Computer Architecture

Lec 7: Caches

Overview of Memory Arrays

How Can We Store Data?

- Flip-Flops (or Latches)
  - Very fast, parallel access
  - Very expensive (one bit costs tens of transistors)

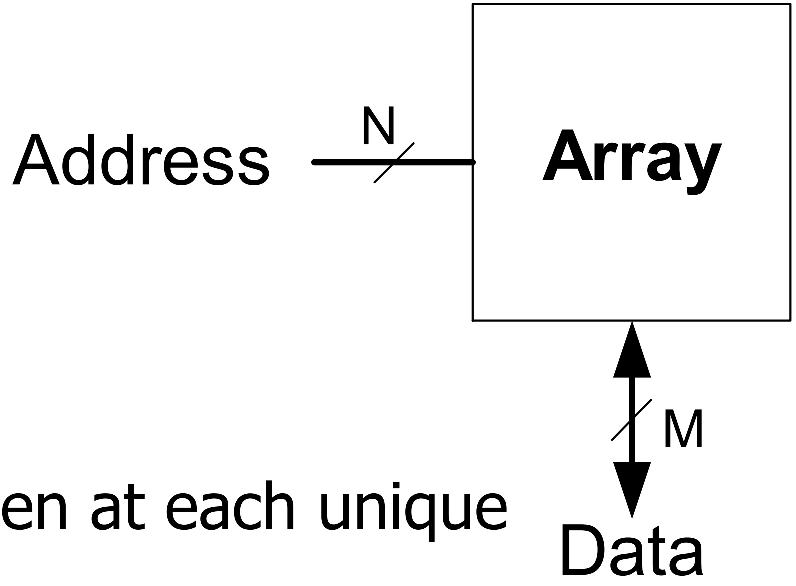
- Static RAM (we will describe them in a moment)
  - Relatively fast, only one data word at a time
  - Expensive (one bit costs 6 transistors)

- Dynamic RAM (we will describe them a bit later)
  - Slower, one data word at a time, reading destroys content (refresh), needs special process for manufacturing
  - Cheap (one bit costs only one transistor plus one capacitor)

- Other storage technology (flash memory, hard disk, tape)
  - Much slower, access takes a long time, non-volatile
  - Very cheap (no transistors directly involved)

Array Organization of Memories

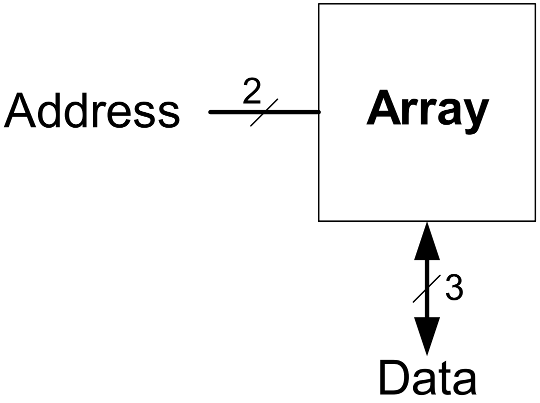
- Goal: Efficiently store large amounts of data
  - A memory array (stores data)
  - Address selection logic (selects one row of the array)
  - Readout circuitry (reads data out)

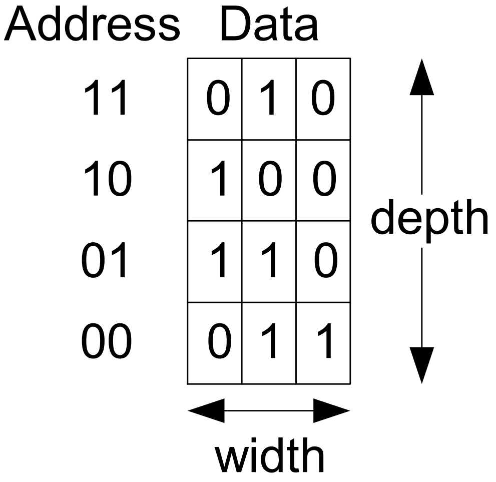


- An M-bit value can be read or written at each unique N-bit address
  - All values can be accessed, but only M-bits at a time
  - Access restriction allows more compact organization

Memory Array Example

- 2^2 × 3-bit array
- Number of words: 4
- Word size: 3-bits
- For example, the 3-bit word stored at address 10 is 100

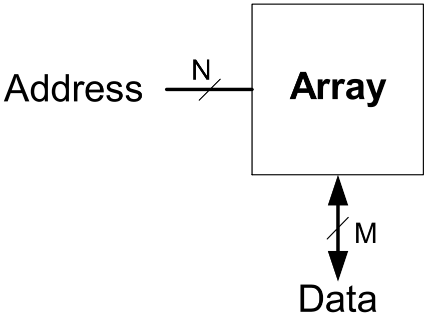


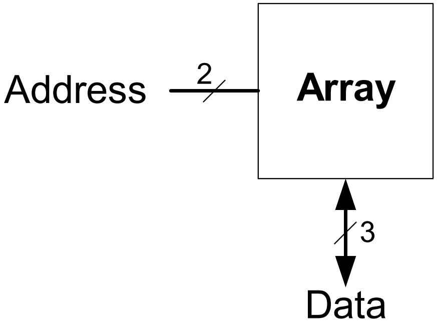


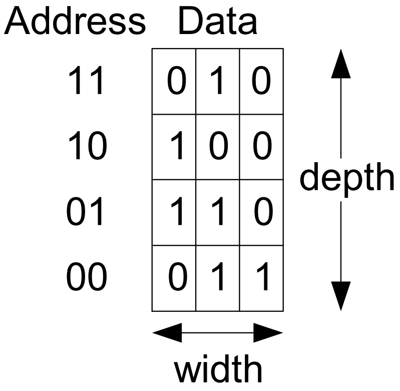
Address	Data		
11	0	1	0
10	1	0	0
01	1	1	0
00	0	1	1

Memory Arrays

- Two-dimensional array of bit cells
  - Each bit cell stores one bit
- An array with N address bits and M data bits:
  - 2^N rows and M columns
  - Depth: number of rows (number of words)
  - Width: number of columns (size of word)
  - Array size: depth × width = $2^N \times M$

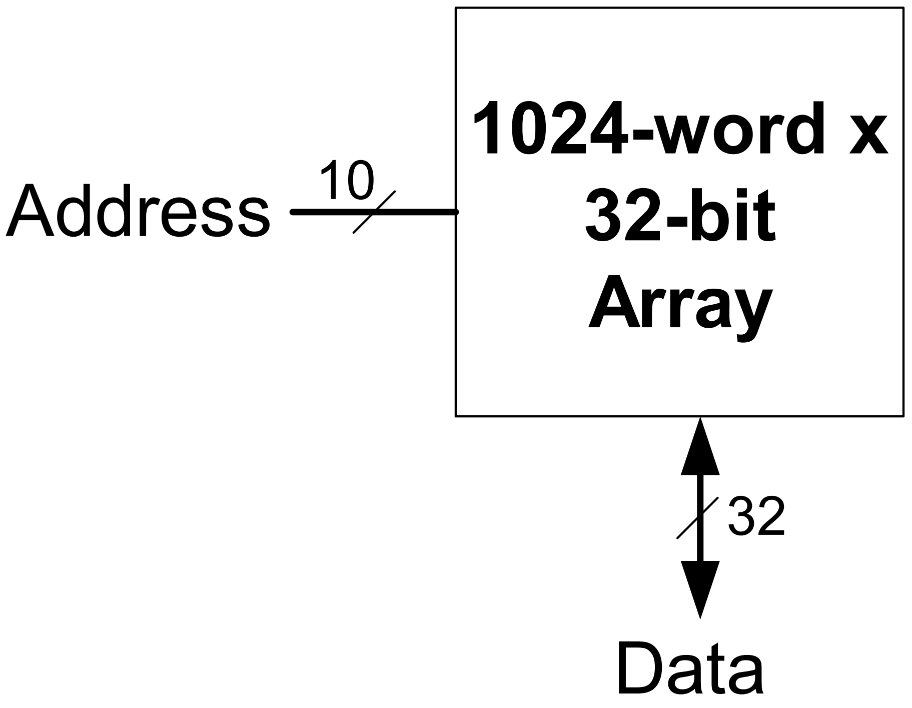






Address	Data		
11	0	1	0
10	1	0	0
01	1	1	0
00	0	1	1

Larger and Wider Memory Array Example



Address

10

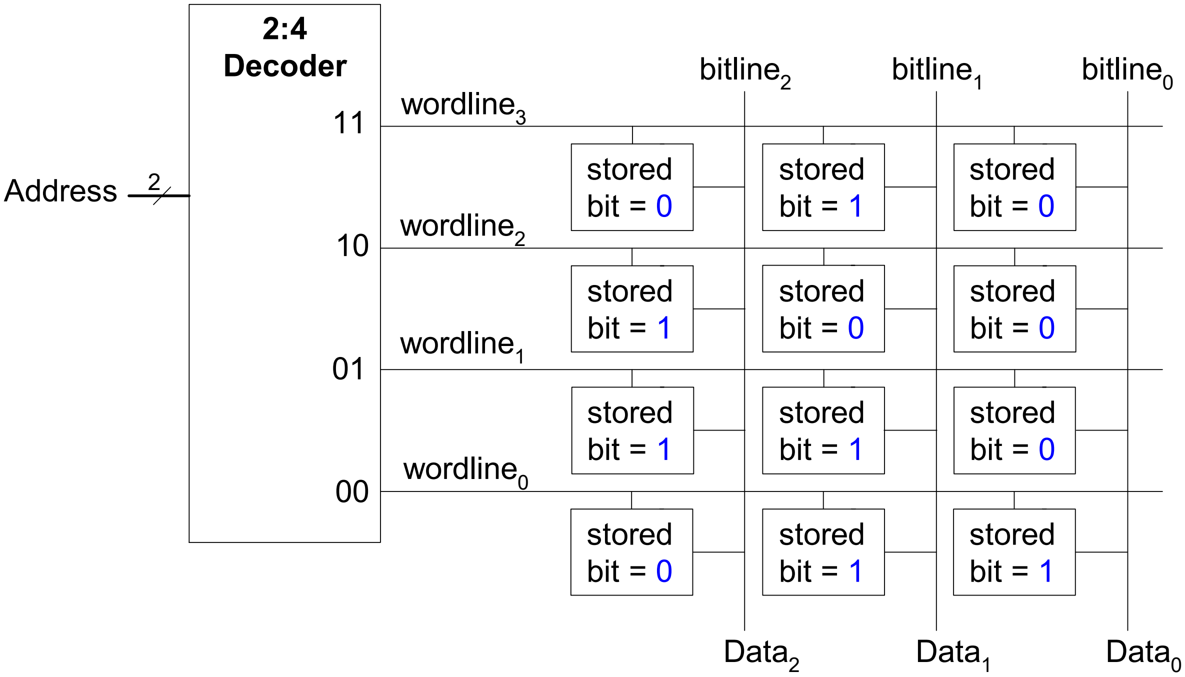
1024-word x 32-bit Array

32

Data

Memory Array Organization (I)

