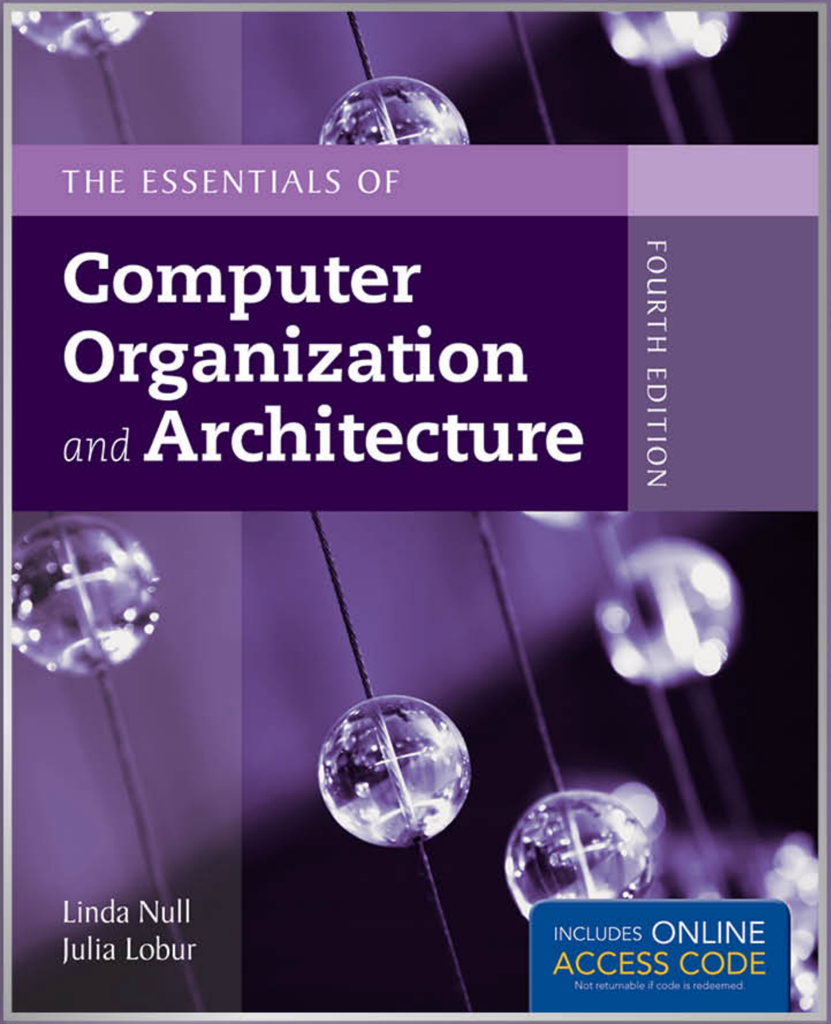
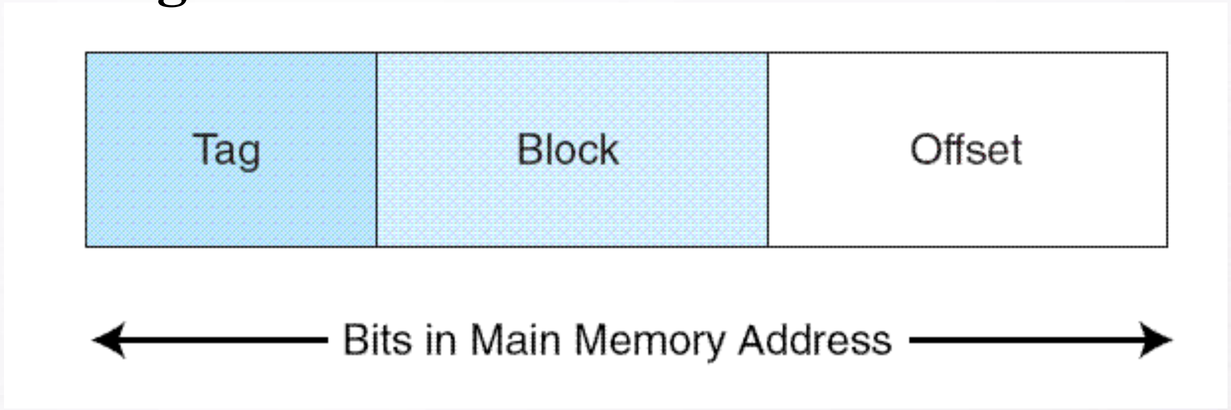
Chapter 6
Memory


