

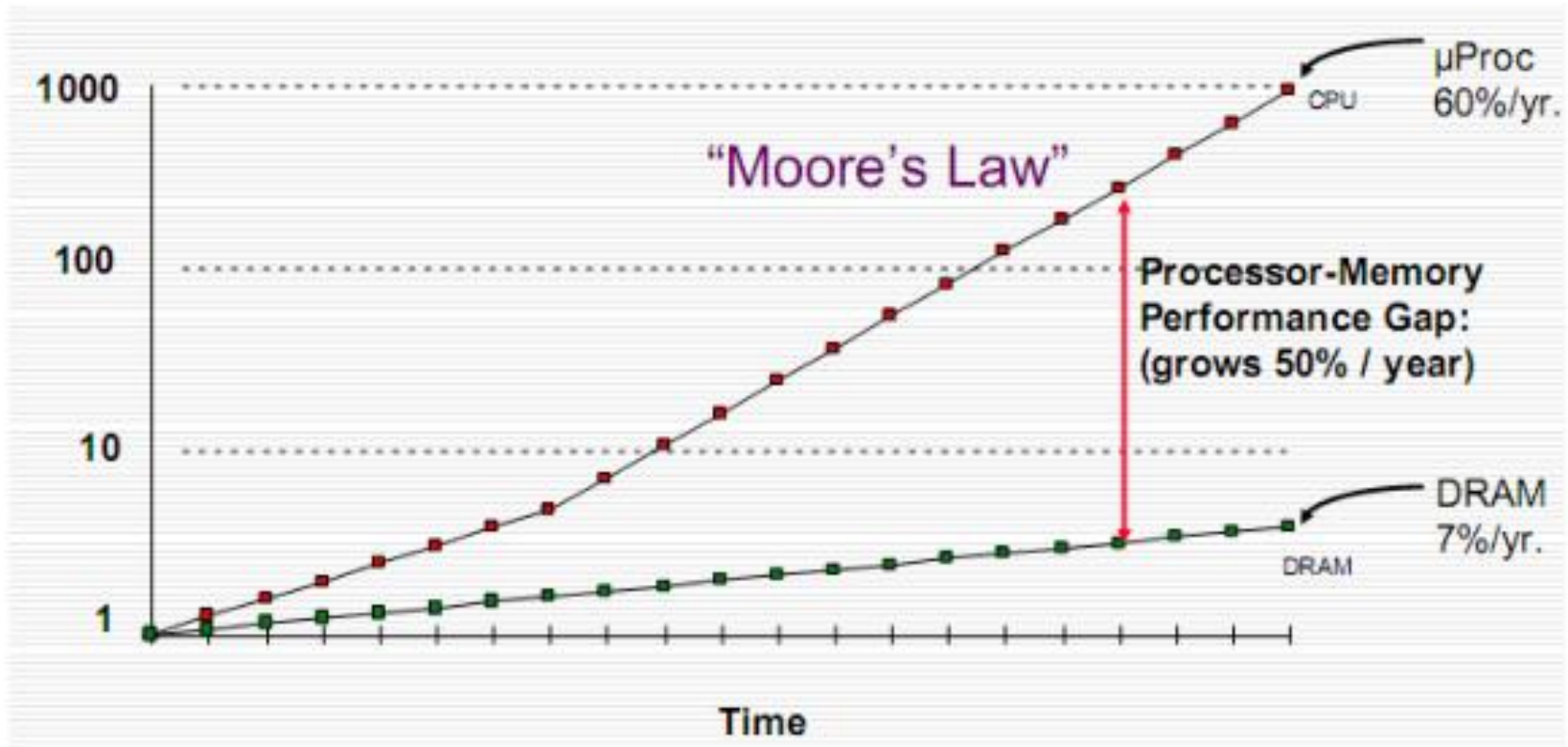
# The Memory Hierarchy

Christina Delimitrou

Reminders: Quiz (11/10)  
and Quiz Review (11/8)  
next week

# Memory Wall

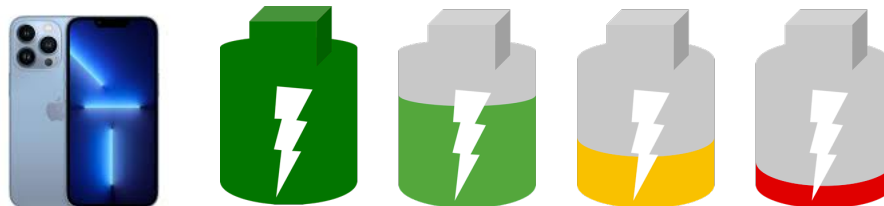
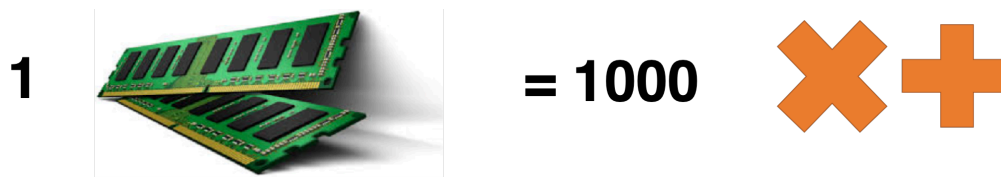
- Processor speed improving faster than memory speed



# Memory is Expensive

Operation	Energy [pJ]
32 bit int ADD	0.1
32 bit float ADD	0.9
32 bit Register File	1
32 bit int MULT	3.1
32 bit float MULT	3.7
32 bit SRAM Cache	5
<b>32 bit DRAM Memory</b>	<b>640</b>

[Han et al, NIPS'15]



# Memory System is Important

---

- Example: AI applications are going “tiny” with many IoT applications.
- However, it’s challenging to fit powerful AI applications with a tight memory budget.
- It’s important to learn about the memory system.



Memory	32GB	4GB	256kB
Storage	~TB	256GB	1MB

# Memory Technologies

- Technologies have vastly different tradeoffs between capacity, latency, bandwidth, energy, and cost

	Capacity	Latency	Cost/GB
Register	100s of bits	20 ps	\$\$\$\$
SRAM	~10KB-10 MB	1-10 ns	~\$1000
DRAM	~10 GB	80 ns	~\$10
Flash*	~100 GB	100 us	~\$1
Hard disk*	~1 TB	10 ms	~\$0.1

