Lecture 1:

Why Parallelism?

Why Efficiency?

Parallel Computer Architecture and Programming
CMU / Tsinghua, Summer 2017

Carnegie Mellon University
School of Computer Science

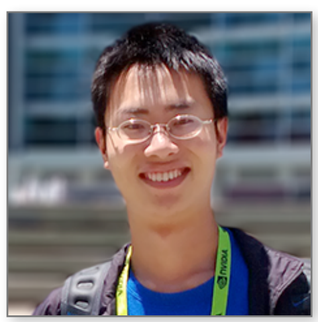
清華大學
Tsinghua University

Welcome!

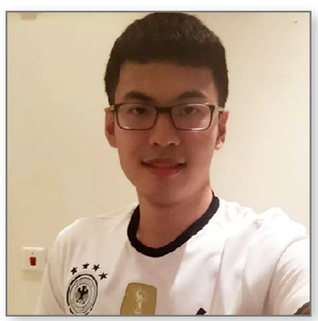
Prof. Kayvon Fatahalian (CMU)

Prof. Wei Xue (薛巍) (Tsinghua)

Teacher's Assistants: (TA's)



Yong He
Ph.D. student
(CMU)



Xu Ji
Ph.D. student
(Tsinghua)

Yan Gu
Ph.D. student
(CMU)

Ping Xu
M.S. student
(Tsinghua)

Today's picture ;-)

So what is a parallel computer?

Sunway TaihuLight

Is this a parallel computer?

One common definition…

A parallel computer is a collection of processing elements that cooperate to solve problems quickly

We care about performance
We care about efficiency

We're going to use multiple processors to get it

DEMO 1

(our first parallel program)

Speedup

One major motivation of using parallel processing: achieve a speedup

For a given problem:

$$\text{speedup(using P processors)} = \frac{\text{execution time (using 1 processor)}}{\text{execution time (using P processors)}}$$

Class observations from demo 1

- **Communication limited the maximum speedup achieved**
  - **In the demo, the communication was telling each other the partial sums**

- **Minimizing the cost of communication improved speedup**
  - **We could have moved students ("processors") closer together (or let them shout to each other)**

DEMO 2

(scaling up to four “processors”)

Class observations from demo 2

- Imbalance in work assignment limited speedup
  - Some students ("processors") ran out work to do (went idle), while others were still working on their assigned task

- Improving the distribution of work improved speedup

DEMO 3

(massively parallel execution)

Class observations from demo 3

- The problem I just gave you has a significant amount of communication compared to computation
- Communication costs can dominate a parallel computation, severely limiting speedup

Course theme 1:

Designing and writing parallel programs ... that scale!

- **Thinking about efficiently performing tasks in parallel**
  1. **Decomposing work into pieces that can safely be performed in parallel**
  2. **Assigning work to processors**
  3. **Managing communication/synchronization between the processors so that it does not limit speedup**

- **Abstractions/mechanisms for performing the above tasks**
  - **Writing code in popular parallel programming languages**

Course theme 2:

Parallel computer hardware implementation: how parallel computers work

- Mechanisms used to implement abstractions efficiently
  - Performance characteristics of implementations
  - Design trade-offs: performance vs. convenience vs. cost

- Why do I need to know about hardware?
  - Because the characteristics of the machine really matter (recall speed of communication issues in earlier demos)
  - Because you care about efficiency and performance (you are writing parallel programs after all!)

Course theme 3:

Thinking about efficiency

- **IT RUNS FAST != IT IS EFFICIENT**

- **Just because your program runs faster on a parallel computer, it does not mean it is using the hardware efficiently**
  - **Is 2x speedup on computer with 10 processors a good result?**

- **Programmer's perspective: make use of provided machine capabilities**

- **HW designer's perspective: choosing the right capabilities to put in system (performance/cost, cost = silicon area?, power?, etc.)**

Course information

What background should you have for this course?

- You should be comfortable writing and debugging C programs
  - You will be expected to be able to pick up new programming languages (ISPC, CUDA) with only a little help from the TAs

- You should understand the basics of how a computer works, but we will review a bit of that today
  - What is a program? What are machine instructions?
  - How does a computer run a program?
  - What are registers? and memory?

Ways to get course information and help

- learn.tsinghua website
  - We will post all materials on this site

- http://15418.courses.cs.cmu.edu/tsinghua2017
  - We will also post materials on this site
  - It will be helpful for asking questions about slides

- WeChat group (please tell the TAs to add you to the group)

There will be three programming assignments

Each assignment uses a different parallel programming environment

Assignment 1: ISPC programming on multi-core CPUs

Assignment 2: CUDA programming on NVIDIA GPUs

Assignment 3: fine-grained locking on multi-core CPUs

Programming assignment grades will be based on the performance of your code

There will be four written assignments

- Written assignments will be handed out each Monday afternoon
- They will be due on Thursday at 18:00
- We will grade your solutions to give feedback, then you can resubmit improved solutions once more for 90% credit
  - There will be extra opportunities for students to earn up to 100% of the original credit (to be announced ;-))

* The first week is an exception:
  We will release the first exercise on Wednesday and it will be due next Monday

Grades

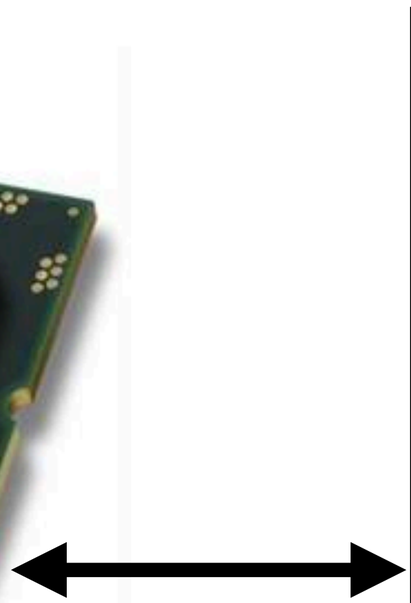
- 60% of grade will be based on programming assignments

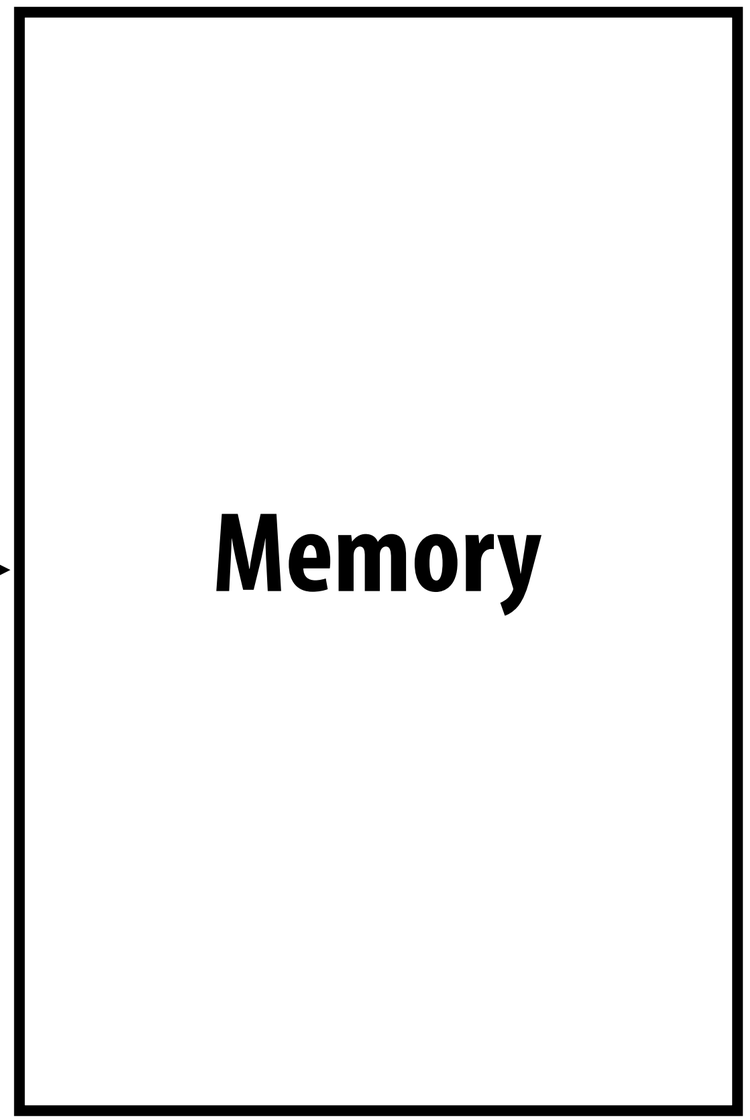
- 40% will be based on written assignments

Review:

Basics of how a computer works

My simple computer: 1 processor + memory





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;
}
```

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

