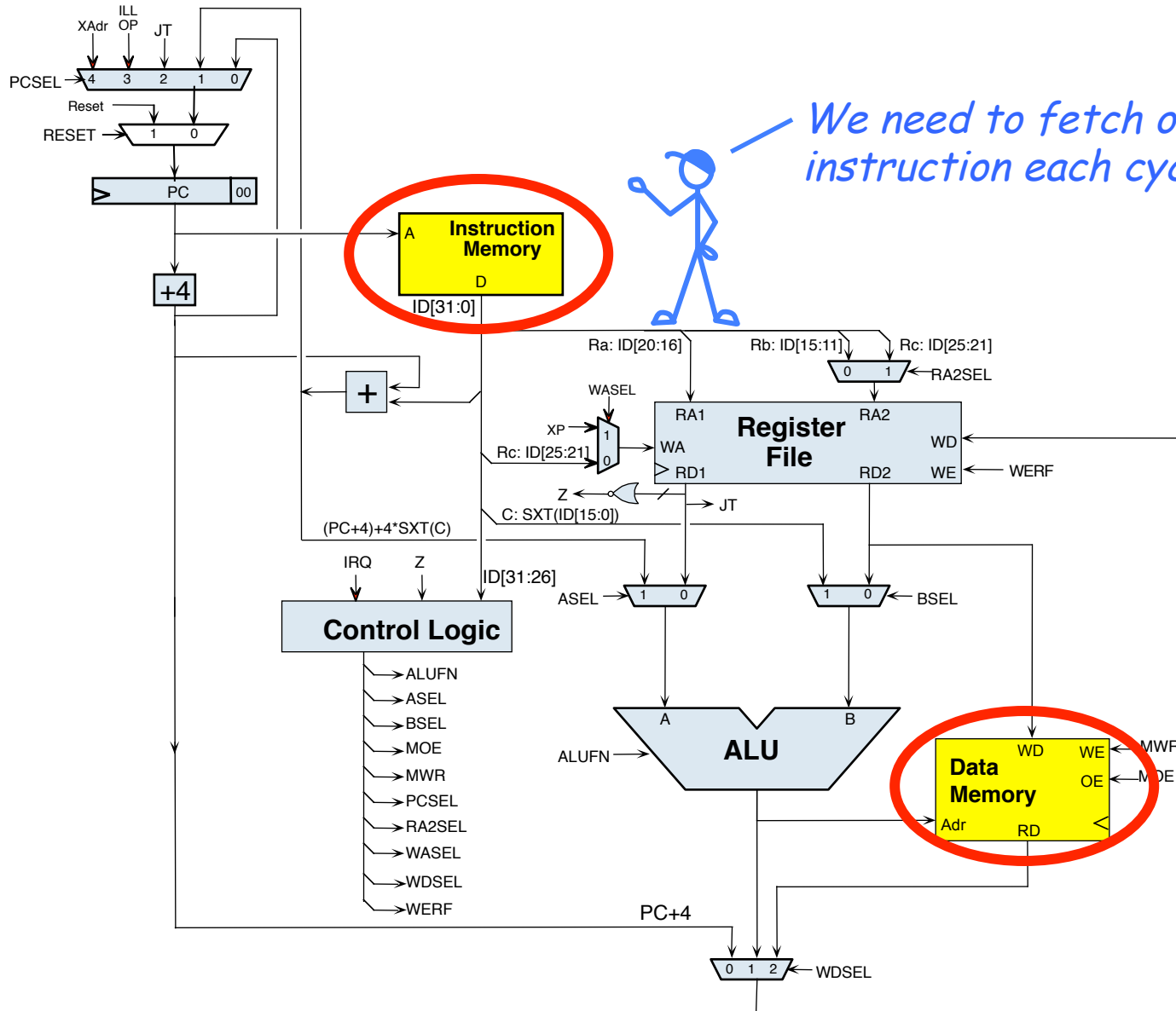


# 14. Caches & The Memory Hierarchy

6.004x Computation Structures  
Part 2 – Computer Architecture

Copyright © 2016 MIT EECS

# Our ~~Computing~~ **Memory** Machine"



We need to fetch one instruction each cycle

Ultimately data is loaded from and results stored to memory

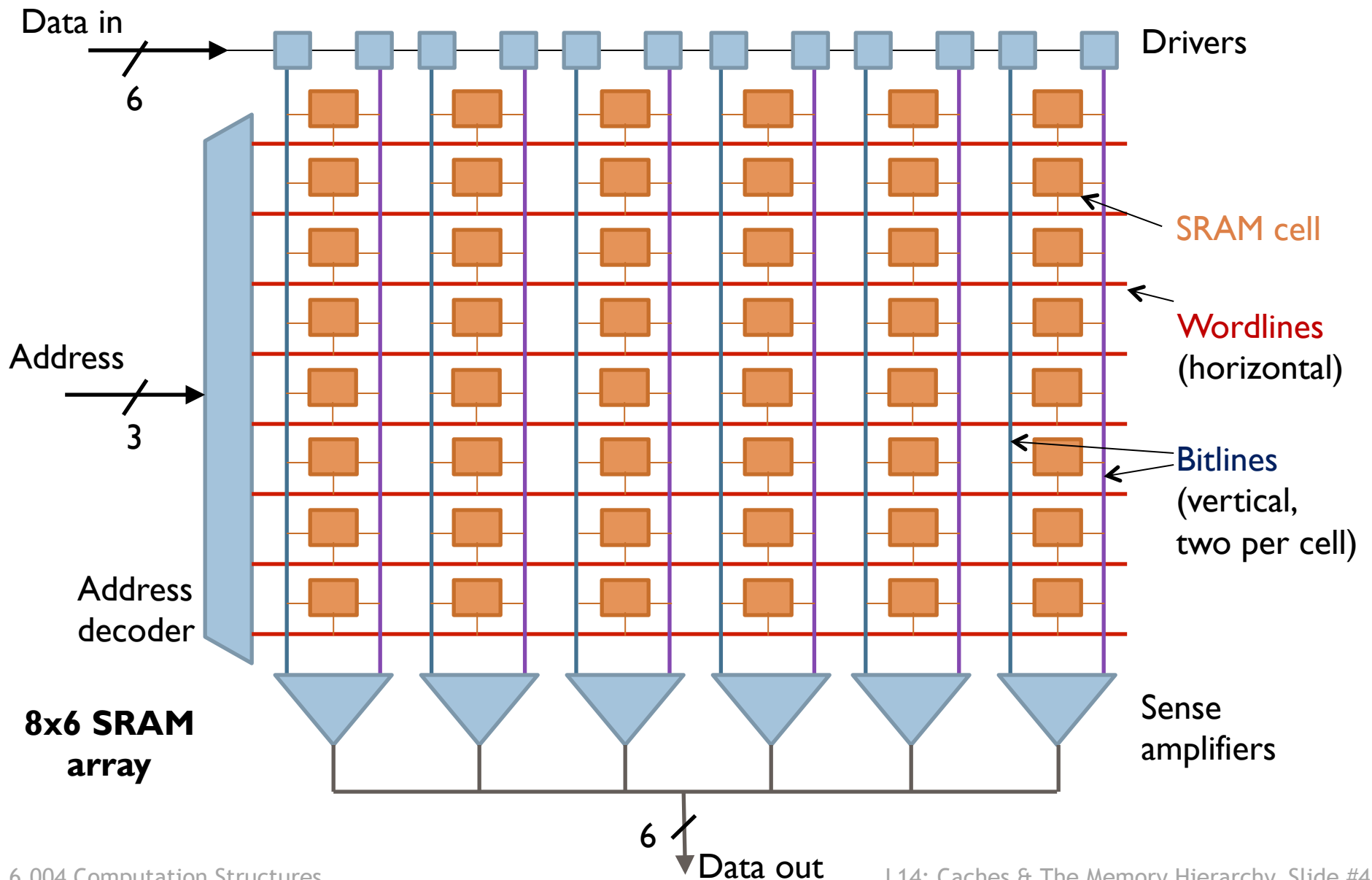
# Memory Technologies

Technologies have vastly different tradeoffs between capacity, access latency, bandwidth, energy, and cost  
– ... and logically, different applications

	Capacity	Latency	Cost/GB	
Register	1000s of bits	20 ps	\$\$\$\$	Processor Datapath
SRAM	~10 KB-10 MB	1-10 ns	~\$1000	
DRAM	~10 GB	80 ns	~\$10	I/O subsystem
Flash*	~100 GB	100 us	~\$1	
Hard disk*	~1 TB	10 ms	~\$0.10	

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

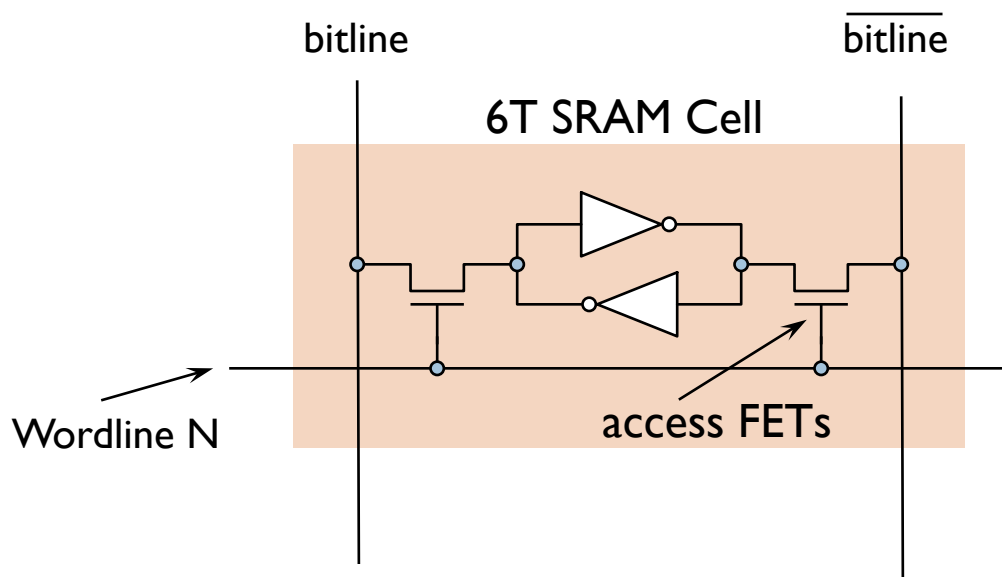
# Static RAM (SRAM)



