15-213 Recitation 7: Caches and Blocking

10 Oct 2016

Agenda

- Reminders
- Revisiting cachelab
- Caching Review
- Blocking to reduce cache misses

Reminders

- Cache Lab is due **Thursday!**
- Exam1 is just a week away!
- Start doing practice problems.
- Come to the review session.

Reminders: Cache Lab

- Two parts
  - Write a cache simulator – hopefully you've started this part by now
  - Optimize some code to minimize cache misses – we'll talk about this today
- Programming style will be graded starting now
  - Worth about a letter grade on this assignment
  - Summary slide is included as an appendix to this recitation and covered in last week's recitation, but be sure to **carefully** read the style guide
- Details are in the writeup!

Cache Lab: Cache Simulator Hints

- You are simply **counting** hits, misses, and evictions
- Use LRU (Least Recently Used) replacement policy
- Structs are a great way to bundle up the different parts of a cache line (valid bit, tag, LRU counter, etc.)
- A cache is just a 2D array of *cache lines*
- one dimension represents associativity E, the other the number of sets S:

```
struct cache_line cache[S][E];
```

- Your simulator needs to handle different values of S, E, and b (block size) given at run time

Cache Lab: Parsing Input with fscanf

- fscanf() is exactly like scanf() except that you specify a stream to use (i.e. an open file) instead of always reading from standard input

- The parameters to fscanf are
  1. a stream pointer of type FILE*, e.g. from fopen()
  2. a format string specifying how to parse the input

  3-n. a **pointer** to each of the variables that will store the parsed data

- fscanf() returns -1 if the data does not match the format string or there is no more input
- Use it to parse the trace files

fscanf() Example

```
FILE *pFile; /* pointer to FILE object */

pFile = fopen("trace.txt","r"); /* open trace file for reading */
/* verify that pFile is non-NULL! */

char access_type;
unsigned long address;
int size;

/* line format is " S 2f,1" or " L 7d0,3" */
/* so we need to read a character, a hex number, and a decimal number */
/* put those in the format string along with the fixed formatting */
while (fscanf(pFile," %c %lx,%d", &access_type, &address, &size) > 0) {
   /* do stuff */
}

fclose(pFile);  /* always close file when done */
```

Class Question / Discussions

- We’ll work through a series of questions
  - For each, take a minute and write down your answer
  - Then discuss with your classmates

Questions based on ones provided by Prof Sat Garcia, USD

What Type of Locality?

The following function exhibits which type of locality? Consider *only* array accesses.

```
void who(int *arr, int size) {
  for (int i = 0; i < size-1; ++i)
    arr[i] = arr[i+1];
}
```

A.	Spatial
B.	Temporal
C.	Both A and B
D.	Neither A nor B

What Type of Locality?

The following function exhibits which type of locality? Consider *only* array accesses.

```
void who(int *arr, int size) {
  for (int i = 0; i < size-1; ++i)
    arr[i] = arr[i+1];
}
```

A.	Spatial
B.	Temporal
C.	Both A and B
D.	Neither A nor B

What Type of Locality?

The following function exhibits which type of locality? Consider *only* array accesses.

```
void coo(int *arr, int size) {
  for (int i = size-2; i >= 0; --i)
    arr[i] = arr[i+1];
}
```

A.	Spatial
B.	Temporal
C.	Both A and B
D.	Neither A nor B

What Type of Locality?

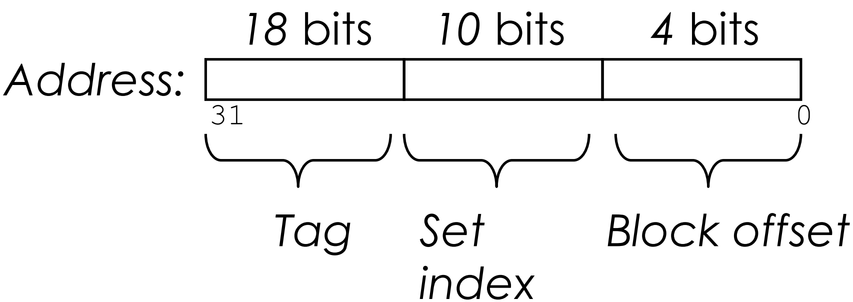
The following function exhibits which type of locality? Consider *only* array accesses.

```
void coo(int *arr, int size) {
  for (int i = size-2; i >= 0; --i)
    arr[i] = arr[i+1];
}
```

