058165 - PARALLEL COMPUTING

Fabrizio Ferrandi

a.a. 2022-2023

Acknowledge

Material from:

Parallel Computing lectures from prof. Kayvon and prof. Olukotun Stanford University

What is a computer program?

What is a program? (from a processor's perspective)

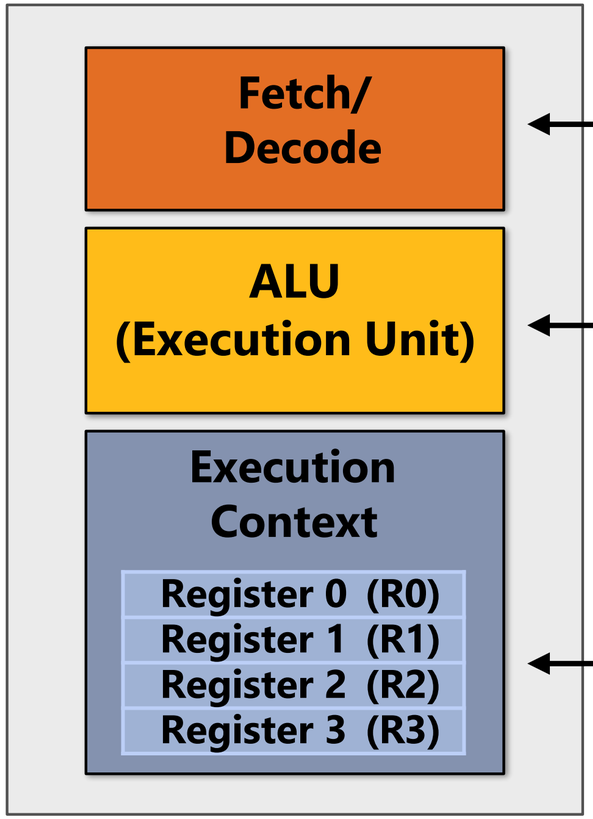
A program is just a list of processor instructions!

```
int main(int argc, char** argv) {

  int x = 1;

  for (int i=0; i<10; i++) {
    x = x + x;
  }

  printf("%d\n", x);

  return 0;
}
```


```
main:
100000f10:      pushq     %rbp
100000f11:      movq %rsp, %rbp
100000f14:      subq $32, %rsp
100000f18:      movl $0, -4(%rbp)
100000f1f:      movl %edi, -8(%rbp)
100000f22:      movq %rsi, -16(%rbp)
100000f26:      movl $1, -20(%rbp)
100000f2d:      movl $0, -24(%rbp)
100000f34:      cmpl $10, -24(%rbp)
100000f38:      jge  23 <_main+0x45>
100000f3e:      movl -20(%rbp), %eax
100000f41:      addl -20(%rbp), %eax
100000f44:      movl %eax, -20(%rbp)
100000f47:      movl -24(%rbp), %eax
100000f4a:      addl $1, %eax
100000f4d:      movl %eax, -24(%rbp)
100000f50:      jmp  -33 <_main+0x24>
100000f55:      leaq 58(%rip), %rdi
100000f5c:      movl -20(%rbp), %esi
100000f5f:      movb $0, %al
100000f61:      callq     14
100000f66:      xorl %esi, %esi
100000f68:      movl %eax, -28(%rbp)
100000f6b:      movl %esi, %eax
100000f6d:      addq $32, %rsp
100000f71:      popq %rbp
100000f72:      rets
```

What does a processor do?

A processor executes instructions



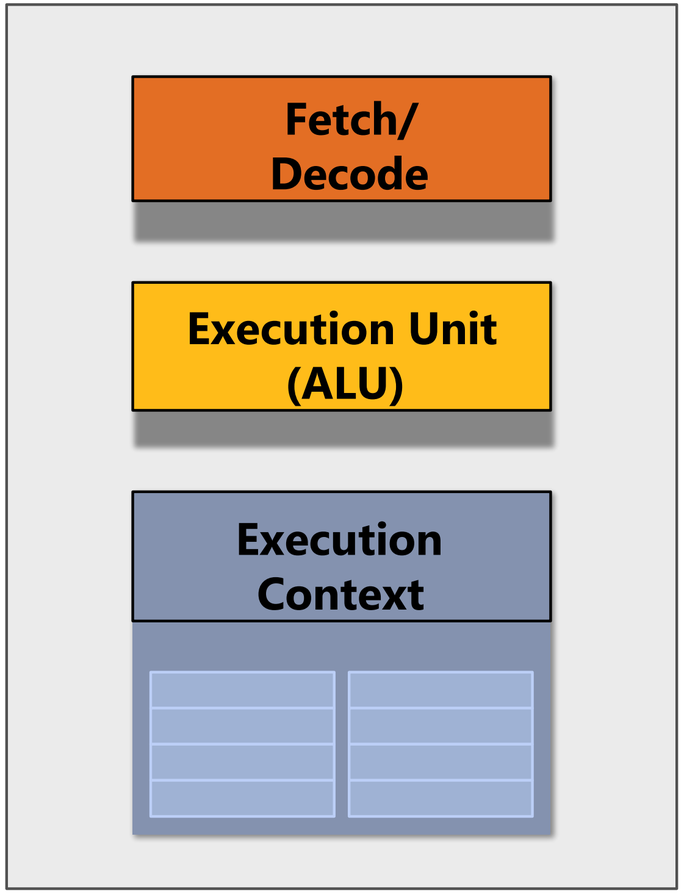
Determine what instruction to run next

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

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

Execute program

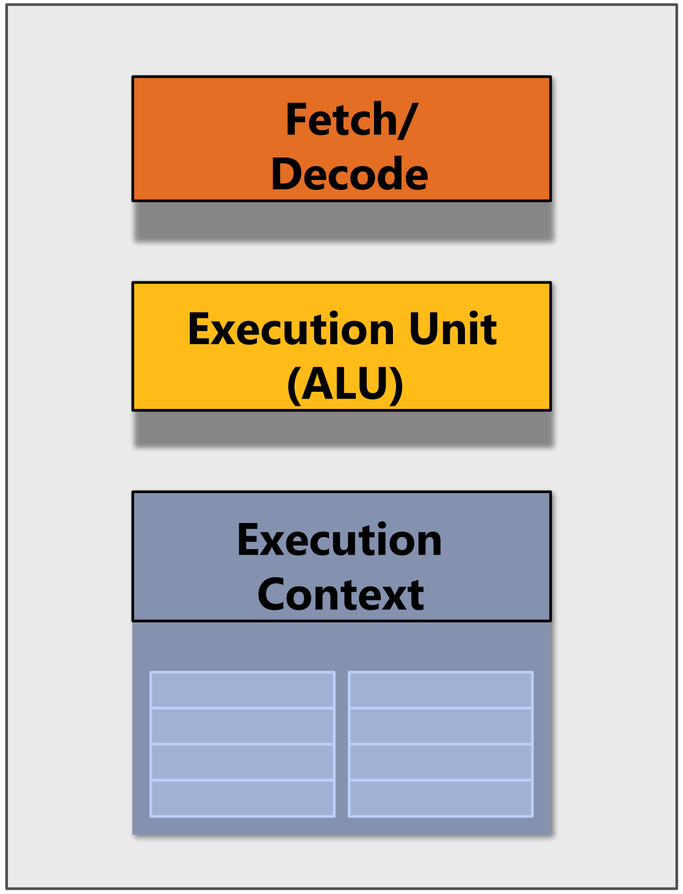
My very simple processor: executes one instruction per clock




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

Execute program

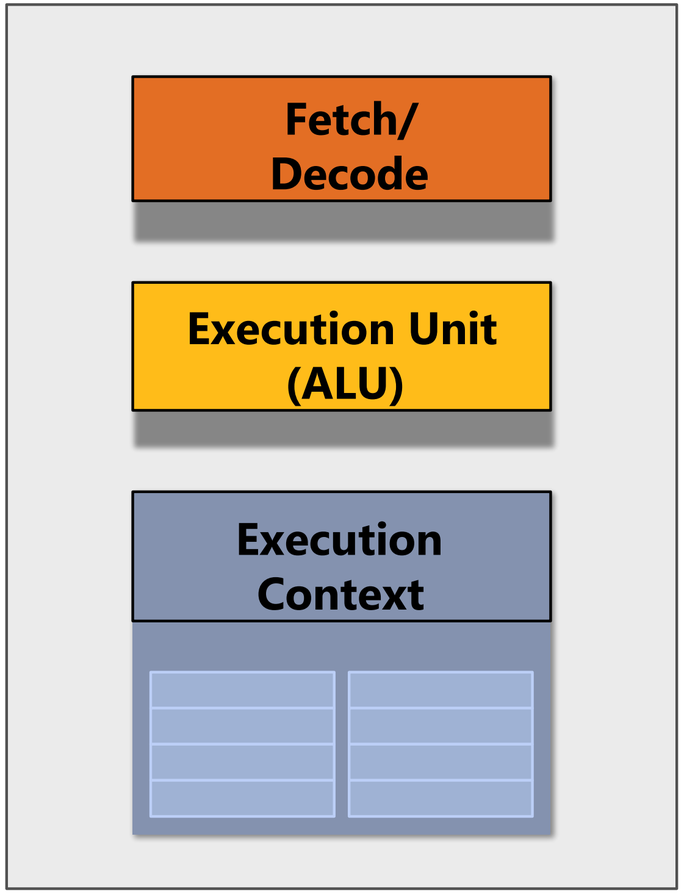
My very simple processor: executes one instruction per clock




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

Execute program

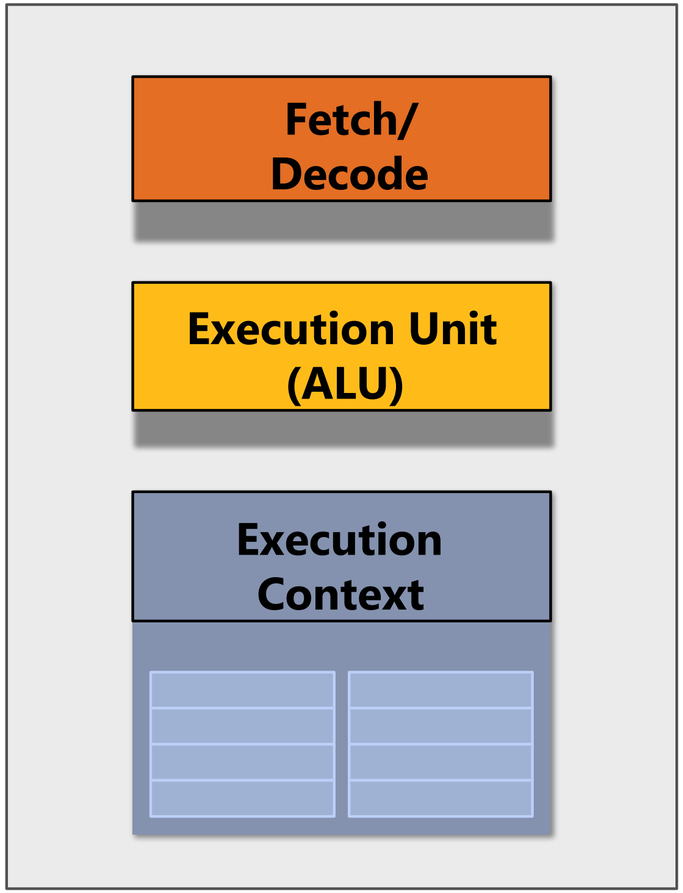
simple processor: executes one instruction per clock




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

Execute program

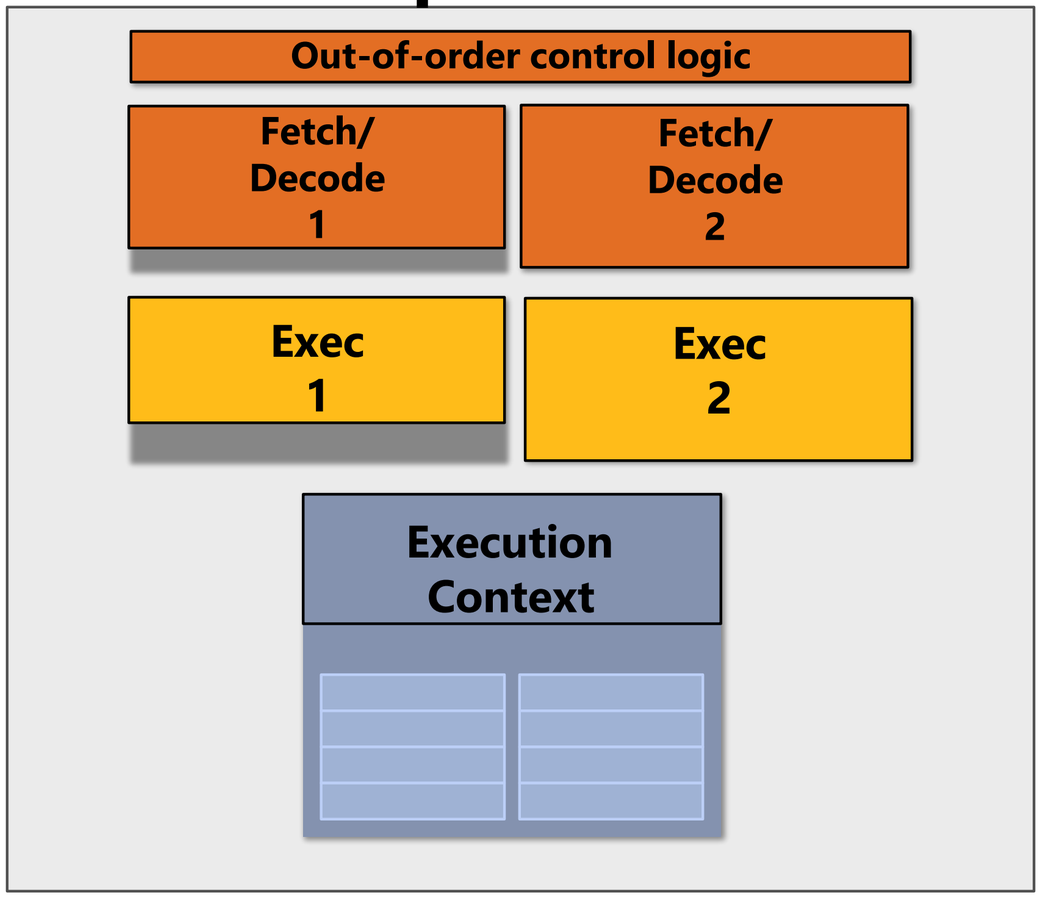
My very simple processor: executes one instruction per clock