- Storage nodes in one column connected to one bitline
- Address decoder activates only ONE wordline
- Content of one line of storage available at output



2:4 Decoder

Address 2

11 wordline$_3$

10 wordline$_2$

01 wordline$_1$

00 wordline$_0$

bitline$_2$ bitline$_1$ bitline$_0$

stored bit = 0 stored bit = 1 stored bit = 0

stored bit = 1 stored bit = 0 stored bit = 0

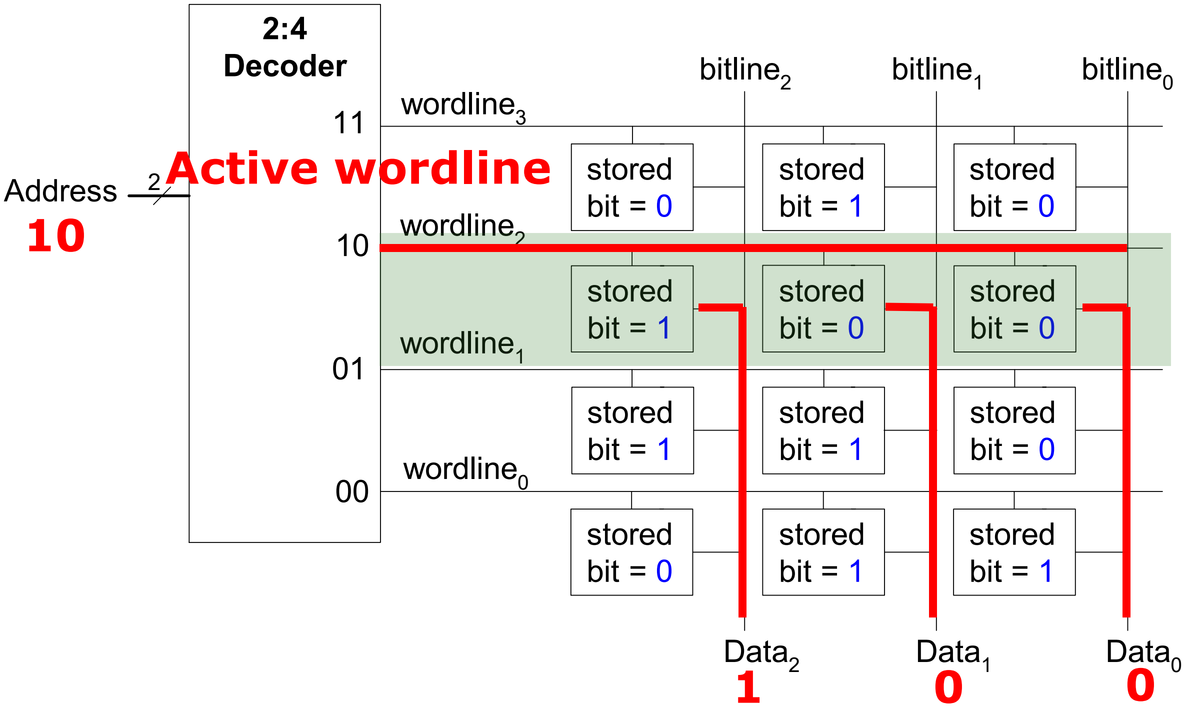
stored bit = 1 stored bit = 1 stored bit = 0

stored bit = 0 stored bit = 1 stored bit = 1

Data$_2$ Data$_1$ Data$_0$

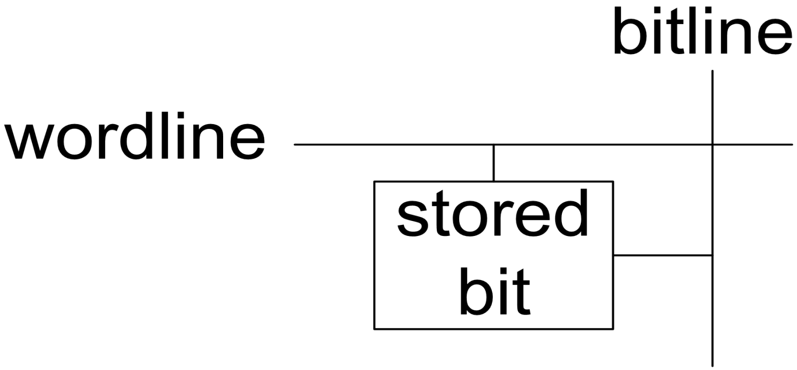
Memory Array Organization (II)

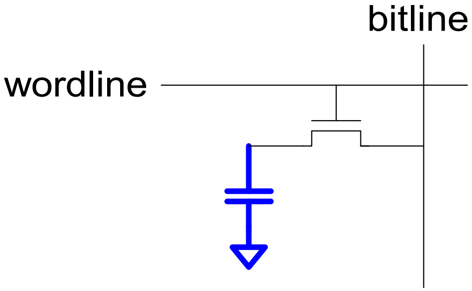
- Storage nodes in one column connected to one bitline
- Address decoder activates only ONE wordline
- Content of one line of storage available at output



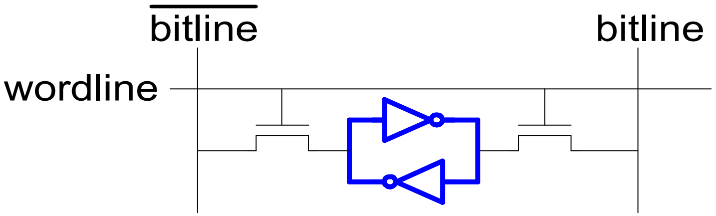
How is Access Controlled?

- Access transistors configured as switches connect the bit storage to the bitline.
- Access controlled by the wordline





DRAM



SRAM

Building Larger Memories

- Requires larger memory arrays

- Large → slow

- How do we make the memory large without making it very slow?

- Idea: Divide the memory into smaller arrays and interconnect the arrays to input/output buses
  - Large memories are hierarchical array structures
  - DRAM: Channel → Rank → Bank → Subarrays → Mats

General Principle: Interleaving (Banking)

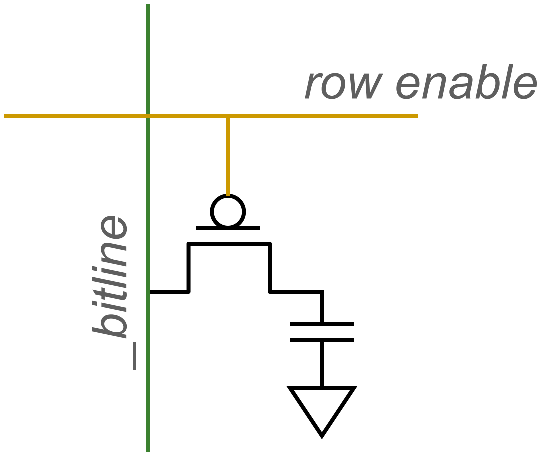
- Interleaving (banking)
  - Problem: a single monolithic large memory array takes long to access and does not enable multiple accesses in parallel
  - Goal: Reduce the latency of memory array access and enable multiple accesses in parallel
  - Idea: Divide a large array into multiple banks that can be accessed independently (in the same cycle or in consecutive cycles)
    - Each bank is smaller than the entire memory storage
    - Accesses to different banks can be overlapped
  - A Key Issue: How do you map data to different banks? (i.e., how do you interleave data across banks?)

Memory Technology: DRAM and SRAM

Memory Technology: DRAM

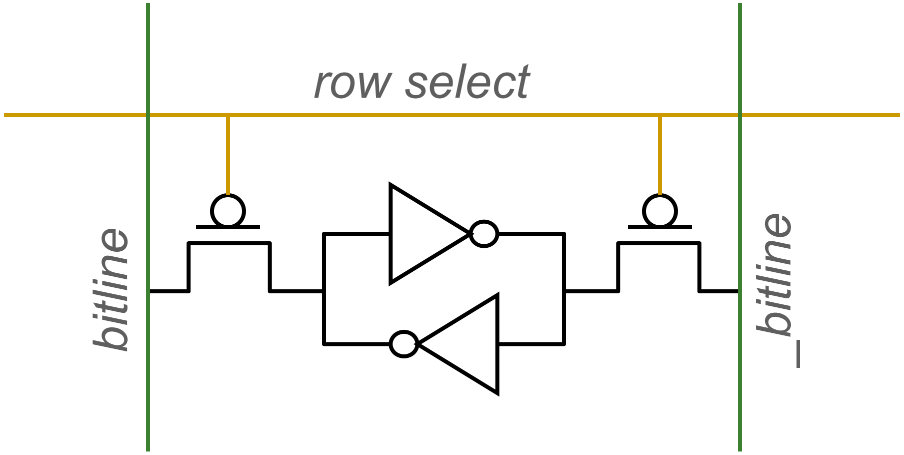
- Dynamic random access memory
- Capacitor charge state indicates stored value
  - Whether the capacitor is charged or discharged indicates storage of 1 or 0
  - 1 capacitor
  - 1 access transistor

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

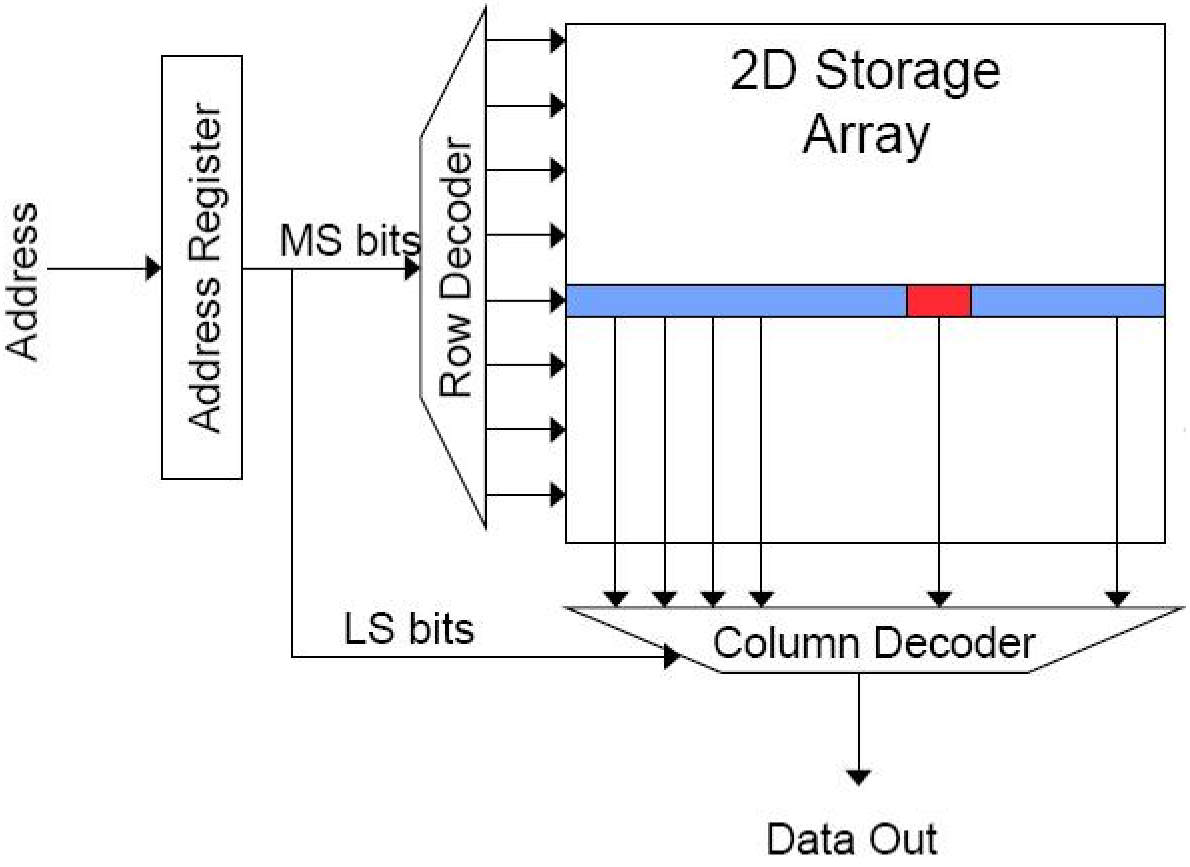


Memory Technology: SRAM

- 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



- Read access sequence:

1. Decode row address & drive word-lines

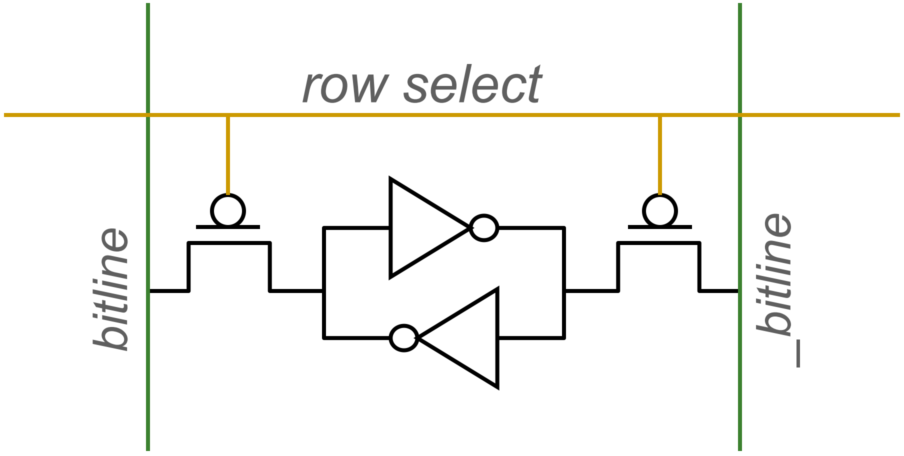
2. Selected bits drive bit-lines
   - Entire row read

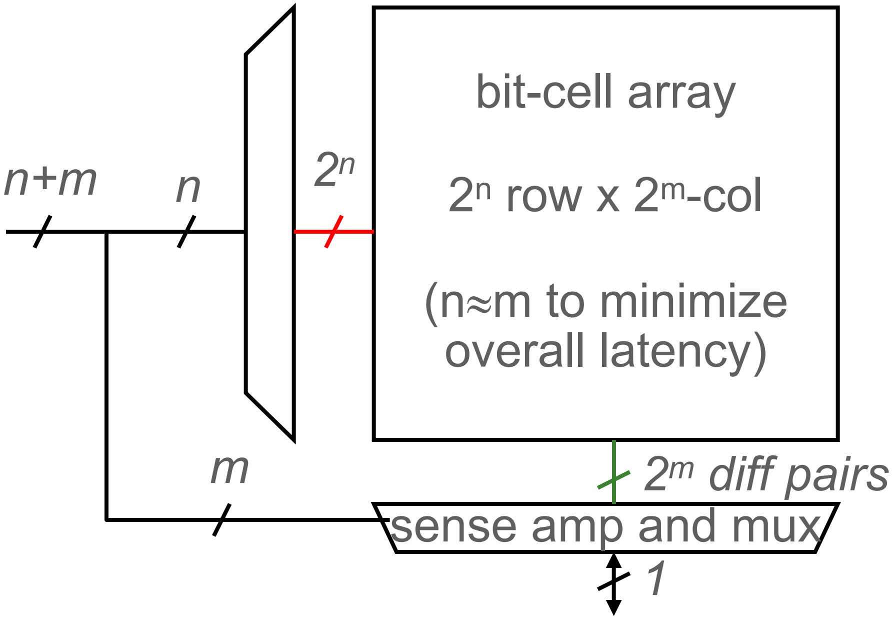
3. Amplify row data

4. Decode column address & select subset of row
   - Send to output

5. Precharge bit-lines
   - For next access

SRAM (Static Random Access Memory)





Read Sequence

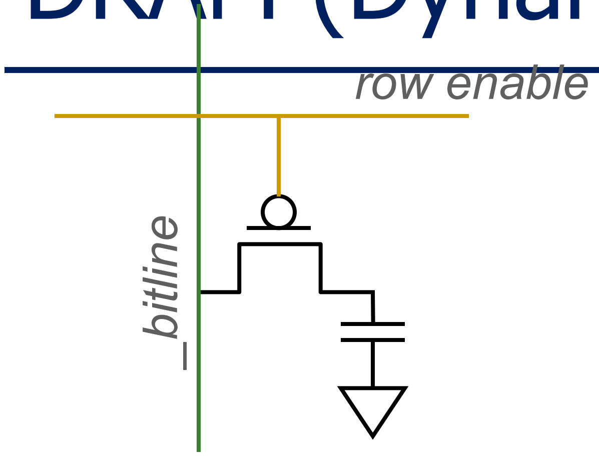
1. address decode
2. drive row select
3. selected bit-cells drive bitlines

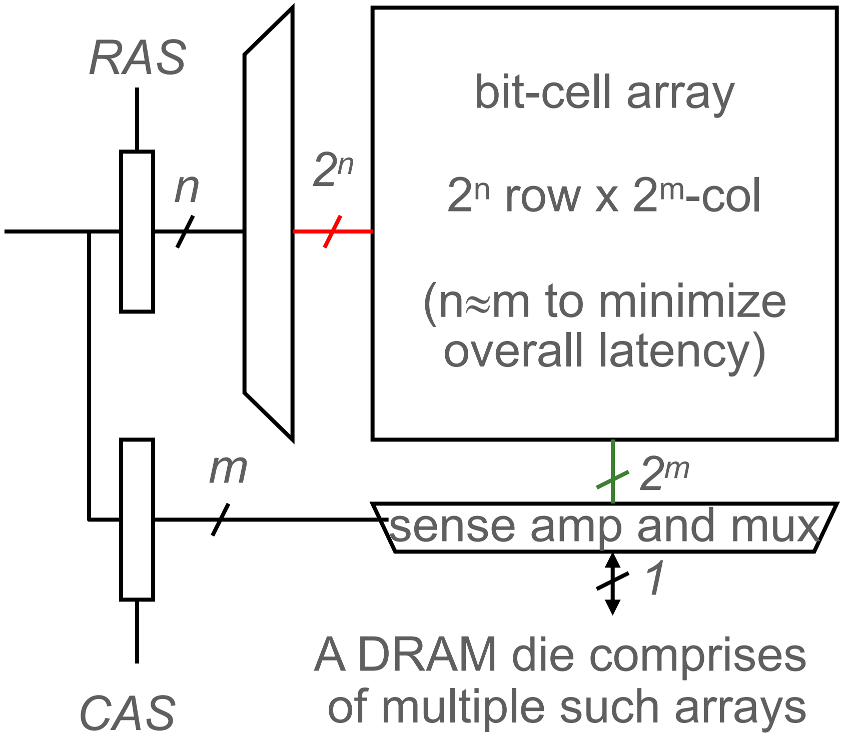
   (entire row is read together)
4. differential sensing and column select

   (data is ready)
5. precharge all bitlines

   (for next read or write)

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 Memory Hierarchy

Ideal Memory

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

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 vs. Disk vs. Tape

- Higher bandwidth is more expensive
  - Need more banks, more ports, higher frequency, or faster technology

The Problem

- 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 (circa ~2011) scale with time
- Other technologies have their place as well
  - Flash memory (mature), PC-RAM, MRAM, RRAM (not mature yet)

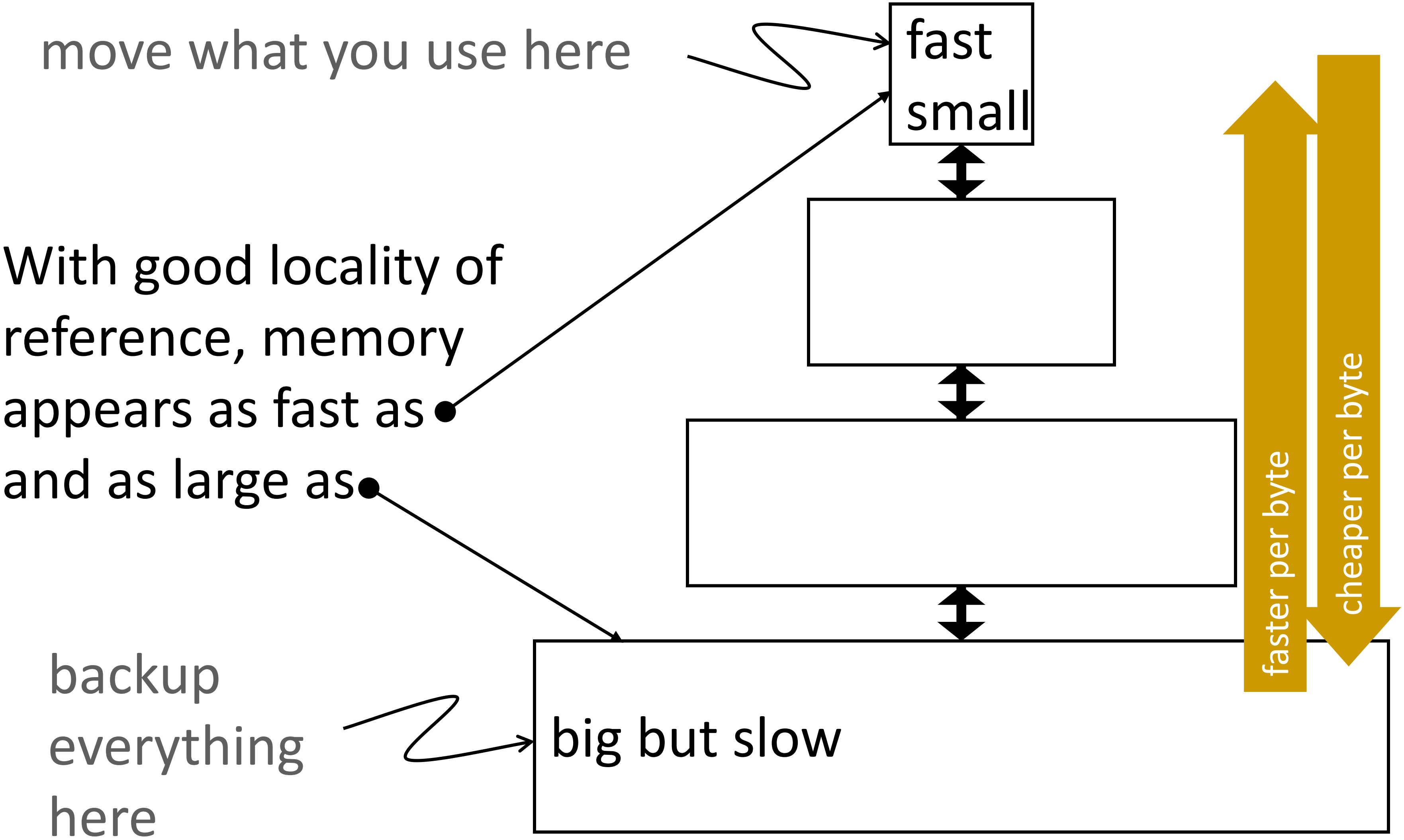
Why Memory Hierarchy?

