



# Computer Systems and Networks

ECPE 170 – Jeff Shafer – University of the Pacific

\$\$\$

\$\$\$

## Cache Memory

\$\$\$

# Schedule

## ➤ **This week**

- Chapter 6 – Memory systems

## ➤ **Next Tuesday**

- **Exam 2** – Tuesday, Nov 1<sup>st</sup>

- **Chapter 4**

- MARIE, etc...

- **Chapter 5**

- Instruction sets, memory addressing modes, etc...

# Review – Quiz 3

# Review – Quiz 4

# Objectives

- **Starting Chapter 6 today**
- **No longer will we treat memory as a big dumb array of bytes!**
- Hierarchical memory organization
  - How does each level of memory contribute to system performance?
  - How do we measure performance?
- New concepts!
  - Cache memory and virtual memory
  - Memory segmentation and paging
  - Address translation

# Types of Memory

## ➤ **RAM versus ROM?**

- RAM – Random access memory (read & write)
- ROM – Read-only memory

# Types of Memory

## ➤ **DRAM versus SRAM?**

### ➤ **DRAM – Dynamic RAM**

- Cheap and simple!
- Capacitors that slowly leak charge over time
- Refresh every few milliseconds to preserve data

### ➤ **SRAM – Static RAM**

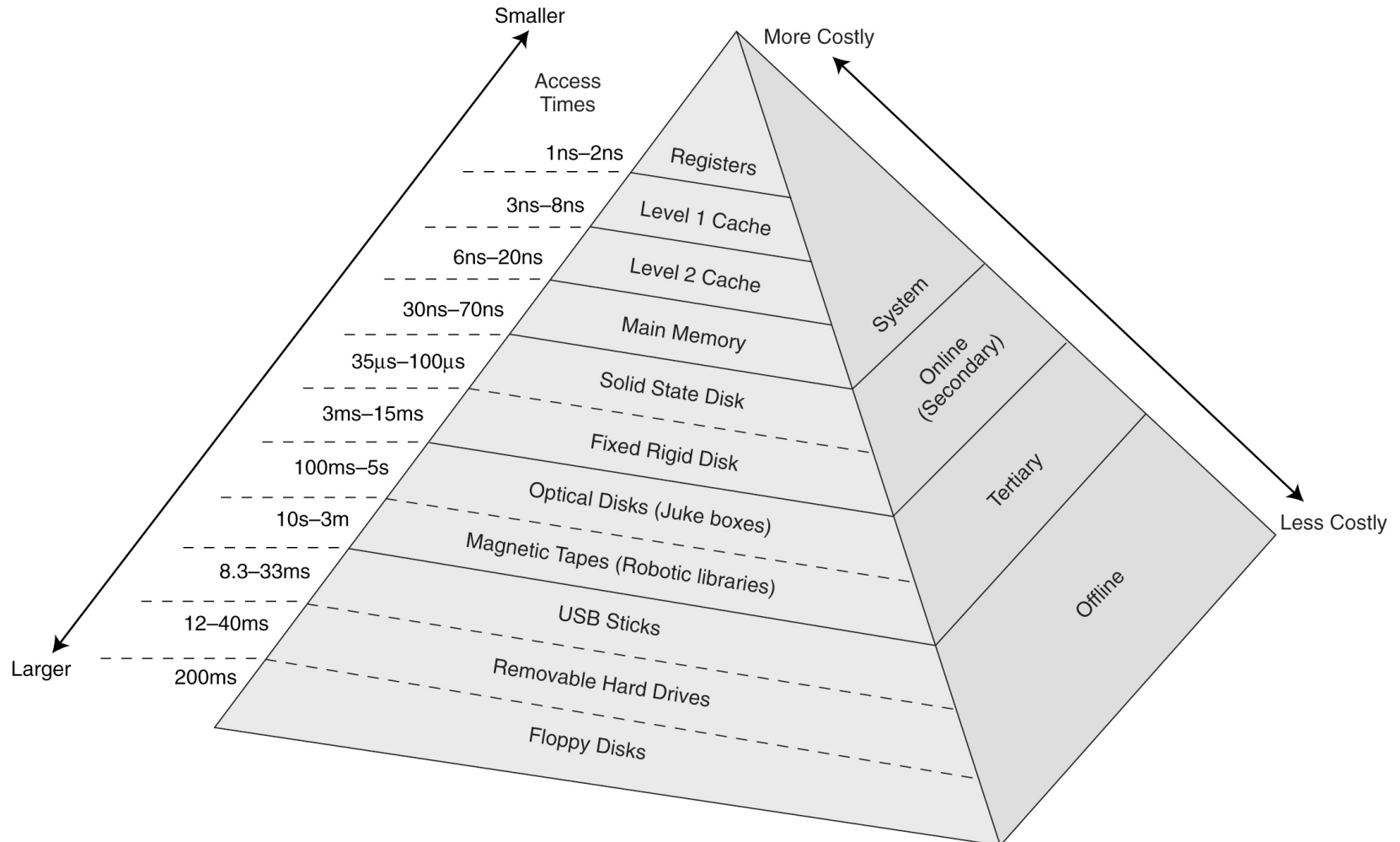
- Similar to D Flip-flops
- No need for refresh
- Fast / expensive (use for cache memory, registers, ...)

