Caches

High-speed (small) storage between the CPU & memory

Nice compromise between:

- registers which are too few to keep locality close to the CPU
- memory which takes too many cycles to access

When fetch an instruction or load data, check in the cache first.

When store load data, write it in the cache first (there are variations).

First used in the IBM 360/85

Originally invisible to the programmer
    now being exposed to the compiler

LOTS of research done on caching

Physical Cache Organization

Data: the contents of the cache (**instructions or data**)

- made up of many **blocks** of data
- **block**:
  - number of bytes transferred on a memory read
  - number of bytes associated with a tag

State: indicates whether the data is valid for a particular cache entry

- is set to invalid before anything has been loaded into the cache

Tag: helps you decide if this is the data being accessed

- a many-to-one relationship between memory and cache locations
- if there are *n* entries in the cache, every *nth* location in memory maps to the same cache location

Addressing a Cache

Partitioning an address to access a cache (when there is one block in each cache line)

- **byte offset**: bits that specify which bytes in a block you're accessing
  field size: log of number of bytes in a block
- **index**: bits used to index a cache
  field size: log of number of entries in the cache
- **tag**: high-order bits used to identify a particular memory address in the accessed cache location
  field size: those bits left over

Picking an entry:

- block address (index + tag) = address of the block in memory
- block address modulo the number of blocks in the cache = line in cache

Checking to see if the entry is the correct one:

- compare the address tag to the cache tag
  - if a match, this is the block we want
  - if no match, this is some other block that maps to the same cache location
- if a match and the valid bit is set, we have a **cache hit**
  - otherwise we have a **cache miss**

How Caches Work (Summary)

In a load/store, the memory address is an index into memory

- each byte (word) of memory has a unique address
  - there is a unique mapping between addresses & memory locations
  - the entire address is used to access memory
- caches are smaller than memory
  - therefore not all memory locations can reside in the cache at once
  - only part of the address bits are used to index the cache

CPU sends an address to the cache:

- **cache hit**
  - the address is already mapped to an entry in the cache
  - if a read, data is sent back to the CPU
  - if a write, data is stored in the cache
- **cache miss**
  - the address is *not* mapped to an entry in the cache
  - look in memory

Caches are loaded on demand:

- when a data transfer instruction is executed
- when an instruction is fetched

Handling a Cache Miss

Cache miss: either the tag doesn't match or the invalid bit is clear

The steps:

- stall the CPU pipeline
- job of the **cache controller**:
  - send the address & read signal to memory
  - wait for the data to come back
  - update the cache:
    - fill the data portion with the new block
    - update the tag with the high-order bits of the address
    - set the valid bit
- restart execution with the cycle that caused the cache miss (IF on an instruction, MEM on a load)

Multi-word Blocks

Takes better advantage of spatial locality

- 1 miss per block rather than 1 miss per word

Picking an entry:

- use the same formula
  - block address modulo number of blocks in the cache
  - just a bigger block
- more specific definition for block address
  - block address = word/byte address divided by the number of words/bytes in a block
- fields of an address for accessing a cache with multi-word blocks:
  - tag
  - index
  - word offset within a block
  - byte offset within a word

  ] byte offset within a block

Writing to the cache:

- usually read the block first

Associativity

Placement of blocks in a cache

- **direct mapped**: a memory address can be mapped to one place in the cache
- **fully associative**: a memory address can be mapped anywhere in the cache
  - all tags must be compared at once
  - comparator for each entry, not one for the whole cache
- **set associative**: a memory address can be mapped to a small, fixed number of cache locations
  - n locations ⇒ n-way set associative
  - cache divided into **sets** that contain the n blocks
    - address indexes into a particular set
    - compare the tags for all blocks in the set

Associativity

Associativity:

- an implementation that indicates a specific number of blocks in a set
- also, the number of blocks checked on a cache access

Refine the cache access formula

- block address modulo number of **sets** in the cache
  - # sets in the cache = # blocks in the cache/# blocks in a set
  - # bits in the index = log (number of entries in the cache/ associativity)

For the same size cache, an increase in the associativity

- increases the number of blocks in a set
- decreases the number of sets (lines) in the cache

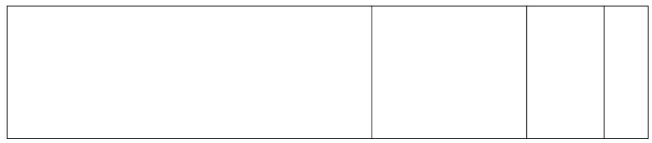
This affects the index bits!

- decreases the size of the index
- increases the size of the tag
- study the example on p. 575

An Example

32b address, 16KB cache, 64B blocks, DM:

- where are the tag, index, word offset within a block & byte offset within a word fields?
- how big are they?
- what are their bit boundaries?



Which numbers change if the 16KB cache is 4-way associative?