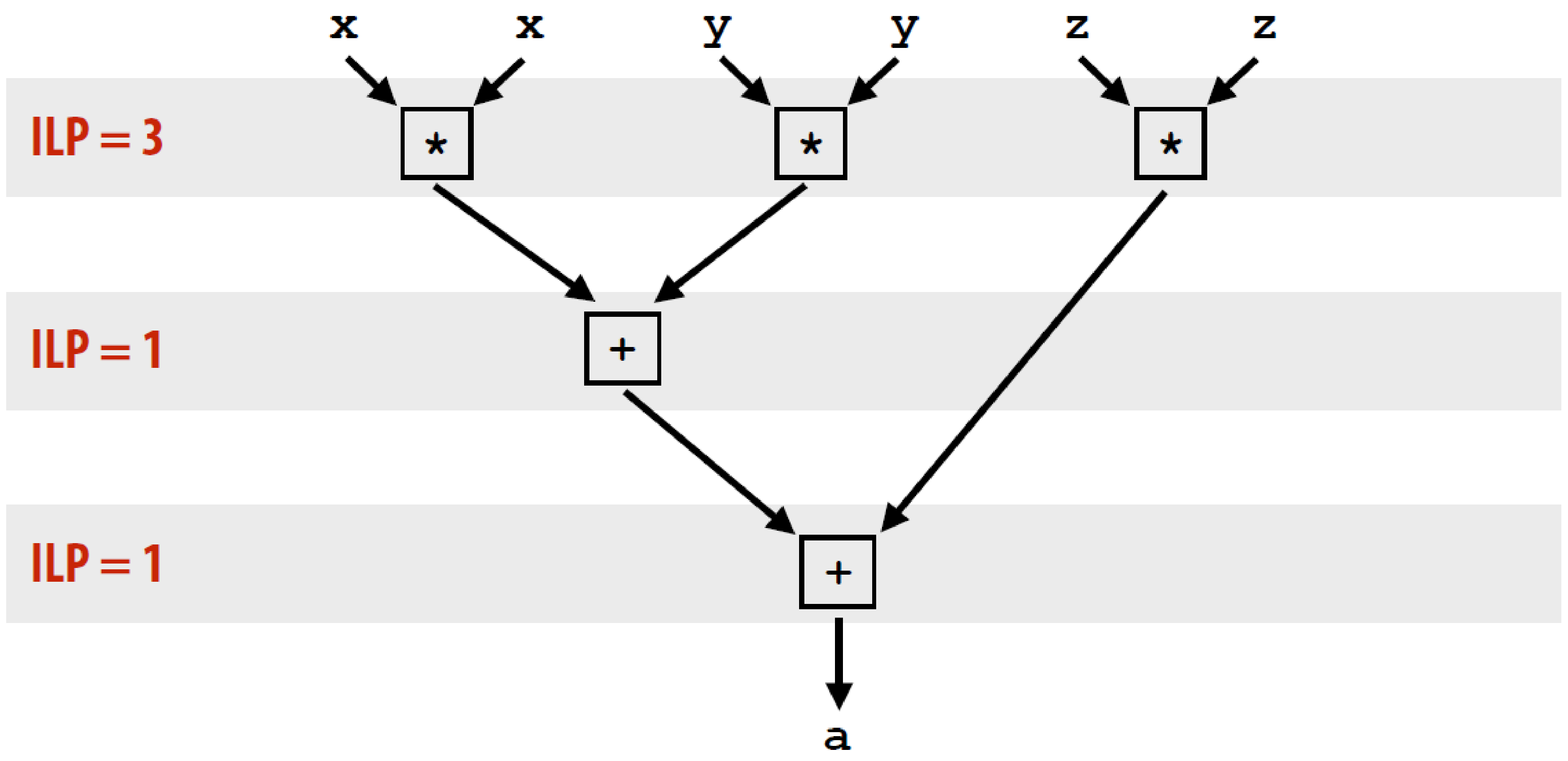
```
ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...
...
...
...
...
...
st   addr[r2], r0
```

TAYLOR'S UNIVERSITY
Wisdom · Integrity · Excellence

Instruction level parallelism (ILP) example

- ILP = 3

```
a = x*x + y*y + z*z
```



Superscalar processor execution

```
a = x*x + y*y + z*z
```

```
Assume register
R0 = x, R1 = y, R2 = z
```

```
1 mul R0, R0, R0
2 mul R1, R1, R1
3 mul R2, R2, R2
4 add R0, R0, R1
5 add R3, R0, R2
```

Idea #1:

Superscalar execution: processor automatically finds* independent instructions in an instruction sequence and executes them in parallel on multiple execution units!

In this example: instructions 1, 2, and 3 can be executed in parallel without impacting program correctness (on a superscalar processor that determines that the lack of dependencies exists)

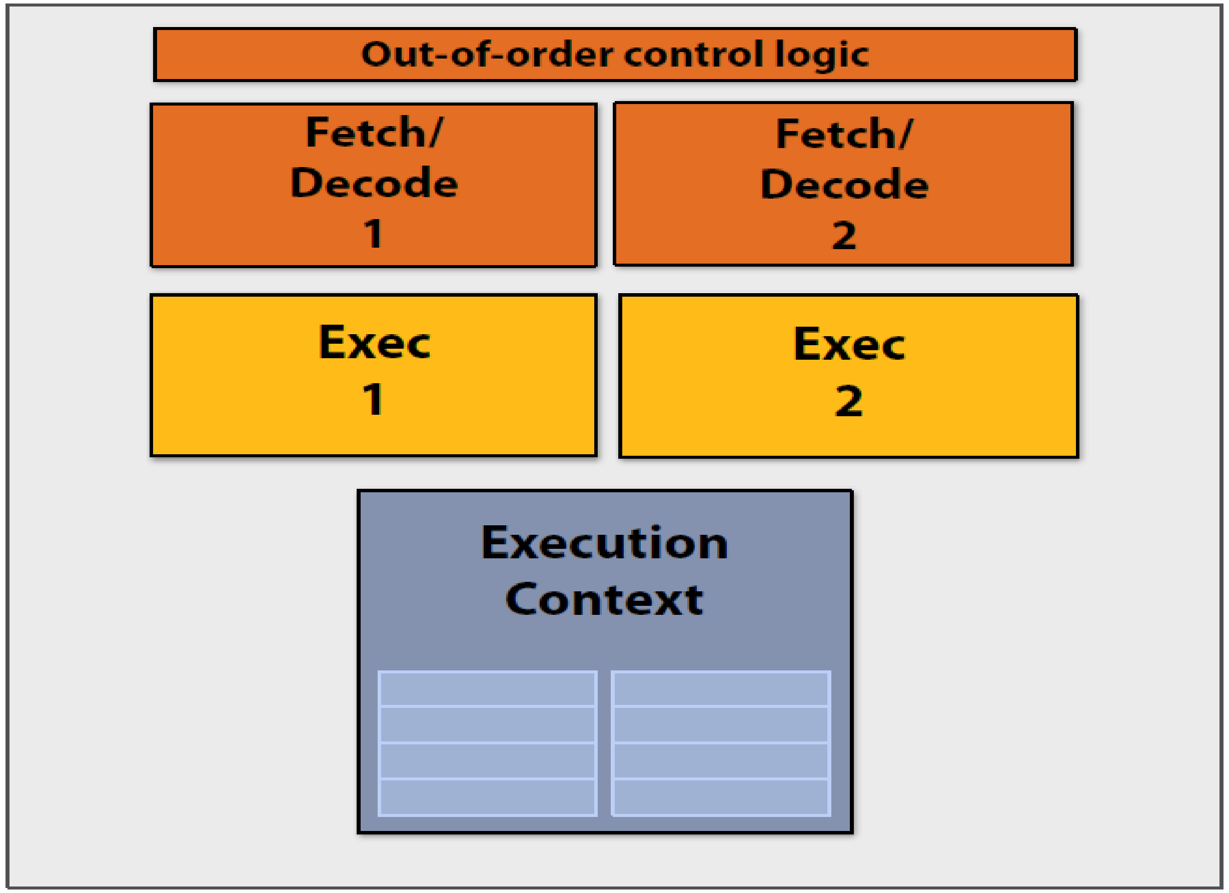
But instruction 4 must be executed after instructions 1 and 2

And instruction 5 must be executed after instruction 4

* Or the compiler finds independent instructions at compile time and explicitly encodes dependencies in the compiled binary.