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

Superscalar processor

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



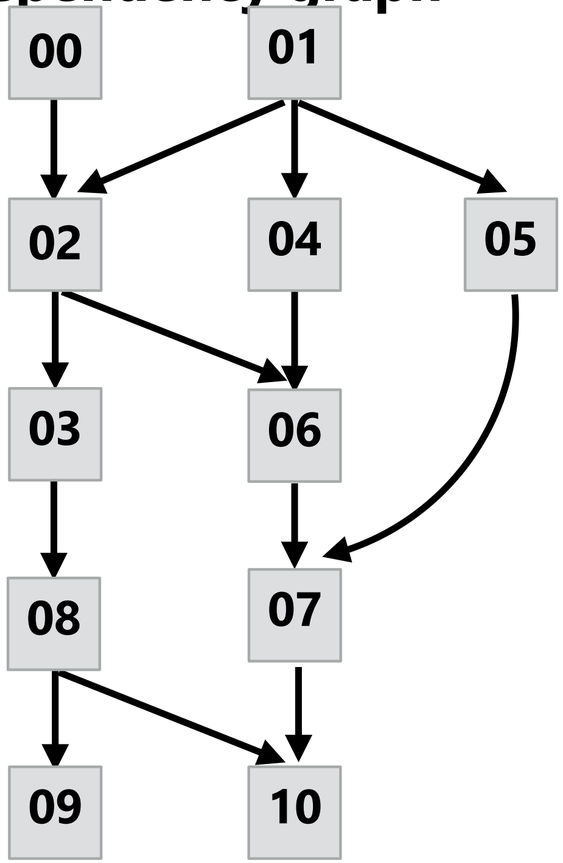
Superscalar execution: processor automatically finds independent instructions in an instruction sequence and can execute them in parallel on multiple execution units.

A more complex example

Program (sequence of instructions)

PC	Instruction	*value during execution*
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`	

Instruction dependency graph



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

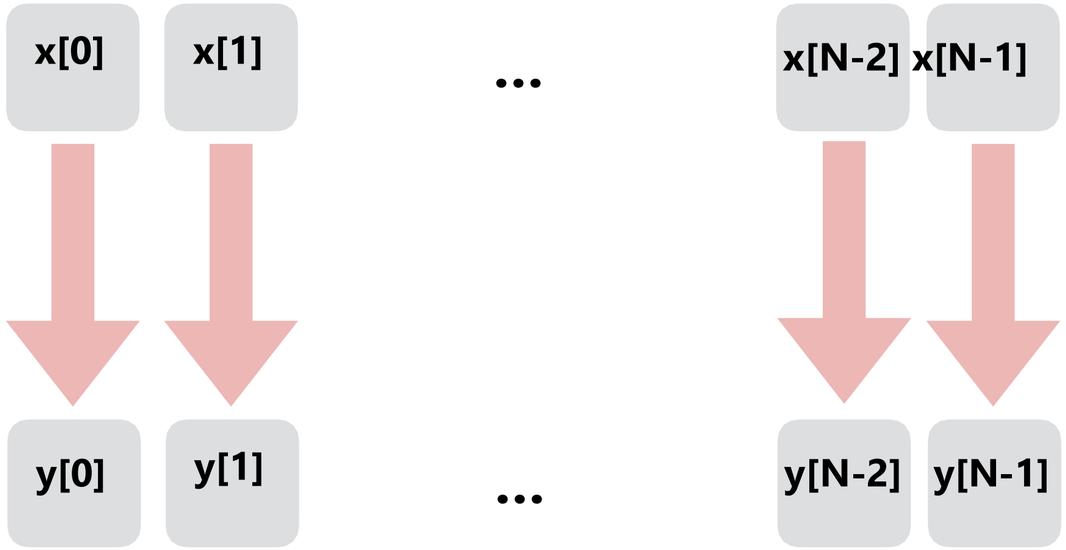
Today's example program

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

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

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

Compute $\sin(x)$ using Taylor expansion:

$$\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \ldots$$

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

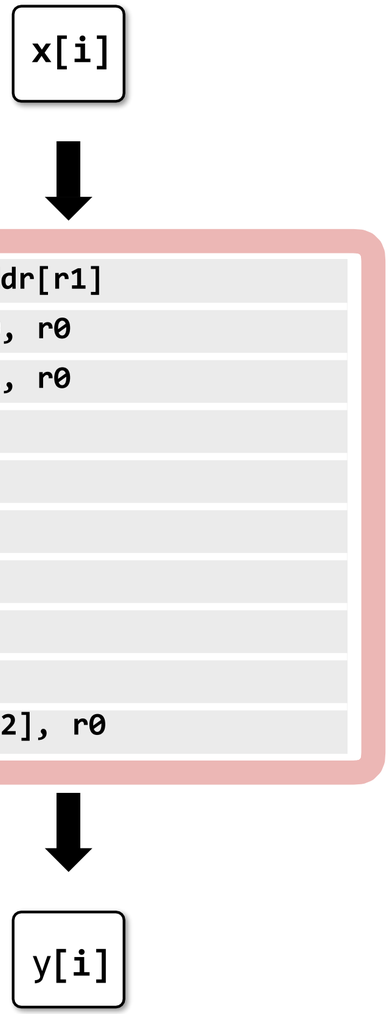


Compile program

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

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

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

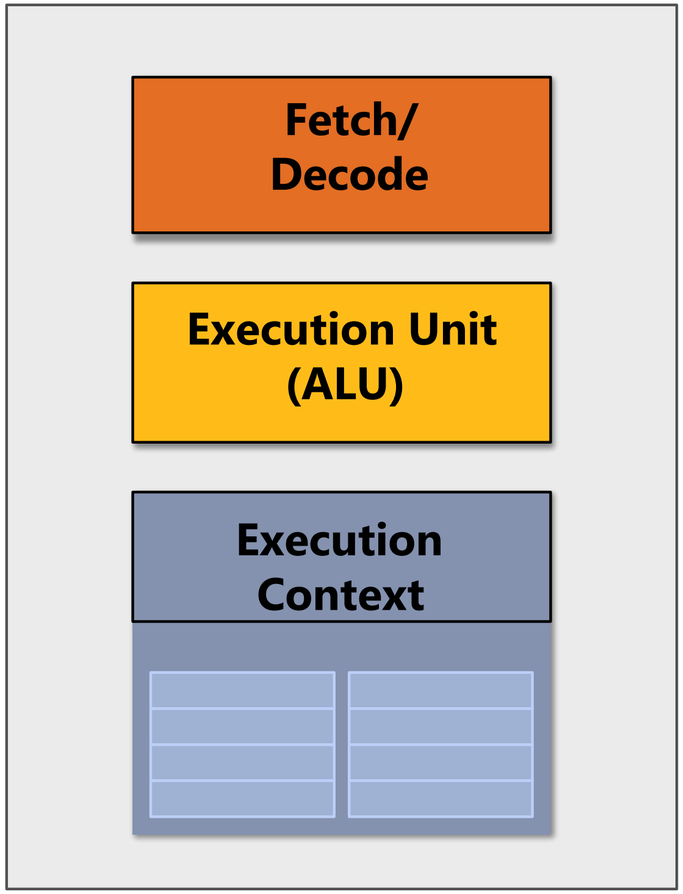
compiler

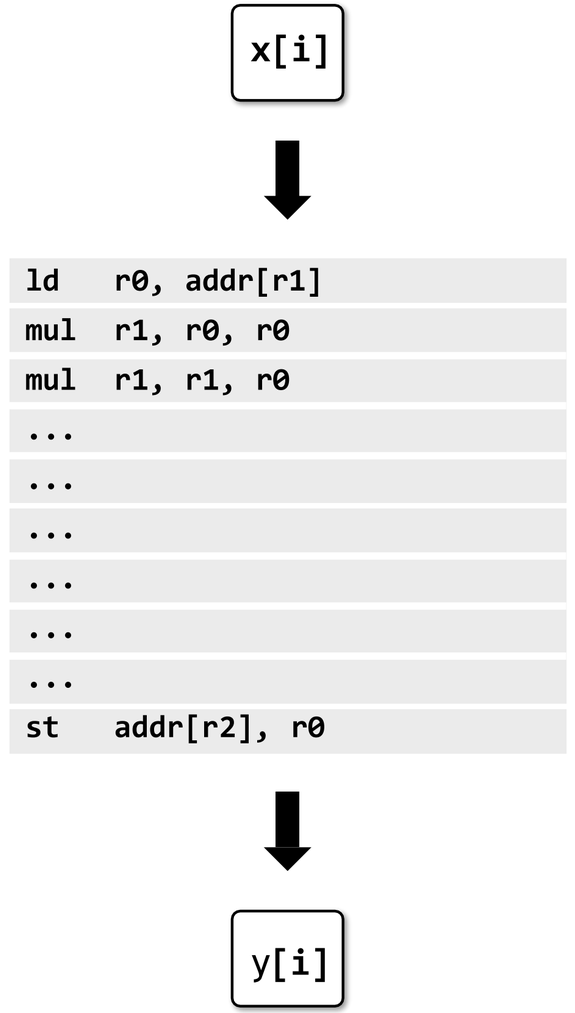
Compiled instruction stream
(scalar instructions)



Execute program

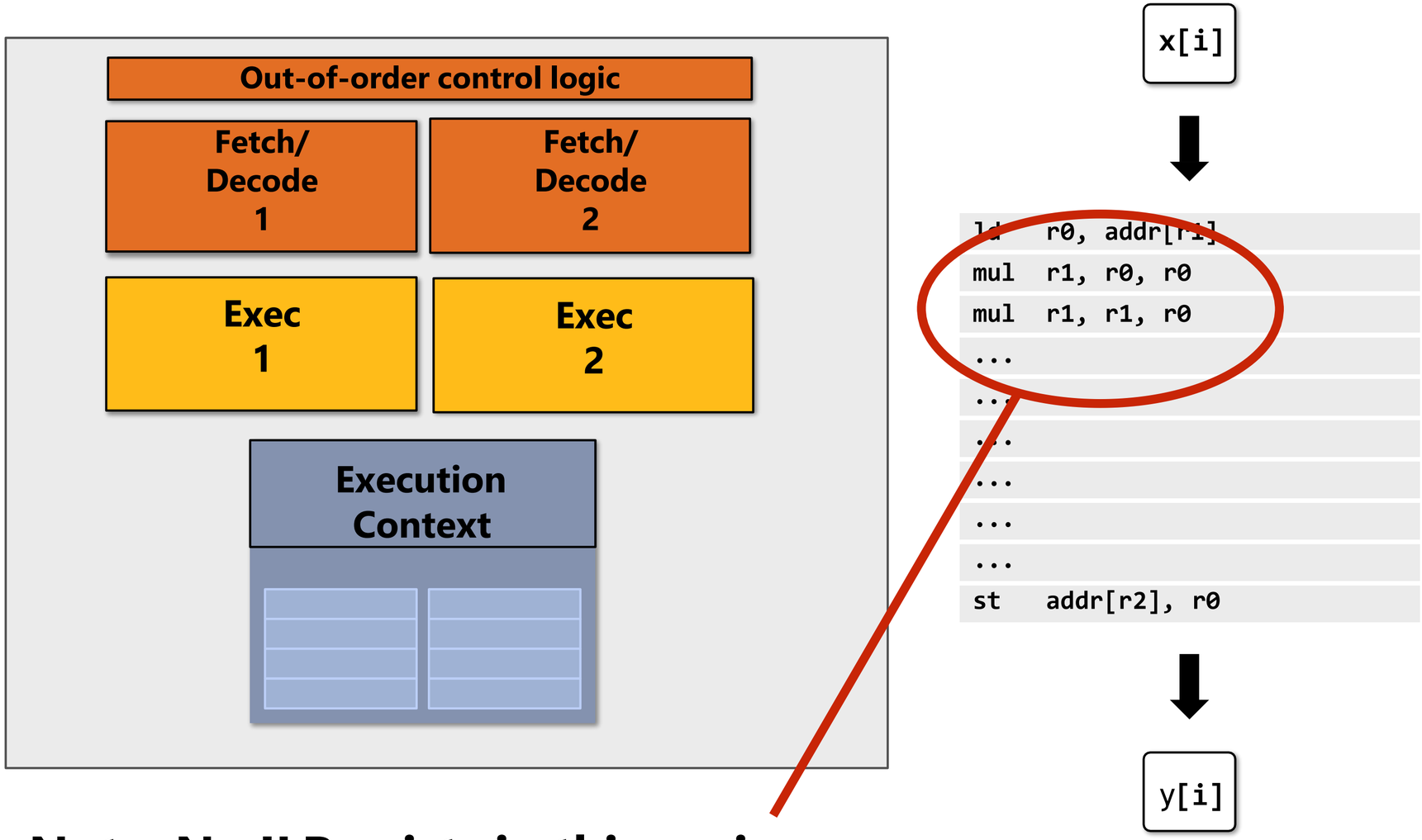
Simple processor: executes one instruction per clock





