

Lecture 2:

A Modern Multi-Core Processor

(Forms of parallelism + understanding latency and bandwidth)

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2015

Tunes

BlakRoc

(featuring Billy Danze, Jim Jones, Nicole Wray)

“What you do to me”

“I had eight wide vector units and after a night of debugging my program was getting a 3X speedup. So called Nikki in to try and find the divergence and I went and wrote the song instead.”

- Billy Danze

Quick review

- 1. Why has single-instruction-stream performance only improved very slowly in recent years? ***
- 2. What prevented us from obtaining maximum speedup from the parallel programs we wrote last time?**

*** Self check 1: What do I mean by “single instruction stream”?**

Self check 2: When we talked about the optimization of superscalar execution, we’re we talking about a single-instruction stream situation?

Today

- **Today we will talk computer architecture**
- **Four key concepts about how modern computers work**
 - **Two concern parallel execution**
 - **Two concern challenges of accessing memory**
- **Understanding these architecture basics will help you**
 - **Understand and optimize the performance of your parallel programs**
 - **Gain intuition about what workloads might benefit from fast parallel machines**

Part 1: parallel execution

Example program

Compute $\sin(x)$ using Taylor expansion: $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \dots$
for each element of an array of N floating-point numbers

```
void sinx(int N, int terms, float* x, float* result)
{
    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;
        }

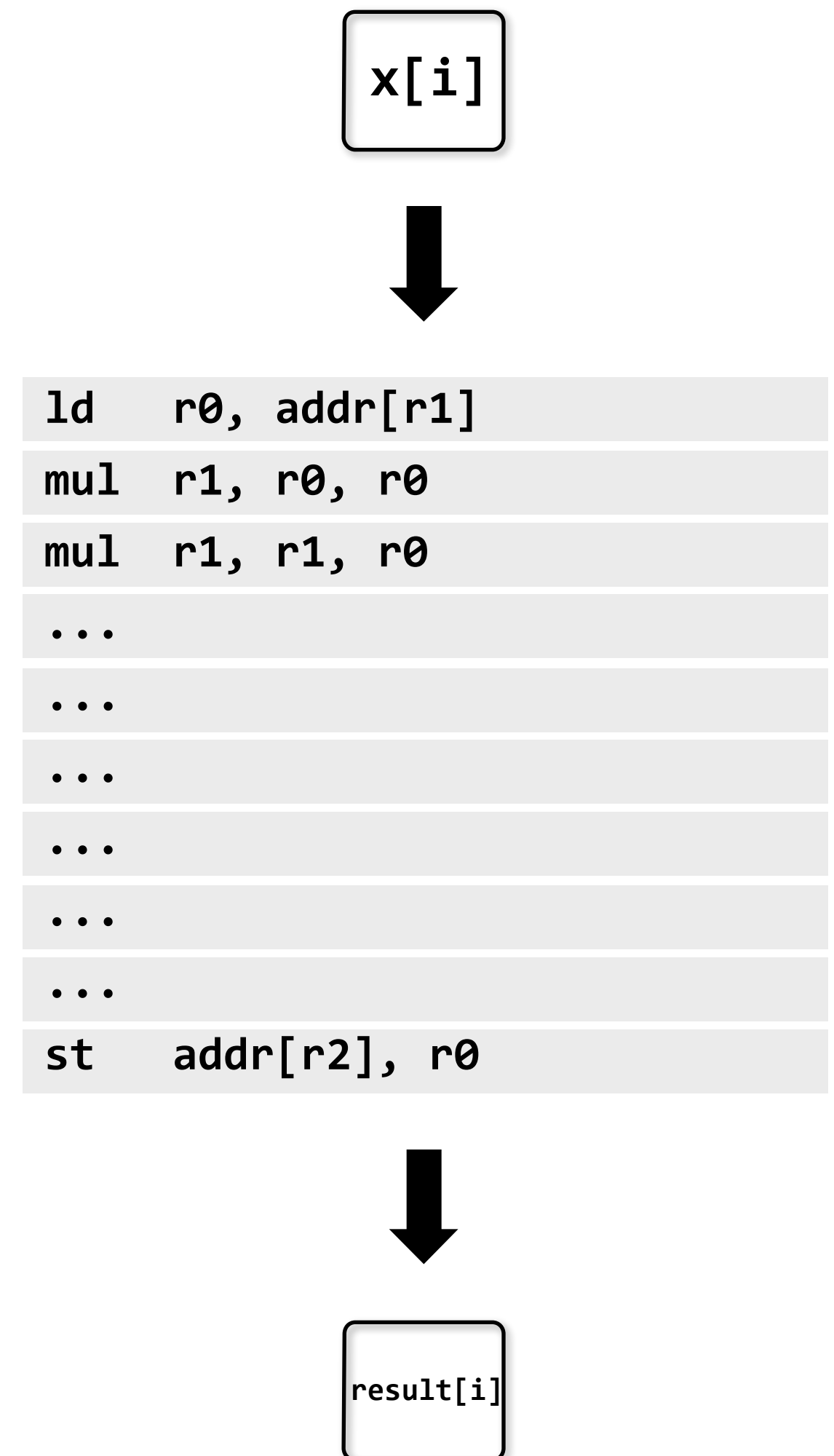
        result[i] = value;
    }
}
```

Compile program

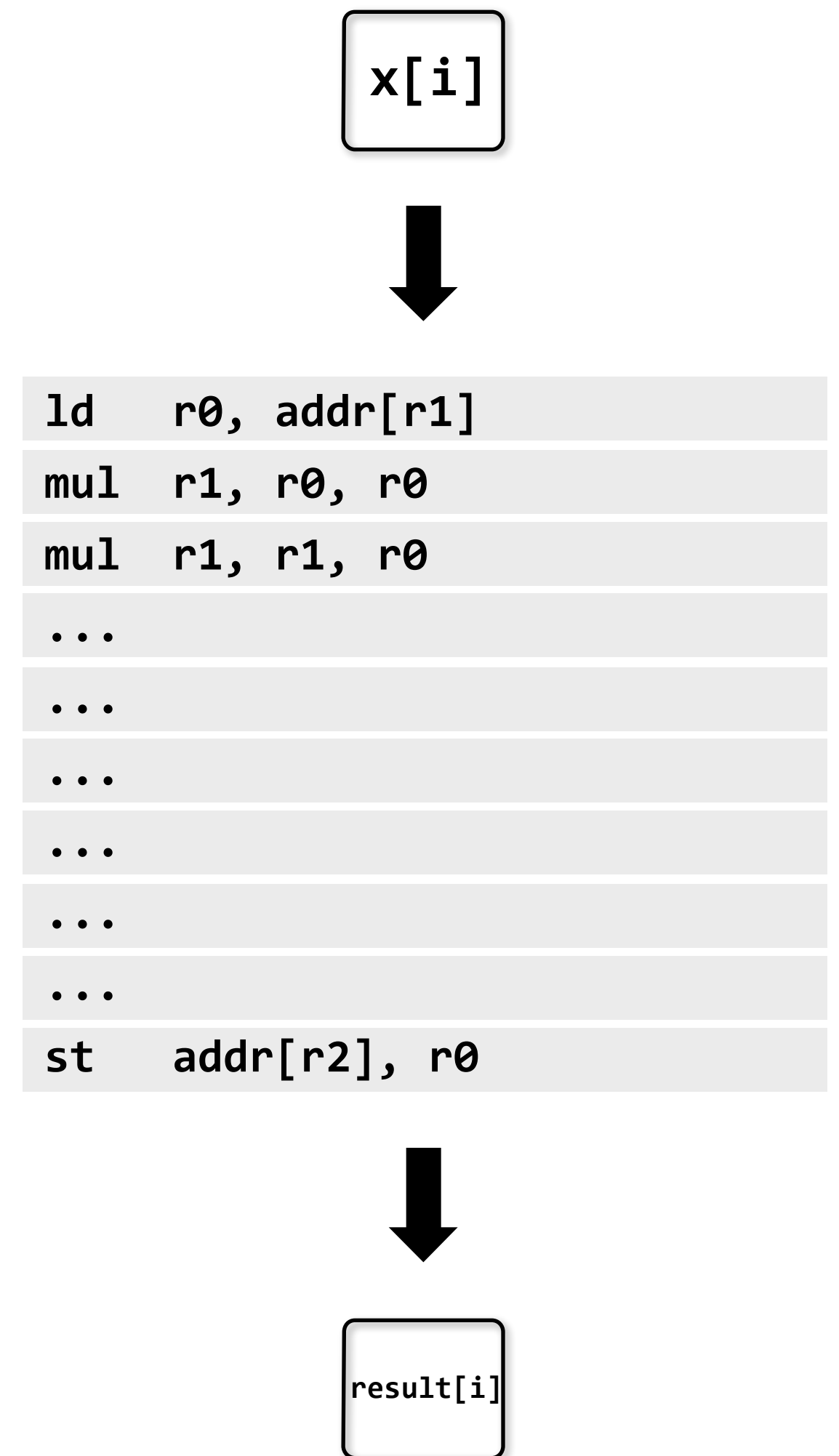
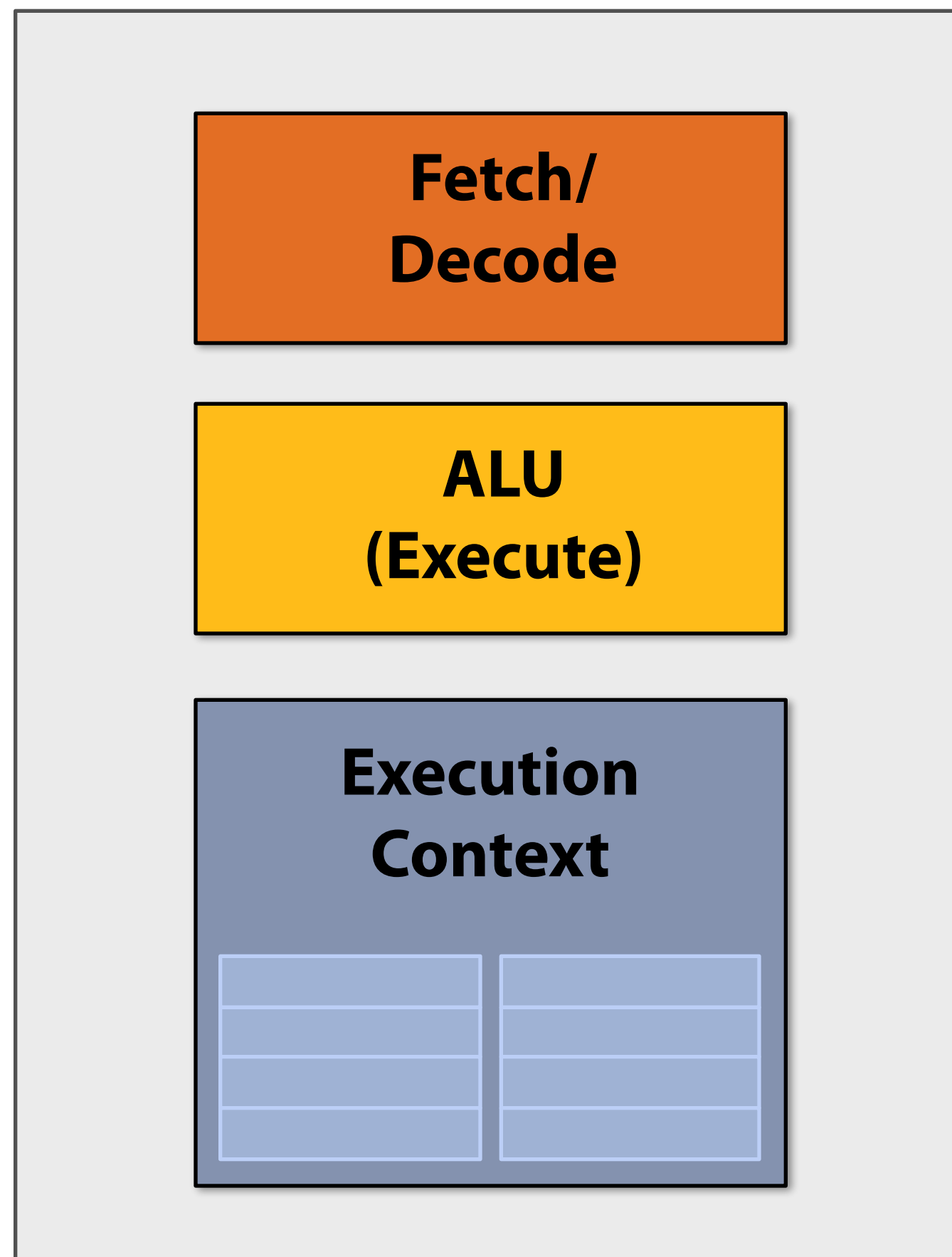
```
void sinx(int N, int terms, float* x, float* result)
{
    for (int i=0; i<N; i++)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6; // 3!
        int sign = -1;

        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;
        }

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

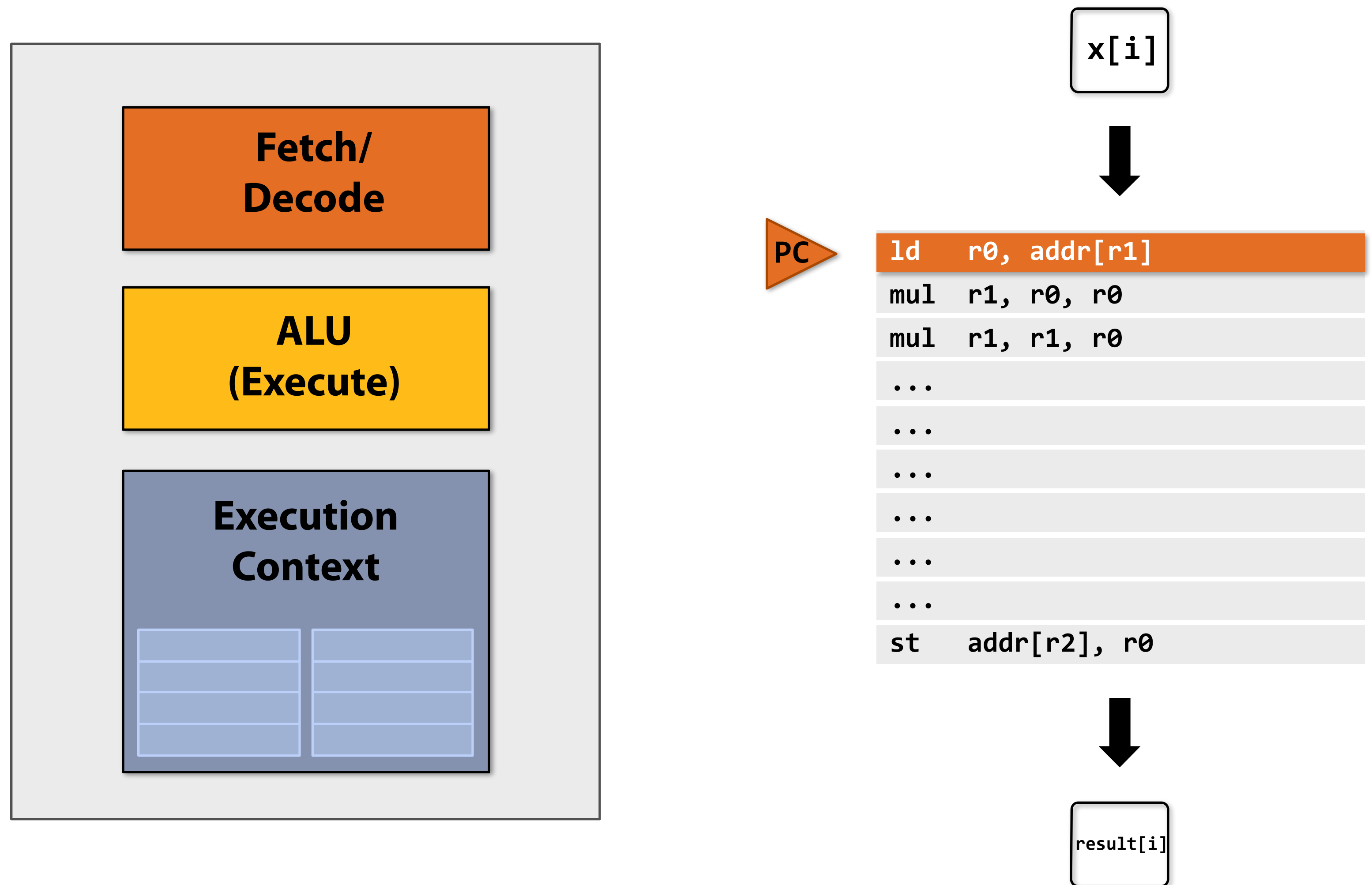


Execute program



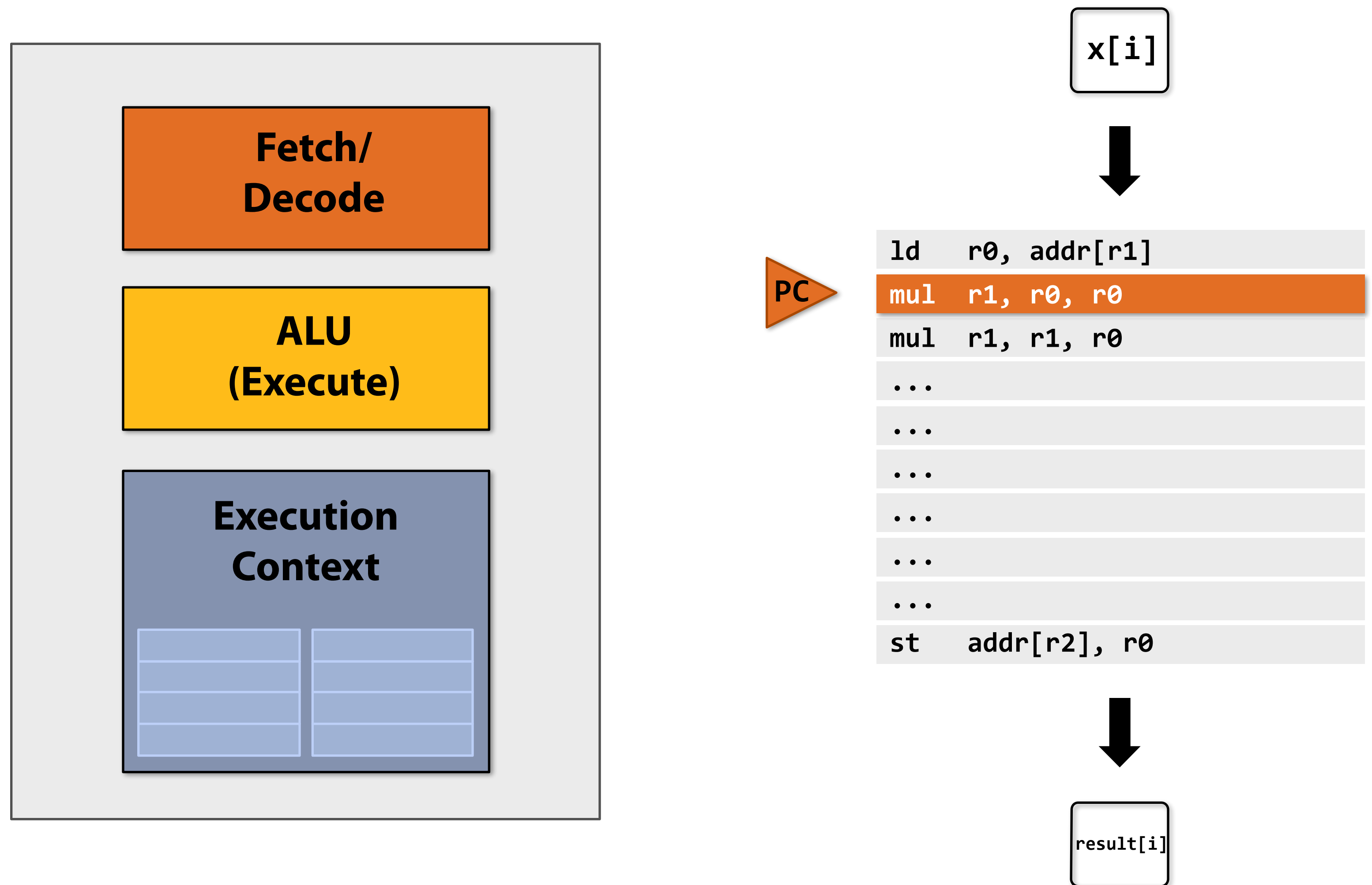
Execute program

My very simple processor: executes one instruction per clock



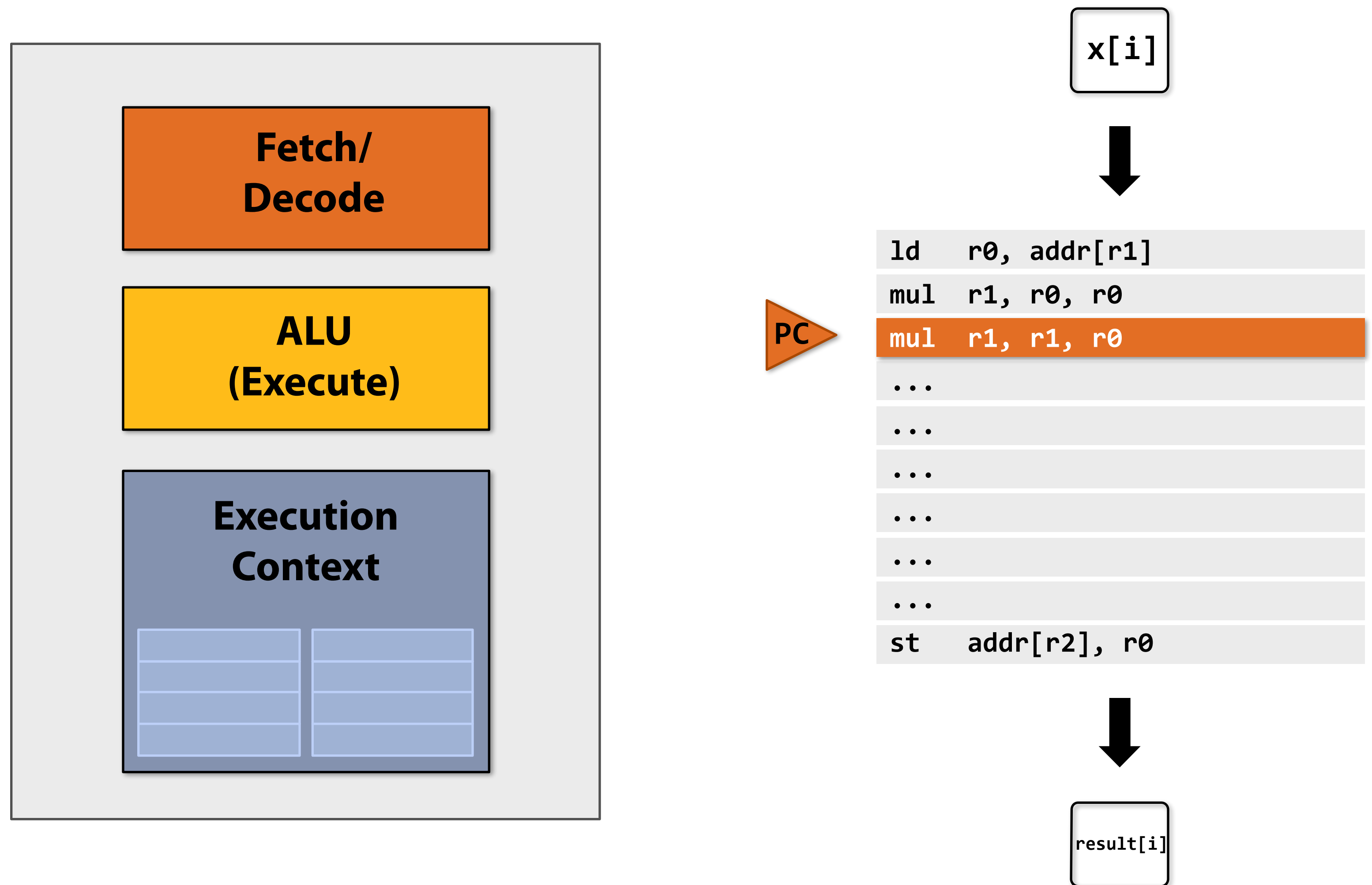
Execute program

My very simple processor: executes one instruction per clock



Execute program

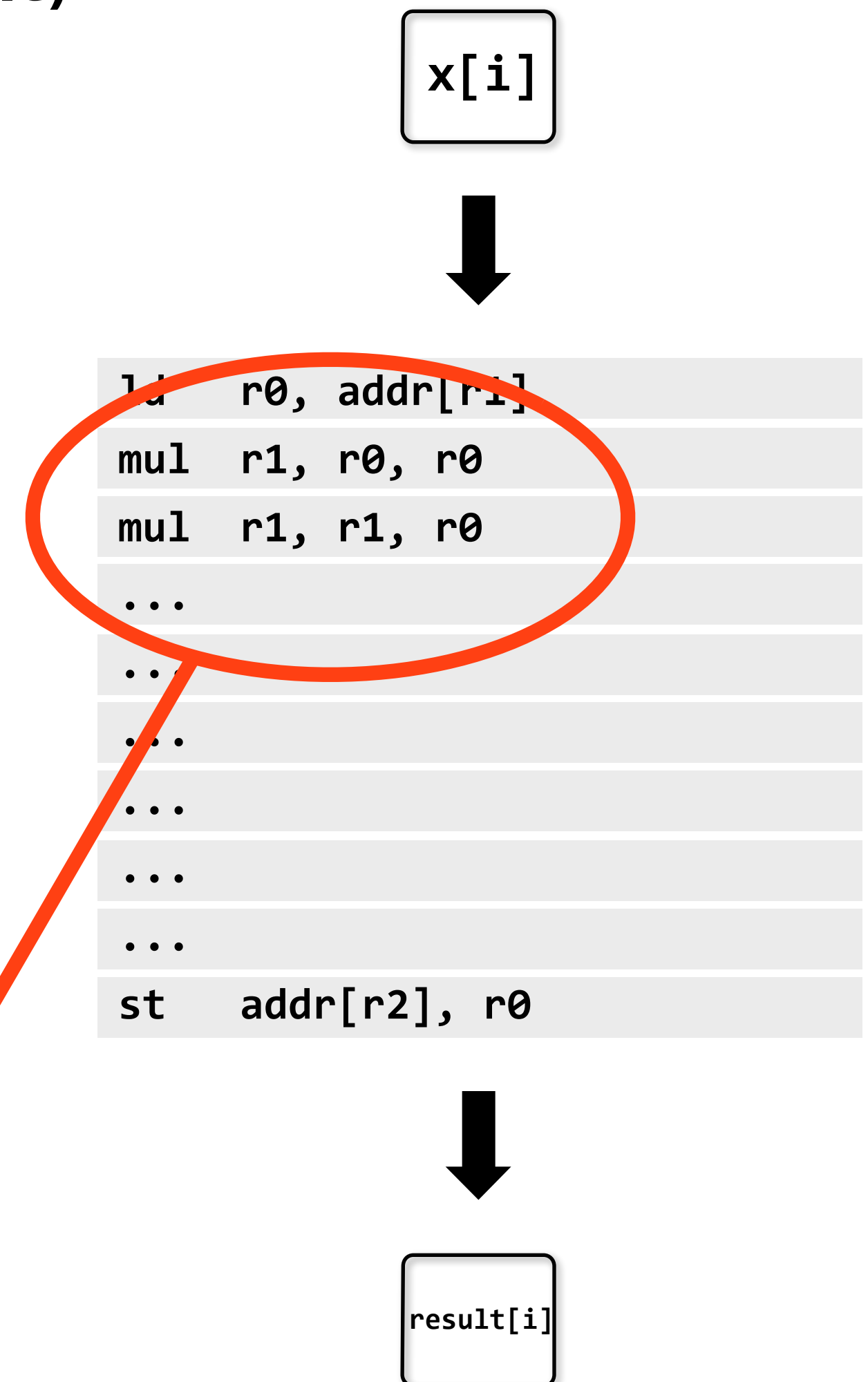
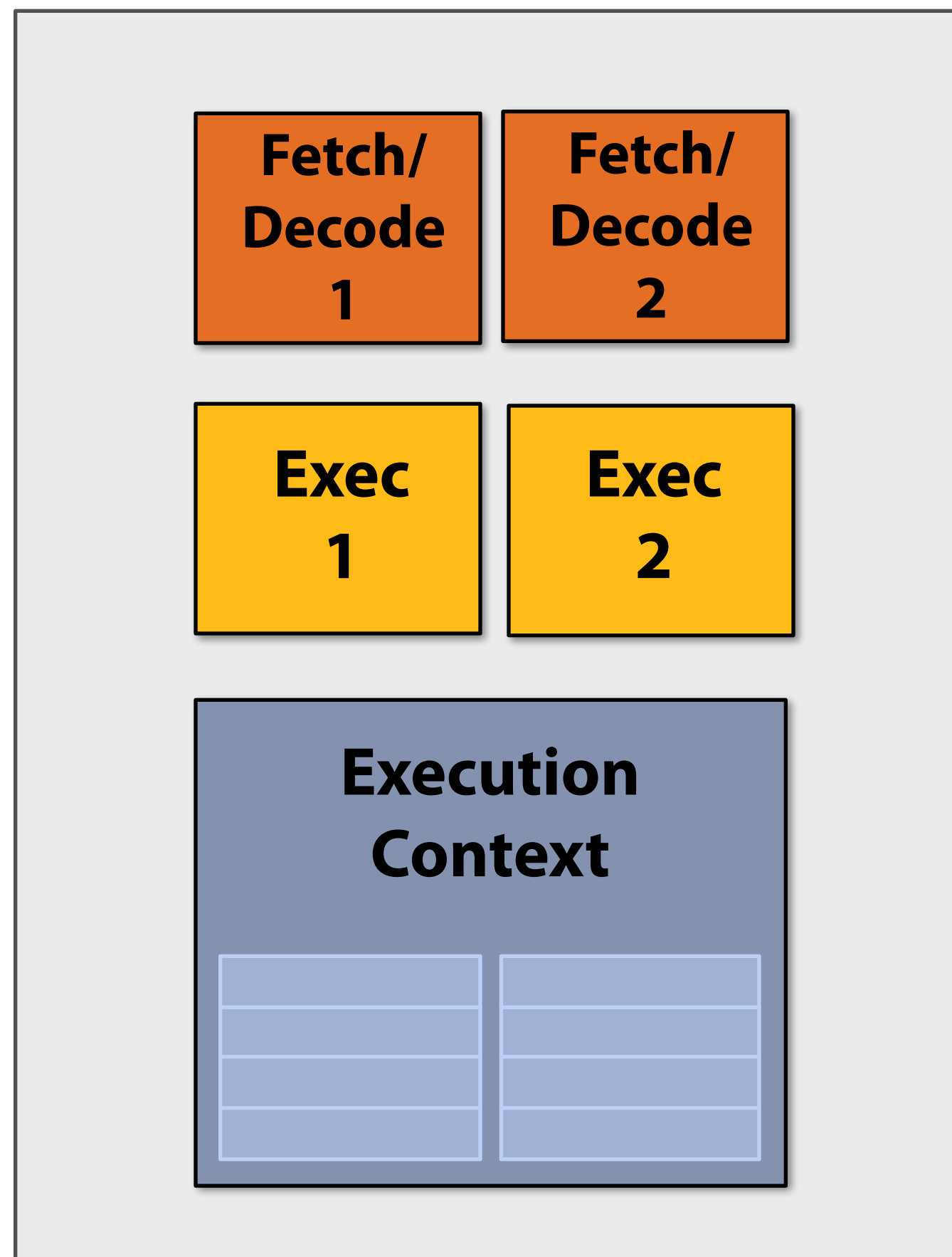
My very simple processor: executes one instruction per clock



Superscalar processor

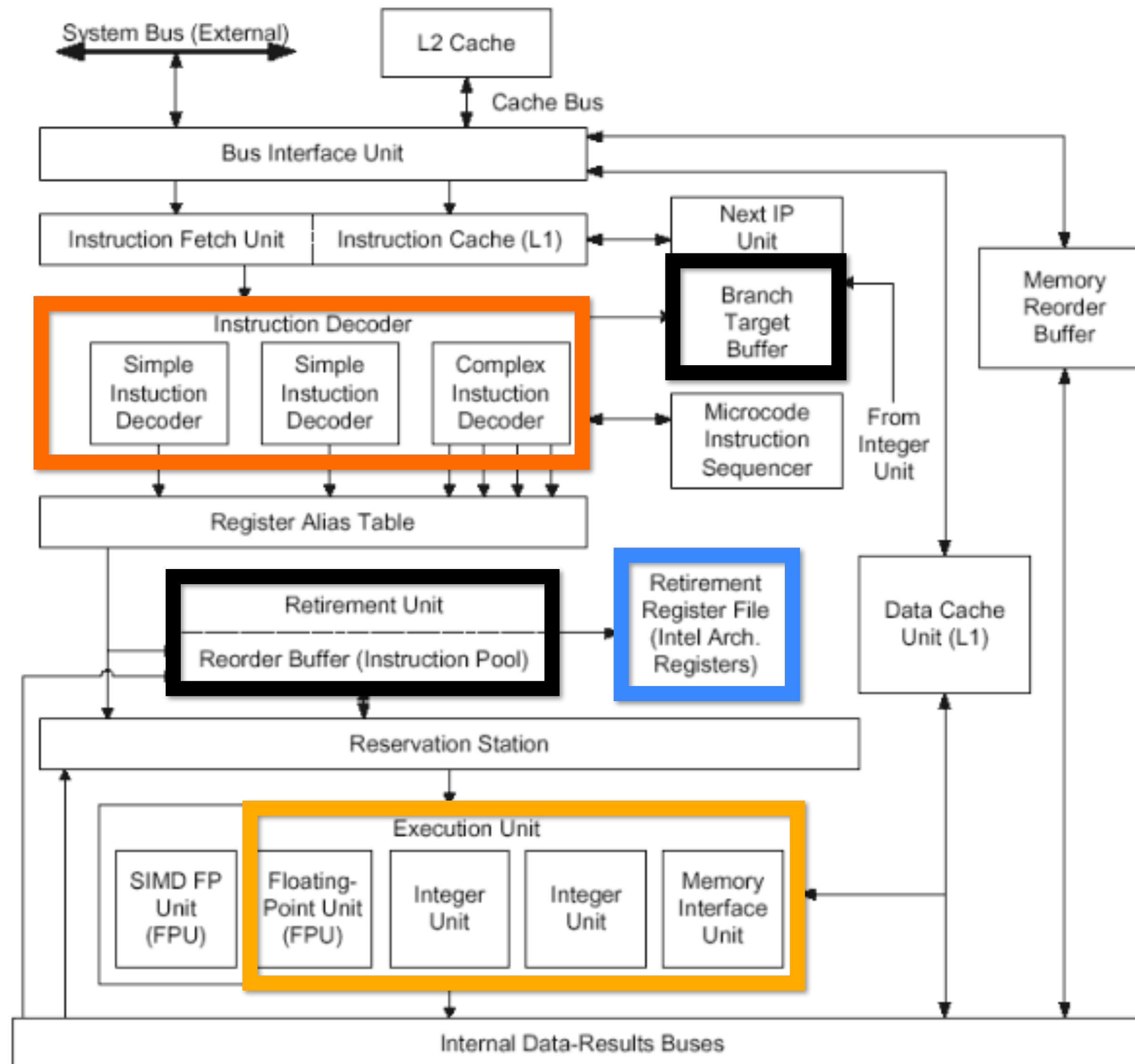
Recall from last class: instruction level parallelism (ILP)

Decode and execute two instructions per clock (if possible)



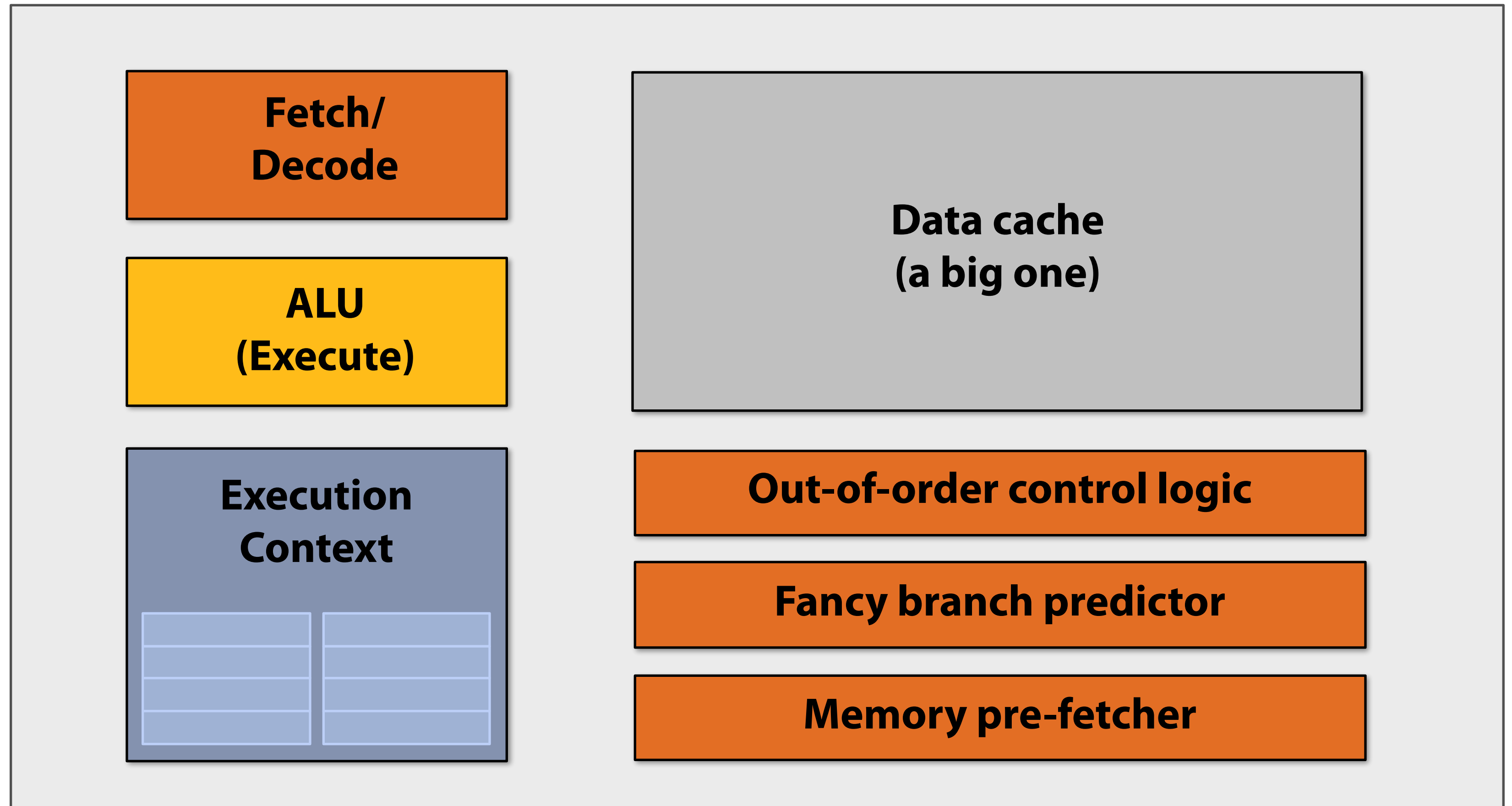
Note: No ILP exists in this region of the program

Aside: Pentium 4



Processor: pre multi-core era

Majority of chip transistors used to perform operations that help a single instruction stream run fast.



More transistors = larger cache, smarter out-of-order logic, smarter branch predictor, etc.

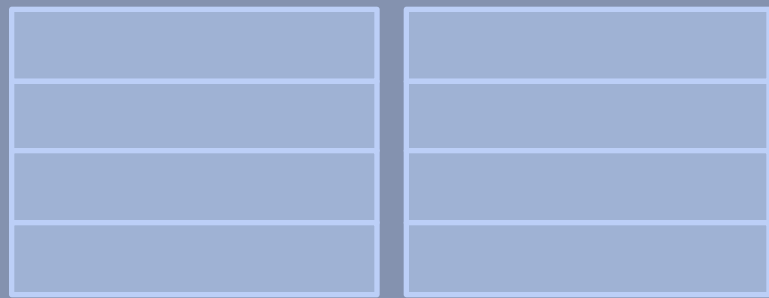
(Also: more transistors → smaller transistors → higher clock frequencies)

Processor: multi-core era

**Fetch/
Decode**

**ALU
(Execute)**

**Execution
Context**

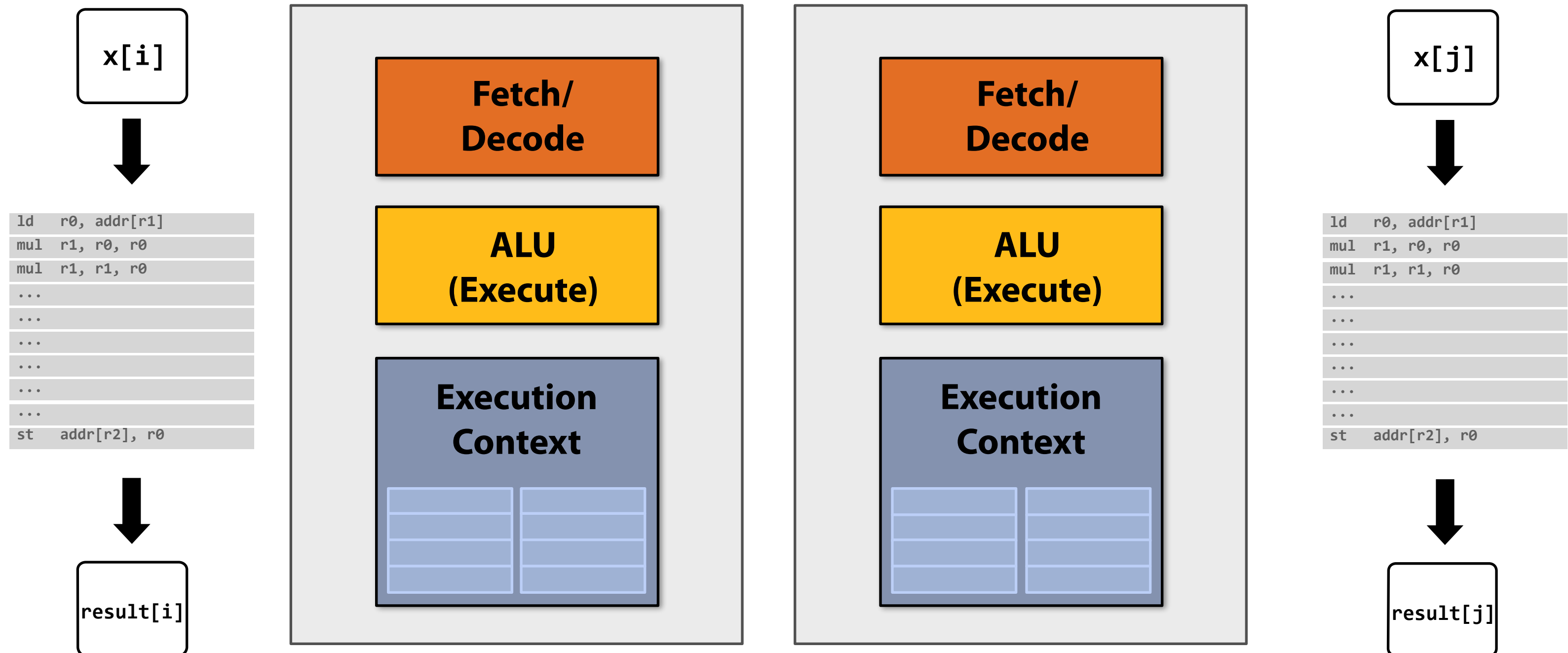


Idea #1:

Use increasing transistor count to add more cores to the processor

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

Two cores: compute two elements in parallel



Simpler cores: each core is slower on a single instruction stream than original “fat” core (e.g., 25% slower)

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

But our program expresses no parallelism

