CS260 – Lecture 9

Yan Gu

Algorithm Engineering (aka. How to Write Fast Code)

An Overview of Computer Architecture

Many slides in this lecture are borrowed from the first and second lecture in Stanford CS149 Parallel Computing. The credit is to Prof. Kayvon Fatahalian, and the instructor appreciates the permission to use them in this course.

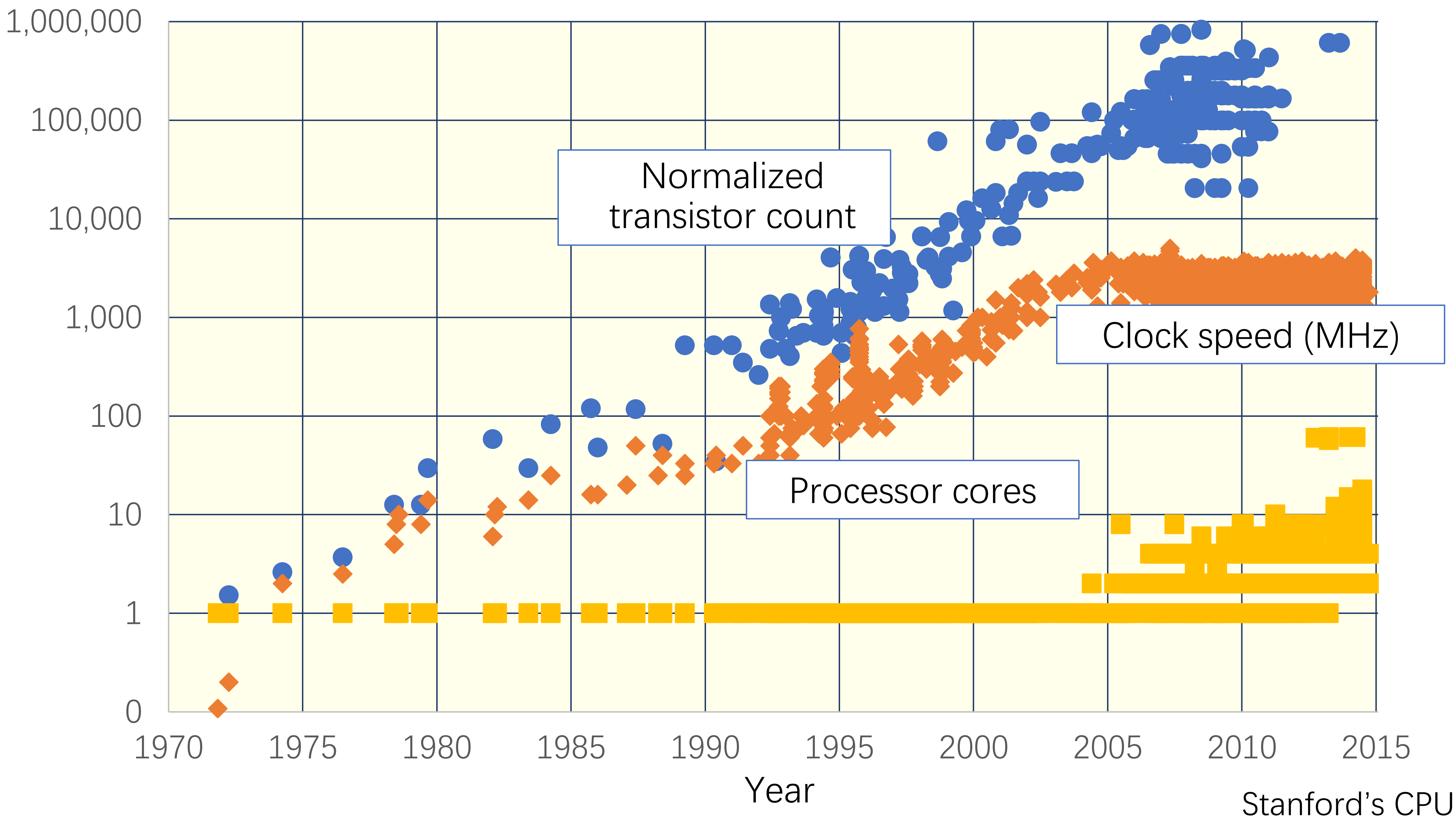
Lecture Overview

- **In this lecture you will learn a brief history of the evolution of architecture**

- **Instruction level parallelism (ILP)**
- **Multiple processing cores**
- **Vector (superscalar, SIMD) processing**
- **Multi-threading (hyper-threading)**
- **Already covered in previous lectures: caching**

- **What we cover:**
  - Programming perspective of view
- **What we do not cover:**
  - How they are implemented in the hardware level (CMU 15-742 / Stanford CS149)

Moore's law: #transistors doubles every 18 months



Stanford's CPU DB [DKM12]

Key question for computer architecture research:
How to use the more transistors for better performance?

Until ~15 years ago: two significant reasons for processor performance improvement

- Increasing CPU clock frequency
- Exploiting instruction-level parallelism (superscalar execution)

What is a computer program?

```
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;
}
```

Review: what is a program?

From a processor's perspective, a program is a sequence of instructions.

```
_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:  retq
```

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…

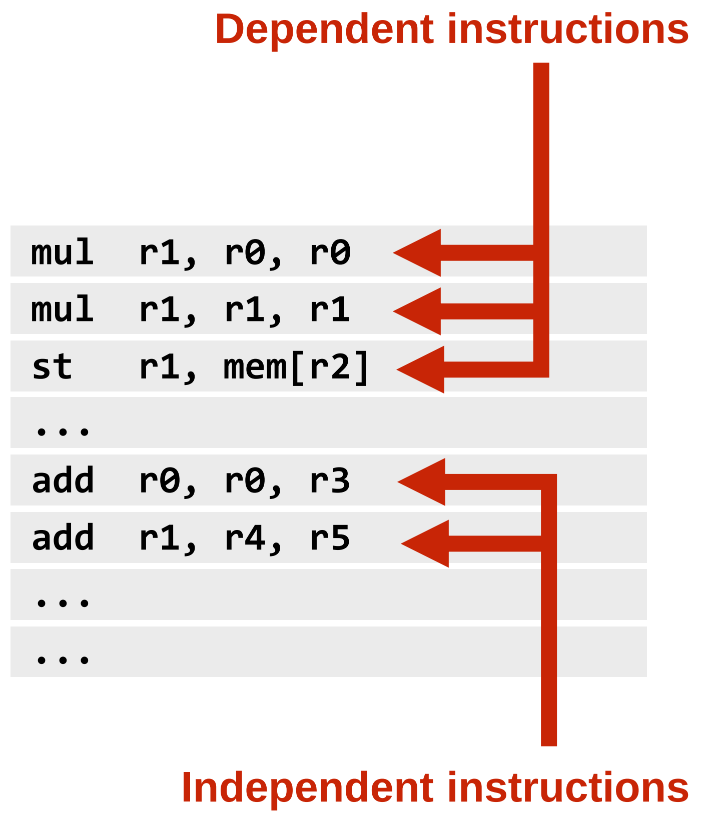
And so on…

```
_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:  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:  retq
```

PC

Instruction level parallelism (ILP)

- Processors did in fact leverage parallel execution to make programs run faster, it was just invisible to the programmer
- Instruction level parallelism (ILP)
  - Idea: Instructions must appear to be executed in program order. BUT independent instructions can be executed simultaneously by a processor without impacting program correctness
  - Superscalar execution: processor dynamically finds independent instructions in an instruction sequence and executes them in parallel



```
mul  r1, r0, r0
mul  r1, r1, r1
st   r1, mem[r2]
...
add  r0, r0, r3
add  r1, r4, r5
...
...
```

ILP example

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

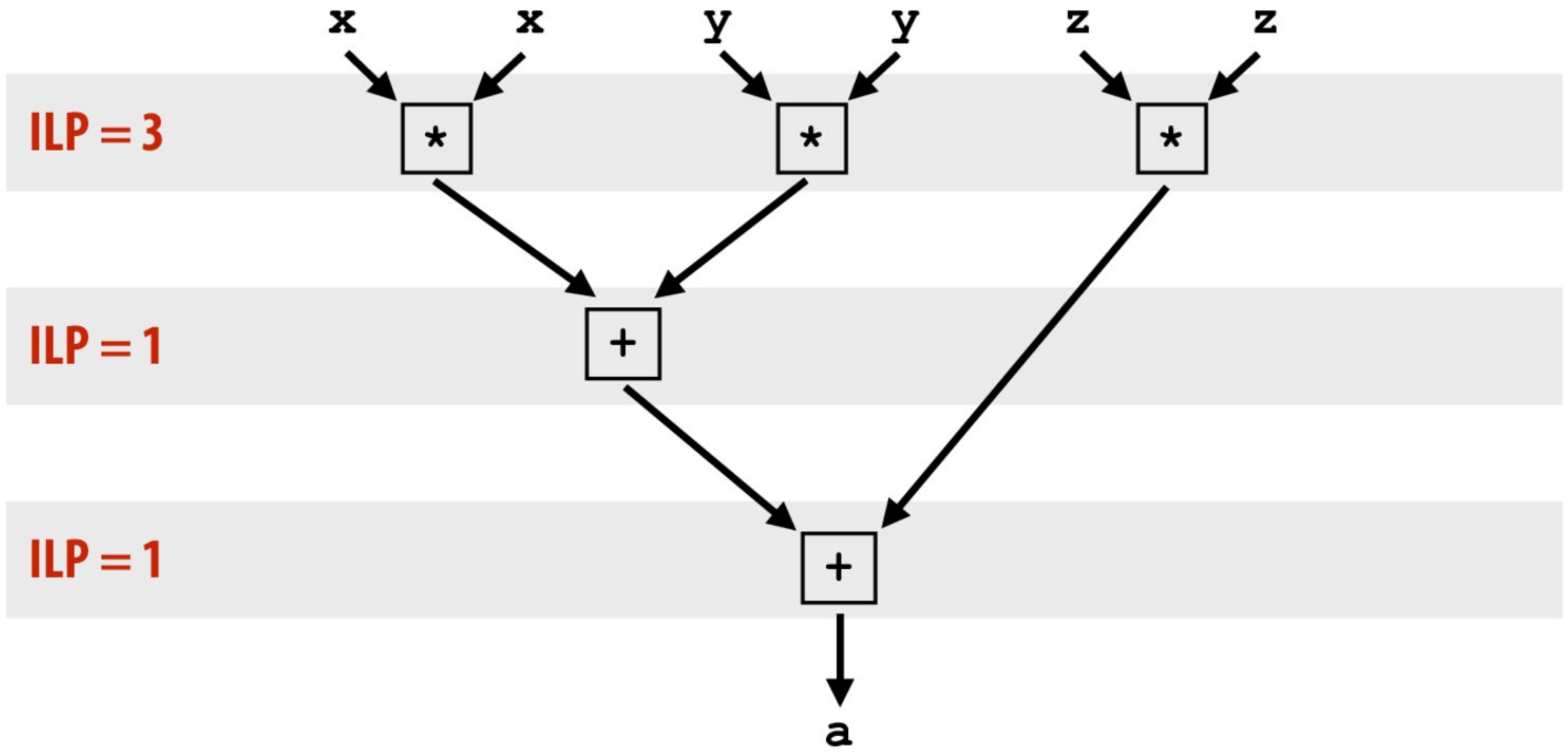
Consider the following program:

```
// 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
```



ILP example

```
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

// now r3 stores value of program 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 instructions 3 and 4

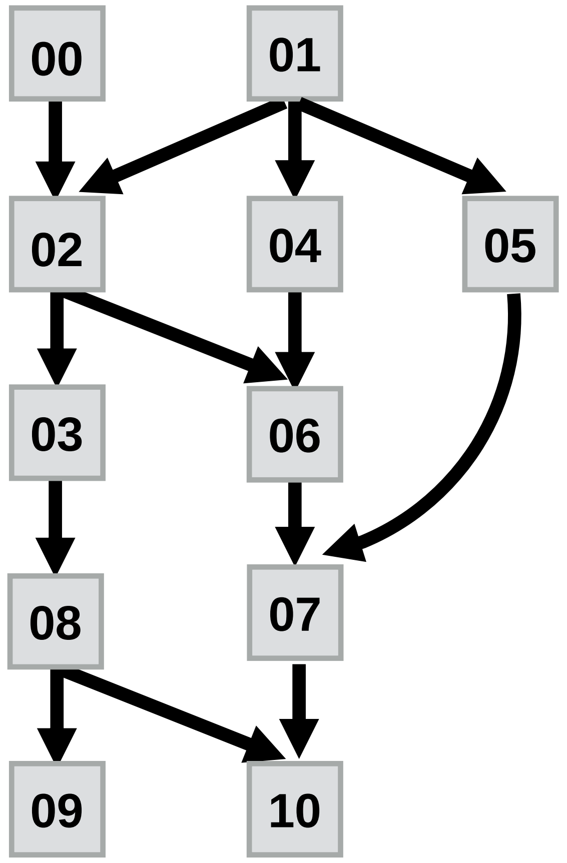
A more complex example

Program (sequence of instructions)

```
PC    Instruction

00    a = 2
01    b = 4

02    tmp2 = a + b          // 6
03    tmp3 = tmp2 + a       // 8
04    tmp4 = b + b          // 8
05    tmp5 = b * b          // 16
06    tmp6 = tmp2 + tmp4    // 14
07    tmp7 = tmp5 + tmp6    // 30

08    if (tmp3 > 7)
09      print tmp3
      else
10      print tmp7
```

value during execution

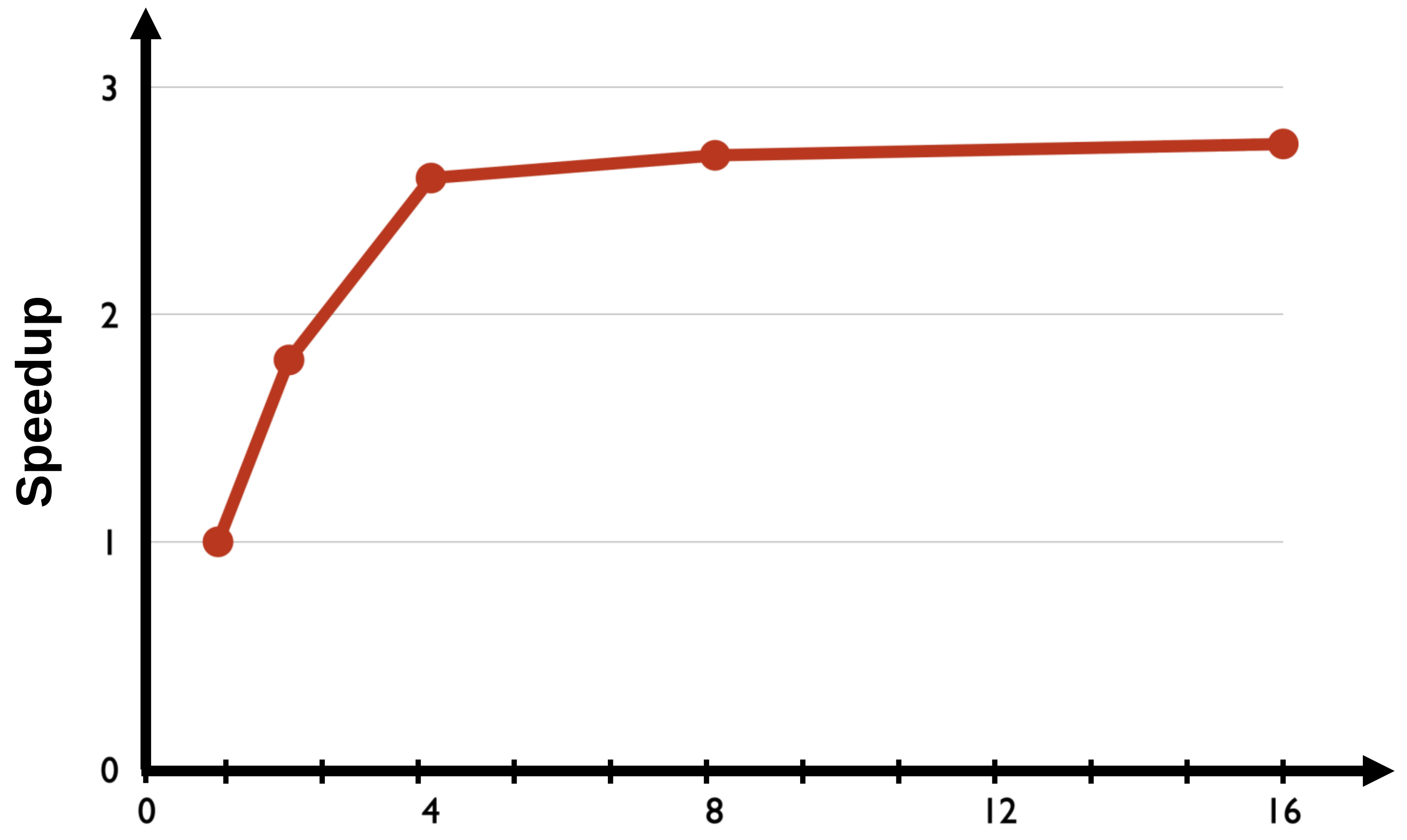
Instruction dependency graph


$$\begin{tikzcd}
\boxed{00} \arrow[d] & \boxed{01} \arrow[dl] \arrow[d] \arrow[dr] & \\
\boxed{02} \arrow[d] \arrow[dr] & \boxed{04} \arrow[d] & \boxed{05} \arrow[ddl, bend left] \\
\boxed{03} \arrow[d] & \boxed{06} \arrow[d] & \\
\boxed{08} \arrow[d] \arrow[dr] & \boxed{07} \arrow[d] & \\
\boxed{09} & \boxed{10} &
\end{tikzcd}$$

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

Diminishing returns of superscalar execution

Most available ILP is exploited by a processor capable of issuing four instructions per clock (Little performance benefit from building a processor that can issue more)



Instruction issue capability of processor (instructions/clock)

Source: Culler & Singh (data from Johnson 1991)

Until ~15 years ago: two significant reasons for processor performance improvement

- Increasing CPU clock frequency

- Exploiting instruction-level parallelism (superscalar execution)

Part 1: Parallel Execution

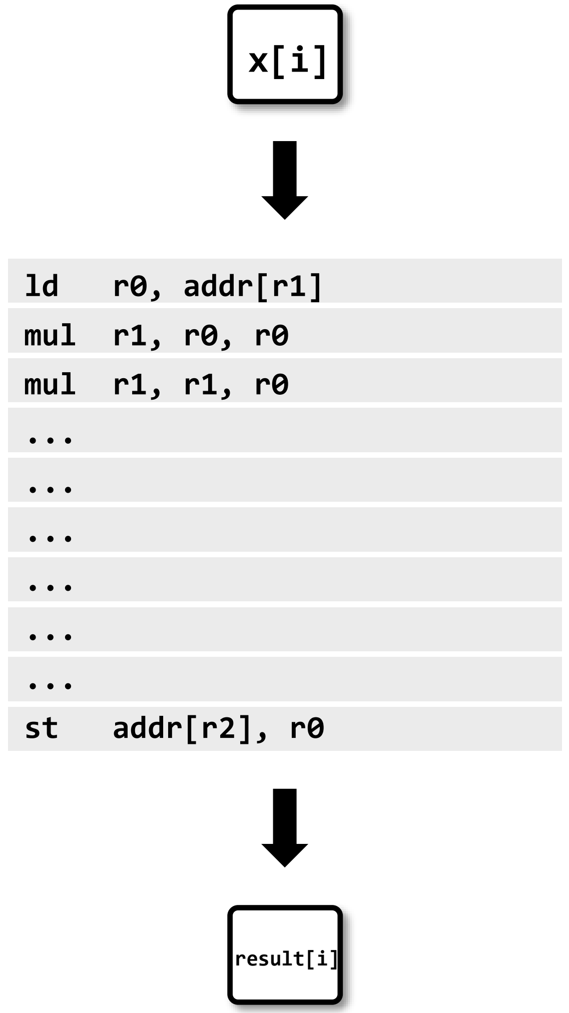
Example program

```
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;
   }
}
```

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

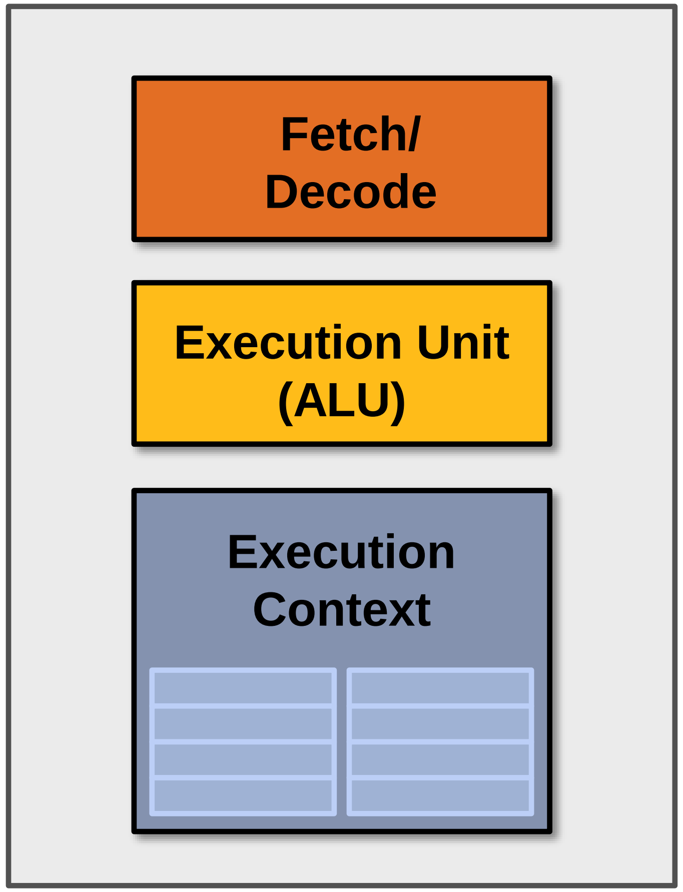
```
void sinx(int N, int terms, float* x, float* result)
{
   for (int i=0; i<N; i++)
   {
         float value = x[i];
         float numer = x[i] * x[i] * x[i];
         int denom = 6;  // 3!
         int sign = -1;