THE ESSENTIALS OF
Computer Organization and Architecture
FOURTH EDITION
Linda Null
Julia Lobur
INCLUDES ONLINE ACCESS CODE
Not returnable if code is redeemed.

6.4 Cache Memory

- **Recap:** in **direct mapped caching**, the main memory **address is partitioned into** the **fields** below.
  - The **offset** field uniquely identifies an address within a specific block.
  - The **block** field selects a unique block of cache.
  - The **tag** field is whatever is left over.



  - **main memory blocks are mapped in a modular fashion to cache blocks**

6.4 Class Exercise

Suppose a computer using direct mapped cache has 2^{20} **bytes** of byte-addressable **main memory** and a **cache of 32 blocks**, where **each cache block contains 16 bytes**.

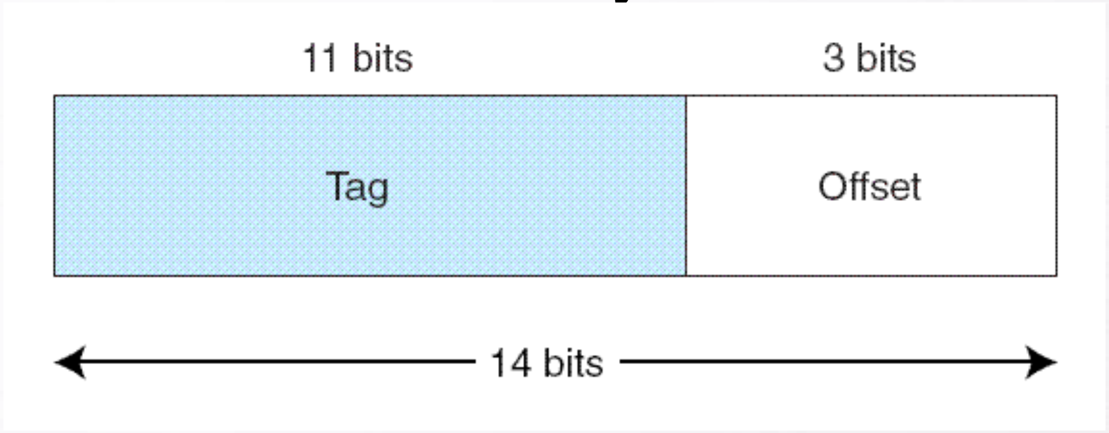
- How **many blocks of main memory** are there?
- What is the format of a memory address as seen by the cache (**tag, block, offset**)?
- To which cache block will the memory address **0000 1101 1011 0110 0011** map?

6.4 Cache Memory

- **Instead of** placing memory blocks in **specific cache locations** based on memory address, **we could allow a block to go anywhere** in cache.
- In this way, the **cache would have to fill up before any blocks are evicted**.
- This is how **fully associative cache** works.
- A **memory address** is **partitioned into** only two fields: the **tag** and the **offset**.

6.4 Cache Memory

- Suppose, 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.

