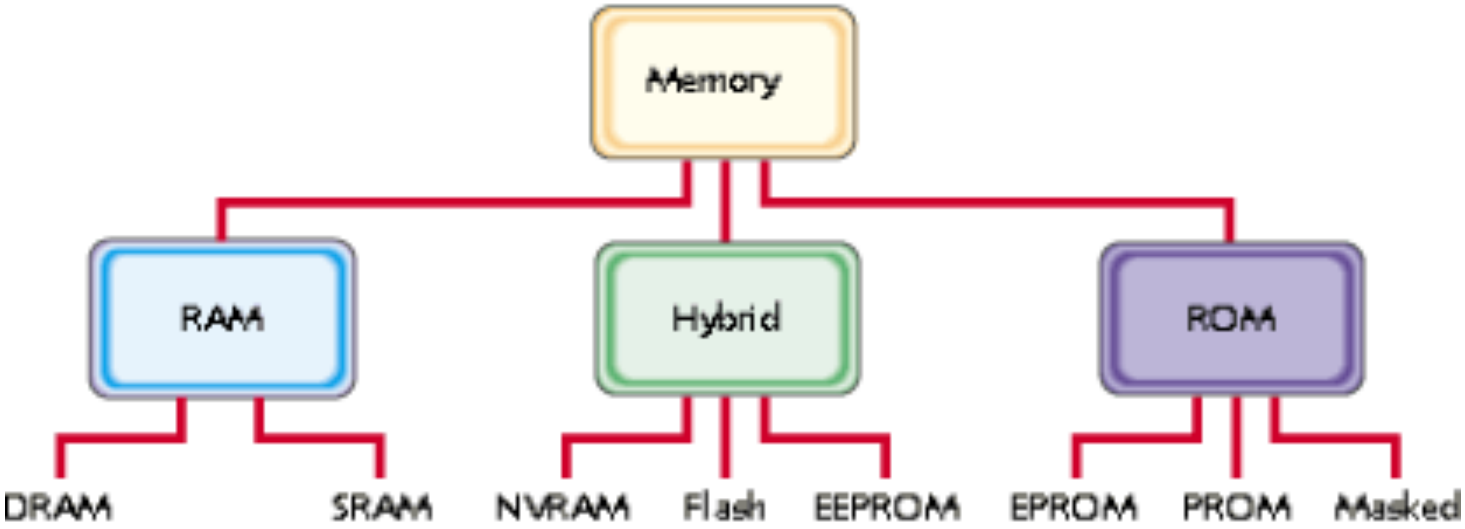
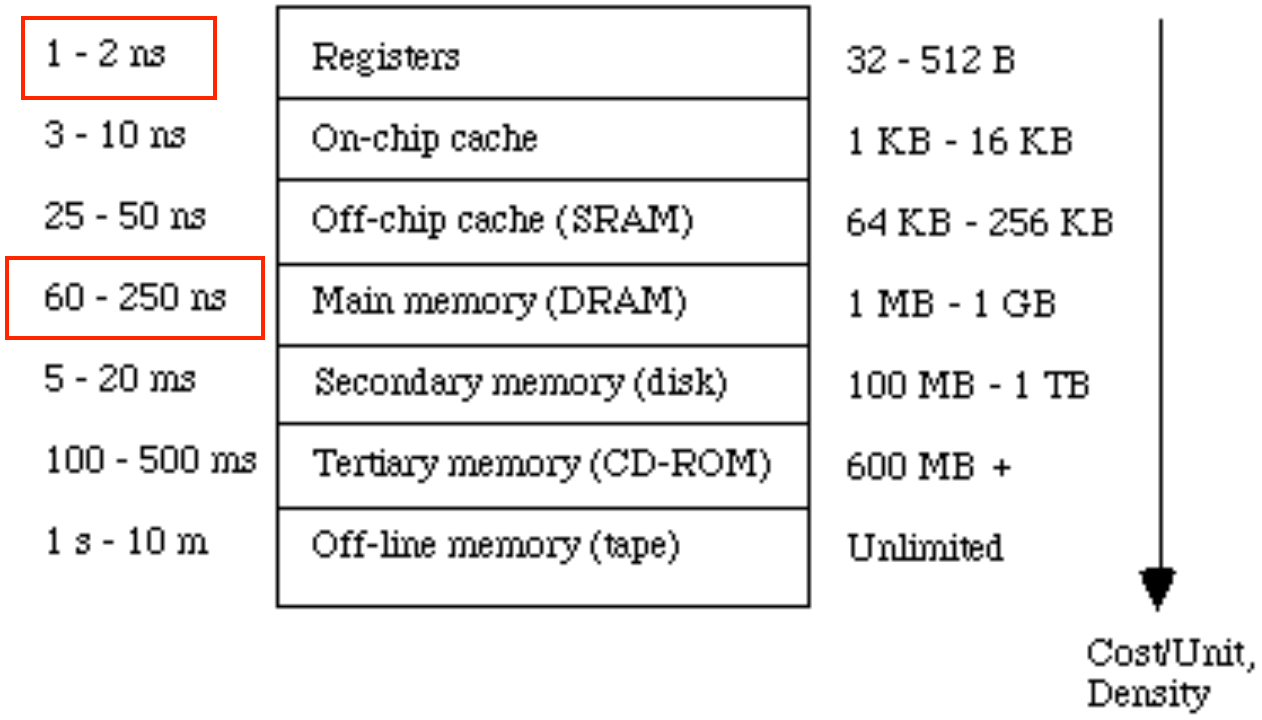
COMP2611: Computer Organization

Memory System



Major Goals

- To introduce the **memory hierarchy** and the **principle of locality**
- To introduce basic **cache** concepts
- To introduce basic **virtual memory** concepts



Memory Revisit

- RAM (Random Access Memory) includes two types
  - DRAM (Dynamic RAM) — people often refer to it as **memory**
  - SRAM (Static RAM) — mostly used for **cache** (explained later)

- From now on, we explicitly distinguish memory and cache

- Memory is
  - Storage for instructions and data of an executing program
  - Has to be of large capacity and cheap (so as to be affordable)
  - DRAM provides this feature

Dynamic Random Access Memory (DRAM)

- DRAM
  - Consists of bits of data stored in a separate capacitors
  - Hold a logical "1" by having capacitor charged, and vice versa
- Capacitors are
  - Structurally simple, only one transistor and a one capacitor per bit
  - High density with low cost
- Major problem of using capacitors:
  - Real-world capacitors are not ideal, hence leak electrons
  - Information stored in capacitors eventually fades
  - Need to recharge the capacitors periodically to restore the "values"
  - Read operation discharges the electrons: read is destructive
- Because of this refresh requirement, it is called dynamic memory

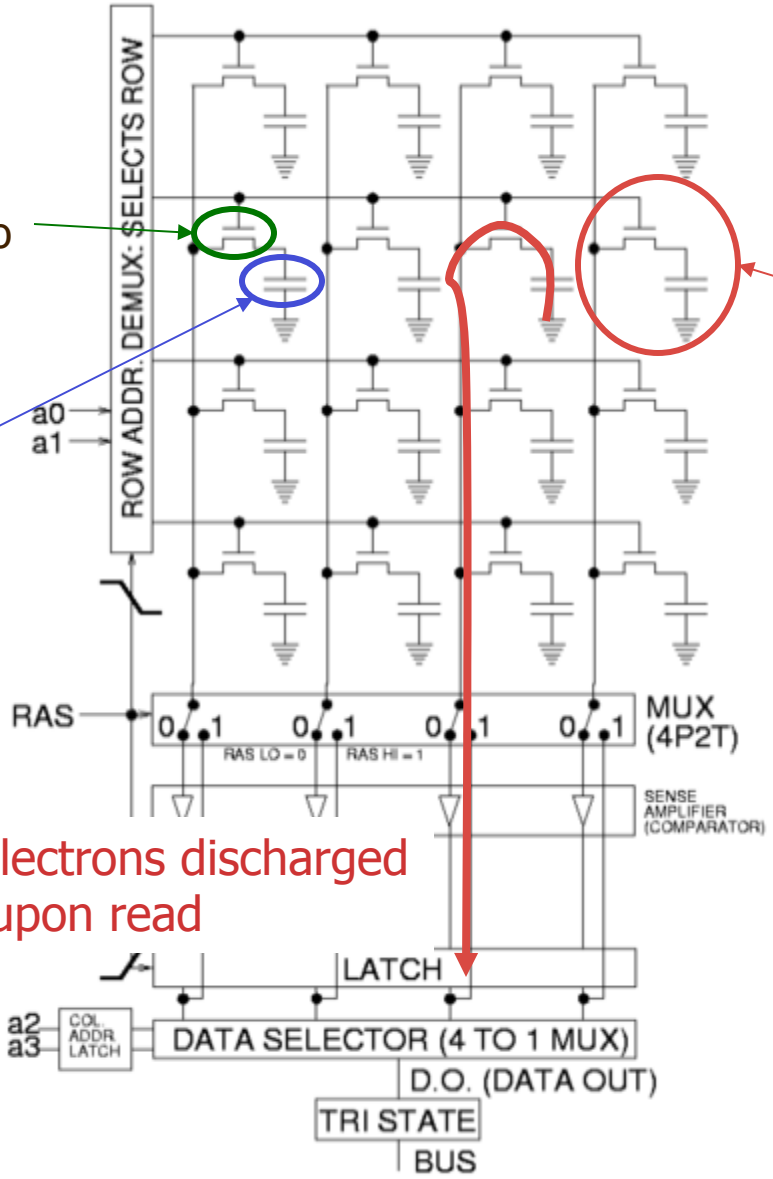
Example: 4x4 DRAM Array

Transistor:

To enable read/write to the capacitor

Capacitor:

To store the "value"



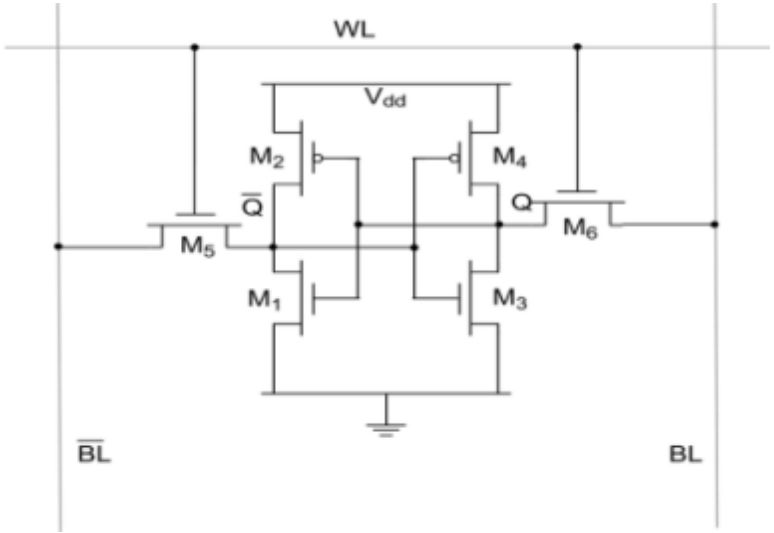
Electrons discharged upon read

A memory cell.

