Memory Hierarchy

Read: Sections 5.1 to 5.3 (4th edition).

- Users want:
  - Lots of fast (quick response) memory.
  - Low cost.
- Solution: Hybrid systems.
  - Memory hierarchy containing different types of memory.

**Memory Types:**

- Fast memory:
  - SRAM — static random access memory.
  - Value stored on a pair of inverting gates; need 6 transistors per bit.
  - Value remains as long as power is supplied to the memory (hence *static*).
  - Fast: 0.5 to 2.5 ns (nanosecond) access time.
Memory Types (continued):

- Not so fast memory — This is what computer manufacturers mean when they advertise memory in a computer.
- DRAM — dynamic RAM.
- Value is stored in a capacitor (charged or not charged), 1 transistor per bit.
- Must be refreshed, “read” value about every 50 ms (milliseconds).
- Dense, many more bits on same size chip (compared to SRAM).
- Slow: 50 to 70 ns, 5 to 10 times slower than SRAM.
- Cheap: $20 - $75 per GigaByte (2008, fm page 453); $12 per GigaByte (purchase made in August 2013).
- DRAM variations:
  - SDRAM — synchronous dynamic RAM.
    - Uses data input register and data output register to buffer data.
    - 3 clock cycles to get first word.
    - 1 clock cycle per word for successive words.
    - Processor does not have to take into account delay, clock does that for it.
  - DDR — double data rate RAM.
    - Read (or write) a value on both the leading and trailing clock edges.
**Memory Hierarchy:**

![Diagram of Memory Hierarchy]

Levels in the memory hierarchy:
- Level 1
- Level 2
- ... (indicated by '...')
- Level n

Size of the memory at each level:
- Increasing distance from the CPU in access time
Locality:

- A principle that makes having a memory hierarchy a good idea.
- If an item is referenced:
  - **Temporal** locality: The item will tend to be referenced again soon.
  - **Spatial** locality: Nearby items will tend to be referenced soon.
- Why does code have locality?
- Why does data have locality?

- Our initial focus:
  - Two levels of memory: upper and lower.
  - Block: minimum unit of memory.
  - Hit: data requested is in the upper level of memory.
  - Miss: data requested is not in the upper level of memory.
Cache — Upper Memory:

- Closest memory to the CPU.

- Two issues:
  - How do we know if a data item is in the cache?
  - If it is in the cache, how do we find it?

- Our first example:
  - Block size is one word of data; 4 bytes.
  - “Direct Mapped”
    - For each block of data at the lower level, there is exactly one location in the cache where it might be.
    - E.g., lots of items at the lower level “share” one location in the upper level.
**Direct Mapped Cache:**

- Mapping: an address is modulo the number of blocks in the cache.
- E.g., an 8 block cache for a 32 block memory:

  - Cache location taken from the 3 least significant bits of the memory address, since $2^3 = 8$.
  - Cache size is always a power of 2 for this reason!
Direct Mapped Cache (continued):

- Another example: a 16-block cache for a 64 block memory:

```
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
```

- Here, we need 4 bits for the cache address — take the 4 least significant bits of the address.
- For comparison: in the 8 block cache, the red memory locations mapped to the gray cache, and green to blue.
Direct Mapped Cache (continued):

- Issues:
  - How do we know which value from the lower memory is currently in the cache location?
    - Store “tag” in the cache, using the upper part of the memory address (part that is not the cache address).
  - How do we know if the value in the cache is a valid value?
    - Use “valid” bits.
    - Set all valid bits to zero when program starts.
    - Also, when a “context switch” occurs.

- Use the tag field to identify the correct cache line.
Direct Mapped Cache (continued):

- **MIPS Example:**
  - Block size is one word, 4 bytes.
    - Need 2 bits for the Byte offset.
  - Cache contains 1,024 blocks.
    - Need 10 bits for the Index.
  - 32 bits - 10 for Index - 2 for Byte = 20 bits for the Tag.

- **Issues:**
  - How do we know which value from the lower memory is currently in the cache location?
    - Store the tag in the cache.
  - How do we know if the value in the cache is a valid value?
    - Valid bit in the cache.

- Cache width = 32 bits data + 20 bits Tag + 1 bit Valid = 53 bits.

- What kind of locality is this?
Direct Mapped Cache (continued):

- Spatial locality.

![Diagram of a Direct Mapped Cache with label and address mapping]
Direct Mapped Cache (continued):

- Calculations:
  - Block size is 4 words, 16 bytes.
    - 2 bits for the Byte offset.
    - 2 bits for the Word offset.
  - Cache contains 4,096 blocks (rows).
    - 12 bits for the Index.
  - 32 bits - 12 bits for Index - 2 bits for Word - 2 bits for Byte = 16 bits for the Tag.

