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.

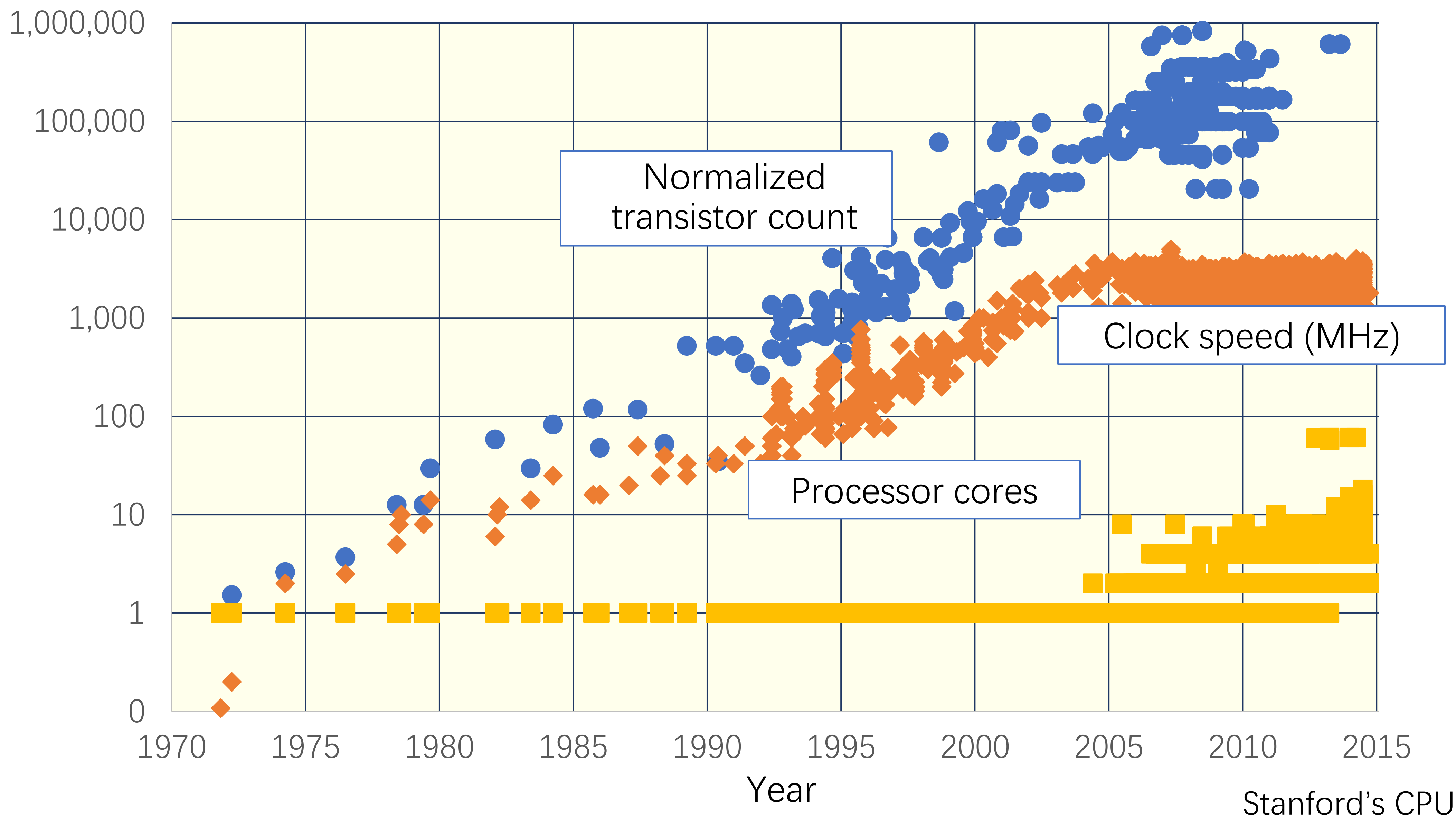
Lecture Overview

- **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**

- **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…

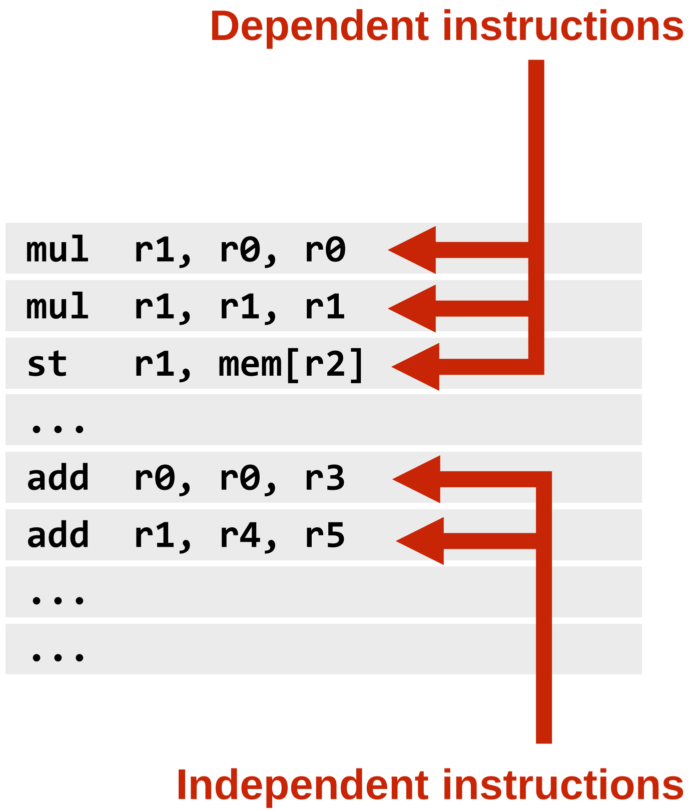
PC

```
_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 <u>appear</u> to be executed in program order. BUT <u>independent</u> instructions can be executed simultaneously by a processor without impacting program correctness
  - <u>Superscalar execution</u>: processor dynamically finds independent instructions in an instruction sequence and executes them in parallel



Dependent instructions

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

Independent instructions

ILP example

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

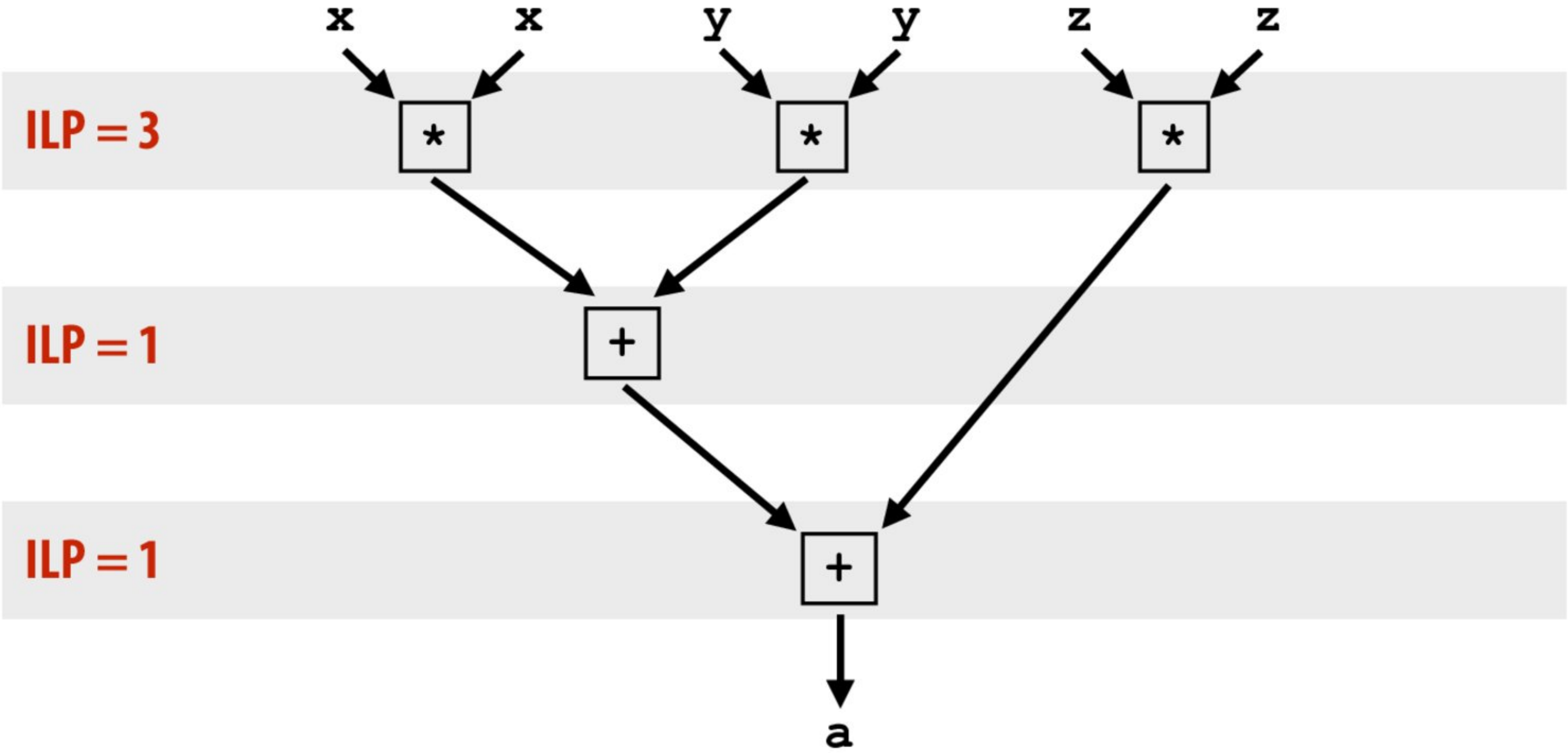
Consider the following program:

```
// 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

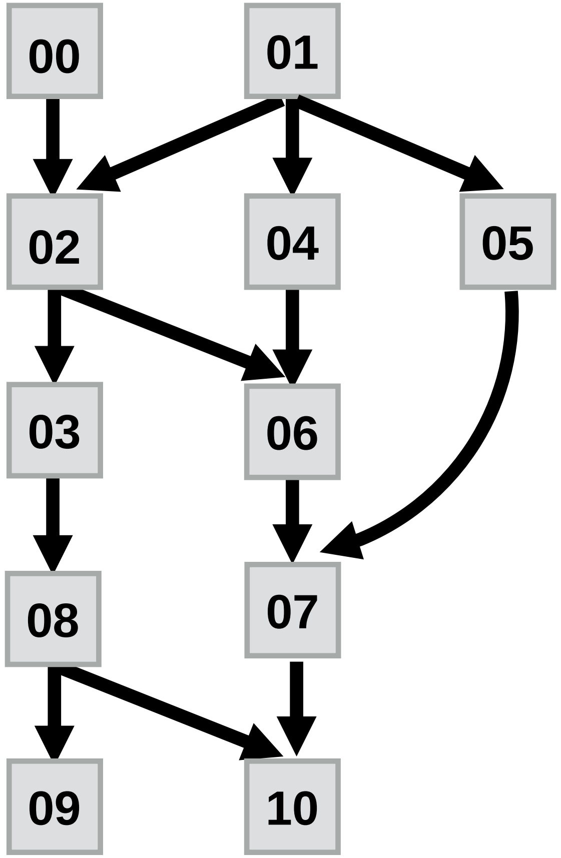
A more complex example

Program (sequence of instructions)

value during execution

```
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
```

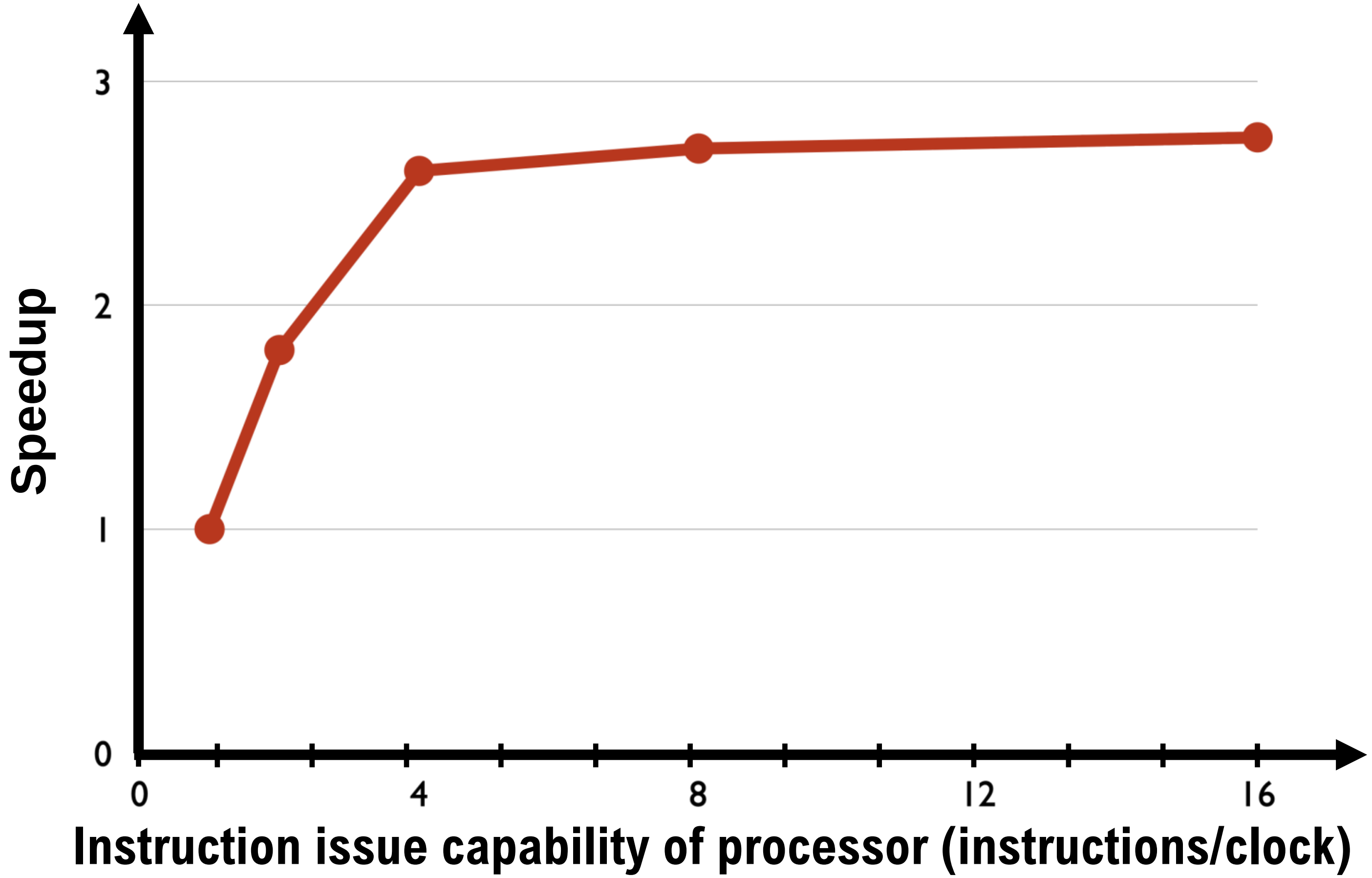
Instruction dependency graph



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**
- **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;
   }
}
```

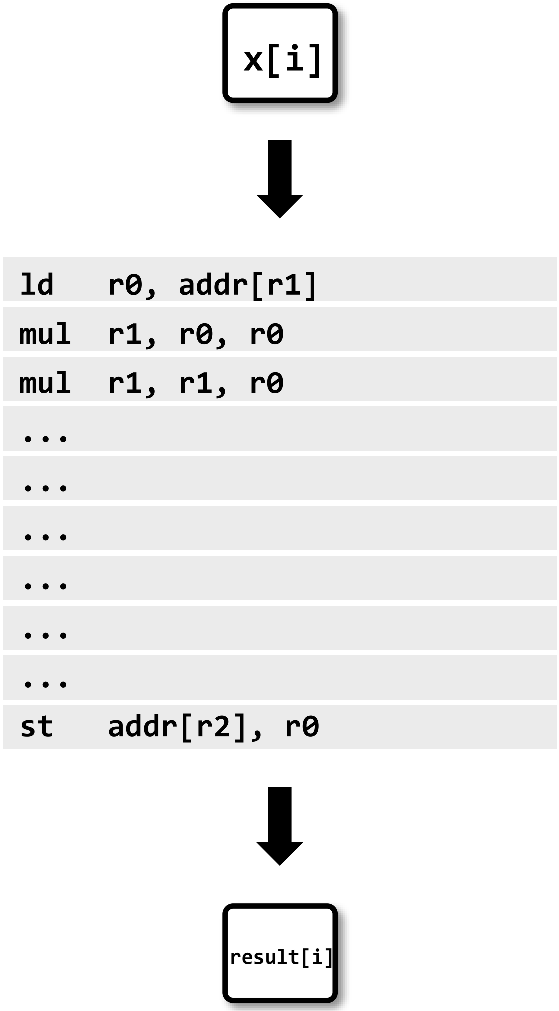
Compute sin(x) **using Taylor expansion:**

sin(x) = x - x^3/3! + x^5/5! - x^7/7! + ...

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

Compile program

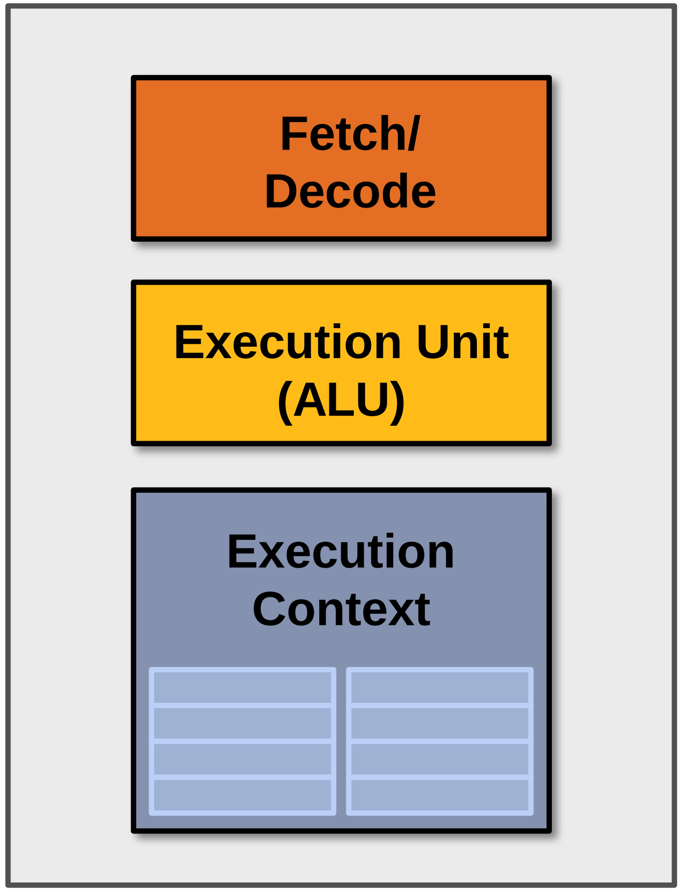
```
void sinx(int N, int terms, float* x, float* result)
{
   for (int i=0; i<N; i++)
   {
         float value = x[i];
         float numer = x[i] * x[i] * x[i];
         int denom = 6;  // 3!
         int sign = -1;

