CS 449 Rec

cache

Announcements

- HW 6 is out
  - Due Nov. 6
- Cache lab is out
  - Due Nov. 12

Schedule

- Cache review
- Quizzes for the lab
- Cache tracing
- Next week: Cache lab

(from textbook)

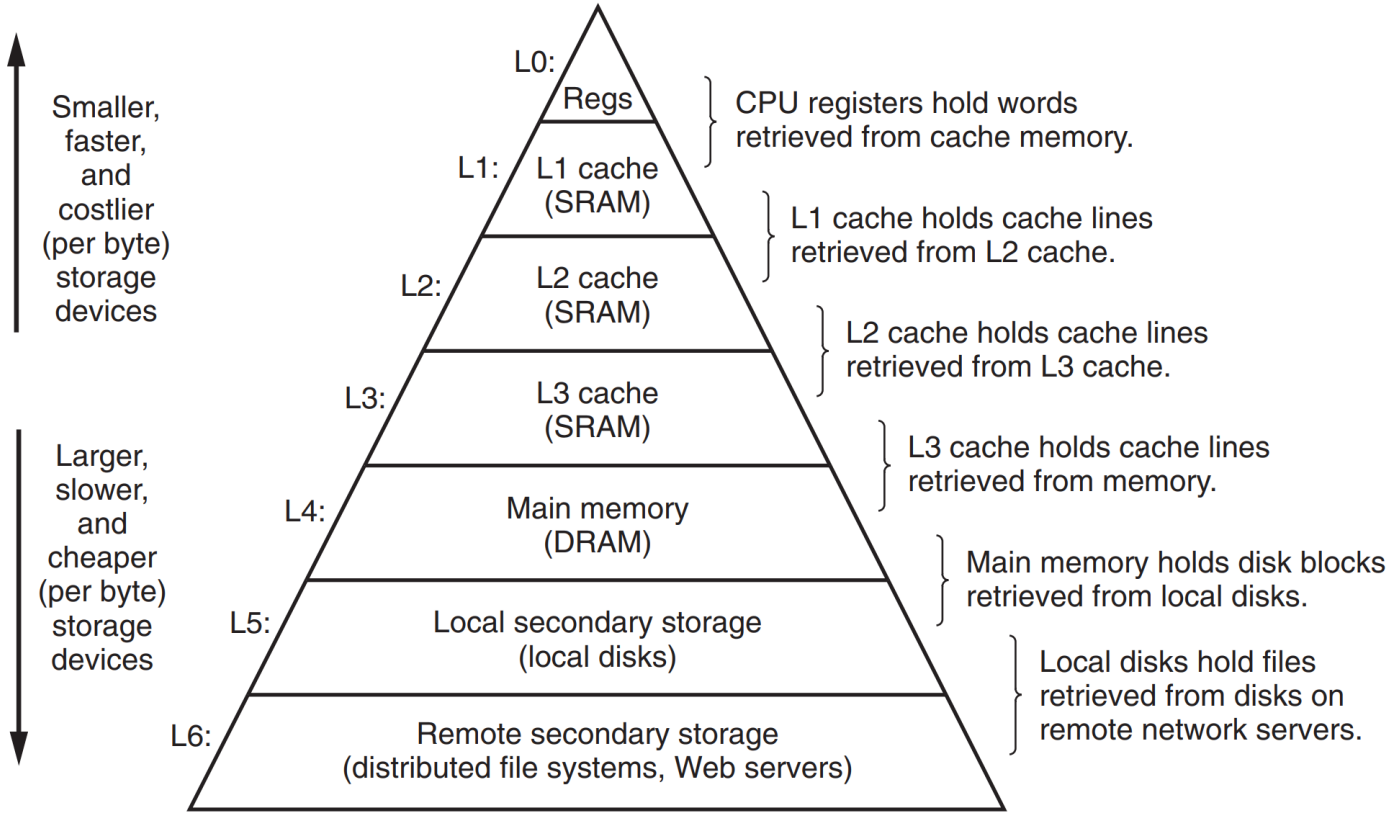
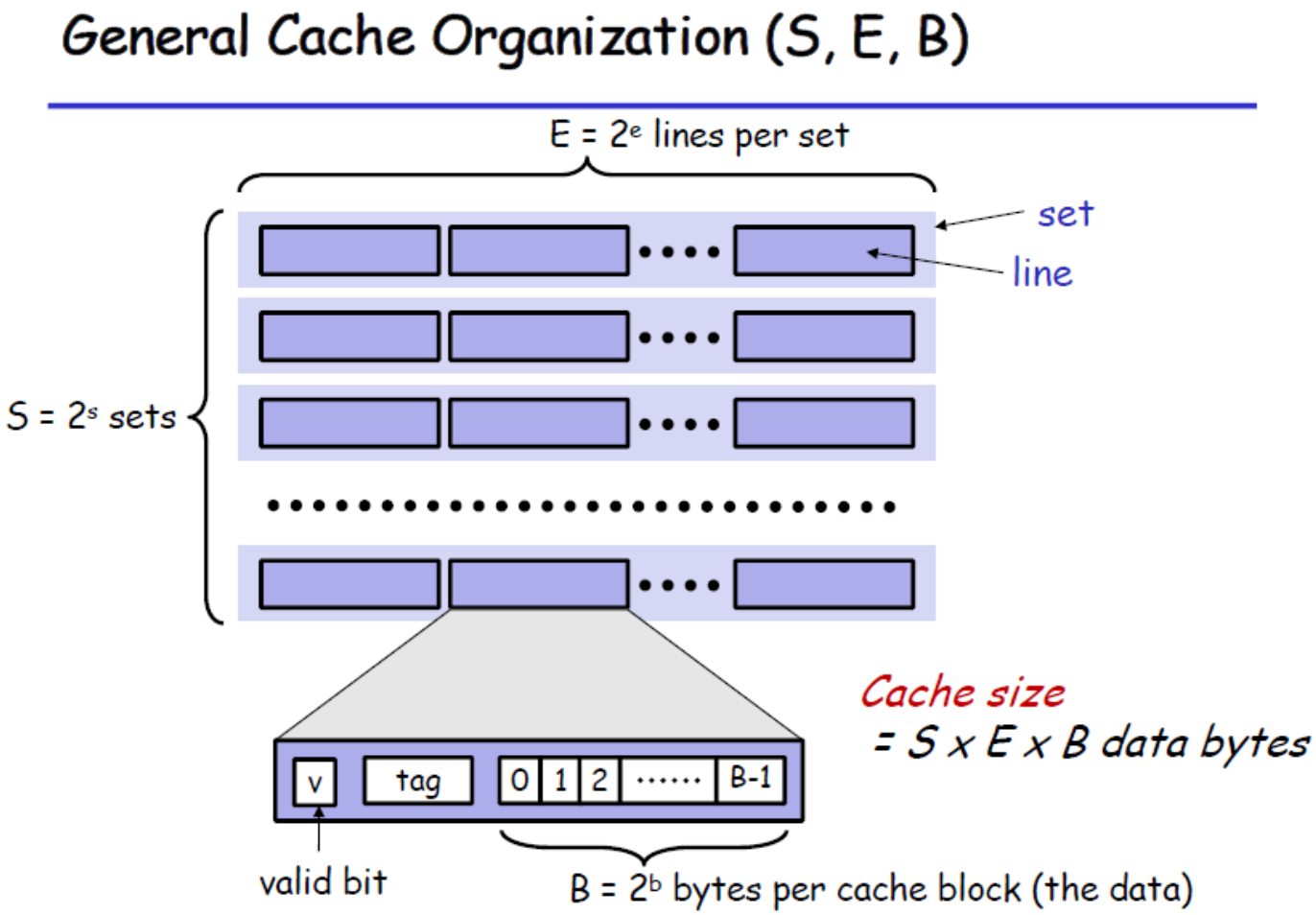


