Introduction to Virtual Memory

CSCI 237: Computer Organization
24th Lecture, Apr 18, 2025

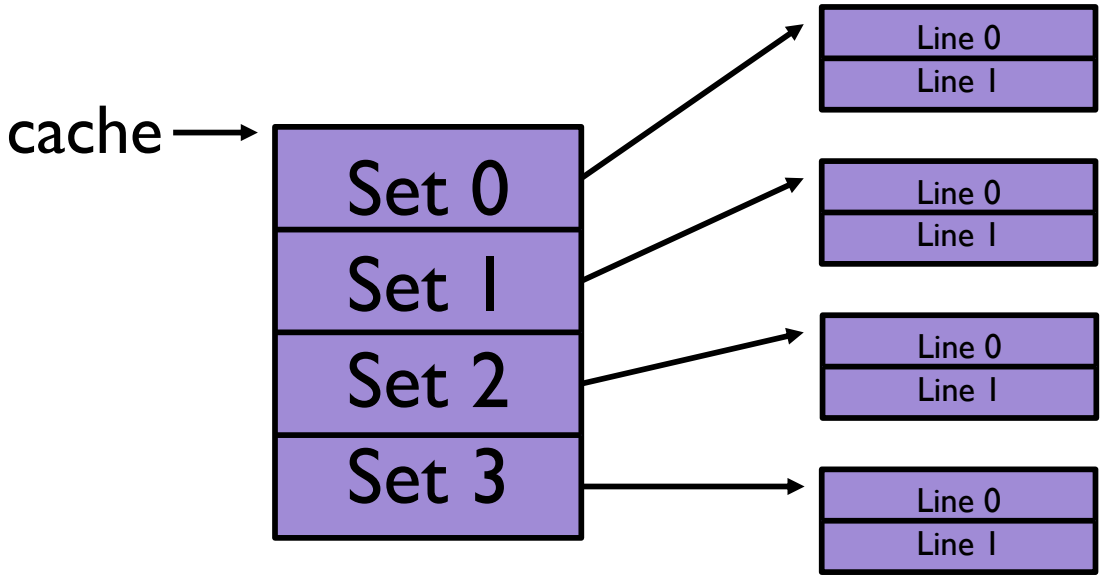
Jeannie Albrecht

Administrative Details

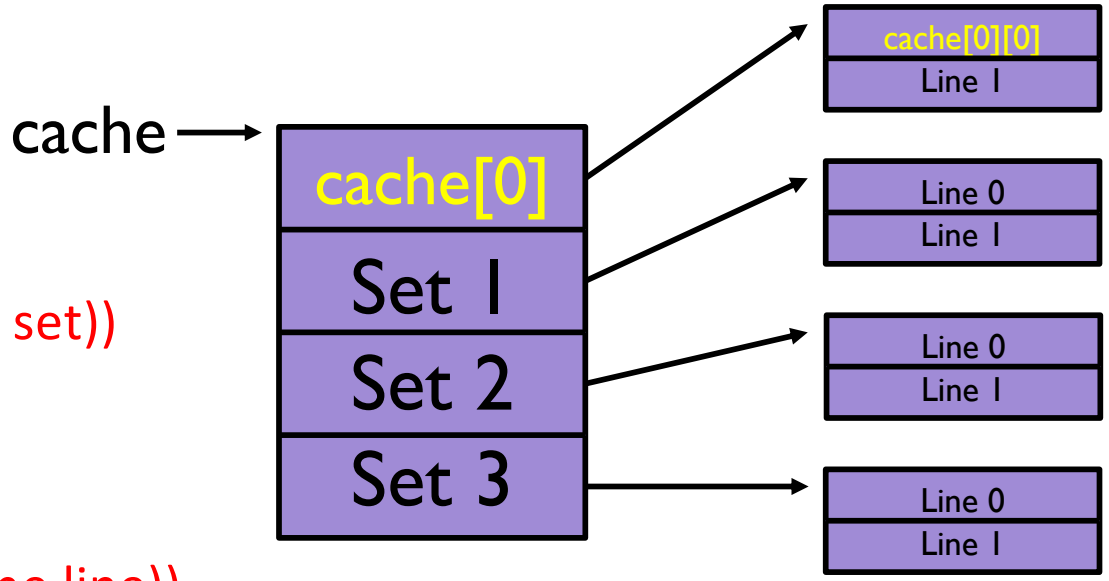
- Lab 5 – Cache simulator
  - Just a simulator, not an actual cache!
  - Keep track of hits and misses *without* moving data around
- Glow HW due today

More Lab 5 Hints

- What is a cache?
  - An array of cache sets
- What is a cache set?
  - An array of cache lines
- What is a cache line?
  - Valid bit, tag, block
  - Note that we are only simulating a cache in Lab 5, so we don't need to represent the actual data blocks
  - Might need a little extra info to implement LRU
  - Probably want a struct to keep track of this!



More Lab 5 Hints



- What is a cache?
  - An array of cache sets
  - cache = malloc(S * sizeof(cache set))
- What is a cache set?
  - An array of cache lines
  - cache[i] = malloc(E * sizeof(cache line))
- What is a cache line?
  - Valid bit, tag, block
  - Note that we are only simulating a cache in Lab 5, so we don't need to represent the actual data blocks
  - Might need a little extra info to implement LRU
  - Probably want a struct to keep track of this!