         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;
   }
}
```

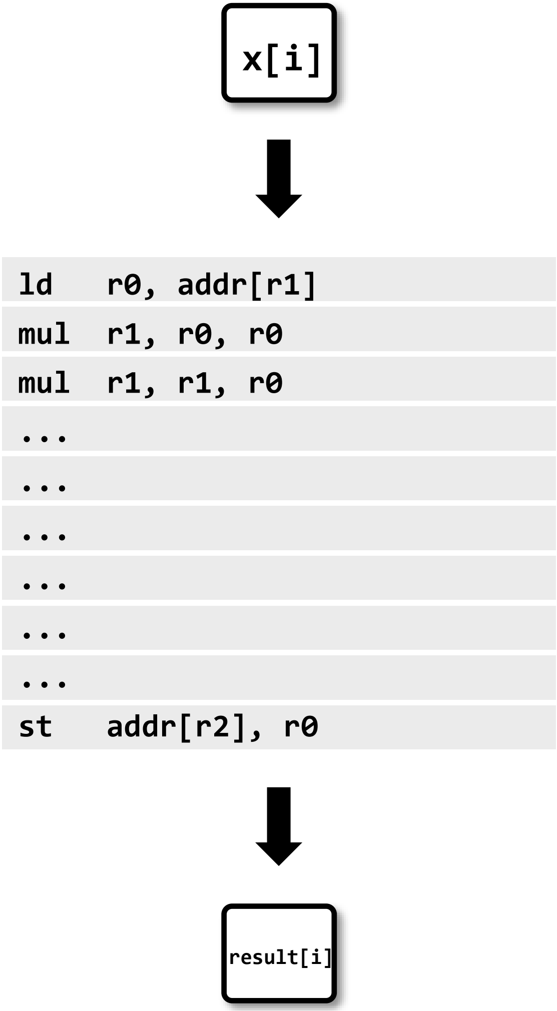


```
ld    r0, addr[r1]
mul   r1, r0, r0
mul   r1, r1, r0
...
...
...
...
...
...
st    addr[r2], r0
```

Execute program

My very simple processor: executes one instruction per clock

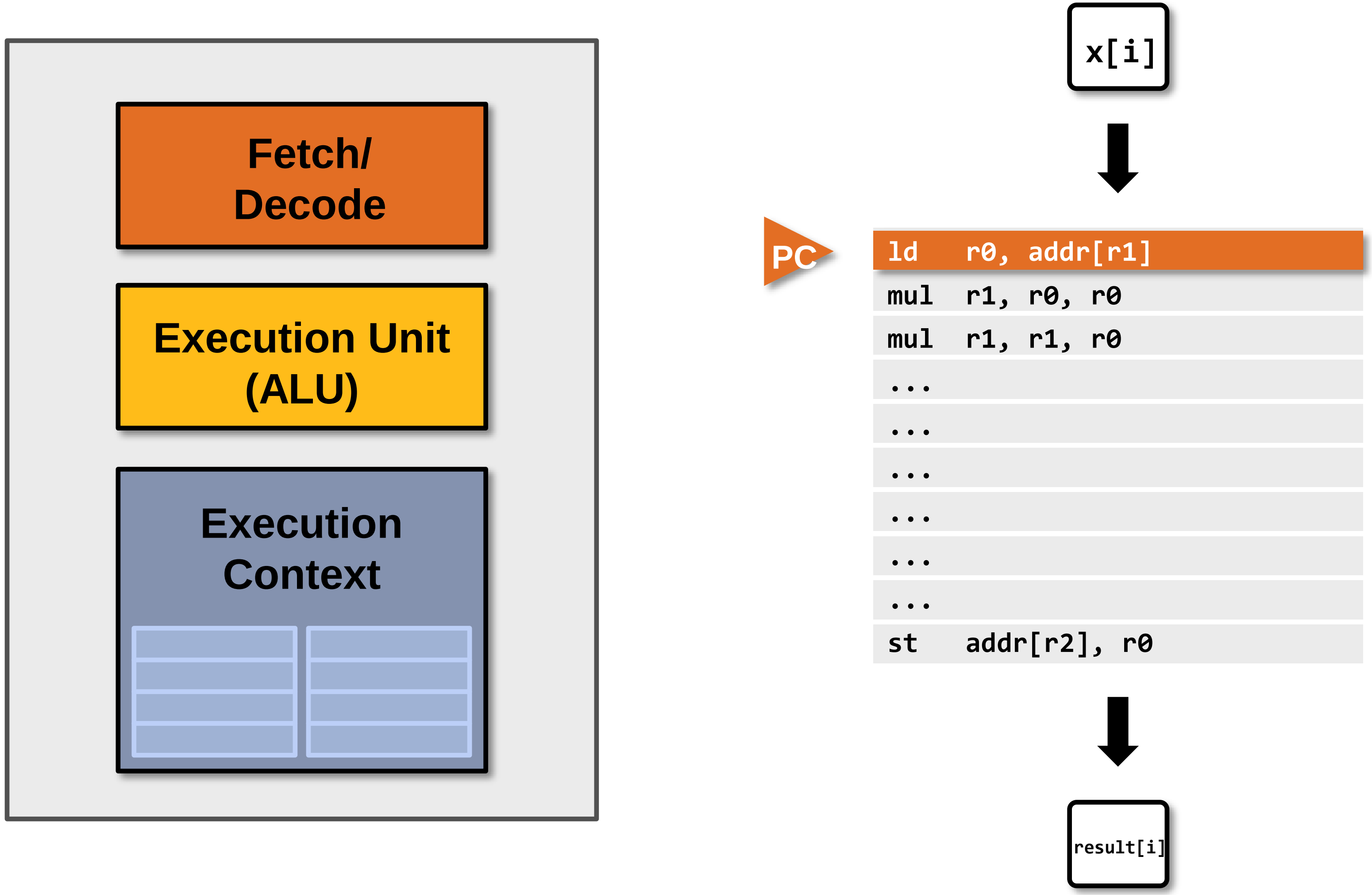




