

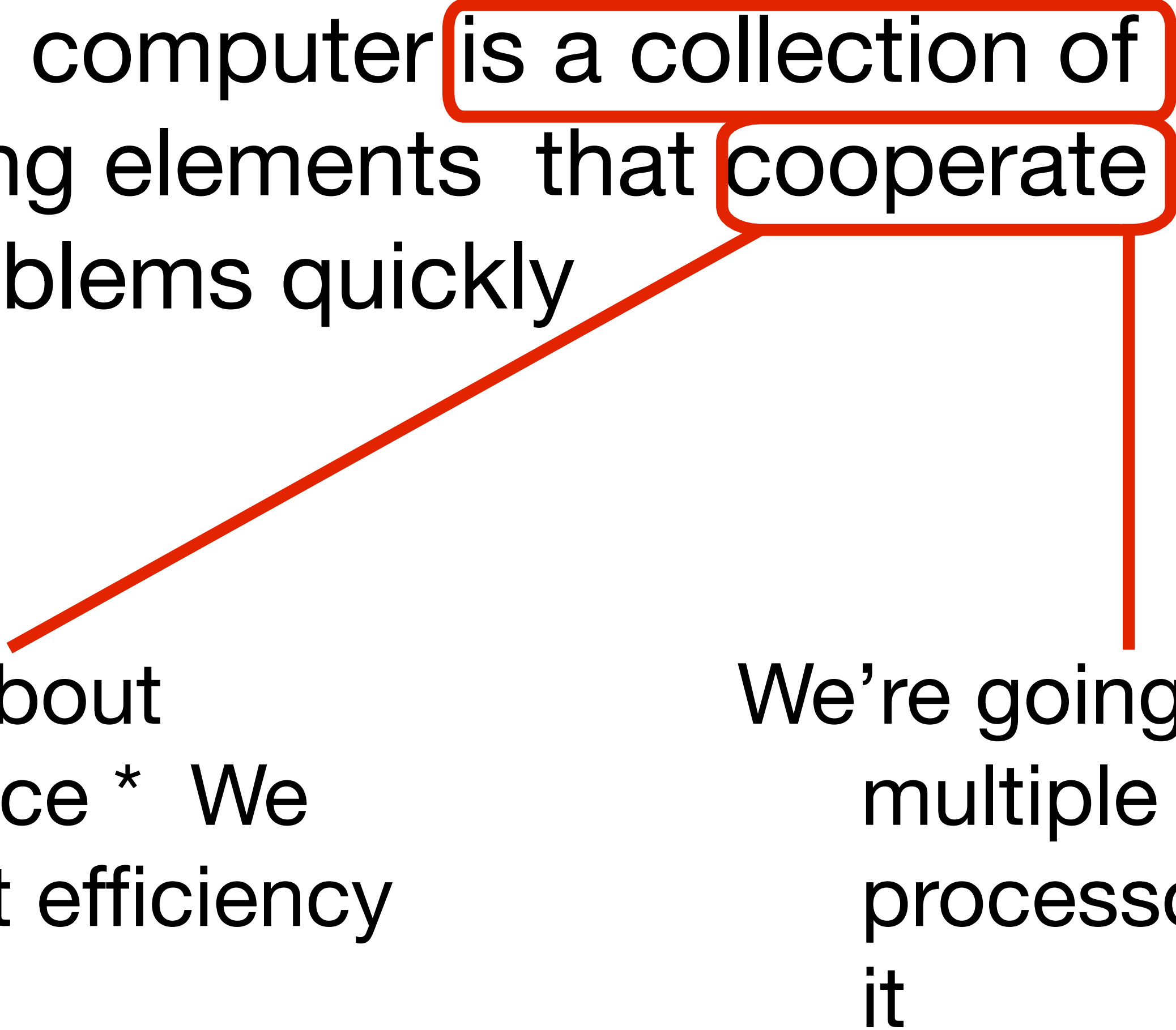
# **Why Parallelism?**

# **Why Efficiency?**

---

# One common definition

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



The diagram features a main definition sentence at the top. Two red rounded rectangular boxes highlight the words 'is a collection of' and 'cooperate'. A red line extends from the bottom of the 'cooperate' box to the left, pointing towards the text 'We care about performance \* We care about efficiency'. Another red line extends from the bottom of the 'cooperate' box to the right, pointing towards the text 'We're going to use multiple processors to get it'.

We care about performance \* We care about efficiency

We're going to use multiple processors to get it

\* Note: different motivation from “concurrent programming” using pthreads

# Speedup

One major motivation of using parallel processing:  
achieve a speedup

For a given  
problem:

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

# Course theme 1:

## Designing and writing parallel programs ... that scale!

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
  - Because you care about efficiency and performance (you are writing parallel programs after all!)

# Course theme 3:

## Thinking about efficiency

- FAST  $\neq$  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 good ?
- 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.)

# Participation suggestion (comments)

- You are encouraged to submit one well-thought-out comment per lecture (only two comments per week). Post it on the google groups
- Why do we write?
  - Because writing is a way many good architects and systems designers force themselves to think (explaining clearly and thinking clearly are highly correlated!)

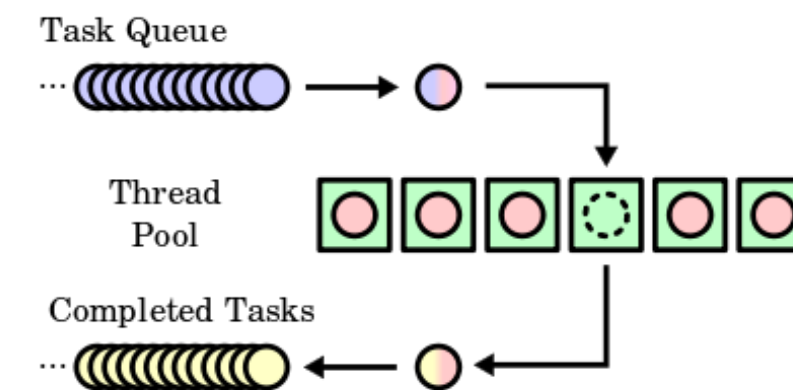
# What we are looking for in comments

- Try to explain the slide (as if you were trying to teach your classmate while studying for an exam)
  - “The instructor said this, but if you think about it this way instead it makes much more sense...”
- Explain what is confusing to you:
  - “What I’m totally confused by here was...”
- Challenge classmates with a question
  - for example, related to an assignment.
- Provide a link to an alternate explanation
  - “This site has a really good description of how multi-threading works...”
- Mention real-world examples
  - For example, describe all the parallel hardware components in the Xbox One
- Constructively respond to another student’s comment or question
  - “@segfault21, are you sure that is correct?....”
  - “@funkysenior19’s point is that the overhead of communication...”