A.	Spatial
B.	Temporal
C.	Both A and B
D.	Neither A nor B

Calculating Cache Parameters

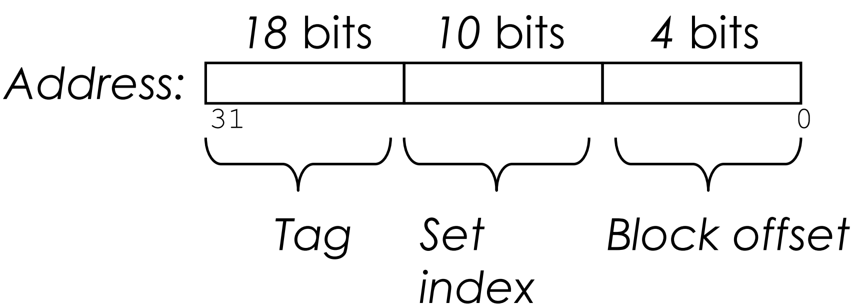
Given the following address partition, how many `int` values will fit in a single data block?



	# of int in block
A.	0
B.	1
C.	2
D.	4
E.	Unknown: We need more info

Calculating Cache Parameters

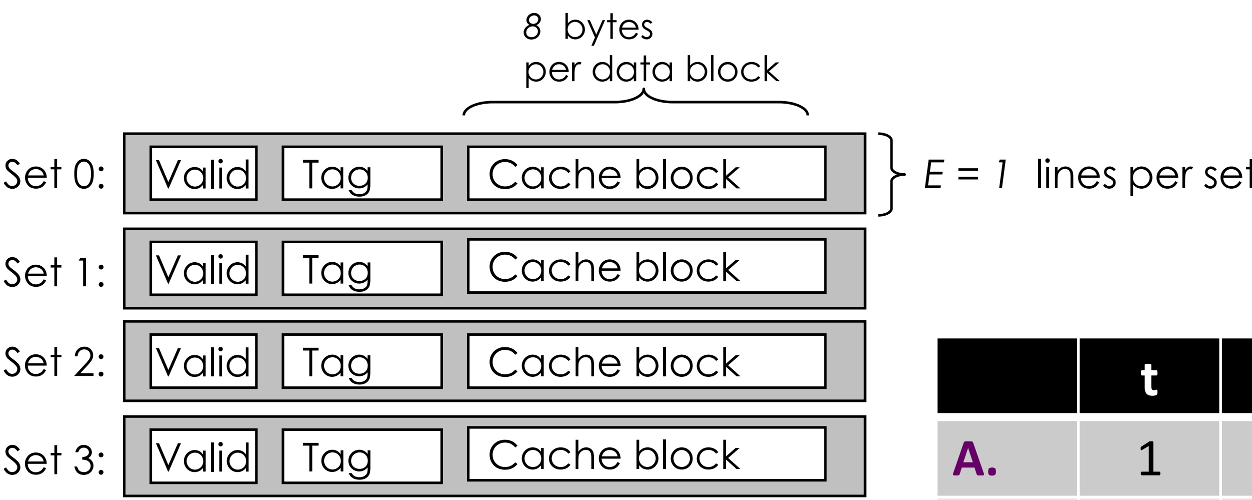
Given the following address partition, how many `int` values will fit in a single data block?

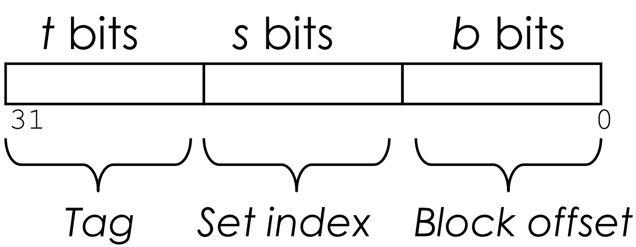


	# of int in block
A.	0
B.	1
C.	2
D.	4
E.	Unknown: We need more info

Direct-Mapped Cache Example

Assuming a 32-bit address (i.e. m=32), how many bits are used for tag (t), set index (s), and block offset (b).