Superscalar processor

This processor can decode and execute up to two instructions per clock



Aside:
Old Intel Pentium 4 CPU

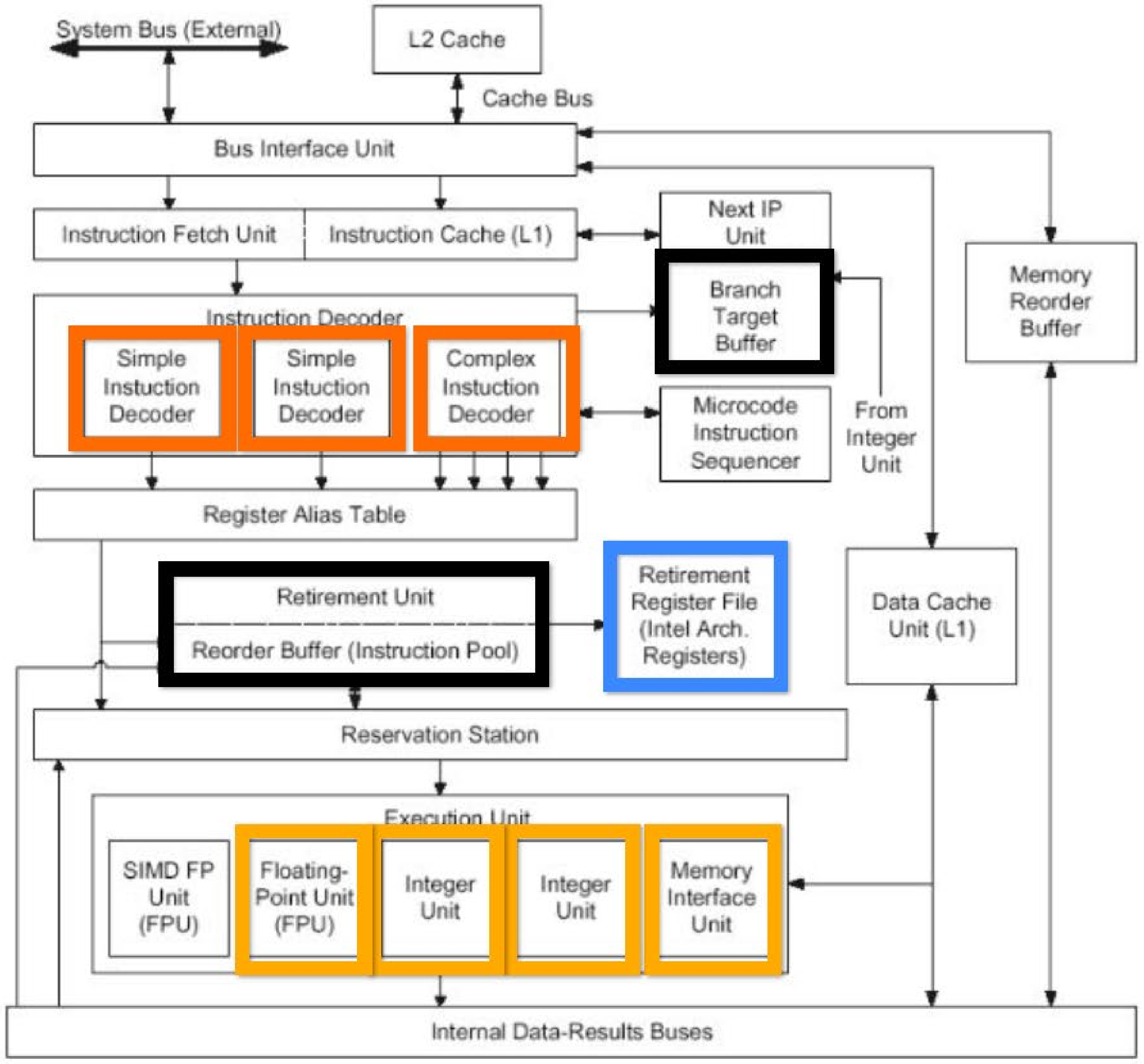
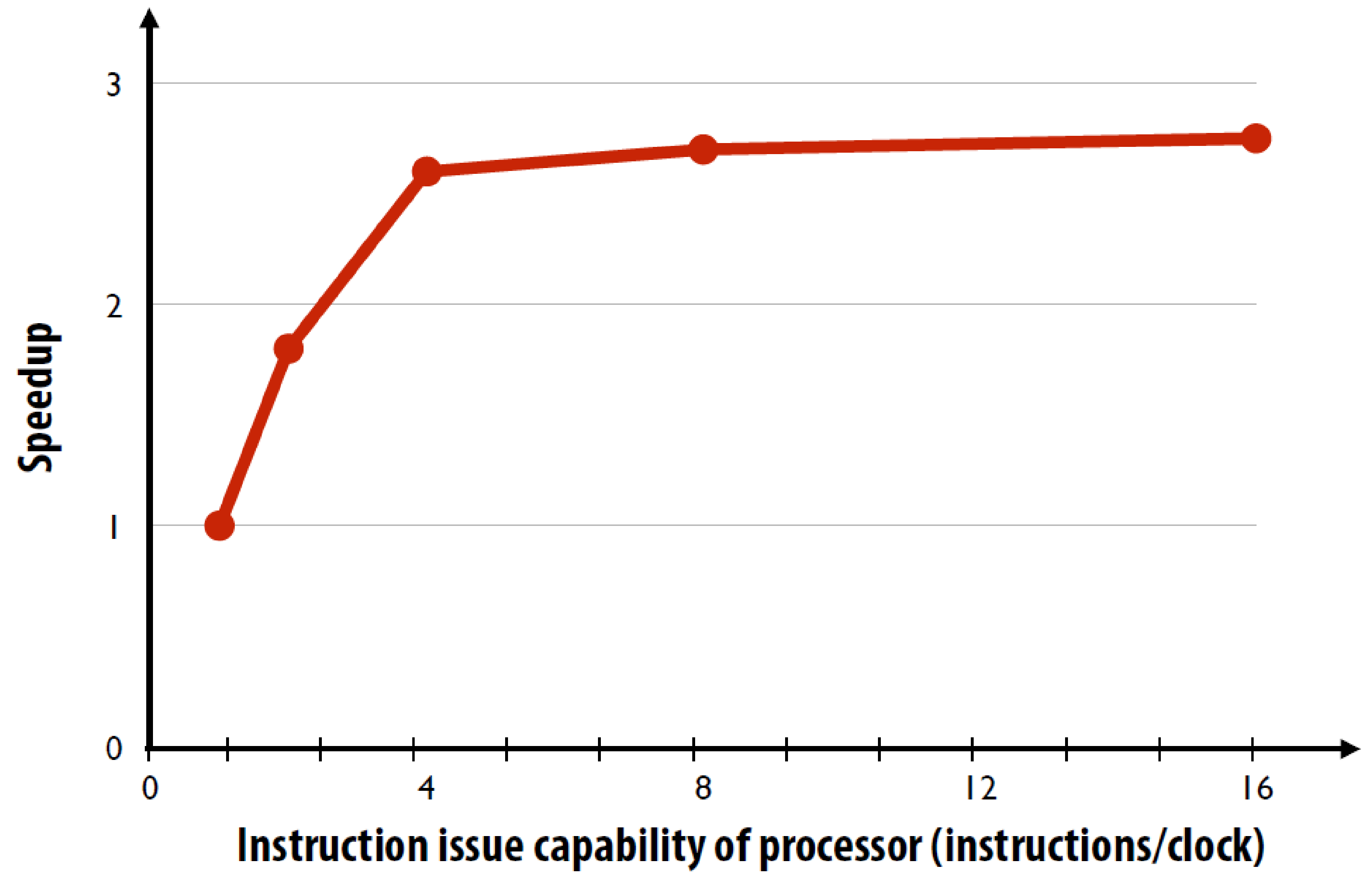


