Lecture 2:

# A Modern Multi-Core Processor

Parallel Computing
Stanford CS149, Fall 2024

# Review from class 1:
# What is a computer program?

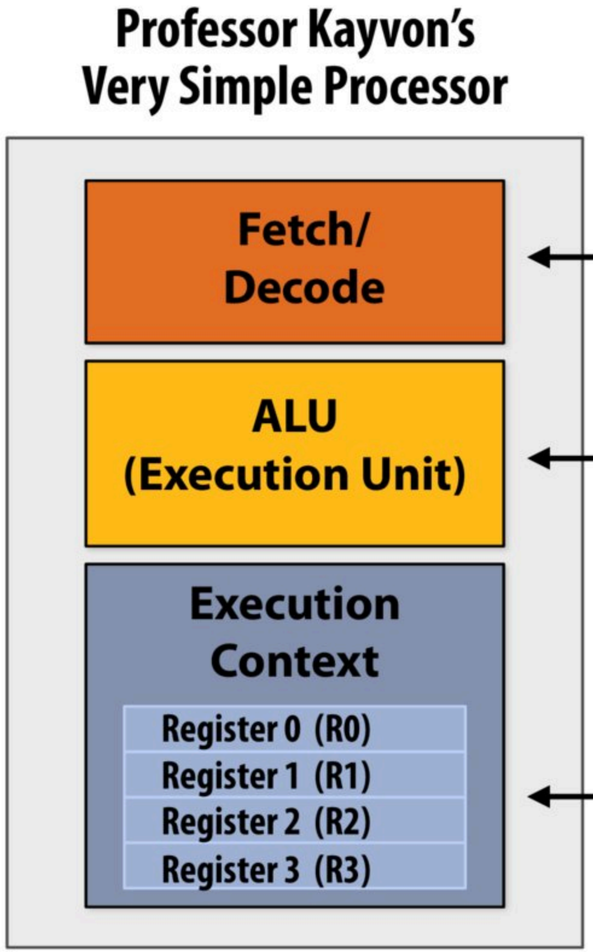
# A program is a list of processor instructions!

```
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;
}
```


```
_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



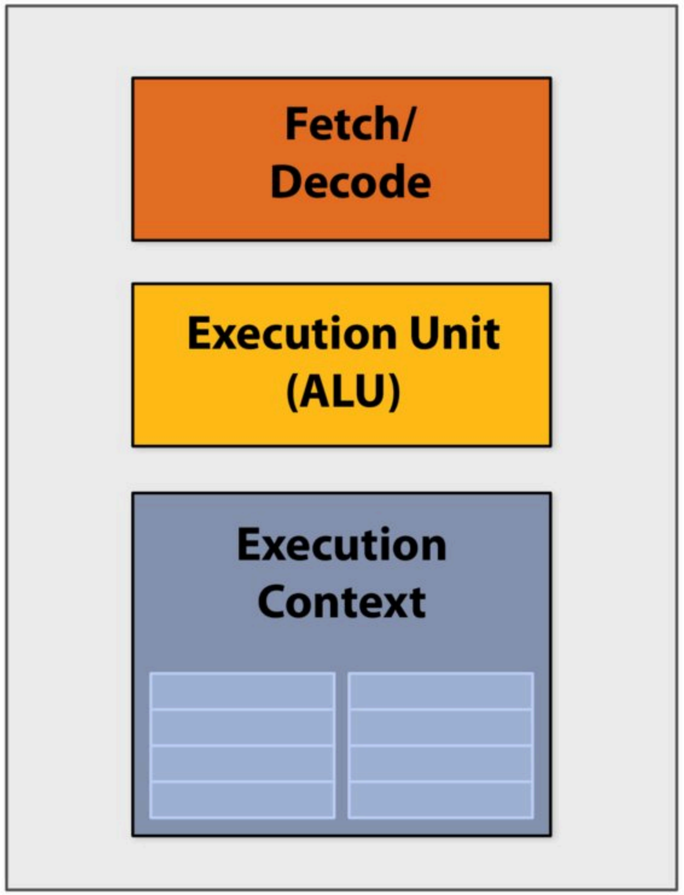

Determine what instruction to run next

Execution unit: performs the operation described by an instruction, which may modify values in the processor's registers or the computer's memory

Registers: maintain program state: store value of variables used as inputs and outputs to operations

# Execute program

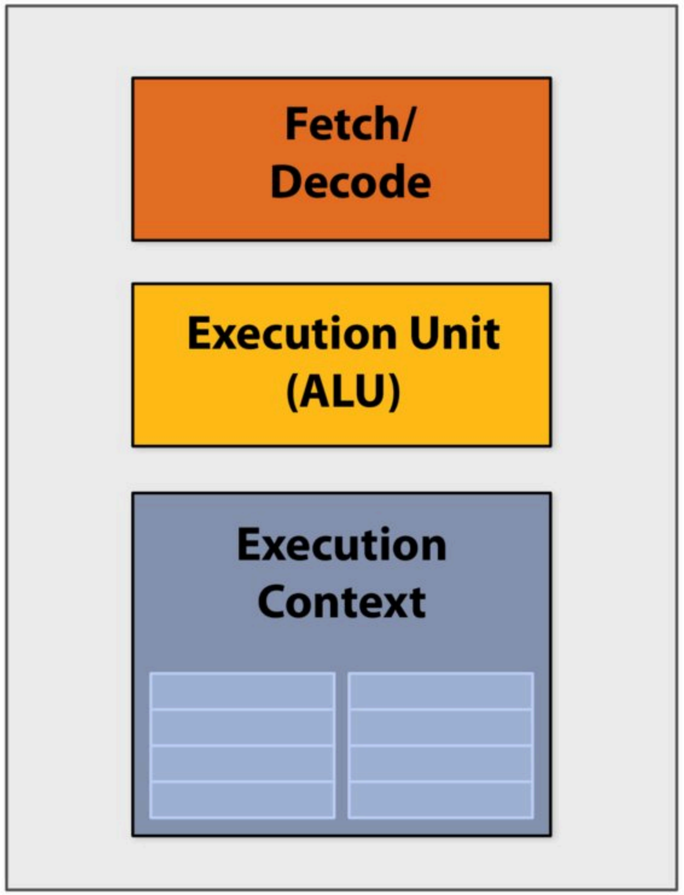
## My very simple processor: executes one instruction per clock




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

# Execute program

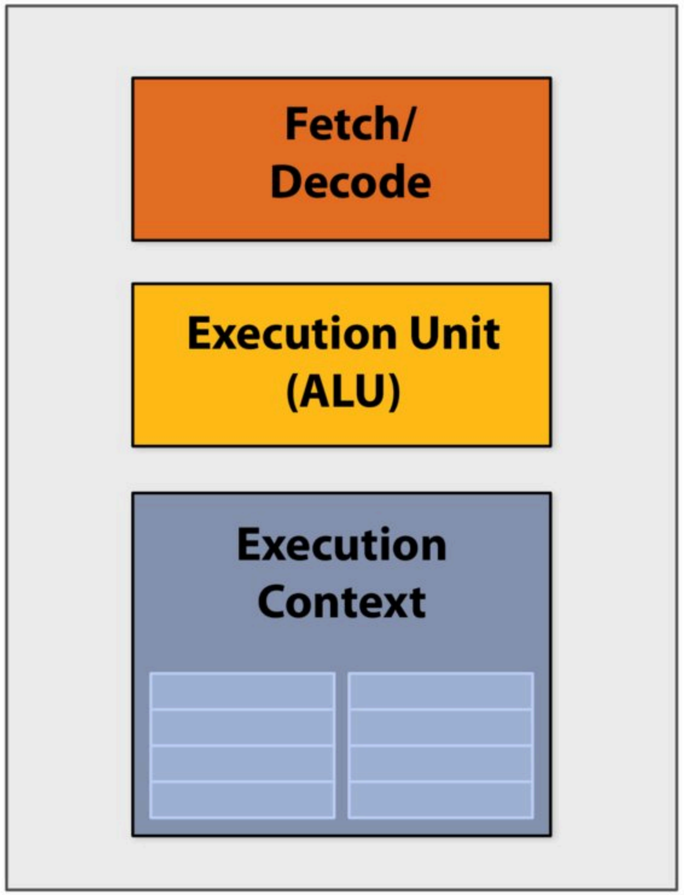
## My very simple processor: executes one instruction per clock




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

# Execute program

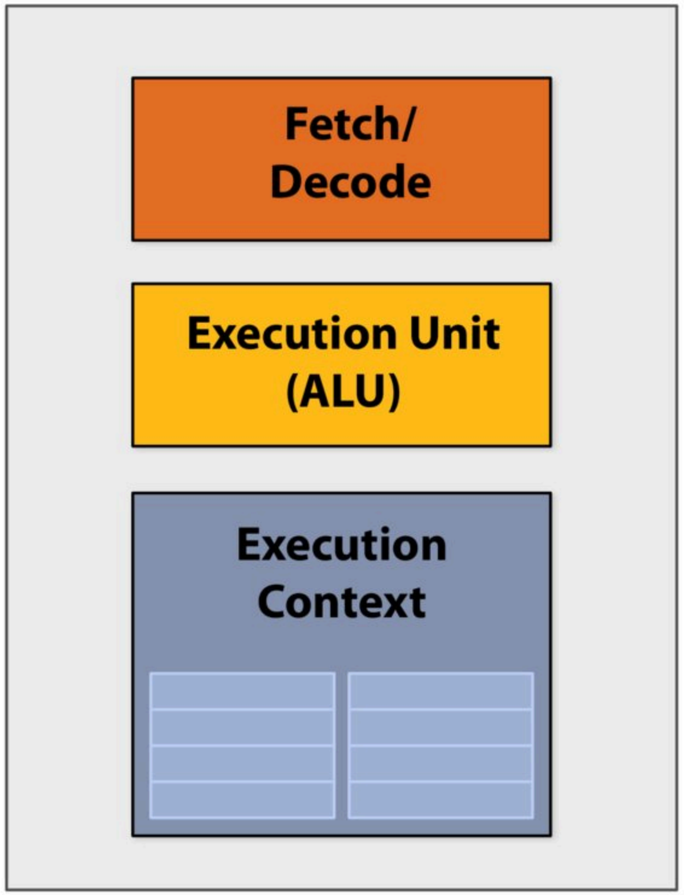
## My very simple processor: executes one instruction per clock




```
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



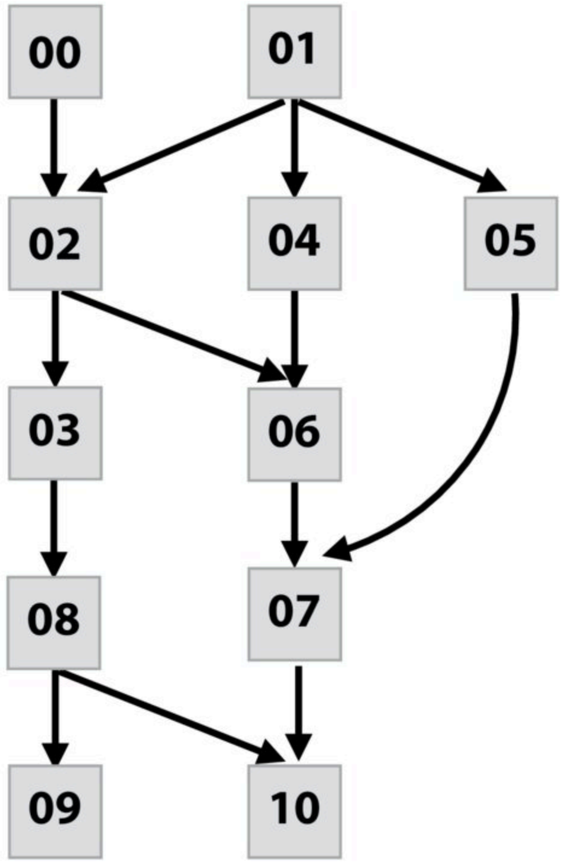

```
ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...
...
...
...
...
...
st   addr[r2], r0
```

# A program with instruction level parallelism

**Program (sequence of instructions)**

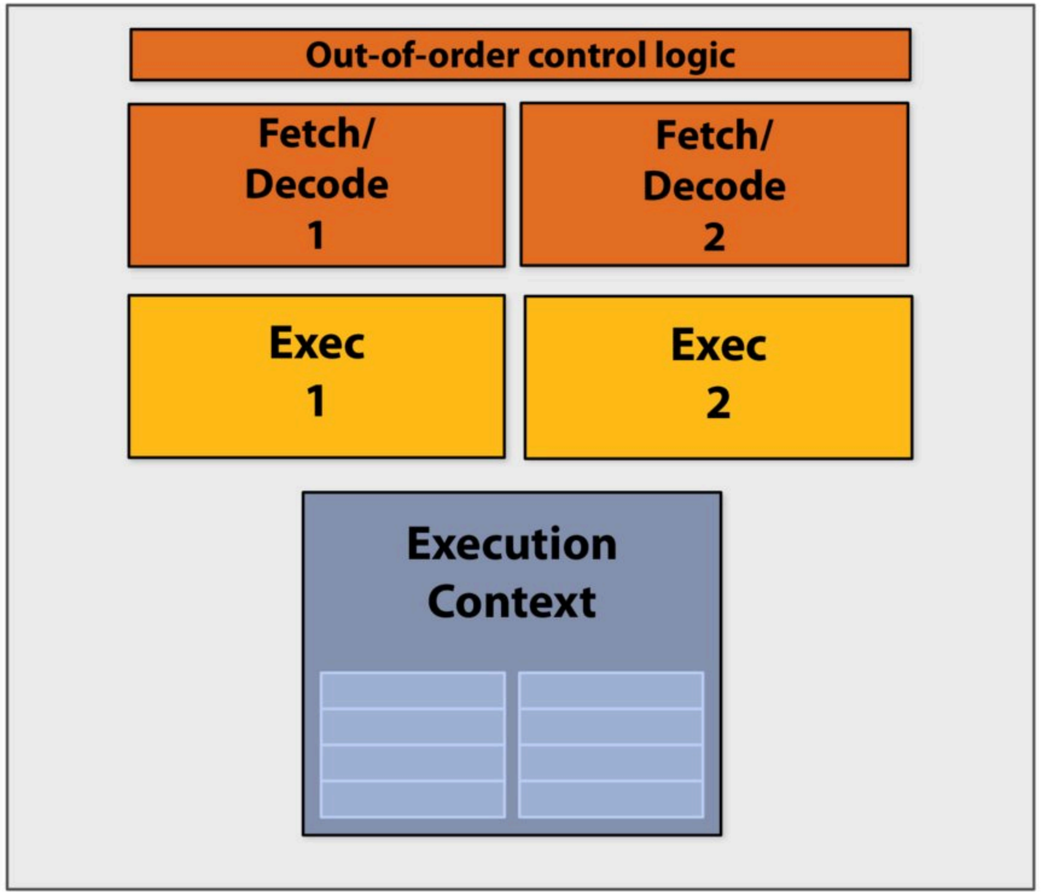
| PC | Instruction | |
|---|---|---|
| 00 | a = 2 | |
| 01 | b = 4 | |
| | | *value during execution* |
| 02 | tmp2 = a + b | // 6 |
| 03 | tmp3 = tmp2 + a | // 8 |
| 04 | tmp4 = b + b | // 8 |
| 05 | tmp5 = b * b | // 16 |
| 06 | tmp6 = tmp2 + tmp4 | // 14 |
| 07 | tmp7 = tmp5 + tmp6 | // 30 |
| | |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

**Instruction dependency graph**



# Superscalar processor

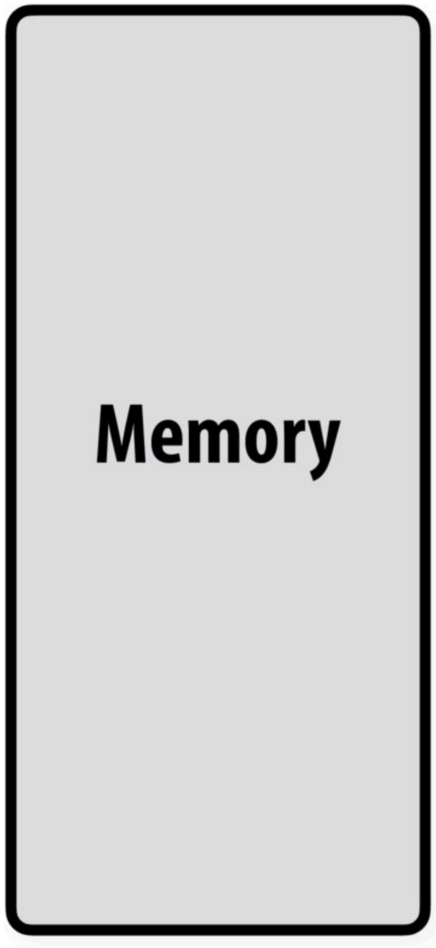
## This processor can decode and execute up to two instructions per clock




**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?

# 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

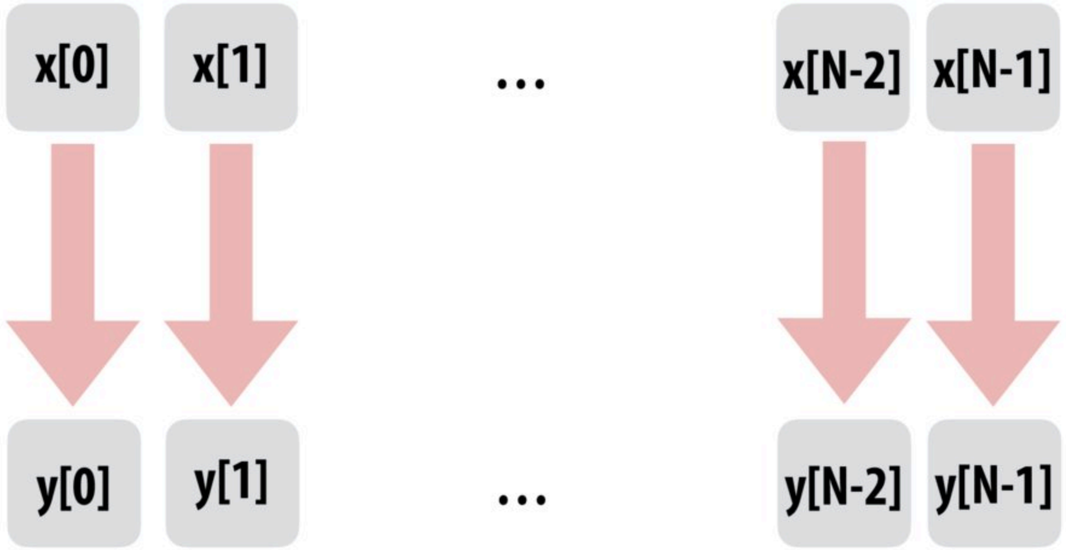
# Today's example program

```
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**

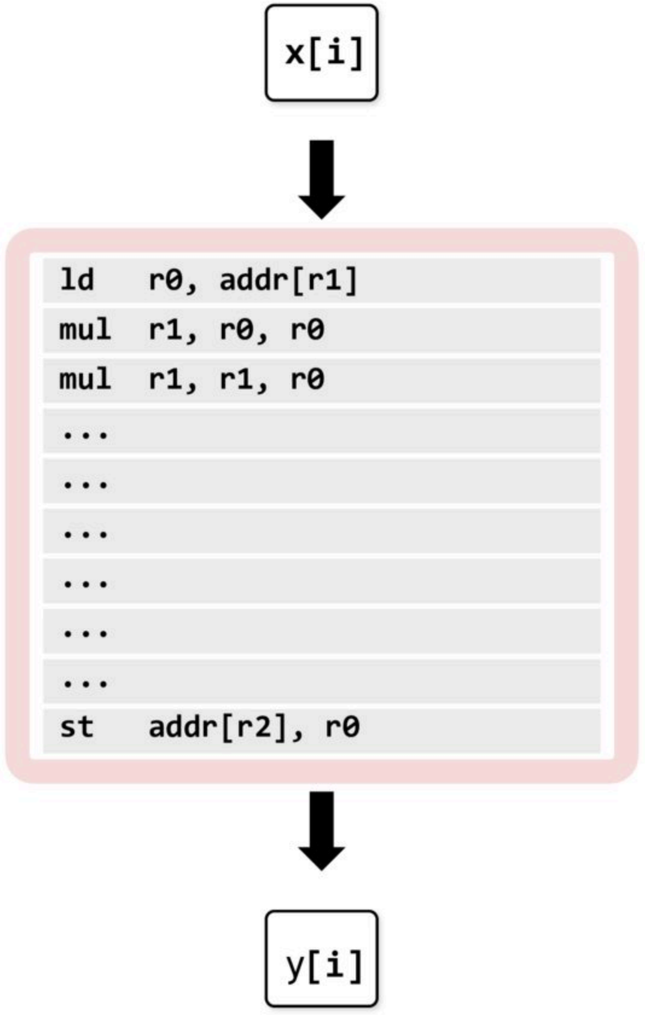


# Compile program