         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;
   }
}
```

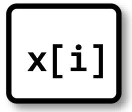


```
ld    r0, addr[r1]
mul   r1, r0, r0
mul   r1, r1, r0
...
...
...
...
...
...
st    addr[r2], r0
```

Execute program

My very simple processor: executes one instruction per clock

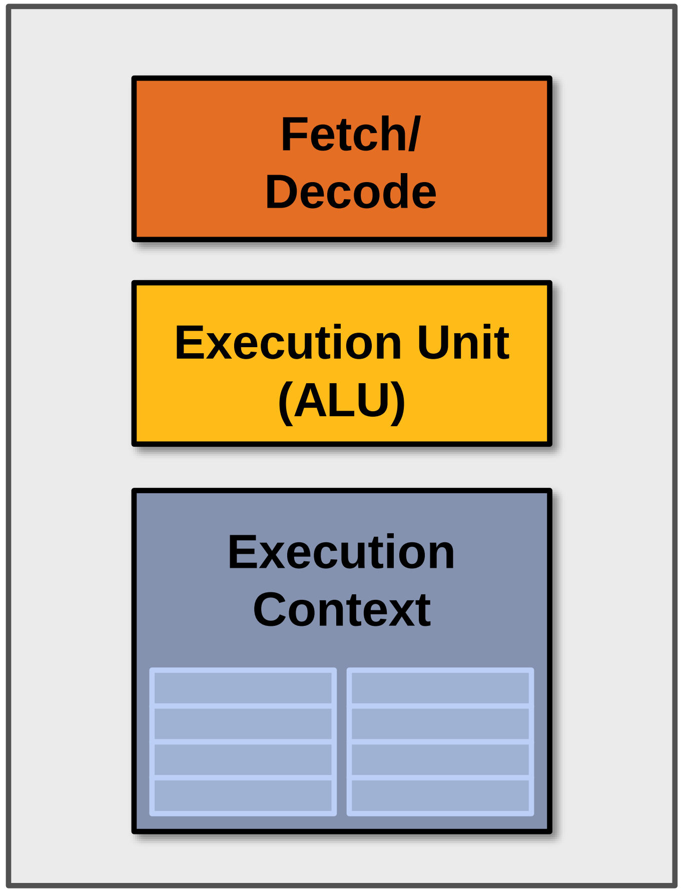


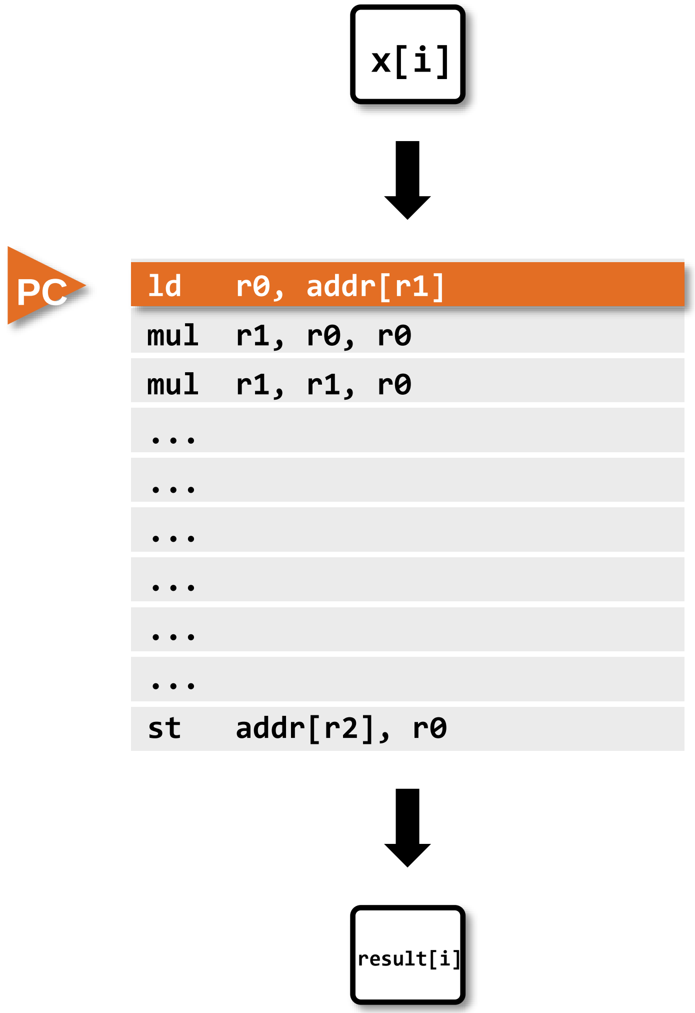


```
ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...
...
...
...
...
...
st   addr[r2], r0
```

Execute program

My very simple processor: executes one instruction per clock

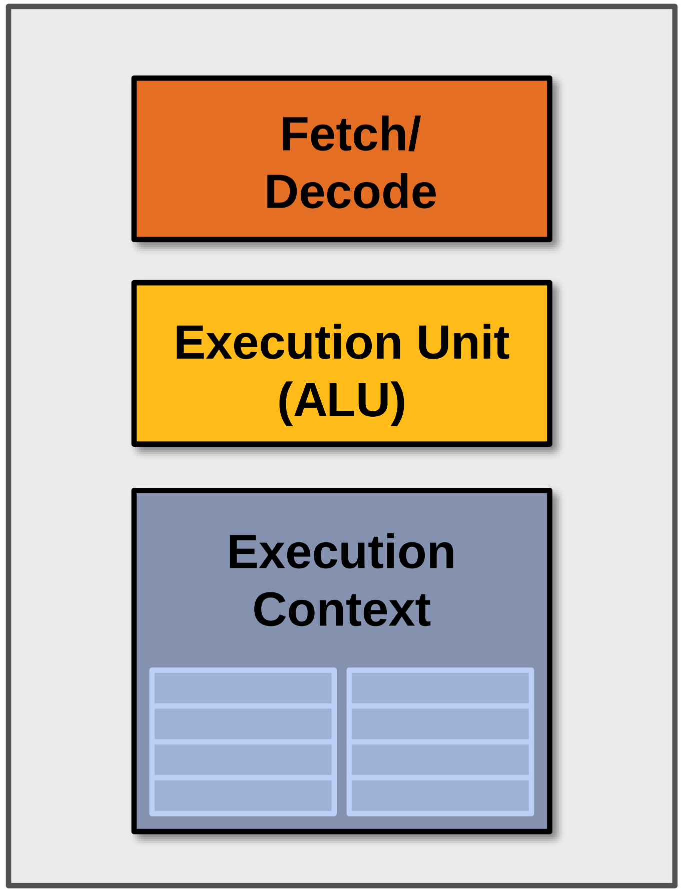


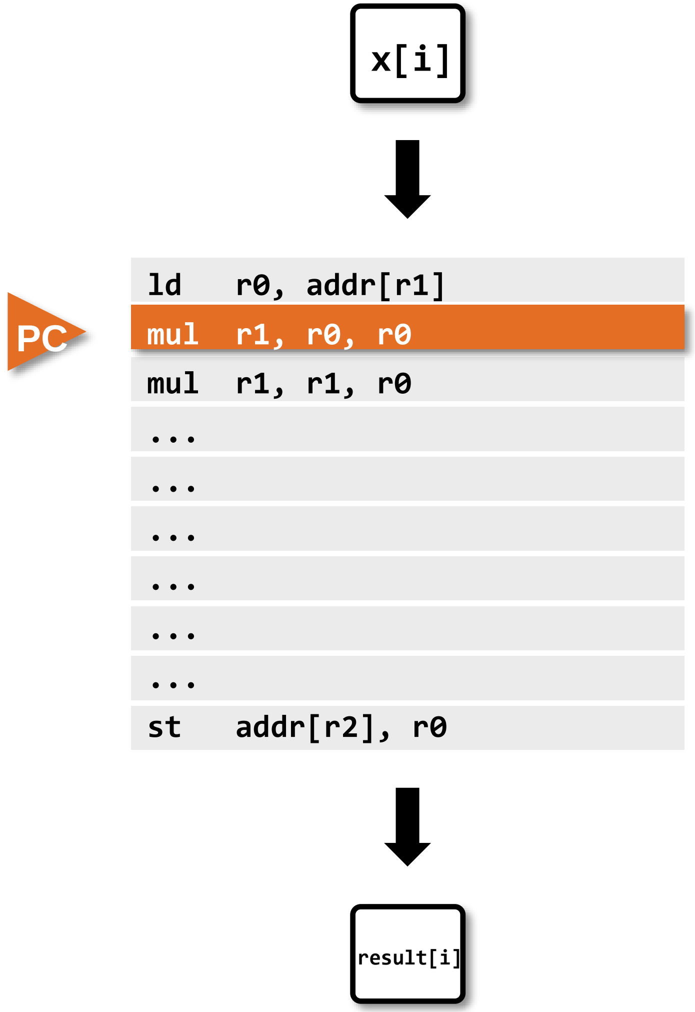


```
ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...
...
...
...
...
...
st   addr[r2], r0
```

Execute program

My very simple processor: executes one instruction per clock

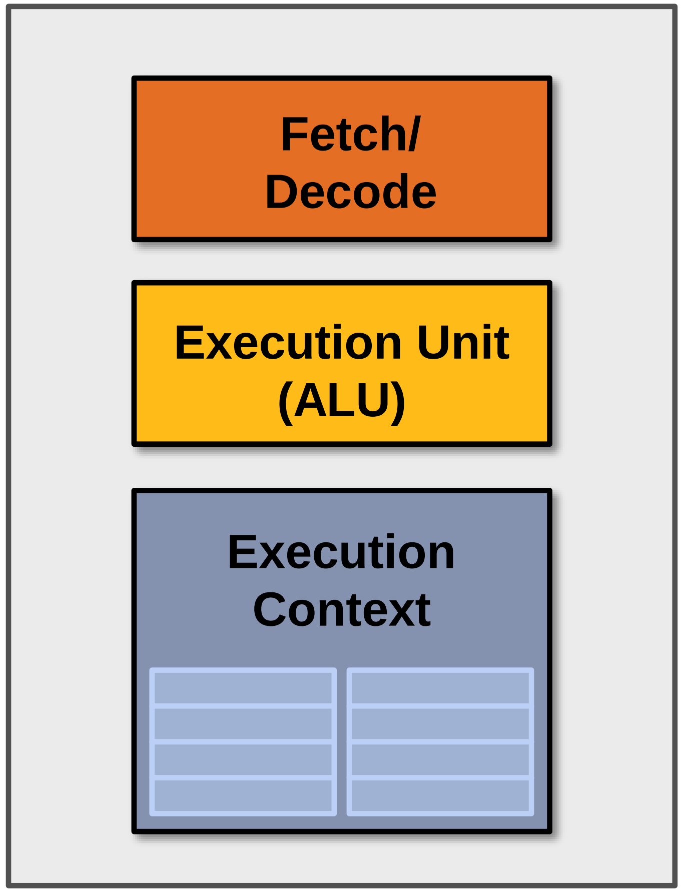


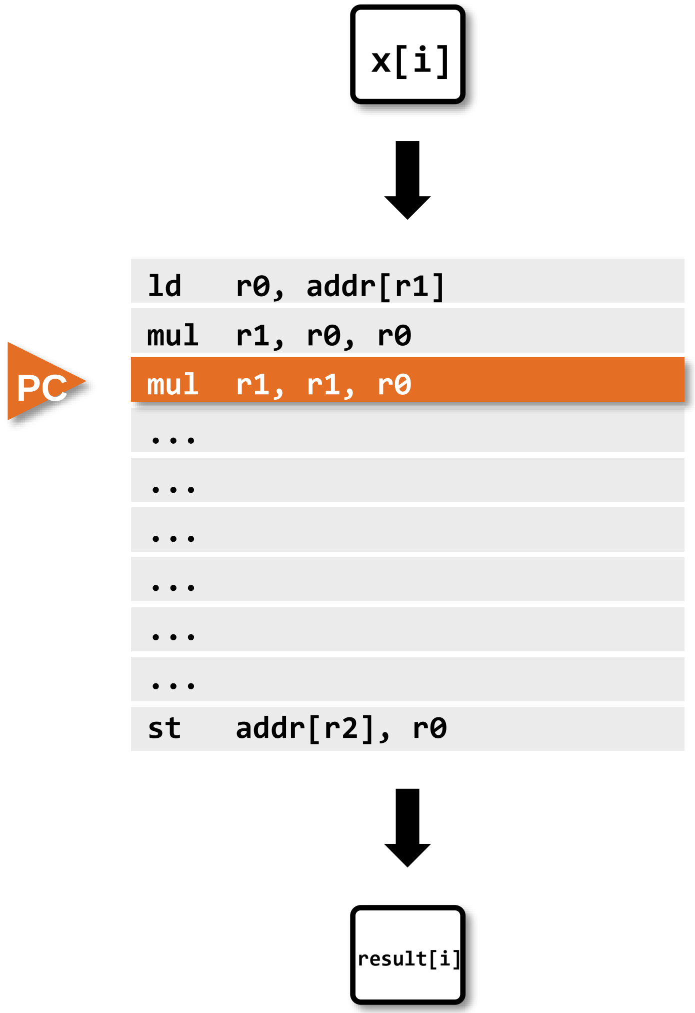


```
ld    r0, addr[r1]
mul   r1, r0, r0
mul   r1, r1, r0
...
...
...
...
...
...
st    addr[r2], r0
```

Execute program

My very simple processor: executes one instruction per clock

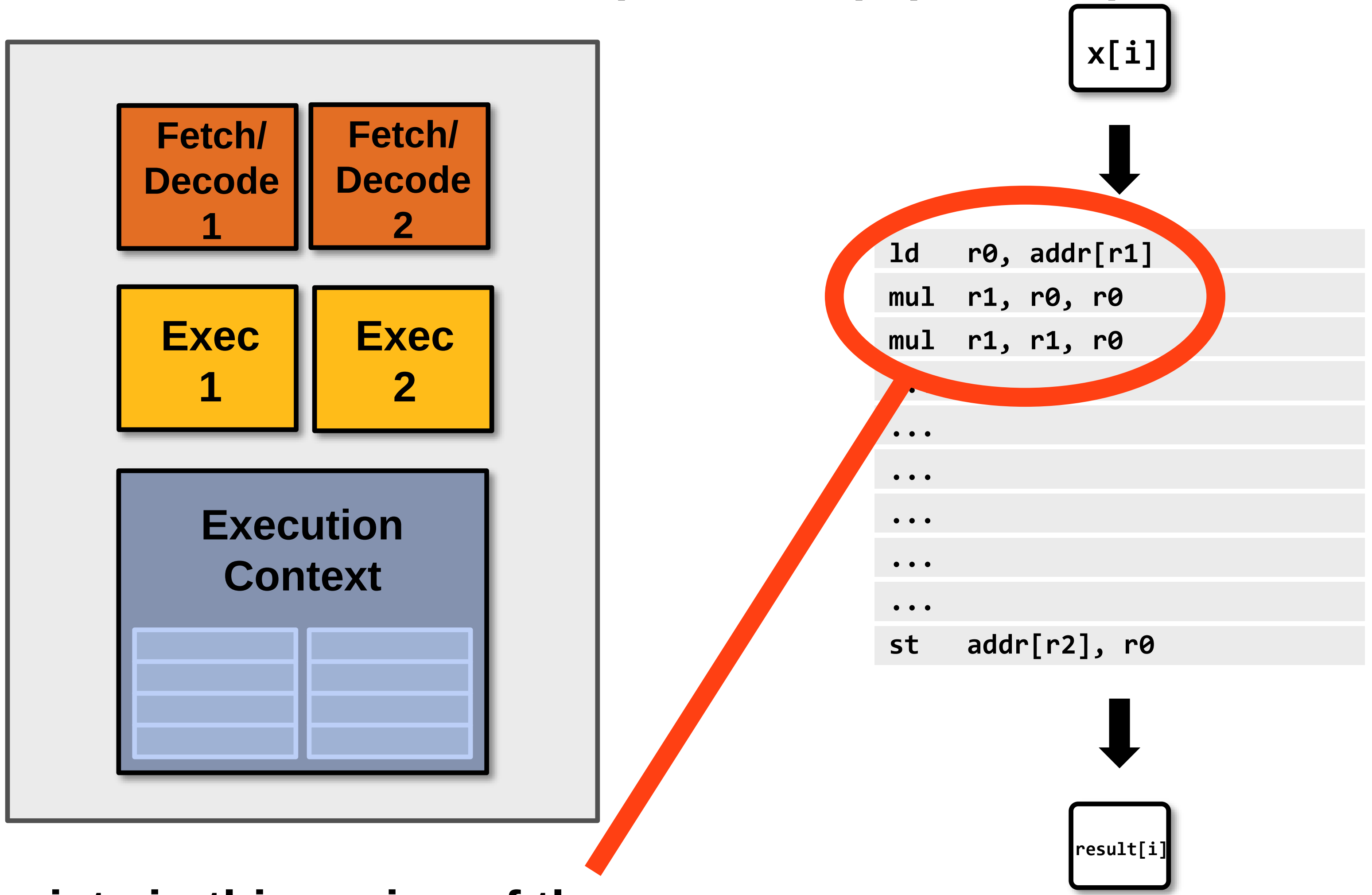




