Lecture 1:

Course Introduction + Review of Throughput HW Architecture

Visual Computing Systems
Stanford CS348K, Spring 2021

Hello from the course staff

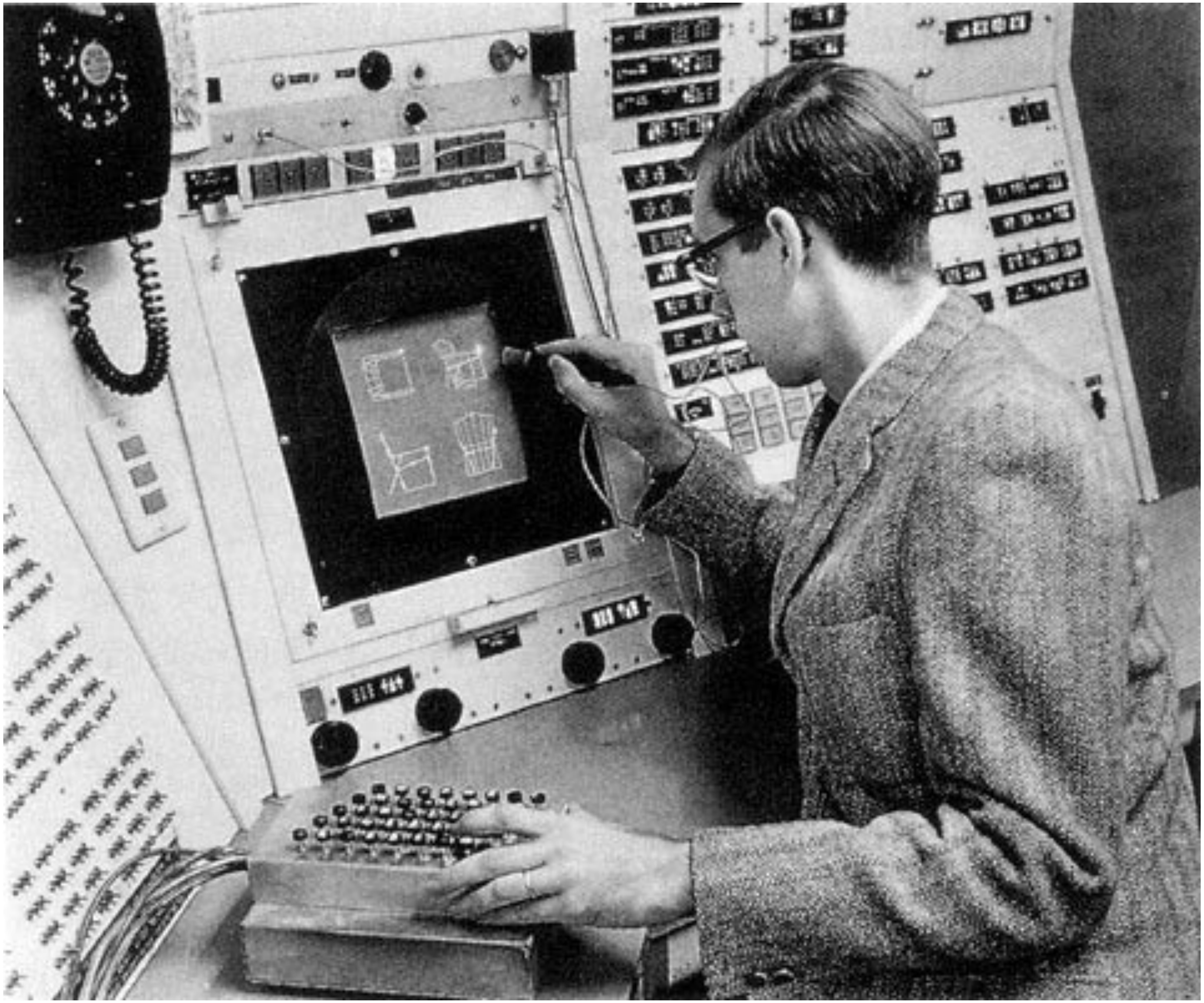
Your instructor (me)

Prof. Kayvon

Your CA

David Durst

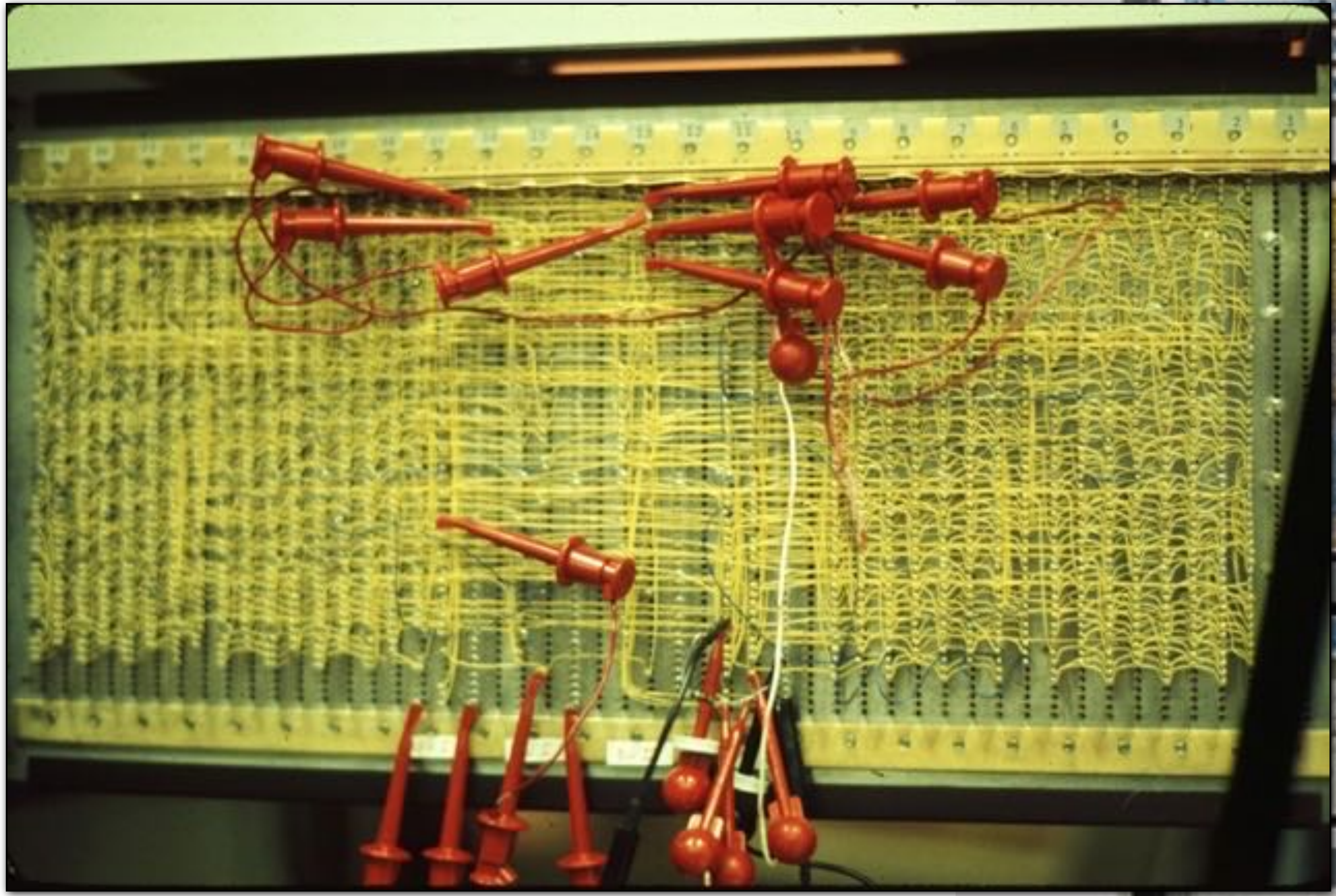
Visual computing applications have always demanded some of the world's most advanced parallel computing systems

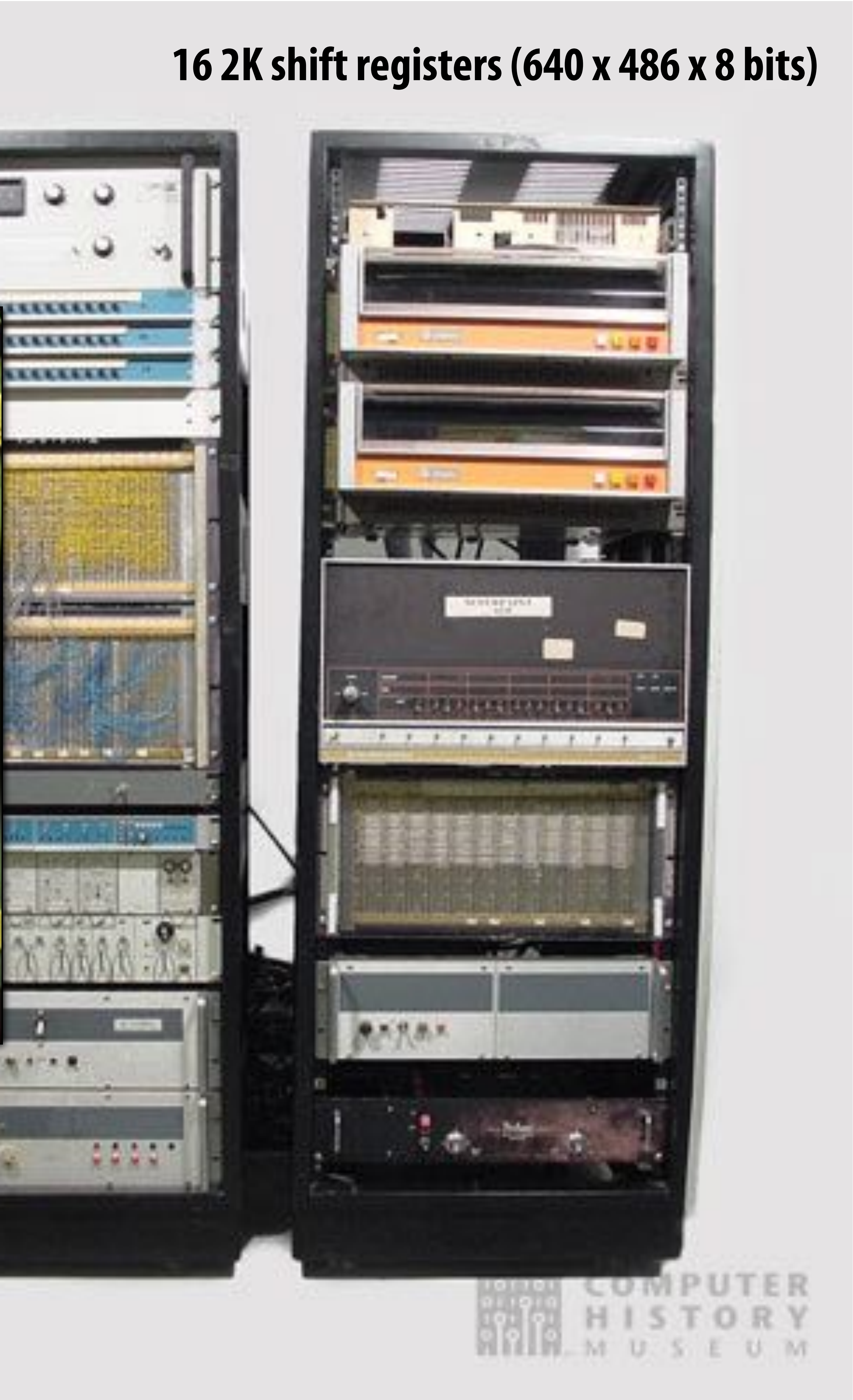


Ivan Sutherland's Sketchpad on MIT TX-2 (1962)

The frame buffer

Shoup's SuperPaint (PARC 1972-73)

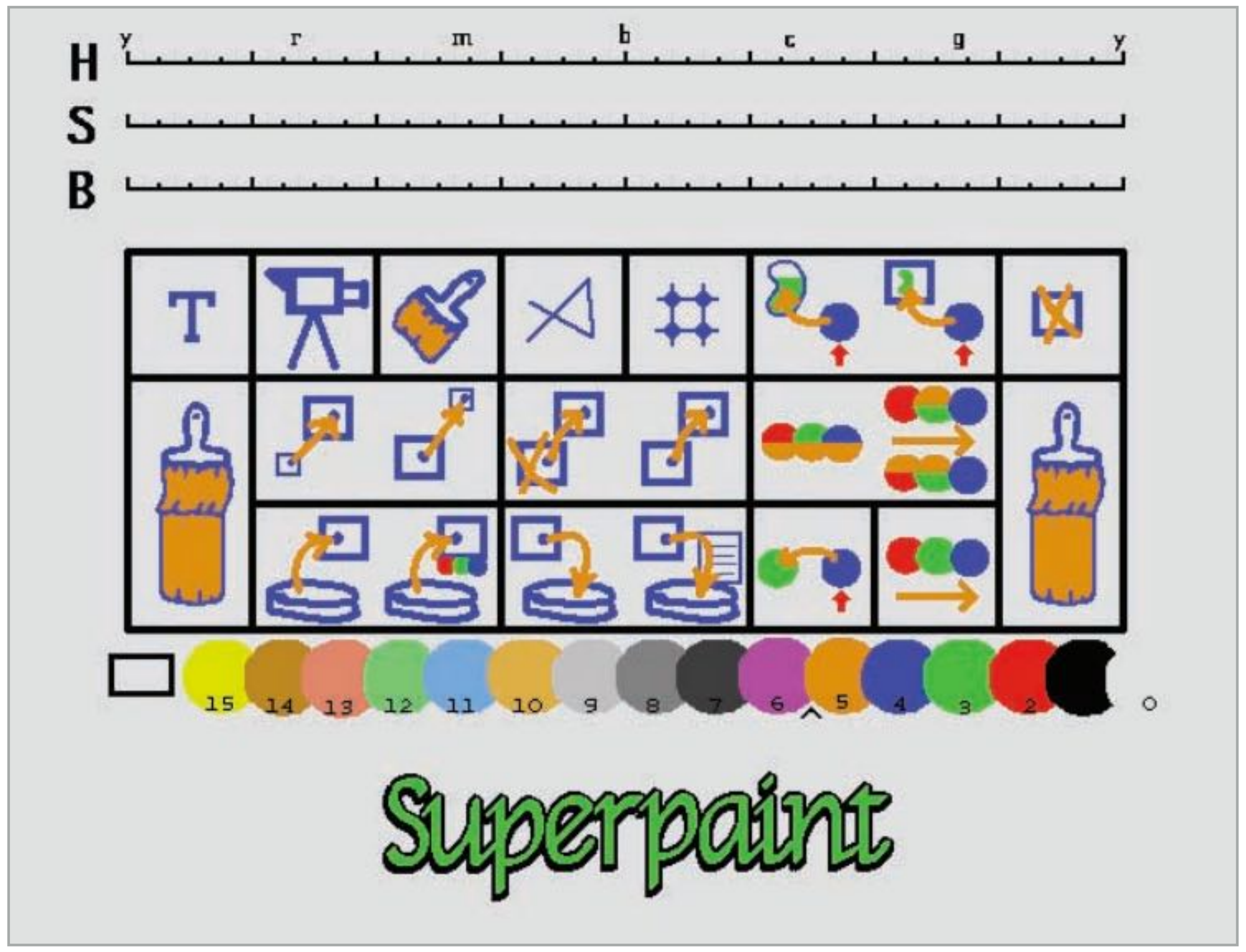




16 2K shift registers (640 x 486 x 8 bits)

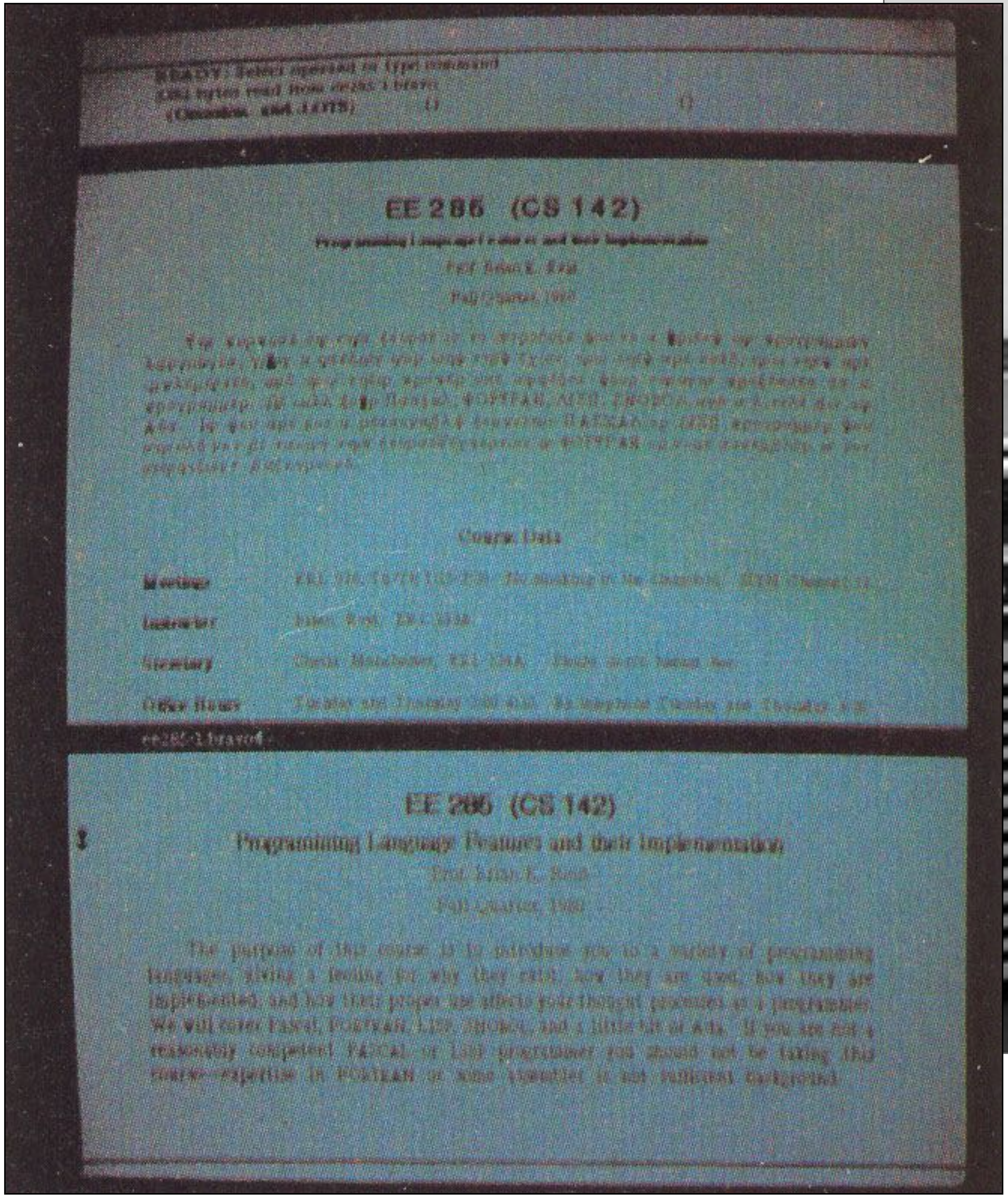
The frame buffer

Shoup's SuperPaint (PARC 1972-73)



16 2K shift registers (640 x 486 x 8 bits)

Xerox Alto (1973)



READY: Select operand or type command
[illegible] bytes read from ee286-1.bravo
([illegible])

EE 286 (CS 142)

[illegible]

[illegible]

[illegible]

Course Data

Meetings [illegible]

Instructor [illegible]

Secretary [illegible]

Office Hours [illegible]

ee286-1.bravo

EE 286 (CS 142)

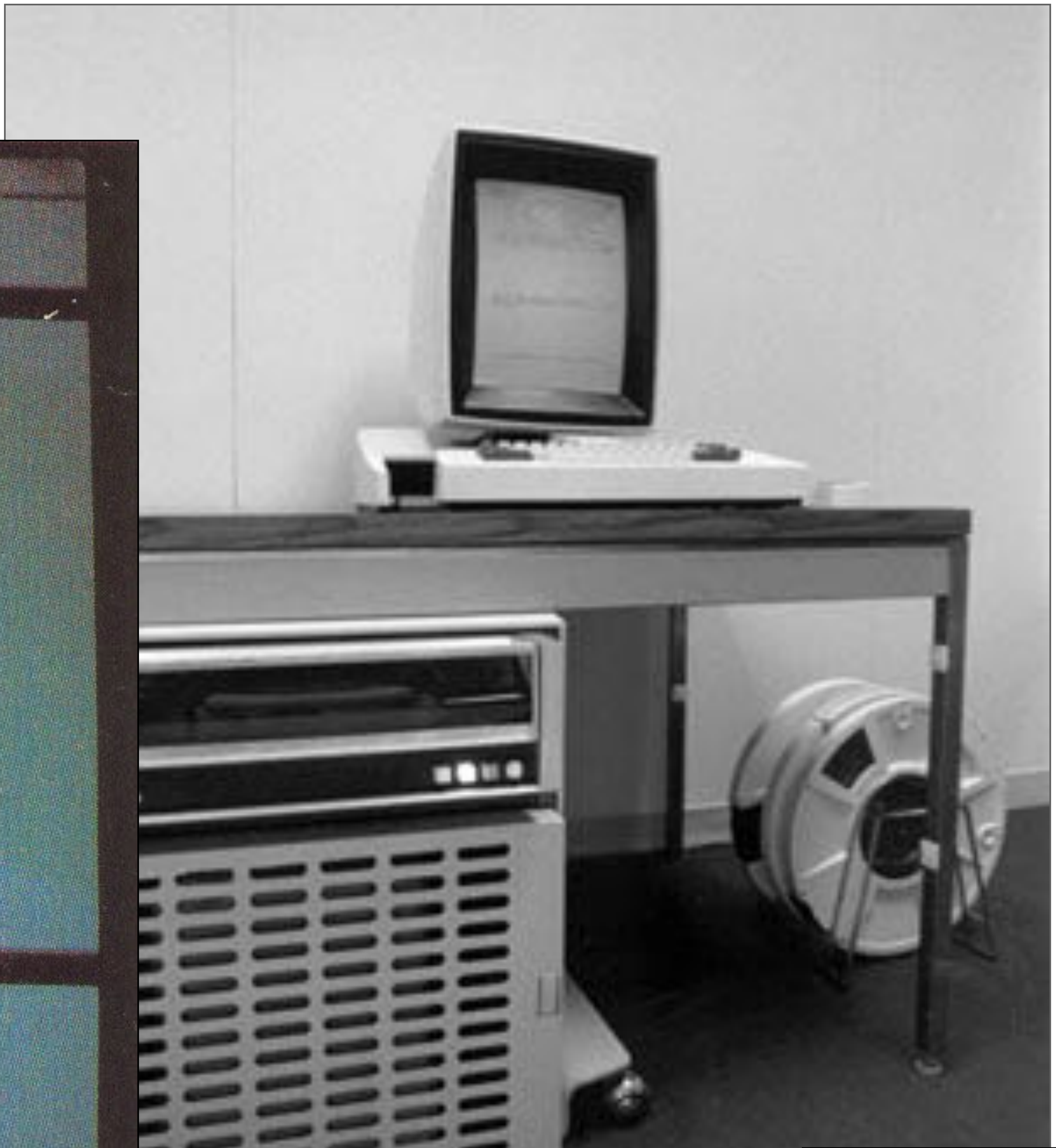
Programming Language Features and their Implementation

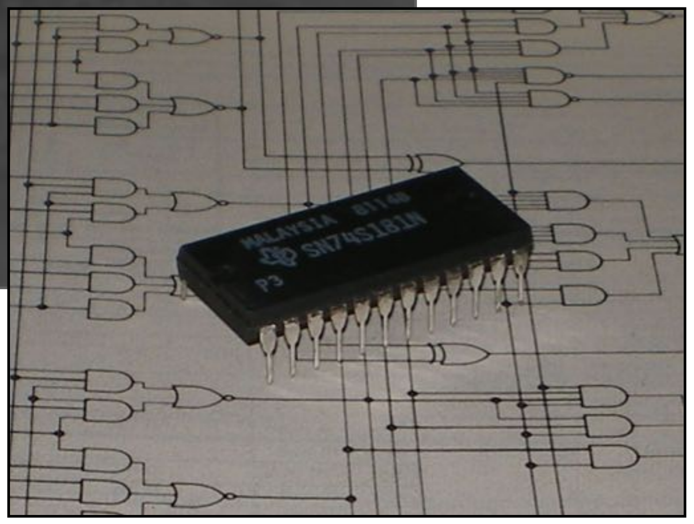
[illegible]

Fall Quarter, 1980

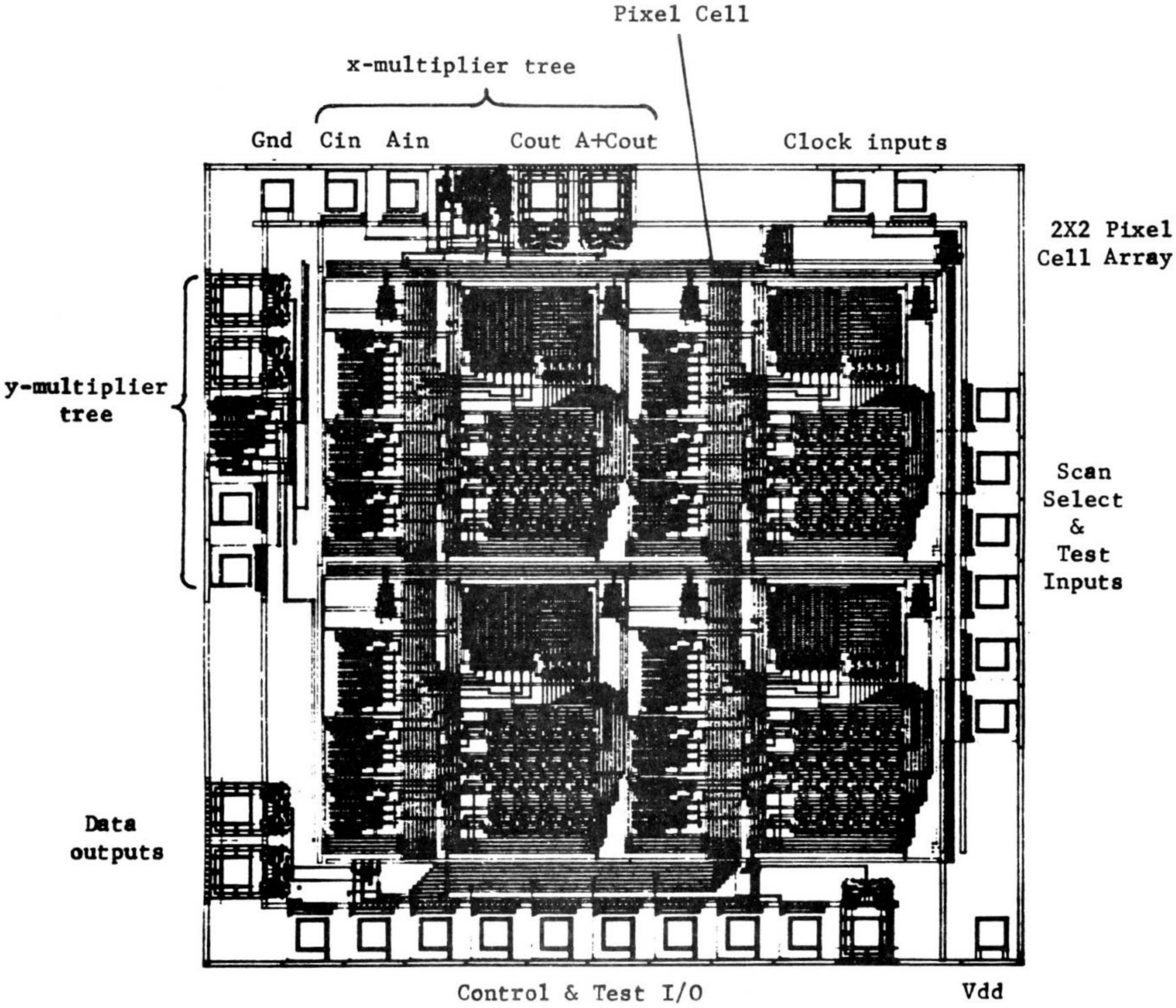
The purpose of this course is to introduce you to a variety of programming languages, giving a feeling for why they exist, how they are used, how they are implemented, and how their proper use affects your thought processes as a programmer. We will cover Pascal, FORTRAN, LISP, SNOBOL, and a little bit of ADA. If you are not a reasonably competent PASCAL or LISP programmer you should not be taking this course--expertise in FORTRAN or some assembler is not sufficient background.

Bravo (WYSIWYG)





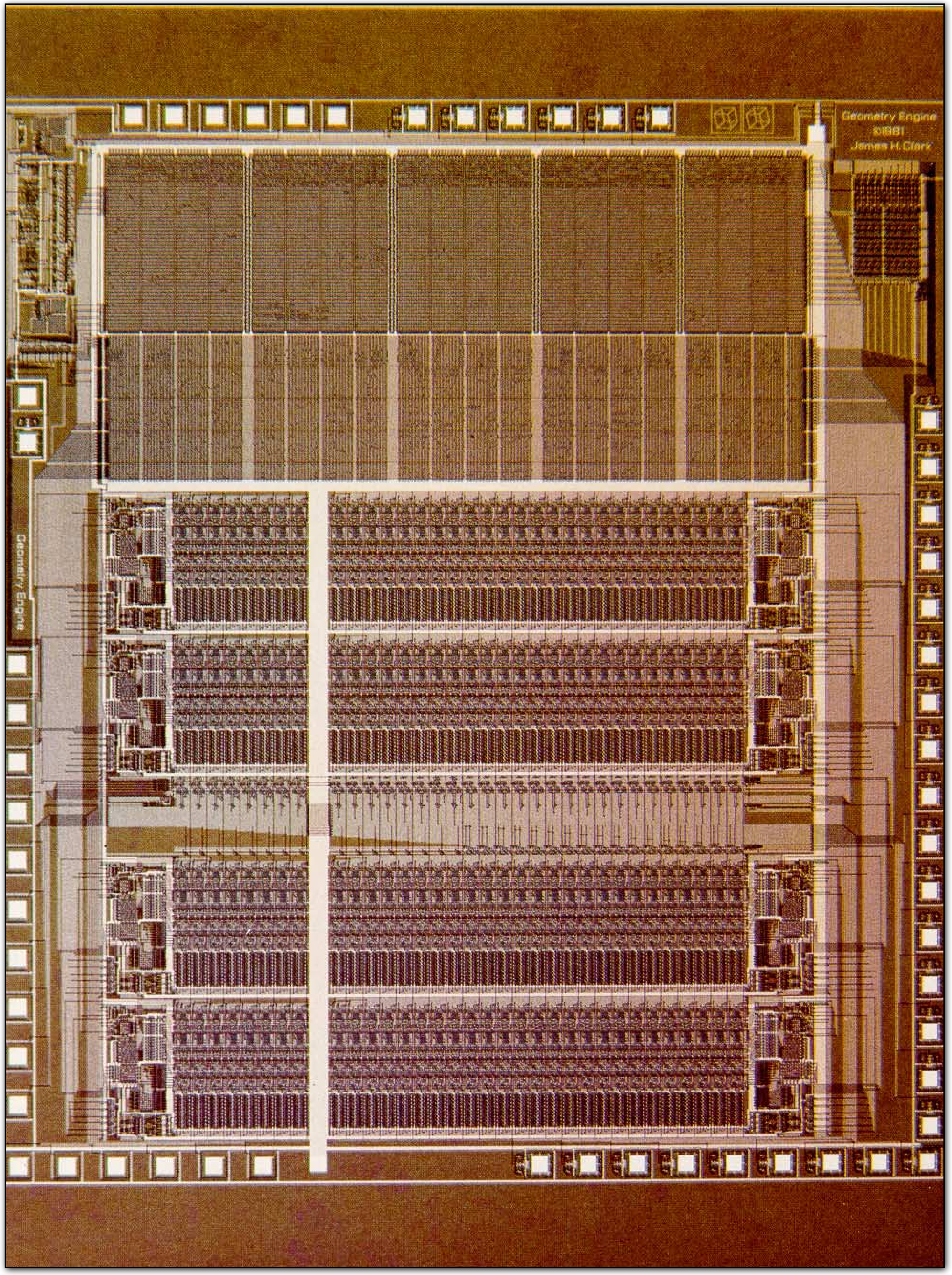
TI 74181 ALU



UNC Pixel Planes (1981), computation-enhanced frame buffer

Jim Clark's Geometry Engine (1982)

ASIC for geometric transforms used in real-time graphics



NVIDIA Titan V Volta GPU (2017)

~ 12 TFLOPs fp32

Similar to ASCI Q (top US supercomputer circa 2002)

Real-time rendering

Screenshot: Red Dead Redemption

Image analysis via deep learning

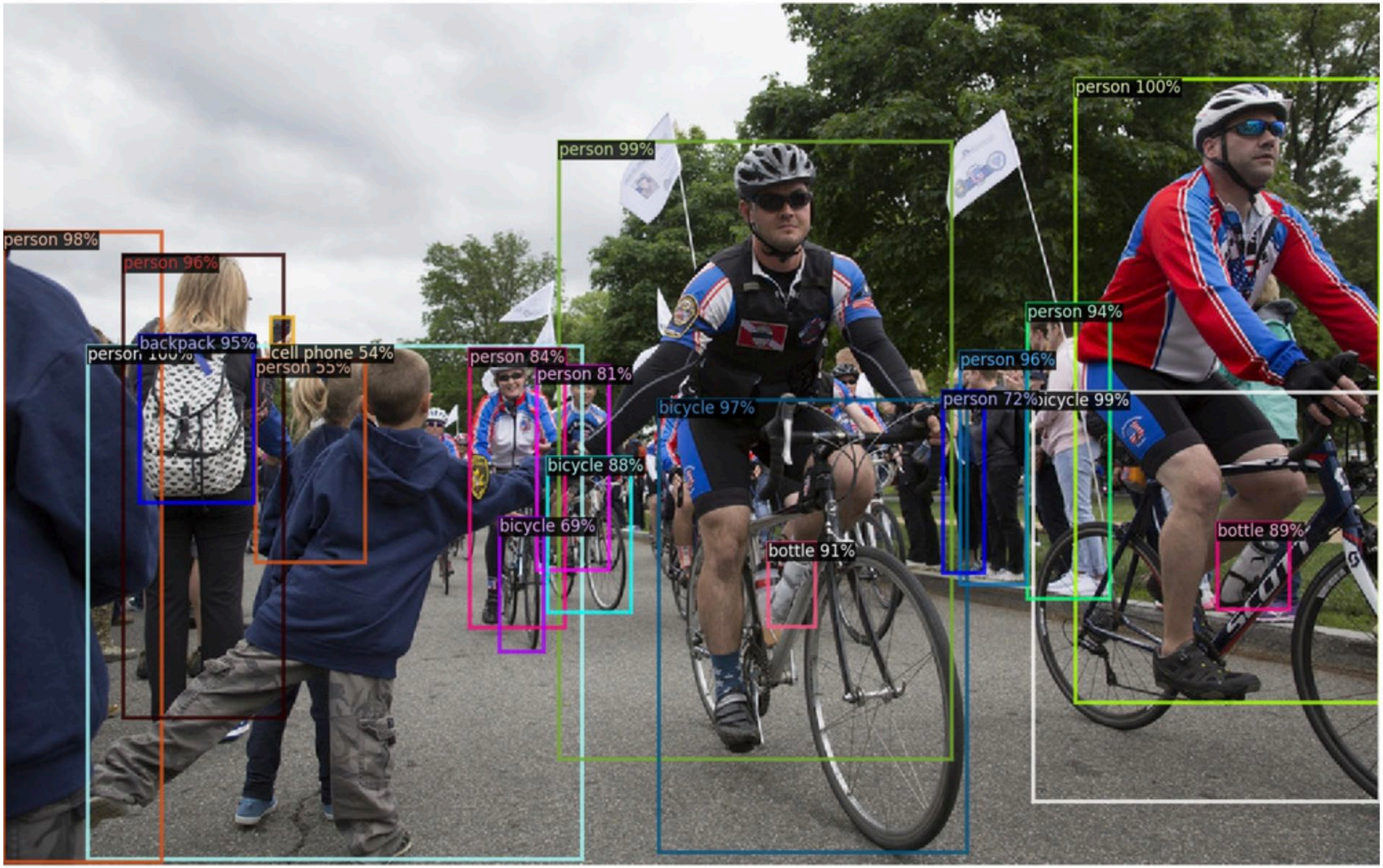
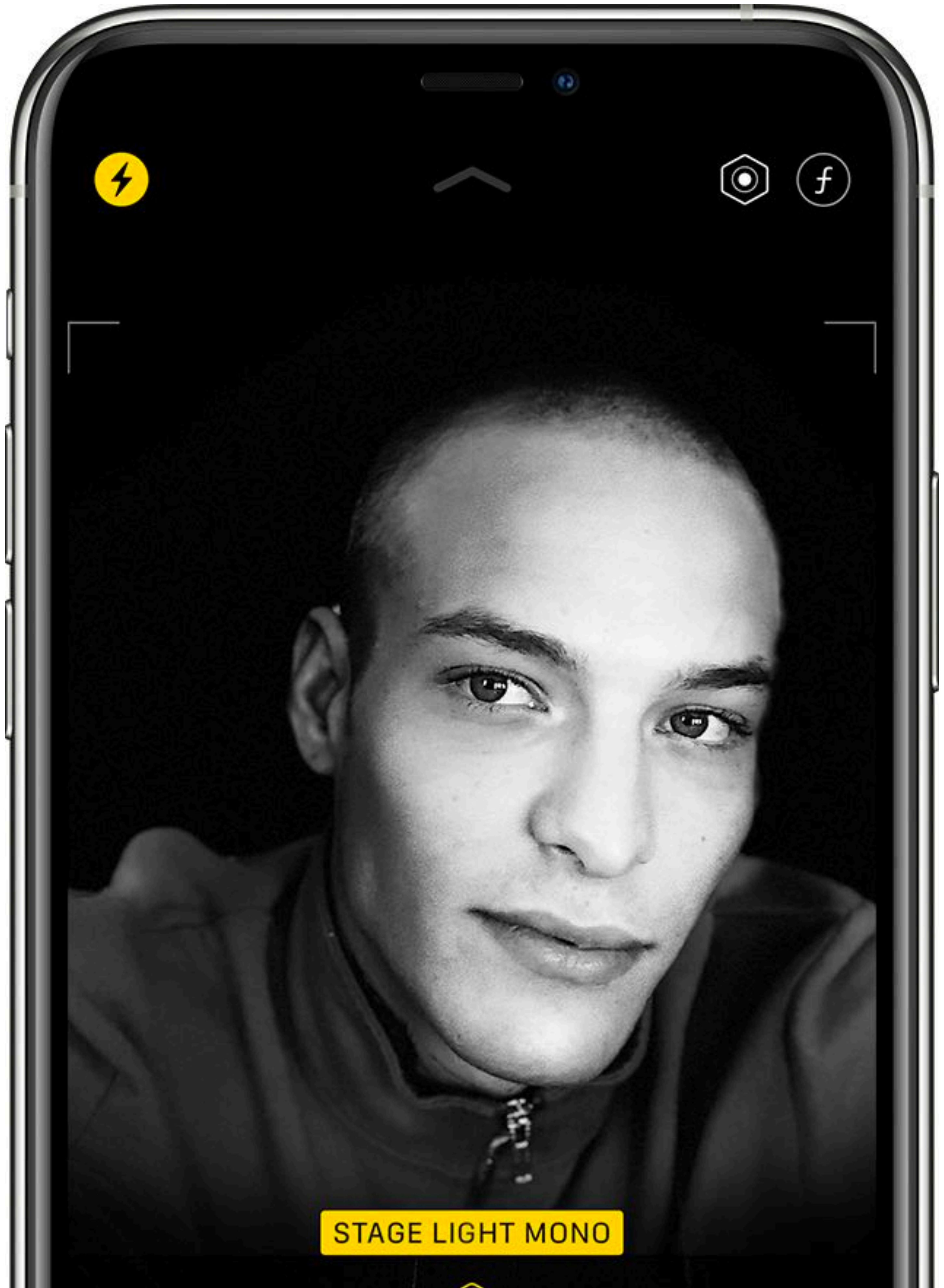


