

THE ESSENTIALS OF

# Computer Organization *and* Architecture

THIRD EDITION

Linda Null  
Julia Lobur

## Chapter 6 Memory

# Chapter 6 Objectives

- Master the concepts of hierarchical memory organization.
- Understand how each level of memory contributes to system performance, and how the performance is measured.
- Master the concepts behind cache memory, virtual memory, memory segmentation, paging and address translation.

# 6.1 Introduction

- Memory lies at the heart of the stored-program computer.
- In previous chapters, we studied the components from which memory is built and the ways in which memory is accessed by various ISAs.
- In this chapter, we focus on memory organization. A clear understanding of these ideas is essential for the analysis of system performance.

## 6.2 Types of Memory

- There are two kinds of main memory: *random access memory, RAM, and read-only-memory, ROM.*
- There are two types of RAM, dynamic RAM (DRAM) and static RAM (SRAM).
- DRAM consists of capacitors that slowly leak their charge over time. Thus, they must be refreshed every few milliseconds to prevent data loss.
- DRAM is “cheap” memory owing to its simple design.

## 6.2 Types of Memory

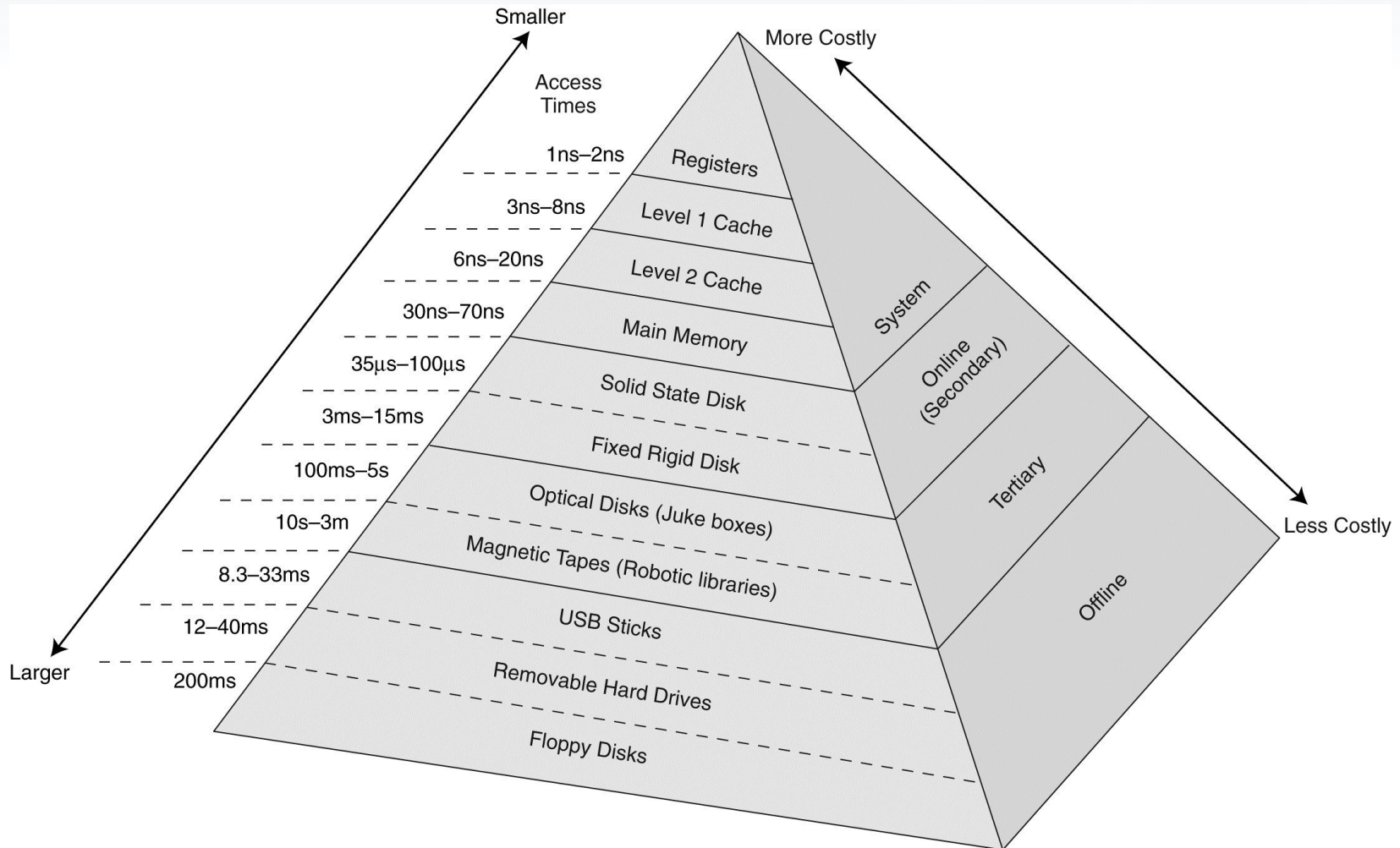
- SRAM consists of circuits similar to the D flip-flop that we studied in Chapter 3.
- SRAM is very fast memory and it doesn't need to be refreshed like DRAM does. It is used to build cache memory, which we will discuss in detail later.
- ROM also does not need to be refreshed, either. In fact, it needs very little charge to retain its memory.
- ROM is used to store permanent, or semi-permanent data that persists even while the system is turned off.

