the essentials of

Computer Organization and Architecture

Linda Null and 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:
  - Cache memory,
  - Virtual memory,
  - Memory segmentation, paging and

6.1 Introduction

- Memory lies at the center of the **stored-program** computer.
- In previous chapter, we studied 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

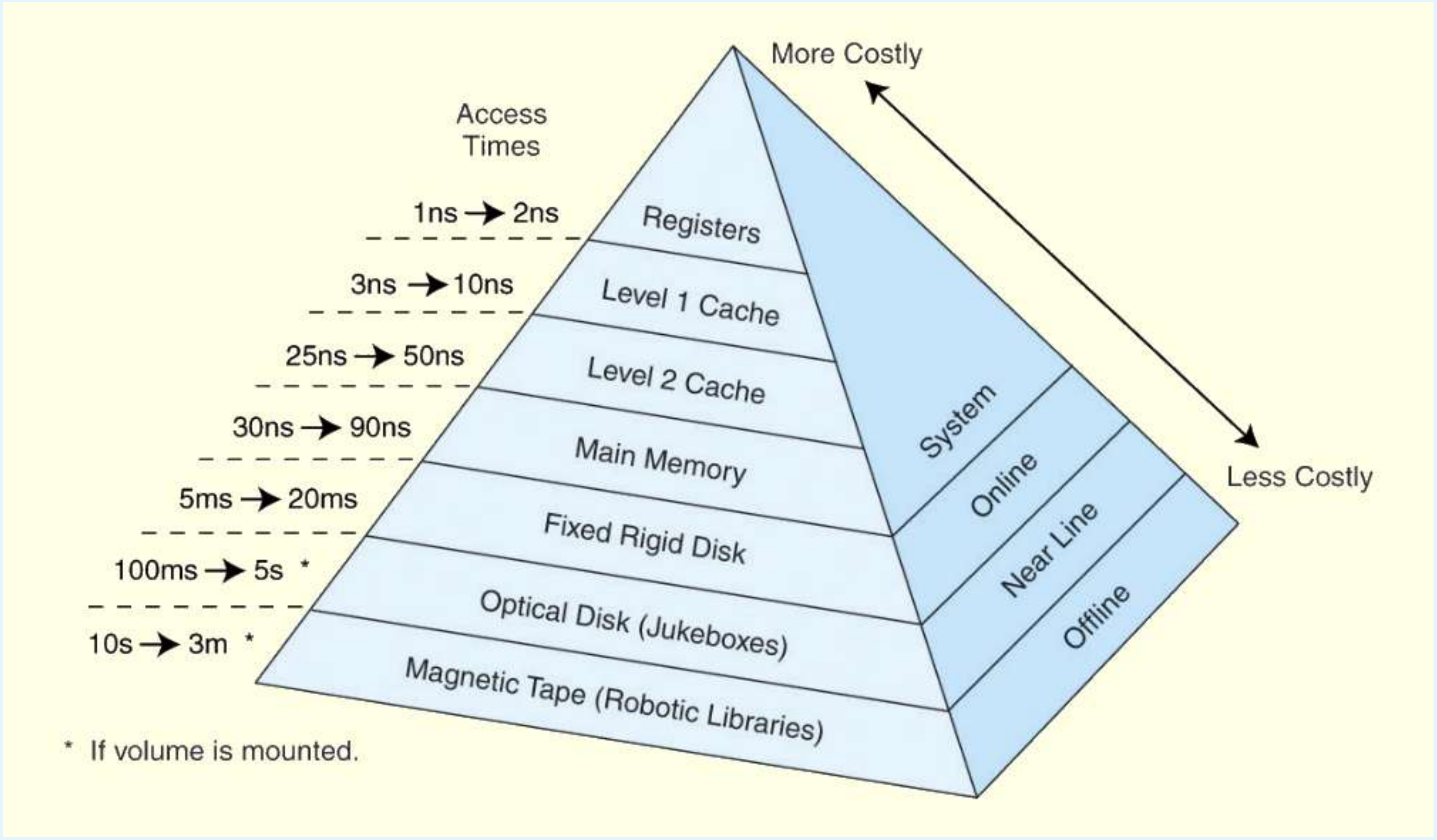
- There are two kinds of main memory:
  - *Random access memory, **RAM**, and*
  - *Read-only-memory, **ROM***.
- There are two types of **RAM**,
  - Static RAM (**SRAM**).
    - flip-flop, very fast, cache memory
  - Dynamic RAM (**DRAM**)
    - capacitors, refresh, slow, simple design, cheap
    - S - DRAM : Synchronous DRAM
    - DR - DRAM : Direct Rambus DRAM
    - DDR - DRAM : Double Data Rate DRAM

6.2 Types of Memory

- **ROM** is used to store permanent data that persists even while the system is turned off (Non-Volatile).
- Types of **ROM**
  - **ROM**
  - **PROM**
  - **EPROM**
  - **EEPROM**
  - **Flash memory**