```
ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...
...
...
...
...
...
st   addr[r2], r0
```

Superscalar processor

Recall from the previous: instruction level parallelism (ILP)
Decode and execute two instructions per clock (if possible)



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

Note: No ILP exists in this region of the program

Aside: Pentium 4

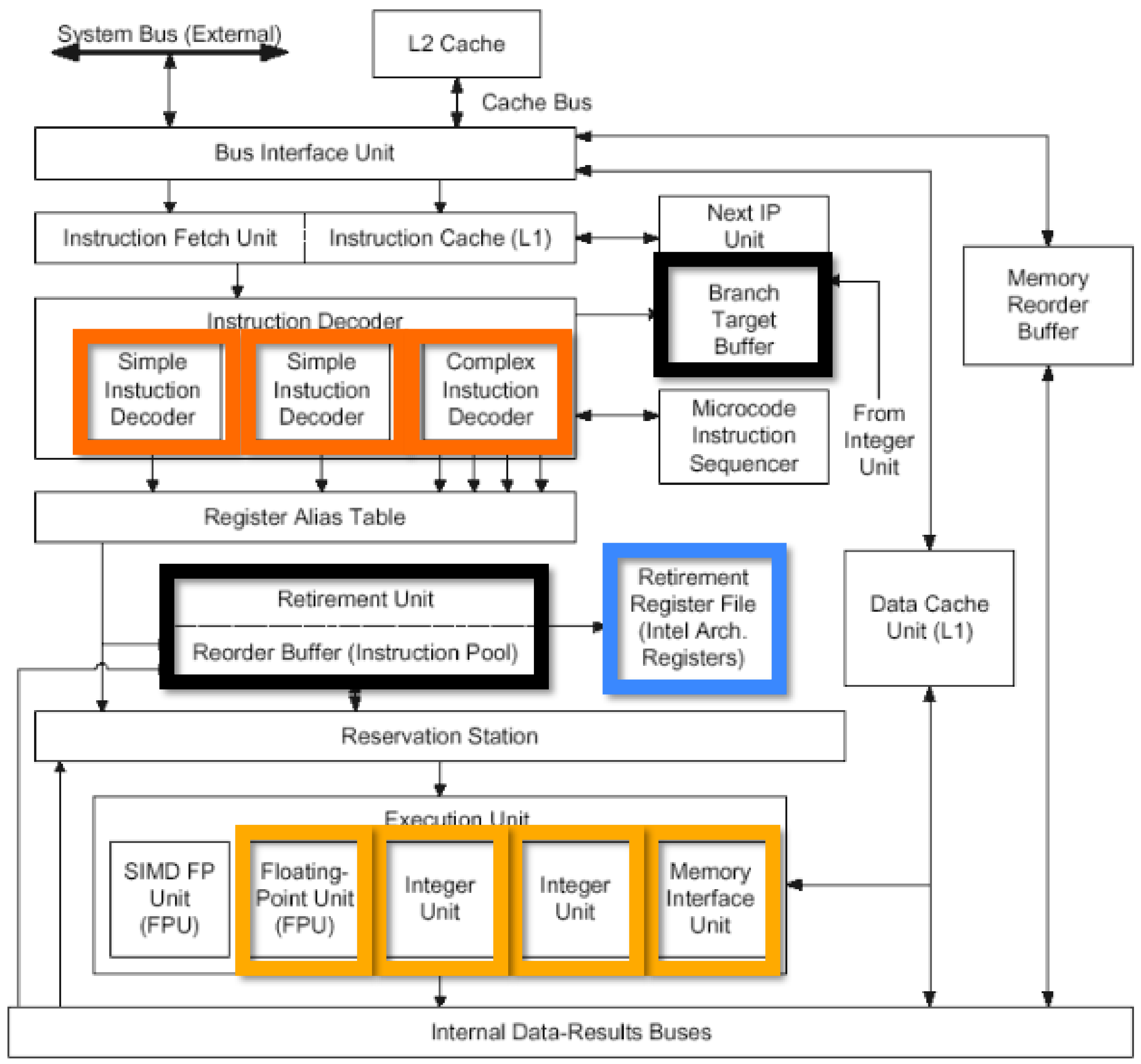
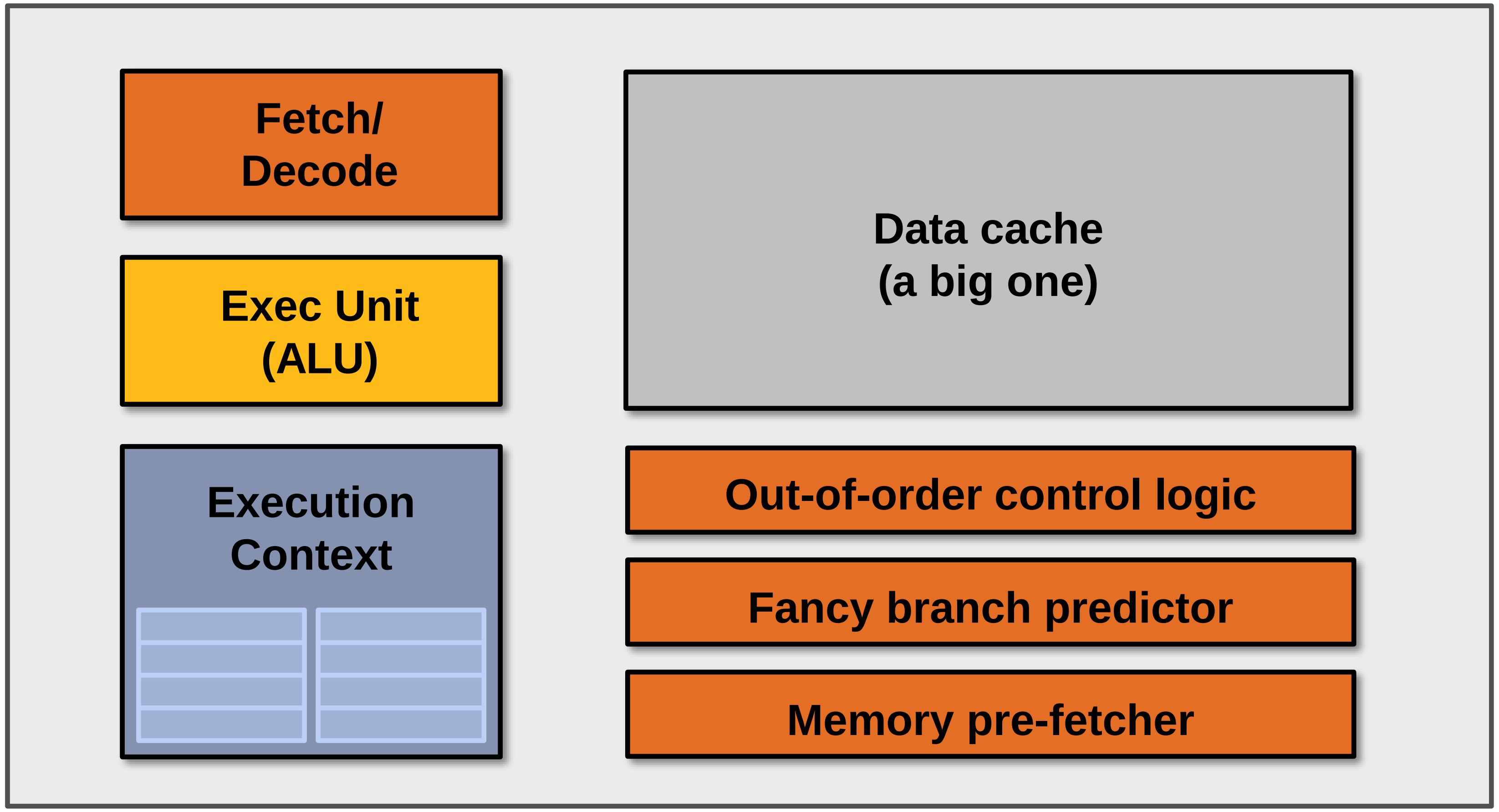


Image credit: http://ixbtlabs.com/articles/pentium4/index.html

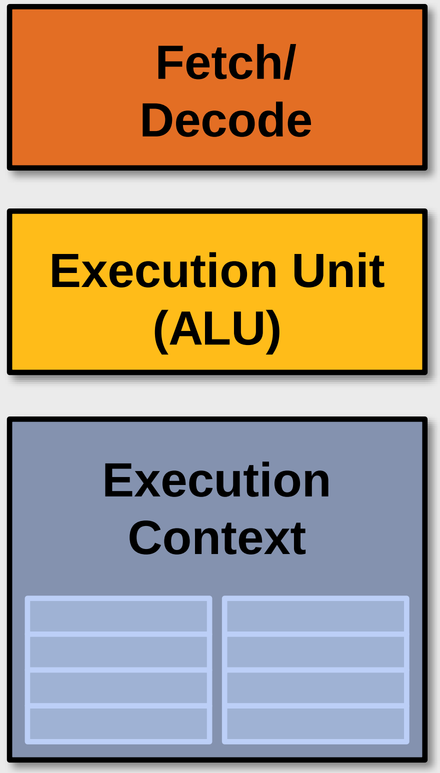
Processor: pre multi-core era

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



More transistors = larger cache, smarter out-of-order logic, smarter branch predictor, etc.
(Also: more transistors → smaller transistors → higher clock frequencies)

Processor: multi-core era (since 2005)

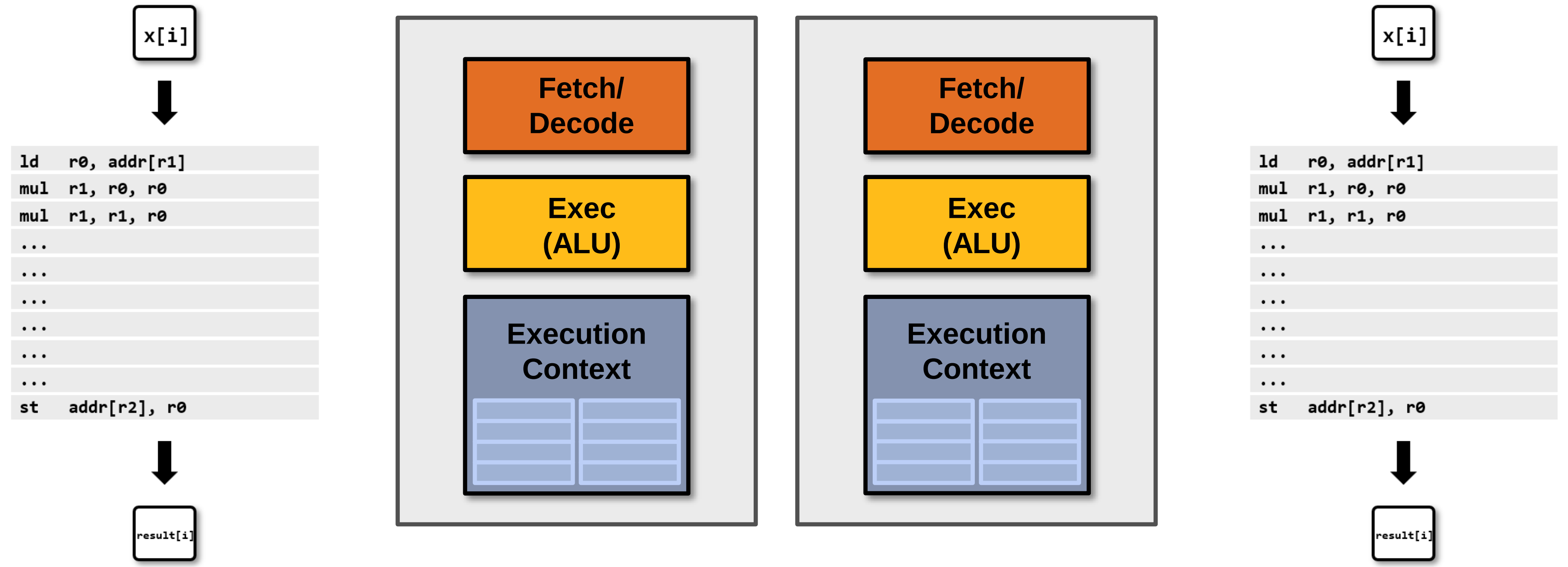


Idea #1:

Use increasing transistor count to add more cores to the processor

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

Two cores: compute two elements in parallel



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

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

Simpler cores: each core is 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* result)
{
   for (int i=0; i<N; i++)
   {
         float value = x[i];
         float numer = x[i] * x[i] * x[i];
         int denom = 6;  // 3!
         int sign = -1;

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

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

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

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

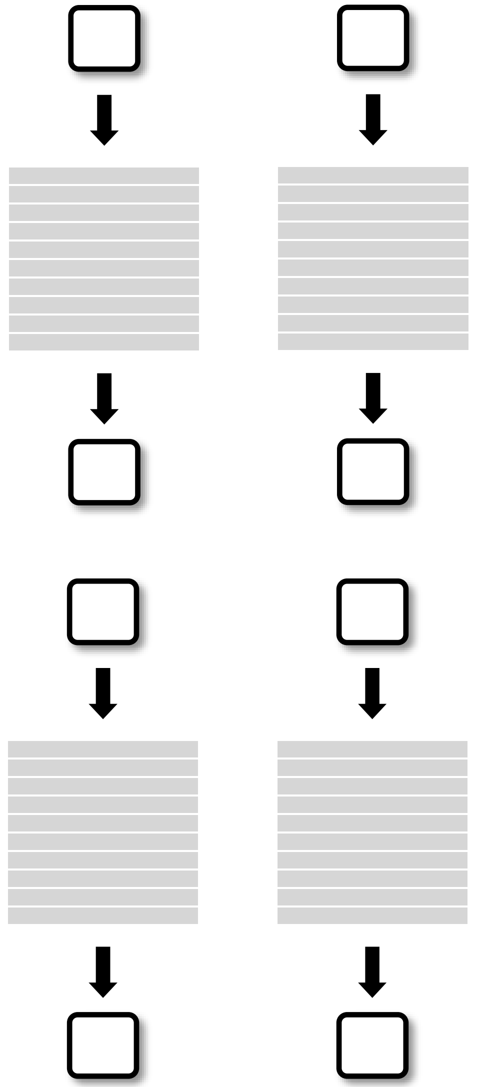
Using Cilk to provide parallelism

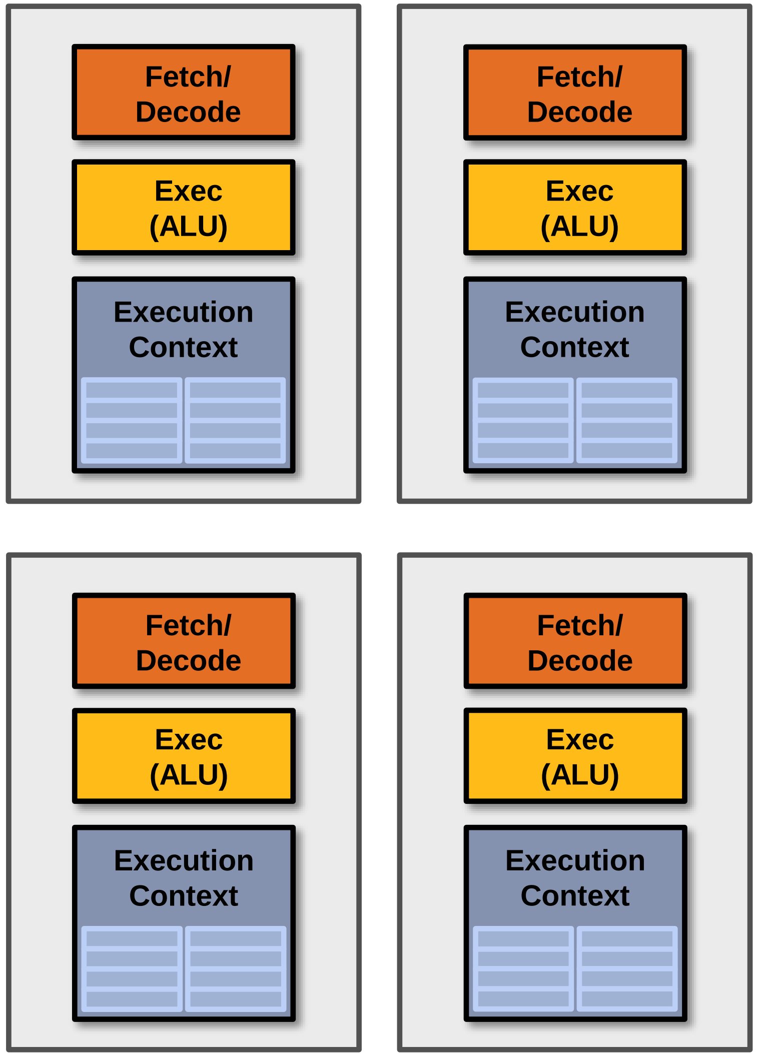
```
void sinx(int N, int terms, float* x, float* result)
{
   cilk_for (int i=0; i<N; i++)
   {
         float value = x[i];
         float numer = x[i] * x[i] * x[i];
         int denom = 6;  // 3!
         int sign = -1;

         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;
   }
}
```

Loop iterations declared by the programmer to be independent

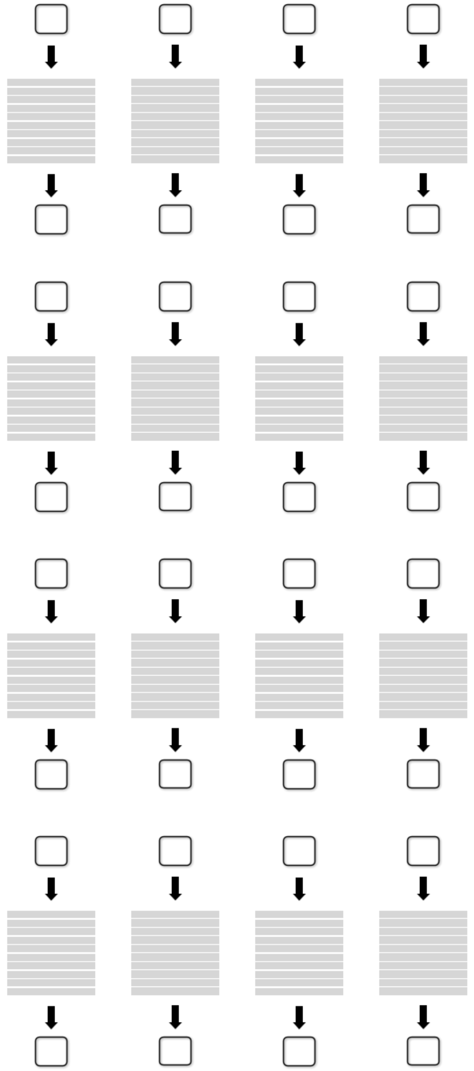
With this information, you could imagine how a compiler might automatically generate parallel threaded code

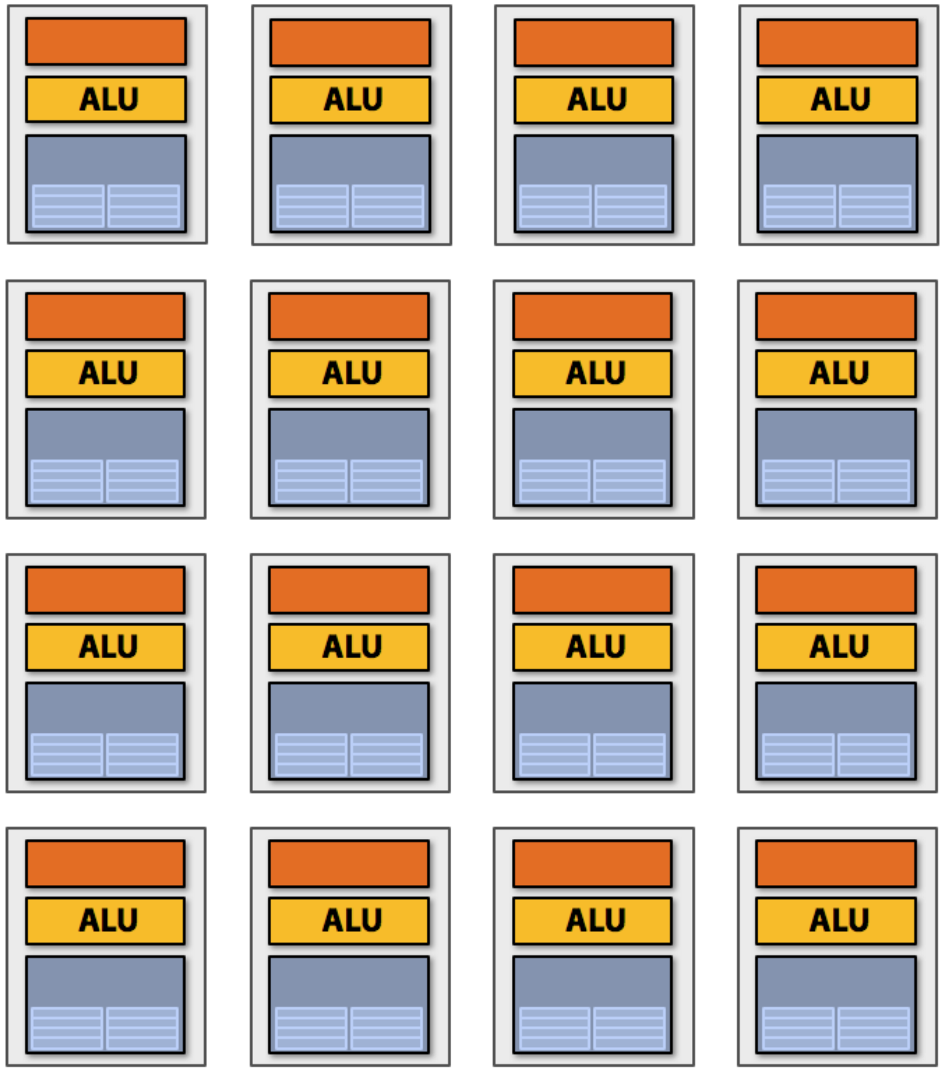
Four cores: compute four elements in parallel





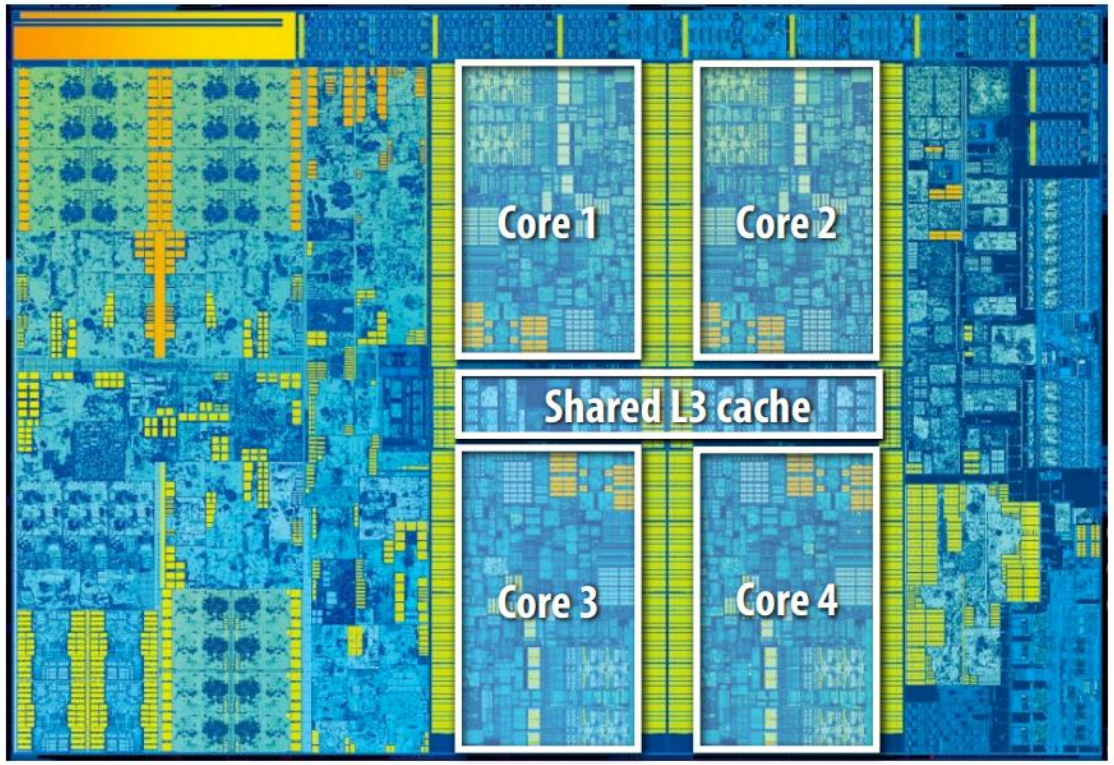
Sixteen cores: compute sixteen elements in parallel



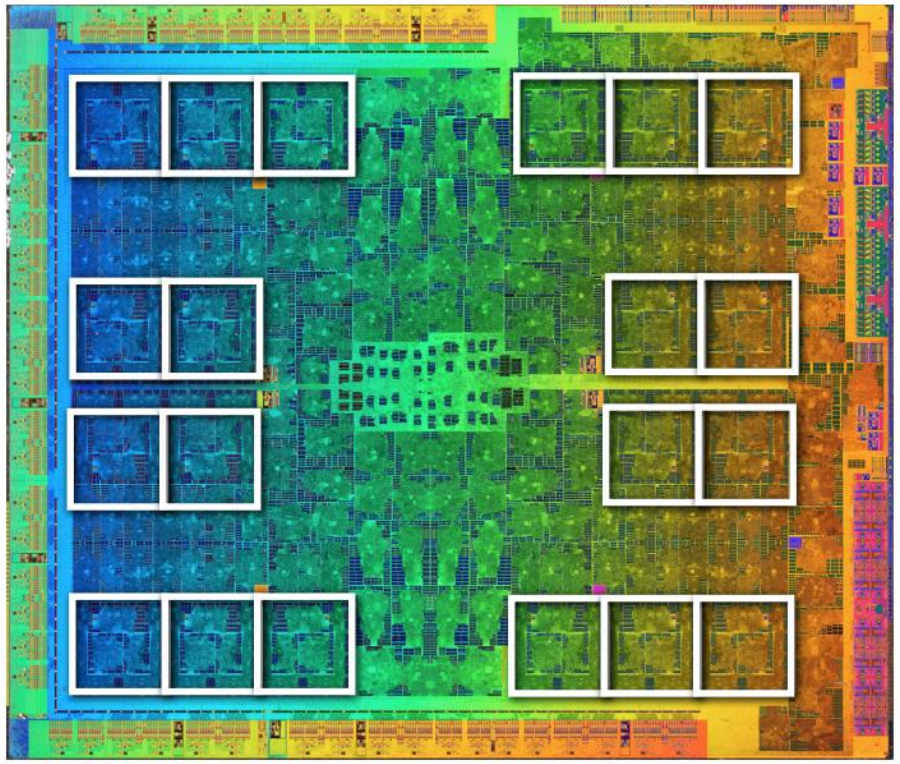


Sixteen cores, sixteen simultaneous instruction streams

Multi-core examples

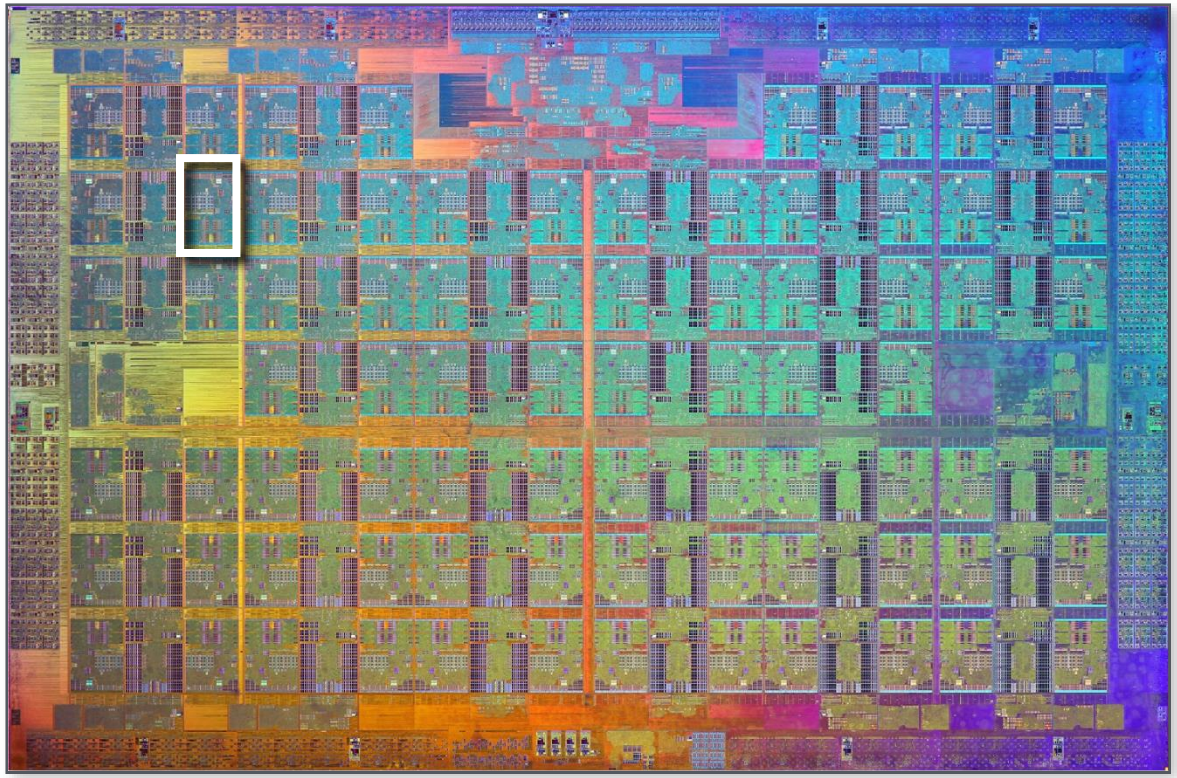


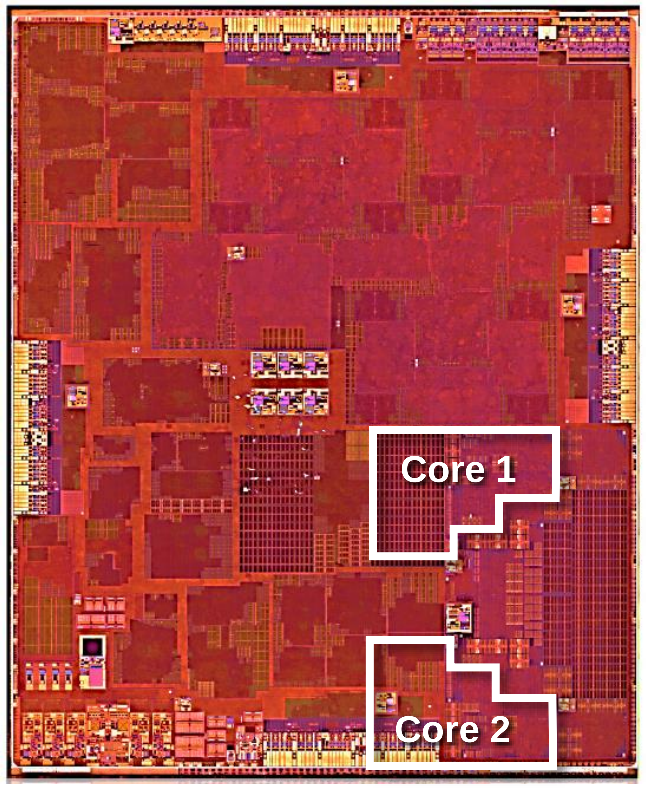
Intel “Skylake” Core i7 quad-core CPU
(2015)



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

More multi-core examples


Intel Xeon Phi “Knights Corner” 72-core CPU (2016)



Apple A9 dual-core CPU (2015)

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

Data-parallel expression

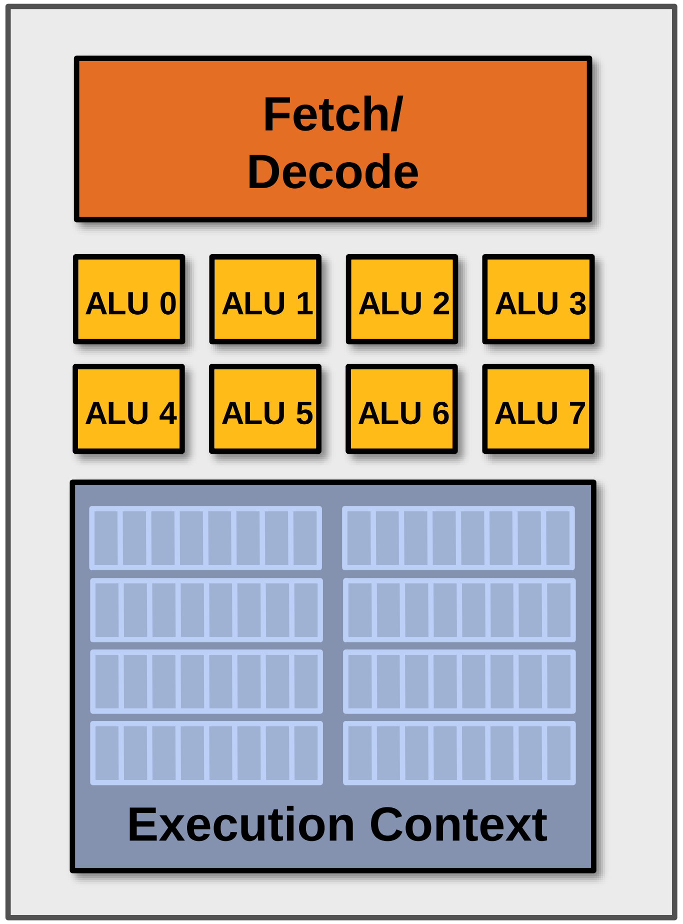
```
void sinx(int N, int terms, float* x, float* result)
{
   cilk_for (int i=0; i<N; i++)
   {
         float value = x[i];
         float numer = x[i] * x[i] * x[i];
         int denom = 6;  // 3!
         int sign = -1;

