



## Chapter 3

# COMPUTER MEMORY

## Part 1

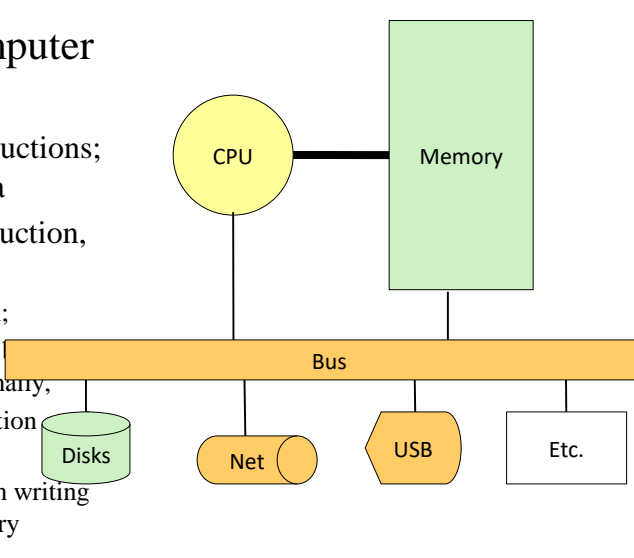


## Contents

- 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.

## Hardware review

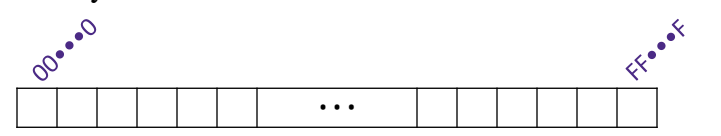
- Overview computer systems
- CPU executes instructions; memory stores data
- To execute an instruction, the CPU must:
  - fetch an instruction;
  - fetch the data used in instruction; and, finally,
  - execute the instruction on data...
  - which may result in writing data back to memory



The diagram illustrates the hardware components of a computer system. A central horizontal orange bar represents the **Bus**. Above the bus, a yellow circle labeled **CPU** and a green rectangle labeled **Memory** are connected to the bus by vertical lines. Below the bus, four components are connected: a green cylinder labeled **Disks**, a yellow circle labeled **Net**, a yellow hexagon labeled **USB**, and a white rectangle labeled **Etc.**. A thick black line connects the CPU directly to the Memory.

## Byte-Oriented Memory Organization

- Conceptually, memory is a single, large array of bytes, each with a unique *address* (index)
  - The value of each byte in memory can be read and written
- Programs refer to bytes in memory by their *addresses*
  - Domain of possible addresses = *address space*
- But not all values fit in a single byte... (e.g. 410)
  - Many operations actually use multi-byte values
- We can store addresses as data to “remember” where other data is in memory



The diagram shows a horizontal sequence of 12 small squares representing bytes in memory. Above the first square is the address `00...0` and above the last square is the address `FF...F`. An ellipsis `...` is placed between the sixth and seventh squares, indicating a continuation of the memory sequence.




## 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.




- 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**).
- Dynamic RAM 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.



## Characteristics of memory systems

- The memory system can be characterised with their Location, Capacity, Unit of transfer, Access method, Performance, Physical type, Physical characteristics, Organisation
- **Location**
  - Processor memory: The memory like registers is included within the processor and termed as processor memory.
  - Internal memory: It is often termed as main memory and resides within the CPU
  - External memory: It consists of peripheral storage devices such as disk and magnetic tape that are accessible to processor via i/o controllers



- **Capacity**
  - Word size: Capacity is expressed in terms of words or bytes. The natural unit of organisation
  - Number of words: Common word lengths are 8, 16, 32 bits etc. or Bytes
- **Unit of Transfer**
  - Internal: For internal memory, the unit of transfer is equal to the number of data lines into and out of the memory module
  - External: For external memory, they are transferred in block which is larger than a word.
  - Addressable unit : Smallest location which can be uniquely addressed— Word internally— Cluster on Magnetic disks



## ■ Access Method

- Sequential acces. Examples tape
- Direct Access: Individual blocks of records have unique address based on location. Access is accomplished by jumping (direct access) to general vicinity plus a sequential search to reach the final location. Example disk
- Random access: example RAM
- Associative access: example cache



## ■ Performance

- Access time
- Memory Cycle time
- Transfer Rate:

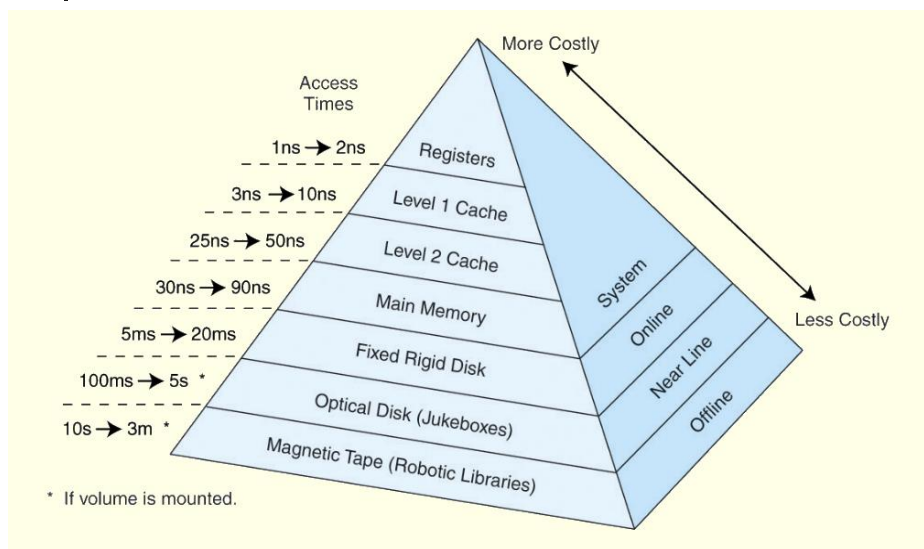
## ■ Physical Types

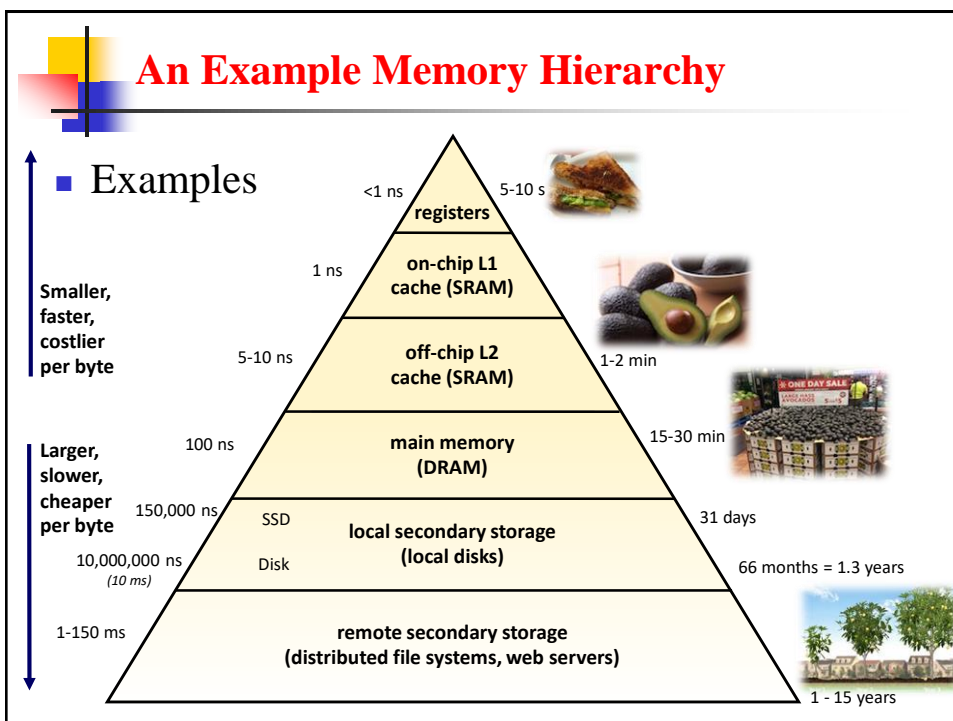
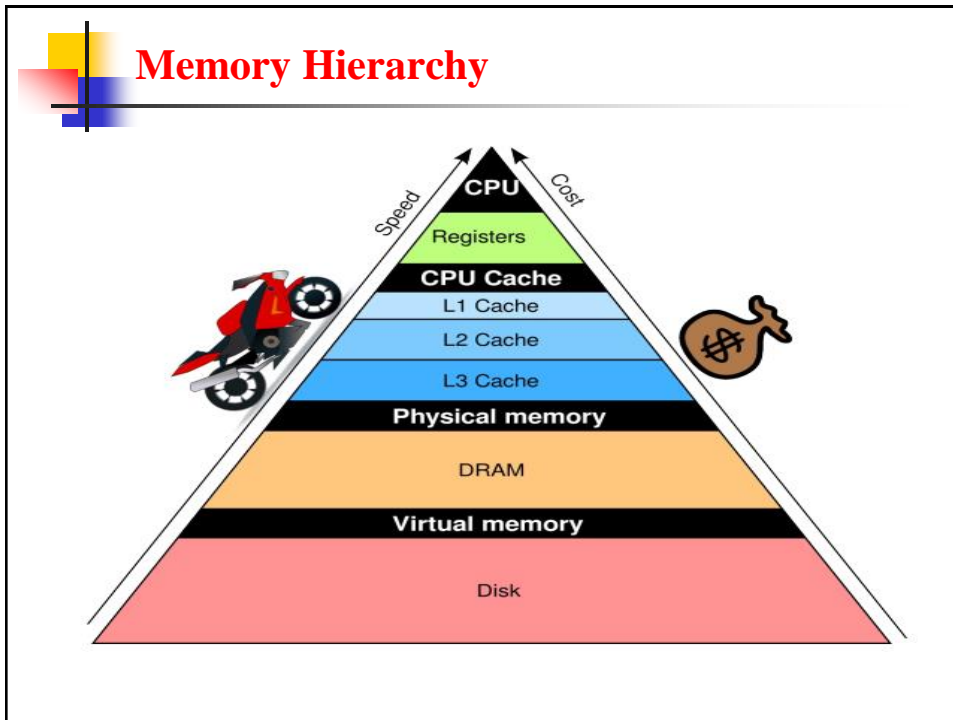
- Semiconductor : RAM
- Magnetic : Disk & Tape
- Optical : CD & DVD
- Others

