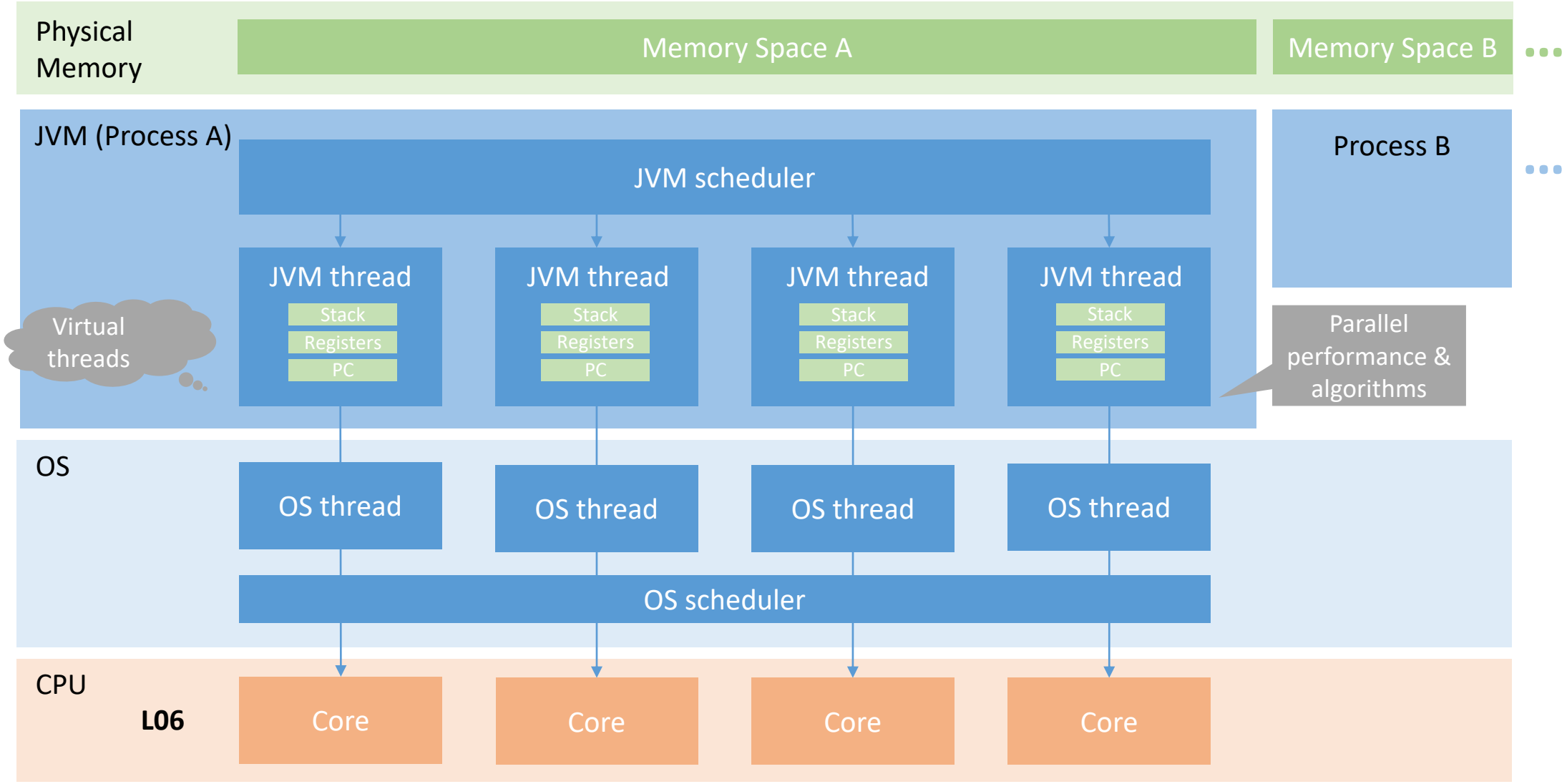
Parallel Programming

Processor Evolution, Caches, Vectorization, ILP, Pipelining

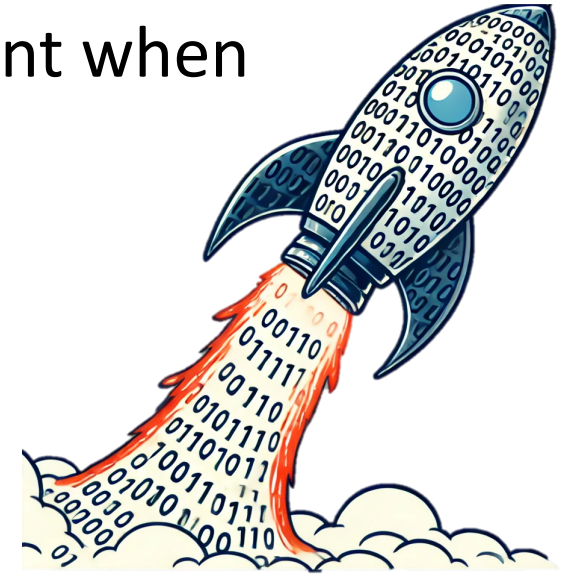
Big picture (Part I)



Code efficiency: More than just parallelism

Earlier: The major motivation behind parallel processing is to achieve speedup

More general: We want our programs to execute as efficiently as possible. To achieve this, we need to understand the hardware-level design decisions that impact code efficiency - also relevant when running on a single core.



Structure of this lecture

From single-core to multi-core processors

Single-core optimization:

1. Memory: Caching
2. Compute optimization: Vectorization, ILP, pipelining

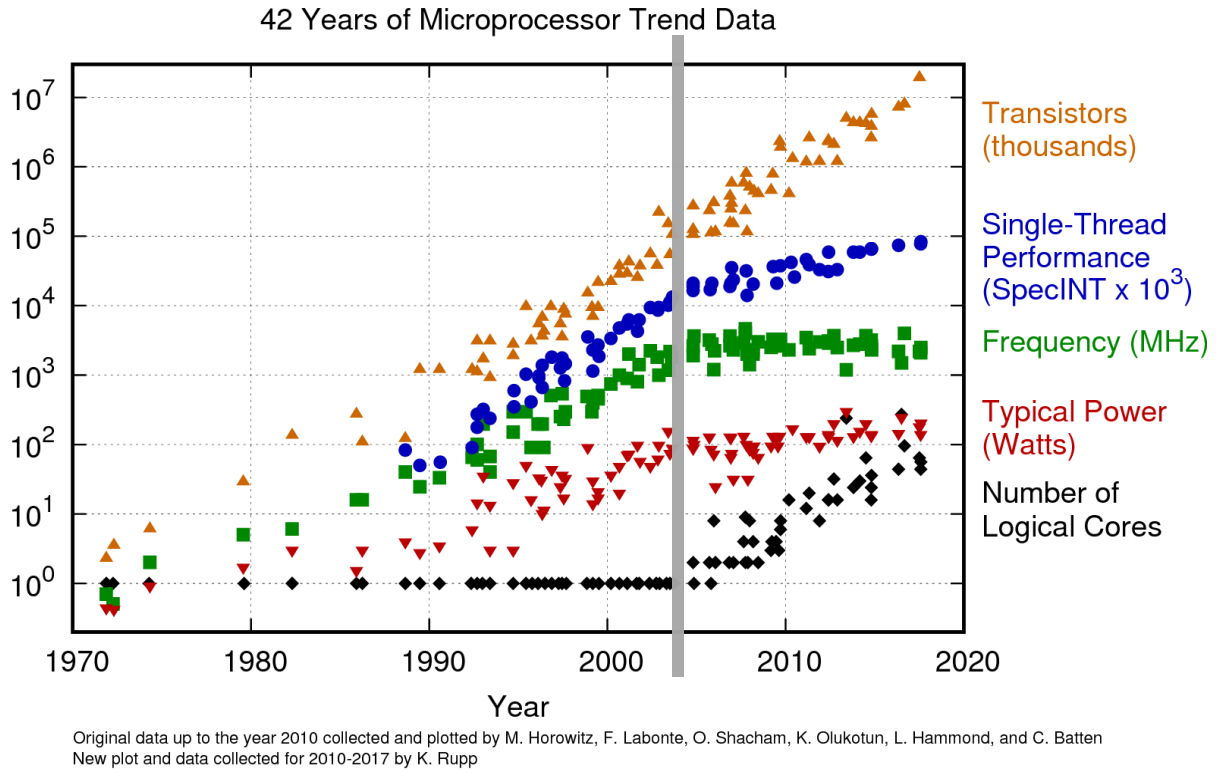
** Some slides are inspired by Kayvon Fatahalian's Parallel Computing course, Stanford University*

From single-core to multi-core processors

More transistors, more performance

- **Moore's Law**: Transistor counts on a chip double approximately every two years
- **Until the early 2000s**: Shrinking transistors enabled both higher transistor densities and higher clock speeds, directly driving computational power growth

→ wait a year and your sequential program became faster!



Karl Rupp's blog

What is a computer program?

A program is just a list of instructions

```
public class Main {
    public static void main(String[] args) {
        float a;
        float x = 2;
        float y = 4;
        float z = 6;

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

JVM's instruction pointer

JVM's instruction stream

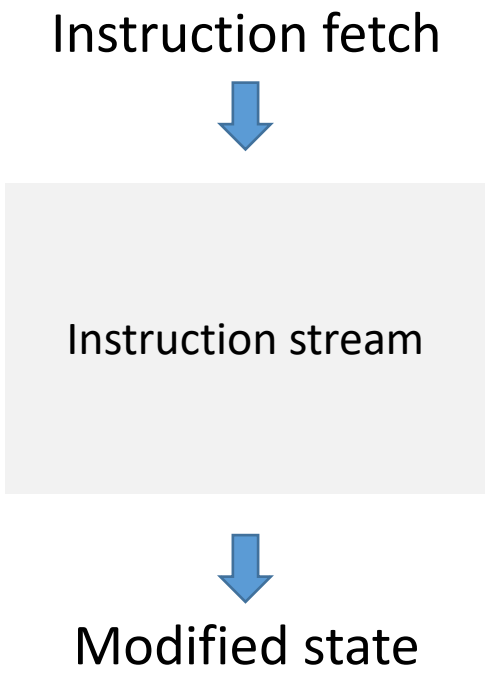
Java bytecode:

```
0:  iconst_2      // Push 2 onto the stack
1:  f2d           // Convert to float
2:  fstore_1      // Store float in local variable 1 (x)
3:  iconst_4      // Push 4 onto the stack
4:  f2d           // Convert to float
5:  fstore_2      // Store float in local variable 2 (y)
6:  bipush 6      // Push 6 onto the stack
7:  f2d           // Convert to float
8:  fstore_3      // Store float in local variable 3 (z)
9:  fload_1       // Load x from local variable 1
10: fload_1       // Load x again
11: fmul          // x * x
12: fload_2       // Load y from local variable 2
13: fload_2       // Load y again
14: fmul          // y * y
15: fadd          // (x * x) + (y * y)
16: fload_3       // Load z from local variable 3
17: fload_3       // Load z again
18: fmul          // z * z
19: fadd          // ((x * x) + (y * y)) + (z * z)
20: fstore_0      // Store the result in a (local variable 0)
21: return        // Return from main
```

What does a processor do?

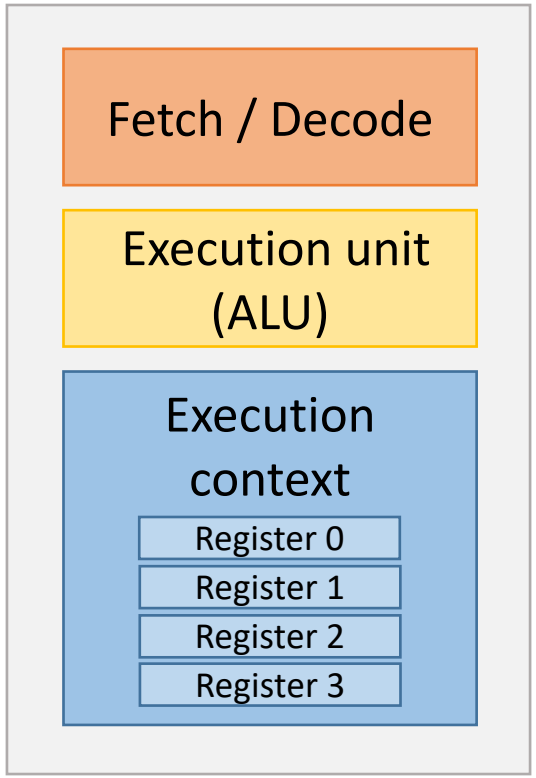
A processor executes instructions

- An instruction describes an operation for a processor to perform
- Executing an instruction typically modifies the computer's state (= values in memory or values in processor registers)
- Executes one instruction per clock



A very simplified processor

A processor executes instructions



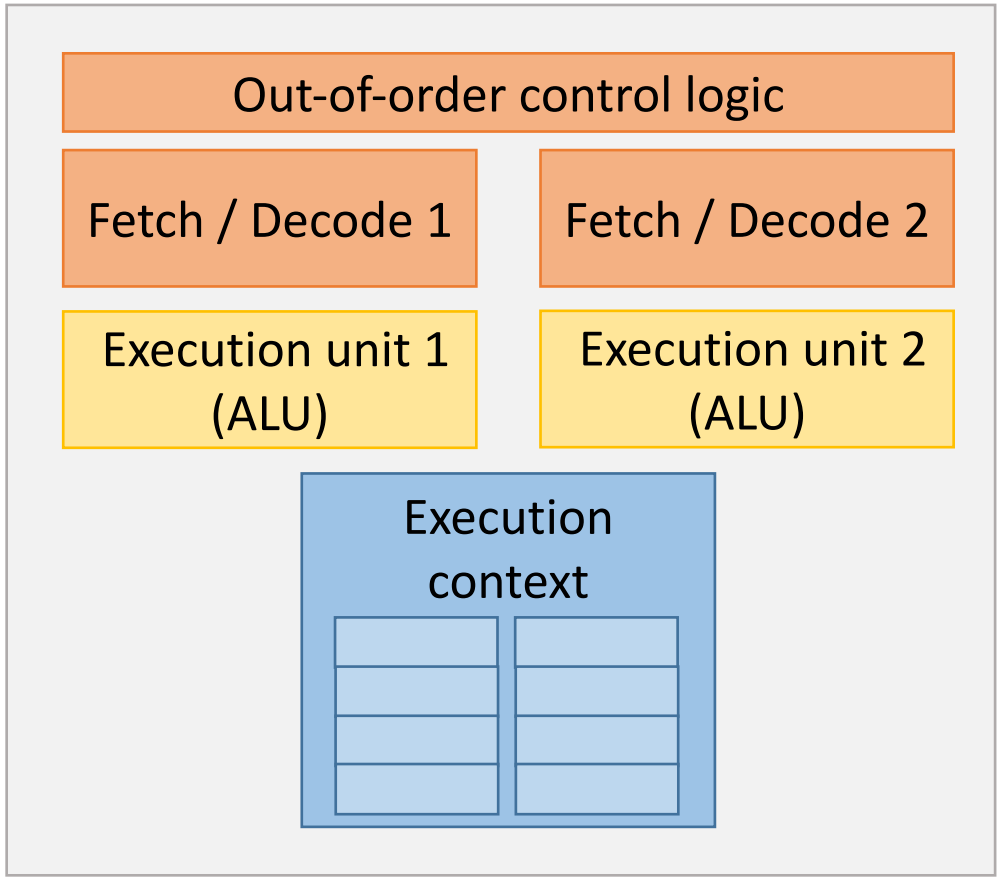
Determine what instruction to run next

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

Pre multi-core era: Parallel execution using multiple execution units

Improving instruction throughput: ILP (instruction level parallelism)

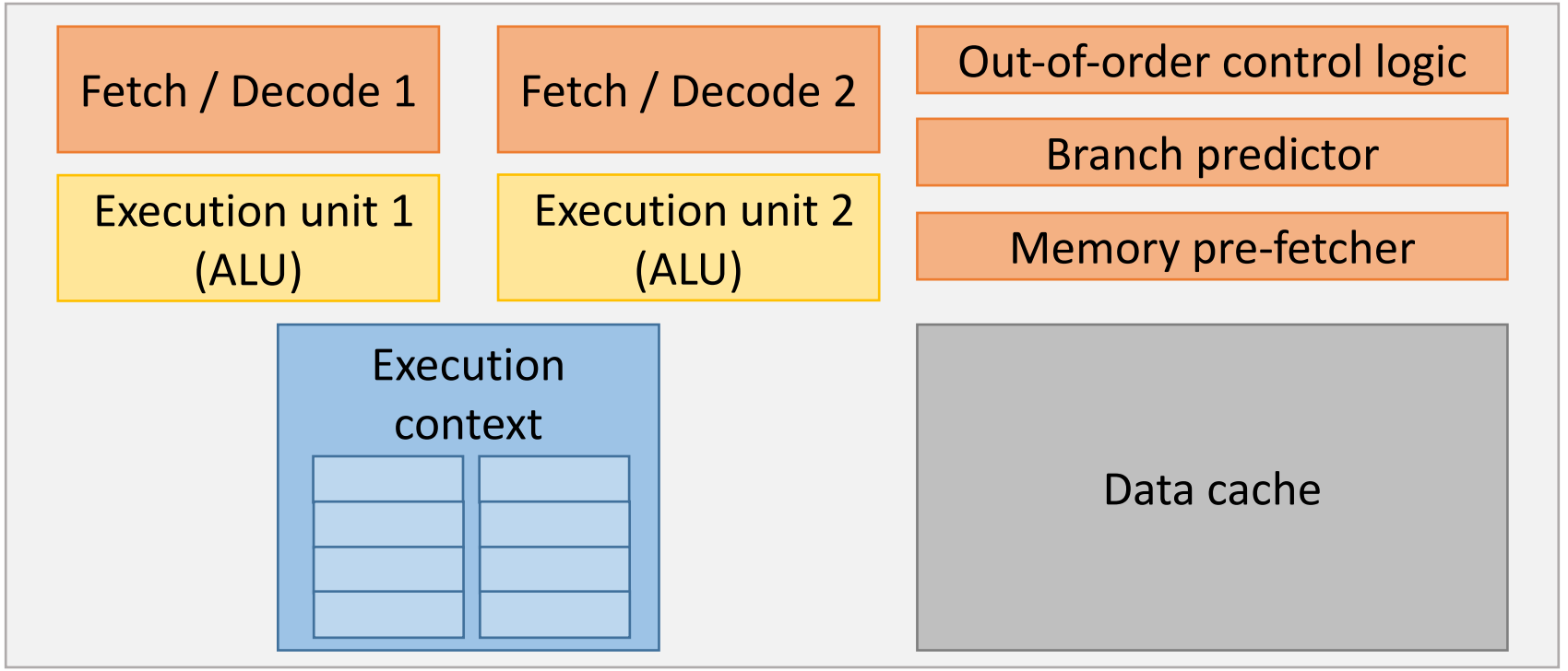


- **Superscalar execution:** Processor can issue multiple instructions per cycle to multiple execution units
- **Out-of-order execution:** Reorder instructions to keep execution units busy