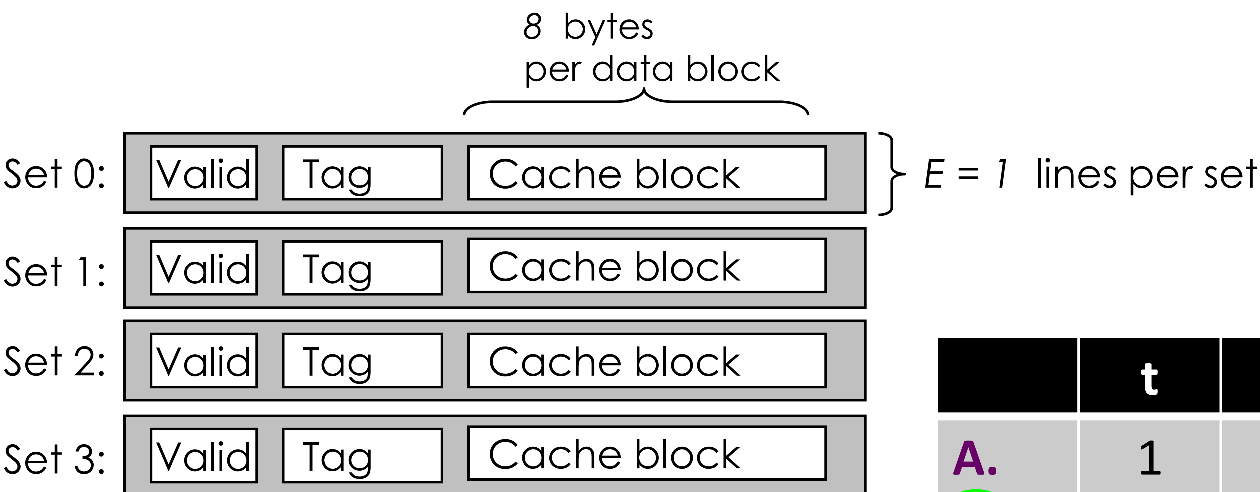


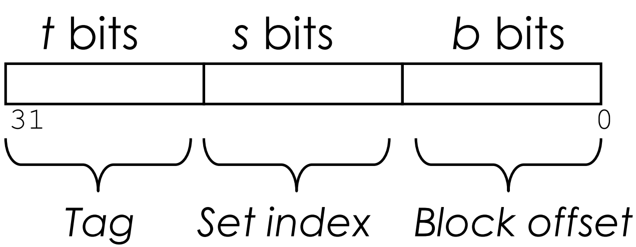


	t	s	b
A.	1	2	3
B.	27	2	3
C.	25	4	3
D.	1	4	8
E.	20	4	8

Direct-Mapped Cache Example

Assuming a 32-bit address (i.e. m=32), how many bits are used for tag (t), set index (s), and block offset (b).