**Processor  
Datapath**

**Memory  
Hierarchy**

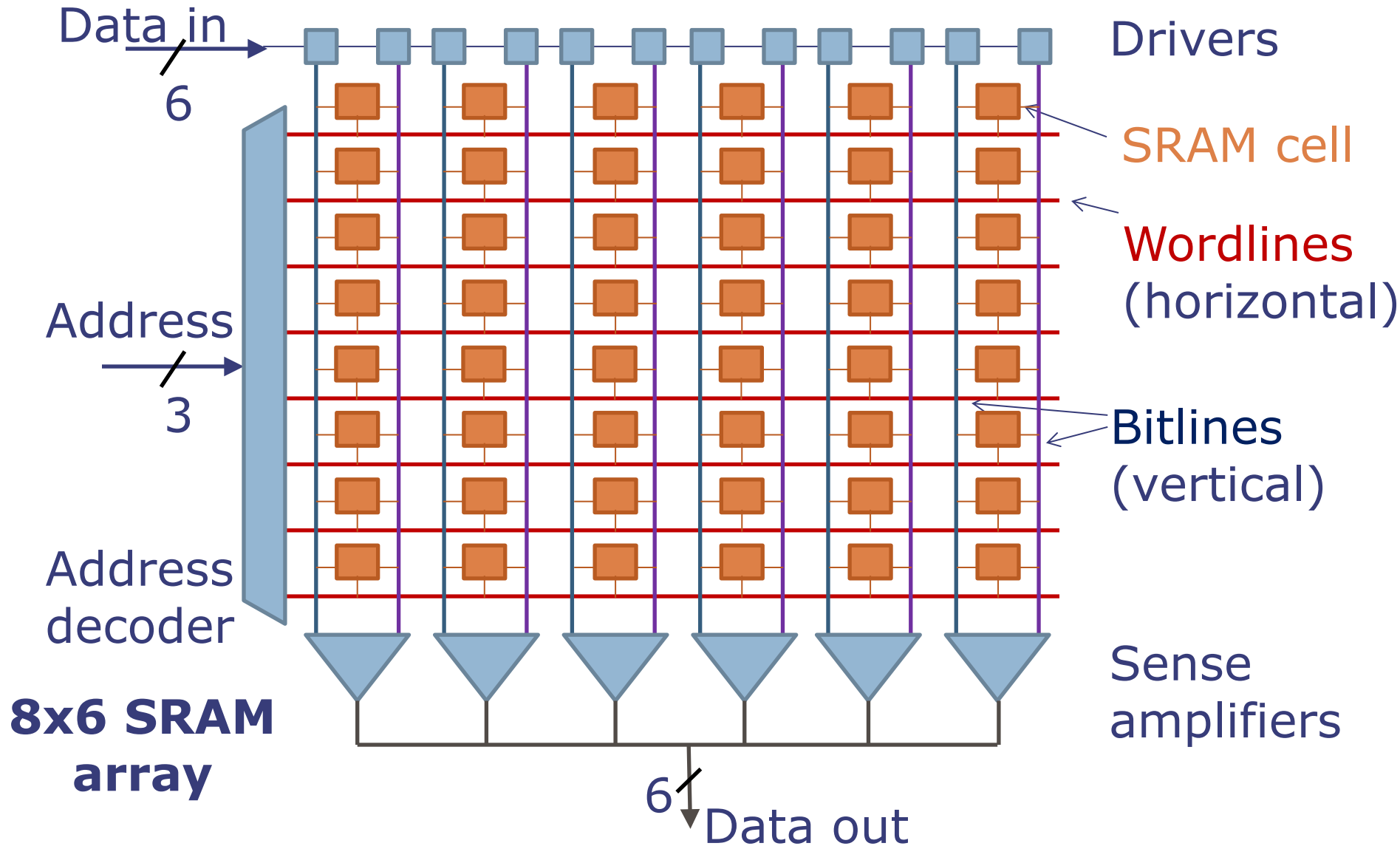
**I/O  
subsystem**

\* non-volatile (retains contents when powered off)

# Memory Technologies: SRAM, DRAM, Flash, Hard Disk

*NOTE: Demystification,  
will not be on the quiz*

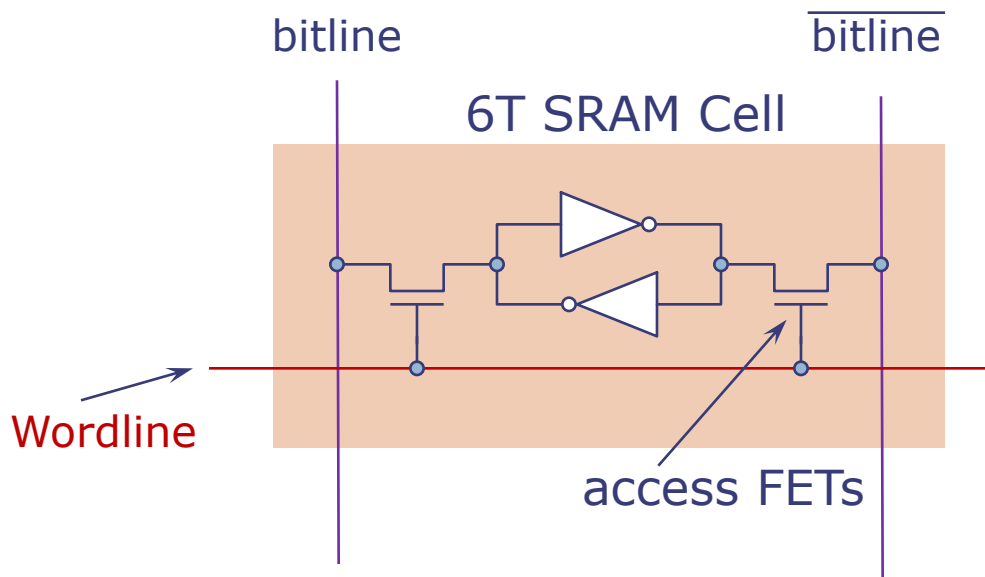
# Static RAM (SRAM)



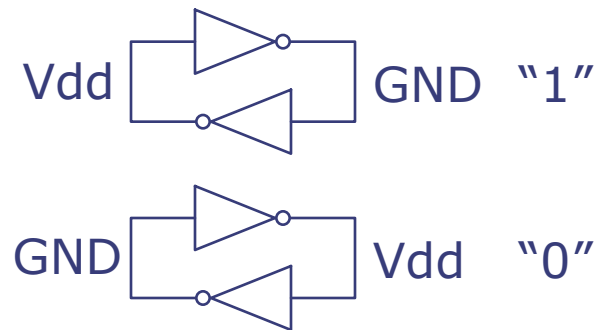
# SRAM Cell

## 6-transistor (6T) cell:

- Two CMOS inverters (4 FETs) forming a **bistable element**
- Two **access transistors**