```
void sinx(int N, int terms, float* x, float* result)
{
    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;
        }

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

This program, compiled with gcc will run as one thread on one of the processor cores.

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

Expressing parallelism using pthreads

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

void parallel_sinx(int N, int terms, float* x, float* result)
{
    pthread_t thread_id;
    my_args args;

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

    pthread_create(&thread_id, NULL, my_thread_start, &args); // launch thread
    sinx(N - args.N, terms, x + args.N, result + args.N); // do work
    pthread_join(thread_id, NULL);
}

void my_thread_start(void* thread_arg)
{
    my_args* thread_args = (my_args*)thread_arg;
    sinx(thread_args->N, thread_args->terms, thread_args->x, thread_args->result); // do work
}
```

```
void sinx(int N, int terms, float* x, float* result)
{
    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;
        }

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

Data-parallel expression

(in Kayvon's fictitious data-parallel language)

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

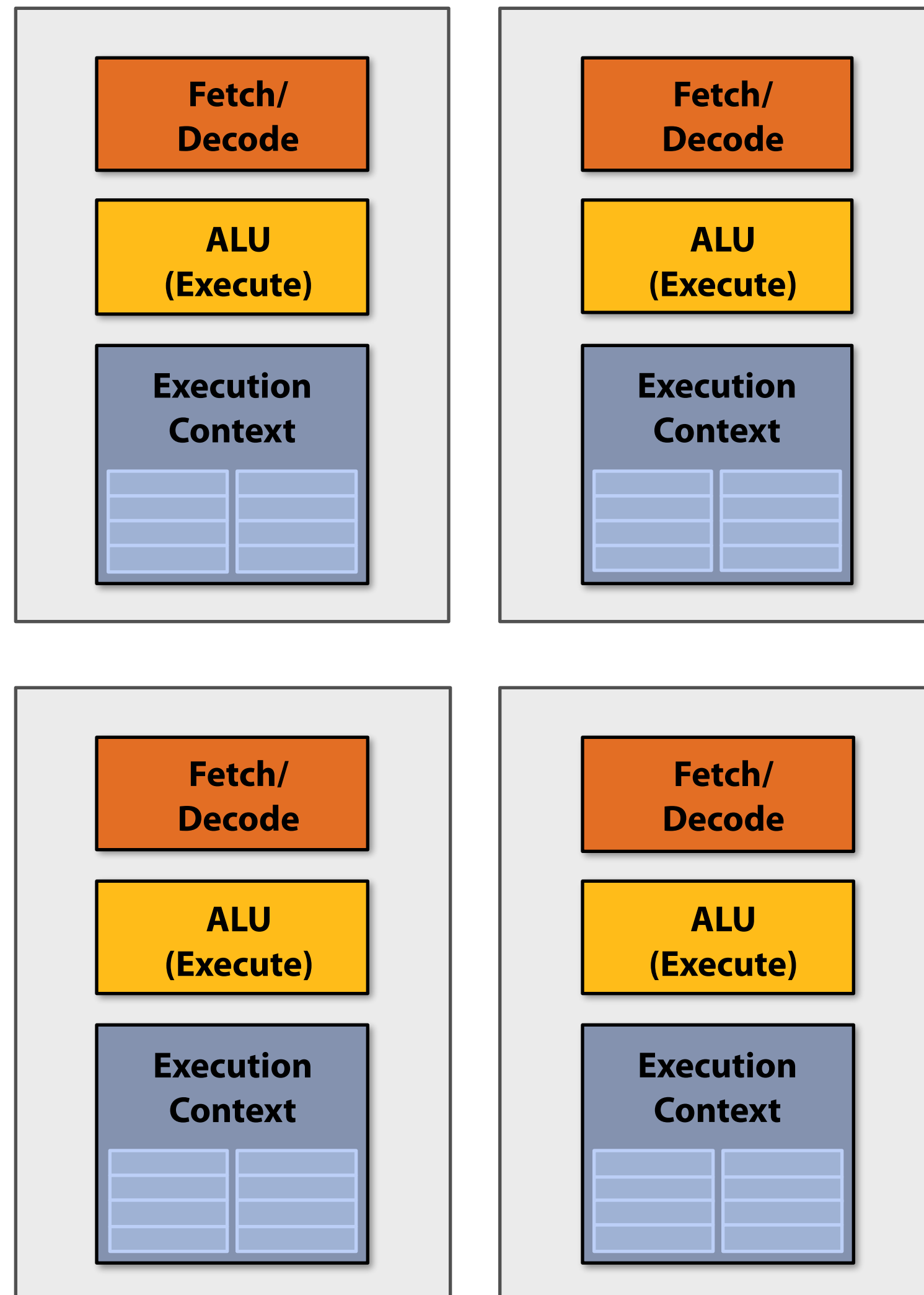
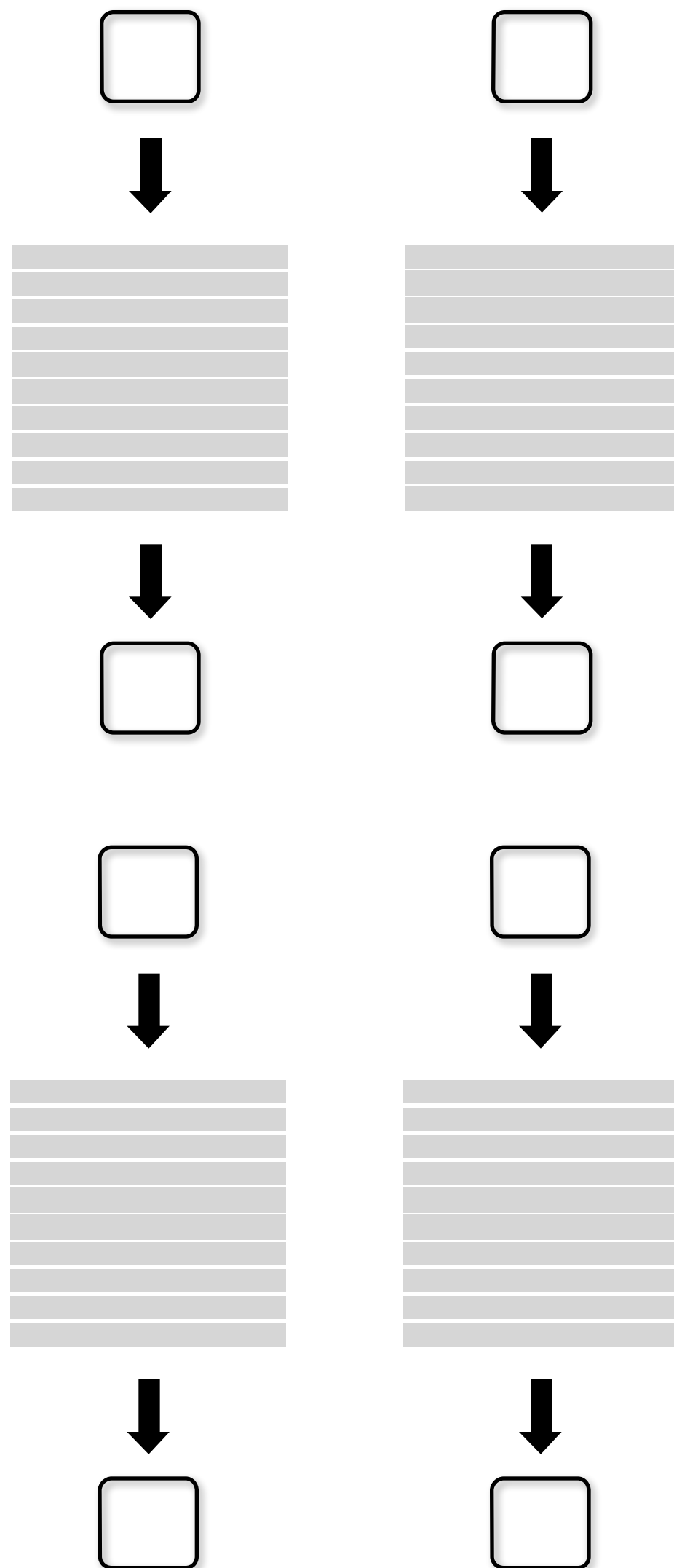
        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