## 6.3 The Memory Hierarchy

- Generally speaking, faster memory is more expensive than slower memory.
- To provide the best performance at the lowest cost, memory is organized in a hierarchical fashion.
- Small, fast storage elements are kept in the CPU, larger, slower main memory is accessed through the data bus.
- Larger, (almost) permanent storage in the form of disk and tape drives is still further from the CPU.

## 6.3 The Memory Hierarchy

- This storage organization can be thought of as a pyramid:



## 6.3 The Memory Hierarchy

- We are most interested in the memory hierarchy that involves registers, cache, main memory, and virtual memory.
- Registers are storage locations available on the processor itself.
- Virtual memory is typically implemented using a hard drive; it extends the address space from RAM to the hard drive.
- Virtual memory provides more space: Cache memory provides speed.



## 6.3 The Memory Hierarchy

- To access a particular piece of data, the CPU first sends a request to its nearest memory, usually cache.
- If the data is not in cache, then main memory is queried. If the data is not in main memory, then the request goes to disk.
- Once the data is located, then the data, and a number of its nearby data elements are fetched into cache memory.

## 6.3 The Memory Hierarchy

- This leads us to some definitions.
  - 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.
  - $\text{Miss rate} = 1 - \text{hit rate}$ .
  - 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.

## 6.3 The Memory Hierarchy

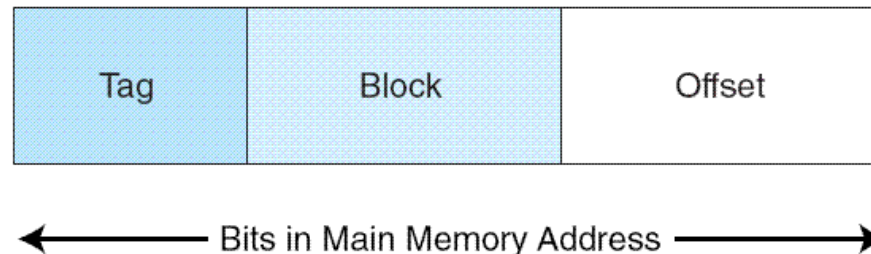
- An entire blocks of data is copied after a hit because the *principle of locality* tells us that once a byte is accessed, it is likely that a nearby data element will be needed soon.
- There are three forms of locality:
  - *Temporal locality*- Recently-accessed data elements tend to be accessed again.
  - *Spatial locality* - Accesses tend to cluster.
  - *Sequential locality* - Instructions tend to be accessed sequentially.

## 6.4 Cache Memory

- The purpose of cache memory is to speed up accesses by storing recently used data closer to the CPU, instead of storing it in main memory.
- Although cache is much smaller than main memory, its access time is a fraction of that of main memory.
- Unlike main memory, which is accessed by address, cache is typically accessed by content; hence, it is often called *content addressable memory*.
- Because of this, a single large cache memory isn't always desirable-- it takes longer to search.

## 6.4 Cache Memory

- The “content” that is addressed in content addressable cache memory is a subset of the bits of a main memory address called a *field*.
  - Many blocks of main memory map to a single block of cache. A *tag field* in the cache block distinguishes one cached memory block from another.
  - A *valid bit* indicates whether the cache block is being used.
  - An *offset field* points to the desired data in the block.