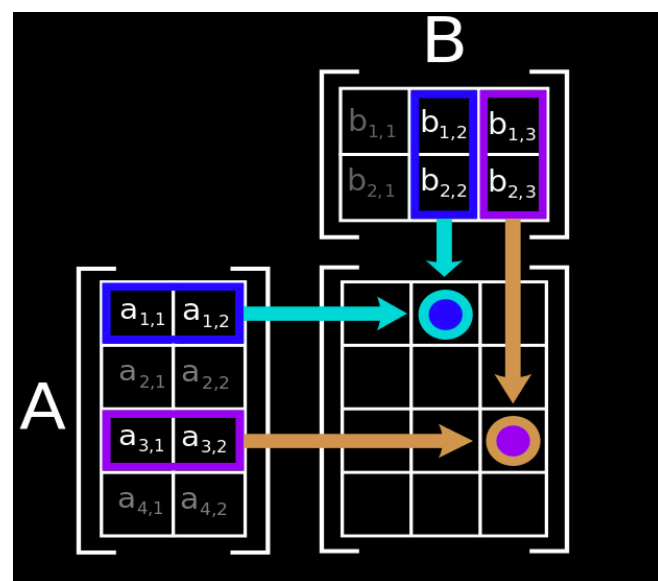
# Five/Six/Seven programming assignments