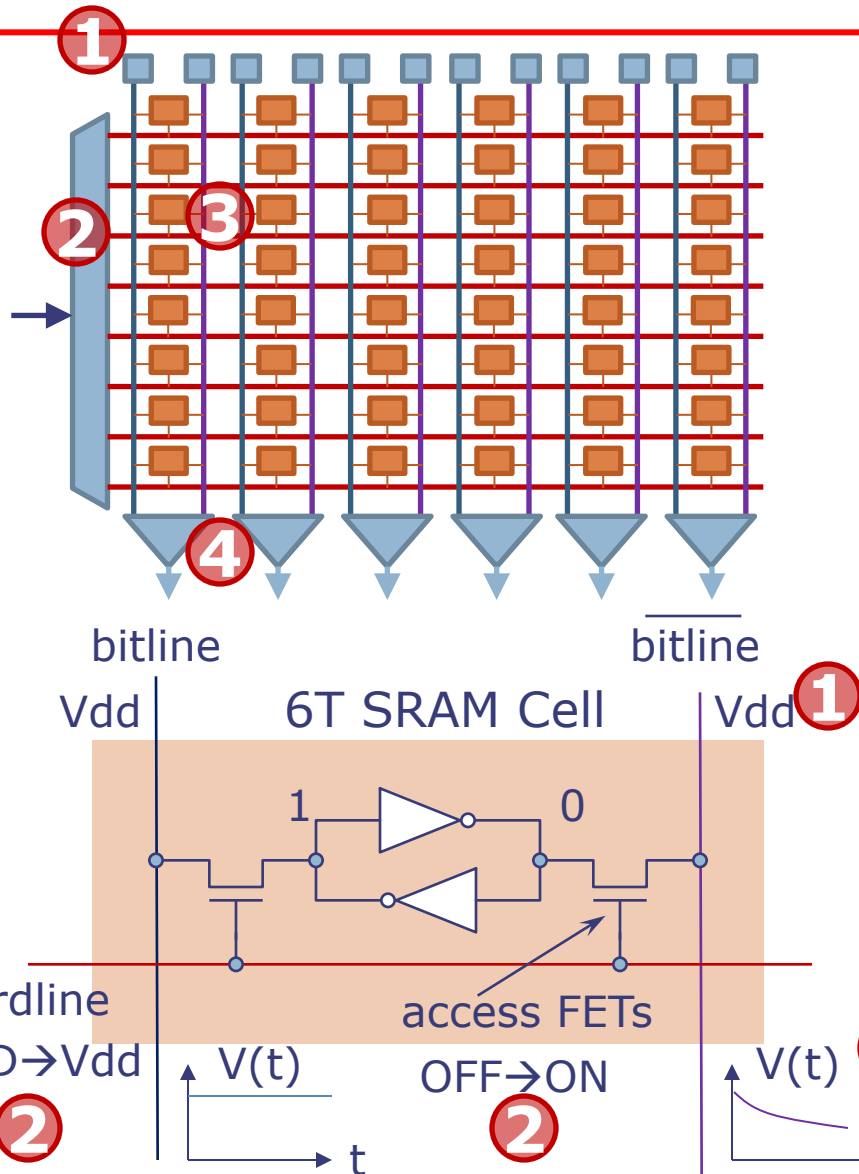


Bistable element  
(two stable states)  
stores a single bit



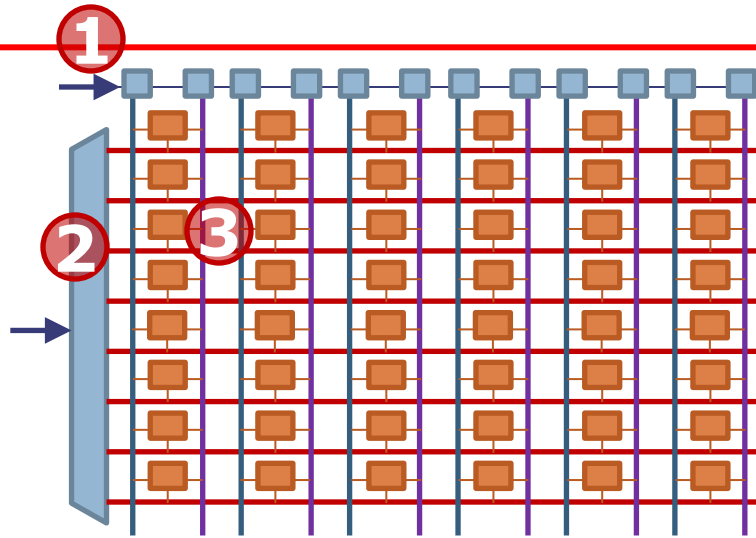


# SRAM Read

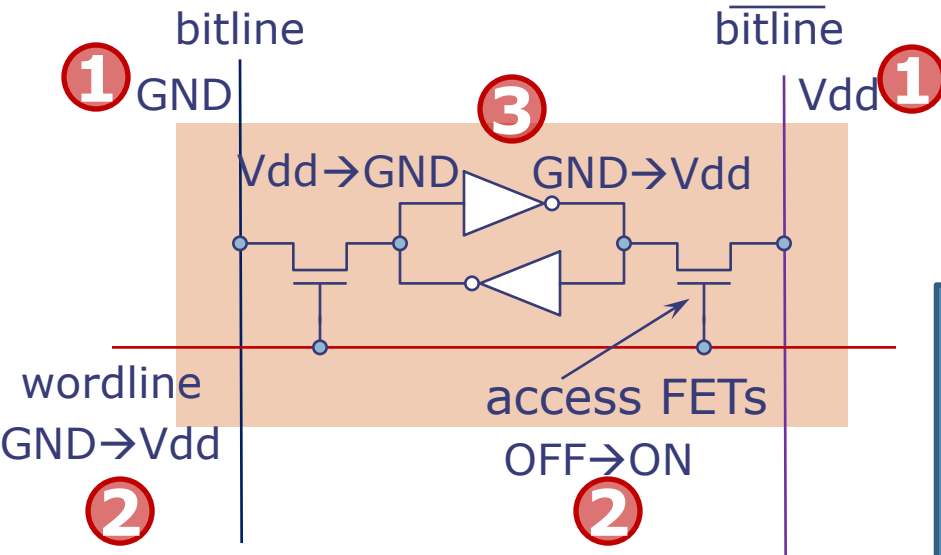


1. Drivers precharge all bitlines to  $V_{dd}$  (1), and leave them floating
2. Address decoder activates one wordline
3. Each cell in the activated word slowly pulls down one of the bitlines to  $GND$  (0)
4. Sense amplifiers sense change in bitline voltages, produce output data

# SRAM Write



1. Drivers set and hold bitlines to desired values (Vdd and GND for 1, GND and Vdd for 0)
2. Address decoder activates one wordline
3. Each cell in word is overpowered by the drivers, stores value



Cell transistors are carefully sized so that bitline GND overpowers cell Vdd, but bitline Vdd does not overpower cell GND

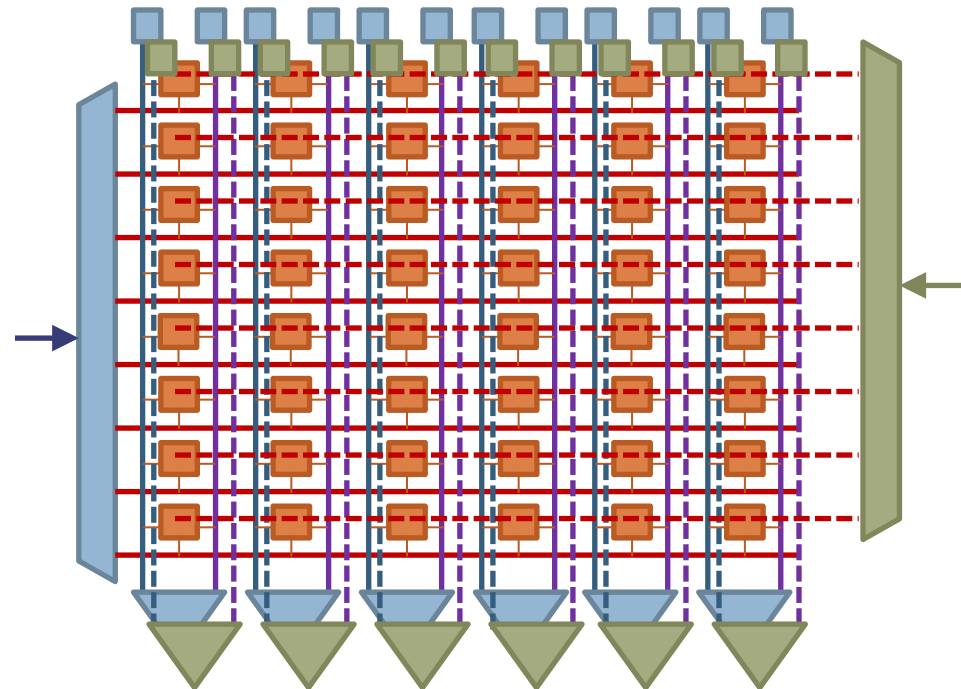
# Multiported SRAMs

- SRAM so far can do either one read or one write/cycle
- We can do multiple reads and writes with multiple ports by adding one set of wordlines and bitlines per port

