Caches

Hiding Memory Access Times
Memory

- Main Memory (DRAM) much slower than processors
- Secondary memory even slower
- Gaps have widened over the past fifteen years, continues to widen
- Memory hierarchy used to hide memory access latency
Memory Hierarchy

Cache on chip

Second Level Cache

Third Level Cache

Main Memory

Secondary Memory (Hard Disk)

SRAM

DRAM

Flash Memory
## Memory Technology (2008)

<table>
<thead>
<tr>
<th>Type</th>
<th>Speed (access time)</th>
<th>cost per GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>0.5-2.5 nanosecs</td>
<td>$2-5,000</td>
</tr>
<tr>
<td>DRAM</td>
<td>50-70 nanosecs</td>
<td>$20-$75</td>
</tr>
<tr>
<td>Mag Disk</td>
<td>5-20 x 10^6 ns</td>
<td>$0.20-$2.00</td>
</tr>
<tr>
<td>Flash(NOR)</td>
<td>80-10,000 ns</td>
<td>$65</td>
</tr>
<tr>
<td>Flash(NAND)</td>
<td>25,000-250,000 ns</td>
<td>$4</td>
</tr>
</tbody>
</table>
Why does a memory hierarchy work?

1. Spatial locality
   - Items to be used next are likely to be near items just used
     - Code is sequential
     - Arrays and records are contiguous in memory

2. Temporal Locality
   - Items just used are likely to be used again, soon
     - Loops in code
     - Multiple accesses to same data items
Structure of a memory hierarchy

- Information in memory is moved in blocks
  - Usually more than one word at a time
  - Called “blocks”, “lines”, or “pages”

- Location in unit
  - Determined by some of the address bits
  - Anywhere in the set of blocks

- Address of a word/byte is split
  - Upper bits are the tag and are stored with the block
  - Middle bits address the line or block where stored
  - Lower bits address the word within the block and byte within the word
Structure of a memory hierarchy

- Items in a higher level of hierarchy (closer to the processor) are also in the lower level.
- If items are changed in the higher level, then the lower level must also be changed at some point.
- When space in the higher level runs out, must have a rule for removing items to make more space.
Memory Level Issues

- Where does true image of data/code reside?
  - Higher level contains working copies, true image is in lower level of memory
    - But higher level may contain the true image for a while until lower level is updated
    - May take time for a changed value to percolate down to main memory level
  - Virtual memory (main memory vs hard disk)
    - True image may be in main memory or in secondary memory -- really a point of view
Terminology for levels of Memory

- Registers – part of the processor
- Main memory – active memory in machine
  - Typically DRAM which loses image when off
- Cache – memory between main and processor
  - Current processors have two or three levels of cache
  - Levels 1 and 2 usually on chip, 3 sometimes on chip
- Virtual memory
  - Stored on hard disk
  - Used to make available memory larger than physical
  - Used to provide separate memory images for separate processes
Cache Issues

- How is cache organized and addressed?
- When cache is written to, how is memory image updated?
- When cache is full and new items need to be put in the cache -- what is removed and replaced?
256 Byte Cache (64 4-byte words)

Each Cache “line” or “block” holds one word (4 bytes)

Byte in cache is addressed by lowest two bits of address

Cache line is addressed by next six bits in address

Each Cache line has a “tag” matching the high 24 bits of the memory address
Address  1100 0000 1010 0000 0000 0110 0010 1000

<table>
<thead>
<tr>
<th>line address</th>
<th>tag</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

11110
11111
Address  1100 0000 1010 0000 0000 0110 0010 1000

Byte Address

<table>
<thead>
<tr>
<th>line address</th>
<th>tag</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001010</td>
<td>1100</td>
<td>0000</td>
<td>1010</td>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

111110
111111
Cache Access

1. Find Cache line address (bits 2 - 7)
2. Compare tag to high 24 bits
   - if matched, cache hit
     » read word or find Byte address, read or write item
   - if not matched, cache miss, go to memory
     » for a read: retrieve item and write to cache, then use
     » for a write: write to memory and to cache
3. Direct mapped cache -- every address can only go to one cache line!
4. What happens when cache is written to?
Write Policy