Superscalar processor

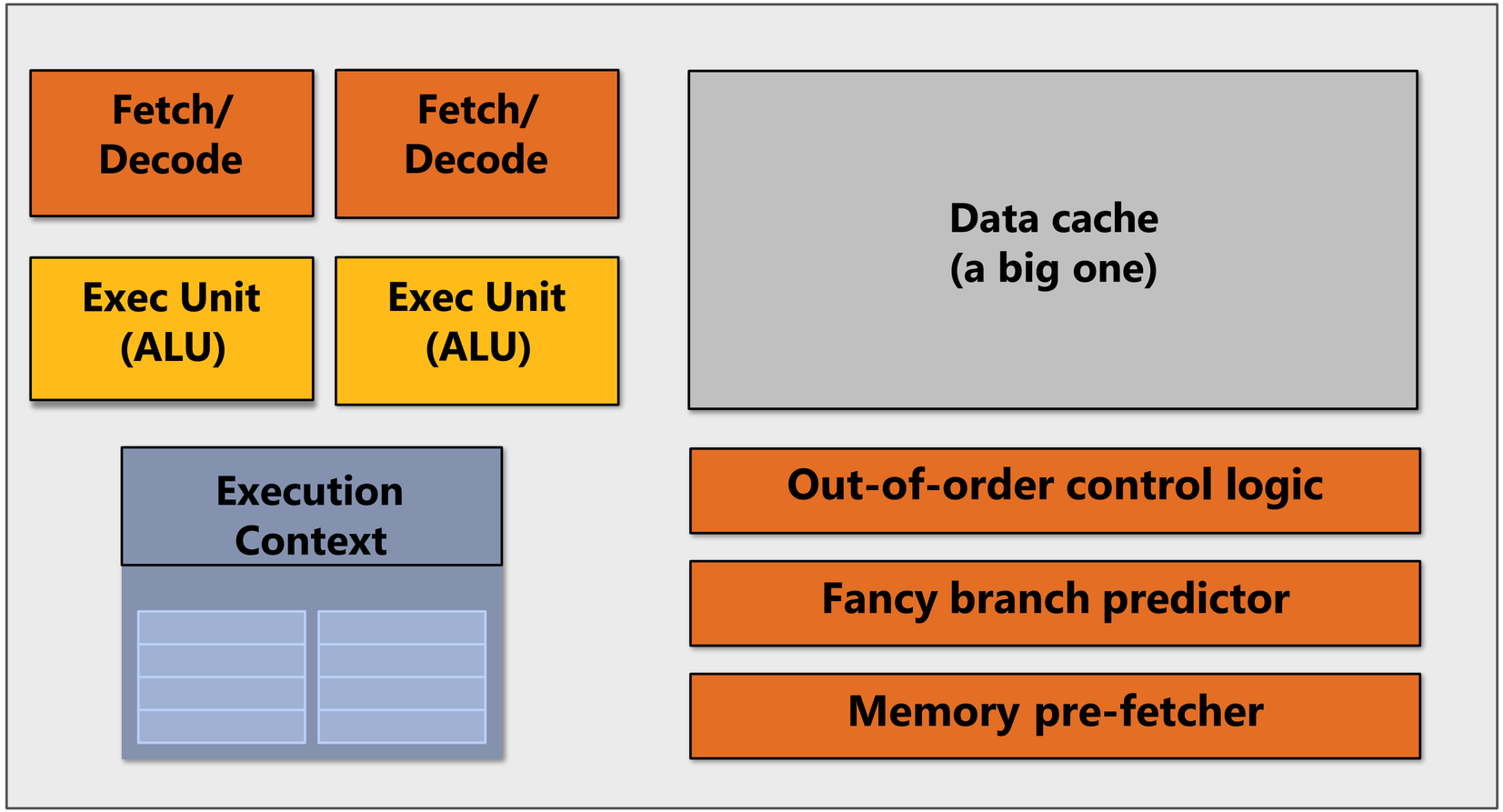
The processor shown here can decode and execute two instructions per clock (if independent instructions exist in an instruction stream)



Note: No ILP exists in this region of the program

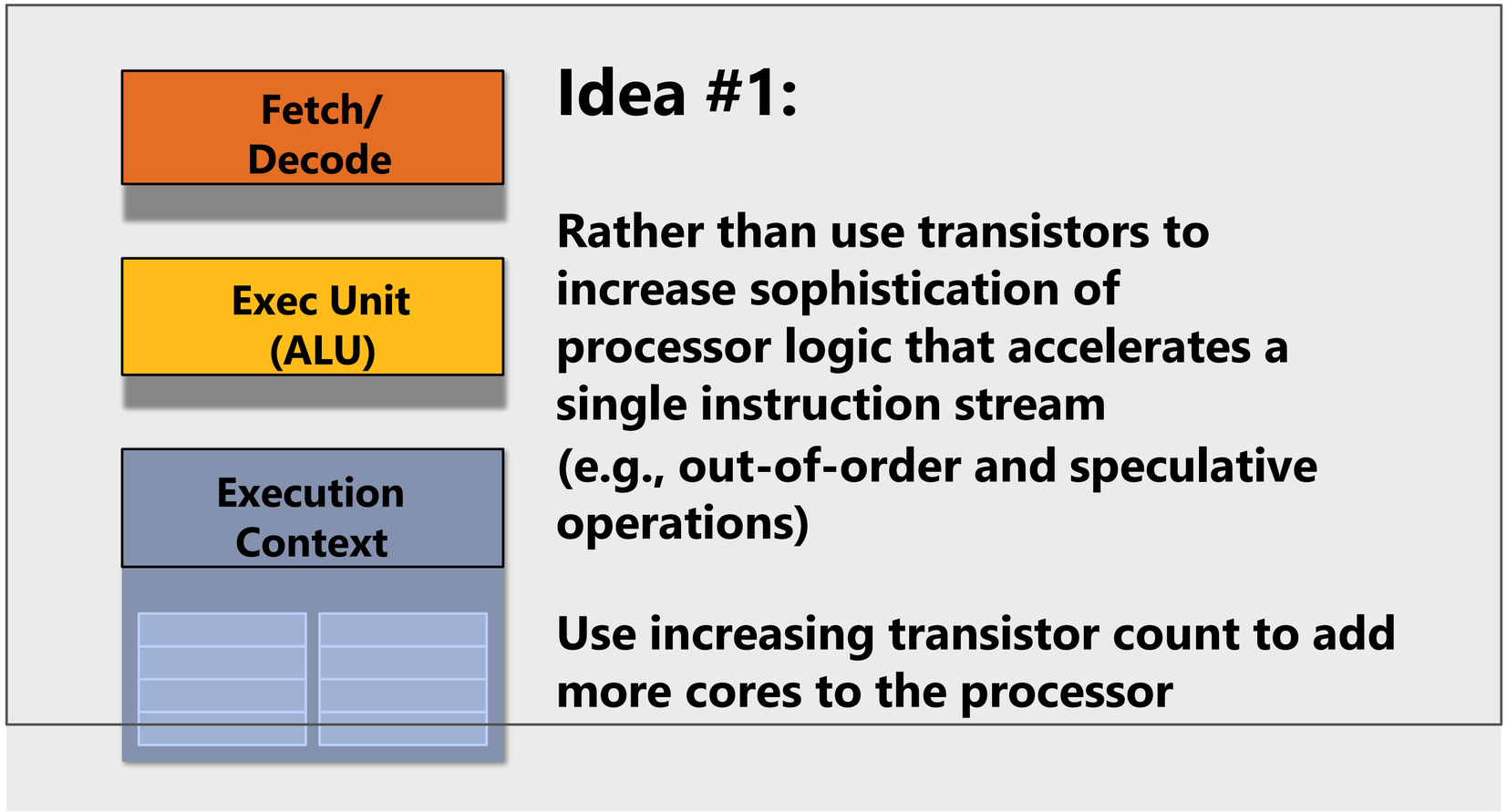
Pre multi-core era processor

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



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

Multi-core era processor

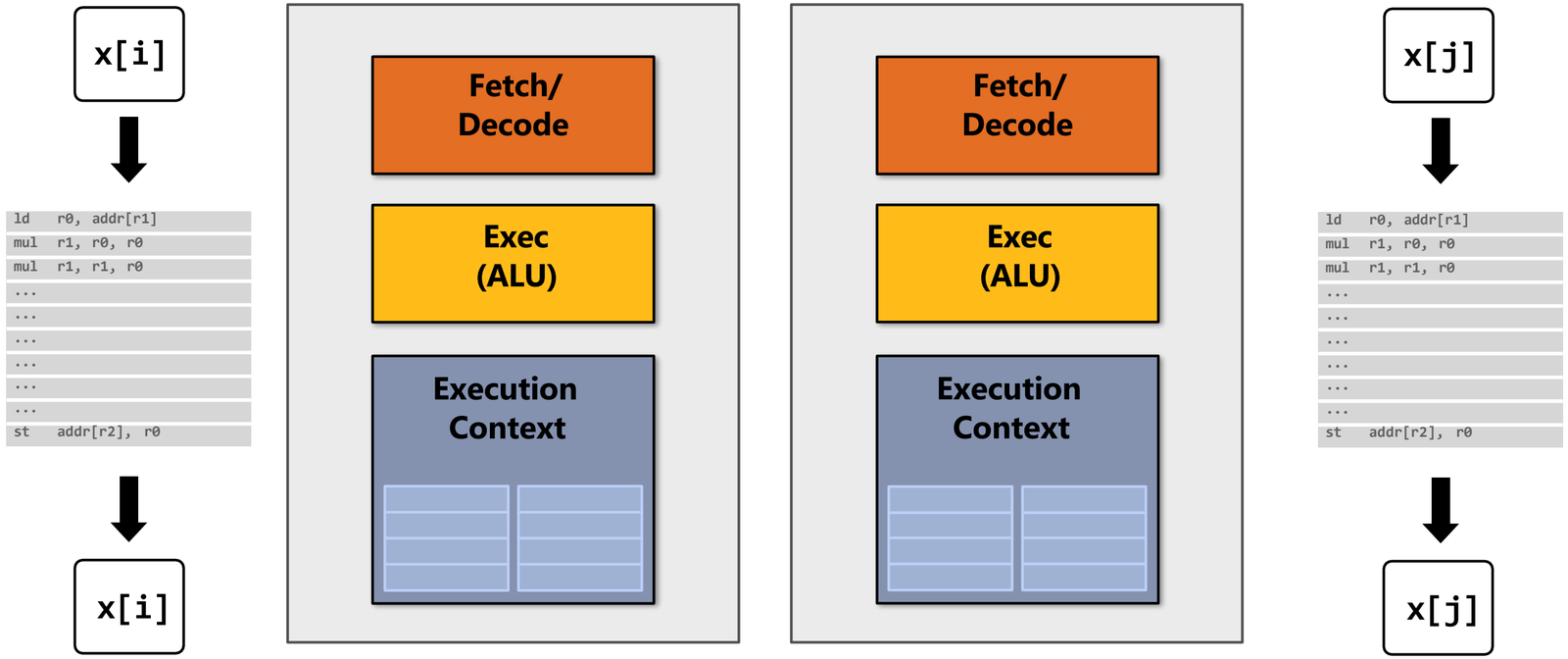


Idea #1:

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

Use increasing transistor count to add more cores to the processor

Two cores: compute two elements in parallel



Simpler cores: each core may be slower at running a single instruction stream than our original "fancy" core (e.g., 25% slower)

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

But our program expresses no parallelism

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

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

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

This C program will compile to an instruction stream that runs as one thread on one processor core.

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

Example: expressing parallelism using C++ threads

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

void my_thread_func(my_args* args)
{
   sinx(args->N, args->terms, args->x, args->y); // do work
}
```

```
void parallel_sinx(int N, int terms, float* x, float* y)
{
   std::thread my_thread;
   my_args args;