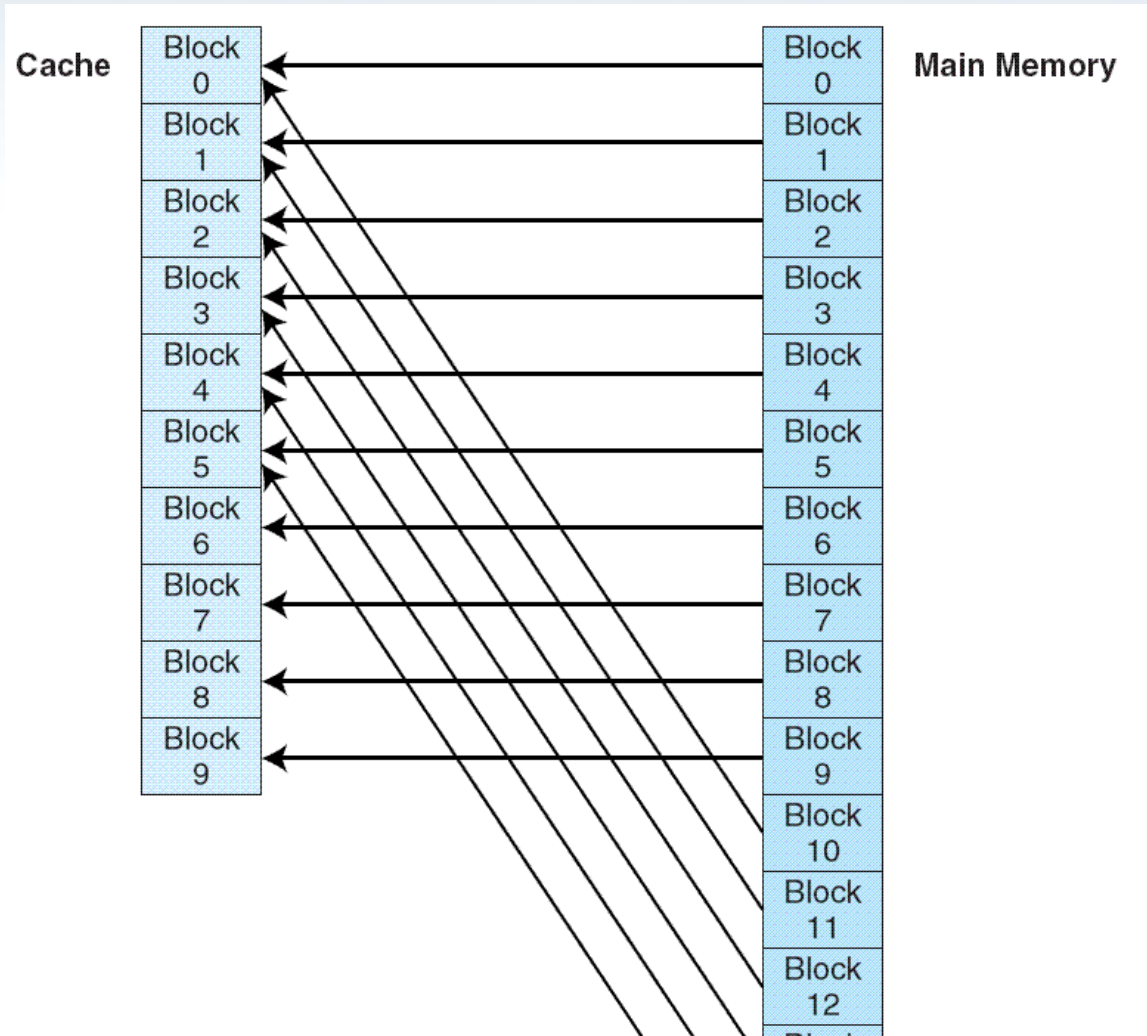
## 6.4 Cache Memory

- The **simplest** cache mapping scheme is ***direct mapped cache***.
- In a **direct mapped cache** consisting of  $N$  blocks of cache, block  $X$  of main memory maps to cache block  $Y = X \bmod N$ .
- Thus, if we have **10 blocks of cache**, block 7 **of cache** may hold blocks **7, 17, 27, 37, . . .** **of main memory**.


