

# Parallelism, Multicore, and Synchronization



# Parallelism & Synchronization

Multicore → more cores!

Cache Coherency

- Processors cache *shared* data → they see different (incoherent) values for the *same* memory location

Threads

- Mechanism to take advantage of parallelism

Synchronizing parallel programs

- Atomic Instructions
- HW support for synchronization

How to write parallel programs

- Threads and processes
- Critical sections, race conditions, and mutexes

# xkcd/619

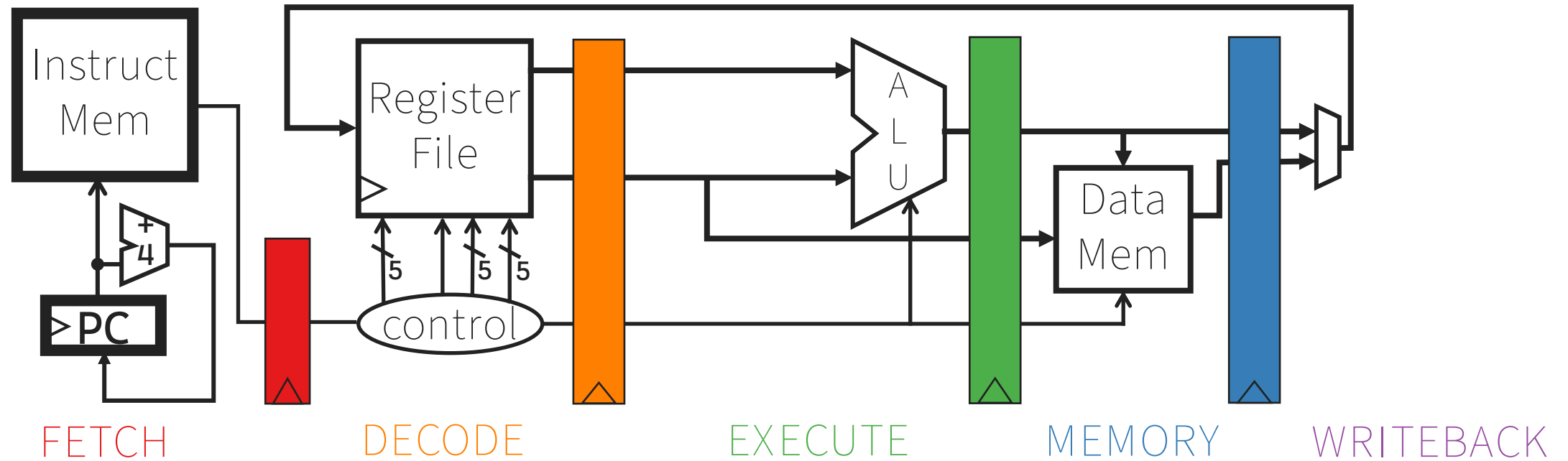
IT TOOK A LOT OF WORK, BUT THIS  
LATEST LINUX PATCH ENABLES SUPPORT  
FOR MACHINES WITH 4,096 CPUs,  
UP FROM THE OLD LIMIT OF 1,024.

/ DO YOU HAVE SUPPORT FOR SMOOTH  
FULL-SCREEN FLASH VIDEO YET?

NO, BUT WHO USES THAT?

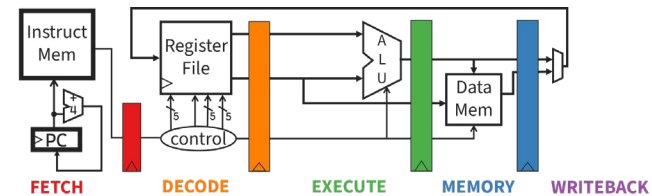
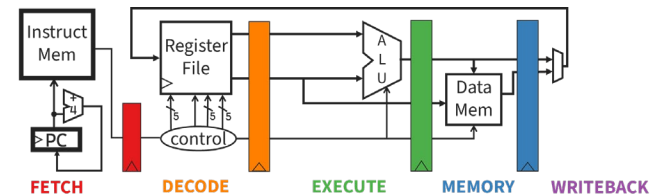
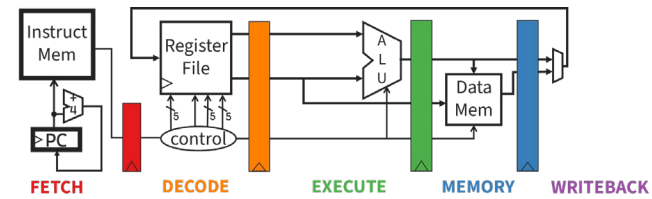
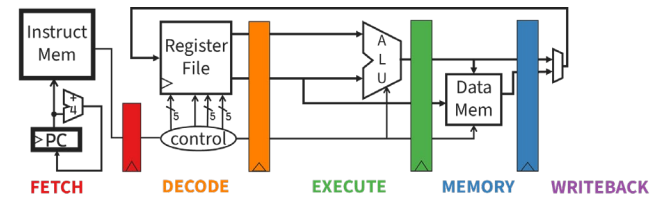


