September 11, 2025

6.S894
Accelerated Computing
Lecture 2: Throughput Processors

Jonathan Ragan-Kelley

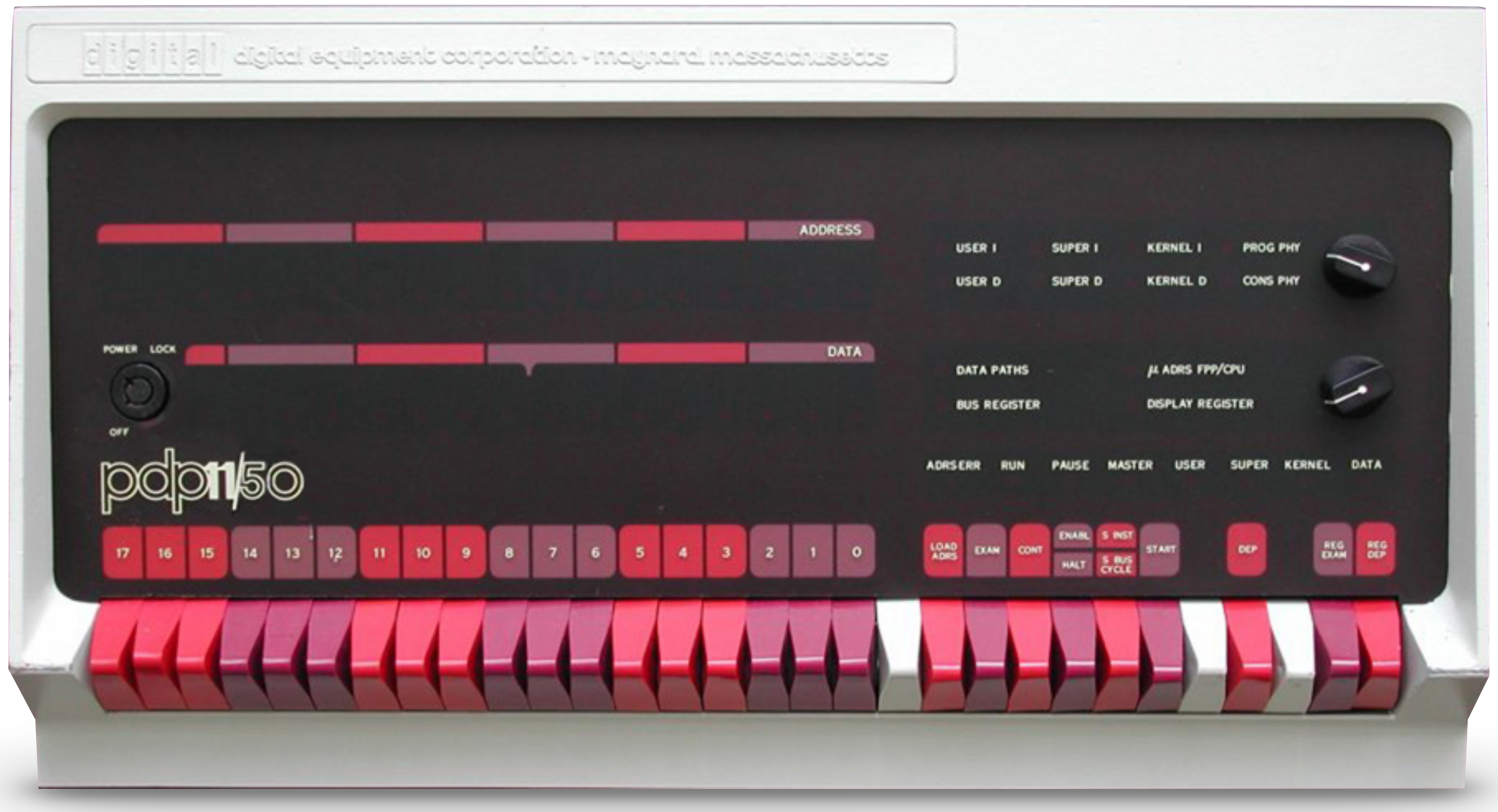
What is a **processor?**

a **programmable** computer

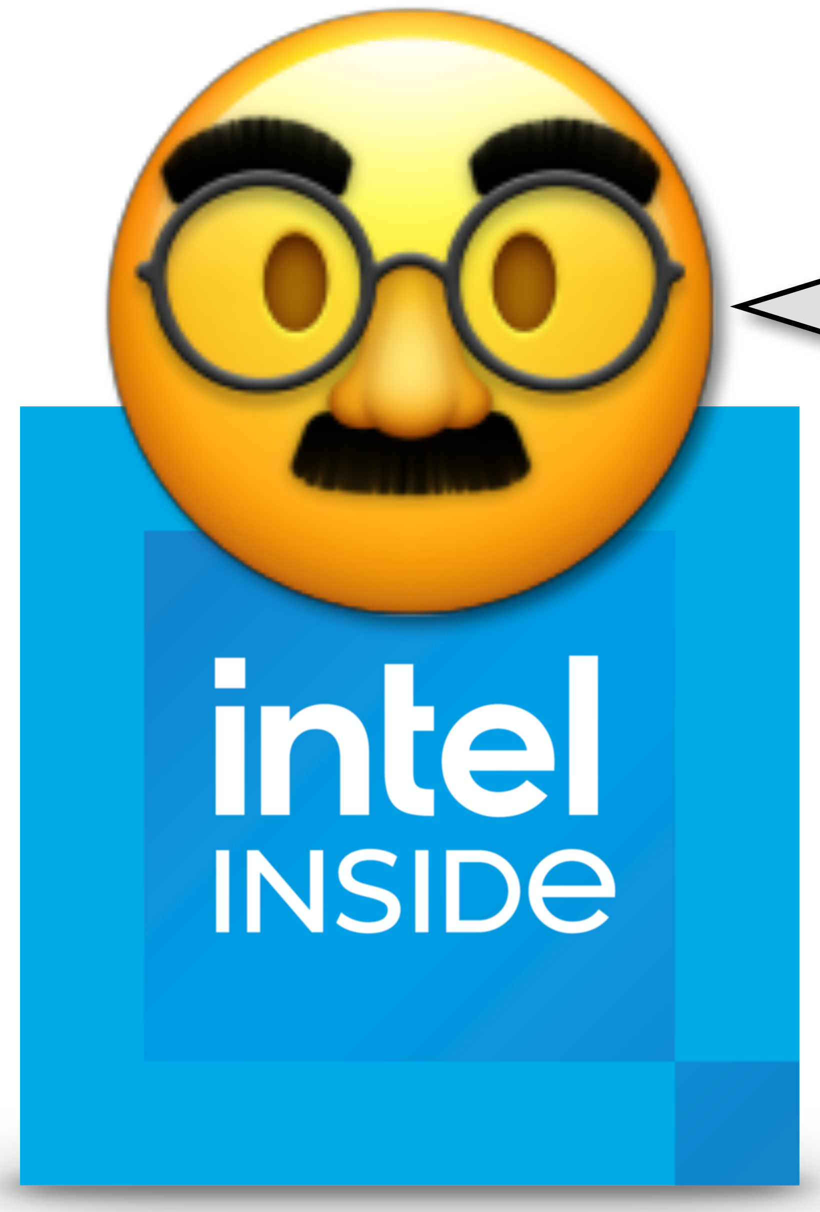
that runs a **sequence of instructions** over time

including control flow, computation & state updates

What is a **processor?**



1970s



Why yes,
I am still
basically a
PDP-11

2020s

A simple processor

an **interpreter** for instructions!

1. Instr. fetch
2. Decode
3. Operand fetch
4. Execute
5. Write back

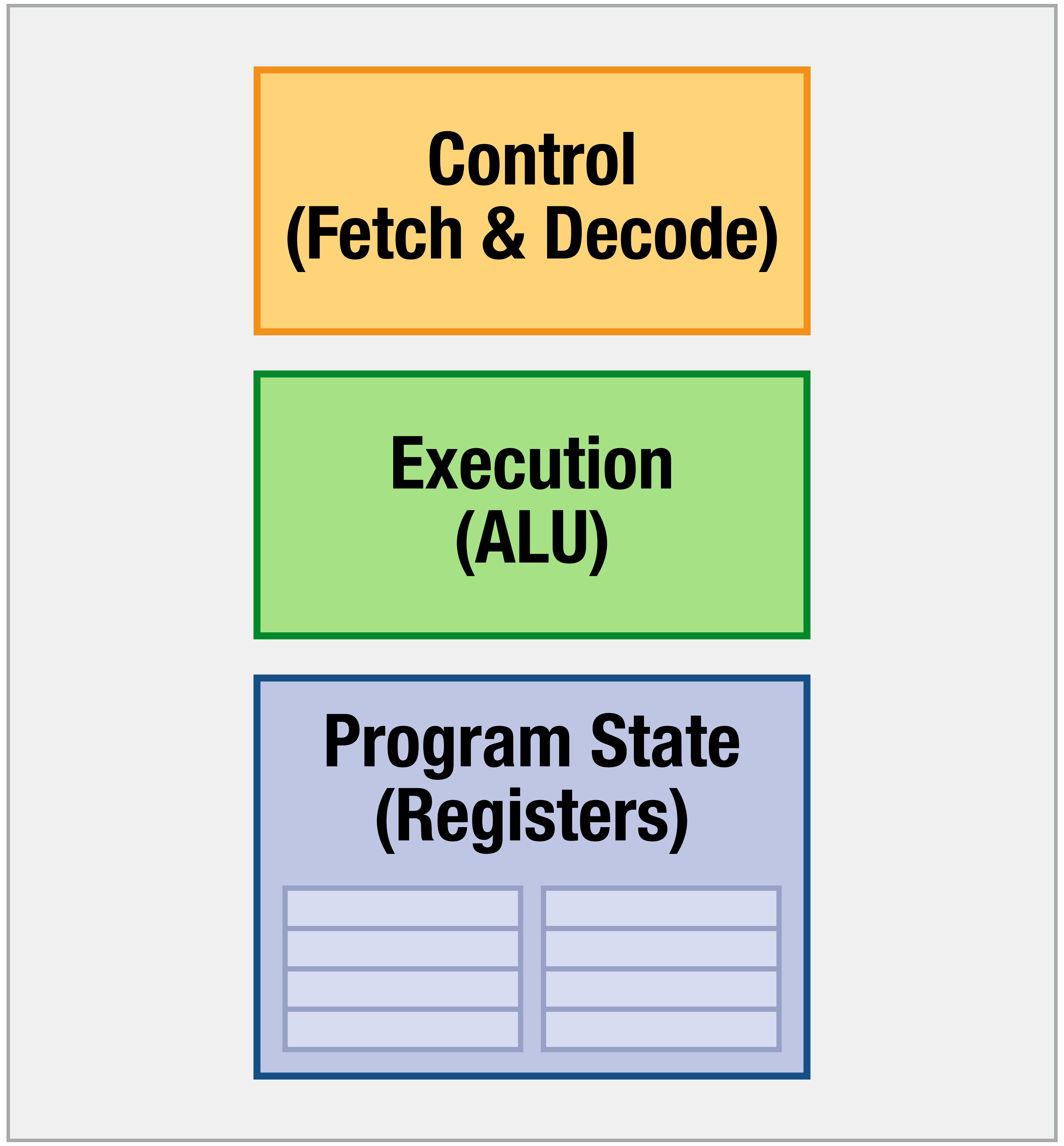
Processor

```
ld      r3, mem[r0+r2]
ld      r4, mem[r1+r2]
mul     r3, r3, r4
add     r5, r5, r3
addi    r2, r2, 4
blt     r2, $400, LOOP
st      addr[r6], r5
...
```

Program

Slides inspired by Kayvon Fatahalian

A simple processor an **interpreter** for instructions!



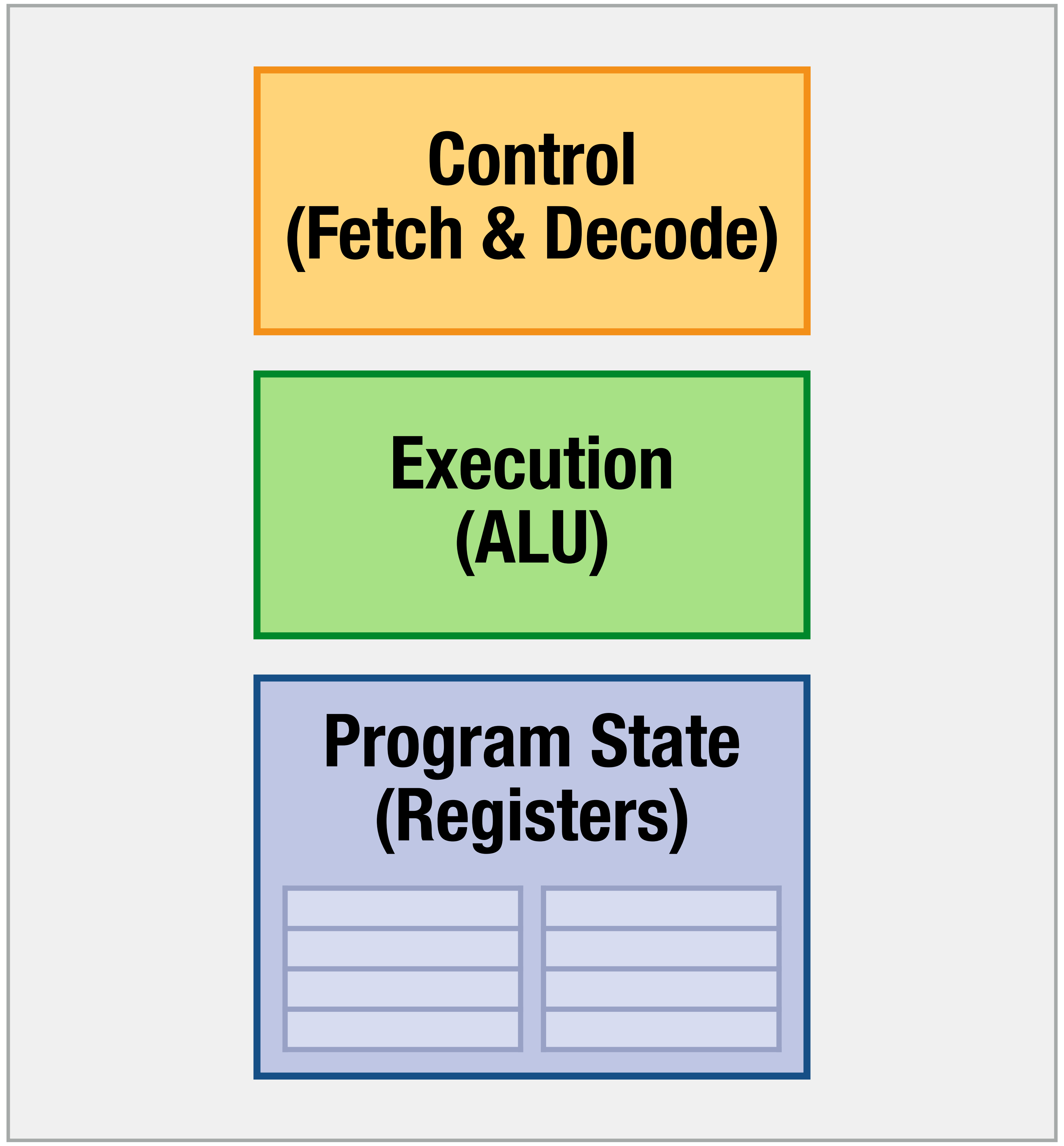
Processor

```
ld      r3, mem[r0+r2]
ld      r4, mem[r1+r2]
mul     r3, r3, r4
add     r5, r5, r3
addi    r2, r2, 4
blt     r2, $400, LOOP
st      addr[r6], r5
...
```

Program

Slides inspired by Kayvon Fatahalian

Optimization 1: Increase the **clock speed**



Processor

Higher **voltage**

↳ power grows with v^2

Faster **transistors**

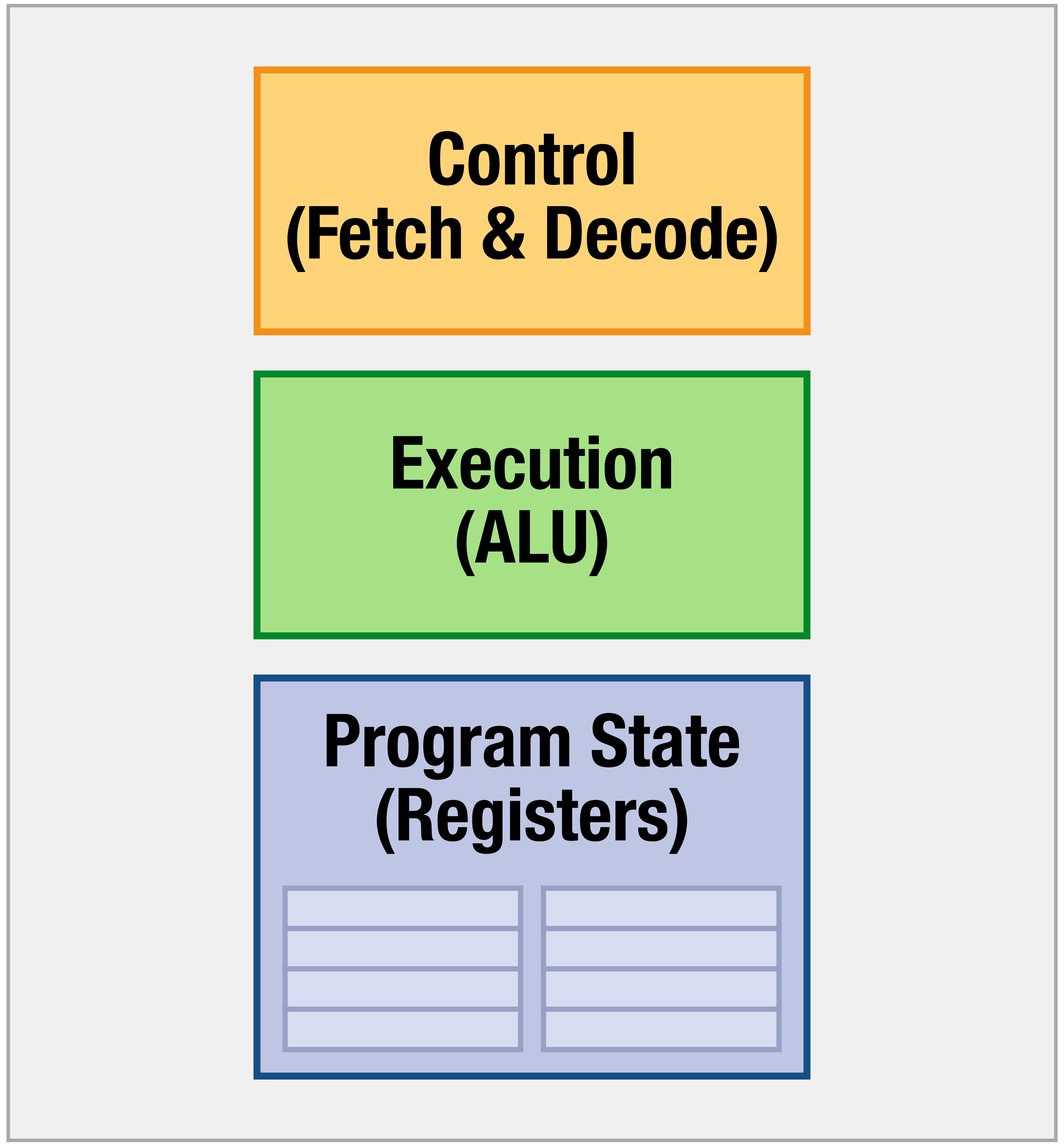
↳ higher leakage

Deeper **pipelining**

↳ lower average IPC

Slides inspired by Kayvon Fatahalian

Optimization 2: Execute **multiple** instructions per cycle (**superscalar**)

