COMPUTER ARCHITECTURE

Code: CT173

Part VIII: Memory Organization

MSc. NGUYEN Huu Van LONG

Department of Computer Networking and Communication,

College of Information & Communication Technology,

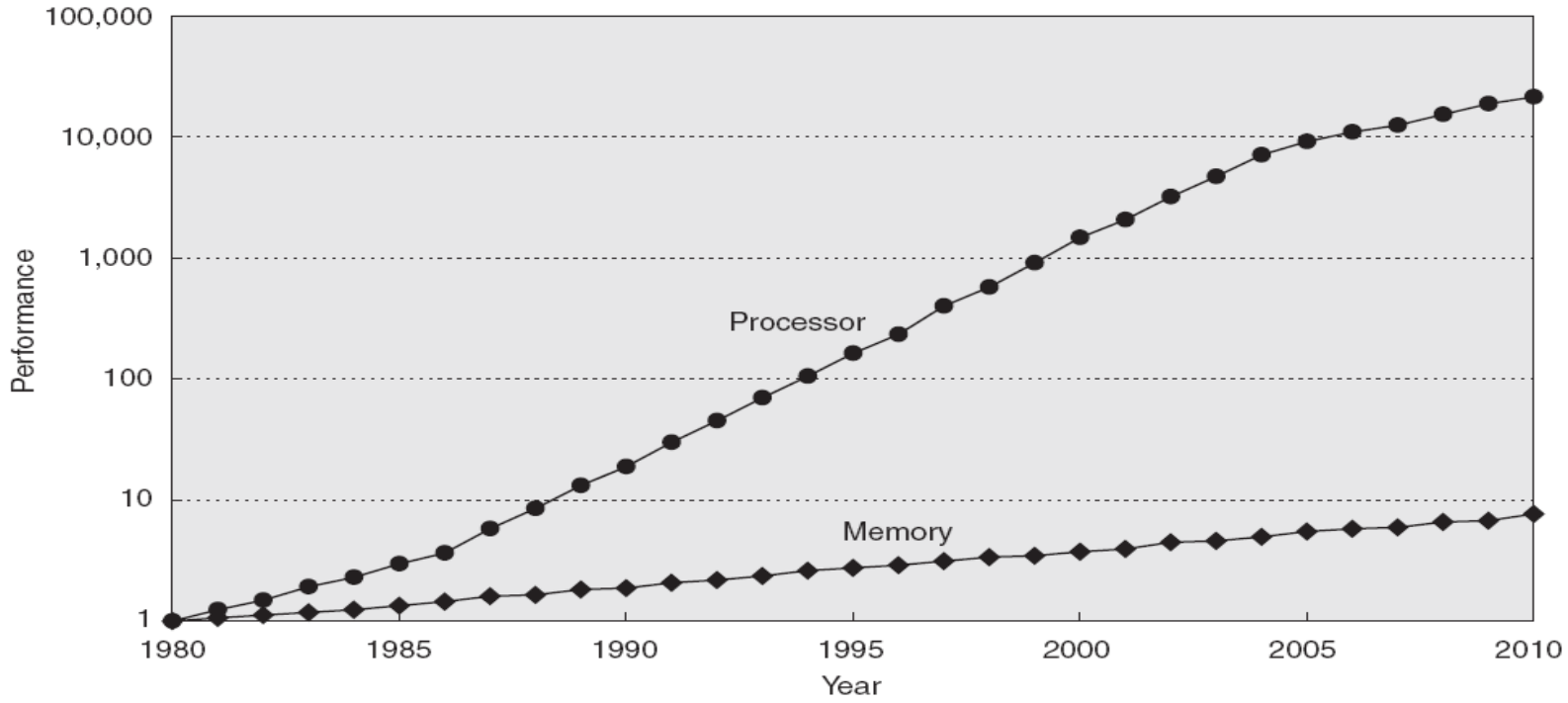
CanTho University

Agenda

- The goal of memory design
- Memory technologies
  - Volatile memory: SRAM, DRAM, SDRAM
  - Non volatile memory: ROM, HDD, SSD
- Principal of locality: Temporal Locality and Spatial Locality
- Memory hierarchy concept
  - The pyramid architecture: Speed and Size
  - Memory operation terminologies: Hit and Miss
- Cache Design
  - Types of Cache
  - Structure of memory address
  - Cache Operation
    - *Block Placement*
    - *Block Identification*
    - *Block Replacement*
    - *Write Strategy*

Goal of Memory Design

- To increase the performance of a computer through the memory system in order to:
  - Provide the user the illusion to use a memory that is simultaneously large and fast
  - Provide the data to the processor at high frequency
- The gap (latency) grows about 50% per year

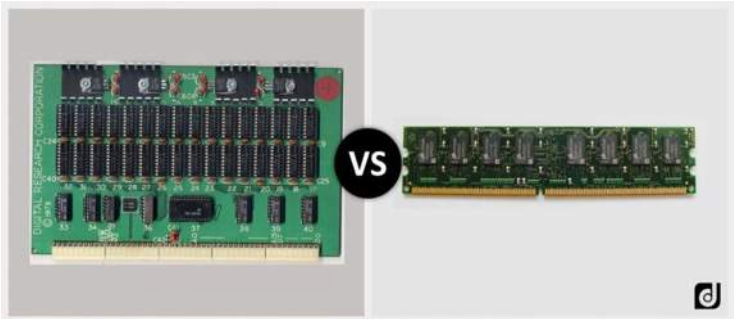


Memory Technologies

- Performance metrics for memory design
  - Latency is concern of cache
  - Bandwidth is concern of multiprocessors and I/O
  - Access time: between read request and when desired word arrives
  - Cycle time: minimum time between unrelated requests to memory
- **Volatile memory** vs **Non volatile memory**
  - Volatile memory needs constant power in order to retain data
  - Non volatile memory can retrieve stored information even after having been power cycled
- **Volatile memory types**
  - Static Random Access Memory **SRAM**
  - Dynamic Random Access Memory **DRAM**
    - Synchronous DRAM: **DDR** SDRAM, **DDR2** SDRAM…
- Non volatile memory types:
  - Read Only Memory **ROM**: PROM, EPROM, EEPROM
  - Hard Disk Drive **HDD**
  - Solid State Drive **SSD**
  - Optical Disc
  - Magnetic Tape

Volatile memory: SRAM vs DRAM

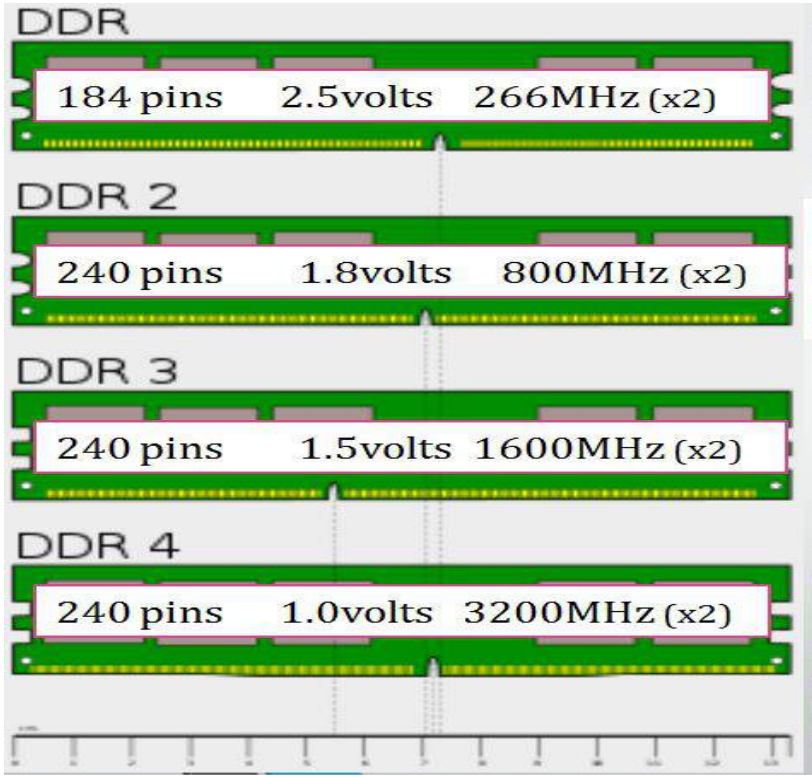
- DRAM used for main memory
  - Must be re-written after being read
  - Must also be periodically refreshed
    - Every nearly 8ms
    - Each row can be refreshed simultaneously
  - Requires **1 transistor/bit**
- SRAM used for cache
  - Requires low power to retain bit
  - Requires **6 transistors/bit**