```
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:  retq
```

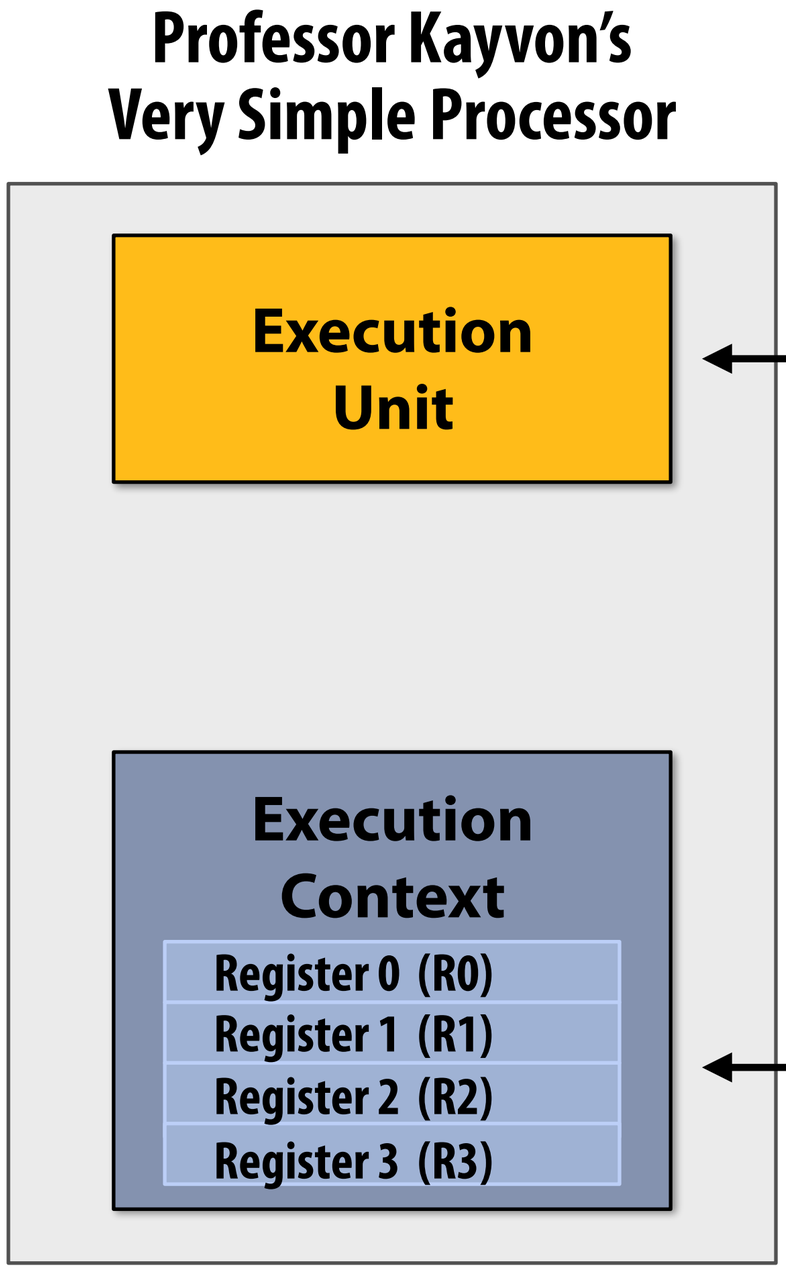
A program is just a list of processor instructions!

What is a processor?

What is a processor?

A processor executes instructions.

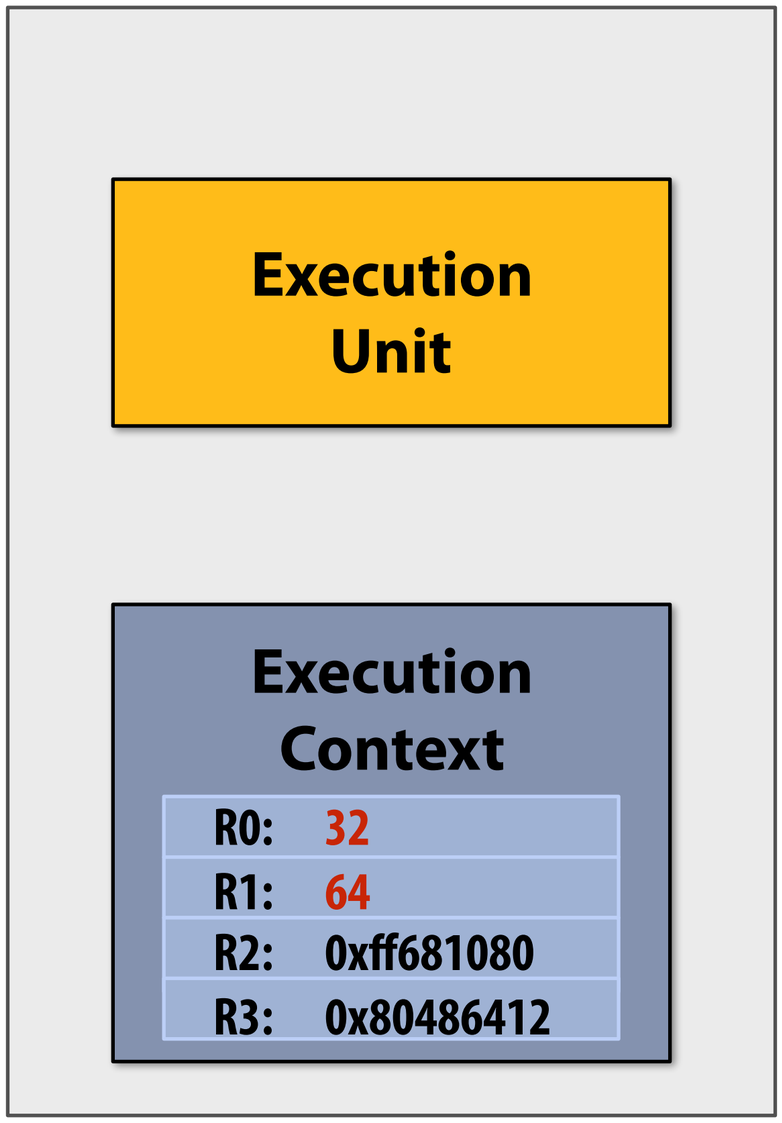
Executing an instruction modifies the computer's "state"



Execution unit: performs the operation required by an instruction, which may modify values in registers or memory

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

One example instruction: add



Step 1:
Processor gets next program instruction from memory (figure out what the processor should do next)

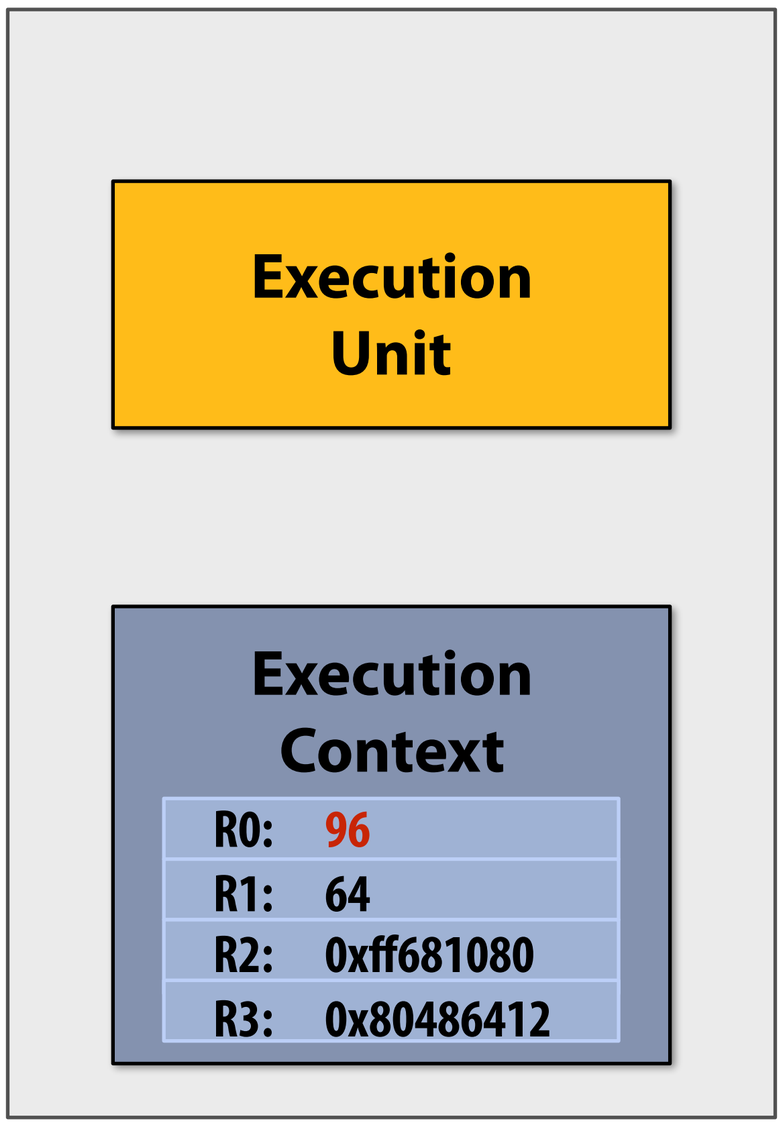
```
add R0 ← R0, R1
```

"Please add the contents of register R0 to the contents of register R1 and put the result in register R0"

Step 2:
Obtain inputs to the operations from registers
Contents of R0 input to execution unit: 32
Contents of R1 input to execution unit: 64

Step 3:
Perform addition operation:
Execution unit performs arithmetic, the result is: 96

One example instruction: add



Step 1:
Processor gets next program instruction from memory (figure out what the processor should do next)

```
add R0 ← R0, R1
```

"Please add the contents of register R0 to the contents of register R1 and put the result in register R0"

Step 2:
Obtain inputs to the operations from registers
Contents of R0 input to execution unit: 32
Contents of R1 input to execution unit: 64

Step 3:
Perform addition operation:
Execution unit performs arithmetic, the result is: 96

Step 4:
Store result back to register R0
(the state of register 0 has changed as a result of the instruction)

Modern processors execute a wide range of instructions

(there are many other examples not shown here, for example…
floating point math operations)

Integer arithmetic operations:

```
add dst, src1, src2          // dst = src1 + src2  “add src1 and src2”
sub dst, src1, src2          // dst = src1 - src2  “subtract src2 from src1”
mul dst, src1, src2          // dst = src1 * src2
div dst, src1, src2          // dst = src1 / src2
rem dst, src1, src2          // dst = src1 % src2
min dst, src1, src2          // dst = min(src1, src2)
max dst, src1, src2          // dst = max(src1, src2)
```

Logical operations:

```
and dst, src1, src2          // dst = src1 & src2 “logical and of src1 and src2”
or  dst, src1, src2          // dst = src1 | src2
xor dst, src1, src2          // dst = src1 ^ src2
```

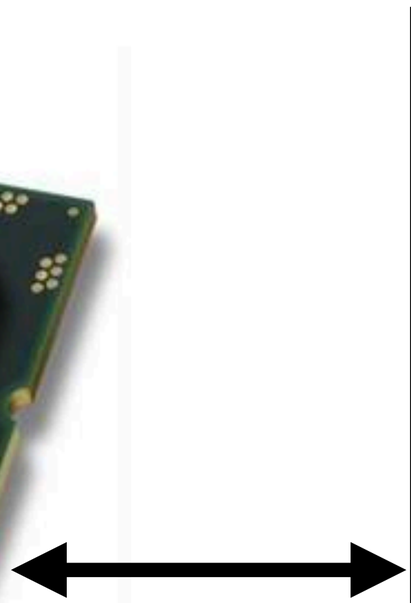
Bit shifts:

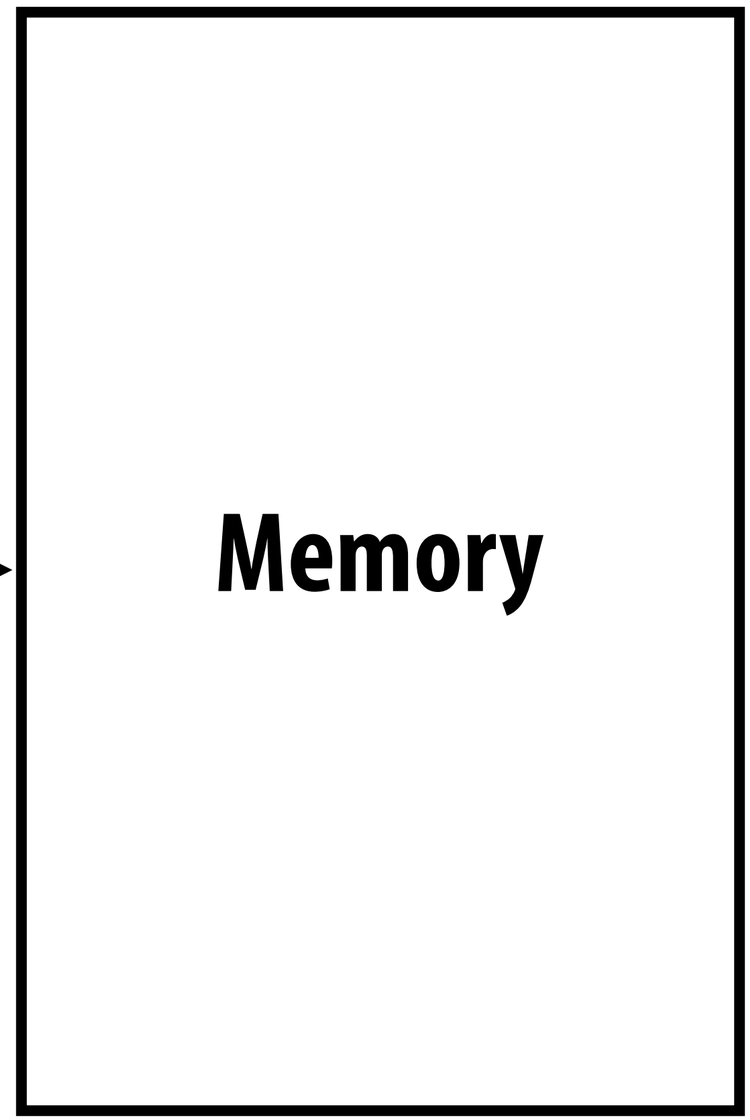
```
sll dst, src                 // dst = src << 1  “shift bits of src1 left by one”
srr dst, src                 // dst = src >> 1  “shift bits of src1 right by one”
```

Comparisons:

```
slt dst, src1, src2          // dst = (src1 < src2) ? 1 : 0
```

What is memory?





A program's memory address space

- A computer's memory is organized as a array of bytes

- Each byte is identified by its "address" in memory (its position in this array)
  (in this class we assume memory is byte-addressable)

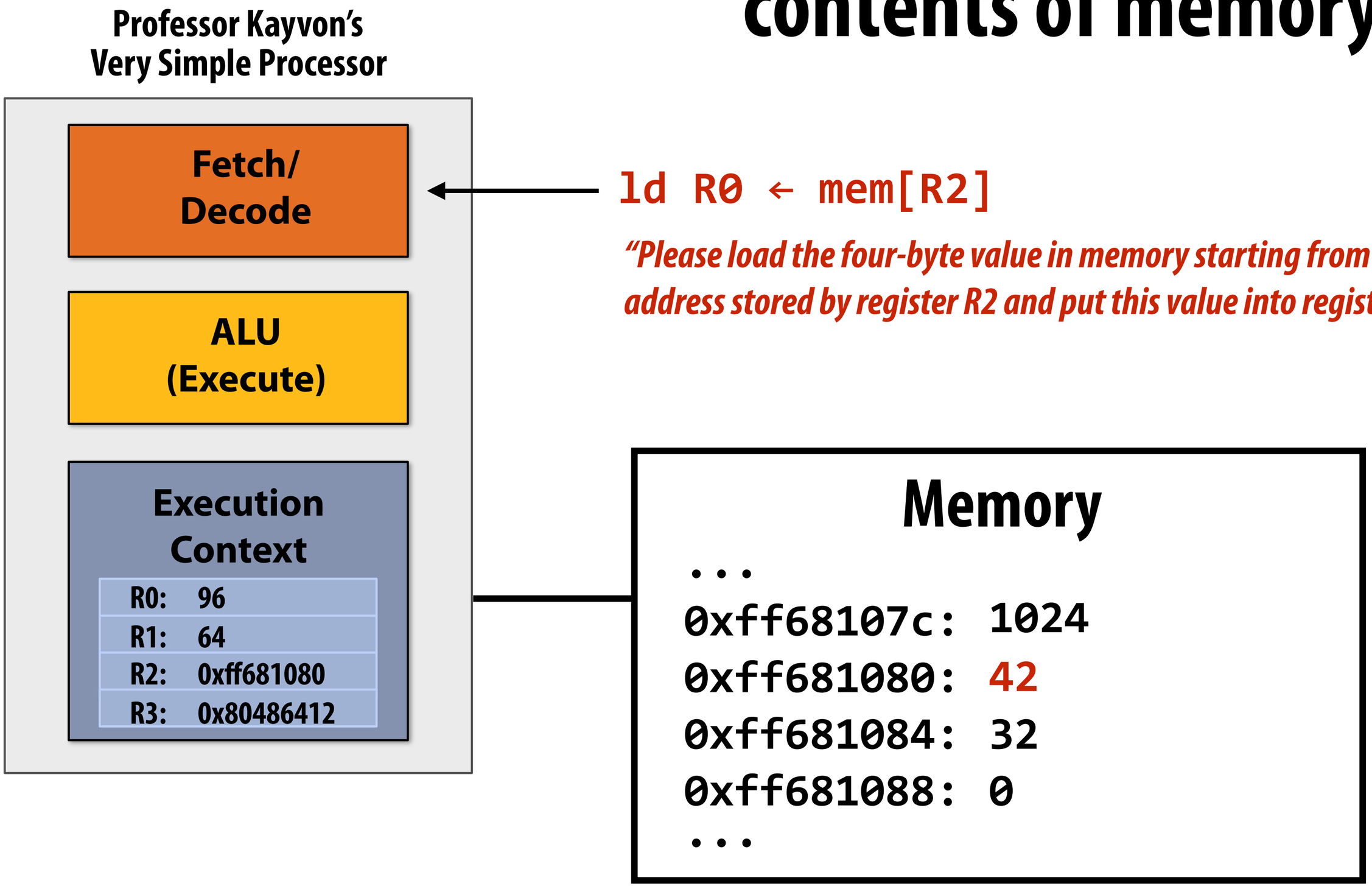
"The byte stored at address 0x8 has the value 32."

"The byte stored at address 0x10 (16) has the value 128."

In the illustration on the right, the program's memory address space is 32 bytes in size (so valid addresses range from 0x0 to 0x1F)

Address	Value
0x0	16
0x1	255
0x2	14
0x3	0
0x4	0
0x5	0
0x6	6
0x7	0
0x8	32
0x9	48
0xA	255
0xB	255
0xC	255
0xD	0
0xE	0
0xF	0
0x10	128
⋮	⋮
0x1F	0

Load: an instruction for accessing the contents of memory



Review of how computers work…

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

It is a list of instructions to execute!

What is an instruction?

It describes an operation for a processor to perform. Executing an instruction typically modifies the computer's state.

What do I mean when I say a computer's "state"?

The values of program data, which are stored in a processor's registers or in memory.

Tip for students

- If the material in the last couple of slides was new to you (or it was difficult to understand), you will probably need to do extra preparation for this class

- Please see the course web site for suggested review readings
  - And talk to the TAs

Why parallel computing?

Let's go back to when I began my undergraduate degree in computer science …

A bit of historical context: why avoid parallel processing?

- Single-threaded CPU performance doubling ~ every 18 months
- Implication: working to parallelize your code was often not worth the time
  - Software developer does nothing, but code gets faster next year. Woot!

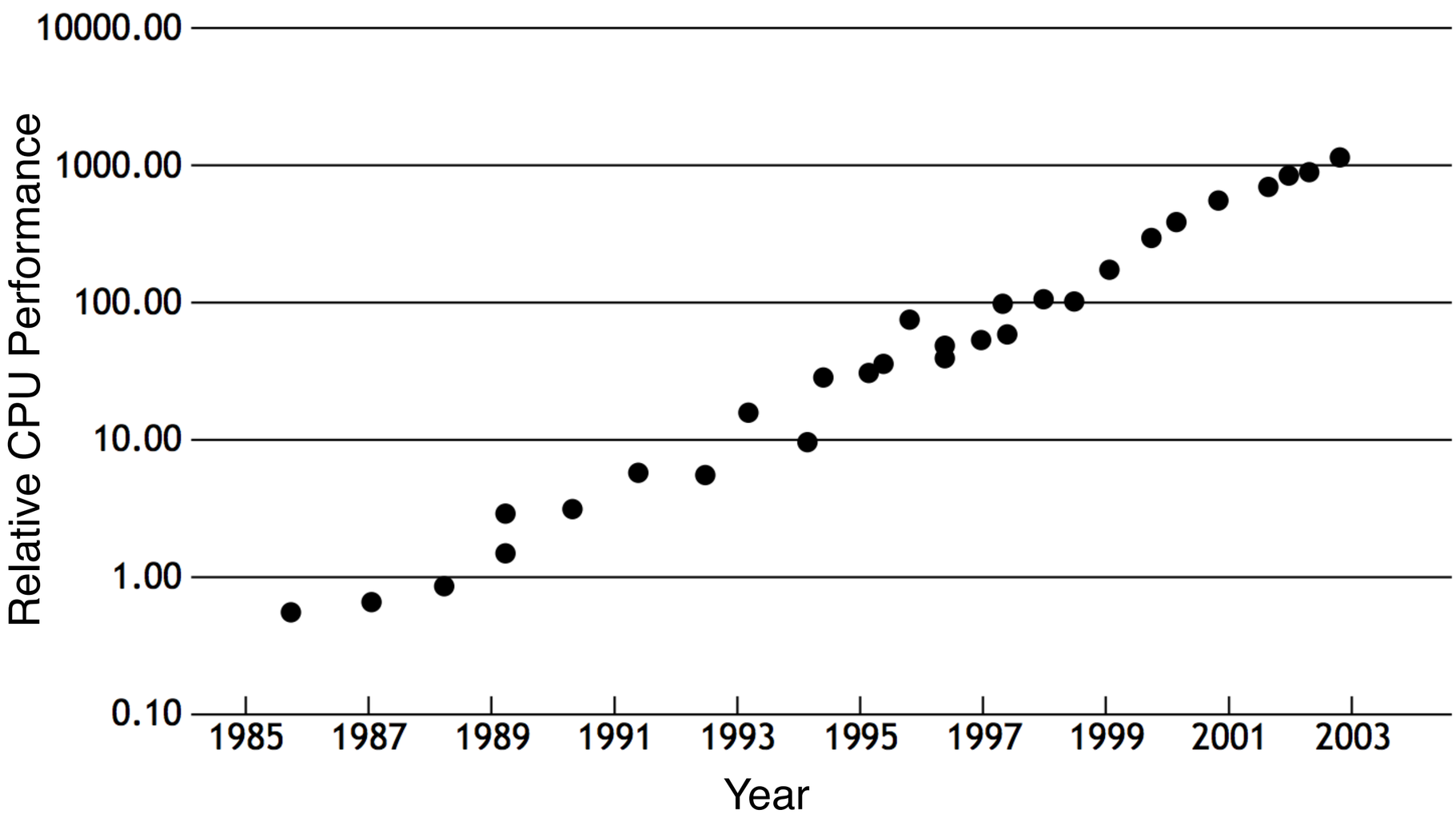
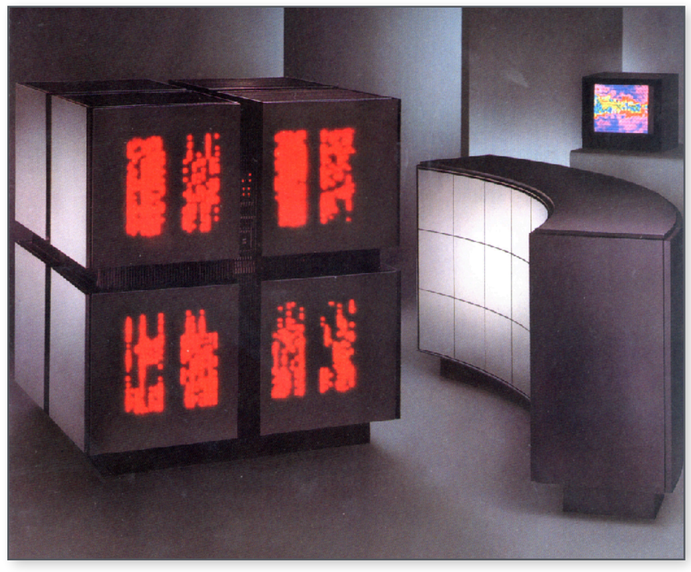


Image credit: Olukutun and Hammond, ACM Queue 2005

Why parallel processing? (80's, 90's, early 2000's)

The answer until 10-15 years ago: to realize performance improvements that exceeded what CPU performance improvements could provide

For supercomputing applications



Thinking Machines (CM2) (1987)

65,536 1-bit processors + 2,048 32 bit FP processors

SGI Origin 2000 — 128 CPUs (1996)

Photo shows ASIC Blue Mountain supercomputer at Los Alamos (48 Origin 2000's)

For database applications

Sun Enterprise 10000 (circa 1997)

64 UltraSPARC-II processors

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

1. Exploiting instruction-level parallelism (running more than one instruction per clock using superscalar execution)
2. Increasing CPU clock frequency (increasing the clock rate)

Review (again): 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. . .

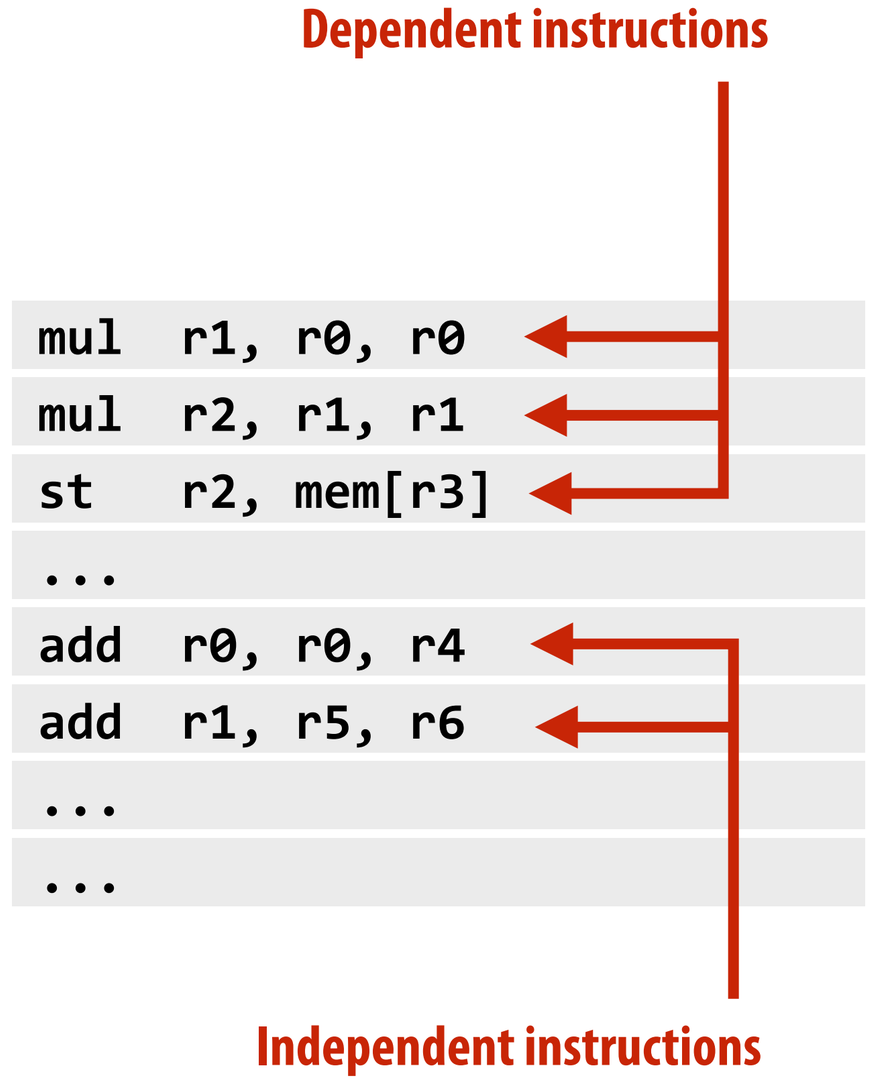
PC

```
_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)**
  - **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**



ILP example

$$a = x*x + y*y + z*z$$

Consider the following program:

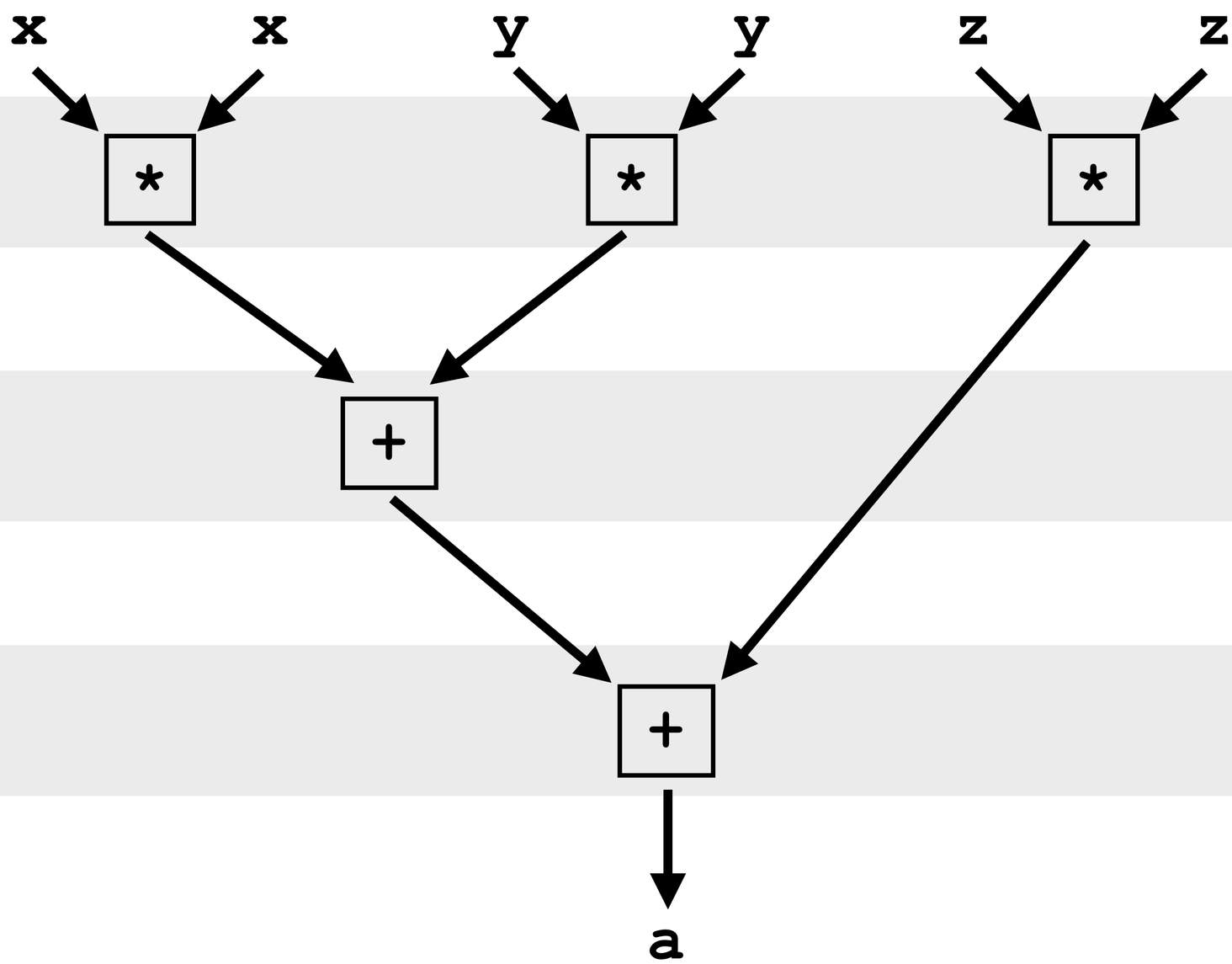
```
// assume r0=x, r1=y, r2=z

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 = 3

ILP = 1

ILP = 1

Superscalar execution

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

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

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

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

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

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

But instruction 4 must come after instructions 1 and 2

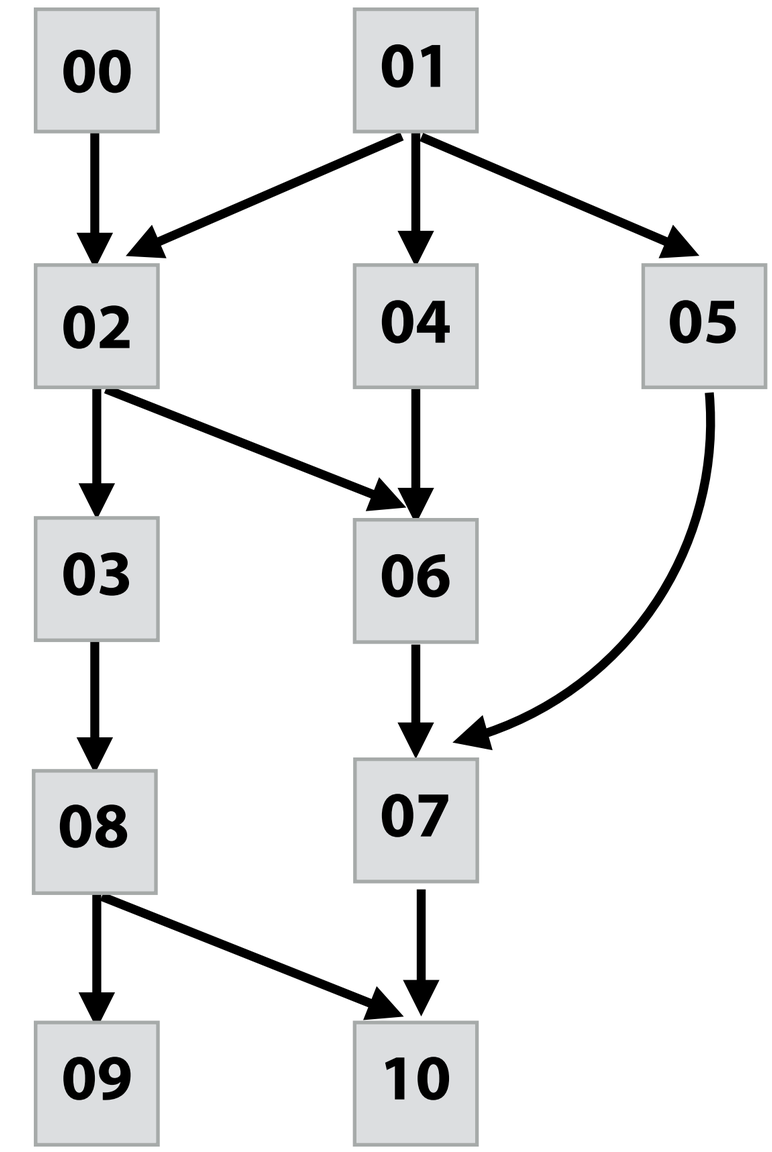
And instruction 5 must come after instruction 4

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	

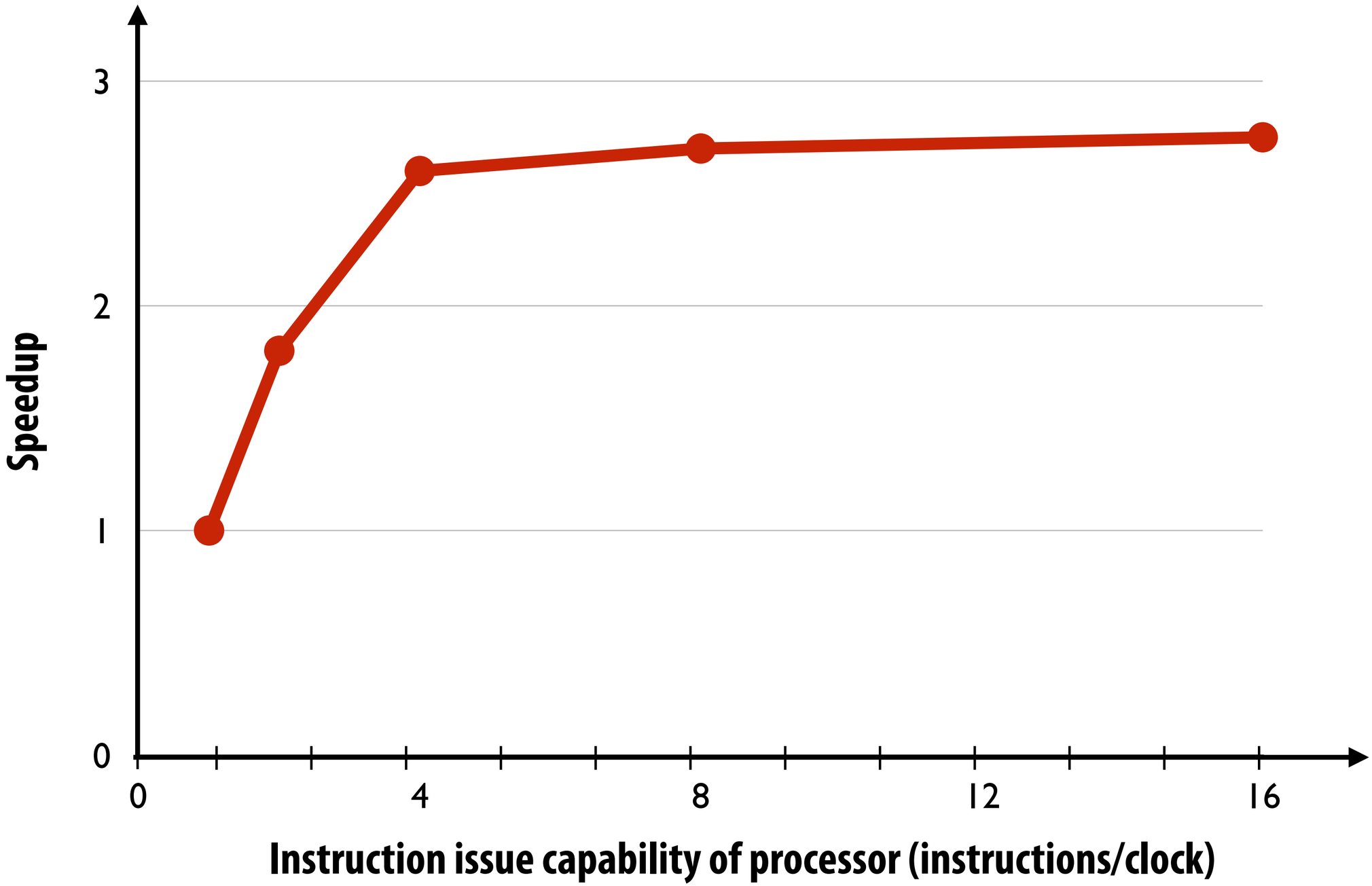
Instruction dependency graph



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

Diminishing benefit of superscalar execution

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



Source: Culler & Singh (data from Johnson 1991)

ILP tapped out + end of frequency scaling

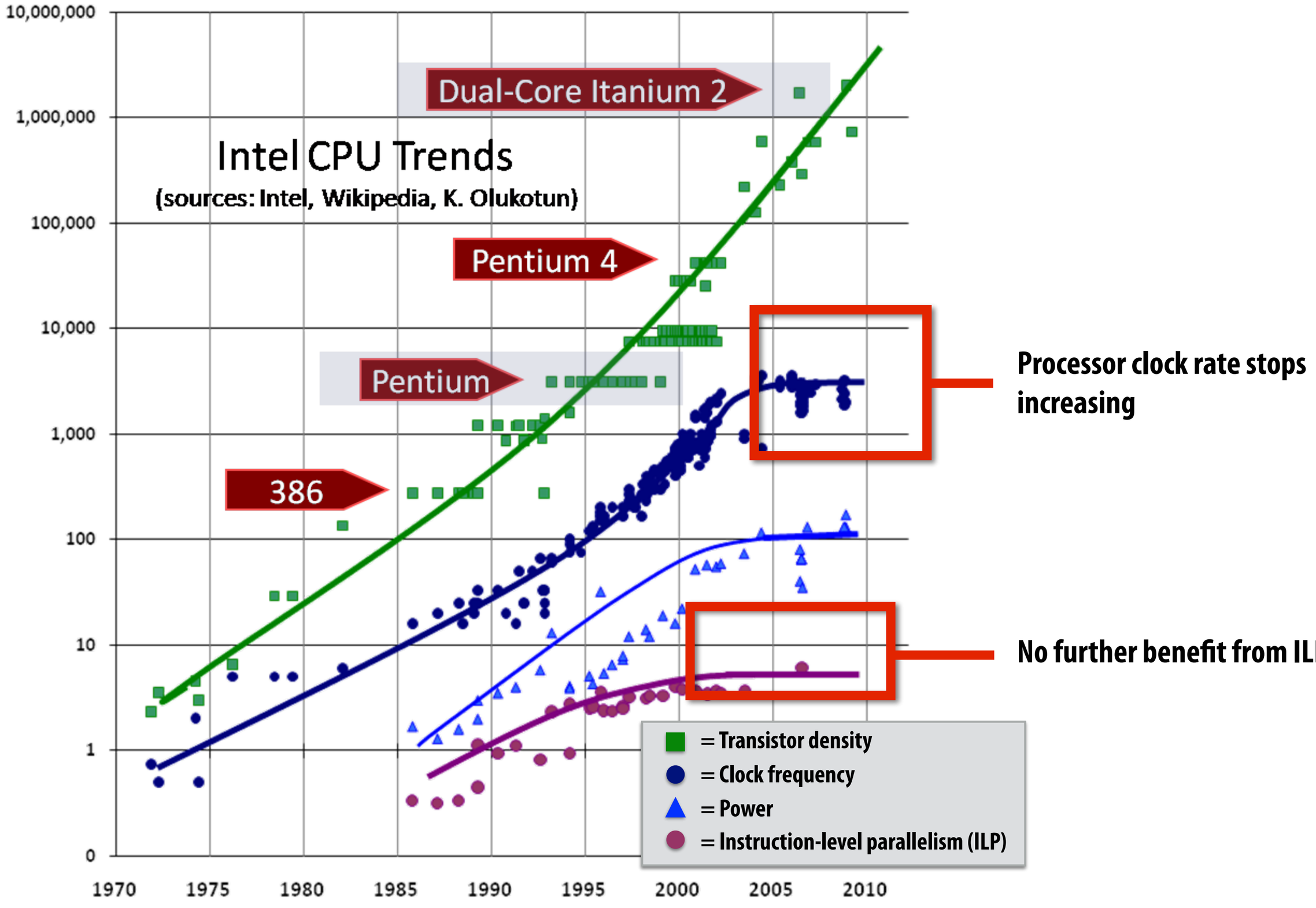


Image credit: "The free Lunch is Over" by Herb Sutter, Dr. Dobbs 2005

The “power wall”

Power consumed by a transistor:

Dynamic power $\propto$ capacitive load $\times$ voltage^2 $\times$ frequency

Static power: transistors burn power even when inactive due to leakage

High power = high heat

Power is a critical design constraint in modern processors

	TDP
Intel Core i7 (in this laptop):	45W
Intel Core i7 2700K (fast desktop CPU):	95W
NVIDIA GTX 780 GPU	250W