Image Credit: Detectron2

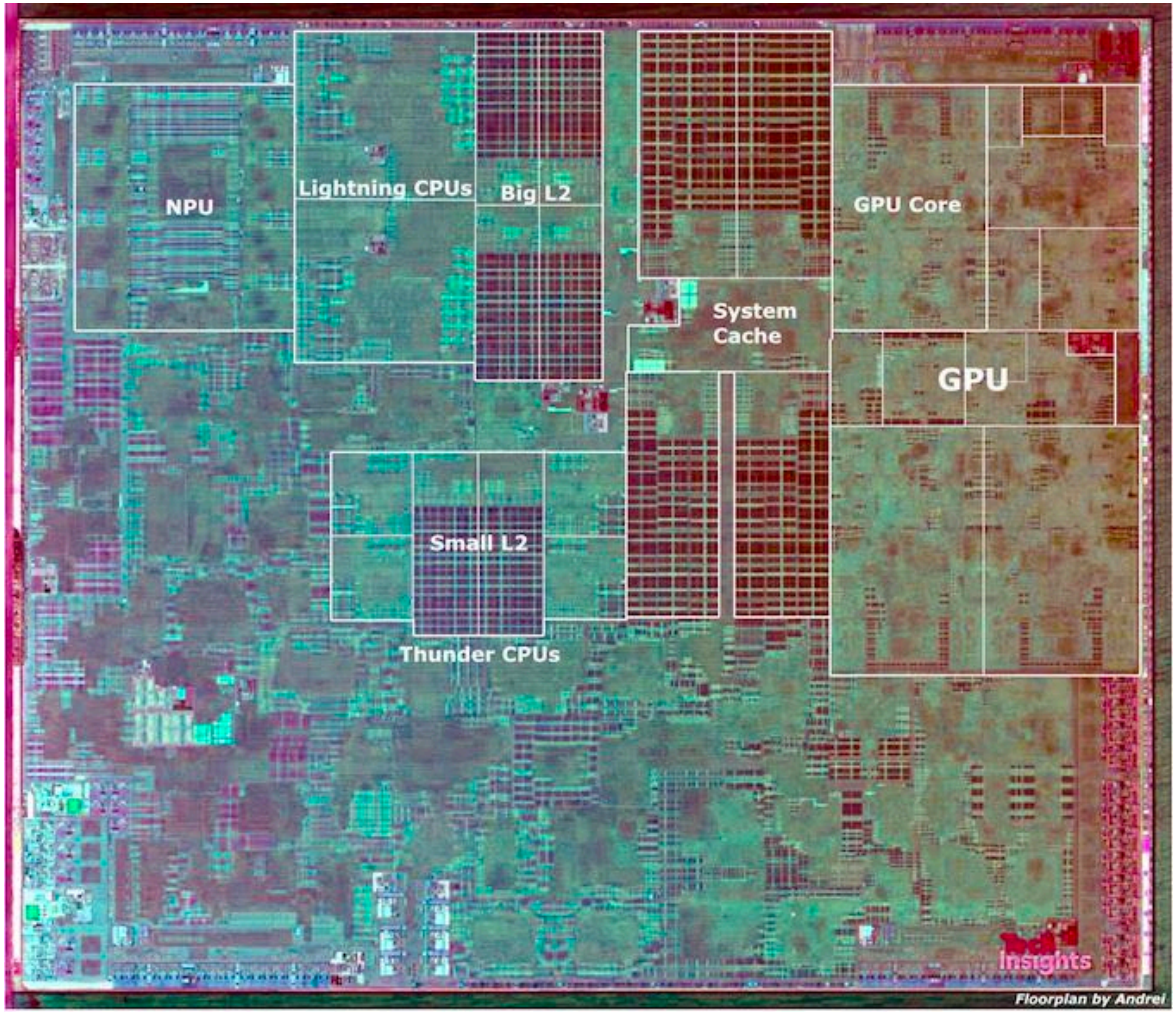
Digital photography: major driver of compute capability of modern smartphones

Portrait mode
(simulate effects of large aperture DSLR lens)



High dynamic range (HDR) photography

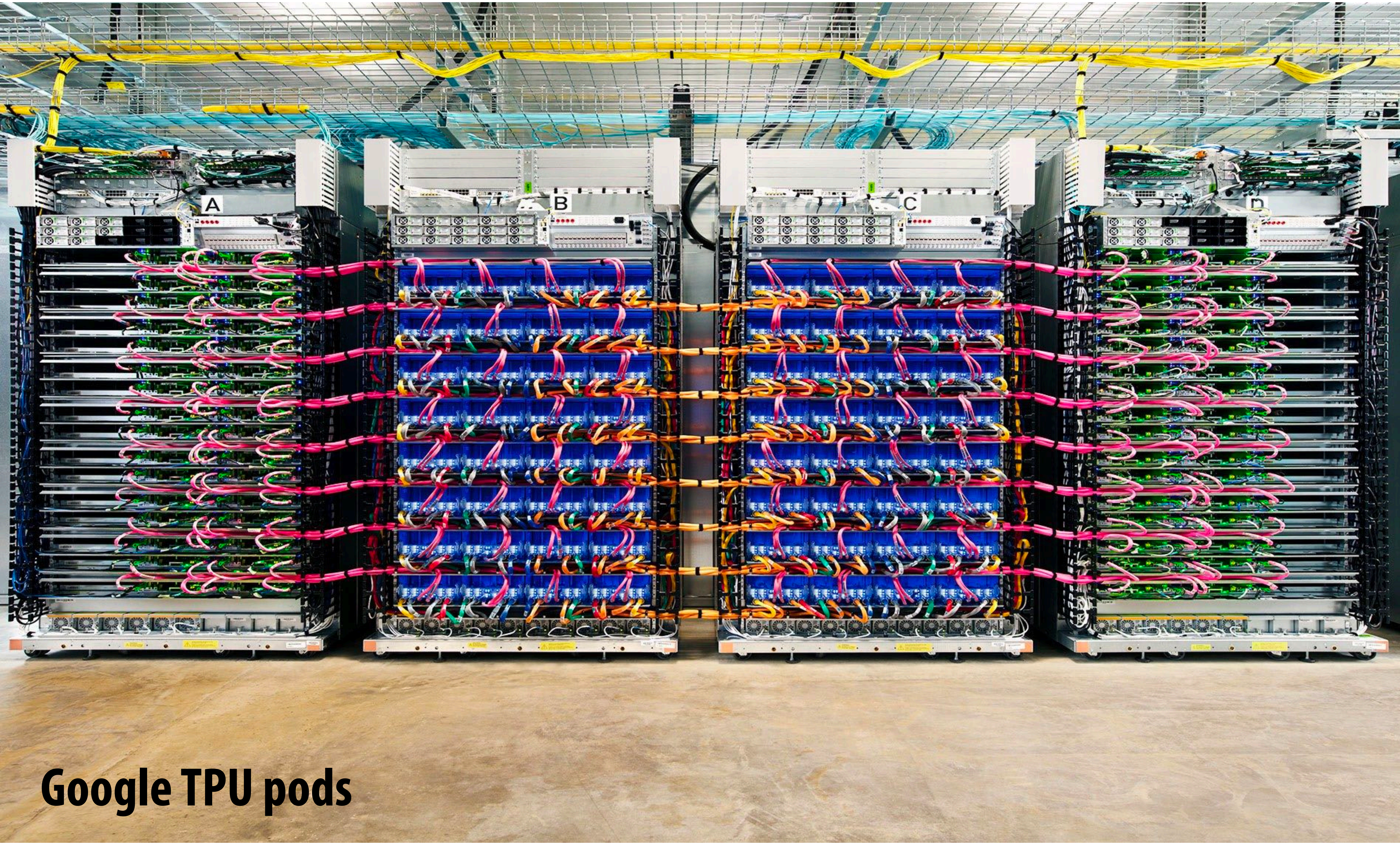
Modern smartphones utilize multiple processing units to quickly generate high-quality images



Apple A13 Bionic

Image Credit: Anandtech / TechInsights Inc.

Datacenter-scale applications

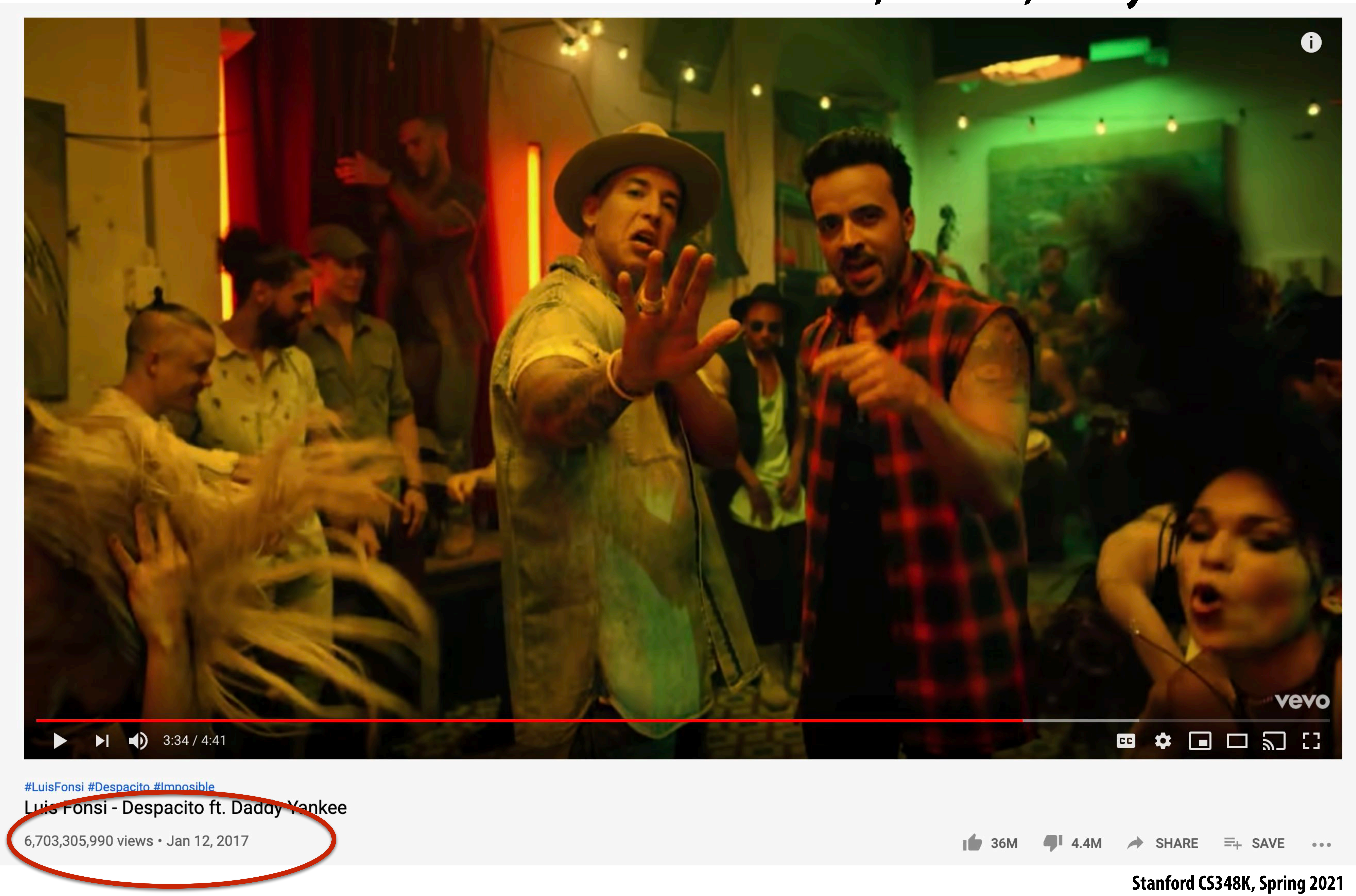


Google TPU pods

Image Credit: TechInsights Inc.

Youtube

Transcode, stream, analyze…

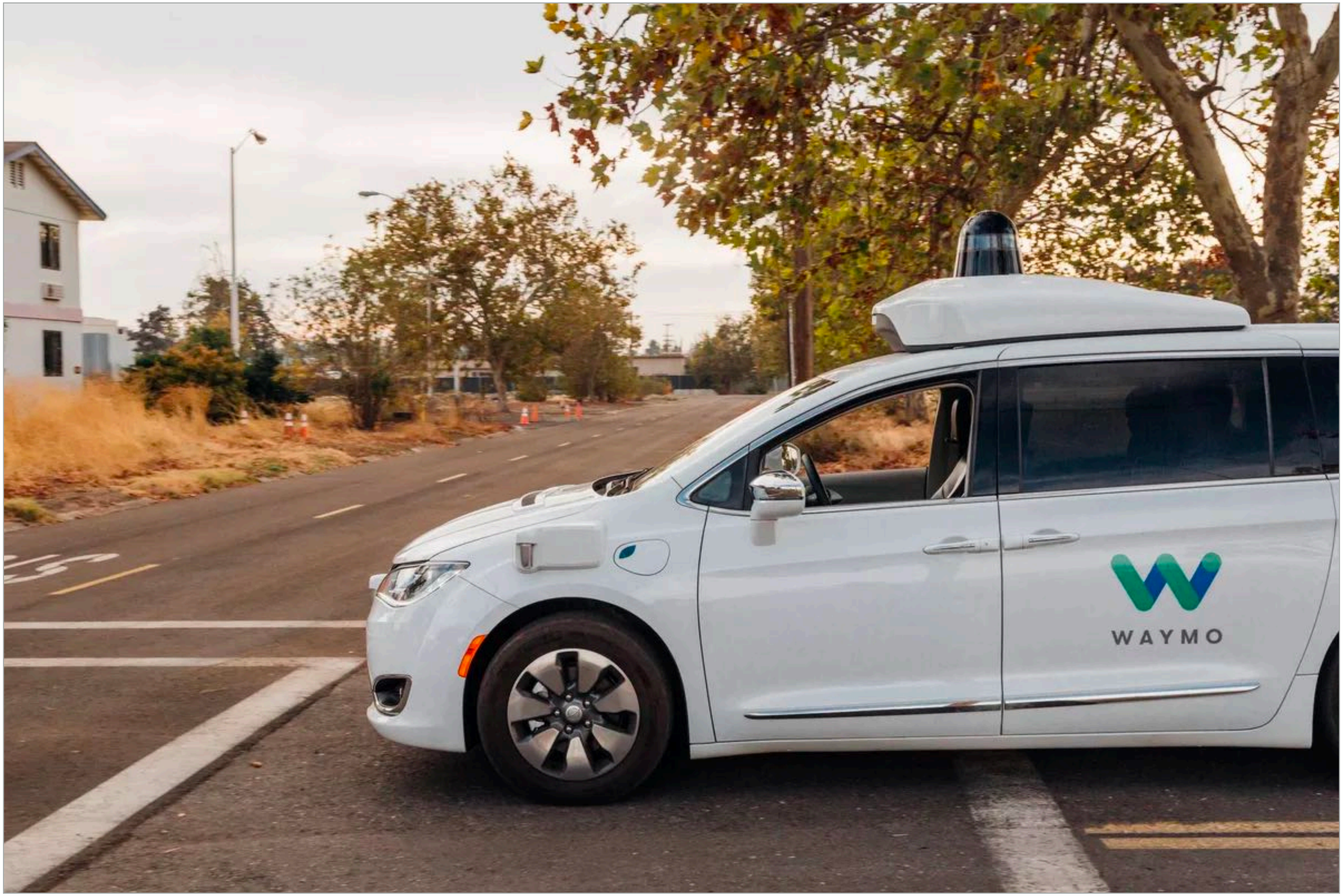


#LuisFonsi #Despacito #Imposible

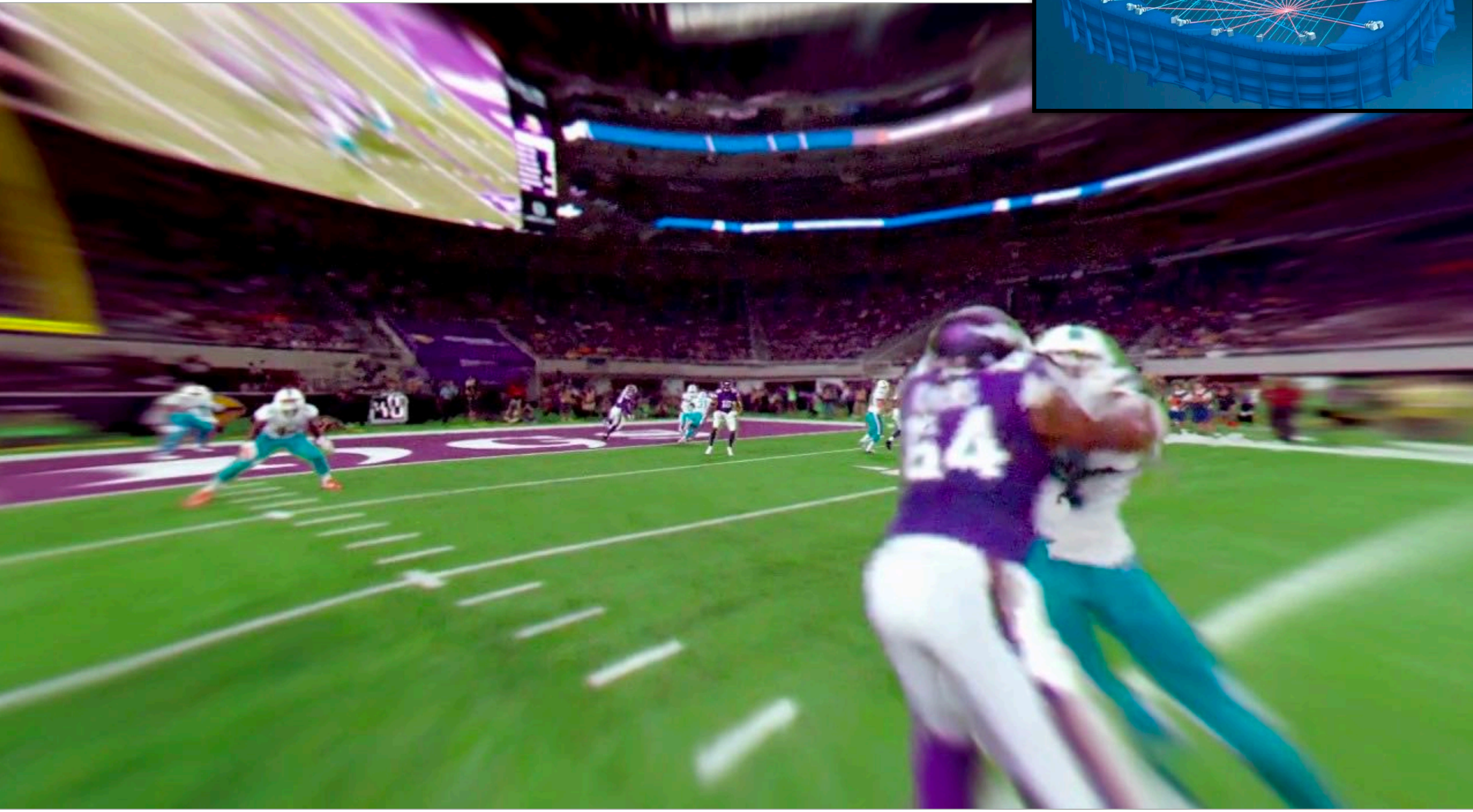
Luis Fonsi - Despacito ft. Daddy Yankee

6,703,305,990 views • Jan 12, 2017

On every vehicle: analyzing images for transportation



Unique visual experiences



Intel “FreeD”: 38 cameras in stadium used to reconstruct 3D, system renders new view from quarterback’s eyes

What is this course about?

Accelerator hardware architecture?

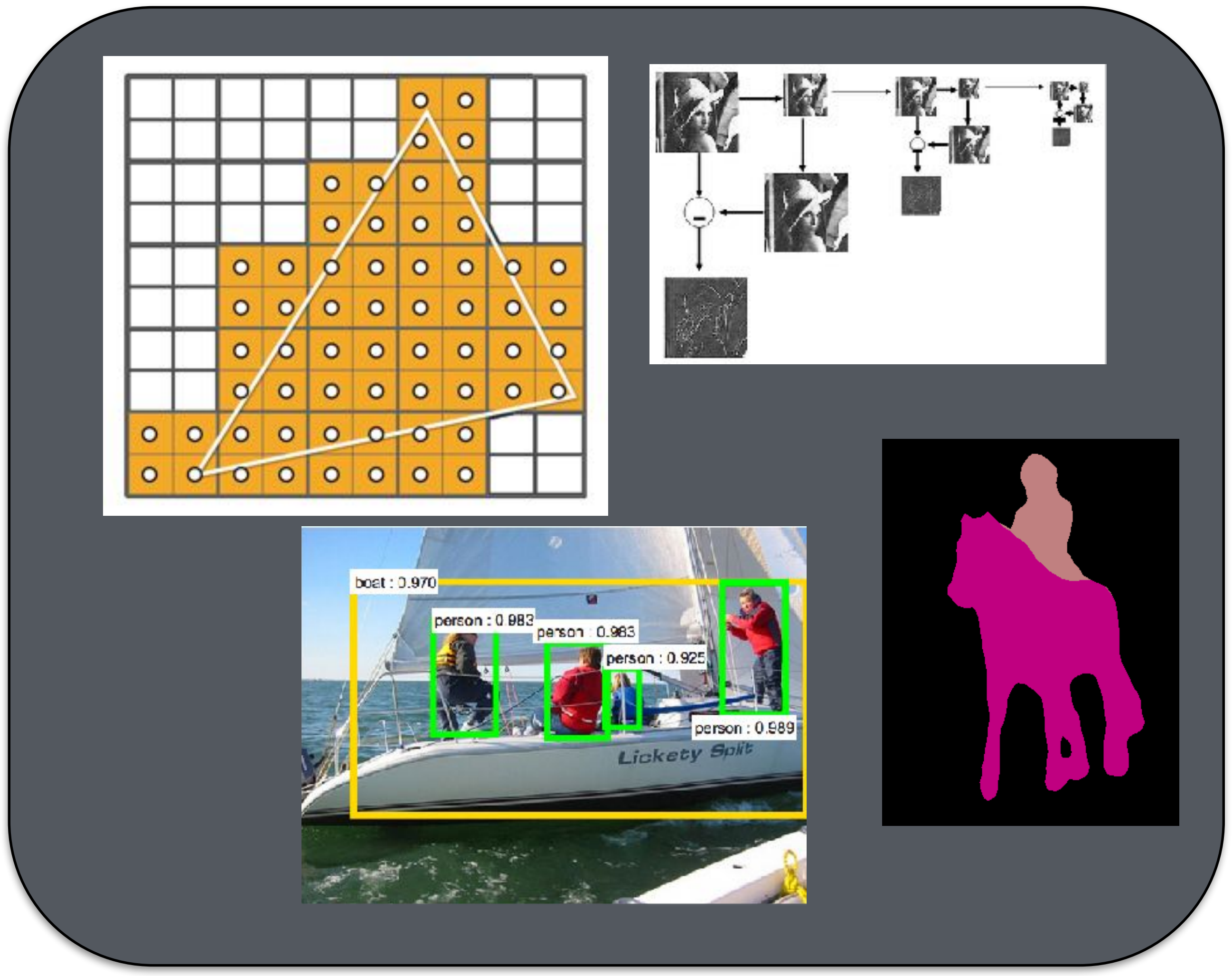
Graphics/vision/digital photography algorithms?

Programming systems?

What we will be learning about

Visual Computing Workloads

Algorithms for image/video processing, DNN evaluation, data compression, etc.

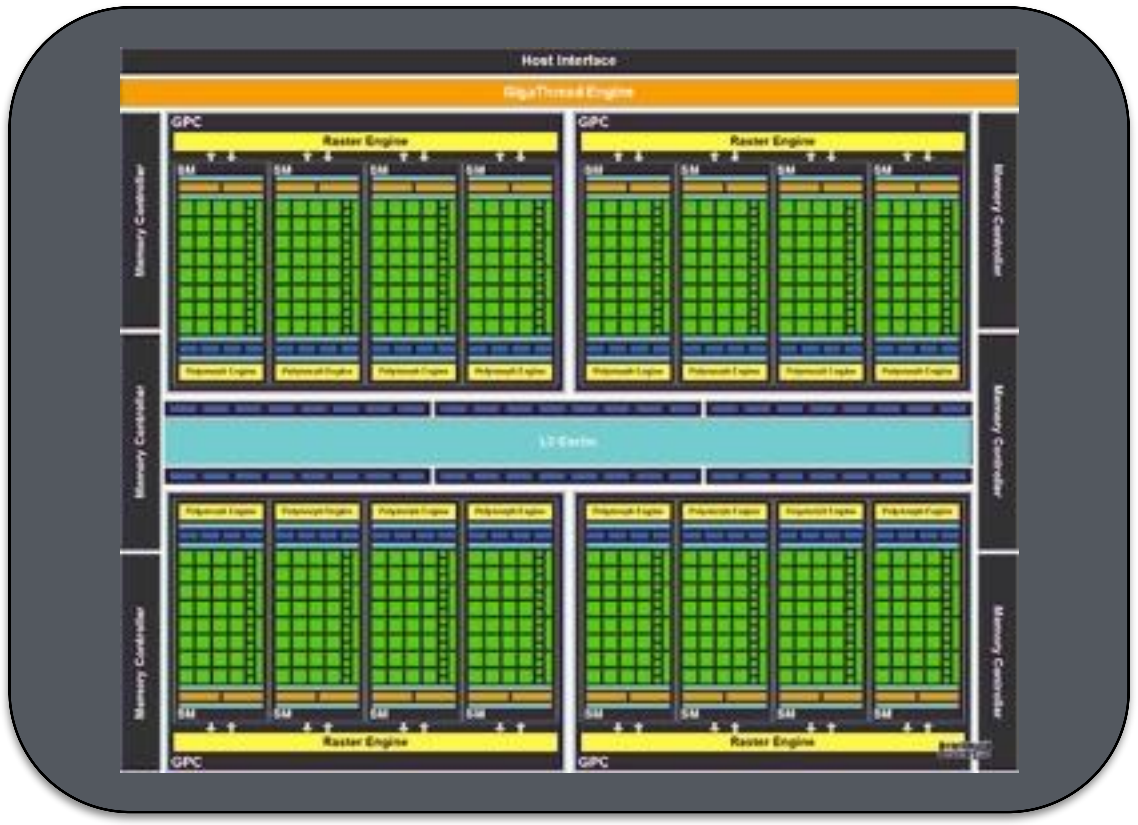


If you don’t understand key workload characteristics, how can you design a “good” system?

What we will be learning about

Modern Hardware Organization

High-throughput hardware designs (parallel, heterogeneous, and specialized) fundamental constraints like area and power



If you don't understand key constraints of modern hardware, how can you design algorithms that are well suited to run on it efficiently?

What we will be learning about

Programming Model Design

Choice of programming abstractions, level of abstraction issues, domain-specific vs. general purpose, etc.

Good programming abstractions enable productive development of applications, while also providing system implementors flexibility to explore highly efficient implementations

This course is about architecting efficient and scalable systems…

It is about the process of understanding the fundamental structure of problems in the visual computing domain, and then leveraging that understanding to…

To design more efficient and more robust algorithms

To build the most efficient hardware to run these algorithms

To design programming systems to make developing new applications simpler, more productive, and highly performant

Course topics

The digital camera photo processing pipeline in modern smartphones

- Basic algorithms (the workload)
- Programming abstractions for writing image processing apps
- Mapping these algorithms to parallel hardware

Systems for creating fast and accurate deep learning models

- Designing efficient DNN topologies, pruning, model search and AutoML
- Hardware for accelerating deep learning (why GPUs are not efficient enough!)
- Algorithms for parallel training (async and sync)
- Raising level of abstraction when designing models
- System support for automating data labeling

Processing and Transmitting Video

- Efficient DNN inference on video
- Trends in video compression (particularly relevant these days)
- How video conferencing systems work

Recent advances in real-time (hardware accelerated) ray tracing

- Ray tracing workload
- Recent API and hardware support for real-time ray tracing
- How deep learning, combined with RT hardware, is making real time ray tracing possible

Course Logistics

Logistics of a all-virtual class

- **Course web site:**
  - **http://cs348k.stanford.edu**
  - **My goal is to post lecture slides the night before class**

- **All announcements will go out via Piazza (not via Canvas)**

Expectations of you

- 40% participation
  - There will be ~1 assigned paper reading per class
  - You will submit a response to each reading by noon on class days
  - We will start the class with a discussion of the reading

- 20% two programming assignments (first 1/2 of course)
  - Implement and optimize a simple HDR photography processing pipeline
  - Understanding why “blocking” a conv layer in a DNN matters

- 40% self-selected term project
  - I suggest you start thinking about projects now

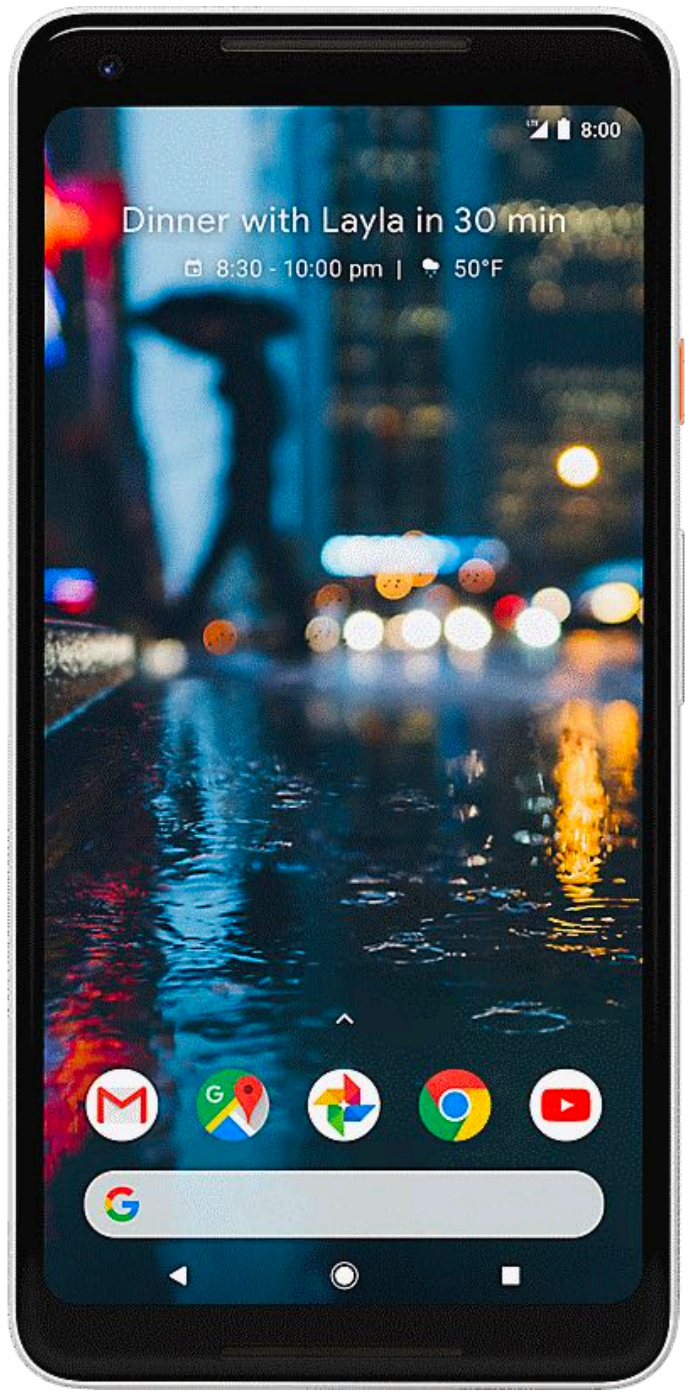
Review (or crash course):

key principles of modern throughput computing hardware

Let's crack open a modern smartphone

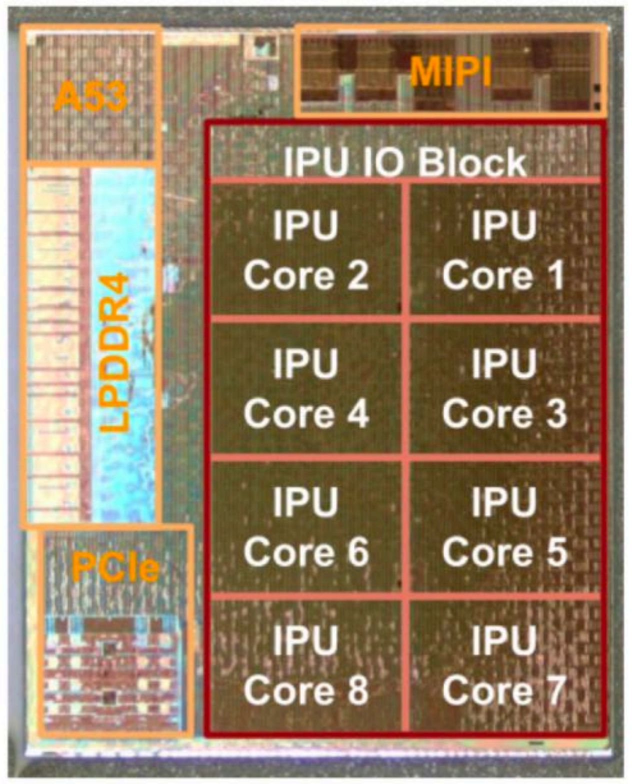
Google Pixel 2 Phone:

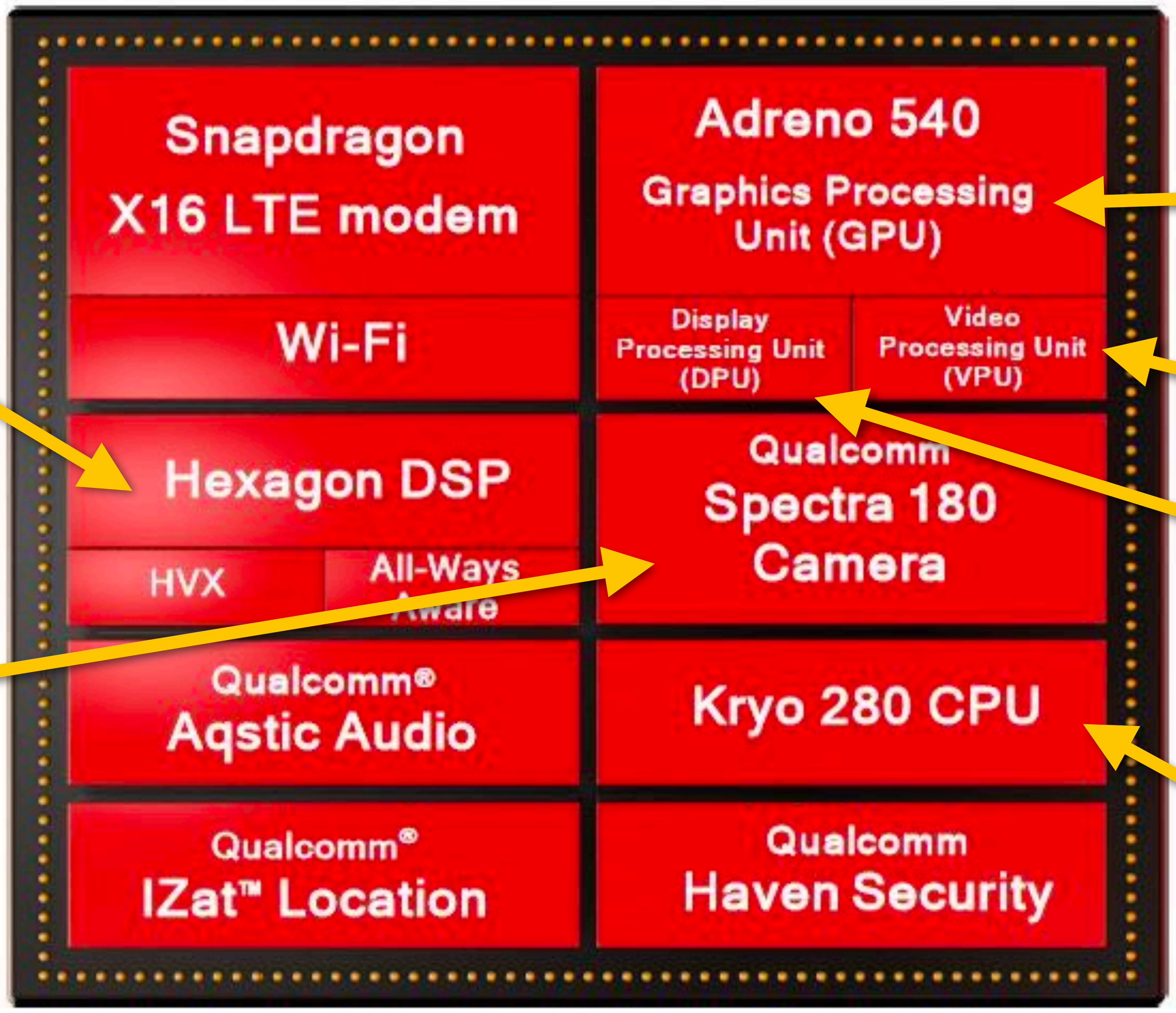
Qualcomm Snapdragon 835 SoC + Google Visual Pixel Core



Visual Pixel Core

Programmable image processor and DNN accelerator





"Hexagon" Programmable DSP

data-parallel multi-media processing

Image Signal Processor

ASIC for processing camera sensor pixels

Multi-core GPU

(3D graphics, OpenCL data-parallel compute)

Video encode/decode ASIC

Display engine

(compresses pixels for transfer to high-res screen)

Multi-core ARM CPU

4 "big cores" + 4 "little cores"

Three things to know

1. What are these three hardware design strategies, and what problem/goals do they address?
    - Muti-core processing
    - SIMD processing
    - Hardware multi-threading

2. What is the motivation for specialization via…
    - Multiple types of processors (e.g., CPUs, GPUs)
    - Custom hardware units (ASIC)

3. Why is memory bandwidth a major constraint (often the most important constraint) when mapping applications to modern computer systems?

Multi-core processing

Review: what does a processor do?

It runs programs!

Processor executes instruction referenced by the program counter (PC)

(executing the instruction will modify machine state: contents of registers, memory, CPU state, etc.)

