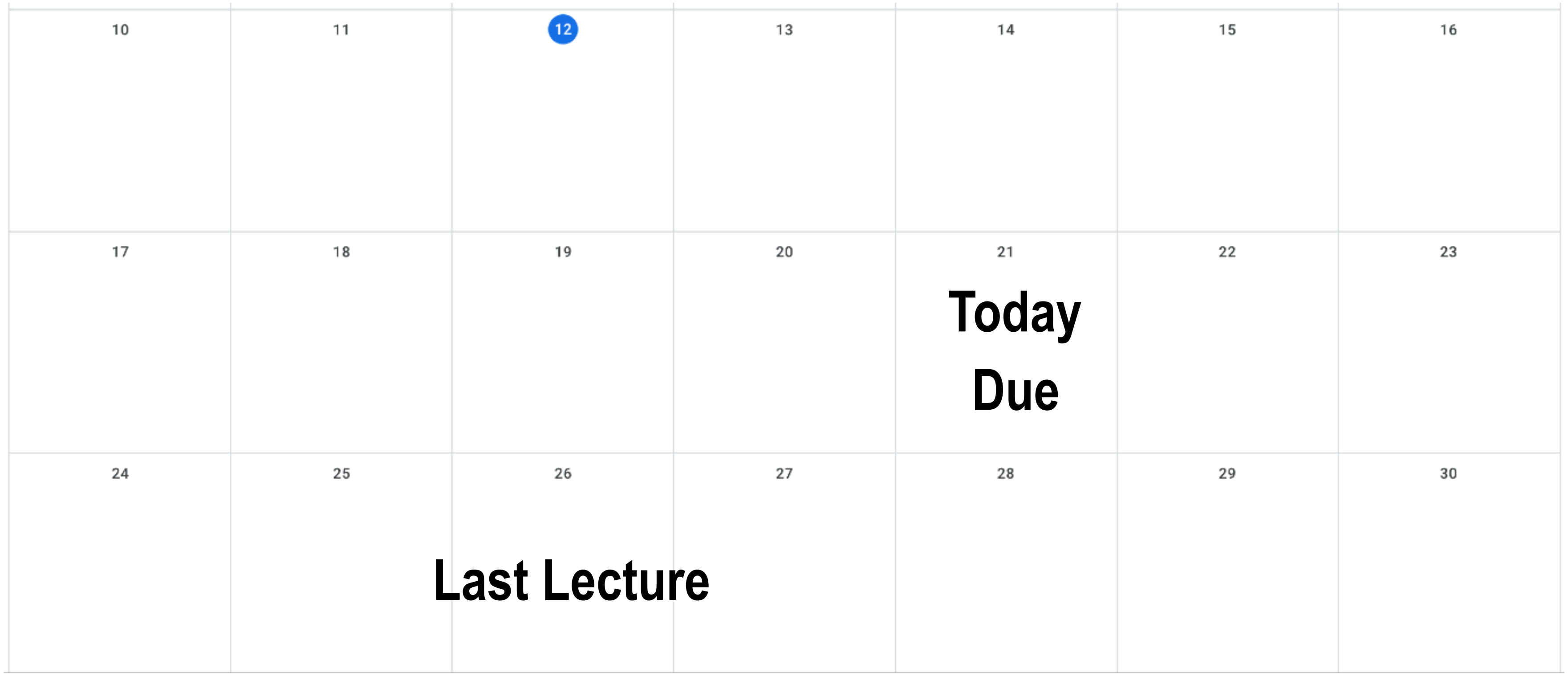
CSC 252: Computer Organization
Spring 2022: Lecture 26

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Announcements

- Assignment 5 due April 21.
- Will release assignment 4 grades soon.



10	11	12	13	14	15	16
17	18	19	20	21 Today Due	22	23
24	25	26 Last Lecture	27	28	29	30

Fine-Grained Switching

- One big bonus of fine-grained switching: no need for branch predictor!!

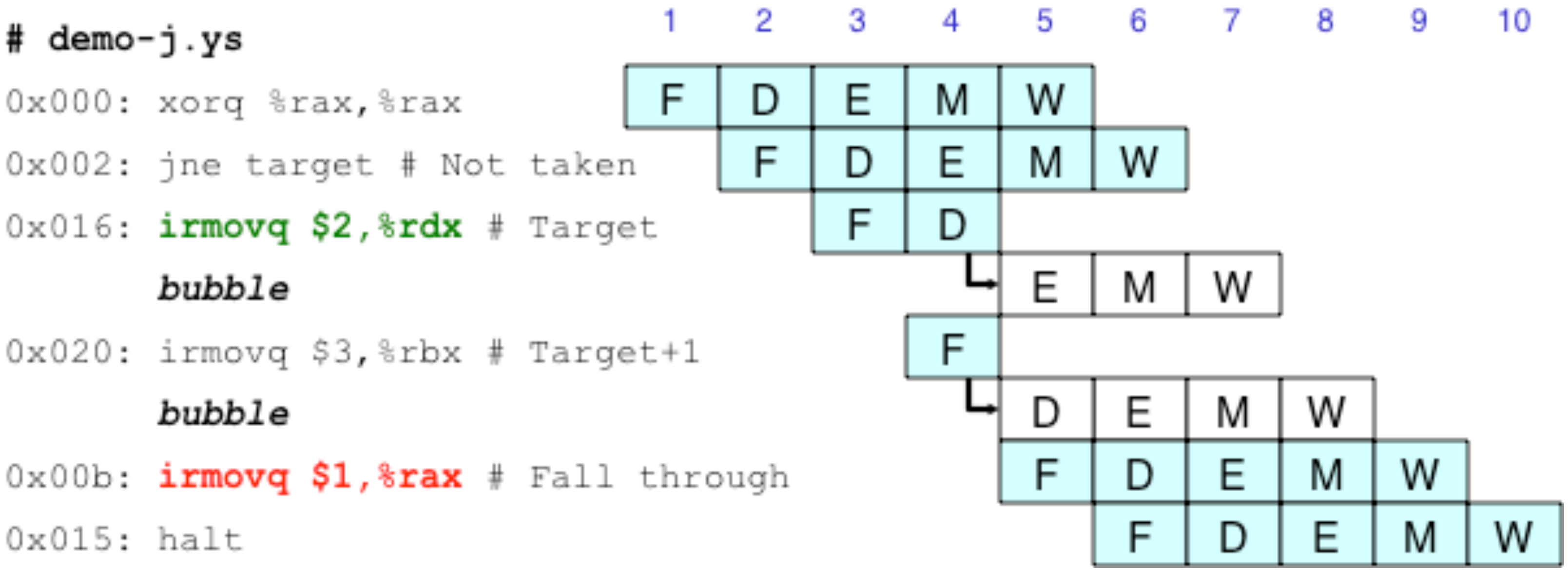
The stalling approach

			1	2	3	4	5	6	7	8	9
	`xorg %rax, %rax`		F	D	E	M	W				
	`jne L1`	`# Not taken`		F	D	E	M	W			
	Stall				Stall						
	Stall					Stall					
	`irmovq $1, %rax`	`# Fall Through`					F	D	E	M	W
`L1`	`irmovq $4, %rcx`	`# Target`						F	D	E	M
	`irmovq $3, %rax`	`# Target + 1`							F	D	E

Fine-Grained Switching

- One big bonus of fine-grained switching: no need for branch predictor!!

The branch prediction approach



# demo-j.ys	1	2	3	4	5	6	7	8	9	10
0x000: xorq %rax,%rax	F	D	E	M	W					
0x002: jne target # Not taken		F	D	E	M	W				
0x016: irmovq $2,%rdx # Target			F	D						
bubble					E	M	W			
0x020: irmovq $3,%rbx # Target+1				F						
bubble					D	E	M	W		
0x00b: irmovq $1,%rax # Fall through					F	D	E	M	W	
0x015: halt						F	D	E	M	W

Fine-Grained Switching

- One big bonus of fine-grained switching: no need for branch predictor!!

The fine-grained multi-threading approach

```
xorg %rax, %rax
Inst x from TID=1
Inst y from TID=2
jne L1              # Not taken
Inst x+1 from TID=1
Inst y+1 from TID=2
irmovq $1, %rax     # Fall Through
Inst x+2 from TID=1
Inst y+2 from TID=2
… …
```

1	2	3	4	5	6	7	8	9
F	D	E	M	W				
	F	D	E	M	W			
		F	D	E	M	W		
			F	D	E	M	W	
				F	D	E	M	W
					F	D	E	M
						F	D	E
							F	D
								F

Fine-Grained Switching

- One big bonus of fine-grained switching: no need for branch predictor!!
    - Context switching overhead would be very high! Use separate hardware contexts for each thread (e.g., separate register files).

The fine-grained multi-threading approach

	1	2	3	4	5	6	7	8	9
`xorg %rax, %rax`	F	D	E	M	W				
`Inst x from TID=1`		F	D	E	M	W			
`Inst y from TID=2`			F	D	E	M	W		
`jne L1             # Not taken`				F	D	E	M	W	
`Inst x+1 from TID=1`					F	D	E	M	W
`Inst y+1 from TID=2`						F	D	E	M
`irmovq $1, %rax    # Fall Through`							F	D	E
`Inst x+2 from TID=1`								F	D
`Inst y+2 from TID=2`									F

… …

Fine-Grained Switching

- One big bonus of fine-grained switching: no need for branch predictor!!
  - Context switching overhead would be very high! Use separate hardware contexts for each thread (e.g., separate register files).
  - GPUs do this (among other things). More later.

The fine-grained multi-threading approach

```
xorg %rax, %rax
Inst x from TID=1
Inst y from TID=2
jne L1                # Not taken
Inst x+1 from TID=1
Inst y+1 from TID=2
irmovq $1, %rax       # Fall Through
Inst x+2 from TID=1
Inst y+2 from TID=2
… …
```

1	2	3	4	5	6	7	8	9
F	D	E	M	W				
	F	D	E	M	W			
		F	D	E	M	W		
			F	D	E	M	W	
				F	D	E	M	W
					F	D	E	M
						F	D	E
							F	D
								F

Thread Switching

- A perhaps not great analogy: yellow traffic light

Thread Switching

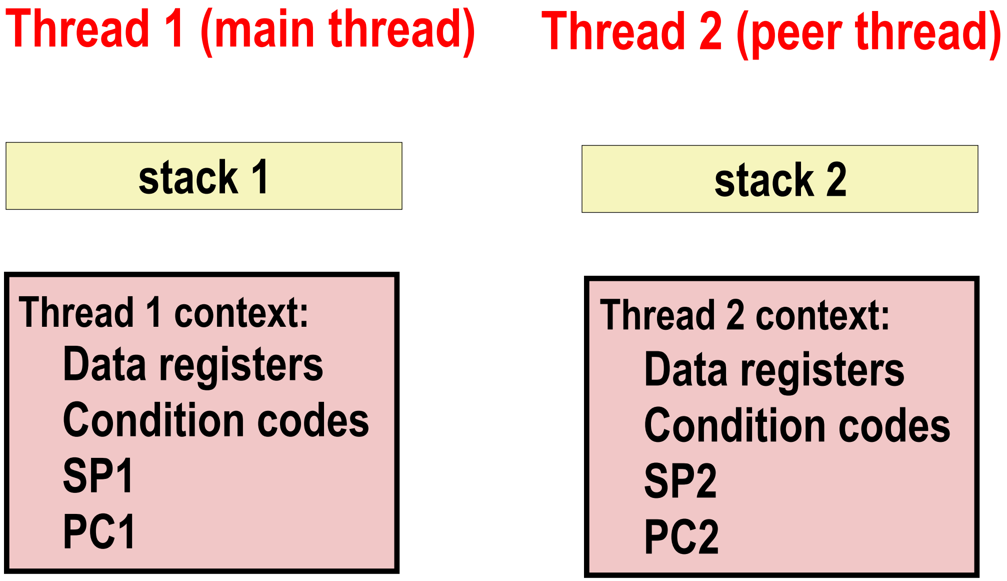
- A perhaps not great analogy: yellow traffic light
- Context switching is pure overhead, but you have to have it in order to support many threads with limited resources

Thread Switching

- A perhaps not great analogy: yellow traffic light
- Context switching is pure overhead, but you have to have it in order to support many threads with limited resources
- What does thread switching do?

Thread Switching

- A perhaps not great analogy: yellow traffic light
- Context switching is pure overhead, but you have to have it in order to support many threads with limited resources
- What does thread switching do?



Thread 1 (main thread)

Thread 2 (peer thread)

stack 1

stack 2

Thread 1 context:
Data registers
Condition codes
SP1
PC1

Thread 2 context:
Data registers
Condition codes
SP2
PC2

Thread Switching

- A perhaps not great analogy: yellow traffic light
- Context switching is pure overhead, but you have to have it in order to support many threads with limited resources
- What does thread switching do?
  - Save the context for the old thread, and restore the context of the new thread

Thread Switching

- A perhaps not great analogy: yellow traffic light
- Context switching is pure overhead, but you have to have it in order to support many threads with limited resources
- What does thread switching do?
  - Save the context for the old thread, and restore the context of the new thread
- How do you implement thread context switching?

Thread Switching

- A perhaps not great analogy: yellow traffic light
- Context switching is pure overhead, but you have to have it in order to support many threads with limited resources
- What does thread switching do?
  - Save the context for the old thread, and restore the context of the new thread
- How do you implement thread context switching?
  - Save/restore thread contexts in memory

Thread Switching

- A perhaps not great analogy: yellow traffic light
- Context switching is pure overhead, but you have to have it in order to support many threads with limited resources
- What does thread switching do?
  - Save the context for the old thread, and restore the context of the new thread
- How do you implement thread context switching?
  - Save/restore thread contexts in memory
  - Have dedicated context for each thread (e.g., each thread has a dedicated register file)

Thread Switching

- A perhaps not great analogy: yellow traffic light
- Context switching is pure overhead, but you have to have it in order to support many threads with limited resources
- What does thread switching do?
  - Save the context for the old thread, and restore the context of the new thread
- How do you implement thread context switching?
  - Save/restore thread contexts in memory
  - Have dedicated context for each thread (e.g., each thread has a dedicated register file)
    - Lots of hardware resources, but is a must if we want to support a lot of threads.

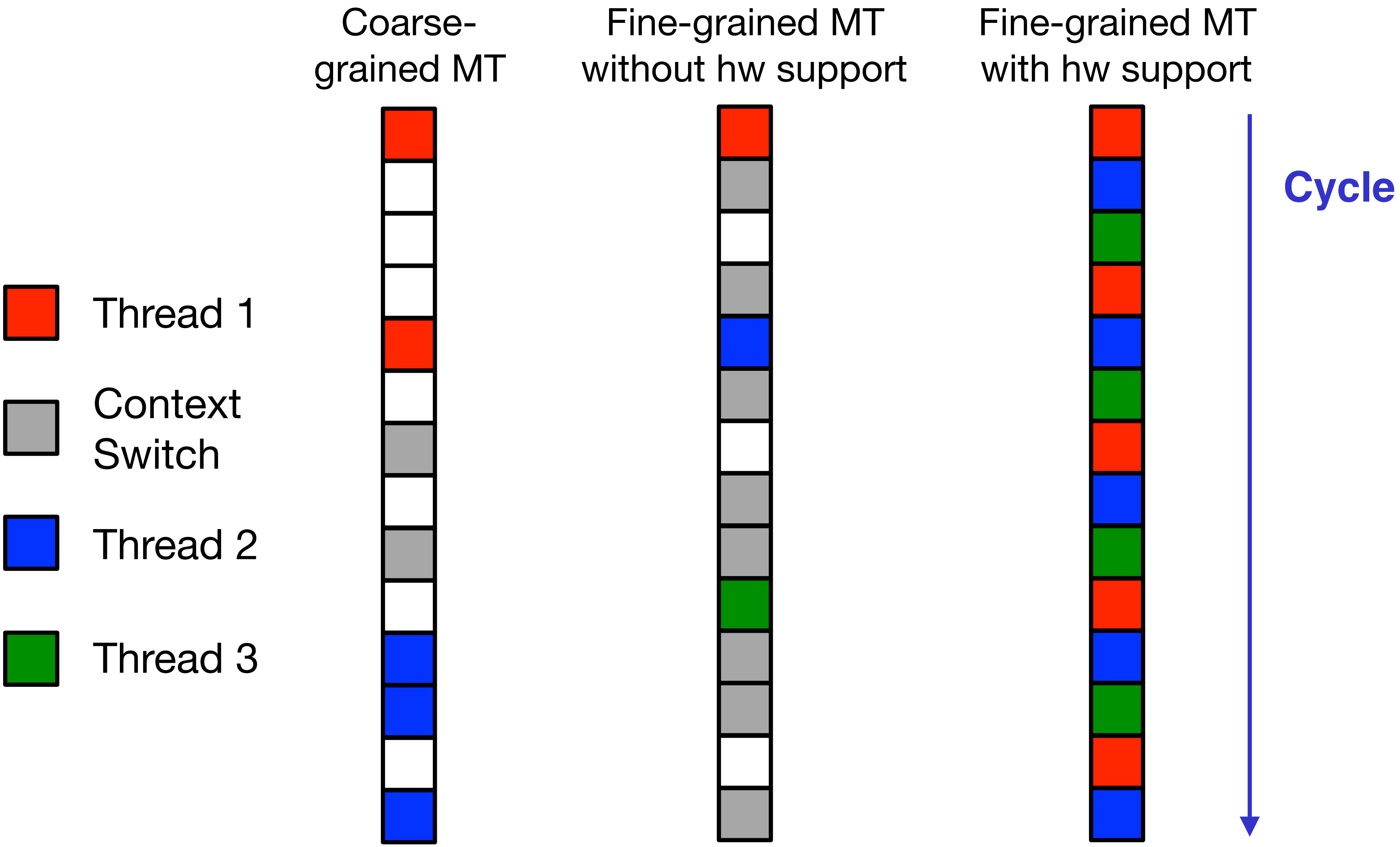
Thread Switching

- A perhaps not great analogy: yellow traffic light
- Context switching is pure overhead, but you have to have it in order to support many threads with limited resources
- What does thread switching do?
    - Save the context for the old thread, and restore the context of the new thread
- How do you implement thread context switching?
    - Save/restore thread contexts in memory
    - Have dedicated context for each thread (e.g., each thread has a dedicated register file)
        - Lots of hardware resources, but is a must if we want to support a lot of threads.
        - GPU does this (later).

Thread Switching

- A perhaps not great analogy: yellow traffic light
- Context switching is pure overhead, but you have to have it in order to support many threads with limited resources
- What does thread switching do?
  - Save the context for the old thread, and restore the context of the new thread
- How do you implement thread context switching?
  - Save/restore thread contexts in memory
  - Have dedicated context for each thread (e.g., each thread has a dedicated register file)
    - Lots of hardware resources, but is a must if we want to support a lot of threads.
    - GPU does this (later).
    - CPU does this for a limited number of threads (hyper-threading, later).

Multi-threading Illustration (so far…)



🟥 Thread 1

▨ Context Switch

🟦 Thread 2

🟩 Thread 3

Coarse-grained MT

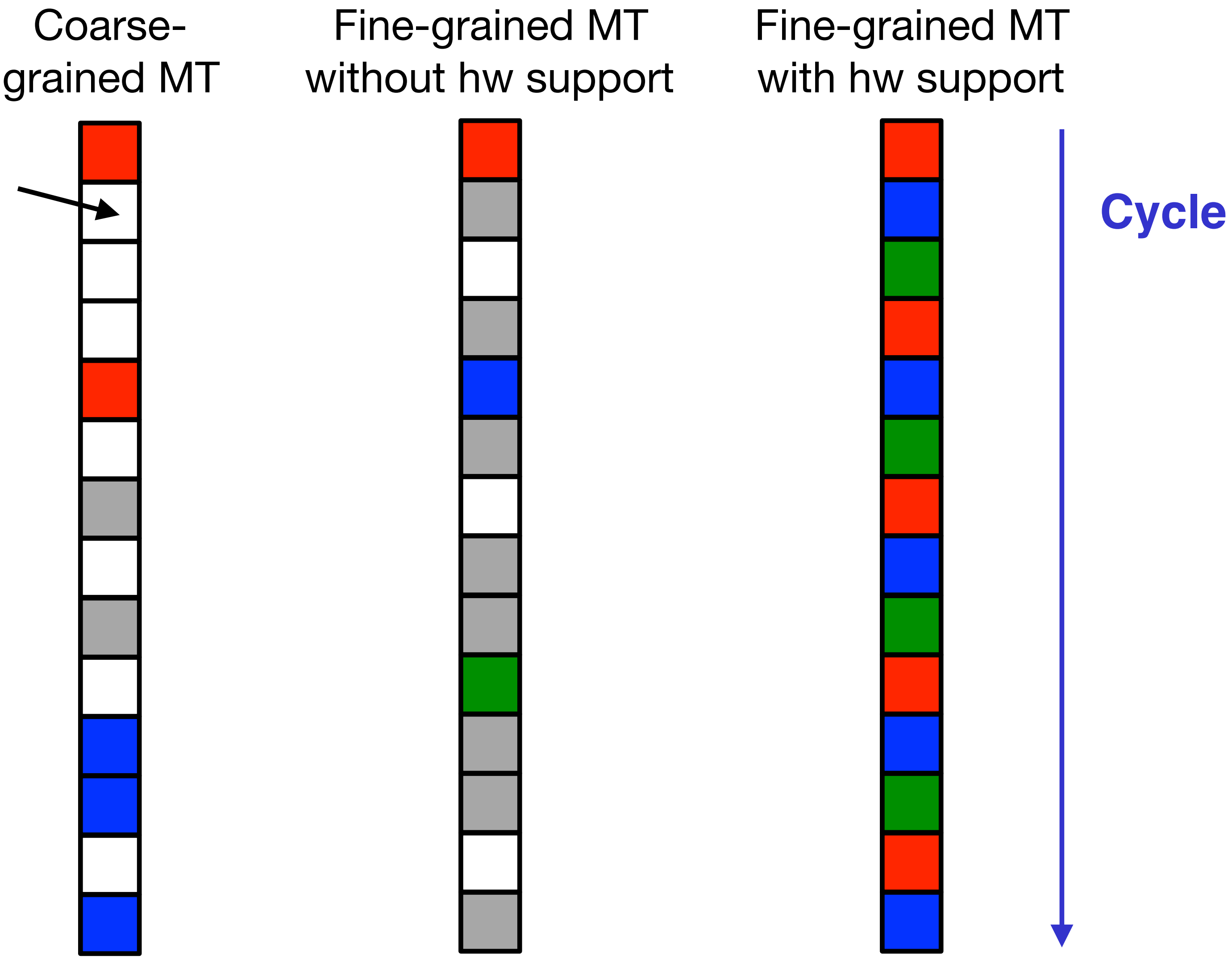
Fine-grained MT without hw support

Fine-grained MT with hw support

Cycle

Multi-threading Illustration (so far…)

Why these empty slots?



Coarse-grained MT

Fine-grained MT without hw support

Fine-grained MT with hw support

Cycle

🟥 Thread 1

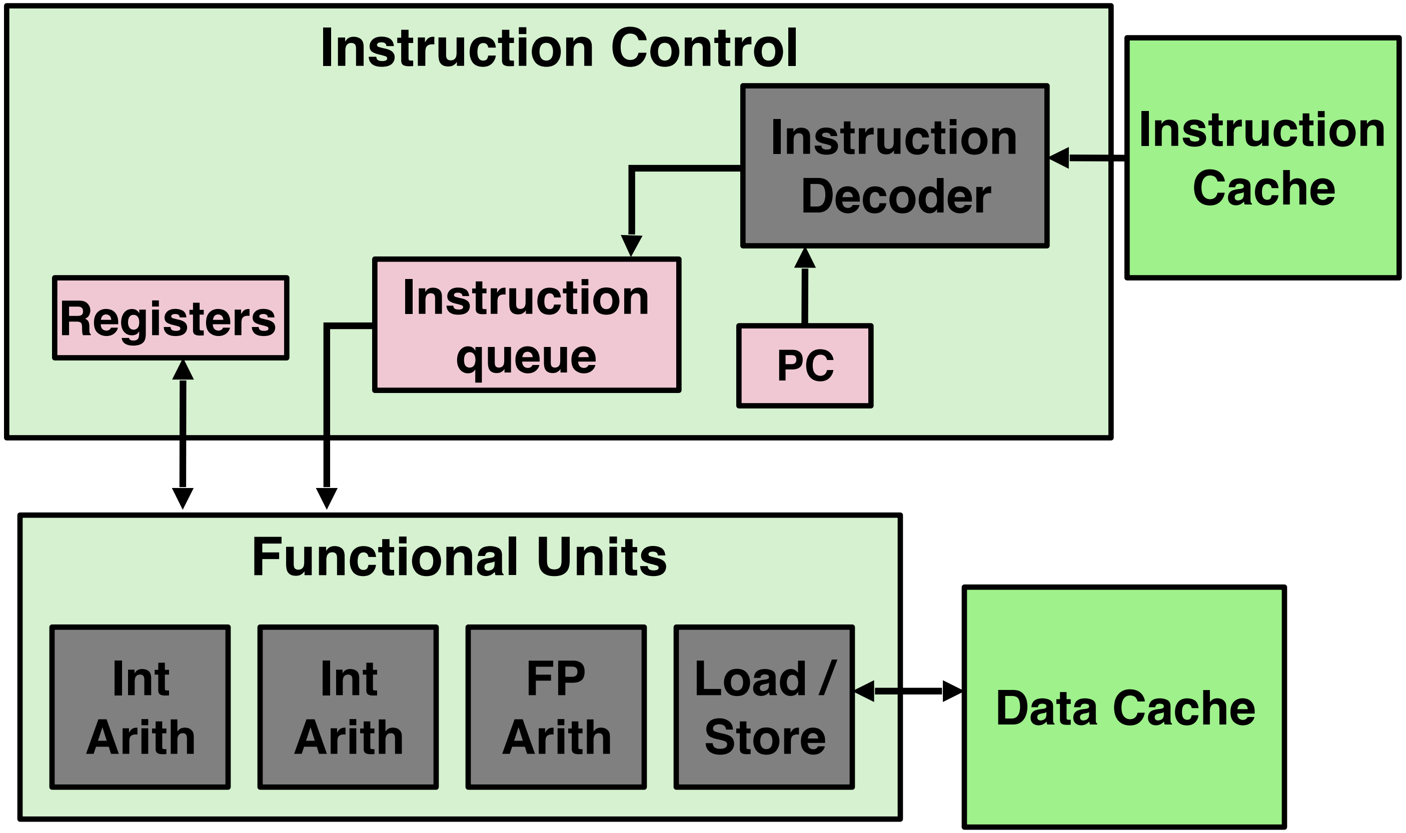
⬜ Context Switch

🟦 Thread 2

🟩 Thread 3

Modern Single-Core: Superscalar

- Typically has multiple function units to allow for decoding and issuing multiple instructions at the same time
- Called "Superscalar"

