Lecture 2:

A Modern Multi-Core Processor

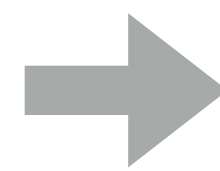Parallel Computing
Stanford CS149, Fall 2024

Review from class 1:
What is a computer program?

A program is 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;
}
```
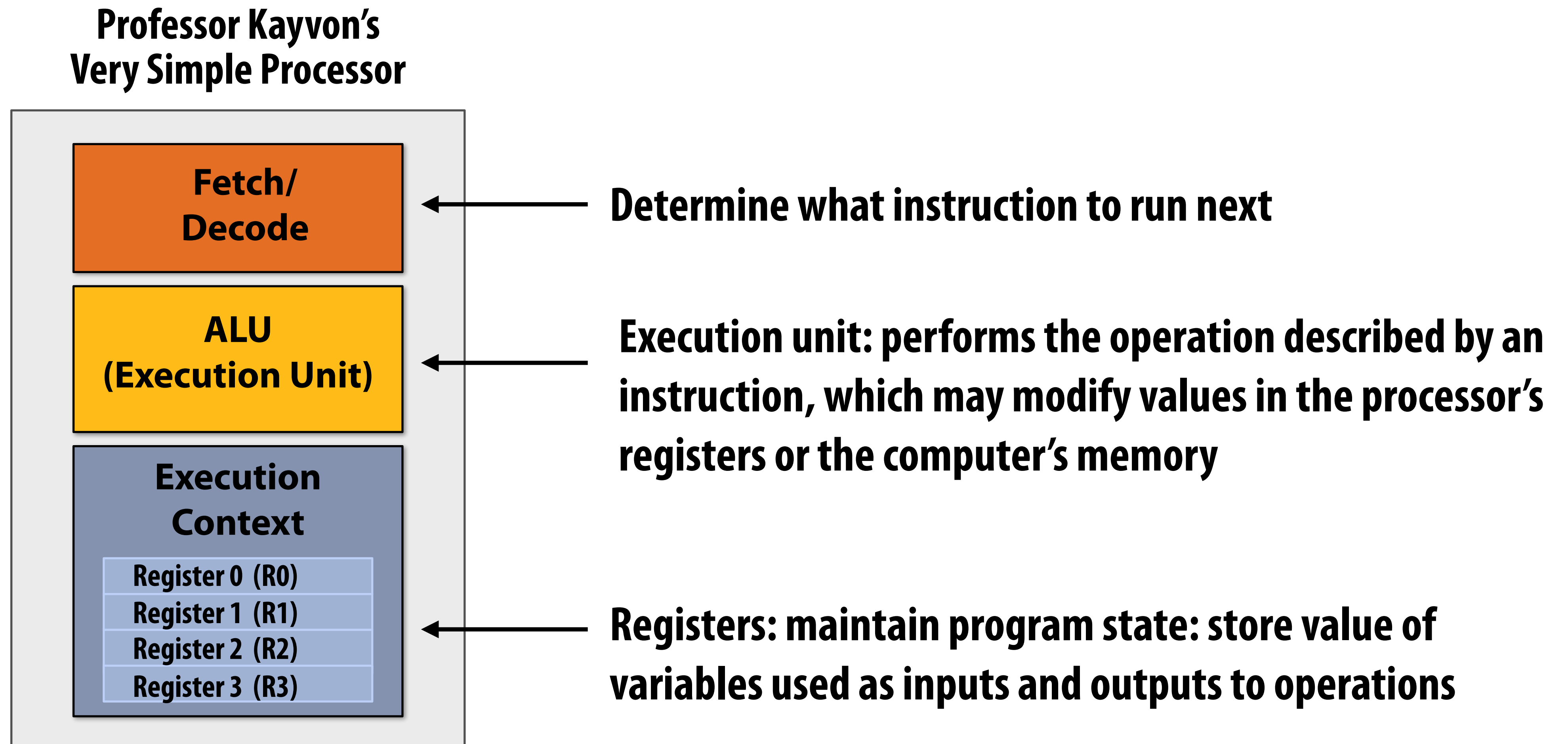
Compile code

```
_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
100000f6d:    addq $32, %rsp
100000f71:    popq %rbp
100000f72:    rets
```

Review from class 1:
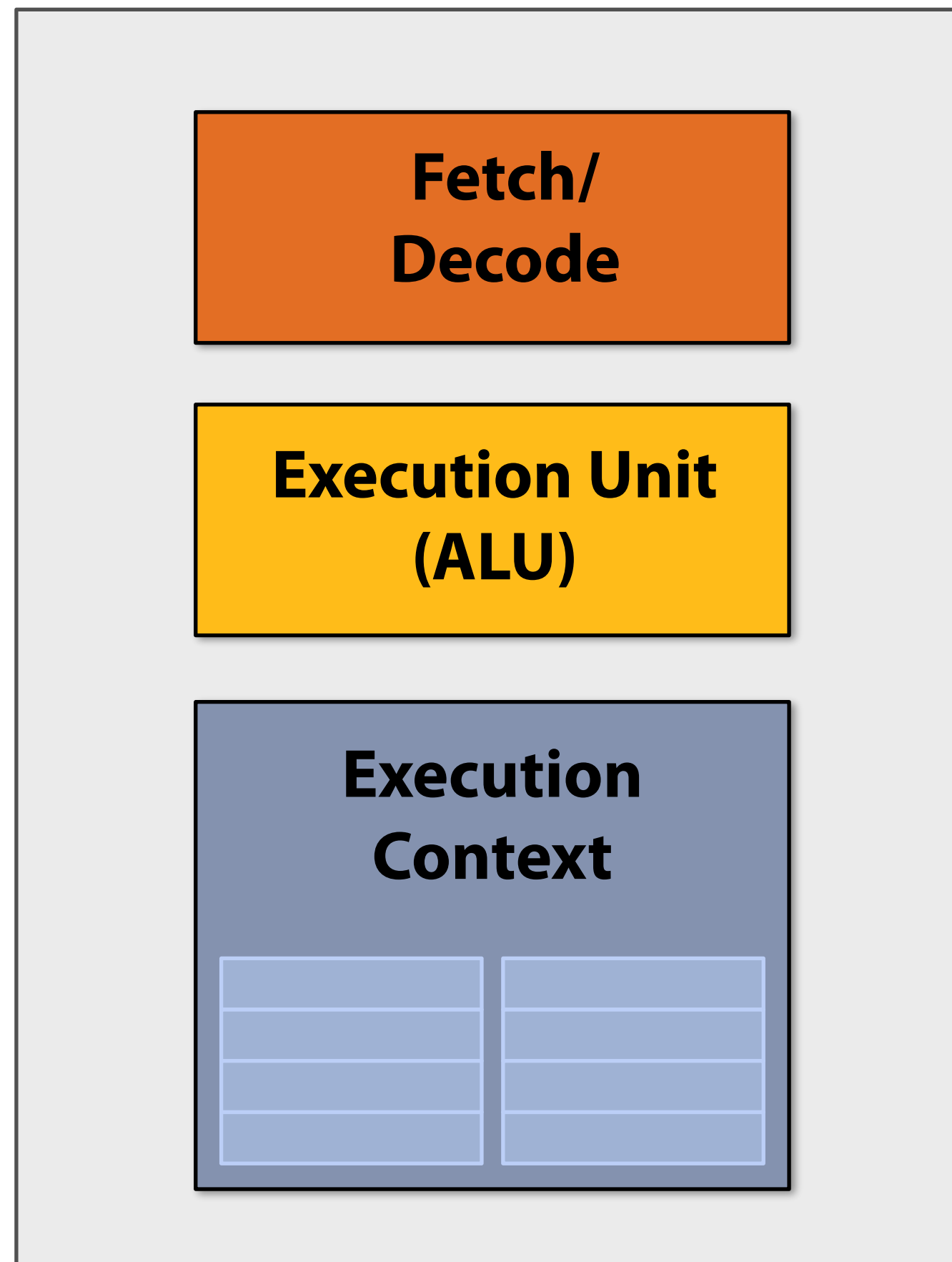# What does a processor do?

A processor executes instructions

**Professor Kayvon's
Very Simple Processor**

**Fetch/
Decode** ← Determine what instruction to run next

**ALU
(Execution Unit)** ← Execution unit: performs the operation described by an instruction, which may modify values in the processor's registers or the computer's memory

**Execution
Context**

Register 0 (R0)
Register 1 (R1)
Register 2 (R2) ← Registers: maintain program state: store value of variables used as inputs and outputs to operations
Register 3 (R3)
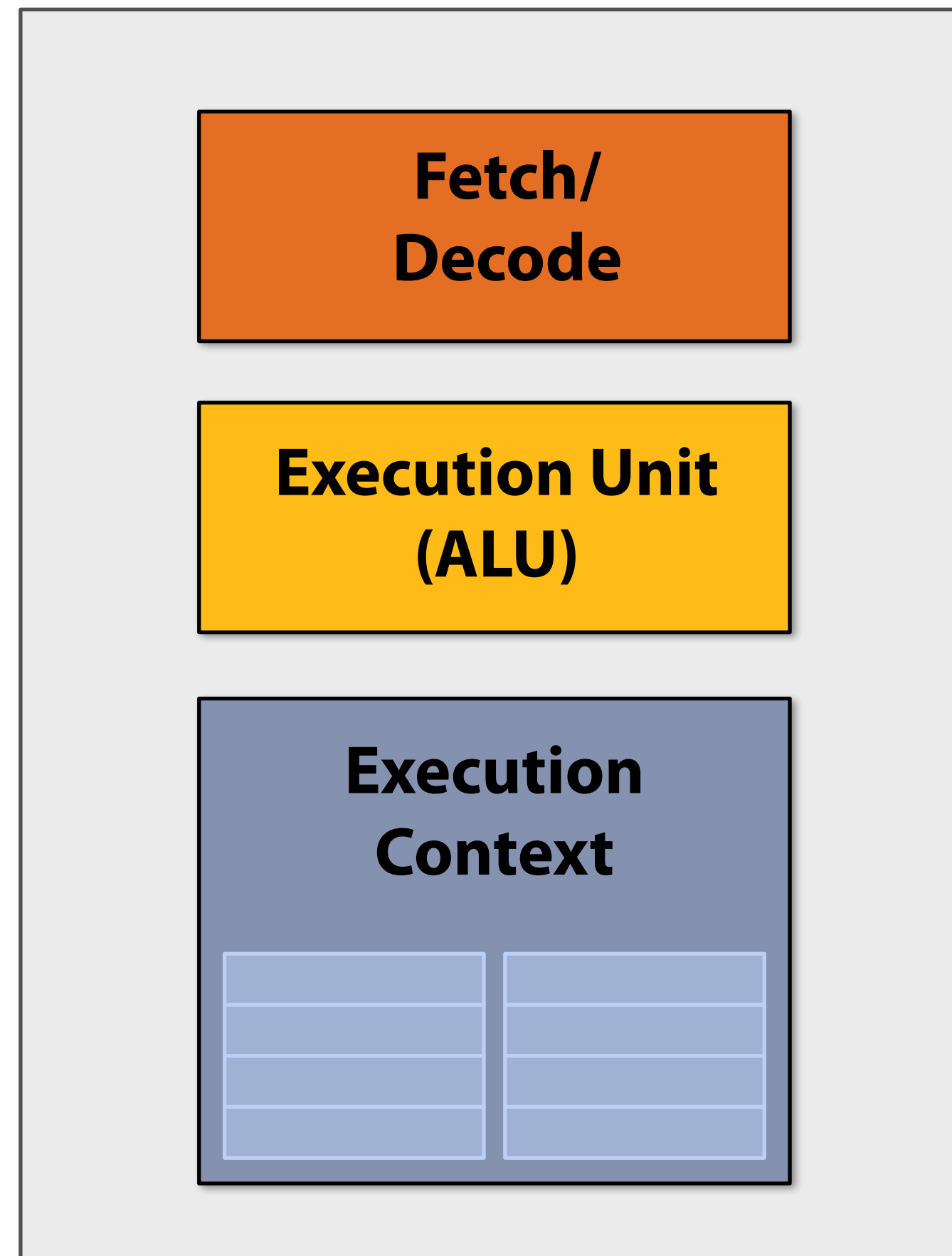
Execute program

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
```

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
```

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
```

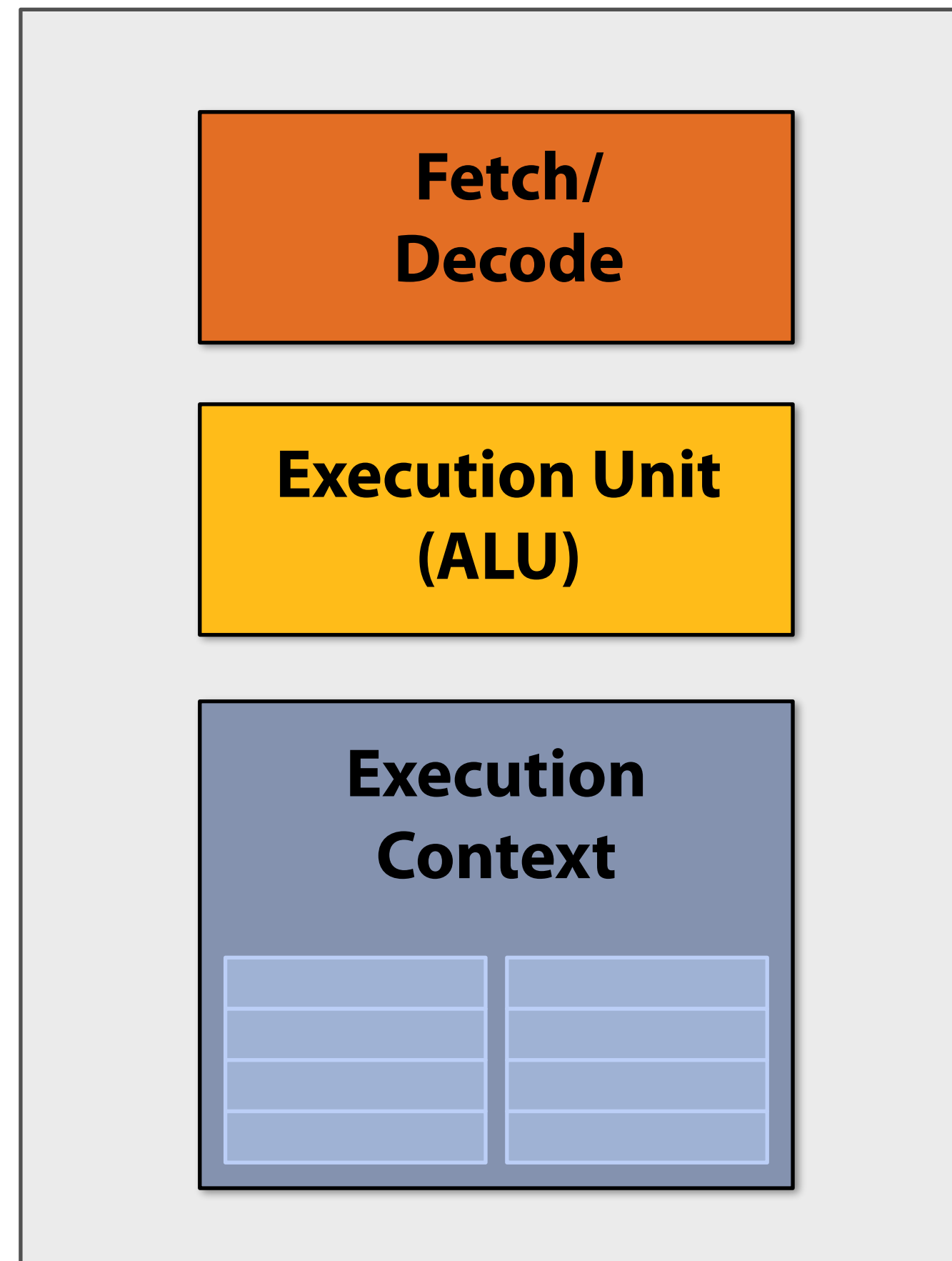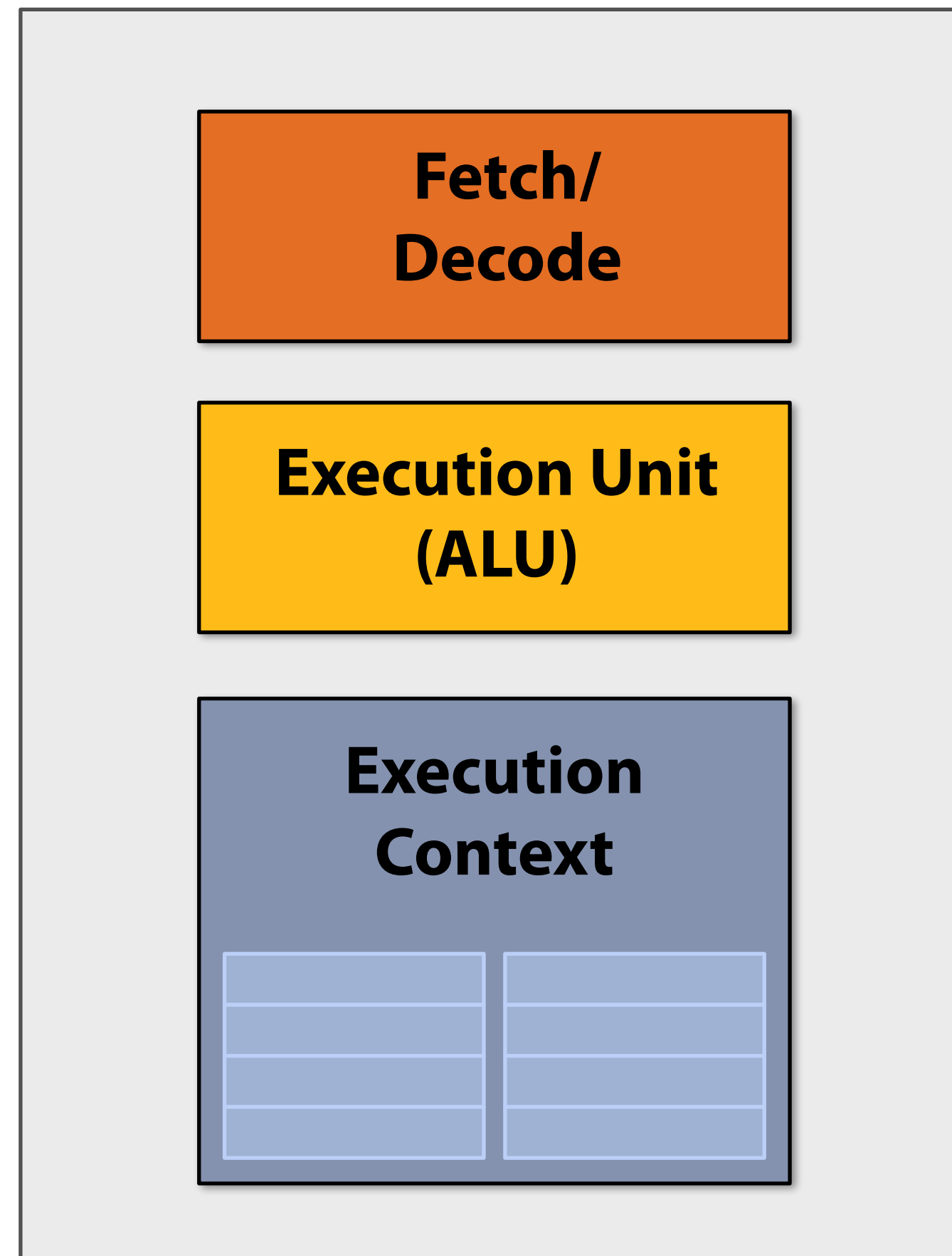# Execute program

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
```

A program with instruction level parallelism

Program (sequence of instructions)

Instruction dependency graph

```
PC    Instruction
───────────────────────────────────
00 │  a = 2
01 │  b = 4
   │
02 │  tmp2 = a + b        // 6
03 │  tmp3 = tmp2 + a     // 8
04 │  tmp4 = b + b        // 8
05 │  tmp5 = b * b        // 16
06 │  tmp6 = tmp2 + tmp4  // 14
07 │  tmp7 = tmp5 + tmp6  // 30
   │
08 │  if (tmp3 > 7)
09 │     print tmp3
   │  else
10 │     print tmp7
```

value during execution
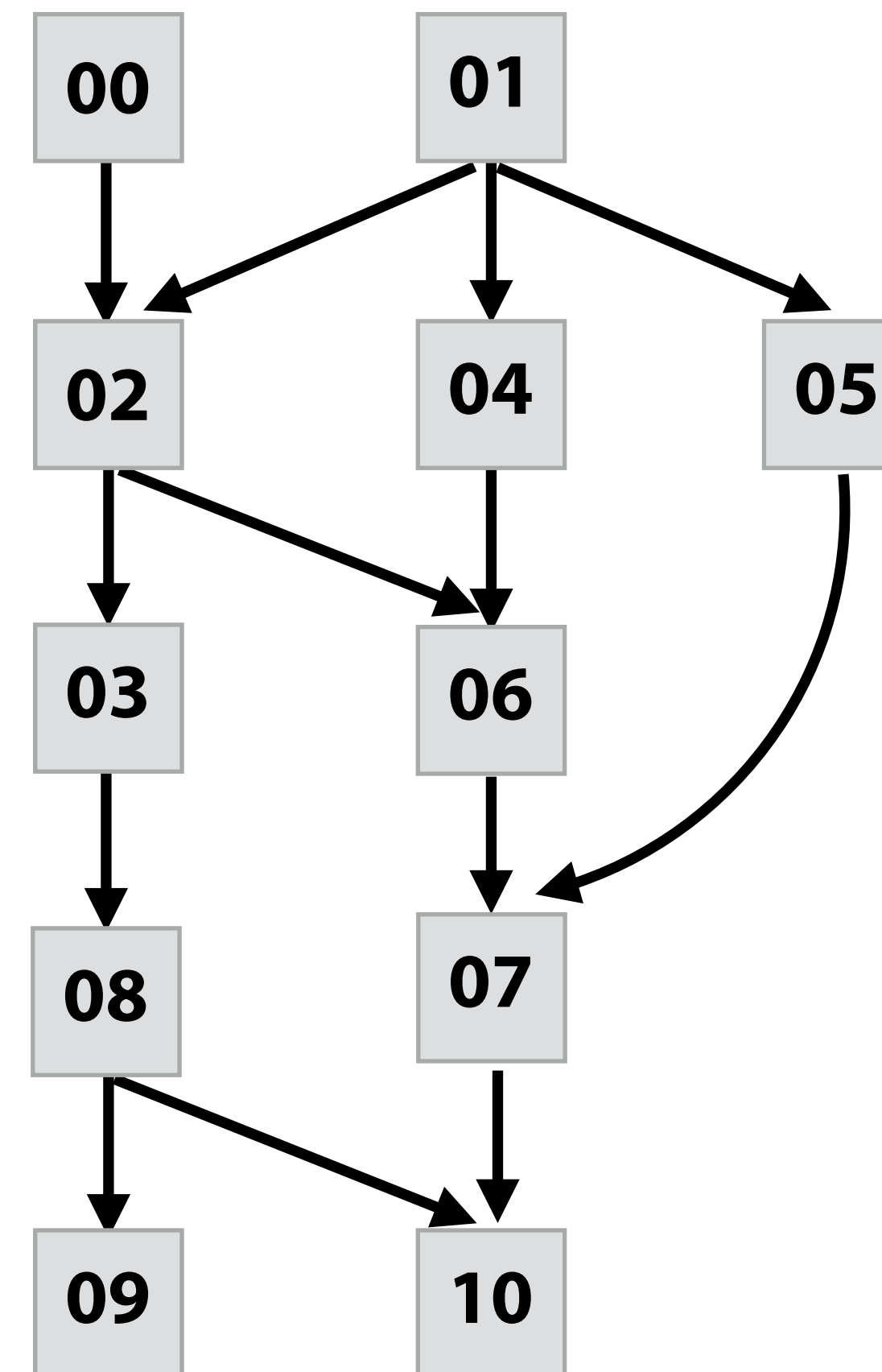
Superscalar processor

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

Out-of-order control logic

Fetch/ Decode 1	Fetch/ Decode 2

Exec 1	Exec 2
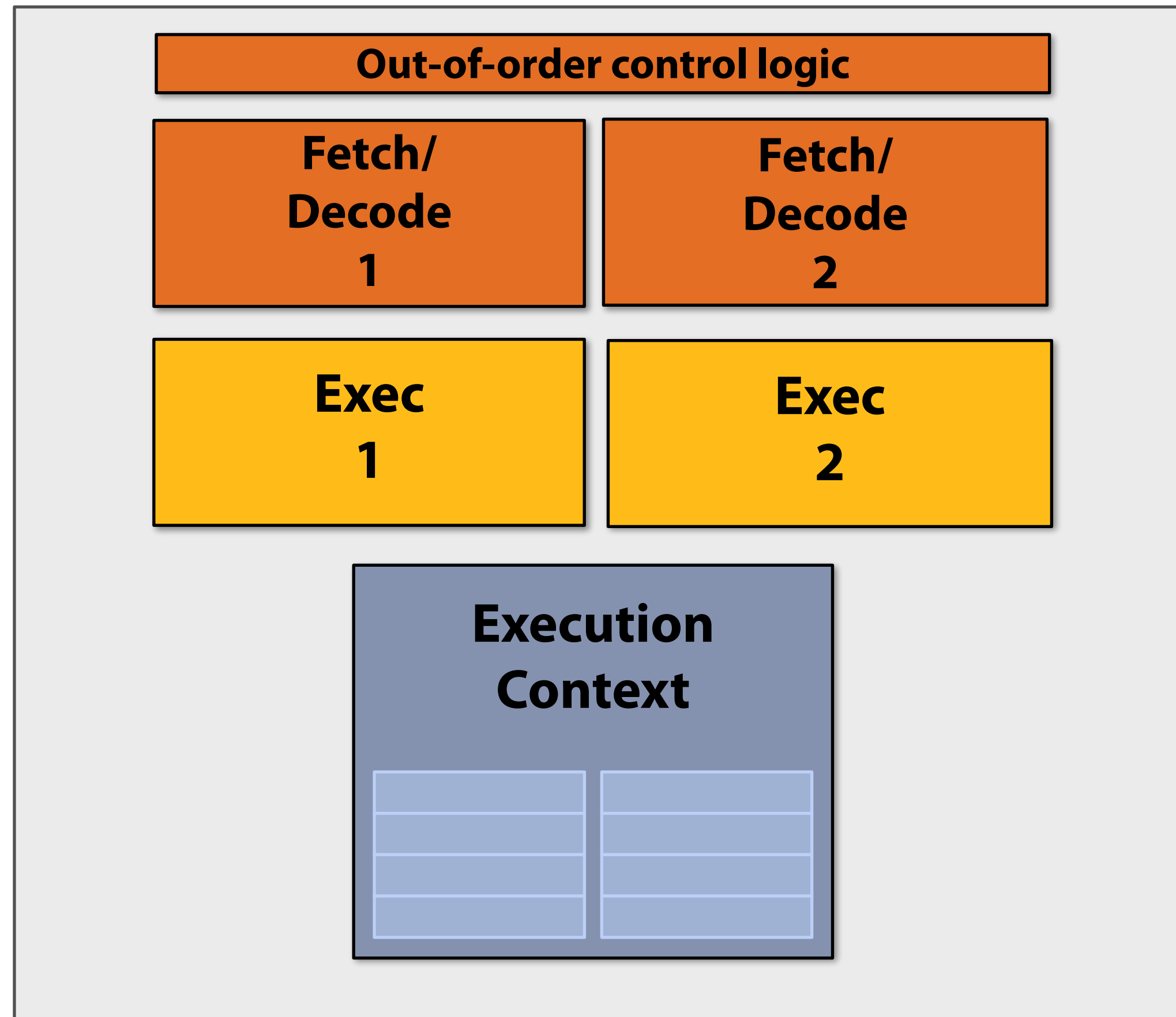
**Execution
Context**

Superscalar execution: processor automatically finds independent instructions in an instruction sequence and can execute them in parallel on multiple execution units.

What does it mean for a superscalar processor to "respect program order"?

Review from class 1:
What is memory?

Memory
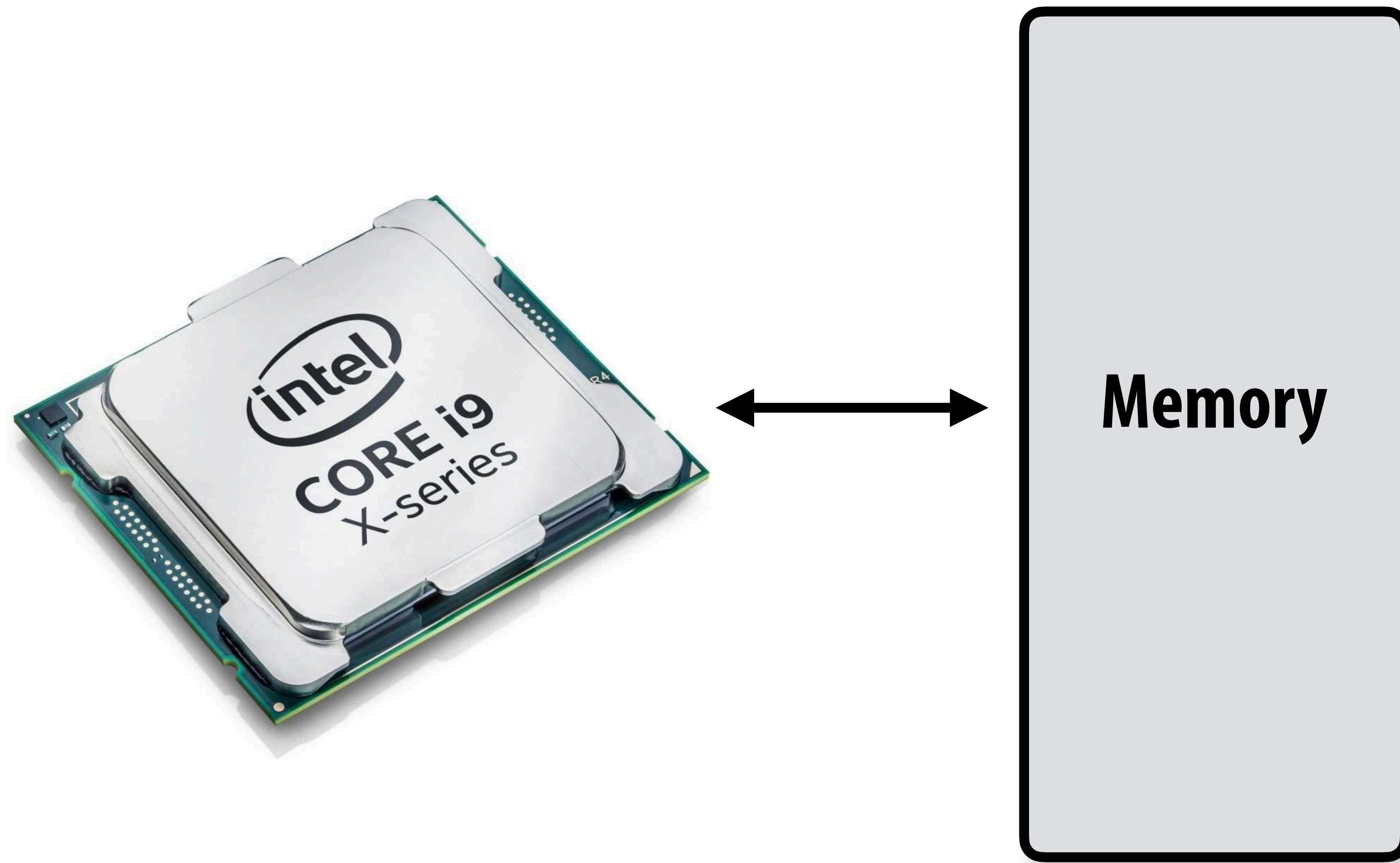
Today

- **Today we're talking computer architecture... from a software engineer's perspective**

- **Key concepts about how modern parallel processors achieve high throughput**
  - Two concern parallel execution (multi-core, SIMD parallel execution)
  - One addresses the challenges of memory latency (multi-threading)

- **Understanding these basics will help you**
  - Understand and optimize the performance of your parallel programs
  - Gain intuition about what workloads might benefit from fast parallel machines

Today's example program

```
void sinx(int N, int terms, float* x, float* y)
{
  for (int i=0; i<N; i++)
  {
    float value = x[i];
    float numer = x[i] * x[i] * x[i];
    int denom = 6;  // 3!
    int sign = -1;

    for (int j=1; j<=terms; j++)
    {
      value += sign * numer / denom;
      numer *= x[i] * x[i];
      denom *= (2*j+2) * (2*j+3);
      sign *= -1;
    }

    y[i] = value;
  }
}
```

Compute $\sin(x)$ **using Taylor expansion:**

$$\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \ldots$$

for each element of an array of N floating-point numbers

| x[0] | x[1] | ... | x[N-2] | x[N-1] |

| y[0] | y[1] | ... | y[N-2] | y[N-1] |

Compile program

