# 15-213

*"The course that gives CMU its Zip!"*

# Cache Memories
# September 30, 2008

## Topics

- Generic cache memory organization
- Direct mapped caches
- Set associative caches
- Impact of caches on performance



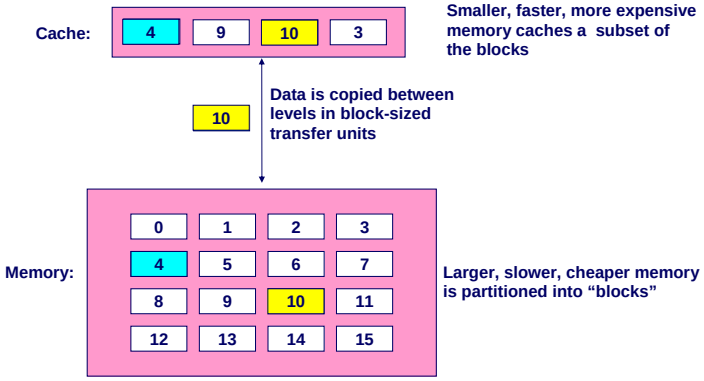
# Announcements

## Exam grading done

- Everyone should have gotten email with score (out of 72)
  - mean was 50, high was 70
  - solution sample should be up on website soon
- Getting your exam back
  - some got them in recitation
  - working on plan for everyone else (worst case = recitation on Monday)
- If you think we made a mistake in grading
  - please read the syllabus for details about the process for handling it



# General cache mechanics






# 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:
  - 1-2 clock cycle for L1
  - 5-20 clock cycles for L2

## Miss Penalty

- Additional time required because of a miss
  - typically 50-200 cycles for main memory (Trend: increasing!)

## Lets think about those numbers

**Huge difference between a hit and a miss**

- 100X, if just L1 and main memory

**Would you believe 99% hits is twice as good as 97%?**

- Consider these numbers:

cache hit time of 1 cycle
miss penalty of 100 cycles

So, average access time is:

97% hits: 1 cycle + 0.03 * 100 cycles = 4 cycles
99% hits: 1 cycle + 0.01 * 100 cycles = 2 cycles



## Many types of caches

**Examples**

- Hardware: L1 and L2 CPU caches, TLBs, …
- Software: virtual memory, FS buffers, web browser caches, …

**Many common design issues**

- each cached item has a "tag" (an ID) plus contents
- need a mechanism to efficiently determine whether given item is cached
  - combinations of indices and constraints on valid locations
- on a miss, usually need to pick something to replace with the new item
  - called a "replacement policy"
- on writes, need to either propagate change or mark item as "dirty"
  - write-through vs. write-back



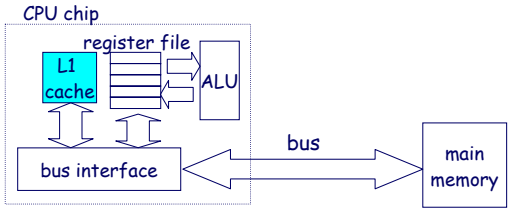
## Hardware cache memories

**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 L1, then in main memory**

**Typical system structure:**

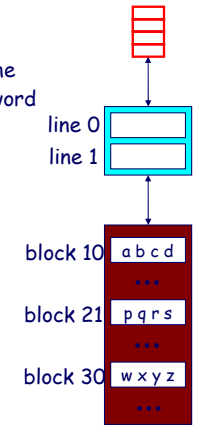





## Inserting an L1 Cache Between the CPU and Main Memory

The transfer unit between the CPU register file and the cache is a 4-byte word

The transfer unit between the cache and main memory is a 4-word block (16 bytes)




The tiny, very fast CPU register file has room for four 4-byte words

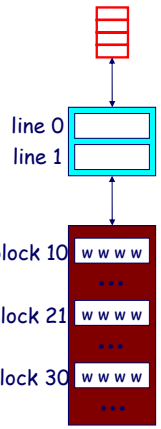
The small fast L1 cache has room for two 4-word blocks

The big slow main memory has room for many 4-word blocks

# Inserting an L1 Cache Between the CPU and Main Memory

The transfer unit between the CPU register file and the cache is a 4-byte word

The transfer unit between the cache and main memory is a 4-word block (16 bytes)