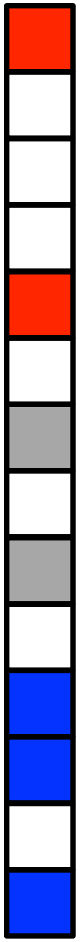


From Scalar to Multi-Scalar Multi-threading

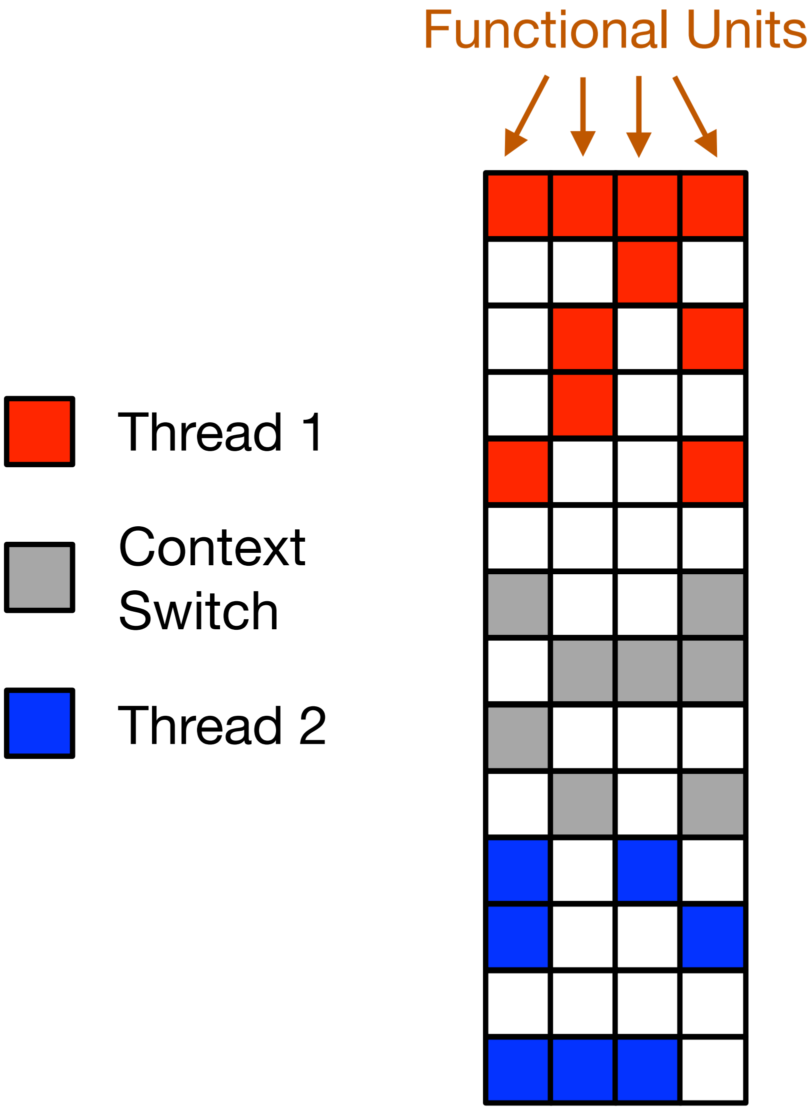
🟥 Thread 1

⬜ Context Switch

🟦 Thread 2



From Scalar to Multi-Scalar Multi-threading



Functional Units

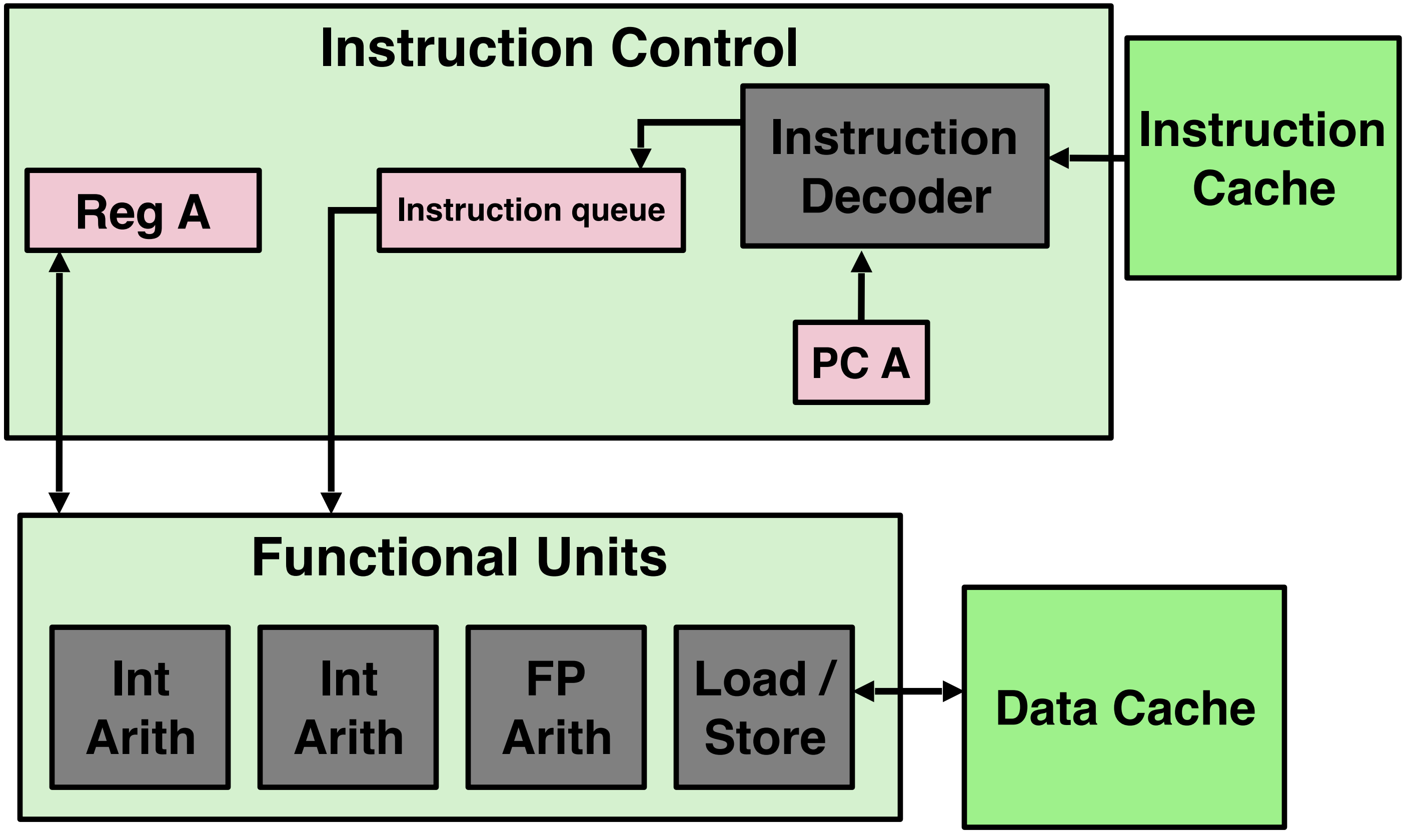
🟥 Thread 1

▨ Context Switch

🟦 Thread 2

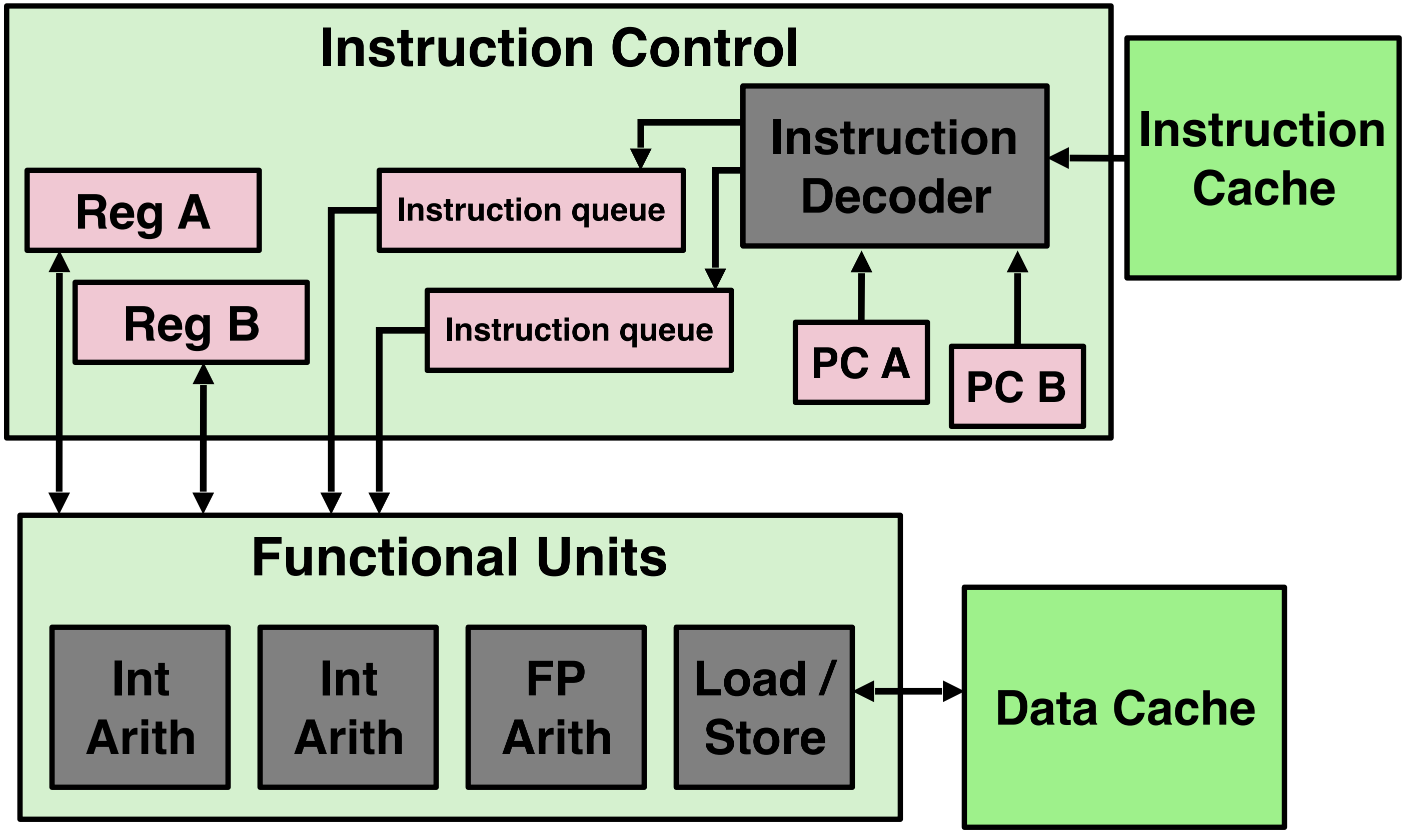
Simultaneous Multi-Threading (SMT)

- Intel call it hyper-threading.
- Replicate enough hardware structures to process K instruction streams, i.e., threads. K copies of all registers. Share functional units.
- SMT = Superscalar + Multi-threading



Simultaneous Multi-Threading (SMT)

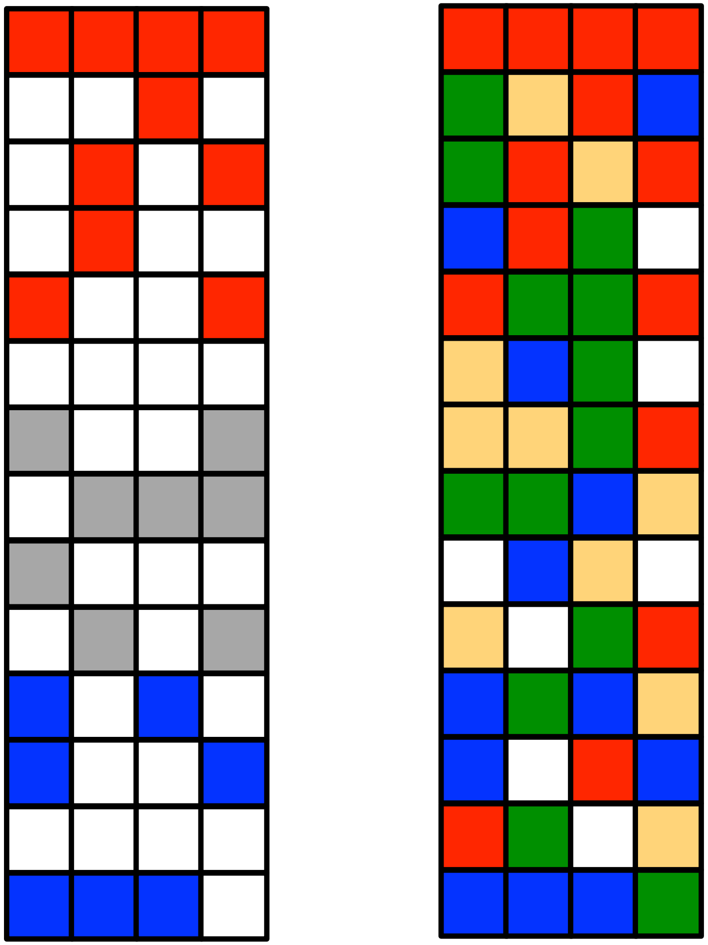
- Intel call it hyper-threading.
- Replicate enough hardware structures to process K instruction streams, i.e., threads. K copies of all registers. Share functional units.
- SMT = Superscalar + Multi-threading



Conventional Multi-threading vs. Hyper-threading

Coarse-grained MT on a superscalar core

SMT



- 🟥 Thread 1
- ⬜ Context Switch
- 🟦 Thread 2
- 🟩 Thread 3
- 🟨 Thread 4

Conventional Multi-threading vs. Hyper-threading

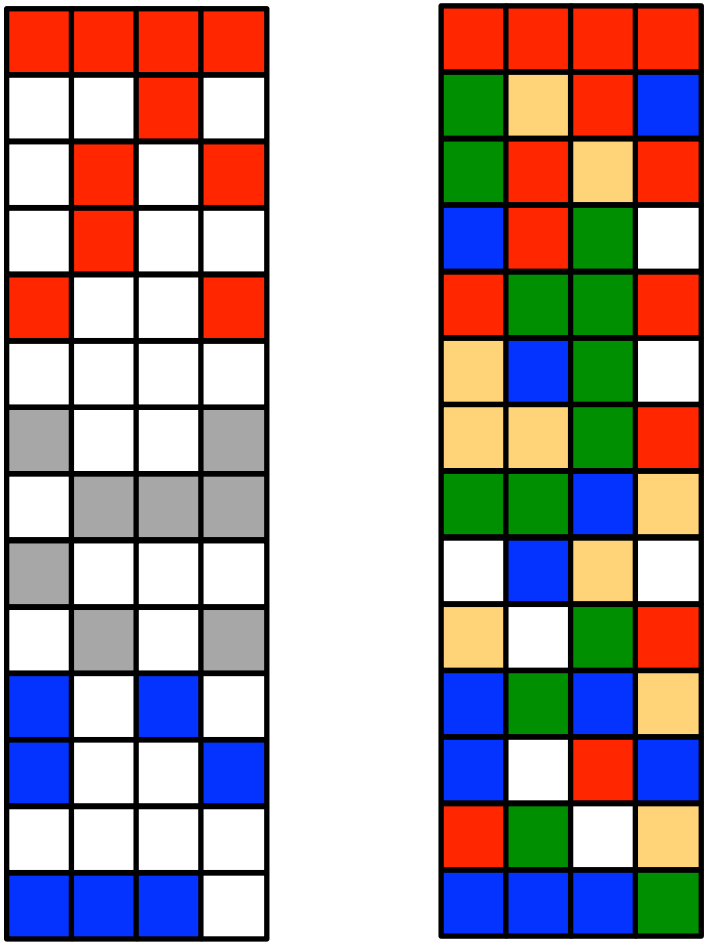
🟥 Thread 1

▨ Context Switch

🟦 Thread 2

🟩 Thread 3

🟨 Thread 4



Coarse-grained MT on a superscalar core

SMT

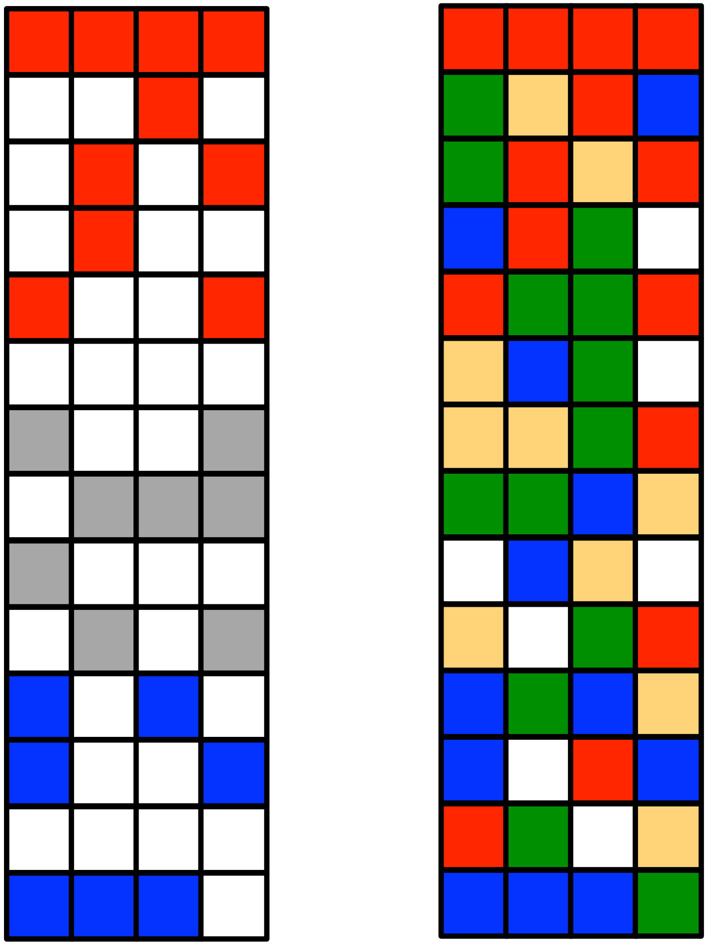
Can now make use of idle issue slots in conventional MT cores.

Conventional Multi-threading vs. Hyper-threading

- 🟥 Thread 1
- ⬜ Context Switch
- 🟦 Thread 2
- 🟩 Thread 3
- 🟨 Thread 4

Coarse-grained MT on a superscalar core

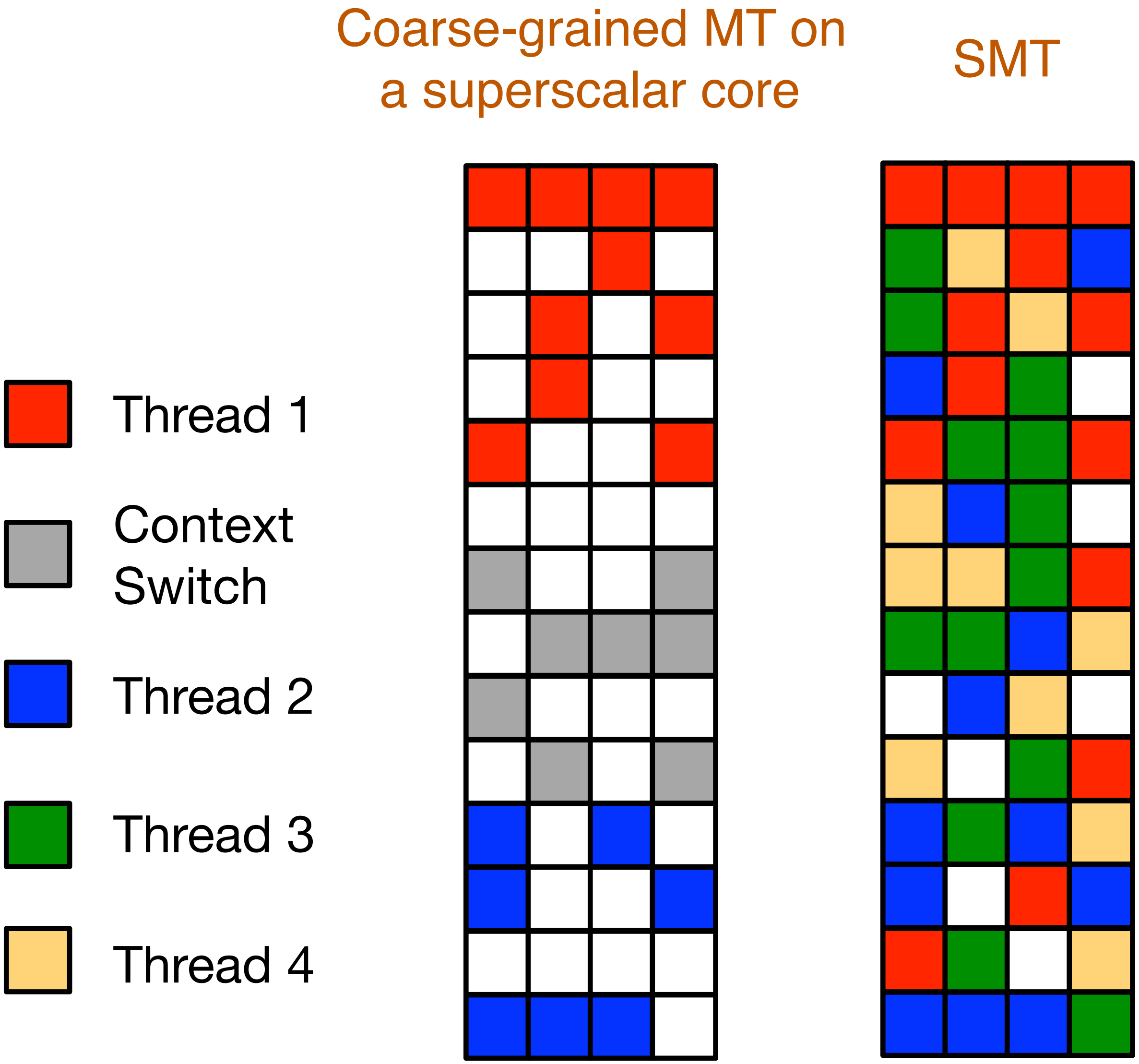
SMT



Can now make use of idle issue slots in conventional MT cores.

Multiple threads actually execute in parallel (even with one single core)

Conventional Multi-threading vs. Hyper-threading



Can now make use of idle issue slots in conventional MT cores.

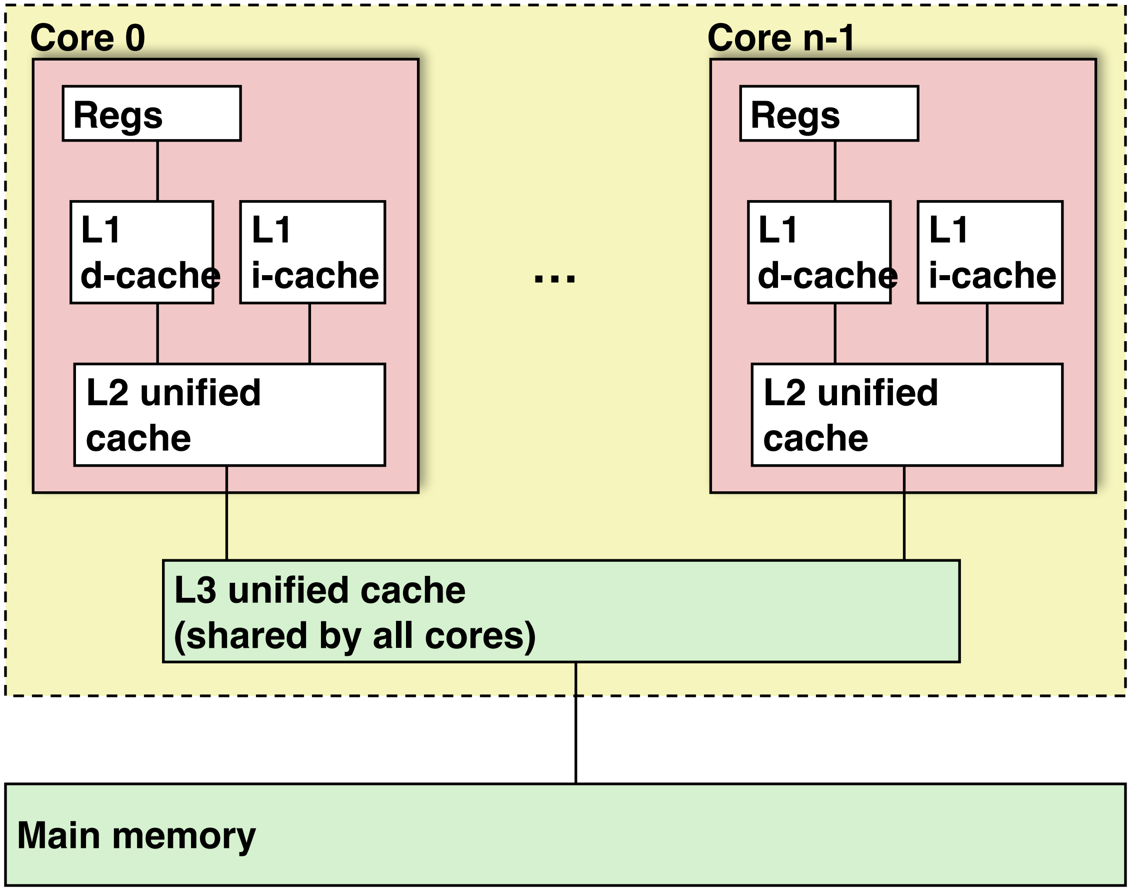
Multiple threads actually execute in parallel (even with one single core)

No/little context switch overhead

Today

- From process to threads
  - Basic thread execution model
- Multi-threading programming
- Hardware support of threads
  - Single core
  - Multi-core
  - Cache coherence

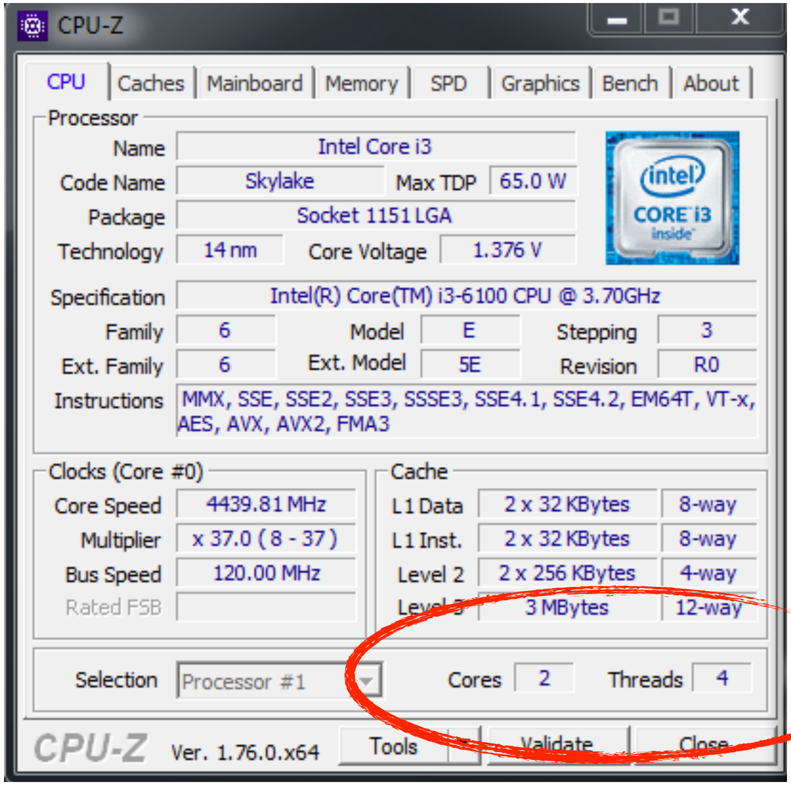
Multi-Threading on a Multi-core Processor



- Each core can run multiple threads, mostly through coarse-grained switching.
- Fine-grained switching on conventional multi-core CPU is too costly.

Combine Multi-core with SMT

- Common for laptop/desktop/server machine. E.g., 2 physical cores, each core has 2 hyper-threads => 4 virtual cores.
- Not for mobile processors (Hyper-threading costly to implement)



Processor

Name: Intel Core i3

Code Name: Skylake | Max TDP: 65.0 W

Package: Socket 1151 LGA

Technology: 14 nm | Core Voltage: 1.376 V

Specification: Intel(R) Core(TM) i3-6100 CPU @ 3.70GHz

Family: 6 | Model: E | Stepping: 3

Ext. Family: 6 | Ext. Model: 5E | Revision: R0

Instructions: MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX, AVX2, FMA3

Clocks (Core #0)

Core Speed: 4439.81 MHz

Multiplier: x 37.0 (8 - 37)

Bus Speed: 120.00 MHz

Rated FSB:

Cache

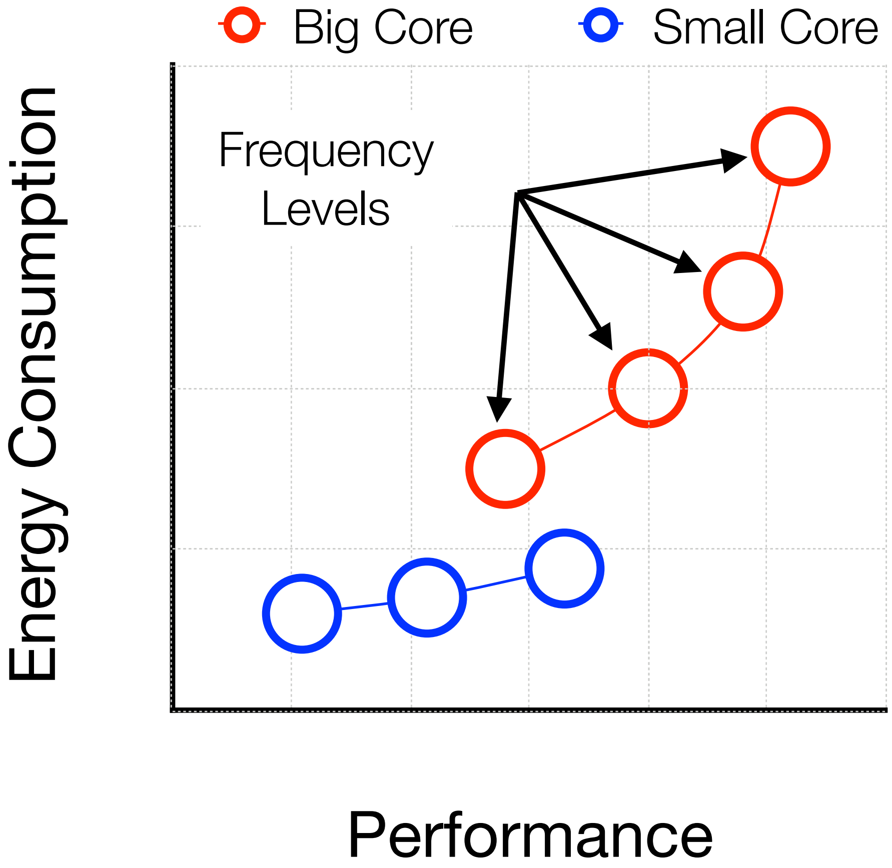
L1 Data	2 x 32 KBytes	8-way
L1 Inst.	2 x 32 KBytes	8-way
Level 2	2 x 256 KBytes	4-way
Level 3	3 MBytes	12-way

Selection: Processor #1 | Cores: 2 | Threads: 4

CPU-Z Ver. 1.76.0.x64

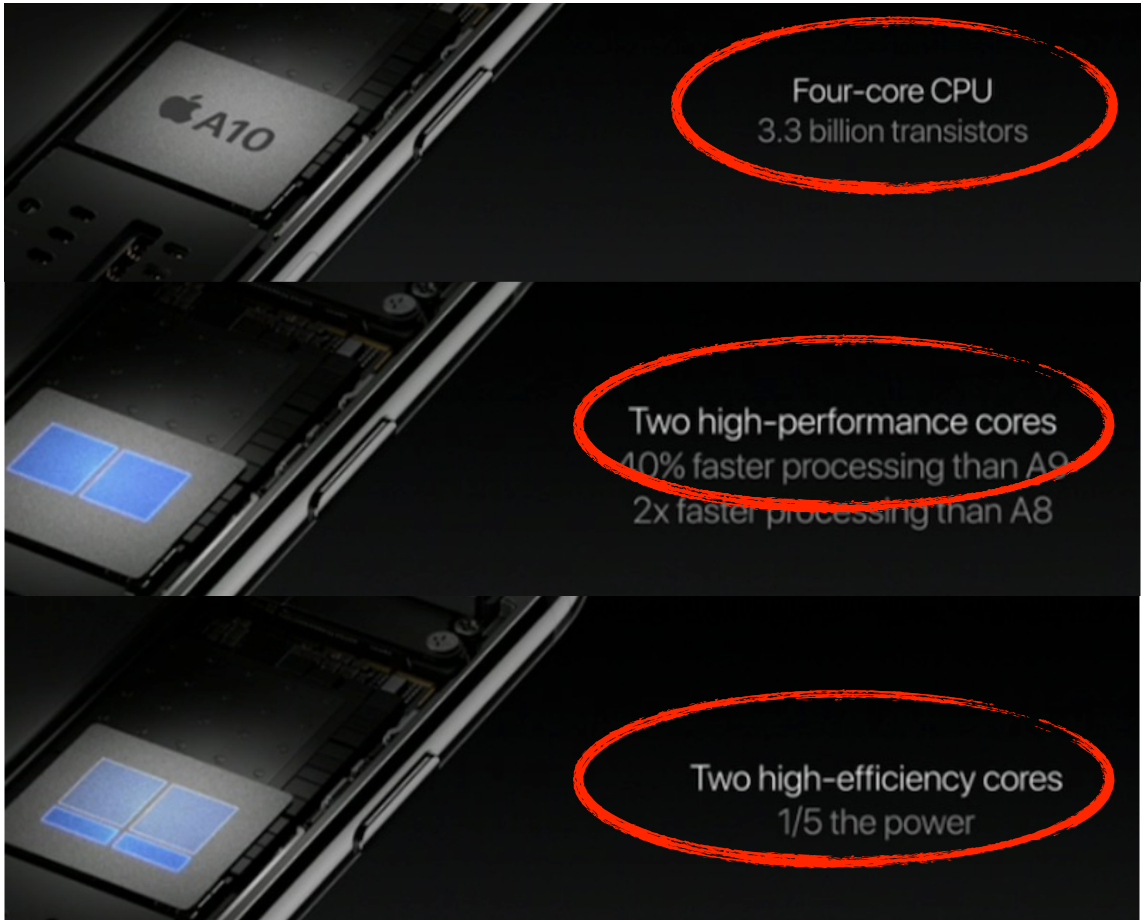
Asymmetric Multiprocessor (AMP)

- Offer a large performance-energy trade-off space



Asymmetric Chip-Multiprocessor (ACMP)

- Already used in commodity devices (e.g., Samsung Galaxy S6, iPhone 7)



Four-core CPU
3.3 billion transistors

Two high-performance cores
40% faster processing than A9
2x faster processing than A8

Two high-efficiency cores
1/5 the power

Today

- From process to threads
  - Basic thread execution model
- Multi-threading programming
- Hardware support of threads
  - Single core
  - Multi-core
  - Cache coherence

The Issue

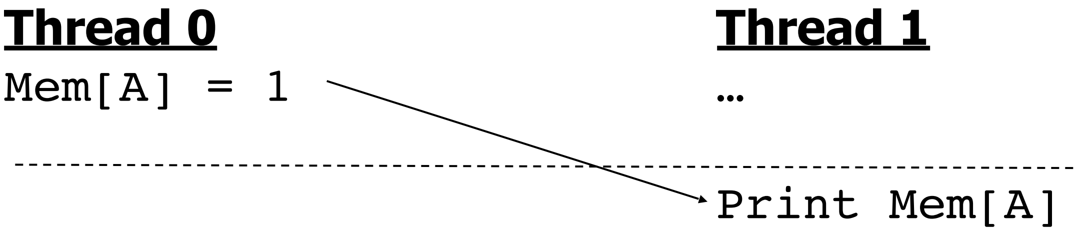
- Assume that we have a multi-core processor. Thread 0 runs on Core 0, and Thread 1 runs on Core 1.