**Assignment 1: ISPC**  
programming on  
multi-core CPUs



**Assignment 2:**  
scheduler for a task  
graph



**Assignment 3:**  
parallel  
algorithms on  
linear algebra

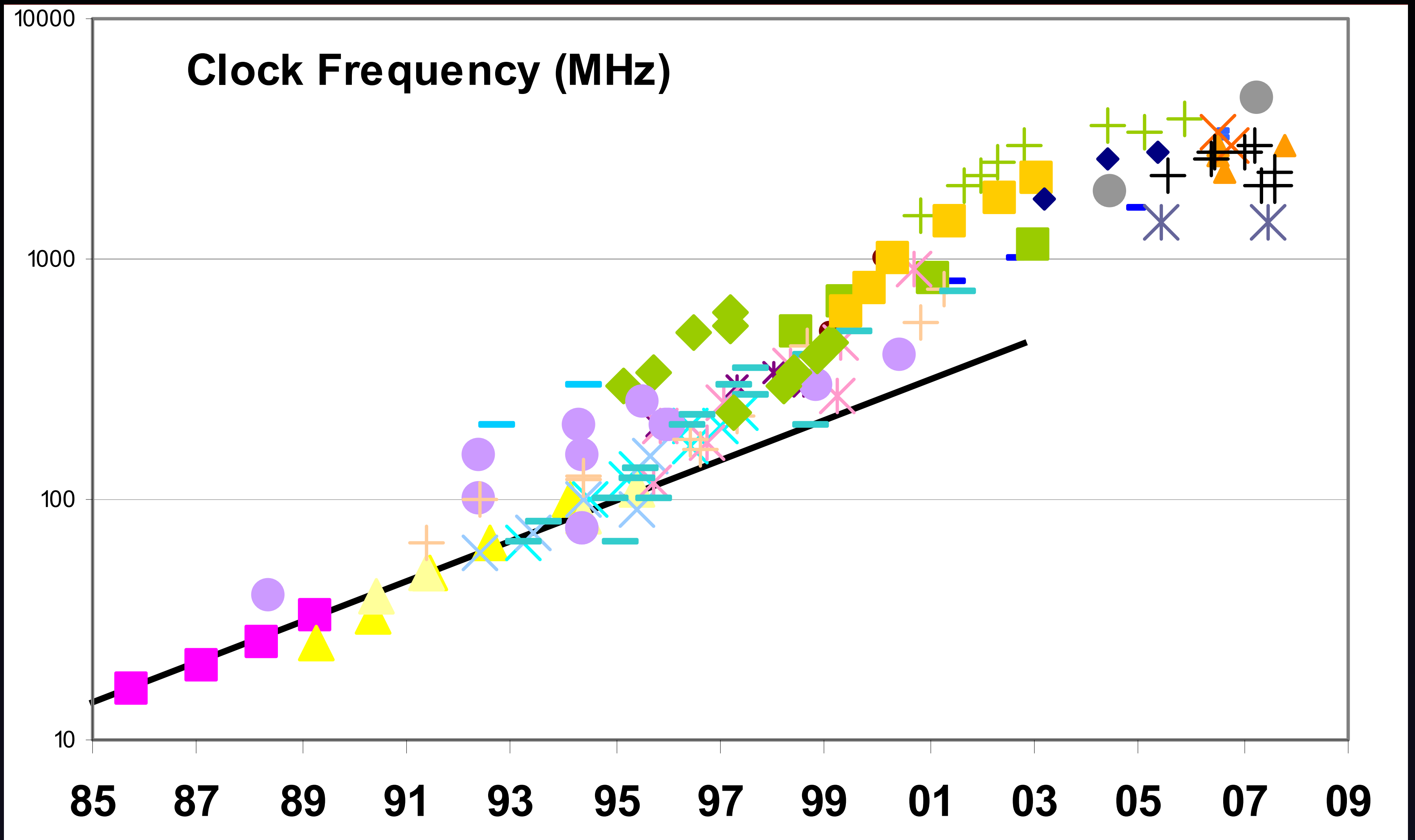


**Assignment 4:**  
Graph  
Processing

**Assignment 5**  
?

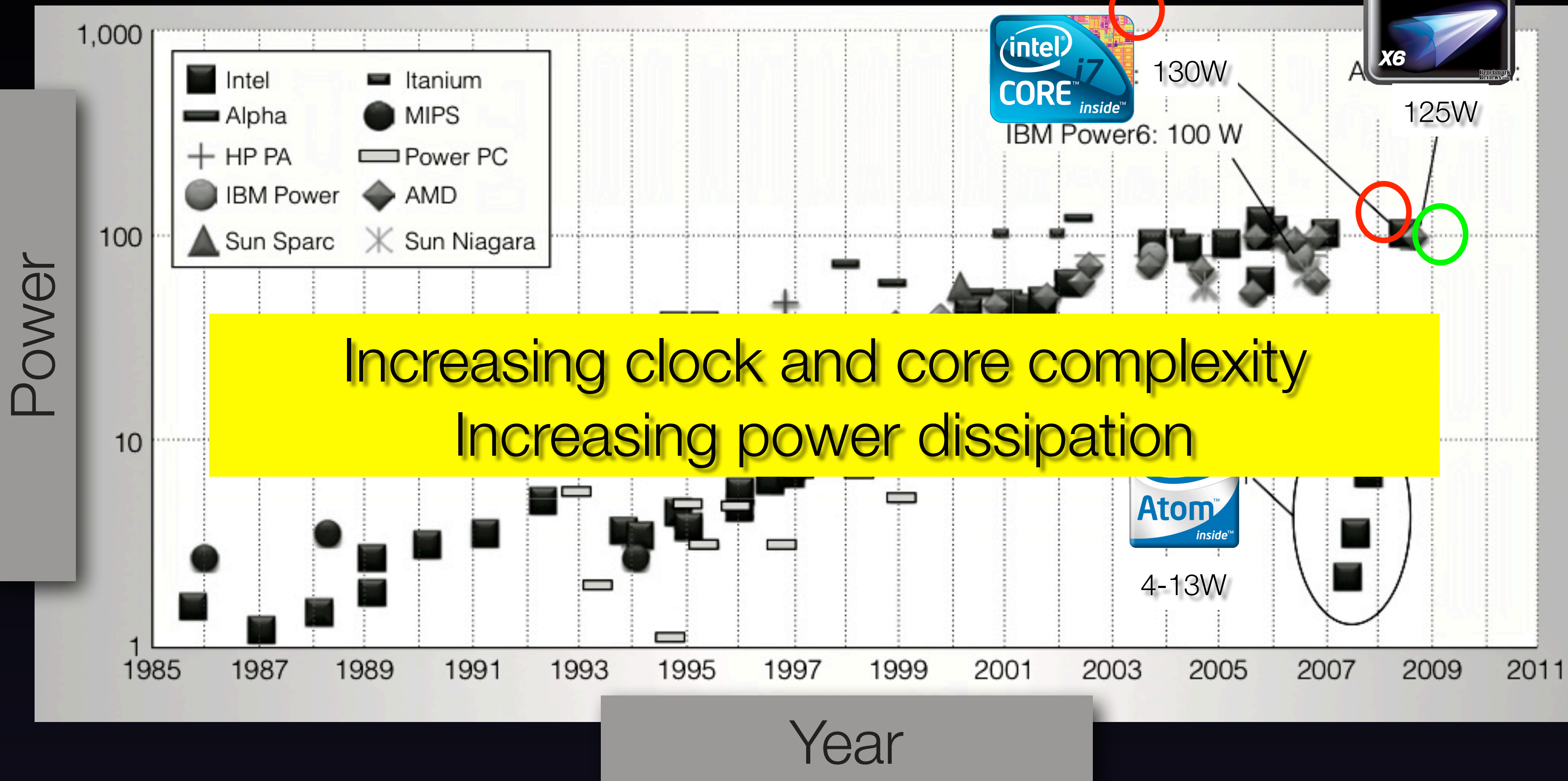
**Why parallel computers?**

# Clock Frequency





# CPUs consume a lot of power

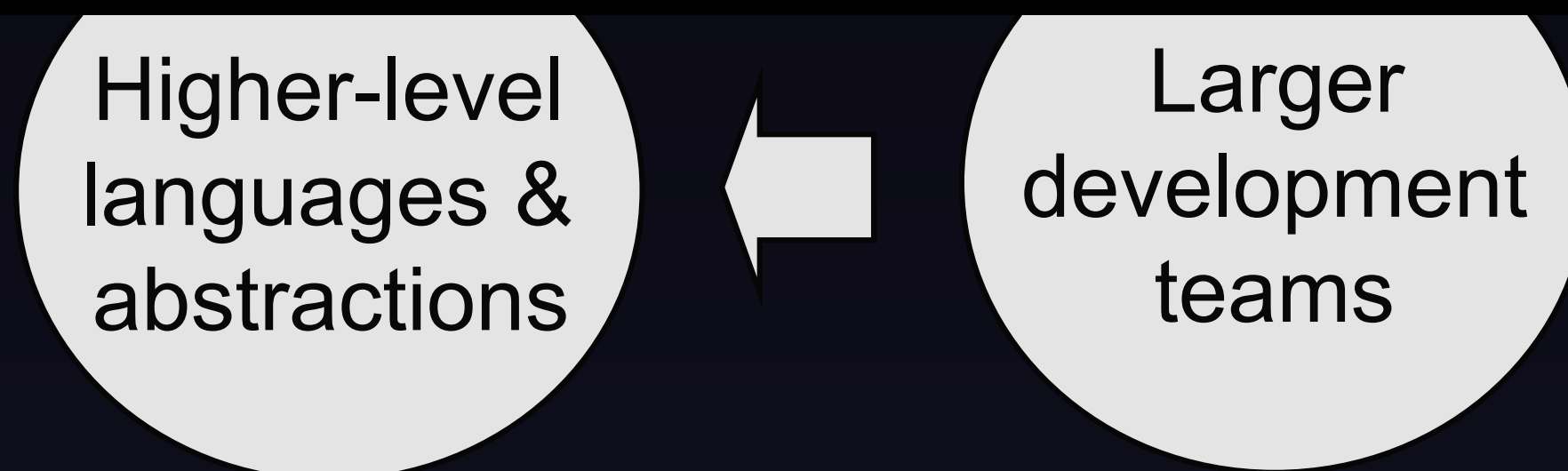




# Development cycle



Game Over  
Next: Multicore



# Multicore Revolution is here!

More cores on a chip

Each core ; 40% ↓ Ghz = 0.25x Power ↓