Last time

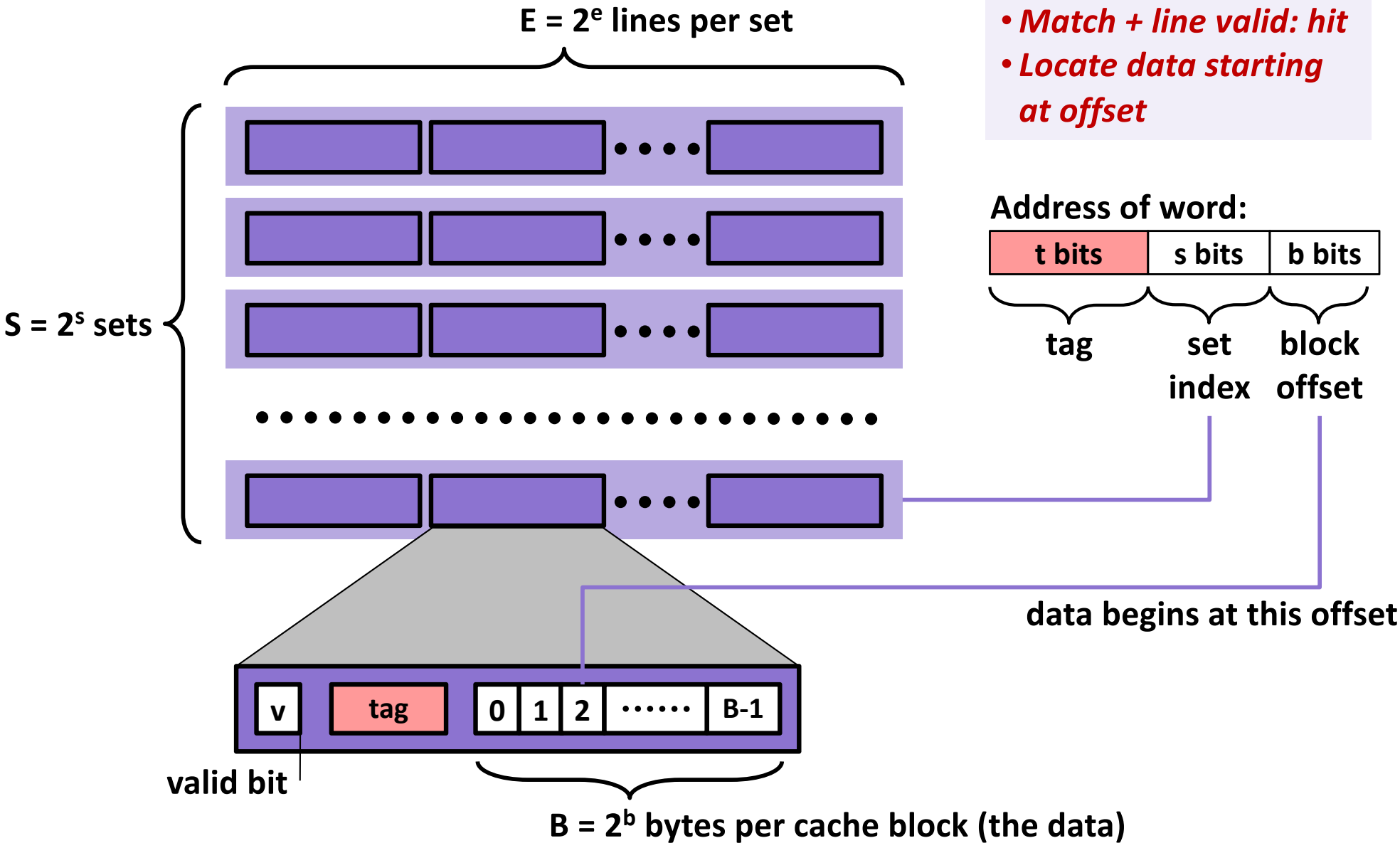
- Cache organization and operation (Ch 6.4)
  - Cache hits and misses

Today – Moving on to Ch 9

- Wrap up discussion about caches (Ch 6.5-6.7)
  - Write-through + No-write-allocate
  - Write-back + Write-allocate (most inline with current trends)
  - Cache performance
- Intro to address spaces (Ch 9.2)
- VM as a tool for caching (Ch 9.3)

Recap: Cache Read

- *Locate set*
- *Check if any line in set has matching tag*
- *Match + line valid: hit*
- *Locate data starting at offset*



Why Index Using Middle Bits?

