CS 61C Great Ideas of Computer Architecture
Fall 2017 Randy Katz, Krste Asanovic

Disc 8

1 Tag Index Offset

When working with caches, we have to be able to break down the memory addresses we work with to understand where they fit into our caches. There are three fields:

Tag - Used to distinguish different blocks that use the same index - Number of bits: leftovers
Index - The block that this piece of memory will be placed in - Number of bits: $\log_2$(# of indices)
Offset - The location in the block - Number of bits: $\log_2$(size of block)

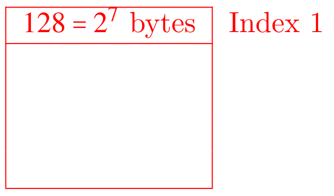
In the following diagrams, each blank box represents 1 byte (8 bits) of data. All of memory is byte addressed.

2 Direct mapped Cache

1. Let's say we have a 8192KiB cache with an 128B block size, how many bits are in tag, index, and offset? What parts of the address of 0xFEEDF00D fit into which sections?

	Tag	Index	Offset
Number of bits	9	16	7
Bits of address	111111101 (0x1FD)	1101101111100000 (0xDBE0)	0001101 (0x0D)

Visualizing our cache,



How many rows are there?
8192 KiB / 128 bytes
$= (2^{13} \cdot 2^{10})/2^7 = 2^{16}$ = rows
$\Rightarrow$ 16 index bits, (to be able index each row)

How many offset bits? $\log_2$(block size) = $\log_2$(128 bytes)
How many tag bits?($\underbrace{\text{address space bits}}_{\text{0xFEEDF00D has 32 bits}}$) - (offset bits) - (index bits)

$\Rightarrow$ 32 - 16 - 7 = 9 tag bits
T : I : O (Tag : Index : Offset) is 9 : 16 : 7

$$\text{0xFEEDF00D} \Rightarrow \text{0b}\underbrace{\overbrace{111111101}^{\text{9 bits}}}_{\text{T: 0x1fD}}\underbrace{\overbrace{1101101111100000}^{\text{16 bits}}}_{\text{I: 0xDBE0}}\underbrace{\overbrace{0001101}^{\text{7 bits}}}_{\text{O: 0x0D}}$$

Keep the diagram above in mind since we will refer to it with differing dimensions (different block sizes, different number of rows), but same conceptual structure (1 set per row, and a table structure), for direct-mapped caches.

2. Now fill in the table below. Assume that we have a write-through cache, so the number of bits per row includes only the cache data, the tag, and the valid bit.

Address size (bits)	Cache Size	Block Size	Tag Bits	Index Bits	Offset Bits	Bits per row
16	4KiB	4B	4	10	2	32 + 4 + 1 = 37
32	32KiB	16B	17	11	4	128 + 17 + 1 = 146
32	64KiB	16B	16	12	4	128 + 16 + 1 = 145
64	2048KiB	128B	43	14	7	1068

Row 1

Given information:

Cache Size: 4KiB = $2^2 \cdot 2^{10}$ bytes = $2^1 \cdot 2$ bytes

Block size: 4B = 2^2 bytes

Offset bits: $\log_2$(block size) = $\log_2(2^2)$ = 2 bits

Index bits: # rows = $\frac{\text{cache size}}{\text{block size}} = \frac{2^{12}\text{bytes}}{2^2\text{bytes}} = 2^{10}$ rows $\Rightarrow$ 10 bits to index them all (refer to the diagram in problem 1 and change the dimensions).

Tag bits: (address space bits) - (index bits) - (offset bits) = 16 - 10 - 2 = 4 bits

Bits per row: $\underbrace{4}_{\text{tag bits}} + \underbrace{4}_{\frac{\text{bytes}}{\text{block}}} \cdot \underbrace{8}_{\frac{\text{bits}}{\text{byte}}} + \underbrace{1}_{\text{valid bit}} = 4 + 32 + 1 = 37$ bits.