Move to next instruction …

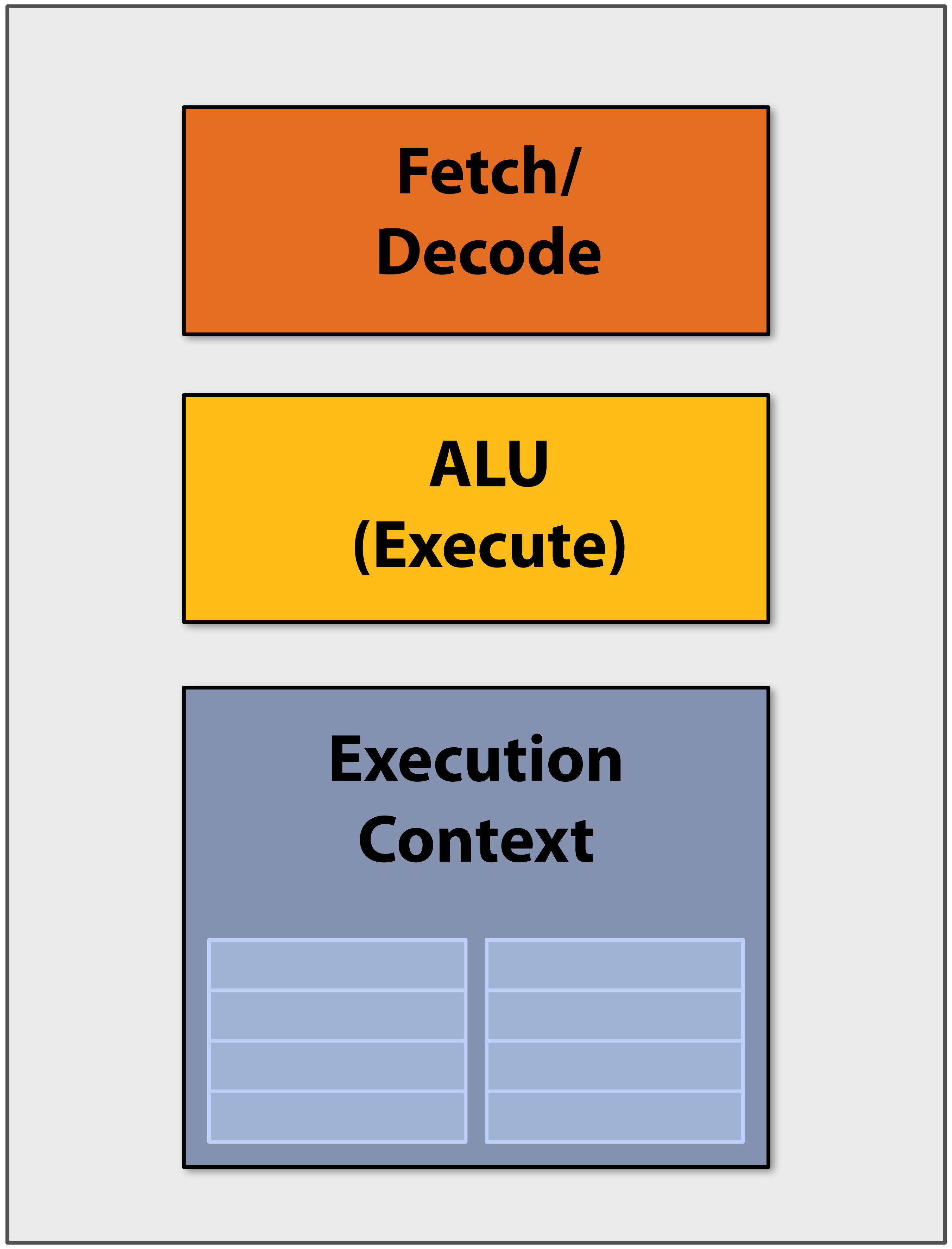
Then execute it…

PC →

And so on…

```
_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:   leaq58(%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:   retq
```

Executing an instruction stream



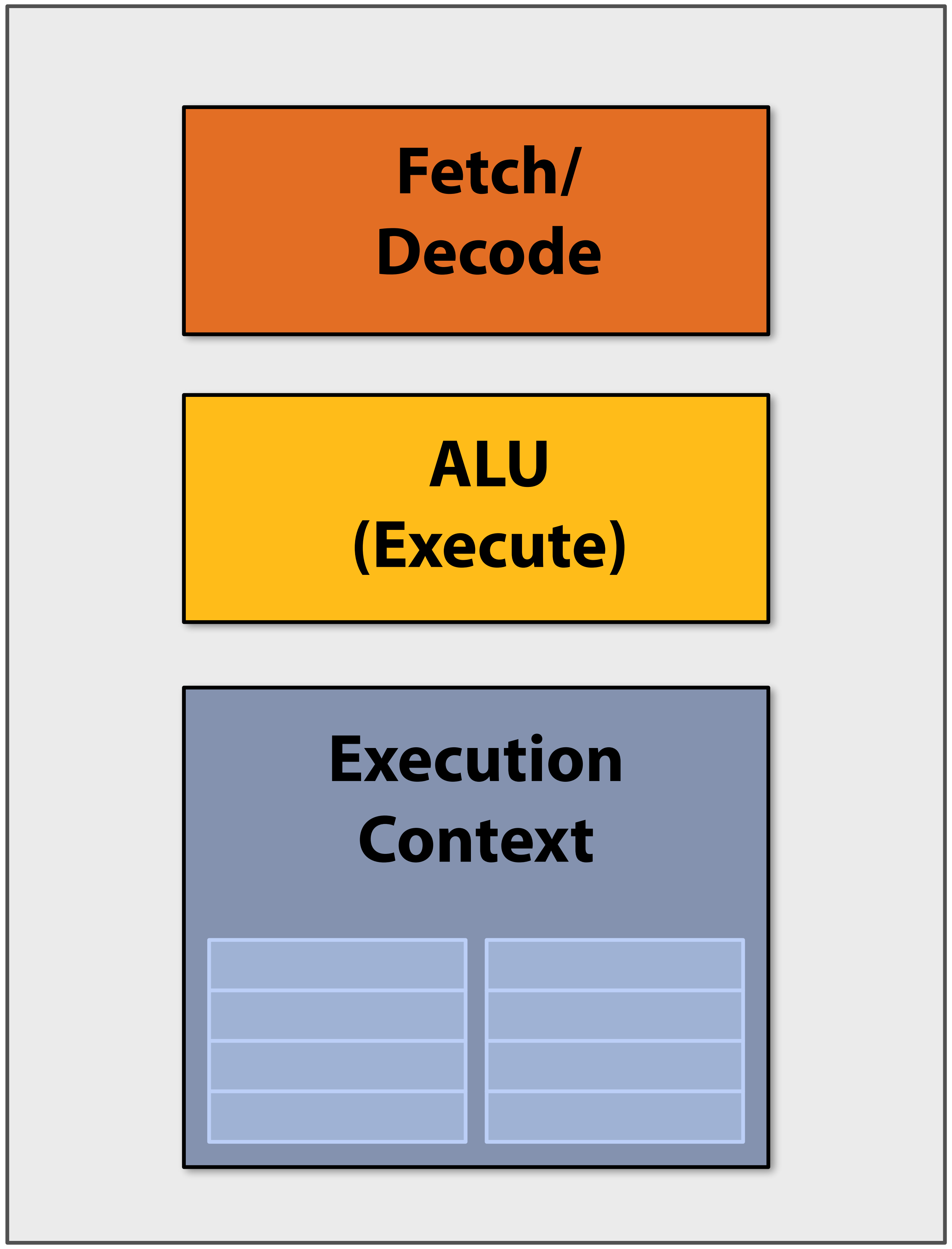
x[i]

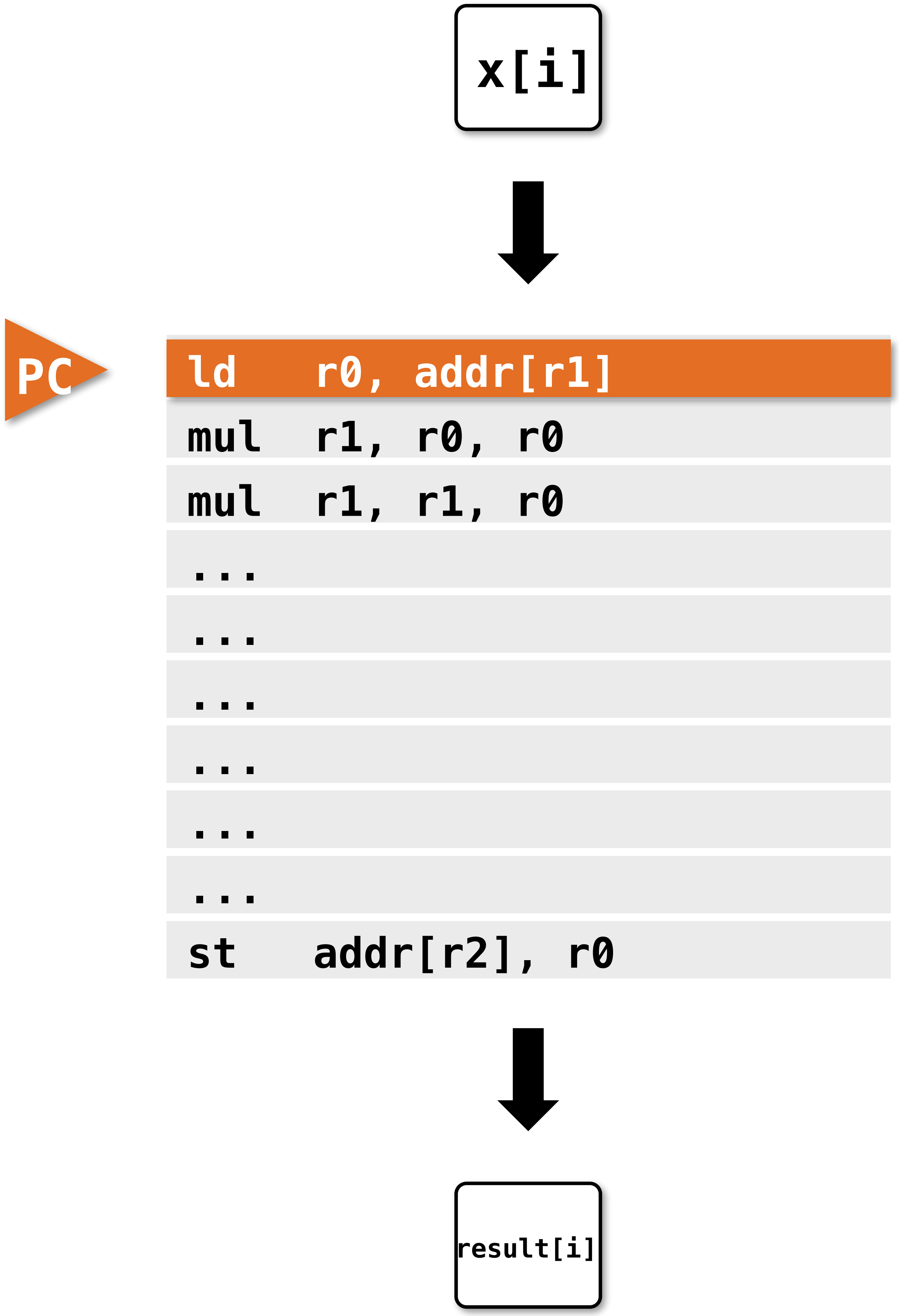
```
ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...
...
...
...
...
...
st   addr[r2], r0
```

result[i]

Executing an instruction stream

My very simple processor: executes one instruction per clock

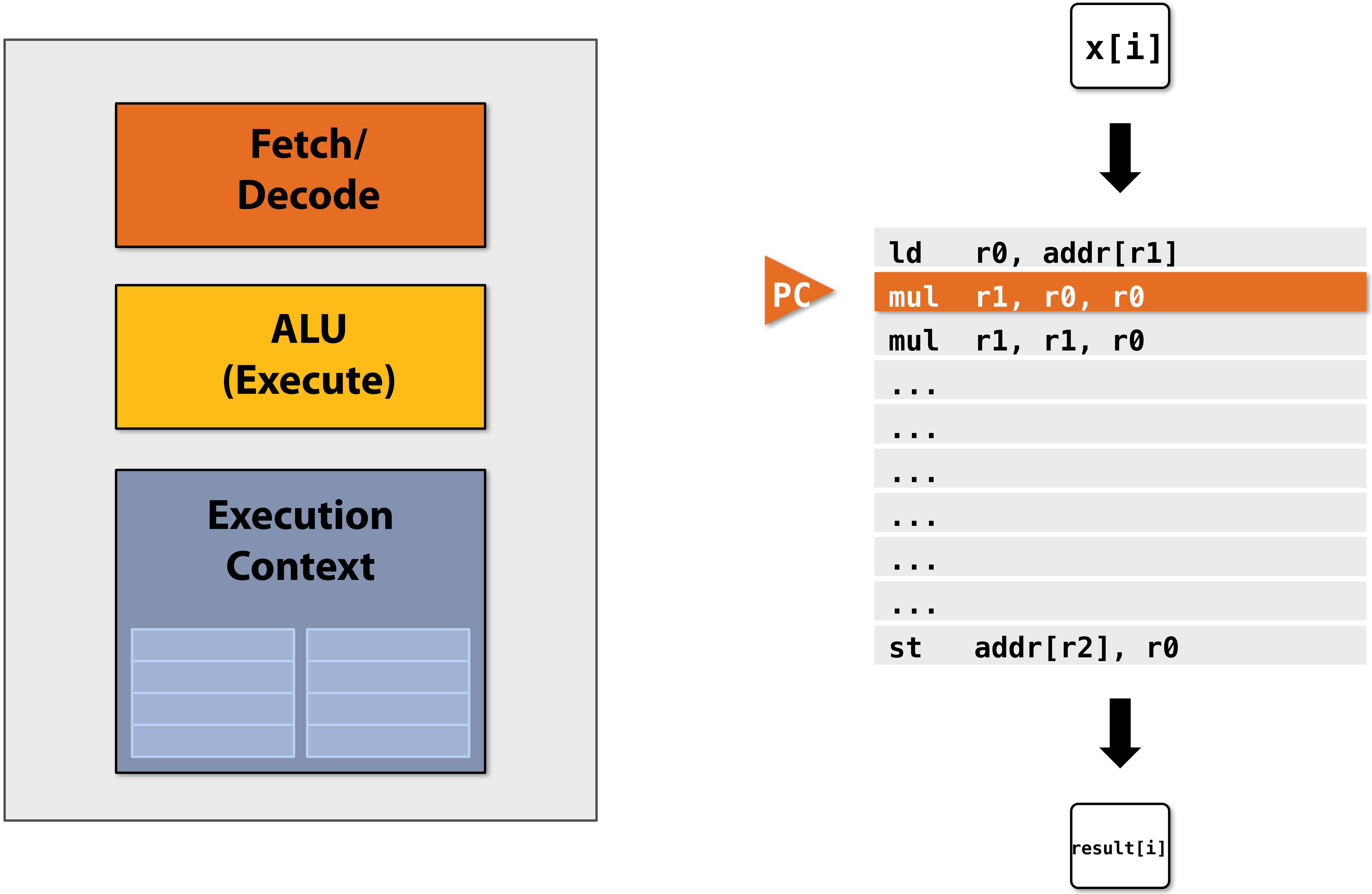




```
ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...
...
...
...
...
...
st   addr[r2], r0
```

Executing an instruction stream

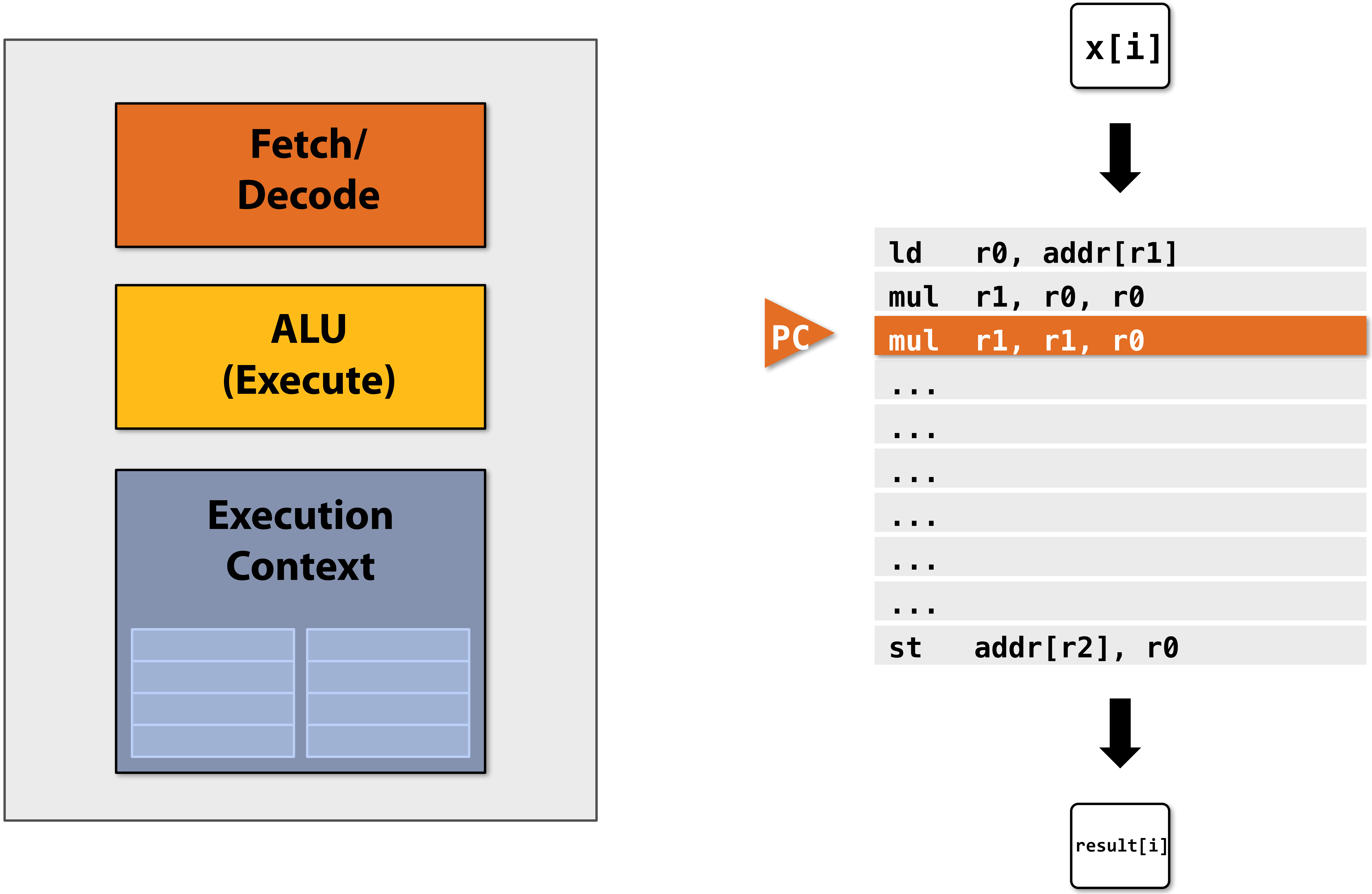
My very simple processor: executes one instruction per clock



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

Executing an instruction stream

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
```

Quick aside:
Instruction-level parallelism and superscalar execution

Instruction level parallelism (ILP) example

```
a = x*x + y*y + z*z
```

Consider the following program:

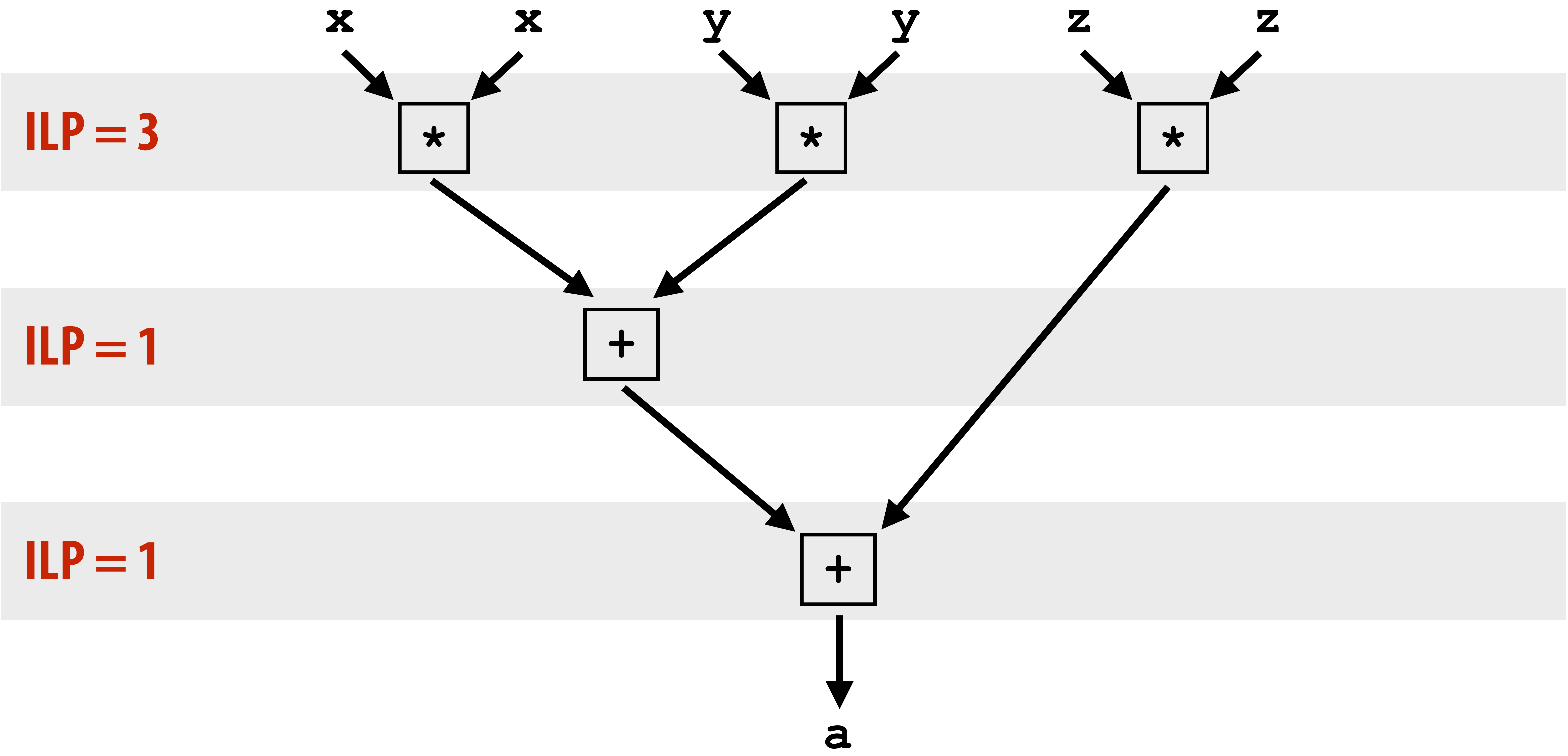
```
// assume r0=x, r1=y, r2=z

mul r0, r0, r0
mul r1, r1, r1
mul r2, r2, r2
add r0, r0, r1
add r3, r0, r2

// now r3 stores value of program variable ‘a’
```

This program has five instructions, so it will take five clocks to execute, correct?

Can we do better?

ILP example

```
a = x*x + y*y + z*z
```



Superscalar execution

```
a = x*x + y*y + z*z
```

```
    // assume r0=x, r1=y, r2=z

1.  mul r0, r0, r0
2.  mul r1, r1, r1
3.  mul r2, r2, r2
4.  add r0, r0, r1
5.  add r3, r0, r2

    // r3 stores value of variable 'a'
```

Superscalar execution: processor automatically finds independent instructions in an instruction sequence and executes them in parallel on multiple execution units!

In this example: instructions 1, 2, and 3 can be executed in parallel
(on a superscalar processor that determines that the lack of dependencies exists)

But instruction 4 must come after instructions 1 and 2

And instruction 5 must come after instruction 4

Superscalar execution

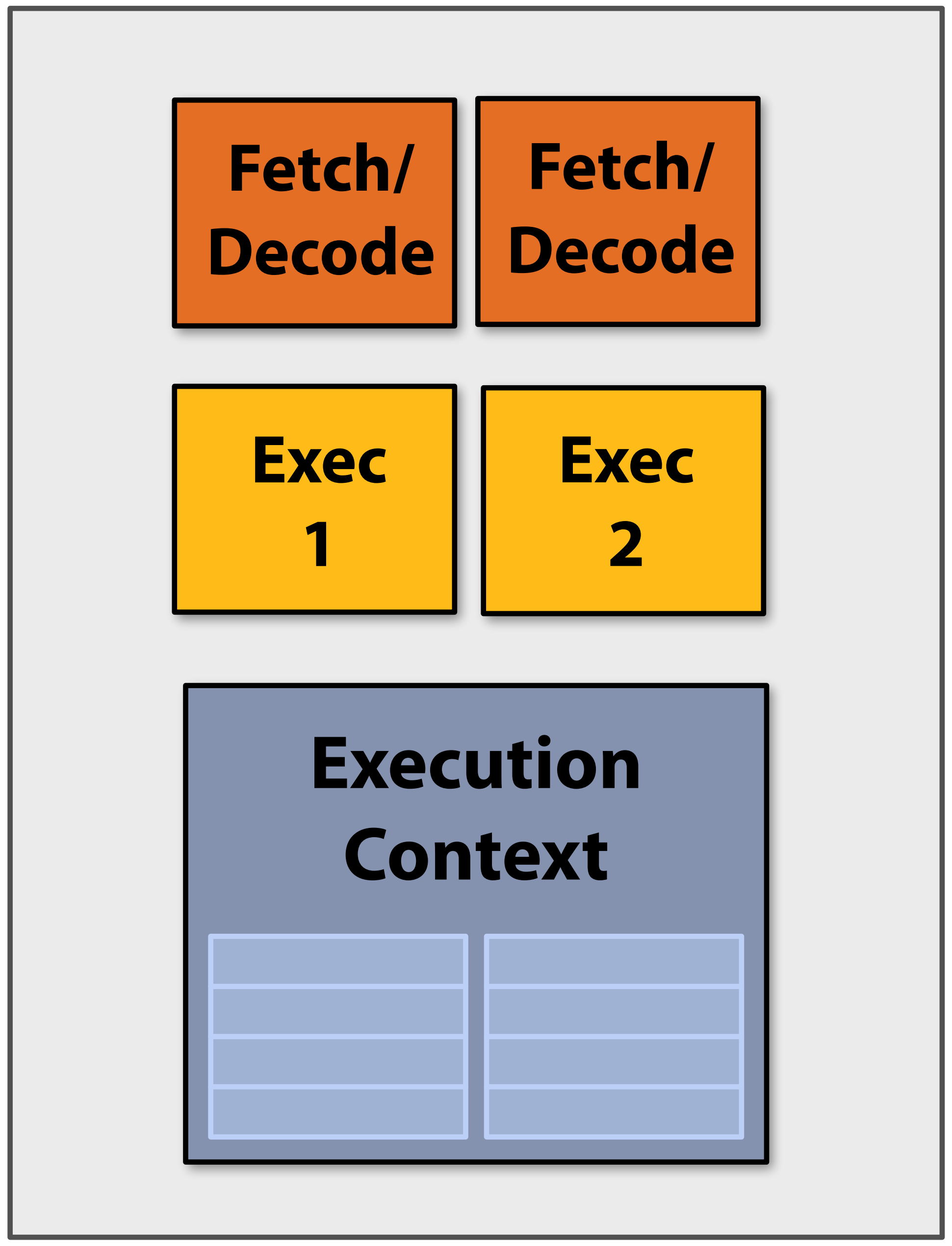
Program: computes sin of input *x* via Taylor expansion

```
void sinx(int N, int terms, float x)
{
      float value = x;
      float numer = x * x * x;
      int denom = 6;  // 3!
      int sign = -1;

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

      return value;
}
```

Independent operations in instruction stream

(They are detected by the processor at run-time and may be executed in parallel on execution units 1 and 2)

My single core, superscalar processor: executes up to two instructions per clock from a single instruction stream.



Now consider a program that computes the sine of <u>many</u> numbers…

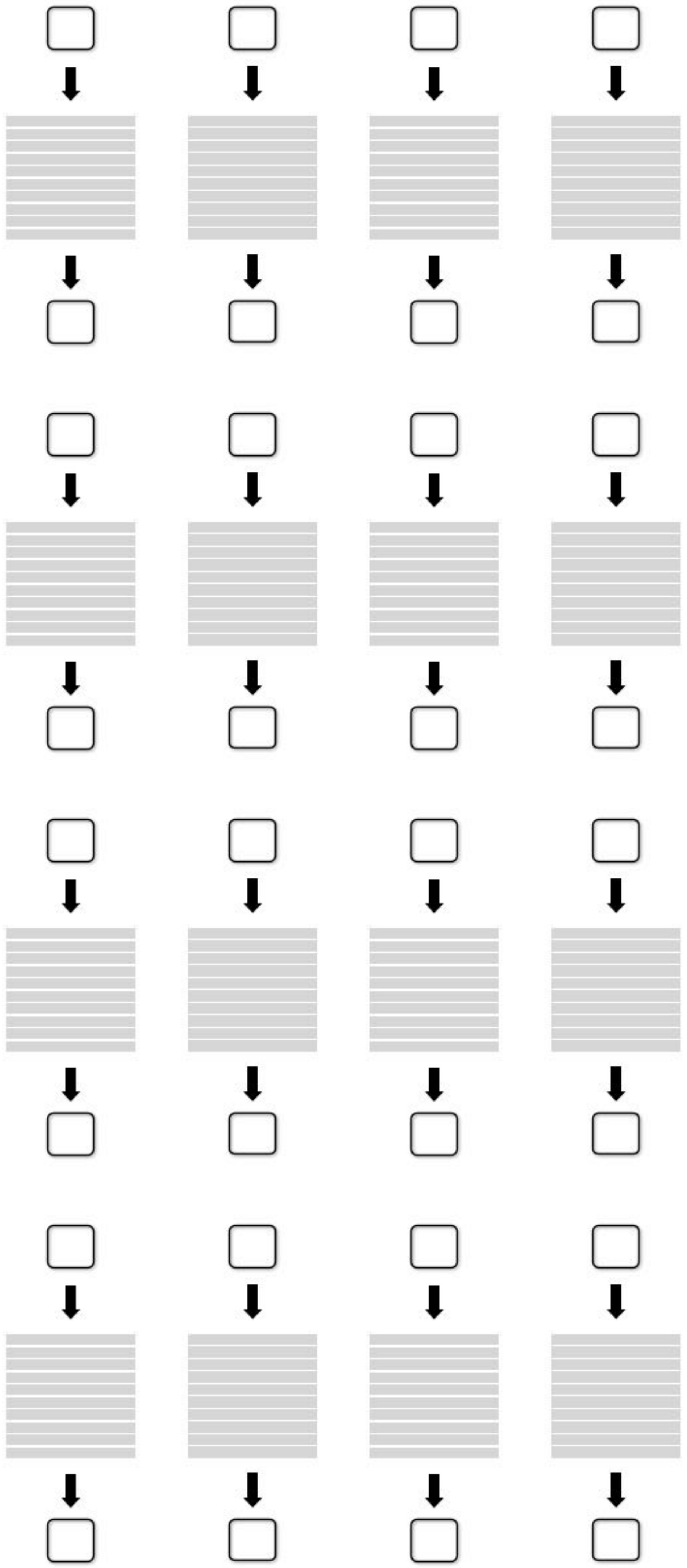
Example program

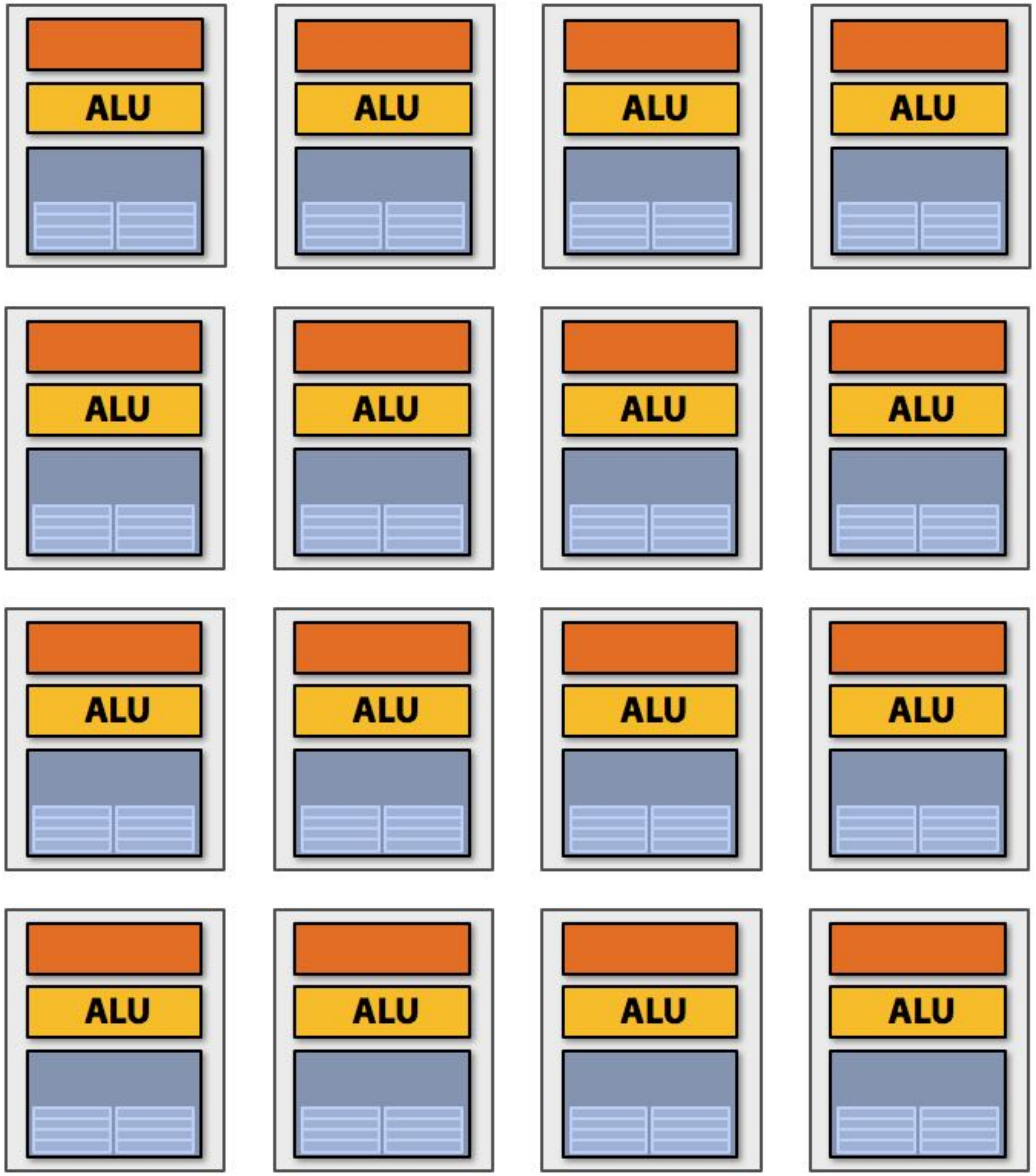
Compute $\sin(x)$ **using Taylor expansion:** $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \ldots$

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