Direct mapped: One line per set

Assume: cache block size 8 bytes

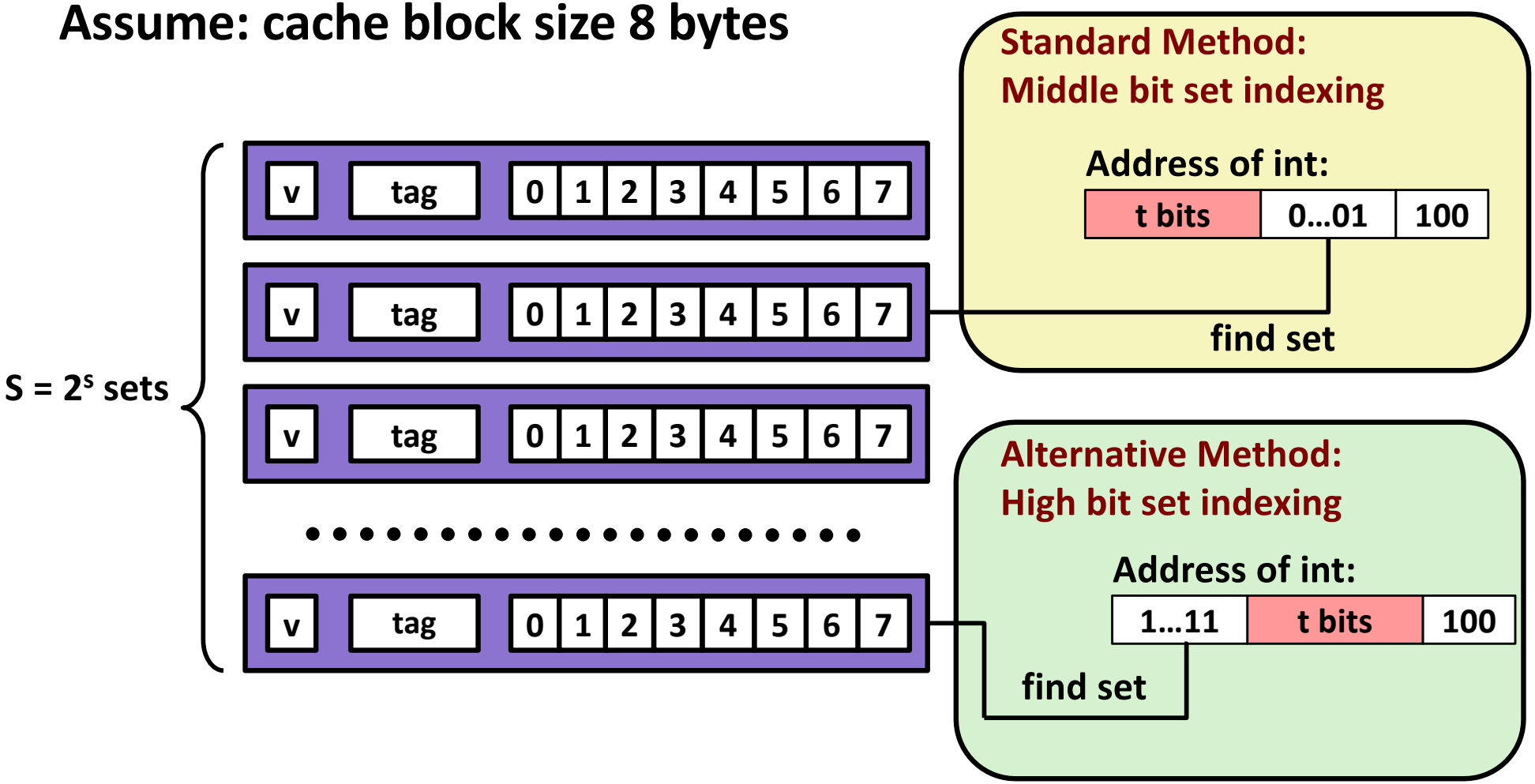
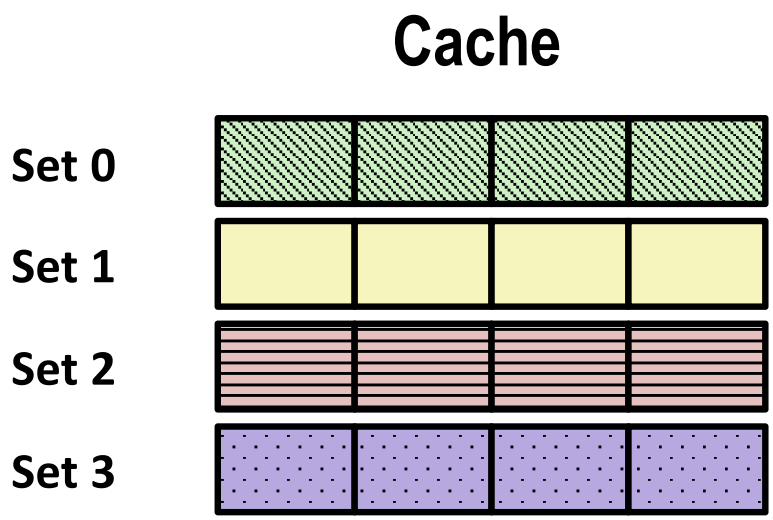
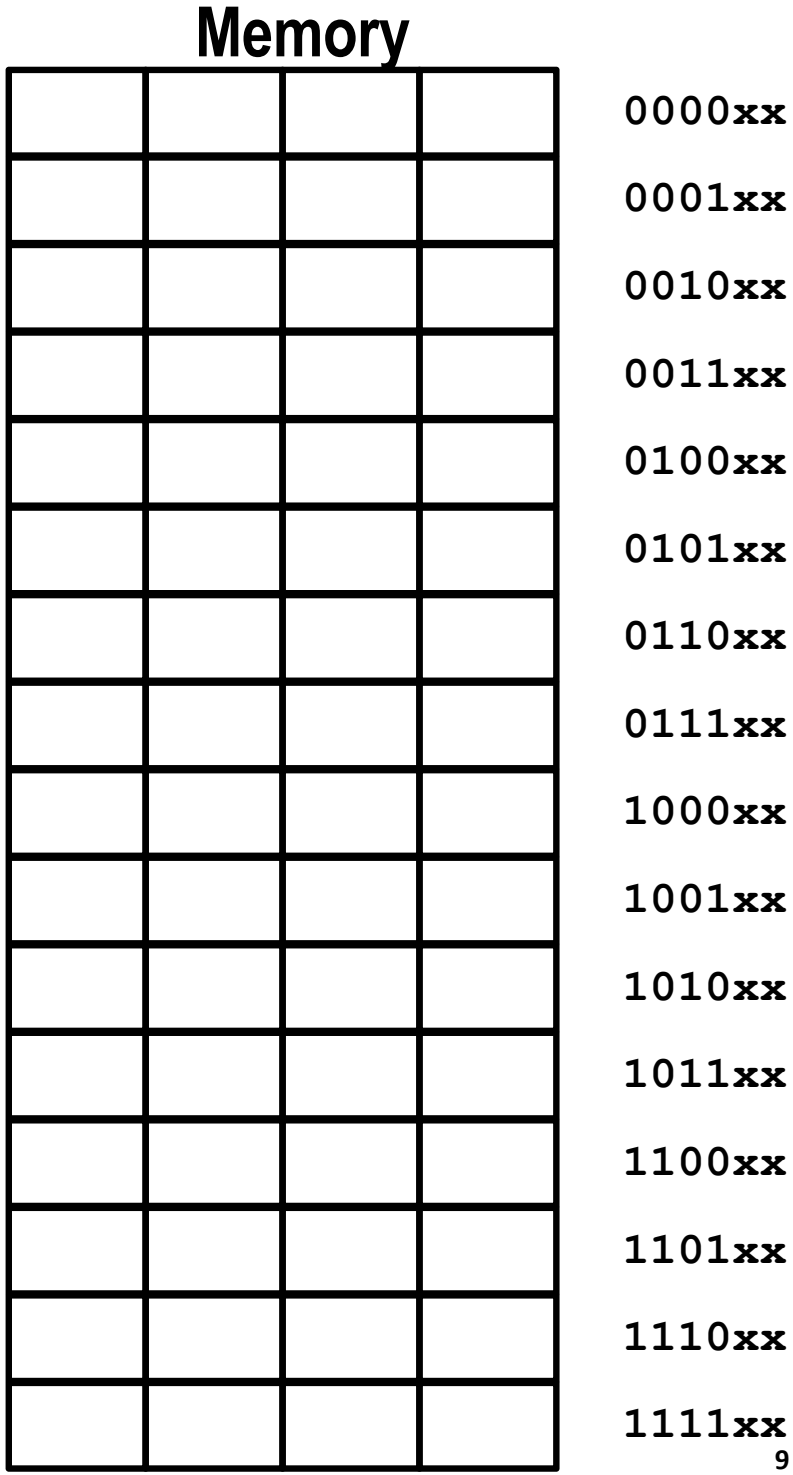


Illustration of Indexing Approaches

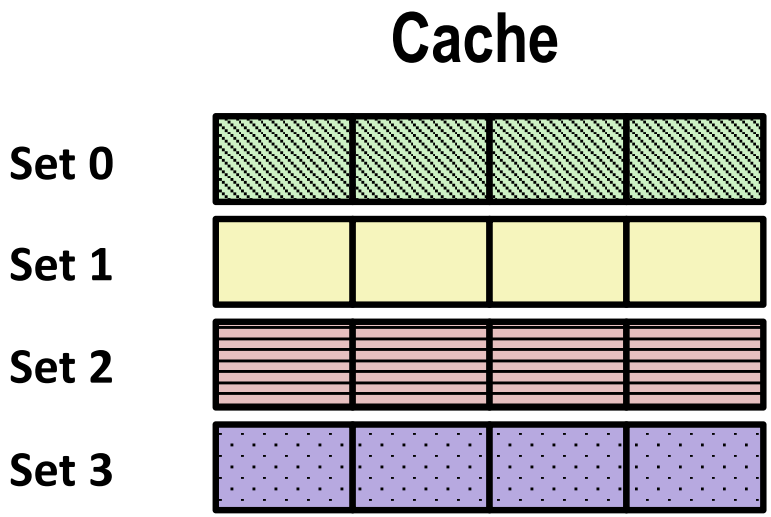
- 64-byte memory
  - 6-bit addresses (2^6=64)
- 16 byte, direct-mapped cache
- Block size = 4 bytes
- 2 bits tag, 2 bits index, 2 bits offset