Overall Performance = 4 cores \* 0.6x/core = 2.4x

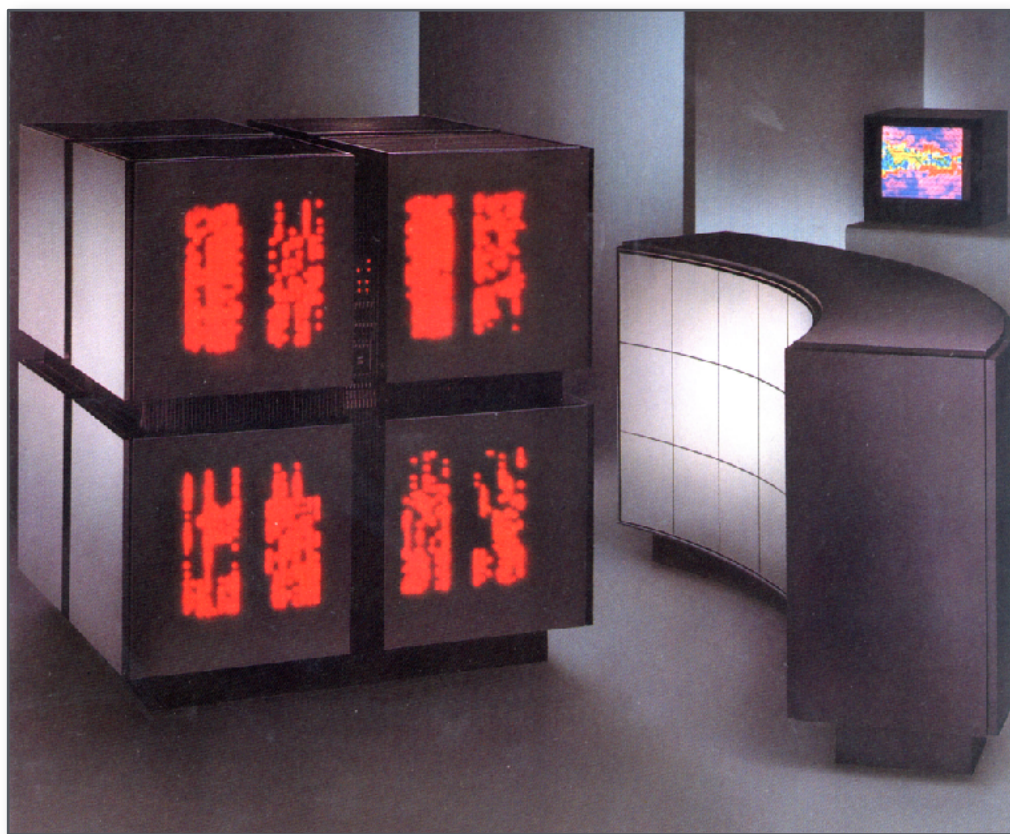




# Why parallel processing? (80's, 90's, early 2000's)

The answer until ~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



Sun Enterprise  
10000

(circa 1997)  
64 UltraSPARC-II  
processors

# 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
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...

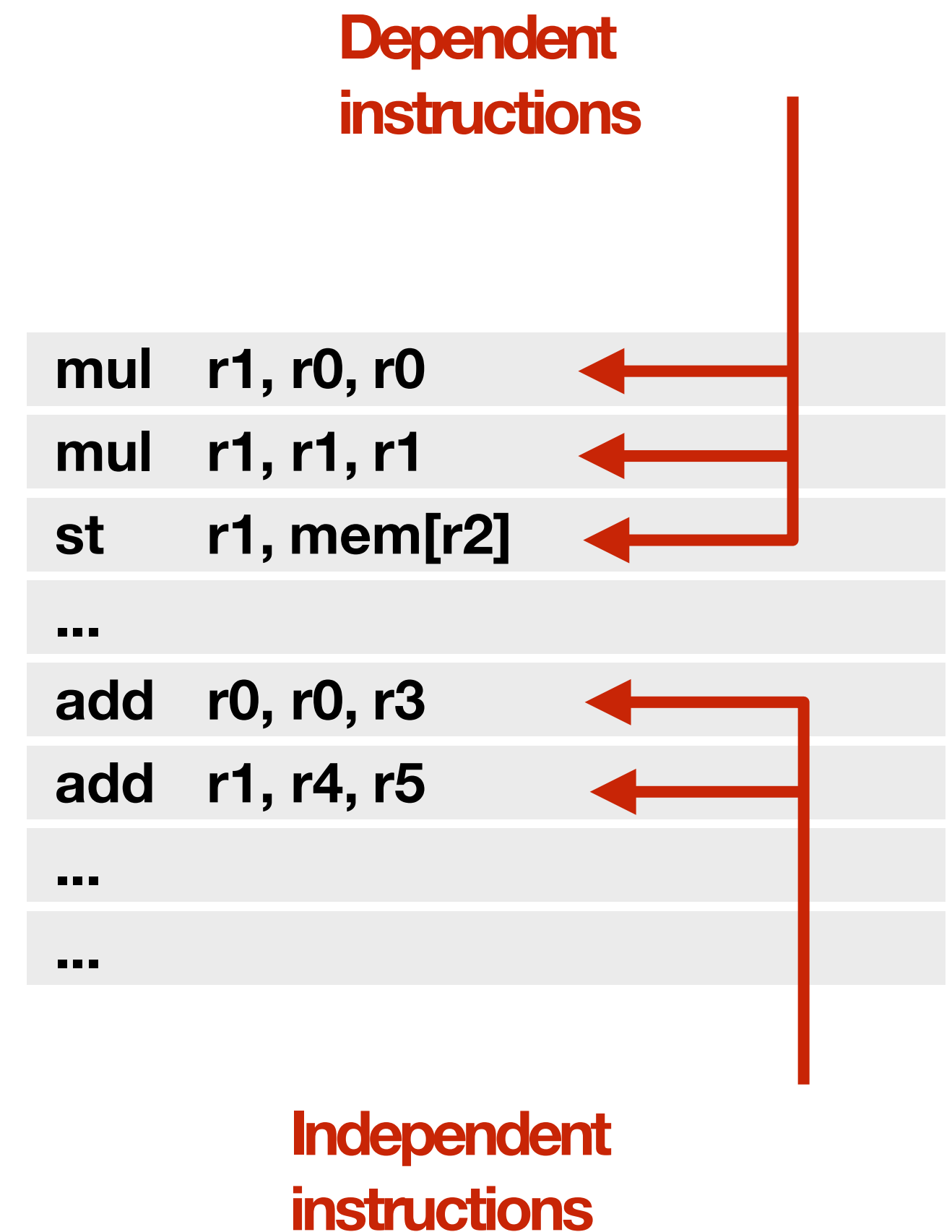


And so on...

```
_main:
100000f10: pushq
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
```

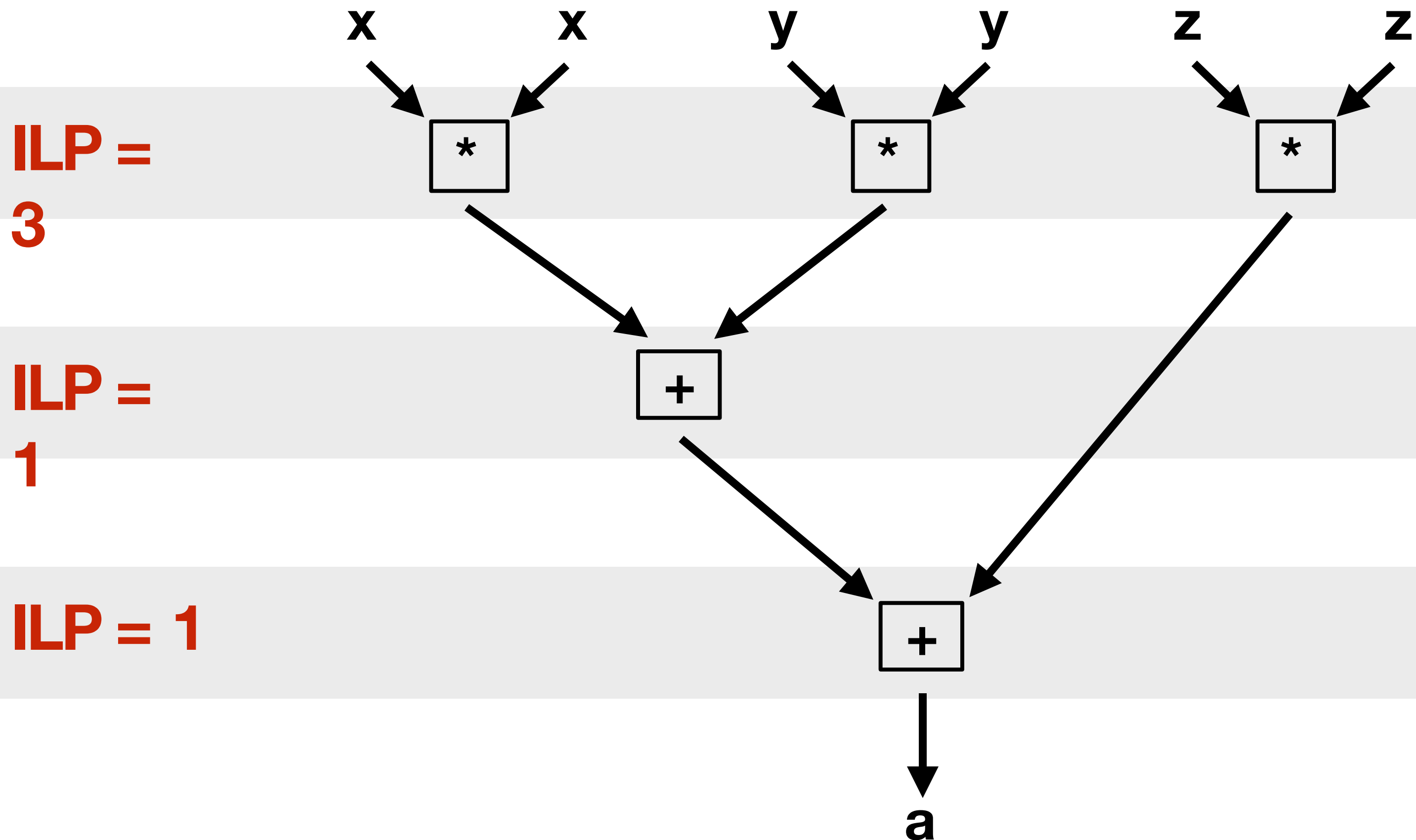
*// now r3 stores value of program variable 'a'*

