Course announcement

- **Training 2 is due today**
- **I realized that many of you lack experience in programming dynamic programming algorithms**
  - The next training (training 3) will be a special session for dynamic programming
- **If you want to know what's wrong with your submission, you can stay here after the lecture and I can help you check your code**
- **For those of you who have the problems, consider to include the correctness analysis**
  - You can get 0.5 candy for B and C if you show the correctness of your algorithm correctly (should be in a separate section)

CS142: Algorithm Engineering

An Overview of Computer Architecture

Yan Gu

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.

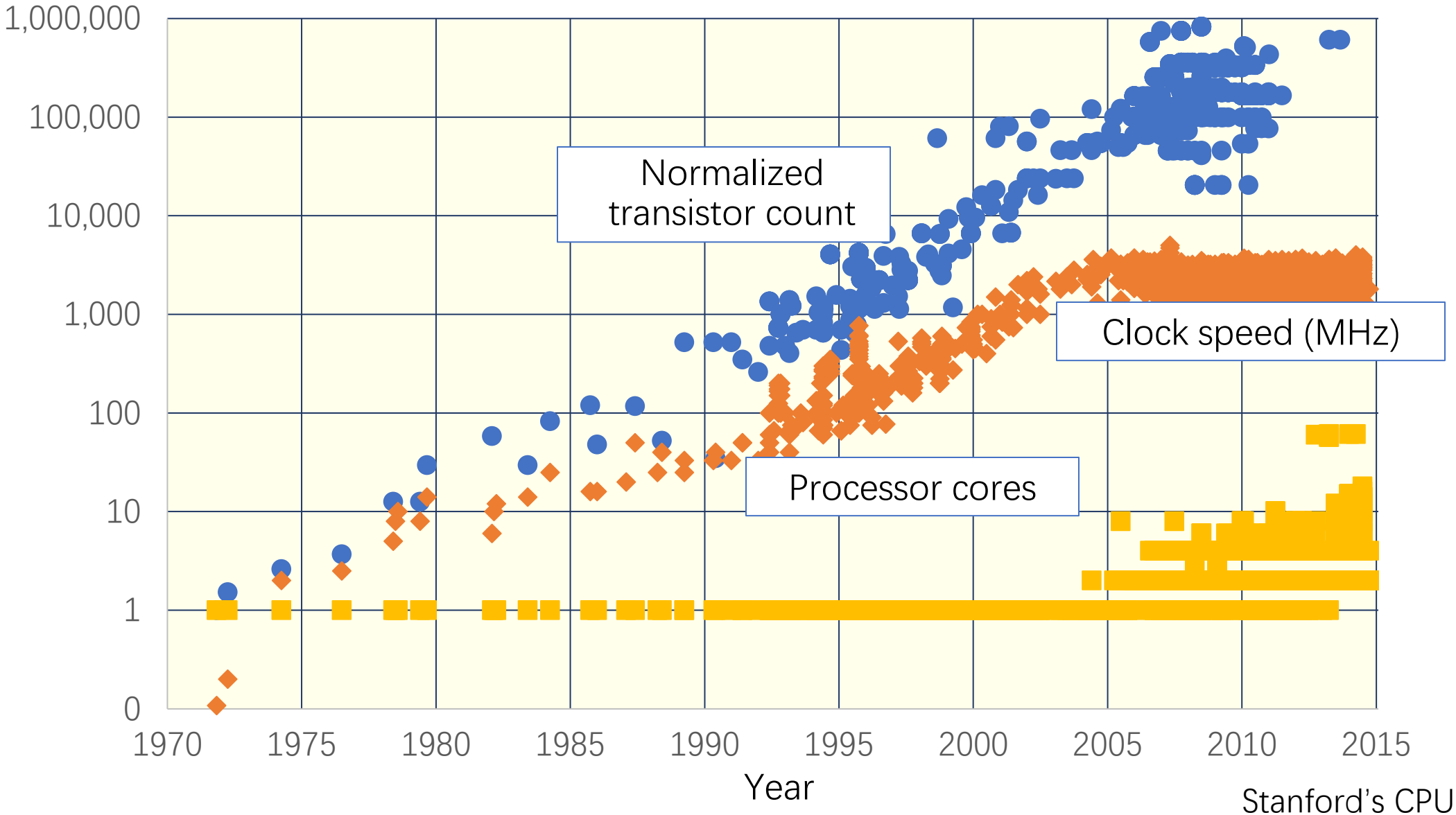
Lecture Overview

- **In the lectures 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)**
- **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**
  - No longer work after 2005 due to energy issues

- **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…

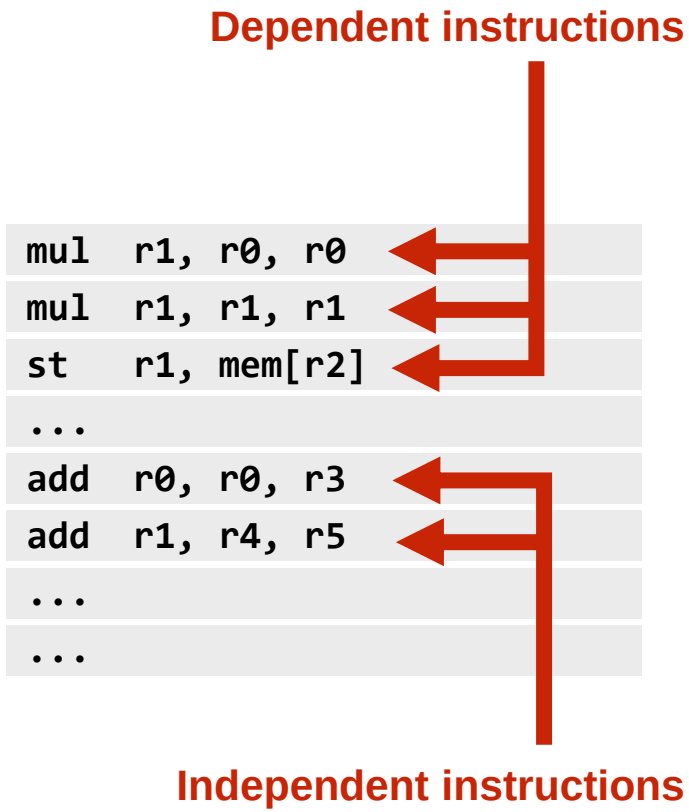
And so on…


```
_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
```

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



ILP example

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

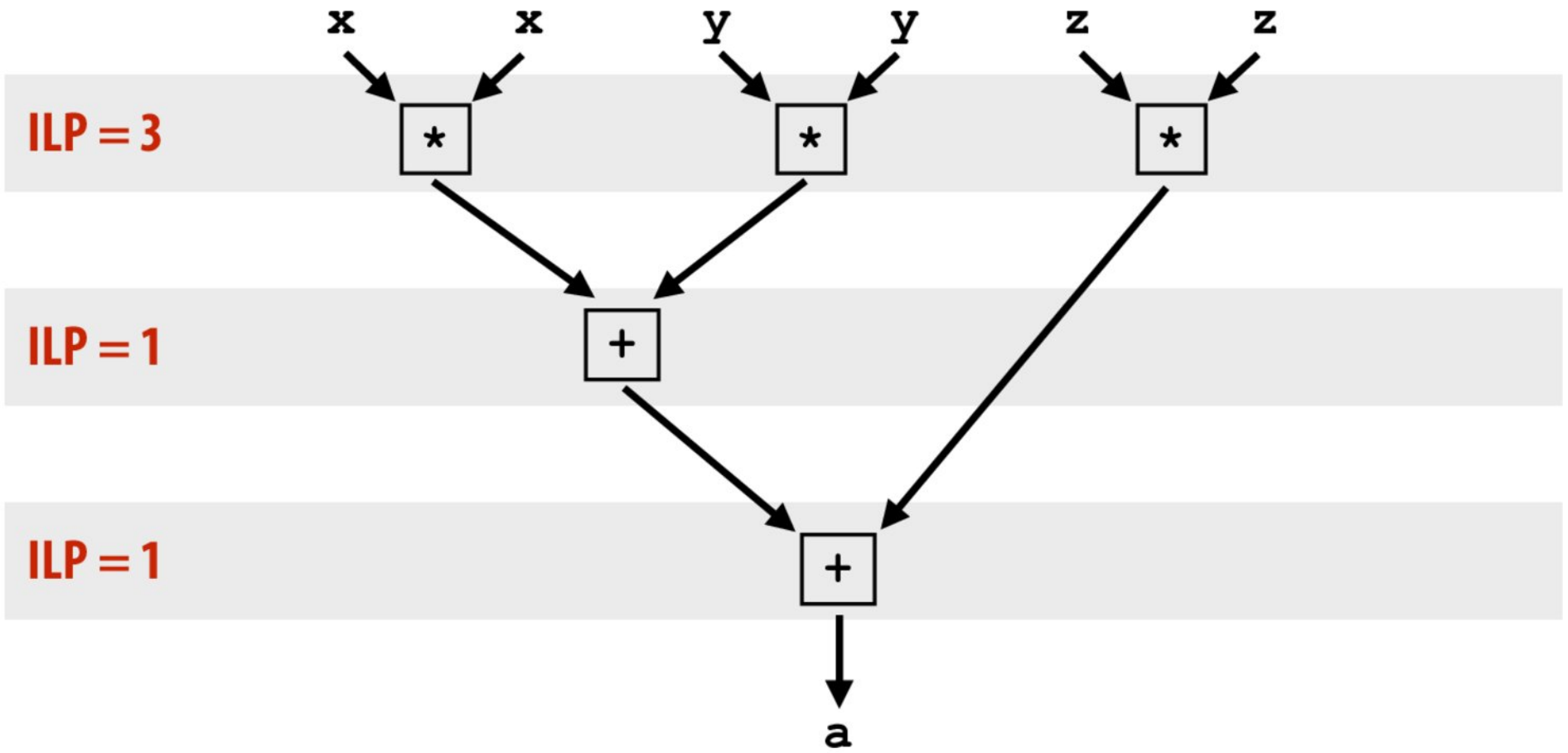
Consider the following program:

```
// 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

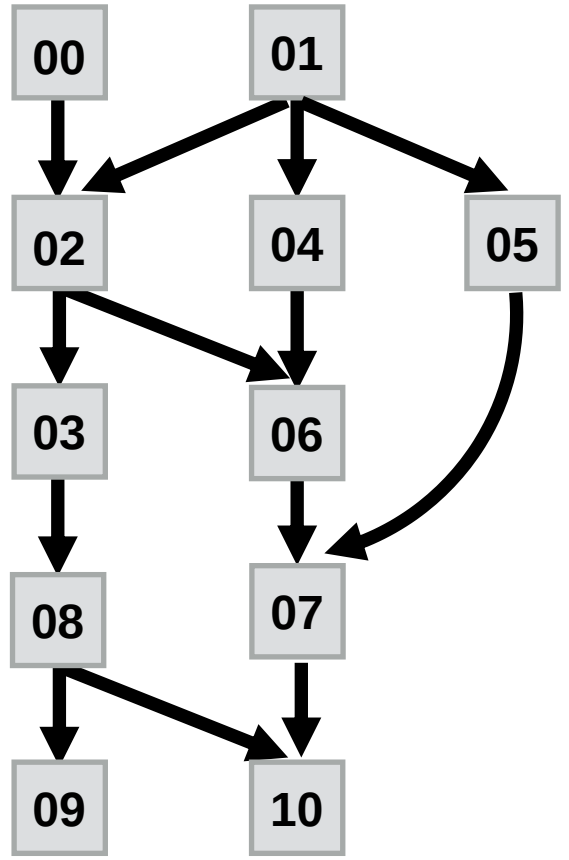
And instruction 5 must come after instructions 3 and 4

A more complex example

Program (sequence of instructions)

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

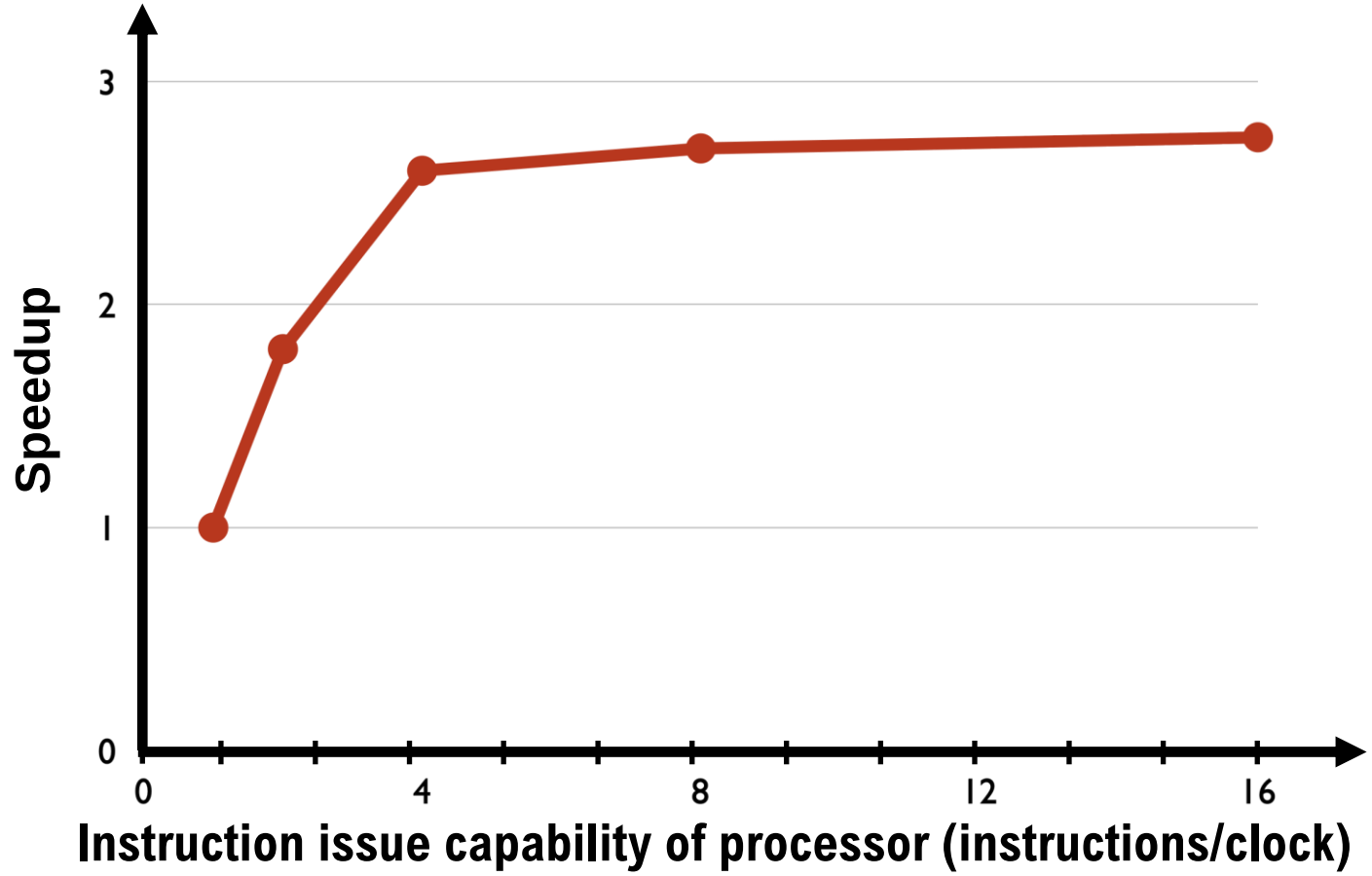
Instruction dependency graph



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)



Source: Culler & Singh (data from Johnson 1991)

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

- **Increasing CPU clock frequency**
  - No longer work after 2005 due to energy issues
- **Exploiting instruction-level parallelism (superscalar execution)**
  - Have a ceiling on the parallelism we can get (<3)

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;
   }
}
```