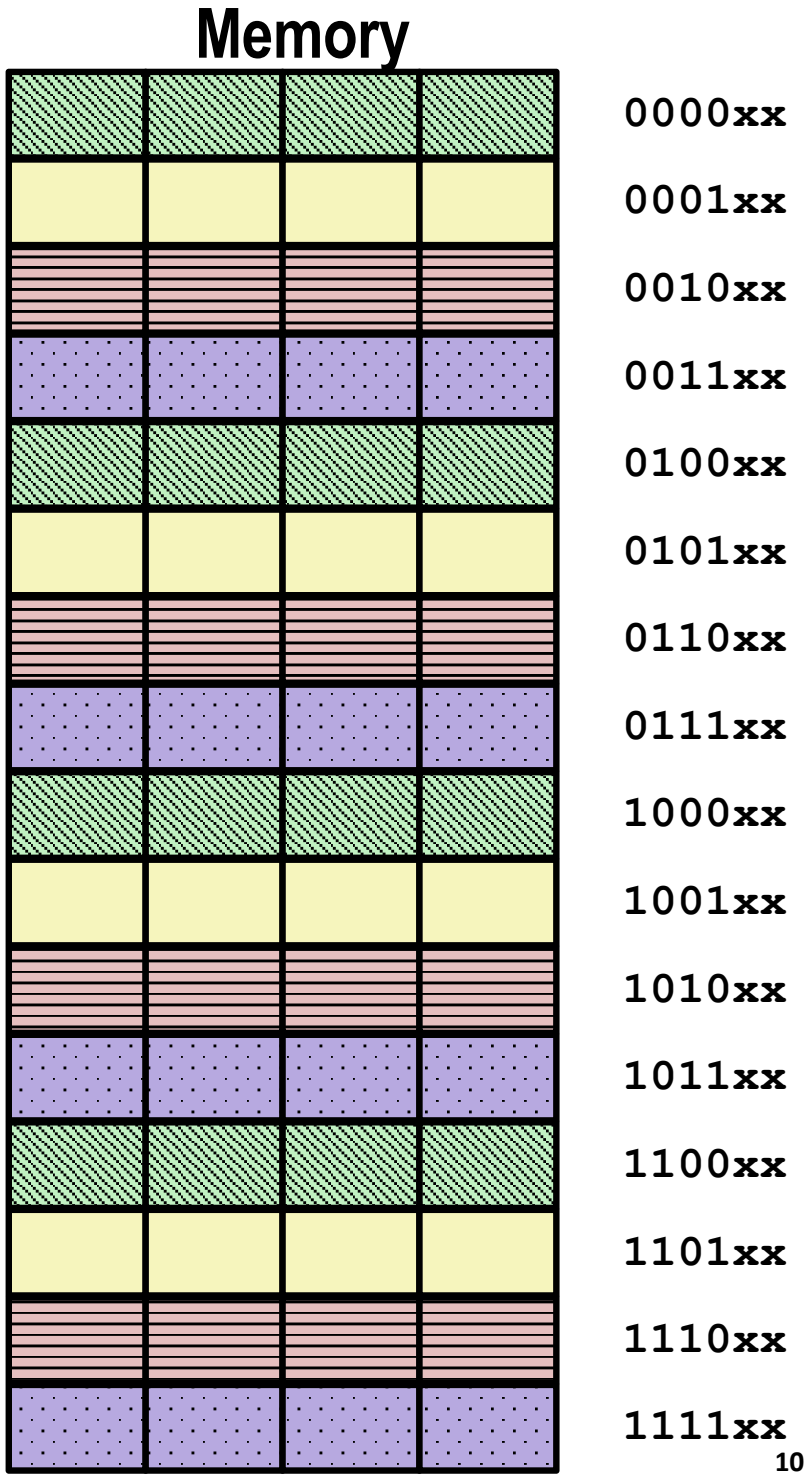




Middle Bit Indexing

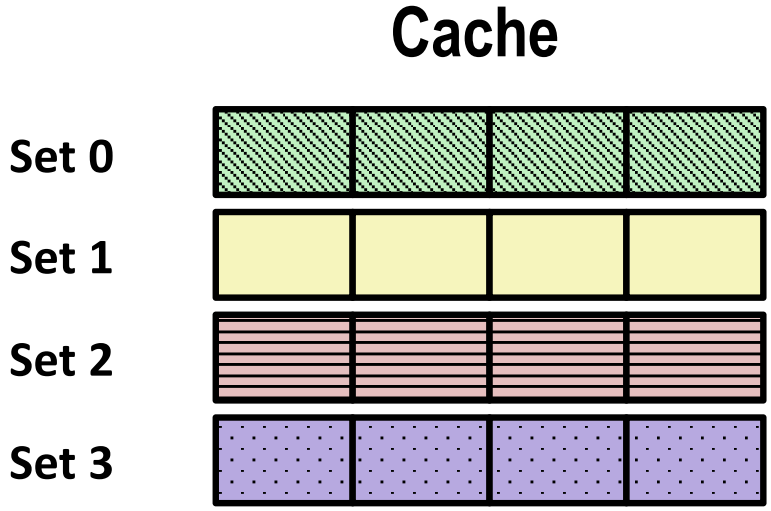
- Addresses of form TTSSBB
  - **TT** Tag bits
  - **SS** Set index bits
  - **BB** Offset bits
- Makes good use of spatial locality