Figure 6.23 **The memory hierarchy.**

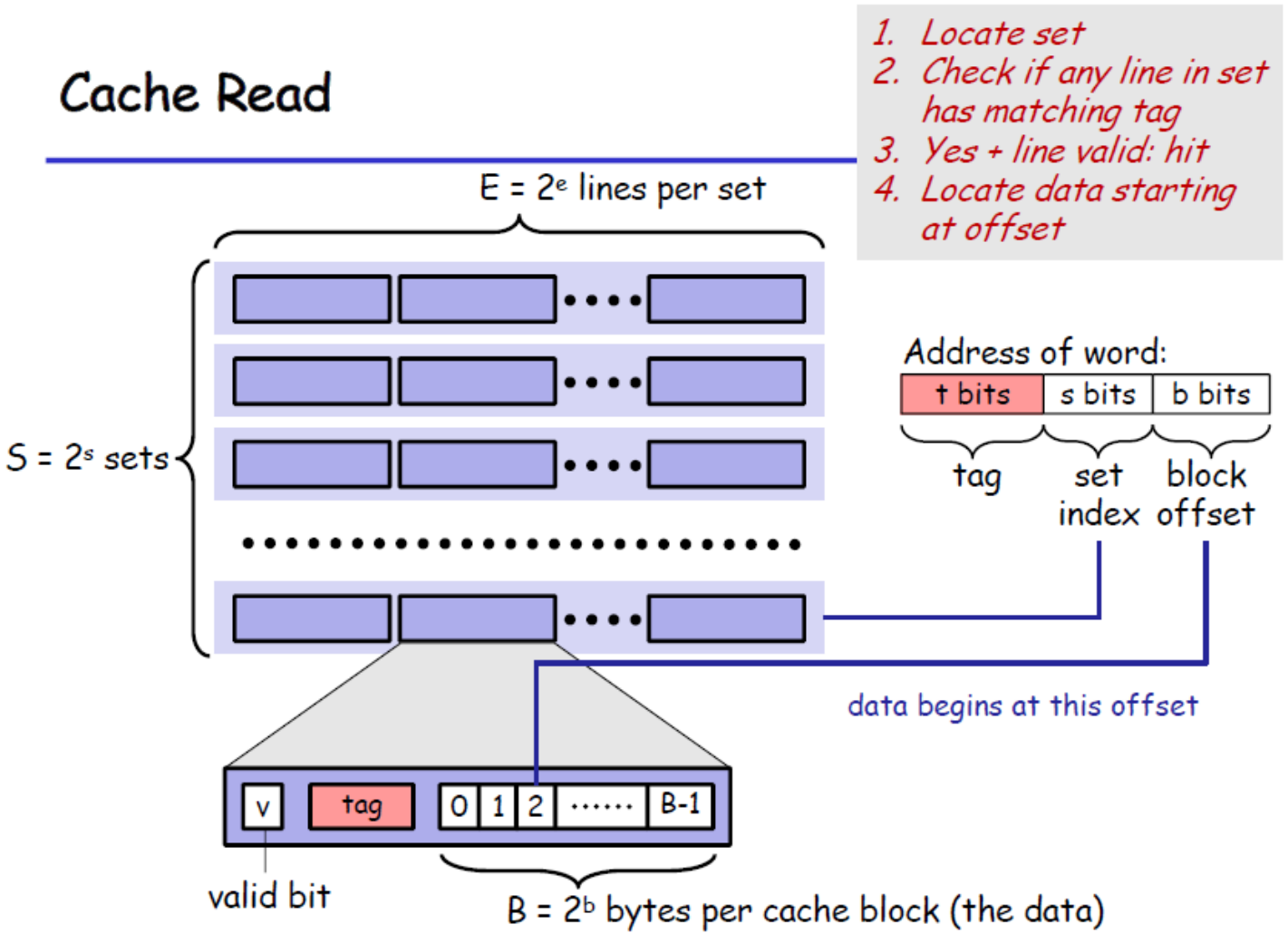
Cache review (from lecture)



Cache basics

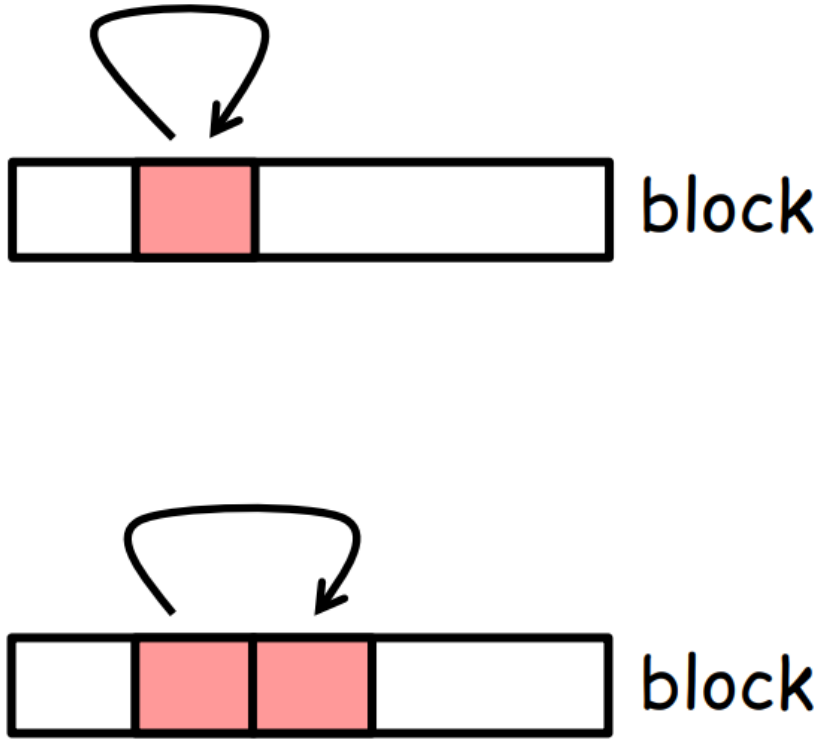
- Cache -> sets -> lines -> blocks (with bytes)
- Address of words: | t bits | s bits | b bits |
  - | tag | set index | block offset |
  - Word size = t+s+b
  - 2^(t+s+b) addresses
- How to locate data?