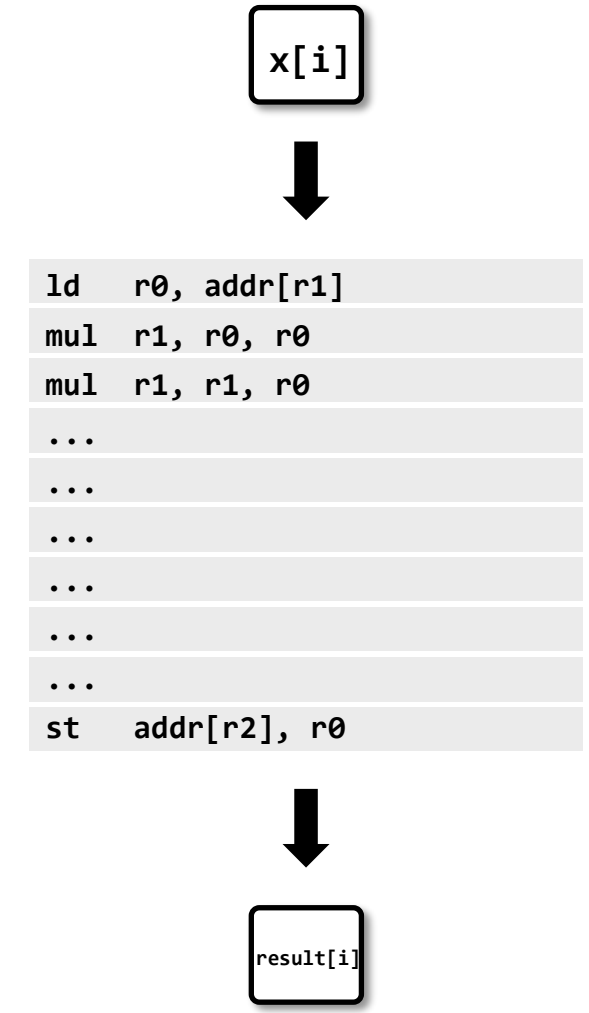
Compute $\sin(x)$ **using Taylor expansion:**

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

for each element of an array of n **floating-point numbers**

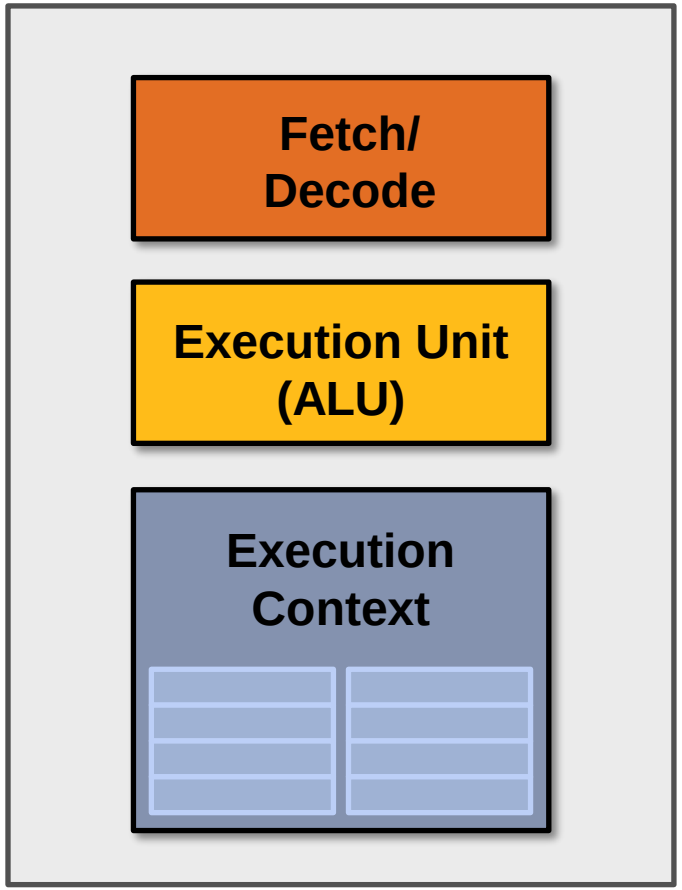
Compile program

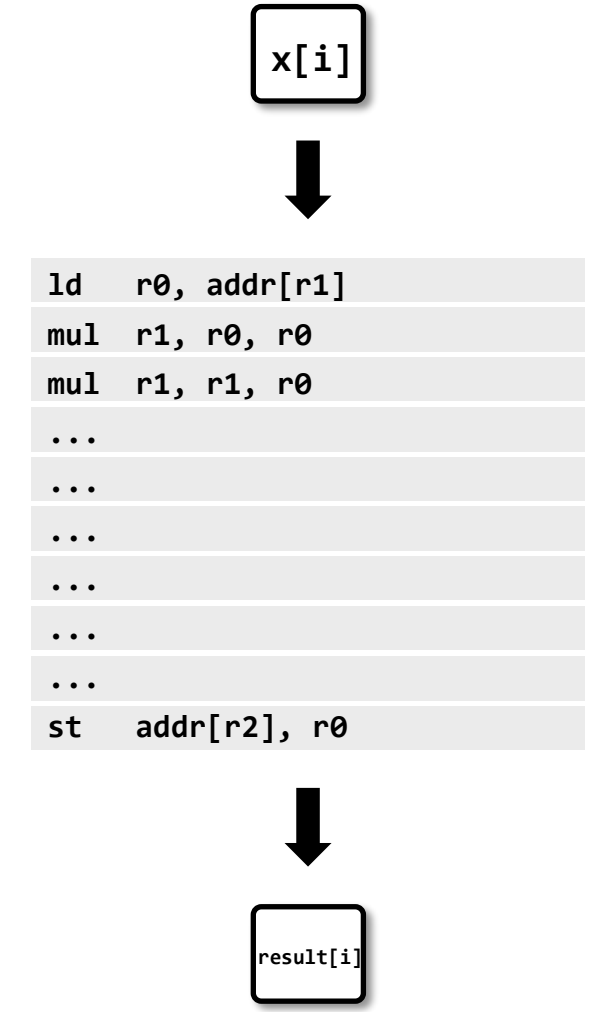
```
void sinx(int N, int terms, float* x, float* result)
{
   for (int i=0; i<N; i++)
   {
       float value = x[i];
       float numer = x[i] * x[i] * x[i];
       int denom = 6;  // 3!
       int sign = -1;

       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;
   }
}
```



Execute program

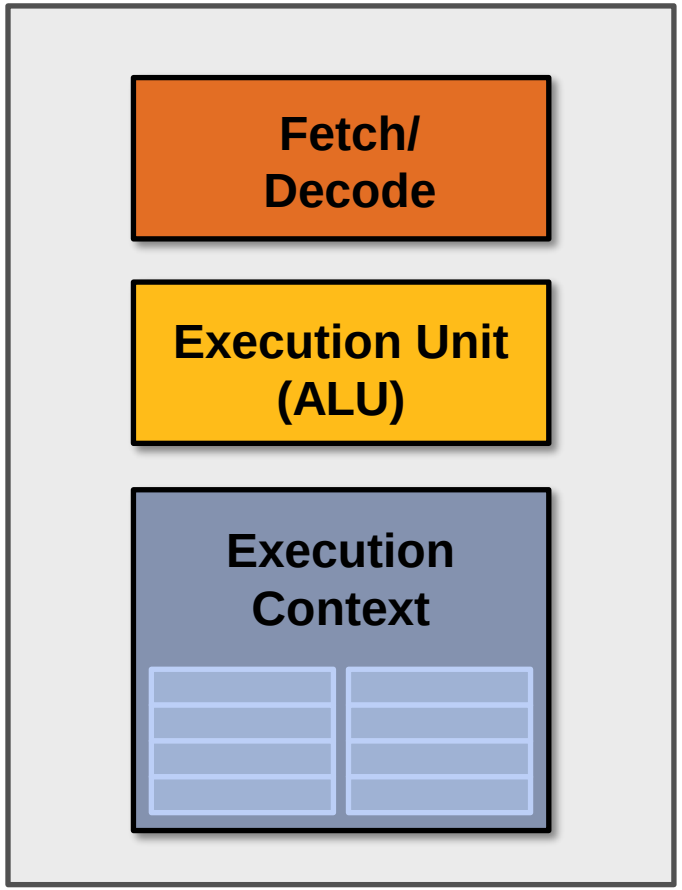
My very simple processor: executes one instruction per clock

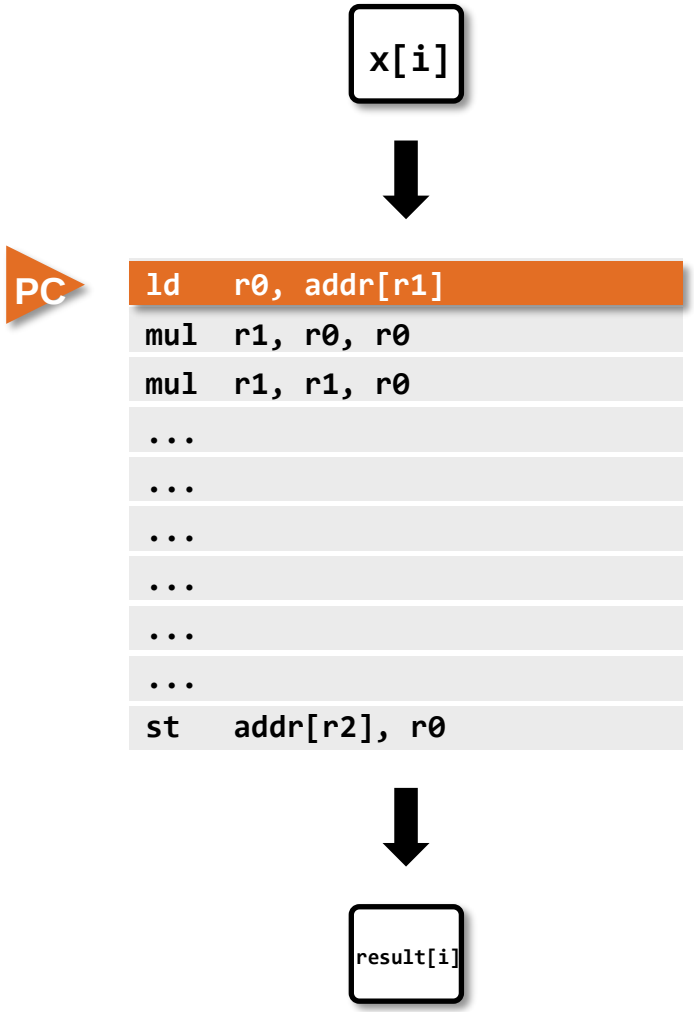




Execute program

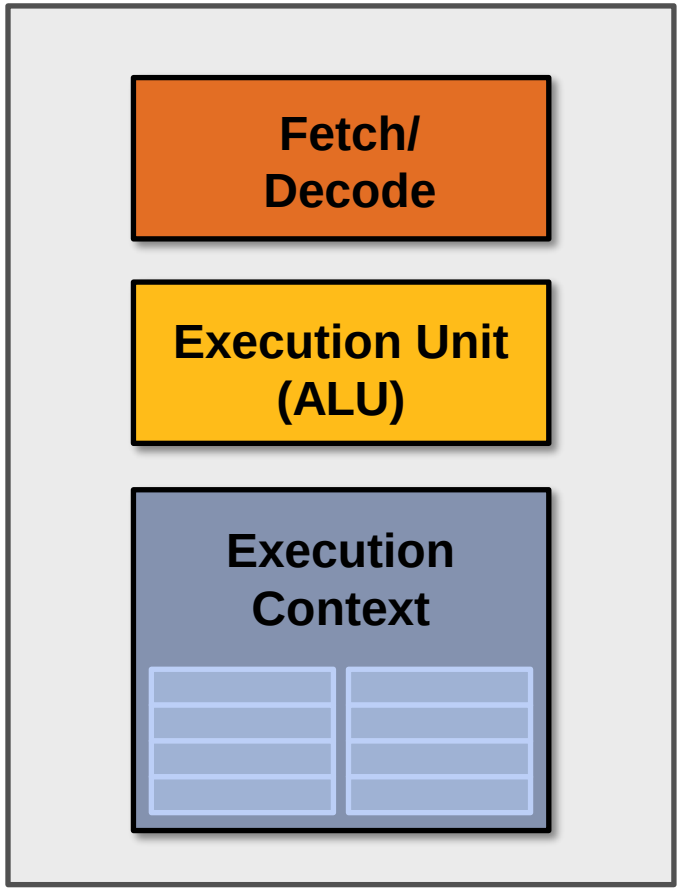
My very simple processor: executes one instruction per clock

