

Cache Memory

Introduction and Types

Introduction

- Von Neumann Model → memory as stored program
- Memory → linear array of locations addressed from $0 \rightarrow 2^n$, n =address bus size.
- Memory hierarchy system → various type of memories
- rationale?

To keep pace with CPU Speed, variations occur.

Cache memory

High speed and small amount of memory acting as buffer for frequently accessed data.

Types of memory

Two main types

1. RAM/ Random Access Memory

Read/Write memory/main memory/primary memory

program instructions + data → during program execution.

volatile → Loss of information on power off.

2 types of chips making RAM

SRAM vs. DRAM

A. DRAM

Capacitors leaking electricity → requires recharge every few milliseconds

B. SRAM

holds contents as long as there is power, D- Flip flops

2. ROM

- Read only information → must for system operation (program to boot the computer)
- Non volatile → data retention
- Used also in embedded systems (where programs do not need to change)
e.g. Laser printers stores their fonts in ROM

Memory Hierarchy

- Different memories with different speed and cost.
- Best combination to gain best performance at reasonable cost → creating memory hierarchy in system
- Following memories combined
 - Registers,
 - Cache,
 - Main memory,
 - Secondary memory etc

Memory Hierarchy

- Registers

Storage locations on processor

- Cache

High speed memory where data from frequently used main memory locations is temporarily stored

Connected to larger

- Main memory

medium speed memory complimented by very large

- Secondary memory

hard disks drives containing data not directly accessible by CPU

Hard disk drives can be magnetic or solid state (flash memory)

Memory Hierarchy insights

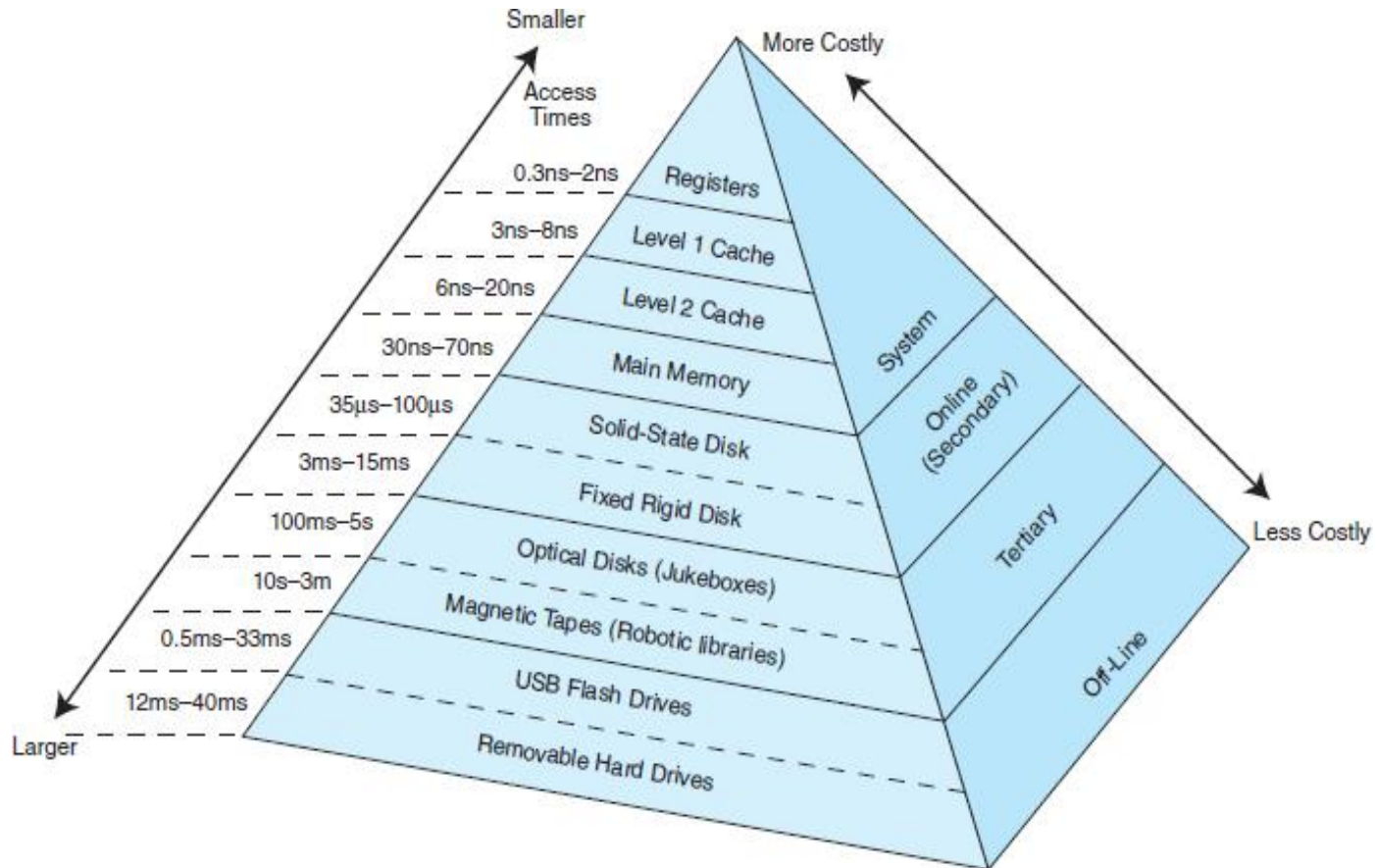
Pyramid → sizes indication

More closer to top → smaller, best performance → high cost per bit

Exception → offline faster than tertiary (USB flash drives)

Register access → one clock cycle

USB + SSD → Same access time but USB slow due to interface



Memory Hierarchy insights

Our interest in

- Registers
- Cache memory
- Main memory
- virtual memory

no system memory but part of hard disk acting as extension to main memory

USB flash drives and remove-bale SSD can also be used as an augmentation to virtual memory (acting as disk cache) due to fast access speed by some software like Ready Boost

How processor make data/instruction requests in memory hierarchies?

- Request goes to memory closer to it (faster)

cache memory

- Data found, loaded to CPU


- Else

Request goes to next level

Data found, whole **block** in which data resides is transferred to previous level

(E.g. Accessing of data X will brought X+1, X+2,... etc as well)

ELSE

 Request goes to next level and so on....

why its is effective → Locality of reference

Terminology introduced due to memory hierarchy

- A *hit* is when data is found at a given memory level.
- A *miss* is when it is not found.
- The *hit rate* is the percentage of time data is found at a given memory level.
- The *miss rate* is the percentage of time it is not.
- Miss rate = $1 - \text{hit rate}$.
- The *hit time* is the time required to access data at a given memory level.
- The *miss penalty* is the time required to process a miss, including the time that it takes to replace a block of memory plus the time it takes to deliver the data to the processor.

Locality of Reference

→ program property when executing

- If no branches then EIP automatically increments
→ if instruction X is accessed, $X+1$, $X+2$ will also be accessed
- Lets group all references

Will you see pattern ?

sequential access → example of locality of reference

This property is exploited in memory hierarchies when we access whole block from next level

Forms of locality

- Three forms
- **Temporal locality**—Recently accessed items tend to be accessed again in the near future.
- **Spatial locality**—Accesses tend to be clustered in the address space (for example, as in arrays or loops).
- **Sequential locality**—Instructions tend to be accessed sequentially.

Help us exploiting majority of memory accesses from small and fast memory

Cache memory

- Wait states → processor speed is much faster than slow memory.
- Can we access instructions/data execution in near future → from fast memory

Cache memory is there to help

hold data that has been accessed + that might be accessed
→ keep frequently used data in cache

Does computer knows which data to be accessed in future
No? Yes?

Locality principle application

where to brought the new block

depends upon

cache mapping policy + cache size

Data access from cache

- Cache is CAM (Not like RAM)
- CPU generates main memory address for any instruction's access.
- Where to see in cache?
 - Mapping scheme maps address to cache location
- Address is divided into 2/3 fields
 - Helps determine where instruction is copied in cache/where to find in cache if already copied

How data is found/copied?

- Main memory + cache → divided into equal sized blocks.
- Address divided into fields

One field (**block number**) tells where instruction is residing(cache hit)/should reside(cache miss) → block finding

Next to see that block is valid/not (valid bit associated with cache block = 0/1)

Other field tells that if valid bit=1, the block is same where our instruction exists (**tag bits**)

Last field (**offset number**) takes us to exact instruction inside block

Mapping schemes

determines cache types

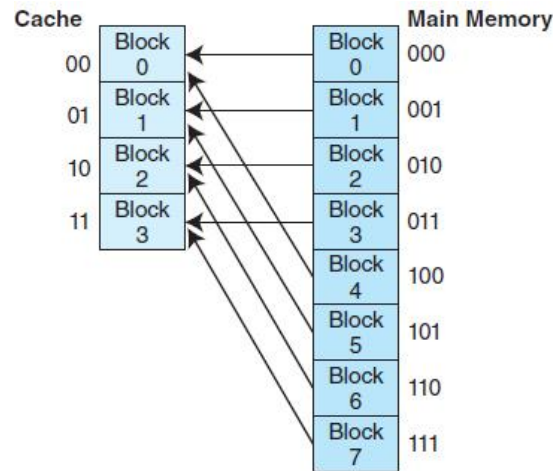
- Three types
 - Direct Mapped
 - Fully Associative Mapped
 - N-way Set Associative mapped

Quiz 2

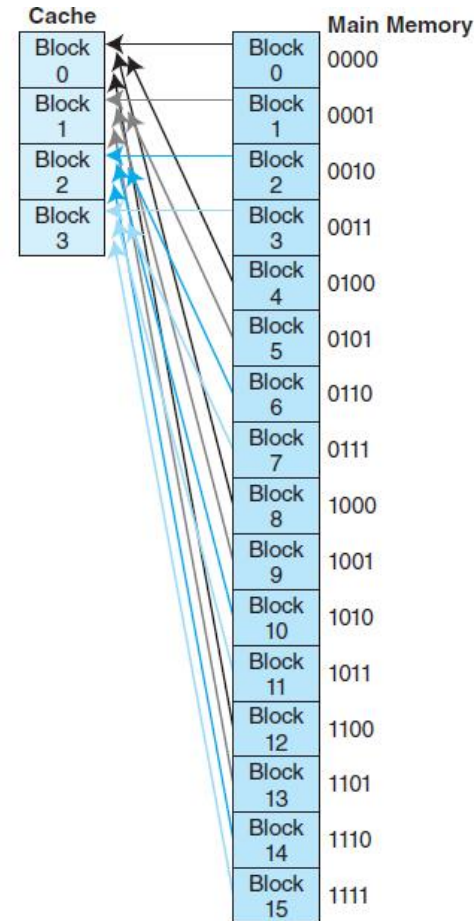
- Write down the names of 5 addressing modes with their pros and cons.
- Differentiate
 - Temporal vs spatial locality
 - Main memory vs. Cache memory
- What is meant by memory hierarchy?
- When any address is generated from CPU to get statement from RAM. How cache controller search in cache using this address. Explain in detail.