The first DRAM cell was invented in 1966 by Robert Dennard, a researcher at IBM's Thomas J. Watson Research Center

Static Random Access Memory (SRAM)

- SRAM is a type of semiconductor memory
  - Each cell is constructed using transistors only (i.e. no capacitor)
  - Density not as high as DRAM, it is more expensive
  - Does not need to be periodically refreshed, the memory retains its contents as long as power remains applied
- Read is not destructive in SRAM design; that's why called static



- SRAM ≠ SDRAM; SDRAM ≡ Synchronous DRAM

Access Time: DRAM vs SRAM

DRAM

- Slow; because capacitor charging and discharging times are long
- Can be as long as hundreds of cycles w.r.t. processor's speed
- Usually used as main memory

SRAM

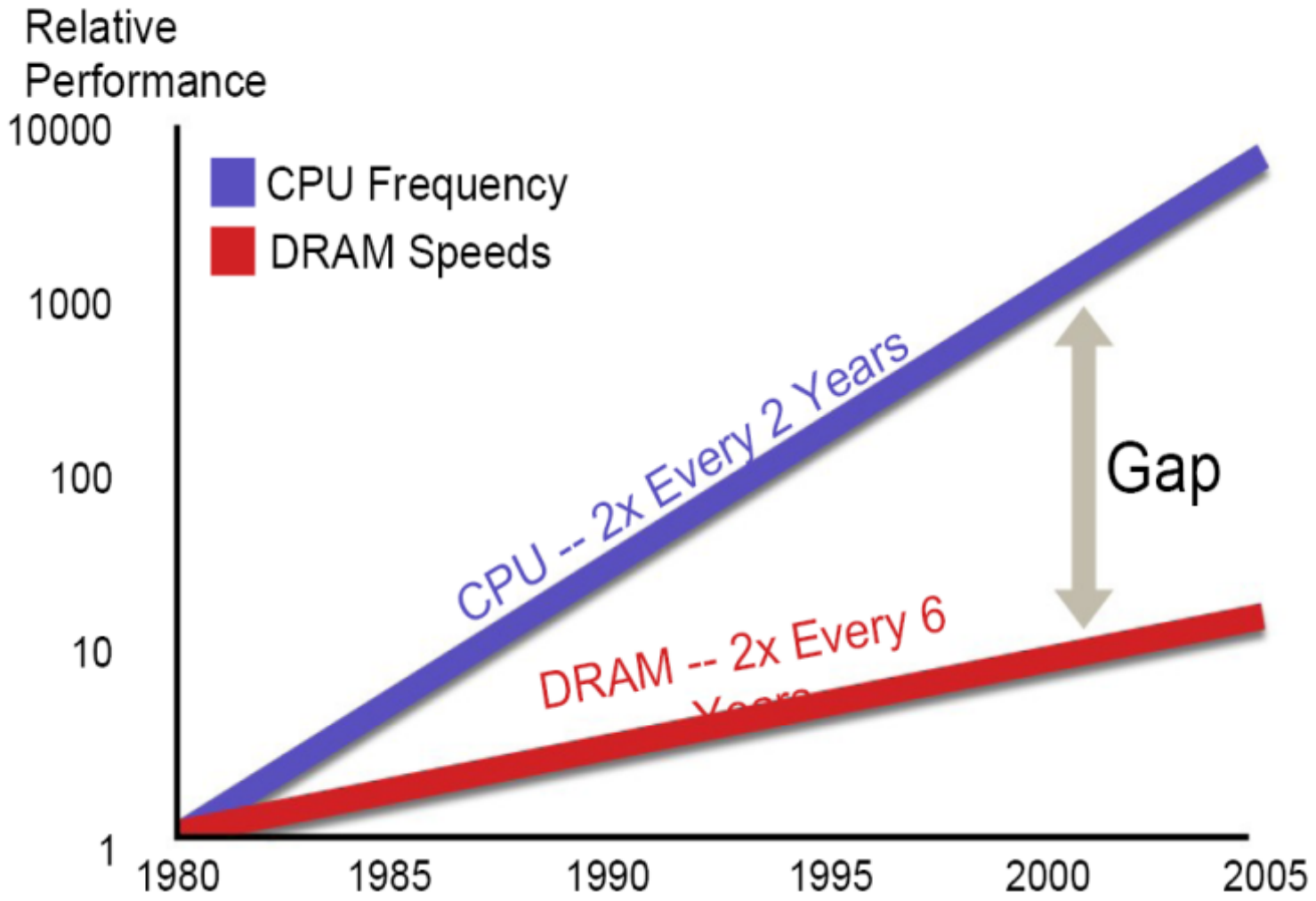
- Fast; because switching time at transistor level is short
- May take a few cycles to a few tens of cycles, depending on the size
- Mostly used as caches inside the CPU

Before: discussions always assumed that memory access takes one cycle
From now on, this assumption is dropped

- Implication ⇒ performance of processor looks bad with this reality

- In this chapter, we'll look into ways to minimize the impact of this

Memory Bottleneck



Memory Bottleneck

- DRAM improvement is not keeping up with processor improvement
  - Processors are getting much faster than memories over time
  - The speed gap is widening!

- Using SRAM?
  - Impractical & uneconomical to have a large amount of SRAM inside the computer

- As a result, memory access can be a bottleneck
  - Because processor spends long time waiting for memory access

Solution

Use a **memory hierarchy** exploiting the **principle of locality**

Principle of Locality

Programs usually access a relatively small portion of their address space (for instructions or data) at any instant of time

Such execution pattern exhibits two types of localities:

- **Temporal locality**:
  If an instruction or data item is referenced
  ⇒ it will tend to be referenced again soon
  e.g. loops in programs contribute to temporal locality
- **Spatial locality**:
  If an instruction or data item is referenced
  ⇒ items whose addresses are close by will tend to be referenced soon
  e.g. data arrays, sequential instructions contribute to spatial locality

This property is the KEY to memory hierarchy!

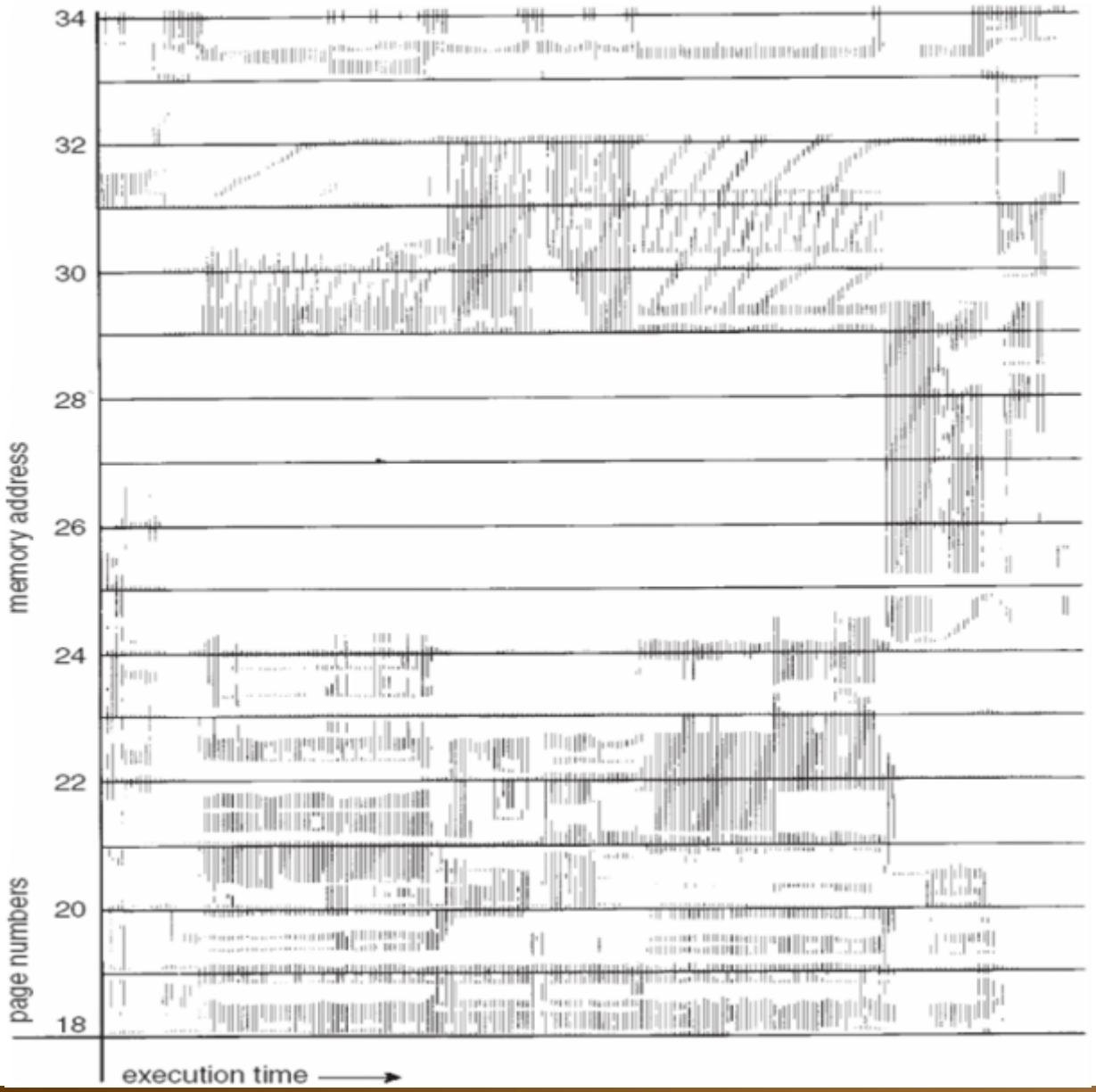
Example: identifying temporal and spatial locality

- Summing up 100 values stored in memory:

```
        addi    $s0, $zero, 0       # $s0: accumulator
        addi    $s1, $zero, 100     # $s1: counter
L1:     lw      $t1, 0($s2)         # $s2: memory addr
        add     $s0, $s0, $t1
        addi    $s2, $s2, 4
        subi    $s1, $s1, 1
        bne     $s1, $zero, L1
```

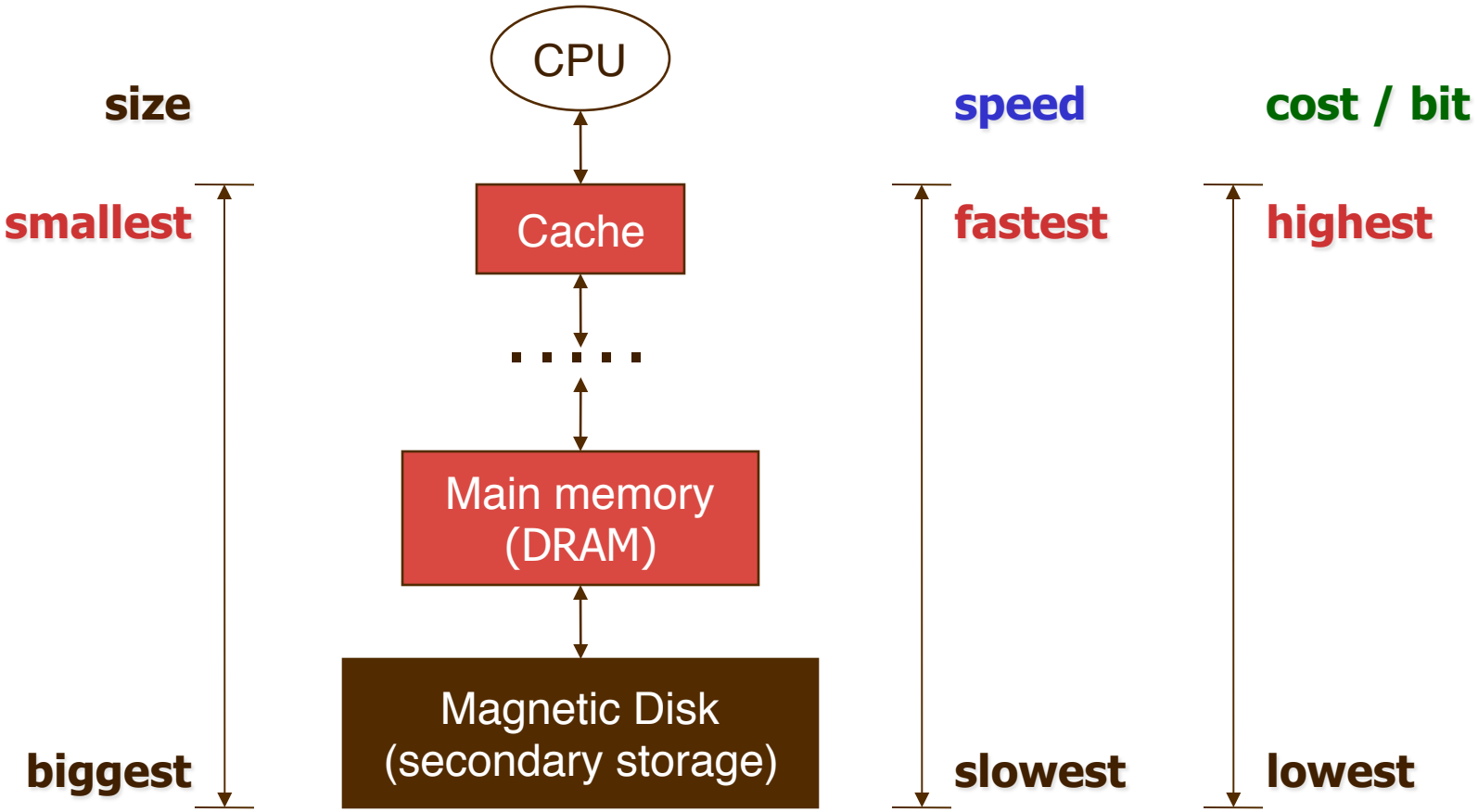
- **Temporal** and **spatial** locality can be observed in instructions & data

