

Lecture 2:

Modern Multi-Core Architecture:

(multiple cores + SIMD + threads)

(and their performance characteristics)

CMU 15-418: Parallel Computer Architecture and Programming (Spring 2012)

Announcements

- **WWW**

- <http://www.cs.cmu.edu/afs/cs/academic/class/15418-s12/www/>

- **We have another TA!**

- **Mike Mu**

(Yes, our TAs are Michael and Mike)

Review

- 1. Why has single threaded performance topped out in recent years?**
- 2. What prevented us from obtaining maximum speedup from the parallel programs we wrote last time?**

Today

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

Part 1: parallel execution

Example program

Compute $\sin(x)$ using Taylor expansion: $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \dots$

```
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 *= (j+3) * (j+4);
            sign *= -1;
        }

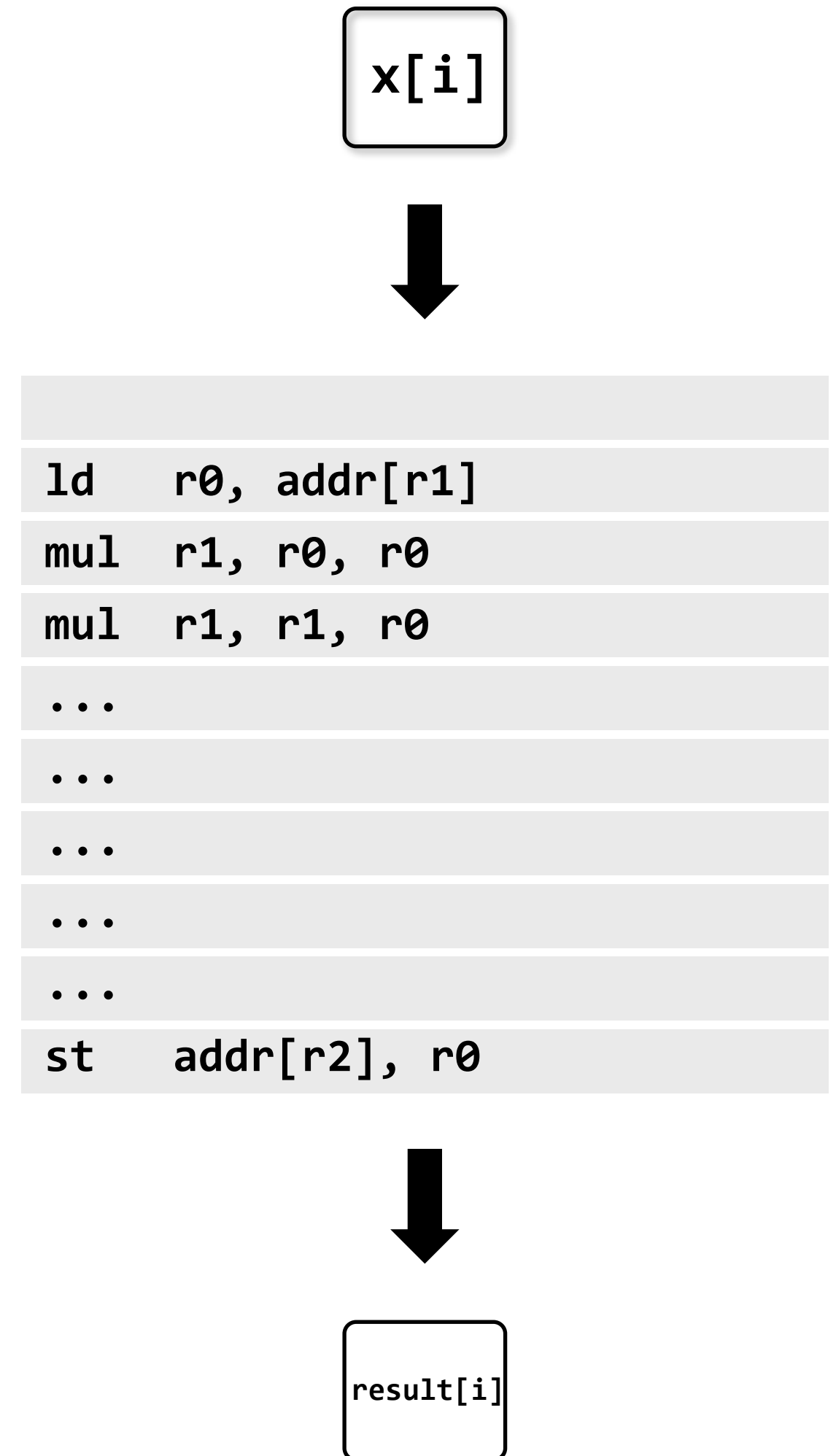
        result[i] = value;
    }
}
```