**This program has five instructions, so it will take five clocks to execute, correct? Can we do better?**



# ILP example

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



# 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'

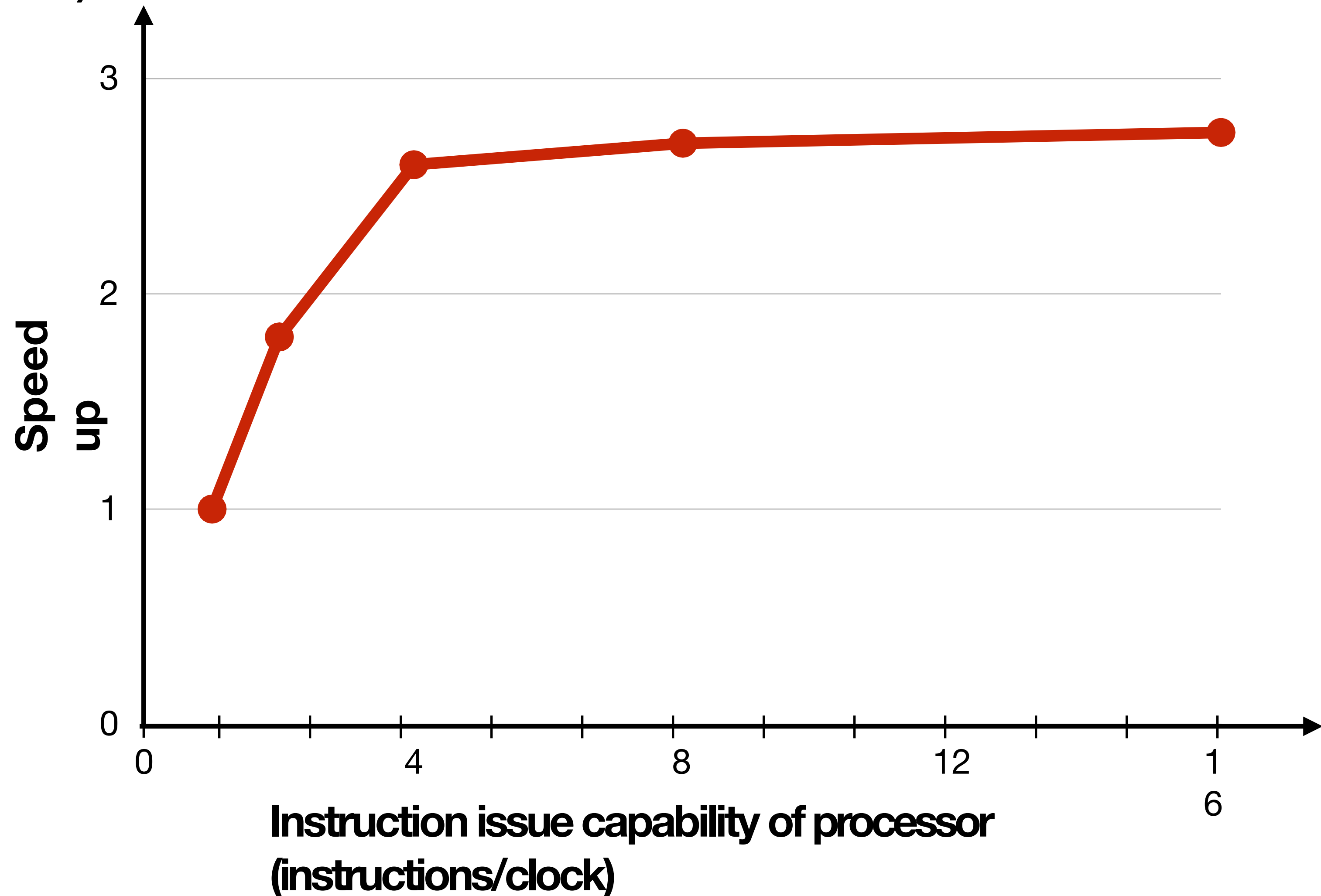
Superscalar execution: processor automatically finds independent instructions in an instruction sequence and executes them in parallel on multiple execution units!

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...