Example: locality during program execution



Memory Hierarchy

- A **memory hierarchy** consists of multiple levels of memory
- Users have the **illusion** of a memory
  - as large as the largest level, as fast as the fastest level
- A trade-off between performance and cost



Memory Hierarchy (2)

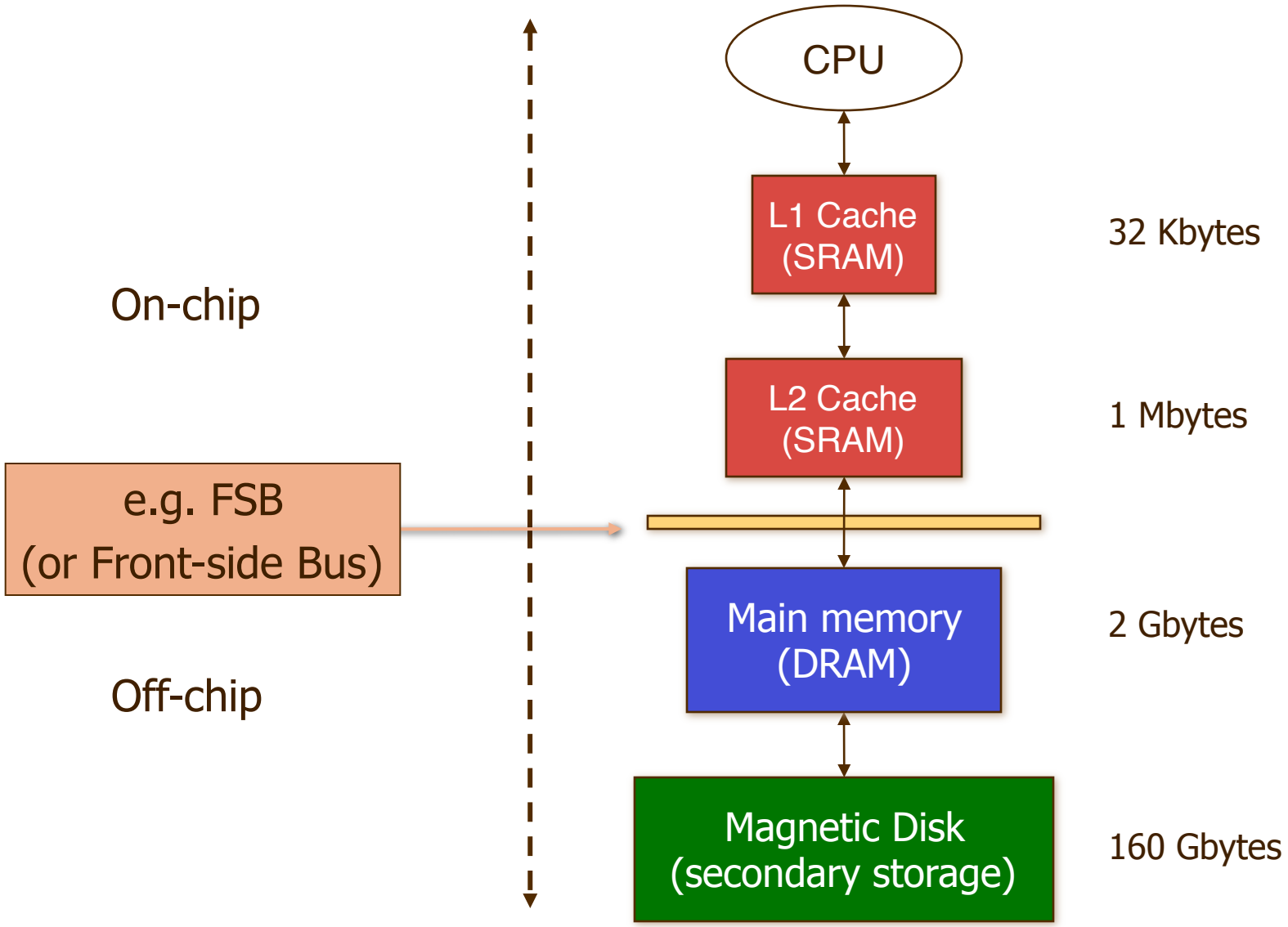
Memory Technology	Typical access time	$ per GB in 2008
SRAM	0.5-2.5 ns	$2000-$5,000
DRAM	50-70ns	$20-$75
Magnetic disk	5,000,000 – 20,000,000 ns	$0.20-$2

In memory hierarchy,

- Different levels of memory use different technologies
- Different technologies give different performance/cost trade-off (example will be given later)

Cache was the name chosen to represent the level of the memory hierarchy between the CPU and the main memory in the first commercial machine

Example with Two Levels of Caches



How Does Memory Hierarchy Operate?

Initially,

- Instructions and data are loaded into memory (DRAM) from disk

Upon first access,

- A copy of the referenced instruction or data item is kept in cache

In subsequent accesses,

- First, look for the requested item in the cache
  - If the item is in the cache, return the item to the CPU
  - If NOT in the cache, look it up in the memory
- We can say
  - If not found in this level, look it up at next level until found
  - Keep (or cache) a copy of the found item at this level after use

Why Memory Hierarchy Improves Performance?

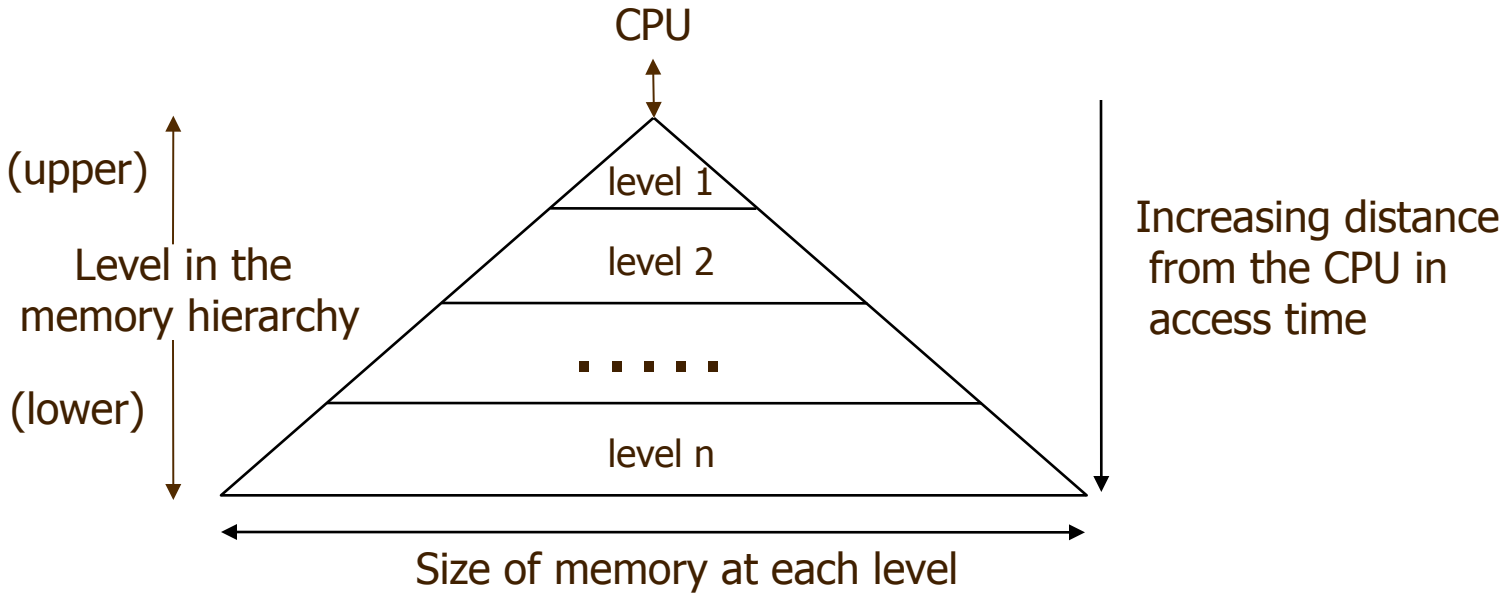
Memory hierarchy takes advantage of:

❶ **Temporal locality** by keeping more recently accessed data items closer to the processor

- Increase the chance that the item is found in shorter time

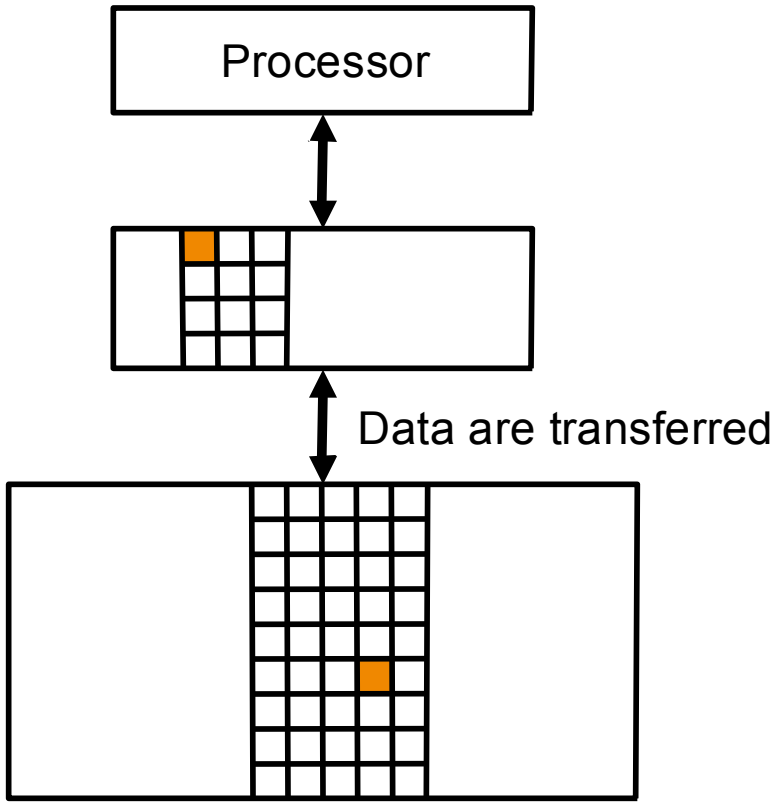
❷ **Spatial locality** by moving blocks consisting of multiple contiguous words in memory to upper levels of the hierarchy

- Increase the chances that the close-by item is found in shorter time



Data Transfer Between Levels

- A level closer to the processor is a subset of any level further away
- Data are copied in **blocks** (block size is usually 32 bytes or 64 bytes) between only two adjacent levels at a time
- **Hit**: when the data item requested by the processor appears in some block in the upper level
- **Miss**: when the data item requested by the processor is not in the upper level; data transfer occurs from the lower level to the upper level



Performance Measures

- **Hit rate** (or **hit ratio**):
  - Fraction of memory accesses found in the upper level
- **Miss rate**: 1 - hit rate

- **Cache hit time**:
  = time to determine miss or hit + time to access the cache

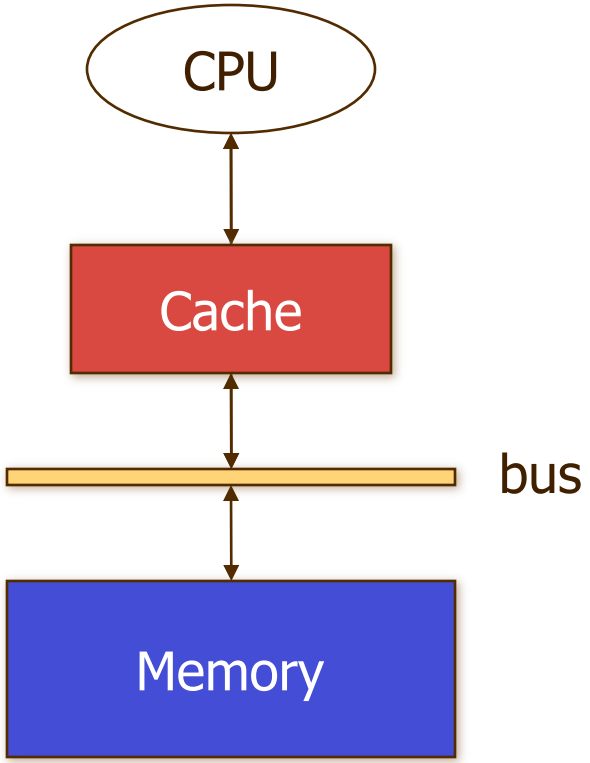
- **Cache miss penalty**:
  = time to bring a block from lower level to upper level

- **Hit time << Miss penalty**

Example with One Level of Cache

Example (assume everything else is ideal)

- 3-cycle cache access latency
- 100-cycle memory access latency
- Assuming 0-cycle for bus and cache lookup
- Data: 70% of the times in cache



Average memory latency

= 0.7 * 3 + 0.3 * (3 + 100)

= 3 + 0.3 * 100

= 33 cycles

Without cache, it is 100 cycles !!!