The tiny, very fast CPU register file has room for four 4-byte words

The small fast L1 cache has room for two 4-word blocks

The big slow main memory has room for many 4-word blocks



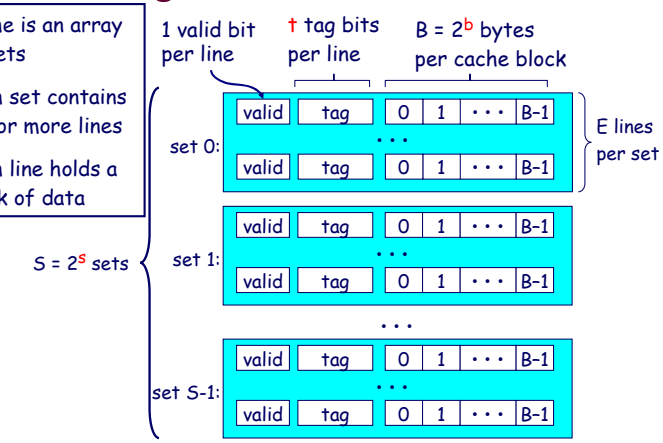


# General Organization of a Cache

Cache is an array of sets

Each set contains one or more lines

Each line holds a block of data




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



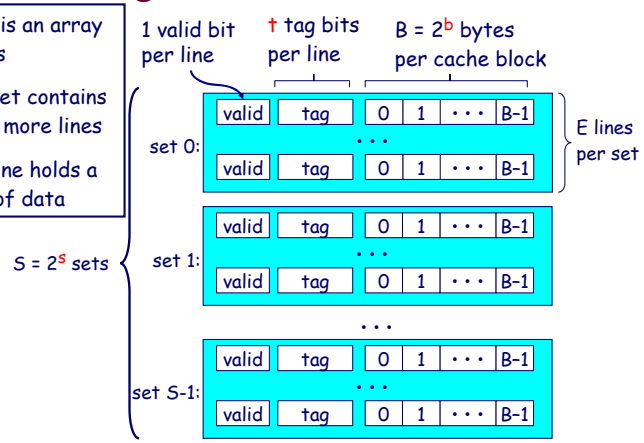


# General Organization of a Cache

Cache is an array of sets

Each set contains one or more lines

Each line holds a block of data

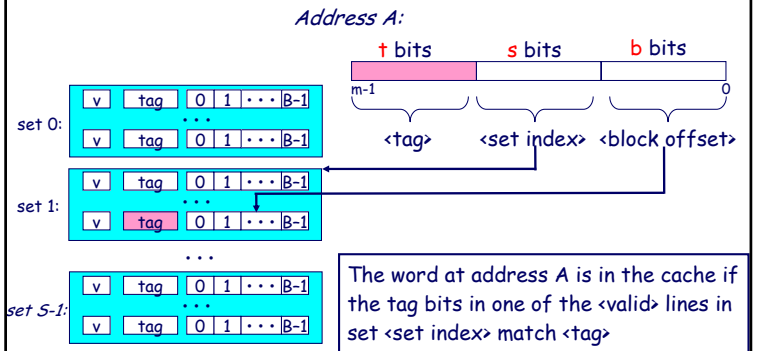



**Cache size: C = B x E x S data bytes**





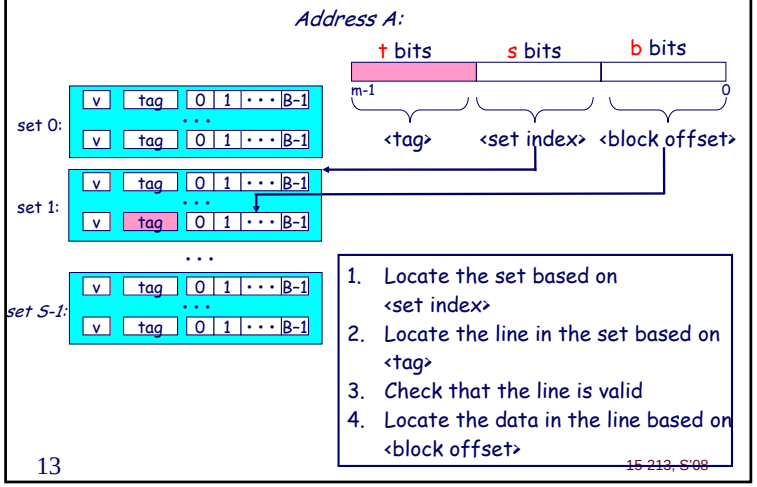
# Addressing Caches




The word at address A is in the cache if the tag bits in one of the <valid> lines in set <set index> match <tag>