```
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;
   }
}
```

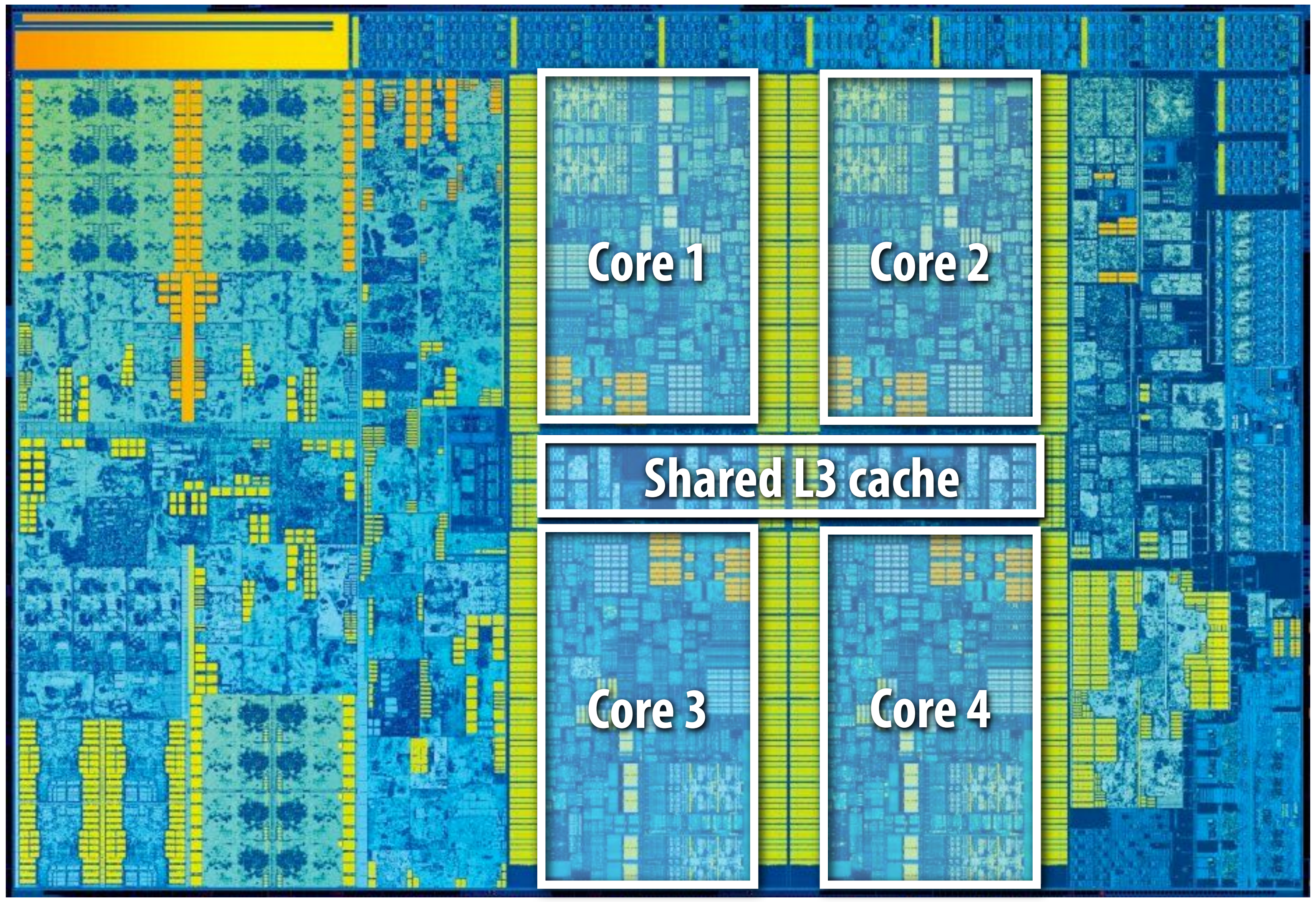
Multi-core: process multiple instruction streams in parallel



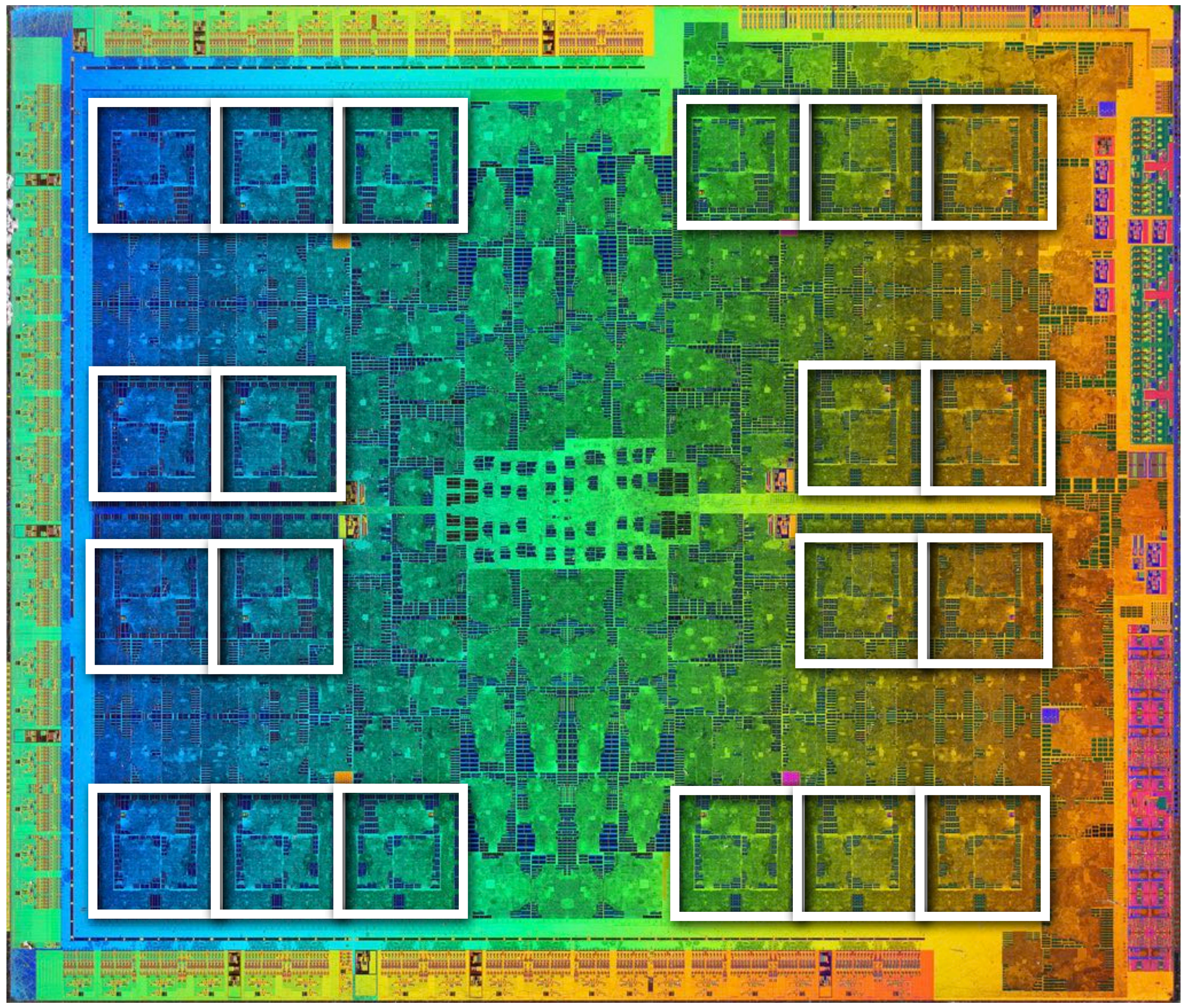


Sixteen cores, sixteen simultaneous instruction streams

Multi-core examples

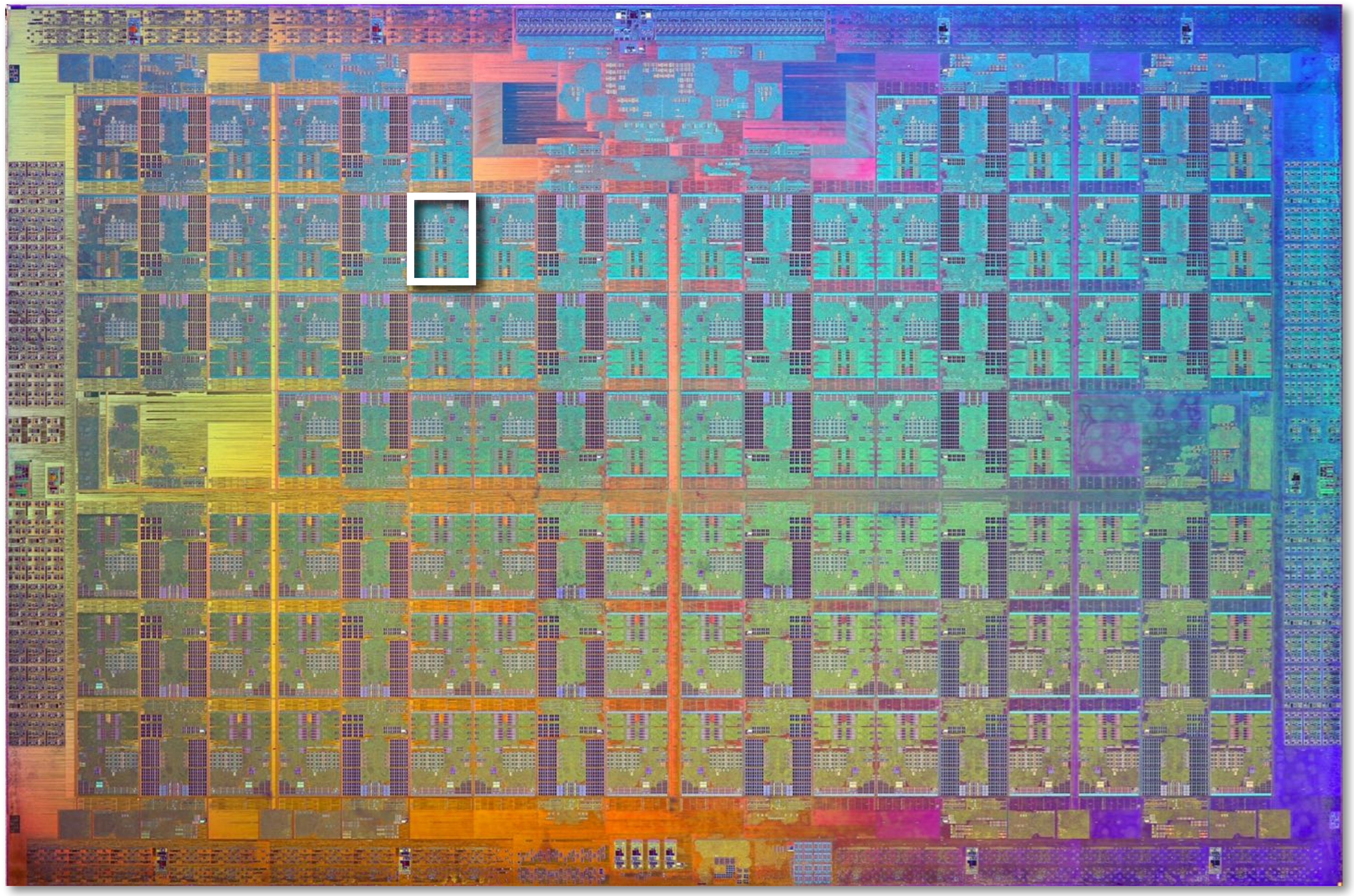


Intel "Skylake" Core i7 quad-core CPU
(2015)

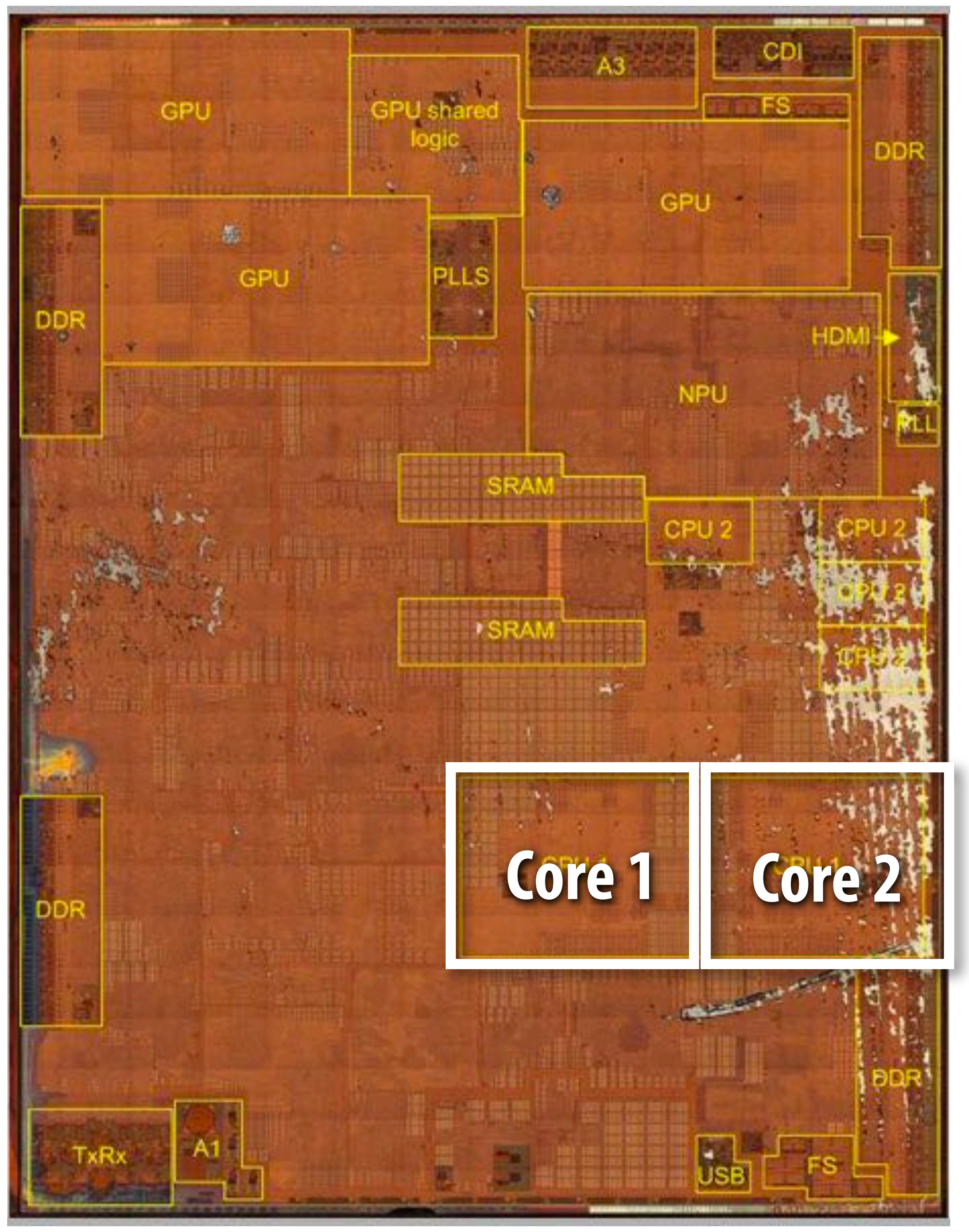


NVIDIA GP104 (GTX 1080) GPU
20 replicated ("SM") cores
(2016)

More multi-core examples



Intel Xeon Phi “Knights Landing “ 76-core CPU
(2015)

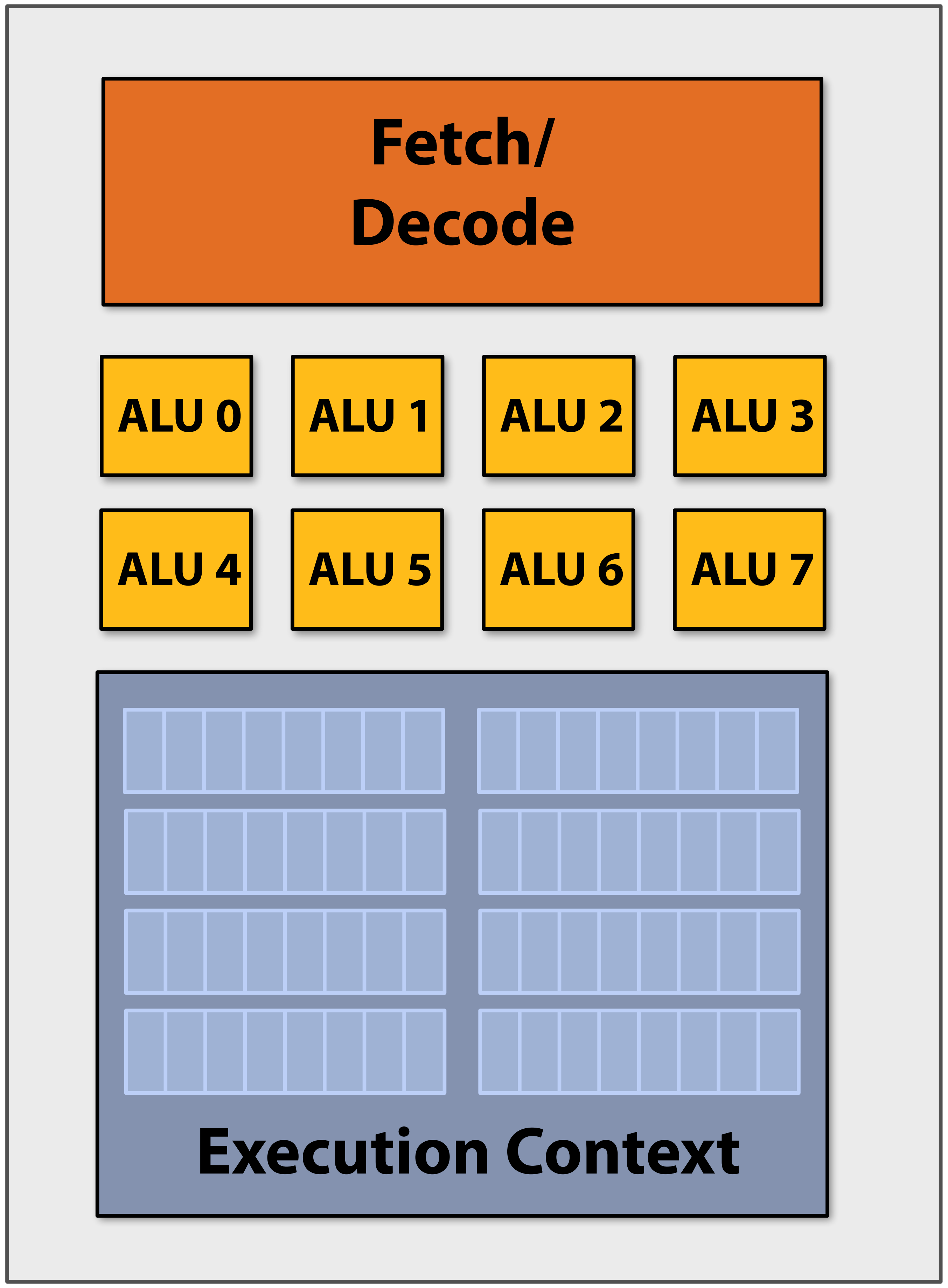


Apple A11 Bionic CPU
Two “big” cores
Four “small cores”
(2017)

A9 image credit: Chipworks (obtained via Anandtech)
http://www.anandtech.com/show/9686/the-apple-iphone-6s-and-iphone-6s-plus-review/3

SIMD processing

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

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)

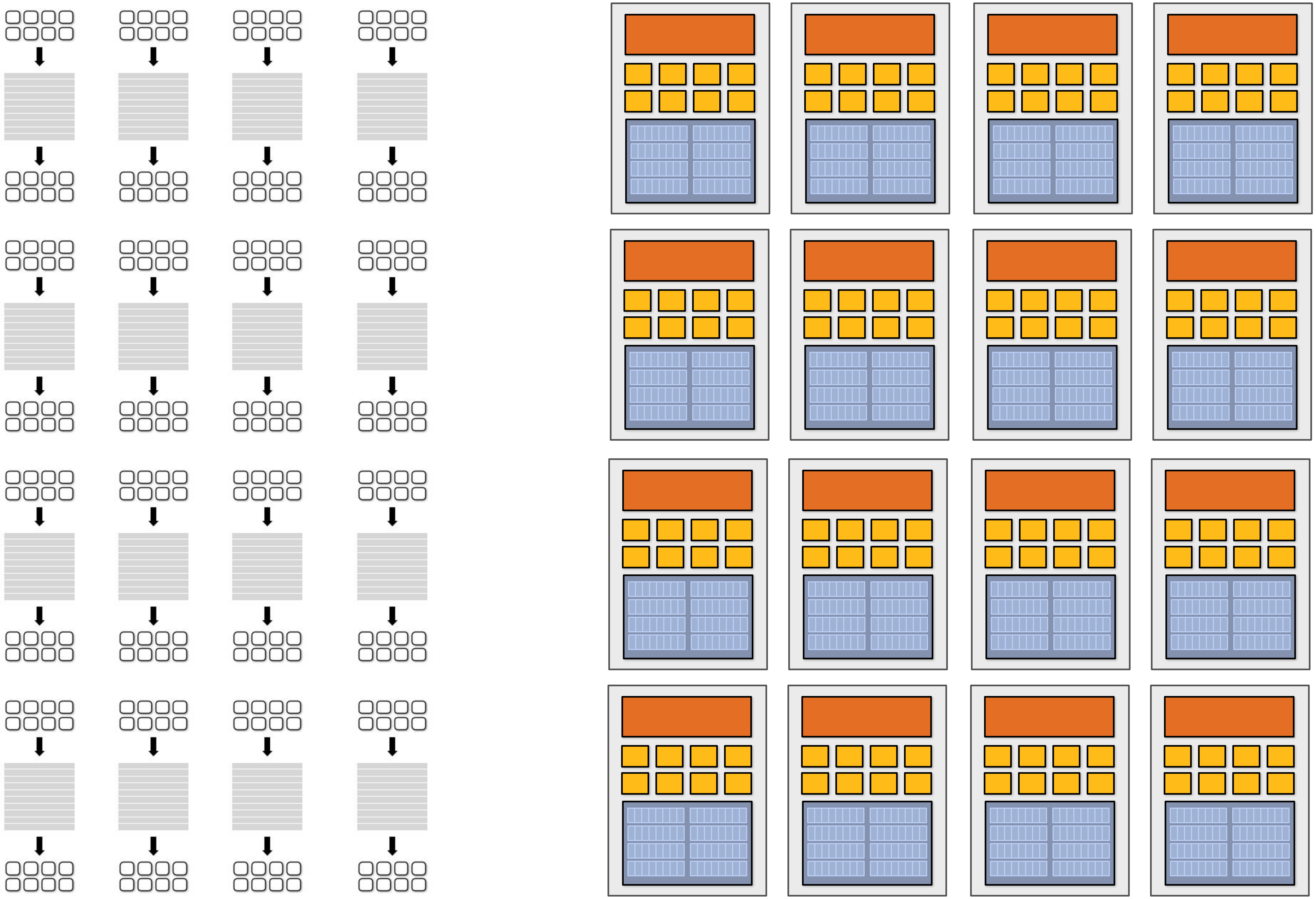
```
#include <immintrin.h>
void sinx(int N, int terms, float* x, float* sinx)
{
   float three_fact = 6;  // 3!
   for (int i=0; i<N; i+=8)
   {
       __m256 origx = _mm256_load_ps(&x[i]);
      __m256 value = origx;
      __m256 numer = _mm256_mul_ps(origx, _mm256_mul_ps(origx, origx));
      __m256 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 of program

(in Kayvon's fictitious data-parallel language)

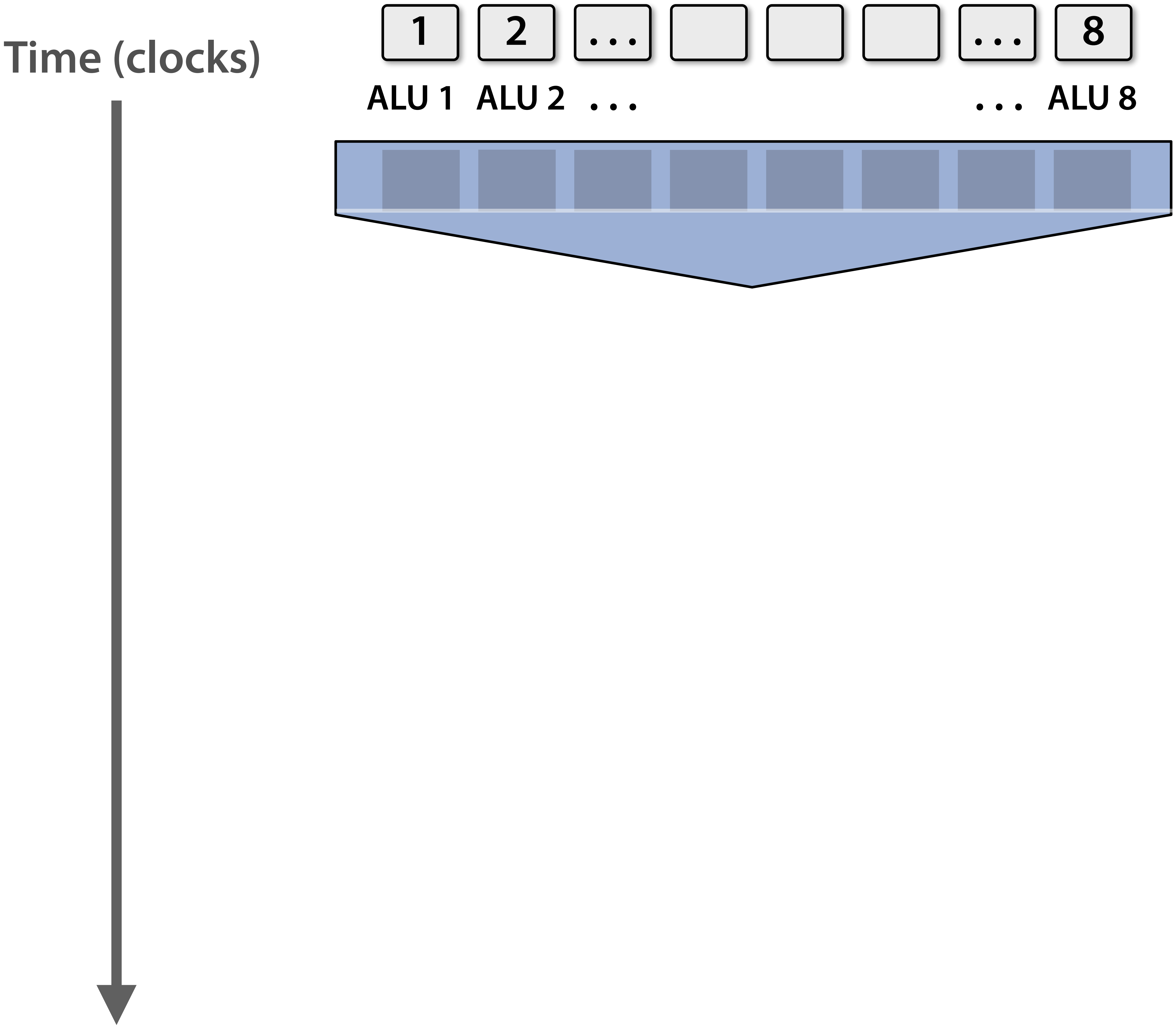
```
void sinx(int N, int terms, float* x, float* result)
{
   // declare independent loop iterations
   forall (int i from 0 to N-1)
   {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      int denom = 6;  // 3!
      int sign = -1;

      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;
   }
}
```

Semantics: loop iterations are "independent"

Q. Why did I say independent and not parallel?

Q. How does this abstraction facilitate automatic generation of both multi-core parallel code, and vector instructions to make use of SIMD processing capabilities within a core?

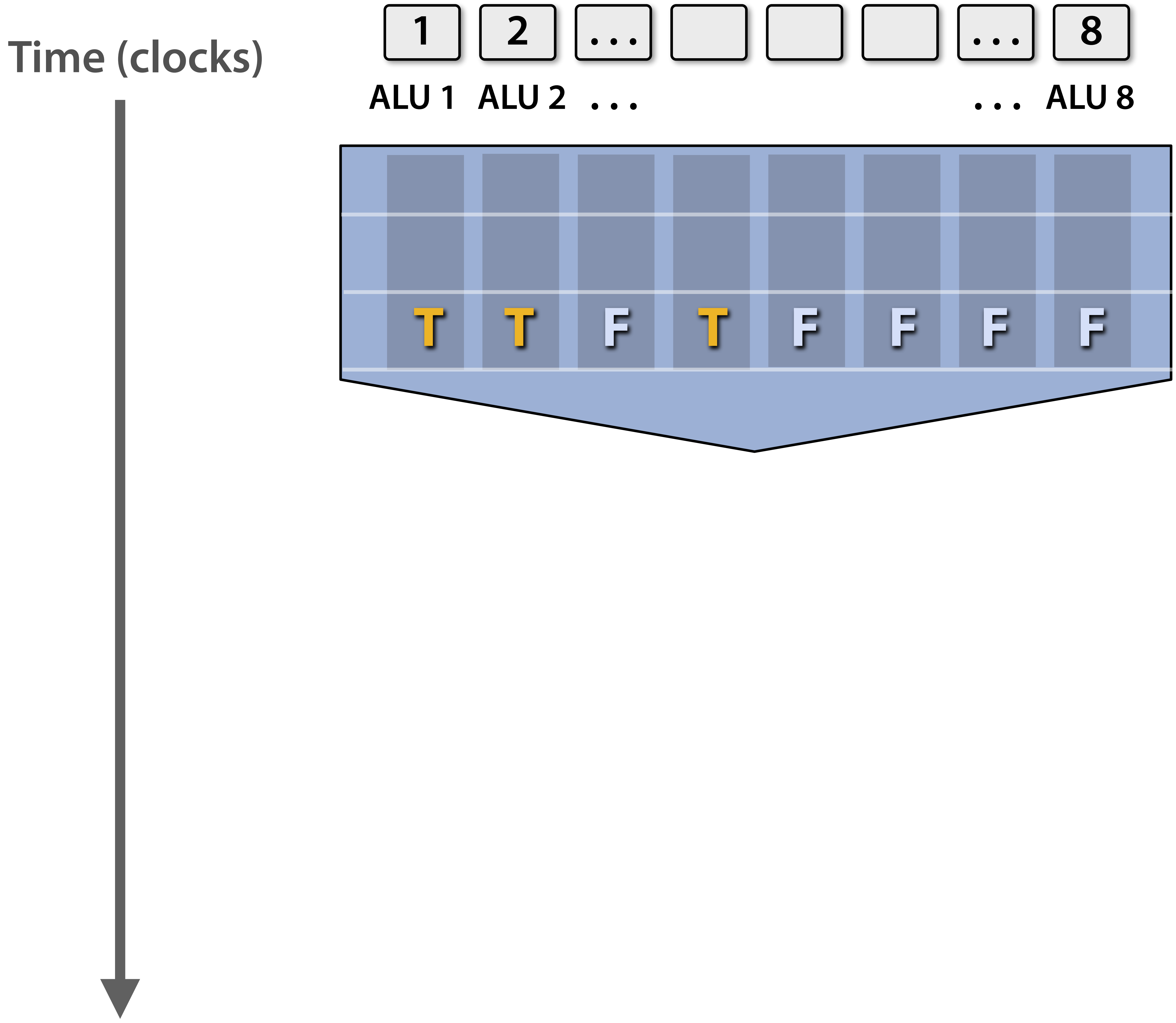
What about conditional execution?



(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;
```

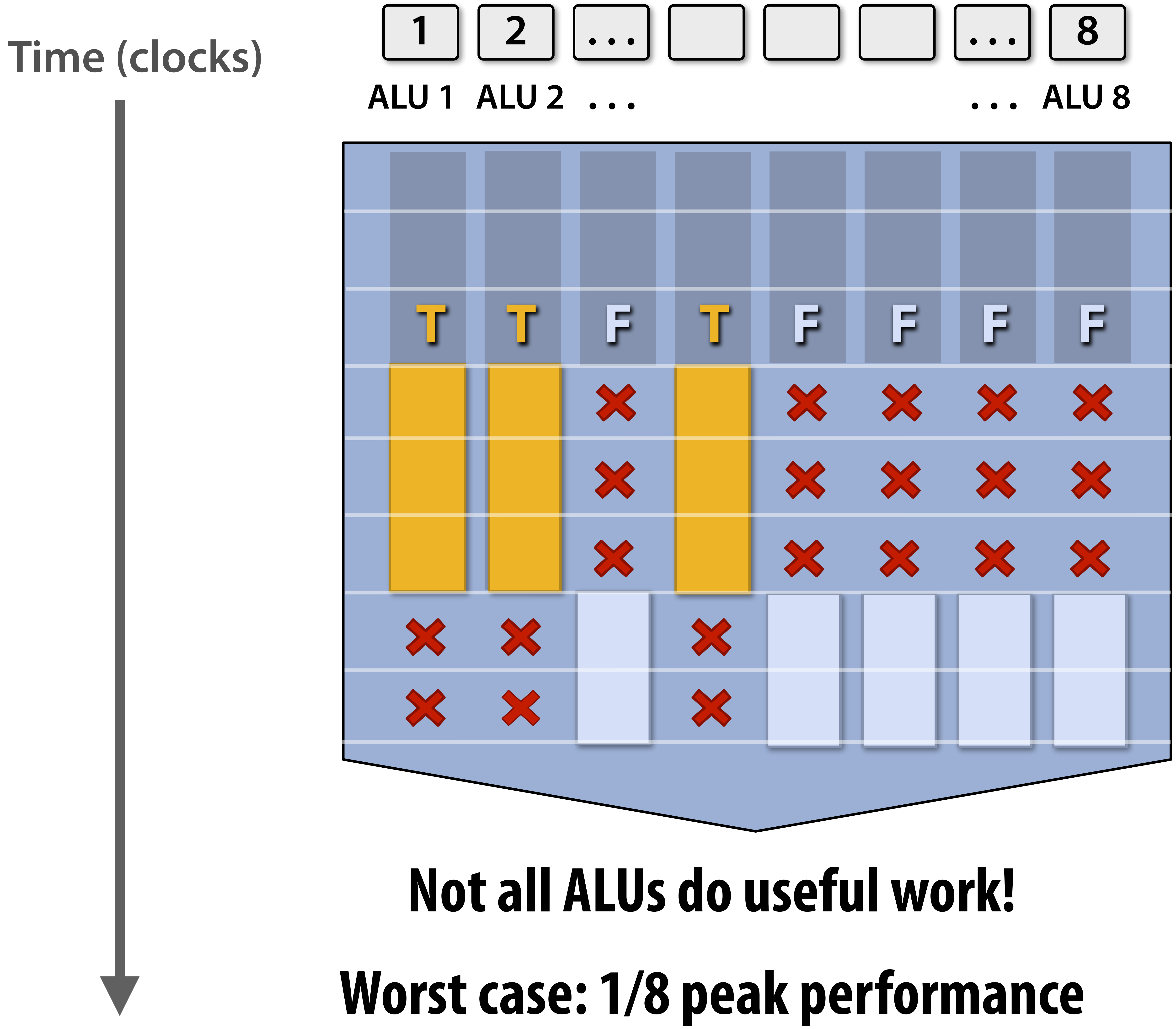
What about conditional execution?



(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



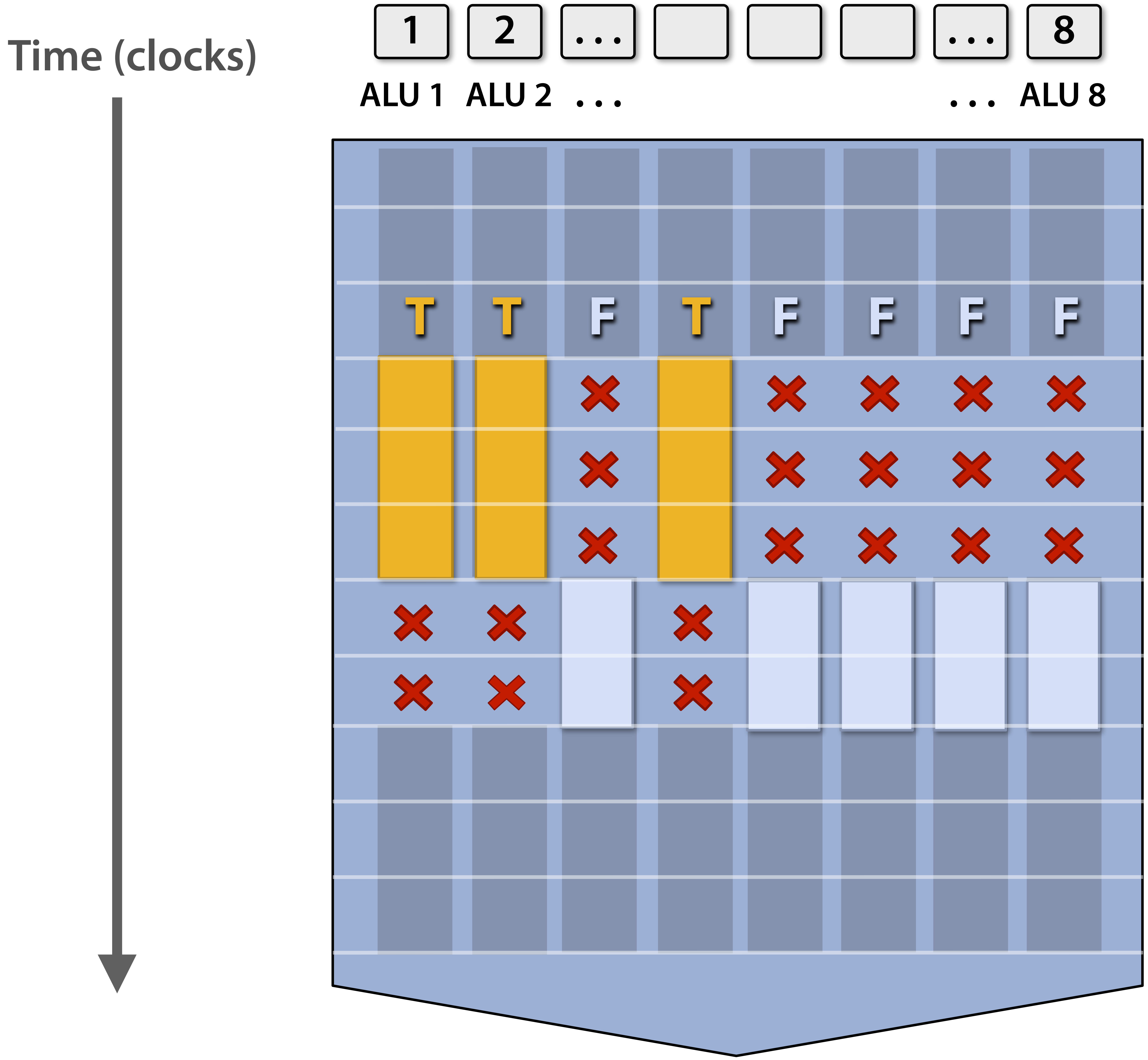
Not all ALUs do useful work!

Worst case: 1/8 peak 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;
```

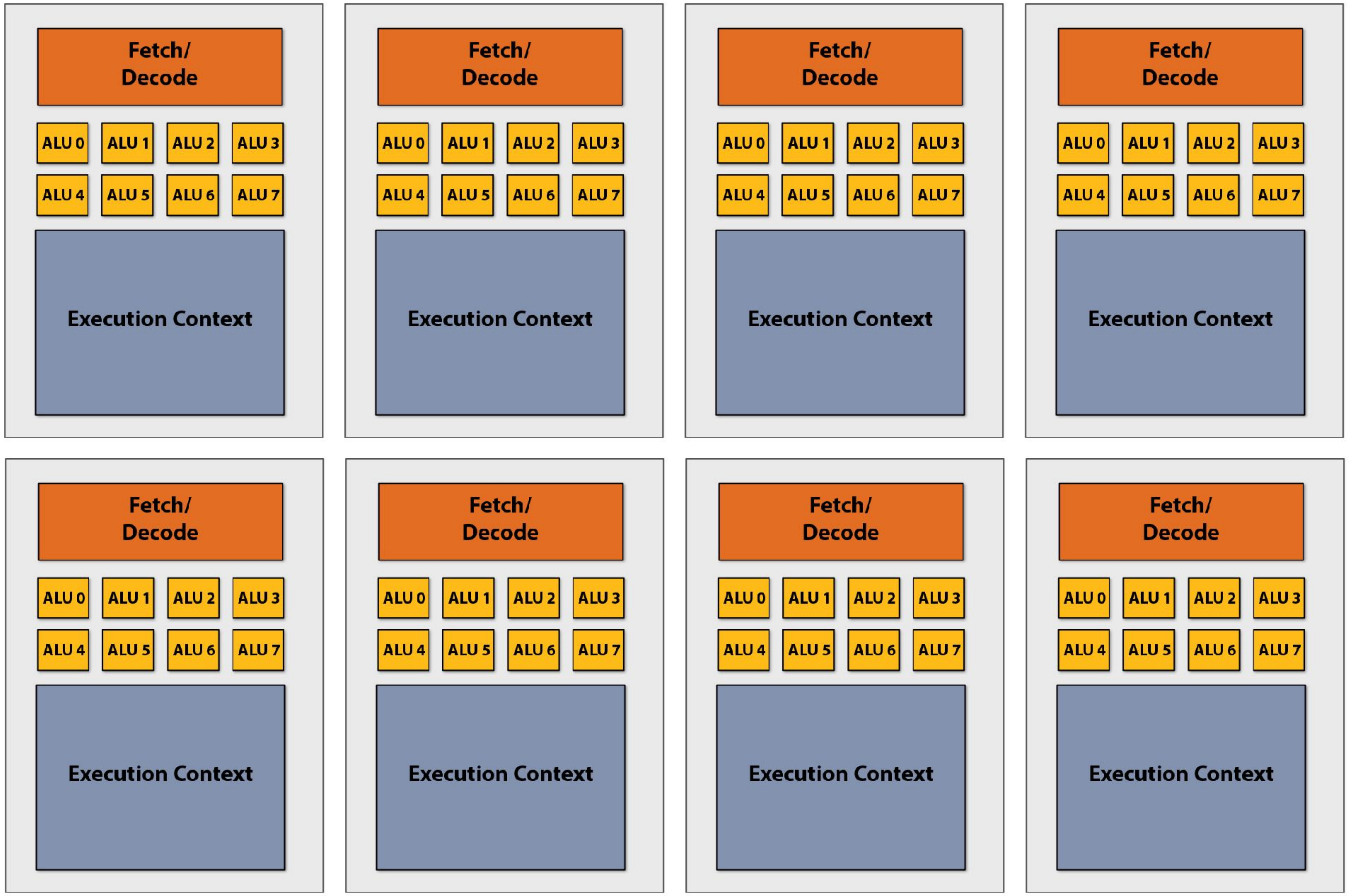
After branch: continue at full performance