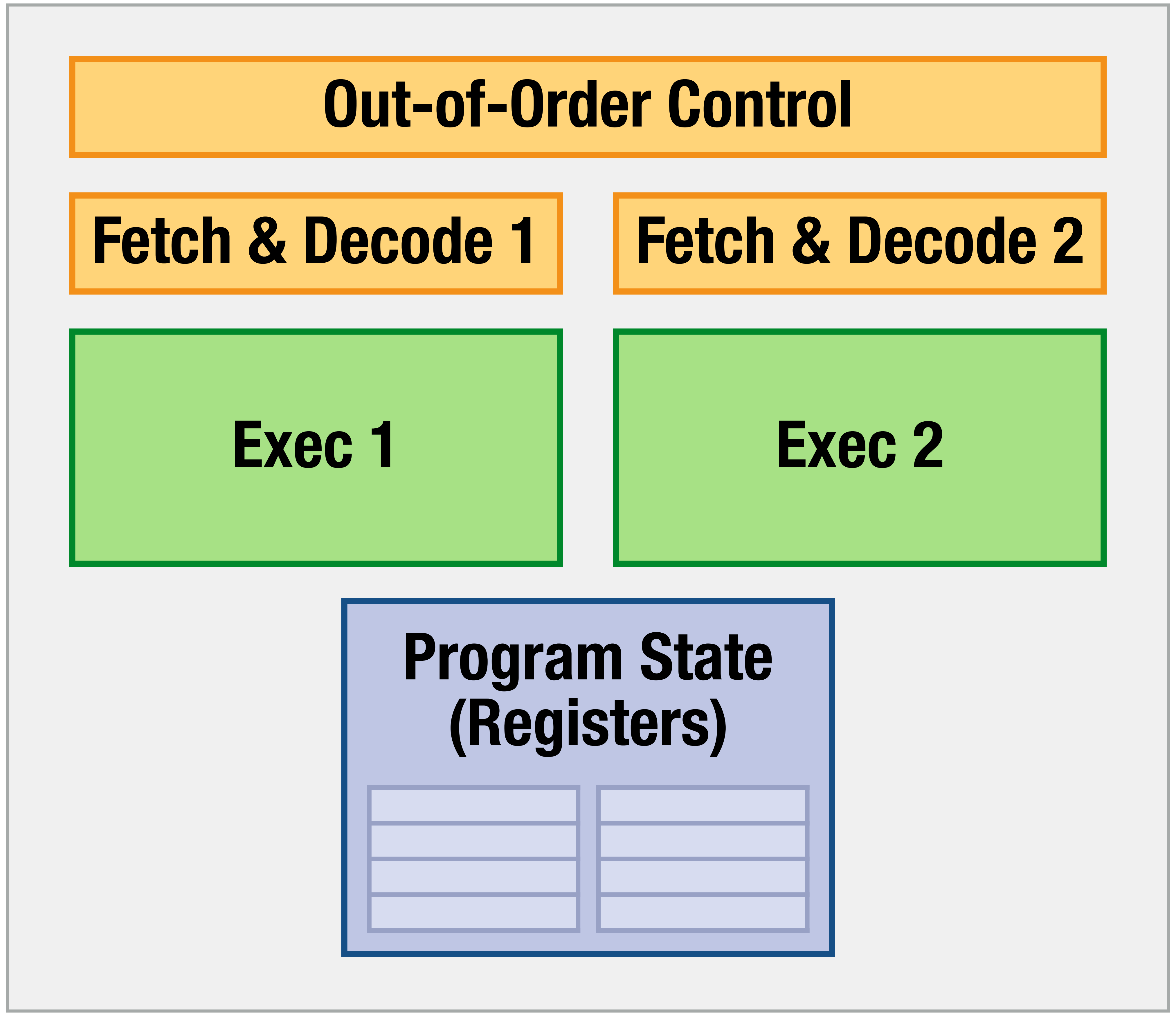


Processor

```
ld      r3, mem[r0+r2]
ld      r4, mem[r1+r2]
mul     r3, r3, r4
add     r5, r5, r3
addi    r2, r2, 4
blt     r2, $400, LOOP
st      addr[r6], r5
...
```

Slides inspired by Kayvon Fatahalian

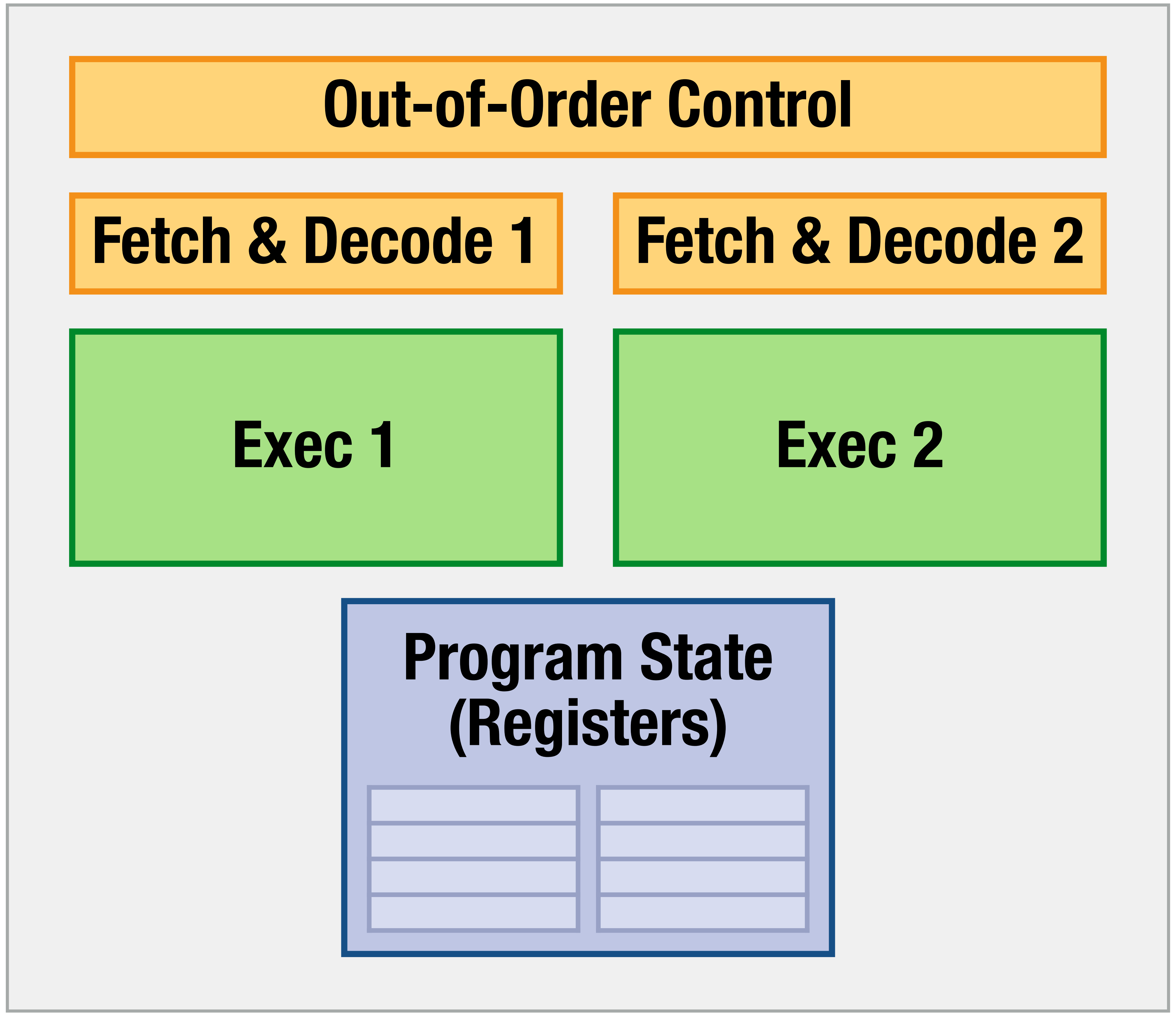
Optimization 2: Execute **multiple** instructions per cycle (**superscalar**)



Processor

```
ld      r3, mem[r0+r2]
ld      r4, mem[r1+r2]
mul     r3, r3, r4
add     r5, r5, r3
addi    r2, r2, 4
blt     r2, $400, LOOP
st      addr[r6], r5
...
```

Optimization 3: Avoid **stalls** through **ILP** & **speculation**

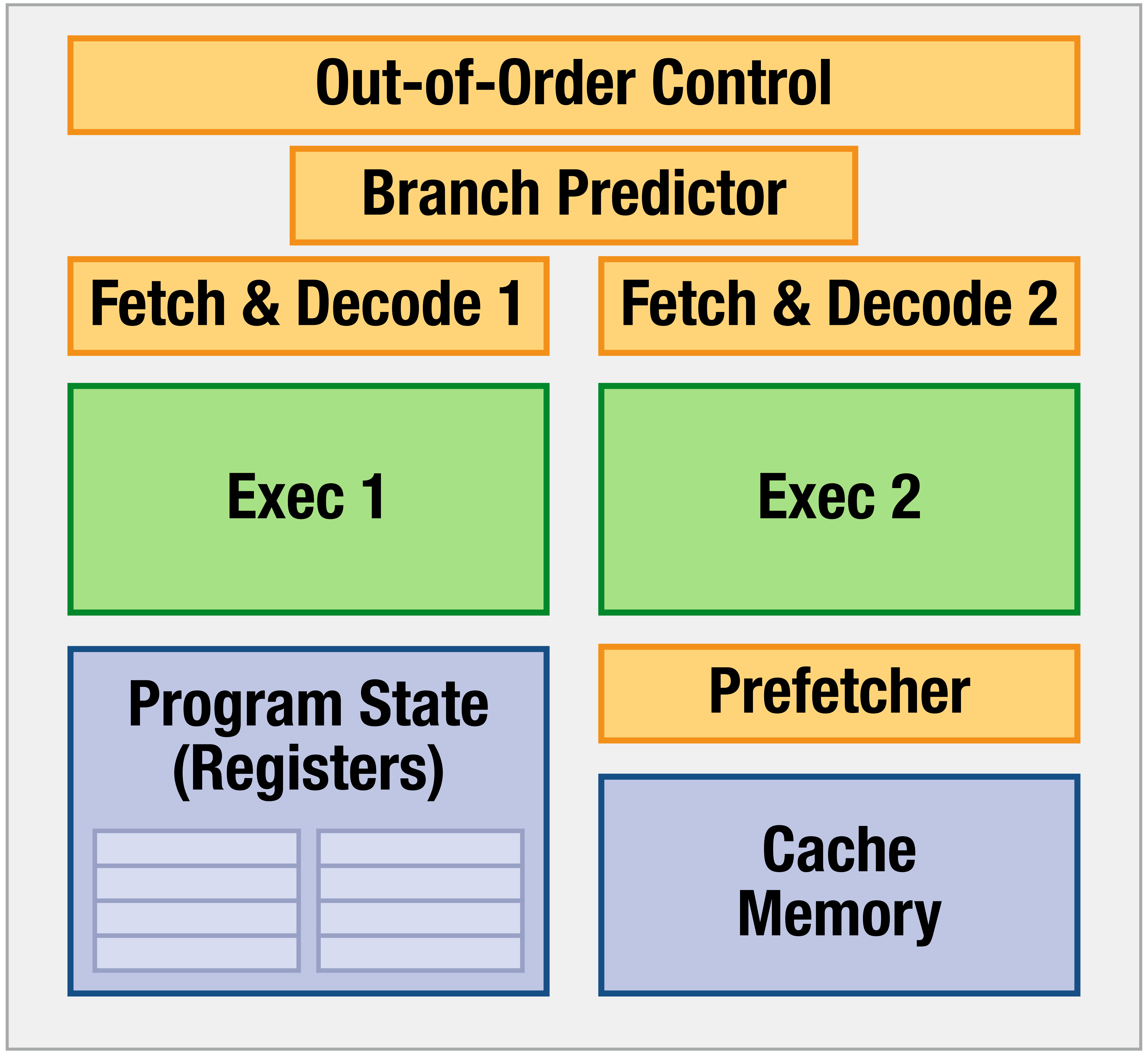


Processor

```
ld      r3, mem[r0+r2]
ld      r4, mem[r1+r2]
mul     r3, r3, r4
add     r5, r5, r3
addi    r2, r2, 4
blt     r2, $400, LOOP
st      addr[r6], r5
...
```

Slides inspired by Kayvon Fatahalian

Optimization 3: Avoid **stalls** through **ILP & speculation**

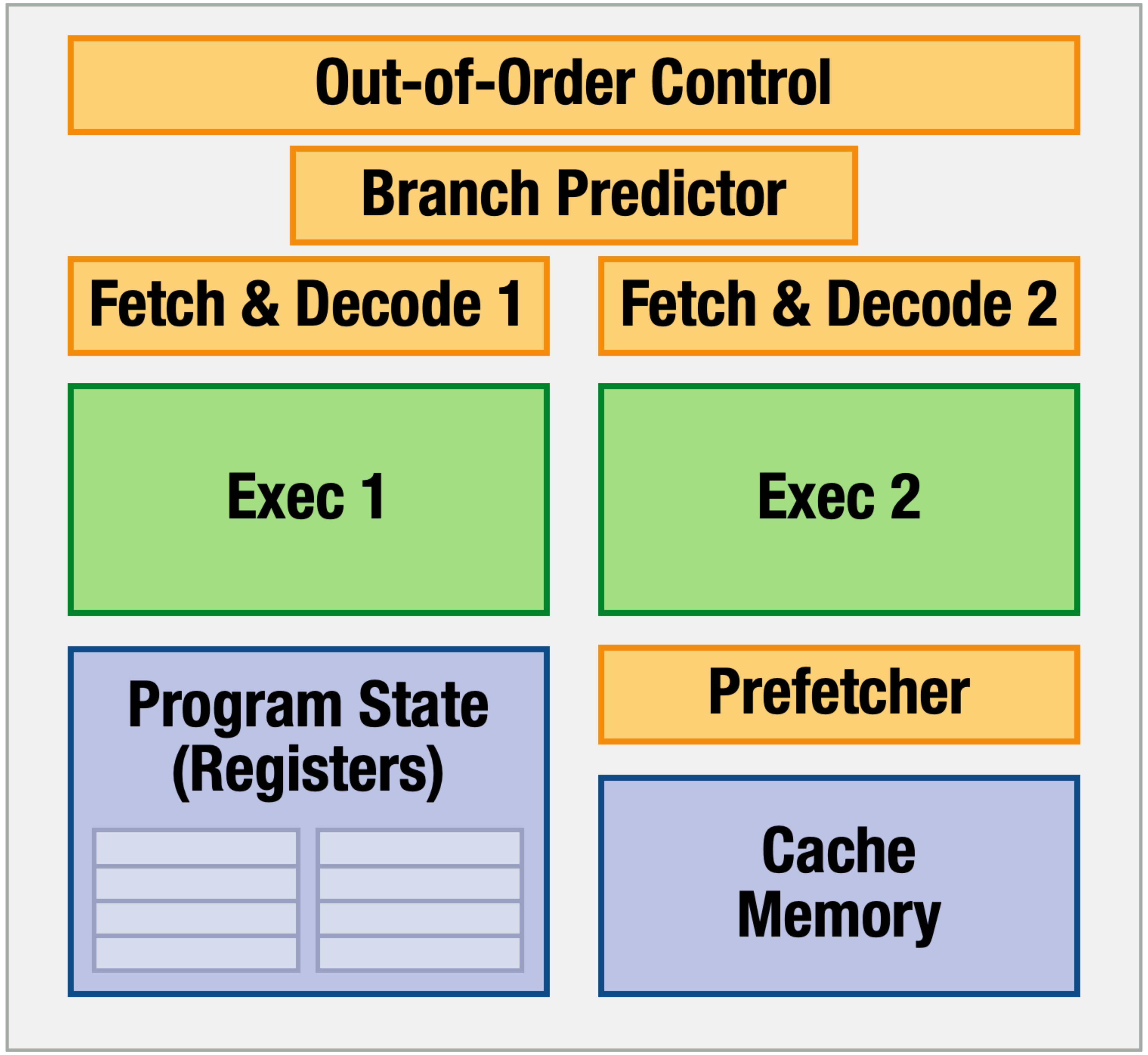


Processor

```
ld      r3, mem[r0+r2]
ld      r4, mem[r1+r2]
mul     r3, r3, r4
add     r5, r5, r3
addi    r2, r2, 4
blt     r2, $400, LOOP
st      addr[r6], r5
...
```