- 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)

The Memory Hierarchy



move what you use here

fast
small

With good locality of reference, memory appears as fast as ● and as large as ●

faster per byte

cheaper per byte

backup everything here

big but slow

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

Spatial and Temporal Locality Example

- Which memory accesses demonstrate spatial locality?
- Which memory accesses demonstrate temporal locality?

```
int sum = 0;
int X[1000];

for(int c = 0; c < 1000; c++){
  sum += X[c];
}
```

Caching Basics: Exploit Temporal Locality

- Idea: Store recently accessed data in automatically managed fast memory (called cache)
- Anticipation: the data will be accessed again soon

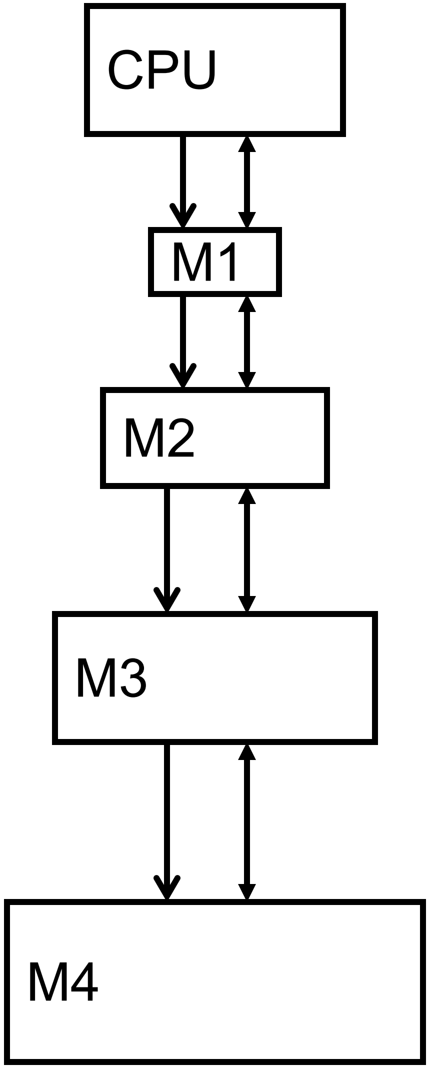
- Temporal locality principle
  - Recently accessed data will be again accessed in the near future

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

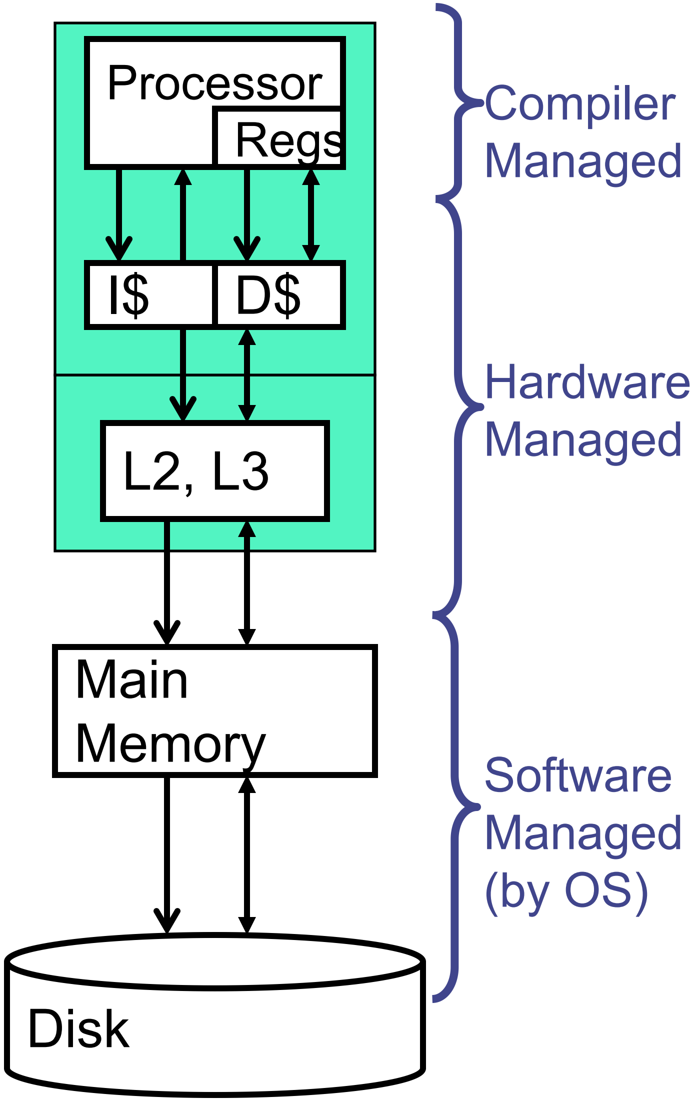
- Spatial locality principle
  - Nearby data in memory will be accessed in the near future
    - E.g., sequential instruction access, array traversal

Exploiting Locality: Memory Hierarchy



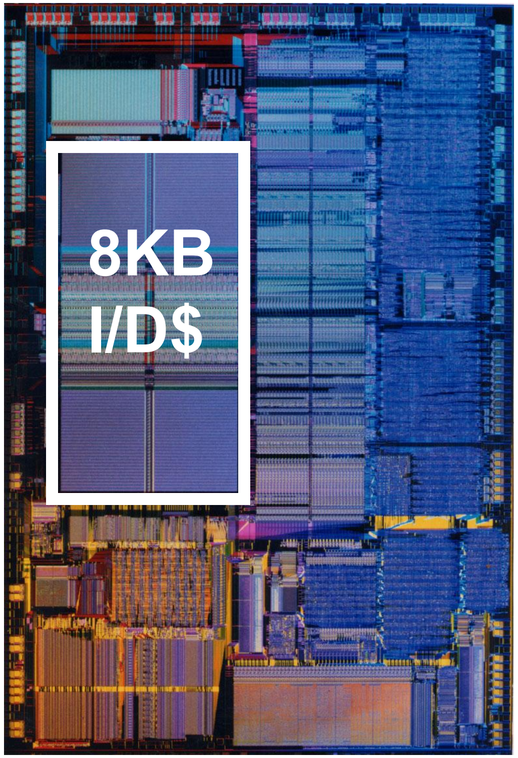
- Hierarchy of memory components
  - Upper components
    - Fast ↔ Small ↔ Expensive
  - Lower components
    - Slow ↔ Big ↔ Cheap
- Connected by “buses”
  - Which also have latency and bandwidth issues
- Most frequently accessed data in M1
  - M1 + next most frequently accessed in M2, etc.
  - Move data up-down hierarchy
- Optimize average access time
  - **$t_{avg}=t_{access} + (\%_{miss}*t_{miss})$**
  - Attack each component

A Modern Memory Hierarchy

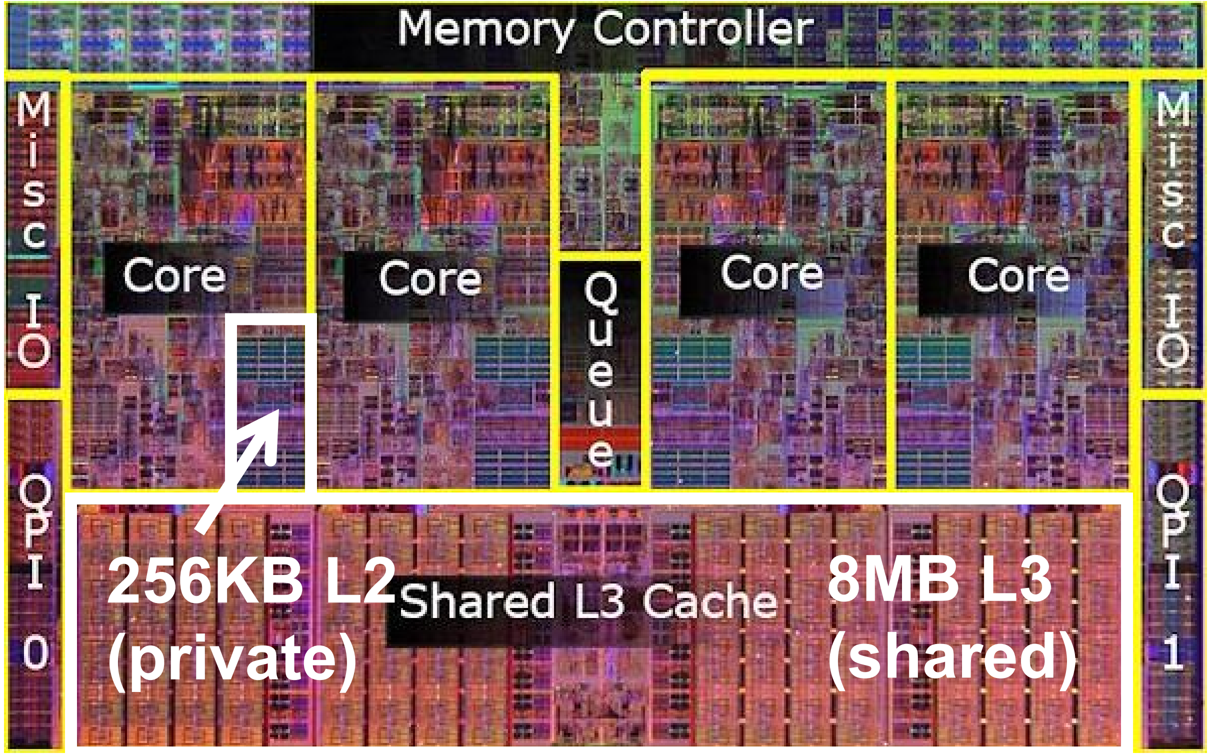


- 0th level: **Registers**
- 1st level: **Primary caches**
  - Split instruction (I$) and data (D$)
  - Typically 8KB to 64KB each
- 2nd level: **2nd and 3rd cache** (L2, L3)
  - On-chip, typically made of SRAM
  - 2nd level typically ~256KB to 512KB
  - “Last level cache” typically 4MB to 16MB
- 3rd level: **main memory**
  - Made of DRAM (“Dynamic” RAM)
  - Typically 1GB to 4GB for desktops/laptops
    - Servers can have 100s of GB
- 4th level: **disk (swap and files)**
  - Uses magnetic disks or flash drives

Evolution of Cache Hierarchies



Intel 486



Intel Core i7 (quad core)

- Chips today are 30–70% cache by area

Cache Basics and Operation

Caching Basics

- Block (line): Unit of storage in the cache
  - Memory is logically divided into cache blocks that map to locations in the cache

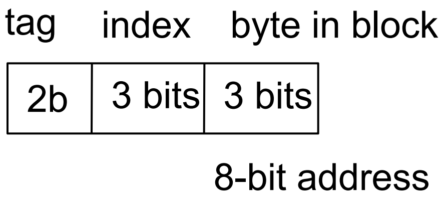
- On a reference:
  - HIT: If in cache, use cached data instead of accessing memory
  - MISS: If not in cache, bring block into cache
    - Maybe have to kick something else out to do it

- 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 or small blocks?
  - Write policy: what do we do about writes?
  - Instructions/data: do we treat them separately?