- *Cost/bit for  $N$  ports?*

- *Wordlines?*  $N$
- *Bitlines?*  $2*N$
- *Access FETs?*  $2*N$

- Wires dominate area  $\rightarrow O(N^2)$  area!

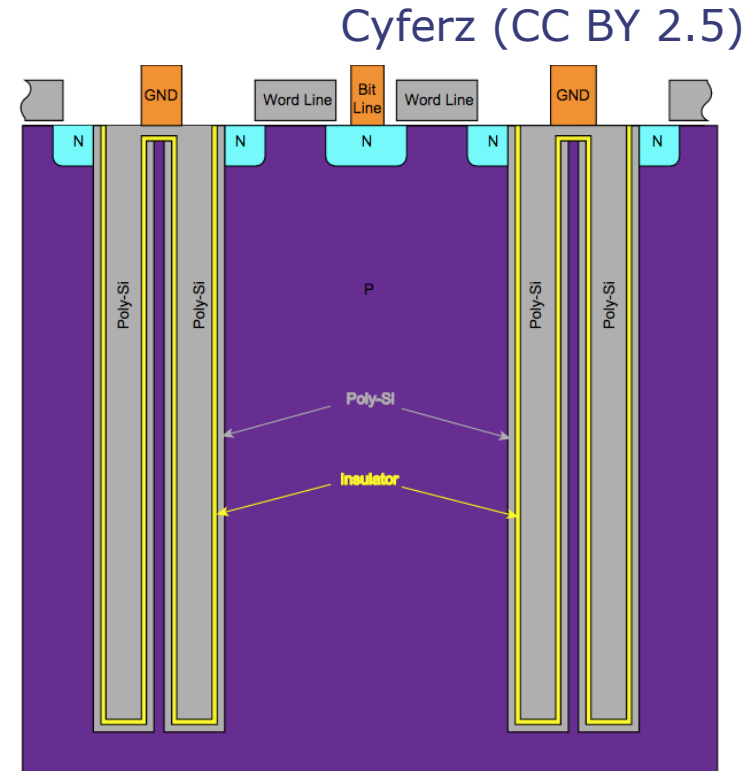
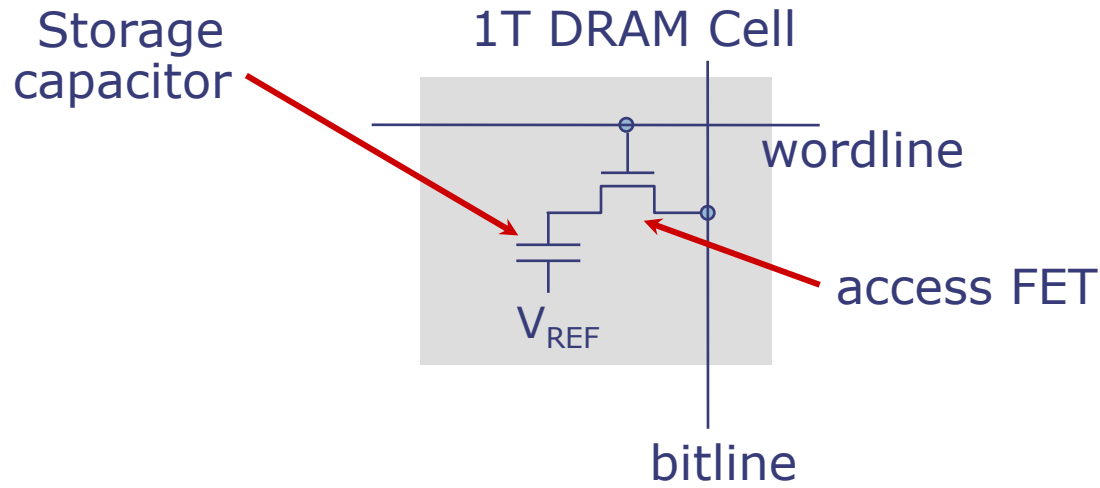


# Summary: SRAMs

---

- Array of  $k \times b$  cells ( $k$  words,  $b$  cells per word)
- Cell is a bistable element + access transistors
  - Analog circuit with carefully sized transistors
- Read: Precharge bitlines, activate wordline, sense
- Write: Drive bitlines, activate wordline, overpower cells
  
- 6 FETs/cell... can we do better?

# 1T Dynamic RAM (DRAM) Cell

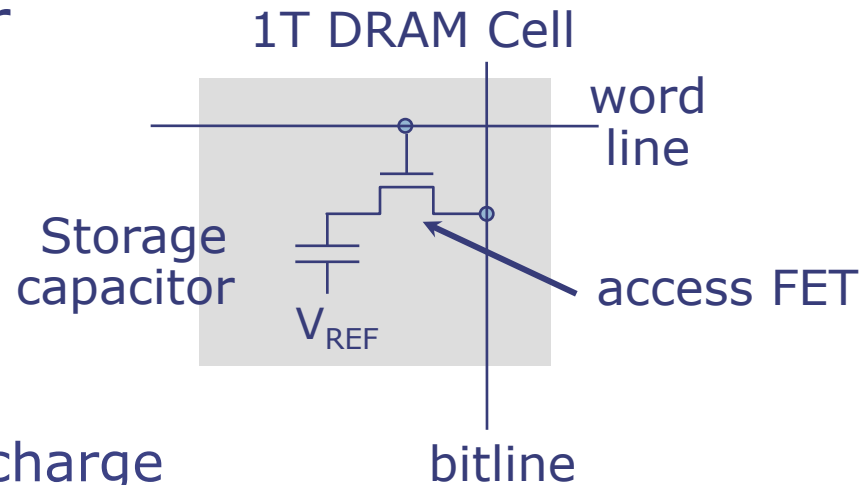


Trench capacitors  
take little area

- ✓ ~20x smaller area than SRAM cell → Denser and cheaper!
- ✗ Problem: Capacitor leaks charge, must be refreshed periodically (~milliseconds)