```
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;
   }
}
```

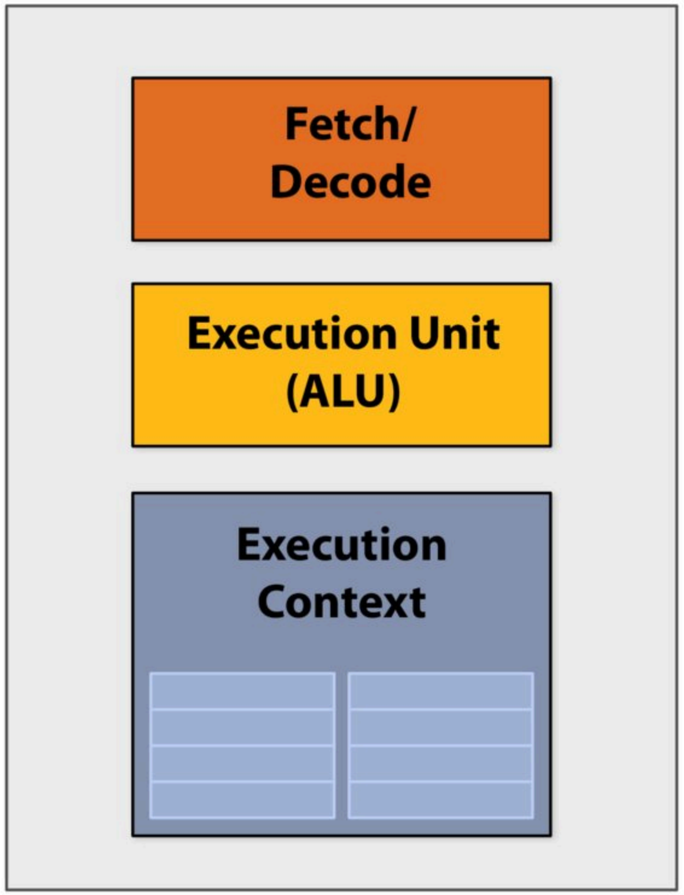
compiler

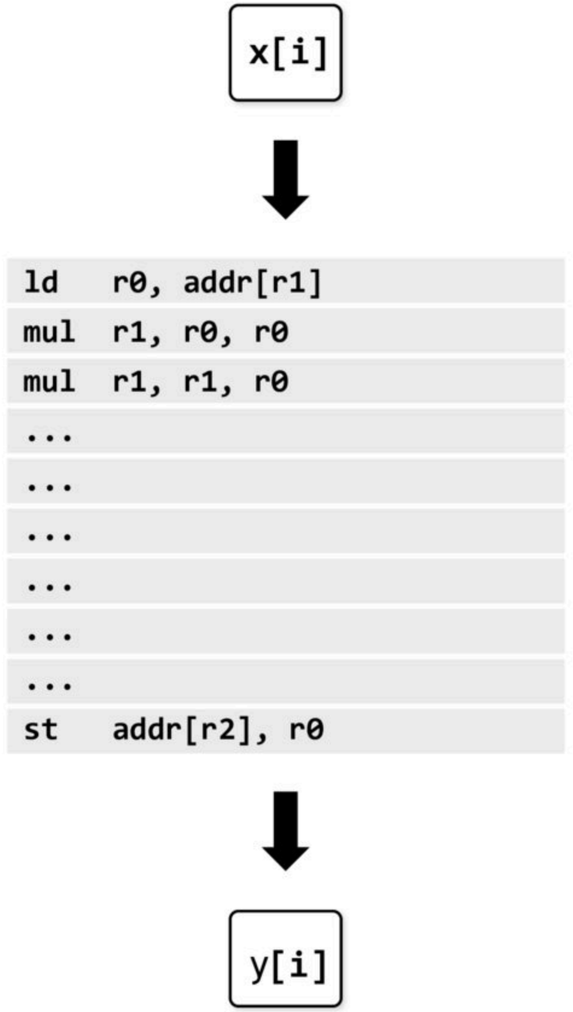
**Compiled instruction stream (scalar instructions)**



# Execute program

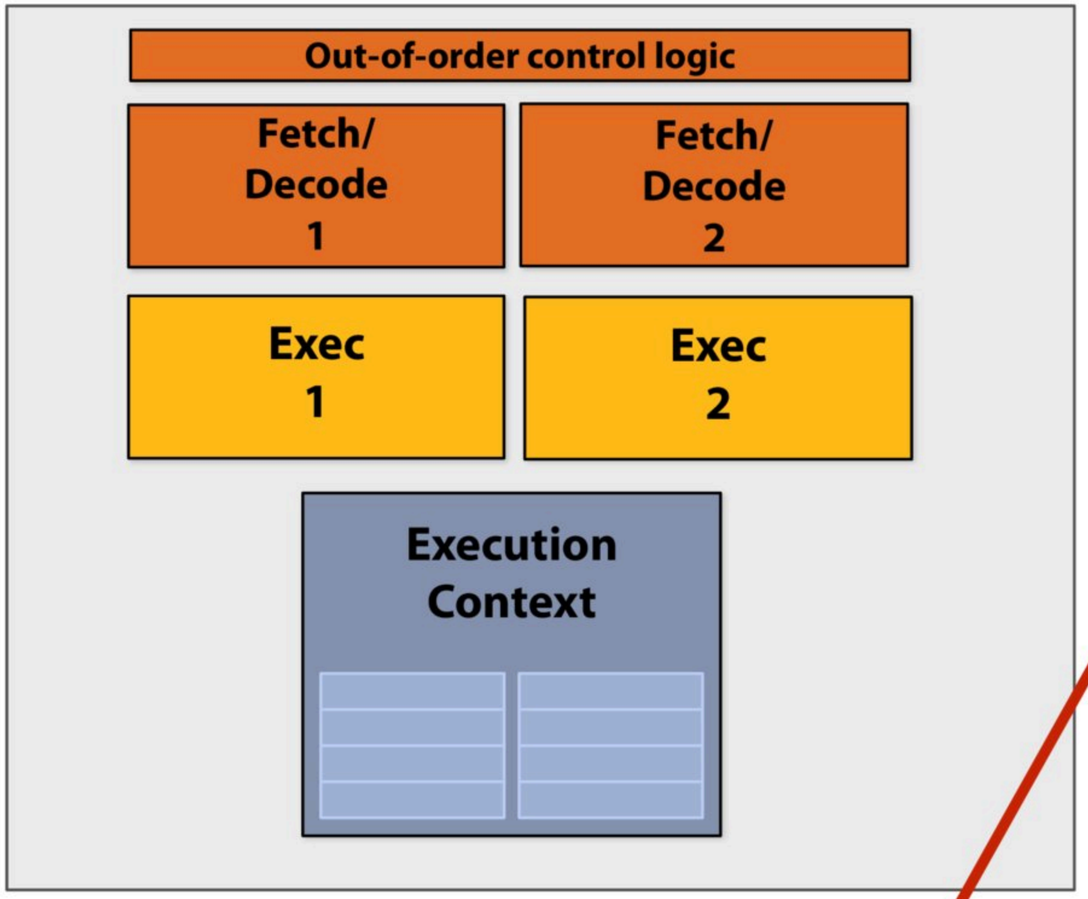
## My very simple processor: executes one instruction per clock

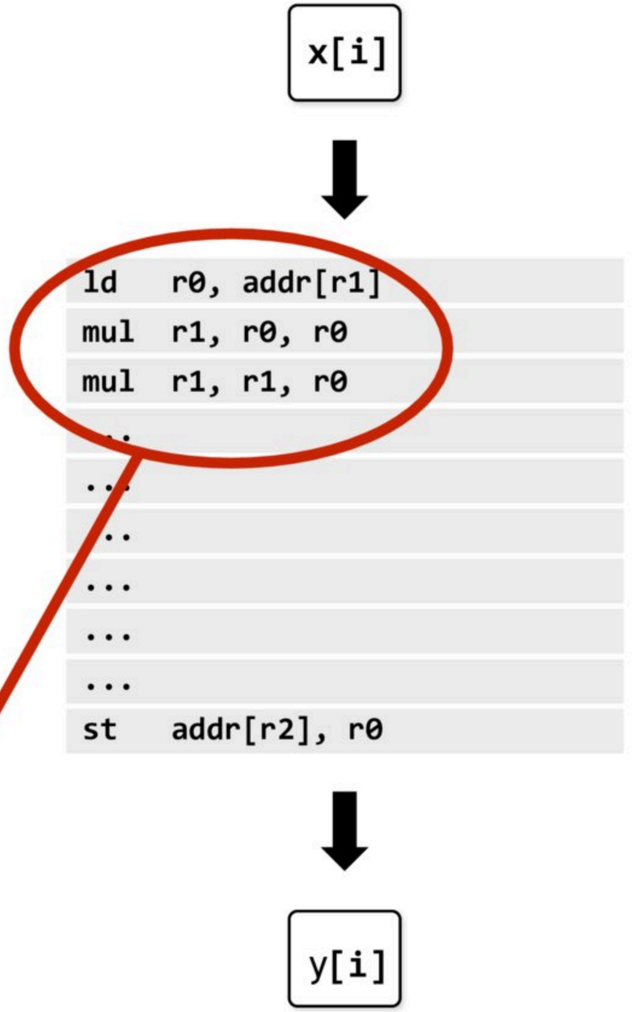


# Superscalar processor

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

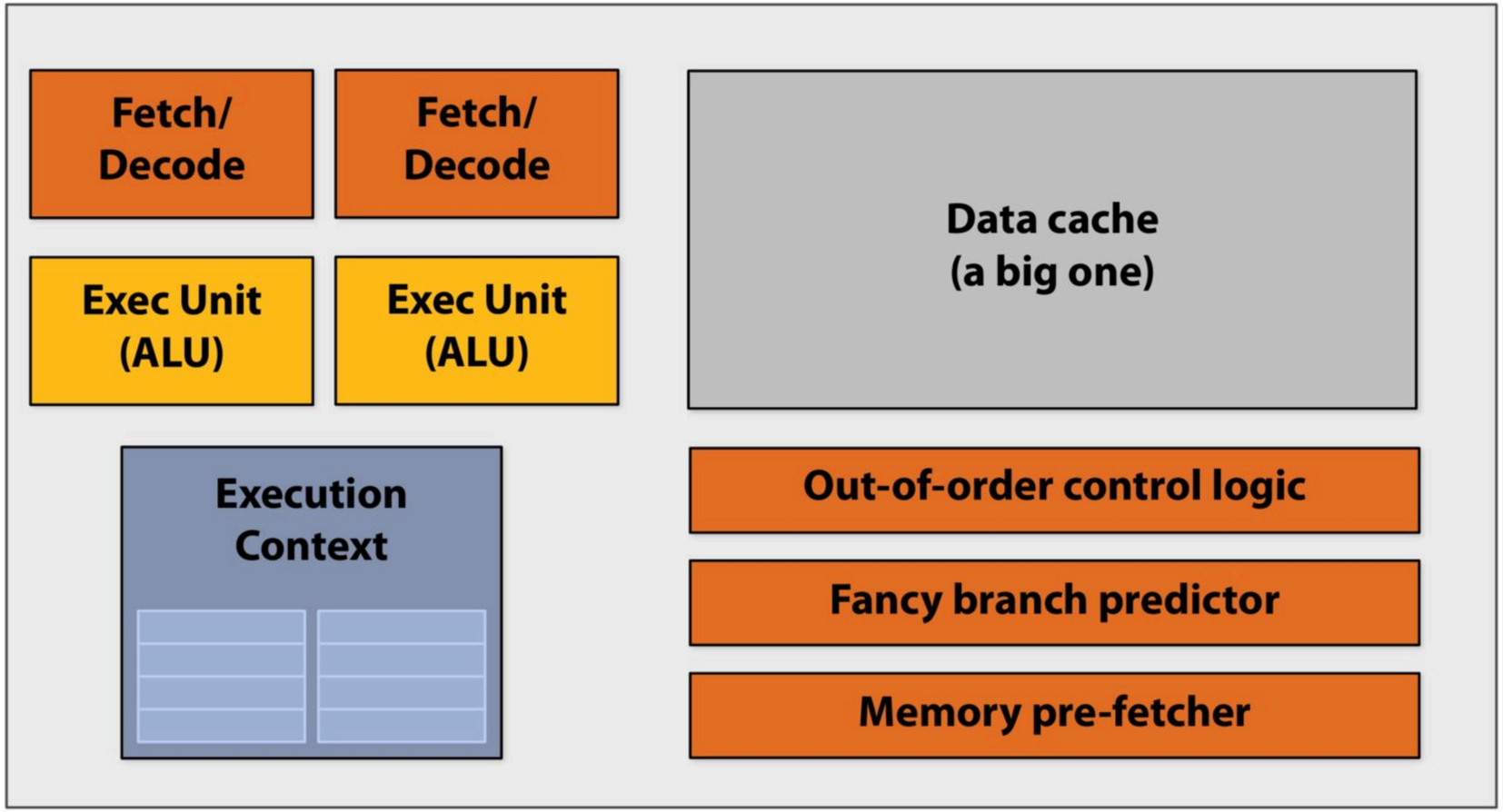



**Note: No ILP exists in this region of the program**

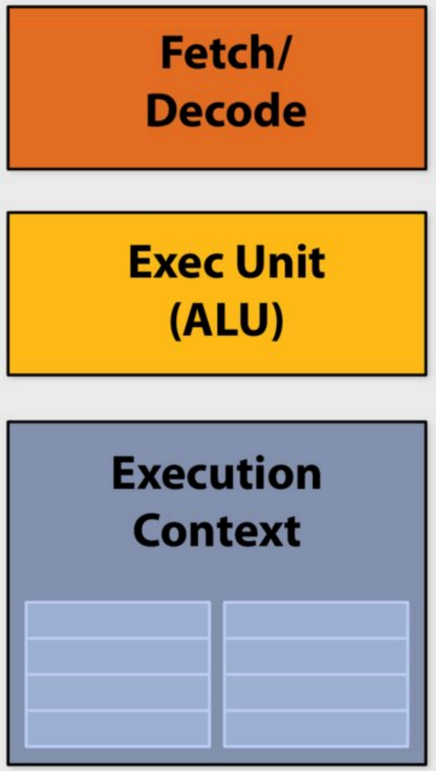
# Pre multi-core era processor

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




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

# Multi-core era processor

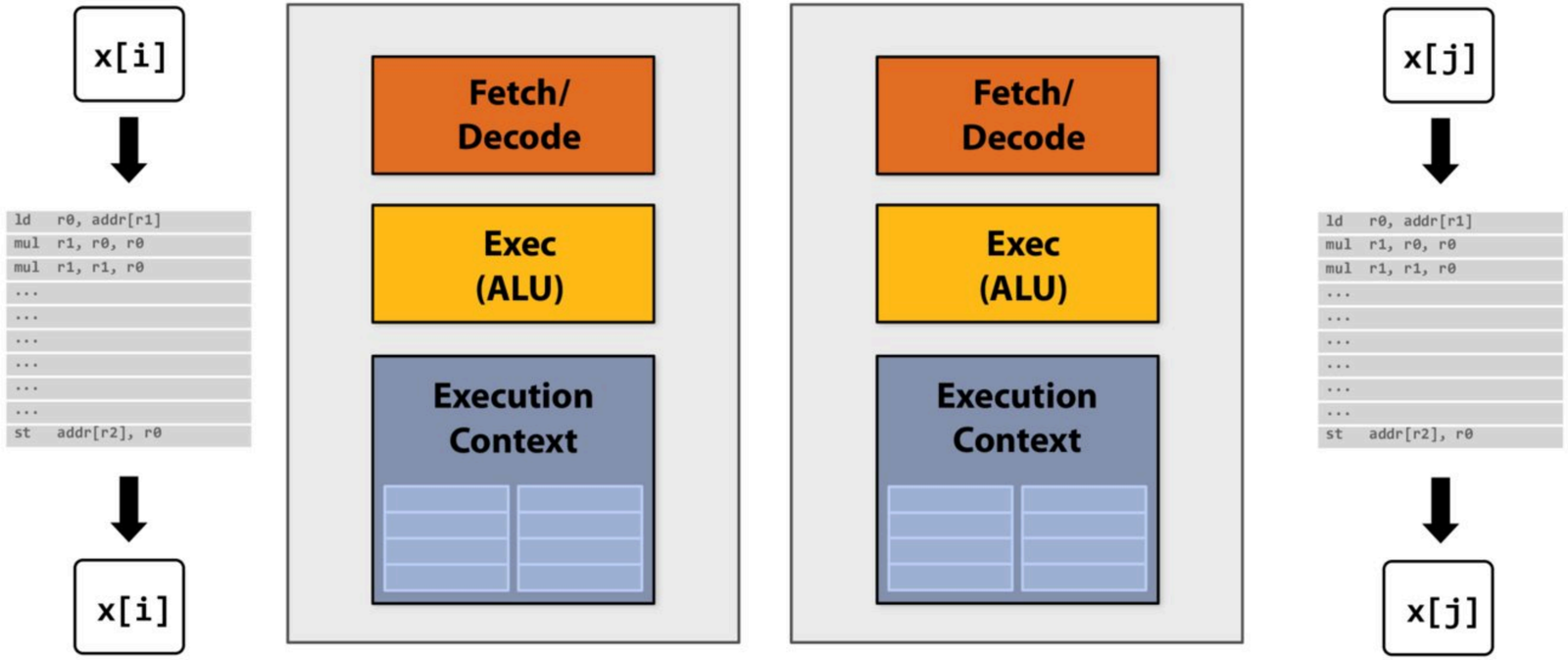



## Idea #1:

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




**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

```
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
}
```

```
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;
   }
}
```

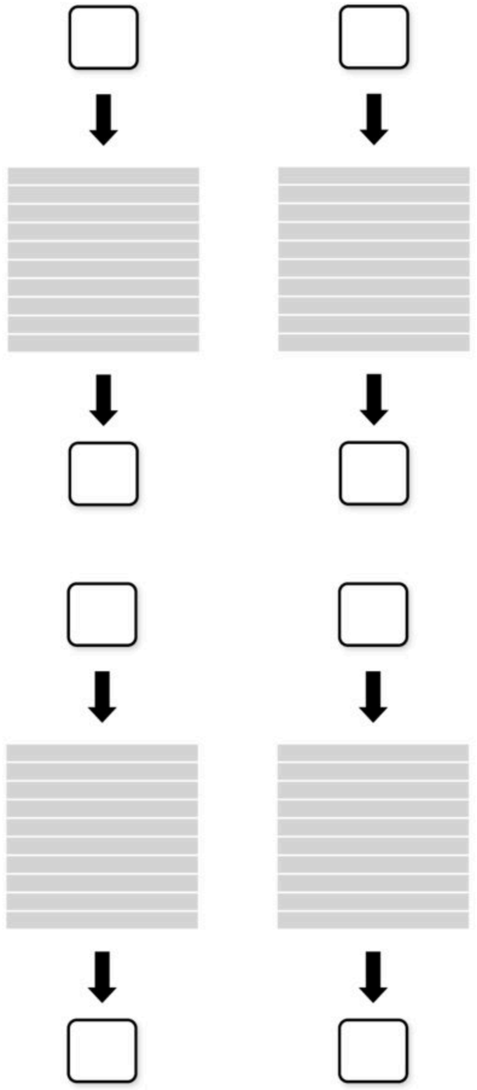
# Data-parallel expression (in Kayvon's fictitious programming language with a "forall" construct)

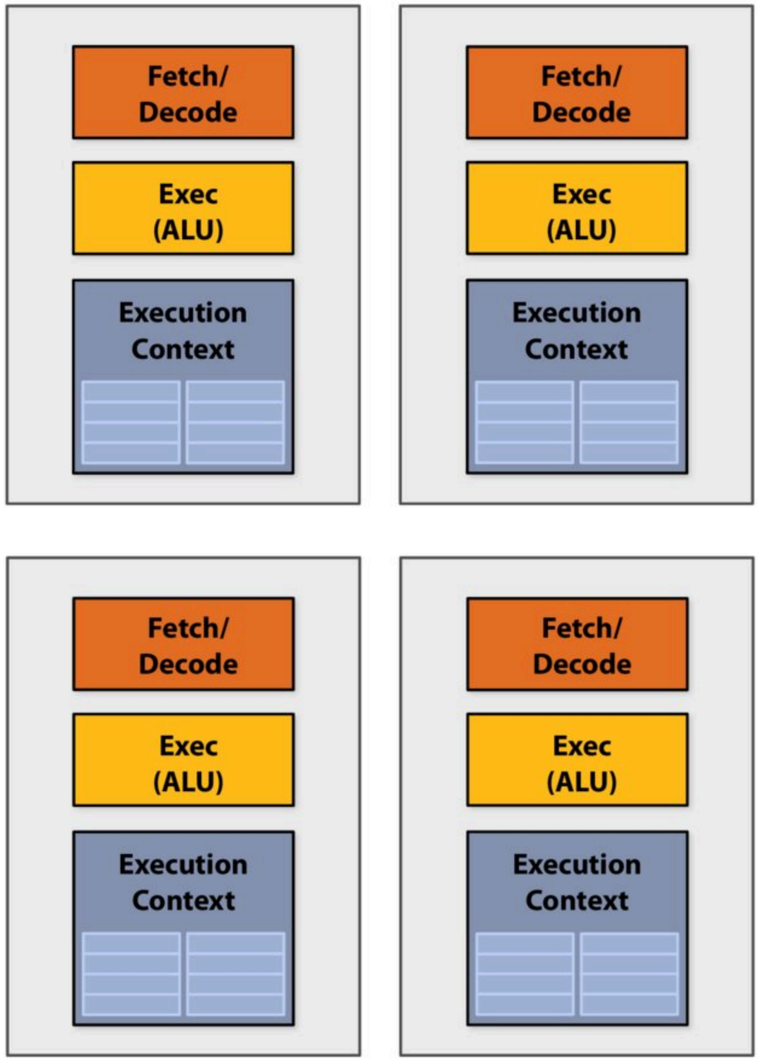
```
void sinx(int N, int terms, float* x, float* y)
{
    // declares that loop iterations are independent
    forall (int i from 0 to N)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6;  // 3!
        int sign = -1;

        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')**

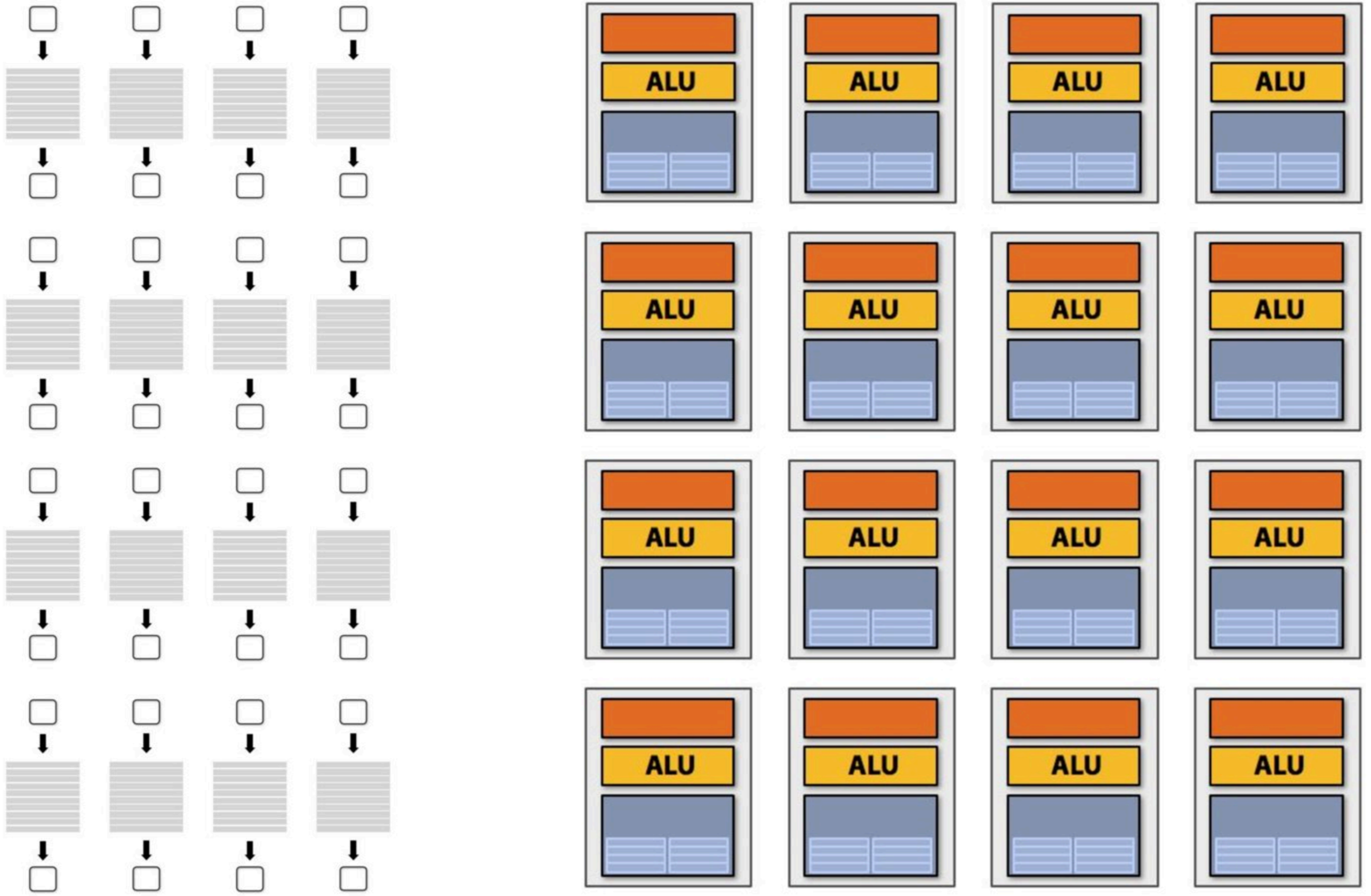
**With this information, you could imagine how a compiler might automatically generate threaded code for you.**

# Four cores: compute four elements in parallel





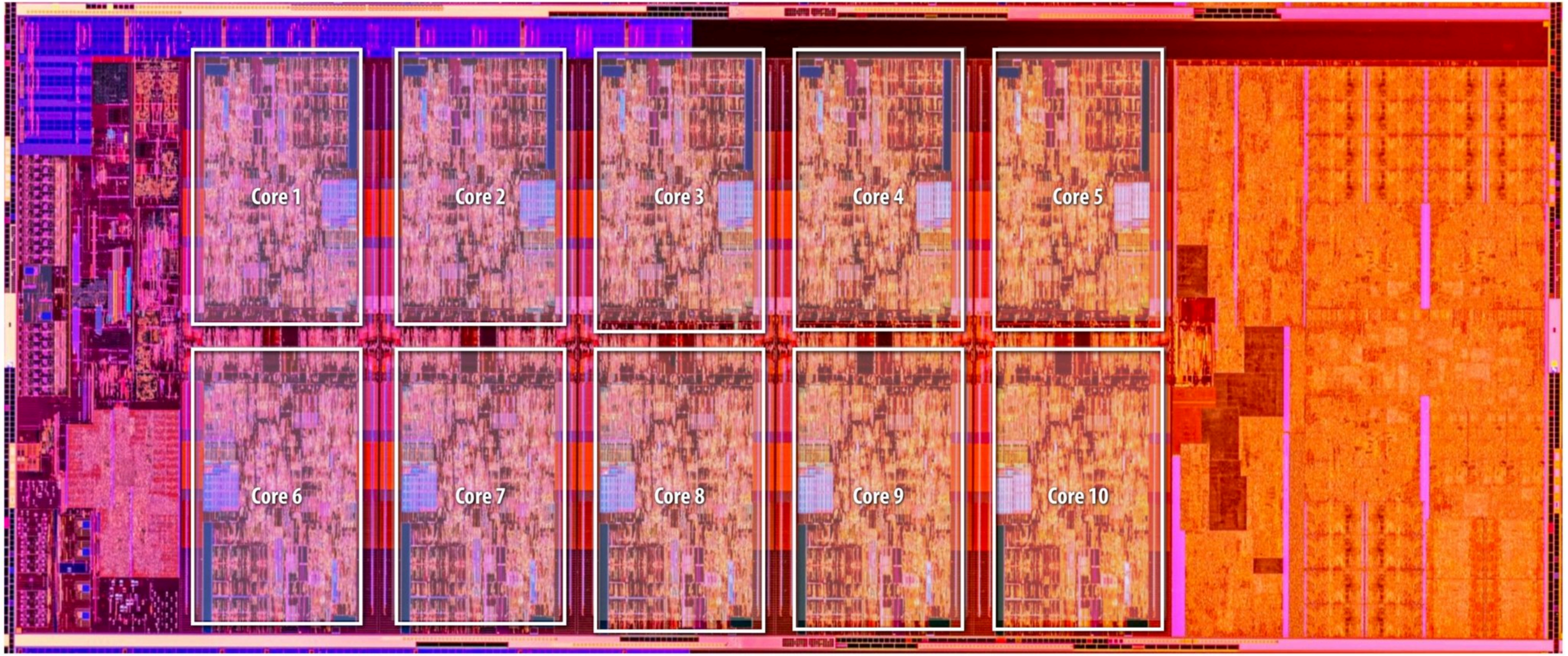
# Sixteen cores: compute sixteen elements in parallel




Sixteen cores, sixteen simultaneous instruction streams

# Example: multi-core CPU

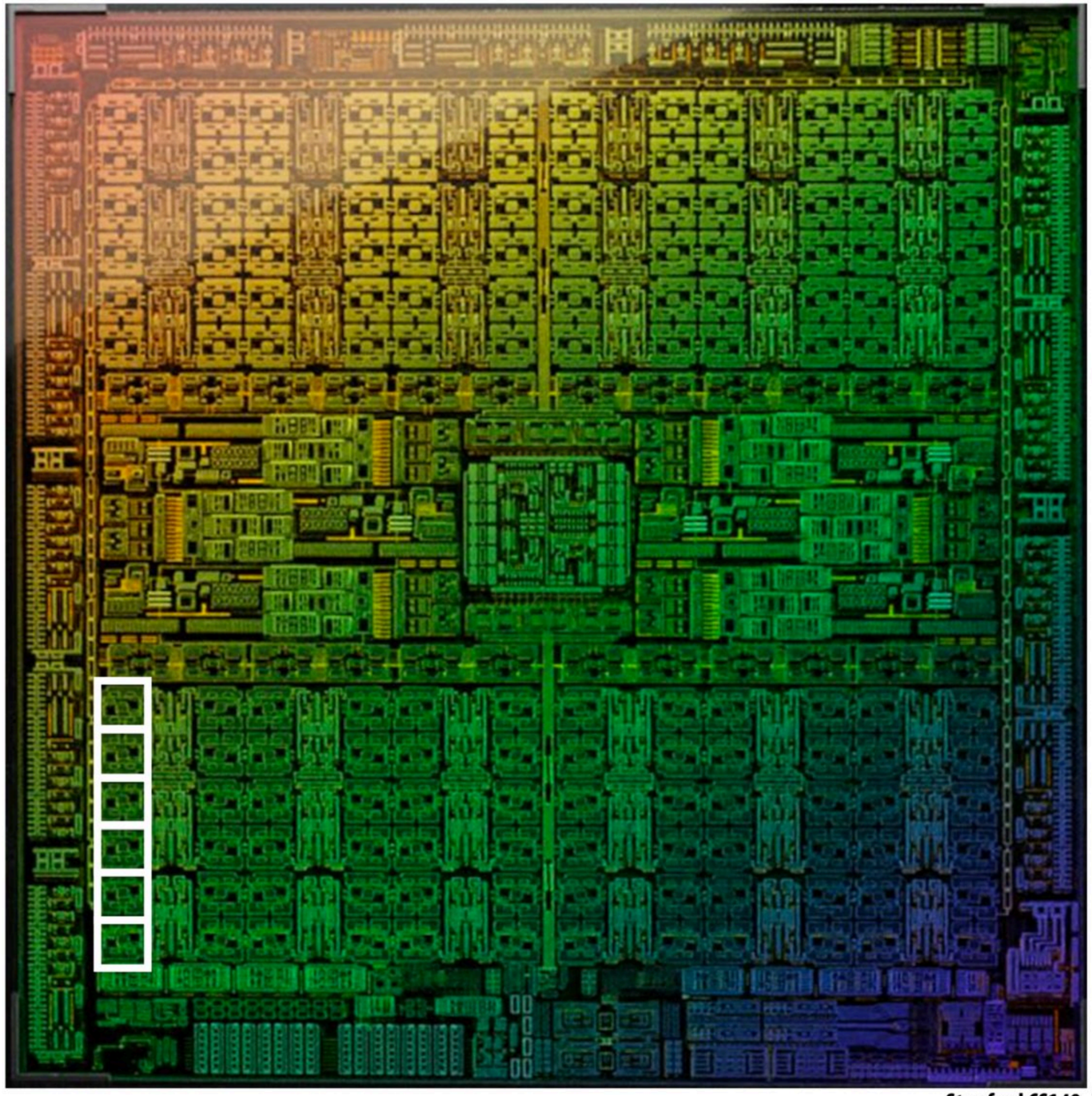
## Intel "Comet Lake" 10th Generation Core i9 10-core CPU (2020)