# Memory Hierarchy

- Goal as system designers:  
**Fast performance and low cost!**
  - Tradeoff: 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, less fast main memory is accessed through the data bus
  - Largest, slowest, permanent storage (disks, etc...) is even further from the CPU



# The Memory Hierarchy



# Memory Hierarchy

- This chapter just focuses on the part of the memory hierarchy that involves **registers, cache, main memory, and virtual memory**
  
- **What is a register?**
  - Storage locations available on the processor itself
  - **Manually** managed by the assembly programmer or compiler
  
- **What is main memory? RAM**
  
- **What is virtual memory?**
  - Extends the address space from RAM to the hard drive
  - Provides more space

# Cache Memory

- **What is a cache?**
  - Speed up memory accesses by storing recently used data closer to the CPU
  - **Closer** than main memory – on the CPU itself!
- Although cache is much smaller than main memory, its access time is a fraction of that of main memory
- Cache is **automatically** managed by the memory system

# Memory Hierarchy

- CPU wishes to access data for an instruction
  - Does the instruction say it is in a register or memory?
    - If register, go get it!
  - If in memory, send request to nearest memory (the cache)
  - If not in cache, send request to main memory
  - If not in main memory, send request to virtual memory (the disk)
  
- Once the data is located and delivered to the CPU, it will also be saved into cache memory for future access

# (Cache) Hits versus Misses

- **Hit** – When data is found at a given memory level.
  - **Miss** – When data is not found at a given level
  
- **Hit rate** – Percentage of time data is found at a given memory level.
  - **Miss rate** – Percentage of time data is not found
  - Miss rate =  $1 - \text{hit rate}$
  
- **Hit time** – Time required to access data at a given memory level
  
- **Miss penalty** – Time required to process a miss
  - Time that it takes to replace a block of memory, plus
  - Time it takes to deliver the data to the processor

# Cache Locality

- When data is loaded into a cache, we save more than just the specific byte(s) requested
  - Often, save neighboring 64 bytes or more!
- **Principle of locality** – 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 in memory
  - **Sequential locality** - Instructions tend to be accessed sequentially (*just a variant of Spatial locality*)

# Cache Design



# Cache Memory

- First, divide main memory and cache memory into **blocks**
  - Cache block size = main memory block size
  - Example: Core i7: **64 bytes** per cache block
- If data is loaded into the cache, we load in the entire block, even if we only needed a byte of it
  - Allows us to take advantage of locality
- Main memory is much larger than the cache
  - **Thus, many blocks of main memory must map to a single block of cache**

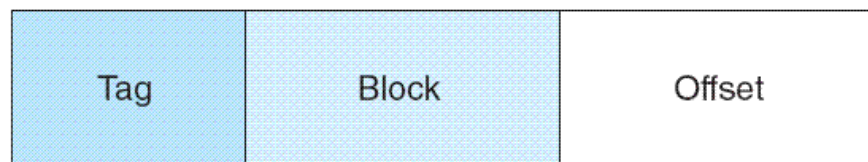


# Cache Memory

- Main memory is usually accessed by address
  - i.e. “Give me the byte stored at address 0x2E3”
- If the data is copied to the cache, it cannot keep the same address
  - Remember, the cache is **much smaller** than main memory!
- We need a scheme to translate between a main memory address and a cache location
  - *Engineers have devised several schemes...*
  - Direct map, fully associative map, set-associative map, ...

