The Basics of Caches

- Until specified otherwise, it will be assumed that a block is one word of data.

- **Three issues:**
  - How do we know if a data item is in the cache?
  - If it is, how do we find it?
  - If it is not, what do we do?

- It boils down to
  - where do we put an item in the cache?
  - how do we identify the items in the cache?

- **Two solutions**
  - put item anywhere in cache (associative cache)
  - associate specific locations to specific items (direct mapped cache)

Note: we will assume that memory requests are for words (4 Bytes) although an instruction can address a byte.
Fully associative cache

To cache a data word, \( d \), whose memory address is \( L \):

- Put \( d \) in any location in the cache
- Tag the data with the memory address \( L \).

**Example:**

- An 8-word cache (indexed by 000, …, 111)
- A 32-words memory (addresses 00000, …, 11111)

**Advantage:** can fully utilize the cache capacity

**Disadvantage:** need to search all tags to find data

Direct Mapped Cache (direct hashing)

Assume that the size of the cache is \( N \) words.

To cache a data word, \( d \), whose memory address is \( L \):

- Put \( d \) in cache at index = \( L \mod N \)
- Tag the data with \( L / N \)

**Example:**

- memory address \( L = 6 \) (00110)
- cached in index 6 mod 8 = 6 (110)
- Tagged with 6 / 8 = 0 (00)

- memory address \( L = 9 \) (01001)
- cached in index 9 mod 8 = 1 (001)
- Tagged with 9 / 8 = 1 (01)

**Drawback:** collision

- memory address \( L = 25 \) (11001)
- cached in index 25 mod 8 = 1 (001)
- Tagged with 25 / 8 = 3 (11)

Should “evict” Location 9 before caching Location 25.
Direct Mapped Cache (example of a 4KB cache)

- Use the 10-bit index to access the cache
- Compare the stored tag with the 20 bit tag from the address
- if tag match and valid bit = 1, then cache hit (Hit = 1)
- Else cache miss (Hit = 0)

A valid bit is used to indicate that a location contains valid data

Assume that a memory address is 32 bit long (byte address, as in MIPS)
- 2 bits for byte offset
- 10 bits for index
- 20 bits for tag

4KB cache, 4Bytes/word
Cache size = $2^{13}$ words

Hints vs. Misses in direct mapped cache

One of the following scenarios occur in the cache when the CPU requests a memory address to read or write memory location $L$:

- **In case of a cache hit** (tag match and valid bit = 1)
  - read from or write the new data into the cache at index $L \mod N$

- **In case of a miss**: (tag mismatch or valid bit = 0)
  - Stall the CPU
  - If cache index "$L \mod N" contains valid data (valid bit =1), **evict** the current data
  - write the current data back to memory
  - Fetch the data from memory and deliver to cache
  - Read from or write the new data into the cache
  - Resume the execution of the CPU

Reason: if data was changed (due to write operations) after it was brought to memory, then the change has to be reflected in memory
**Two policies when writing data into the cache**

- **Write back policy** – uses a “dirty bit”
  - When you bring data into the cache, set its *dirty bit* to 0
  - On a write operation, write only into the cache and set the *dirty bit* to 1
  - When the data is evicted from the cache, write the data back to memory only if the *dirty bit* = 1.

- **Write through policy** – does not use a “dirty bit”
  - Every time you write into the cache, write also into memory
  - Hence, do not need to “write back” when you evict data from the cache
  - On a write miss, may even write only to memory and not bring data to the cache (called a “no write allocate policy”, as opposed to “write allocate”)

**Storage overhead (example of a 4KB cache)**

The total number of bits needed for a 4KB cache assuming 32-bit address:

- 1024 * 32 bits for data
- 1024 * 20 bits for tags
- 1024 * 1 valid bits

\[2^{10} \times 53 = 53 \text{ Kbits}\]

In addition, there is usually a “dirty bit” per cache entry.
Exploiting spatial locality by using “multiple word” blocks

Example: two words per block

Block index

Block offset (offset of the word within the block)

Tag offset(1) offset(0)

Tag = (memory word address) / (cache size in words)
Word index = (memory word address) mod (cache size in words)

Block index = Word Index / block size in words
Block offset = Word Index mod block size in words

Alternatively
Block address = Word address / block size in words
Block index = Block address / cache size in blocks

- One tag per block ==> improves efficiency
- Write misses have to preserve block integrity
  - Copy block from memory to cache
  - Write the word into the block in the cache.

Example: block size = 2 words

- When a CPU issues a memory address, it is decomposed into a tag and a word index
- The word index is decomposed into block index and a block offset
Example: 64KB cache with blocks of 4 words (16 bytes)

Word = $2^2$ bytes
Block = $2^2$ words
Cache = $2^{12}$ blocks

The total number of bits needed for a 64KB cache assuming 32-bit address is:

- $2^{12} * 128$ bits for data
- $2^{12} * 16$ bits for tags
- $2^{12} * 1$ valid bits

$2^{12} * 145 = 580$ Kbits

Final note on notation

- We assumed that the words in memory are addressed sequentially as 0, 1, 2, ...
- When referring to a “word address” in byte-addressable architectures (such as MIPS), we ignore the low order two bits that specify the byte offset within a word.
- When referring to a “block address”, we ignore the bits specifying the block offset.

- Example, in a 32-bit architecture with 4-words blocks
  Byte address $\rightarrow$ b31, b30, ..., b4, b3, b2, b1, b0
  Word address $\rightarrow$ b31, b30, ..., b4, b3, b2
  Block address $\rightarrow$ b31, b30, ..., b4
- In decimal, if the byte address is $A$, then
  Word address, $W = A / 4$
  Block address, $B = W / 4$