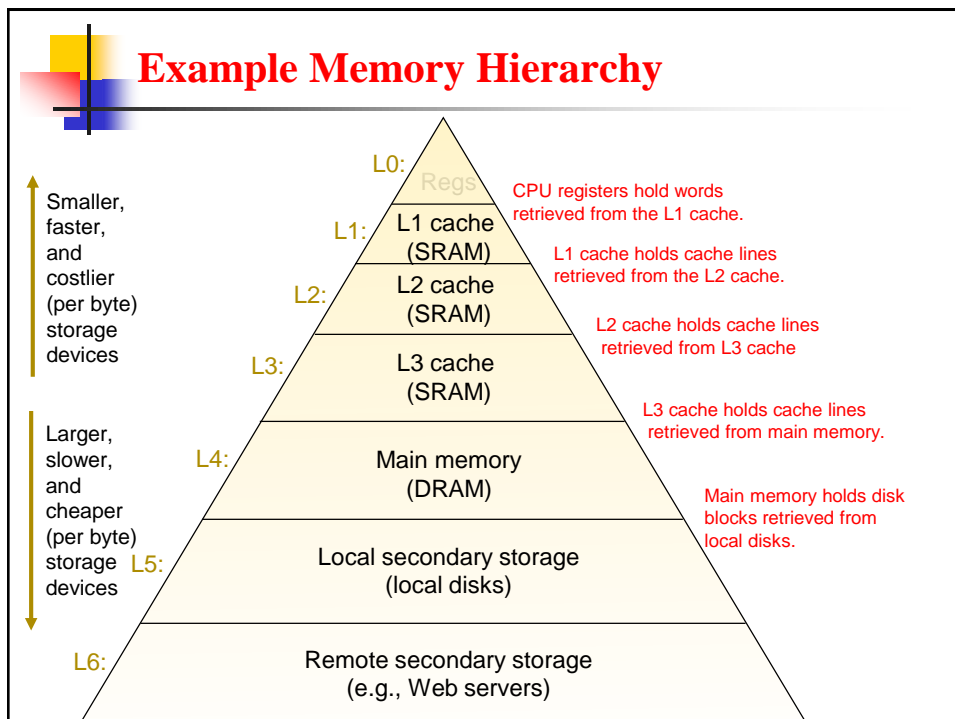
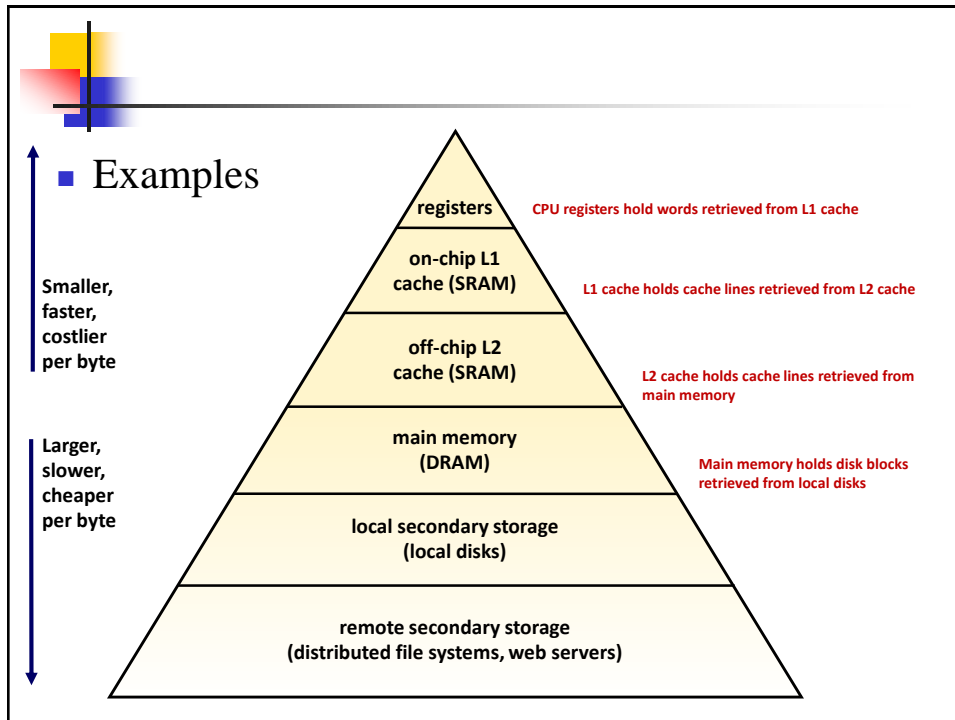


## The Memory Hierarchy

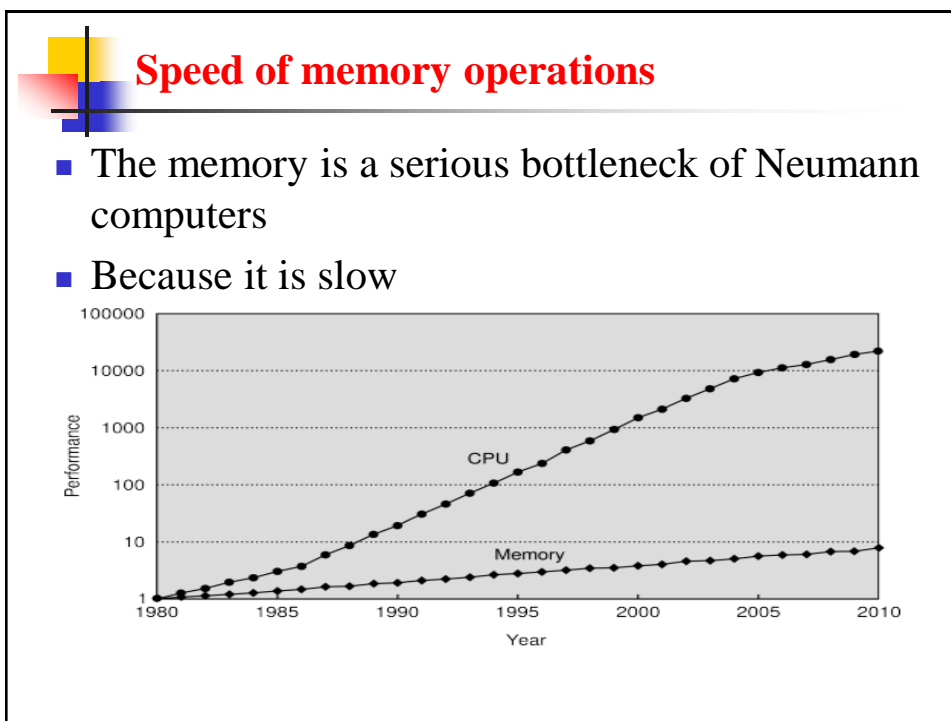
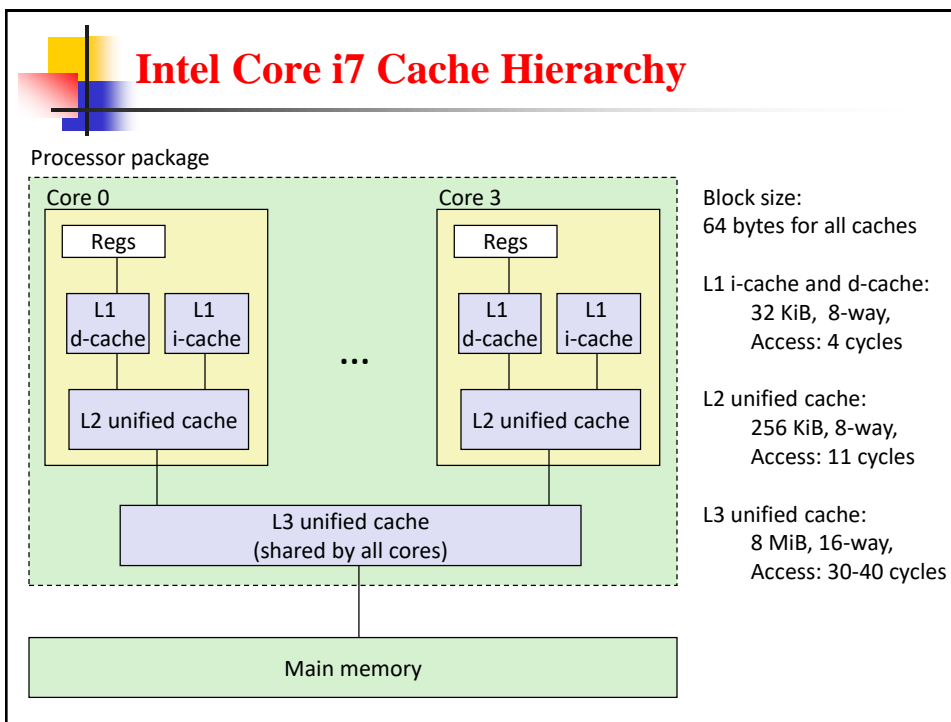
- Generally speaking, **faster memory is more expensive** than slower memory.
- To provide the best performance at the lowest cost, memory is organized in a hierarchical fashion.
- **Small, fast storage elements** are kept in the CPU, **larger, slower main memory** is accessed through the data bus.
-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



