RUTGERS

# CS211 Computer Architecture

Fall 2020

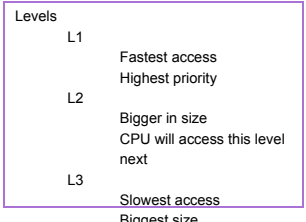
Recitation 11

# Today

- Caches
  - Cache misses
  - Cache layouts
  - Cache sizes
  - Cache writes
  - Example
  - Performance

# Caches

- What is a cache?
  - Subset of memory storage that stores data in SRAM
  - Allows for faster retrieval of data
    - Reduces need to access deeper levels of memory
    - However it is more expensive
- **Caching** – storing data about previous accesses to make future retrievals faster (makes predictions)
  - If you access data from address X recently, it will store it in the cache in case that you want to access it again in future – don't have to go to deeper in storage again (since that increases access time)
  - **Cache hit** – requested data was found within cache
  - **Cache miss** – requested data not found; need to look deeper to find it



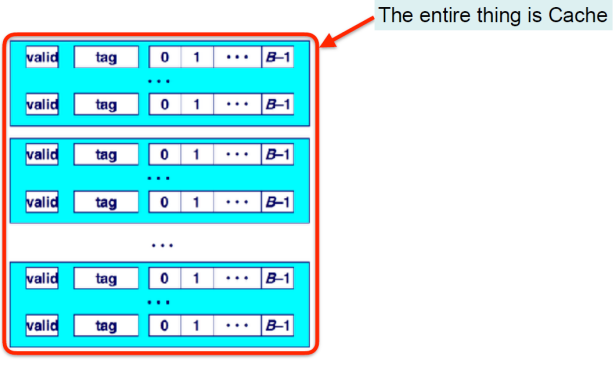
# Cache Misses

- Cold miss
  - Aka compulsory miss
  - Happens when a **block** was accessed for the first time (cache block is empty)
  - Prefetching prevents this
- Capacity miss
  - Cache is full; working set is larger than capacity
- Conflict miss
  - Collision when multiple blocks map to the same set
  - Can be prevented by making block size bigger

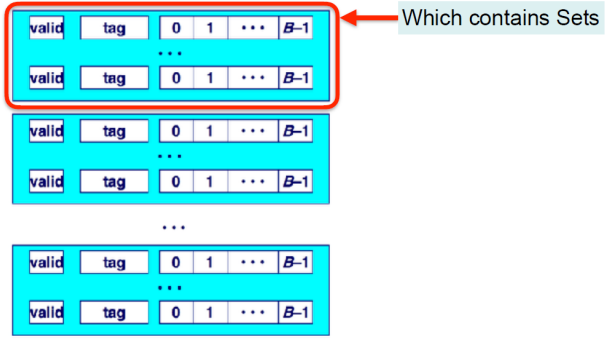
# Cache Misses – School Example

- Cold miss
  - You are the first to arrive at the school
- Capacity miss
  - Your entire school and its individual classrooms are full
- Conflict miss
  - Your classroom is full

# Caches (contd.)



# Caches (contd.)



# Caches (contd.)

# Terms

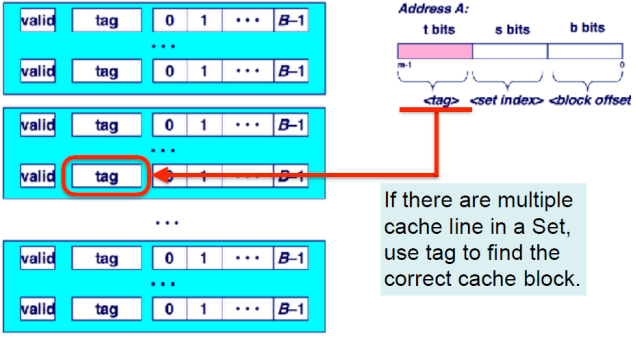
- Cache size – how many bytes are in the entire cache
- Block size – how many bytes make up each line/block
- Associativity
- Writing policies
- Replacement policies

