Computer Systems and Networks

ECPE 170 – Jeff Shafer – University of the Pacific

$$$

$$$

Cache Memory

$$$

Schedule

- **Today**
  - Memory systems - Caches
  - Exam 2 review
- **Next Monday – Exam 2**
- **Next Wednesday / Friday**
  - Memory systems - Virtual memory

Recap - Cache

- **Which is bigger – a cache or main memory?**
  - Main memory
- **Which is faster to access – the cache or main memory?**
  - Cache – It is **smaller** (which is faster to search) and **closer** to the processor (signals take less time to propagate to/from the cache)
- **Why do we add a cache between the processor and main memory?**
  - Performance – hopefully frequently-accessed data will be in the faster cache (so we don't have to access slower main memory)

Recap – Cache

- **Which is manually controlled – a cache or a register?**
  - Cache is automatically controlled by hardware
  - Registers (typically) are manually controlled by the assembly language program
- **Suppose a program wishes to read from a particular memory address. Which is searched first – the cache or main memory?**
  - Search the cache first – otherwise, there's no performance gain

Recap – Cache

- **Suppose there is a cache miss (data not found) during a 1 byte memory read operation. How much data is loaded into the cache?**
  - Trick question – we always load data into the cache **1 block at a time**.
  - Block size varies – 64 bytes on a Core i7 processor

Recap – Direct Mapped Cache Search

1. Take the main memory address of desired data
   1. Split into **tag**, **block**, and **offset** fields (varies by cache and block size)
2. Go to the indicated block in the cache
3. Does the tag saved in the cache match the search tag?
   Is the block marked as valid?
   1. Yes on both – we have a **cache hit**! ☺
      1. Retrieve the data (go to the byte/word indicated by offset)
   2. Otherwise, we have a **cache miss**! ☹
      1. Need to go to main memory and get the data
      2. Load in the full block from main memory into the cache

Exercise – Direct Mapped Cache

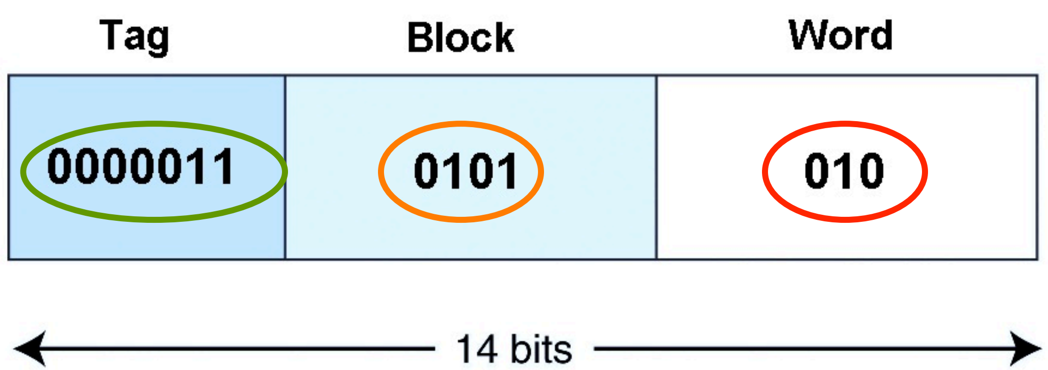
- Exercise: Suppose you have a main memory that stores 128Kbytes and a direct-mapped cache that stores 256 32-byte blocks
  - **What are the sizes of the tag, block and offset fields?**
  - **How many block of main memory does the system have?**
  - **What is the total size of the cache in bytes?**
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

Cache Thrashing

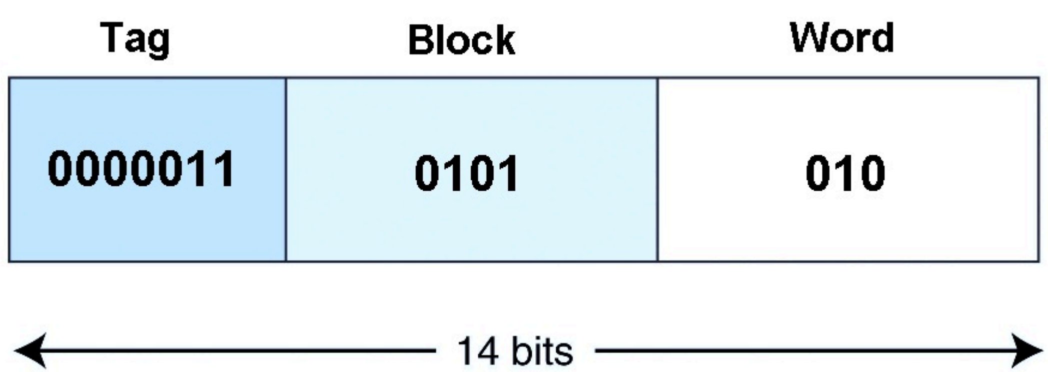
- Back to Example 2 (from last class), assume a program generates the address **`0x1AA`**
  - In 14-bit binary, this number is: 00000110101010
  - 7 bit tag, 4 bit block, and 3 bit offset fields