```
void sinx(int N, int terms, float* x, float* y)
{
    for (int i=0; i<N; i++)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6;  // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        y[i] = value;
    }
}
```
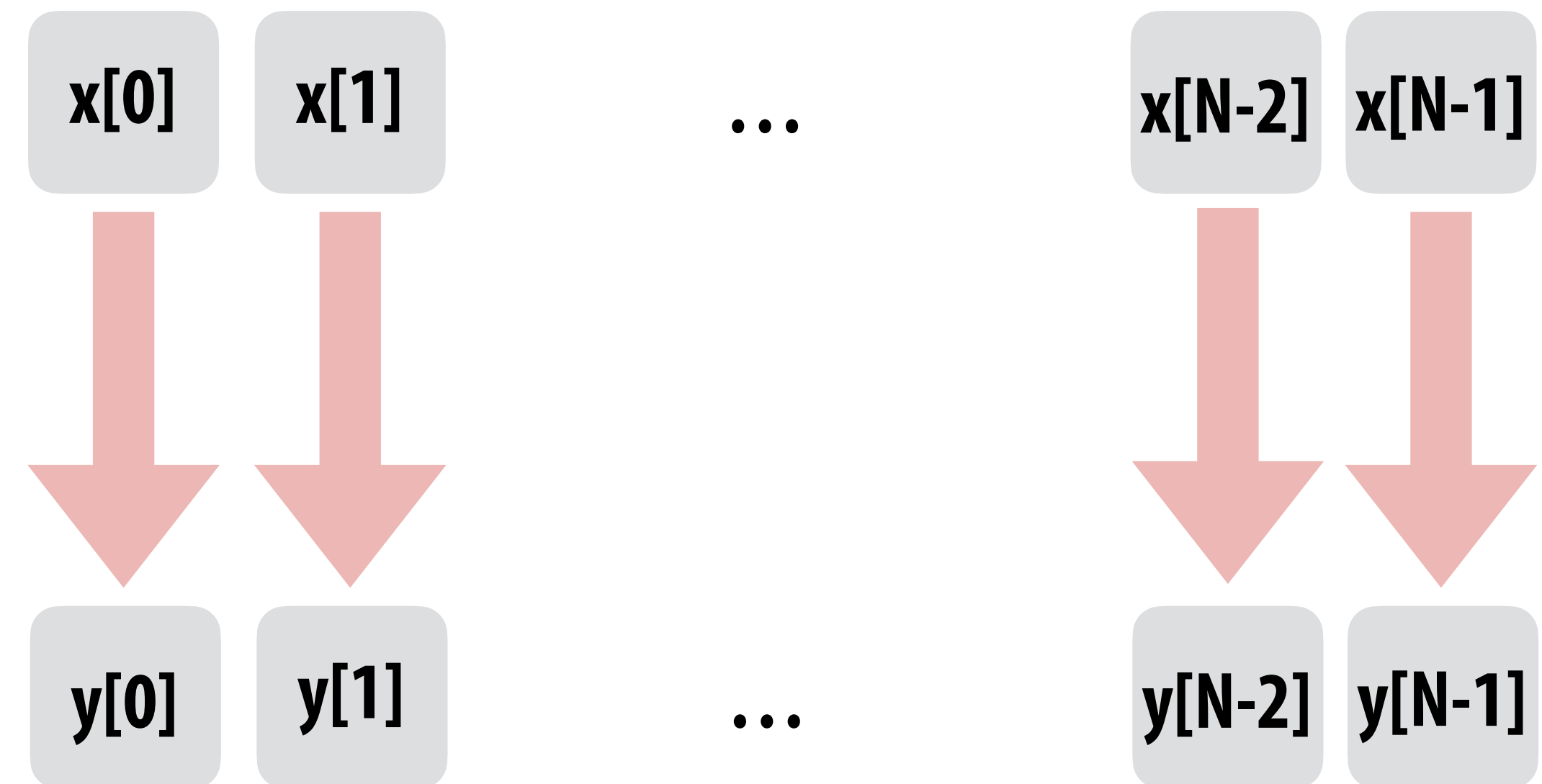
compiler

Compiled instruction stream
(scalar instructions)

x[i]

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

y[i]

Execute program

My very simple processor: executes one instruction per clock

x[i]

↓

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

↓

y[i]

Fetch/Decode

Execution Unit (ALU)

Execution Context

Superscalar processor

The processor shown here can decode and execute two instructions per clock (if independent instructions exist in an instruction stream)

x[i]

Out-of-order control logic

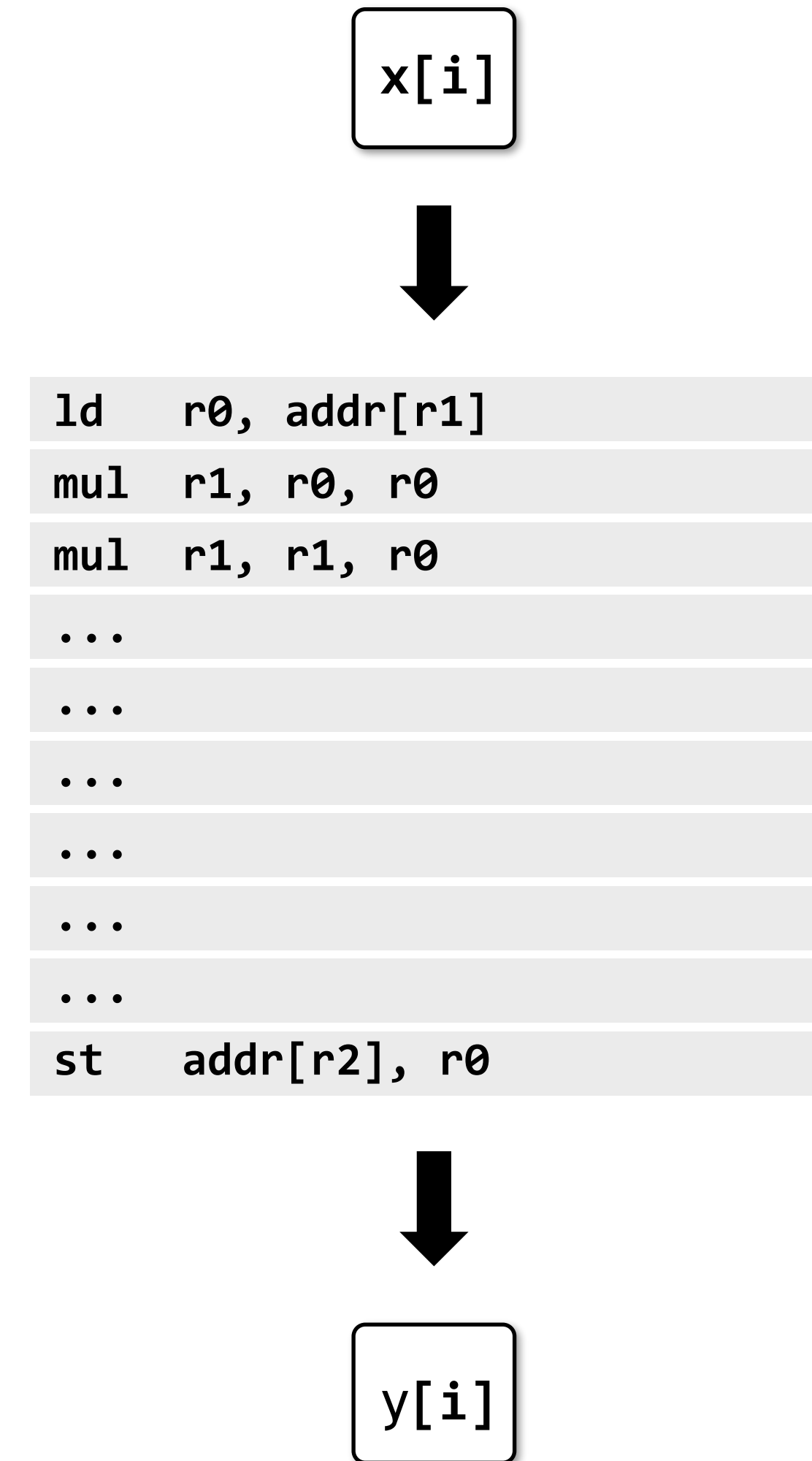| Fetch/ Decode 1 | Fetch/ Decode 2 |

| Exec 1 | Exec 2 |

Execution Context

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

y[i]
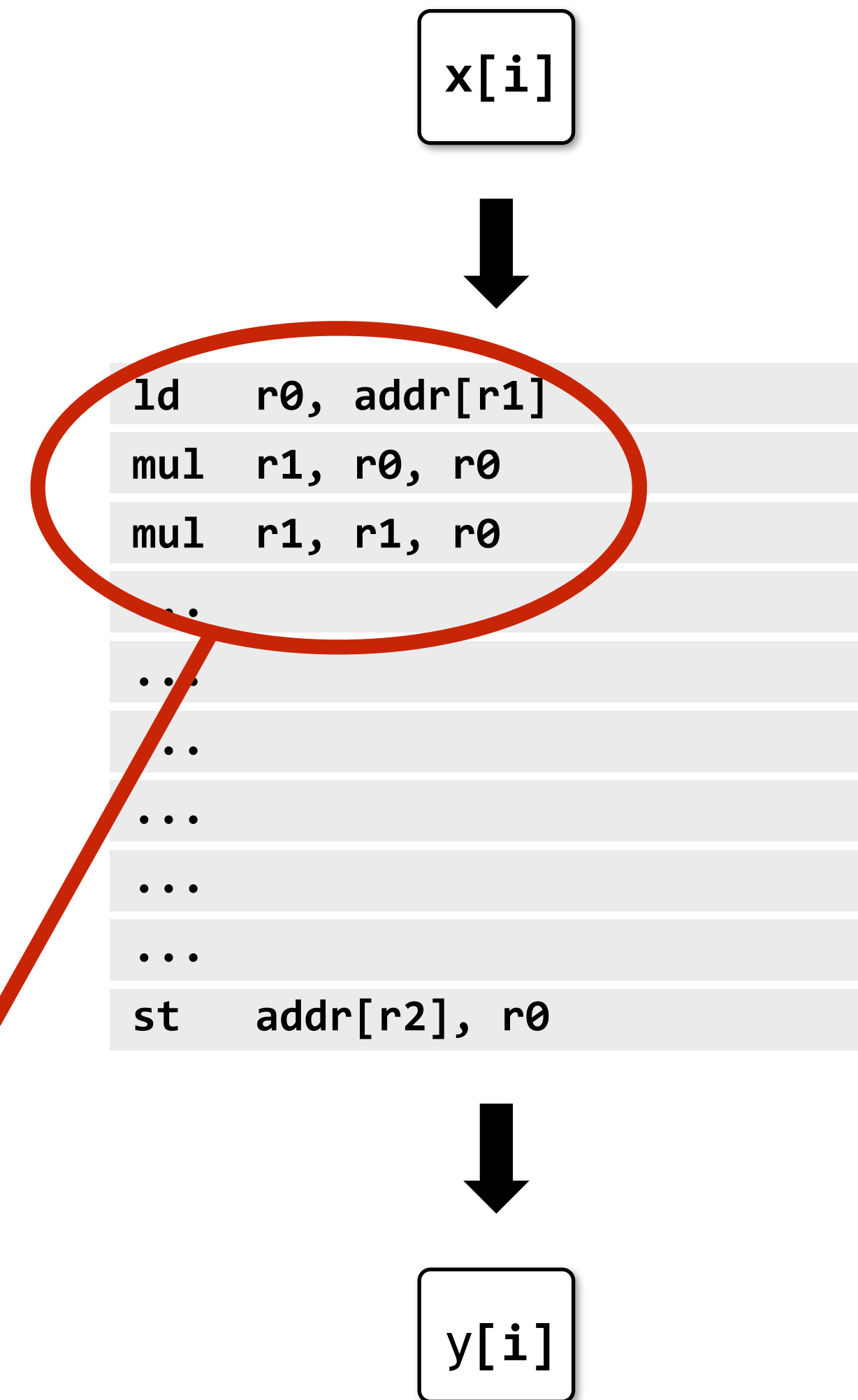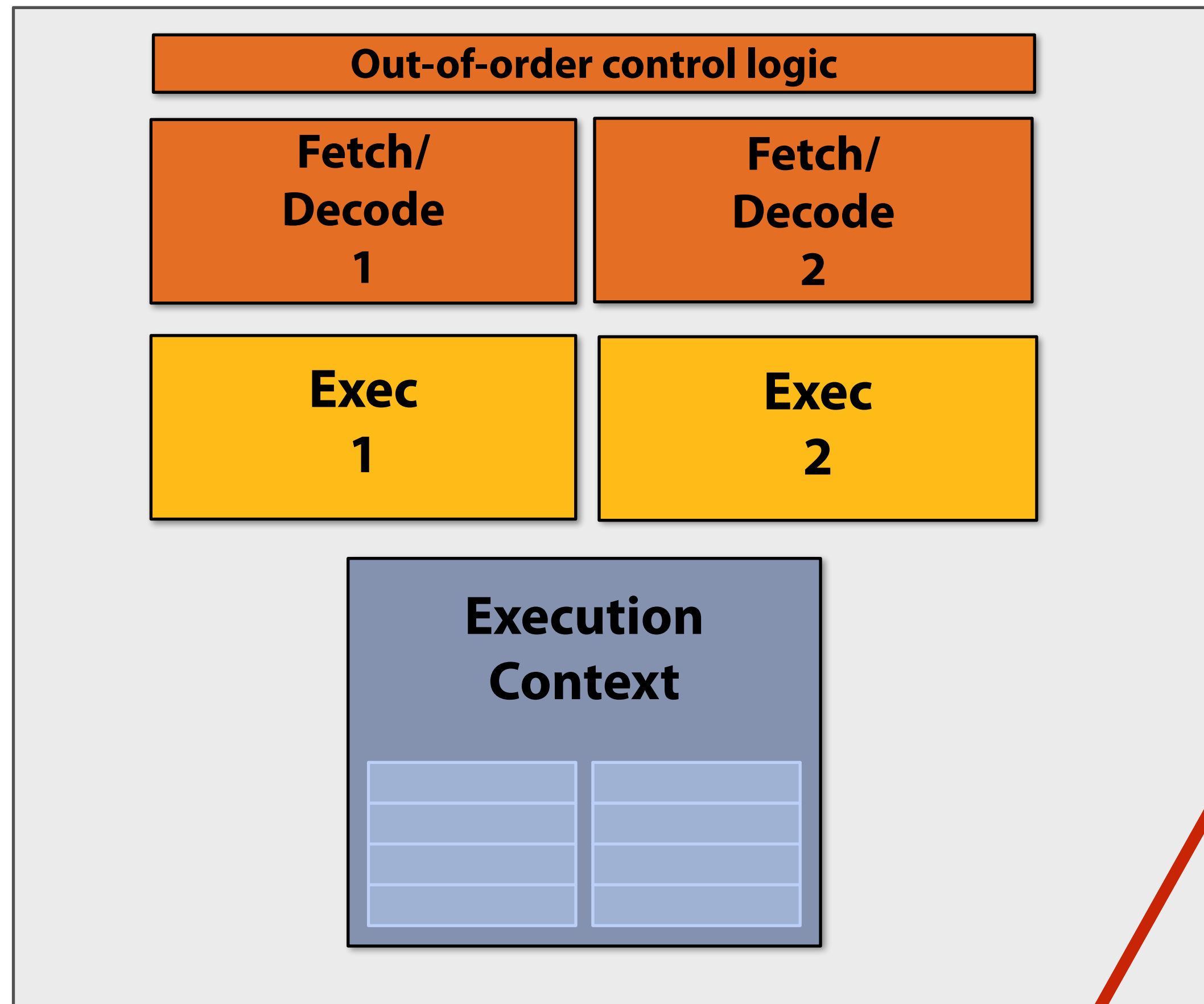
Note: No ILP exists in this region of the program

Pre multi-core era processor

Majority of chip transistors used to perform operations that
help make a <u>single</u> instruction stream run fast

Fetch/ Decode	Fetch/ Decode	Data cache (a big one)
Exec Unit (ALU)	Exec Unit (ALU)	
Execution Context		Out-of-order control logic
		Fancy branch predictor
		Memory pre-fetcher
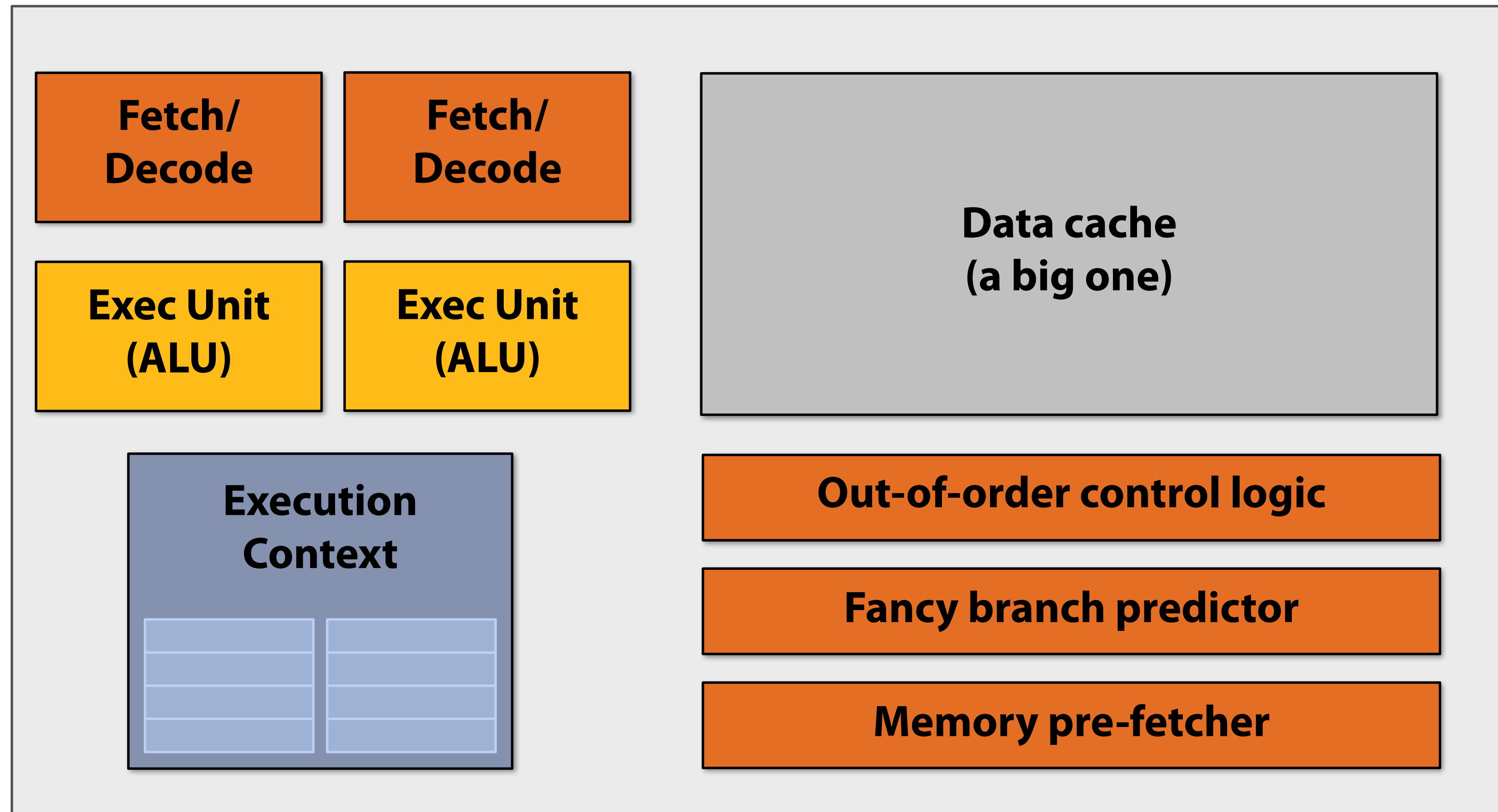
More transistors = larger cache, smarter out-of-order logic, smarter branch predictor, etc.

Multi-core era processor

Fetch/ Decode
Exec Unit (ALU)
Execution Context

Idea #1:

Rather than use transistors to increase sophistication of processor logic that accelerates a single instruction stream (e.g., out-of-order and speculative operations)

Use increasing transistor count to add more cores to the processor
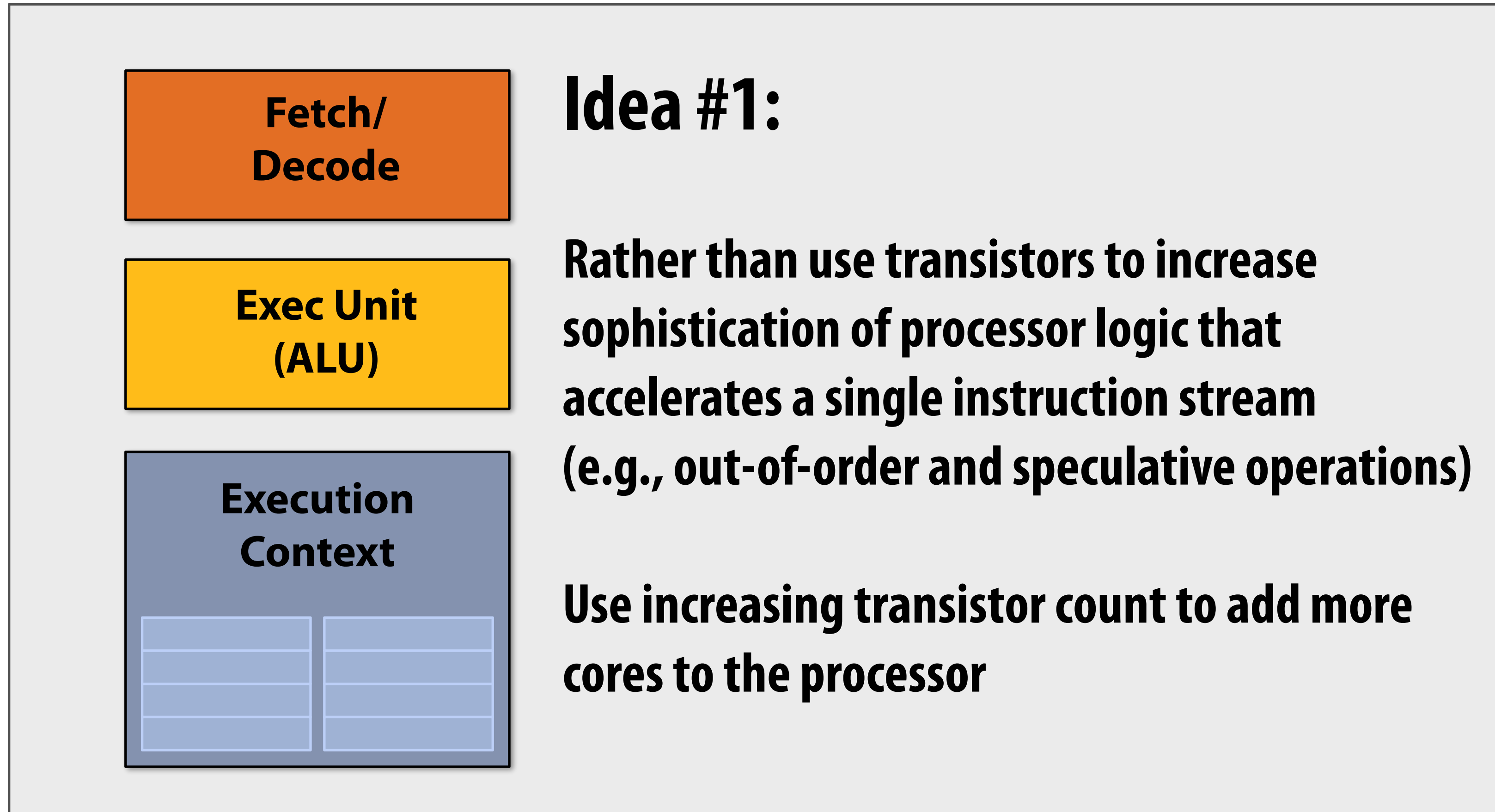
Two cores: compute two elements in parallel

x[i]

Fetch/ Decode
Exec (ALU)
Execution Context

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

x[i]

x[j]

Fetch/ Decode
Exec (ALU)
Execution Context

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

x[j]
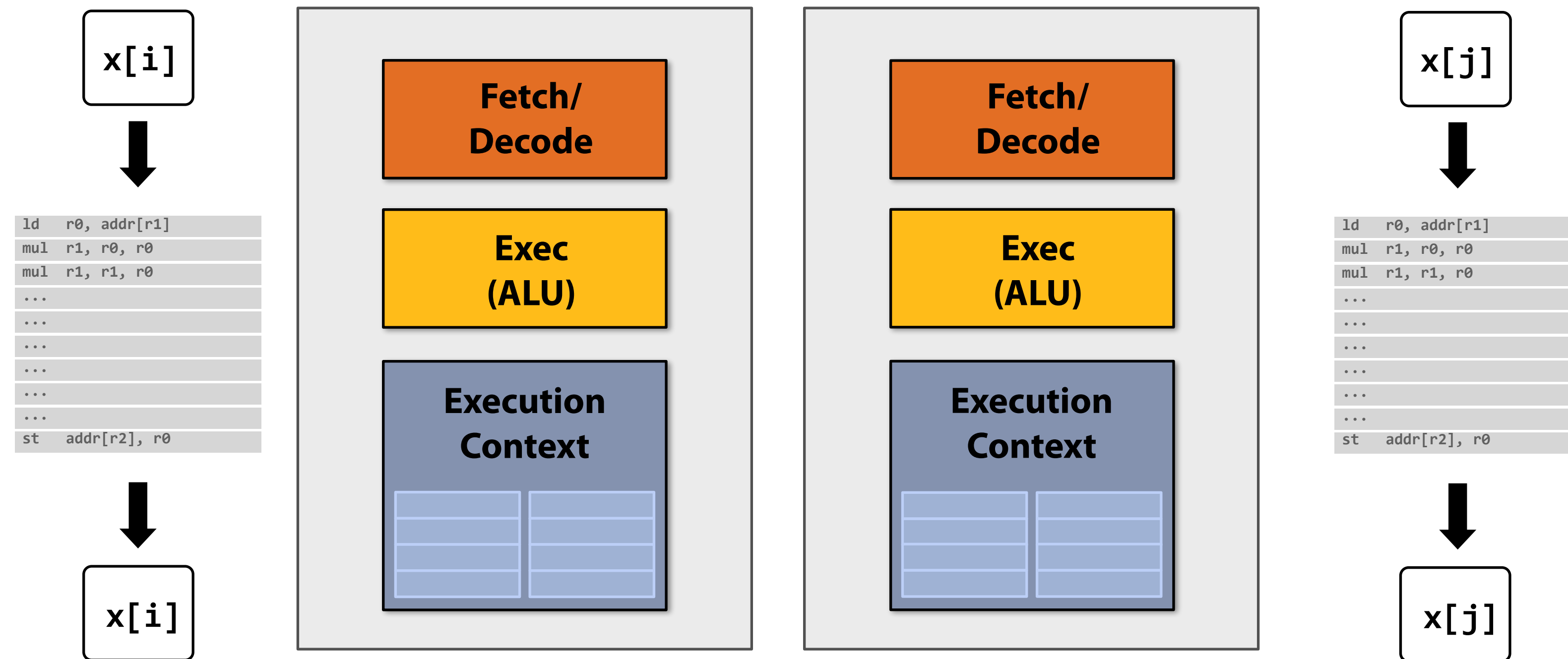
Simpler cores: each core may be slower at running a single instruction stream than our original "fancy" core (e.g., 25% slower)

But there are now two cores: $2 \times 0.75 = 1.5$ (potential for speedup!)

But our program expresses no parallelism

```
void sinx(int N, int terms, float* x, float* y)
{
    for (int i=0; i<N; i++)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6;   // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        y[i] = value;
    }
}
```

This C program will compile to an instruction stream that runs as one thread on one processor core.

If each of the simpler processor cores was 25% slower than the original single complicated one, our program now runs 25% slower than before.

😭

Example: expressing parallelism using C++ threads

```cpp
typedef struct {
    int N;
    int terms;
    float* x;
    float* y;
} my_args;

void my_thread_func(my_args* args)
{
    sinx(args->N, args->terms, args->x, args->y); // do work
}

void parallel_sinx(int N, int terms, float* x, float* y)
{
    std::thread my_thread;
    my_args args;

    args.N = N/2;
    args.terms = terms;
    args.x = x;
    args.y = y;

    my_thread = std::thread(my_thread_func, &args); // launch thread
    sinx(N - args.N, terms, x + args.N, y + args.N); // do work on main thread
    my_thread.join();  // wait for thread to complete
}
```

```cpp
void sinx(int N, int terms, float* x, float* y)
{
    for (int i=0; i<N; i++)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6;  // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        y[i] = value;
    }
}
```

Data-parallel expression (in Kayvon's fictitious programming language with a "forall" construct)