# Cache Memory

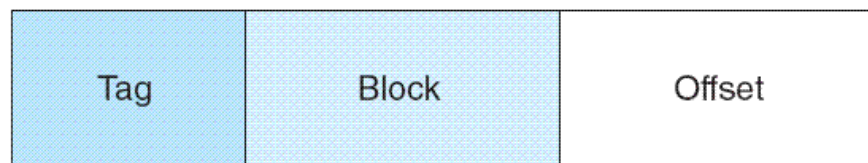
- Cache memory is typically accessed by content
  - Often called *content addressable memory*
  - This content is not *data*. Rather, it is (part of) the original **address of the data** in main memory!
- The original main memory address is divided into **fields**, each with special meaning



← Bits in Main Memory Address →

# Cache Memory

- **Tag field** – Distinguishes between multiple main memory blocks that could map to the same cache block
- **Block field** – Which block # in the cache is this?
- **Offset field** – Points to the desired data within the block



# Direct Mapped Cache



# Direct Mapped Cache

- Simplest cache mapping scheme.
- If the cache stores  $N$  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 could hold block 7 or 17 or 27 or 37 or ... 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 would happen if there was no valid bit?**

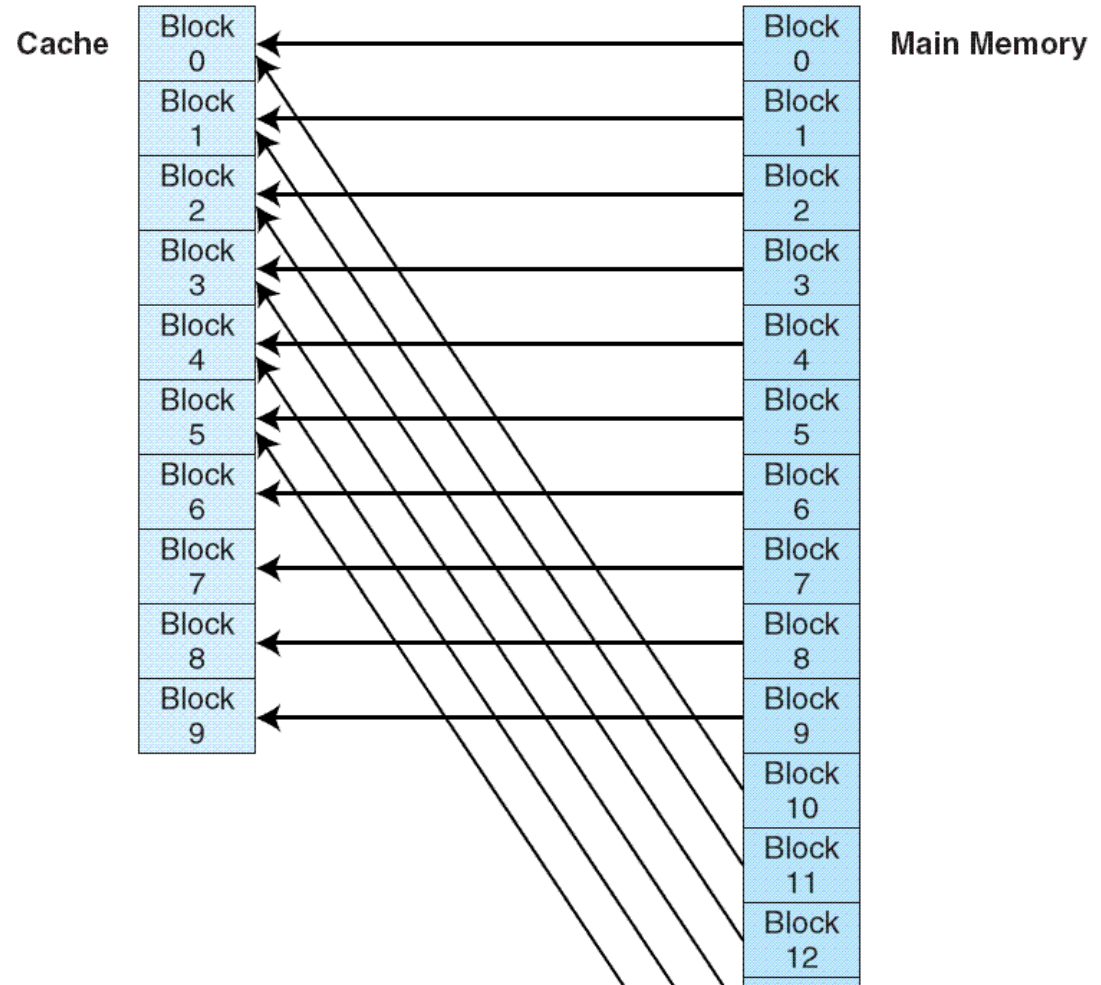
# Direct Mapped Cache

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

- Example of cache contents
  - Block 0 (tag 00000000)
    - Contains multiple words from main memory
  - Block 1 (tag 11110101)
    - Contains multiple words from memory
  - Blocks 2 and 3 are not valid (yet)