(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;
```

Example: eight-core Intel Xeon E5-1660 v4



8 cores

8 SIMD ALUs per core

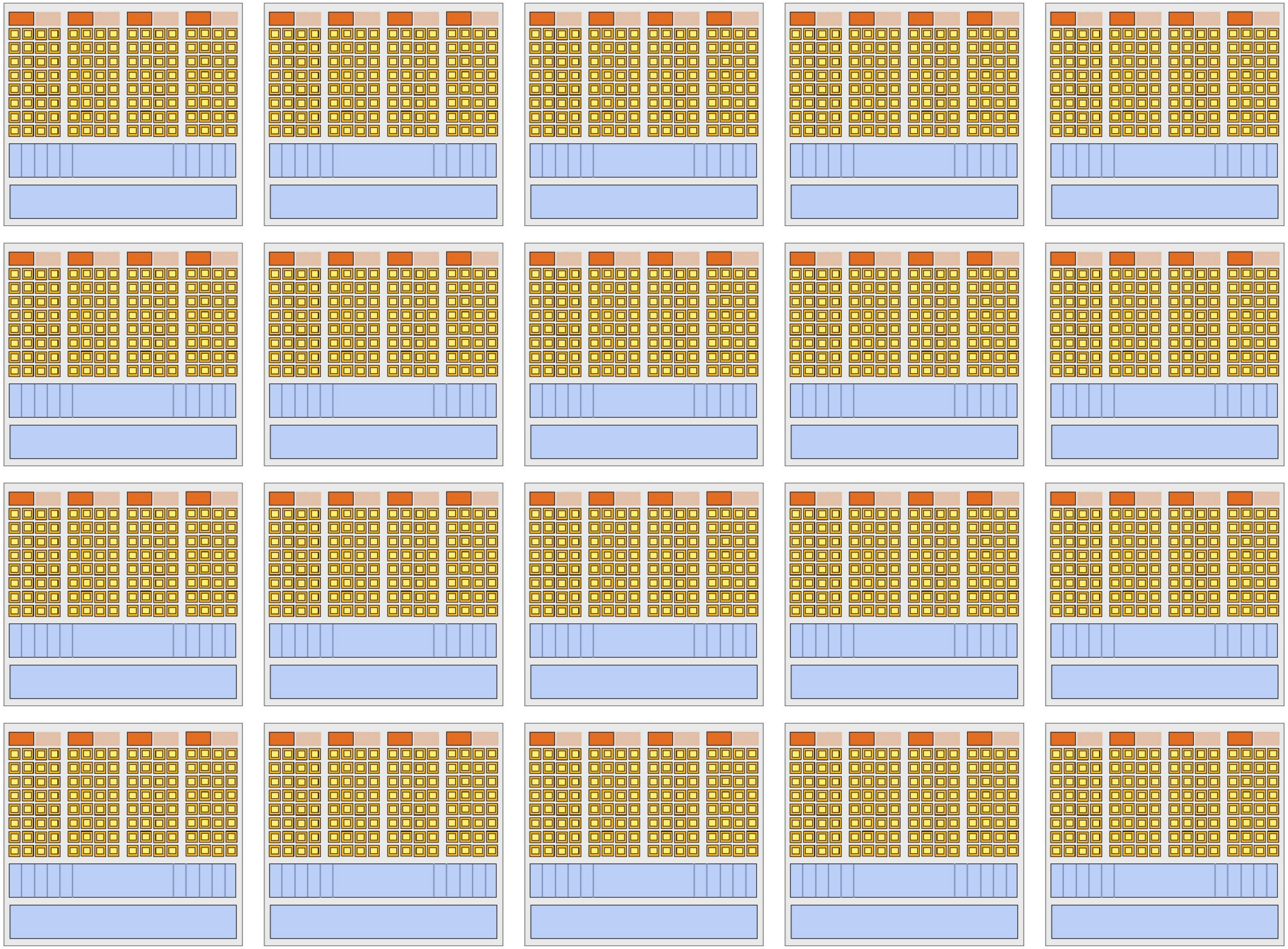
(AVX2 instructions)

490 GFLOPs (@3.2 GHz)

(140 Watts)

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

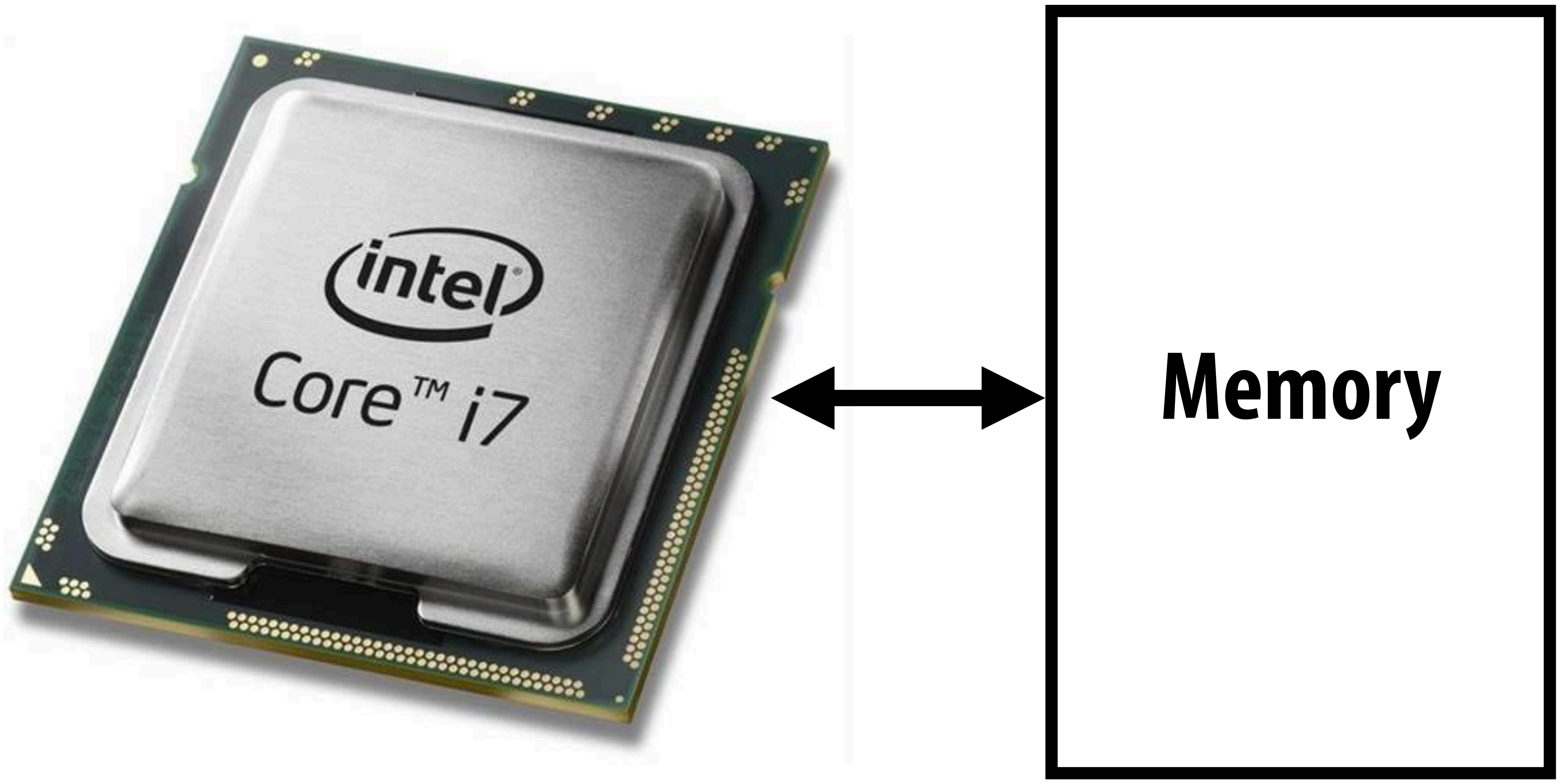
Example: NVIDIA GTX 1080 GPU



20 cores ("SMs")

128 SIMD ALUs per core (@1.6 GHz) = 8.1 TFLOPs (180 Watts)

Part 2:
accessing memory



Memory

Hardware multi-threading

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.

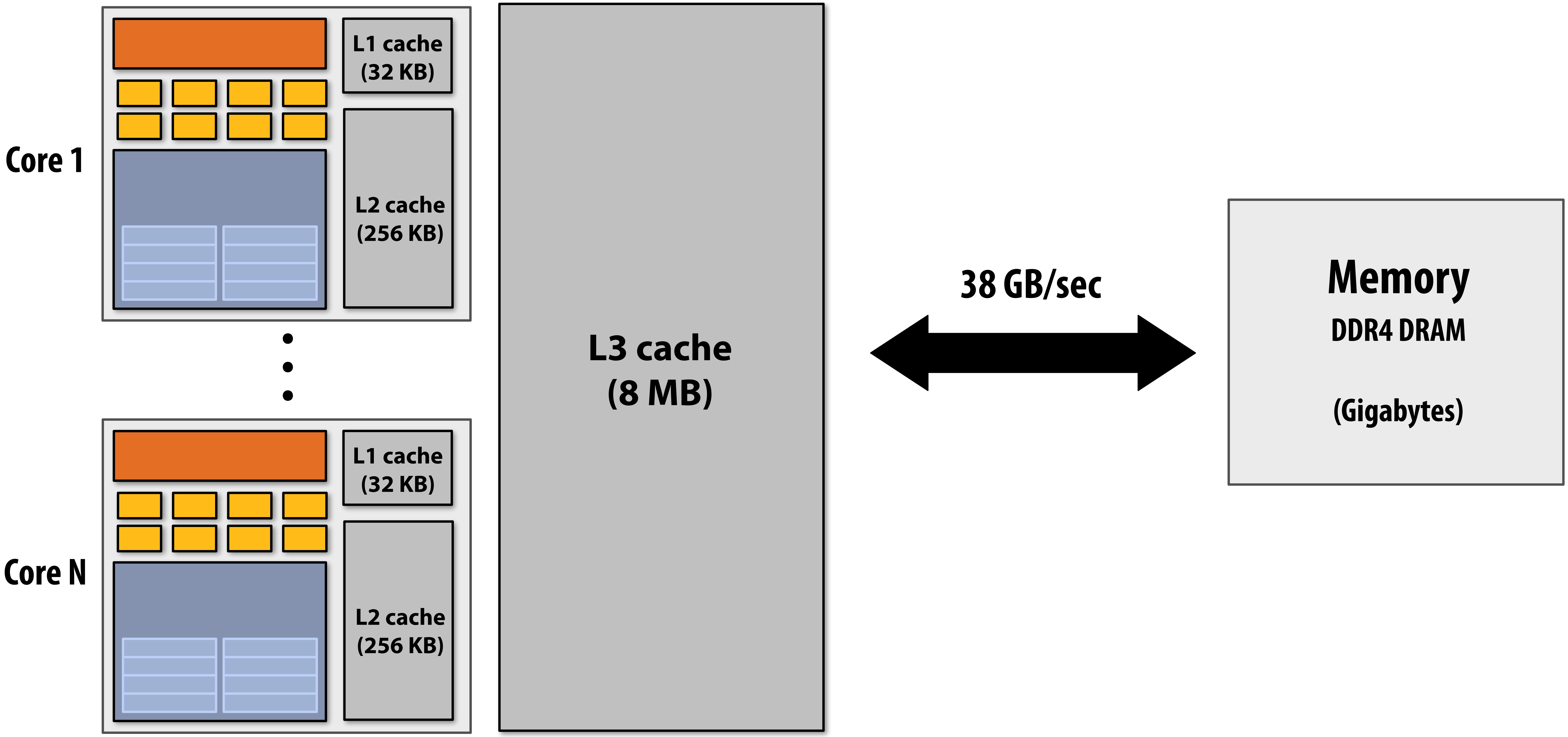
- Accessing memory is a major source of stalls

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

Dependency: cannot execute 'add' instruction until data at mem[r2] and mem[r3] have been loaded from memory

- Memory access times ~ 100's of cycles
  - Memory "access time" is a measure of latency

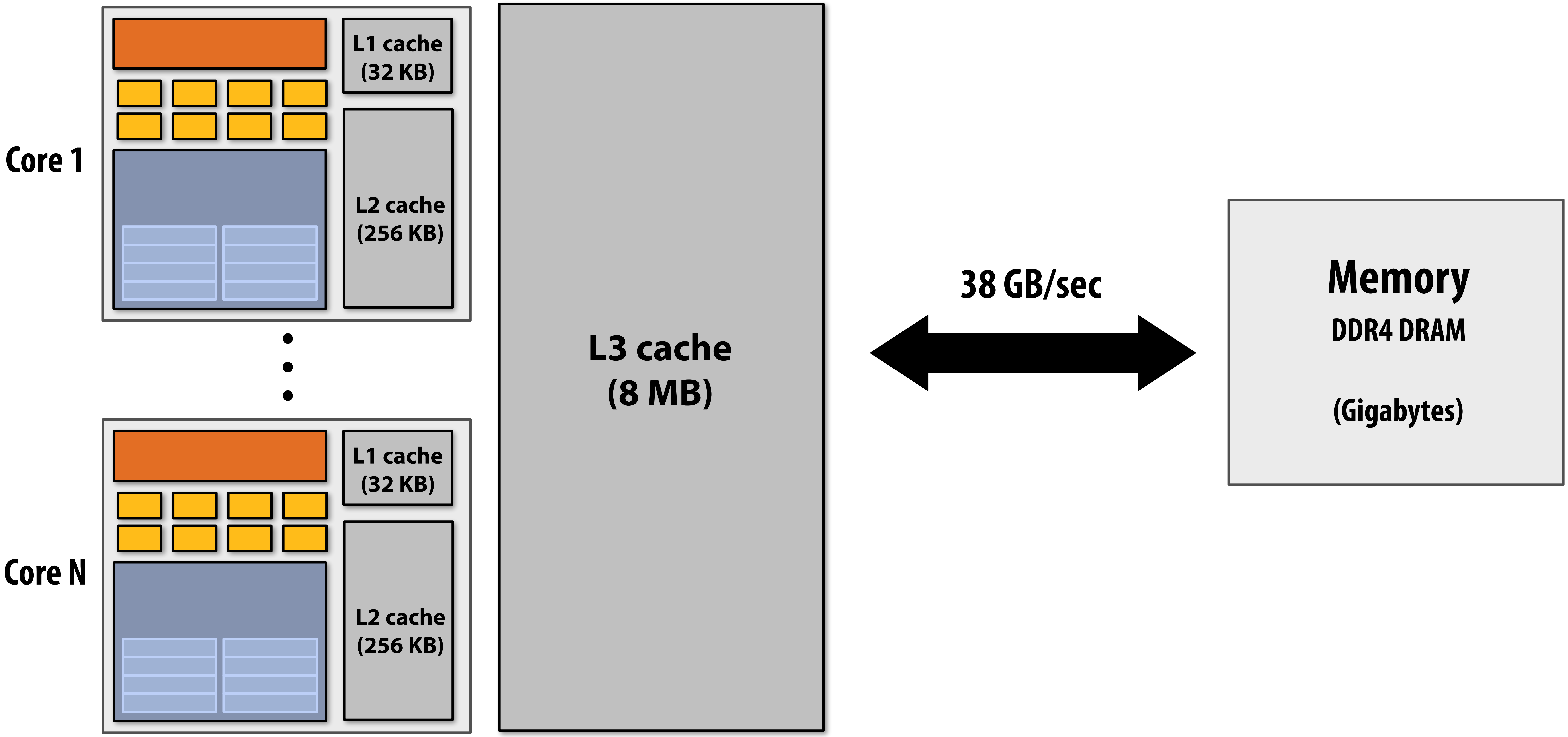
Review: why do modern processors have caches?



Caches reduce length of stalls (reduce latency)

Processors run efficiently when data is resident in caches

Caches reduce memory access latency *

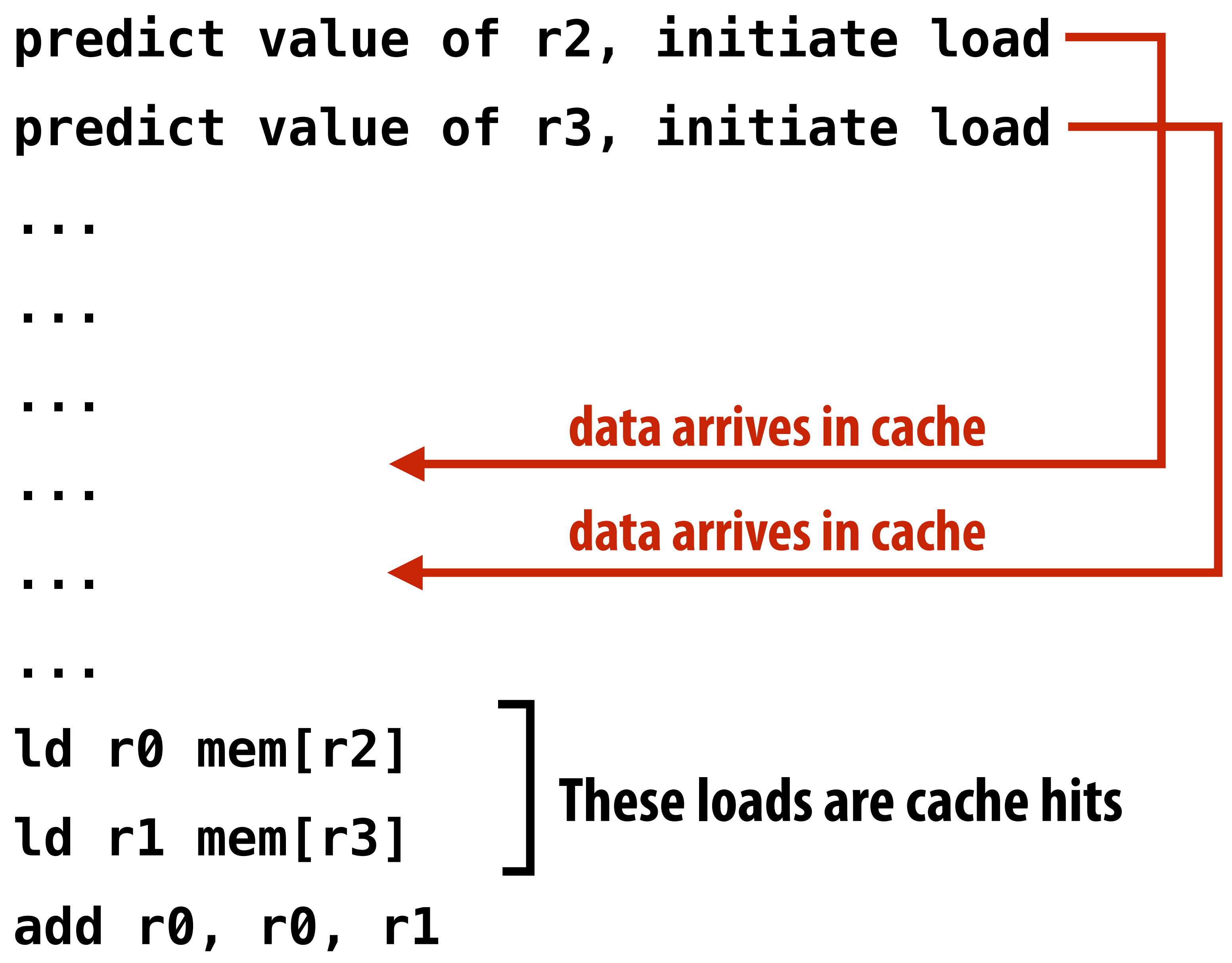


* Caches also provide high bandwidth data transfer to CPU

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

data arrives in cache

data arrives in cache

These loads are cache hits

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

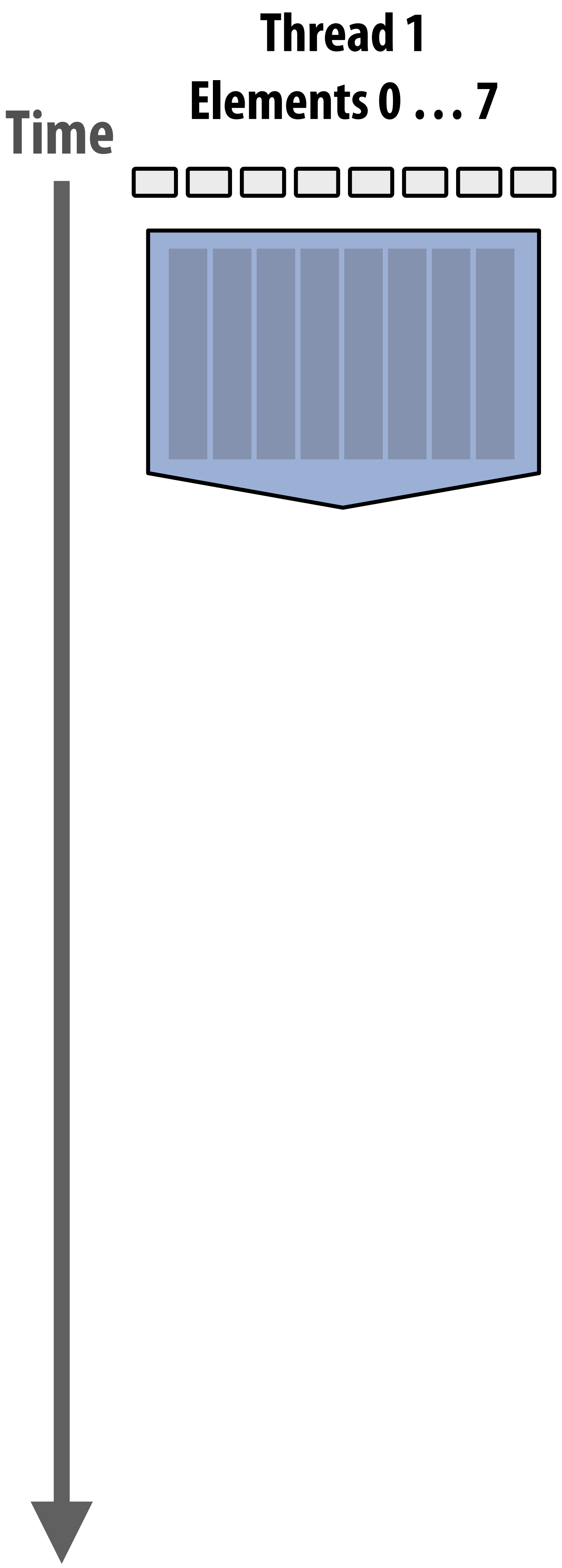
(more detail later in course)

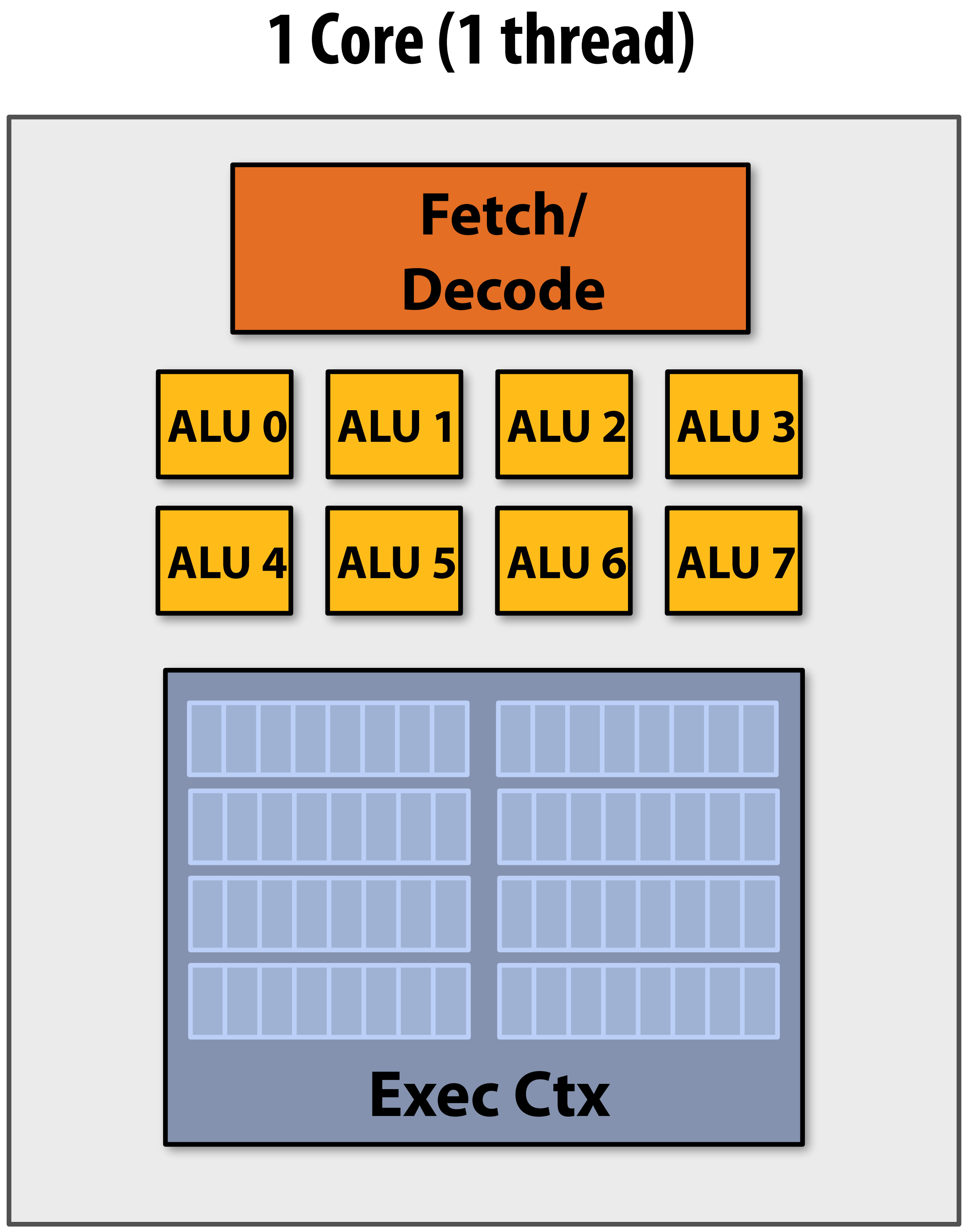
Multi-threading reduces stalls

- Idea: <u>interleave</u> processing of multiple threads on the same core to hide stalls

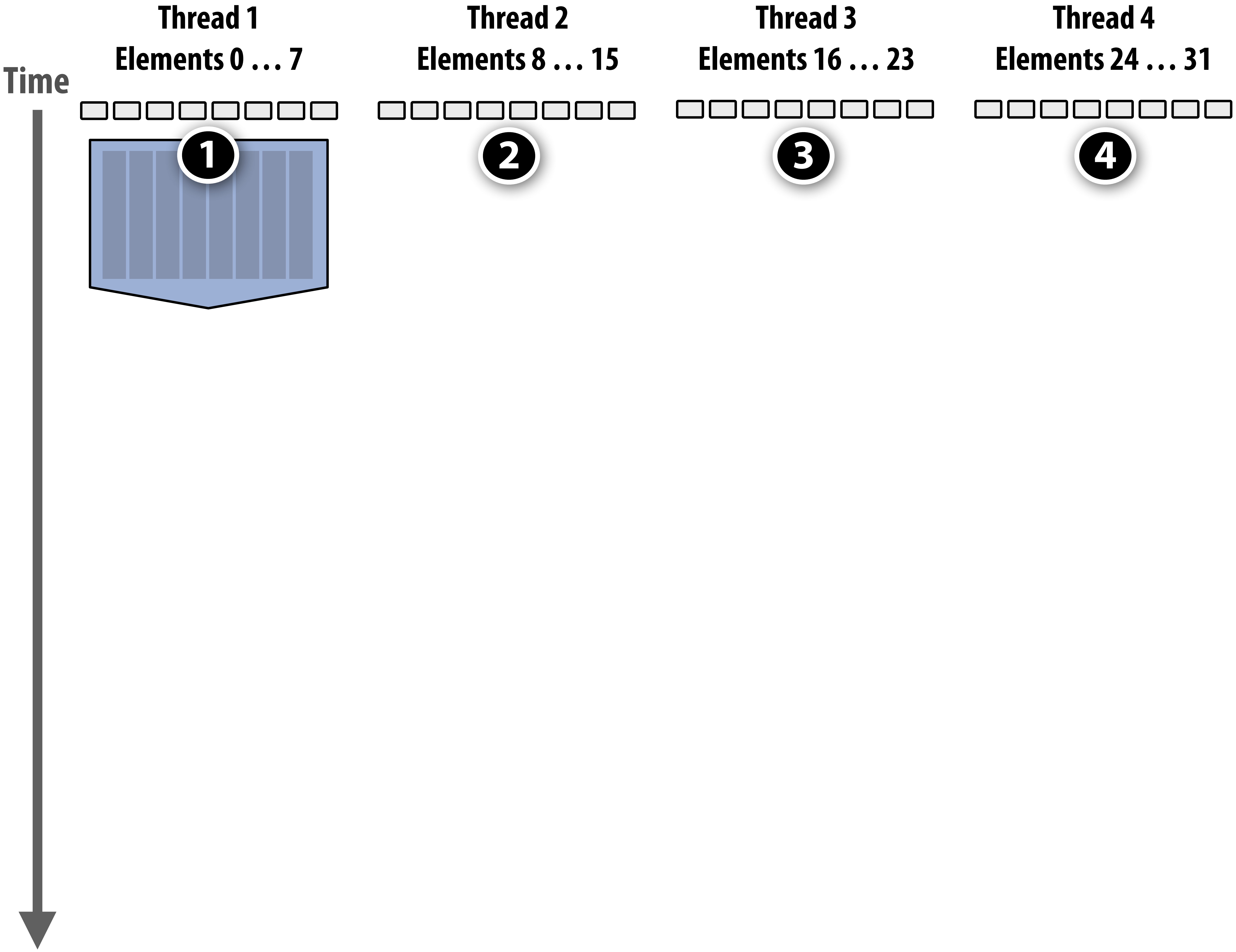
- Like prefetching, multi-threading is a latency <u>hiding</u>, not a latency <u>reducing</u> technique

Hiding stalls with multi-threading





Hiding stalls with multi-threading



Time

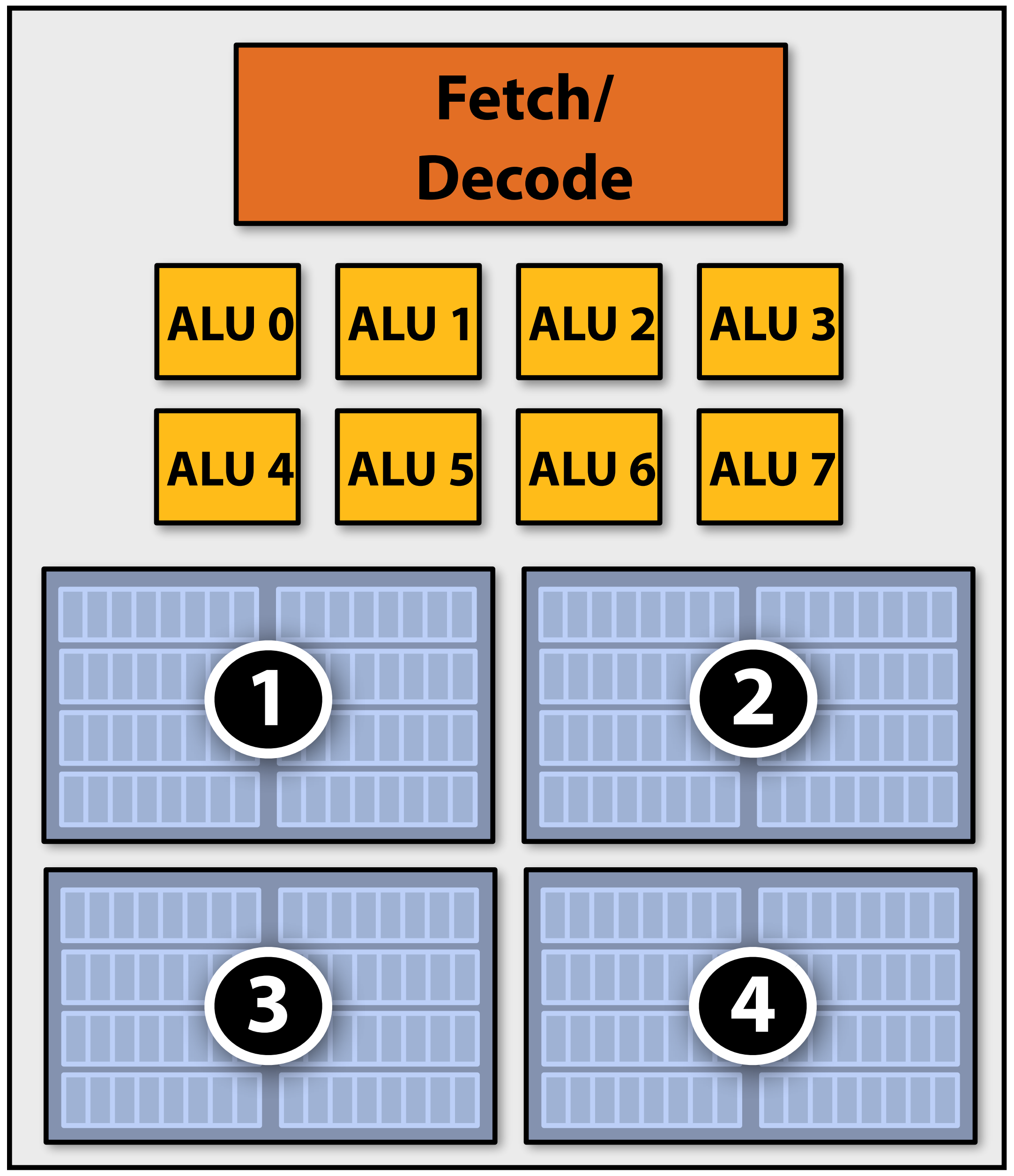
Thread 1
Elements 0 … 7

Thread 2
Elements 8 … 15

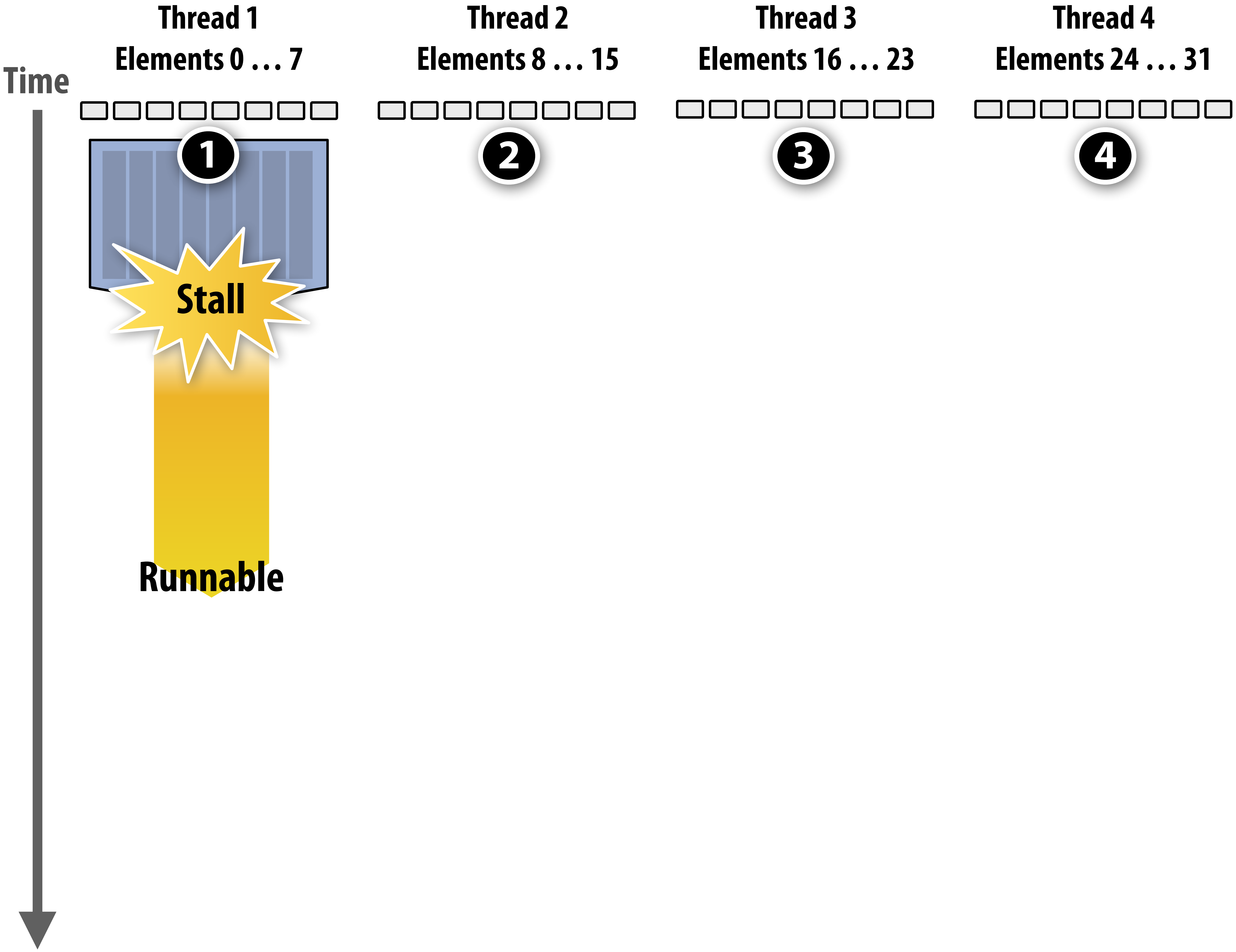
Thread 3
Elements 16 … 23

Thread 4
Elements 24 … 31

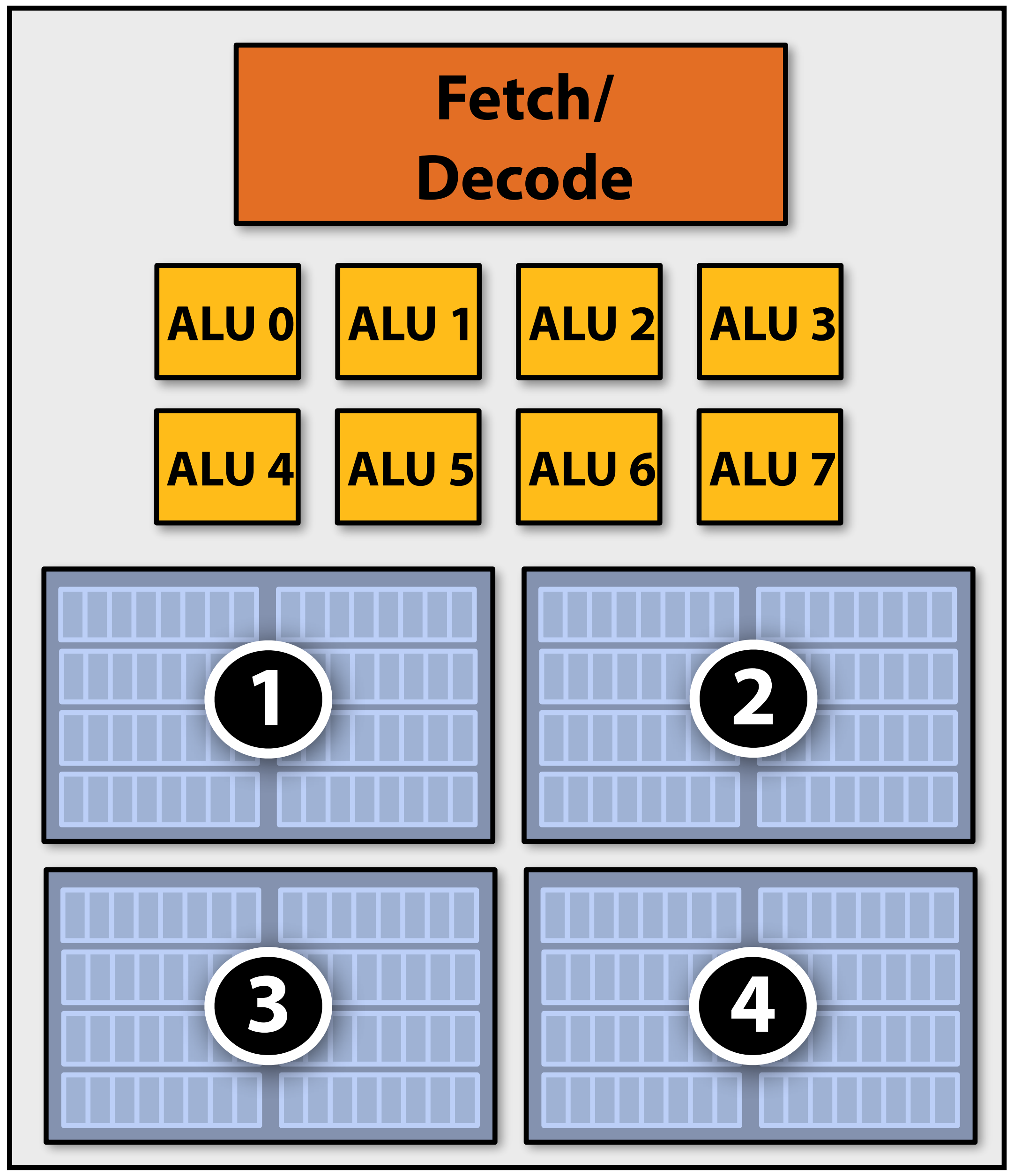
1 Core (4 hardware threads)



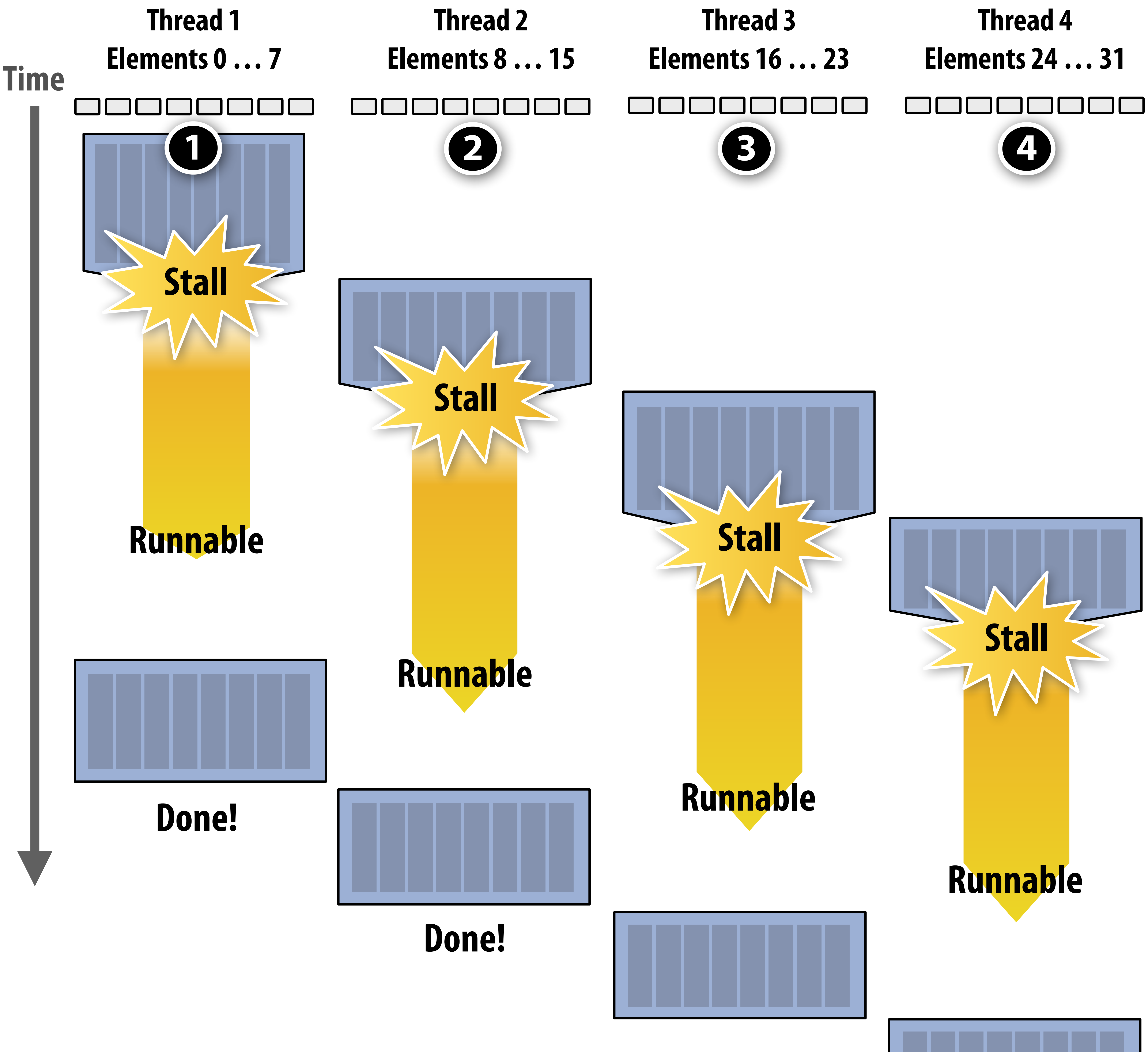
Hiding stalls with multi-threading

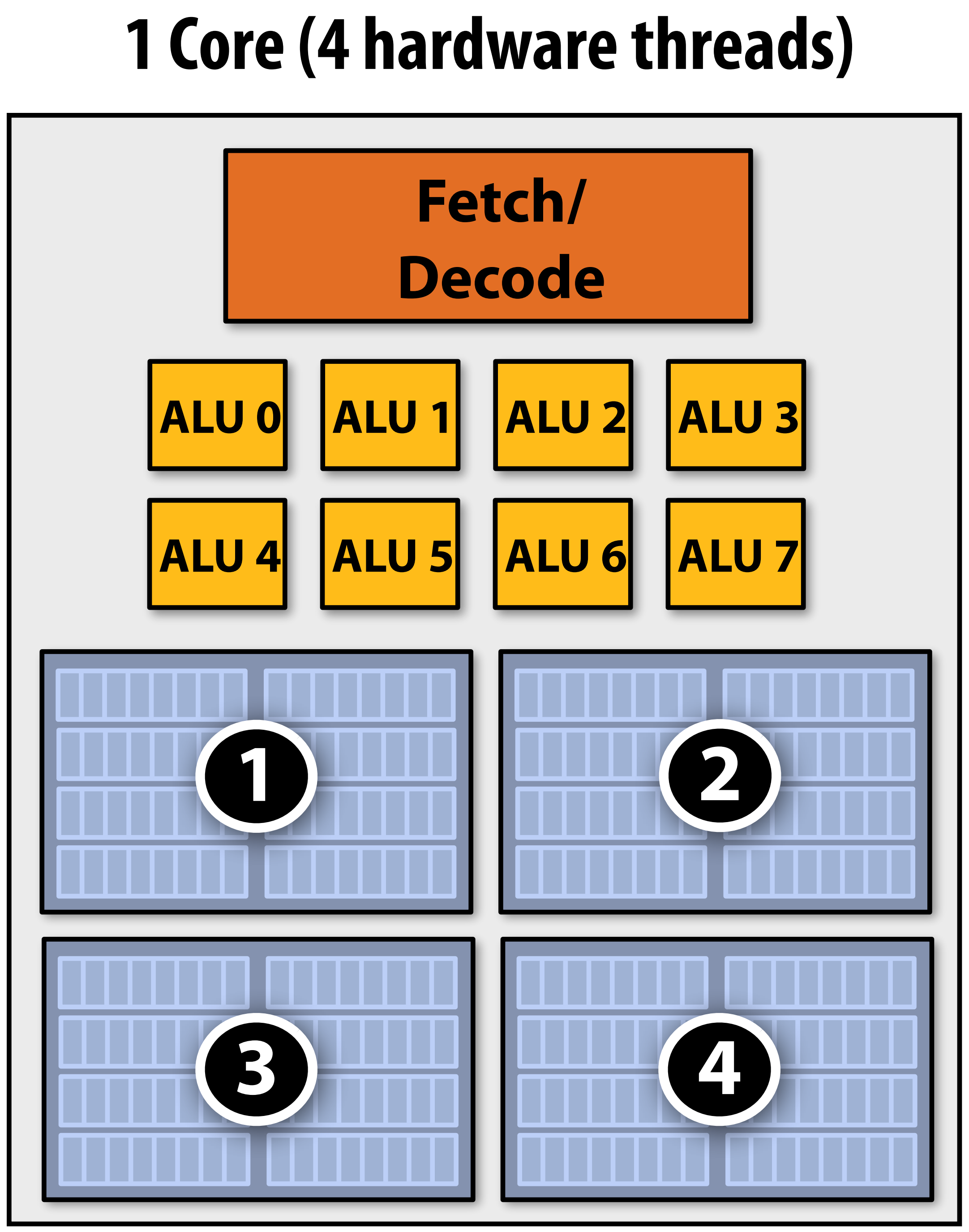


1 Core (4 hardware threads)

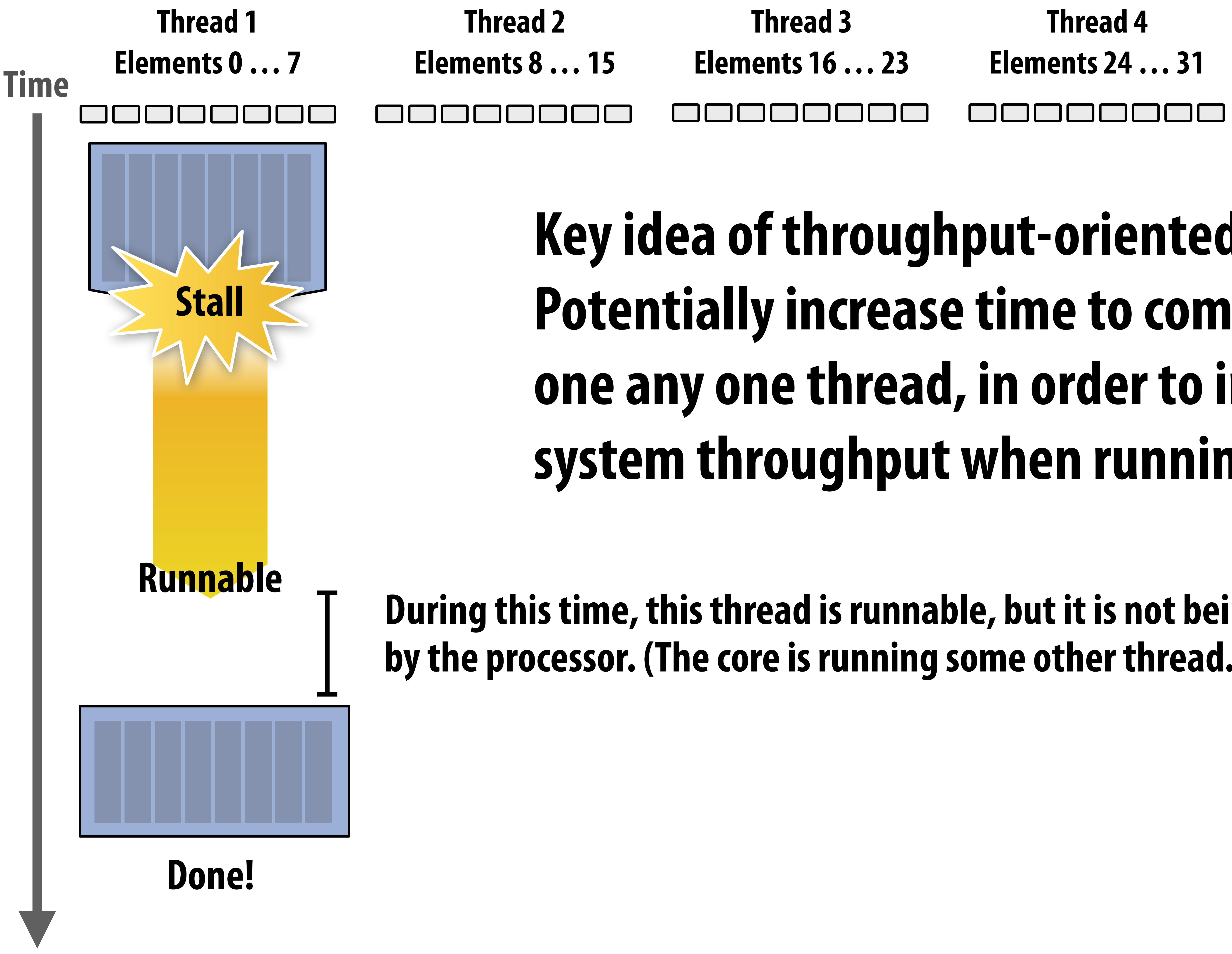


Hiding stalls with multi-threading





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.

During this time, this thread is runnable, but it is not being executed by the processor. (The core is running some other thread.)

Kayvon's fictitious multi-core chip

16 cores

8 SIMD ALUs per core

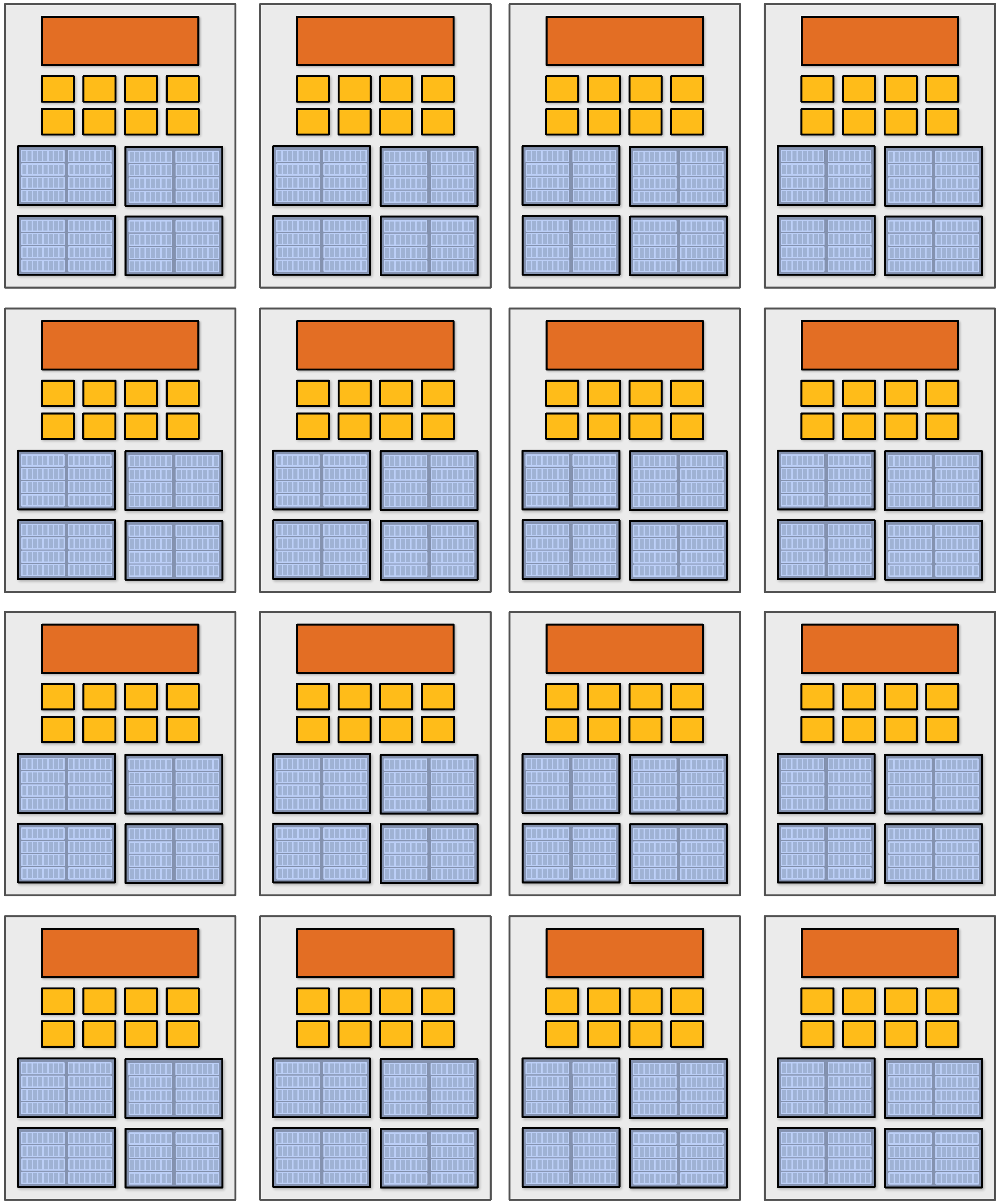
(128 total)

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

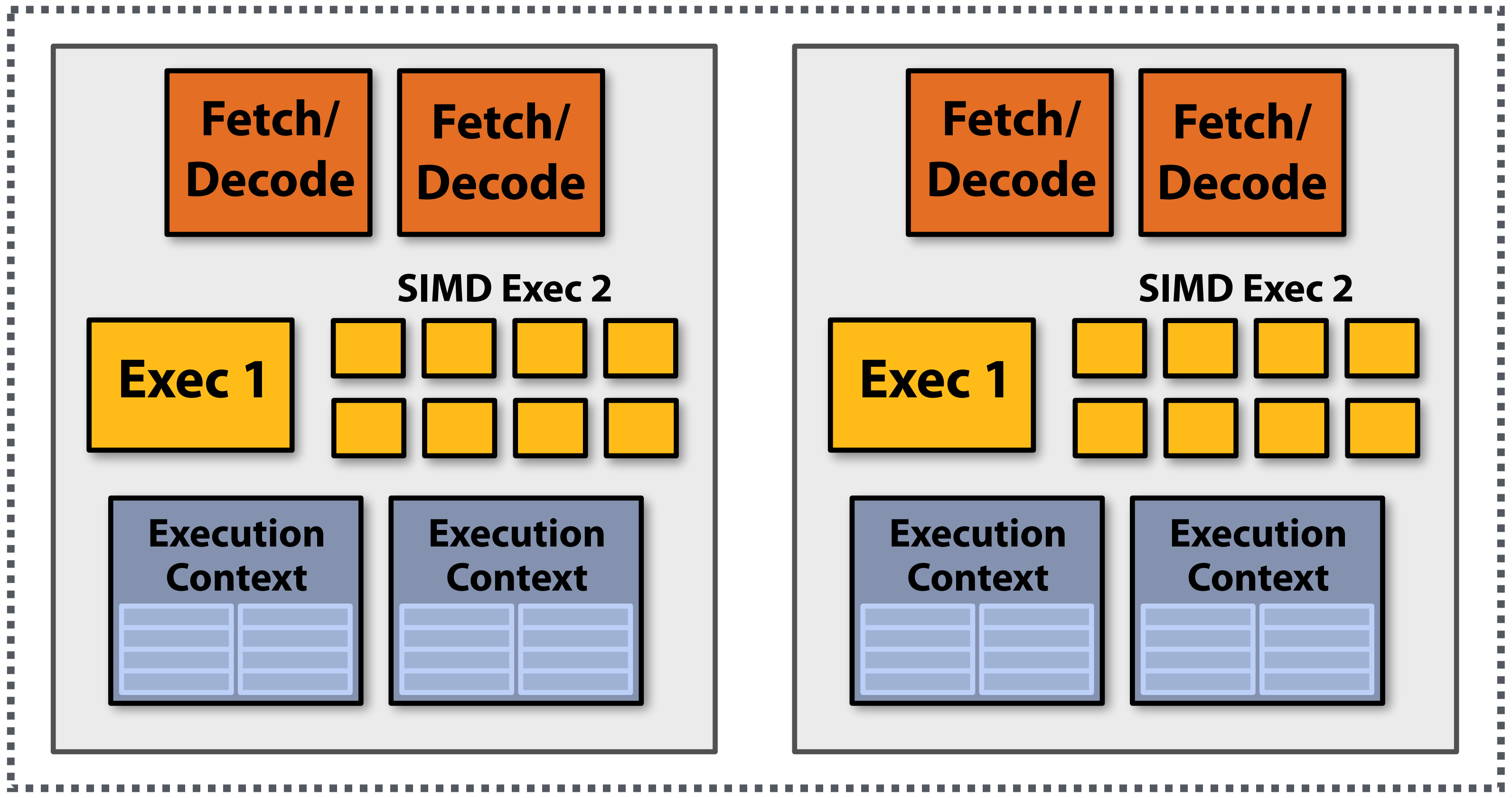


Thought experiment

- You write a C application that spawns two pthreads
- 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 your pthreads to the processor's thread execution contexts?

  Answer: the operating system

- Question: If you were the OS, how would to assign the two threads to the four available execution contexts?
- Another question: How would you assign threads to execution contexts if your C program spawned five pthreads?

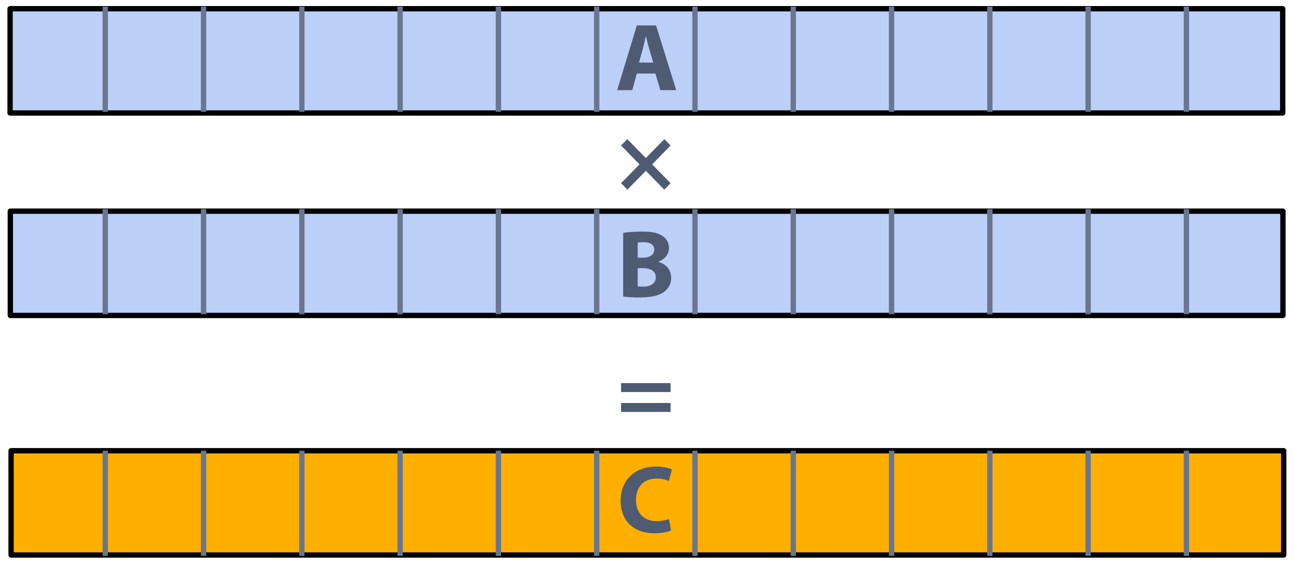


Another thought experiment

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]



Three memory operations (12 bytes) for every MUL

NVIDIA GTX 1080 GPU can do 2560 MULs per clock (@ 1.6 GHz)

Need ~50 TB/sec of bandwidth to keep functional units busy (only have 320 GB/sec)

<1% GPU efficiency… but 4.2x faster than eight-core CPU!

(3.2 GHz Xeon E5v4 eight-core CPU connected to 76 GB/sec memory bus will exhibit ~3% efficiency on this computation)

Bandwidth limited!

Bandwidth limited!

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

No amount of latency hiding helps this.

Bandwidth is a critical resource

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

Which program performs better?

Program 1