Compile program

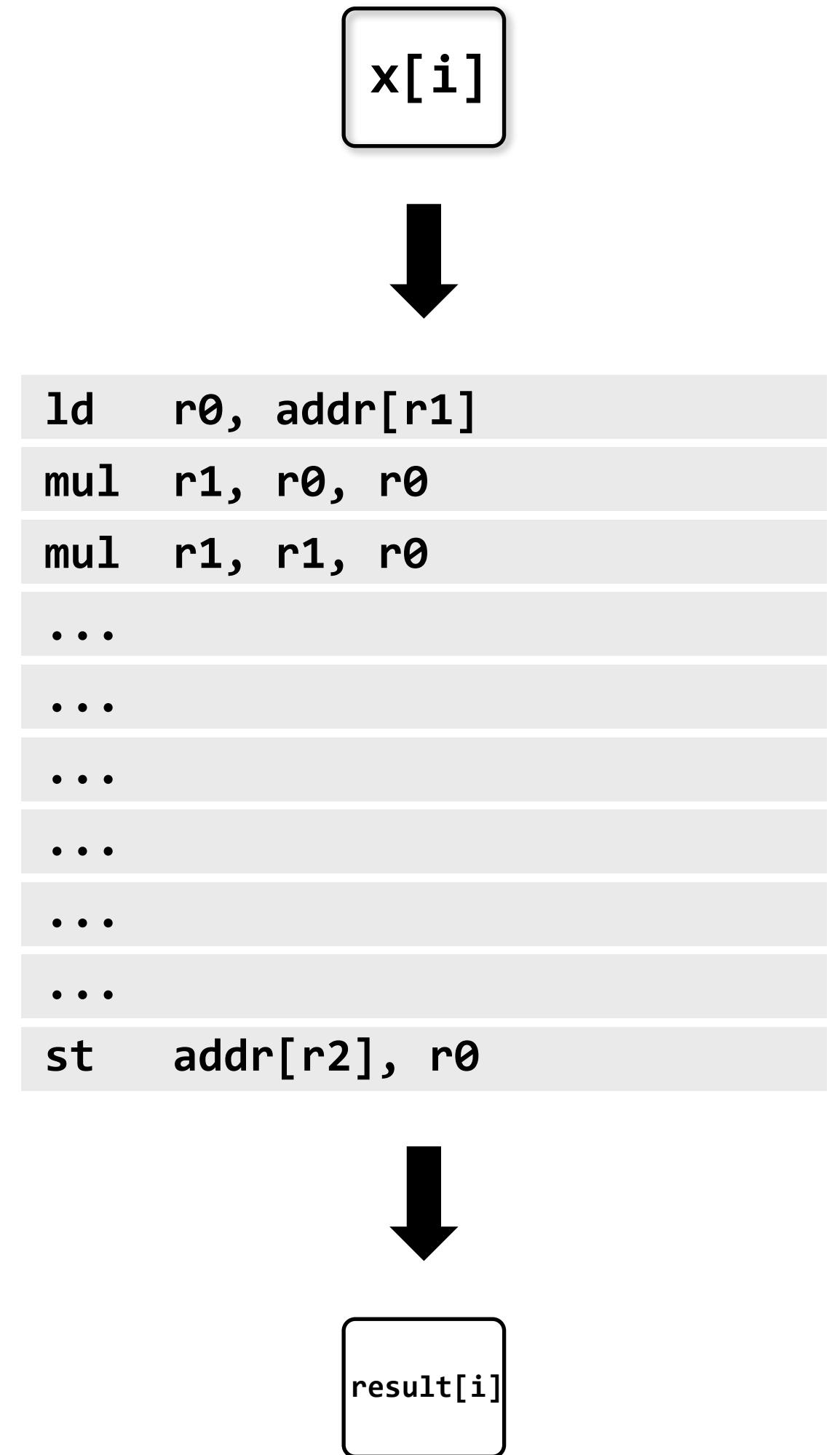
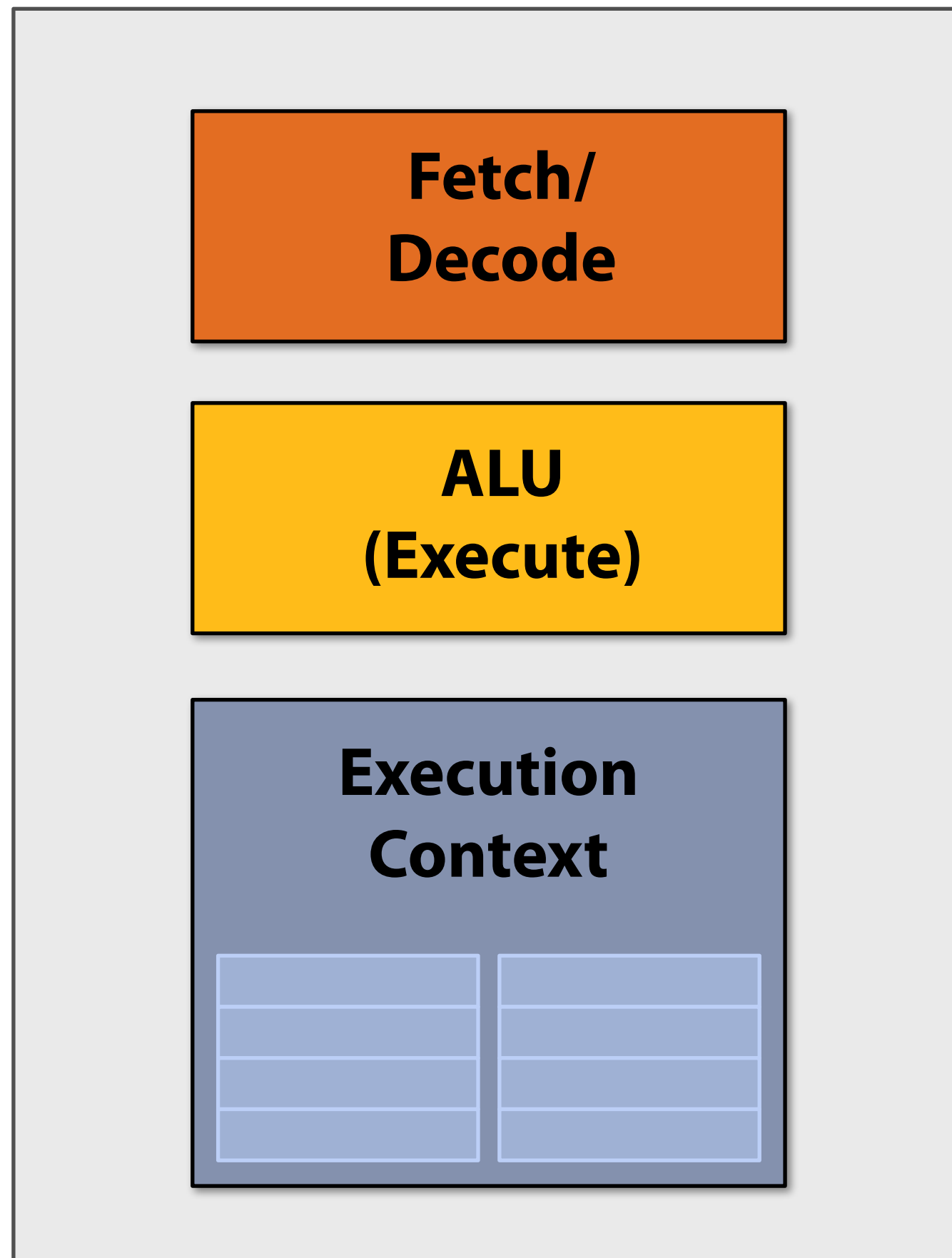
```
void sinx(int N, int terms, float* x, float* result)
{
    for (int i=0; i<N; i++)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6; // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom
            numer *= x[i] * x[i];
            denom *= (j+3) * (j+4);
            sign *= -1;
        }

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

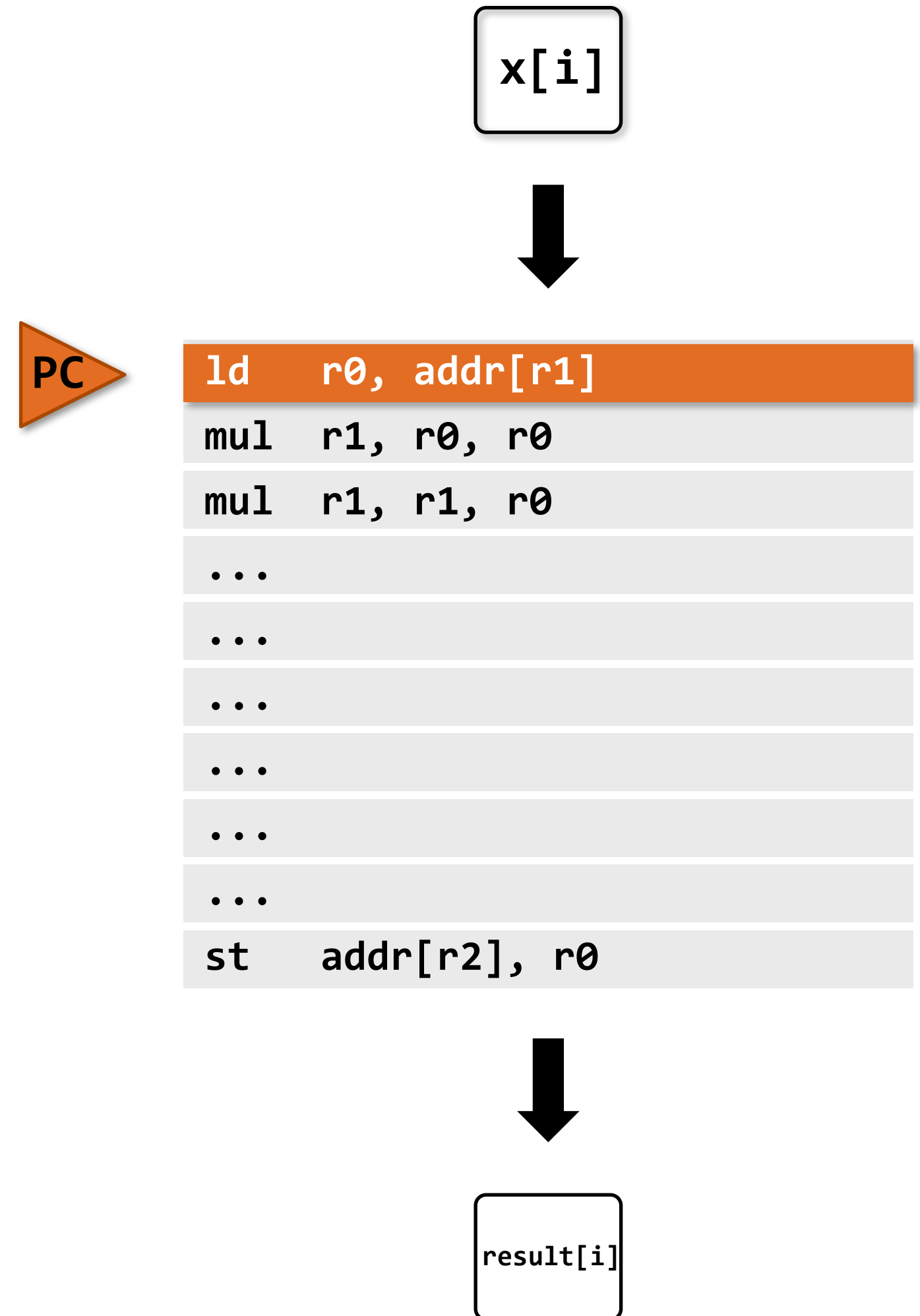
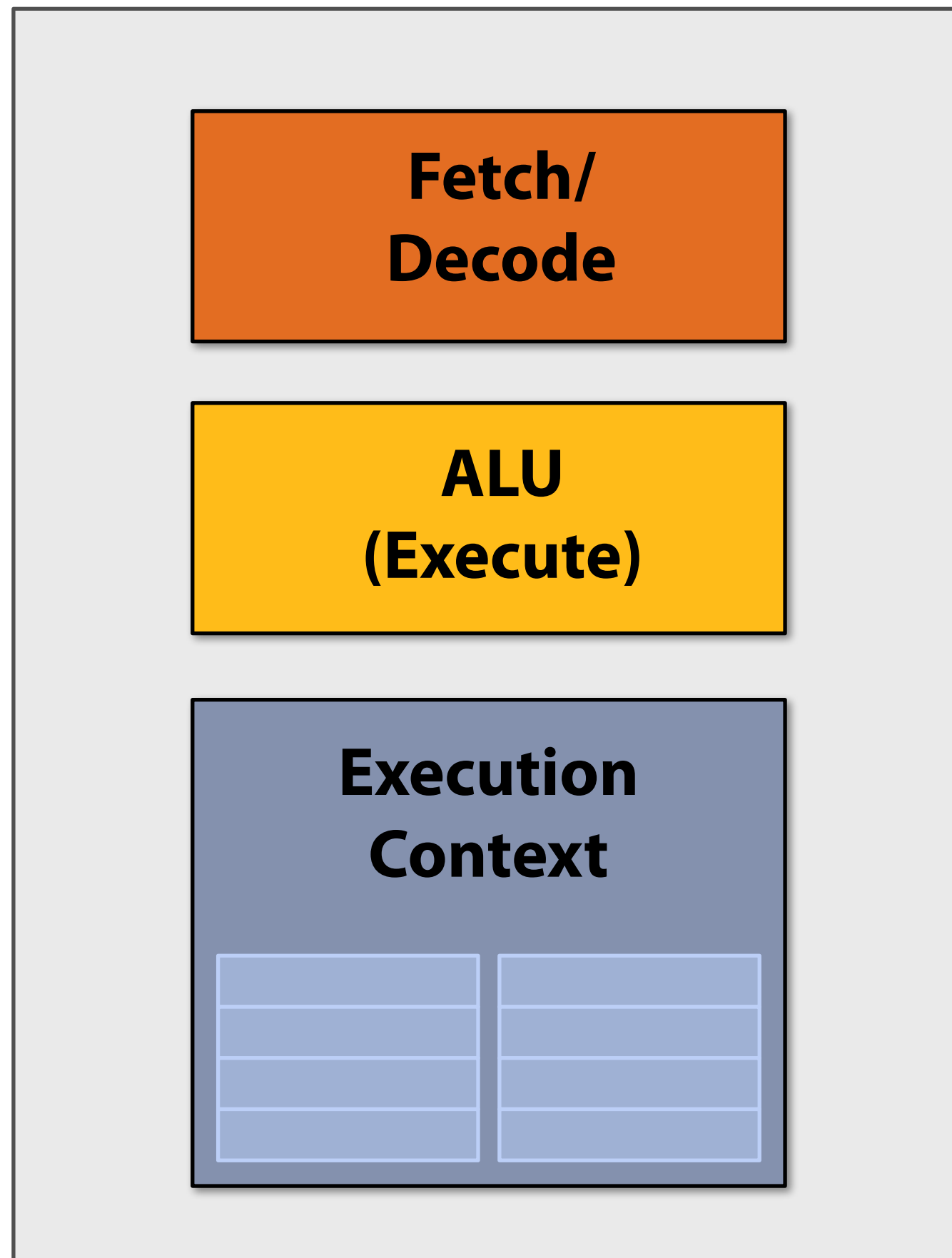


Execute program



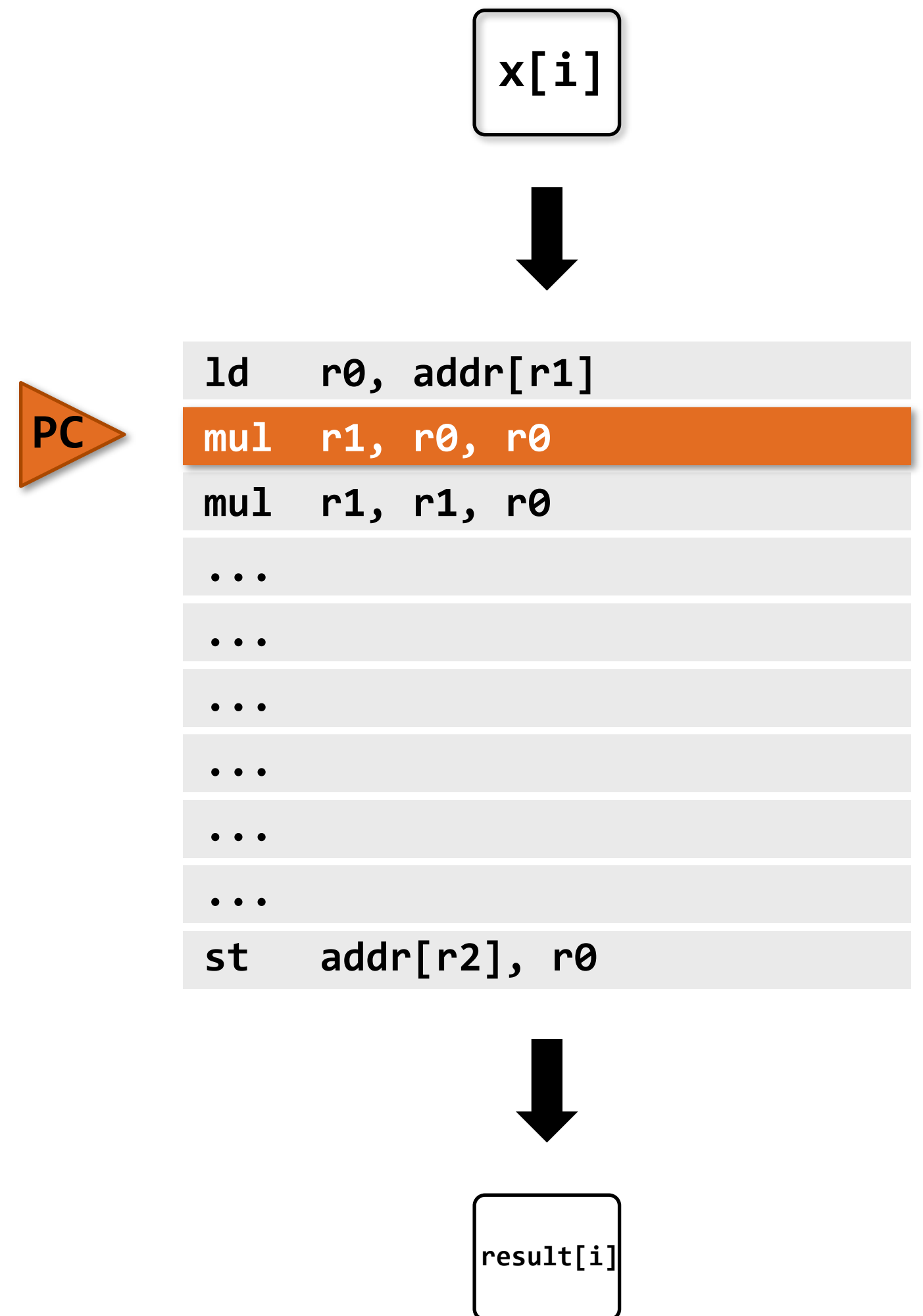
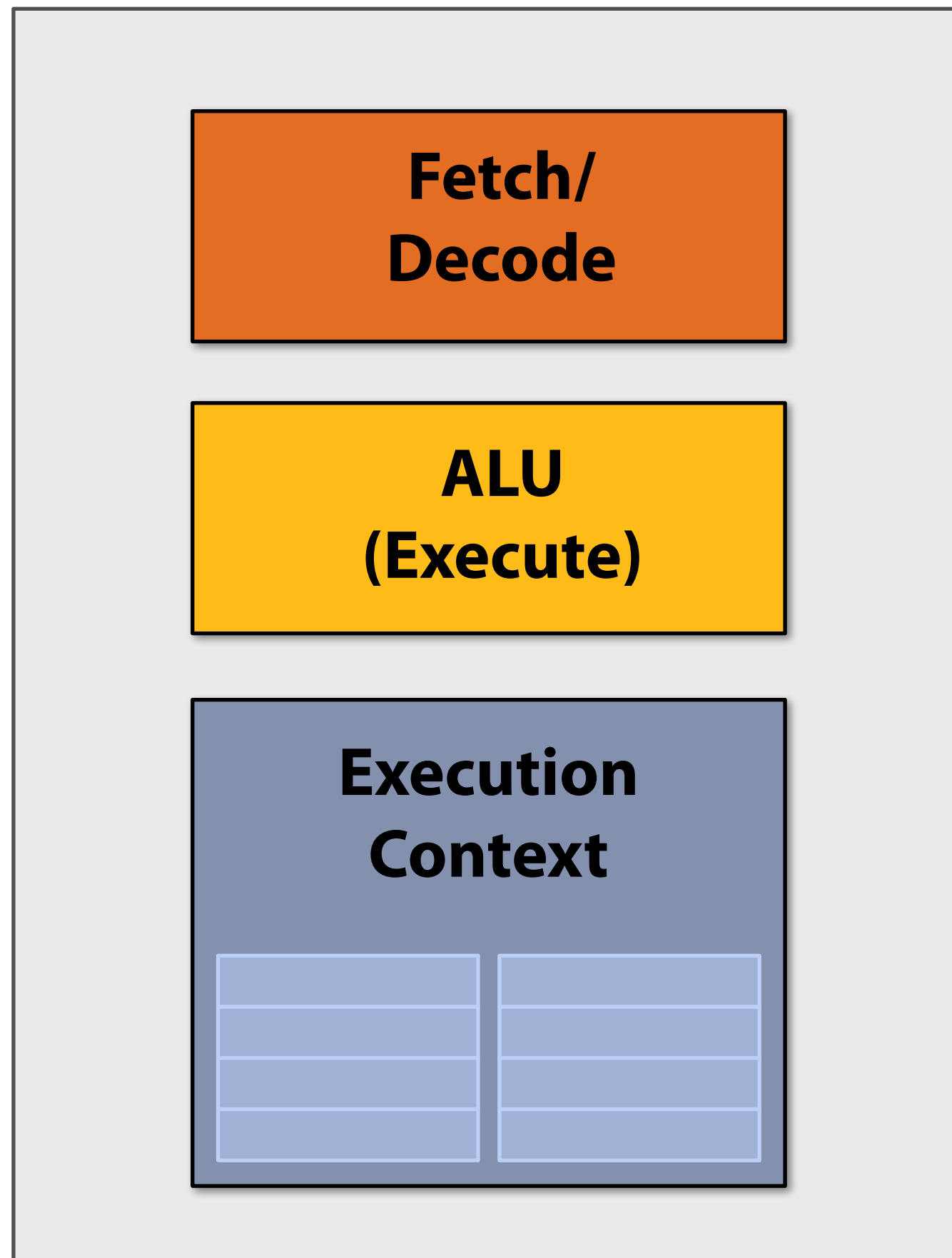
Execute program

Very simple processor: execute one instruction per clock



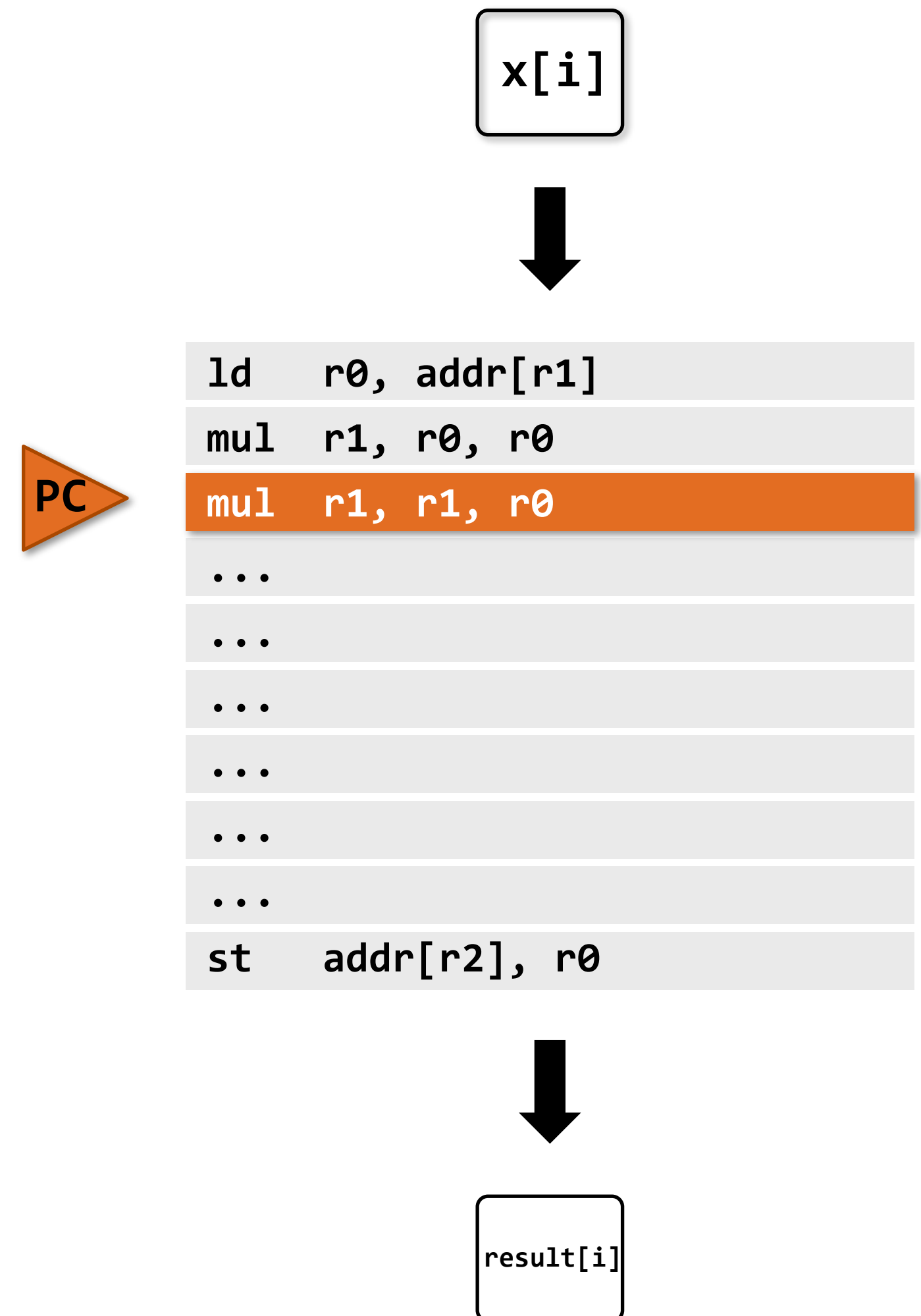
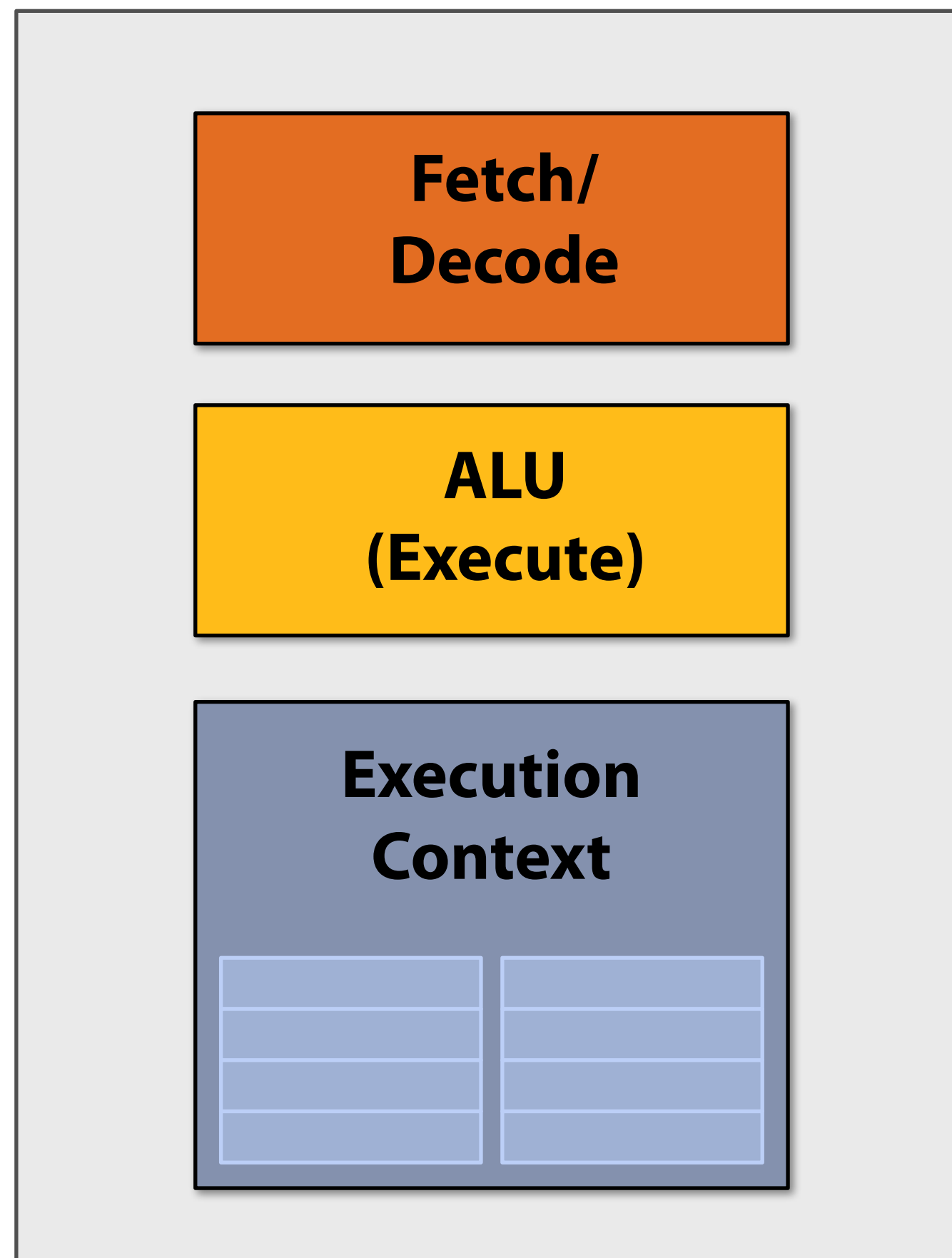
Execute program

Very simple processor: execute one instruction per clock



Execute program

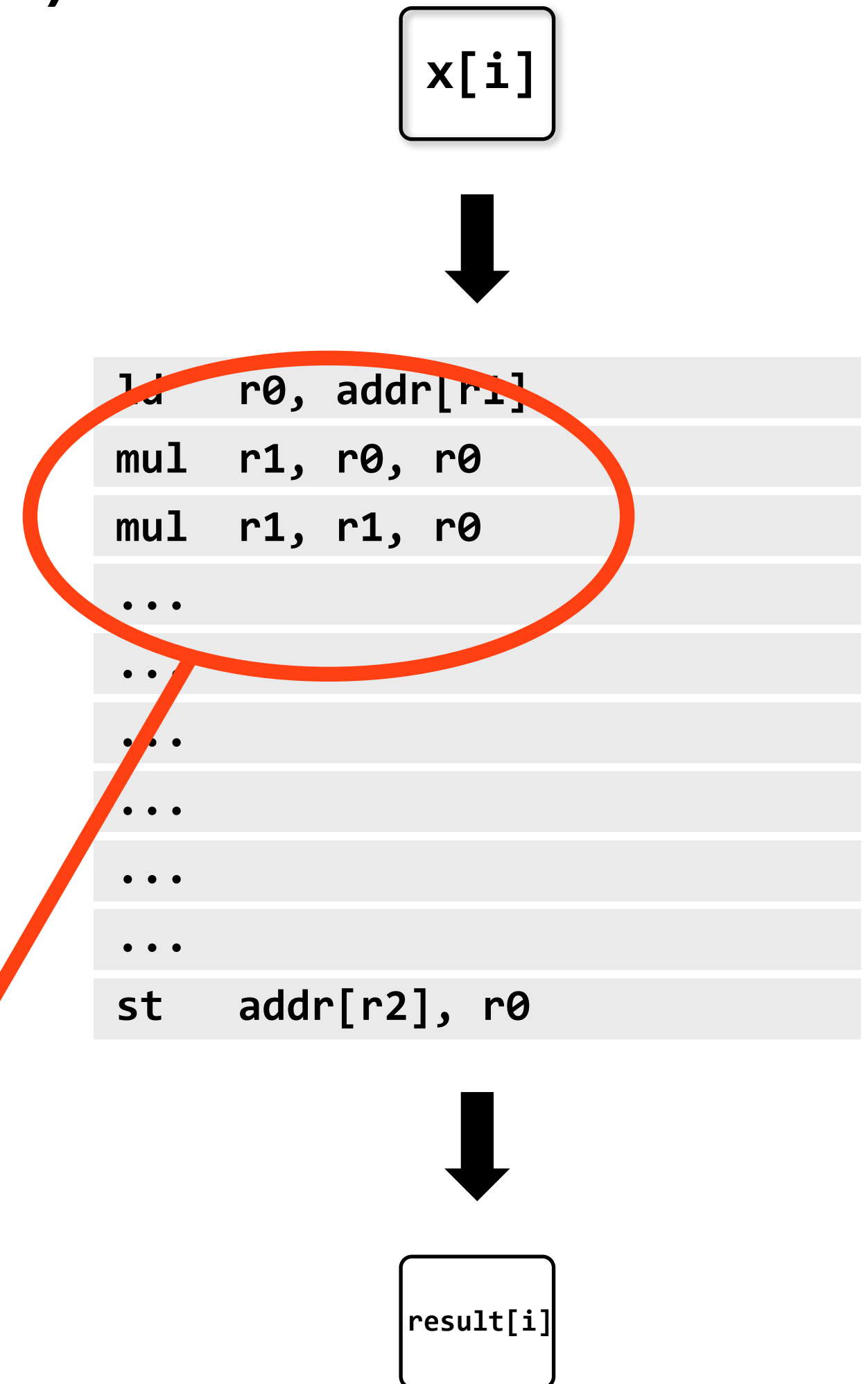
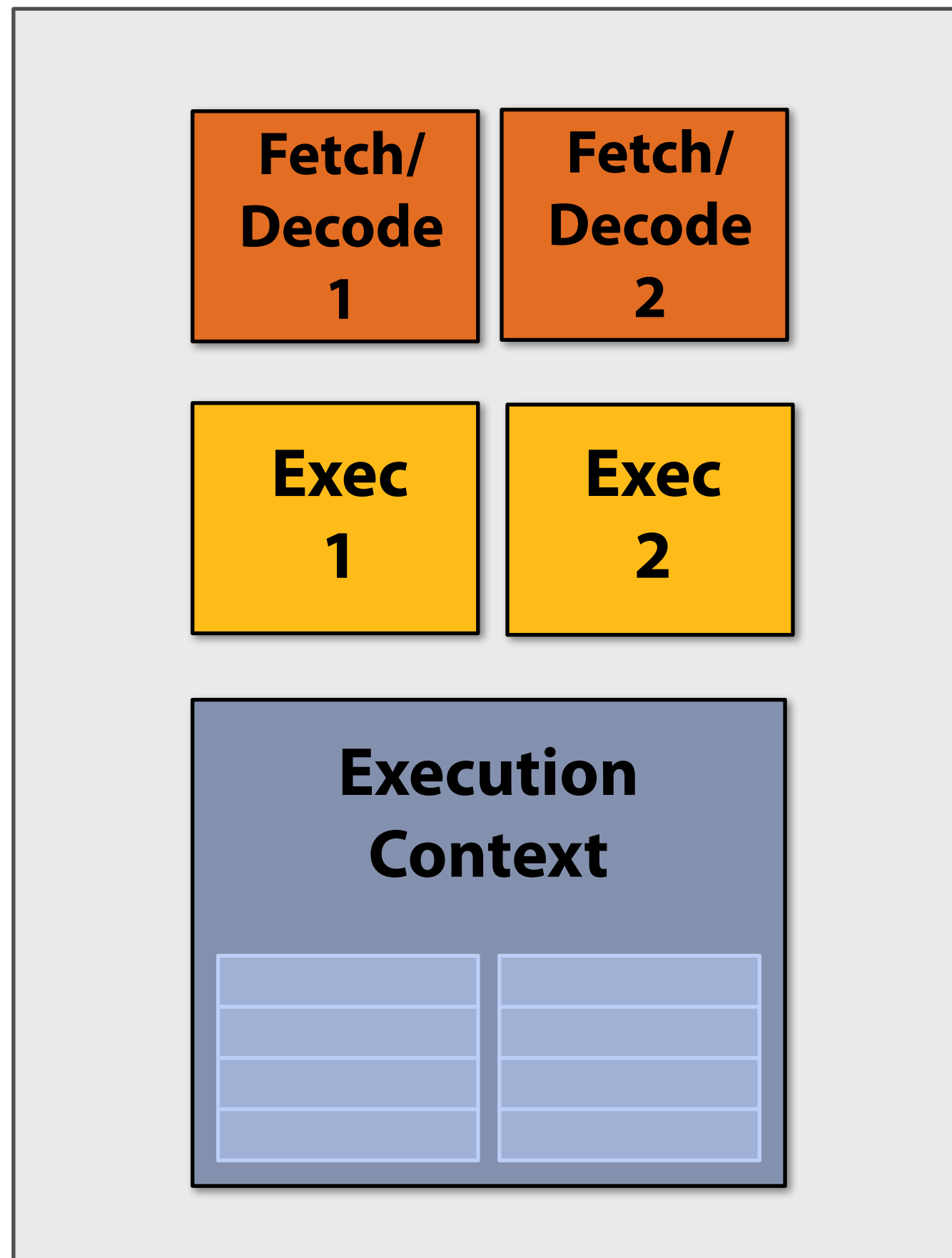
Very simple processor: execute one instruction per clock



Superscalar processor

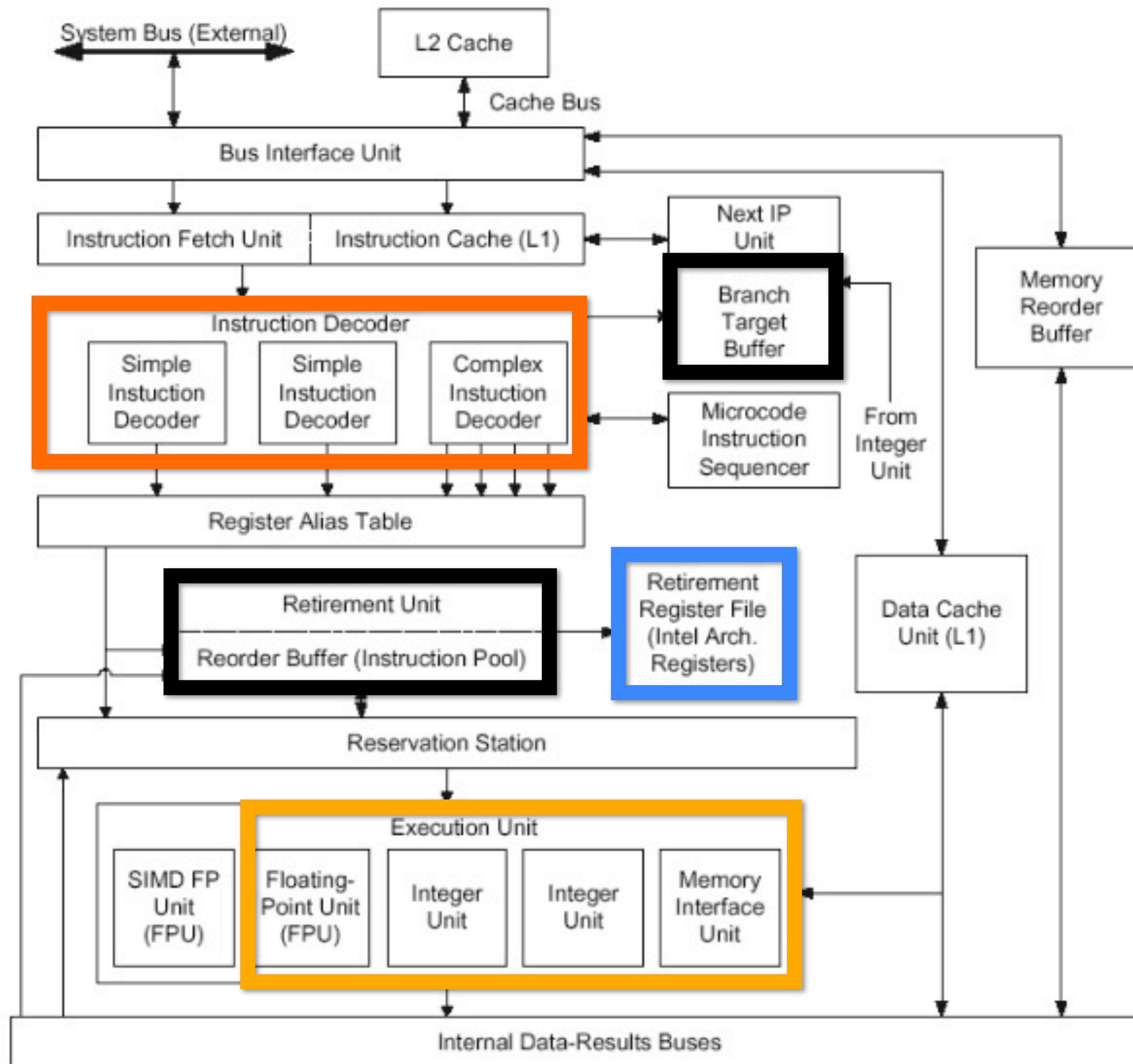
Recall from last class: instruction level parallelism (ILP)

Decode and execute two instructions per clock (if possible)



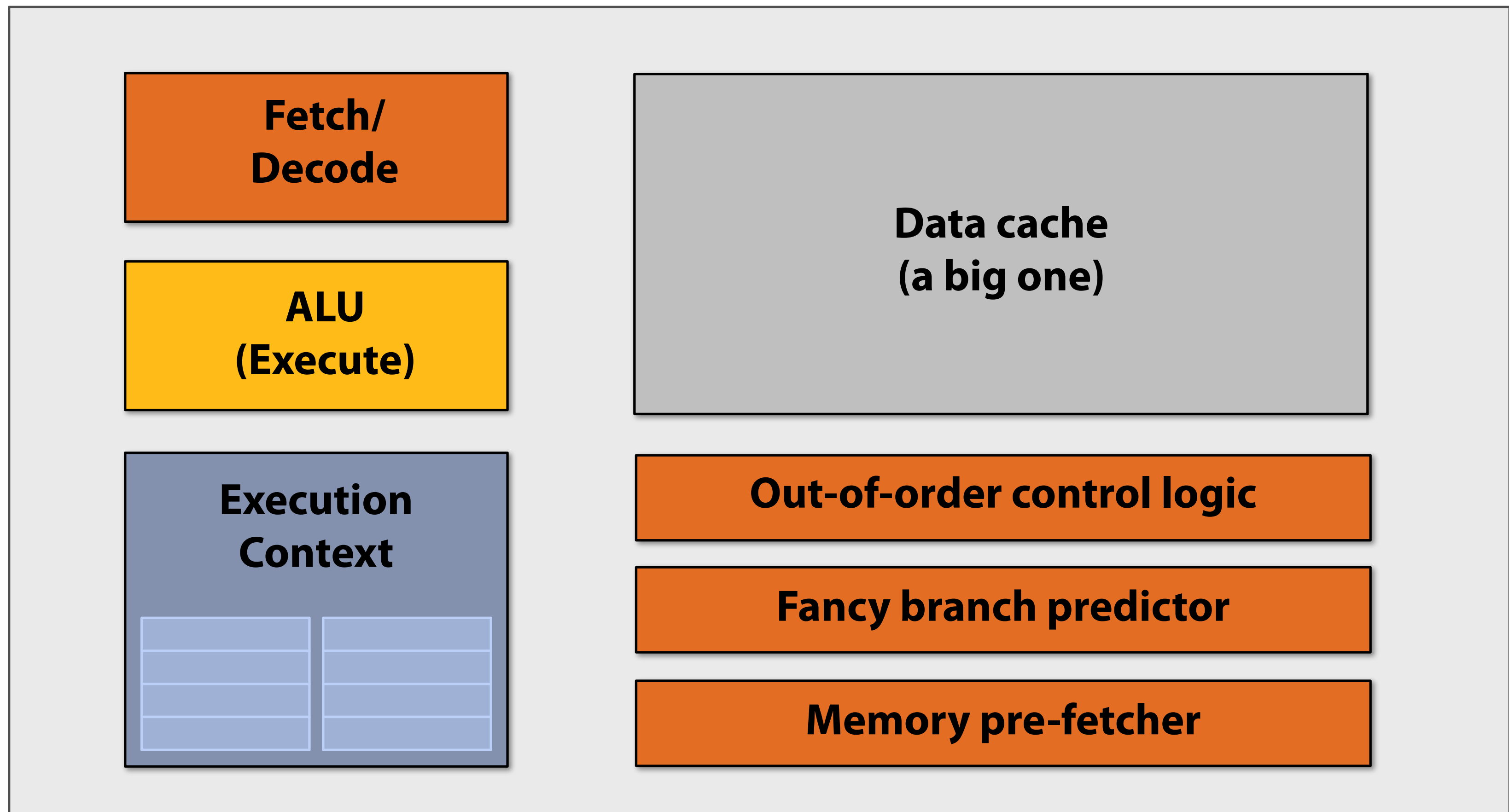
Note: No ILP exists in this region of the program

Aside: Pentium 4



CPU: pre multi-core era

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



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

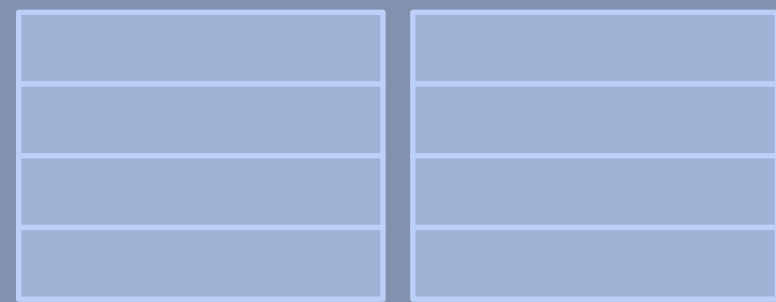
(Also: more transistors --> smaller transistors --> higher clock frequencies)

CPU: multi-core era

**Fetch/
Decode**

**ALU
(Execute)**