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

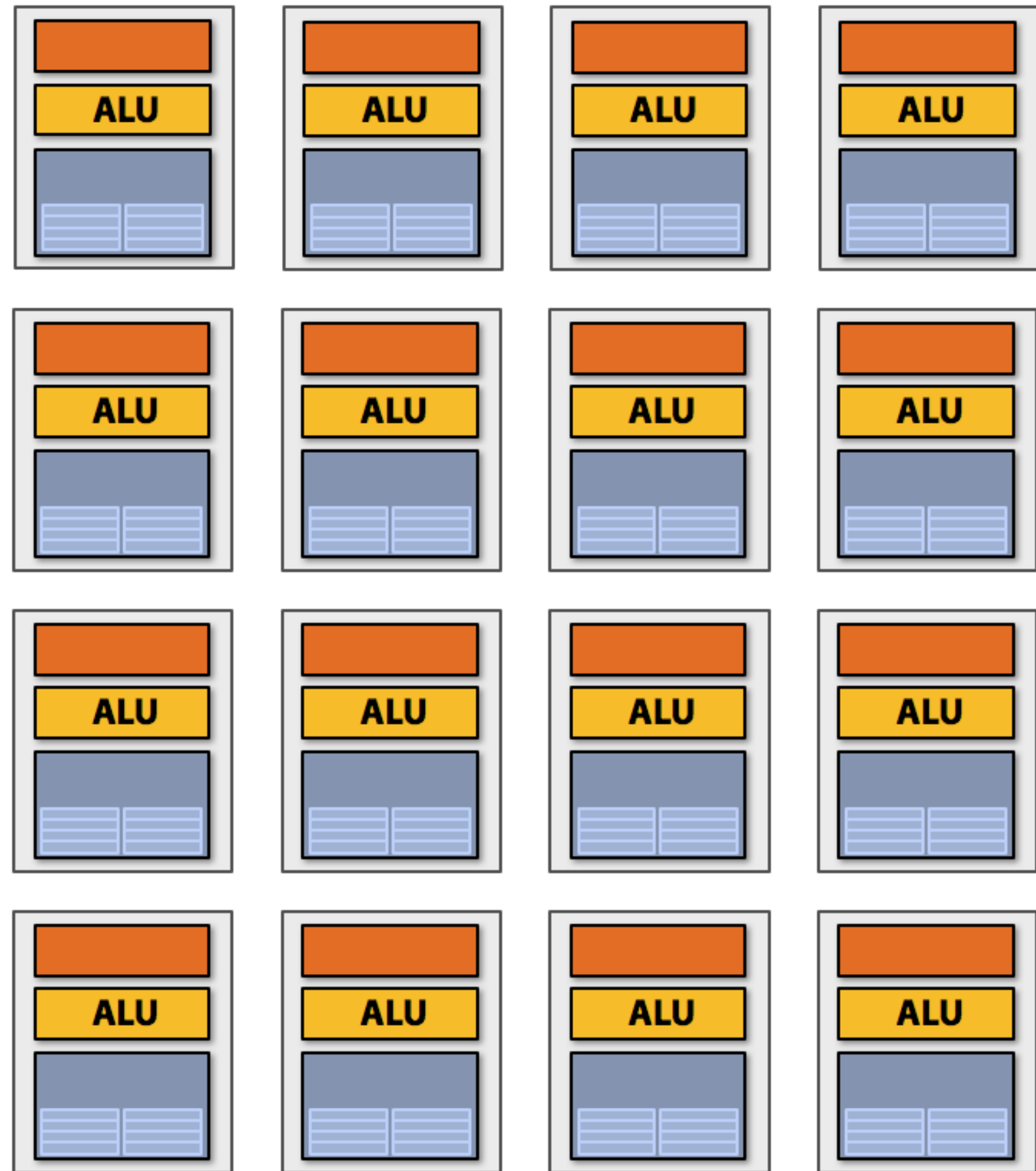
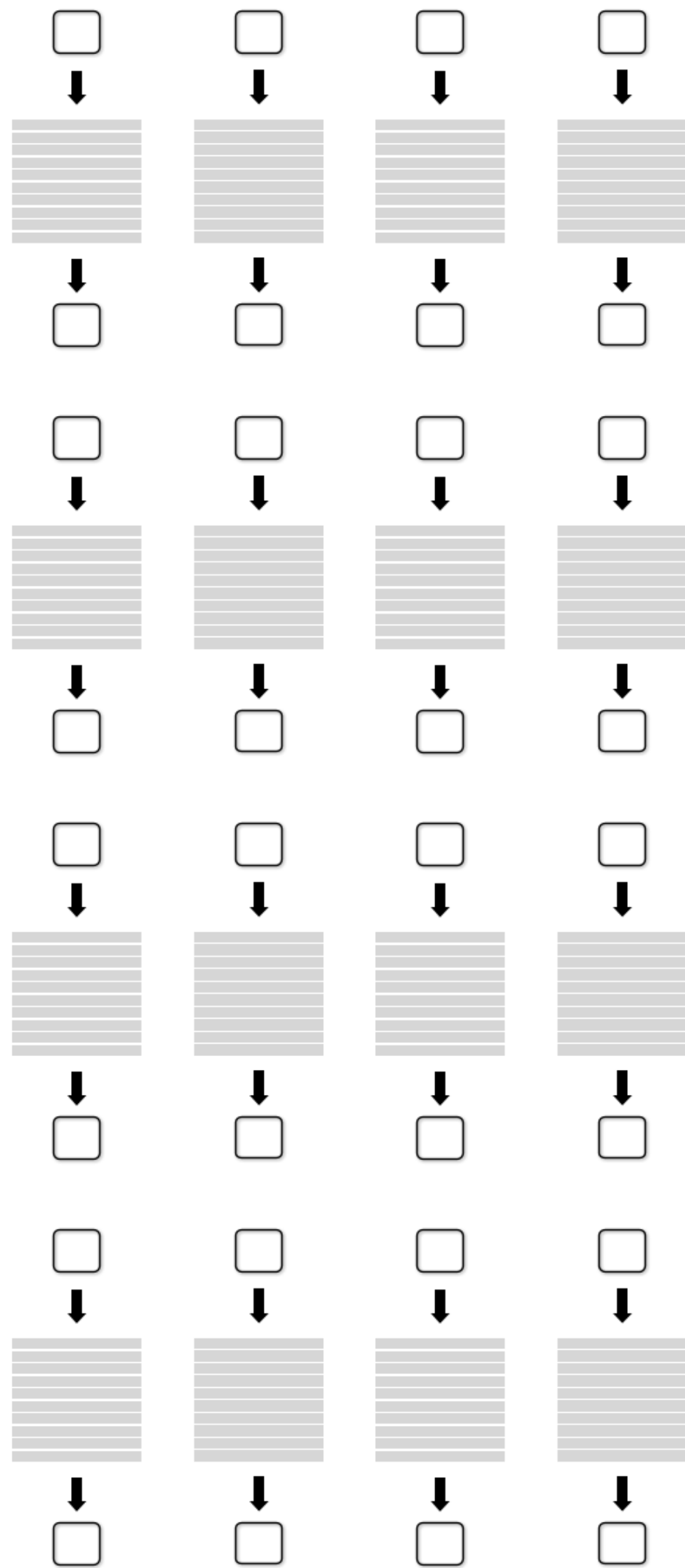
Parallelism visible to compiler
(loop iterations are known to be independent)

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

Four cores: compute four elements in parallel

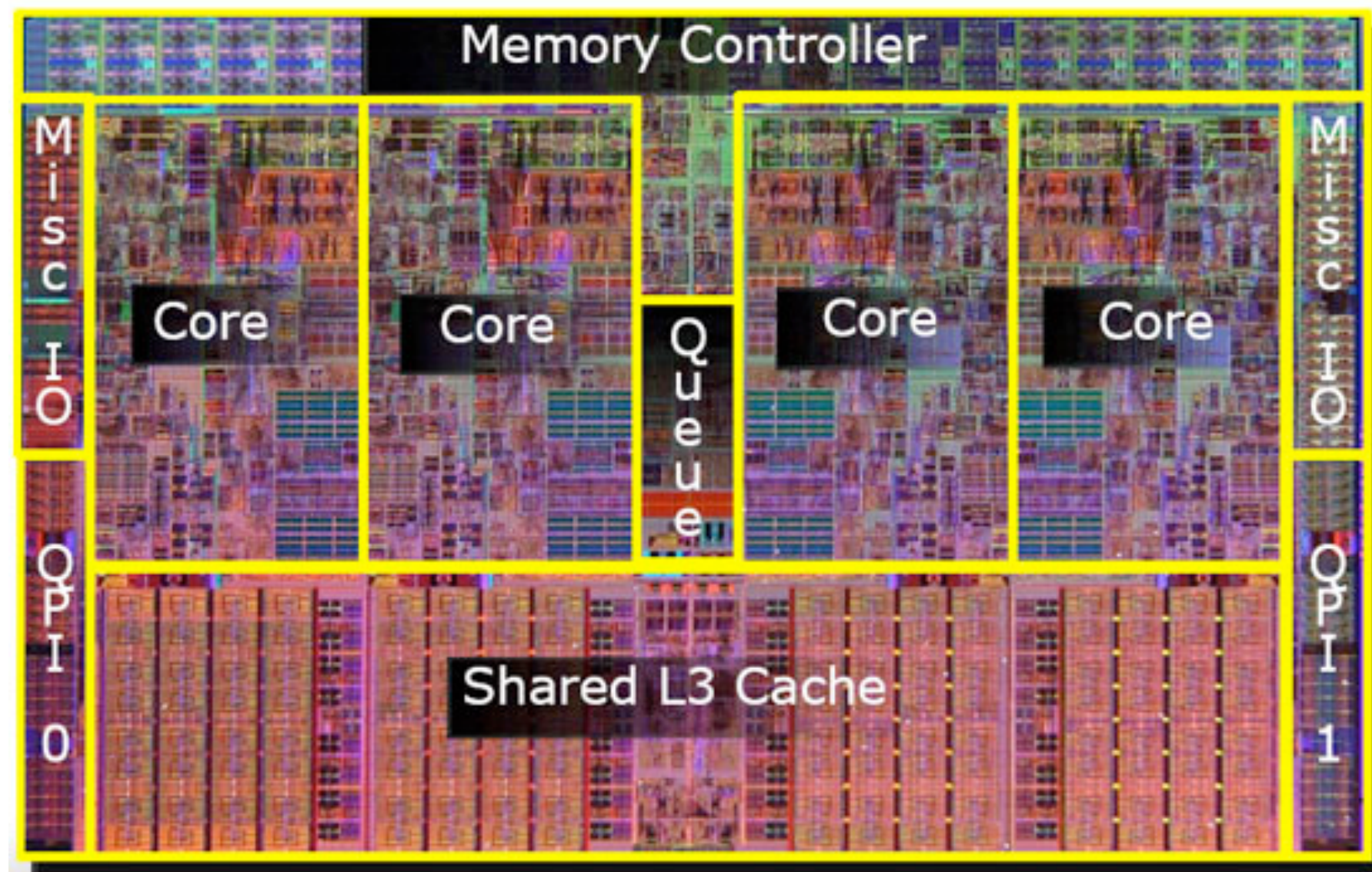


Sixteen cores: compute sixteen elements in parallel

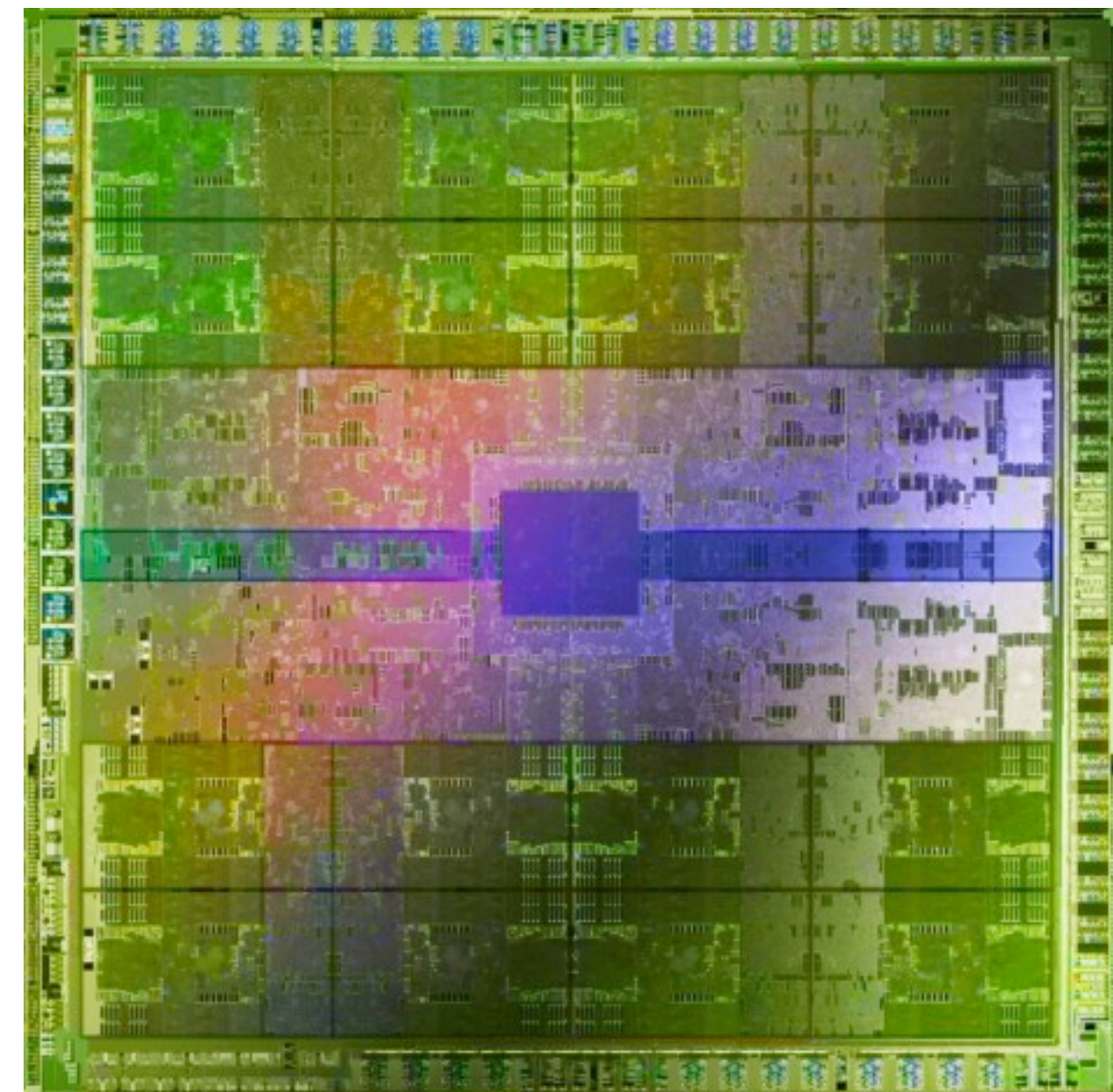


Sixteen cores, sixteen simultaneous instruction streams

Multi-core examples

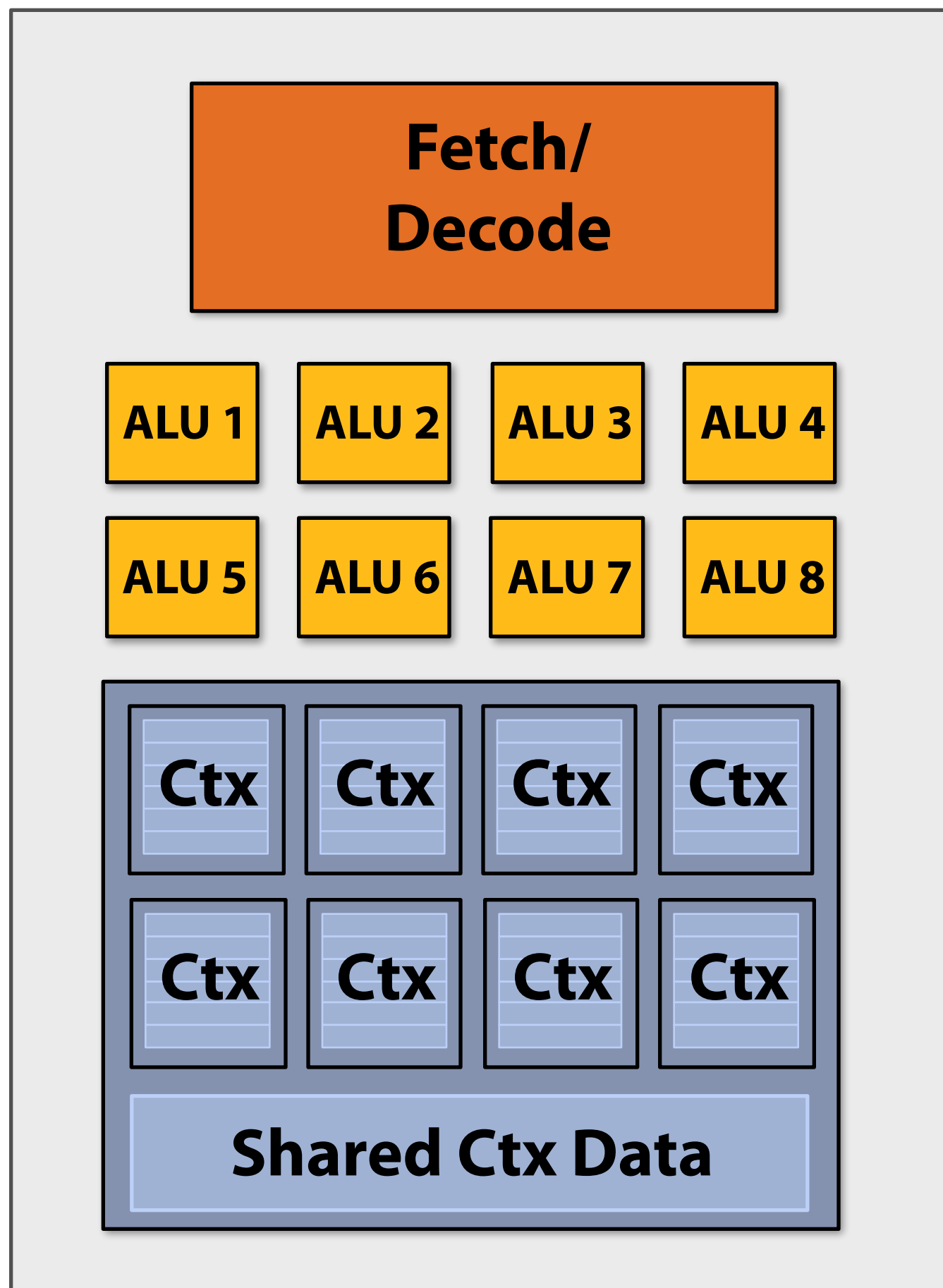


Intel Core i7 quad-core CPU (2010)



NVIDIA Tesla GPU (2009)
16 cores

Add ALUs to increase compute capability



Idea #2:

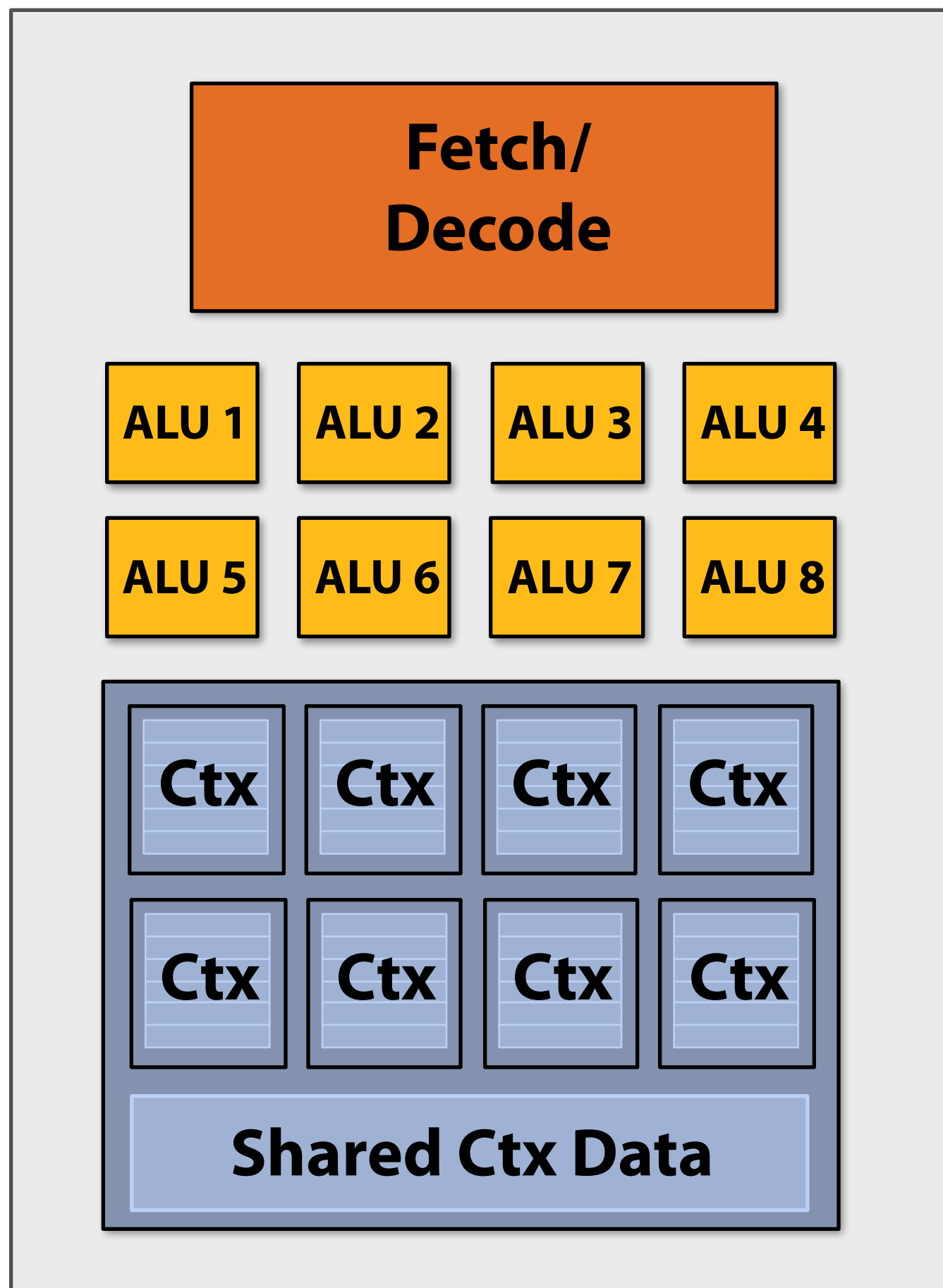
Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