Blocks and Addressing the Cache

- Memory is logically divided into fixed-size 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:

  1) index into the tag and data stores with index bits in address

  2) check valid bit in tag store

  3) compare tag bits in address with the stored tag in tag store

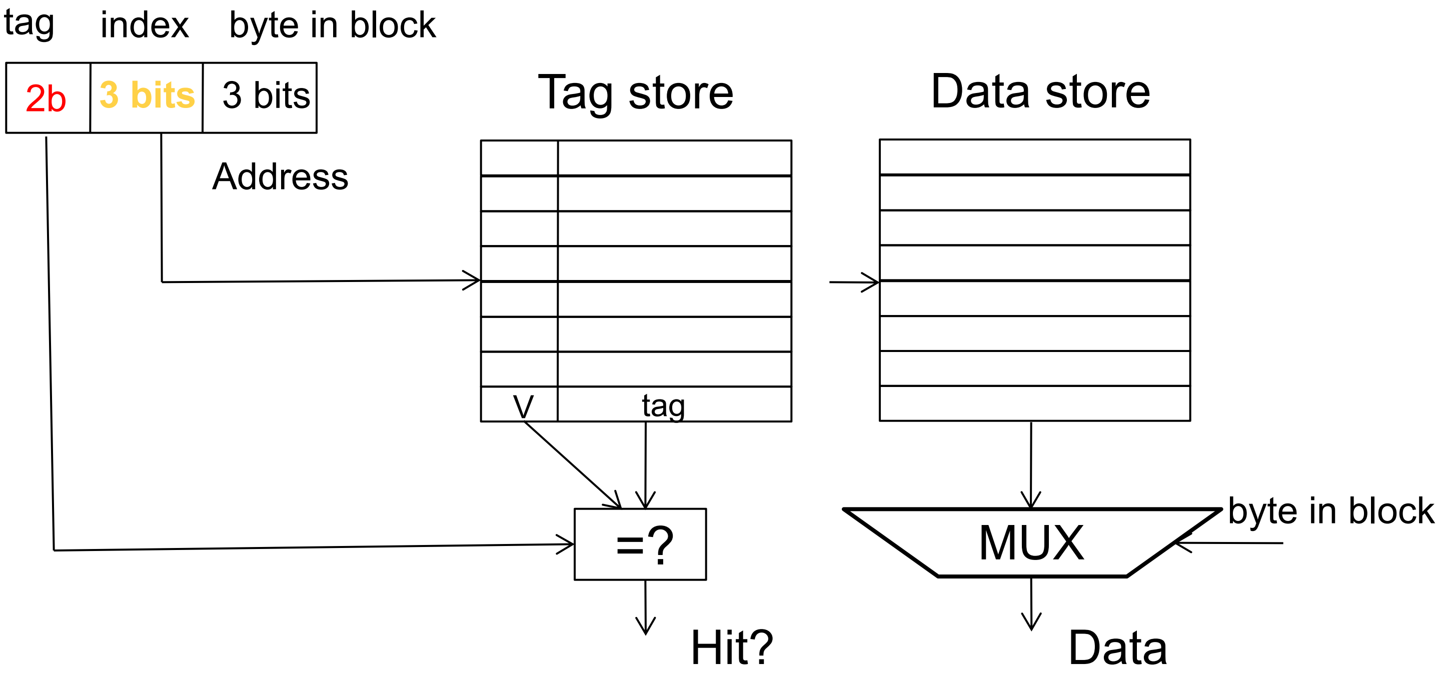
- If a block is in the cache (cache hit), the stored tag should be valid and match the tag of the block

Direct-Mapped Cache: Placement and Access

Block: 00000
Block: 00001
Block: 00010
Block: 00011
Block: 00100
Block: 00101
Block: 00110
Block: 00111
Block: 01000
Block: 01001
Block: 01010
Block: 01011
Block: 01100
Block: 01101
Block: 01110
Block: 01111
Block: 10000
Block: 10001
Block: 10010
Block: 10011
Block: 10100
Block: 10101
Block: 10110
Block: 10111
Block: 11000
Block: 11001
Block: 11010
Block: 11011
Block: 11100
Block: 11101
Block: 11110
Block: 11111

Main memory

- 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

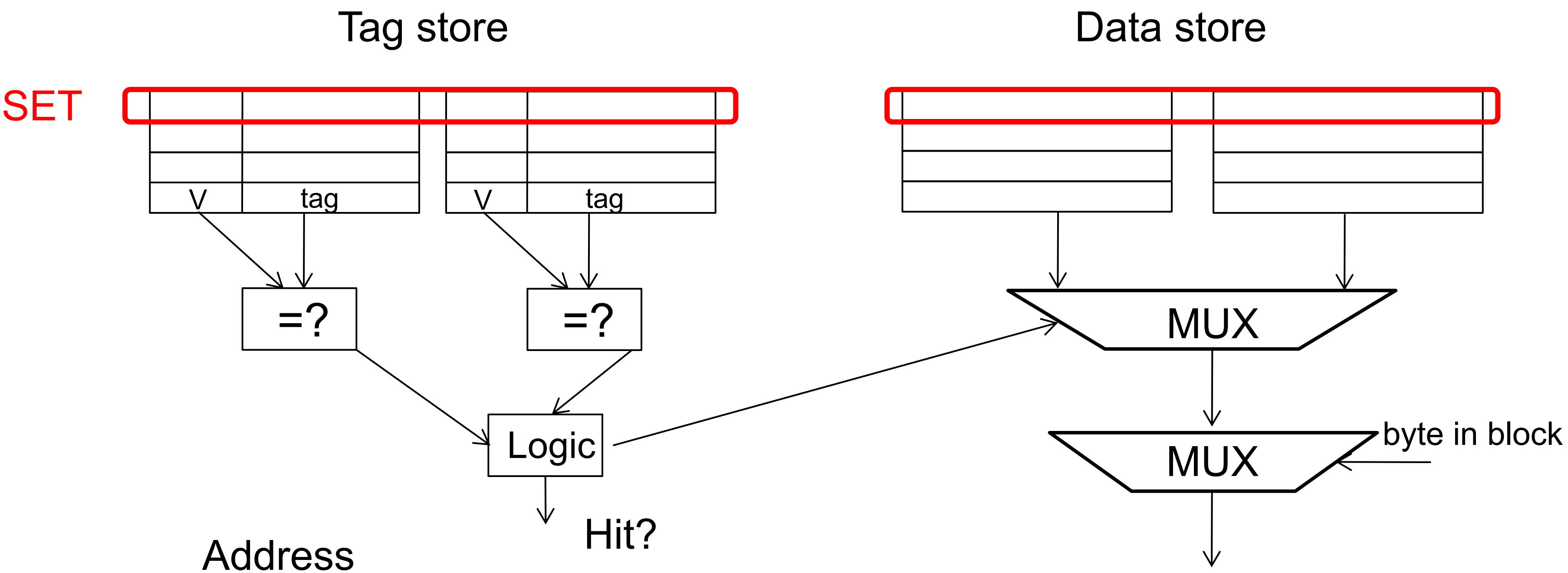
Direct-Mapped Caches

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

- Can lead to 0% hit rate if more than one block accessed 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