Mobile phone processor	1/2 - 2W
World’s fastest supercomputer	megawatts
Standard microwave oven	700W

Source: Intel, NVIDIA, Wikipedia, Top500.org

Power draw as a function of frequency

Dynamic power $\propto$ capacitive load $\times$ voltage^2 $\times$ frequency

Static power: transistors burn power even when inactive due to leakage

Maximum allowed frequency determined by processor's core voltage

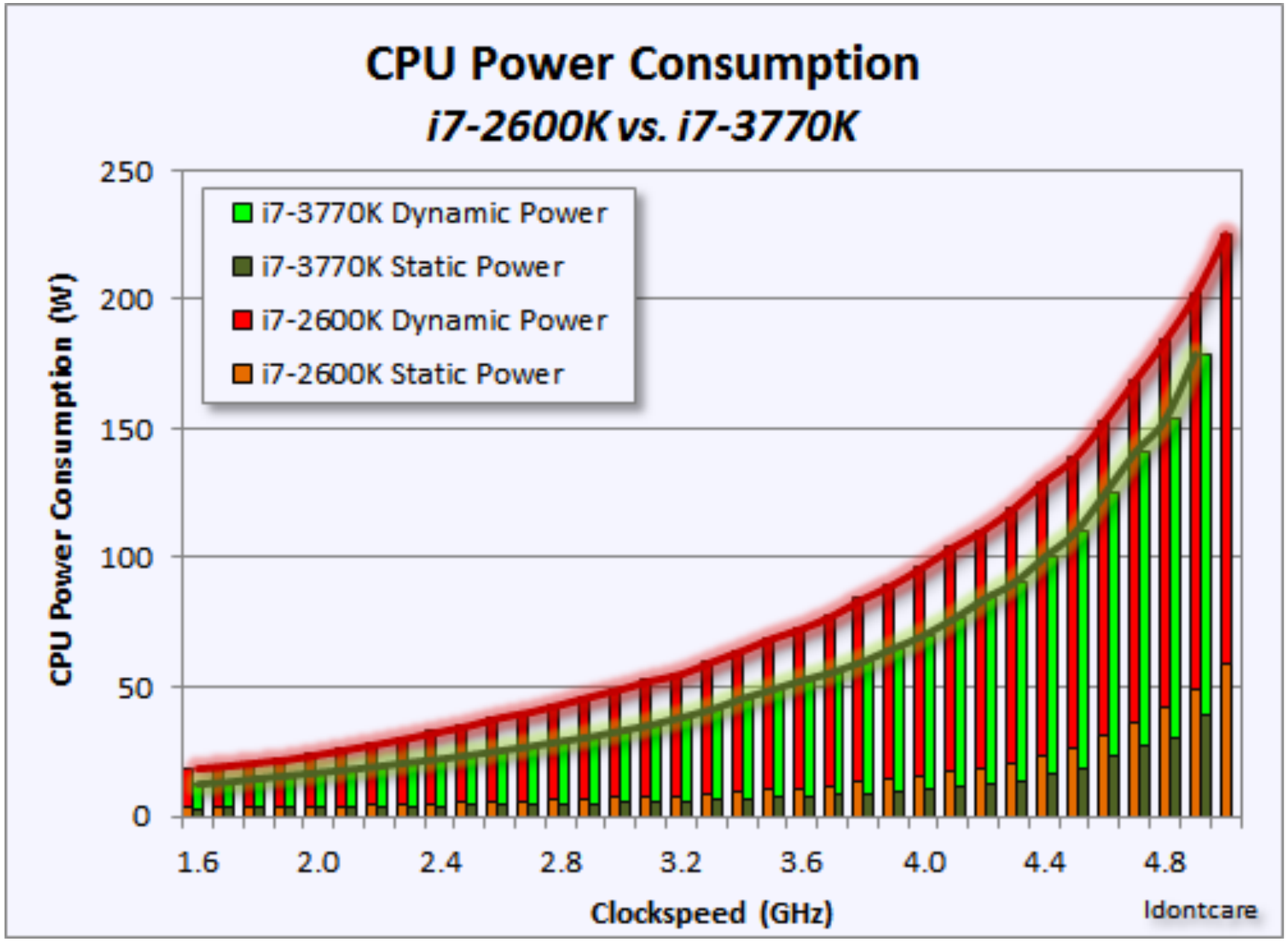


Image credit: "Idontcare": posted at: http://forums.anandtech.com/showthread.php?t=2281195

Single-core performance scaling

The rate of single-instruction stream performance scaling has decreased (almost to zero)

1. Frequency scaling limited by power

2. ILP scaling tapped out

Architects are now building faster processors by adding more execution units that run in parallel

(Or units that are specialized for a specific task (like graphics, or audio/video playback)

Software must be written to be parallel to see performance gains. No more free lunch for software developers!

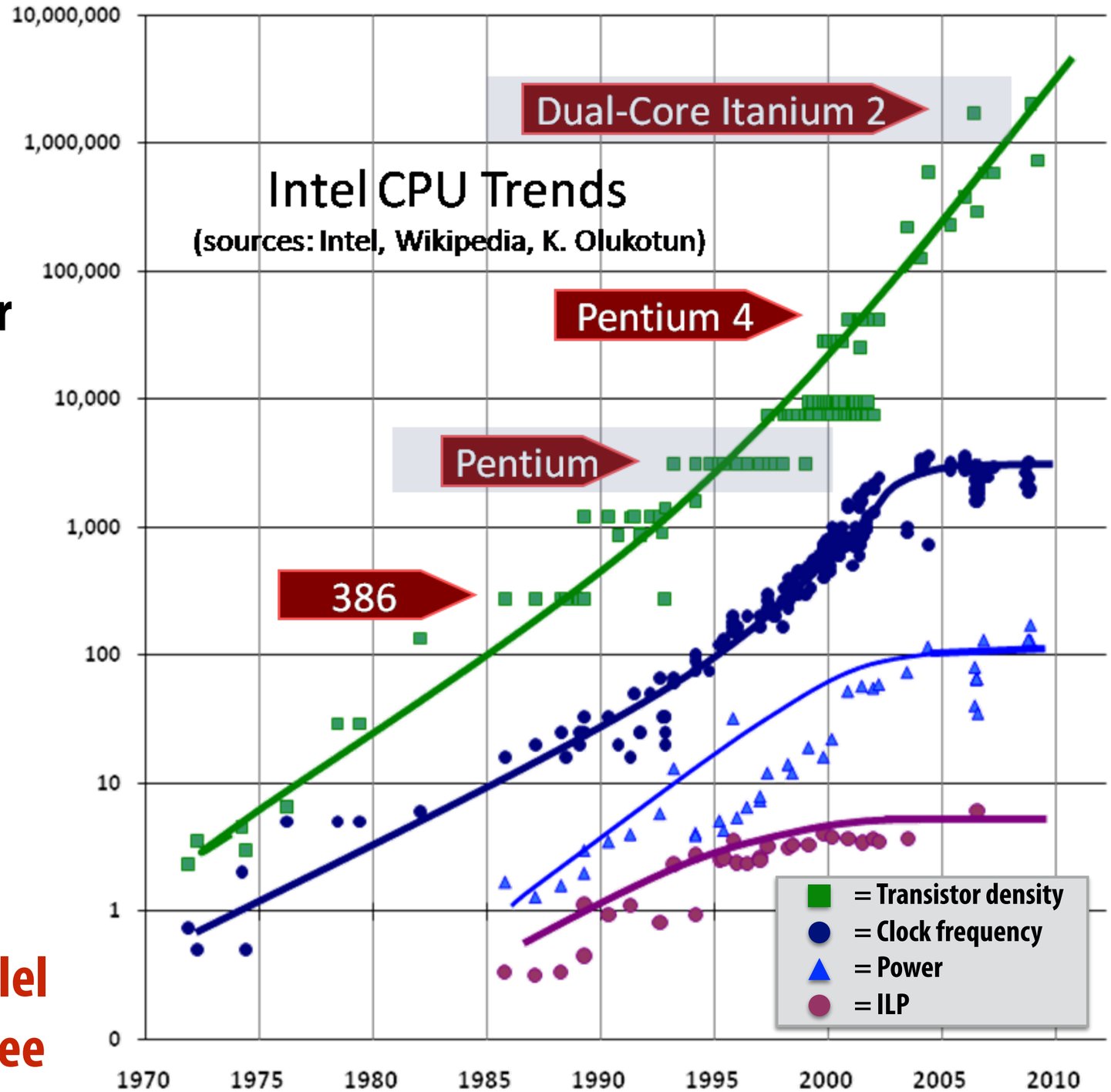


Image credit: "The free Lunch is Over" by Herb Sutter, Dr. Dobbs 2005

Shift to parallel software was a big change

Intel's Big Shift After Hitting Technical Wall

The warning came first from a group of hobbyists that tests the speeds of computer chips. This year, the group discovered that the Intel Corporation's newest microprocessor was running slower and hotter than its predecessor.