Single instruction, multiple data

**Same instruction broadcast to all ALUs
Executed in parallel on all ALUs**

Add ALUs to increase compute capability



```
ld    r0, addr[r1]
```

```
mul   r1, r0, r0
```

```
mul   r1, r1, r0
```

```
...
```

```
...
```

```
...
```

```
...
```

```
...
```

```
...
```

```
st    addr[r2], r0
```

Recall original compiled program:

Instruction stream processes one array element at a time using scalar instructions on scalar registers (e.g., 32-bit floats)

Scalar program

```
void sinx(int N, int terms, float* x, float* result)
{
    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;
        }

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

Original compiled program:

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

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

Vector program (using AVX intrinsics)

Intrinsics available to C programmers

```
#include <immintrin.h>

void sinx(int N, int terms, float* x, float* result)
{
    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(&result[i], value);
    }
}
```

Vector program (using AVX intrinsics)

```
#include <immintrin.h>

void sinx(int N, int terms, float* x, float* sinx)
{
    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_broadcast_ss(sign), numer), denom);
            value = _mm256_add_ps(value, tmp);

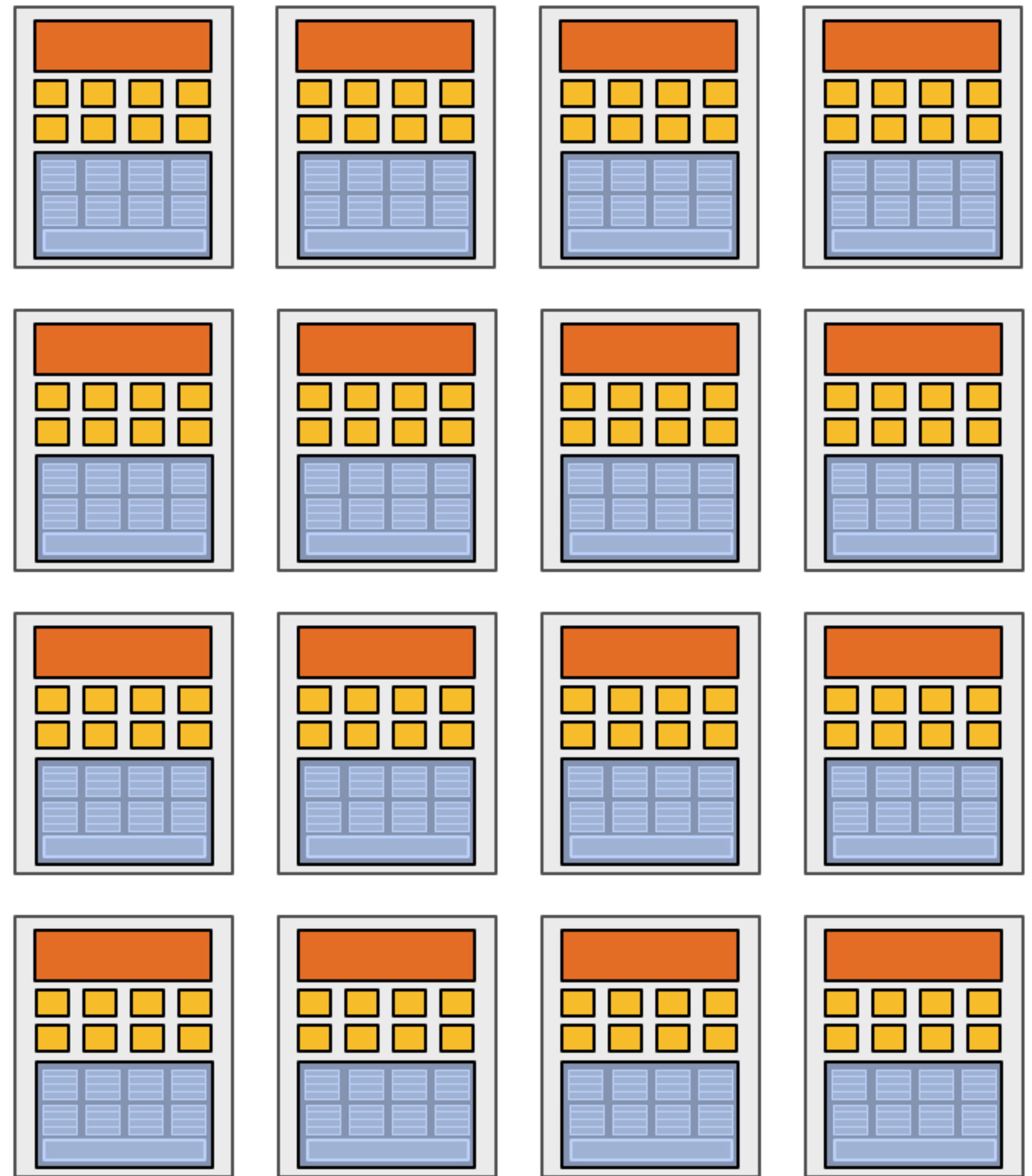
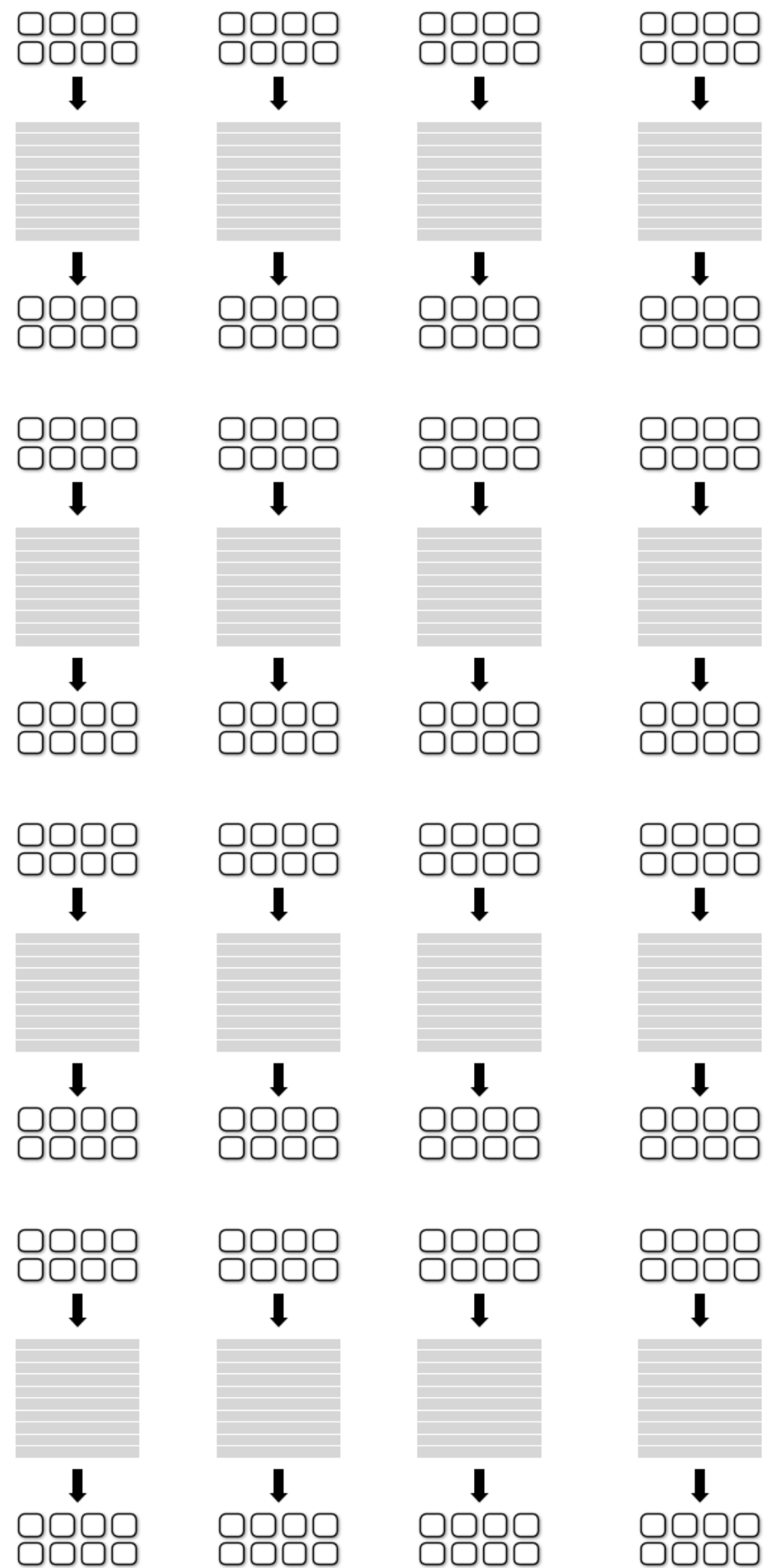
            numer = _mm256_mul_ps(numer, _mm256_mul_ps(origx, origx));
            denom = _mm256_mul_ps(denom, _mm256_broadcast_ss((2*j+2) * (2*j+3)));
            sign *= -1;
        }
        _mm256_store_ps(&sinx[i], value);
    }
}
```

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

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 data-parallel language)

```
void sinx(int N, int terms, float* x, float* result)
{
    // declare independent loop iterations
    forall (int i from 0 to N-1)
    {
        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;
        }

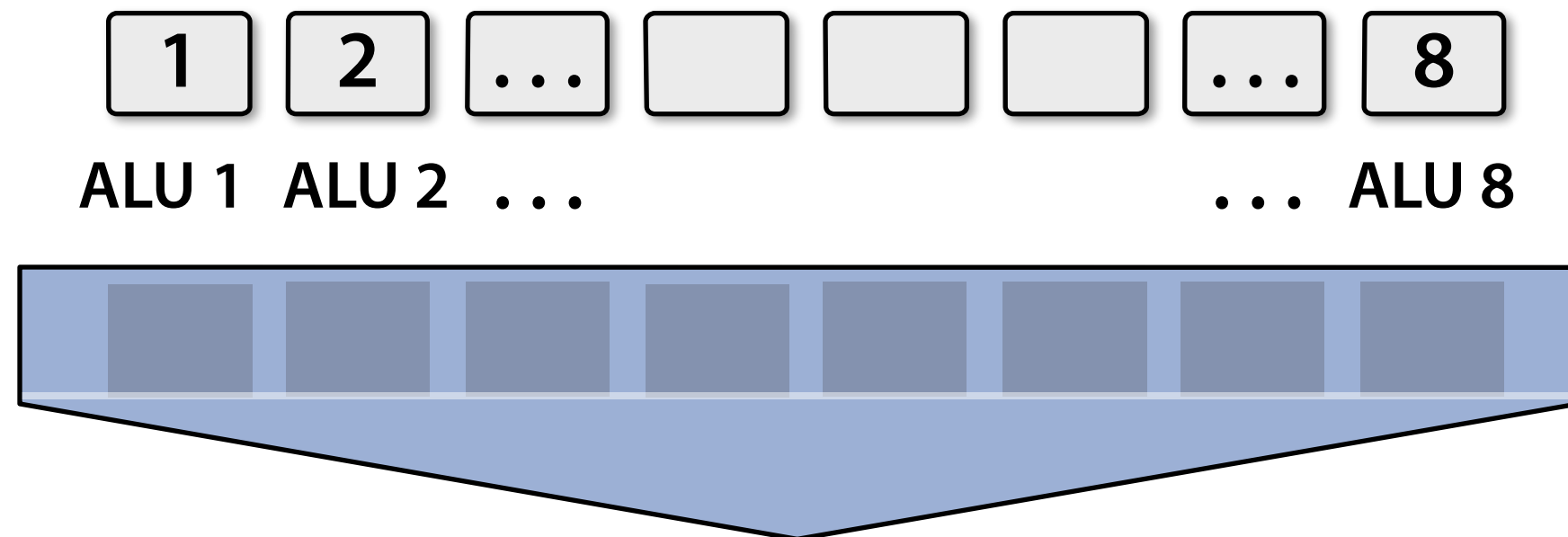
        result[i] = value;
    }
}
```

Parallelism visible to compiler

Facilitates automatic generation of vector instructions

What about conditional execution?

Time (clocks)



(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

<unconditional code>

```
float x = A[i];  
  
if (x > 0) {  
    float tmp = exp(x, 5.f);  
    tmp *= kMyConst1;  
    x = tmp + kMyConst2;  
} else {  
    float tmp = kMyConst1;  
    x = 2.f * tmp;  
}
```

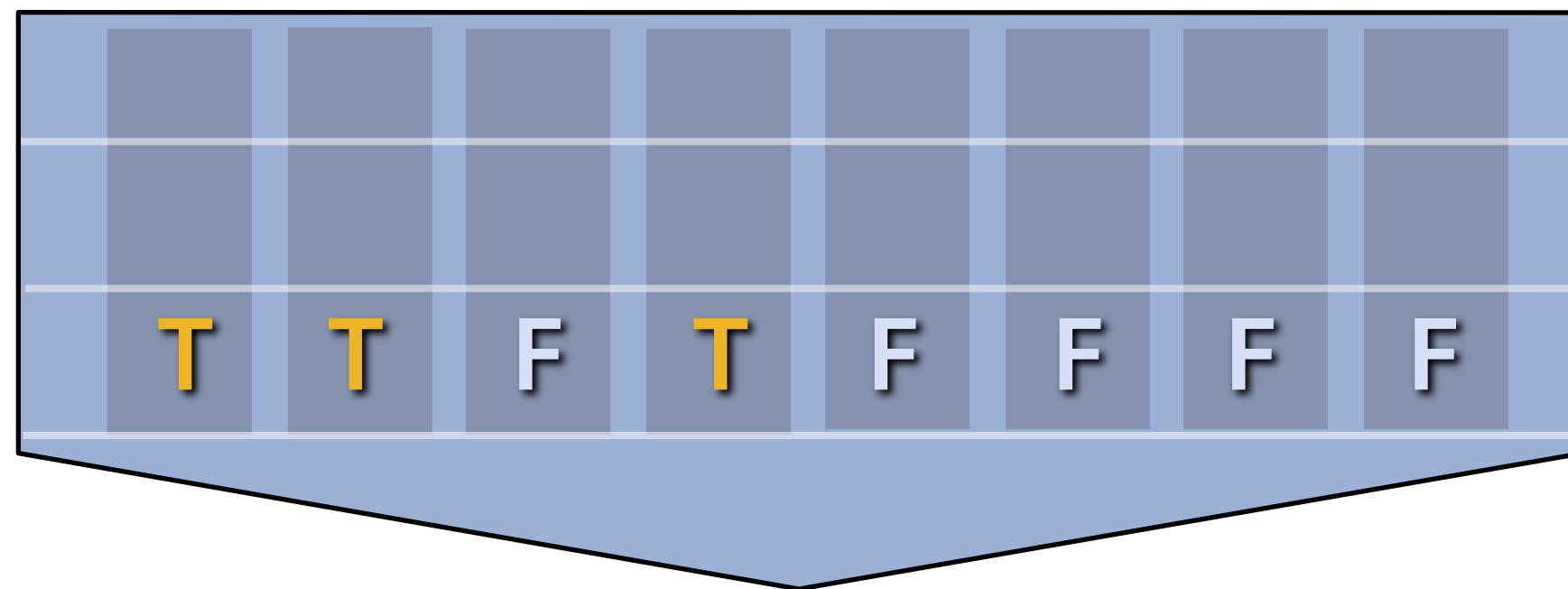
<resume unconditional code>

```
result[i] = x;
```

What about conditional execution?

Time (clocks)

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



(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

<unconditional code>

```
float x = A[i];
```

```
if (x > 0) {
```

```
    float tmp = exp(x, 5.f);
```

```
    tmp *= kMyConst1;
```

```
    x = tmp + kMyConst2;
```

```
} else {
```

```
    float tmp = kMyConst1;
```

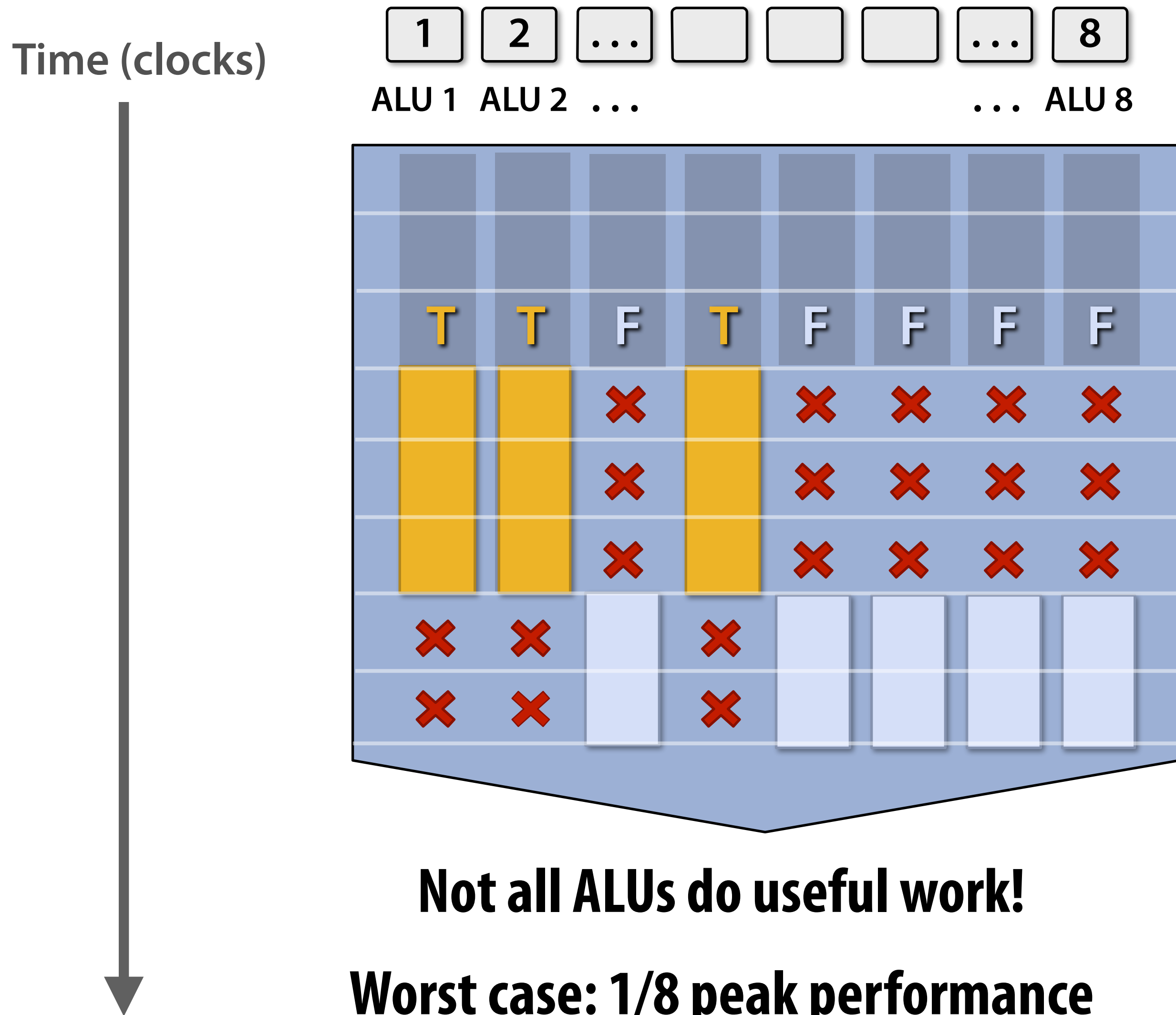
```
    x = 2.f * tmp;
```

```
}
```

<resume unconditional code>

```
result[i] = x;
```

Mask (discard) output of ALU



(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

<unconditional code>

```
float x = A[i];
```

```
if (x > 0) {
```

```
    float tmp = exp(x, 5.f);
```

```
    tmp *= kMyConst1;
```

```
    x = tmp + kMyConst2;
```

```
} else {
```

```
    float tmp = kMyConst1;
```

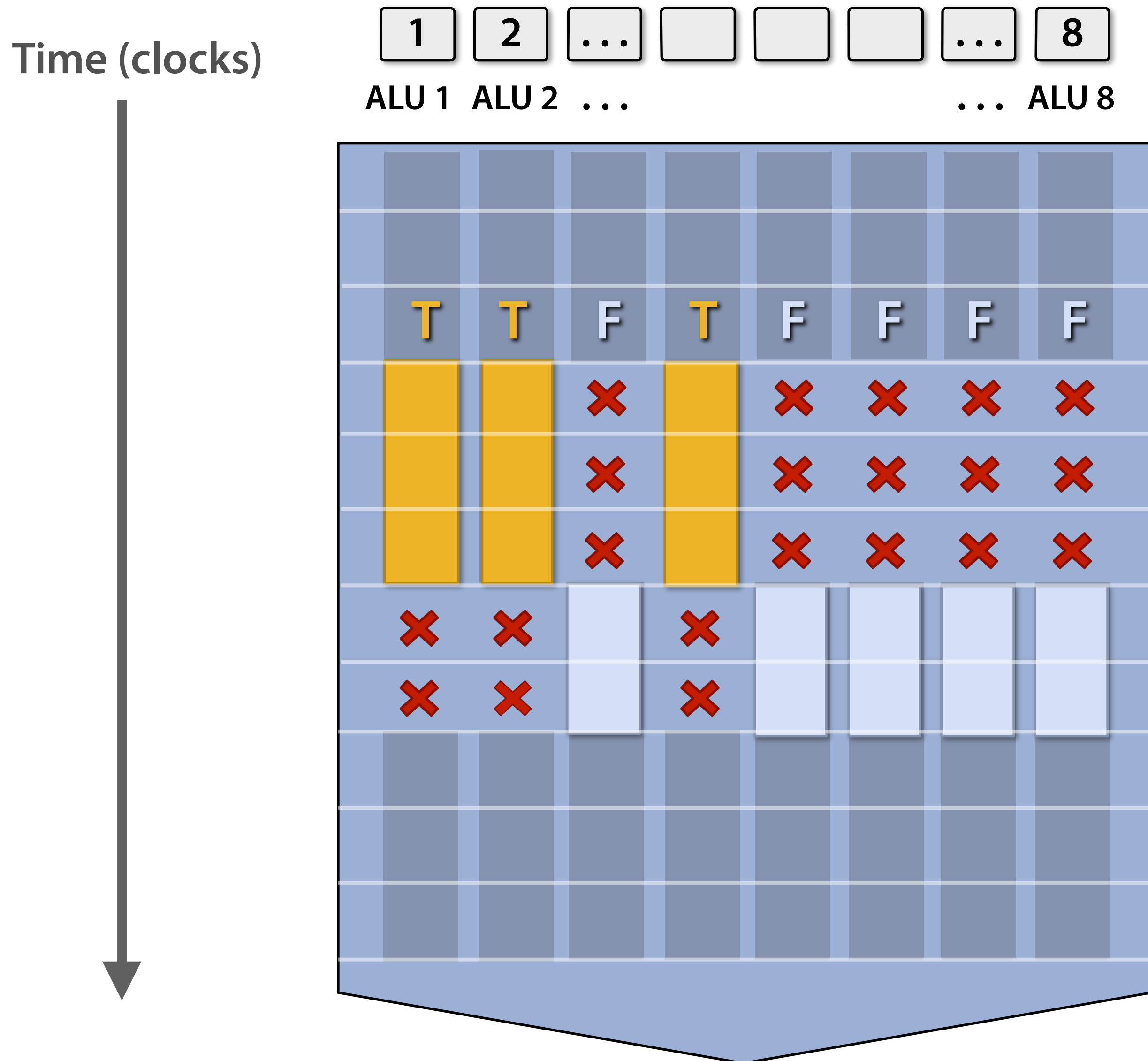
```
    x = 2.f * tmp;
```

```
}
```

<resume unconditional code>

```
result[i] = x;
```


After branch: continue at full performance



(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

<unconditional code>