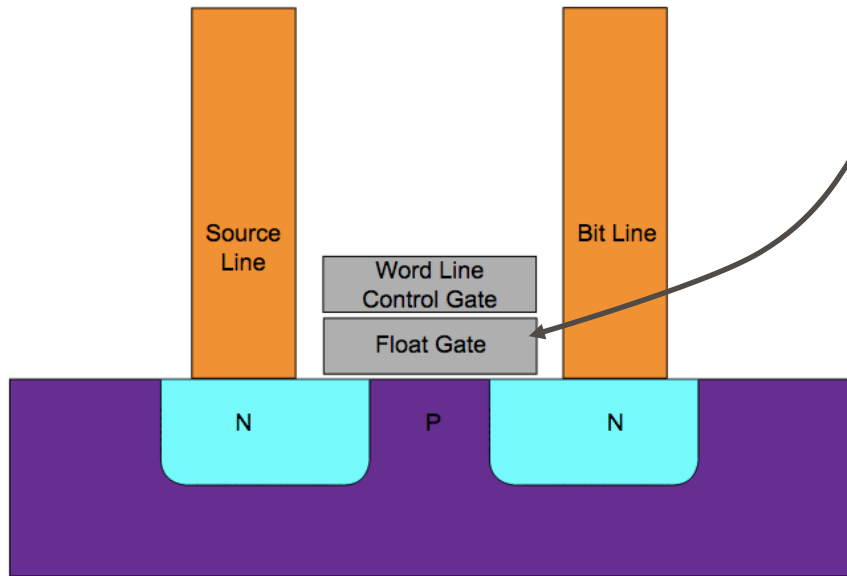
# DRAM Writes and Reads

- Writes: Drive bitline to  $V_{dd}$  or GND, activate wordline, charge or discharge capacitor
- Reads:
  1. Precharge bitline to  $V_{dd}/2$
  2. Activate wordline
  3. Capacitor and bitline share charge
    - If capacitor was discharged, bitline voltage decreases slightly
    - If capacitor was charged, bitline voltage increases slightly
  4. Sense bitline to determine if 0 or 1
- Issue: **Reads are destructive!** (charge is gone!)
  - Data must be rewritten to cells at end of read



# Non-Volatile Storage: Flash

---



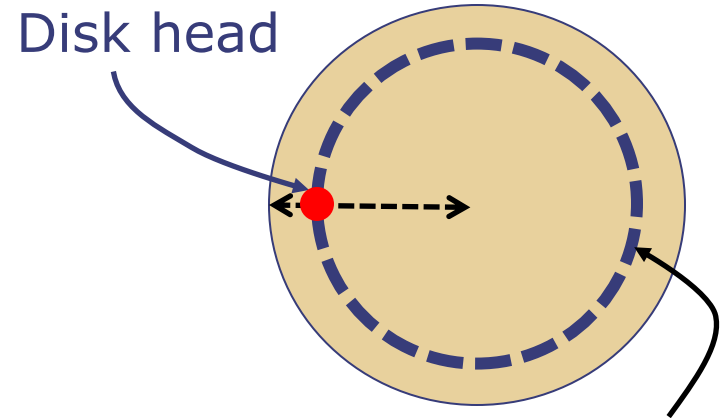
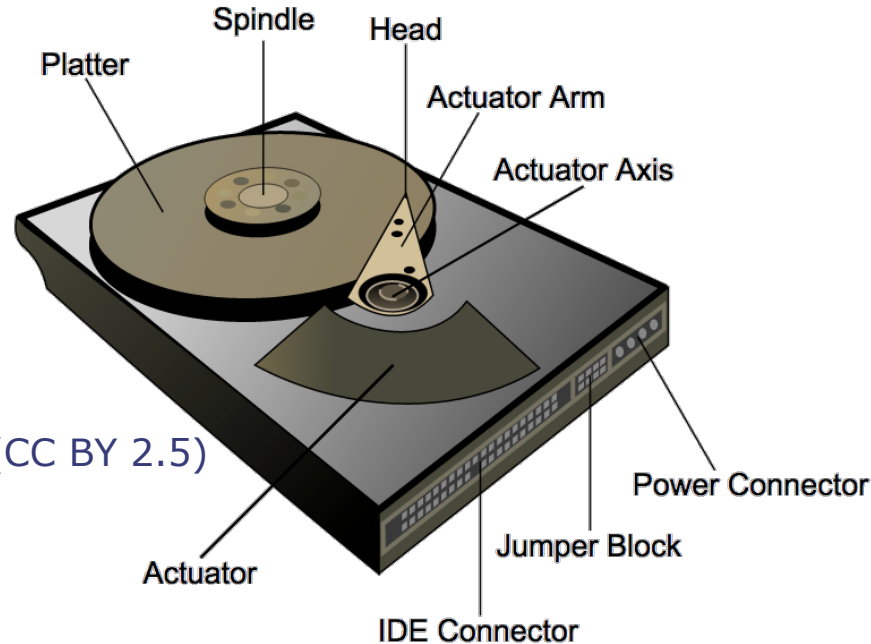
Electrons here diminish strength of field from control gate  $\Rightarrow$  no inversion  $\Rightarrow$  NFET stays off even when word line is high.

Cyferz (CC BY 2.5)

Flash Memory: Use “floating gate” transistors to store charge (floating gate is a well insulated conductor)

- **Very dense:** Multiple bits/transistor, read and written in blocks
- **Slow** (especially on writes), 10-100  $\mu$ s
- **Limited number of writes:** charging/discharging the floating gate (writes) requires large voltages that damage transistor

# Non-Volatile Storage: Hard Disk



Circular track divided into sectors

Hard Disk: Rotating magnetic platters + read/write head

- **Extremely slow** ( $\sim 10\text{ms}$ ): Mechanically move head to position, wait for data to pass underneath head
- $\sim 100\text{MB/s}$  for sequential read/writes
- $\sim 100\text{KB/s}$  for random read/writes
- **Cheap**



# The Memory Hierarchy

# Summary: Memory Technologies

---

	Capacity	Latency	Cost/GB
Register	100s of bits	20 ps	\$\$\$\$
SRAM	~10 KB-10 MB	1-10 ns	~\$1000
DRAM	~10 GB	80 ns	~\$10
Flash	~100 GB	100 us	~\$1
Hard disk	~1 TB	10 ms	~\$0.1

- Different technologies have vastly different tradeoffs
- Size is a **fundamental limit**, even setting cost aside:
  - Small + low latency, high bandwidth, low energy, **or**
  - Large + high-latency, low bandwidth, high energy
- Can we get best of both worlds? (large, fast, cheap)

# The Memory Hierarchy

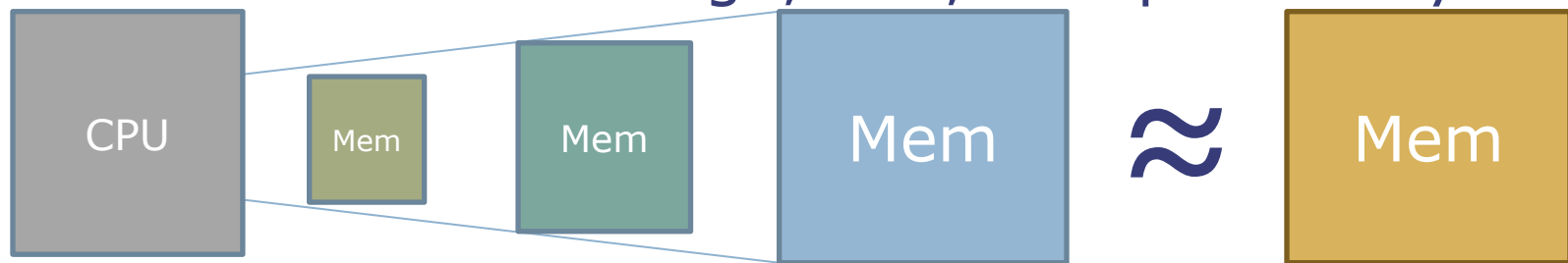
---

Want large, fast, and cheap memory, but...