# Multi-core GPU

## GeForce RTX 4090 (2022)

## 144 processing blocks (called SMs)



# Apple A15 Bionic

## Two “big cores” + four “small” cores

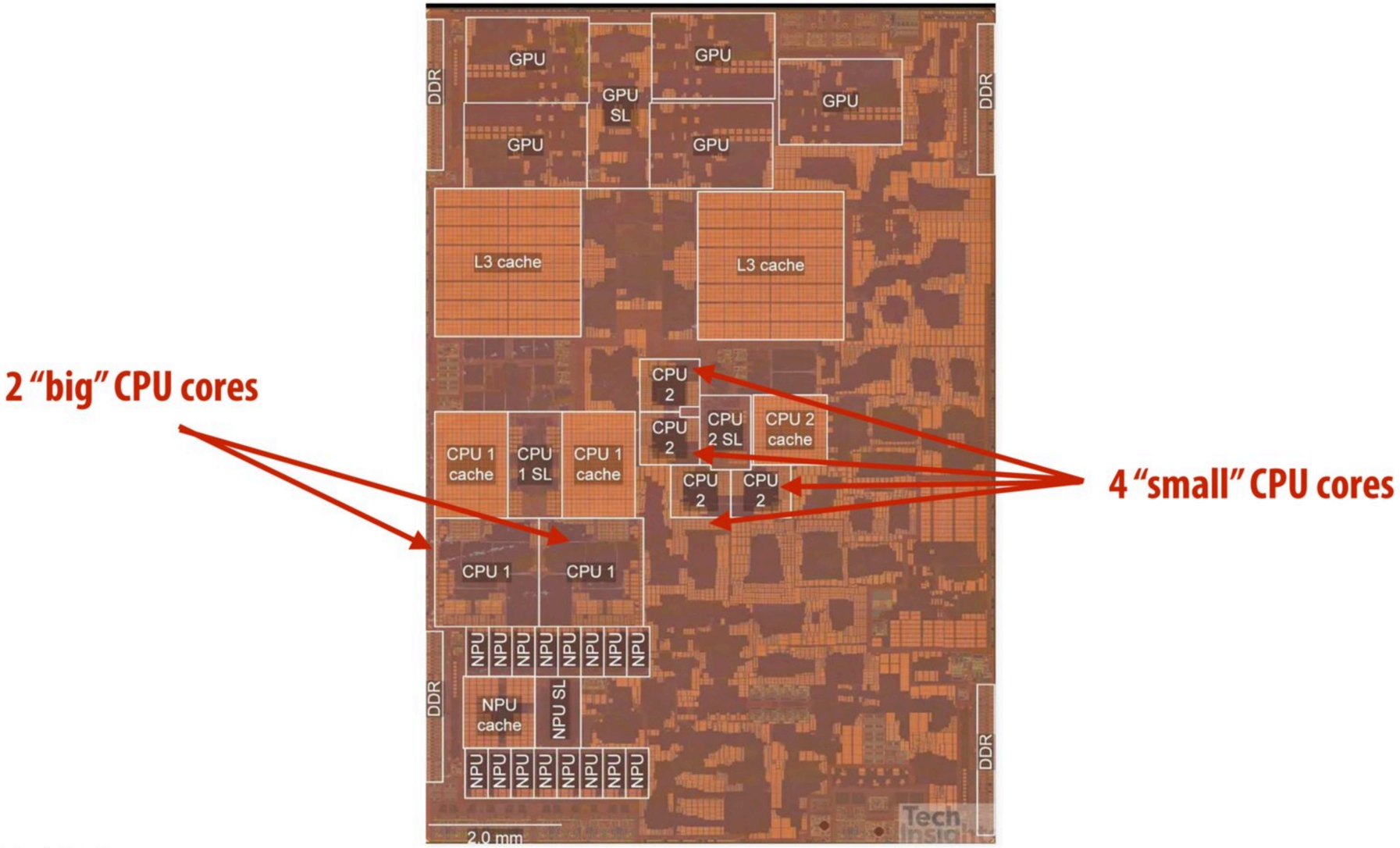
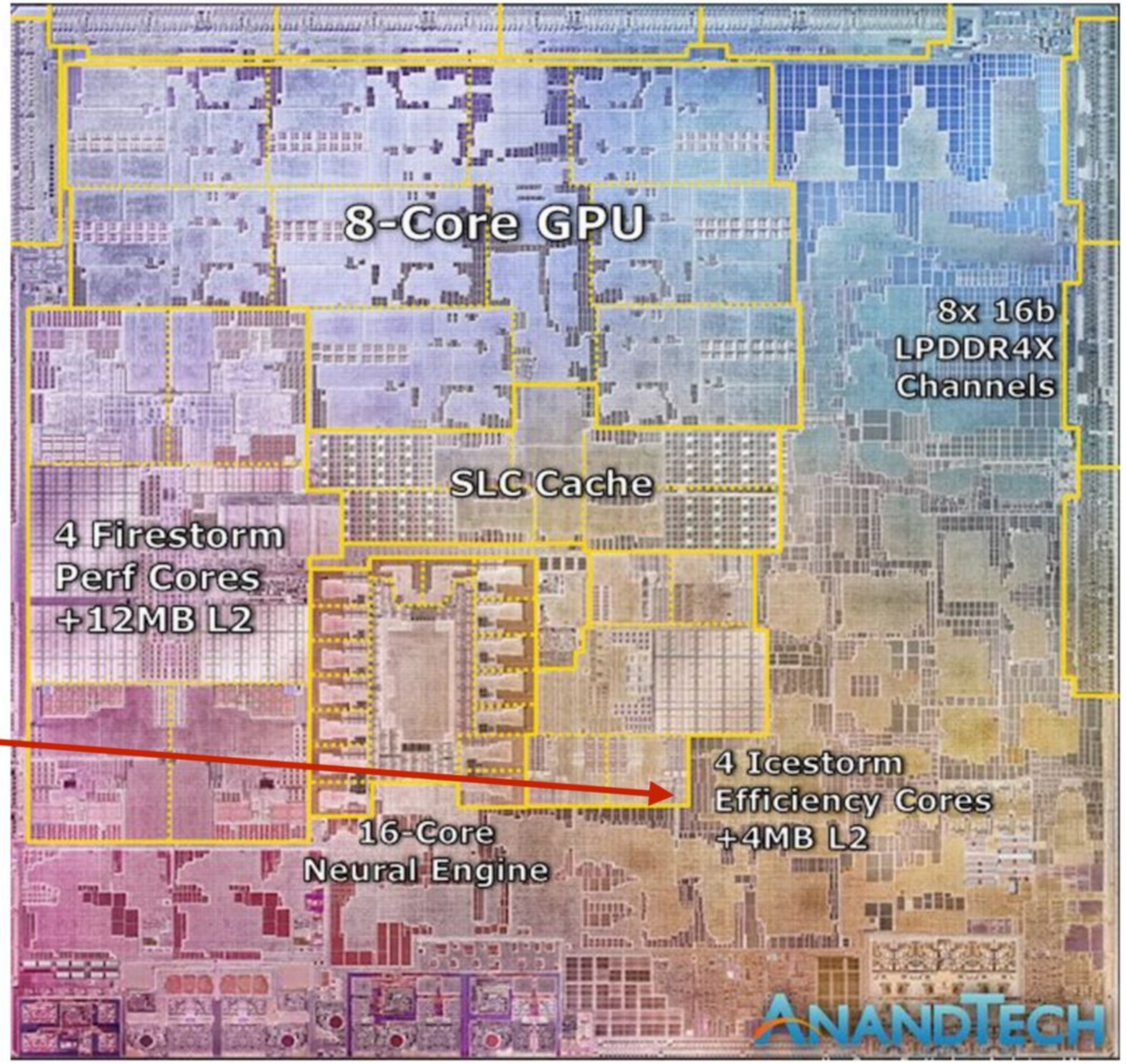



Image Credit: TechInsights Inc.

# 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

Image Credit: Anandtech

# 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;
   }
}
```

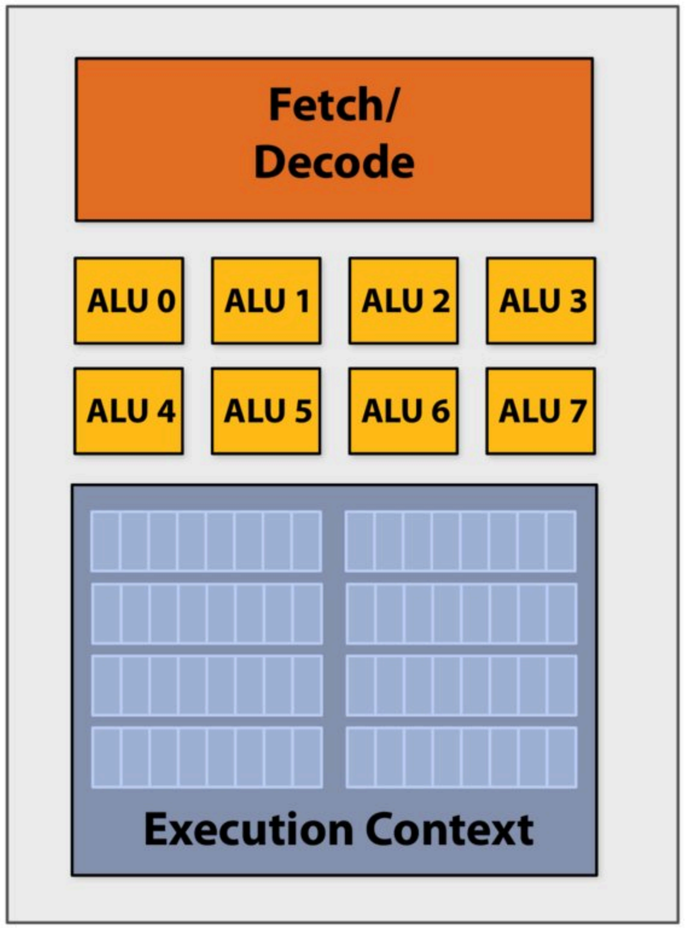
**Another interesting property of this code:**

**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**

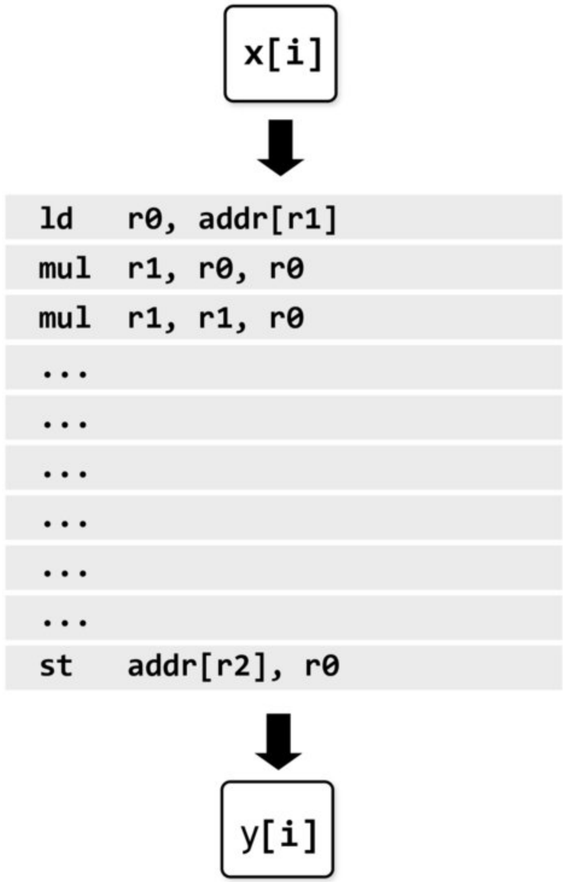
**Same instruction broadcast to all ALUs**
**This operation is executed in parallel on all ALUs**

# 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)**



# Vector program (using AVX intrinsics)

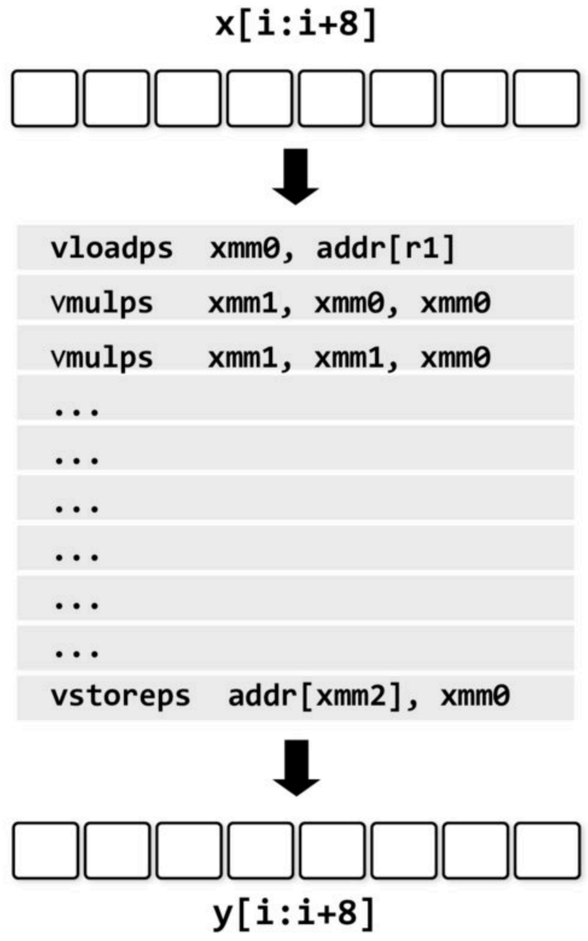
```
#include <immintrin.h>

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)

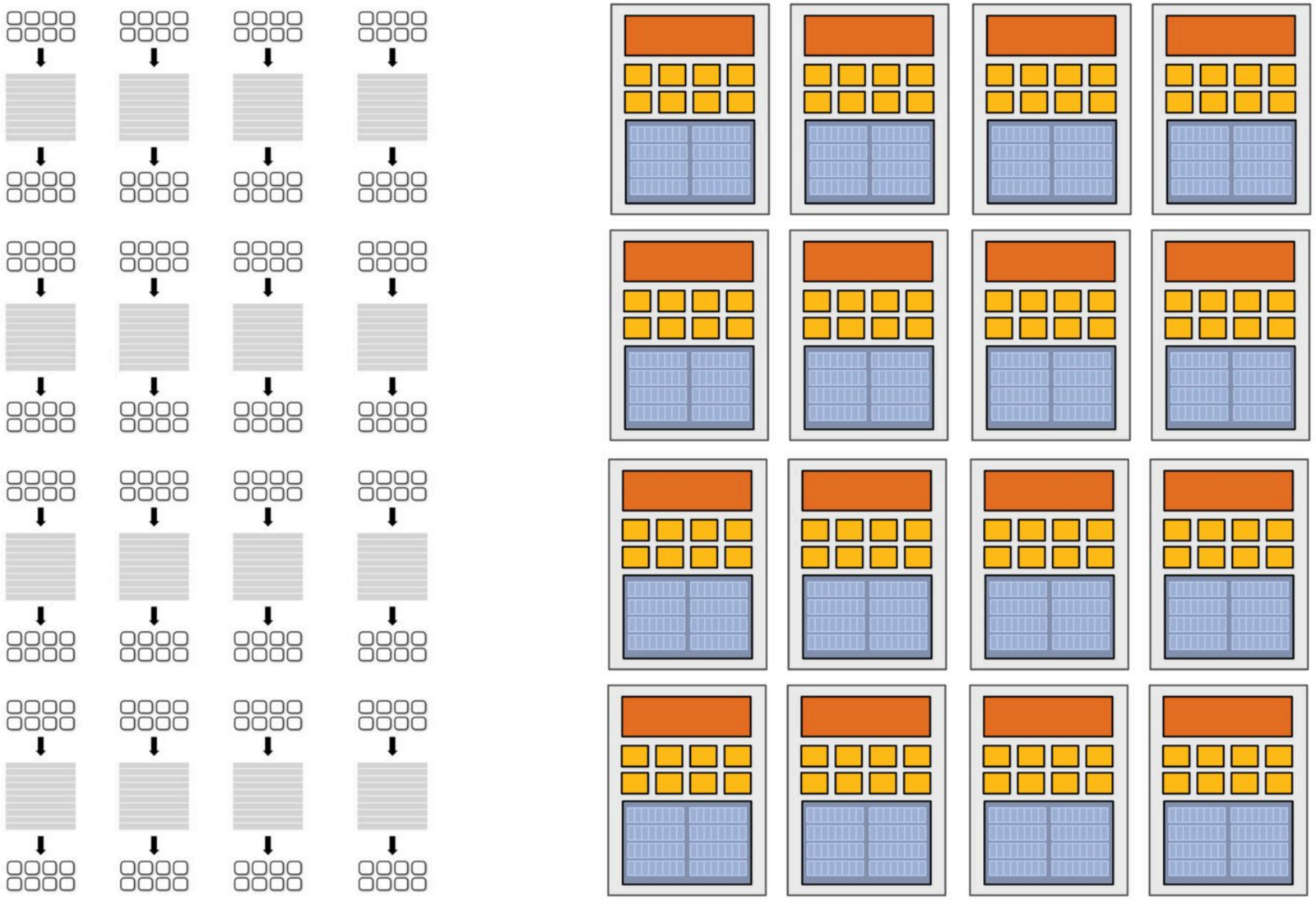
```
#include <immintrin.h>

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);
   }
}
```




**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**

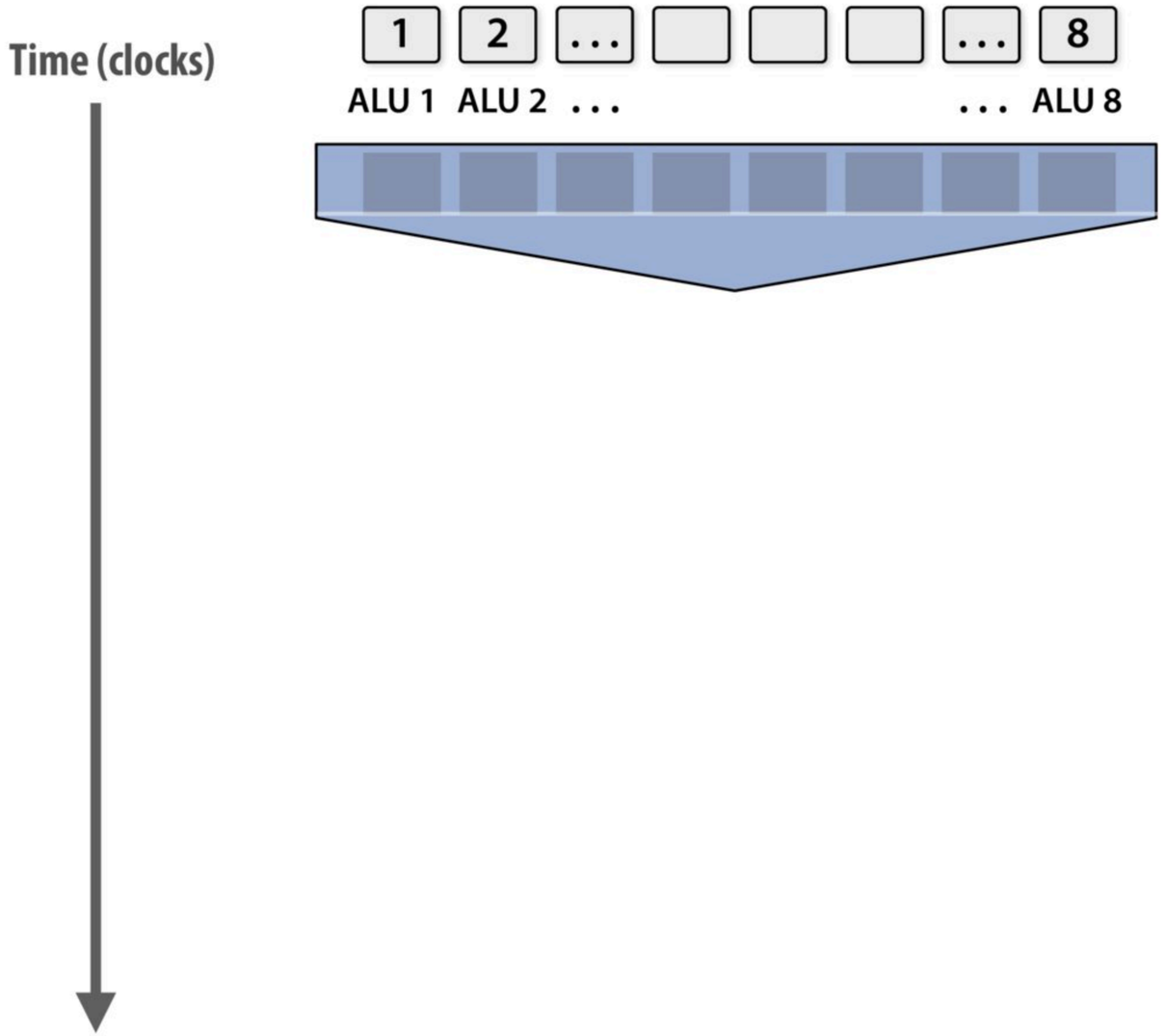
# Data-parallel expression (in Kayvon's fictitious programming language with a "forall" construct)

```
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.**

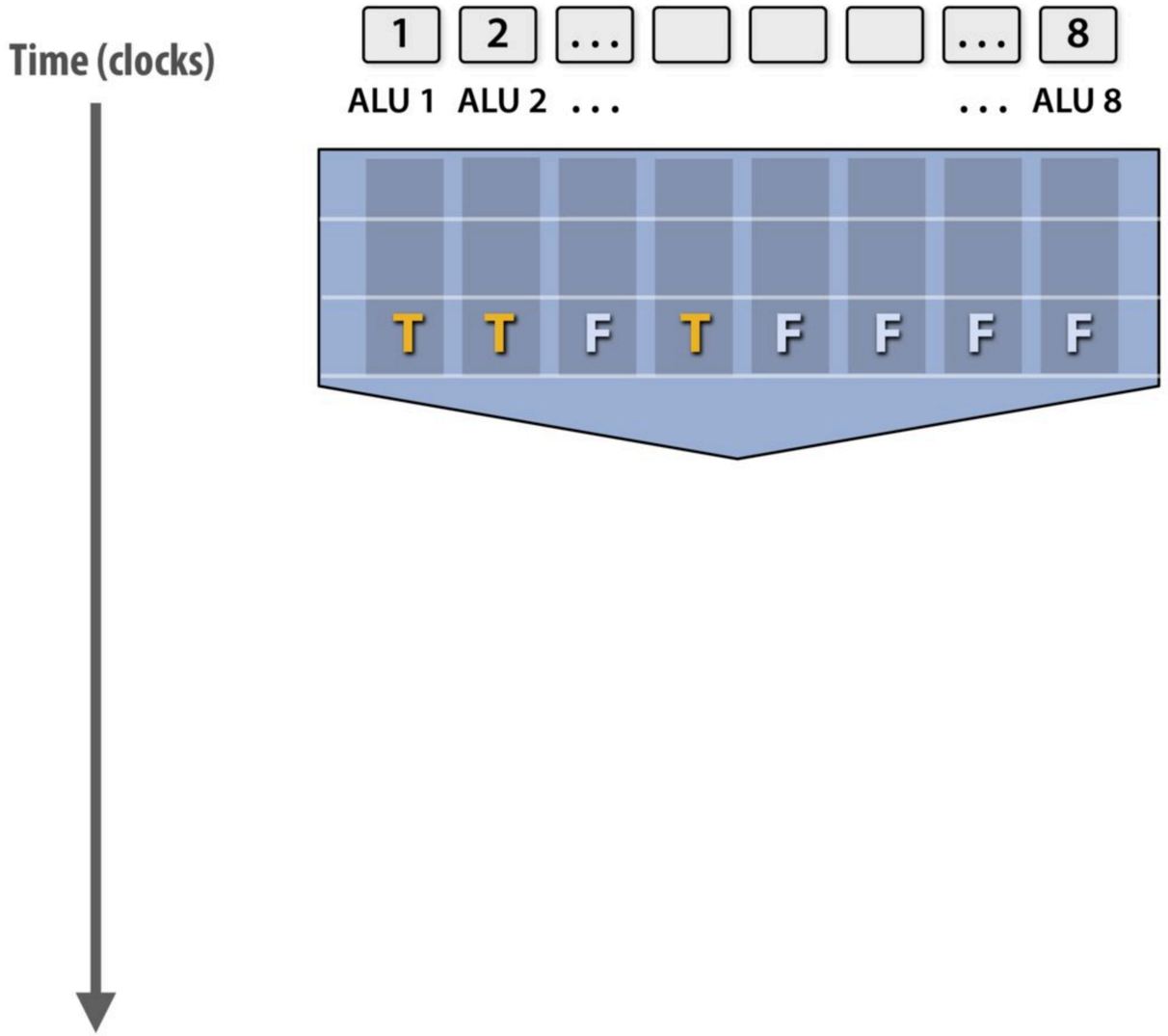
**This abstraction can facilitate automatic generation of both multi-core parallel code, and vector instructions to make use of SIMD processing capabilities within a core.**

# What about conditional execution?



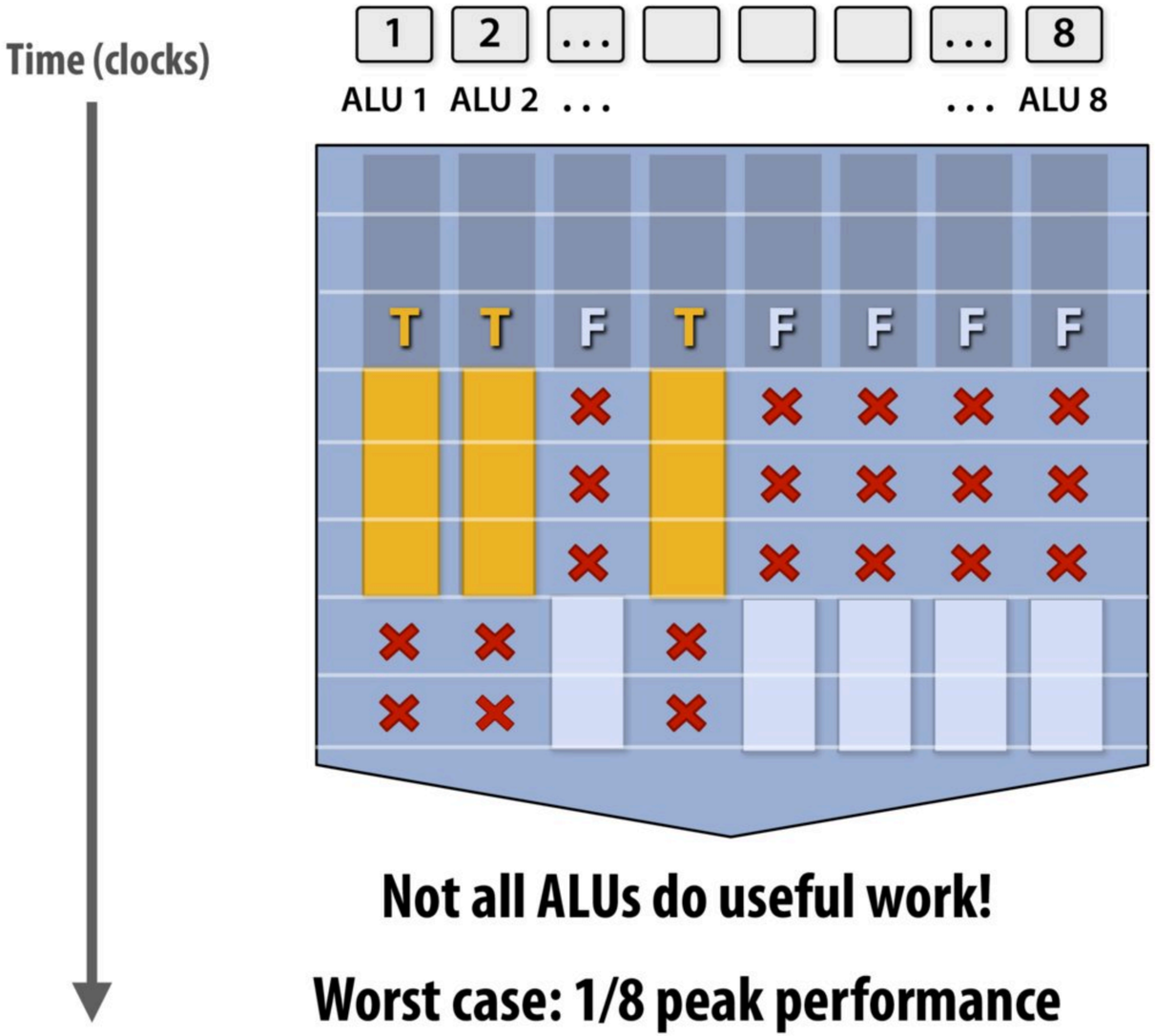