         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;
   }
}
```

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, but on different input data
(to compute the sine of the input number)

Add ALUs to increase compute capability



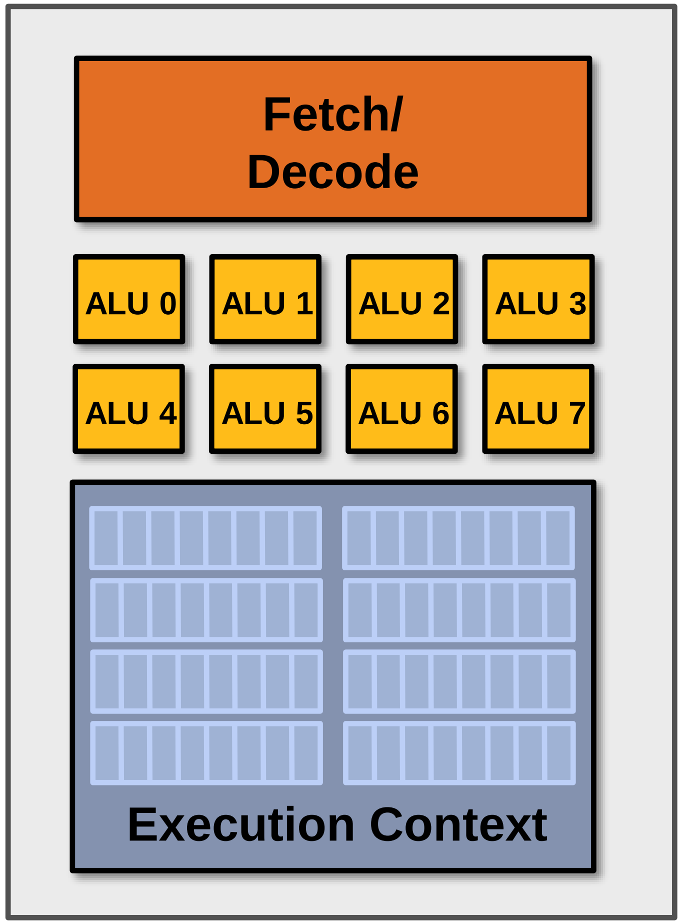
Idea #2:
Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

Single instruction, multiple data

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

Add ALUs to increase compute capability



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

Recall original compiled program:

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

Scalar program

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

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

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

Original compiled program:

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

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

Vector program (using AVX intrinsics)

Intrinsics available to C programmers

```
#include <immintrin.h>

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

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

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

Vector program (using AVX intrinsics)

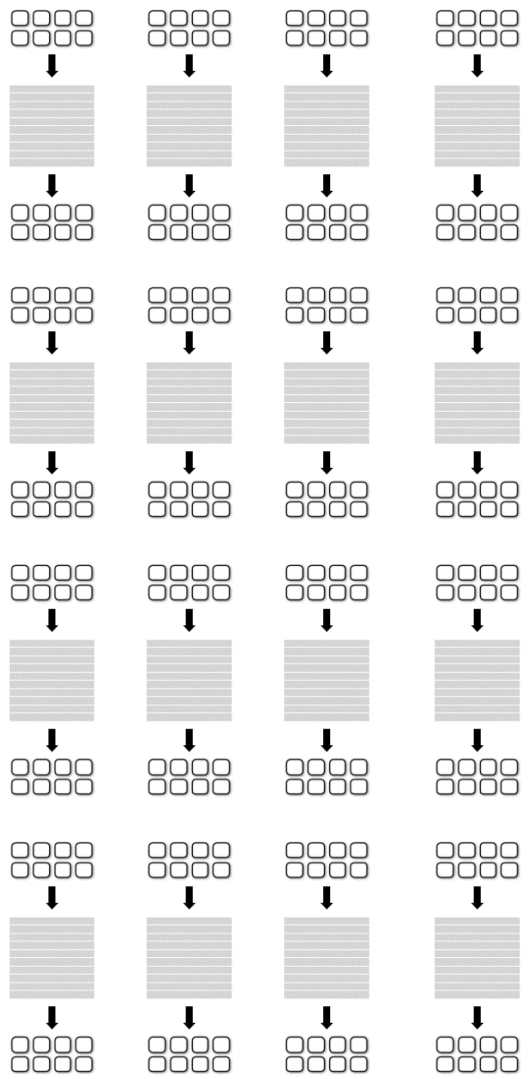
```
#include <immintrin.h>

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

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

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

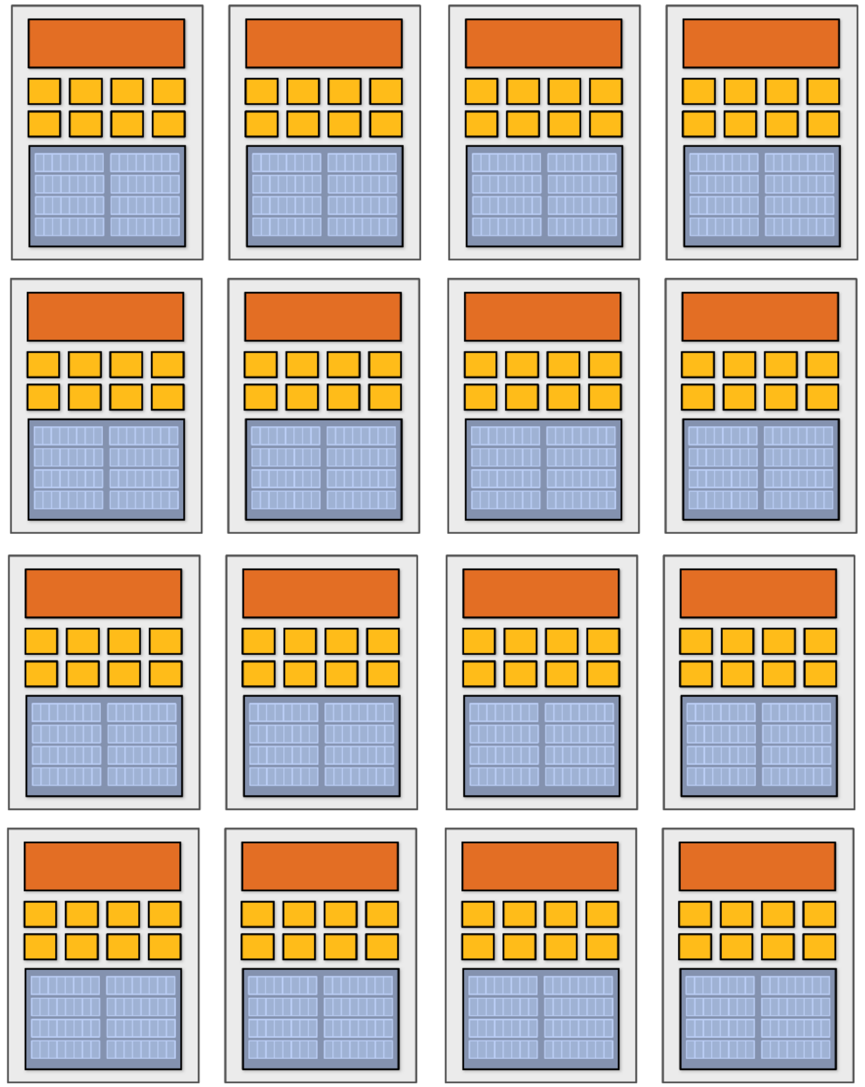
```
vloadps   xmm0, addr[r1]
vmulps    xmm1, xmm0, xmm0
vmulps    xmm1, xmm1, xmm0
...
...
...
...
...
...
vstoreps  addr[xmm2], xmm0
```

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

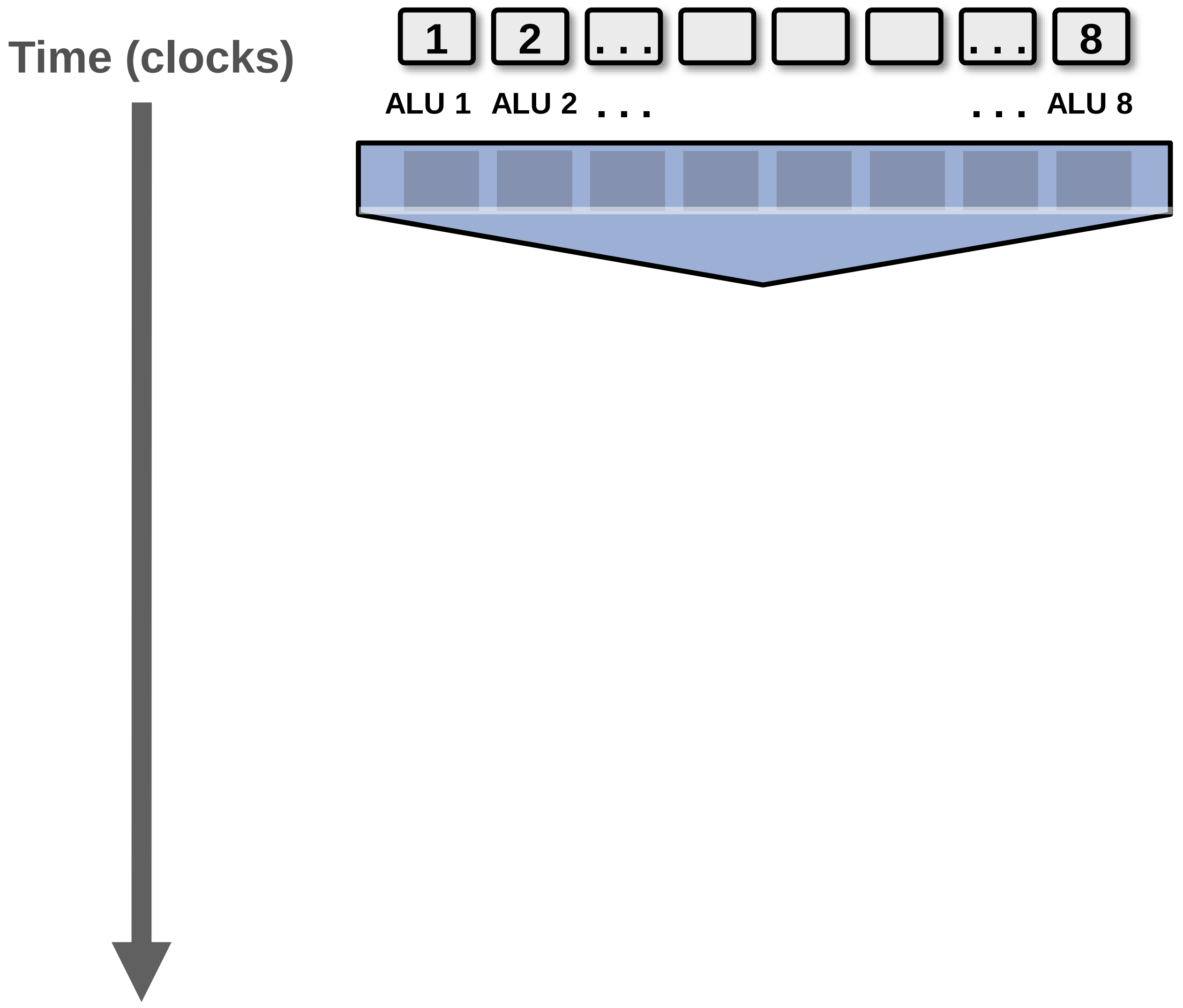
Data-parallel expression

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

Compiler understands loop iterations are independent, and that same loop body will be executed on a large number of data elements.

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

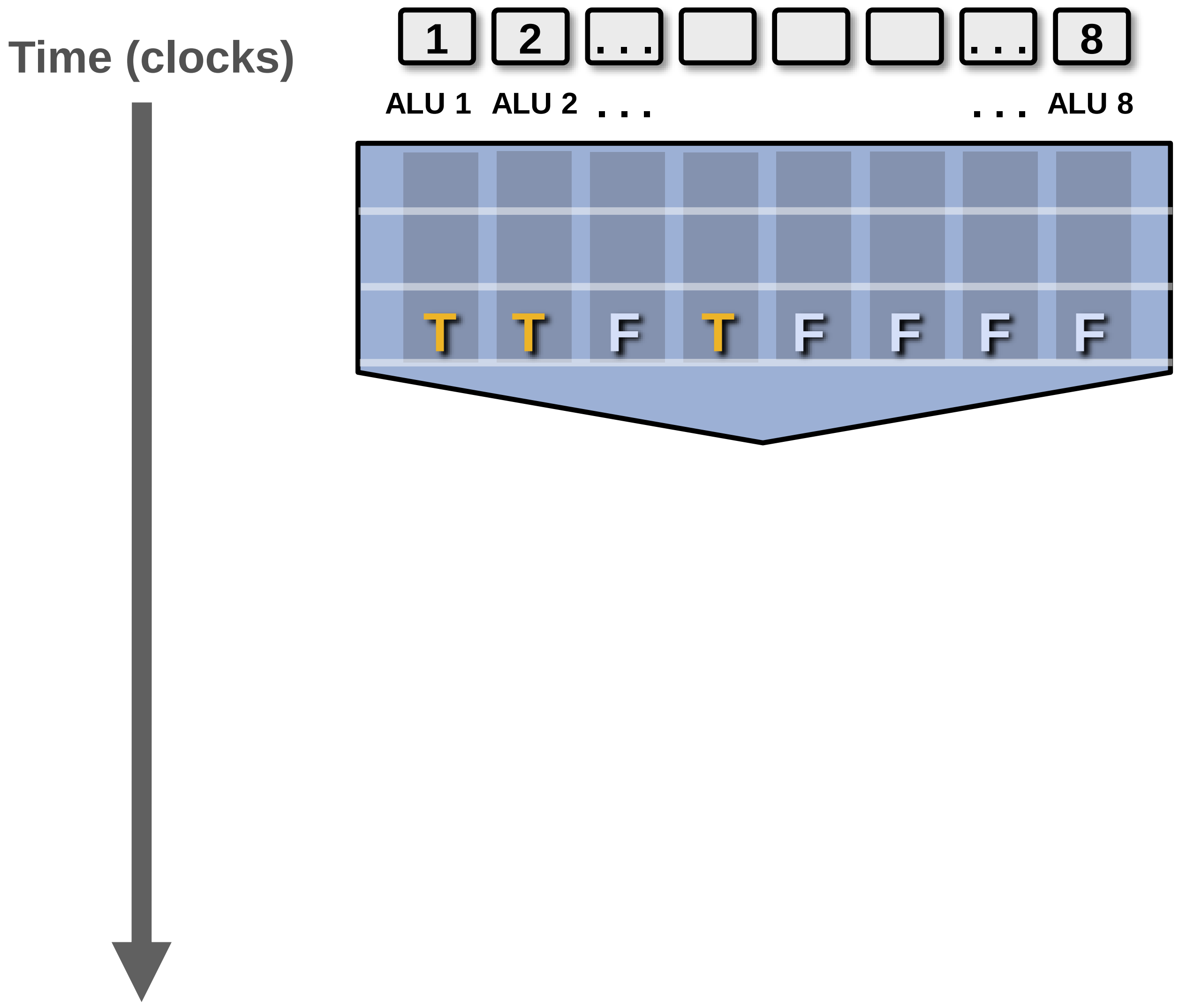
What about conditional execution?



(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;
```