# Big Picture: Multicore and Parallelism



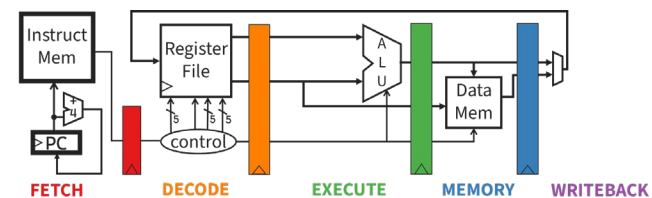
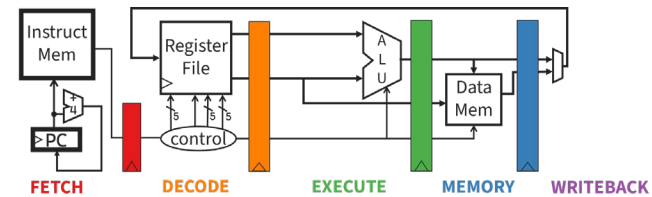
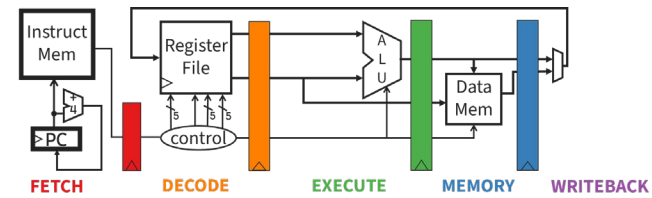
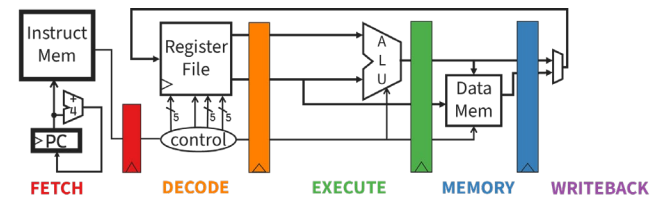
# Big Picture: Multicore and Parallelism

Why do I need *four* computing cores on my phone?!



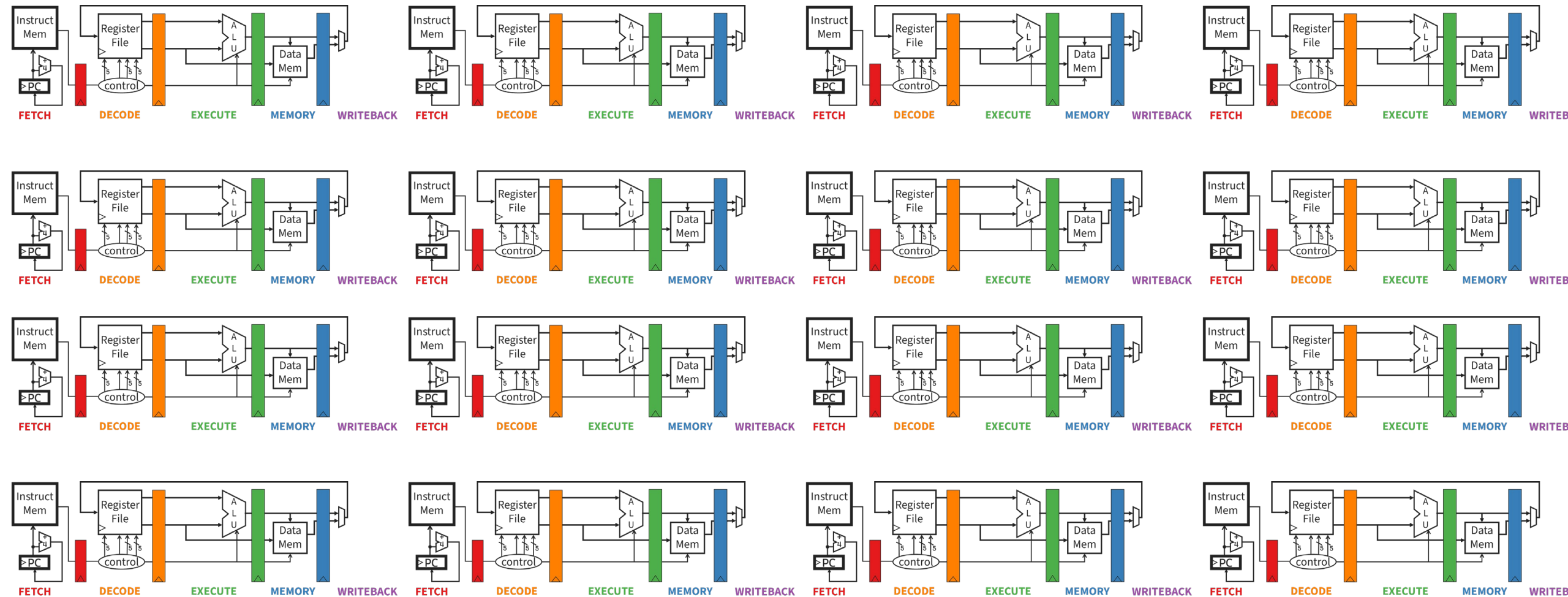
Why do I need *eight* computing cores on my phone?!

Why do I need *eight* computing cores on my phone?!



# Big Picture: Multicore and Parallelism

Why do I need *sixteen* computing cores on my phone?!



# Pitfall: Amdahl's Law

Execution time after improvement =

affected execution time

---

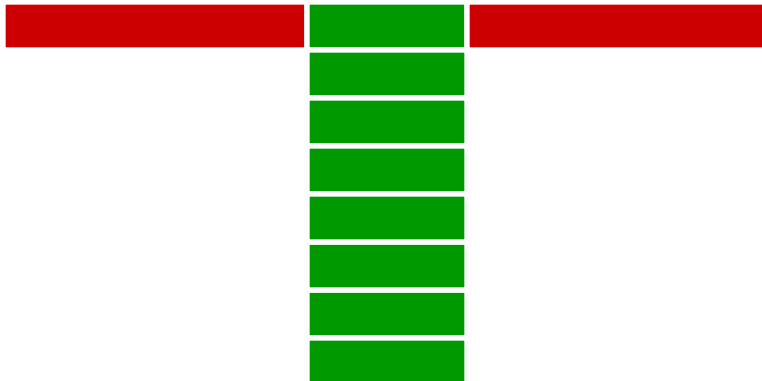
amount of improvement

+ execution time unaffected

$$T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvement factor}} + T_{\text{unaffected}}$$



# Amdahl's Law



# Scaling Example

Workload: sum of 10 scalars, and  $10 \times 10$  matrix sum

- Speed up from 10 to 100 processors?