```
void add(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
       C[i] = A[i] + B[i];
}

void mul(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
       C[i] = A[i] * B[i];
}


float* A, *B, *C, *D, *E, *tmp1, *tmp2;

// assume arrays are allocated here

// compute E = D + ((A + B) * C)
add(n, A, B, tmp1);
mul(n, tmp1, C, tmp2);
add(n, tmp2, D, E);
```

(Note: an answer probably needs to state its assumptions.)

Which code structuring style would you rather write?

Program 2

```
void fused(int n, float* A, float* B, float* C, float* D, float* E) {
    for (int i=0; i<n; i++)
       E[i] = D[i] + (A[i] + B[i]) * C[i];
}

// compute E = D + (A + B) * C
fused(n, A, B, C, D, E);
```

How it all fits together:

superscalar,
SIMD,
multi-threading,
and multi-core

Running code on a simple processor

C program:
compute $\sin(x)$ using Taylor expansion

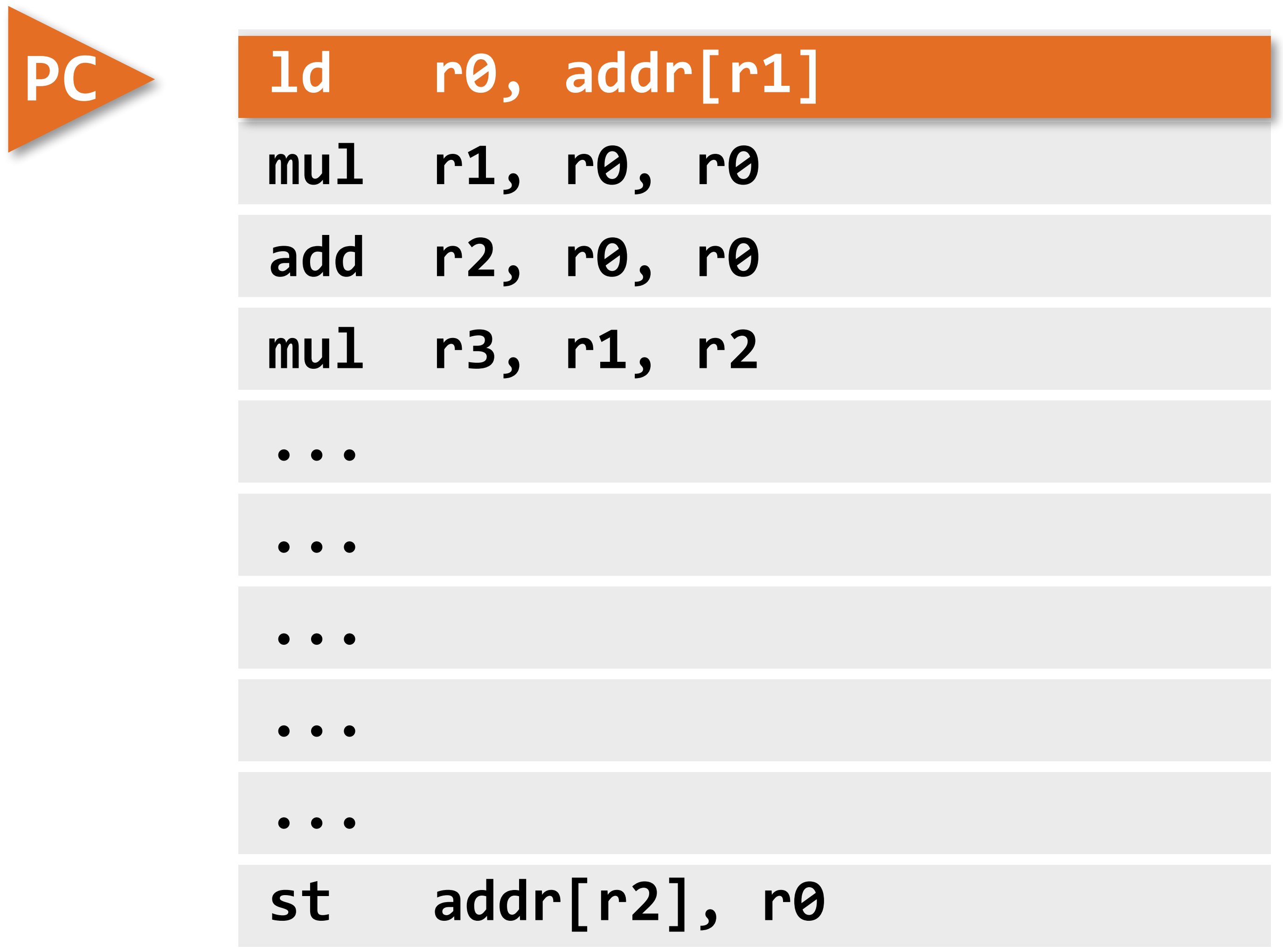
```c
void sinx(int N, int terms, float* x, float* result)
{
   for (int i=0; i<N; i++)
   {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      int denom = 6;  // 3!
      int sign = -1;

      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;
   }
}
```

Compiled instruction stream (scalar instructions)

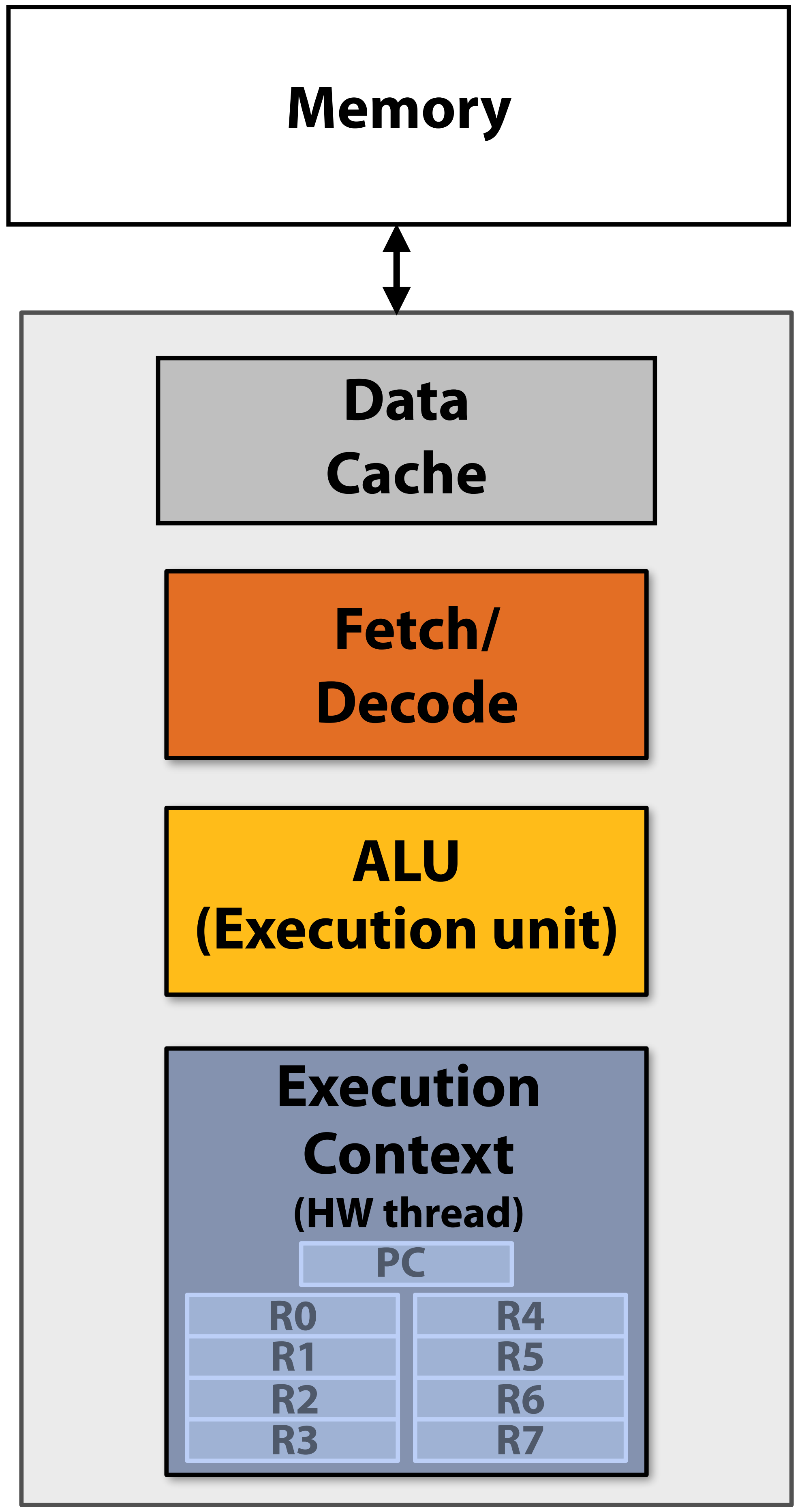
```
ld   r0, addr[r1]
mul  r1, r0, r0
add  r2, r0, r0
mul  r3, r1, r2
...
...
...
...
...
st   addr[r2], r0
```

Running code on a simple processor

Instruction stream



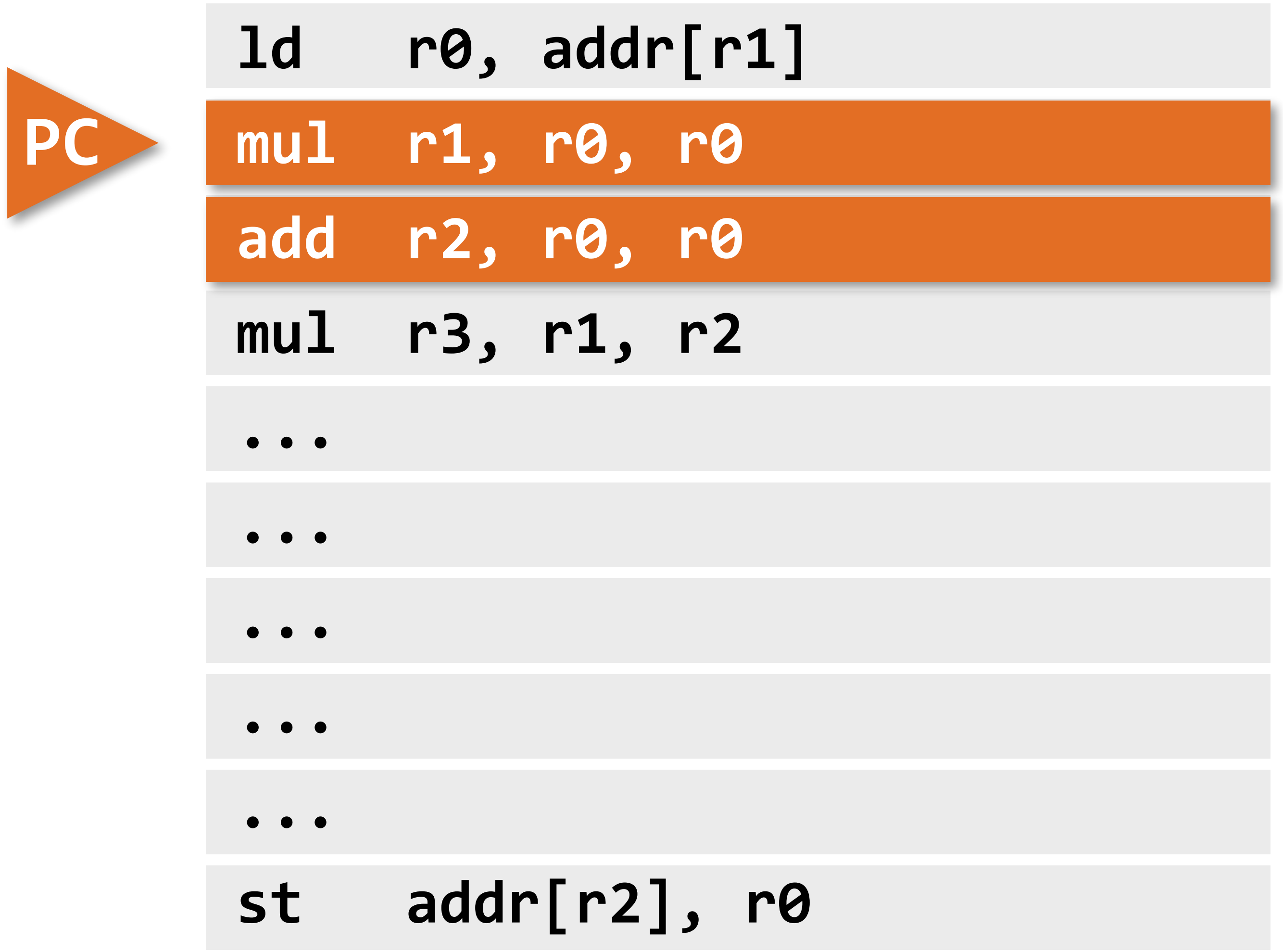
```
ld    r0, addr[r1]
mul   r1, r0, r0
add   r2, r0, r0
mul   r3, r1, r2
...
...
...
...
...
st    addr[r2], r0
```



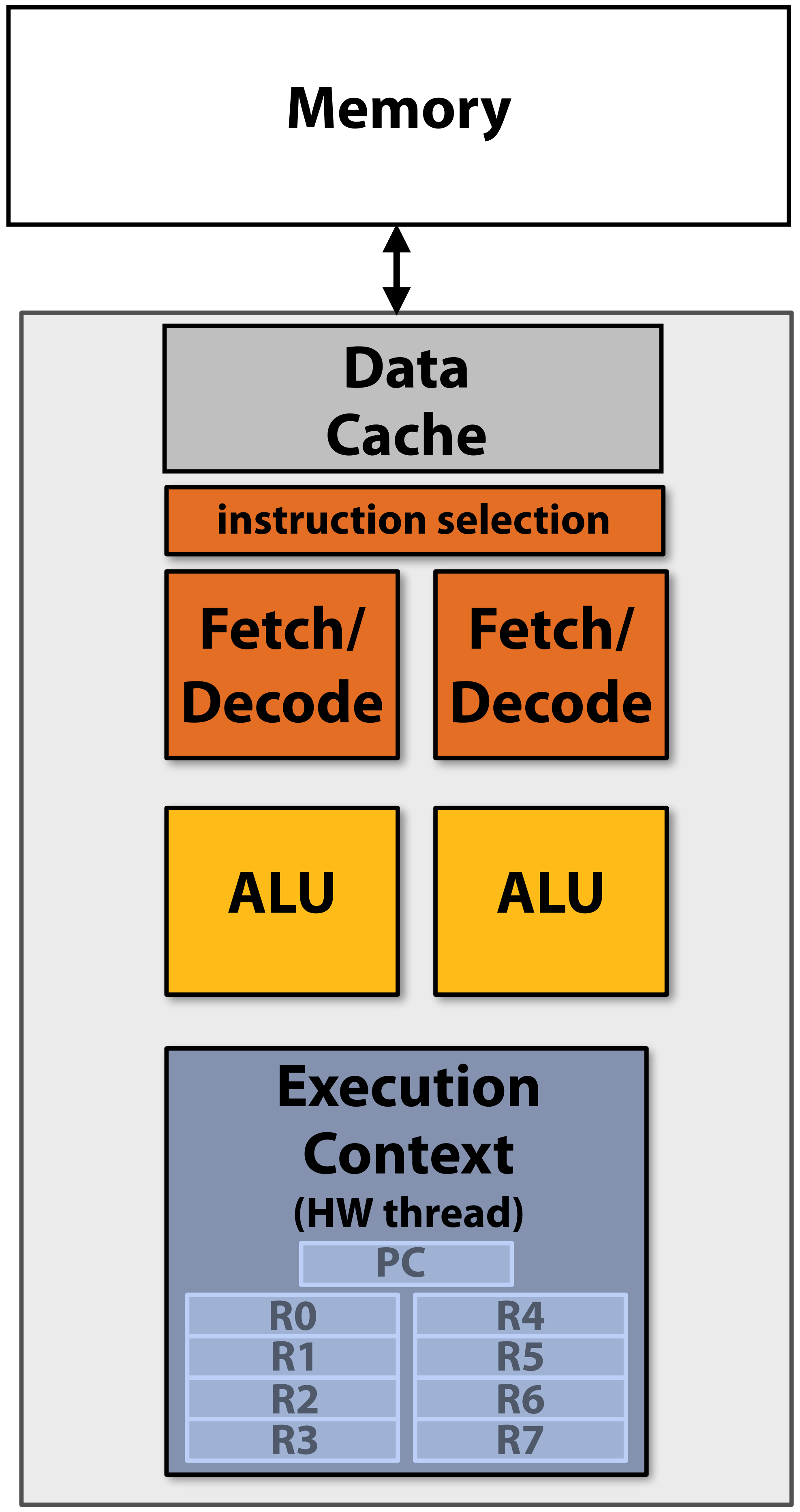
Single core processor, single-threaded core.
Can run one scalar instruction per clock

Superscalar core

Instruction stream



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



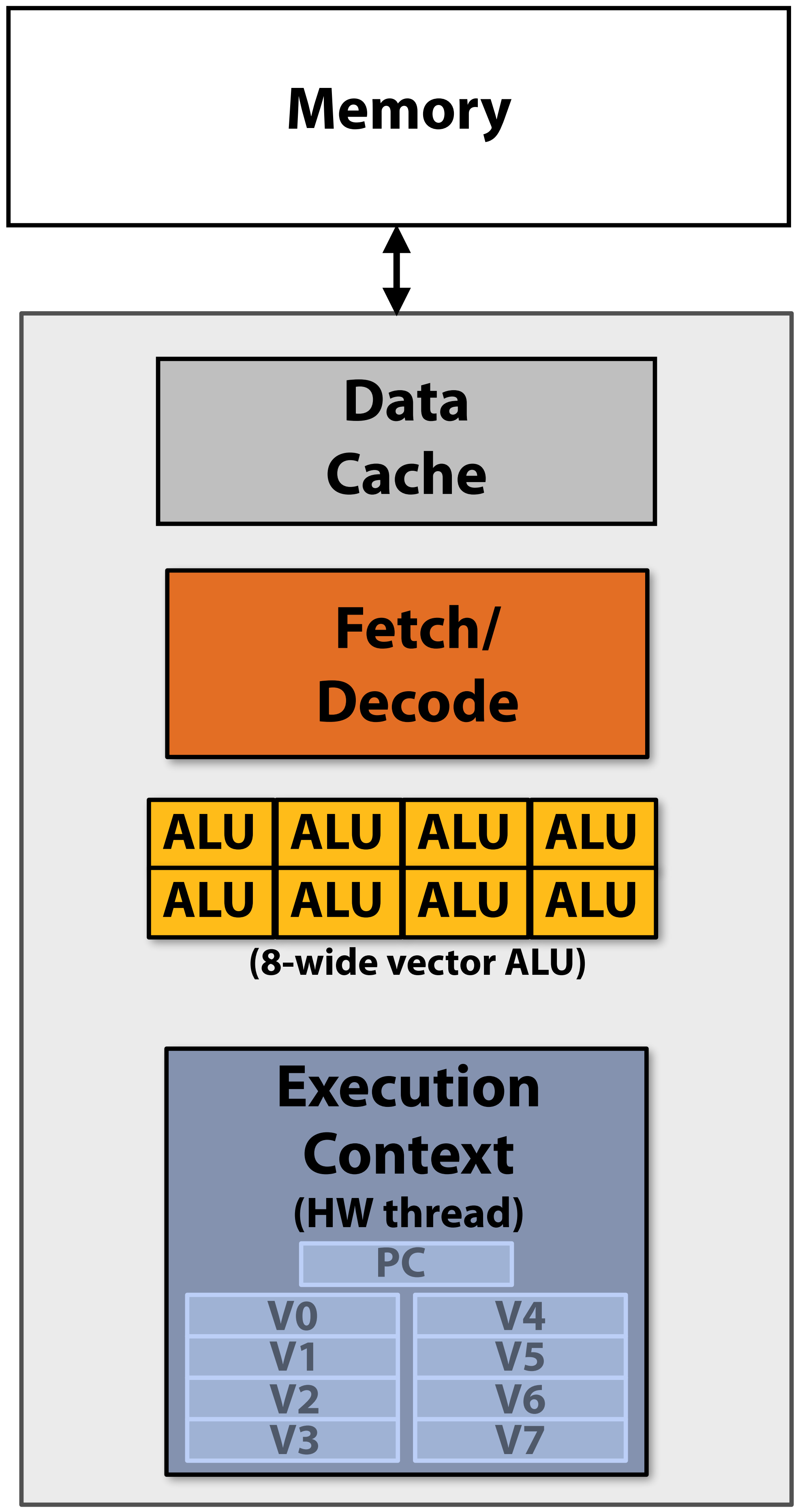
Single core processor, single-threaded core.
Two-way superscalar core:
can run up to two independent scalar instructions
per clock from one instruction stream (one hardware thread)

SIMD execution capability

Instruction stream
(now with vector instructions)

PC

```
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
```

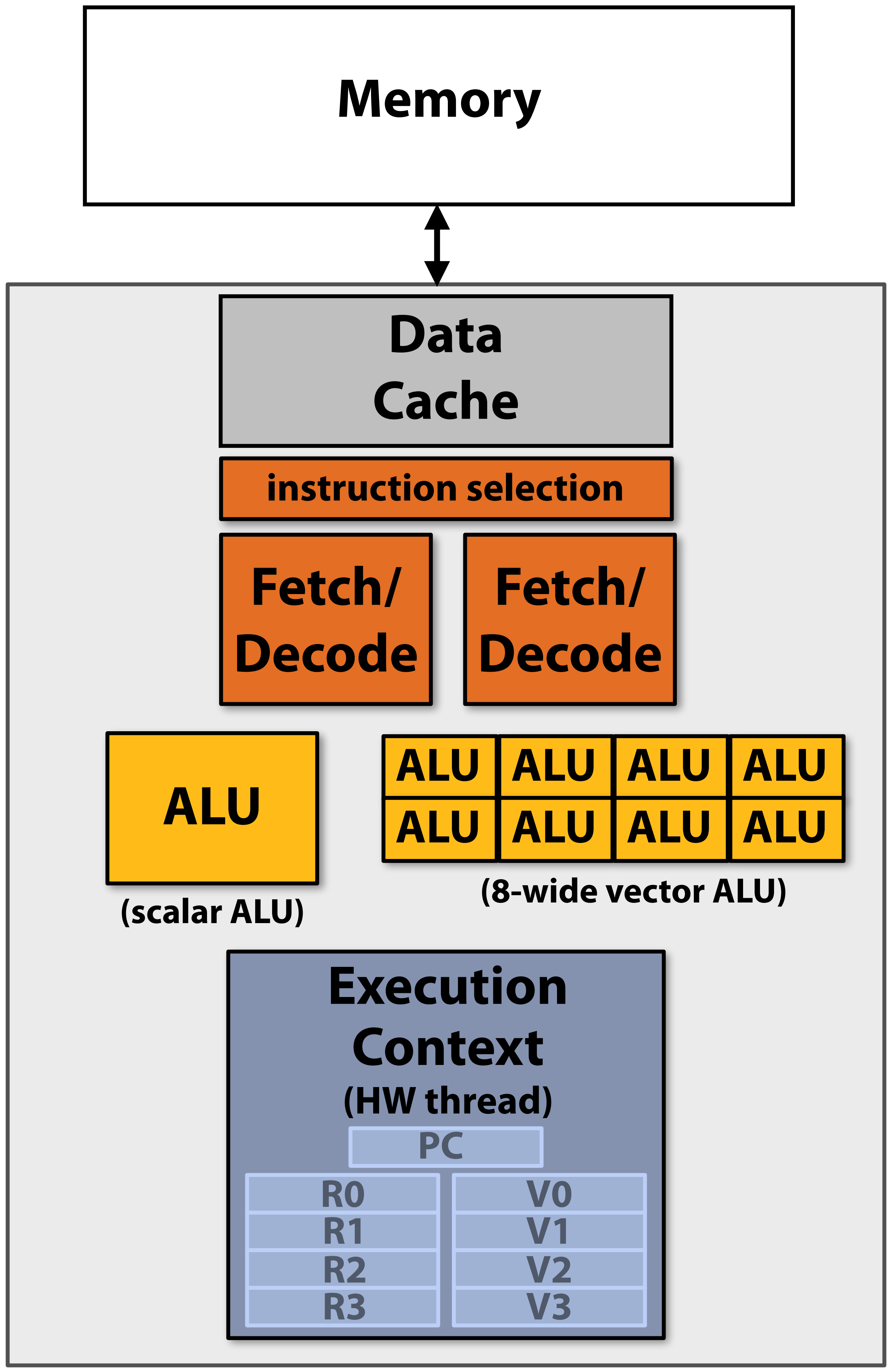


Single core processor, single-threaded core.
can run one 8-wide SIMD vector instruction from one instruction stream

Heterogeneous superscalar (scalar + SIMD)

Instruction stream

```
    vector_ld    v0, vector_addr[r1]
PC  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

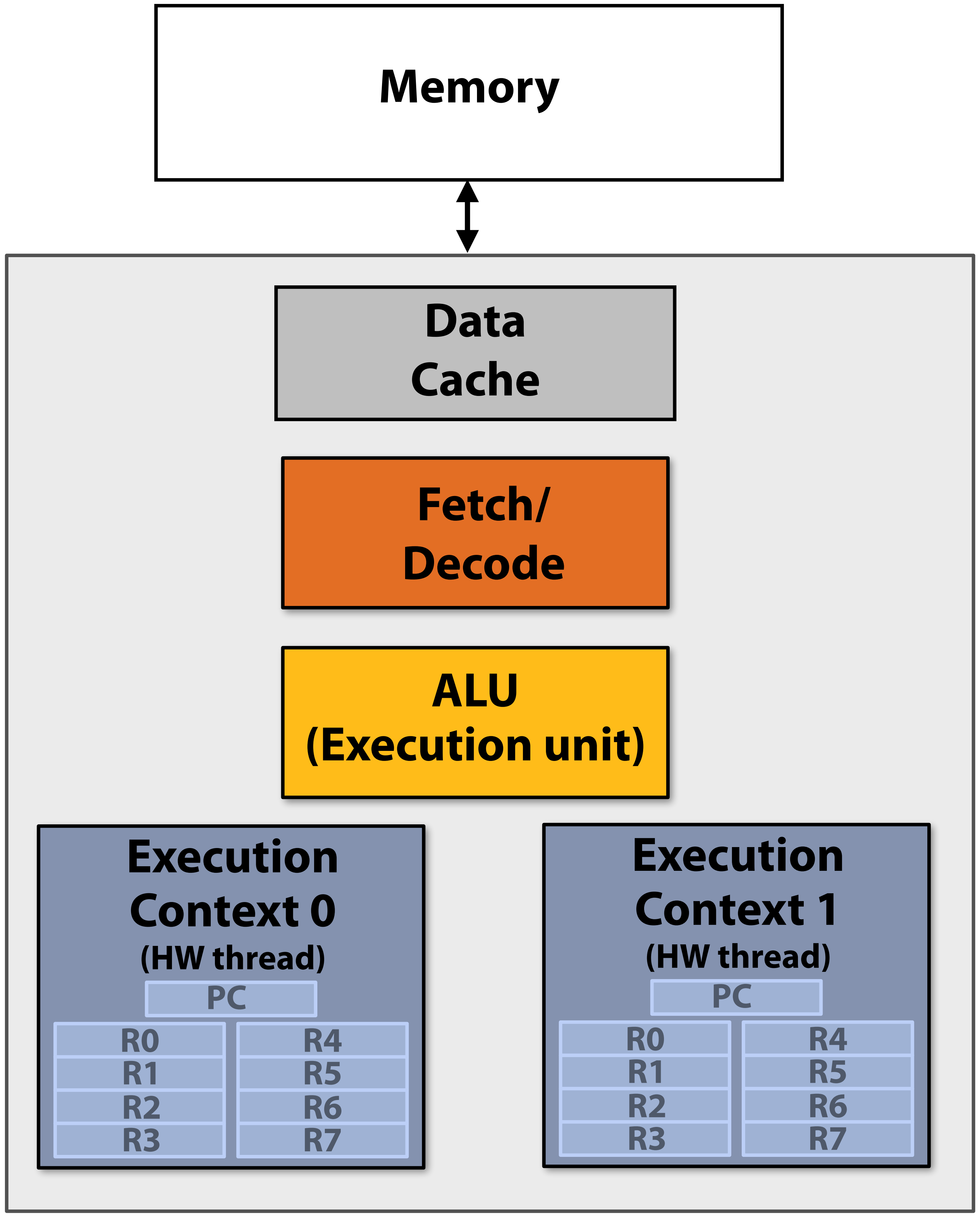
```
ld   r0, addr[r1]
mul  r1, r0, r0
add  r2, r0, r0
mul  r3, r1, r2
...
...
...
...
...
st   addr[r2], r0
```