Image credit: http://ixbtlabs.com/articles/pentium4/index.html

Diminishing returns of superscalar execution

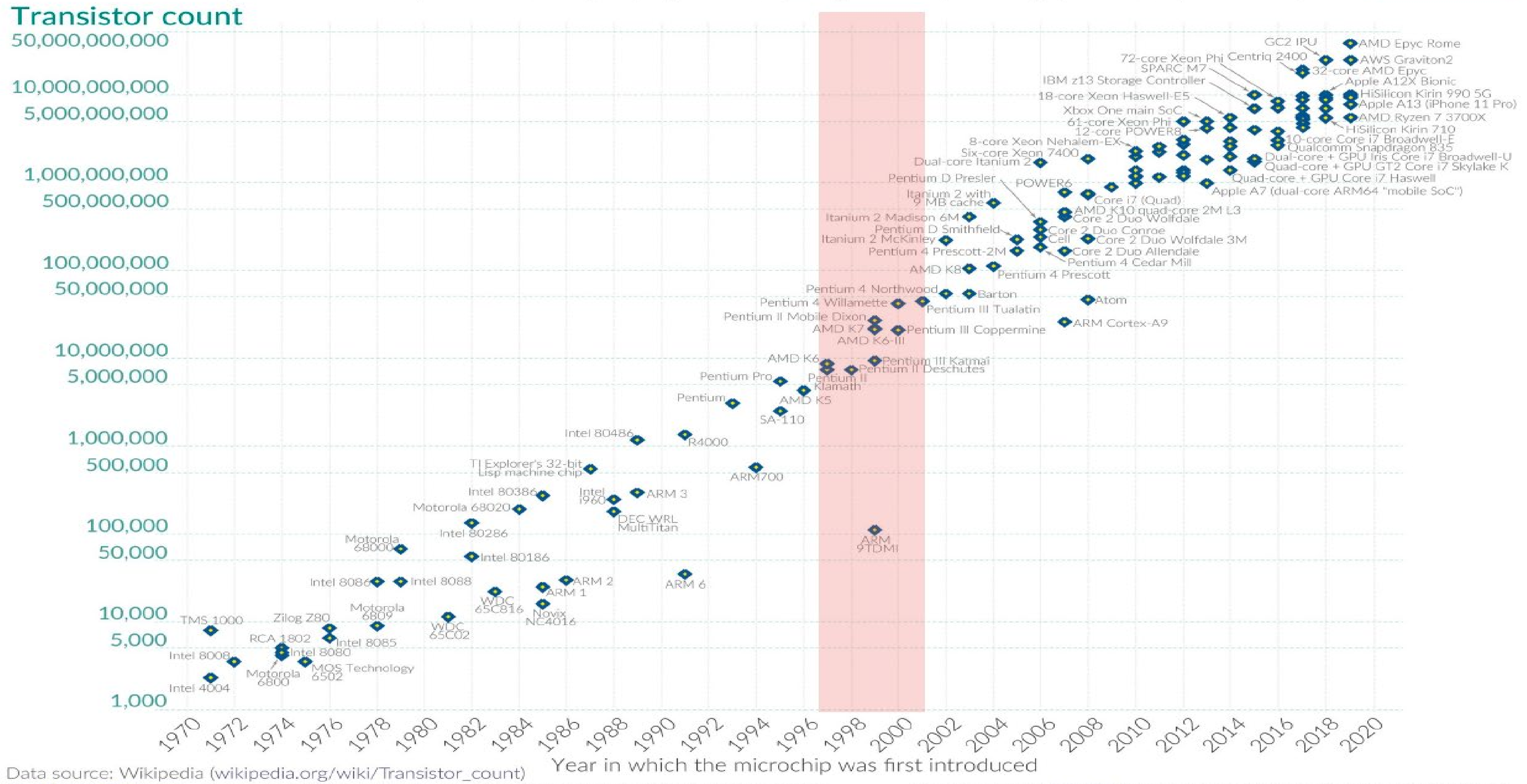
Most available ILP is exploited by a processor capable of issuing four instructions per clock
(Little performance benefit from building a processor that can issue more)



Source: Culler & Singh (data from Johnson 1991)

Moore's Law: The number of transistors on microchips doubles every two years

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.



Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)
OurWorldinData.org – Research and data to make progress against the world's largest problems.

Types of Parallelism

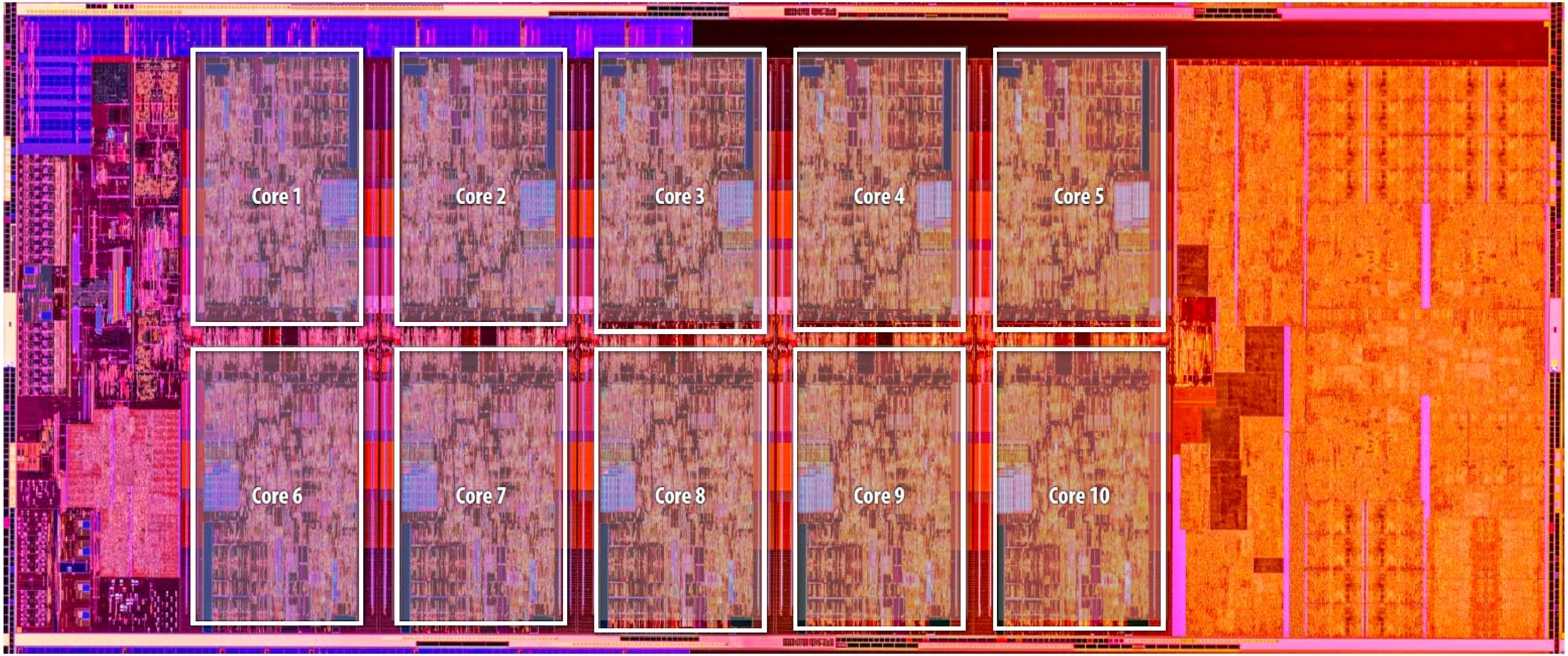
Instruction-Level Parallelism (ILP): executing multiple **machine-level instructions** simultaneously within a **single processor core**. Achieved through hardware features like **pipelining**, **superscalar execution**, and **out-of-order execution**.

Task Parallelism: Running independent threads/tasks concurrently on multiple cores or processors. Example: Sorting different parts of an array in parallel.

Data Parallelism: Performing the same operation on different pieces of data simultaneously. Example: Applying a filter to pixels in an image concurrently.