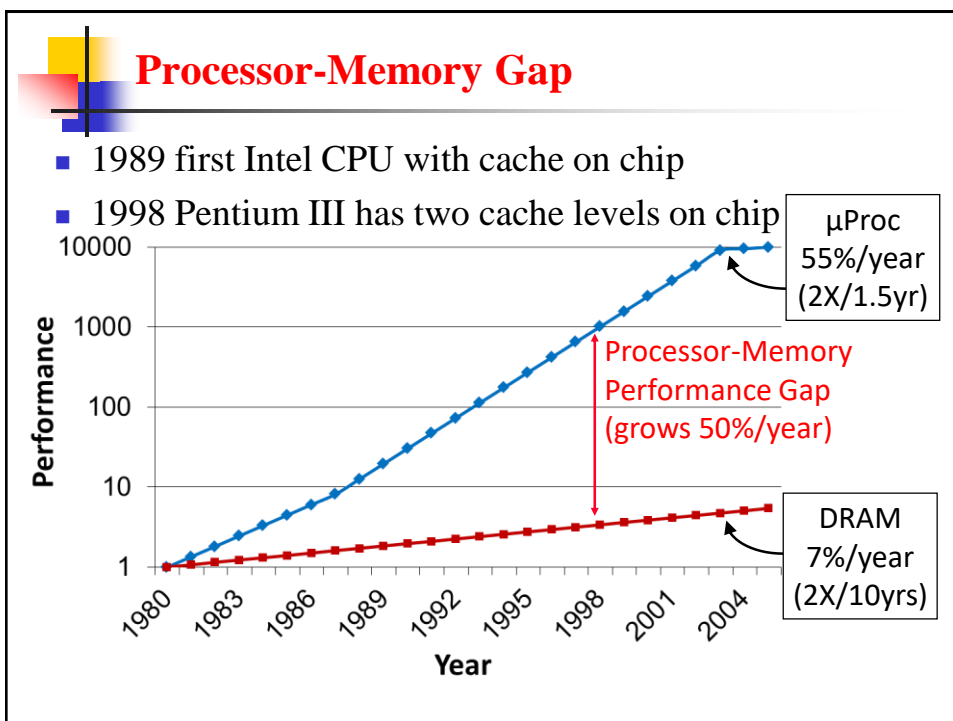


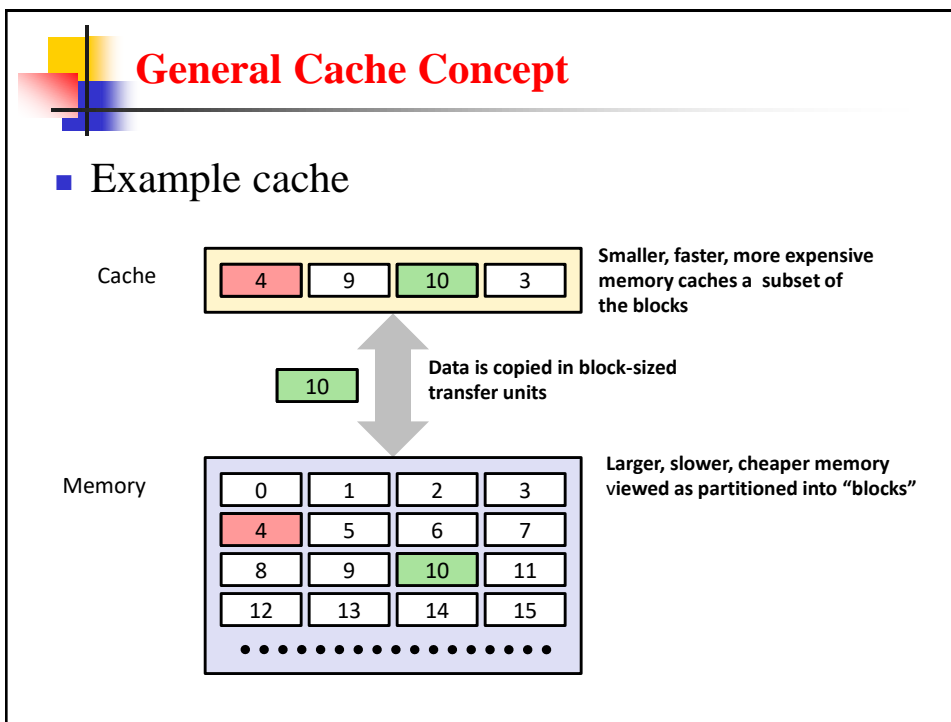






- Programs do not refer to memory addresses randomly
- Memory addresses referenced by the programs show a special pattern → we can utilize it!
- Locality of reference:
  - Temporal: a memory content referenced will be referenced again in the near future
  - Spatial: if a memory content has been referenced, its neighborhood will be referenced as well in the near future
  - Algorithmic: some algorithms (like traversing linked lists) refer to the memory in a systematic way
- Examples:
  - Media players:
  - Spatial locality: yes, temporal locality: no
  - A "for" loop in the C language:
  - Both temporal and spatial locality hold





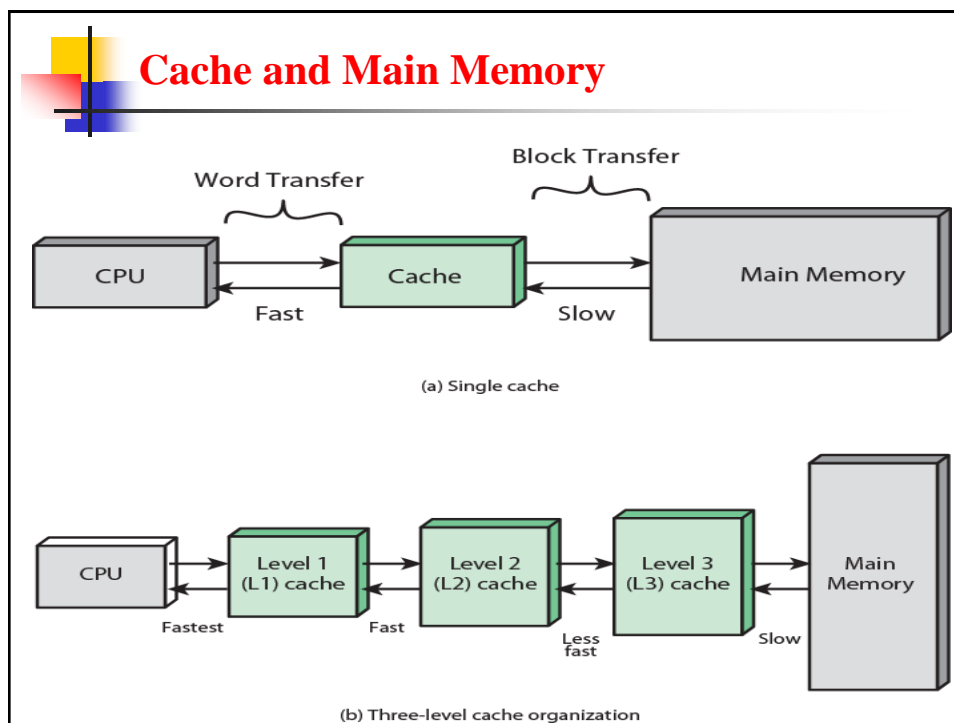
## Cache memory

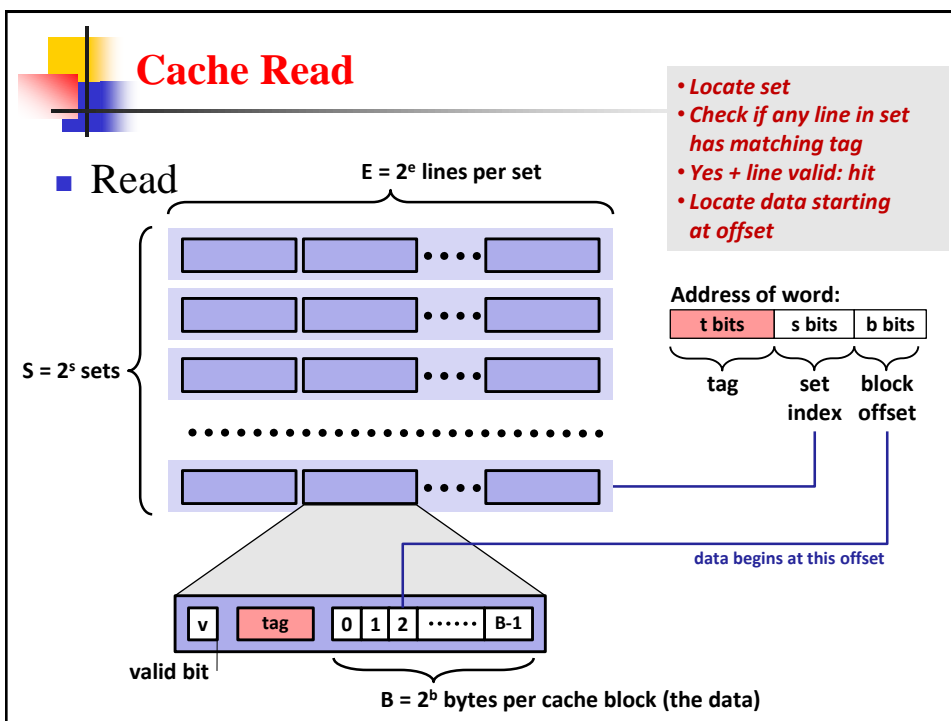
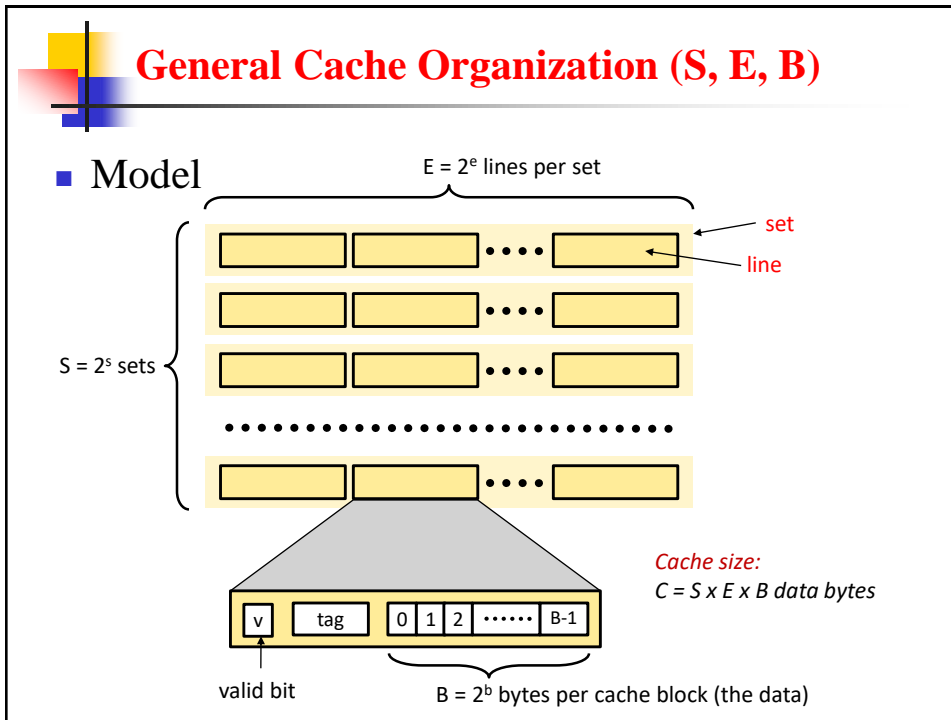
- Small amount of fast memory
- Sits between normal main memory and CPU
- May be located on CPU chip or module
- What we are going to cover are:
- Cache organization:
  - How to store data in the cache in an efficient way
- Cache content management:
  - When to put a data to the cache and when not
  - What shall we throw out from the cache if we want to put new data there

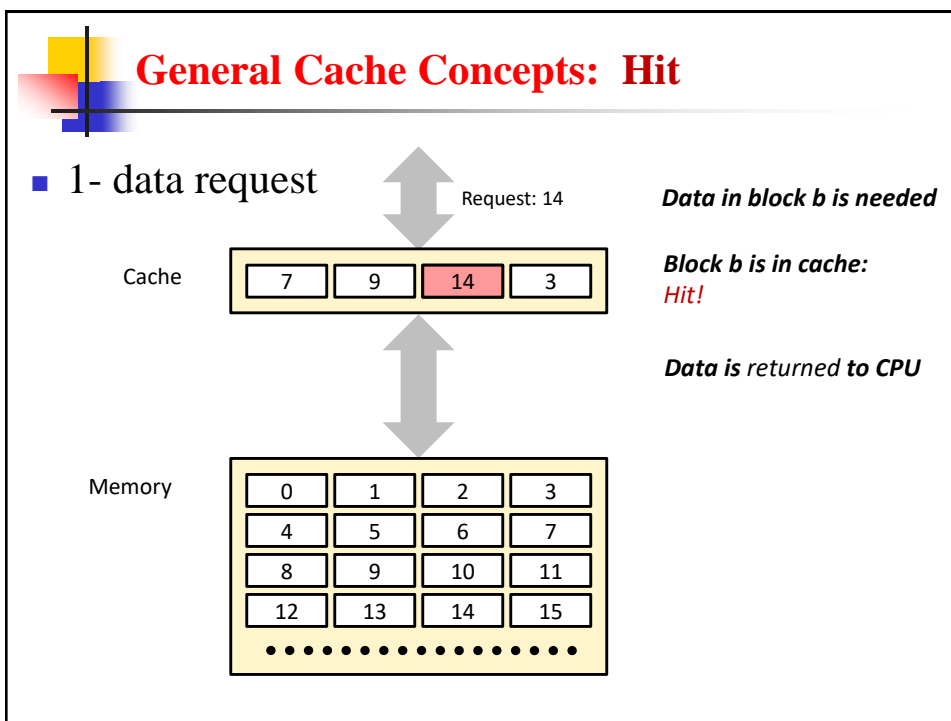
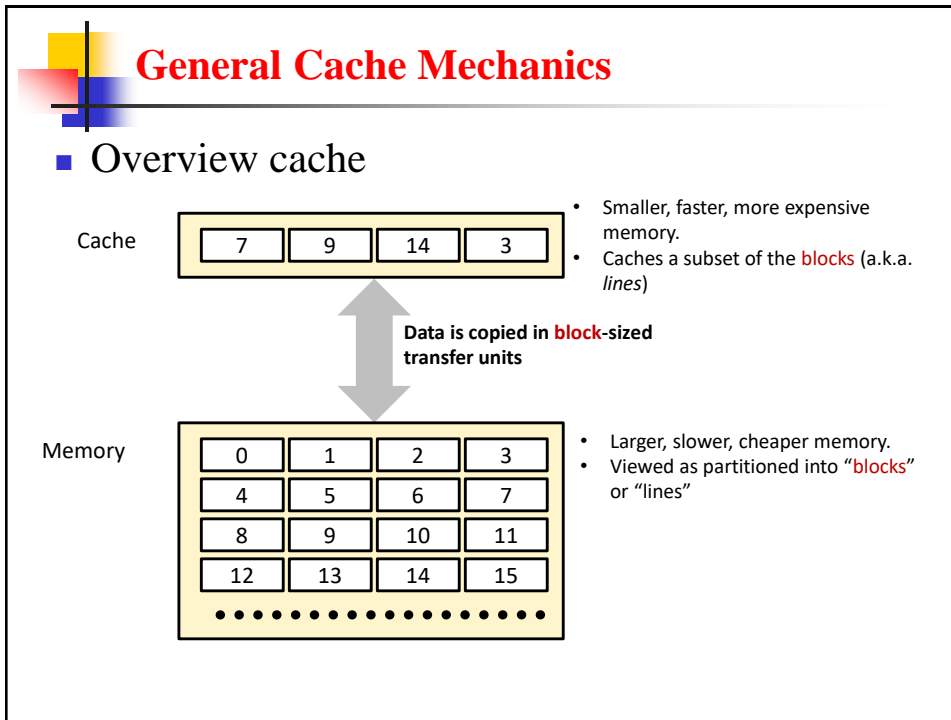
## Cache Memories