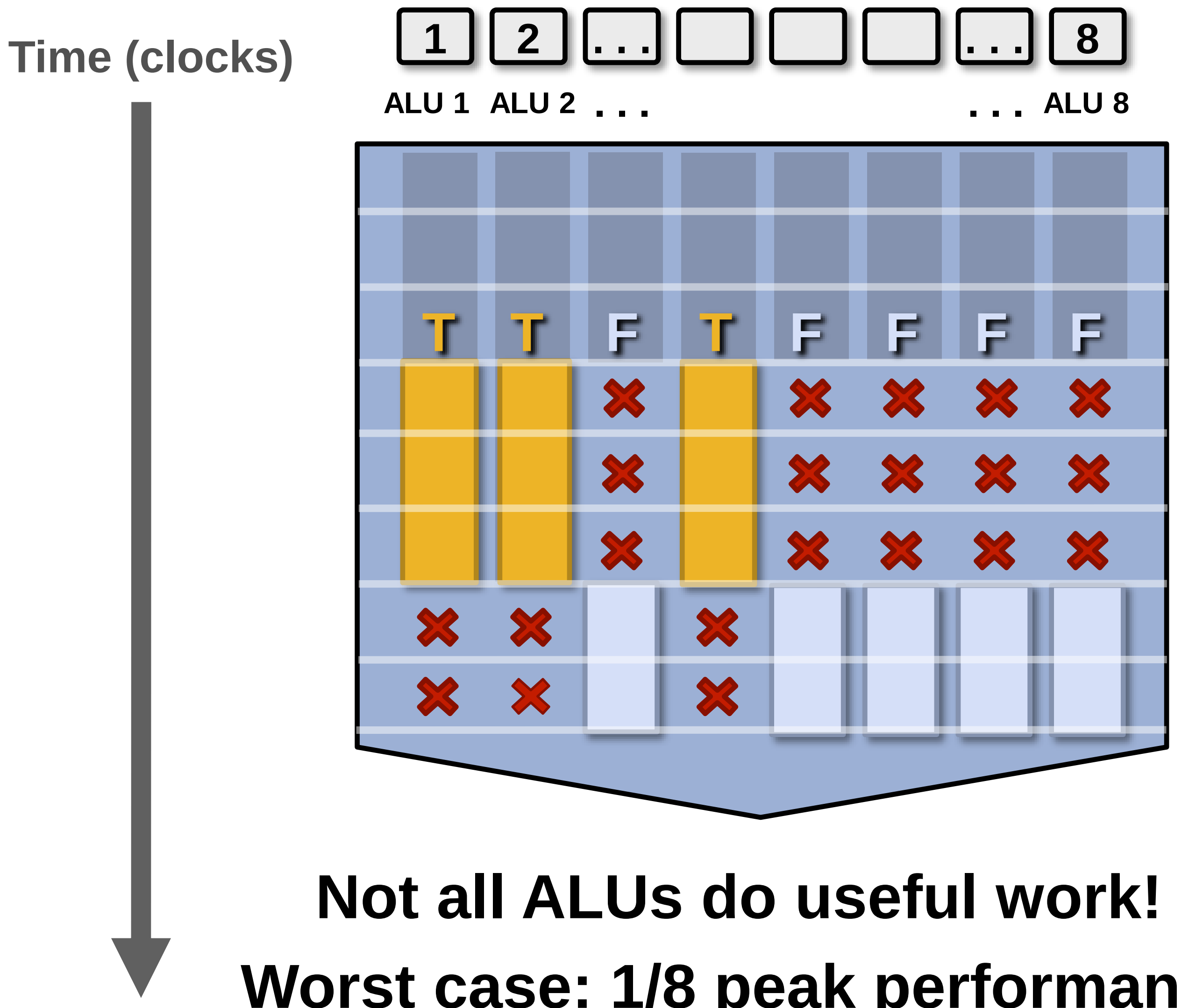
What about conditional execution?



(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



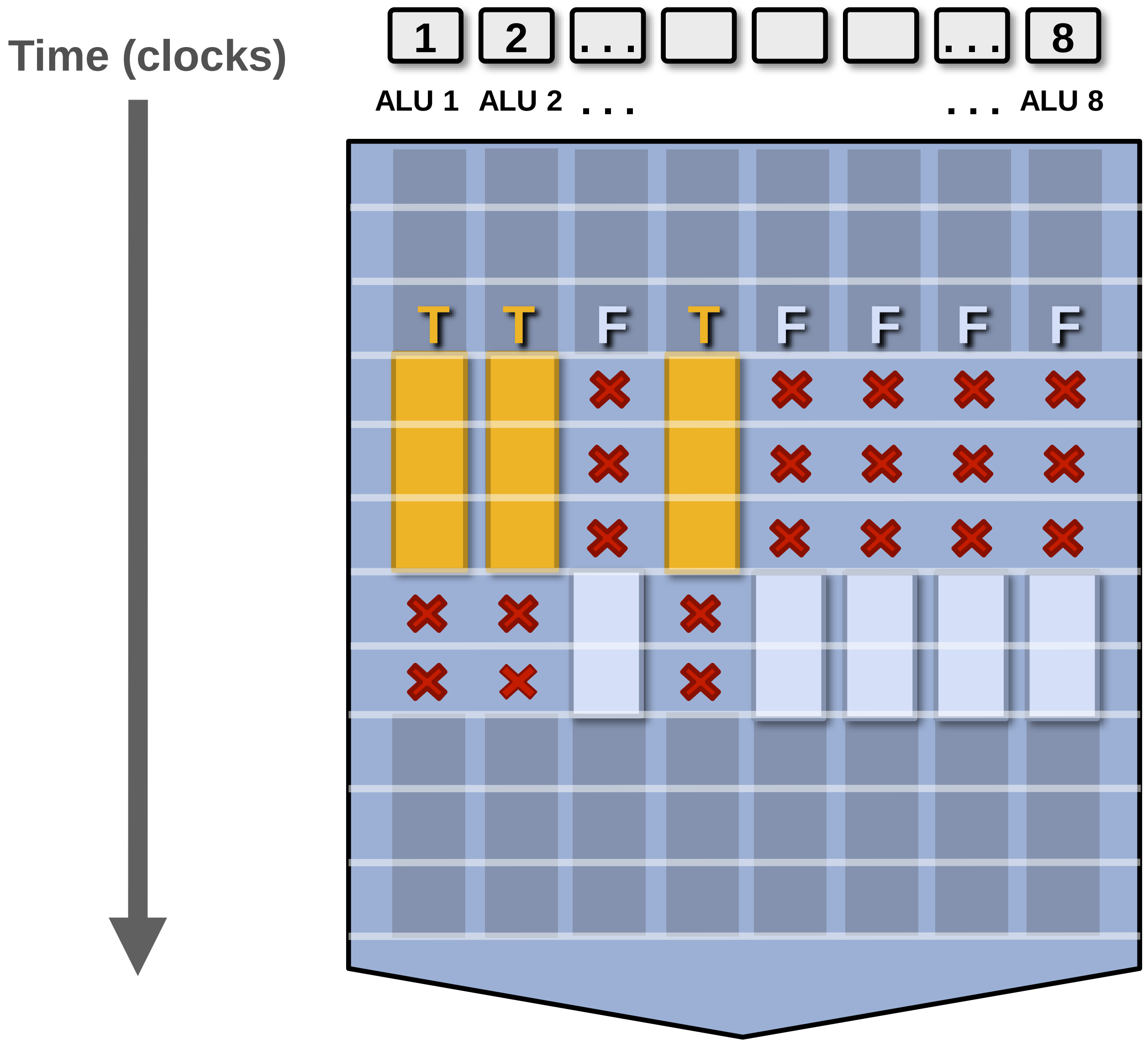
Not all ALUs do useful work!

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;
```

SIMD execution on modern CPUs

- **SSE instructions: 128-bit operations: 4x32 bits or 2x64 bits (4-wide float vectors)**
- **AVX2 instructions: 256 bit operations: 8x32 bits or 4x64 bits (8-wide float vectors)**
- **AVX512 instruction: 512 bit operations: 16x32 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 (hard problem, even best compilers are not great on arbitrary C/C++ code)

- **Terminology: "explicit SIMD": SIMD parallelization is performed at compile time**
  - Can inspect program binary and see instructions (vstoreps, vmulps, etc.)

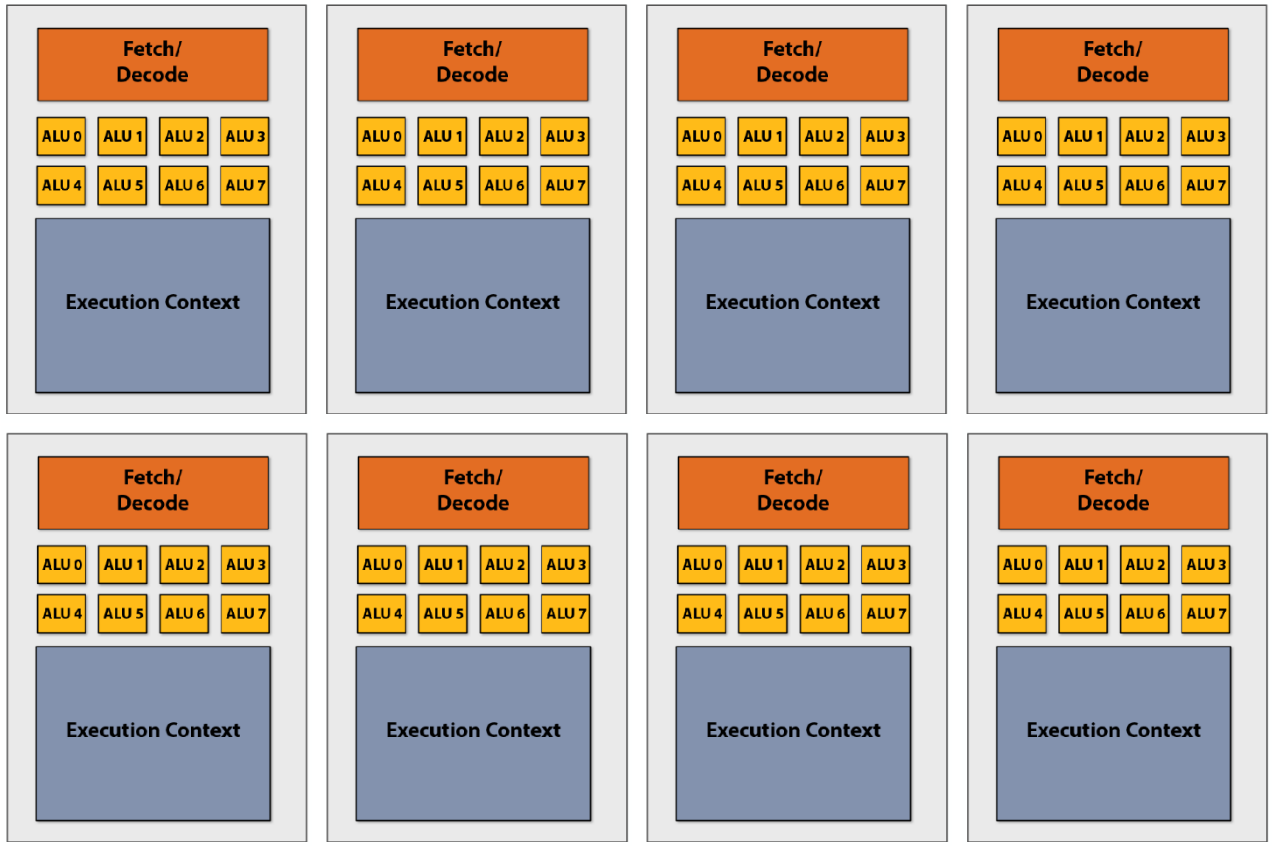
SIMD execution on many modern GPUs

- **“Implicit SIMD”**
  - Compiler generates a scalar binary (scalar instructions)
  - But N instances of the program are *always run* together on the processor
    execute(my_function, N) // execute my_function N times
  - In other words, the interface to the hardware itself is data parallel
  - Hardware (not compiler) is responsible for simultaneously executing the same instruction from multiple instances on different data on SIMD ALUs

- **SIMD width of most modern GPUs ranges from 8 to 32**
  - Divergence can be a big issue

  (poorly written code might execute at 1/32 the peak capability of the machine!)

Example: eight-core Intel Xeon E5-1660 v4

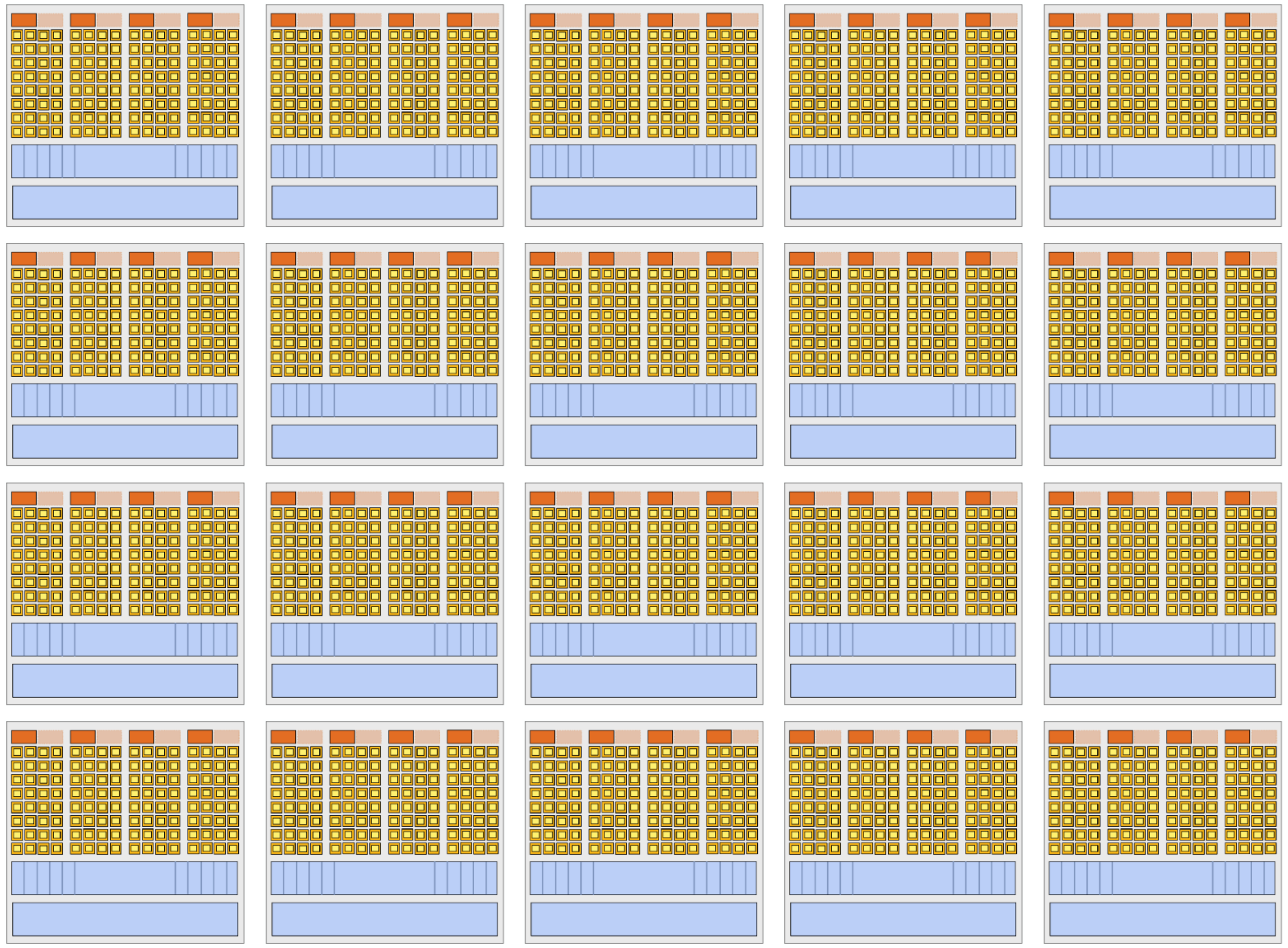


8 cores
8 SIMD ALUs per core (AVX2 instructions)

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

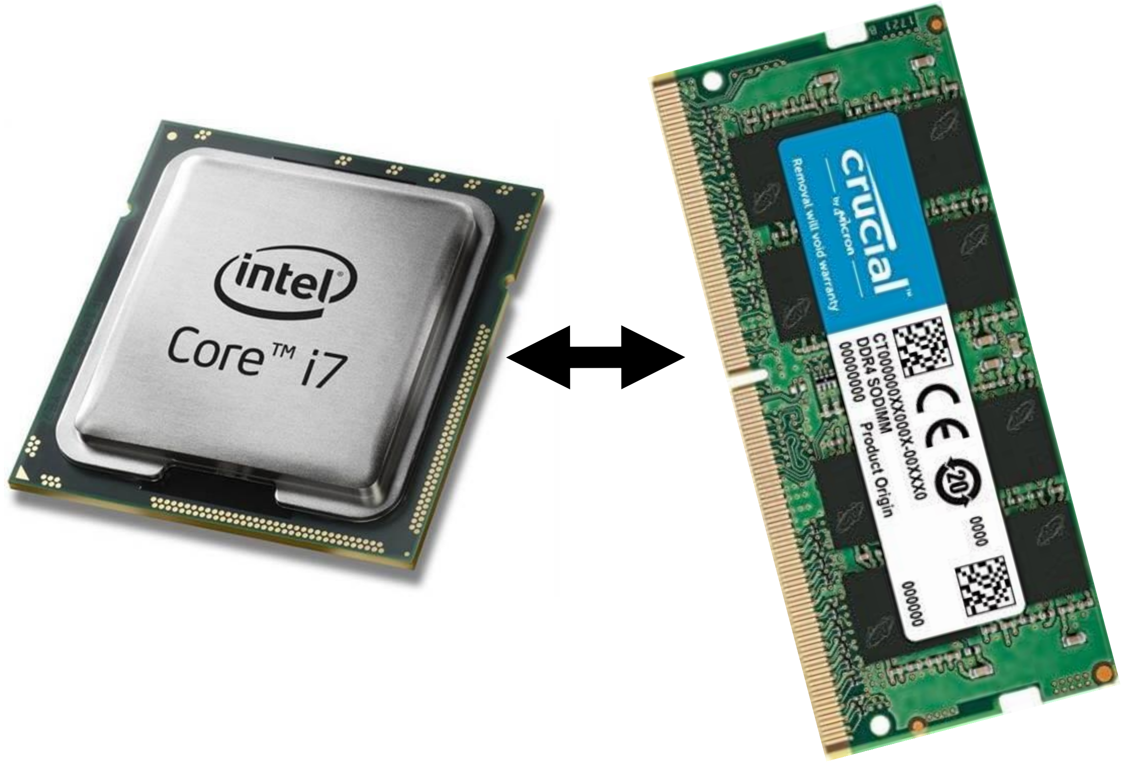


20 cores ("SMs") 128 SIMD ALUs per core (@1.6 GHz) = 8.1 TFLOPs (180 Watts)

Summary: parallel execution

- **Several forms of parallel execution in modern processors**
  - **Multi-core: use multiple processing cores**
    - Provides thread-level parallelism: simultaneously execute a completely different instruction stream on each core
    - Software/algorithms decides when to create threads (e.g., via cilk_spawn, cilk_for)
  - **SIMD: use multiple ALUs controlled by same instruction stream (within a core)**
    - Efficient design for data-parallel workloads: control amortized over many ALUs
    - Vectorization can be done by compiler (explicit SIMD) or at runtime by hardware
    - [Lack of] dependencies is known prior to execution (usually declared by programmer, but can be inferred by loop analysis by advanced compiler)
  - **Superscalar: exploit ILP within an instruction stream. Process different instructions from the <u>same</u> instruction stream in parallel (within a core)**
    - Parallelism automatically and dynamically discovered by the hardware during execution (not programmer visible)

Part 2: Accessing Memory



Terminology

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

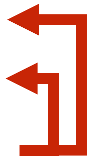
- **Memory bandwidth**
  - The rate at which the memory system can provide data to a processor
  - Example: 20 GB/s

Stalls

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

- Accessing memory is a major source of stalls

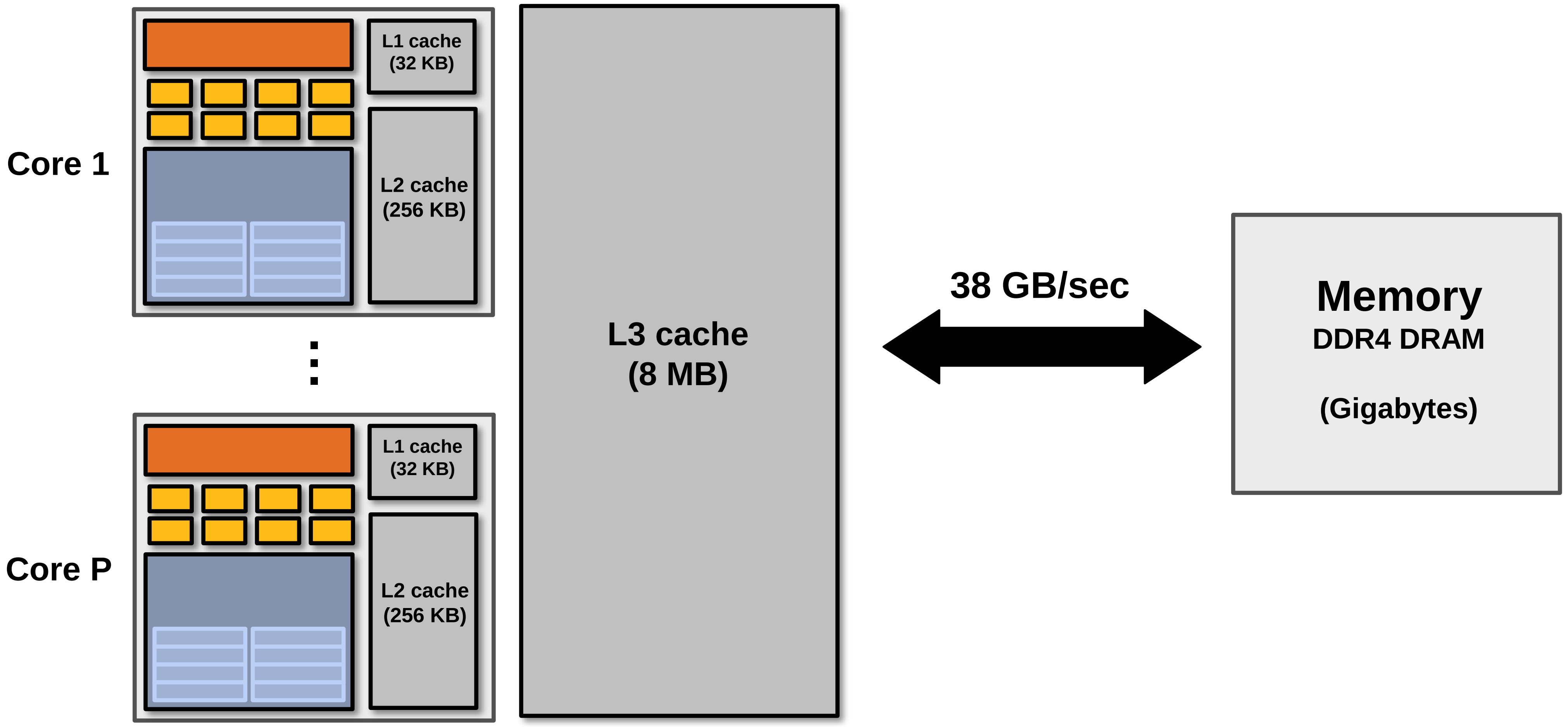
```
ld r0 mem[r2]
ld r1 mem[r3]
add r0, r0, r1
```



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

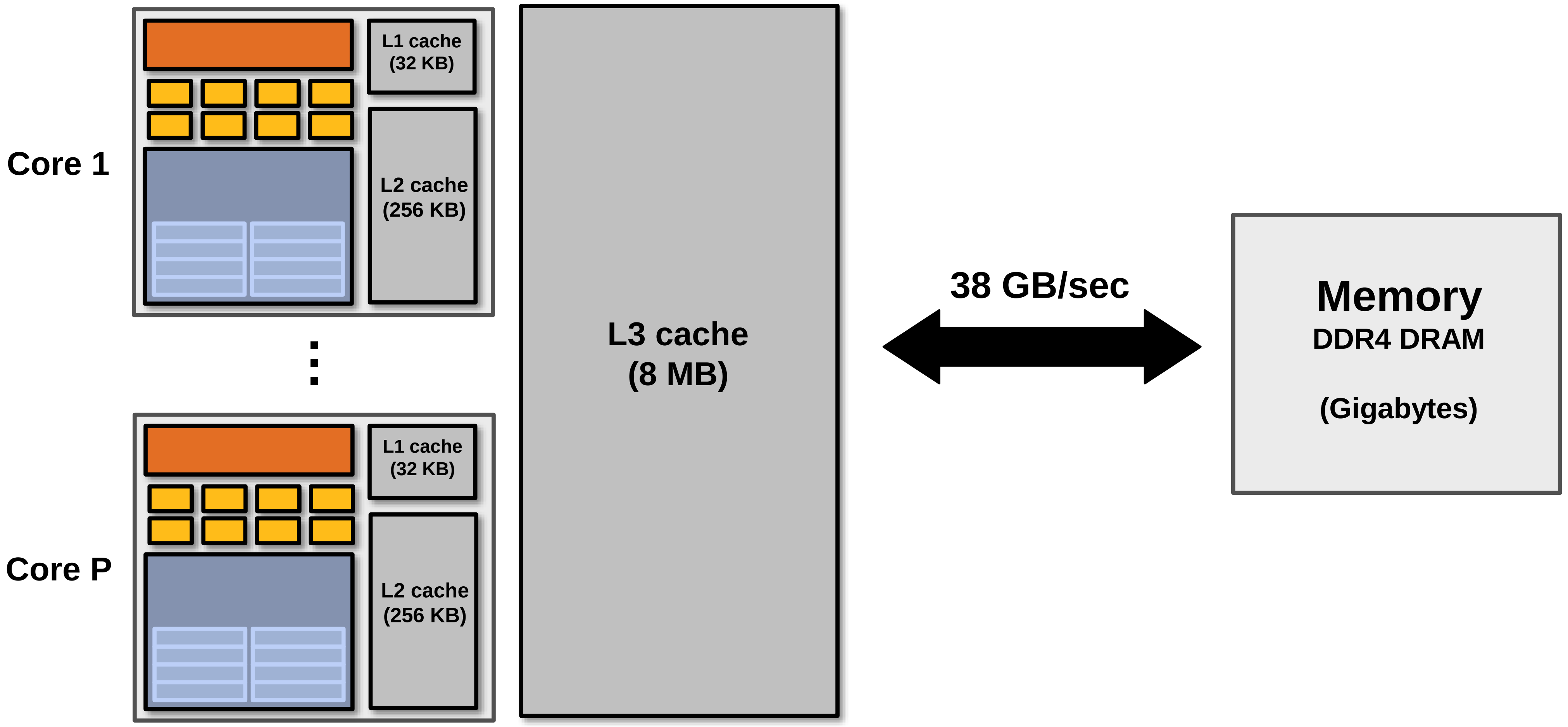
- Memory access times ~ 100's of cycles
  - Memory "access time" is a measure of latency

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 *

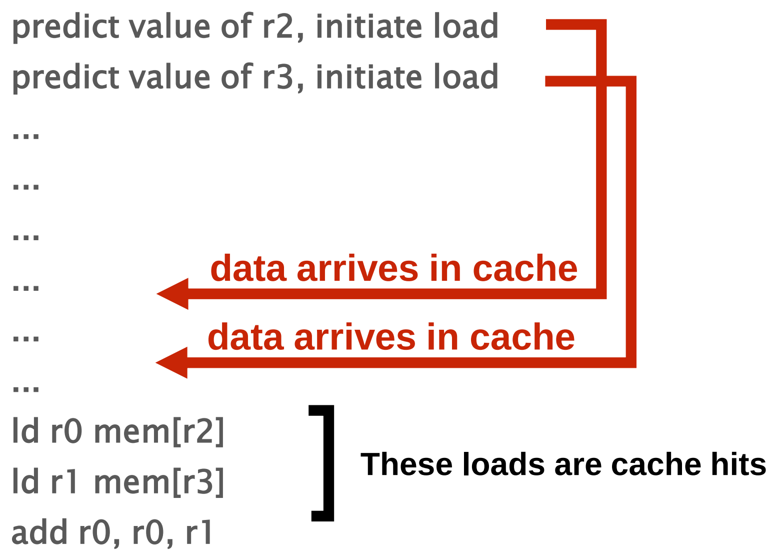


*** Caches also provide high bandwidth data transfer to CPU**

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
```