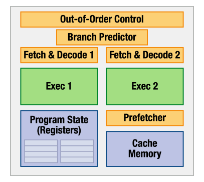
Slides inspired by Kayvon Fatahalian

This makes sense when you have only **one processor**



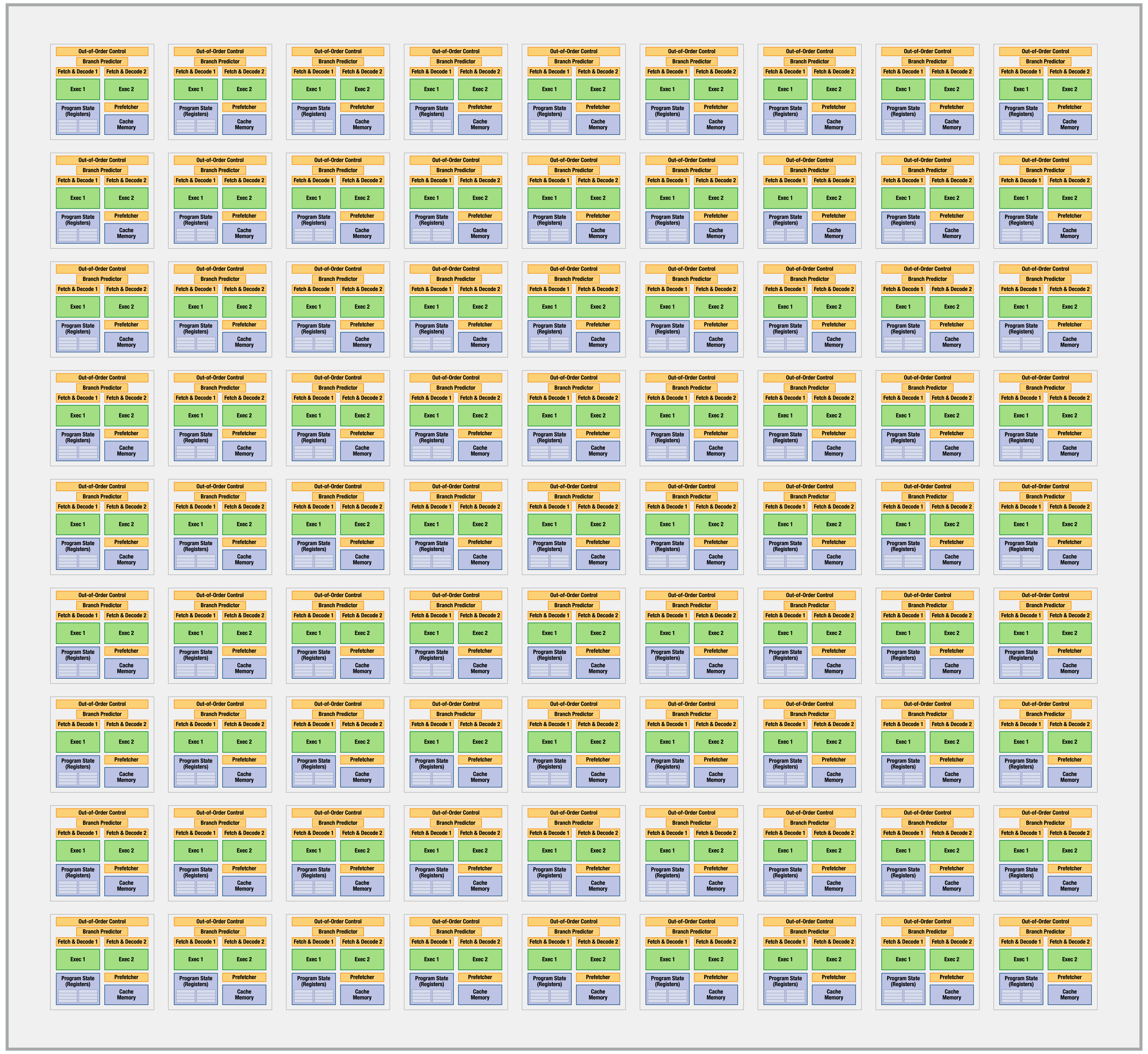
Processor

This makes sense when you have only **one processor**



Processor **(to scale)**

The reality today:



Silicon Chip

At the same time:

We **most need performance** when processing **lost of stuff**

↳ abundant **data-parallelism**

A **throughput-oriented** view of performance:

Optimize **aggregate rate** of processing **many items**

Three things drive throughput:

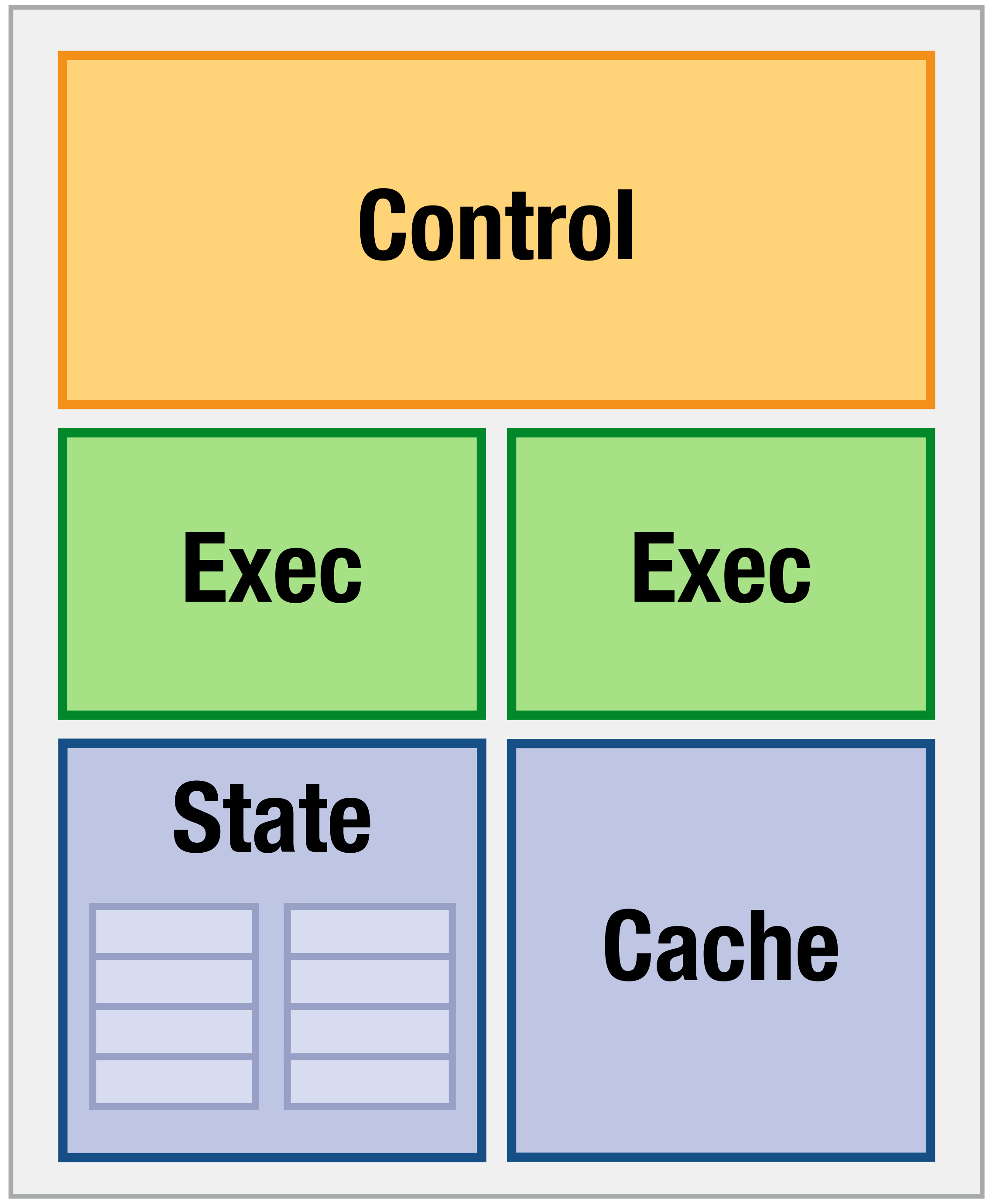
1. Amount of **work** to be done
2. Amount of **resources** to be applied (silicon, energy)

Constraints set by application, Si process node

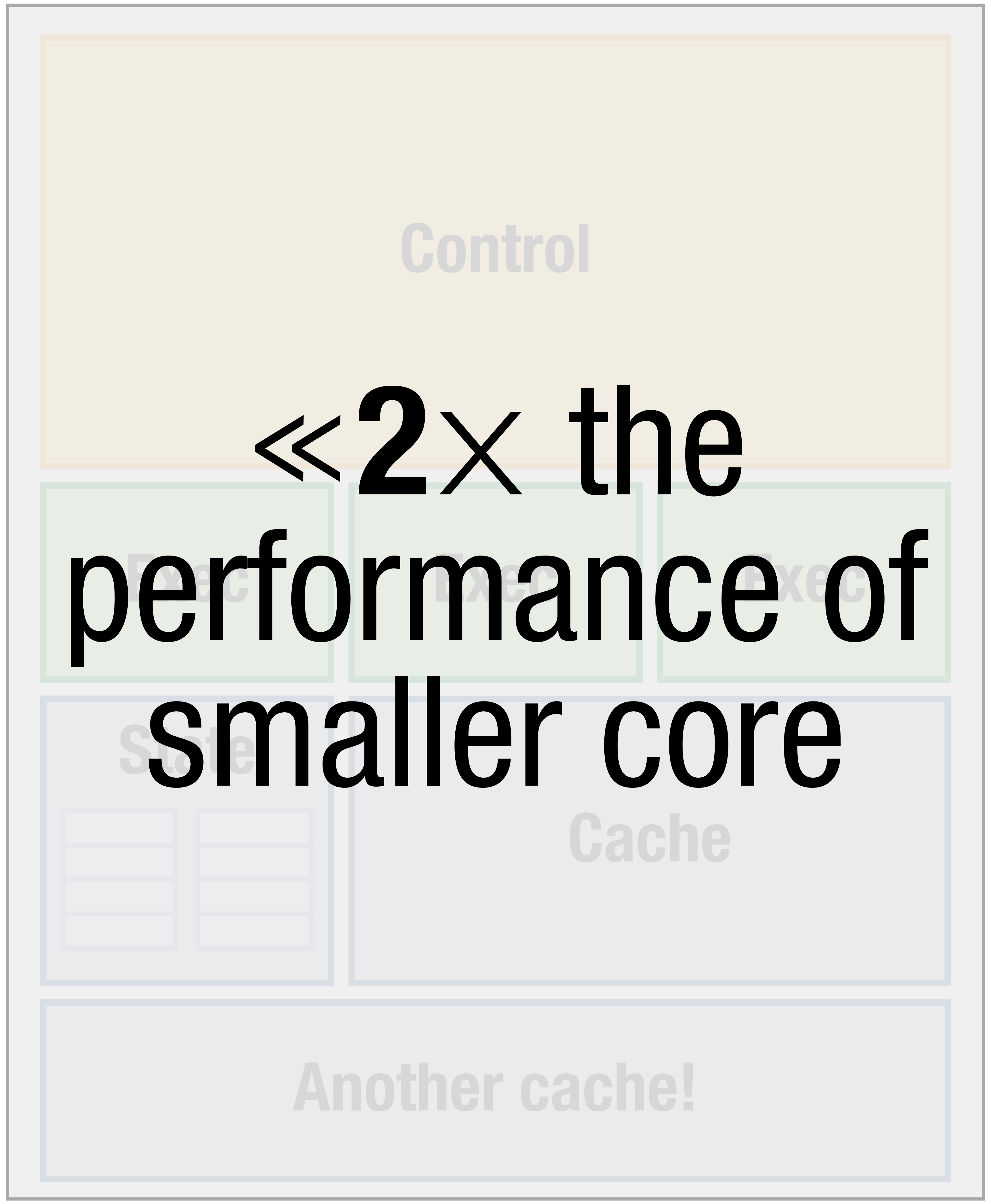
3. **Efficiency** of applying them to useful work

← Goal: **optimize this!**

What to do with twice the silicon?

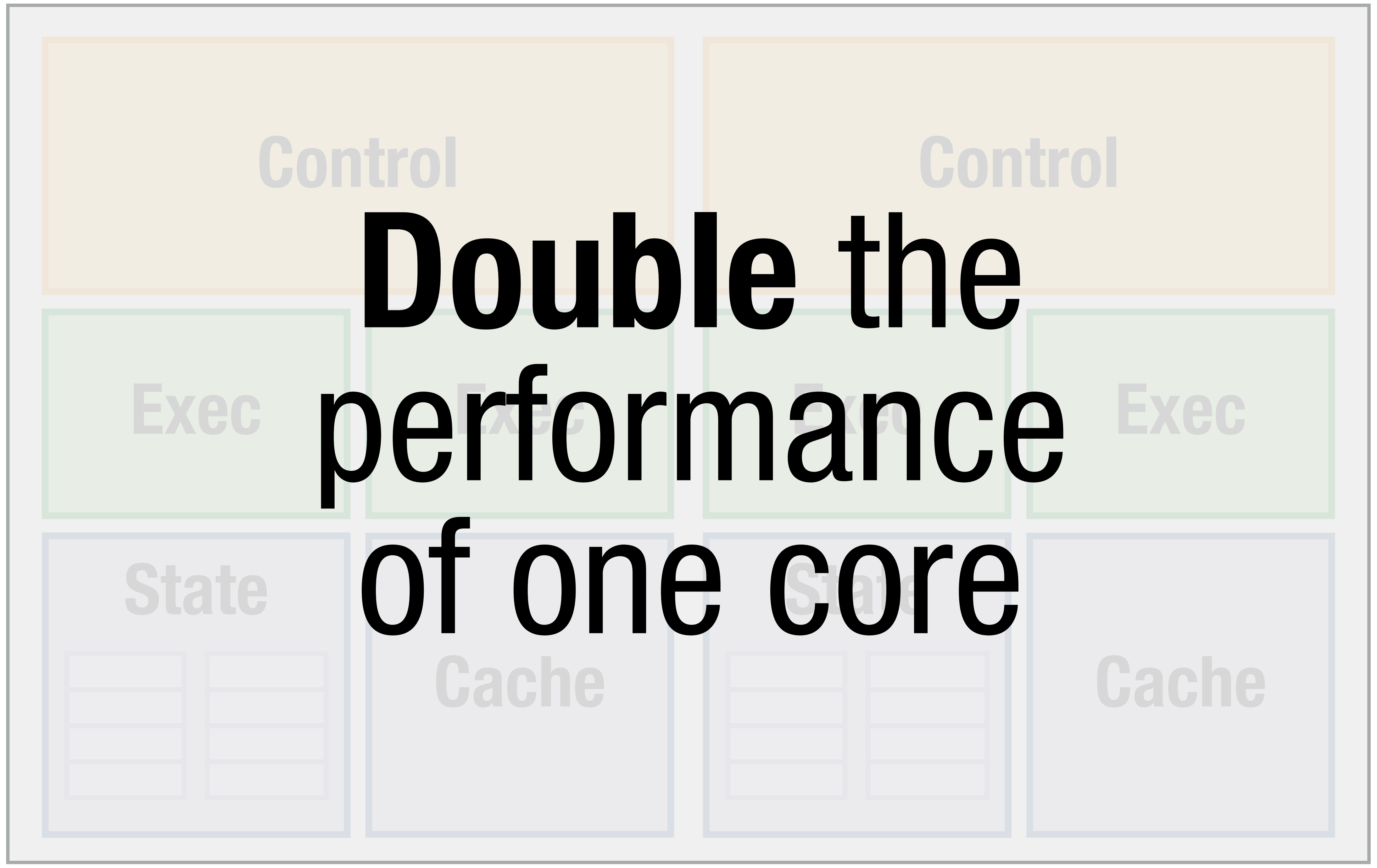


What to do with twice the silicon?