6.4 Cache Memory

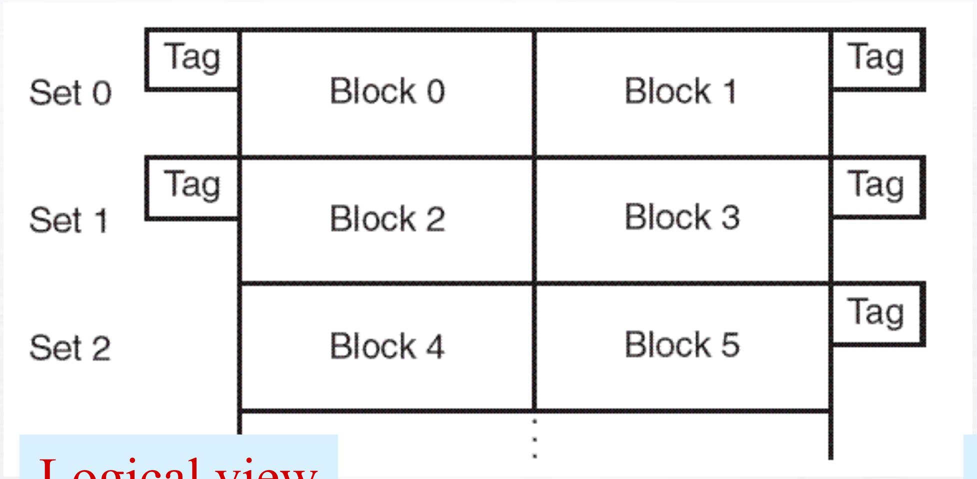
- recall that **direct mapped cache evicts a block whenever another memory reference needs that block**.
- With **fully associative cache**, we have no such mapping, thus **we must** devise an algorithm to **determine which block to evict** from the cache.
- The block that is evicted is the **victim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

6.4 Cache Memory

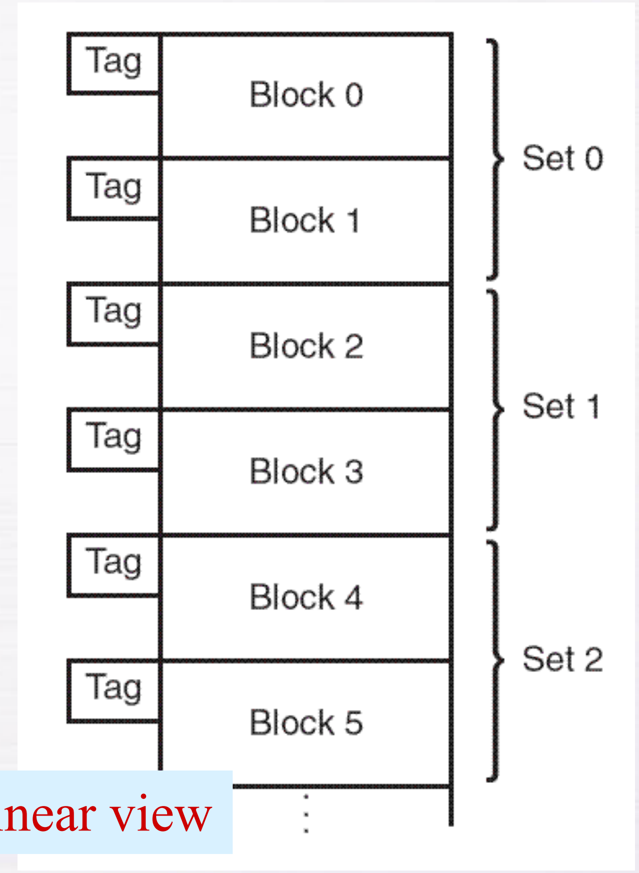
- **Set associative cache combines** the ideas of **direct mapped cache** and **fully associative cache**.
- An ***N*-way set associative cache** mapping is like direct mapped cache in that a memory reference **maps to a particular set** in the cache.
- Unlike direct mapped cache, a memory reference maps to a **set of several cache blocks**, similar to the way in which fully associative cache works.
- Instead of mapping anywhere in the entire cache, **a memory reference can map only to the subset of cache slots.**

6.4 Cache Memory

- The number of cache blocks per set in set associative cache varies according to overall system design.
  - For **example, a 2-way set associative cache** can be conceptualized as shown in the schematic below.
  - **Each set contains two different memory blocks.**



