Memory basic: address & values

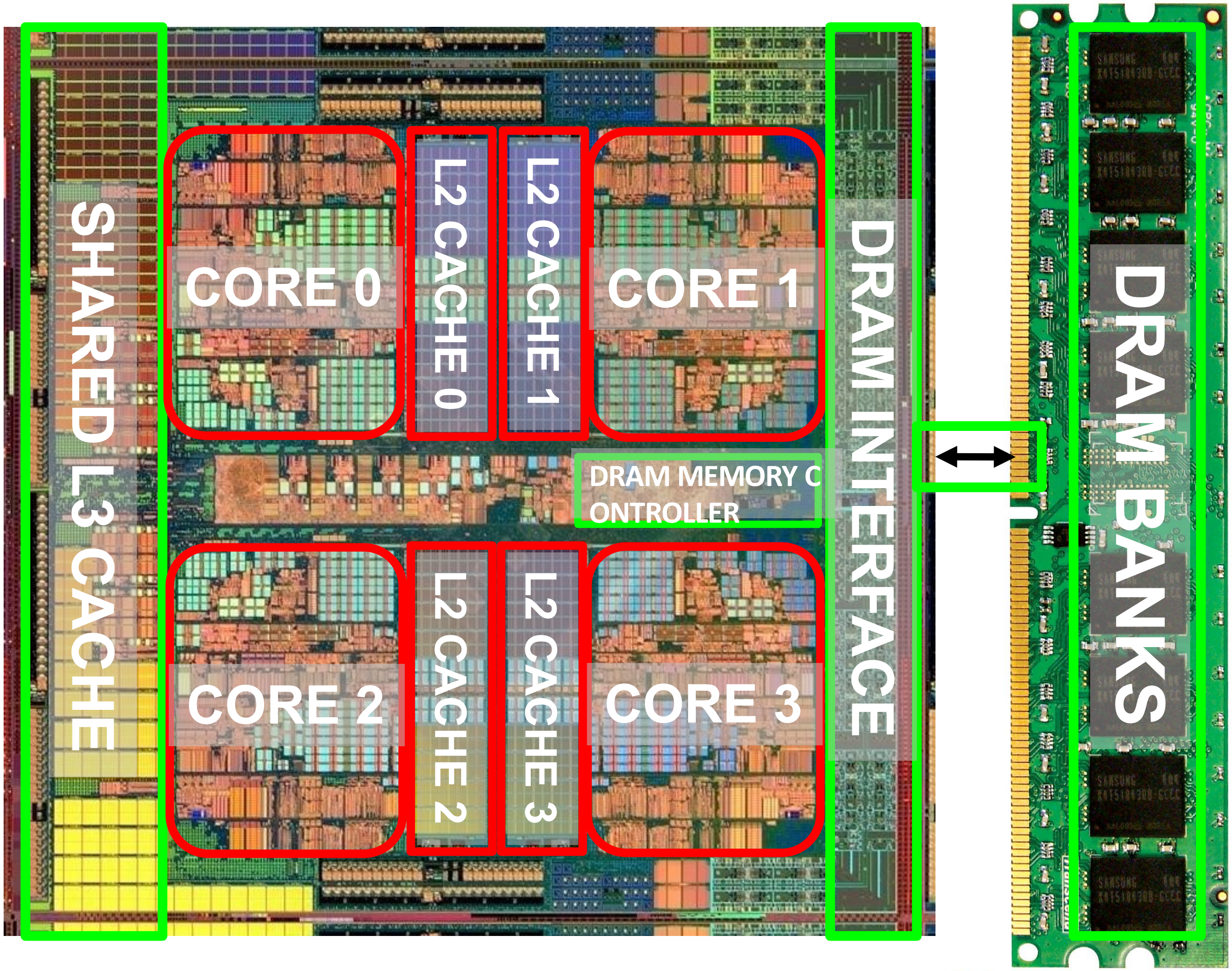
Seehwan Yoo

Dept. of Mobile Systems

Dankook University

DANKOOK UNIVERSITY

Memory in a Modern System



Primary storage device

- Persistency
  - Volatile / non-volatile
- Physical structure
  - Random access vs. sequential access
- Byte-address
  - Possibly bit-address word-address
  - Different byte, different address
  - 1byte == 8bits (0x00~0xFF), 2digits in hexadecimal number

Memory address and values

c. memory is filled with the following patterns

```
---
(from the address 0x8000:0000 to 0x80FF:FFFF)
0x0000 0x1111 0x2222 0x3333 0x4444 0x5555 0x6666 0x7777
0x7777 0x6666 0x5555 0x4444 0x3333 0x2222 0x1111 0x0000
(from the address 0x8100:0000 to 0x81FF:FFFF)
0x8888 0x9999 0xaaaa 0xbbbb 0xcccc 0xdddd 0xeeee 0xffff
0xffff 0xeeee 0xdddd 0xcccc 0xbbbb 0xaaaa 0x9999 0x8888
---
```

a. memory access pattern is as follows:

0x8000:0000, 0x8000:000C, 0x8000:0044, 0x8000:0000, 0x8100:0000, 0x8100:0040, 0x8000:0044, 0x8000:001C, 0x8100:0044, 0x8100:0000, 0x8100:0060, 0x8100:0040, 0x8000:0088, 0x8000:002C, 0x8100:0044, 0x8100:0000, 0x8100:0060, 0x8100:0040

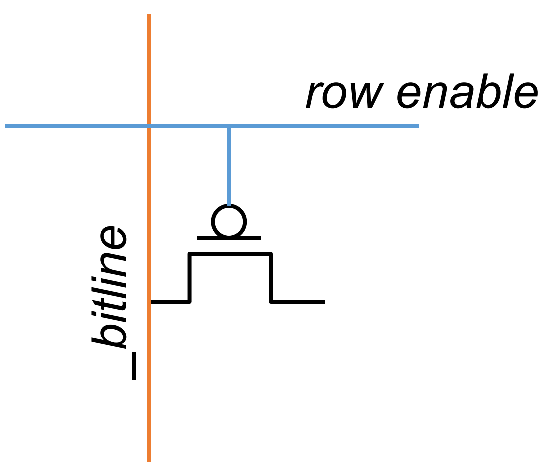
Q1. Write the accessed data (sequentially)

Ideal Memory

- Zero access time (latency)
- Infinite capacity
- Zero cost
- Infinite bandwidth (to support multiple accesses in p
arallel)

Memory Technology: DRAM

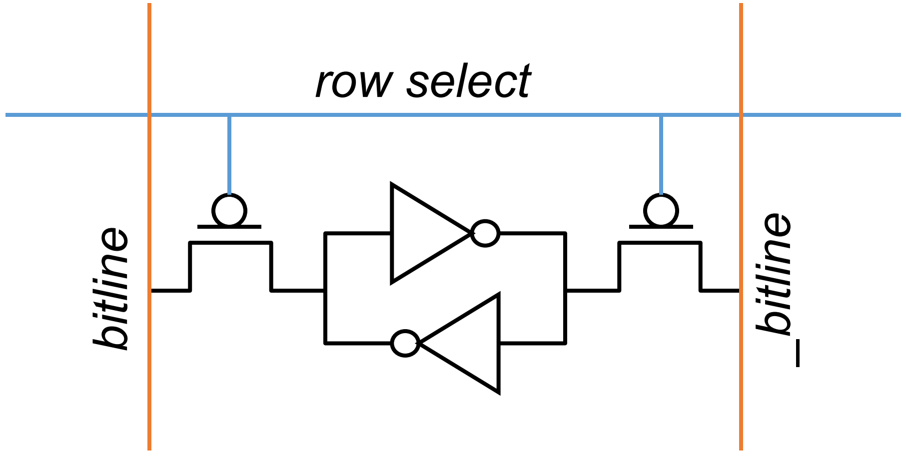
- Dynamic random access memory
- Capacitor charge state indicates stored value
  - Whether the capacitor is charged or discharged indicates storage of 1 or 0
  - 1 capacitor
  - 1 access transistor