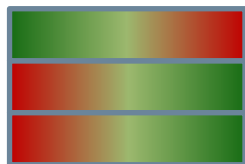
Large memories are slow (even if built with fast components)

Fast memories are expensive

Solution: Use a **hierarchy** of memories with different tradeoffs to **fake** a large, fast, cheap memory



Speed: **Fastest**  
Capacity: **Smallest**  
Cost: **Highest**



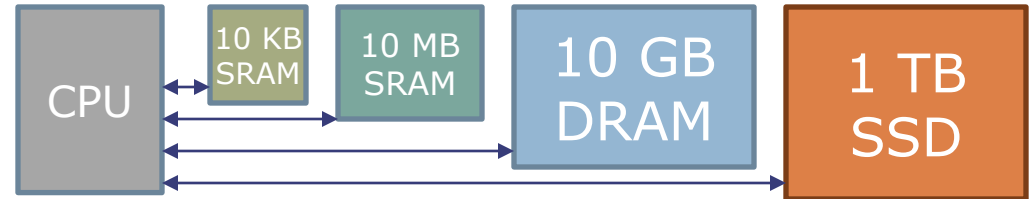
**Slowest**  
**Largest**  
**Lowest**

**Fast**  
**Large**  
**Cheap**

# Memory Hierarchy Interface

## Approach 1: Expose Hierarchy

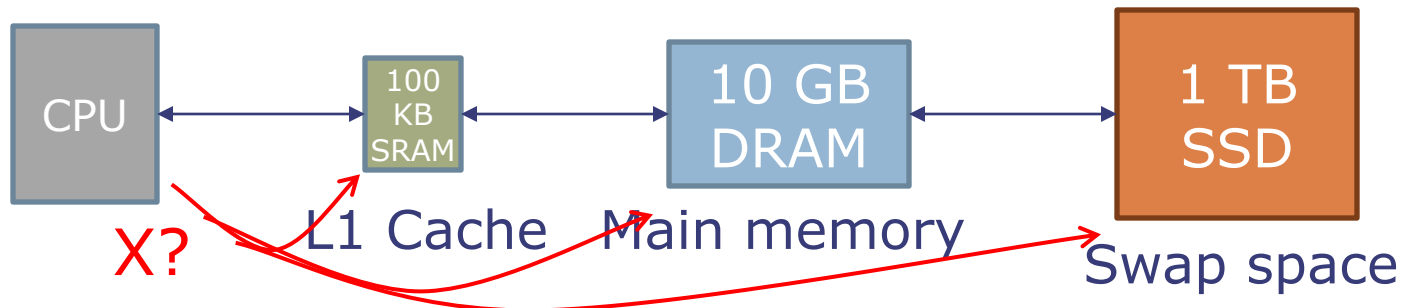
- Registers, SRAM, DRAM, Flash, Hard Disk each available as storage alternatives



- Tell programmers: "Use them cleverly"

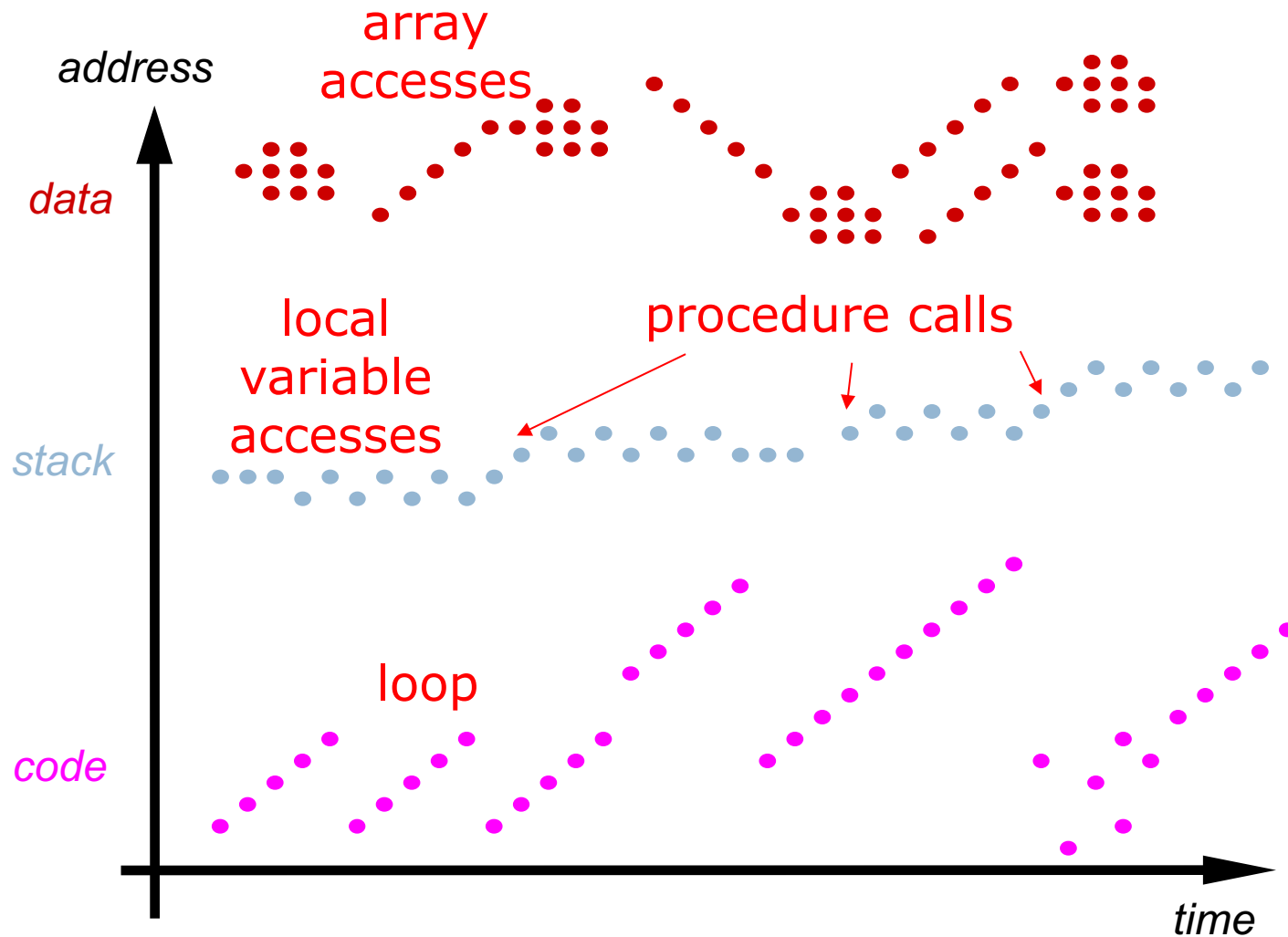
## Approach 2: Hide Hierarchy

- Programming model: Single memory, single address space
- Machine transparently stores data in fast or slow memory, depending on usage patterns