Single processor: Time =  $(10 + 100) \times t_{\text{add}}$

10 processors

- Time =  $100/10 \times t_{\text{add}} + 10 \times t_{\text{add}} = 20 \times t_{\text{add}}$
- Speedup =  $110/20 = 5.5$

100 processors

- Time =  $100/100 \times t_{\text{add}} + 10 \times t_{\text{add}} = 11 \times t_{\text{add}}$
- Speedup =  $110/11 = 10$

Assumes load can be balanced across processors

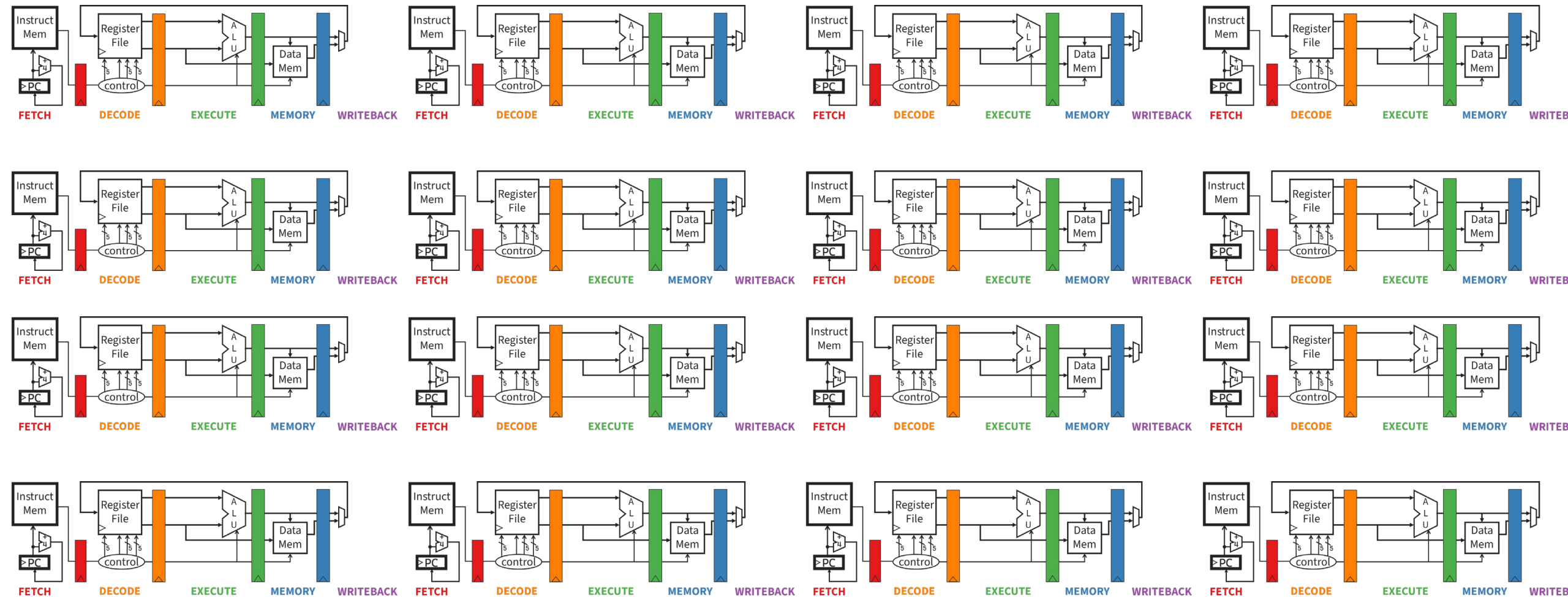
# Takeaway

Unfortunately, we cannot not obtain unlimited scaling (speedup) by adding unlimited parallel resources, eventual performance is dominated by a component needing to be executed sequentially.

Amdahl's Law is a caution about this diminishing return

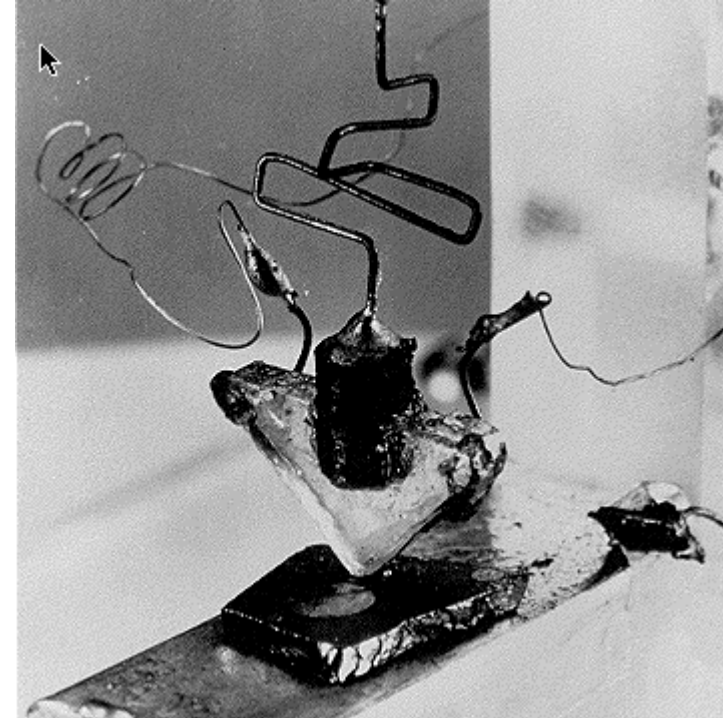
# Big Picture: Multicore and Parallelism

Why do I need *sixteen* computing cores on my phone?!