Direct Mapped Cache

- More memory blocks than cache blocks
- Blocks competition for cache
- Block X copied to block $Y \% N$, Where $N = \text{no of blocks in cache}$



a) 8 Memory Blocks to 4 Cache Blocks

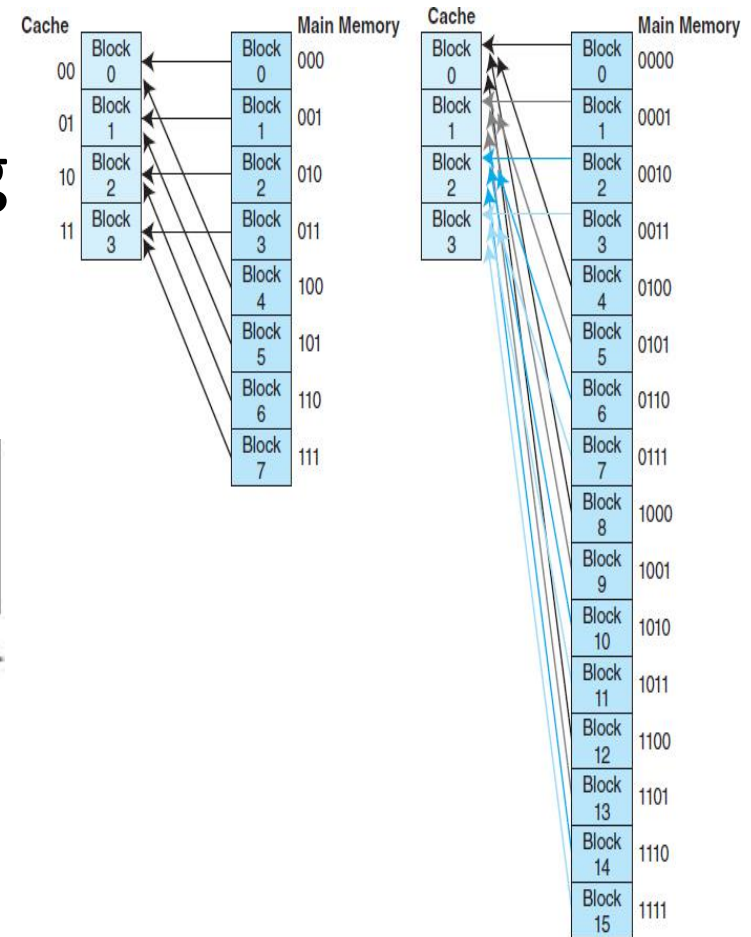


b) 16 Memory Blocks to 4 Cache Blocks

Direct Mapped Cache

- Block 0 and 4 → copied to same block 0
- Which block is there?

Each block has different tag that is stored with it



a) 8 Memory Blocks to 4 Cache Blocks

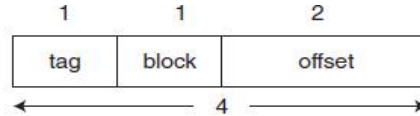
b) 16 Memory Blocks to 4 Cache Blocks

Example 1

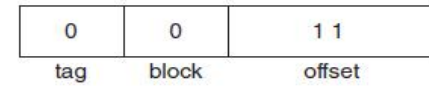
RAM=16 bytes

CACHE= 8 bytes

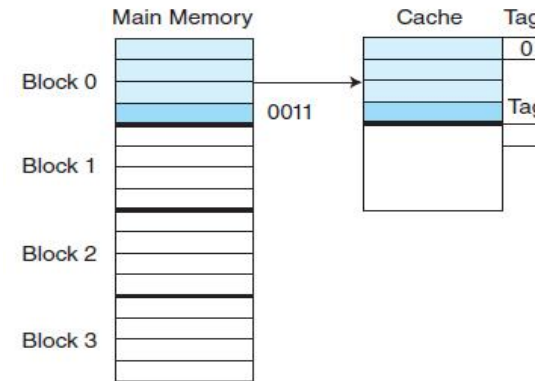
BLOCK SIZE= 4 bytes



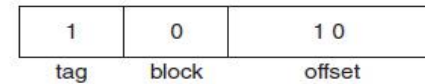
a) Main Memory Format



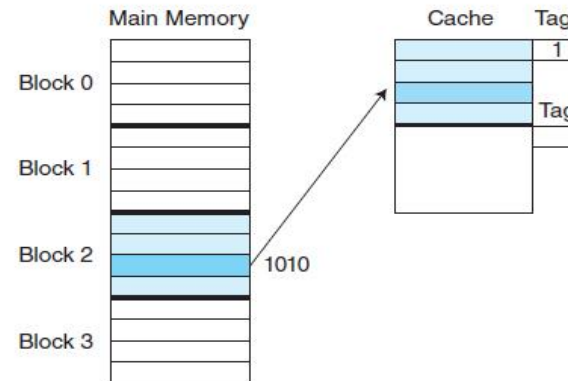
b) The Address 0011 Partitioned into Fields



c) Mapping of Block Containing Address 0011 = 0x3



d) The Address 1010 Partitioned into Fields



e) Mapping of Block Containing Address 1010 = 0xA

Your turn

- Given

RAM = 2^{14} bytes

CACHE= 16 blocks

Block size = 8 bytes

- Determine main memory address format

Offset field = -----bits

Block # field = -----bits

Tag # field = -----bits

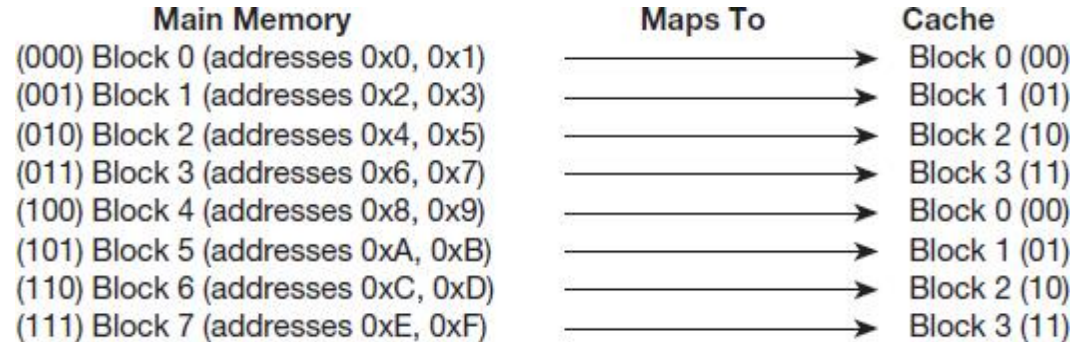
Your turn

- Given

RAM = 16 bytes

CACHE= 8 bytes

Block size = 2 bytes



- Determine main memory address format

Offset field = -----bits

Block # field = -----bits

Tag # field = -----bits

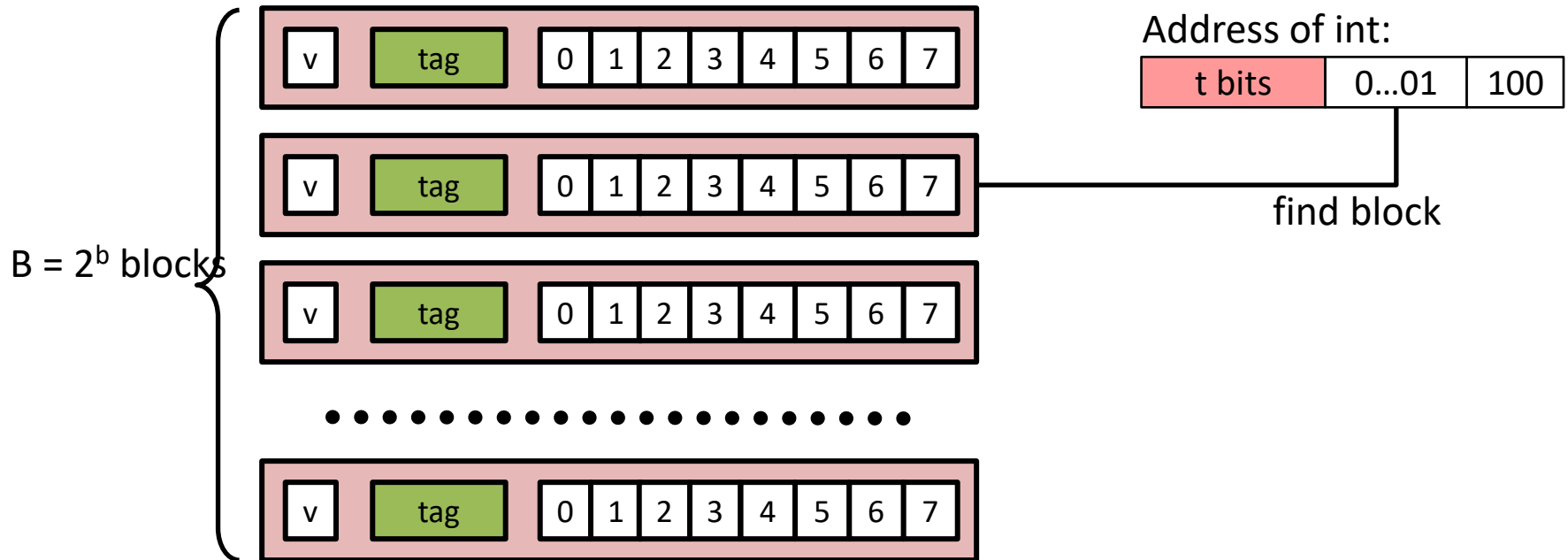
- Determine main memory address 0x09 maps to which block in cache?