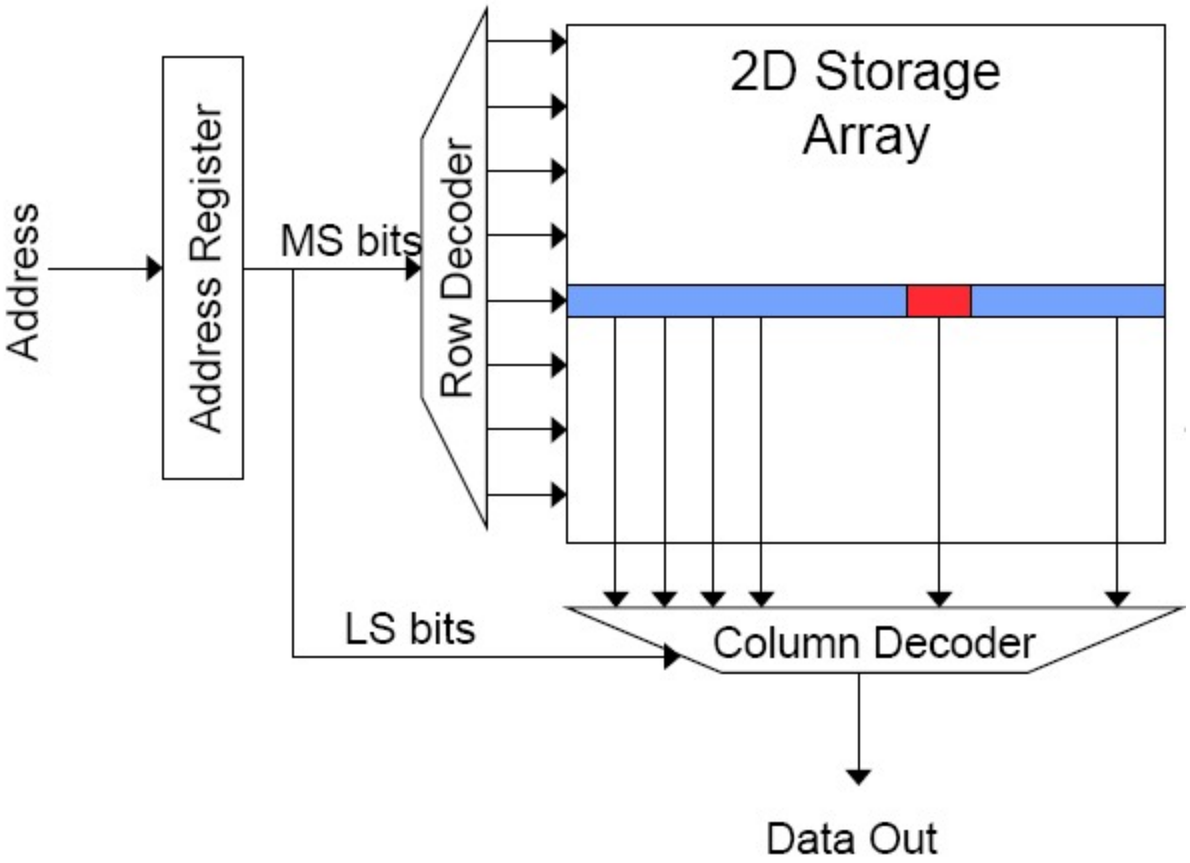
- Capacitor leaks through the RC path
  - DRAM cell loses charge over time
  - DRAM cell needs to be refreshed

Memory Technology: SRAM

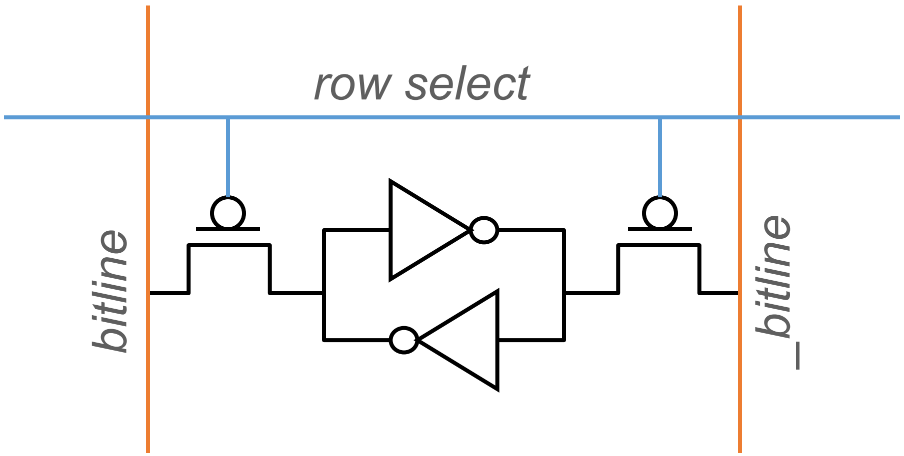
- Static random access memory
- Two cross coupled inverters store a single bit
  - Feedback path enables the stored value to persist in the " cell"
  - 4 transistors for storage
  - 2 transistors for access

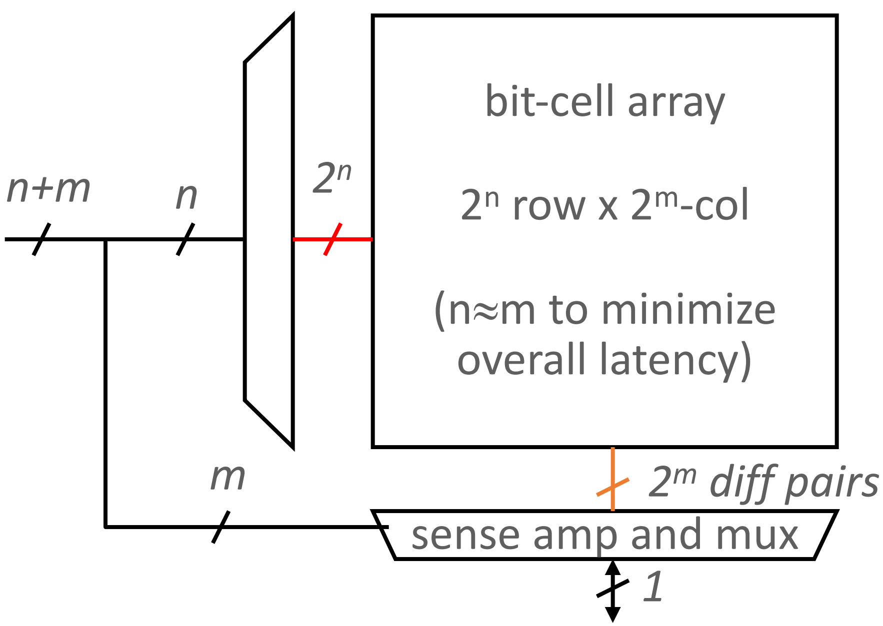


Memory Bank Organization and Operation



SRAM (Static Random Access Memory)





Read Sequence

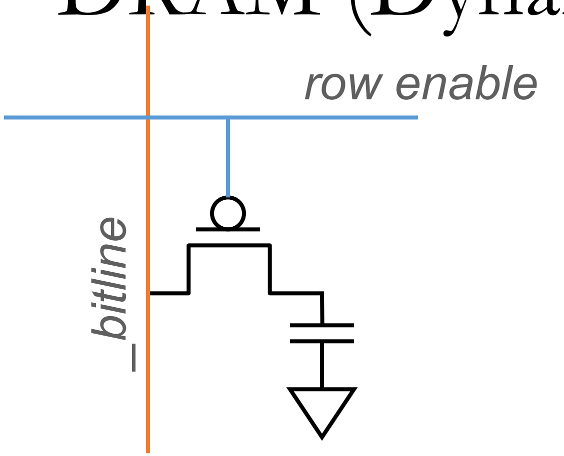
1. address decode
2. drive row select
3. selected bit-cells drive bitlines
   (entire row is read together)
4. differential sensing and column select
   (data is ready)
5. precharge all bitlines
   (for next read or write)

