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

Cache Design

- 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?
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.

---

**direct-mapped cache**

```plaintext
Memory Address  Memory
0   a
1   b
2   c
3   d
4   e
5   f
6   g
7   h
8   i
9   k
A   l
B   m
C   n
D   o
E   q
```

**4 Byte Direct Mapped Cache**

```plaintext
Cache Index
0
1
2
3
```

- 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

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.
For MIPS:

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

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>
</tr>
</thead>
<tbody>
<tr>
<td>22 (10110)</td>
<td>Miss 6</td>
<td></td>
</tr>
<tr>
<td>26 (11010)</td>
<td>Miss 2</td>
<td></td>
</tr>
<tr>
<td>22 (10110)</td>
<td>hit 6</td>
<td></td>
</tr>
<tr>
<td>26 (11010)</td>
<td>hit 2</td>
<td></td>
</tr>
<tr>
<td>16 (10000)</td>
<td>miss 0</td>
<td></td>
</tr>
<tr>
<td>3 (00011)</td>
<td>miss 3</td>
<td></td>
</tr>
<tr>
<td>16 (10000)</td>
<td>hit 0</td>
<td></td>
</tr>
<tr>
<td>18 (10010)</td>
<td>miss 2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache Block</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (000)</td>
<td>Y</td>
<td>10</td>
<td>Mem[16]</td>
</tr>
<tr>
<td>1 (001)</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 (010)</td>
<td>Y</td>
<td>10</td>
<td>Mem[18]</td>
</tr>
<tr>
<td>3 (011)</td>
<td>Y</td>
<td>00</td>
<td>Mem[3]</td>
</tr>
<tr>
<td>4 (100)</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 (101)</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 (110)</td>
<td>Y</td>
<td>10</td>
<td>Mem[22]</td>
</tr>
<tr>
<td>7 (111)</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How to find a word in cache?

- Cache address components

- For example:
  \[ \text{lw \$s1, 0000 1001 1100 0110 1010 1111 0101 1000} \]

- Solution: divide memory address into three fields

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

- **Index**: specifies the cache index (which “row” of the cache we should look in)

- **Offset**: once we’ve found correct block, specifies which byte within the block we want

- **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

- **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 = $\frac{2^{14} \text{ bytes/cache}}{2^2 \text{ bytes/row}} = 2^{12} \text{ rows/cache}$
- 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
  \[= \text{number of rows} \times \text{size of each row}\]
  \[= 2^{12} \times (32 \text{ bits} + 18 \text{ bits} + 1 \text{ bits})\]
  \[= 4096 \times 51 \text{ bits} = 208896 \text{ bits} = 26\,112 \text{B} > 16 \text{ KB}\]

Set Associative Mapping

- Generalizes all Cache Mapping Schemes
  - Assume cache contains \(N\) blocks
  - 1-way SA cache: Direct Mapping
  - \(M\)-way SA cache: if \(M = N\), then fully assoc.

- Advantage
  - Decreases miss rate (more places to find B)

- Disadvantage
  - Increases hit time (more places to look for B)
  - More complicated hardware
Associative Cache Example

Three mapping schemes

Direct-mapped

Set associative

Fully associative
Set-Associative Cache

- **mapping**: (memory address) modulo (# of sets in cache)
- **Example**: two-way set associative cache with 4 1-word block
- **Direct-mapped**: 2-way SA: Fully SA

<table>
<thead>
<tr>
<th>address</th>
<th>block</th>
<th>address</th>
<th>Set</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 or 1</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>0 or 1</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>7</td>
<td>1</td>
<td>2 or 3</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>

- Which block to replace for set-associative cache?
  - **Random**
  - The *least recently used* (LRU) block within a set.
  - First-in-First-Out (FIFO)

Replacement

- **Choice of entry to replace on a miss**
  - Least recently used (LRU)
    - Complex and costly hardware for high associativity
  - Random
    - Close to LRU, easier to implement
- **Virtual memory**
  - LRU approximation with hardware support
### Example

- **cache with 4 1-word block, Reference: 0 8 0 6 8**

**Direct-mapped:**

<table>
<thead>
<tr>
<th>address</th>
<th>block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>