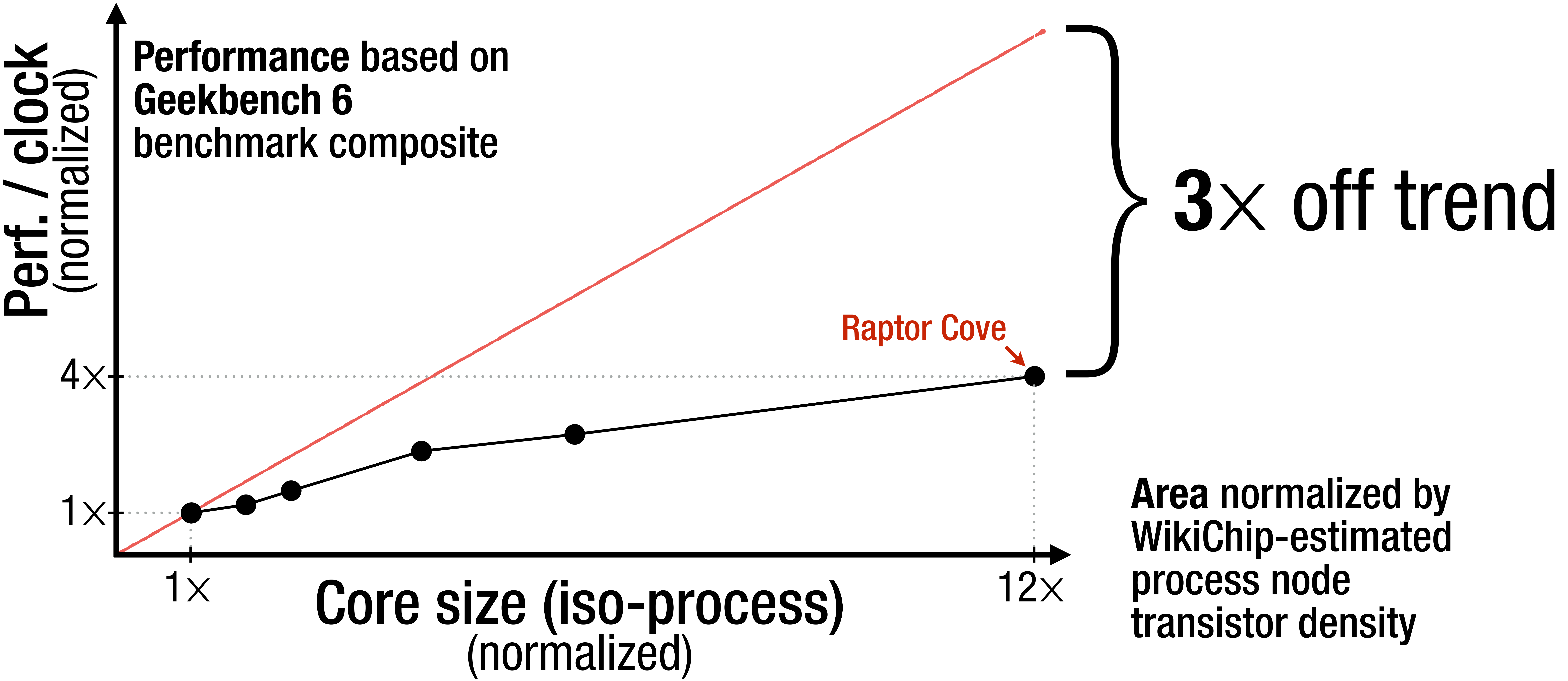


=

equal
area

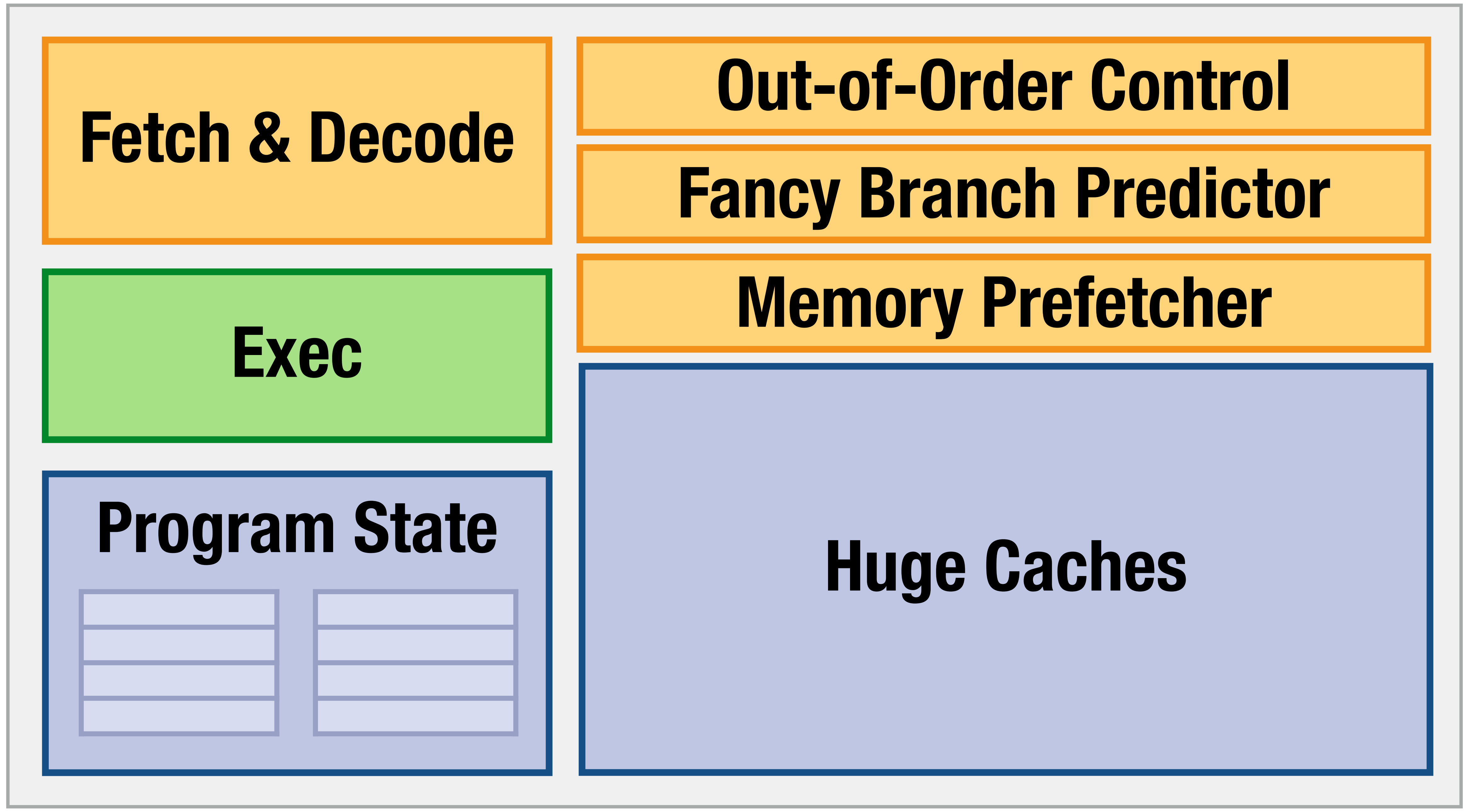


Diminishing returns to **scaling single-core** performance

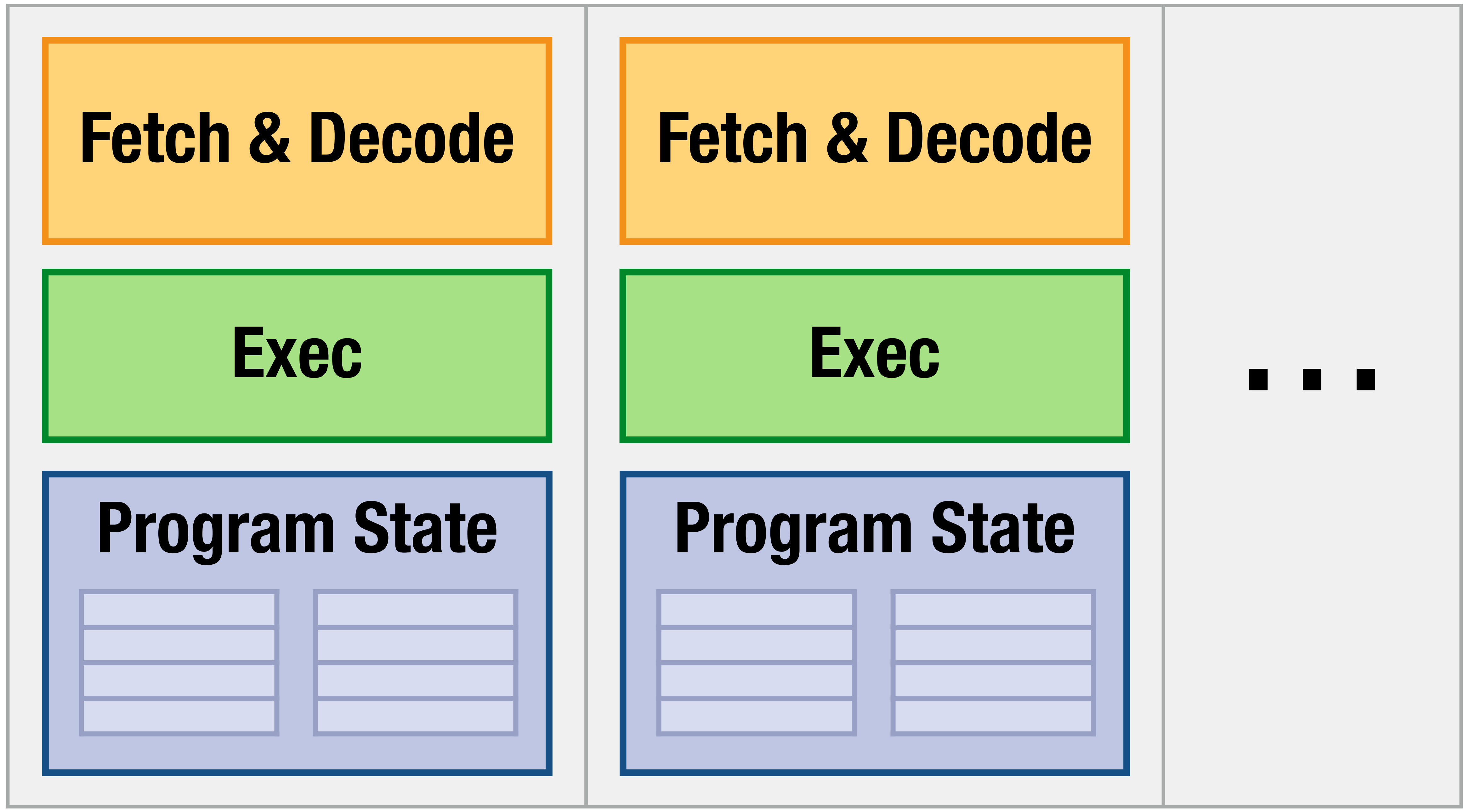


If we want to optimize **throughput,** is there a **better way** to **scale performance?**

Idea 1: remove hardware to optimize single-thread performance



Idea 1: remove hardware to optimize single-thread performance



Invest savings in **parallelism**

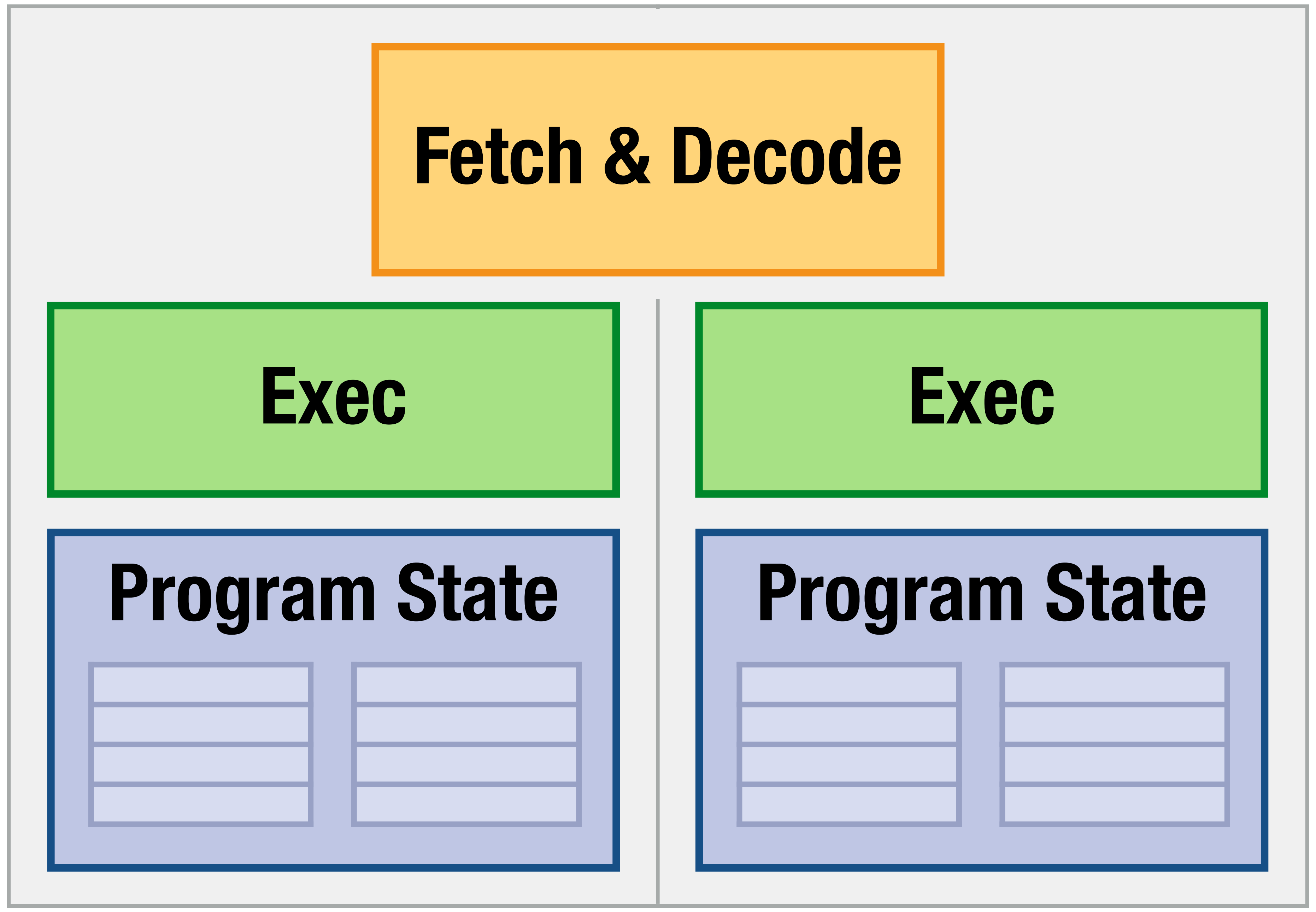
Slides inspired by Kayvon Fatahalian

```
for (int i = 0; i < N; ++i)
  for (int k = 0; k < N; ++k)
    for (int j = 0; j < N; ++j)
      C[i*N + j] += A[i*N + k] * B[k*N + j];
```

```
for (int j = 0; j < N; ++j)
  C[i*N + j] += A[i*N + k] * B[k*N + j];
```

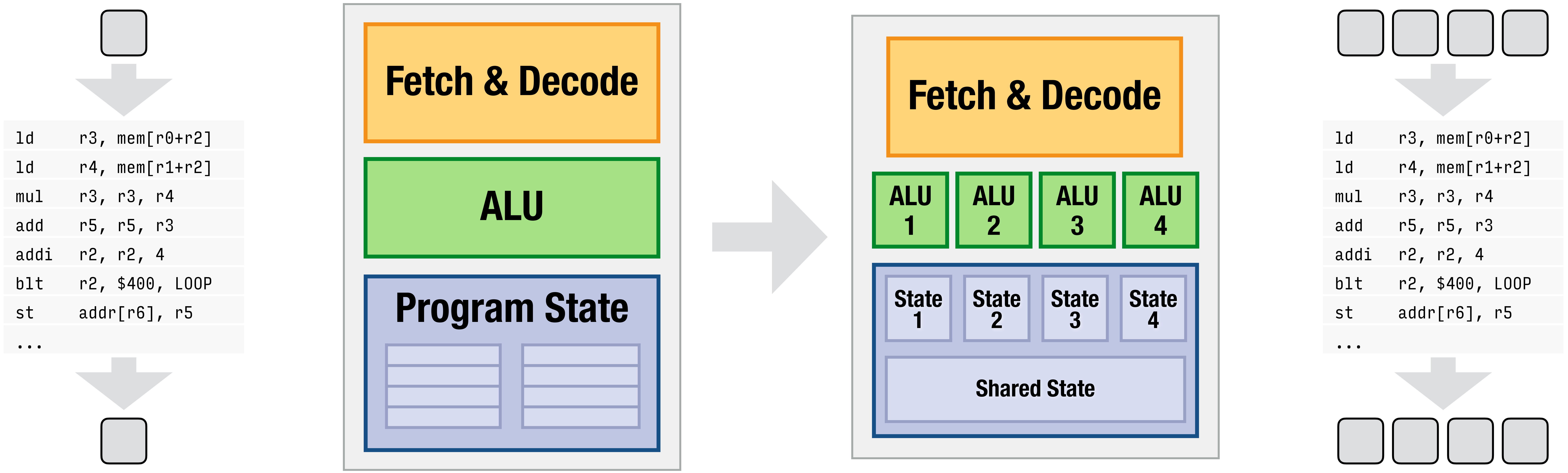
```
for all (int j = 0; j < N; ++j)
  C[i*N + j] += A[i*N + k] * B[k*N + j];
```

```
ld      r3, mem[r0+r2]
ld      r4, mem[r1+r2]
mul     r3, r3, r4
add     r5, r5, r3
addi    r2, r2, 4
blt     r2, $400, LOOP
st      addr[r6], r5
...
```




```
ld      r3, mem[r0+r2]
ld      r4, mem[r1+r2]
mul     r3, r3, r4
add     r5, r5, r3
addi    r2, r2, 4
blt     r2, $400, LOOP
st      addr[r6], r5
...
```

Idea 2: amortize **control overhead** with **SIMD execution**



Slides inspired by Kayvon Fatahalian

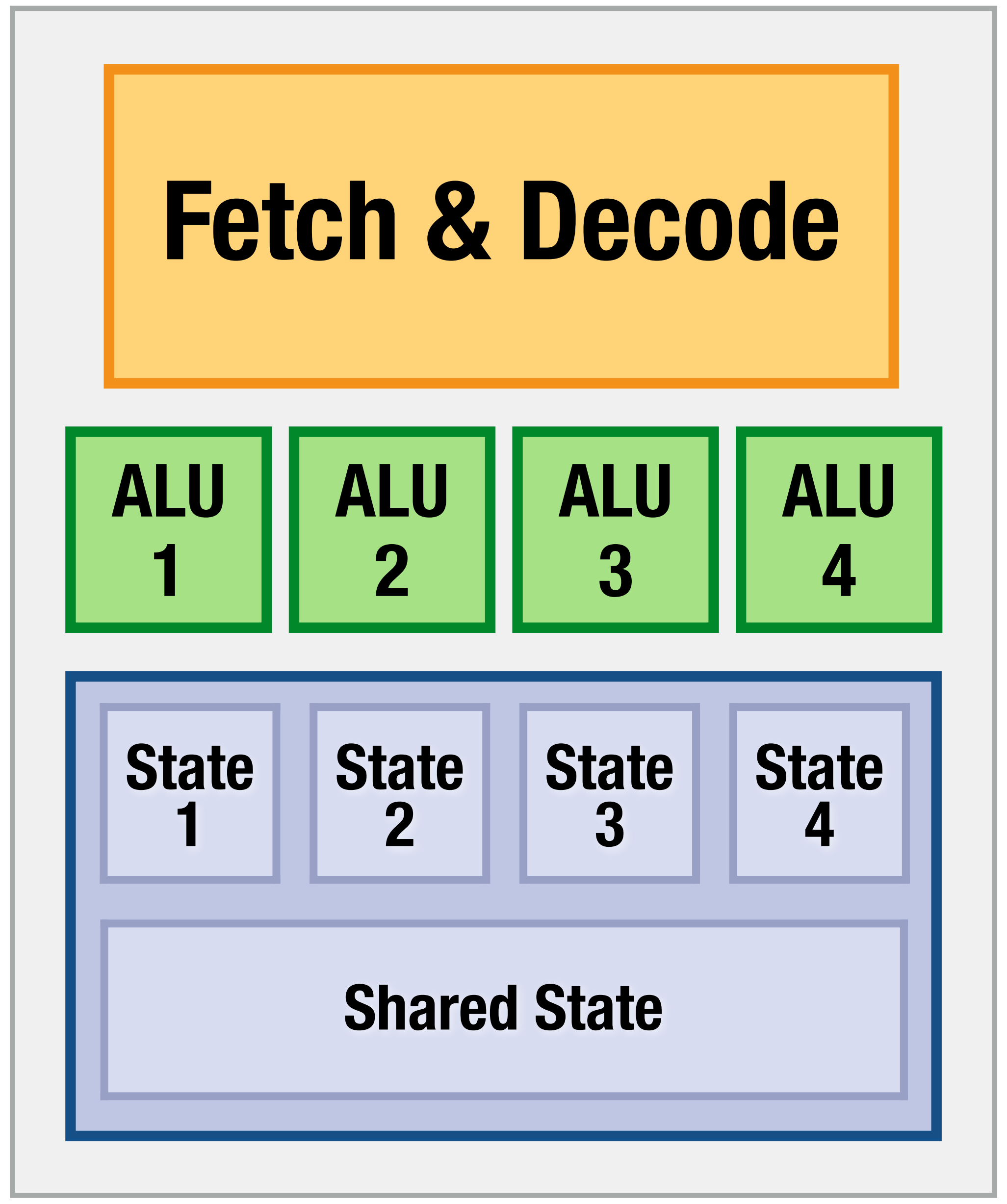
Idea 2: amortize **control overhead** with **SIMD execution**

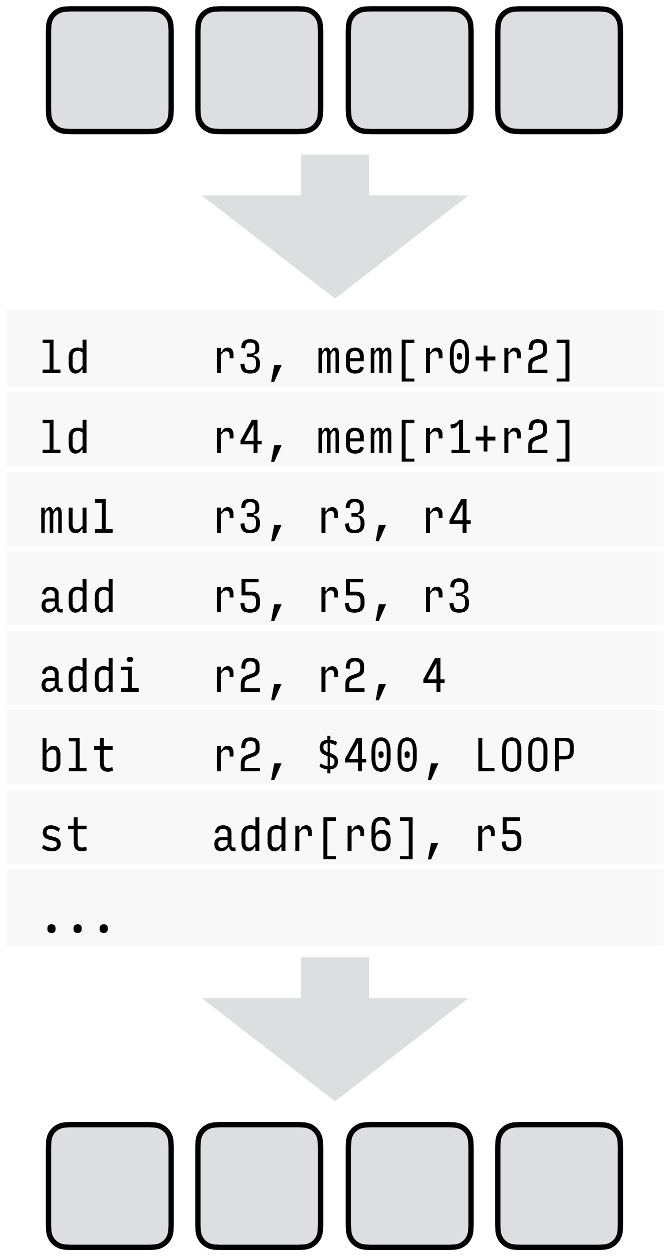
SIMD can be **explicit in ISA**

Intel AVX, ARM NEON, etc.