# Diminishing returns of superscalar execution

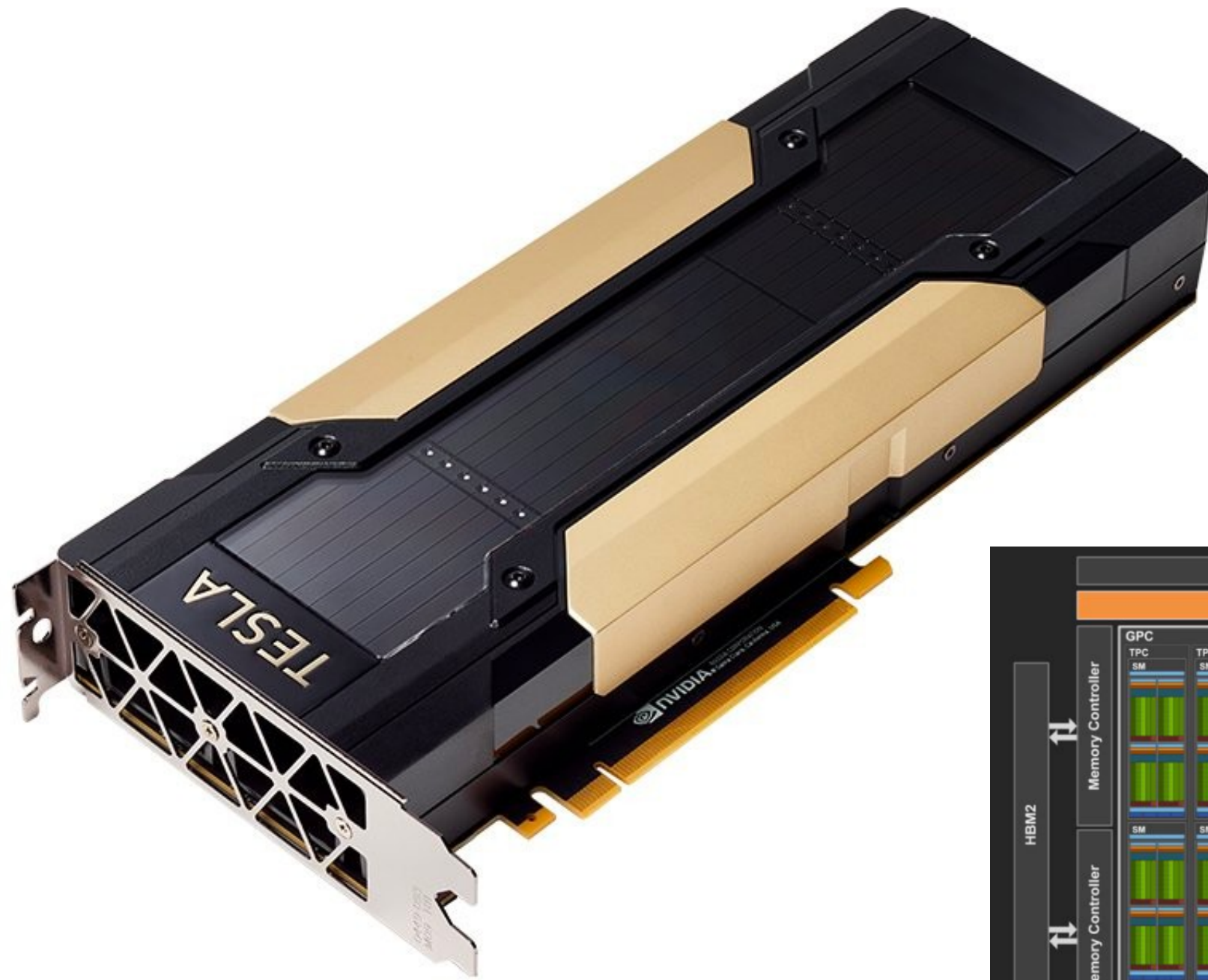
Most available ILP is exploited by a processor capable of issuing four instructions per clock (Little performance benefit from building a processor that can issue more)





# NVIDIA Tesla V100 GPU (2017)

5,376 fp32 units grouped into 84 major processing blocks



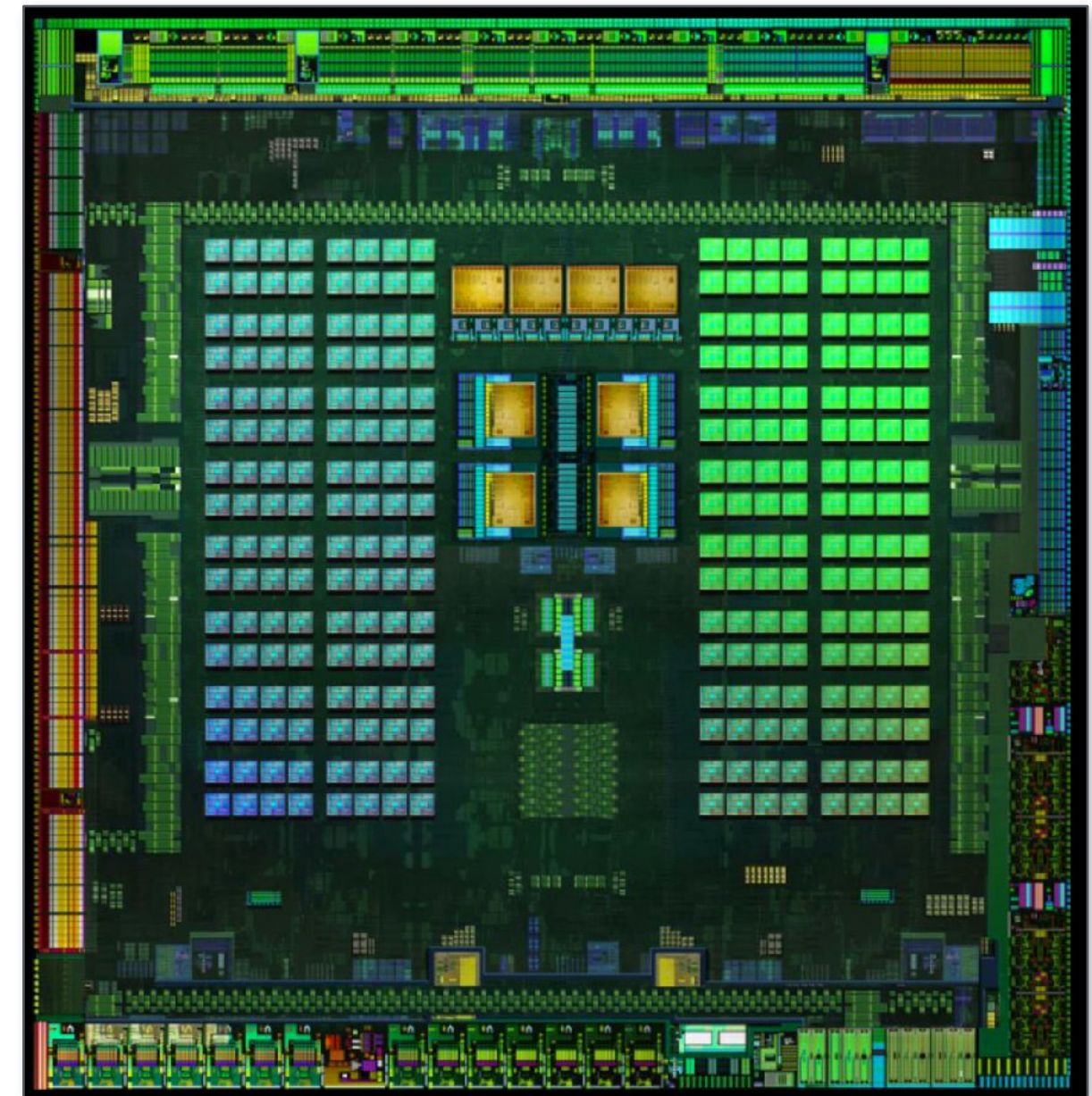


# Mobile parallel processing

**Power constraints heavily influence the design of mobile systems**



**Apple A11 Bionic: (in iPhone 8)**  
2 “big” CPU cores + 4 “small” CPU  
cores + Three “core” Apple-  
designed GPU+ Image  
processor +  
Neural Engine for DNN  
acceleration + Motion  
processor