Your turn

- Given
 - RAM = 16 bit address
 - CACHE= 64 blocks
 - Block size = 8 bytes
- Determine main memory address format
 - Offset field = -----bits
 - Block # field = -----bits
 - Tag # field = -----bits
- Determine main memory address 0x0404 maps to which block in cache?

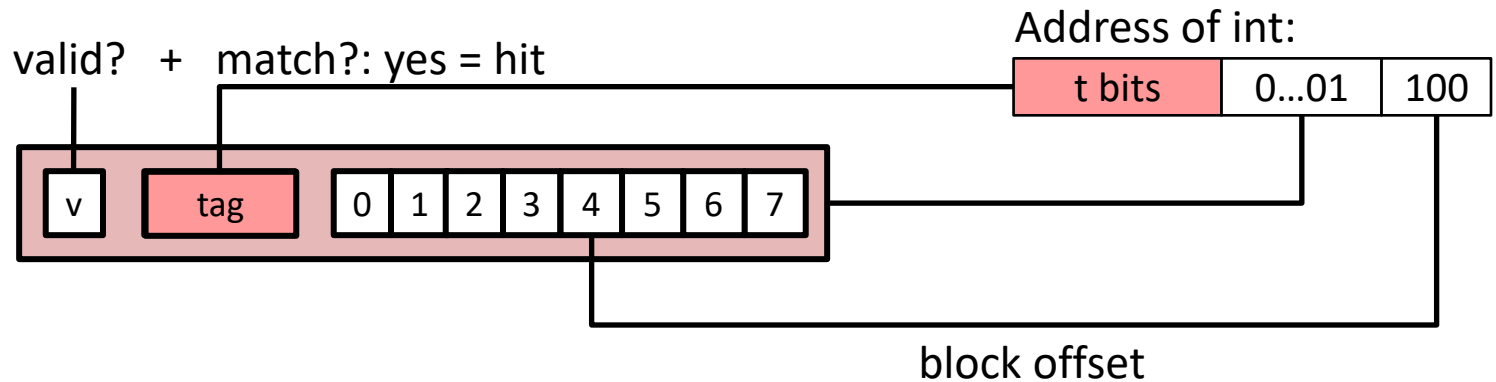
Example: Direct-Mapped Cache

Assume: cache block size = 8 bytes



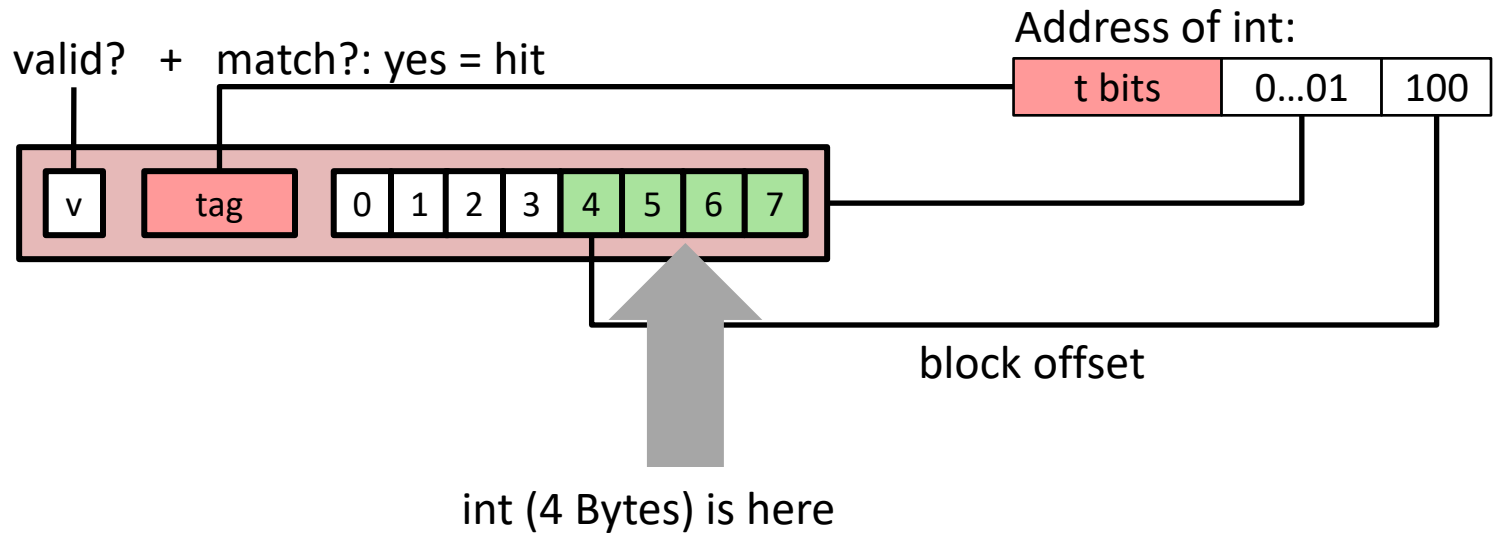
Example: Direct-Mapped

Assume: cache block size 8 bytes



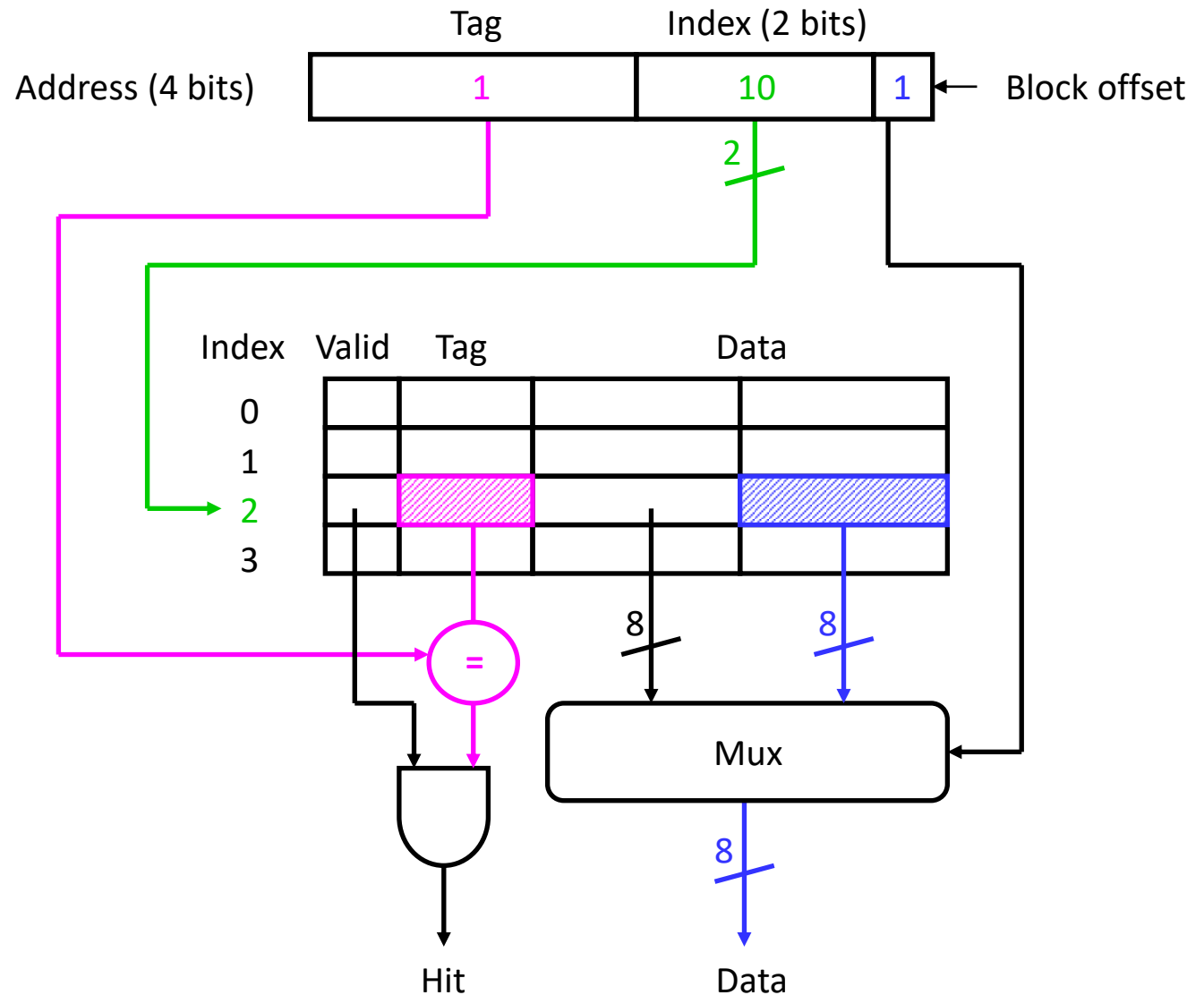
Example: Direct-Mapped Cache

Assume: cache block size 8 bytes

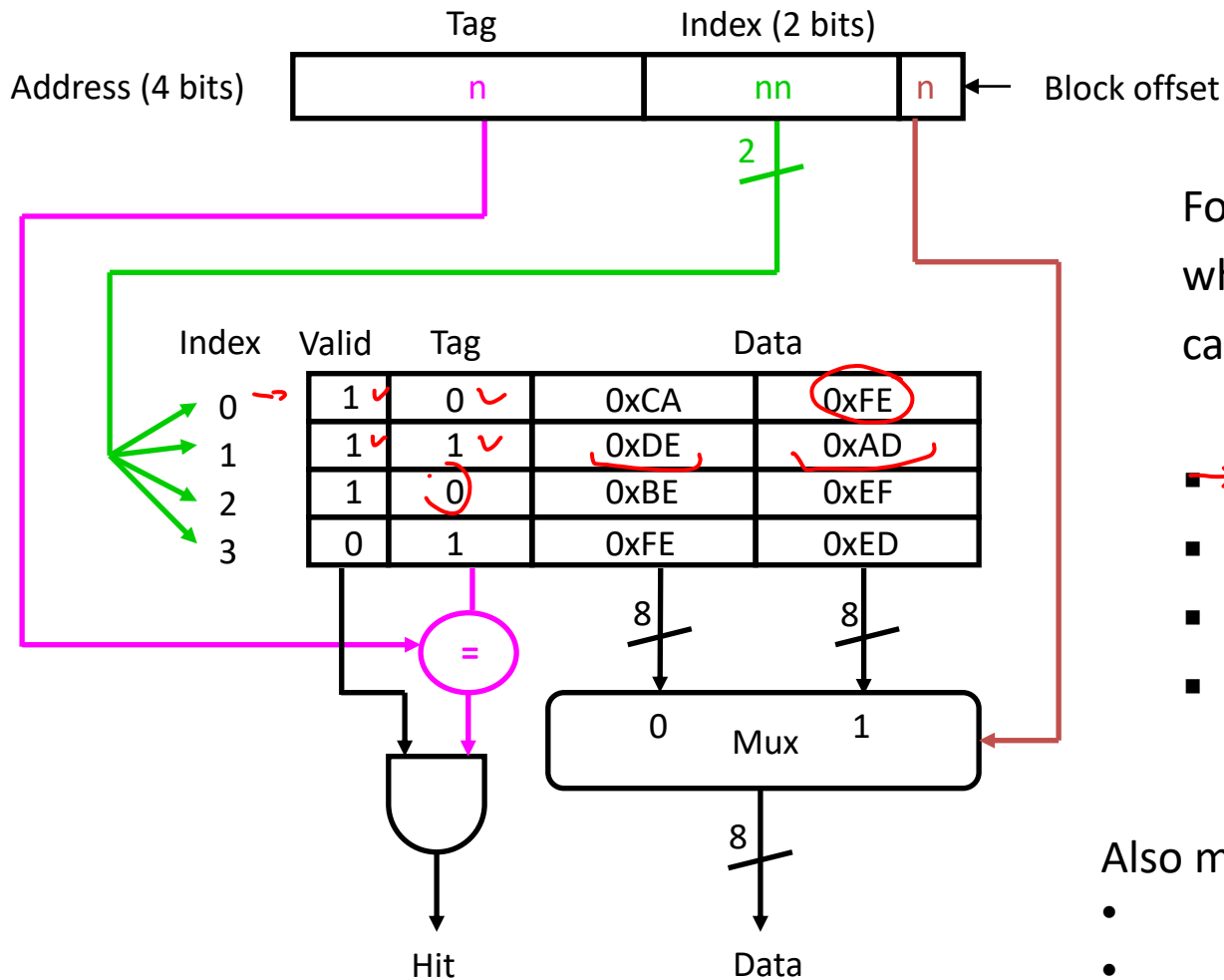


No match: old line is evicted and replaced

Cache controller logic



An exercise



For the addresses below, what byte is read from the cache (or is there a miss)?

- → 1010 0xDE
- 1110 invalid
- 0001 0xFE
- 1101 miss, bad tag

Also mention, miss is due to

- Invalid block
- tag mismatch

Cache Misses

- **Compulsory** (cold start and first reference): first access to a block
 - We can not do something about it BUT
If you are going to run “billions” of instruction, Compulsory Misses are insignificant
- **Capacity**:
 - Cache cannot contain all blocks access by the program
 - Solution: increase cache size
- **Conflict** (collision):
 - Multiple memory locations mapped to the same cache location → Direct Mapped Cache
 - Solution 1: increase cache size
 - Solution 2: increase associativity

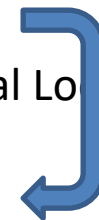
Misses on Cache (remaining constant cost)

	Direct Mapped	N-way Set Associative	Fully Associative
Cache Size	Big	Medium	Small
Compulsory Miss	Same	Same	Same
Conflict Miss	High	Medium	Zero
Capacity Miss	Low	Medium	High

Program is having spatial locality → what to do in cache?

large block size OK then make V-large block **BUT**

Note: block size=1 byte → no advantage of Spatial but Temporal Locality but
V-large block → large miss penalty (long time to fill block)



Want to reduce conflict misses → what to do in cache ?

increase associativity → But costly design

2-16 way set associative cache → good balance

Disadvantages of Direct Mapped Cache

- **Conflict Misses** → main problem
 - **Thrashing** → large amount
 - Suppose a program generates a series of memory references such as: **1AB**, **3AB**, **1AB**, **3AB**, ...
The cache will continually evict
- Other cache mapping schemes are designed to prevent this kind of thrashing.

Fully Associative Cache

- Allow a block to go anywhere in cache.
- In this way, cache would have to fill up before any blocks are evicted.