**Execution
Context**

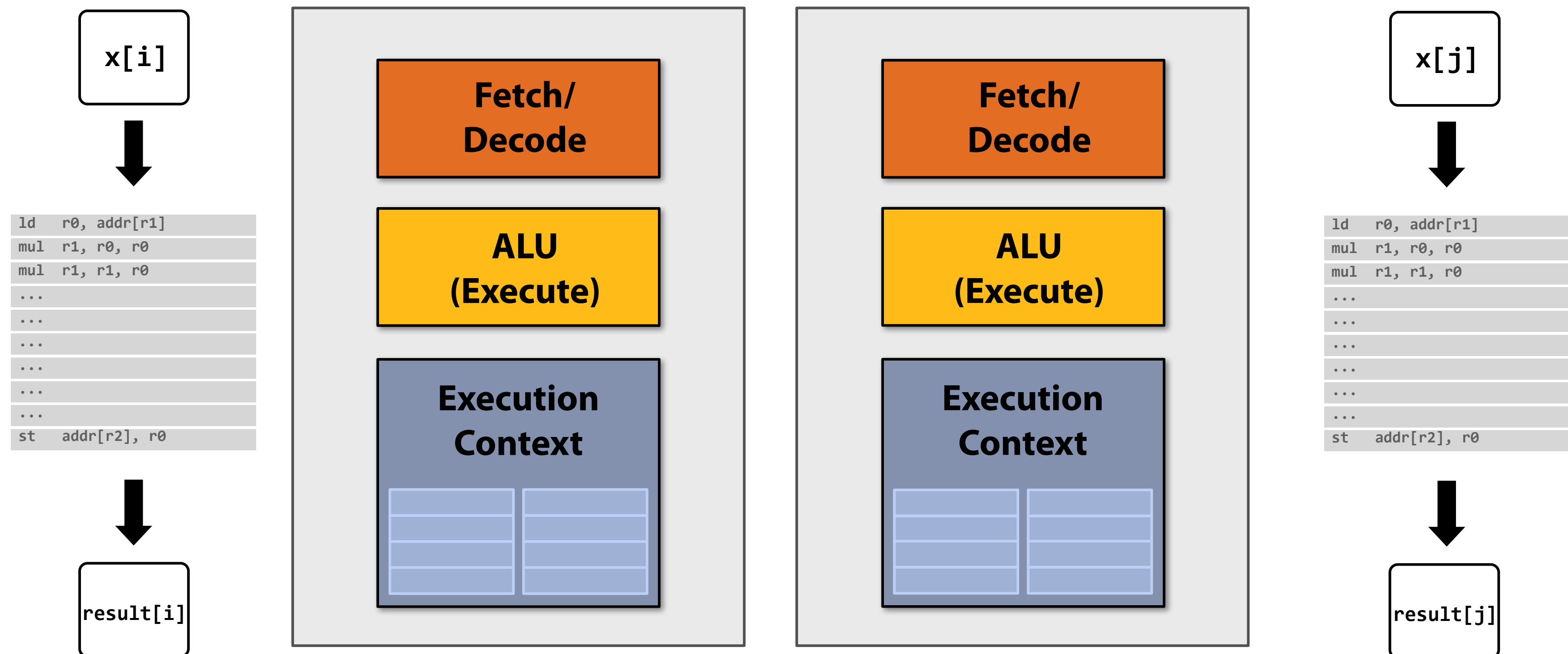


Idea #1:

Decrease sophistication of out-of-order and speculative operations

Use increasing transistor count to provide more processing cores

Two cores: compute two elements in parallel



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

But there are now two: $2 * 0.75 = 1.5$ (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 *= (j+3) * (j+4);
            sign *= -1;
        }

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

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

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 *= (j+3) * (j+4);
            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;

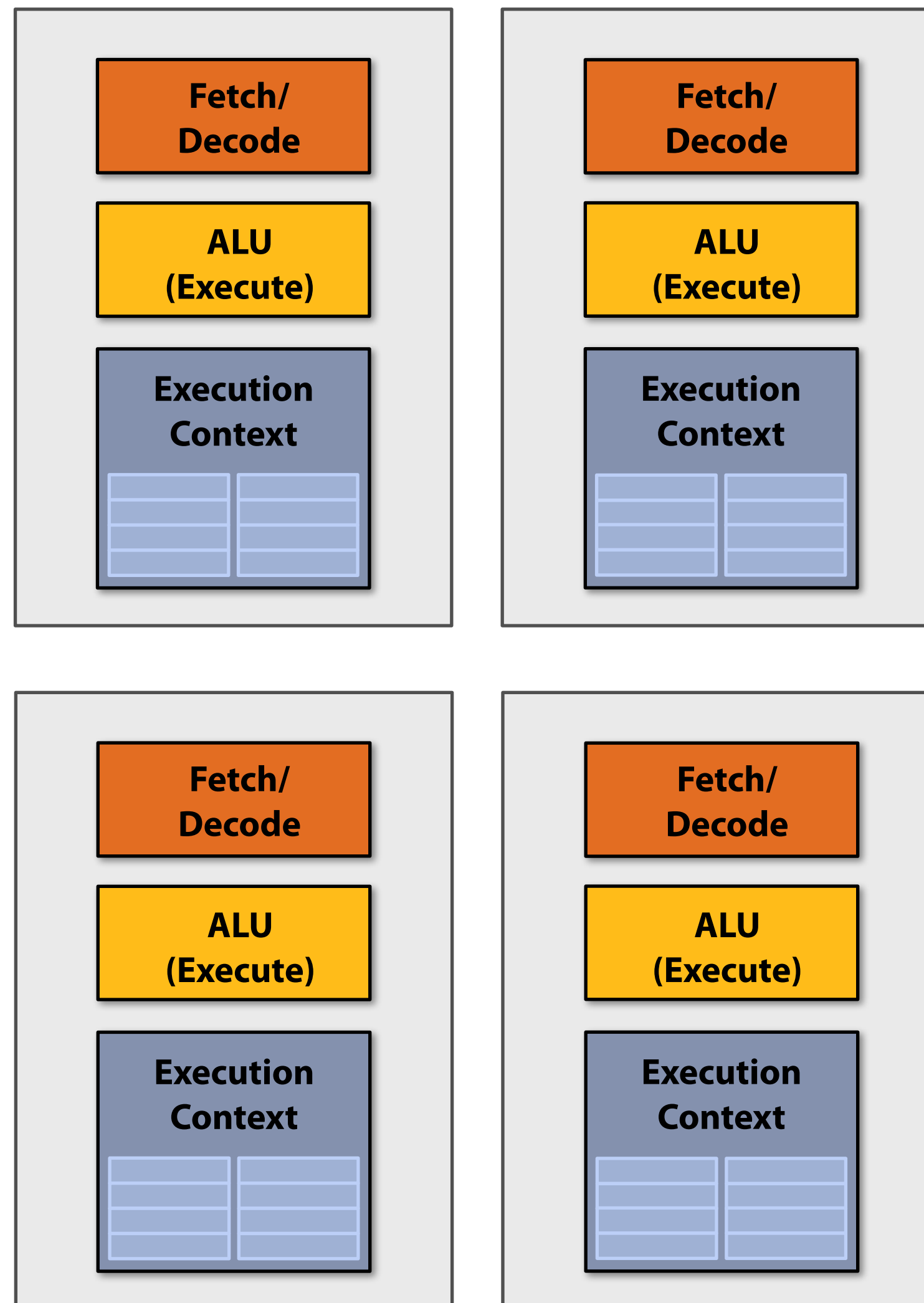
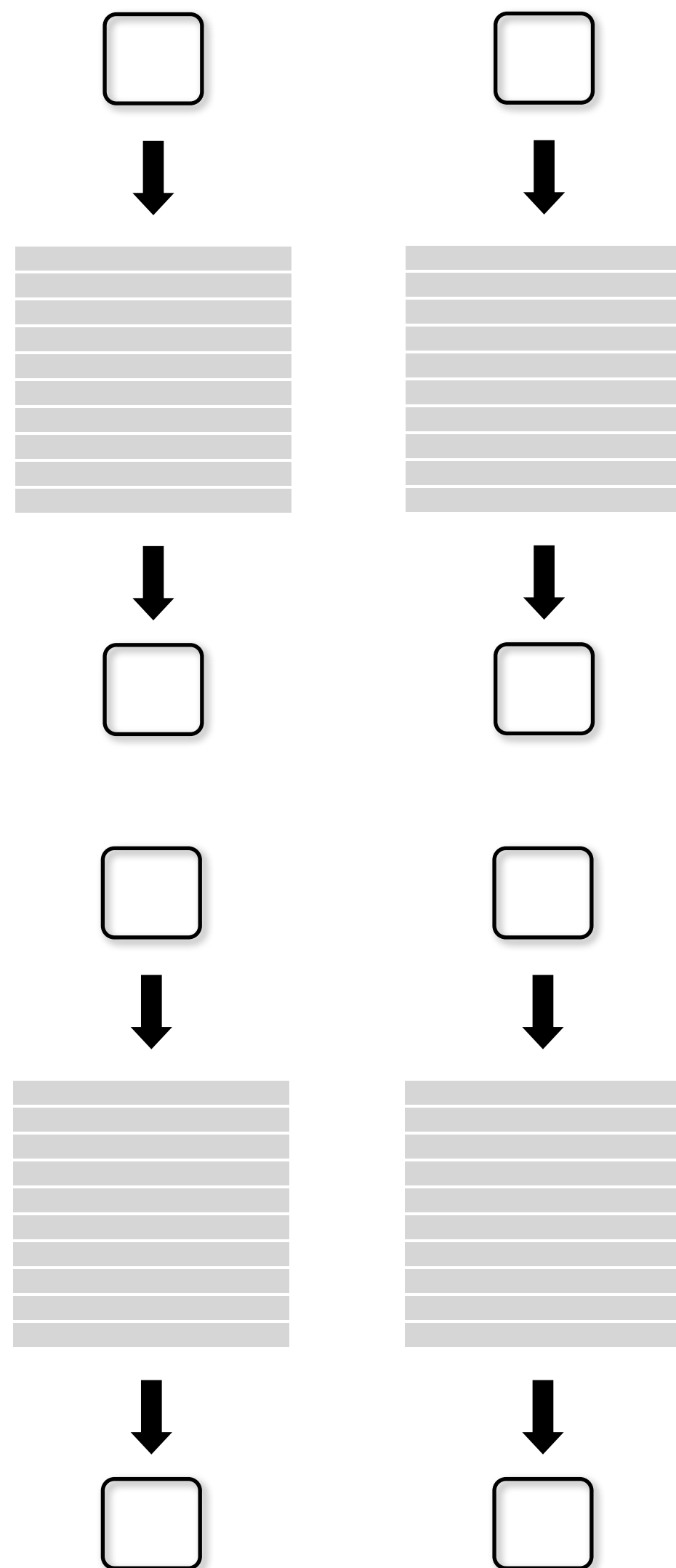
        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom
            numer *= x[i] * x[i];
            denom *= (j+3) * (j+4);
            sign *= -1;
        }

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

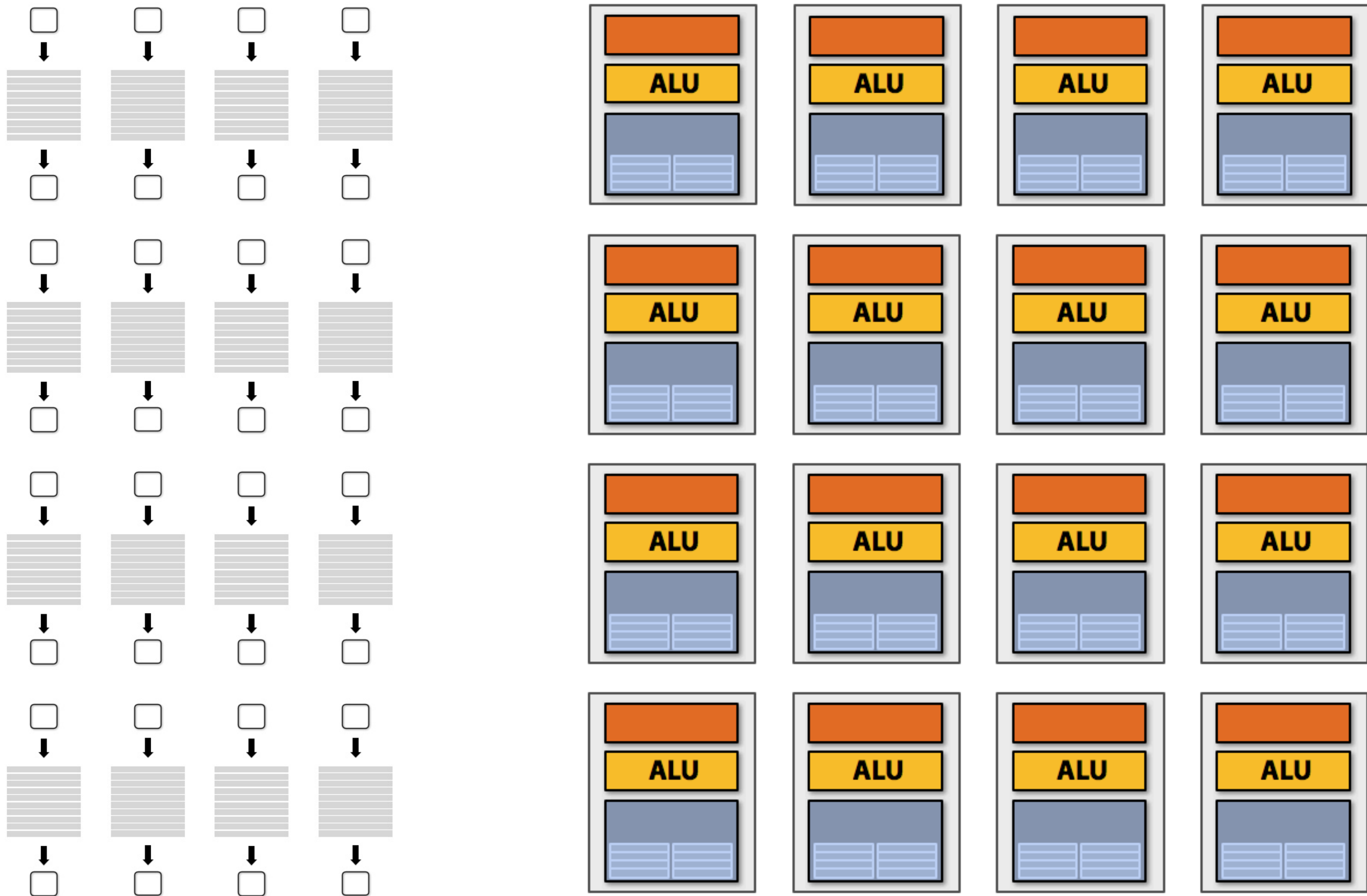
Parallelism visible to compiler

Facilitates automatic generation of threaded code

Four cores: compute four elements in parallel

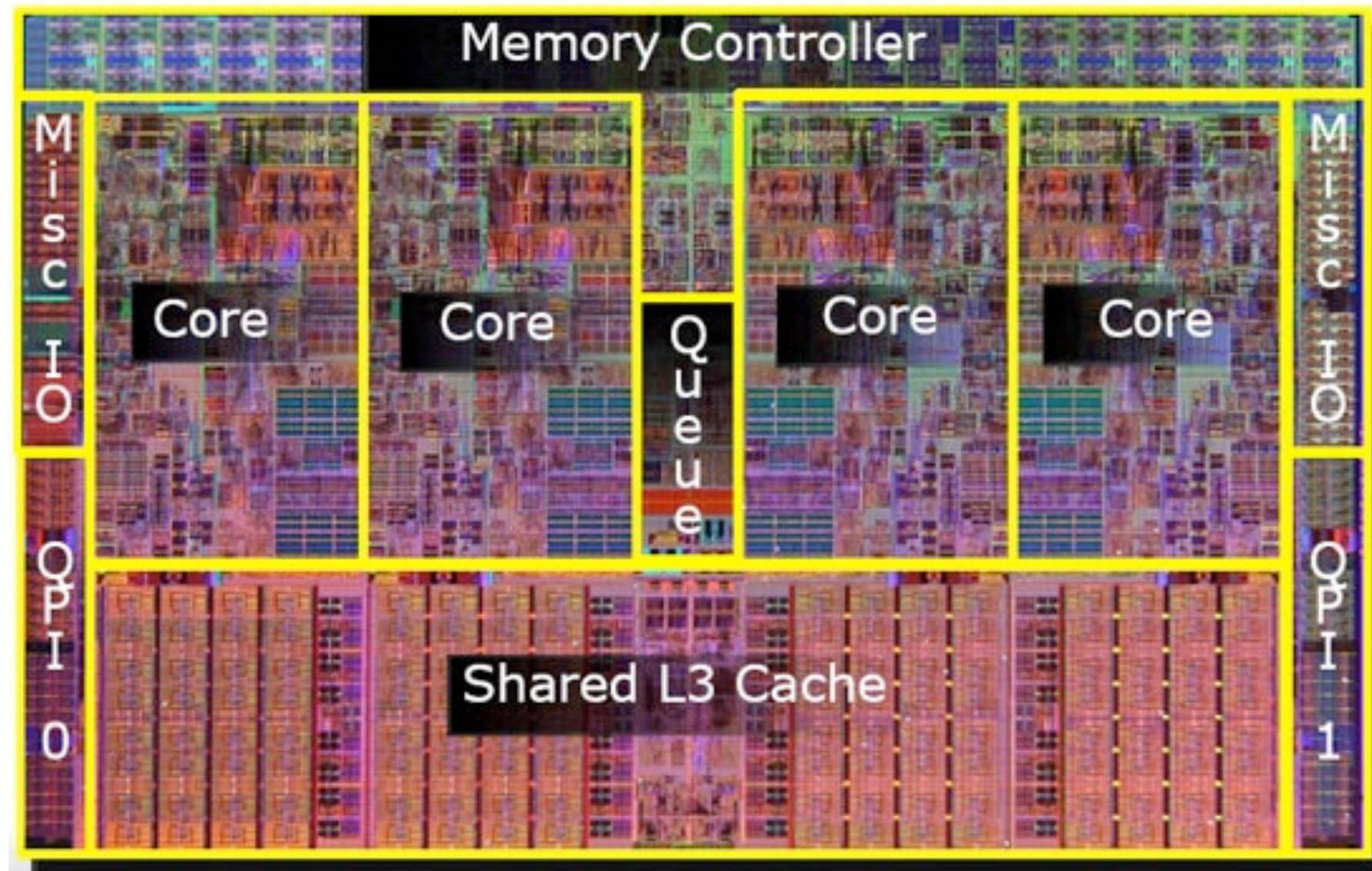


Sixteen cores: compute sixteen elements in parallel

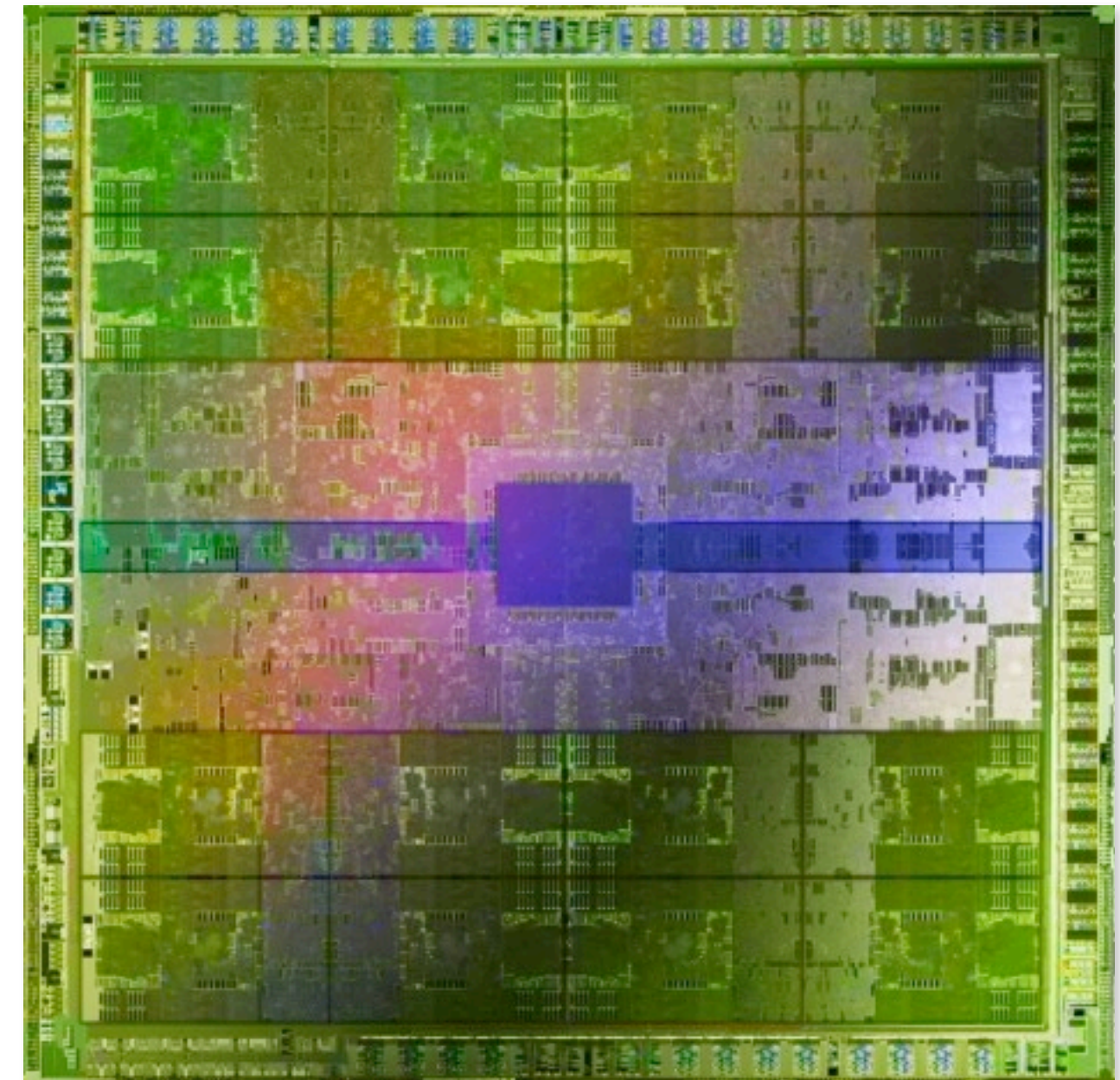


Sixteen cores, sixteen simultaneous instruction streams

Multi-core examples

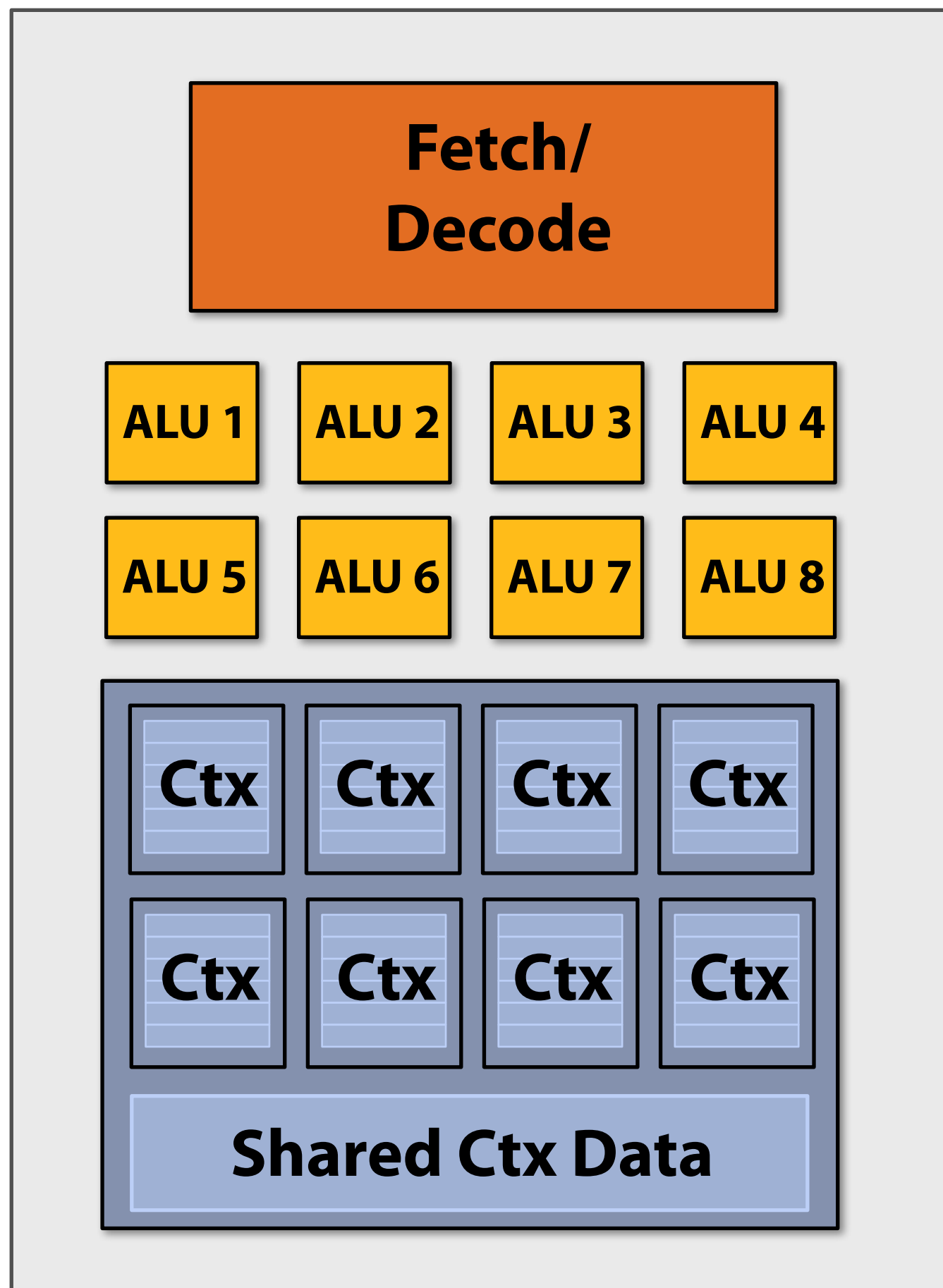


Intel Core i7 quad-core CPU (2010)



**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

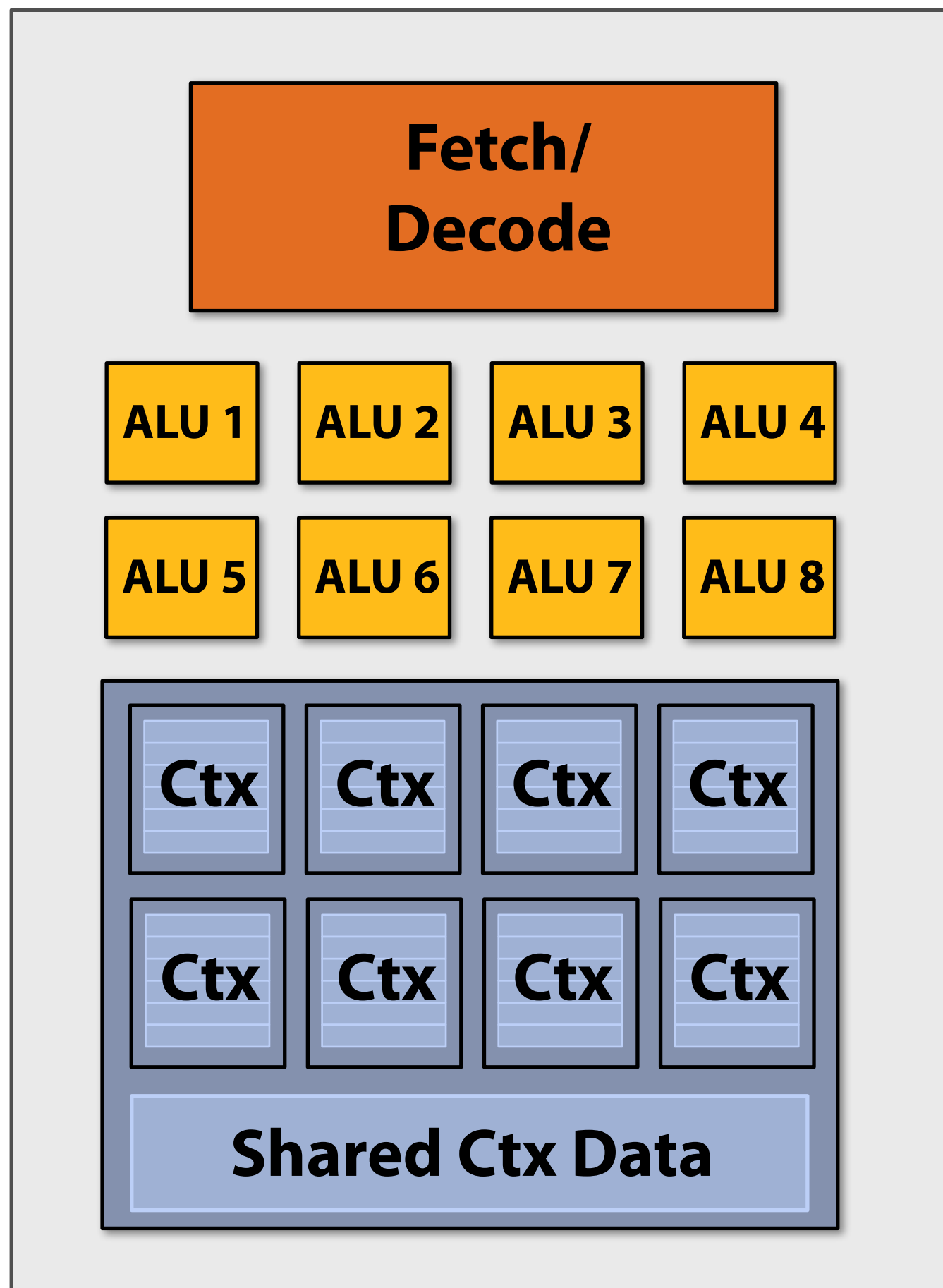
SIMD processing

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
```