```
forall (int i from 0 to N) {
    float t = x[i];
    <unconditional code>

    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?




```
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

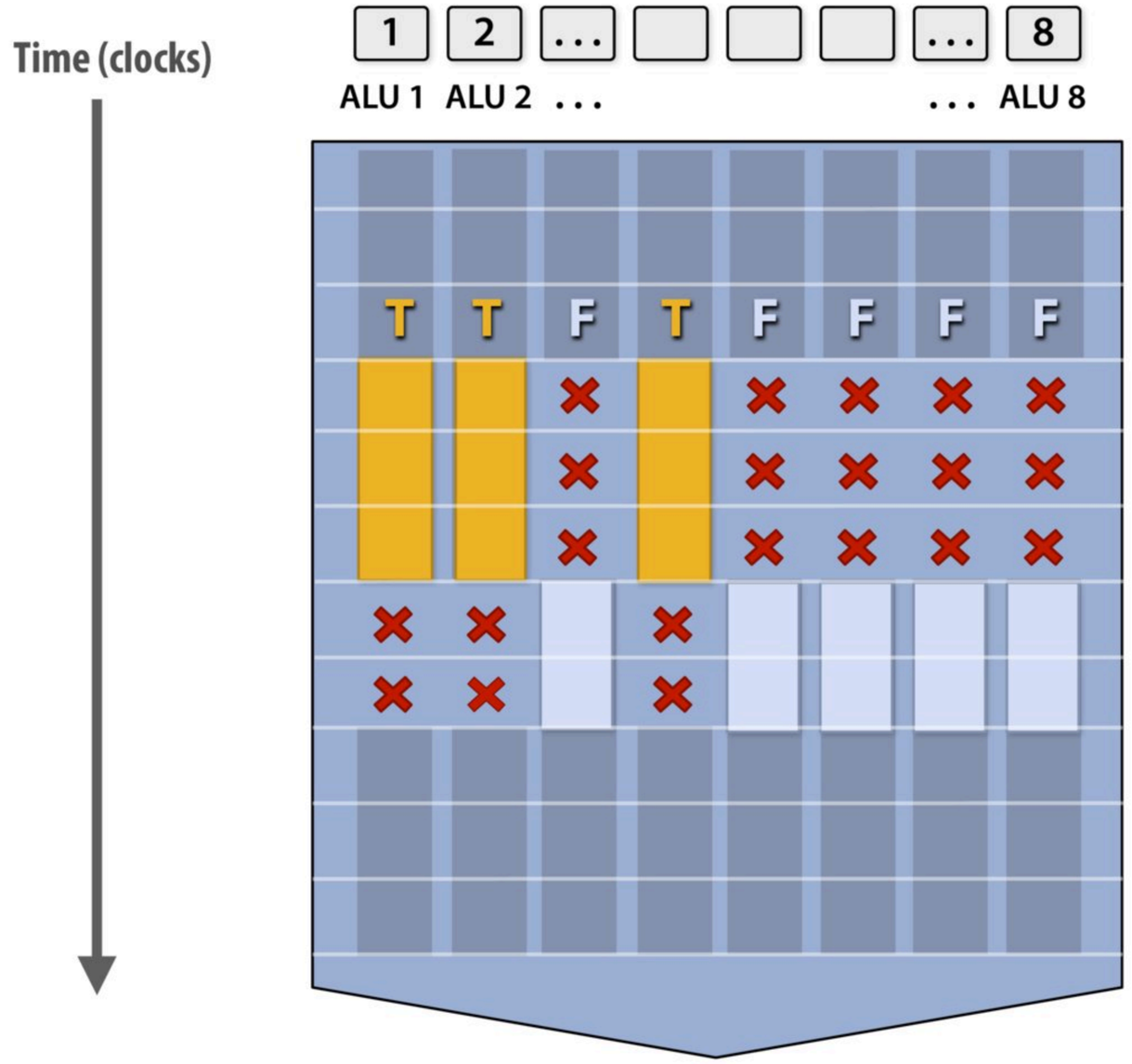



```
forall (int i from 0 to N) {
    float t = x[i];
    <unconditional code>

    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;
}
```

**Not all ALUs do useful work!**

**Worst case: 1/8 peak performance**

# After branch: continue at full performance

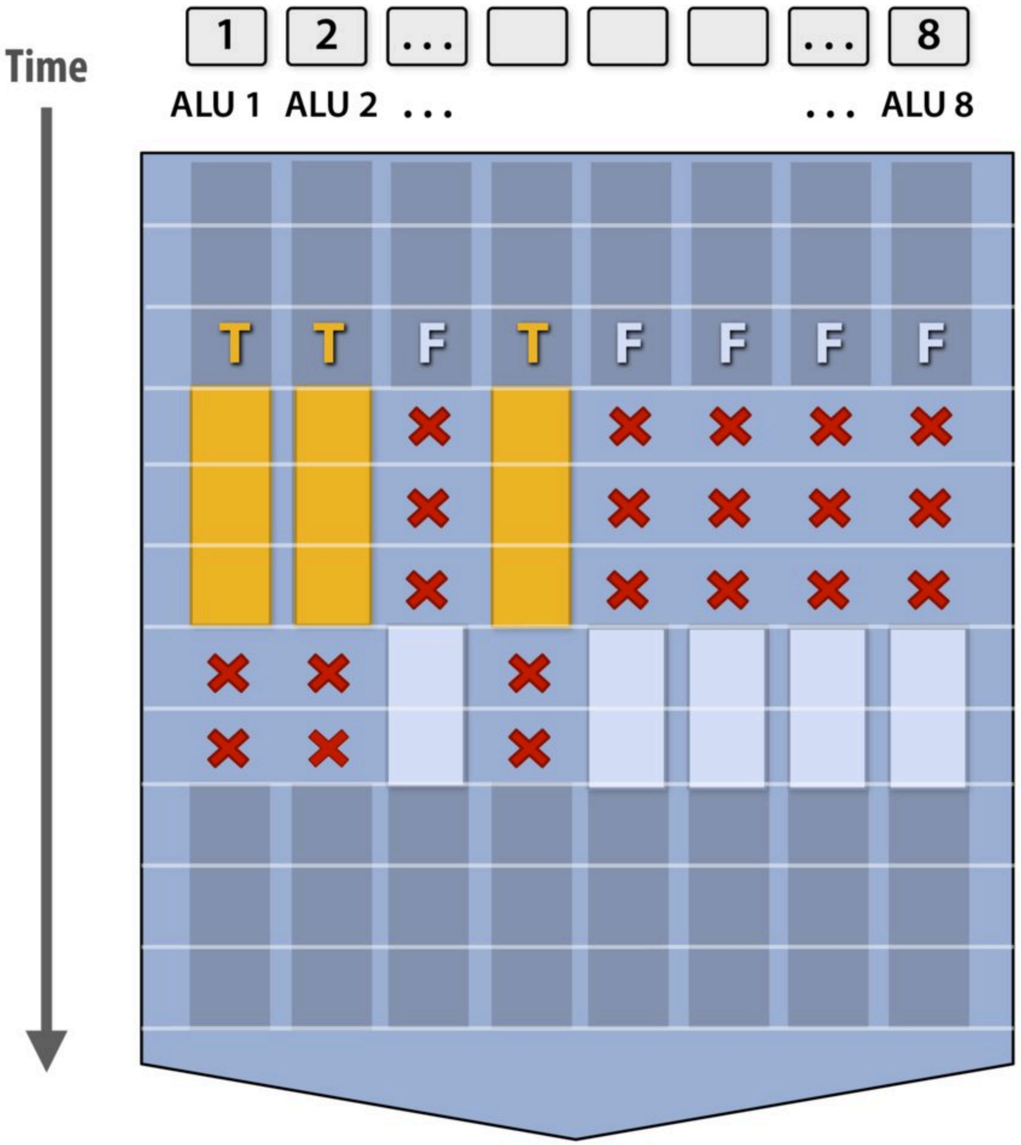