(from lecture)

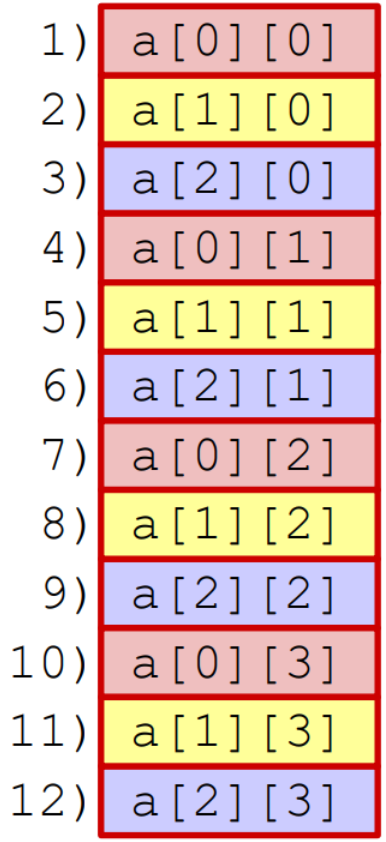


Locality

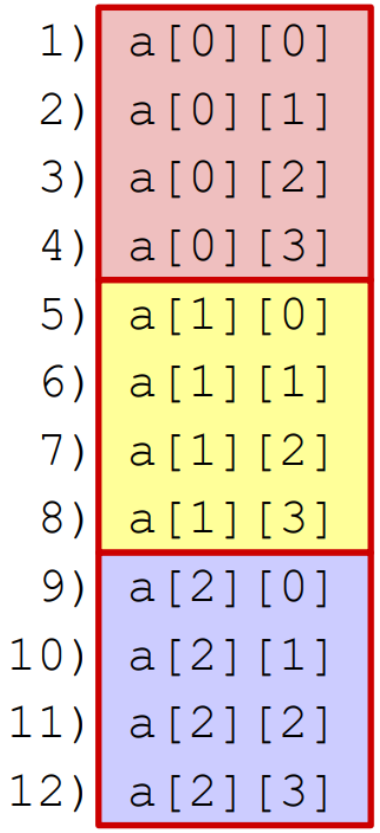
- Temporal
  - a memory location that is referenced once is likely to be referenced again multiple times in the near future
  - **Repeated references** to variables are good
- Spatial
  - the program is likely to reference a nearby memory location in the near future
  - **Stride-1 reference** patterns are good



References (spatial locality)



VS



Hit/Miss

- Hit Rate (HR) = hits/accesses
- Miss Rate (MR) = misses/accesses = 1 - Hit Rate
- Hit Time (HT) = Time to deliver a block in the cache to the processor
  - Includes time to check hit/miss
- Miss Penalty (MP) = Additional time required because of a miss
- Average Memory Access Time (AMAT) = HT + MR × MP

(from textbook)

Parameter	Description
Fundamental parameters	
$S = 2^s$	Number of sets
E	Number of lines per set
$B = 2^b$	Block size (bytes)
$m = \log_2(M)$	Number of physical (main memory) address bits
Derived quantities	
$M = 2^m$	Maximum number of unique memory addresses
$s = \log_2(S)$	Number of *set index bits*
$b = \log_2(B)$	Number of *block offset bits*
$t = m - (s + b)$	Number of *tag bits*
$C = B \times E \times S$	Cache size (bytes), not including overhead such as the valid and tag bits

Figure 6.26 Summary of cache parameters.

Quizzes

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 =	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 =

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

Direct mapped

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)

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: _____

Index bits: _____

Tag bits: _____

	Hit or Miss?	Data returned
a) Read 1 byte at 0x7AC		
b) Read 1 byte at 0x024		
c) Read 1 byte at 0x99F		

Answer

- # of sets = 16
- # of bytes per block = 4
- s = $\log_2 16 = 4$
- b = $\log_2 4 = 2$
- t = 12 – s - b = 6

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

Answer

a) t = 1E; s = B; b = 0

b) t = 0; s = 9; b = 0

c) t = 26; s = 7; b = 3

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

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: _____

Index bits: _____

Tag bits: _____

	Hit or Miss?	Data returned
a) Read 1 byte at 0x435		
b) Read 1 byte at 0x388		
c) Read 1 byte at 0x0D3		

Answer

- # of sets = 8
- # of bytes per block = 4
- s = $\log_2 8 = 3$
- b = $\log_2 4 = 2$
- t = 12 – s - b = 7

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

Answer

a) t = 21; s = 5; b = 1

b) t = 1C; s = 2; b = 0

c) t = 6; s = 4; b = 3

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

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: _____

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		

Answer

- # of sets = 1
- # of bytes per block = 4
- s = $\log_2 1 = 0$
- b = $\log_2 4 = 2$
- t = 12 – s - b = 10

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

Answer

a) t = 77; s = 0; b = 1

b) t = 1C6; s = 0; b = 1

c) t = AA; s = 0; b = 2

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	—

Miss rate

Code Analysis

Consider the following code that accesses a two-dimensional array (of size 64×64 `int`s). 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?