- **2-way SA:**

<table>
<thead>
<tr>
<th>address</th>
<th>Set</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0 or 1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0 or 1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ref</th>
<th>H/M?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>M</td>
</tr>
<tr>
<td>8</td>
<td>M</td>
</tr>
<tr>
<td>0</td>
<td>M</td>
</tr>
<tr>
<td>6</td>
<td>M</td>
</tr>
<tr>
<td>8</td>
<td>M</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Block</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>M[8]</td>
</tr>
<tr>
<td>2</td>
<td>M[6]</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

### Example (continued)

- **Reference: 0 8 0 6 8**

**Fully associative**

<table>
<thead>
<tr>
<th>address</th>
<th>Set</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0–3</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0–3</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0–3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ref</th>
<th>H/M?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>M</td>
</tr>
<tr>
<td>8</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>6</td>
<td>M</td>
</tr>
<tr>
<td>8</td>
<td>H</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Block</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>M[0]</td>
</tr>
<tr>
<td>1</td>
<td>M[8]</td>
</tr>
<tr>
<td>2</td>
<td>M[6]</td>
</tr>
</tbody>
</table>

- **2-way SA:**

<table>
<thead>
<tr>
<th>address</th>
<th>Set</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0 or 1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0 or 1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ref</th>
<th>H/M?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>M</td>
</tr>
<tr>
<td>8</td>
<td>M</td>
</tr>
<tr>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>6</td>
<td>M</td>
</tr>
<tr>
<td>8</td>
<td>M</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set</th>
<th>block</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>M[0]</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>M[8]</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
Locating a block in SA cache

**Cache Address Components:**
- **Index i:** Selects the set $S_i$
- **Tag:** Used to find the memory block you’re looking for, by comparing with the $n$ blocks in the selected set $S$
- **Block Offset:** Address of desired data within the block

```
Tag  Index  Offset
```

2-way Cache Hardware

**N-way cache:**
$N$ muxes, and gates, comparators
4-Way Set Associative Cache Organization

Performance

Miss rate

Associativity

0%  3%  6%  9%  12%  15%

One-way  Two-way  Four-way  Eight-way

Associativity

1 KB  16 KB

2 KB  32 KB

4 KB  64 KB

8 KB  128 KB
How Much Associativity

- Increased associativity decreases miss rate
  - But with diminishing returns

- Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%

Finding a Block

<table>
<thead>
<tr>
<th>Associativity</th>
<th>Location method</th>
<th>Tag comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct mapped</td>
<td>Index</td>
<td>1</td>
</tr>
<tr>
<td>n-way set associative</td>
<td>Set index, then search entries within the set</td>
<td>n</td>
</tr>
<tr>
<td>Fully associative</td>
<td>Search all entries</td>
<td>#entries</td>
</tr>
<tr>
<td></td>
<td>Full lookup table</td>
<td>0</td>
</tr>
</tbody>
</table>

- Hardware caches
  - Reduce comparisons to reduce cost

- Virtual memory
  - Full table lookup makes full associativity feasible
  - Benefit in reduced miss rate
Partition of memory address in SA cache

- Suppose we have a 32KB 2-way SA cache with 2 word blocks.
- Determine the size of the tag, index and offset fields if we’re using a 32-bit architecture.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Set Index</th>
<th>Offset</th>
</tr>
</thead>
</table>

- # of bits for offset: specify the correct byte within a block, depends on the size of block
- # of bits for set index: specify the correct cache set, depends on the number of sets
- # of bits for tag: remaining bits

---

2-Way SA Cache Example (2/4)

- Set Index
  - need to specify correct row in cache
  - cache contains 32 KB = $2^{15}$ bytes
  - each block contains $2^3$ bytes (2 words)
  - # rows/cache = \[
  \frac{2^{15} \text{ bytes/cache}}{2^3 \text{ bytes/block} \times 2 \text{ blocks/row}} = 2^{11} \text{ rows/cache}
  \]
  - need 11 bits to specify this many rows
2-Way SACache Example (3/4)

• Block offset: 3 bits for 2-word block
• Tag
  – used remaining bits as tag
  – tag length = mem addr length
    - offset
    - index
    = 32 - 3 - 11 bits
    = 18 bits
  – so tag is leftmost **18 bits** of memory address

2-way SA Cache Example (4/4)

• Total size
  = number of rows x size of each row
  = number of rows x \( \bar{N} \) x size of each block
  = \( 2^{11} \) rows x 2 x (64 bits + 18 bits + 1 bits)
  = 4096 x 83 bits = 339968 bits = 42,496B
  > 32 KB