# CSE 351 Section 7 – Caches

Hi there! Welcome back to section, we're happy that you're here ☺

## Locality!

Recall that we have two types of locality that we can have in code:

**Temporal locality**: when recently referenced items are likely to be referenced again in the near future.
**Spatial locality**: when nearby addresses tend to be referenced close together in time.

For each type of locality, can you give an example of when we might see it in code?

Temporal Locality:

Accessing a sum counter over and over; reading and writing to the same variable; etc.

Spatial Locality:

Accessing a[0] in an array, then a[1], then a[2] in order; accessing the first field in a struct, then the second, then the third; etc.

## Accessing a Direct-Mapped Cache (Hit or Miss?)

Assume the cache has block size $K = 4$ and is in the current state shown (you can ignore "−").
All values are shown in hex. Tag fields are padded, and bytes of the cache blocks are shown in full. The word size for the machine with these caches is 12 bits (i.e. addresses are 12 bits long) and the machine is little-endian.

Direct-Mapped:

| Set | Valid | Tag (8 bits) | B0 | B1 | B2 | B3 | Set | Valid | Tag (8 bits) | B0 | B1 | B2 | B3 |
|-----|-------|------|----|----|----|----|-----|-------|------|----|----|----|----|
| 0 | 1 | 15 | 63 | B4 | C1 | A4 | 8 | 0 | − | − | − | − | − |
| 1 | 0 | − | − | − | − | − | 9 | 1 | 00 | 01 | 12 | 23 | 34 |
| 2 | 0 | − | − | − | − | − | A | 1 | 01 | 98 | 89 | CB | BC |
| 3 | 1 | 0D | DE | AF | BA | DE | B | 0 | 1E | 4B | 33 | 10 | 54 |
| 4 | 0 | − | − | − | − | − | C | 0 | − | − | − | − | − |
| 5 | 0 | − | − | − | − | − | D | 1 | 11 | C0 | 04 | 39 | AA |
| 6 | 1 | 13 | 31 | 14 | 15 | 93 | E | 0 | − | − | − | − | − |
| 7 | 0 | − | − | − | − | − | F | 1 | 0F | FF | 6F | 30 | 0 |

Offset bits: **2**

Index bits: **4**

Tag bits: **6**

| | Hit or Miss? | Data returned |
|---|---|---|
| a) Read 1 byte at `0x7AC` | Miss | — |
| b) Read 1 byte at `0x024` | Hit | 0x01 |
| c) Read 2 bytes at `0x34E` | Hit | 0xDEBA |

2-way Set Associative:

| Set | Valid | Tag (8 bits) | B0 | B1 | B2 | B3 | Set | Valid | Tag (8 bits) | B0 | B1 | B2 | B3 |
|-----|-------|------|----|----|----|----|-----|-------|------|----|----|----|----|
| 0 | 0 | − | − | − | − | − | 0 | 0 | − | − | − | − | − |
| 1 | 0 | − | − | − | − | − | 1 | 1 | 2F | 01 | 20 | 40 | 03 |
| 2 | 1 | 03 | 4F | D4 | A1 | 3B | 2 | 1 | 0E | 99 | 09 | 87 | 56 |
| 3 | 0 | − | − | − | − | − | 3 | 0 | − | − | − | − | − |
| 4 | 0 | 06 | CA | FE | F0 | 0D | 4 | 0 | − | − | − | − | − |
| 5 | 1 | 21 | DE | AD | BE | EF | 5 | 0 | − | − | − | − | − |
| 6 | 0 | − | − | − | − | − | 6 | 1 | 37 | 22 | B6 | DB | AA |
| 7 | 1 | 11 | 00 | 12 | 51 | 55 | 7 | 0 | − | − | − | − | − |

Offset bits: **2**

Index bits: **3**

Tag bits: **7**

|  | Hit or Miss? | Data returned |
|---|---|---|
| a) Read 1 byte at `0x435` | Hit | 0xAD |
| b) Read 1 byte at `0x388` | Miss | — |