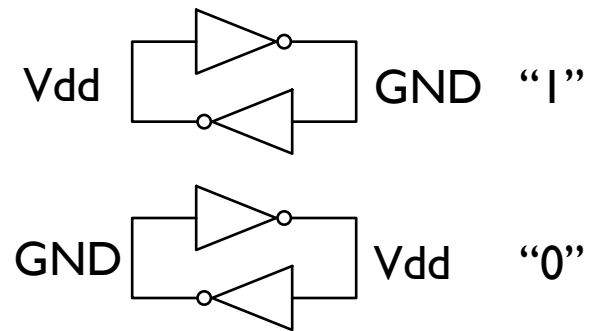
# SRAM Cell

## 6-MOSFET (6T) cell:

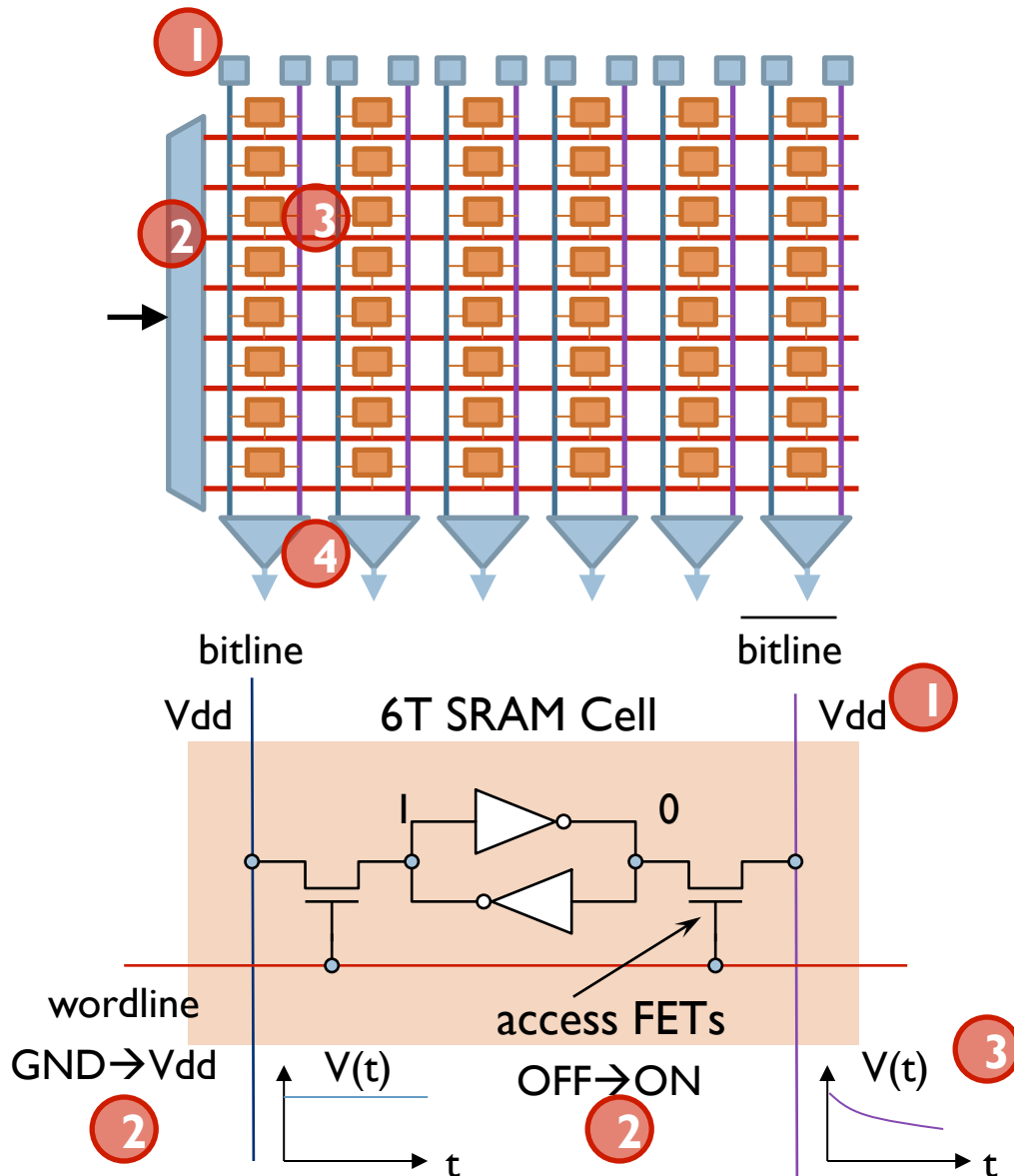
- Two CMOS inverters (4 MOSFETs) forming a **bistable element**
- Two **access transistors**



Bistable element  
(two stable states)  
stores a single bit

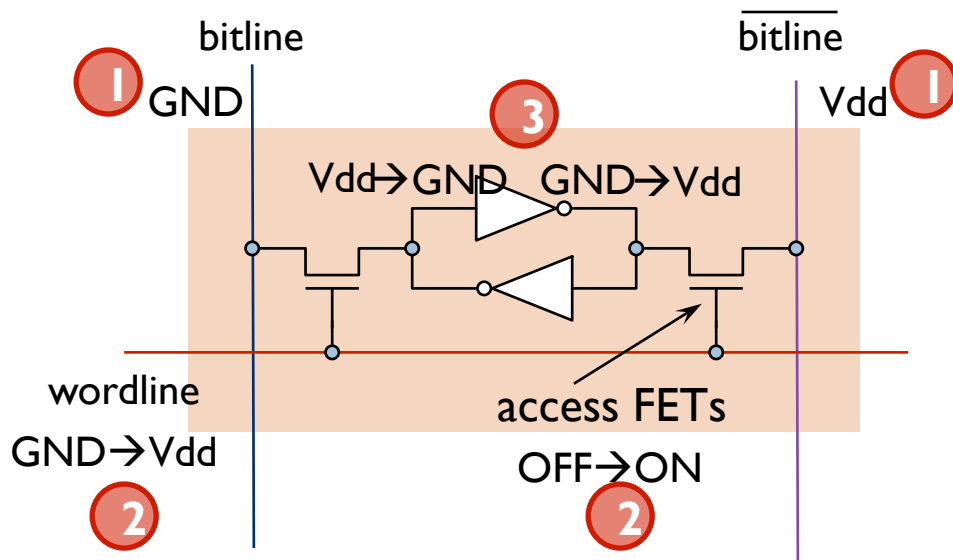
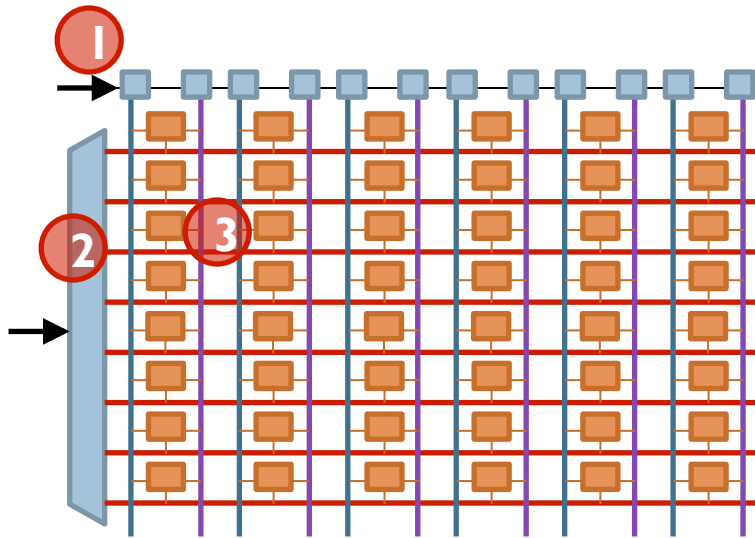


# SRAM Read



1. Drivers precharge all bitlines to Vdd (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, producing output data

# SRAM Write

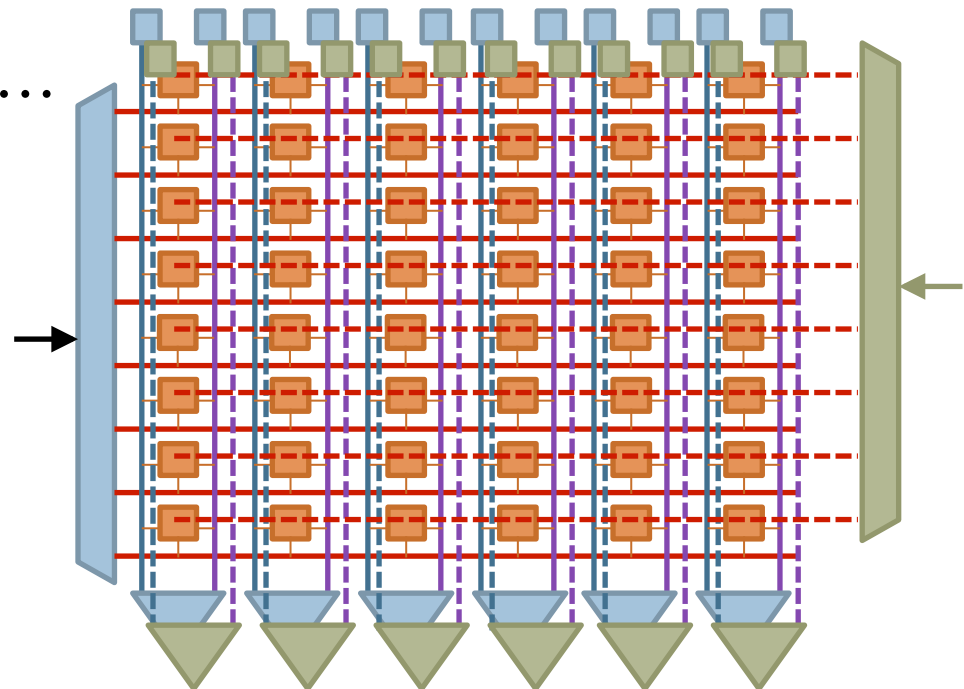


1. Drivers set and hold bitlines to desired values (Vdd and GND for 1, GND and Vdd for 0)
2. Address decoder activates one wordline
3. Each cell in word is overpowered by the drivers, stores value

All transistors are carefully sized so that bitline GND overpowers cell Vdd, but bitline Vdd does not overpower cell GND (why?)

# 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:  $\frac{N}{2}$
  - Bitlines:  $\frac{2*N}{2}$
  - Access FETs:  $\frac{2*N}{2}$
- Wires often dominate area  $\rightarrow O(N^2)$  area!

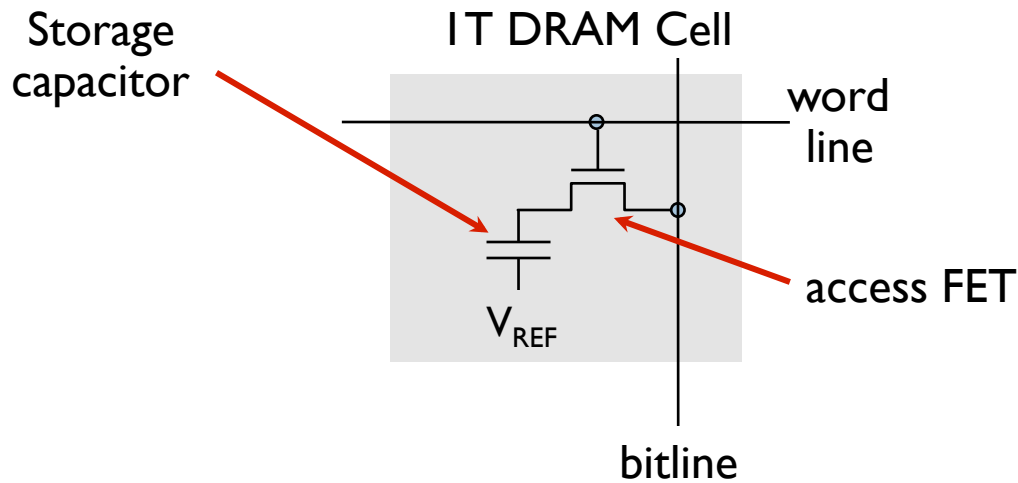




# Summary: SRAMs

- Array of  $k \cdot b$  cells ( $k$  words,  $b$  cells per word)
- Cell is a bistable element + access transistors
  - Analog circuit with carefully sized transistors to allow reads and writes
- Read: Precharge bitlines, activate wordline, sense
- Write: Drive bitlines, activate wordline, overpower cells
  
- 6 MOSFETs/cell... can we do better?
  - What's the minimum number of MOSFETs needed to store a single bit?