- 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
- More sophisticated cache mapping schemes can prevent this extreme kind of thrashing

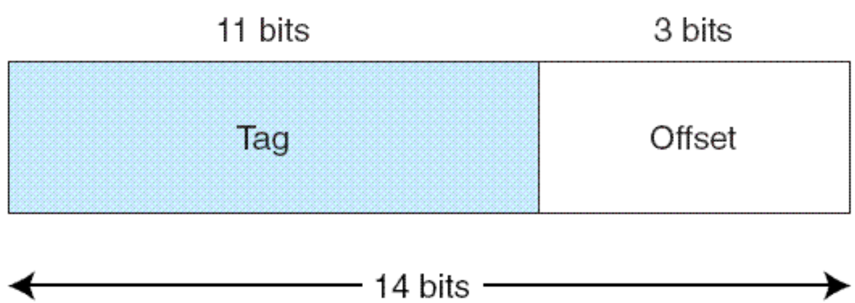
Fully Associative Cache

Fully Associative Cache

- Idea: instead of placing memory blocks in specific cache locations (based on memory address), allow a block to **go anywhere in the cache**
  - The cache would have to completely fill up before any blocks are evicted
- New design: **fully associative cache**
- Memory address is partitioned into only two fields
  - *Tag* and *Offset*

Fully Associative Cache

- Example for 14-bit memory addresses
  - Cache size: 16 blocks
  - Block size: 8 ($2^3 = 8$, thus 3 bits for offset)



- How to retrieve?
  - Search all tags in parallel!
  - This requires **special**, **costly hardware** (i.e. a **CAM**)

Fully Associative Cache

- The block that is evicted from a cache is the **victim block**
- Direct-Mapped cache
  - The victim is always the cache block with the matching block number
- Fully-Associated cache
  - No fixed mapping
  - How does hardware pick a victim?
- There are a number of ways to pick a victim
  - Example: Evict the "least recently used" block
  - Will discuss more next week…

Set Associative Cache

Set Associative Cache

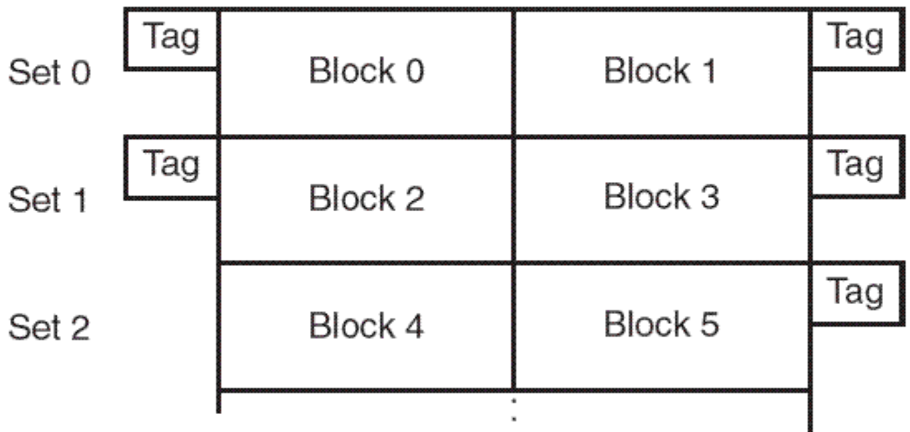
- **Hybrid** between direct mapped cache and fully associative cache
- Reduces hardware complexity and improves performance