Logical view



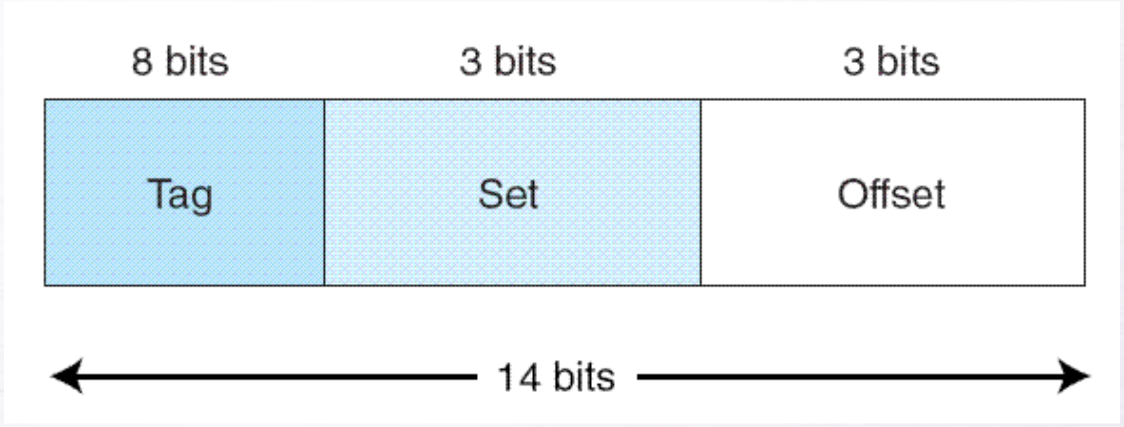
Linear view

6.4 Cache Memory

- In **set associative cache** mapping, a memory reference is divided into **three fields: tag, set, and offset.**
- As with direct-mapped cache, the **offset** field **determines** the **word/byte within the block**,
- the **tag** field uniquely **identifies** the **memory address**.
- The **set field determines the set to which the memory block maps.**

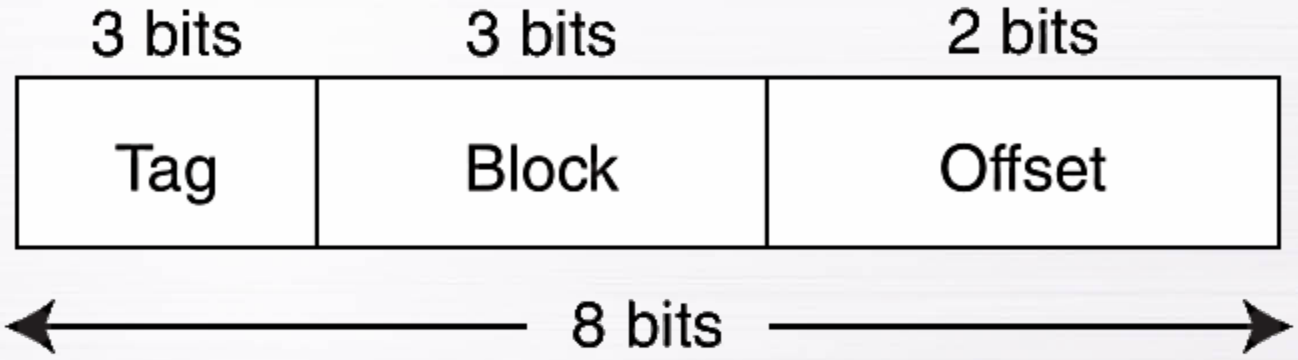
6.4 Cache Memory

- EXAMPLE 6.5 Suppose we are using **2-way set associative mapping** with a **word-addressable** main memory of **2^{14} words** and a **cache with 16 blocks**, where each **block contains 8 words**.
  - A total of **16 cache blocks**, and each **set has 2 blocks**, then there are **8 sets** in cache.
  - Thus, the **set field is 3 bits**, the **offset field is 3 bits**, and the **tag field is 8 bits**.