```
void sinx(int N, int terms, float* x, float* y)
{
    // declares that loop iterations are independent
    forall (int i from 0 to N)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6;  // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        y[i] = value;
    }
}
```
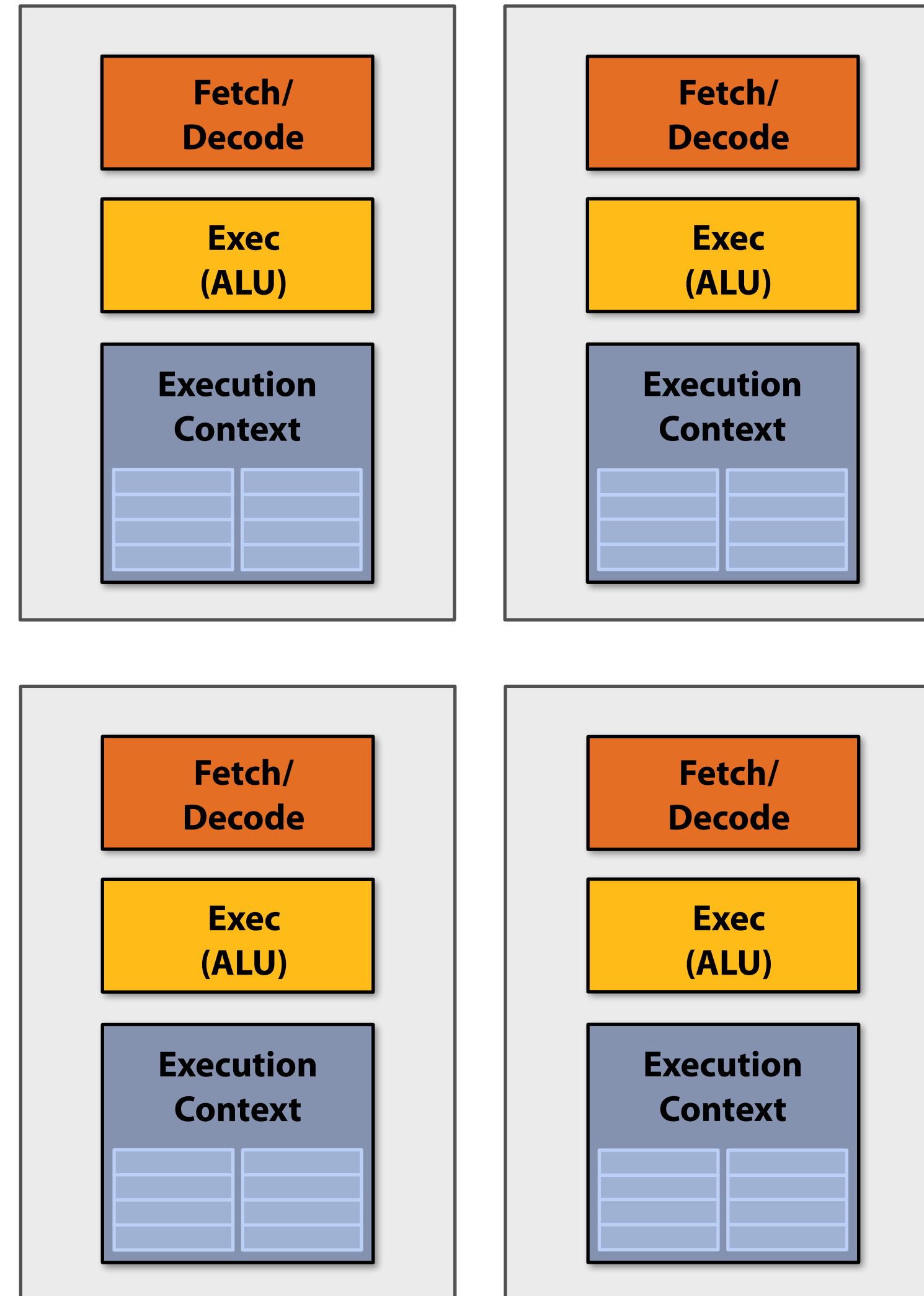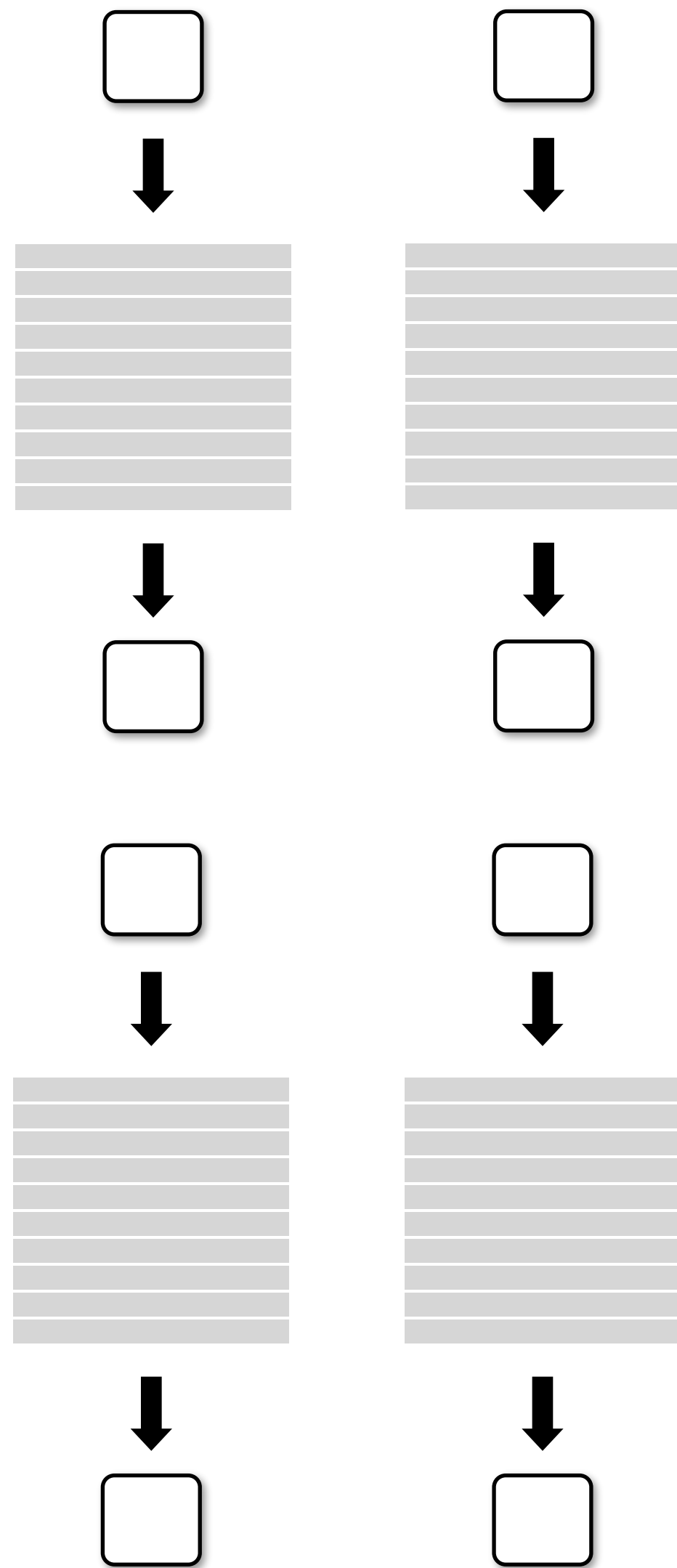
In this code, loop iterations are declared by the programmer to be independent (see the 'forall')

With this information, you could imagine how a compiler might automatically generate threaded code for you.

Four cores: compute four elements in parallel

Sixteen cores: compute sixteen elements in parallel
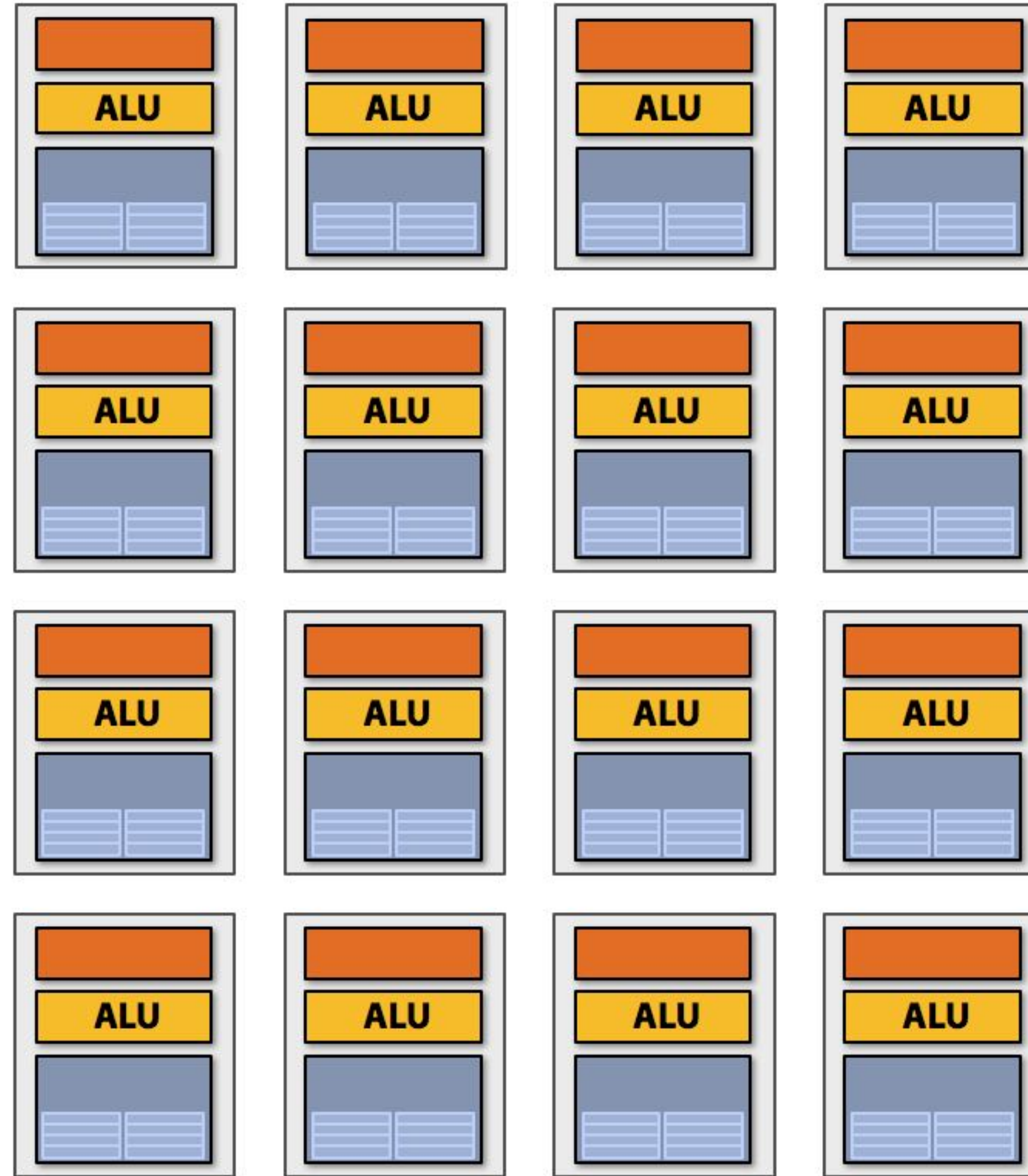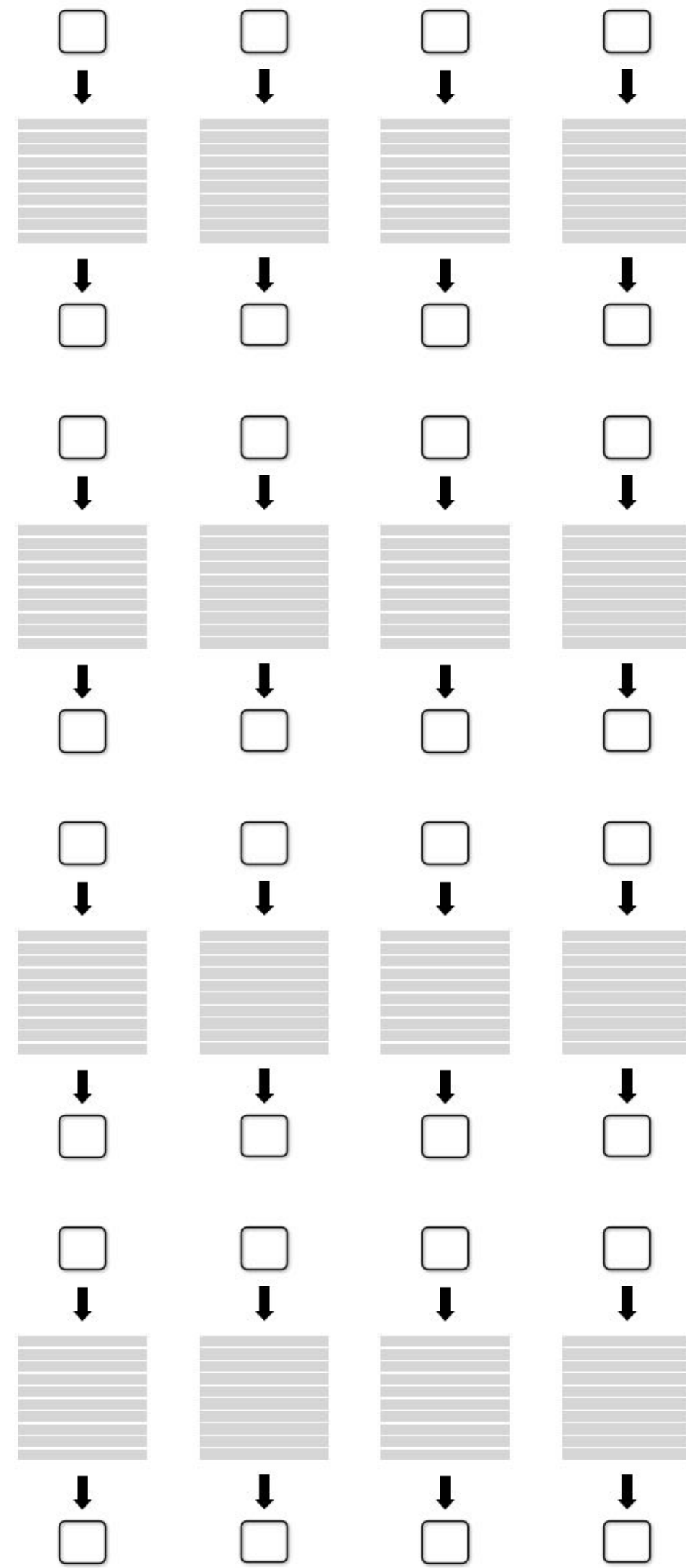
Sixteen cores, sixteen simultaneous instruction streams

Example: multi-core CPU

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

Core 1 Core 2 Core 3 Core 4 Core 5

Core 6 Core 7 Core 8 Core 9 Core 10

Multi-core GPU

GeForce RTX 4090 (2022)

144 processing blocks (called SMs)

Apple A15 Bionic

Two "big cores" + four "small" cores

2 "big" CPU cores
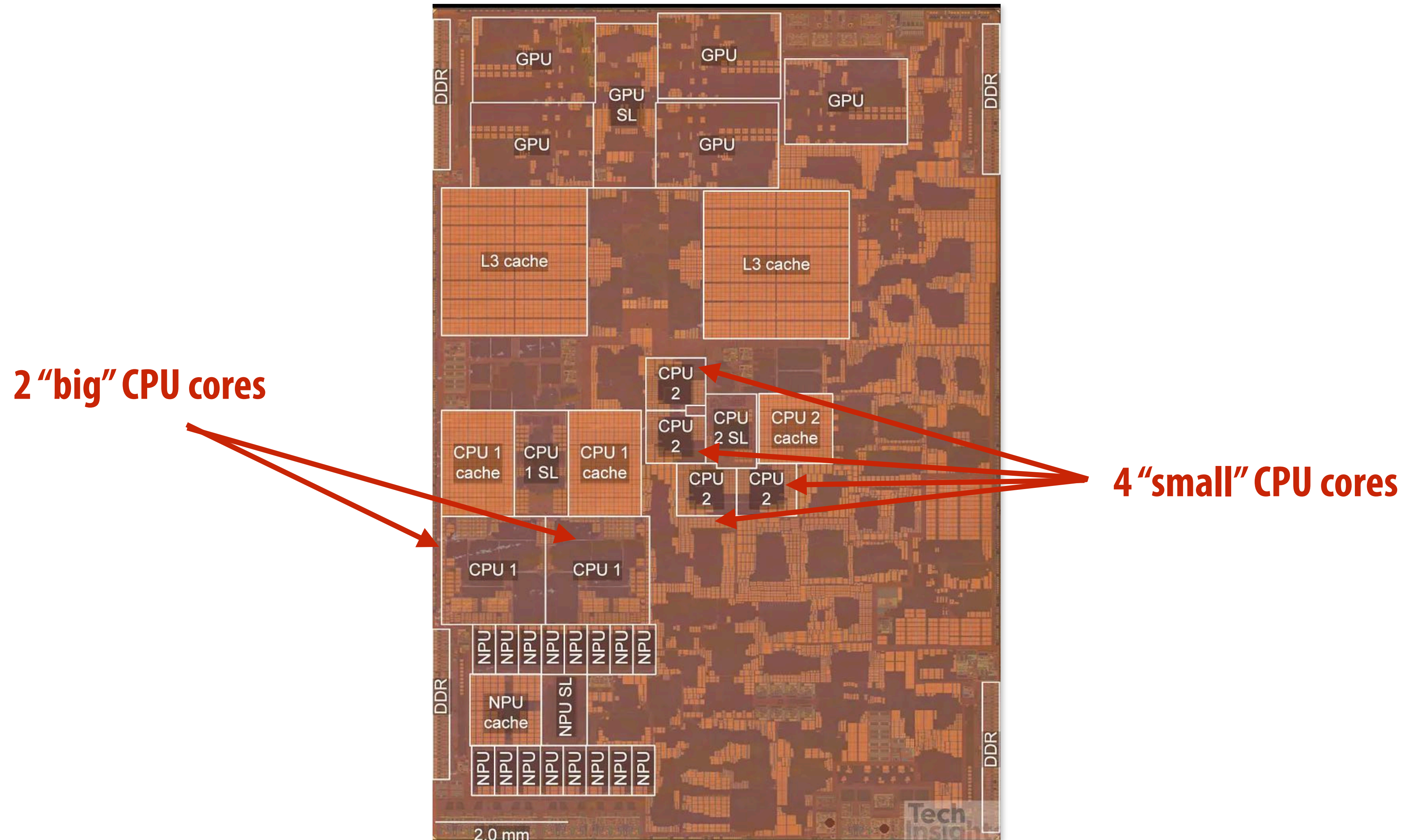
4 "small" CPU cores

Apple M1 Silicon (also heterogenous cores)

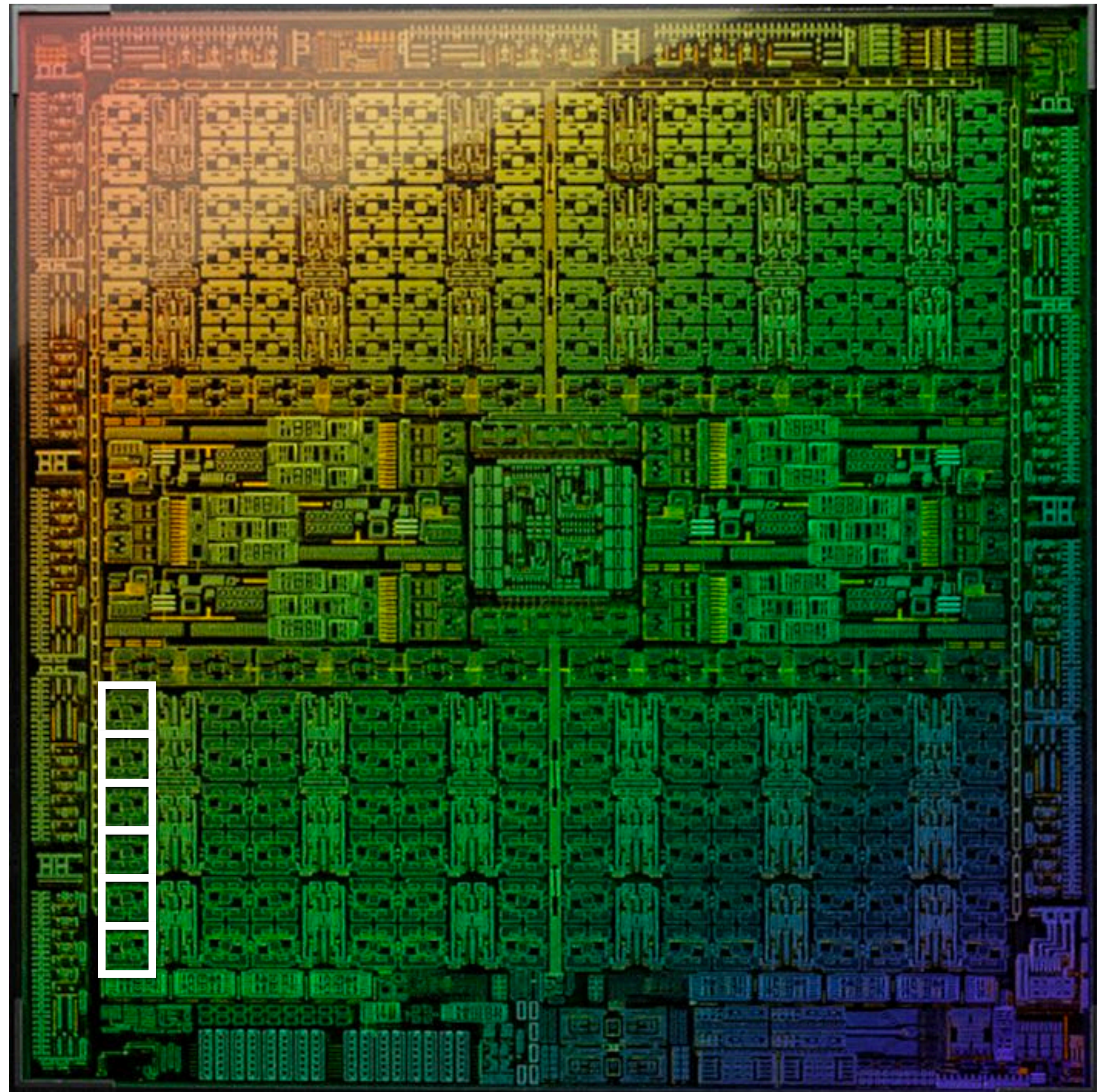**Four "big cores" + four "small" CPU cores ***

4"big" cores

4 "small" CPU cores
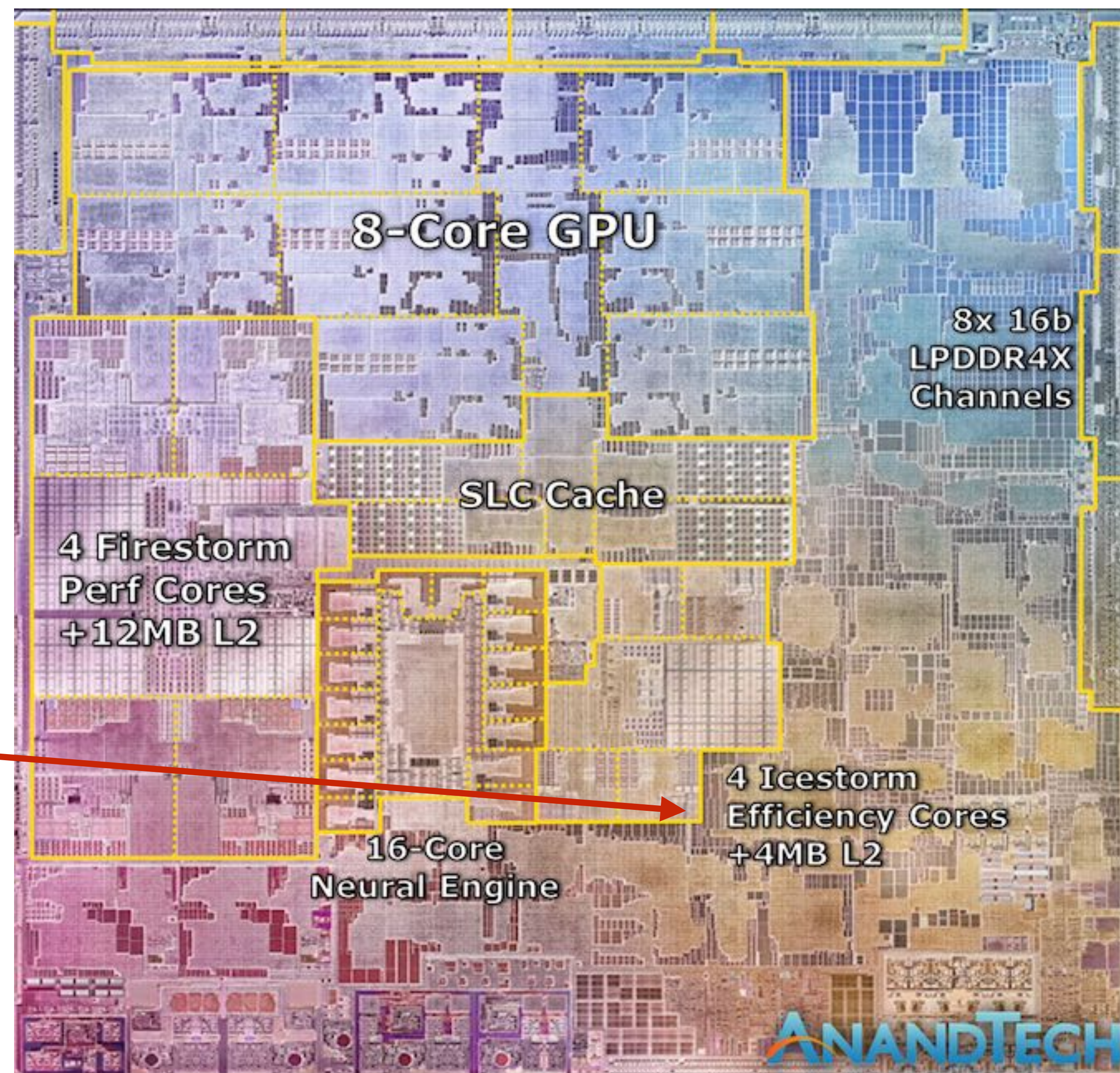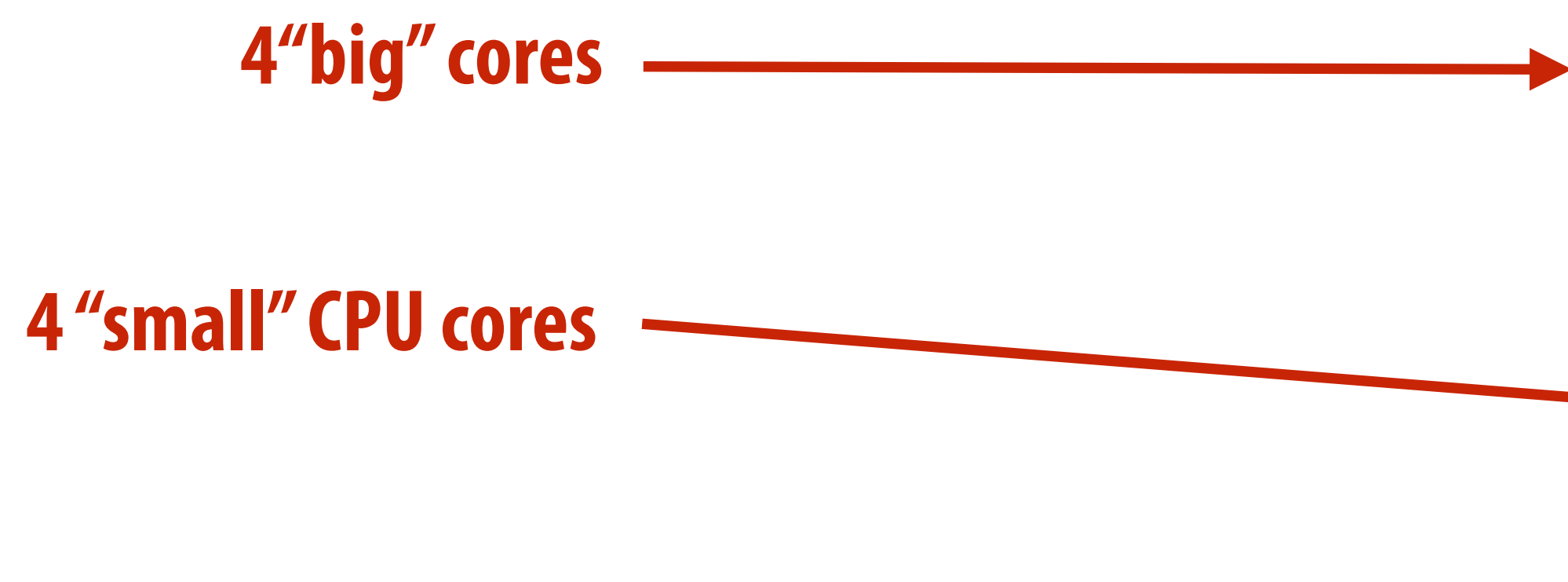
***** not even counting the GPU cores or the neural acceleration hardware**

Data-parallel expression (in Kayvon's fictitious programming language with a "forall" construct)

```
void sinx(int N, int terms, float* x, float* result)
{
    // declares that loop iterations are independent
    forall (int i from 0 to N)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6;  // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}
```

Another interesting property of this code:

Parallelism is across iterations of the loop.

All the iterations of the loop carry out the exact same sequence of instructions (defined by the loop body), but on different input data given by x[i]

(the loop body computes sine(x[i]))

Add execution units (ALUs) to increase compute capability
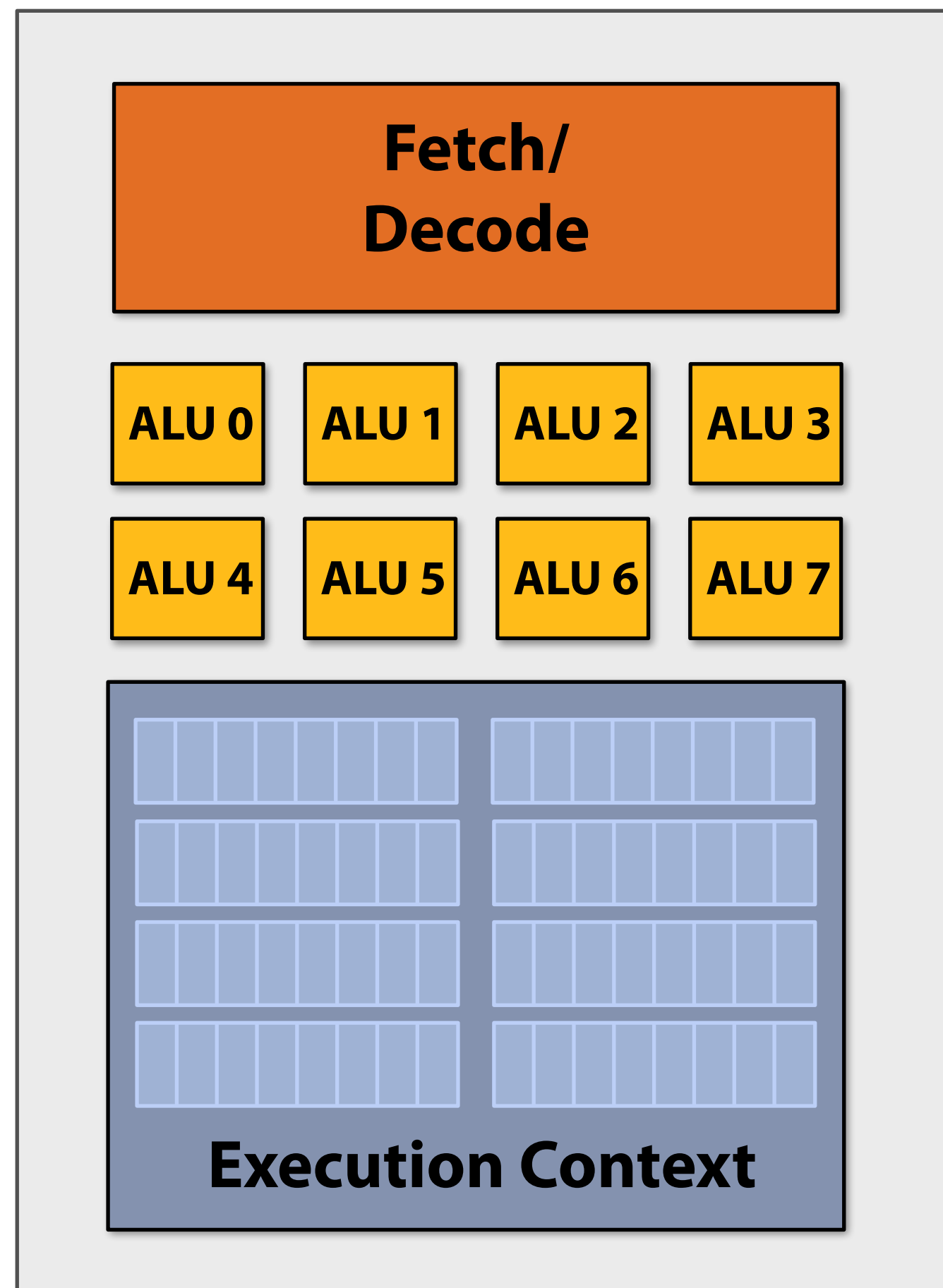
Idea #2:
Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

Single instruction, multiple data

Same instruction broadcast to all ALUs
This operation is executed in parallel on all ALUs

Recall our original scalar program

```
void sinx(int N, int terms, float* x, float* y)
{
    for (int i=0; i<N; i++)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6;  // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        y[i] = value;
    }
}
```

Original compiled program:

Processes one array element using scalar instructions on scalar registers (e.g., 32-bit floats)

x[i]

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

y[i]

Vector program (using AVX intrinsics)

```c
#include <immintrin.h>

void sinx(int N, int terms, float* x, float* y)
{
    float three_fact = 6;  // 3!
    for (int i=0; i<N; i+=8)
    {
        __m256 origx = _mm256_load_ps(&x[i]);
        __m256 value = origx;
        __m256 numer = _mm256_mul_ps(origx, _mm256_mul_ps(origx, origx));
        __m256 denom = _mm256_broadcast_ss(&three_fact);
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            // value += sign * numer / denom
            __m256 tmp = _mm256_div_ps(_mm256_mul_ps(_mm256_set1ps(sign), numer), denom);
            value = _mm256_add_ps(value, tmp);

            numer = _mm256_mul_ps(numer, _mm256_mul_ps(origx, origx));
            denom = _mm256_mul_ps(denom, _mm256_broadcast_ss((2*j+2) * (2*j+3)));
            sign *= -1;
        }
        _mm256_store_ps(&y[i], value);
    }
}
```

Intrinsic datatypes and functions available to C programmers

Intrinsic functions operate on vectors of eight 32-bit values (e.g., vector of 8 floats)
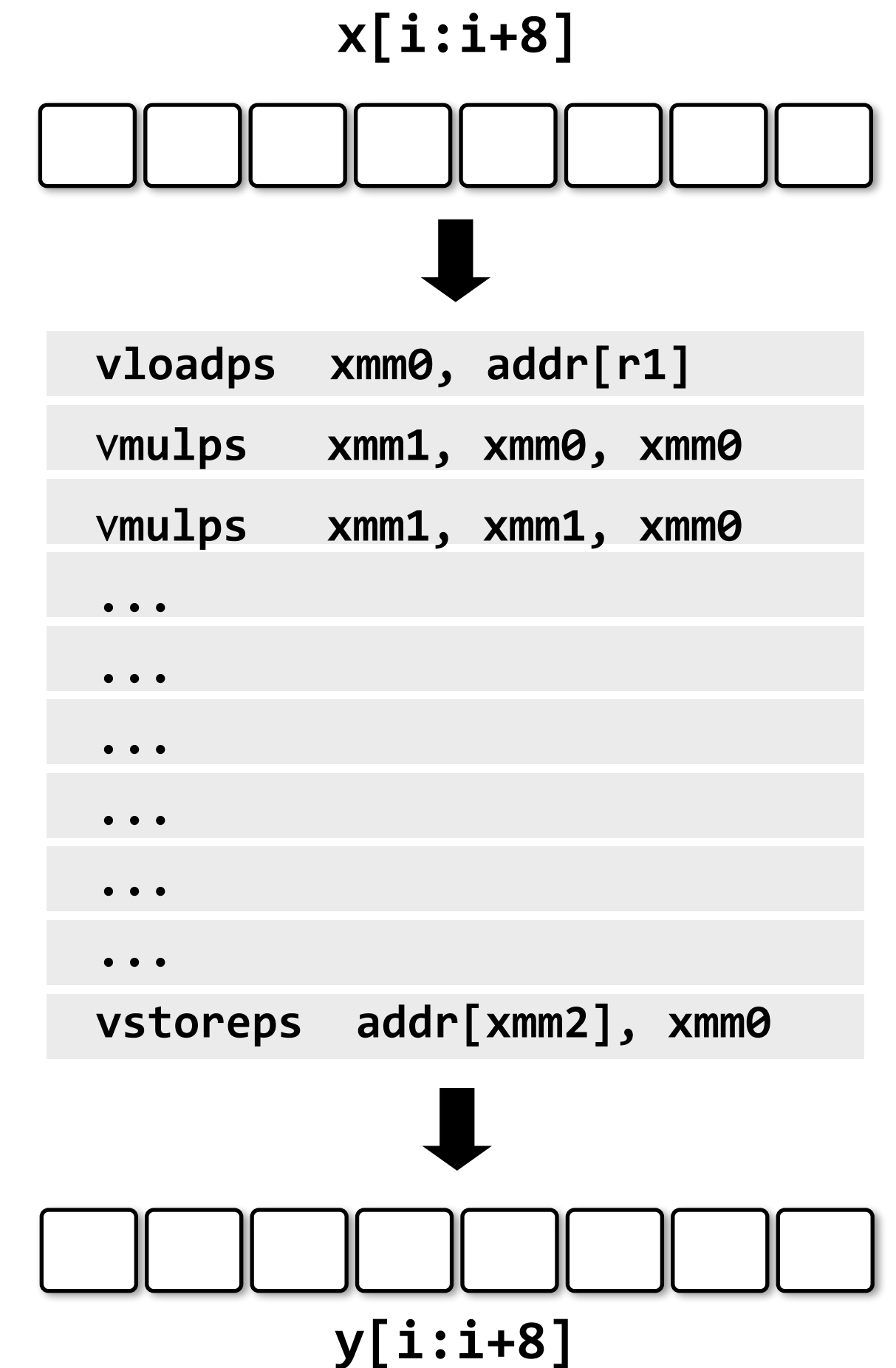
Vector program (using AVX intrinsics)

```c
#include <immintrin.h>


void sinx(int N, int terms, float* x, float* y)
{
    float three_fact = 6;  // 3!
    for (int i=0; i<N; i+=8)
    {
        __m256 origx = _mm256_load_ps(&x[i]);
        __m256 value = origx;
        __m256 numer = _mm256_mul_ps(origx, _mm256_mul_ps(origx, origx));
        __m256 denom = _mm256_broadcast_ss(&three_fact);
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            // value += sign * numer / denom
            __m256 tmp = _mm256_div_ps(_mm256_mul_ps(_mm256_set1ps(sign), numer), denom);
            value = _mm256_add_ps(value, tmp);

            numer = _mm256_mul_ps(numer, _mm256_mul_ps(origx, origx));
            denom = _mm256_mul_ps(denom, _mm256_broadcast_ss((2*j+2) * (2*j+3)));
            sign *= -1;
        }
        _mm256_store_ps(&y[i], value);
    }
}
```

x[i:i+8]


```
vloadps   xmm0, addr[r1]
vmulps    xmm1, xmm0, xmm0
vmulps    xmm1, xmm1, xmm0
...
...
...
...
...
...
vstoreps  addr[xmm2], xmm0
```

y[i:i+8]

Compiled program:

Processes eight array elements simultaneously using vector instructions on 256-bit vector registers

16 SIMD cores: 128 elements in parallel
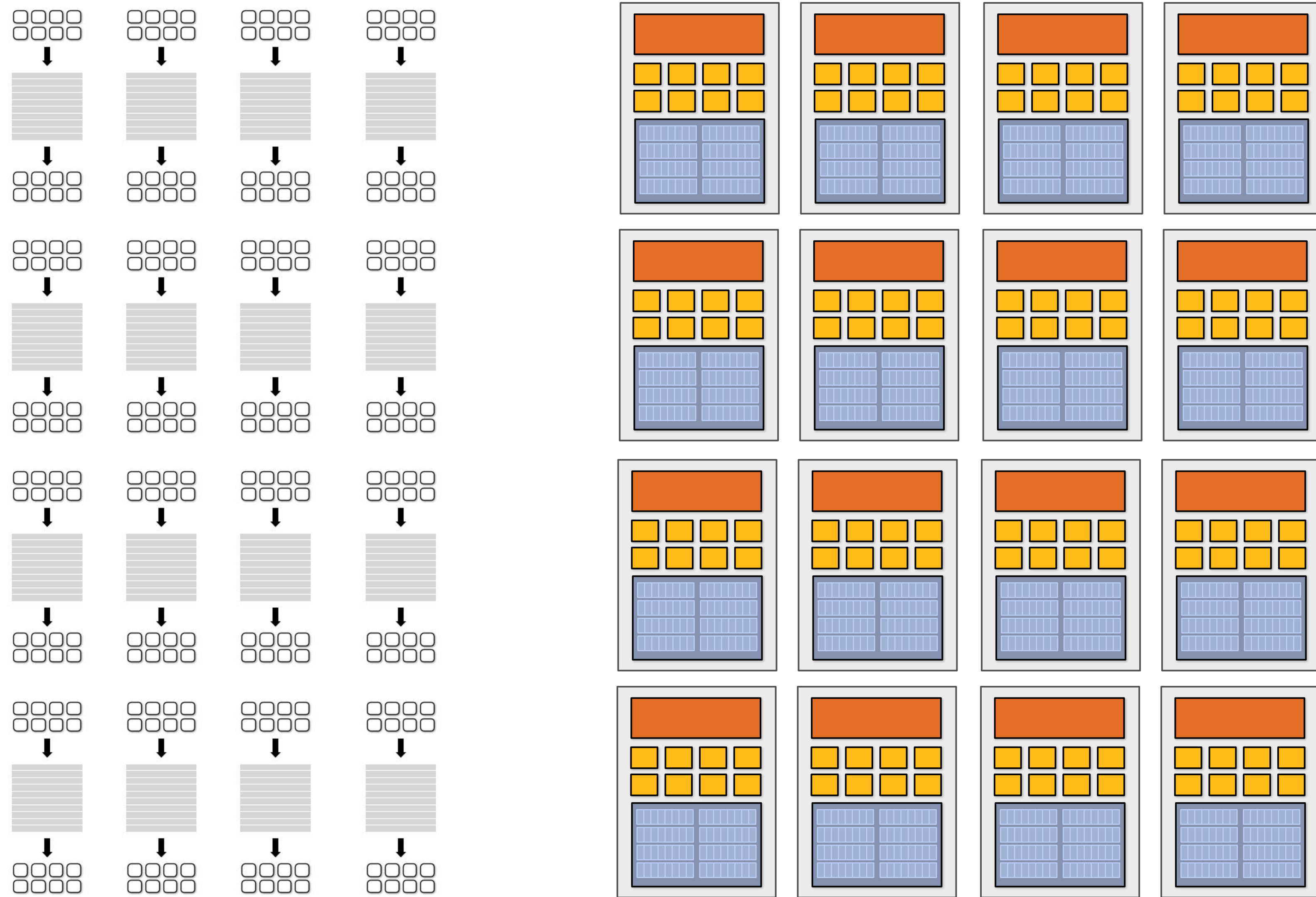
16 cores, 128 ALUs, 16 simultaneous instruction streams

Data-parallel expression (in Kayvon's fictitious programming language with a "forall" construct)

```
void sinx(int N, int terms, float* x, float* result)
{
    // declares that loop iterations are independent
    forall (int i from 0 to N)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6;  // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}
```

The program's use of "forall" declares to the compiler that loop iterations are independent, and that same loop body will be executed on a large number of data elements.

This abstraction can facilitate automatic generation of <u>both</u> multi-core parallel code, and vector instructions to make use of SIMD processing capabilities within a core.

What about conditional execution?

Time (clocks)

| 1 | 2 | ... | | | | ... | 8 |

ALU 1 ALU 2 ...   ... ALU 8
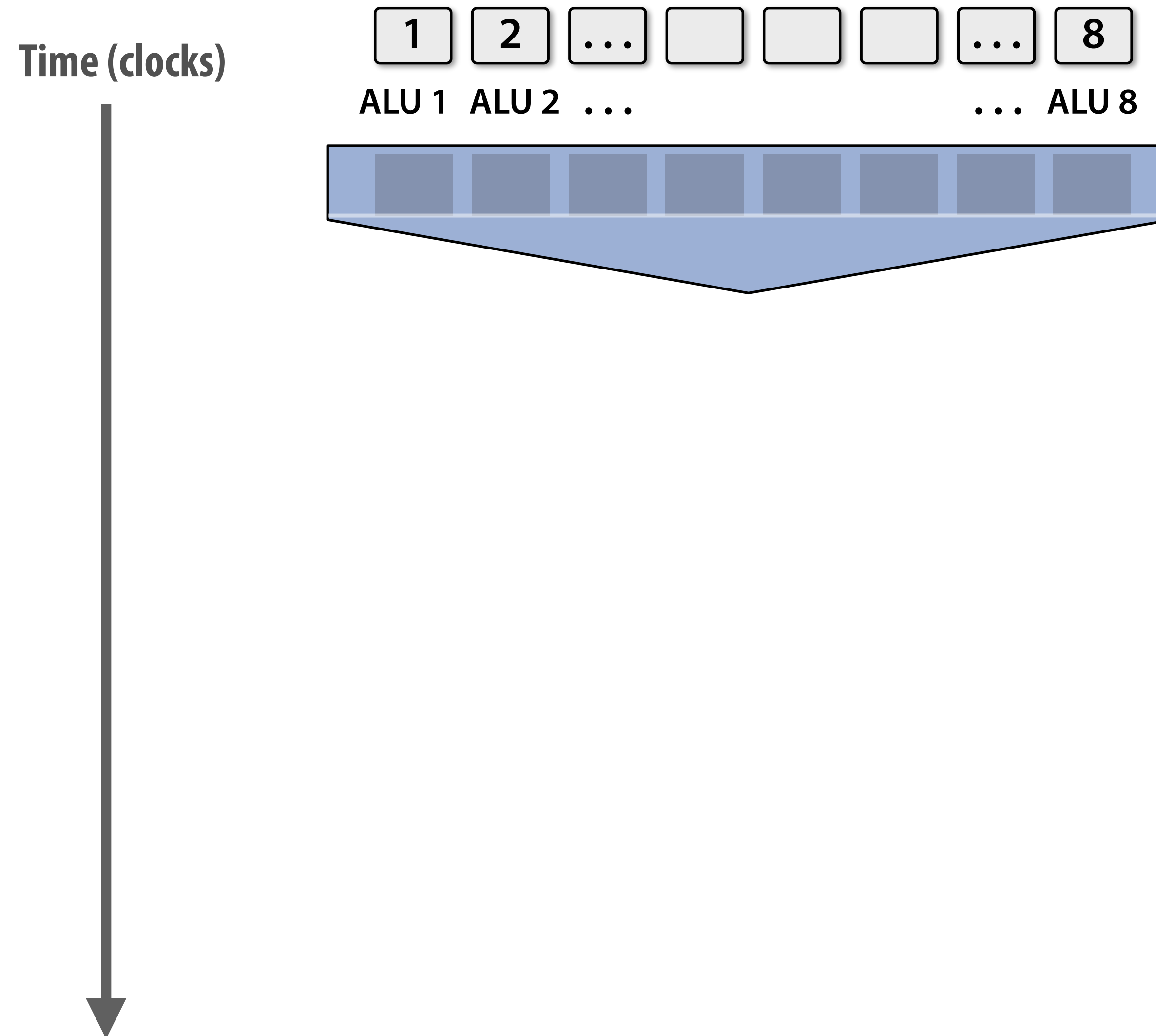
```
forall (int i from 0 to N) {

    float t = x[i];

    <unconditional code>

    if (t > 0.0) {

        t = t * t;

        t = t * 50.0;

        t = t + 100.0;
    } else {
        t = t + 30.0;

        t = t / 10.0;
    }

    <resume unconditional code>

    y[i] = t;

}
```

What about conditional execution?