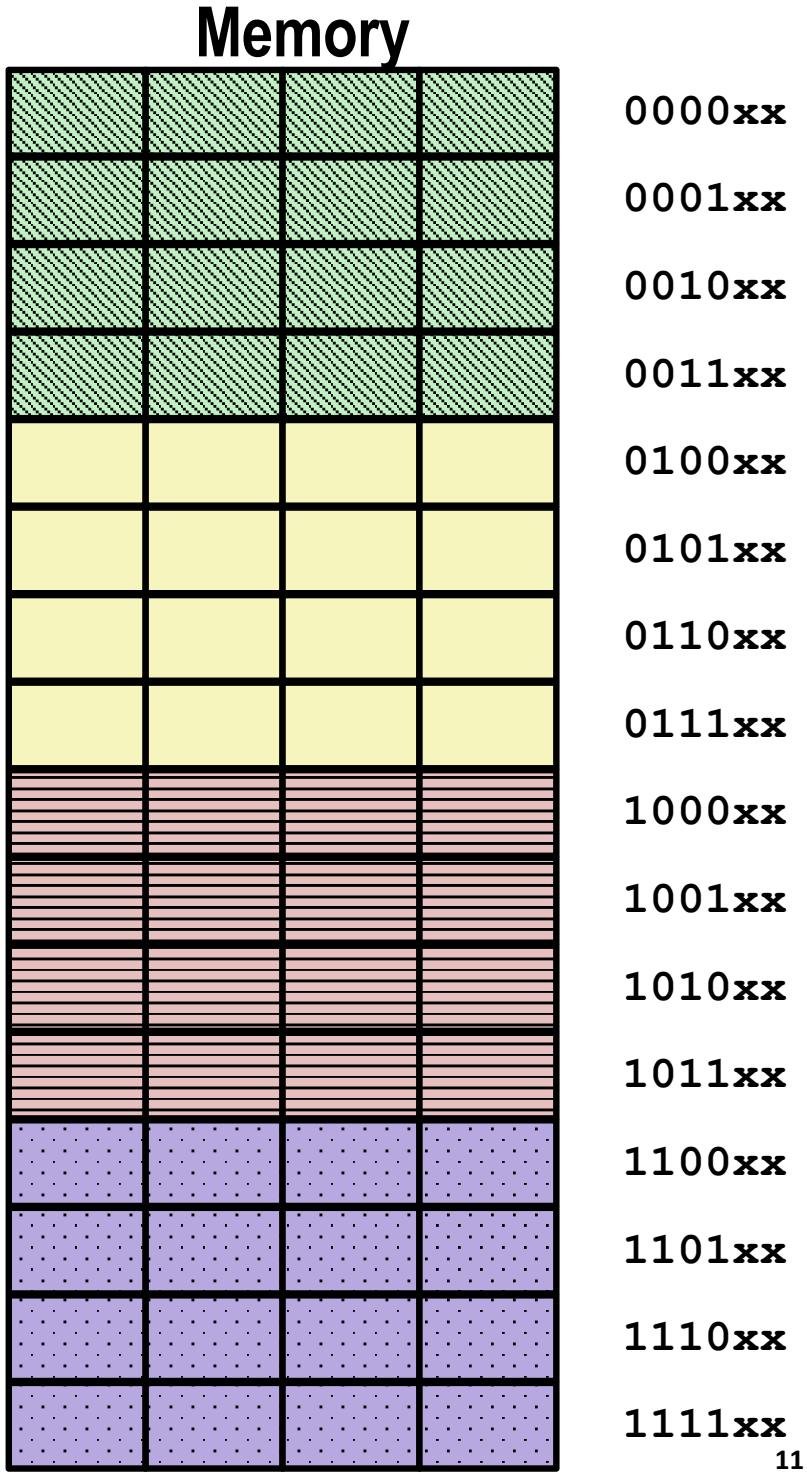




High Bit Indexing

- Addresses of form SSTTBB
  - **SS** Set index bits
  - **TT** Tag bits
  - **BB** Offset bits
- Program with high spatial locality would generate lots of conflicts

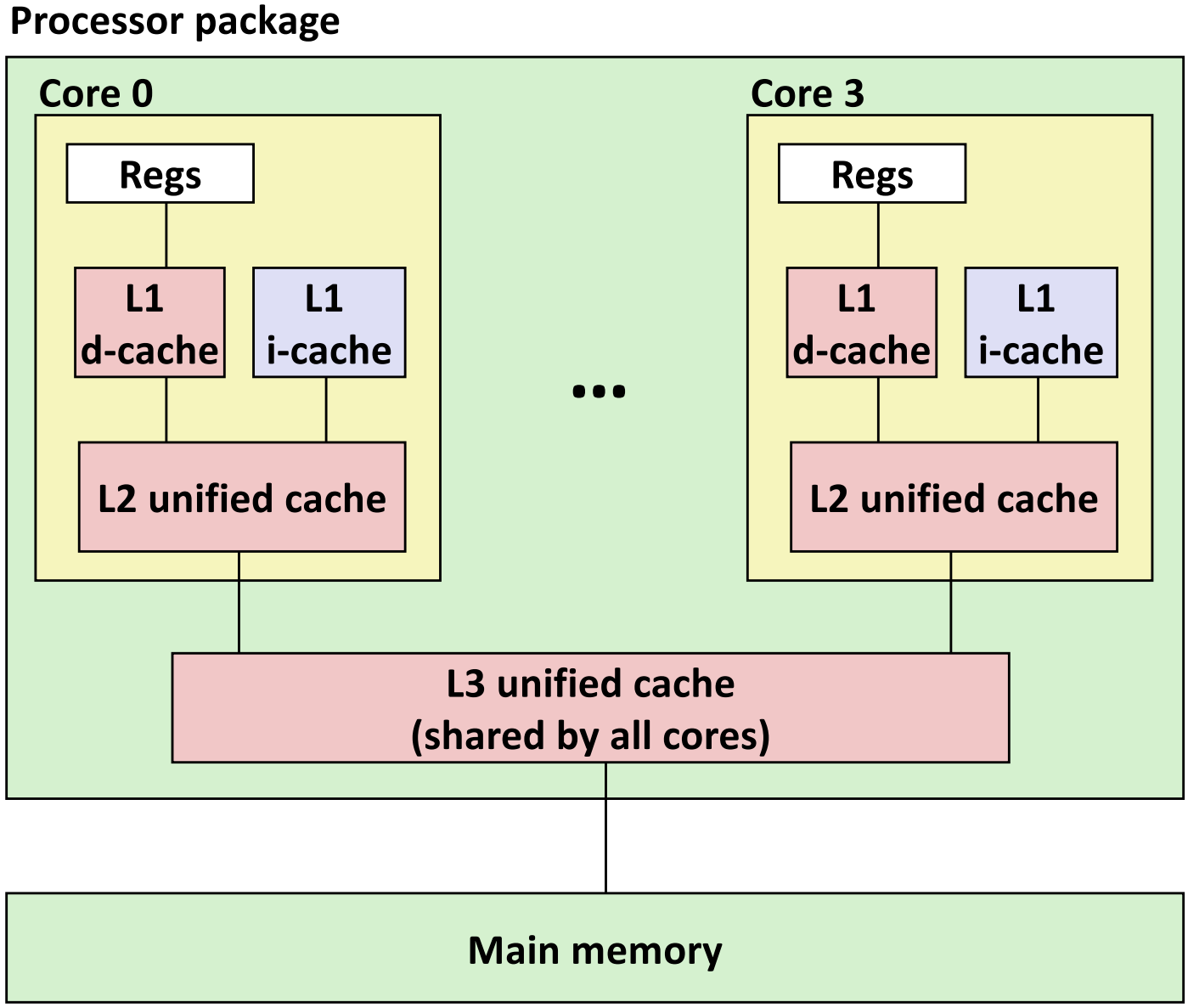




What about writes?

- Multiple copies of data (potentially) exist:
  - L1, L2, L3, Main Memory, Disk
- What to do on a write-hit?
  - Write-through (write immediately to main memory)
  - Write-back (defer write to main memory until replacement of line in cache)
    - Need a "dirty bit" in cache (is the line different from memory or not)
- What to do on a write-miss?
  - Write-allocate (load into cache, update line in cache)
    - Good if more writes to the location follow
  - No-write-allocate (writes straight to memory, does not load into cache)
- Typical
  - Write-through + No-write-allocate
  - **Write-back + Write-allocate (most inline with current trends)**

Real Example: Intel Core i7 Cache Hierarchy



L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 10 cycles

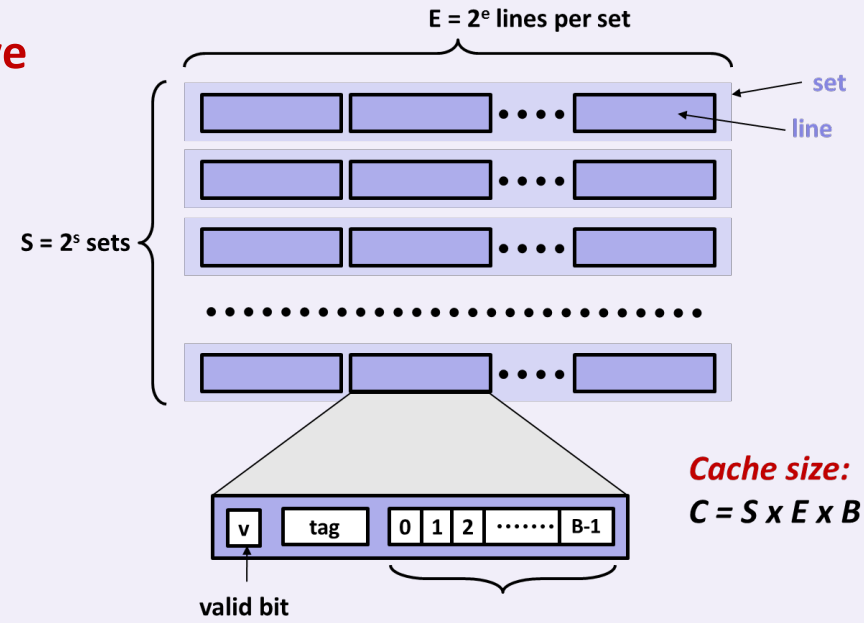
L3 unified cache:
8 MB, 16-way,
Access: 40-75 cycles

Block size: 64 bytes for all caches.