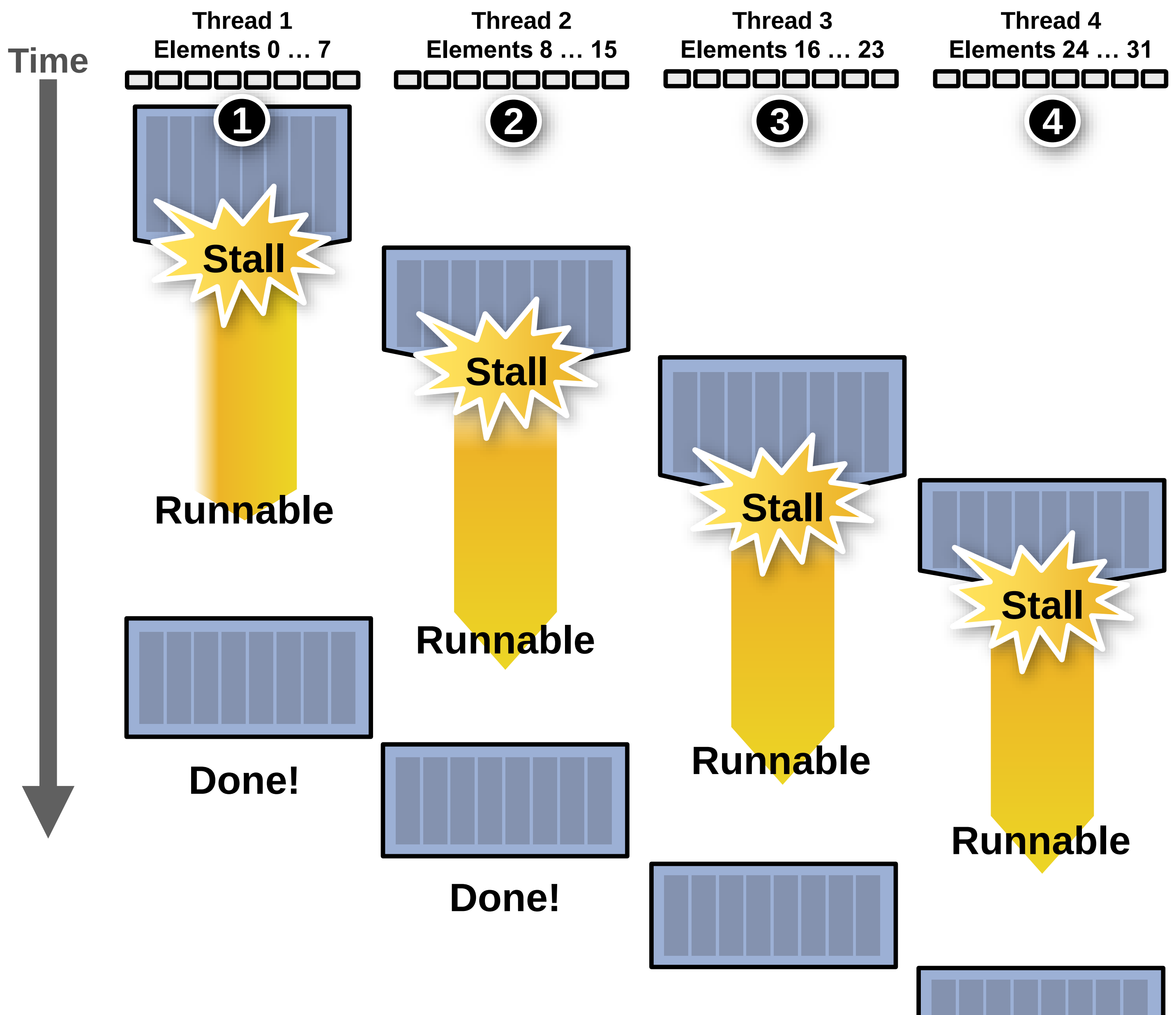
Note: Prefetching can also reduce performance if the guess is wrong (hogs bandwidth, pollutes caches)

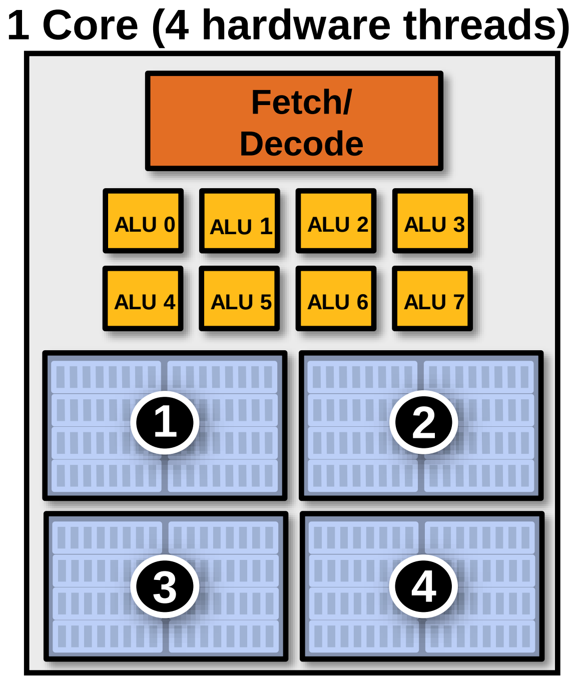
(more detail later in course)

Multi-threading reduces stalls

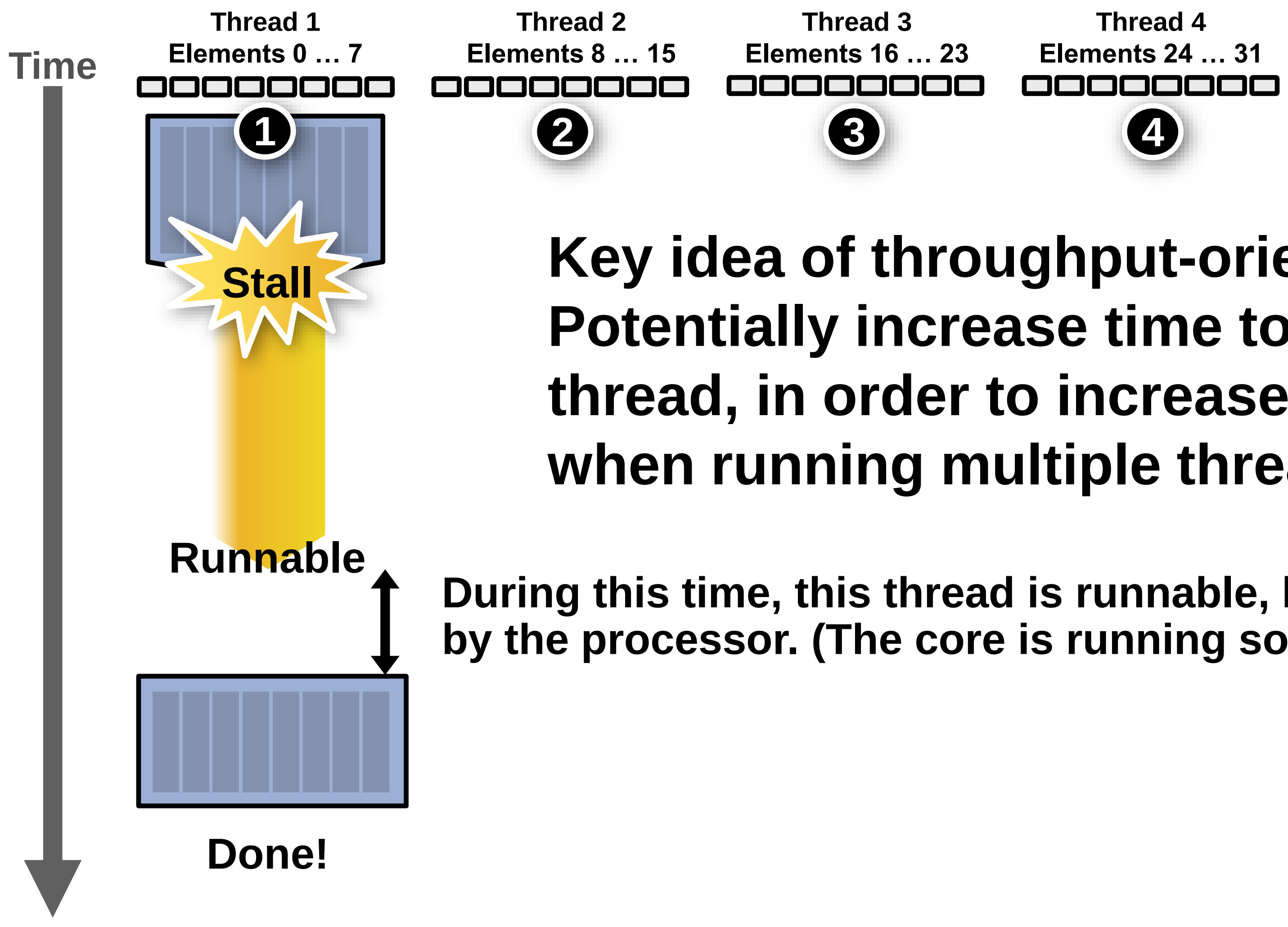
- Idea: interleave processing of multiple threads on the same core to hide stalls
- Like prefetching, multi-threading is a latency hiding, not a latency reducing technique

Hiding stalls with multi-threading





Throughput computing trade-off

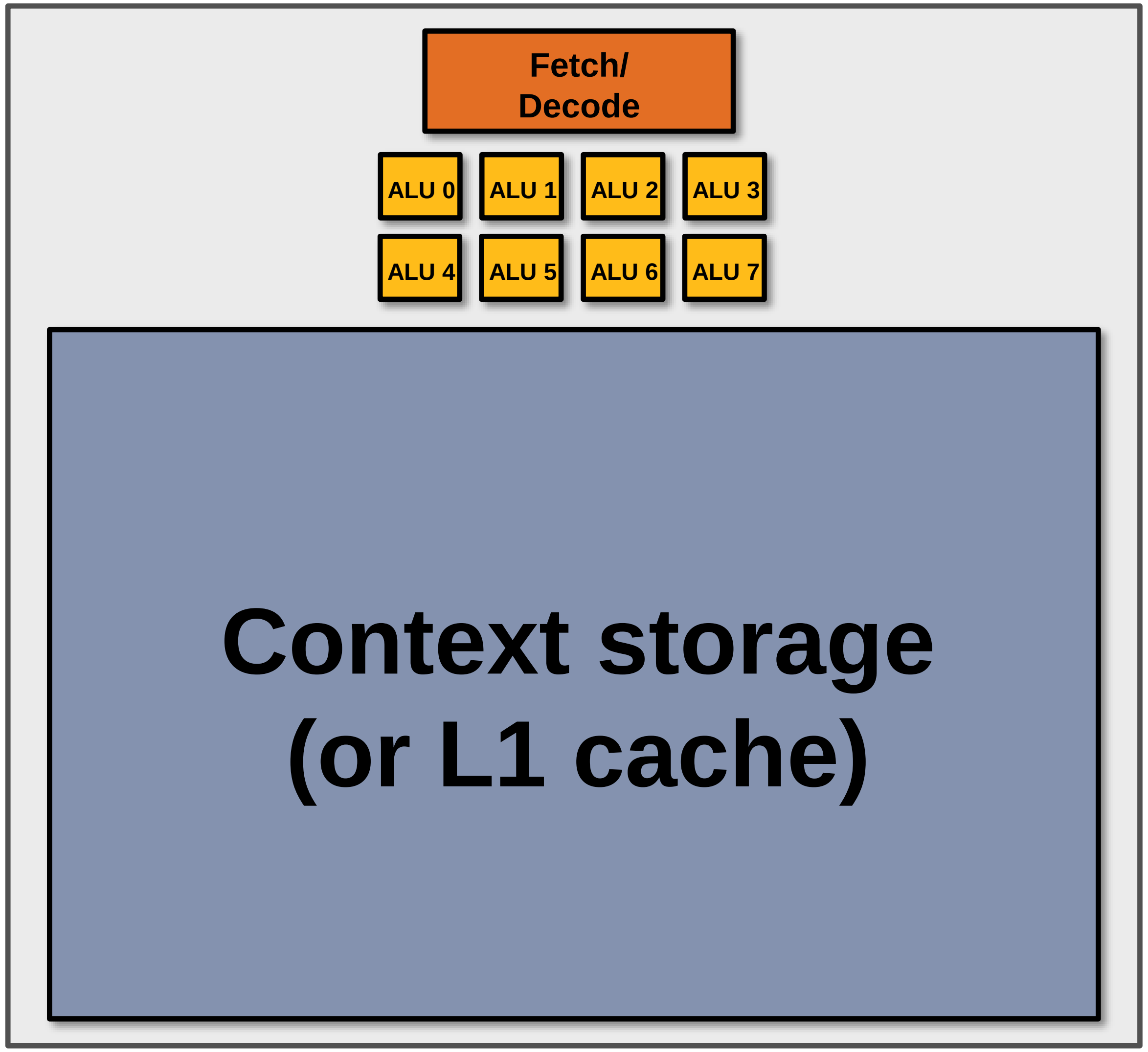


Key idea of throughput-oriented systems: Potentially increase time to complete work by any one thread, in order to increase overall system throughput when running multiple threads.