Original compiled program:

Processes one fragment 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+=4)
    {
        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 *= (j+3) * (j+4);
            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* sinx)
{
    for (int i=0; i<N; i+=8)
    {
        __m256 origx = _mm256_load_ps(&x[i]);
        __m256 value = origx;
        __m256 numer = _mm256_mul_ps(origx, mm_mul_ps(origx, origx));
        __m256 denom = _mm256_broadcast_ss(6); // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            // value += sign * numer / numer
            __m256 tmp = _mm256_div_ps(_mm_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((j+3) * (j+4)));
            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)
{
    for (int i=0; i<N; i+=8)
    {
        __m256 origx = _mm256_load_ps(&x[i]);
        __m256 value = origx;
        __m256 numer = _mm256_mul_ps(origx, mm_mul_ps(origx, origx));
        __m256 denom = _mm256_broadcast_ss(6); // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            // value += sign * numer / numer
            __m256 tmp = _mm256_div_ps(_mm_mul_ps(_mm256_broadcast_ss(sign), numer), denom);
            value = _mm256_add_ps(value, tmp);

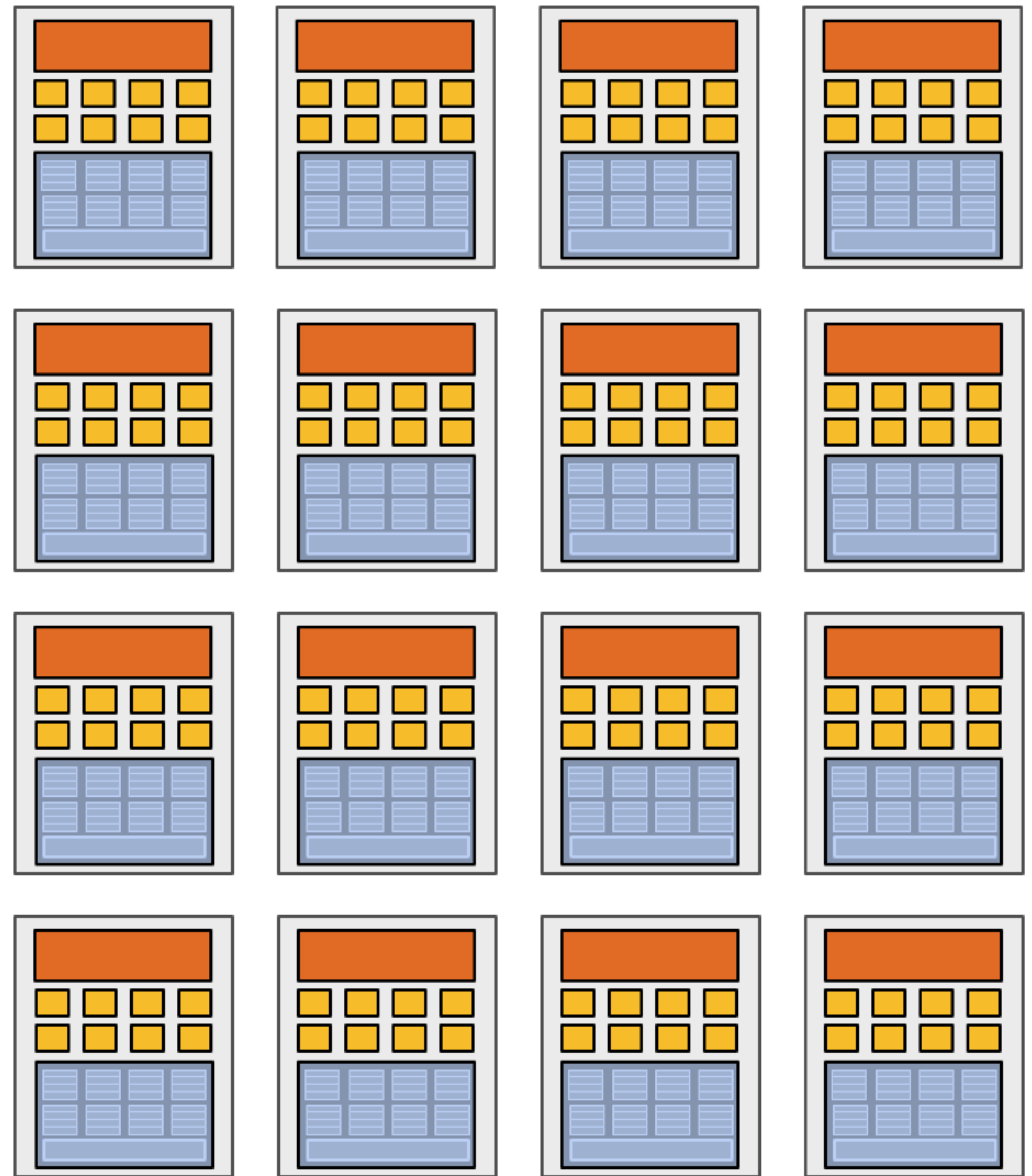
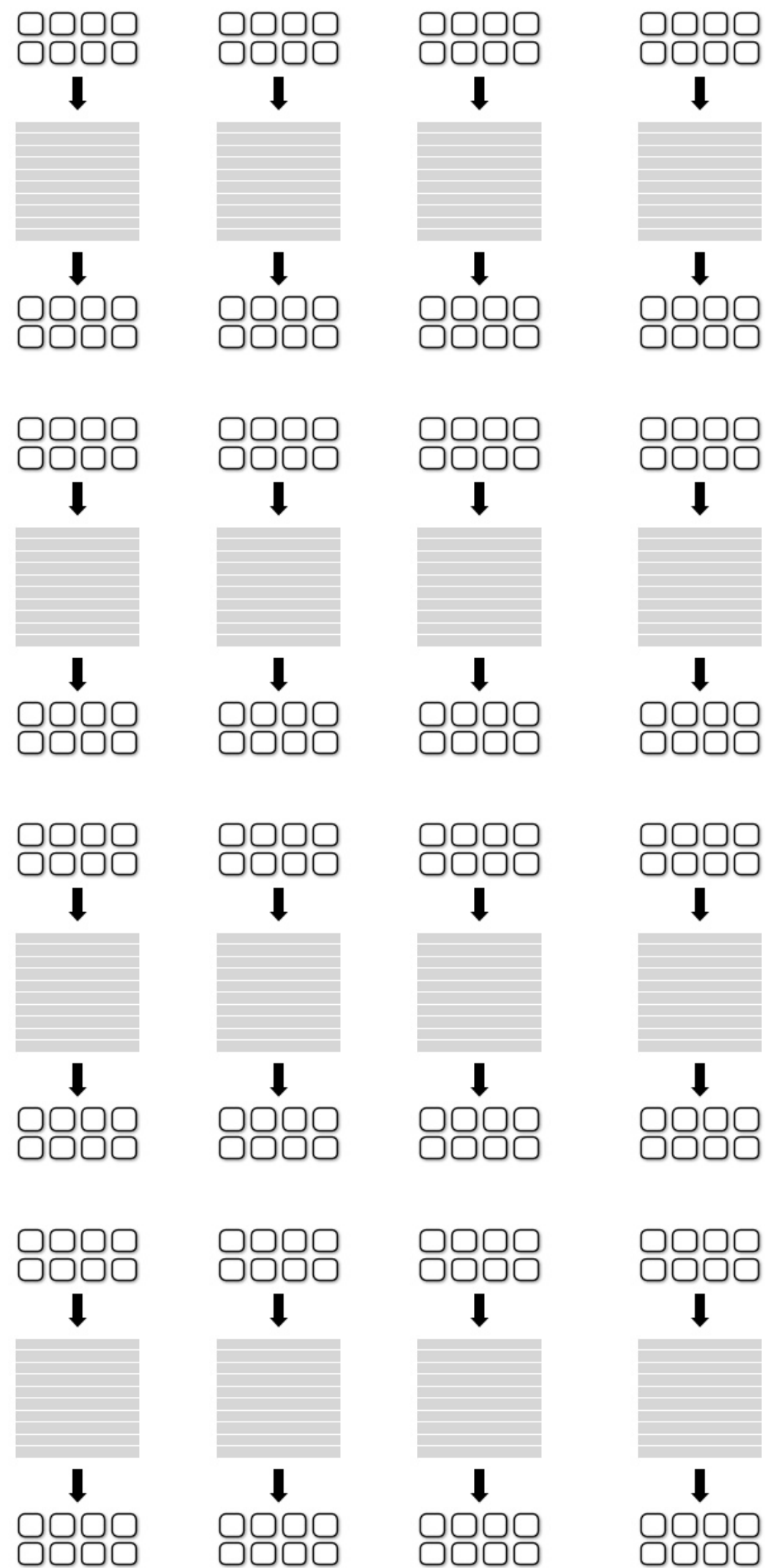
            numer = _mm256_mul_ps(numer, _mm256_mul_ps(origx, origx));
            denom = _mm256_mul_ps(denom, _mm256_broadcast_ss((j+3) * (j+4)));
            sign *= -1;
        }
        _mm256_store_ps(&sinx[i], value);
    }
}
```

```
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 *= (j+3) * (j+4);
            sign *= -1;
        }

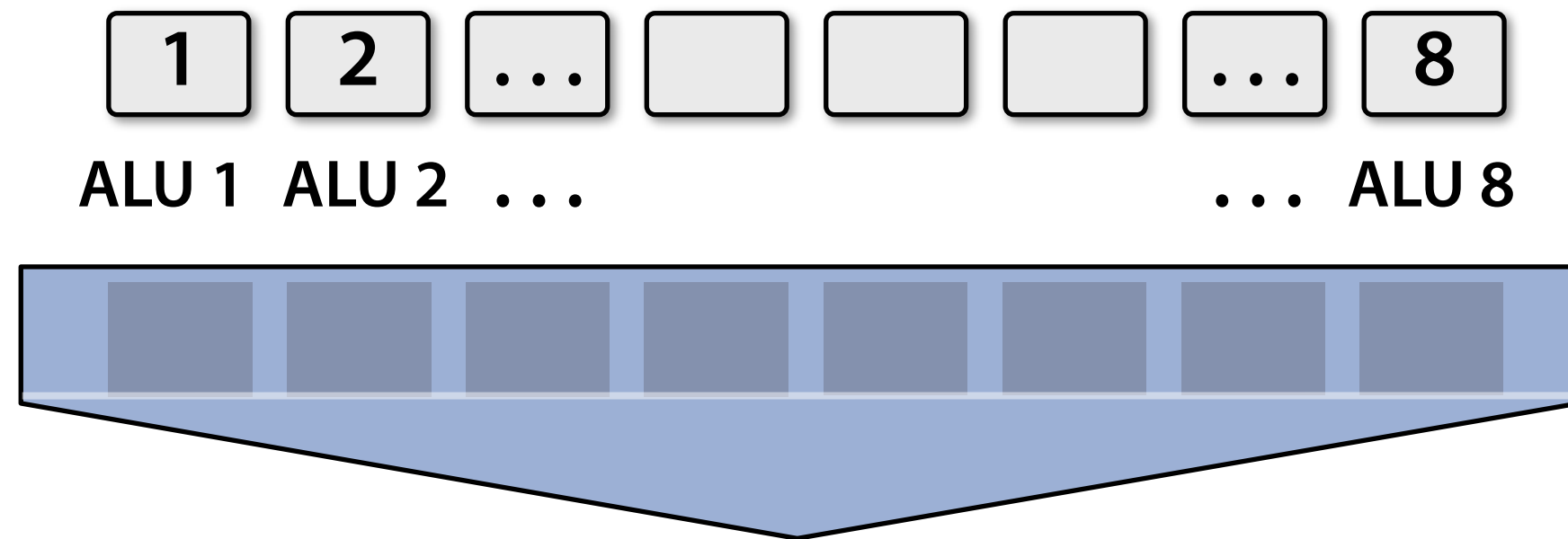
        result[i] = value;
    }
}
```

Parallelism visible to compiler

Facilitates automatic generation of vector instructions

What about conditional execution?

Time (clocks)



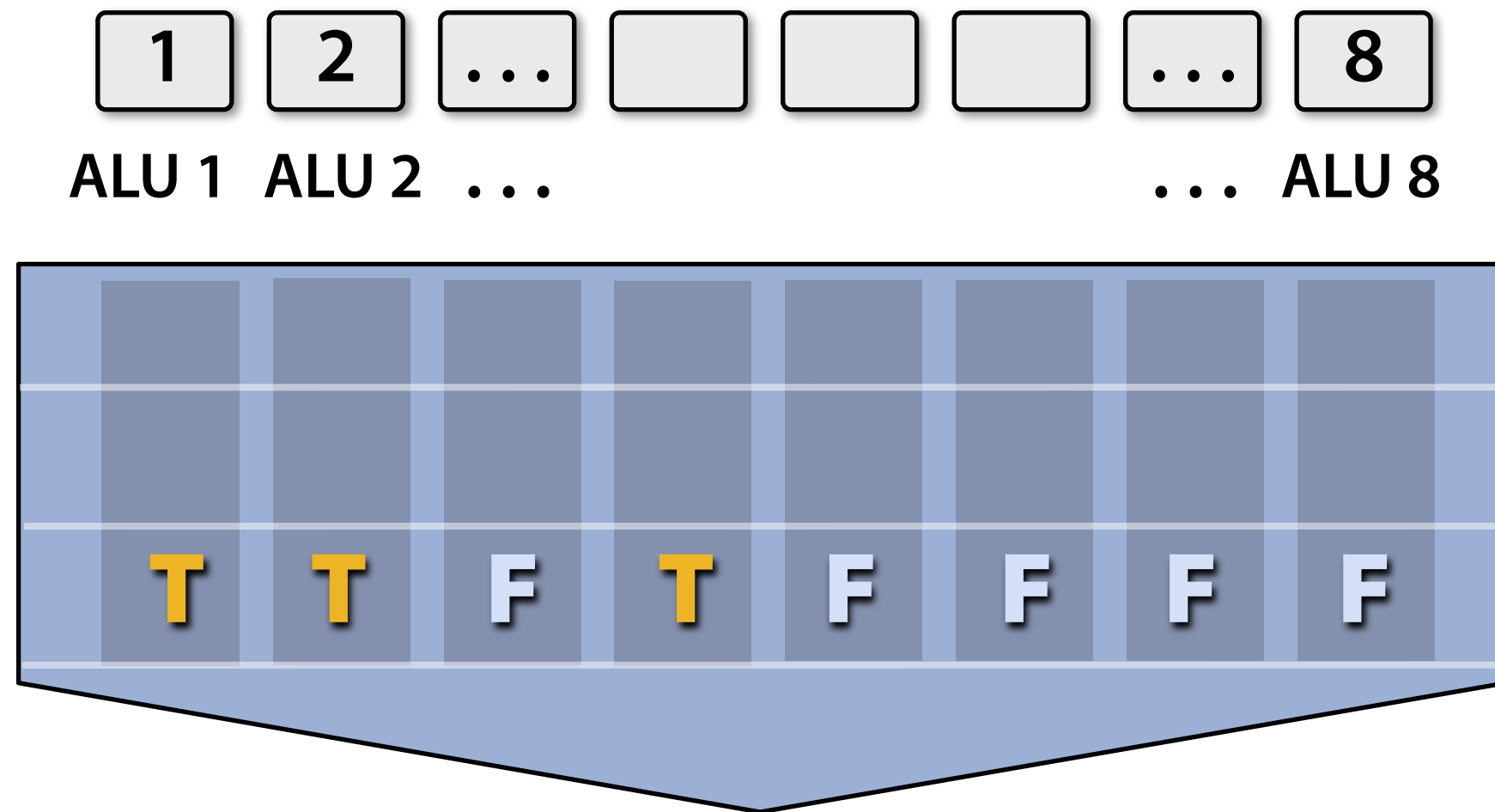
<unconditional code>