With caching, performance is better we are closer to the ideal case!

Example with Two Levels of Caches

Example

- 3-cycle L1 latency
- 12-cycle L2 latency
- 100-cycle memory latency
- Assuming 0-cycle for bus and cache lookup
- Data: 70% in L1
- Data: 60% in L2

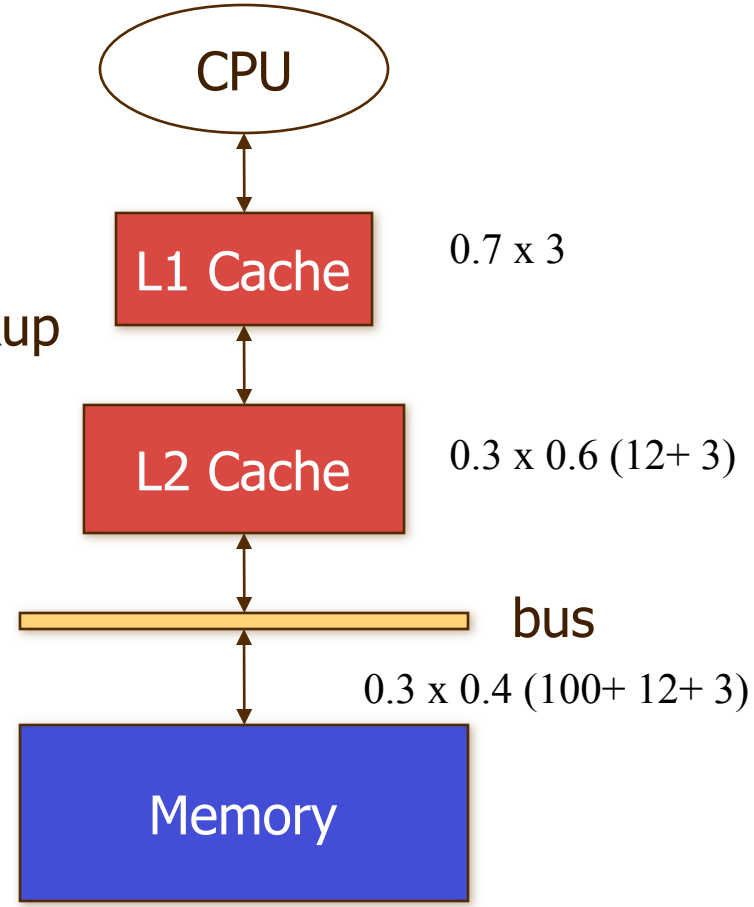
Average memory latency

= 3 + 0.3 * 12 + 0.3 * 0.4 * 100

= 3 + 3.6 + 12.0

= 18.6 cycles

(better than one level)



= 0.7 x 3 + 0.3 x 0.6 (12+3) + 0.3 x 0.4 (100+ 12+ 3)

Cache Performance

Average memory access latency = hit time + miss rate * miss penalty

For a configuration with a single level of cache, average memory latency

= hit time + miss ratio * miss penalty

= hit time + (1 - hit ratio) * miss penalty

For two levels of caches, average memory latency

= hit time_1 + miss ratio_1 * miss penalty_1

= hit time_1 + miss ratio_1 * (hit time_2 + miss ratio_2 * miss penalty_2)

= hit time_1 +

miss ratio_1 * hit time_2 +

miss ratio_1 * miss ratio_2 * miss penalty_2

The same idea can be extended to multiple levels of caches

Example

Suppose 1000 memory references

40 misses in the 1st-level cache and 20 misses in the 2nd-level cache

Hit $time_1$ = 3 cycles;

Hit $time_2$ = 12 cycles;

Memory latency = 100 cycles

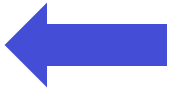
- Miss $rate_1$ = 40 / 1000 = 0.04 or 4%
- Miss $rate_2$ = 20 / 40 = 0.5 or 50%
- Miss $penalty_2$ = memory latency
- Average memory access latency

  = 3 + 0.04 * (12 + 0.5 * 100)

  = 3 + 0.04 * 12 + 0.04 * 0.5 * 100

  = 5.48 cycles

Cache



Issues to Consider

Block placement:

- Where is a block placed in the cache?

Block identification:

- How can a block be found if it is in the cache?

Block replacement:

- Upon miss, how the victim block in the cache is selected for replacement?

Write strategy:

- When a write occurs, is the information written only to the cache?

Implementation of Caches

Cache is organized as an array of cache blocks

Each memory location is mapped to ONE location in the cache

Cache block

- A minimum unit of information that can be present in cache or not

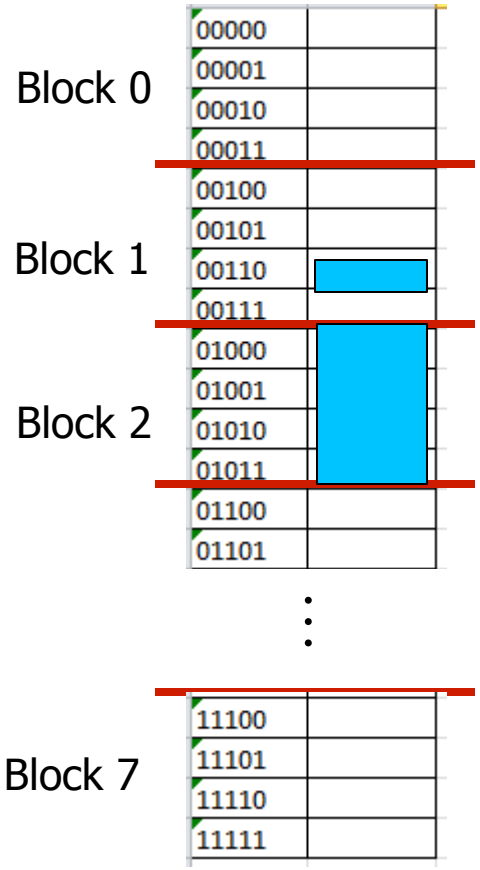
Cache block sizes

- Commonly used today: 32 bytes and 64 bytes

Three basic organizations

- **Direct-mapped** (one memory block to one possible cache block)
- **Set-associative** (one memory block to one set of possible cache blocks)
- **Fully-associative** (one memory block to all possible cache blocks)

Revisit Memory Address



00	Set 0
01	Set 1
10	
11	Set 3

Method 1:
cache_location = block_address MOD
number_of_blocks_in_cache

Method 2: if m cache sets, lower m bits of block address

01010

Block address

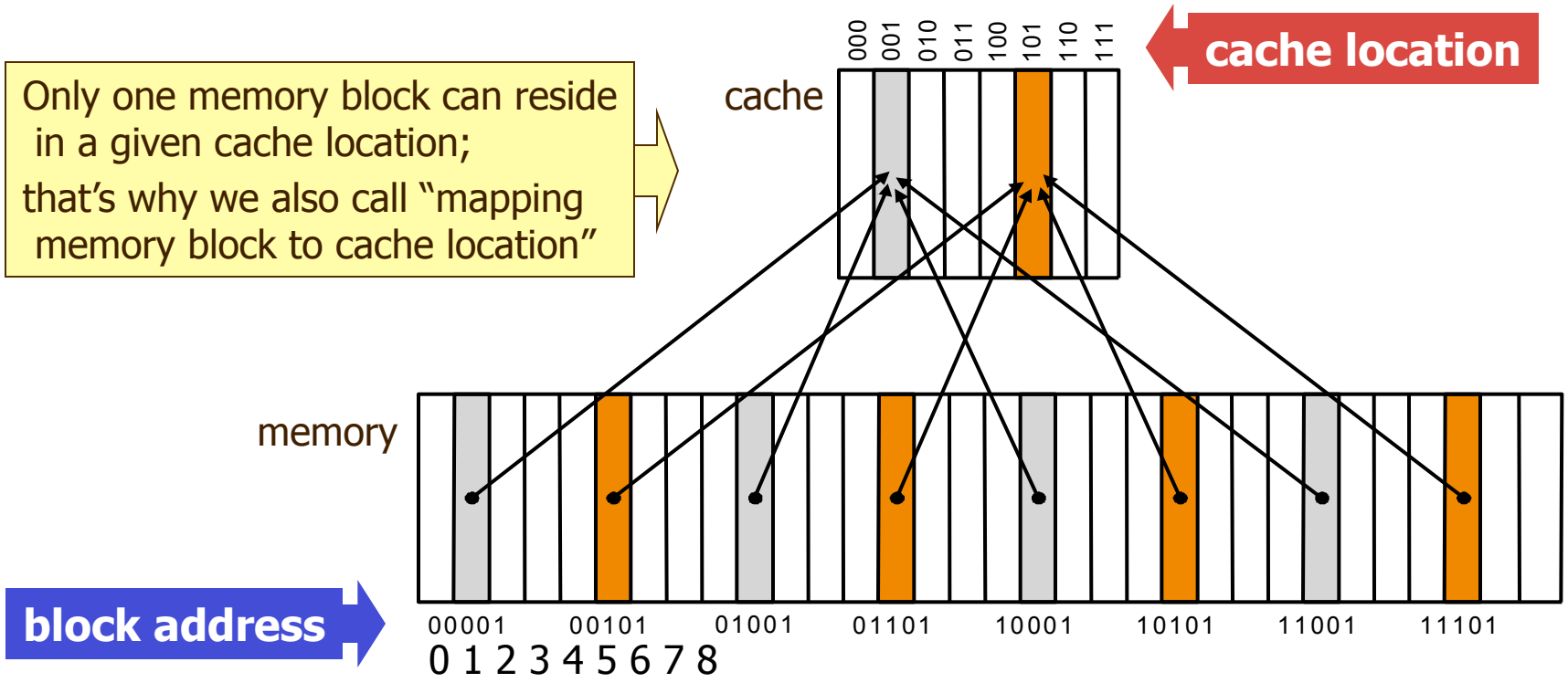
Offset within block

Direct Mapped: A Simple Block Placement Scheme

A common mapping strategy:

cache_location = block_address MOD number_of_blocks_in_cache

- If the number of cache blocks (N) is a power of 2; N = 2^m
- Long-latency MOD operation can be avoided
- Instead, cache_location = the low-order m bits of block address



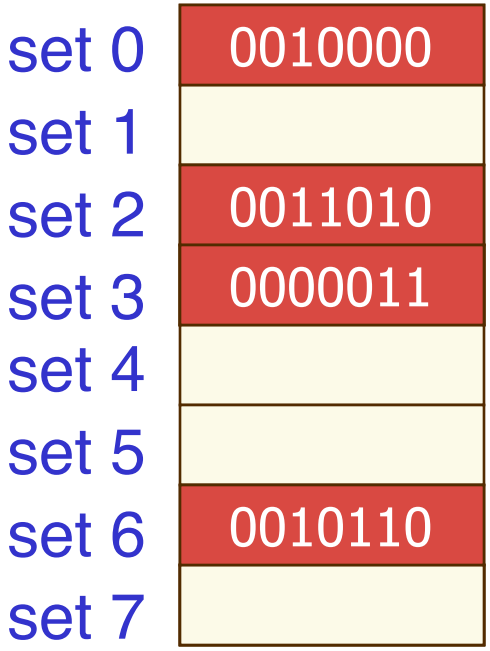
Example

Consider block address stream below (from left to right):

0010110_2, 0011010_2, 0010110_2, 0010110_2, 0010000_2, 0000011_2, 0010000_2, …

M **M** **H** **H** **M** **M** **H**

- Let's call cache location "set"
- Block address mapping to the frames:



8-entry cache

M: cache miss

H: cache hit

Mapping an Address to a Multiword Cache Block

Example: `lw $t0, 1200($zero)` , with a direct-mapped cache with 8 entries (or cache blocks) and a block size of 32 bytes

- What block number does byte address 1200 map to?