# Typical Memory Access Patterns

---



# Common Predictable Patterns

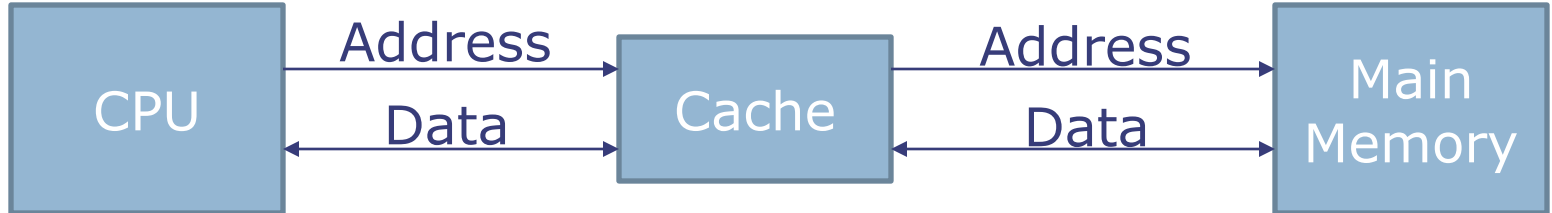
---

- Two predictable properties of memory accesses:
  - **Temporal locality**: If a location has been accessed recently, it is likely to be accessed (reused) soon
  - **Spatial locality**: If a location has been accessed recently, it is likely that nearby locations will be accessed soon

# Caches

---

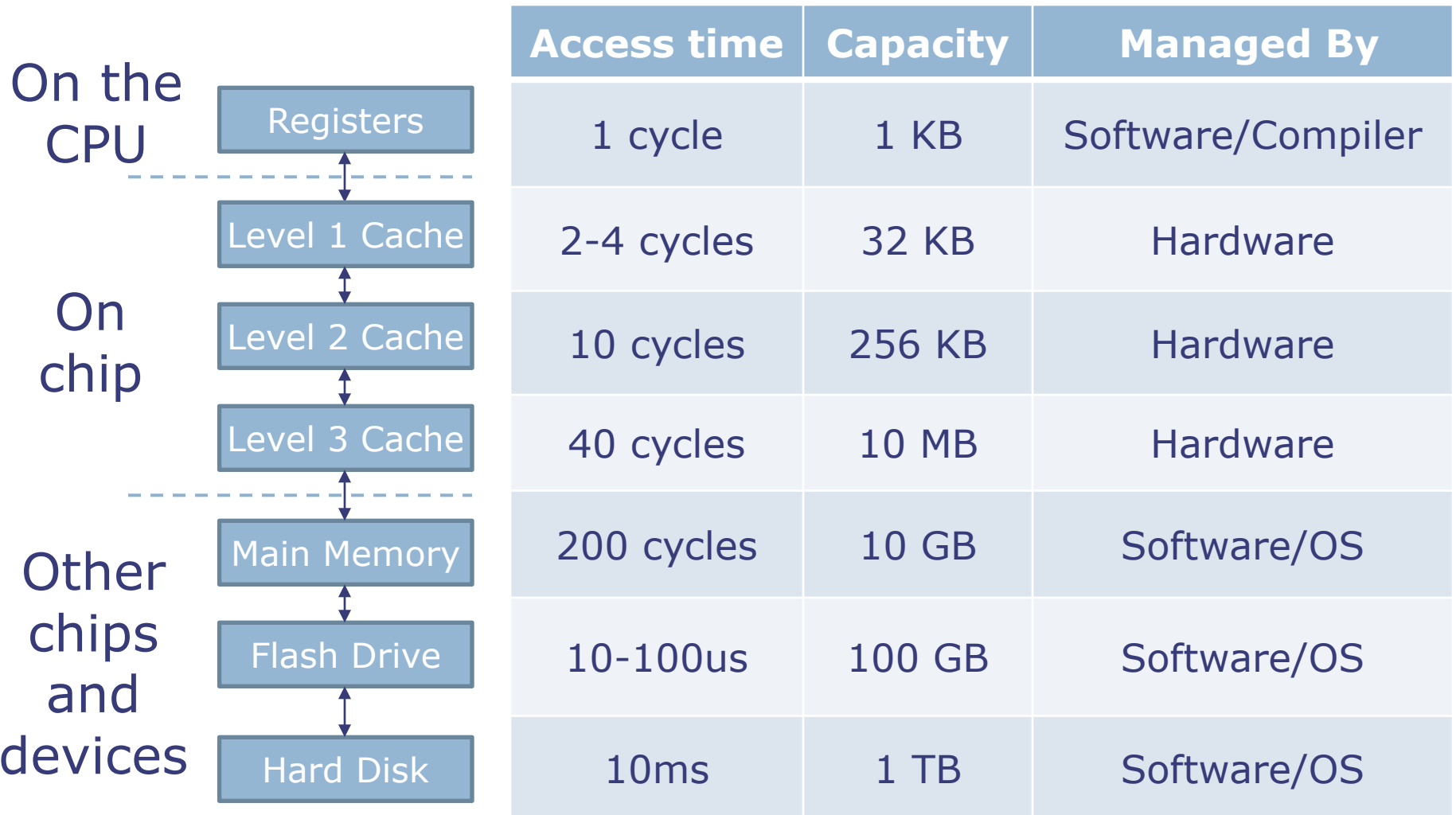
- Cache: A small, interim storage component that transparently retains (caches) data from recently accessed locations



- Processor sends accesses to cache. Two options:
  - Cache hit:** Data for this address in cache, returned quickly
  - Cache miss:** Data not in cache
    - Fetch data from memory, send it back to processor
    - Retain this data in the cache (replacing some other data)
- Processor must deal with variable memory access time

# A Typical Memory Hierarchy

Computers use many levels of caches:





# Cache Metrics

---

- Hit Ratio:  $HR = \frac{hits}{hits + misses} = 1 - MR$

- Miss Ratio:  $MR = \frac{misses}{hits + misses} = 1 - HR$

- Average Memory Access Time (AMAT):

$$AMAT = HitTime + MissRatio \times MissPenalty$$

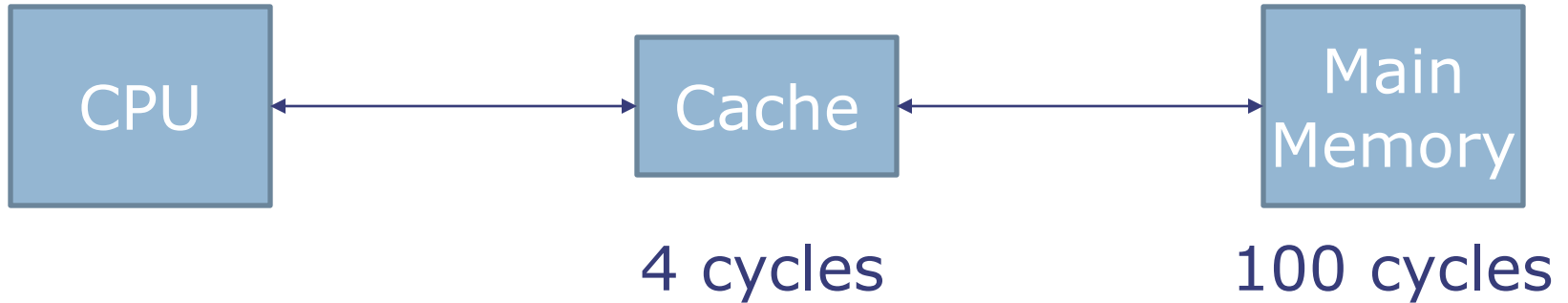
- Goal of caching is to improve AMAT
- Formula can be applied recursively in multi-level hierarchies:

$$AMAT = HitTime_{L1} + MissRatio_{L1} \times AMAT_{L2} =$$

$$AMAT = HitTime_{L1} + MissRatio_{L1} \times (HitTime_{L2} + MissRatio_{L2} \times AMAT_{L3}) = \dots$$