# 1T Dynamic RAM (DRAM) Cell



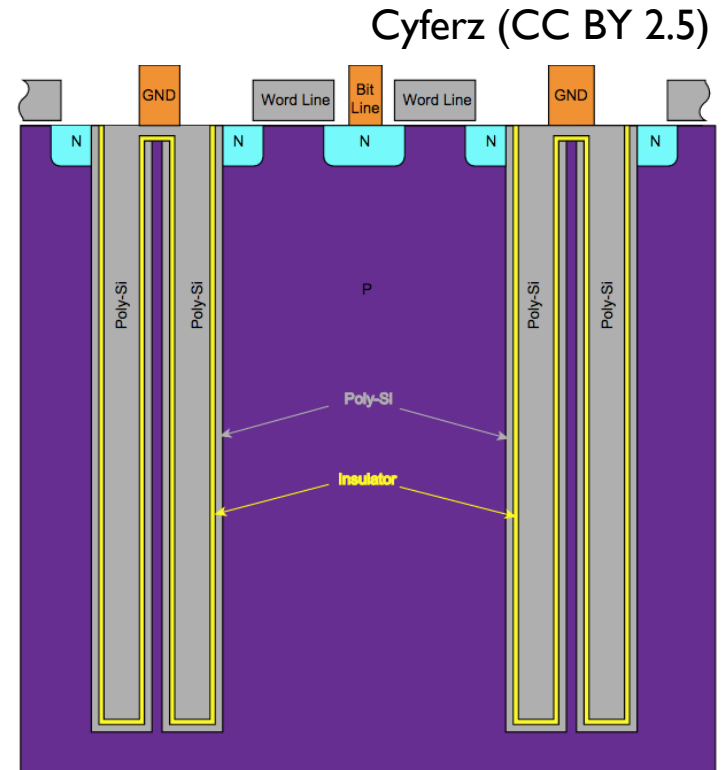
C in storage capacitor determined by:

better dielectric

more area

$$C = \frac{\epsilon A}{d}$$

thinner film



Trench capacitors  
take little area

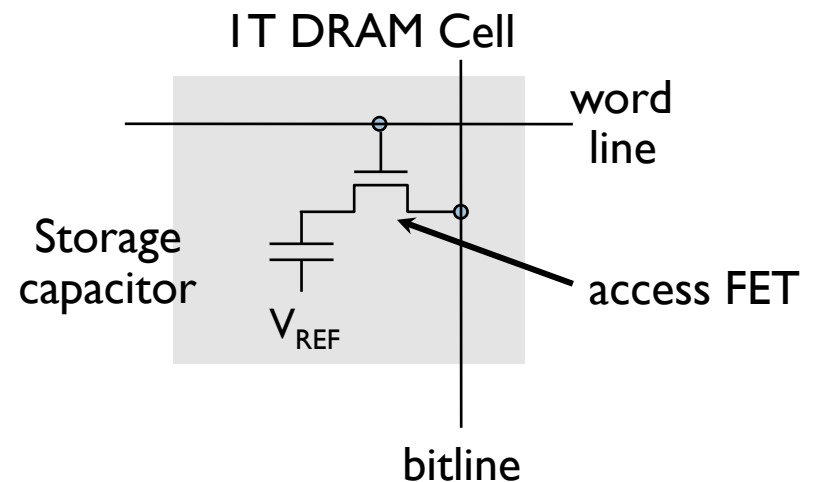
- ✓ ~20x smaller area than SRAM cell → Denser and cheaper!
- ✗ Problem: Capacitor leaks charge, must be refreshed periodically (~milliseconds)

# DRAM Writes and Reads

- Writes: Drive bitline to Vdd or GND, activate wordline, charge or discharge capacitor

- Reads:

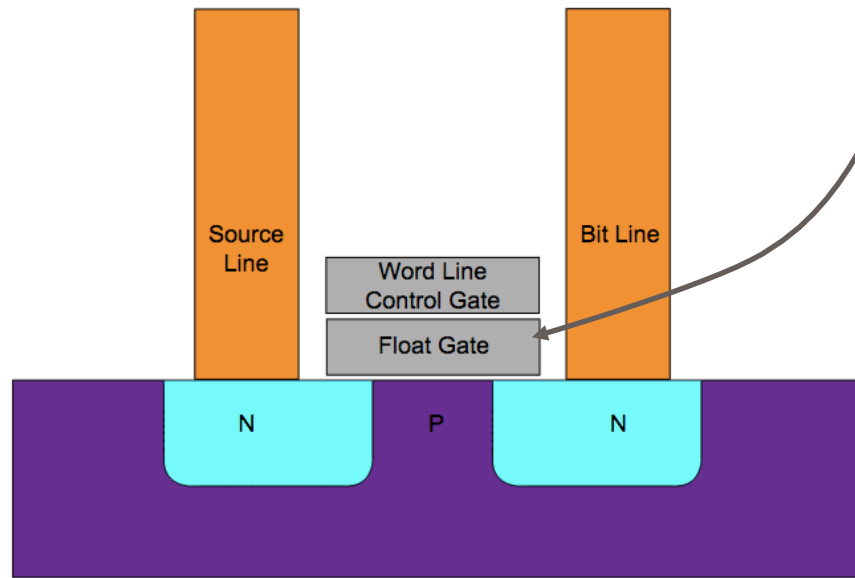
1. Precharge bitline to  $V_{dd}/2$
2. Activate wordline
3. Capacitor and bitline share charge
  - If capacitor was discharged, bitline voltage decreases slightly
  - If capacitor was charged, bitline voltage increases slightly
4. Sense bitline to determine if 0 or 1
  - Issue: **Reads are destructive!**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



# Summary: DRAM

- 1T DRAM cell: transistor + capacitor
- **Smaller** than SRAM cell, but **destructive reads** and **capacitors leak** charge
- DRAM arrays include circuitry to:
  - Write word again after every read (to avoid losing data)
  - Refresh (read+write) every word periodically
- DRAM vs SRAM:
  - **~20x denser than SRAM**
  - **~2-10x slower than SRAM**

# Non-Volatile Storage: Flash



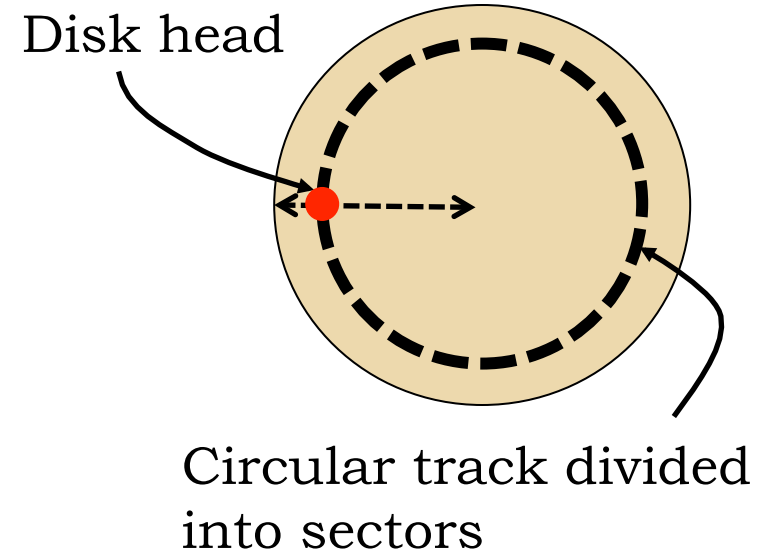
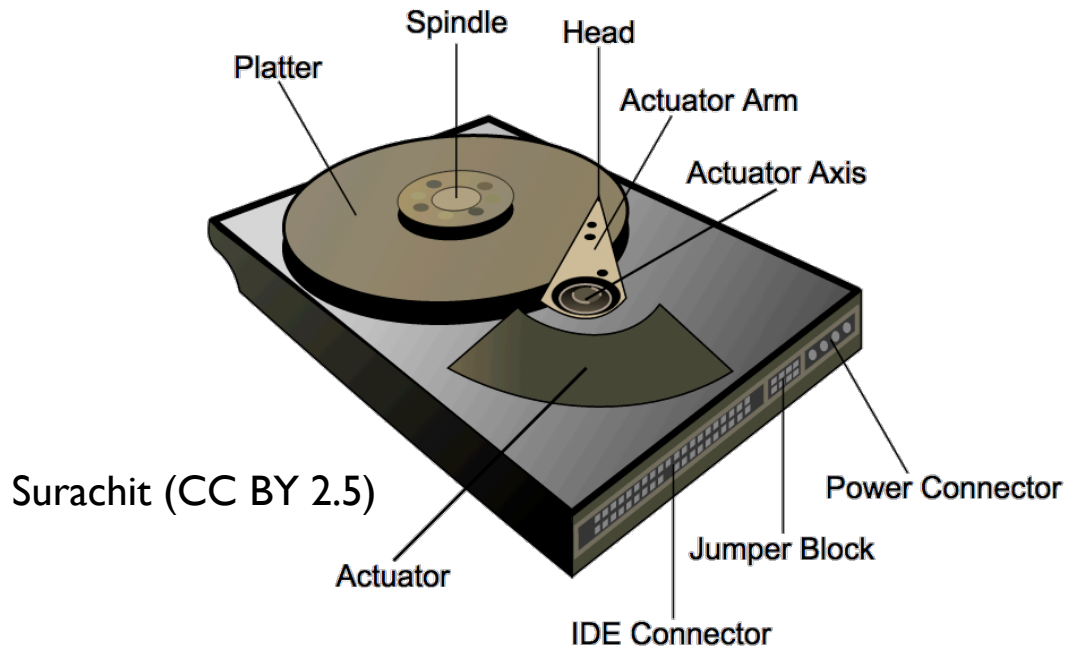
Electrons here diminish strength of field from control gate  $\Rightarrow$  no inversion  $\Rightarrow$  NFET stays off even when word line is high.