**NVIDIA Tegra X1:**  
4 ARMA57 CPU  
cores + 4 ARM  
A53 CPU cores +  
NVIDIA GPU+ image  
processor...



# Parallel + specialized HW

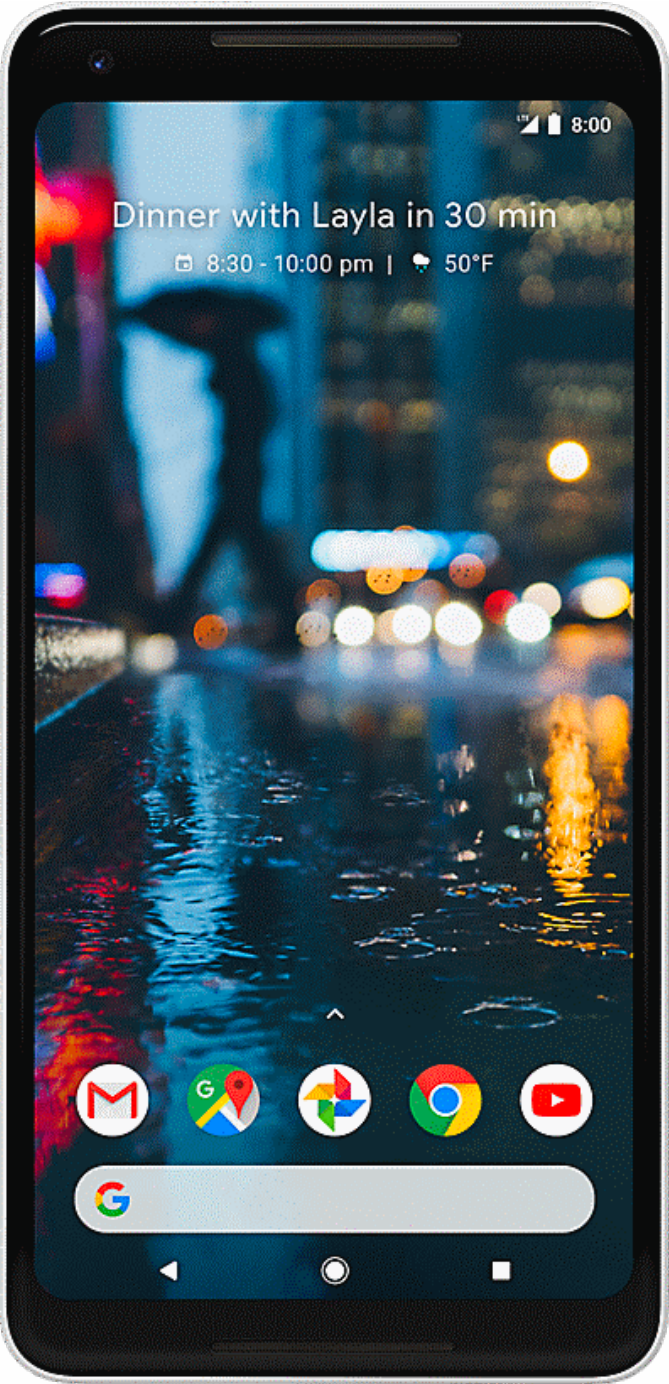
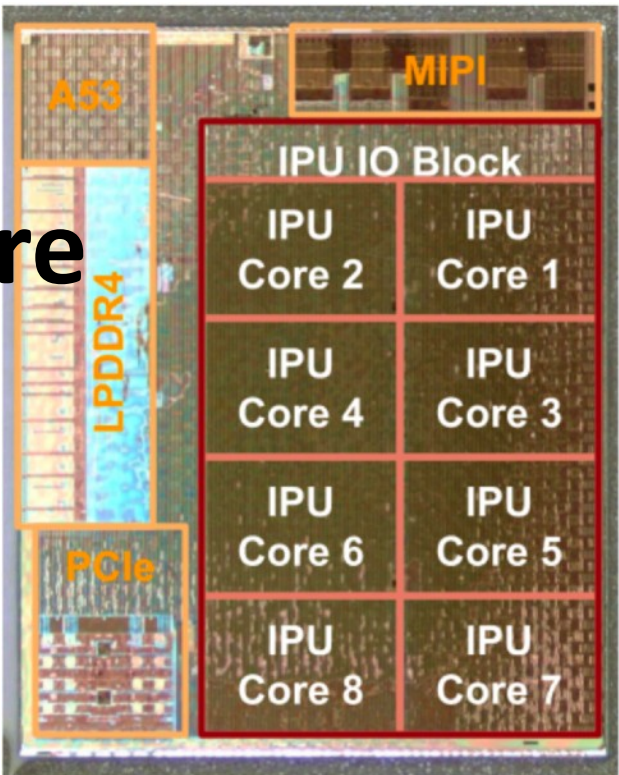
- Achieving high efficiency will be a key theme in this class
- We will discuss how modern systems are not only parallel, but also specialize processing to achieve high levels of power efficiency



# Let's crack open a modern smartphone

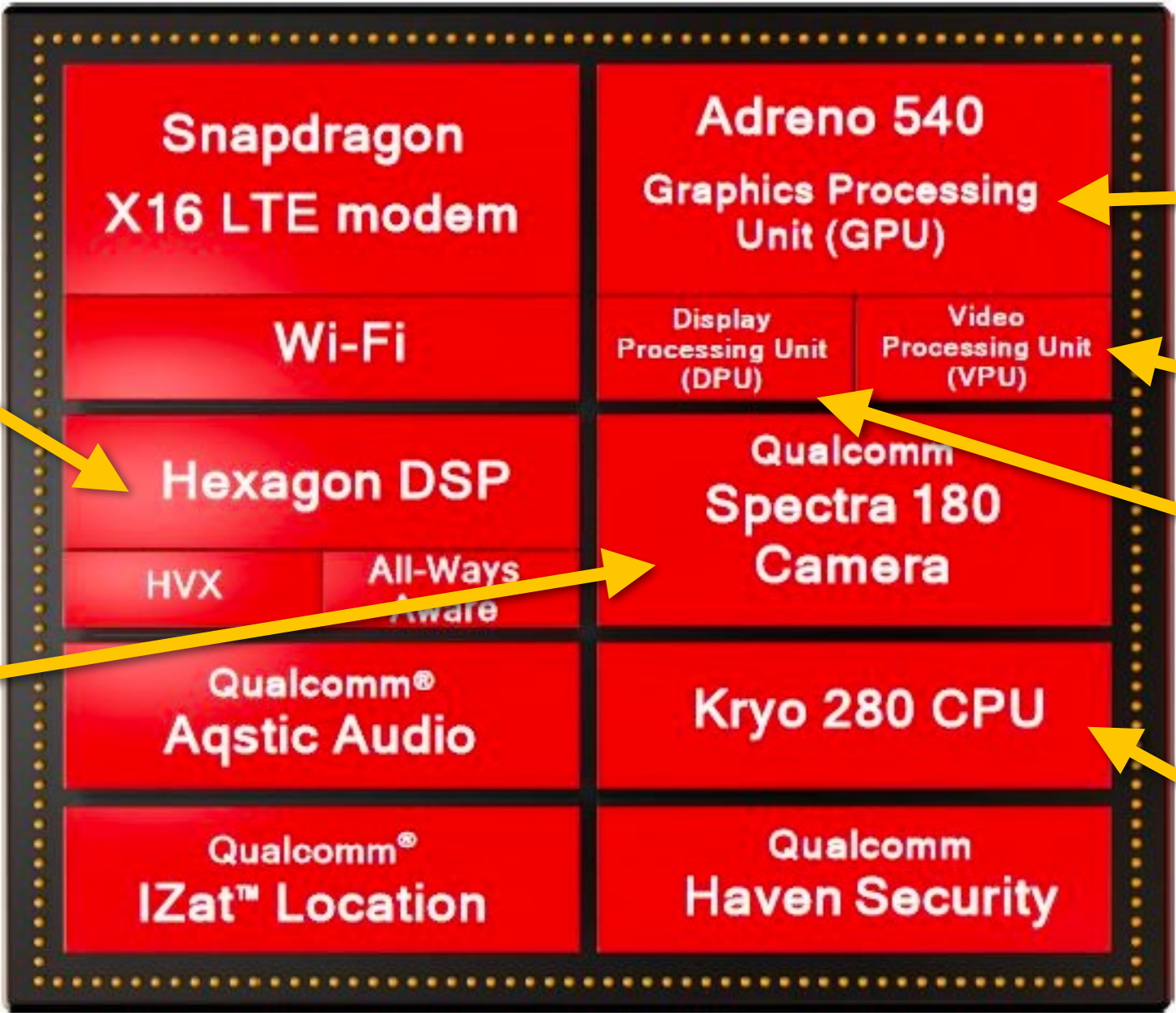
Google Pixel 2 Phone:  
Qualcomm Snapdragon 835 SoC + Google Visual  
Pixel Core