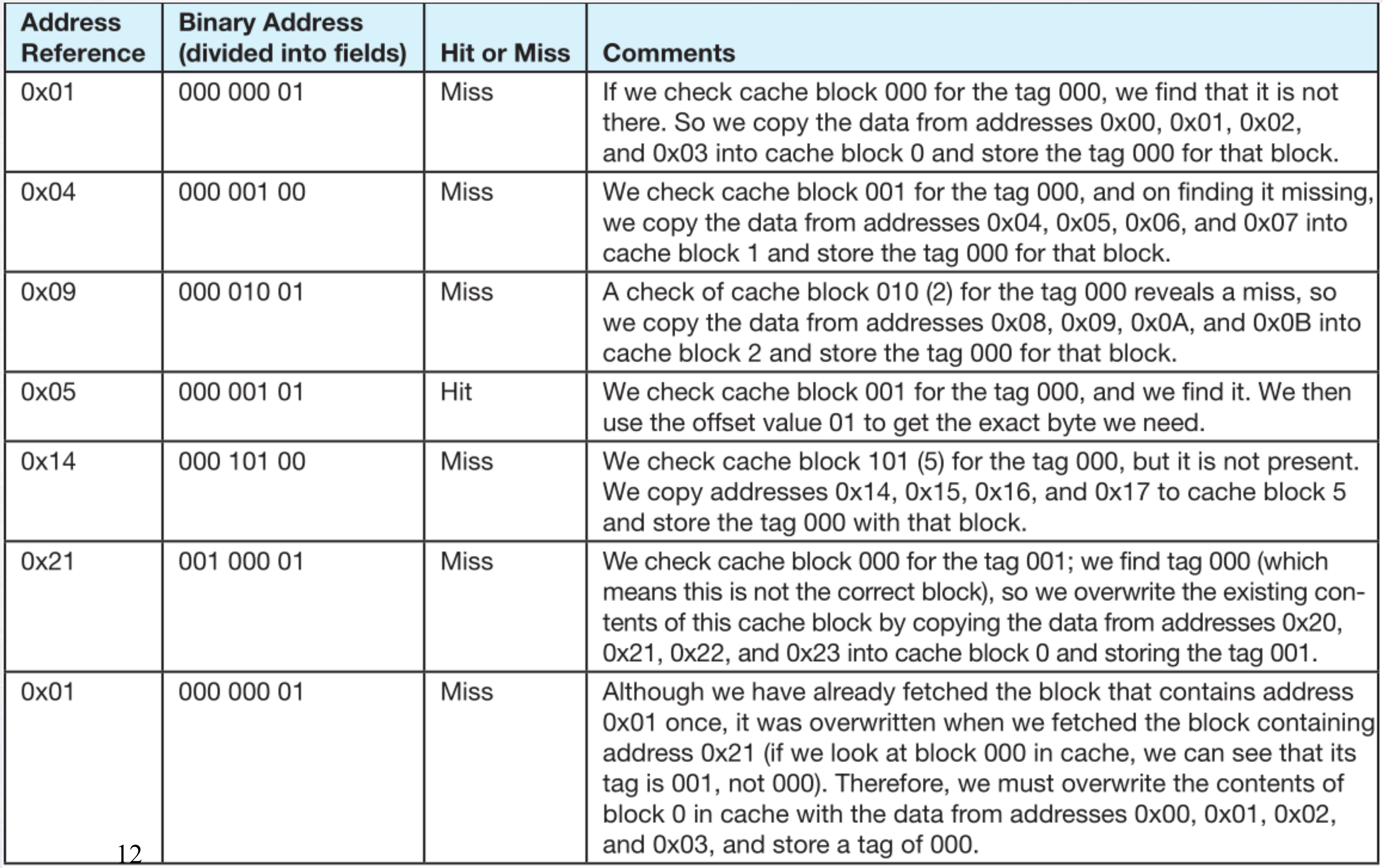
6.4 Cache Memory

- EXAMPLE 6.7 A byte-addressable computer with an **8-block cache** of **4 bytes each**, we compare memory accesses: **0x01, 0x04, 0x09, 0x05, 0x14, 0x21, and 0x01** for each mapping approach.
- The **address format** for **direct mapped cache** is:



Our trace is on the next slide.