or **implicit in hardware**

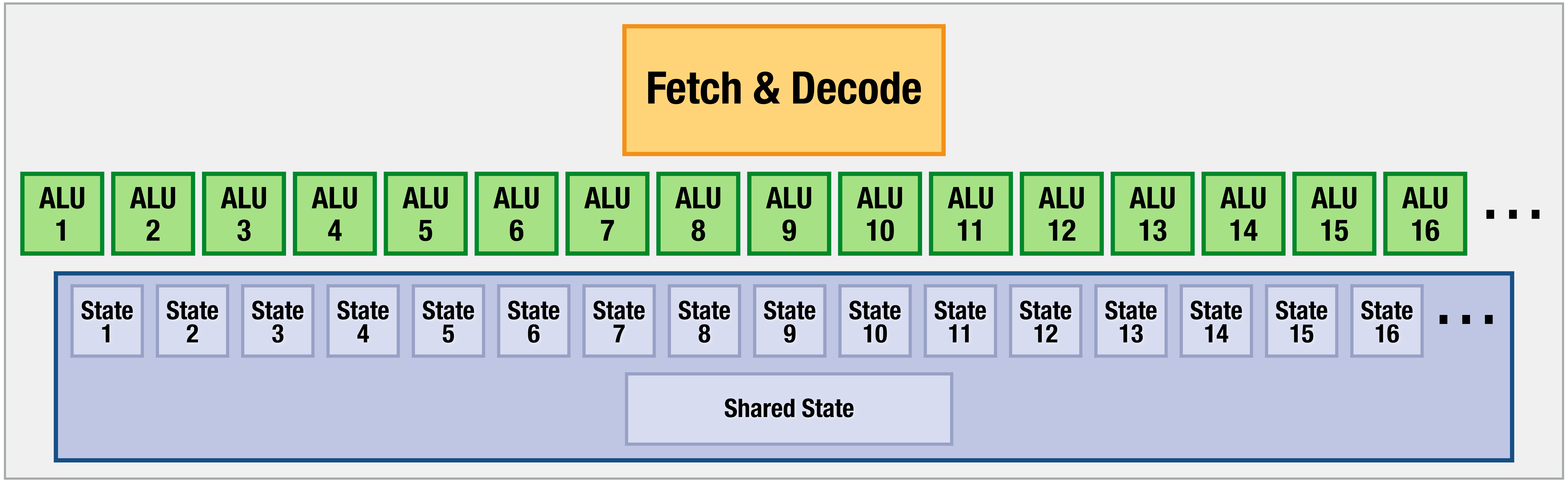
many GPUs





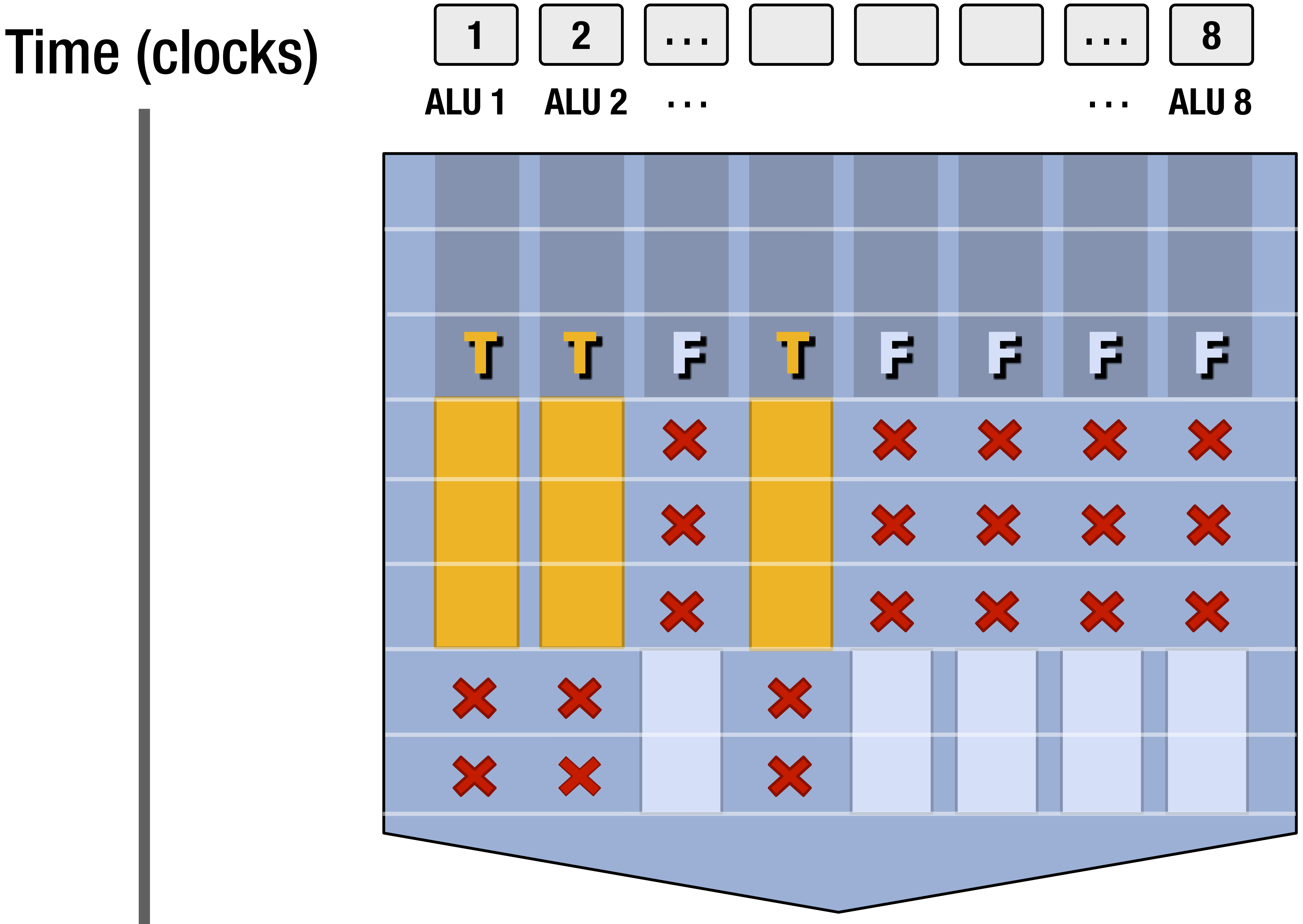
Slides inspired by Kayvon Fatahalian

SIMD execution requires **coherent control**



Slides inspired by Kayvon Fatahalian

SIMD execution requires **coherent control**

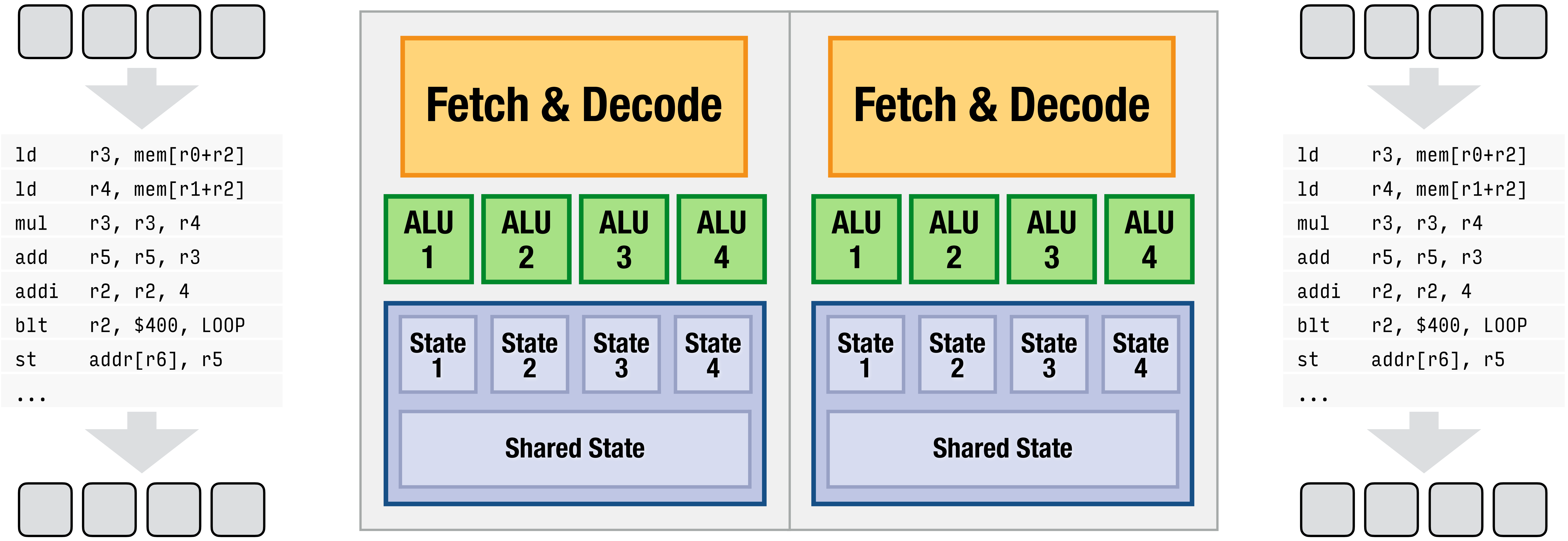



```
<unconditional code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
        code>
```

Not all ALUs do useful work every cycle!
Worst case: 1/n peak performance

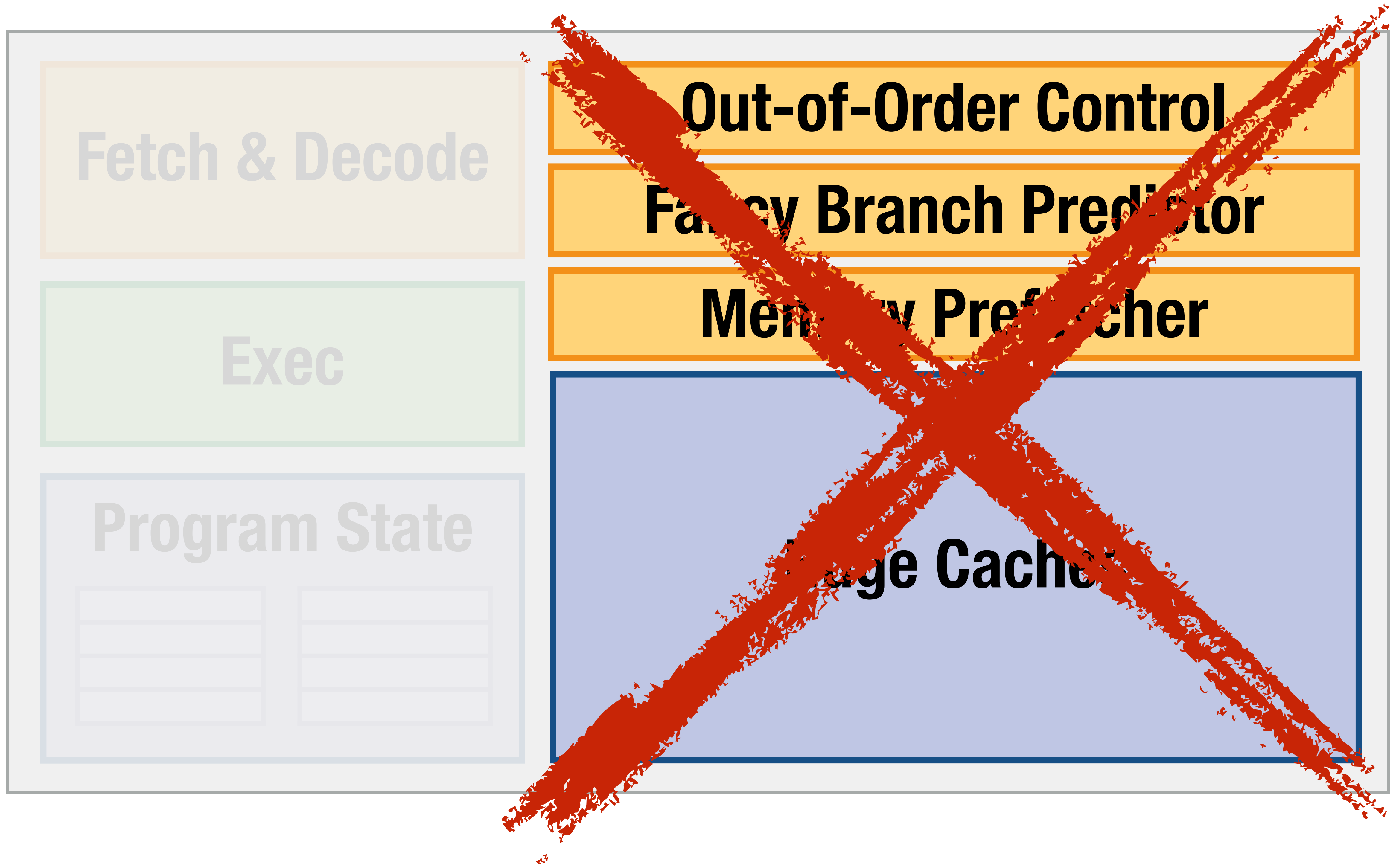
Slide adapted from Kayvon Fatahalian

Diminishing returns,
scale further with **multicore**



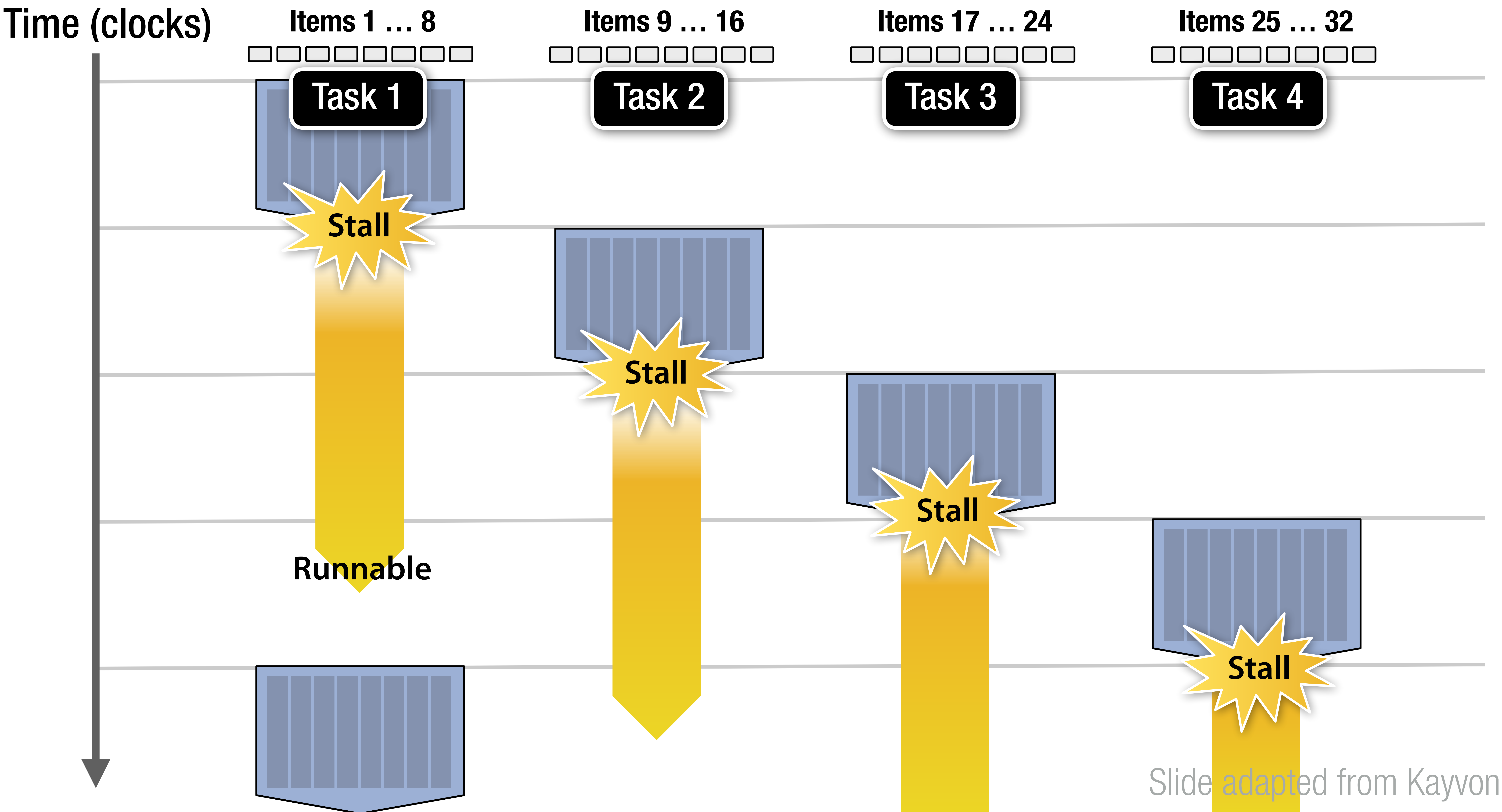
Typical SIMD width: 8 ~ 64

How to deal with **latency** without **stalling**



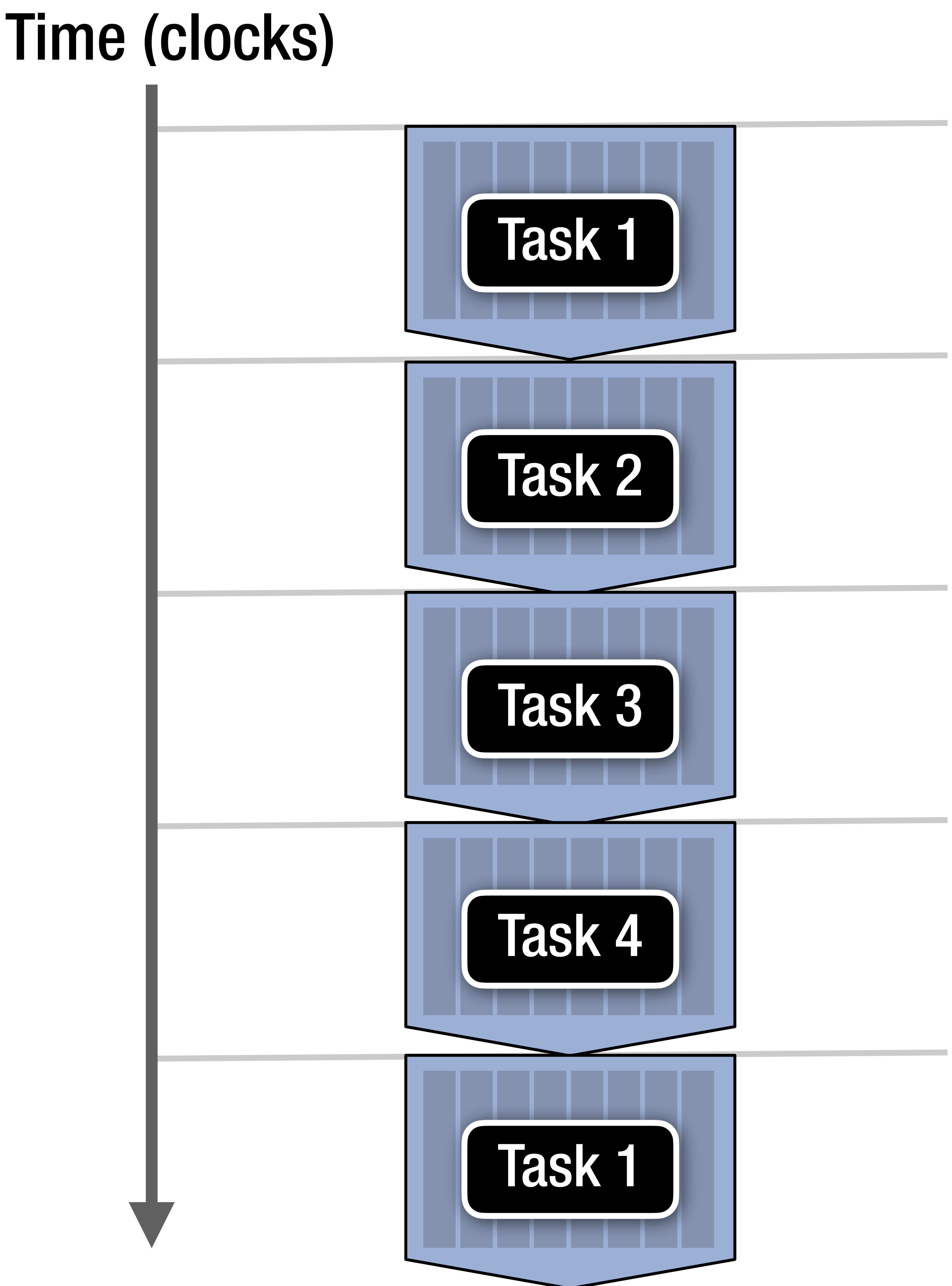
Opportunity:
exploit **parallelism**
to hide latency