# Answer Part I: Moore's Law

- Electrical Switch
  - On/Off
  - Binary
- Transistor



The first transistor on a workbench at AT&T Bell Labs in 1947

# Moore's Law

- 1965
  - # of transistors integrated on a die doubles every 18-24 months (*i.e.*, grows exponentially with time)
- Amazingly visionary
  - 2300 transistors, 1 MHz clock (Intel 4004) - 1971
  - 16 Million transistors (Ultra Sparc III)
  - 42 Million transistors, 2 GHz clock (Intel Xeon) – 2001
  - 55 Million transistors, 3 GHz, 130nm technology, 250mm<sup>2</sup> die (Intel Pentium 4) – 2004
  - 290+ Million transistors, 3 GHz (Intel Core 2 Duo) – 2007
  - 721 Million transistors, 2 GHz (Nehalem) - 2009
  - 1.4 Billion transistors, 3.4 GHz Intel Haswell (Quad core) – 2013
  - 7.2 Billion transistors, 3-3.9 GHz Intel Broadwell (22-core) – 2016
  - 20 Billion transistors, 3.49 GHz Apple M2 (8 core) — 2022
  - 28 Billion transistors, 4.4 GHz Apple M4 (16x core) — 2024





# Why Multicore?

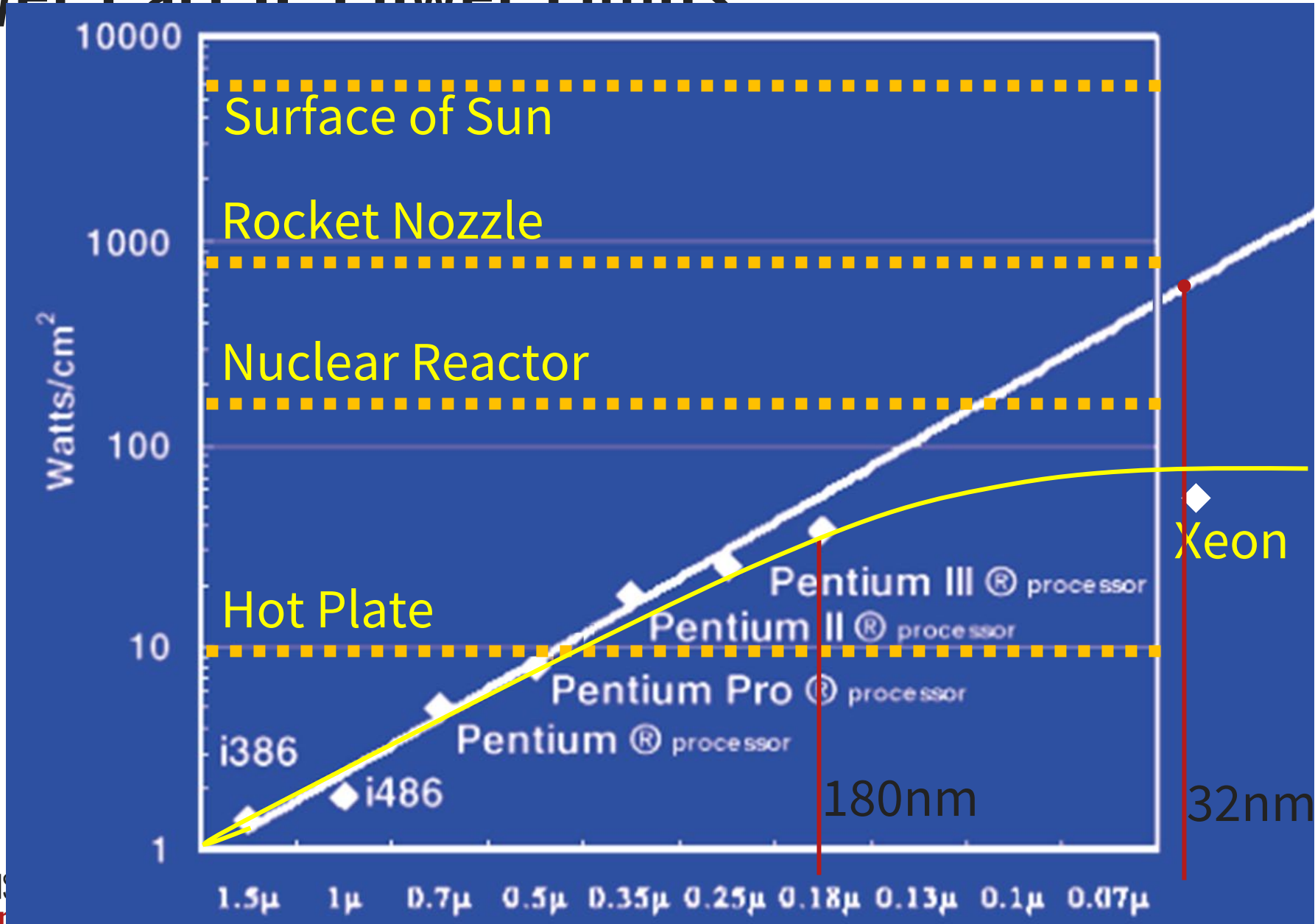
## Moore's law

- A law about transistors
- Smaller means more transistors per die
- And smaller means faster too

But: Power consumption growing too...