Row 2

Given information:

Cache Size: 32 KiB = $2^5 \cdot 2^{10}$ bytes = 2^{13} bytes

Block Size: 16B = 2^4 bytes

Offset bits: $\log_2$(block size) = $\log_2(2^4)$ = 4 bits

Index bits: #rows = $\frac{\text{cache size}}{\text{block size}} = \frac{2^{16}\text{bytes}}{2^4\text{bytes}} = 2^{11}$rows $\Rightarrow$ 11 bits to index them all

Tag bits: (address space bits) - (index bits) - (offset bits) = 32 - 11 - 4 = 17 bits

Bits per row: $\underbrace{17}_{\text{tag bits}} + \underbrace{16}_{\frac{\text{bytes}}{\text{block}}} \cdot \underbrace{8}_{\frac{\text{bits}}{\text{byte}}} + \underbrace{1}_{\text{valid bit}} = 17 + 128 + 1 = 146$ bits.

Row 3

Given information:

Tag Bits: 16 bits, Index bits: 12 bits

12 index bits $\Rightarrow 2^{12}$ rows

Offset bits: (address space bits) - (tag bits) - (index bits) = 32 - 16 - 12 = 4 bits

$\Rightarrow$ block size of $2^4 = 16$B

$\Rightarrow$ cache size of 2^{12} rows$\cdot \frac{16\text{B}}{\text{row}} = 2^{12} \cdot 2^4\text{B} = 2^6 \cdot 2^{10}\text{B}$ = 64 KiB

Bits per row: $\underbrace{16}_{\text{tag bits}} + \underbrace{16}_{\frac{\text{bytes}}{\text{block}}} \cdot \underbrace{8}_{\frac{\text{bits}}{\text{byte}}} + \underbrace{1}_{\text{valid bit}} = 16 + 128 + 1 = 145$ bits.

Row 4

Given information:

Cache Size: 2048 KiB = $2^11 \cdot 2^{10}$ bytes = 2^{21} bytes
Index bits: 14 bits
Bits per row: 1068 bits (unneeded)

14 index bits $\Rightarrow 2^{14}$ rows

Block Size: $\frac{\text{cache size}}{\text{\# rows}} = \frac{2^{21}\text{bytes}}{2^{14}\text{bytes}} = 2^7\text{B} = 128\text{B}$

Offset bits: $\log_2(\text{block size}) = \log_2(2^7) = 7$ bits

Tag bits: (address space bits) - (index bits) - (offset bits) = 64 - 14 - 7 = 43 bits

3 Cache Hits and Misses

3.1 Finding Hits and Misses

Assume we have a byte-addressed cache with capacity 32B and block size of 8B. Of the 32 bits in each address, which bits do we use to find the index of the cache to use? We use the 4th and 5th least significant bit since the offset is 3 bits

Classify each of the following byte memory accesses as acache hit (H), cache miss (M), or cache miss with replacement(R). It is probably best to try drawing out the cache before going through so that you can have an easier time seeing the replacements in the cache. The following white space is to do this:

1. 0x00000004 Index 0, Tag 0: M, Compulsory
2. 0x00000005 Index 0, Tag 0: H
3. 0x00000068 Index 1, Tag 3: M, Compulsory
4. 0x000000C8 Index 1, Tag 6: R, Compulsory
5. 0x00000068 Index 1, Tag 3: R, Conflict
6. 0x000000DD Index 3, Tag 6: M, Compulsory
7. 0x00000045 Index 0, Tag 2: R, Compulsory
8. 0x00000004 Index 0, Tag 0: R, Capacity
9. 0x000000C8 Index 1, Tag 6: R, Capacity

Detailed Explanation:
The problem assumes a direct-mapped cache. Since the leading bits of all addresses are 0, we will only write the bottom 8 bits.

Step 1: Calculate T-I-O bits.
Offset bits = $\log_2(\text{block size}) = \log_2(8) = 3$
Index bits = $\log_2(\#\text{ rows}) = \log_2(\frac{\text{cache size}}{\text{block size * associativity}}) = \log_2(\frac{2^5}{2^3 * 1}) = 2$
Note that cache size refers to how much data (in bytes) that the cache can store. It does not include bits for tag, valid, dirty, LRU etc.
Tag bits = $32 - 2 - 3 = 27$

T	I	O
27	2	3

Step 2: Caching
Logic for classifying misses:

```
If (This is the first time this block is accessed):
    Compulsory Miss
Else:
    Assume you have been using a same-size fully associative LRU cache from the
        start of the accesses
    If (The miss would be avoided by using a fully associative LRU cache):
        Conflict Miss
    Else: Miss would NOT be avoided even if you used a fully associative LRU cache:
        Capacity Miss
```

Note that the index is not written and refers to the row. The first column is the tag while the 2nd column should contain the 8B block but it's omitted here for simplicity. A "*" is included here to signal any change that occurs on a particular step.