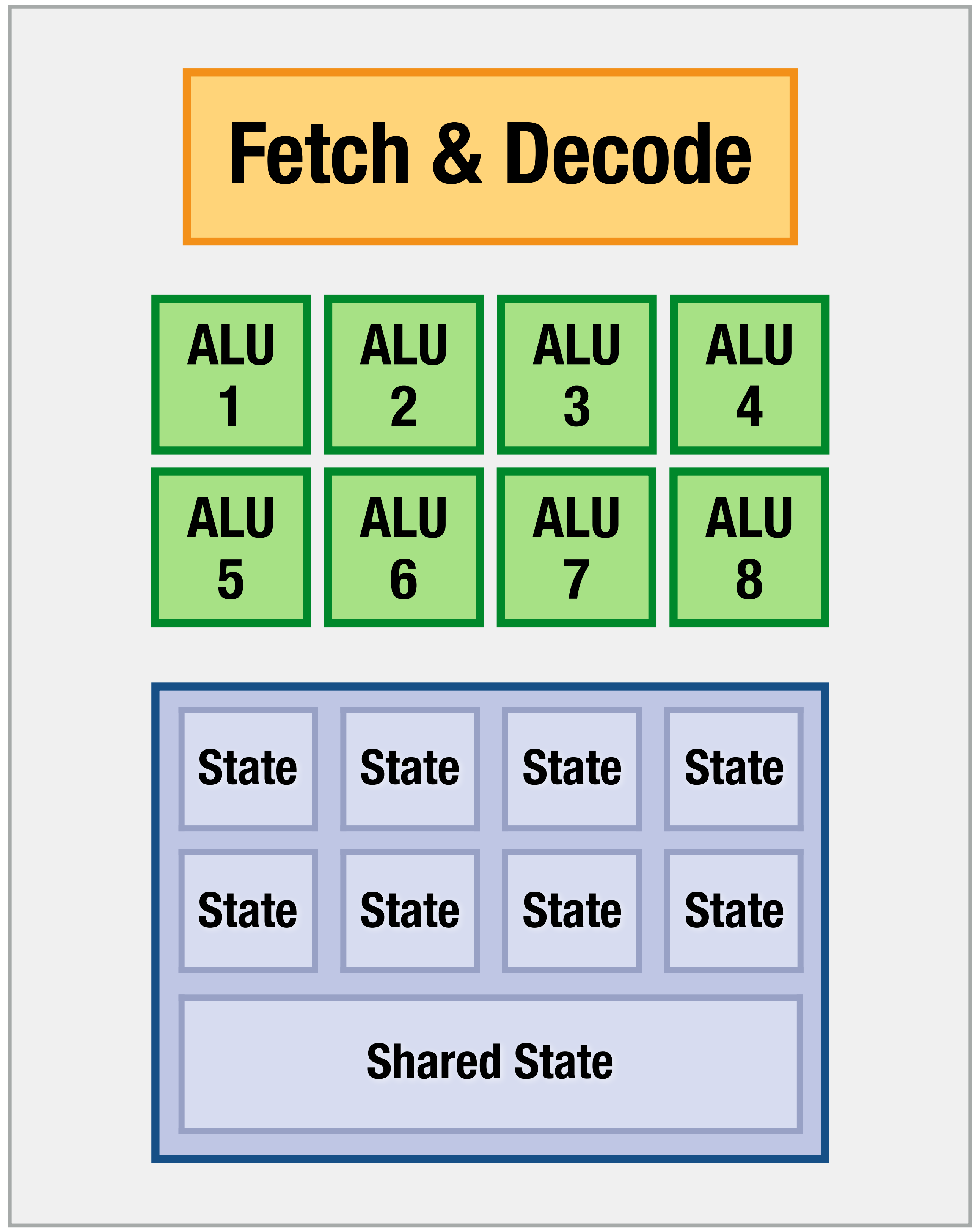
Idea 3: Interleave parallel tasks to hide latency



Slide adapted from Kayvon Fatahalian

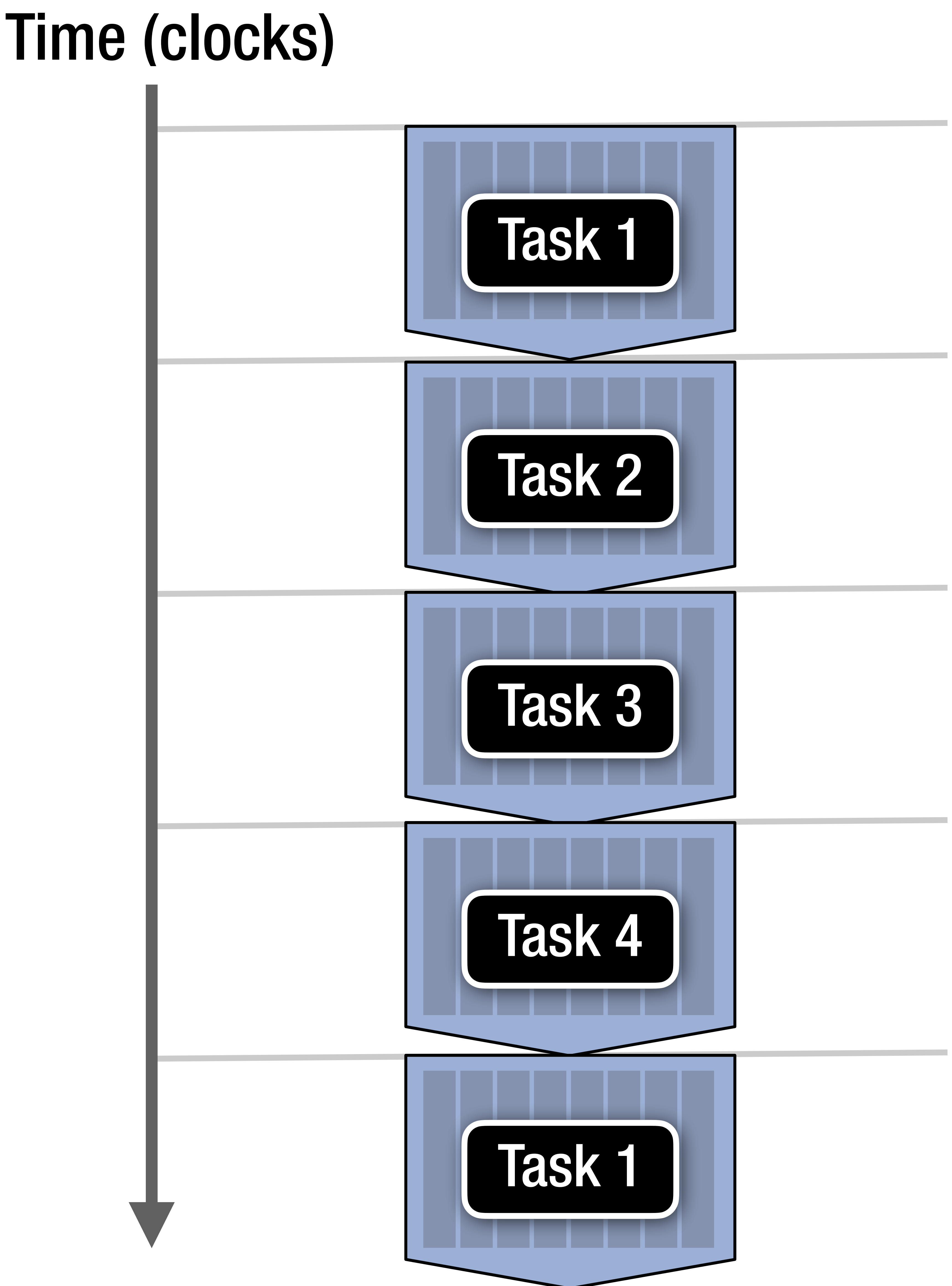
Idea 3: Interleave parallel tasks to hide latency

