Review: Cache Memory

- Memory Tech: SRAM, DRAM
- Memory hierarchy: reg, cache, mem, hard disc, tape
- What is cache?
  - the level of the memory hierarchy between CPU and main memory. Also used to refer to any storage managed to take advantage of locality.
- **Block**: Group of words
- **Set**: Group of Blocks

```
CPU ← word
    ↓
  cache ← Mapping function
     ↓
block → Main memory
```

- How do we organize cache? **In blocks/sets**
- Where does each memory address map to? (Remember that cache is subset of memory, so multiple memory addresses map to the same cache location.)
  - Direct mapped, set-associative, fully-associative
- How do we know which elements are in cache?
- How do we quickly locate them?
**Direct-Mapped Cache**

- In a **direct-mapped cache**, each memory address is associated with exactly one **block** within the cache:
  - e.g., lots of items at the lower level (memory) share locations in the upper level (cache)
  - Mapping: memory address is **modulo** the number of blocks in the cache
  - Therefore, we only need to look in a single location in the cache for the data if it exists in the cache
  - Block is the unit of transfer between cache and memory

### Memory Address vs. Cache Location

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Memory</th>
<th>Cache Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>e</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>f</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>g</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>h</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>i</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>k</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>l</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>m</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>n</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>o</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>q</td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td>a</td>
<td>0</td>
</tr>
</tbody>
</table>

- Cache Location 0 can be occupied by data from:
  - Memory location 0, 4, 8, ...
  - (memory address is modulo the # of blocks)
Example

- Block size is one word of data
- Cache (8 words), Memory (32 words)
- Low-order 3 bits used as cache index

![Diagram of cache and memory]

How do we know whether a requested word is in the cache or not?

- Since each cache location can contain the contents of a number of different memory locations, how do we tell which one is in there?
  - Adding a set of tags to the cache (to detect a hit)
  - The tag needs only to contain the upper portion of the memory address, corresponding to the bits that are not used as an index into the cache.
  - For example: `lw $s1, (01001)` for the last slide
    - Need only have 2 of the 5 address bits in the tag since the lowest 3 bits of the address select the block

Which memory address

Cache block index

01001
Cache Structure

Does the cache block have valid info.?

- For instance,
  - when a processor starts up, the cache will be empty
  - even after executing many instructions, some of the cache entries may still be empty
  - Need to know that the tag should be ignored for such entries.

  - Add a valid bit to indicate whether an entry contains a valid address.

  a. Before the reference to $X_n$

  b. After the reference to $X_n$
For MIPS:

Each cache block has [valid bit + tag + data]

Direct Mapped Cache

Example

- Direct-mapped cache with 8 one-word cache block
- Memory reference sequence: 22, 26, 22, 26, 16, 3, 16, 18

<table>
<thead>
<tr>
<th>Memory reference</th>
<th>Hit or miss</th>
<th>Cache block</th>
<th>Hit or miss</th>
<th>Cache block</th>
</tr>
</thead>
<tbody>
<tr>
<td>22 (10110)</td>
<td>Miss</td>
<td>6</td>
<td>0 (000)</td>
<td>Y</td>
</tr>
<tr>
<td>26 (11010)</td>
<td>Miss</td>
<td>2</td>
<td>1 (001)</td>
<td>N</td>
</tr>
<tr>
<td>22 (10110)</td>
<td>hit</td>
<td>6</td>
<td>2 (010)</td>
<td>Y</td>
</tr>
<tr>
<td>26 (11010)</td>
<td>hit</td>
<td>2</td>
<td>3 (011)</td>
<td>Y</td>
</tr>
<tr>
<td>16 (10000)</td>
<td>miss</td>
<td>0</td>
<td>4 (100)</td>
<td>N</td>
</tr>
<tr>
<td>3 (00011)</td>
<td>miss</td>
<td>3</td>
<td>5 (101)</td>
<td>N</td>
</tr>
<tr>
<td>16 (10000)</td>
<td>hit</td>
<td>0</td>
<td>6 (110)</td>
<td>Y</td>
</tr>
<tr>
<td>18 (10010)</td>
<td>miss</td>
<td>2</td>
<td>7 (111)</td>
<td>N</td>
</tr>
</tbody>
</table>
How to find a word in cache?  
-Cache address components

• For example:

   \text{lwx} \$s1, \, 0000 \, 1001 \, 1100 \, 0110 \, 1010 \, 1111 \, 0101 \, 1000$

• Solution: divide memory address into three fields

<table>
<thead>
<tr>
<th>Tag</th>
<th>Block Index</th>
<th>Blockoffset</th>
</tr>
</thead>
</table>

• \textbf{Index}: specifies the cache index (which “row” of the cache we should look in)

• \textbf{Offset}: once we’ve found correct block, specifies which byte within the block we want

• \textbf{Tag}: the remaining bits after offset and index are determined; these are used to distinguish between all the memory addresses that map to the same location

Direct-Mapped Cache Example (1/4)

• Suppose we have a 16KB direct-mapped cache with 1 word blocks.

• Determine the size of the tag, index and offset fields if we’re using a 32-bit architecture.

• Determine the total size of cache

• \textbf{Block Offset}

  – need to specify correct byte within a block

  – block contains 1 word = 4 bytes = $2^2$ bytes

  – need 2 bits to specify correct byte
Direct-Mapped Cache Example (2/4)

• Block Index
  – need to specify correct row in cache
  – cache contains 16 KB = \(2^{14}\) bytes
  – each block contains \(2^2\) bytes (1 word)
  – \# rows/cache = \(\frac{2^{14} \text{ bytes/cACHE}}{2^2 \text{ bytes/row}}\) = \(2^{12}\) rows/cache
  – need 12 bits to specify this many rows

Direct-Mapped Cache Example (3/4)

• Tag
  – used remaining bits as tag
  – tag length = mem addr length
    - offset
    - index
    = 32 - 2 - 12 bits
    = 18 bits
  – so tag is leftmost 18 bits of memory address
Direct-Mapped Cache Example (4/4)

• Total size
  = number of rows x size of each row
  = \(2^{12}\) rows x (32 bits + 18 bits + 1 bits)
  = 4096 x 51 bits = 208896 bits = 26,112B
  > 16 KB

Data (1 word)  Tag  Valid bit