Instruction stream 1

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

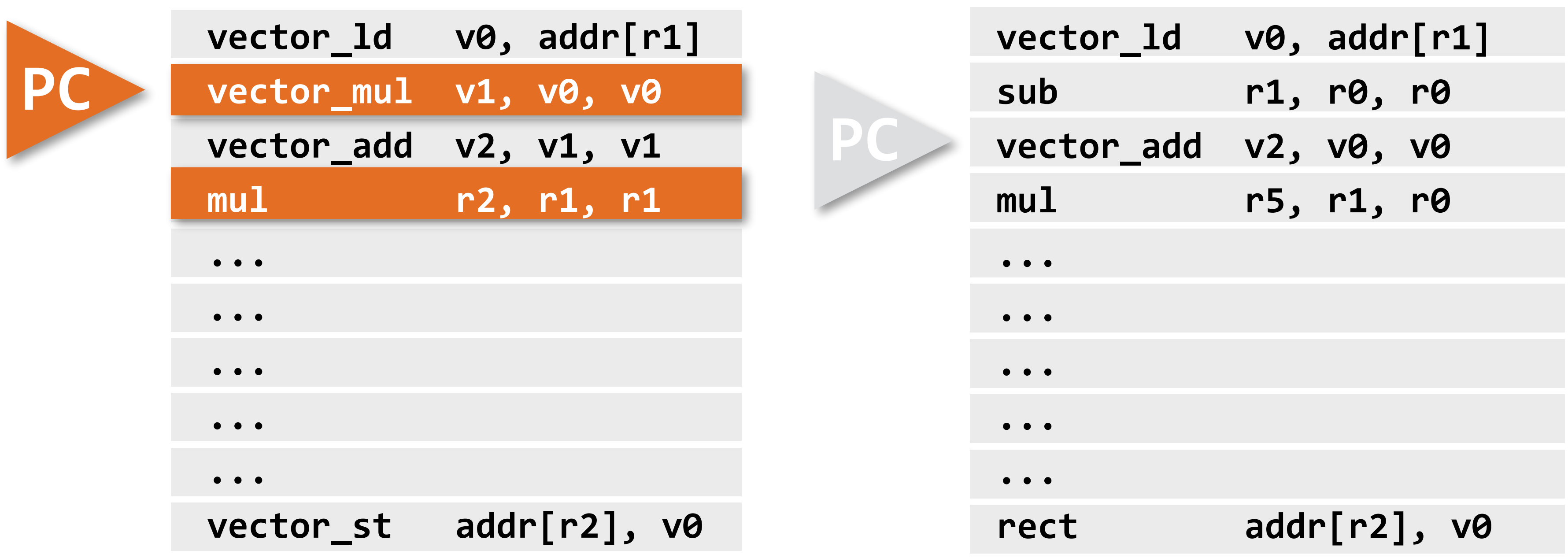
Note: threads can be running completely different instruction streams (and be at different points in these streams)

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

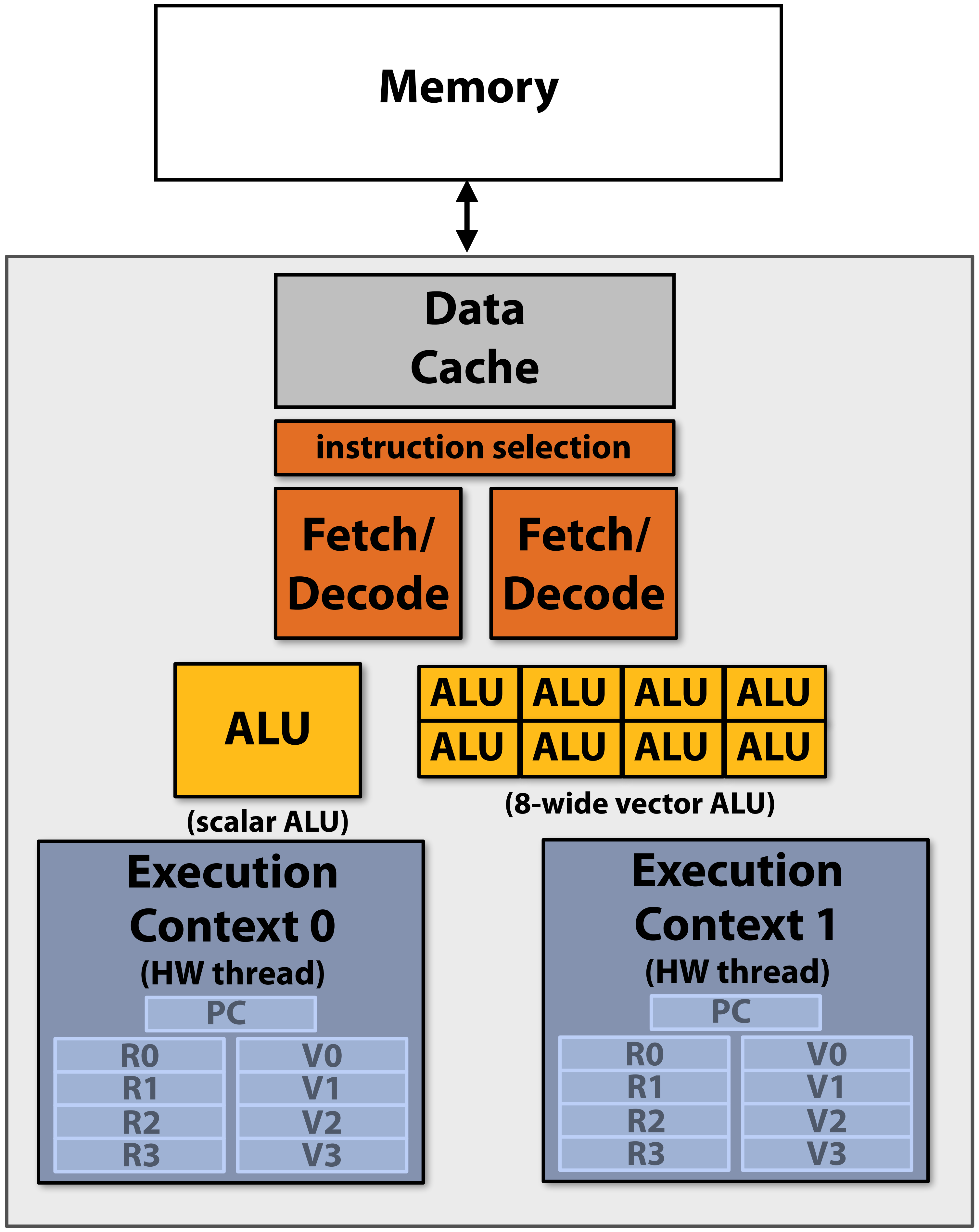


Instruction stream 0

```
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

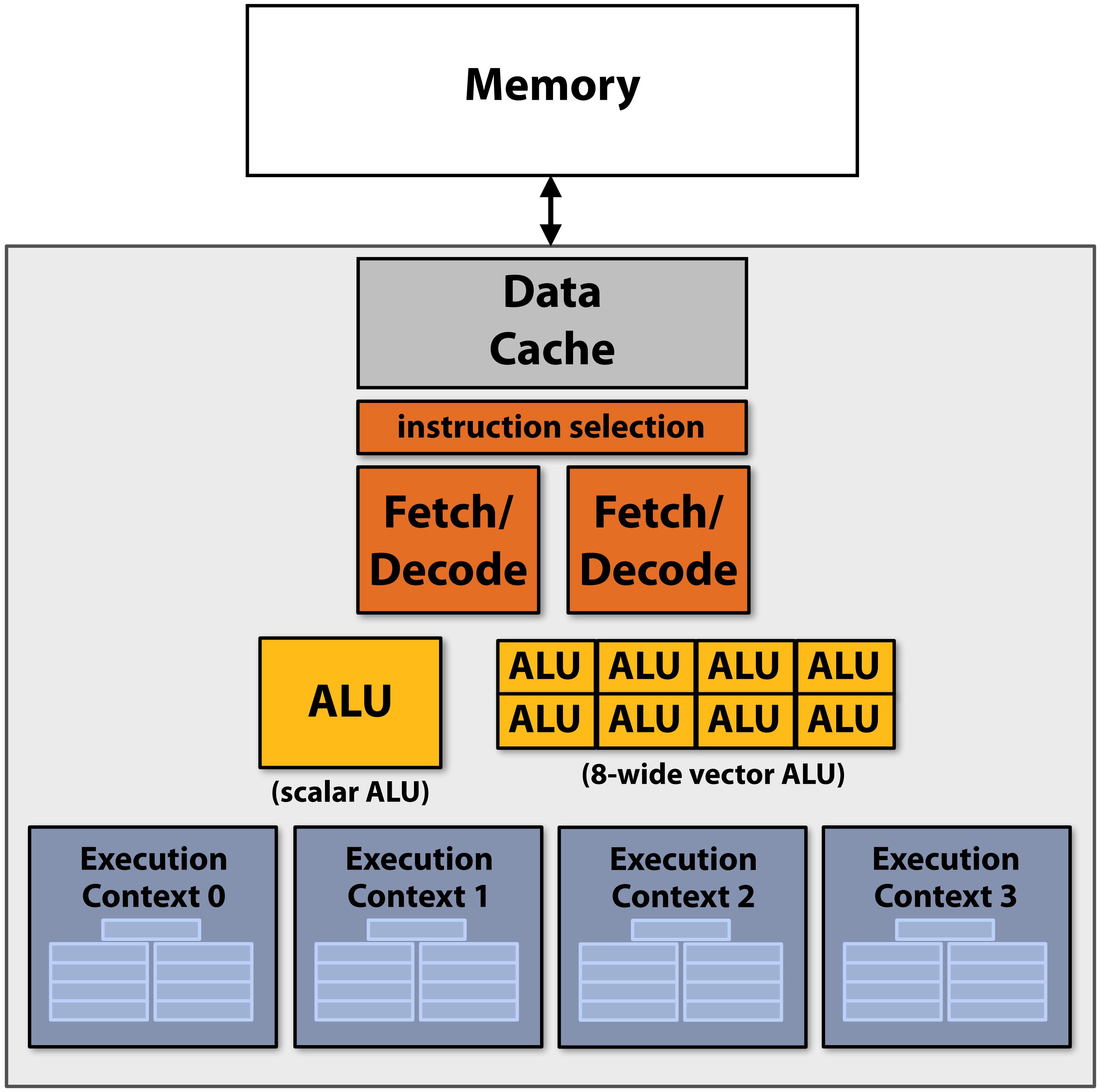
```
vector_ld    v0, addr[r1]
sub          r1, r0, r0
vector_add   v2, v0, v0
mul          r5, r1, r0
...
...
...
...
...
rect         addr[r2], v0
```

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
```

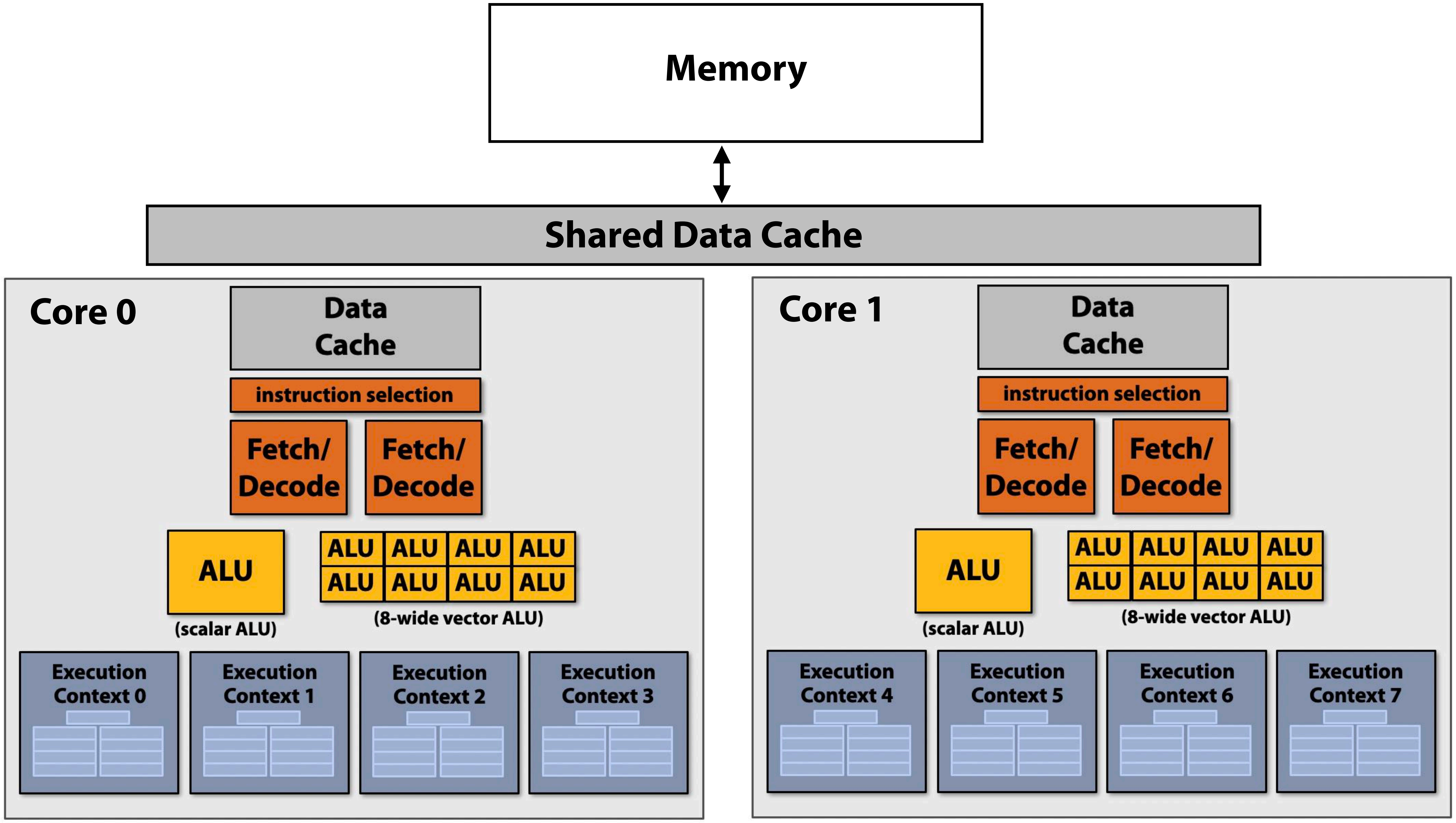


Execution of hardware threads may or may not be interleaved in time

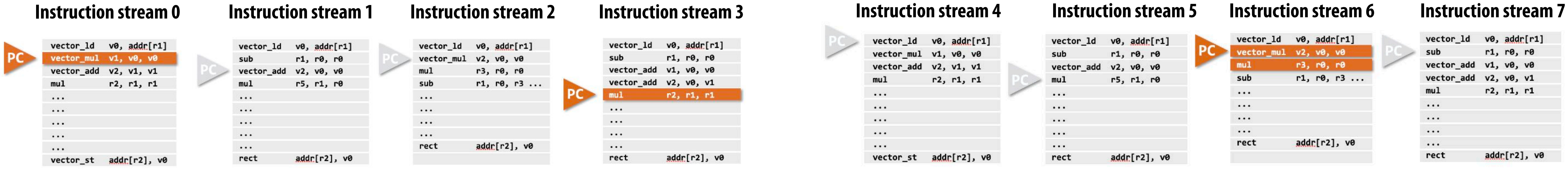
(instructions from different threads may be running simultaneously)

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



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
```

Instruction stream 4

```
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 5

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

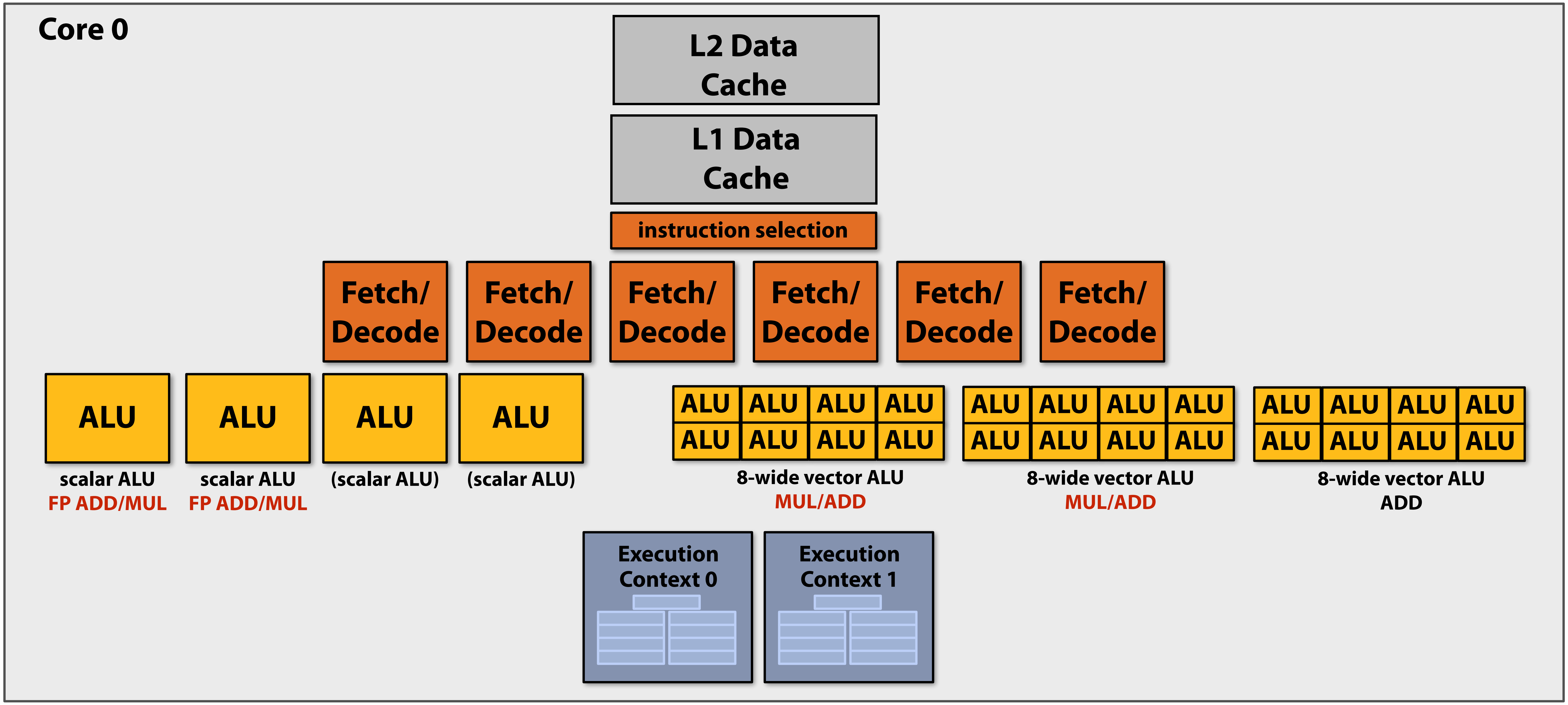
Instruction stream 6

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

Instruction stream 7

```
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
```

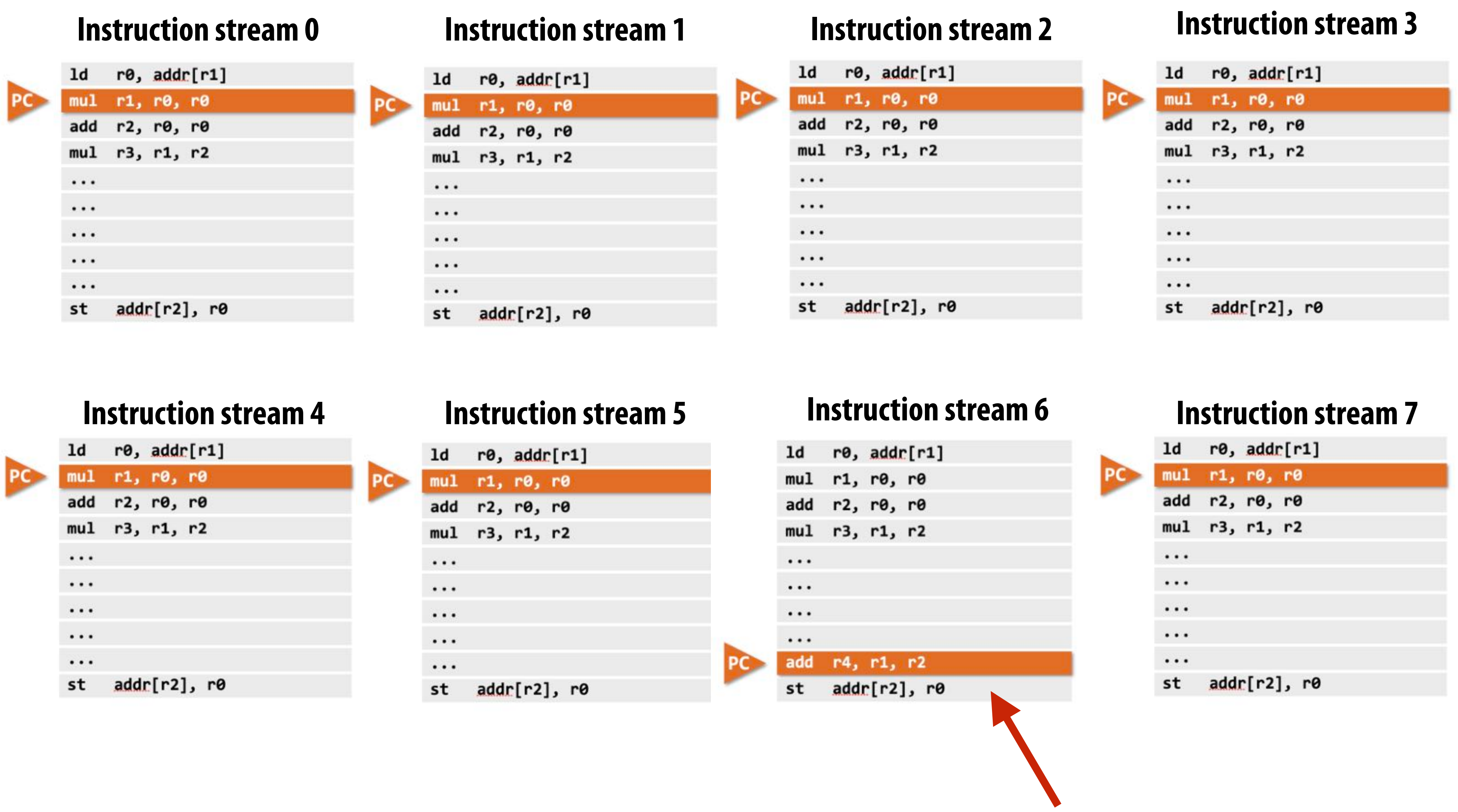
Intel Skylake/Kaby Lake core



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)



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

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

Instruction stream 2

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

Instruction stream 3

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

Instruction stream 4

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

Instruction stream 5

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

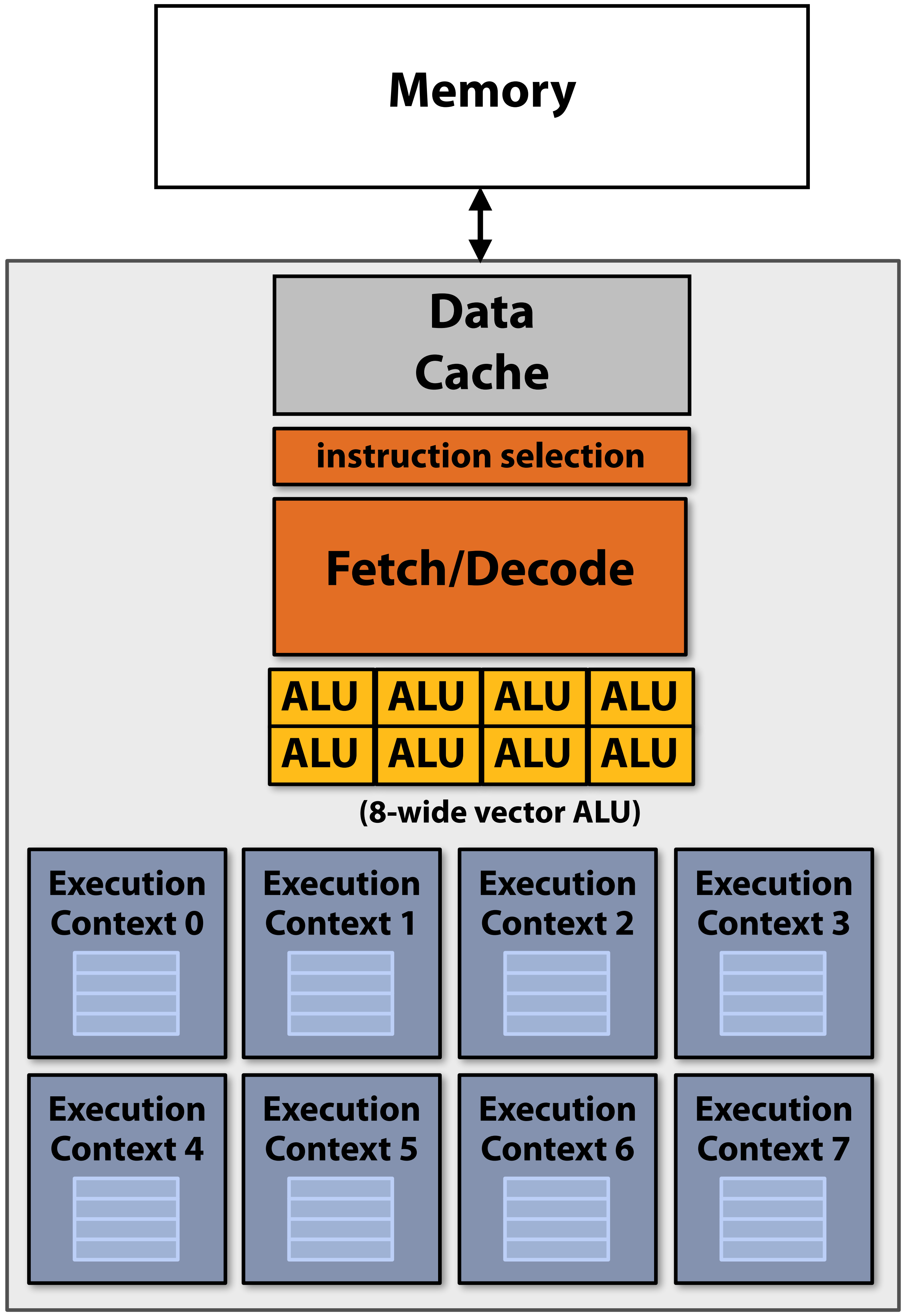
Instruction stream 6

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

Instruction stream 7

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

divergent execution



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.

NVIDIA V100 SM “sub-core”

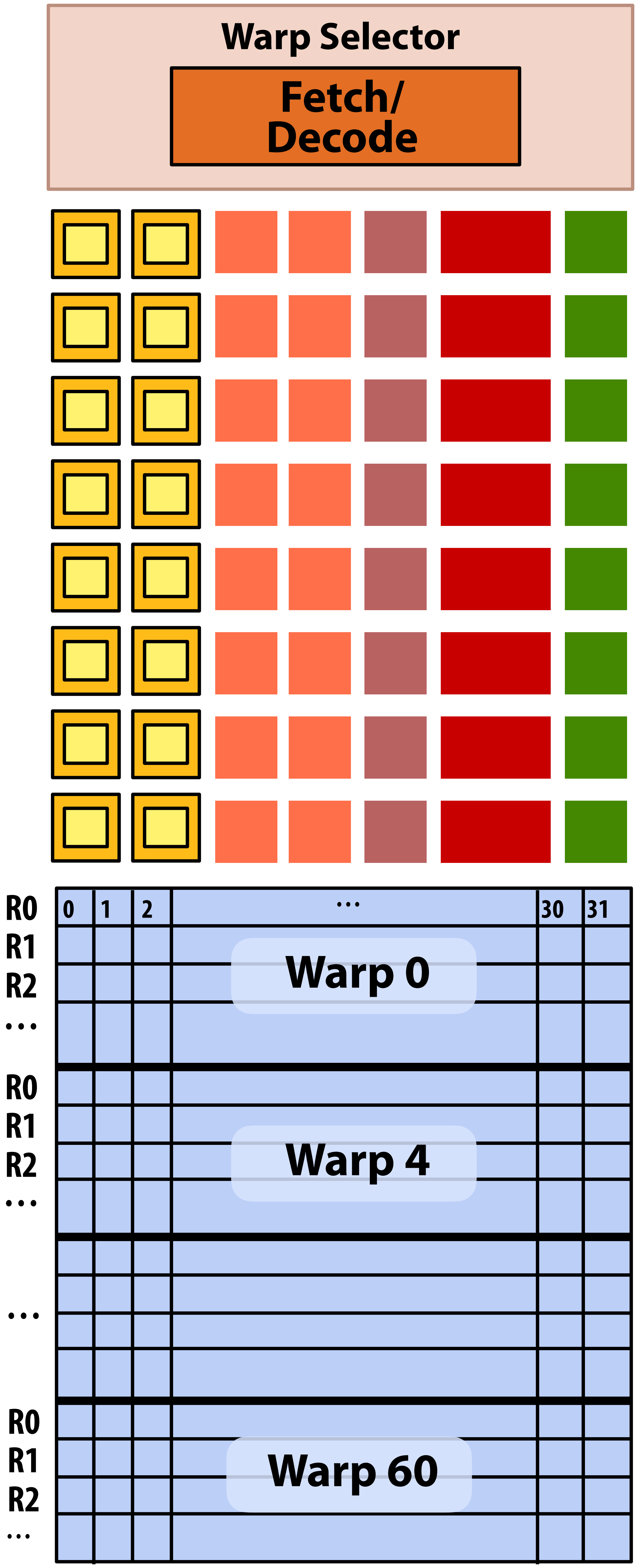
🟨 = SIMD fp32 functional unit,
control shared across 16 units
(16 x MUL-ADD per clock *)

🟧 = 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



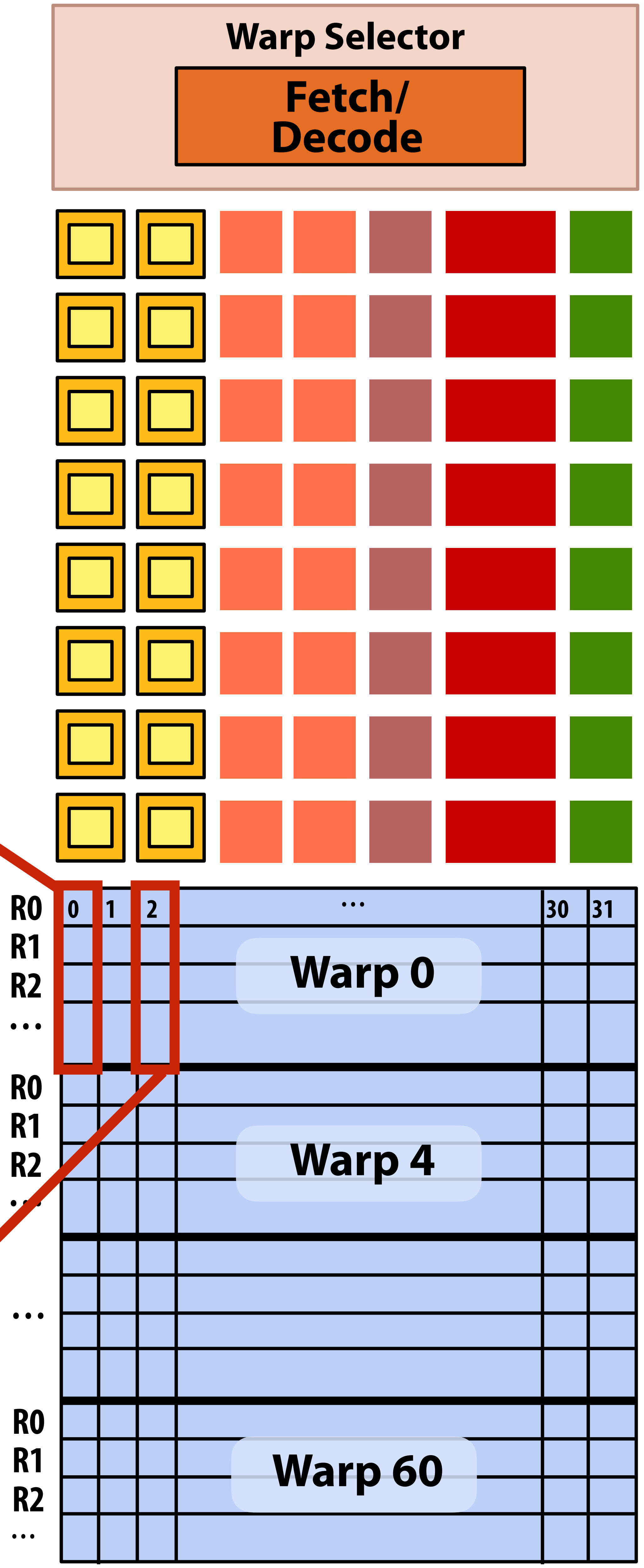
* one 32-wide SIMD operation every two clocks

** one 32-wide SIMD operation every four clocks

NVIDIA V100 SM “sub-core”

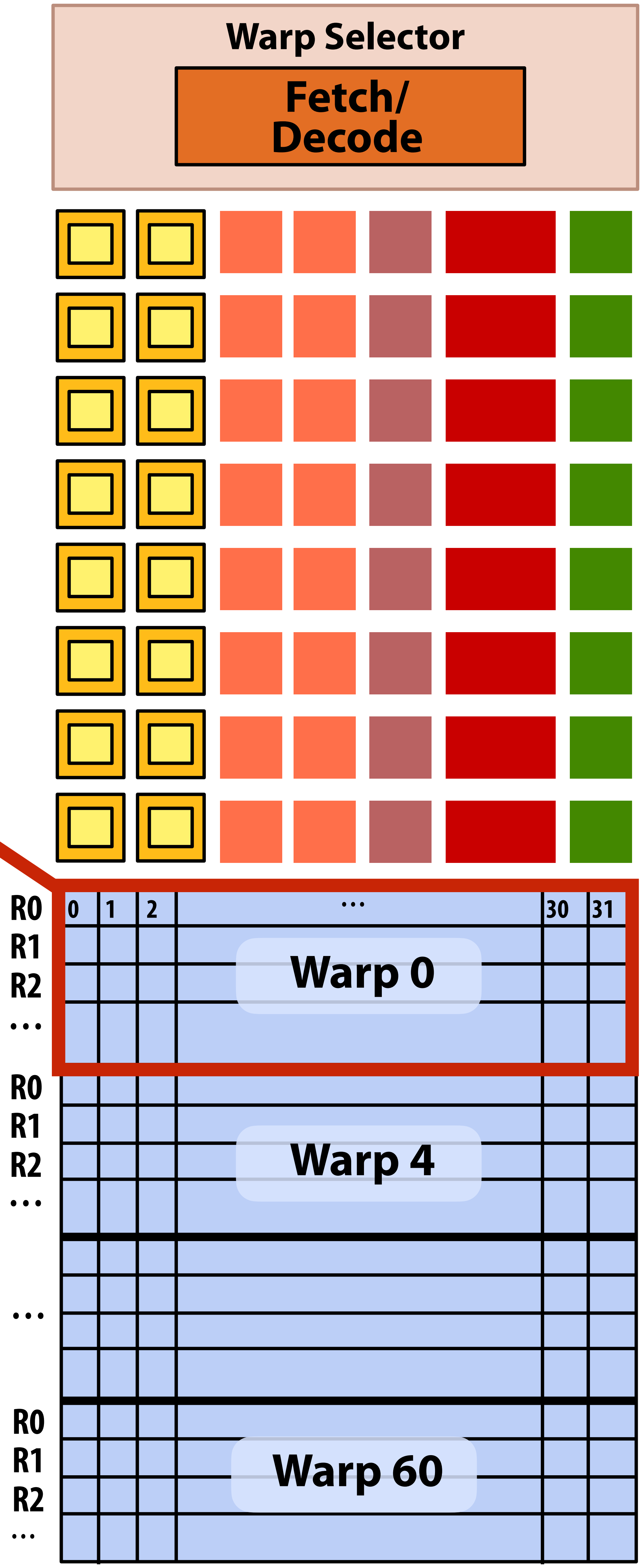
Scalar registers for one CUDA thread: R0, R1, etc…

Scalar registers for another CUDA thread: R0, R1, etc…



NVIDIA V100 SM “sub-core”

Scalar registers for 32 threads in the same “warp”



A group of 32 threads in thread block is called a warp.

- In a thread block, threads 0-31 fall into the same warp (so do threads 32-63, etc.)
- Therefore, a thread block with 256 CUDA threads is mapped to 8 warps.
- Each sub-core in the V100 is capable of scheduling and interleaving execution of up to 16 warps

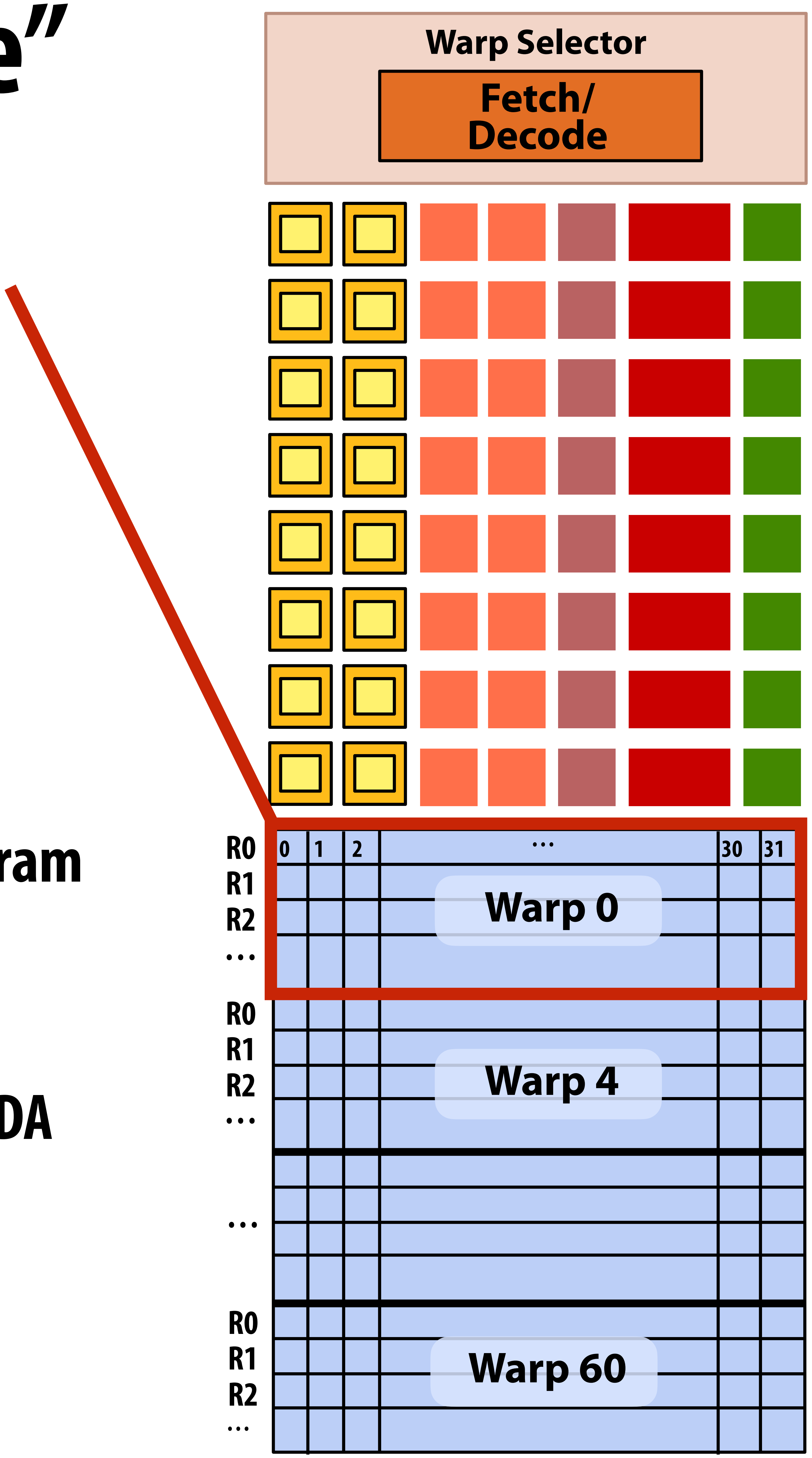
NVIDIA V100 SM “sub-core”

Scalar registers for 32 threads in the same “warp”

Threads in a warp are executed in a SIMD manner *if they share the same instruction*

- If the 32 CUDA threads do not share the same instruction, performance can suffer due to divergent execution.
- This mapping is similar to how ISPC runs program instances in a gang *

A warp is not part of CUDA, but is an important CUDA implementation detail on modern NVIDIA GPUs

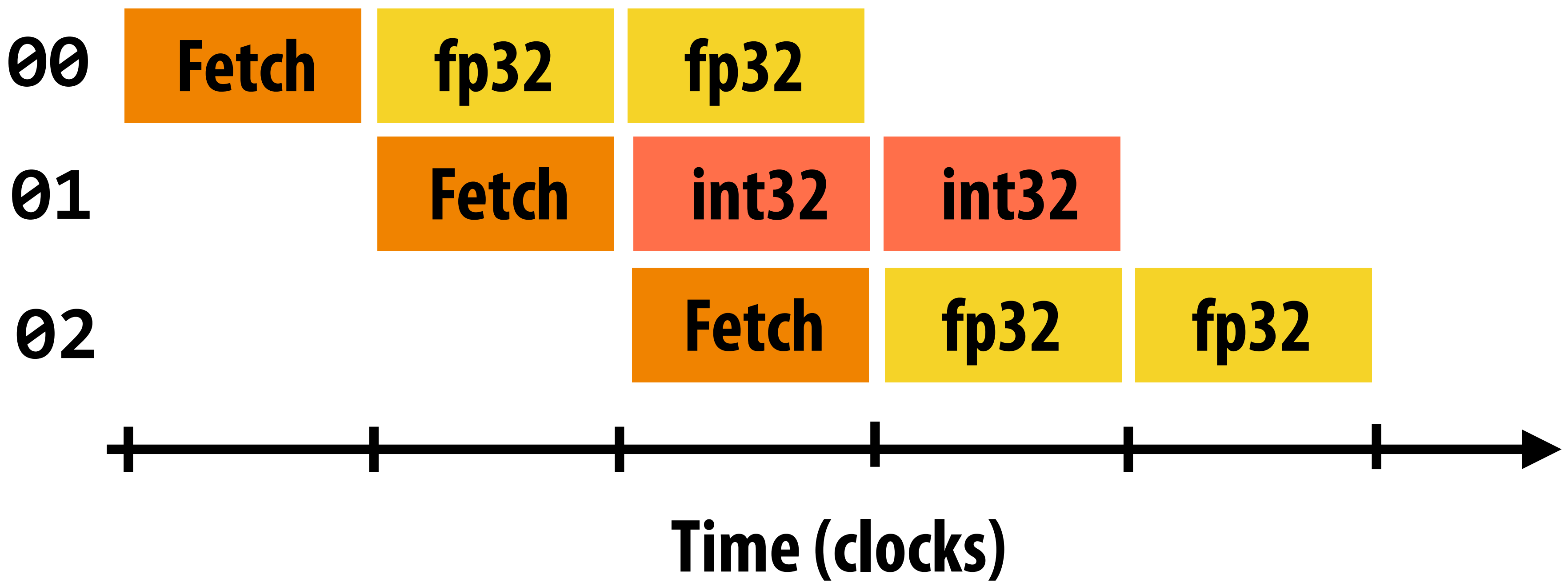


* But GPU hardware is dynamically checking whether 32 independent CUDA threads share an instruction, and if this is true, it executes all 32 threads in a SIMD manner. The CUDA program is not compiled to SIMD instructions like ISPC gangs.

