CSE 351 Section 8 – Lab 4 Preparation

Source of Cache Misses ("3 C's")

- Compulsory / Cold: first access to a particular cache block
  - Parameter fix: Increase the block size
  - Code fix: Make data accessed more compact (*e.g.*, reduce struct size)
- Conflict: cache is large enough, but too many blocks map to the same set
  - Parameter fix: Increase associativity (none in fully-associative caches)
  - Code fix: Change access pattern, use padding
- Capacity: the set of active blocks (working set) is larger than the cache
  - Parameter fix: Increase cache size
  - Code fix: Reduce size of working set / subdivide problem

Lab 4 Part 1

We *strongly recommend* using the Cache Simulator to visualize the behavior of your algorithms for this part of the lab. See the Cache Simulator Tutorial lesson on Ed to get started!

Query a simulated cache in a programmatic way:

- `void flush_cache(void);` – resets cache to cold state
- `bool_t access_cache(addr_t address);` – returns TRUE/1 on a hit, FALSE/0 on a miss (plus updates the cache)

Examples:

- ```
  flush_cache();              // reset cache
  while(!access_cache(0));    // runs 2x: M loads block 0, H exits loop
  ```
- ```
  for(int i = 0; 1; i+=2)     // infinite loop
    access_cache(i);          // access even addresses in increasing order
  ```

Exercise 1: Coding Accesses

Using the Lab 4 Part 1 functions given above, write a C for-loop that generates this specific sequence of accesses:

1) `access_cache(0)`
2) `access_cache(8)`
3) `access_cache(0)`
4) `access_cache(16)`

```
for ( _____ ; _____ ; _____ ) {
```

```
}
```

Exercise 2: Benedict Cumbercache

Given the following sequence of access results (decimal addresses) on a cold/empty cache of size 16 bytes, what can we *deduce* about its properties? Assume an LRU replacement policy.

```
1) access_cache(0)  # False/0
2) access_cache(8)  # False/0
3) access_cache(0)  # True /1
4) access_cache(16) # False/0
5) access_cache(8)  # False/0
```

A. What can we say about the block size (*i.e.*, what range is possible)?

B. Now assume that the block size is 8 bytes. Are the following associativity values possible (*i.e.*, would that cache produce the given access results)? Hint: Draw out each cache and simulate the access pattern.

   a. Can this cache be direct-mapped?

   b. Can this cache be 2-way set associative?

   c. Can this cache be 4-way set associative?

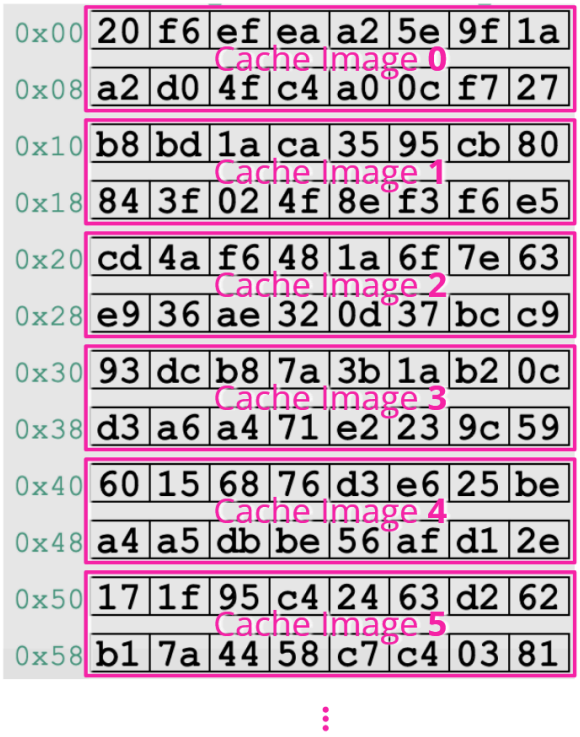
Cache Images

Cache images are a useful concept when approaching cache analysis and cache optimization problems that allows us to intuit cache mappings without having to do an address TIO breakdown every time.

A ***cache image*** is a cache-sized (*C* bytes) chunk of memory that we can divide memory with (the same way we do with cache blocks). The "cache image number" corresponds to the Tag (*i.e.*, all cache blocks within the same cache image share the same Tag field encoding).

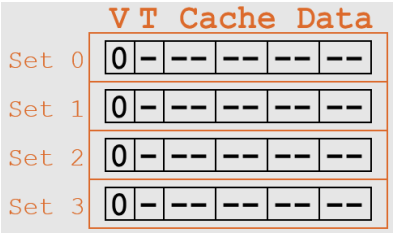
Within a cache image, cache blocks always map into the cache in the same order, *i.e.*, 1st block maps into Set 0, 2nd block maps into Set 1, each subsequent block maps into the next set. From this property, we can reason through which blocks will map into the same set in the cache based on the blocks' positions within their corresponding cache image. In particular, in a direct-mapped cache (like Lab 4 Part 2!), accessing the same block of two different cache images is guaranteed to cause an eviction.

A small example is shown where the beginning of memory is divided into cache images of size *C* = 16 bytes:



Example: Cache Block Mapping

If we have a direct-mapped cache of size 16 bytes with 4-byte blocks, it will look something like what is shown below on the right:



There are 4 cache blocks per cache image and these will map onto the corresponding set number because it is direct-mapped.

For example, every "Block 1" (*i.e.*, the second cache block) within a cache image is guaranteed to map into Set 1 of the cache. In the memory diagram above, this includes addresses **`0x04-07`, `0x14-17`, `0x24-27`, etc.**

Exercise 3: Matrix Conflicts

Assume our machine has 8-bit addresses and a **32 B direct-mapped cache** with **8 B block size**.

Given two 4 x 4 `int` matrices (2D arrays) A and Z, where A starts at address `0x00` and Z starts immediately after A. Which elements of matrix Z conflict with `A[1][1]` in the cache? Visualize the mappings using cache images!

Main 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
0x40	60	15	68	76	d3	e6	25	be
0x48	a4	a5	db	be	56	af	d1	2e
0x50	17	1f	95	c4	24	63	d2	62
0x58	b1	7a	44	58	c7	c4	03	81
0x60	54	84	69	8c	ab	cc	1f	d9
0x68	41	64	55	fc	83	cc	0d	98
0x70	55	20	4e	d5	3d	17	45	b5
0x78	93	e8	a9	91	27	31	b5	84