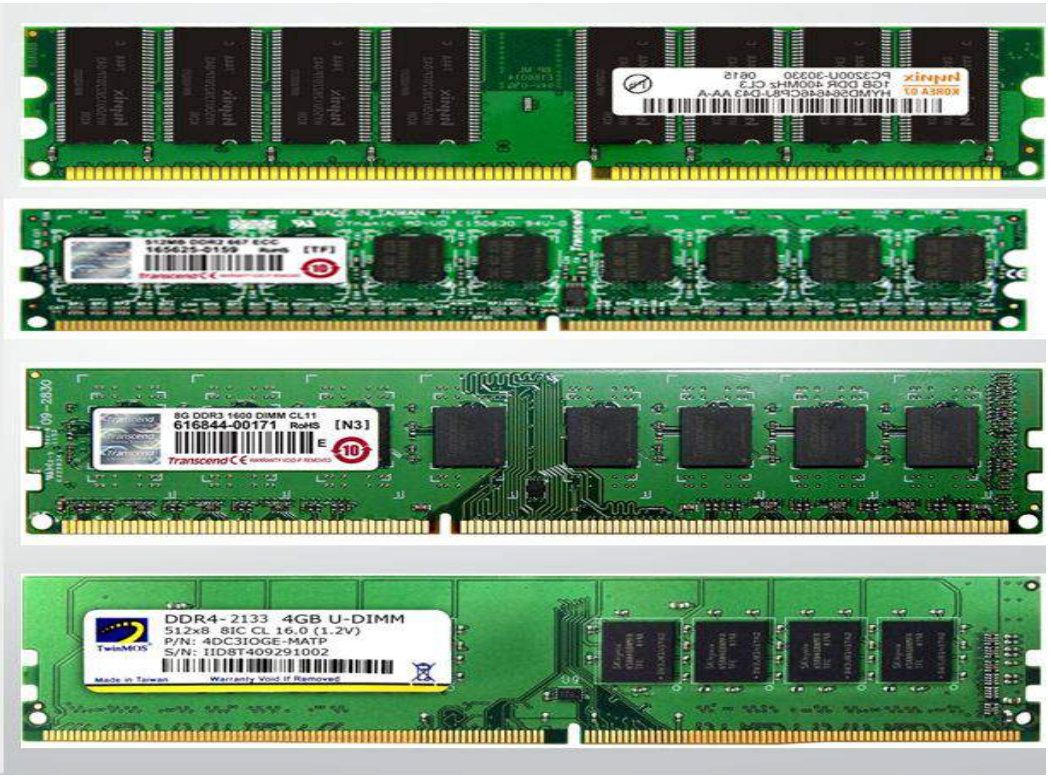


SRAM vs. DRAM

	SRAM	DRAM
Storage	Store data till the power is supplied	Store data only for few miliseconds even when power is supplied
Architecture	Uses an array of 6 transistors for each memory cell	Use a single transistor and capacitor for each memory cell
Data refreshing	Does not refresh the memory cell	Needs to refresh the memory cell after each reading of capacitor
Data access speed	Data access is faster	Data access is lower
Power usage	Consume more power	Consume less power
Implement density	Low density/Less memory per chip	High density/More memory per chip
Price	Cost per bit is high	Cost per bit is low
Size range	In MB	In GB

Volatile Memory: Evolution of SDRAM





DDR SDRAM Standard	Internal rate (MHz)	Bus clock (MHz)	Prefetch	Data rate (MT/s)	Transfer rate (GB/s)	Voltage (V)
SDRAM	100-166	100-166	1n	100-166	0.8-1.3	3.3
DDR	133-200	133-200	2n	266-400	2.1-3.2	2.5/2.6
DDR2	133-200	266-400	4n	533-800	4.2-6.4	1.8
DDR3	133-200	533-800	8n	1066-1600	8.5-14.9	1.35/1.5
DDR4	133-200	1066-1600	8n	2133-3200	17-21.3	1.2

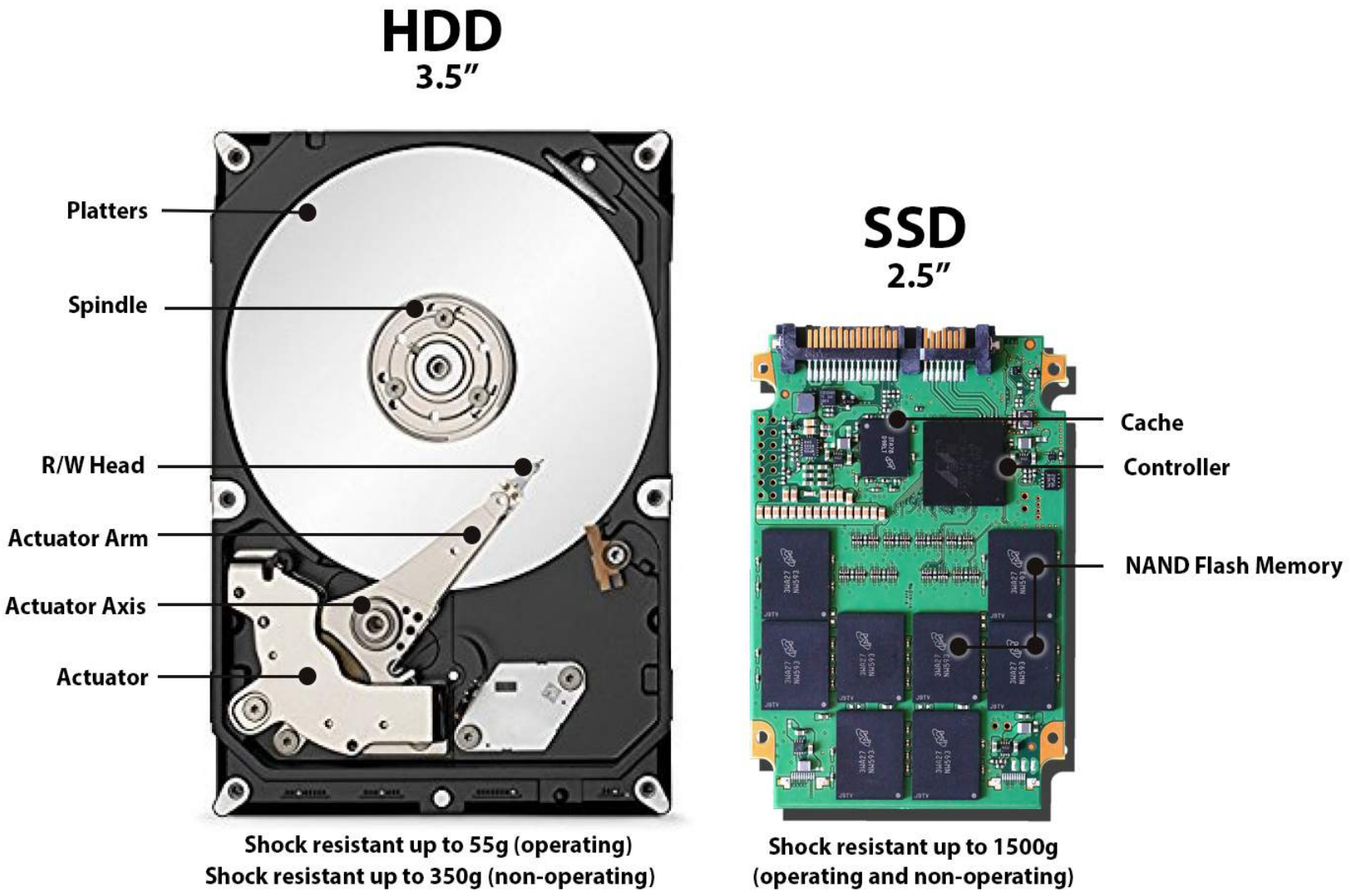
Non Volatile Memory: ROM

- ROM (Read Only Memory): mainly used to store firmware which could not be changed after manufacture
  - Disadvantage in many applications because bugs, security issues, new features can not be fixed
  - Used only for large production runs with well verified data
- PROM (Programmable ROM): the data could be reprogrammed after manufacture
  - Allow to test on a subset of the devices in an order before burning data into all of them
  - Used in: Microcontrollers, Video game consoles, Mobile phones...
- EPROM (Erasable Programmable ROM): the data could be erased by exposing it to strong ultraviolet source
  - Become unreliable after several thousand cycles of erasing
- EEPROM (Electrically Erasable Programmable ROM): the data could be erased by using the electric
  - Limited life for erasing and reprogramming
- Flash Memory: a type of EEPROM, high speed and high density
  - Two types: NAND and NOR