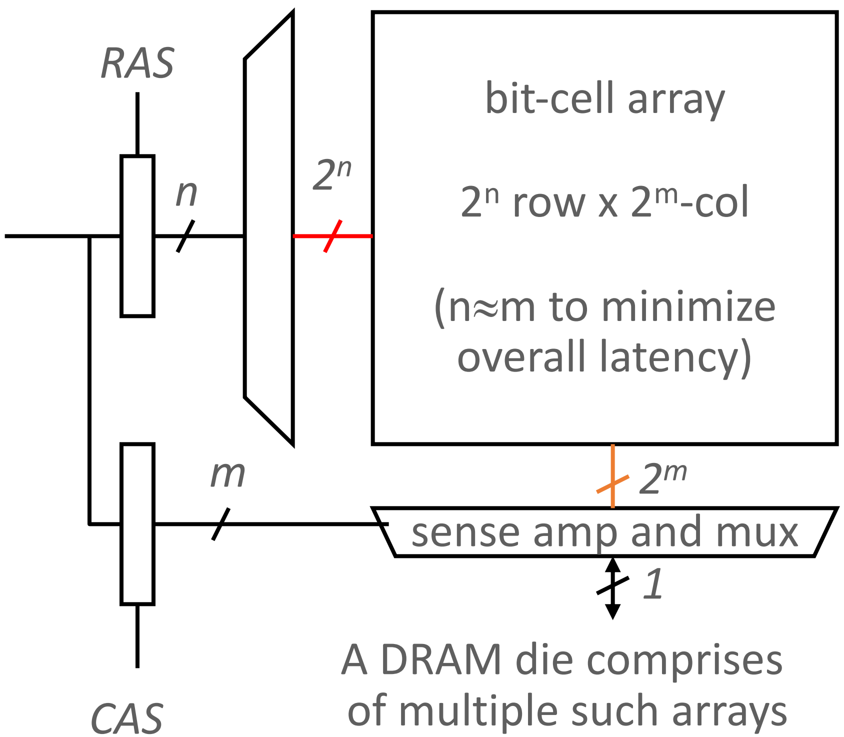
Access latency dominated by steps 2 and 3

Cycling time dominated by steps 2, 3 and 5

- step 2 proportional to 2^m
- step 3 and 5 proportional to 2^n

DRAM (Dynamic Random Access Memory)





A DRAM die comprises of multiple such arrays

Bits stored as charges on node capacitance (non-restorative)

- bit cell loses charge when read
- bit cell loses charge over time

Read Sequence

1~3 same as SRAM

4. a "flip-flopping" sense amp amplifies and regenerates the bitline, data bit is mux'ed out
5. precharge all bitlines

Destructive reads

Charge loss over time

Refresh: A DRAM controller must periodically read each row within the allowed refresh time (10s of ms) such that charge is restored

DRAM vs. SRAM

- DRAM
  - Slower access (capacitor)
  - Higher density (1T 1C cell)
  - Lower cost
  - Requires refresh (power, performance, circuitry)
  - Manufacturing requires putting capacitor and logic together

- SRAM
  - Faster access (no capacitor)
  - Lower density (6T cell)
  - Higher cost
  - No need for refresh
  - Manufacturing compatible with logic process (no capacitor)

The Problem

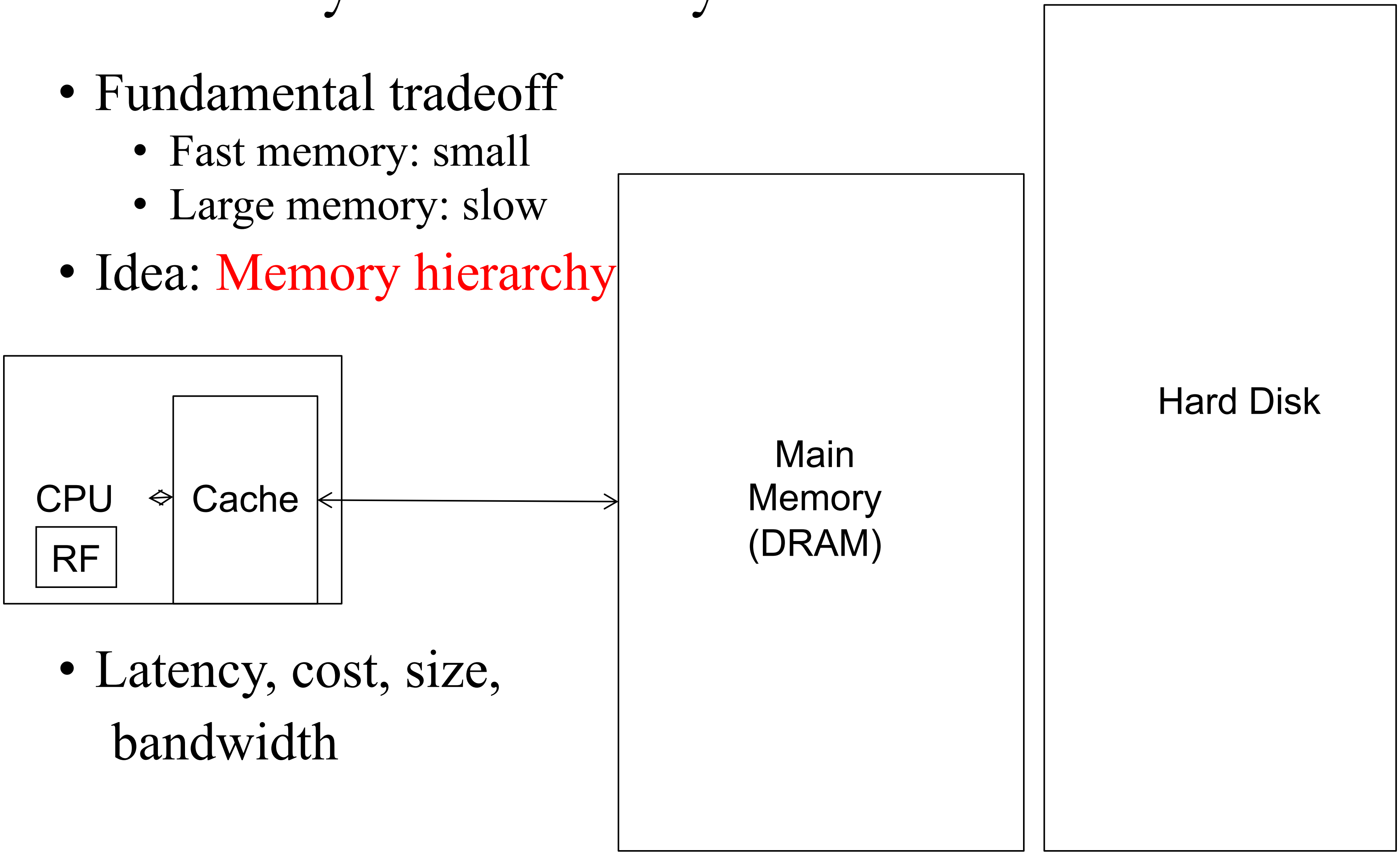
- Ideal memory's requirements oppose each other
- Bigger is slower
  - Bigger → Takes longer to determine the location
- Faster is more expensive
  - Memory technology: SRAM vs. DRAM
- Higher bandwidth is more expensive
  - Need more banks, more ports, higher frequency, or faster technology

Trade-off relationship in memory

- Bigger is slower
  - SRAM, 512 Bytes, sub-nanosec
  - SRAM, KByte~MByte, ~nanosec
  - DRAM, Gigabyte, ~50 nanosec
  - Hard Disk, Terabyte, ~10 millisec
- Faster is more expensive (dollars and chip area)
  - SRAM, < 10$ per Megabyte
  - DRAM, < 1$ per Megabyte
  - Hard Disk < 1$ per Gigabyte
  - These sample values scale with time
- Other technologies have their place as well
  - Flash memory, Phase-change memory (not mature yet)