```
ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...
...
...
...
...
...
st   addr[r2], r0
```

Execute program

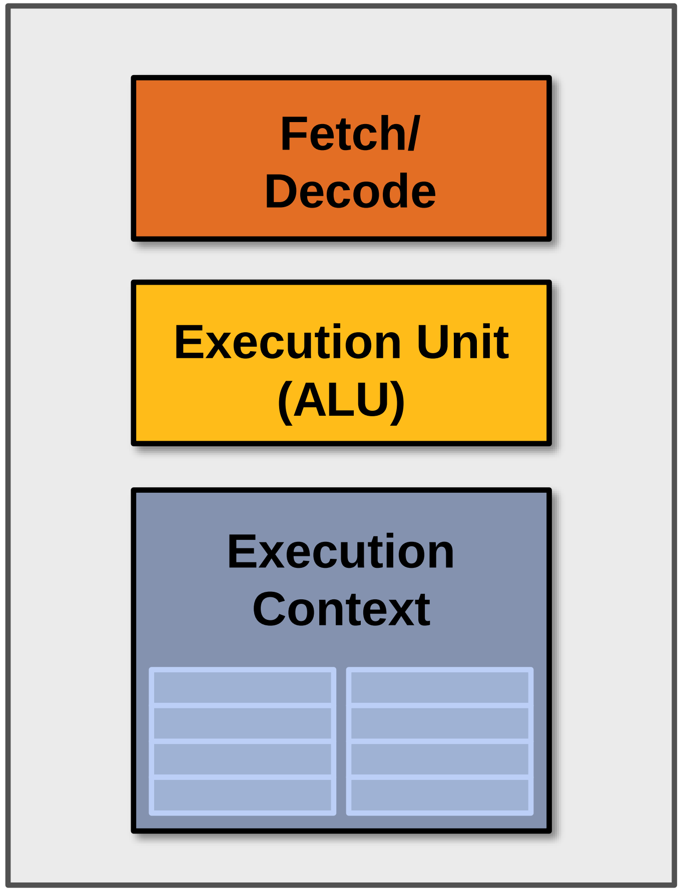
My very simple processor: executes one instruction per clock

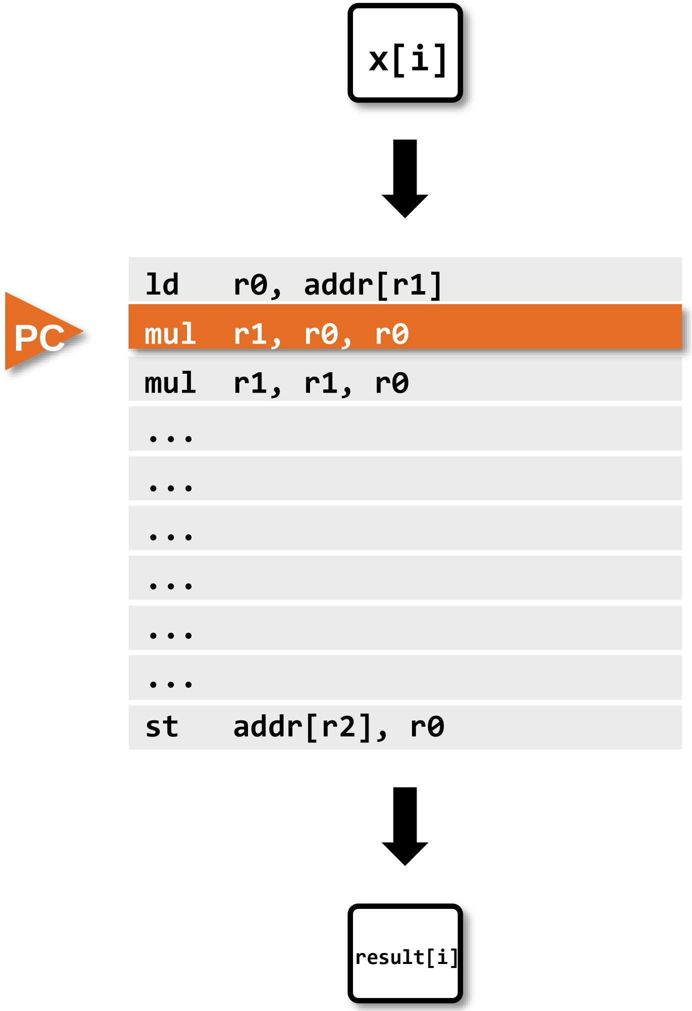


```
ld    r0, addr[r1]
mul   r1, r0, r0
mul   r1, r1, r0
...
...
...
...
...
...
st    addr[r2], r0
```

Execute program

My very simple processor: executes one instruction per clock

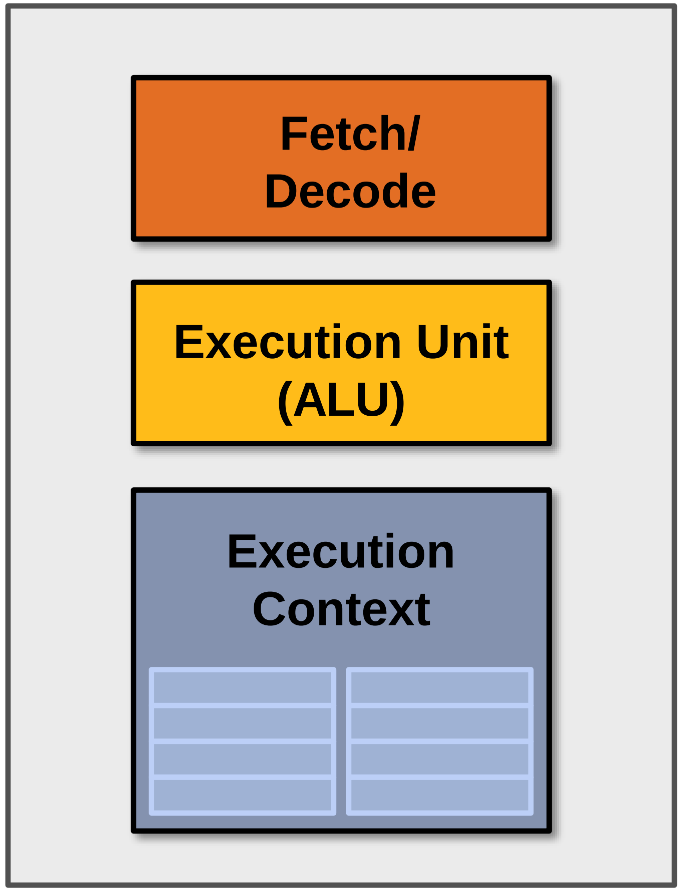


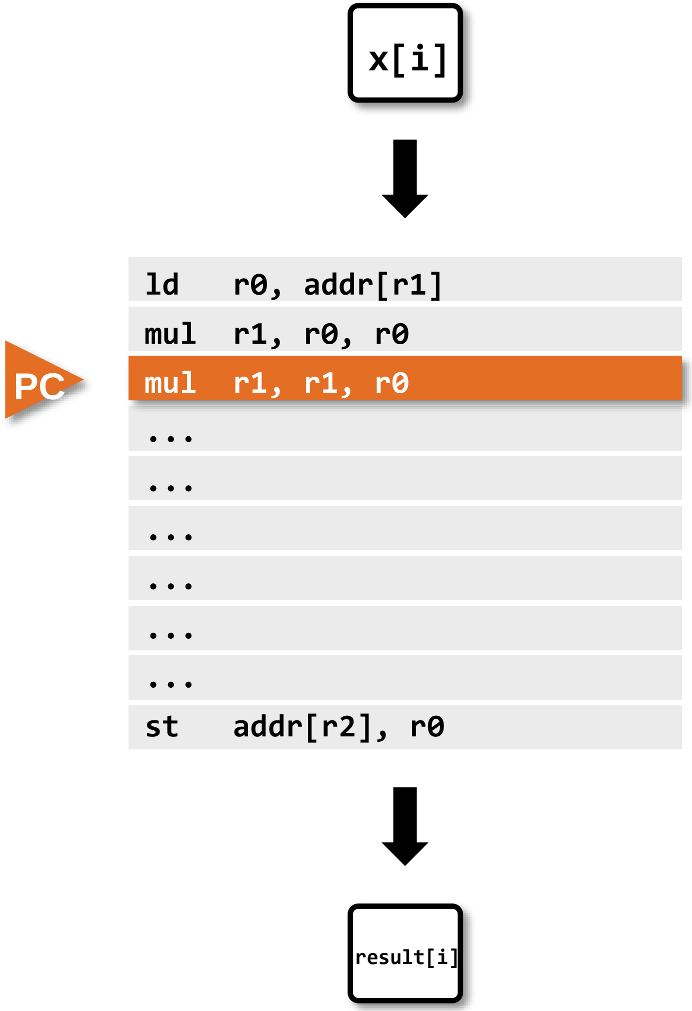