- **Pipelining:** Different stages of multiple instructions are processed simultaneously
- **SIMD:** Vector units that allow instructions to perform the same operation on multiple data elements in parallel

Pre multi-core era: Mechanisms to support ILP

Increasing peak throughput: Wider issue width, more execution units, SIMD

Reducing stalls: Out-of-order scheduling, branch prediction, memory prefetching, larger caches



For a long time...

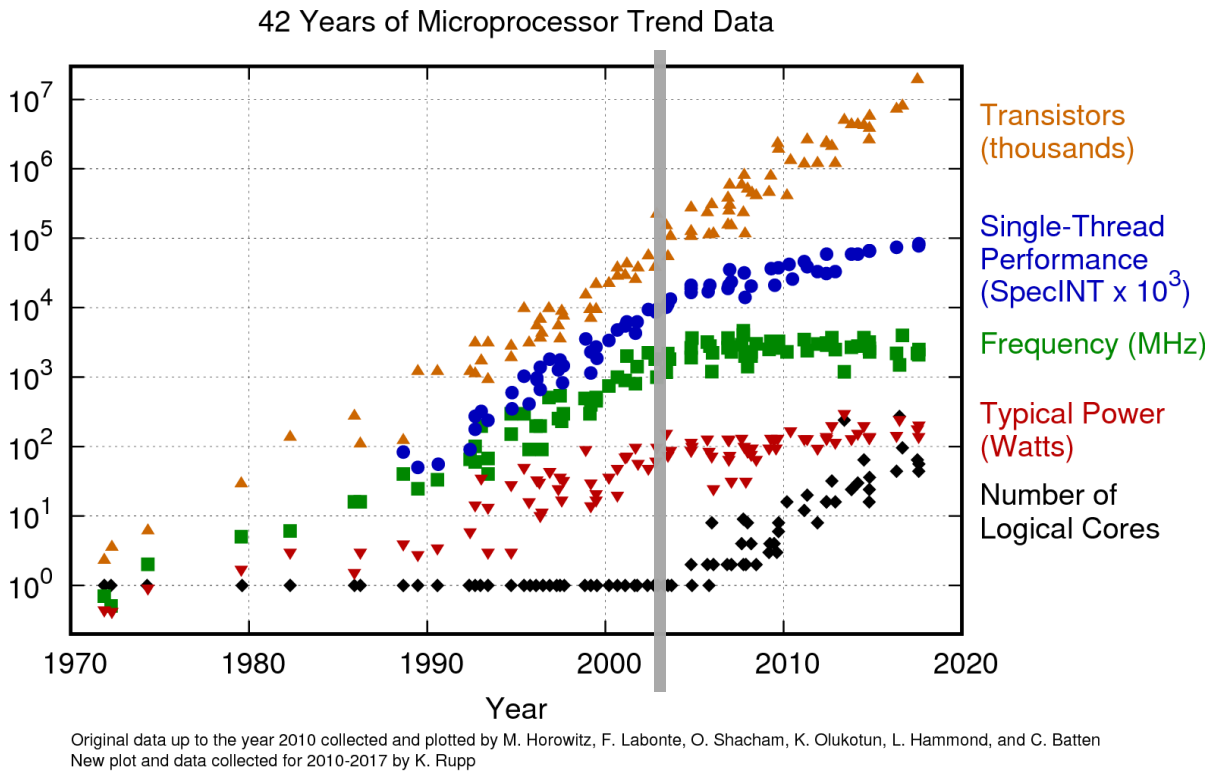
- CPU architects improved sequential execution by exploiting Moore's law and ILP
- Sequential programs were becoming exponentially faster with each new CPUs
    - Most programmers did not worry about performance
    - They waited for the next CPU generation

But architects hit walls

- Power consumption and heat dissipation wall
  - Faster CPU → consumes more energy → expensive to cool
  - Limited further increase in clock speeds
- ILP wall
  - Limits in inherent program's ILP
    → diminishing returns in extracting parallelism from sequential code
- Memory wall
  - CPUs faster than memory access

Multi-core era

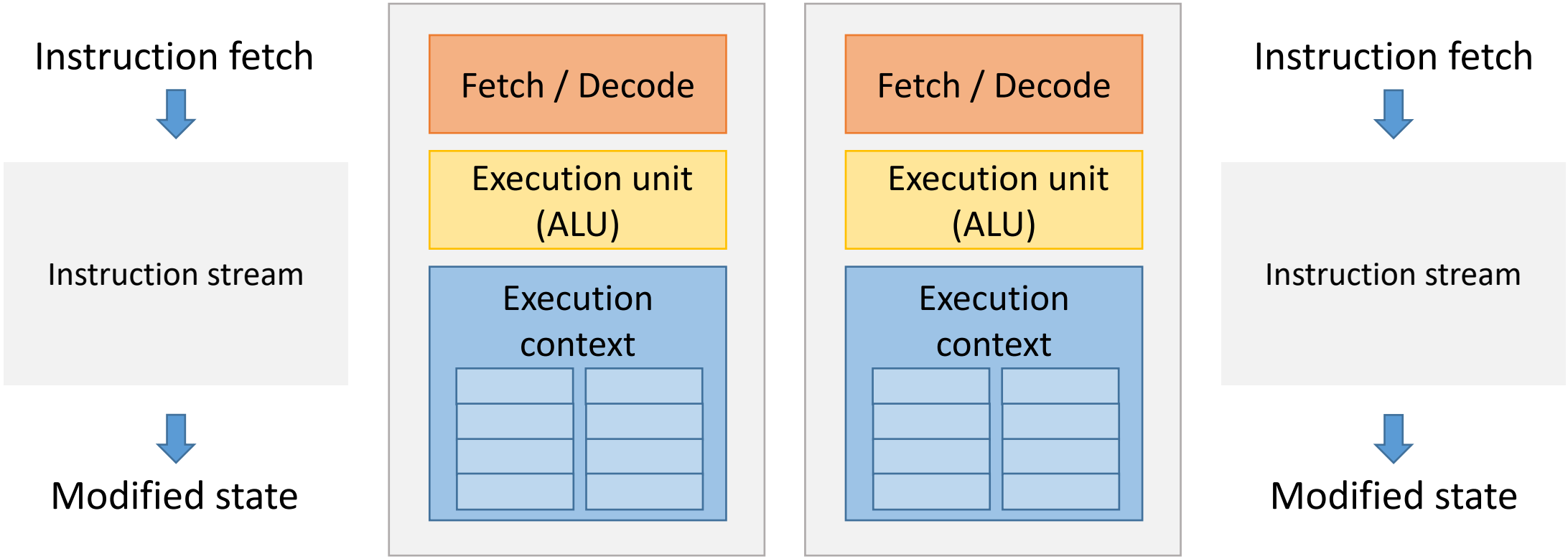
- **Post-2003:** Transistor counts continued to increase, but the rate of growth slowed significantly
- **Shift to multi-core architectures**



Karl Rupp's blog

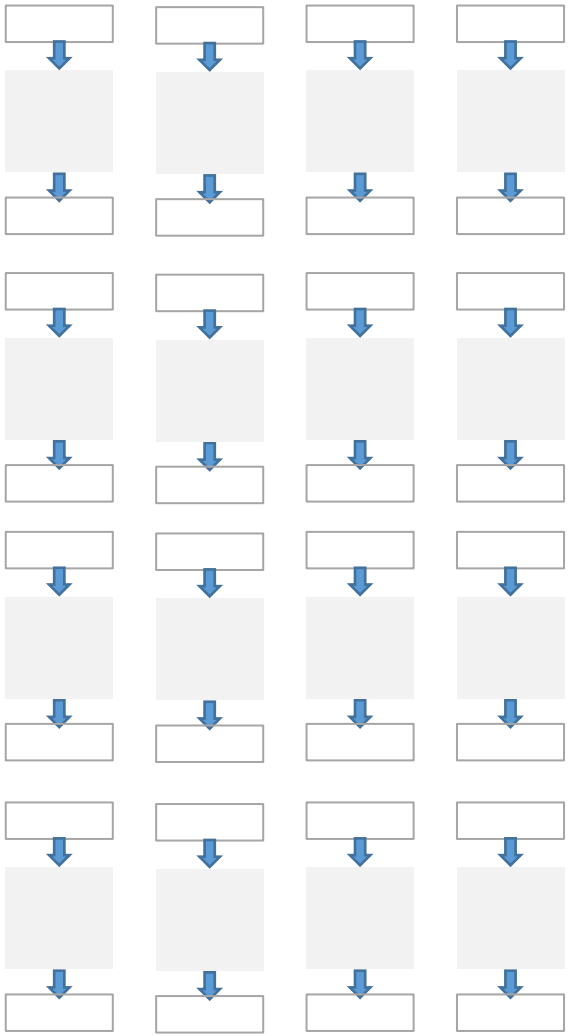
Multi-core era processor

Instead of making a single core faster, put multiple cores on a single chip



Independent execution: Each core operates independently, fetching and executing its own instructions (unless the threads/tasks need to synchronize or share data)

Sixteen cores

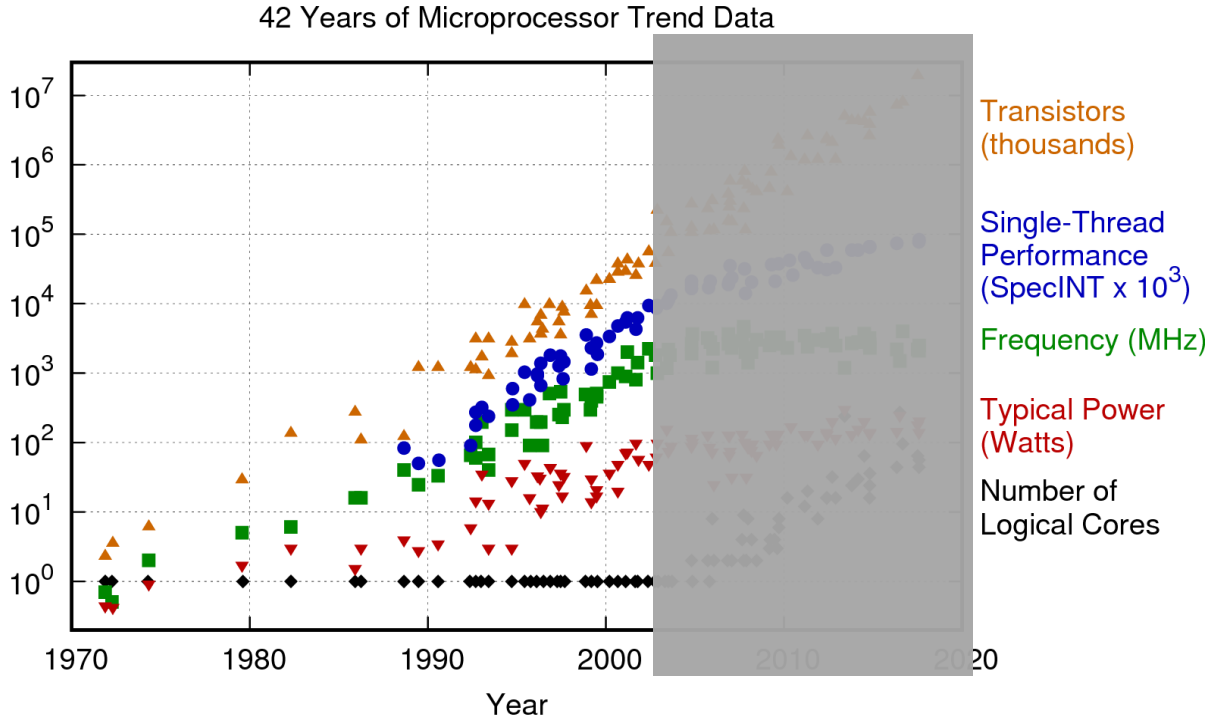


More cores, more work for us!

- But: The original program will compile to an instruction stream that runs as one thread on one of the processor cores
- To fully utilize multiple cores, we need to parallelize tasks