Fully Associative:

| Set | Valid | Tag (12 bits) | B0 | B1 | B2 | B3 | Set | Valid | Tag (12 bits) | B0 | B1 | B2 | B3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1F4 | 00 | 01 | 02 | 03 | 0 | 0 | — | — | — | — | — |
| 0 | 0 | — | — | — | — | — | 0 | 1 | 0AB | 02 | 30 | 44 | 67 |
| 0 | 1 | 100 | F4 | 4D | EE | 11 | 0 | 1 | 034 | FD | EC | BA | 23 |
| 0 | 1 | 077 | 12 | 23 | 34 | 45 | 0 | 0 | — | — | — | — | — |
| 0 | 0 | — | — | — | — | — | 0 | 1 | 1C6 | 00 | 11 | 22 | 33 |
| 0 | 1 | 101 | DA | 14 | EE | 22 | 0 | 1 | 045 | 67 | 78 | 89 | 9A |
| 0 | 0 | — | — | — | — | — | 0 | 1 | 001 | 70 | 00 | 44 | A6 |
| 0 | 1 | 016 | 90 | 32 | AC | 24 | 0 | 0 | — | — | — | — | — |

Offset bits: **2**

Index bits: **0**

Tag bits: **10**

|  | Hit or Miss? | Data returned |
|---|---|---|
| a) Read 1 byte at `0x1DD` | Hit | 0x23 |
| b) Read 1 byte at `0x719` | Hit | 0x11 |
| c) Read 1 byte at `0x2AA` | Miss | — |

## Code Analysis

Consider the following code that accesses a <u>two-dimensional</u> array (of size 64×64 `ints`).
Assume we are using a direct-mapped, 1 KiB cache with 16 B block size, and that the cache starts cold.
Also assume that the variables `sum`, `i`, and `j` are stored in registers.

```
int sum = 0;
for (int i = 0; i < 64; i++)
    for (int j = 0; j < 64; j++)
        sum += array[i][j];        // assume &array = 0x600000
```

a) What is the miss rate of the execution of the entire loop?
Every block can hold 4 ints (16B/4B per int), so we will need to pull a new block from memory every 4 accesses of the array. This means this miss rate is $\frac{4\ bytes\ per\ int}{16\ bytes\ per\ block} = \frac{1\ block}{4\ ints} = 0.25 = 25\%$

b) If we have an average memory access time (AMAT) of 60 ns and a hit time of 10 ns, what is the miss penalty?

With the 25% miss rate from part **(a)**, we have $10 + 0.25 * MP = 60$. Solving for MP tells us the miss penalty is 200 ns.

c) What code modifications can <u>change</u> the miss rate?  Brainstorm before trying to analyze.
Possible answers: switch the loops (i.e. make j the outer loop and i the inner loop), switch j and i in the array access, make the array a different type (e.g. char[ ][ ], long[ ][ ], etc.), make array an array of Linked Lists or a 2-level array, etc.

d) What cache parameter changes (size, associativity, block size) can <u>change</u> the miss rate?
Let's consider each of the three parameters individually.

First, let's consider modifying the size of the cache. Will it change the miss rate?

No, it doesn't matter how big the cache is in this case (if the block size doesn't change). We will still be pulling the same amount of data each miss, and we will still have to go to memory every time we exhaust that data

Next, let's consider modifying the associativity of the cache. Will it change the miss rate?
No, this is helpful if we want to reduce conflict misses, but since the data we're accessing is all in contiguous memory (thanks arrays!), booting old data to replace it with new data isn't an issue.

Finally, let's consider modifying the block size of the cache. Will it change the miss rate?
Yes, bigger blocks mean we pull bigger chunks of contiguous elements in the array every time we have a miss. Bigger chunks at a time means fewer misses down the line. Likewise, smaller blocks increase the frequency with which we need to go to memory (think back to the calculations we did in part **(a)** to see why this is the case)

So, in conclusion, changing block size can change the miss rate. Changing size or associativity will NOT change the miss rate.

NOTE: Remember that the results we got were for this specific example. There are some code examples in which changing the size or associativity of the cache will change the miss rate.

## Cache Simulator!

If you need help on using the cache sim, take a look at additional supplemental material that will guide you through using the cache sim (posted with today's section handouts)! We haven't covered all the material contained in the cache simulator yet, but we hope you'll find it useful for lab 4 and corresponding homework assignments.