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

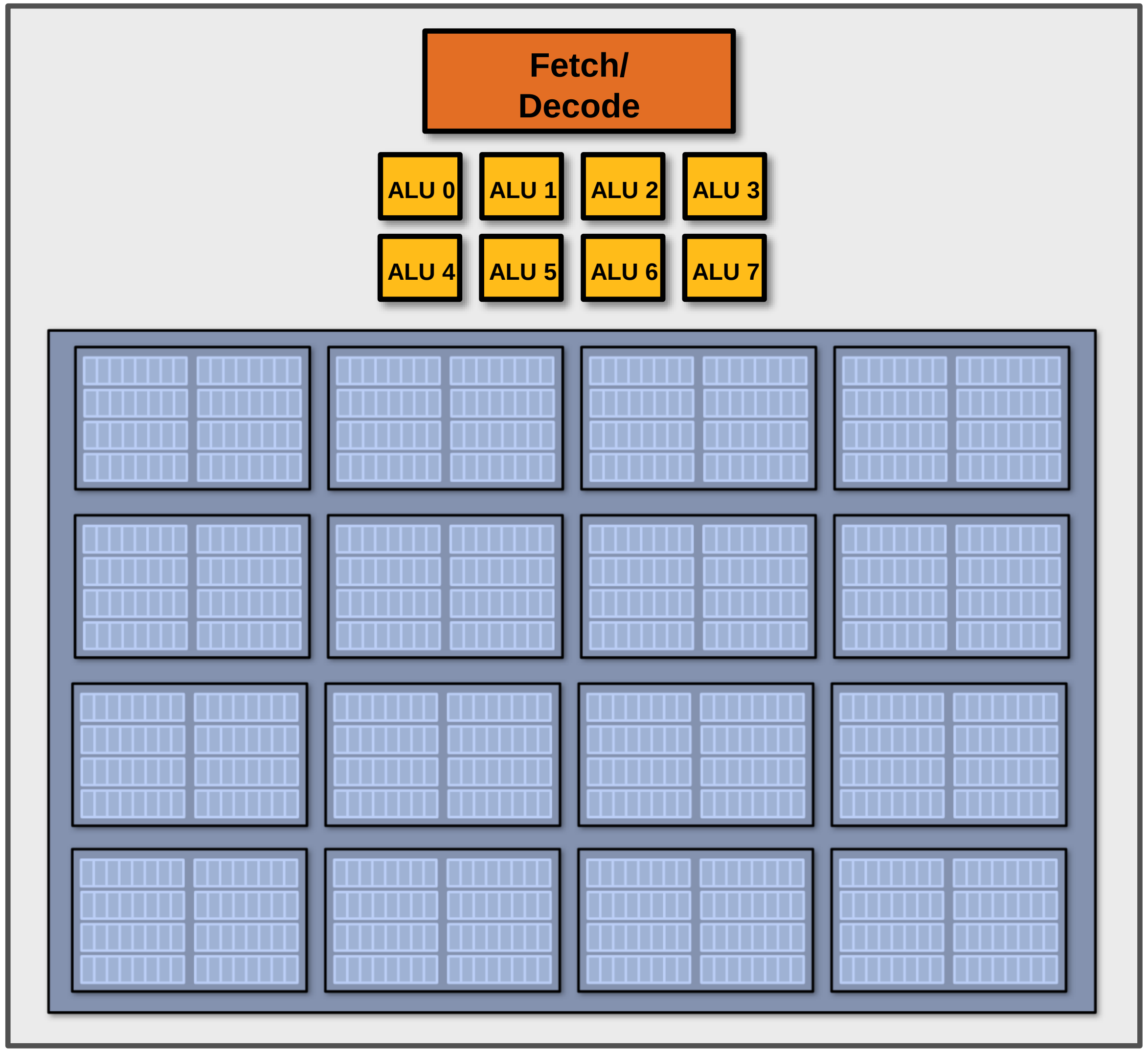
Storing execution contexts

Consider on-chip storage of execution contexts a finite resource.



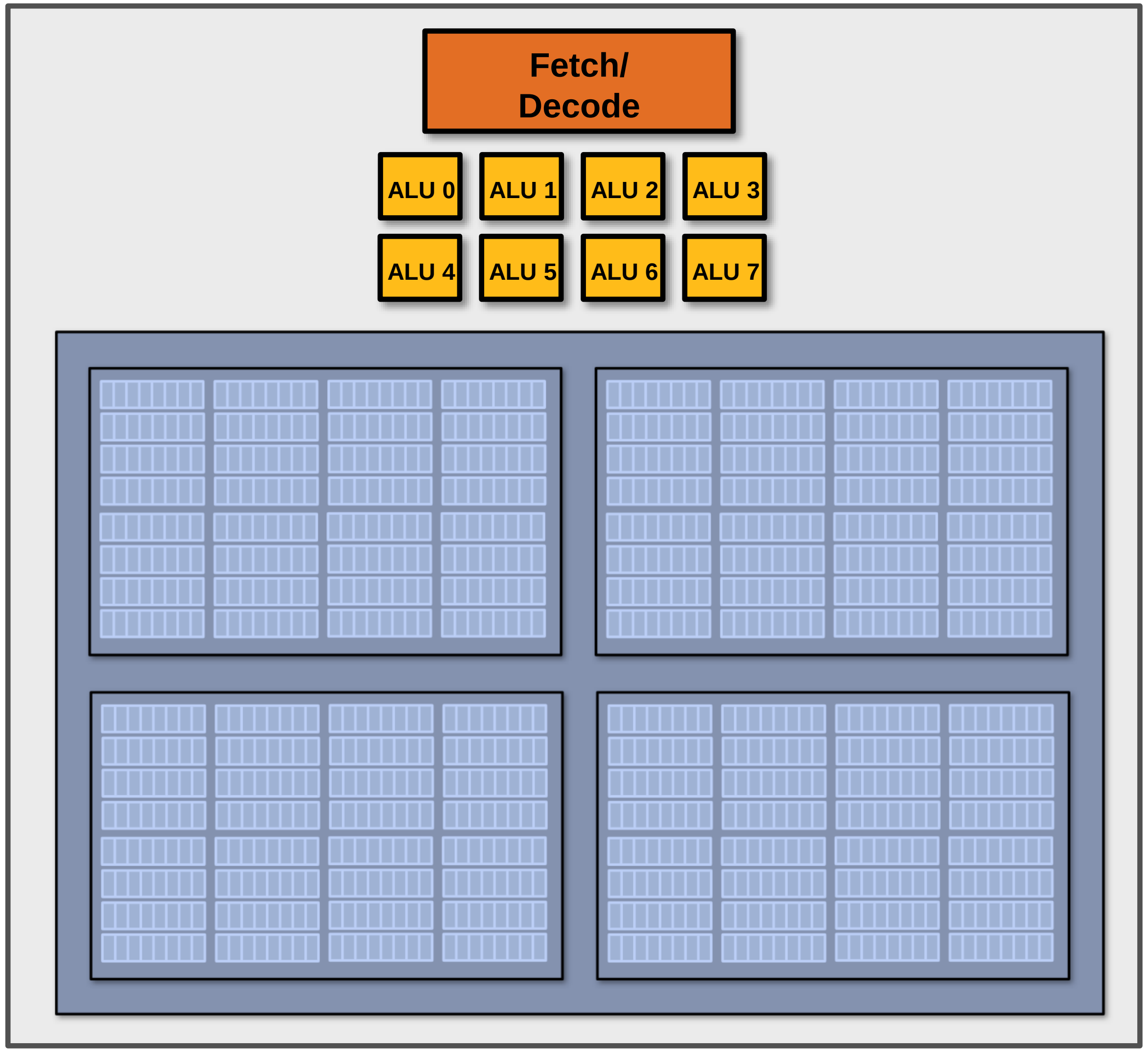
Many small contexts (high latency hiding ability)

1 core (16 hardware threads, storage for small working set per thread)



Four large contexts (low latency hiding ability)

1 core (4 hardware threads, storage for larger working set per thread)



Hardware-supported multi-threading

- **Core manages execution contexts for multiple threads**
  - Runs instructions from runnable threads (processor makes decision about which thread to run each clock, not the operating system)
  - Core still has the same number of ALU resources: multi-threading only helps use them more efficiently in the face of high-latency operations like memory access
- **Interleaved multi-threading (a.k.a. temporal multi-threading)**
  - What I described on the previous slides: each clock, the core chooses a thread, and runs an instruction from the thread on the ALUs
- **Simultaneous multi-threading (SMT)**
  - Each clock, core chooses instructions from multiple threads to run on ALUs
  - Extension of superscalar CPU design
  - Example: Intel Hyper-threading (2 threads per core)

Multi-threading summary

- **Benefit: use a core's execution resources (ALUs) more efficiently**
  - Hide memory latency
  - Fill multiple functional units of superscalar architecture
  - (when one thread has insufficient ILP)

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

A fictitious multi-core chip

16 cores

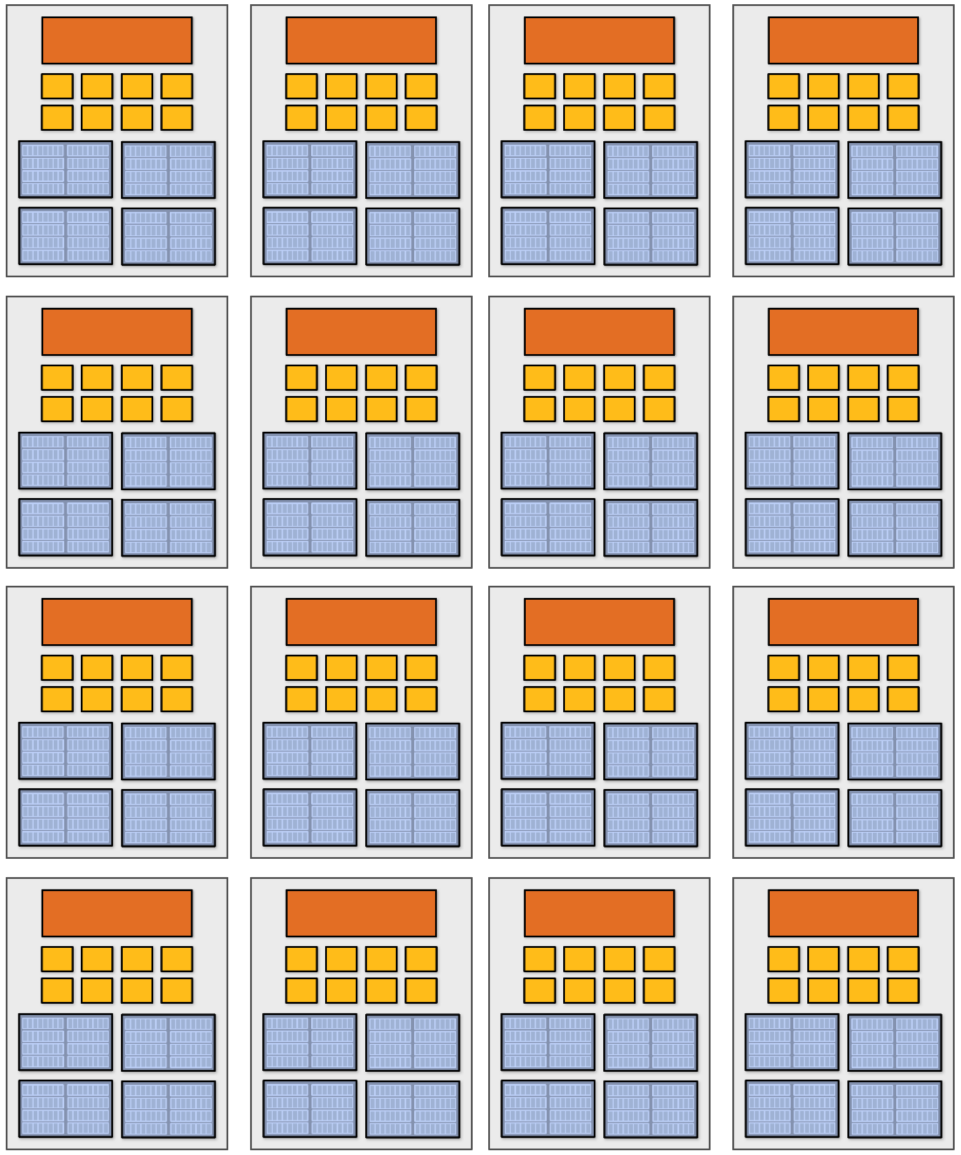
8 SIMD ALUs per core (128 total)

4 threads per core

16 simultaneous instruction streams

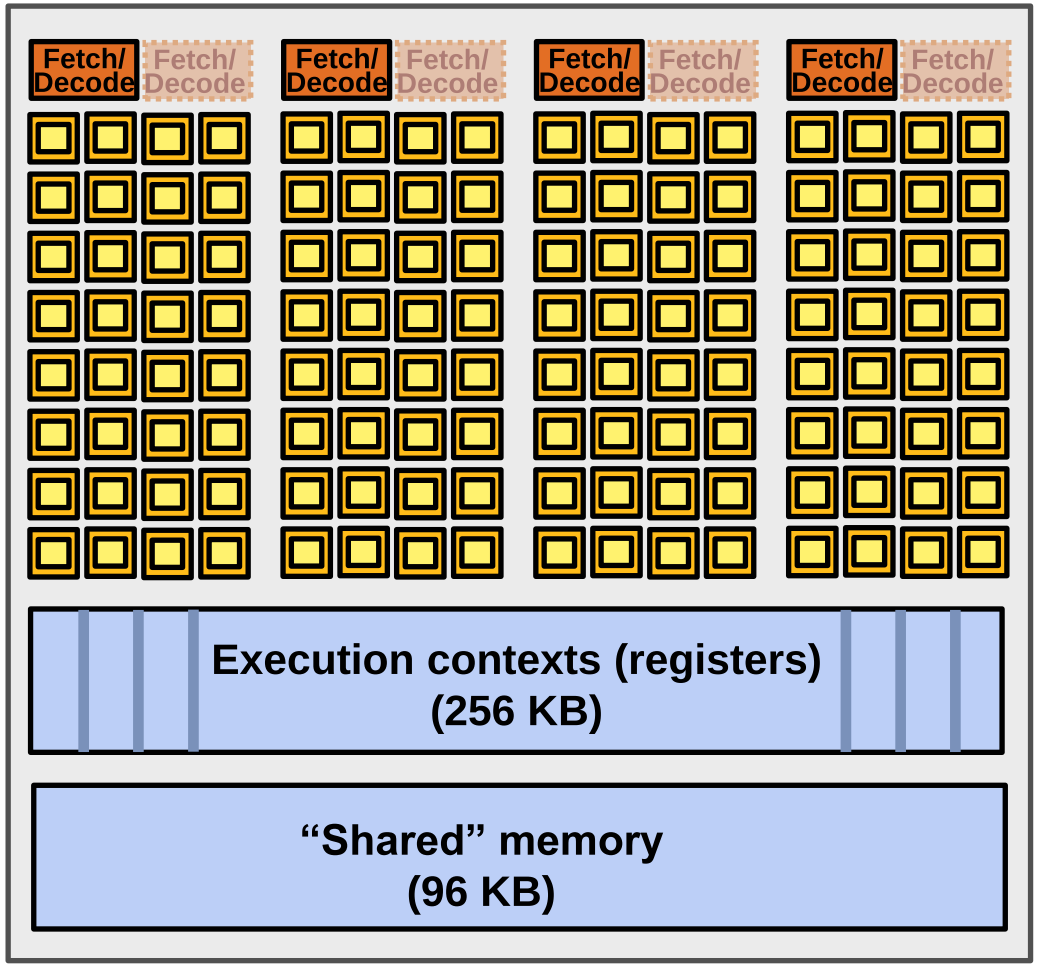
64 total concurrent instruction streams

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



GPUs: extreme throughput-oriented processors

NVIDIA GTX 1080 core ("SM")

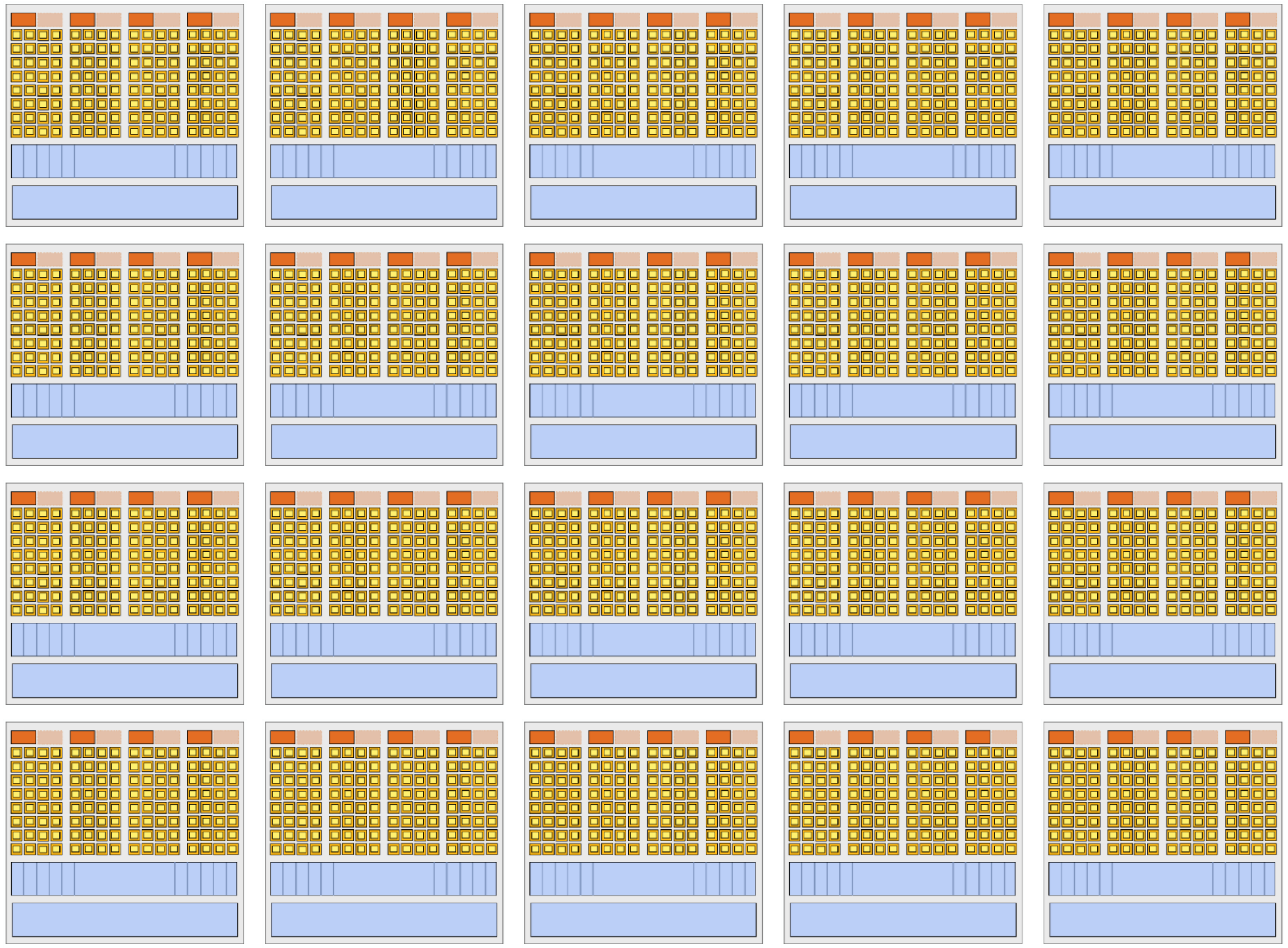


Source: NVIDIA Pascal Tuning Guide

= SIMD function unit, control shared across 32 units (1 MUL-ADD per clock)

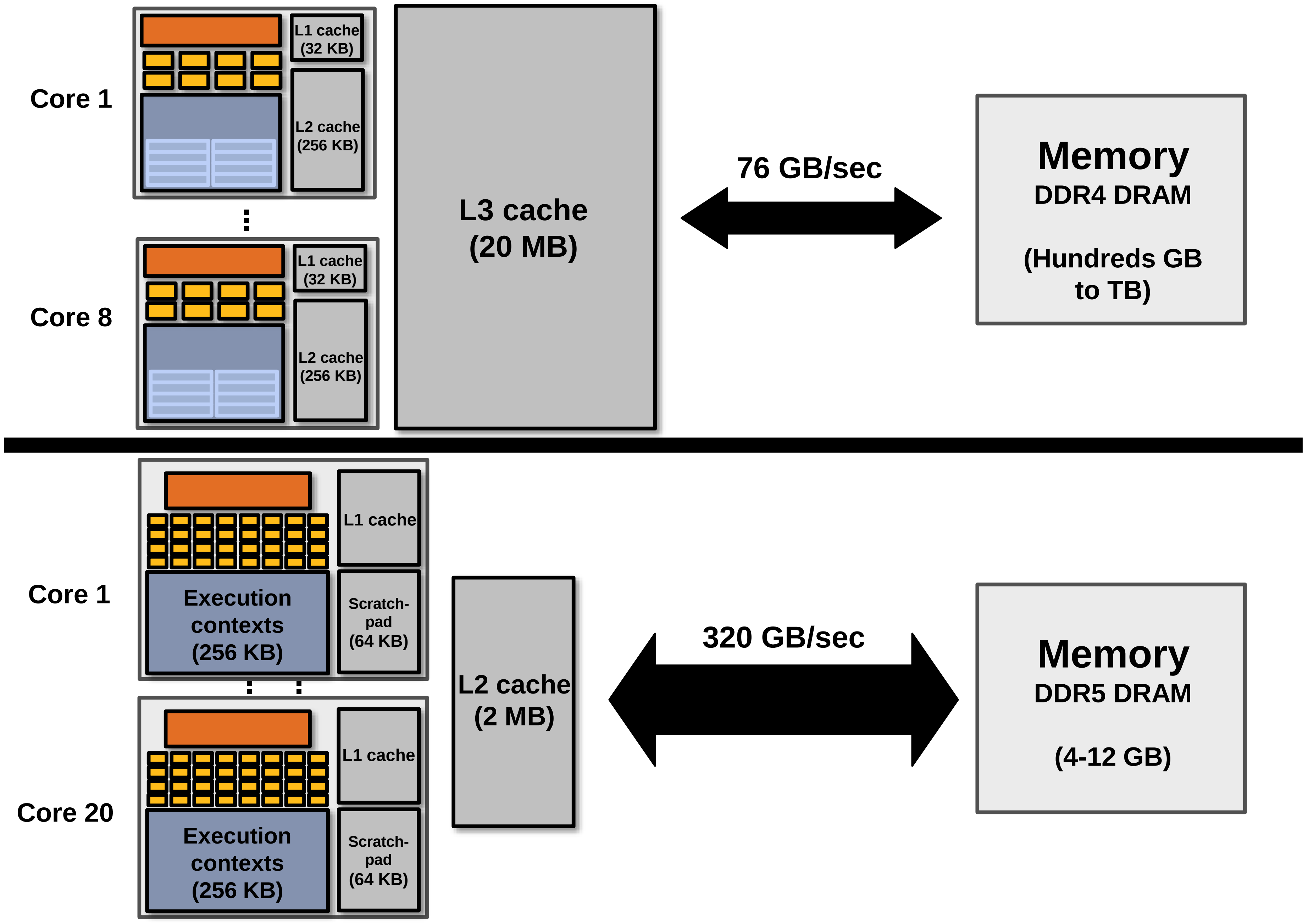
- **Instructions operate on 32 pieces of data at a time (instruction streams called "warps").**
- **Think: warp = thread issuing 32-wide vector instructions**
- **Different instructions from up to four warps can be executed simultaneously (simultaneous multi-threading)**
- **Up to 64 warps are interleaved on the SM (interleaved multi-threading)**
- **Over 2,048 elements can be processed concurrently by a core**

NVIDIA GTX 1080



There are 20 SM cores on the GTX 1080:
That’s 40,960 pieces of data being processed concurrently to get maximal latency hiding!

