# CSE 351: The Hardware/Software Interface

Section 7

Caches, lab 4

# Caches

* Caches speed up accesses to memory through *temporal* and *spatial* locality
* Temporal locality in caches: recently-accessed data is more likely to be contained in the cache
* Spatial locality in caches: if `a[i]` is pulled into the cache, then `a[i + j]` for small `j` is likely to be pulled into the cache too
  * This depends on the size of cache lines, though

# Temporal locality example

- Pretend that the following code is executed more-or-less as written (with `result`, `b`, and `c` in registers):

```
int example(int* a, int b, int c) {
  int result = *a;
  result += b;
  result += c;
  result += *a;
  return result;
}
```

- `*a` is likely to be in the cache already going into the second access, so there is no need for the CPU to access memory twice (due to a *cache hit*)

# Temporal locality example

```
int example(int* a, int b, int c) {
  int result = *a;
  result += b;
  result += c;
  // (generate the Mandelbrot fractal
  // to some high recursive depth, e.g.)
  result += *a;
  return result;
}
```

- If we perform some memory-intensive operation prior to the second access to `*a`, then `*a` is less likely to be cached when the CPU attempts to read it again (resulting in a *cache miss*)

# Spatial locality example

```
int example(int* array, int len) {
  int sum = 0;
  for (int i = 0; i < len; ++i) {
    sum += array[i];
  }
  return sum;
}
```

- Accessing memory causes neighboring memory to be cached as well
- If cache lines are 64-bits in size, for example, then accessing `array[0]` will pull `array[1]` into the cache too, so `len / 2` memory accesses are required in total

# Types of caches

* There are a variety of different cache types, but the most commonly-used are direct-mapped caches, set-associative caches, and fully-associative caches
  * Which type to use where depends on size, speed, hardware cost, and access pattern considerations

# Direct-mapped caches

* Direct-mapped caches are hash tables where the entries are cache lines (data blocks) of size *B* containing cached memory

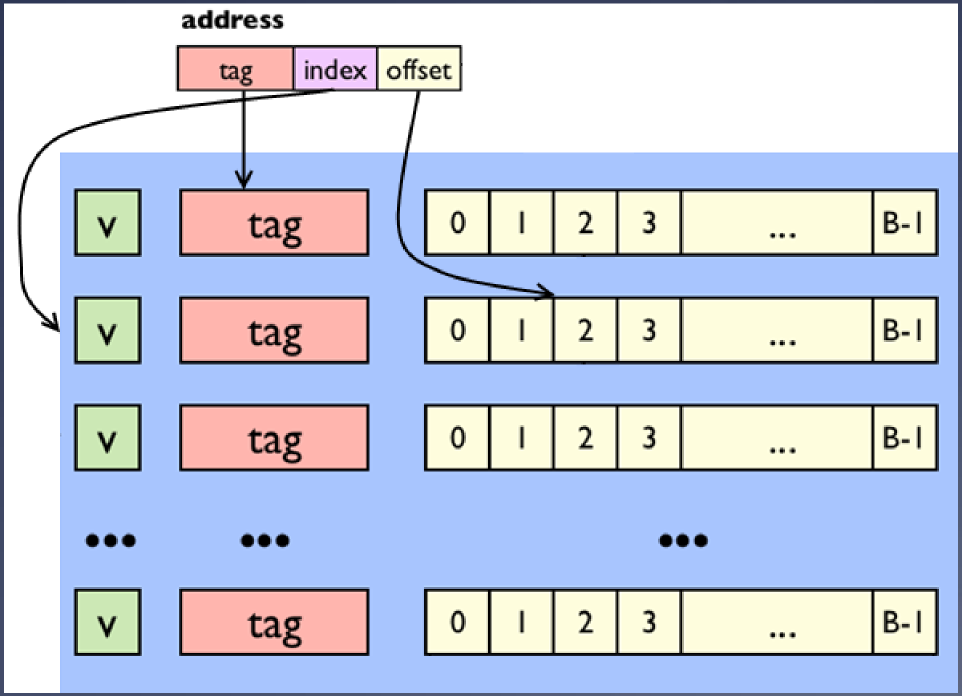
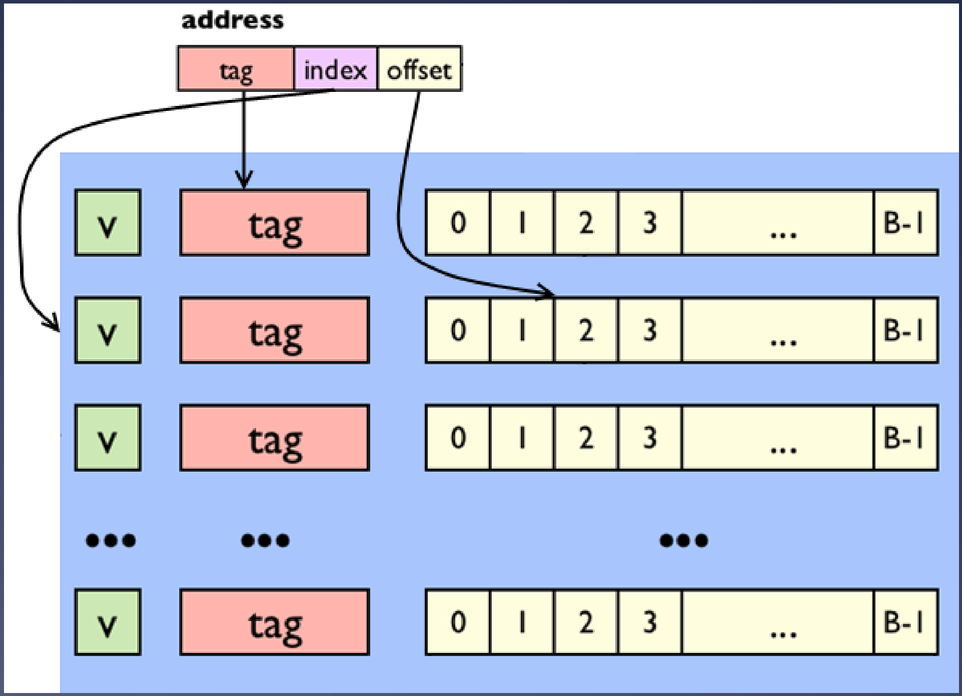