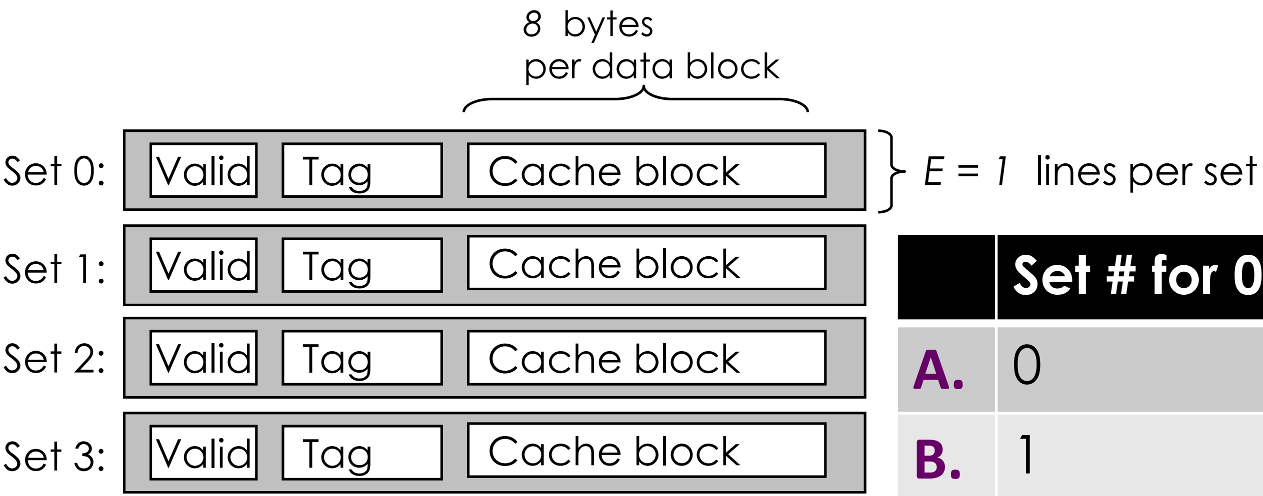




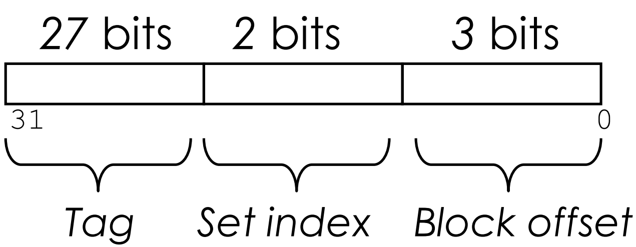
	t	s	b
A.	1	2	3
B.	27	2	3
C.	25	4	3
D.	1	4	8
E.	20	4	8

Which Set Is it?

Which set may the address **0xFA1C** be located in?

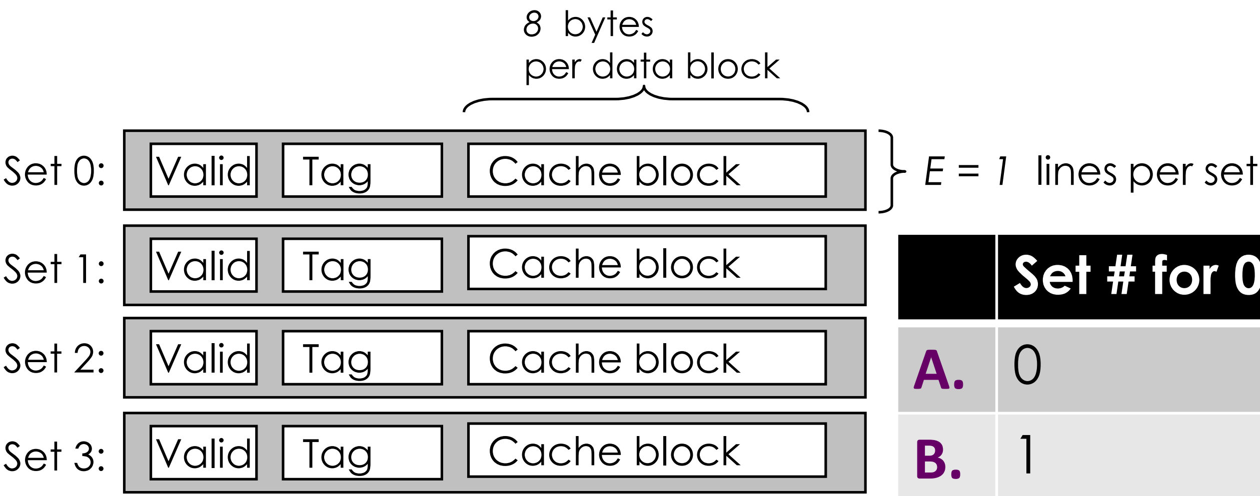


	Set # for 0xFA1C
A.	0
B.	1
C.	2
D.	3
E.	More than one of the above

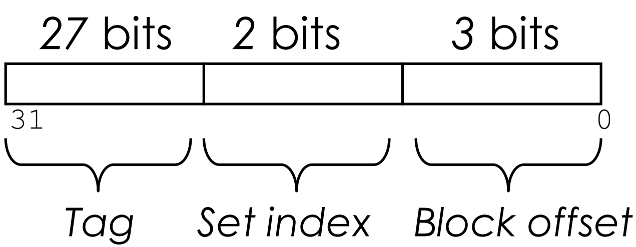


Which Set Is it?

