notes originally by Matt Johnson

Caches!

Conceptual Questions: Why do we cache? What is the end result of our caching, in terms of capability?

-We cache because limited memory closer to the chip is faster. Caching gets us the speed of the fast memory with the spatial capacity of the largest memory.

What are temporal and spatial locality? Give high level examples in software of when these occur.

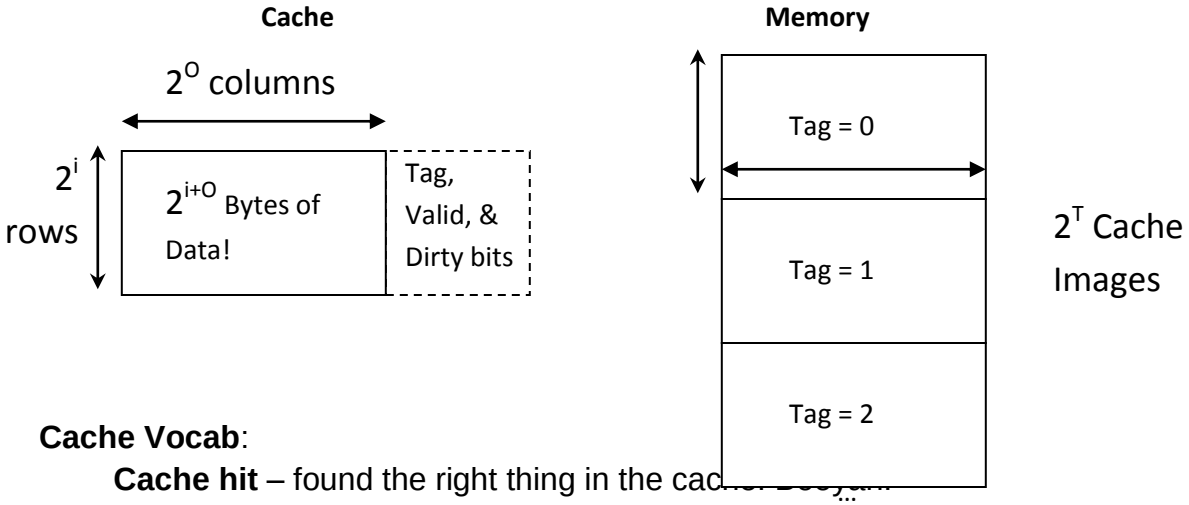
-Temporal locality – We access the same items that have been used recently. Ie., a commonly executed piece of code, such as a menu or library function.
-Spacial locality – We access items nearby other items that have been accessed recently. This is demonstrated in structs or sequential array accesses.

Break up an address:

Tag	**I**ndex	**O**ffset

Offset: "column index" (O bits)
Index: "row index" (i bits)
Tag: "cache number" that the block/row* came from. (T bits) [*difference?]

Segmenting the address into TIO implies a geometrical structure (and size) on our cache. Draw memory with that same geometry!



Cache Vocab:

Cache hit – found the right thing in the cache. Boo-yah!
Cache miss – Nothing in the cache block we checked, so read from memory and write to cache!
Cache miss, block replacement – We found a block, but it had the wrong tag!

Cache Exercises!

C1: Fill this one in… Everything here is Direct-Mapped!

notes originally by Matt Johnson

Address Bits	Cache Size	Block Size	Tag Bits	Index Bits	Offset Bits	Bits per Row
16	4KB	4B	4	10	2	38
16	16KB	8B	2	11	3	68
32	8KB	8B	19	10	3	85
32	32KB	16B	17	11	4	147
32	64KB	16B	16	12	4	146
32	512KB	32B	13	14	5	271
64	1024KB	64B	44	14	6	558
64	2048KB	128B	43	14	7	1069

C2: Assume 16 B of memory and an 8B direct-mapped cache with 2-byte blocks. Classify each of the following memory accesses as hit (H), miss (M), or miss with replacement (R).

a. 4 M
b. 5 H
c. 2 M
d. 6 M
e. 1 M
f. 10 R
g. 7 H
h. 2 R

C3: This composite question was inspired by exam questions but NOT identical since the exam questions use associative caches. Direct-mapped here!

You know you have 1 MiB of memory (maxed out for processor address size) and a 16 KiB cache (data size only, not counting extra bits) with 1 KiB blocks.

```
#define NUM_INTS 8192
int *A = malloc(NUM_INTS * sizeof(int)); // returns address 0x100000
int i, total = 0;
for (i = 0; i < NUM_INTS; i += 128) A[i]   = i; // Line 1
for (i = 0; i < NUM_INTS; i += 128) total += A[i]; // Line 2
```

notes originally by Matt Johnson

a) What is the T:I:O breakup for the cache (assuming byte addressing)?
6:4:10

b) Calculate the hit percentage for the cache for the line marked "**`Line 1`**".
Each step is 512 bytes or 128 ints. There are 256 ints per cache block. Thus we have a 50% hit rate.

c) Calculate the hit percentage for the cache for the line marked "**`Line 2`**".
We covered a 2^13 * 2^2 = 2^15 = 32 KiB array with our first loop, meaning we knocked all our cache entries out in the second pass. Same hit rate as before!

d) How could you optimize the computation?
We could have fewer cache misses if we break up the loops to cover half the array at a time.

Now a completely different setup… Your cache now has 8-byte blocks and 128 rows, and memory has 22 bit addresses. The **`ARRAY_SIZE`** is 4 MiB and **`A`** starts at a block boundary.

```
for (i = 0; i < (ARRAY_SIZE/STRETCH); i += 1) {
    for (j = 0; j < STRETCH; j += 1) sum     += A[i*STRETCH + j];
    for (j = 0; j < STRETCH; j += 1) product *= A[i*STRETCH + j];
}
```

a) What is the T:I:O breakup for the cache (assuming byte addressing)?
12:7:3

b) What is the cache size (data only, no tag and extra bits) in bytes?
d1 KiB

c) What is the largest **`STRETCH`** that minimizes cache misses?
1024 for 1 Kibi chars.

d) Given the **`STRETCH`** size from (c), what is the # of cache misses?
One for every 8-byte block, so 4 Mi / 8 = 512 Kibi misses. Another check would be to say we have 128 blocks in our cache, we miss each block once per outer loop iteration, and we have 4 MiB / 1 KiB = 4 KiB outer loop iterations.

e) Given the **`STRETCH`** size from (c), if **`A`** **does not** start at a block boundary, *roughly* what is the # of *cache misses* for this case to the number you calculated in

notes originally by Matt Johnson

question (d) above? (e.g., 8x, 1/16th)

Now our last block will collide with our first block, meaning the first inner loop has an extra miss and the second inner loop has two misses. Thus where we used to have 128 misses per step of the outer loop, now we have 128+3 = 131 misses.