*Diagram originally from Tom Bergan

# Direct-mapped caches

- Addresses are broken up into [tag, index, offset]
  - *tag* helps prevent against hash collisions
  - *index* specifies which data block to access
  - *offset* specifies the offset at which to read/write data
- The *valid* bit simply indicates whether data block contains data



# Direct-mapped cache example

- Let's say we have an address of 8 bits in length (say 0xF6), where the tag is 2 bits, the index is 4 bits, and the offset is 2 bits
  - 0xF6 = 0b11110110 = [tag, index, offset] = [0b11, 0b1101, 0b10]
  - How big are data blocks? At most how many cache entries can be represented? How big are cache entries in total?
- To read from this address in a direct-mapped cache, look at the valid bit and tag at line *index*
  - If the valid bit is set and *tag* matches what is stored there, return the data at *offset* (cache hit)
  - Otherwise perform a memory access and store retrieved data in the cache (cache miss)

# Direct-mapped cache example

- To write to this address in a direct-mapped cache, set the valid bit, tag, and data at line *index*
  - Subsequent reads that match this tag will now result in a cache hit
- What happens if an entry at that index with a different tag already exists?
  - Overwrite the tag and data with the new values
  - …but this can cause poor performance, since now attempting to access the data will result in a cache miss
- Also need to update data stored in memory: can either *write-through* (update on all memory writes) or *write-back* (update on cache overwrites due to either memory reads or writes)

# Set-associative caches

- Set-associative caches help to mitigate the situation where particular cache lines are frequently invalidated
  - Which part of the address affects whether such invalidations happen?
- Addresses are taken to have the same [tag, index, offset] form when indexing into set-associative caches
- Each *index* maps to a set of *N* cache entries in an *N*-way associative cache