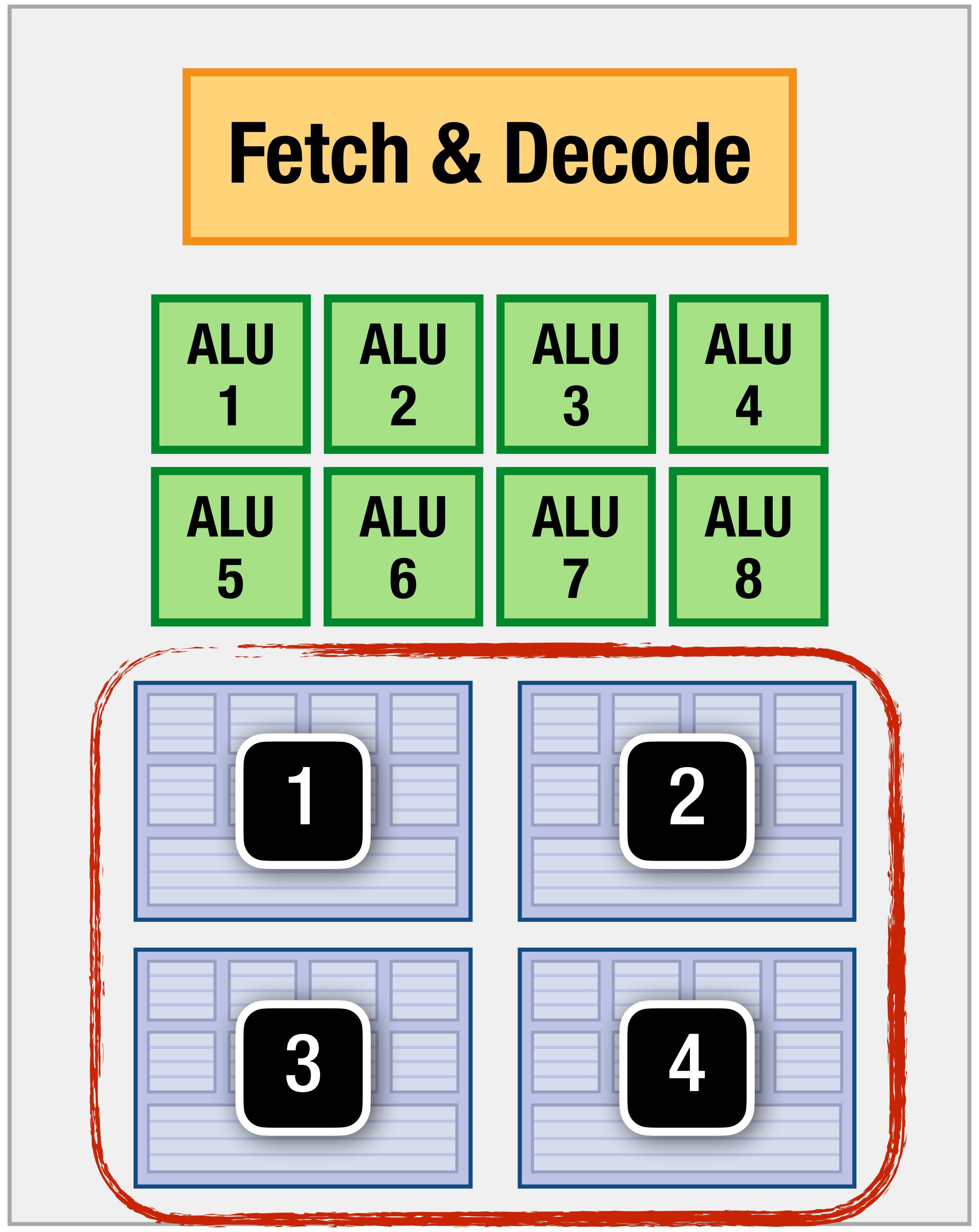




Slide adapted from Kayvon Fatahalian

Interleaving requires more **state storage**





Multithreading requires **extra state storage** for idle threads

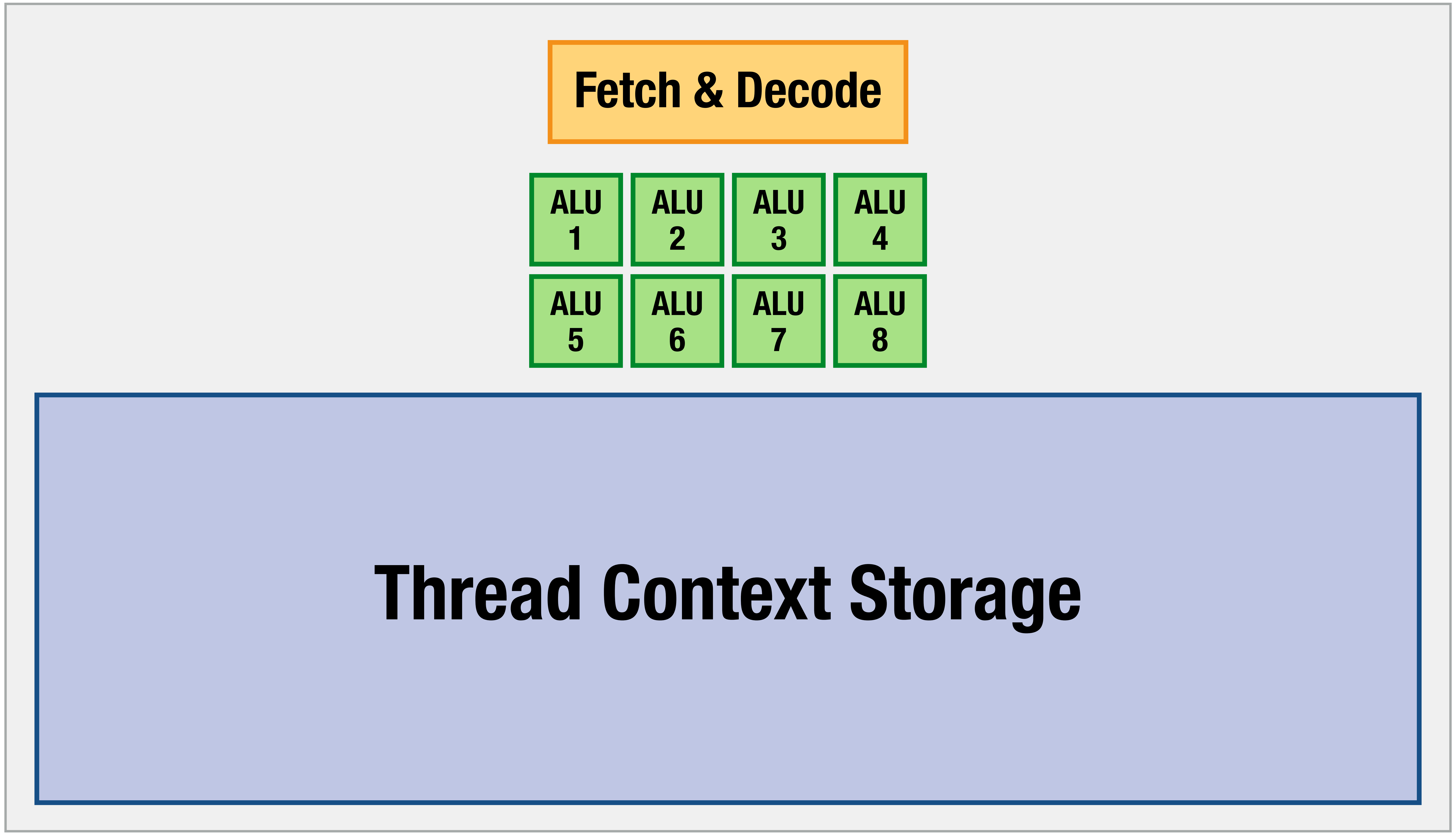
Slide adapted from Kayvon Fatahalian

Interleaving requires more **state storage**

DRAM latency:
100s of cycles

AMD/Intel Hyperthreadng:
2 threads

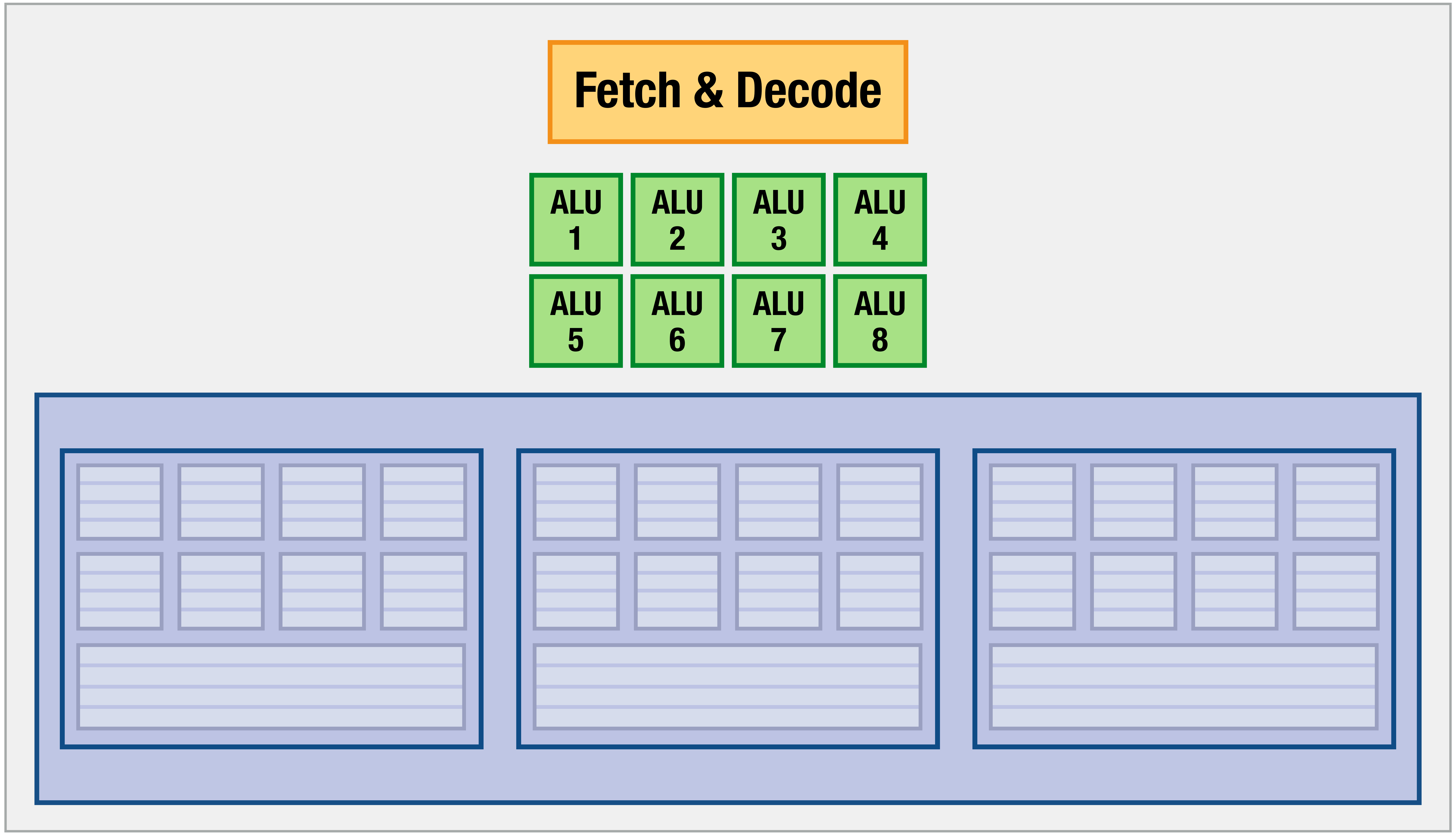
NVIDIA H100:
16k 32-bit words per warp scheduler (core)
↳ **32MB** per GPU



Slide adapted from Kayvon Fatahalian

Tradeoff: per-thread state vs. latency hiding

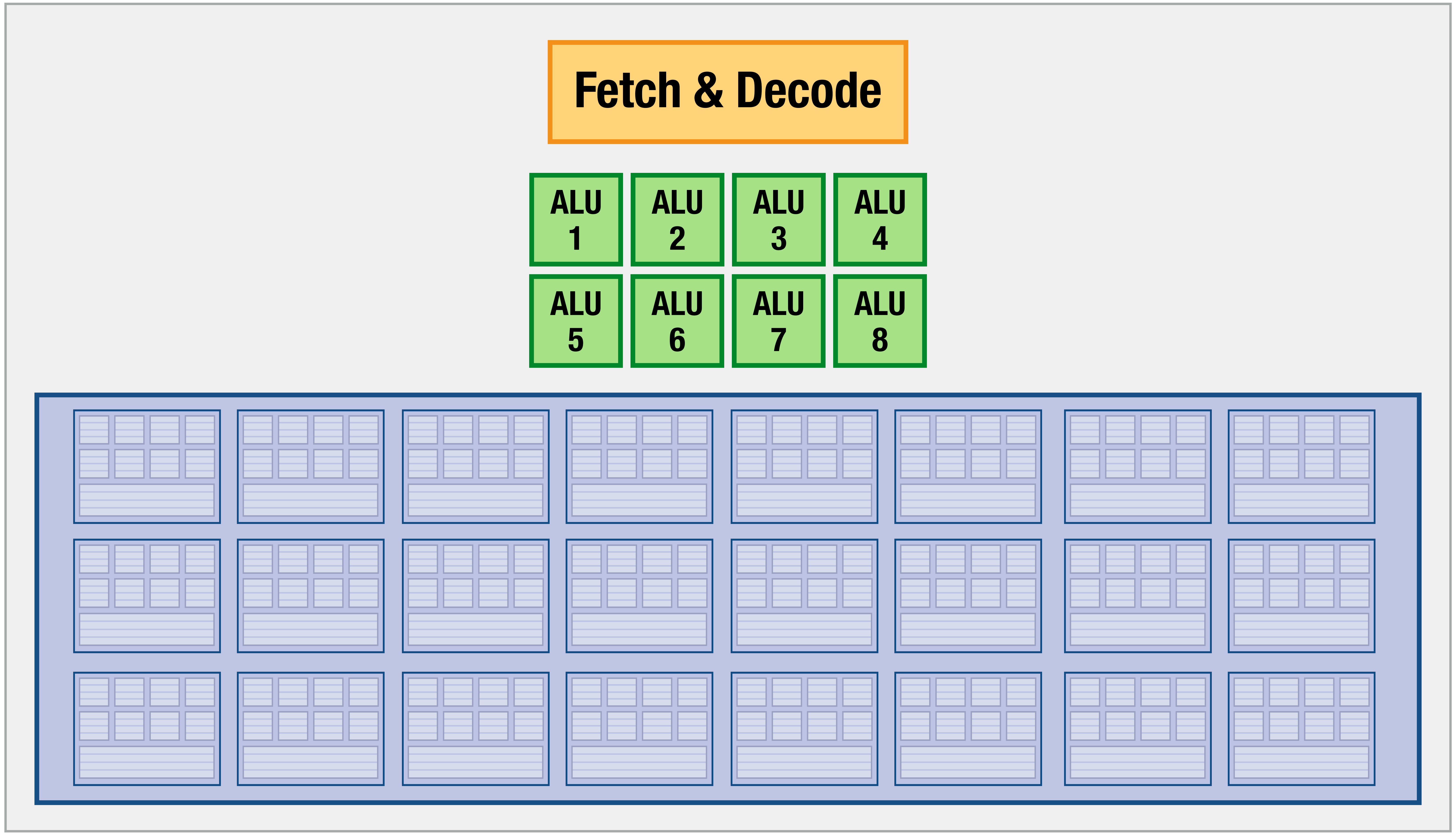
Few, large contexts:
limited latency hiding



Slide adapted from Kayvon Fatahalian

Tradeoff: per-thread state vs. latency hiding

Many, small contexts:
maximal latency hiding



A **throughput-oriented processor** exploits abundant parallelism for efficiency

1. Scale performance with **multicore**, not instruction-level parallelism

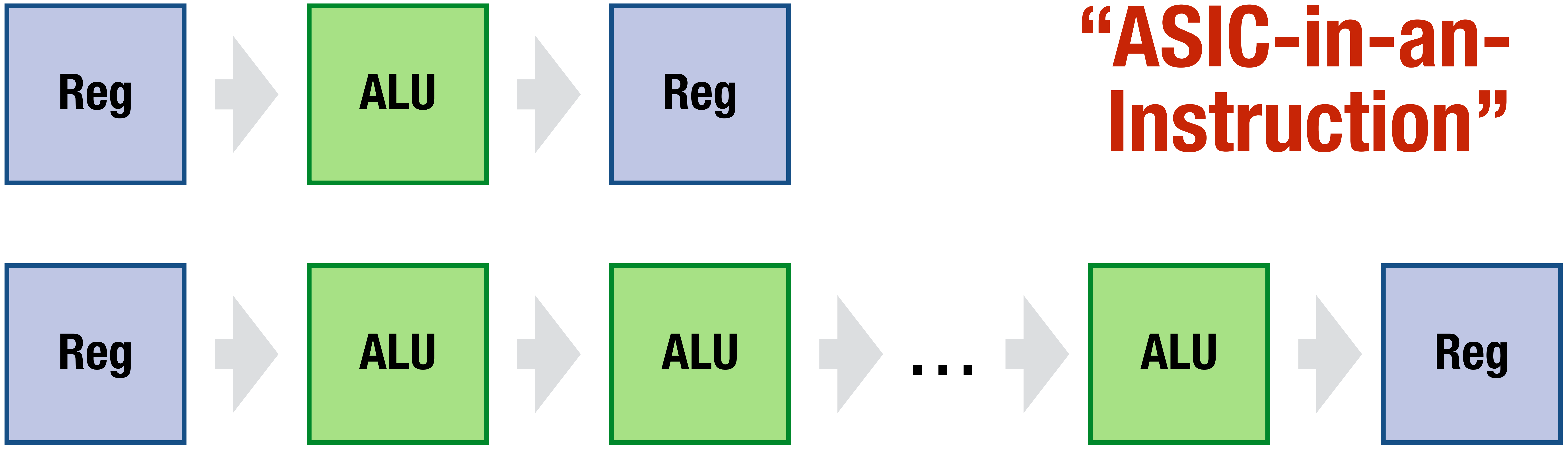
2. Amortize **control overhead** with **SIMD execution**

3. Hide **latency** with **concurrent threads**, not speculation

Idea 4: amortize instruction overheads with **more complex instructions**

Primitive op:
(“RISC”)

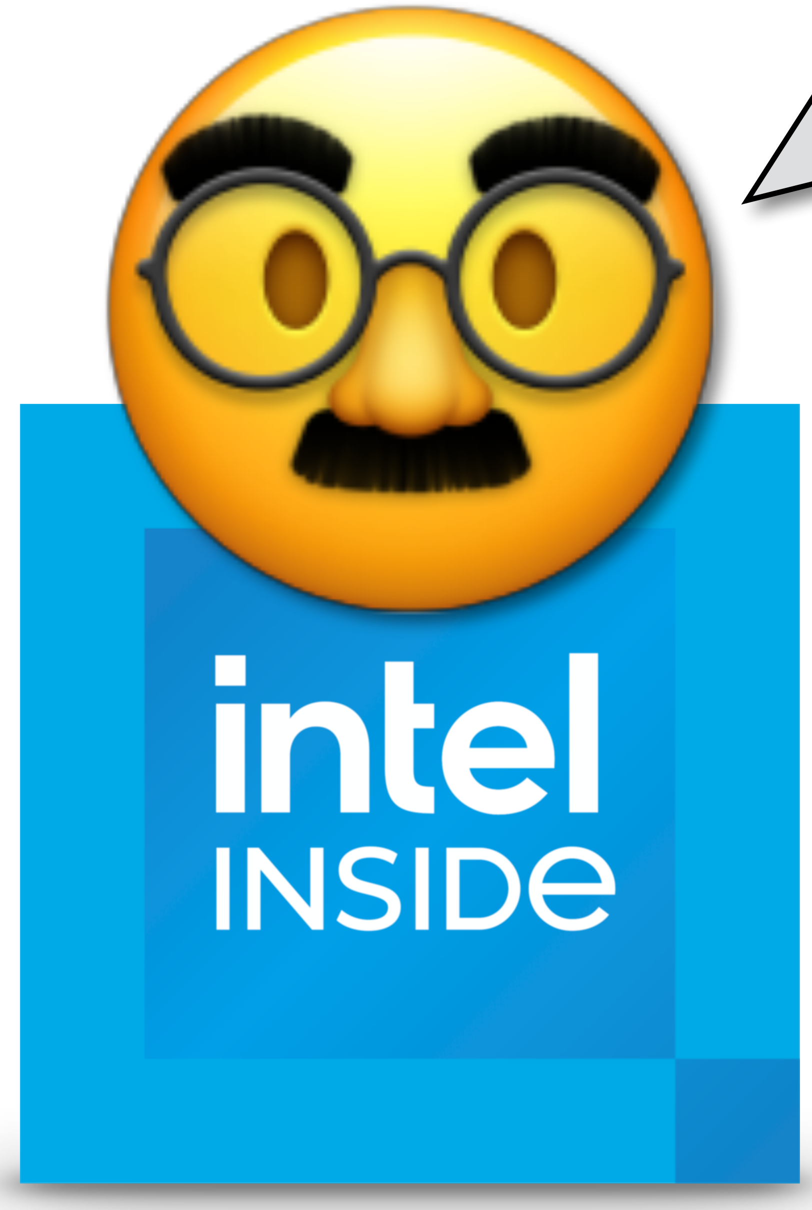
Complex op:
(“CISC++”)



“ASIC-in-an-Instruction”

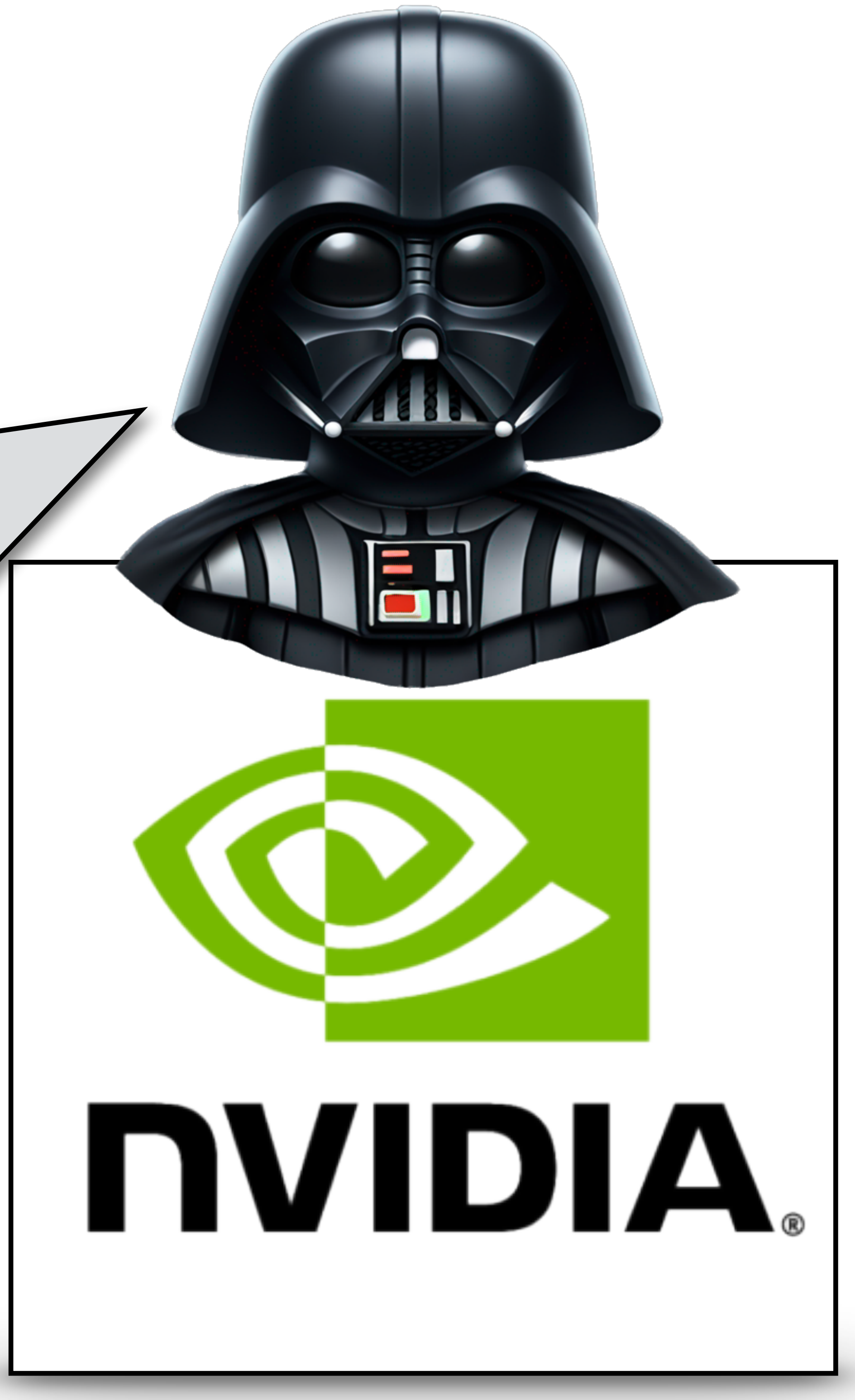
e.g., AES, video encode/decode, DSP, texture filtering, …

and especially **matrix multiply!**



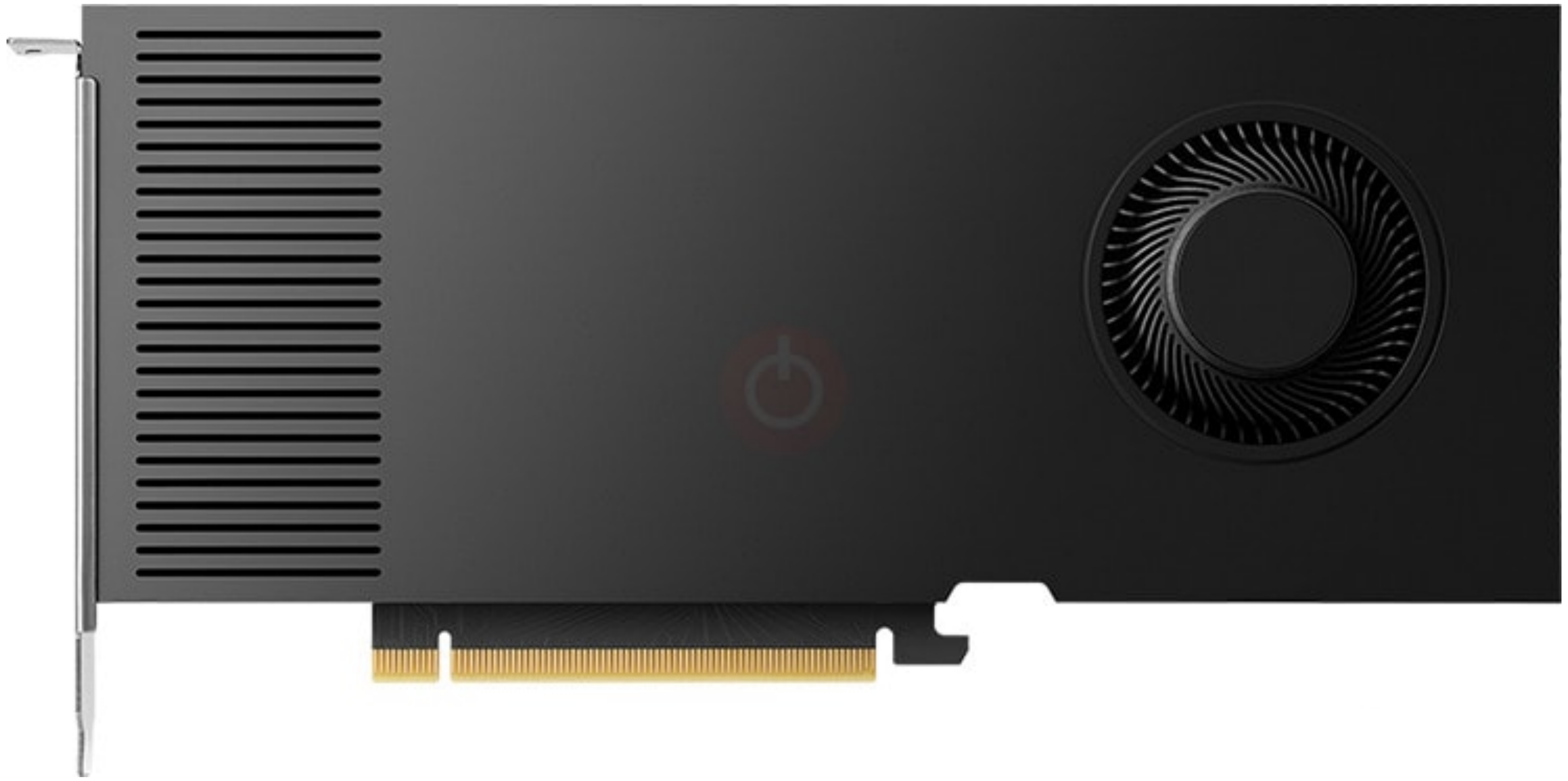
Why yes,
I am still
basically a
PDP-11

You will **rewrite**
all your code in
CUDA . . .
and pray I don’t
alter the deal
any further!