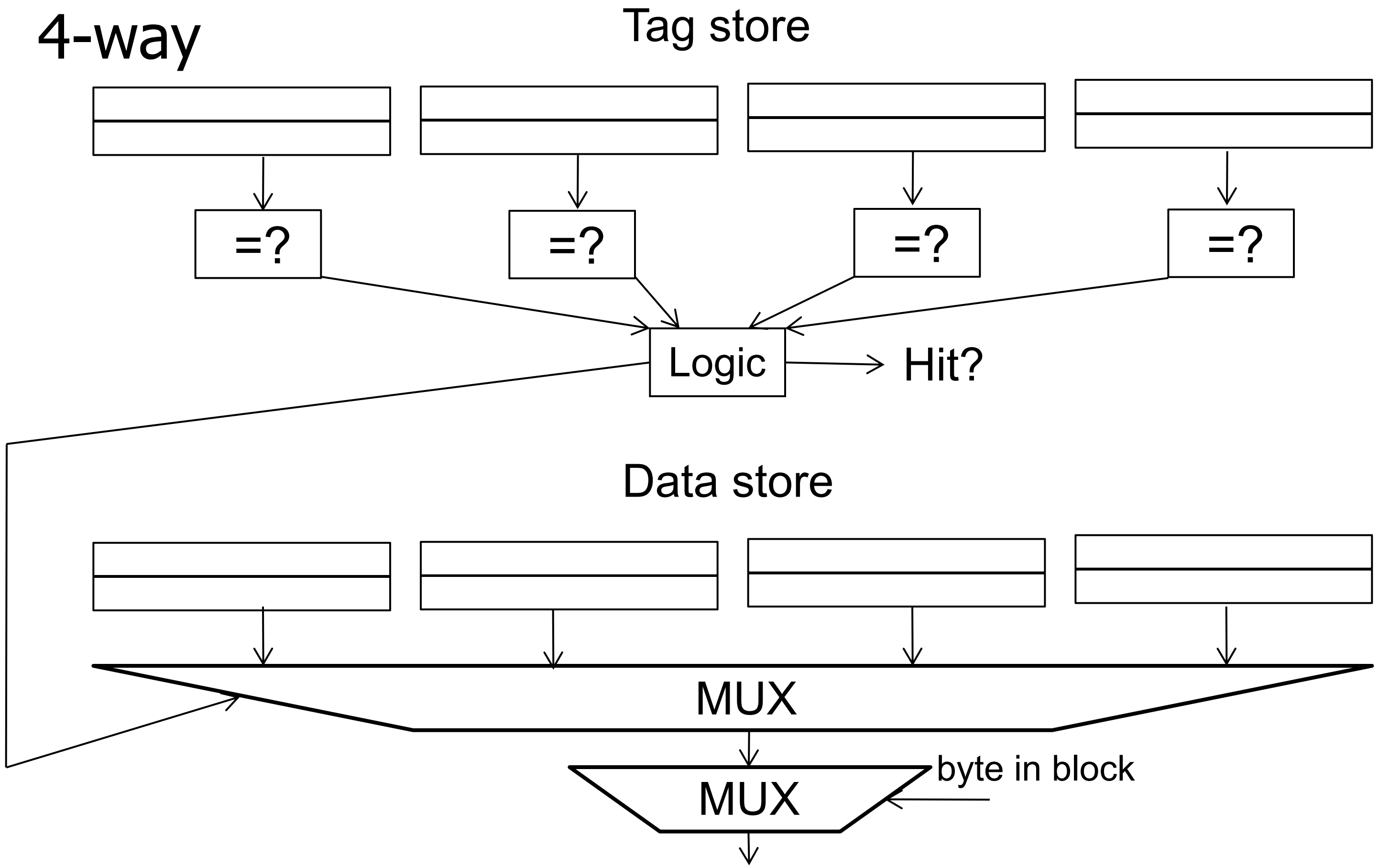
Address

tag	index	byte in block
3b	2 bits	3 bits

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

Higher Associativity

- 4-way

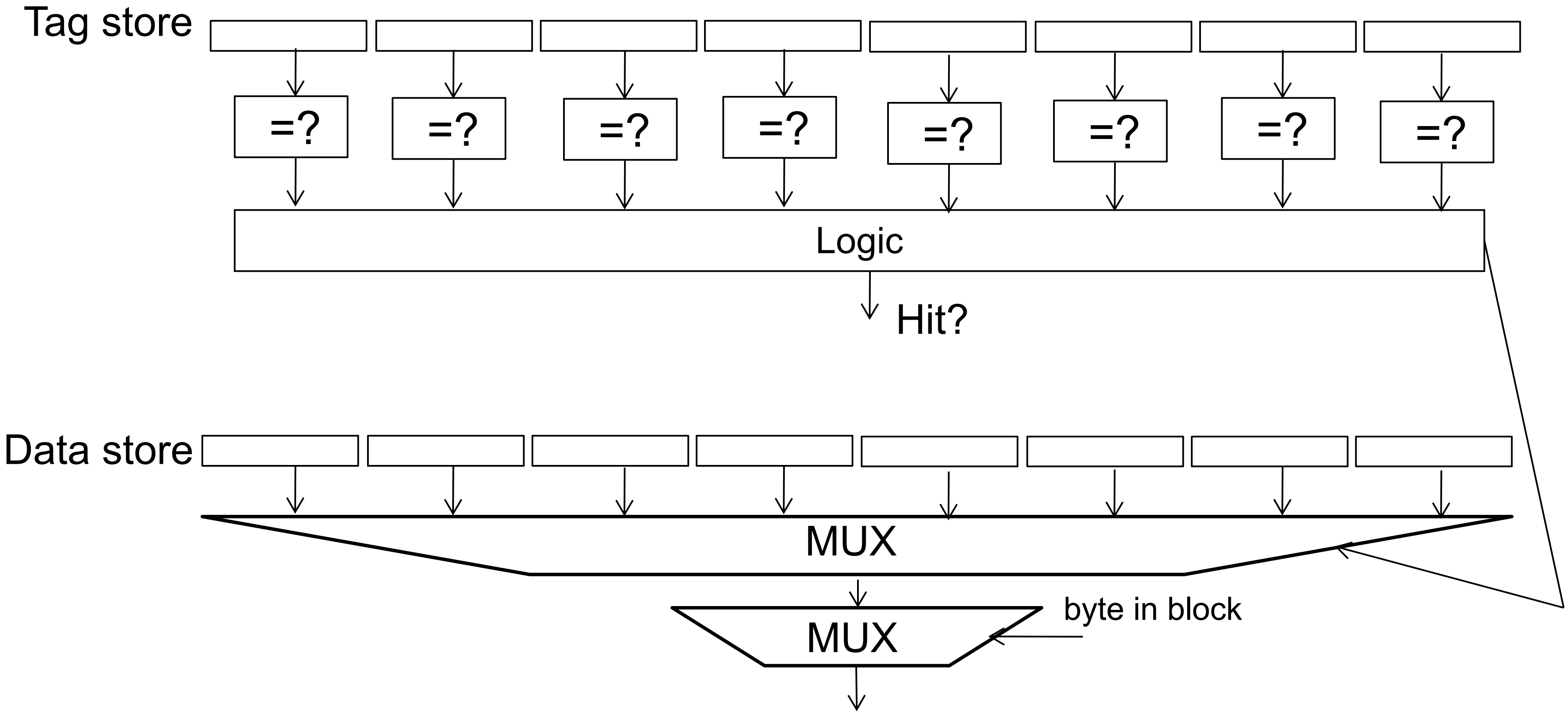


+ Likelihood of conflict misses even lower

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

Full Associativity

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



Associativity (and Tradeoffs)

- Degree of associativity: How many blocks can map to the same index (or set)?

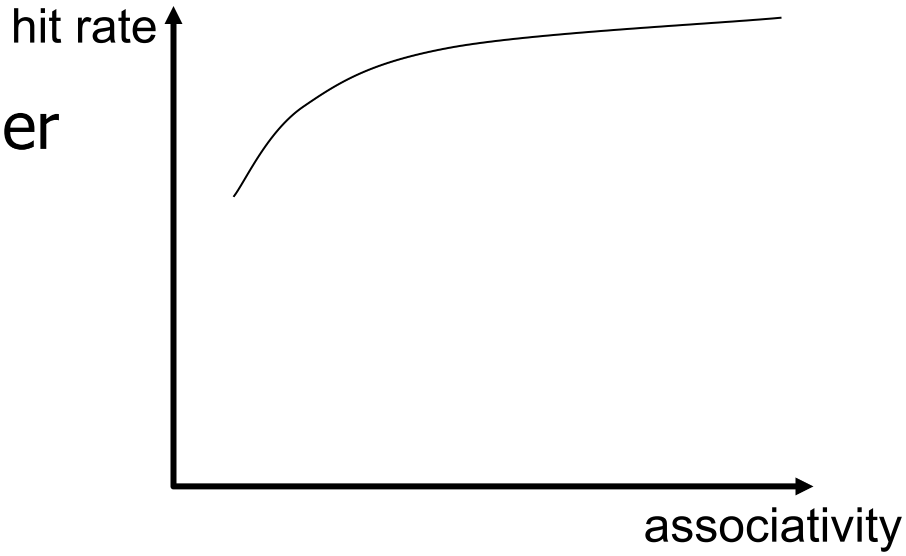
- Higher associativity

  ++ Higher hit rate

  -- Slower cache access time (hit latency and data access latency)

  -- More expensive hardware (more comparators)

- Diminishing returns from higher associativity



Eviction/Replacement Policy

- 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 (first-in first-out)**
    - **LRU (least recently used)**
      - Fits with temporal locality, LRU = least likely to be used in future
    - **NMRU (not most recently used)**
      - An easier to implement approximation of LRU
      - Is LRU for 2-way set-associative caches
    - **Belady's**: replace block that will be used furthest in future
      - Unachievable optimum

Implementing LRU

- Idea: Evict the least recently accessed block
- Problem: Need to keep track of access ordering of blocks

- Question: 2-way set associative cache:
  - What do you need to implement LRU perfectly?

- Question: 4-way set associative cache:
  - What do you need to implement LRU perfectly?
  - How many different orderings possible for the 4 blocks in the set?
  - How many bits needed to encode the LRU order of a block?
  - What is the logic needed to determine the LRU victim?

Approximations of LRU

- Most modern processors do not implement "true LRU" (also called "perfect LRU") in highly-associative caches

- Why?
  - True LRU is complex
  - LRU is an approximation to predict locality anyway (i.e., not the best possible cache management policy)

- Examples:
  - Not MRU (not most recently used)
  - Hierarchical LRU: divide the N-way set into M "groups", track the MRU group and the MRU way in each group

Cache Replacement Policy: LRU or Random

- LRU vs. Random: Which one is better?
  - Example: 4-way cache, cyclic references to A, B, C, D, E
    - 0% hit rate with LRU policy
- Set thrashing: When the "program working set" in a set is larger than set associativity
  - Random replacement policy is better when thrashing occurs
- In practice:
  - Depends on workload
  - Average hit rate of LRU and Random are similar

What Is the Optimal Replacement Policy?

- Belady's OPT
  - Replace the block that is going to be referenced furthest in the future by the program
  - Belady, "A study of replacement algorithms for a virtual-storage computer," IBM Systems Journal, 1966.
  - How do we implement this? Simulate?

- Is this optimal for minimizing miss rate?
- Is this optimal for minimizing execution time?
  - No. Cache miss latency/cost varies from block to block!
  - Two reasons: Remote vs. local caches and miss overlapping
  - Qureshi et al. "A Case for MLP-Aware Cache Replacement,“ ISCA 2006.

Write Issues

- So far we have looked at reading from cache
  - Instruction fetches, loads
- What about writing into cache?

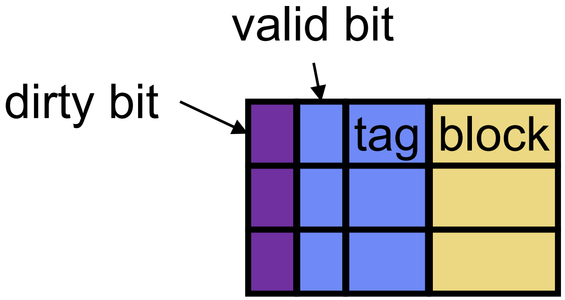
- Several new issues
  - Write-through vs. write-back
  - Write-allocate vs. write-not-allocate

Write Propagation

- When to propagate new value to lower-level caches/memory?
- **Option #1: Write-through**: immediately
  - On hit, update cache
  - Immediately send the write to the next level
- **Option #2: Write-back**: when block is replaced
  - Now we have multiple versions of the same block in various caches and in memory!
  - Requires additional "**dirty**" bit per block
    - Evict **clean** block: **no extra traffic**
      - there was only 1 version of the block
    - Evict **dirty** block: **extra "writeback" of block**
      - the dirty block is the most up-to-date version

Write-back Cache Operation

- Each cache block has a **dirty bit** associated with it
  - state is either **clean** or **dirty**



initial state	-	I	-	-
after `ld r0 <= [A]`	C	V	A	0x01
after `st r1 => [A]`	D	V	A	0x02

Write Propagation Comparison

- **Write-through**
  - – Requires additional bus bandwidth
    - Consider repeated write hits
  - – Next level must handle small writes (1, 2, 4, 8-bytes)
  - + No need for dirty bits in cache
  - + No need to handle “writeback” operations
    - Simplifies miss handling
    - Used in GPUs, as they have low write temporal locality
- **Write-back**
  - + Key advantage: uses less bandwidth
  - Reverse of other pros/cons above
  - Used in most CPU designs

Write Miss Handling

- How is a write miss actually handled?