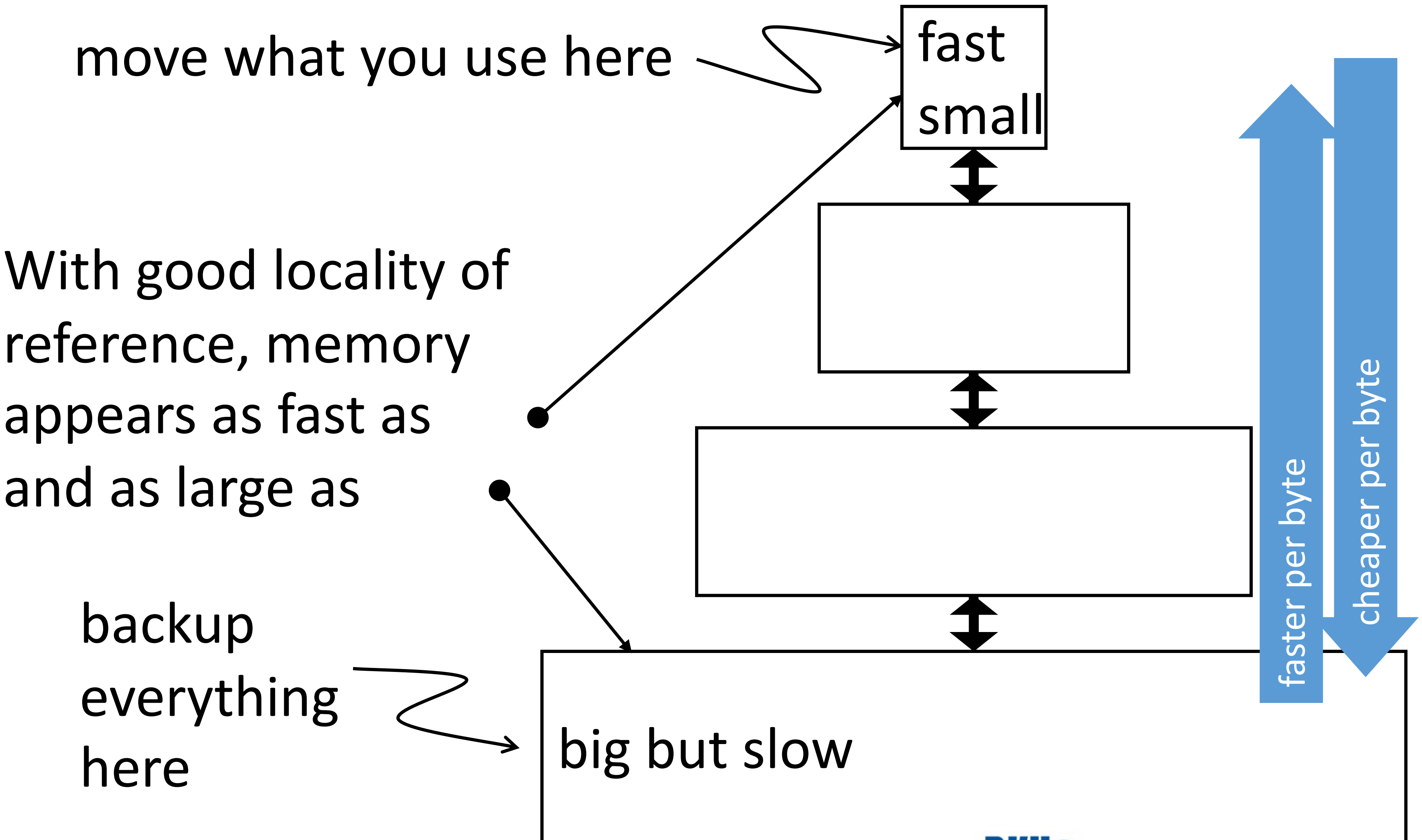
Memory Hierarchy

- Fundamental tradeoff
  - Fast memory: small
  - Large memory: slow
- Idea: Memory hierarchy



- Latency, cost, size, bandwidth

The Memory Hierarchy



Why Memory Hierarchy?

- We want both fast and large

- But we cannot achieve both with a single level of memory

- Idea: Have multiple levels of storage (progressively bigger and slower as the levels are farther from the processor) and ensure most of the data the processor needs is kept in the fast(er) level(s)

Locality

- One's recent past is a very good predictor of his/her near future.
- Temporal Locality: If you just did something, it is v ery likely that you will do the same thing again soon
  - since you are here today, there is a good chance you will be here again and again regularly
- Spatial Locality: If you did something, it is very lik ely you will do something similar/related (in space)
  - every time I find you in this room, you are probably sitting close to the same people

Memory Locality

- A "typical" program has a lot of locality in memory references
  - typical programs are composed of "loops"
- Temporal: A program tends to reference the same memory location many times and all within a small window of time
- Spatial: A program tends to reference a cluster of memory locations at a time
  - most notable examples:
    - 1. instruction memory references
    - 2. array/data structure references

Caching Basics: Exploit Temporal Locality

- Idea: Store recently accessed data in automatically mana ged fast memory (called cache)
- Anticipation: the data will be accessed again soon
- Temporal locality principle
  - Recently accessed data will be again accessed in the near futur e
  - This is what Maurice Wilkes had in mind:
    - Wilkes, "Slave Memories and Dynamic Storage Allocation," IEEE Trans. On Electronic Computers, 1965.
    - "The use is discussed of a fast core memory of, say 32000 words as a slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory."

Caching Basics: Exploit Spatial Locality

- Idea: Store addresses adjacent to the recently accessed one in automatically managed fast memory
  - Logically divide memory into equal size blocks
  - Fetch to cache the accessed block in its entirety
- Anticipation: nearby data will be accessed soon

- Spatial locality principle
  - Nearby data in memory will be accessed in the near future
    - E.g., sequential instruction access, array traversal
  - This is what IBM 360/85 implemented
    - 16 Kbyte cache with 64 byte blocks
    - Liptay, "Structural aspects of the System/360 Model 85 II: the cache," IBM Systems Journal, 1968.