```
if (x > 0) {  
    y = exp(x, 5.f);  
    y *= Ks;  
    refl = y + Ka;  
} else {  
    x = 0;  
    refl = Ka;  
}
```

<resume unconditional code>

What about conditional execution?

Time (clocks)

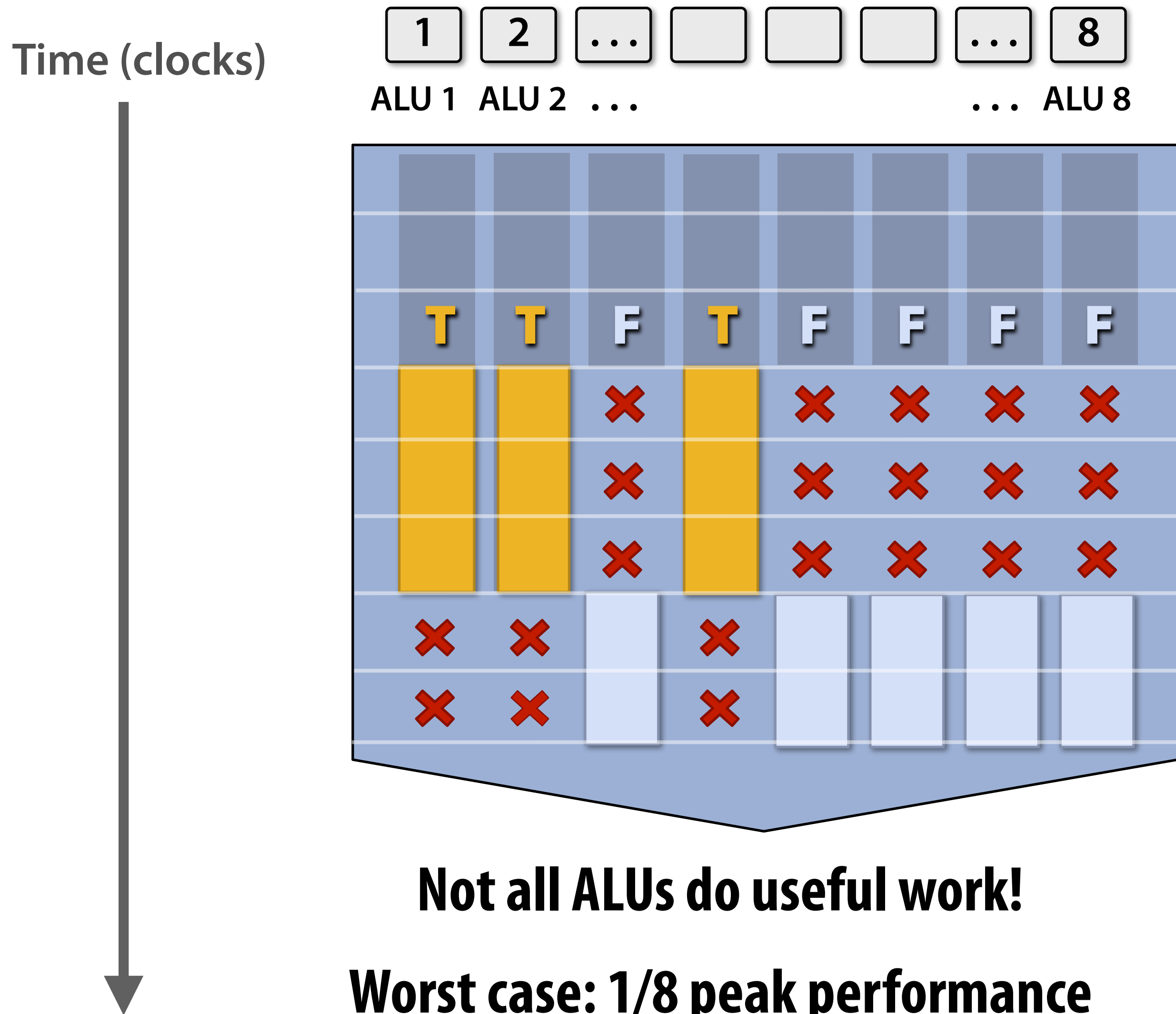


<unconditional code>

```
if (x > 0) {  
    y = exp(x, 5.f);  
    y *= Ks;  
    refl = y + Ka;  
} else {  
    x = 0;  
    refl = Ka;  
}
```

<resume unconditional code>

Mask (discard) output of ALU



<unconditional code>

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

```
    y = exp(x, 5.f);
```

```
    y *= Ks;
```

```
    refl = y + Ka;
```

```
} else {
```

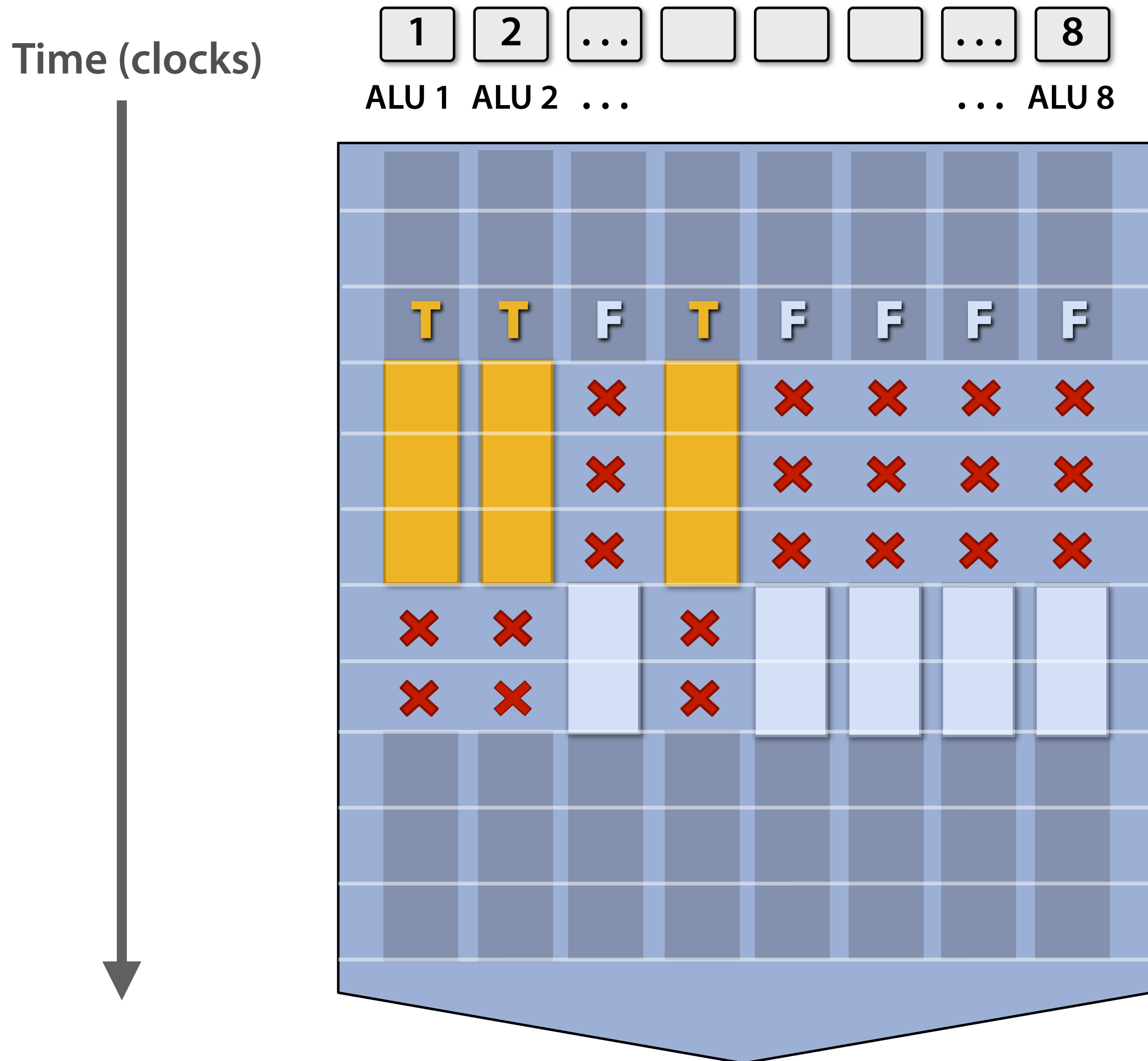
```
    x = 0;
```

```
    refl = Ka;
```

```
}
```

<resume unconditional code>

After branch: continue at full performance



```

<unconditional code>

if (x > 0) {
    y = exp(x, 5.f);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional code>
    
```

Terminology

■ “Coherent” execution

- All elements executed upon simultaneously use the same instruction sequence
- Coherent execution is required for efficient use of SIMD processing resources

■ “Divergent” execution

- A lack of coherence

■ Not to be confused 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)**
 - **New in 2011**
- **Instructions generated by compiler**
 - **Parallelism explicitly given by programmer using intrinsics**
 - **Parallelism conveyed by parallel language semantics (e.g., `forall` example)**
 - **Parallelism inferred by dependency analysis of loops (not that great)**
- **“Explicit SIMD”: SIMD parallelization performed at compile time**
 - **Can inspect program binary and see instructions (`vstoreps`, `vmulps`, etc.)**

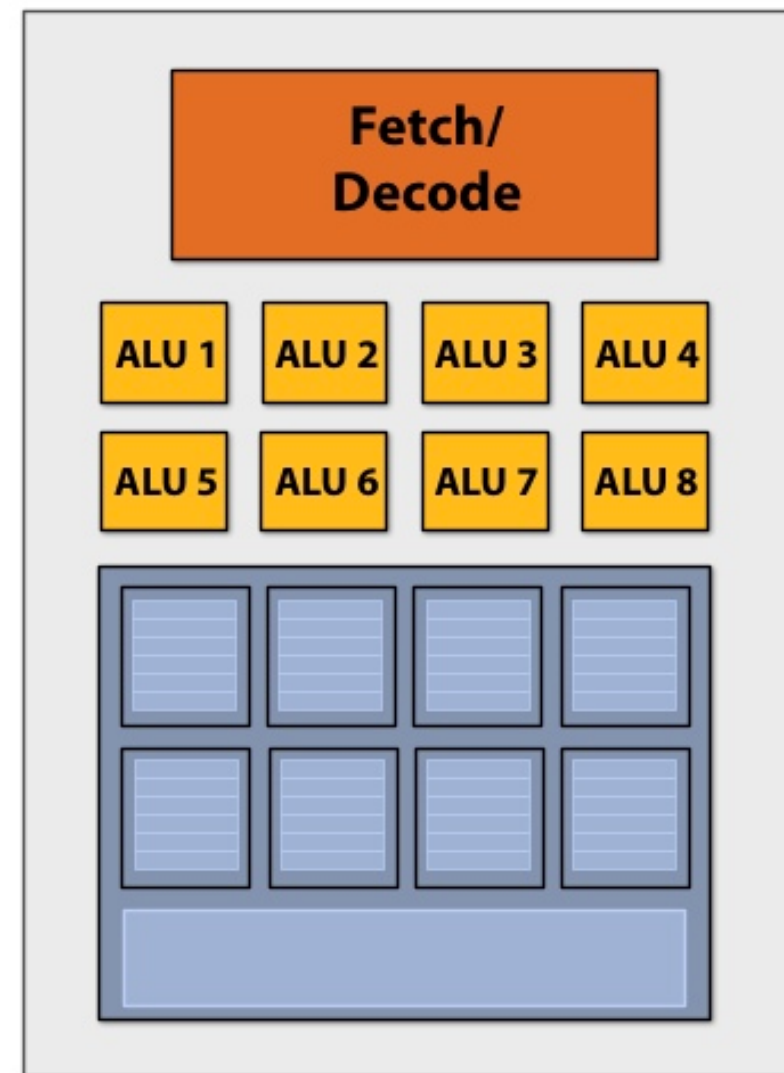
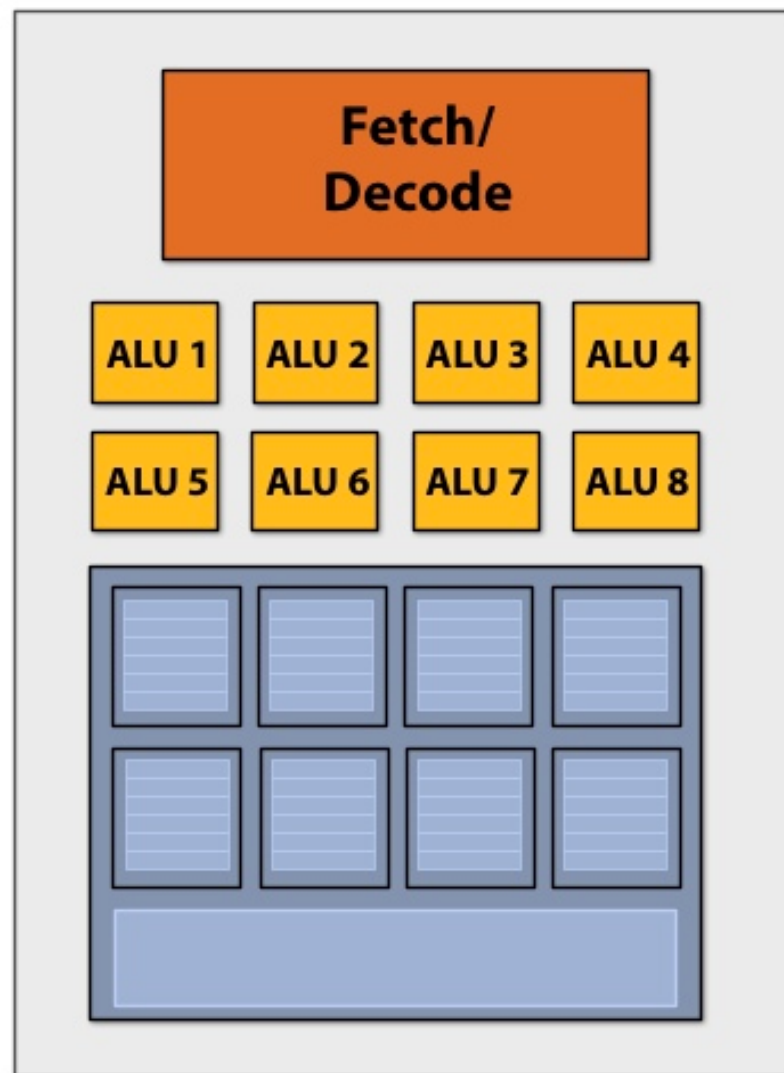
SIMD execution on modern GPUs

- **“Implicit SIMD”**
 - **Compiler generates a scalar binary**
 - **Program is *always run* with N instances: `run(my_function, N)`**
 - **N much greater than number of SIMD ALUs in a core**
 - **Hardware maps instances to ALUs**

- **SIMD width of modern GPUs is 32 to 64 floats ****
 - **Divergence can be a big issue**

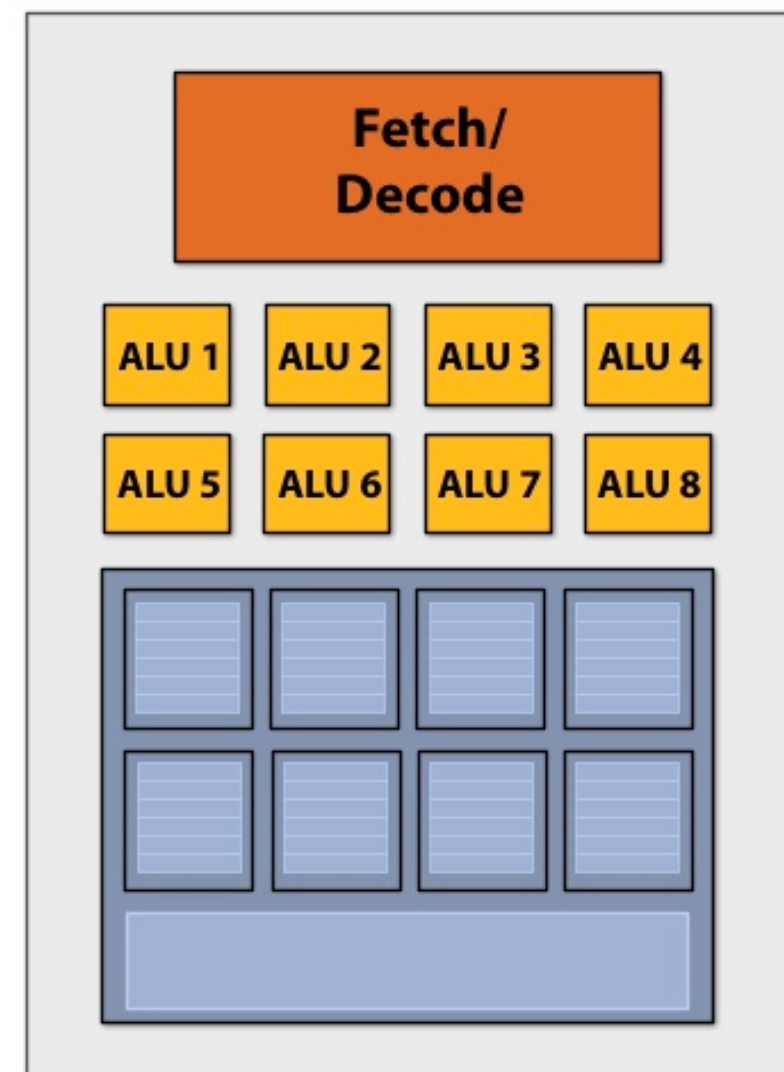
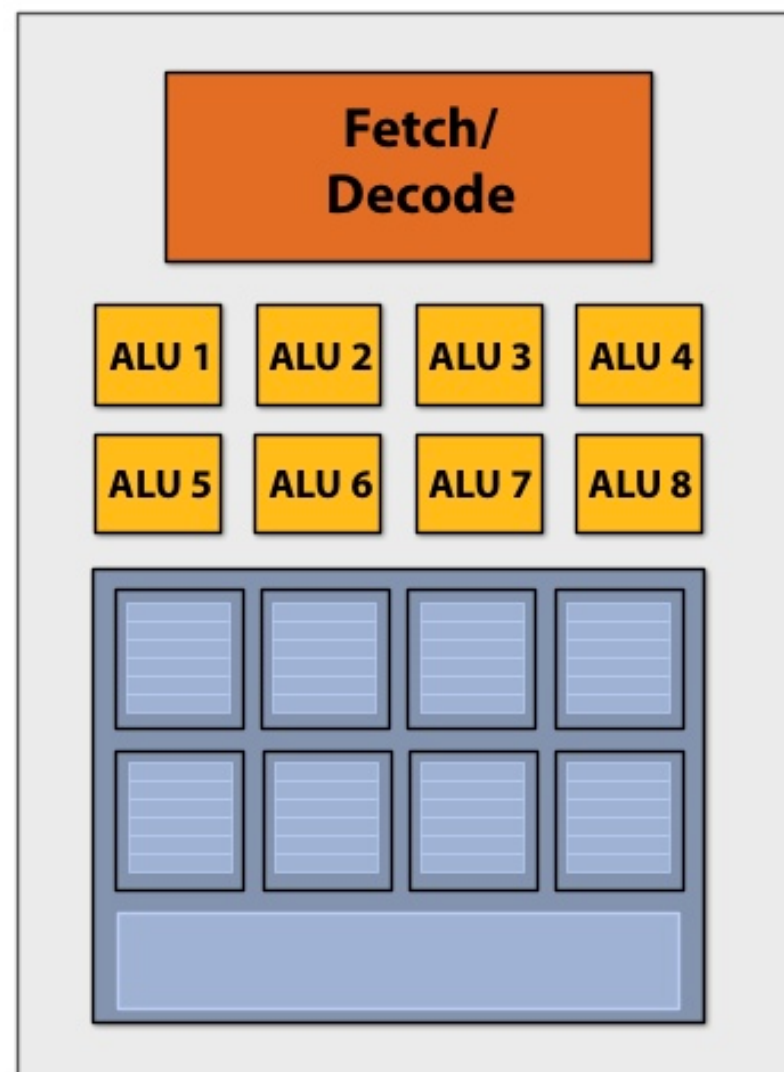
**** It's a little more complicated than that, but if you think about it this way it's fine**

Example: Intel Core i7 (Sandy Bridge)



4 cores

8 SIMD ALUs per core

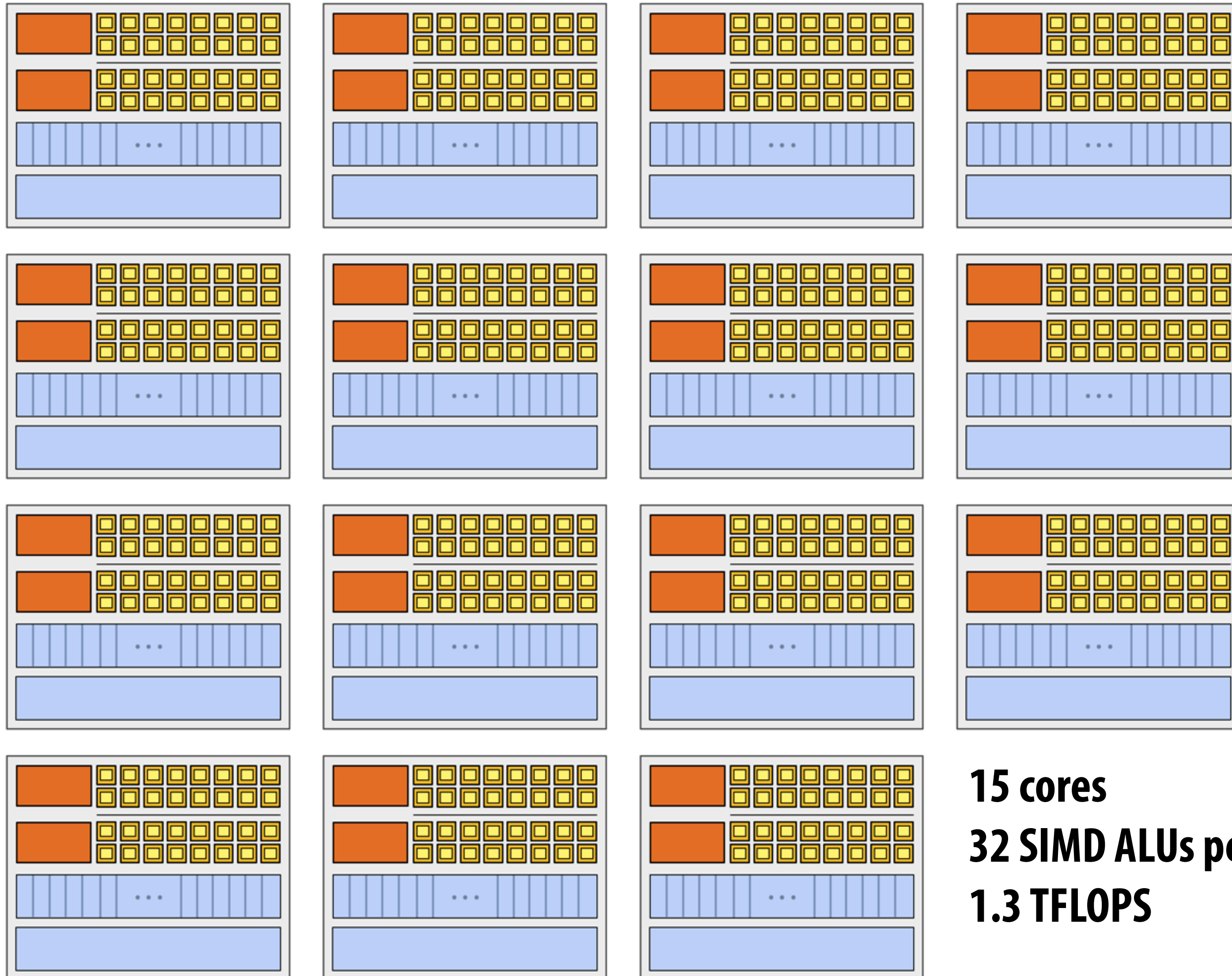


Machines in GHC 5205:

4 cores

4 SIMD ALUs per core

Example: NVIDIA GTX 480



15 cores
32 SIMD ALUs per core
1.3 TFLOPS

(in GHC 5205)

Summary: parallel execution

■ Several types of parallel execution in modern processors

- **Multi-core: use multiple cores**
 - **Provides thread-level parallelism: completely different instruction streams**
 - **Software decides when to create threads**
- **SIMD: use multiple ALUs controlled by same instruction stream (within a core)**
 - **Efficient: control amortized over many ALUs**
 - **Vectorization be done by compiler or by HW**
 - **[Lack of] dependencies known prior to execution (declared or inferred)**
- **Superscalar: exploit ILP. Process different instructions from the same instruction stream in parallel (within a core)**
 - **Automatically by the hardware, dynamically during execution (not programmer visible)**

Not addressed
further in this
class

Part 2: 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

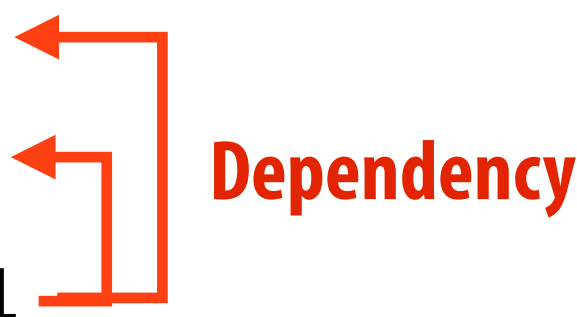
- **The rate at which the memory system can provide data to a processor**
- **Example: 20 GB/s**

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
```