```
float x = A[i];
```

```
if (x > 0) {
```

```
    float tmp = exp(x, 5.f);
```

```
    tmp *= kMyConst1;
```

```
    x = tmp + kMyConst2;
```

```
} else {
```

```
    float tmp = kMyConst1;
```

```
    x = 2.f * tmp;
```

```
}
```

<resume unconditional code>

```
result[i] = x;
```

Terminology

- **Instruction stream coherence (“coherent execution”)**
 - Same instruction sequence applies to all elements operated upon simultaneously
 - Coherent execution is necessary for efficient use of SIMD processing resources
 - Coherent execution is NOT necessary for efficient parallelization across cores, since each core has the capability to fetch/decode a different instruction stream
- **“Divergent” execution**
 - A lack of instruction stream coherence
- **Note: don’t confuse instruction stream coherence with “cache coherence” (a major topic later in the course)**

SIMD execution on modern CPUs

- **SSE instructions: 128-bit operations: 4x32 bits or 2x64 bits (4-wide float vectors)**
- **AVX instructions: 256 bit operations: 8x32 bits or 4x64 bits (8-wide float vectors)**
- **Instructions are generated by compiler**
 - **Parallelism explicitly given by programmer using intrinsics**
 - **Parallelism conveyed using parallel language semantics (e.g., `forall` example)**
 - **Parallelism inferred by dependency analysis of loops (hard problem, even best compilers are not great on arbitrary C/C++ code)**
- **“Explicit SIMD”: SIMD parallelization is performed at compile time**
 - **Can inspect program binary and see instructions (`vstoreps`, `vmulps`, etc.)**

SIMD execution on many modern GPUs

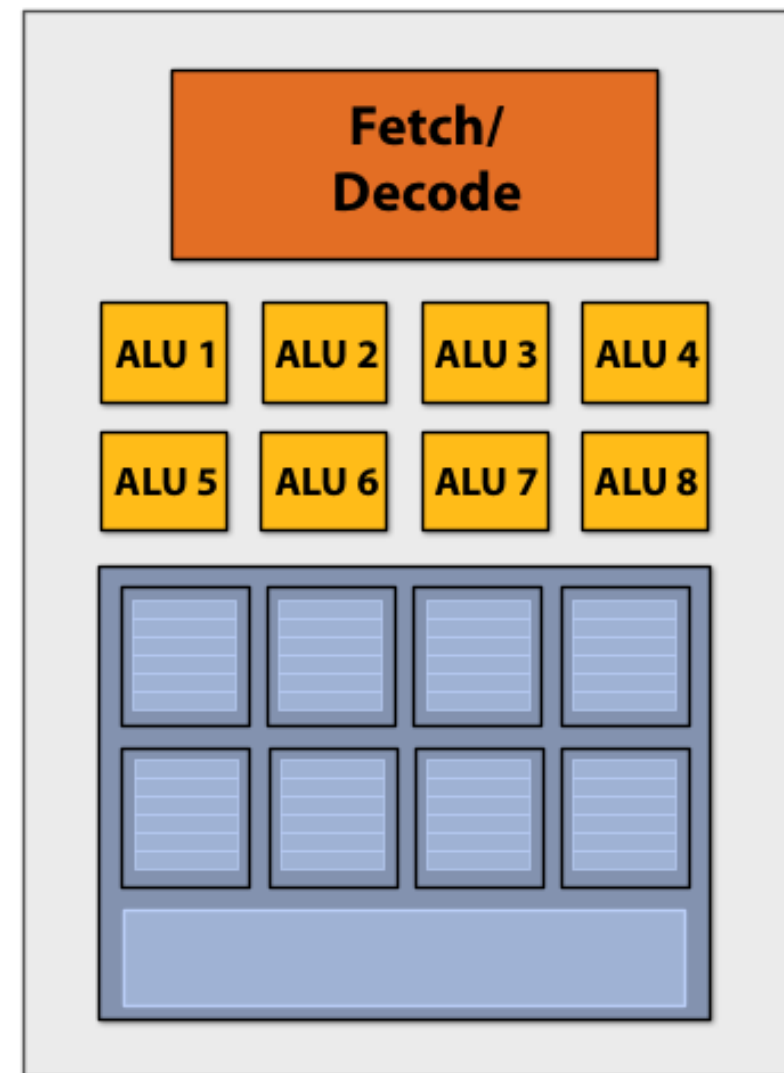
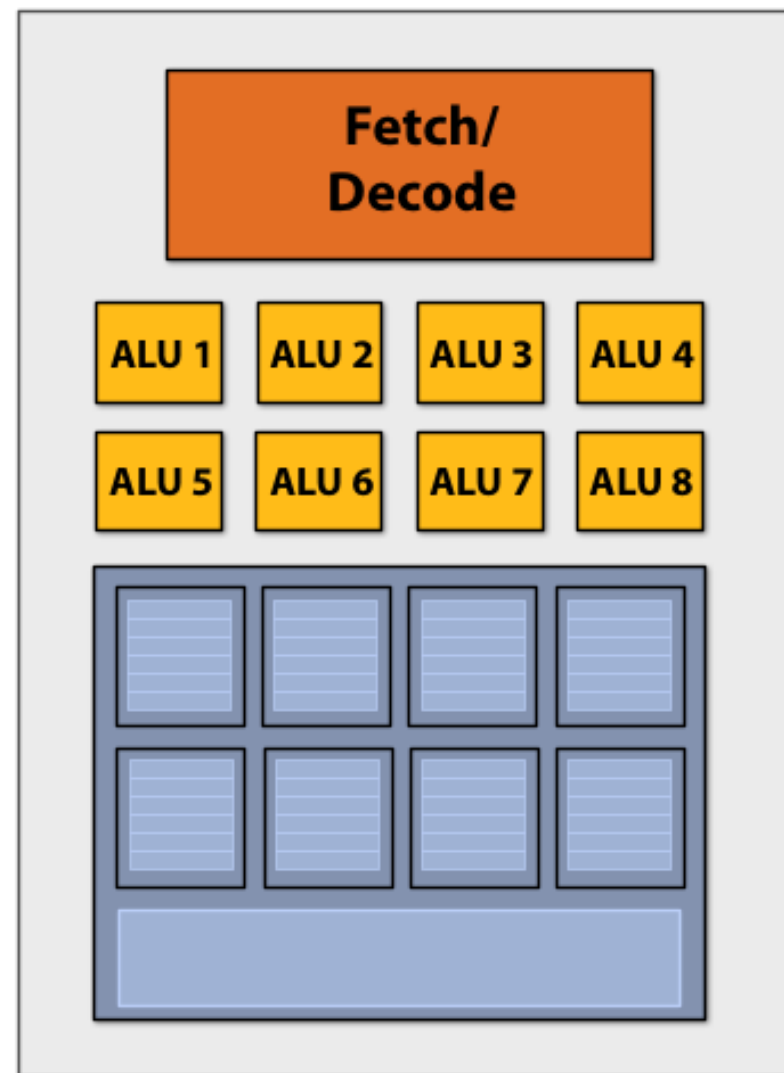
- **“Implicit SIMD”**

- **Compiler generates a scalar binary**
- **But N instances of the program are **always run** together:**
`execute(my_function, N) // execute my_function N times`
- **Hardware maps instances to ALUs**

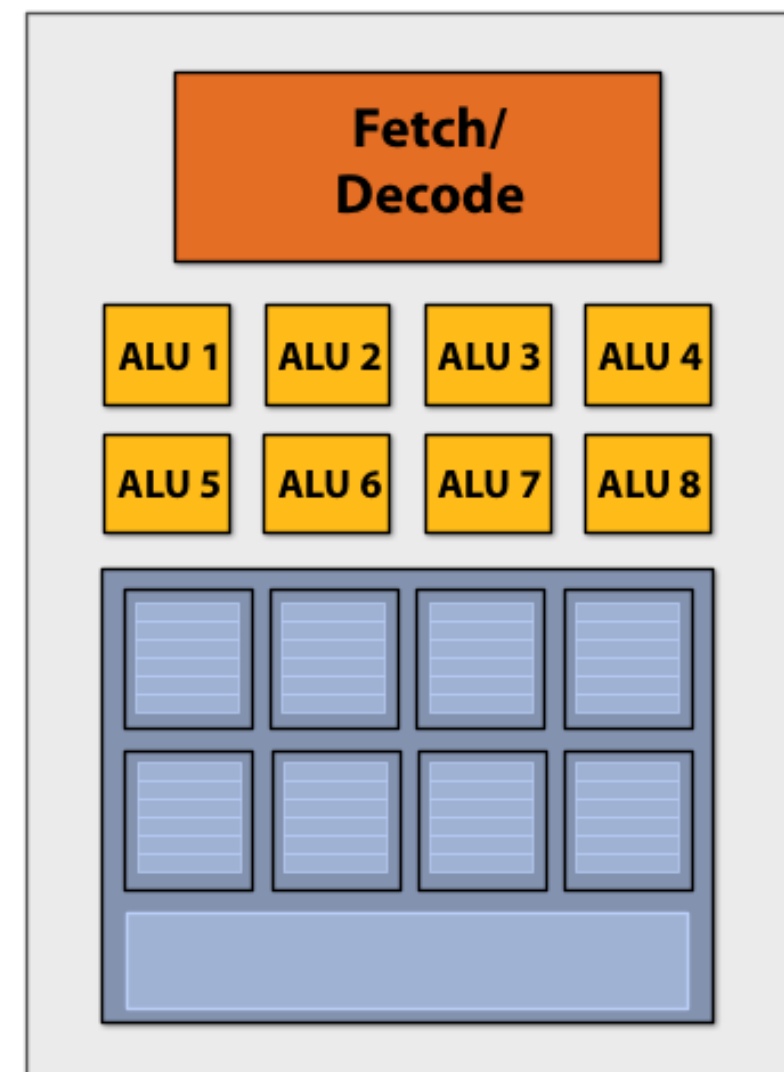
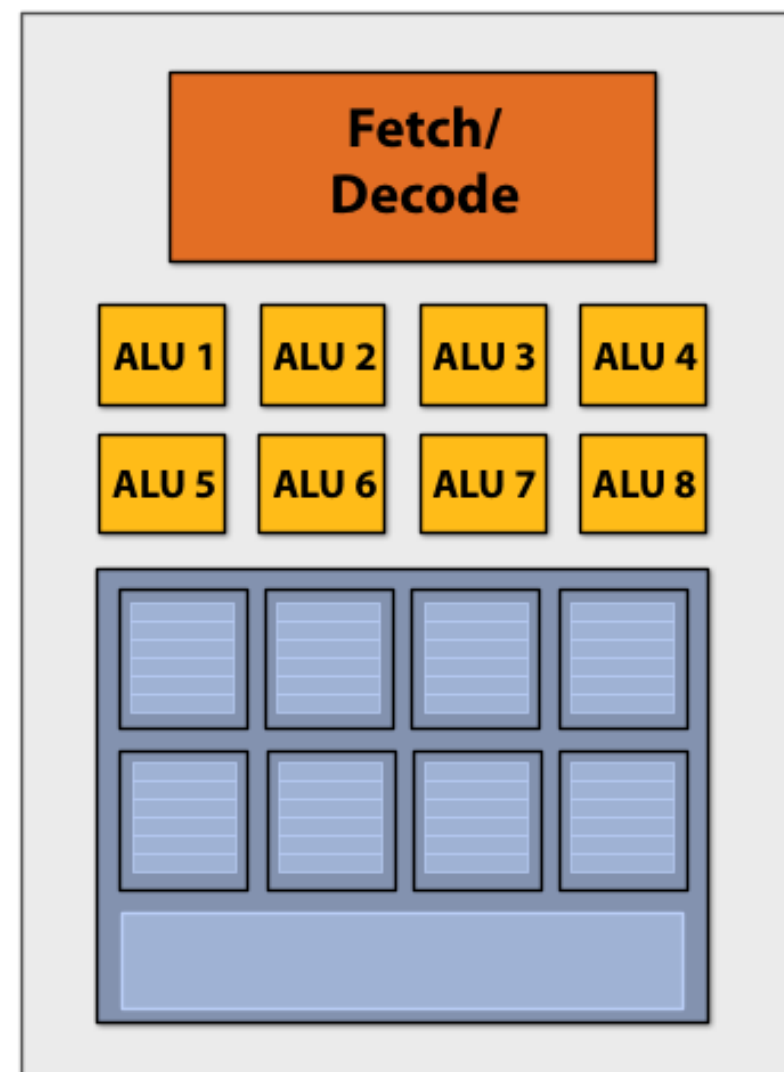
- **SIMD width of most modern GPUs is 16 to 32**

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

Example: Intel Core i7 (Sandy Bridge)



4 cores
8 SIMD ALUs per core



On campus:

New GHC machines:

4 cores

8 SIMD ALUs per core

Machines in GHC 5207:

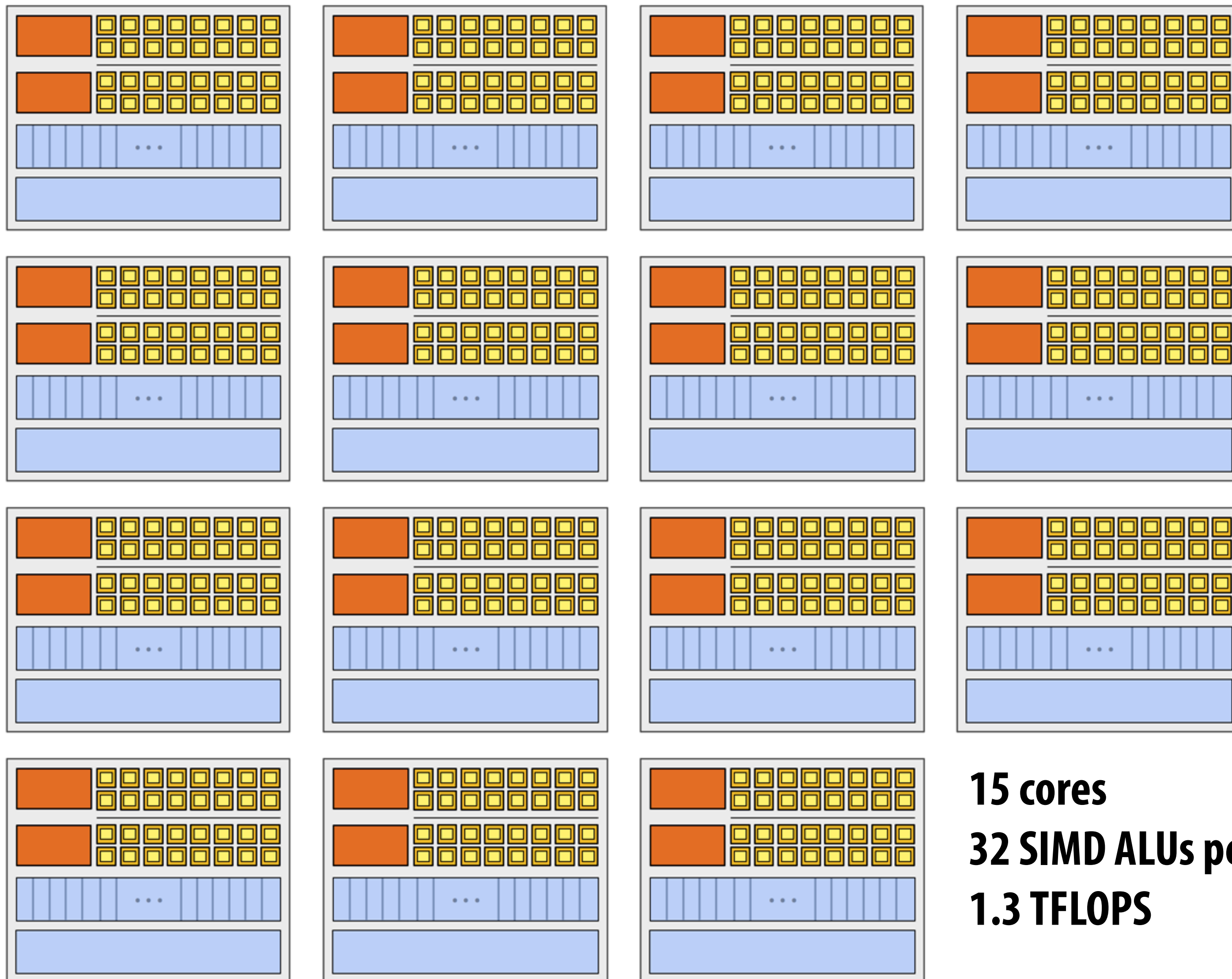
(old GHC 3000 machines)

6 cores

4 SIMD ALUs per core

Example: NVIDIA GTX 480

(in the Gates lab)



15 cores
32 SIMD ALUs per core
1.3 TFLOPS

Summary: parallel execution

■ Several types of parallel execution in modern processors

- **Multi-core: use multiple processing cores**
 - **Provides thread-level parallelism: simultaneously execute a completely different instruction stream on each core**
 - **Software decides when to create threads (e.g., via pthreads API)**
- **SIMD: use multiple ALUs controlled by same instruction stream (within a core)**
 - **Efficient: control amortized over many ALUs**
 - **Vectorization can be done by compiler (explicit SIMD) or at runtime by hardware**
 - **[Lack of] dependencies is known prior to execution (declared or inferred)**
- **Superscalar: exploit ILP. Process different instructions from the same instruction stream in parallel (within a core)**
 - **Parallelism automatically and dynamically discovered by the hardware during execution (not programmer visible)**

Not addressed
further in this
class

Part 2: accessing memory

Terminology

■ Memory latency

- The amount of time for a memory request (e.g., load, store) from a processor to be serviced by the memory system
- Example: 100 cycles, 100 nsec

■ Memory bandwidth


- The rate at which the memory system can provide data to a processor
- Example: 20 GB/s

Stalls

- A processor “stalls” when it cannot run the next instruction in an instruction stream because of a dependency on a previous instruction.

- Accessing memory is a major source of stalls

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

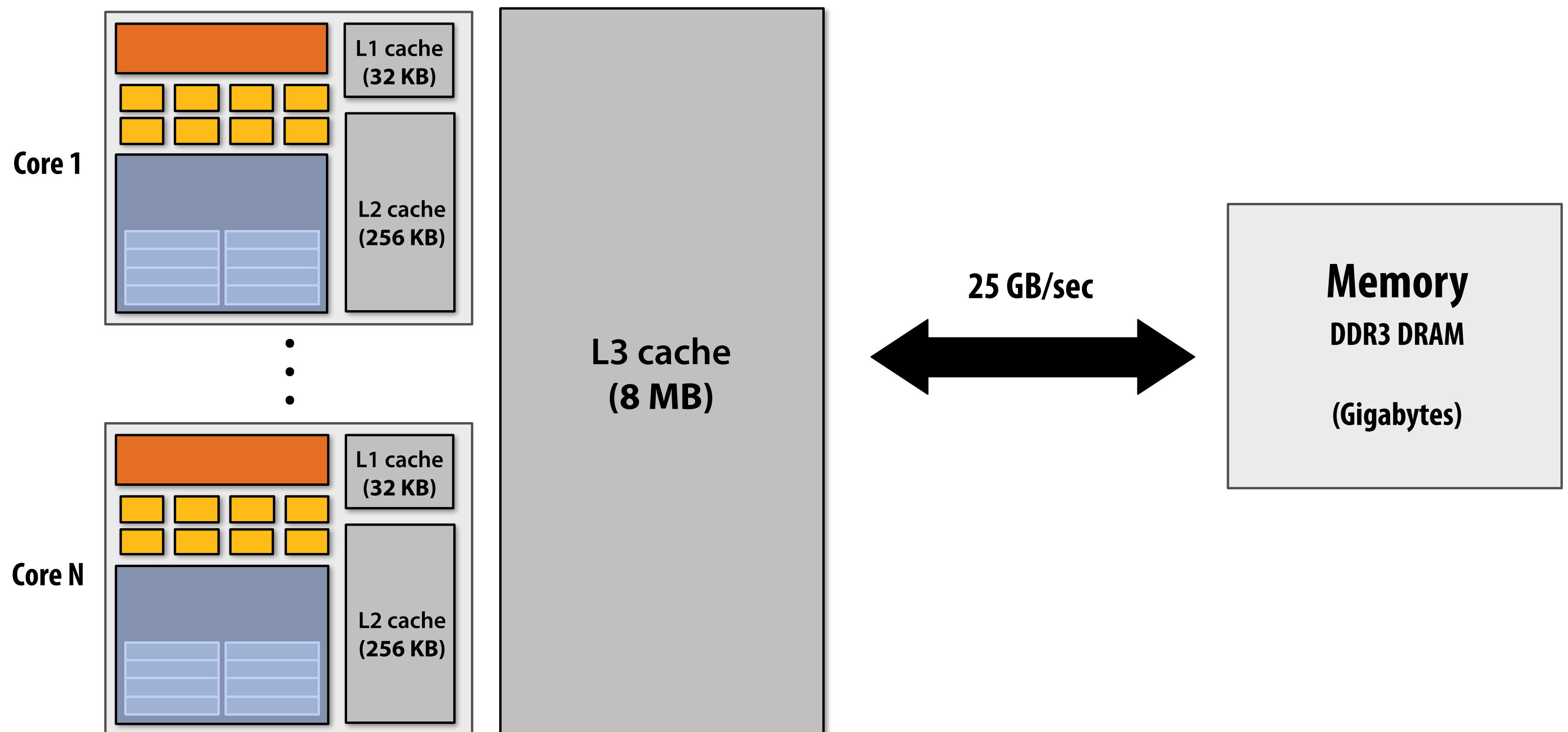


The diagram illustrates a data hazard dependency. Three instructions are listed: `ld r0 mem[r2]`, `ld r1 mem[r3]`, and `add r0, r0, r1`. Red arrows point from the `ld r0 mem[r2]` and `ld r1 mem[r3]` instructions to the `add r0, r0, r1` instruction, indicating that the `add` instruction depends on the values loaded into `r0` and `r1` by the previous instructions. The word "Dependency" is written in red to the right of the arrows.

- Memory access times ~ 100's of cycles
 - LATENCY: the time it takes to complete an operation

Caches reduce length of stalls (reduce latency)

Processors run efficiently when data is resident in caches
(caches reduce memory access latency, also provide high bandwidth to CPU)



Prefetching reduces stalls (hides latency)

- **All modern CPUs have logic for prefetching data into caches**
 - Dynamically analyze program's access patterns, predict what it will access soon
- **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]
ld r1, mem[r3]
add r0, r0, r1
```