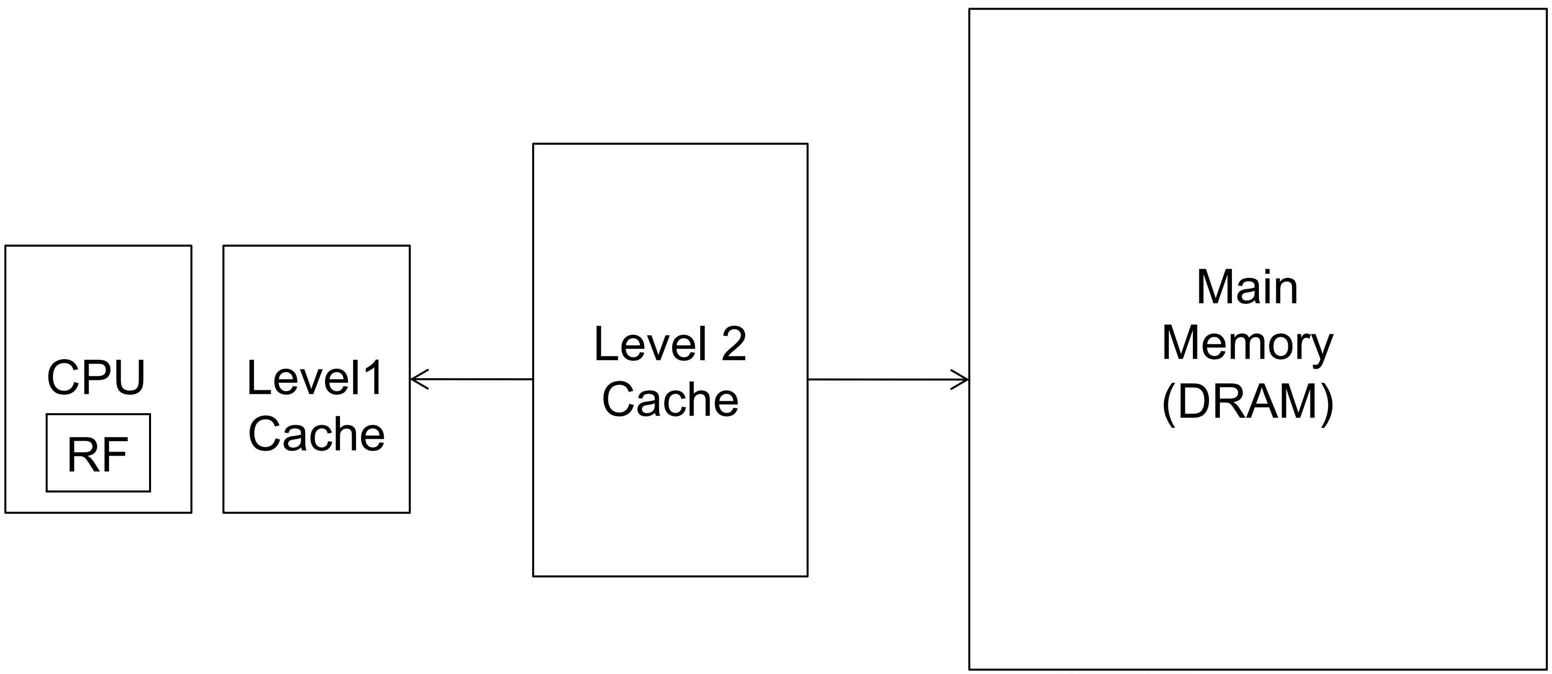
The Bookshelf Analogy

- Book in your hand
- Desk
- Bookshelf
- Boxes at home
- Boxes in storage

- Recently-used books tend to stay on desk
  - Comp Arch books, books for classes you are currently taking
  - Until the desk gets full
- Adjacent books in the shelf needed around the same time
  - If I have organized/categorized my books well in the shelf

Caching in a Pipelined Design

- The cache needs to be tightly integrated into the pipeline
  - Ideally, access in 1-cycle so that dependent operations do not stall
- High frequency pipeline → Cannot make the cache large
  - But, we want a large cache AND a pipelined design
- Idea: Cache hierarchy



A Note on Manual vs. Automatic Management

- Manual: Programmer manages data movement across le vels
  -- too painful for programmers on substantial programs
  - "core" vs "drum" memory in the 50's
  - still done in some embedded processors (on-chip scratch pad SRAM in lieu of a cache)

- Automatic: Hardware manages data movement across le vels, transparently to the programmer
  ++ programmer's life is easier
  - simple heuristic: keep most recently used items in cache
  - the average programmer doesn't need to know about it
    - You don't need to know how big the cache is and how it works to w rite a "correct" program! (What if you want a "fast" program?)

Automatic Management in Memory Hierarchy

- Wilkes, "Slave Memories and Dynamic Storage Allocation," IEEE Trans. On Electronic Computers, 1965.

Slave Memories and Dynamic Storage Allocation

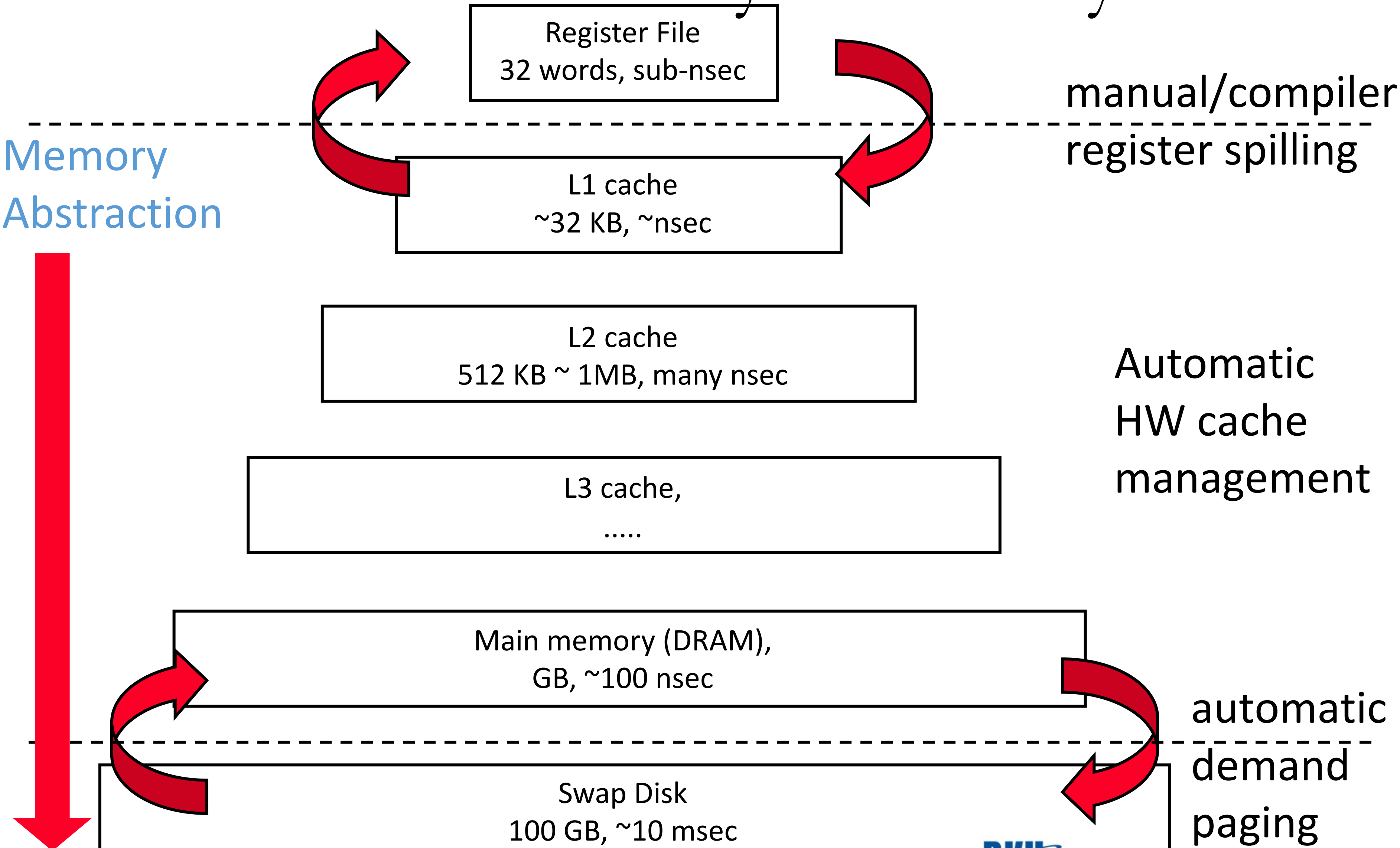
M. V. WILKES

SUMMARY

The use is discussed of a fast core memory of, say, 32 000 words as a slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory.

- "By a slave memory I mean one which automatically accumulates to itself words that come from a slower main memory, and keeps them available for subsequent use without it being necessary for the penalty of main memory access to be incurred again."

A Modern Memory Hierarchy



Hierarchical Latency Analysis

- For a given memory hierarchy level i it has a technology-intrinsic access time of $t_{i,}$ The perceived access time T_i is longer than t_i
- Except for the outer-most hierarchy, when looking for a given add ress there is
  - a chance (hit-rate h_i) you "hit" and access time is t_i
  - a chance (miss-rate m_i) you "miss" and access time $t_i + T_{i+1}$
  - $h_i + m_i = 1$
- Thus

$$T_i = h_i \cdot t_i + m_i \cdot (t_i + T_{i+1})$$

$$T_i = t_i + m_i \cdot T_{i+1}$$

keep in mind, h_i and m_i are defined to be the hit-rate and miss-rate of just the references that missed at L_{i-1}

Hierarchy Design Considerations

- Recursive latency equation

$$T_i = t_i + m_i \cdot T_{i+1}$$

- The goal: achieve desired T_1 within allowed cost
- $T_i \approx t_i$ is desirable

- Keep m_i low
  - increasing capacity C_i lowers m_i, but beware of increasing t_i
  - lower m_i by smarter management (replacement::anticipate what you don't need, prefetching::anticipate what you will need)

- Keep T_{i+1} low
  - faster lower hierarchies, but beware of increasing cost
  - introduce intermediate hierarchies as a compromise