```
1  2  ...  □  □  □  ...  8
ALU 1 ALU 2 ...        ... ALU 8
```

T T F T F F F F

```
forall (int i from 0 to N) {

    float t = x[i];

    <unconditional code>

    if (t > 0.0) {

        t = t * t;

        t = t * 50.0;

        t = t + 100.0;
    } else {
        t = t + 30.0;

        t = t / 10.0;
    }

    <resume unconditional code>

    y[i] = t;

}
```

Mask (discard) output of ALU

Time (clocks)

| 1 | 2 | ... | | | | ... | 8 |

ALU 1 ALU 2 ...   ... ALU 8

Not all ALUs do useful work!

Worst case: 1/8 peak performance

```
forall (int i from 0 to N) {
    float t = x[i];

    <unconditional code>

    if (t > 0.0) {
        t = t * t;

        t = t * 50.0;

        t = t + 100.0;
    } else {
        t = t + 30.0;

        t = t / 10.0;
    }

    <resume unconditional code>
    y[i] = t;
}
```

After branch: continue at full performance

Time (clocks)
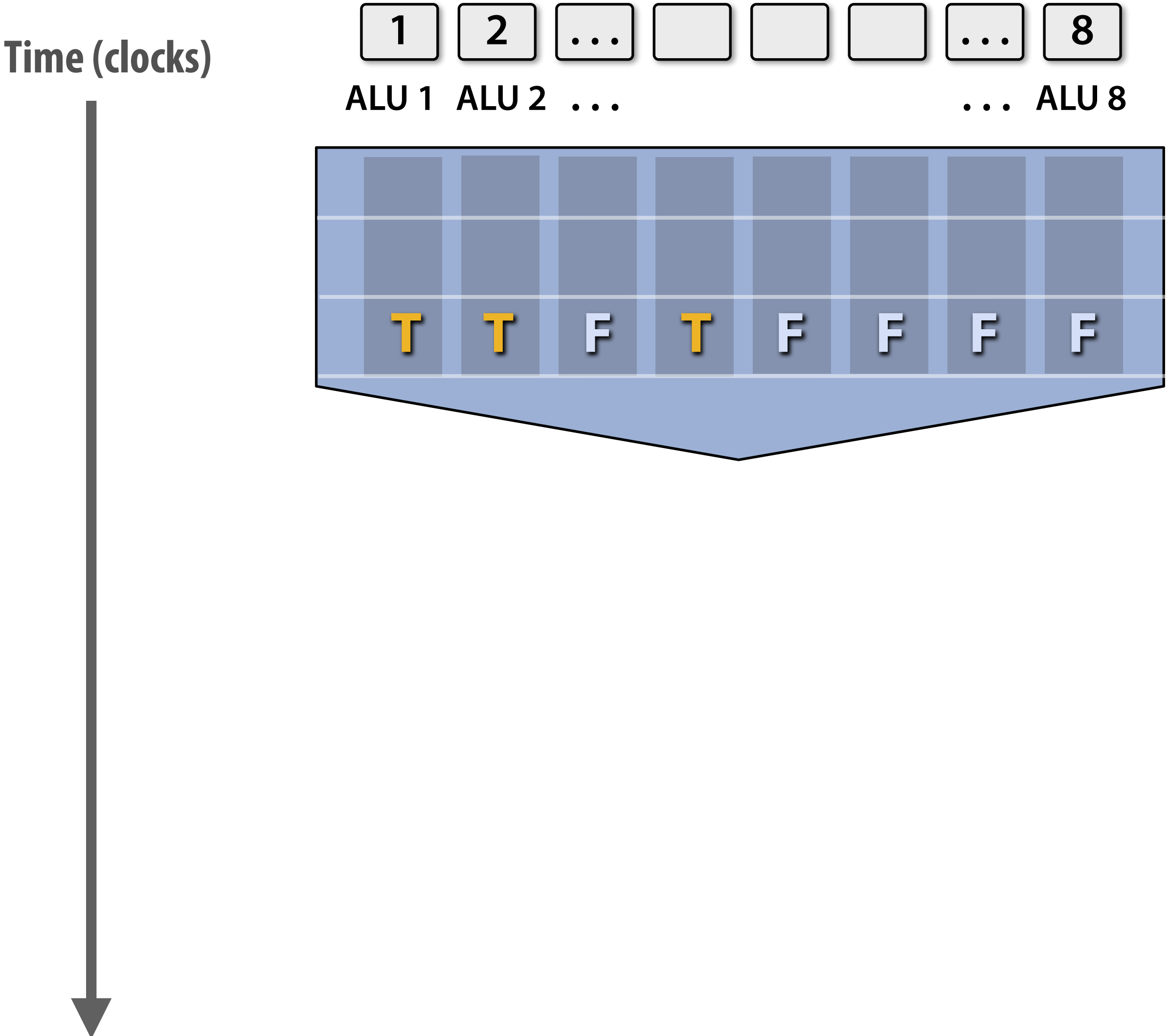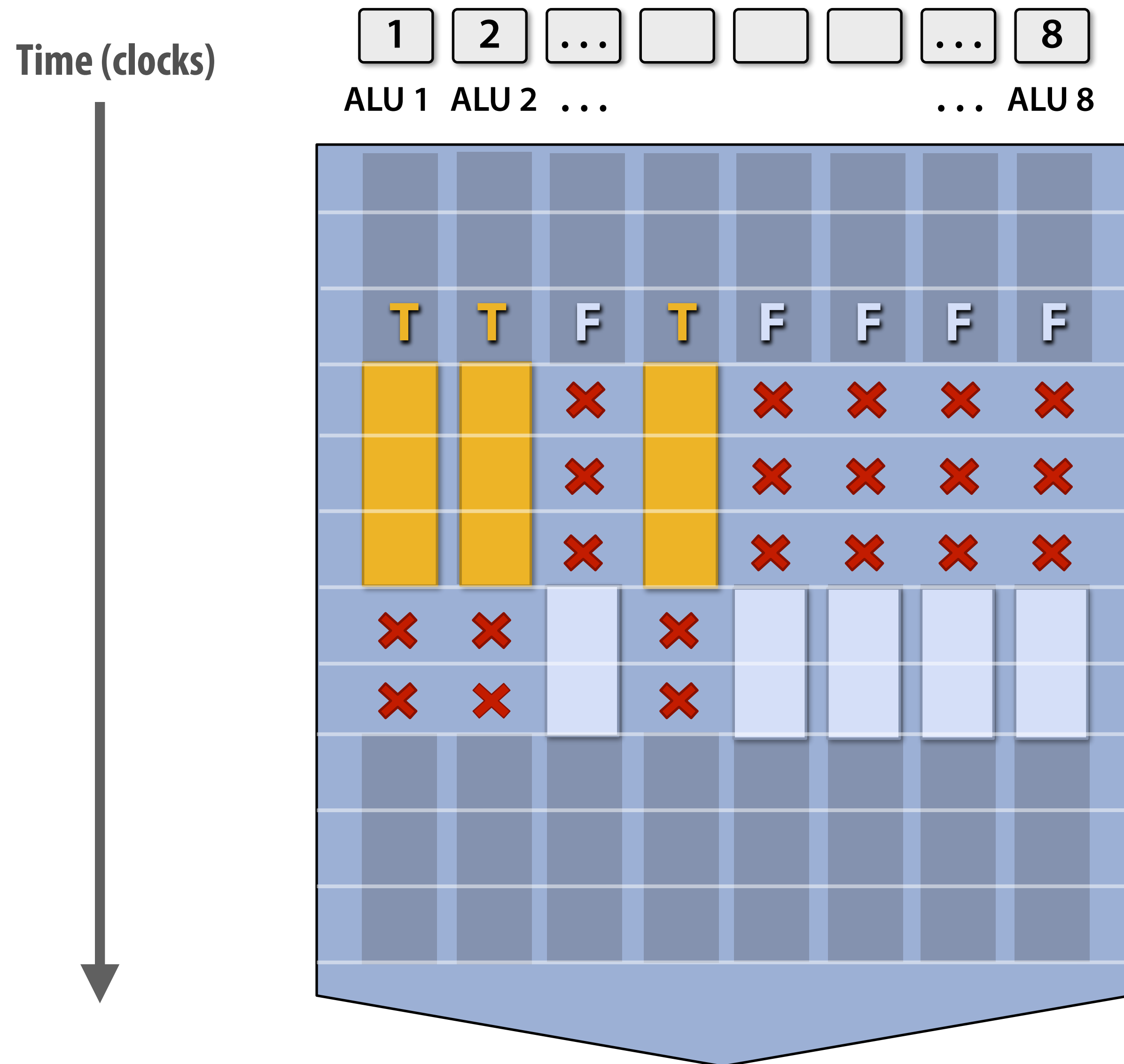


```
forall (int i from 0 to N) {
    float t = x[i];

    <unconditional code>

    if (t > 0.0) {
        t = t * t;

        t = t * 50.0;

        t = t + 100.0;
    } else {
        t = t + 30.0;

        t = t / 10.0;
    }

    <resume unconditional code>
    y[i] = t;
}
```
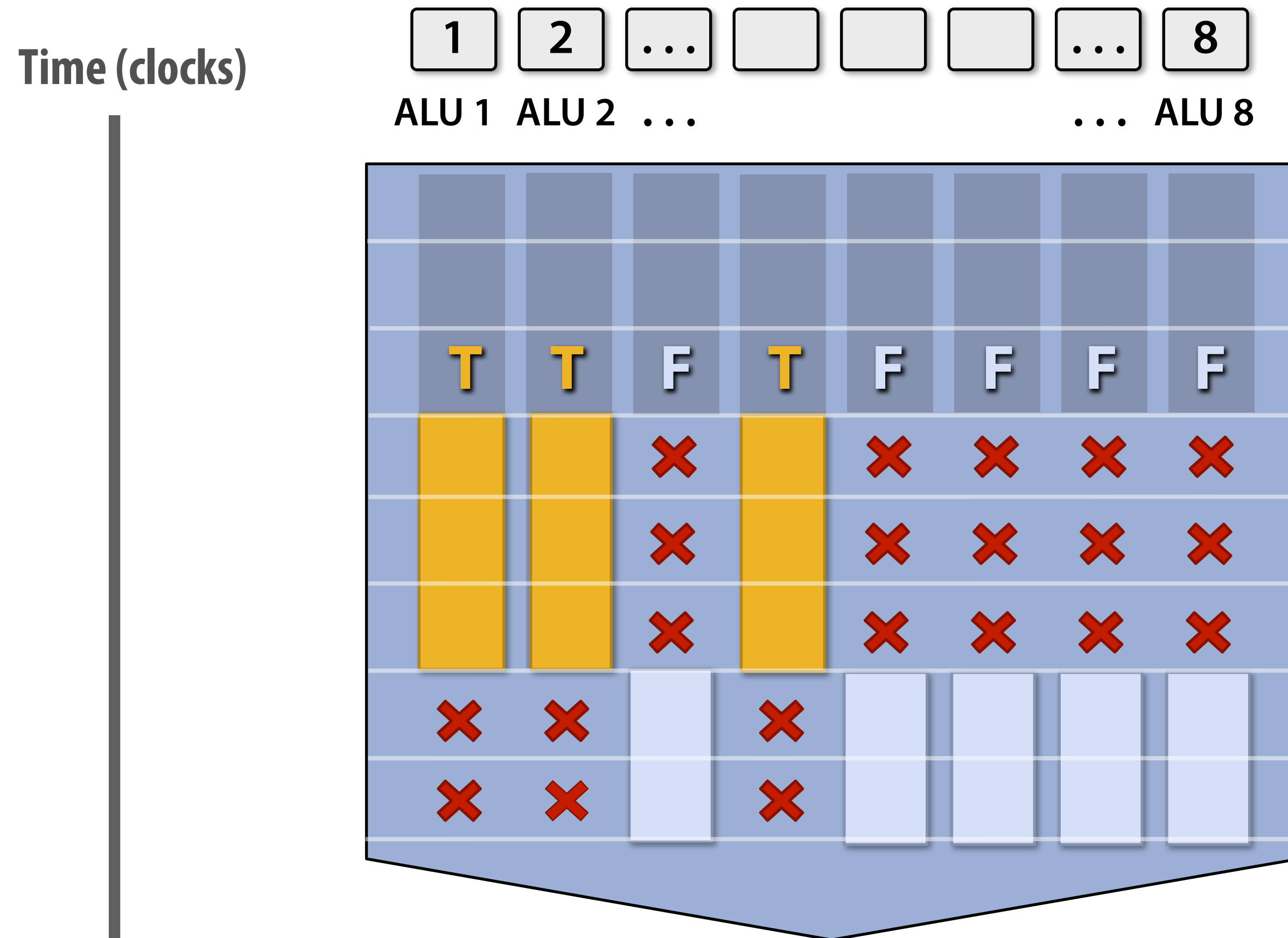
Breakout question

Can you think of piece of code that yields the worst case performance on a processor with 8-wide SIMD execution?

Hint: can you create it using only a single "if" statement?
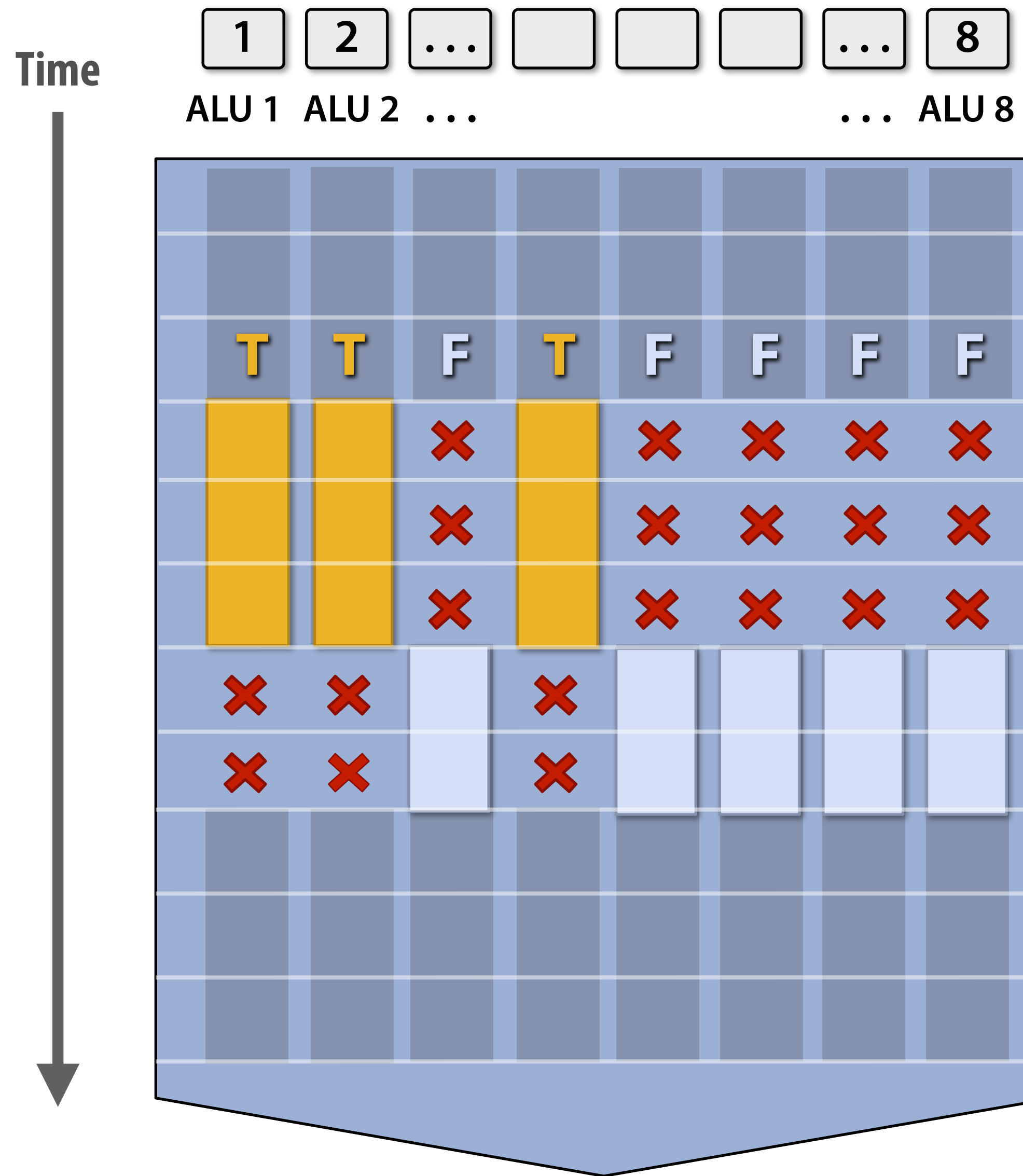
```
forall (int i from 0 to N) {

    float t = x[i];

    <unconditional code>

    if (t > 0.0) {

        ???

    } else {

        ???