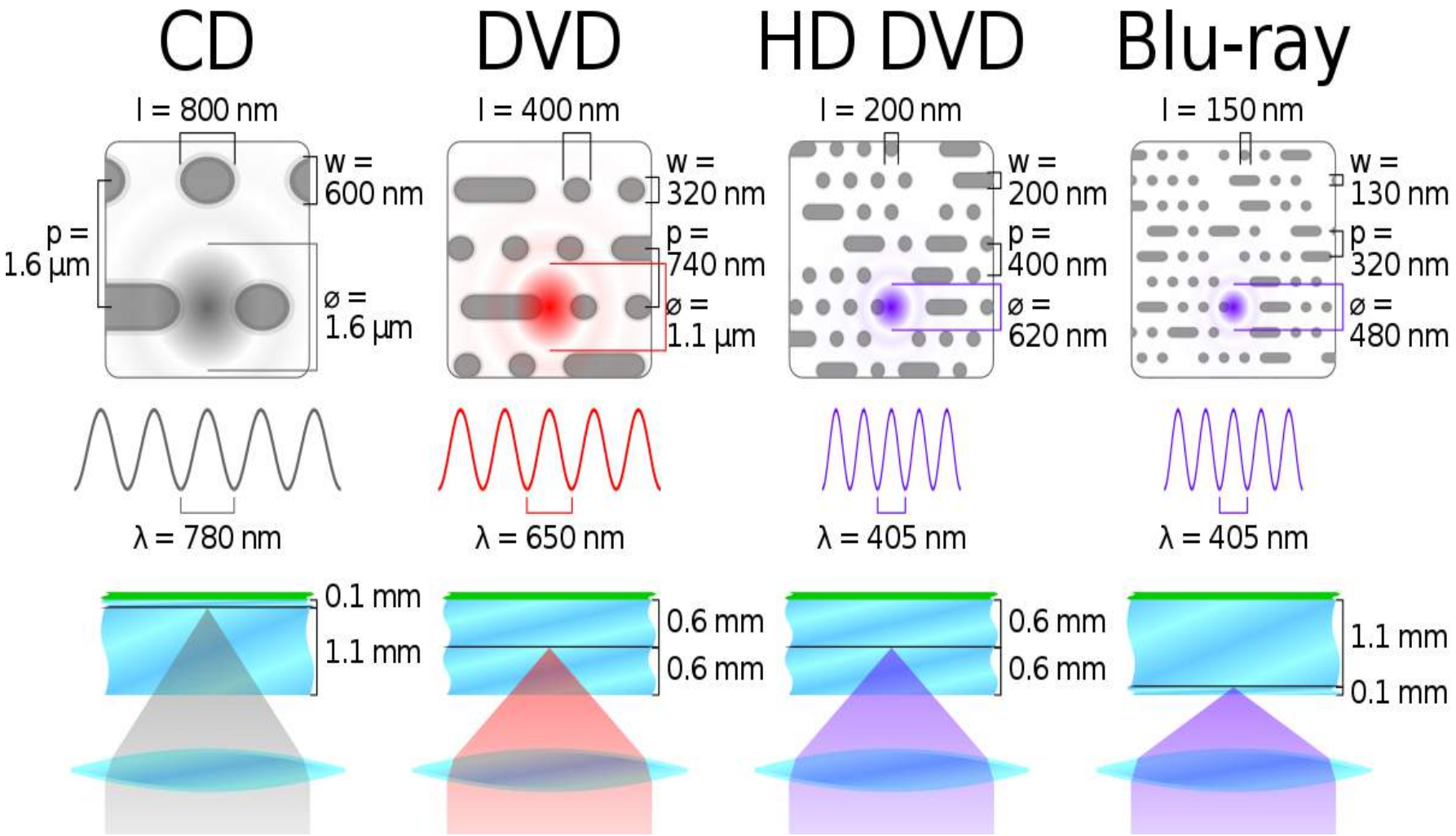
Non volatile memory: HDD vs SSD

	HDD	SSD
Stands for	Hard Disk Drive	Solid State Drive
Speed	HDD has higher latency, longer read/write times, and supports fewer IOPs (input output operations per second) compared to SSD.	SSD has lower latency, faster read/writes, and supports more IOPs (input output operations per second) compared to HDD.
Heat, Electricity, Noise	Hard disk drives use more electricity to rotate the platters, generating heat and noise.	Since no such rotation is needed in solid state drives, they use less power and do not generate heat or noise.
Defragmentation	The performance of HDD drives worsens due to fragmentation; therefore, they need to be periodically defragmented.	SSD drive performance is not impacted by fragmentation. So defragmentation is not necessary.
Components	HDD contains moving parts - a motor-driven spindle that holds one or more flat circular disks (called platters) coated with a thin layer of magnetic material. Read-and-write heads are positioned on top of the disks; all this is encased in a metal cas	SSD has no moving parts; it is essentially a memory chip. It is interconnected, integrated circuits (ICs) with an interface connector. There are three basic components - controller, cache and capacitor.
Weight	HDDs are heavier than SSD drives.	SSD drives are lighter than HDD drives because they do not have the rotating disks, spindle and motor.
Dealing with vibration	The moving parts of HDDs make them susceptible to crashes due to vibration.	SSD drives can withstand vibration up to 2000Hz, which is much more than HDD.