# Direct Mapped Cache

- ➔ Direct mapped cache that stores N blocks
- ➔ Block X of main memory maps to cache block  
 $Y = X \text{ mod } N$
- ➔ **But only one block can actually be mapped to a cache location at a time!**



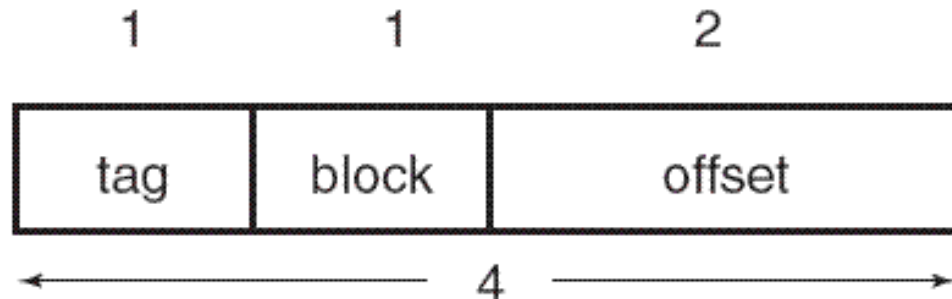
# Example 1 – Direct Mapped Cache

- Example 1
  - Main memory – stores 4 blocks
    - Word addressable
  - Cache memory – stores 2 blocks
  - Block size = 4 words (don't care how big a word is)
- Mapping?
  - Block 0 and 2 of main memory map to Block 0 of cache
  - Blocks 1 and 3 of main memory map to Block 1 of cache
- Let's look at tag, block, and offset fields to see this mapping...



# Example 1 – Direct Mapped Cache

- **Determine the address format for mapping**
  - Each block is 4 words
    - Thus, the offset field must contain 2 bits (so we can select any word inside the block)
  - There are 2 blocks in the cache
    - Thus, the block field must contain 1 bit (so we can select each possible block)
  - This leaves 1 bit for the tag (main memory address has 4 bits because there are a total of  $2^4=16$  words)



# Example 1 – Direct Mapped Cache

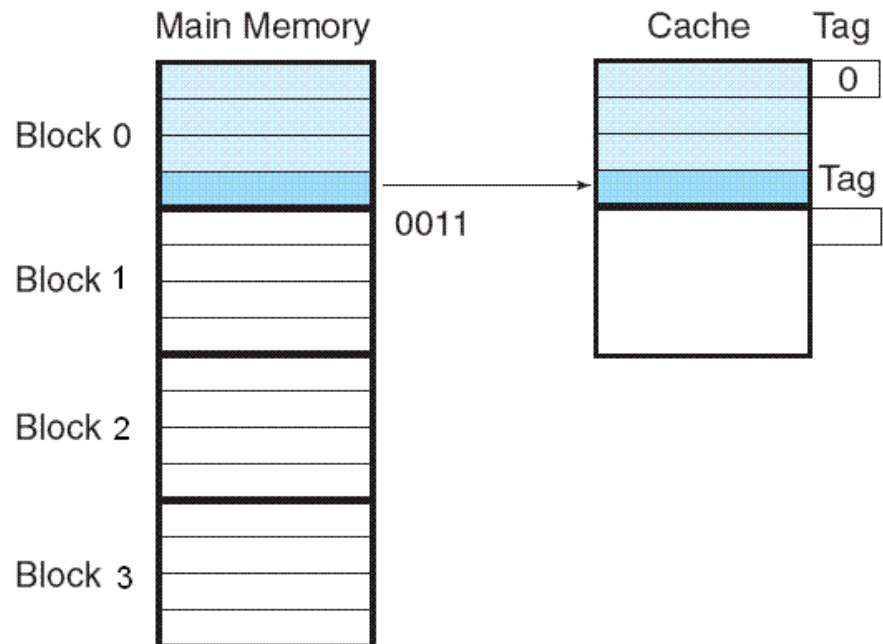
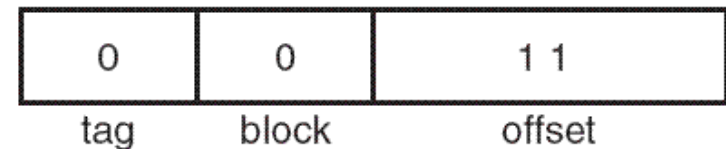
➤ Suppose we need to access main memory address  $3_{16}$  (0011 in binary)