What they had stumbled upon was a major threat to Intel's longstanding approach to dominating the semiconductor industry - relentlessly raising the clock speed of its chips.

Then two weeks ago, Intel, the world's largest chip maker, publicly acknowledged that it had hit a "thermal wall" on its microprocessor line. As a result, the company is changing its product strategy and disbanding one of its most advanced design groups. Intel also said that it would abandon two advanced chip development projects, code-named Tejas and Jayhawk.

Now, Intel is embarked on a course already adopted by some of its major rivals: obtaining more computing power by stamping multiple processors on a single chip rather than straining to increase the speed of a single processor.

…

John Markoff, New York Times, May 17, 2004

Recap: why parallelism?

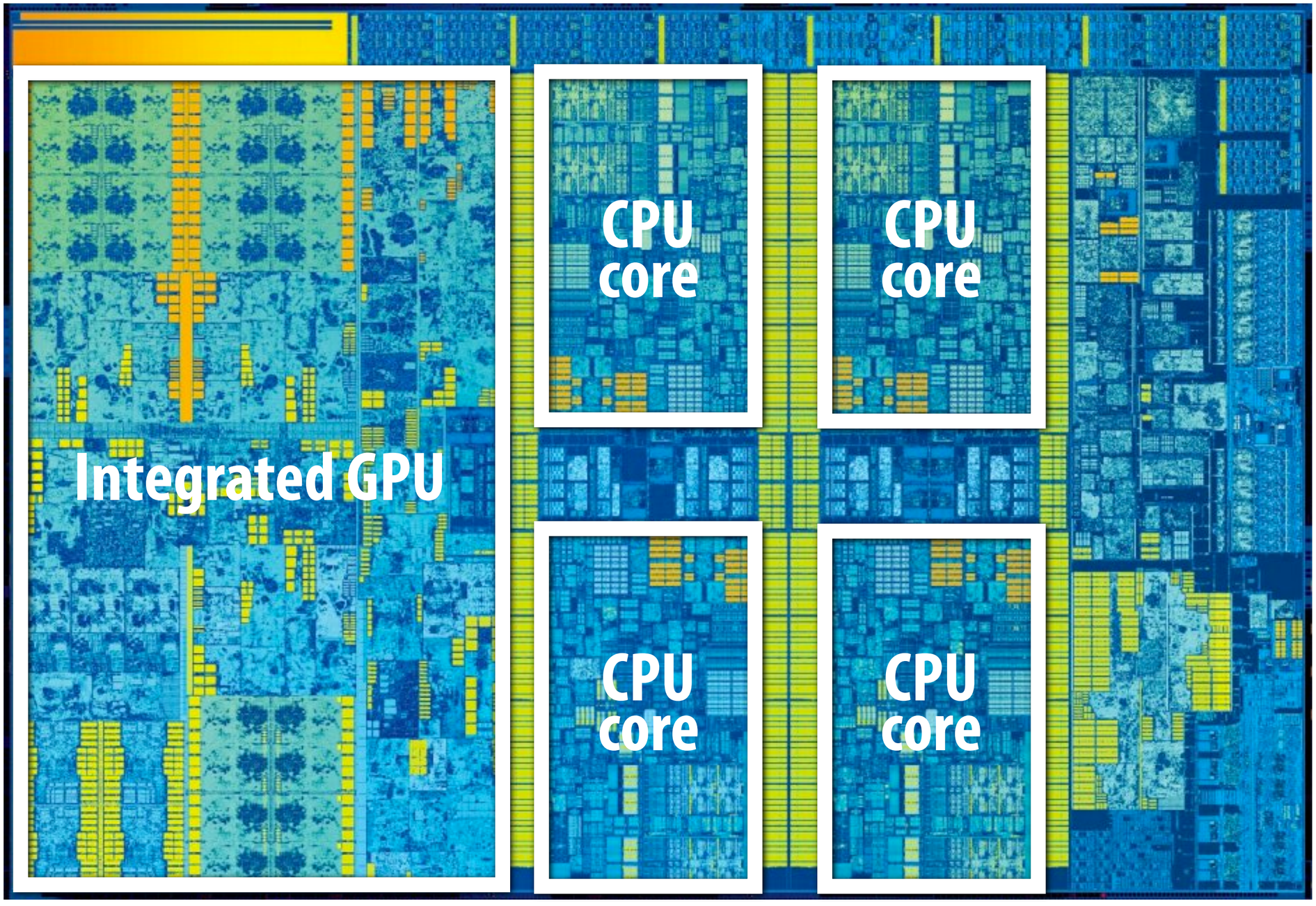
- **The answer 15 years ago**
  - To realize performance improvements that exceeded what CPU performance improvements could provide
    (specifically, in the early 2000's, what clock frequency scaling could provide)
  - Because if you just waited until next year, your code would run faster on the next generation CPU

- **The answer today:**
  - Because it is <u>the primary way</u> to achieve significantly higher application performance for the foreseeable future *

* We'll revisit this comment later in the heterogeneous processing lecture

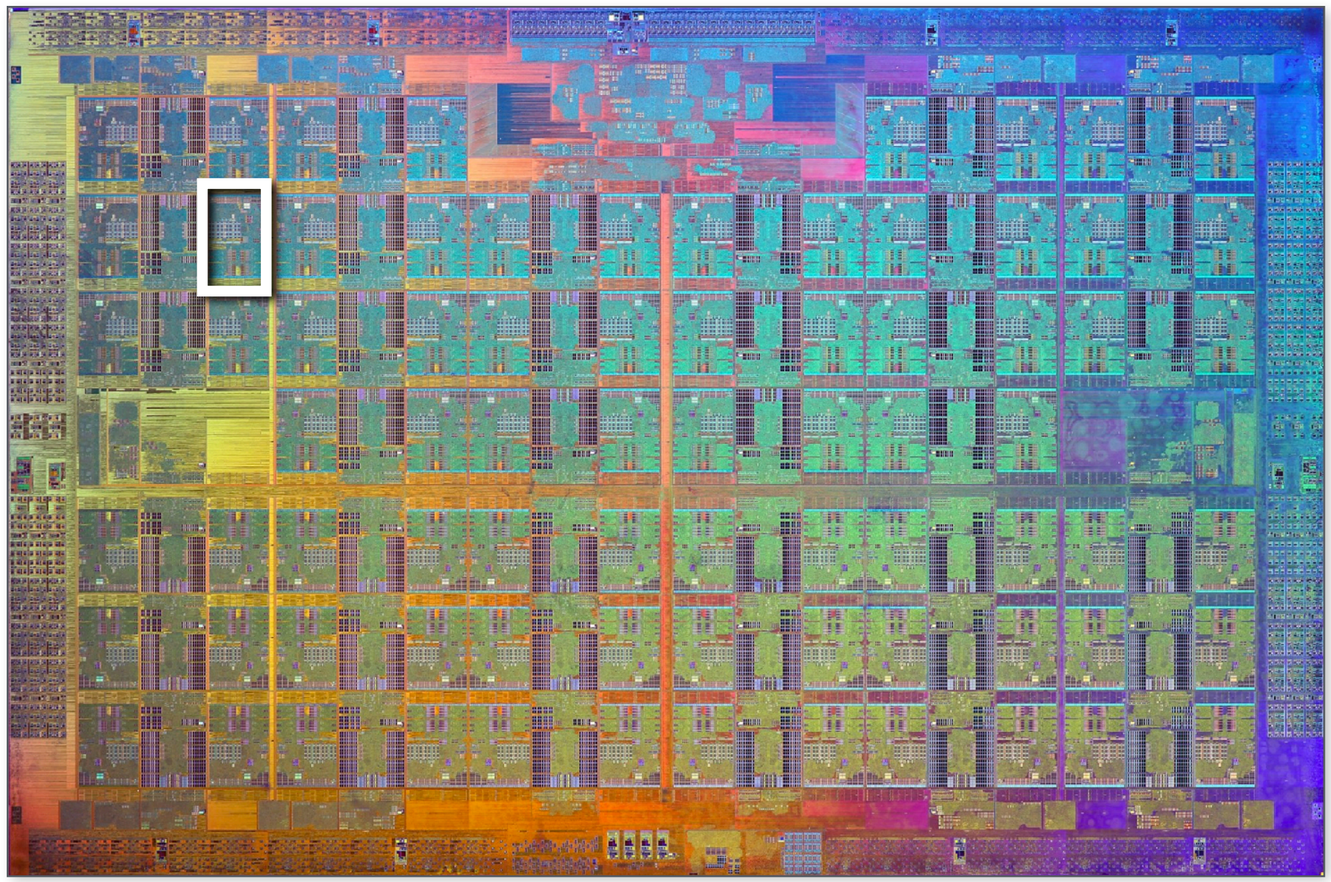
Intel Skylake (2015) (aka "6th generation Core i7")

Quad-core CPU + multi-core GPU integrated on one chip



Intel Xeon Phi 7290 "coprocessor" (2016)