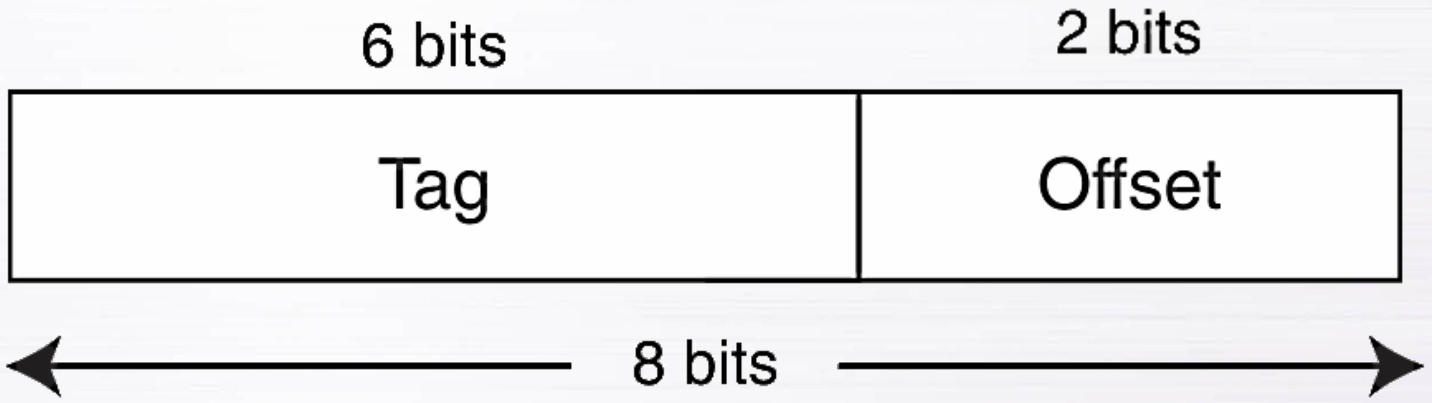
6.4 Cache Memory



Address Reference	Binary Address (divided into fields)	Hit or Miss	Comments
0x01	000 000 01	Miss	If we check cache block 000 for the tag 000, we find that it is not there. So we copy the data from addresses 0x00, 0x01, 0x02, and 0x03 into cache block 0 and store the tag 000 for that block.
0x04	000 001 00	Miss	We check cache block 001 for the tag 000, and on finding it missing, we copy the data from addresses 0x04, 0x05, 0x06, and 0x07 into cache block 1 and store the tag 000 for that block.
0x09	000 010 01	Miss	A check of cache block 010 (2) for the tag 000 reveals a miss, so we copy the data from addresses 0x08, 0x09, 0x0A, and 0x0B into cache block 2 and store the tag 000 for that block.
0x05	000 001 01	Hit	We check cache block 001 for the tag 000, and we find it. We then use the offset value 01 to get the exact byte we need.
0x14	000 101 00	Miss	We check cache block 101 (5) for the tag 000, but it is not present. We copy addresses 0x14, 0x15, 0x16, and 0x17 to cache block 5 and store the tag 000 with that block.
0x21	001 000 01	Miss	We check cache block 000 for the tag 001; we find tag 000 (which means this is not the correct block), so we overwrite the existing contents of this cache block by copying the data from addresses 0x20, 0x21, 0x22, and 0x23 into cache block 0 and storing the tag 001.
0x01	000 000 01	Miss	Although we have already fetched the block that contains address 0x01 once, it was overwritten when we fetched the block containing address 0x21 (if we look at block 000 in cache, we can see that its tag is 001, not 000). Therefore, we must overwrite the contents of block 0 in cache with the data from addresses 0x00, 0x01, 0x02, and 0x03, and store a tag of 000.