Which set may the address **0xFA1C** be located in?

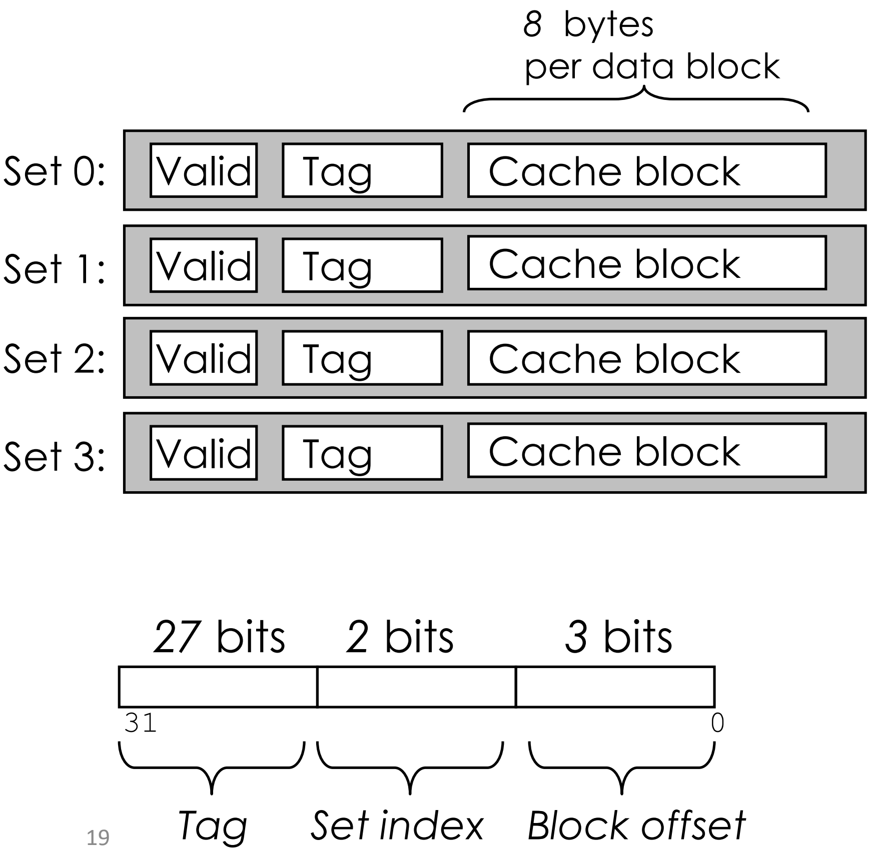


	Set # for 0xFA1C
A.	0
B.	1
C.	2
D.	3
E.	More than one of the above



Cache Block Range

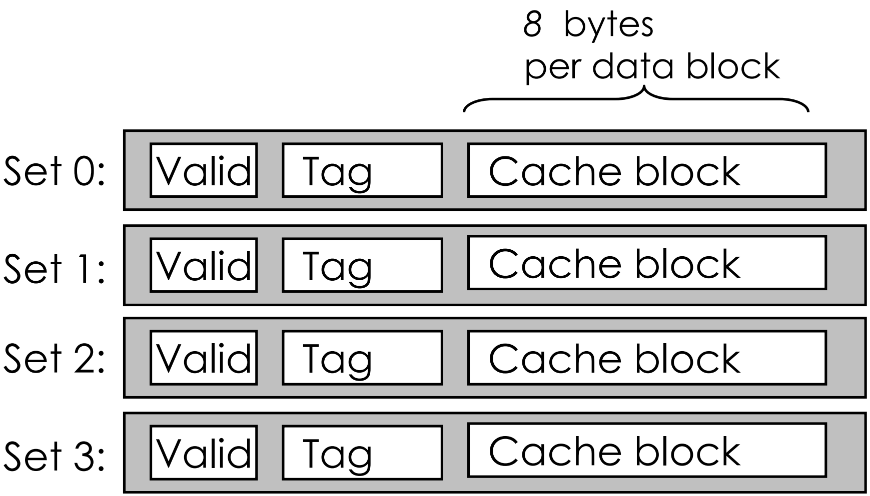
What range of addresses will be in the same block as address **0xFA1C**?