➤ Partition address

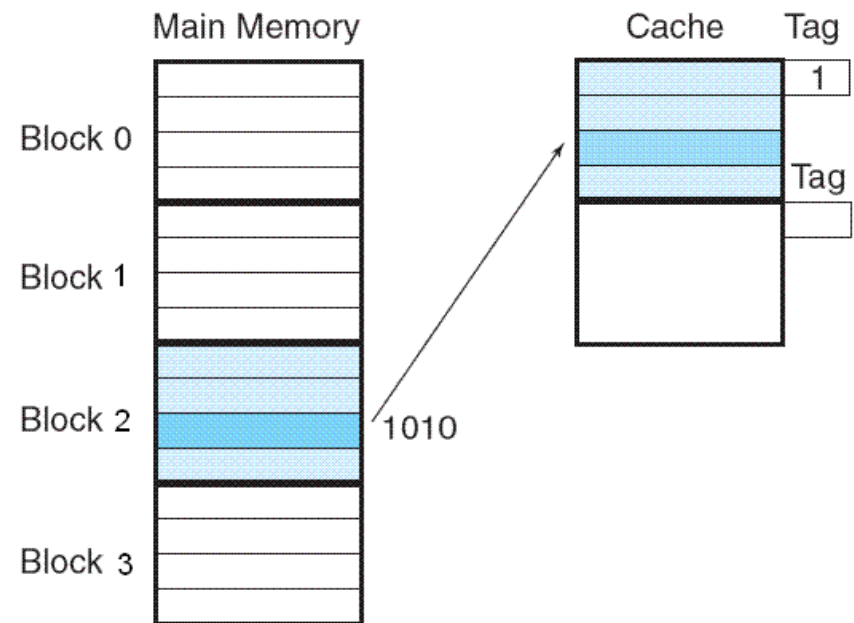
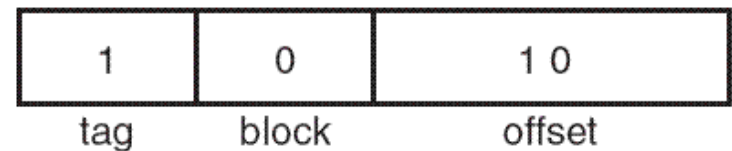
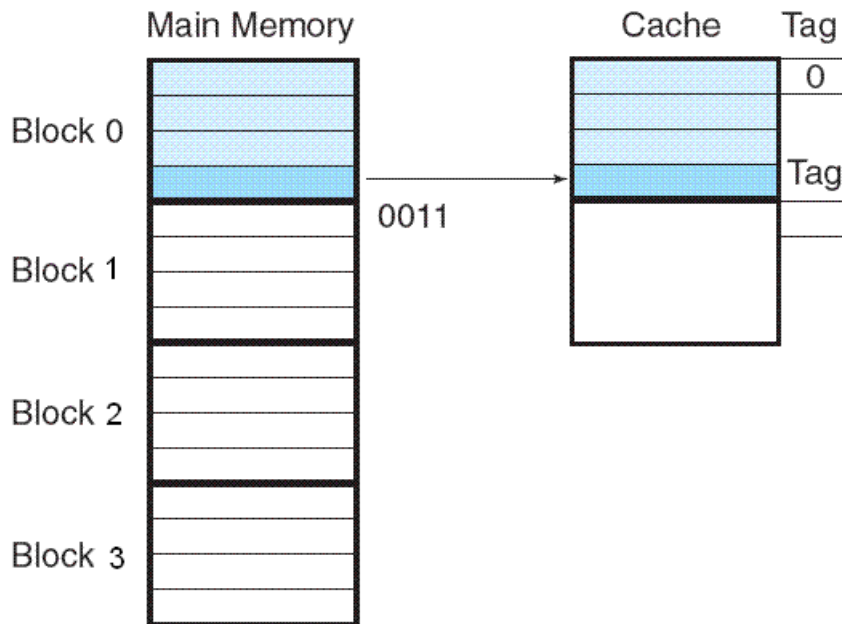
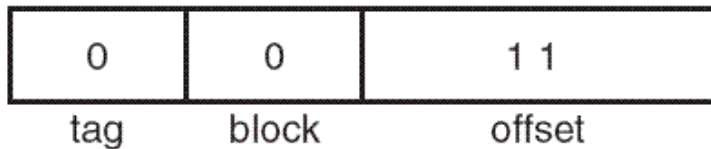
➤ Thus, this main memory address maps to cache block 0