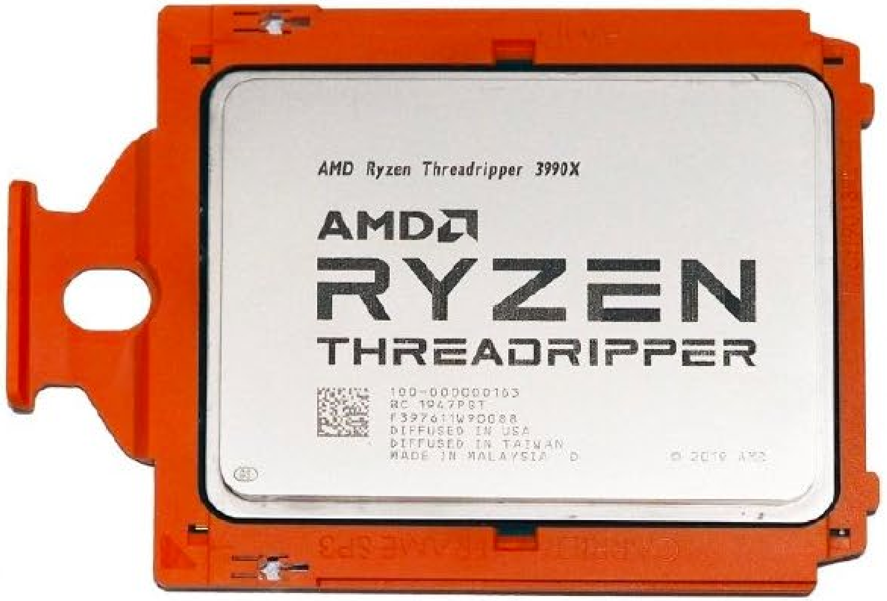
Example: multi-core CPU

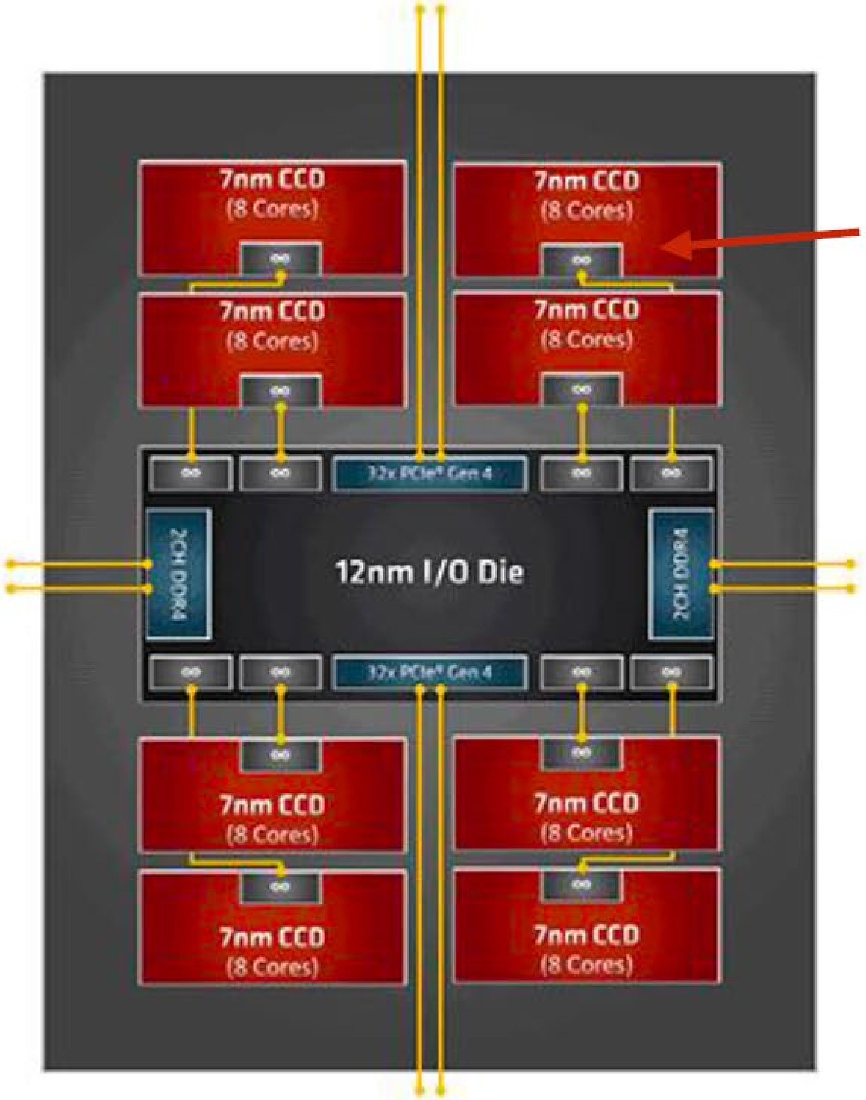
Intel "Comet Lake" 10th Generation Core i9 10-core CPU (2020)