The second cache in each step represents a fully associative cache where the index bits are absorbed as tag bits (a fully associative cache does not have index bits). The second column of the fully associative cache displays the LRU bits where 00 is the least recently used and 11 is the most recently used. When we need to evict a block, we evict the block with the lowest LRU (00). When a new block is accessed, its LRU value becomes 11 (the max) and the LRU values of all other blocks are decremented by 1. Again, the block of data itself is omitted for simplicity.

1. 0x04 = 000 00 100

Tag	LRU
000	*

1. Fully Associative Version:

Tag	LRU
00000	* 01

Unique blocks accessed so far: 1
Compulsory Miss

2. 0x05 = 000 00 101

Tag	LRU
000	

2. Fully Associative Version:

Tag	LRU
00000	01

Unique blocks accessed so far: 1
Hit (the tag matches, the block is in the cache, we just want a different byte inside the block when compared to the first access)

3. 0x68 = 011 01 000

Tag	LRU
000	
011	

3. Fully Associative Version:

Tag	LRU
00000	00
01101	* 01

Unique blocks accessed so far: 2
Compulsory Miss

4. 0xC8 = 110 01 000

Tag	LRU
000	
110	*

4. Fully Associative Version:

Tag	LRU
00000	00
01101	01
11001	* 10

Unique blocks accessed so far: 3
Compulsory Miss: note that the block with tag 011 was evicted to make room for the new block with tag 110. This is the first time we are accessing this block from memory and it is therefore a compulsory miss (no way to avoid it)

5. 0x68 = 011 01 000

Tag	LRU
000	
011	*

5. Fully Associative Version:

Tag	LRU
00000	00
01101	* 10
11001	01

Unique blocks accessed so far: 3
Conflict Miss: note that the only reason we had a cache miss for this block with tag 011 was because it was evicted in the 4th step to make space for another block. This occurred because both blocks mapped to the same cache index (01) and there's only space for 1 block in that index. To avoid this cache miss, we could use a fully associative cache and avoid evicting the 011 block in step 4. Therefore, it's a conflict miss.

6. 0xDD = 110 11 101

Tag	LRU
000	
011	
110	*

6. Fully Associative Version:

Tag	LRU
00000	00
01101	10
11001	01
11011	* 11

Unique blocks accessed so far: 4
Compulsory Miss

7. 0x45 = 010 00 101

Tag	LRU
010	*
011	
110	

7. Fully Associative Version:

Tag	LRU
01000	* 11
01101	01
11001	00
11011	10

Unique blocks accessed so far: 5
Compulsory Miss: (similar to step 4) note that even though we evicted a block to make room for this new one, this is the first time we are accessing this block from memory and it is therefore a compulsory miss (no way to avoid it). This could also count as a capacity miss because even if we made the cache fully associative, there was no way to avoid this miss.

8. 0x04 = 000 00 100

Tag	LRU
000	*
011	
110	

8. Fully Associative Version:

Tag	LRU
01000	10
01101	00
00000	* 11
11011	01

Unique blocks accessed so far: 5
Capacity Miss: while we did evict 000 previously, this is not classified as a conflict miss because 5 unique blocks have been in/out of the cache so far. Even if we used a fully associative cache that can store 4 blocks, we could not store all 5 blocks and would still miss this cache access. The only way to avoid this miss is to increase the cache size and it is therefore a capacity miss.

9. 0xC8 = 110 01 000

Tag	LRU
000	
110	*
110	

9. Fully Associative Version:

Tag	LRU
01000	01
01101	* 11
00000	10
11011	00

Unique blocks accessed so far: 5
Capacity Miss: (similar logic to step 8) while we did evict 110 previously, this is not a conflict miss because 5 unique blocks have been in/out of the cache so far. Even if we used a fully associative cache that can store 4 blocks, we could not store all 5 blocks and would still miss this cache access. The only way to avoid this miss is to increase the cache size and it is therefore a capacity miss.