Cyferz (CC BY 2.5)

Flash Memory: Use “floating gate” transistors to store charge

- **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



Hard Disk: Rotating magnetic platters + read/write head

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

# Summary: Memory Technologies

	Capacity	Latency	Cost/GB
Register	1000s 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	~\$0.10

- 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 the best of both worlds? (large, fast, cheap)

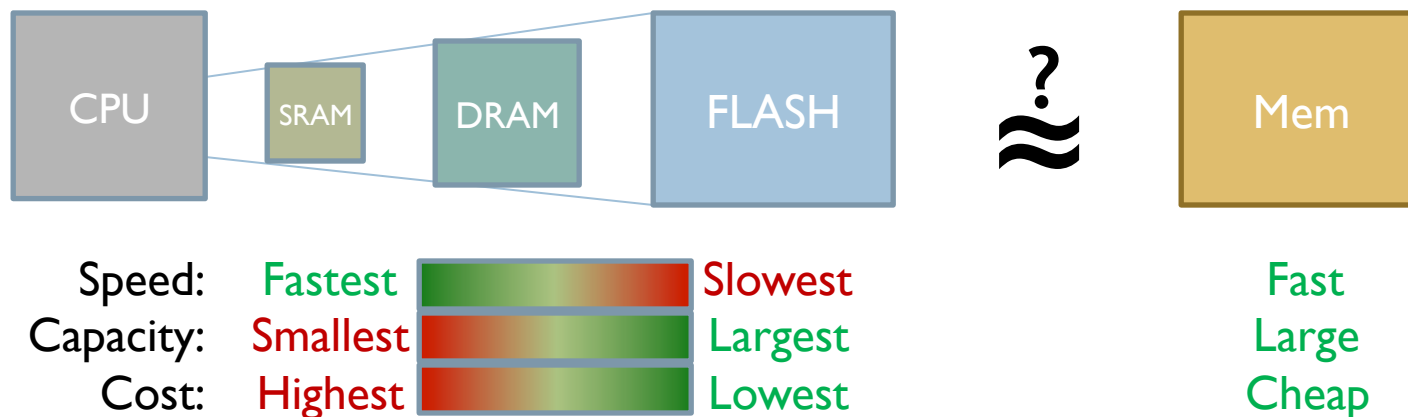
# The Memory Hierarchy

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

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

Fast memories are expensive

Idea: Can we use a **hierarchal system** of memories with different tradeoffs to **emulate** a large, fast, cheap memory?

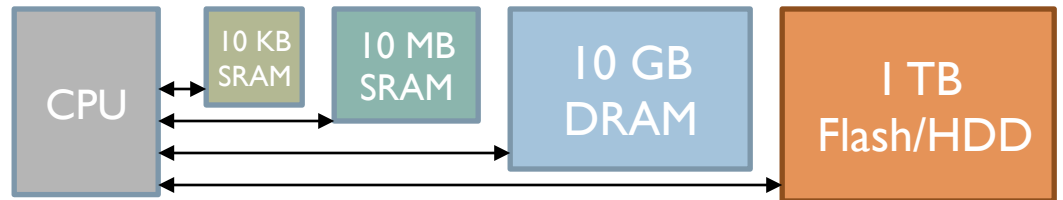




# Memory Hierarchy Interface

## Approach 1: Expose Hierarchy

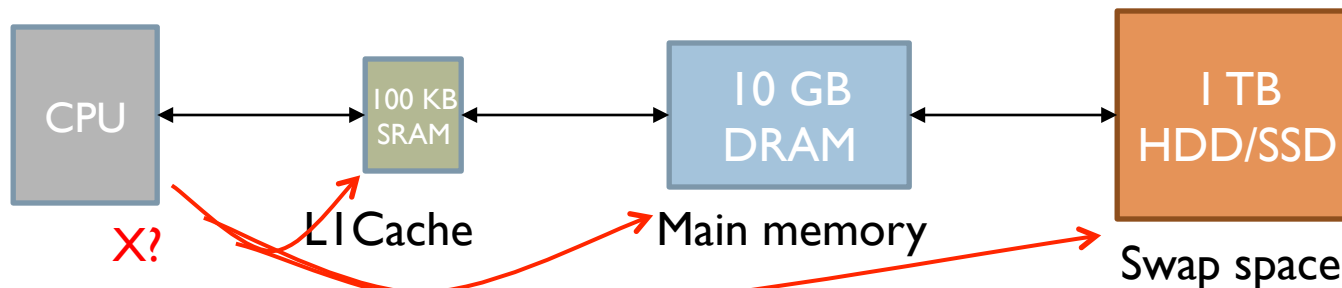
- Registers, SRAM, DRAM, Flash, Hard Disk each available as storage alternatives



- 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



# The Locality Principle

Keep the most often-used data in a small, fast SRAM (often local to CPU chip)

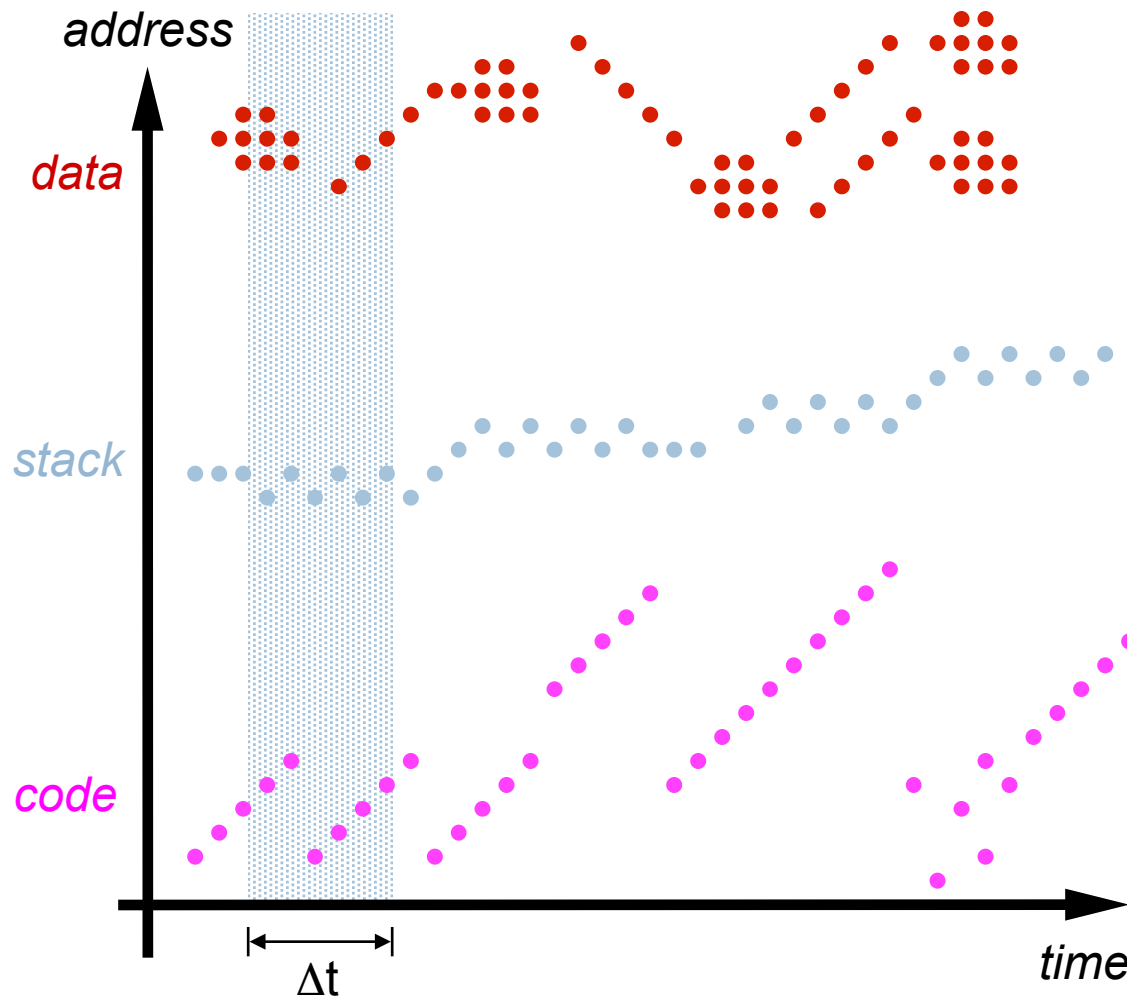
Refer to Main Memory only rarely, for remaining data.

The reason this strategy works: LOCALITY

## Locality of Reference:

Access to address  $X$  at time  $t$  implies that access to address  $X + \Delta X$  at time  $t + \Delta t$  becomes more probable as  $\Delta X$  and  $\Delta t$  approach zero.

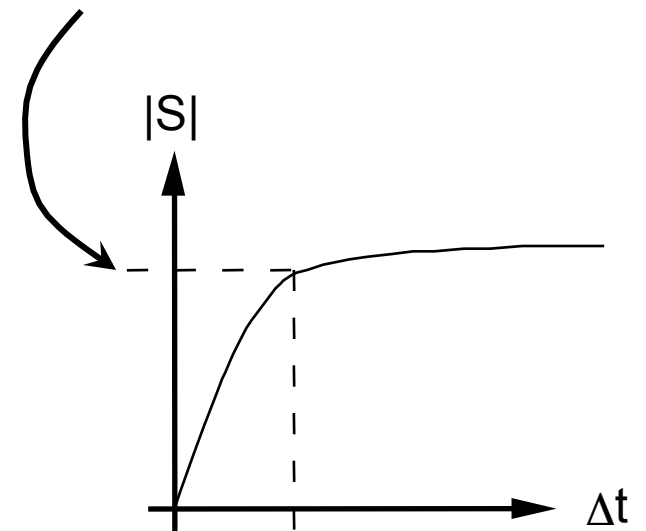
# Memory Reference Patterns



$S$  is the set of locations accessed during  $\Delta t$ .

*Working set*: a set  $S$  which changes slowly wrt access time.