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

   my_thread = std::thread(my_thread_func, &args); // launch thread
   sinx(N - args.N, terms, x + args.N, y + args.N); // do work on main thread
   my_thread.join();  // wait for thread to complete
}
```

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

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

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

Data-parallel expression

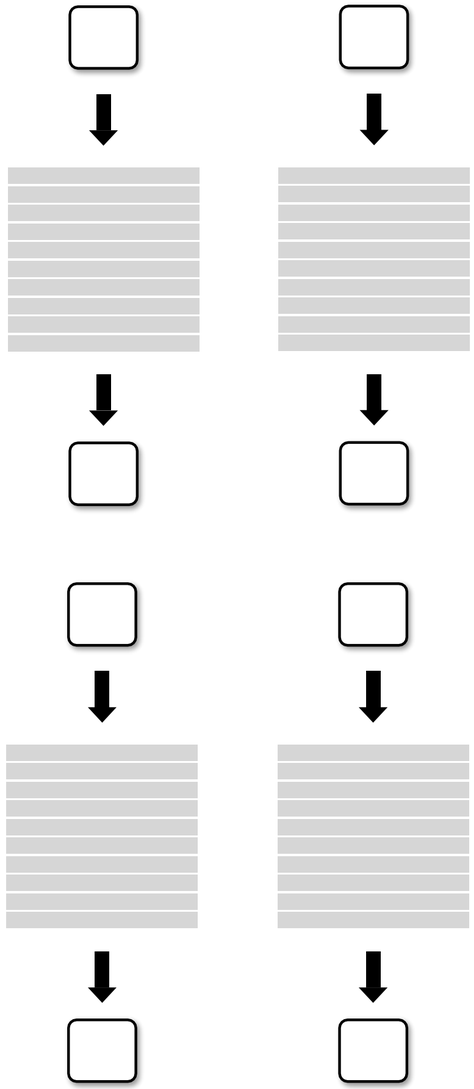
(in Kayvon's fictitious data-parallel language)

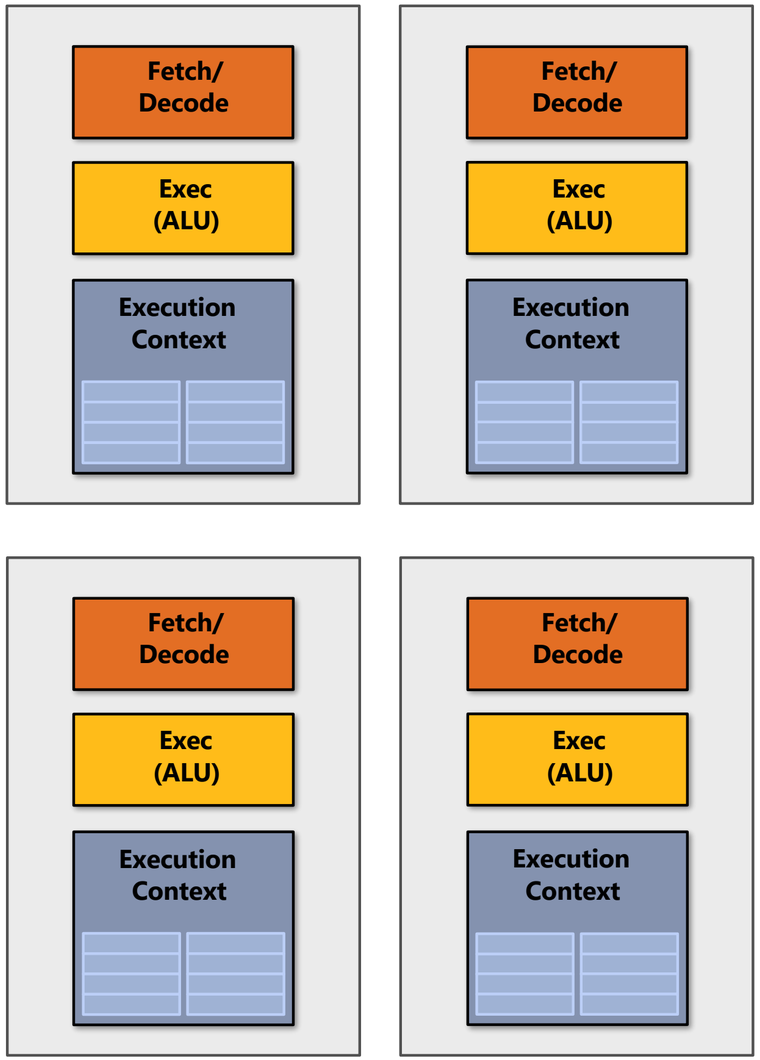
```
void sinx(int N, int terms, float* x, float* y)
{
    // declares that loop iterations are independent
    forall (int i from 0 to N)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6;  // 3!
        int sign = -1;

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

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

In this code, loop iterations are declared by the programmer to be independent (see the 'forall')

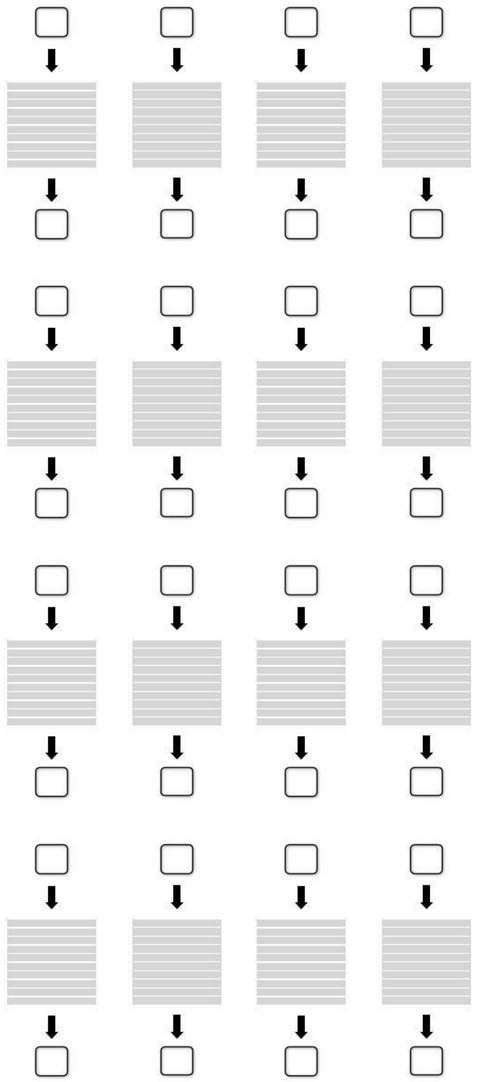
With this information, you could imagine how a compiler might automatically generate parallel threaded code for you.

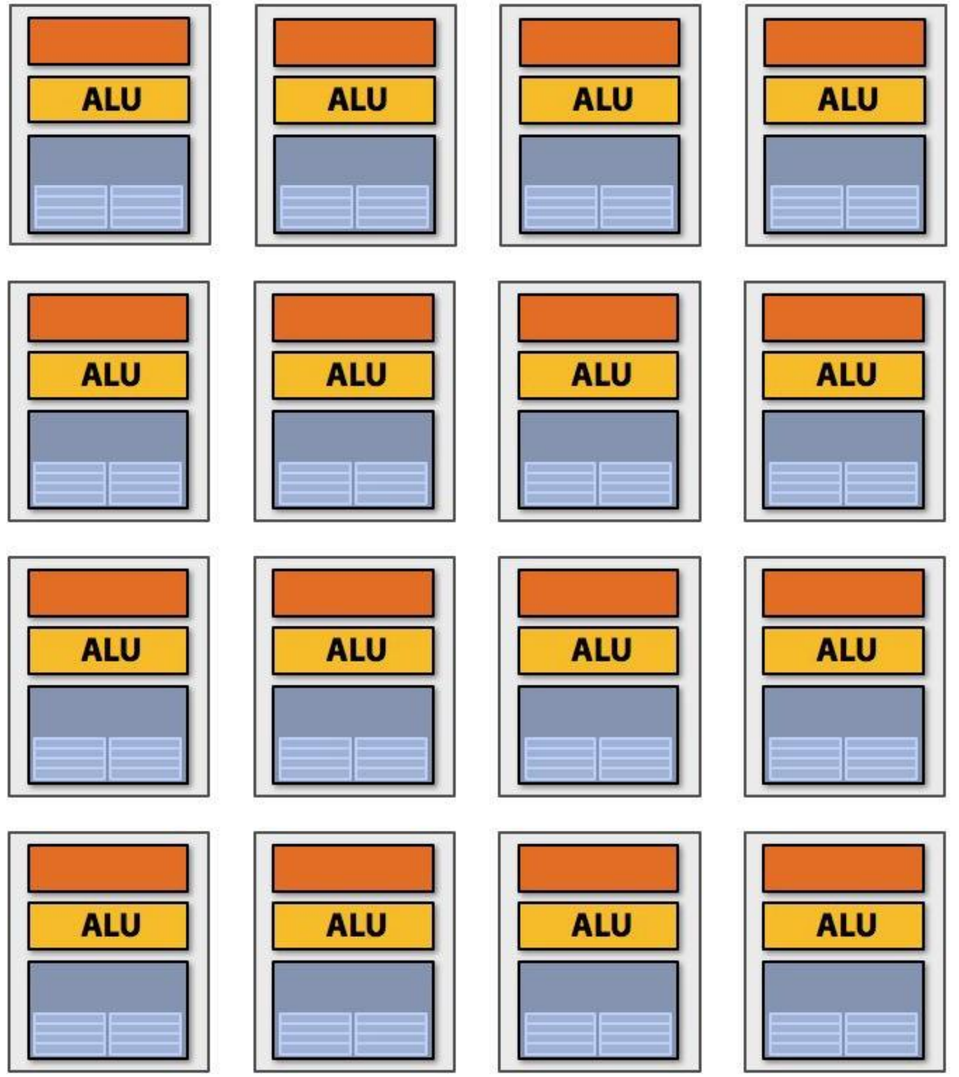
Four cores: compute four elements in parallel





Sixteen cores: compute sixteen elements in parallel

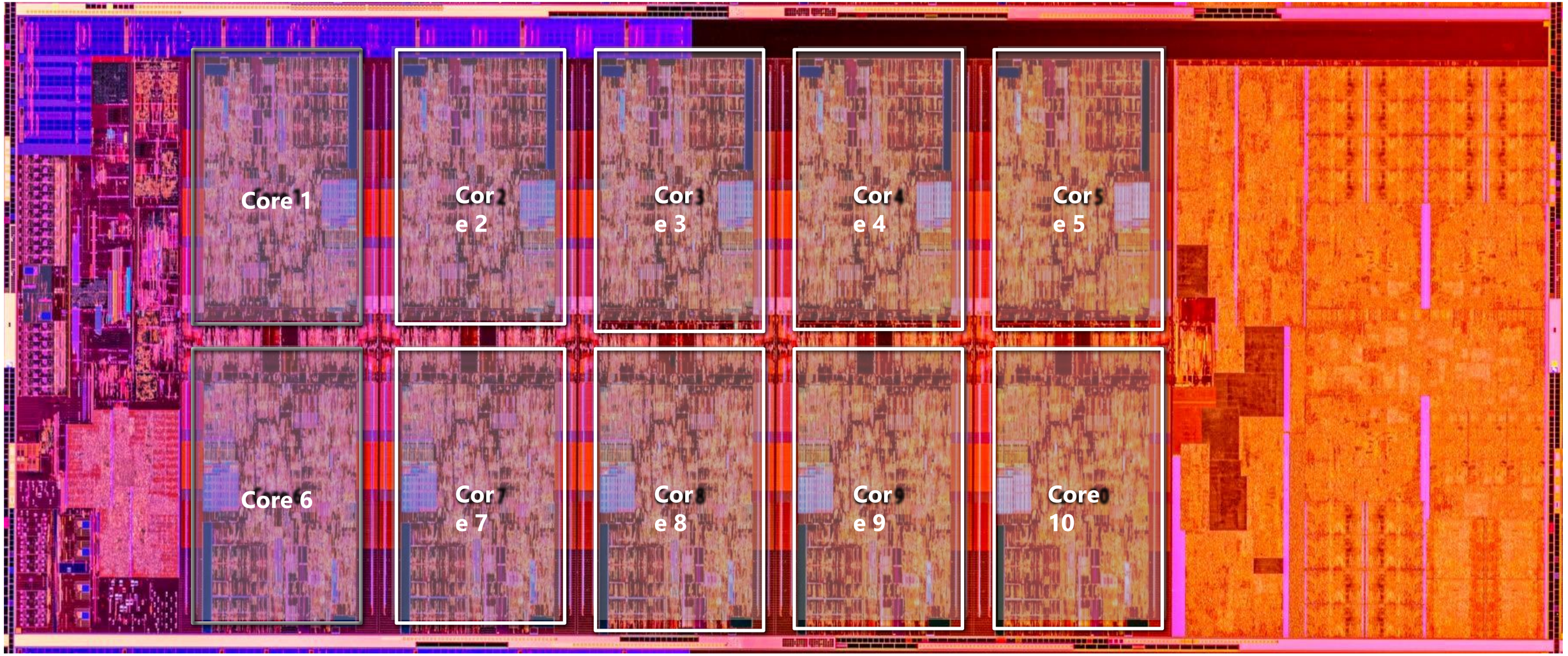




Sixteen cores, sixteen simultaneous instruction streams

Example: multi-core CPU

Intel “Comet Lake” 10th Generation Core i9 10-core CPU (2020)

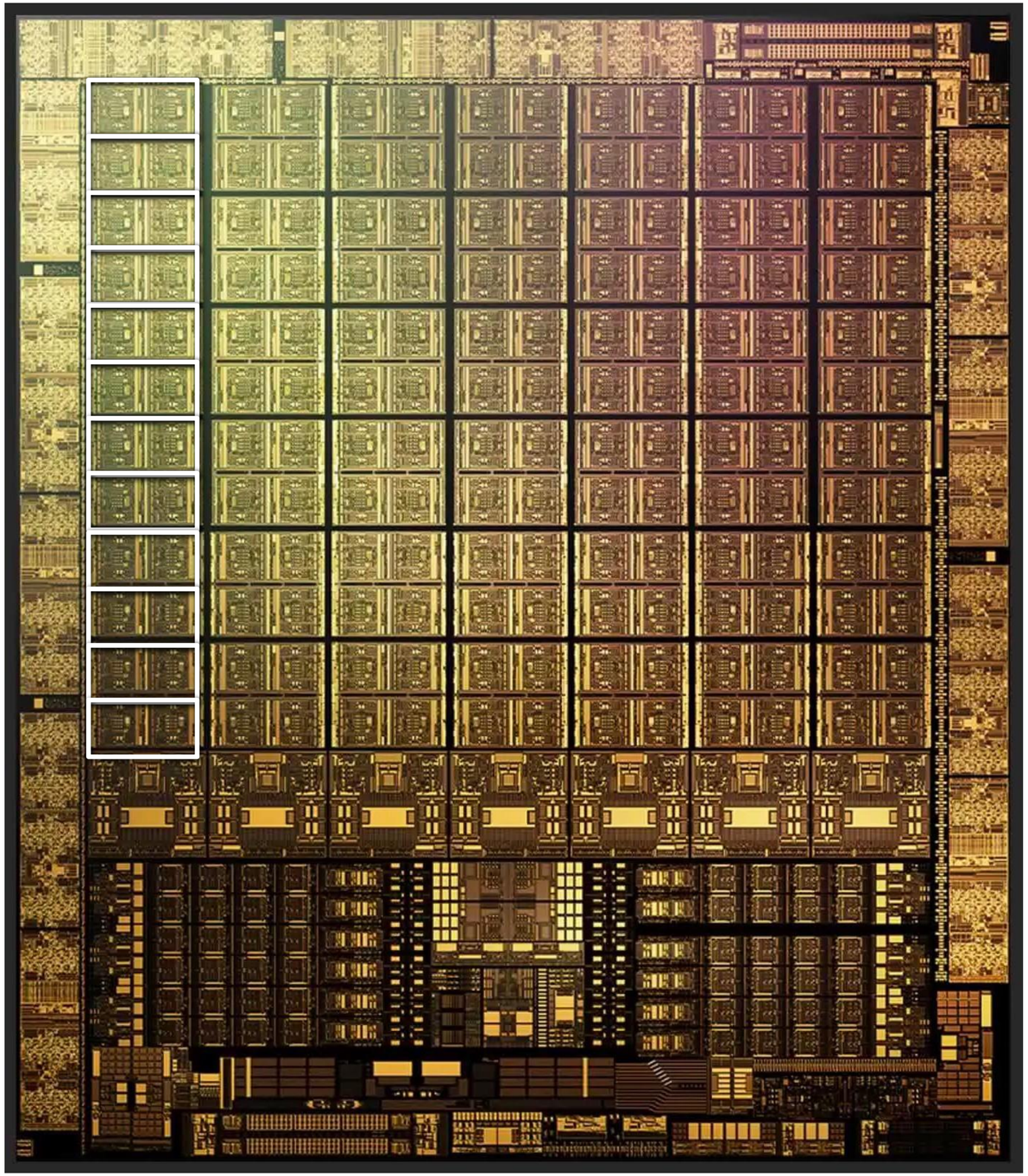


Multi-core GPU

NVIDIA Ampere

GPU 84 “SM” cores

(2020)