- First, find the block address, which equals to $\frac{\text{byte address}}{\text{bytes per block}}$
- i.e. block address for byte address 1200 = floor(1200/32) = 37
- The block address is the block containing all addresses between

$$\left\lfloor \frac{\text{byte address}}{\text{bytes per block}} \right\rfloor \times \text{bytes per block}$$ and

e.g. Block 0 contains byte 0 to byte 31
Block 1 contains byte 32 to byte 63

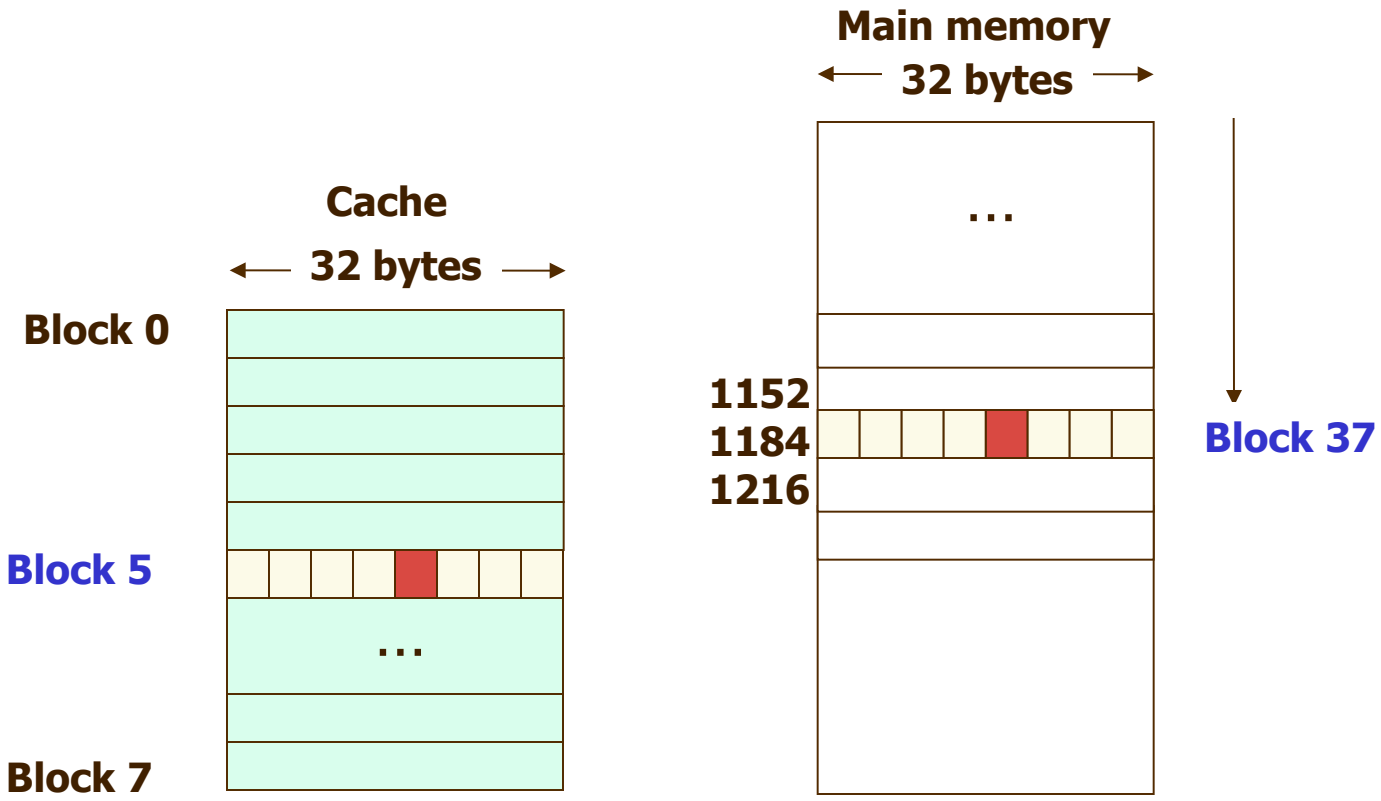
$$\left\lfloor \frac{\text{byte address}}{\text{bytes per block}} \right\rfloor \times \text{bytes per block} + (\text{bytes per block} - 1)$$

- Next, block location = (block address) MOD (number of cache blocks)
- i.e. map to cache block number (37 MOD 8) = 5

Explanation

Example: `lw $t0, 1200($zero)`

- The memory address generated by CPU upon execution = 1200
- 1200 will be first sent to the cache to look for the data
- By the mapping method discussed, the data block will be in entry 5



Example: Accessing a Cache

Consider a direct-mapped cache with

- 8 cache frames and a block size of 32 bytes

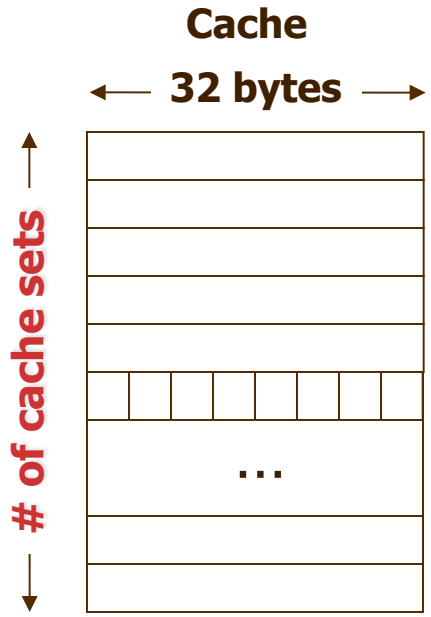
Memory (byte) address generated by CPU	Block address	Hit or miss in cache	Assigned cache block (where found or placed)
$0010\ 1100\ 0010_2$	$0010\ 110_2$	**Miss**	$(0010110_2 \bmod 8) = 110_2$
$0011\ 0100\ 0000_2$	$0011\ 010_2$	**Miss**	$(0011010_2 \bmod 8) = 010_2$
$0010\ 1100\ 0100_2$	$0010\ 110_2$	**Hit**	$(0010110_2 \bmod 8) = 110_2$
$0010\ 1100\ 0010_2$	$0010\ 110_2$	**Hit**	$(0010110_2 \bmod 8) = 110_2$
$0010\ 0000\ 1000_2$	$0010\ 000_2$	**Miss**	$(0010000_2 \bmod 8) = 000_2$
$0000\ 0110\ 0000_2$	$0000\ 011_2$	**Miss**	$(0000011_2 \bmod 8) = 011_2$
$0010\ 0001\ 0000_2$	$0010\ 000_2$	**Hit**	$(0010000_2 \bmod 8) = 000_2$
$0010\ 0100\ 0001_2$	$0010\ 010_2$	**Miss**	$(0010010_2 \bmod 8) = 010_2$

- **Miss rate = (# of misses) / (# total memory accesses)** = 5 / 8

Deciding Number of Sets in Direct-Mapped Cache

Example: Direct-mapped (DM) cache

- Cache size = 16 Kbytes
- Cache block size = 32 bytes
- N = # of blocks in cache
- # of cache sets = ? (in DM, N = Sets)



➢ # of cache sets = 16K / 32

$= 2^{14} / 2^{5}$

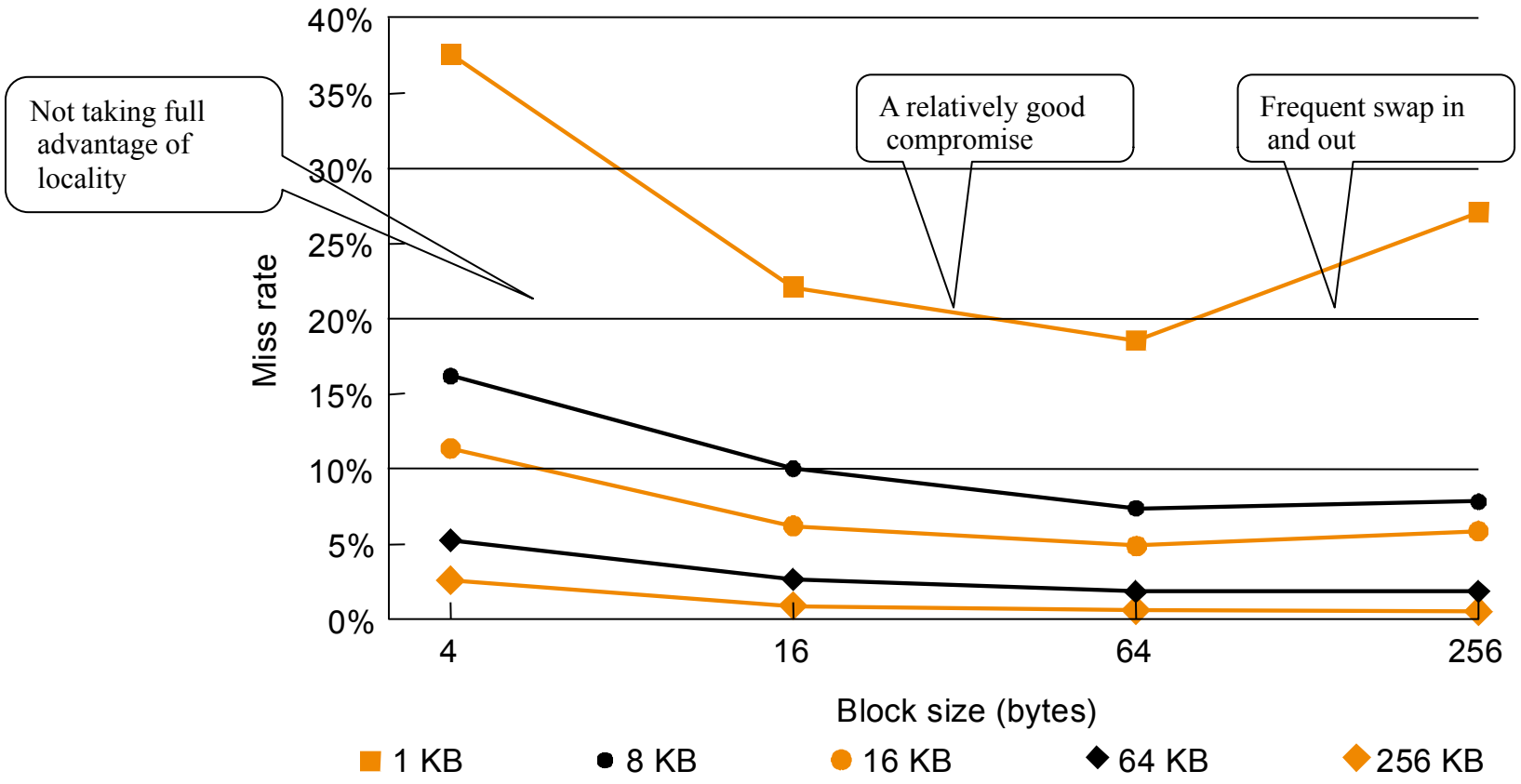
$= 2^{9}$

$= 512$

If the answer is 2^{m}, it implies that we need rightmost **m** bits of the block address to form the index (cache location) into the cache; in this example, we need **9** bits

Miss Rate vs. Block Size

- Bigger cache block size, less number of cache blocks; and vice versa
  - Given cache size is fixed, varying cache block size changes miss rate