    }

    <resume unconditional code>
    y[i] = t;
}
```

Some common jargon

- **Instruction stream coherence ("coherent execution")**
  - Property of a program where the same instruction sequence applies to many data elements
  - Coherent execution IS NECESSARY for SIMD processing resources to be used efficiently
  - Coherent execution IS NOT NECESSARY for efficient parallelization across different cores, since each core has the capability to fetch/decode a different instructions from their thread's instruction stream

- **"Divergent" execution**
  - A lack of instruction stream coherence in a program

SIMD execution: modern CPU examples

- Intel AVX2 instructions: 256 bit operations: 8x32 bits or 4x64 bits (8-wide float vectors)

- Intel AVX512 instruction: 512 bit operations: 16x32 bits…

- ARM Neon instructions: 128 bit operations: 4x32 bits…

- Instructions are generated by the compiler

  - Parallelism explicitly requested by programmer using intrinsics

  - Parallelism conveyed using parallel language semantics (e.g., `forall` example)

  - Parallelism inferred by dependency analysis of loops by "auto-vectorizing" compiler

- Terminology: "explicit SIMD": SIMD parallelization is performed at compile time

  - Can inspect program binary and see SIMD instructions (`vstoreps`, `vmulps`, etc.)

SIMD execution on many modern GPUs

TL;DR — see Kayvon's supplemental "going farther" video

- **"Implicit SIMD"**

  - **Compiler generates a binary with scalar instructions**

  - **But N instances of the program are always run together on the processor**

  - **Hardware (not compiler) is responsible for simultaneously executing the same instruction from multiple program instances on different data on SIMD ALUs**

- **SIMD width of most modern GPUs ranges from 8 to 32**

  - **Divergent execution can be a big issue (poorly written code might execute at 1/32 the peak capability of the machine!)**

Summary: three different forms of parallel execution

- **Superscalar:** exploit ILP within an instruction stream. Process different instructions from the <u>same</u> instruction stream in parallel (within a core)
    - Parallelism automatically discovered by the hardware during execution

- **SIMD:** multiple ALUs controlled by same instruction (within a core)
    - Efficient for data-parallel workloads: amortize control costs over many ALUs
    - Vectorization done by compiler (explicit SIMD) or at runtime by hardware (implicit SIMD)

- **Multi-core:** use multiple processing cores
    - Provides thread-level parallelism: simultaneously execute a completely different instruction stream on each core
    - Software creates threads to expose parallelism to hardware (e.g., via threading API)
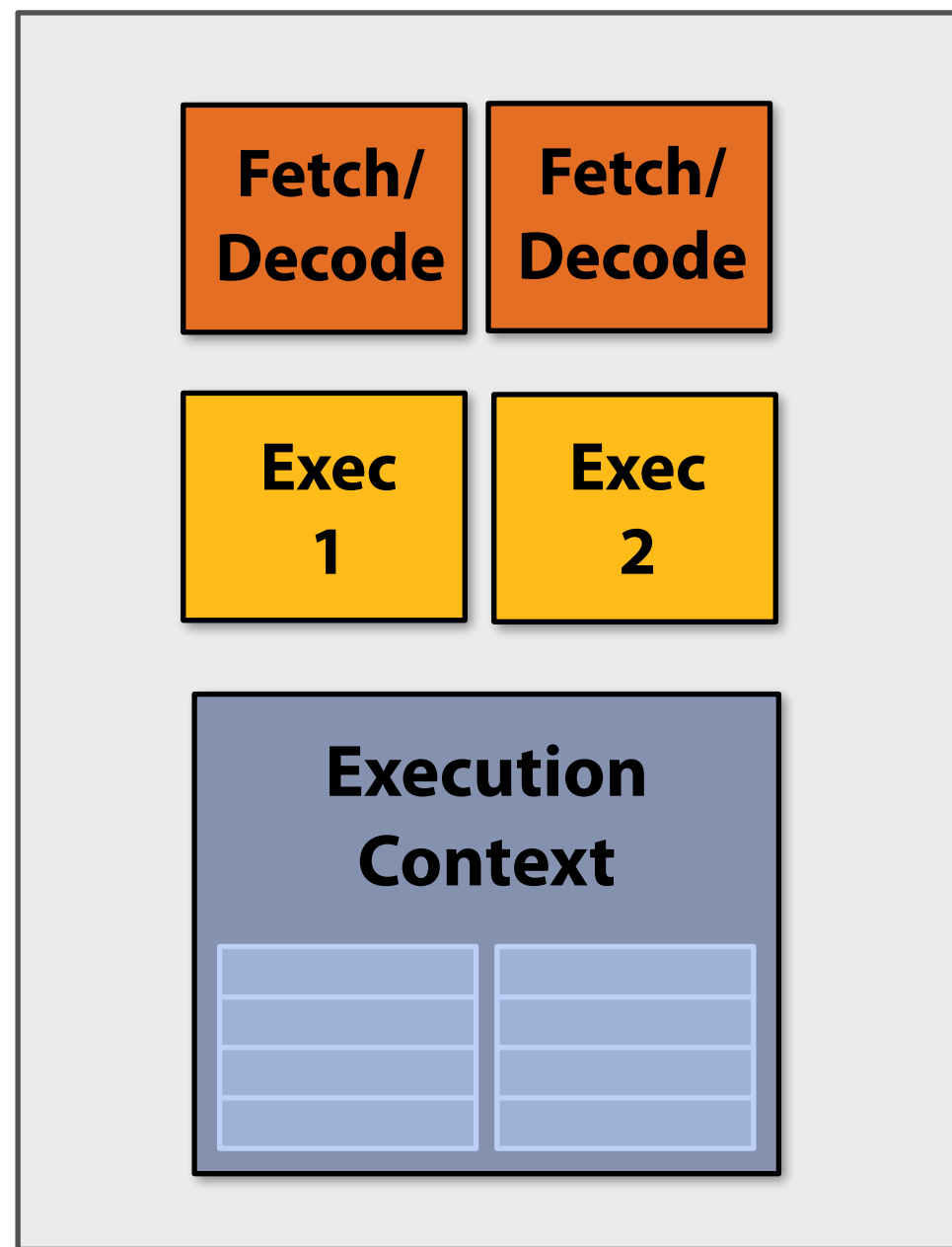
My single core, superscalar processor:
executes up to <u>two instructions</u> per clock from <u>a single instruction stream</u> (if the instructions are independent)

| Fetch/Decode | Fetch/Decode |
| Exec 1 | Exec 2 |

Execution Context
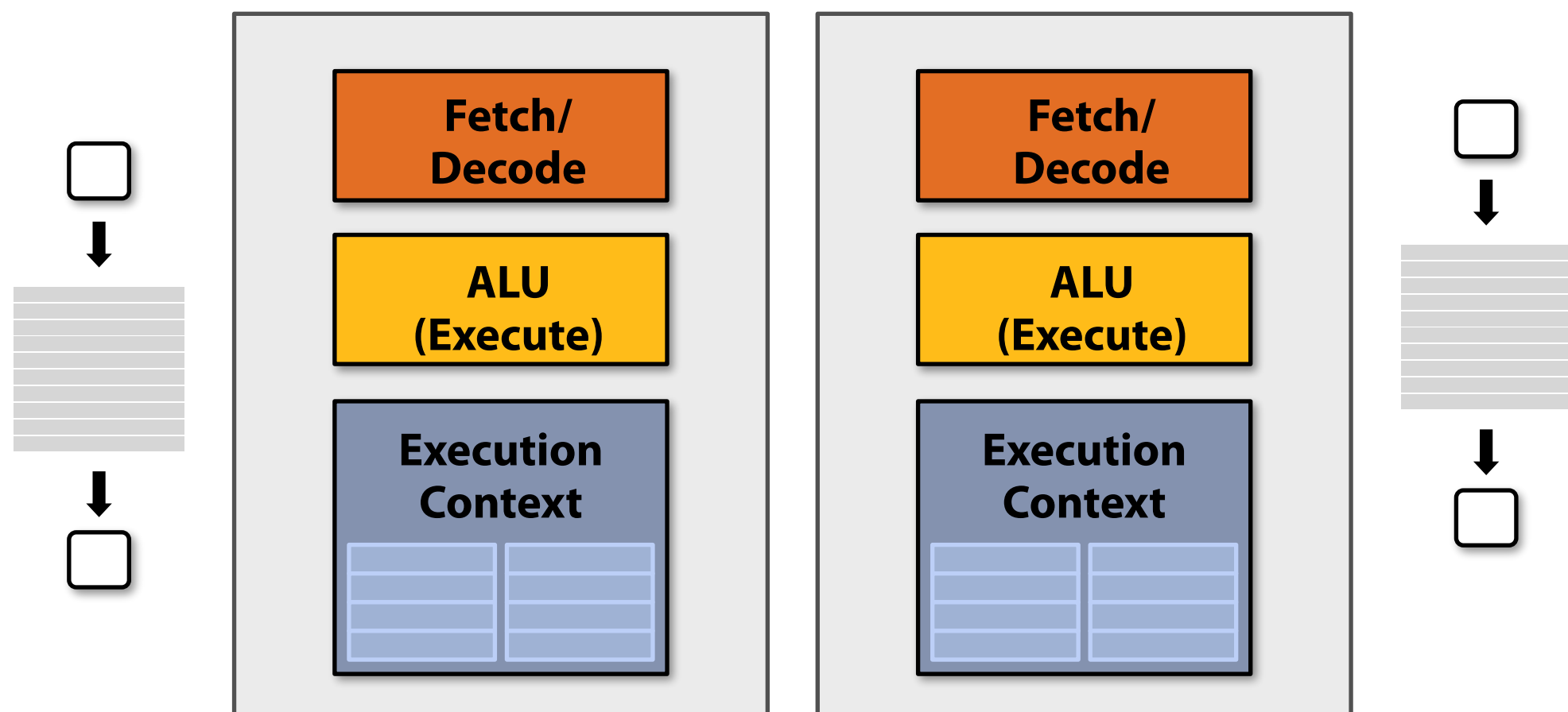
My SIMD quad-core processor:
executes one 8-wide SIMD instruction per clock from one instruction stream on <u>each</u> core.

| Fetch/Decode |
Execution Context

| Fetch/Decode |
Execution Context

| Fetch/Decode |
Execution Context

| Fetch/Decode |
Execution Context

My dual-core processor:
executes one instruction per clock from one instruction stream on <u>each</u> core.

Fetch/Decode
ALU (Execute)
Execution Context

Fetch/Decode
ALU (Execute)
Execution Context
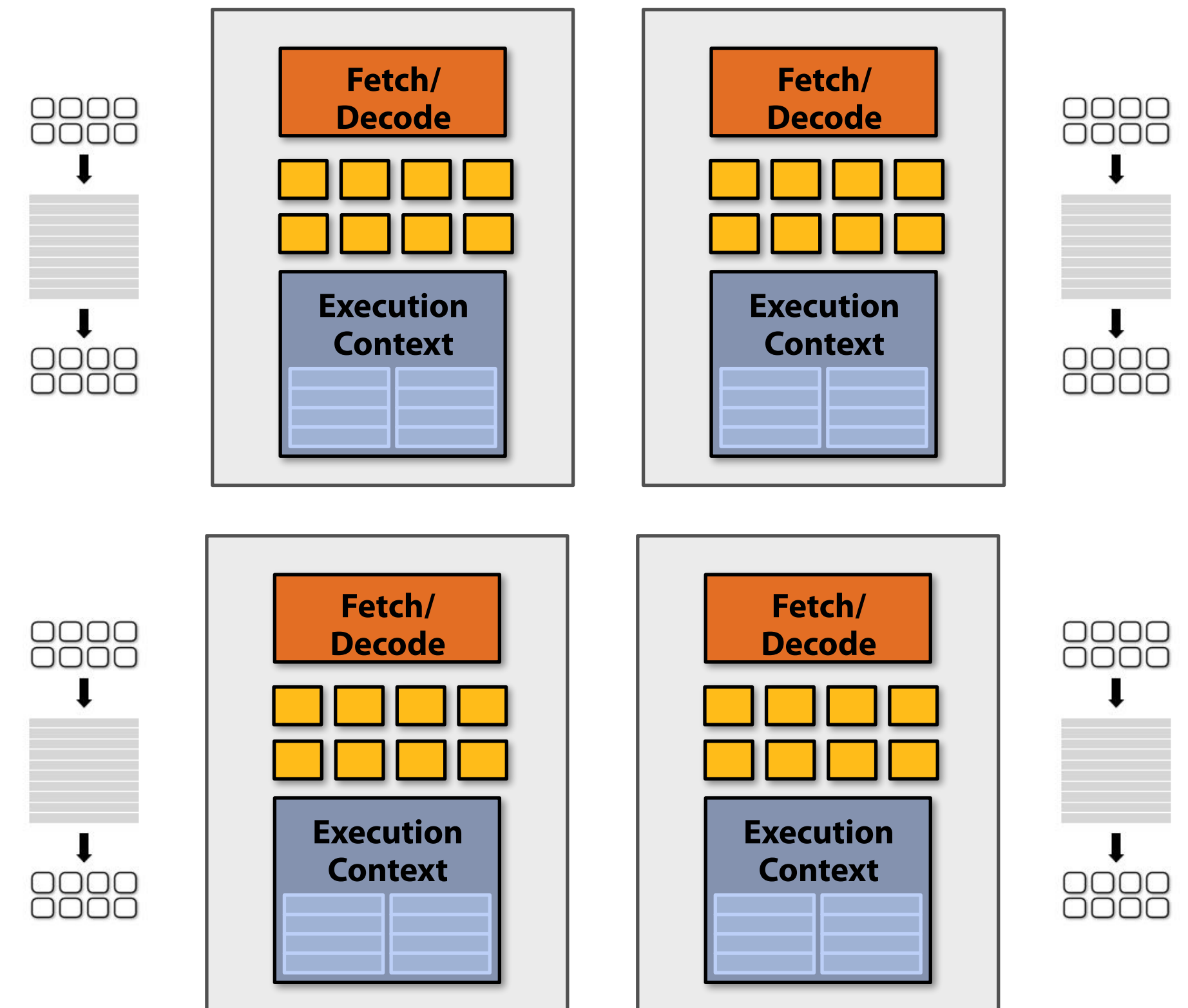
Example: four-core Intel i7-7700K CPU (Kaby Lake)
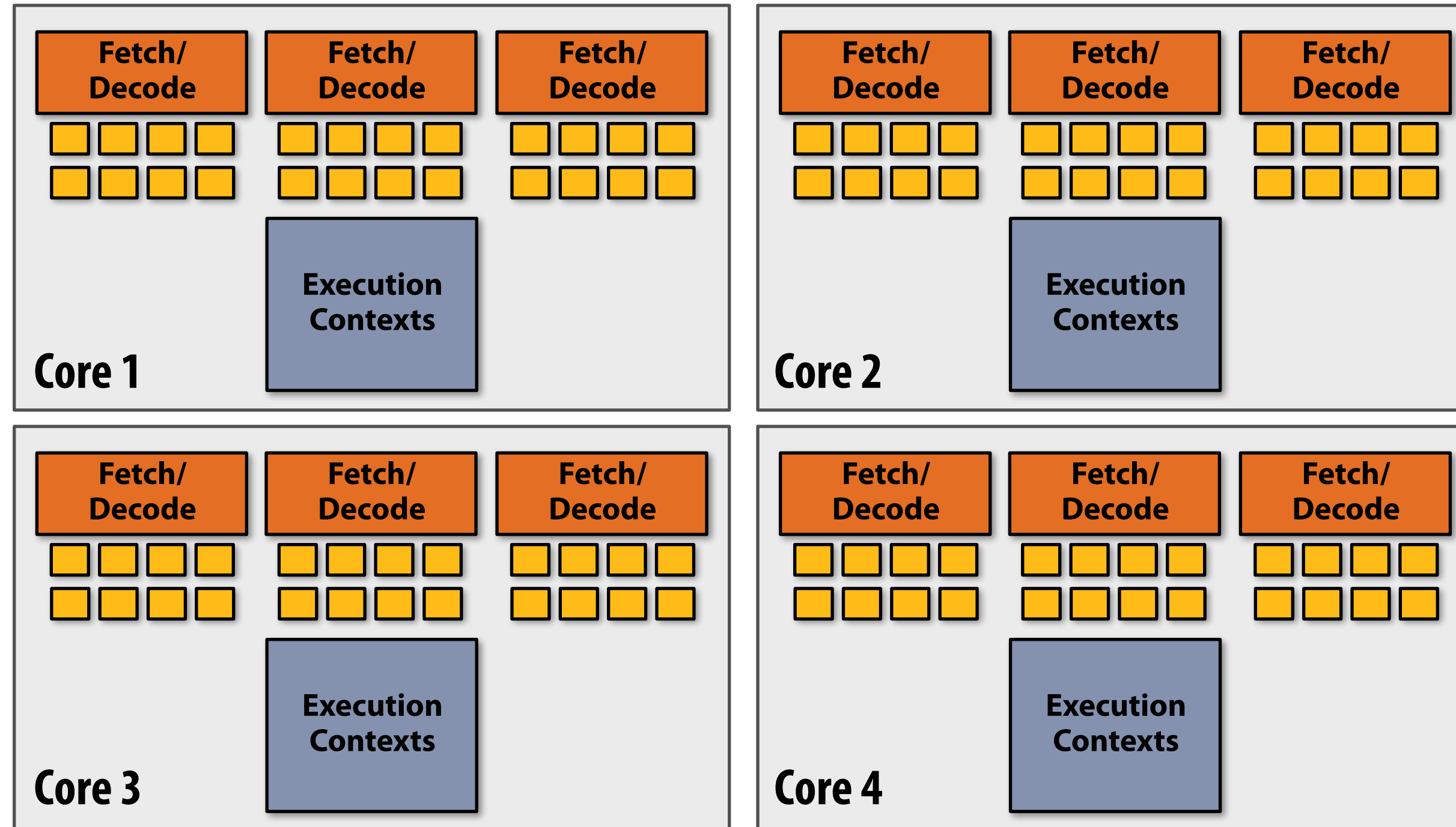
4 core processor

Three 8-wide SIMD ALUs per core

(AVX2 instructions)

4 cores x 8-wide SIMD x 3 x 4.2 GHz = 400 GFLOPs

* Showing only AVX math units, and fetch/decode unit for AVX (additional capability for integer math)

Example: NVIDIA V100 GPU

L2 Cache (6 MB)
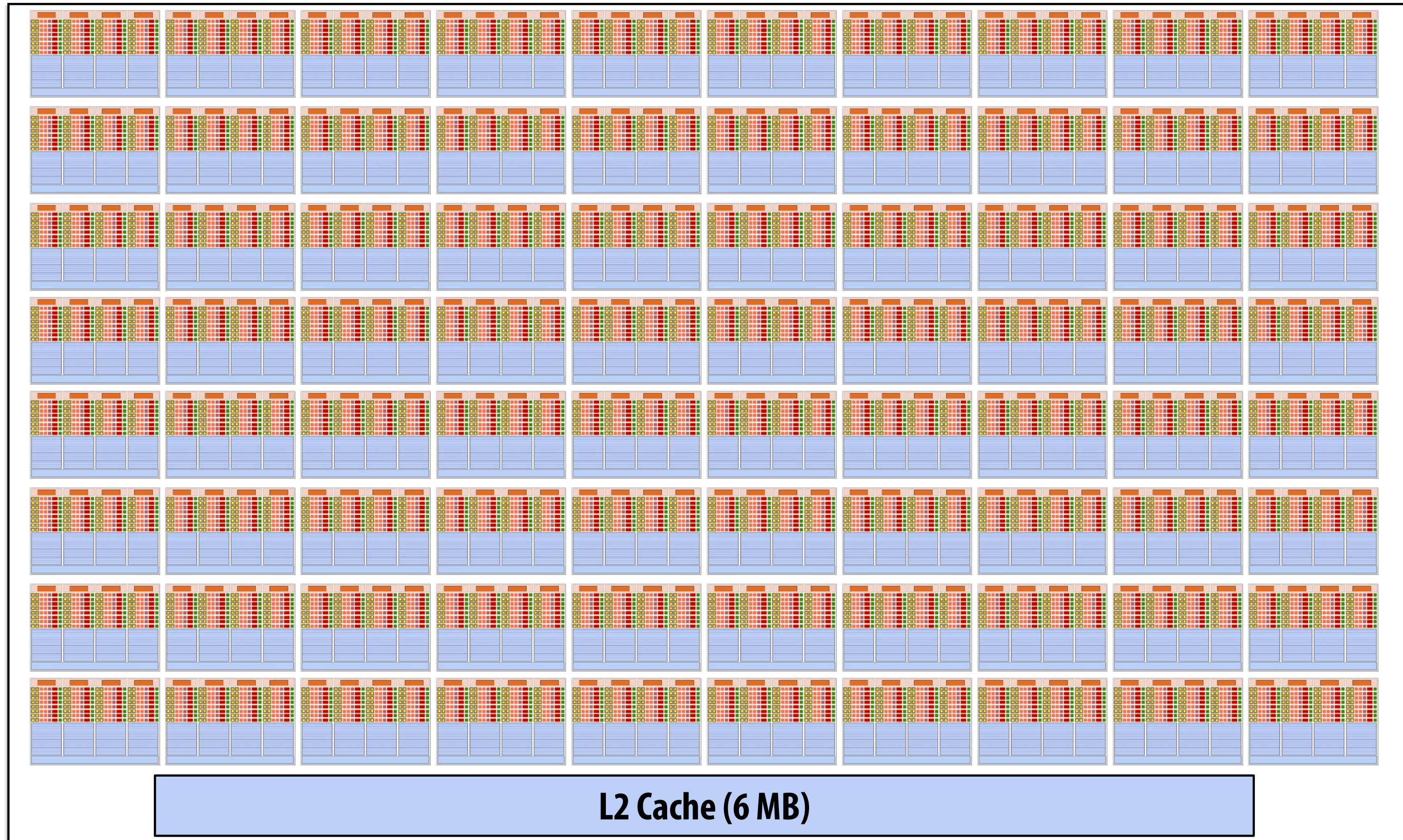
80 "SM" cores

128 SIMD ALUs per "SM" (@1.6 GHz) = 16 TFLOPs (~250 Watts)

Part 2: accessing memory
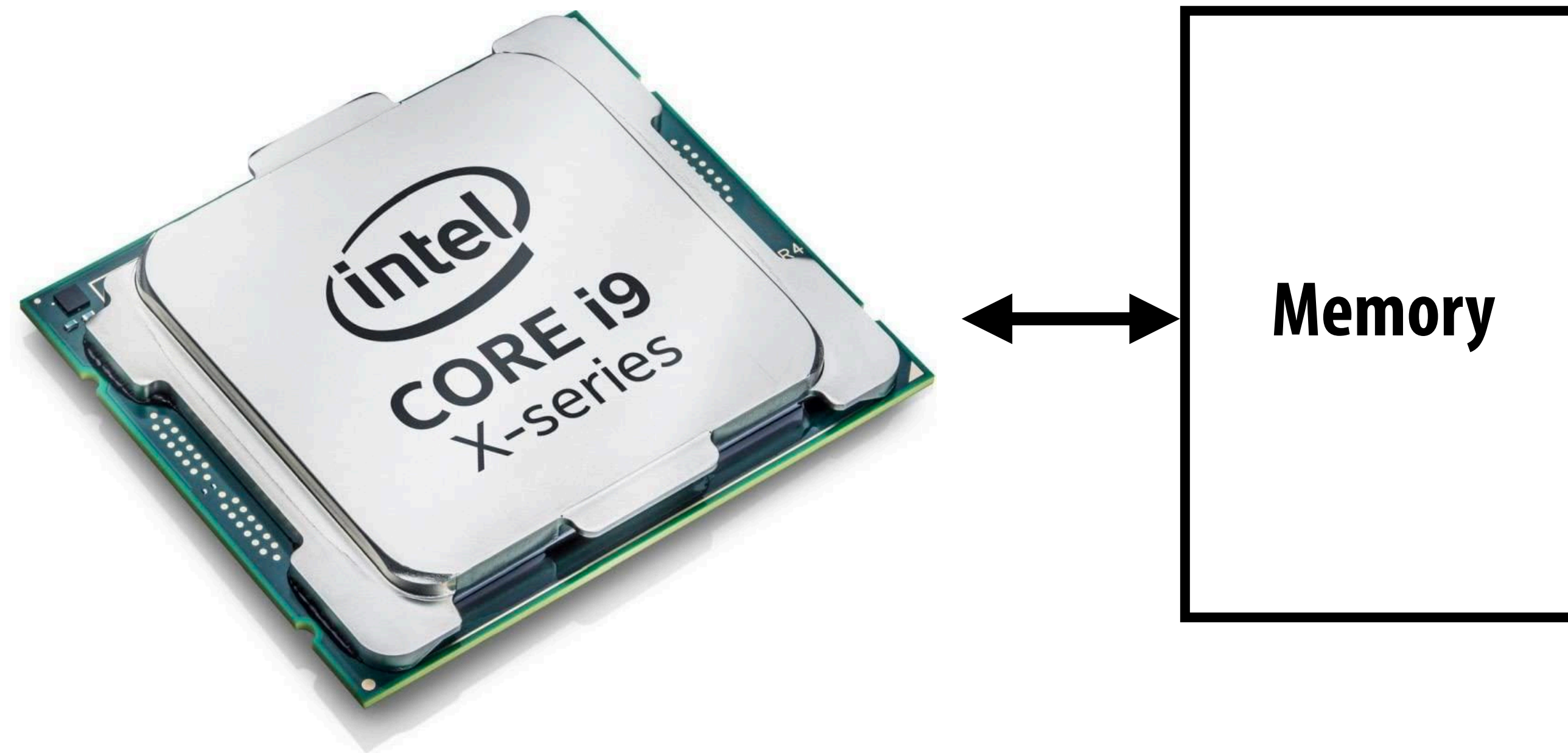
Memory

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! *

Core 1

L1 cache (32 KB)

L2 cache (256 KB)

Core N

L1 cache (32 KB)

L2 cache (256 KB)

L3 cache (8 MB)

38 GB/sec

Memory
DDR4 DRAM

(Gigabytes)
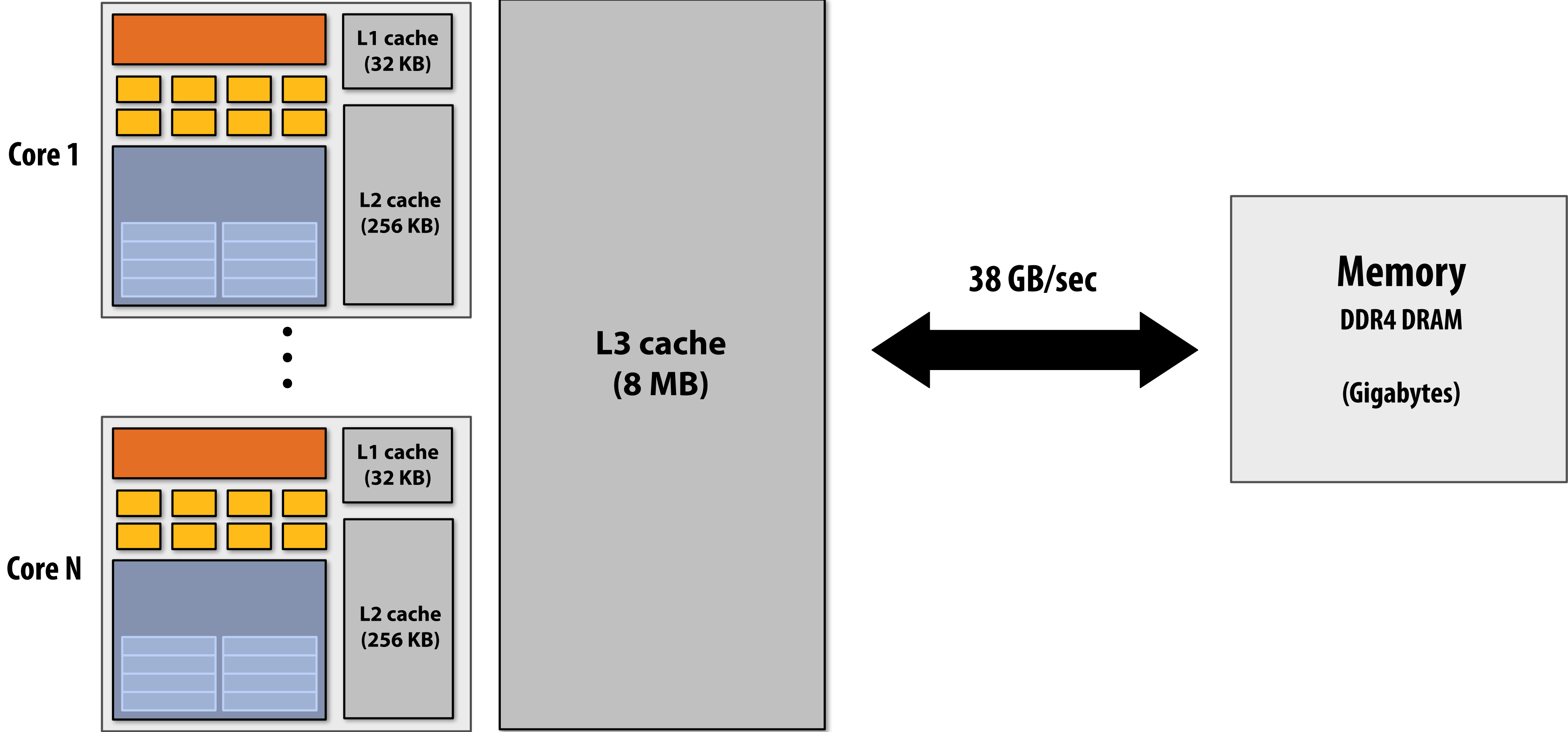
* Caches also provide high bandwidth data transfer

Recall: [very] long latency of data access

(Kaby Lake CPU)

Latency (number of cycles at 4 GHz)

Data in L1 cache	4
Data in L2 cache	12
Data in L3 cache	38
Data in DRAM (best case)	~248
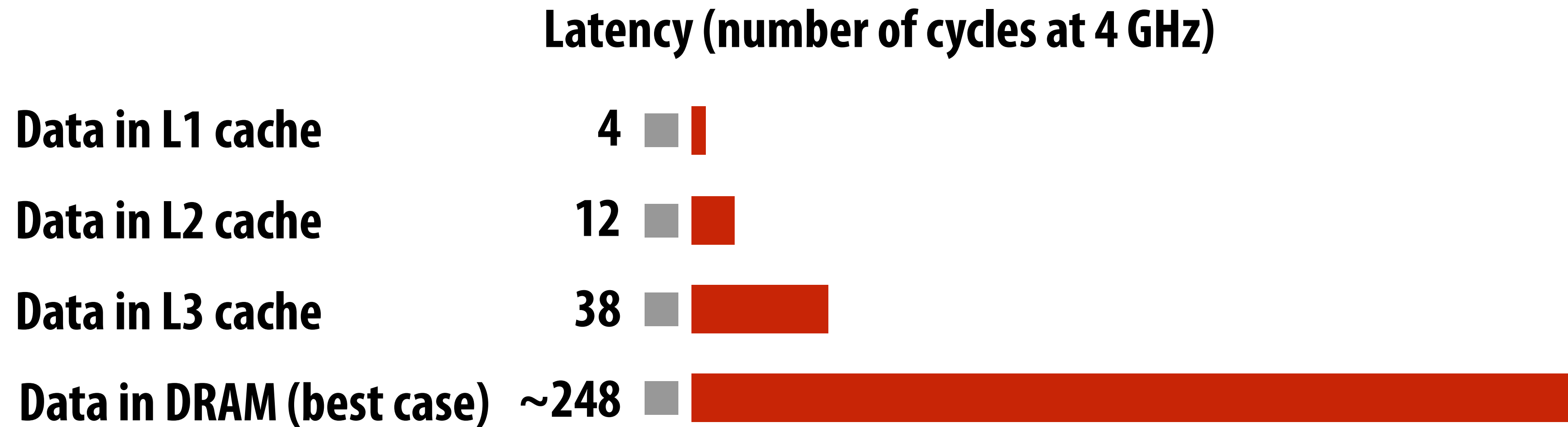
Recall this access pattern

Program reads entire array of 16 bytes, then reads entire array again in the future.

Assume:

Total cache capacity = 8 bytes

Cache has 4-byte cache lines
(So 2 lines fit in cache)

Least recently used (LRU)
replacement policy

Discussion Questions:

Why is there no "hit" on second read of address 0x0?

What about second read of address 0x4?

Would your answer change if the cache had a capacity of 4 lines?

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

Address accessed	Cache action
0x0	"cold miss", load 0x0
0x1	hit
0x2	hit
0x3	hit
0x4	"cold miss", load 0x4
0x5	hit
0x6	hit
0x7	hit
0x8	"cold miss", load 0x8 (evict 0x0)
0x9	hit
0xA	hit
0xB	hit
0xC	"cold miss", load 0xC (evict 0x4)
0xD	hit
0xE	hit
0xF	hit
0x0	"capacity miss", load 0x0 (evict 0x8)

Data prefetching reduces stalls (<u>hides</u> latency)

- **Many modern CPUs have logic for guessing what data will be accessed in the future and "pre-fetching" this data into caches**

  - **Dynamically analyze program's memory access patterns to make predictions**

- **Prefetching reduces stalls since data is resident in cache when accessed**

```
predict value of r2, initiate load
predict value of r3, initiate load
...
...
...          data arrives in cache
...
...          data arrives in cache
...
ld r0 mem[r2]    These loads are cache hits
ld r1 mem[r3]
add r0, r0, r1
```
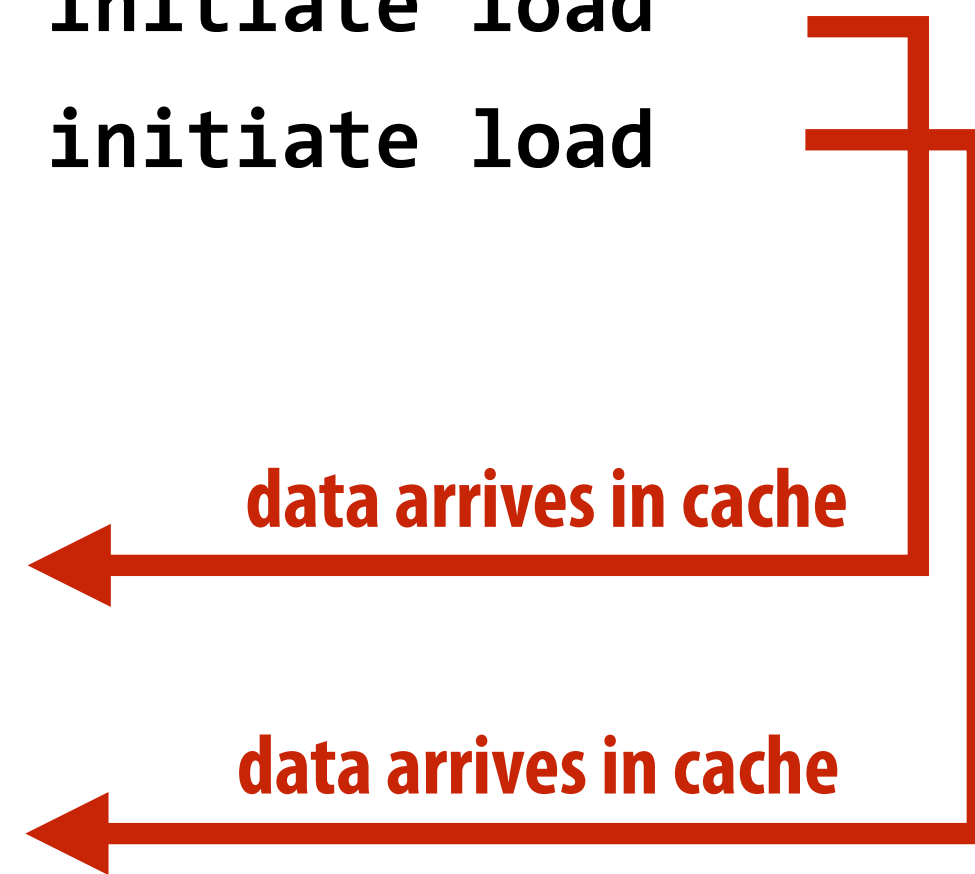
Note: Prefetching can also reduce performance if the guess is wrong (consumes bandwidth, pollutes caches)

But what if data hasn't been read recently, so does not reside in cache?

And the next piece of data to read is not easily predictable?