6.4 Cache Memory

- EXAMPLE 6.7 A byte-addressable computer with an **8-block cache** of **4 bytes each**, we compare memory accesses: **0x01, 0x04, 0x09, 0x05, 0x14, 0x21, and 0x01** for each mapping approach.
- The **address format** for **fully associative cache** is:



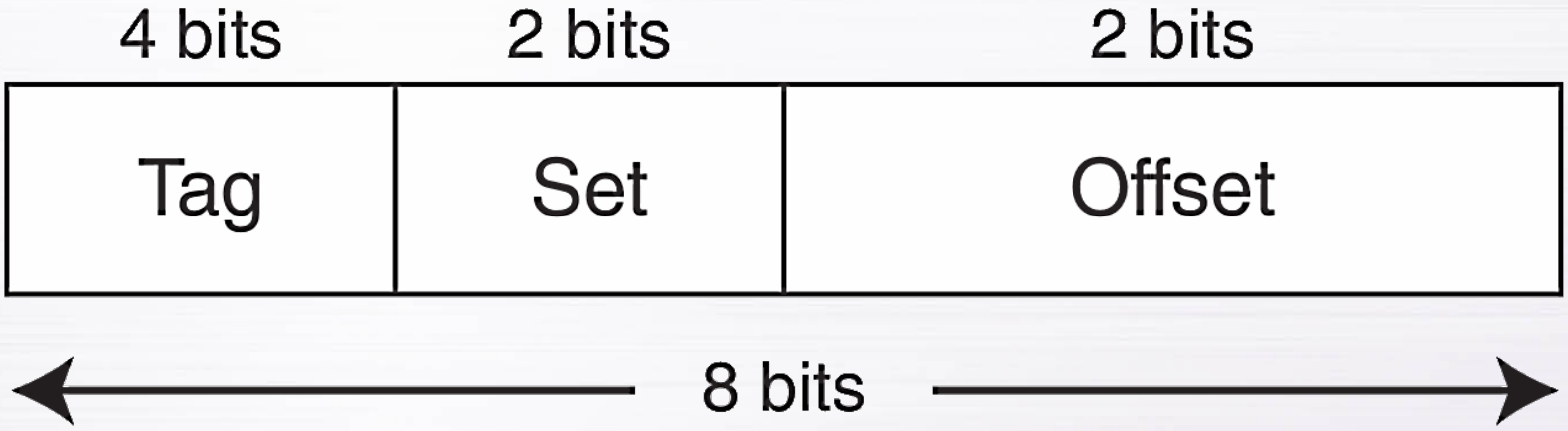
Our trace is on the next slide.

6.4 Cache Memory

Address Reference	Binary Address (divided into fields)	Hit or Miss	Comments
0x01	000000 01	Miss	We search all of cache for the tag 000000, and we don't find it. So we copy the data from addresses 0x00, 0x01, 0x02, and 0x03 into cache block 0 and store the tag 000000 for that block.
0x04	000001 00	Miss	We search all of cache for the tag 000001, and on finding it missing, we copy the data from addresses 0x04, 0x05, 0x06, and 0x07 into cache block 1 and store the tag 000001 for that block.
0x09	000010 01	Miss	We don't find the tag 000010 in cache, so we copy the data from addresses 0x08, 0x09, 0x0A, and 0x0B into cache block 2 and store the tag 000010 for that block.
0x05	000001 01	Hit	We search all of cache for the tag 000001, and we find it stored with cache block 1. We then use the offset value 01 to get the exact byte we need.
0x14	000101 00	Miss	We search all of cache for the tag 000101, but it is not present. We copy addresses 0x14, 0x15, 0x16, and 0x17 to cache block 3 and store the tag 000101 with that block.
0x21	001000 01	Miss	We search all of cache for the tag 001000; we don't find it, so we copy the data from addresses 0x20, 0x21, 0x22, and 0x23 into cache block 4 and store the tag 001000.
0x01	000000 01	Hit	We search cache for the tag 000000 and find it with cache block 0. We use the offset of 1 to find the data we want.

6.4 Cache Memory

- <u>EXAMPLE 6.7</u> A byte-addressable computer with an **8-block cache** of **4 bytes each**, we compare memory accesses: **0x01, 0x04, 0x09, 0x05, 0x14, 0x21, and 0x01** for each mapping approach.
- The **address format** for **2-way set-associative cache** is:



Our trace is on the next slide.