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

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

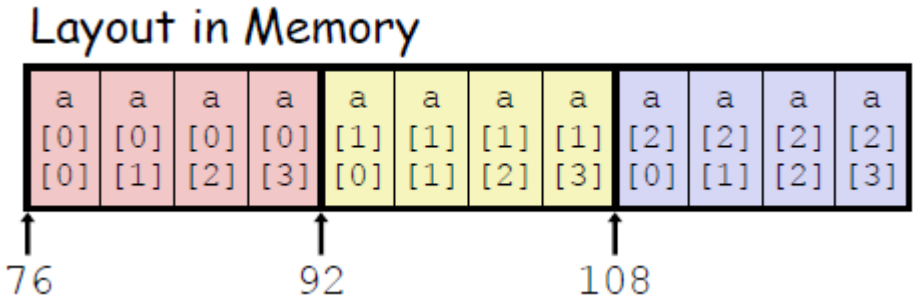
Code Analysis

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 $\frac{4\ bytes\ per\ int}{16\ bytes\ per\ block} = \frac{1\ block}{4\ ints} = 0.25 = 25\%$

→ Assume cache starts empty
→ Remember how 2D arrays are laid out in memory (image from lecture slides, for a 3 x 4 array). The miss rate would be higher if we iterated through rows in each column.



→ Miss rate = #misses/#references, we'll miss once for every 4 ints we read in

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
```

b) What code modifications can change 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.

→ See comments on loop switching on previous slide

→ Switching to a narrower/wider datatype would imply that more/fewer values would fit within each block, implying a lower/higher miss rate.

→ Linked lists would make the situation less predictable (memory locations might not be contiguous…)

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
```

c) What cache parameter changes (size, associativity, block size) can change 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

Summary: A larger cache won't improve the miss rate, since we need to fetch each new block regardless.

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
```

c) What cache parameter changes (size, associativity, block size) can change 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.

Summary: Since we're moving through contiguous locations in memory, the new blocks will map onto different sets (switching to 2-way associative would make more of a difference if we were frequently pulling in blocks with conflicting set indices).

Code Analysis

Consider the following code that accesses a two-dimensional array (of size 64×64 `int`s).
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
```

c) What cache parameter changes (size, associativity, block size) can change 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)

Summary: Larger blocks → less frequent fetches from lower levels of memory hierarchy → lower miss rate.