AMD Ryzen Threadripper 3990X

64 cores, 4.3 GHz





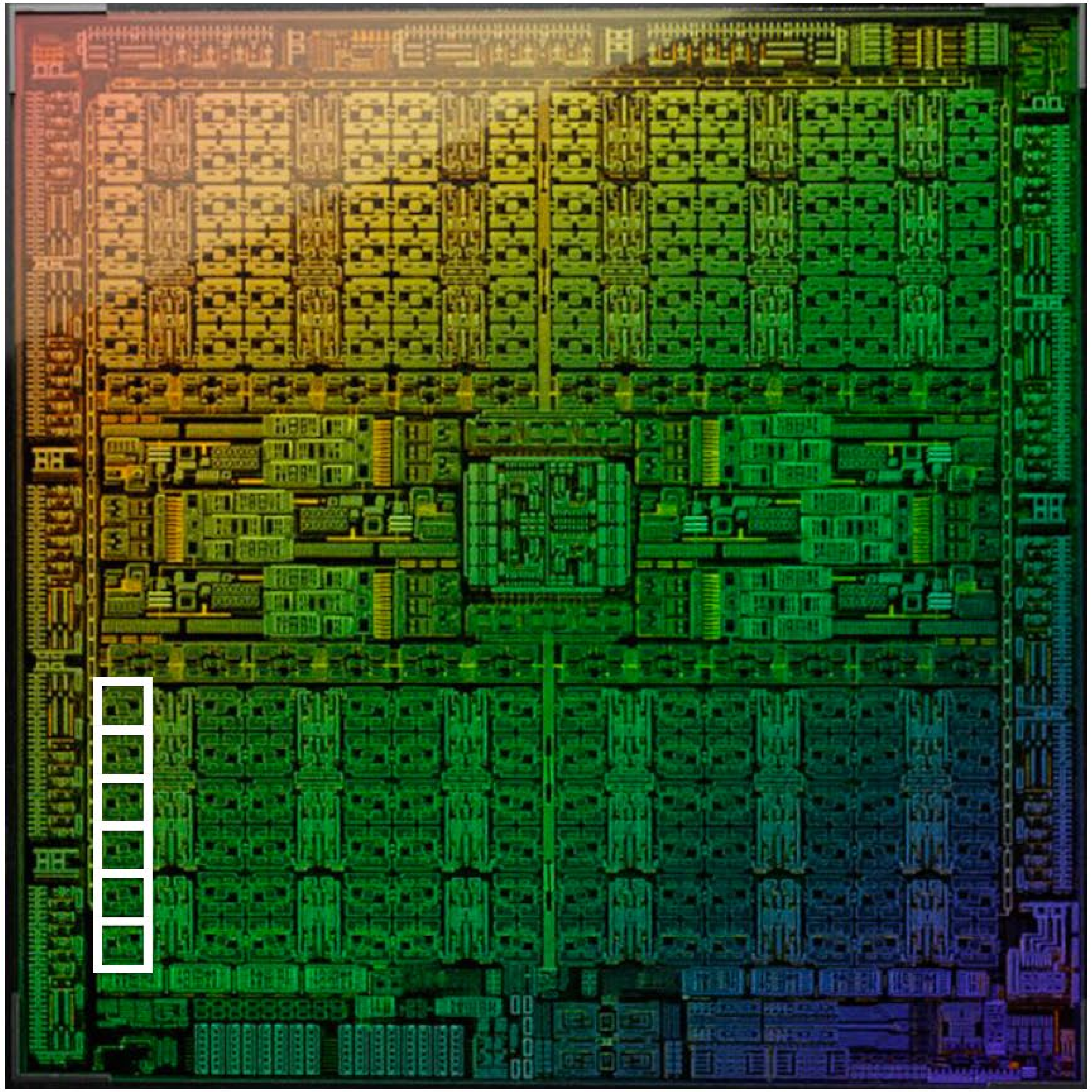
Four 8-core chiplets

NVIDIA AD102 GPU

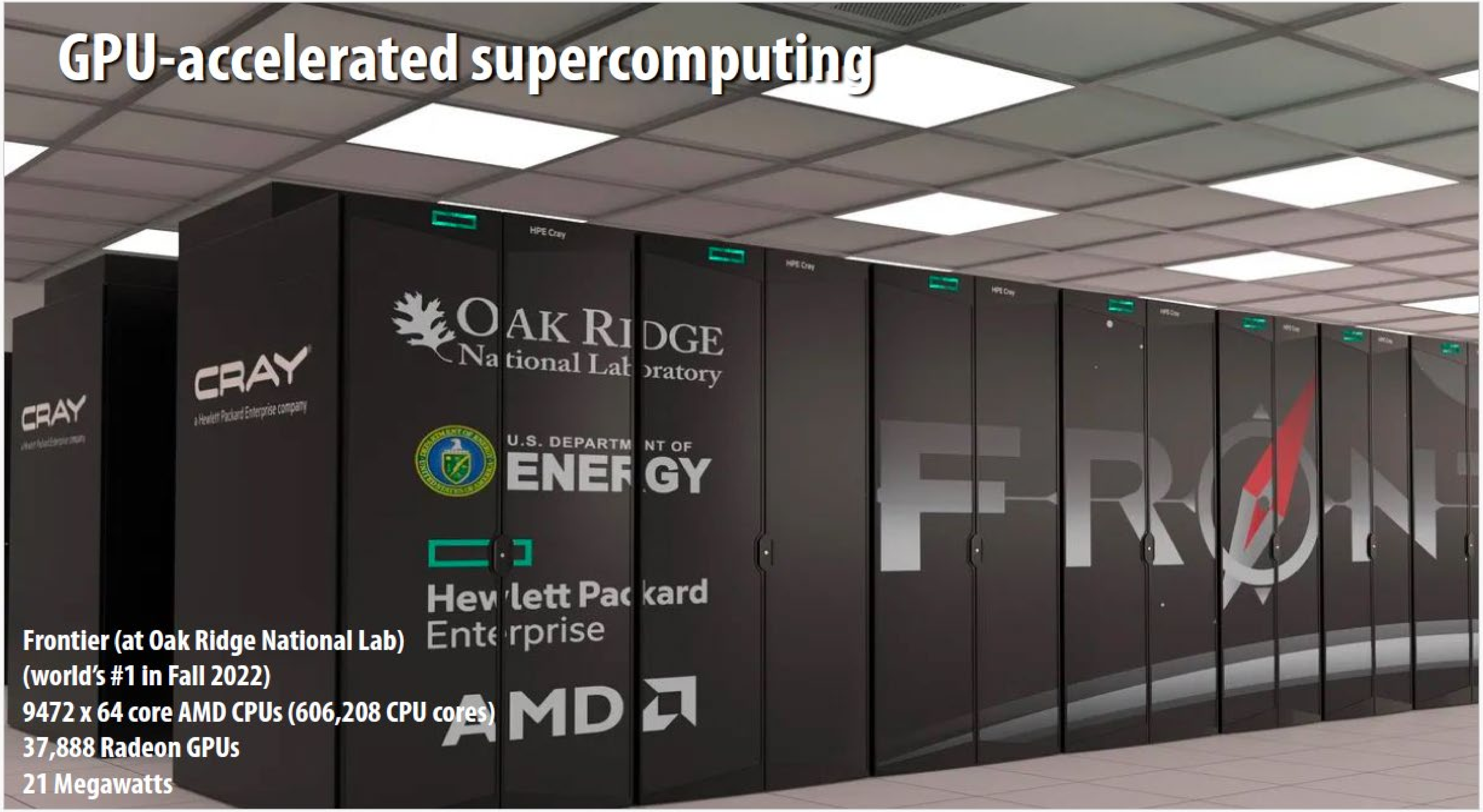
GeForce RTX 4090 (2022)

76 billion transistors

18,432 fp32 multipliers organized in 144 processing blocks (called SMs)

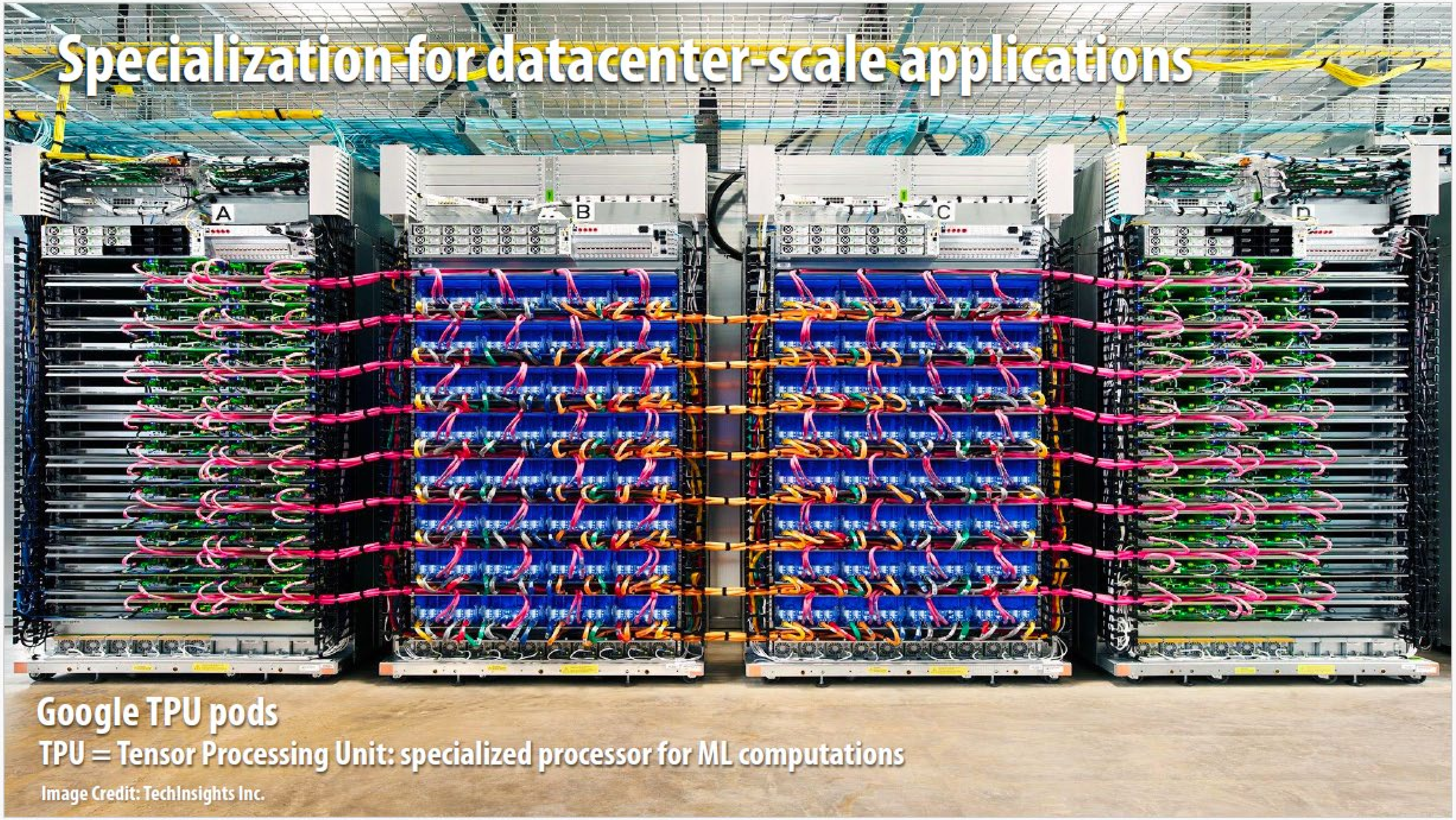


GPU-accelerated supercomputing


CRAY
a Hewlett Packard Enterprise company
CRAY
Oak Ridge National Laboratory
U.S. DEPARTMENT OF ENERGY
Hewlett Packard Enterprise
AMD
HPE Cray
FRONTIER

Frontier (at Oak Ridge National Lab)
(world's #1 in Fall 2022)
9472 x 64 core AMD CPUs (606,208 CPU cores)
37,888 Radeon GPUs
21 Megawatts