**The next slide illustrates this mapping.**

## 6.4 Cache Memory

- With direct mapped cache consisting of  $N$  blocks of cache, block  $X$  of main memory maps to cache block  $Y = X \bmod N$ .



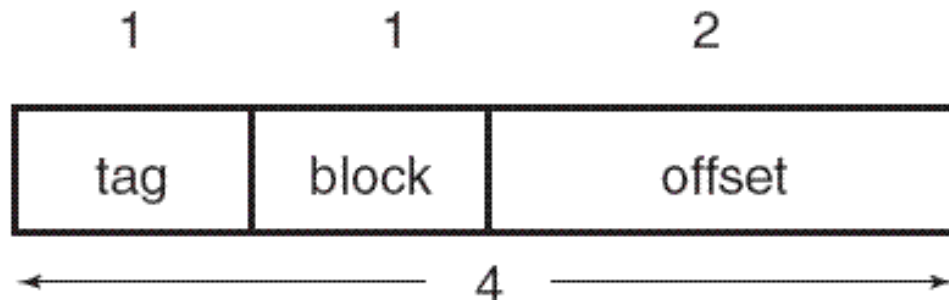
## 6.4 Cache Memory

- **EXAMPLE 6.1** Consider a word-addressable main memory consisting of four blocks, and a cache with two blocks, where each block is 4 words.
- This means Block 0 and 2 of main memory map to Block 0 of cache, and Blocks 1 and 3 of main memory map to Block 1 of cache. 
- Using the tag, block, and offset fields, we can see how main memory maps to cache as follows.



## 6.4 Cache Memory

- **EXAMPLE 6.1** Consider a word-addressable main memory consisting of four blocks, and a cache with two blocks, where each block is 4 words.
  - First, we need to determine the address format for mapping. Each block is 4 words, so the offset field must contain 2 bits; there are 2 blocks in cache, so the block field must contain 1 bit; this leaves 1 bit for the tag (as a main memory address has 4 bits because there are a total of  $2^4=16$  words).