The Issue

- Assume that we have a multi-core processor. Thread 0 runs on Core 0, and Thread 1 runs on Core 1.
- Threads share variables: e.g., Thread 0 writes to an address, followed by Thread 1 reading.

The Issue

- Assume that we have a multi-core processor. Thread 0 runs on Core 0, and Thread 1 runs on Core 1.
- Threads share variables: e.g., Thread 0 writes to an address, followed by Thread 1 reading.



Thread 0

`Mem[A] = 1`

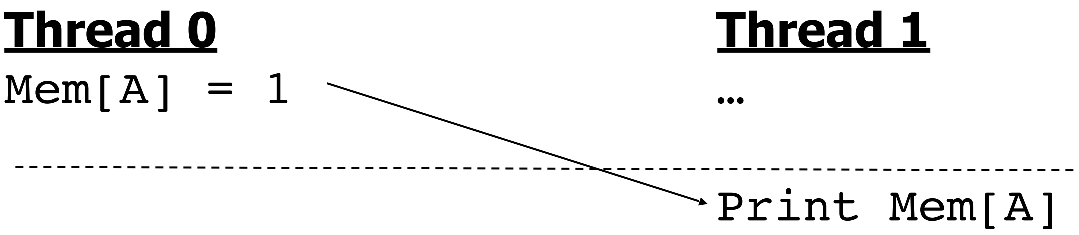
Thread 1

…

`Print Mem[A]`

The Issue

- Assume that we have a multi-core processor. Thread 0 runs on Core 0, and Thread 1 runs on Core 1.
- Threads share variables: e.g., Thread 0 writes to an address, followed by Thread 1 reading.
- Each read should receive the value last written by anyone



Thread 0

`Mem[A] = 1`

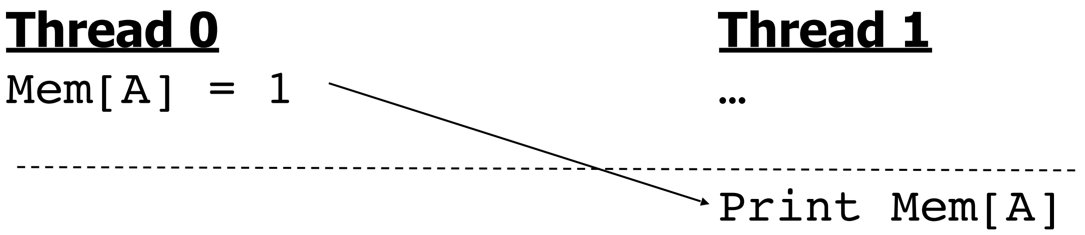
Thread 1

…

`Print Mem[A]`

The Issue

- Assume that we have a multi-core processor. Thread 0 runs on Core 0, and Thread 1 runs on Core 1.
- Threads share variables: e.g., Thread 0 writes to an address, followed by Thread 1 reading.
- Each read should receive the value last written by anyone
- **Basic question**: If multiple cores access the same data, how do they ensure they all see a consistent state?



Thread 0

```
Mem[A] = 1
```

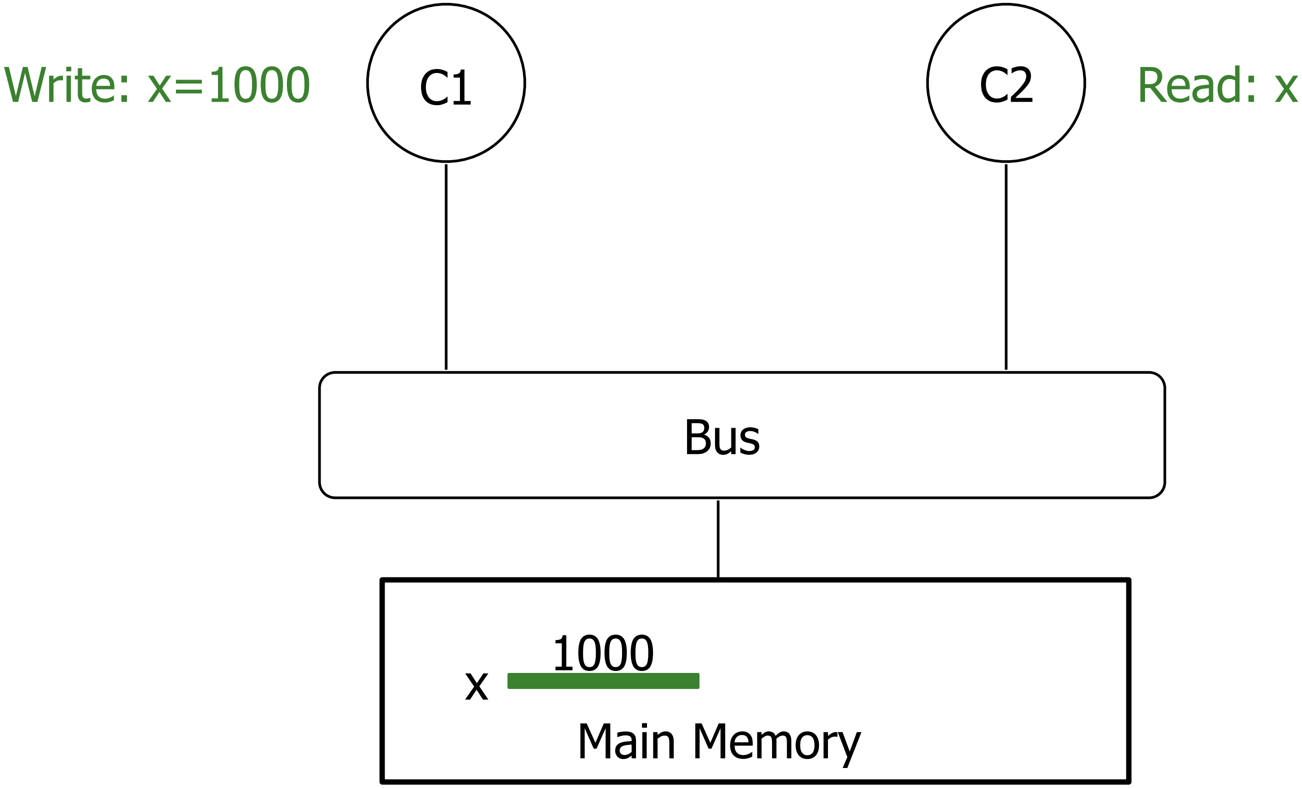
Thread 1

…

```
Print Mem[A]
```

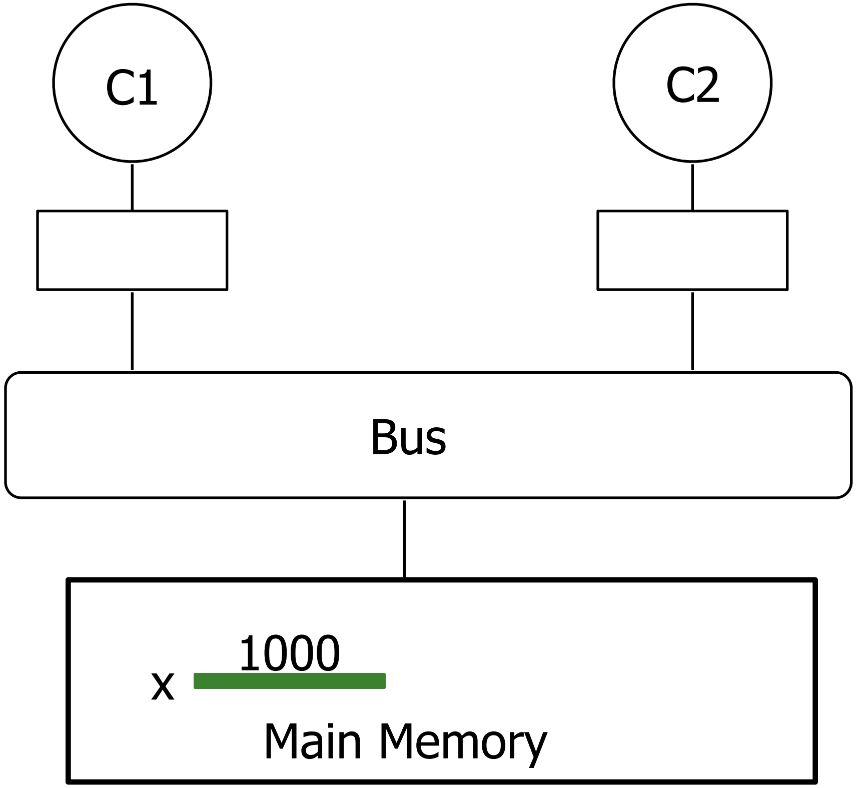
The Issue

- Without cache, the issue is (theoretically) solvable by using mutex.
- …because there is only one copy of x in the entire system. Accesses to x in memory are serialized by mutex.



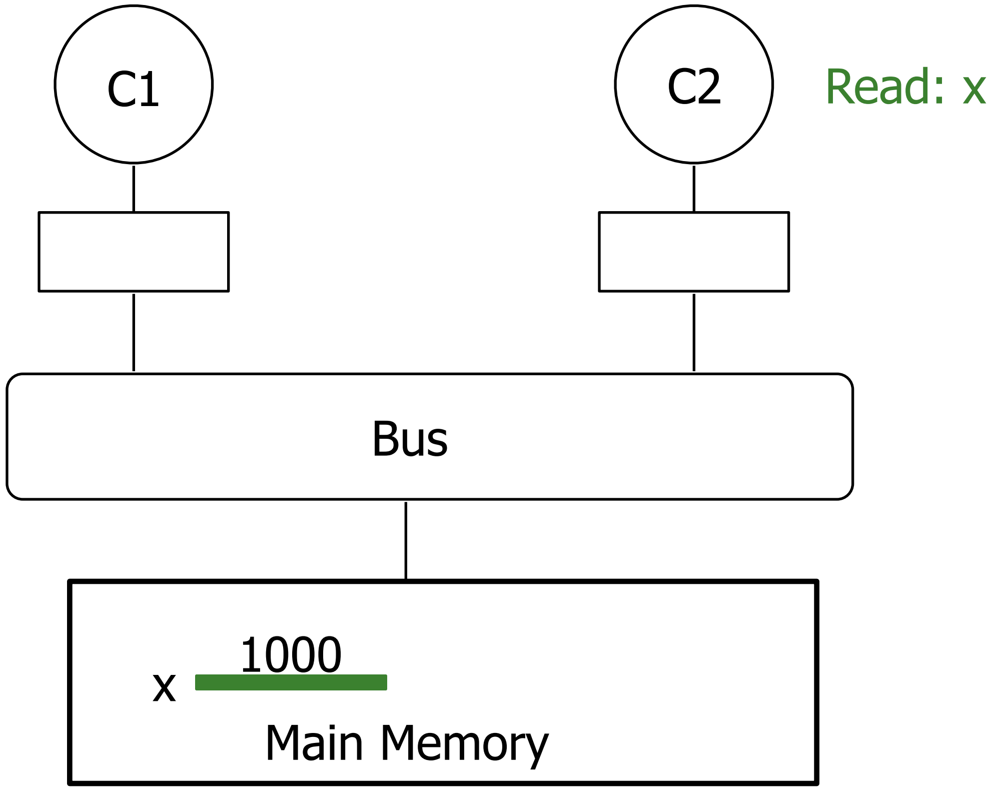
The Issue

- What if each core **cache** the same data, how do they ensure they all see a consistent state? (assuming a write-back cache)



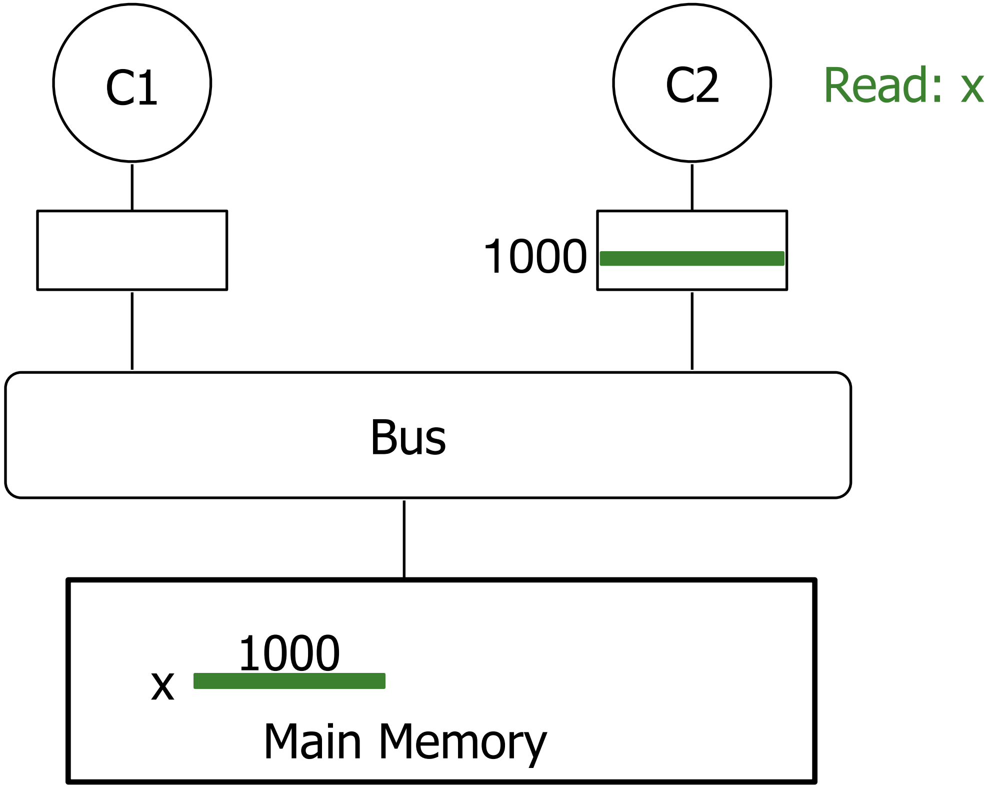
The Issue

- What if each core **cache** the same data, how do they ensure they all see a consistent state? (assuming a write-back cache)



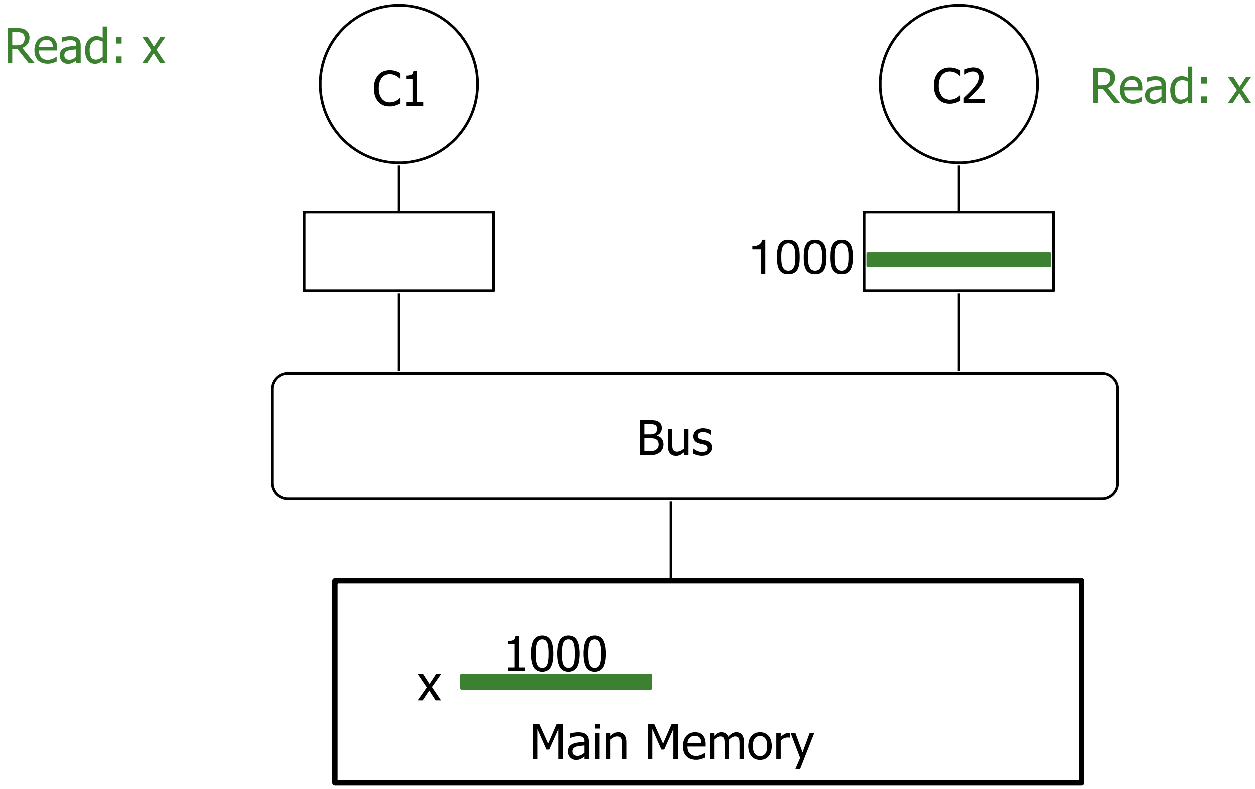
The Issue

- What if each core **cache** the same data, how do they ensure they all see a consistent state? (assuming a write-back cache)



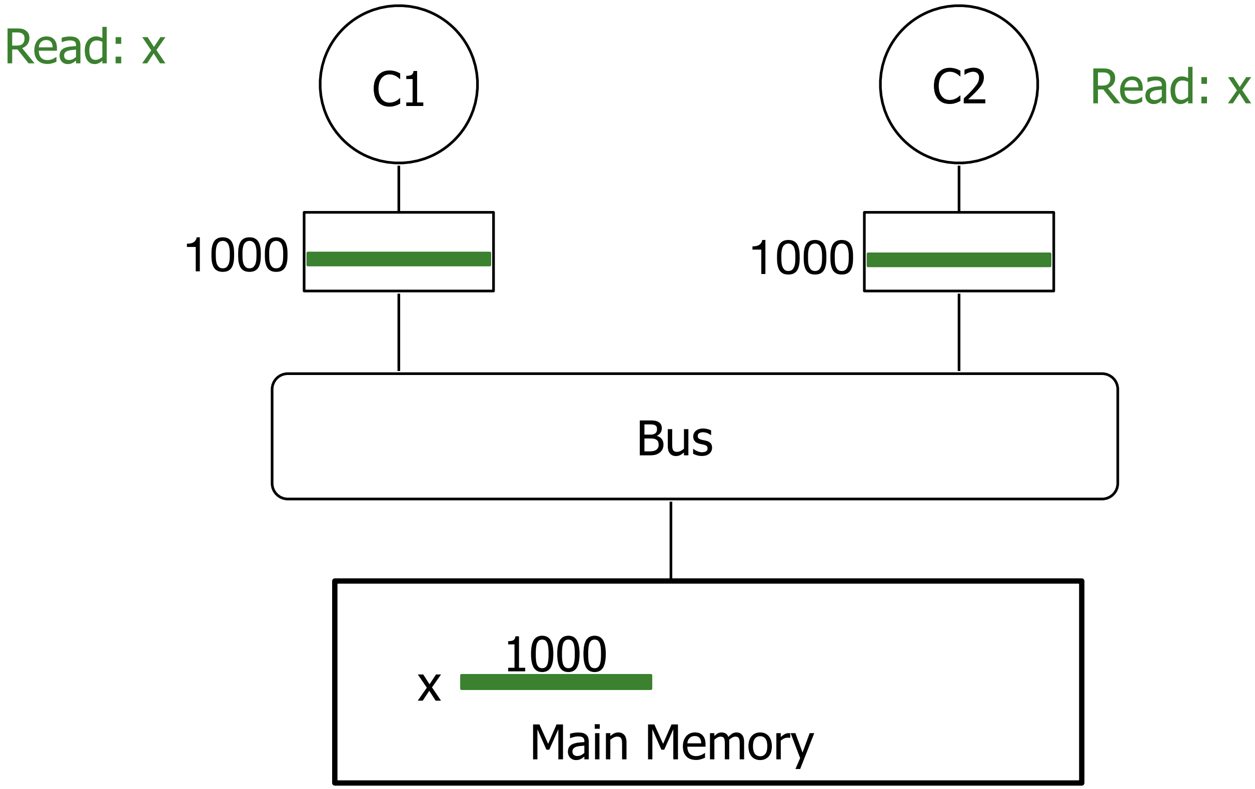
The Issue

- What if each core **cache** the same data, how do they ensure they all see a consistent state? (assuming a write-back cache)



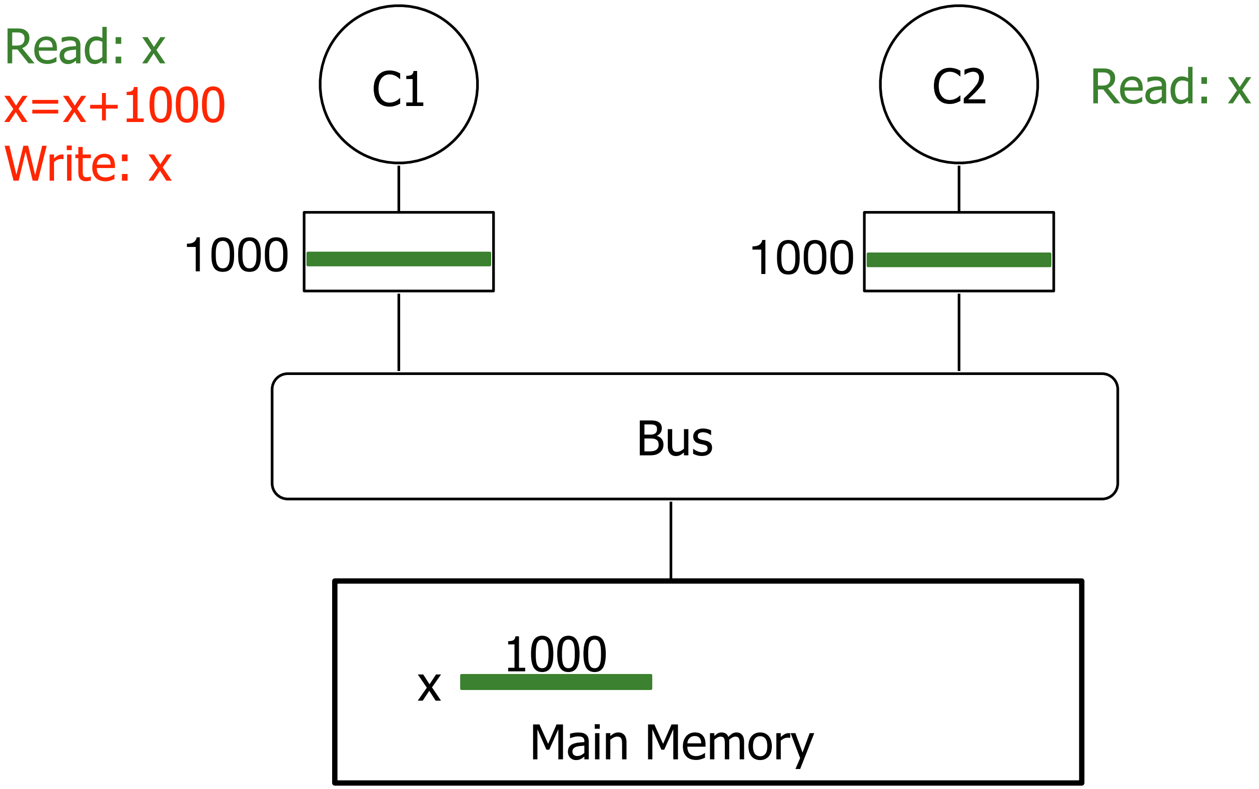
The Issue

- What if each core **cache** the same data, how do they ensure they all see a consistent state? (assuming a write-back cache)



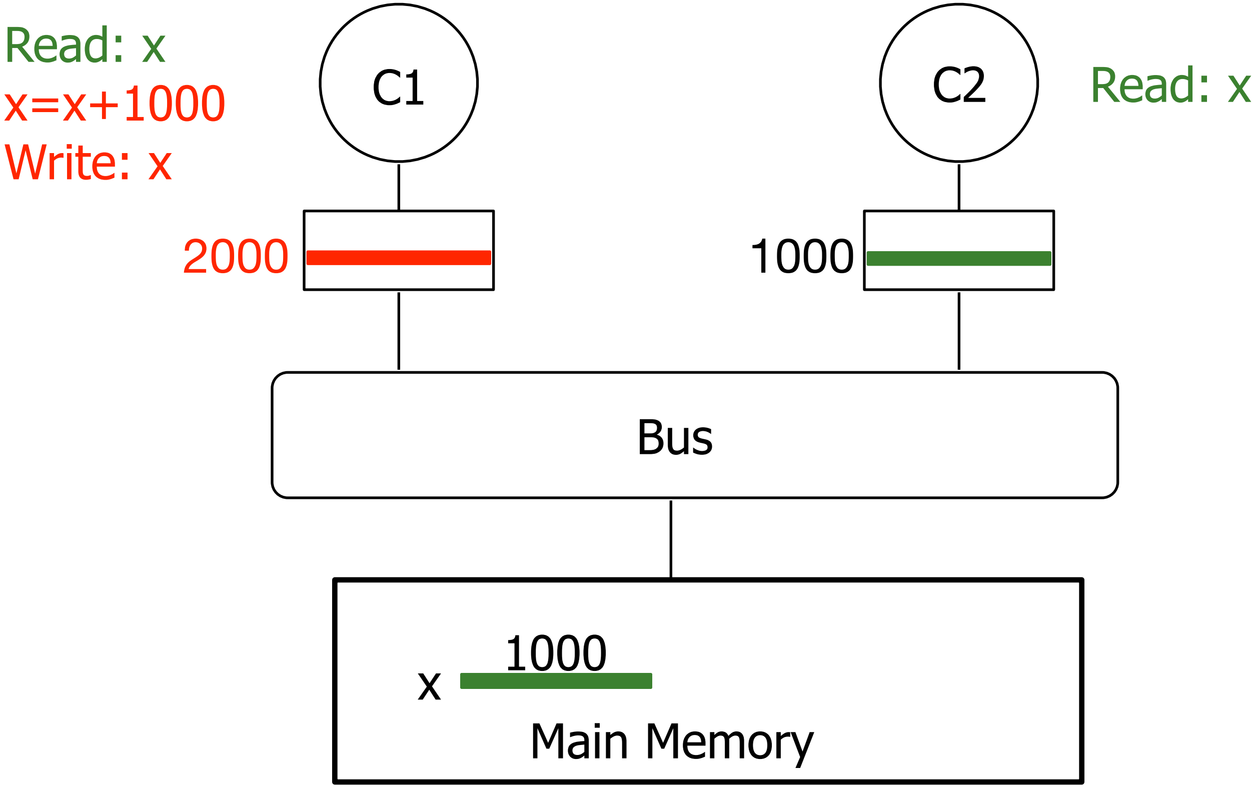
The Issue

- What if each core **cache** the same data, how do they ensure they all see a consistent state? (assuming a write-back cache)



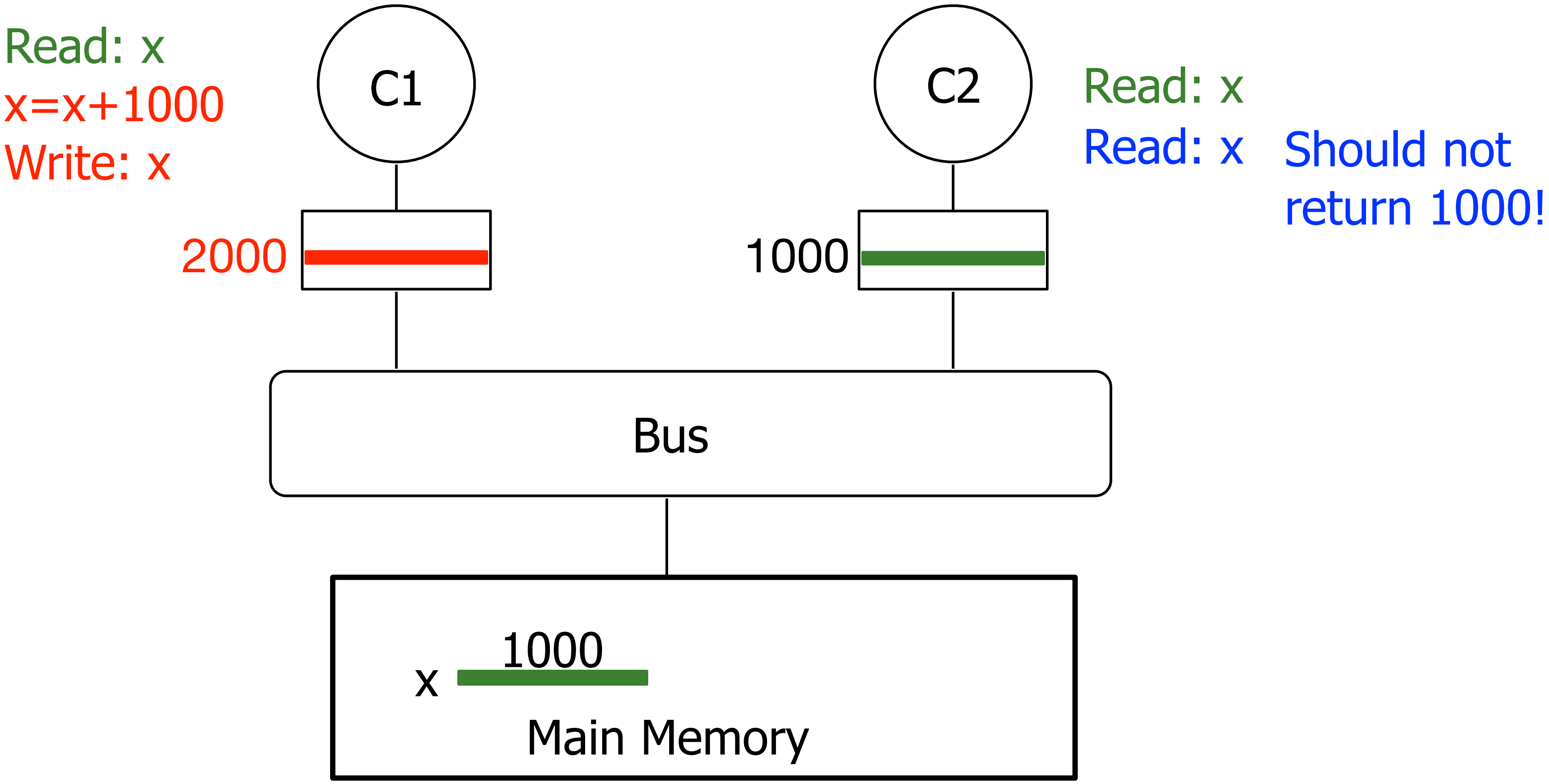
The Issue

- What if each core **cache** the same data, how do they ensure they all see a consistent state? (assuming a write-back cache)



The Issue

- What if each core **cache** the same data, how do they ensure they all see a consistent state? (assuming a write-back cache)



Cache Coherence: The Idea

- **Issue**: there are multiple copies of the same data in the system, and they could have different values at the same time.