```
ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...
...
...
...
...
...
st   addr[r2], r0
```

Execute program

My very simple processor: executes one instruction per clock



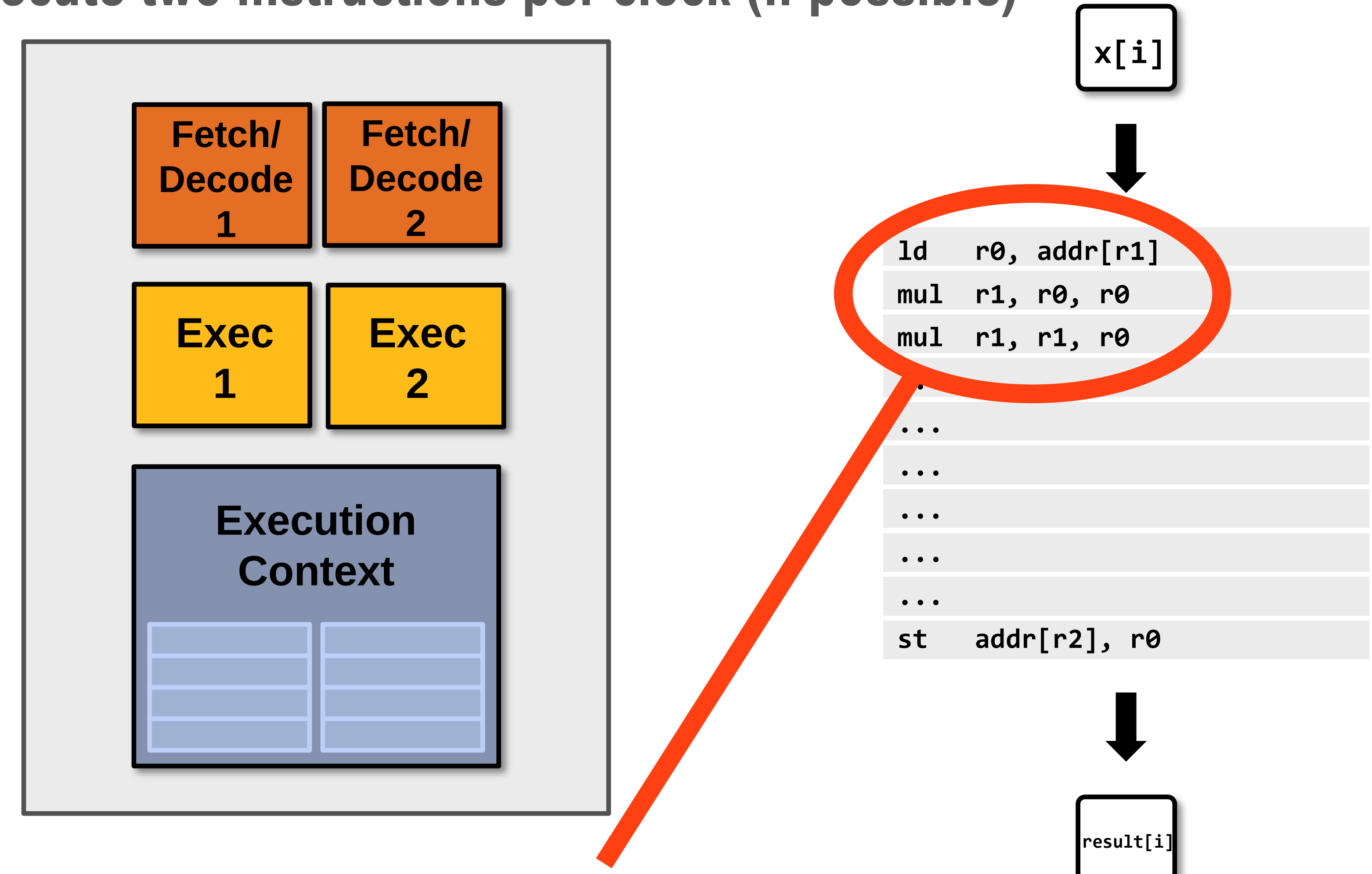