- **Write-allocate**: fill block from next level, then write it
  - + Decreases read misses (next read to block will hit)
  - – Requires additional bandwidth
  - Commonly used (especially with write-back caches)
- **Write-non-allocate**: just write to next level, no allocate
  - – Potentially more read misses
  - + Uses less bandwidth
  - Use with write-through

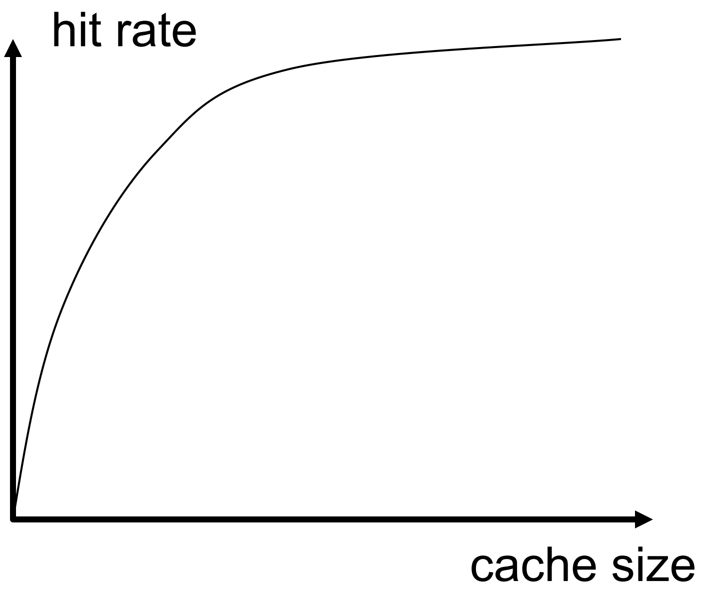
Improve Cache Performance

Cache Parameters vs. Miss/Hit Rate

- Cache size
- Block size
- Associativity
- Replacement policy
- Insertion/Placement policy

Cache Size

- Cache size: total data (not including tag) capacity
  - bigger can exploit temporal locality better
  - not ALWAYS better
- Too large a cache adversely affects hit and miss latency
  - smaller is faster => bigger is slower
  - access time may degrade critical path
- Too small a cache
  - doesn't exploit temporal locality well
  - useful data replaced often



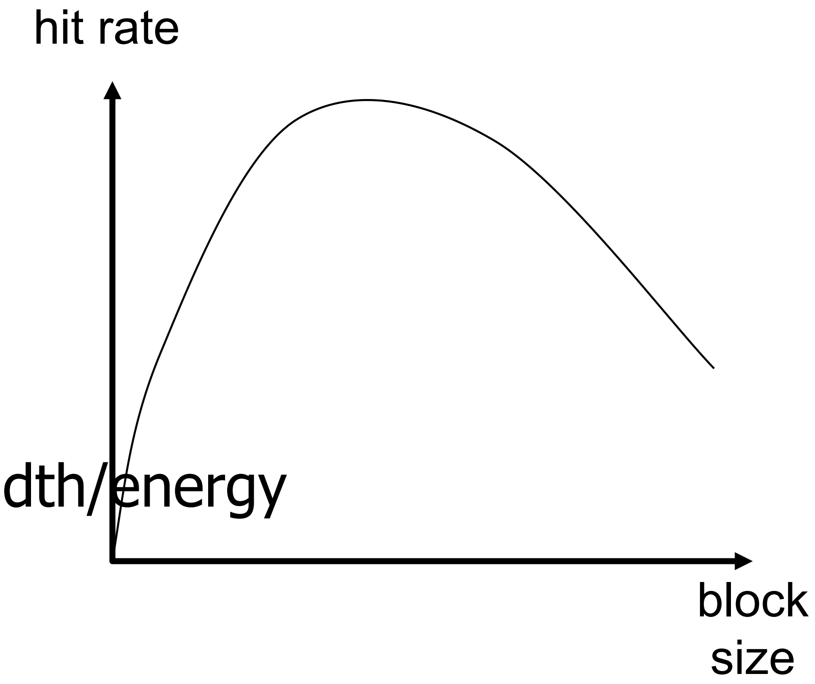
- Working set: the whole set of data the executing application references
  - Within a time interval

Block Size

- Block size is the data that is associated with an address tag

- Too small blocks
  - don't exploit spatial locality well
  - have larger tag overhead

- Too large blocks
  - too few total # of blocks → less temporal locality exploitation
  - waste of cache space and bandwidth/energy if spatial locality is not high

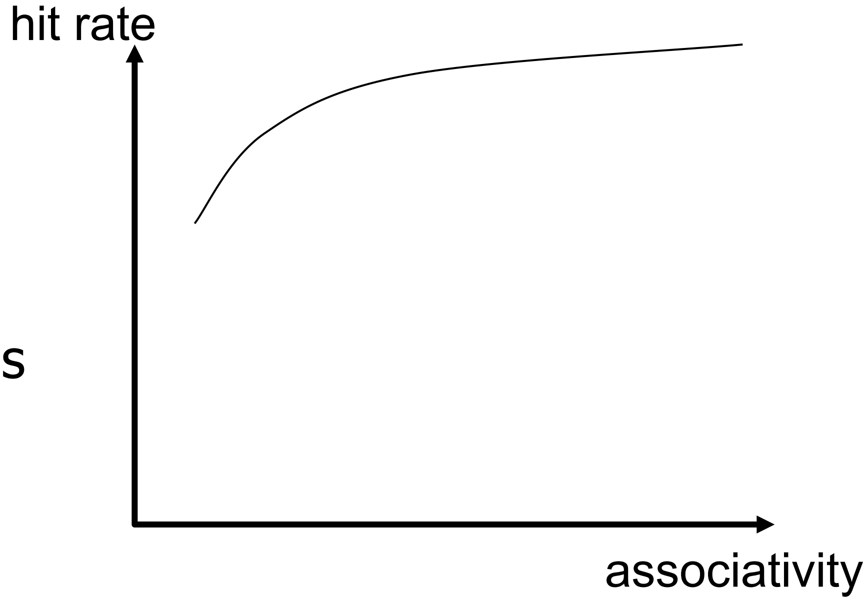


Associativity

- How many blocks can be present in the same index (i.e., set)?

- Larger associativity
  - lower miss rate (reduced conflicts)
  - higher hit latency and area cost (plus diminishing returns)

- Smaller associativity
  - lower cost
  - lower hit latency
    - Especially important for L1 caches



Classification of Cache Misses (3C)

- Compulsory miss
  - first reference to an address (block) always results in a miss
  - subsequent references should hit unless the cache block is displaced for the reasons below

- Capacity miss
  - cache is too small to hold everything needed (identify?)
  - defined as the misses that would occur even in a fully-associative cache (with optimal replacement) of the same capacity

- Conflict miss
  - defined as any miss that is neither a compulsory nor a capacity miss

How to Reduce Each Miss Type

- Compulsory
  - Caching cannot help
  - Prefetching can: Anticipate which blocks will be needed soon
- Conflict
  - More associativity
  - Other ways to get more associativity without making the cache associative
    - Victim cache
    - Better, randomized indexing
- Capacity
  - Utilize cache space better: keep blocks that will be referenced
  - Software management: divide working set and computation such that each "computation phase" fits in cache

How to Improve Cache Performance

- Three fundamental goals

- Reducing miss rate

- Reducing miss latency or miss cost

- Reducing hit latency or hit cost

- The above three **together** affect performance

Improving Basic Cache Performance

- Reducing miss rate
  - More associativity
  - Alternatives/enhancements to associativity
    - Victim caches, hashing, pseudo-associativity, skewed associativity
  - Better replacement/insertion policies
  - Software approaches
- Reducing miss latency/cost
  - Multi-level caches
  - Critical word first
  - Subblocking/sectoring
  - Better replacement/insertion policies
  - Non-blocking caches (multiple cache misses in parallel)
  - Multiple accesses per cycle
  - Software approaches

Software Approaches for Higher Hit Rate

- Restructuring data access patterns
- Restructuring data layout

- Loop interchange
- Data structure separation/merging
- Blocking
- …

Restructuring Data Access Patterns (I)

- **Loop interchange:** spatial locality
  - Example: row-major matrix: `X[i][j]` followed by `X[i][j+1]`
  - Poor code: `X[i][j]` followed by `X[i+1][j]`

```
for (j = 0; j<NCOLS; j++)
    for (i = 0; i<NROWS; i++)
        sum += X[i][j];
```

  - Better code

```
for (i = 0; i<NROWS; i++)
    for (j = 0; j<NCOLS; j++)
        sum += X[i][j];
```

Restructuring Data Access Patterns (II)

- **Loop blocking**: temporal locality
  - Poor code

```
for (k=0; k<NUM_ITERATIONS; k++)
    for (i=0; i<NUM_ELEMS; i++)
        X[i] = f(X[i]);    // say
```

  - Better code
    - Cut array into CACHE_SIZE chunks
    - Run all phases on one chunk, proceed to next chunk

```
for (i=0; i<NUM_ELEMS; i+=CACHE_SIZE)
    for (k=0; k<NUM_ITERATIONS; k++)
        for (j=0; j<CACHE_SIZE; j++)
            X[i+j] = f(X[i+j]);
```

  – Assumes you know **CACHE_SIZE**, do you?

Restructuring Data Layout (I)

```
struct Node {
    struct Node* next;
    int key;
    char [256] name;
    char [256] school;
}

while (node) {
    if (node→key == input-key) {
        // access other fields of node
    }
    node = node→next;
}
```

- Pointer based traversal (e.g., of a linked list)
- Assume a huge linked list (1B nodes) and unique keys

- Why does the code on the left have poor cache hit rate?
    - “Other fields” occupy most of the cache line even though rarely accessed!