6.3 The Memory Hierarchy

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

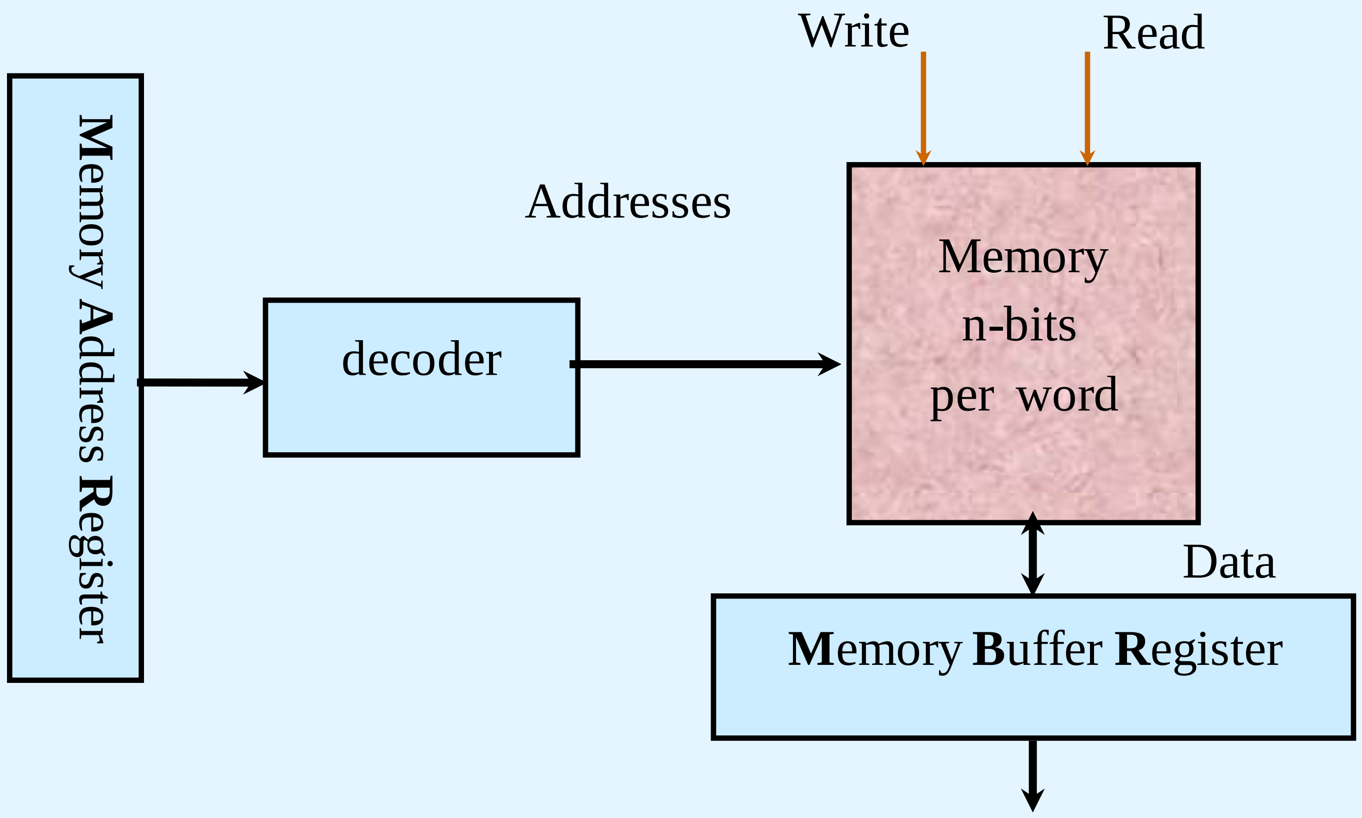


6.3 The Memory Hierarchy

- 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

- To access a particular piece of data,
  The CPU sends a request to the memory



6.3 The Memory Hierarchy

- If

  The data is not in cache,

  Then

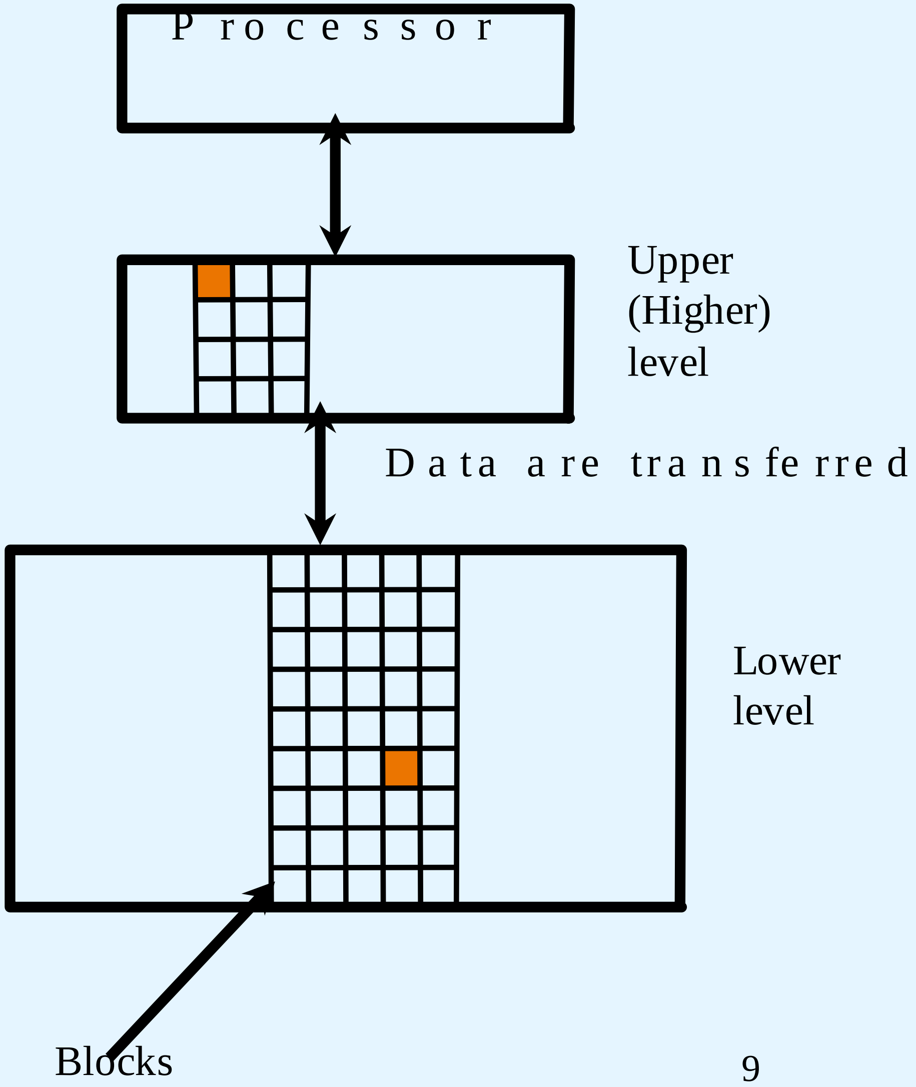
  Main memory is queried.

- If

  The data is not in main memory,

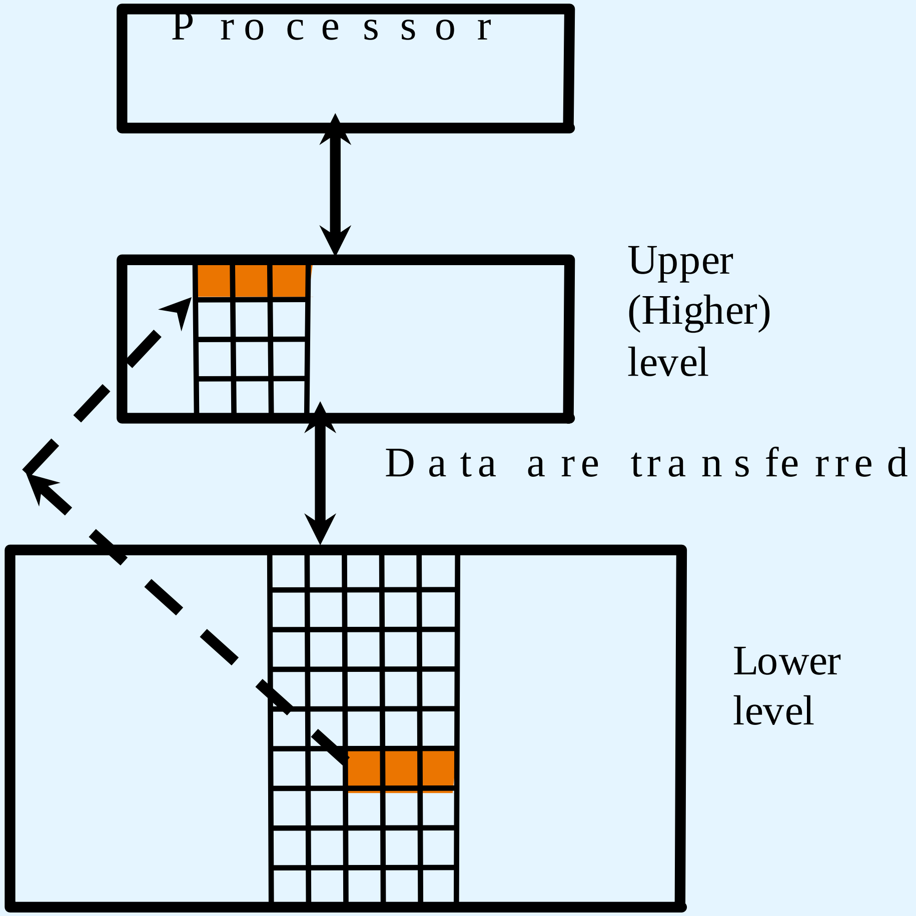
  Then

  The request goes to disk



6.3 The Memory Hierarchy

- 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

- locality of reference (the principle of locality):
  - The same value or related storage locations being **frequently accessed**.

- 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.

6.3 The Memory Hierarchy

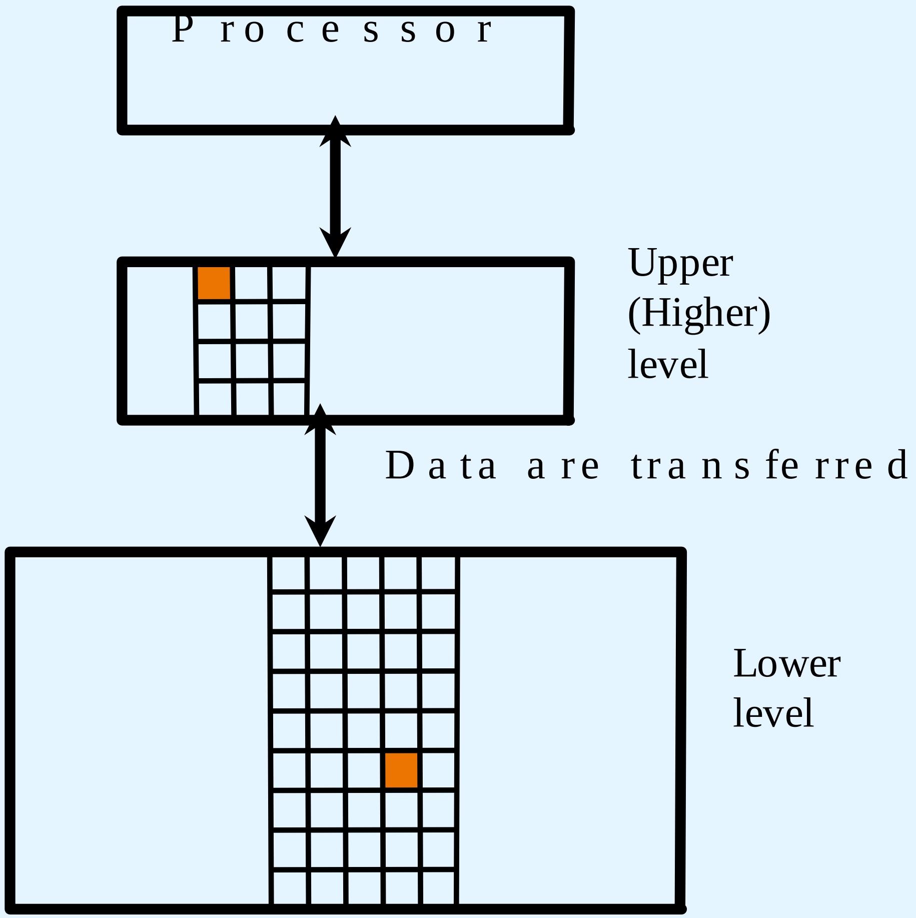
100	Load	109
101	Add	10A
102	Store	10B
103	Jns	200
104	Load	10C
105	Subt	10A
106	Skipcond	400
107	Jump	104
108	Halt	
109	17	
10A	1	
10B	0	
10C	5	