Cache Coherence: The Idea

- **Issue**: there are multiple copies of the same data in the system, and they could have different values at the same time.
- **Idea**: ensure multiple copies have same value, i.e., *coherent*

Cache Coherence: The Idea

- **Issue**: there are multiple copies of the same data in the system, and they could have different values at the same time.
- **Idea**: ensure multiple copies have same value, i.e., *coherent*
- **How?** Two options:

Cache Coherence: The Idea

- **Issue**: there are multiple copies of the same data in the system, and they could have different values at the same time.
- **Idea**: ensure multiple copies have same value, i.e., *coherent*
- **How?** Two options:
    - Update: push new value to all copies (in other caches)

Cache Coherence: The Idea

- **Issue**: there are multiple copies of the same data in the system, and they could have different values at the same time.
- **Idea**: ensure multiple copies have same value, i.e., *coherent*
- **How?** Two options:
    - Update: push new value to all copies (in other caches)
    - Invalidate: invalidate other copies (in other caches)

Readings: Cache Coherence

- Most helpful
  - Culler and Singh, Parallel Computer Architecture
    - Chapter 5.1 (pp 269 – 283), Chapter 5.3 (pp 291 – 305)
  - Patterson&Hennessy, Computer Organization and Design
    - Chapter 5.8 (pp 534 – 538 in 4th and 4th revised eds.)
  - Papamarcos and Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," ISCA 1984.
- Also very useful
  - Censier and Feautrier, "A new solution to coherence problems in multicache systems," IEEE Trans. Computers, 1978.
  - Goodman, "Using cache memory to reduce processor-memory traffic," ISCA 1983.
  - Laudon and Lenoski, "The SGI Origin: a ccNUMA highly scalable server," ISCA 1997.
  - Martin et al, "Token coherence: decoupling performance and correctness," ISCA 2003.
  - Baer and Wang, "On the inclusion properties for multi-level cache hierarchies," ISCA 1988.

Does Hardware Have to Keep Cache Coherent?

- Hardware-guaranteed cache coherence is complex to implement.

Does Hardware Have to Keep Cache Coherent?

- Hardware-guaranteed cache coherence is complex to implement.
- Can the programmers ensure cache coherence themselves?

Does Hardware Have to Keep Cache Coherent?

- Hardware-guaranteed cache coherence is complex to implement.
- Can the programmers ensure cache coherence themselves?
- Key: ISA must provide cache flush/invalidate instructions
  - FLUSH-LOCAL A: Flushes/invalidates the cache block containing address A from a processor's local cache.
  - FLUSH-GLOBAL A: Flushes/invalidates the cache block containing address A from all other processors' caches.
  - FLUSH-CACHE X: Flushes/invalidates all blocks in cache X.

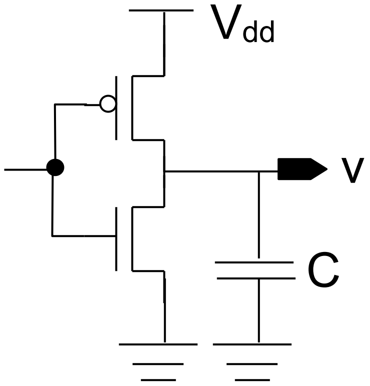
Does Hardware Have to Keep Cache Coherent?

- Hardware-guaranteed cache coherence is complex to implement.
- Can the programmers ensure cache coherence themselves?
- Key: ISA must provide cache flush/invalidate instructions
  - FLUSH-LOCAL A: Flushes/invalidates the cache block containing address A from a processor's local cache.
  - FLUSH-GLOBAL A: Flushes/invalidates the cache block containing address A from all other processors' caches.
  - FLUSH-CACHE X: Flushes/invalidates all blocks in cache X.
- Classic example: TLB
  - Hardware does not guarantee that TLBs of different core are coherent
  - ISA provides instructions for OS to flush PTEs
  - Called "TLB shootdown"

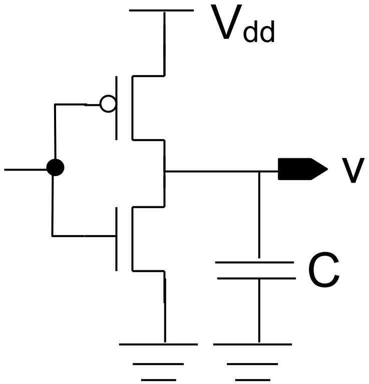
Today

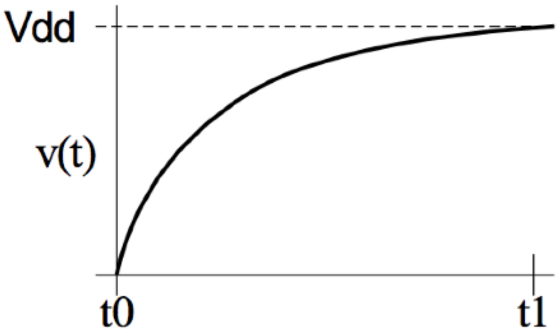
- Power consumption and dark silicon
- GPU
- Accelerators

Dynamic Power

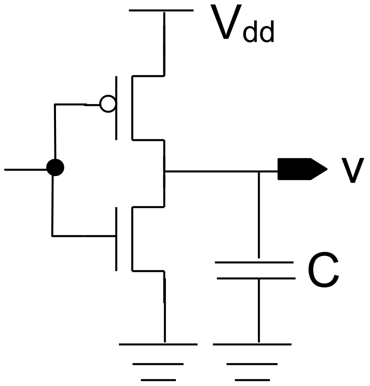


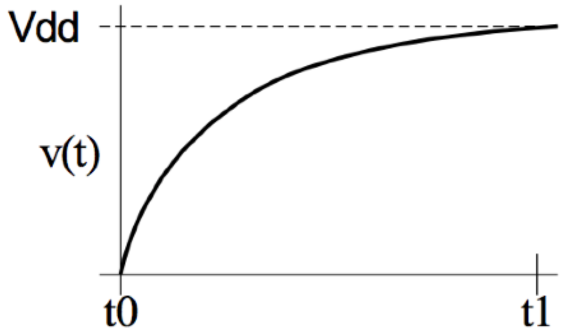
Dynamic Power





Dynamic Power

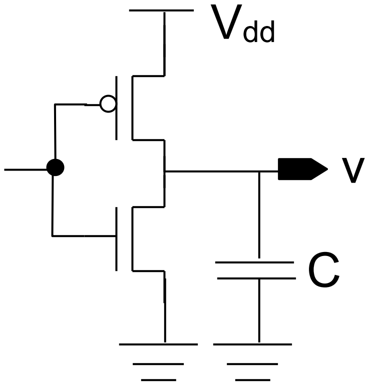


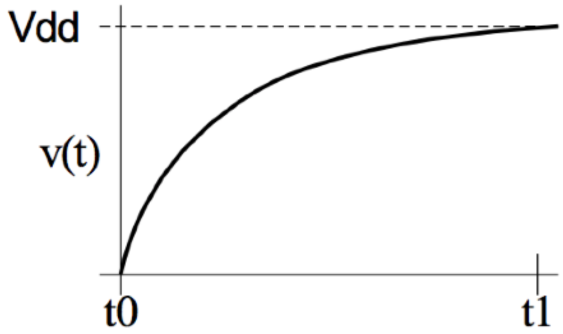


$$E_{sw} = \int_{t_0}^{t_1} P(t)dt = \int_{t_0}^{t_1} (V_{dd} - v)\cdot i(t)dt = \int_{t_0}^{t_1} (V_{dd} - v)\cdot c\,(dv/dt)\,dt =$$

$$= cV_{dd}\int_{t_0}^{t_1} dv - c\int_{t_0}^{t_1} v\cdot dv = cV_{dd}{}^{2} - 1/2cV_{dd}{}^{2} = \boxed{1/2\,cV_{dd}{}^{2}}$$

Dynamic Power



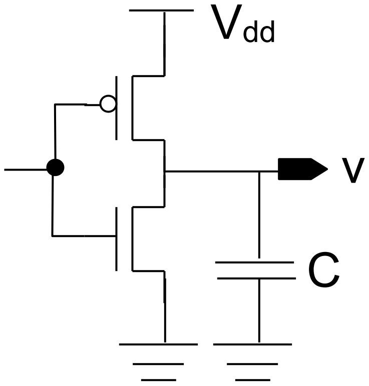


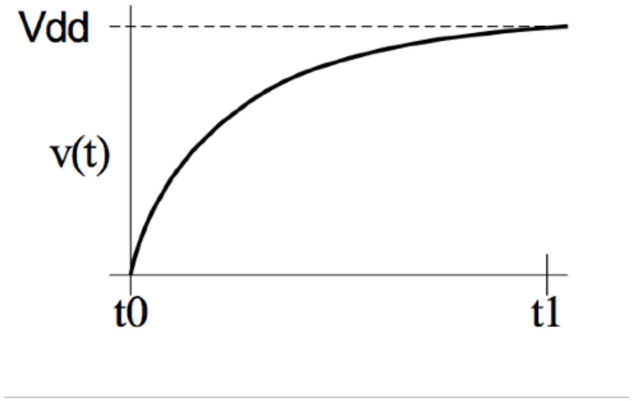
Energy dissipated for every transition (0->1 or 1->0)

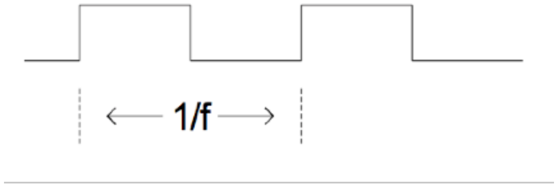
$$E_{sw} = \int_{t_0}^{t_1} P(t)dt = \int_{t_0}^{t_1} (V_{dd} - v)\cdot i(t)dt = \int_{t_0}^{t_1} (V_{dd} - v)\cdot c\,(dv/dt)\,dt =$$

$$= cV_{dd}\int_{t_0}^{t_1} dv - c\int_{t_0}^{t_1} v\cdot dv = cV_{dd}{}^2 - 1/2cV_{dd}{}^2 = \boxed{1/2\,cV_{dd}{}^2}$$

Dynamic Power







Average dynamic power of a transistor:

$P = \alpha \cdot (E / T) = \alpha \cdot E f = \frac{1}{2} \alpha C V_{dd}^2 f$

α: switch activity factor. No switching, no dynamic power consumption

Dynamic Power

$$P = k\,C\,V^2\,f$$

Dynamic Power

$$P = k\, C\, V^2 f$$

- Increasing f requires V to be increased proportionally

Dynamic Power

$$P = k\,C\,V^2\,f$$

- Increasing f requires V to be increased proportionally
  - Intuitively: a higher frequency means a shorter cycle time, which means the critical path of your processor needs to be shorter, which requires faster transistors, which you get by increasing the voltage

Dynamic Power

$$P = k\,C\,V^2\,f$$

- Increasing f requires V to be increased proportionally
    - Intuitively: a higher frequency means a shorter cycle time, which means the critical path of your processor needs to be shorter, which requires faster transistors, which you get by increasing the voltage
    - “Overclocking” just increases the clock speed without increasing voltage => machine might crash (cycle time shorter than the critical path delay)

Dynamic Power

$$P = k\,C\,V^2\,f$$

- Increasing f requires V to be increased proportionally
  - Intuitively: a higher frequency means a shorter cycle time, which means the critical path of your processor needs to be shorter, which requires faster transistors, which you get by increasing the voltage
  - “Overclocking” just increases the clock speed without increasing voltage => machine might crash (cycle time shorter than the critical path delay)
  - Corollary: reducing voltage requires reducing frequency

Dynamic Power

$$P = k\, C\, V^2 f$$

- Increasing f requires V to be increased proportionally
  - Intuitively: a higher frequency means a shorter cycle time, which means the critical path of your processor needs to be shorter, which requires faster transistors, which you get by increasing the voltage
  - "Overclocking" just increases the clock speed without increasing voltage => machine might crash (cycle time shorter than the critical path delay)
  - Corollary: reducing voltage requires reducing frequency
  - 15% reduction in voltage requires about 15% slow down in frequency

Dynamic Power

$$P = k\,C\,V^2\,f$$

- Increasing f requires V to be increased proportionally
    - Intuitively: a higher frequency means a shorter cycle time, which means the critical path of your processor needs to be shorter, which requires faster transistors, which you get by increasing the voltage
    - “Overclocking” just increases the clock speed without increasing voltage => machine might crash (cycle time shorter than the critical path delay)
    - Corollary: reducing voltage requires reducing frequency
    - 15% reduction in voltage requires about 15% slow down in frequency
    - What’s the impact on dynamic power? $0.85^3 \approx 60\%$ -> 40% dynamic power reduction.

Dynamic Power Favors Parallelisms

$$P = k\,C\,f^3$$

- Dynamic power favors parallel processing over higher clock rate

Dynamic Power Favors Parallelisms

$$P = k\,C\,f^3$$

- Dynamic power favors parallel processing over higher clock rate
  - Take a core and replicate it 4 times: 4x speedup & **4x power**

Dynamic Power Favors Parallelisms

$$P = k\,C\,f^3$$

- Dynamic power favors parallel processing over higher clock rate
  - Take a core and replicate it 4 times: 4x speedup & **4x power**
  - Take a core and clock it 4 times faster: 4x speedup but **64x dynamic power**!

Dynamic Power Favors Parallelisms

$$P = k\,C\,f^3$$

- Dynamic power favors parallel processing over higher clock rate
  - Take a core and replicate it 4 times: 4x speedup & **4x power**
  - Take a core and clock it 4 times faster: 4x speedup but **64x dynamic power**!
- Another way to think about this

Dynamic Power Favors Parallelisms

$$P = k\,C\,f^3$$

- Dynamic power favors parallel processing over higher clock rate
    - Take a core and replicate it 4 times: 4x speedup & **4x power**
    - Take a core and clock it 4 times faster: 4x speedup but **64x dynamic power**!
- Another way to think about this
    - If a task can be perfectly parallelized by 4 cores, we can reduce the clock frequency of each core to 1/4 while retaining the same performance

Dynamic Power Favors Parallelisms

$$P = k\,C\,f^3$$

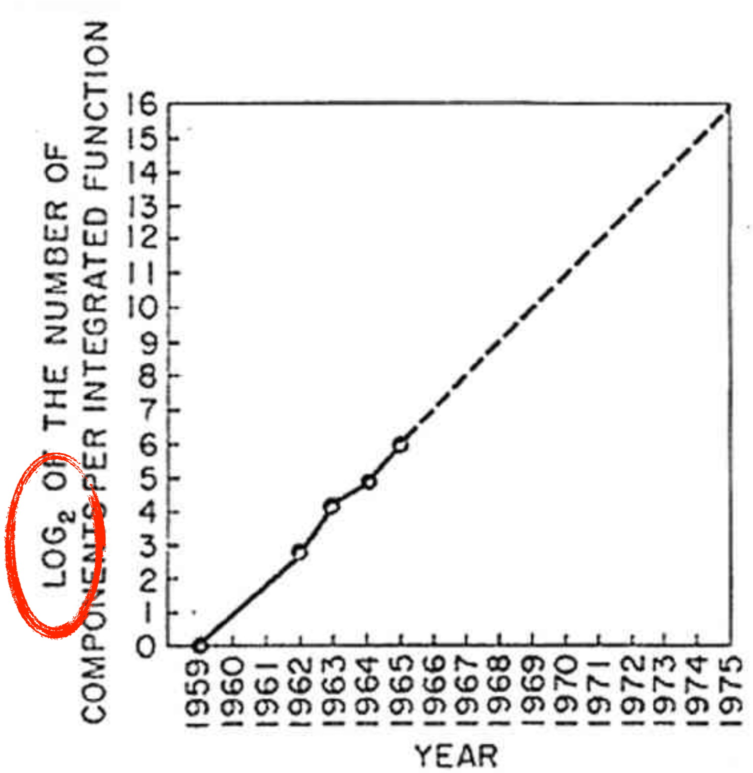
- Dynamic power favors parallel processing over higher clock rate
    - Take a core and replicate it 4 times: 4x speedup & **4x power**
    - Take a core and clock it 4 times faster: 4x speedup but **64x dynamic power**!
- Another way to think about this
    - If a task can be perfectly parallelized by 4 cores, we can reduce the clock frequency of each core to 1/4 while retaining the same performance
    - Dynamic power becomes $4 \times (1/4)^3 = 1/16$

Moore’s Law

- Gordon Moore in 1965 predicted that the number of transistors doubles every year

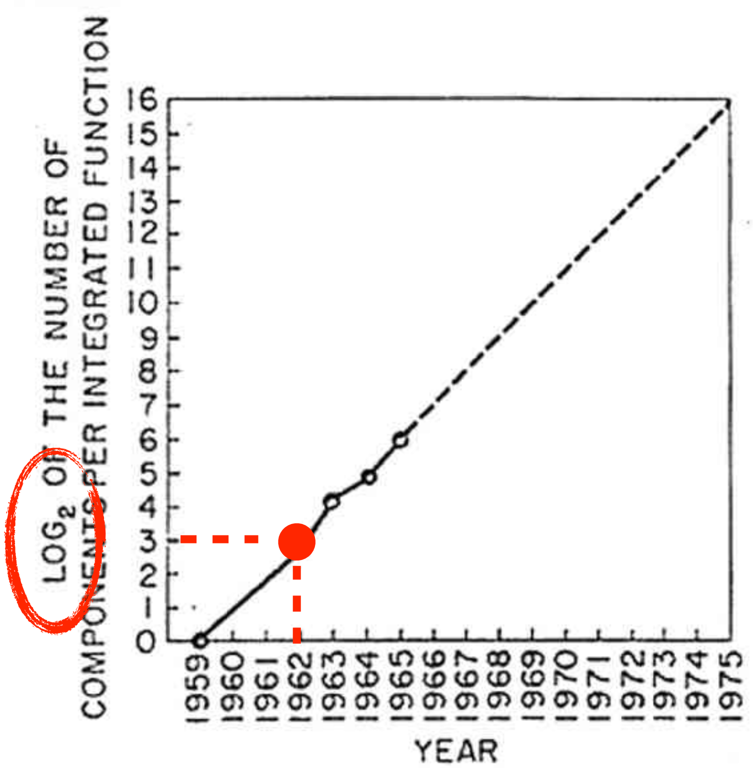
Moore’s Law

- Gordon Moore in 1965 predicted that the number of transistors doubles every year



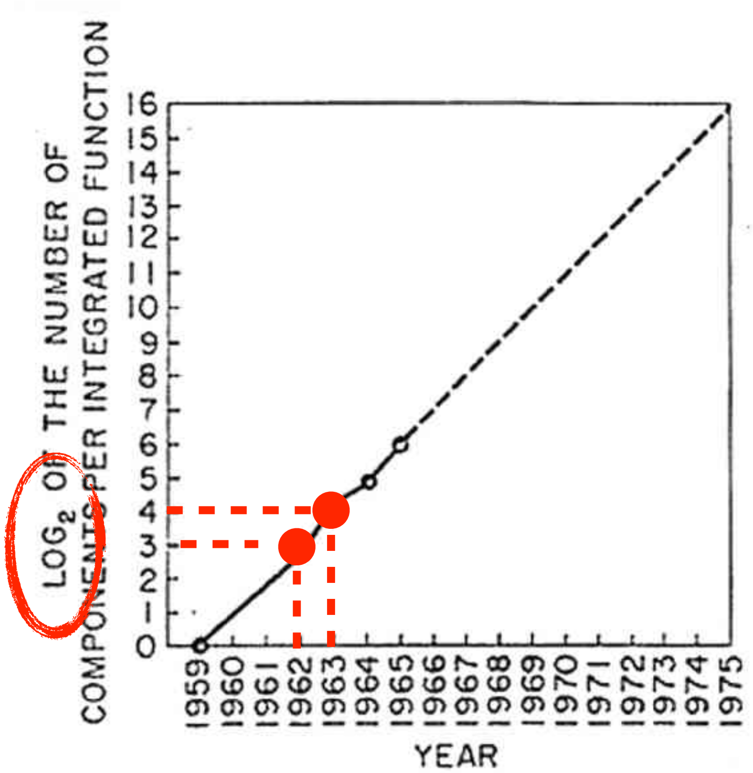
Moore’s Law

- Gordon Moore in 1965 predicted that the number of transistors doubles every year



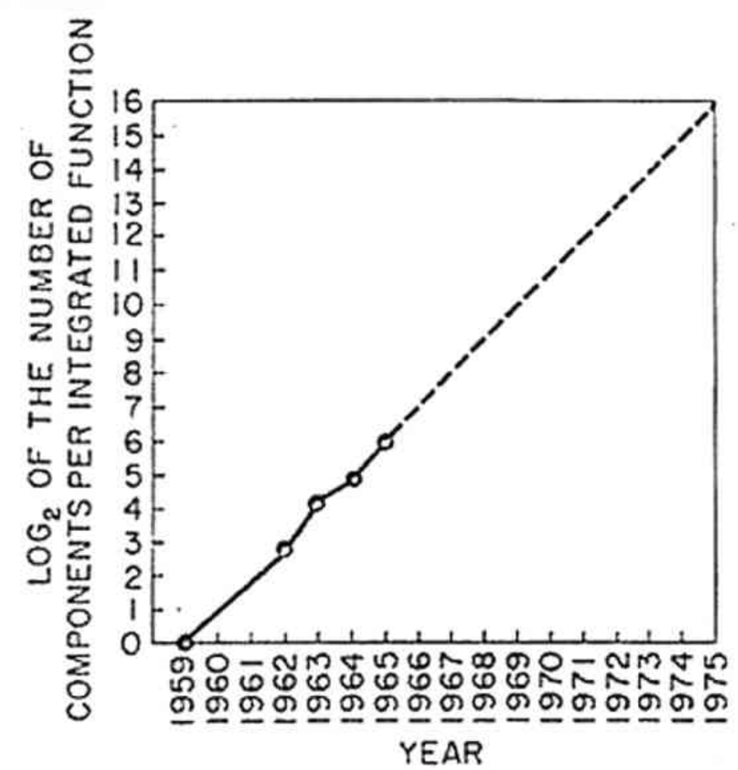
Moore’s Law

- Gordon Moore in 1965 predicted that the number of transistors doubles every year



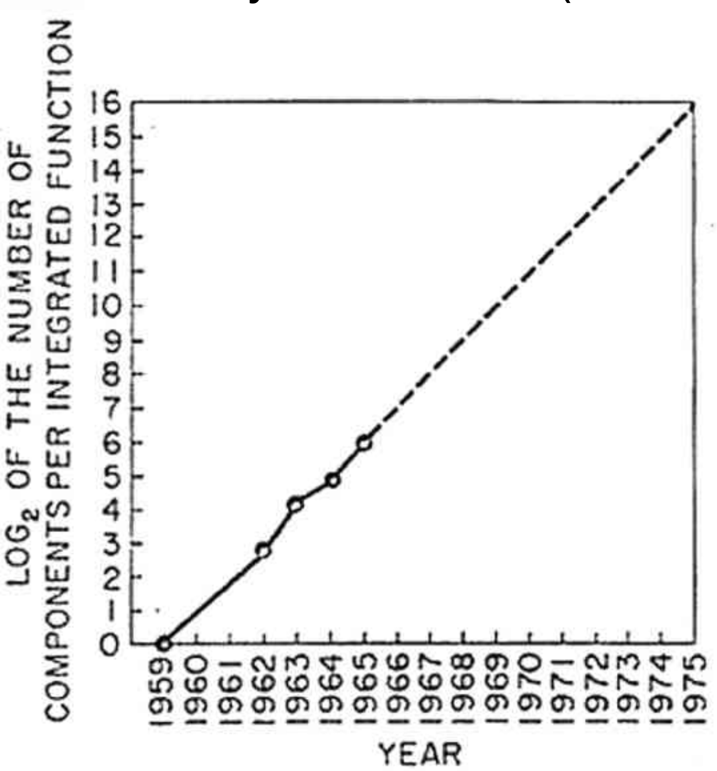
Moore’s Law

- Gordon Moore in 1965 predicted that the number of transistors doubles every year
- In 1975 he revised the prediction to doubling every 2 years

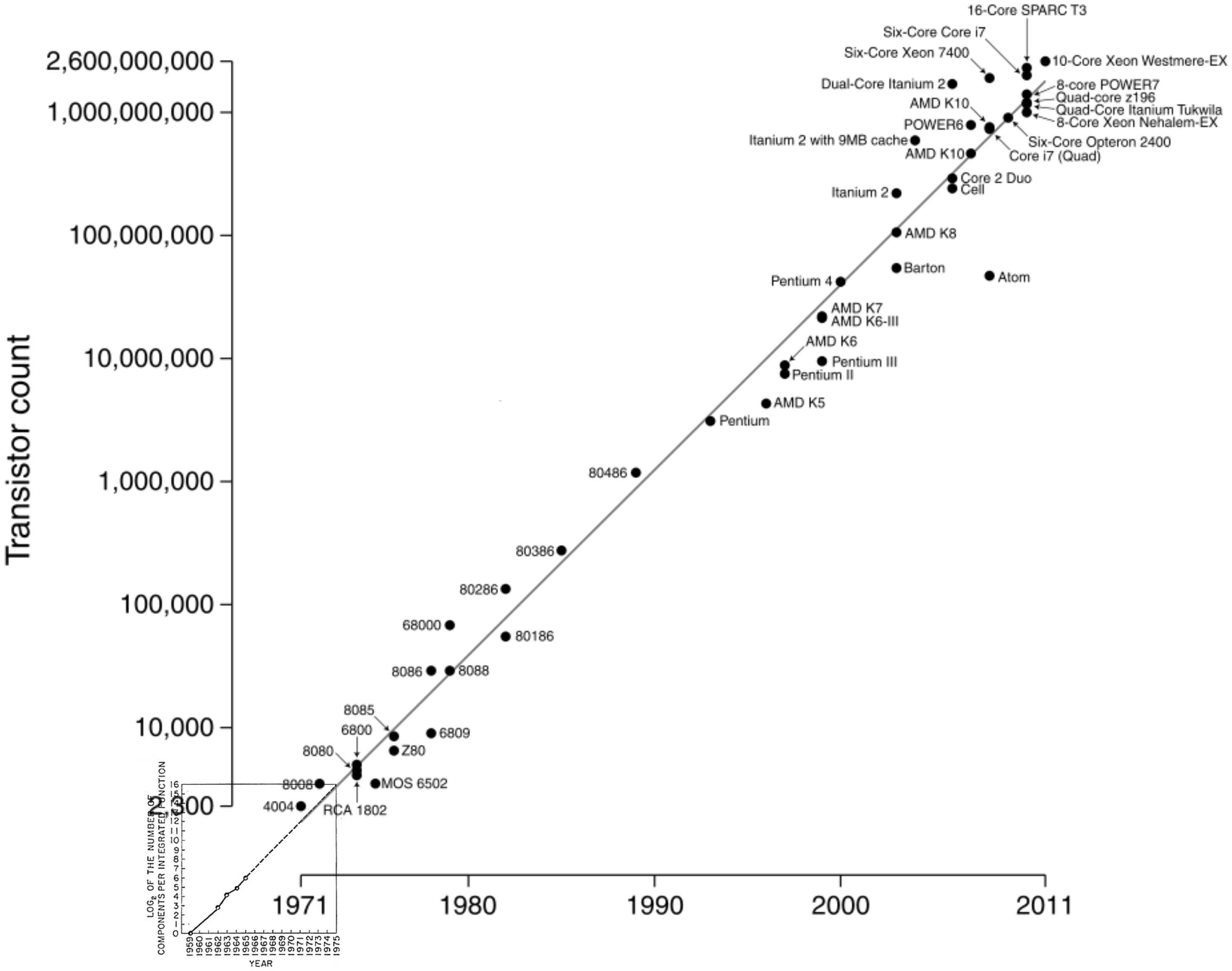


Moore’s Law

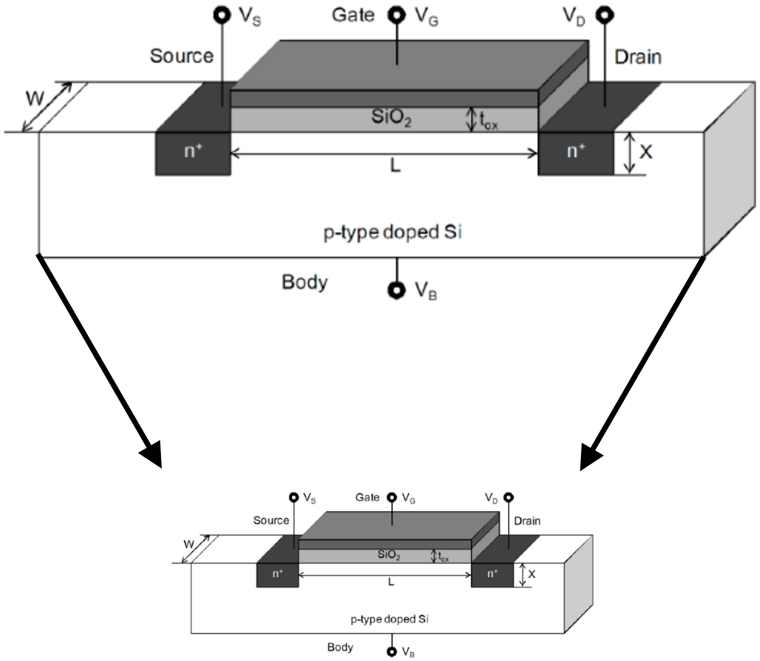
- Gordon Moore in 1965 predicted that the number of transistors doubles every year
- In 1975 he revised the prediction to doubling every 2 years
- Today’s widely-known Moore’s Law: number of transistors double about every 18 months (Moore never used the number 18…)



Moore’s Law


Transistor count
2,600,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
2,300
1971
1980
1990
2000
2011
16-Core SPARC T3
Six-Core Core i7
Six-Core Xeon 7400
10-Core Xeon Westmere-EX
Dual-Core Itanium 2
8-core POWER7
Quad-core z196
Quad-Core Itanium Tukwila
8-Core Xeon Nehalem-EX
AMD K10
POWER6
Itanium 2 with 9MB cache
Six-Core Opteron 2400
Core i7 (Quad)
AMD K10
Core 2 Duo
Cell
Itanium 2
AMD K8
Barton
Atom
Pentium 4
AMD K7
AMD K6-III
AMD K6
Pentium III
Pentium II
AMD K5
Pentium
80486
80386
80286
68000
80186
8086
8088
8085
6800
6809
Z80
8080
8008
MOS 6502
4004
RCA 1802
LOG2 OF THE NUMBER OF COMPONENTS PER INTEGRATED FUNCTION
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
YEAR

Dennard Scaling

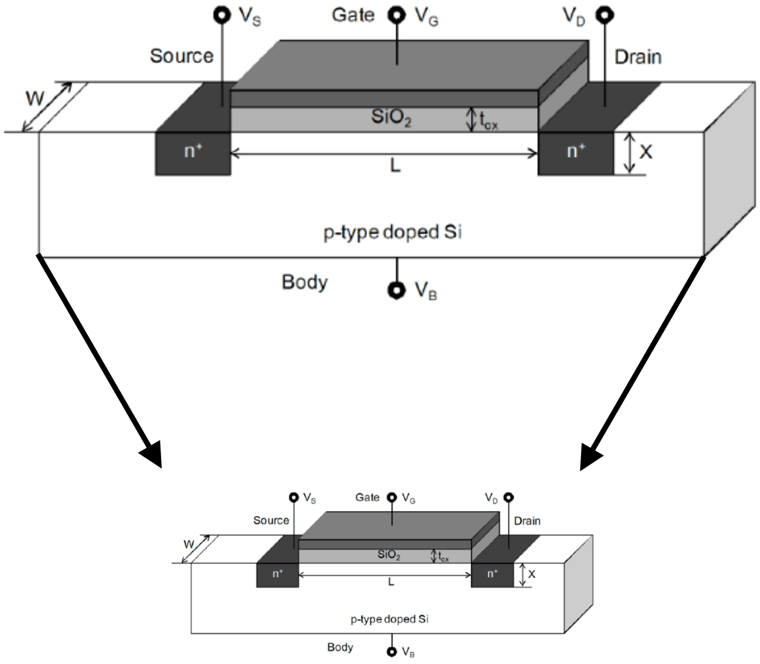


Scale factor α*<1*

$\alpha = 0.7$ => 2X more transistors!

Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. Proc. of IEEE. Dennart et al. 1974.

Dennard Scaling



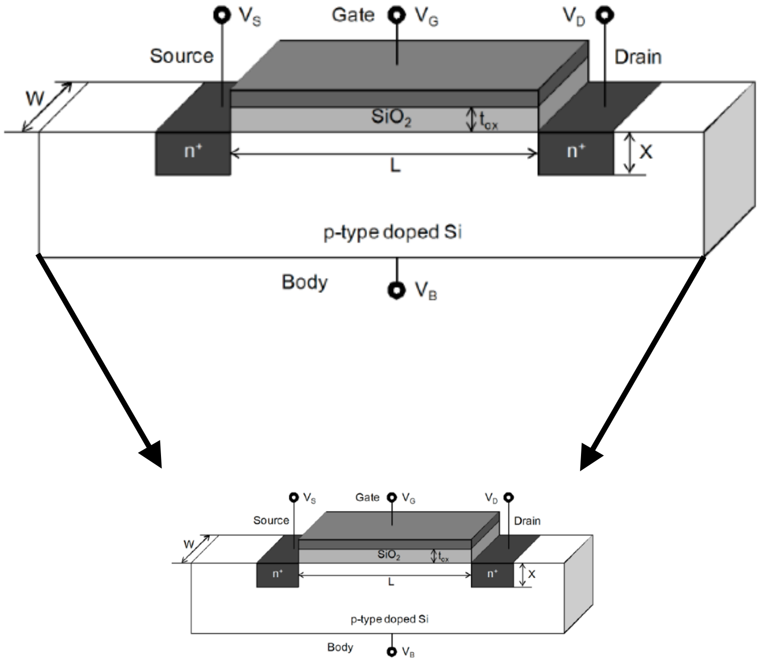
Parameter	**Value**	**Scaled Value**
Dopant concentrations	Na, Nd	Na/α, Nd/α
Dimensions	L, W, Tox	αL, αW, αTox
Field	E	E
Voltage	V	αV
Capacitance	C	αC
Current	I	αI

Scale factor α*<1*

$\alpha = 0.7$ => 2X more transistors!

Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. Proc. of IEEE. Dennart et al. 1974.

Dennard Scaling



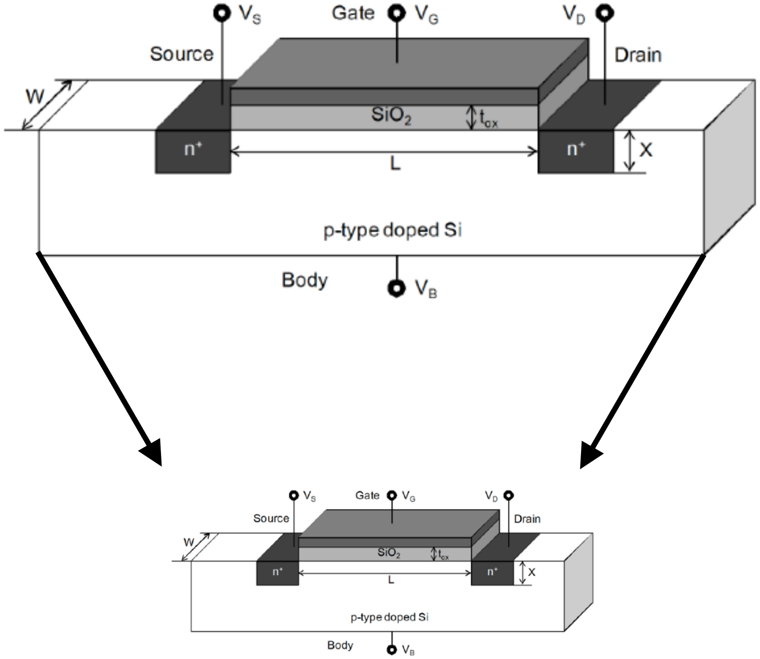
Parameter	Value	Scaled Value
Dopant concentrations	Na, Nd	Na/α, Nd/α
Dimensions	L, W, Tox	αL, αW, αTox
Field	E	E
Voltage	V	αV
Capacitance	C	αC
Current	I	αI
Transistors/Area	d	d/α^2

Scale factor α<1

α = 0.7 => 2X more transistors!

Design of Ion-Implanted MOSFET’s with Very Small Physical Dimensions. Proc. of IEEE. Dennart et al. 1974.

Dennard Scaling



Parameter	Value	Scaled Value
Dopant concentrations	Na, Nd	Na/α, Nd/α
Dimensions	L, W, Tox	αL, αW, αTox
Field	E	E
Voltage	V	αV
Capacitance	C	αC
Current	I	αI

Transistors/Area	d	d/α^2

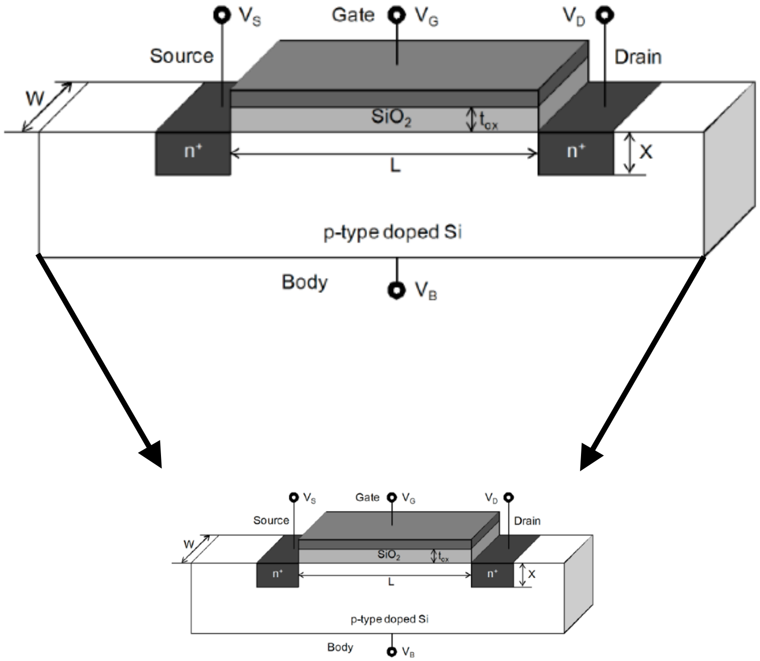
Propagation time (~CV/I)	t	αt
Frequency (1/t)	f	f/α

Scale factor $\alpha<1$

$\alpha = 0.7$ => 2X more transistors!

Design of Ion-Implanted MOSFET’s with Very Small Physical Dimensions. Proc. of IEEE. Dennart et al. 1974.

Dennard Scaling



Scale factor $\alpha<1$

$\alpha = 0.7$ => 2X more transistors!

Parameter	**Value**	**Scaled Value**
Dopant concentrations	Na, Nd	Na/α, Nd/α
Dimensions	L, W, Tox	αL, αW, αTox
Field	E	E
Voltage	V	αV
Capacitance	C	αC
Current	I	αI
Transistors/Area	d	d/α^2
Propagation time (~CV/I)	t	αt
Frequency (1/t)	f	f/α
Power (CV^2f)	P	$\alpha^2 P$
Power/area (Power density)	**Pd**	**Pd**

Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. Proc. of IEEE. Dennart et al. 1974.

Implications of Dennard Scaling and Moore’s Law

- Each new processor generation can have more transistors (Moore’s Law), and will run at higher frequency but won’t consume more power under the same area budget.

Implications of Dennard Scaling and Moore’s Law

- Each new processor generation can have more transistors (Moore’s Law), and will run at higher frequency but won’t consume more power under the same area budget.
- More transistors means better microarchitecture, which leads to better performance even under the same frequency.

Implications of Dennard Scaling and Moore’s Law

- Each new processor generation can have more transistors (Moore’s Law), and will run at higher frequency but won’t consume more power under the same area budget.
- More transistors means better microarchitecture, which leads to better performance even under the same frequency.
- Higher frequency means better performance even under the same microarchitecture.

Implications of Dennard Scaling and Moore’s Law

- Each new processor generation can have more transistors (Moore’s Law), and will run at higher frequency but won’t consume more power under the same area budget.
- More transistors means better microarchitecture, which leads to better performance even under the same frequency.
- Higher frequency means better performance even under the same microarchitecture.
- Overall, software gets a free ride: wait for the next generation of hardware and performance will naturally increase without consuming more power.

Implications of Dennard Scaling and Moore’s Law

- Each new processor generation can have more transistors (Moore’s Law), and will run at higher frequency but won’t consume more power under the same area budget.
- More transistors means better microarchitecture, which leads to better performance even under the same frequency.
- Higher frequency means better performance even under the same microarchitecture.
- Overall, software gets a free ride: wait for the next generation of hardware and performance will naturally increase without consuming more power.

> Moore’s law gave us more transistors;
>
> Dennard scaling made them useful.
>
> Bob Colwell, DAC 2013, June 4, 2013

2005: End of Dennard Scaling

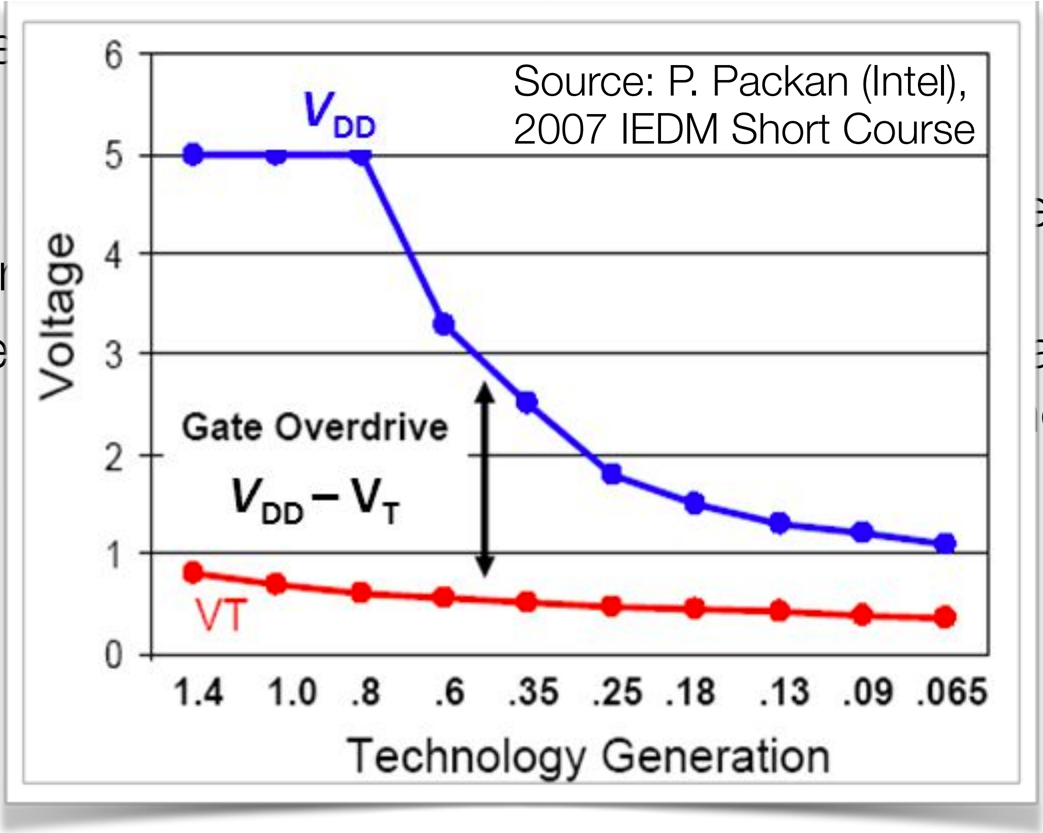
- What Happened?
  - Supply voltage V_{dd} stops scaling (Can’t drop voltage below ~1 V)
  - Remember Power = CV^2f

2005: End of Dennard Scaling

- What Happened?
  - Supply voltage V_{dd} stops scaling (Can't drop voltage below ~1 V)
  - Remember Power = CV^2f
- Why?
  - There is a fundamental limit as to how much voltage we need to switch a transistor, called threshold voltage (V_{th}).
  - V_{th} stopped scaling because leakage power/reliability/variation becomes huge issues, and accordingly V_{dd} stops scaling

2005: End of Dennard Scaling

- What Happened?
  - Supply voltage V_{dd} stops scaling (Can’t drop voltage below ~1 V)
  - Remembe
- Why?
  - There is a e need to switch a tr
  - V_{th} stoppe ariation becomes g

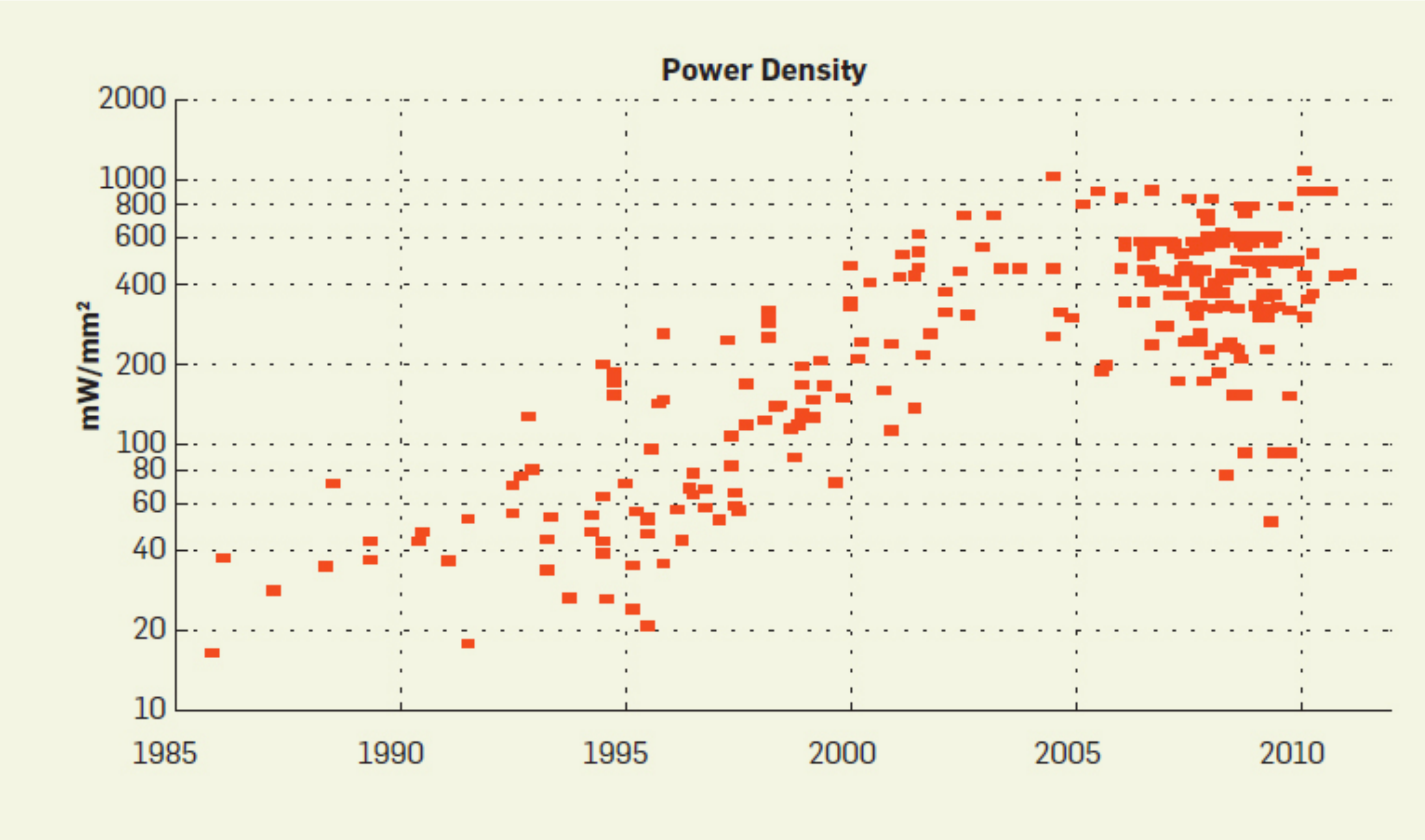


Source: P. Packan (Intel), 2007 IEDM Short Course

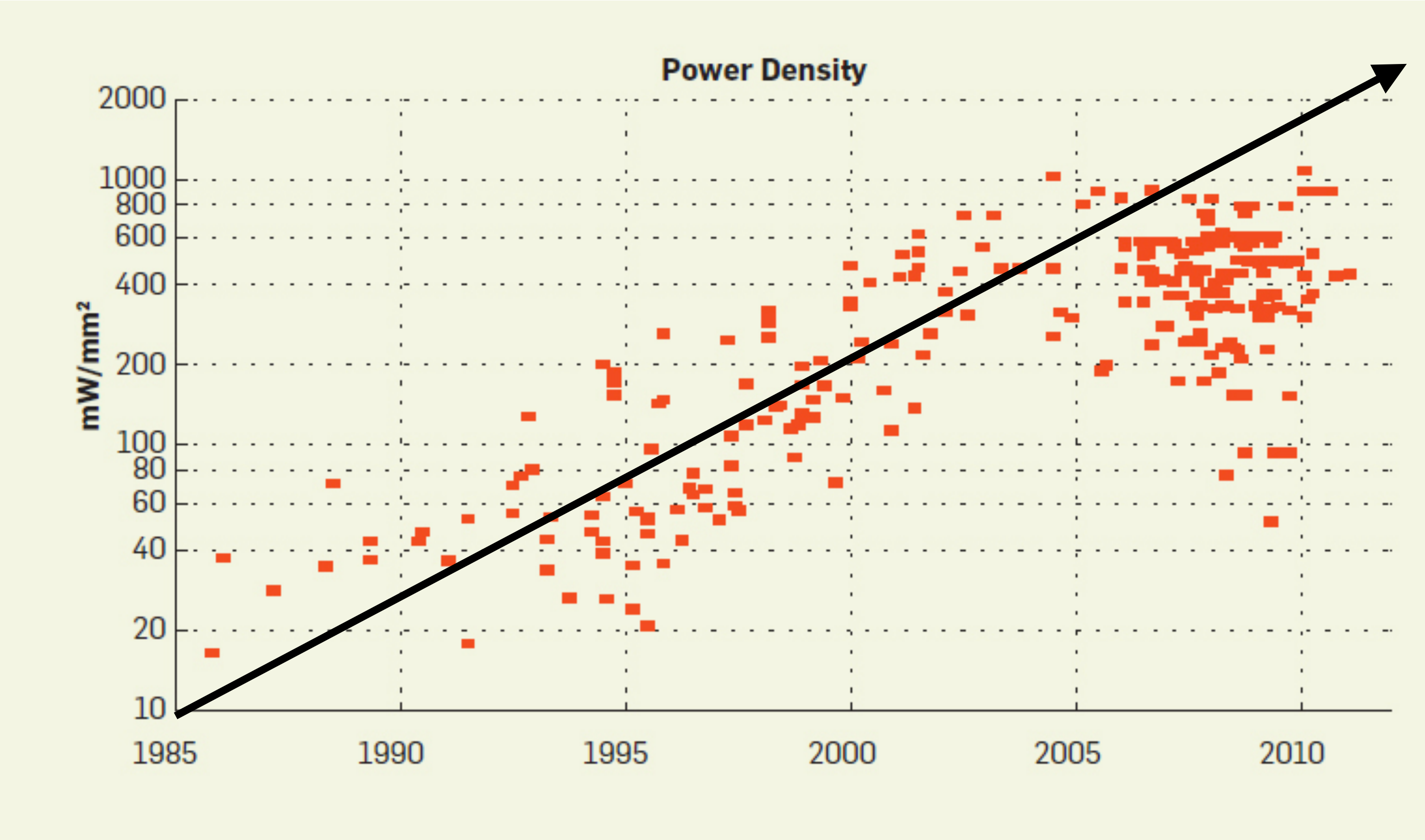
2005: End of Dennard Scaling

- What Happened?
  - Supply voltage V_{dd} stops scaling (Can’t drop voltage below ~1 V)
  - Remember Power = CV^2f
- Why?
  - There is a fundamental limit as to how much voltage we need to switch a transistor, called threshold voltage (V_{th}).
  - V_{th} stopped scaling because leakage power/reliability/variation becomes huge issues, and accordingly V_{dd} stops scaling
- The demise of Dennard Scaling means the power density (power consumption per unit area) will increase rather than staying stable.

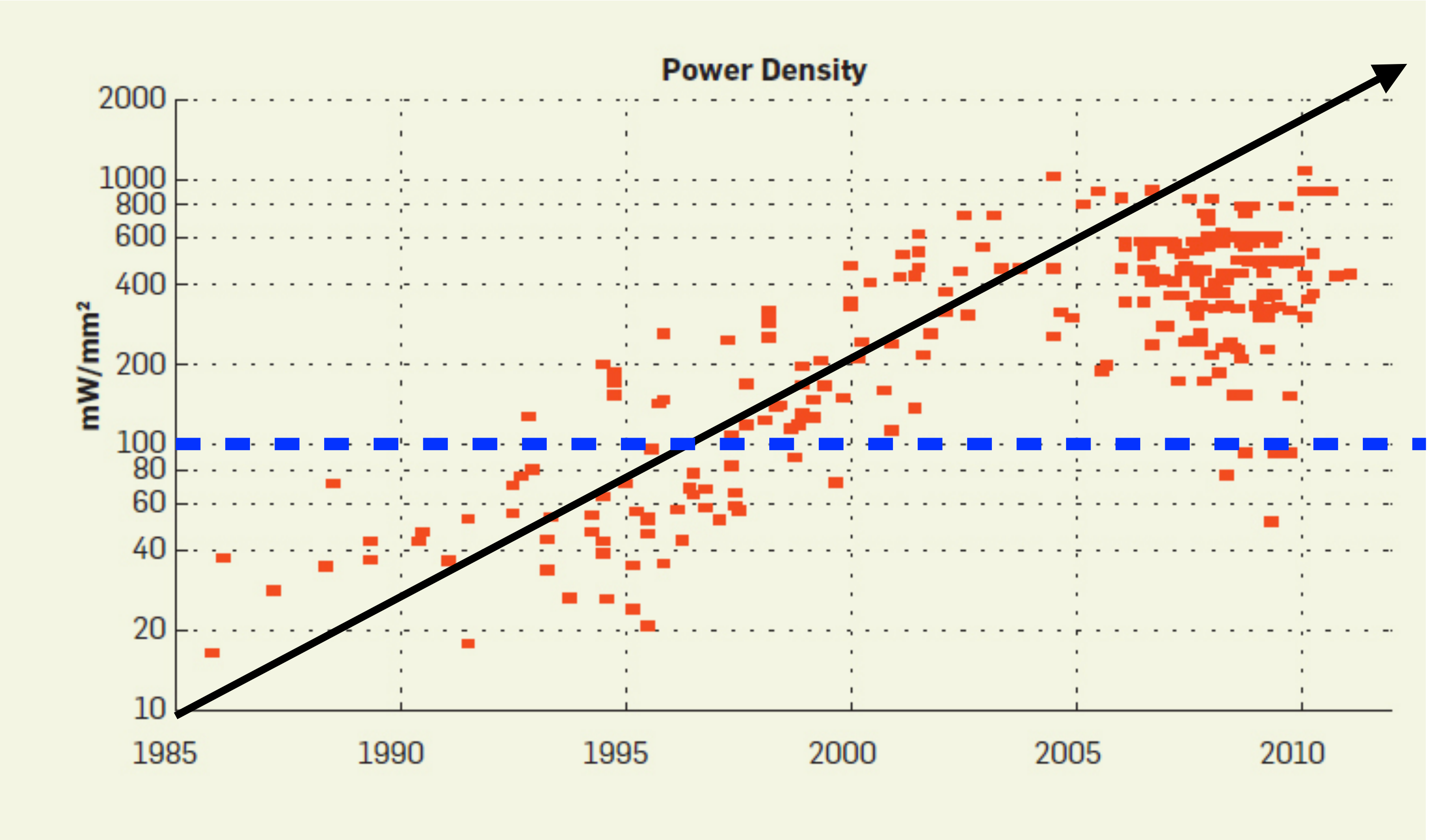
2005: End of Dennard Scaling



2005: End of Dennard Scaling

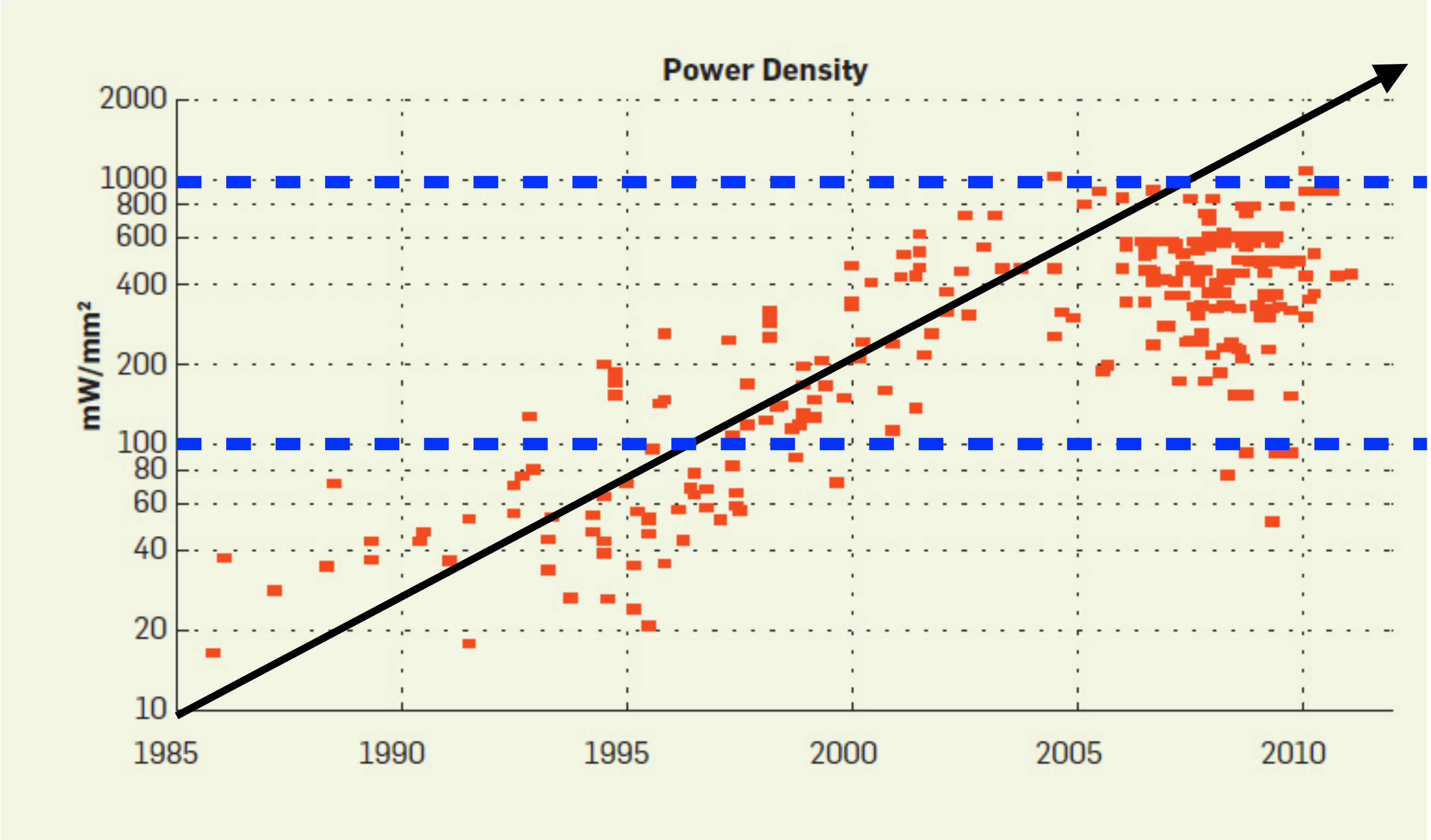


2005: End of Dennard Scaling



Hot Plate

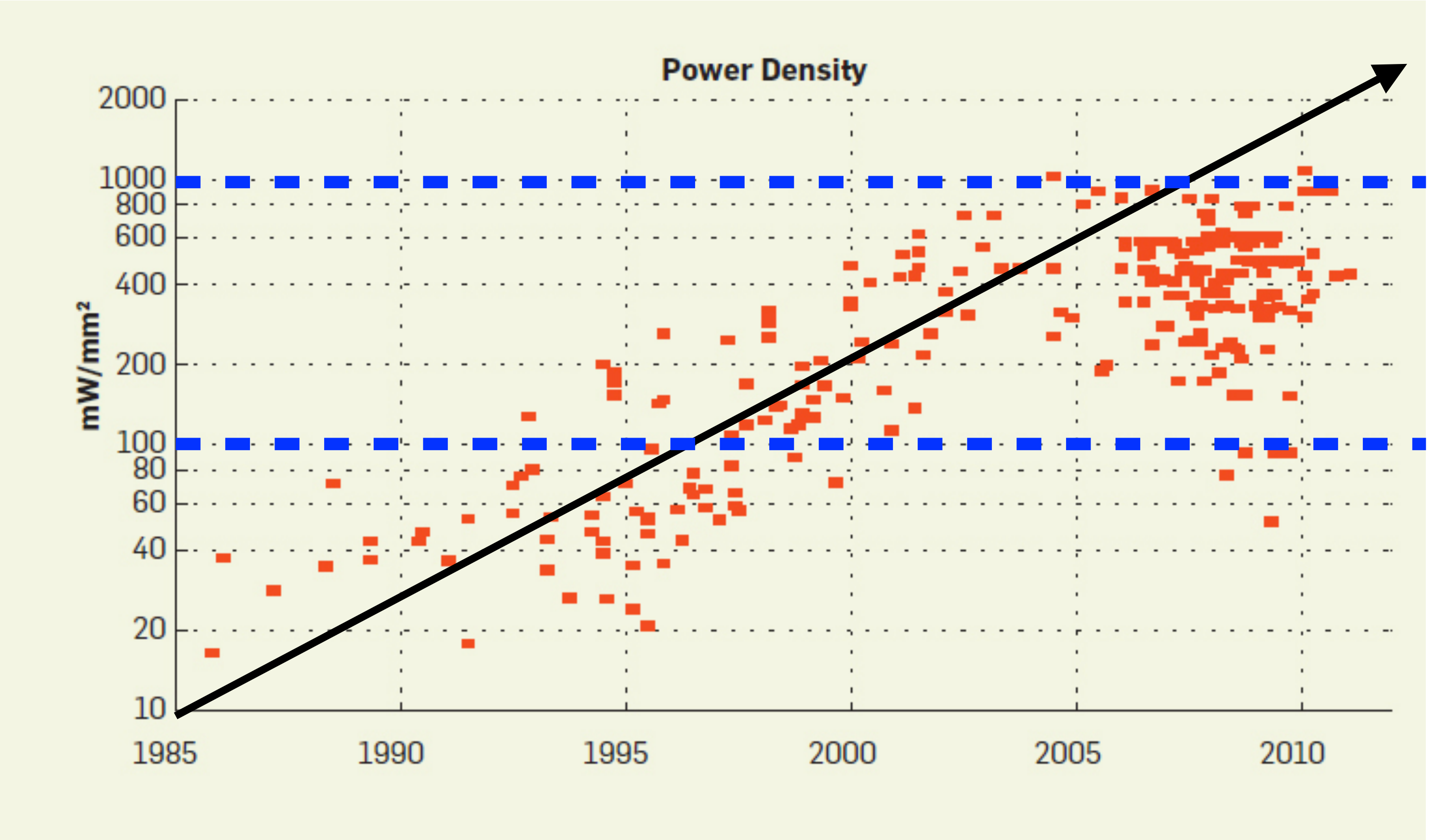
2005: End of Dennard Scaling



Nuclear Reactor

Hot Plate

2005: End of Dennard Scaling



Sun Surface (~10^4)