Specialization for datacenter-scale applications



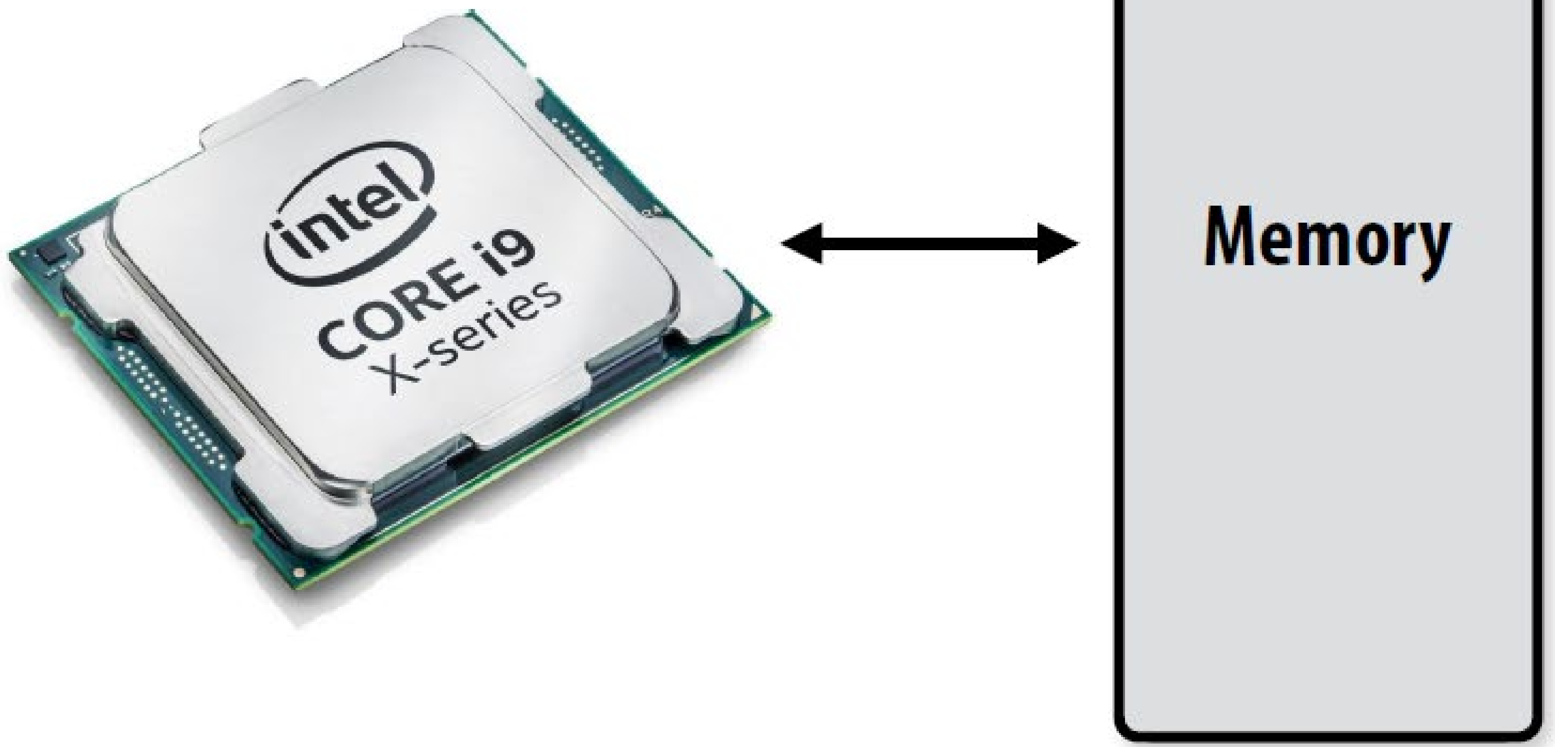
Google TPU pods

TPU = Tensor Processing Unit: specialized processor for ML computations

Image Credit: TechInsights Inc.

Achieving efficient processing almost always comes down to accessing data efficiently.

What is memory?



TAYLOR'S UNIVERSITY
Wisdom · Integrity · Excellence

A program's memory address space

- A computer's memory is organized as an array of bytes
- Each byte is identified by its "address" in memory (its position in this array)

  (We'll assume memory is byte-addressable)

"The byte stored at address 0x8 has the value 32."

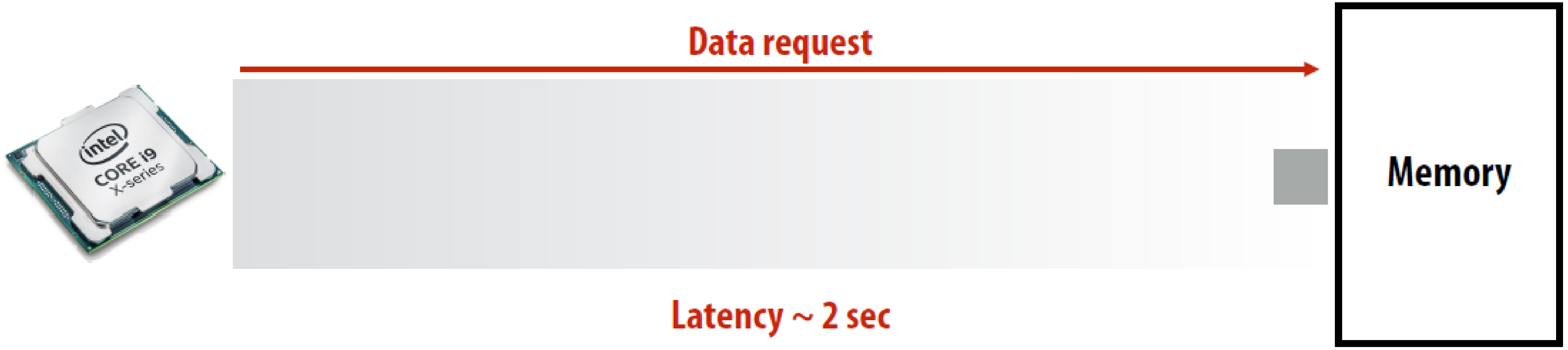
"The byte stored at address 0x10 (16) has the value 128."

In the illustration on the right, the program's memory address space is 32 bytes in size (so valid addresses range from 0x0 to 0x1F)

Address	Value
0x0	16
0x1	255
0x2	14
0x3	0
0x4	0
0x5	0
0x6	6
0x7	0
0x8	32
0x9	48
0xA	255
0xB	255
0xC	255
0xD	0
0xE	0
0xF	0
0x10	128
⋮	⋮
0x1F	0

Terminology

- **Memory access latency**
  - The amount of time it takes the memory system to provide data to the processor
  - Example: 100 clock cycles, 100 nsec



Stalls

- A processor "stalls" (can't make progress) when it cannot run the next instruction in an instruction stream because future instructions depend on a previous instruction that is not yet complete.

- Accessing memory is a major source of stalls

```
ld r0 mem[r2]
ld r1 mem[r3]
add r0, r0, r1
```

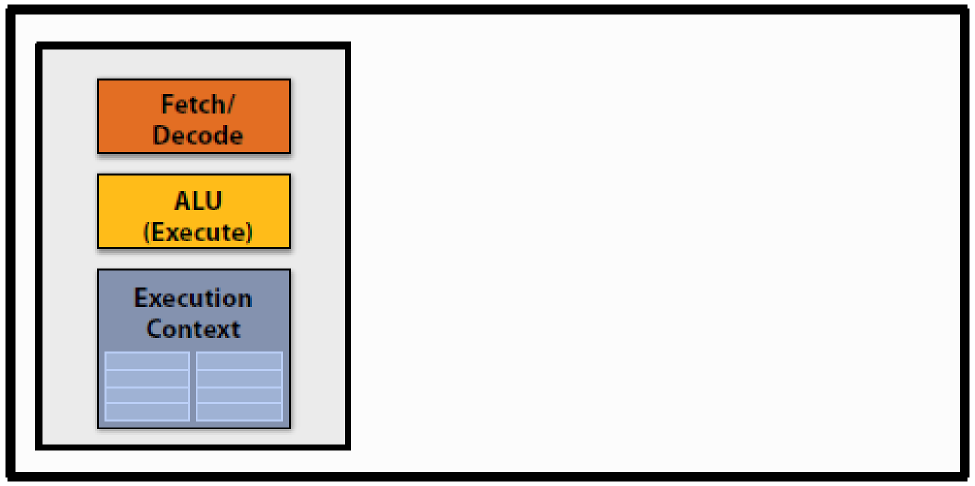
Dependency: cannot execute 'add' instruction until data from mem[r2] and mem[r3] have been loaded from memory

- Memory access times ~ 100's of cycles
  - Memory "access time" is a measure of latency

What are caches?

- Recall memory is just an array of values
- And a processor has instructions for moving data from memory into registers (load) and storing data from registers into memory (store)