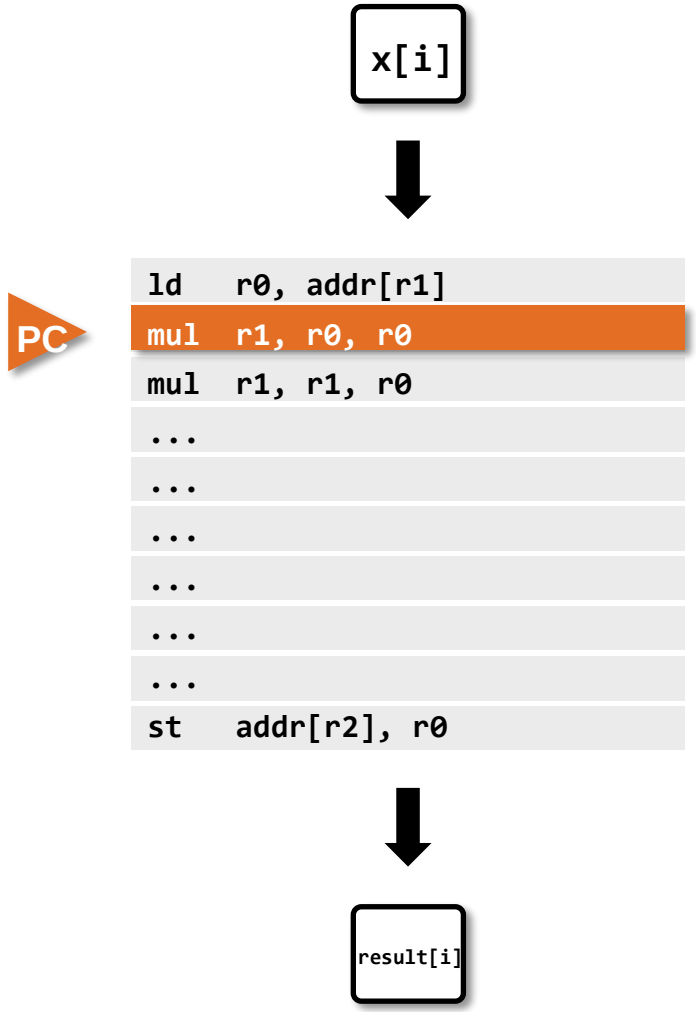




Execute program

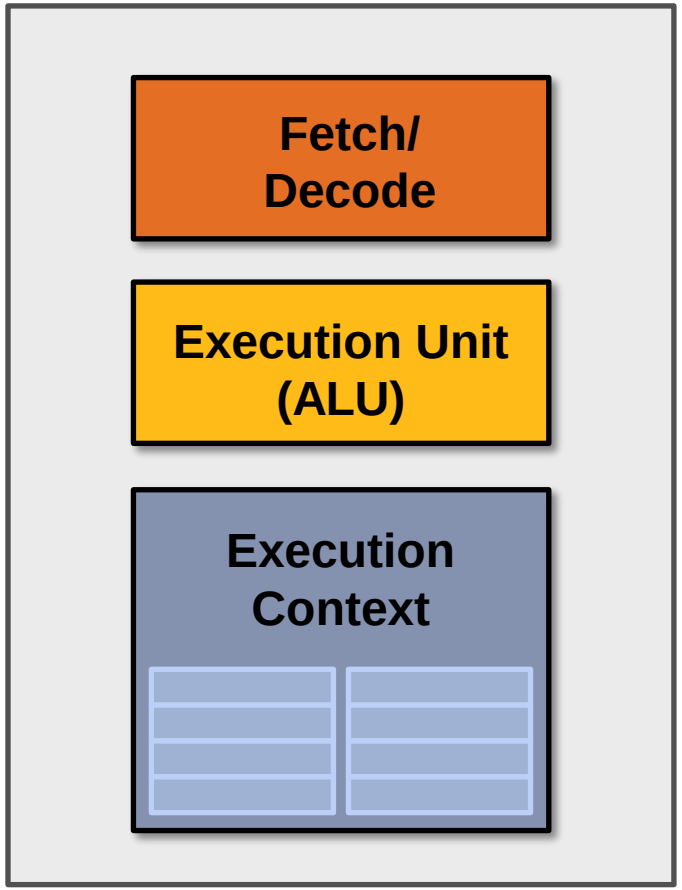
My very simple processor: executes one instruction per clock

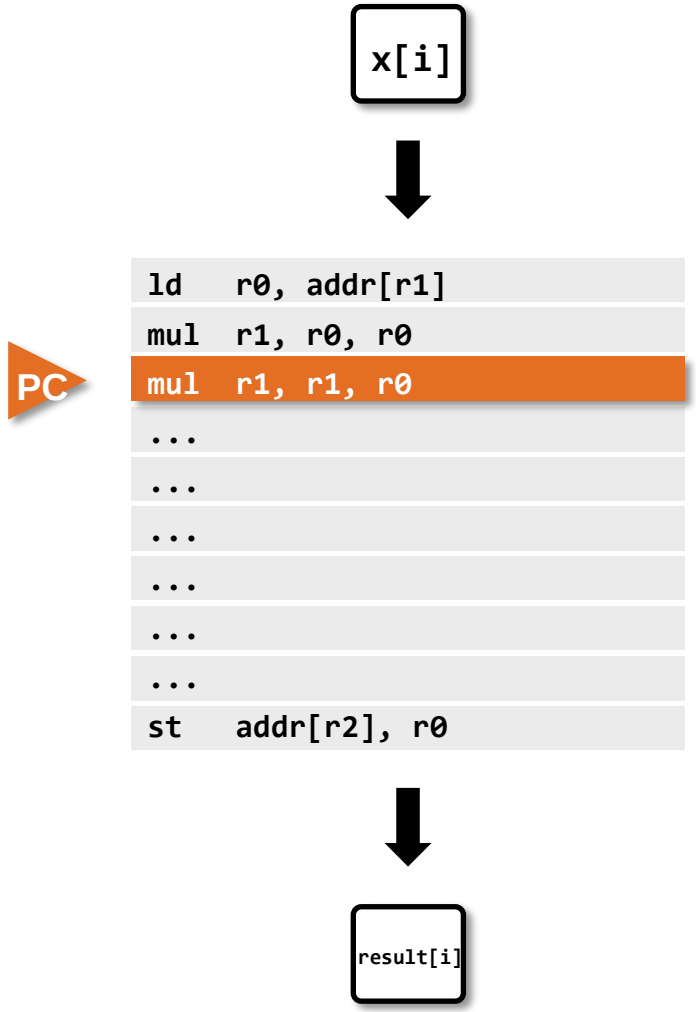




Execute program

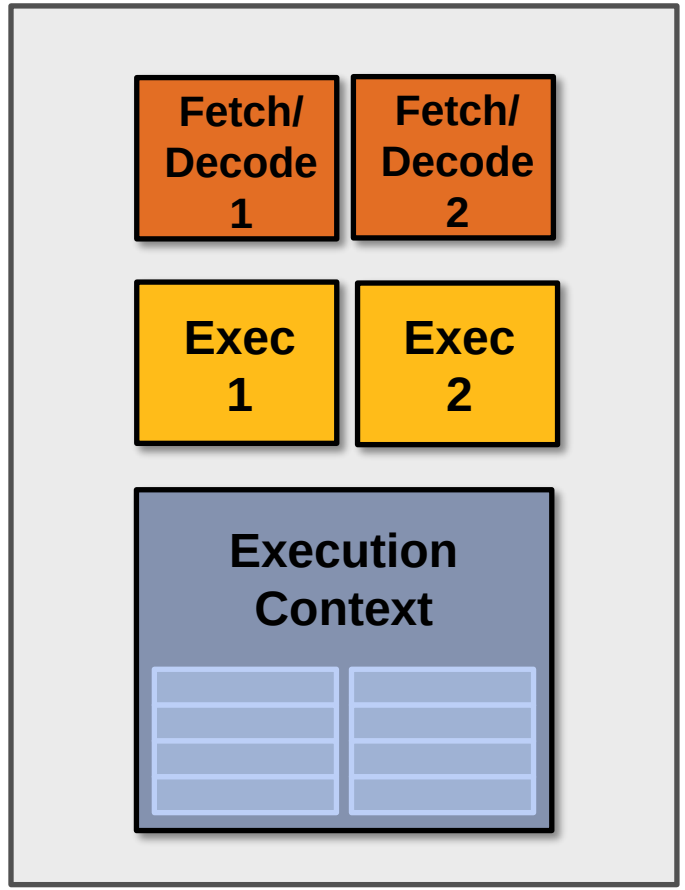
My very simple processor: executes one instruction per clock