# Answer Part II- Power Limits



# Answer Part II: Power Limits

Power = capacitance \* voltage<sup>2</sup> \* frequency

In practice: Power ~ voltage<sup>3</sup>

Lower Frequency

Reducing voltage helps (a lot)

... so does reducing clock speed

Better cooling helps

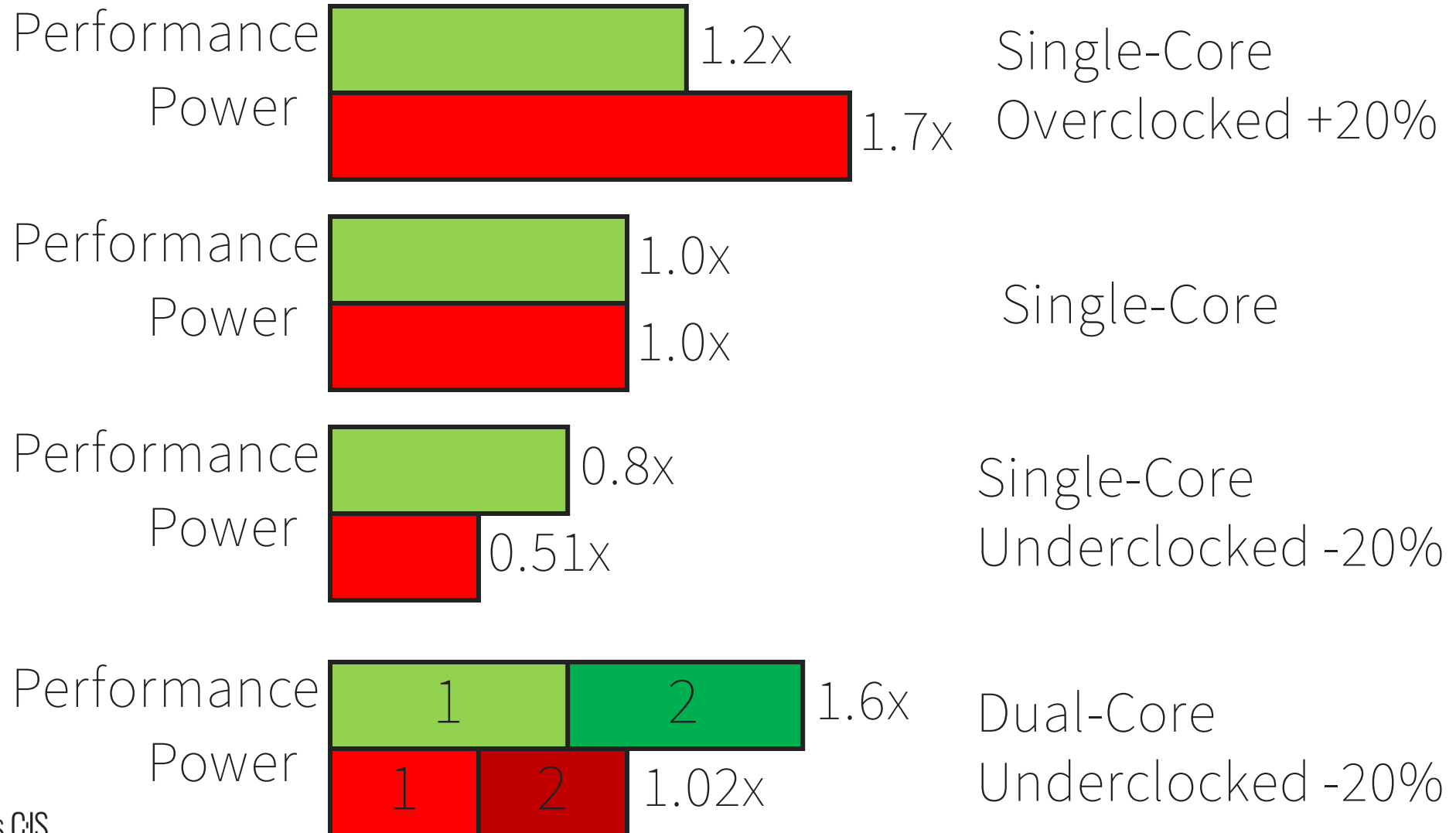
The power wall

- We can't reduce voltage further
- We can't remove more heat
- See the end of Dennard scaling:

[https://en.wikipedia.org/wiki/Dennard\\_scaling#Breakdown\\_of\\_Dennard\\_scaling\\_around\\_2006](https://en.wikipedia.org/wiki/Dennard_scaling#Breakdown_of_Dennard_scaling_around_2006)



# Why Multicore?



# Next

So, How do we get performance, especially with an increasing number of transistors?

# Performance Improvement 101

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$



# Performance Improvement 101

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

Examples:

Single-cycle → Multi-cycle

↓ Clock period, ↑ CPI

CISC → RISC

↑ insn count, ↓ CPI, ↓ Clock period

Pipelining

↓ Clock period, ↓ CPI

# Increasing Clock Frequencies

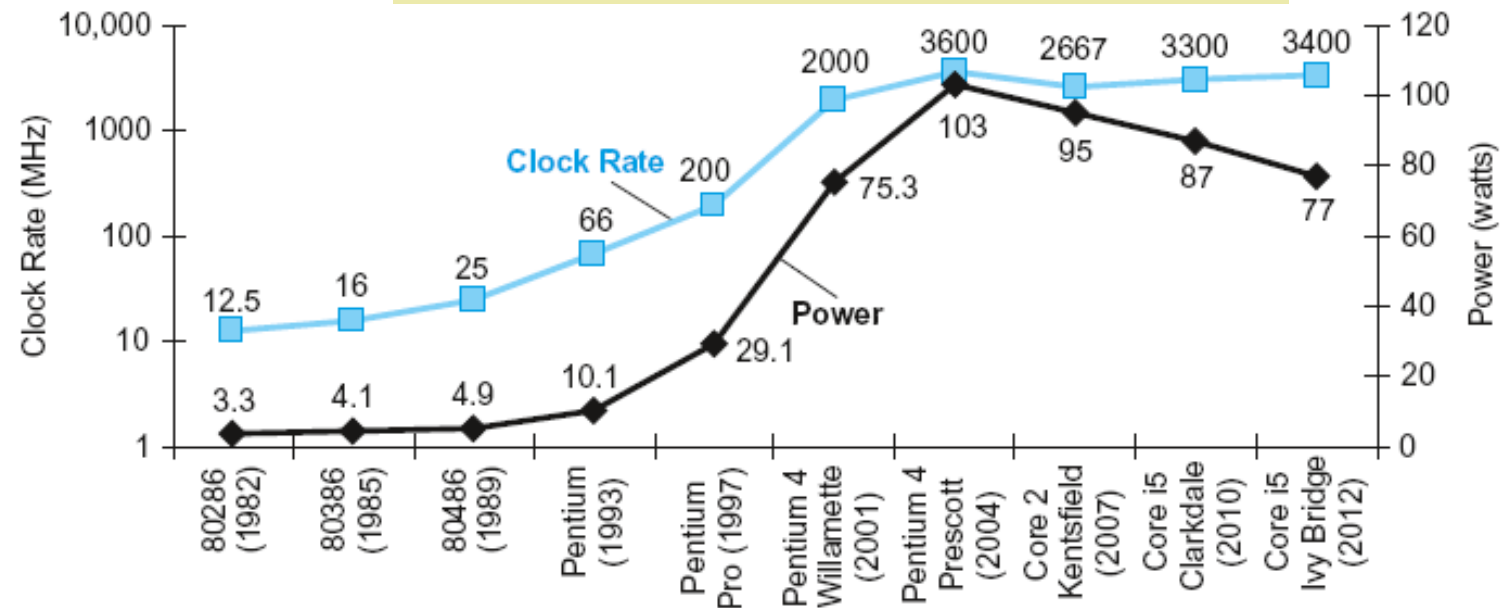
Darling of performance improvement for *decades*

Why is this no longer the strategy?

Hitting Frequency Limits:

- Heat
- Power

$$\text{Power} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$$



# Increasing Clock Frequencies

Darling of performance improvement for *decades*

Why is this no longer the strategy?

## Hitting Frequency Limits:

- Heat
- Power
- Pipeline depth
- Technology Scaling
  - [Intel retires “tick-tock” development model](#) (2016)
  - [Intel hits 10nm goals and signals a shift away from traditional CPUs](#) (2019)





# Improving CPI/IPC via ILP

*You've seen:*

Exploiting Intra-instruction parallelism:

Pipelining (decode A while fetching B)

*You haven't seen:*

Exploiting Instruction Level Parallelism (ILP):

- Multiple issue (2-wide, 4-wide, etc.)
- Dynamic Scheduling (OoO)

# Static Multiple Issue

a.k.a. [Very Long Instruction Word \(VLIW\)](#)

Compiler pairs instructions

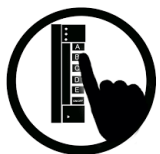
Example: Static Dual-Issue 32-bit RISC-V, 64-bit pairs

1. ALU/Branch instruction (or nop)
2. Load/Store instruction (or nop)

How does HW detect and resolve hazards?

It doesn't. 😊 Compiler must avoid hazards

Insn Addr	Insn type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB



# Scheduling Example

```
Loop: lw    t0, 0(s1)      # t0=array element
      add   t0, t0, s2     # add scalar in s2
      sw    t0, 0(s1)     # store result
      addi  s1, s1, -4     # decrement pointer
      bne   s1, zero, Loop # branch s1!=0
```

	ALU/branch	Load/store	cycle
Loop:	nop	lw t0, 0(s1)	1
	addi s1, s1, -4	nop	2
	add t0, t0, s2	nop	3
	bne s1, zero, Loop	sw t0, 4(s1)	4

# Improving IPC via ILP

Exploiting Intra-instruction parallelism:

Pipelining (decode A while fetching B)

Exploiting Instruction Level Parallelism (ILP):

- Multiple issue (2-wide, 4-wide, *etc.*)
- Dynamic Scheduling (OoO)

# Dynamic Scheduling

## Speculation/Out-of-order Execution

- Fetch many instructions
- Execute the ones whose inputs are ready
- Guess results of branches, loads, etc.
- Roll back/Flush/Zap if guesses were wrong
- Don't commit results until all previous insns committed
  - fetch in order, execute OoO, commit in order



# Effectiveness of OoO Superscalar

Kept improving performance... until it stopped  
2-wide? Yes please! 4-wide? Also great. 8? 16?

Limiting factors:

- Program dependencies
- Memory dependence detection → be conservative
  - e.g. Pointer Aliasing: `A[0] += 1; B[0] *= 2;`
- Hard to expose parallelism
  - Still limited by the fetch stream of the static program
- Structural limits
  - Memory delays and limited bandwidth
- Hard to keep pipelines full, especially with branches

# Improving IPC via ~~ILP~~ TLP

ILP reaching its limits...