# Set-associative caches

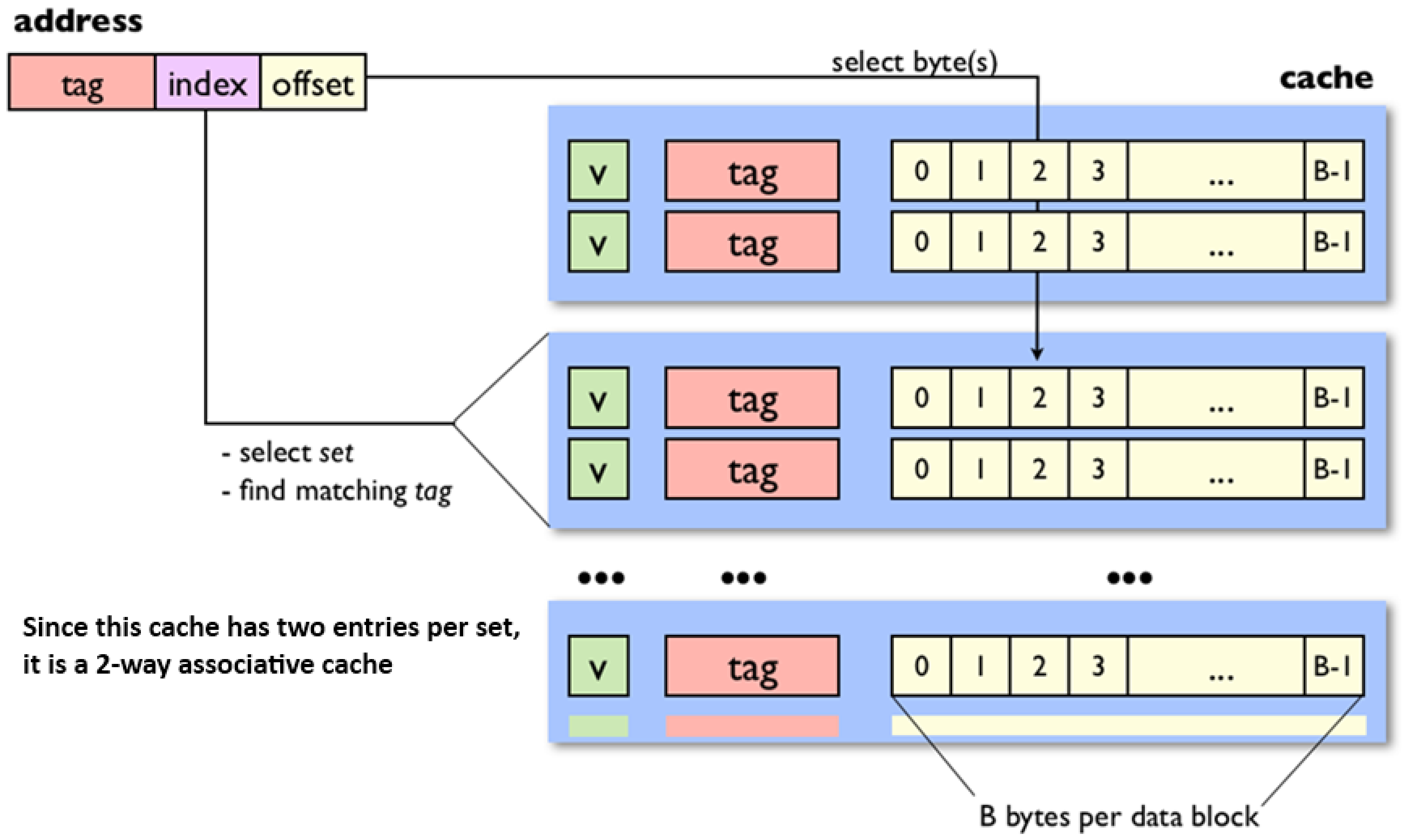



*Diagram from Tom Bergan

# Set-associative caches

* When performing a read from a set-associative cache, check every entry in the set under *index*
  * If an entry has a matching tag and its valid bit is set, then return the data at the address' offset
  * If no entry has both a matching tag and valid bit, then perform a fetch from memory and add a new entry for this address/data
* If all cache entries in a set fill up, pick one of them to evict using a *replacement policy*

# Set-associative caches

- When performing a write to a set-associative cache, check every entry in the set under *index*
  - If there is an existing entry, simply update it
  - Otherwise add new entry and (optionally) write the data to memory as with direct-mapped cache

# Set-associative caches

- Given addresses of the form [tag, index, offset] with *s* bits for the index and *b* bits for the offset:
  - There can be at most $2^s$ addressable sets
  - There are exactly $2^b$ addressable bytes in the data blocks

# Fully-associative caches

- Instead of having multiple sets of cache entries, keep just one
  - What are the implications of this in terms of hardware costs versus access times?
- Fully-associative caches are not very common, but the translation lookaside buffer (TLB), which facilitates virtual address to physical address translation, is one such example
  - Expect more on the TLB in operating systems or (maybe?) hardware design and implementation

# Lab 4

* Analyze the cache-related performance gains/losses of programs, and infer the geometries (cache size, associativity, and block size) of unknown caches
* Make sure to use the functions provided in mystery-cache.h for the second part of the lab—these caches are simulated, so regular memory accesses won't pass through them!