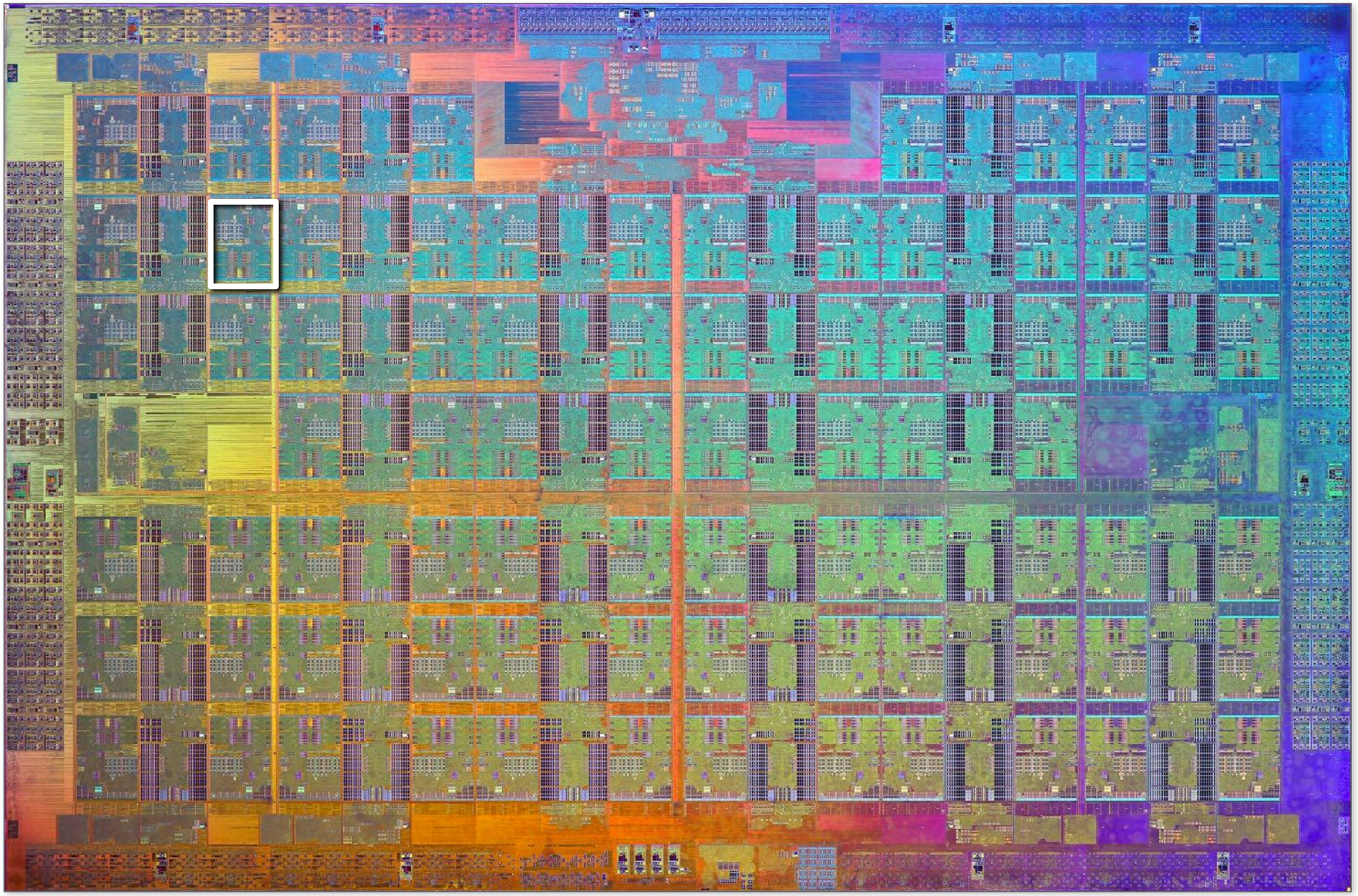


Intel Xeon Phi

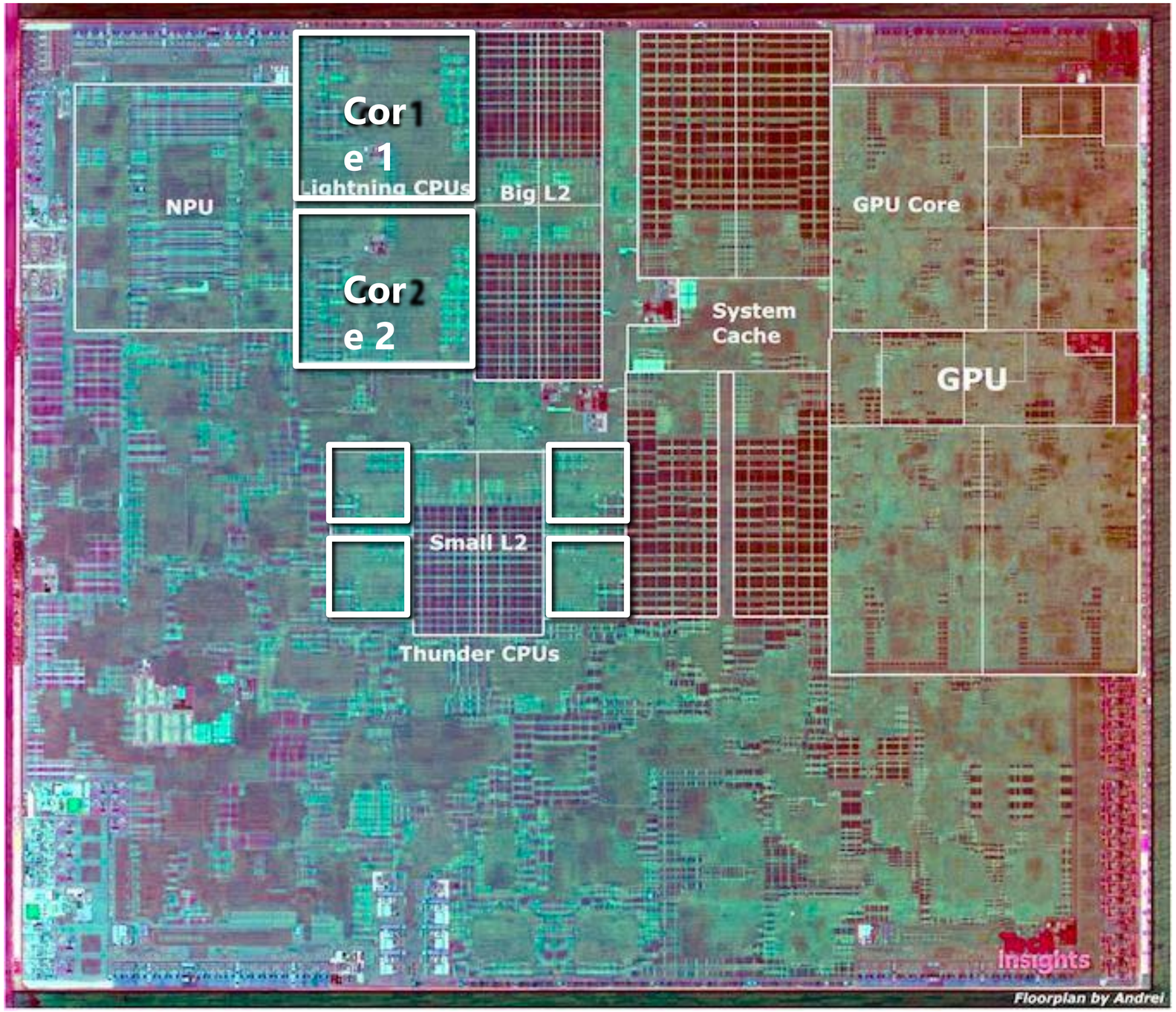
“Knights Corner“

72-core CPU

(2016)



Apple A13:
Two "big" cores + four "small" cores (2019)



A13 Image Credit: Anandtech / TechInsights Inc.

Data-parallel expression

(fictitious data-parallel language)

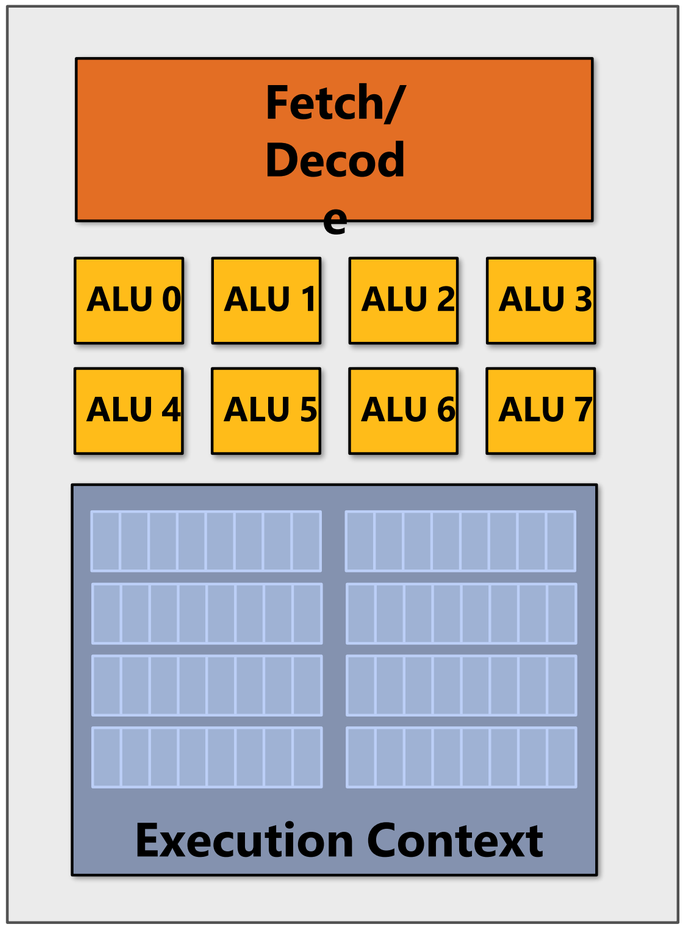
```
void sinx(int N, int terms, float* x, float* result)
{
   // declares that loop iterations are independent
   forall (int i from 0 to N)
   {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      int denom = 6;  // 3!
      int sign = -1;

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

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

Another interesting property of this code:

Parallelism is across iterations of the loop.

All the iterations of the loop carry out the exact same sequence of instructions, but on di$erent input data (Here: to compute the sine of x[i])

Add execution units (ALUs) to increase compute capability



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

SIMD processing

Single instruction, multiple data

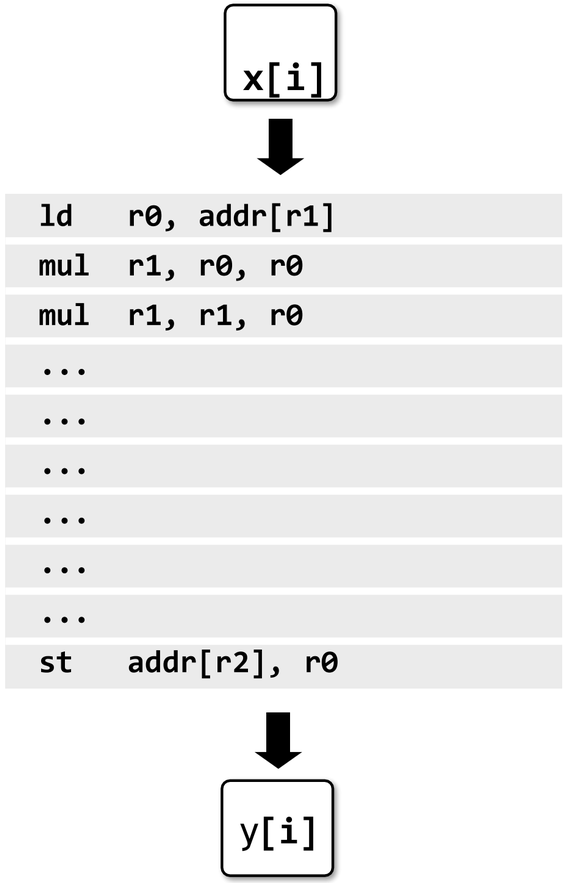
Same instruction broadcast to all ALUs
This operation is executed in parallel on all ALUs

Recall our original scalar program

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

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

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

Original compiled program:

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



Vector program (using AVX intrinsics)

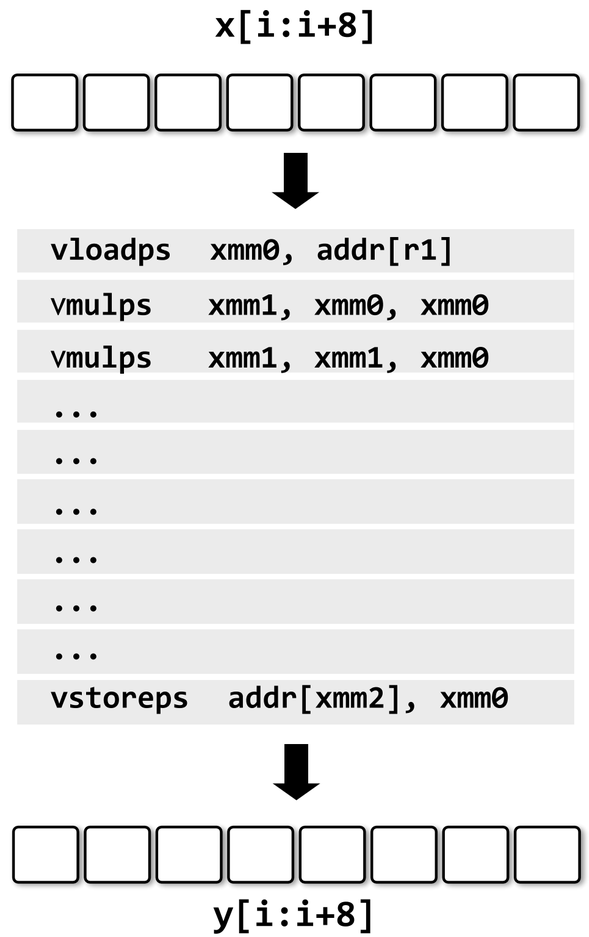
```
#include <immintrin.h>
void sinx(int N, int terms, float* x, float* y)
{
   float three_fact = 6;   // 3!
   for (int i=0; i<N; i+=8)
   {
      __m256 origx = _mm256_load_ps(&x[i]);
      __m256 value = origx;
      __m256 numer = _mm256_mul_ps(origx, _mm256_mul_ps(origx, origx));
      __m256 denom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