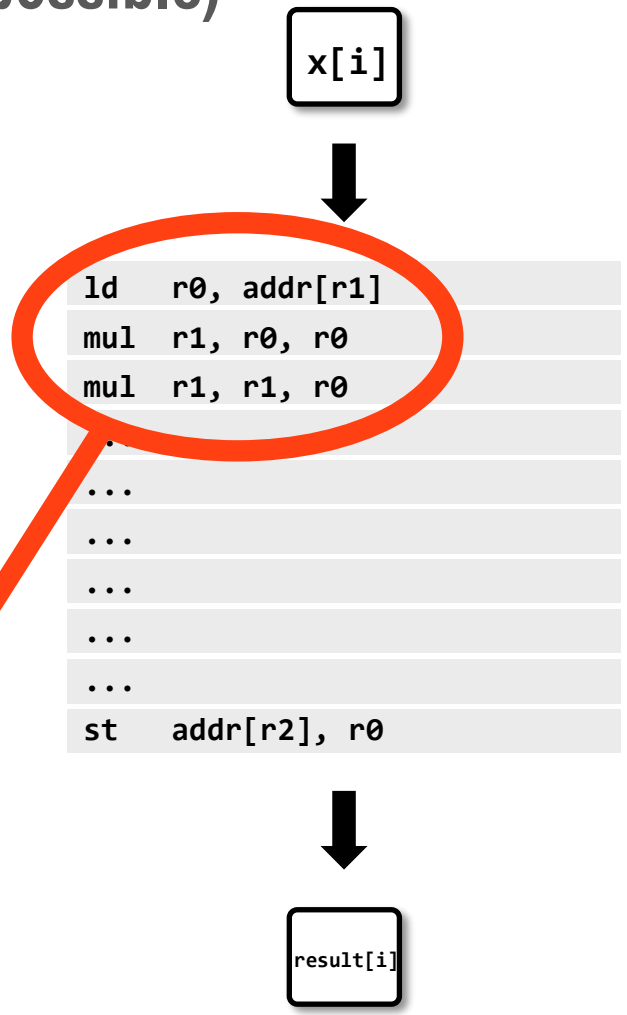




Superscalar processor

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





Note: No ILP exists in this region of the program

Aside: Pentium 4

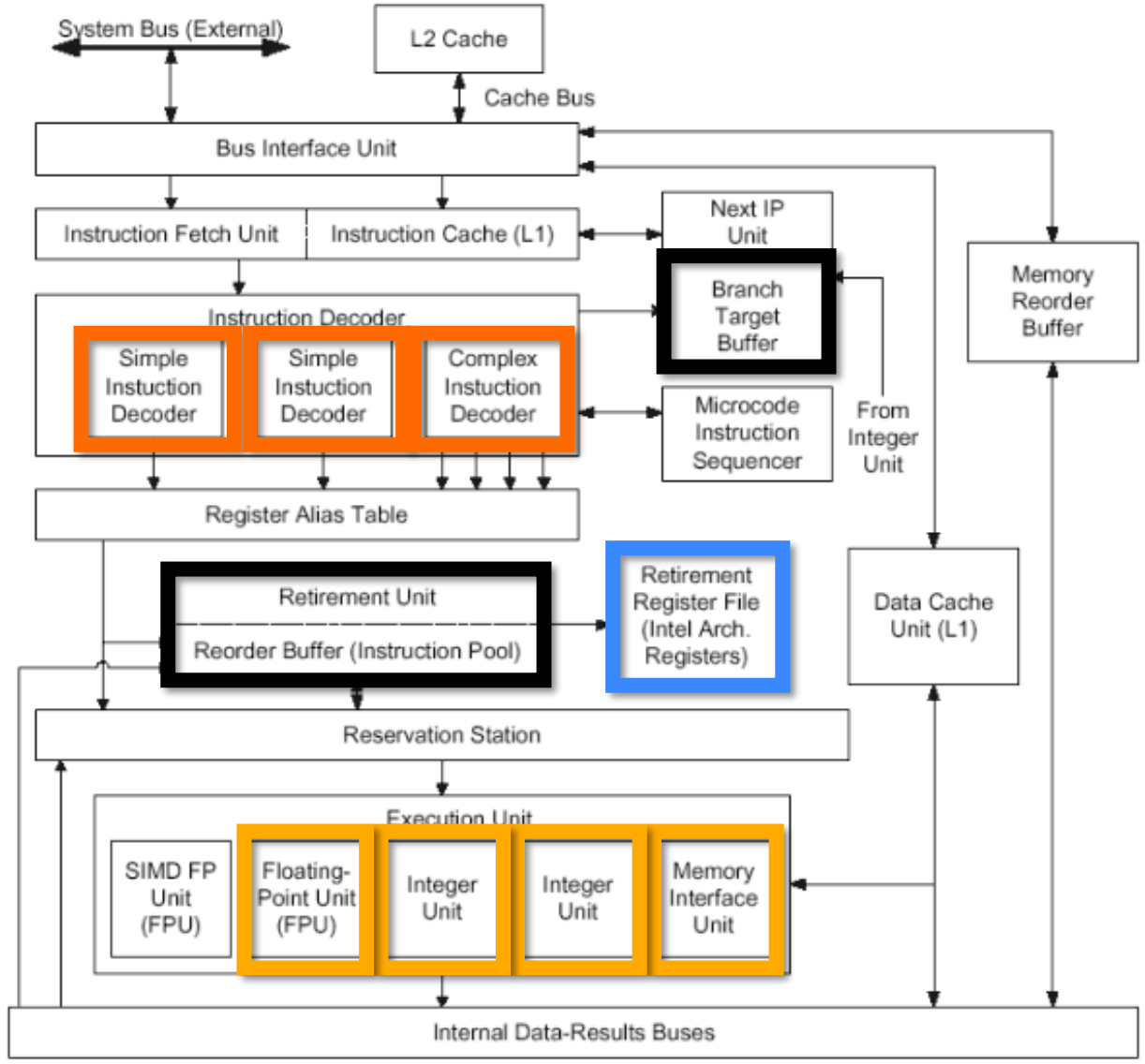
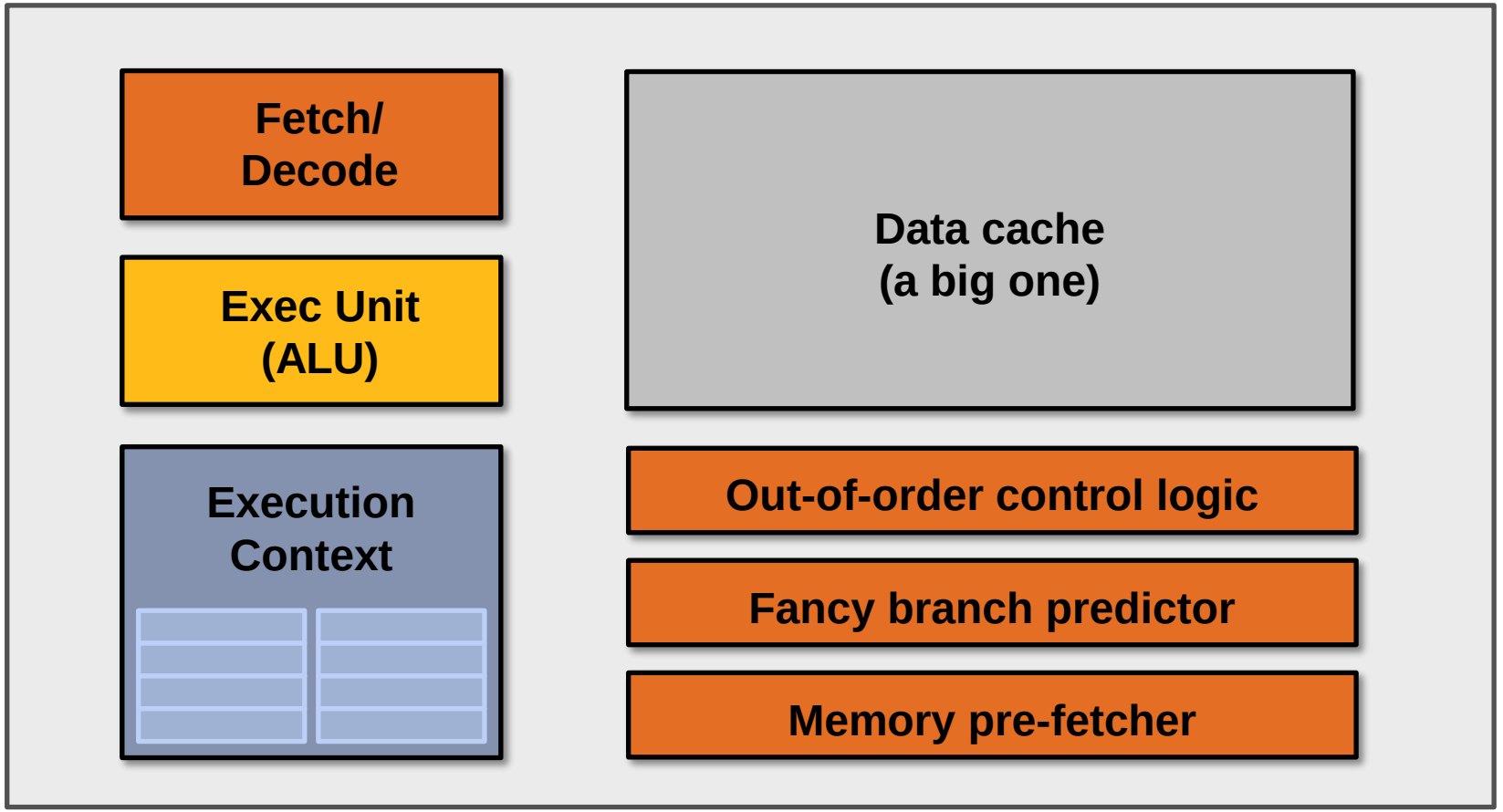


Image credit: http://ixbtlabs.com/articles/pentium4/index.html

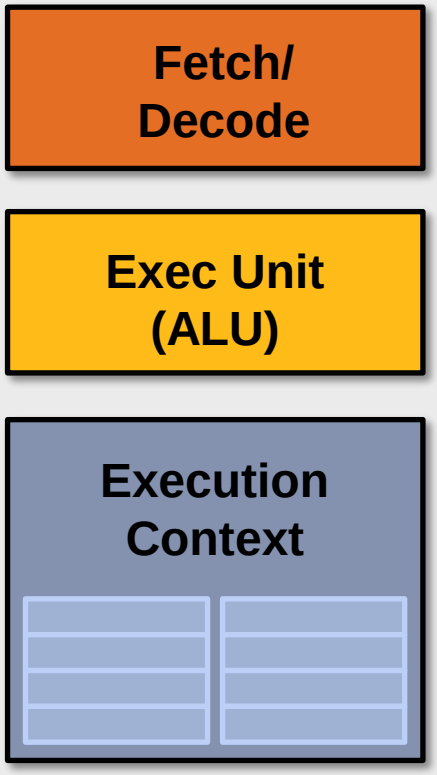
Processor: pre multi-core era

Majority of chip transistors used to perform operations that help a single 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)

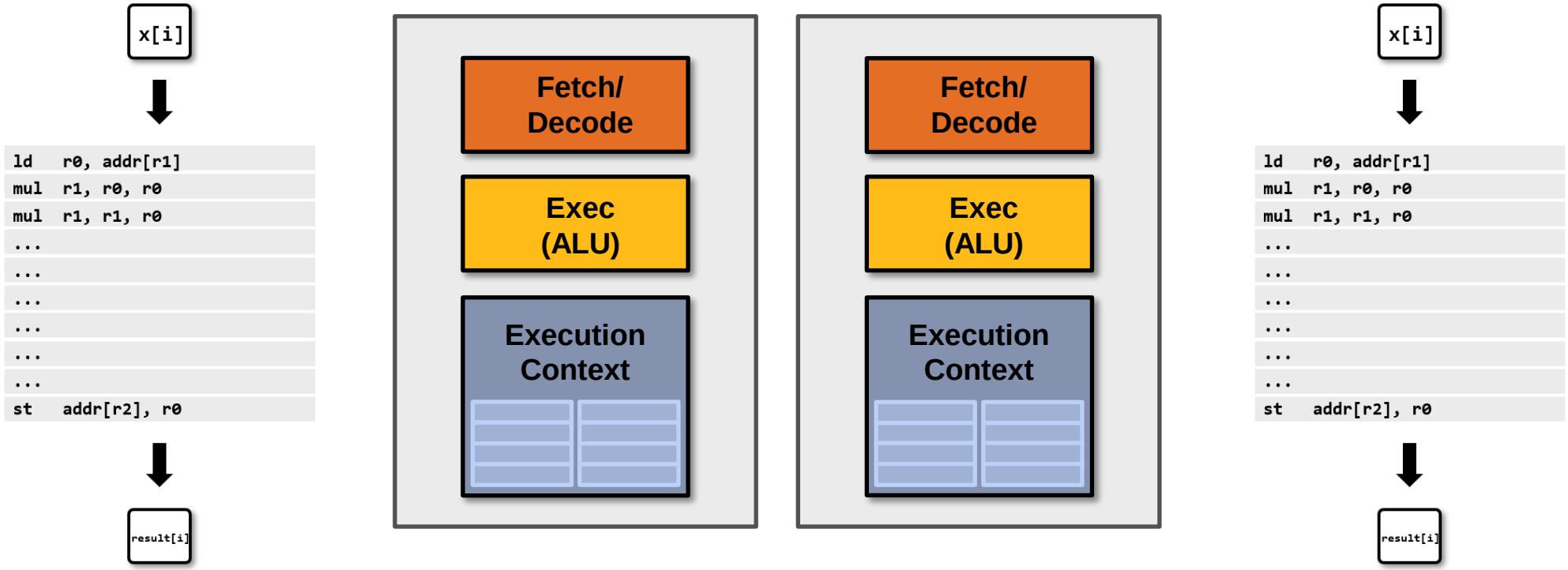


Idea #1:

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



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. :-(

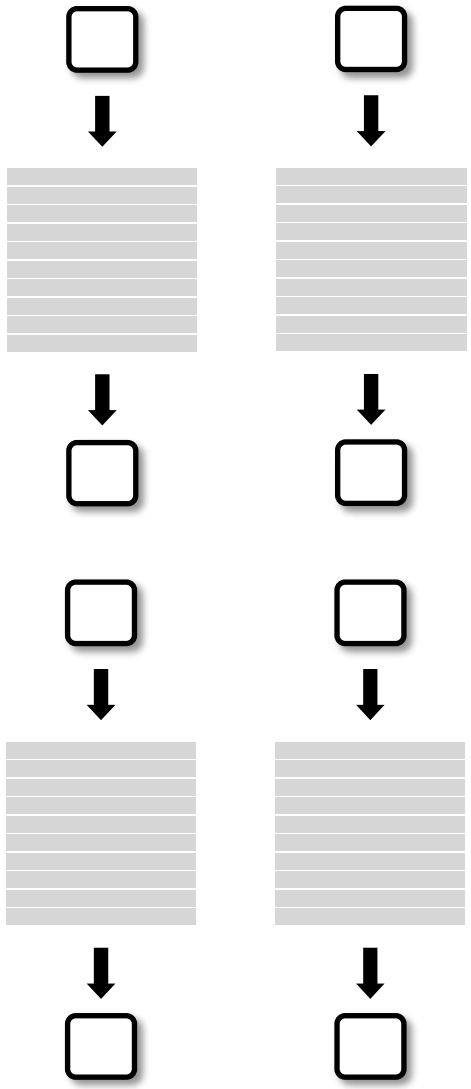
Using Cilk to provide parallelism

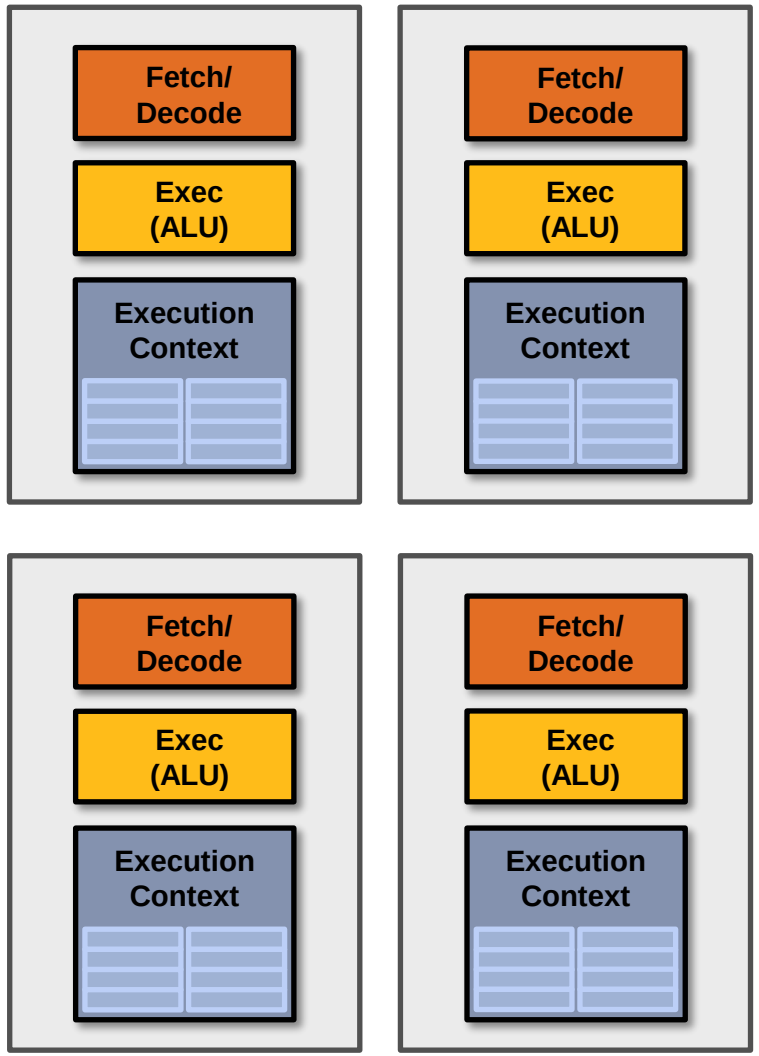
```
void sinx(int N, int terms, float* x, float* result)
{
   cilk_for (int i=0; i<N; i++)
   {
         float value = x[i];
         float numer = x[i] * x[i] * x[i];
         int denom = 6;  // 3!
         int sign = -1;

         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

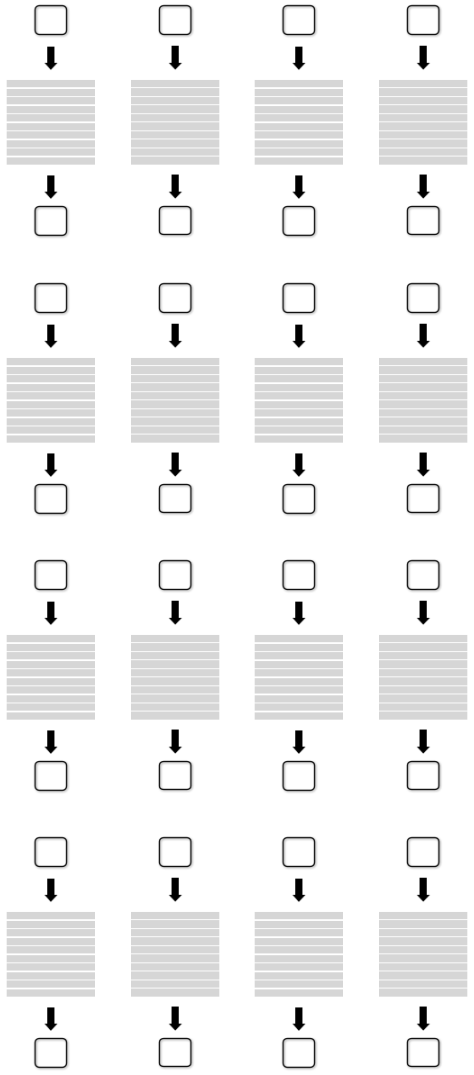
With this information, you could imagine how a compiler might automatically generate parallel threaded code

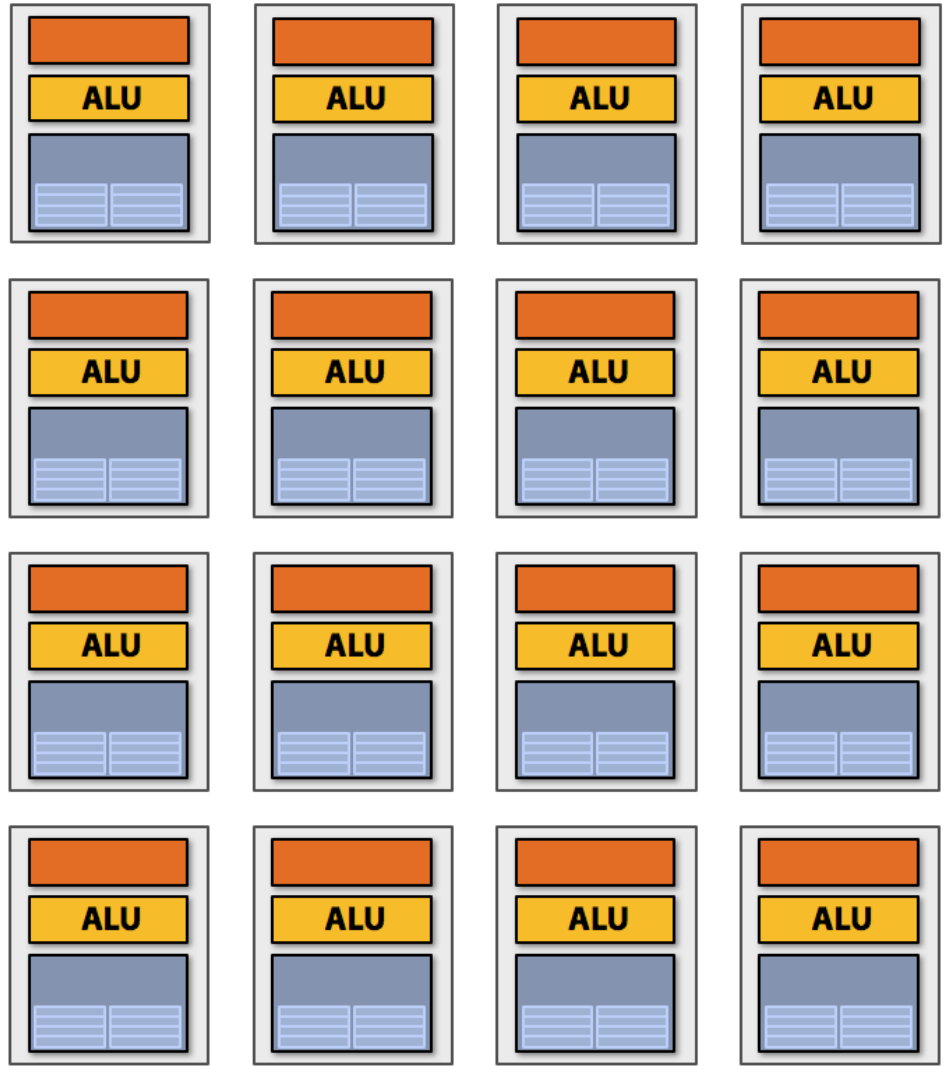
Four cores: compute four elements in parallel





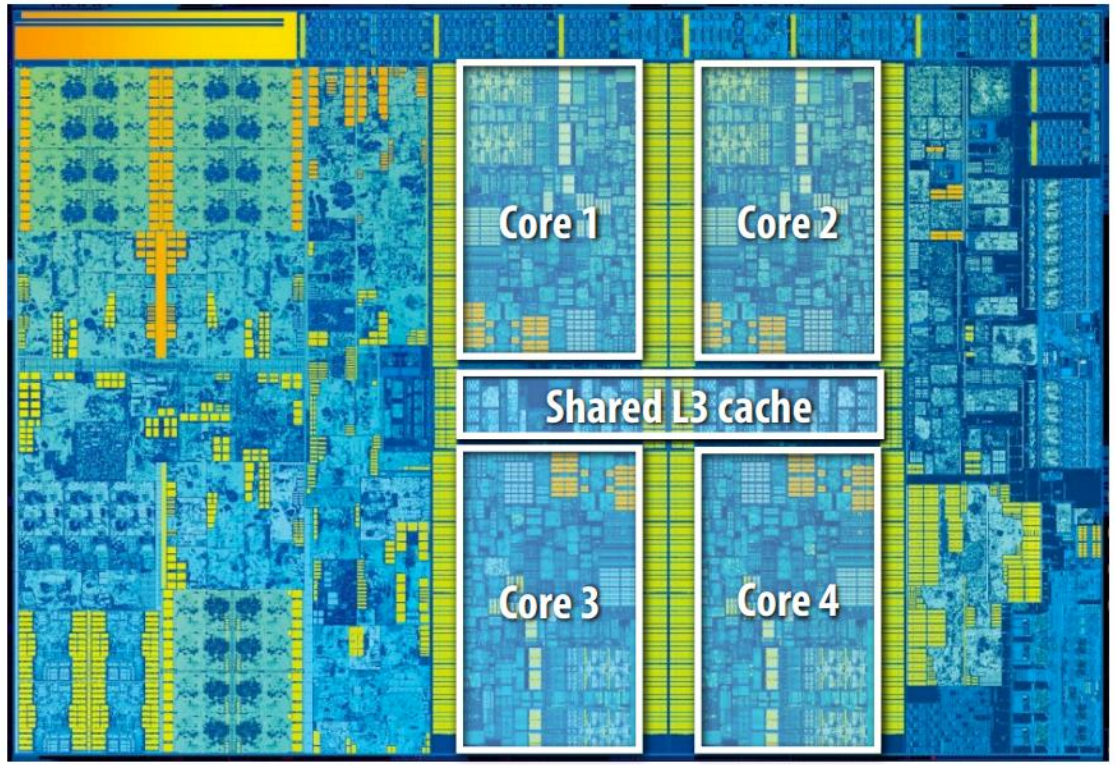
Sixteen cores: compute sixteen elements in parallel



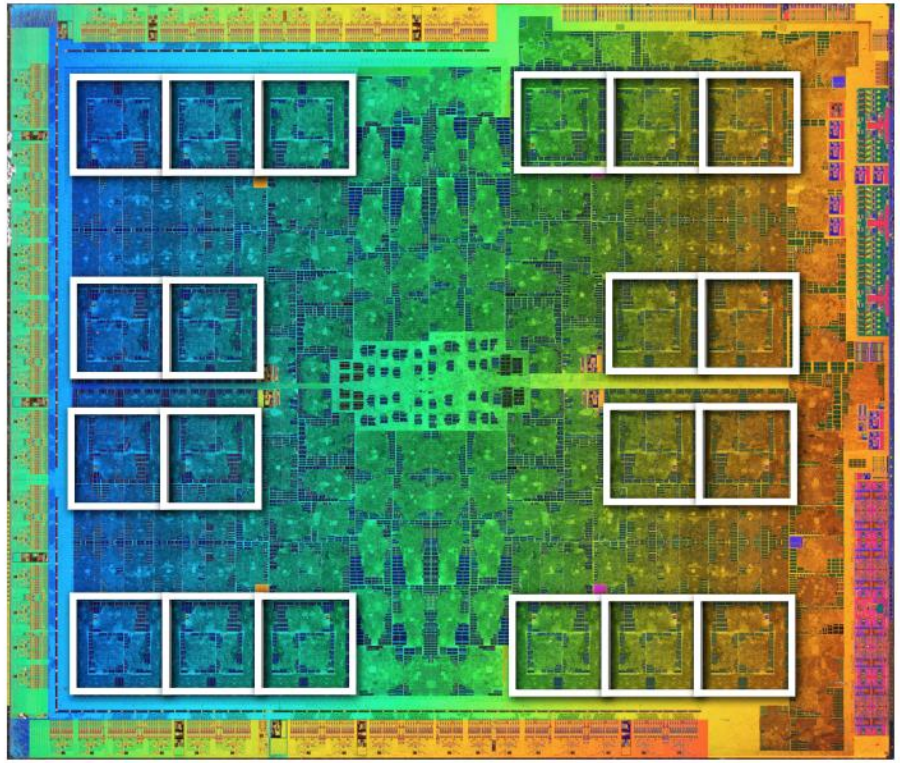


Sixteen cores, sixteen simultaneous instruction streams

Multi-core examples

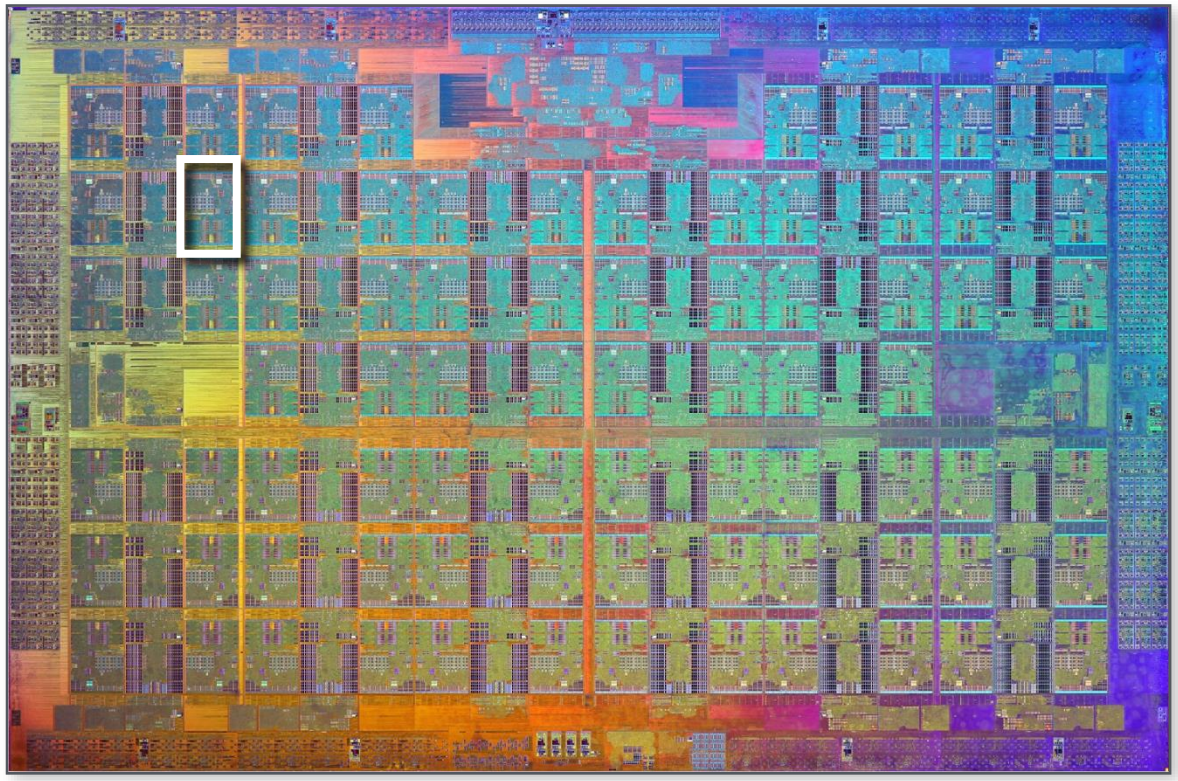


Intel “Skylake” Core i7 quad-core CPU (2015)

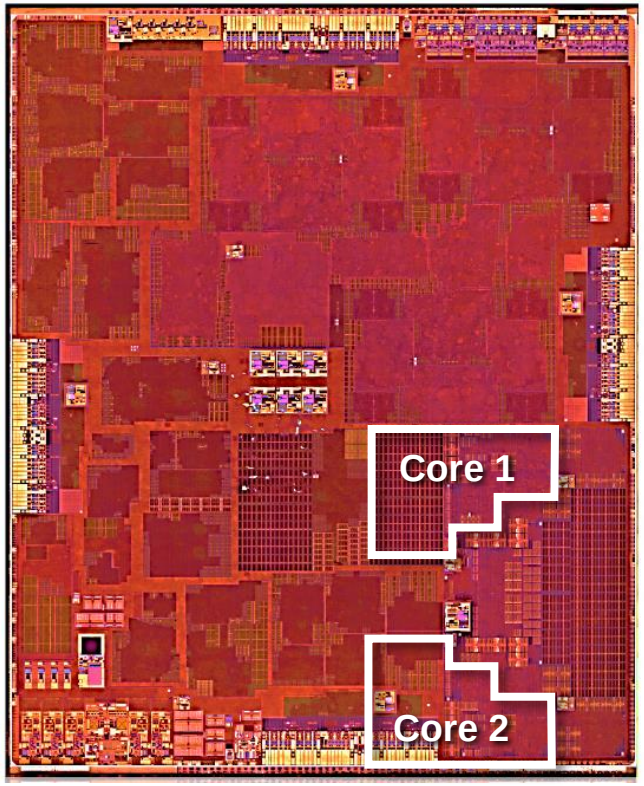


NVIDIA GP104 (GTX 1080) GPU 20 replicated (“SM”) cores (2016)

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

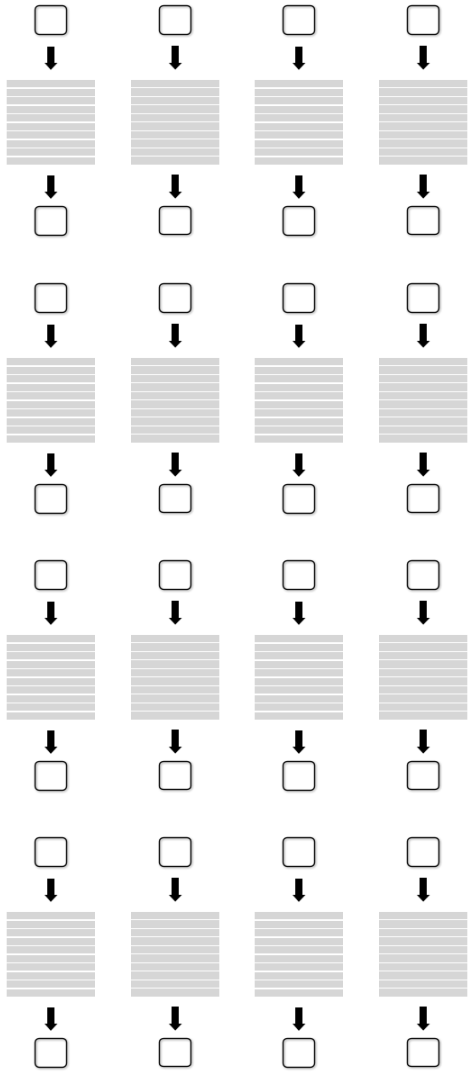
```
void sinx(int N, int terms, float* x, float* result)
{
   cilk_for (int i=0; i<N; i++)
   {
         float value = x[i];
         float numer = x[i] * x[i] * x[i];
         int denom = 6;  // 3!
         int sign = -1;