- Issues:
  - How do we know which value from the lower memory is currently in the cache location?
    - Store tag in the cache (same answer).
  - How we know if the value in the cache is a valid value?
    - Valid bit in the cache (same answer).
  - Cache width = 4 * (32 bits of data per word) + 16 bits Tag + 1 bit Valid = 145 bits.
  - The block offset determines which word passes the multiplexor.

**Hits vs. Misses:**

- **Read hits.**
  - This is what we want!

- **Read misses.**
  - Stall the CPU.
  - Fetch block from memory.
  - Deliver block to the cache.
  - Restart the CPU.

- **Write hits:**
  - Can replace data in the cache and memory (*write-through*).
  - Write the data only into the cache (*write-back* the cache later).

- **Write misses:**
  - Read the entire block into the cache.
  - Then write the word into the cache.
  - Then, replace data in memory when writing to cache (*write-through*), or later (*write-back*).
**Split Cache:**

- Most systems use a *split* cache:
  - Usually for the Level 1 cache (the one closest to the CPU).
- Using one cache (instead of a split cache) allows the sharing of the cache resource:
  - The space in the cache can be applied to code or data, as needed for individual programs.
  - But:
    - Code tends to exhibit strong *temporal* locality.
      - And, also has spatial locality.
    - Data tends to exhibit strong *spatial* locality.
- Splitting the cache allows the data cache to have spatial locality.
**Associativity:**

- Can reduce the miss ratio of a cache by using associativity.
- Allows multiple locations in the cache where the contents of a particular memory location might reside.
- Can have 2-way, 3-way, 4-way, etc., associativity.
  - Note: 1-way set associative == direct mapped.
- Consider an array of integers where we want to process every other element.
  - Direct mapped cache can hold only 4 values from the array. Other 4 elements of the cache are unused.
  - 2-way set associative allows 8 elements of the array to be in the cache at once. All elements of the cache are utilized.
  - Decreases the miss ratio!
Associativity (continued):

- More possibilities:

**4-way set associative**

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each memory location can map to four possible spots in the cache.

**8-way set associative**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each memory location can map to eight possible spots in the cache.
Associativity (continued):

- An implementation of a 4-way set associative cache:

```plaintext
0 1 2
253 254 255
```

```
V  Tag  Data
V  Tag  Data
V  Tag  Data
V  Tag  Data
```

```
31 30 ........ 11 10 9 ... 4 3 2 1 0
```

Byte offset (2 bits)

4 to 1 Mux

OR

Hit

Data

CSc 252 — Computer Organization
Decreasing miss penalty:

- Add a second level cache:
  - Often the primary cache is on the same chip as the processor.
  - Secondary (level-2) cache is off-chip.
- For dual-core (and multi-core) designs:
  - The primary (level-1) cache is with the core
    - 1 instruction memory cache per core.
    - 1 data memory cache per core.
  - The secondary (level-2) cache is on the chip and shared by all the cores.
  - There may (or may not) be a third (level-3) cache. This would be off the chip.
**Sun Example:**
- Sun UltraSPARC III CPU.
  - 32 byte (256 bit) dedicated data path for L2 cache.
  - 128 bit data path to System (memory, I/O, any remote CPUs)
    - Runs at 1/8 of the CPU’s clock speed.
    - 2.4 GB/sec transfer rate.
  - Memory controller: up to 15 outstanding load/store requests, with out-of-order completion.
  - Cache tags for L2 on chip to support cache coherency and snooping.
  - System interface on each chip (not shown in diagram)
    - Connects to System interconnect.
      - Connects to I/O and other CPUs.
  - 29 million transistors.
  - 1368 pins
Sun Example (continued):

- Sun UltraSPARC IV: available February 2004
  - Two UltraSPARC III processor cores on a single chip.
  - 1369 pins (almost pin-compatible).
  - Each core has its own L1 Data and L1 Instruction cache.
  - L2 Cache not on the chip.
  - L2 Tags are on the chip; each core has its own copy.
Sun Example (continued):

  - Each core has L1 data and L1 instruction cache.
    - L1 instruction cache: 64 KB, 64-byte line size.
    - L1 data cache: 64 KB (same as before). Uses a “write-through” policy to maintain cache coherency.
  - Chip has on-board L2 Cache, both Tag and Data.
    - Shared by the two cores.
    - Reduced from 16 MB to 2MB.
    - One read or write request every 2 clock cycles.
    - Uses “copy-back” policy on writes.