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

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

Intrinsic datatypes and functions available to C programmers

Intrinsic functions operate on vectors of eight 32-bit values (e.g., vector of 8 floats)

Vector program (using AVX intrinsics)

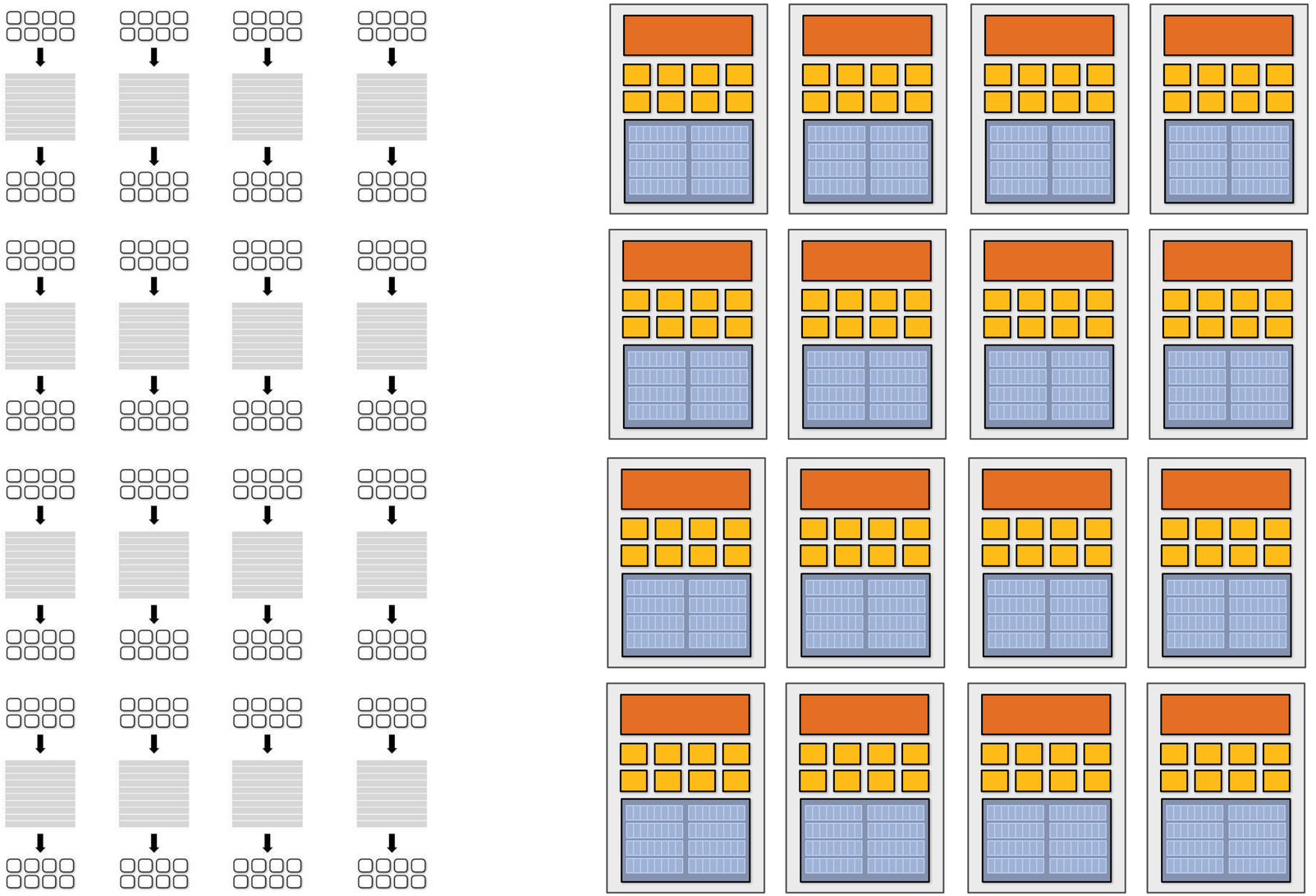
```
#include <immintrin.h>

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

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

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



Compiled program:

Processes eight array elements simultaneously using vector instructions on 256-bit vector registers

16 SIMD cores: 128 elements in parallel



16 cores, 128 ALUs, 16 simultaneous instruction streams

Data-parallel expression

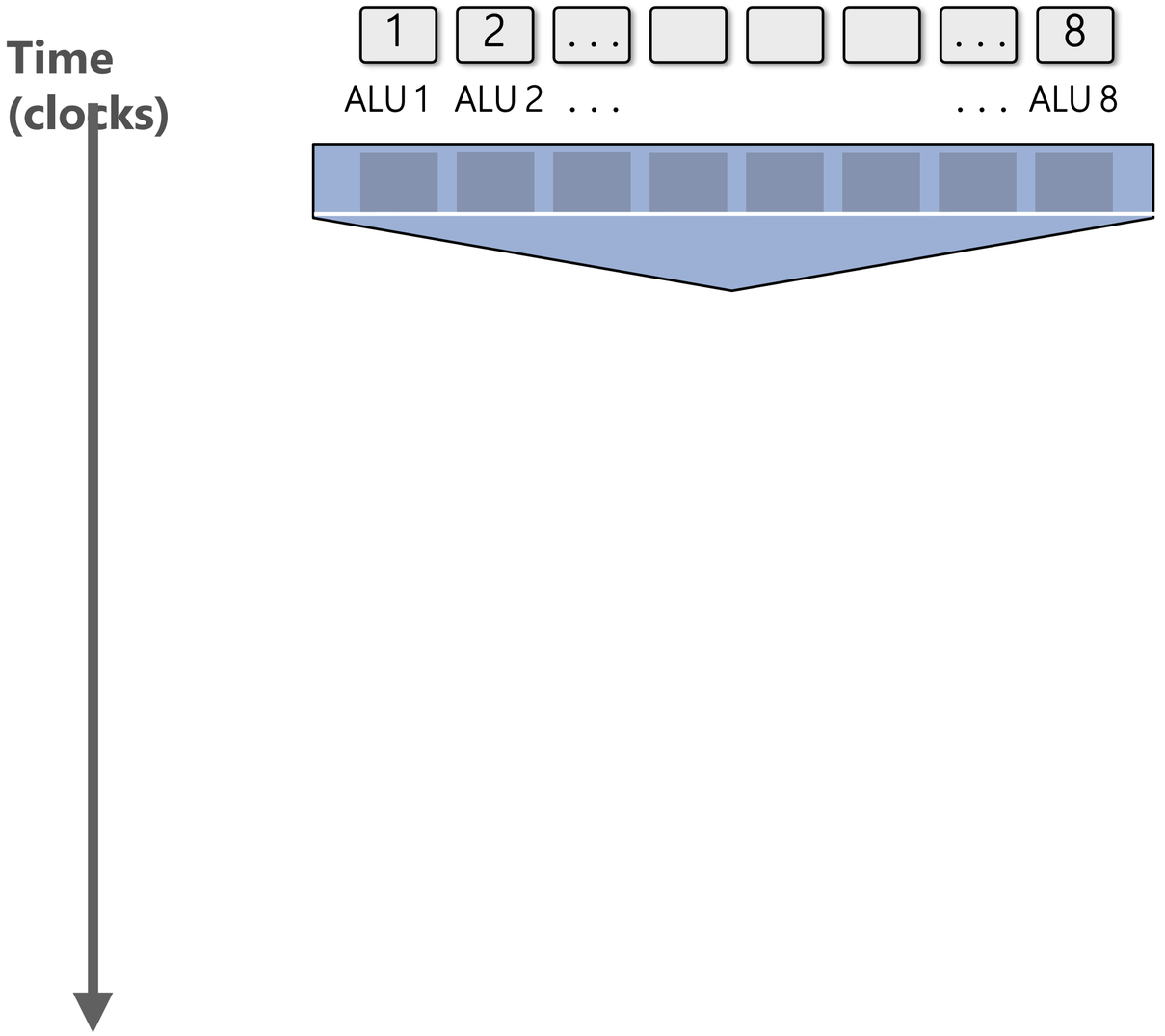
(fictitious data-parallel language)

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

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

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

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

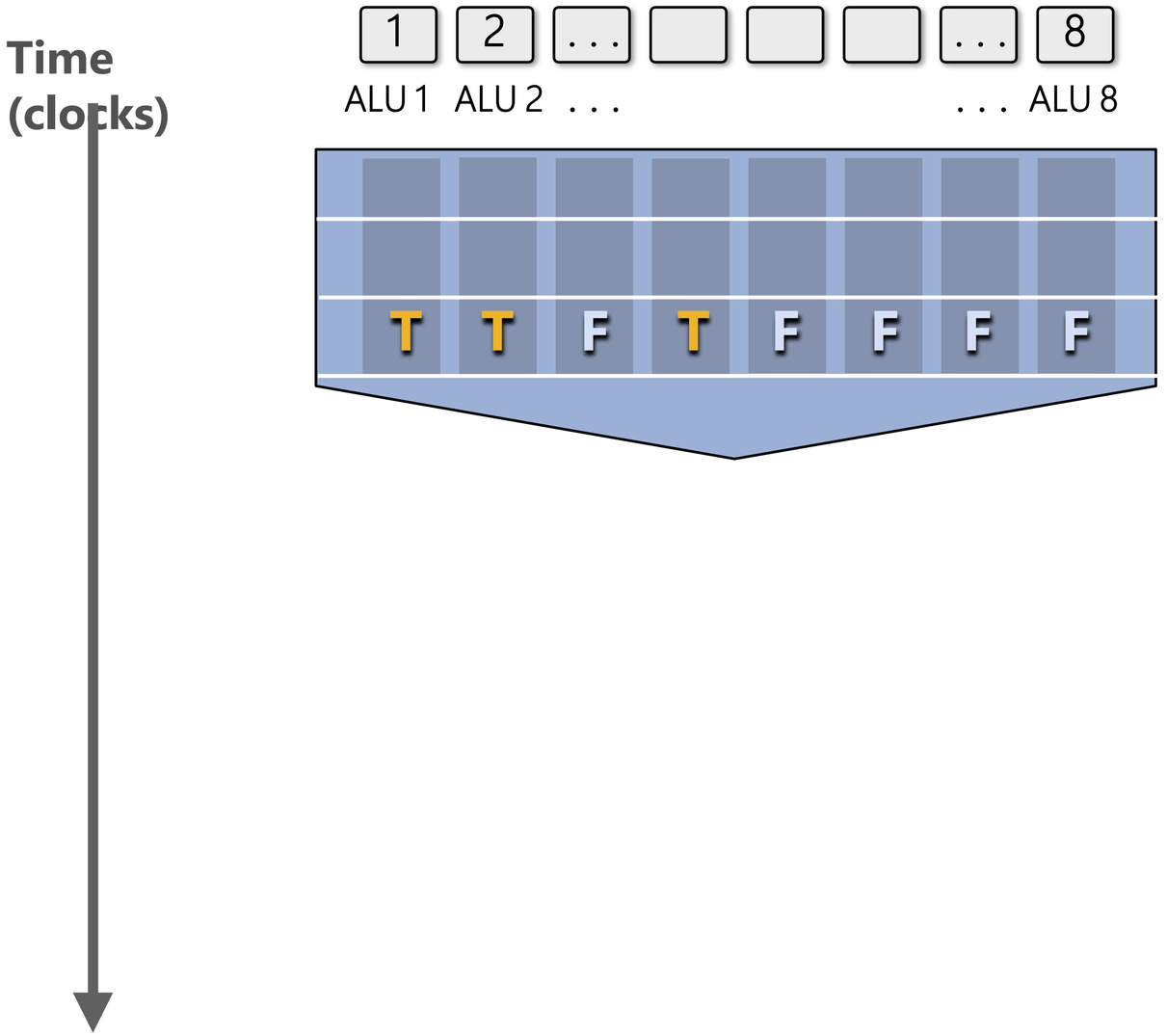
Abstraction facilitates automatic generation of both multi-core parallel code, and vector instructions to make use of SIMD processing capabilities within a core.

What about conditional execution?



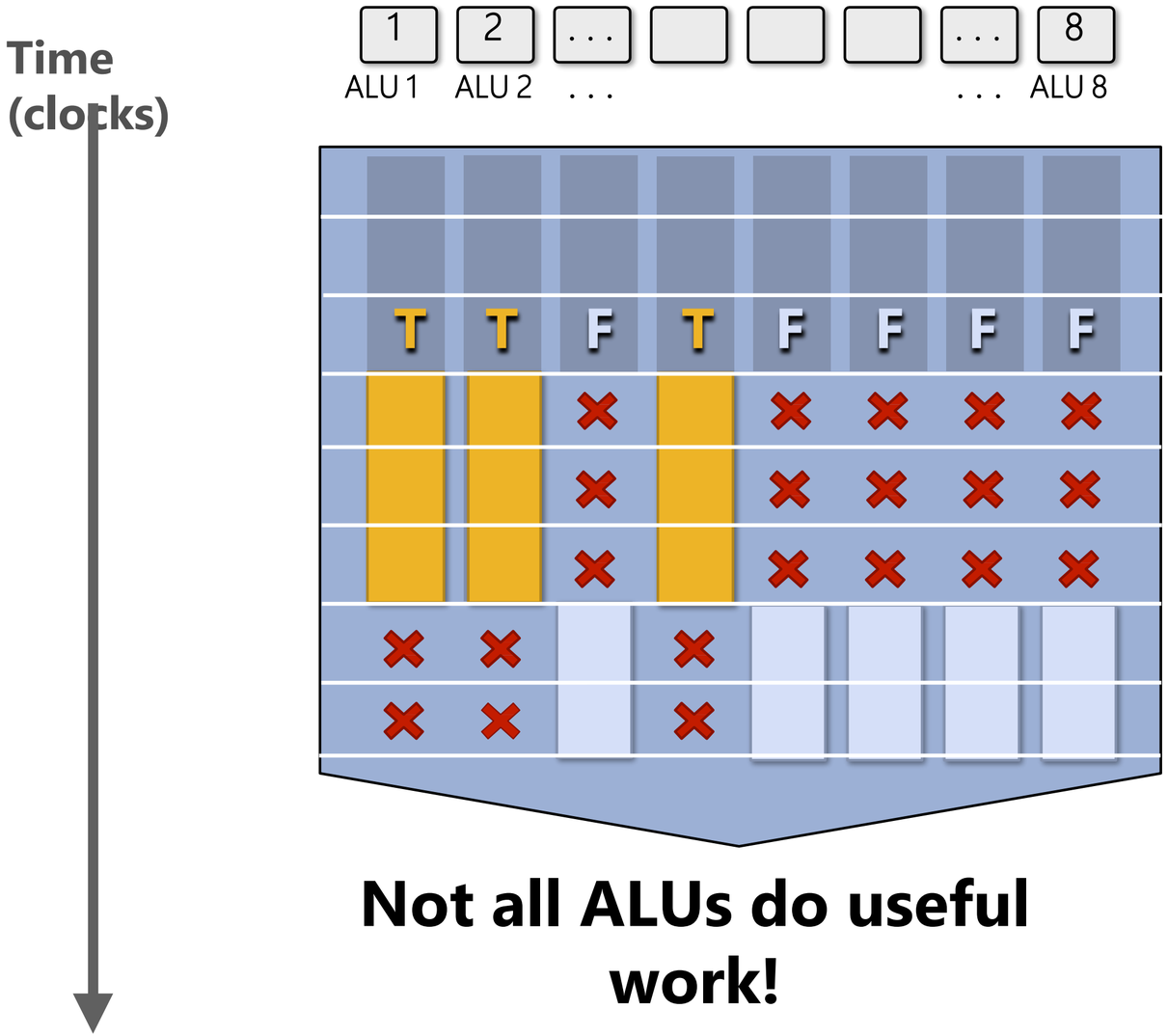