➤ Mapping shown (along with the tag that is also stored with the data)

**The next slide illustrates another mapping.**



# Example 1 – Direct Mapped Cache



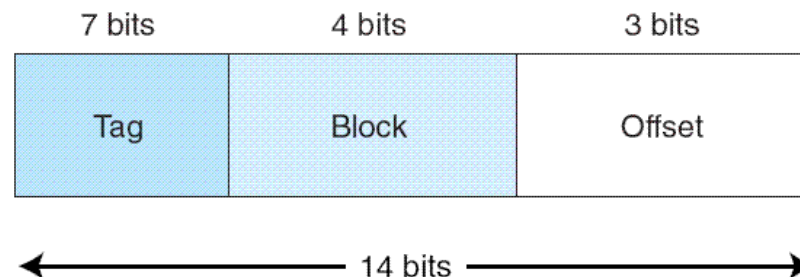
# Example 2 – Direct Mapped Cache

## ➤ **Example Configuration**

- Main memory stores  $2^{14}$  bytes (byte-addressable)
  - Cache memory with 16 blocks
  - Block size = 8 bytes
- **Determine the address format for mapping**

# Example 2 – Direct Mapped Cache

- **Determine the address format for mapping**
- Each main memory address is 14 bits long
  - Each block is 8 bytes long
    - Offset field is 3 bits wide ( $2^3 = 8$ ) to select inside block
  - There are 16 blocks in the cache to select from
    - Block field is 4 bits wide ( $2^4 = 16$ )
  - All remaining bits (7 bits) make up the tag field.



# Example 3 – Direct Mapped Cache

- **Example** – Main memory addresses are divided into
  - 12 bit tag field
  - 9 bit block field
  - 6 bit offset field
  - **What do we know about the main memory and cache?**

# Example 3 – Direct Mapped Cache

- **What do we know about the main memory and cache?**
  - The total main memory size is  $2^{(12+9+6)} = 2^{27}$  bytes, or 128MB
  - The cache has  $2^9 = 512$  blocks
  - Each block contains  $2^6 = 64$  bytes
  - The total cache size is  $2^{(9+6)} = 2^{15} = 32\text{kB}$
  - Main memory contains  $2^{(12+9)} = 2^{21} = 2097152$  blocks

# Direct Mapped Cache 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 in the cache
    - Which determines the size of the block field
  - How many addresses (bytes or words) are in a block
    - Which determines the size of the offset field