Disadvantage of DM Cache

Blocks mapped to same cache frame can't be present simultaneously

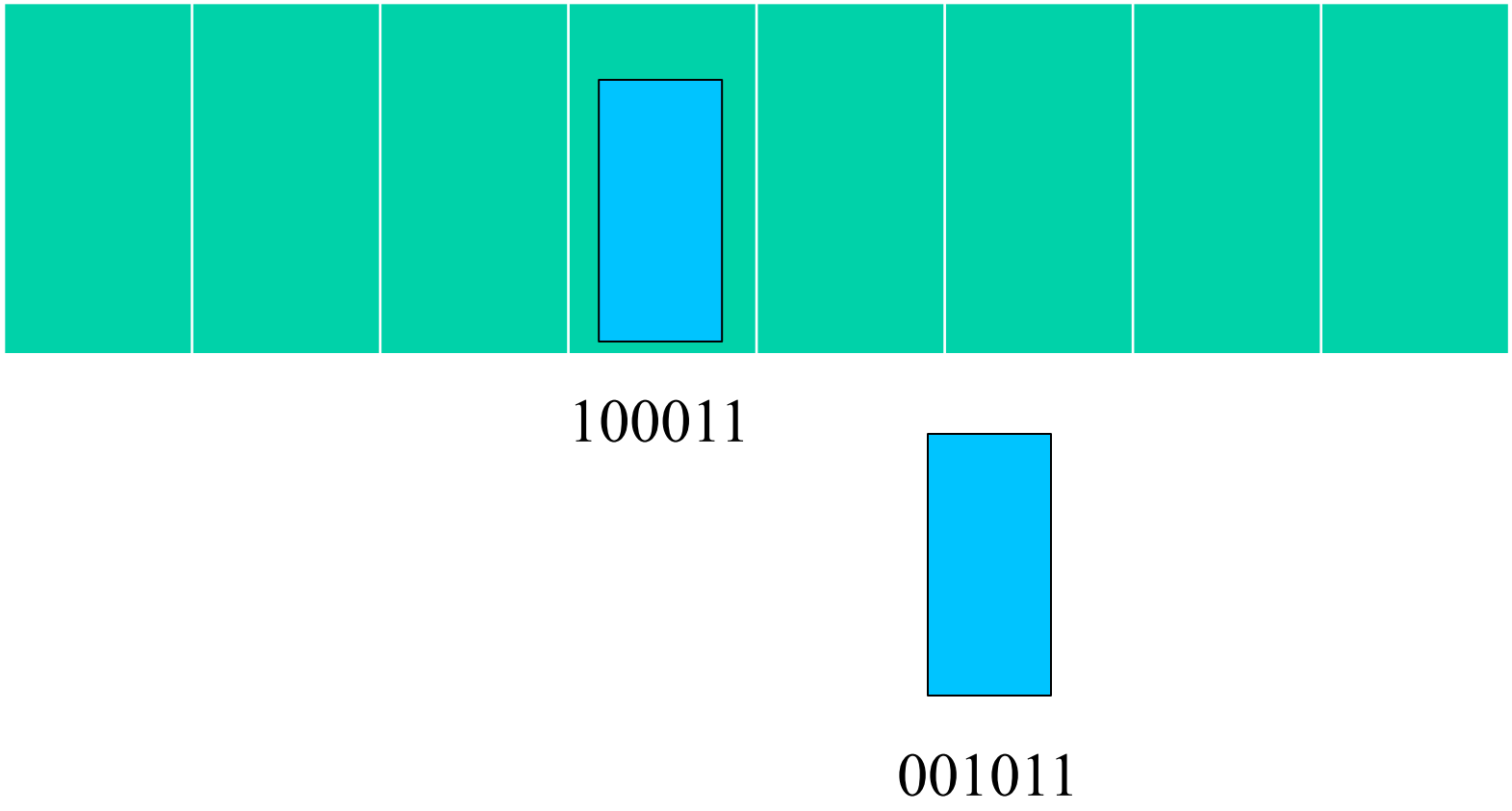
Example:

- # of cache sets = 8
- Consider block access sequence: 100011_2, 001011_2, 100011_2
- Block 100011_2 & 001011_2 are mapped to the same cache set 011_2
- Arrival of block 001011_2 will kick (i.e. replace) 100011_2 out
  - i.e. both blocks can't be present simultaneously
  - Subsequent access to 100011_2 will **not hit** but **miss**
  - Less performance improvement with such **cache conflict**

Question:

- Can we alleviate this conflict problem to increase the cache hit rate?

DM Disadvantage



Other Block Placement Schemes

Fully associative (FA)

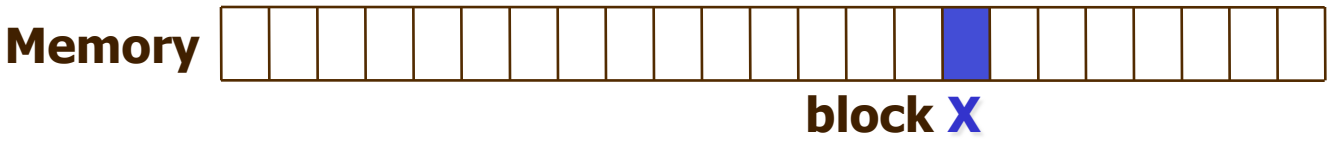
- Each block can be placed **anywhere** in the cache
- **Advantage:**
  - No cache conflict ⇒ better cache hit rate than **direct mapped**
  - But, still see misses due to size (capacity miss)
- **Disadvantage**:
  - Costly (hardware and time) to search for a block in the cache

Set associative (SA)

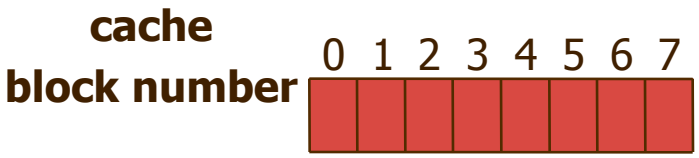
- Each block can be placed in a **certain number** of cache locations
- A good **compromise** between direct mapped & fully associative
  - In terms of cost and hit rate

Fully Associative vs. Set Associative

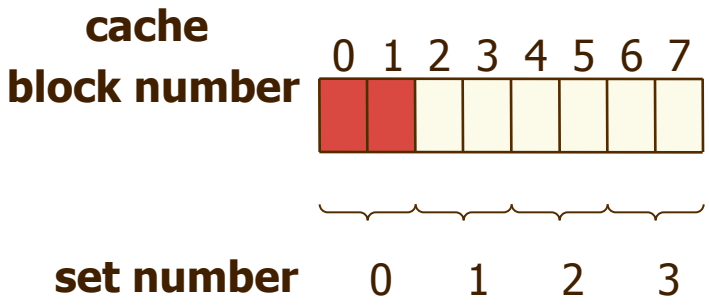
- Highlighted locations are placement candidates for memory block **X**



Fully associative cache



2-way set associative cache

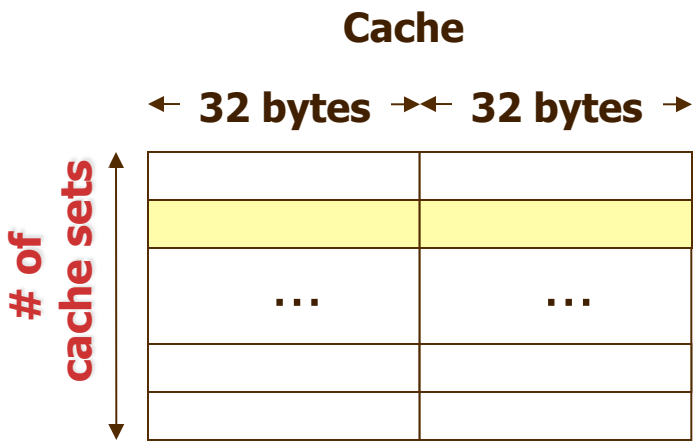


- Assume both FA and SA have same number of cache blocks

Deciding Number of Sets in Set-Associate Cache

Example: 2-way set-associative (SA) cache

- Associativity = 2
- Cache size = 16Kbytes
- Cache block size = 32bytes
- **N** = # of blocks in cache
- **# of cache sets** = ? (in SA, N ≠ Sets)



- **N** = 16K / 32 = 512

- **# of cache sets** = N / 2
  = 256

In general, for a m-way set-associative cache,

- # of cache sets = cache size / cache block size / m

Deciding Number of Sets in Fully-Associate Cache

Fully-associative (FA) cache is a special case of set-associative cache

- i.e. there is only **ONE set** in FA since a block can be placed anywhere

Example:

- Cache size = 16Kbytes
- Cache block size = 32bytes
- **N** = # of blocks in cache = 16K / 32 = 512 (still the same calculation)
- **# of cache sets** = **1**

Placement for N-way Set-Associative Cache

An **N-way set associative cache**

- Consists of a number of sets, each of which consists of **N** blocks

Mapping strategy:

- number_of_sets_in_cache = cache size / cache block size / **N**
- cache_location = block_address MOD number_of_sets_in_cache

Placement of the memory block

- Can be in any "way" within the set
- e.g. if 4-way, there are 4 feasible locations to cache the block (how to choose the location to place it will be answered later)

Special cases:

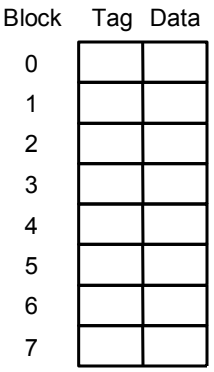
- A direct mapped cache as a 1-way set associative cache
- A fully associative cache with M blocks an M-way set associative

Possible Associativity Structures

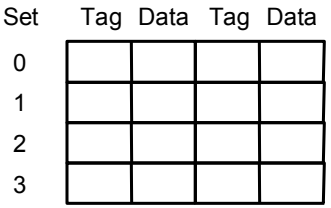
Example:

- Cache Size = 256 bytes
  = 64 words
- Block Size = 32 bytes
  = 8 words
- Number of blocks = 8

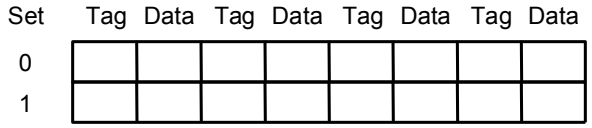
One-way set associative (direct mapped)



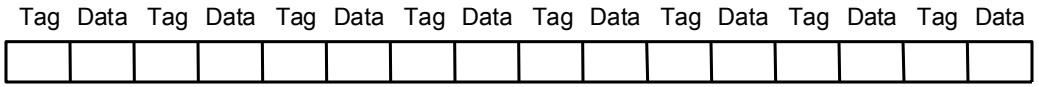
Two-way set associative



Four-way set associative



Eight-way set associative (fully associative)

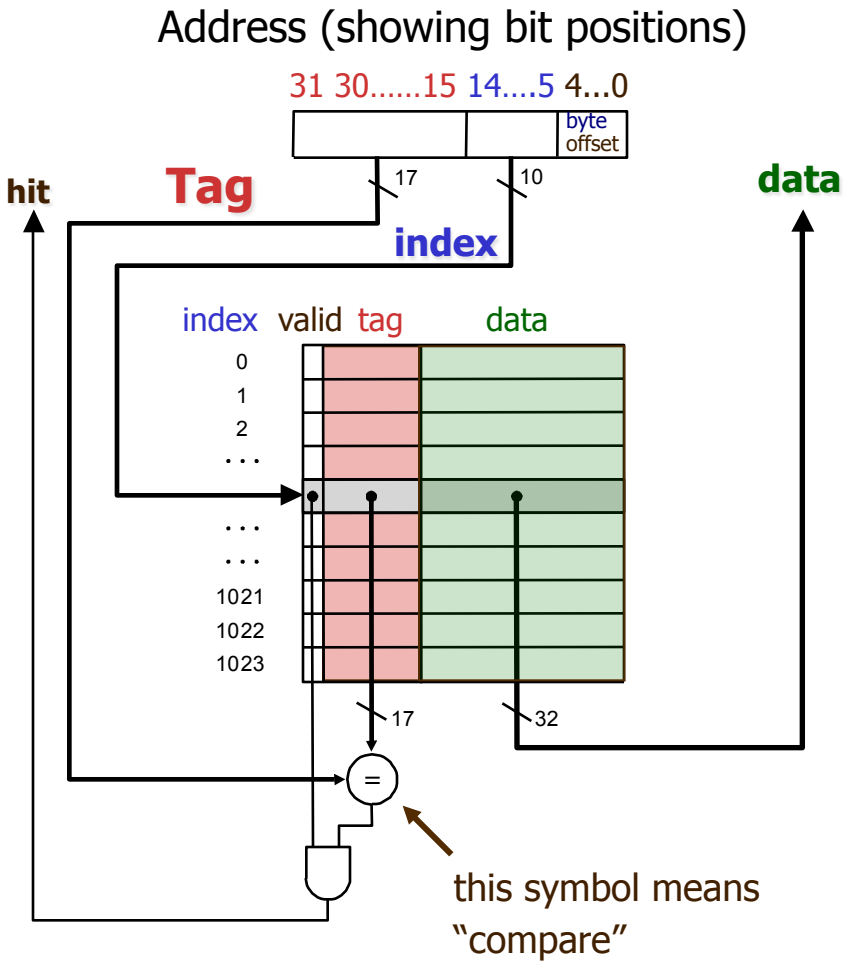


- **Increase** in degree of associativity ⇒
  - **Advantage: Decrease** in miss rate
  - **Disadvantage: Increase** in hit time (due to longer search time)

Block Identification in DM

- Assume a DM with 1024 cache sets
- Assume the block size is 32 bytes

- Each cache location can contain a block from a number of different memory locations
- Question: how do we know which block is actually in the location, i.e. how to tell hit or miss?
  - A **tag** is used to store the address information
  - The tag needs only to contain those high-order bits that are not used as an index into the cache
  - A **valid bit** is needed to indicate whether a cache block contains valid information



Example: Bits in a Cache

- A DM cache with 16KB of data and 8-word (2^5 bytes) blocks
- Assume a 32-bit address
- How many total bits are required?