```
int x = some_function();
int y = A[x];
```

Consider doing your laundry…

Or cooking a meal…
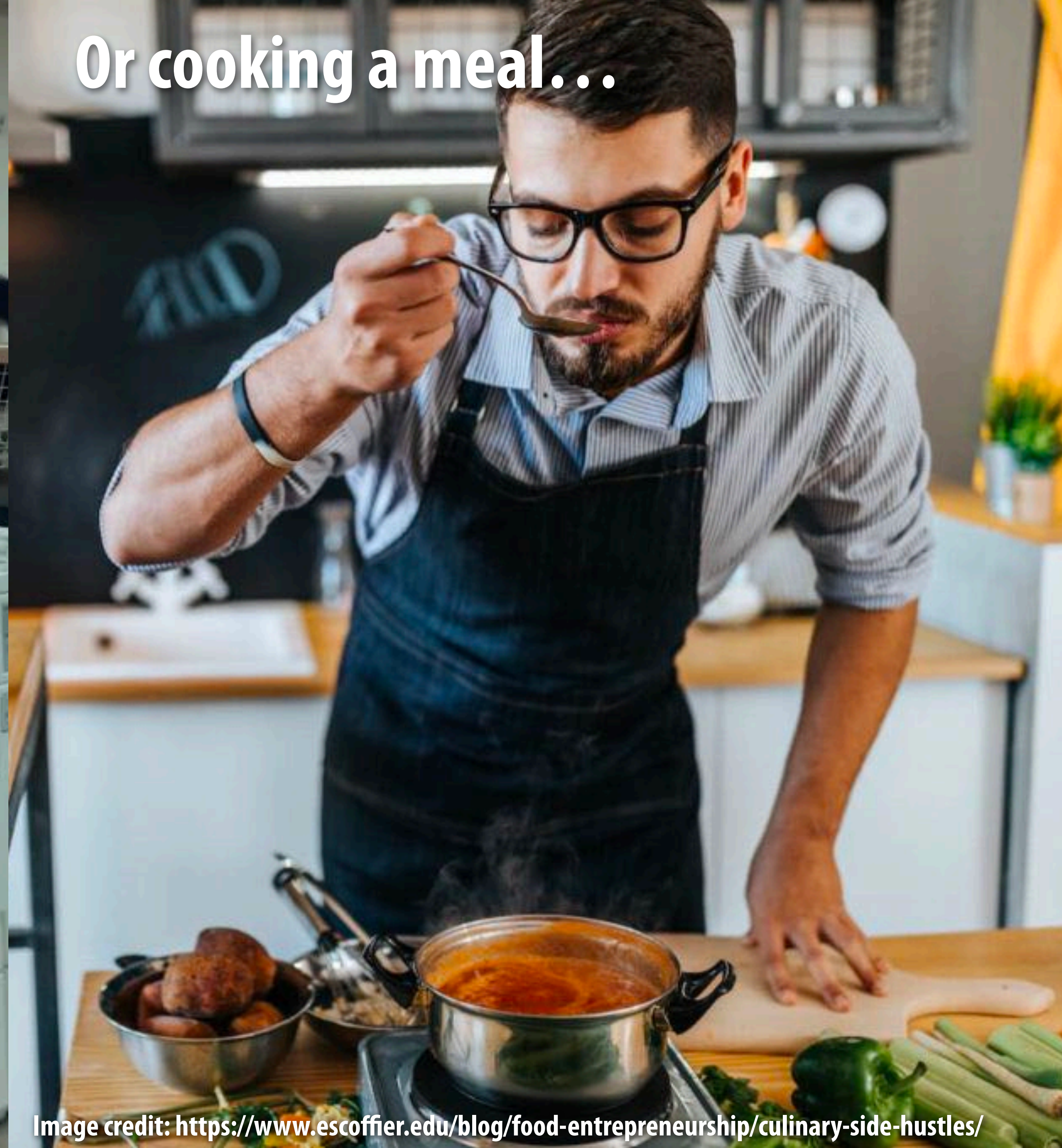
Multi-threading reduces stalls

- **Idea #3: <u>interleave</u> processing of multiple threads on the same core to hide stalls**
  - **If you can't make progress on the current thread… work on another one**

Hiding stalls with multi-threading

Thread 1
Elements 0 … 7

Time

1 Core (1 thread)

Fetch/ Decode

ALU 0 **ALU 1** **ALU 2** **ALU 3**

ALU 4 **ALU 5** **ALU 6** **ALU 7**

Exec Ctx

Hiding stalls with multi-threading

Time

| Thread 1
Elements 0 ... 7 | Thread 2
Elements 8 ... 15 | Thread 3
Elements 16 ... 23 | Thread 4
Elements 24 ... 31 |

1 Core (4 hardware threads)

Hiding stalls with multi-threading

Time

Thread 1
Elements 0 … 7

Thread 2
Elements 8 … 15

Thread 3
Elements 16 … 23

Thread 4
Elements 24 … 31

❶ ❷ ❸ ❹

Stall

Runnable

1 Core (4 hardware threads)

Fetch/
Decode

ALU 0 | ALU 1 | ALU 2 | ALU 3

ALU 4 | ALU 5 | ALU 6 | ALU 7

❶ ❷
❸ ❹

Hiding stalls with multi-threading

Thread 1
Elements 0 … 7

Thread 2
Elements 8 … 15

Thread 3
Elements 16 … 23

Thread 4
Elements 24 … 31

Time

① **Stall**

Runnable

② **Stall**

Runnable

Done!

③ **Stall**

Runnable

④ **Stall**

Runnable

Done!

1 Core (4 hardware threads)

Fetch/
Decode

ALU 0 ALU 1 ALU 2 ALU 3
ALU 4 ALU 5 ALU 6 ALU 7

① ② ③ ④

Throughput computing: a trade-off

Thread 1
Elements 0 … 7

Thread 2
Elements 8 … 15

Thread 3
Elements 16 … 23

Thread 4
Elements 24 … 31

Time

Stall

Runnable

Done!
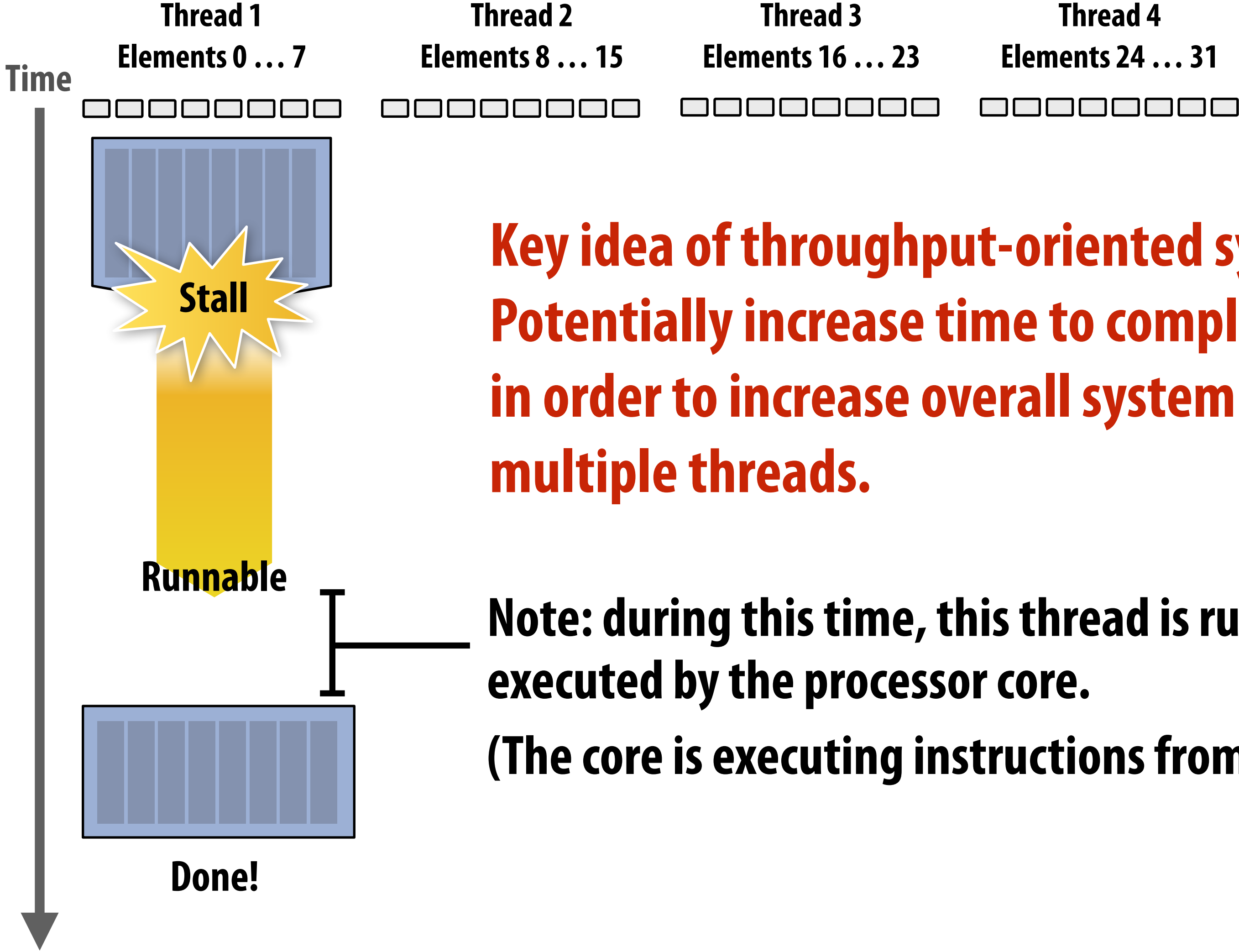
Key idea of throughput-oriented systems:
Potentially increase time to complete work by any one thread,
in order to increase overall system throughput when running
multiple threads.

Note: during this time, this thread is runnable, but it is not being executed by the processor core.

(The core is executing instructions from another thread.)

No free lunch: storing execution contexts

Consider on-chip storage of execution contexts as a finite resource
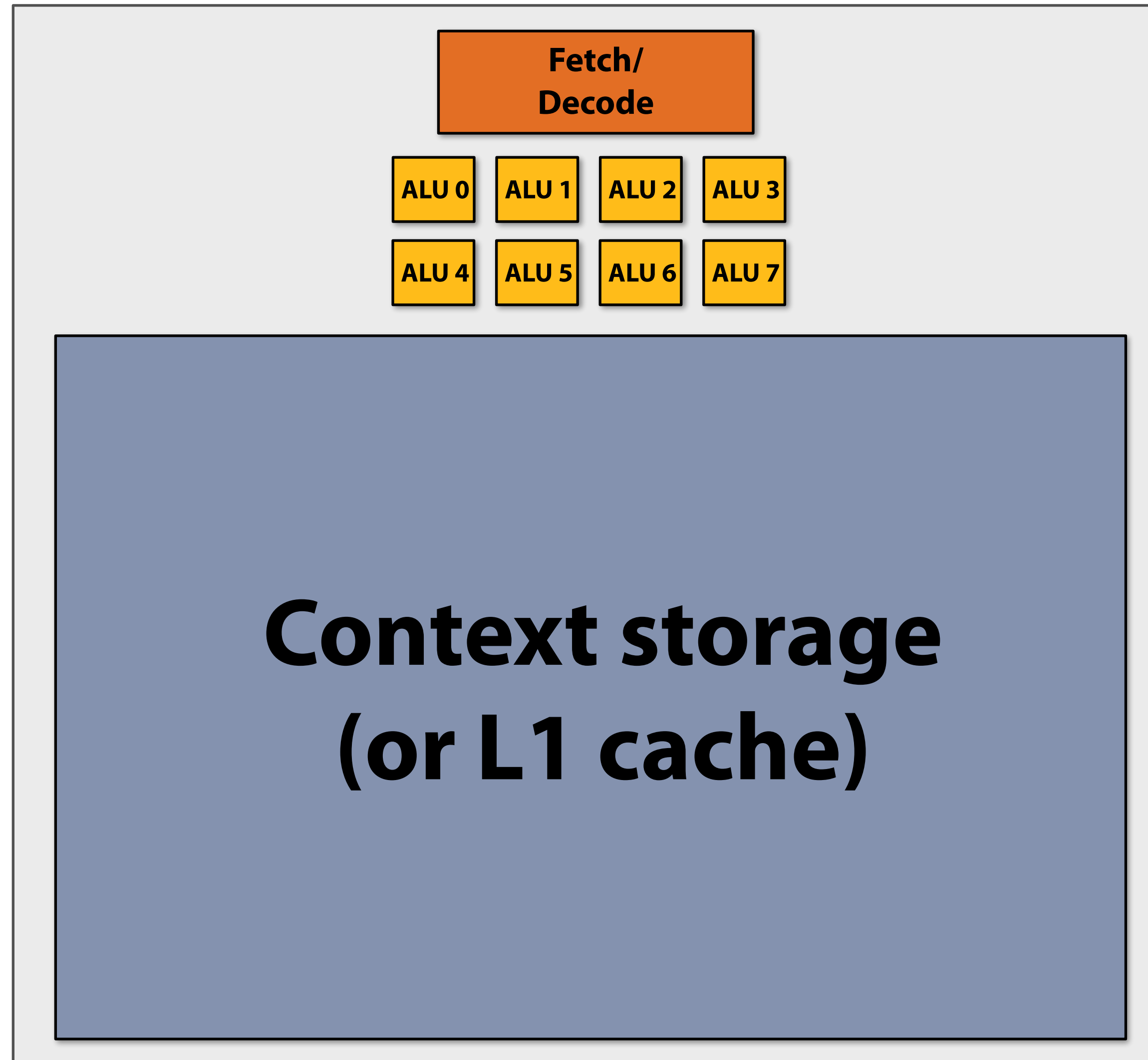
Many small contexts (high latency hiding ability)

16 hardware threads: storage for small working set per thread
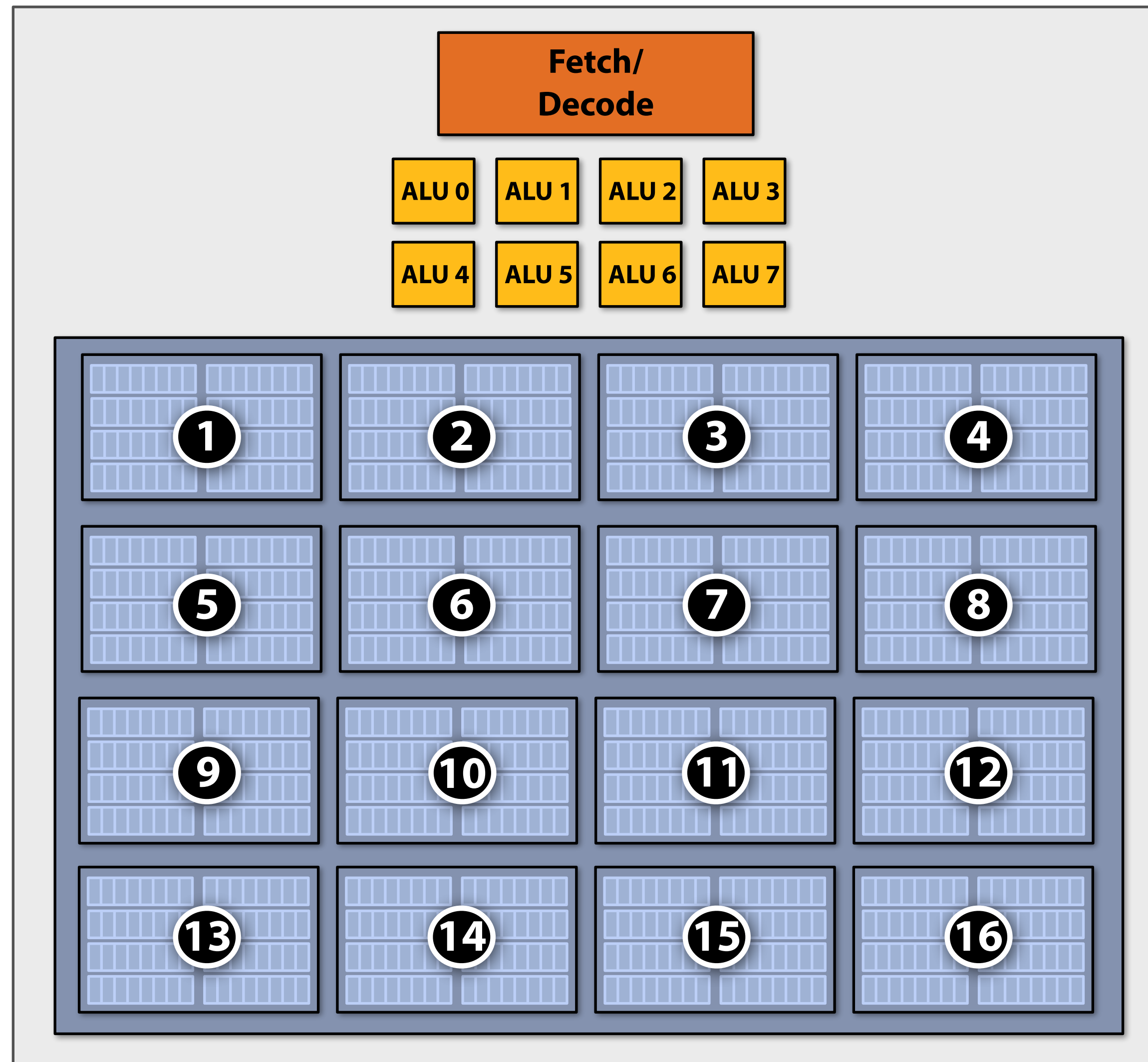
Four large contexts (low latency hiding ability)

4 hardware threads: storage for large working set per thread
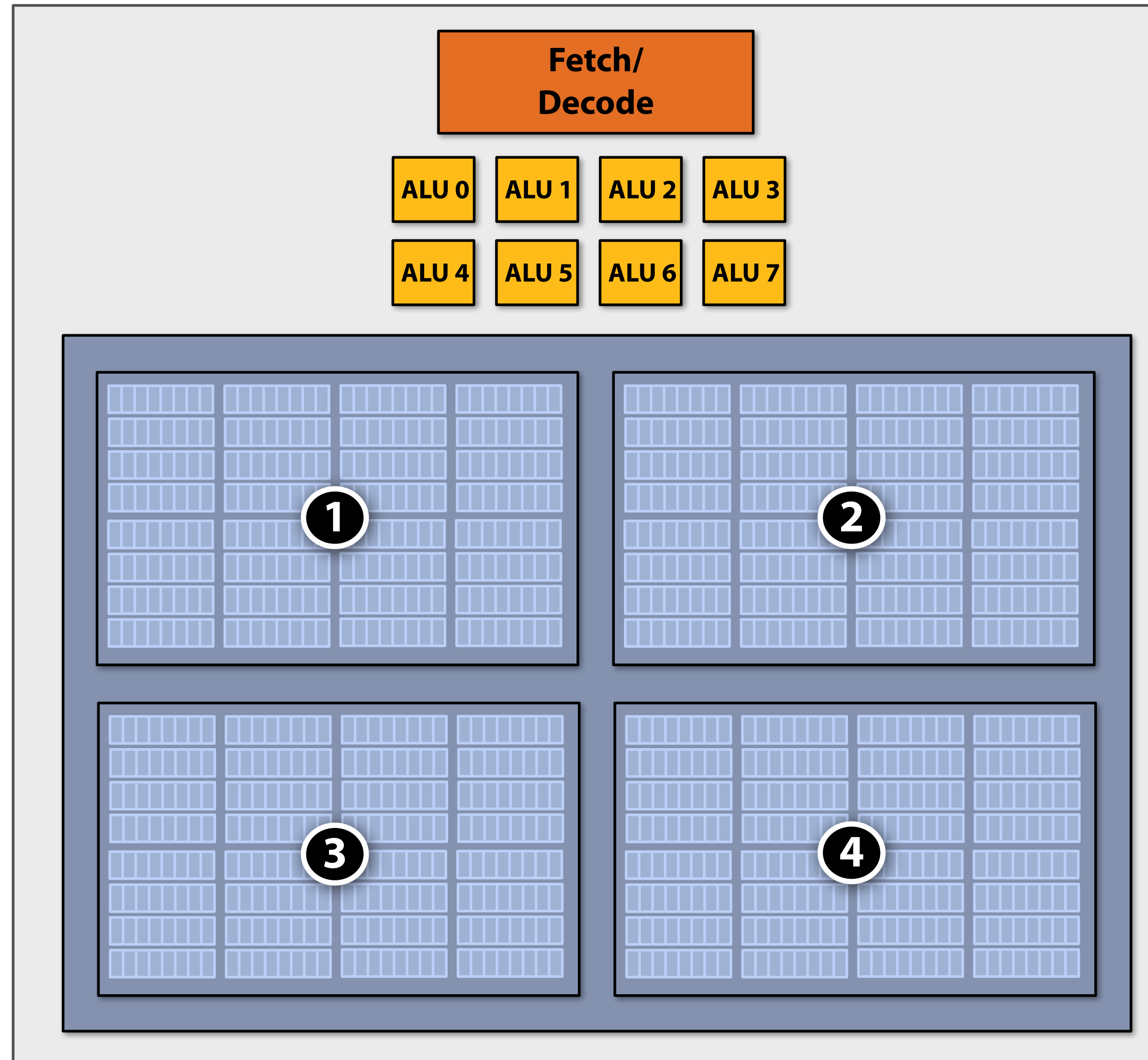
Exercise: consider a simple two-threaded core
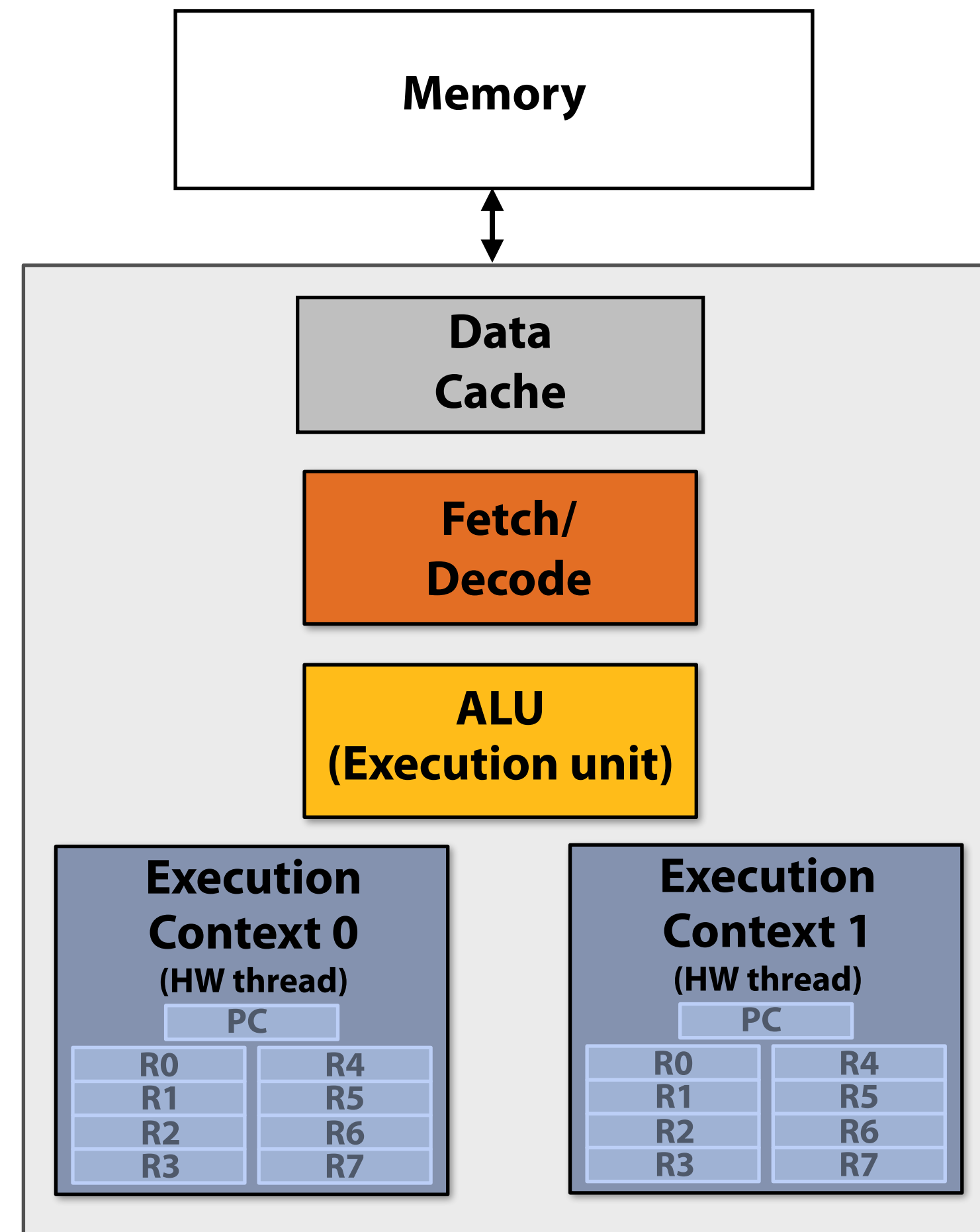
Single core processor, multi-threaded core (2 threads).
Can run one scalar instruction per clock from
one of the hardware threads

What is the utilization of the core? (one thread)
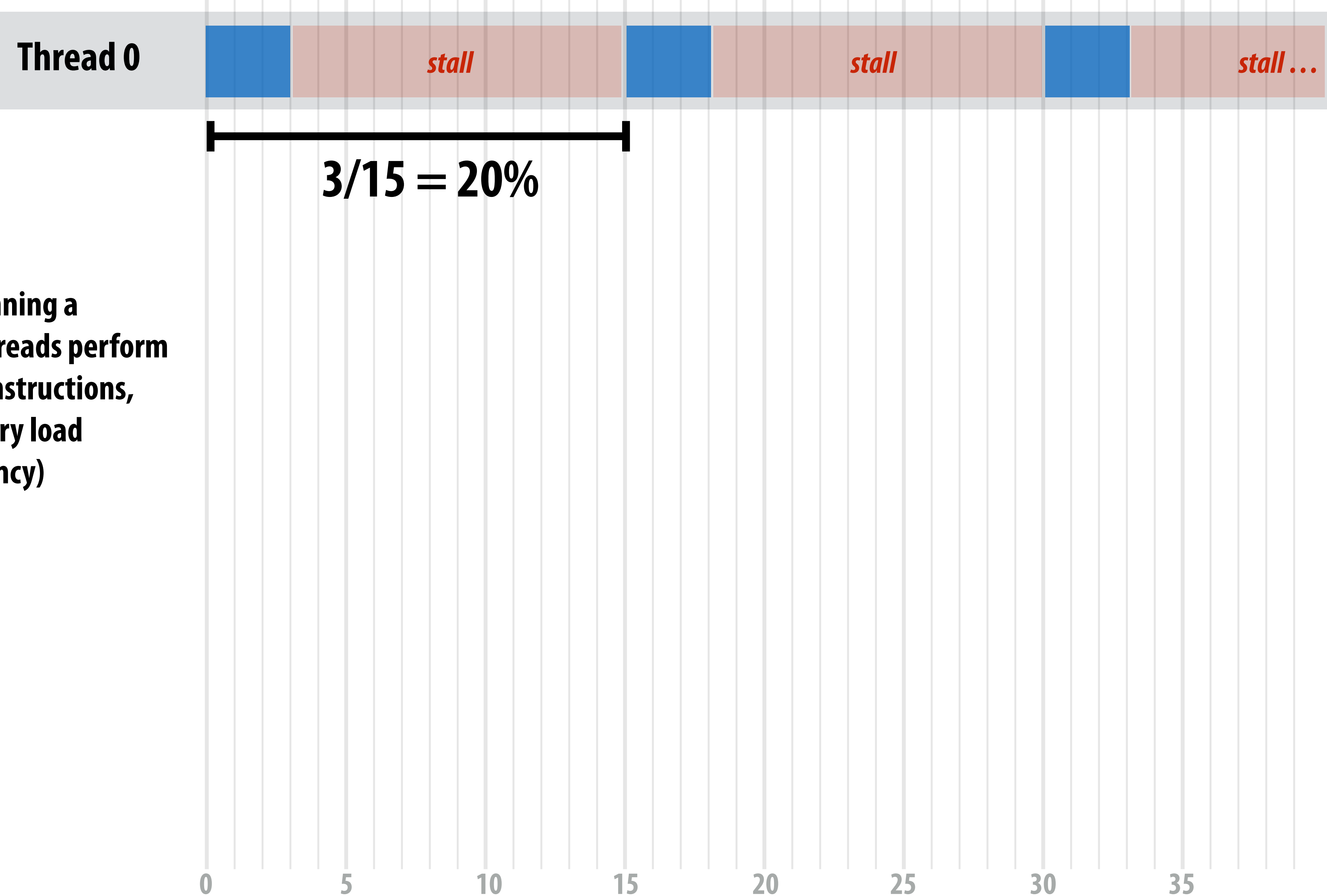
Thread 0

$3/15 = 20\%$

Assume we are running a
program where threads perform
three arithmetic instructions,
followed by memory load
(with 12 cycle latency)

0   5   10   15   20   25   30   35

What is the utilization of the core? (two threads)
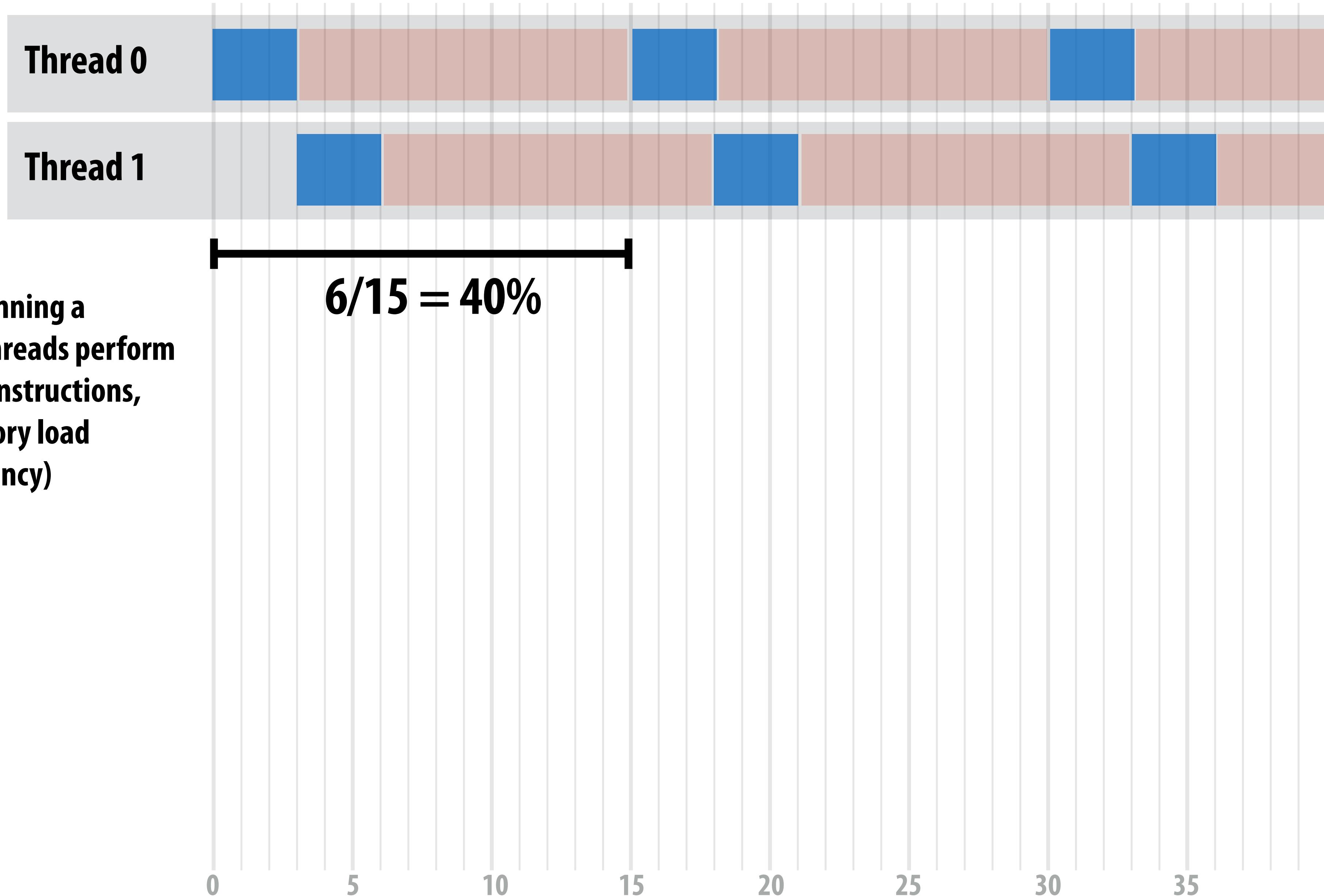
Thread 0	
Thread 1	

$$6/15 = 40\%$$

Assume we are running a
program where threads perform
three arithmetic instructions,
followed by memory load
(with 12 cycle latency)

0   5   10   15   20   25   30   35

How many threads are needed to achieve 100% utilization?
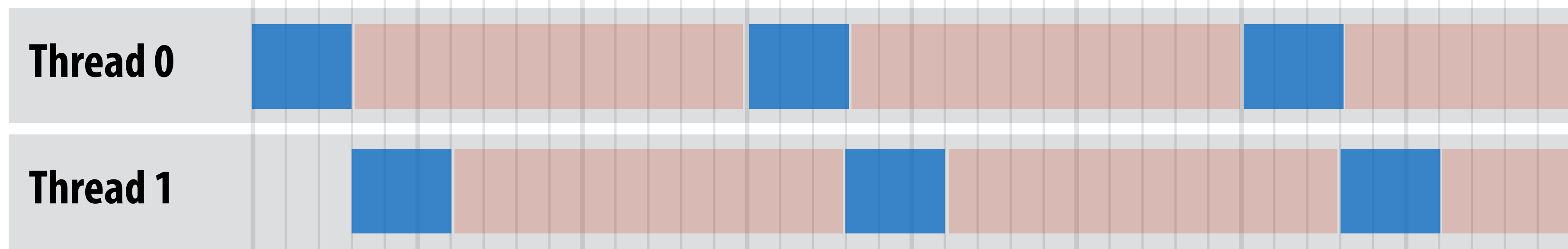
Assume we are running a
program where threads perform
three arithmetic instructions,
followed by memory load
(with 12 cycle latency)

Five threads needed to obtain 100% utilization

Thread 0

Thread 1

Thread 2
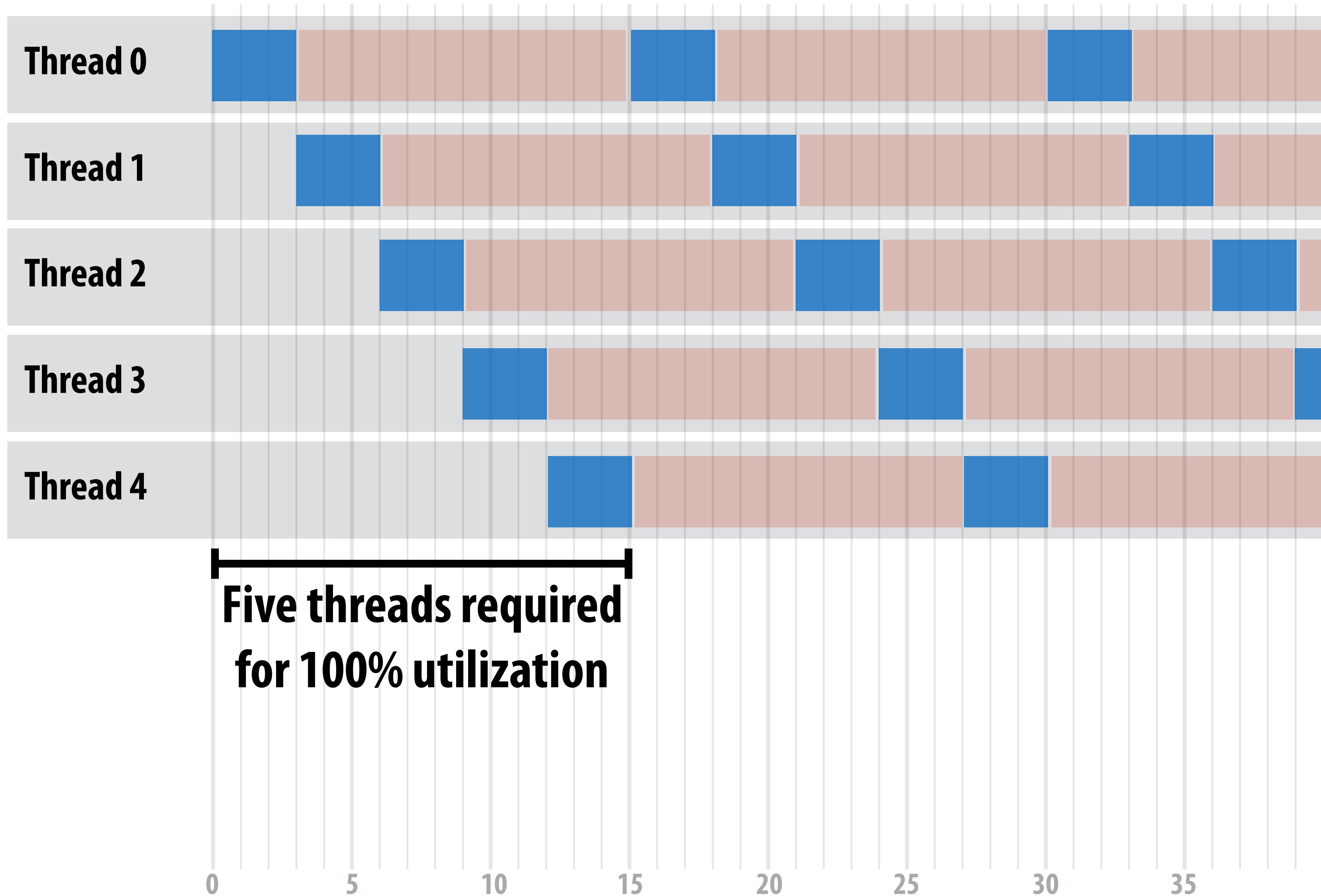
Thread 3

Thread 4

Five threads required for 100% utilization

0   5   10   15   20   25   30   35

Additional threads yield no benefit (already 100% utilization)

Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7

0   15   20   25   30   35

Still 100%

How many threads are needed to achieve 100% utilization?

Threads now perform *six arithmetic instructions*, followed by memory load (with 12 cycle latency)
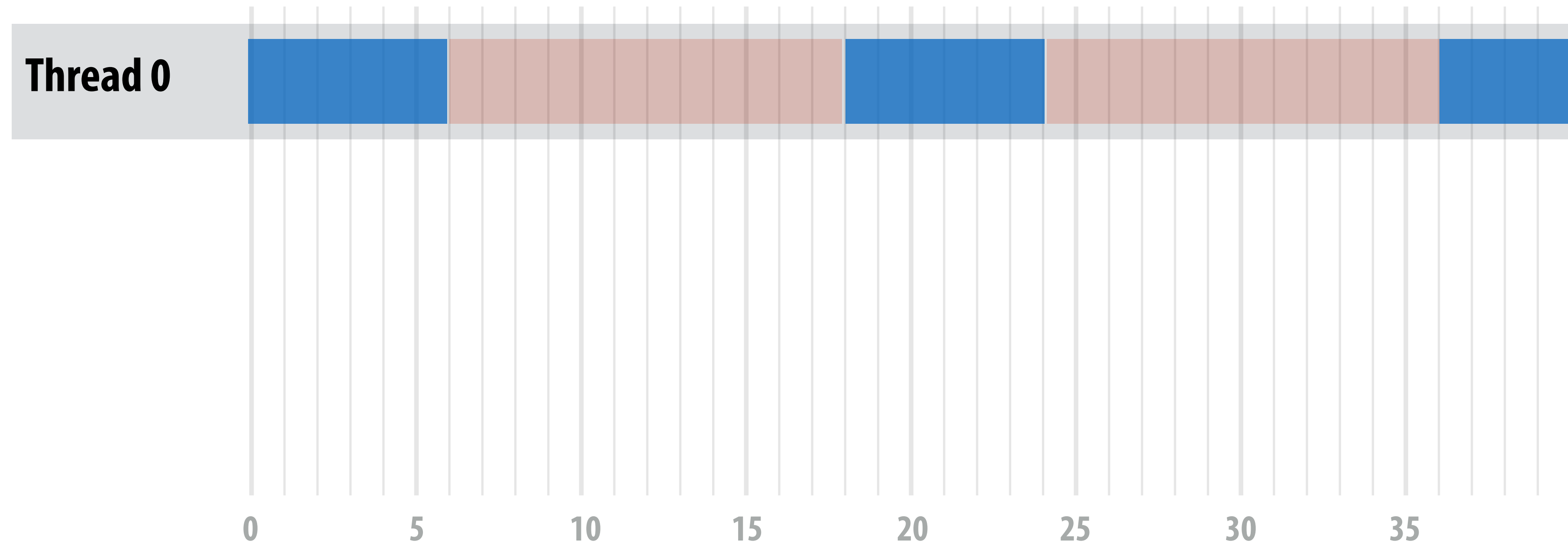
Now only three threads needed for 100% utilization?

Threads now perform *six arithmetic instructions*, followed by memory load (with 12 cycle latency)
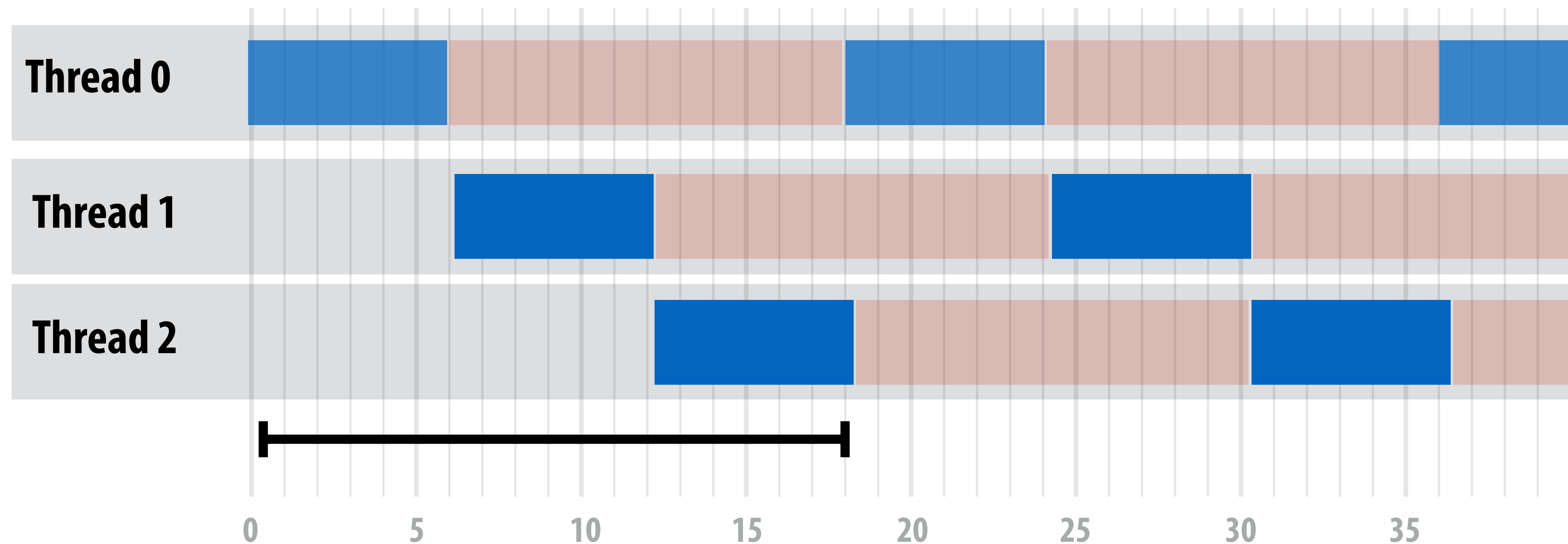
100% utilization using
only three threads

How does a higher ratio of math instructions to memory latency affect the number of threads needed for latency hiding?

Takeaway (point 1):

A processor with multiple hardware threads has the ability to *avoid stalls* by performing instructions from other threads when one thread must wait for a long latency operation to complete.

Note: the latency of the memory operation is not changed by multi-threading, it just no longer causes reduced processor utilization.

Takeaway (point 2):

A multi-threaded processor hides memory latency by performing arithmetic from other threads.

Programs that feature more arithmetic per memory access need fewer threads to hide memory stalls.

Hardware-supported multi-threading

- **Core manages execution contexts for multiple threads**

  - Core still has the same number of ALU resources: multi-threading only helps use them more efficiently in the face of high-latency operations like memory access
  - Processor makes decision about which thread to run each clock

- **Interleaved multi-threading (a.k.a. temporal multi-threading)**

  - What I described on the previous slides: each clock, the core chooses a thread, and runs an instruction from the thread on the core's ALUs

- **Simultaneous multi-threading (SMT)**

  - Each clock, core chooses instructions from multiple threads to run on ALUs
  - Example: Intel Hyper-threading (2 threads per core)
  - See "going further videos" provided online

Kayvon's fictitious multi-core chip

16 cores

8 SIMD ALUs per core
(128 total)

4 threads per core

16 simultaneous
instruction streams

64 total concurrent
instruction streams

512 independent pieces of
work are needed to run chip
with maximal latency
hiding ability
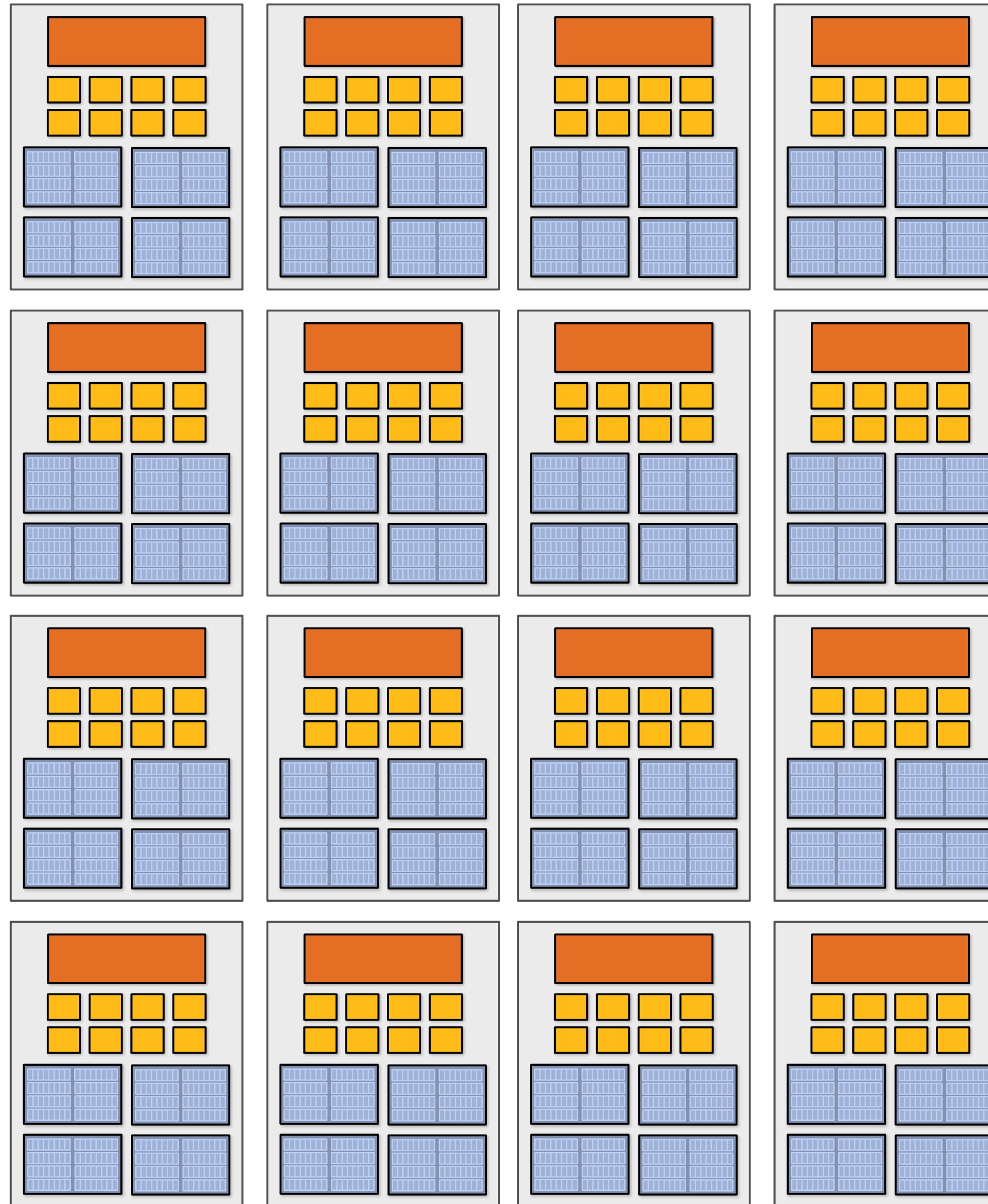
Example: Intel Skylake/Kaby Lake core
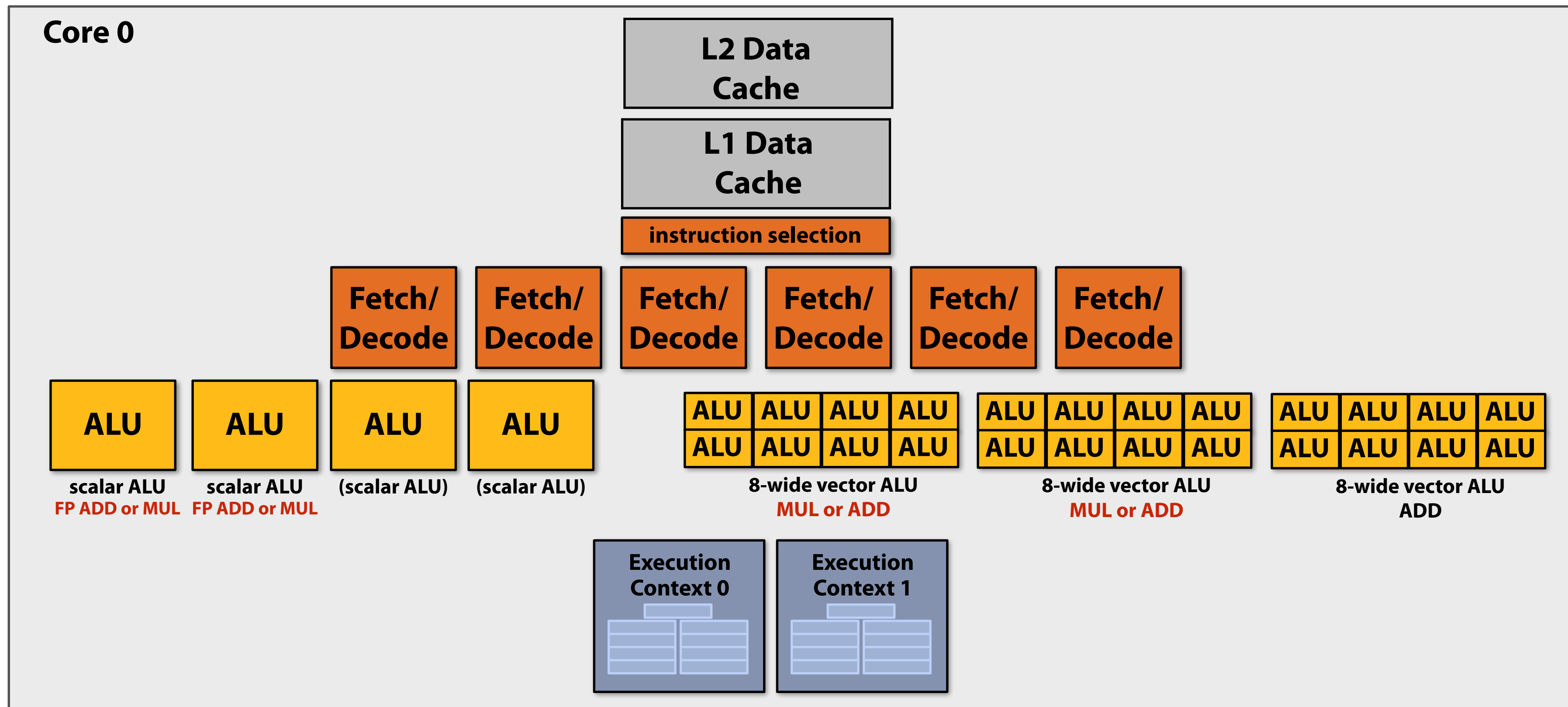
Core 0

L2 Data Cache

L1 Data Cache

instruction selection

Fetch/ Decode **Fetch/ Decode** **Fetch/ Decode** **Fetch/ Decode** **Fetch/ Decode** **Fetch/ Decode**

ALU **ALU** **ALU** **ALU**

| ALU | ALU | ALU | ALU |
| ALU | ALU | ALU | ALU |

| ALU | ALU | ALU | ALU |
| ALU | ALU | ALU | ALU |

| ALU | ALU | ALU | ALU |
| ALU | ALU | ALU | ALU |

scalar ALU
FP ADD or MUL

scalar ALU
FP ADD or MUL

(scalar ALU)

(scalar ALU)

8-wide vector ALU
MUL or ADD

8-wide vector ALU
MUL or ADD

8-wide vector ALU
ADD

Execution Context 0

Execution Context 1

**Two-way multi-threaded cores (2 threads).
Each core can run up to four independent scalar instructions
and up to three 8-wide vector instructions
(up to 2 vector mul or 3 vector add)**

NVIDIA V100

- SM = "Streaming Multi-processor"
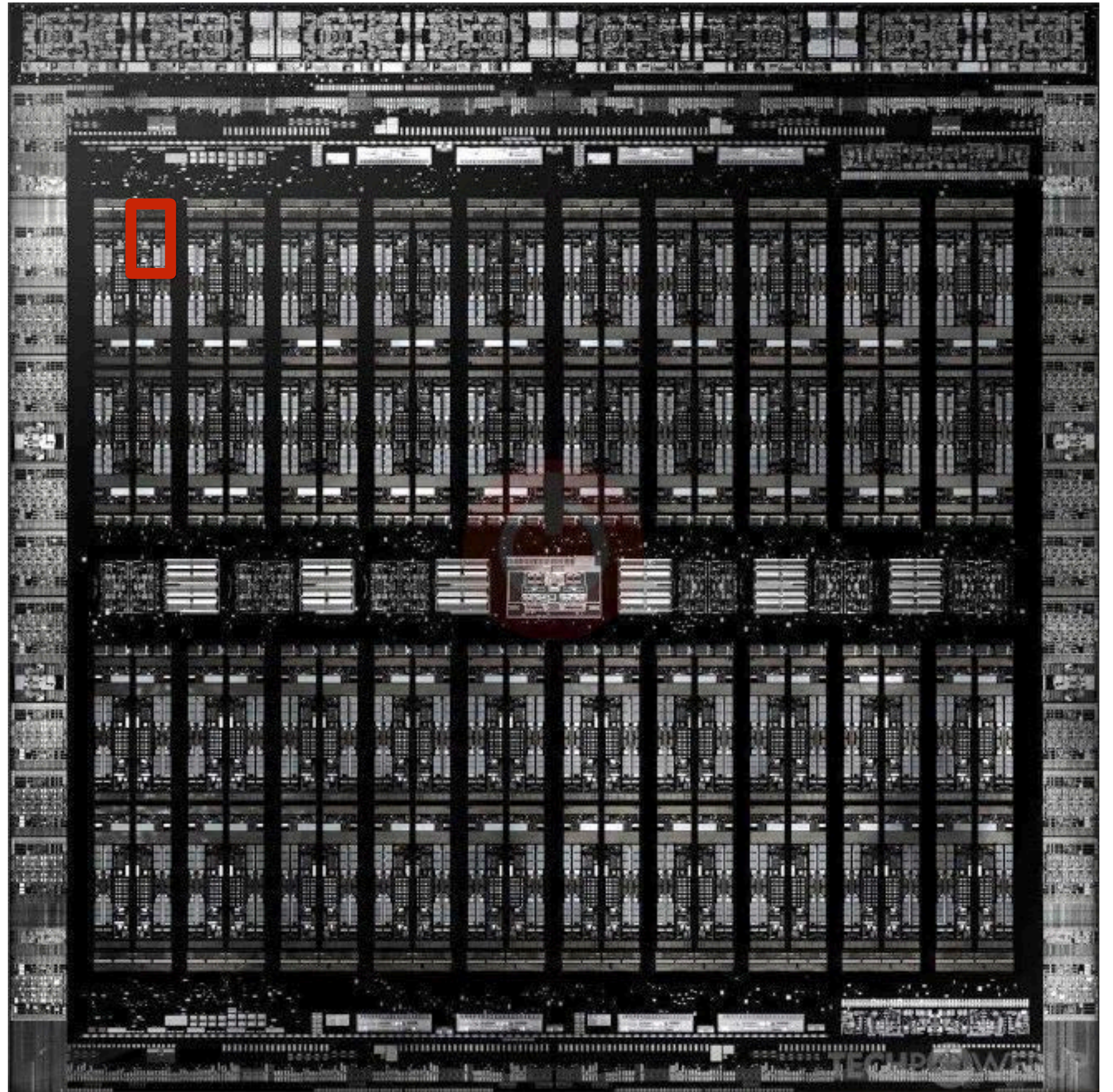
GPUs: extreme throughput-oriented processors

This is one NVIDIA V100 streaming multi-processor (SM) unit

Warp Selector	Warp Selector	Warp Selector	Warp Selector
Fetch/Decode	Fetch/Decode	Fetch/Decode	Fetch/Decode
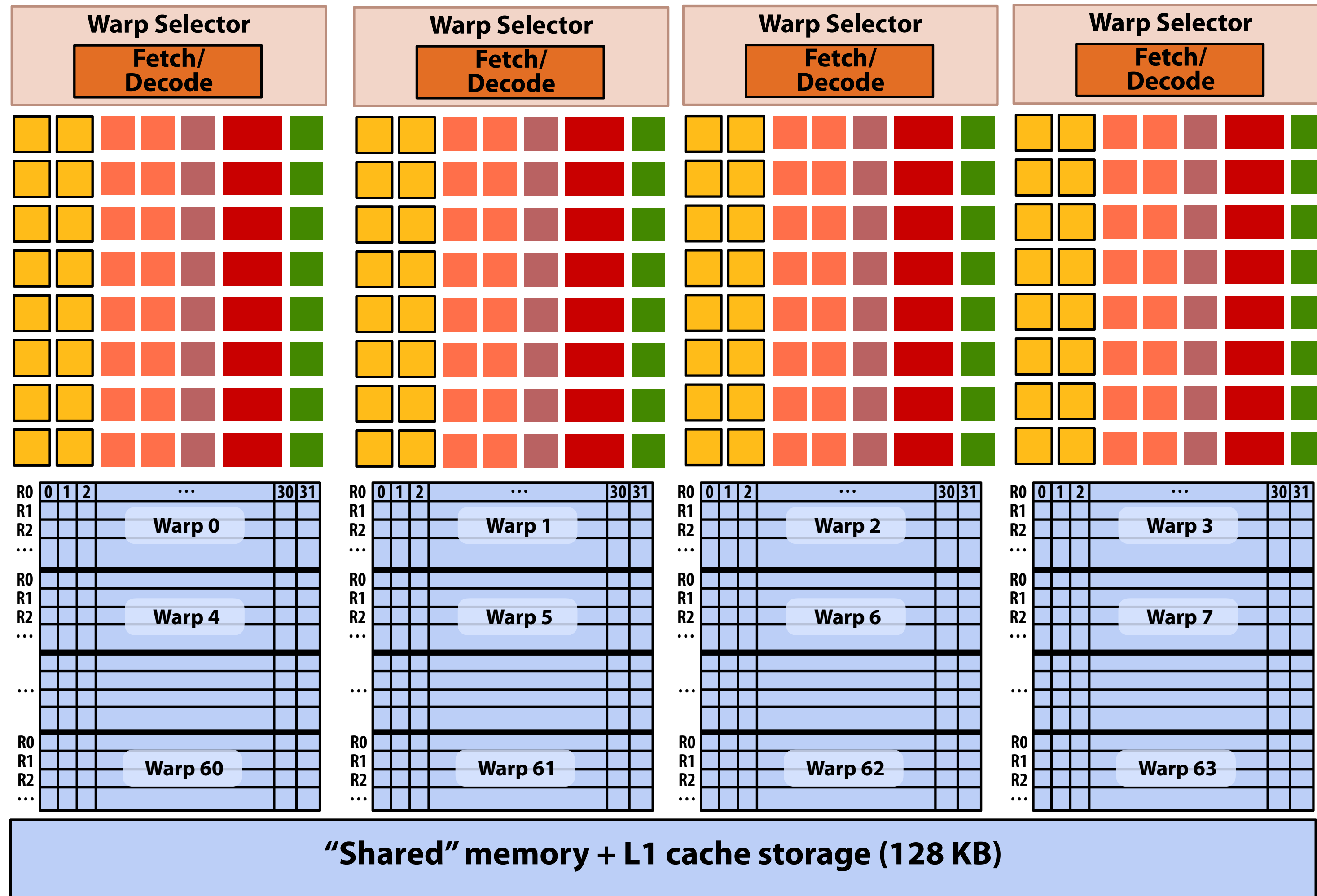
64 "warp" execution contexts per SM

Wide SIMD: 16-wide SIMD ALUs (carry out 32-wide SIMD execute over 2 clocks)

64 x 32 = up to 2048 data items processed concurrently per "SM" core

R0 R1 R2 ...	0 1 2 ... 30 31
Warp 0	Warp 1	Warp 2	Warp 3
Warp 4	Warp 5	Warp 6	Warp 7
Warp 60	Warp 61	Warp 62	Warp 63

64 KB registers per sub-core

256 KB registers in total per SM

Registers divided among (up to) 64 "warps" per SM

"Shared" memory + L1 cache storage (128 KB)

■ = SIMD fp32 functional unit, control shared across 16 units (16 x MUL-ADD per clock *)

■ = SIMD int functional unit, control shared across 16 units (16 x MUL/ADD per clock *)

■ = SIMD fp64 functional unit, control shared across 8 units (8 x MUL/ADD per clock **)

■ = Tensor core unit

■ = Load/store unit

* one 32-wide SIMD operation every 2 clocks

** one 32-wide SIMD operation every 4 clocks

NVIDIA V100

There are 80 SM cores on the V100:

That's 163,840 pieces of data being processed concurrently to get maximal latency hiding!

L2 Cache (6 MB)

**900 GB/sec
(4096 bit interface)**

**GPU memory (HBM)
(16 GB)**
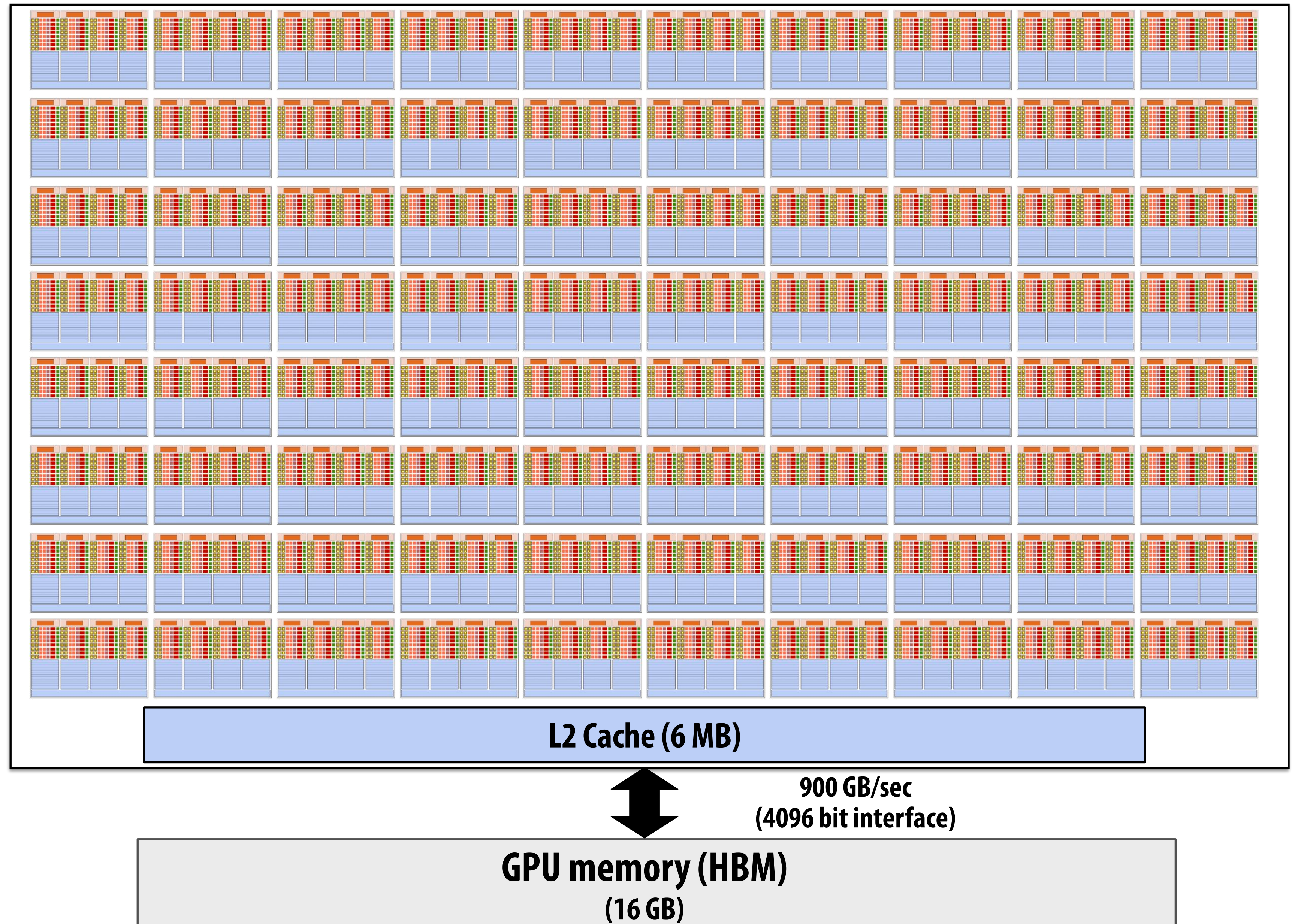
The story so far…

To utilize modern parallel processors efficiently, an application must:

1. **Have sufficient parallel work to utilize all available execution units (across many cores and many execution units per core)**

2. **Groups of parallel work items must require the same sequences of instructions (to utilize SIMD execution)**

3. **Expose more parallel work than processor ALUs to enable interleaving of work to hide memory stalls**

Thought experiment

Task: element-wise multiplication of two vectors A and B

Assume vectors contain millions of elements

- Load input A[i]
- Load input B[i]
- Compute A[i] × B[i]
- Store result into C[i]
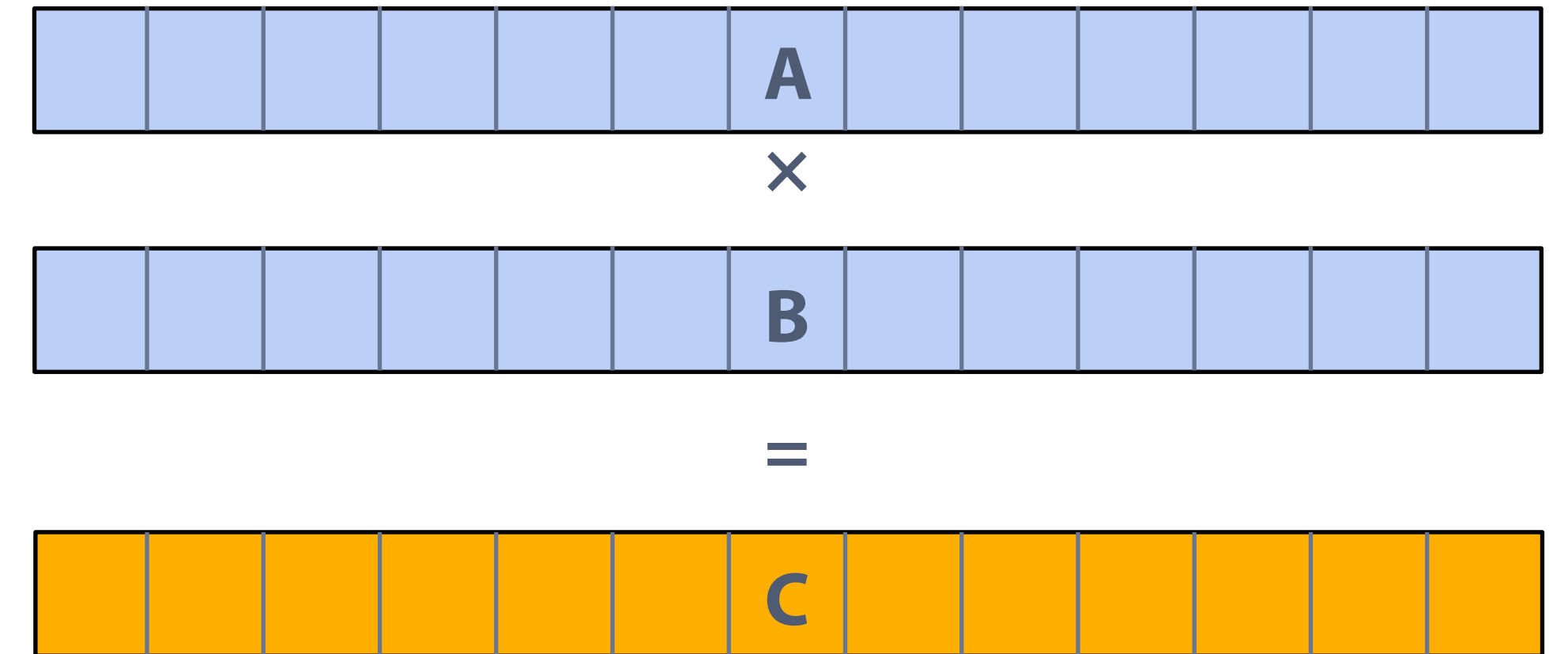
Is this a good application to run on a modern throughput-oriented parallel processor? 🤔

To answer this question, we first have to understand the difference between latency and bandwidth

(We'll start there next time)

Suggestion to students: know these terms

- **Instruction stream**

- **Multi-core processor**

- **SIMD execution**

- **Coherent control flow**

- **Hardware multi-threading**
  - **Interleaved multi-threading**
  - **Simultaneous multi-threading**

Bonus slides:

REVIEW

HOW IT ALL FITS TOGETHER:

**superscalar execution,
SIMD execution,
multi-core execution,
and hardware multi-threading**

(If you understand this sequence you understand lecture 2)

Running code on a simple processor

C program source