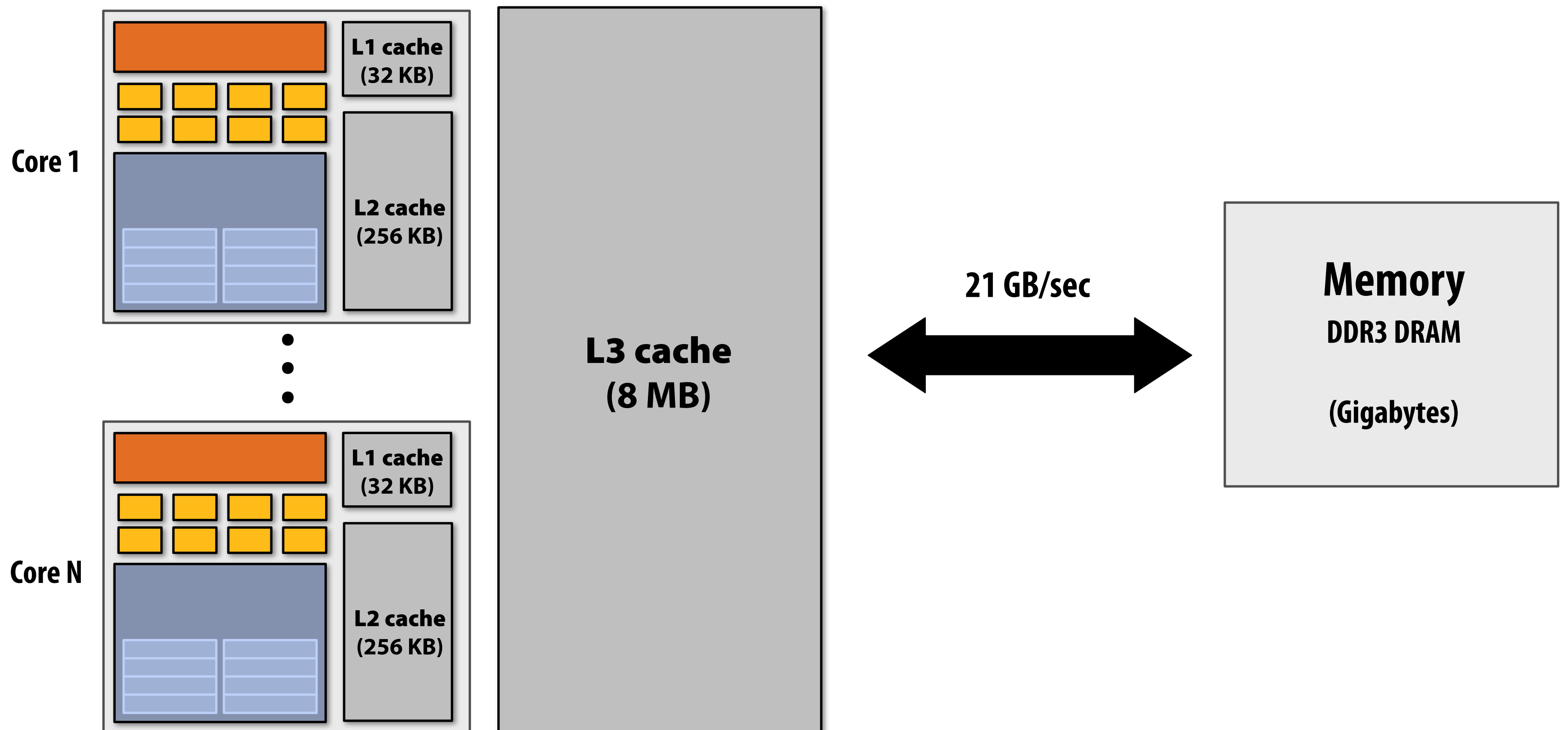


Dependency

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

Caches: reduce stalls (reduce latency)

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



Prefetching: reduces stalls (hides latency)

- All modern CPUs have logic for prefetching data into caches
 - 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

...

...

...

...

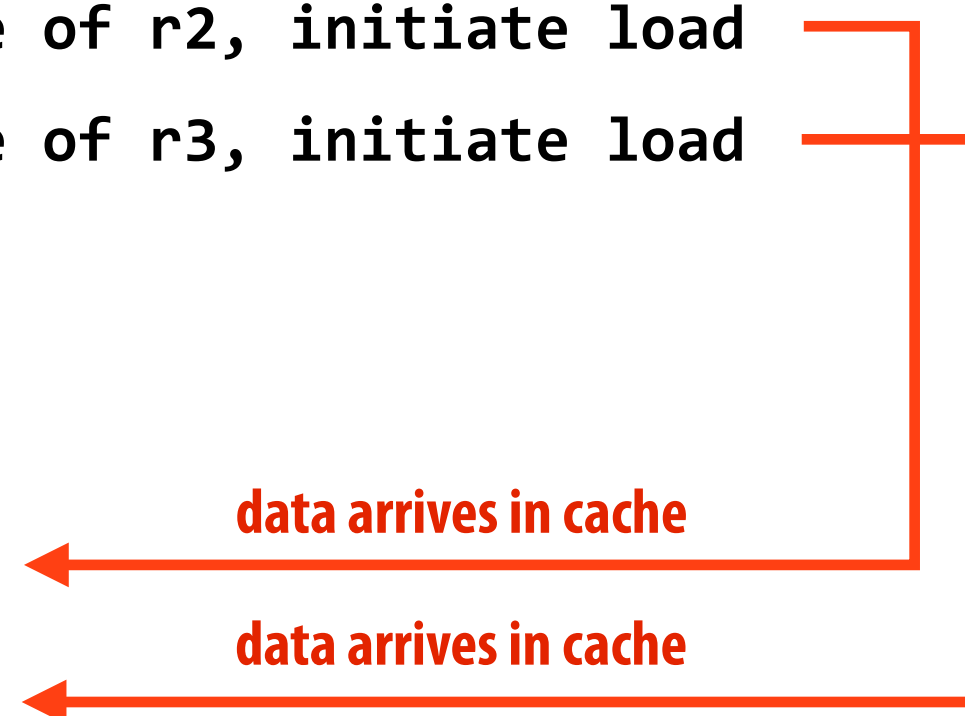
...

...

ld r0 mem[r2]

ld r1 mem[r3]

add r0, r0, r1



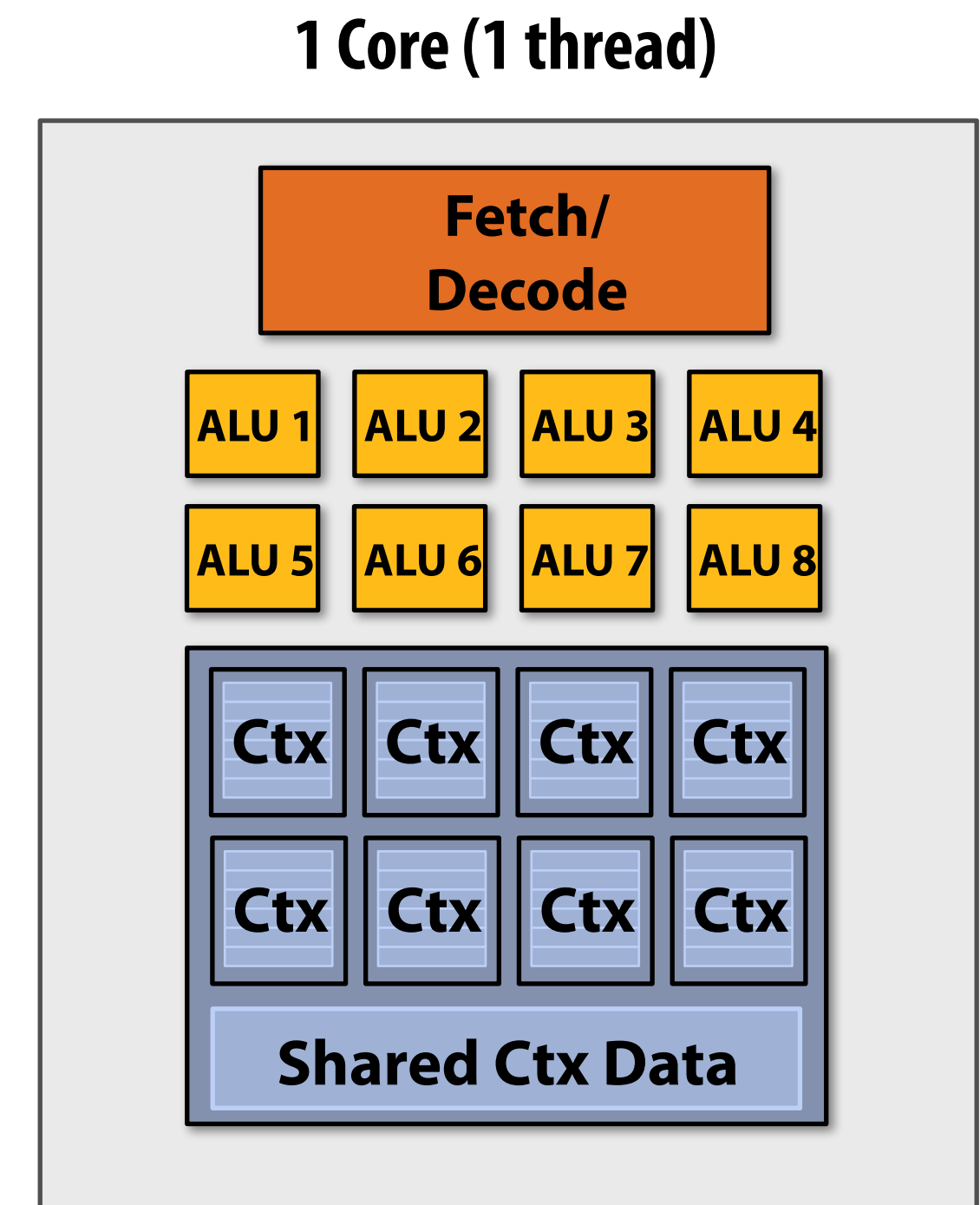
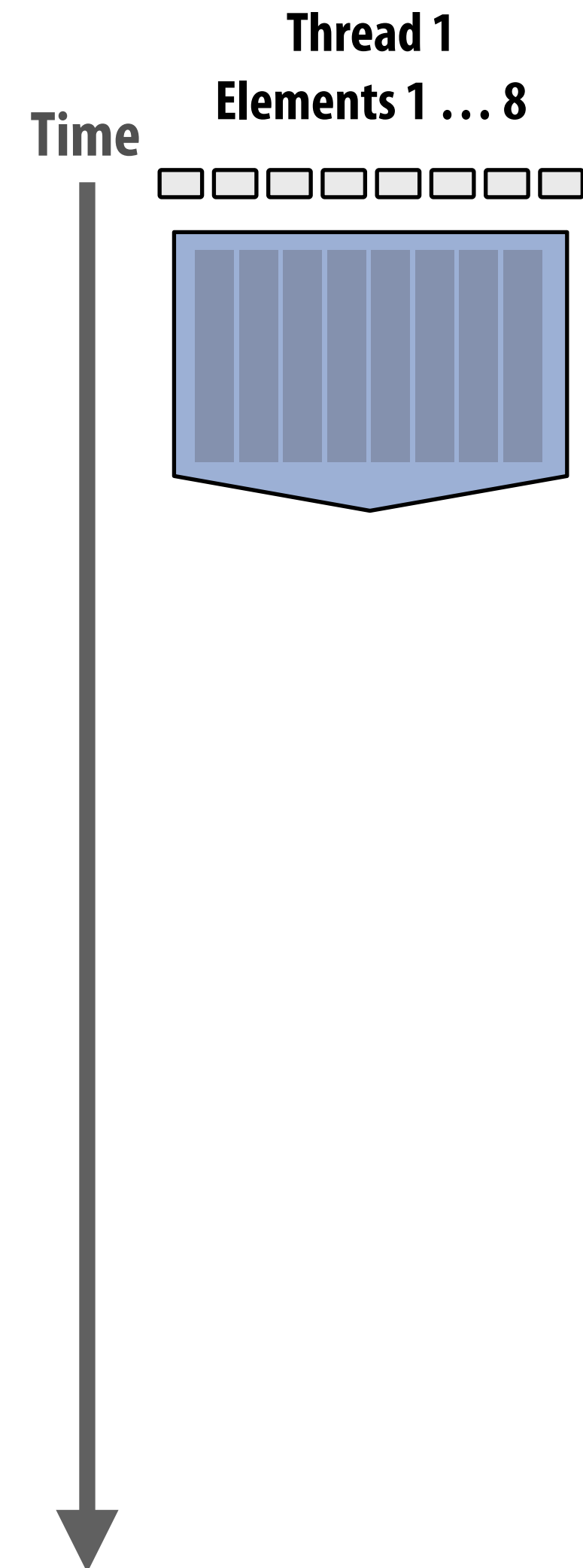
Note: Prefetching can also reduce performance if guess is wrong (hogs bandwidth, pollutes caches)

(more detail later in course)

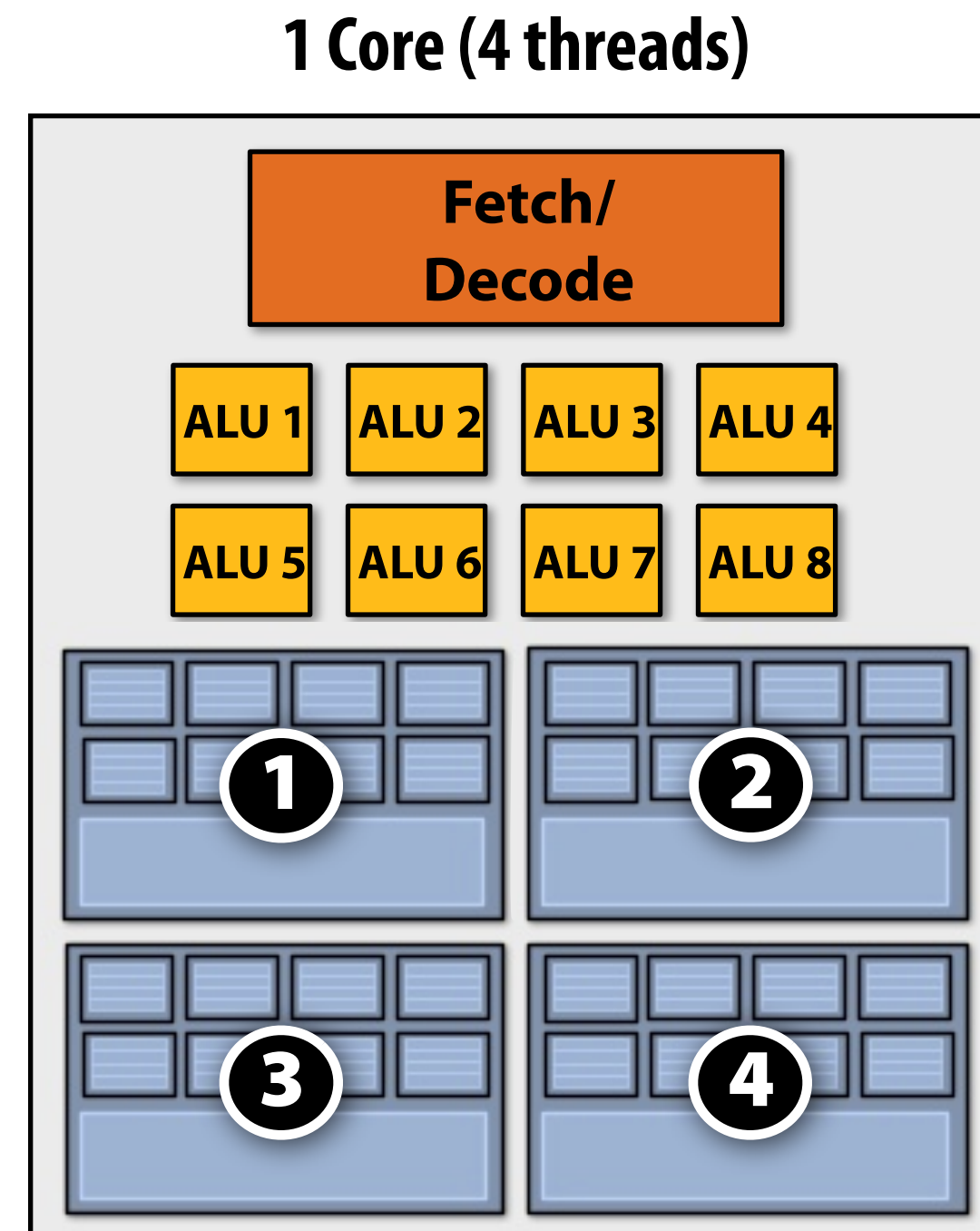
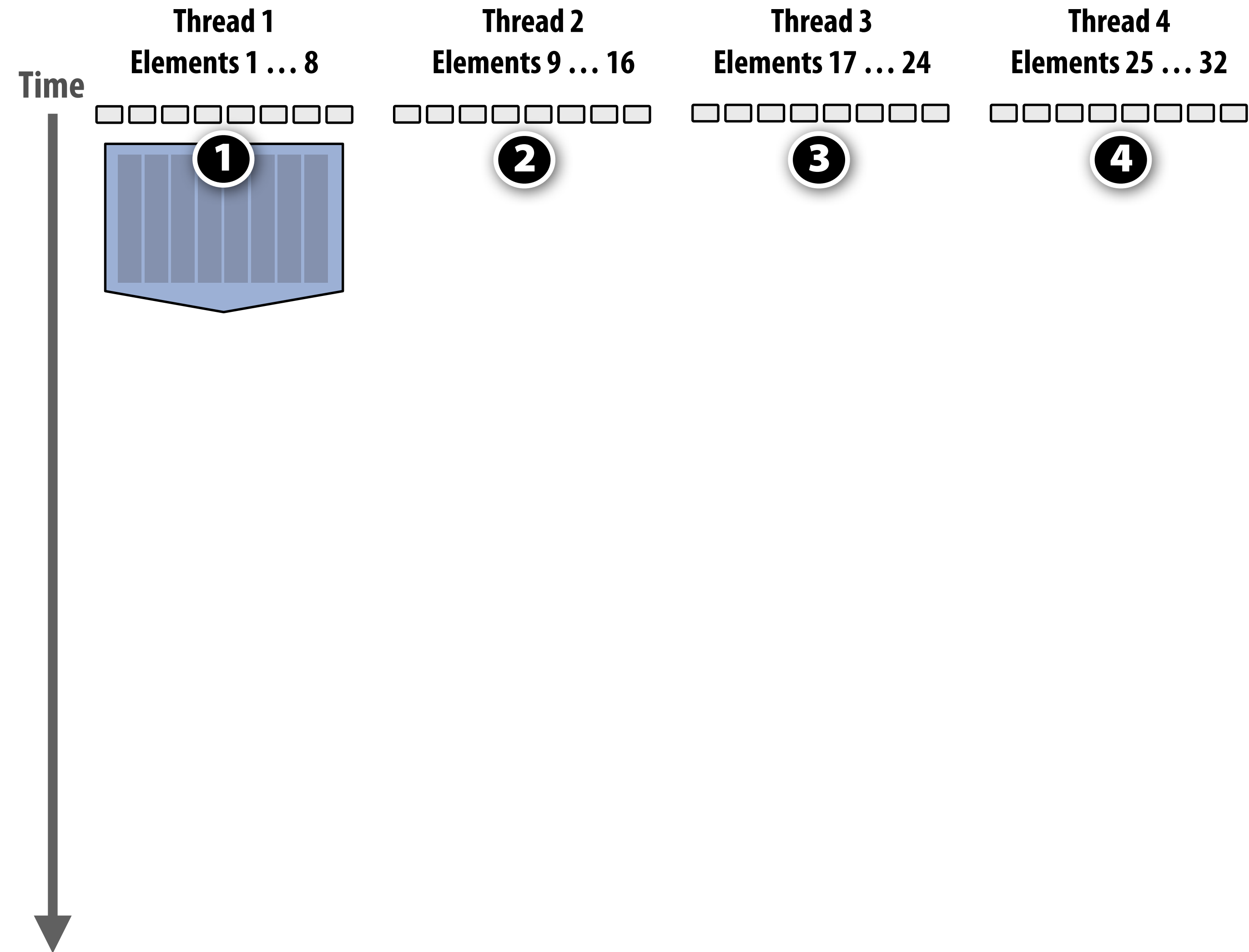
Multi-threading: reduces stalls

- **Idea: Interleave processing of multiple threads on the same core to hide stalls**
- **Like prefetching, a latency hiding technique**