         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;
   }
}
```

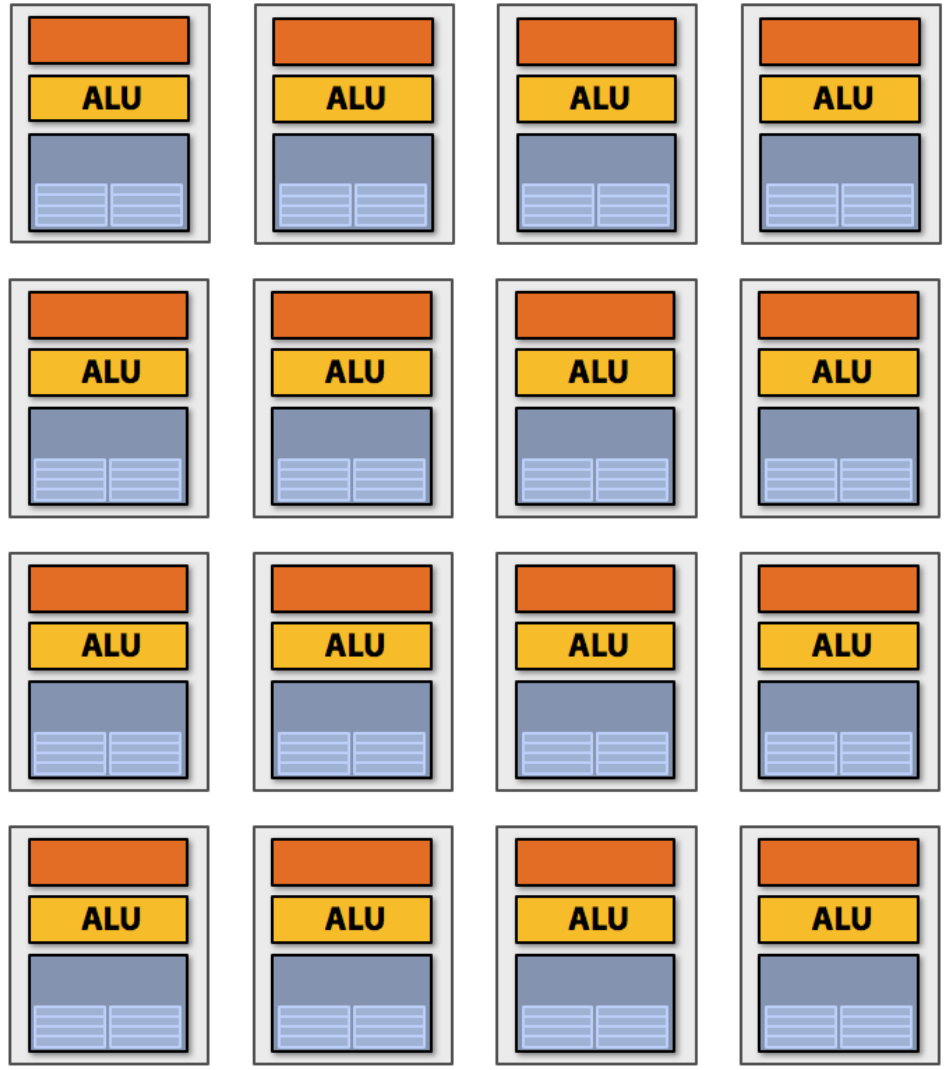
Another interesting property of this code:

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)

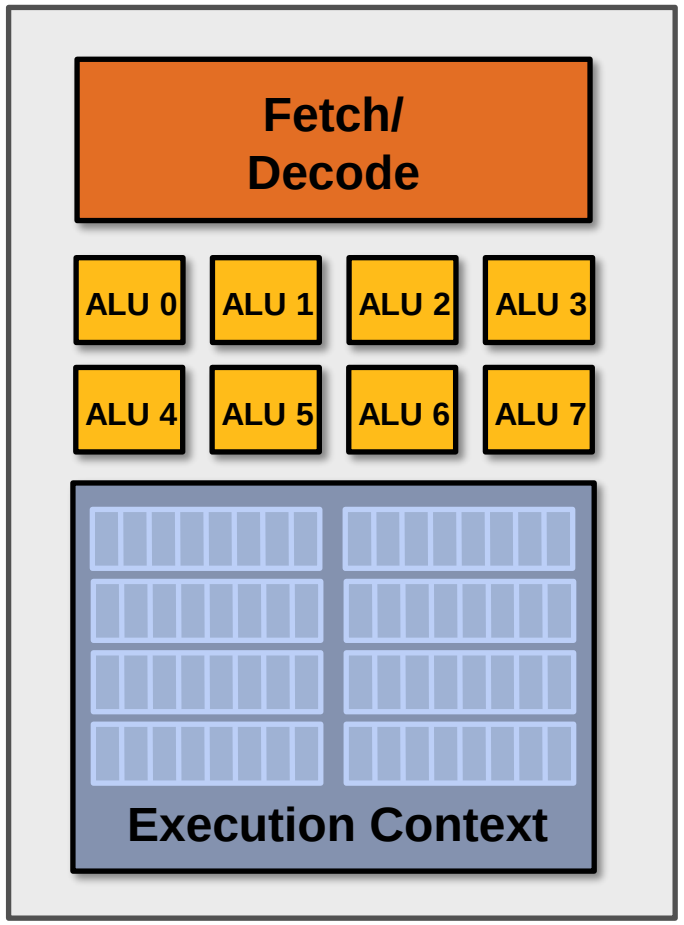
Sixteen cores: compute sixteen elements in parallel





Sixteen cores, sixteen simultaneous instruction streams

Add ALUs to increase compute capability



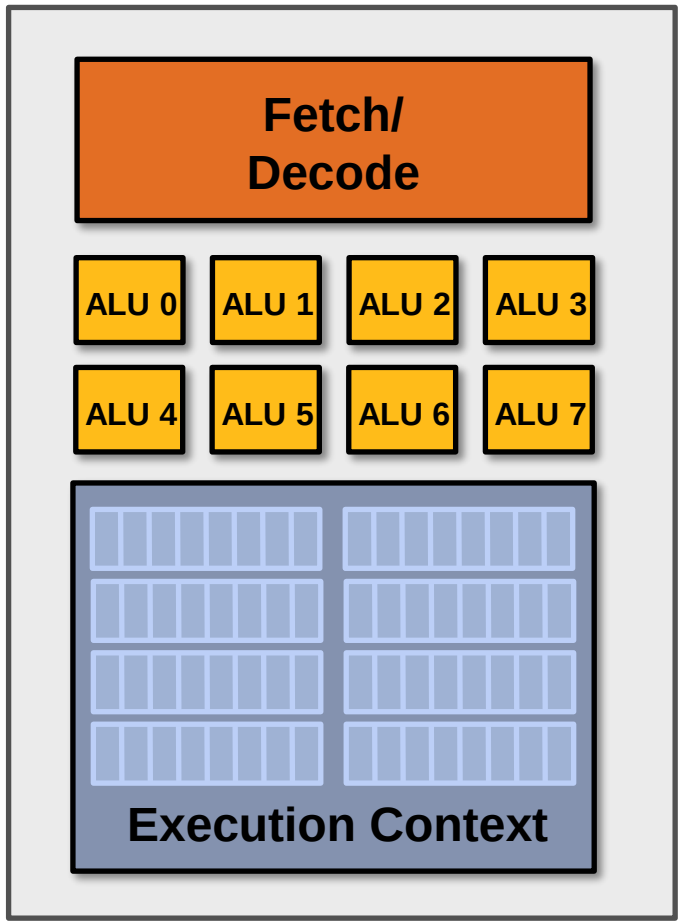
Idea #2:
Amortize cost/complexity of managing an instruction stream across many ALUs

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)

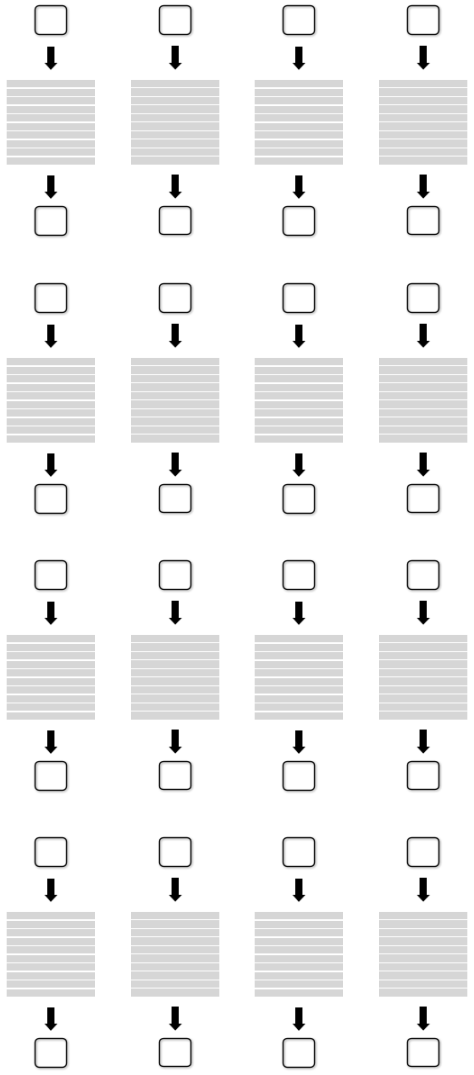
```
#include <immintrin.h>