- A memory address is partitioned into only two fields: the tag and the word.

Fully Associative Cache

- Suppose, as before, we have 14-bit memory addresses and a cache with 16 blocks, each block of size 8. The field format of a memory reference is:



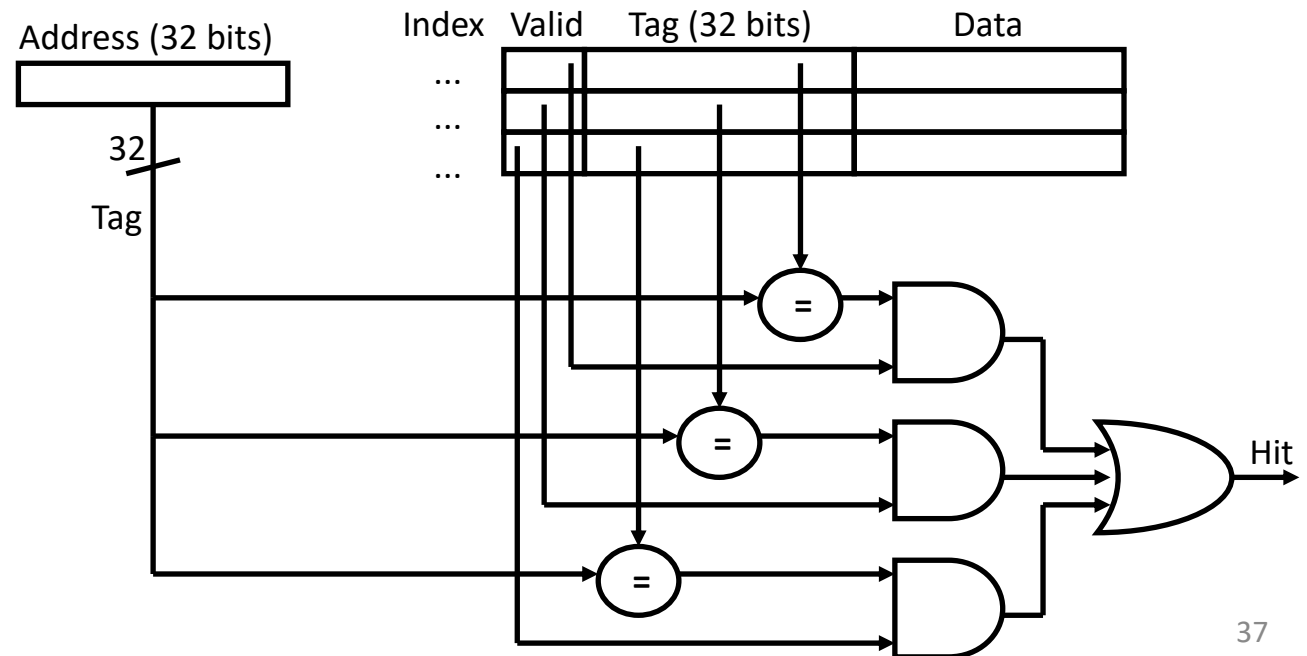
- When the cache is searched, all tags are searched in parallel to retrieve the data quickly.
- This requires special, costly hardware.

Fully Associative Cache

- You will recall that direct mapped cache evicts a block whenever another memory reference needs that block.
- With fully associative cache, we have no such mapping, thus we must devise an algorithm to determine which block to evict from the cache.
- The block that is evicted is the *victim block*.
- There are a number of ways to pick a victim, we will discuss them shortly.

The price of full associativity

- However, a fully associative cache is expensive to implement.
 - Because there is no index field in the address anymore, the *entire* address must be used as the tag, increasing the total cache size.
 - Data could be anywhere in the cache, so we must check the tag of *every* cache block. That's a lot of comparators!



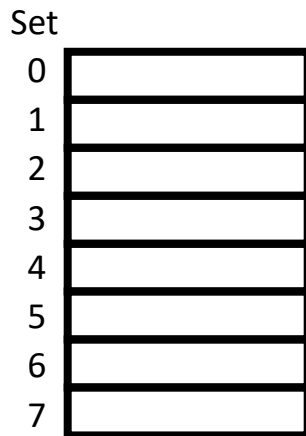
Set Associative Cache

- An intermediate possibility is a **set-associative cache**.
 - The cache is divided into *groups* of blocks, called **sets**.
 - Each memory address maps to exactly one set in the cache, but data may be placed in any block within that set.

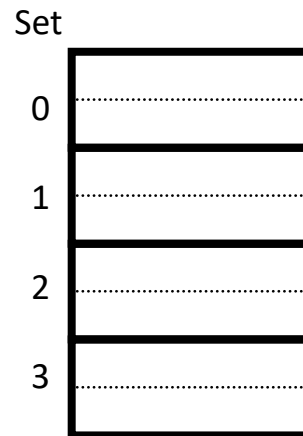
Set associatively

- An intermediate possibility is a **set-associative cache**.
 - The cache is divided into *groups* of blocks, called **sets**.
 - Each memory address maps to exactly one set in the cache, but data may be placed in any block within that set.
- If each set has 2^x blocks, the cache is an **2^x -way associative cache**.
- Here are several possible organizations of an eight-block cache.

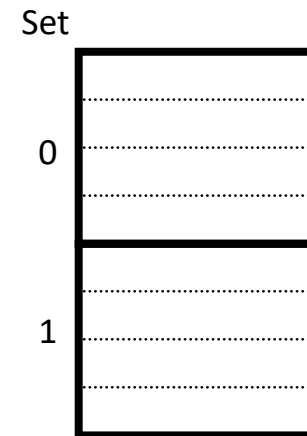
1-way associativity
8 sets, 1 block each



2-way associativity
4 sets, 2 blocks each



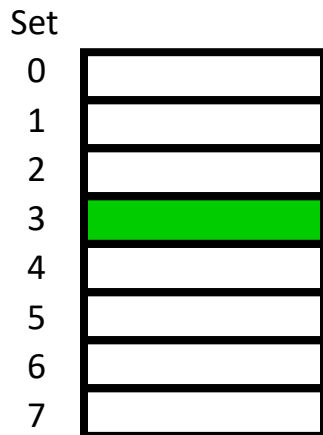
4-way associativity
2 sets, 4 blocks each



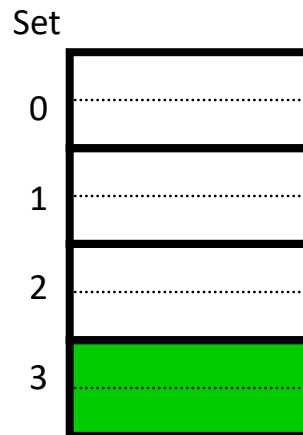
Block replacement

- Any empty block in the correct set may be used for storing data.
- If there are no empty blocks, the cache controller will attempt to replace the least recently used block, just like before.
- For highly associative caches, it's expensive to keep track of what's really the least recently used block, so some approximations are used. We won't get into the details.