- **Write Through**
  - Write to memory and to cache
  - Time to write to memory could delay instruction
    » write buffer can hide this latency

- **Write Back (also called Copy Back)**
  - Write only to cache
    » mark cache line as “dirty”, using an additional bit
  - When cache line is replaced, if dirty, then write back to memory
Accelerating Memory Access

- **Bus Bandwidth limited**
  - Wider bus, or burst mode

- **Memory width limited**
  - Wider memory access

- **Memory Address limited**
  - Burst mode access
    - one address to retrieve several successive words from memory
Improving memory to cache throughput

a. One-word-wide memory organization

b. Wider memory organization

c. Interleaved memory organization
How can Cache take advantage of faster memory access?

Store more than one word at a time on each “line” in the cache
  – Any cache miss brings the whole line containing the item into the cache

Takes advantage of spatial locality
  – next item needed is likely to be at the next address
Cache with multi-word line

- 256 Byte cache -- 64 4-byte words
- Each block (line) contains four words (16 bytes)
  - 2 bits to address byte in word
  - 2 bits to address word in line
- Cache contains sixteen four-word blocks
  - 4 bits to address cache block (line)
- Each cache line has tag field for upper 24 bits of address
Address  1100 0000 1010 0000 0000 0110 0010 1000

<table>
<thead>
<tr>
<th>line address</th>
<th>tag</th>
<th>Byte Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Word Address

00 01 10 11
Instruction Cache  Hit / Miss

◆ Hit or Miss:
  – Instruction is fetched from Cache and placed in Pipeline buffer register
  – PC is latched into Memory Address Register
◆ Hit:
  – Control sees hit, execution continues
  – Mem Addr unused
Instruction Cache  Hit / Miss

◆ Miss
  – Control sees miss, execution stalls
  – PC reset to PC - 4
  – Values fetched from registers are unused
  – Memory Read cycle started, using Mem Addr

◆ Memory Read completes
  – Value stored in cache, new tag written
  – Instruction execution restarts, cache hit
Set Associative Cache

- Direct Mapped Cache
  - Misses caused by collisions -- two address with same cache line

- Set Associative
  - Two or more (power of 2) lines for each address
  - More than one item with same cache line address can be in cache
  - Check means tags for all lines in set must be checked, one which matches yields hit, if none match, a miss
Two-way set associative cache

<table>
<thead>
<tr>
<th>line address</th>
<th>tag</th>
<th>Word Address</th>
<th>Byte Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td></td>
<td>00 01 10 11</td>
<td>00 01 10 11</td>
</tr>
<tr>
<td>001</td>
<td></td>
<td>00 01 10 11</td>
<td>00 01 10 11</td>
</tr>
<tr>
<td>010</td>
<td></td>
<td>00 01 10 11</td>
<td>00 01 10 11</td>
</tr>
<tr>
<td>011</td>
<td></td>
<td>00 01 10 11</td>
<td>00 01 10 11</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>00 01 10 11</td>
<td>00 01 10 11</td>
</tr>
<tr>
<td>101</td>
<td></td>
<td>00 01 10 11</td>
<td>00 01 10 11</td>
</tr>
<tr>
<td>110</td>
<td></td>
<td>00 01 10 11</td>
<td>00 01 10 11</td>
</tr>
<tr>
<td>111</td>
<td></td>
<td>00 01 10 11</td>
<td>00 01 10 11</td>
</tr>
</tbody>
</table>
4-way Set Associative (from book)
Cache Summary - types

- **Direct Mapped**
  - Each line in cache takes one address
  - Line size may accommodate several words

- **Set Associative**
  - Sets of lines serve the same address
  - Needs replacement policy for which line to purge when set is full
  - More flexible, but more complex
  - Fully Associative (one set)
Cache Summary

- **Cache Hit**
  - Item is found in the cache
  - CPU continues at full speed
  - Need to verify valid and tag match

- **Cache Miss**
  - Item must be retrieved from memory
  - Whole Cache line is retrieved
  - CPU stalls for memory access
    - Out-or-order execution?
    - Switch to another thread?
Cache Summary

- **Write Policies**
  - Write Through (always write to memory)
  - Write Back (uses “dirty” bit)
- **Associative Cache Replacement Policy**
  - LRU (Least Recently Used)
  - Random