- **Cache memories** are small, fast SRAM-based memories managed automatically in hardware
  - Hold frequently accessed blocks of main memory
- CPU looks first for data in cache
- Typical system structure:

The diagram illustrates the typical system structure. A CPU chip is shown with internal components: Cache memory, Register file, and ALU. The Cache memory is connected to the Register file, which is connected to the ALU. The Cache memory is also connected to a Bus interface. The Bus interface is connected to a System bus. The System bus is connected to an I/O bridge. The I/O bridge is connected to a Memory bus. The Memory bus is connected to Main memory.

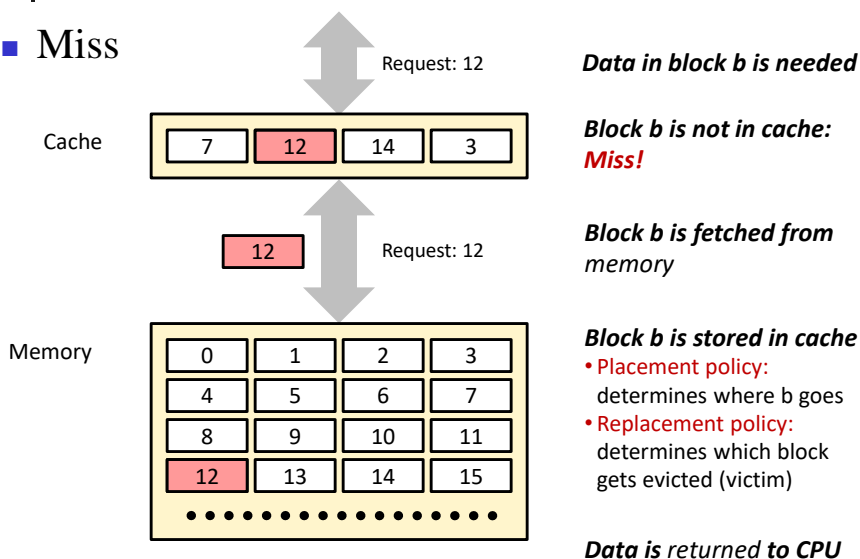






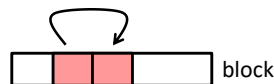
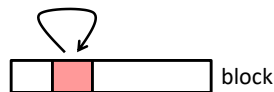
## General Cache Concepts: Miss

### ■ Miss



## Why Caches Work

- **Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently
- **Temporal locality:**
  - Recently referenced items are *likely* to be referenced again in the near future
- **Spatial locality:**
  - Items with nearby addresses *tend* to be referenced close together in time
- How do caches take advantage of this?





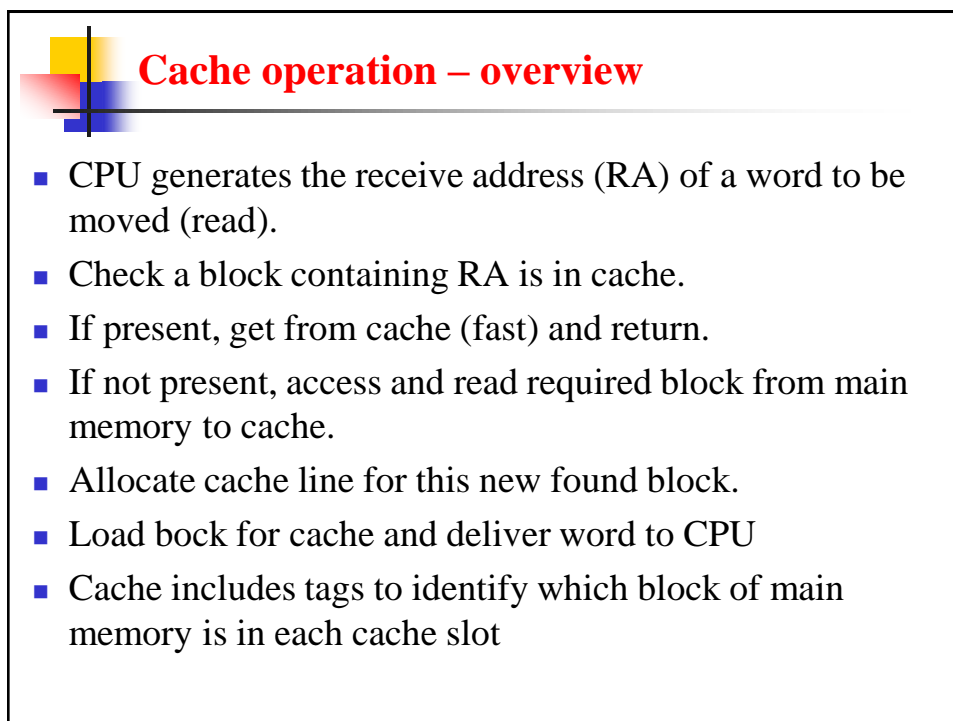
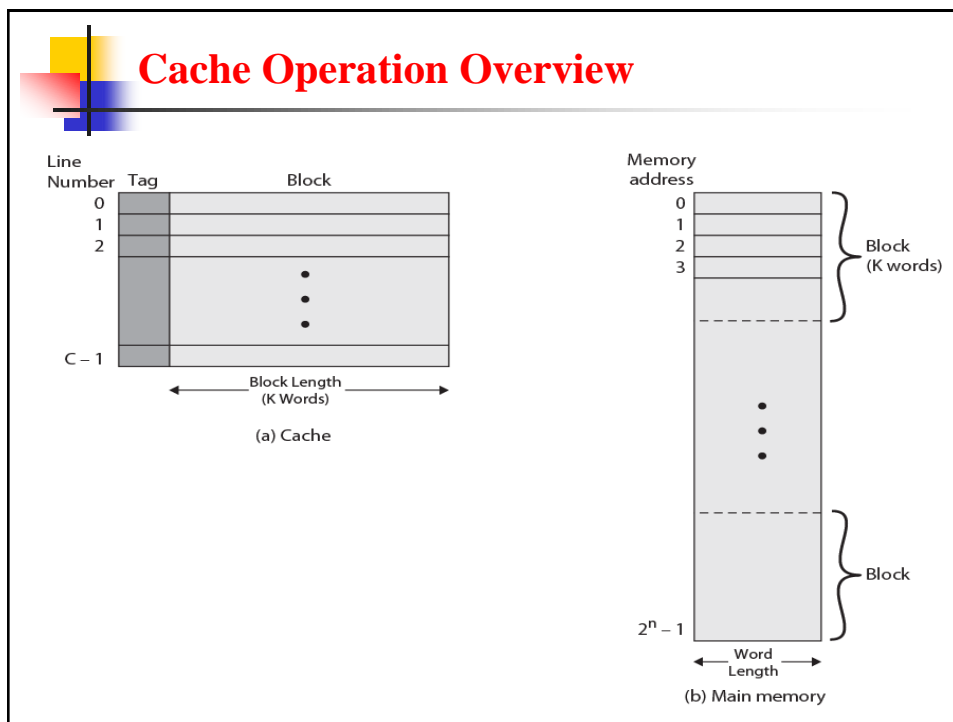
## Operations of cache


- When the CPU needs to access memory, cache is examined. If the word is found in cache, it is read from the cache and if the word is not found in cache, main memory is accessed to read word. A block of word containing the one just accessed is then transferred from main memory to cache memory.
- Cache connects to the processor via data control and address line. The data and address lines also attached to data and address buffer which attached to a system bus from which main memory is reached.



- When a cache hit occurs, the data and address buffers are disabled and the communication is only between processor and cache with no system bus traffic. When a cache miss occurs, the desired word is first read into the cache and then transferred from cache to processor. For later case, the cache is physically interposed between the processor and main memory for all data, address and control lines








## Elements of Cache Design

- Addressing
- Size
- Mapping Function
- Replacement Algorithm
- Write Policy
- Block Size
- Number of Caches



## Cache Addressing

- Where does cache sit?
  - Between processor and virtual memory management unit
  - Between MMU and main memory
- Logical cache (virtual cache) stores data using virtual addresses
  - Processor accesses cache directly, not thorough physical cache
  - Cache access faster, before MMU address translation
  - Virtual addresses use same address space for different applications
    - Must flush cache on each context switch
- Physical cache stores data using main memory physical addresses




## Cache size

- Size of the cache to be small enough so that the overall average cost per bit is close to that of main memory alone and large enough so that the overall average access time is close to that of the cache alone
- The larger the cache, the larger the number of gates involved in addressing the cache
- Large caches tend to be slightly slower than small ones – even when built with the same integrated circuit technology and put in the same place on chip and circuit board.
- The available chip and board also limits cache size.




## Mapping function

- The transformation of data from main memory to cache memory is referred to as memory mapping process
- Because there are fewer cache lines than main memory blocks, an algorithm is needed for mapping main memory blocks into cache lines.
- There are three different types of mapping functions in common use and are direct, associative and set associative. All the three include following elements in each example.