Intel Pentium 4 Example

- 90nm P4, 3.6 GHz
- L1 D-cache
  - C_1 = 16K
  - t_1 = 4 cyc int / 9 cycle fp
- L2 D-cache
  - C_2 =1024 KB
  - t_2 = 18 cyc int / 18 cyc fp
- Main memory
  - t_3 = ~ 50ns or 180 cyc
- Notice
  - best case latency is not 1
  - worst case access latencies are into 500+ cycles

if m_1=0.1, m_2=0.1
T_1=7.6, T_2=36

if m_1=0.01, m_2=0.01
T_1=4.2, T_2=19.8

if m_1=0.05, m_2=0.01
T_1=5.00, T_2=19.8

if m_1=0.01, m_2=0.50
T_1=5.08, T_2=108

Cache

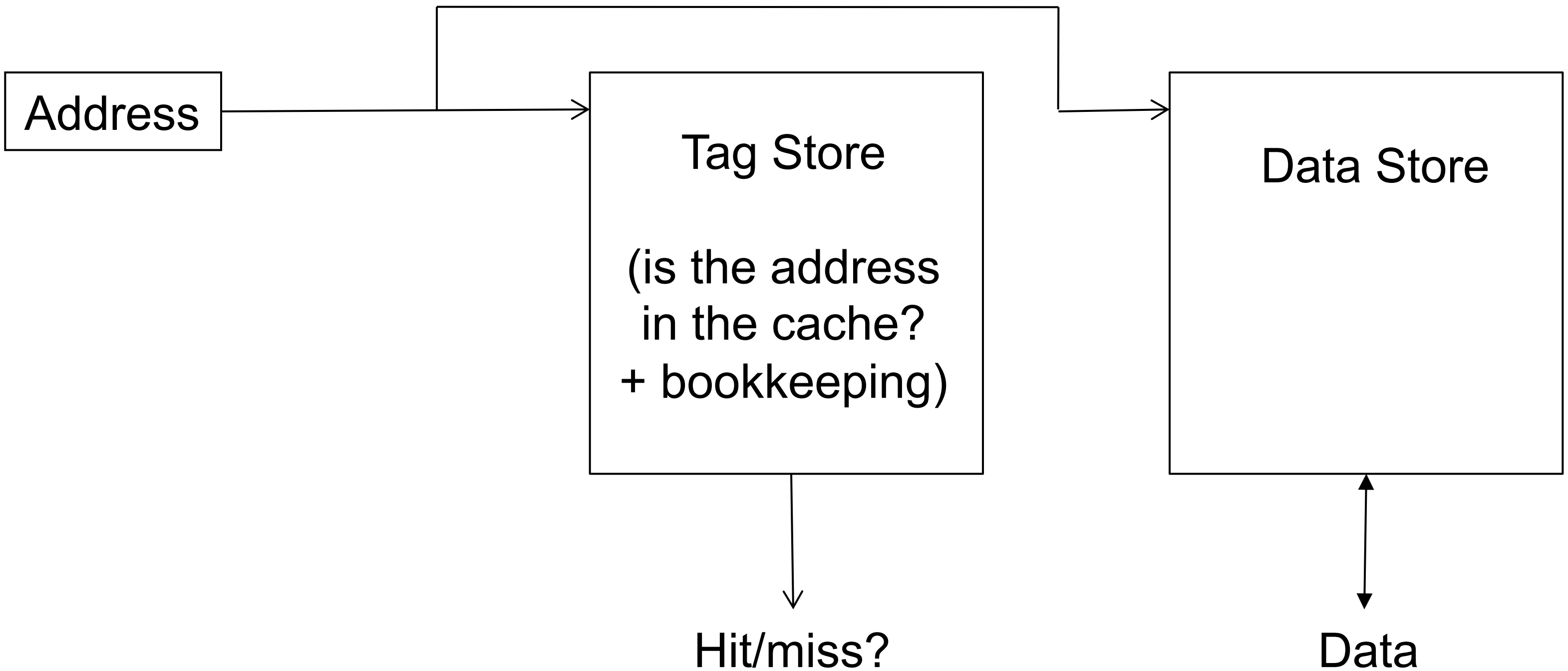
- Generically, any structure that “memorizes” frequently used results to avoid repeating the long-latency operations required to reproduce the results from scratch, e.g. a web cache

- Most commonly, an automatically-managed memory hierarchy based on SRAM
  - memorize in SRAM the most frequently accessed DRAM memory locations to avoid repeatedly paying for the DRAM access latency

Caching Basics

- Block (line): Unit of storage in the cache
  - Memory is logically divided into cache blocks that map to locations in the cache
- When data referenced
  - HIT: If in cache, use cached data instead of accessing memory
  - MISS: If not in cache, bring block into cache
    - Maybe have to kick something else out to do it
- Some important cache design decisions
  - Placement: where and how to place/find a block in cache?
  - Replacement: what data to remove to make room in cache?
  - Granularity of management: large, small, uniform blocks?
  - Write policy: what do we do about writes?
  - Instructions/data: Do we treat them separately?

Cache Abstraction and Metrics



- Cache hit rate = (# hits) / (# hits + # misses) = (# hits) / (# accesses)
- Average memory access time (AMAT)

  = (hit-rate * hit-latency) + (miss-rate * miss-latency)
- Aside: *Can reducing AMAT reduce performance?*

Blocks and Addressing the Cache

- Memory is logically divided into cache blocks

- Each block maps to a location in the cache, determined by the index bits in the address
  - used to index into the tag and data stores

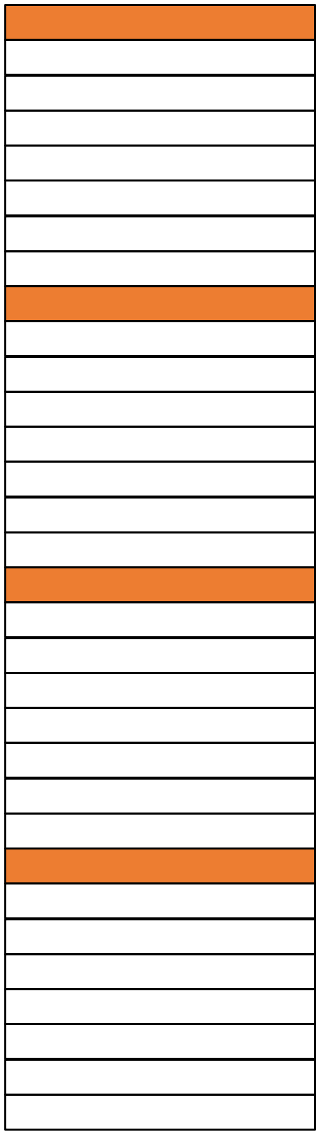
tag	index	byte in block
2b	3 bits	3 bits