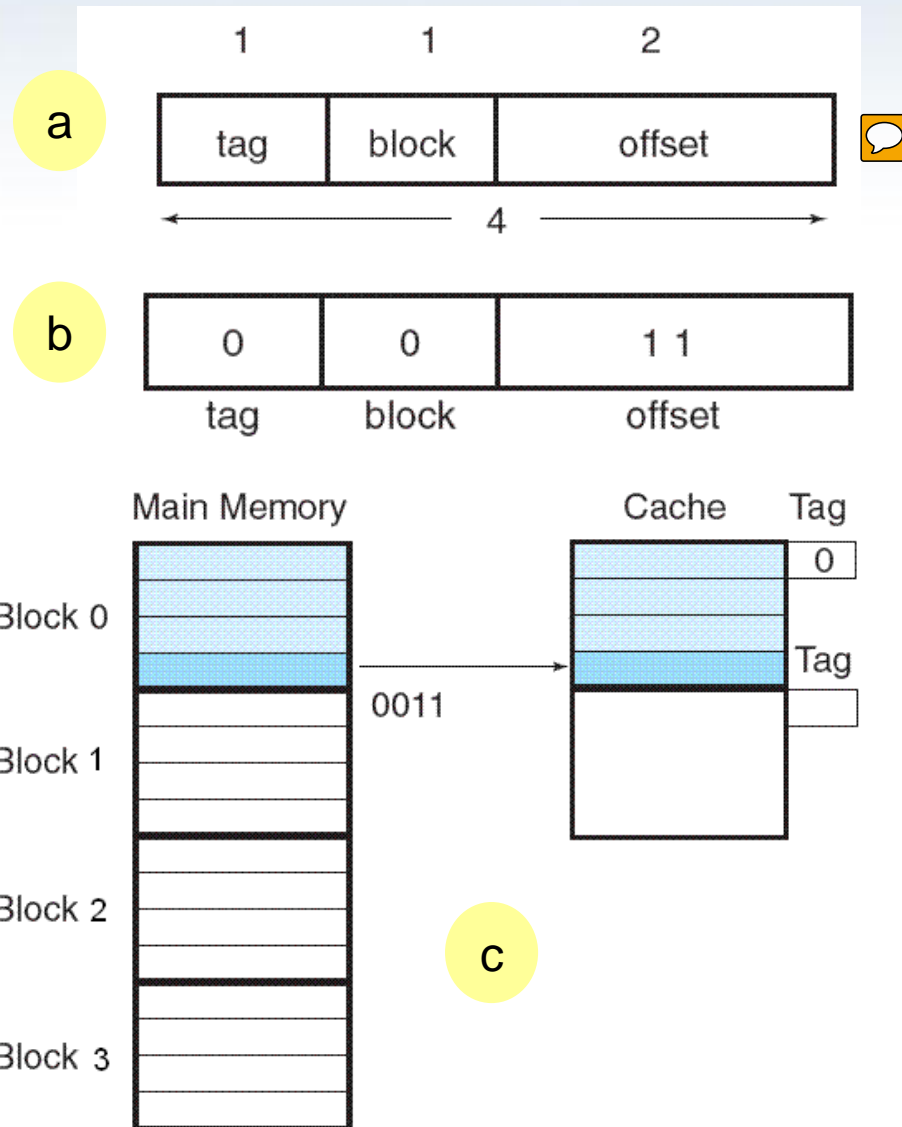


## 6.4 Cache Memory

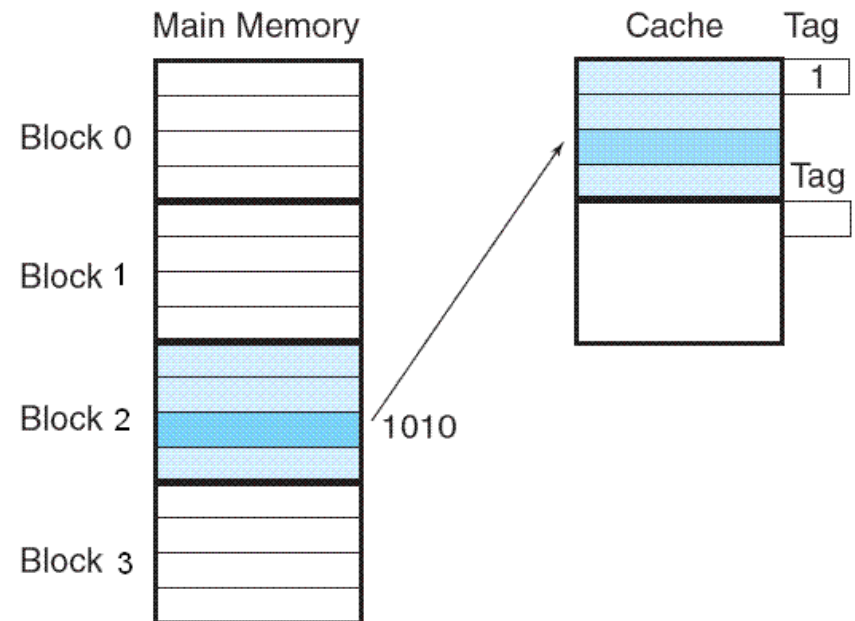
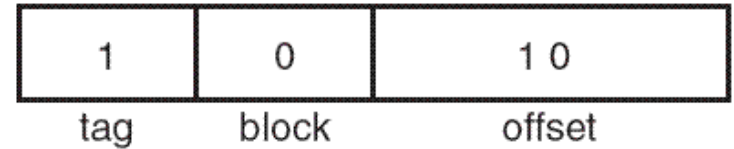
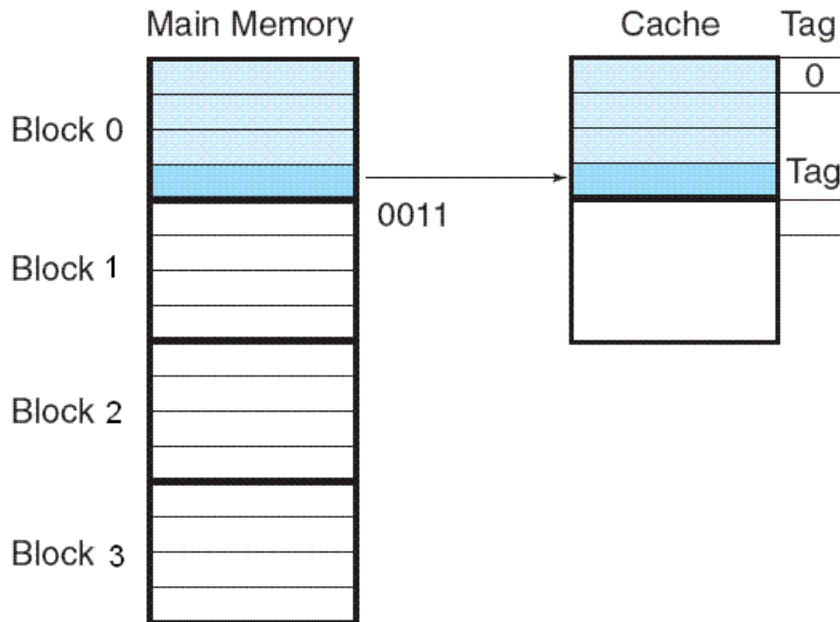
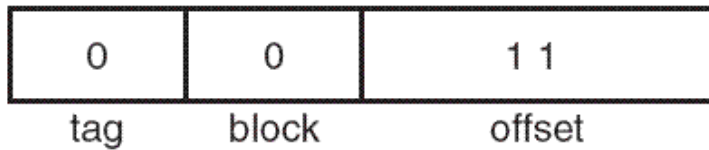
- EXAMPLE 6.1 Cont'd