```
ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...
...
...
...
...
...
st   addr[r2], r0
```

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

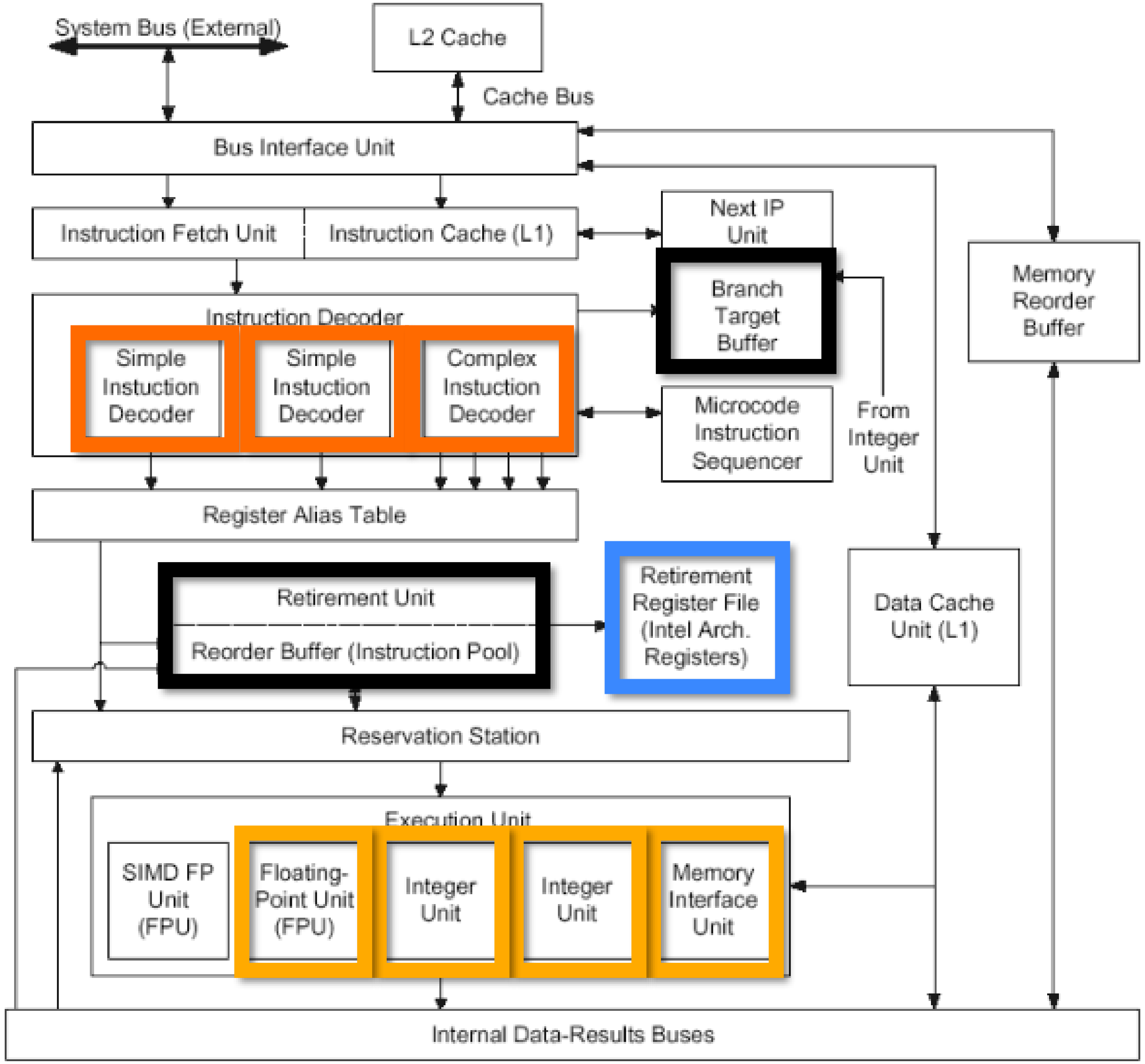
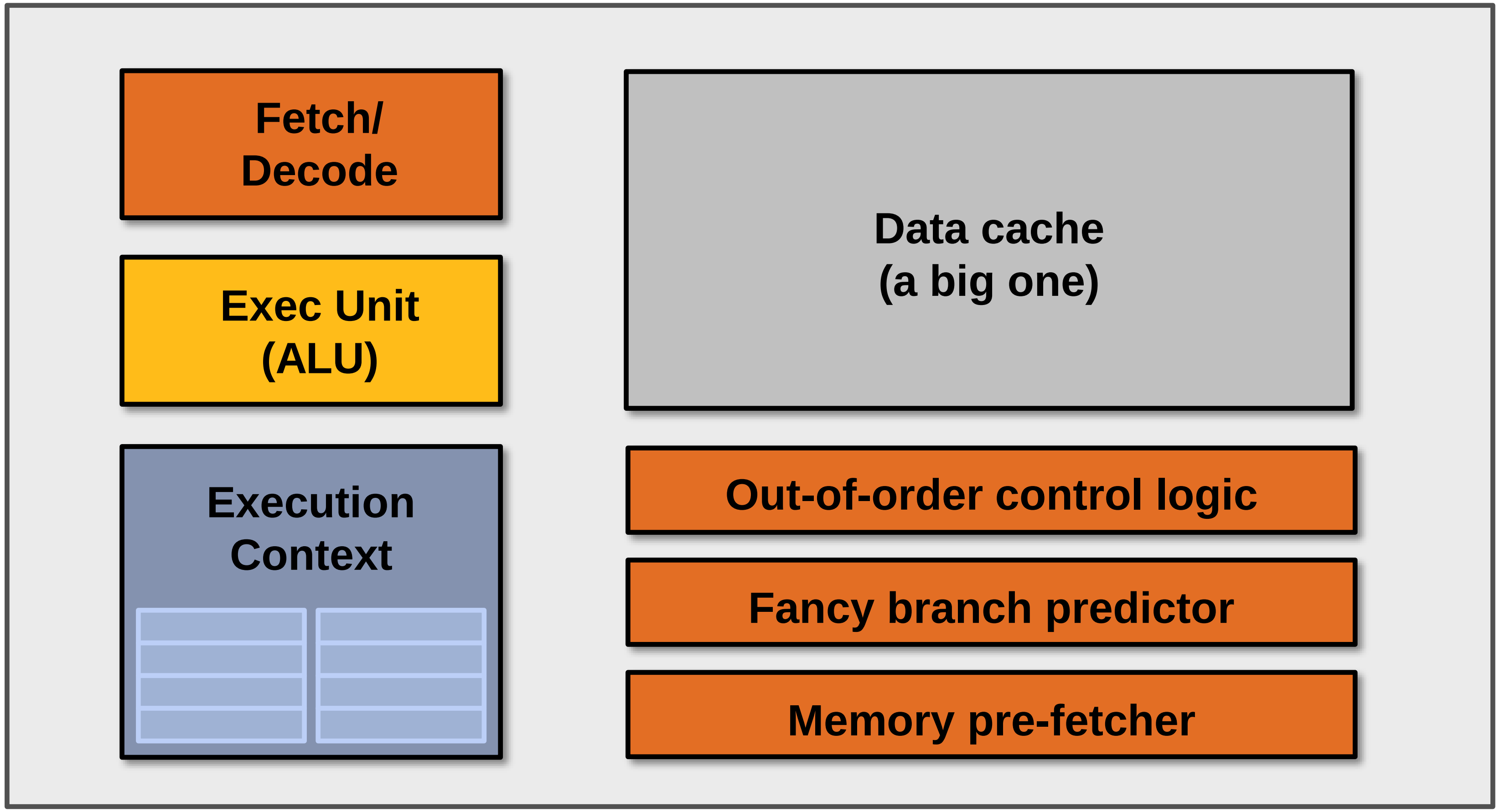


Image credit: http://ixbtlabs.com/articles/pentium4/index.html

Processor: pre multi-core era

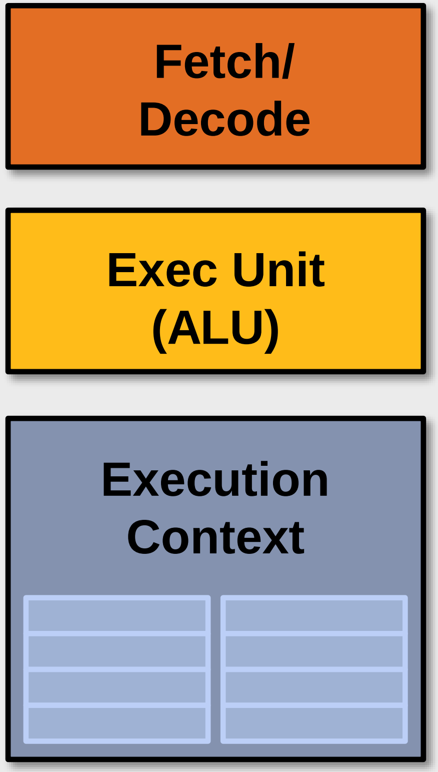
Majority of chip transistors used to perform operations that help a <u>single</u> instruction stream run fast



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)

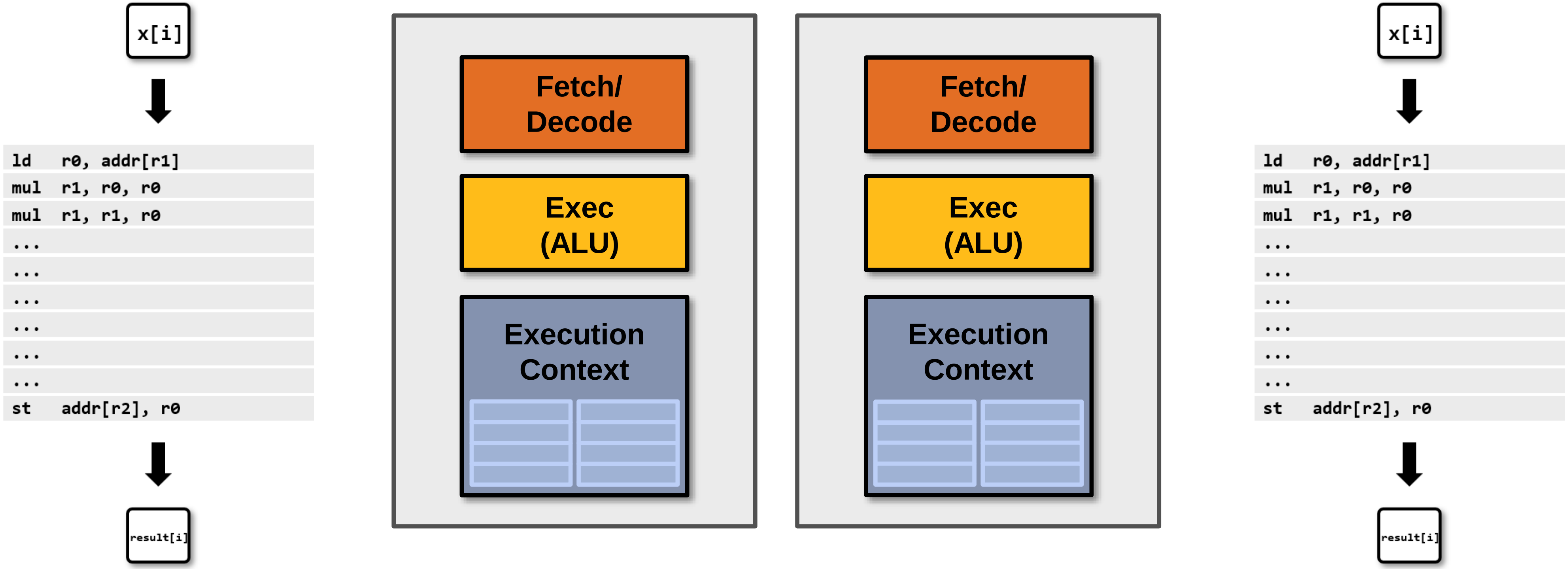


Idea #1:

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. :-(

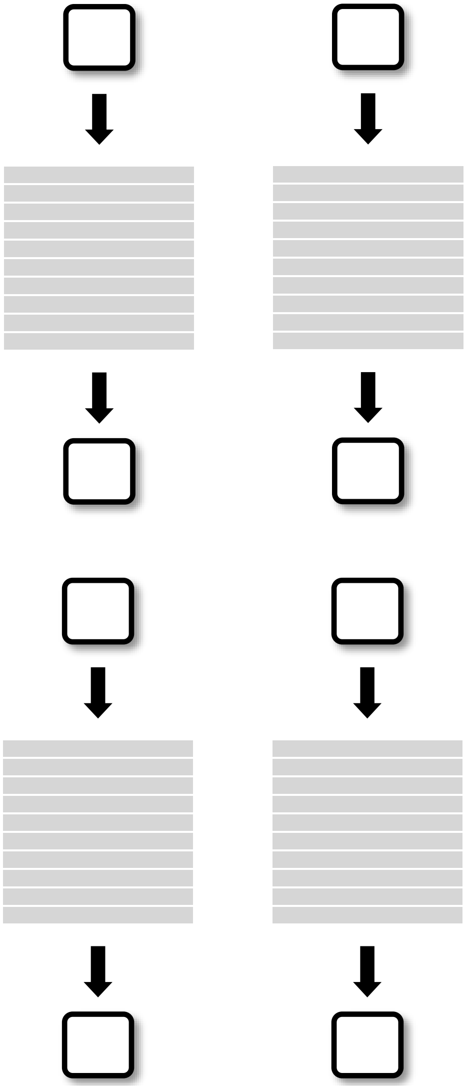
Using Cilk to provide parallelism

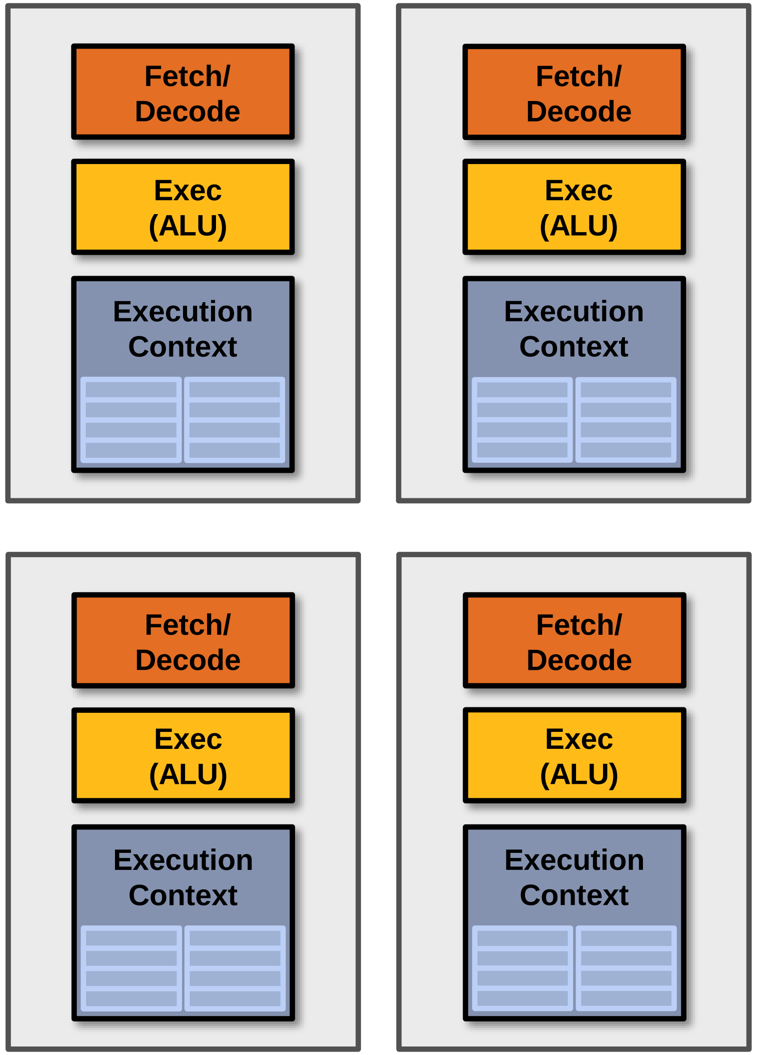
```
void sinx(int N, int terms, float* x, float* result)
{
   cilk_for (int i=0; i<N; i++)
   {
         float value = x[i];
         float numer = x[i] * x[i] * x[i];
         int denom = 6;  // 3!
         int sign = -1;