The word contents begin at offset <block offset> bytes from the beginning of the block

# Addressing Caches



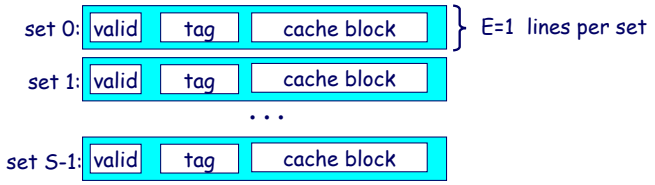

1. Locate the set based on <set index>
2. Locate the line in the set based on <tag>
3. Check that the line is valid
4. Locate the data in the line based on <block offset>




# Example: Direct-Mapped Cache

**Simplest kind of cache, easy to build (only 1 tag compare required per access)**

**Characterized by exactly one line per set.**



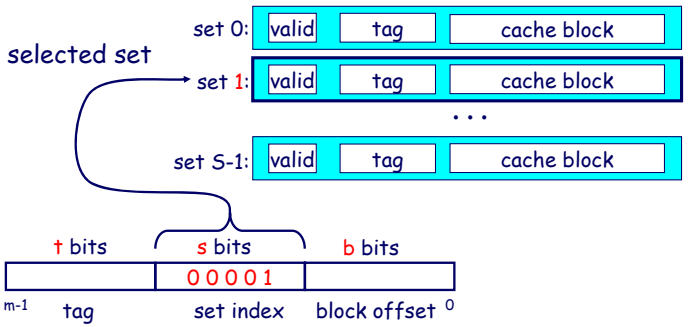

Cache size: C = B x S data bytes




# Accessing Direct-Mapped Caches

**Set selection**

- **Use the set index bits to determine the set of interest.**



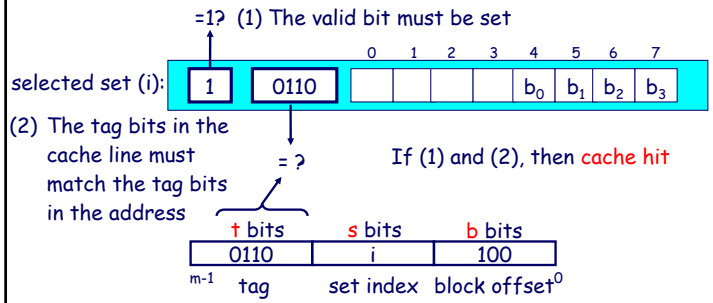




# Accessing Direct-Mapped Caches

**Line matching and word selection**

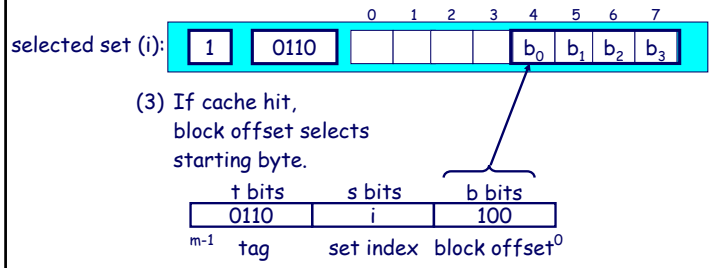
- **Line matching: Find a valid line in the selected set with a matching tag**
- **Word selection: Then extract the word**



# Accessing Direct-Mapped Caches

**Line matching and word selection**

- **Line matching: Find a valid line in the selected set with a matching tag**
- **Word selection: Then extract the word**






# Direct-Mapped Cache Simulation

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

M=16 byte addresses, B=2 bytes/block, S=4 sets, E=1 entry/set

Address trace (reads):

