CSE 351 Section 7 – Caches

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

IEC Prefixing System

We often need to express large numbers and the preferred tool for doing so is the IEC Prefixing System!

Kibi-	(Ki)	$2^{10} \approx 10^3$	**Pebi-**	(Pi)	$2^{50} \approx 10^{15}$
Mebi-	(Mi)	$2^{20} \approx 10^6$	**Exbi-**	(Ei)	$2^{60} \approx 10^{18}$
Gibi-	(Gi)	$2^{30} \approx 10^9$	**Zebi-**	(Zi)	$2^{70} \approx 10^{21}$
Tebi-	(Ti)	$2^{40} \approx 10^{12}$	**Yobi-**	(Yi)	$2^{80} \approx 10^{24}$

Prefix Exercises:

Write the following as powers of 2. The first one has been done for you:

2 Ki-bytes = **2^{11} bytes**	64 Gi-bits = 2^{36} bits	16 Mi-integers = 2^{24} integers
256 Pi-pencils = 2^{58} pencils	512 Ki-books = 2^{19} books	128 Ei-students = 2^{67} students

Write the following using IEC Prefixes. The first one has been done for you:

2^{15} cats = **32 Ki-cats**	2^{34} birds = 16 Gi-birds	2^{43} huskies = 8 Ti-huskies
2^{61} things = 2 Ei-things	2^{27} caches = 128 Mi-caches	2^{58} addresses = 256 Pi-addresses

---

Accessing a Cache (Hit or Miss?)

Assume the following caches all have block size B = 4 and are in the current state shown (you can ignore "−").
All values are shown in hex. Tag fields are NOT padded, while 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)

Direct-Mapped:

Set	Valid	Tag	B0	B1	B2	B3
0	1	15	63	B4	C1	A4
1	0	—	—	—	—	—
2	0	—	—	—	—	—
3	1	D	DE	AF	BA	DE
4	0	—	—	—	—	—
5	0	—	—	—	—	—
6	1	13	31	14	15	93
7	0	—	—	—	—	—

Set	Valid	Tag	B0	B1	B2	B3
8	0	—	—	—	—	—
9	1	0	01	12	23	34
A	1	1	98	89	CB	BC
B	0	1E	4B	33	10	54
C	0	—	—	—	—	—
D	1	11	C0	04	39	AA
E	0	—	—	—	—	—
F	1	F	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 1 byte at `0x99F`	Miss	—

2-way Set Associative:

Set	Valid	Tag	B0	B1	B2	B3
0	0	—	—	—	—	—
1	0	—	—	—	—	—
2	1	3	4F	D4	A1	3B
3	0	—	—	—	—	—
4	0	6	CA	FE	F0	0D
5	1	21	DE	AD	BE	EF
6	0	—	—	—	—	—
7	1	11	00	12	51	55

Set	Valid	Tag	B0	B1	B2	B3
0	0	—	—	—	—	—
1	1	2F	01	20	40	03
2	1	0E	99	09	87	56
3	0	—	—	—	—	—
4	0	—	—	—	—	—
5	0	—	—	—	—	—
6	1	37	22	B6	DB	AA
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	—
c)	Read 1 byte at `0x0D3`	Miss	—

Fully Associative:

Set	Valid	Tag	B0	B1	B2	B3
0	1	1F4	00	01	02	03
0	0	—	—	—	—	—
0	1	100	F4	4D	EE	11
0	1	77	12	23	34	45
0	0	—	—	—	—	—
0	1	101	DA	14	EE	22
0	0	—	—	—	—	—
0	1	16	90	32	AC	24

Set	Valid	Tag	B0	B1	B2	B3
0	0	—	—	—	—	—
0	1	AB	02	30	44	67
0	1	34	FD	EC	BA	23
0	0	—	—	—	—	—
0	1	1C6	00	11	22	33
0	1	45	67	78	89	9A
0	1	1	70	00	44	A6
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 two-dimensional array (of size 64×64 `ints`).
Assume we are using a direct-mapped, 1 KiB cache with 16 B block size.

```
for (int i = 0; i < 64; i++)
    for (int j = 0; j < 64; j++)
        array[i][j] = 0;        // 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) 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.

(NOTE: Answer to part **(c)** on next page)

c) 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.

Question F5: Caching [10 pts]

We have 16 KiB of RAM and two options for our cache. Both are two-way set associative with 256 B blocks, LRU replacement, and write-back policies. **Cache A** is size 1 KiB and **Cache B** is size 2 KiB.

(A) Calculate the TIO address breakdown for **Cache B**: [1.5 pt]

Tag bits	Index bits	Offset bits
4	2	8

14 address bits. $\log_2 256 = 8$ offset bits. 2 KiB cache = 8 blocks. 2 blocks/set → 4 sets.

(B) The code snippet below accesses an integer array. Calculate the **Miss Rate** for **Cache A** if it starts *cold*. [3 pt]

```
#define LEAP 4
#define ARRAY_SIZE 512
int nums[ARRAY_SIZE];              // &nums = 0x0100 (physical addr)
for (i = 0; i < ARRAY_SIZE; i+=LEAP)
    nums[i] = i*i;
```

1/16

Access pattern is a single write to nums[i]. Stride = LEAP = 4 ints = 16 bytes. 256/16 = 16 strides per block. First access is a compulsory miss and the next 15 are hits. Since we never revisit indices, this pattern continues for all cache blocks. You can also verify that the offset of &nums is 0x00, so we start at the beginning of a cache block.

(C) For each of the proposed (independent) changes, write **MM** for "higher miss rate", **NC** for "no change", or **MH** for "higher hit rate" to indicate the effect on **Cache A** for the code above:[3.5 pt]

Direct-mapped _**NC**_   Increase block size _**MH**_

Double LEAP _**MM**_   Write-through policy _**NC**_

Since we never revisit blocks, associativity doesn't matter. Larger block size means more strides/block. Doubling LEAP means fewer strides/block. Write hit policy has no effect.

(D) Assume it takes 200 ns to get a block of data from main memory. Assume **Cache A** has a hit time of 4 ns and a miss rate of 4% while **Cache B**, being larger, has a hit time of 6 ns. What is the worst miss rate Cache B can have in order to perform as well as Cache A? [2 pt]

0.03 or 3%

$AMAT_A = HT_A + MR_A \times MP = 4 + 0.04*200 = 12$ ns.
$AMAT_B = HT_B + MR_B \times MP \le 12 \rightarrow 200\ MR_B \le 6 \rightarrow MR_B \le 0.03$