1-way associativity
8 sets, 1 block each



2-way associativity
4 sets, 2 blocks each

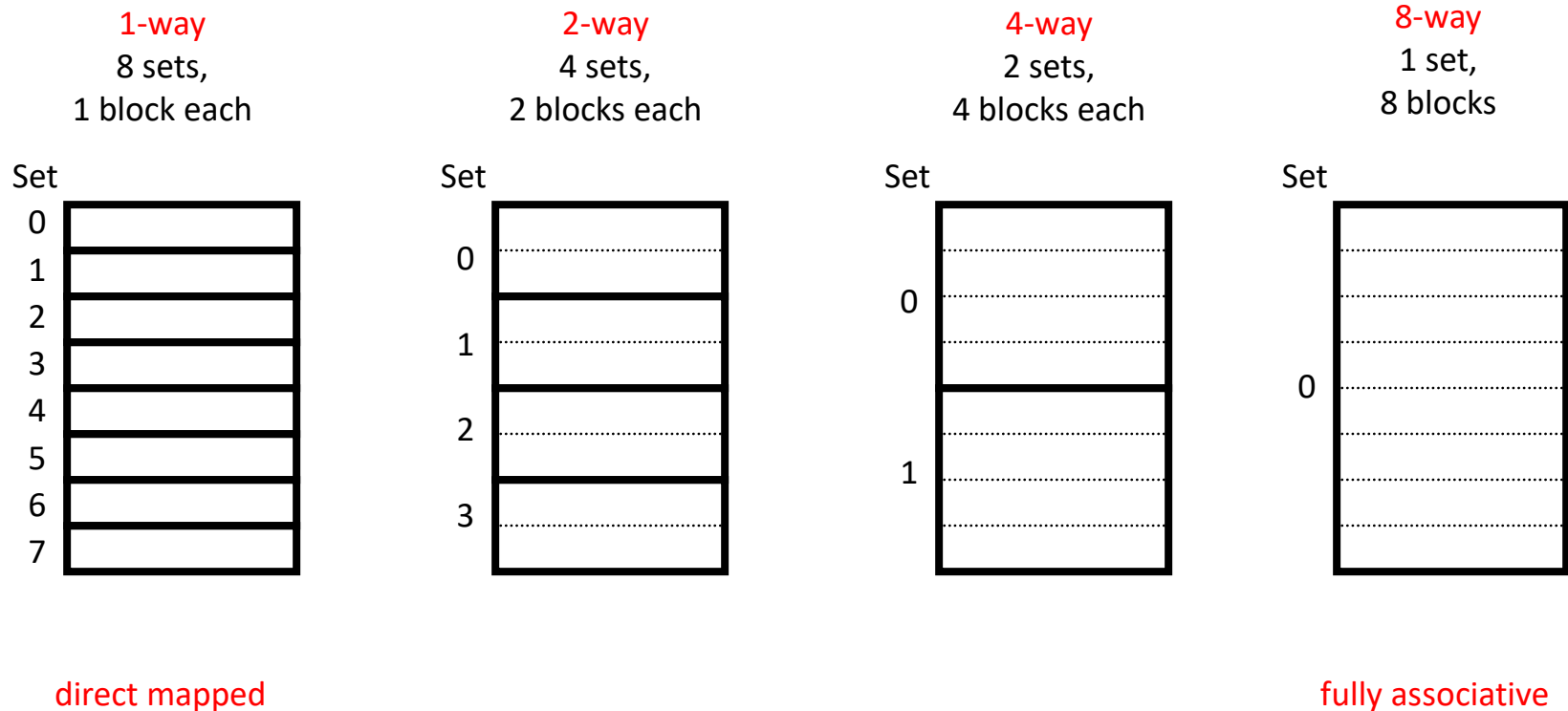


4-way associativity
2 sets, 4 blocks each



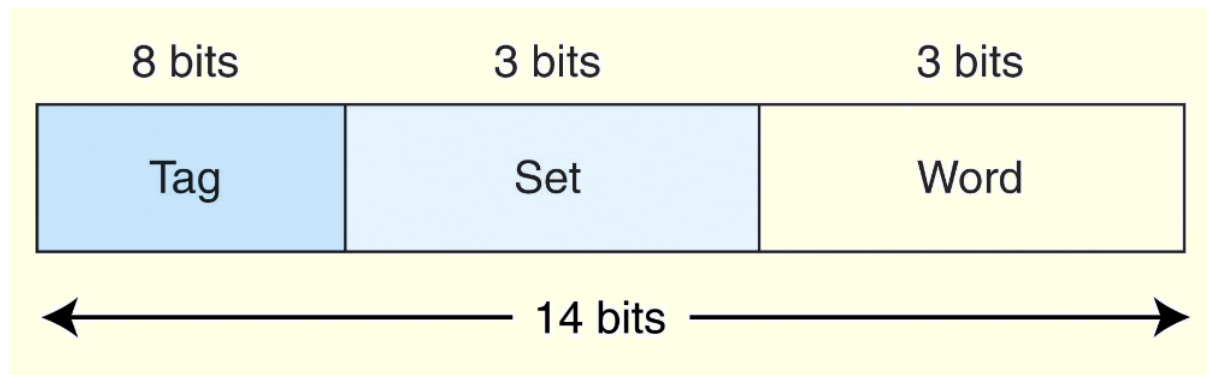
Set associative caches are a general idea

- By now you may have noticed the 1-way set associative cache is the same as a **direct-mapped** cache.
- Similarly, if a cache has 2^k blocks, a 2^k -way set associative cache would be the same as a **fully-associative** cache.



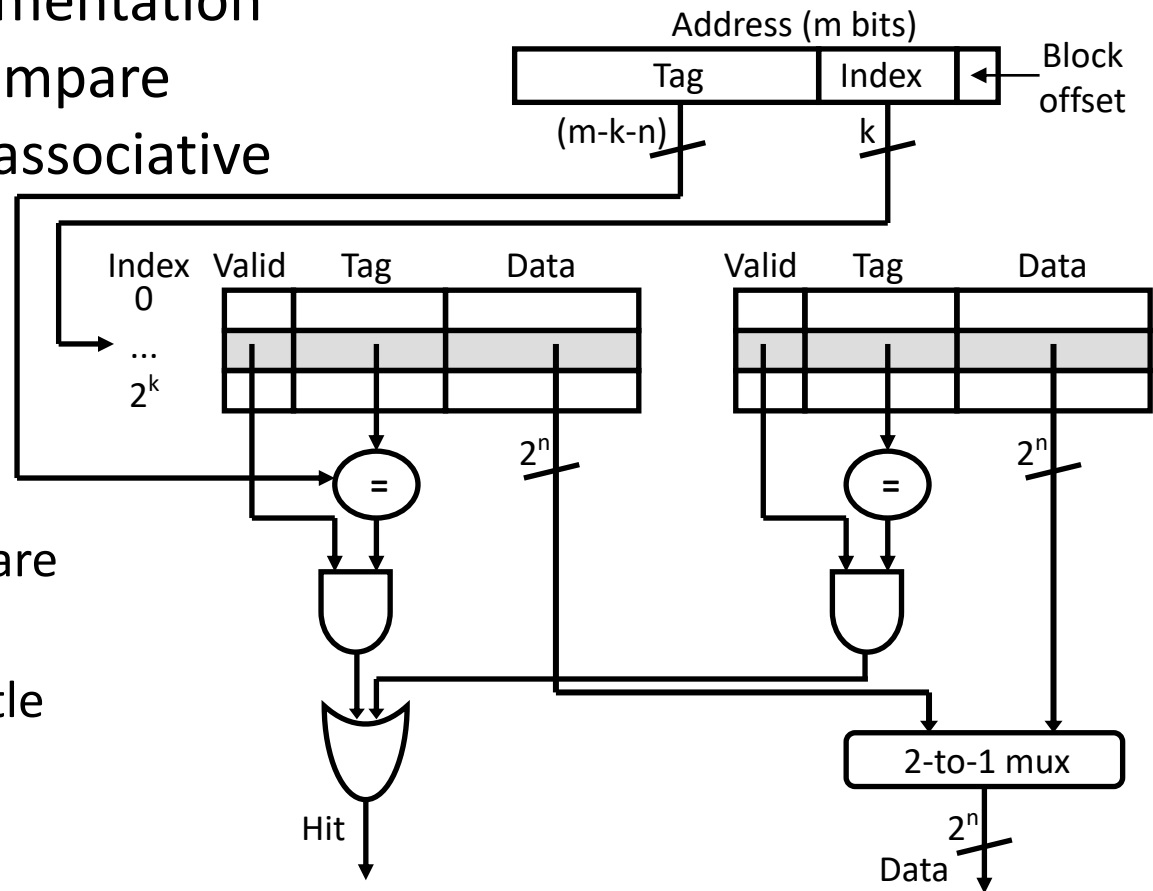
Set Associative Cache

- Suppose we have a main memory of 2^{14} bytes.
- This memory is mapped to a 2-way set associative cache having 16 blocks where each block contains 8 words.
- Since this is a 2-way cache, each set consists of 2 blocks, and there are 8 sets.
- Thus, we need 3 bits for the set, 3 bits for the word, giving 8 leftover bits for the tag:



2-way set associative cache implementation

- How does an implementation of a 2-way cache compare with that of a fully-associative cache?



- Only two comparators are needed.
- The cache tags are a little shorter too.