These loads are cache hits

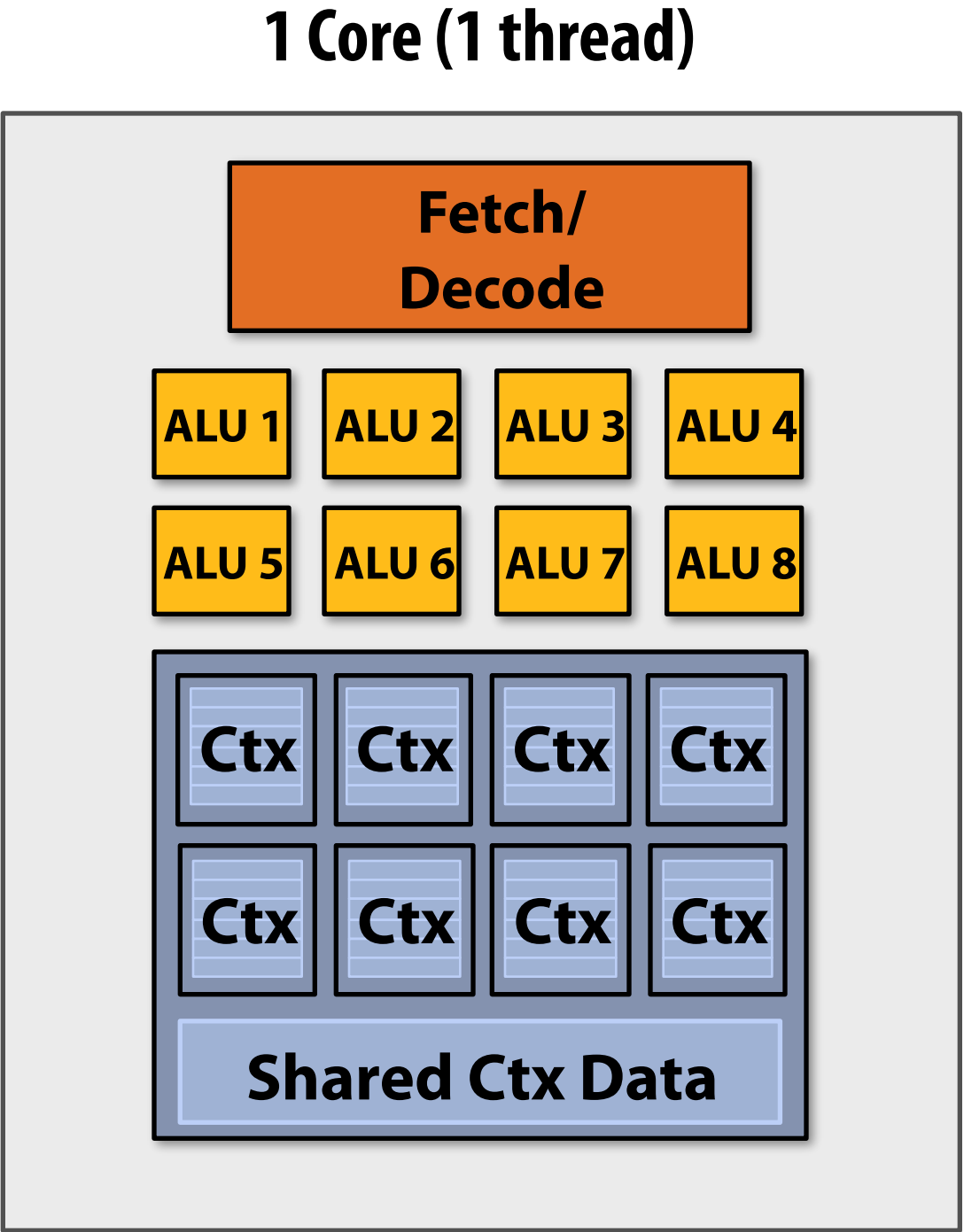
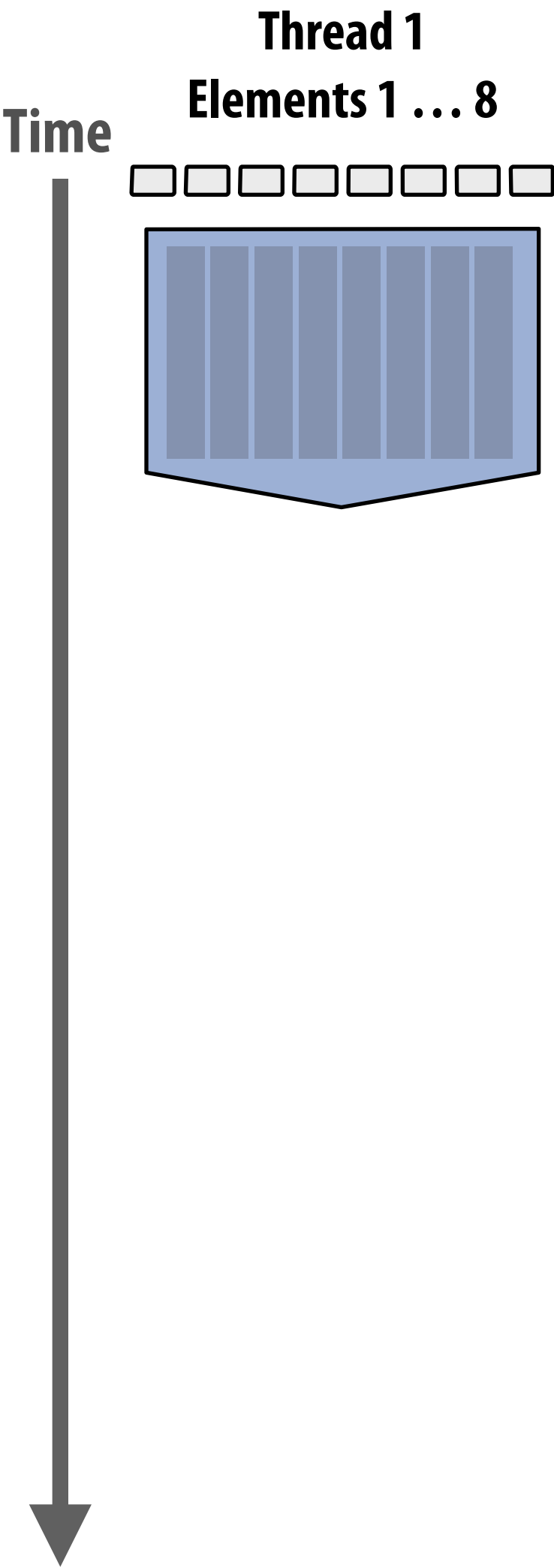
Note: Prefetching can also reduce performance if the guess is wrong (hogs bandwidth, pollutes caches)

(more detail later in course)

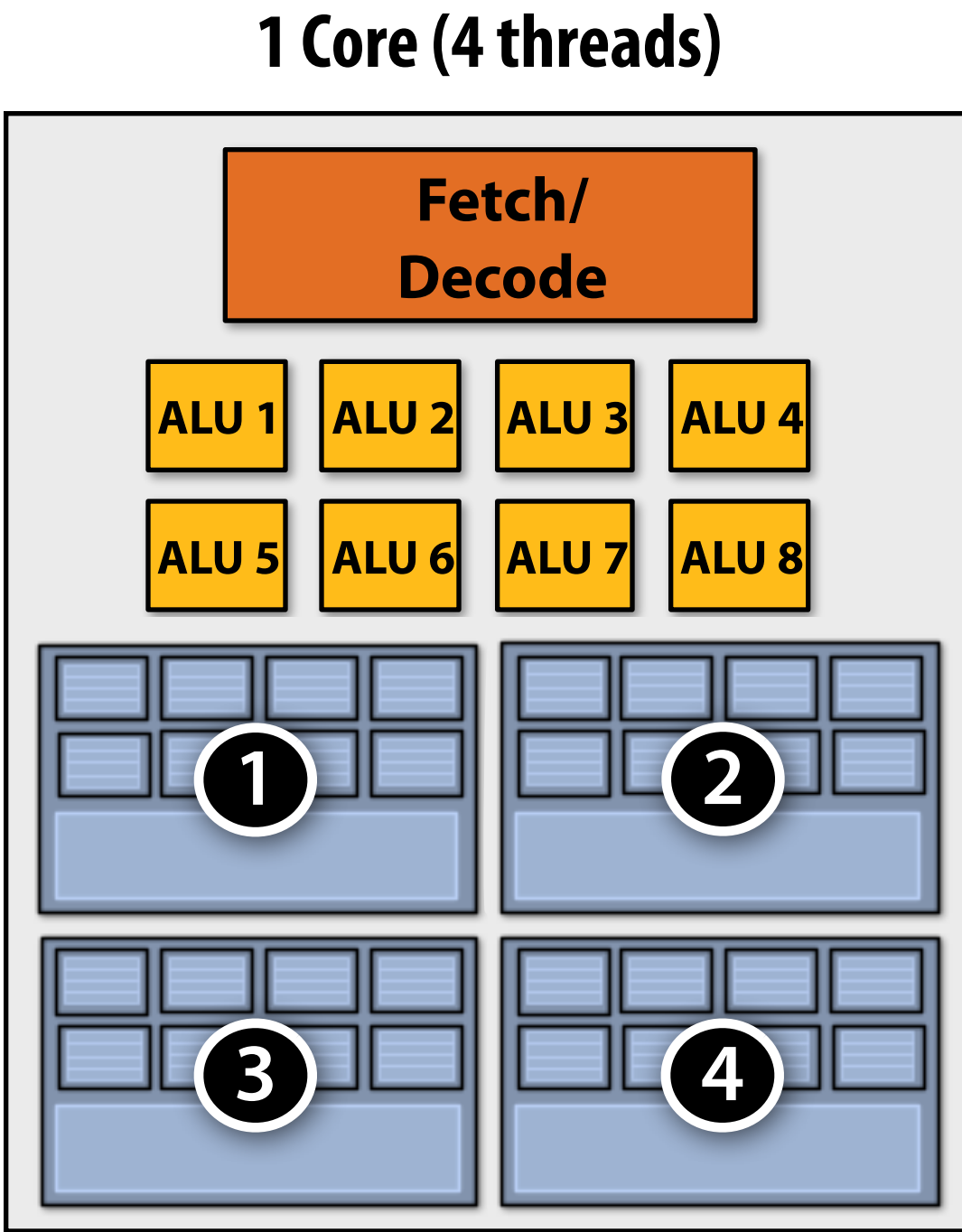
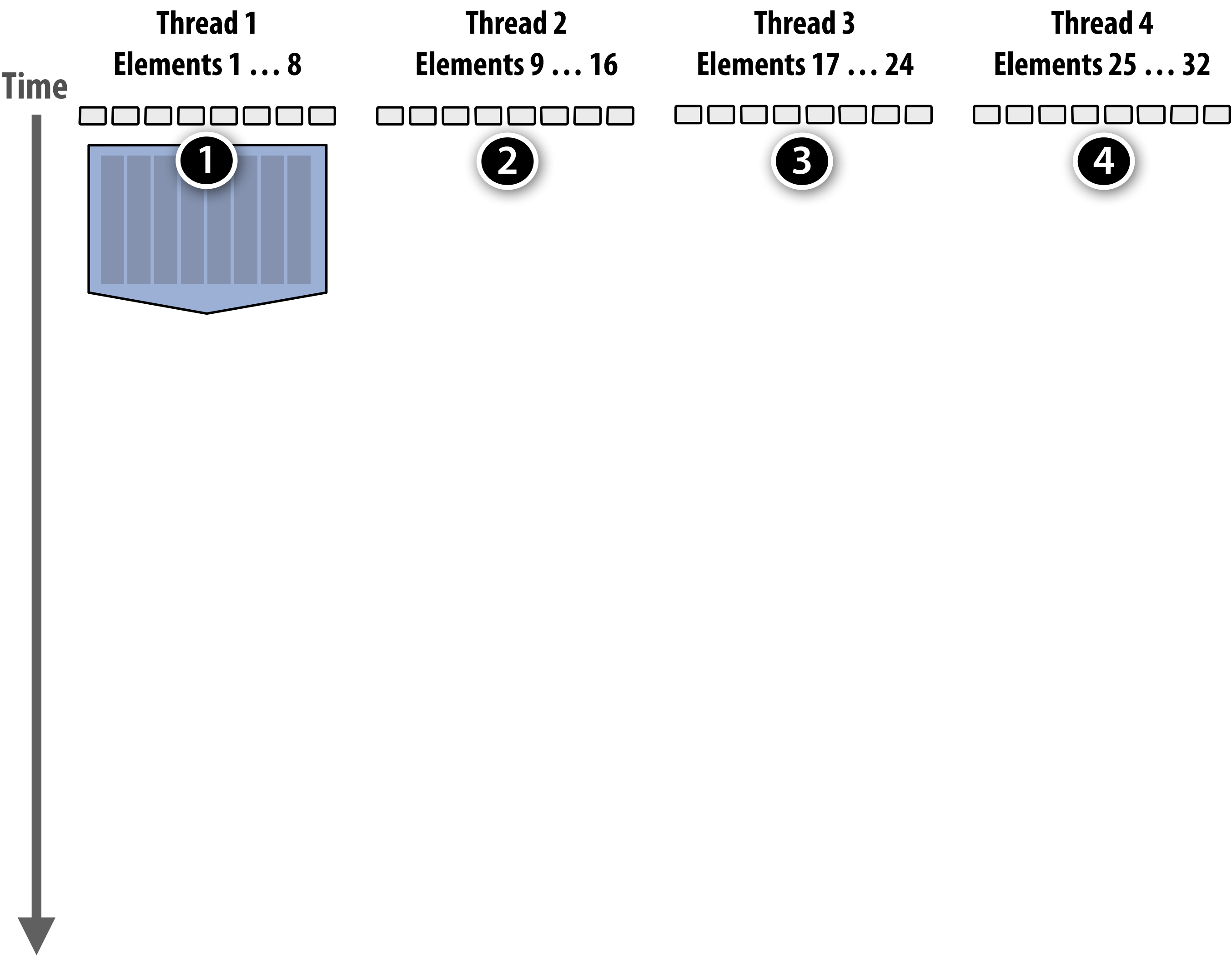
Multi-threading reduces stalls

- Idea: interleave processing of multiple threads on the same core to hide stalls
- Like prefetching, multi-threading is a latency hiding, not a latency reducing technique

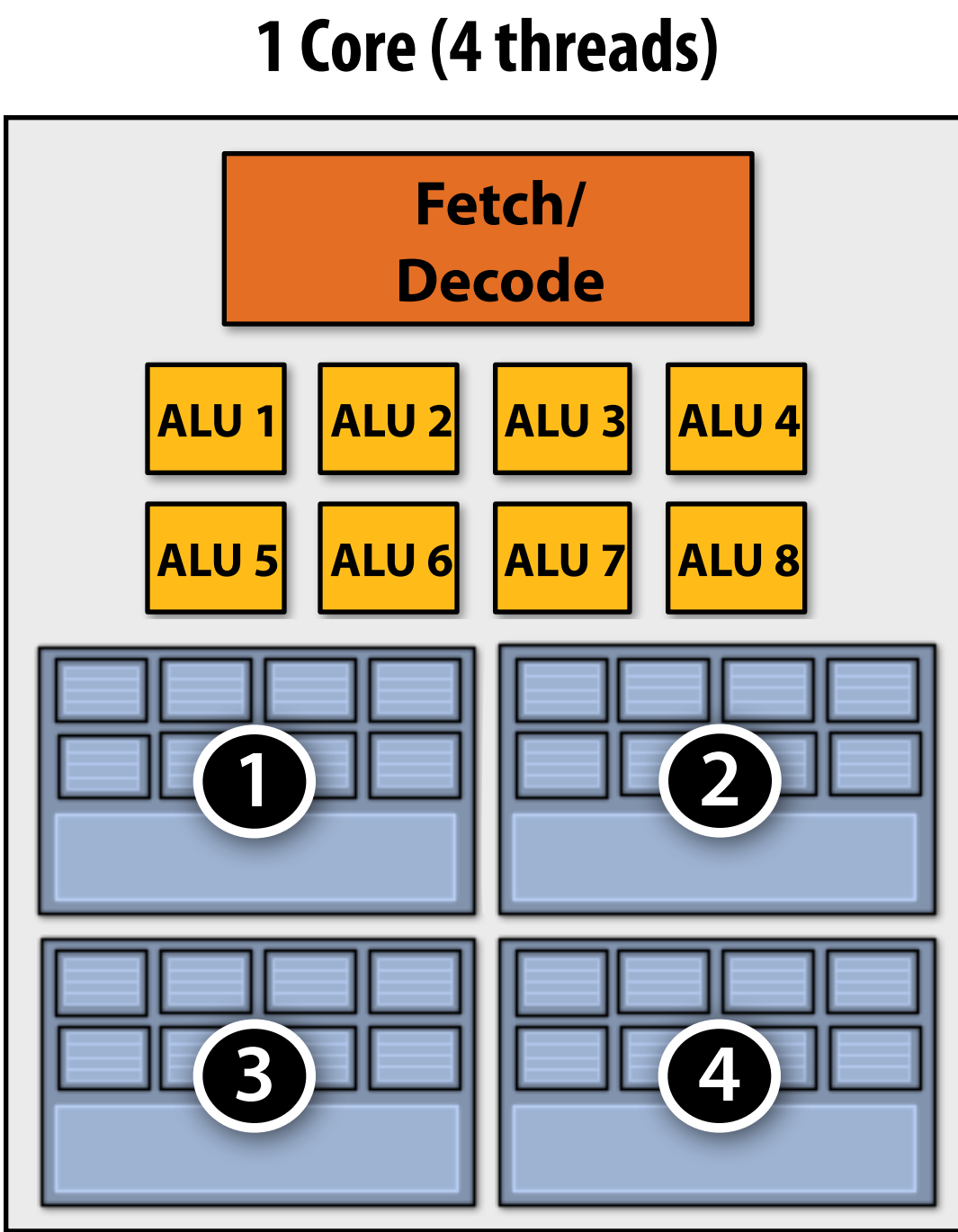
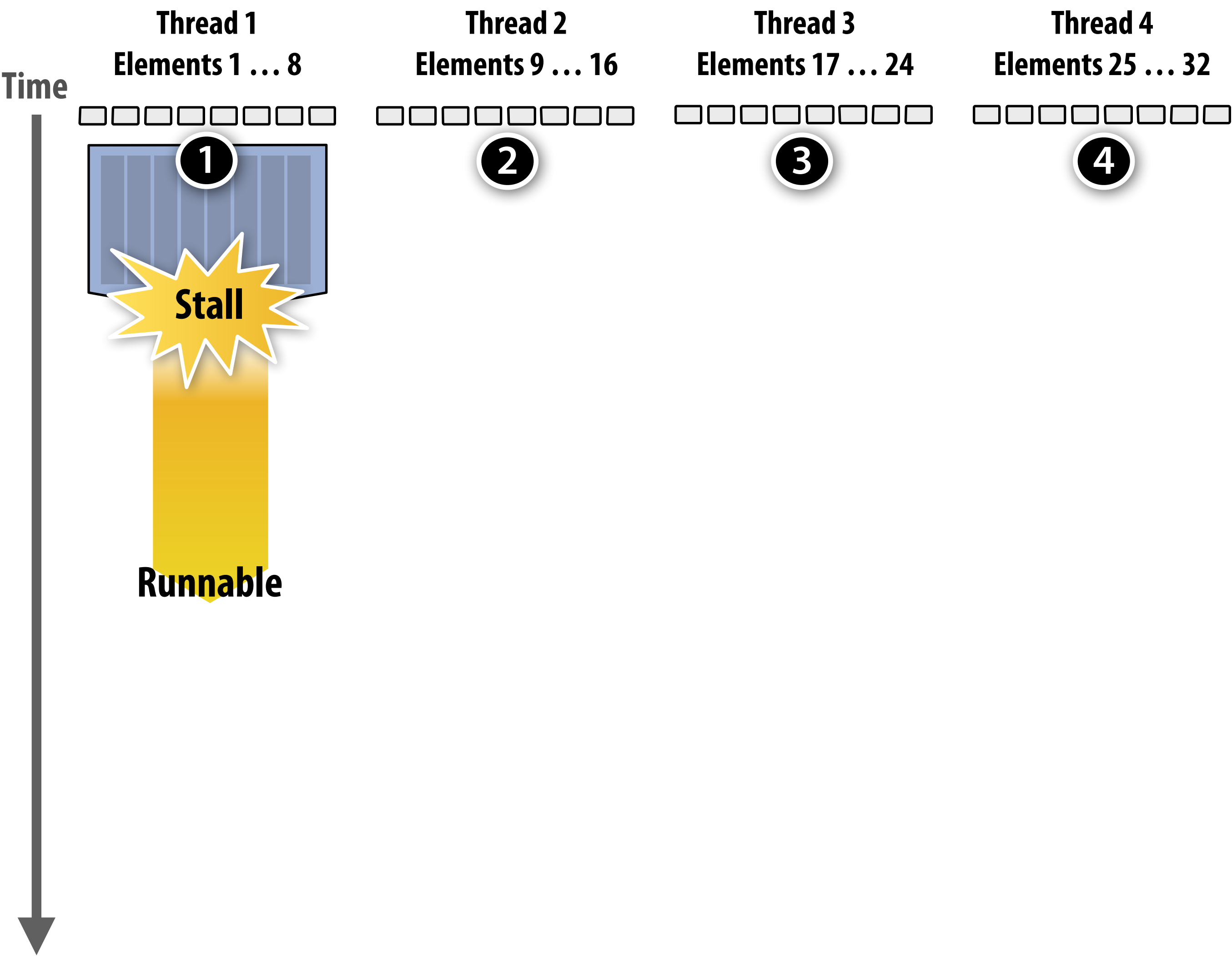
Hiding stalls with multi-threading



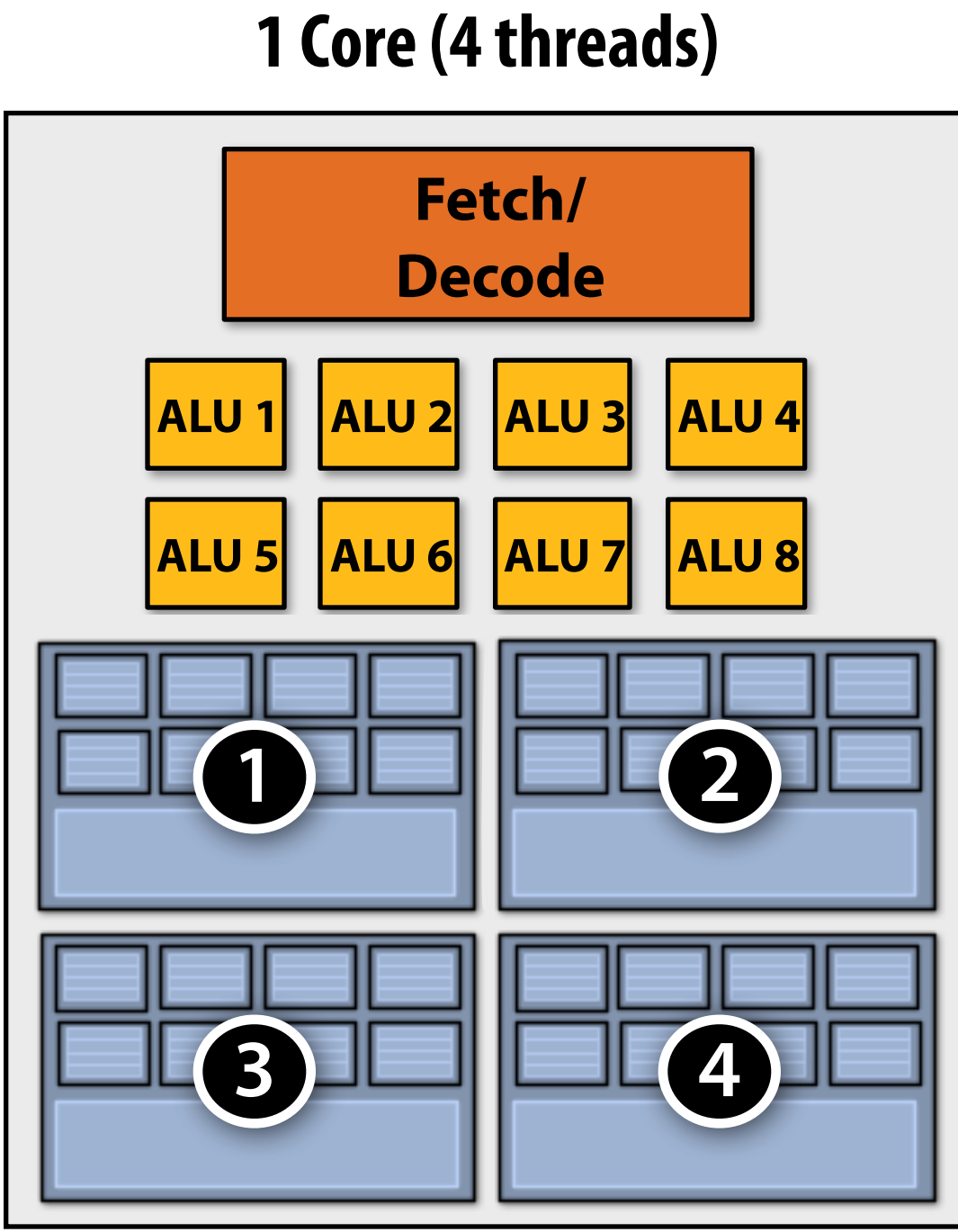
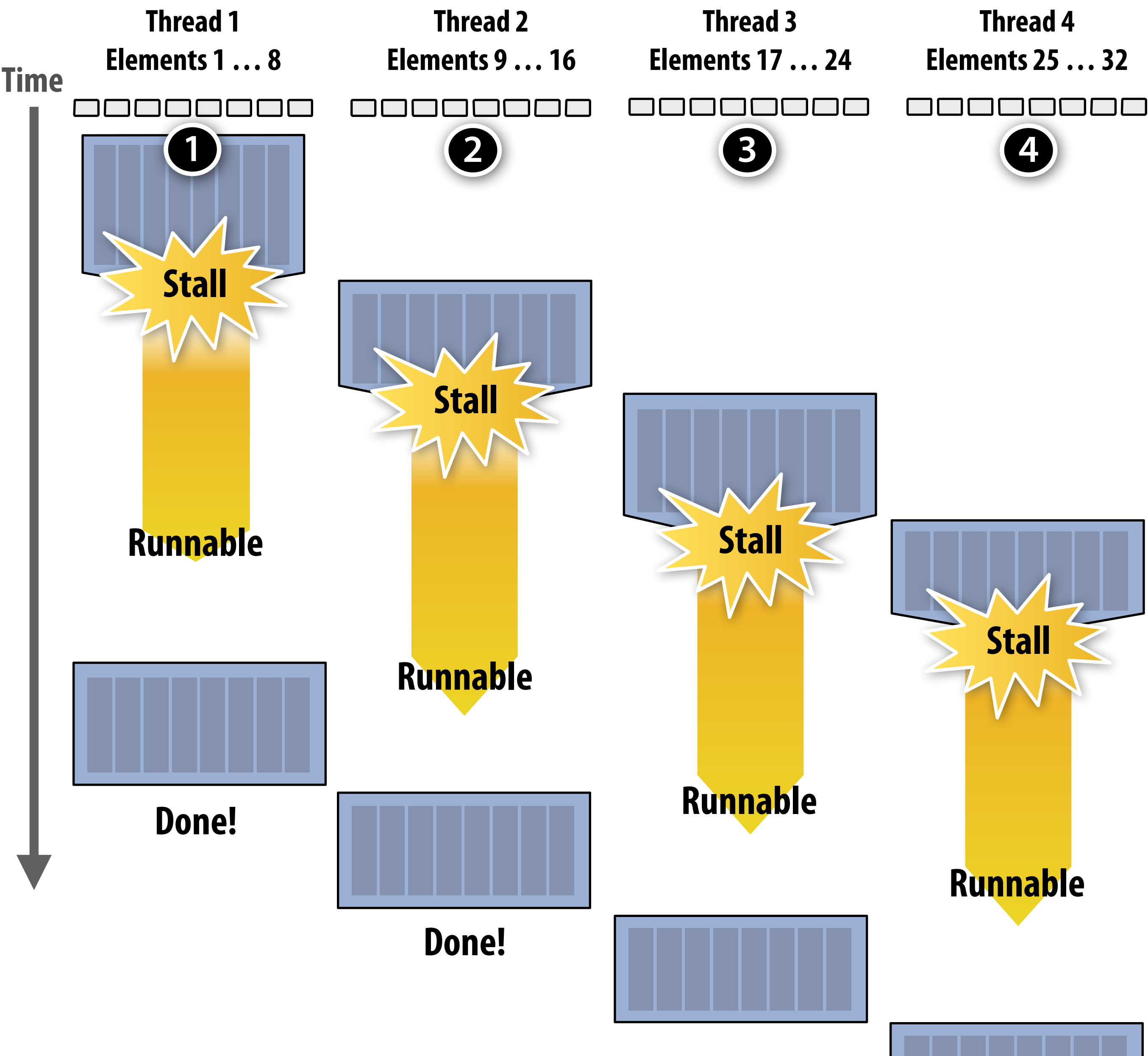
Hiding stalls with multi-threading