```
forall (int i from 0 to N) {
    float t = x[i];
    <unconditional code>

    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;
}
```

# 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

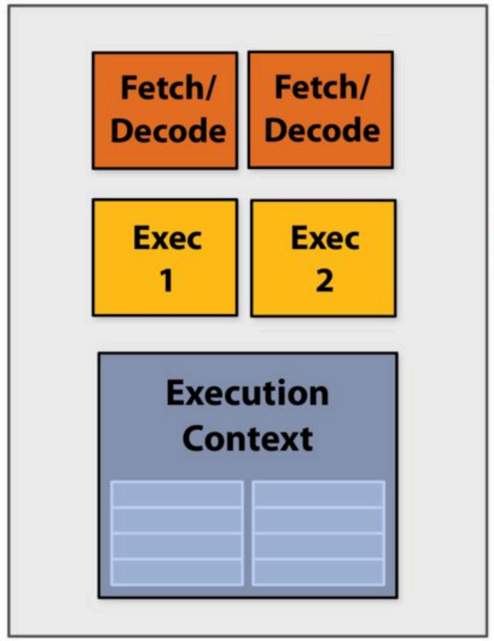
- SIMD width of most modern GPUs ranges from 8 to 32
  - Divergent execution can be a big issue
    (poorly written code might execute at 1/32 the peak capability of the machine!)

# Summary: three different forms of parallel execution

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

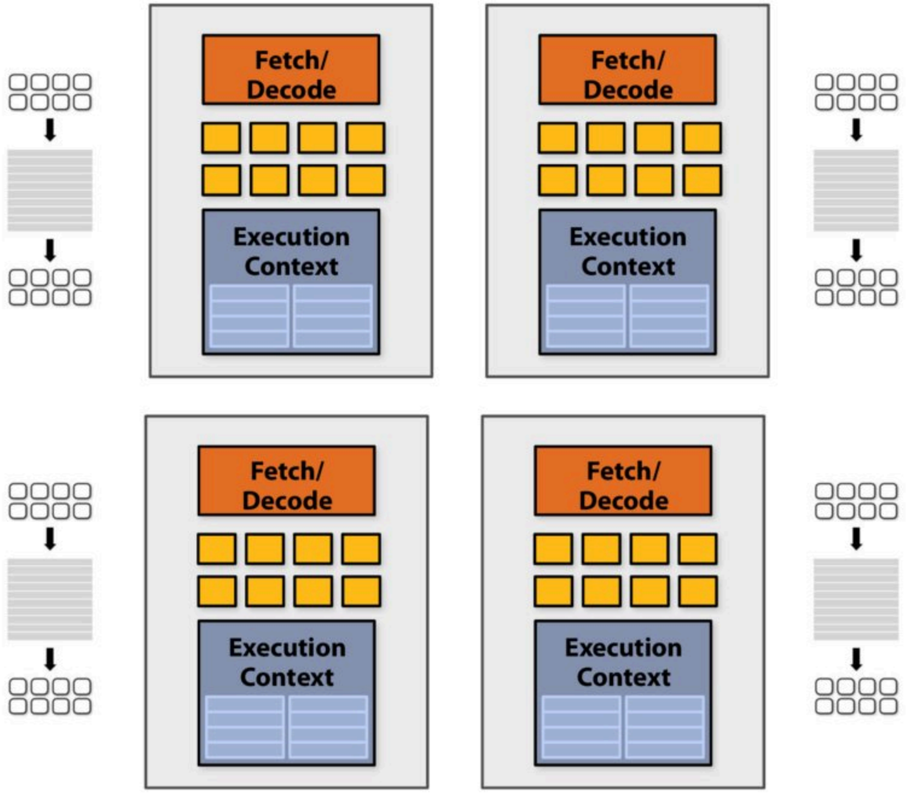
- **SIMD:** multiple ALUs controlled by same instruction (within a core)
  - Efficient for data-parallel workloads: amortize control costs over many ALUs
  - Vectorization done by compiler (explicit SIMD) or at runtime by hardware (implicit SIMD)

- **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)

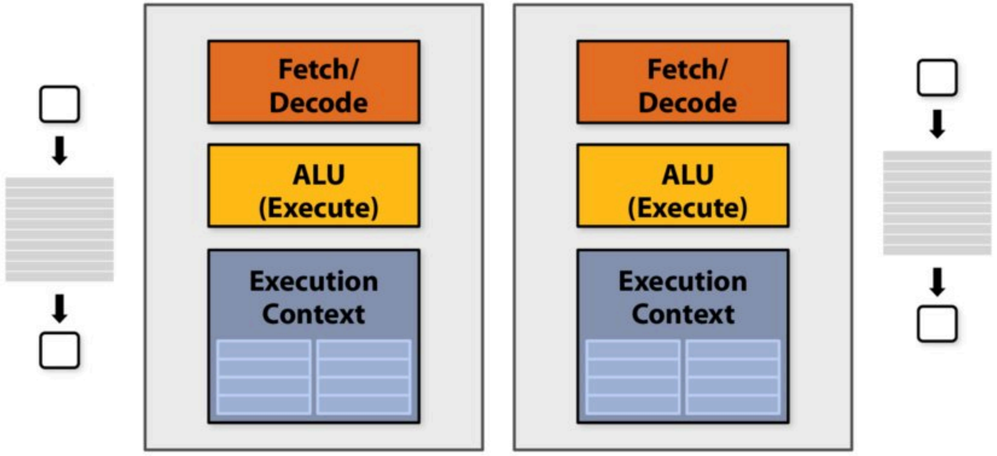



My single core, superscalar processor:
executes up to two instructions per clock
from a single instruction stream (if the
instructions are independent)

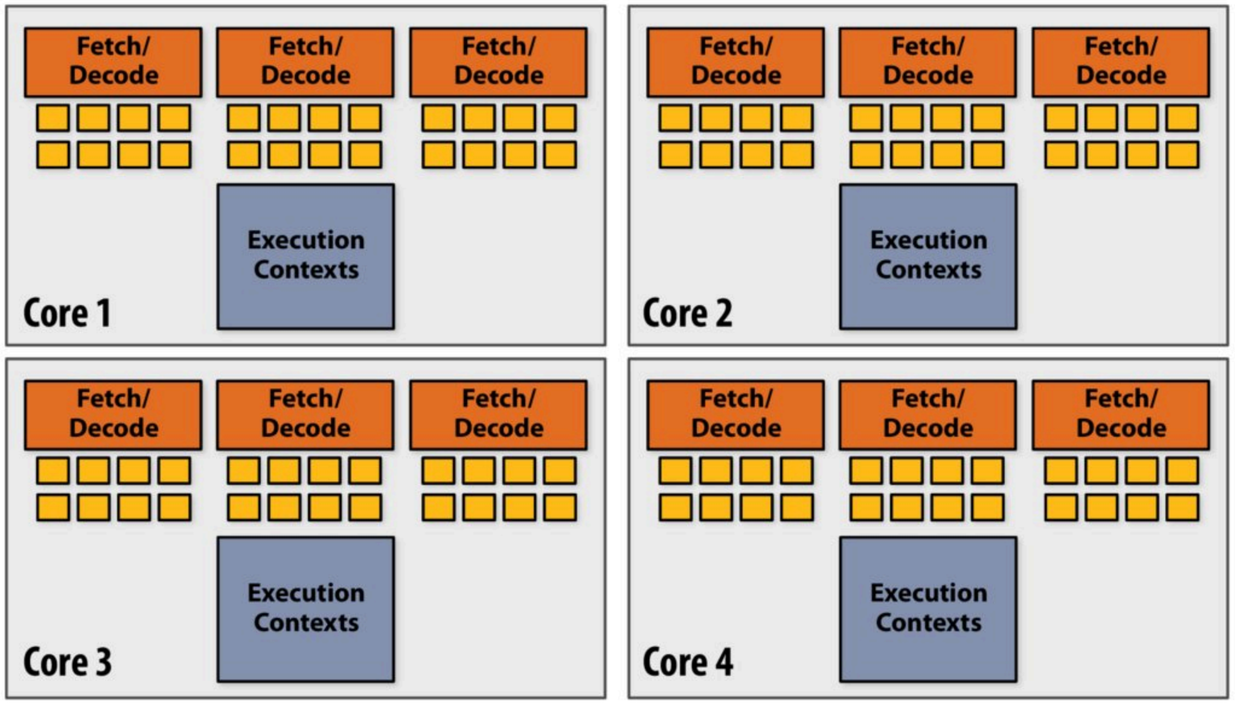
My SIMD quad-core processor:
executes one 8-wide SIMD instruction per clock
from one instruction stream on each core.




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



# Example: four-core Intel i7-7700K CPU (Kaby Lake)

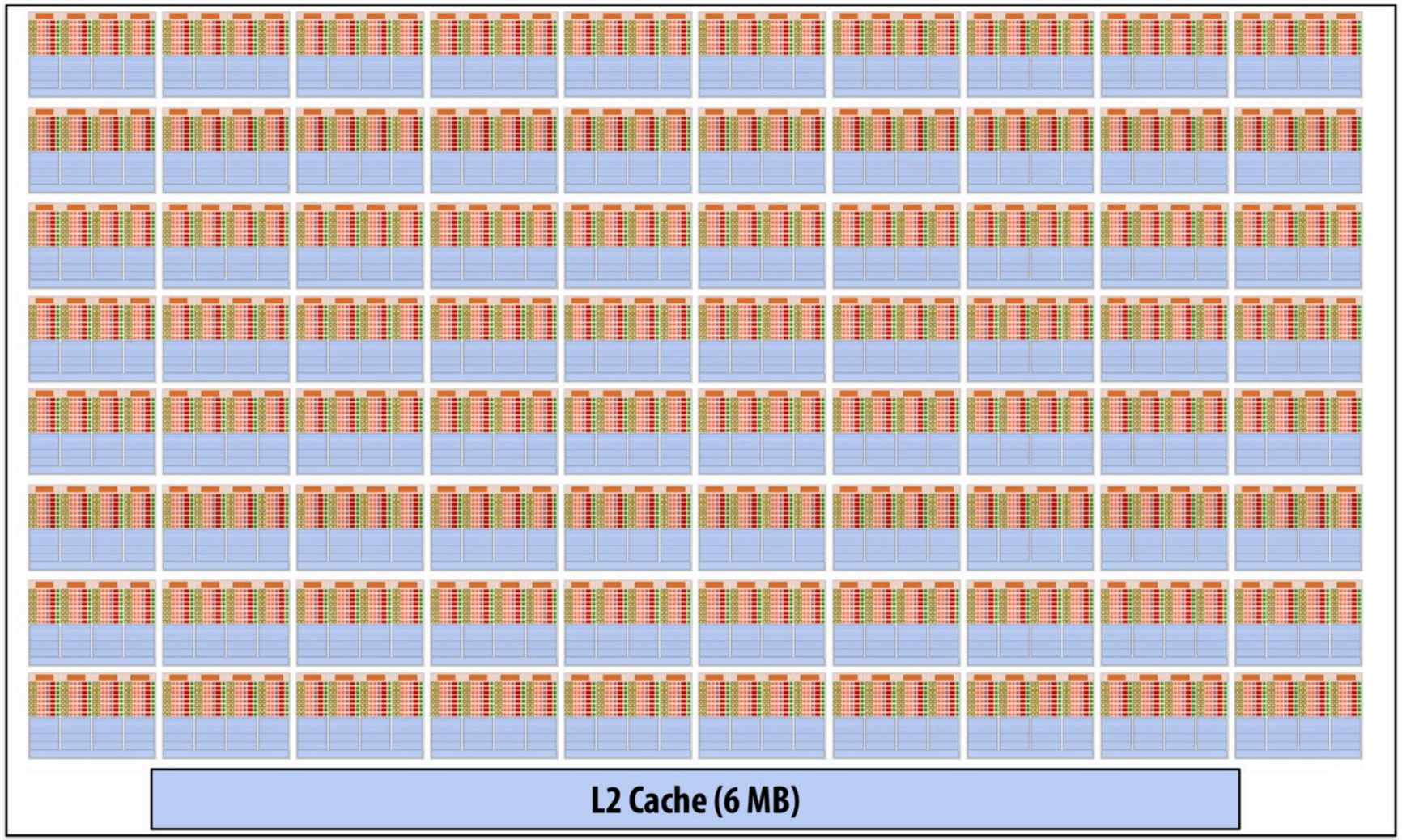



**4 core processor**
**Three 8-wide SIMD ALUs per core**
**(AVX2 instructions)**

**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)**

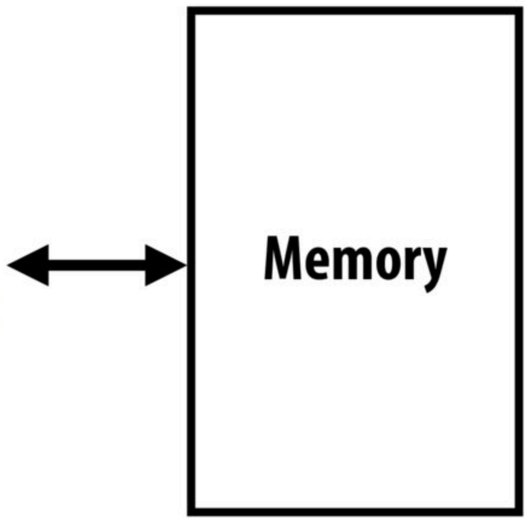
# Example: NVIDIA V100 GPU




80 “SM” cores

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

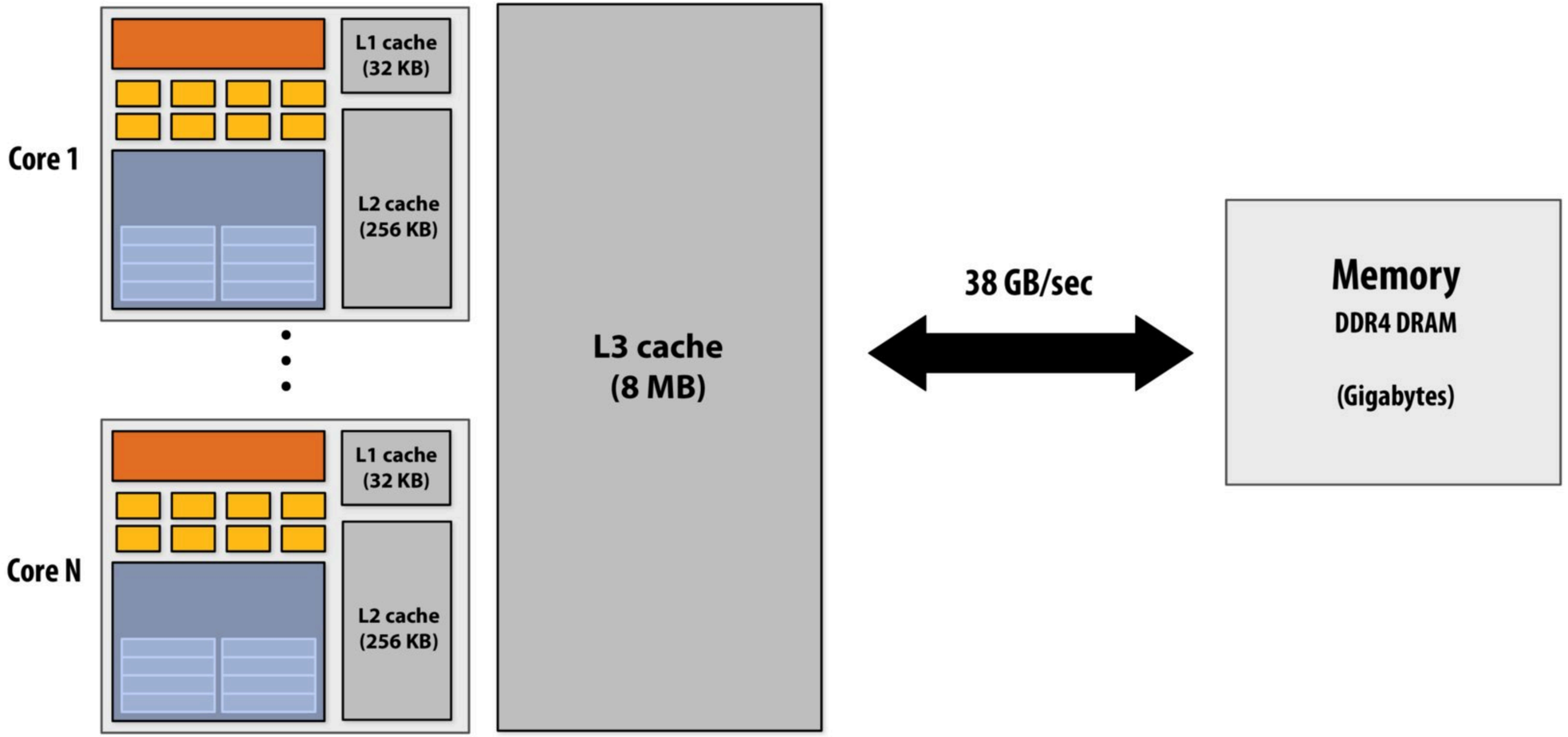
# Part 2: accessing memory

# Caches reduce length of stalls (reduce memory access latency)

**Processors run efficiently when they access data resident in caches**

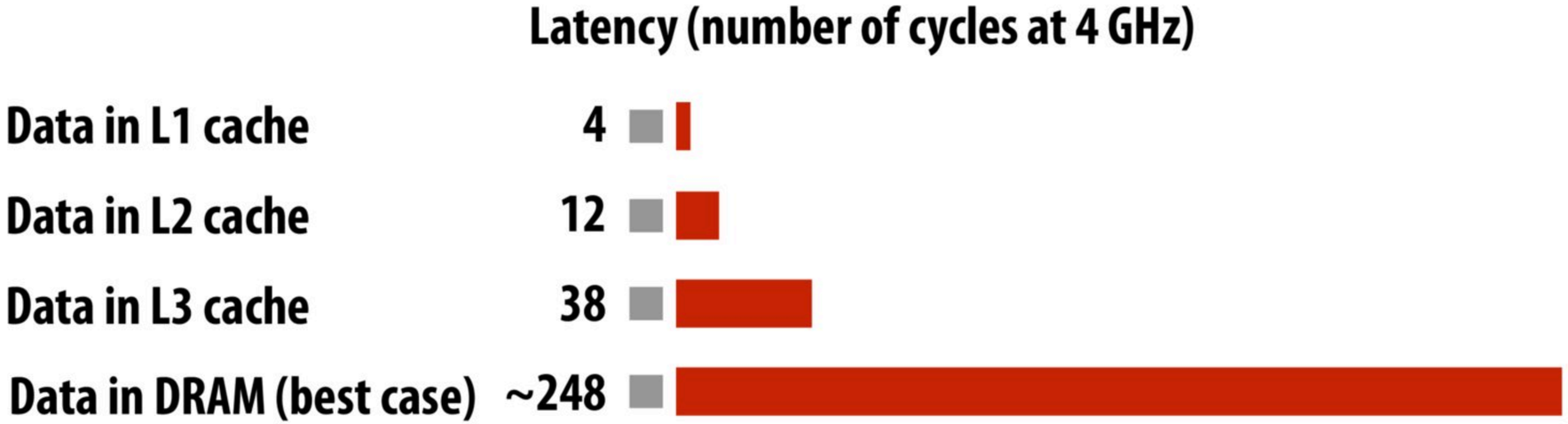
**Caches reduce memory access latency when accessing data that they have recently accessed! ***




* Caches also provide high bandwidth data transfer

# Recall: [very] long latency of data access

(Kaby Lake CPU)



# 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?**

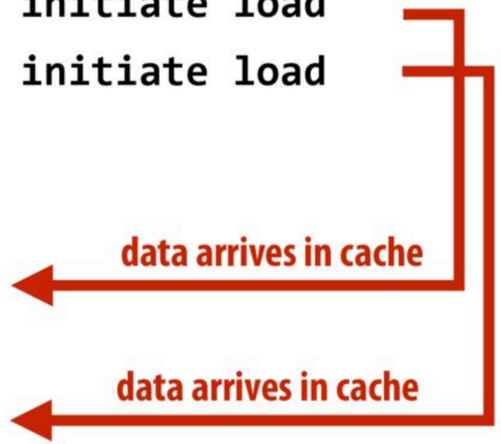
**Would your answer change if the cache had a capacity of 4 lines?**


| Address 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 (hides 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
...
...
...
...
...
...
...
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 (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…


Credit: https://www.theodysseyonline.com/the-dos-and-donts-of-dorm-laundry

Or cooking a meal…

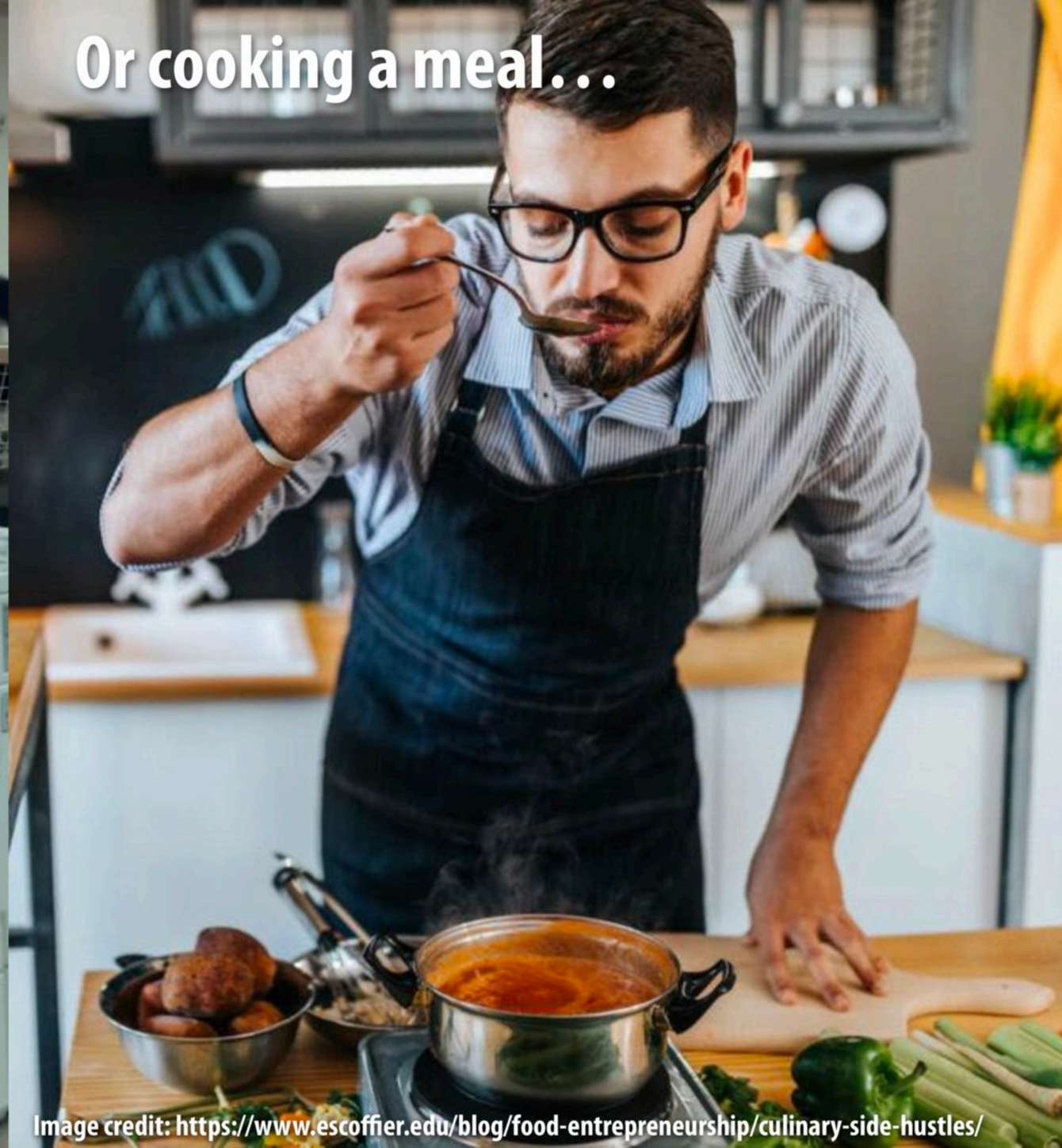
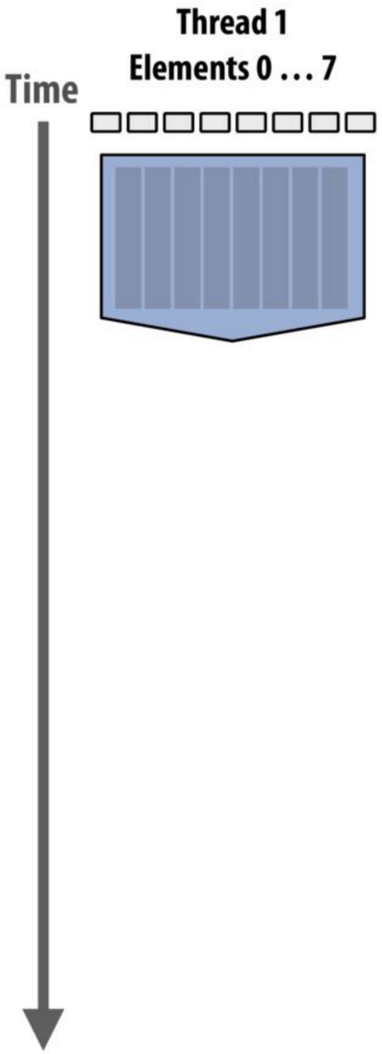


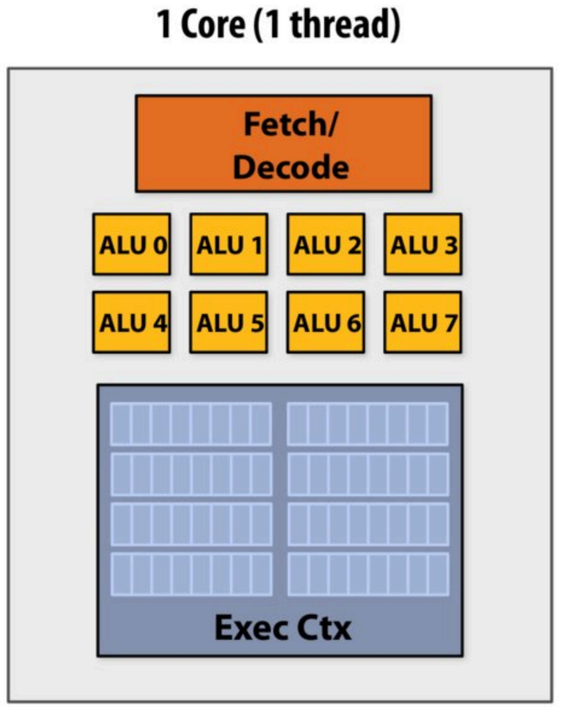
Image credit: https://www.escoffier.edu/blog/food-entrepreneurship/culinary-side-hustles/

# Multi-threading reduces stalls

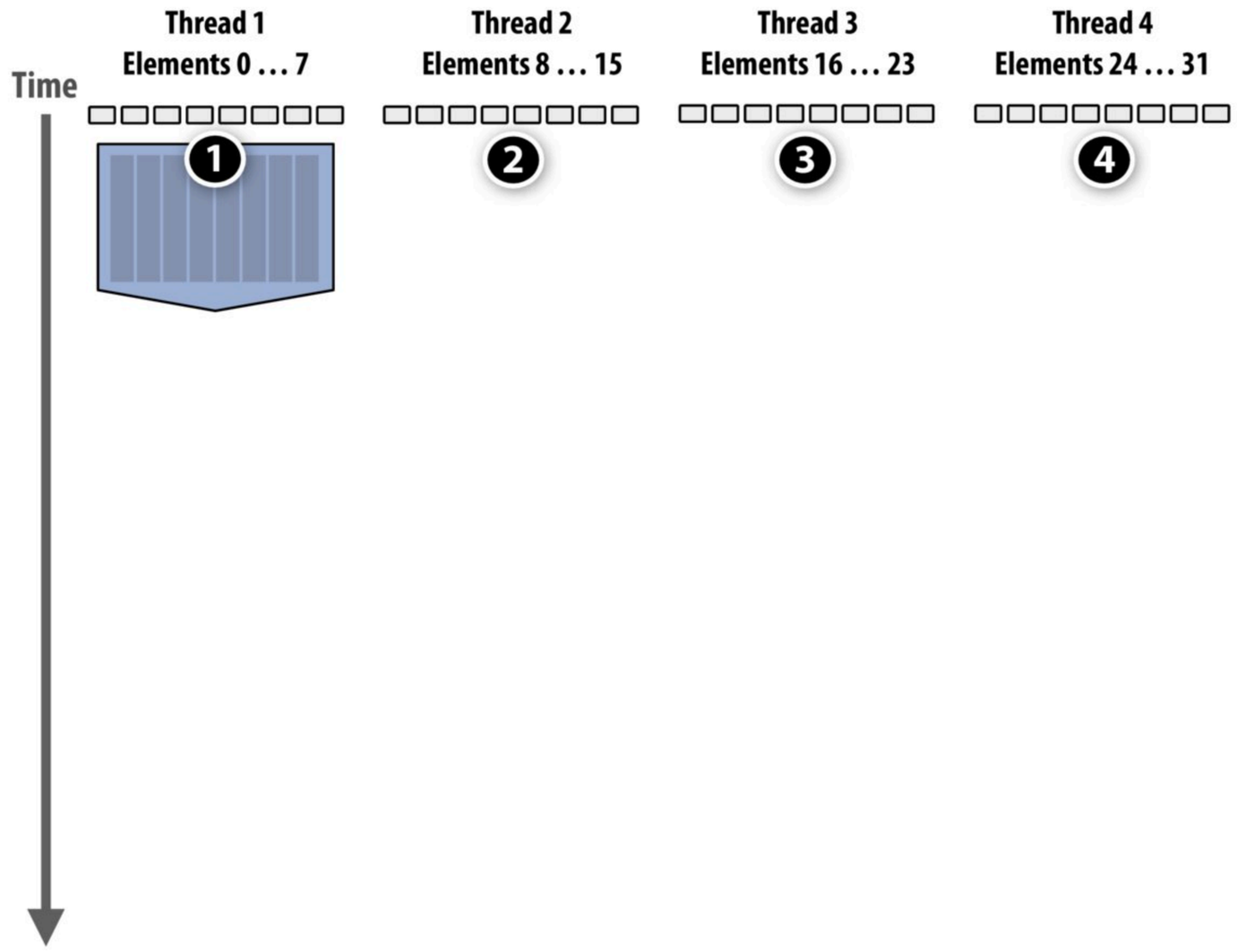
- Idea #3: interleave processing of multiple threads on the same core to hide stalls
  - If you can't make progress on the current thread… work on another one

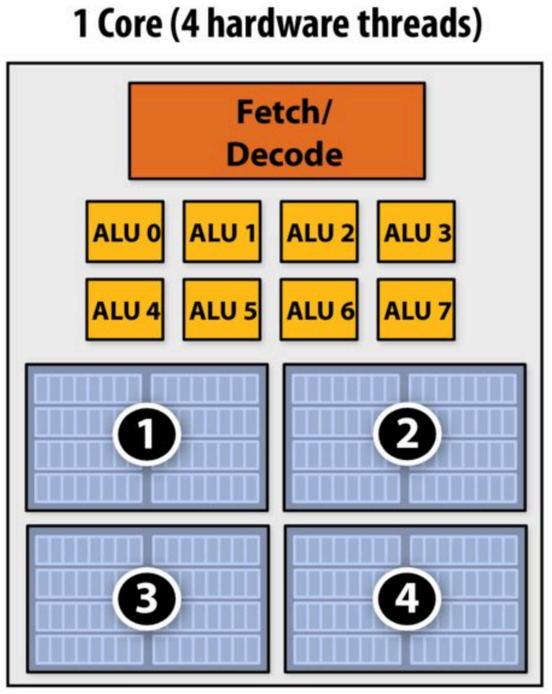
# Hiding stalls with multi-threading



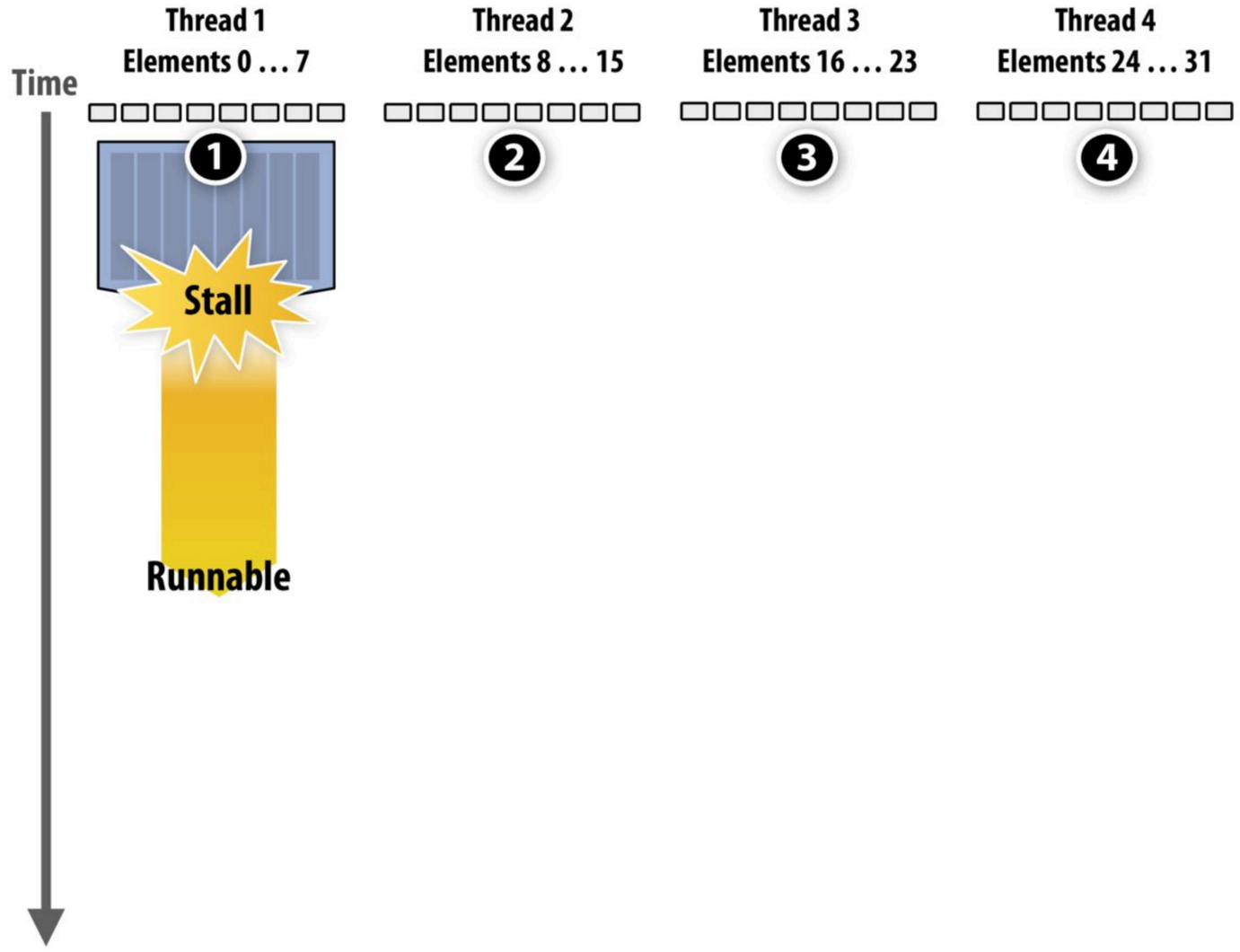



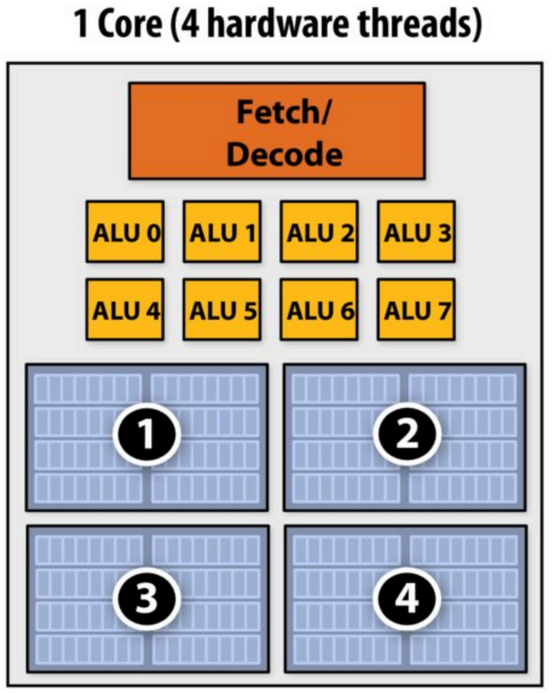
# Hiding stalls with multi-threading



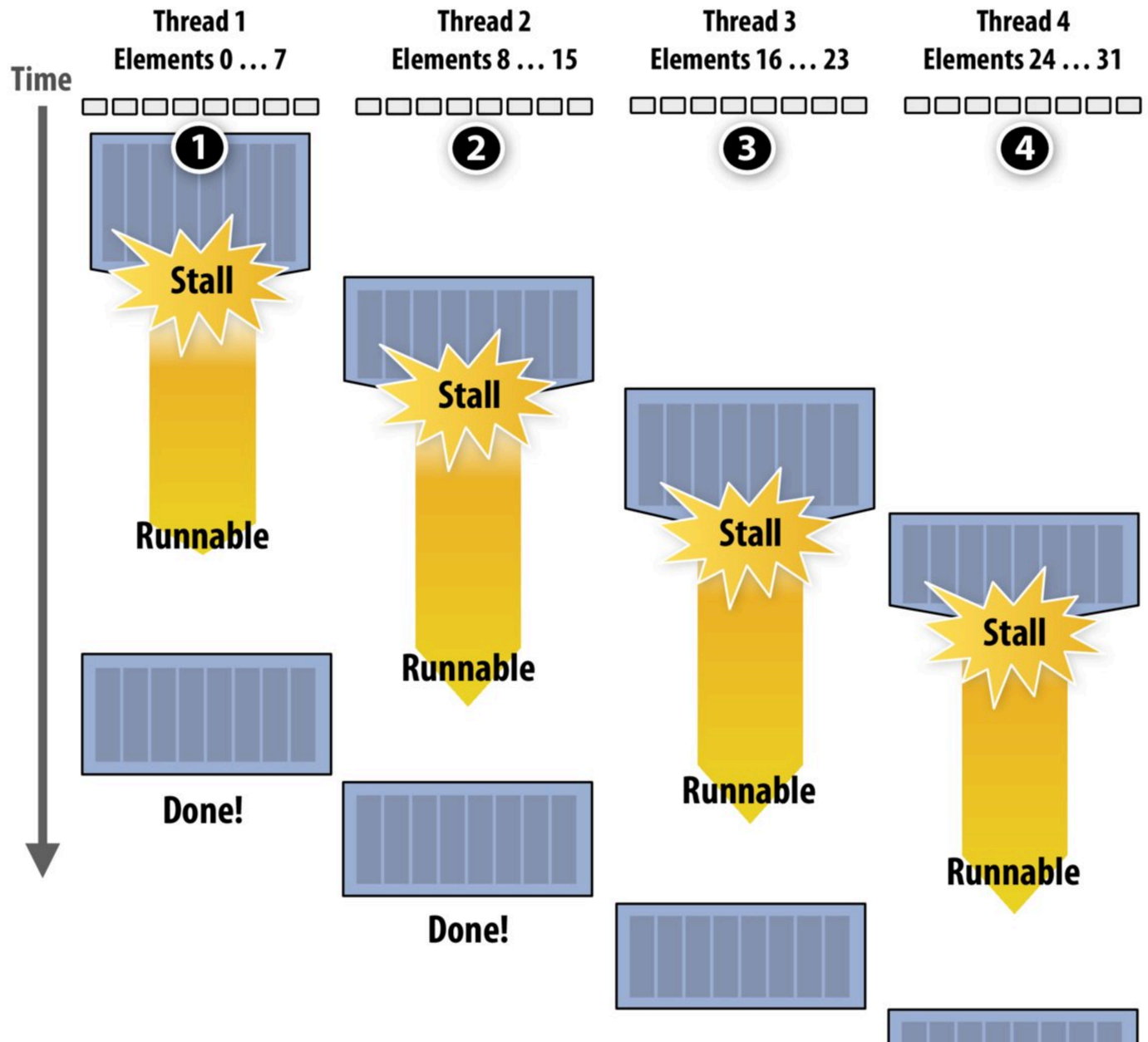



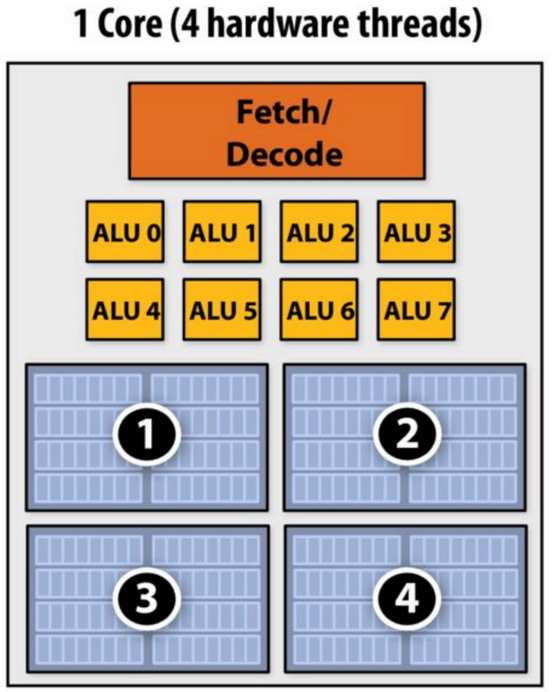
# Hiding stalls with multi-threading



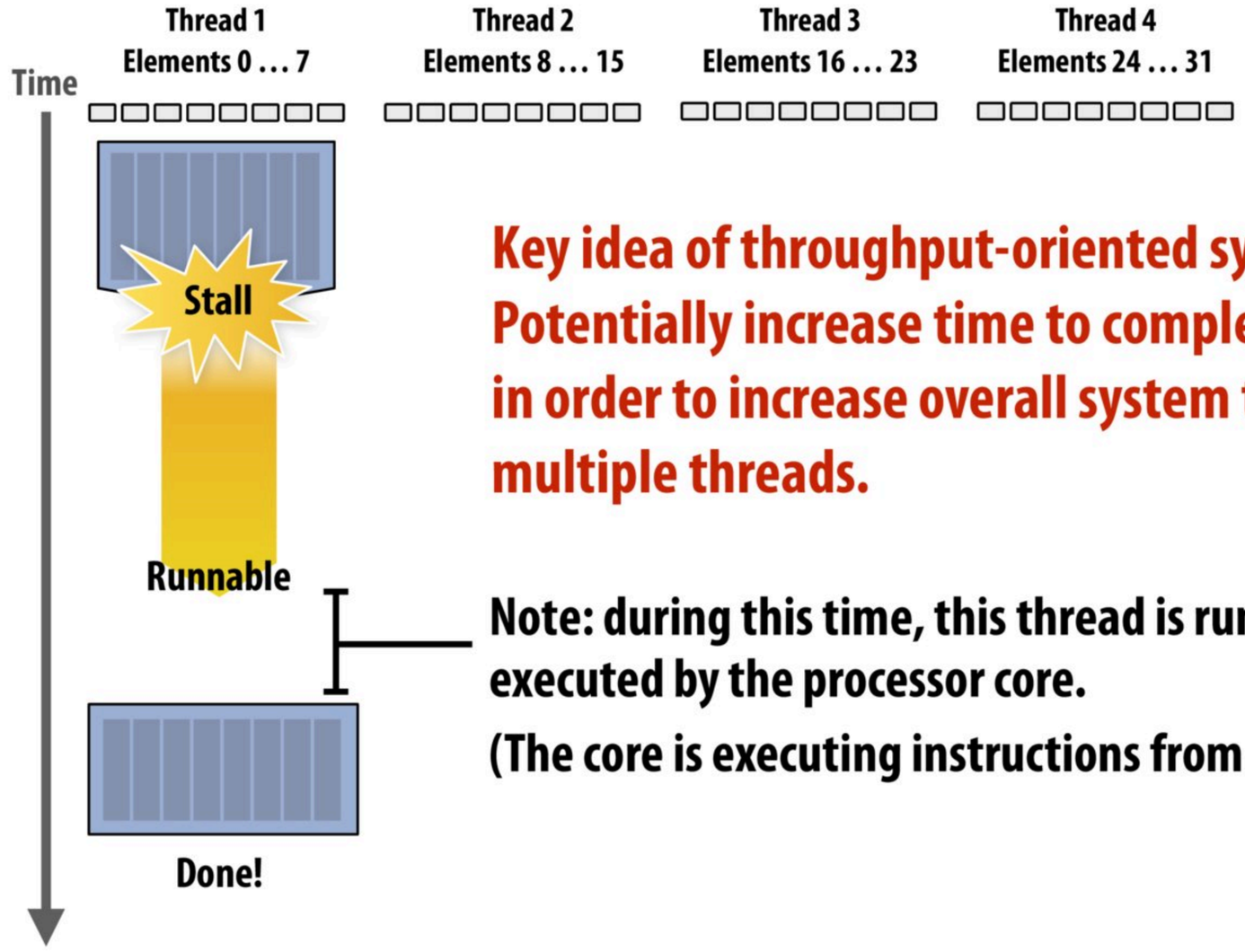



# Hiding stalls with multi-threading



# Throughput computing: a trade-off



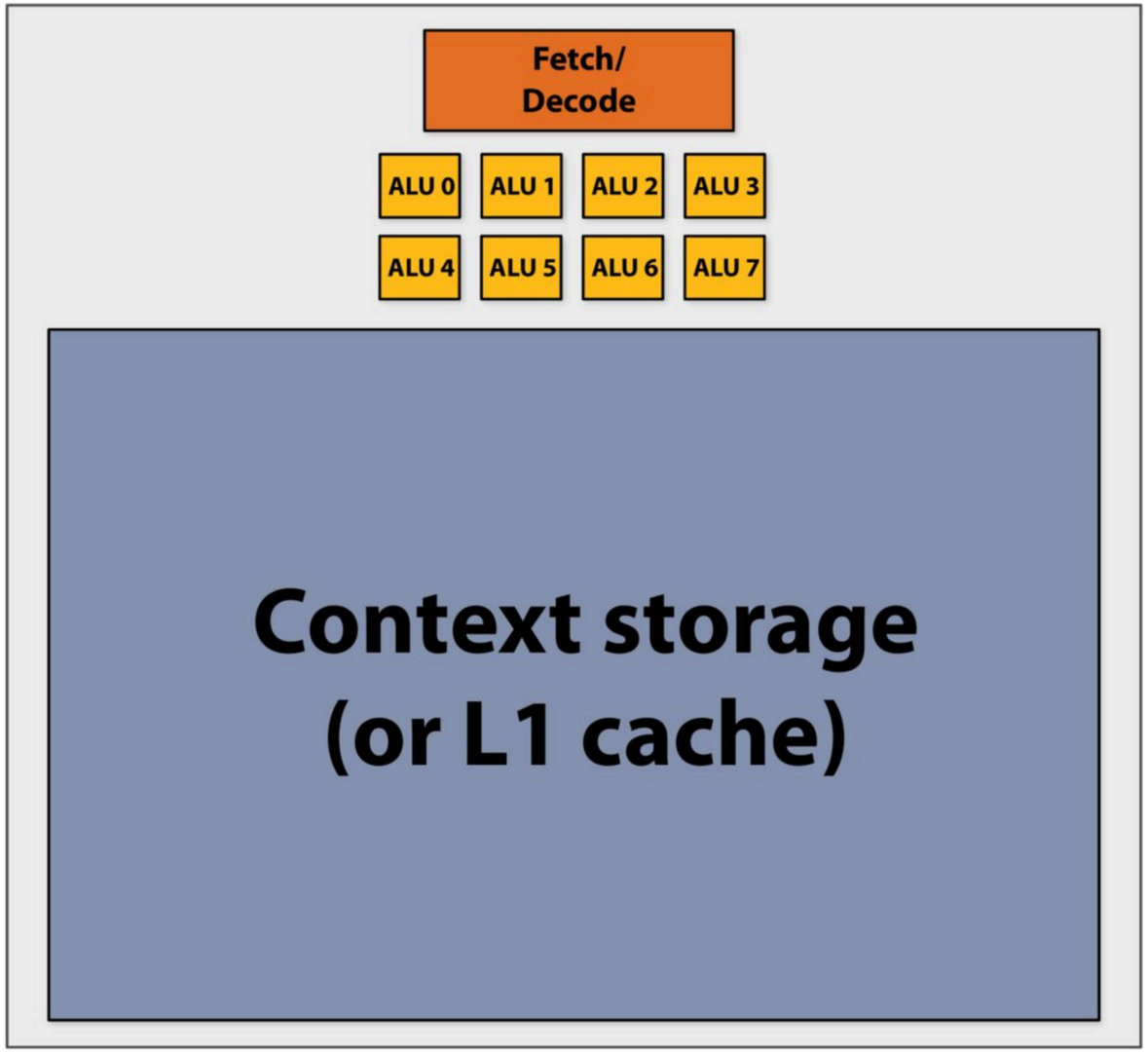