```
forall (int i from 0 to N) {
    float t = x[i];
    <unconditional code>

    if (t > 0.0) {
       t = t * t;
       t = t * 50.0;
       t = t + 100.0;
    } else {
       t = t + 30.0;
       t = t / 10.0;
    }
   <resume unconditional code>
   y[i] = t;
}
```

What about conditional execution?




```
forall (int i from 0 to N) {
    float t = x[i];
    <unconditional code>
    if (t > 0.0) {
       t = t * t;
       t = t * 50.0;
       t = t + 100.0;
    } else {
       t = t + 30.0;
       t = t / 10.0;
    }
   <resume unconditional code>
   y[i] = t;
}
```

Mask (discard) output of ALU

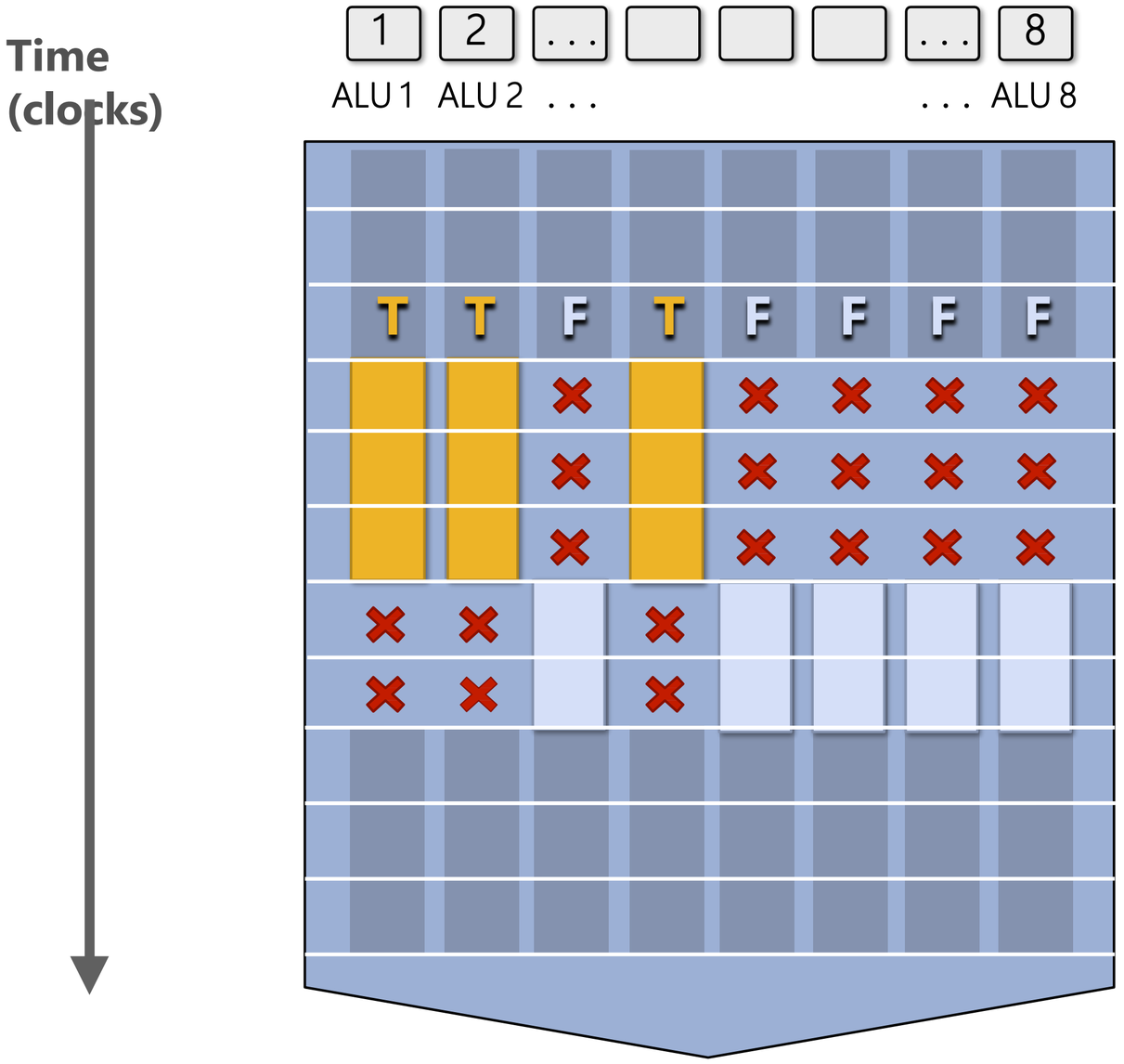



```
forall (int i from 0 to N) {
    float t = x[i];
    <unconditional code>
    if (t > 0.0) {
        t = t * t;
        t = t * 50.0;
        t = t + 100.0;
    } else {
        t = t + 30.0;
        t = t / 10.0;
    }
    <resume unconditional code>
    y[i] = t;
}
```

Not all ALUs do useful work!

Worst case: 1/8 peak performance

After branch: continue at full performance

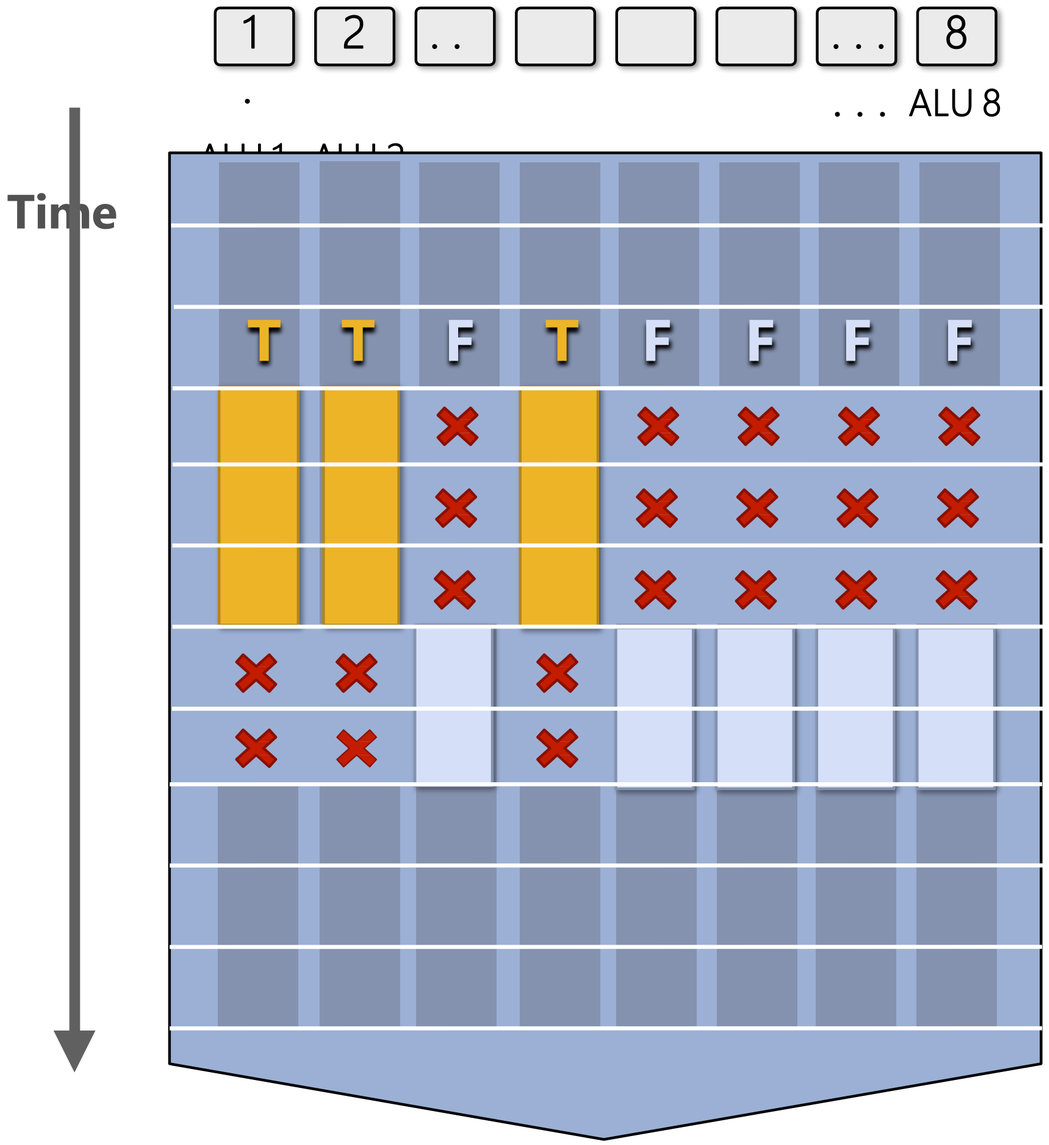



```
forall (int i from 0 to N) {
    float t = x[i];
    <unconditional code>
    if (t > 0.0) {
        t = t * t;
        t = t * 50.0;
        t = t + 100.0;
    } else {
        t = t + 30.0;
        t = t / 10.0;
    }
    <resume unconditional code>
    y[i] = t;
}
```

Breakout question

Can you think of piece of code that yields the worst case performance on a processor with 8-wide SIMD execution?

Hint: can you create it using only a single "if" statement?




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

    if (t > 0.0) {
        ???
    } else {
        ???
    }

    <resume unconditional code>
    y[i] = t;
}
```