- **Implication:** Programmers must design software to split work into independent tasks or threads to take advantage of the hardware
- Effective parallel programming is key to maximizing the performance of multi-core systems
- Going back to sentence on previous slide:
  Each core operates independently, fetching and executing its own instructions (unless the threads/tasks need to synchronize or share data)

Before cores: Diving deeper into single-core optimization

Understanding how the hardware works allows programmers to write more efficient parallel programs by optimizing software to align with the underlying hardware capabilities



Karl Rupp's blog

Before cores: Diving deeper into single-core optimization

Overview of the remaining part of the lecture:

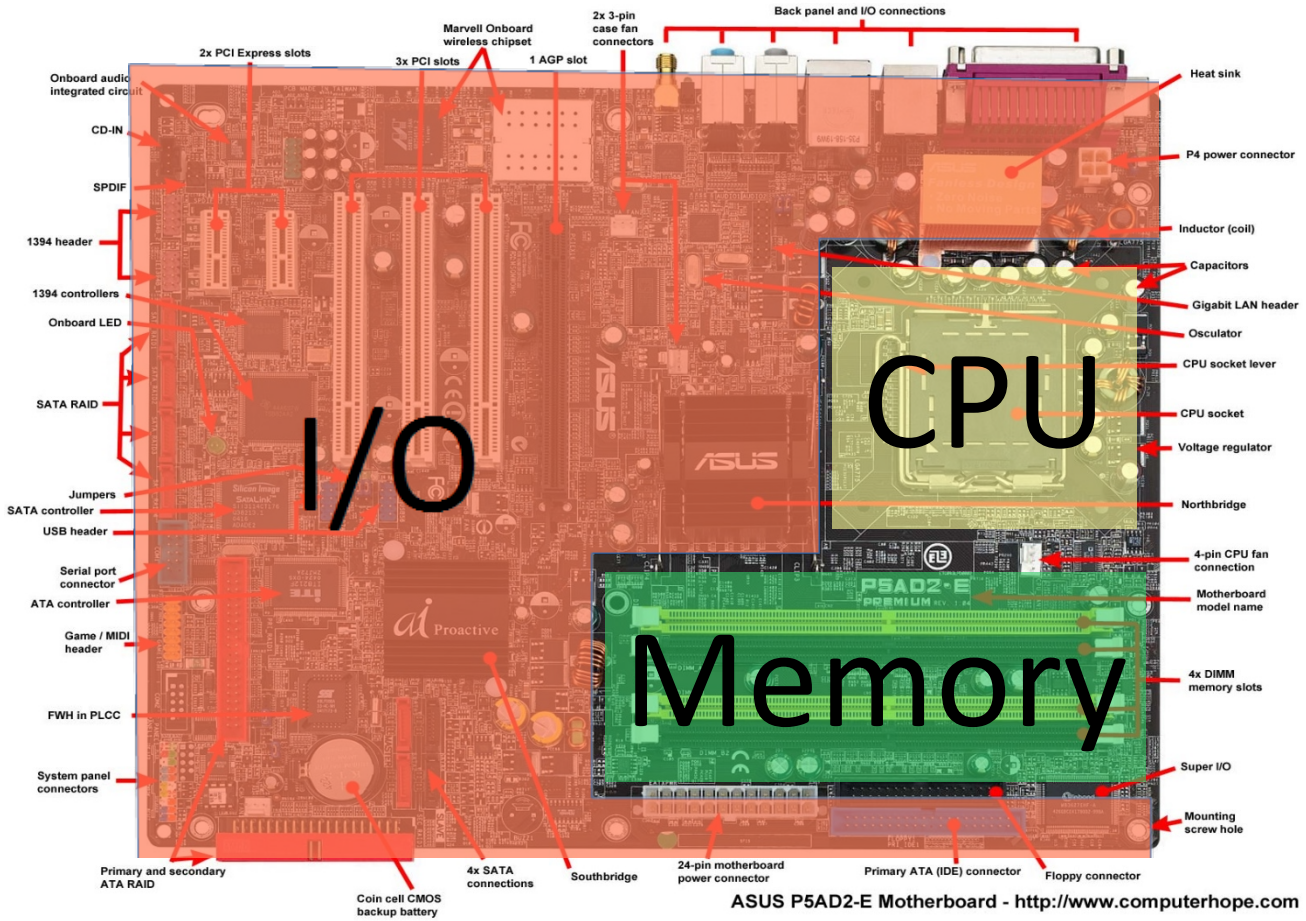
1. Memory is a bottleneck
   → Caching

2. Compute optimizations
   → Vectorization, ILP and pipelining

Caching

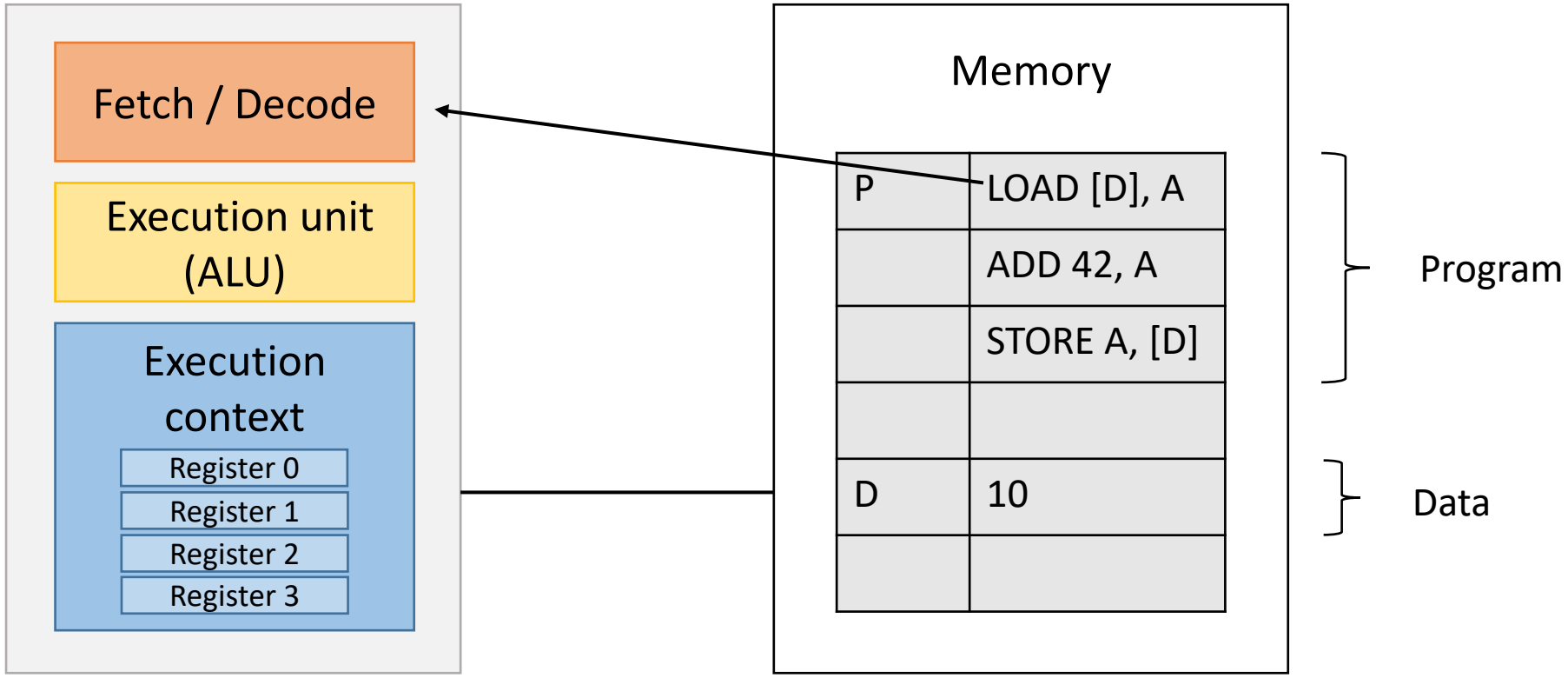
Today's computers: different appearances …

… similar from the inside



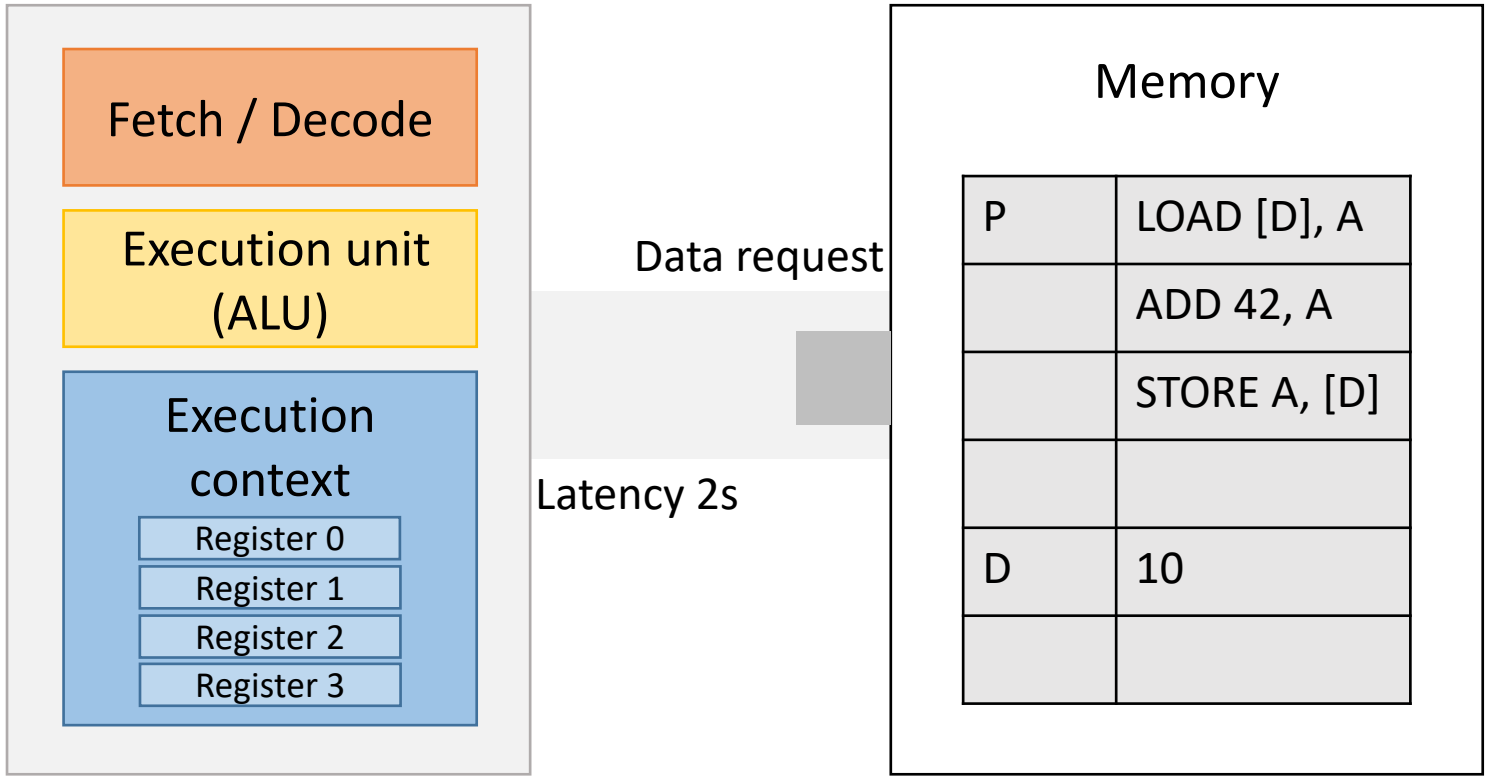
Basic principles of today's computers

Von Neumann architecture: program data and program instructions in the same memory (“matches” imperative programming languages)



Memory access

Memory access latency: the amount of time it takes the system to provide data to the processor (e.g., 100 clock cycles)



CPU caches

CPUs grew faster

Memories grew bigger

A processor “stalls” when it cannot run the next instruction in an instruction stream

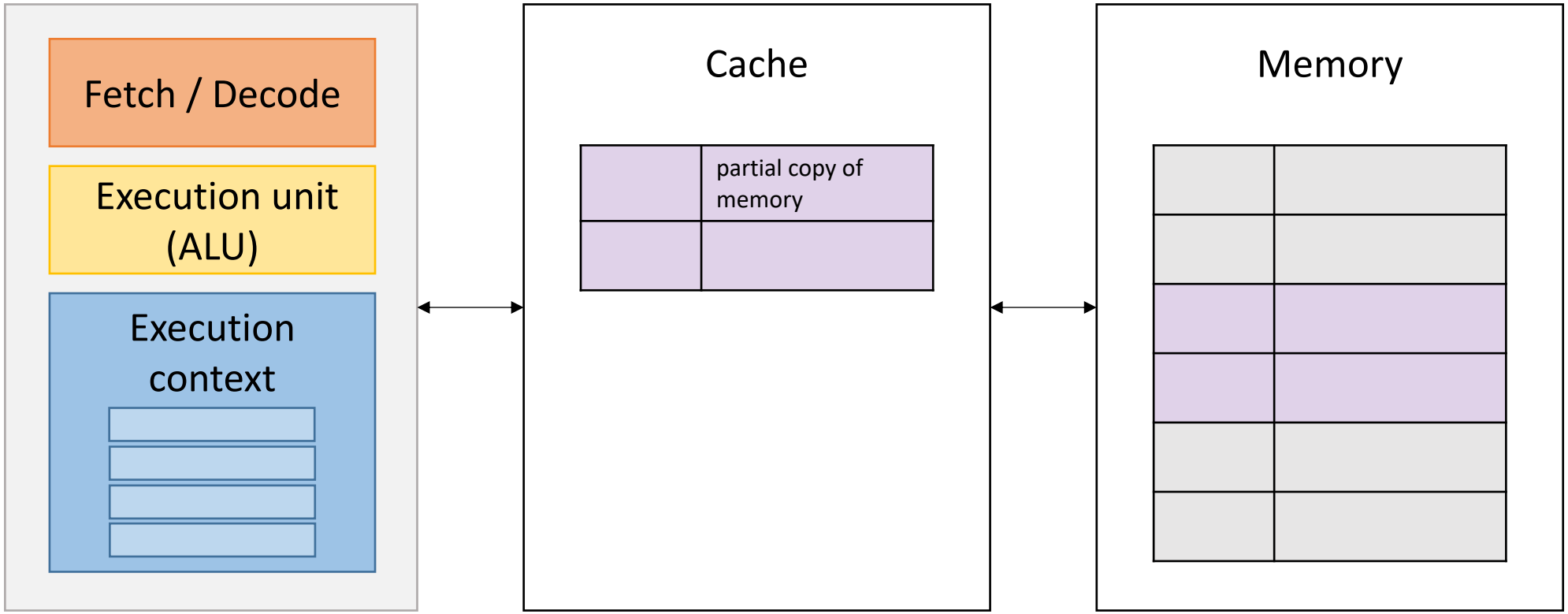
Accessing memory is a major source of stalls

Caches that are smaller but faster than memory

Caches reduce length of stalls

Caches reduce memory access latency

Data **locality**: related storage locations (spatial) are often accessed shortly after each other (temporal)



Code example: L06_01_cache_effects

Cache effects