**Key idea of throughput-oriented systems:**
**Potentially increase time to complete work by any one thread, in order to increase overall system throughput when running multiple threads.**

**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.)**

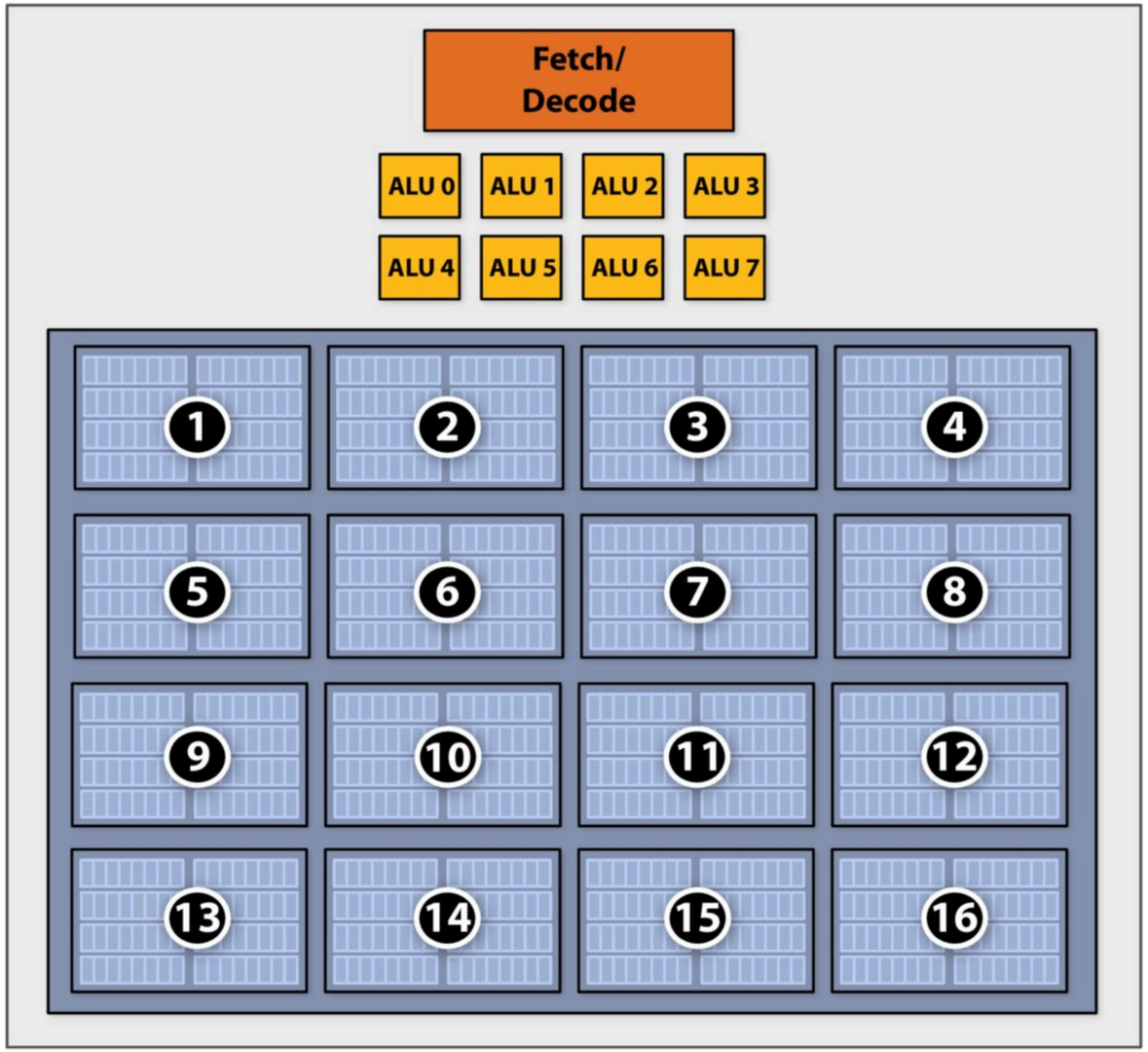
# No free lunch: storing execution contexts

## Consider on-chip storage of execution contexts as a finite resource



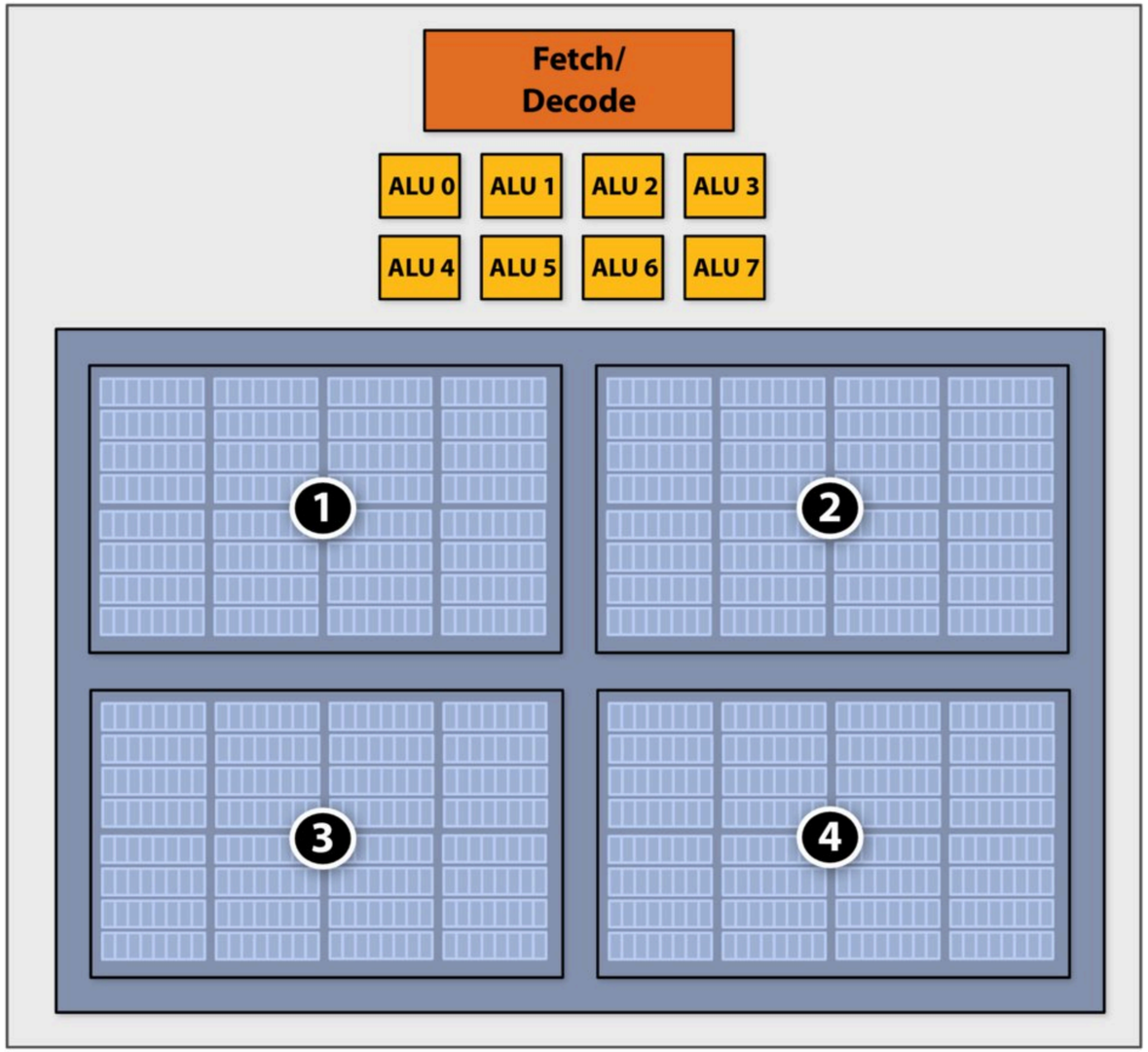
# Many small contexts (high latency hiding ability)

## 16 hardware threads: storage for small working set per thread

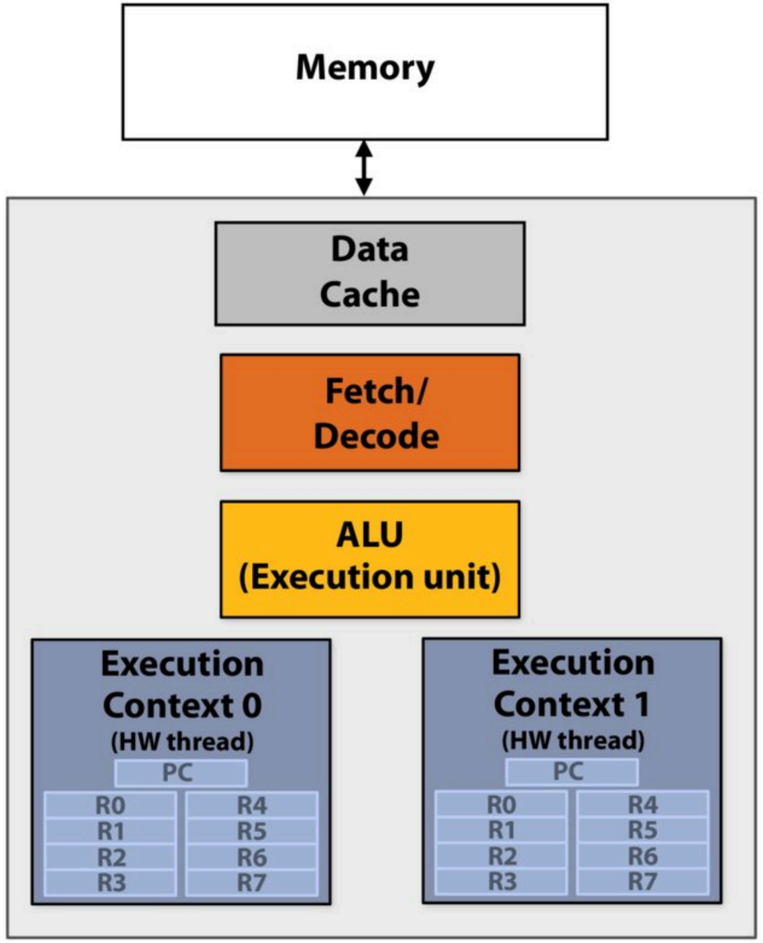


# Four large contexts (low latency hiding ability)

## 4 hardware threads: storage for large working set per thread

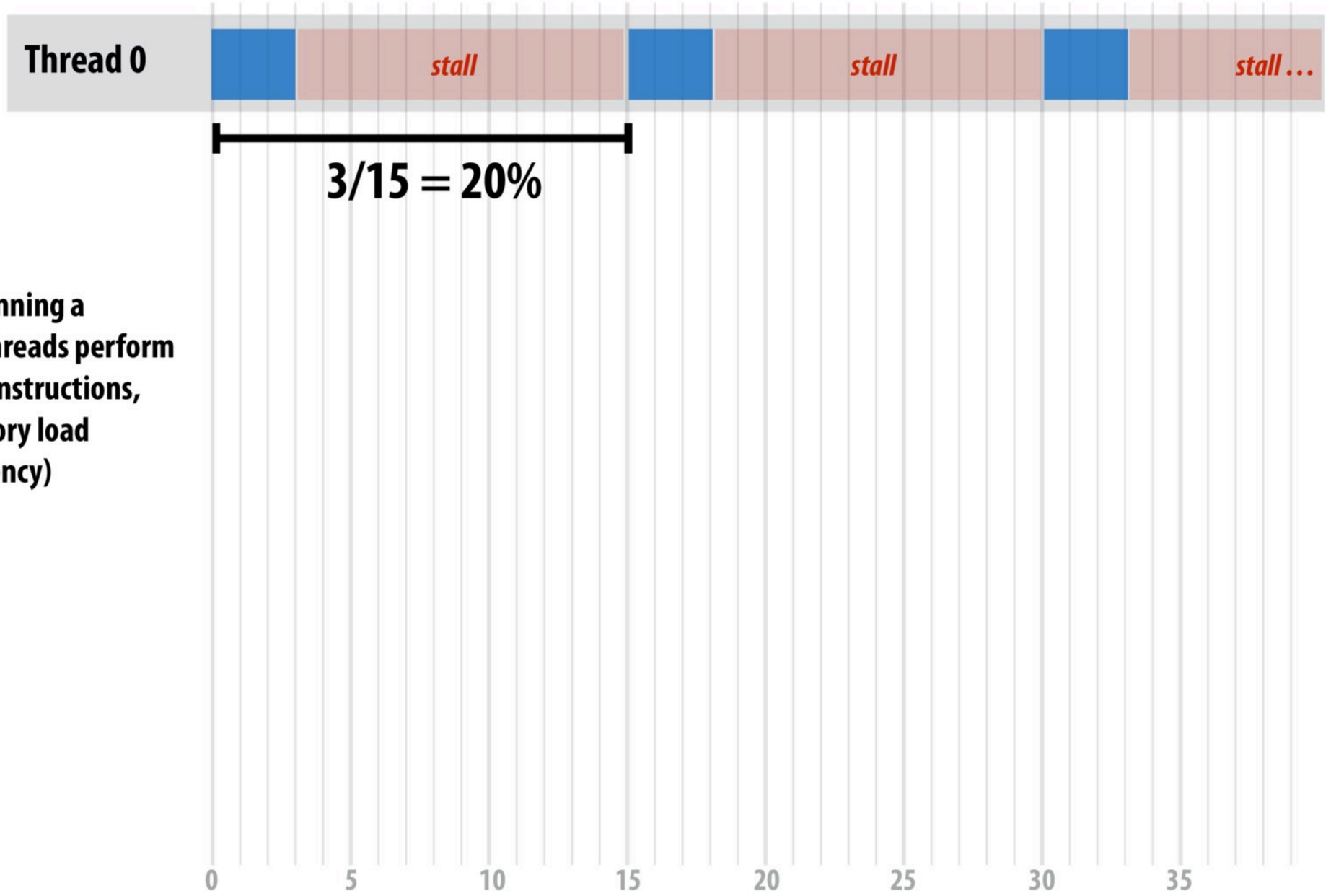


# Exercise: consider a simple two-threaded core



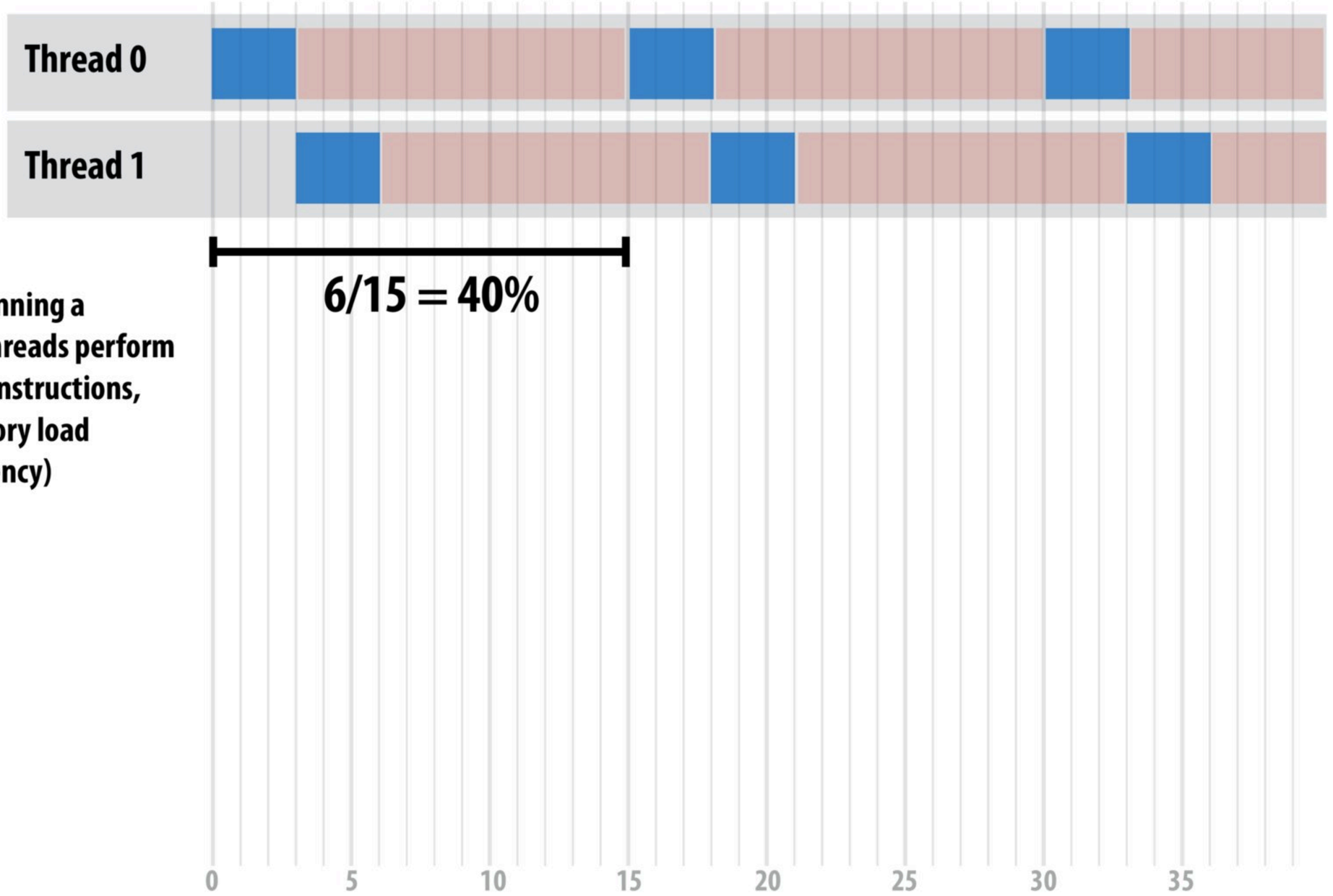

Single core processor, multi-threaded core (2 threads).
Can run one scalar instruction per clock from
one of the hardware threads

# What is the utilization of the core? (one thread)



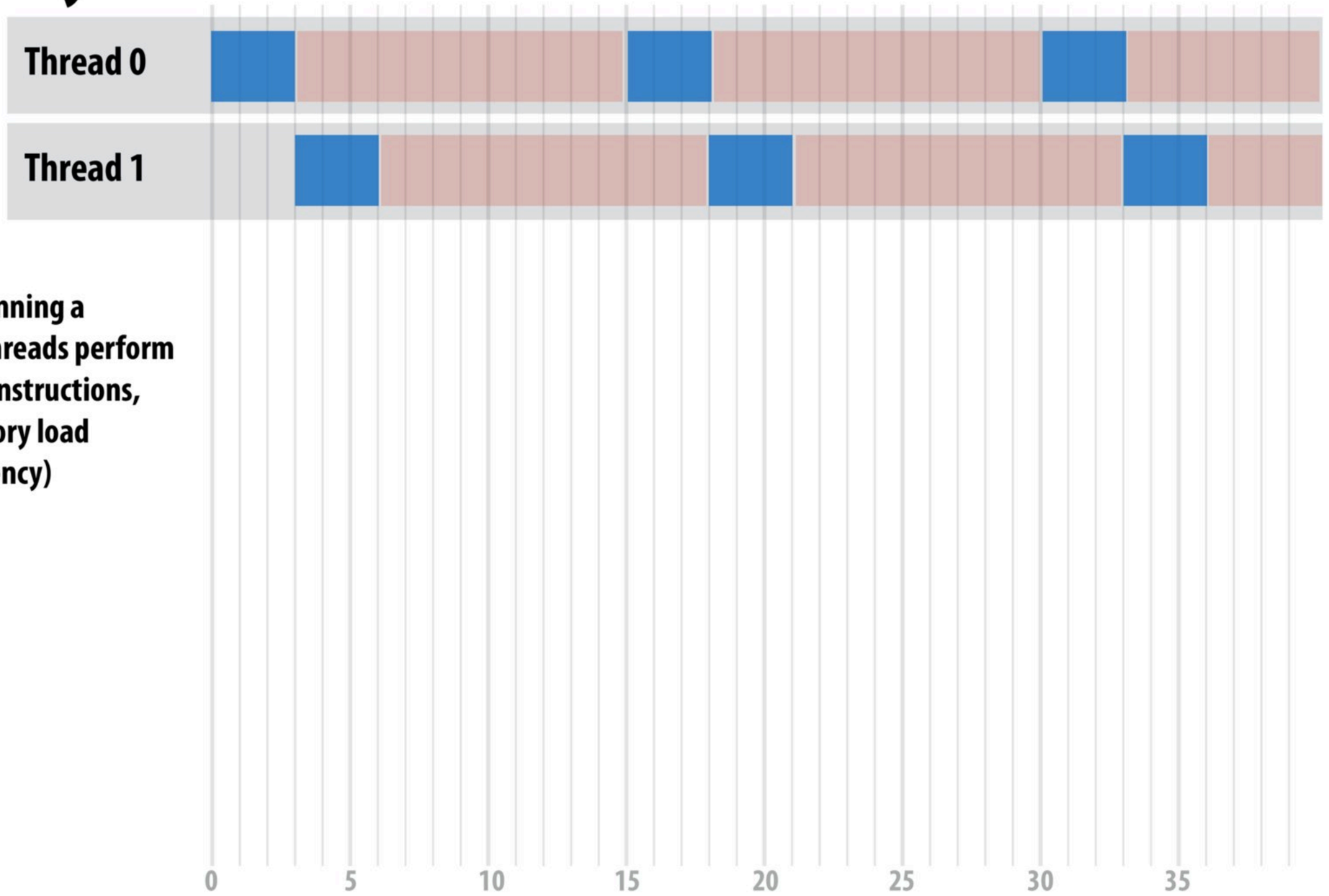

Assume we are running a program where threads perform three arithmetic instructions, followed by memory load (with 12 cycle latency)

# What is the utilization of the core? (two threads)



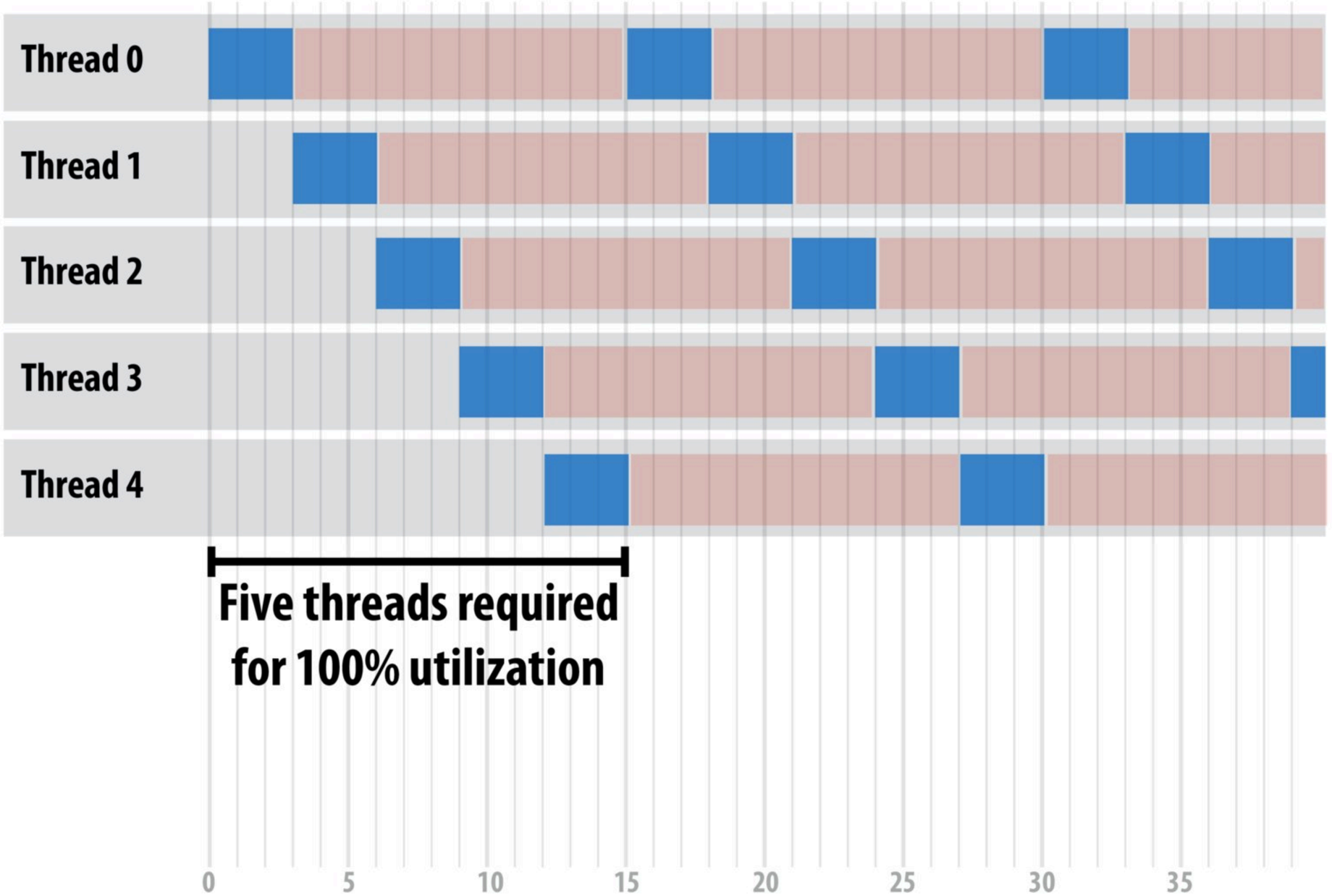

Assume we are running a program where threads perform three arithmetic instructions, followed by memory load (with 12 cycle latency)

# How many threads are needed to achieve 100% utilization?

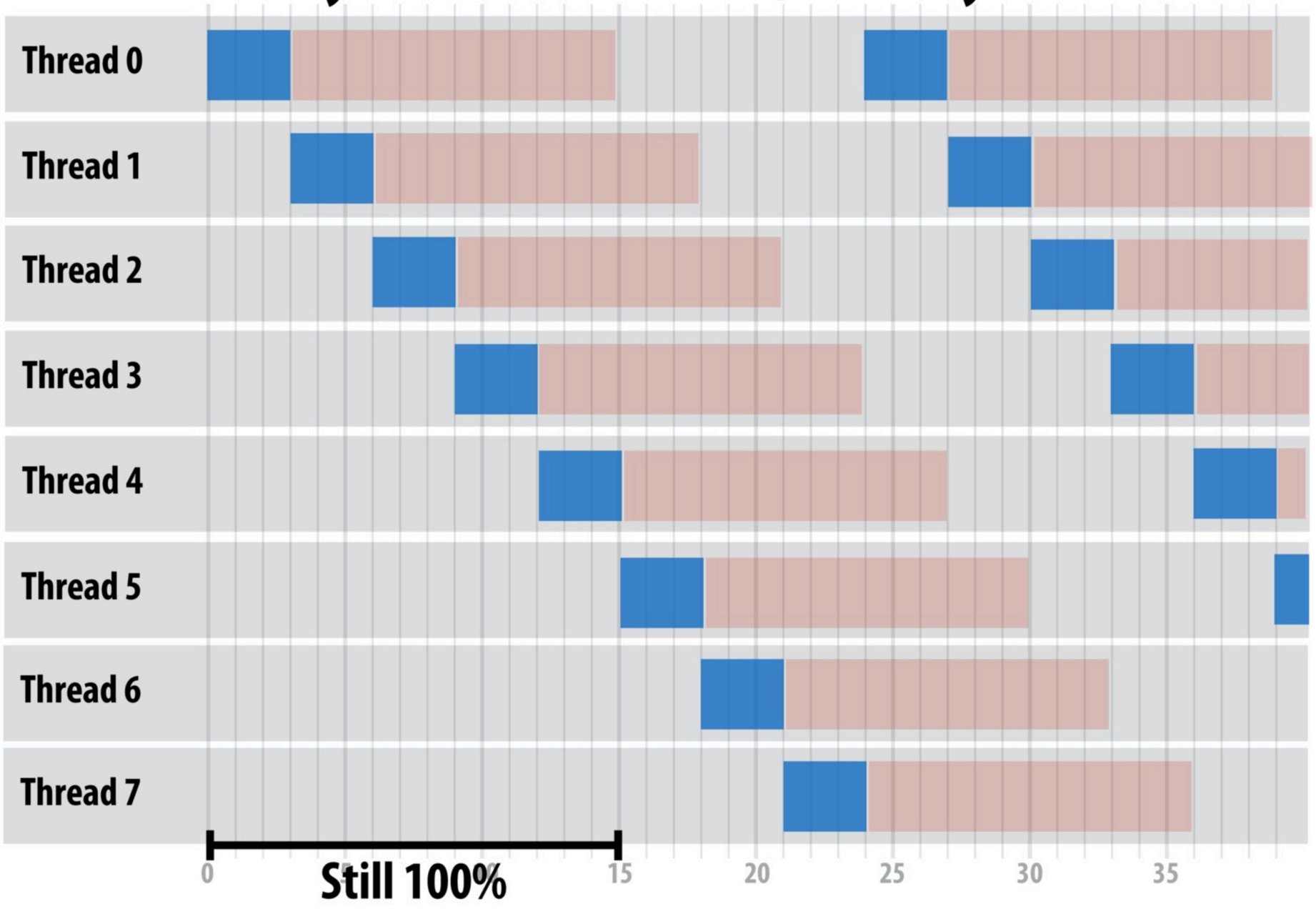



**Assume we are running a program where threads perform three arithmetic instructions, followed by memory load (with 12 cycle latency)**

# Five threads needed to obtain 100% utilization

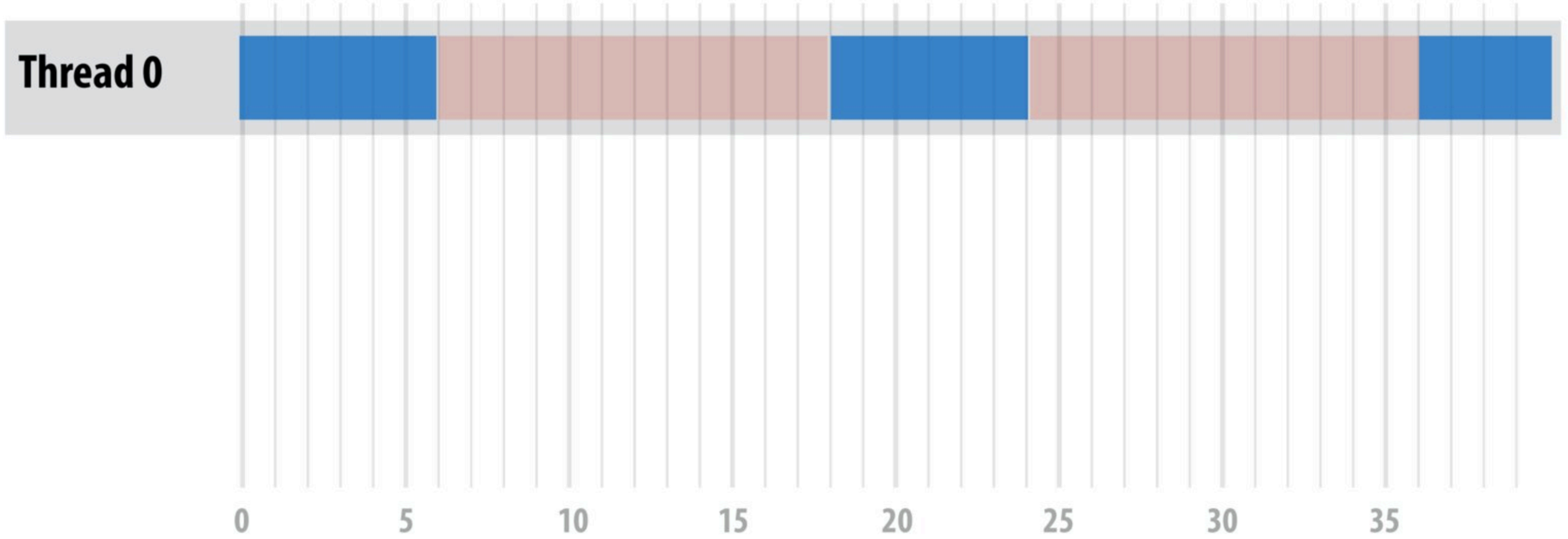


# Additional threads yield no benefit (already 100% utilization)



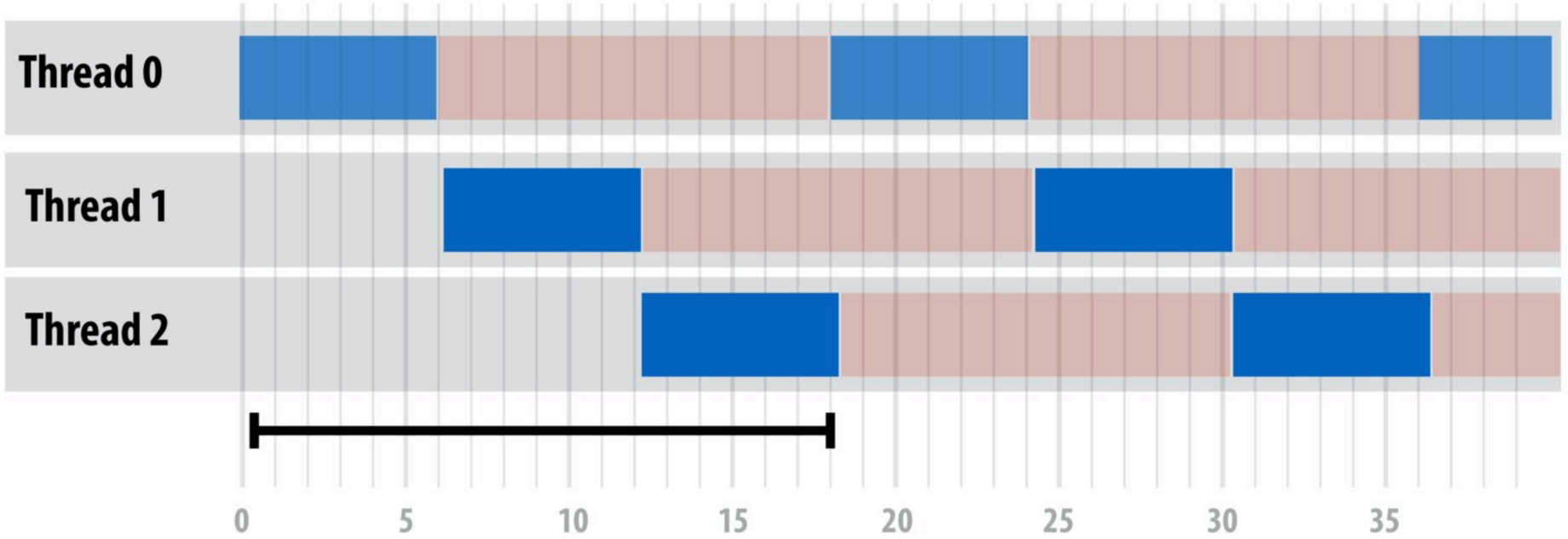
# How many threads are needed to achieve 100% utilization?

**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

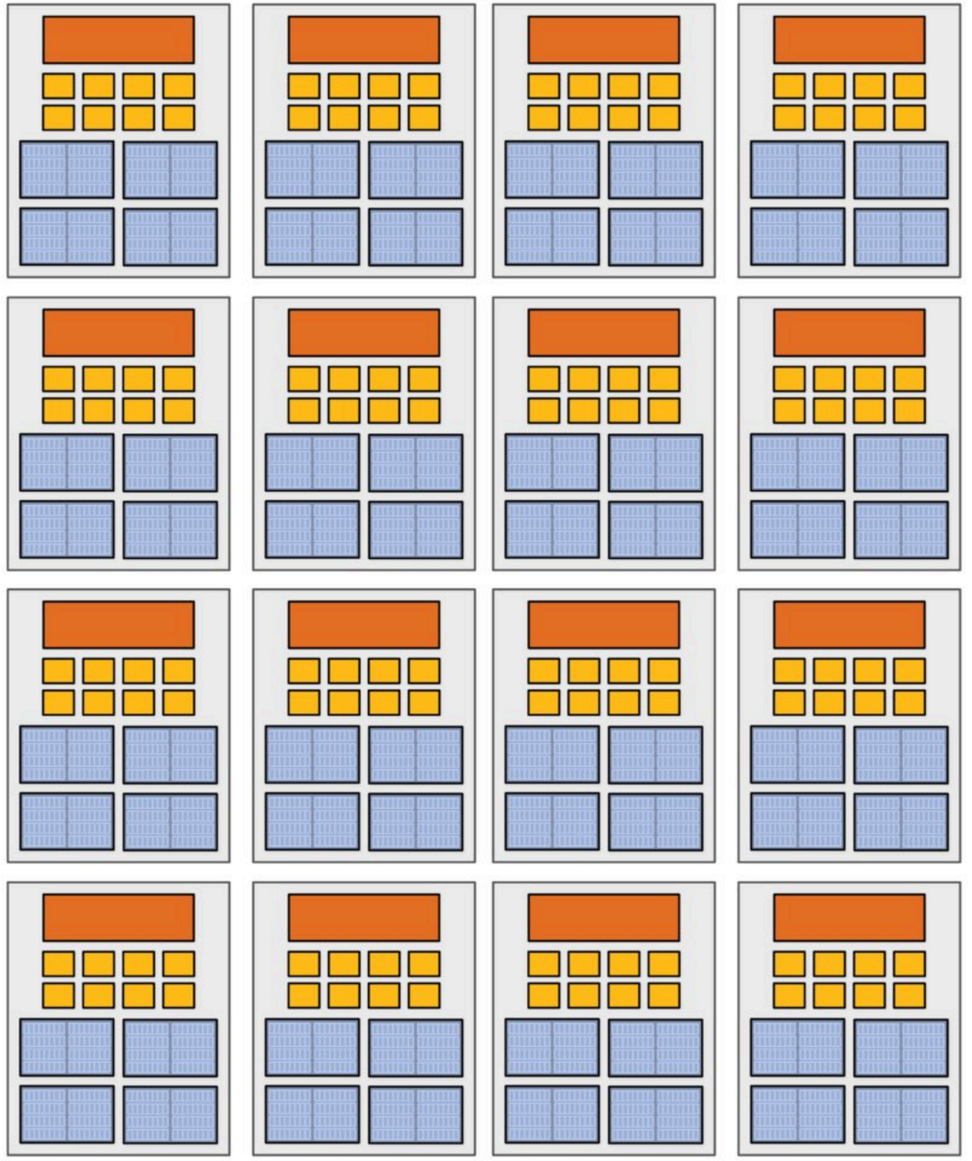
(128 total)

4 threads per core

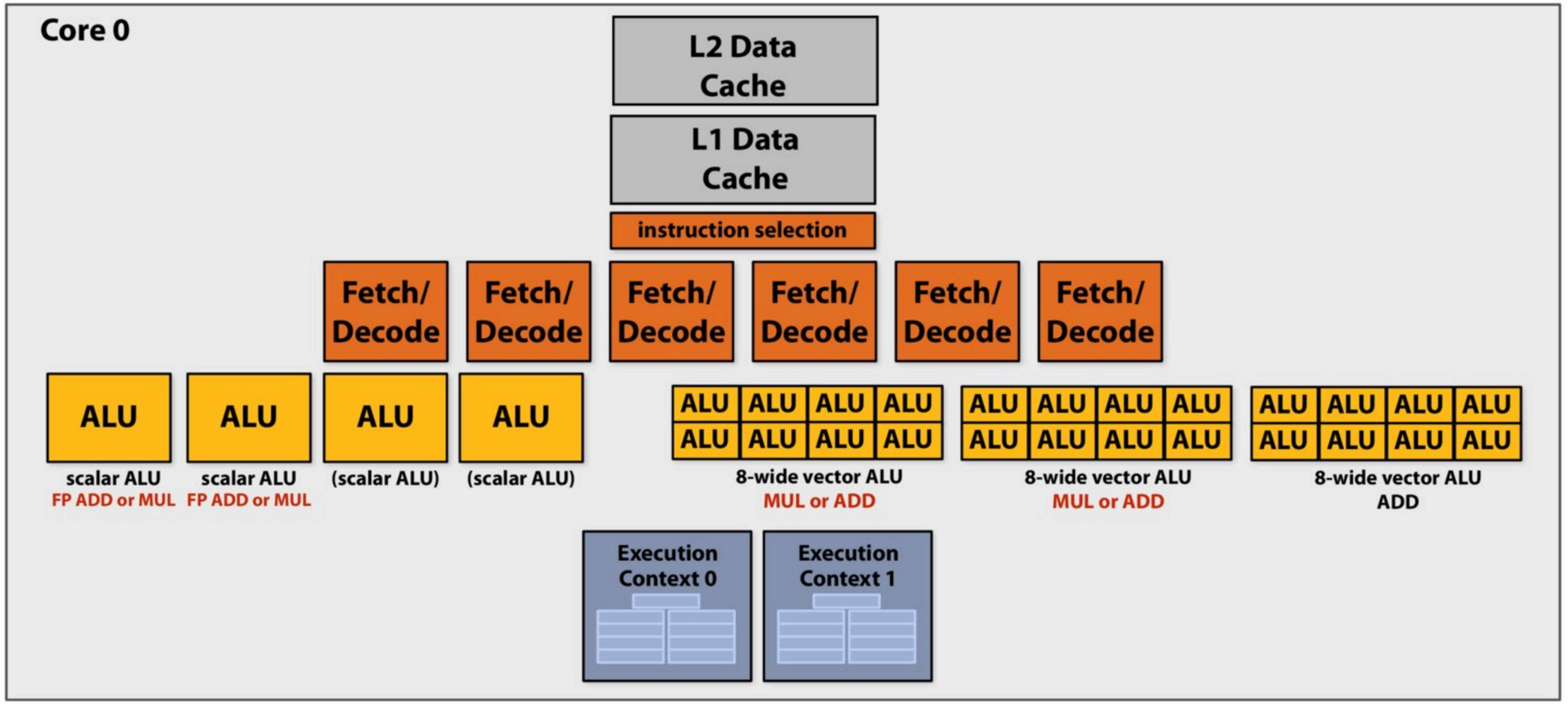
16 simultaneous instruction streams

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

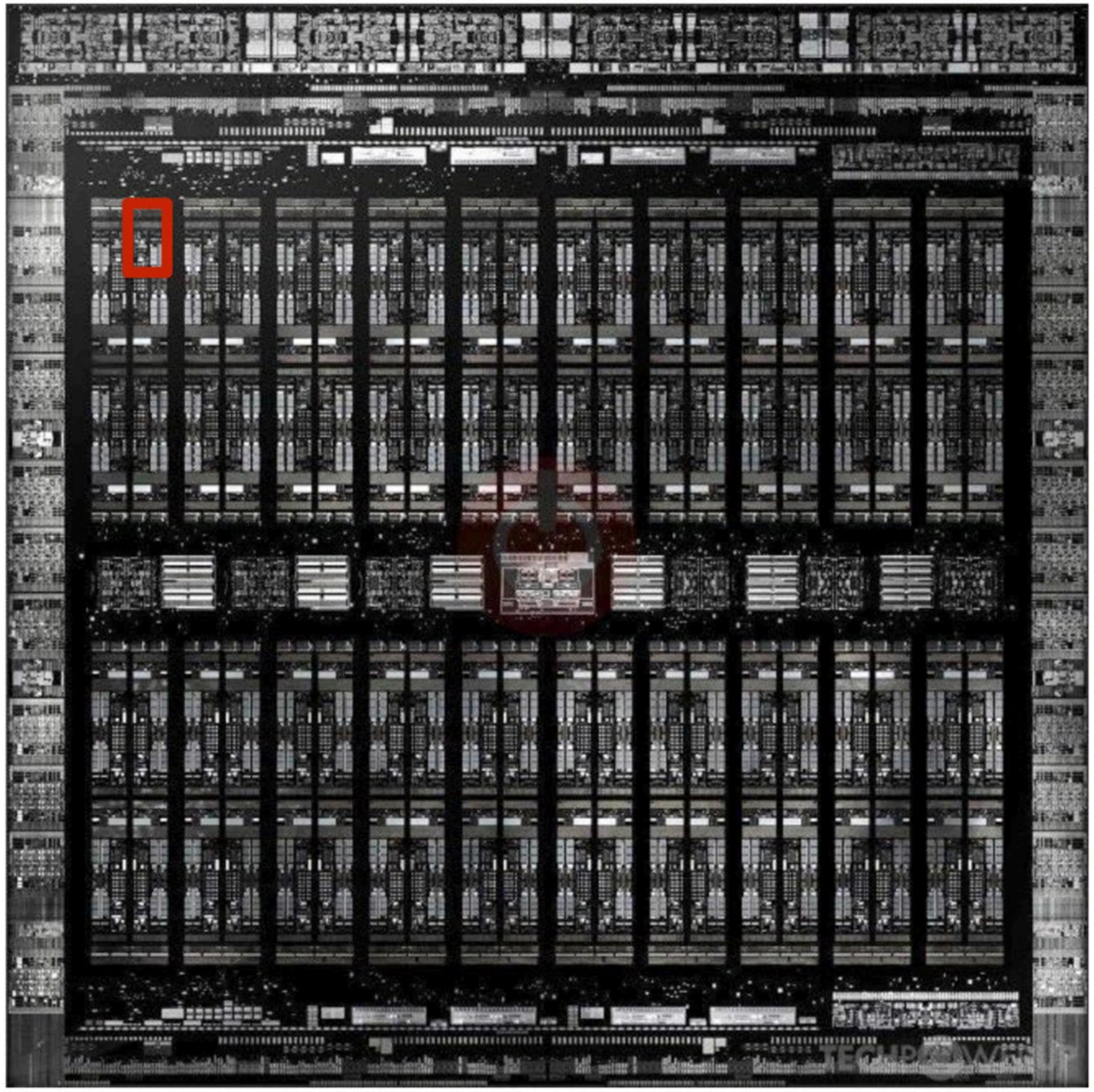



**Two-way multi-threaded cores (2 threads).**
**Each core can run up to four independent scalar instructions and up to three 8-wide vector instructions (up to 2 vector mul or 3 vector add)**

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

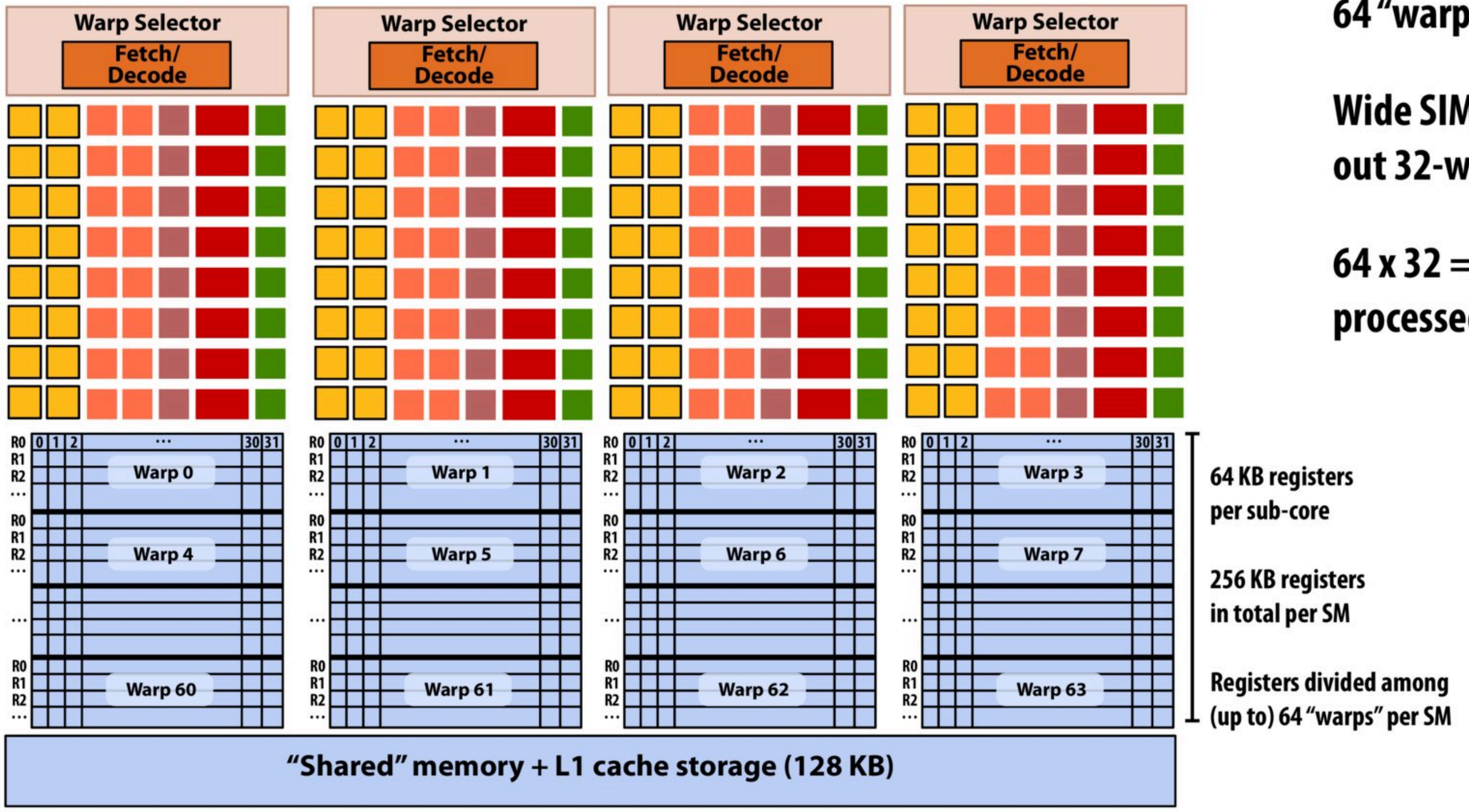
# NVIDIA V100

- SM = "Streaming Multi-processor"



# GPUs: extreme throughput-oriented processors

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




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

64 KB registers per sub-core

256 KB registers in total per SM

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

= 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

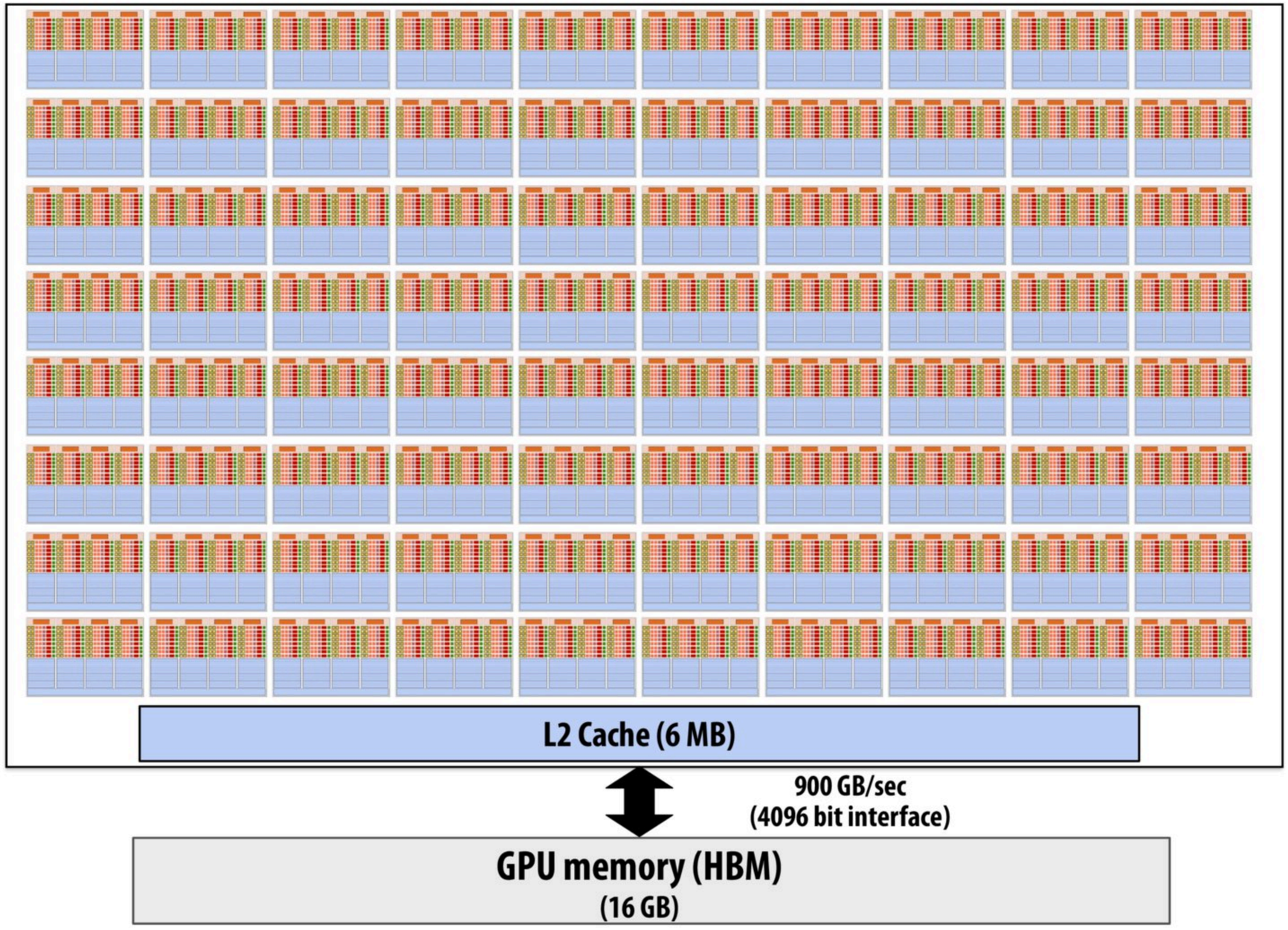
* one 32-wide SIMD operation every 2 clocks

** 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!**



# The story so far…

**To utilize modern parallel processors efficiently, an application must:**

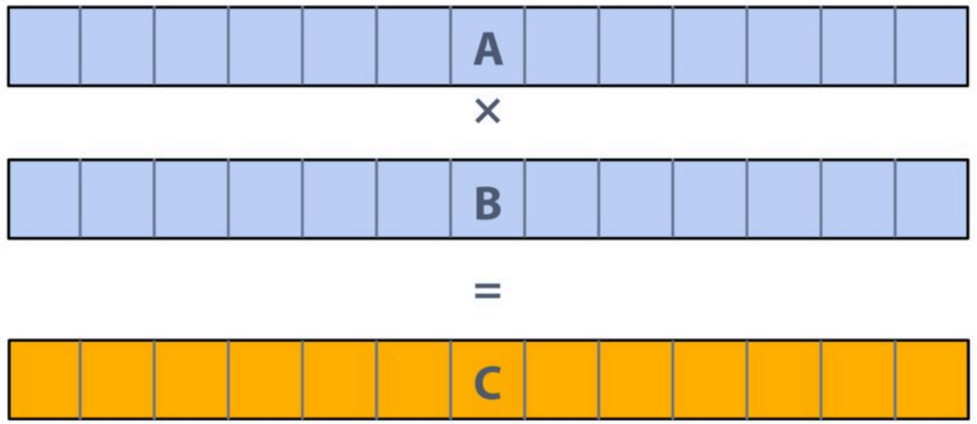
1. **Have sufficient parallel work to utilize all available execution units (across many cores and many execution units per core)**
2. **Groups of parallel work items must require the same sequences of instructions (to utilize SIMD execution)**
3. **Expose more parallel work than processor ALUs to enable 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

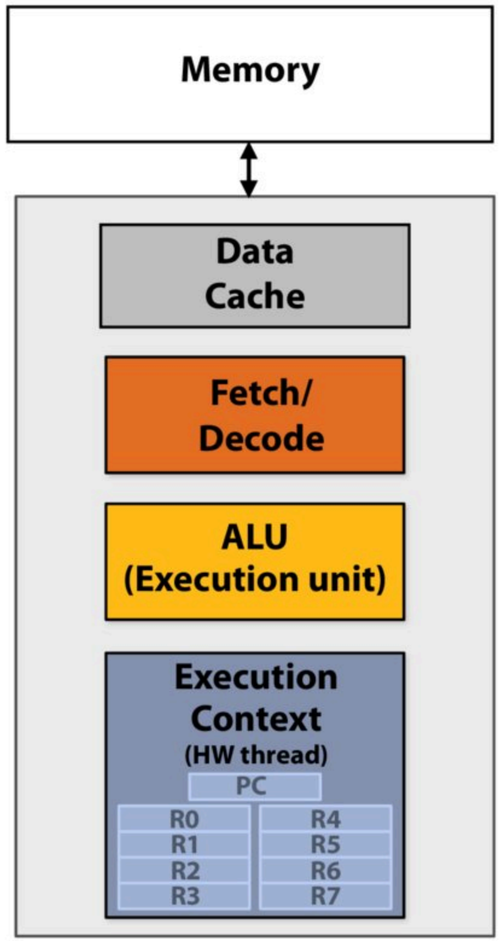
## Compiled instruction stream (scalar instructions)