## Information about examples

- Cache of 64kByte
- Cache block of 4 bytes → cache is 16k ( $2^{14}$ ) lines of 4 bytes
- 16MBytes main memory
- 24 bit address ( $2^{24}=16\text{M}$ )
- Thus, for mapping purposes, we can consider main memory to consist of 4Mbytes blocks of 4 bytes each.



- **Block Size ( $K$ ):** unit of transfer between *cache* and Mem
  - Given in bytes and always a power of 2 (*e.g.* 64 B)
  - Blocks consist of adjacent bytes (differ in address by 1)
    - Spatial locality!
- **Offset field**
  - Low-order  $\log_2(K) = k$  bits of address tell you which byte within a block
    - (address) mod  $2^n = n$  lowest bits of address
  - (address) modulo (# of bytes in a block)

	$m - k$ bits	$k$ bits
<b><math>m</math>-bit address:</b>	Block Number	Block Offset
(refers to byte in memory)		



- **Cache Size ( $C$ ):** amount of *data* the *cache* can store
  - Cache can only hold so much data (subset of next level)
  - Given in bytes ( $C$ ) or number of blocks ( $C/K$ )
  - Example:  $C = 32 \text{ KiB} = 512 \text{ blocks}$  if using 64-B blocks
- Where should data go in the cache?
  - We need a mapping from memory addresses to specific locations in the cache to make checking the cache for an address **fast**
- What is a data structure that provides fast lookup?
  - Hash table!



## Cache organization

- Data units in the cache are called: **block**
  - Size of blocks =  $2^L$
  - Lower  $L$  bit of memory addresses: position inside a block
  - Upper bits: number (ID) of the block
- Additional information to be stored in the cache with each block:
  - Cache **tag** (which block of the system memory is stored here)
  - **Valid** bit: if =1, this cache block stores valid data
  - **Dirty** bit: if =1, this cache block has been modified since in the cache
- The principal questions of cache organization are:
  - How to store blocks in the cache
  - To enable finding a block fast
  - To make it simple and relatively cheap



## Checking for a Requested Address

- CPU sends address request for chunk of data
  - Address and requested data are not the same thing!
    - Analogy: your friend  $\neq$  his or her phone number
- TIO address breakdown:


*m*-bit address:

Tag ( <i>t</i> )	Index ( <i>s</i> )	Offset ( <i>k</i> )
------------------	--------------------	---------------------

└──────────┘

Block Number

- **Index** field tells you where to look in cache
- **Tag** field lets you check that data is the block you want
- **Offset** field selects specified start byte within block
- **Note:** *t* and *s* sizes will change based on hash function



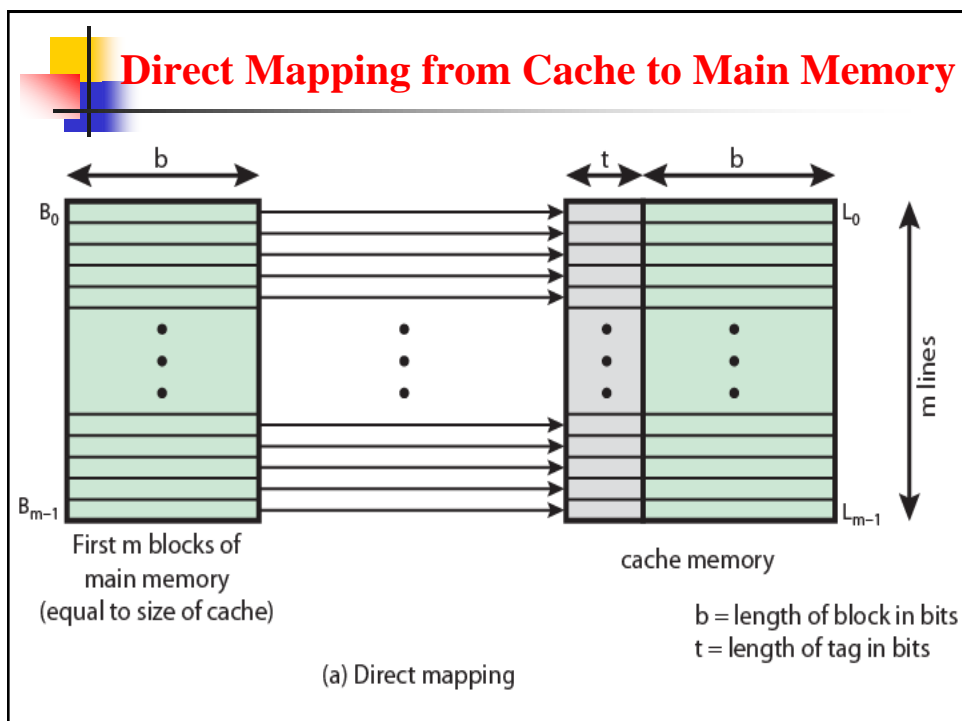
## Direct Mapping


- Each block of main memory maps to only one cache line
  - i.e. if a block is in cache, it must be in one specific place
- Address is in two parts
- Least Significant *w* bits identify unique word
- Most Significant *s* bits specify one memory block
- The MSBs are split into a cache line field *r* and a tag of *s-r* (most significant)

## Direct Mapping Address Structure

Tag s-r	Line or Slot r	Word w
8	14	2

- 24 bit address
- 2 bit word identifier (4 byte block)
- 22 bit block identifier
  - 8 bit tag (=22-14)
  - 14 bit slot or line
- No two blocks in the same line have the same Tag field
- Check contents of cache by finding line and checking Tag

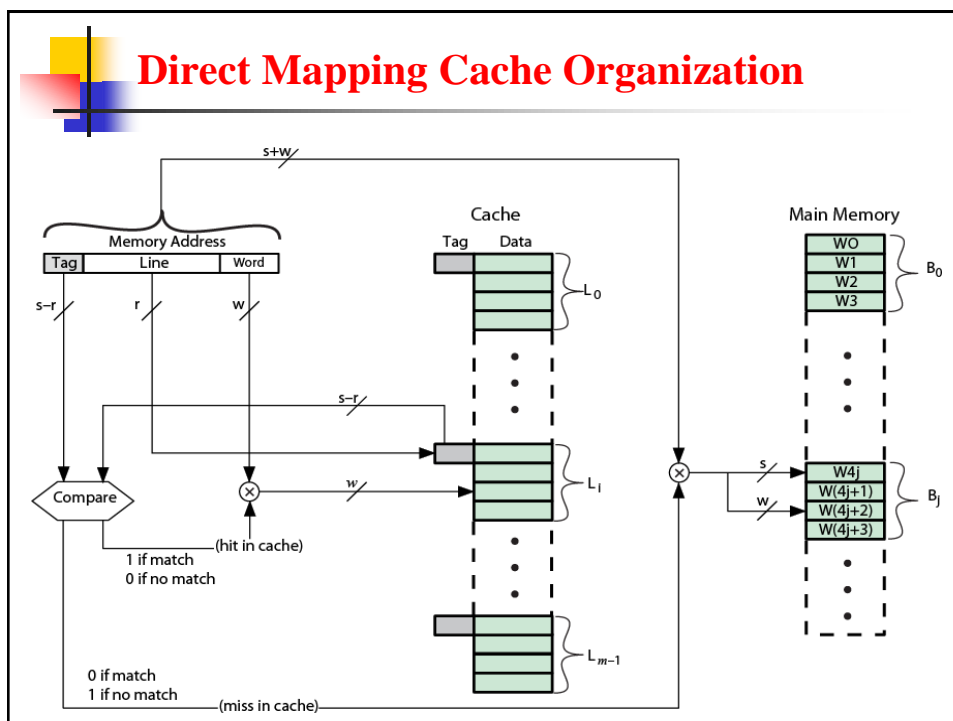




## Direct Mapping Cache Line Table

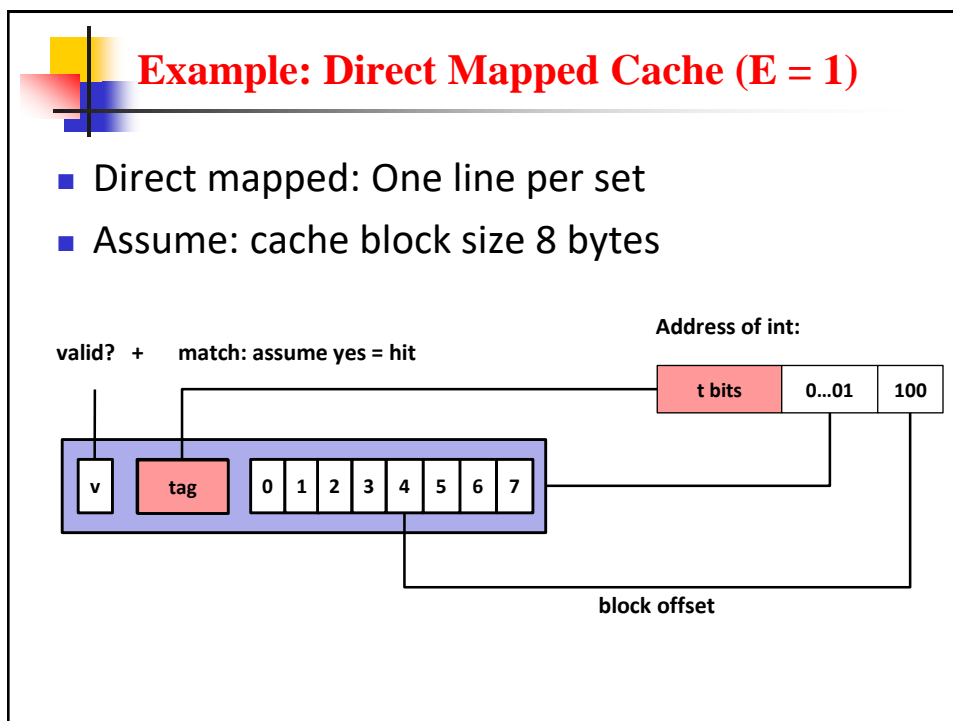
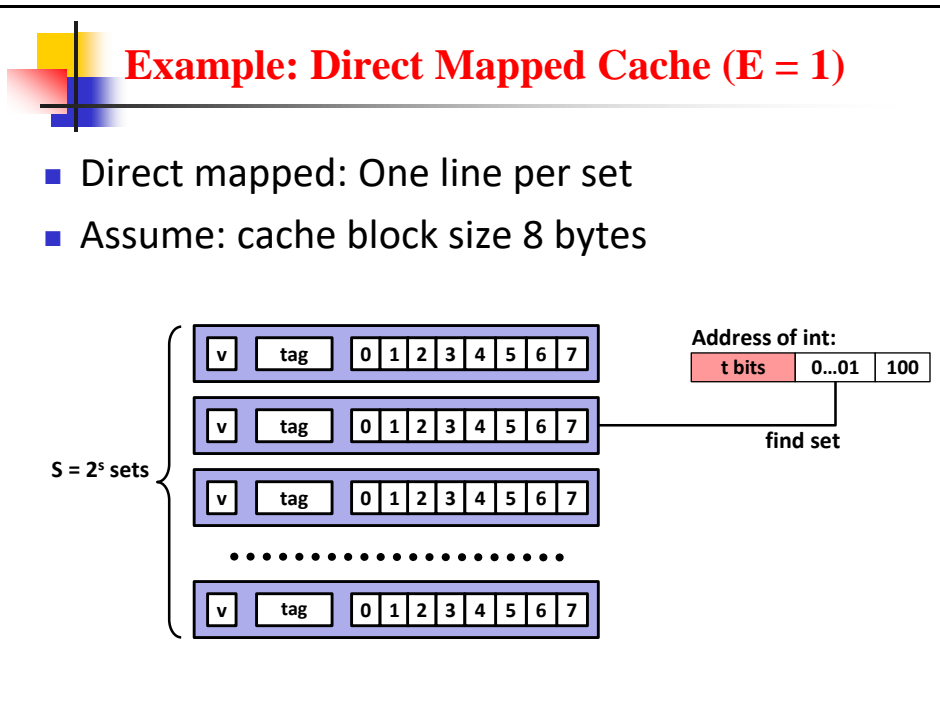
- Table mapping main memory block and cache line