Some common terminology

- **Instruction stream coherence (“coherent execution”)**
  - **Property of a program where the same instruction sequence applies to many data elements**
  - **Coherent execution IS NECESSARY for SIMD processing resources to be used efficiently**
  - **Coherent execution IS NOT NECESSARY for efficient parallelization across different cores, since each core has the capability to fetch/decode a different instructions from their thread’s instruction stream**

- **“Divergent” execution**
  - **A lack of instruction stream coherence**

SIMD execution: modern CPU examples

- **Intel AVX2 instructions: 256 bit operations: 8x32 bits or 4x64 bits (8-wide float vectors)**
- **Intel AVX512 instruction: 512 bit operations: 16x32 bits…**
- **ARM Neon instructions: 128 bit operations: 4x32 bits…**

- **Instructions are generated by the compiler**
  - **Parallelism explicitly requested by programmer using intrinsics**
  - **Parallelism conveyed using parallel language semantics (e.g., `forall` example)**
  - **Parallelism inferred by dependency analysis of loops by "auto-vectorizing" compiler**

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

SIMD execution on many modern GPUs

- "Implicit SIMD"
  - Compiler generates a binary with scalar instructions
  - But N instances of the program are always run together on the processor
  - Hardware (not compiler) is responsible for simultaneously executing the same instruction from multiple program instances on different data on SIMD ALUs

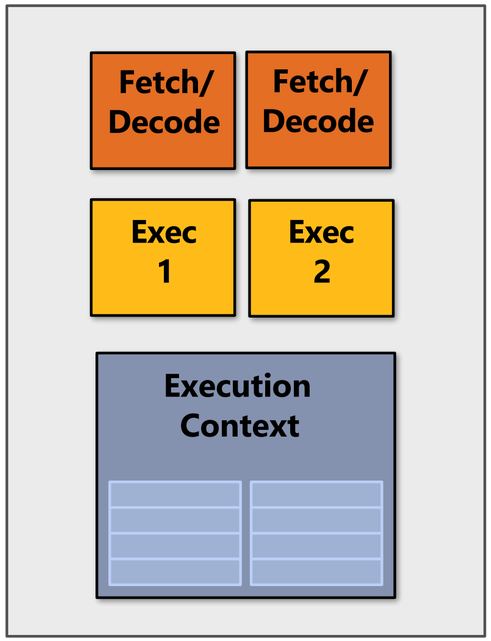
- SIMD width of most modern GPUs ranges from 8 to 32
  - Divergent execution can be a big issue
    (poorly written code might execute at 1/32 the peak capability of the machine!)

Summary: three different forms of parallel execution

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

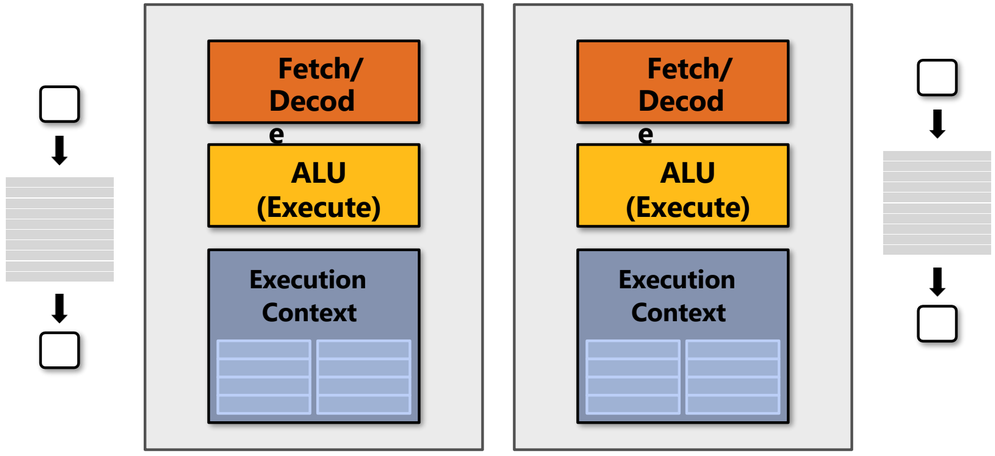
- **SIMD:** multiple ALUs controlled by same instruction (within a core)
  - Efficient for data-parallel workloads: amortize control costs over many ALUs
  - Vectorization done by compiler (explicit SIMD) or at runtime by hardware (implicit SIMD)

- **Multi-core:** use multiple processing cores
  - Provides thread-level parallelism: simultaneously execute a completely different instruction
    stream on each core
  - Software creates threads to expose parallelism to hardware (e.g., via threading API)

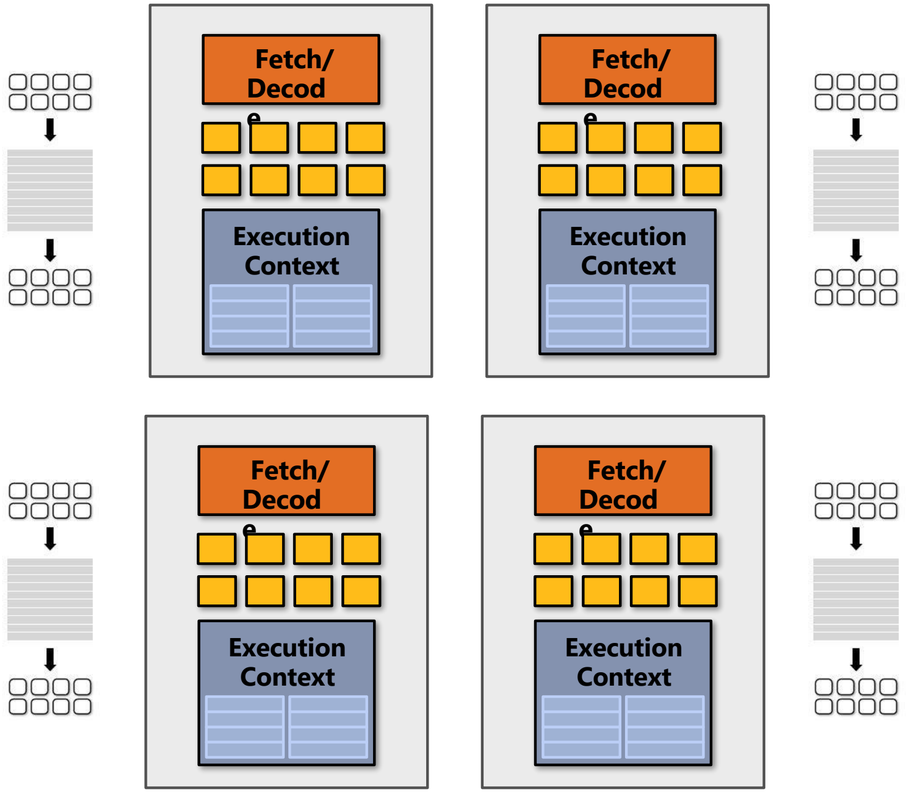


My single core, superscalar processor: executes up to two instructions per clock from a single instruction stream (if the instructions are independent)

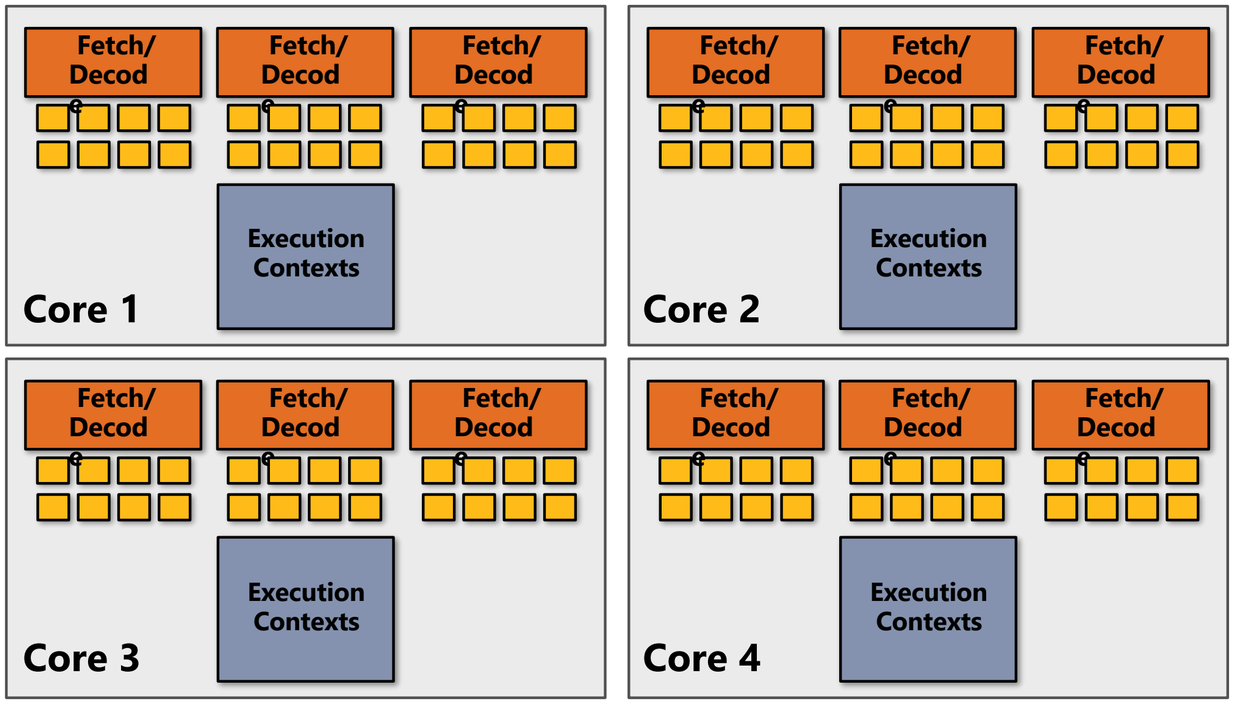
My dual-core processor:
executes one instruction per clock
from one instruction stream on each core.



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



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

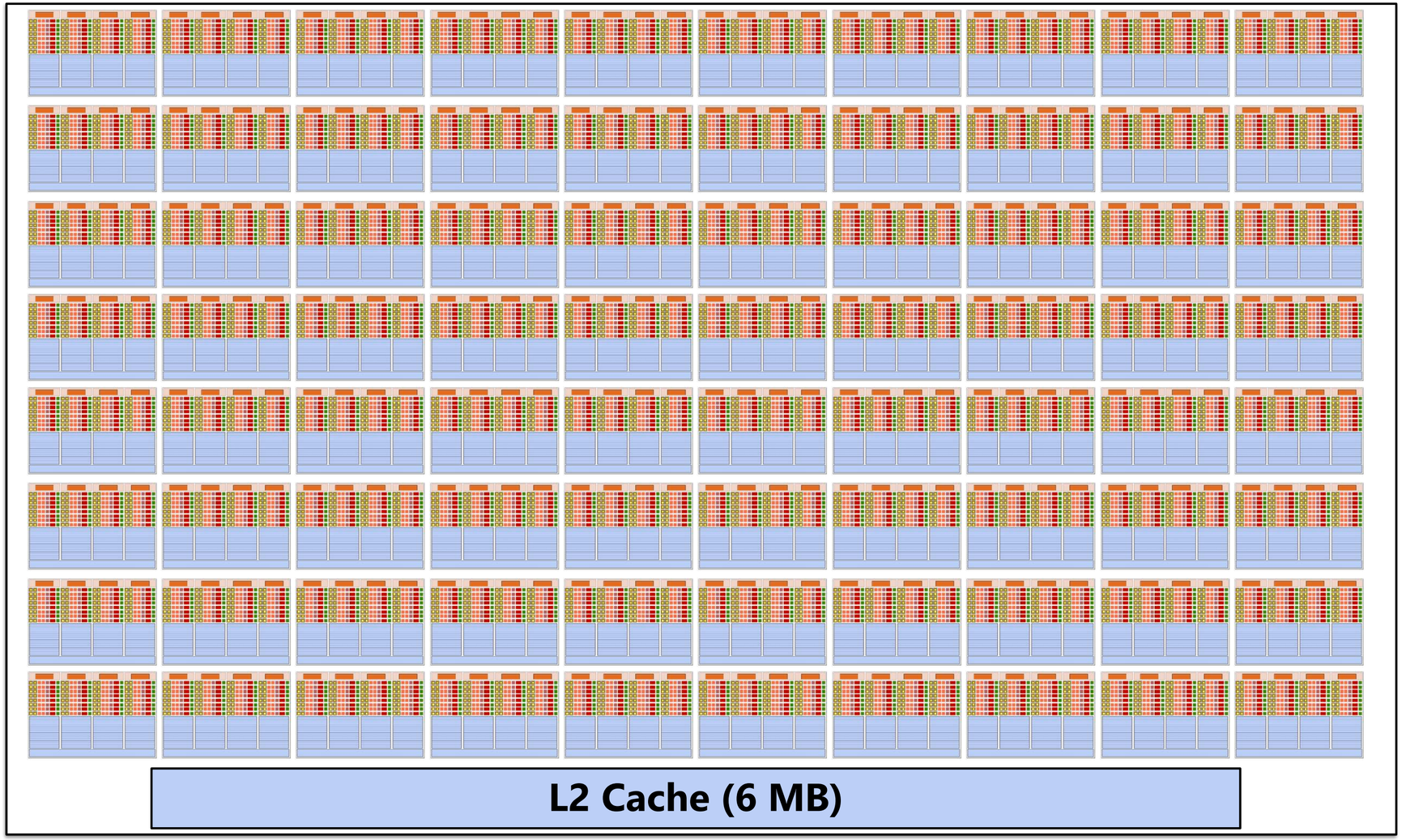


4 core processor
Three 8-wide SIMD ALUs per core (AVX2 instructions)

4 cores x 8-wide SIMD x 3 x 4.2 GHz = 400 GFLOPs

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

Example: NVIDIA V100 GPU



80 “SM” cores

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