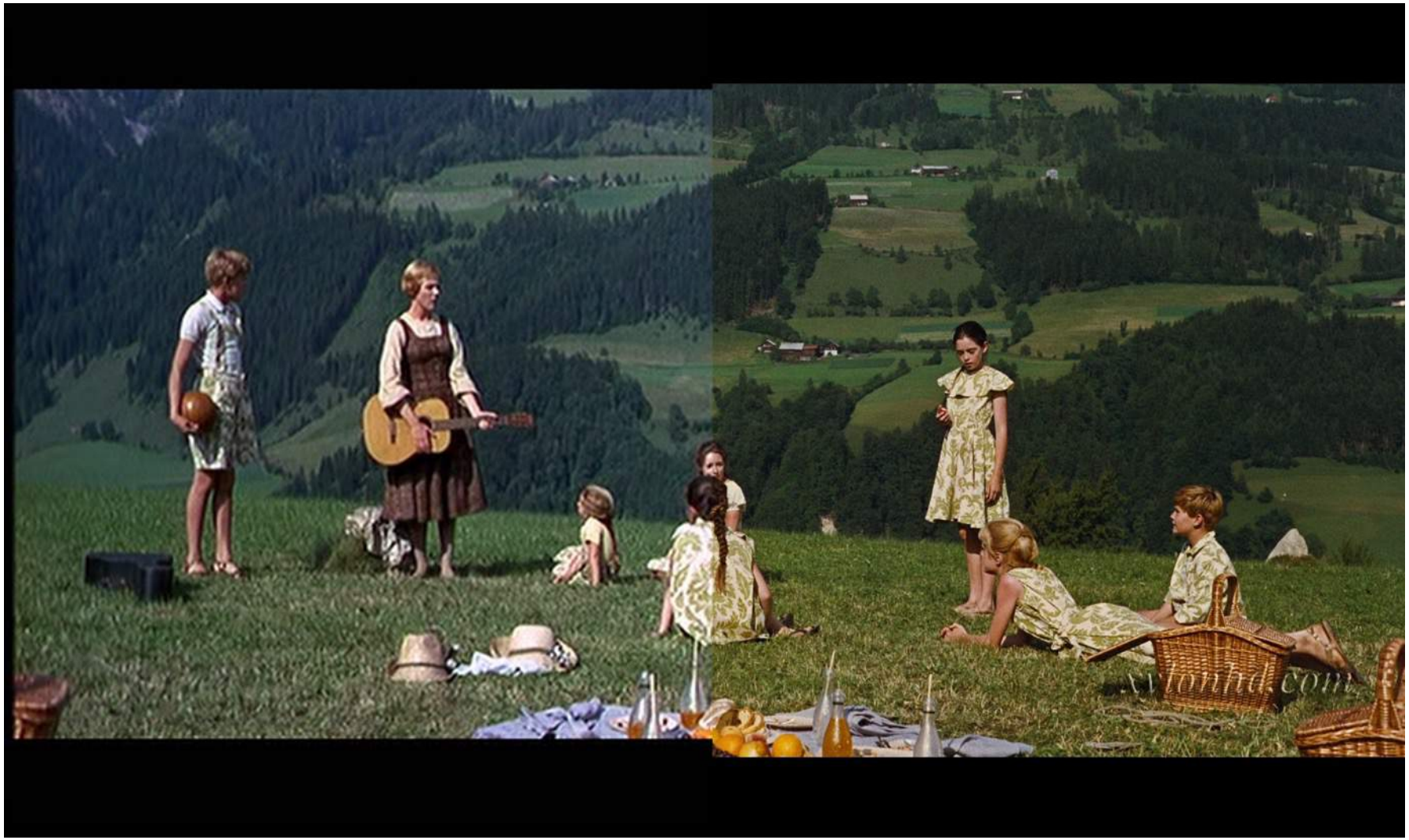
Non volatile memory: HDD vs SSD



Non volatile memory: Optical Disc



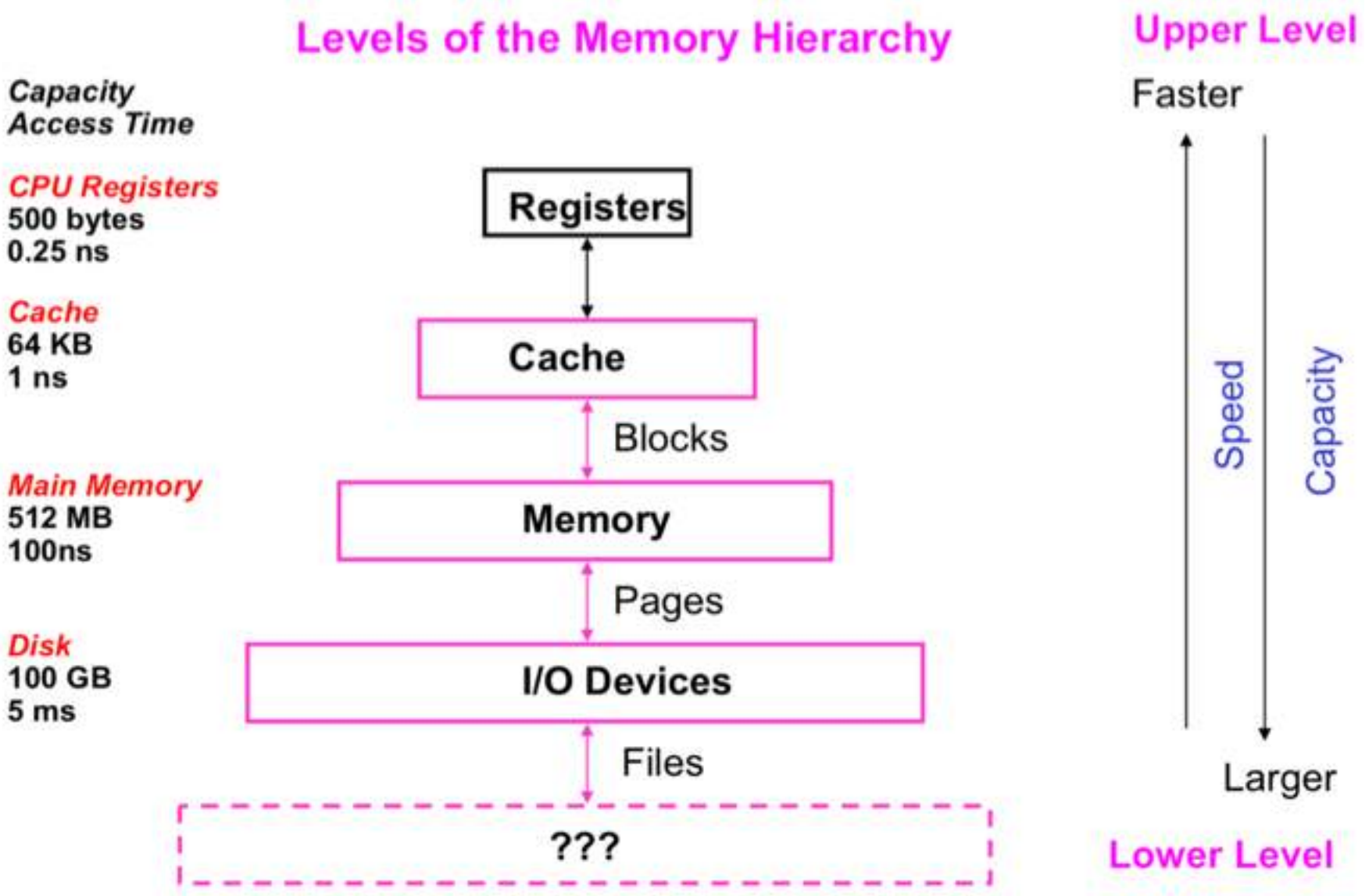
Non volatile memory: Optical Disc



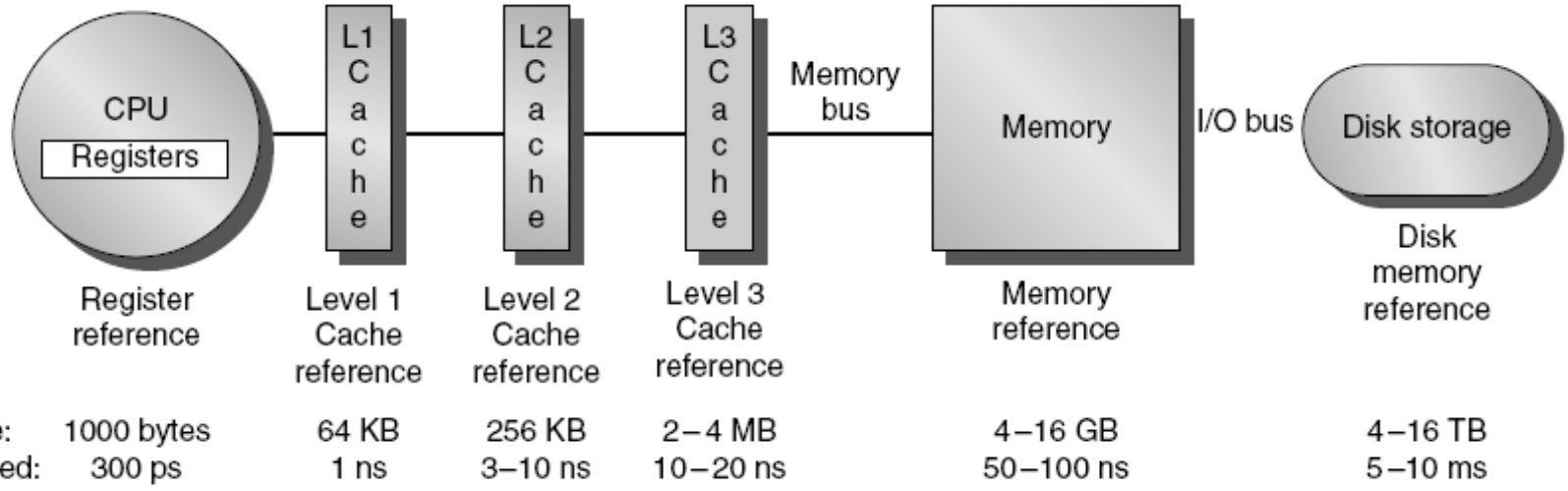
Cache Design

- **Locality of references**: is a term for the phenomenon in which the same values, or related storage locations, are frequently accessed, depending on the memory access pattern
  - Program access a relatively small portion of the address space at any instant of time
  - 90/10 rule: 10% of code executed 90% of time
- There are two types of locality:
  - Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loop, reuse)
  - Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straight-line code, array access)
- **Cache** exploit both types of predictability to enhance the average memory time access of computer system
  - Exploit temporal locality by keeping the contents of recently accessed locations
  - Exploit spatial locality by fetching blocks of data around recently accessed location