72 cores (1.5 Ghz)



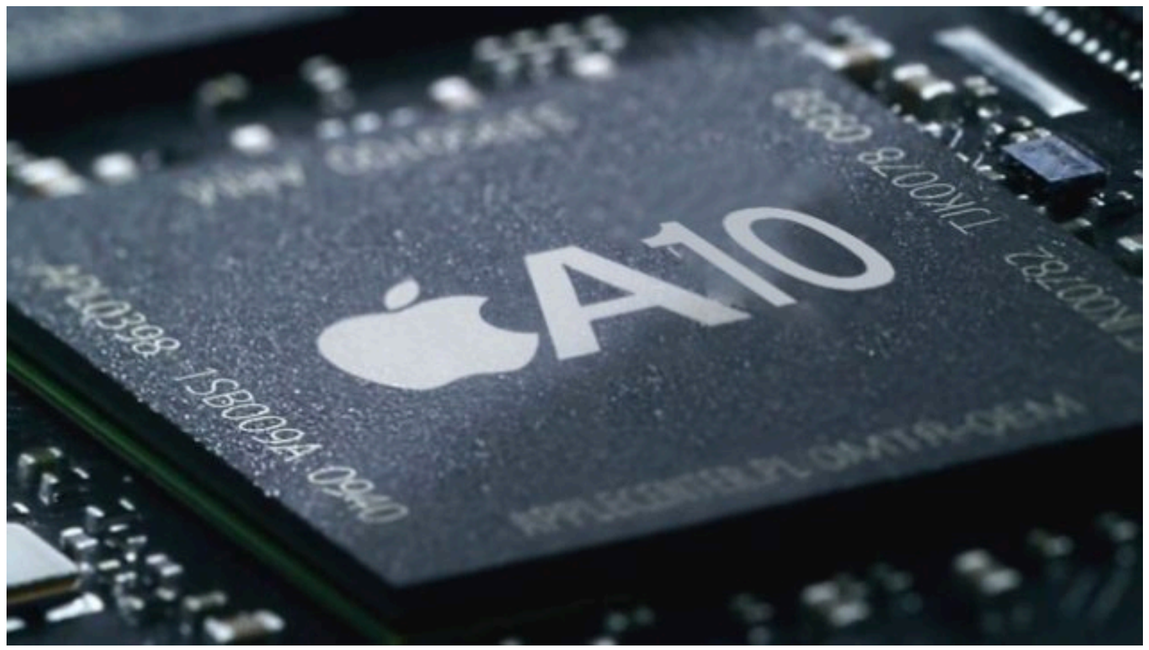
NVIDIA Maxwell GTX 1080 GPU (2016)

20 major processing blocks

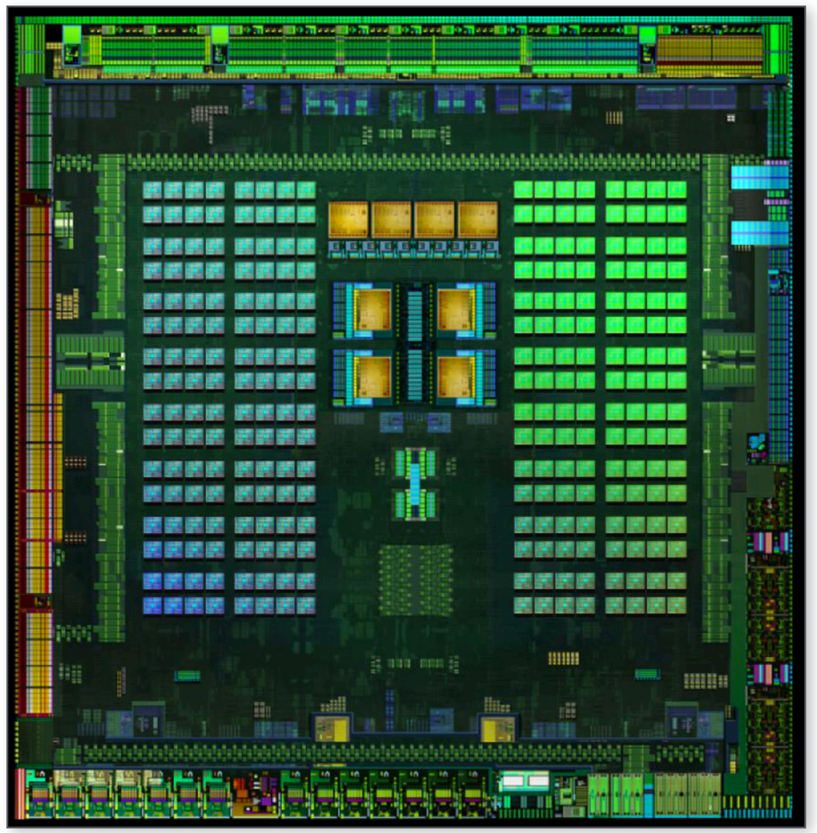
(but much, much more parallelism available... details coming in a future class)

Mobile parallel processing

Power constraints heavily influence design of mobile systems



Apple A10: (in iPhone 7)
2 “big” CPU cores + 2 “small” CPU cores + GPU
+ image processor (and more!) on one chip

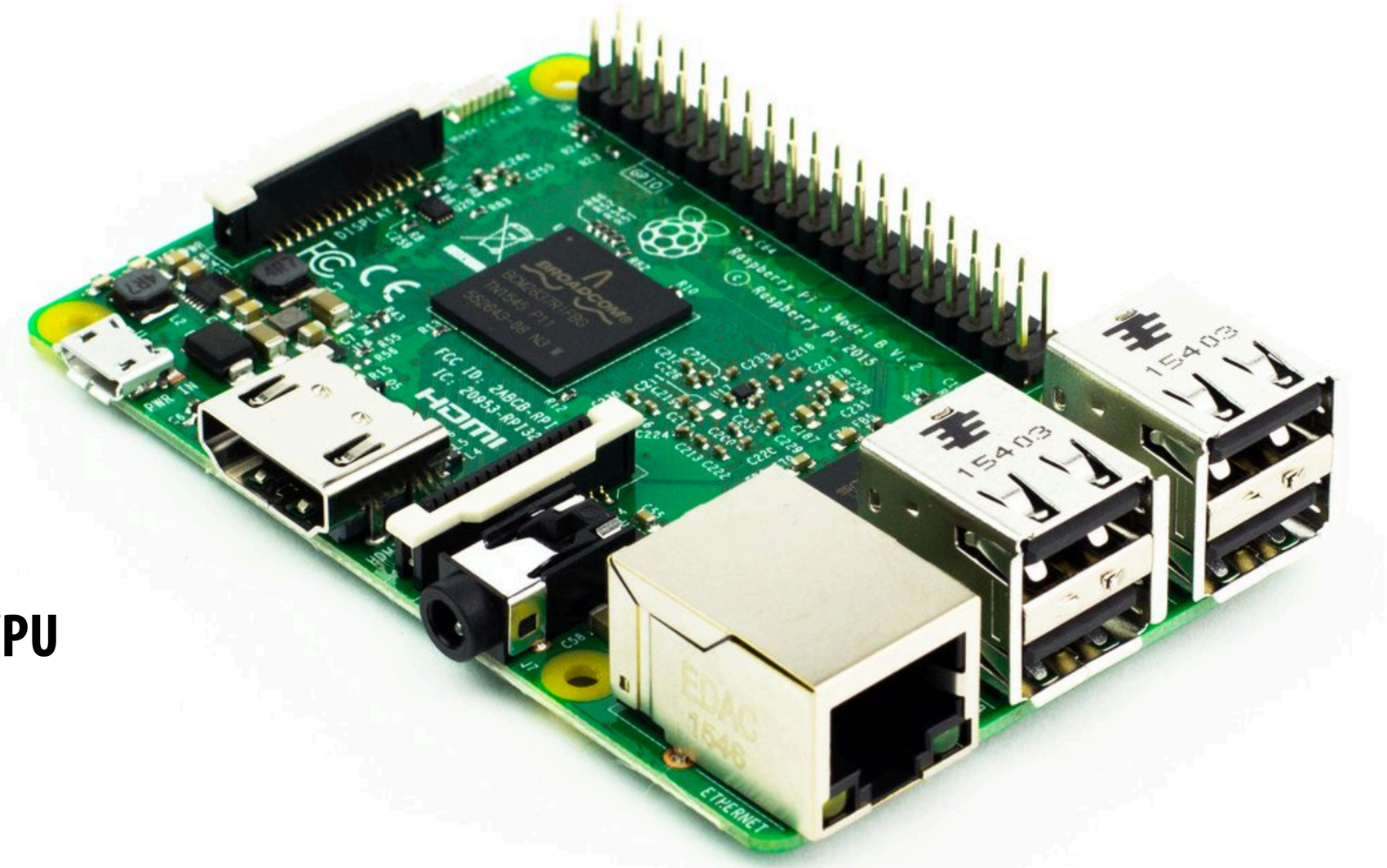


NVIDIA Tegra X1:
4 ARM A57 CPU cores +
4 ARM A53 CPU cores +
NVIDIA GPU + image processor...

Mobile parallel processing

Raspberry Pi 3

Quad-core ARM A53 CPU



Supercomputing

- Today: clusters of multi-core CPUs + GPUs
- Oak Ridge National Laboratory USA: Titan (#3 supercomputer in world)
  - 18,688 x 16 core AMD CPUs + 18,688 NVIDIA K20X GPUs

Supercomputing

- Sunway TaihuLight (#1 supercomputer in world)
  - 40,960 x 256 core processors = 10,485,760 CPU cores

Summary

- Today, single-thread-of-control performance is improving very slowly
  - To run programs significantly faster, programs must utilize multiple processing elements
  - Which means you need to know how to write parallel code

- Writing parallel programs can be challenging
  - Requires problem partitioning, communication, synchronization
  - Knowledge of machine characteristics is important

- I suspect you will find that modern computers have tremendously more processing power than you might realize, if you just use it!

- Welcome to the class!