3.2 The 3 C's of Misses

3 types of cache misses:

I. Compulsory: First time you ask the cache for a certain block. A miss that must occur when you first bring in a block. Reduce compulsory misses by having a longer cache line, which brings in locations before we ask for them. Can also pre-fetch blocks beforehand or have large block sizes, which means more addresses go into the same block.

II. Conflict: Occurs if you hypothetically went through the ENTIRE string of accesses with a fully associative cache and wouldn't have missed for that specific access. Increasing the associativity or improving the replacement policy would remove the miss.

III. Capacity: The only way to remove the miss is to increase the cache capacity, as even with a fully associative cache, we had to kick a block out at some point.

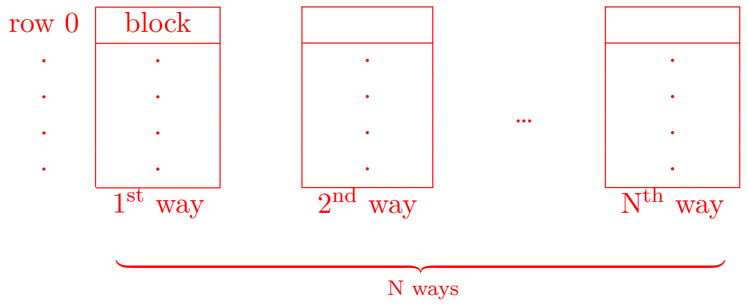
Note: There are many different ways of fixing misses. The name of the miss doesn't necessarily tell us the best way to reduce the number of misses.

Classify each M and R above as one of the 3 misses above.

4 N-Way Set Associative Caches

1. Assuming 32 bits of physical memory, for an 8-way set associative 4KiB cache with 16B blocks, how big are the T, I, and O fields? T = 23, I = 5, O = 4

   First, a general visualization:

   Now, onto the problem. Given information: 32-bit address space, 8-way set associativity, cache size: 4KiB $= 2^2 \cdot 2^{10}$B $= 2^{12}$B, block size: 16B $= 2^4$B.

   $\#$ blocks $= \frac{\text{cache size}}{\text{block size}} = \frac{2^{12}\text{B}}{2^4\text{B}} = 2^8$ blocks

   $\#$ rows $= \frac{\#\text{ blocks}}{\#\text{ ways}} = \frac{2^8\text{ blocks}}{2^3\text{ ways}} = 2^5$ rows $\Rightarrow$ 5 bits to index them all

   $\Rightarrow$ 5 index bits (refer to the diagram, this may initially be confusing)!

   Offset bits: $\log_2$(block size) $= \log_2(2^4\text{B}) = 4$ offset bits

   Tag bits: (address space bits) - (index bits) - (offset bits) = 32 - 5 - 4 = 23 bits

   $\Rightarrow$ T : I : O $\Rightarrow$ 23 : 5 : 4

2. How many total bits of storage are required for the cache if it uses write back and LRU replacement? 2^8 rows, each row has Tag, Data, Valid, LRU Type of Dirty ($\log_2$(Associativity)) = 23 + 128 + 1 + 1 + 3 = 156 bits. $156(2^8) = 39936$ bits.

   Let's look at what bits go into a row. Since we have a write-back cache, our cache may be dirty with respect to memory, so when we evict a block, we need to know whether to write it back to memory. Thus we need a dirty bit. We also need all the typical bits (valid, tag, and data [bits in a block] bits). Additionally, we want LRU replacement. So what are we replacing and how many of them are we replacing? Whenever we get an address we first find a row (its index), from there, there are 8 potential options for replacement (see the diagram from part 1), since we have an 8-way associative cache.

More generally, LRU bits = $\log_2$(associativity).

Consolidating all of this:

Tag bits: 23 bits
Data bits: $16 \cdot 8 = 128$ bits
Valid bit: 1 bit
Dirty bit: 1 bit
LRU bits: $\log_2(8) = 3$ bits(per block, referring to its usage status across a row, see the explanation above and the diagram in part 1).

$\Rightarrow$ Bits per block: $128 + 23 + 1 + 1 + 3 = 156$ bits

From part 1, there are 2^8 blocks

$\Rightarrow$ cache size in bits including additional information bits is

$2^8 \cdot 156$ bits $= 39936$ bits