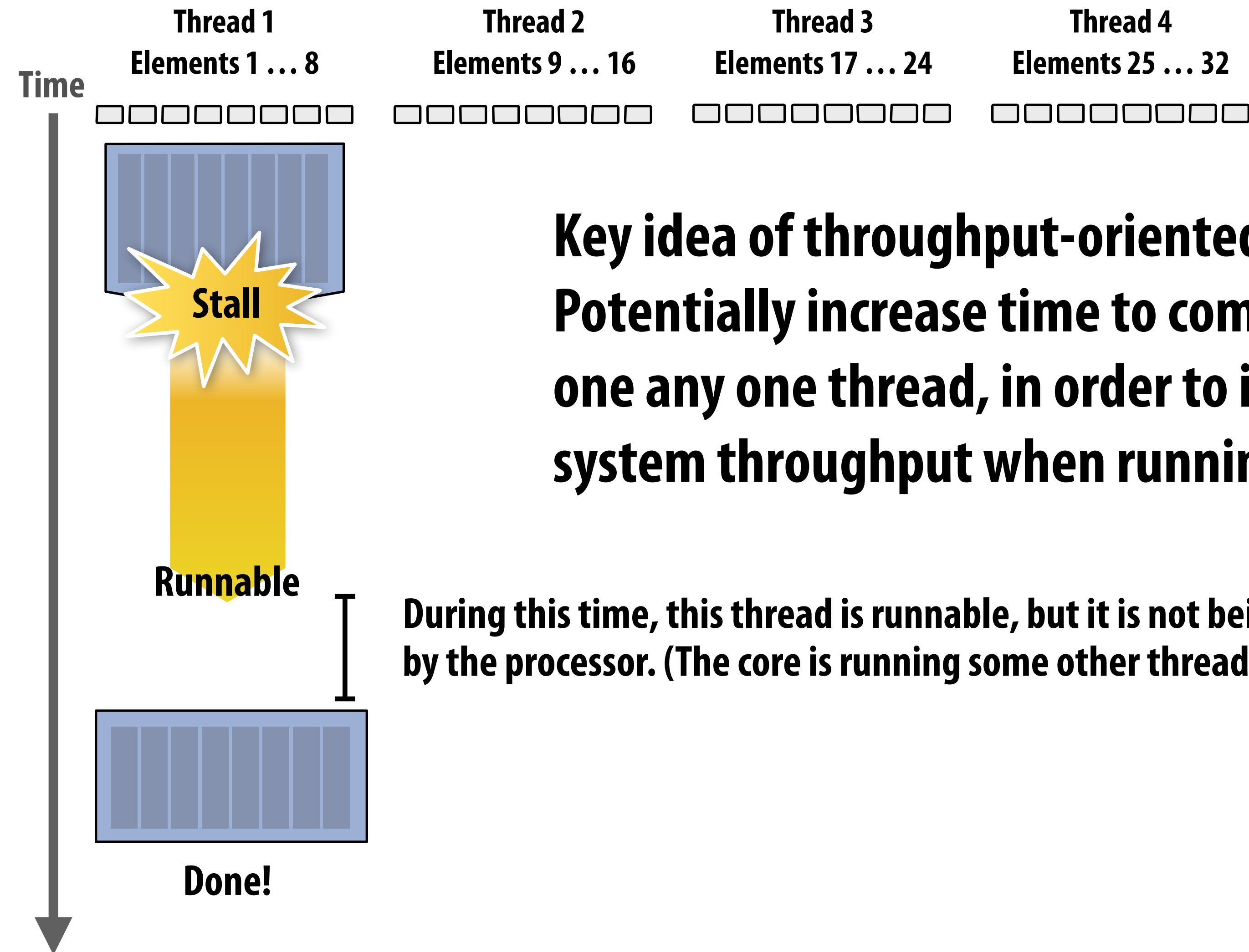
Hiding stalls with multi-threading



Hiding stalls with multi-threading

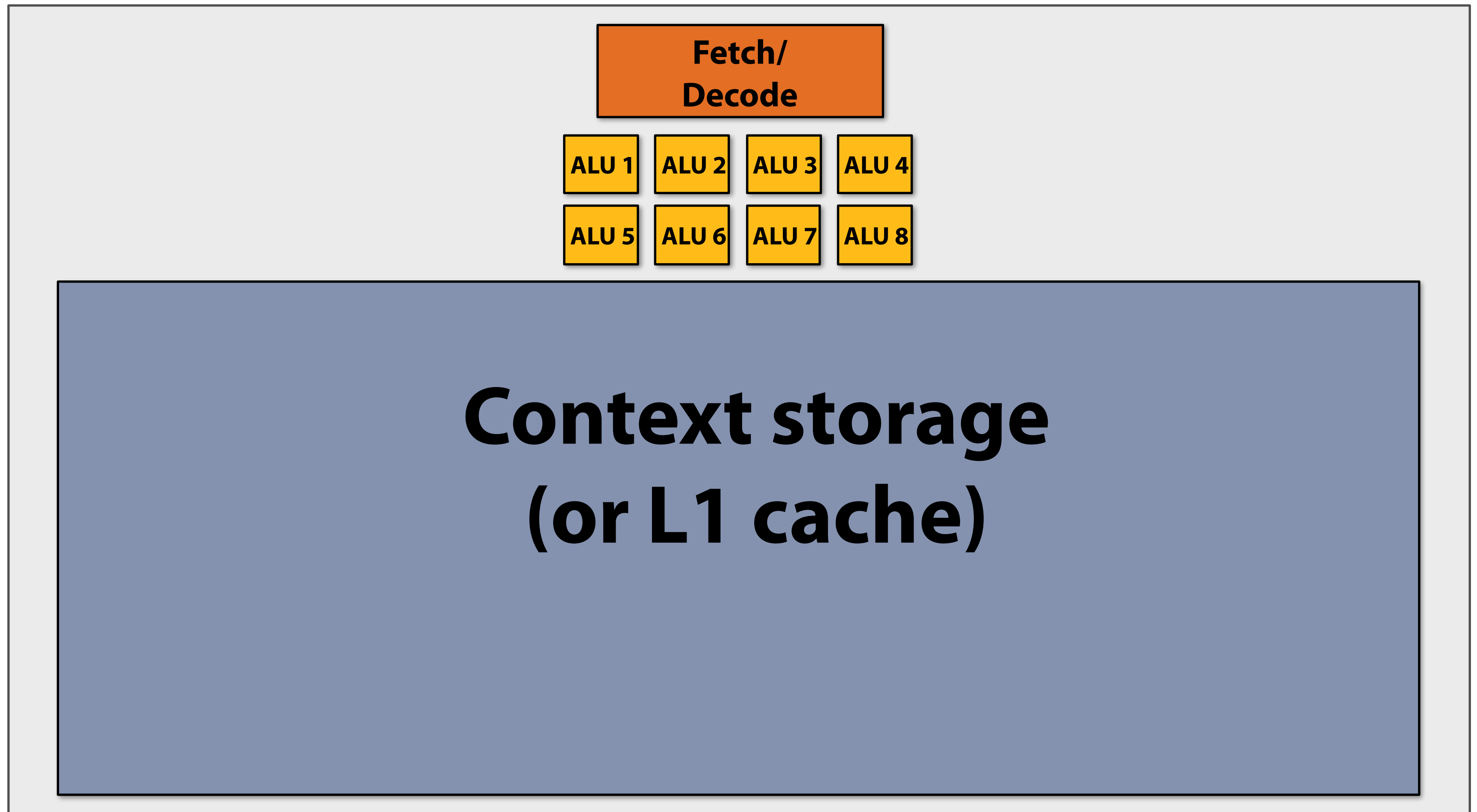


Throughput computing trade-off



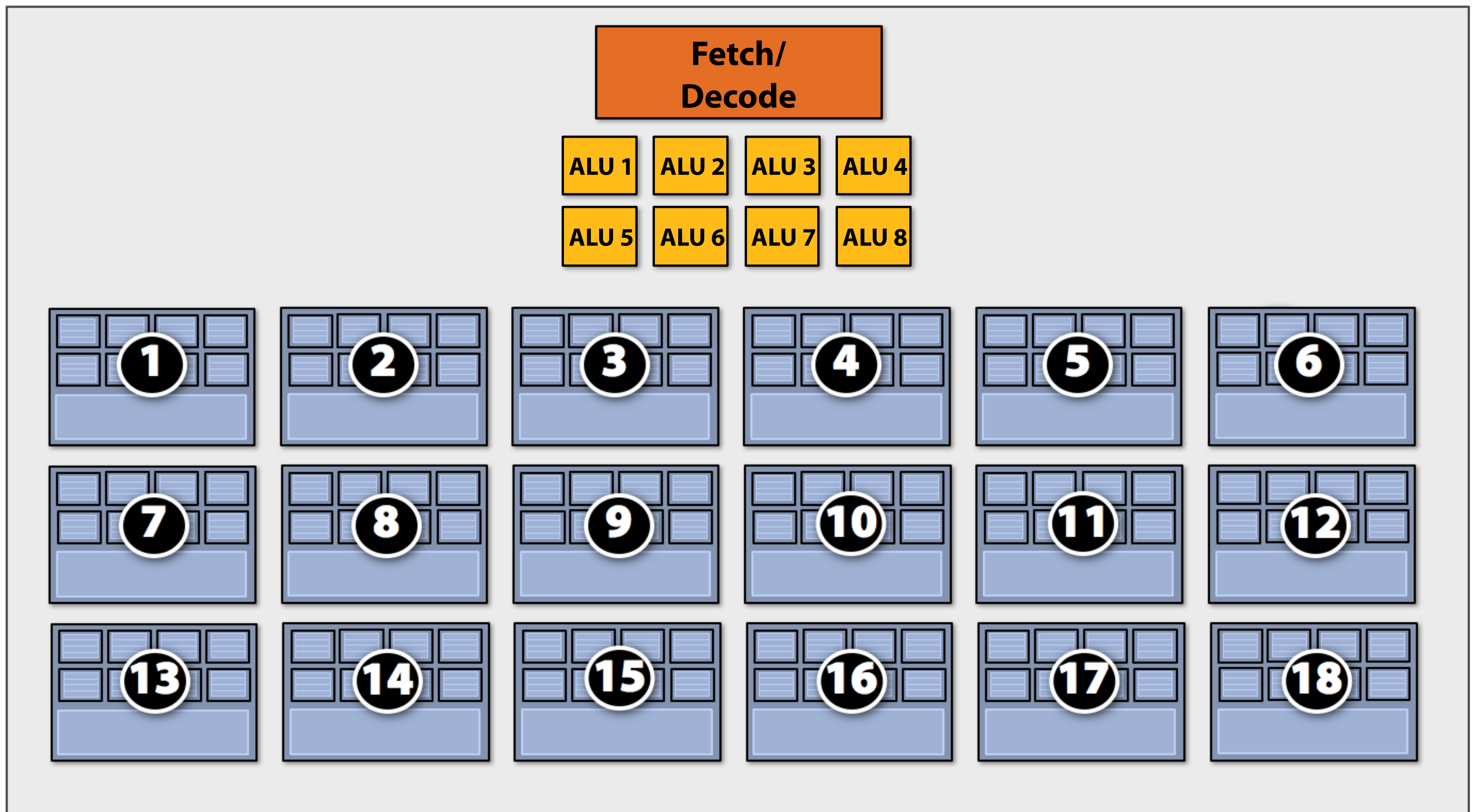
Key idea of throughput-oriented systems:
Potentially increase time to complete work by any one any one thread, in order to increase overall system throughput when running multiple threads.

Storing execution contexts



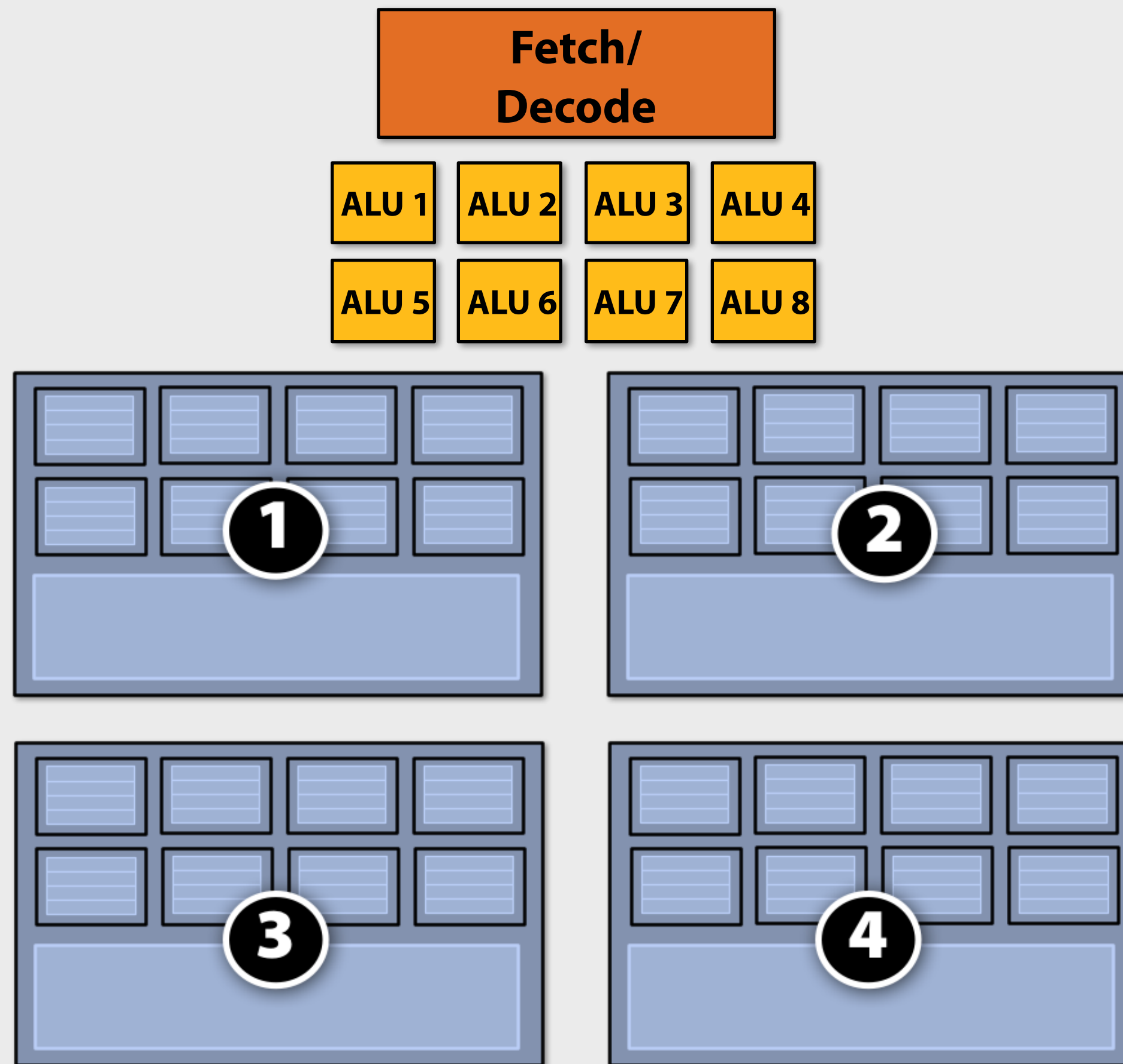
Many small contexts (high latency hiding ability)

1 core (18 threads)



Four large contexts (low latency hiding ability)

1 core (4 threads)



Hardware-supported multi-threading

- **Core manages execution contexts for multiple threads**
 - Runs instructions from runnable threads (processor makes decision about which thread to run each clock, not the operating system)
 - 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
- **Interleaved multi-threading (a.k.a. temporal multi-threading)**
 - What I've described here: each clock, core chooses a thread to run on the ALUs
- **Simultaneous multi-threading (SMT)**
 - Each clock, core chooses instructions from multiple threads to run on ALUs
 - Extension of superscalar CPU design
 - Intel Hyper-threading (2 threads per core)

Multi-threading summary

■ Benefit: more efficiently use core's ALU resources

- Hide memory latency
- Fill multiple functional units of superscalar architecture (when one thread has insufficient ILP)

■ Costs

- Requires additional storage for thread contexts
- Increases run time of any single thread (often not a problem, we usually care about throughput in parallel apps)
- Requires additional independent work in a program (more independent work than ALUs!)
- Relies heavily on memory bandwidth
 - More threads → larger working set → less cache space per thread
 - May go to memory more often, but can hide the latency

Kayvon's fictitious multi-core chip

16 cores

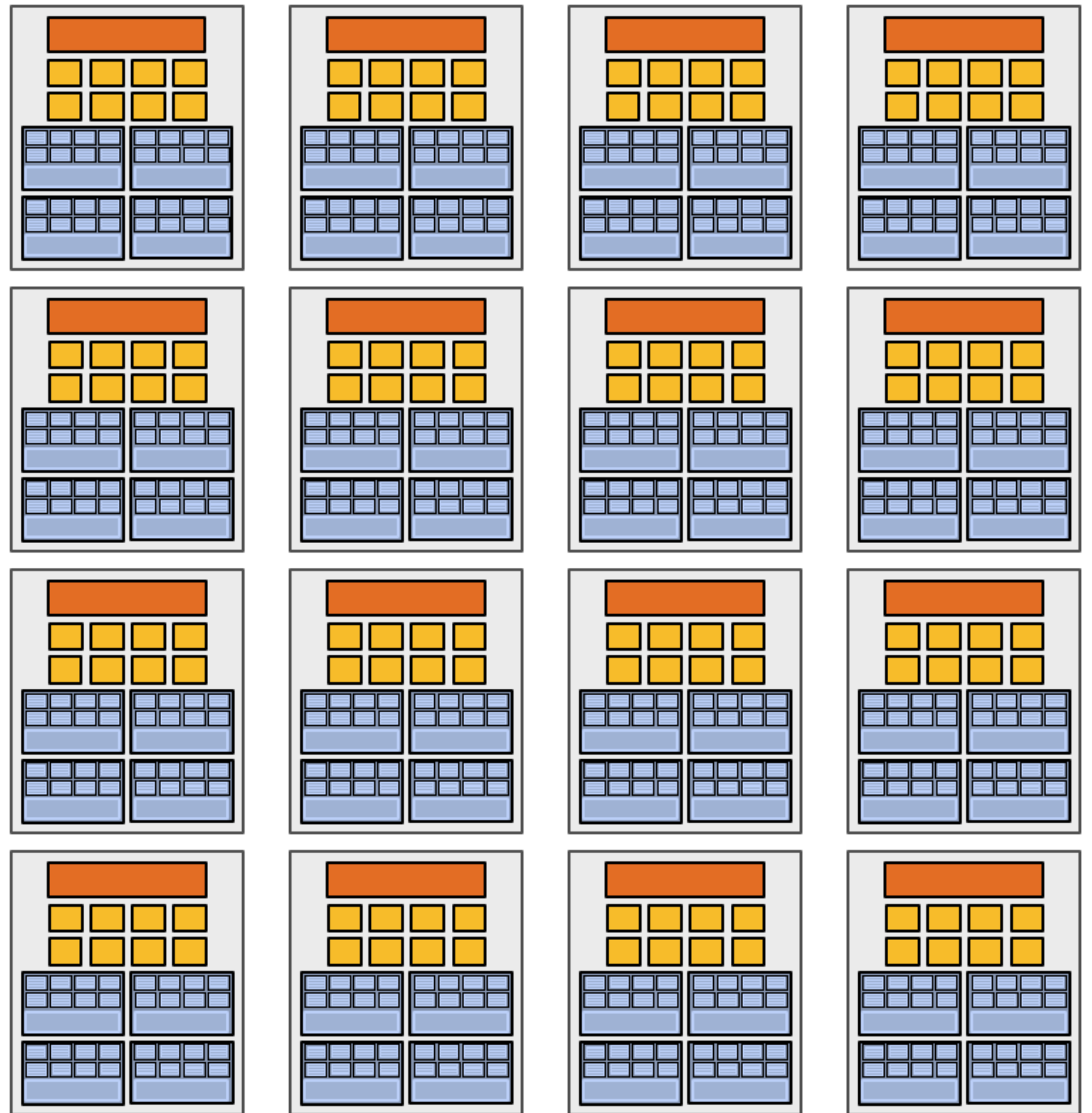
**8 SIMD ALUs per core
(128 total)**

4 threads per core

**16 simultaneous
instruction streams**

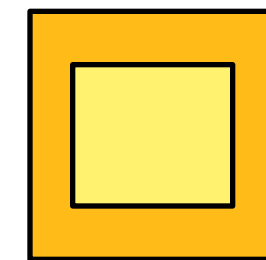
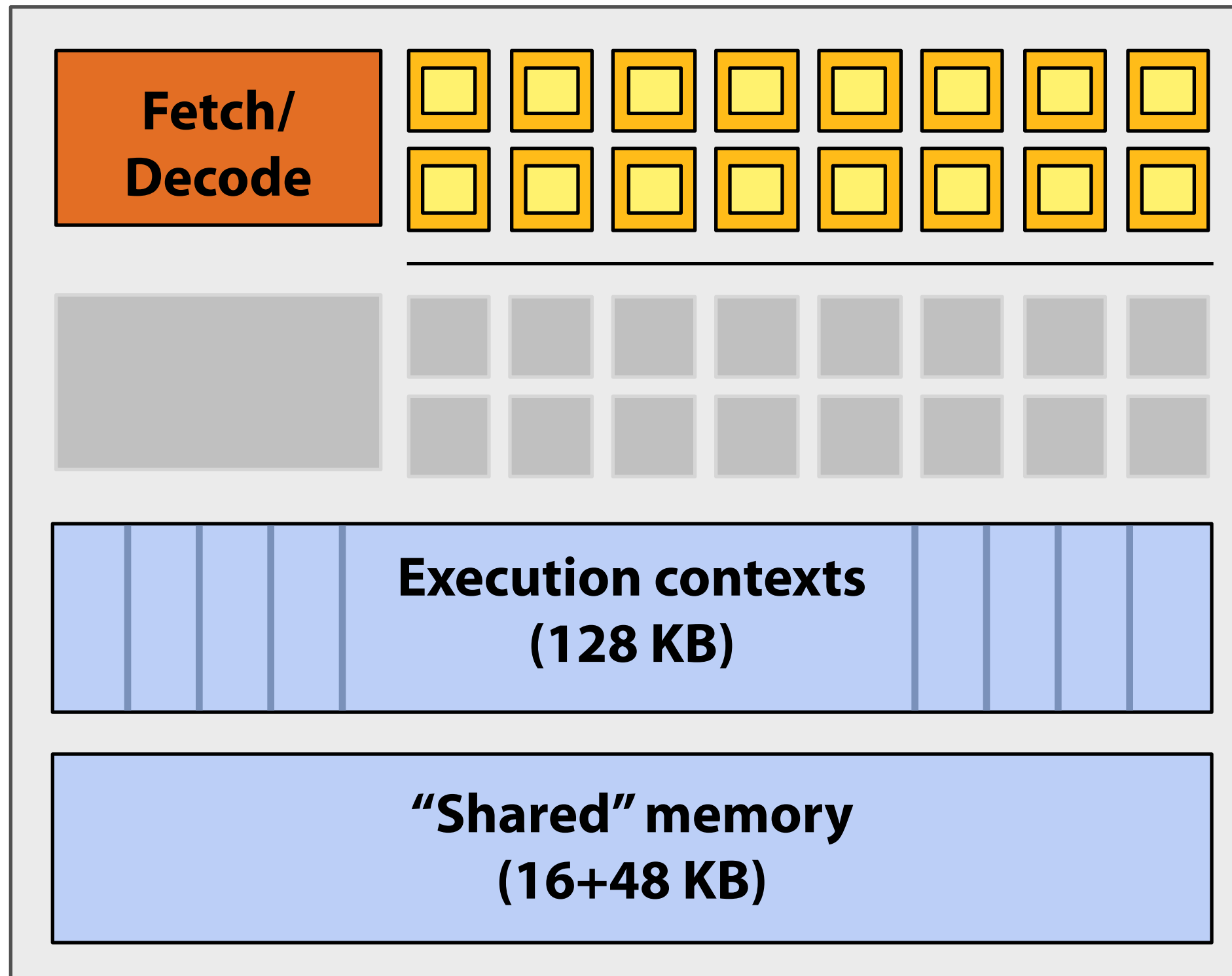
**64 total concurrent
instruction streams**