Larger sets and higher associativity lead to fewer cache conflicts and lower miss rates, but they also increase the hardware cost

General Organization of a Cache

Cache is an array of sets

Each set contains one or more lines

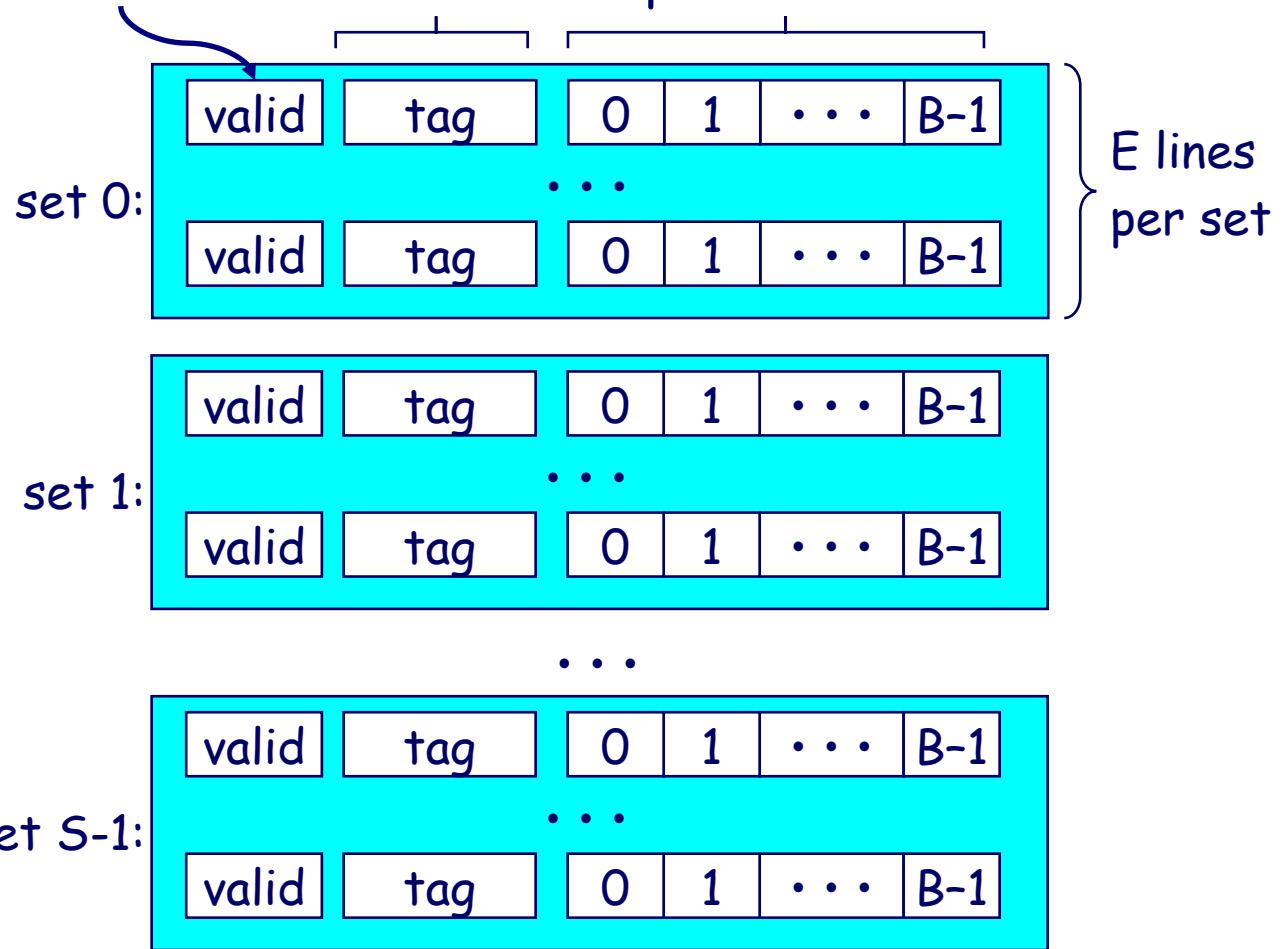
Each line holds a block of data

$$S = 2^s \text{ sets}$$

1 valid bit
per line

+ tag bits
per line

$B = 2^b$ bytes
per cache block



Cache size: $C = B \times E \times S$ data bytes

General Organization of a Cache

Cache is an array of sets

Each set contains one or more lines

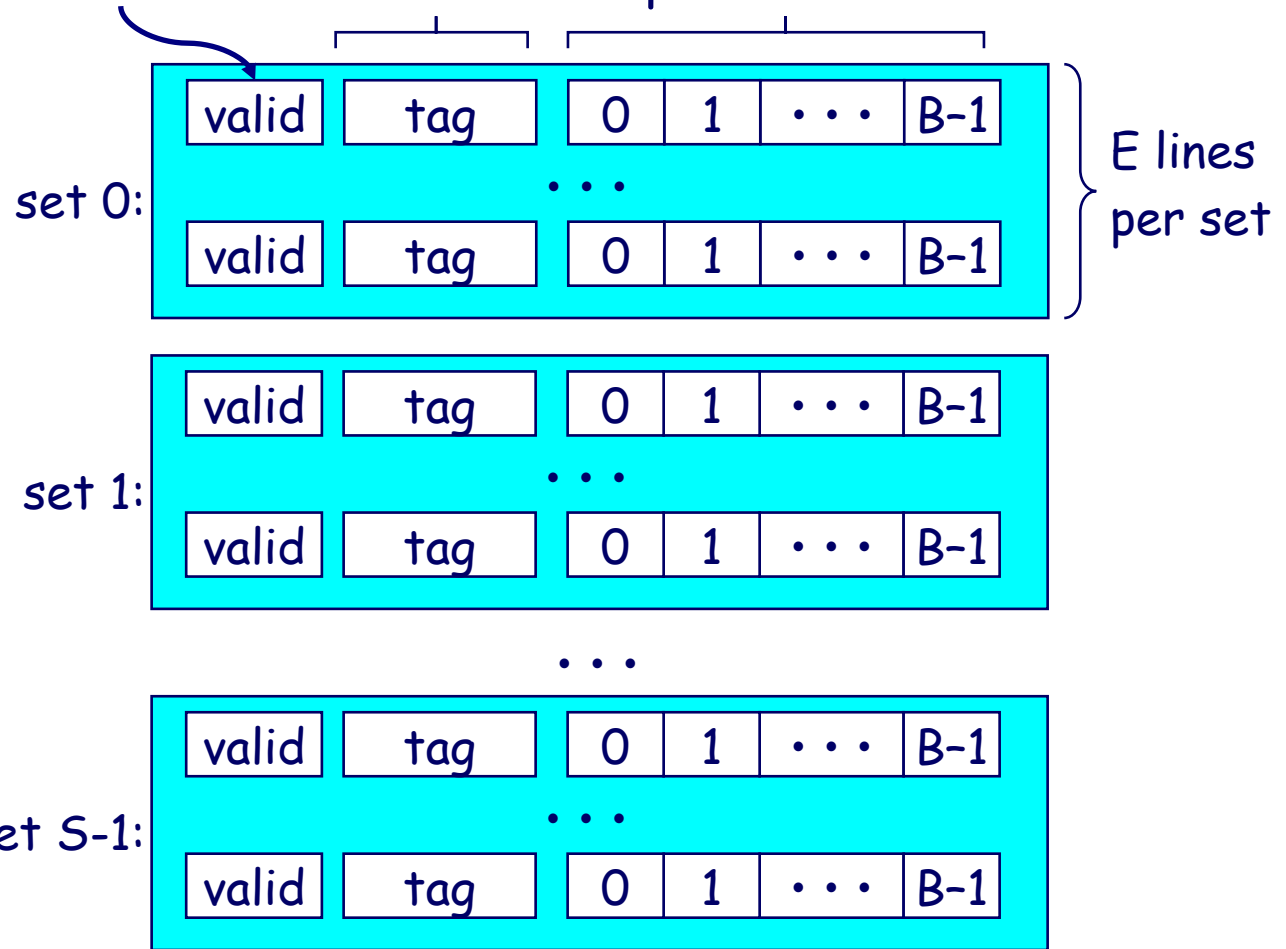
Each line holds a block of data

$$S = 2^s \text{ sets}$$

1 valid bit
per line

+ tag bits
per line

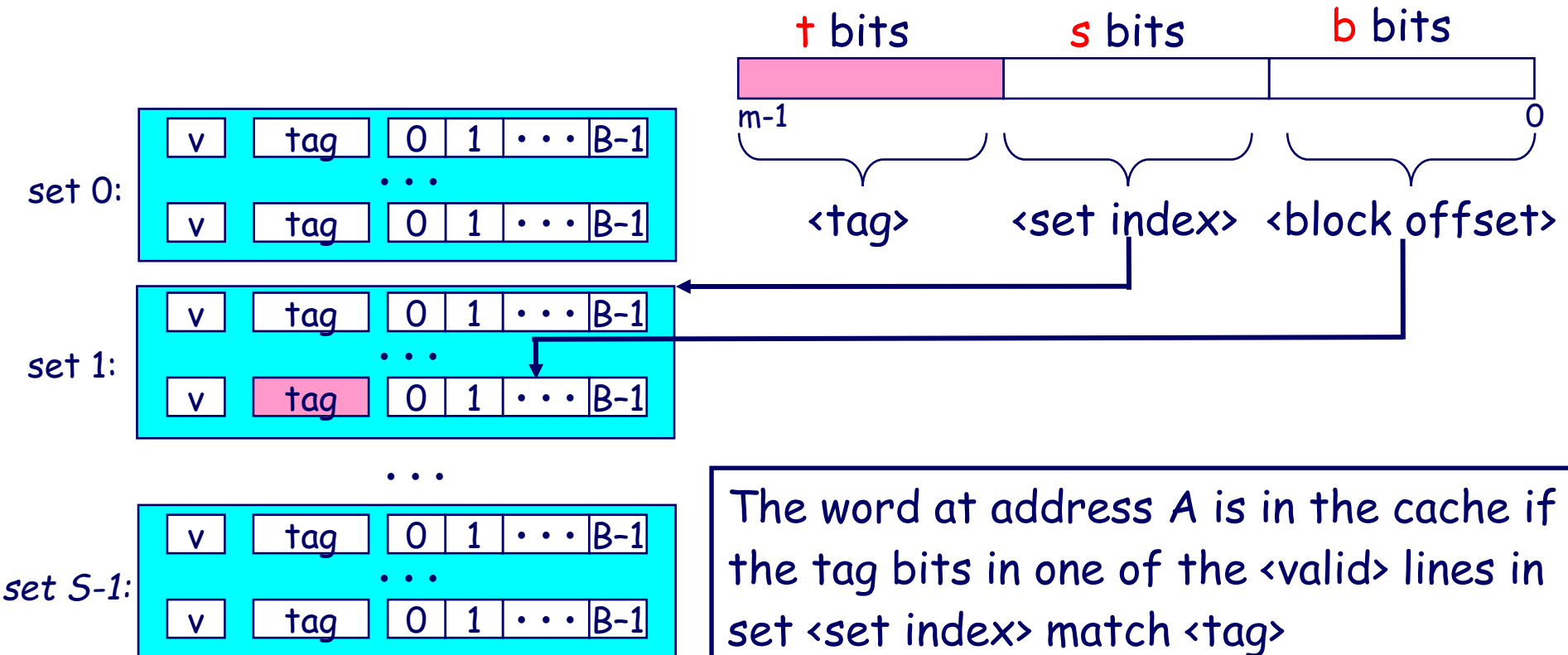
$B = 2^b$ bytes
per cache block



Cache size: $C = B \times E \times S$ data bytes

Addressing Caches

Address A:

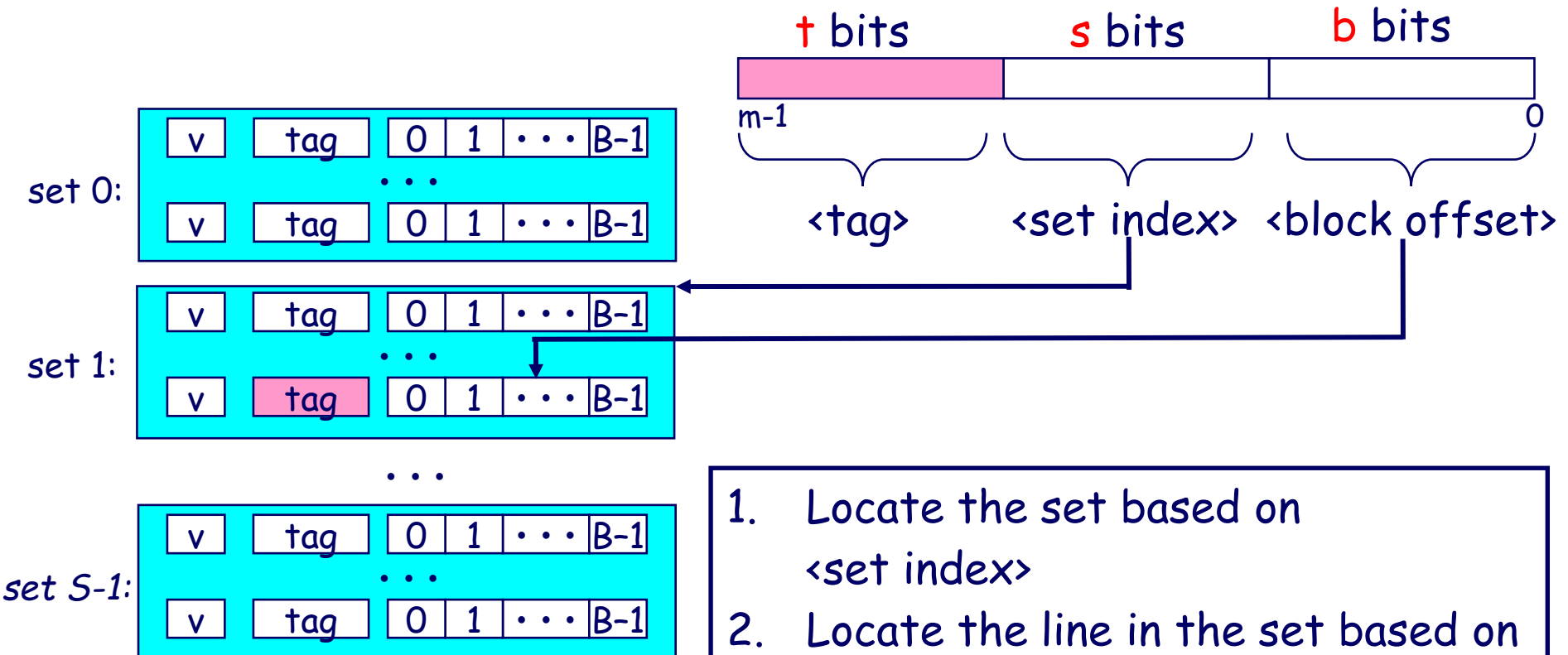


The word at address A is in the cache if the tag bits in one of the <valid> lines in set <set index> match <tag>

The word contents begin at offset <block offset> bytes from the beginning of the block

Addressing Caches

Address A:

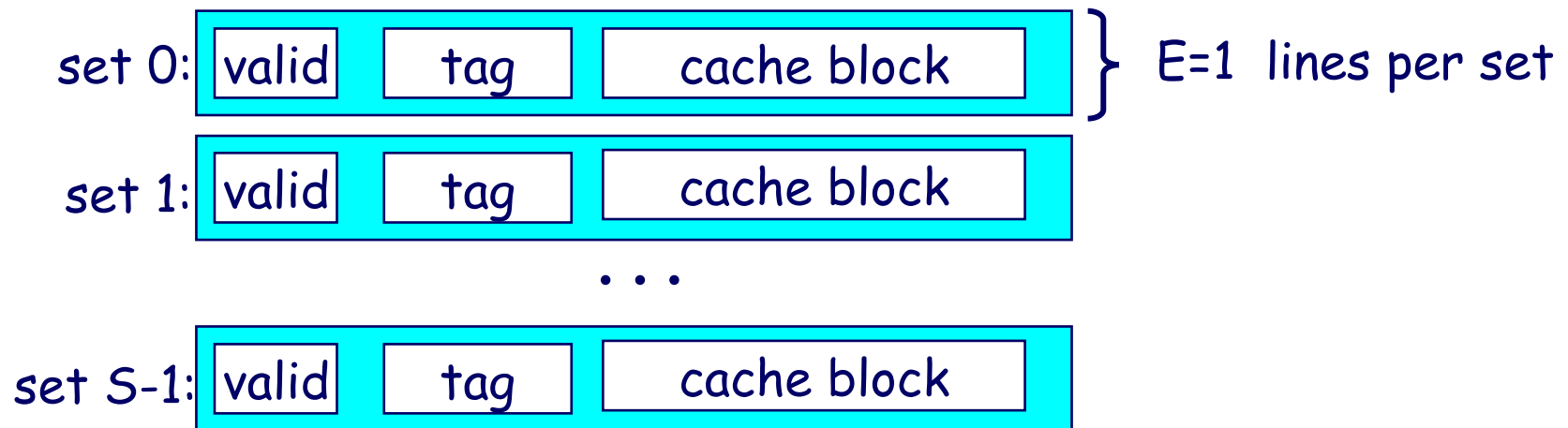


1. Locate the set based on $\langle \text{set index} \rangle$
2. Locate the line in the set based on $\langle \text{tag} \rangle$
3. Check that the line is valid
4. Locate the data in the line based on $\langle \text{block offset} \rangle$

Example: Direct-Mapped Cache

Simplest kind of cache, easy to build
(only 1 tag compare required per access)

Characterized by exactly one line per set.