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.
  - Miss rate = 1 - 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

- Definitions.
  - ***Hit***
  - ***Miss***
  - ***Hit rate***
  - ***Miss rate***
  - Miss rate = 1 - hit rate.
  - ***Hit time***
  - ***Miss penalty***



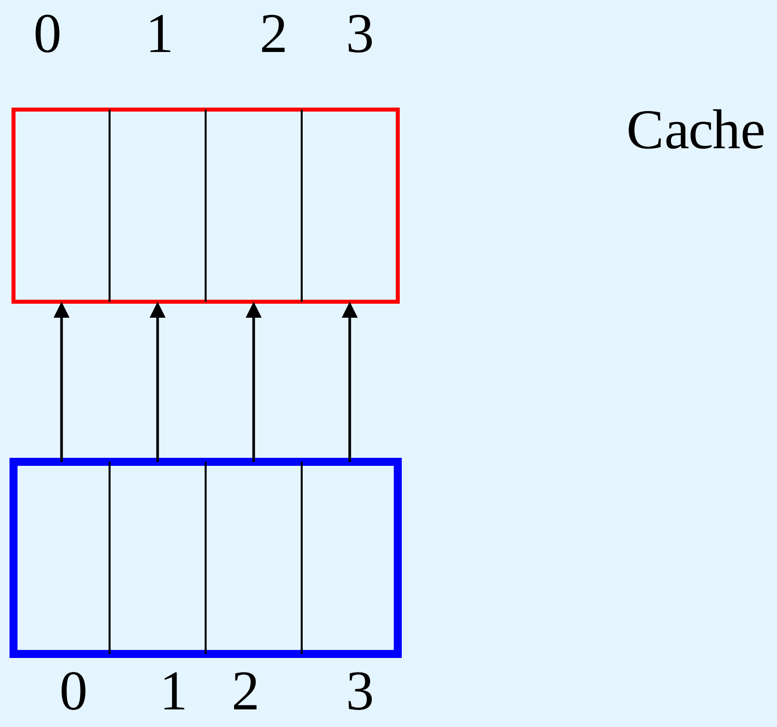
6.4 Cache Memory

- 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* mod *N*.
- Thus, if we have 10 blocks of cache, block 7 of cache may hold blocks 7, 17, 27, 37, . . . of main memory.
- Once a block of memory is copied into its slot in cache, a *valid* bit is set for the cache block to let the system know that the block contains valid data.

What could happen without having a valid bit?