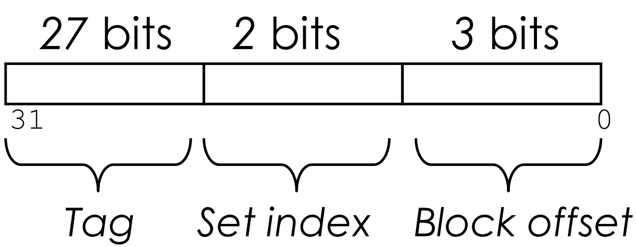


	Addr. Range
A.	0xFA1C
B.	0xFA1C – 0xFA23
C.	0xFA1C – 0xFA1F
D.	0xFA18 – 0xFA1F
E.	It depends on the access size (byte, word, etc)

Cache Block Range

What range of addresses will be in the same block as address **0xFA1C**?





	Addr. Range
A.	0xFA1C
B.	0xFA1C – 0xFA23
C.	0xFA1C – 0xFA1F
D.	0xFA18 – 0xFA1F
E.	It depends on the access size (byte, word, etc)

Cache Misses

If N = 16, how many bytes does the loop access of A?

```
int foo(int* a, int N)
{
    int i, sum = 0;
    for(i = 0; i < N; i++)
        sum += a[i];
    return sum;
}
```

	Accessed Bytes
A	4
B	16
C	64
D	256

Cache Misses

If N = 16, how many bytes does the loop access of A?

```
int foo(int* a, int N)
{
    int i, sum = 0;
    for(i = 0; i < N; i++)
        sum += a[i];
    return sum;
}
```

	Accessed Bytes
A	4
B	16
C	64
D	256

Cache Misses

If there is a 48B cache with 8 bytes per block and two blocks per set, how many misses if foo is called twice? N still equals 16

```
int foo(int* a, int N)
{
    int i, sum = 0;
    for(i = 0; i < N; i++)
        sum += a[i];
    return sum;
}
```

	Misses
A	0
B	8
C	12
D	14
E	16

Cache Misses

If there is a 48B cache with 8 bytes per block and two blocks per set, how many misses if foo is called twice? N still equals 16

```
int foo(int* a, int N)
{
    int i, sum = 0;
    for(i = 0; i < N; i++)
        sum += a[i];
    return sum;
}
```

	Misses
A	0
B	8
C	12
D	14
E	16

Cache-Friendly Code

- Keep memory accesses bunched together
    - in both time and space (address)
    - the *working set* at any time should be smaller than the cache
- Avoid access patterns that cause conflict misses
    - memory *strides* in powers of two that cause all accesses to use only a few (or just one!) cache set

Blocking Example

- We have a 2D array of 16 elements.
  - Cache is fully associative and can hold two lines
  - Each line can hold two elements

- Discuss the following questions with your neighbor.

- What is the best miss rate for traversing the array once?
  - What order does of traversal did you use?

- What other traversal orders can achieve this miss rate?

Class Discussion

- When comparing between the other traversal orders, what did they have in common?

If You Get Stuck

Please read the writeup.

Please read the writeup.

<u>Please read the writeup.</u>

Please read the writeup!

- CS:APP Chapter 6
- View lecture notes and course FAQ at http://www.cs.cmu.edu/~213
- Office hours Sunday through Thursday 5:00-9:00pm in WeH 5207
- Post a **private** question on Piazza
- `man malloc`, `man gdb`, gdb's `help` command
- http://csapp.cs.cmu.edu/public/waside/waside-blocking.pdf

If I had a penny for every time someone

asked a question answered in the writeup....

Appendix: C Programming Style

- Properly document your code
  - Header comments, overall operation of large blocks, any tricky bits
- Write robust code – check error and failure conditions
- Write modular code
  - Use interfaces for data structures, e.g. create/insert/remove/free functions for a linked list
  - No magic numbers – use #define
- Formatting
  - 80 characters per line
  - Consistent braces and whitespace
- No memory or file descriptor leaks