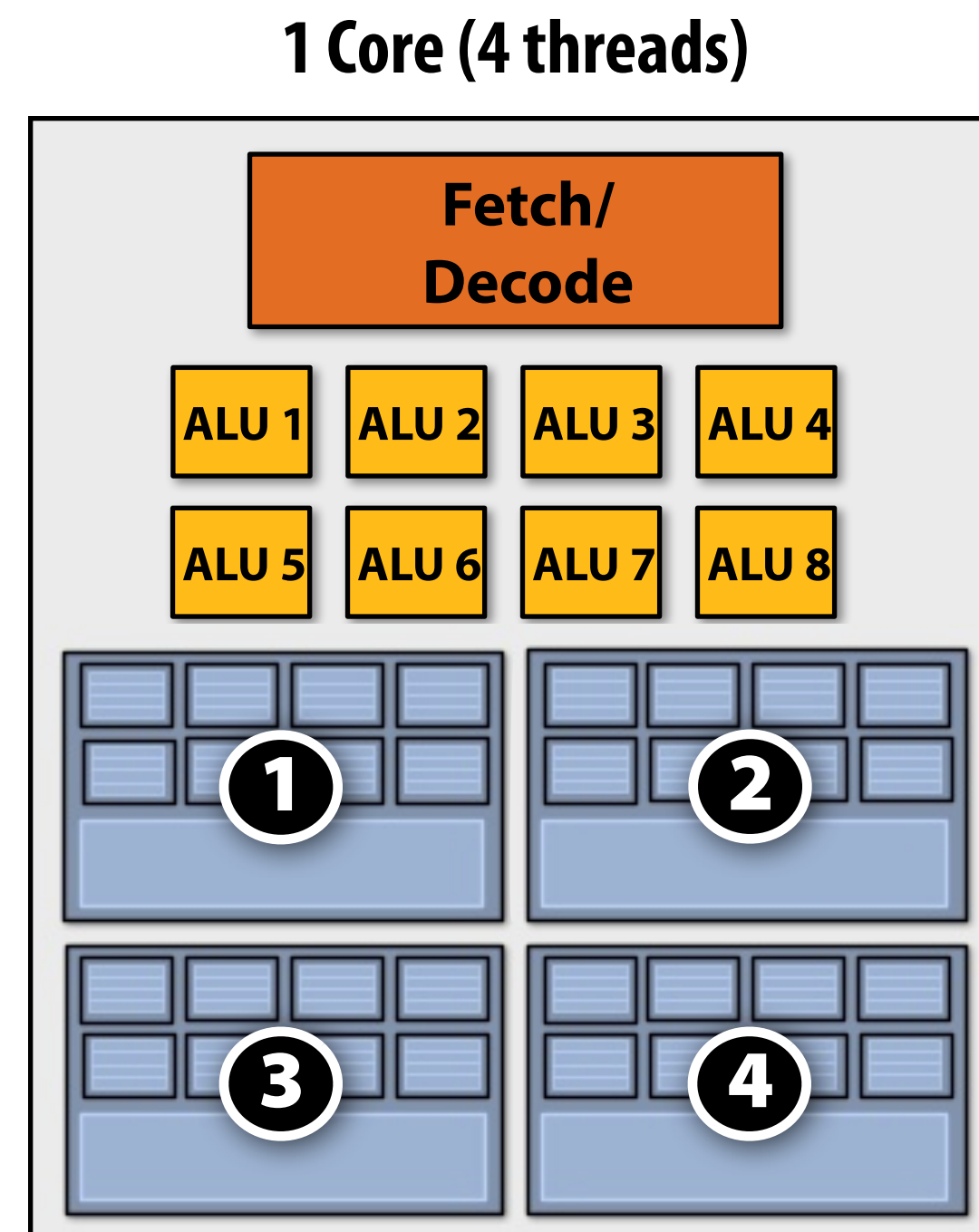
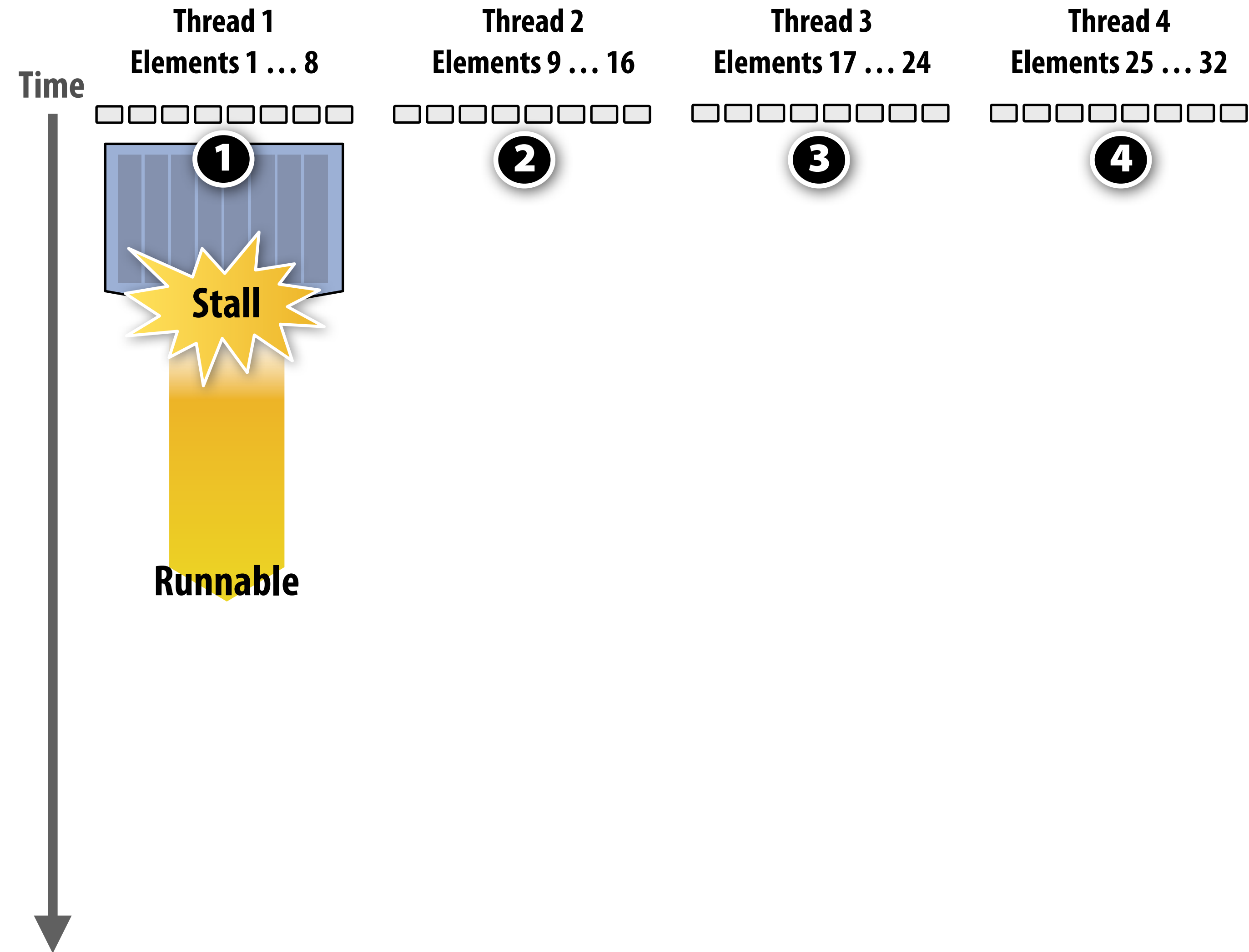
Hiding stalls with multi-threading



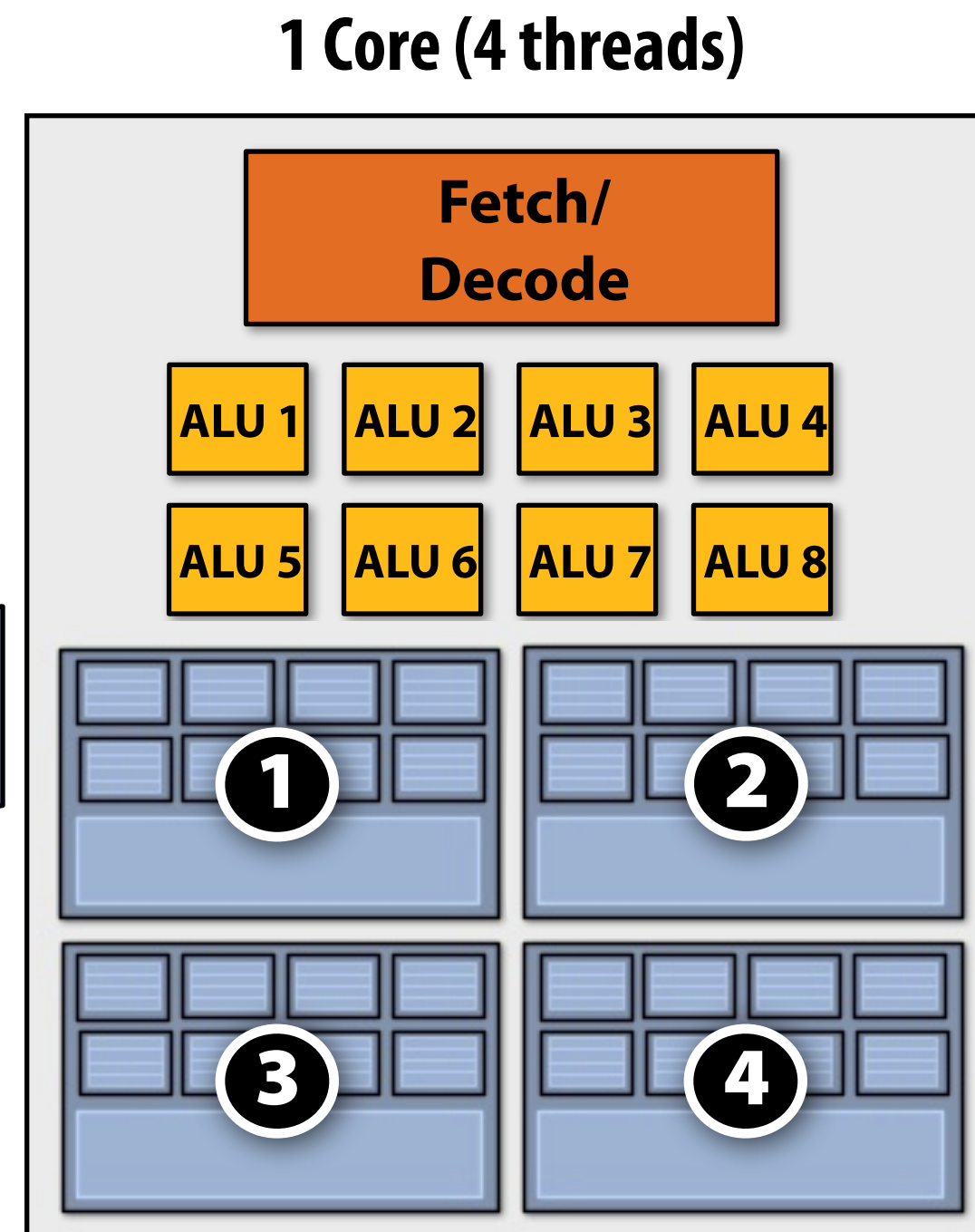
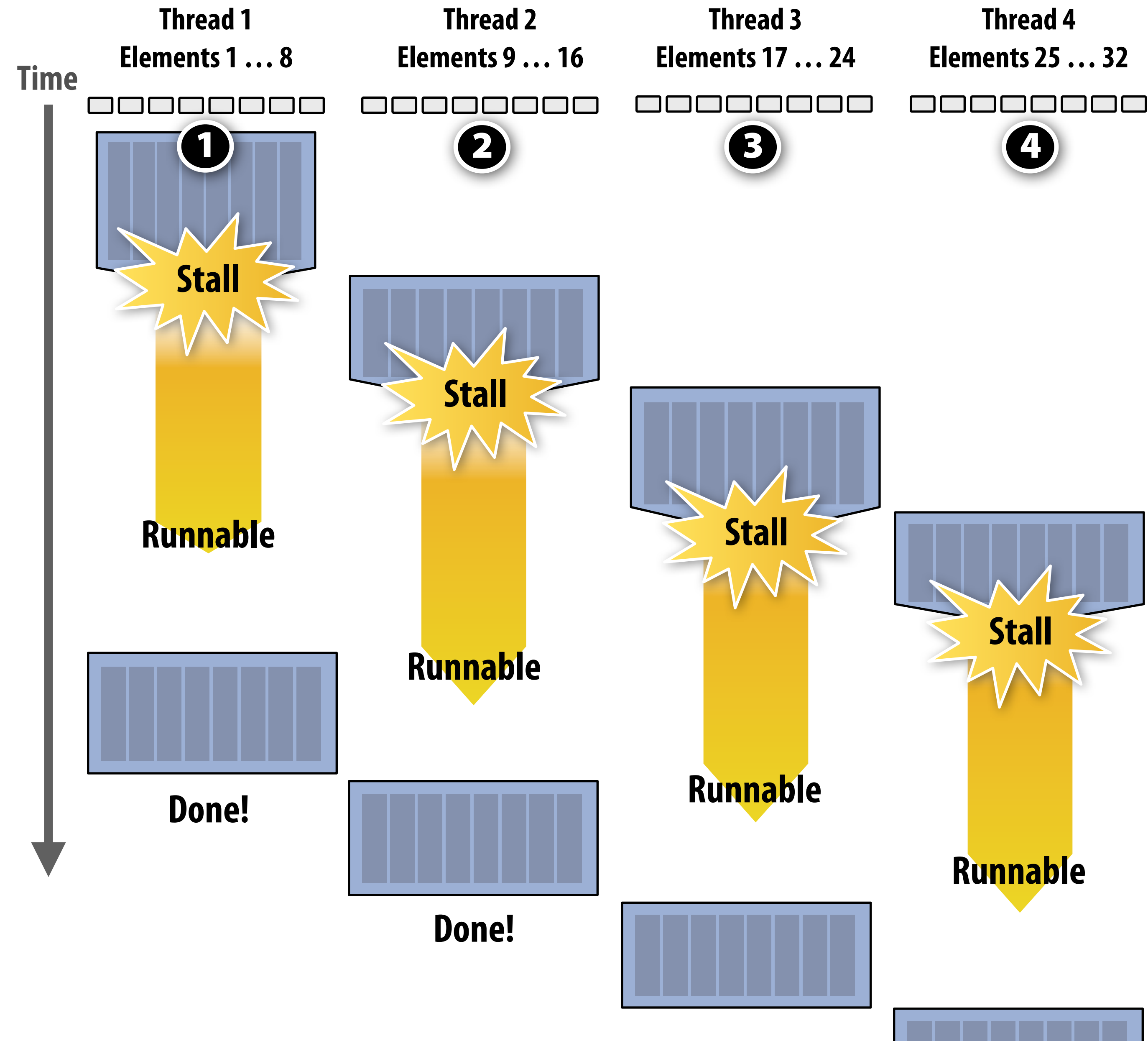
Hiding stalls with multi-threading



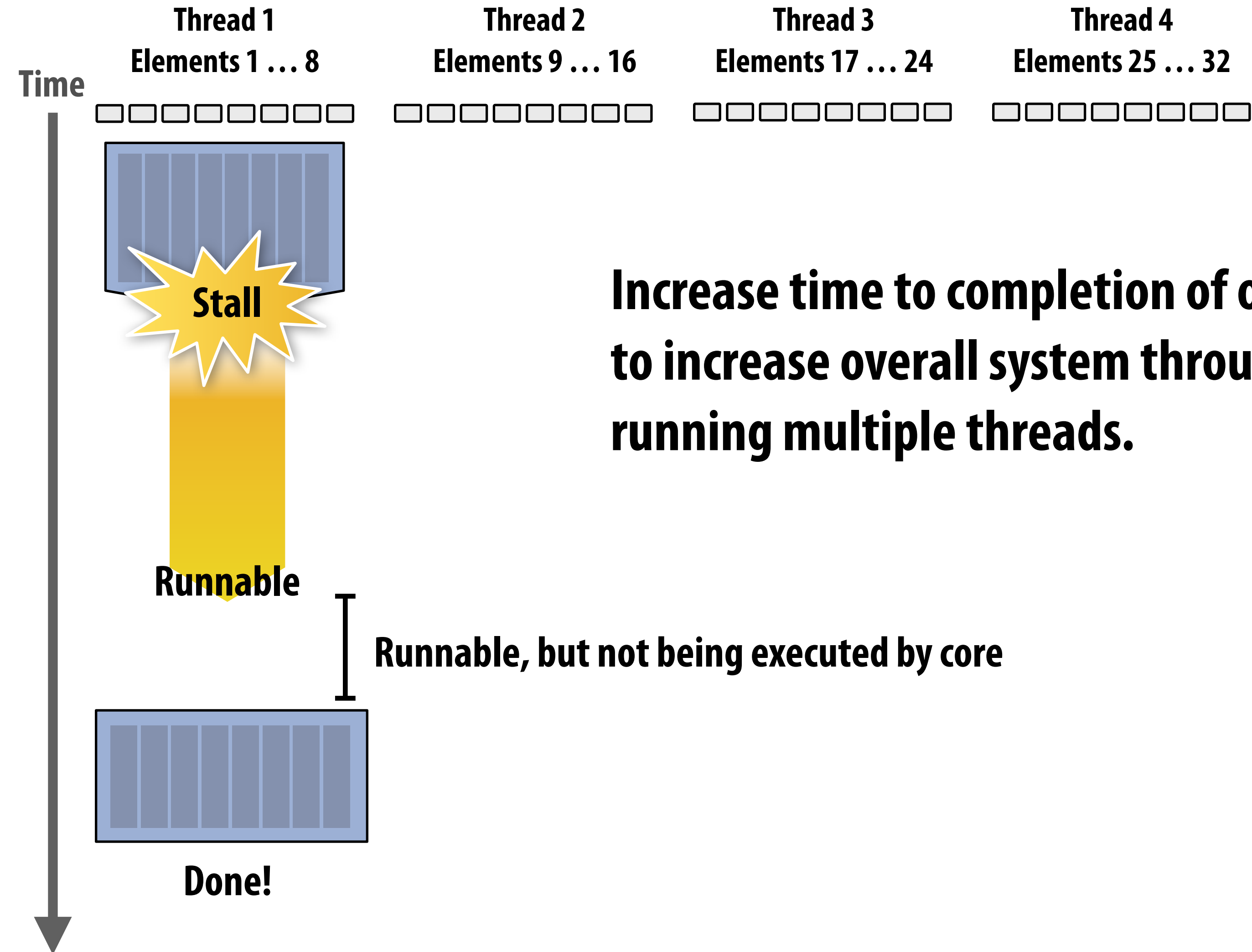
Hiding stalls with multi-threading



Hiding stalls with multi-threading



Throughput computing



Increase time to completion of one thread to increase overall system throughput when running multiple threads.

Storing contexts

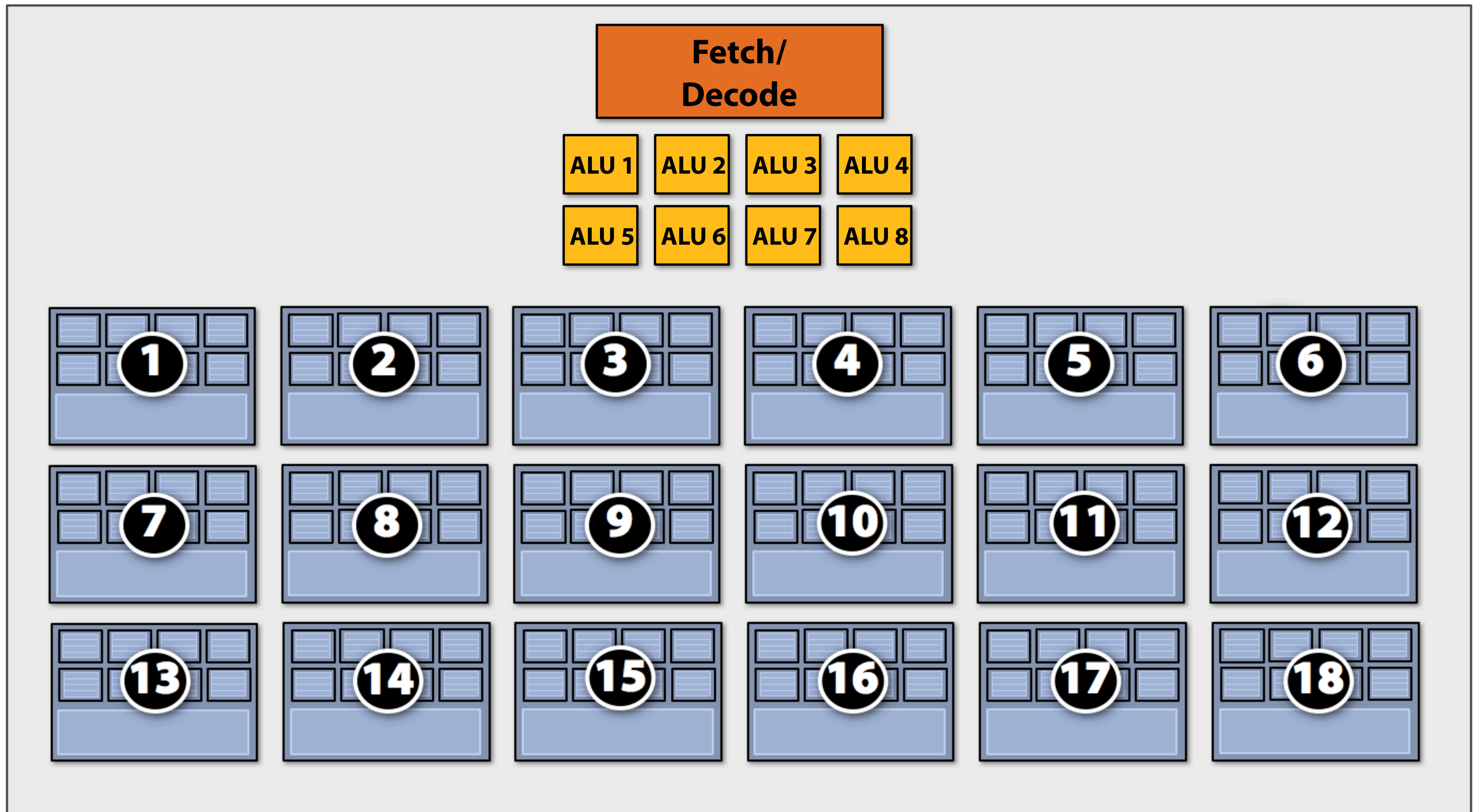
Fetch/
Decode



**Context storage
(or L1 cache)**

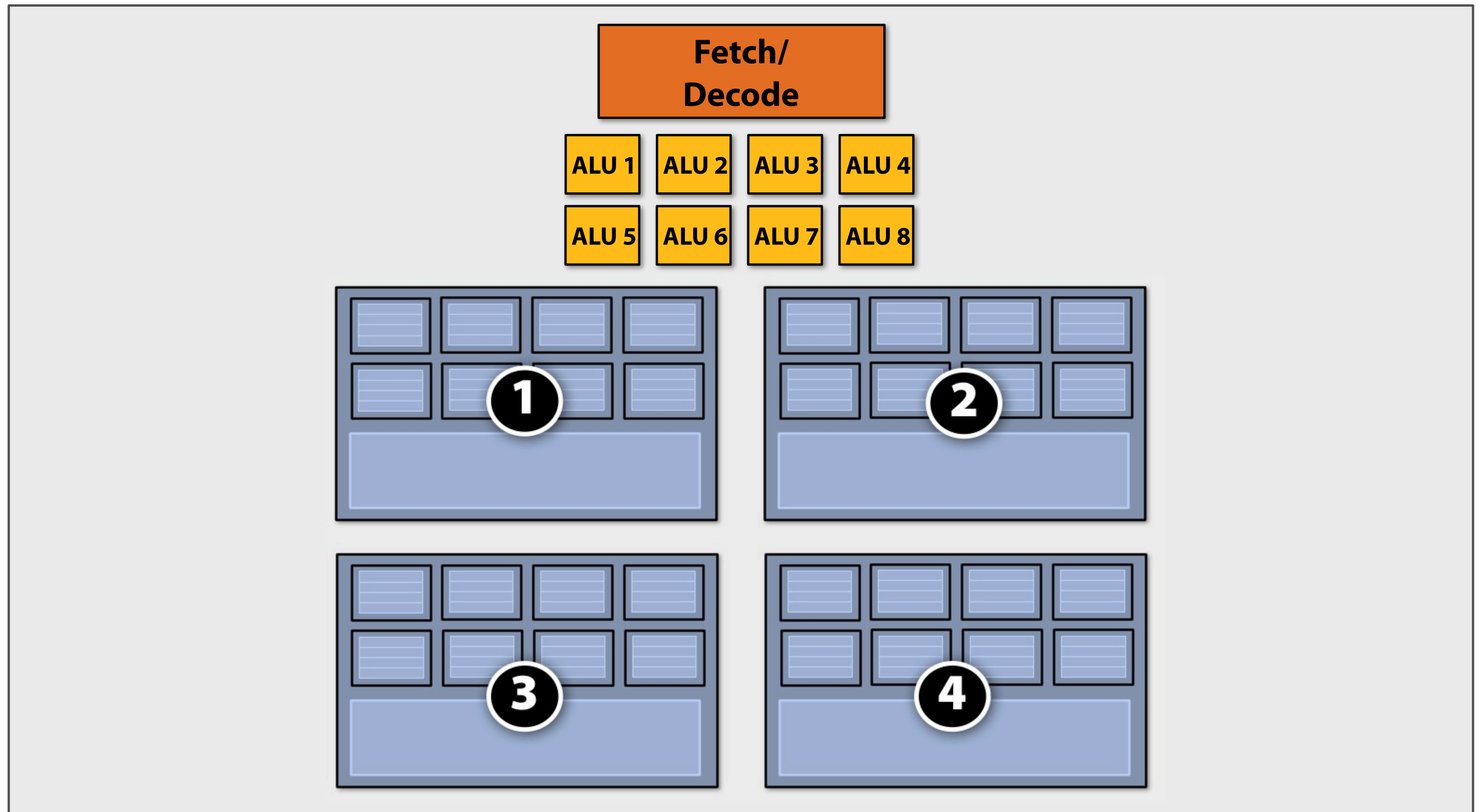
Many small contexts (high latency hiding ability)

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 (chip makes decision each clock, not OS)
 - Remember: core has the same number of ALU resources
- **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
 - This is GPU-style multi-threading (GPUs: many threads per core)
- **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**

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

■ **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 parallelism in a program (more parallelism than ALUs)**
- **Relies heavily on memory bandwidth**
 - **More threads → larger working set → less cache space per thread**
 - **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**