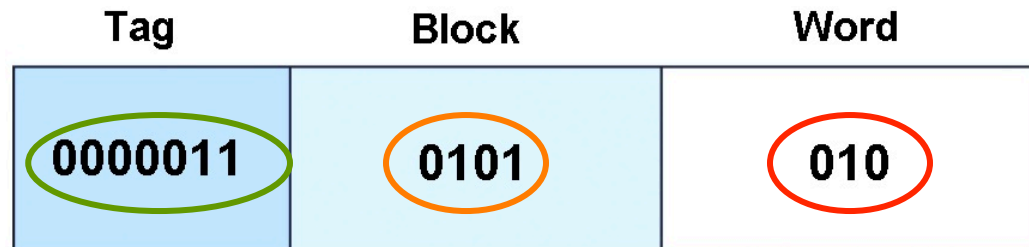


# Cache Thrashing

➤ Back to Example 2, assume a program generates the address **0x1AA**

➤ In 14-bit binary, this number is: **00000110101010**

➤ 7 bit tag, 4 bit block, and 3 bit offset fields

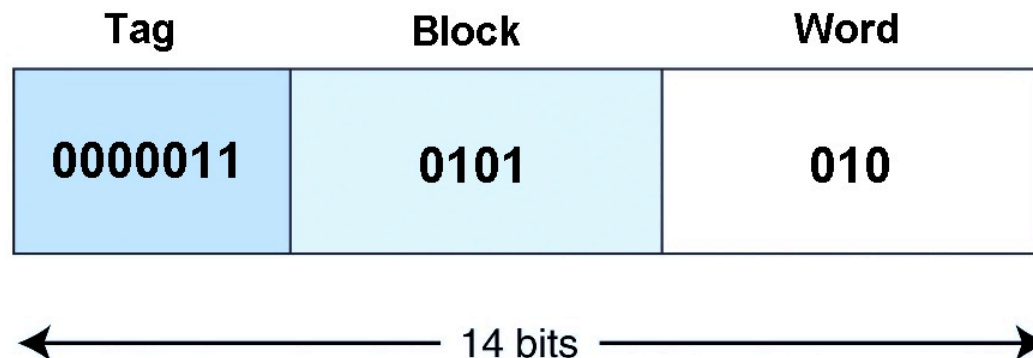


← 14 bits →

➤ Words 1A8 through 1AF are loaded into the block

# Cache Thrashing

- Another way to view what happened:
  - Blocks in main memory are contiguous addresses
    - When we load a **block**, we start with the byte in the block whose offset (word) field contains all 0's
    - The offset (word) field of the last byte contains all 1's
    - **Entire block is loaded into cache**
  - 0000011 0101 **000** = 1A8
  - 0000011 0101 **111** = 1AF



# Cache Thrashing

➤ **What if the program later reads from the address 0x1AB?**

➤ **Cache hit!**

➤ Data found in block 0101 (with matching tag), word 011

Tag	Block	Word
0000011	0101	011

➤ **What if the program reads from the address 0x3AB?**

➤  $0x3AB = 0000111\ 0101\ 011$  – A new tag number!

➤ **Cache miss!**

➤ Block 0101 (tag 0000011) is evicted (removed) from cache

➤ Block 0101 (tag 0000111) is added to the cache

# Cache Thrashing

- Suppose a program generates a series of memory references such as: 0x1AB, 0x3AB, 0x1AB, 0x3AB, ...
  - The cache will continually evict and replace blocks
    - This is called “**thrashing**”
  - The theoretical advantage offered by the cache is lost in this extreme case
  
- Main disadvantage of direct mapped cache
  - Each main memory block can only go one place in the cache
  
- Other (more sophisticated) cache mapping schemes prevent this kind of thrashing
  - **Topic for next class!**

# Exercise – Direct Mapped Cache

- Exercise: Suppose you have a main memory with 128Kbytes and a direct-mapped cache made up of 256 32-byte blocks
  - **What are the sizes of the tag, block and offset fields?**
  - **How many block of main memory does the system have?**
  - **What is the total size of the cache in bytes?**
  - **How many memory blocks map to each cache block?**

# Exercise – Direct Mapped Cache

- Address layout
  - Tag: 4 bits (main memory addresses are 17 bits, and 13 are used by block/offset, leaving 4 bits remaining)
  - Block: 8 bits (256 blocks in the cache,  $2^8 = 256$ )
  - Offset: 5 bits (32 bytes per cache block, thus: 5 bits specify the correct byte)
  
- Blocks in main memory: 4096
  - $2^{17}$  bytes of main memory,  $2^5$  bytes per block, thus  $2^{17} / 2^5 = 2^{12}$  blocks in main memory.
  
- Total size of the cache: 8192 bytes
  - $256 * 32 = 2^8 * 2^5 = 2^{13}$
  
- Main memory blocks mapped to each cache block: 16
  - 4096 blocks in main memory / 256 blocks in the cache