```
void sinx(int N, int terms, float* x, float* y)
{
    for (int i=0; i<N; i++)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6;  // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        y[i] = value;
    }
}
```

compiler →

Compiled instruction stream (scalar instructions)

```
ld    r0, addr[r1]
mul   r1, r0, r0
add   r2, r0, r0
mul   r3, r1, r2
...
...
...
...
...
...
st    addr[r2], r0
```

Running code on a simple processor

Instruction stream

`ld    r0, addr[r1]`
`mul  r1, r0, r0`
`add  r2, r0, r0`
`mul  r3, r1, r2`
`...`
`...`
`...`
`...`
`...`
`st    addr[r2], r0`

Memory

Data Cache

Fetch/ Decode

ALU (Execution unit)

Execution Context (HW thread)

PC

R0	R4
R1	R5
R2	R6
R3	R7
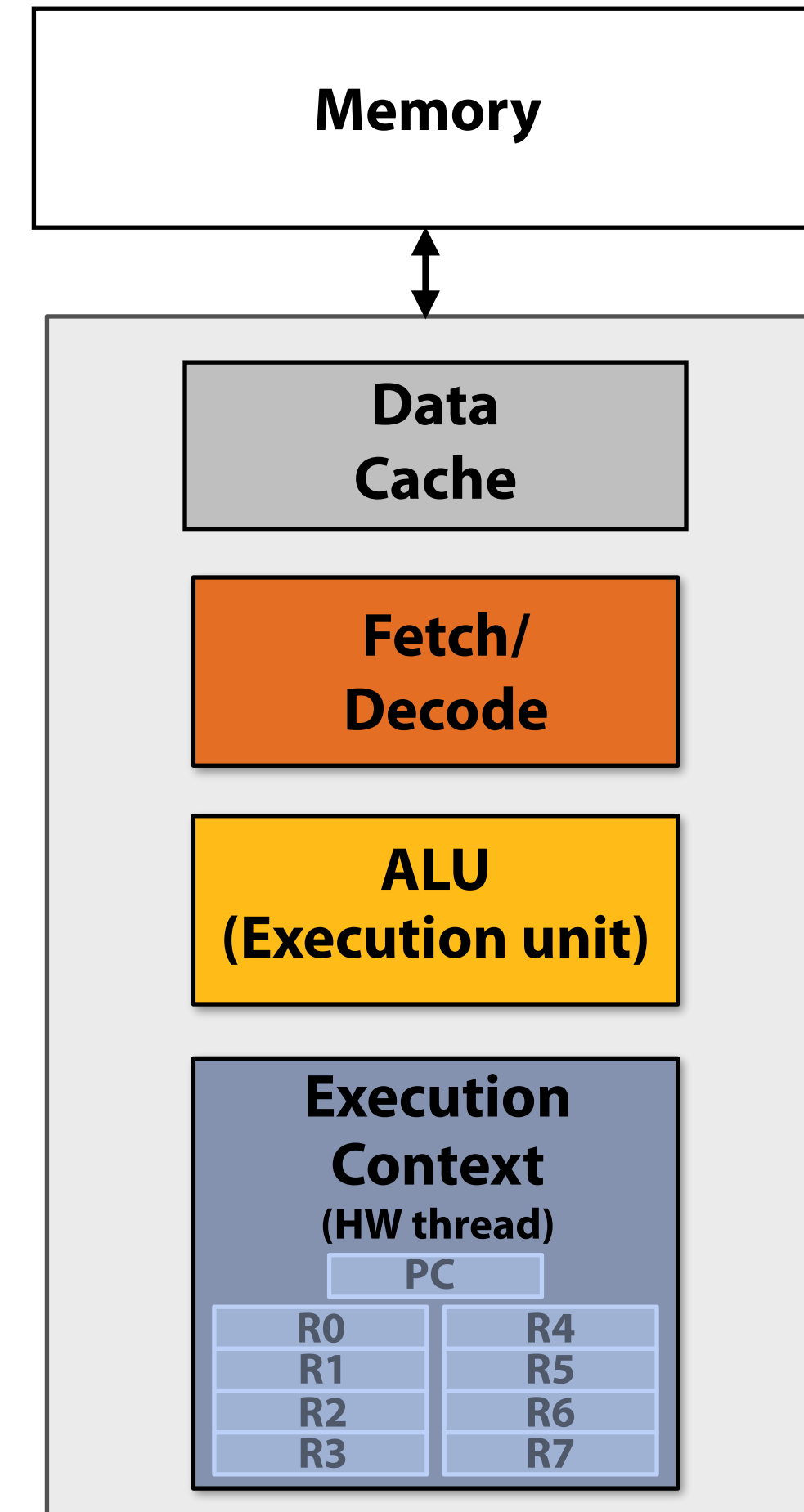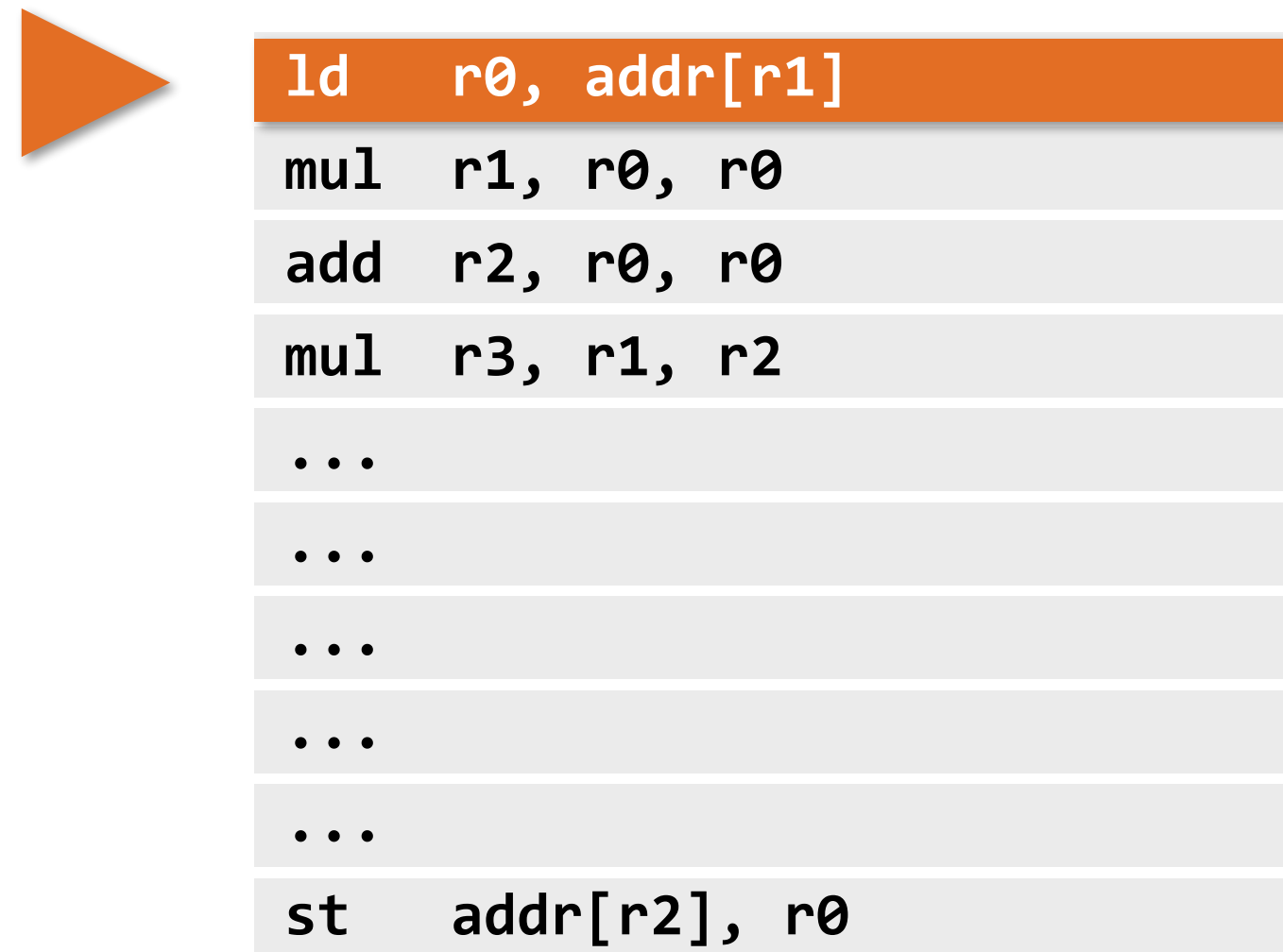
**Single core processor, single-threaded core.
Can run one scalar instruction per clock**

Superscalar core
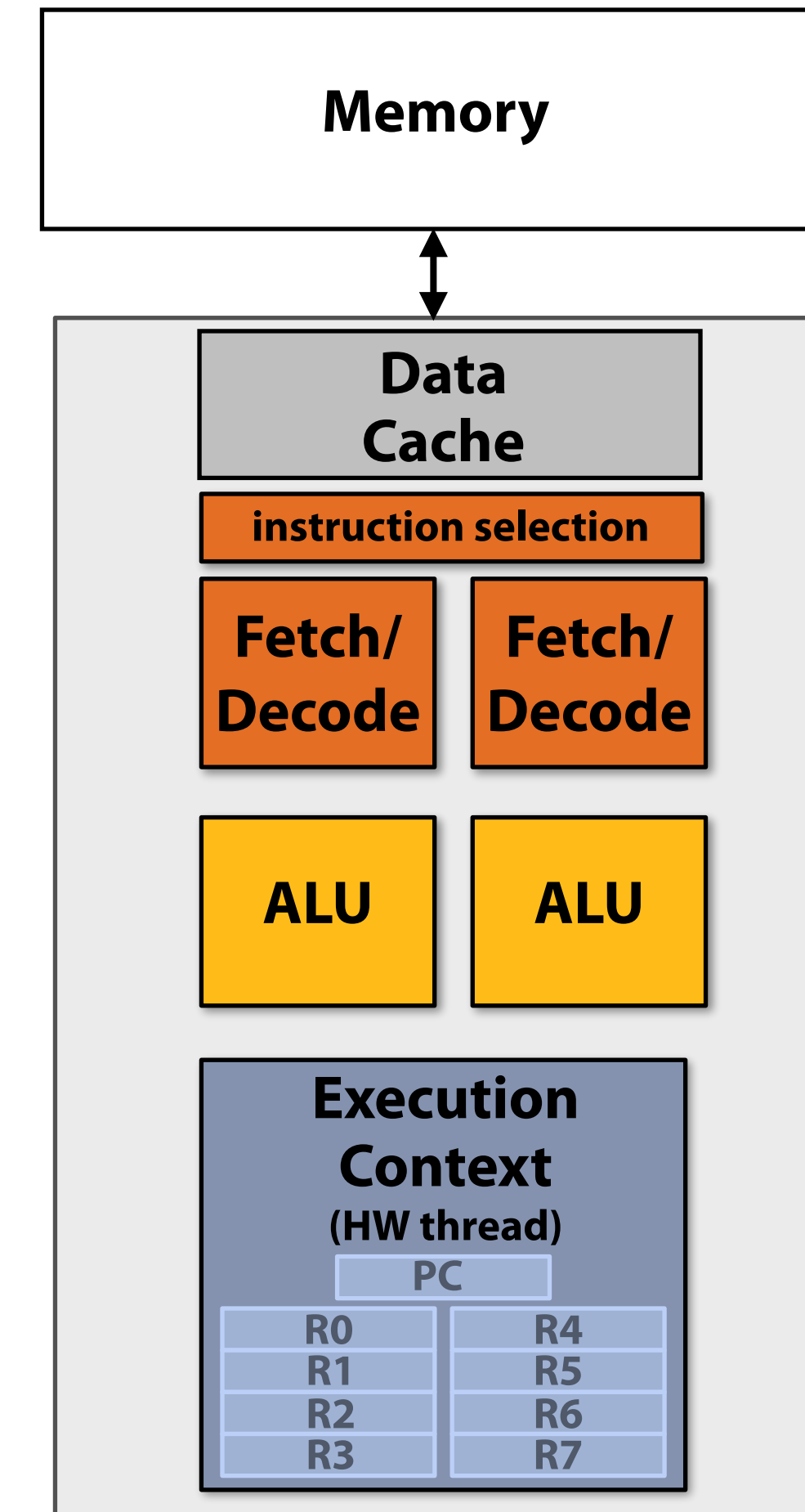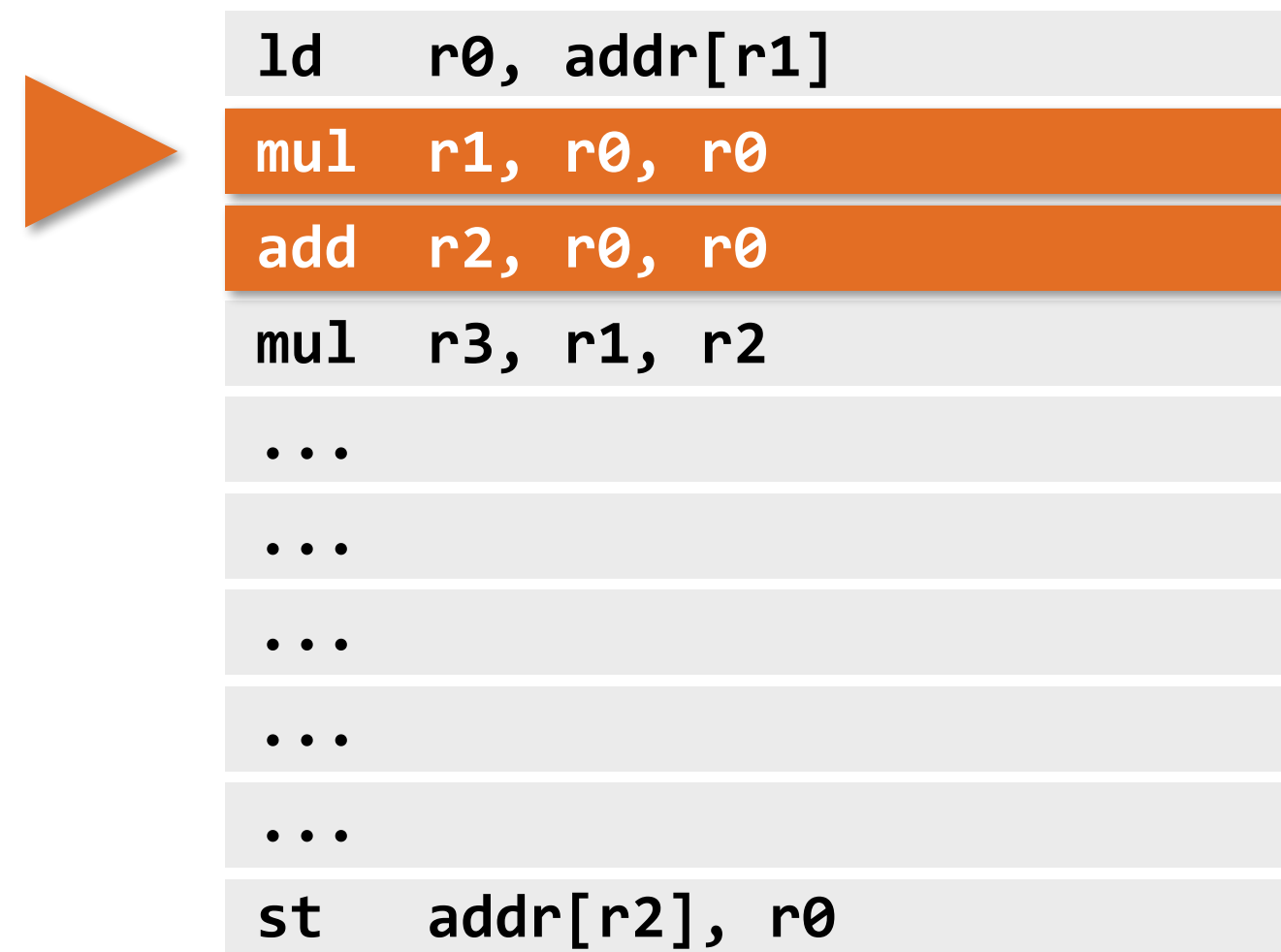
Instruction stream