```
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**

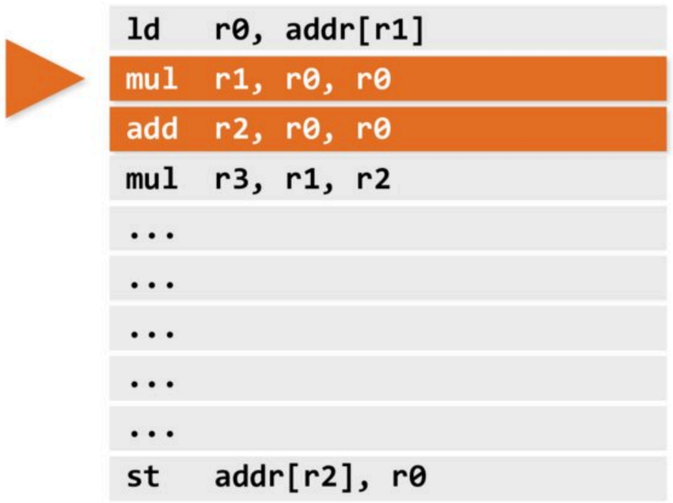
```
ld   r0, addr[r1]
mul  r1, r0, r0
add  r2, r0, r0
mul  r3, r1, r2
...
...
...
...
...
st   addr[r2], r0
```