```
const int N = 800000000;
const int STRIDE = 1;

int main()
{
  std::vector<int> data(N, 1); // Fill the vector with 1s
  long long result = 0;

  // Loop over all elements but access them in a strided way
  for (int start = 0; start < STRIDE; ++start) //Start at different offsets
  {
        for (int j = start; j < N; j += STRIDE) // Strided access
        {
           result += data[j];
        }
    }
    std::cout << "Sum: " << result << std::endl;
    return 0;
}
```

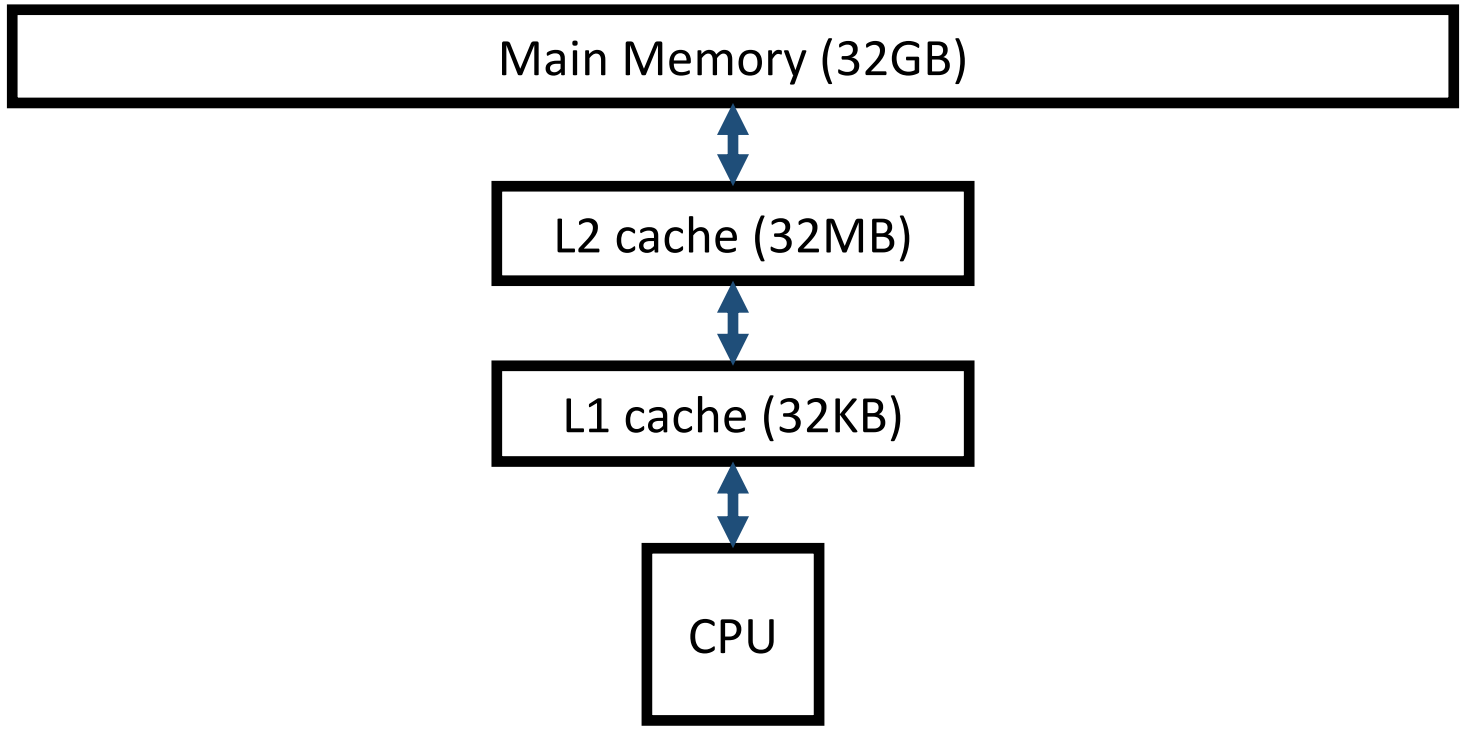
STRIDE = 1: best performance, sequential access, efficient due to cache locality, cache prefetching works well, ideal for vectorization

STRIDE > 1 (e.g. 64): simulated cache-unfriendly access patterns, causing cache misses, prefetching ineffective, increased memory latency

g++ -o stridedAccess stridedAccess.cpp –O3
Time ./ stridedAccess

STRIDE = 1: 2.1s
STRIDE = 64: 7.2s

CPUs and memory hierarchies



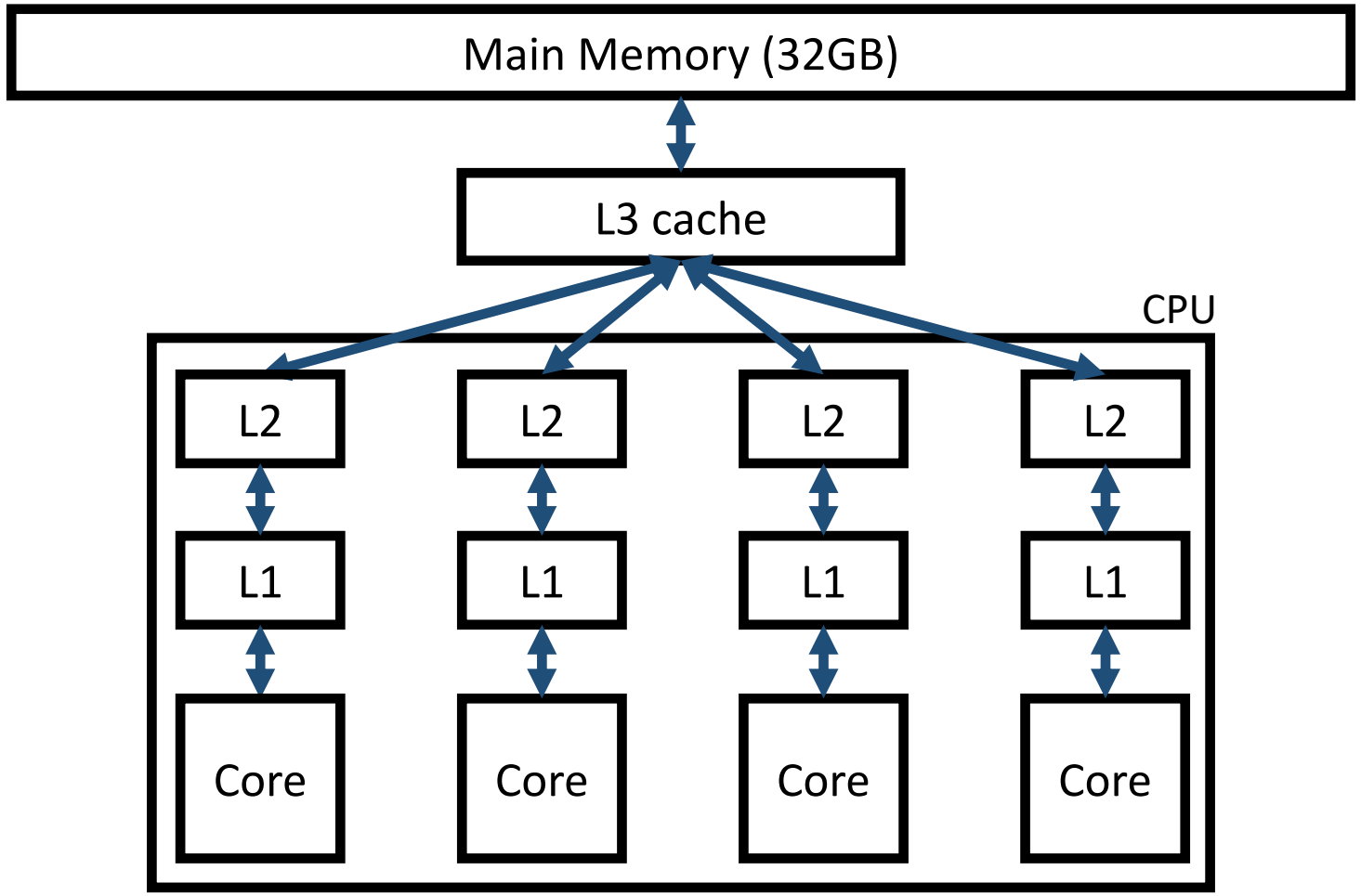
CPU reads/writes values from/to main memory

Hierarchy of memory caches in between

Faster memory is more expensive, hence smaller: L1 is 5x faster than L2, which is 30x faster than main memory, which is 350x faster than disk

Memory size and speed are approximated but realistic numbers

CPUs and memory hierarchies



Multi-core CPUs have caches per core → more complicated hierarchies

Compute optimization

Compute optimization

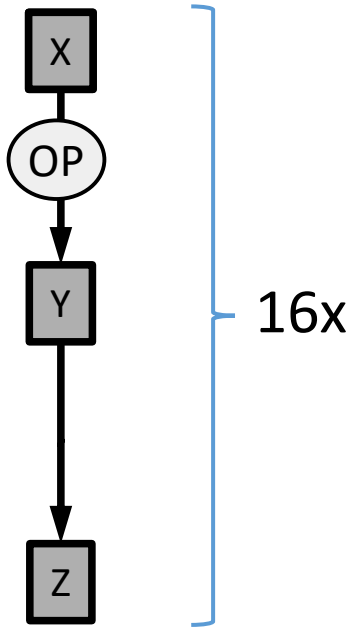
Three approaches to apply parallelism to improve sequential processor performance

- **Vectorization:** Exposed to developers
- **Instruction Level Parallelism (ILP):** Inside CPU
- **Pipelining:** Also internal, but transfers to software

Sequential: Element-wise addition