Processor

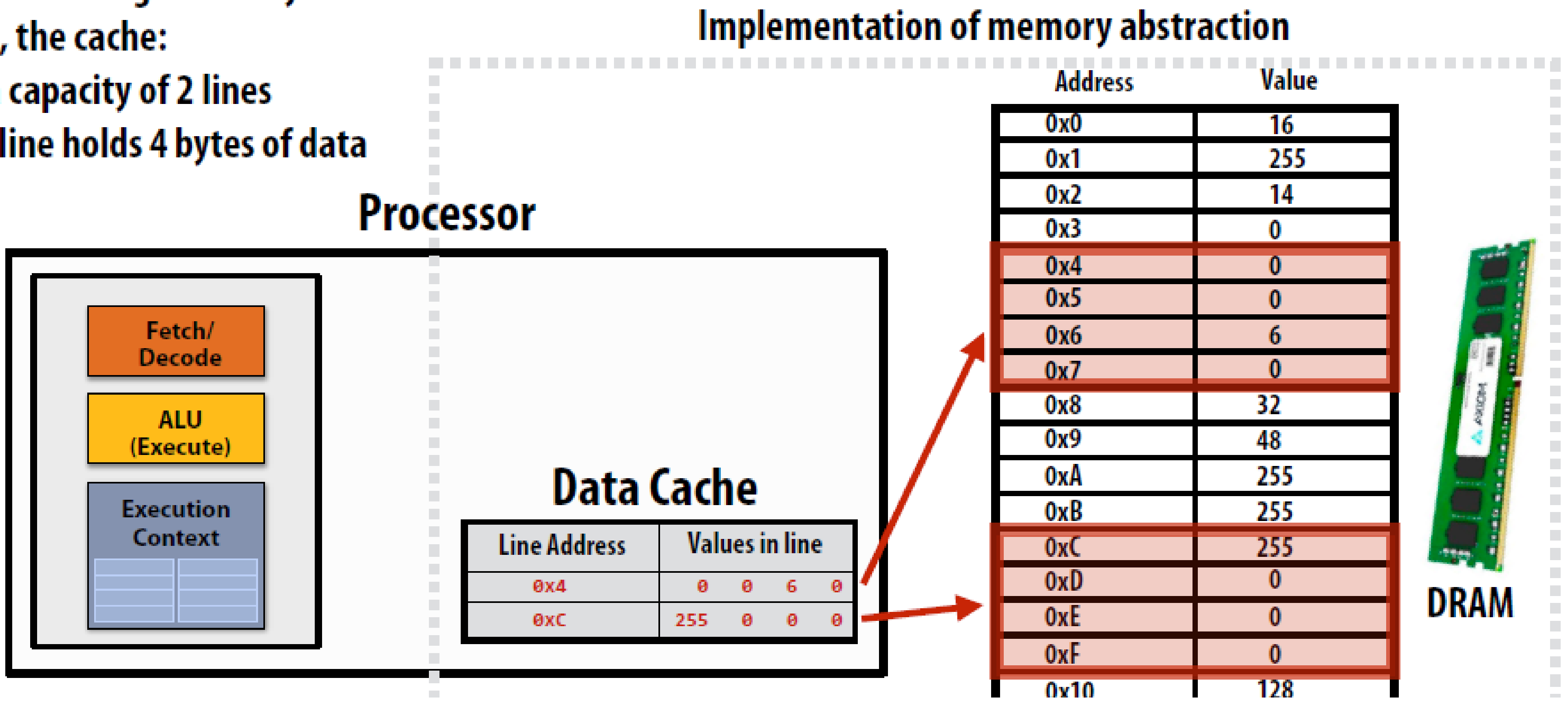


Memory

Address	Value
0x0	16
0x1	255
0x2	14
0x3	0
0x4	0
0x5	0
0x6	6
0x7	0
0x8	32
0x9	48
0xA	255
0xB	255
0xC	255
0xD	0
0xE	0
0xF	0
0x10	128
⋮	⋮
0x1F	0

What are caches?

- A cache is a hardware implementation detail that does not impact the output of a program, only its performance
- Cache is on-chip storage that maintains a copy of a subset of the values in memory
- If an address is stored "in the cache" the processor can load/store to this address more quickly than if the data resides only in DRAM
- Caches operate at the granularity of "cache lines".
  In the figure, the cache:
  - Has a capacity of 2 lines
  - Each line holds 4 bytes of data



Data Cache

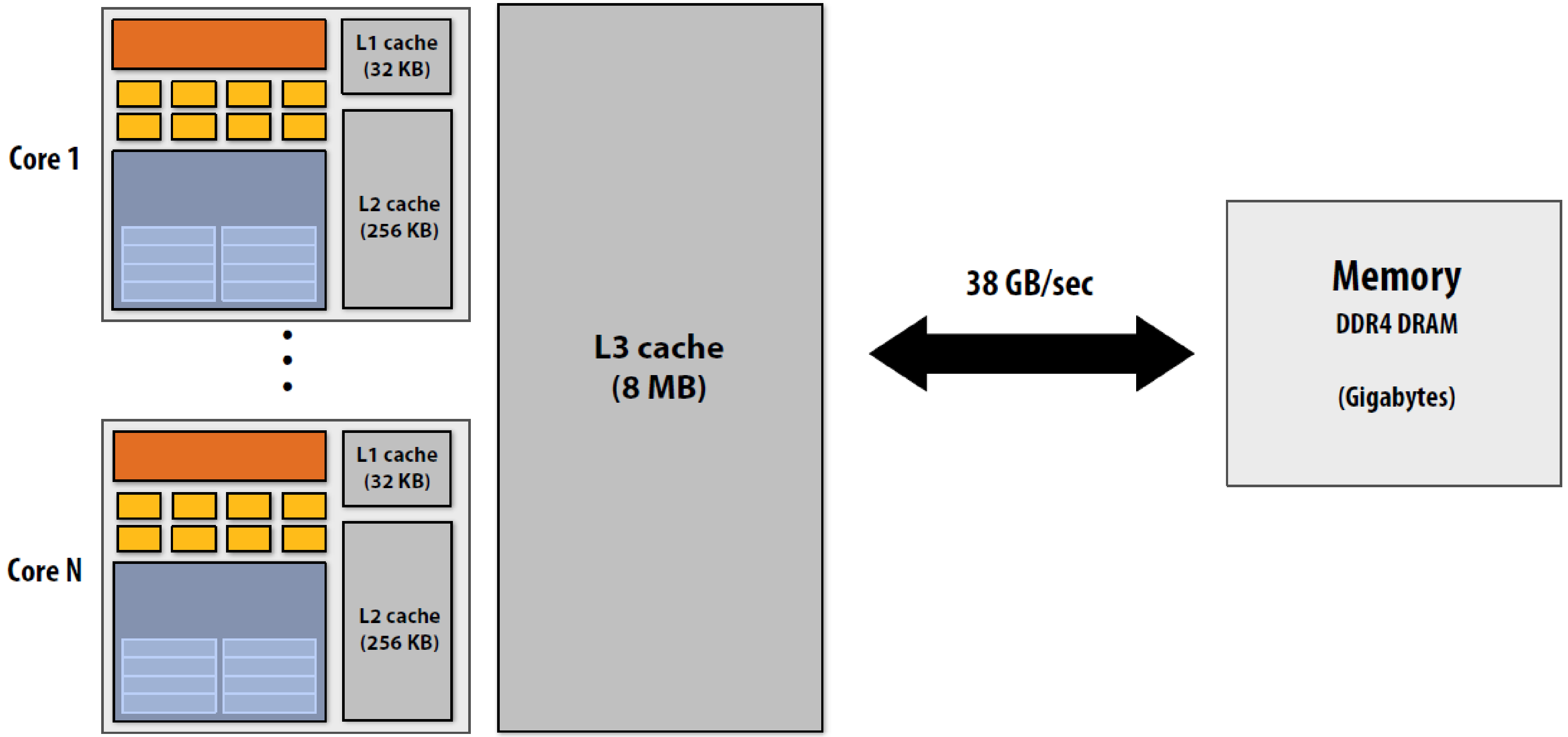
Line Address	Values in line
0x4	0 0 6 0
0xC	255 0 0 0

Address	Value
0x0	16
0x1	255
0x2	14
0x3	0
0x4	0
0x5	0
0x6	6
0x7	0
0x8	32
0x9	48
0xA	255
0xB	255
0xC	255
0xD	0
0xE	0
0xF	0
0x10	128

Caches reduce length of stalls (reduce memory access latency)

Processors run efficiently when they access data resident in caches

Caches reduce memory access latency when accessing data that they have recently accessed! *



* Caches also provide high bandwidth data transfer

Parallel Computer Memory Architectures

Memory architectures

- Shared Memory

- is **memory** that may be simultaneously accessed by multiple programs with an intent to provide communication among them or avoid redundant copies. **Shared memory** is an efficient means of passing data between programs.

- Distributed Memory

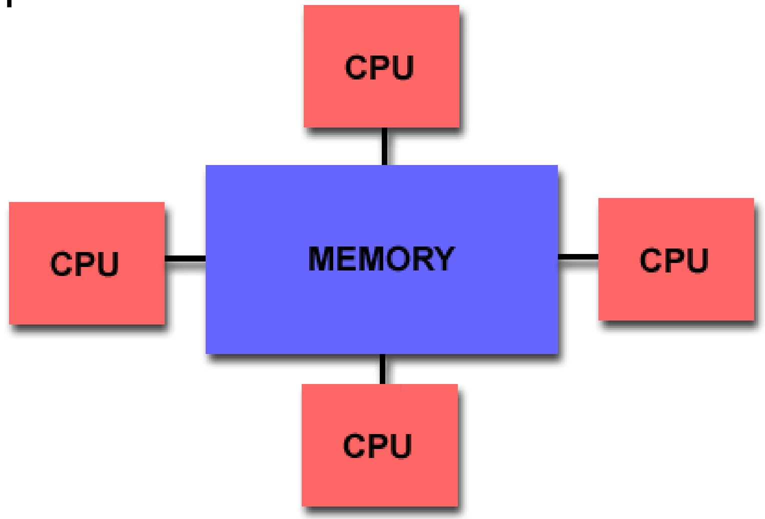
- refers to a multiprocessor computer system in which each processor has its own private memory. Computational tasks can only operate on local data, and if remote data is required, the computational task must communicate with one or more remote processors.