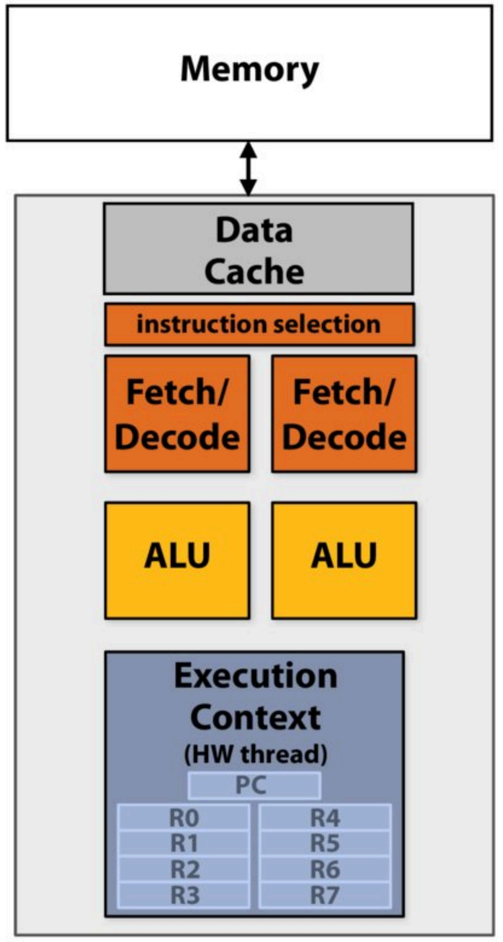



**Single core processor, single-threaded core.**
**Can run one scalar instruction per clock**

# Superscalar core

**Instruction stream**



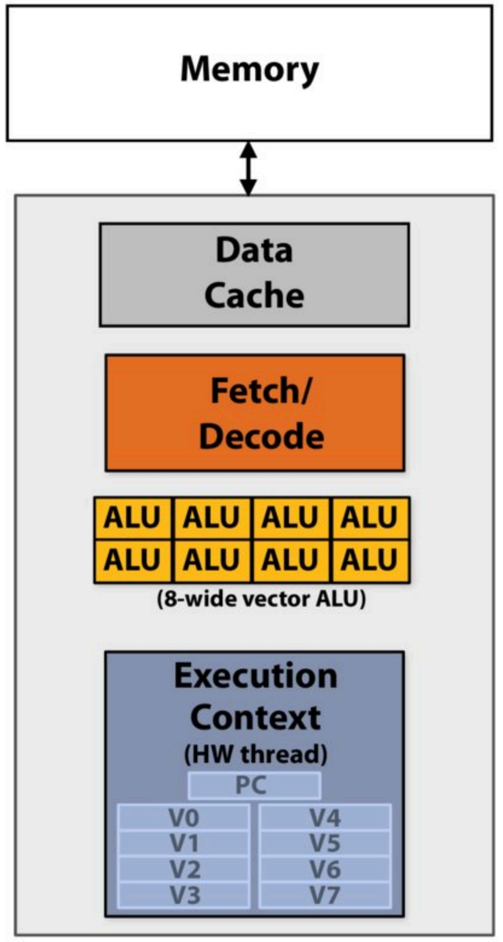




**Single core processor, single-threaded core.**
**Two-way superscalar core:**
**can run up to two independent scalar instructions per clock from one instruction stream (one hardware thread)**

# 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
```

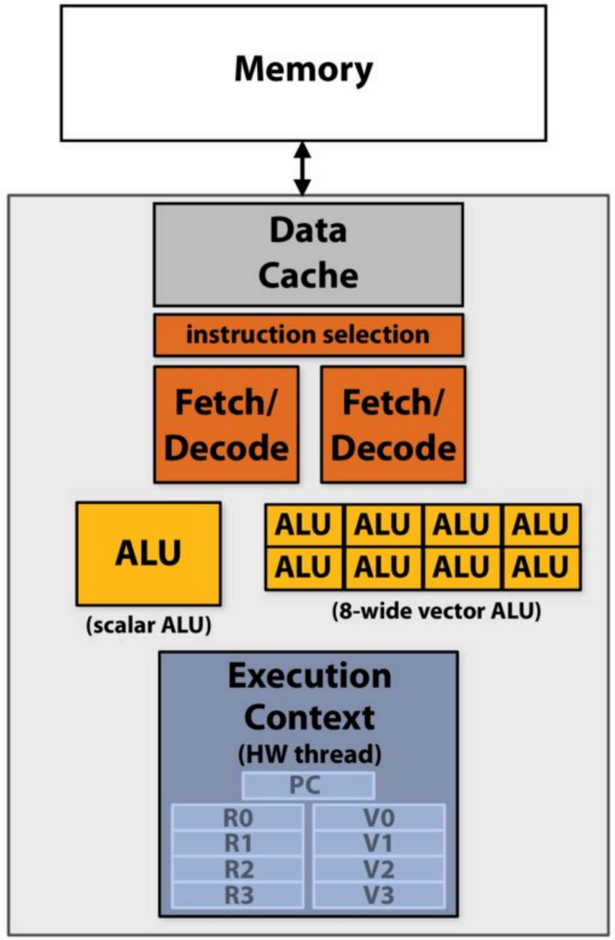



**Single core processor, single-threaded core. can run one 8-wide SIMD vector instruction from one instruction stream**

# 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
```




**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**

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

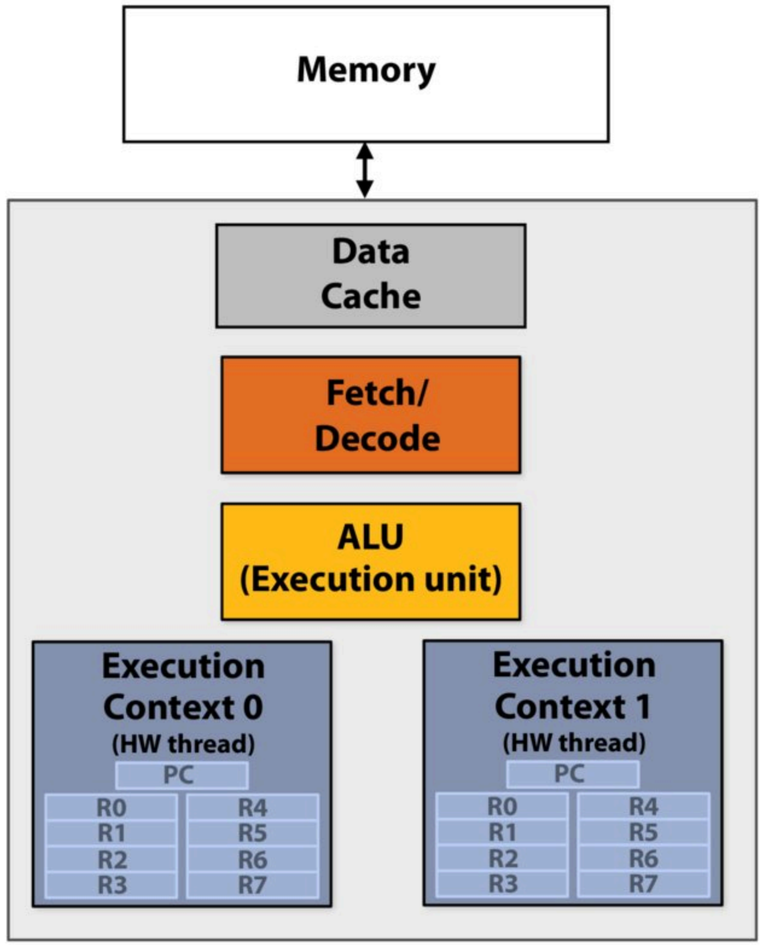
**Instruction stream 1**

PC

```
ld   r0, addr[r1]
sub  r1, r0, r0
add  r2, r1, r0
mul  r5, r1, 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)**

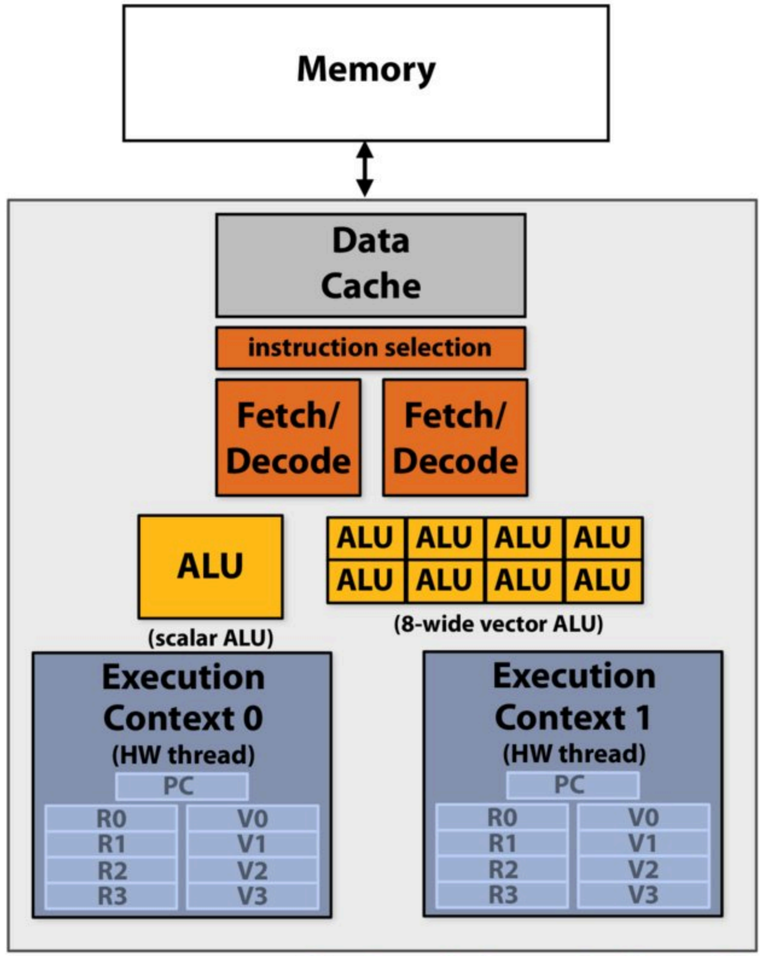
# Multi-threaded, superscalar core

**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.**

**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**

*** 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.**

# 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**

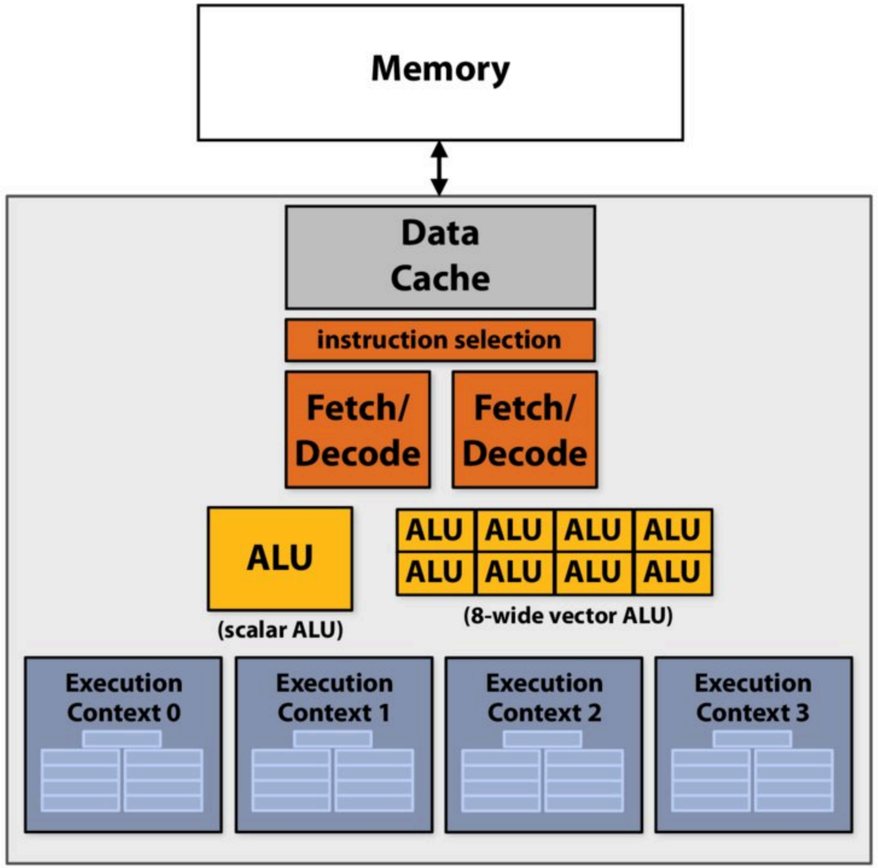
```
vector_ld   v0, addr[r1]
vector_mul  v2, v0, v0
mul         r3, r0, r0
sub         r1, r0, r3 ...
...
...
...
...
rect        addr[r2], v0
```

**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
```

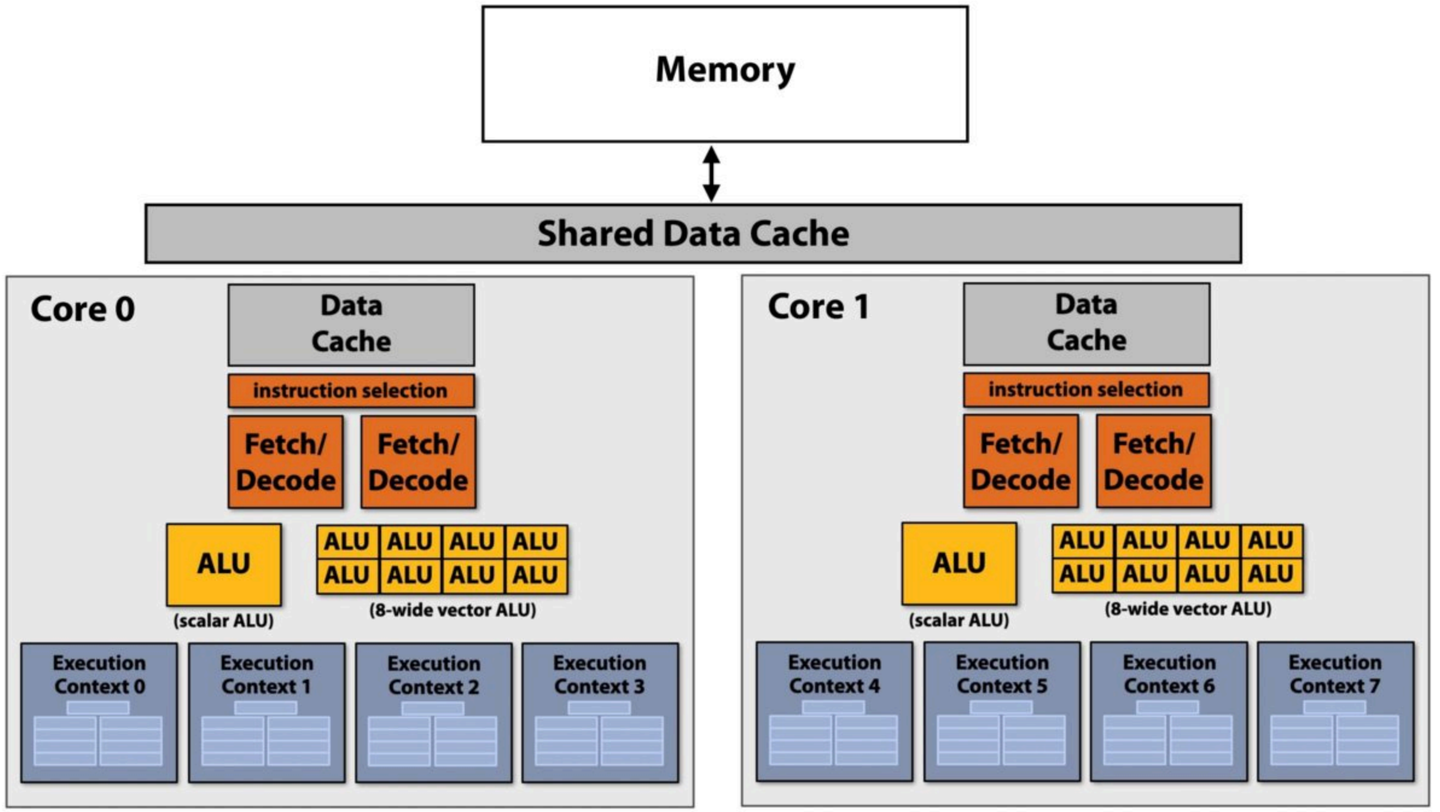
**Execution of hardware threads may or may not be interleaved in time**

**(instructions from different threads may be running simultaneously)**

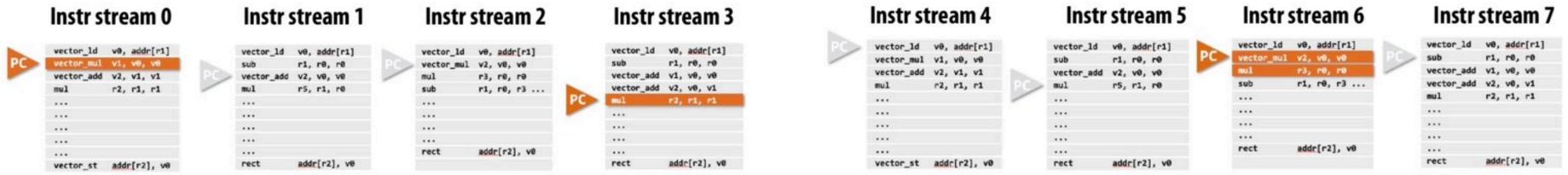



**Single core processor, multi-threaded core (4 threads).**
**Two-way superscalar core:**
**can run up to two independent instructions per clock from any of the threads, provided one is scalar and the other is vector**

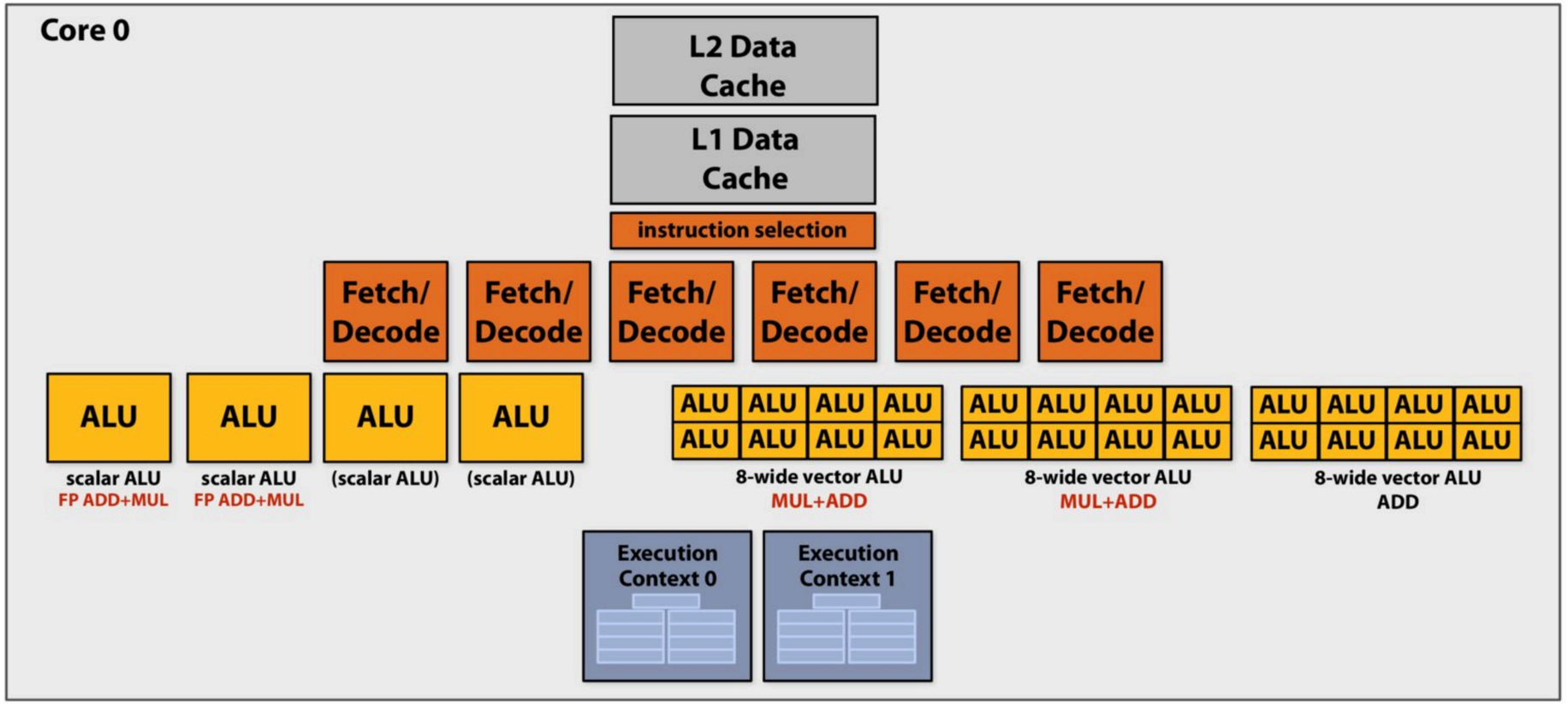
# Multi-core, with multi-threaded, superscalar cores




**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 any of its threads, provided one is scalar and the other is vector**



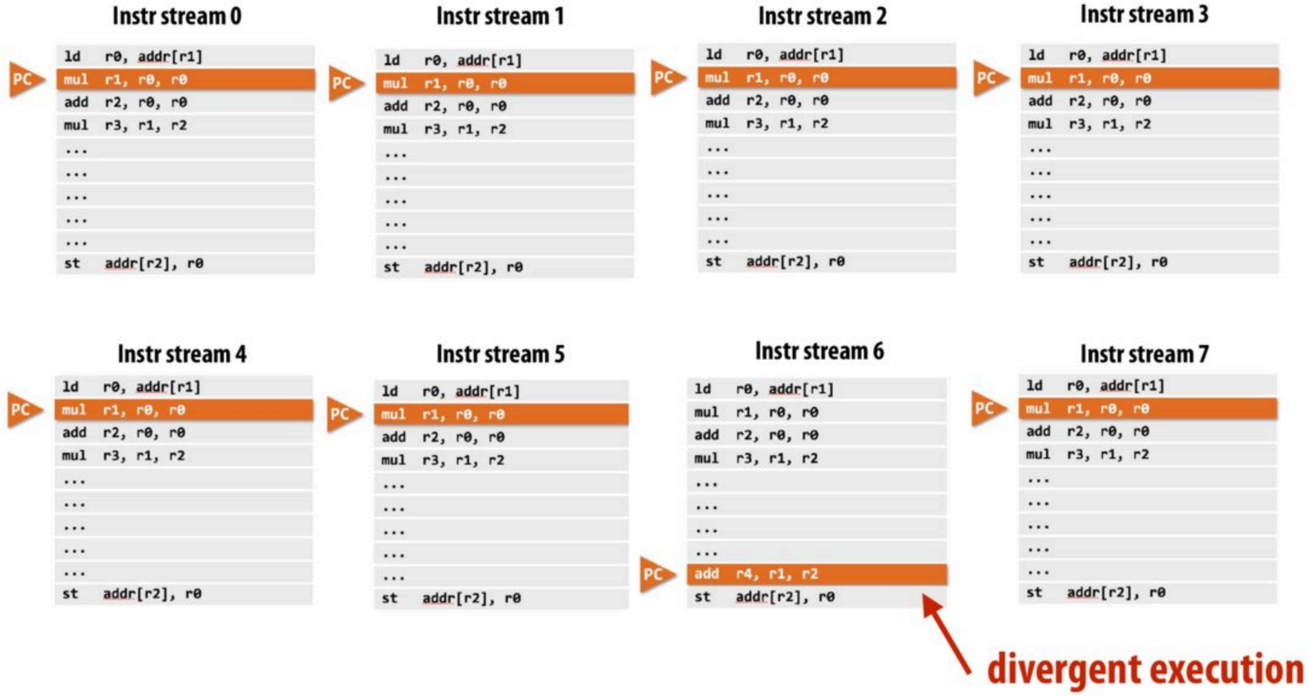
# Example: Intel Skylake/Kaby Lake core

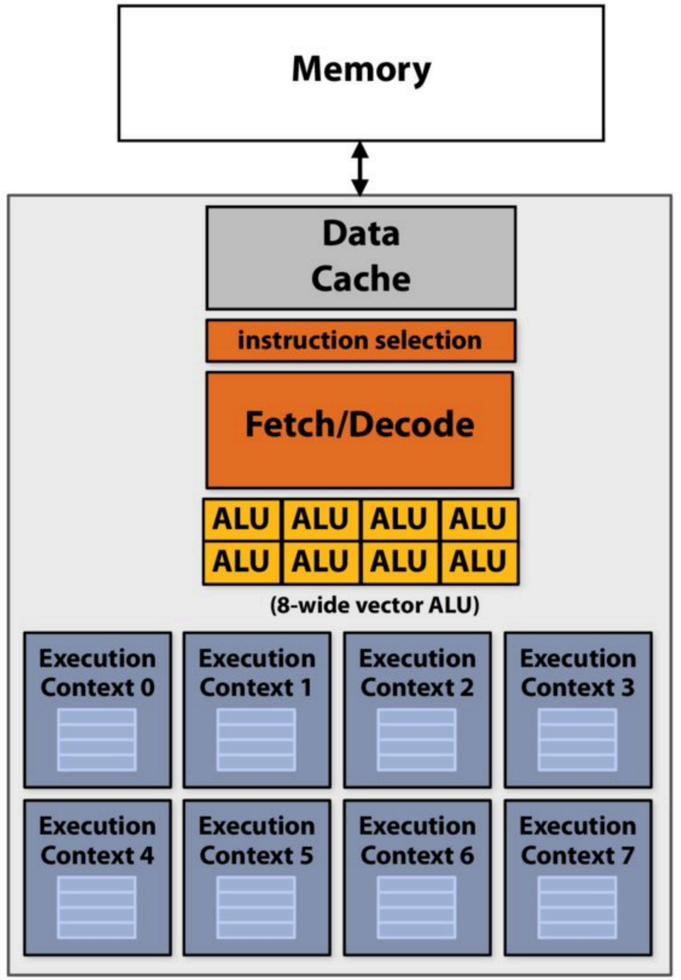



**Two-way multi-threaded cores (2 threads).**
**Each core can run up to four independent scalar instructions and up to three 8-wide vector instructions (up to 2 vector mul or 3 vector add)**

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

# GPU "SIMT" (single instruction multiple thread)




**Many modern GPUs execute hardware threads that run instruction streams with only scalar instructions.**

**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 two 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 five threads?