*Working set size*,  $|S|$



# Caches

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

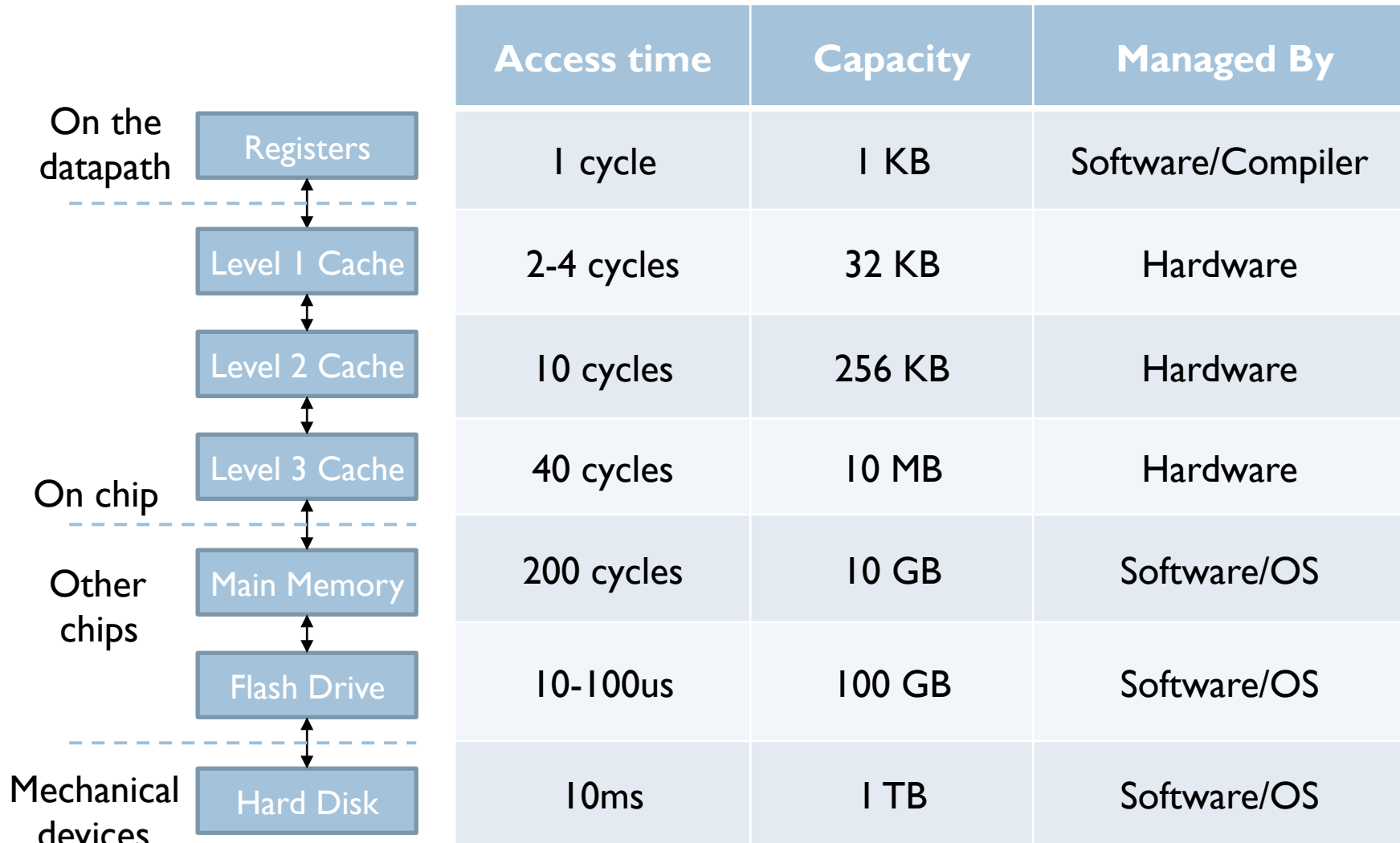
- Very fast access if data is cached, otherwise accesses slower, larger cache or memory
- Exploits the locality principle

Computer systems often use multiple levels of caches

Caching widely applied beyond hardware (e.g., web caches)

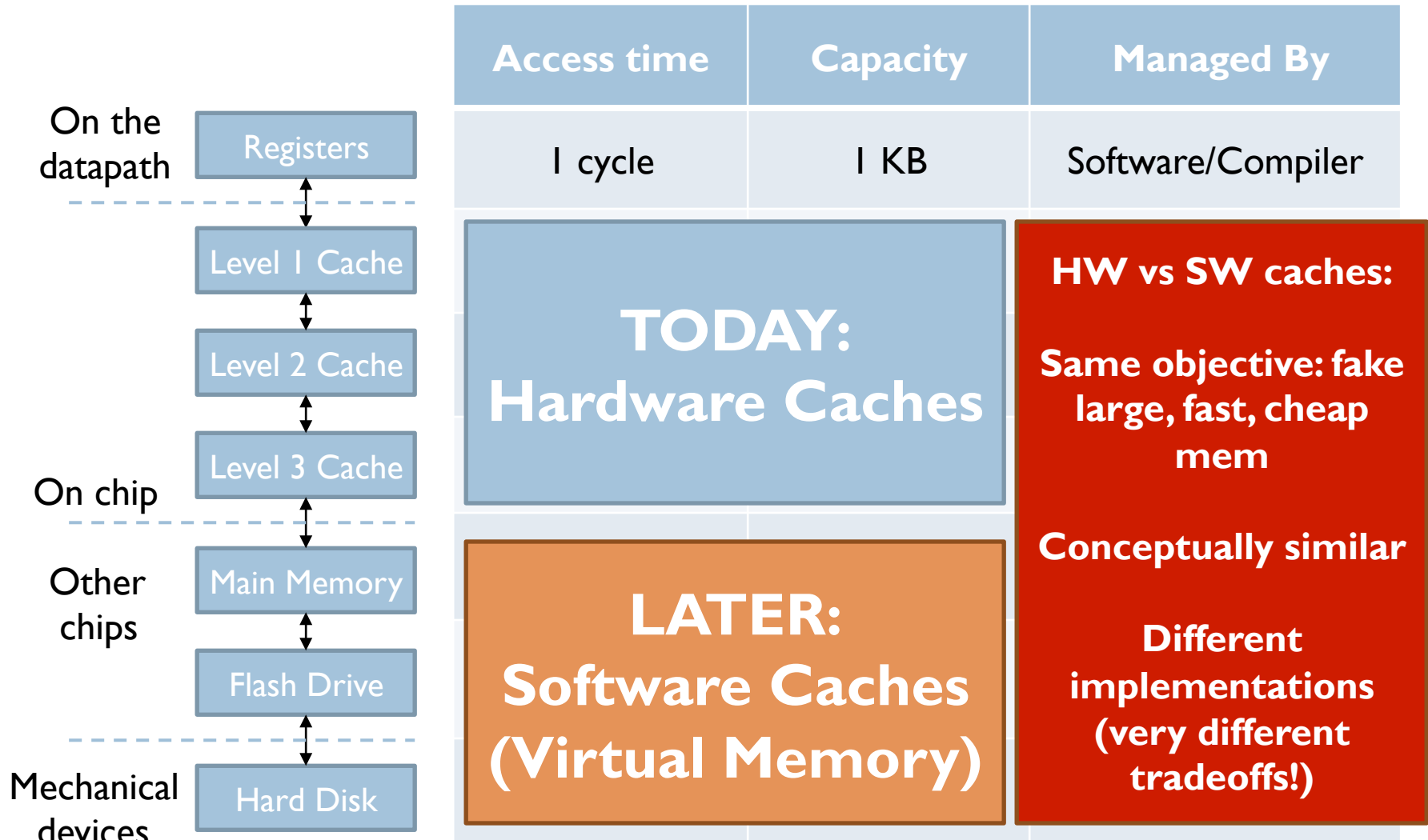
# A Typical Memory Hierarchy

- Everything is a cache for something else...

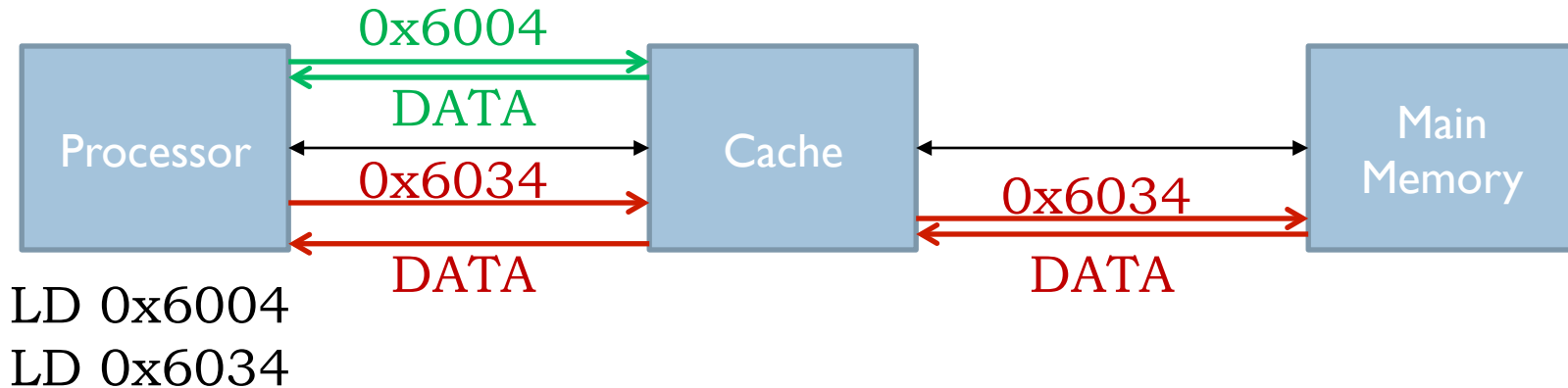


# A Typical Memory Hierarchy

- Everything is a cache for something else...



# Cache Access



- Processor sends address 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

# 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):

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

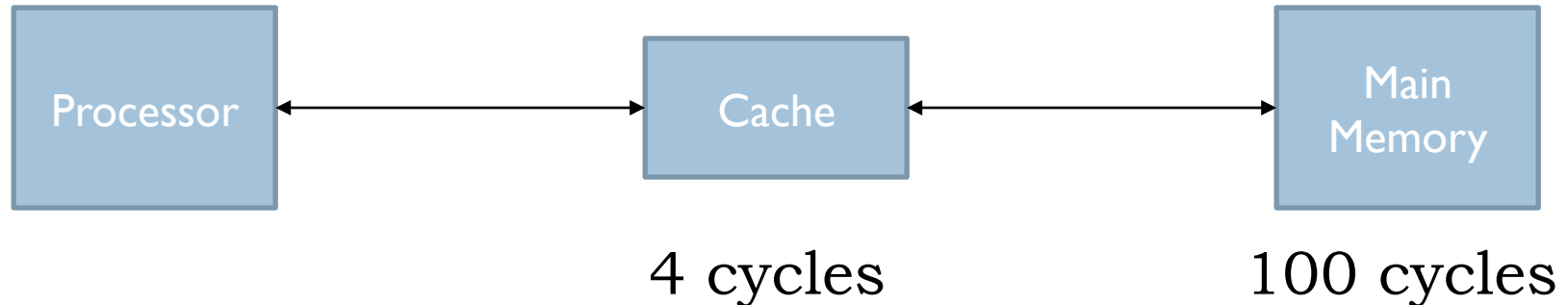
- Goal of caching is to improve AMAT
- Formula can be applied recursively in multi-level hierarchies:

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

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



# Example: How High of a Hit Ratio?



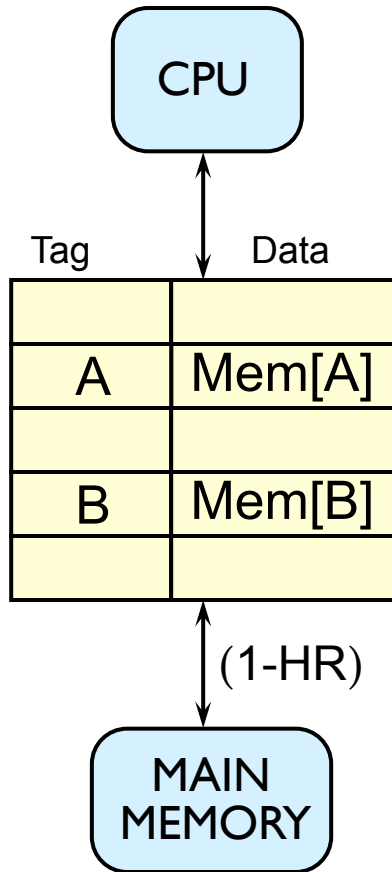
What hit ratio do we need to break even?  
(Main memory only: AMAT = 100)

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

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

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

# Basic Cache Algorithm



ON REFERENCE TO Mem[X]:

Look for X among cache tags...