         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

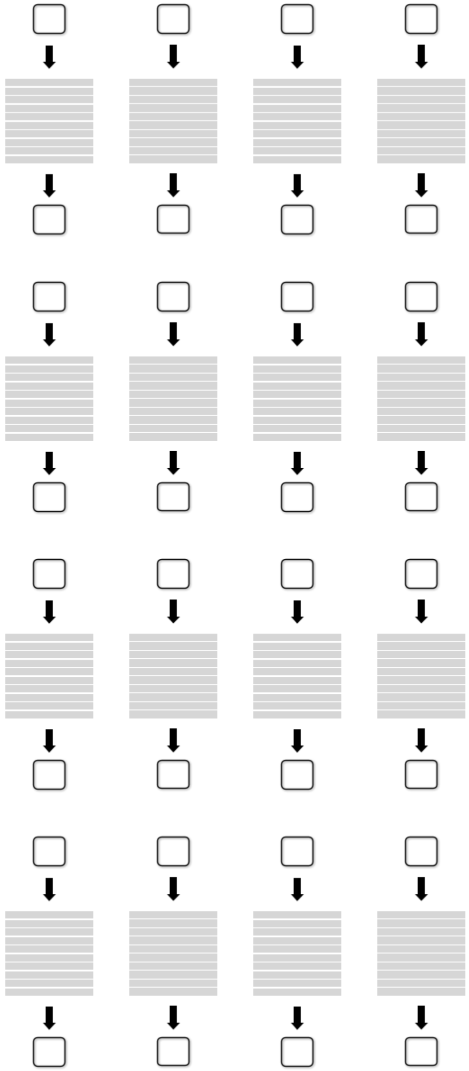
With this information, you could imagine how a compiler might automatically generate parallel threaded code

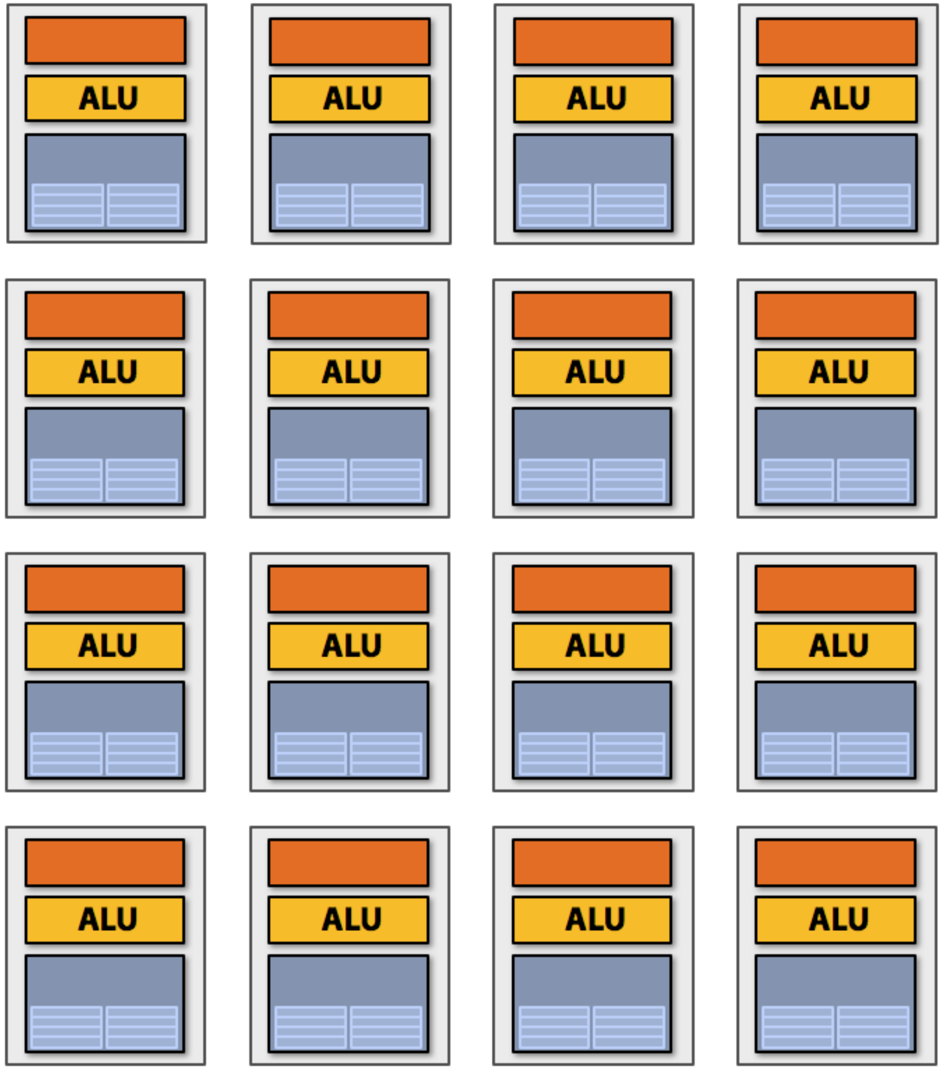
Four cores: compute four elements in parallel





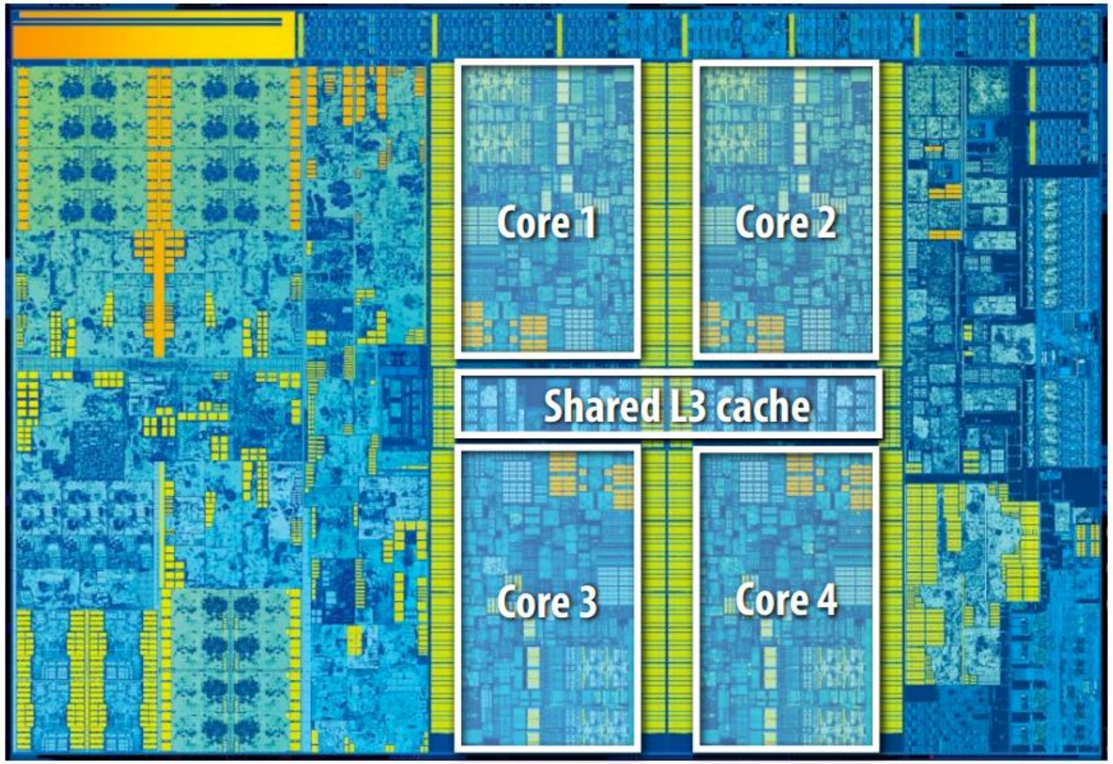
Sixteen cores: compute sixteen elements in parallel