16KB = **4K** words = 2^{12} words

Block size of **8** words ➔ 2^9 blocks

Each block has **8 x 32 = 256** bits of data

A tag has **32 – 9 – 5 = 18** bits

And, **1** valid bit

Total bits per entry = **(256 + 18 + 1)**

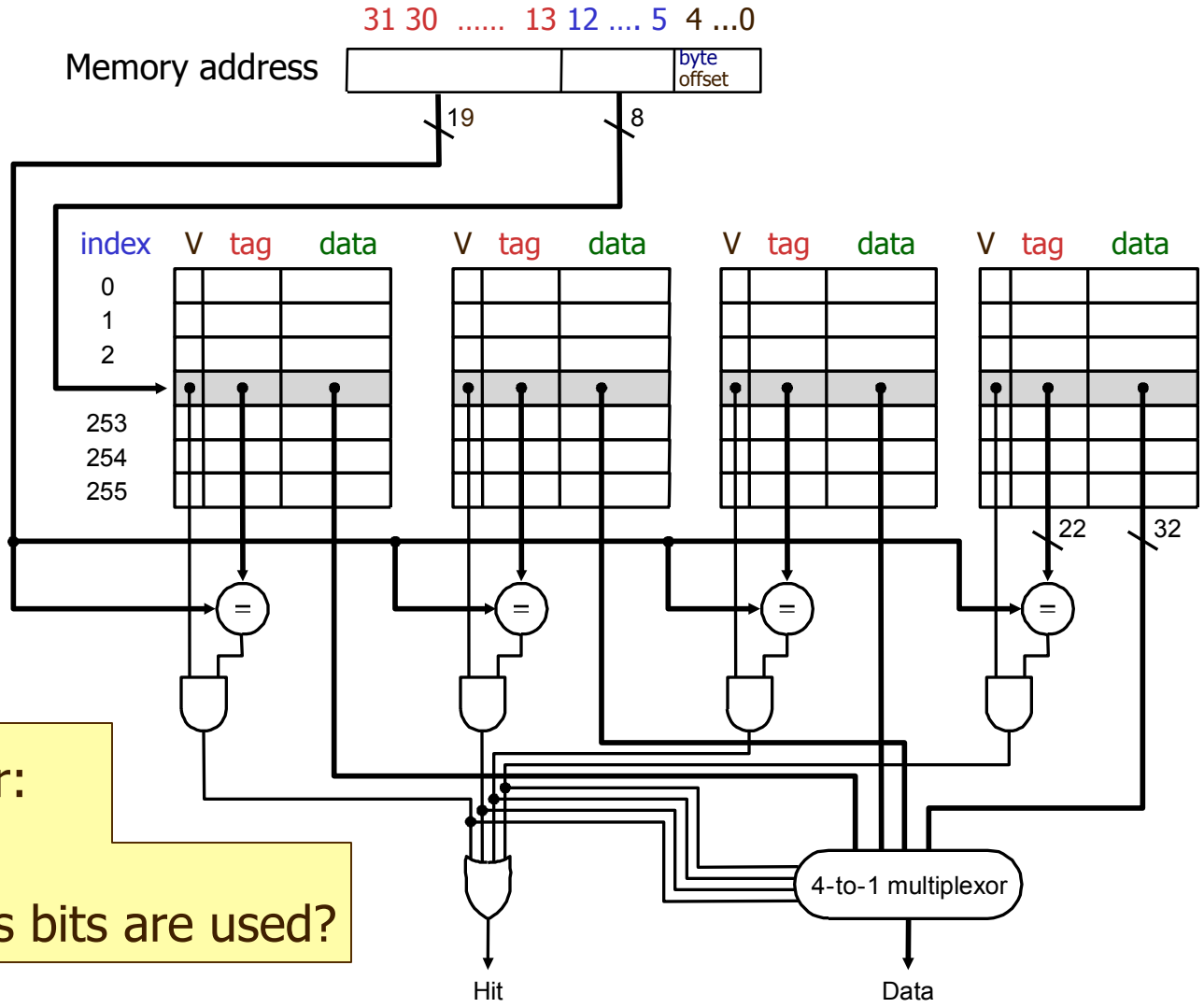
Total cache size

2^9 **x (256 + 18 + 1) = 2^9 x 275 = 140800 = 137.5** Kbits

- How much overhead is needed?

Block Identification in N-way Set Associative Cache

- Parallel lookup for the requested address in the cache set



Questions to answer:

- Which cache set?
- i.e. which address bits are used?

Block Identification in N-way Set Associative Cache

Recall the N-way SA's **mapping strategy**:

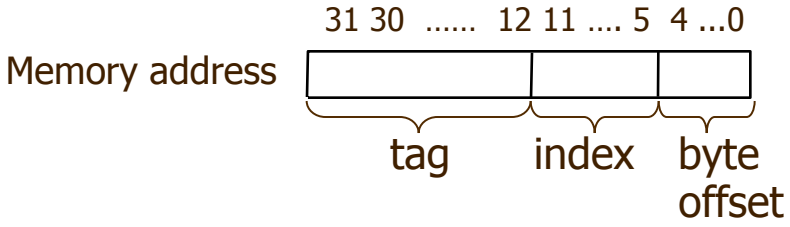
- number_of_sets_in_cache = cache size / cache block size / N
- cache_location = block_address MOD number_of_sets_in_cache

If number_of_sets_in_cache = 2^m,

- Use next m address bits after the byte offset to serve as the index

Example:

- 16KB cache, 4 ways, 32-byte cache block
  - number_of_sets_in_cache = 16K / 32 / 4 = 128 = 2^7
  - Byte offset = 5 (because 32 = 2^5)
  - So, bit 0~4 are for byte offset, bit 5~11 are for index, the rest is tag



Handling Cache Misses

Upon a memory access by CPU, a request is sent to the first level cache

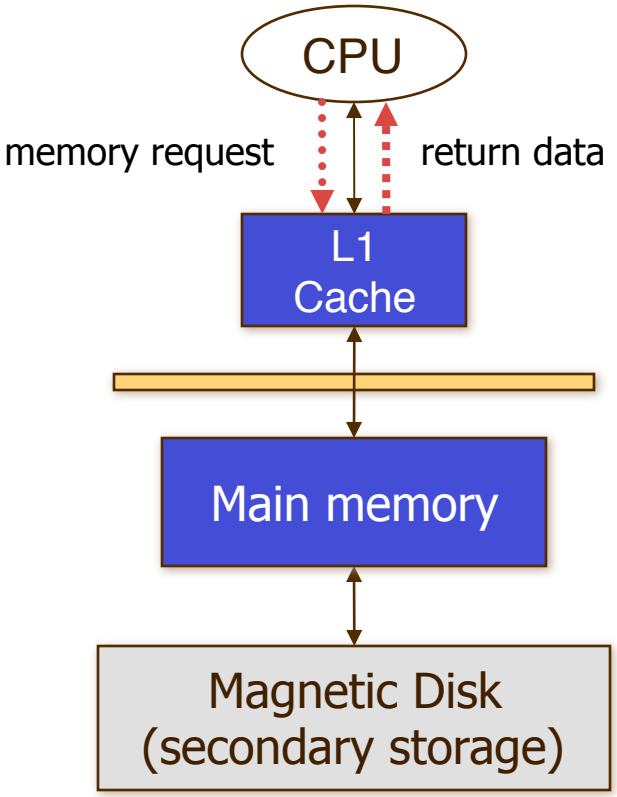
If the cache reports a hit

- CPU continues using the data from cache as if nothing had happened

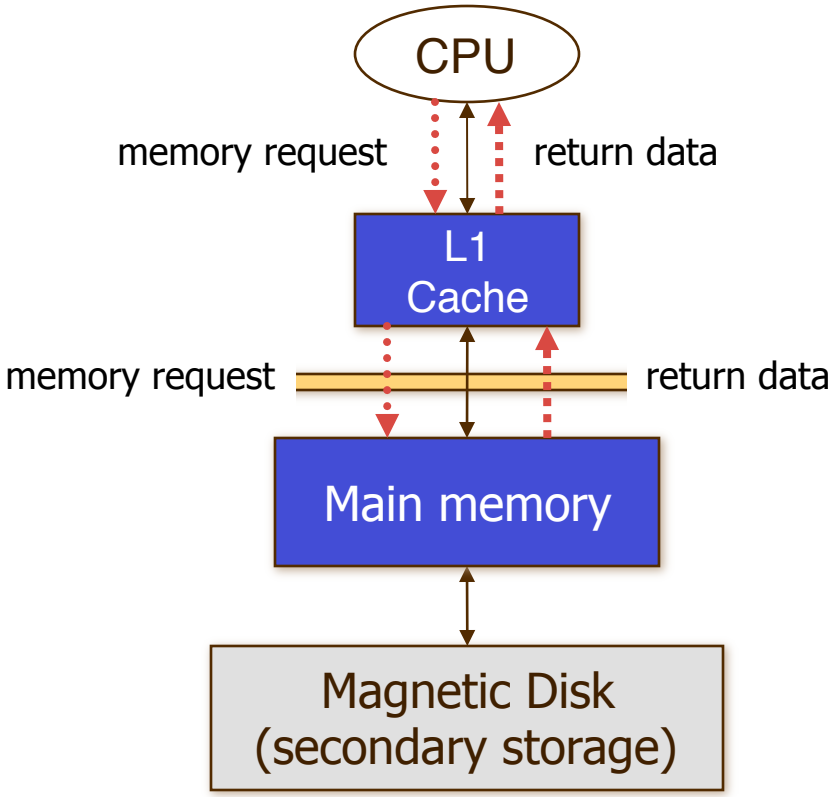
If the cache reposts a miss, some extra work is needed

- A separate controller initiates a request to refill the cache
  - This request is sent to the next level cache or memory
    (Keep going to next level until data are found)
- During the process of refilling, CPU is stalled
  - Stall ⇒ entire CPU stops operating until the 1st-level cache is filled
  - Unlike interrupts, stall does not need saving the state of all registers