HIT:  $X = TAG(i)$ , for some cache line  $i$

- READ: return DATA( $i$ )
- WRITE: change DATA( $i$ ); Start Write to Mem(X)

MISS:  $X$  not found in TAG of any cache line

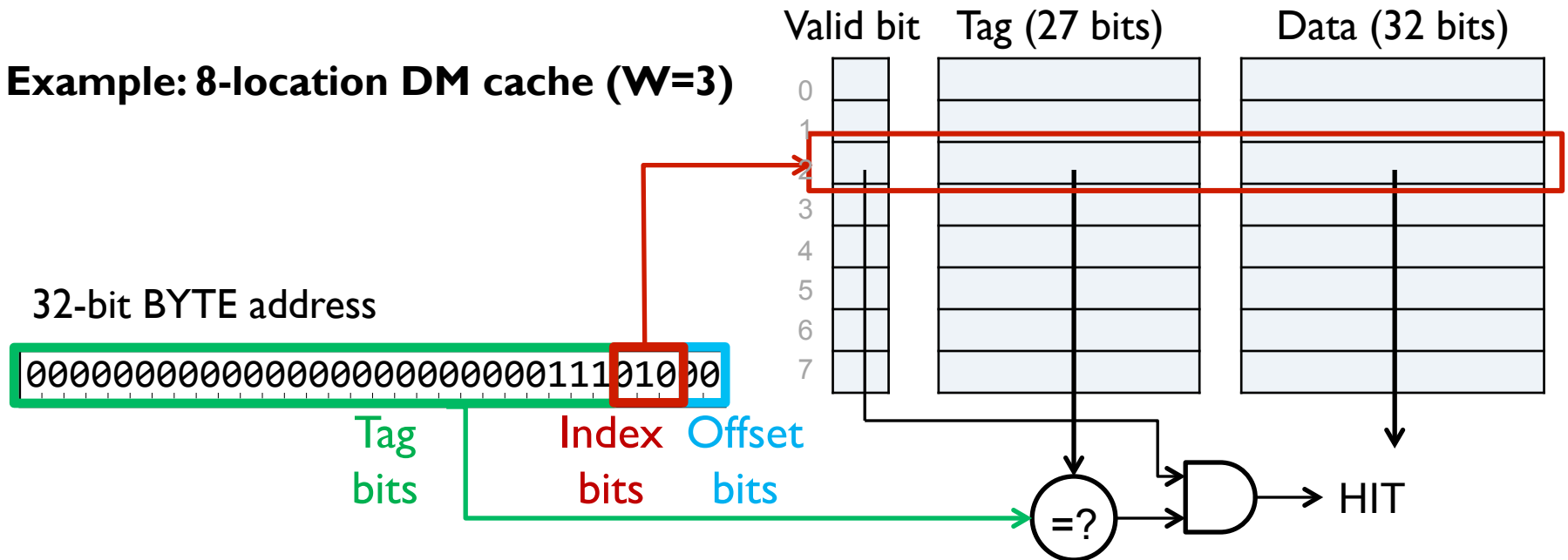
- REPLACEMENT SELECTION:  
Select some line  $k$  to hold Mem[X] (Allocation)
- READ: Read Mem[X]  
Set TAG( $k$ )=X, DATA( $k$ )=Mem[X]
- WRITE: Start Write to Mem(X)  
Set TAG( $k$ )=X, DATA( $k$ )= new Mem[X]

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-location DM cache ( $W=3$ )**



# Example: Direct-Mapped Caches

64-line direct-mapped cache → 64 indexes → **6 index bits**

Read Mem[0x400C]

0100 0000 0000 1100

TAG: 0x40

INDEX: 0x3

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	0	0x000058	0x00000007
3	1	0x000040	0x42424242
4	1	0x000007	0x6FBA2381
	⋮	⋮	⋮
63	1	0x000058	0xF7324A32

What are the addresses of data in indexes 0, 1, and 2?

TAG: 0x58 → 0101 1000 iiii ii00 (substitute line # for iiiii) → 0x5800, 0x5804, 0x5808

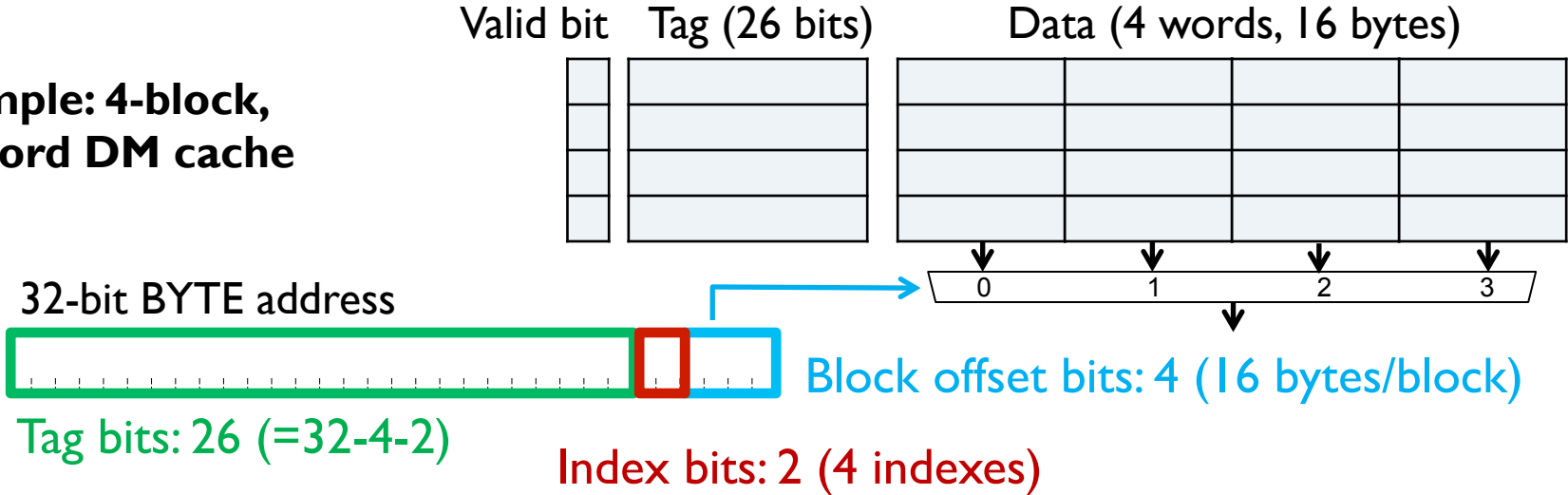
Part of the address (index bits) is **encoded in the location!**  
 Tag + Index bits unambiguously identify the data's address

# Block Size

Take advantage of locality: increase block size

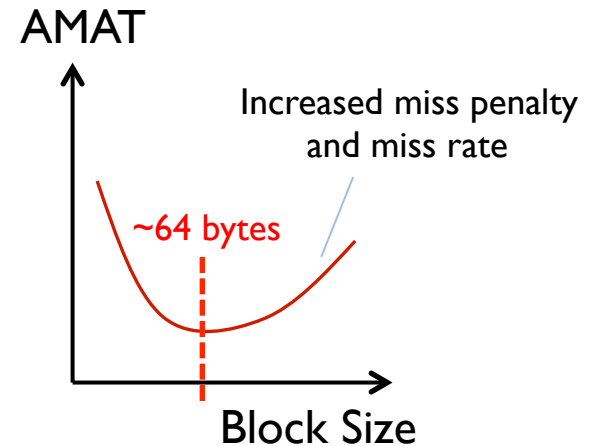
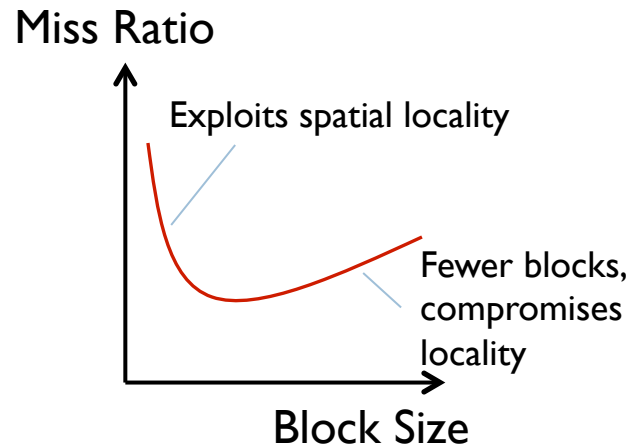
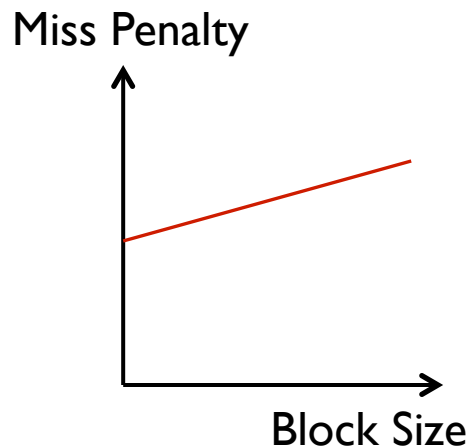
- Another advantage: Reduces size of tag memory!
- Potential disadvantage: Fewer blocks in the cache

**Example: 4-block,  
16-word DM cache**



# Block Size Tradeoffs

- Larger block sizes...
  - Take advantage of spatial locality
  - Incur larger miss penalty since it takes longer to transfer the block into the cache
  - Can increase the average hit time and miss rate
- Average Access Time (AMAT) = HitTime + MissPenalty\*MR



# Direct-Mapped Cache Problem: Conflict Misses

Loop A:  
Pgm at  
1024,  
data at  
37:

Word Address	Cache Line index	Hit/ Miss
1024	0	HIT
37	37	HIT
1025	1	HIT
38	38	HIT
1026	2	HIT
39	39	HIT
1024	0	HIT
37	37	HIT
...		