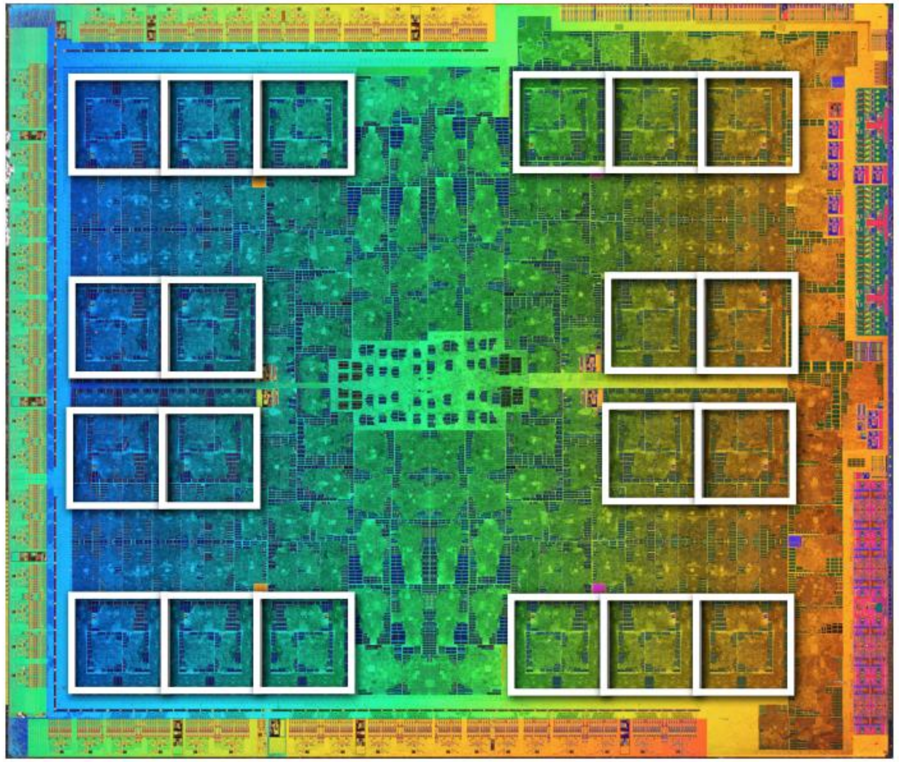


Sixteen cores, sixteen simultaneous instruction streams

Multi-core examples

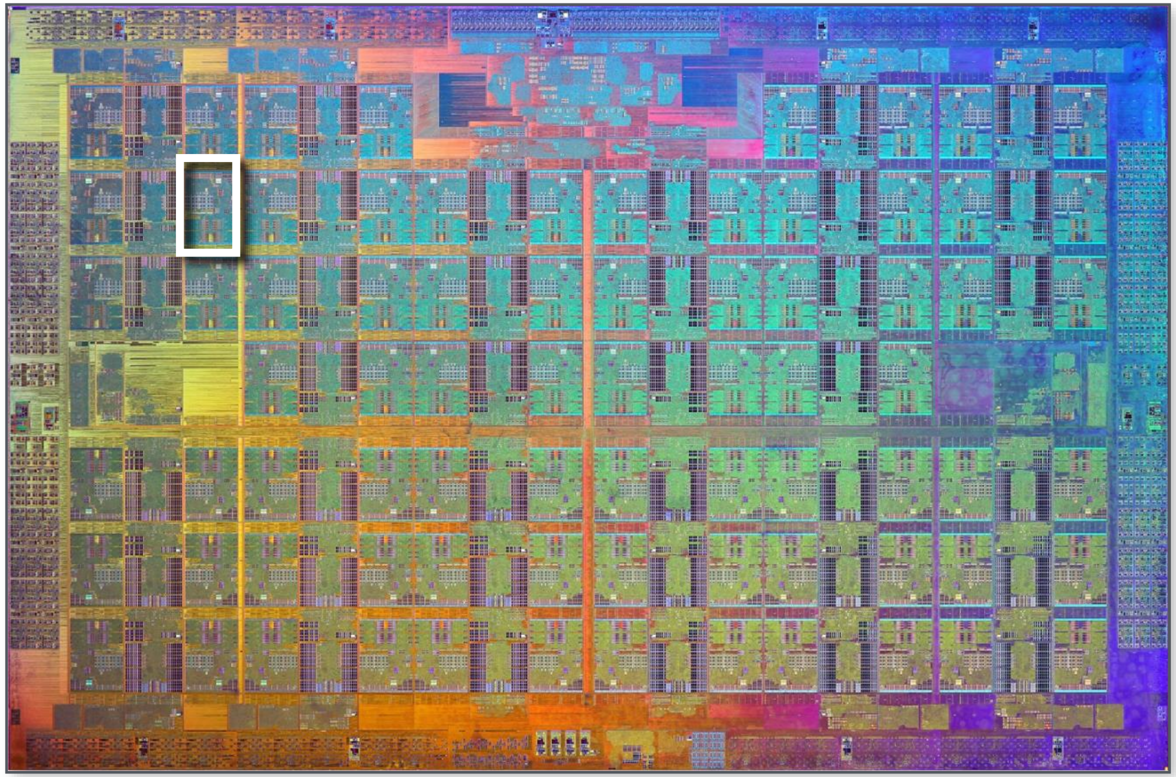


Intel “Skylake” Core i7 quad-core CPU
(2015)

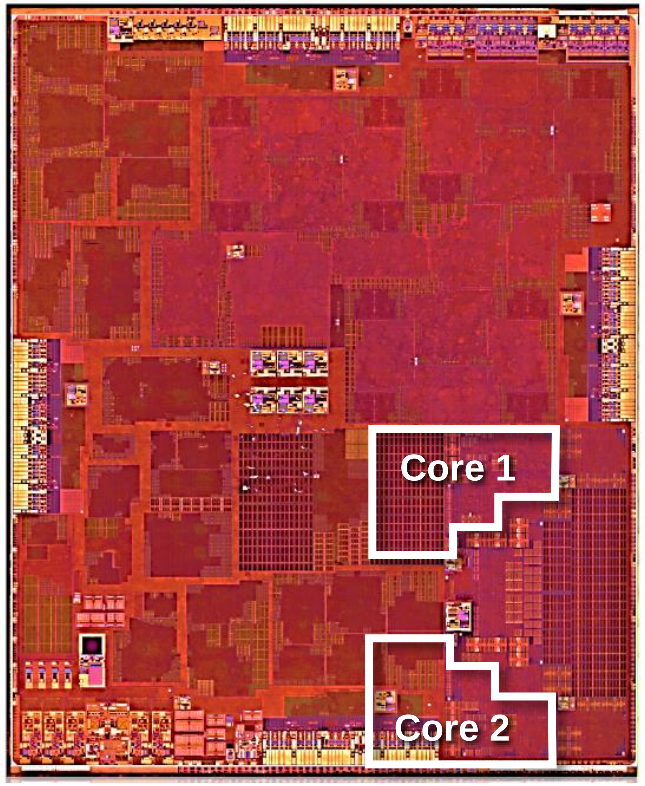


NVIDIA GP104 (GTX 1080) GPU
20 replicated (“SM”) cores
(2016)

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

Parallel 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
```

Summary so far

- **How to use the more transistors for better performance?**

- **Instruction-level parallelism (ILP)**
  - Automatically detect instructions that can be processed in parallel
  - Oblivious to users
  - Problem: can only achieve limited parallelism (<3)

- **Multiple processors**
  - Put more processors on the same chip for the additional space (transistors)
  - Can put more and more when we have more space on the chip
  - Users must write parallel algorithms and codes to utilize it