- Suppose we need to access main memory address  $3_{16}$  (0011 in binary). If we partition 0011 using the address format from Figure a, we get Figure b.
- Thus, the main memory address 0011 maps to cache block 0.
- Figure c shows this mapping, along with the tag that is also stored with the data.

**The next slide illustrates another mapping.**

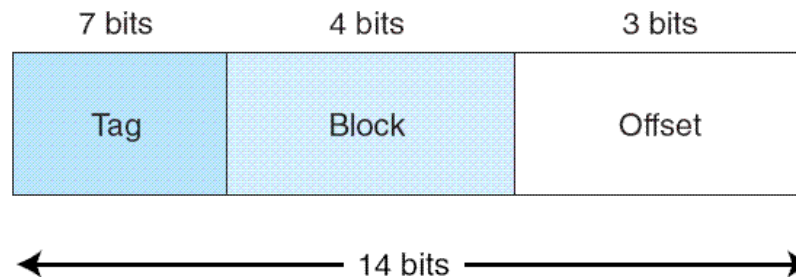


# 6.4 Cache Memory



## 6.4 Cache Memory

- **EXAMPLE 6.2** Assume a byte-addressable memory consists of  $2^{14}$  bytes, cache has 16 blocks, and each block has 8 bytes.  
$$\frac{2^{14}}{2^3} = 2^{11}$$
  - The number of **memory blocks** are:
  - Each main memory address requires 14 bits. Of this 14-bit address field, the **rightmost 3 bits reflect** the offset field
  - We need 4 bits to select a specific block in cache, so the block field consists of the middle 4 bits.
  - The remaining 7 bits make up the tag field.



## 6.4 Cache Memory

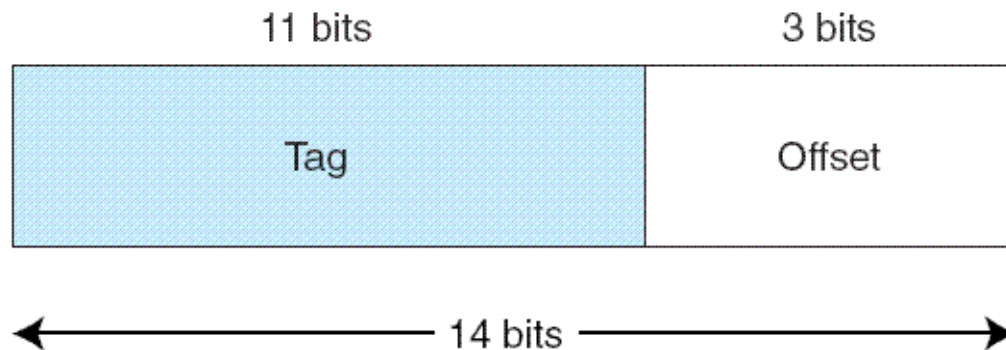
- In summary, direct mapped cache maps main memory blocks in a modular fashion to cache blocks. The mapping depends on:
- The number of bits in the main memory address (how many addresses exist in main memory)
- The number of blocks are in cache (which determines the size of the block field)
- How many addresses (either bytes or words) are in a block (which determines the size of the offset field)

## 6.4 Cache Memory

- Suppose 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, 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 word.