**512 independent pieces of
work are needed to run chip
with maximal latency
hiding ability**



GPUs: Extreme throughput-oriented processors

NVIDIA GTX 480 core



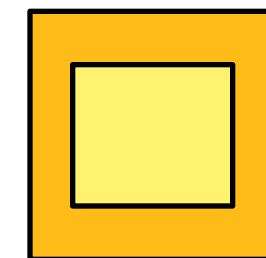
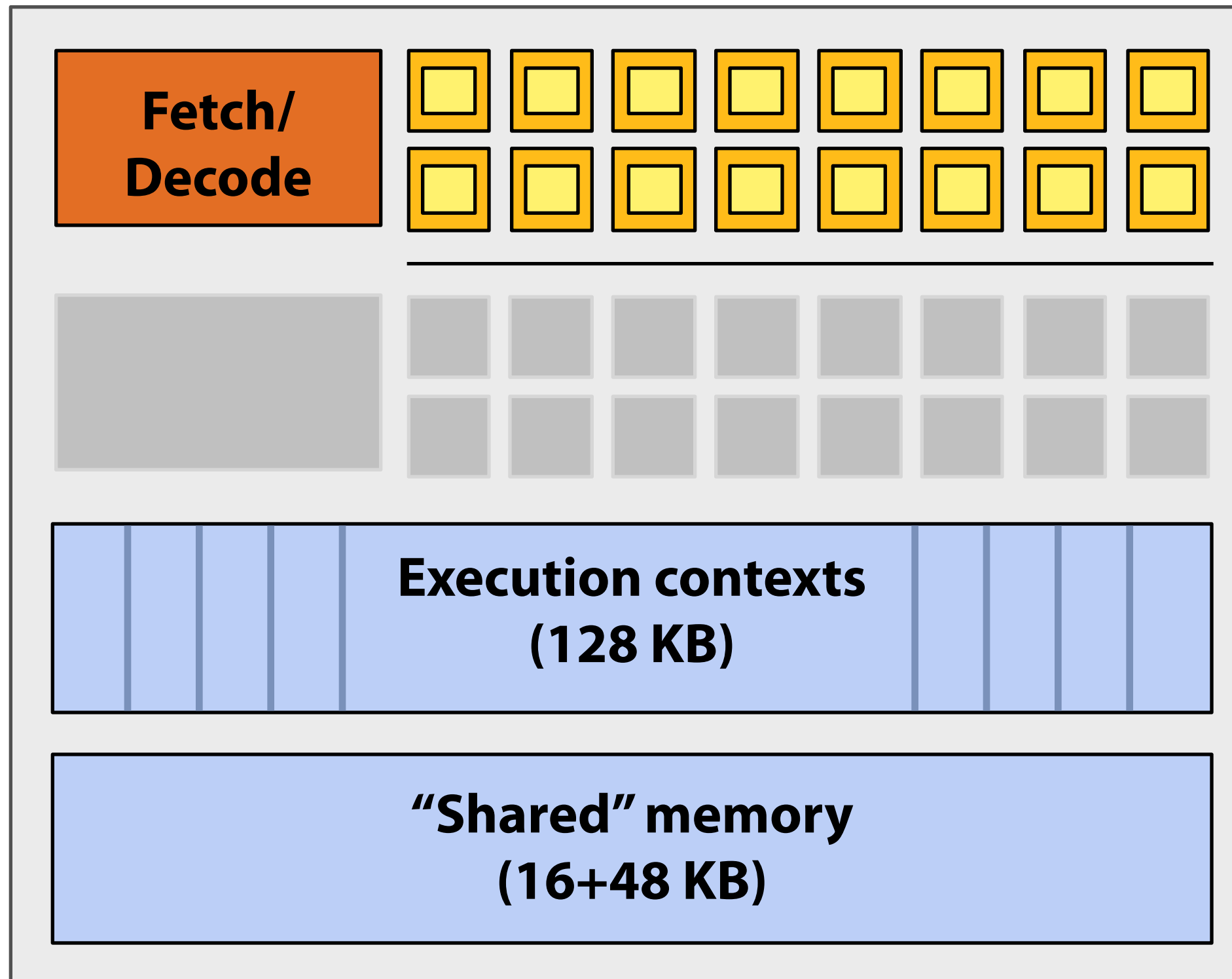
= SIMD function unit,
control shared across 16 units
(1 MUL-ADD per clock)

- Instructions operate on 32 pieces of data at a time (called "warps").
- Think: warp = thread issuing 32-wide vector instructions
- Up to 48 warps are simultaneously interleaved
- Over 1500 elements can be processed concurrently by a core

Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G

NVIDIA GTX 480: more detail (just for the curious)

NVIDIA GTX 480 core



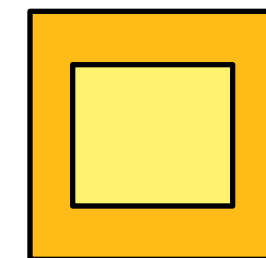
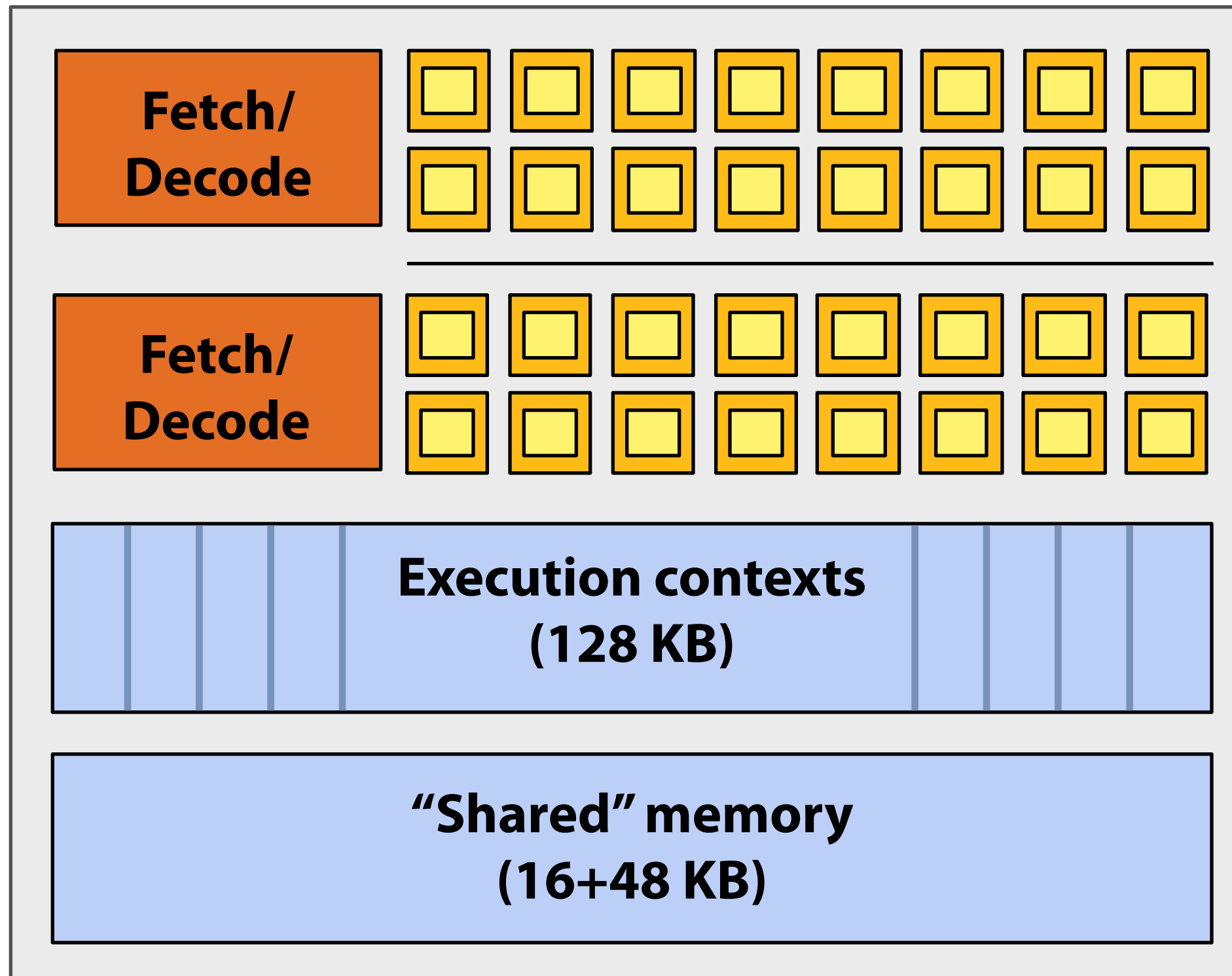
= SIMD function unit,
control shared across 16 units
(1 MUL-ADD per clock)

- Why is a warp 32 elements and there are only 16 SIMD ALUs?
- It's a bit complicated: ALUs run at twice the clock rate of rest of chip. So each decoded instruction runs on 32 pieces of data on the 16 ALUs over two ALU clocks. (but to the programmer, it behaves like a 32-wide SIMD operation)

Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G

NVIDIA GTX 480: more detail (just for the curious)

NVIDIA GTX 480 core

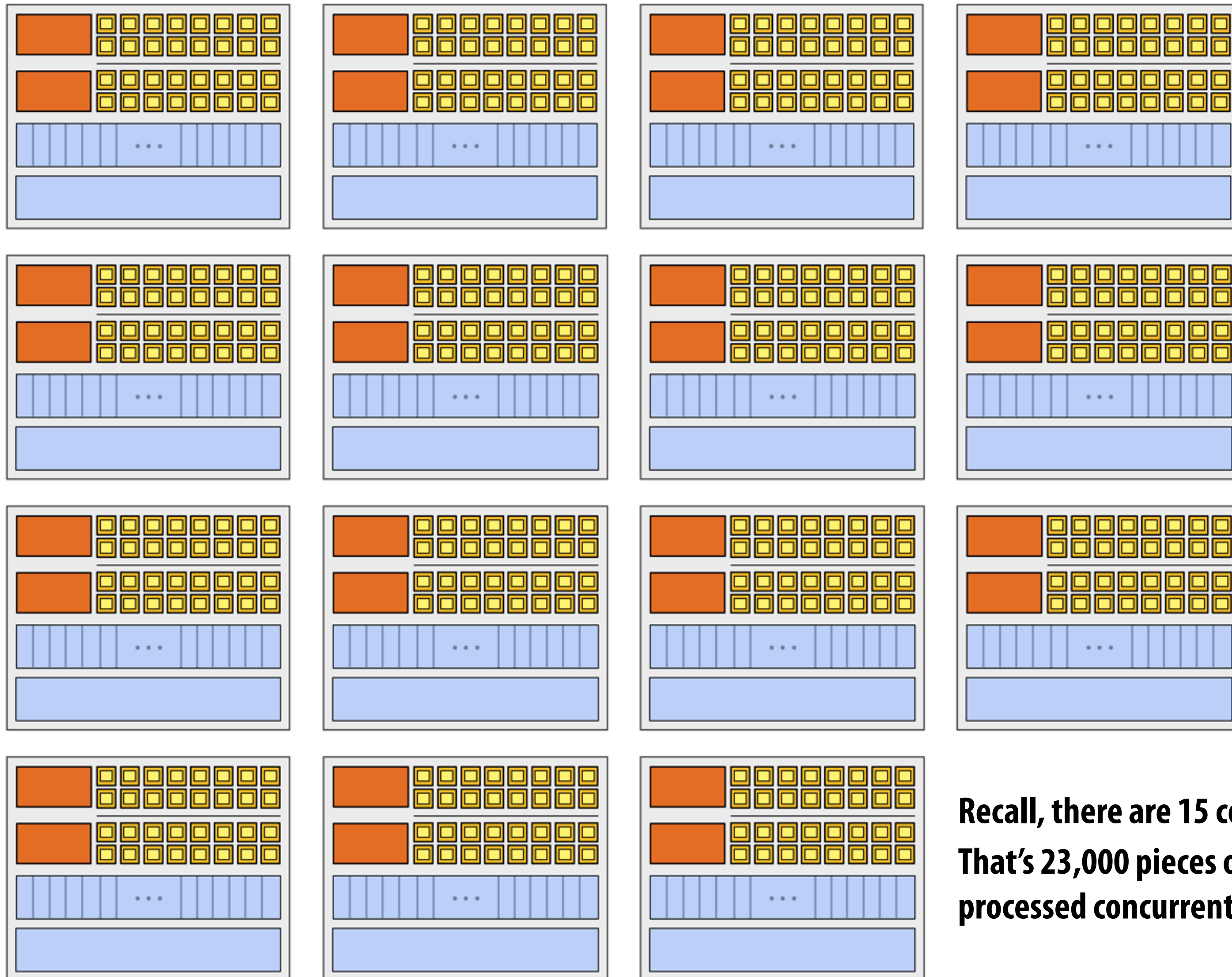


= SIMD function unit,
control shared across 16 units
(1 MUL-ADD per clock)

- This process occurs on another set of 16 ALUs as well
- So there are 32 ALUs per core
- $15 \times 32 = 480$ ALUs per chip

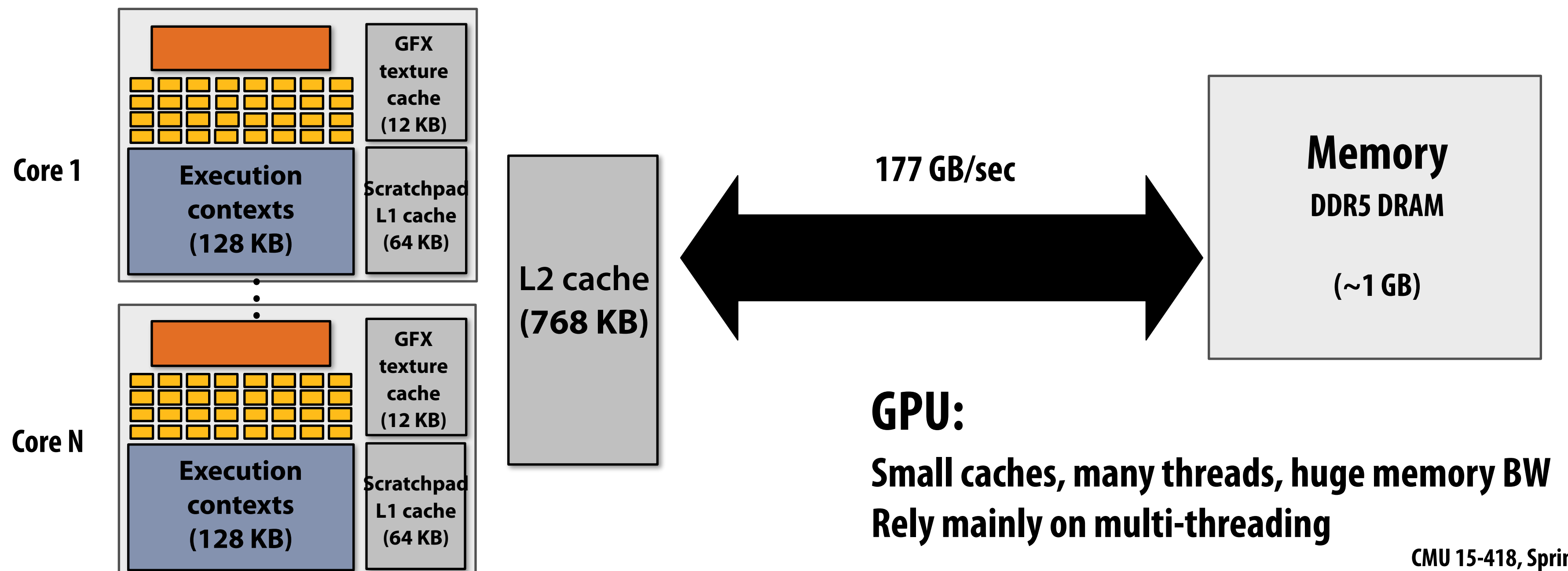
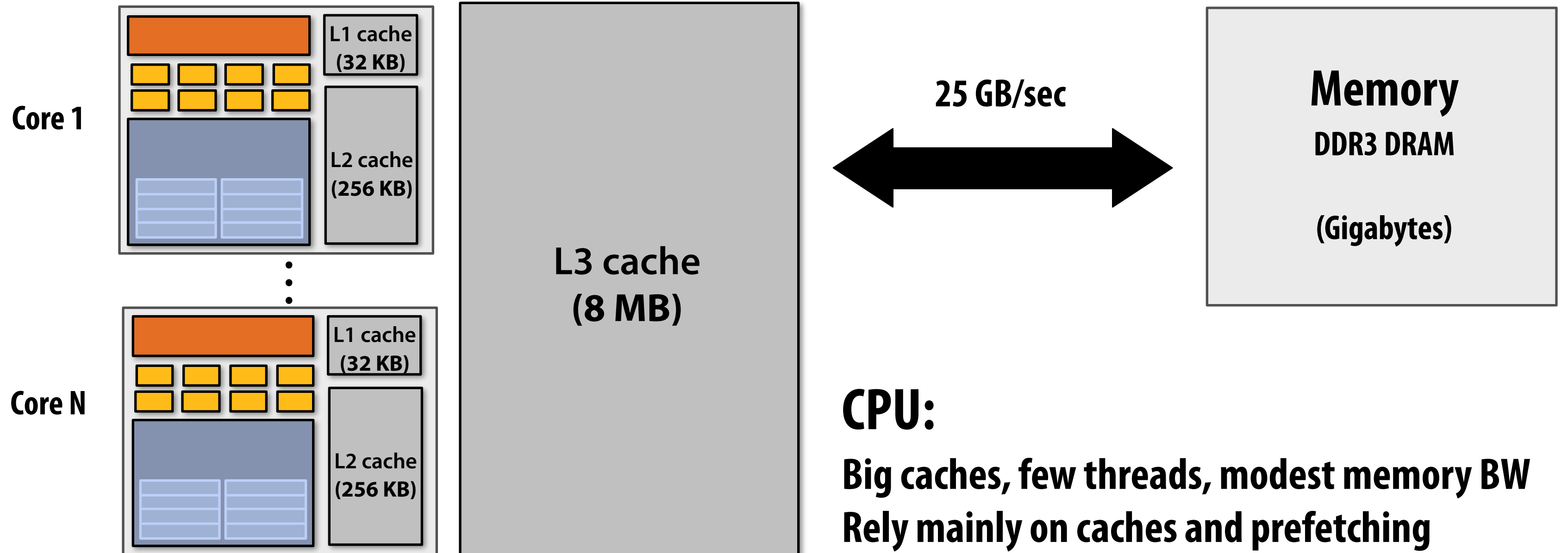
Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G

NVIDIA GTX 480



**Recall, there are 15 cores on the GTX 480:
That's 23,000 pieces of data being
processed concurrently!**

CPU vs. GPU memory hierarchies

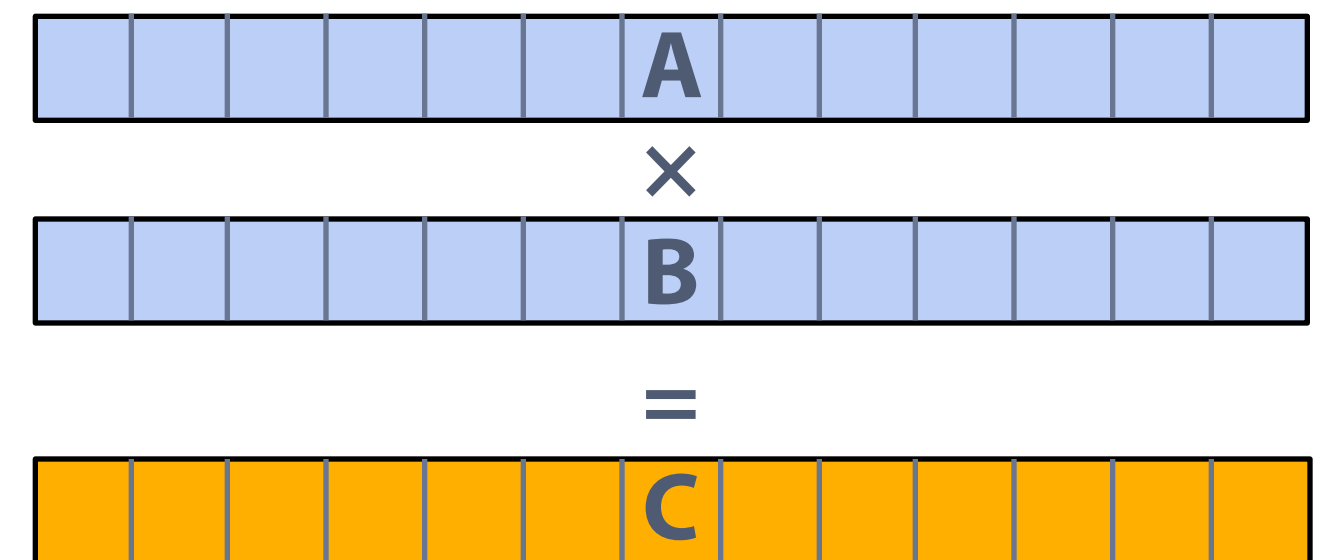


Thought experiment

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 $A[i] \times B[i]$**
- **Store result into C[i]**



Three memory operations (12 bytes) for every MUL

NVIDIA GTX 480 GPU can do 480 MULs per clock (@ 1.2 GHz)

Need ~6.4 TB/sec of bandwidth to keep functional units busy (only have 177 GB/sec)

~ 3% efficiency... but 7x faster than quad-core CPU!

(2.6 GHz Core i7 Gen 4 quad-core CPU connected to 25 GB/sec memory bus will exhibit similar efficiency on this computation)

Bandwidth limited!

If processors request data at too high a rate, the memory system cannot keep up.

No amount of latency hiding helps this.

Overcoming bandwidth limits are a common challenge for application developers on throughput-optimized systems.

Bandwidth is a critical resource

Performant parallel programs will:

- **Organize computation to fetch data from memory less often**
 - **Reuse data while processing instruction stream for a single thread (traditional temporal/spatial locality optimizations)**
 - **Share data across threads (inter-thread cooperation)**
- **Request data less often (instead, do more arithmetic: it's "free")**
 - **Term: "arithmetic intensity" — ratio of math operations to data access operations in an instruction stream**
 - **Main point: programs must have high arithmetic intensity to utilize modern processors efficiently**

Summary

- **Three major ideas that all modern processors employ to varying degrees**
 - **Employ multiple processing cores**
 - **Simpler cores (embrace thread-level parallelism over instruction-level parallelism)**
 - **Amortize instruction stream processing over many ALUs (SIMD)**
 - **Increase compute capability with little extra cost**
 - **Use multi-threading to make more efficient use of processing resources (hide latencies, fill all available resources)**
- **Due to high arithmetic capability on modern chips, many parallel applications (on both CPUs and GPUs) are bandwidth bound**
- **GPUs push throughput computing concepts to extreme scales**
 - **Notable differences in memory system design**

Know these terms

- **Multi-core processor**
- **SIMD execution**
- **Coherent control flow**
- **Hardware multi-threading**
 - **Interleaved multi-threading**
 - **Simultaneous multi-threading**
- **Memory latency**
- **Memory bandwidth**
- **Bandwidth bound application**
- **Arithmetic intensity**