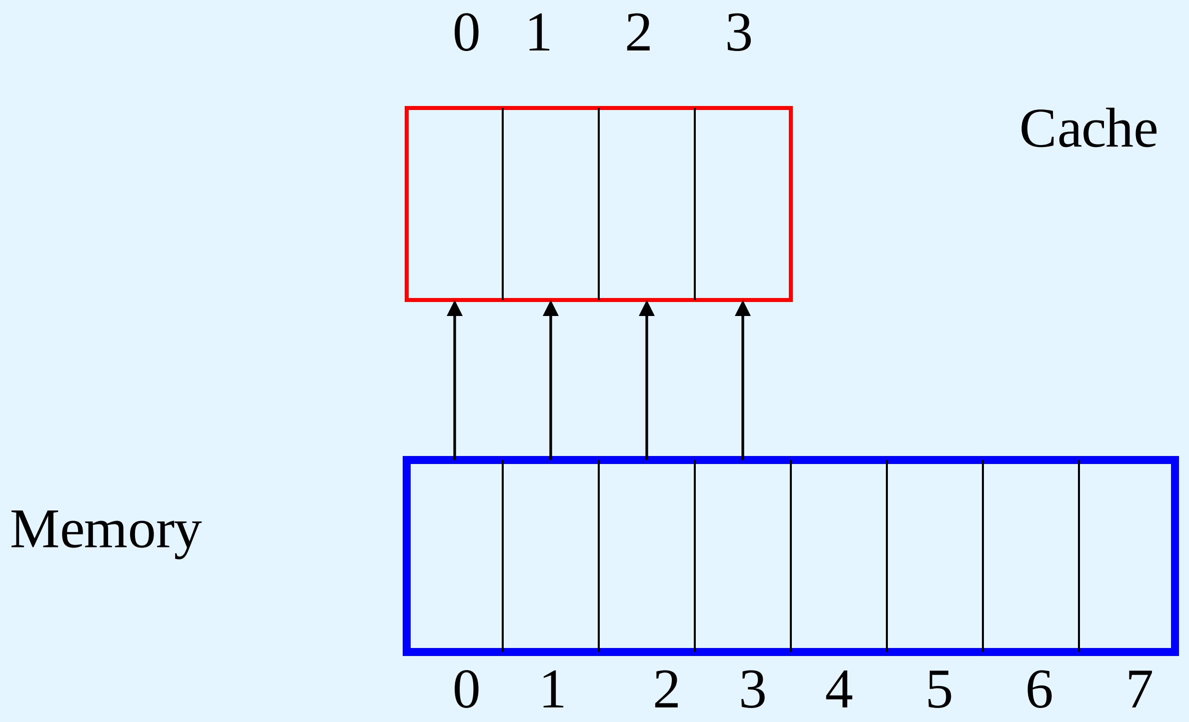
6.4 Cache Memory

- Example:
  - Block size is one word of data



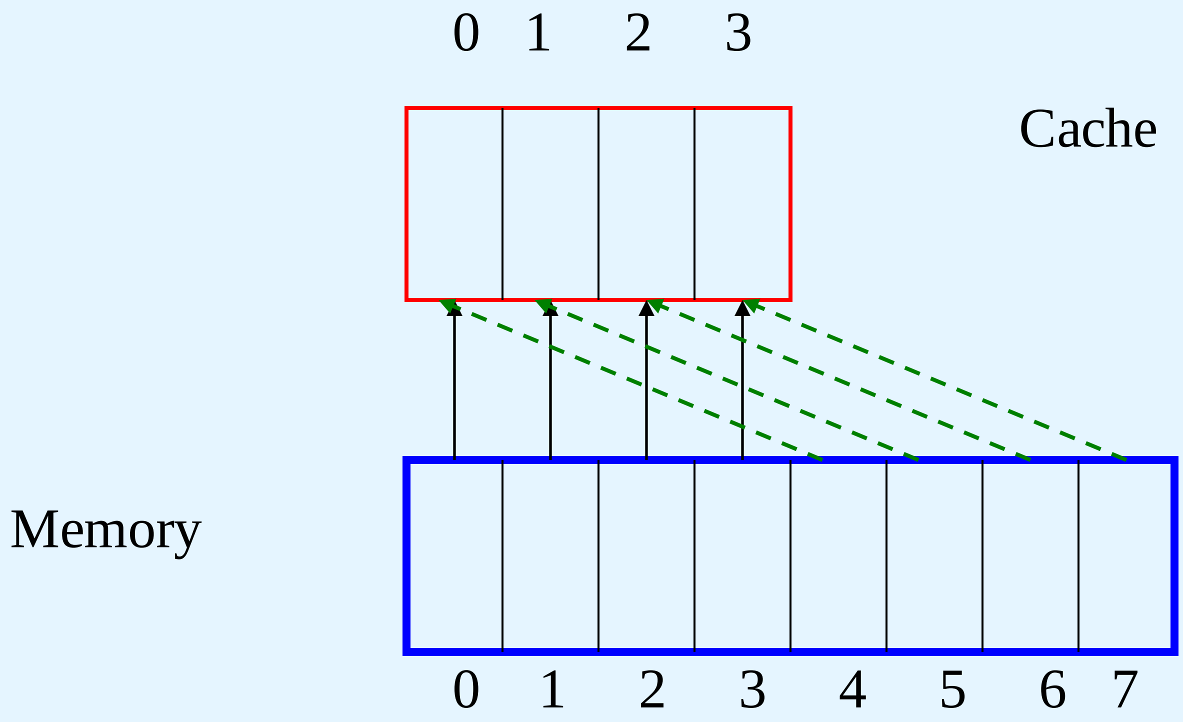
6.4 Cache Memory

- Example:
  - Block size is one word of data



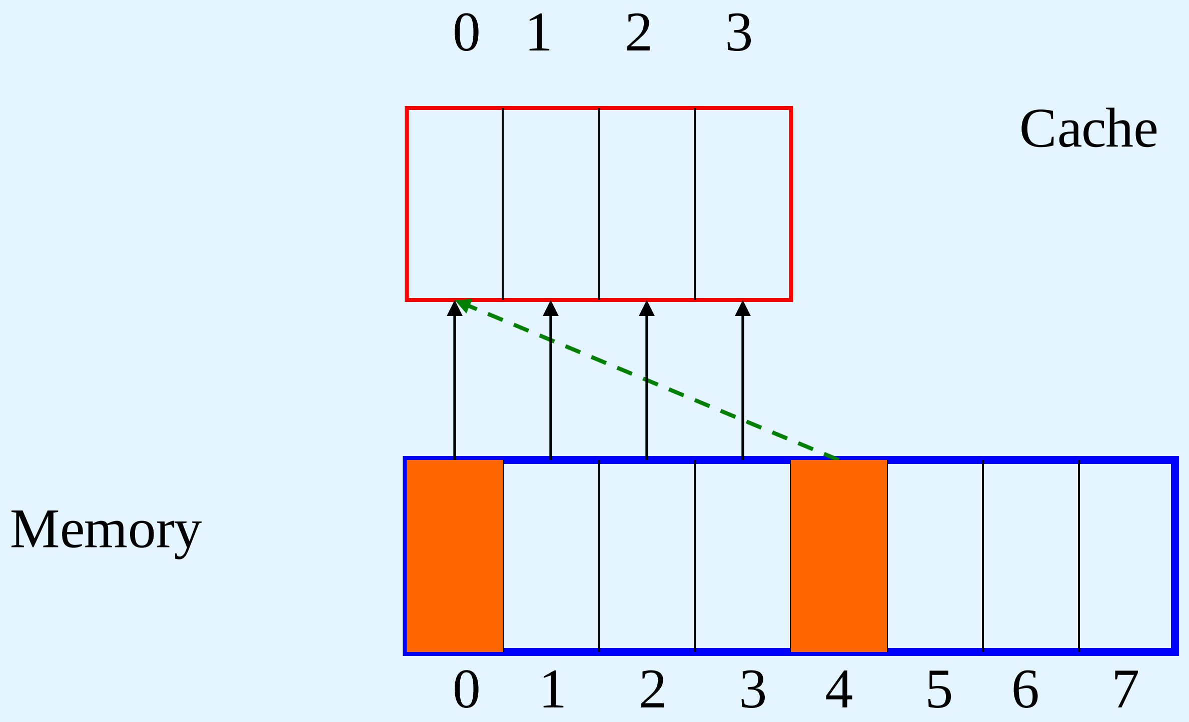
6.4 Cache Memory

- Example:
  - Block size is one word of data



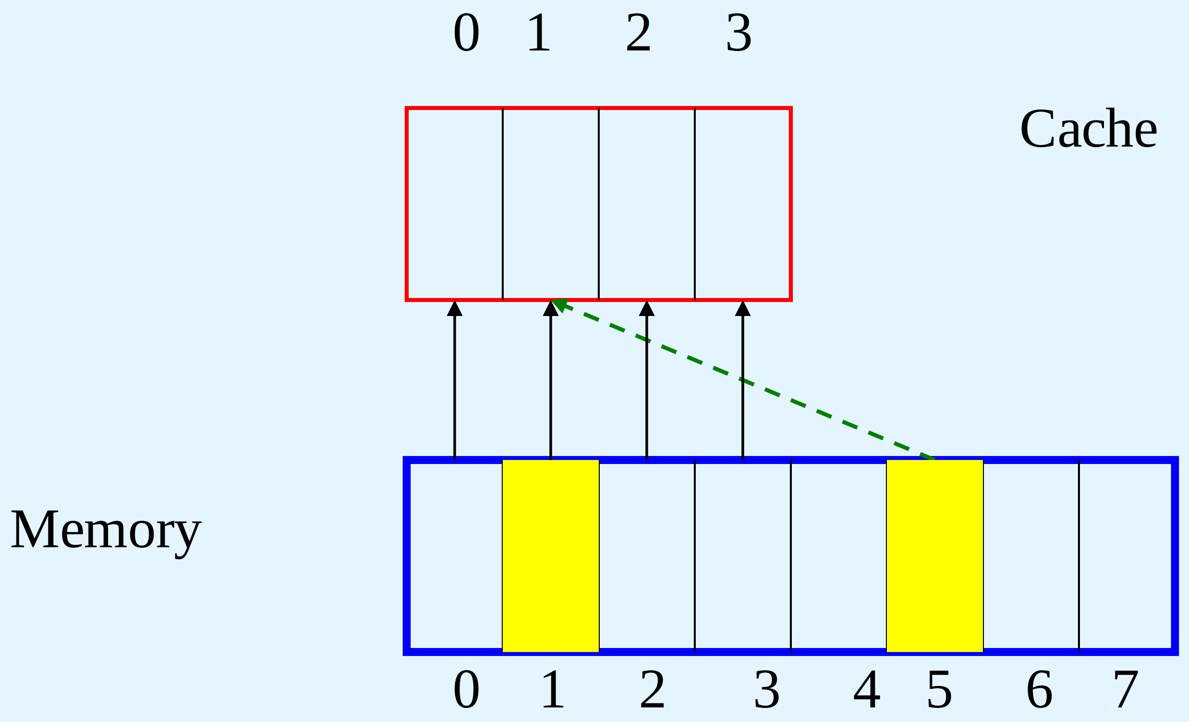
6.4 Cache Memory

- Example:
  - Block size is one word of data



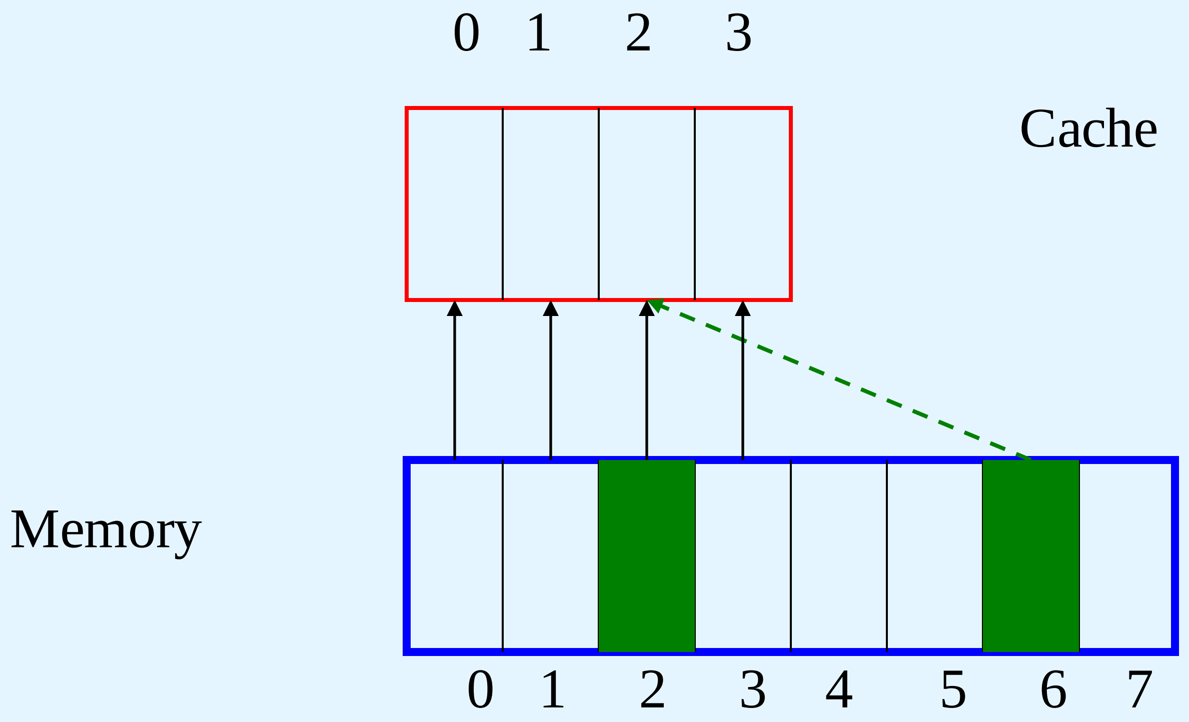
6.4 Cache Memory

- Example:
  - Block size is one word of data



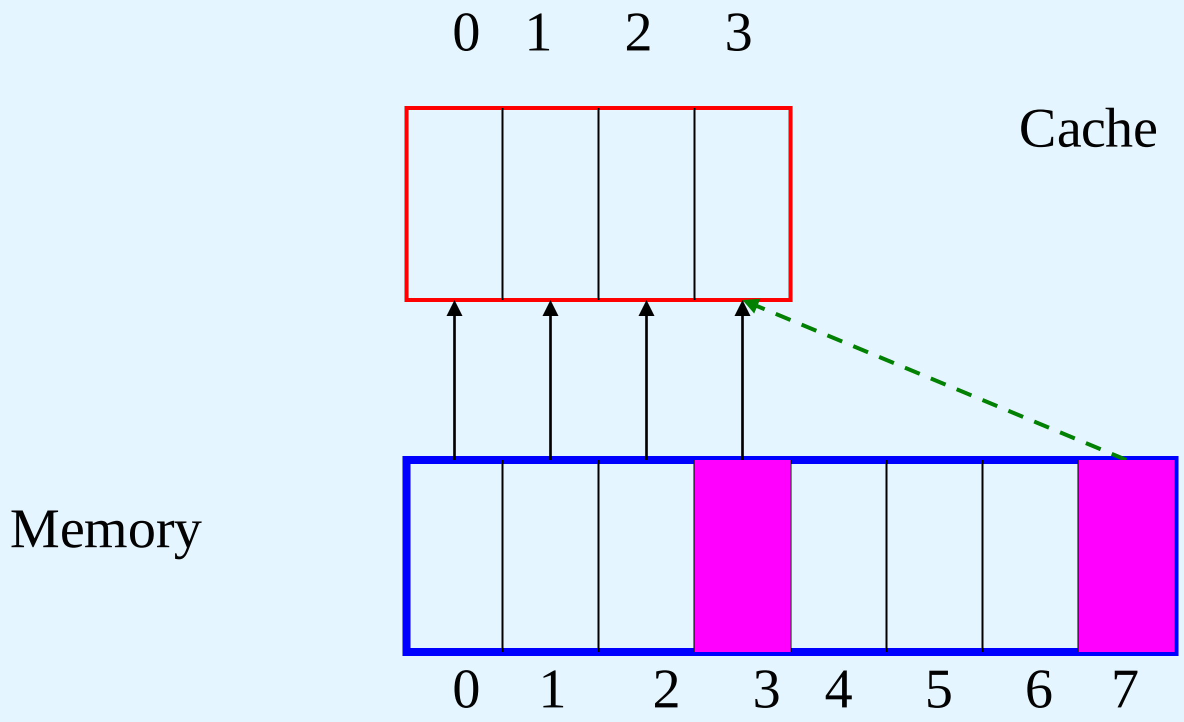
6.4 Cache Memory

- Example:
  - Block size is one word of data



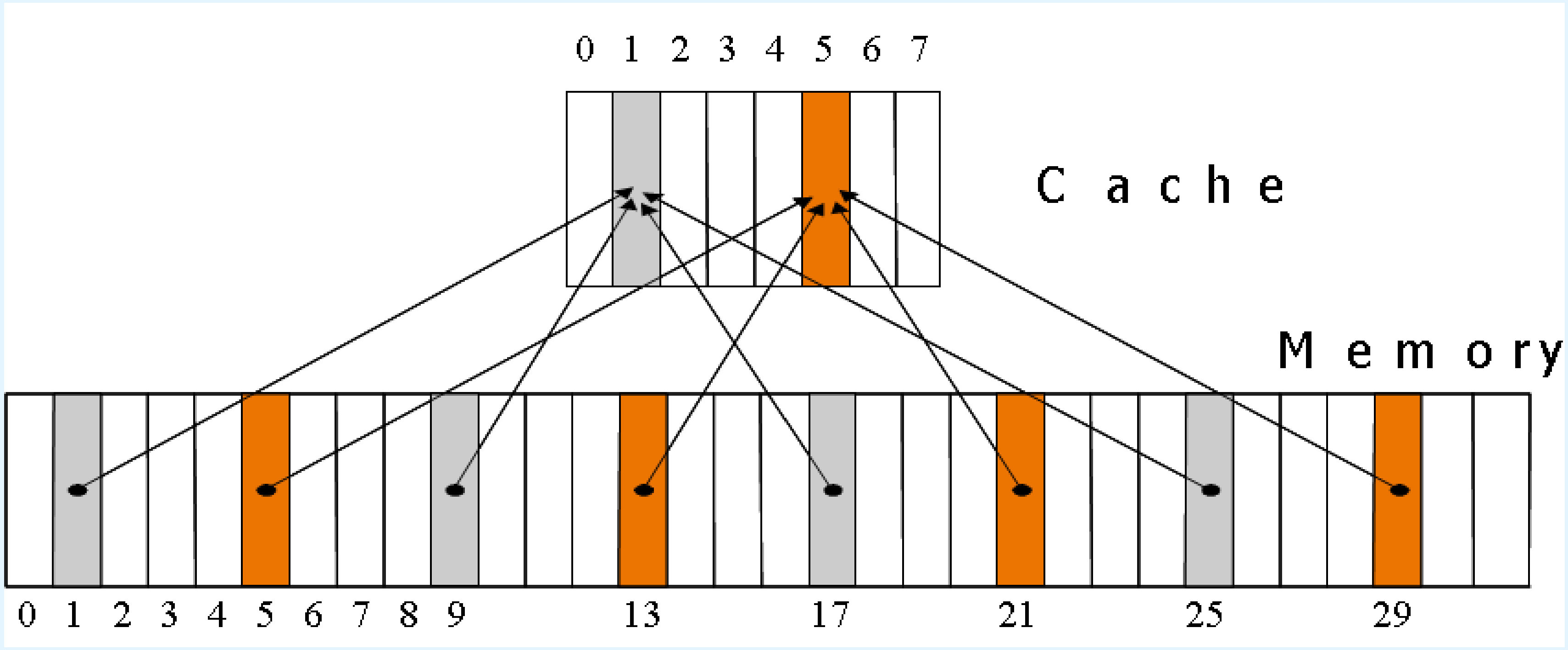
6.4 Cache Memory

- Example:
  - Block size is one word of data



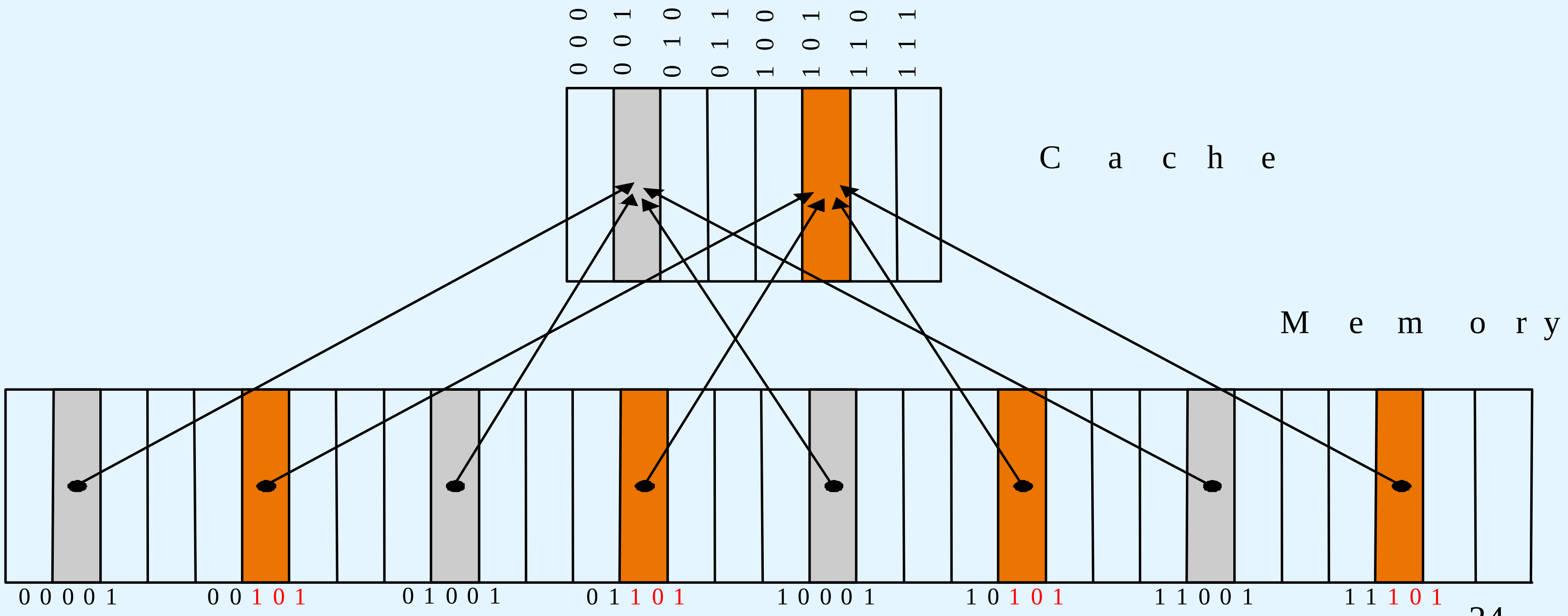
6.4 Cache Memory

- Mapping:
  - Block size is one word of data
  - Cache has 8 entries
  - Address is modulo the number of blocks in the cache



6.4 Cache Memory

- Mapping:
  - Block size is one word of data
  - Cache has 8 entries
  - Address is modulo the number of blocks in the cache