# Example: How High of a Hit Ratio?

---



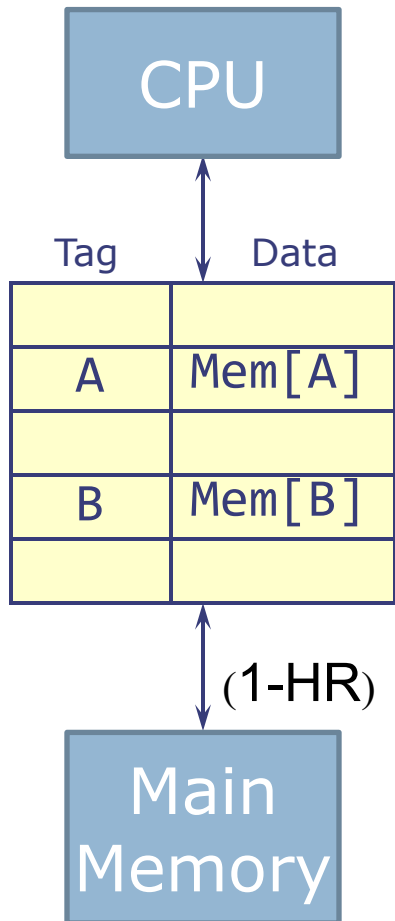
What hit ratio do we need to break even?  
(Main memory only: AMAT = 100)

$$100 = 4 + (1 - HR) \times 100 \Rightarrow HR = 4\%$$

What hit ratio do we need to achieve AMAT = 5 cycles?

$$5 = 4 + (1 - HR) \times 100 \Rightarrow HR = 99\%$$

# Basic Cache Algorithm (Reads)



On reference to Mem[X],  
look for X among cache tags

HIT:  $X = \text{Tag}(i)$   
for some  
cache line  $i$

Return Data( $i$ )

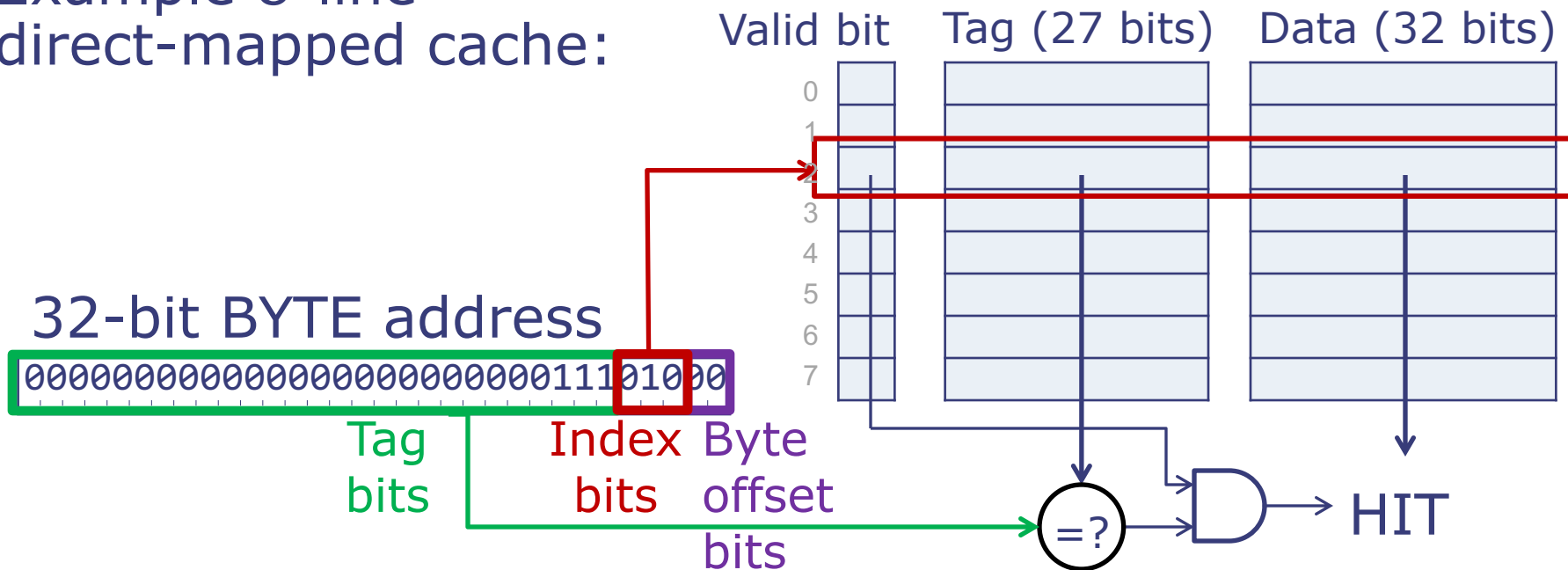
MISS: X not  
found in Tag  
of any cache line

Read Mem[X]  
Return Mem[X]  
Select a line  $k$   
to hold Mem[X]  
Write Tag( $k$ ) = X,  
Data( $k$ ) = Mem[X]

*Q: How do we "search" the cache?*

# Direct-Mapped Caches

- Each word in memory maps into a single cache line
- Access (for cache with  $2^W$  lines):
  - Index into cache with  $W$  address bits (the **index bits**)
  - Read out valid bit, tag, and data
  - If valid bit == 1 and tag matches upper address bits, HIT
- Example 8-line direct-mapped cache:



# Example: Direct-Mapped Caches

64-line direct-mapped cache → 64 indices → 6 index bits

*Read Mem[0x400C]*

0100 0000 0000 1100  
TAG: 0x40  
INDEX: 0x3  
BYTE OFFSET: 0x0

HIT, DATA 0x42424242

*Would 0x4008 hit?*

INDEX: 0x2 → tag mismatch  
→ MISS

	Valid bit	Tag (24 bits)	Data (32 bits)
0	1	0x000058	0xDEADBEEF
1	1	0x000058	0x00000000
2	1	0x000058	0x00000007
3	1	0x000040	0x42424242
4	0	0x000007	0x6FBA2381
	⋮	⋮	⋮
63	1	0x000058	0xF7324A32

Part of the address (index bits) is encoded in the location  
Tag + Index bits unambiguously identify the data's address

Thank you!

*Next lecture: Cache  
Tradeoffs*