Set Associative Cache

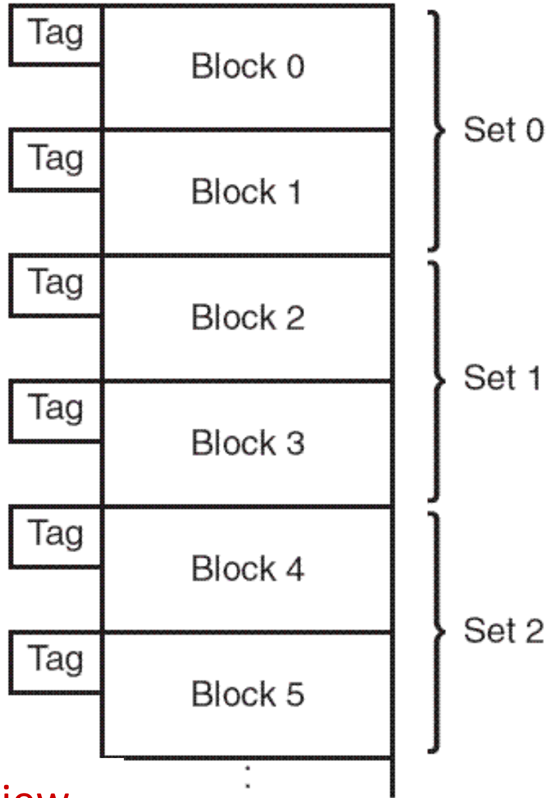
- ***N*-way set associative cache**
(where N is a number, i.e. 2)
  - Instead of mapping anywhere in the cache, a memory reference can map only to the *subset* of cache slots
  - Similar to direct mapped cache
    - Memory reference maps to a limited number of locations
  - Similar to fully associated cache
    - Memory reference maps to more than one potential location (so we need to search in parallel)

Set Associative Cache

- The number of cache blocks per set can vary
- Example: 2-way set associative cache
  - Each set contains two different memory blocks



Logical view



Linear view

Set Associative Cache

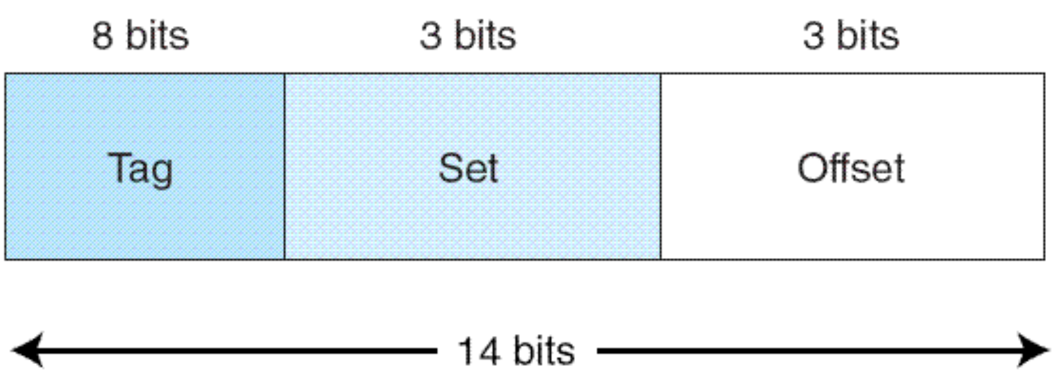
- Memory references are divided into three fields
  - Tag – Uniquely identifies the memory address
  - Set – *New!* Which set does the address map to?
  - Offset – Chooses the word within the cache block

Example 1 – Set Associative Cache

- Memory configuration
  - 2-way set associative cache
  - Word-addressable main memory of 2^{14} words
  - Cache size: 16 blocks
  - Block size: 8 words
- **What do we know about the main memory and cache?**

Example 1 – Set Associative Cache

- **What do we know about the main memory and cache?**
  - Cache has 16 blocks
    - Each set has 2 blocks
    - There are 8 sets in cache
  - Divide up address
    - Set field is 3 bits (2^3 = 8 sets)
    - Offset field is 3 bits (2^3 = 8 words in a block)
    - Tag field is 8 bits (all remaining bits from 14-bit long address)



Example 2 – Set Associative Cache

- Memory configuration
  - 4-way set associative cache
  - 2^{24} words of main memory
  - Cache size: 128 blocks
  - Block size: 8 words
- **How many blocks of main memory are there?**
  - Each block contains 8 (2^3) words
  - 2^{24} words / 2^3 words per block = 2^{21} blocks

Example 2 – Set Associative Cache

- Memory configuration
  - 4-way set associative cache
  - 2^{24} words of main memory
  - Cache size: 128 blocks
  - Block size: 8 words
- **What is the format of a memory address as seen by the cache?**
  - **Offset** field: 3 bits (to specify one of the 8 words in each block)
  - **Set** field: 5 bits (128 total blocks / 4 blocks per set = 32 sets)
  - **Tag** field: 16 bits (remaining bits of 24-bit address)

Example 2 – Set Associative Cache

- Memory configuration
  - 4-way set associative cache
  - 2^{24} words of main memory
  - Cache size: 128 blocks
  - Block size: 8 words
- **To what cache set will address `0x138F29` map?**
  - `0x138F29` = 0001 0011 1000 1111 0010 1001
  - Set field is 00101 = Set 5
    - Any one of the 4 blocks within that set!