Assume:  
1024-line DM cache  
Block size = 1 word  
Consider looping code, in steady state  
Assume WORD, not BYTE, addressing

Loop B:  
Pgm at  
1024,  
data at  
2048:

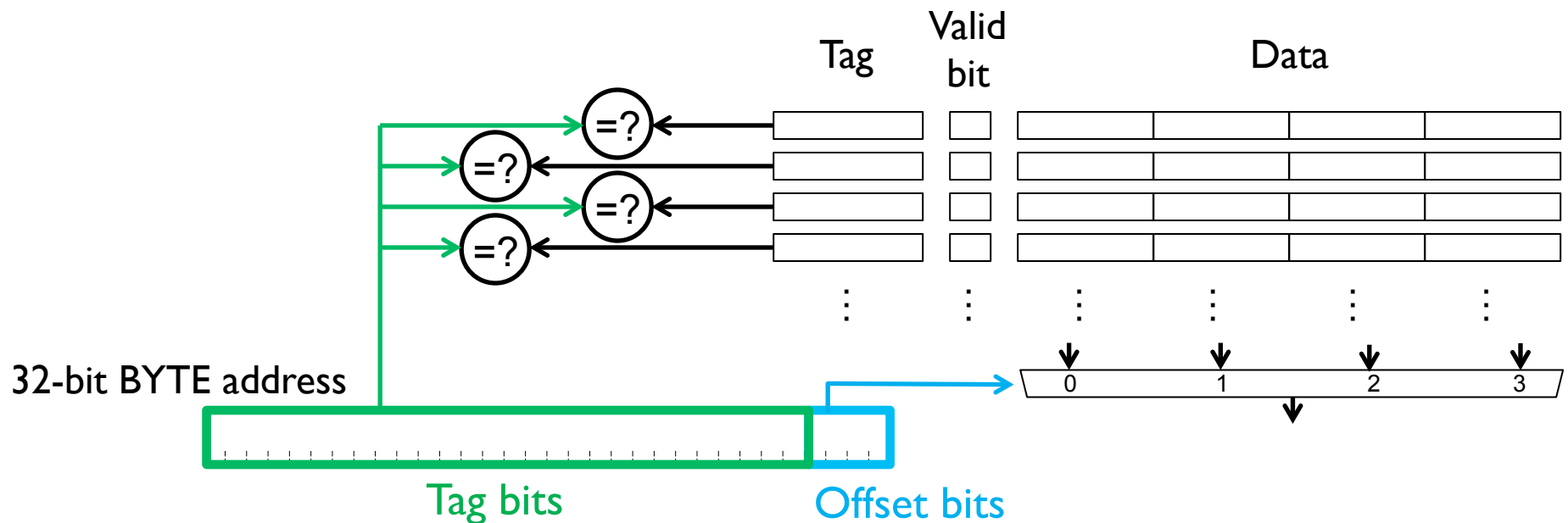
1024	0	MISS
2048	0	MISS
1025	1	MISS
2049	1	MISS
1026	2	MISS
2050	2	MISS
1024	0	MISS
2048	0	MISS
...		

Inflexible mapping (each address can only be in one cache location) → **Conflict misses!**

# Fully-Associative Cache

Opposite extreme: Any address can be in any location

- No cache index!
- **Flexible** (no conflict misses)
- **Expensive**: Must compare tags of all entries in parallel to find matching one (can do this in hardware, this is called a CAM)





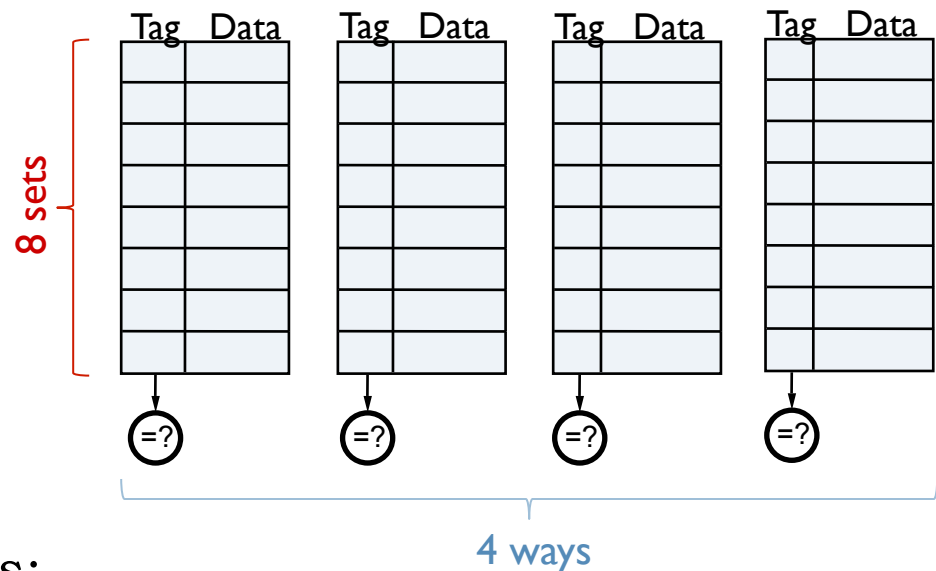
# N-way Set-Associative Cache

- Compromise between direct-mapped and fully associative

- Nomenclature:

- # Rows = # Sets
- # Columns = # Ways
- Set size = #ways  
= “set associativity”  
(e.g., 4-way → 4 entries/set)

- compare all tags from all ways in parallel

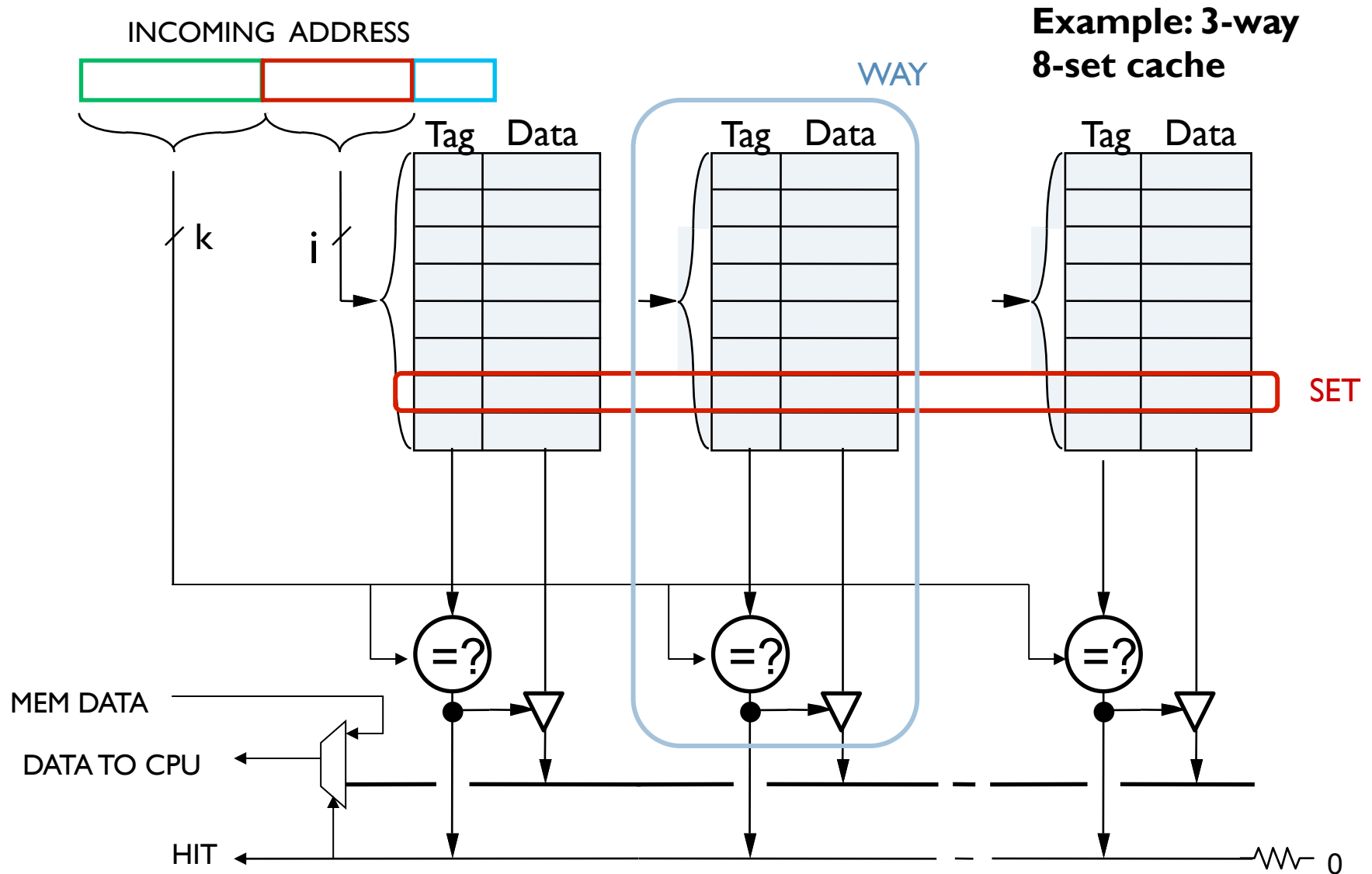


- An N-way cache can be seen as:

- N direct-mapped caches in parallel

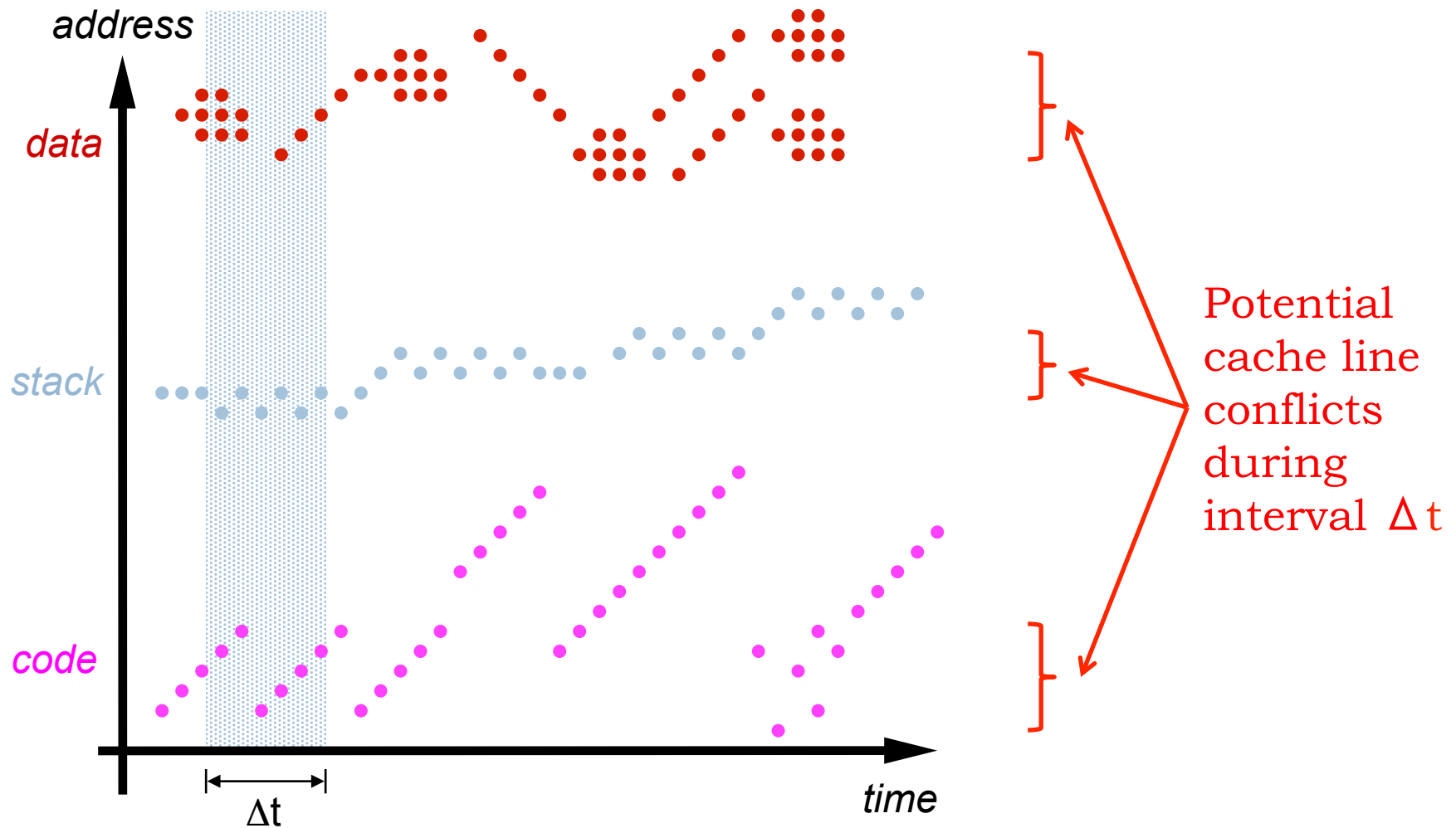
- Direct-mapped and fully-associative are just special cases of N-way set-associative