Nuclear Reactor

Hot Plate

Dark Silicon

n. [därk, sĭl′ĭ-kən, -kŏn′]

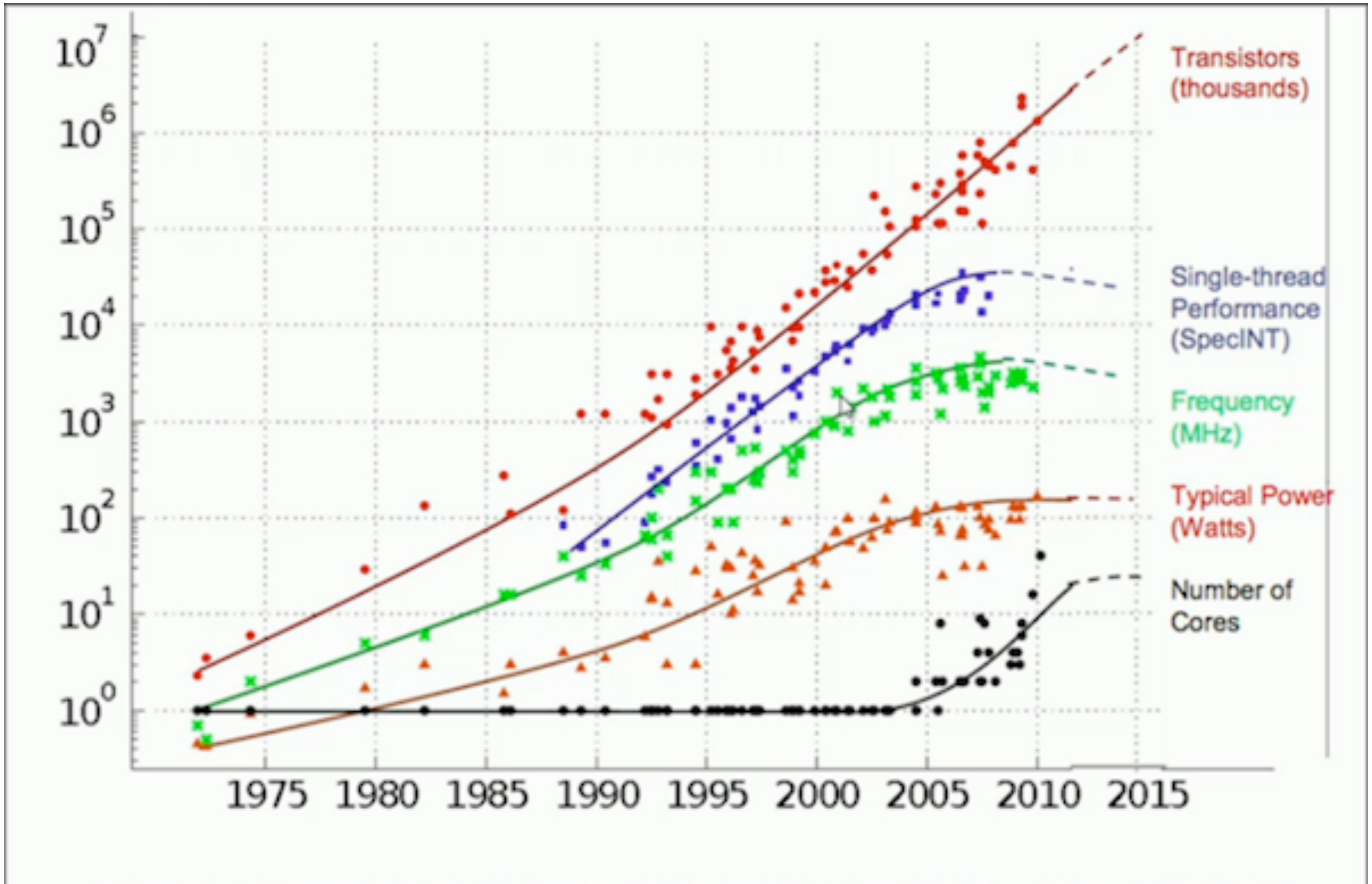
More transistors on chip (due to Moore’s Law), but a growing fraction cannot actually be used due to power limits (due to the end of Dennard Scaling).

2005: End of Dennard Scaling

- Initial response has been to lower frequency and increase cores / chip

2005: End of Dennard Scaling

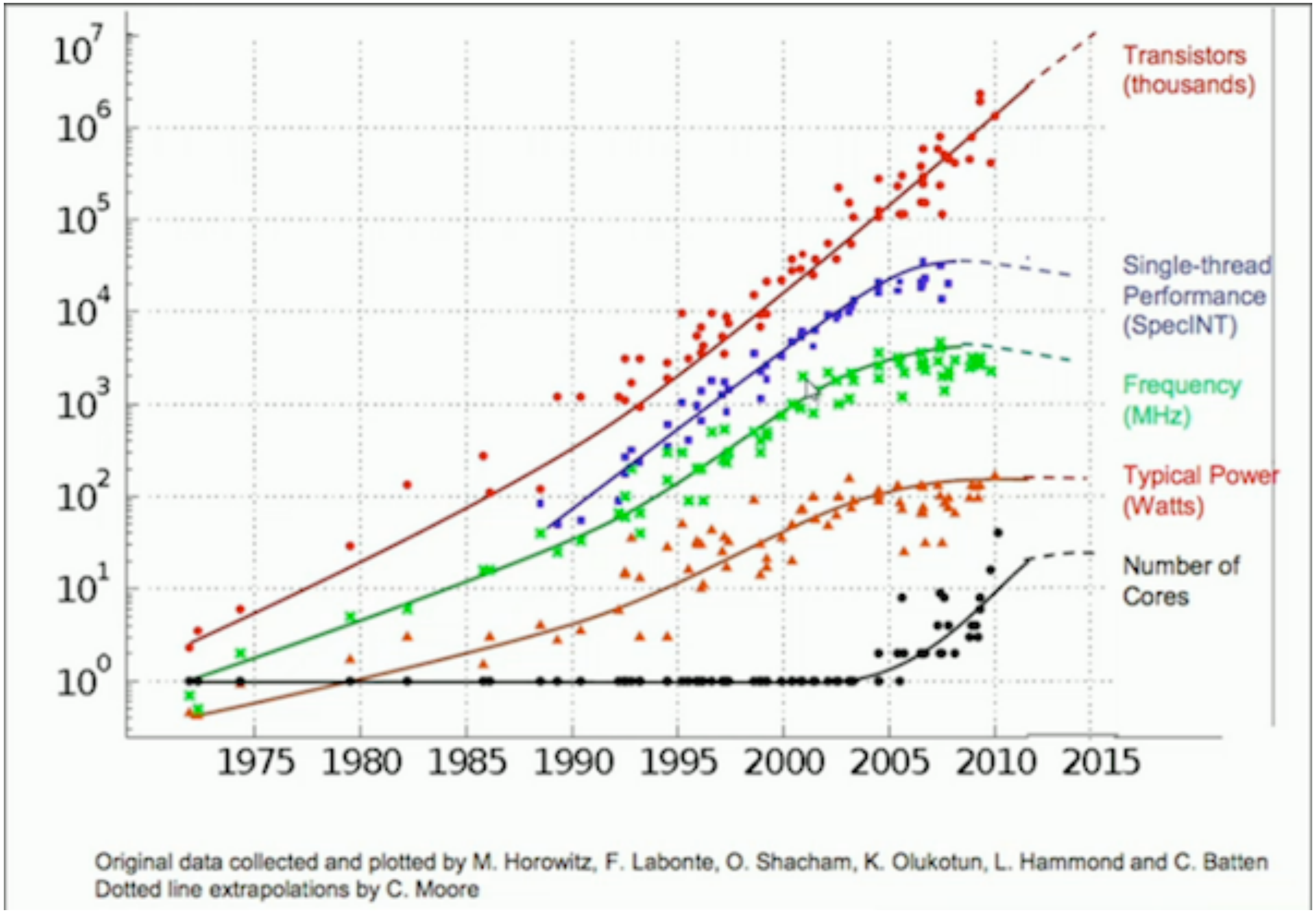
- Initial response has been to lower frequency and increase cores / chip



Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

2005: End of Dennard Scaling

- Initial response has been to lower frequency and increase cores / chip
- There is a limit to core scaling. Why?



Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

2007: A Revolutionary New Computer

OPINION

No Moore's Law for batteries

Fred Schlachter[1]

American Physical Society, Washington, DC 20045

The public has become accustomed to rapid progress in mobile phone technology, computers, and access to information; tablet computers, smart phones, and other powerful new devices are familiar to most people on the planet.

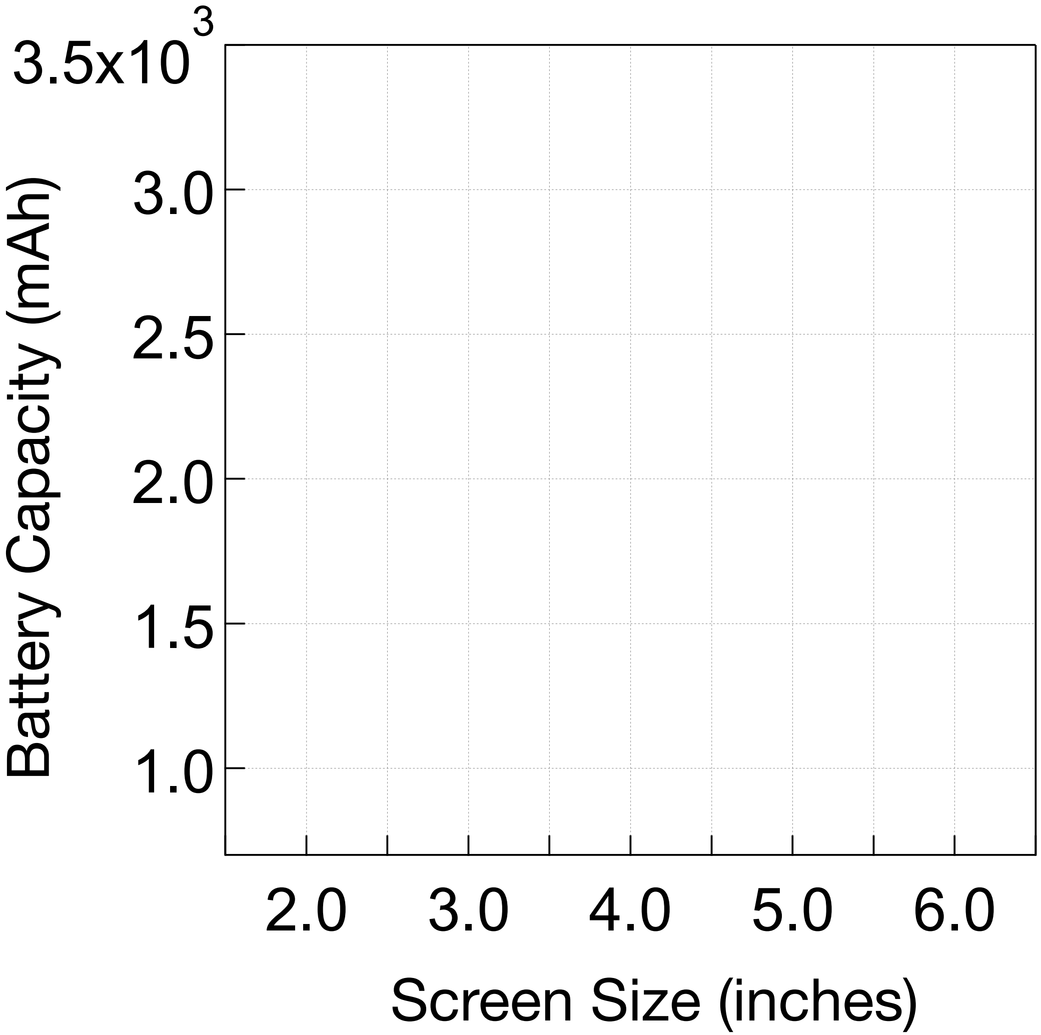
These developments are due in part to the ongoing exponential increase in computer processing power, doubling approximately every 2 years for the past several decades. This pattern is usually called Moore's Law and is named for Gordon Moore, a cofounder of Intel. The law is not a law like that for gravity; it is an empirical observation, which has become a self-fulfilling prophecy. Unfortunately, much of the public has come to expect that all technology does, will, or should follow such a law, which is not consistent with our everyday observations: For example, the maximum there is a Moore's Law for computer processors is that electrons are small and they do not take up space on a chip. Chip performance is limited by the lithography technology used to fabricate the chips; as lithography improves ever smaller features can be made on processors. Batteries are not like this. Ions, which transfer charge in batteries, are large, and they take up space, as do anodes, cathodes, and electrolytes. A D-cell battery stores more energy than an AA-cell. Potentials in a battery are dictated by the relevant chemical reactions, thus limiting eventual battery performance. Significant improvement in battery capacity can only be made by changing to a different chemistry.

Scientists and battery experts, who have been optimistic in the recent past about improving lithium-ion batteries and about de-

“Improving” Energy Capacity

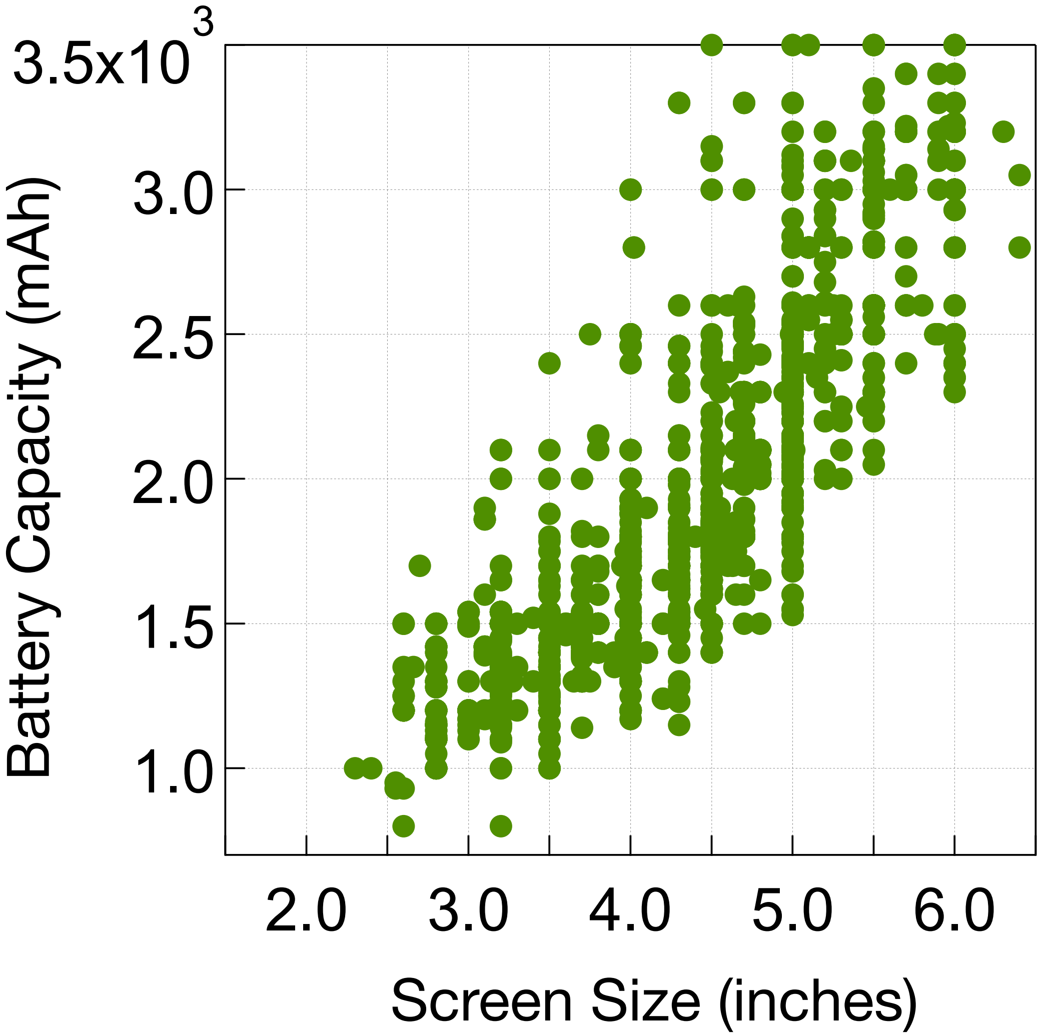
600 smartphone from 2006 to 2014 on http://www.gsmarena.com/makers.php3

“Improving” Energy Capacity



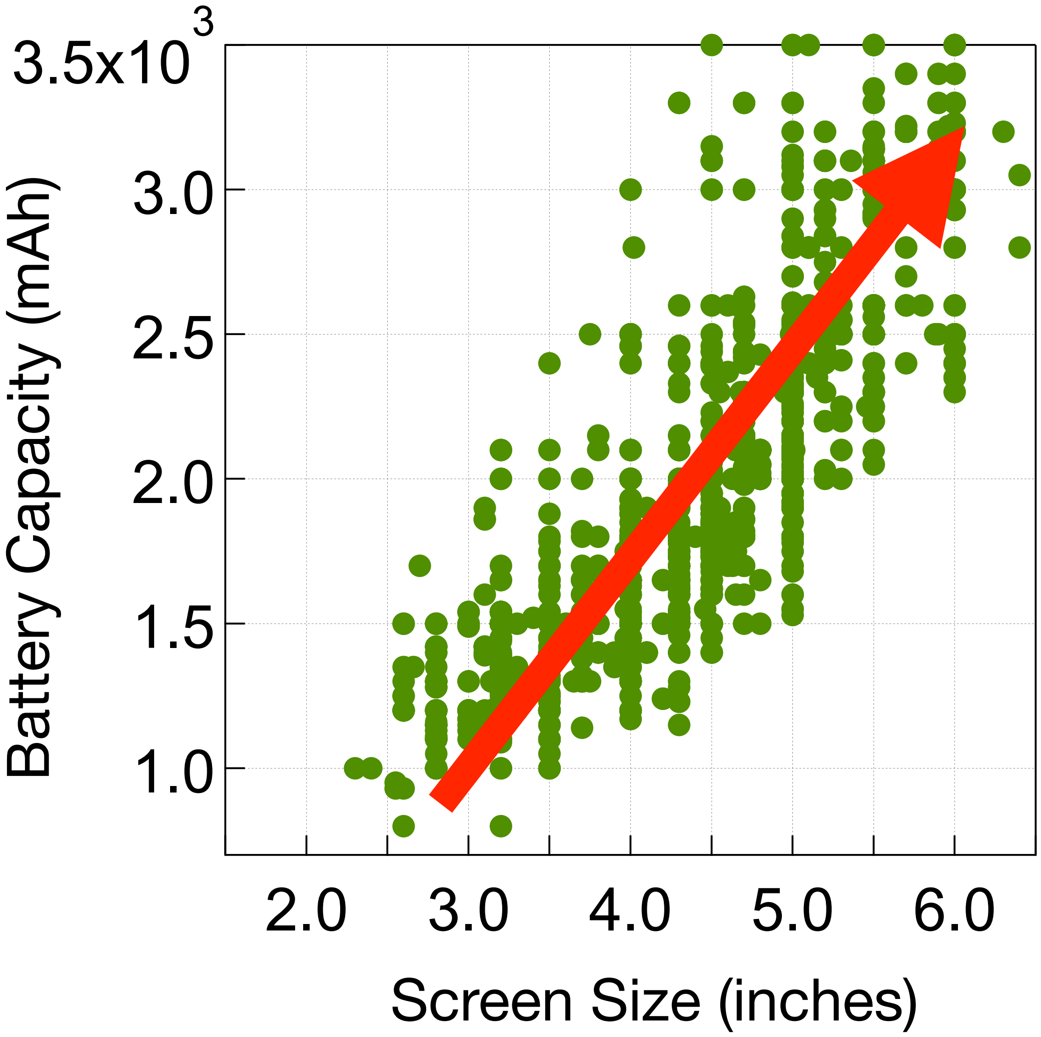
600 smartphone from 2006 to 2014 on http://www.gsmarena.com/makers.php3

“Improving” Energy Capacity



600 smartphone from 2006 to 2014 on http://www.gsmarena.com/makers.php3

“Improving” Energy Capacity



600 smartphone from 2006 to 2014 on http://www.gsmarena.com/makers.php3

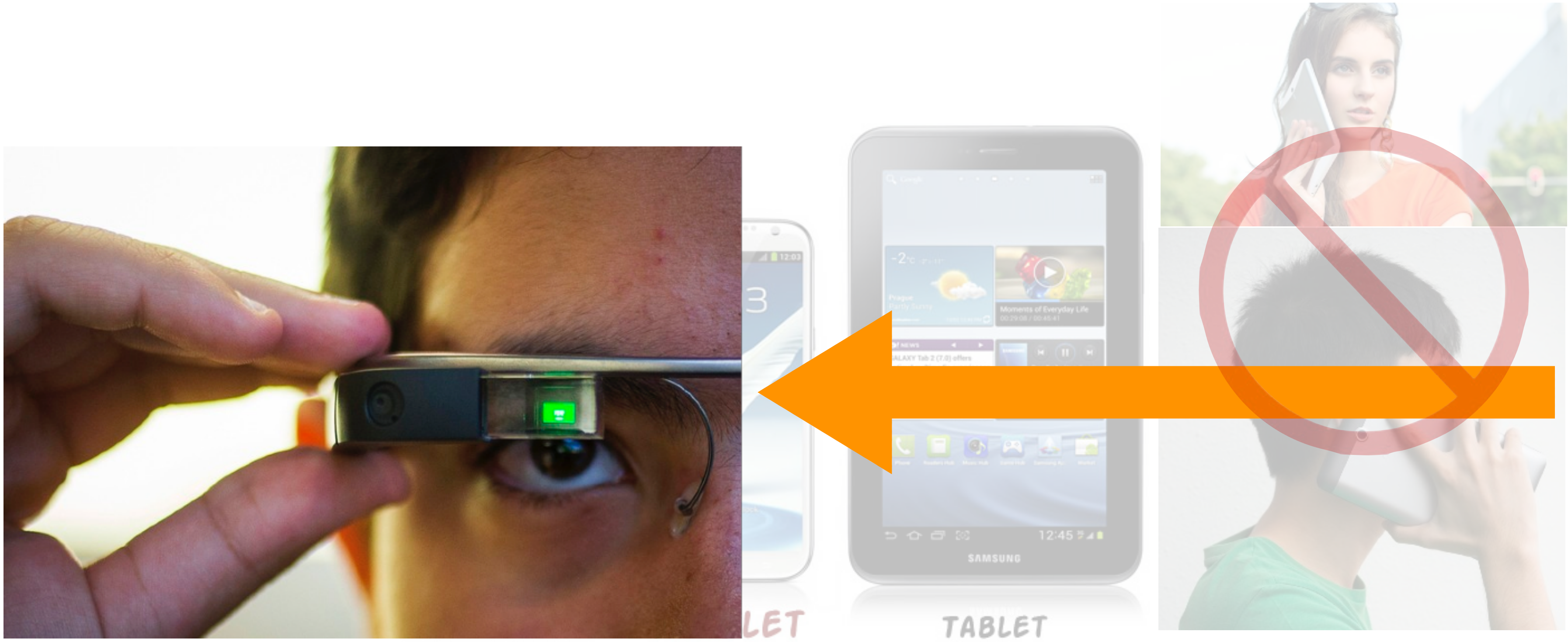
“Improving” Energy Capacity

“Improving” Energy Capacity

“Improving” Energy Capacity

“Improving” Energy Capacity

“Improving” Energy Capacity



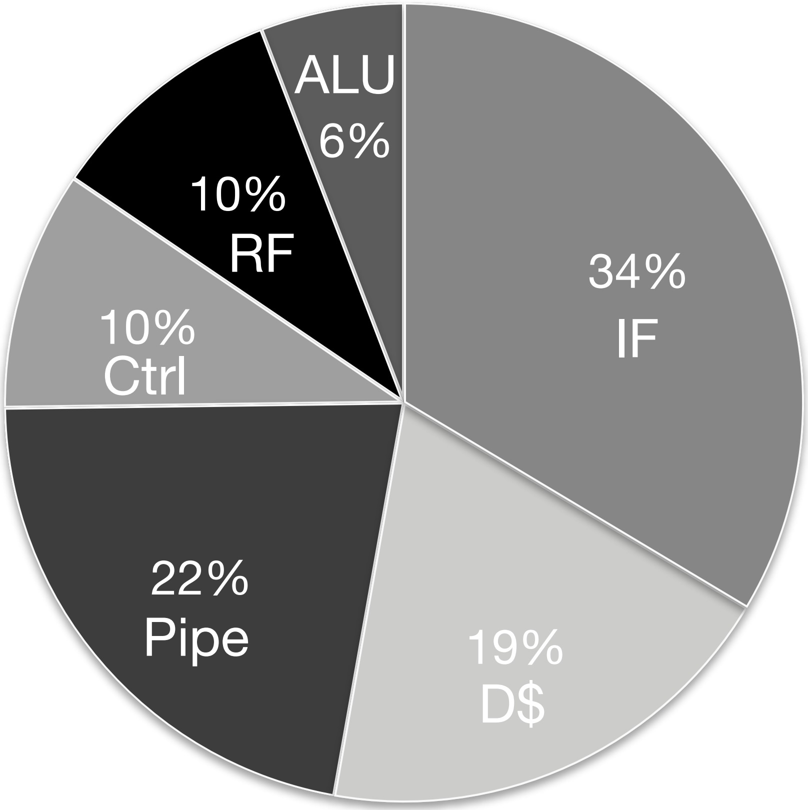
Sources of Energy-Inefficiencies

General-Purpose CPU = Instruction Delivery + Data Feeding + Execution + Control, where instruction delivery, data feeding & control are pure overhead

Understanding sources of inefficiency in general-purpose chips, Hameed et al., ISCA 2010

Sources of Energy-Inefficiencies

General-Purpose CPU = Instruction Delivery + Data Feeding + Execution + Control, where instruction delivery, data feeding & control are pure overhead



IF: Instruction fetch

Ctrl: Other control logics

Pipe: Pipeline reg, bus, clock

D$: Data cache

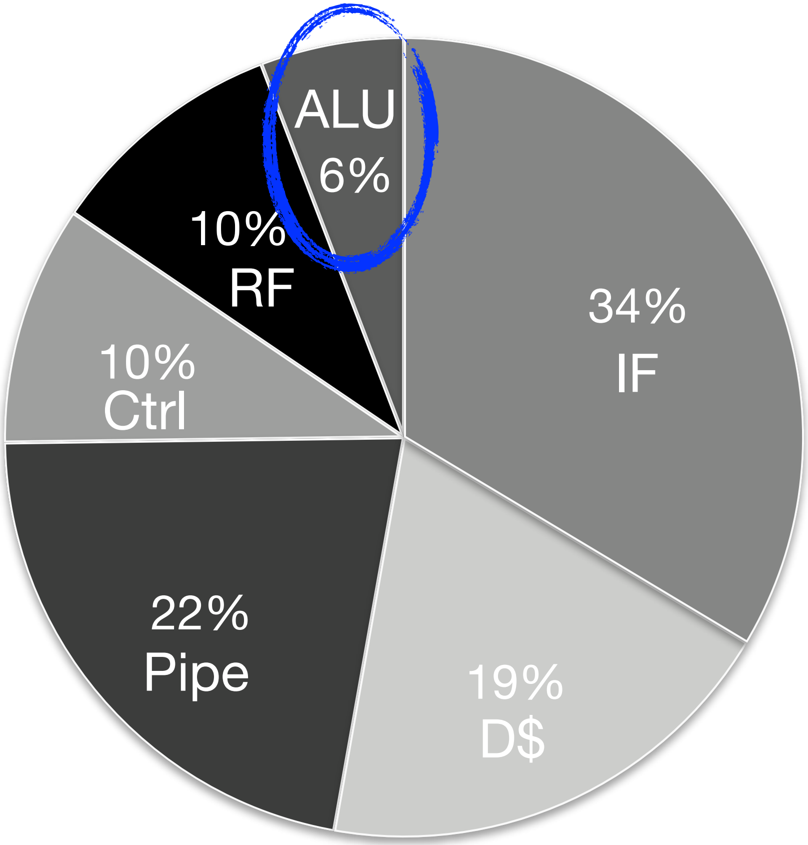
RF: Register file

ALU: Functional units

Understanding sources of inefficiency in general-purpose chips, Hameed et al., ISCA 2010