Cache line	Main Memory blocks held
0	$0, m, 2m, 3m \dots 2s-m$
1	$1, m+1, 2m+1 \dots 2s-m+1$
...	
$m-1$	$m-1, 2m-1, 3m-1 \dots 2s-1$





- Address length =  $(s + w)$  bits
- Number of addressable units =  $2^{s+w}$  words or bytes
- Block size = line size =  $2^w$  words or bytes
- Number of blocks in main memory =  $2^{s+w}/2^w = 2^s$
- Number of lines in cache =  $m = 2^r$
- Size of tag =  $(s - r)$  bits
- Simple
- Inexpensive
- Fixed location for given block
  - If a program accesses 2 blocks that map to the same line repeatedly, cache misses are very high



### Example: Direct Mapped Cache (E = 1)

- Direct mapped: One line per set
- Assume: cache block size 8 bytes

valid? + match: assume yes = hit

Address of int: t bits 0...01 100

block offset

int (4 Bytes) is here

- If tag doesn't match: old line is evicted and replaced

### Direct-Mapped Cache Simulation

- Example of direct mapped

t=1 s=2 b=1

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

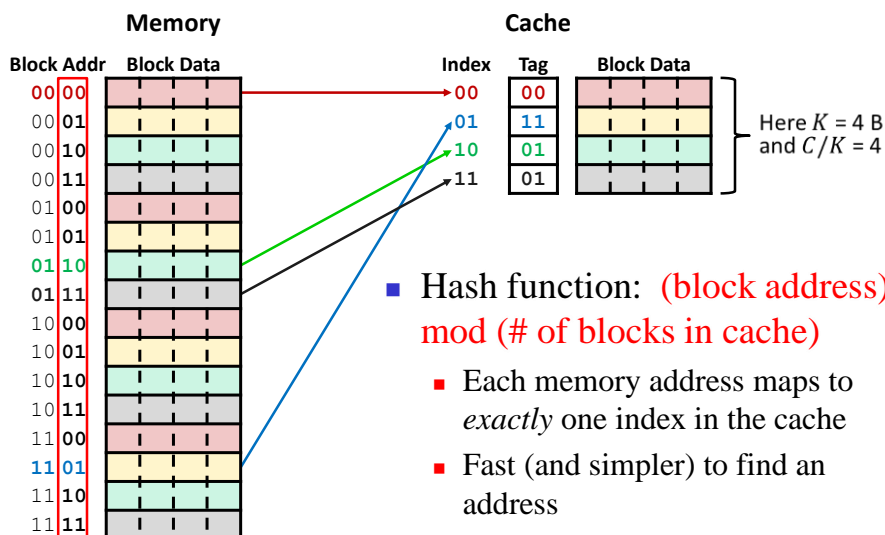
M=16 bytes (4-bit addresses), B=2 bytes/block, S=4 sets, E=1 Blocks/set

Address trace (reads, one byte per read):

0	[0000] <sub>2</sub> ,	miss
1	[0001] <sub>2</sub> ,	hit
7	[0111] <sub>2</sub> ,	miss
8	[1000] <sub>2</sub> ,	miss
0	[0000] <sub>2</sub>	miss

	v	Tag	Block
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

## Direct-Mapped Cache-example

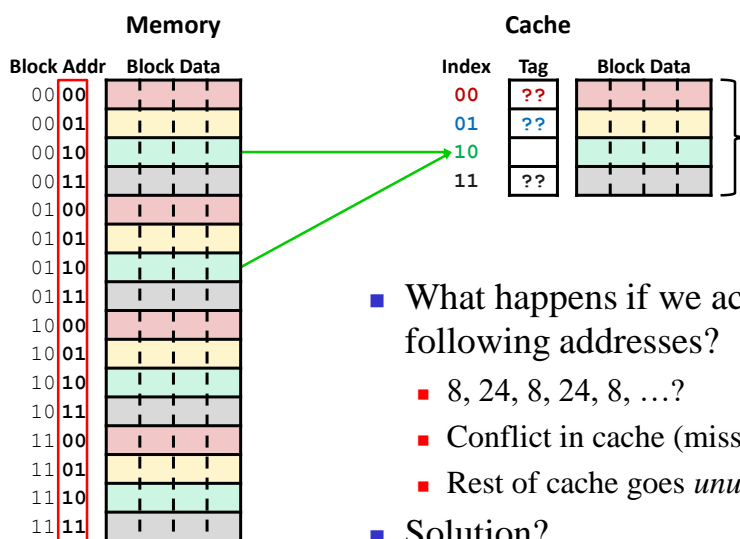


## Direct-Mapped Cache Problem

■ 8 = 00 10 00

■ 24 = 01 10 00

(t) (s) (k)





### Note that

- *All locations in a single block of memory have the same higher order bits (call them the block number), so the lower order bits can be used to find a particular word in the block.*
- *Within those higher-order bits, their lower-order bits obey the modulo mapping given above (assuming that the number of cache lines is a power of 2), so they can be used to get the cache line for that block*
- *The remaining bits of the block number become a tag, stored with each cache line, and used to distinguish one block from another that could fit into that same cache line*



### Associated Mapping

- It overcomes the disadvantage of direct mapping by permitting each main memory block to be loaded into any line of cache.
- Cache control logic interprets a memory address simply as a tag and a word field
- Tag uniquely identifies block of memory
- Cache control logic must simultaneously examine every line's tag for a match which requires fully associative memory
- very complex circuitry, complexity increases exponentially with size
- Cache searching gets expensive



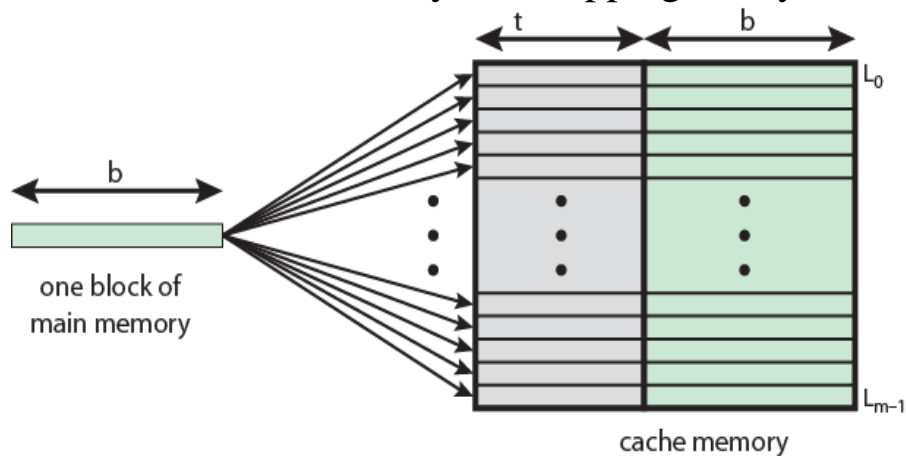
## Associative Mapping

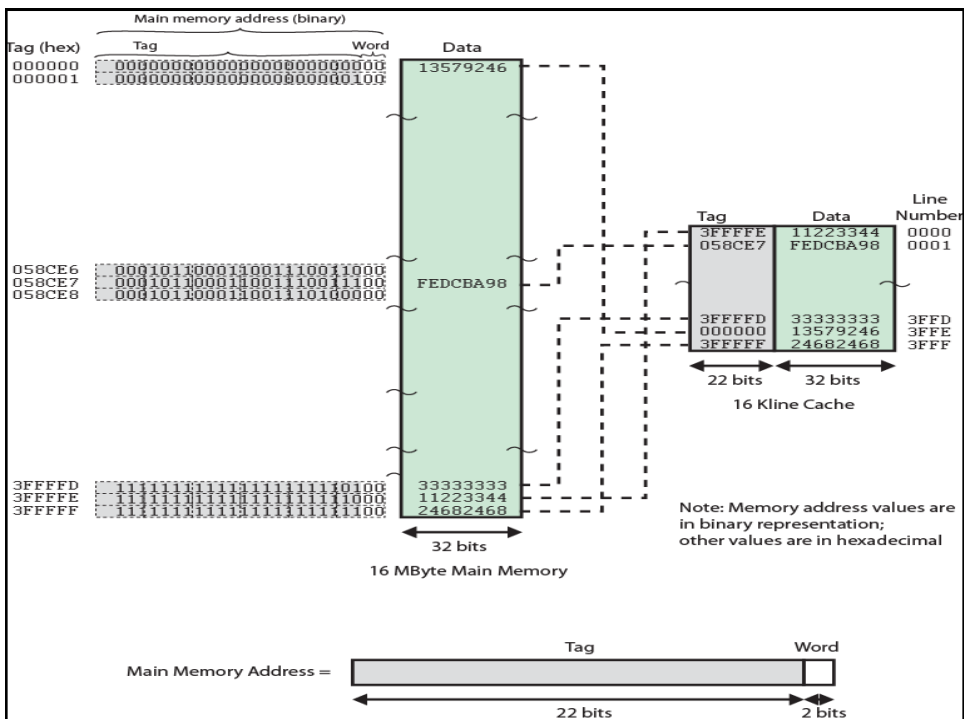
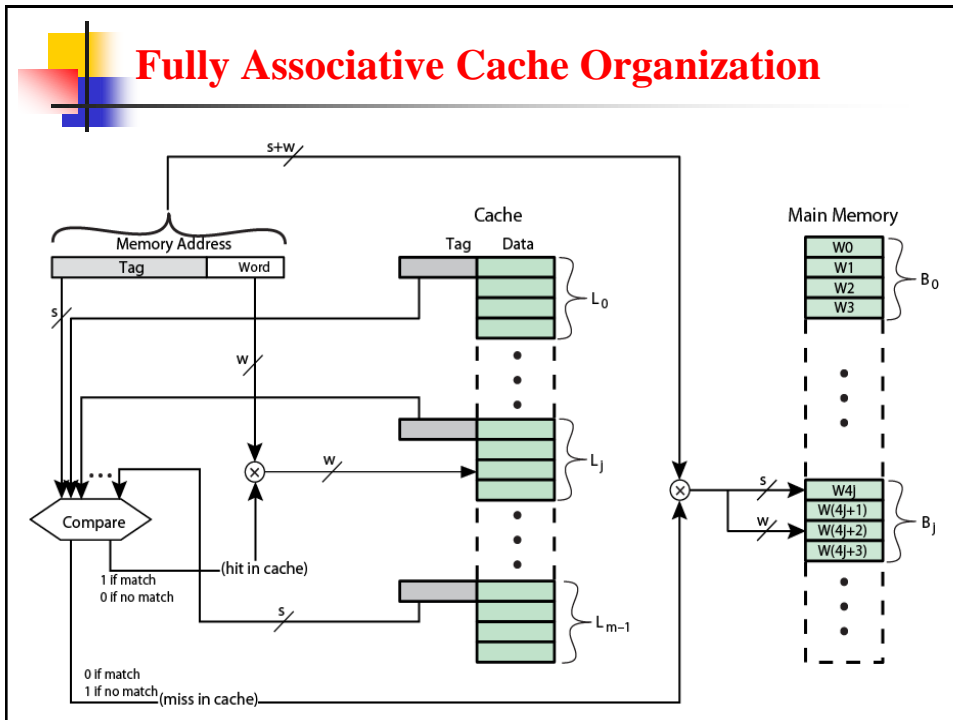
- A main memory block can load into any line of cache
- Memory address is interpreted as tag and word
- Tag uniquely identifies block of memory
- Every line's tag is examined for a match
- Cache searching gets expensive




## Associative Mapping from Cache to Main Memory

- Block of main memory can mapping to any line








## Associative Mapping Address Structure

Tag 22 bit	Word 2 bit
------------	---------------

- 22 bit tag stored with each 32 bit block of data
- Compare tag field with tag entry in cache to check for hit
- Least significant 2 bits of address identify which 16 bit word is required from 32 bit data block
- e.g.
 