- Hybrid Distributed-Shared Memory

- hybrid programming techniques combining the best of distributed and shared memory programs are becoming more popular.

Shared Memory

- Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.



- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location effected by one processor are visible to all other processors.
- Shared memory machines can be divided into two main classes based upon memory access times: ***UMA*** and ***NUMA***.

Shared Memory : UMA vs. NUMA

- Uniform Memory Access (UMA):
  - Most commonly represented today by **Symmetric Multiprocessor (SMP) machines**
  - Identical processors
  - **Equal access and access times** to memory
  - Sometimes called CC-UMA - **Cache Coherent UMA**. Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.
- Non-Uniform Memory Access (NUMA):
  - Often made by physically linking **two or more SMPs**
  - One SMP can directly access memory of another SMP
  - Not all processors have equal access time to all memories
  - Memory access across link is slower
  - If cache coherency is maintained, then may also be called CC-NUMA - Cache Coherent NUMA

UMA vs NUMA

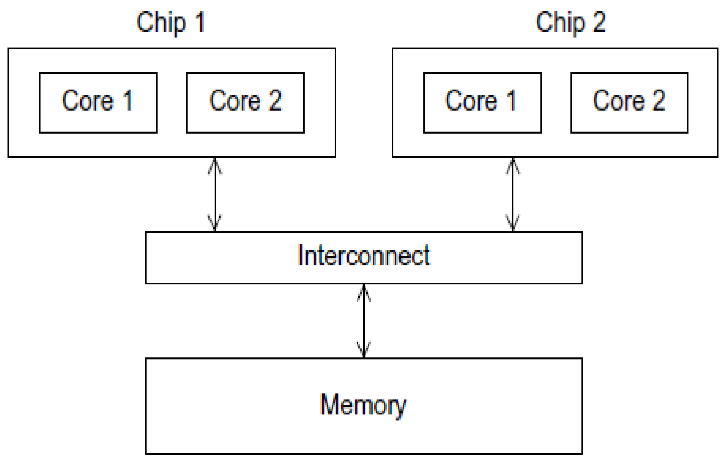


FIGURE 2.5

A UMA multicore system

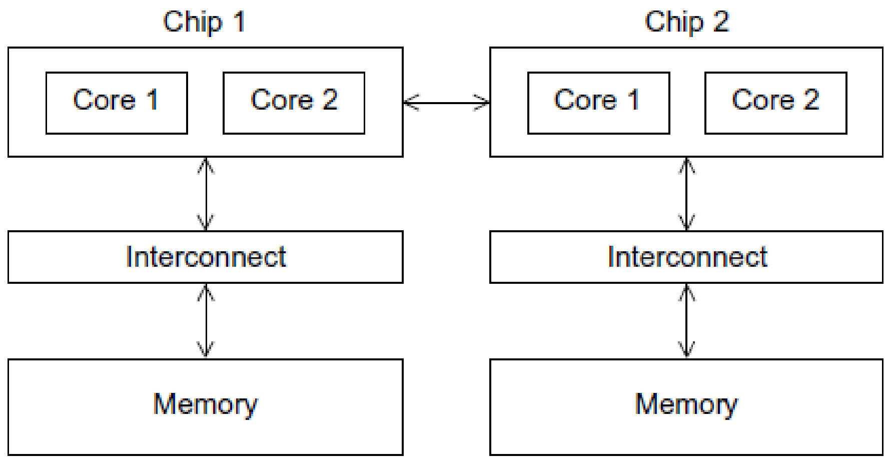


FIGURE 2.6

A NUMA multicore system

Shared Memory: Pro and Con

Advantages

- Global address space provides a user-friendly programming perspective to memory
- Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs

Disadvantages:

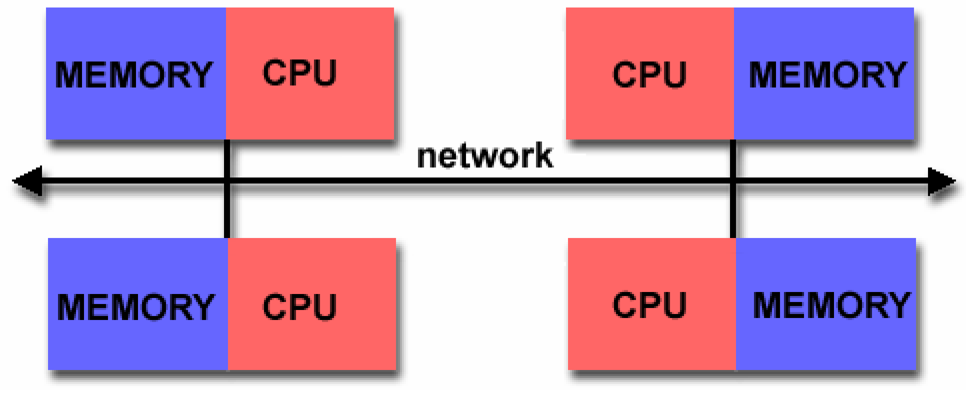
- Primary disadvantage is the **lack of scalability** between memory and CPUs. Adding more CPUs can geometrically increases traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.
- Programmer responsibility for **synchronization** constructs that insure "correct" access of global memory.
- ***Expense***: it becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.

Distributed Memory

In a **distributed memory system**, each processor has its own local memory. Processors do not share memory directly; instead, they communicate with one another through a network interconnect to exchange data. Unlike shared memory systems, there is no global memory address space.

Key Characteristics:

1. **Local Memory**:
2. **No Cache Coherency**:
3. **Explicit Communication**:
4. **Network Interconnect**:



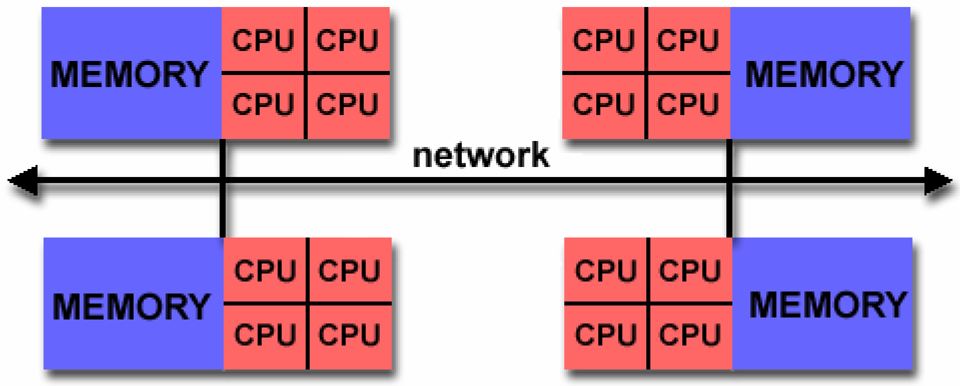
Distributed Memory: Pro and Con

- Advantages
  - Memory is **scalable** with number of processors. Increase the number of processors and the size of memory increases proportionately.
  - Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
  - Cost effectiveness: can use commodity, off-the-shelf processors and networking.
- Disadvantages
  - The programmer is responsible for many of the details associated with data communication between processors.
  - It may be difficult to map existing data structures, based on global memory, to this memory organization.
  - Non-uniform memory access (NUMA) times

Hybrid Distributed-Shared Memory

- Hybrid distributed-shared memory systems combine the strengths of **shared memory** and **distributed memory** architectures.
- Supports scalability, flexibility, and efficient resource utilization, making it a dominant choice for modern **high-performance computing (HPC)** systems
- **Key Features:**

1. **Shared Memory Component:**
2. **Distributed Memory Component:**
3. **Communication Between Nodes:**



Hybrid Distributed-Shared Memory

Comparison of Shared and Distributed Memory Architectures			
Architecture	CC-UMA	CC-NUMA	Distributed
Examples	SMPs Sun Vexx DEC/Compaq SGI Challenge IBM POWER3	Bull NovaScale SGI Origin Sequent HP Exemplar DEC/Compaq IBM POWER4 (MCM)	Cray T3E Maspar IBM SP2 IBM BlueGene
Communications	MPI Threads OpenMP shmem	MPI Threads OpenMP shmem	MPI
Scalability	to 10s of processors	to 100s of processors	to 1000s of processors
Draw Backs	Memory-CPU bandwidth	Memory-CPU bandwidth Non-uniform access times	System administration Programming is hard to develop and maintain
Software Availability	many 1000s ISVs	many 1000s ISVs	100s ISVs

TAYLOR'S UNIVERSITY
Wisdom · Integrity · Excellence

Any Question?