Flow of Handling Cache Accesses



Cache hit

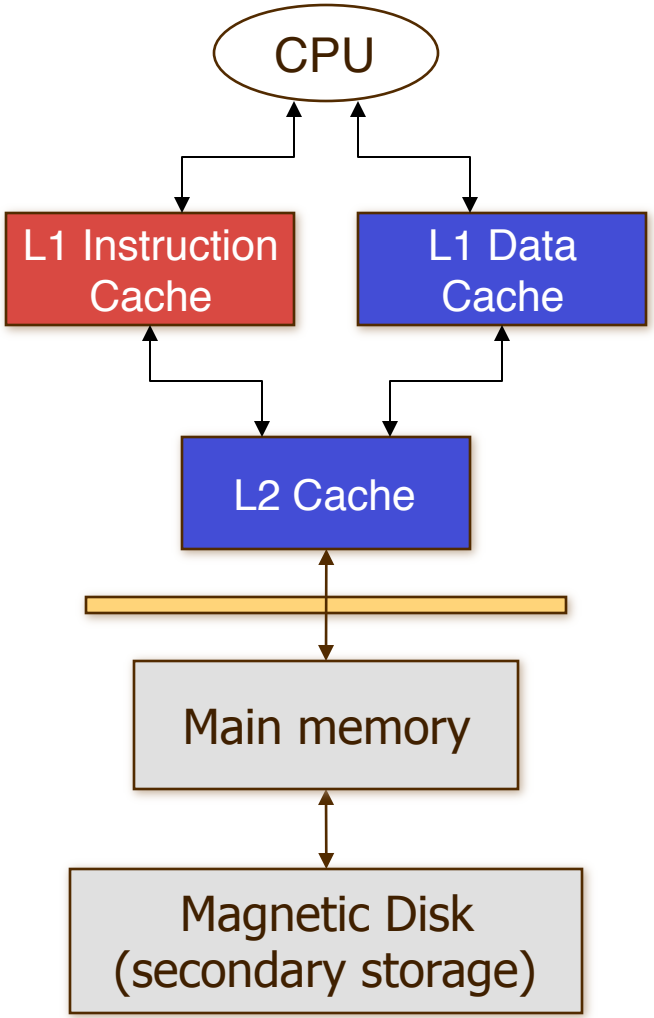


Cache Miss

Caching the Instruction

Instructions are also stored in memory

- Upon execution of a program, CPU fetches instructions from memory
- To avoid long instruction fetch, same caching idea can be applied to instructions
  - An example of instruction cache in a two-level cache hierarchy is shown in the figure

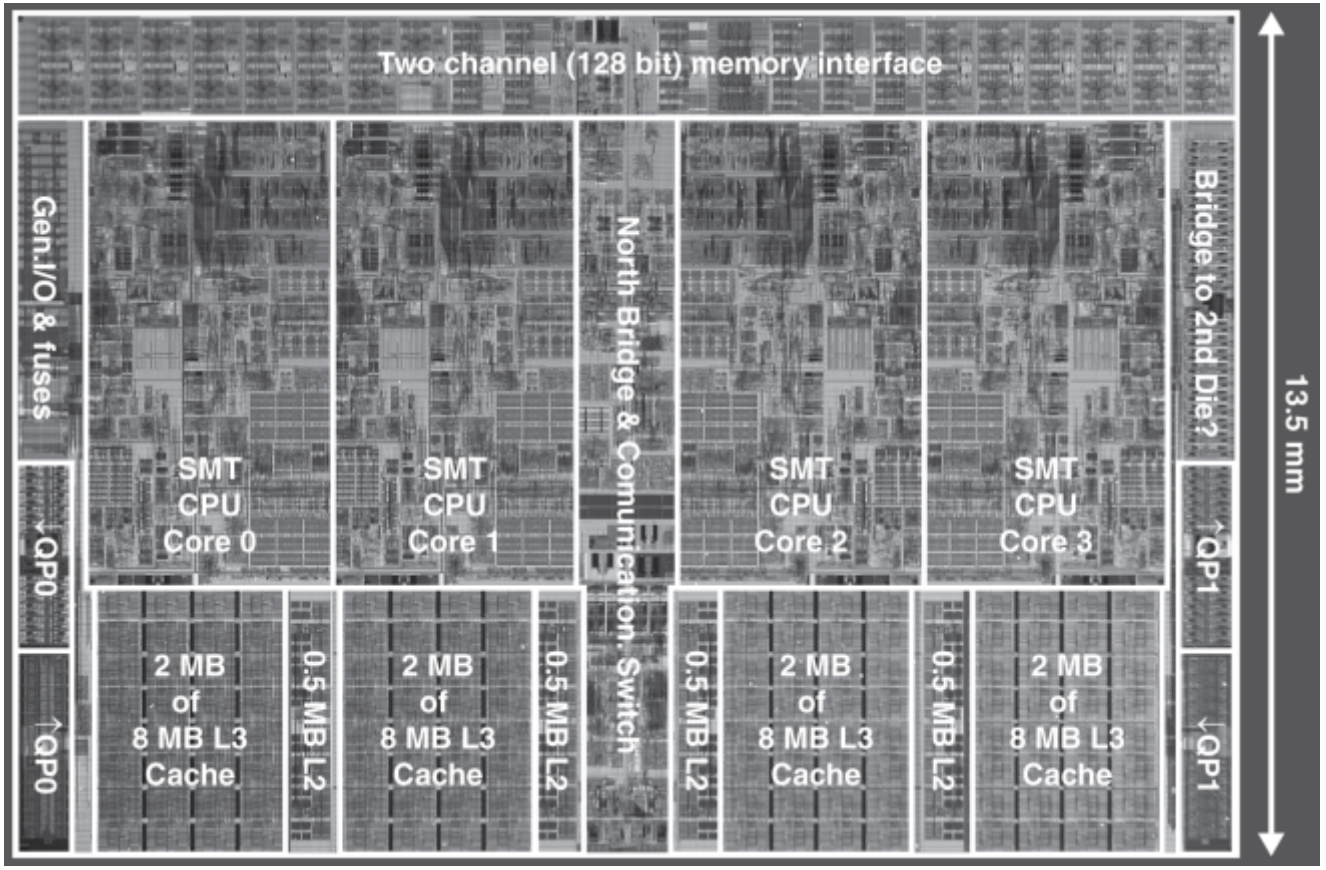


Example: Steps on an Instruction Cache Miss

1. Send the PC value to the memory
2. Instruct main memory to perform a read and wait for completion
3. Write the (instruction) cache entry, i.e.
   - ❑ Put the data from memory in the data portion of the entry
   - ❑ Write the upper bits of the PC into the tag field
   - ❑ Turn the valid bit on
4. Restart the instruction execution at the first step
   - ➢ Which will re-fetch the instruction, this time finding it in the cache

- ❑ Control of the cache on an instruction access is essentially identical:
- ➢ On a miss, simply stall the CPU until the memory responds with instr.

Intel Nehalem 4-core processor



Per core: 32KB L1 I-cache, 32KB L1 D-cache, 512KB L2 cache

Block Replacement

- When a cache miss occurs, we must decide which block to replace
- Why block replacement is needed?

Block replacement for

- Direct mapped: only one candidate (trivial case)
- Fully associative: all blocks are candidates
- Set associative: only blocks within a particular set are candidates

Two primary replacement policies for associative caches

1. **Random**:
   - Candidate blocks are randomly selected to spread allocation uniformly
2. **Least recently used** (**LRU**):
   - The candidate is the block that has not been used for the longest time
   - Costly to implement for a degree of associativity higher than 2 or 4

Example of LRU Replacement Policy

Assume the cache is 4-way set-associative

Consider block address stream below (from left to right):

0011010_2, 0010010_2, 0110010_2, 1000010_2, 0001010_2, 0010010_2, 0100010_2

- All these block addresses mapped to same set "010"
- Question: which blocks remain in the set at the end?

- Answer: (just look at the tag)

		0011	0010	0110	1000	0001	0010	0100
MRU →	—	0011	0010	0110	1000	0001	0010	0100
	—	—	0011	0010	0110	1000	0001	0010
	—	—	—	0011	0010	0110	1000	0001
LRU →	—	—	—	—	0011	0010	0110	1000
		miss	miss	miss	miss	miss	hit	miss
						replace 0011		replace 0110

LRU Replacement Policy

Upon miss:

- Replace the victim from the LRU position with miss address
- Move the miss address to the MRU position
- Heuristic: the LRU is not referenced for a long time, good candidate

Upon hit:

- Move the hit address to the MRU position
- Then, pack the rest
- Heuristic: the one just got hit should be the last one to be replaced

Dealing with Dirty Data

Definition

- If the content of a cache entry is modified, it is considered as dirty

If a line is dirty, what should we do upon block replacement?

- It is a question about how we update the memory

Two options:

1. Write-back
2. Write-through

Write-Back Strategy

Upon write access by CPU,

- The information is written only to the block in the cache, not memory

When the block becomes the candidate of replacement

- The dirty block is written back to the main memory

Advantages:

- CPU can write individual words at the rate of the cache (not memory)
- Multiple writes to a block are merged into one write to main memory
  - Since the entire block is written, system can make effective use of a high bandwidth transfer

★ As CPU speed increases at a rate faster than DRAM-based memory, more and more caches use the write-back strategy

Write-Through Strategy

Upon all memory accesses,

- The information is written to both the block in the cache and memory (memory is always up-to-date)

Advantages:

- Handling of misses is simpler and cheaper
  - Because they do not require a block to be written back to memory
- Write-through is easier to implement than write-back

Disdvantages:

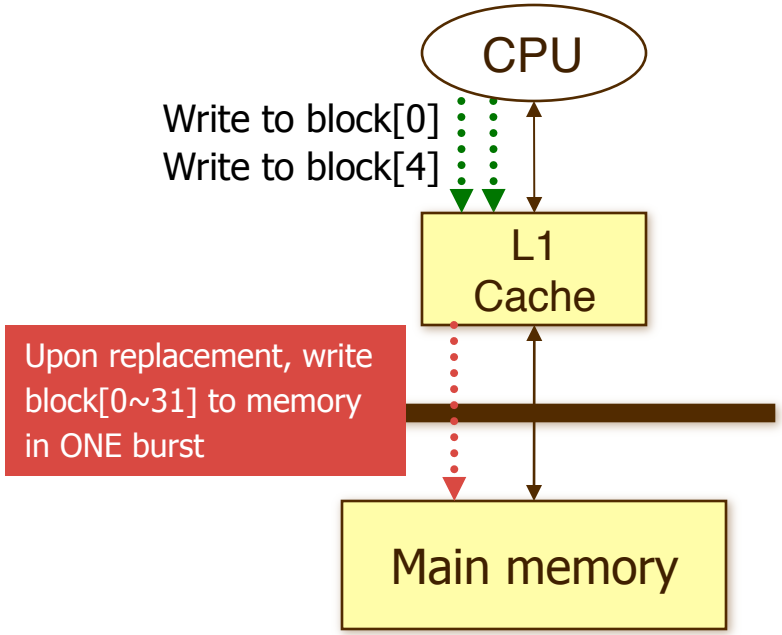
- Multiple writes to the same location consume memory bandwidth

⇒ Waste of bandwidth as compared to write-back strategy

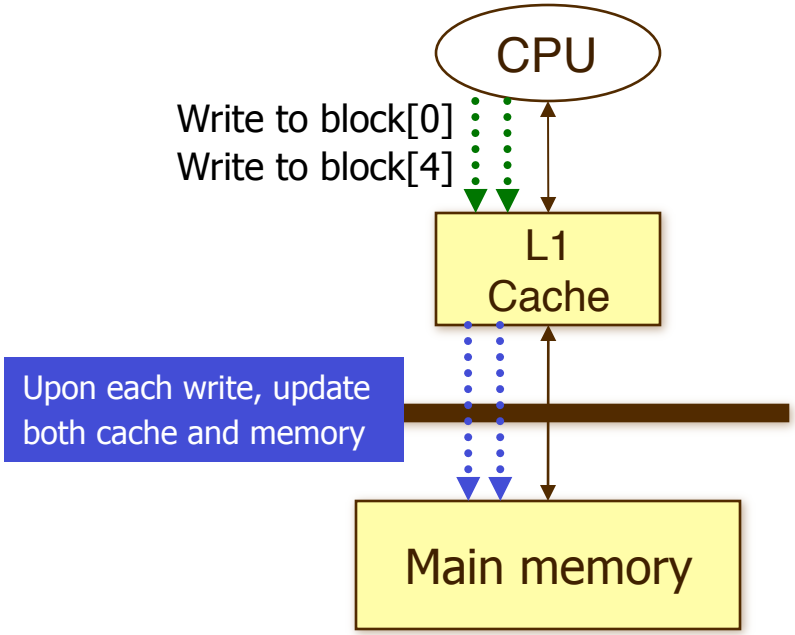
⇒ Potentially slow down the reads to memory

Notes: A write buffer is needed for a high-speed system if write-through strategy is used
A write buffer is a queue to hold data while data are waiting to be written to memory

Write-Back vs Write-Through



Write-Back



Write-Through