8-bit address

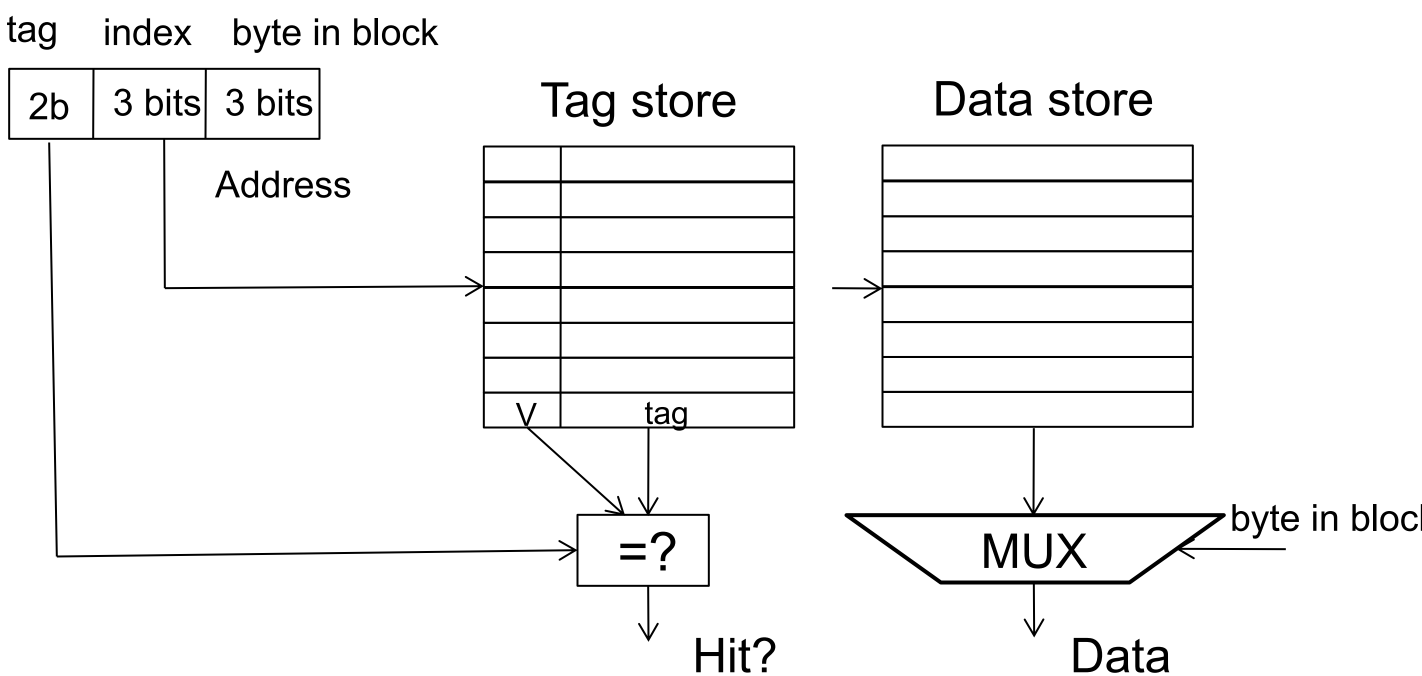
- Cache access: index into the tag and data stores with ind ex bits in address, check valid bit in tag store, compare t ag bits in address with the stored tag in tag store

- If a block is in the cache (cache hit), the tag store should have the tag of the block stored in the index of the block

Direct-Mapped Cache: Placement and Access



- Assume byte-addressable memory:
  256 bytes, 8-byte blocks → 32 blocks
- Assume cache: 64 bytes, 8 blocks
  - Direct-mapped: A block can go to only one location



  - Addresses with same index contend for the same location
    - Cause conflict misses

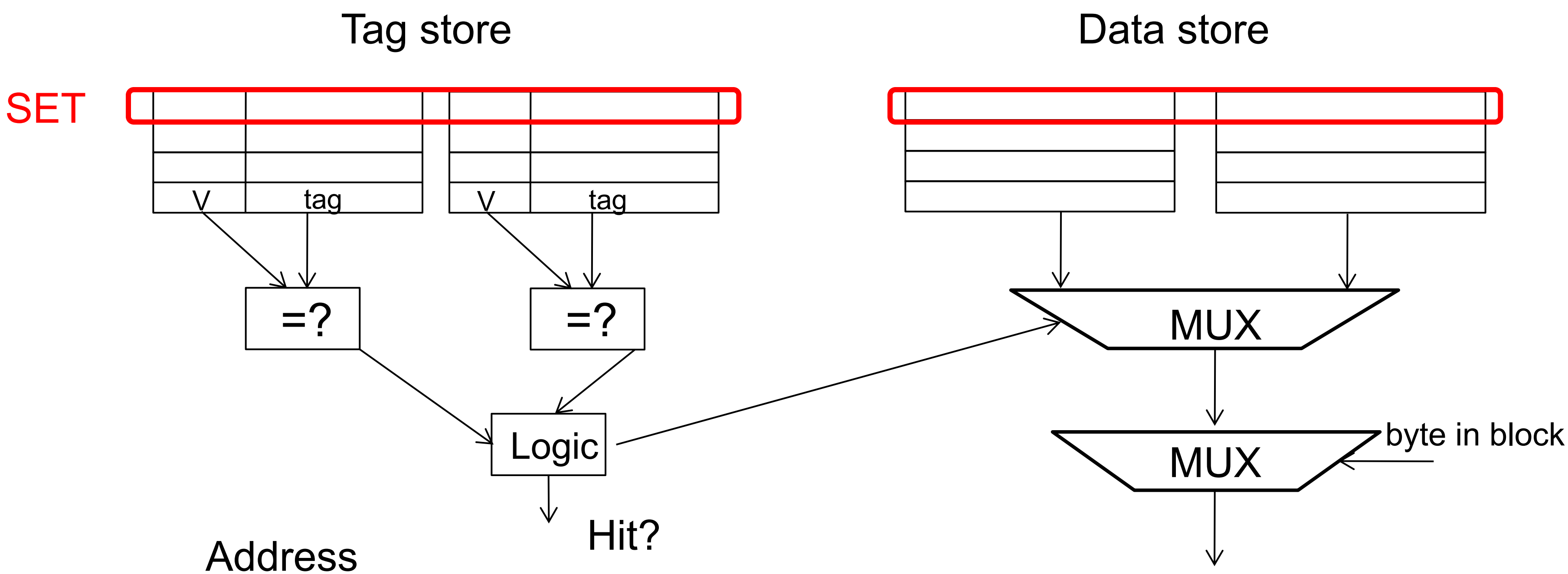
Direct-Mapped Caches

- Direct-mapped cache: Two blocks in memory that map to the same index in the cache cannot be presen t in the cache at the same time
  - One index → one entry

- Can lead to 0% hit rate if more than one block acces sed in an interleaved manner map to the same index
  - Assume addresses A and B have the same index bits but different tag bits
  - A, B, A, B, A, B, A, B, … → conflict in the cache index
  - All accesses are conflict misses

Set Associativity

- Addresses 0 and 8 always conflict in direct mapped cache
- Instead of having one column of 8, have 2 columns of 4 blocks