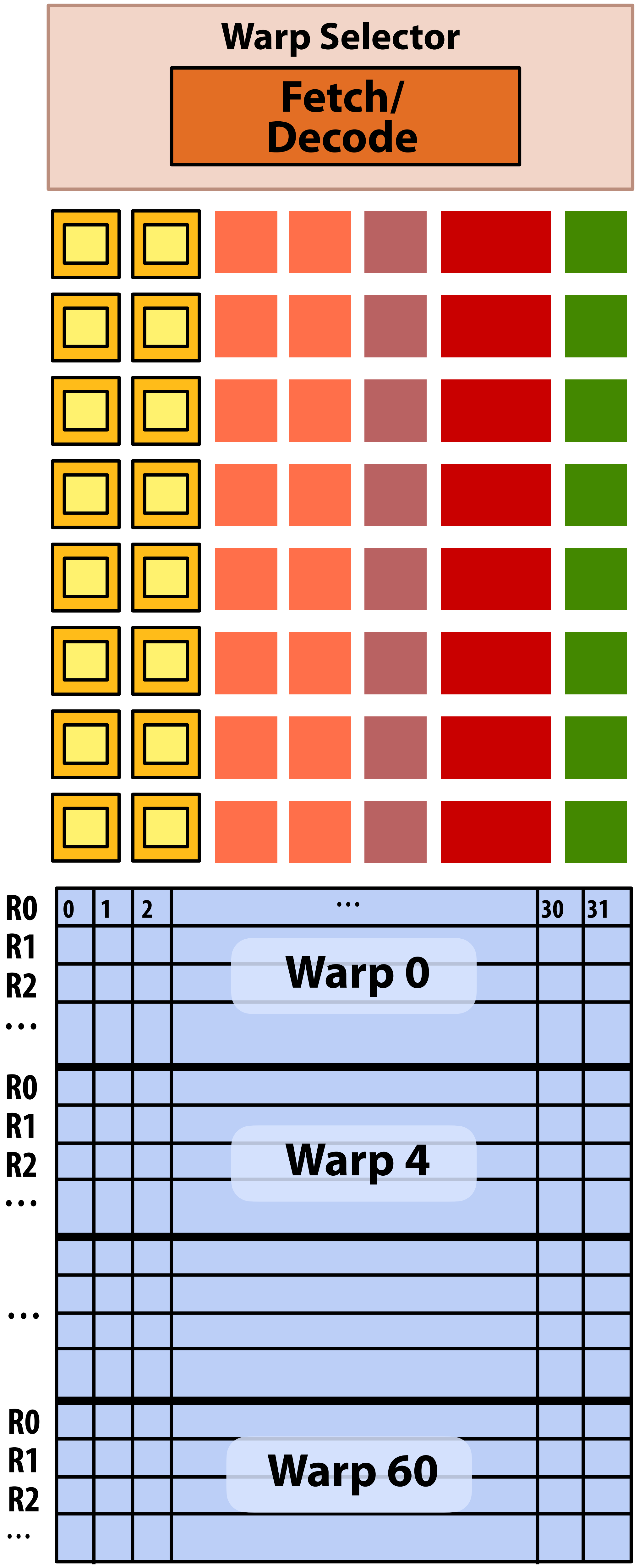
Instruction execution

Instruction stream CUDA threads in a warp…
(note in this example all instructions are independent)

```
00  fp32  mul r0 r1 r2
01  int32 add r3 r4 r5
02  fp32  mul r6 r7 r8
...
```

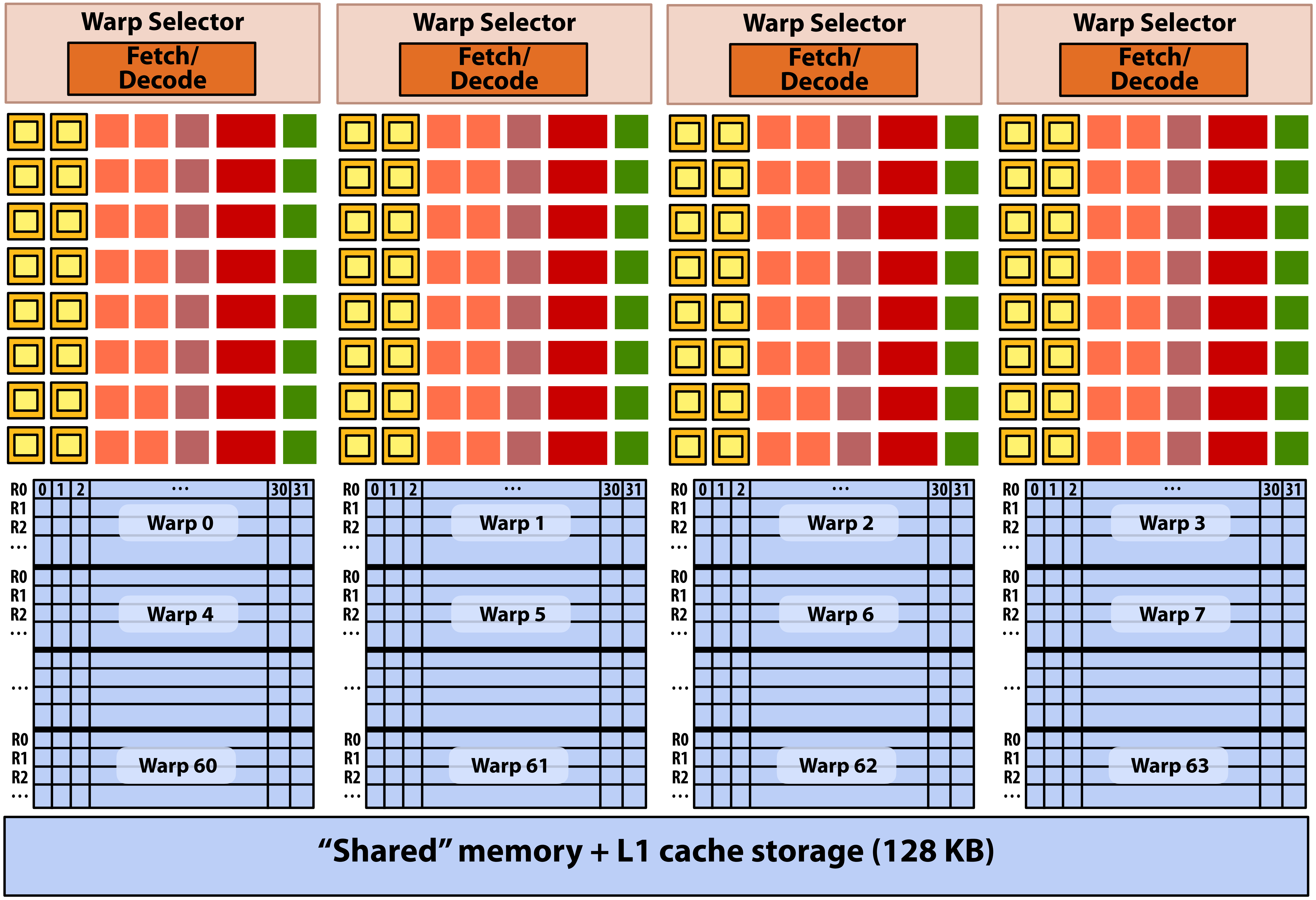


Remember, entire warp of CUDA threads is running this instruction stream.
So each instruction is run by all 32 CUDA threads in the warp.
Since there are 16 ALUs, running the instruction for the entire warp takes two clocks.



NVIDIA V100 GPU SM

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



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

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 **)

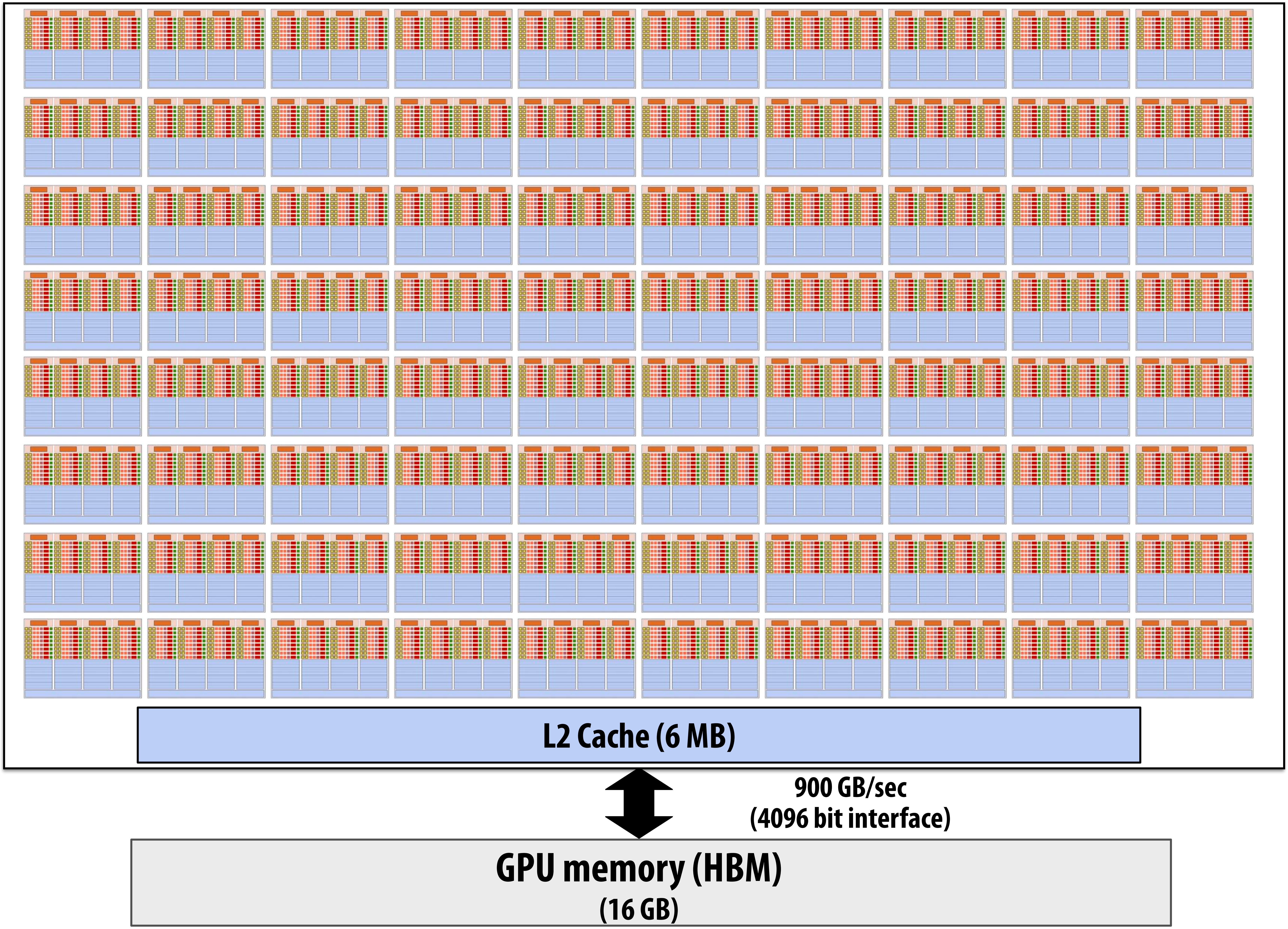
🟥 = Tensor core unit

🟩 = Load/store unit

* one 32-wide SIMD operation every 2 clocks

** one 32-wide SIMD operation every 4 clocks

NVIDIA V100 GPU (80 SMs)



L2 Cache (6 MB)

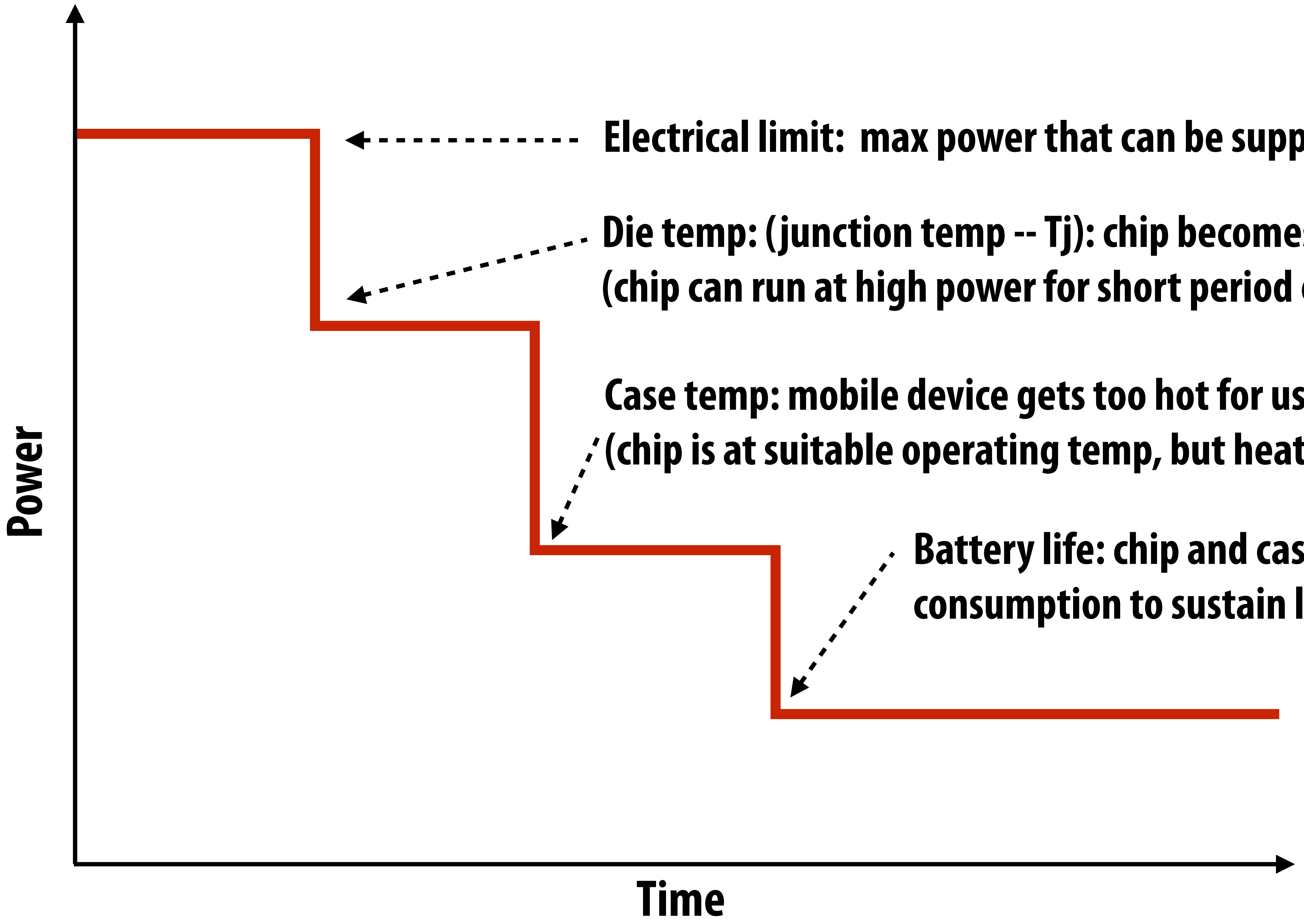
900 GB/sec
(4096 bit interface)

GPU memory (HBM)
(16 GB)

Hardware specialization

Limits on chip power consumption

- General mobile processing rule: the longer a task runs the less power it can use
  - Processor's power consumption is limited by heat generated (efficiency is required for more than just maximizing battery life)



Electrical limit: max power that can be supplied to chip

Die temp: (junction temp -- Tj): chip becomes unreliable above this temp (chip can run at high power for short period of time until chip heats to Tj)

Case temp: mobile device gets too hot for user to comfortably hold (chip is at suitable operating temp, but heat is dissipating into case)

Battery life: chip and case are cool, but want to reduce power consumption to sustain long battery life for given task

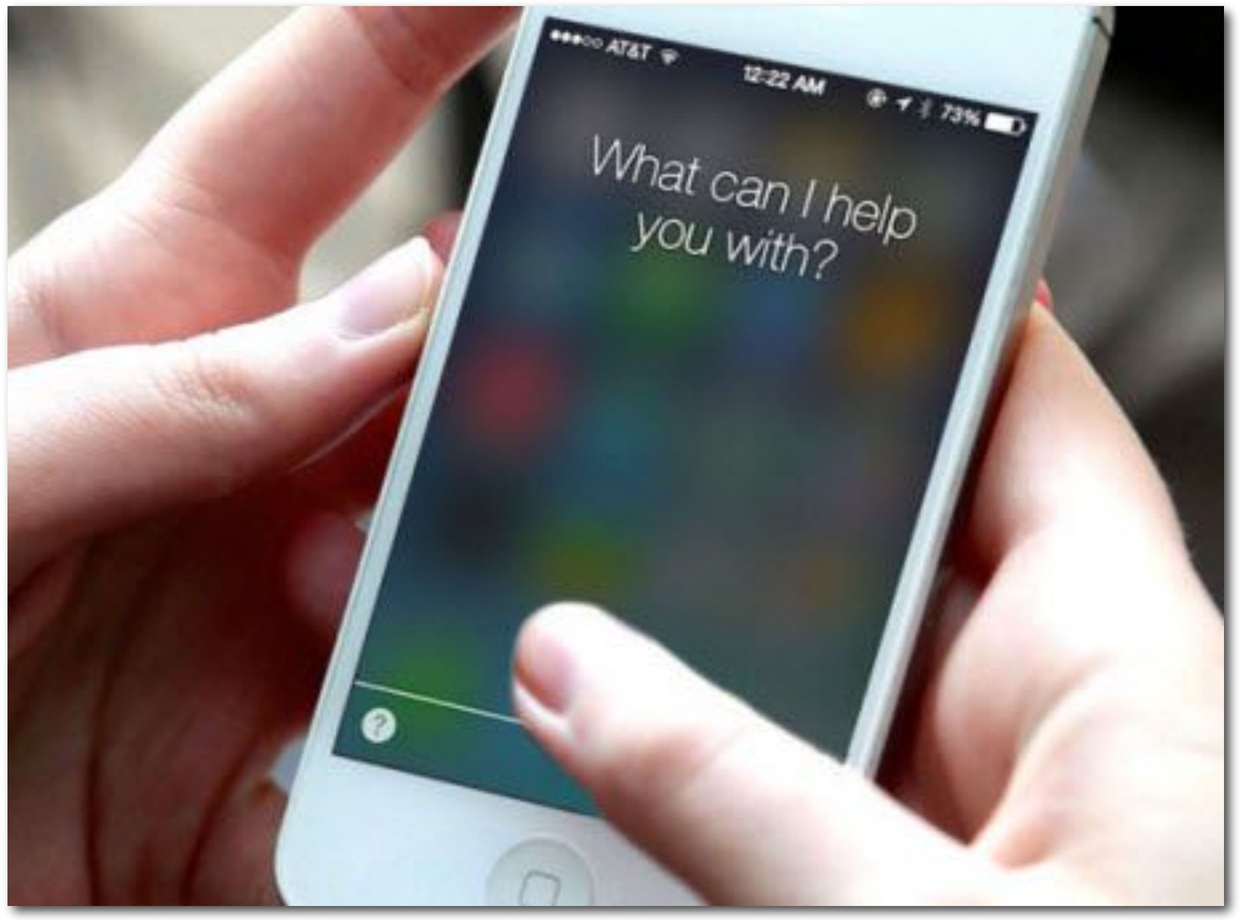
iPhone 6 battery: 7 watt-hours
9.7in iPad Pro battery: 28 watt-hours
15in Macbook Pro: 99 watt-hours

Slide credit: adopted from original slide from M. Shebanow: HPG 2013 keynote

Mobile: benefits of increasing efficiency

- **Run faster for a fixed period of time**
  - Run at higher clock, use more cores (reduce latency of critical task)
  - Do more at once

- **Run at a fixed level of performance for longer**
  - e.g., video playback, health apps
  - Achieve "always-on" functionality that was previously impossible

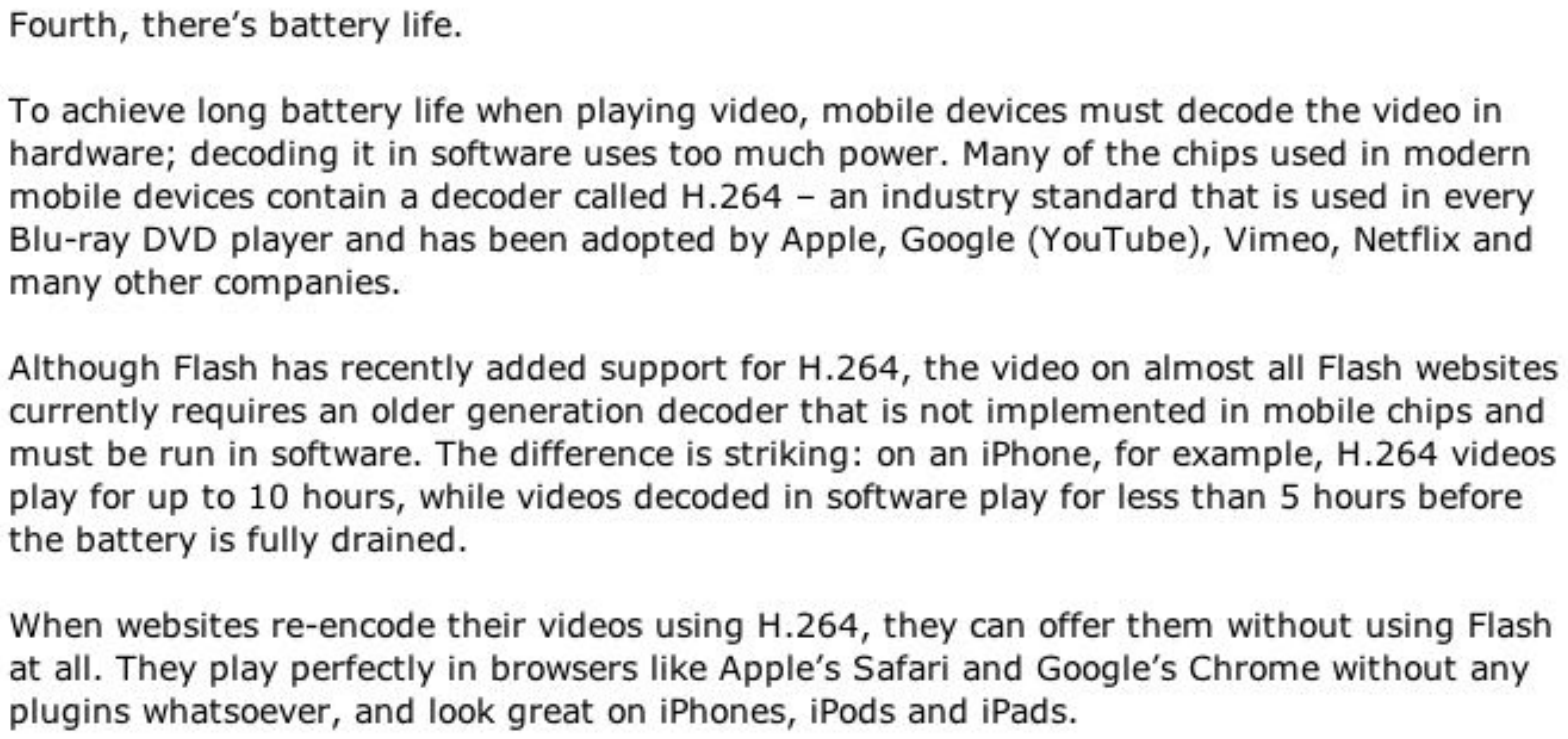


iPhone:
Siri activated by button press or holding phone up to ear

Amazon Echo / Google Home
Always listening

Google Glass: ~40 min recording per charge (nowhere near "always on")

Modern computing: efficiency often matters more than in the past, not less



Fourth, there's battery life.

To achieve long battery life when playing video, mobile devices must decode the video in hardware; decoding it in software uses too much power. Many of the chips used in modern mobile devices contain a decoder called H.264 – an industry standard that is used in every Blu-ray DVD player and has been adopted by Apple, Google (YouTube), Vimeo, Netflix and many other companies.

Although Flash has recently added support for H.264, the video on almost all Flash websites currently requires an older generation decoder that is not implemented in mobile chips and must be run in software. The difference is striking: on an iPhone, for example, H.264 videos play for up to 10 hours, while videos decoded in software play for less than 5 hours before the battery is fully drained.

When websites re-encode their videos using H.264, they can offer them without using Flash at all. They play perfectly in browsers like Apple's Safari and Google's Chrome without any plugins whatsoever, and look great on iPhones, iPods and iPads.

Steve Jobs' "Thoughts on Flash", 2010

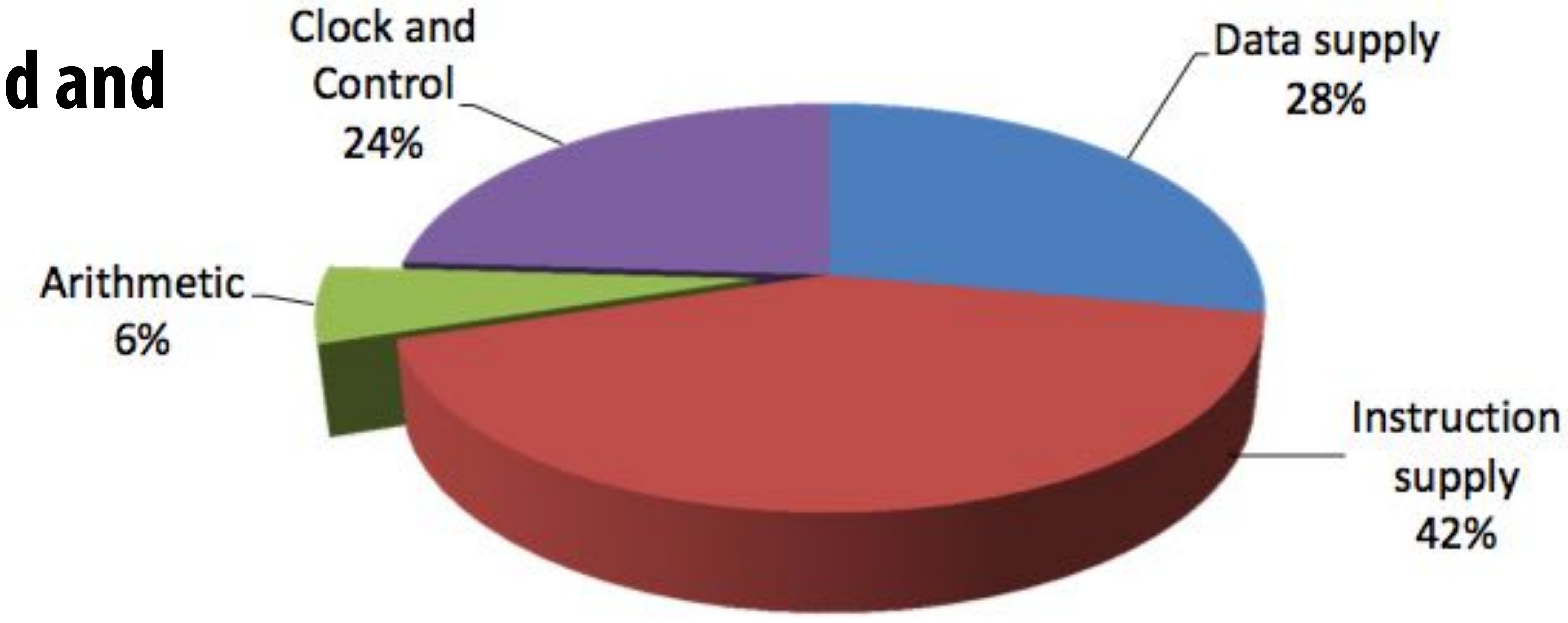
http://www.apple.com/hotnews/thoughts-on-flash/

Efficiency benefits of compute specialization

- **Rules of thumb: compared to high-quality C code on CPU...**

- **Throughput-maximized processor architectures: e.g., GPU cores**
  - **Approximately 10x improvement in perf / watt**
  - **Assuming code maps well to wide data-parallel execution and is compute bound**

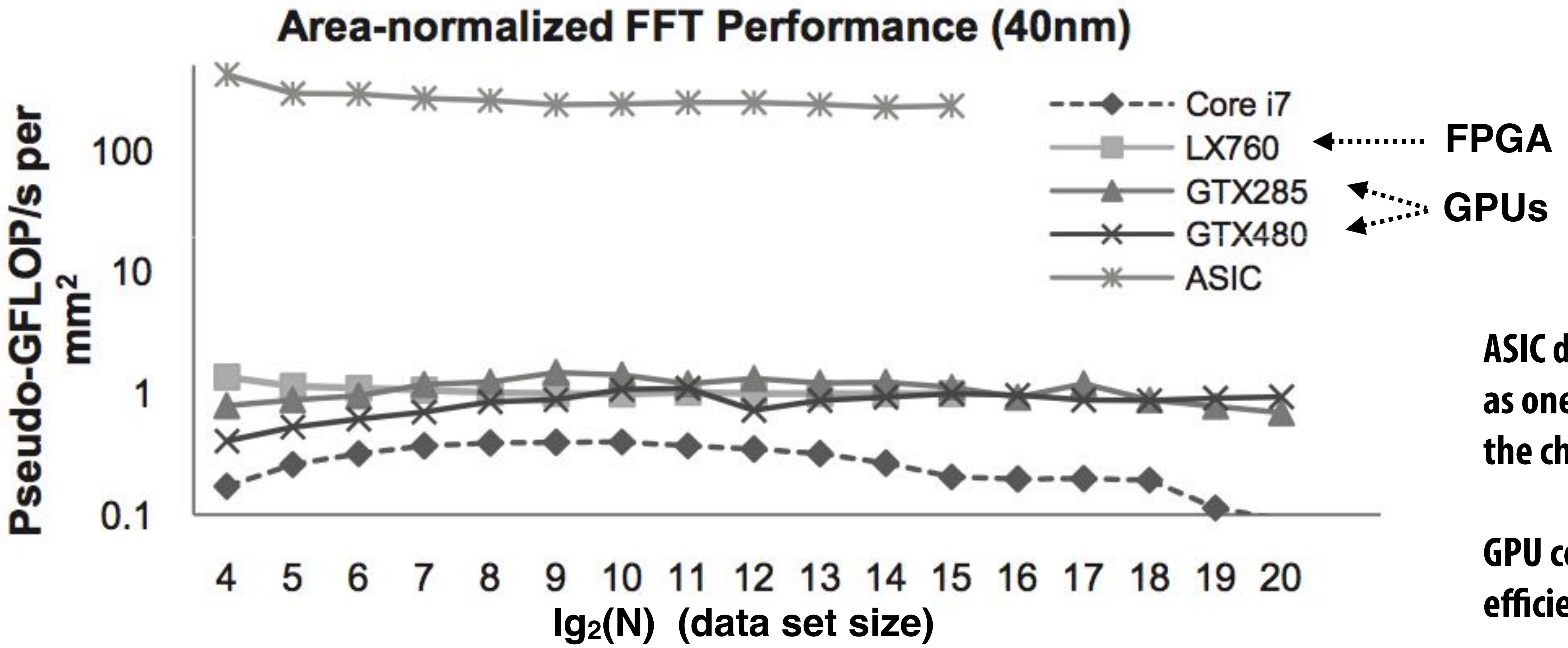
- **Fixed-function ASIC ("application-specific integrated circuit")**
  - **Can approach 100-1000x or greater improvement in perf/watt**
  - **Assuming code is compute bound and and is not floating-point math**



Efficient Embedded Computing [Dally et al. 08]
[Figure credit Eric Chung]

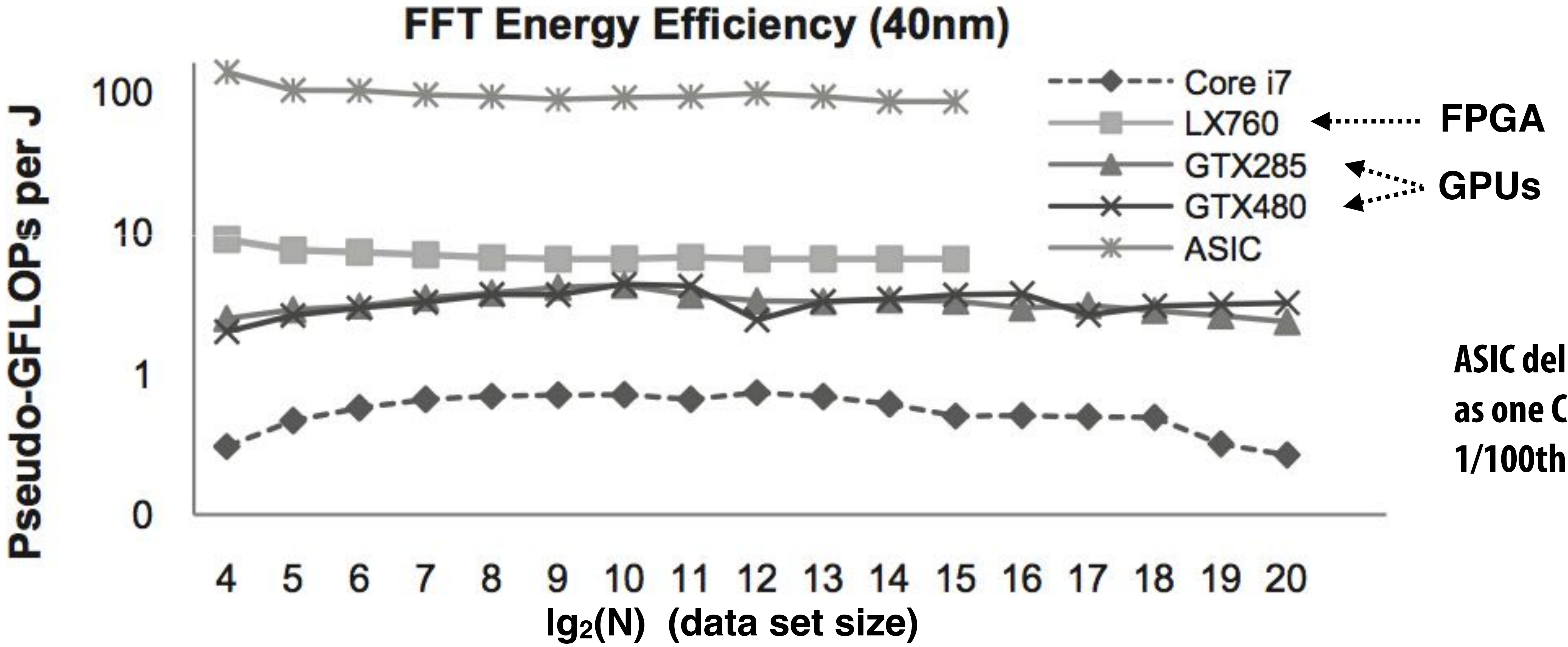
[Source: Chung et al. 2010 , Dally 08]

Hardware specialization increases efficiency



ASIC delivers same performance as one CPU core with ~ 1/1000th the chip area.

GPU cores: ~ 5-7 times more area efficient than CPU cores.



ASIC delivers same performance as one CPU core with only ~ 1/100th the power.

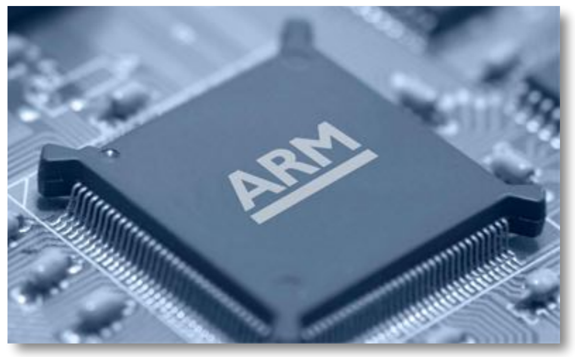
[Chung et al. MICRO 2010]

Modern systems use specialized HW for…

- Image/video encode/decode (e.g., H.264, JPG)
- Audio recording/playback
- Voice "wake up" (e.g., Ok Google)
- Camera "RAW" processing: processing data acquired by image sensor into images that are pleasing to humans
- Many 3D graphics tasks (rasterization, texture mapping, occlusion using the Z-buffer)
- Deep network evaluation (Google's Tensor Processing Unit, Apple Neural engine, etc.)

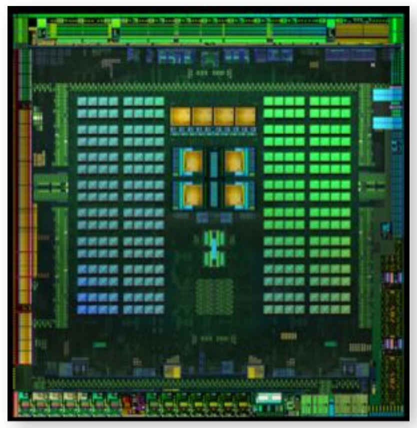
Choosing the right tool for the job

Energy-optimized CPU



Easiest to program

Throughput-oriented processor (GPU)



~10X more efficient

Programmable DSP

HEXAGON™

Google's Pixel Visual Core

FPGA/Future reconfigurable HW

~100X???
(jury still out)

Difficult to program (making it easier is active area of research)

ASIC

Video encode/decode, Audio playback, simple camera RAW, neural computations

~100-1000X more efficient

Not programmable + costs 10-100's millions of dollars to design / verify / create

Credit Pat Hanrahan for this taxonomy

Data movement has high energy cost

- Rule of thumb in mobile system design: always seek to reduce amount of data transferred from memory
  - Earlier in class we discussed minimizing communication to reduce stalls (poor performance). Now, we wish to reduce communication to reduce energy consumption
- "Ballpark" numbers [Sources: Bill Dally (NVIDIA), Tom Olson (ARM)]
  - Integer op: ~ 1 pJ *
  - Floating point op: ~20 pJ *
  - Reading 64 bits from small local SRAM (1mm away on chip): ~ 26 pJ
  - Reading 64 bits from low power mobile DRAM (LPDDR): ~1200 pJ ←

Suggests that recomputing values, rather than storing and reloading them, is a better answer when optimizing code for energy efficiency!

- Implications
  - Reading 10 GB/sec from memory: ~1.6 watts
  - Entire power budget for mobile GPU: ~1 watt
    (remember phone is also running CPU, display, radios, etc.)
  - iPhone 6 battery: ~7 watt-hours (note: my Macbook Pro laptop: 99 watt-hour battery)
  - Exploiting locality matters!!!

* Cost to just perform the logical operation, not counting overhead of instruction decode, load data from registers, etc.

Welcome to CS348K!

- Make sure you are signed up on Piazza so you get announcements

- See website for tonight's reading