## 6.4 Cache Memory

- 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.

## 6.4 Cache Memory

- 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.

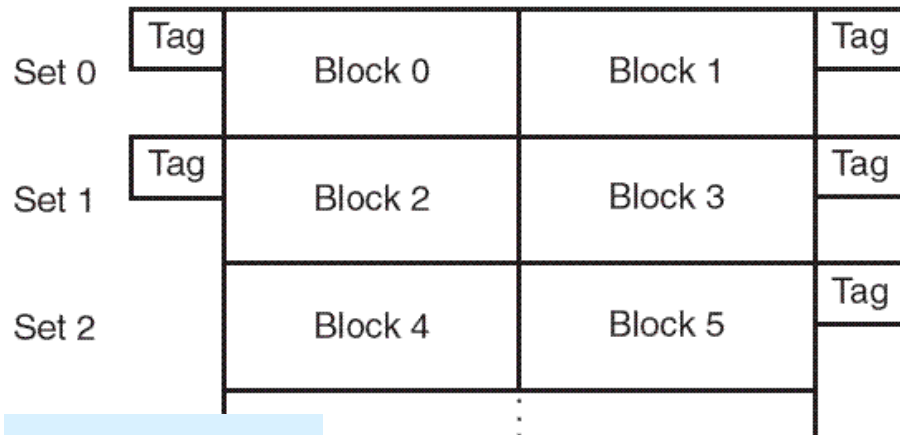


## 6.4 Cache Memory

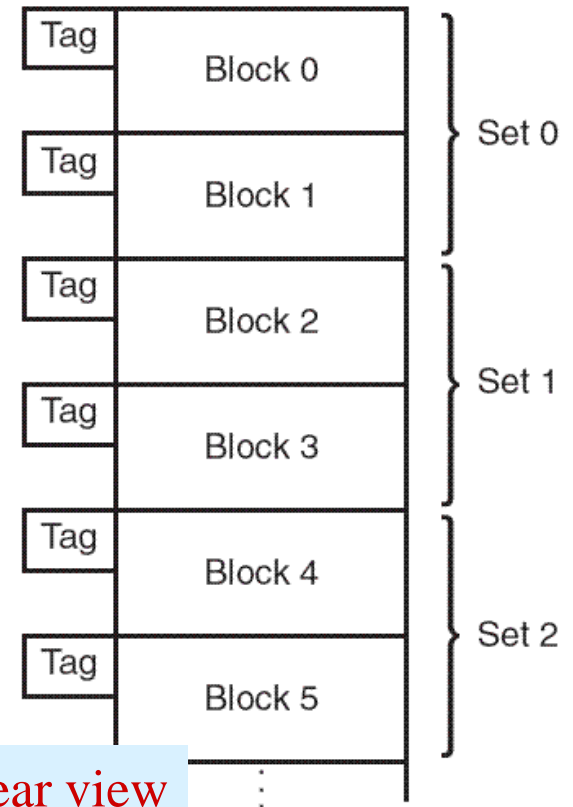
- Set associative cache combines the ideas of direct mapped cache and fully associative cache.
- An  $N$ -way set associative cache mapping is like direct mapped cache in that a memory reference maps to a particular location in cache.
- Unlike direct mapped cache, a memory reference maps to a set of 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



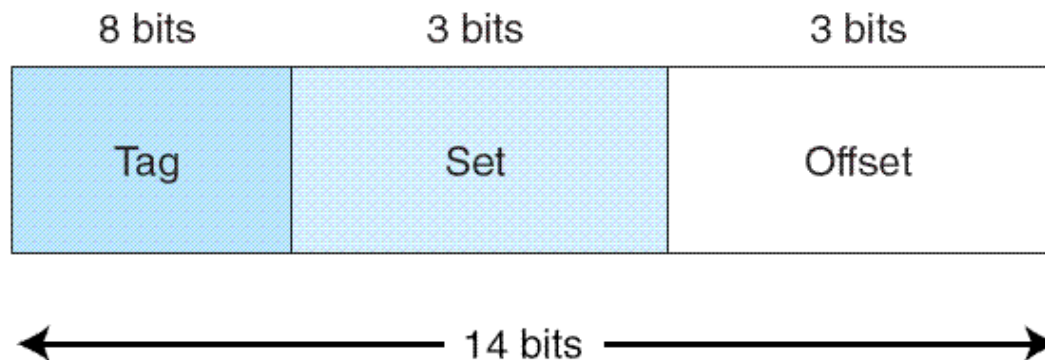
Linear view

## 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 chooses the word within the cache block, and 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.
  - Cache has a total of 16 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

- 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 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 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.
- 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.