# N-way Set-Associative Cache



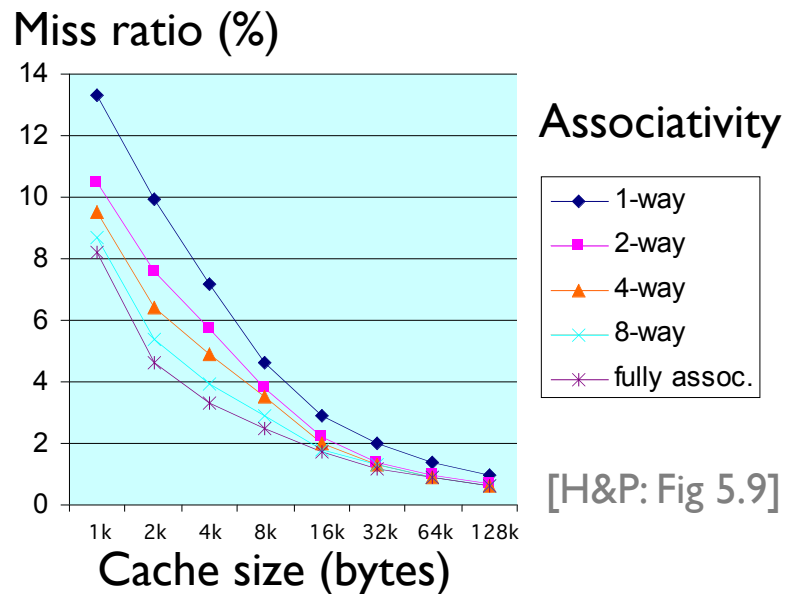
# “Let me count the ways.”

*Elizabeth Barrett Browning*



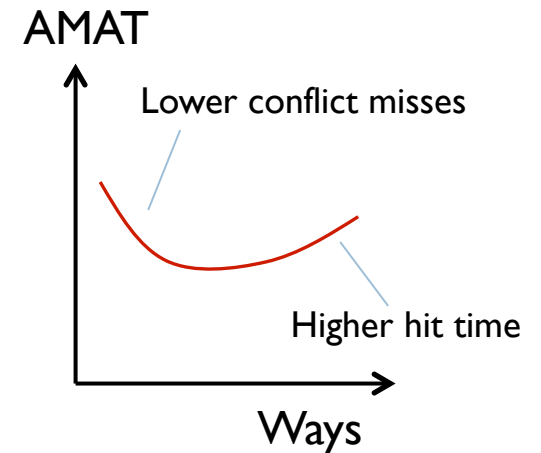
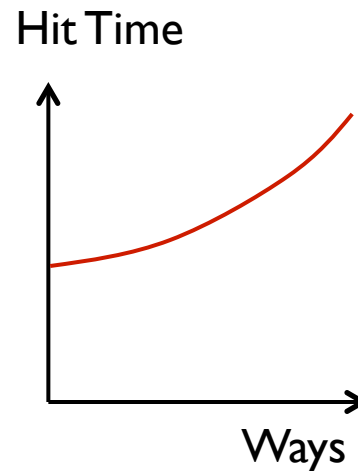
# Associativity Tradeoffs

- More ways...
  - Reduce conflict misses
  - Increase hit time



Little additional benefits  
beyond 4 to 8 ways

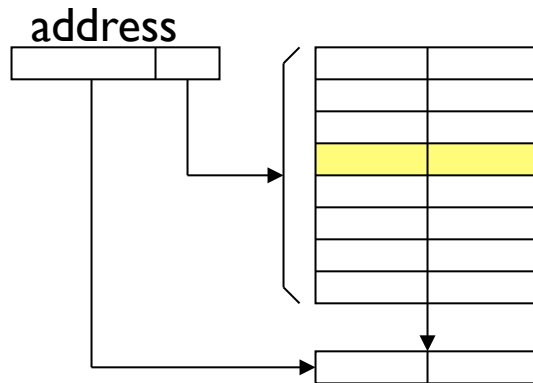
$$AMAT = HitTime + MissRatio \times MissPenalty$$



# Associativity Implies Choices

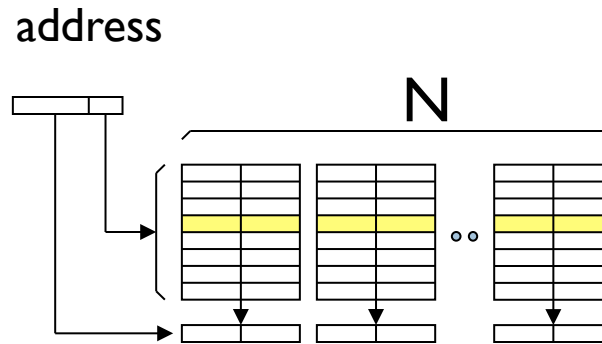
## Issue: Replacement Policy

Direct-mapped



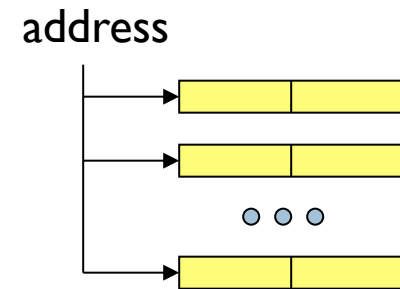
- Compare addr with only one tag
- Location A can be stored in exactly one cache line

N-way set-associative



- Compare addr with N tags simultaneously
- Location A can be stored in exactly one set, but in any of the N cache lines belonging to that set

Fully associative



- Compare addr with each tag simultaneously
- Location A can be stored in any cache line

# Replacement Policies

- Optimal policy (Belady's MIN): Replace the block that is accessed furthest in the future
  - Requires knowing the future...
- Idea: Predict the future from looking at the past
  - If a block has not been used recently, it's often less likely to be accessed in the near future (a locality argument)
- **Least Recently Used (LRU)**: Replace the block that was accessed furthest in the past
  - Works well in practice
  - Need to keep ordered list of  $N$  items  $\rightarrow N!$  orderings  
 $\rightarrow O(\log_2 N!) = O(N \log_2 N)$  "LRU bits" + complex logic
  - Caches often implement cheaper approximations of LRU
- Other policies:
  - First-In, First-Out (least recently replaced)
  - Random: Choose a candidate at random
    - Not very good, but does not have adversarial access patterns

# Write Policy

**Write-through:** CPU writes are cached, but also written to main memory immediately (stalling the CPU until write is completed). Memory always holds current contents

- Simple, slow, wastes bandwidth

**Write-behind:** CPU writes are cached; writes to main memory may be buffered. CPU keeps executing while writes are completed in the background

- Faster, still uses lots of bandwidth


**Write-back:** CPU writes are cached, but not written to main memory until we replace the block. Memory contents can be “stale”

- Fastest, low bandwidth, more complex
- Commonly implemented in current systems


# Write-Back

ON REFERENCE TO Mem[X]: Look for X among tags...


HIT:  $TAG(X) == Tag[i]$  , for some cache block  $i$

- READ: return Data[i]
- WRITE: change Data[i]; ~~Start Write to Mem[X]~~ 

MISS:  $TAG(X)$  not found in tag of any cache block that X can map to

- REPLACEMENT SELECTION:
  - Select some line  $k$  to hold Mem[X]
  - Write Back: Write Data[k] to Mem[Address from Tag[k]] 

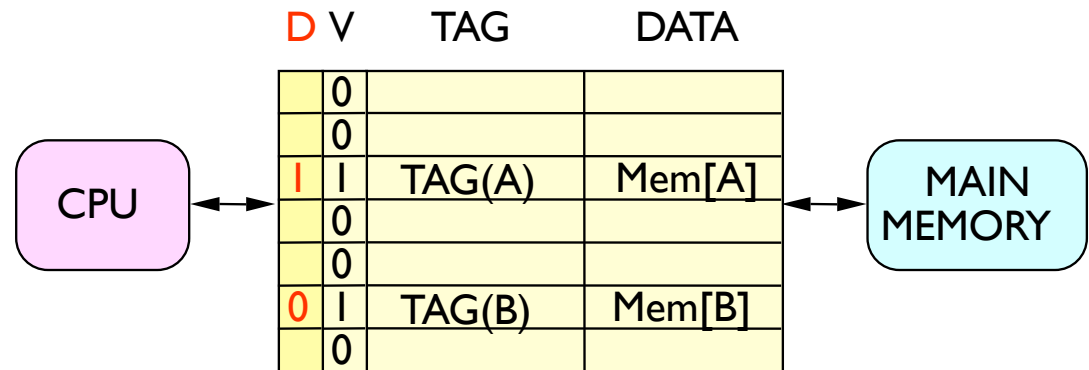
- READ: Read Mem[X]
  - Set  $Tag[k] = TAG(X)$ ,  $Data[k] = Mem[X]$

- WRITE: ~~Start Write to Mem[X]~~ 
  - Set  $Tag[k] = TAG(X)$ ,  $Data[k] = new Mem[X]$



# Write-Back with “Dirty” Bits

Add 1 bit per block to record whether block has been written to. Only write back dirty blocks.



ON REFERENCE TO Mem[X]: Look for TAG(X) among tags...

HIT:  $TAG(X) == Tag[i]$ , for some cache block  $i$

- READ: return Data[i]
- WRITE: change Data[i] ~~Start Write to Mem[X]~~  $D[i]=1$

MISS: TAG(X) not found in tag of any cache block that X can map to

- REPLACEMENT SELECTION:
  - Select some block  $k$  to hold Mem[X]
  - If  $D[k] == 1$  (Writeback) Write Data[k] to Mem[Address of Tag[k]]
- READ: Read Mem[X]; Set Tag[k] = TAG(X), Data[k] = Mem[X],  $D[k]=0$
- WRITE: ~~Start Write to Mem[X]~~  $D[k]=1$ 
  - Set Tag[k] = TAG(X), Data[k] = new Mem[X]

# Summary: Cache Tradeoffs

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

- Larger **cache size**: Lower miss rate, higher hit time
- Larger **block size**: Trade off spatial for temporal locality, higher miss penalty
- More **associativity** (ways): Lower miss rate, higher hit time
- More intelligent **replacement**: Lower miss rate, higher cost
- **Write policy**: Lower bandwidth, more complexity
- How to navigate all these dimensions? Simulate different cache organizations on real programs