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)	$2^{80} \approx 10^{24}$

<u>Prefix Exercises</u>:

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

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

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

2^{15} cats = **32 Ki-cats**	2^{34} birds =	2^{43} huskies =
2^{61} things =	2^{27} caches =	2^{58} addresses =

Accessing a Cache (Hit or Miss?)

Assume the following caches all have block size $K = 4$ 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)

16 sets, so there must be 4 index bits
4 bytes per block so there must be 2 offset bits
the tag bits must be the remaining 6 bits

<u>Direct-Mapped</u>:

Set	Valid	Tag	B0	B1	B2	B3	Set	Valid	Tag	B0	B1	B2	B3
0	1	15	63	B4	C1	A4	8	0	–	–	–	–	–
1	0	–	–	–	–	–	9	1	0	01	12	23	34
2	0	–	–	–	–	–	A	1	1	98	89	CB	BC
3	1	D	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	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` 11110101100 set = D, tag = 11110 = 1E, offset = 0	miss	
b) Read 1 byte at `0x024` 100100 set = 9, tag = 0, offset = 0	hit	01 (data at b0)
c) Read 1 byte at `0x99F` 100110011111 set = 7, tag = 100110 =26, offset = 3	miss	

tag 6 bits	index 4 bits	offset 2 bits

8=2^3 sets so there are 3 index bits
4=2^2 block size so there are 2 offset bits.
The remaining 7 bits are for the tag

2-way Set Associative:

Set	Valid	Tag	B0	B1	B2	B3	Set	Valid	Tag	B0	B1	B2	B3
0	0	—	—	—	—	—	0	0	—	—	—	—	—
1	0	—	—	—	—	—	1	1	2F	01	20	40	03
2	1	3	4F	D4	A1	3B	2	1	0E	99	09	87	56
3	0	—	—	—	—	—	3	0	—	—	—	—	—
4	0	6	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` 10000110101 set = 5, tag = 100001=21, offset = 1	hit	AD
b) Read 1 byte at `0x388` 1110001000 set = 010= 2, tag = 11100=1c, offset = 0	miss	
c) Read 1 byte at `0x0D3`		

Fully Associative:

Set	Valid	Tag	B0	B1	B2	B3	Set	Valid	Tag	B0	B1	B2	B3
0	1	1F4	00	01	02	03	0	0	—	—	—	—	—
0	0	—	—	—	—	—	0	1	AB	02	30	44	67
0	1	100	F4	4D	EE	11	0	1	34	FD	EC	BA	23
0	1	77	12	23	34	45	0	0	—	—	—	—	—
0	0	—	—	—	—	—	0	1	1C6	00	11	22	33
0	1	101	DA	14	EE	22	0	1	45	67	78	89	9A
0	0	—	—	—	—	—	0	1	1	70	00	44	A6
0	1	16	90	32	AC	24	0	0	—	—	—	—	—

Offset bits: ________

Index bits: ________

Tag bits: ________

	Hit or Miss?	Data returned
a) Read 1 byte at `0x1DD`		
b) Read 1 byte at `0x719`		
c) Read 1 byte at `0x2AA`		