```
int[] a = { ... };
int[] b = { ... };
int[] c = new int[16];

for (int i = 0; i < 16; i++) {
    c[i] = a[i] + b[i];
}
```



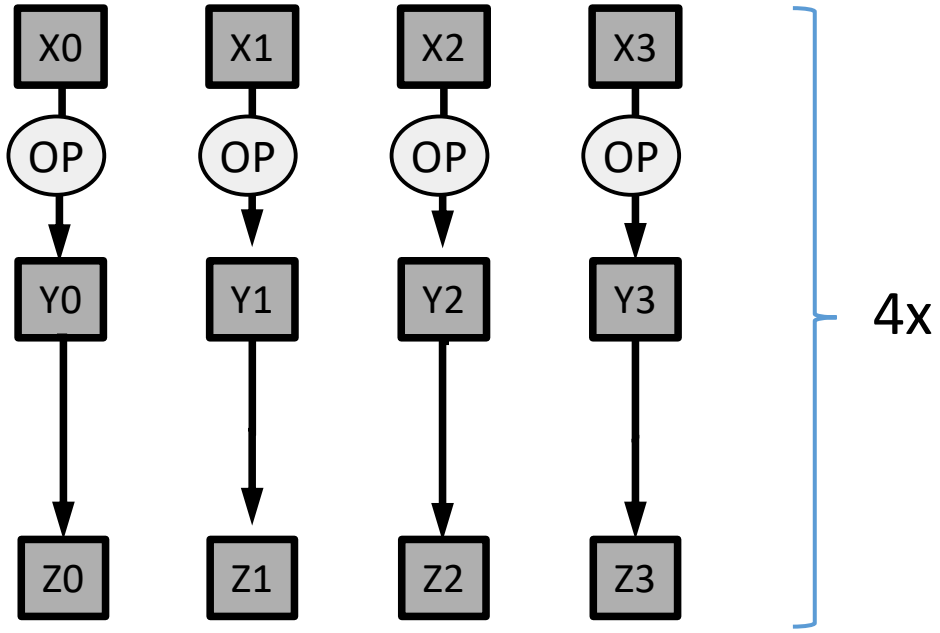
For loop: Every single iteration is doing the exact same thing, exact same code

Sequential code, but: compiler detects data parallelism so the CPU can execute faster

Sequential: Element-wise addition

```
int[] a = { ... };
int[] b = { ... };
int[] c = new int[16];

for (int i = 0; i < 16; i+=4) {
    c[i+0] = a[i+0] + b[i+0];
    c[i+1] = a[i+1] + b[i+1];
    c[i+2] = a[i+2] + b[i+2];
    c[i+3] = a[i+3] + b[i+3];
}
```

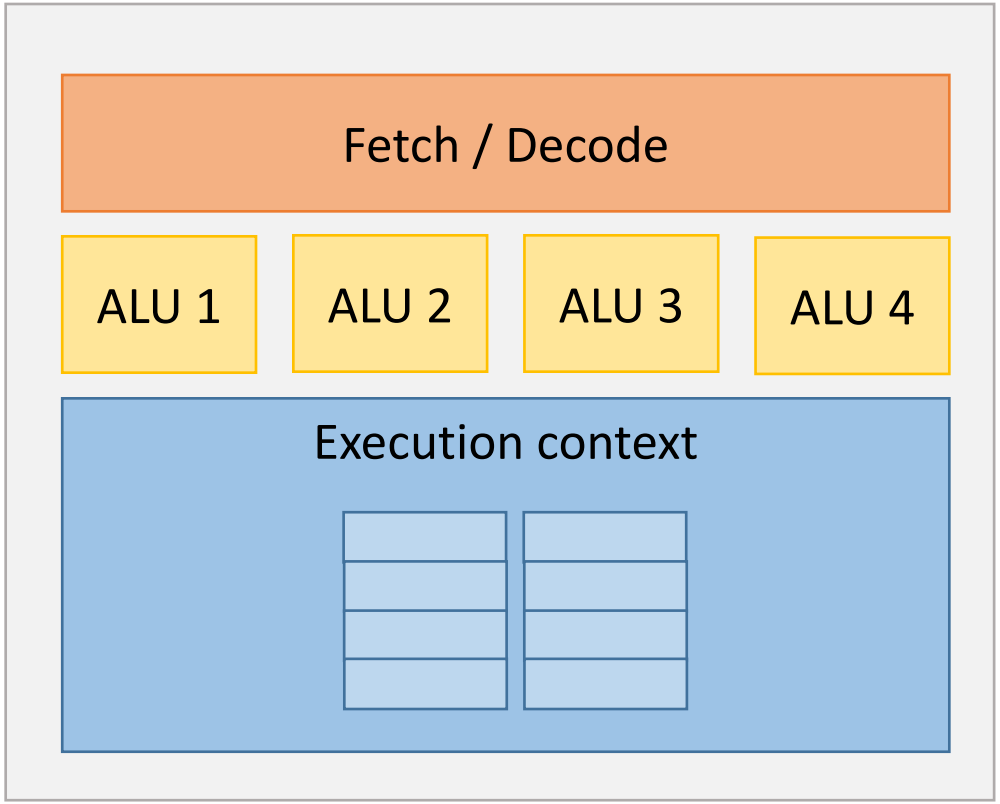


Vectorized

```
for (int i = 0; i < 16; i+=4) {
    c[i:i+3] = a[i:i+3] + b[i:i+3];
}
```

Opportunity for higher efficiency: decode the instruction once and broadcast it to multiple ALUs

Leverage execution units (ALUs)



Note: conceptual view; real CPUs implement this using vector units with multiple lanes

Managing an instruction stream across many ALUs

SIMD processing:
Single instruction, multiple data

Same instruction broadcast to multiple ALUs

Operation executed in parallel on different data elements

Code example: L06_02_gcc_vectorize

Vectorized program

```
int[] a = { ... };
int[] b = { ... };
int[] c = new int[16];

for (int i = 0; i < 16; i++) {
    c[i] = a[i] + b[i];
}
```

Compiled with –O3

```
movdqa   xmm0, XMMWORD PTR [rsi+rax]
paddd    xmm0, XMMWORD PTR [r8+rax]
movaps   XMMWORD PTR [rdx+rax], xmm0
```

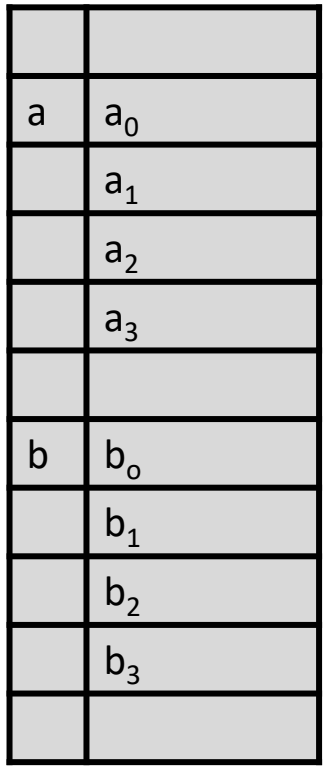
Step 1: load (mem->registers)

Step 2: Operation

Step 3: store (registers->mem)

Standard way: 1-at-a-time

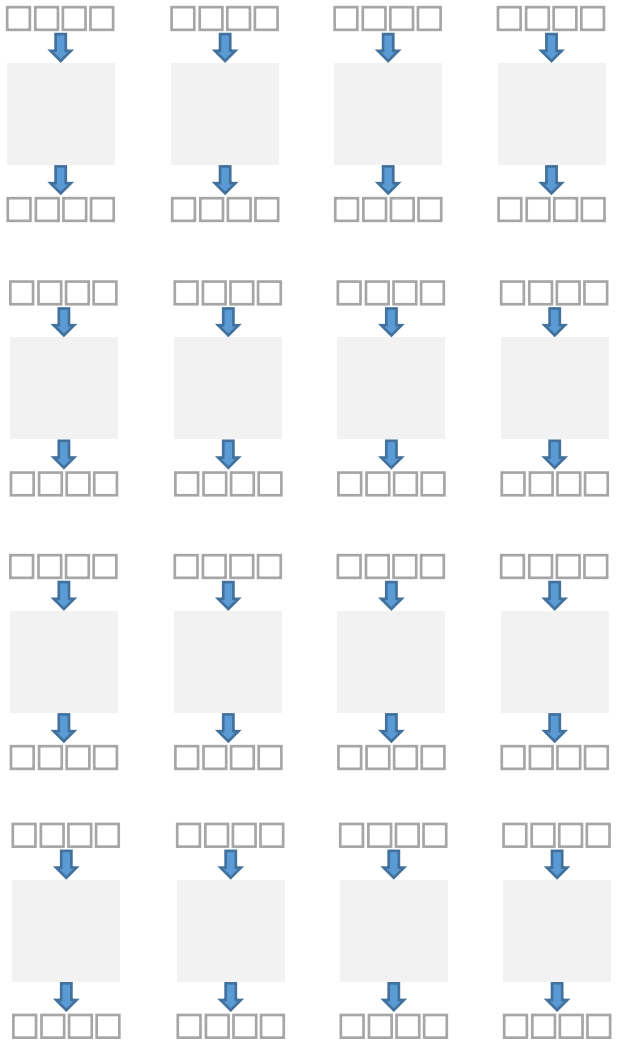
Vectorized way: N-at-a-time



movdqa: SIMD instruction. Performs a SIMD add. Moves 128 bits of data between registers and memory (i.e., move four 32 bit integers into SIMD registers to process them at once)

paddd: Vector instruction. Operates on multiple data elements simultaneously, performs a SIMD add

16 SIMD cores, 64 elements in parallel



16 cores -> 16 instruction streams, SIMD within each core -> up to 64 operations in parallel

Independent instructions

```
public class Main {
    public static void main(String[] args) {
        float a;
        float x = 2;
        float y = 4;
        float z = 6;