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);
   }
}
```

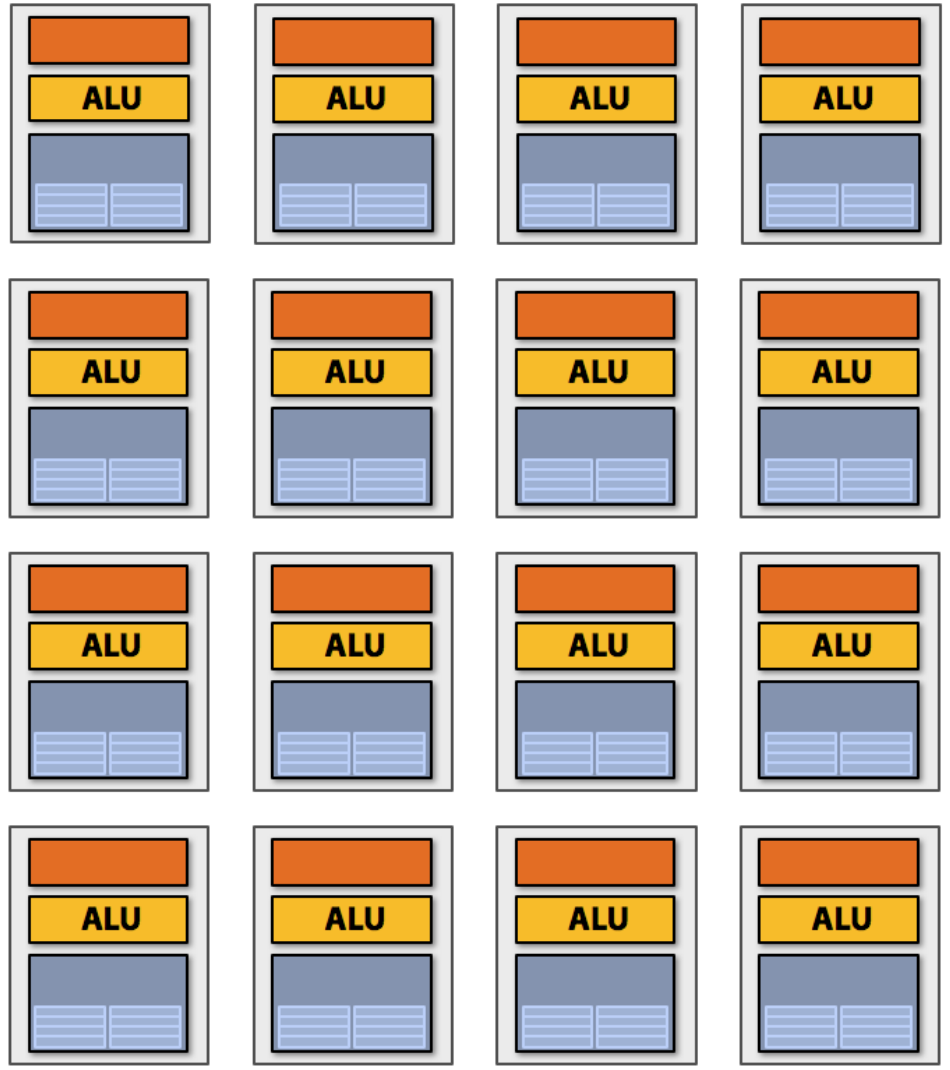
```
vloadps  xmm0, addr[r1]
vmulps   xmm1, xmm0, xmm0
vmulps   xmm1, xmm1, xmm0
...
...
...
...
...
...
vstoreps  addr[xmm2], xmm0
```

Compiled program:

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

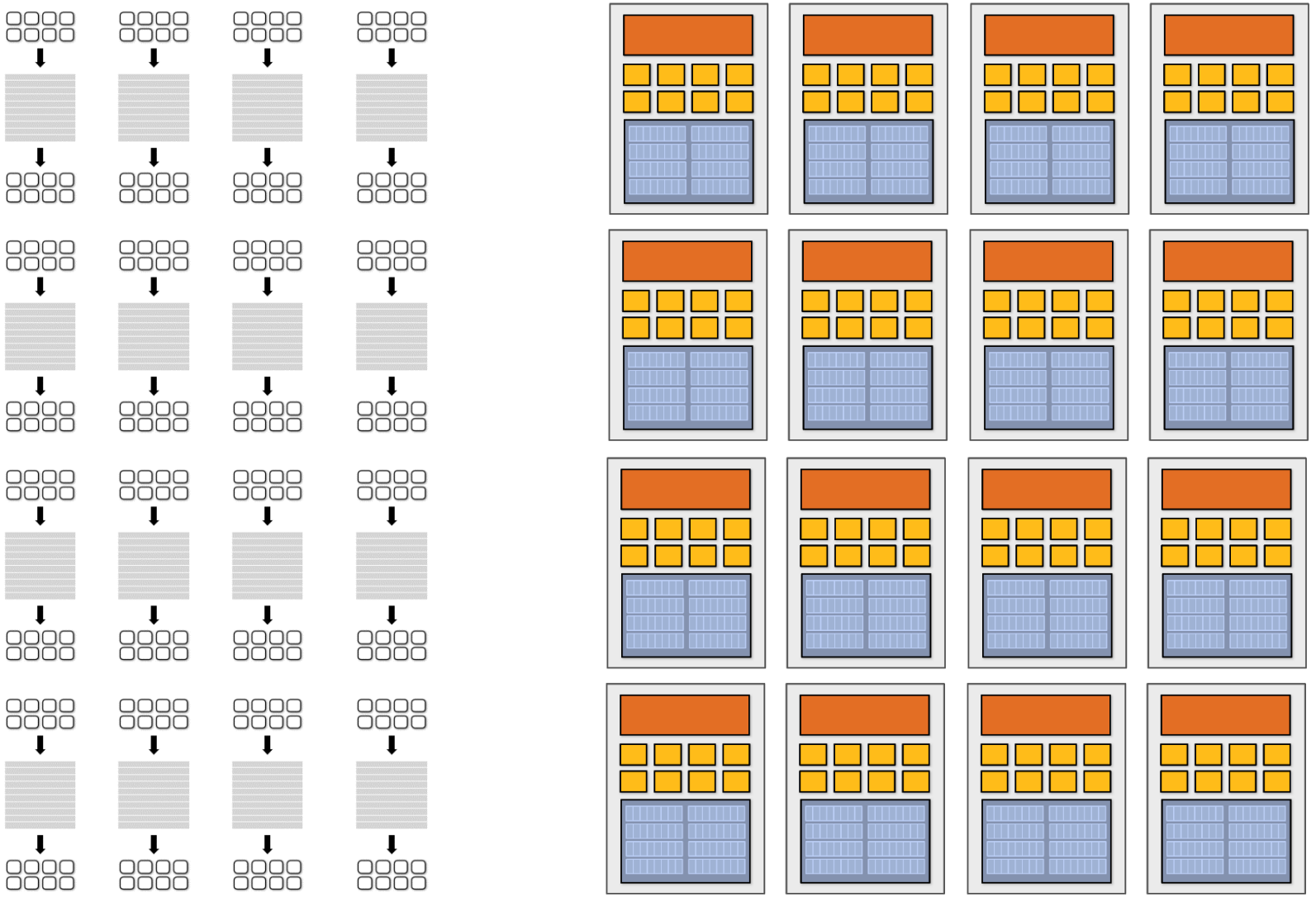
Sixteen cores: compute sixteen elements in parallel





Sixteen cores, sixteen simultaneous instruction streams

16 SIMD cores: 128 elements in parallel



16 cores, 128 ALUs, 16 simultaneous instruction streams

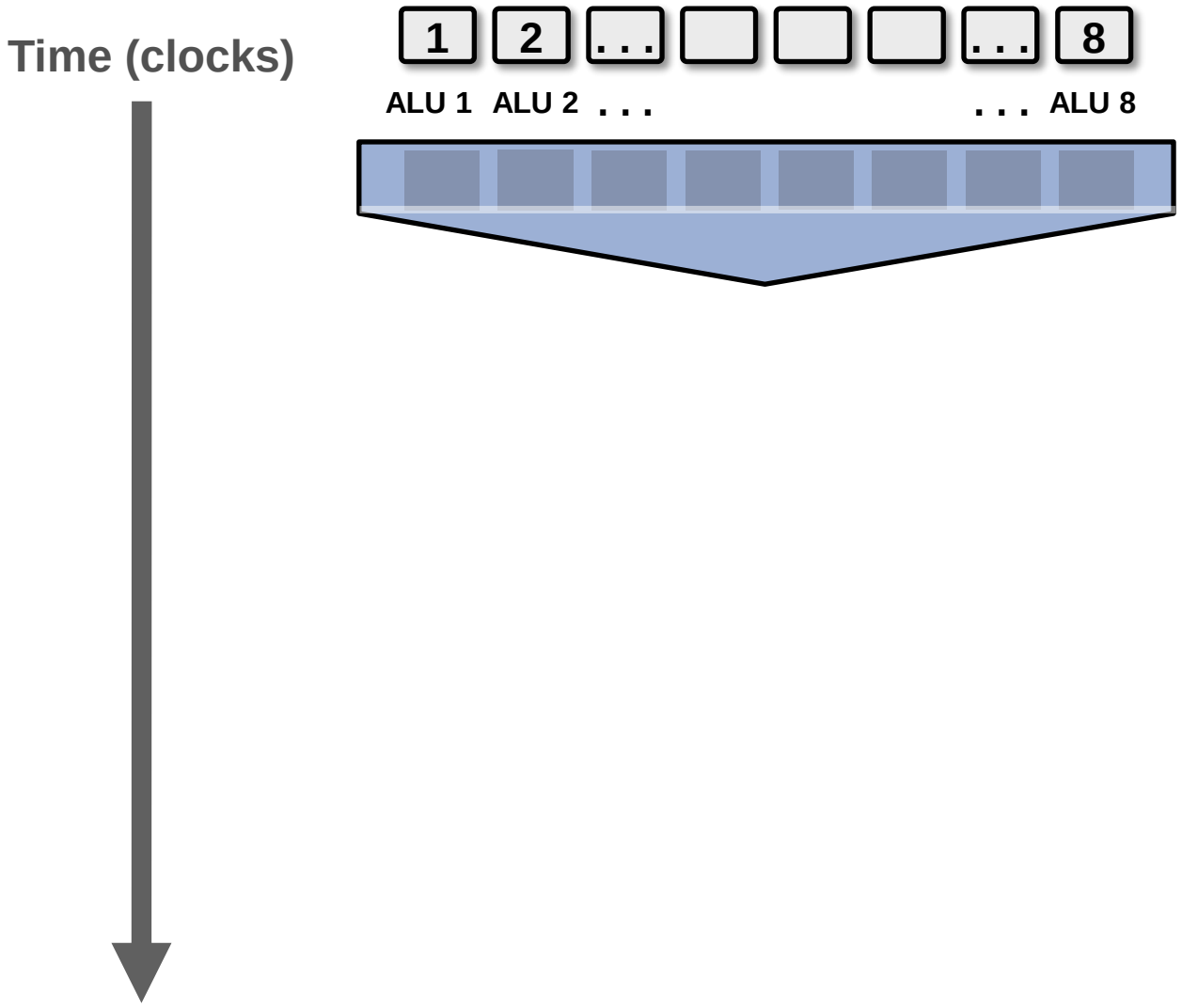
Data-parallel expression

```
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.

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

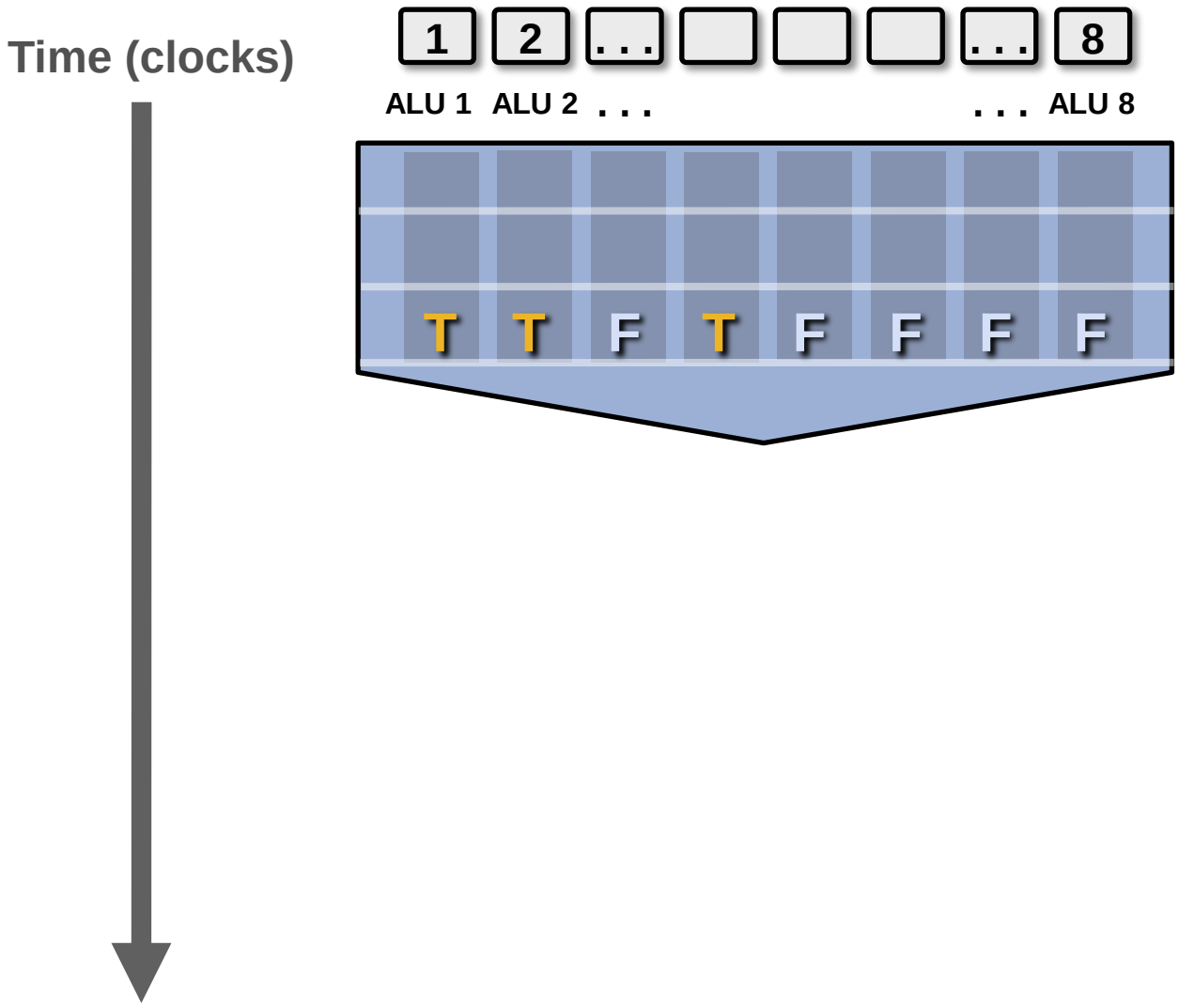
What about conditional execution?