6.4 Cache Memory

Address Reference	Binary Address (divided into fields)	Hit or Miss	Comments
0x01	0000 00 01	Miss	We search in set 0 of cache for a block with the tag 0000, and we find it is not there. So we copy the data from addresses 0x00, 0x01, 0x02, and 0x03 into set 0 (so now set 0 has one used block and one free block) and store the tag 0000 for that block. It does not matter which set we use; for consistency, we put the data in the first set.
0x04	0000 01 00	Miss	We search set 1 for a block with the tag 0000, and on finding it missing, we copy the data from addresses 0x04, 0x05, 0x06, and 0x07 into set 1, and store the tag 0000 for that block.
0x09	0000 10 01	Miss	We search set 2 (10) for a block with the tag 0000, but we don't find one, so we copy the data from addresses 0x08, 0x09, 0x0A, and 0x0B into set 2 and store the tag 0000 for that block.
0x05	0000 01 01	Hit	We search set 1 for a block with the tag 0000, and we find it. We then use the offset value 01 within that block to get the exact byte we need.
0x14	0001 01 00	Miss	We search set 1 for a block with the tag 0001, but it is not present. We copy addresses 0x14, 0x15, 0x16, and 0x17 to set 1 and store the tag 0001 with that block. Note that set 1 is now full.
0x21	0010 00 01	Miss	We search cache set 0 for a block with the tag 0010; we don't find it, so we copy the data from addresses 0x20, 0x21, 0x22, and 0x23 into set 0 and store the tag 0010. Note that set 0 is now full.
0x01	0000 00 01	Hit	We search cache set 0 for a block with the tag 0000, and we find it. We use the offset of 1 within that block to find the data we want.

6.4 Cache Memory

- 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.

6.4 Cache Memory

- The **replacement policy** that we choose **depends on** 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 assessed 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.

6.4 Cache Memory

- ***First-in, first-out*** **(FIFO)** is a popular cache replacement policy.
- the **block that has been in the cache the longest will be evicted**, 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.

6.4 Cache Memory

- 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:

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

where **H** is the **cache hit rate** and **Access**$_C$ and **Access**$_{MM}$ are the **access times** for **cache** and **main memory**, respectively.

6.4 Cache Memory

- 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%**.
- Suppose **access** to cache and main memory **occurs concurrently**. (The accesses overlap.)
- The EAT is:

0.99(10ns) + 0.01(200ns) = 9.9ns + 2ns = 11.9ns

6.4 Cache Memory

- 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%.**
- **If the accesses do not overlap**, the EAT is:

$$\mathbf{0.99(10ns) + 0.01(10ns + 200ns)}$$
$$\mathbf{= 9.9ns + 2.1ns = 12ns}$$

- The equation for determining the effective access time can be extended to any number of memory levels

6.4 Cache Memory

- Most of today's small systems employ **multilevel cache** hierarchies.
- The levels of cache form their own small memory hierarchy.
- **Level1** cache (8KB to 64KB) is situated on the processor itself.
  - Access time is typically about **4ns**.
- **Level 2** cache (64KB to 2MB) may be on the motherboard, or on an expansion card.
  - Access time is usually around **15ns**.

6.4 Cache Memory

- **Cache replacement** policies must take into account **dirty blocks**, those blocks that have been **updated while they were in the cache**.
- **Dirty blocks must be written back to memory**. A **write policy** determines how this will be done.
- There are two types of write policies, **write through** and **write back**.
- **Write through updates cache and main memory simultaneously on every write.**

6.4 Cache Memory

- **Write back updates memory only when the block is selected for replacement.**
- The **disadvantage of write through** is that **memory must be updated with each cache write**, which **slows down the access time** on updates.

- The **advantage of write back** is that **memory traffic is minimized**, but its **disadvantage** is that **memory does not always agree with the value in cache**, causing problems in systems with many concurrent users.