6.4 Cache Memory

- Problem:
  - How do we know whether the requested word is in the cache?

- Solution: Tag field
  - Added to data word in cache
  - Contain address information to where the data belongs
  - Need only contain the upper portion of the address

6.4 Cache Memory

- Problem:
  - How do we know that the block has the valid information?

- Solution: Valid bit
  - Added to data word in cache
  - 1 means data is valid (valid address)
  - 0 means data is invalid (Invalid address)

Direct Mapped Cache

- Example
  - Draw the cache after inserting the following blocks

    10110, 11010, 10000, 00011, 10010
- Initial Cache

Index	V	Tag	Data
000	0		
001	0		
010	0		
011	0		
100	0		
101	0		
110	0		
111	0		

Direct Mapped Cache

- After 10110

Index	V	Tag	Data
000	0		
001	0		
010	0		
011	0		
100	0		
101	0		
110	1	10	Memory(10110)
111	0		

- After 11010

Index	V	Tag	Data
000	0		
001	0		
010	1	11	Memory(11010)
011	0		
100	0		
101	0		
110	1	10	Memory(10110)
111	0		

Direct Mapped Cache

- After 10000

Index	V	Tag	Data
000	1	10	Memory(10000)
001	0		
010	1	11	Memory(11010)
011	0		
100	0		
101	0		
110	1	10	Memory(10110)
111	0		

- After 00011

Index	V	Tag	Data
000	1	10	Memory(10000)
001	0		
010	1	11	Memory(11010)
011	1	00	Memory(00011)
100	0		
101	N		
110	1	10	Memory(10110)
111	0		

Direct Mapped Cache

- After 10010

Index	V	Tag	Data
000	1	10	Memory(10000)
001	0		
010	1	10	Memory(10010)
011	1	00	Memory(00011)
100	0		
101	0		
110	1	10	Memory(10110)
111	0		

Direct Mapped Cache

- Check:

10001

00110

10010

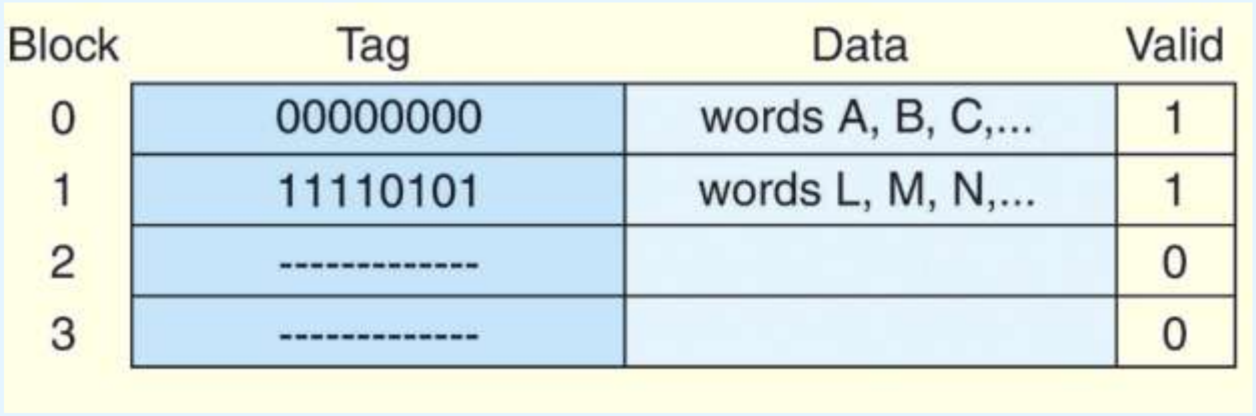
11111

01110

Index	V	Tag	Data
000	1	10	5120
001	0		
010	1	10	124A
011	1	00	74B0
100	0		
101	0		
110	1	10	381F
111	0		

Direct Mapped Cache

- The a schematic of what cache looks like.

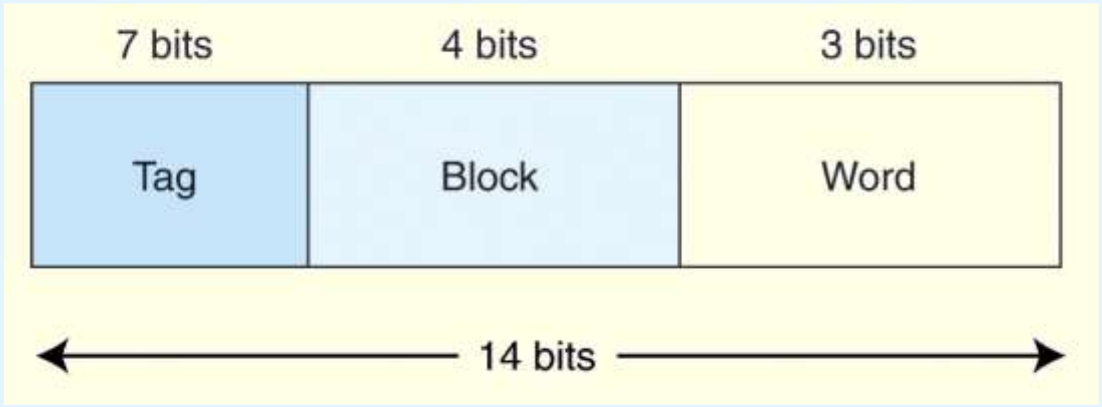


Block	Tag	Data	Valid
0	00000000	words A, B, C,...	1
1	11110101	words L, M, N,...	1
2	-------------		0
3	-------------		0

- Block 0 contains multiple words from main memory, identified with the tag 00000000.
- Block 1 contains words identified with the tag 11110101.
- The other two blocks are not valid.