# Cache Line

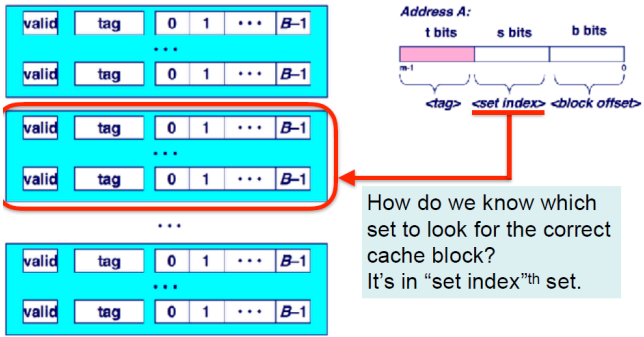
Line contains

- **Valid bit**
  - Tells whether the data in block is valid (in the cache)
- **Tag**
  - Identifier for an address in the CPU (maps cache block to CPU address)
  - Determines hit or miss
- **Set Index**
  - For each set, there is an index that “labels” each set
  - Which set (if more than 1) are we looking at
- **Block Offset**
  - Takes up the rest of the bits in the line
  - = Memory address % 2n

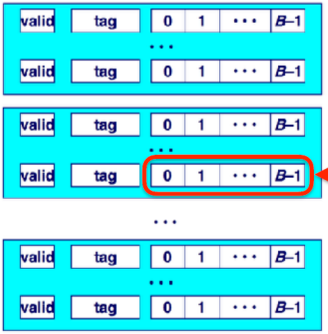
# Tag

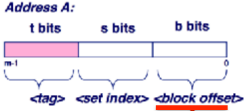

valid
tag
0
1
• • •
B–1
Address A:
t bits
s bits
b bits
m-1
0
<tag>
<set index>
<block offset
If there are multiple cache line in a Set, use tag to find the correct cache block.

# Set


valid
tag
0
1
• • •
B–1
Address A:
t bits
s bits
b bits
m-1
0
<tag>
<set index>
<block offset>
How do we know which set to look for the correct cache block?
It's in "set index"th set.

# Block Offset



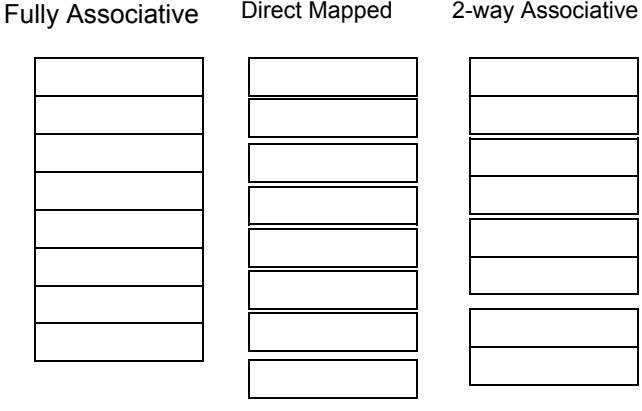




Data from Address A is in "block offset"th index of the cache block.

# Cache Associativity

- What is it?
  - Associativity determines the “layout” of the cache
  - Caches are divided into 3 categories:
- **Direct Mapped**
  - Number of sets = cache size / block size
  - Associativity = 1 (each set has only 1 line/block)
- **Fully Associative**
  - Number of sets = 1
  - Associativity = cache size / block size
- **N-Way Associative**
  - Number of sets = cache size / (block size * n)
  - Associativity = n (n lines/blocks per set)

# Associativity (contd)



# Calculations

- Depending on the associativity, we can get more information about the specific cache line bits
  - Offset size = **log2(block size)**
  - Index size = **log2(number of sets)**
  - Tag size = **A – (index size + offset size)** , where A is the length of the memory address
    - 32-bit architecture  A = 32
    - 64-bit architecture  A = 64