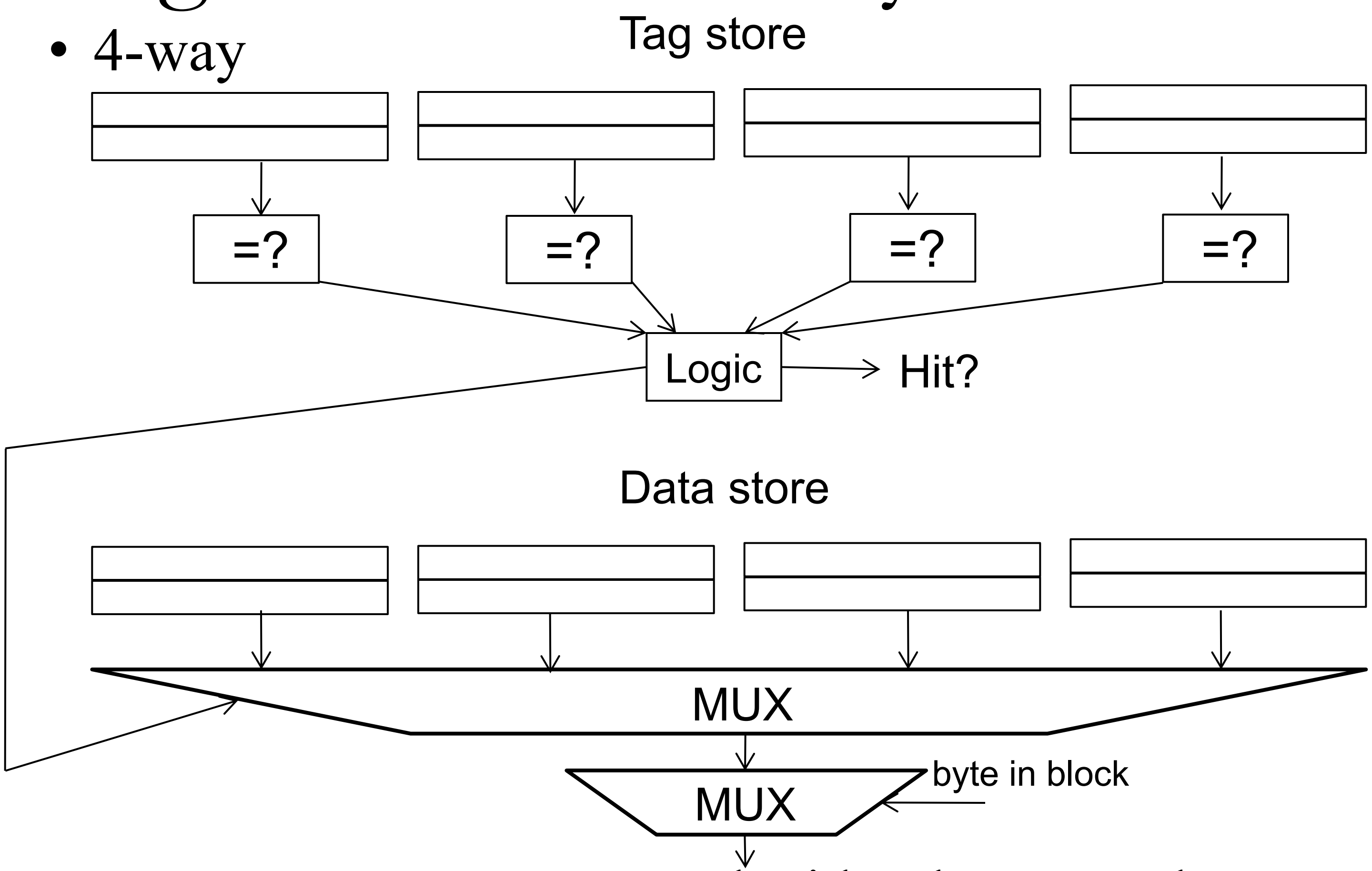
Address

tag	index	byte in block
3b	2 bits	3 bits

Associative memory within the set
-- More complex, slower access, larger tag store
+ Accommodates conflicts better (fewer conflict misses)

Higher Associativity

- 4-way

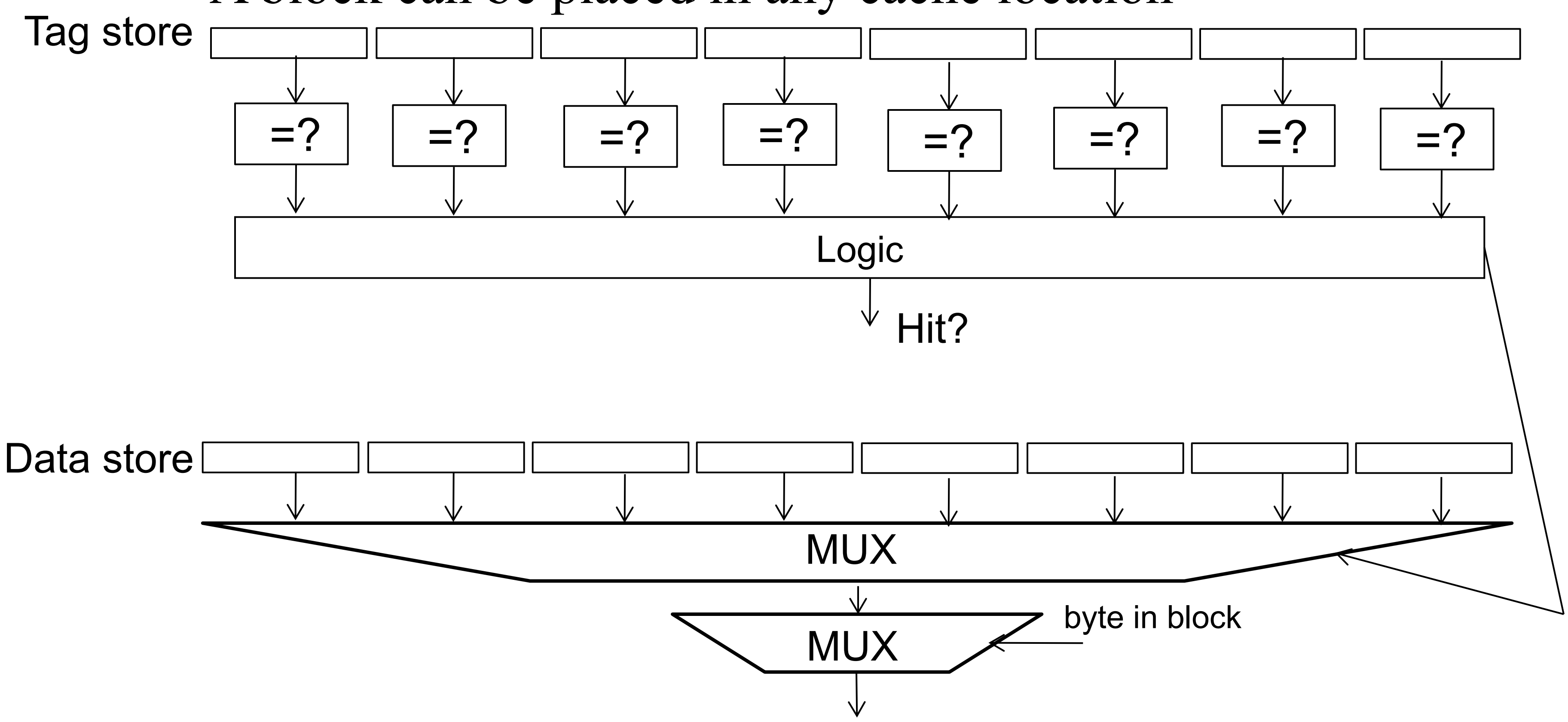


-- More tag comparators and wider data mux; larger tags

+ Likelihood of conflict misses even lower

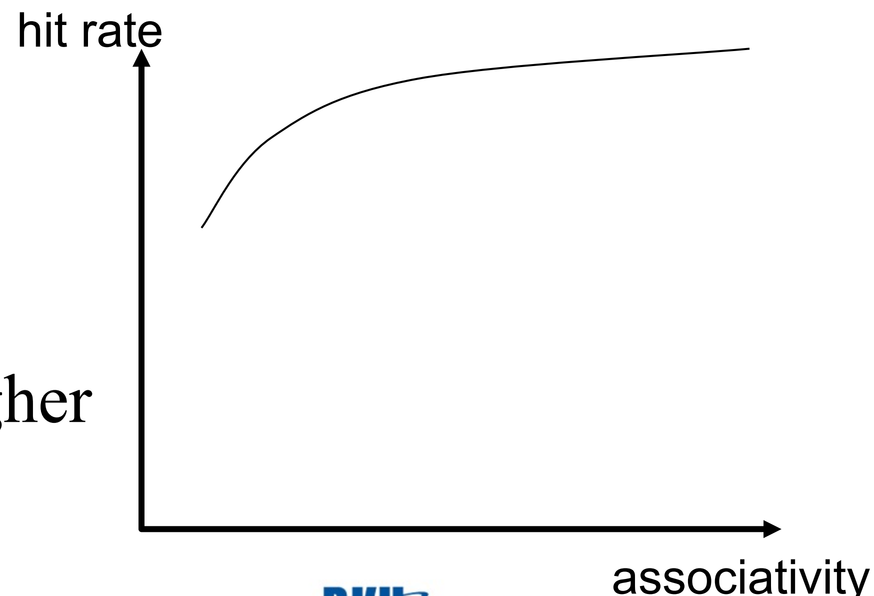
Full Associativity

- Fully associative cache
  - A block can be placed in any cache location



Associativity (and Tradeoffs)

- How many blocks can map to the same index (or set)?
- Higher associativity
  ++ Higher hit rate
  -- Slower cache access time (hit latency and data access latency)
  -- More expensive hardware (more comparators)



- Diminishing returns from higher associativity

Set-Associative Caches (I)

- Diminishing returns in hit rate from higher associativity
- Longer access time with higher associativity
- Which block in the set to replace on a cache miss?
  - Any invalid block first
  - If all are valid, consult the replacement policy
    - Random
    - FIFO
    - Least recently used (how to implement?)
    - Not most recently used
    - Least frequently used?
    - Least costly to re-fetch?
      - Why would memory accesses have different cost?
    - Hybrid replacement policies
    - Optimal replacement policy?