Restructuring Data Layout (II)

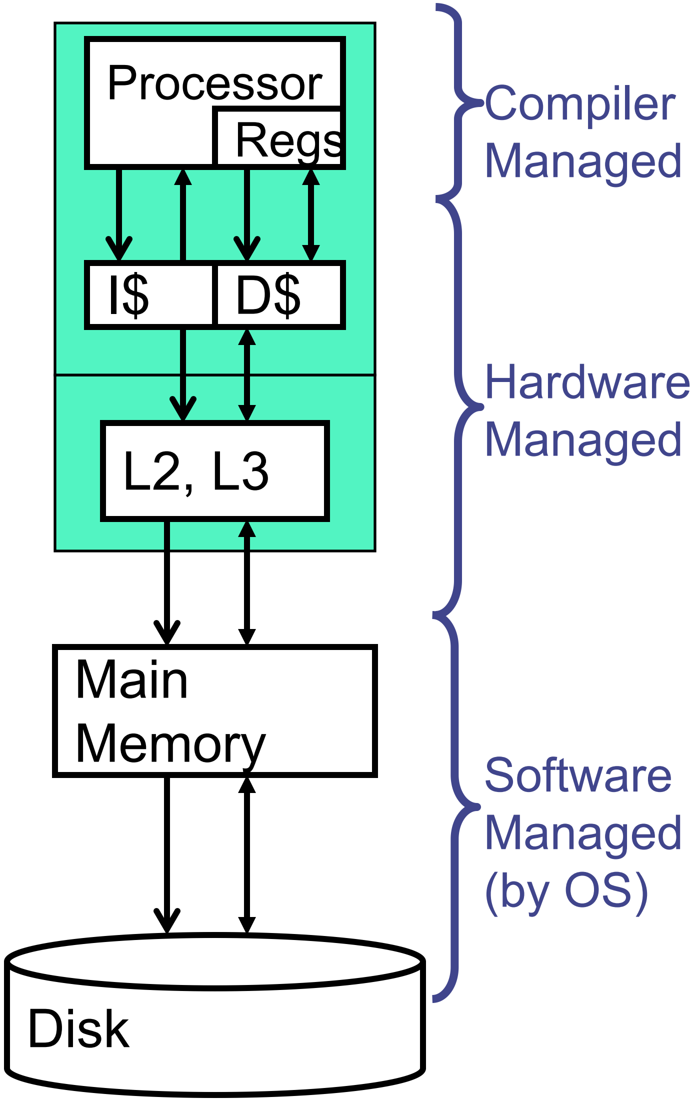
```
struct Node {
    struct Node* next;
    int key;
    struct Node-data* node-data;
}

struct Node-data {
    char [256] name;
    char [256] school;
}

while (node) {
     if (node→key == input-key) {
          // access node→node-data
     }
     node = node→next;
}
```

- Idea: separate frequently-used fields of a data structure and pack them into a separate data structure

- Who should do this?
  - Programmer
  - Compiler
    - Profiling vs. dynamic
  - Hardware?
  - Who can determine what is frequently used?

Memory Hierarchy Performance

A Modern Memory Hierarchy



- 0th level: **Registers**
- 1st level: **Primary caches**
  - Split instruction (I$) and data (D$)
  - Typically 8KB to 64KB each
- 2nd level: **2nd and 3rd cache** (L2, L3)
  - On-chip, typically made of SRAM
  - 2nd level typically ~256KB to 512KB
  - "Last level cache" typically 4MB to 16MB
- 3rd level: **main memory**
  - Made of DRAM ("Dynamic" RAM)
  - Typically 1GB to 4GB for desktops/laptops
    - Servers can have 100s of GB
- 4th level: **disk (swap and files)**
  - Uses magnetic disks or flash drives

Designing a Cache Hierarchy

- For any memory component: t_{access} vs. $\%_{miss}$ tradeoff

- Upper components (I$, D$) emphasize low t_{access}
  - Frequent access → t_{access} important
  - t_{miss} is not bad → $\%_{miss}$ less important
  - Lower capacity and lower associativity (to reduce t_{access})
  - Small-medium block-size (to reduce conflicts)

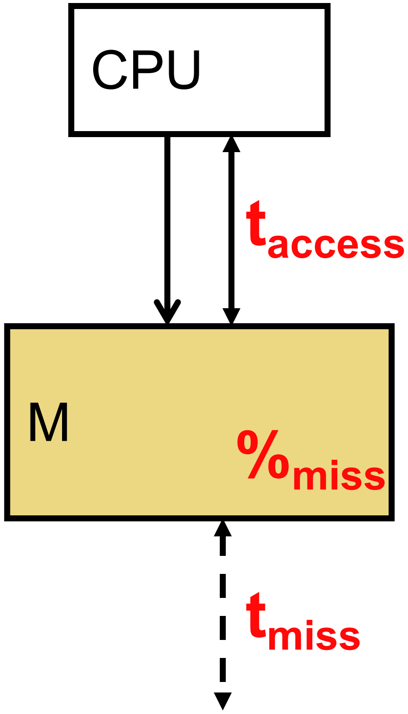
- Moving down (L2, L3) emphasis turns to $\%_{miss}$
  - Infrequent access → t_{access} less important
  - t_{miss} is bad → $\%_{miss}$ important
  - High capacity, associativity, and block size (to reduce $\%_{miss}$)

Memory Hierarchy Parameters

Parameter	**I$/D$**	**L2**	**L3**	**Main Memory**
t_{access}	2ns	10ns	30ns	100ns
t_{miss}	**10ns**	**30ns**	**100ns**	**10ms (10M ns)**
Capacity	8KB–64KB	256KB–8MB	2–16MB	1-4GBs
Block size	16B–64B	32B–128B	32B-256B	NA
Associativity	2-8	4–16	4-16	NA

- Some other design parameters
  - Split vs. unified insns/data
  - Inclusion vs. exclusion vs. nothing

Memory Performance Equation



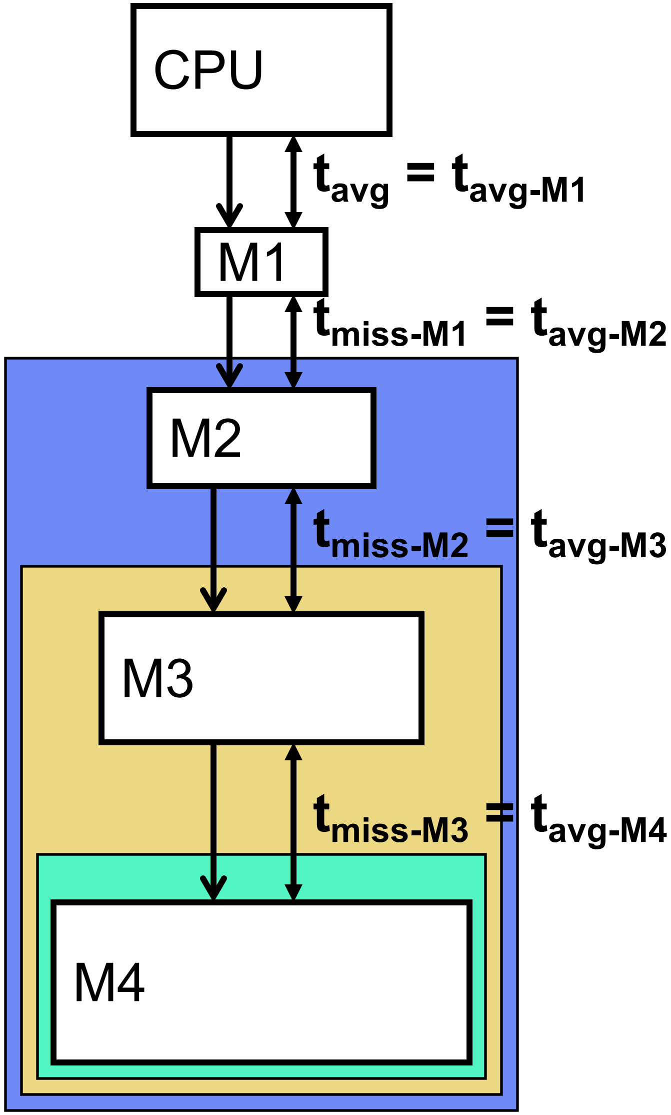
- For memory component M
  - **Access**: read or write to M
  - **Hit**: desired data found in M
  - **Miss**: desired data not found in M
    - Must get from another (slower) component
  - **Fill**: action of placing data in M

  - $\%_{miss}$ (miss-rate): #misses / #accesses
  - t_{access}: time to read data from (write data to) M
  - t_{miss}: time to read data into M

- Performance metric
  - t_{avg}: average access time

$$t_{avg} = t_{access} + (\%_{miss} * t_{miss})$$

Hierarchy Performance


CPU
$t_{avg} = t_{avg\text{-}M1}$
M1
$t_{miss\text{-}M1} = t_{avg\text{-}M2}$
M2
$t_{miss\text{-}M2} = t_{avg\text{-}M3}$
M3
$t_{miss\text{-}M3} = t_{avg\text{-}M4}$
M4

t_{avg}

$t_{avg\text{-}M1}$

$t_{acc\text{-}M1} + (\%_{miss\text{-}M1} * t_{miss\text{-}M1})$

$t_{acc\text{-}M1} + (\%_{miss\text{-}M1} * t_{avg\text{-}M2})$

$t_{acc\text{-}M1} + (\%_{miss\text{-}M1} * (t_{acc\text{-}M2} + (\%_{miss\text{-}M2} * t_{miss\text{-}M2})))$

$t_{acc\text{-}M1} + (\%_{miss\text{-}M1} * (t_{acc\text{-}M2} + (\%_{miss\text{-}M2} * t_{avg\text{-}M3})))$

…

Split vs. Unified Caches

- **Split I$/D$**: insns and data in different caches
  - To minimize structural hazards and t_{access}
  - Larger unified I$/D$ would be slow, 2nd port even slower
  - Optimize I$ and D$ separately
    - Not writes for I$, smaller reads for D$

- **Unified L2, L3**: insns and data together
  - To minimize $\%_{miss}$

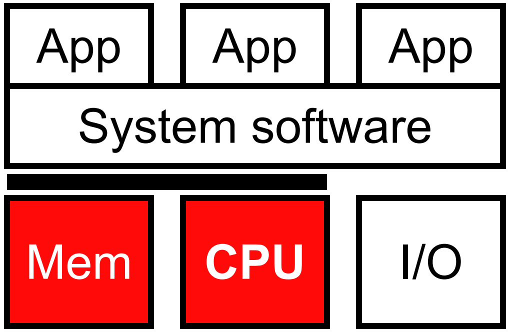
  + Fewer capacity misses: unused insn capacity can be used for data

  – More conflict misses: insn/data conflicts
    - A much smaller effect in large caches
  - Insn/data structural hazards are rare: simultaneous I$/D$ miss
  - Go even further: unify L2, L3 of multiple cores in a multi-core

Hierarchy: Inclusion versus Exclusion

- **Inclusion**
  - Bring block from memory into L2 then L1
    - A block in the L1 is always in the L2
  - If block evicted from L2, must also evict it from L1
    - Why? more on this when we talk about multicore
- **Exclusion**
  - Bring block from memory into L1 but not L2
    - Move block to L2 on L1 eviction
      - L2 becomes a large victim cache
    - Block is either in L1 or L2 (never both)
  - Good if L2 is small relative to L1
    - Example: AMD's Duron 64KB L1s, 64KB L2

Summary



- **Average access time** of a memory component
  - $t_{avg} = t_{access} + \%_{miss} * t_{miss}$
  - Hard to get low t_{access} and $\%_{miss}$ in one structure → build a hierarchy instead
- **Memory hierarchy**
  - Cache (SRAM) → memory (DRAM) → swap (Disk)
  - Smaller, faster, more expensive → bigger, slower, cheaper
- **Associativity, block size, capacity**
  - 3C miss model: compulsory, capacity, conflict
- **Performance optimizations**
  - $\%_{miss}$: prefetching
  - t_{miss}: victim buffer, critical-word-first
- **Write issues**
  - Write-back vs. write-through, write-allocate vs. write-no-allocate