Code Analysis

Consider the following code that accesses a two-dimensional array (of size 64×64 `int`s).
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
```

c) What cache parameter changes (size, associativity, block size) can change 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

https://www.cs.pitt.edu/~vinicius/cachesim/

Cache Simulator

System Parameters:

Address width: 8 bits
Cache size: 32 bytes
Block size: 2 4 8 bytes
Associativity: 1 2 4 way(s)
Write Hit: Write back
Write Miss: Write-allocate
Replacement: Least Recently Used

Generate System
Explain

Manual Memory Access:

Explain
Read Addr: 0x
Write Addr: 0x , Byte: 0x
Flush

Tag	Index	Offset	Cache Hits	Cache Misses
–	–	–	–	–

Simulation Messages:

History:

>

Load || ↑ ↓

Cache Simulator

https://www.cs.pitt.edu/~vinicius/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.

Practice questions

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:

i) Highest memory address: 0b_____ ii) Number of sets in cache: _____

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

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

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

iii) The 4 bytes of *data* in this block are (in order): 0x_____, 0x_____, 0x_____, 0x_____

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

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

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

[*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? _____

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

(cont.)

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*]

(cont.)

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)` = _____

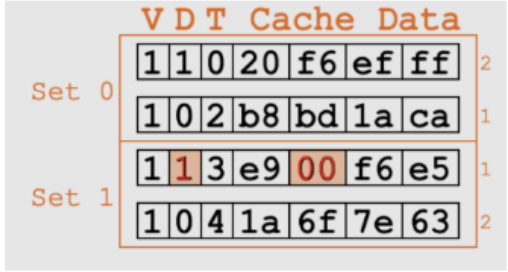
iii) `R(0x10)` = _____

iv) `W(0x1d, 0x00)` = _____