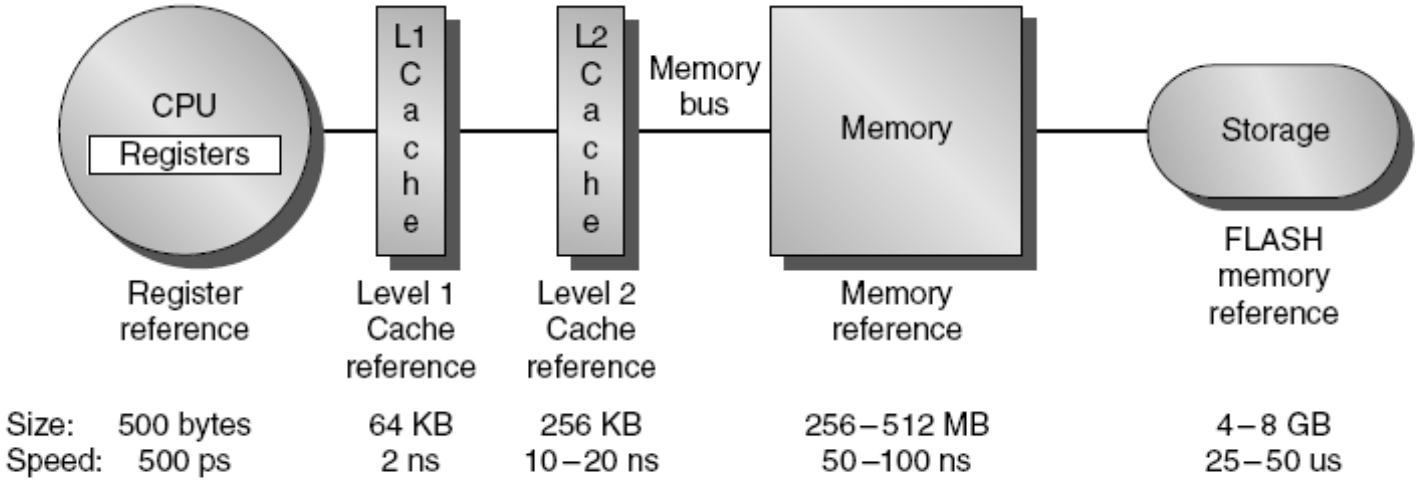
Memory Hierarchy



Memory Hierarchy



(a) Memory hierarchy for server



(b) Memory hierarchy for a personal mobile device

Memory Hierarchy Terminology

- **Hit**: *data found in a block of the upper level*
  - **Hit Rate**: number of memory accesses that find the data in the upper level with respect to the total number of memory accesses

    Hit Rate = # Hits / # Memory Accesses
  - **Hit Time**: time to access the data in the upper level of the hierarchy, including the time needed to decide if the attempt of access will result in a hit or miss
- **Miss**: *data must be taken from the lower level*
  - **Miss Rate**: number of memory accesses not finding the data in the upper level with respect to the total number of memory accesses

    Miss Rate = # Misses / # Memory Accesses

    Hit Rate + Miss Rate = 1
  - **Miss Penalty** : time needed to access the lower level and to replace the block in the upper level

    Miss Time = Hit Time + Miss Penalty

    Hit Time << Miss Penalty

Basic concepts of Cache

- The memory hierarchy is composed of several levels, but data are copied between two adjacent levels, i.e. cache and main memory
- The cache (**upper** level) is smaller, faster and more expensive than the main memory (**lower** level)
  - The minimum chunk of data that can be copied in the cache is called the **block** or **cache line**
  - To exploit the spatial locality, the block size must be a **multiple** of the word size in memory
    - Example: 128-bit block size = 4 words of 32-bit
- **Each entry** in the typical cache includes
  - **Valid bit** to indicate if this position contains valid data or not. At the bootstrap, all the entries in the cache are marked as INVALID
  - **Cache Tag** contains the value that univocally identifies the memory address corresponding to the stored data
  - **Cache Data** contains a copy of data (block or cache line)

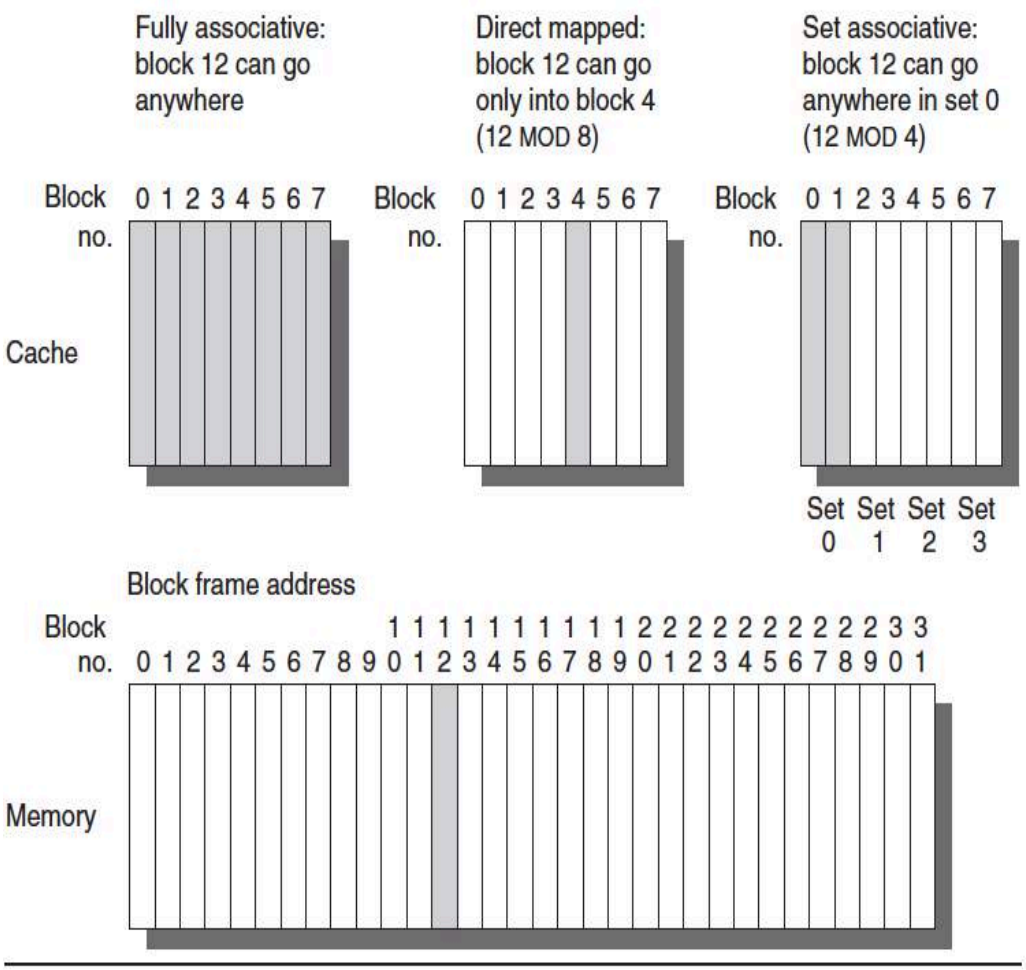
Basic concepts of Cache

- The Number of blocks in cache is given by: Cache Size/Block Size
  - Cache size is 64 KByte
  - Block size is 128-bit (16 Bytes)

  ➔ Number of blocks = 4 K blocks
- Cache Hit and Cache Miss in the Cache
  - Cache Hit: the requested data is found in one of the cache blocks
  - Cache Miss: the requested data is not found in one of the cache blocks → need to access to the lower level of the memory hierarchy
- Four memory hierarchy questions
  - **Q1: Block Placement** → Where can a block be placed in the upper level?
  - **Q2: Block Identification** → How is a block found if it is in the upper level?
  - **Q3: Block replacement** → Which bock should be replaced on a miss?
  - **Q4 Write strategy** → What happens on a write?