Example: Core i7 L1 Data Cache

32 kB 8-way set associative
64 bytes/block
47 bit address range

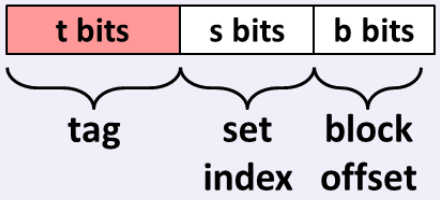
B =
S = , s =
E = , e =
C =



Cache size:
C = S x E x B data bytes

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Address of word:



Block offset: . bits
Set index: . bits
Tag: . bits

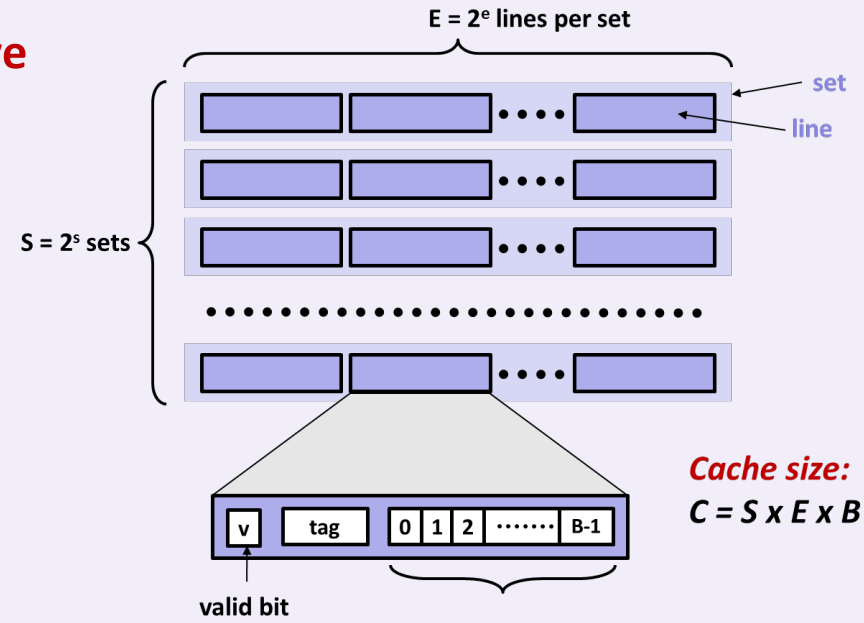
Stack Address:
0x00007f7262a1e010

Block offset: 0x??
Set index: 0x??
Tag: 0x??

Example: Core i7 L1 Data Cache

32 kB 8-way set associative
64 bytes/block
47 bit address range

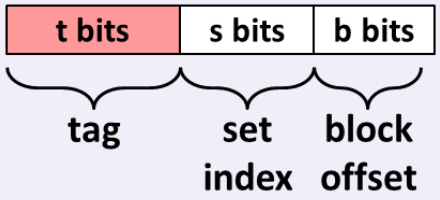
B = 64
S = 64, s = 6
E = 8, e = 3
C = 64 x 64 x 8 = 32,768



Cache size:
C = S x E x B data bytes

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Address of word:



Block offset: 6 bits
Set index: 6 bits
Tag: 35 bits

Stack Address:
0x00007f7262a1e010

0000 0001 0000

Block offset: 0x10
Set index: 0x0
Tag: 0x7f7262a1e

Cache Performance Metrics

- **Miss Rate**
  - Fraction of memory references not found in cache (misses / accesses)
    = 1 – hit rate
  - Typical numbers (in percentages):
    - 3-10% for L1
    - can be quite small (e.g., < 1%) for L2, depending on size, etc.
- **Hit Time**
  - Time to deliver a line in the cache to the processor
    - includes time to determine whether the line is in the cache
  - Typical numbers:
    - 4 clock cycles for L1
    - 10 clock cycles for L2
- **Miss Penalty**
  - Additional time required because of a miss
    - Typically 50-200 cycles for main memory (Trend: increasing!)

Let's think about those numbers

- Huge difference between a hit and a miss
  - Could be 100x, if just L1 and main memory

- Would you believe 99% hits is twice as good as 97%?
  - Consider:
    cache hit time of 1 cycle
    miss penalty of 100 cycles

  - Average access time:

    97% hits: 1 cycle + 0.03 x 100 cycles = **4 cycles**

    99% hits: 1 cycle + 0.01 x 100 cycles = **2 cycles**

- This is why "miss rate" is used instead of "hit rate"

Writing Cache Friendly Code

- Make the common case go fast
  - Focus on the inner loops of the core functions

- Minimize the misses in the inner loops
  - Repeated references to variables are good (temporal locality)
  - Stride-1 reference patterns are good (spatial locality)

Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories

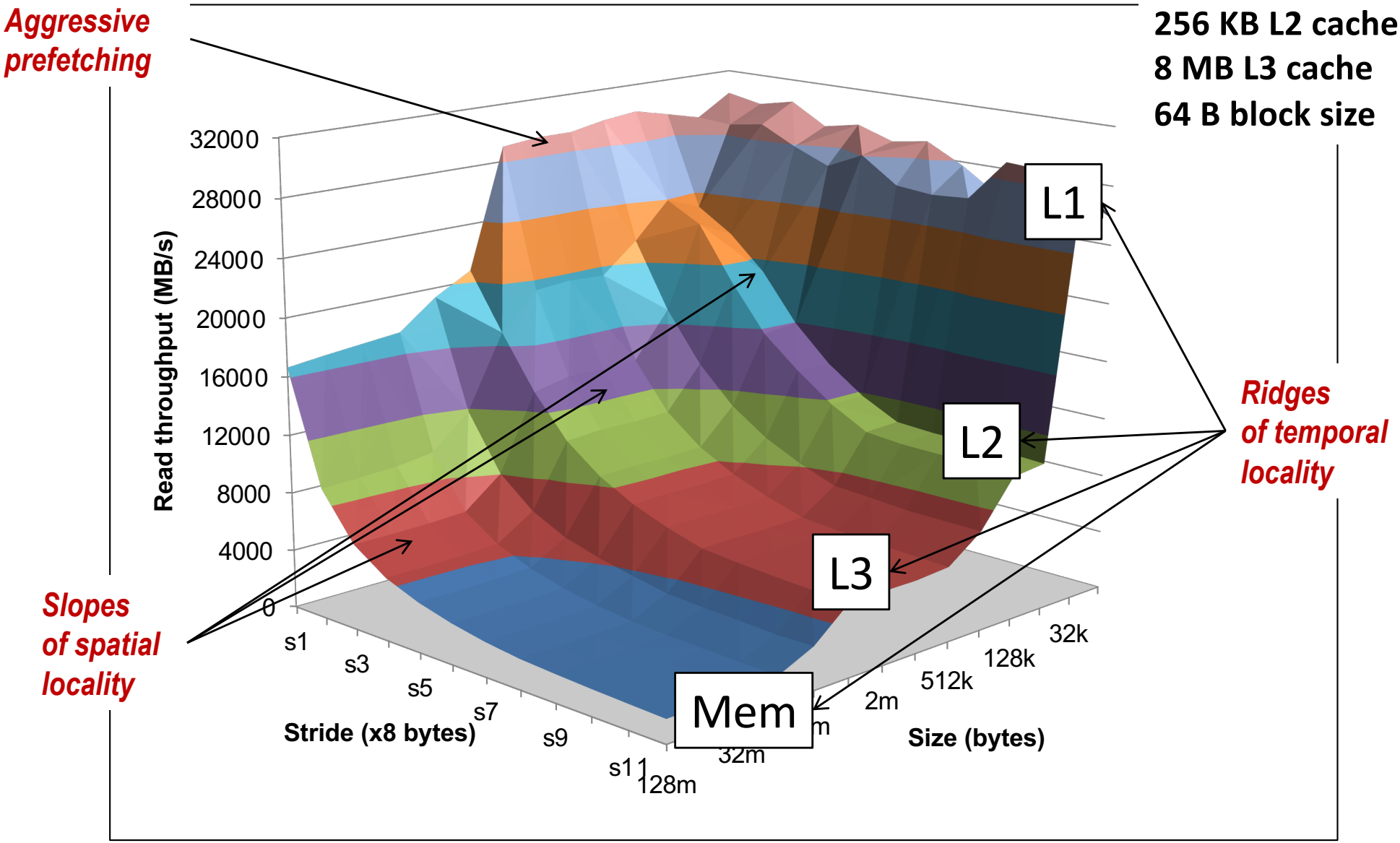
The Memory Mountain

- Read throughput (read bandwidth)
  - Number of bytes read from memory per second (MB/s)

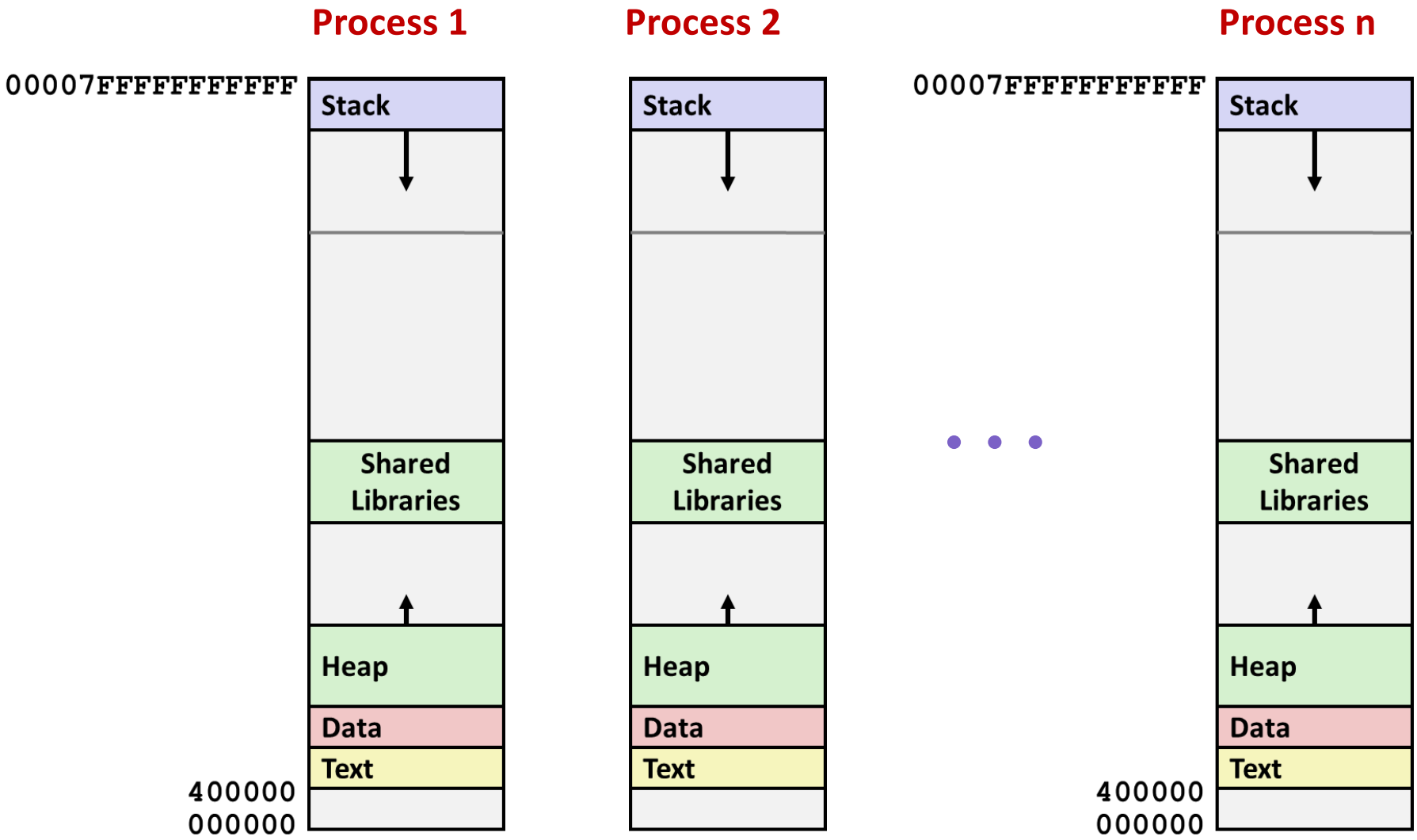
- Memory mountain: Measured read throughput as a function of spatial and temporal locality.
  - Compact way to characterize memory system performance.
  - Can make some pretty pictures, too.