How do these ideas appear in **real hardware?**

Our GPU: NVIDIA RTX 4000 Ada (AD104 chip)



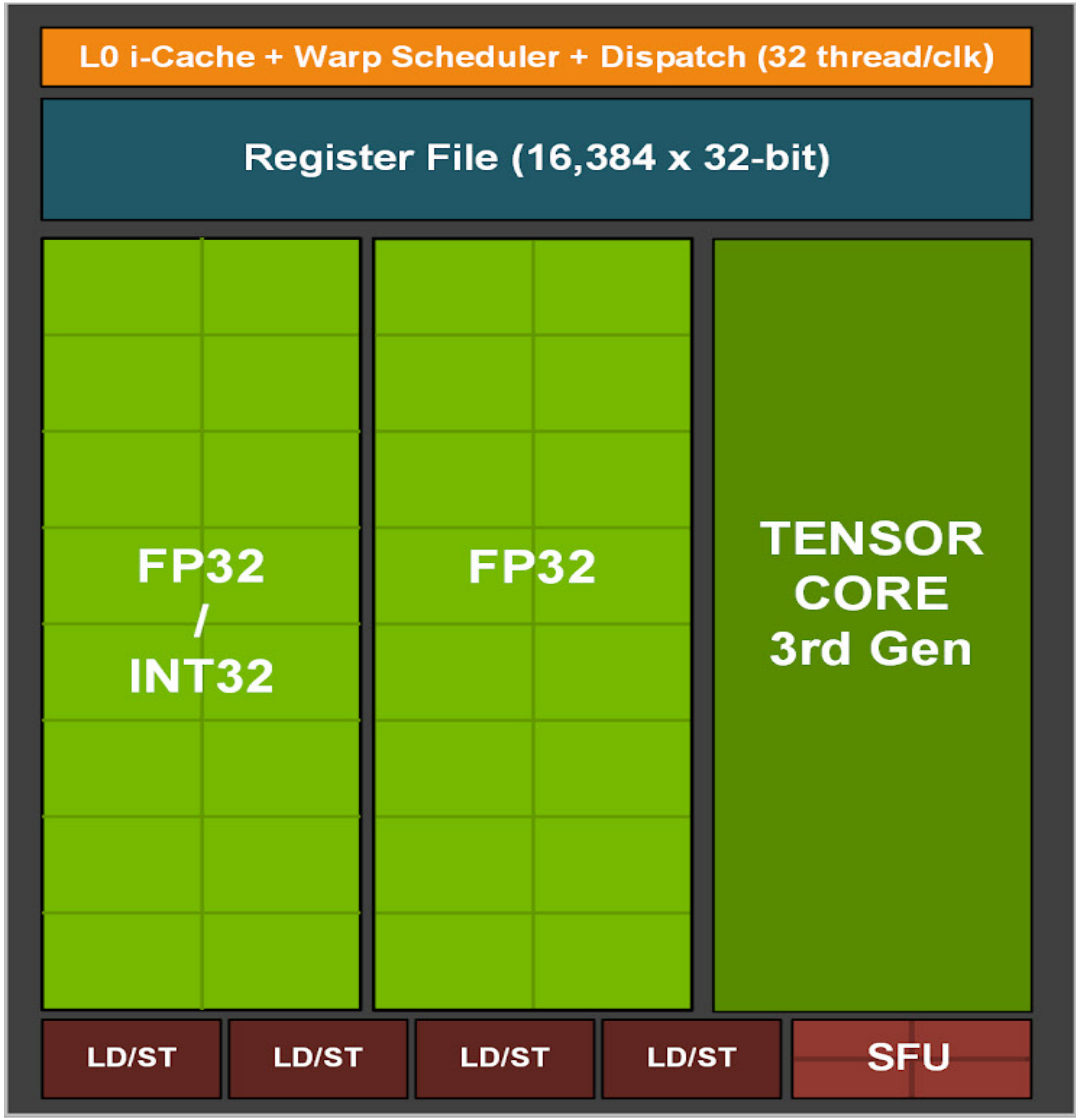
One “core” of our GPU
(“warp scheduler”)

32 x 32-bit Exec. Units
(ALUs / vector lanes)

512 x 32 x 32-bit regs

1 warp instruction / clock
(32 lanes)

Up to 12 live threads
(independent warps)



One “SM” is a cluster of 4 warp schedulers

Our whole GPU

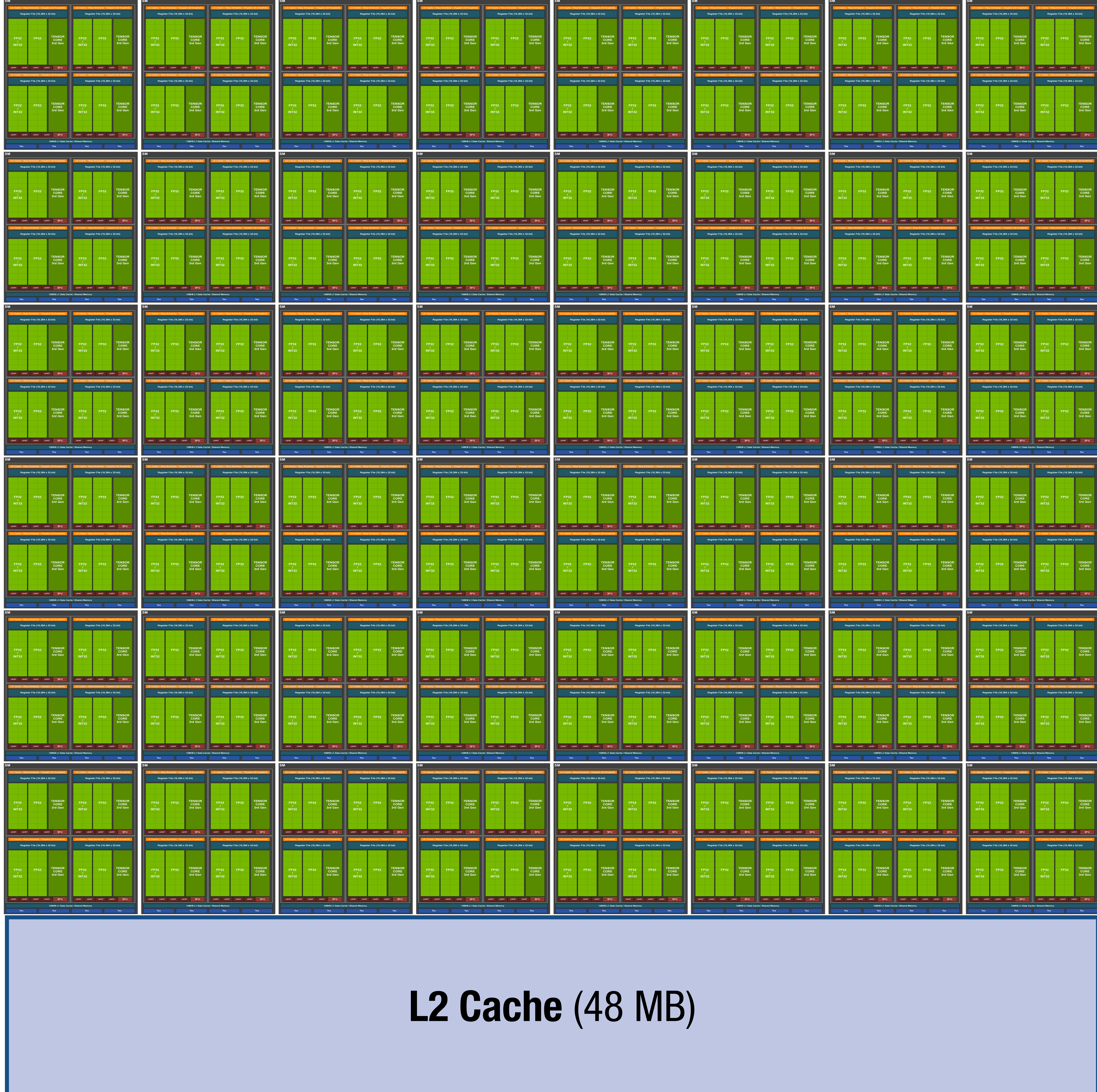
48 SMs
x 4 = 192 cores
x32 = 6144 exec. units

@ 2.175 GHz
= 26.7 TFLOPS

192x12 = 2,304 threads

96k 1024-bit registers
= 12 MB

48 MB L2 cache



Our CPU: AMD Ryzen 7 7700

One core of our CPU
(AMD Zen 4)

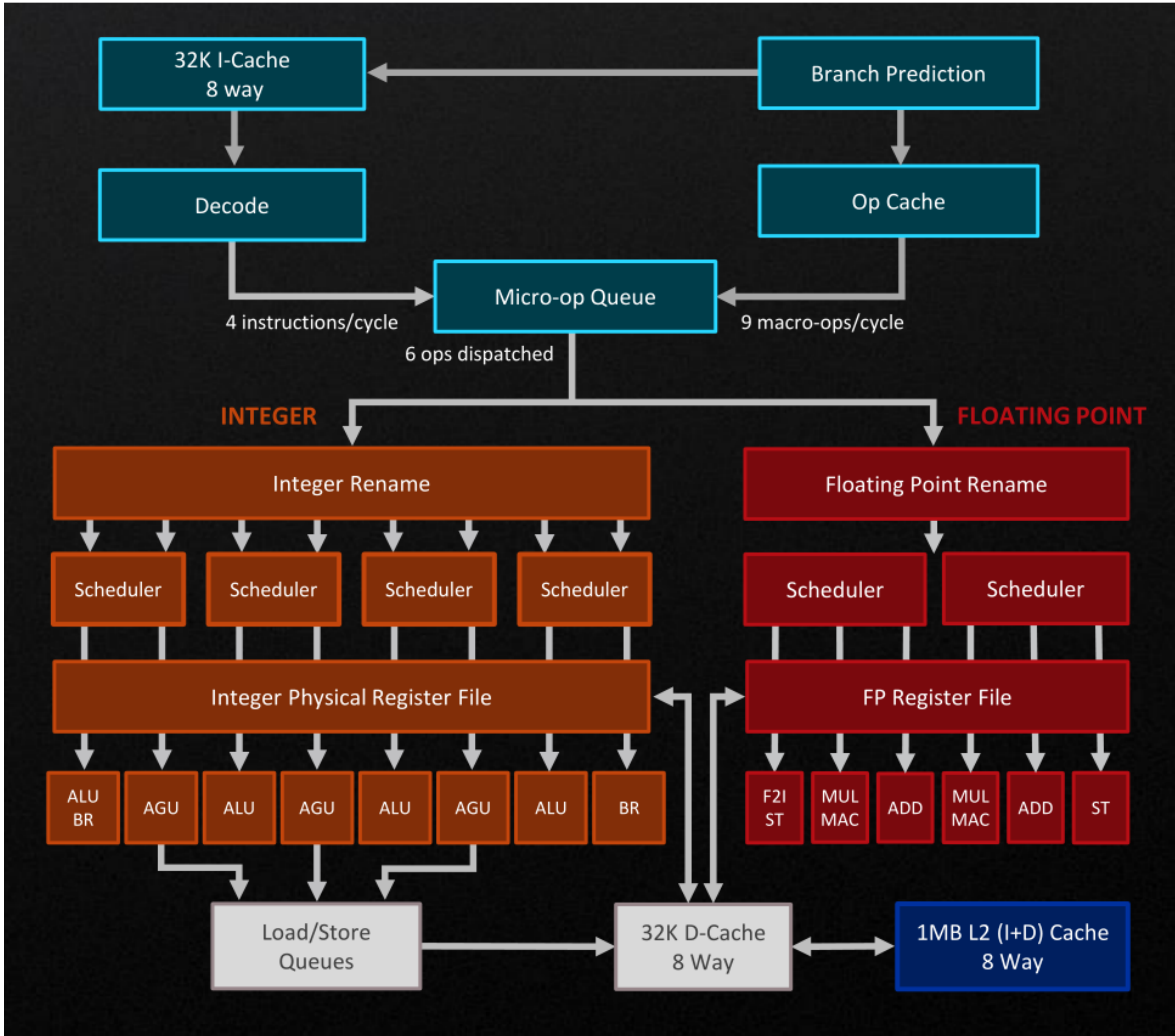
Huge, complex control logic (6-wide, OoO, …)

Two 256-bit vector ALUs = 16 x 32-bit exec. units

192 vector registers

1 MB cache

2 “hyperthreads”



Our whole CPU

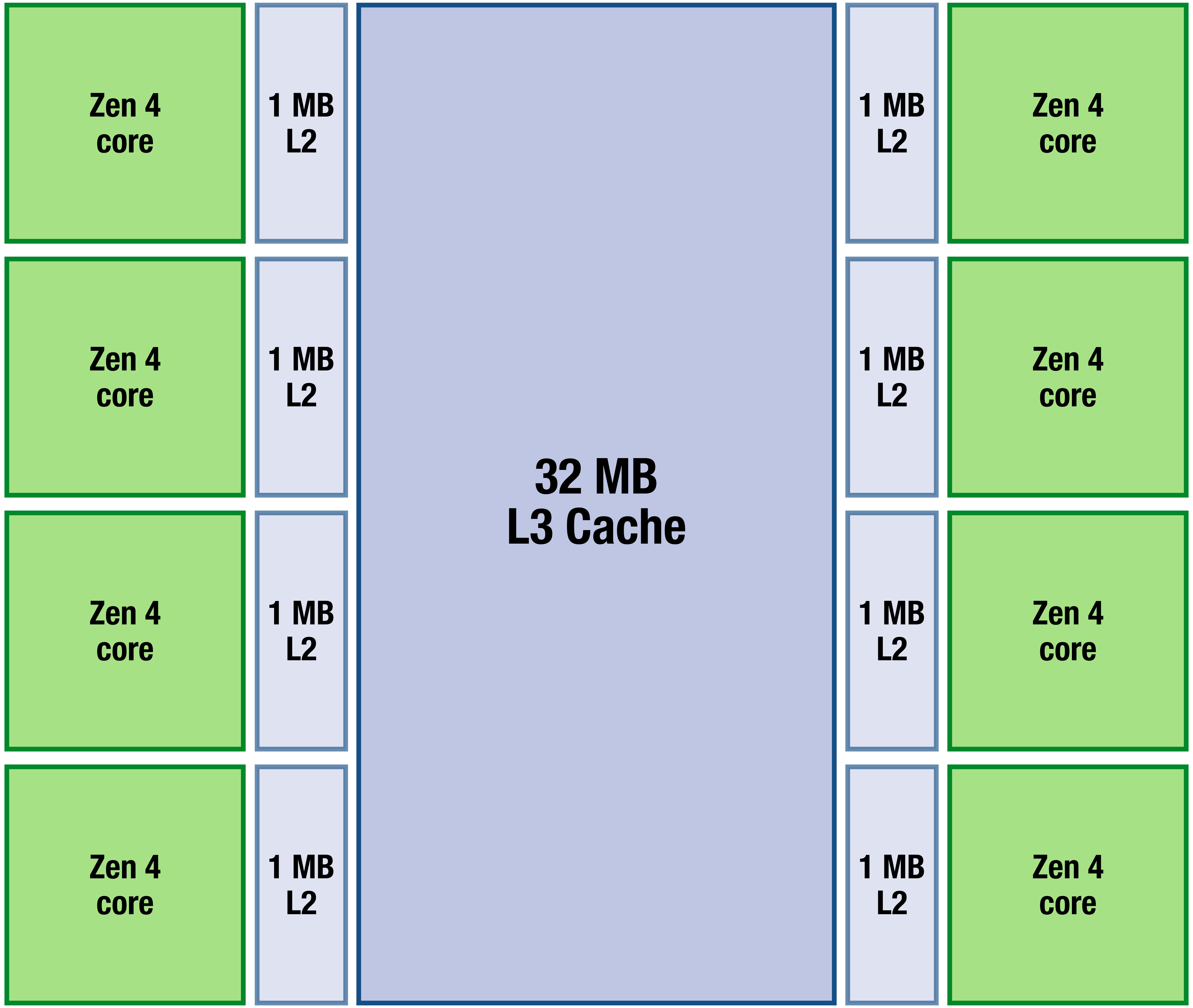
8 “Zen 4” cores
x 16 = 128 exec units

@ 3.8 ~ 5.5 GHz
= 0.97 ~ 1.4 TFLOPS

8 x 2 = 16 threads

1.5k 512-bit registers
= 24 KB

40 MB total L2+L3



Our CPU

8 "Zen 4" cores
x 16 = 128 exec units

@ 3.8 ~ 5.3 GHz
= 0.97 ~ 1.4 TFLOPS

8 x 2 = 16 threads

1.5k 512-bit registers
= 24 KB

40 MB total L2+L3

Our GPU

48 SMs
x 4 = 192 cores
x32 = 6144 exec. units

@ 2.175 GHz
= 26.7 TFLOPS

192x12 = 2,304 threads

96k 1024-bit registers
= 12 MB

48 MB L2 cache

Our CPU

70 mm^2 (CCD only)

70 mm^2

/ 8 cores

= 8.75 mm^2

6-7x faster on Mandelbrot

Our GPU

294 mm^2 (full die)

~200 mm^2

/ 192 cores

= 1.05 mm^2 / core

8x smaller

A **throughput processor** is **still a processor!**
(or actually, many of them)

But we need to **change our programs** to use it efficiently
expose **explicit parallelism**
within & across instruction streams