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

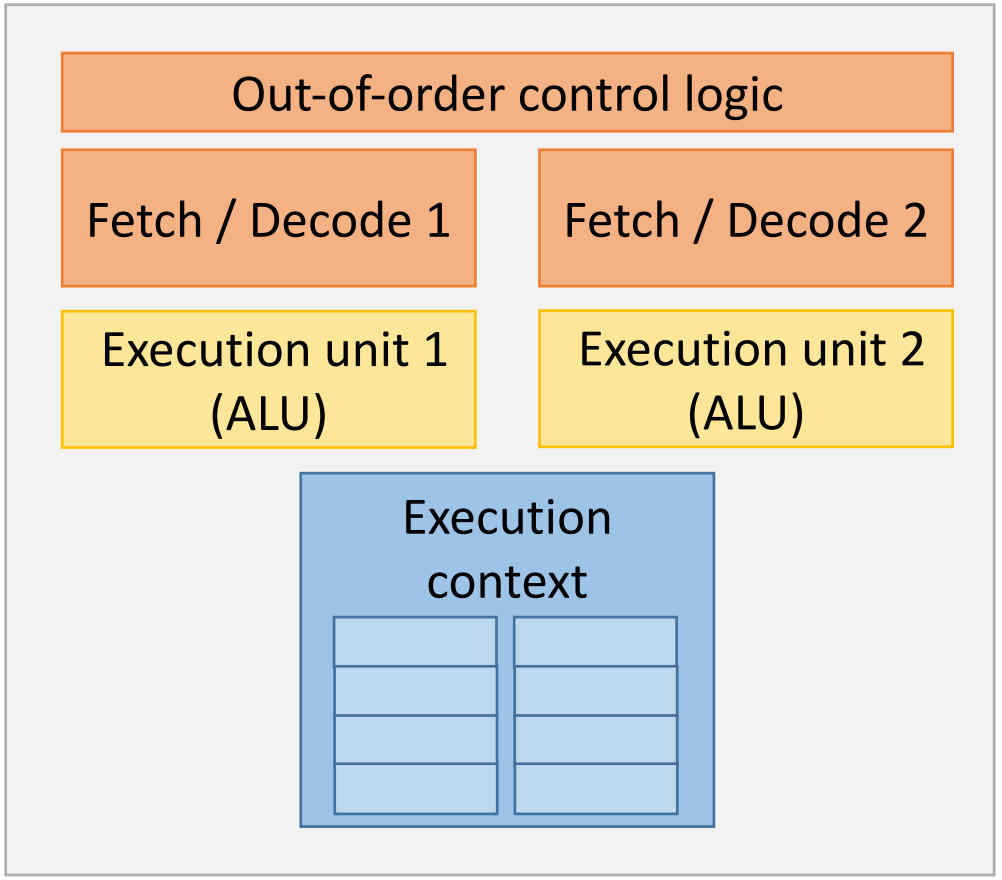
JVM's instruction stream

```
Java bytecode:
0:  iconst_2        // Push 2 onto the stack
1:  f2d             // Convert to float
2:  fstore_1        // Store float in local variable 1 (x)
3:  iconst_4        // Push 4 onto the stack
4:  f2d             // Convert to float
5:  fstore_2        // Store float in local variable 2 (y)
6:  bipush 6        // Push 6 onto the stack
7:  f2d             // Convert to float
8:  fstore_3        // Store float in local variable 3 (z)
9:  fload_1         // Load x from local variable 1
10: fload_1         // Load x again
11: fmul            // x * x
12: fload_2         // Load y from local variable 2
13: fload_2         // Load y again
14: fmul            // y * y
15: fadd            // (x * x) + (y * y)
16: fload_3         // Load z from local variable 3
17: fload_3         // Load z again
18: fmul            // z * z
19: fadd            // ((x * x) + (y * y)) + (z * z)
20: fstore_0        // Store the result in a (local variable 0)
21: return          // Return from main
```

Superscalar execution

Processor finds independent instructions in a single instruction stream and executes them in parallel on multiple execution units

- Superscalar execution is part of ILP (instruction level parallelism)



Independent instructions

```
public class Main {
    public static void main(String[] args) {
        float a;
        float x = 2;
        float y = 4;
        float z = 6;

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

JVM's instruction stream

Java bytecode:

```
0:  iconst_2        // Push 2 onto the stack
1:  f2d             // Convert to float
2:  fstore_1        // Store float in local variable 1 (x)
3:  iconst_4        // Push 4 onto the stack
4:  f2d             // Convert to float
5:  fstore_2        // Store float in local variable 2 (y)
6:  bipush 6        // Push 6 onto the stack
7:  f2d             // Convert to float
8:  fstore_3        // Store float in local variable 3 (z)
9:  fload_1         // Load x from local variable 1
10: fload_1         // Load x again
11: fmul            // x * x
12: fload_2         // Load y from local variable 2
13: fload_2         // Load y again
14: fmul            // y * y
15: fadd            // (x * x) + (y * y)
16: fload_3         // Load z from local variable 3
17: fload_3         // Load z again
18: fmul            // z * z
19: fadd            // ((x * x) + (y * y)) + (z * z)
20: fstore_0        // Store the result in a (local variable 0)
21: return          // Return from main
```

Optimizing ILP – Prefetching

Instruction prefetching: Preloading instructions into the CPU cache before they are needed → maximizing opportunities for superscalar execution

Consider the following program:

```
x = a + b;      // this and the one below are independent
y = c + d;      // can execute these 2 instructions in parallel
z = x * y;      // this one depends on results above, so has to wait
```

Independent, if

- different register names
- different memory addresses

Optimizing ILP – Speculative execution

Consider the following program:

```
x = a + b;
y = c + d;
if (p) l = m + n;      // instruction 3: no dependencies
else z = x * y;        // depends on x and y
```

we don’t know yet if true or false

CPU is allowed to speculatively execute independent instruction 3
-> keep the processor's execution units busy!

Optimizing ILP – Reordering

Consider the following program:

```
result = f(x);
done = true;
```

Independent instructions may be reordered by both the compiler (at compile time) and the CPU (during execution).

Code example: L06_03_reordering

Optimizing ILP – Reordering

Consider the following program, executed by two threads:

```
4 result = f(x);          ||    2 while (done!=true);
1 done = true;            ||    3 print(result);
```

Core 1:
1) dependency analysis
→ independent
2) potential instruction reordering

Core 2:
1) dependency analysis
→ independent
2) potential instruction reordering

Sequential code: smooth sailing. Parallel code: things get trickier.

Programmers must use synchronization when accessing shared data (results, done) to prevent reorderings that break correctness in multithreaded programs.

Optimizing ILP – Reordering

Consider the following program, executed by two threads:

```
synchronized {
  result = f(x);
  done = true;
}
```

||

```
synchronized {
   while (done!=true);
   print(result);
}
```

Note: busy waiting under a lock can cause a deadlock! use wait / notify

Programmers must use synchronization when accessing shared data (results, done) to prevent reorderings that break correctness in multithreaded programs.

Stages of the CPU pipeline

- Reads instructions starting at the address stored in the instruction pointer
- May fetch multiple instructions ahead of time (prefetching) to support superscalar and speculative execution

Stages of the CPU pipeline

- Prepares instructions for execution, e.g.
  - Decodes bit sequence `01101…` into `ADD A1 A2`
  - “Understands” registers denoted by `A1/A2`

  (RISC instructions either transfer data between memory and CPU registers, or compute on registers)
- Optimized by immediate dispatch of instructions to different execution units, SIMD can optimize decode process for vectorized instructions

Stages of the CPU pipeline

- Executes decoded instruction (in CPU, no data transfer yet)
- Superscalar execution: allows multiple instructions to be dispatched to multiple execution units simultaneously. This means more than one instruction can be executed per clock cycle
- SIMD: can allow a single instruction to execute on multiple data elements simultaneously

Stages of the CPU pipeline

- Exchange data between CPU and memory
  (if needed, depending on executed instruction)
- Memory hierarchy, caching, pre-fetching

Stages of the CPU pipeline

- Updates registers
  (if needed, depending on executed instruction; also special registers such as flags)
- Superscalar Execution: Multiple instructions can write their results back to different registers or memory locations at the same time

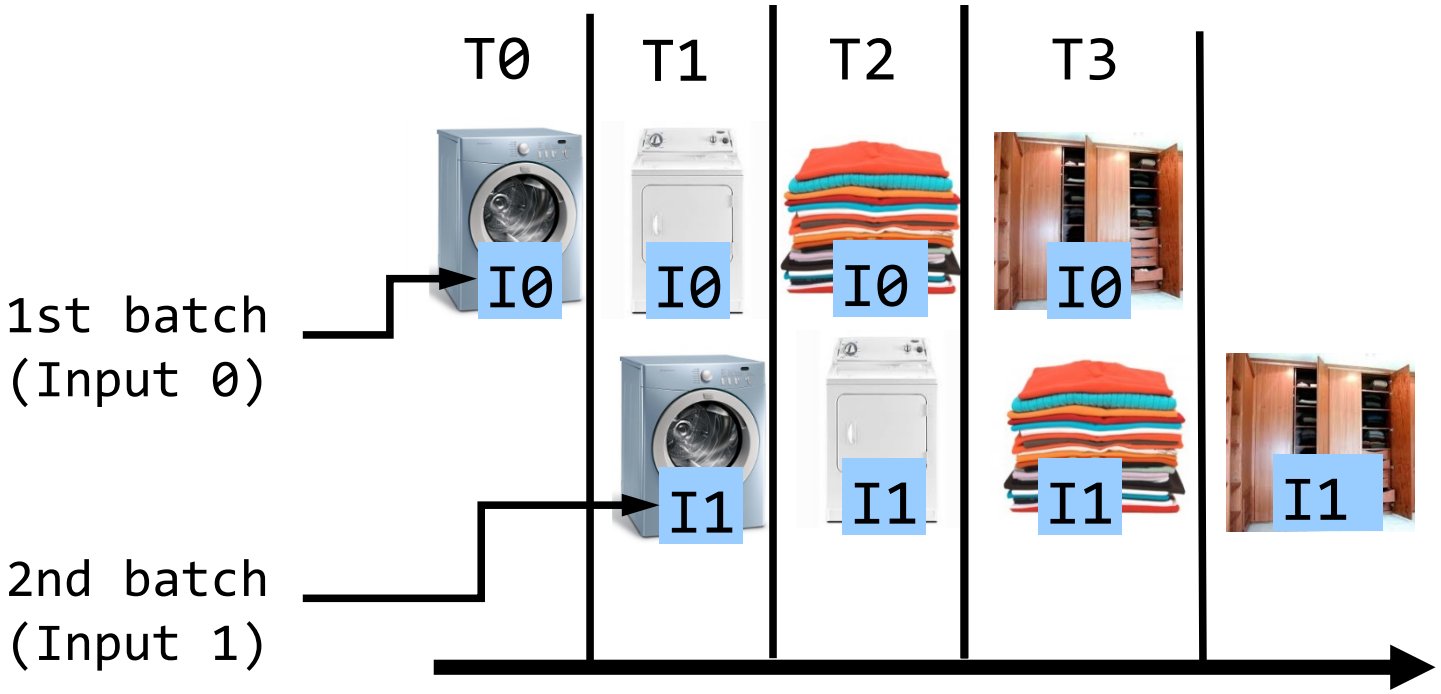
Stages of the CPU pipeline

Instruction 1

- Pipelining increases instruction throughput
- Multiple instructions occupy different pipeline stages simultaneously
- Additional execution units allow parallel execution within stages

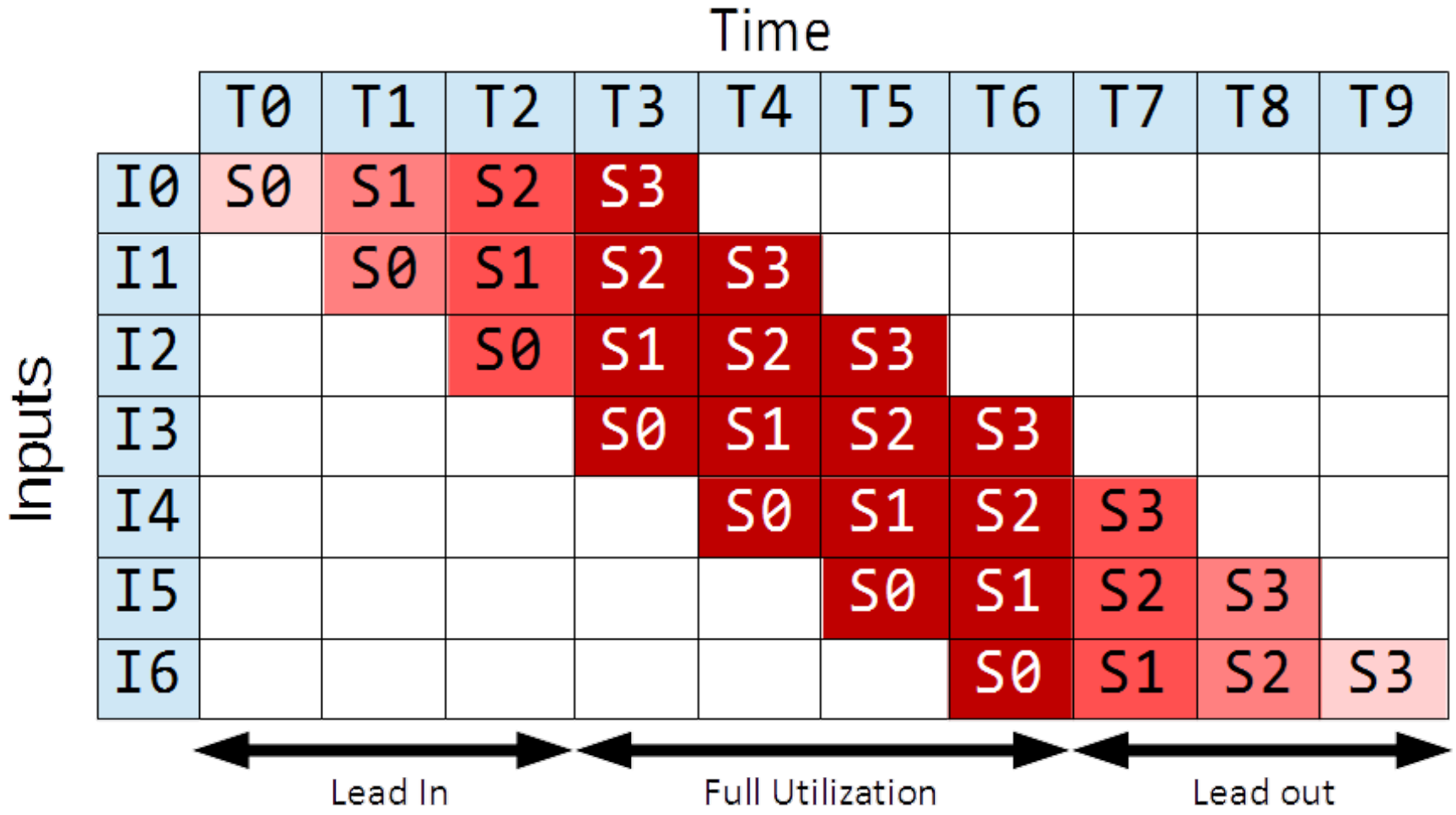
Pipelining: Washing clothes

Pipelining: Washing clothes



Balanced pipeline

balanced = all steps require same time



Pipeline characteristics / metrics

- **Throughput** = amount of work that can be done by a system in a given period of time
- **Latency** = time needed to perform a given computation (e.g., a CPU instruction)

More exists, e.g. bandwidth (amount of work done in parallel)

Throughput

- Throughput = amount of work that can be done by a system in a given period of time
- In CPUs: # of instructions completed per second
- Larger is better

$$\text{Throughput bound} = \frac{1}{\max\left(computationtime(stages)\right)}$$

(ignoring lead-in and lead-out time in pipeline with large number of states; cannot do better than this)

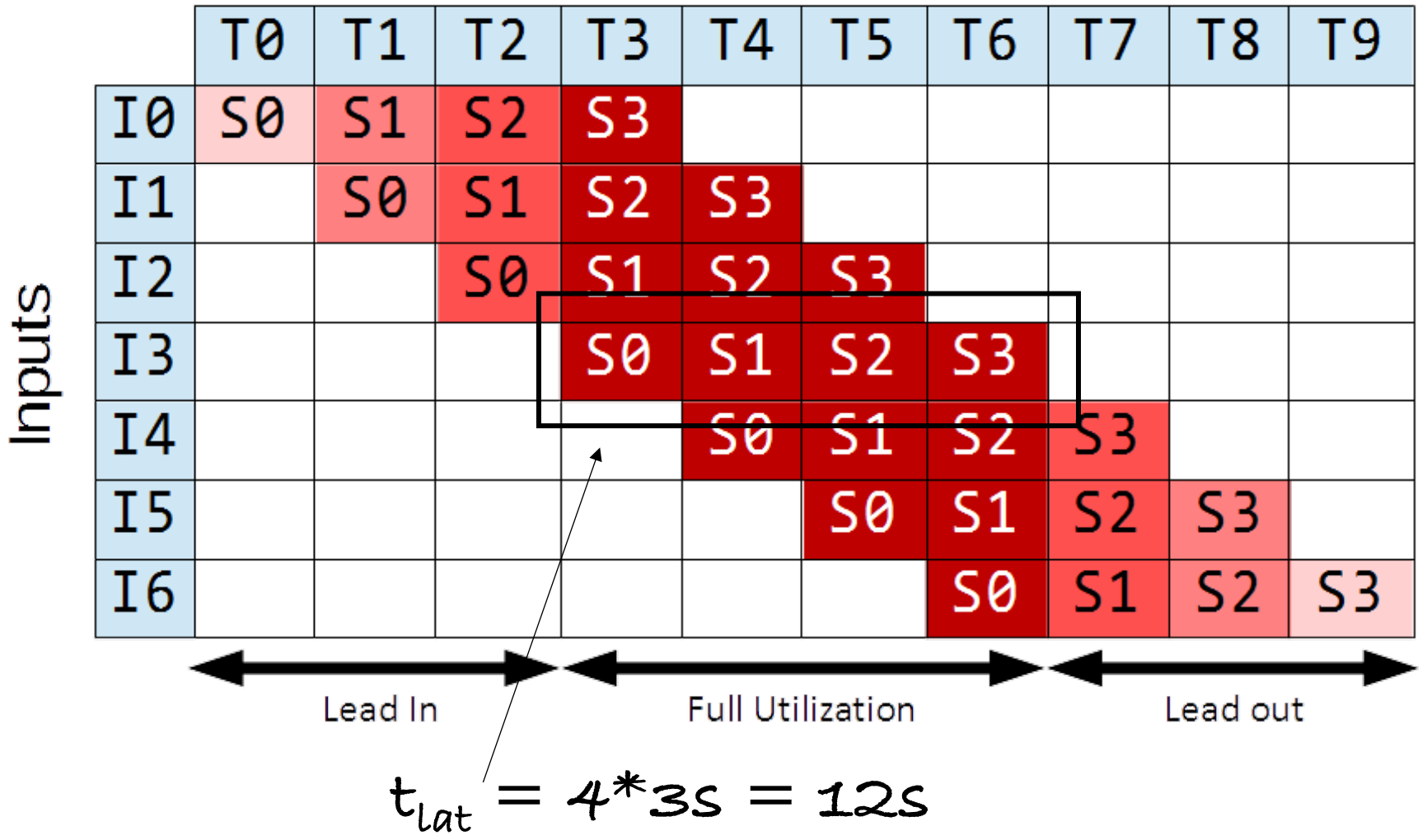
Balanced pipeline

balanced = all steps require same time

$T_i = 3s$

$t_{put} = 1/3s \rightarrow 1$ instruction per 3s

7 instructions -> 21s



here:

10*3=30s

(lead-in / -out)

$t_{lat} = 4*3s = 12s$

Latency

- Latency = time to perform a computation (e.g., a CPU instruction)
- In CPU: time required to execute a single instructions in the pipeline
- Lower is better

Latency bound = $\#\text{stages} \cdot \max\big(computationtime(stages)\big)$

- Pipeline latency **constant over time if pipeline balanced**: sum of execution times of each stage

Washing clothes – Unbalanced pipeline

Takes 5 seconds. We use “w” for Washer next.

Takes 10 seconds. We use “d” for Dryer next.

Takes 5 seconds. We use “f” for Folding next.

Takes 10 seconds. We use “c” for Closet next.

Designing a pipeline: 1st attempt (lets consider 5 washing loads)

Time (s) / Load #	0	5	10	15	20	25	30	35	40	45	50	55	60	65	70
Load 1	w	d	d	f	c	c									
Load 2		w	_	d	d	f	c	c							
Load 3			w	_	_	d	d	f	c	c					
Load 4				w	_	_	_	d	d	f	c	c			
Load 5					w	_	_	_	_	d	d	f	c	c	

The total time for all 5 loads is **70 seconds**.

This pipeline can work, however it cannot bound the latency of a Load as it keeps growing. If we want to bound this latency, one approach is to make each stage take as much time as the longest one, thus balancing it. In our example, the longest time is 10 seconds, so we can do the following:

Make Pipeline balanced by increasing time for each stage to match longest stage

Now takes 10 seconds.

Takes 10 seconds, as before.

Now takes 10 seconds.

Takes 10 seconds, as before.

Designing a pipeline: 2nd attempt

Time (s) / Load #	0	10	20	30	40	50	60	70	80	90	100	110	60	65	70	75	80	85	90
Load 1	w	d	f	c															
Load 2		w	d	f	c														
Load 3			w	d	f	c													
Load 4				w	d	f	c												
Load 5					w	d	f	c											

This pipeline is a bit wasteful, but the latency is bound at **40 seconds** for each Load. Throughput here is about 1 load / 10 seconds, so about 6 loads / minute.

So now we have the total time for all 5 loads at **80 seconds**, higher than before.

Can we somehow get a bound on latency while improving the time/throughput?

Step 1: make the pipeline from 1st attempt a bit more fine-grained

Like in the 1st attempt, this takes 5 seconds.

Let's have 2 dryers working in a row. The first dryer is referred to as **d1** and takes 4 seconds, the second as **d2** and takes 6 sec.

Like in the 1st attempt, it takes 5 seconds.

Let's have 2 closets working in a row. The first closet is referred to as **c1** and takes 4 seconds, the second as **c2** and takes 6 sec.

Note: splitting slow stages into two pipeline stages, not parallel dryers / cupboards

Step 2: and also, like in the 2nd pipeline, make each stage take as much time as the longest stage does from Step 1 [this is 6 seconds due to d2 and c2]

It now takes 6 seconds.

Each of d1 and d2 dryers take 6 seconds.

Now takes 6 seconds.

Each of c1 and c2 closets now take 6 seconds.

Note: splitting slow stages into two pipeline stages, not parallel dryers / cupboards

Designing a pipeline: 3rd attempt (let's consider 5 washing loads)

Time (s) / Load #	0	6	12	18	24	30	36	42	48	54	60	110	60	65	70	75	80	85	90
Load 1	w	d1	d2	f	c1	c2													
Load 2		w	d1	d2	f	c1	c2												
Load 3			w	d1	d2	f	c1	c2											
Load 4				w	d1	d2	f	c1	c2										
Load 5					w	d1	d2	f	c1	c2									

The bound on latency for each load is now: 6 * 6 = 36 seconds.

The throughput is approximately: 1 load / 6 seconds = ~ 10 loads / minute.

The total time for all 5 loads is **60 seconds**.

Throughput vs. latency

Throughput optimization may increase the latency: in our 3rd pipeline attempt, we split the dryers into 2, but it could be that the split of ‘d’ into d1 and d2 leads to higher times for d1 and d2 than 4 and 6.

Pipelining typically adds constant time overhead between individual stages (synchronization, communication)

→ Infinitely small pipeline steps not practical

→ Time it takes to get one complete task through the pipeline may take longer than with a serial implementation

Take-away

- Modern CPUs exploit parallelism internally
  (caching, vectorization, ILP, pipelining)
- Programmers do not control these mechanisms...
- ...but code structure determines how effective they are
- These effects exist even on a single core