Code Analysis

Consider the following code that accesses a <u>two-dimensional</u> array (of size 64×64 `int`s). 32 bit addresses
Assume we are using a direct-mapped, 1 KiB cache with 16 B block size. 1KiB =2^10 bytes

```
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 of the 4 byte ints in it. Since the inner loop moves rowwise and 2D arrays are stored contiguously rowwise, we have good spatial locality. At i=0 we load in array[0, 0] to array[0, 3], then the next 4 elements in the row, and so on. Hence we only have one miss every four reads (at the first of the four elements). Thus the miss rate is 25%

b) What code modifications can <u>change</u> the miss rate? Brainstorm before trying to analyze.

Loop over the array columnwise rather than rowwise: this causes a 100% miss rate.
Change how the array is stored in memory (see the section 6 worksheet) to reduce spatial locality of data.

c) What cache parameter changes (size, associativity, block size) can <u>change</u> the miss rate?

Increasing block size will decrease the miss rate (and vice versa)

Other changes will have no effect because we only access the data at each array entry once and we use all 4 entries we load as soon as we load them (
conflict misses)

Cache Simulator Demo

Let's get some practice with the cache simulator! First, go to:

`https://courses.cs.washington.edu/courses/cse351/cachesim/`

At the top you'll see 4 boxed regions:

- System Parameters † — This lets you play around with the structure/format of the cache
- Manual Memory Access † — This is where you actually make reads and writes to memory
- History — An interactive log of executed accesses. You can type/paste accesses here, too!
- Simulation Messages — Describes the most recent actions made by the simulator.

† These include "Explain" toggles that walk you through execution step-by-step.

---

a) Set the following System Parameters (but *don't* generate the system yet):

Address Width → **6**, Cache Size → **16**, Block Size → **4**, Associativity → **2**, leave the rest at default values.

Based on just the system parameter numbers above shown, predict the following: cache size / (block size * associativity)

i) Highest memory address: 0b_F___________ ii) Number of sets in cache: num sets16/4/2 = 2 ______________

[*Click "Generate System" to verify your responses*]

10 1010

b) We are about to **READ** the byte at the address **0x2A**. Predict the following:

i) This block will be placed in set #: __0__ ii) The stored tag bits will be: 0b__101__

iii) The 4 bytes of *data* in this block are (in order): 0x__e9__, 0x__36__, 0x__ae__, 0x__32__

[*Enter "2a" into the Read Addr and click "Read" to verify your responses*]

01 1011

c) We are about to **WRITE** the byte **0xB1** to the address **0x1B**. Predict the following:

i) This block will be placed in set #: __0__ ii) The stored tag bits will be: 0b__011__

[*Enter "1b" into the Write Addr and "b1" into the Write Byte and then click "Write" to verify your responses*]

iii) Notice that the value of the byte at address 0x1B is different in the cache and memory.

What indicates this disparity in the cache? __the dirty bit is set to a 1 in cache__

What would have happened if our write miss policy were "**No Write-Allocate**" instead?

It would have written to memory immediately rather than waiting for the entry to be cleared in cache

d) We are about to **READ** the byte at address **0x01**. Predict the following:

i) This block will be placed in set #: ______ ii) The stored tag bits will be: 0b______

iii) Will this access cause a conflict/replacement? (circle one) Yes No

iv) If yes, which block will be evicted? (circle one) Read from (b) Write from (c)

[*Enter "01" into the Read Addr and click "Read" to verify your responses*]

e) We are about to **WRITE** the byte **0xE9** to the address **0x1C**. Predict the following:

i) This block will be placed in set #: ______ ii) The stored tag bits will be: 0b______

iii) Will this access cause a conflict/replacement? (circle one) Yes No

iv) If yes, which block will be evicted? Read from (b) Write from (c) Read from (d)

[*Enter "1c" into the Write Addr and "e9" into the Write Byte and then click "Write" to verify your responses*]

f) At this point, your **History** should show:

```
 R(0x2a) = M
 W(0x1b, 0xb1) = M
 R(0x01) = M
 W(0x1c, 0xe9) = M
>
```

Append the bolded text below so that your History looks like:

```
 R(0x2a) = M
 W(0x1b, 0xb1) = M
 R(0x01) = M
 W(0x1c, 0xe9) = M
> W(0x03, 0xff)
 R(0x27)
 R(0x10)
 W(0x1d, 0x00)
```

[*Click "Load." You'll notice that "* `= ?`*" is appended to each of these new memory accesses*]

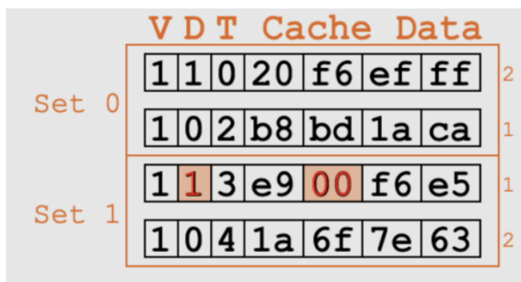
Predict if '`?`' will resolve to Hit (`H`) or Miss (`M`) for each of the new accesses:

i) `W(0x03, 0xff) =` ______ ii) `R(0x27) =` ______

iii) `R(0x10) =` ______ iv) `W(0x1d, 0x00) =` ______

[*Click the down arrow (↓) to verify your responses for each access*]

g) The cache, after the 8 executions detailed above, should look like this:



The small numbers on the right (outside of the sets) indicate how recently used each line is within the set, with smaller numbers being *more recently* used).

i) An **LRU** replacement policy will evict which block on the next conflict in set 0? Line 1 Line 2

ii) What is one benefit of using **LRU** over **Random**?

iii) What is one benefit of using **Random** over **LRU**?

h) If we were to flush the cache right now how many bytes in memory would change? ______

How many bytes would change if we were using **Write Through** instead of **Write Back**? ______

Can you explain why these numbers are the same/different? (if not, try changing the write hit policy and re-running using the history above).