# Example

- Ex: 8 byte cache block, 4 way associative set, 128 bytes cache size, 32 bit addressing
  - Offset size = log2(8) = 3
  - Index size = log2(4) = 2
  - Tag size = 32 - 3 – 2 = **27**

# More Examples

- In 32-bit Addressing with 16-byte cache block and 16-set cache, how many bits do we need for tag?
  - Offset size = log2(16) = 4
  - Number of sets = 16
  - Index size = log2(16) = 4
  - Tag size = 32 – 4 – 4 = **24 bits**
- What is the size of the cache with 8-byte cache block, 4 lines per set (4-associative), with a total of 4 sets?
  - Number of sets = (cache size) / (block size * n)
  - Cache size = (number of sets) * block size * n
  - Cache size = 4 * 8 * 4 = **128 bytes**

# Writing Policies

- **cache read** = getting data from the cache; **write** = inputting data into the cache
- A *write policy* determines *when* cache data is overwritten
- On hit:
  - **Write-through** – data is written to the cache and main memory at the same time
    - Main memory is always maintained
  - **Write-back** – data is written to the cache, and then to main memory at a latter time
    - Sometimes, updated data can be in the cache while outdated data is in main memory
- On miss:
  - **Write-allocate** – data is written to the cache and the line is updated
  - **No-write-allocate** – data is written straight to main memory, not the cache

# Replacement Policies **

- Since caches are fixed in size, when inserting new blocks, we must decide which of the other block we will remove to make space

- **Least Recently Used (LRU)**
  - The least recently accessed block is replaced when a new one is added
- **First In, First Out (FIFO)**
  - The oldest item in the cache is replaced

# Example

- From final review F19

. Suppose we have a 1 KB cache that's 4-way way associative with 64 B blocks, on an architecture with a total memory size of 16 KB. For the following memory accesses, which will hit and miss?

```
0x1234
0x2000
0x3BCD
0x121F
0x3BF4
```

# Example (contd)

- Note: 1 KB = 210

1. Find how many bits are needed

1. How many addresses = 1KB * 16KB = 210 * 24 □ 14 bits for address
2. Offset bits = log(64) = **6 bits**
3. Num lines = cache size/block size = 210/26 = 24 = 16 lines
4. Num sets = cache size / (n * block size) = 210/(4 * 26) = 4
5. Index bits = log(4) = **2 bits**
6. Tag bits = 14 – 2 – 6 = **6 bits** □ 14 bits for address

Suppose we have a 1 KB cache that's 4-way way associative with 64 B blocks, on an architecture with a total memory size of 16 KB. For the following memory accesses, which will hit and miss?

0x1234
0x2000
0x3BCD
0x121F
0x3BF4

# Example (contd)

2. Extract the info from the addresses

3. Go through each address and determine hit or miss

1. **0x1234** – at this point, cache is empty, so **miss**
1. Insert tag into set 00
2. **0x2000** – set 00 is not empty, check tag; **miss**
1. each set can have 4 lines, so insert tag into set 00
3. **0x3BCD** – set 11 is empty, so **miss**
1. Insert into set 11
4. **0x121F** – set 00 is not empty, check tag; 010010 exists from address 0x1234, so **hit**
5. **0x3BF4** – set 11 is not empty; check tag; 111011 exists from address 0x2BF4, so **hit**

| Address | Binary | Tag | Set | Block |
|---|---|---|---|---|
| 0x1234 | 01 0010 0011 0100 | 010010 | 00 | 110100 |
| 0x2000 | 10 0000 0000 0000 | 100000 | 00 | 000000 |
| 0x3BCD | 11 1011 1100 1101 | 111011 | 11 | 001101 |
| 0x121F | 01 0010 0001 1111 | 010010 | 00 | 011111 |
| 0x3BF4 | 11 1011 1111 0100 | 111011 | 11 | 110100 |

Performance

- **Hit rate** $= \frac{\#\ hits}{\#\ hits + \#\ misses}$
- **Miss rate** $= 1 - hit\ ratio$
- **Hit time** – time it takes to de
  then deliver it