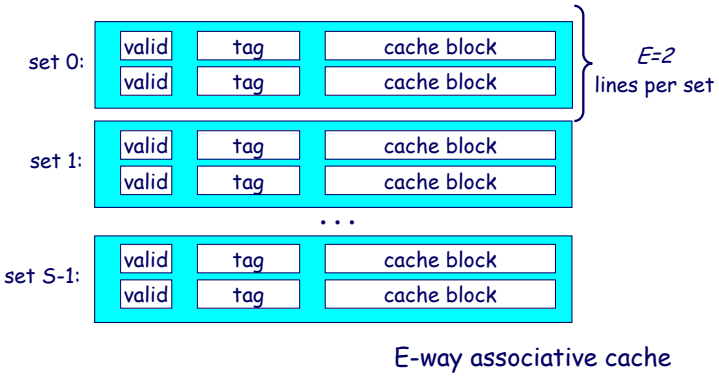
| | | |
|---|---|---|
| 0 | $[0000_2]$, | miss |
| 1 | $[0001_2]$, | hit |
| 7 | $[0111_2]$, | miss |
| 8 | $[1000_2]$, | miss |
| 0 | $[0000_2]$ | miss |

| v | tag | data |
|---|---|---|
| 1 | 0 | M[0-1] |
| | | |
| | | |
| 1 | 0 | M[6-7] |



# Example: Set Associative Cache

**Characterized by more than one line per set**

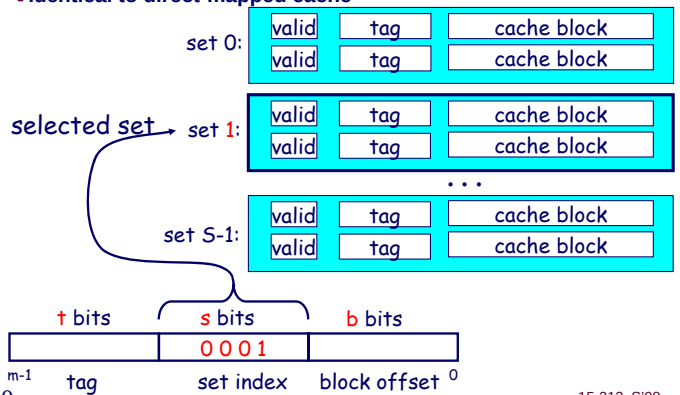





# Accessing Set Associative Caches

**Set selection**

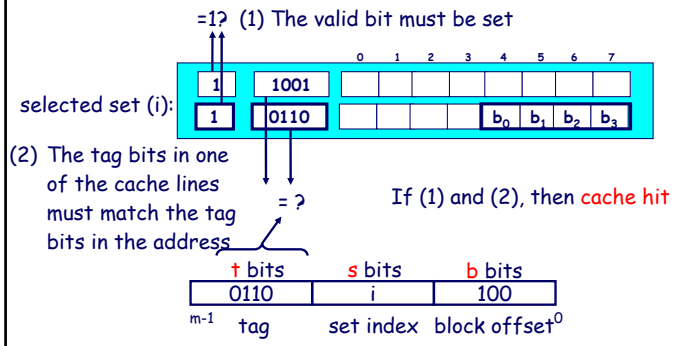
- **identical to direct-mapped cache**



# Accessing Set Associative Caches

**Line matching and word selection**

- must compare the tag in each valid line in the selected set.

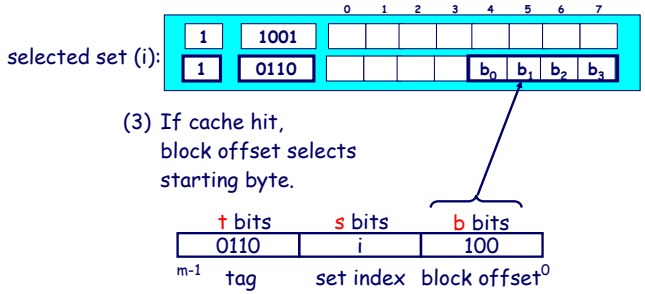





# Accessing Set Associative Caches

**Line matching and word selection**

- Word selection is the same as in a direct mapped cache






# 2-Way Associative Cache Simulation

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

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

Address trace (reads):

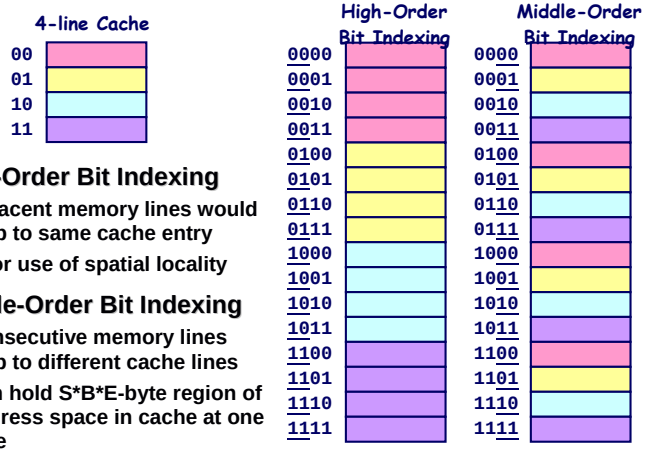
| | | |
|---|---|---|
| 0 | $[0000_2]$, | miss |
| 1 | $[0001_2]$, | hit |
| 7 | $[0111_2]$, | miss |
| 8 | $[1000_2]$, | miss |
| 0 | $[0000_2]$ | hit |