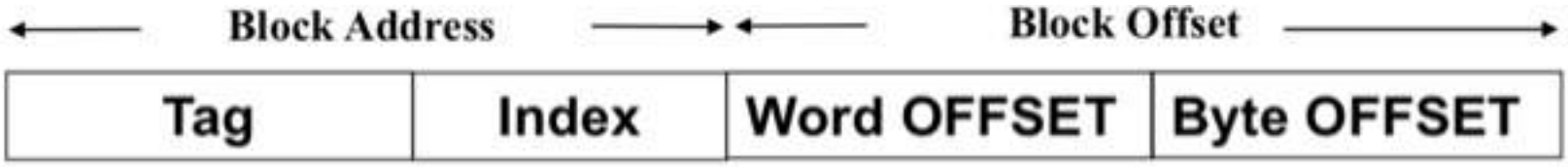
Question 1 and Question 2

- **Direct Mapped**: each memory location corresponds to one and only one cache location
- **Fully associative cache**: the memory block can be placed in any position of the cache
- **N-way set associative**: cache composed of sets, each set composed of n blocks. Each memory block corresponds to a single set of the cache and the block can be placed in whatever block of the n blocks of the set



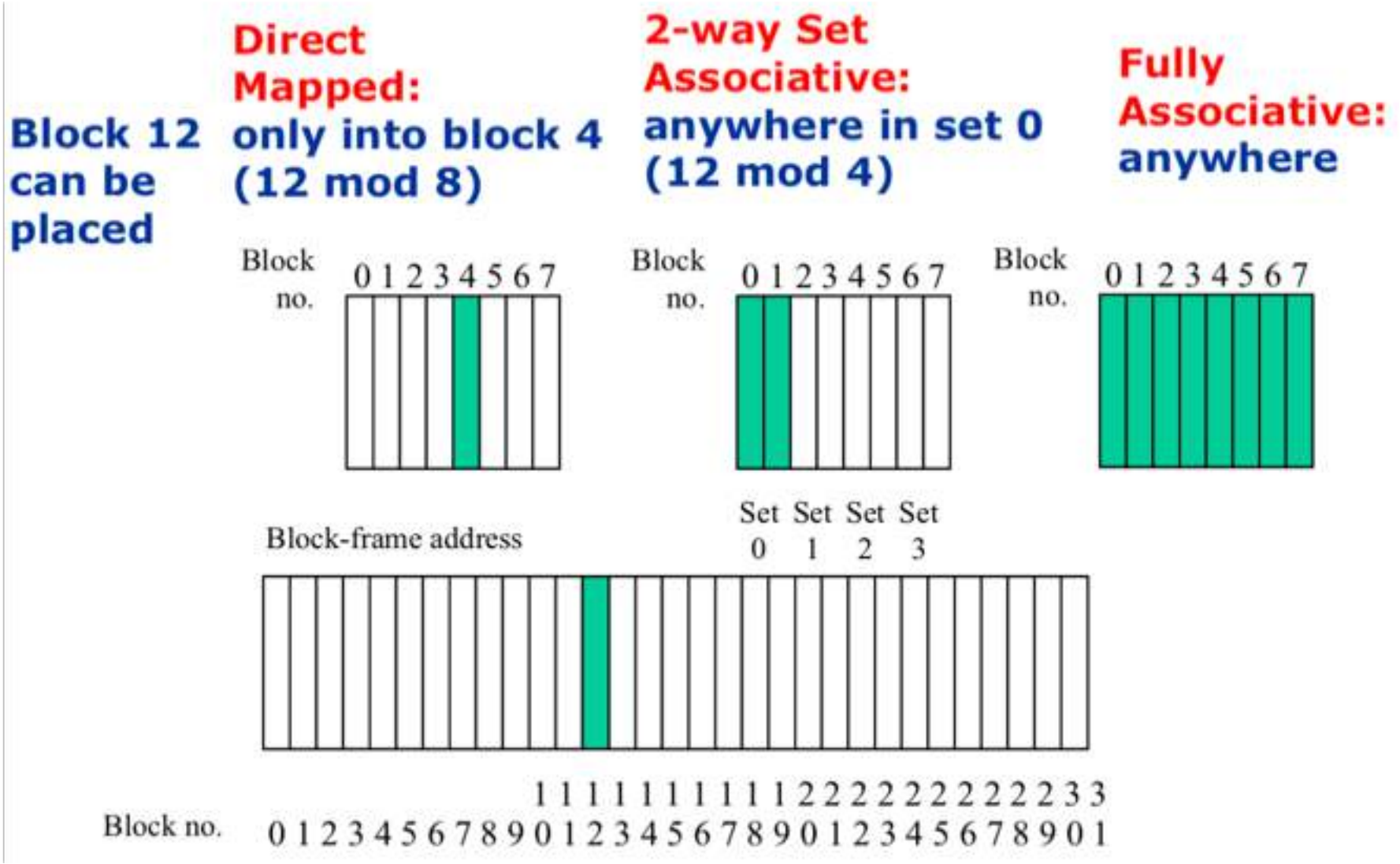
Question 1 and Question 2

- A **Physical Memory Address** (**N** bit) could compose:
  - **Byte offset** (**B** bit): identify the given byte *within the word*
    - If the memory *is not byte addressable*: B = 0
  - **Word offset** (**W** bit): identify the given word *within the block*
    - If the block contains only one word: W = 0
  - **Index** (**I** bit) identifies the block within the cache (upper level)
  - **Tag** (**T = N – B – W – I** bit) to be compared to the cache tag associated to the block selected by the index
- The physical memory address is varied in size and some fields in order to the type of mapping data in cache

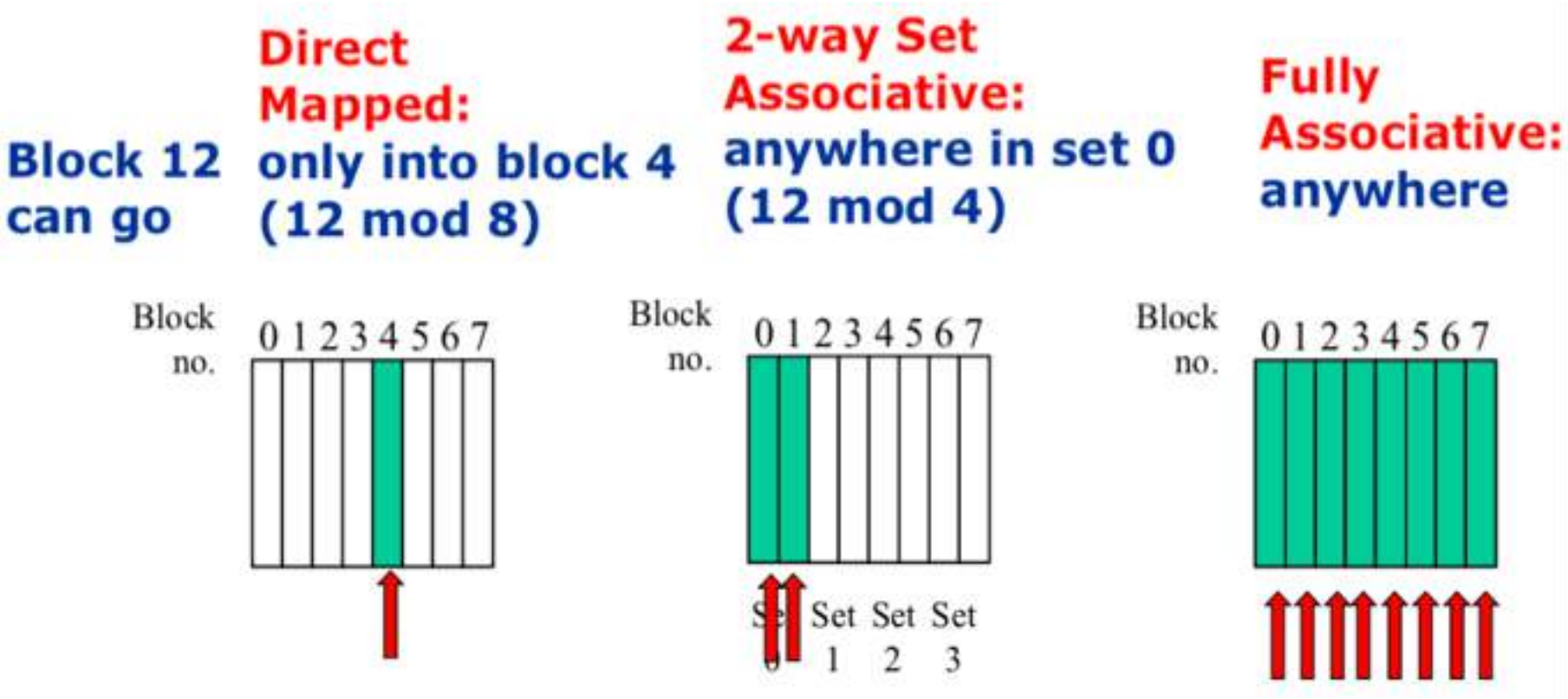


Recap Q1: Block Placement

- How can block 12 be placed in a 8 block cache?