■ Address	Tag	Data	Cache line
■ FFFFC	FFFFC	24682468	3FFF



## Associative Mapping Summary

- Address length =  $(s + w)$  bits
- Number of addressable units =  $2^{s+w}$  words or bytes
- Block size = line size =  $2^w$  words or bytes
- Number of blocks in main memory =  $2^{s+w}/2^w = 2^s$
- Number of lines in cache = undetermined
- Size of tag =  $s$  bits





## Set Associated Mapping

- It is a compromise between direct and associative mappings that exhibits the strength and reduces the disadvantages
- Cache is divided into  $v$  sets, each of which has  $k$  lines;  
number of cache lines =  $vk$   
 $M = v \times k$   
 $I = j \text{ modulo } v$   
Where,  $i$  = cache set number;  $j$  = main memory block number;  $m$  = number of lines in the cache
- So a given block will map directly to a particular set, but can occupy any line in that set (associative mapping is used within the set)



- Cache control logic interprets a memory address simply as three fields tag, set and word. The  $d$  set bits specify one of  $v = 2^d$  sets. Thus  $s$  bits of tag and set fields specify one of the  $2^s$  block of main memory.
- The most common set associative mapping is 2 lines per set, and is called two-way set associative. It significantly improves hit ratio over direct mapping, and the associative hardware is not too expensive.



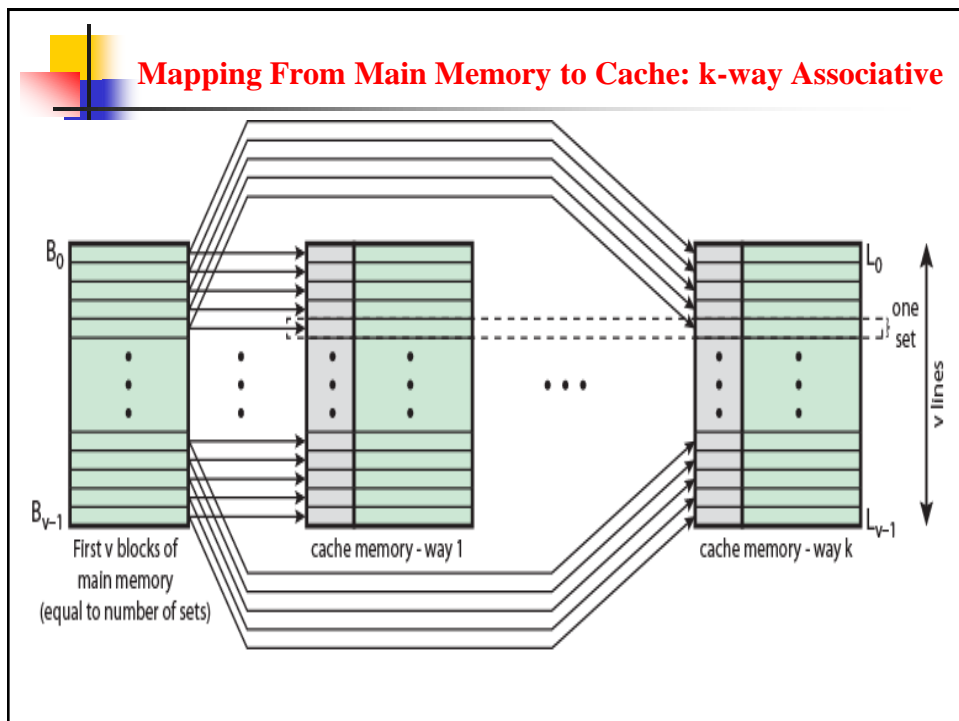
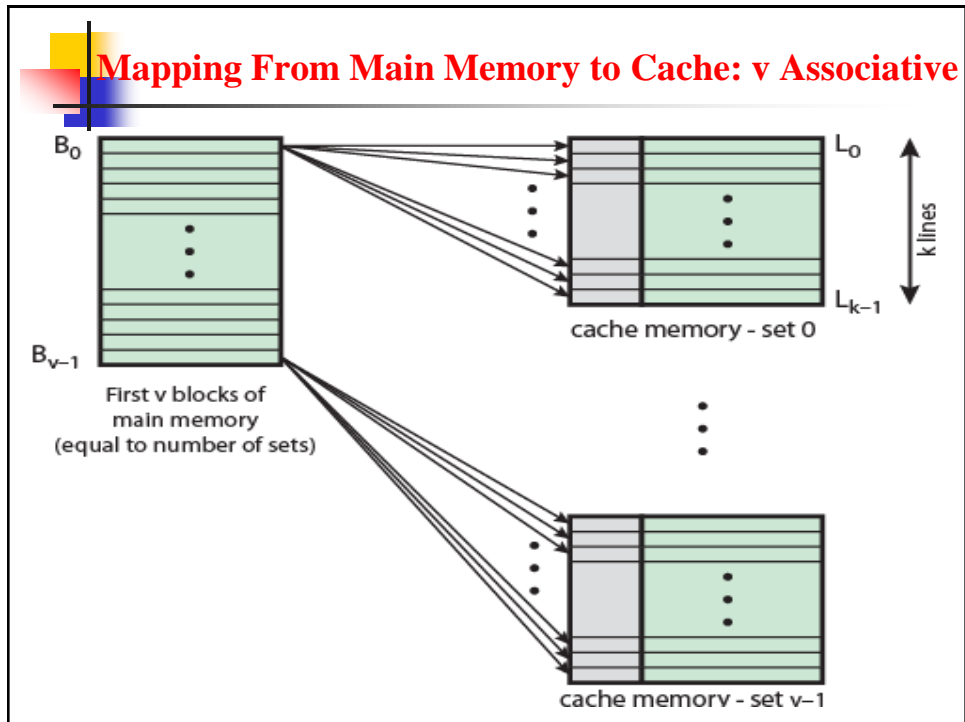
## Set Associative Mapping

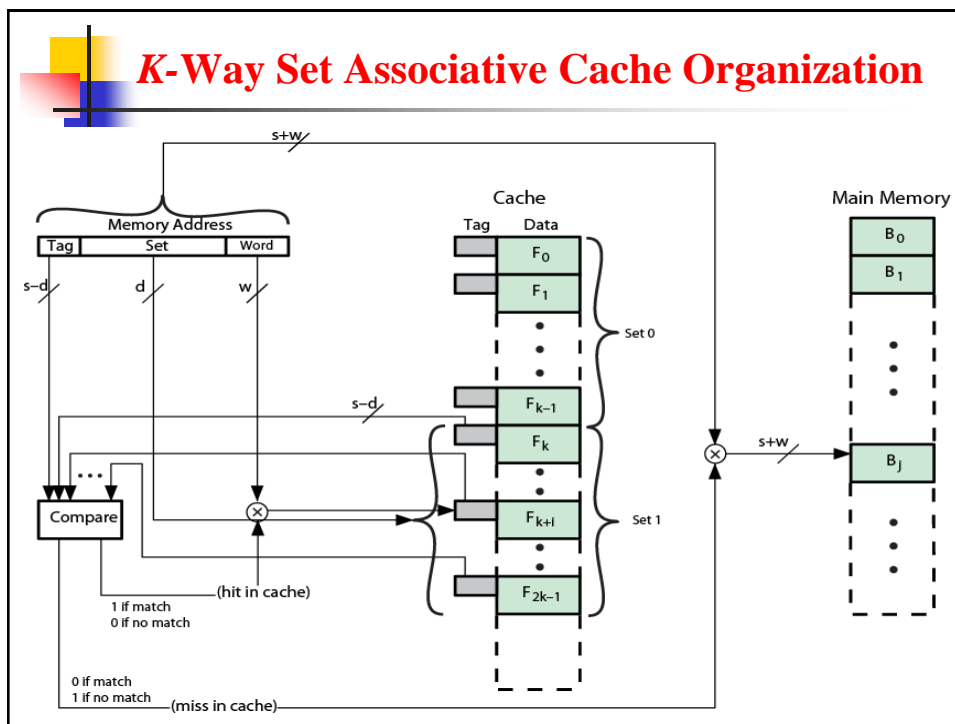
- Cache is divided into a number of sets
- Each set contains a number of lines
- A given block maps to any line in a given set
  - e.g. Block B can be in any line of set i
- e.g. 2 lines per set
  - 2 way associative mapping
  - A given block can be in one of 2 lines in only one set



## Set Associative Mapping Example

- 13 bit set number
- Block number in main memory is modulo  $2^{13}$
- 000000, 00A000, 00B000, 00C000 ... map to same set





### Set Associative Mapping Address Structure

Tag 9 bit	Set 13 bit	Word 2 bit
-----------	------------	------------

- Use set field to determine cache set to look in
- Compare tag field to see if we have a hit
- Examples
 

Address	Tag	Data	Set number
1FF 7FFC	1FF	12345678	1FFF
001 7FFC	001	11223344	1FFF

The diagram illustrates the mapping of a 16 MByte Main Memory to a 16 Kline Cache. The Main Memory address (binary) is split into a Tag (9 bits) and Set + Word (13 bits). The Main Memory is divided into 16 MByte blocks (000 to 1FF). The 16 Kline Cache is divided into 16 Kline blocks (000 to 1FF). The mapping shows that the 16 MByte Main Memory is mapped to the 16 Kline Cache, with the Tag and Set + Word fields used for addressing.