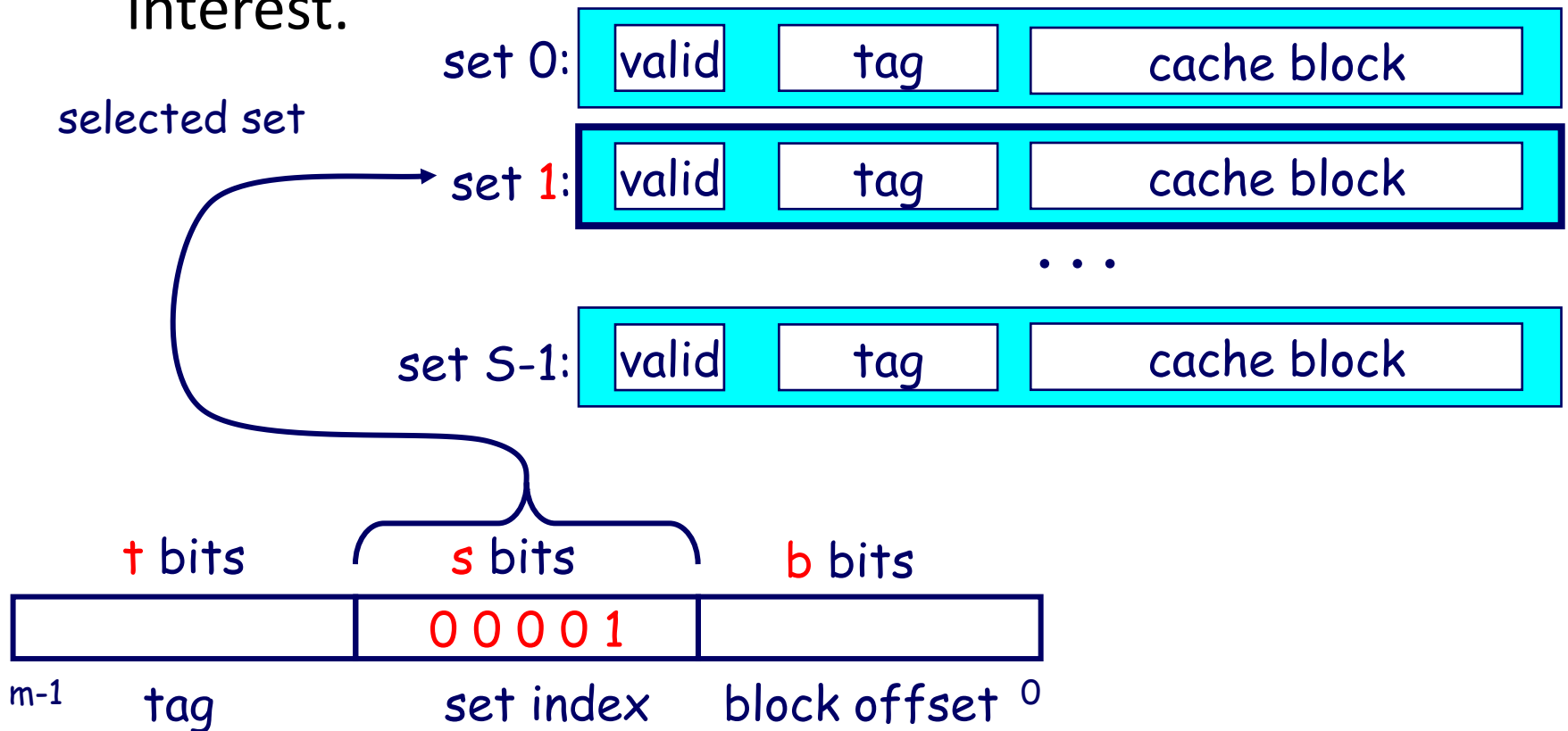


Cache size: $C = B \times S$ data bytes

Accessing Direct-Mapped Caches

Set selection

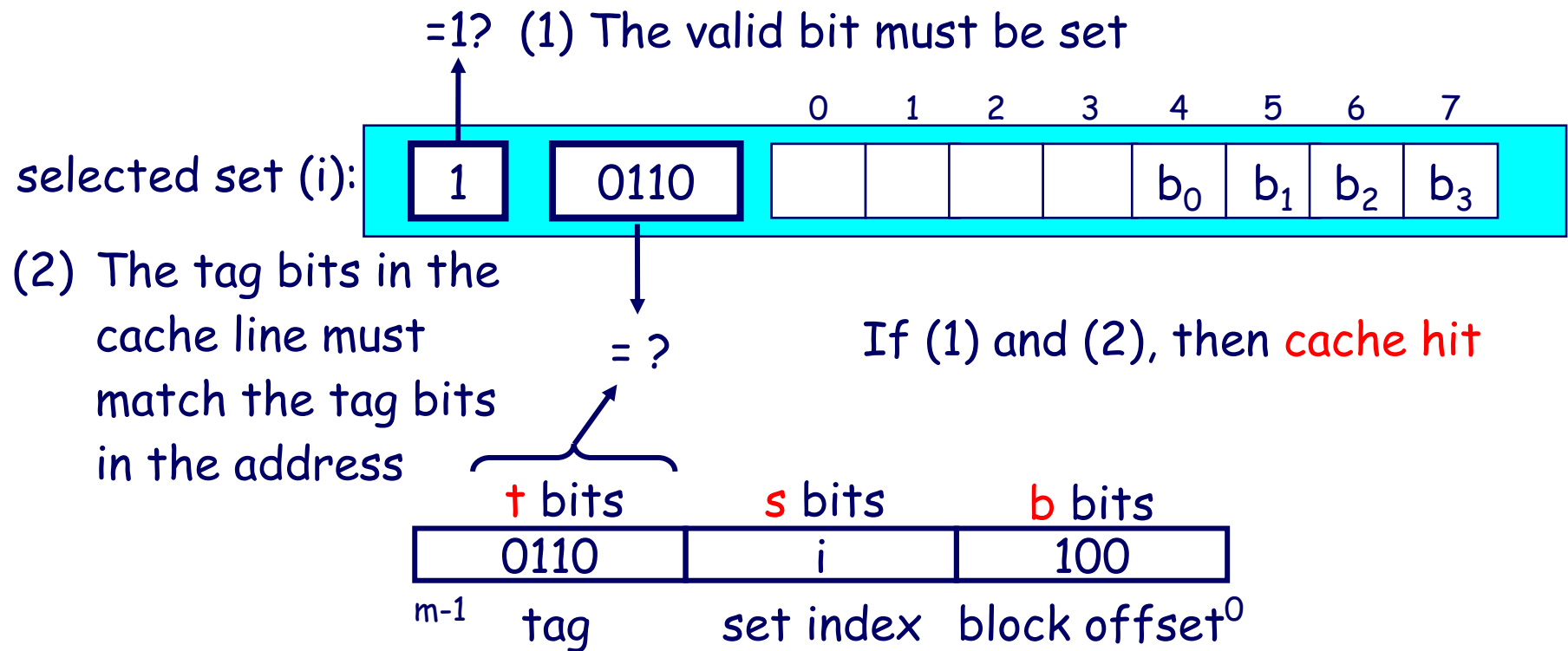
- Use the set index bits to determine the set of interest.



Accessing Direct-Mapped Caches

Line matching and word selection

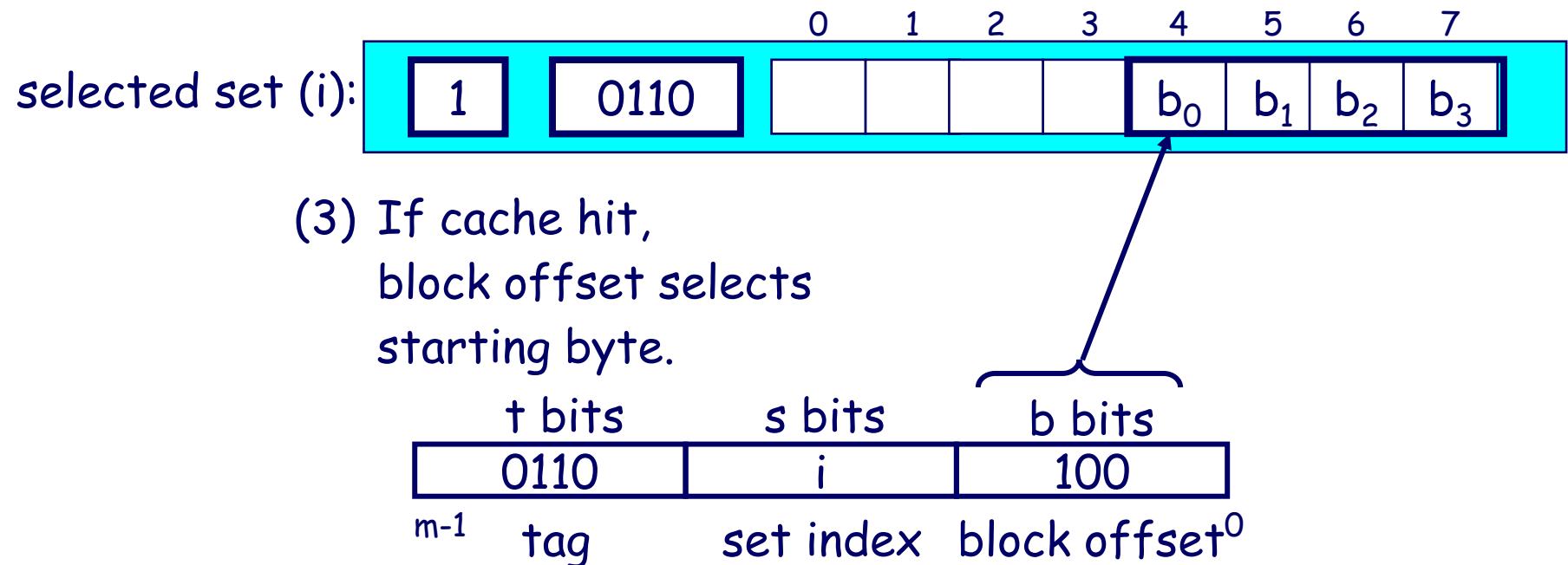
- **Line matching**: Find a valid line in the selected set with a matching tag
- **Word selection**: Then extract the word



Accessing Direct-Mapped Caches

Line matching and word selection

- **Line matching**: Find a valid line in the selected set with a matching tag
- **Word selection**: Then extract the word



Direct-Mapped Cache Simulation

$M=16$ byte addresses, $B=2$ bytes/block,
 $S=4$ sets, $E=1$ entry/set

$t=1$ $s=2$ $b=1$

x	xx	x
---	----	---

Address trace (reads):

0	$[0000_2]$,	miss
1	$[0001_2]$,	hit
7	$[0111_2]$,	miss
8	$[1000_2]$,	miss
0	$[0000_2]$	miss

v	tag	data
1	0	$M[0-1]$
1	0	$M[6-7]$

Given



Parameters

Descriptions

$$S = 2^s$$

Number of sets

E

Number of lines per set

$$B = 2^b$$

Block size(bytes)

$$m = \log_2(M)$$

Number of physical(main memory) address bits

Derived

Parameters

Descriptions

$$M = 2^m$$

Maximum number of unique memory address

$$s = \log_2(S)$$

Number of *set index bits*

$$b = \log_2(B)$$

Number of block offset bits

$$t = m - (s + b)$$

Number of tag bits

$$C = B \times E \times S$$

Cache size (bytes) not including overhead such as the valid and tag bits

Replacement Algorithm

- With fully associative and set associative cache, a *replacement policy* is invoked when it becomes necessary to evict a block from cache.
- An *optimal* replacement policy would be able to look into the future to see which blocks won't be needed for the longest period of time.
- Although it is impossible to implement an optimal replacement algorithm, it is instructive to use it as a benchmark for assessing the efficiency of any other scheme we come up with.

Replacement Algorithm

- The replacement policy that we choose depends upon the locality that we are trying to optimize-- usually, we are interested in temporal locality.
- A *least recently used* (LRU) algorithm keeps track of the last time that a block was accessed and evicts the block that has been unused for the longest period of time.
- The disadvantage of this approach is its complexity: LRU has to maintain an access history for each block, which ultimately slows down the cache.

Replacement Algorithm

- *First-in, first-out* (FIFO) is a popular cache replacement policy.
- In FIFO, the block that has been in the cache the longest, regardless of when it was last used.
- A *random* replacement policy does what its name implies: It picks a block at random and replaces it with a new block.
- Random replacement can certainly evict a block that will be needed often or needed soon, but it never thrashes.

Effective Access Time (EAT)

- The performance of hierarchical memory is measured by its *effective access time* (EAT).
- EAT is a weighted average that takes into account the hit ratio and relative access times of successive levels of memory.
- The EAT for a two-level memory is given by:

$$\text{EAT} = H \times \text{Access}_C + (1-H) \times \text{Access}_{MM}.$$

where H is the cache hit rate and Access_C and Access_{MM} are the access times for cache and main memory, respectively.

Effective Access Time (EAT)

- For example, consider a system with a main memory access time of 200ns supported by a cache having a 10ns access time and a hit rate of 99%.
- The EAT is:
$$0.99(10\text{ns}) + 0.01(200\text{ns}) = 9.9\text{ns} + 2\text{ns} = 11\text{ns}.$$
- This equation for determining the effective access time can be extended to any number of memory levels

Thanks!