The Memory Mountain

Core i5 Haswell
3.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size

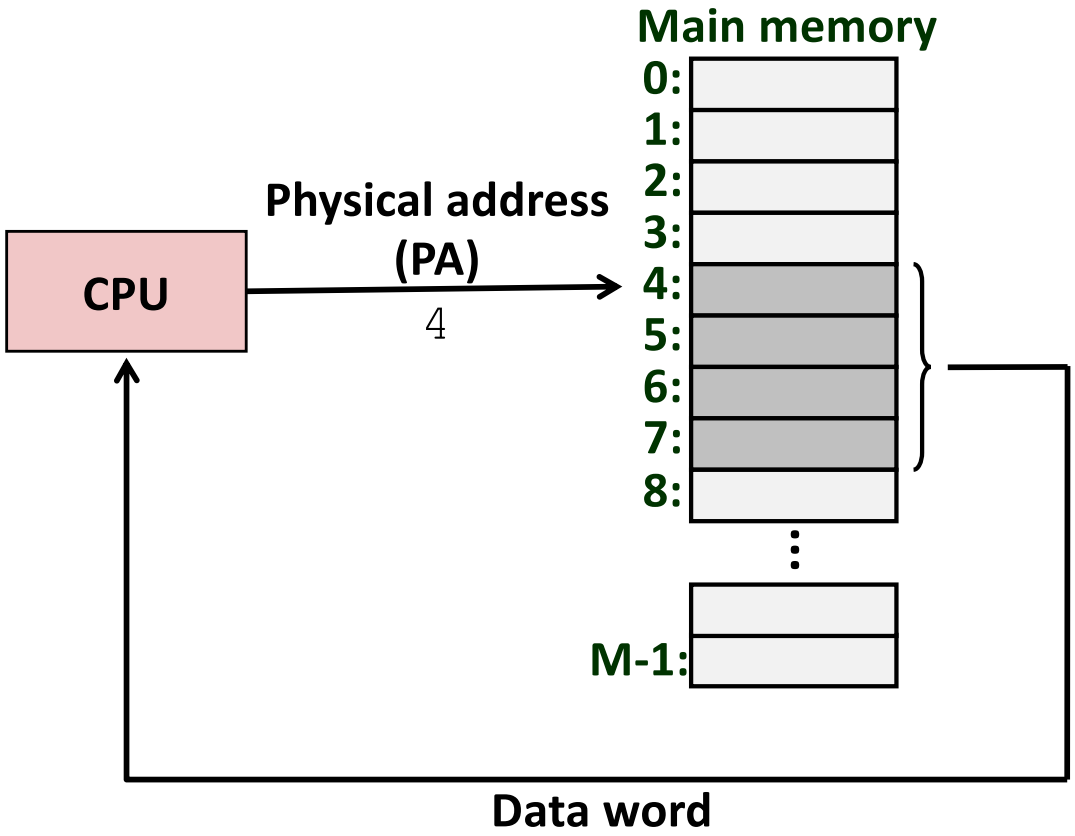


Moving on to Ch 9…
Hmmm, How Does This Work?!



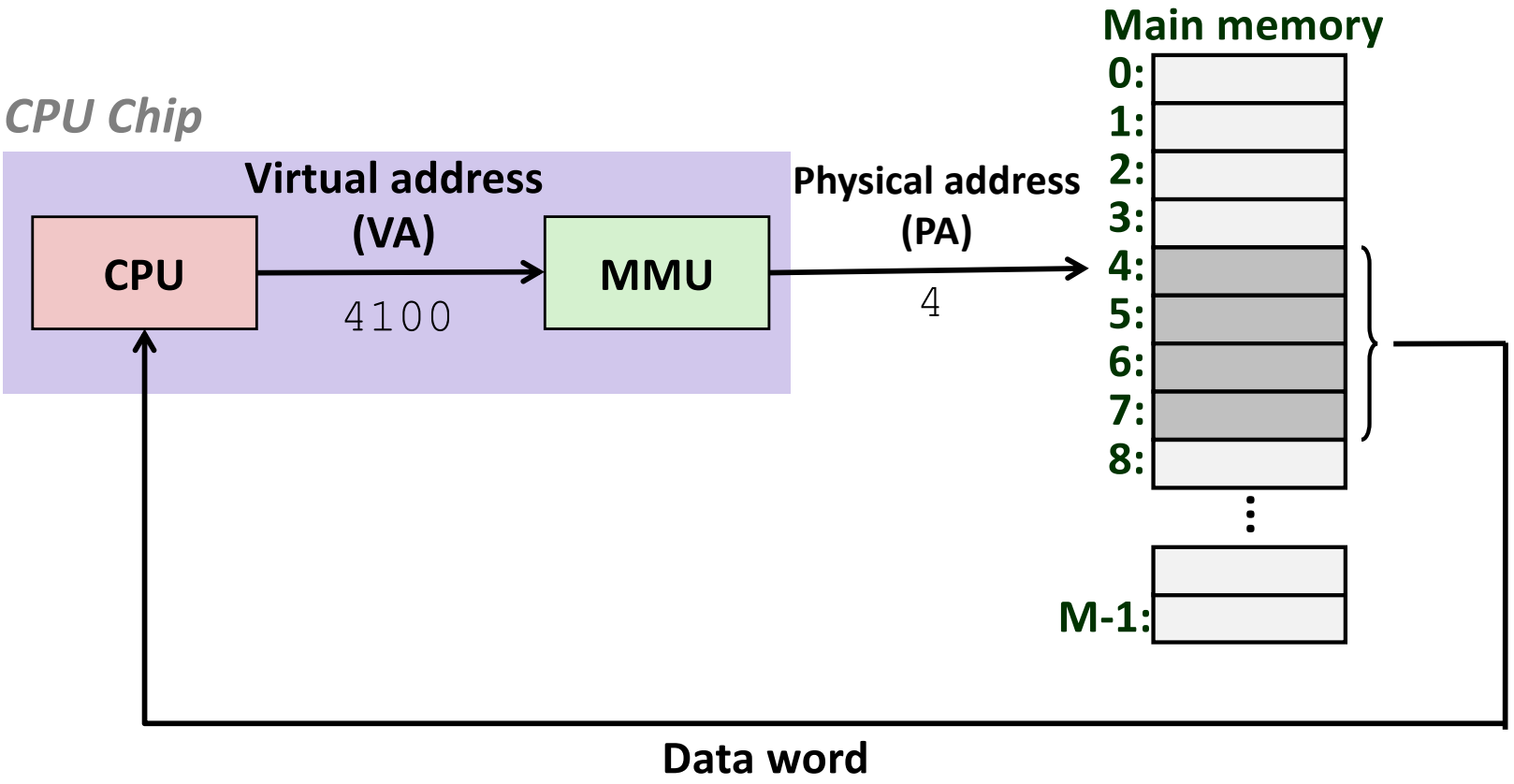
Solution: Virtual Memory (next topic…big part of OS, too)

A System Using Physical Addressing



- Used in “simple” systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames

A System Using Virtual Addressing



- Used in all modern servers, laptops, PCs, and smart phones
- One of the great ideas in computer science->use indirection!

Address Spaces

- Linear address space: Ordered set of contiguous non-negative integer addresses (we always assume this):

  {0, 1, 2, 3 … }

- Virtual address space: Set of $N = 2^n$ virtual addresses

  {0, 1, 2, 3, …, N-1}

- Physical address space: Set of $M = 2^m$ physical addresses

  {0, 1, 2, 3, …, M-1}

Why Virtual Memory (VM)?

- Uses main memory efficiently
  - Use DRAM as a cache for parts of a virtual address space
- Simplifies memory management
  - Each process gets the same uniform linear address space
- Isolates address spaces
  - One process can't interfere with another's memory
  - *User* program cannot access privileged *kernel* information and code

VM as a Tool for Caching

- Conceptually, *virtual memory* is an array of N contiguous bytes stored **on disk** (we've moved down in the hierarchy!)
- The contents of the array on disk are cached in *physical memory* (*DRAM cache*)
  - The cache blocks in this context are called *pages* (size is P = 2^p bytes)