```
ld   r0, addr[r1]
mul  r1, r0, r0
add  r2, r0, r0
mul  r3, r1, r2
...
...
...
...
...
st   addr[r2], r0
```

Memory

Data Cache

instruction selection

Fetch/ Decode   **Fetch/ Decode**

ALU   **ALU**

Execution Context
(HW thread)

PC

R0	R4
R1	R5
R2	R6
R3	R7

Single core processor, single-threaded core.
Two-way superscalar core:
can run up to two independent scalar instructions
per clock from one instruction stream (one hardware thread)

SIMD execution capability

**Instruction stream
(now with vector instructions)**


```
vector_ld    v0, vector_addr[r1]
vector_mul   v1, v0, v0
vector_add   v2, v0, v0
vector_mul   v3, v1, v2
...
...
...
...
...
vector_st    addr[r2], v0
```

Memory

Data Cache

Fetch/ Decode

| ALU | ALU | ALU | ALU |
| ALU | ALU | ALU | ALU |

(8-wide vector ALU)

Execution Context
(HW thread)

PC

V0	V4
V1	V5
V2	V6
V3	V7
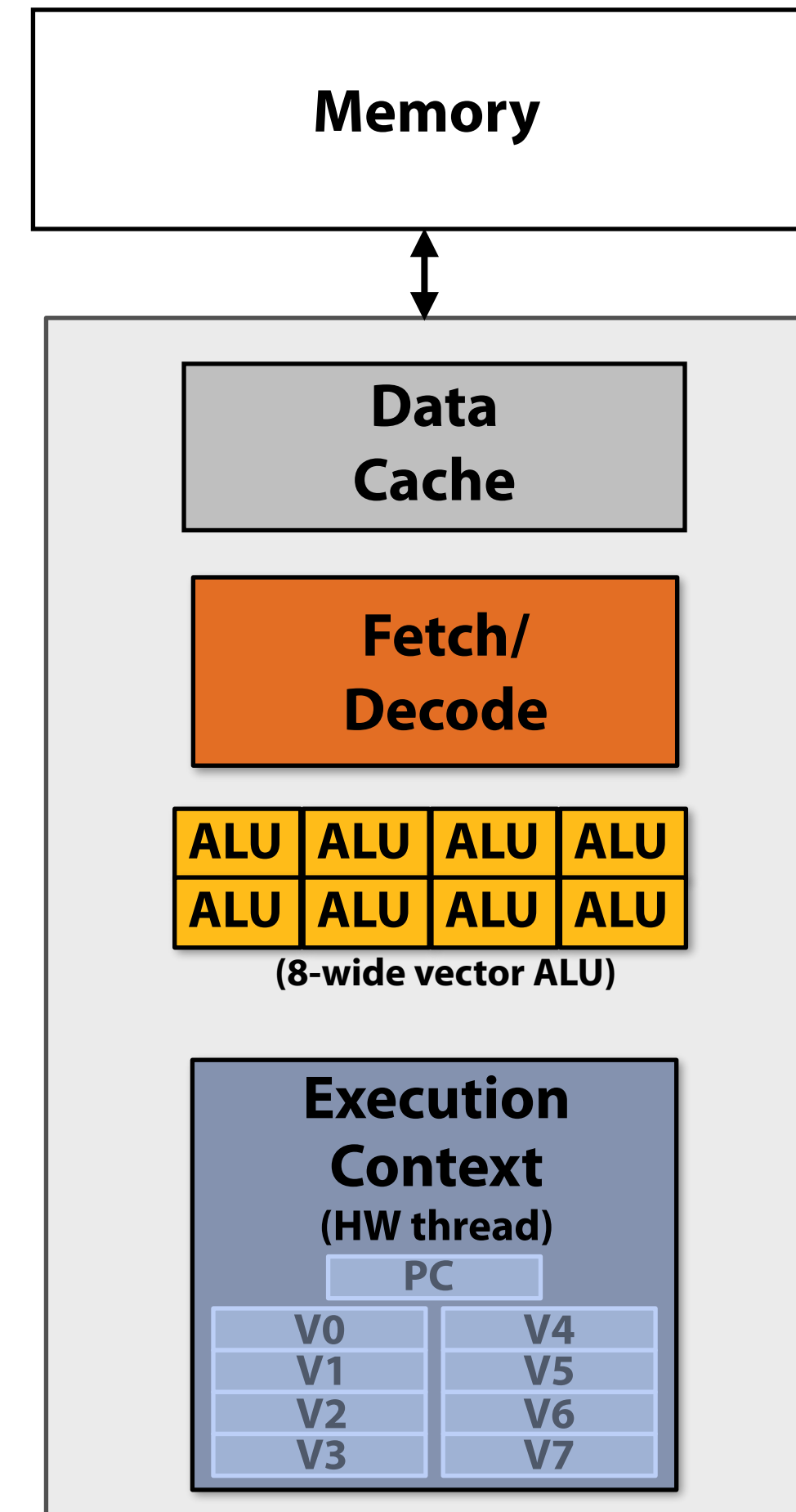
**Single core processor, single-threaded core.
can run one 8-wide SIMD vector instruction from
one instruction stream**

Heterogeneous superscalar (scalar + SIMD)

Instruction stream

```
vector_ld    v0, vector_addr[r1]
vector_mul   v1, v0, v0
add          r2, r1, r0
vector_add   v2, v0, v0
vector_mul   v3, v1, v2...
...
...
...
...
vector_st    addr[r2], v0
```

Memory

Data Cache

instruction selection

Fetch/ Decode   **Fetch/ Decode**

ALU | ALU | ALU | ALU | ALU |
          | ALU | ALU | ALU | ALU |

(scalar ALU)   **(8-wide vector ALU)**

Execution Context
(HW thread)

PC	
R0	V0
R1	V1
R2	V2
R3	V3

Single core processor, single-threaded core.
Two-way superscalar core:
can run up to two independent instructions per clock from one
instruction stream, provided one is scalar and the other is vector

Multi-threaded core

Instruction stream 0   **Instruction stream 1**

```
ld    r0, addr[r1]              ld    r0, addr[r1]
mul  r1, r0, r0                 sub  r1, r0, r0
add  r2, r0, r0                 add  r2, r1, r0
mul  r3, r1, r2                 mul  r5, r1, r0
...                             ...
...                             ...
...                             ...
...                             ...
...                             ...
st   addr[r2], r0               st   addr[r2], r0
```

Note: threads can be running completely different instruction streams (and be at different points in these streams)

Execution of hardware threads is interleaved in time.

**Single core processor, multi-threaded core (2 threads).
Can run one scalar instruction per clock from
one of the instruction streams (hardware threads)**

Multi-threaded, superscalar core

Instruction stream 0

```
vector_ld   v0, addr[r1]
vector_mul  v1, v0, v0
vector_add  v2, v1, v1
mul         r2, r1, r1
...
...
...
...
...
vector_st   addr[r2], v0
```

Instruction stream 1

```
vector_ld   v0, addr[r1]
sub         r1, r0, r0
vector_add  v2, v0, v0
mul         r5, r1, r0
...
...
...
...
...
rect        addr[r2], v0
```

Note: threads can be running completely different instruction streams (and be at different points in these streams)

Execution of hardware threads is interleaved in time.

* This detail was an arbitrary decision on this slide:
a different implementation of "instruction selection" might run two
instructions where one is drawn from each thread, see next slide.

Memory

Data Cache

instruction selection

Fetch/ Decode **Fetch/ Decode**

ALU

(scalar ALU)

| ALU | ALU | ALU | ALU |
| ALU | ALU | ALU | ALU |

(8-wide vector ALU)

Execution Context 0
(HW thread)

PC

R0	V0
R1	V1
R2	V2
R3	V3

Execution Context 1
(HW thread)

PC

R0	V0
R1	V1
R2	V2
R3	V3

**Single core processor, multi-threaded core (2 threads).
Two-way superscalar core: in this example I defined my core
as being capable of running up to two independent instructions
per clock from a single instruction stream*, provided one is scalar
and the other is vector**

Multi-threaded, superscalar core

(that combines interleaved and simultaneous execution of multiple hardware threads)

Instruction stream 0

```
vector_ld    v0, addr[r1]
vector_mul   v1, v0, v0
vector_add   v2, v1, v1
mul          r2, r1, r1
...
...
...
...
...
vector_st    addr[r2], v0
```
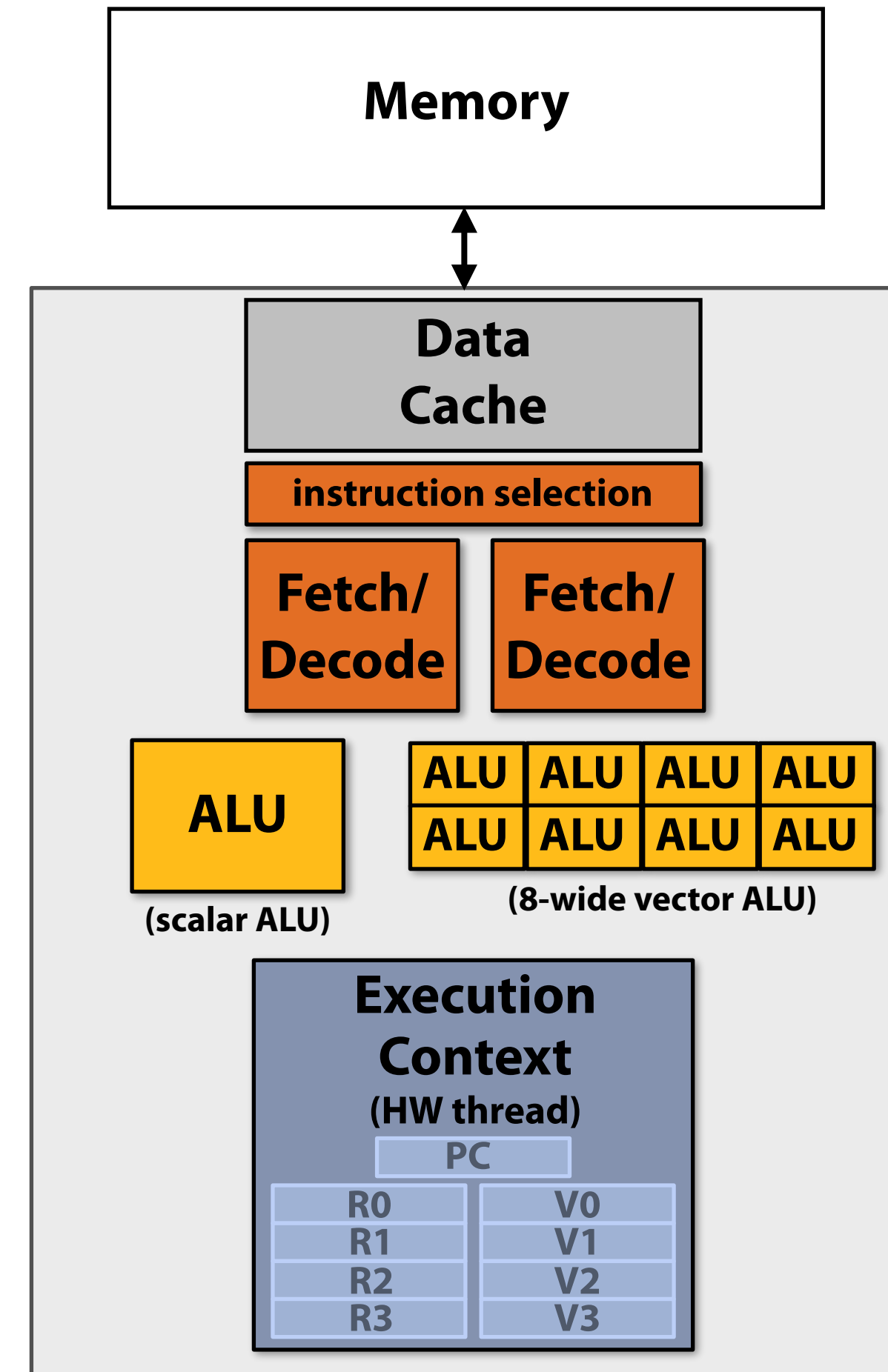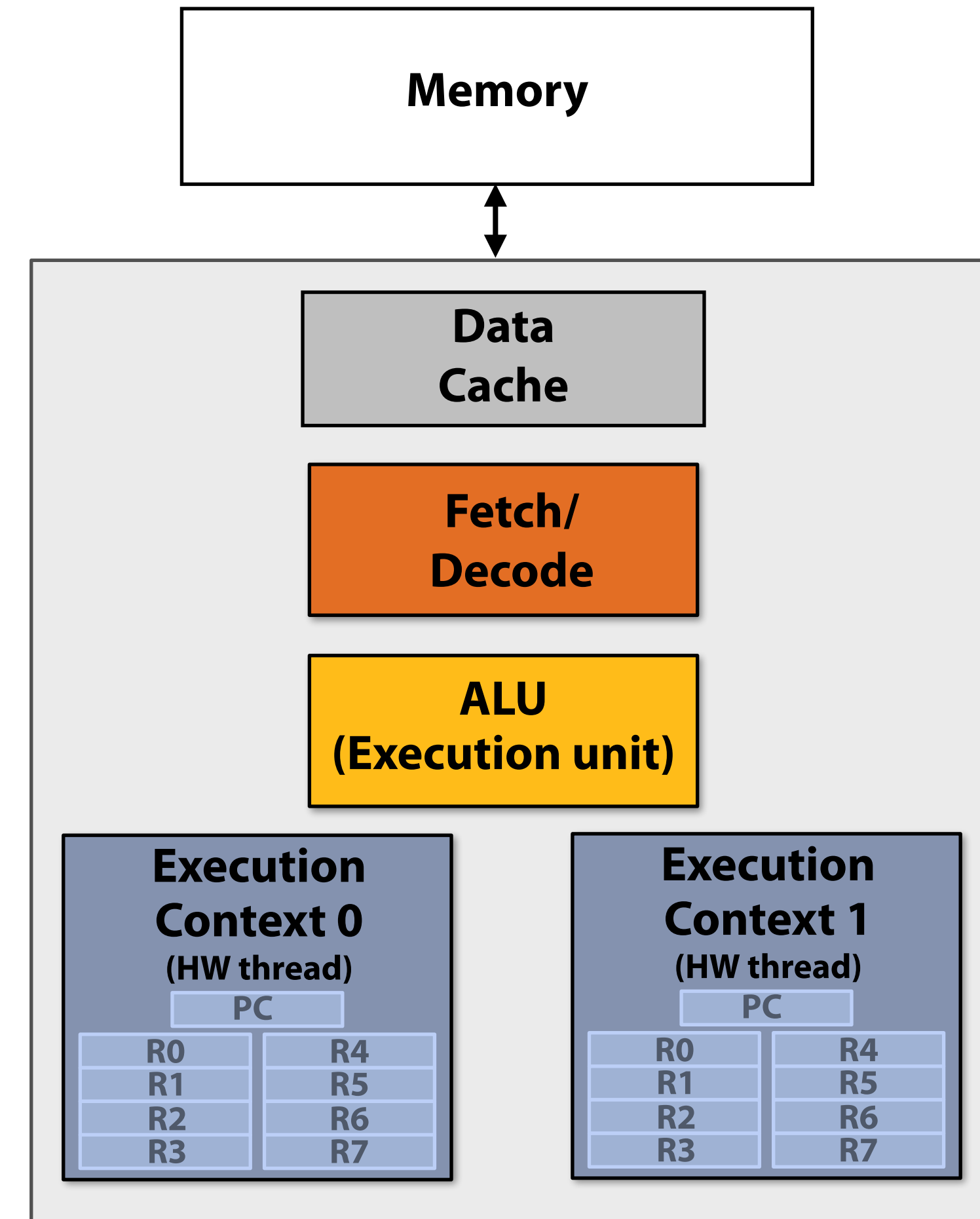
Instruction stream 1

```
vector_ld    v0, addr[r1]
sub          r1, r0, r0
vector_add   v2, v0, v0
mul          r5, r1, r0
...
...
...
...
...
rect         addr[r2], v0
```

Instruction stream 2

```
vector_ld    v0, addr[r1]
vector_mul   v2, v0, v0
mul          r3, r0, r0
sub          r1, r0, r3 ...
...
...
...
...
rect         addr[r2], v0
```

Instruction stream 3

```
vector_ld    v0, addr[r1]
sub          r1, r0, r0
vector_add   v1, v0, v0
vector_add   v2, v0, v1
mul          r2, r1, r1
...
...
...
...
rect         addr[r2], v0
```

**Execution of hardware threads may or may not be interleaved in time
(instructions from different threads may be running simultaneously)**
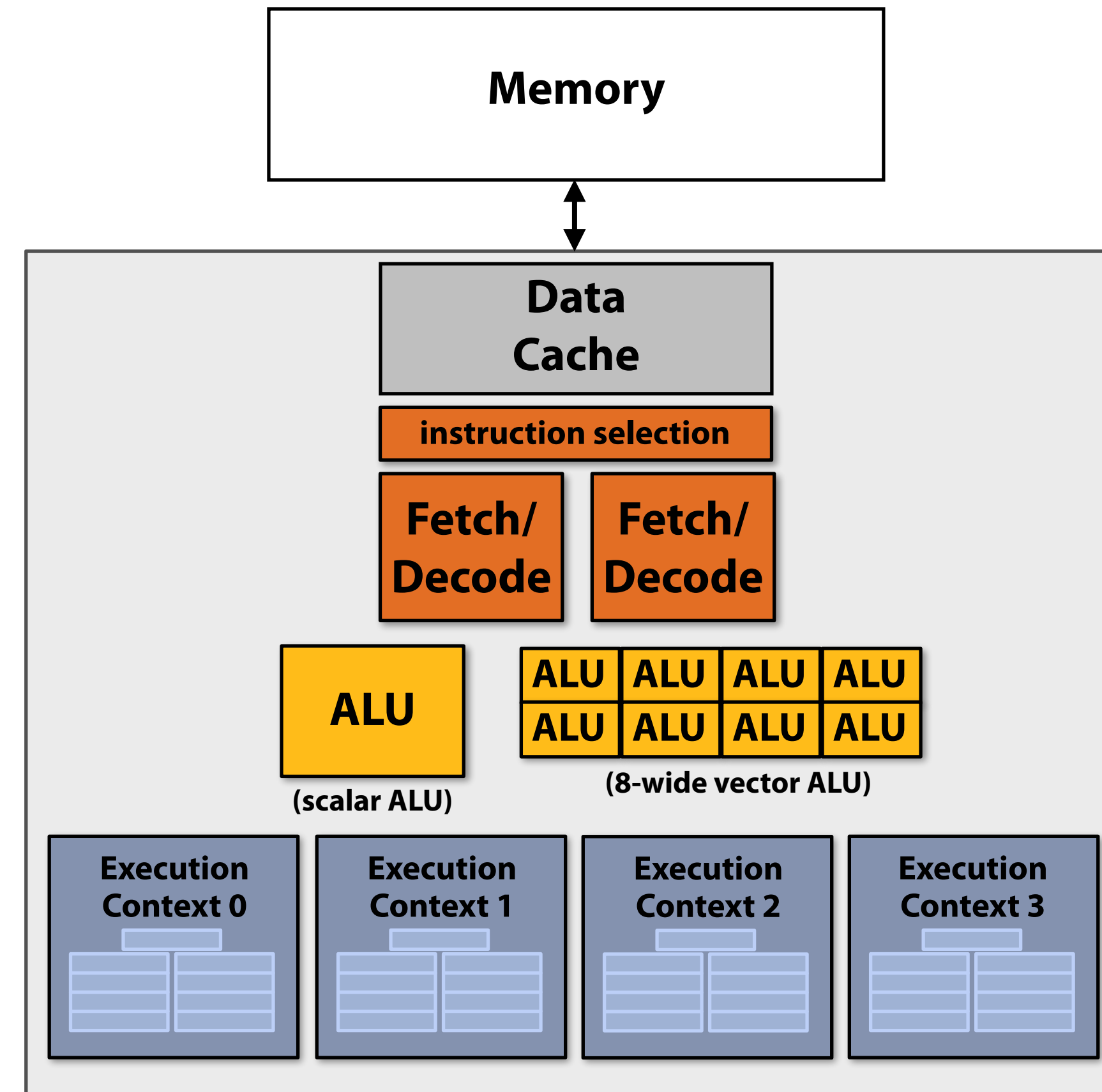
**Single core processor, multi-threaded core (4 threads).
Two-way superscalar core:
can run up to two independent instructions
per clock from <u>any of the threads</u>,
provided one is scalar and the other is vector**

Multi-core, with multi-threaded, superscalar cores
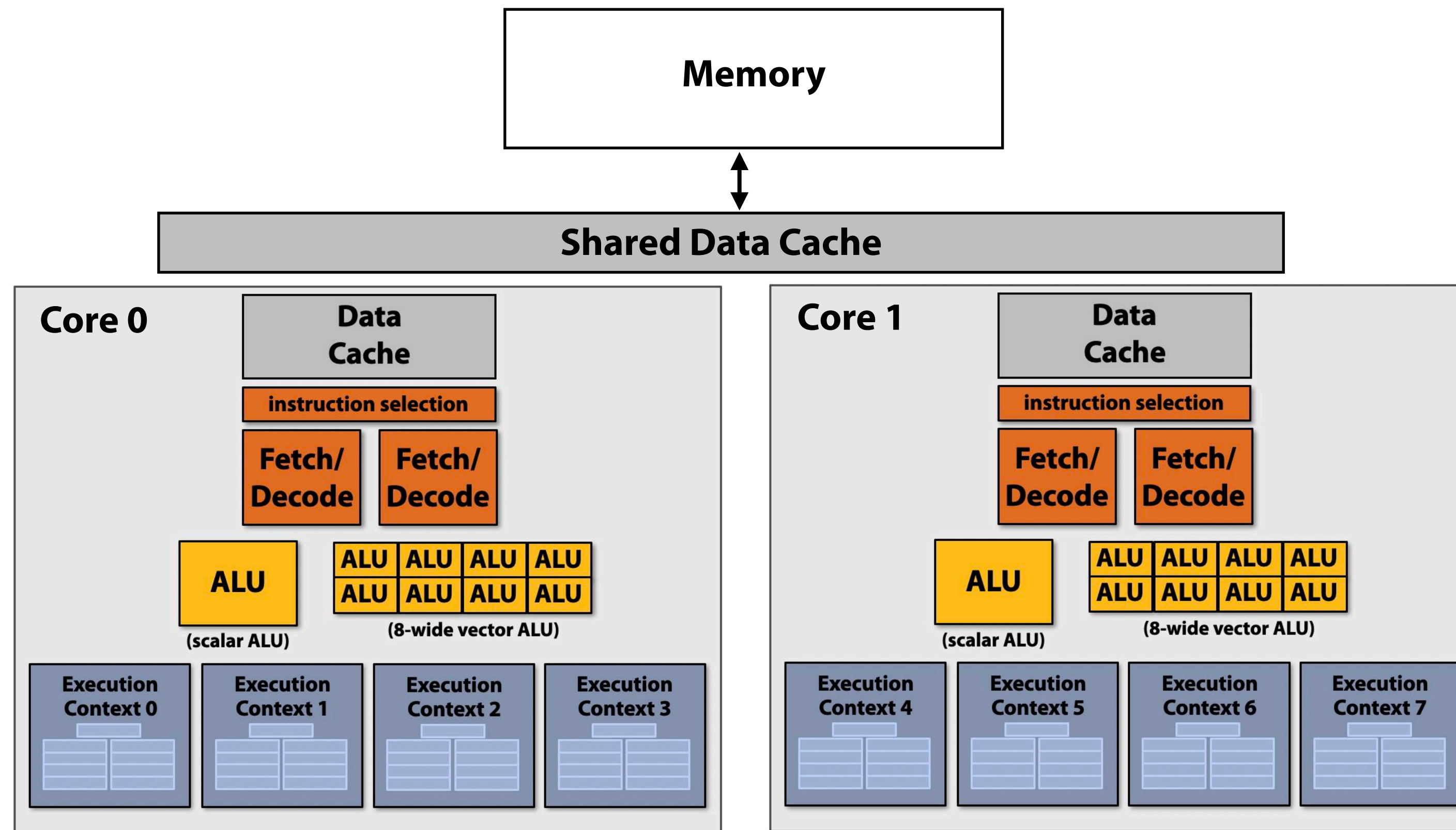
Dual-core processor, multi-threaded cores (4 threads/core).
Two-way superscalar cores: each core can run up to two independent instructions
per clock from <u>any of its threads</u>, provided one is scalar and the other is vector

Example: Intel Skylake/Kaby Lake core
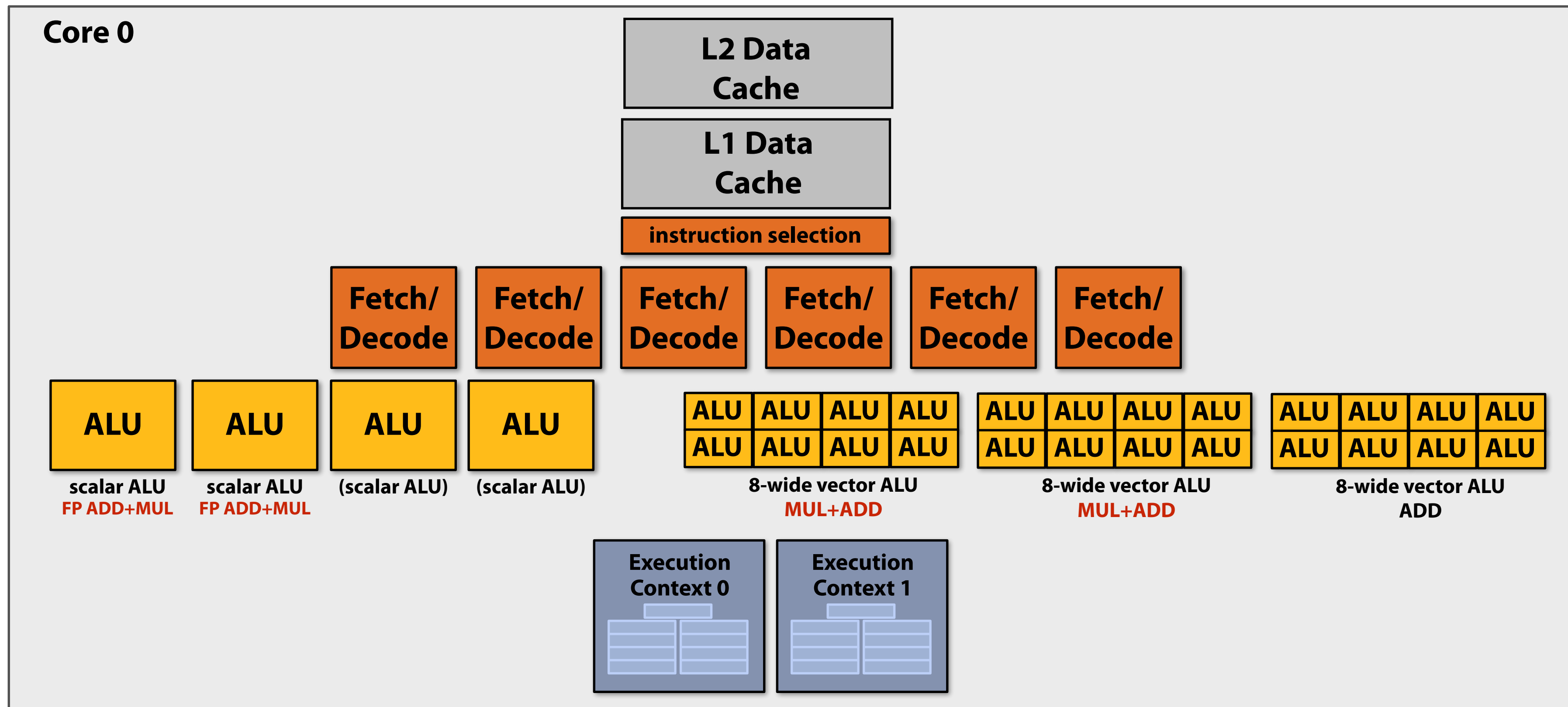
Core 0

L2 Data Cache

L1 Data Cache

instruction selection

Fetch/Decode | Fetch/Decode | Fetch/Decode | Fetch/Decode | Fetch/Decode | Fetch/Decode

ALU — scalar ALU **FP ADD+MUL**
ALU — scalar ALU **FP ADD+MUL**
ALU — (scalar ALU)
ALU — (scalar ALU)
8-wide vector ALU **MUL+ADD**
8-wide vector ALU **MUL+ADD**
8-wide vector ALU ADD

Execution Context 0 | Execution Context 1

**Two-way multi-threaded cores (2 threads).
Each core can run up to four independent scalar
instructions and up to three 8-wide vector instructions
(up to 2 vector mul or 3 vector add)**

Not shown on this diagram: units for LD/ST operations

GPU "SIMT" (single instruction multiple thread)

Instr stream 0

```
        ld   r0, addr[r1]
PC ▶    mul  r1, r0, r0
        add  r2, r0, r0
        mul  r3, r1, r2
        ...
        ...
        ...
        ...
        ...
        st   addr[r2], r0
```

Instr stream 1

```
        ld   r0, addr[r1]
PC ▶    mul  r1, r0, r0
        add  r2, r0, r0
        mul  r3, r1, r2
        ...
        ...
        ...
        ...
        ...
        st   addr[r2], r0
```

Instr stream 2

```
        ld   r0, addr[r1]
PC ▶    mul  r1, r0, r0
        add  r2, r0, r0
        mul  r3, r1, r2
        ...
        ...
        ...
        ...
        st   addr[r2], r0
```

Instr stream 3

```
        ld   r0, addr[r1]
PC ▶    mul  r1, r0, r0
        add  r2, r0, r0
        mul  r3, r1, r2
        ...
        ...
        ...
        ...
        st   addr[r2], r0
```

Instr stream 4

```
        ld   r0, addr[r1]
PC ▶    mul  r1, r0, r0
        add  r2, r0, r0
        mul  r3, r1, r2
        ...
        ...
        ...
        ...
        st   addr[r2], r0
```

Instr stream 5

```
        ld   r0, addr[r1]
PC ▶    mul  r1, r0, r0
        add  r2, r0, r0
        mul  r3, r1, r2
        ...
        ...
        ...
        ...
        st   addr[r2], r0
```

Instr stream 6

```
        ld   r0, addr[r1]
        mul  r1, r0, r0
        add  r2, r0, r0
        mul  r3, r1, r2
        ...
        ...
        ...
        ...
PC ▶    add  r4, r1, r2
        st   addr[r2], r0
```

Instr stream 7

```
        ld   r0, addr[r1]
PC ▶    mul  r1, r0, r0
        add  r2, r0, r0
        mul  r3, r1, r2
        ...
        ...
        ...
        ...
        st   addr[r2], r0
```

divergent execution

Memory

Data Cache

instruction selection

Fetch/Decode

| ALU | ALU | ALU | ALU |
| ALU | ALU | ALU | ALU |

(8-wide vector ALU)

| Execution Context 0 | Execution Context 1 | Execution Context 2 | Execution Context 3 |
| Execution Context 4 | Execution Context 5 | Execution Context 6 | Execution Context 7 |

Many modern GPUs execute hardware threads that run instruction streams with only <u>scalar instructions</u>.

GPU cores detect when different hardware threads are executing the same instruction, and implement simultaneous execution of up to SIMD-width threads using SIMD ALUs.

Here ALU 6 would be "masked off" since thread 6 is not executing the same instruction as the other hardware threads.
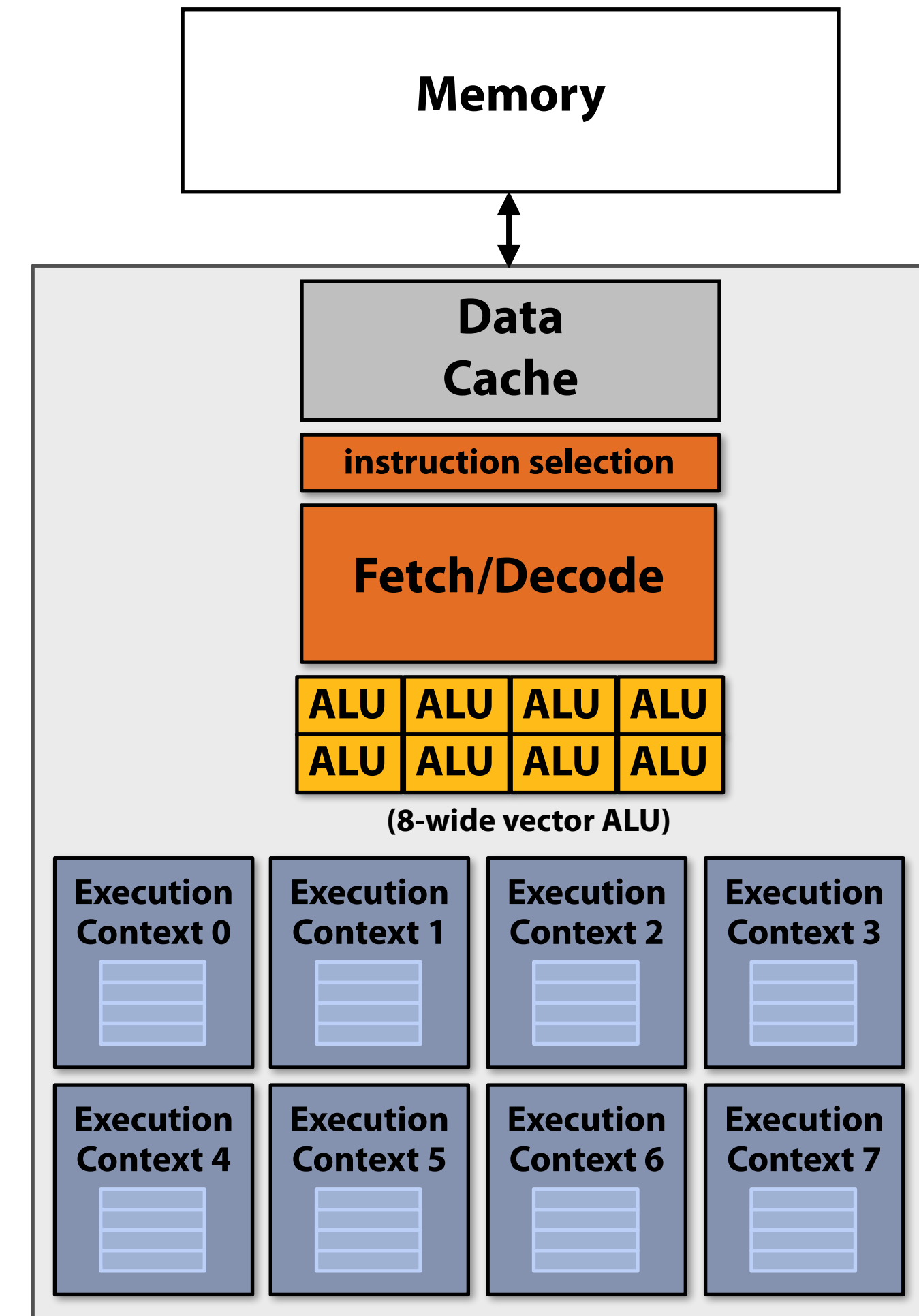
Thought experiment

- **You write a C application that spawns <u>two</u> threads**

- **The application runs on the processor shown below**
  - **Two cores, two-execution contexts per core, up to instructions per clock, one instruction is an 8-wide SIMD instruction**

- **Question: "who" is responsible for mapping the applications's threads to the processor's thread execution contexts?**

  Answer: the operating system

- **Question: If you were implementing the OS, how would to assign the two threads to the four execution contexts?**

- **Another question: How would you assign threads to execution contexts if your C program spawned <u>five</u> threads?**