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

(cont.)

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

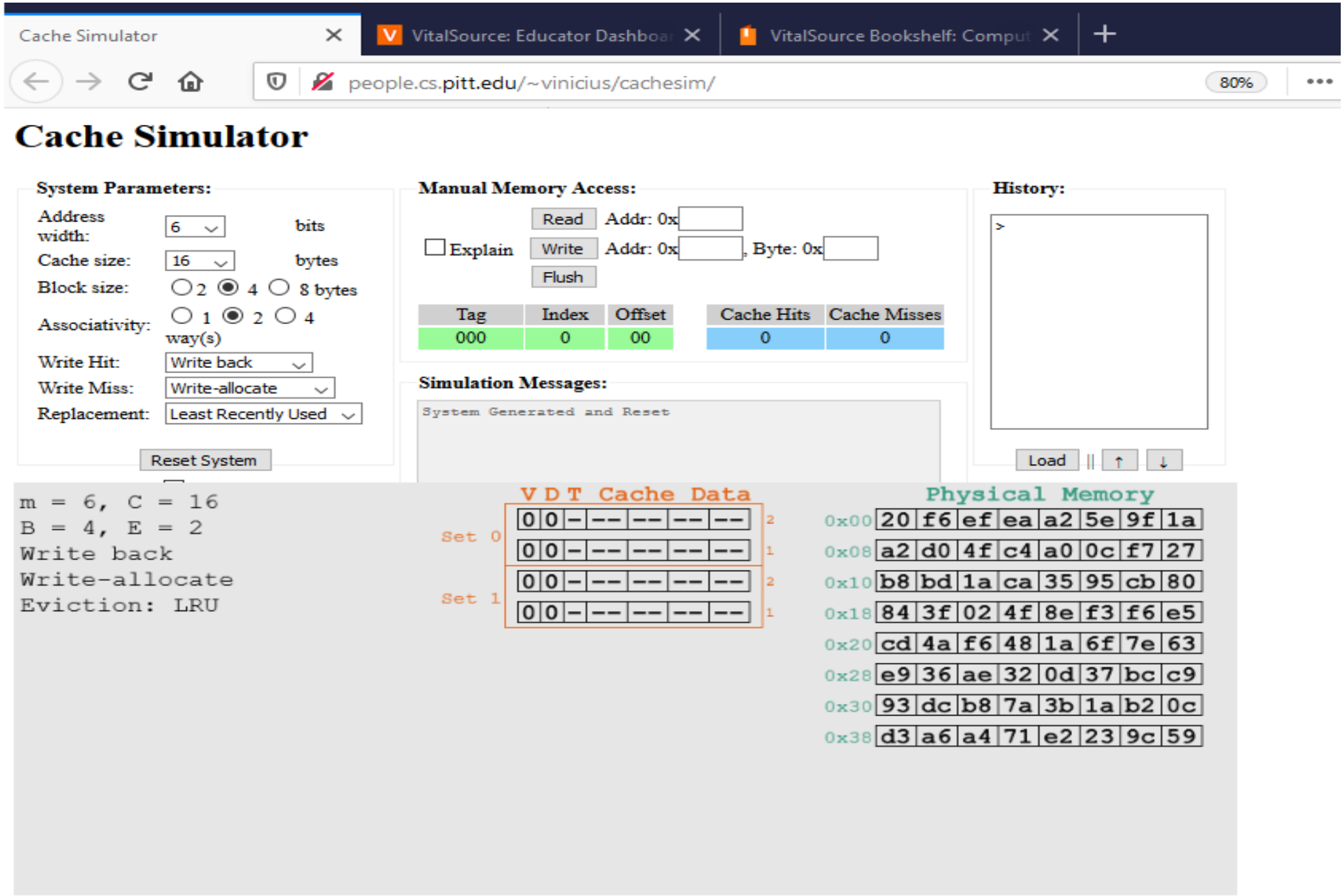
ii) What is one benefit of using **LRU** over **Random**?

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).


Cache Simulator
VitalSource: Educator Dashboa
VitalSource Bookshelf: Comput
people.cs.pitt.edu/~vinicius/cachesim/
80%
Cache Simulator
System Parameters:
Address width: 6 bits
Cache size: 16 bytes
Block size: 2 4 8 bytes
Associativity: 1 2 4 way(s)
Write Hit: Write back
Write Miss: Write-allocate
Replacement: Least Recently Used
Reset System
Manual Memory Access:
Read Addr: 0x
Explain
Write Addr: 0x , Byte: 0x
Flush
Tag Index Offset Cache Hits Cache Misses
000 0 00 0 0
Simulation Messages:
System Generated and Reset
History:
>
Load
m = 6, C = 16
B = 4, E = 2
Write back
Write-allocate
Eviction: LRU
V D T Cache Data
Set 0
0 0 - -- -- -- -- 2
0 0 - -- -- -- -- 1
Set 1
0 0 - -- -- -- -- 2
0 0 - -- -- -- -- 1
Physical Memory
0x00 20 f6 ef ea a2 5e 9f 1a
0x08 a2 d0 4f c4 a0 0c f7 27
0x10 b8 bd 1a ca 35 95 cb 80
0x18 84 3f 02 4f 8e f3 f6 e5
0x20 cd 4a f6 48 1a 6f 7e 63
0x28 e9 36 ae 32 0d 37 bc c9
0x30 93 dc b8 7a 3b 1a b2 0c
0x38 d3 a6 a4 71 e2 23 9c 59

Cache Simulator | VitalSource: Educator Dashboa | VitalSource Bookshelf: Comput | Live Stream | Governor T

people.cs.pitt.edu/~vinicius/cachesim/ 80%

Cache Simulator

System Parameters:

Address width: 6 bits
Cache size: 16 bytes
Block size: ○ 2 ◉ 4 ○ 8 bytes
Associativity: ○ 1 ◉ 2 ○ 4 way(s)
Write Hit: Write back
Write Miss: Write-allocate
Replacement: Least Recently Used

Reset System

Manual Memory Access:

☐ Explain
Read Addr: 0x 2A
Write Addr: 0x , Byte: 0x
Flush

Tag	Index	Offset	Cache Hits	Cache Misses
101	0	10	0	1

Simulation Messages:

```
Invalid Line 0 chosen for replacement.
Block read into cache from memory at address 0x28.
LRU statuses updated.
Data: 0xae
```

History:

```
R(0x2a) = M
>
```

Load || ↑ ↓

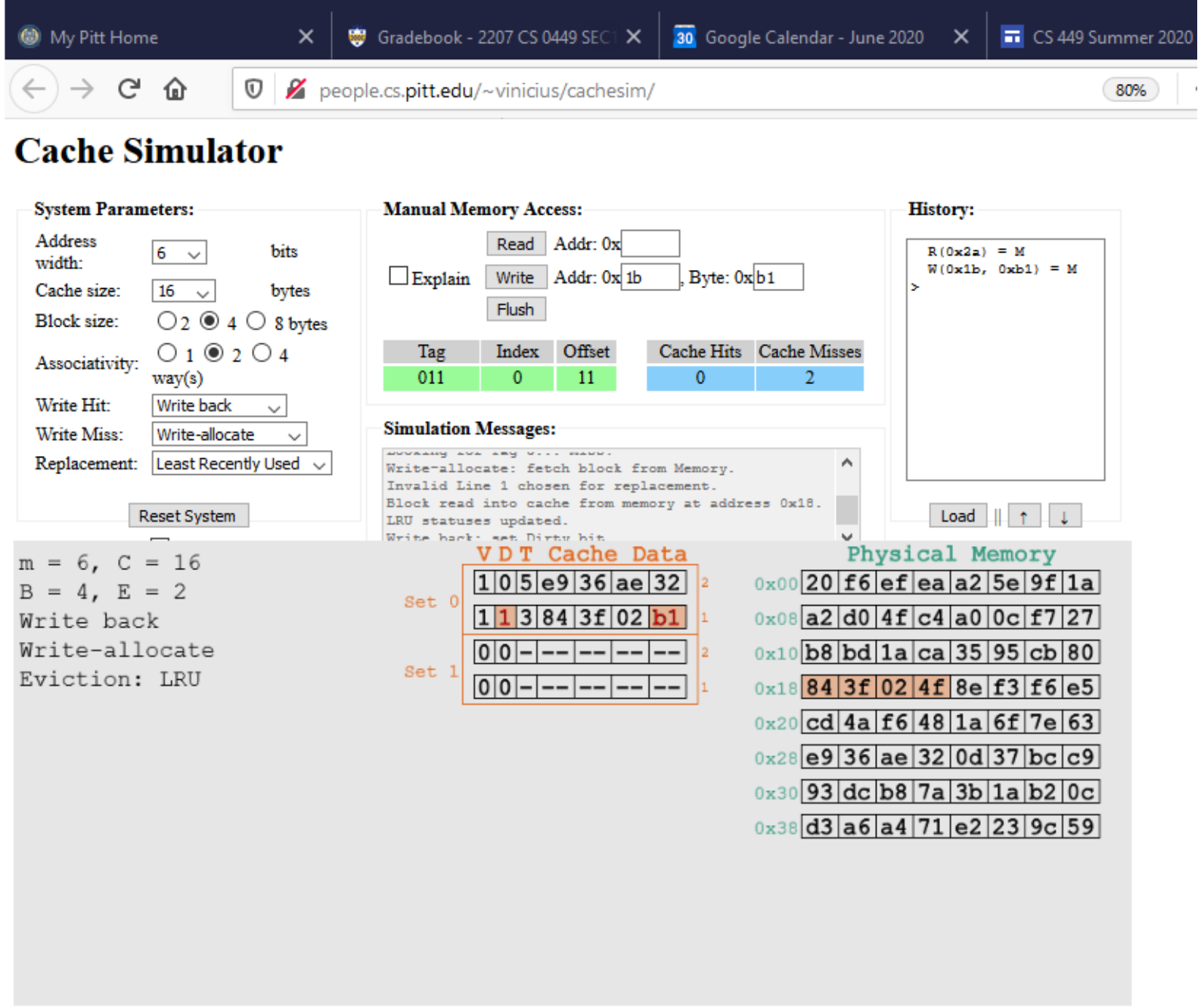
```
m = 6, C = 16
B = 4, E = 2
Write back
Write-allocate
Eviction: LRU
```

V D T Cache Data

		V	D	T	Data	
Set 0	1	1	0	5	e9 36 ae 32	1
	2	0	0	-	-- -- -- --	2
Set 1	1	0	0	-	-- -- -- --	2
	2	0	0	-	-- -- -- --	1

Physical Memory

0x00	20	f6	ef	ea	a2	5e	9f	1a
0x08	a2	d0	4f	c4	a0	0c	f7	27
0x10	b8	bd	1a	ca	35	95	cb	80
0x18	84	3f	02	4f	8e	f3	f6	e5
0x20	cd	4a	f6	48	1a	6f	7e	63
0x28	e9	36	ae	32	0d	37	bc	c9
0x30	93	dc	b8	7a	3b	1a	b2	0c
0x38	d3	a6	a4	71	e2	23	9c	59


My Pitt Home
Gradebook - 2207 CS 0449 SEC
Google Calendar - June 2020
CS 449 Summer 2020
people.cs.pitt.edu/~vinicius/cachesim/
80%
Cache Simulator
System Parameters:
Address width: 6 bits
Cache size: 16 bytes
Block size: 2 4 8 bytes
Associativity: 1 2 4 way(s)
Write Hit: Write back
Write Miss: Write-allocate
Replacement: Least Recently Used
Reset System
Manual Memory Access:
Read Addr: 0x
Explain
Write Addr: 0x 1b , Byte: 0x b1
Flush
Tag Index Offset Cache Hits Cache Misses
011 0 11 0 2
History:
R(0x2a) = M
W(0x1b, 0xb1) = M
>
Load
Simulation Messages:
Write-allocate: fetch block from Memory.
Invalid Line 1 chosen for replacement.
Block read into cache from memory at address 0x18.
LRU statuses updated.
m = 6, C = 16
B = 4, E = 2
Write back
Write-allocate
Eviction: LRU
V D T Cache Data
Set 0
1 0 5 e9 36 ae 32 2
1 1 3 84 3f 02 b1 1
Set 1
0 0 - -- -- -- -- 2
0 0 - -- -- -- -- 1
Physical Memory
0x00 20 f6 ef ea a2 5e 9f 1a
0x08 a2 d0 4f c4 a0 0c f7 27
0x10 b8 bd 1a ca 35 95 cb 80
0x18 84 3f 02 4f 8e f3 f6 e5
0x20 cd 4a f6 48 1a 6f 7e 63
0x28 e9 36 ae 32 0d 37 bc c9
0x30 93 dc b8 7a 3b 1a b2 0c
0x38 d3 a6 a4 71 e2 23 9c 59

people.cs.pitt.edu/~vinicius/cachesim/ 80%

Cache Simulator

System Parameters:

Address width: 6 bits
Cache size: 16 bytes
Block size: 2 **4** 8 bytes
Associativity: 1 **2** 4 way(s)
Write Hit: Write back
Write Miss: Write-allocate
Replacement: Least Recently Used

Reset System

Manual Memory Access:

Read Addr: 0x01
Explain Write Addr: 0x , Byte: 0x
Flush

Tag	Index	Offset	Cache Hits	Cache Misses
000	0	01	0	3

Simulation Messages:

```
Not dirty, so no need to write back to memory.
Block read into cache from memory at address 0x0.
LRU statuses updated.
Data: 0xf6
```

History:

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

Load ↑ ↓

```
m = 6, C = 16
B = 4, E = 2
Write back
Write-allocate
Eviction: LRU
```

V D T Cache Data

		V	D	T	Cache Data	
Set 0	1	1	0	20	f6 ef ea	1
	2	1	1	84	3f 02 b1	2
Set 1	2	0	0	-	-- -- -- --	2
	1	0	0	-	-- -- -- --	1

Physical Memory

0x00	20	f6	ef	ea	a2	5e	9f	1a
0x08	a2	d0	4f	c4	a0	0c	f7	27
0x10	b8	bd	1a	ca	35	95	cb	80
0x18	84	3f	02	4f	8e	f3	f6	e5
0x20	cd	4a	f6	48	1a	6f	7e	63
0x28	e9	36	ae	32	0d	37	bc	c9
0x30	93	dc	b8	7a	3b	1a	b2	0c
0x38	d3	a6	a4	71	e2	23	9c	59

Cache tracing

- Load vs Store
  - L: from memory to cache (read)
  - S: from cache to memory (write)
- Trace file (.trace); e.g. L 0,4 S 8,4

Example

(Summary from lecture)

Memory Hierarchy

- Successively higher levels contain “most used” data from lower levels
- Exploits temporal and spatial locality
- Caches are intermediate storage levels used to optimize data transfers between any system elements with different characteristics

Cache Performance

- Ideal case: found in cache (hit)
- Bad case: not found in cache (miss), search in next level
- Average Memory Access Time (AMAT) = HT + MR × MP
  - Hurt by Miss Rate and Miss Penalty