| v | tag | data |
|---|---|---|
| 1 | 00 | M[0-1] |
| 1 | 10 | M[8-9] |
| 1 | 01 | M[6-7] |
| 0 | | |



# Notice that middle bits used as index

# Why Use Middle Bits as Index?




**High-Order Bit Indexing**

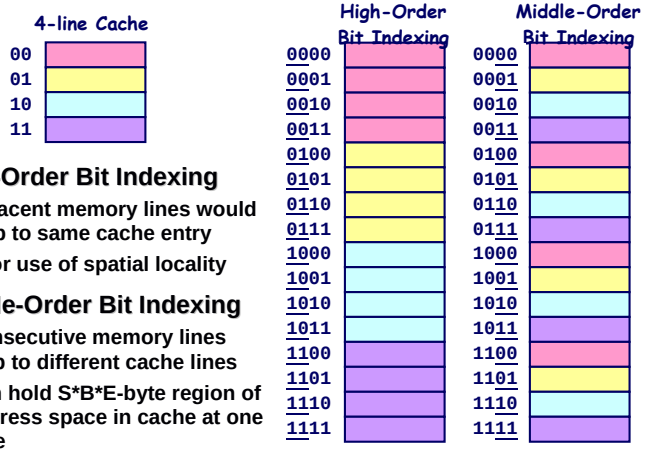
- Adjacent memory lines would map to same cache entry
- Poor use of spatial locality

**Middle-Order Bit Indexing**

- Consecutive memory lines map to different cache lines
- Can hold S*B*E-byte region of address space in cache at one time



# Why Use Middle Bits as Index?




**High-Order Bit Indexing**

- Adjacent memory lines would map to same cache entry
- Poor use of spatial locality

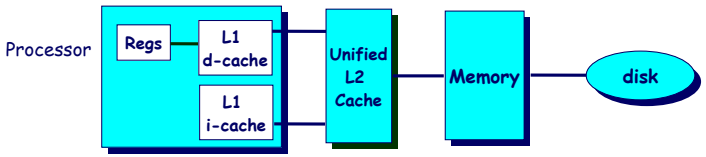
**Middle-Order Bit Indexing**

- Consecutive memory lines map to different cache lines
- Can hold S*B*E-byte region of address space in cache at one time



# Sidebar: Multi-Level Caches

**Options: separate data and instruction caches, or a unified cache**




| | | | | | |
|---|---|---|---|---|---|
| size: | 200 B | 8-64 KB | 1-4MB SRAM | 128 MB DRAM | 30 GB |
| speed: | 3 ns | 3 ns | 6 ns | 60 ns | 8 ms |
| $/Mbyte: | | | $100/MB | $1.50/MB | $0.05/MB |
| line size: | 8 B | 32 B | 32 B | 8 KB | |

larger, slower, cheaper



# What about writes?

**Multiple copies of data exist:**

- L1
- L2
- Main Memory
- Disk

**What to do when we write?**

- Write-through
- Write-back
  - need a dirty bit
  - What to do on a write-miss?

**What to do on a replacement?**

- Depends on whether it is write through or write back

## Software caches are more flexible

**Examples**

- File system buffer caches, web browser caches, etc.

**Some design differences**

- Almost always fully associative
  - so, no placement restrictions
  - index structures like hash tables are common
- Often use complex replacement policies
  - misses are very expensive when disk or network involved
  - worth thousands of cycles to avoid them
- Not necessarily constrained to single "block" transfers
  - may fetch or write-back in larger units, opportunistically



## Locality Example #1

**Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer**

**Question: Does this function have good locality?**

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```



## Locality Example #2

**Question: Does this function have good locality?**

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



## Locality Example #3

**Question: Can you permute the loops so that the function scans the 3-d array a[ ] with a stride-1 reference pattern (and thus has good spatial locality)?**

```
int sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                sum += a[k][i][j];
    return sum;
}
```