## 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:

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

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



## 6.4 Cache Memory

- 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%.
- Suppose access to cache and main memory occurs concurrently. (The accesses overlap.)
- The EAT is:  
$$0.99(10\text{ns}) + 0.01(200\text{ns}) = 9.9\text{ns} + 2\text{ns} = 11\text{ns}.$$

## 6.4 Cache Memory

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

$$\begin{aligned} 0.99(10\text{ns}) + 0.01(10\text{ns} + 200\text{ns}) \\ = 9.9\text{ns} + 2.01\text{ns} = 12\text{ns}. \end{aligned}$$

- This equation for determining the effective access time can be extended to any number of memory levels, as we will see in later sections.

## 6.4 Cache Memory

- Caching is depends upon programs exhibiting good locality.
  - Some object-oriented programs have poor locality owing to their complex, dynamic structures.
  - Arrays stored in column-major rather than row-major order can be problematic for certain cache organizations.
- With poor locality, caching can actually cause performance degradation rather than performance improvement.

## 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** (also called *copyback*) **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. This **slowdown is usually negligible**, because the majority of accesses tend to be reads, not writes.
- 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.

## 6.4 Cache Memory

- The cache we have been discussing is called a *unified or integrated cache* where both instructions and data are cached.
- Many modern systems employ separate caches for data and instructions.
  - This is called a *Harvard cache*.
- The separation of data from instructions provides better locality, at the cost of greater complexity.
  - Simply making the cache larger provides about the same performance improvement without the complexity.

## 6.4 Cache Memory

- Cache performance can also be improved by adding a small associative cache to hold blocks that have been evicted recently.
  - This is called a *victim cache*.
- A *trace cache* is a variant of an instruction cache that holds decoded instructions for program branches, giving the illusion that noncontiguous instructions are really contiguous.

## 6.4 Cache Memory

- Most of today's small systems employ multilevel cache hierarchies.
- The levels of cache form their own small memory hierarchy.
- Level 1 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 15 - 20ns.



## 6.4 Cache Memory

- In systems that employ three levels of cache, the Level 2 cache is placed on the same die as the CPU (reducing access time to about 10ns)
- Accordingly, the Level 3 cache (2MB to 256MB) refers to cache that is situated between the processor and main memory.
- Once the number of cache levels is determined, the next thing to consider is whether data (or instructions) can exist in more than one cache level.

## 6.4 Cache Memory

- If the cache system used an *inclusive* cache, the same data may be present at multiple levels of cache.
- *Strictly inclusive* caches guarantee that all data in a smaller cache also exists at the next higher level.
- *Exclusive* caches permit only one copy of the data.
- The tradeoffs in choosing one over the other involve weighing the variables of access time, memory size, and circuit complexity.

## 6.5 Virtual Memory

- Cache memory enhances performance by providing faster memory access speed.
- Virtual memory enhances performance by providing greater memory capacity, without the expense of adding main memory.
- Instead, a portion of a disk drive serves as an extension of main memory.
- If a system uses paging, virtual memory partitions main memory into individually managed *page frames*, that are written (*or paged*) to disk when they are not immediately needed.

## 6.5 Virtual Memory

- A *physical address* is the actual memory address of physical memory.
- **Programs create** *virtual addresses* that are *mapped* to physical addresses by the memory manager.
- **Page faults occur** when a logical address requires that a page be brought in from disk.
- **Memory fragmentation occurs** when the paging process results in the creation of small, unusable clusters of memory addresses.

## 6.5 Virtual Memory

- Main memory and virtual memory are divided into equal sized pages.
- The entire address space required by a process need not be in memory at once. Some parts can be on disk, while others are in main memory.
- Further, the pages allocated to a process do not need to be stored contiguously-- either on disk or in memory.
- In this way, only the needed pages are in memory at any time, the unnecessary pages are in slower disk storage.

## 6.5 Virtual Memory

- Information concerning the location of each page, whether on **disk** or in **memory**, is maintained in a data structure called a **page table** (shown below).
- There is one page table for each active process.