**Main Memory Address (binary)**

Tag (9 bits) | Set + Word (13 bits)

**Main Memory Address =**

Tag (9 bits) | Set (13 bits) | Word (2 bits)

**16 MByte Main Memory**

000 00000000 00000000 00000000 00000000 13579246

000 00000000 11111111 11111111 1000 11235813

02C 000101000000000000000000 77777777

02C 000101100000000000000000 11235813

02C 000101100011001100110011 FEDCBA98

02C 000101100111111111111111 12345678

1FF 111111110000000000000000

1FF 111111110000000000000000

1FF 111111111111111111111111 11223344

1FF 111111111111111111111111 24682468

**16 Kline Cache**

Tag (9 bits) | Data (32 bits) | Set Number (9 bits) | Tag (9 bits) | Data (32 bits)

000 13579246 0000 02C 77777777

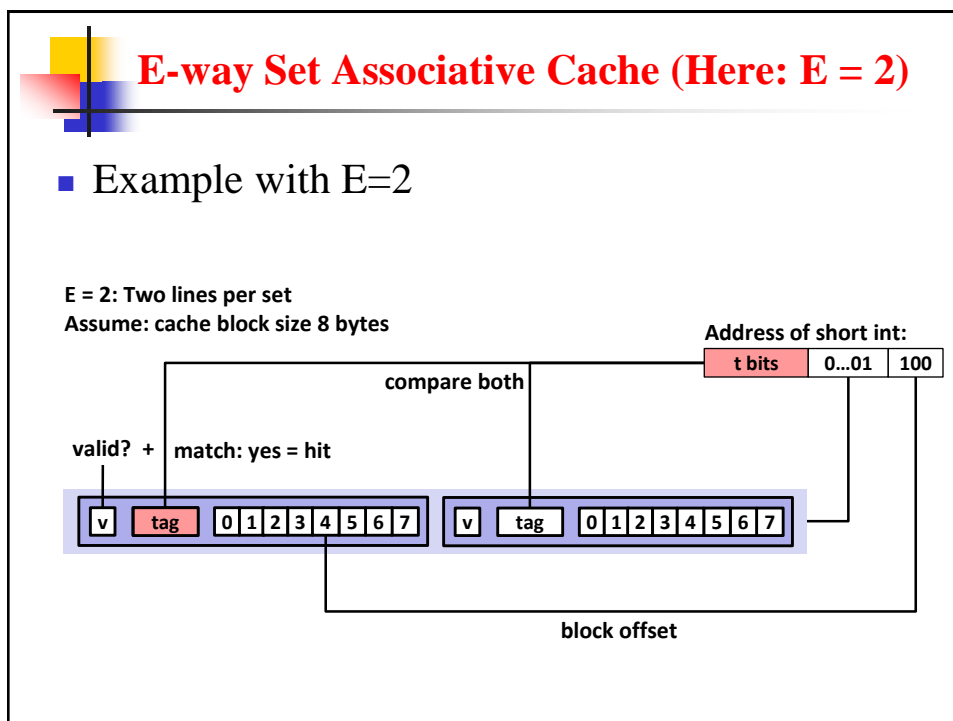
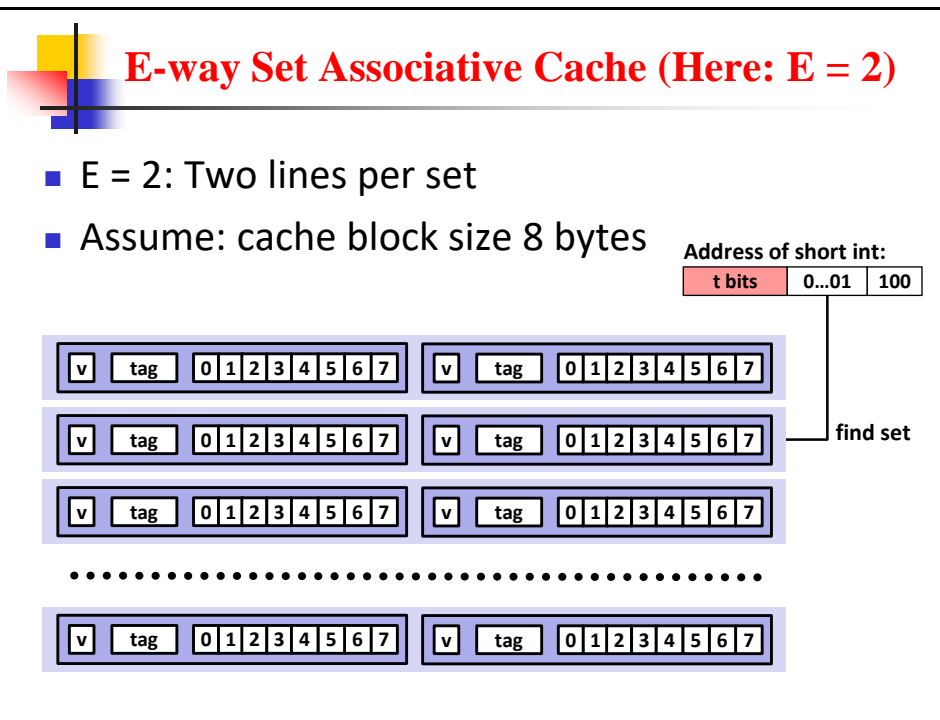
02C 11235813 0001 02C 11235813

02C FEDCBA98 0CE7 02C 77777777

1FF 11223344 1FFE 1FF 11223344

02C 12345678 1FFF 02C 24682468

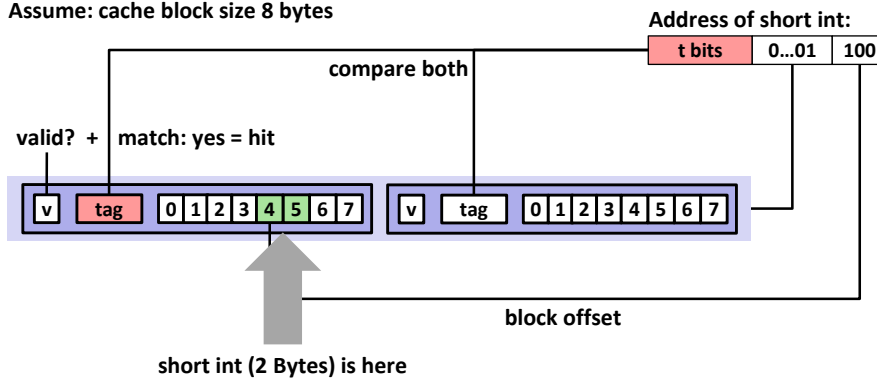
## 37



## E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

Assume: cache block size 8 bytes



### No match:

- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), ...

## 2-Way Set Associative Cache Simulation

### Example of 2 way


t=2	s=1	b=1
xx	x	x

M=16 byte addresses, B=2 bytes/block,  
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0	[0000 <sub>2</sub> ],	miss
1	[0001 <sub>2</sub> ],	hit
7	[0111 <sub>2</sub> ],	miss
8	[1000 <sub>2</sub> ],	miss
0	[0000 <sub>2</sub> ]	hit


	v	Tag	Block
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	0		



## Replacement algorithm

---


- When all lines are occupied, bringing in a new block requires that an existing line be overwritten.
- **Direct mapping**
  - No choice possible with direct mapping
  - Each block only maps to one line
  - Replace that line
- **Associative and Set Associative mapping**
  - Algorithms must be implemented in hardware for speed
  - Least Recently used (LRU)
    - replace that block in the set which has been in cache longest with no reference to it




---


- Implementation: with 2-way set associative, have a USE bit for each line in a set. When a block is read into cache, use the line whose USE bit is set to 0, then set its USE bit to one and the other line's USE bit to 0.
- Probably the most effective method
- **First in first out (FIFO)**
  - replace that block in the set which has been in the cache longest
  - Implementation: use a round-robin or circular buffer technique (keep up with which slot's "turn" is next)
- **Least-frequently-used (LFU)**
  - replace that block in the set which has experienced the fewest references or hits
  - Implementation: associate a counter with each slot and increment when used






↓

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B											
2		E	*									
3				R								



↓


Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*							
2		E	*									
3				R								



# LRU

↓


Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*							
2		E	*									
3				R								



# LRU

↓


Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*							
2		E	*			A						
3				R								



# LRU

↓


Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*							
2		E	*			A						
3				R			*					



# LRU

↓


Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*							
2		E	*			A						
3				R			*					



# LRU

↓


Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*			E				
2		E	*			A						
3				R			*					



# LRU

↓


Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*			E				
2		E	*			A						
3				R			*					



# LRU

↓


Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*			E				
2		E	*			A			B			
3				R			*					



# LRU

↓


Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*			E		*		
2		E	*			A			B			
3				R			*					



# LRU

↓


Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*			E		*		
2		E	*			A			B			
3				R			*					



# LRU

↓


Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*			E		*		
2		E	*			A			B			
3				R			*				A	



# LRU

↓

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*			E		*		
2		E	*			A			B			
3				R			*				A	



# LRU

↓

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*			E		*		
2		E	*			A			B			R
3				R			*				A	



## LRU

- 8 page faults

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*			E		*		
2		E	*			A			B			R
3				R			*				A	




## LFU




Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B											
2												
3												






↓

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B											
2												
3												




↓

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B											
2		E										
3												




↓

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B											
2		E	2									
3												




↓

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B											
2		E	2									
3				R								




↓

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				2							
2		E	2									
3				R								



↓


Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				2							
2		E	2									
3				R		A						



LFU

↓


Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				2							
2		E	2									
3				R		A	R					



LFU

↓


Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				2							
2		E	2					3				
3				R		A	R					



LFU

↓


Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				2				3			
2		E	2					3				
3				R		A	R					



LFU

↓


Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				2				3			
2		E	2					3		4		
3				R		A	R					



LFU

↓


Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				2				3			
2		E	2					3		4		
3				R		A	R				A	



LFU

↓


Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				2				3			
2		E	2					3		4		
3				R		A	R				A	R



## LFU

- 7 page faults

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				2				3			
2		E	2					3		4		
3				R		A	R				A	R



## Cache write policy

- When a line is to be replaced, must update the original copy of the line in main memory if any addressable unit in the line has been changed
- If a block has been altered in cache, it is necessary to write it back out to main memory before replacing it with another block (writes are about 15% of memory references)
- Must not overwrite a cache block unless main memory is up to date
- I/O modules may be able to read/write directly to memory
- Multiple CPU's may be attached to the same bus, each with their own cache



## ■ Write Through

- All write operations are made to main memory as well as to cache, so main memory is always valid
- Other CPU's monitor traffic to main memory to update their caches when needed
- This generates substantial memory traffic and may create a bottleneck
- Anytime a word in cache is changed, it is also changed in main memory
- Both copies always agree
- Generates lots of memory writes to main memory
- Multiple CPUs can monitor main memory traffic to keep local (to CPU) cache up to date
- Lots of traffic
- Slows down writes

Remember to write through to cache!



## ■ Write back

- When an update occurs, an UPDATE bit associated with that slot is set, so when the block is replaced it is written back first
- During a write, only change the contents of the cache
- Update main memory only when the cache line is to be replaced
- Causes “cache coherency” problems -- different values for the contents of an address are in the cache and the main memory
- Complex circuitry to avoid this problem
- Accesses by I/O modules must occur through the cache





- Multiple caches still can become invalidated, unless some cache coherency system is used. Such systems include:
  - Bus Watching with Write Through - other caches monitor memory writes by other caches (using write through) and invalidates their own cache line if a match
  - Hardware Transparency - additional hardware links multiple caches so that writes to one cache are made to the others
  - Non-cacheable Memory - only a portion of main memory is shared by more than one processor, and it is non-cacheable