Recap Q2: Block Identification



- **Direct Mapping**: Compute Block position, compare Tag of block and verify Valid bit
- **Fully Associative**: Compare Tags in every block and verify Valid bit
- **N-way set associative**: Compute Set position, compare Tags of the set and verify Valid bit

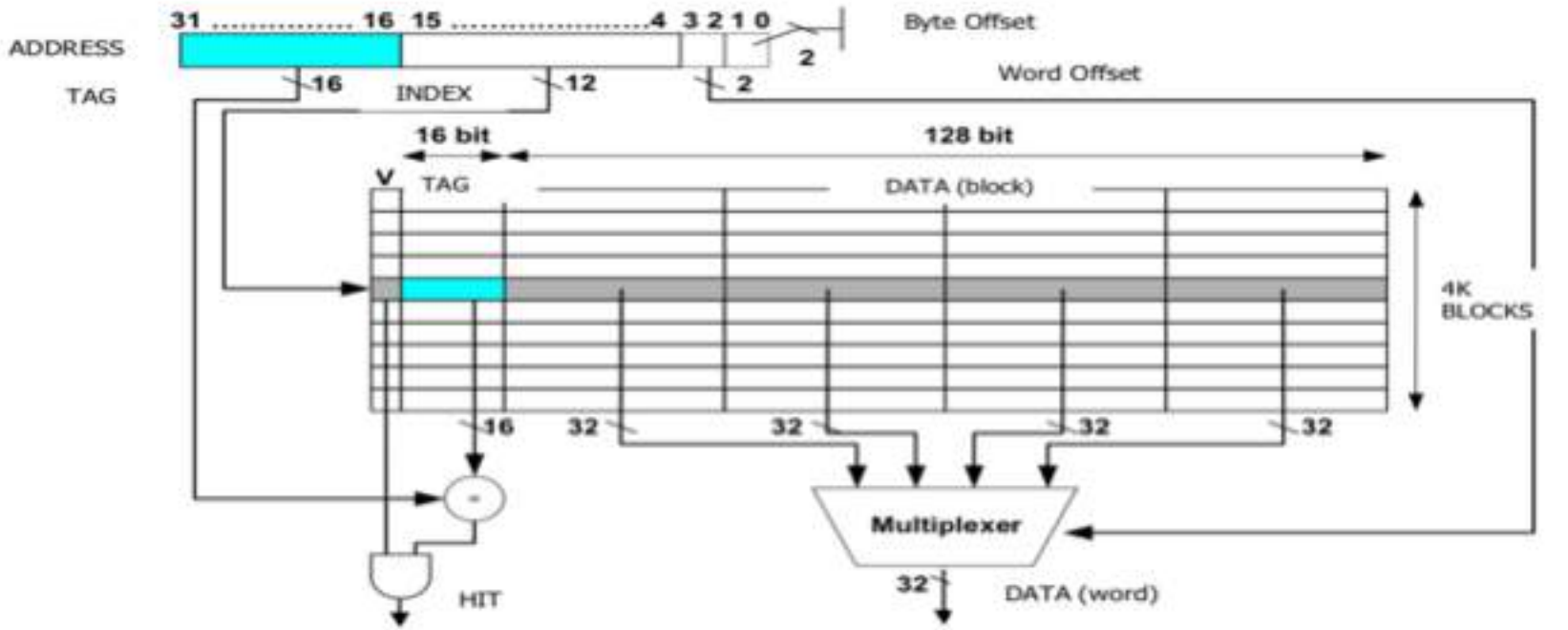
Direct Mapped Cache

- To find which block in cache to place (map) a block from memory

Cache_Block_Index = Mem_Block_Index MOD (# Cache Blocks)

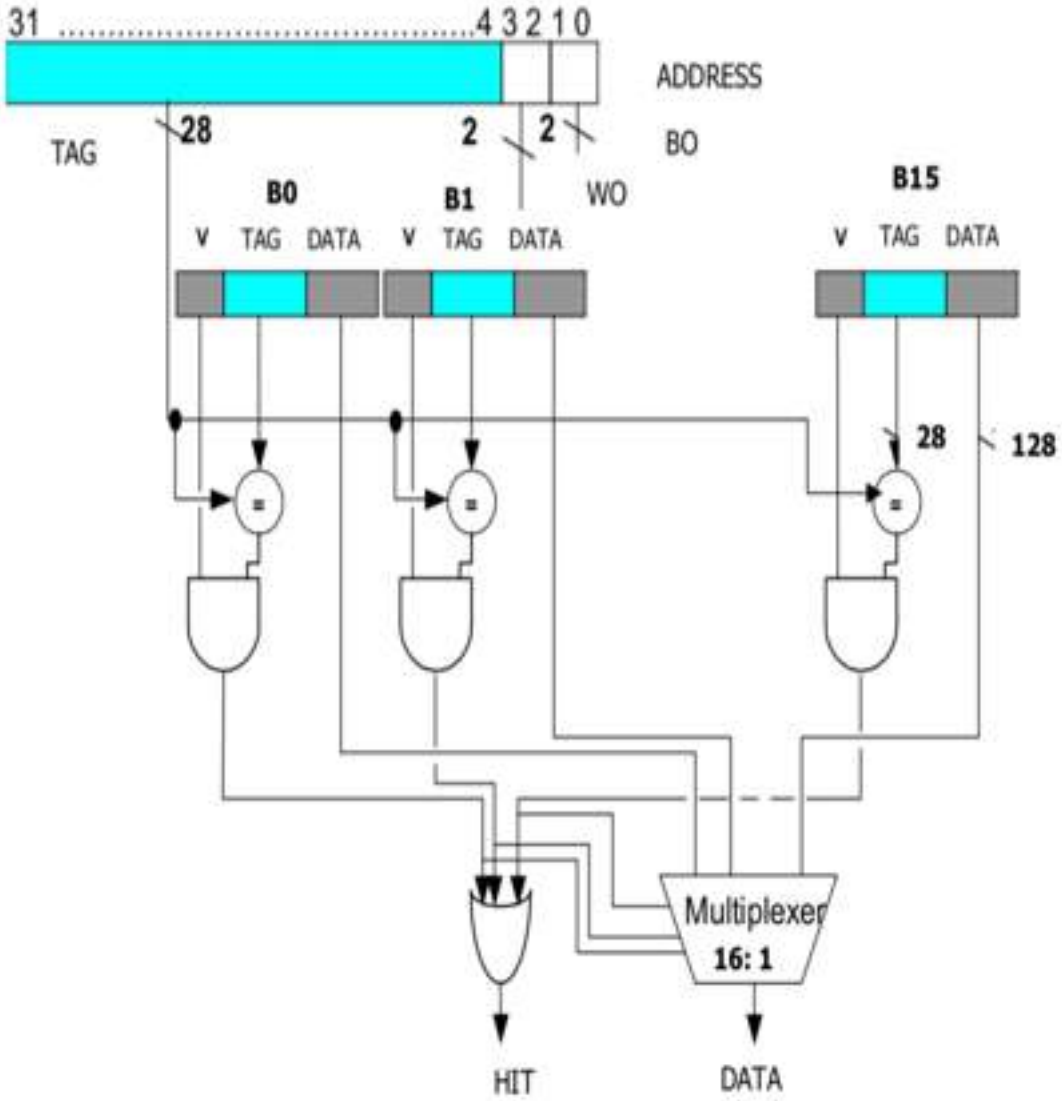
- Memory address composed of N = 32 bit
- Cache size 64 KB = 2^{16} Bytes; Block size 128 bit = 16 Bytes
- # Cache Blocks = Cache Size / Block Size = $2^{16}/16 = 2^{12}$ = 4K blocks → I = 12 bits
- 1 Word = 4 Byte → B = 2 bits; 1 Block = 4 Words → W = 2 bits
- Structure of the memory address

Tag (16 bit)	Index (12 bit)	WO (2 bit)	BO (2 bit)



Fully Associative Cache

- In a fully associative cache, all the cache blocks must be checked during the search of the block
- The block index does not exist in the memory address, there are the Tag bits only
- Memory address N = 32 bits
- Cache size 256 Bytes
- Block size 128 bit = 16 Bytes = 4 Words
- # blocks = Cache Size / Block Size = 256 / 16 = 16 blocks
- Structure of the memory address:



Tag (n-4 bit)	WO (2 bit)	BO (2 bit)

N way Set Associative Cache

- Cache composed of sets, each set composed of n blocks:

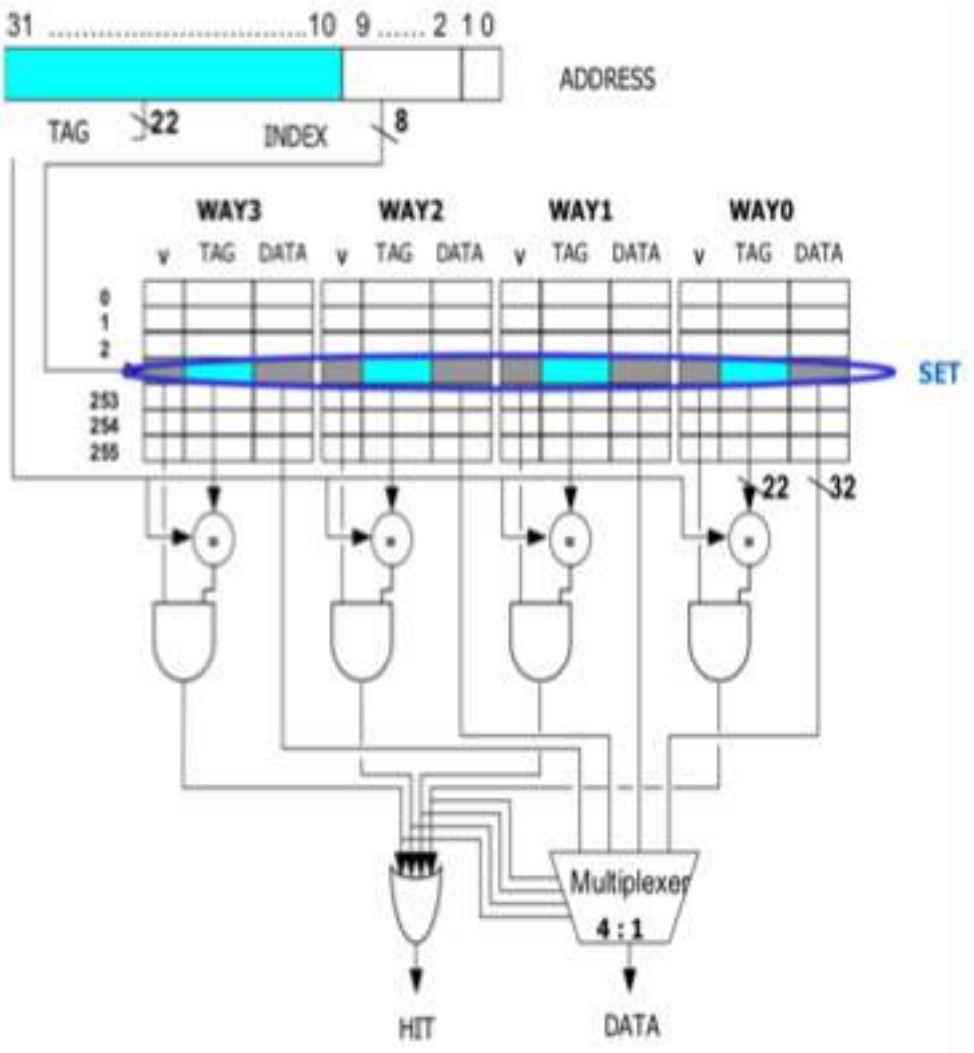
blocks = Cache Size / Block Size

sets = Cache Size / (Block Size x n)

- To find which set in cache to place the block of data:

Set_Index = Block_Index_Mem MOD # Set

- The Block index field changes to Set index field
- Memory address: N = 32 bit
- Cache size 4KByte
- Block size 32 bit
- # blocks = Cache Size / Block Size = 1K blocks
- n = 4
- # sets = Cache Size / (Block size x n) = 256 sets
- Structure of the memory address:



Cache Mapping Techniques

- There is a **critical tradeoff** in cache performance between various cache mapping techniques. In order for the cache to have good performance, these two aspect should be maximizing: **Hit Ratio (Miss Ratio)** and **Hit Time (Miss Time)**

Cache Type	Hit Ratio	Hit Time
Direct Mapped	Good	Best
Fully Associative	Best	Moderate
N-Way Set Associative, N>1	Very Good, Better as N Increases	Good, Worse as N Increases

Question 3: Block Replacement

- **For a direct mapped cache**, there is only one candidate to be replaced → *no need of any block replacement strategy when Cache Miss*
- For a **fully associative cache**, in case of a Cache Miss, we need to decide which block to replace → *any block is a potential candidate for the replacement*
- For a **set-associative cache**, we need to *select among the blocks in the given set*
- **Main strategies** used to choose the block to be replaced *in fully or N-way set associative caches*:
  - **Random (or pseudo random):** candidate blocks are randomly selected
  - **LRU (Least Recently Used):** the block replaced is the one that has been unused for the longest time
  - **FIFO (First In First Out)**: the block replaced is the one that has been placed for the oldest time
  - The other algorithms: **Most Frequently Used – MFU**, **Most Recently Used – MRU**, **Least Frequently Used – LRU**

Question 3: Block Replacement

- LRU: hardest
- FIFO: easier, approximates LRU (oldest rather than LRU)
- Random: easiest
- Results:
  - Data cache misses per 1000 in L1, on average
  - Average across ten SPECint2000 / SPECfp2000 benchmarks

Associative	2 - ways			4 - ways			8 - ways		
Size	LRU	Rand	FIFO	LRU	Rand	FIFO	LRU	Rand	FIFO
16K	114.1	117.3	115.5	111.7	115.1	113.1	109.0	111.8	110.4
64K	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
256K	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

Question 4: Write Strategy

- **Two option** when successfully writing to cache (**Write Hit**):
  - **Write through**: The information is *written to both* the block in the *cache* and to the block in the *lower-level memory*
  - **Write back**: The information is *written only* to the block in the *cache*. The modified cache block *is written to main memory only when it is replaced* (**Cache Miss**)
- To reduce the frequency of writing back blocks on replacement, a **Dirty bit** is used to indicate whether the block was modified in the cache (dirty) or not (clean)
  - If clean, no write back since identical information to the cache is found
- **Write Through** vs **Write Back**
  - **Write Through**: *simply to be implemented*, but requires a **Write Buffer** to do not wait for the lower level
    - Write Buffer is just a FIFO implementation 4 entries in queue
  - **Write Back**: *writes occur at the speed of the cache*. And multiple writes within a block require only one write to the lower-level memory

Q4: Write Strategy

- **Two option** when unsuccessfully writing to cache (**Write Miss**)
  - **Write allocate**: the block is allocated on a write miss, followed by the write hit actions
    - Usually means that you have to do a Read Miss to *fill in rest of the cache block*
  - **No-write allocate**: write misses do not affect the cache. *The block is modified only in the lower-level memory*
- To manage a write miss, both options can be used for both write policies (Write Back and Write- through), but usually:
  - **Write-Back** cache uses the **Write Allocate** option
  - **Write Through** cache uses the **No Write Allocate** option

Hit and Miss: Read vs Write

- **Read Hit**
  - Read data in the cache
- **Read Miss**
  - CPU stalls
  - Data request to memory, copy in cache (write in cache), repeat of cache read operation
- **Read Miss**
  - Write data both in cache and in memory (**Write through**)
  - Write data in cache only (**Write back**): memory copy only when it is replaced due to a miss
- **Write Miss**
  - CPU stalls
  - Data request to memory, copy in cache (write in cache), repeat of cache write operation (**Write allocate**)
  - Simple send write data to lower level memory (**No write allocate**)

Case study

- A Direct mapping cache which has 8 blocks. Each block contains 16 words. Each word has 1 byte data
- A main memory (RAM) has 256 blocks
- Write strategy using is Write back
  - If Cache Miss, using Write Allocate
- After the first initialization, cache contains 8 blocks coming from the RAM. They are (in order): 8, 17, 23, 34, 38, 67, 69, 132
- Questions:
  - Draw the structure of address using for main memory accessing and cache accessing
  - Draw the structure of the cache after the first initialization
  - Update the cache in several case respectively below:
    - Read data: 43FH
    - Read data: 82AH
    - Read data: 915H
    - Write data: 08CH
    - Write data: B4AH
    - Write data: 45DH

Question?