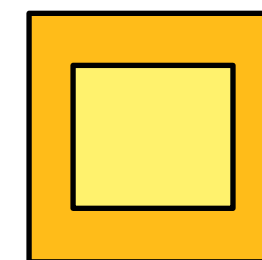
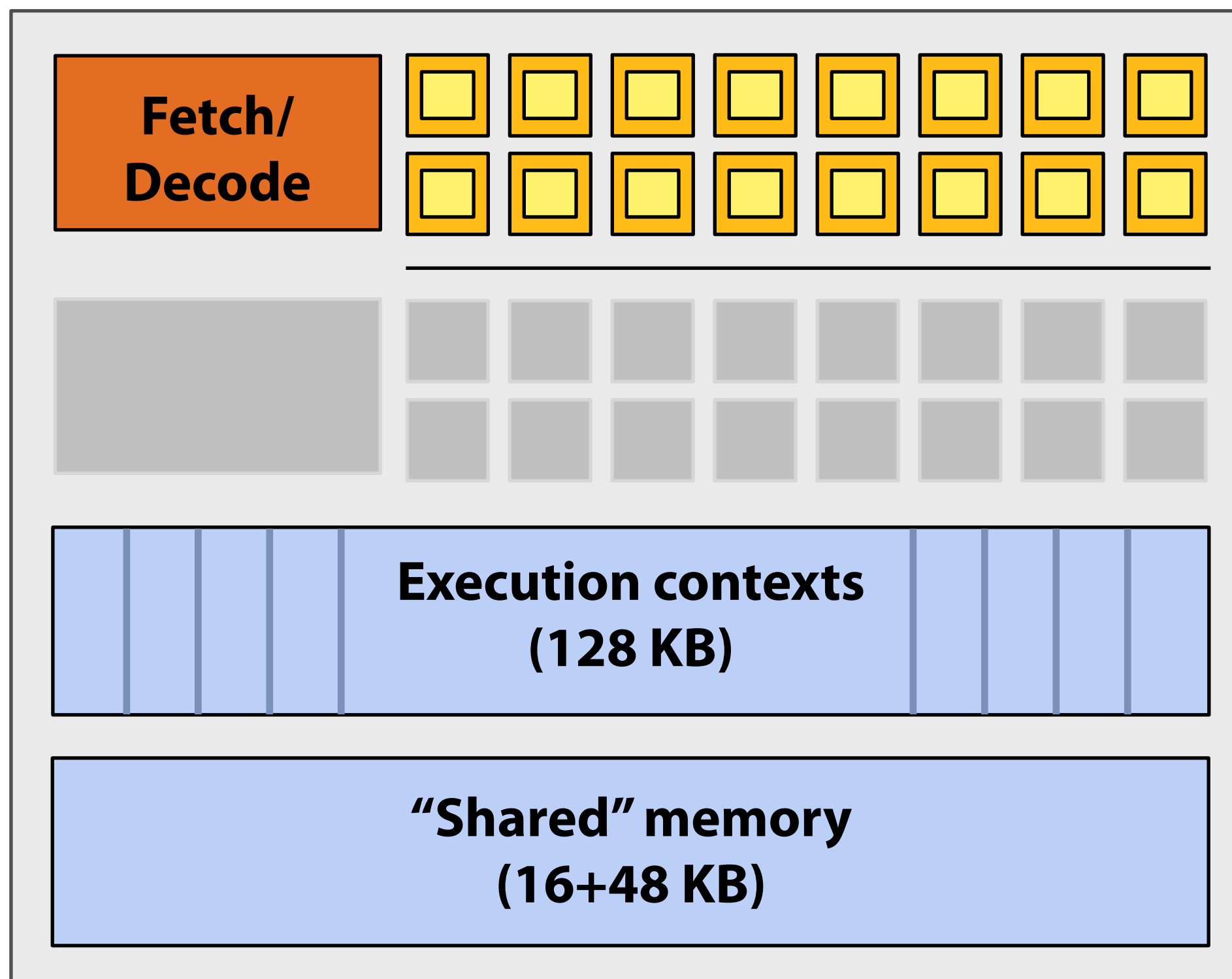
**64 total concurrent
instruction streams**

**512 independent pieces of
work to run chip at maximal
latency hiding ability**



GPUs: Extreme throughput-oriented processors

NVIDIA GTX 480 core



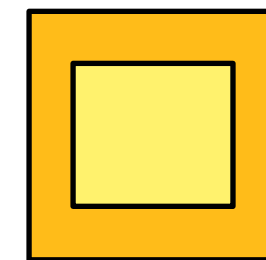
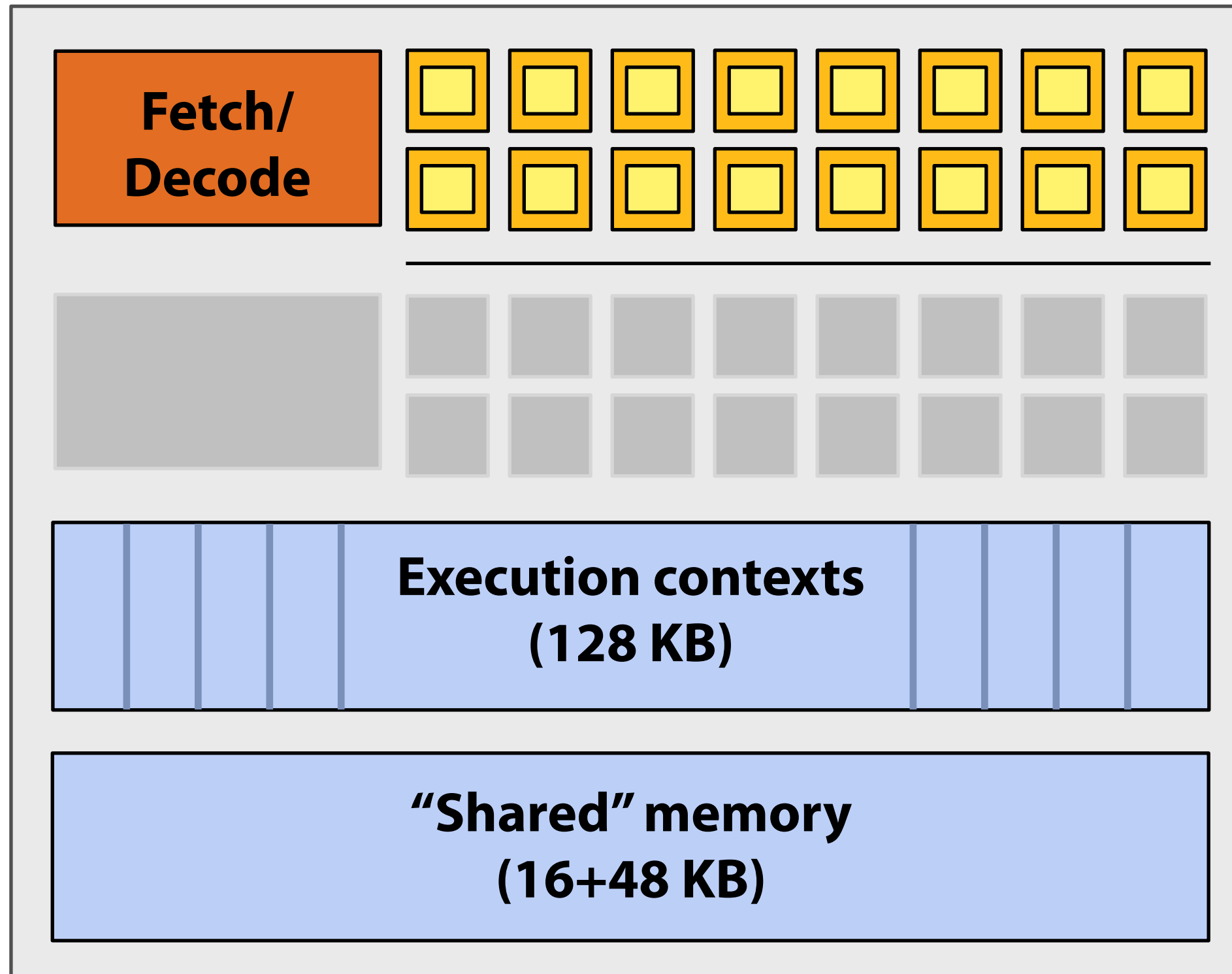
= SIMD function unit,
control shared across 16 units
(1 MUL-ADD per clock)

- 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



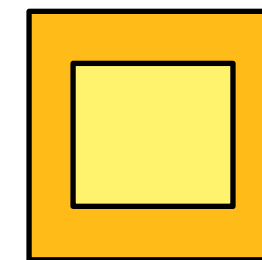
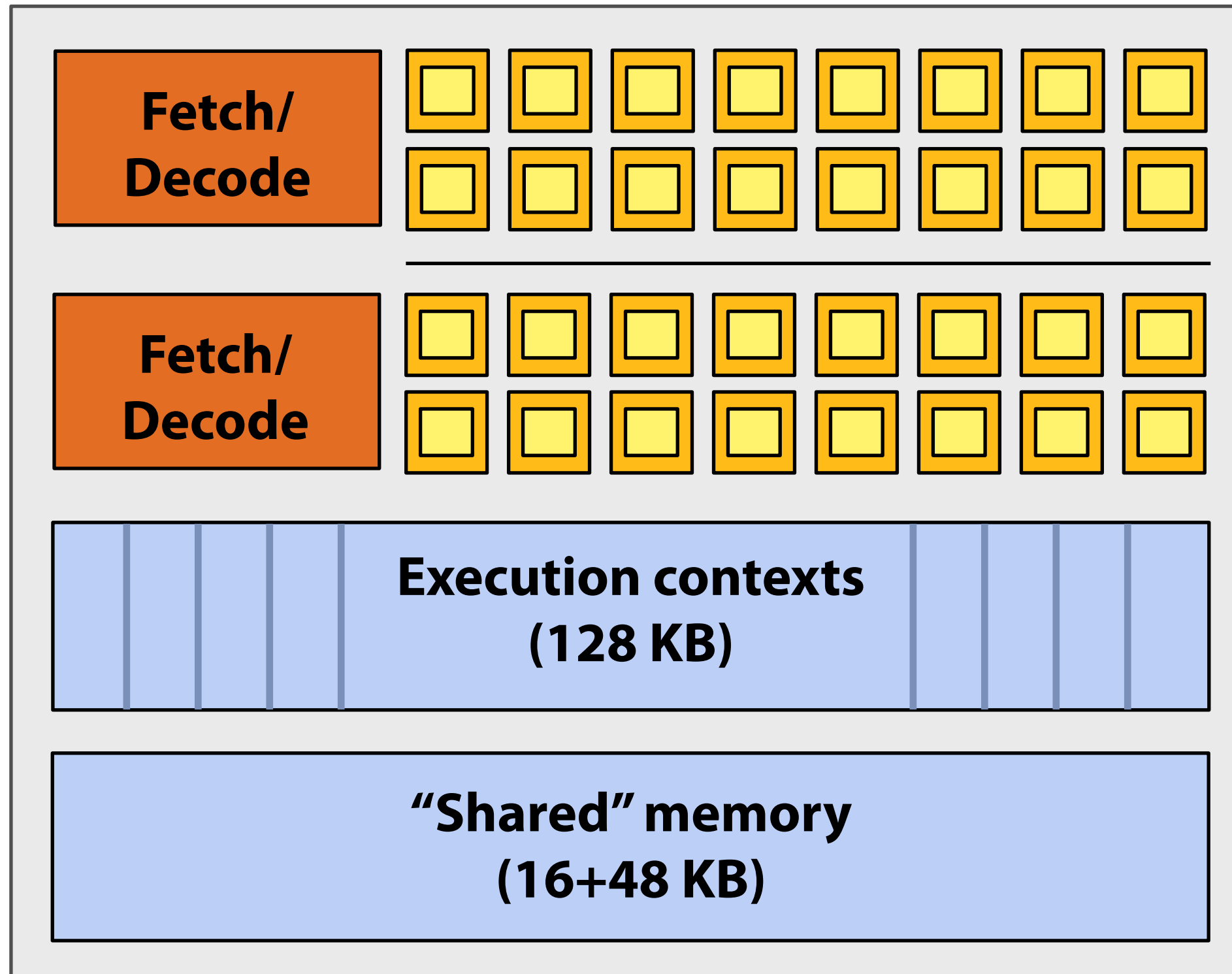
= SIMD function unit,
control shared across 16 units
(1 MUL-ADD per clock)

- 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

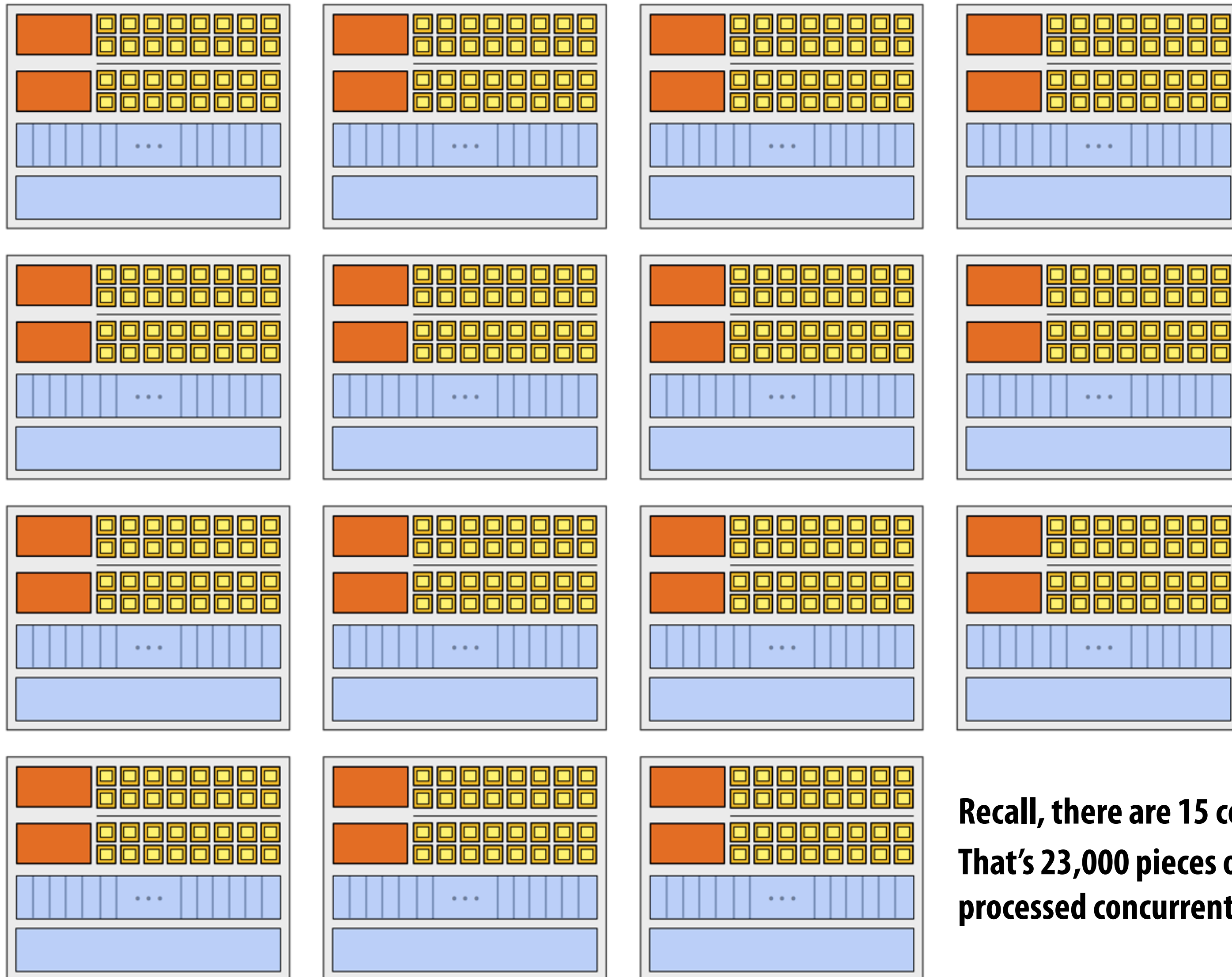


= SIMD function unit,
control shared across 16 units
(1 MUL-ADD per clock)

- This process occurs on another set of 16 ALUs as well
- So there's 32 ALUs per core
- $15 * 32 = 480$ ALUs per chip

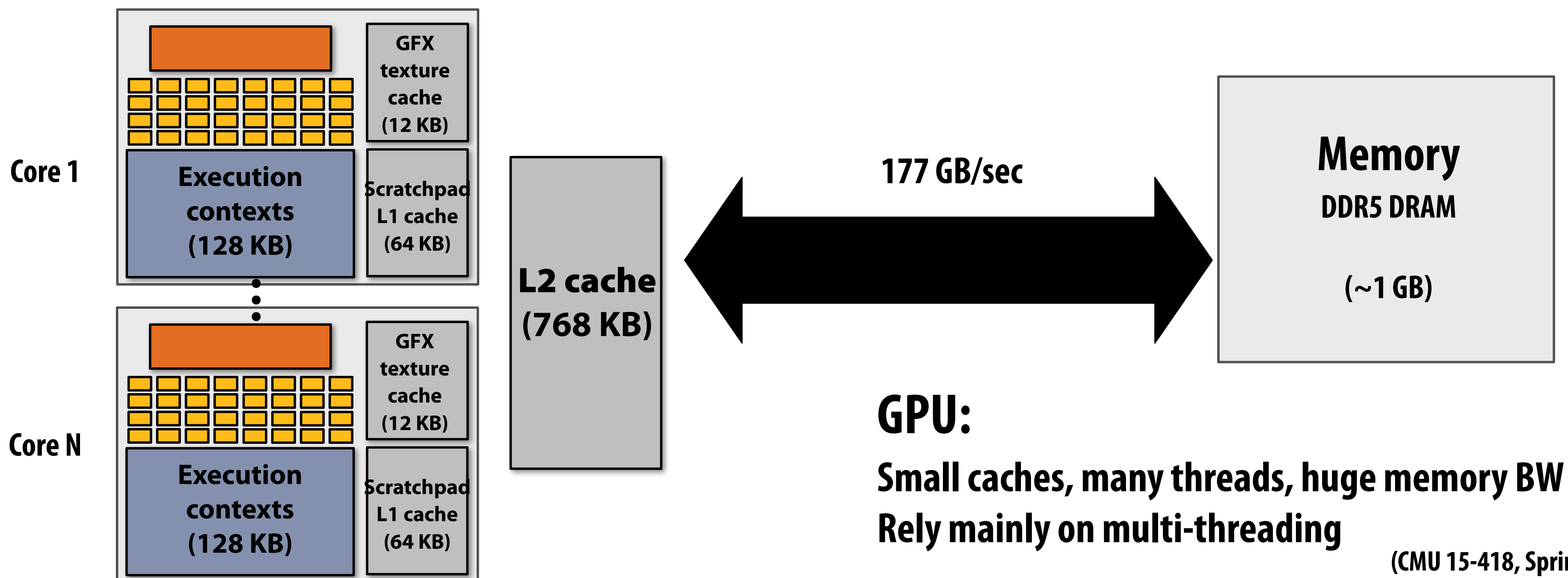
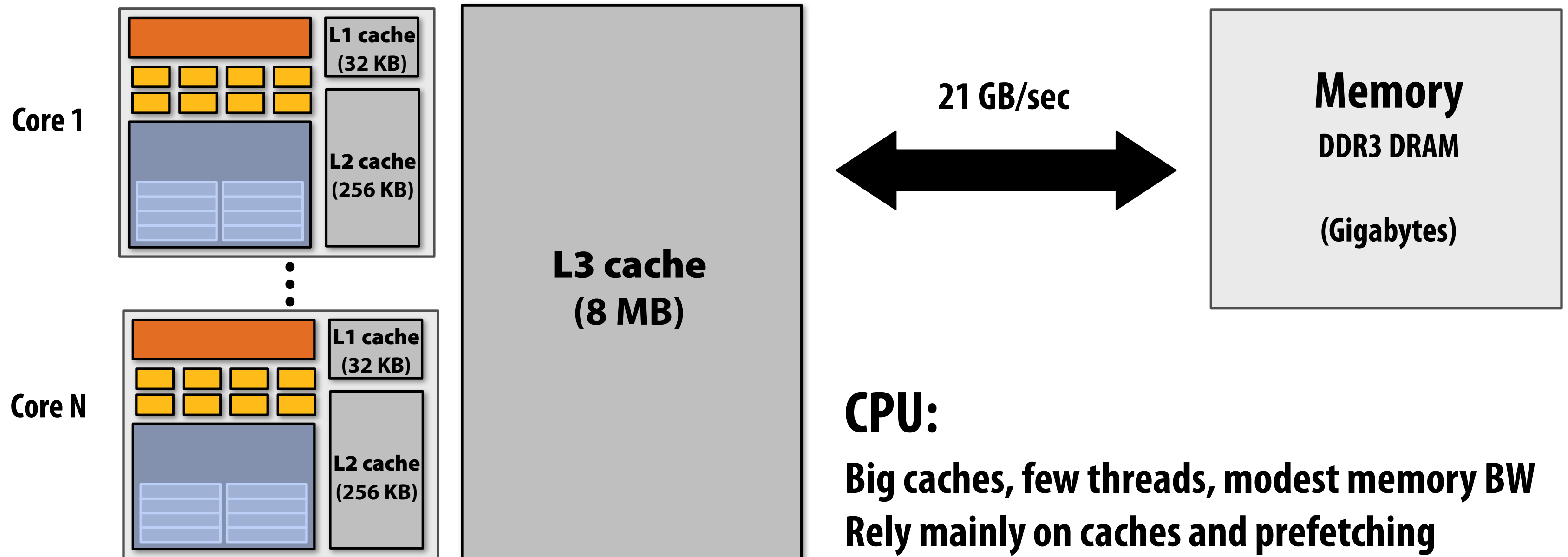
Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G

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

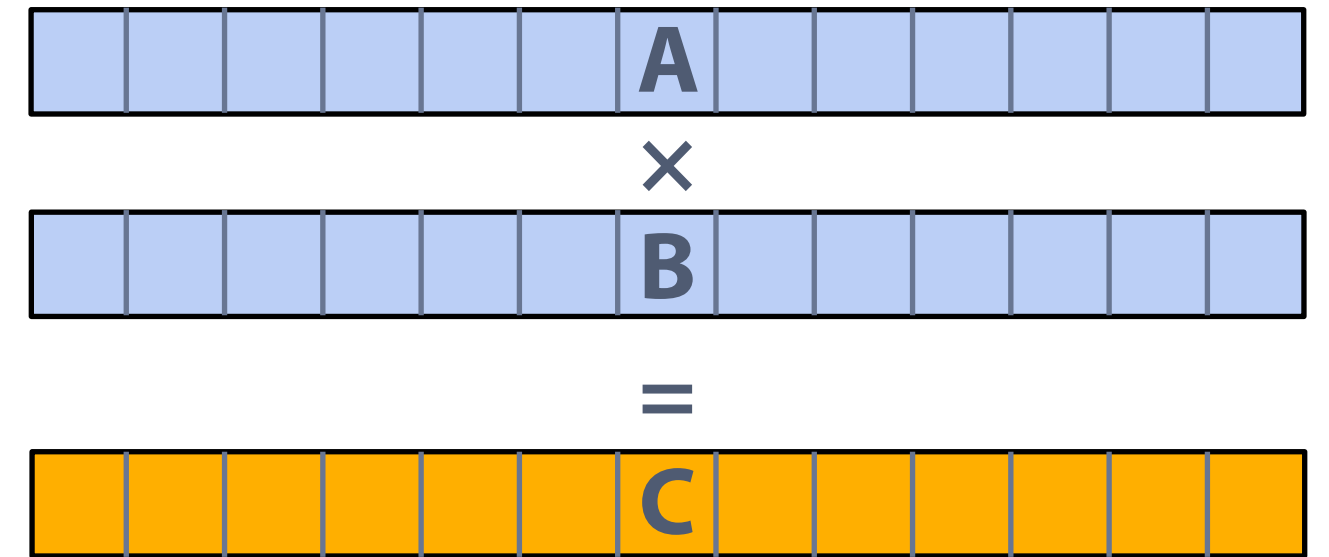


Thought experiment

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

Assume vectors contain millions of elements

- 1. Load input A[i]**
- 2. Load input B[i]**
- 3. Compute $A[i] \times B[i]$**
- 4. Store result into C[i]**



Three memory operations (12 bytes) for every MUL

NVIDIA GTX 480 GPU can do 480 MULs per clock (1.4 GHz)

Need ~8.4 TB/sec of bandwidth to keep functional units busy

~ 2% efficiency... but 8x faster than CPU!

(3GHz Core i7 quad-core CPU: 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:

- Fetch data from memory less often
 - Reuse data in a thread (traditional locality optimizations)
 - Share data across threads (cooperation)
- Request data less often (instead, do more math: it is “free”)
 - Term: “arithmetic intensity” - ratio of math to data access

Summary

- **Three major ideas that all modern processors employ to varying degrees**
 - **Employ multiple processing cores**
 - **Simpler cores (embrace thread-level parallelism over ILP)**
 - **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**

Terminology

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