Sources of Energy-Inefficiencies

General-Purpose CPU = Instruction Delivery + Data Feeding + Execution + Control, where instruction delivery, data feeding & control are pure overhead



IF: Instruction fetch

Ctrl: Other control logics

Pipe: Pipeline reg, bus, clock

D$: Data cache

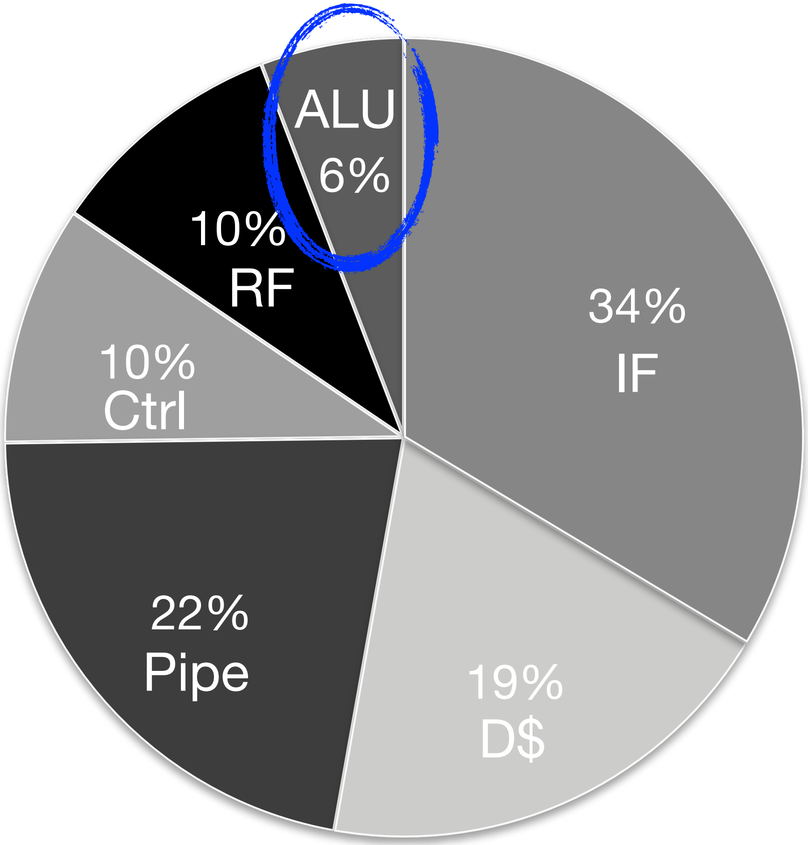
RF: Register file

ALU: Functional units

Understanding sources of inefficiency in general-purpose chips, Hameed et al., ISCA 2010

Sources of Energy-Inefficiencies

General-Purpose CPU = Instruction Delivery + Data Feeding + Execution + Control, where instruction delivery, data feeding & control are pure overhead



Pure Overhead

IF: Instruction fetch

Ctrl: Other control logics

Pipe: Pipeline reg, bus, clock

D$: Data cache

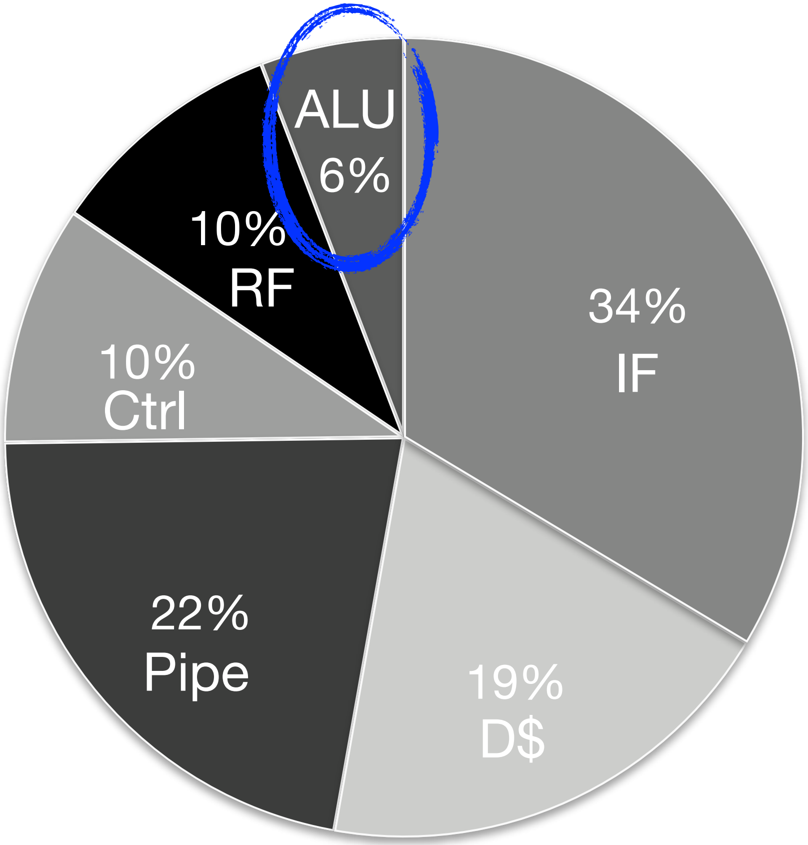
RF: Register file

ALU: Functional units

Understanding sources of inefficiency in general-purpose chips, Hameed et al., ISCA 2010

Sources of Energy-Inefficiencies

General-Purpose CPU = Instruction Delivery + Data Feeding + Execution + Control, where instruction delivery, data feeding & control are pure overhead



Pure Overhead

IF: Instruction fetch

Ctrl: Other control logics

Pipe: Pipeline reg, bus, clock

D$: Data cache

RF: Register file

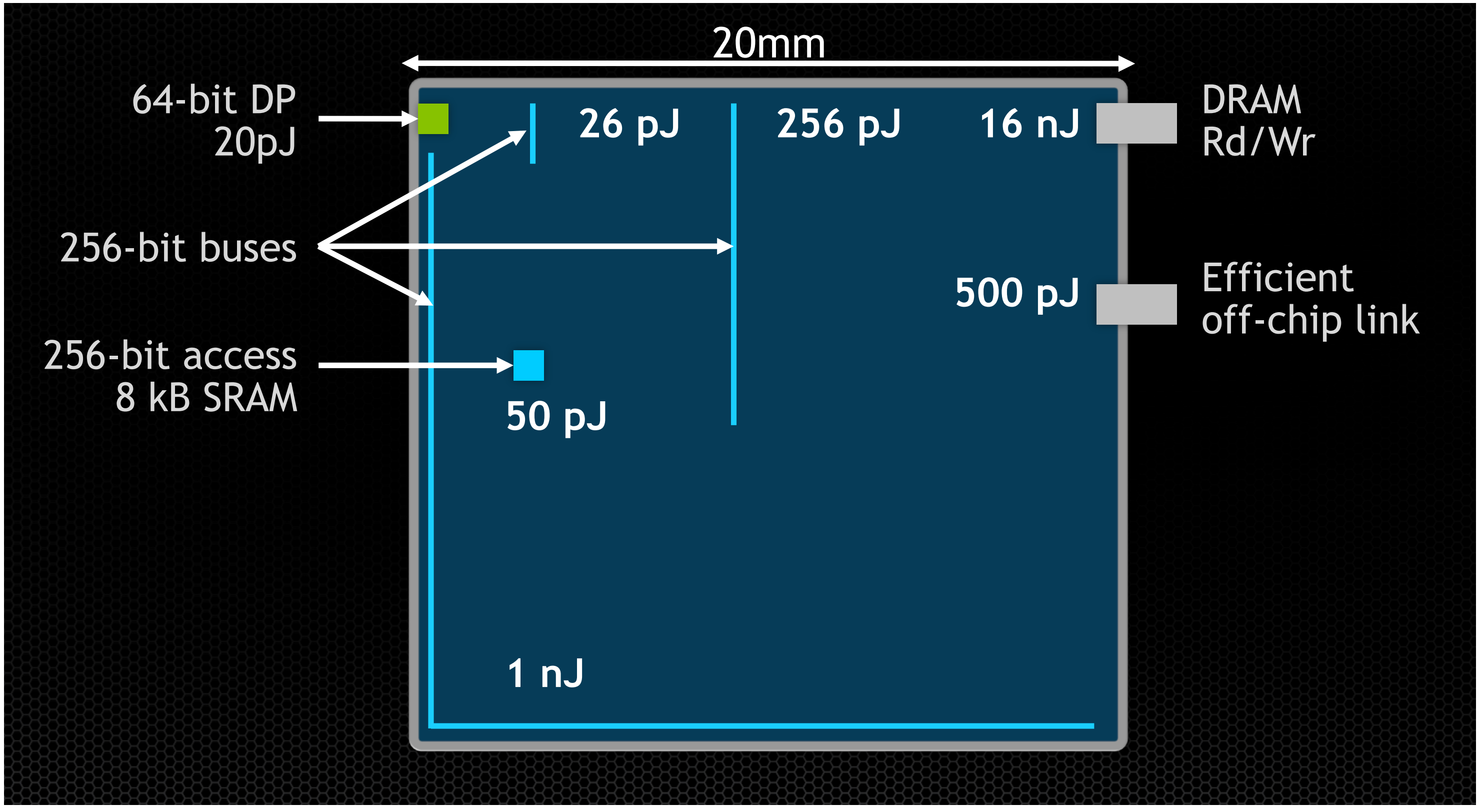
ALU: Functional units

Doing Actual Work

Understanding sources of inefficiency in general-purpose chips, Hameed et al., ISCA 2010

Computation vs. Data Movement

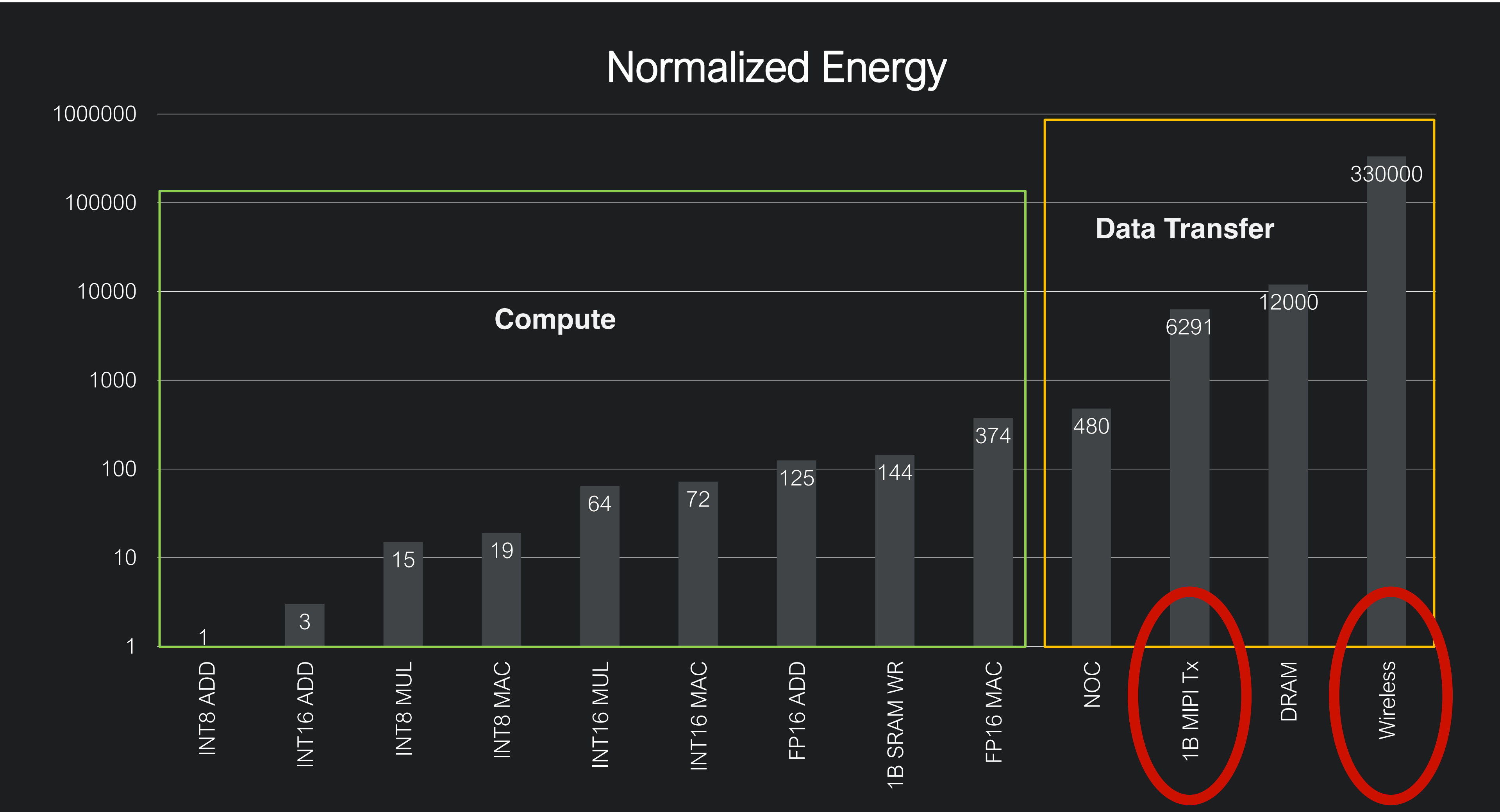
Data movement energy >> computation energy



Challenges for future computing systems, Bill Dally, 2015

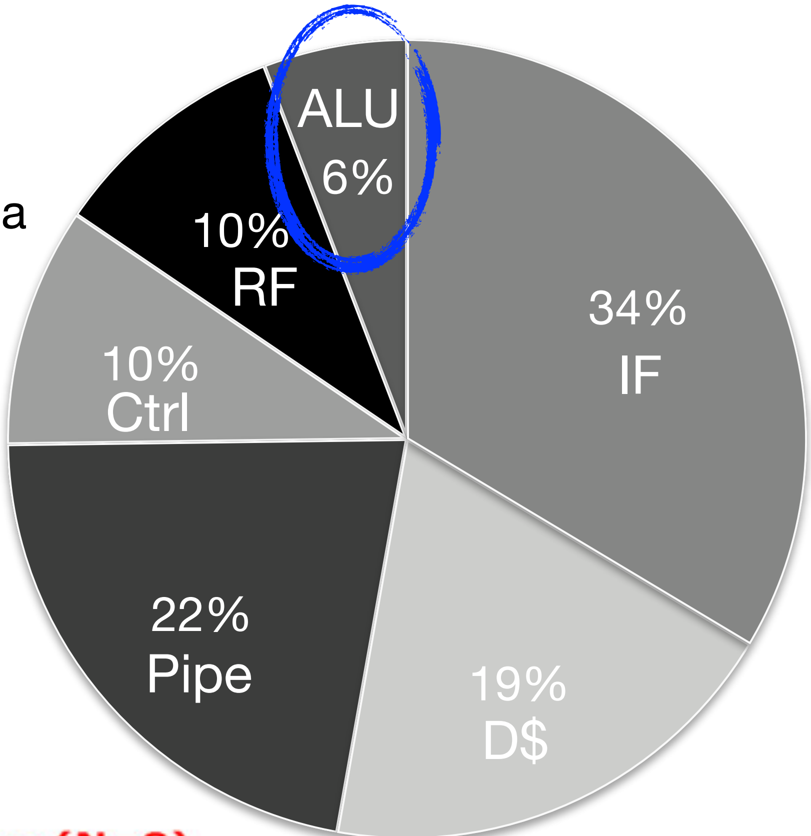
Computation vs. Data Movement

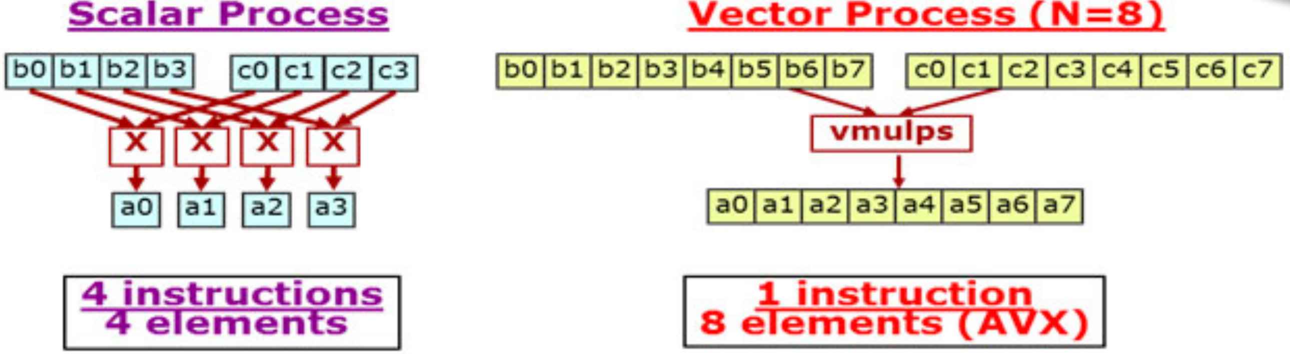
Data movement energy >> computation energy



SIMD

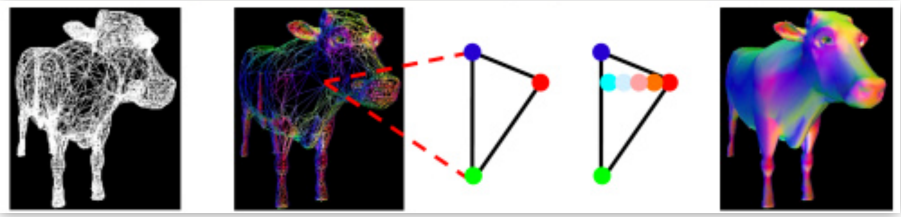
- Single Instruction (operating on) Multiple Data
- Amortizing the cost of instruction delivery/control across many execution units (even cores).
- Almost all modern ISAs provide such instructions:
    - x86: MMX/SSE/AVX
    - Arm: Neon



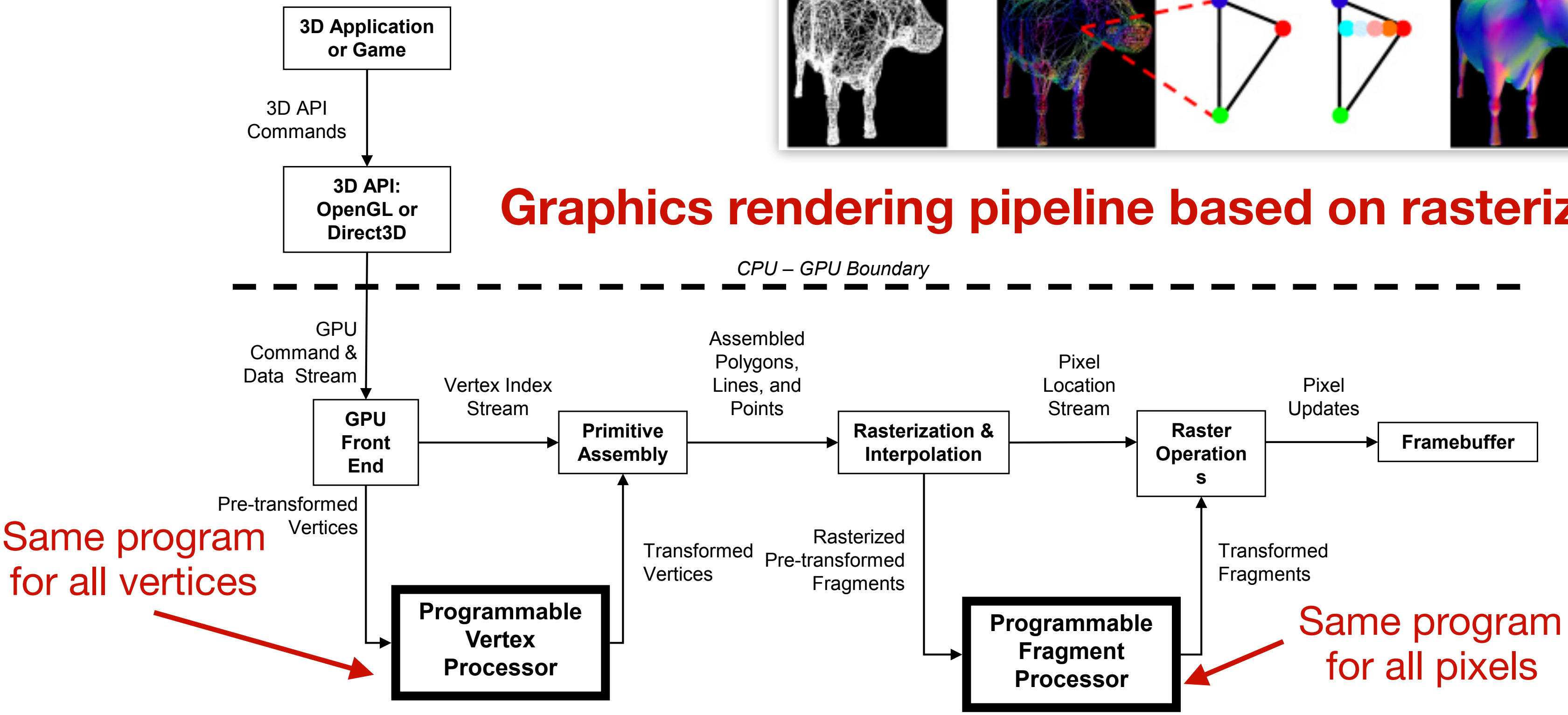


Graphics Processing Units/GPUs (SIMT)

- Designed for graphics rendering, which is massively parallel.



Graphics rendering pipeline based on rasterization



3D Application or Game

3D API Commands

3D API: OpenGL or Direct3D

CPU – GPU Boundary

GPU Command & Data Stream

GPU Front End

Vertex Index Stream

Primitive Assembly

Assembled Polygons, Lines, and Points

Rasterization & Interpolation

Pixel Location Stream

Raster Operations

Pixel Updates

Framebuffer

Pre-transformed Vertices

Programmable Vertex Processor

Transformed Vertices

Rasterized Pre-transformed Fragments

Programmable Fragment Processor

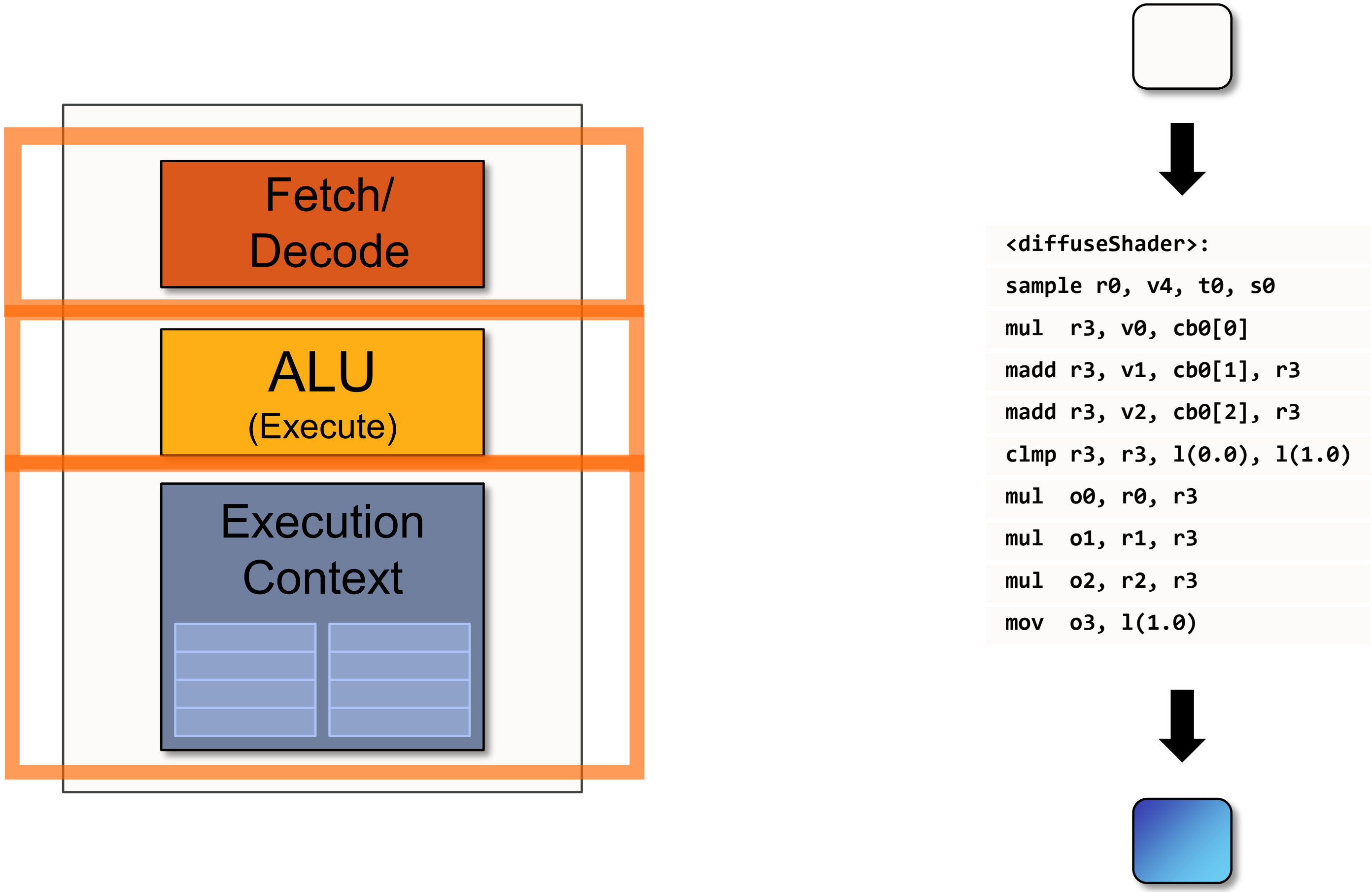
Transformed Fragments

Same program for all vertices

Same program for all pixels

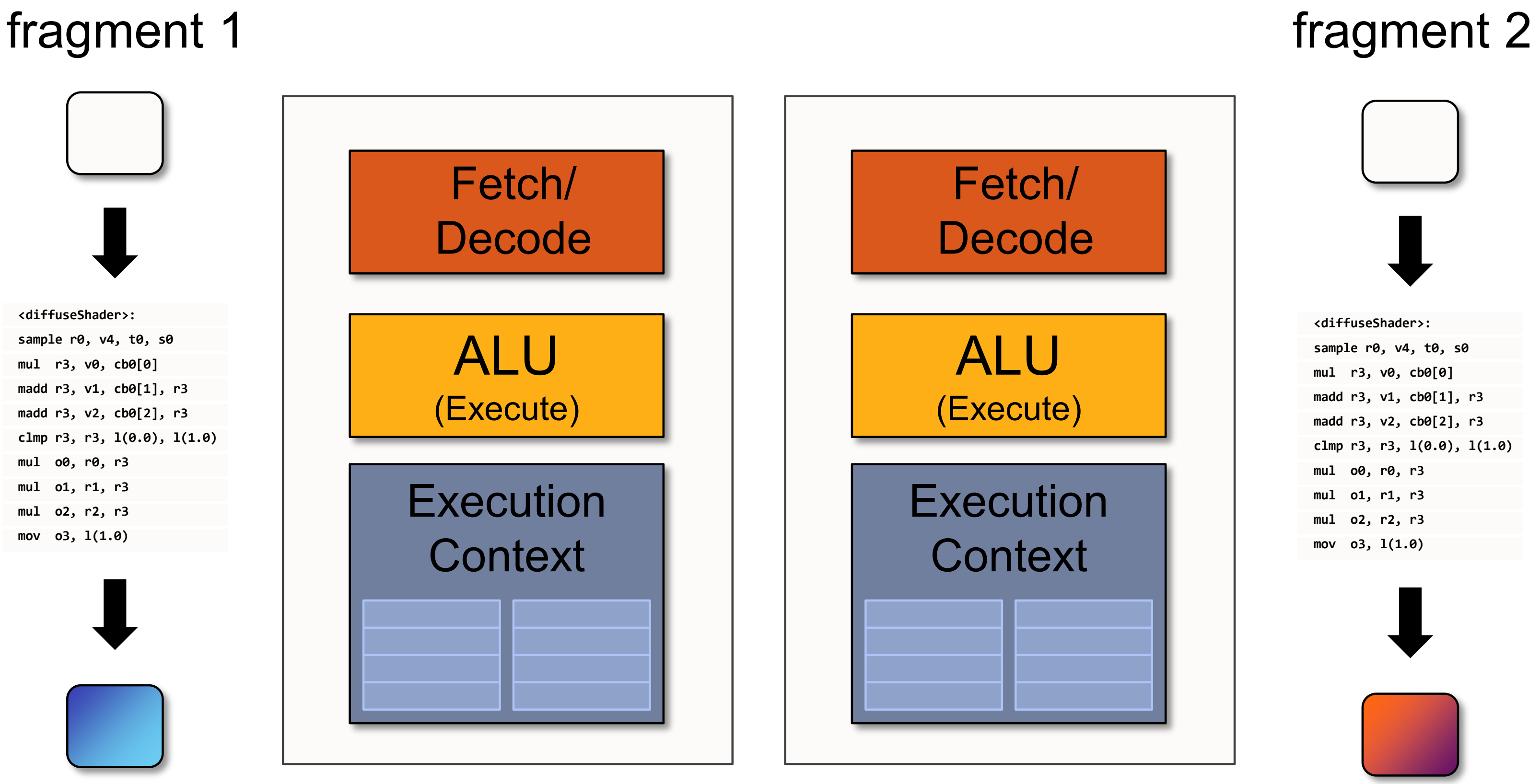
© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign

Execute shader



```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

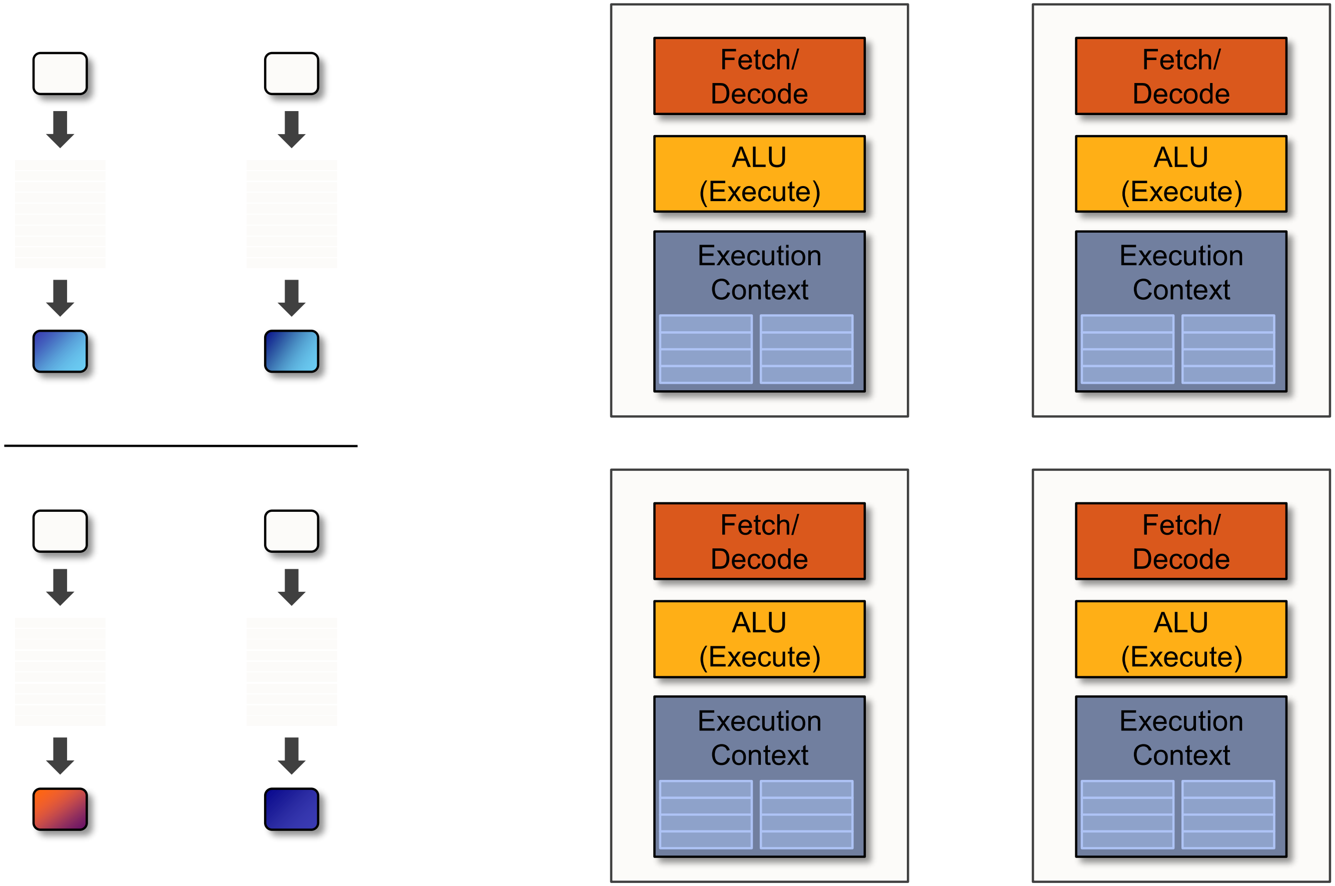
Two cores (two fragments in parallel)



```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Four cores (four fragments in parallel)



Fetch/ Decode

ALU (Execute)

Execution Context

Fetch/ Decode

ALU (Execute)

Execution Context

Fetch/ Decode

ALU (Execute)

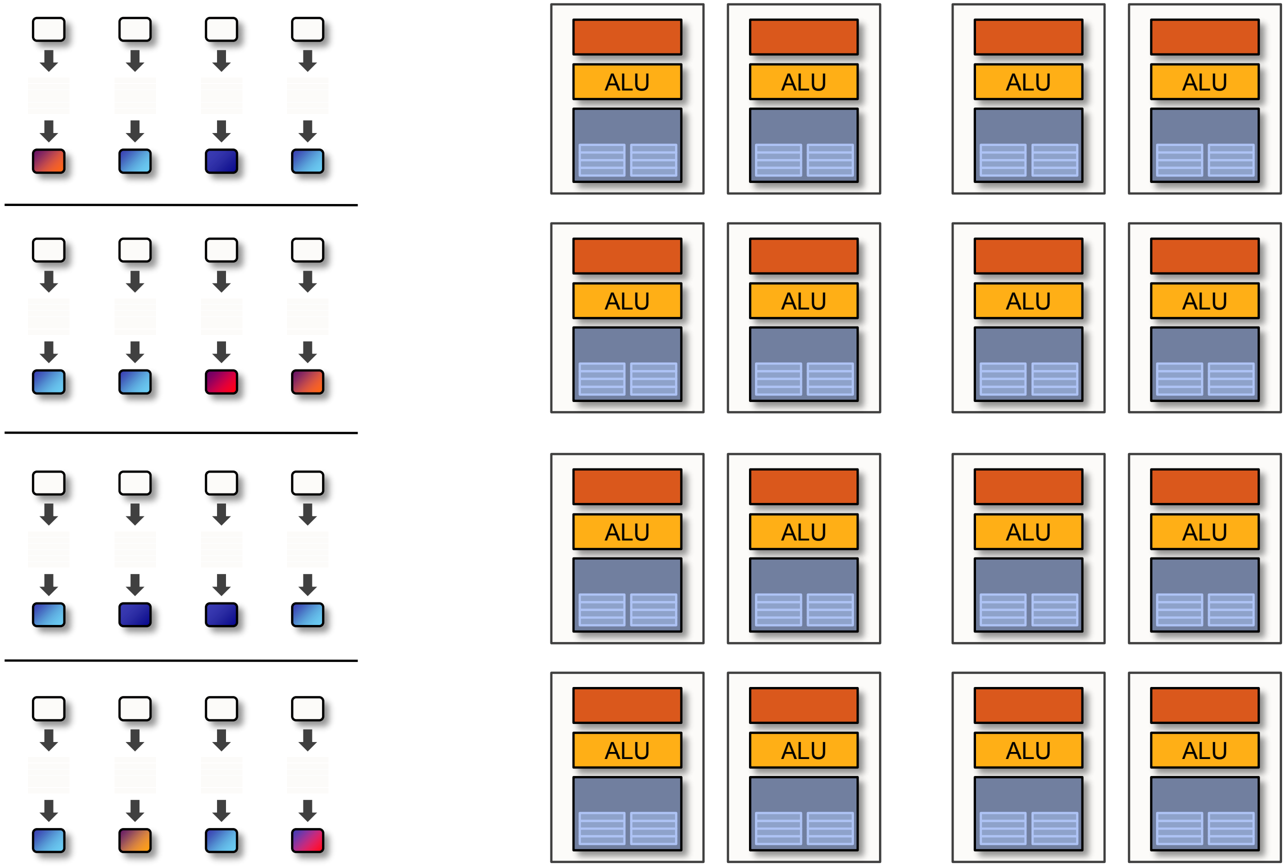
Execution Context

Fetch/ Decode

ALU (Execute)

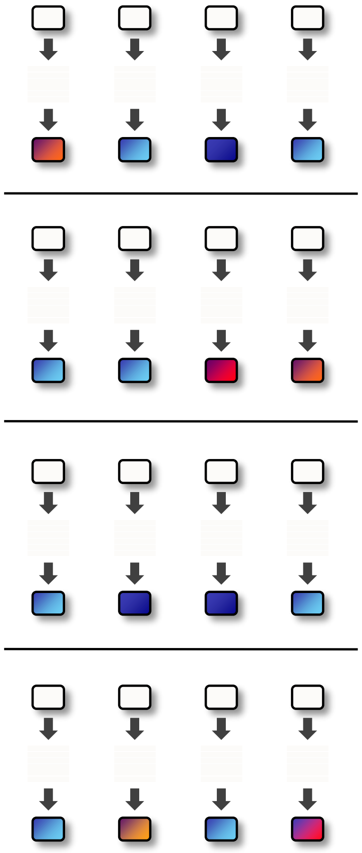
Execution Context

Sixteen cores (sixteen fragments in parallel)



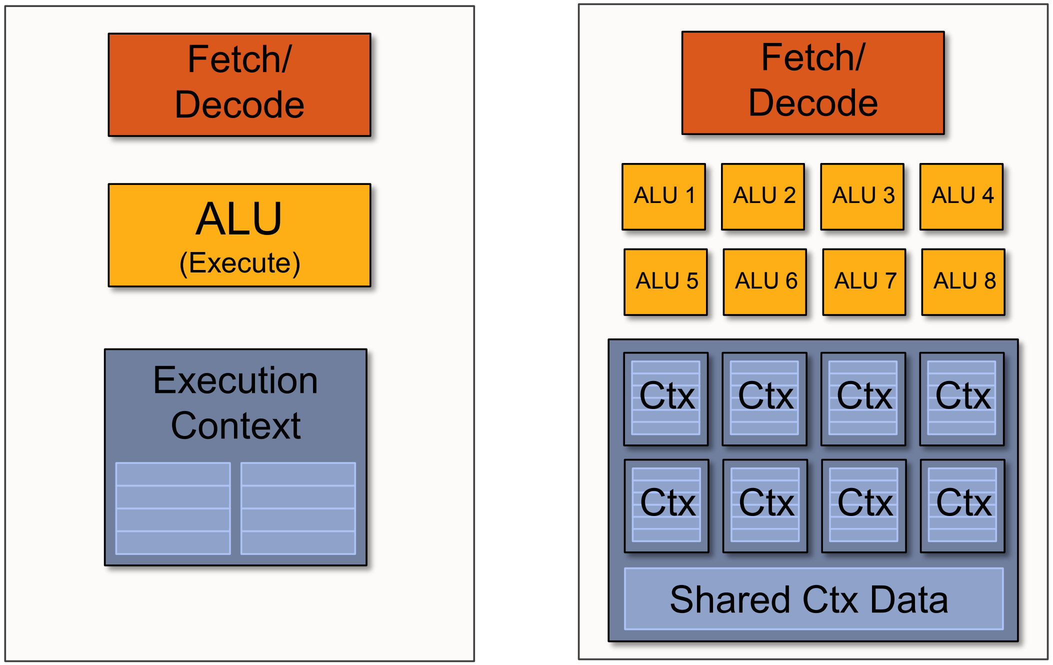
16 cores = 16 simultaneous instruction streams

Instruction stream coherence



But… many fragments should be able to share an instruction stream!

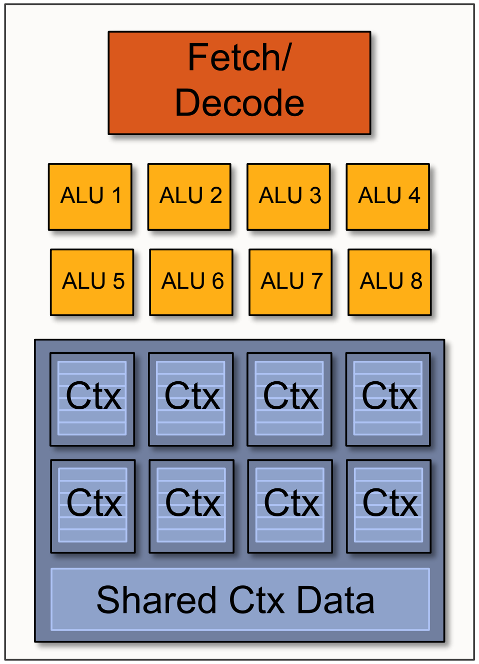
```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

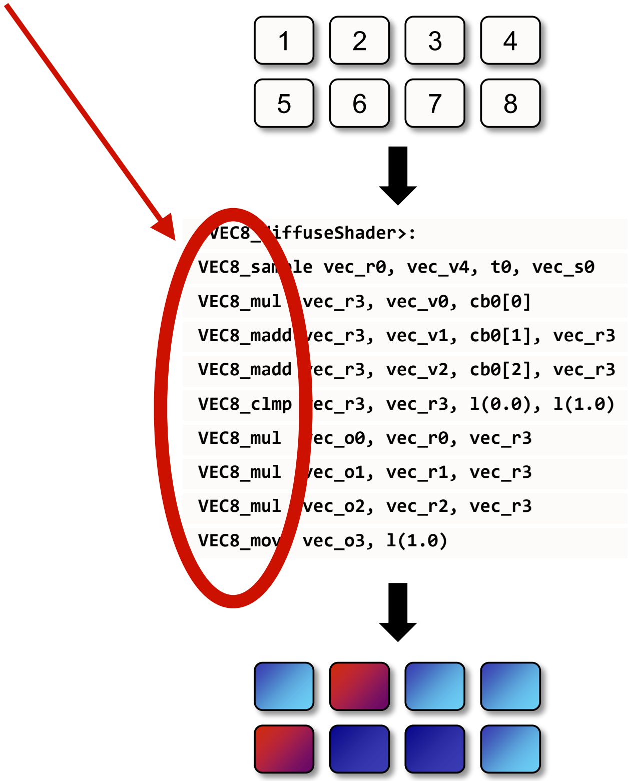


Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

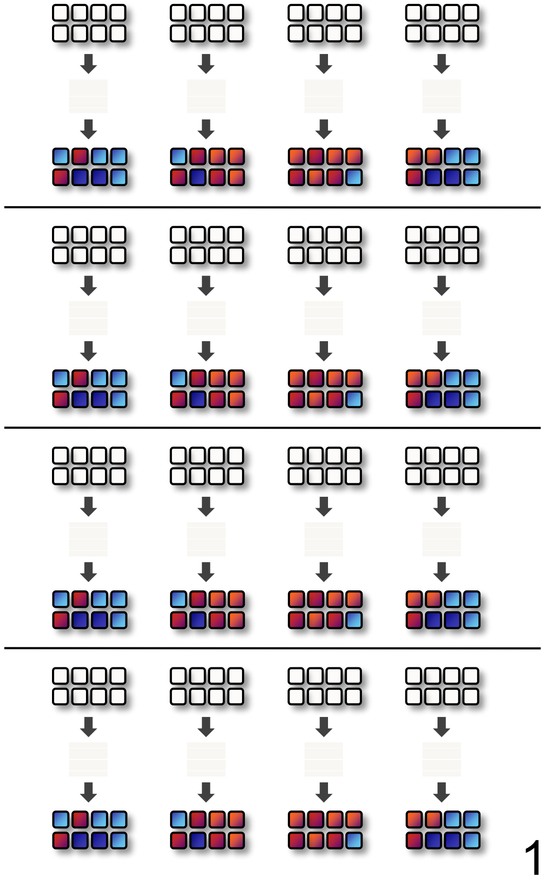
SIMD/vector instructions, each operates on a vector of 8 elements here.

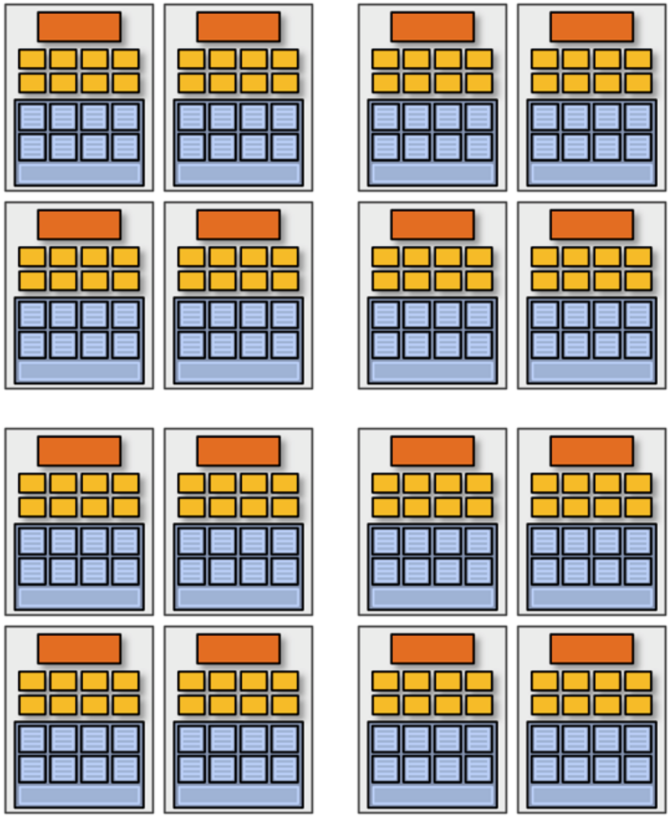




```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2, vec_r2, vec_r3
VEC8_mov  vec_o3, l(1.0)
```

16 cores, each with 8 ALUs. Each core here runs the same program (fragment shader)



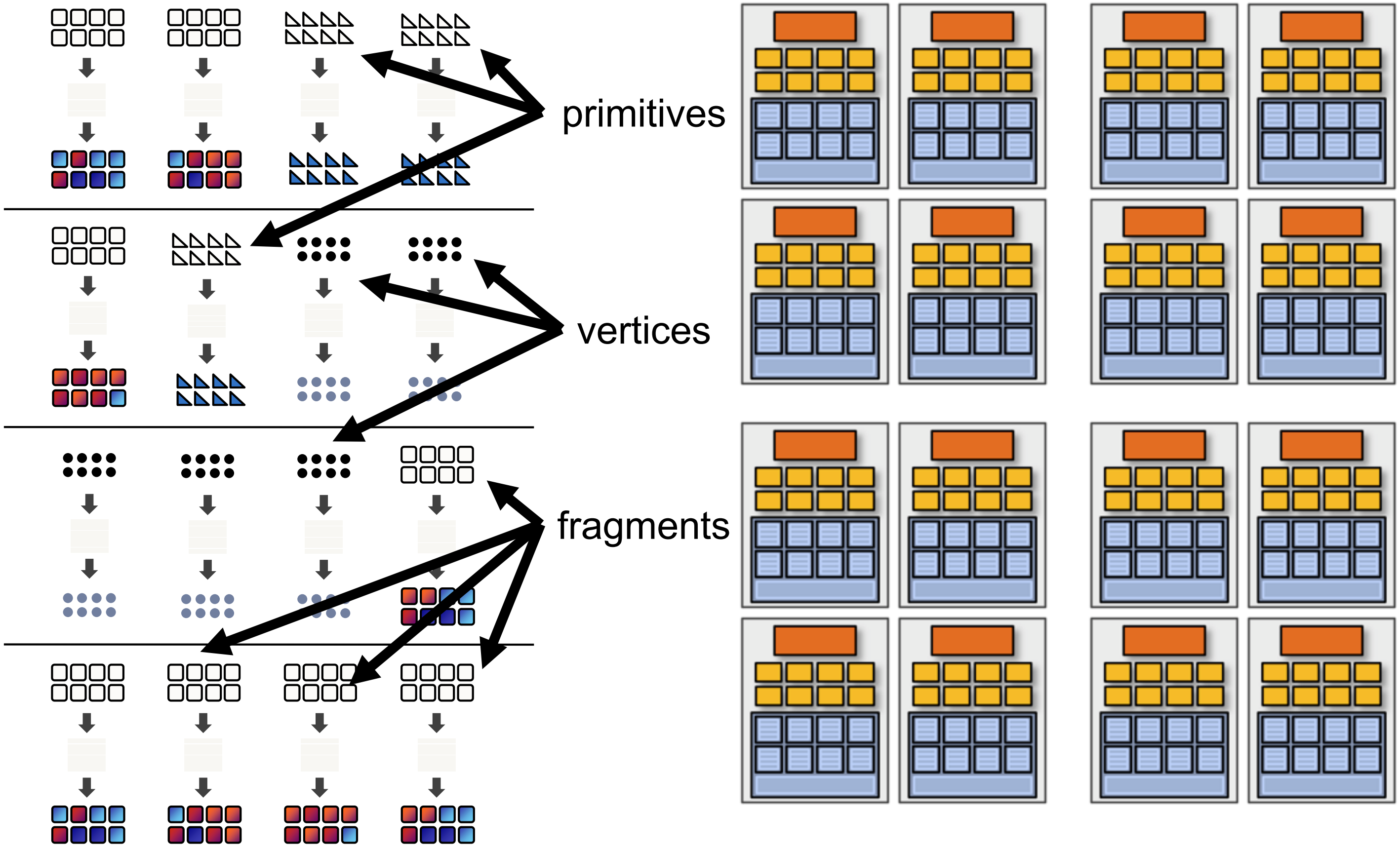


16 cores = 128 ALUs

= 16 simultaneous instruction streams

16 cores, each with 8 ALUs. Cores here run different programs

(some are processing vertices, some are processing fragments)



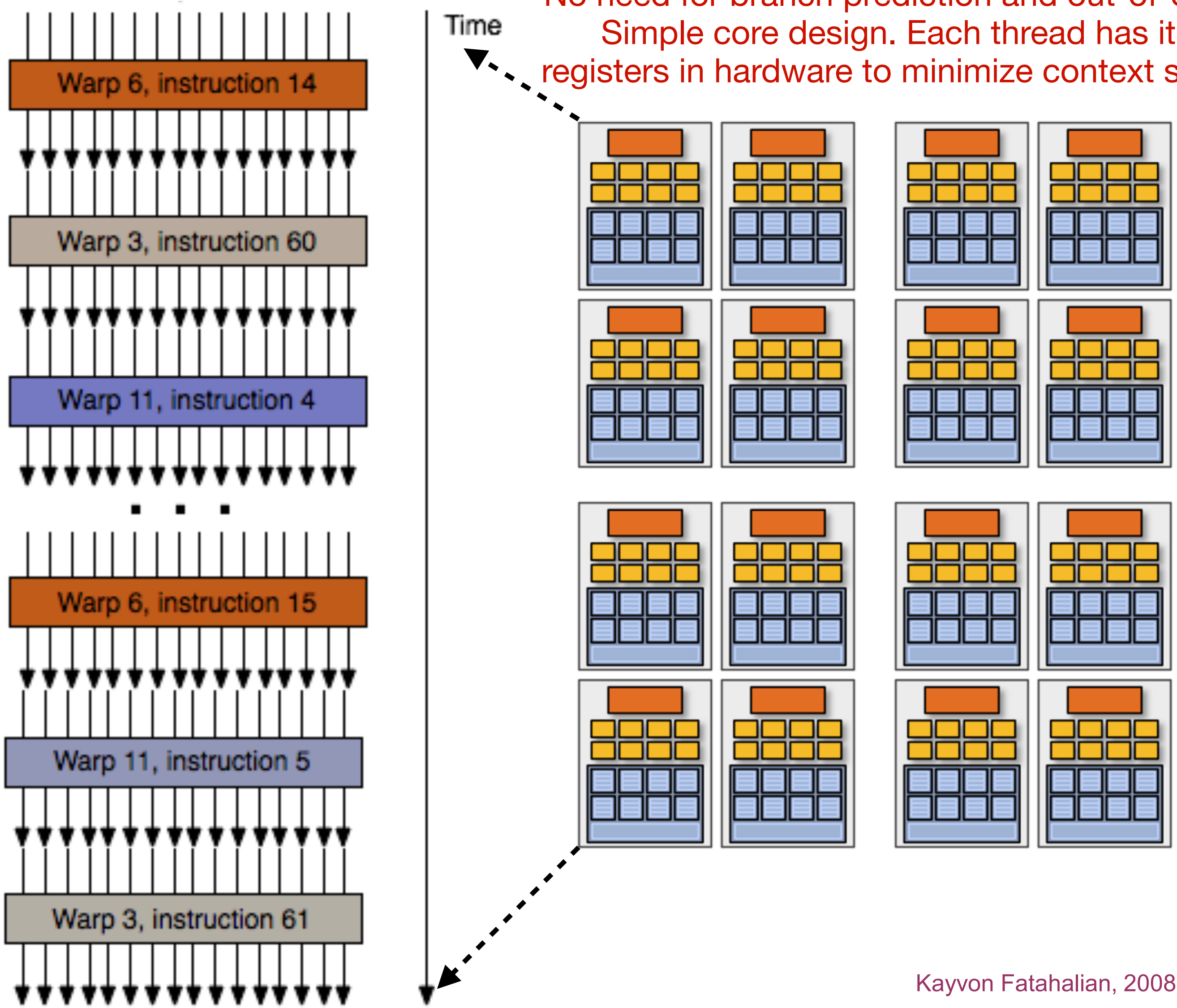
primitives

vertices

fragments

Each Core Does Fine-Grained Multi-threading

Warp: a group of threads (8 here)



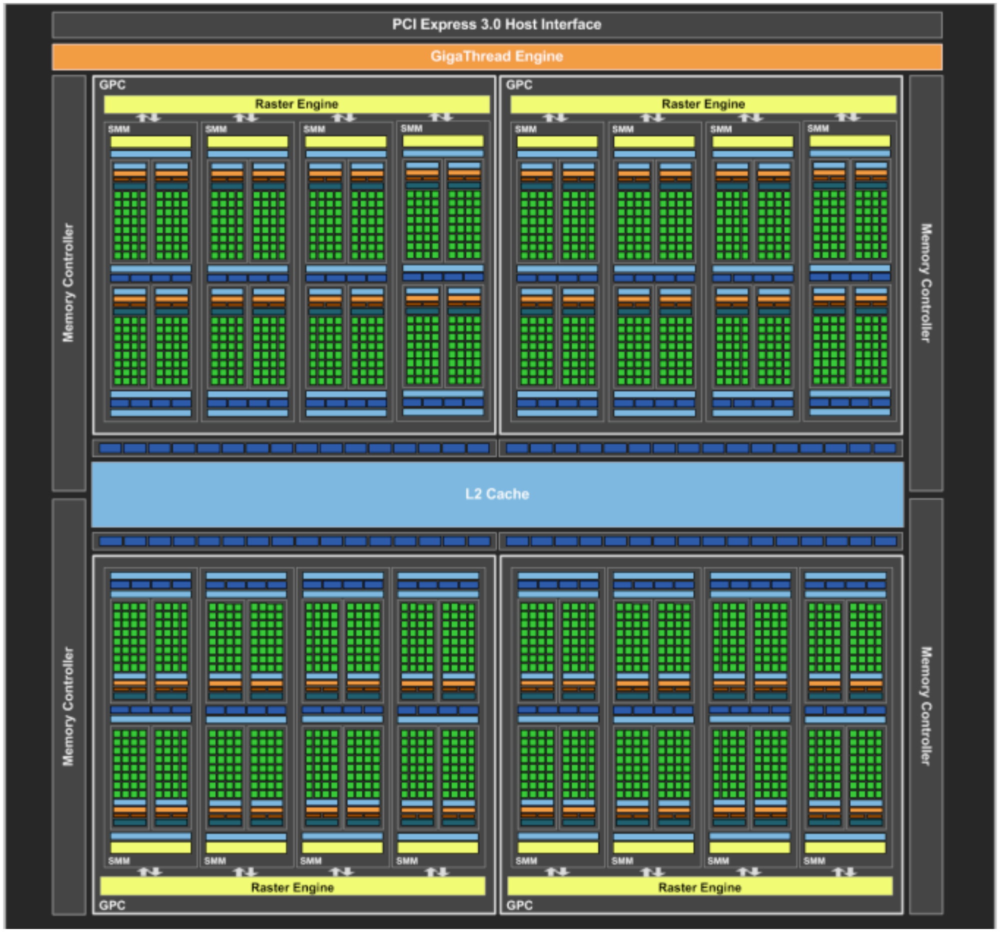
No need for branch prediction and out-of-order execution. Simple core design. Each thread has its own set of registers in hardware to minimize context switch overhead.

Kayvon Fatahalian, 2008

33

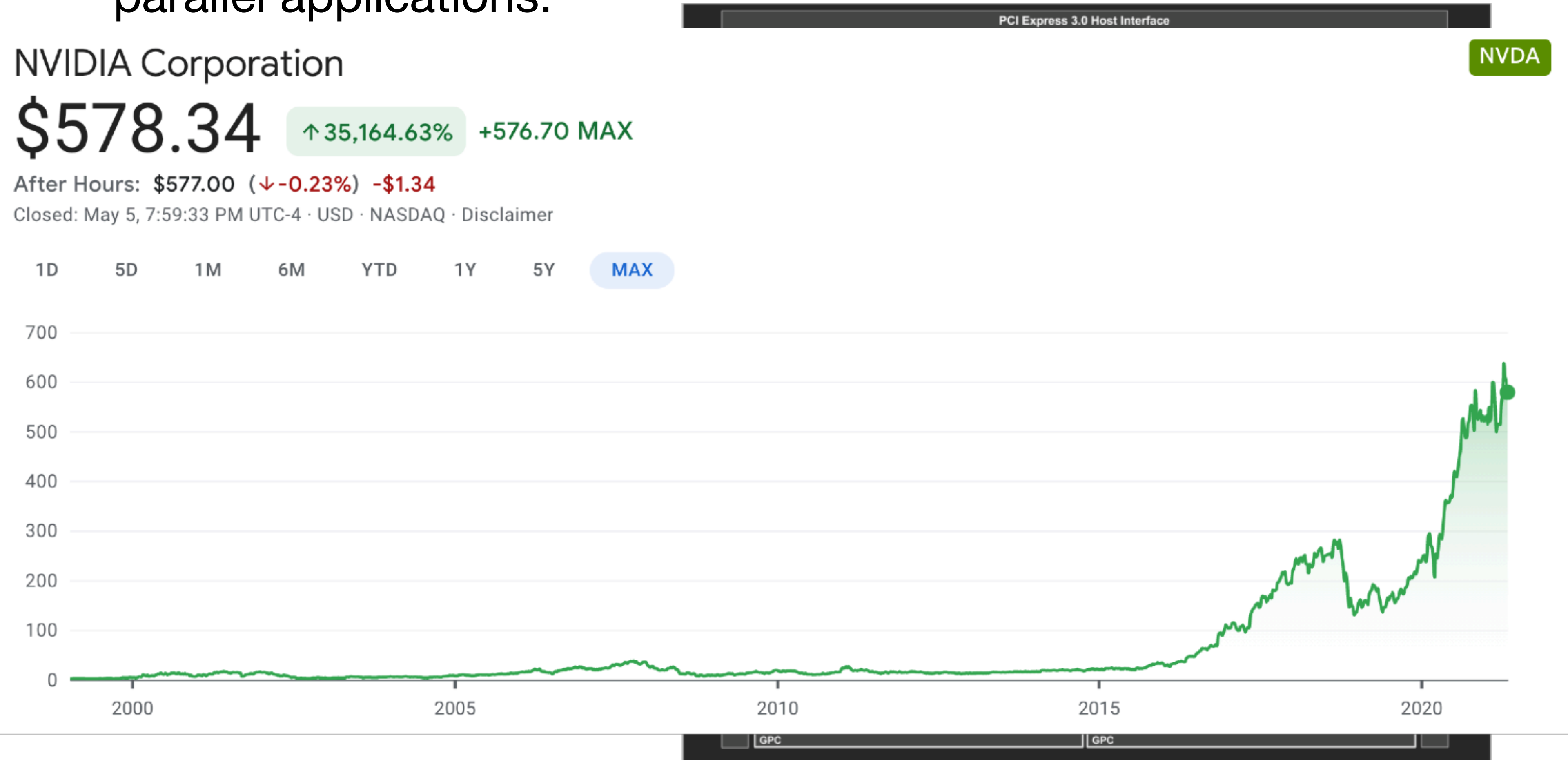
Nvidia Maxwell GPU (2014)

- Today: General Purpose GPU (GPGPU), used for any massive parallel applications:
  - Physics simulation
  - Deep learning
  - Computer vision



Nvidia Maxwell GPU (2014)

- Today: General Purpose GPU (GPGPU), used for any massive parallel applications:



NVIDIA Corporation

NVDA

$578.34 ↑35,164.63% +576.70 MAX

After Hours: $577.00 (↓-0.23%) -$1.34

Closed: May 5, 7:59:33 PM UTC-4 · USD · NASDAQ · Disclaimer