Look for parallelism at a different granularity

Introducing: Thread-Level parallelism

Threads are separate tasks within the same process

Threads can run:

- On separate cores
- Taking turns on one core
- On one core at the same time (hyperthreading)

# What is a thread?

Process: multiple threads, code, data and OS state

Threads: concurrent computations that share the same address space

- **Share:** code, data, files
- **Do not share:** registers or stack



# Threads vs Processes

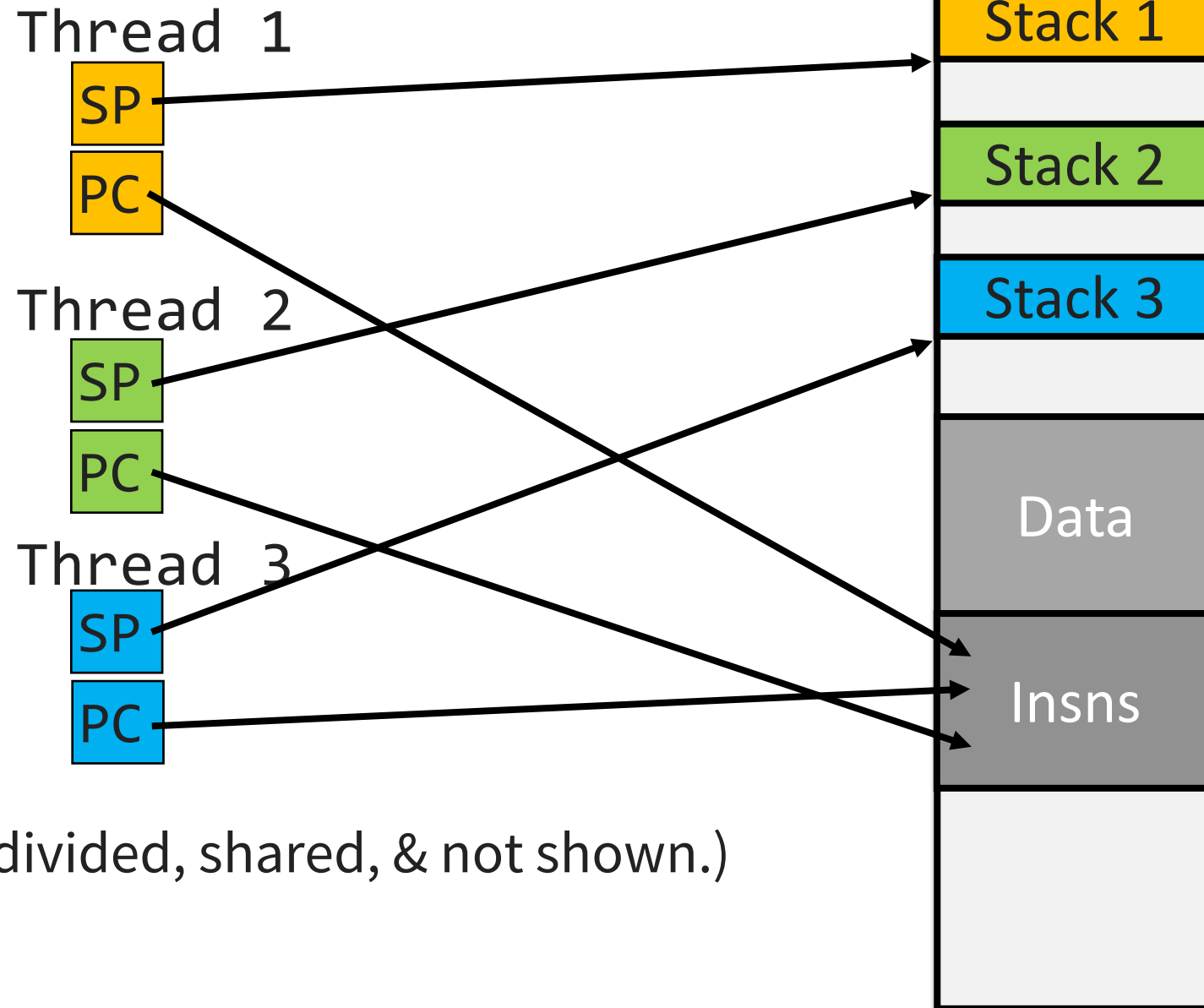
## Threads

- create
- join
- Defined by
  - Stack, SP, PC, registers
- Multiple threads share
  - address space (text, data, heap) , files

## Processes

- fork (and exec)
- wait
- Defined by
  - SP, PC, registers, address space

# Thread Memory Layout



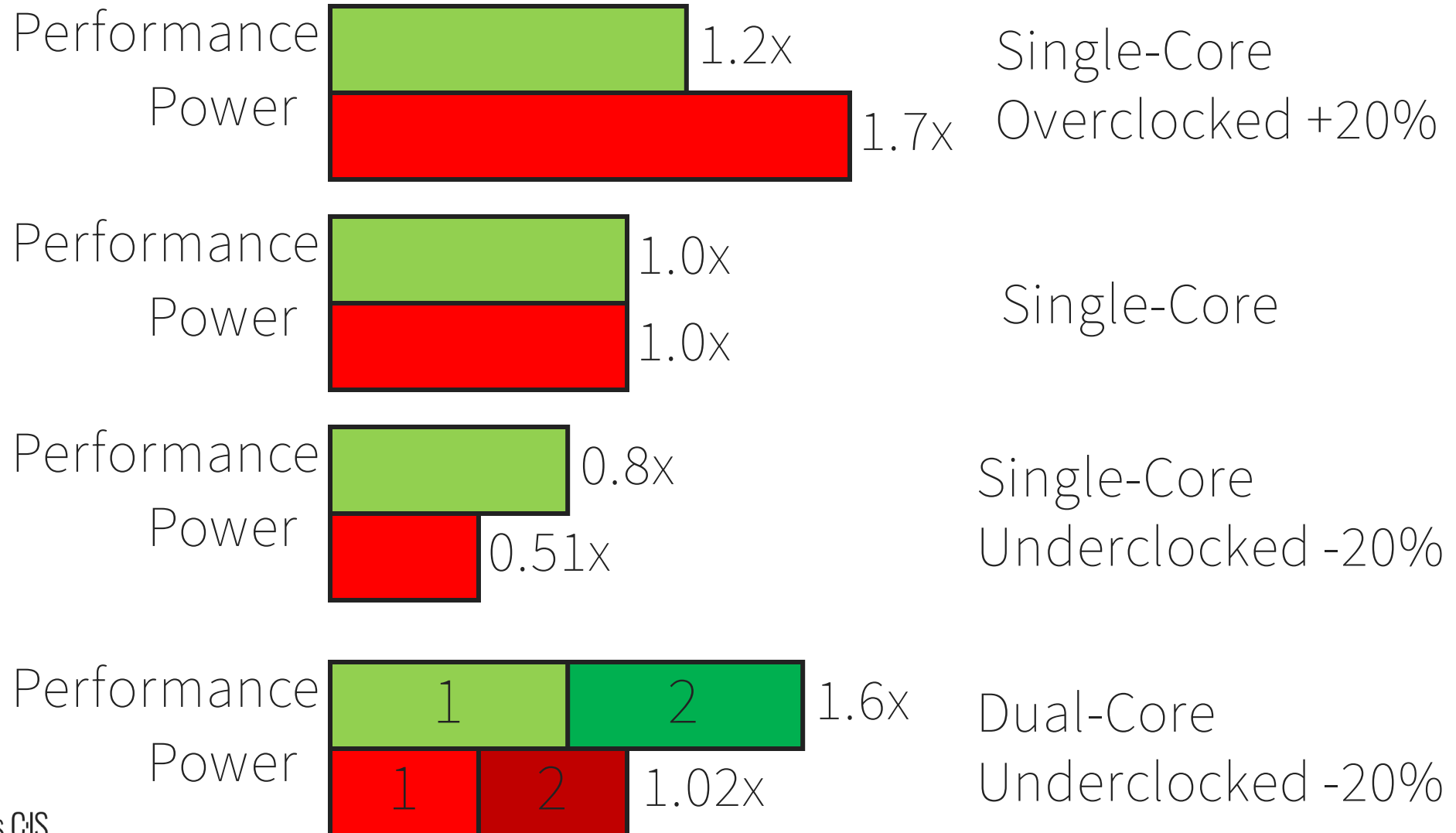
(Heap subdivided, shared, & not shown.)