**Visual Pixel Core**  
Programmable  
image processor and  
DNN accelerator



**“Hexagon”  
Programmable  
DSP**  
data-parallel multi  
media processing

**Image Signal  
Processor**  
ASIC for processing  
camera sensor pixels



**Multi-core GPU**  
(3D graphics,  
OpenCL data-parallel  
compute)  
**Video encode/decode**  
**ASIC Display engine**  
(compresses pixels for  
transfer to high-res screen)  
**Multi-core ARM CPU**  
4 “big cores” + 4 “little cores”

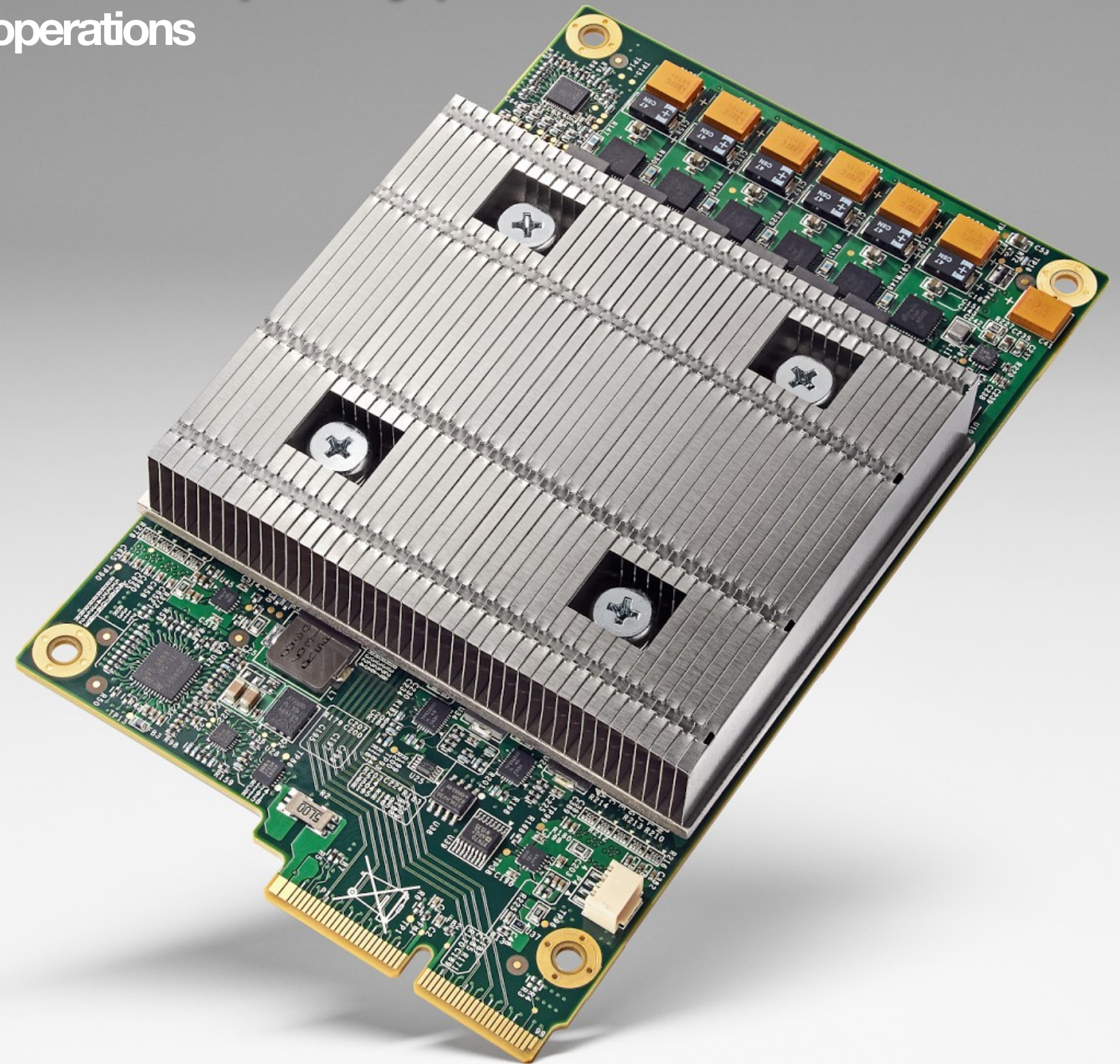


# Specialized processors for evaluating machine learning

Countless recent papers at top computer architecture research conferences on the topic of ASICs or accelerators for deep learning or evaluating deep networks...

- **Cambricon: an instruction set architecture for neural networks**, Liu et al. ISCA 2016
- **EIE: Efficient Inference Engine on Compressed Deep Neural Network**, Han et al. ISCA 2016
- **Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing**, Albericio et al. ISCA 2016
- **Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators**, Reagen et al. ISCA 2016
- **vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design**, Rhu et al. MICRO 2016
- **Fused-Layer CNN Architectures**, Alwani et al. MICRO 2016
- **Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Network**, Chen et al. ISCA 2016
- **PRIME: A Novel Processing-in-memory Architecture for Neural Network Computation in ReRAM-based Main Memory**, Chi et al. ISCA 2016
- **DNNWEAVER: From High-Level Deep Network Models to FPGA Acceleration**, Sharma et al. MICRO 2016

Example: Google's Tensor Processing Unit (TPU) Accelerates deep learning operations



**Intel Lake Crest ML  
accelerator  
(formerly Nervana)**





# 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!

# **This is a 400-level Course**

- **Ask Questions!**
- **We assume you are familiar with git, make, C/C++ programming.**
- **Be prepared to read header files and lookup APIs**
- **Good luck!**