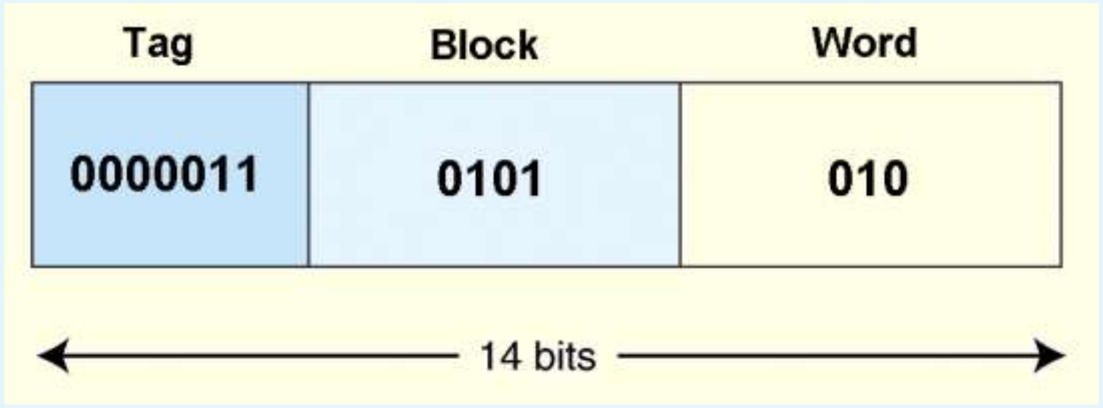
Direct Mapped Cache

- The size of each field into which a memory address is divided depends on the size of the cache.
- Suppose our memory consists of 2^{14} words, cache has $16 = 2^4$ blocks, and each block holds 8 words.
  - Thus memory is divided into $2^{14} / 2^{8} = 2^{11}$ blocks.
- For our field sizes, we need 4 bits for the block, 3 bits for the word, and the tag is what's left over:



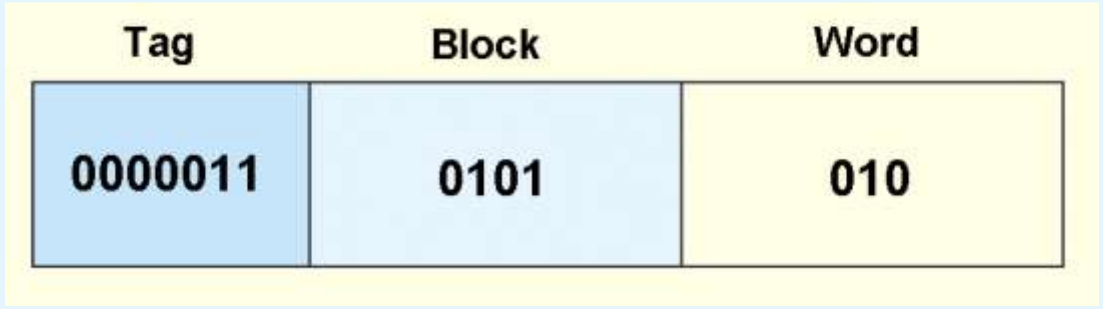
Direct Mapped Cache

- As an example, suppose a program generates the address **`1AA`**. In 14-bit binary, this number is: **`00000110101010`**.
- The first 7 bits of this address go in the tag field, the next 4 bits go in the block field, and the final 3 bits indicate the word within the block.



Direct Mapped Cache

- If subsequently the program generates the address **`1AB`**, it will find the data it is looking for in block **`0101`**, word **`011`**.



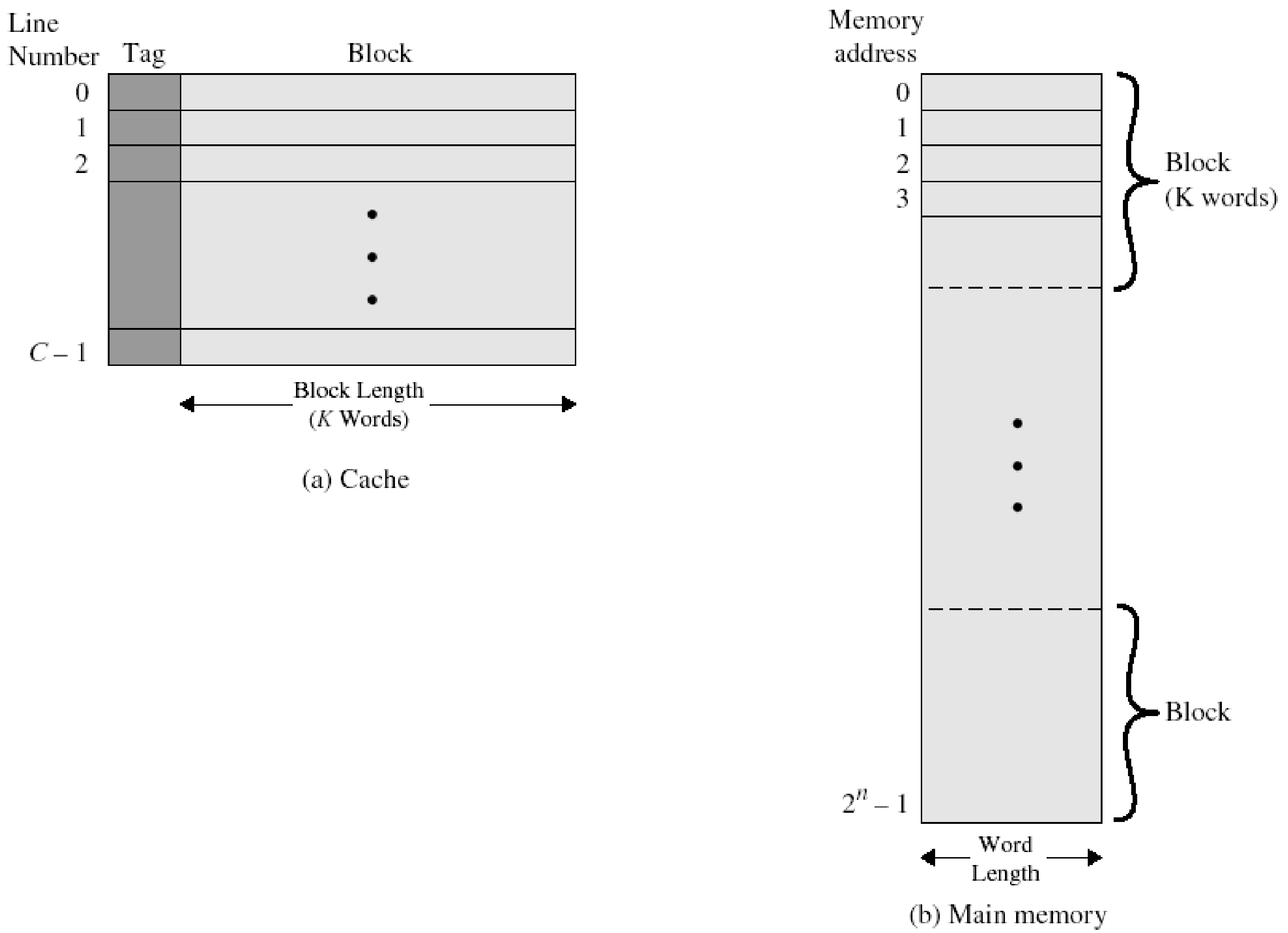
Tag	Block	Word
0000011	0101	010

- However, if the program generates the address, **`3AB`**, instead, the block loaded for address **`1AA`** would be evicted from the cache, and replaced by the blocks associated with the **`3AB`** reference.

Fully Associative Cache

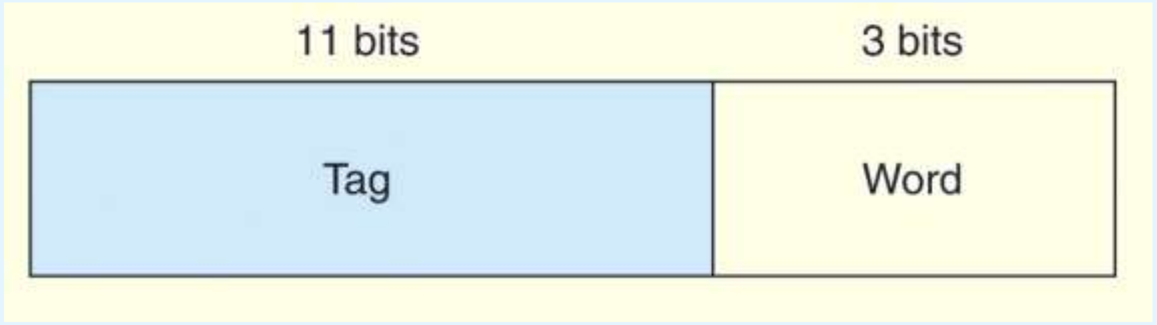
- Instead of placing memory blocks in specific cache locations based on memory address, we could allow a block to go **anywhere** in cache.
- In this way, cache would have to fill up before any blocks are replaced.
- This is how ***fully associative* cache** works.
- A memory address is partitioned into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

Fully Associative Cache



Fully Associative Cache

- If 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

- No such mapping,
- Each memory block can be placed at any cache block
- The block that is replaced is the ***victim block***.
- Which block you will replace?
- There are a number of ways to pick a victim,

Replacement Policy

- 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.
- It is **impossible** to implement an optimal replacement algorithm,
- It is instructive to use it as a **benchmark** for assessing the efficiency of other schemes.