CPU vs. GPU memory hierarchies



CPU:

Big caches, few threads per core, modest memory BW
Rely mainly on caches and prefetching (automatic)

GPU:

Small caches, many threads, huge memory BW
Rely heavily on multi-threading for performance (manual)

Bandwidth limited!

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

No amount of latency hiding helps this.

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

Bandwidth is a critical resource

Performant parallel programs will:

- **Organize computation to fetch data from memory less often**
  - Reuse data previously loaded by the same thread (traditional intra-thread temporal locality optimizations)
  - Share data across threads (inter-thread cooperation)

- **Request data less often (instead, do more arithmetic: it's "free")**
  - Useful term: "arithmetic intensity" — ratio of math operations to data access operations in an instruction stream
  - Main point: programs must have high arithmetic intensity to utilize modern processors efficiently

Summary

- Three major ideas that all modern processors employ to varying degrees
  - Provide multiple processing cores
    - Simpler cores (embrace thread-level parallelism over instruction-level parallelism)
  - Amortize instruction stream processing over many ALUs (SIMD)
    - Increase compute capability with little extra cost
  - Use multi-threading to make more efficient use of processing resources (hide latencies, fill all available resources)

- Due to high arithmetic capability on modern chips, many parallel applications (on both CPUs and GPUs) are bandwidth bound

- GPU architectures use the same throughput computing ideas as CPUs: but GPUs push these concepts to extreme scales

Review slides

(additional examples for review and to check our understanding)

Putting together the concepts from this lecture:

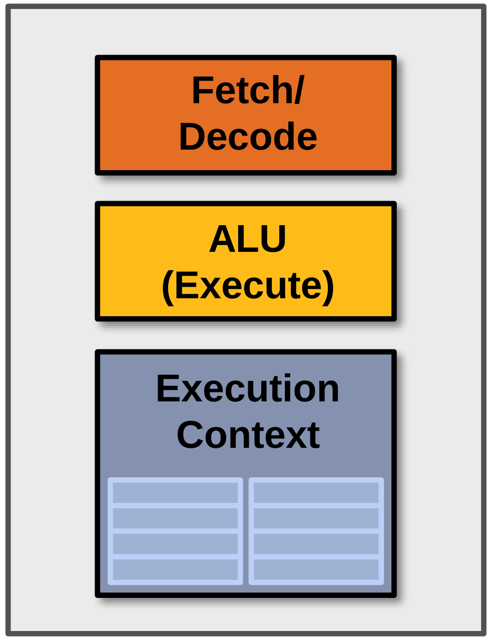
(if you understand the following sequence you understand this lecture)

Running code on a simple processor

My very simple program:
compute $\sin(x)$ using Taylor expansion

```
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;
   }
}
```

**My very simple processor:
completes one instruction per clock**



Review: superscalar execution

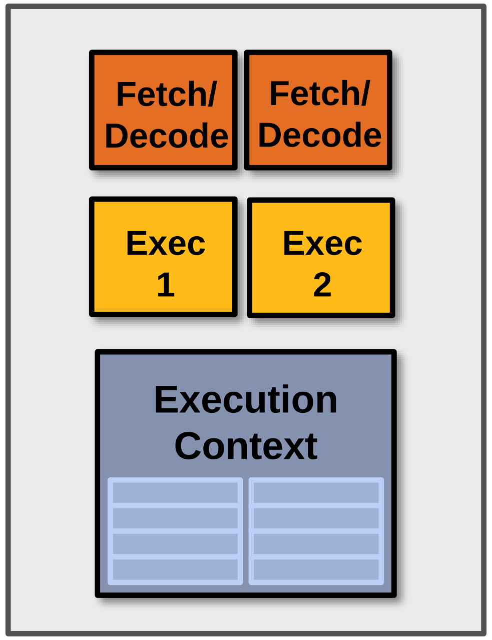
Unmodified program

```
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;
   }
}
```

Independent operations in instruction stream

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

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

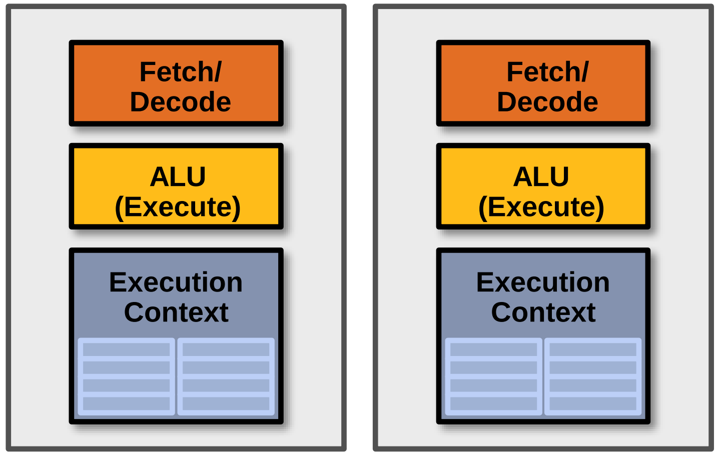


Review: multi-core execution (two cores)

Modify program to create two threads of control (two instruction streams)

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

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

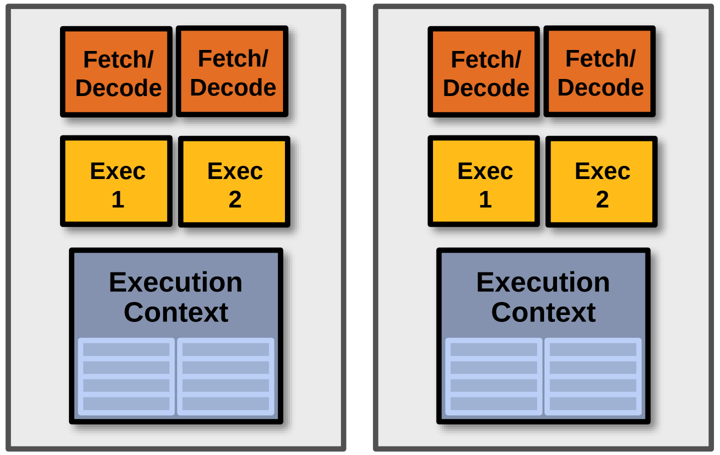


Review: multi-core + superscalar execution

Modify program to create two threads of control (two instruction streams)

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

My superscalar dual-core processor: executes up to two instructions per clock from an instruction stream on each core.

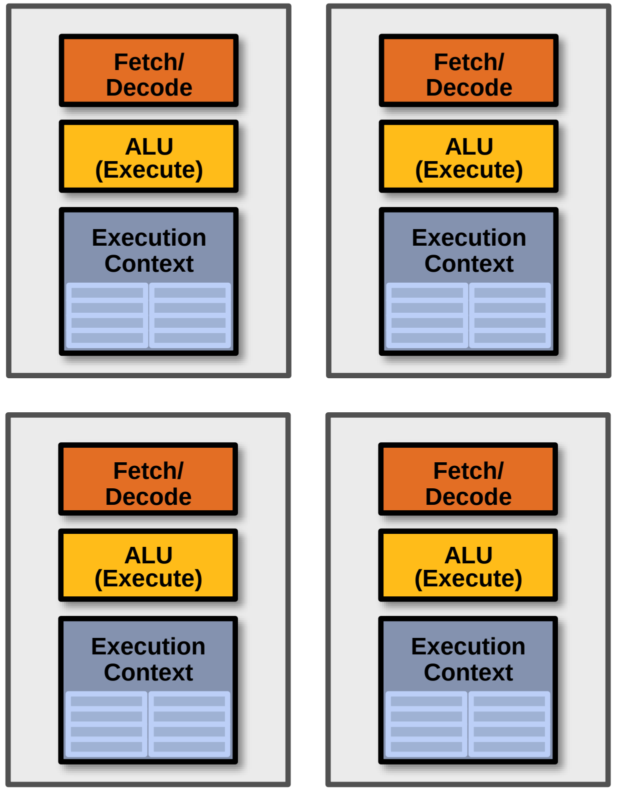


Review: multi-core (four cores)

Modify program to create many threads of control

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

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



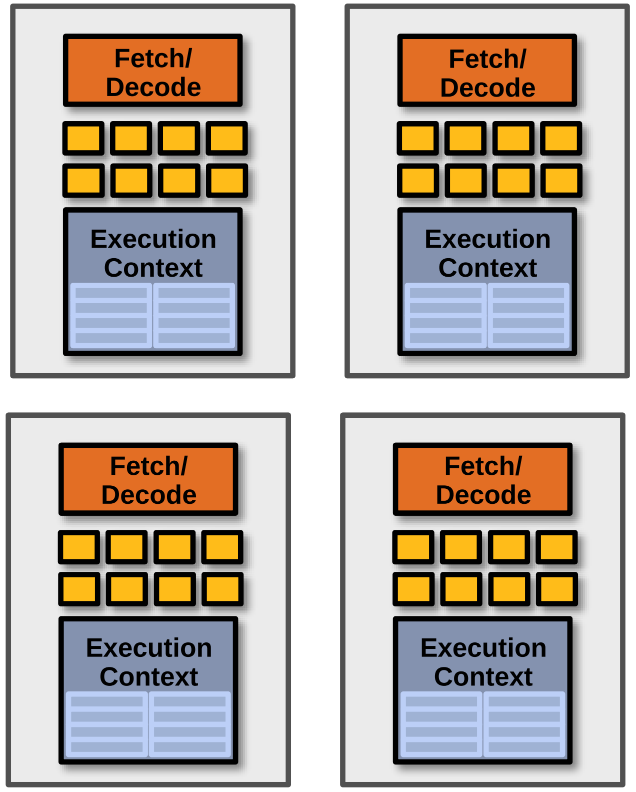
Review: four, 8-wide SIMD cores

Observation: program must execute many iterations of the same loop body.

Optimization: share instruction stream across execution of multiple iterations (single instruction multiple data = SIMD)

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

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



Review: four SIMD, multi-threaded cores

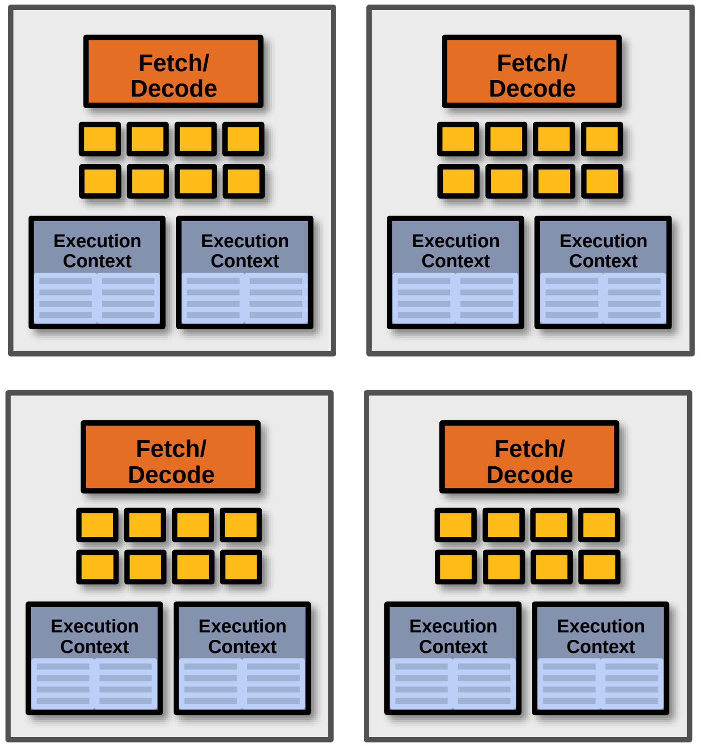
Observation: memory operations have very long latency
Solution: hide latency of loading data for one iteration by executing arithmetic instructions from other iterations

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

Memory load

Memory store

My multi-threaded, SIMD quad-core processor: executes one SIMD instruction per clock from one instruction stream on each core. But can switch to processing the other instruction stream when faced with a stall.

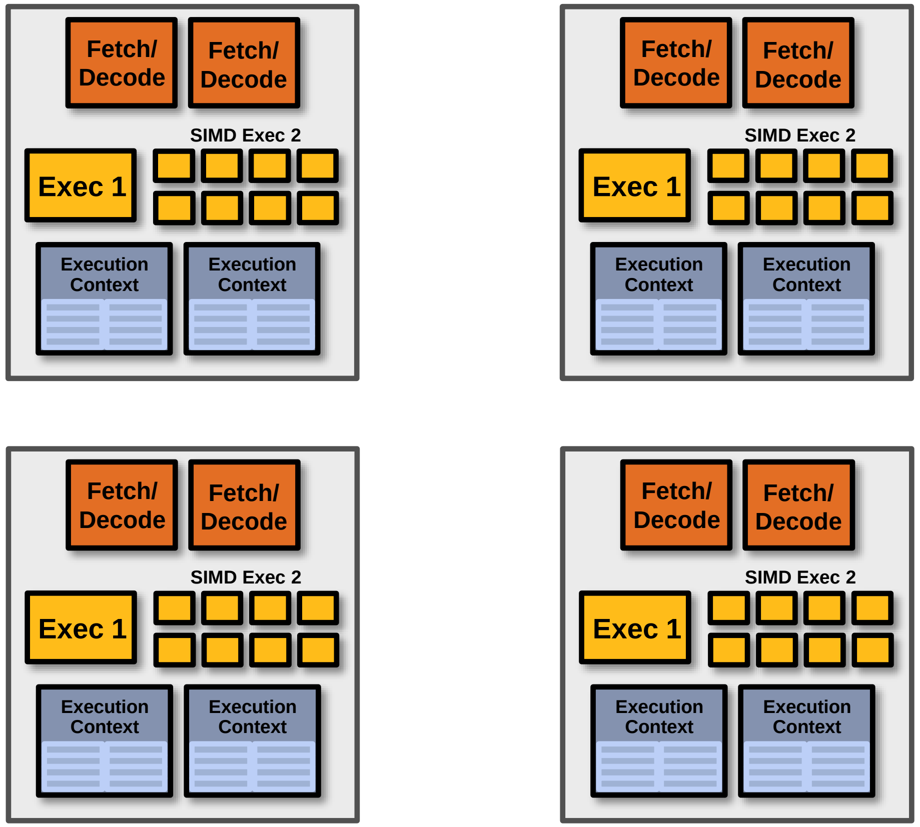


Summary: four superscalar, SIMD, multi-threaded cores

My multi-threaded, superscalar, SIMD quad-core processor:

executes up to two instructions per clock from one instruction stream on each core (in this example: one SIMD instruction + one scalar instruction).

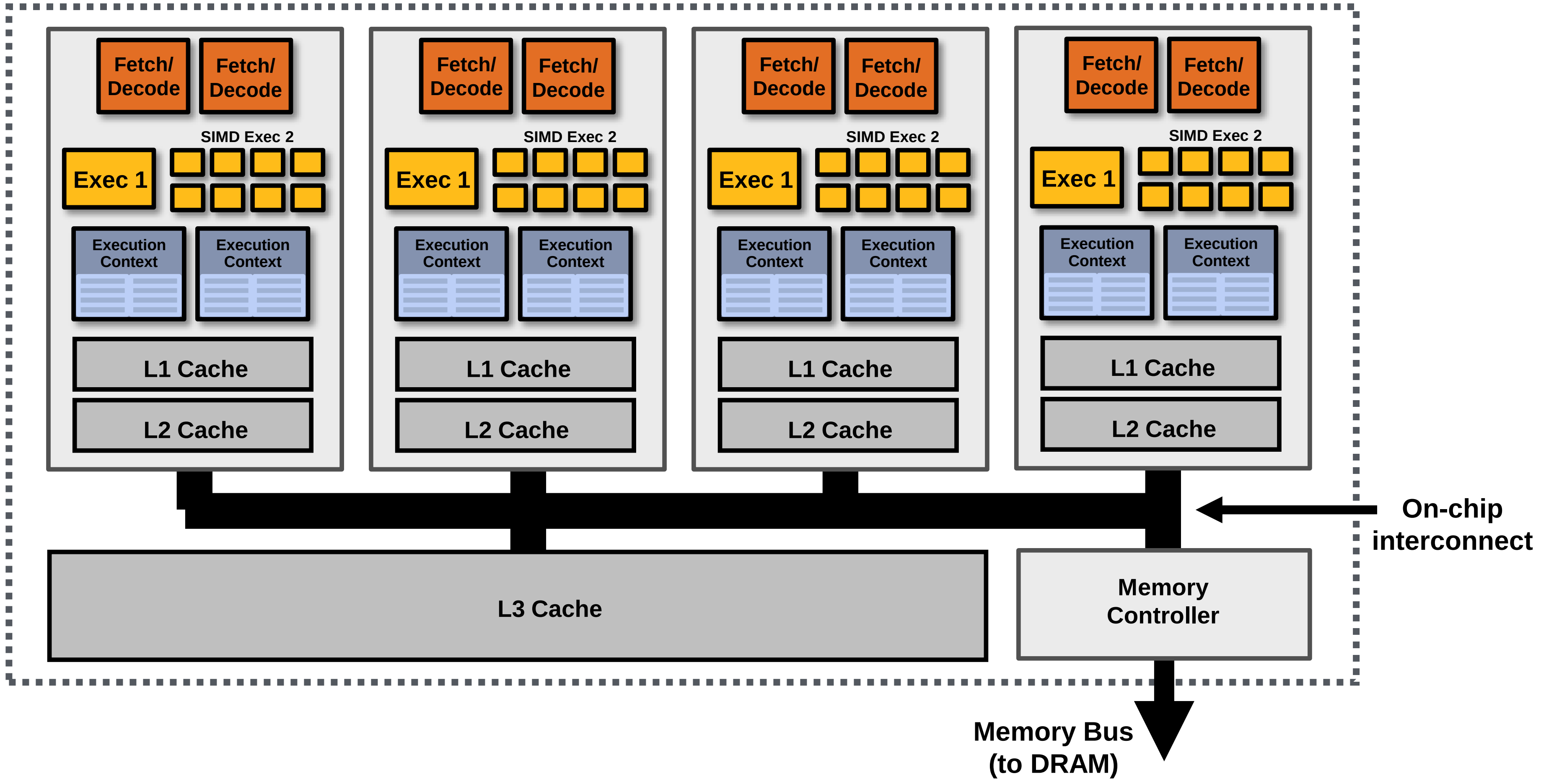
Processor can switch to execute the other instruction stream when faced with stall.



Connecting it all together

A simple quad-core processor:

Four cores, two-way multi-threading per core (max eight threads active on chip at once), up to two instructions per clock per core (one of those instructions is 8-wide SIMD)


Fetch/ Decode
Fetch/ Decode
SIMD Exec 2
Exec 1
Execution Context
Execution Context
L1 Cache
L2 Cache
Fetch/ Decode
Fetch/ Decode
SIMD Exec 2
Exec 1
Execution Context
Execution Context
L1 Cache
L2 Cache
Fetch/ Decode
Fetch/ Decode
SIMD Exec 2
Exec 1
Execution Context
Execution Context
L1 Cache
L2 Cache
Fetch/ Decode
Fetch/ Decode
SIMD Exec 2
Exec 1
Execution Context
Execution Context
L1 Cache
L2 Cache
On-chip interconnect
L3 Cache
Memory Controller
Memory Bus (to DRAM)