(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;
```

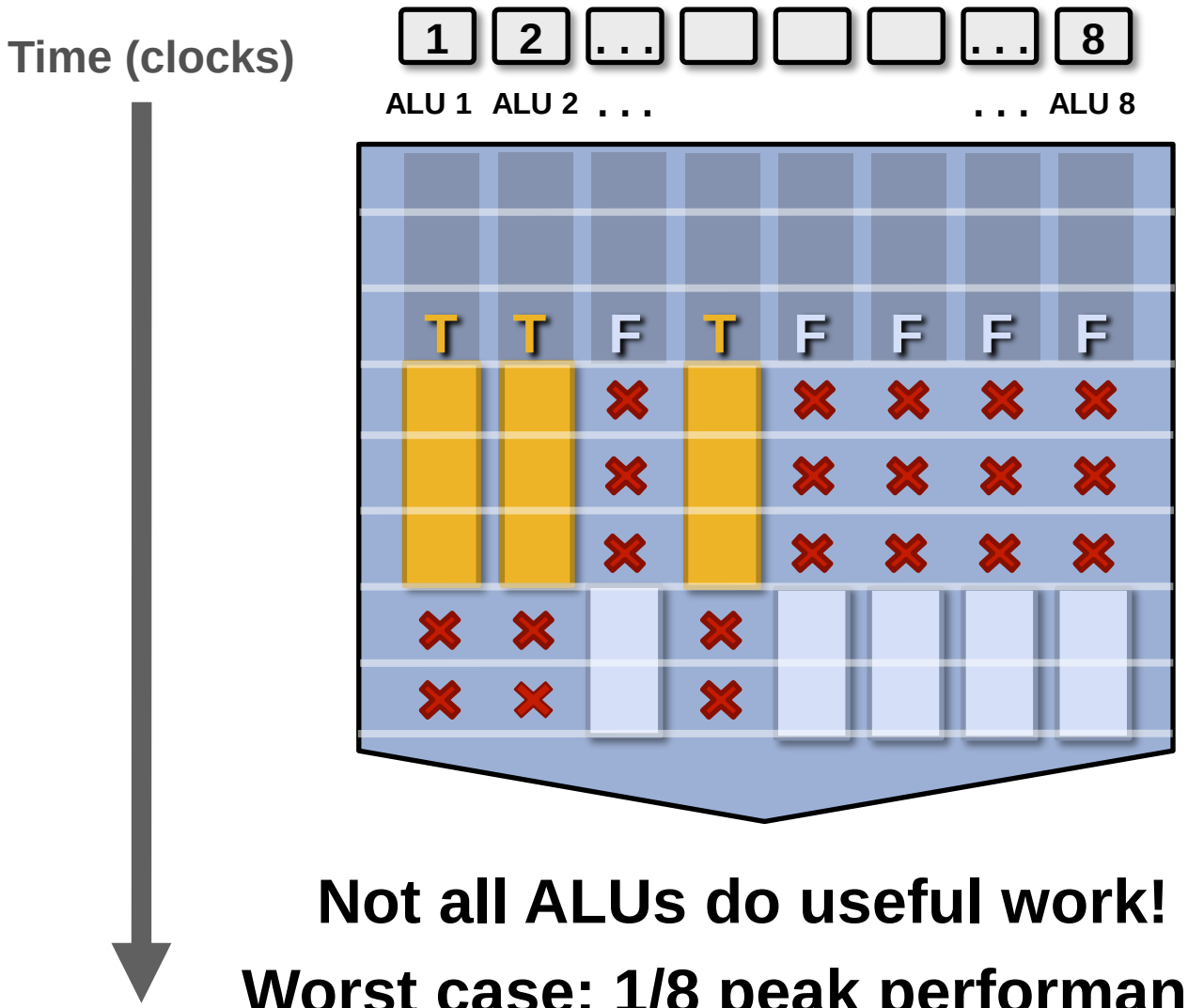
What about conditional execution?



(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



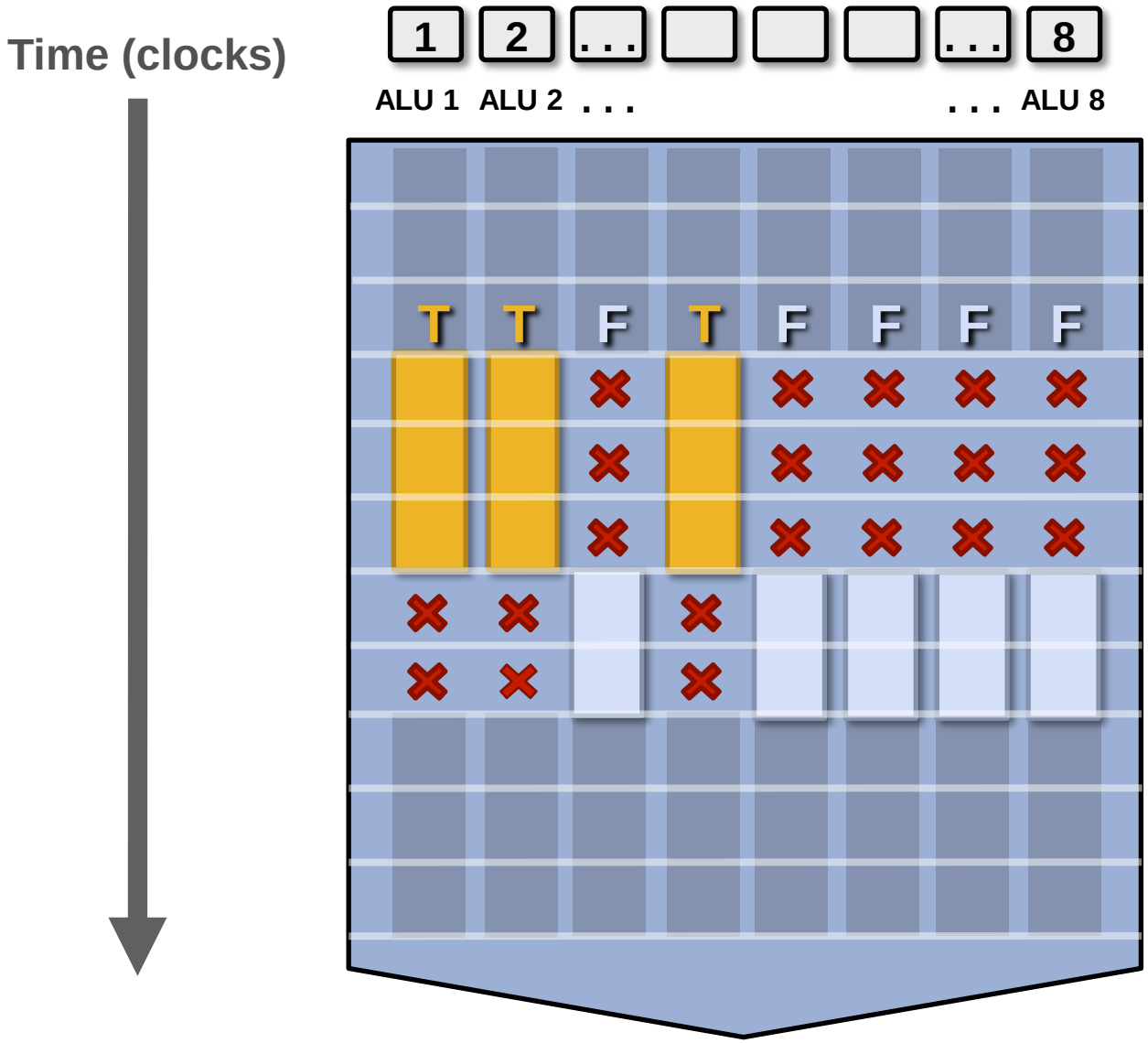
Not all ALUs do useful work!

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;
```

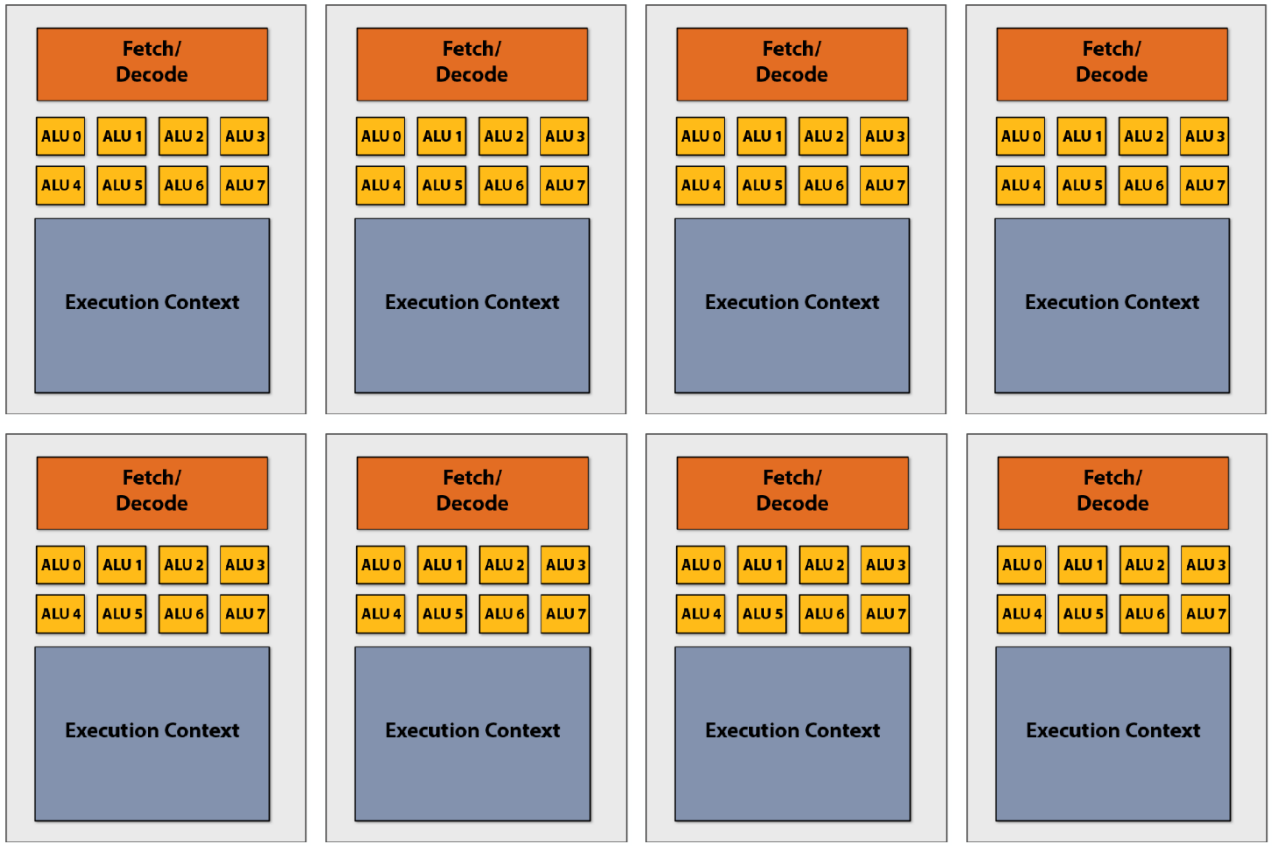
After branch: continue at full performance



(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

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**
  - **Superscalar: exploit ILP within an instruction stream. Process different instructions from the same instruction stream in parallel (within a core)**
    - Parallelism automatically and dynamically discovered by the hardware during execution (not programmer visible)
  - **Multi-core: use multiple processing cores**
    - Provides thread-level parallelism: simultaneously execute a completely different instruction stream on each core
    - Programmers/algorithms decide when to do so (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 usually declared by programmer, but can be inferred by loop analysis by advanced compiler

Next lecture

- **Continuing on computer architecture**

- **Memory access:**
  - Multi-threading (hyper-threading)
  - Caching