Replacement Policy

- 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.

Replacement Policy

- *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.

Replacement Policy

- *A r**andom** 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.*

Cache performance

- 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:

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

H is the cache hit rate

Access$_{\mathbf{C}}$ and **Access**$_{\mathbf{MM}}$ are the access times for cache and main memory, respectively.

Cache performance

- 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(10ns) + 0.01(200ns) = 9.9ns + 2ns = 11ns.

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

Hits vs. Misses

- Read hits
  - This is what we want!
- Read misses
  - Stall the CPU,
  - Fetch block from memory
  - Deliver to cache
  - Restart
- Write hits:
  - Can replace data in cache and memory (write-through)
  - Write data only into the cache (write-back the cache later)
- Write misses:
  - Read the entire block into the cache, then write the word

Handling Cache Misses

- The action taken for cache miss depends on whether the action is to access instruction or data
- Data Access
  - Stall the processor until the memory responds with the data
- Instruction access:
  - The contents of the instruction register are invalid
    - Get the instruction from the lower-level memory into the cache

Write-Through Approach

- Idea:
  - Always write data into both memory & cache
- Steps:
  1. Index the cache (Bits 15-2 of address)
  2. Write the tag, data, & valid bit into cache
  3. Write the word to main memory using the entire address
- Advantages
  - Simple algorithm
- Disadvantages
  - Poor performance
    - Writing into main memory slows down the machine

Write-Back Approach

- Handles writes by updating values only to the block in the cache, then writing the modified block to the lower level of the hierarchy when the block is replaced
- Steps:
  1. Index the cache
  2. Write the tag, data, & valid bit into cache
  3. The modified block is written to the memory only when it is replaced
- Advantages:
  – Improves performance
- Disadvantages:
  – Complex algorithm

Cache Write Policy

- Cache replacement policies must also 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

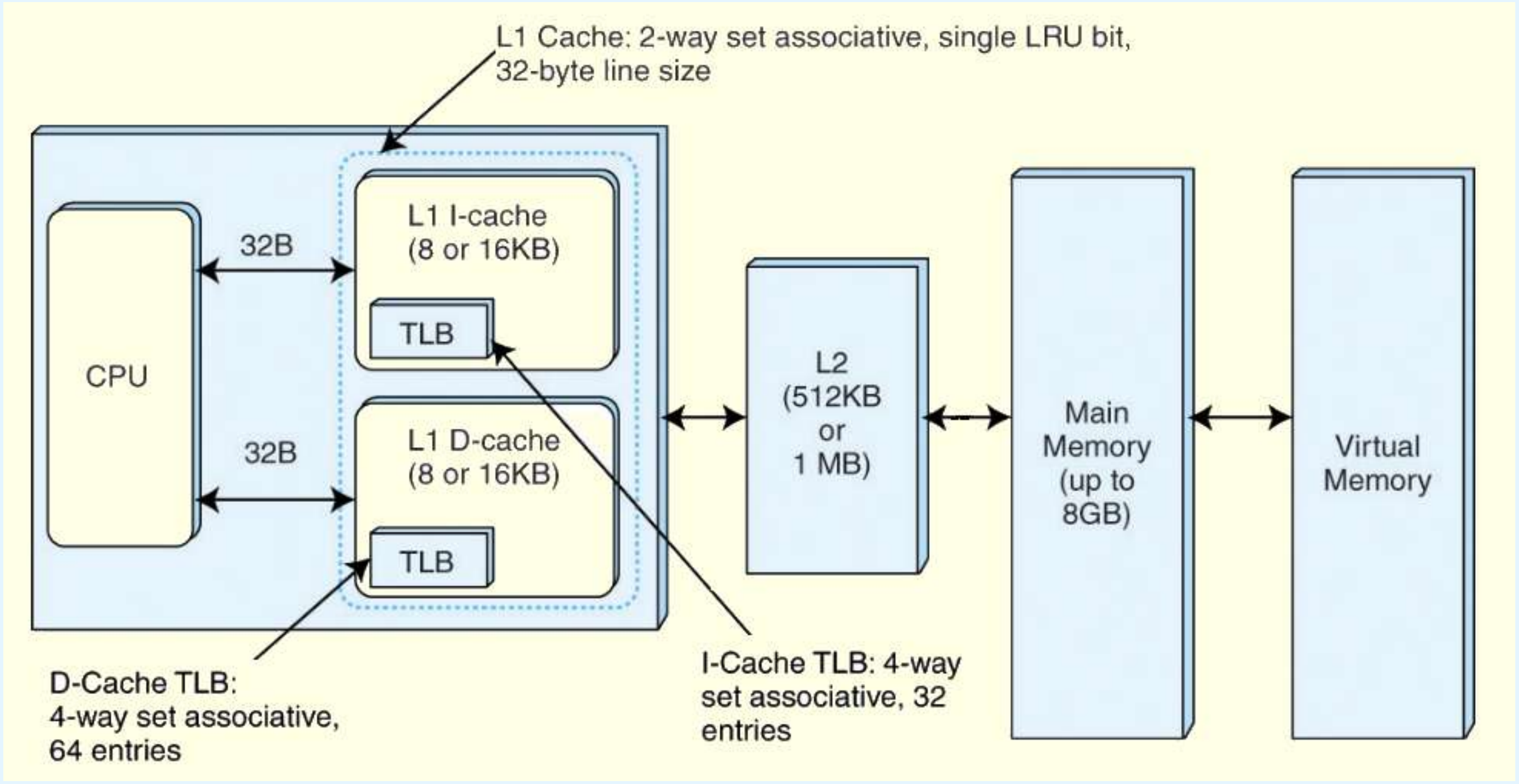
- Write back (also called *copyback*) updates memory only when the block is selected for replacement.
- The disadvantage of write through is that memory must be updated with each cache write, which slows down the access time on updates. This slowdown is usually negligible, because the majority of accesses 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.

6.6 A Real-World Example

- The Pentium architecture supports both paging and segmentation, and they can be used in various combinations including unpaged unsegmented, segmented unpaged, and unsegmented paged.
- The processor supports two levels of cache (L1 and L2), both having a block size of 32 bytes.
- The L1 cache is next to the processor, and the L2 cache sits between the processor and memory.
- The L1 cache is in two parts: and instruction cache (I-cache) and a data cache (D-cache).

The next slide shows this organization schematically.

6.6 A Real-World Example


L1 Cache: 2-way set associative, single LRU bit, 32-byte line size
CPU
32B
32B
L1 I-cache (8 or 16KB)
TLB
L1 D-cache (8 or 16KB)
TLB
L2 (512KB or 1 MB)
Main Memory (up to 8GB)
Virtual Memory
D-Cache TLB: 4-way set associative, 64 entries
I-Cache TLB: 4-way set associative, 32 entries

Chapter 6 Conclusion

- Computer memory is organized in a hierarchy, with the smallest, fastest memory at the top and the largest, slowest memory at the bottom.
- Cache memory gives faster access to main memory, while virtual memory uses disk storage to give the illusion of having a large main memory.
- Cache maps blocks of main memory to blocks of cache memory. Virtual memory maps page frames to virtual pages.
- There are three general types of cache: Direct mapped, fully associative and set associative.

Chapter 6 Conclusion

- With fully associative and set associative cache, as well as with virtual memory, replacement policies must be established.
- Replacement policies include LRU, FIFO, or LFU. These policies must also take into account what to do with dirty blocks.