# Power Efficiency

CPU	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
UltraSparc III	2003	1950MHz	14	4	No	1	90W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W

*Those simpler cores did something very right.*

# Why Multicore?



# Power Efficiency

CPU	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
UltraSparc III	2003	1950MHz	14	4	No	1	90W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4	Yes	2	75W
Core i5 Nehal	2010	3300MHz	14	4	Yes	1	87W
Core i5 Ivy Br	2012	3400MHz	14	4	Yes	8	77W
UltraSparc T1	2005	1200MHz	6	1	No	8	70W

*Those simpler cores did something very right.*



# Parallel Programming

So lets just all use multicore from now on!

... but software must be written as parallel program

Multicore difficulties

- Partitioning work
- Coordination & synchronization
- Communications overhead
- How do you write parallel programs?  
... without knowing exact underlying architecture?

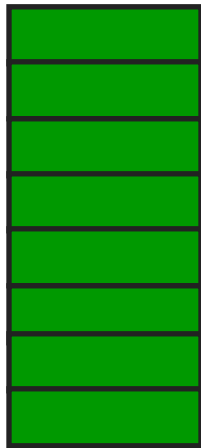


# Work Partitioning

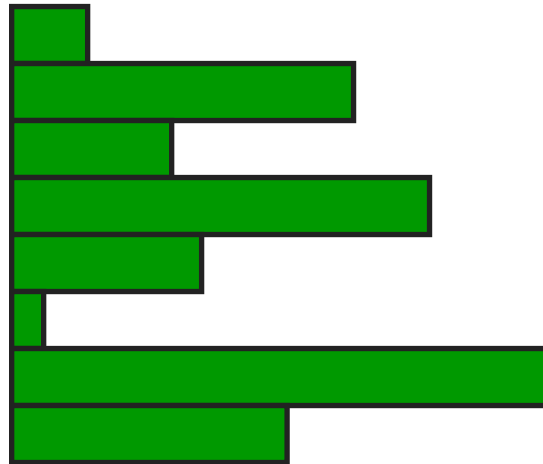
Partition work so all cores have something to do



Want:



Might Get:



And that's if the parts are parallelizable

# Amdahl's Law

If tasks have a **serial part** and a **parallel part**...

Example:

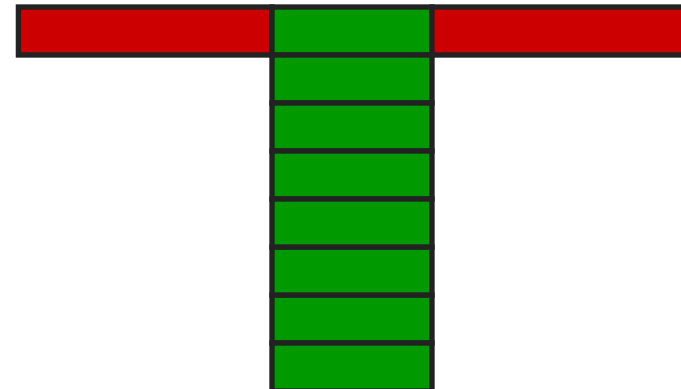


1. divide input data into  $n$  pieces
2. do work on each piece
3. combine all results



Amdahl's Law. As # of cores increases ...

- time to execute **parallel part**? *goes to zero*
- time to execute **serial part**? *remains the same*
- *Serial part eventually dominates*





# Parallel Programming

So lets just all use multicore from now on!

... but software must be written as parallel program

Multicore difficulties

- Partitioning work
- Coordination & synchronization
- Communications overhead
- How do you write parallel programs?  
... without knowing exact underlying architecture?



# Big Picture: Parallelism and Synchronization

How do I take advantage of *parallelism*?

How do I write (correct) parallel programs?

What primitives do I need to implement correct parallel programs?