Full-Associativity with CAMs

- **CAM**: content addressable memory
  - Array of words with built-in comparators
  - Matchlines instead of bitlines
  - Output is “one-hot” encoding of match

- **FA cache?**
  - Tags as CAM
  - Data as RAM

- **Hardware is not software**
  - Example I: parallel computation with carry select adder
  - Example II: parallel search with CAM
    - No such thing as software CAM
CAM Circuit

- **Matchlines** (correspond to bitlines in SRAM): inputs
- **Wordlines**: outputs

- Two phase match
  - Phase I: $clk=1$, pre-charge wordlines to 1
  - Phase II: $clk=0$, enable matchlines, non-matched bits dis-charge wordlines
CAM Circuit In Action

Phase I: clk=1
- Pre-charge wordlines to 1
CAM Circuit In Action

- Phase I: clk=0
  - Enable matchlines (notice, match bits are flipped)
  - Any non-matching bit discharges entire wordline
    - Implicitly ANDs all bit matches (NORs all bit non-matches)
  - Similar technique for doing a fast OR for hit detection

Looking for match with 01
CAM Upshot

- CAMs are effective but expensive
  - Matchlines are very expensive (for nasty circuit-level reasons)
- CAMs are used but only for 16 or 32 way (max) associativity
  - See an example soon
- Not for 1024-way associativity
  - No good way of doing something like that
  + No real need for it either
Analyzing Cache Misses: 3C Model

• Divide cache misses into three categories
  • **Compulsory (cold):** never seen this address before
    • Easy to identify
  • **Capacity:** miss caused because cache is too small
    • Consecutive accesses to block separated by accesses to at least
      \( N \) other distinct blocks where \( N \) is number of frames in cache
  • **Conflict:** miss caused because cache associativity is too low
    • All other misses
# Cache Performance Simulation

- **Parameters:** 8-bit addresses, 32B cache, 4B blocks
  - Initial contents: 0000, 0010, 0020, 0030, 0100, 0110, 0120, 0130
  - Initial blocks accessed in increasing order

## Cache contents

<table>
<thead>
<tr>
<th>Cache contents</th>
<th>Address</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000, 0010, 0020, 0030, 0100, 0110, 0120, 0130</td>
<td>3020</td>
<td>Miss (compulsory)</td>
</tr>
<tr>
<td>0000, 0010, <strong>3020</strong>, 0030, 0100, 0110, 0120, 0130</td>
<td>3030</td>
<td>Miss (compulsory)</td>
</tr>
<tr>
<td>0000, 0010, 3020, <strong>3030</strong>, 0100, 0110, 0120, 0130</td>
<td>2100</td>
<td>Miss (compulsory)</td>
</tr>
<tr>
<td>0000, 0010, 3020, 3030, <strong>2100</strong>, 0110, 0120, 0130</td>
<td>0012</td>
<td>Hit</td>
</tr>
<tr>
<td>0000, 0010, 3020, 3030, 2100, 0110, 0120, 0130</td>
<td>0020</td>
<td>Miss (capacity)</td>
</tr>
<tr>
<td>0000, 0010, <strong>0020</strong>, 3030, 2100, 0110, 0120, 0130</td>
<td>0030</td>
<td>Miss (capacity)</td>
</tr>
<tr>
<td>0000, 0010, 0020, <strong>0030</strong>, 2100, 0110, 0120, 0130</td>
<td>0110</td>
<td>Hit</td>
</tr>
<tr>
<td>0000, 0010, 0020, 0030, 2100, 0110, 0120, 0130</td>
<td>0100</td>
<td>Miss (capacity)</td>
</tr>
<tr>
<td>0000, 1010, 0020, 0030, <strong>0100</strong>, 0110, 0120, 0130</td>
<td>2100</td>
<td>Miss (conflict)</td>
</tr>
<tr>
<td>1000, 1010, 0020, 0030, <strong>2100</strong>, 0110, 0120, 0130</td>
<td>3020</td>
<td>Miss (capacity)</td>
</tr>
</tbody>
</table>
- **Associativity** (increase)
  - + Decreases conflict misses
  - - Increases $t_{\text{hit}}$
- **Block size** (increase)
  - - Increases conflict misses
  - + Decreases compulsory misses
  - ± Increases or decreases capacity misses
  - • Negligible effect on $t_{\text{hit}}$
- **Capacity** (increase)
  - + Decreases capacity misses
  - - Increases $t_{\text{hit}}$
Two (of many possible) Optimizations

- **Victim buffer**: for conflict misses
- **Prefetching**: for capacity/compulsory misses
Victim Buffer

- Conflict misses: not enough associativity
  - High-associativity is expensive, but also rarely needed
    - 3 blocks mapping to same 2-way set and accessed (ABC)*

- Victim buffer (VB): small FA cache (e.g., 4 entries)
  - Sits on I$/D$ fill path
  - VB is small $\rightarrow$ very fast
  - Blocks kicked out of I$/D$ placed in VB
  - On miss, check VB: hit? Place block back in I$/D$
  - 4 extra ways, shared among all sets
    - Only a few sets will need it at any given time
    - Very effective in practice
Prefetching

- **Prefetching**: put blocks in cache proactively/speculatively
  - Key: anticipate upcoming miss addresses accurately
  - Can do in software or hardware

- Simple example: **next block prefetching**
  - Miss on address $X \rightarrow$ anticipate miss on $X + \text{block-size}$
  - Works for insns: sequential execution
  - Works for data: arrays

- **Timeliness**: initiate prefetches sufficiently in advance
- **Accuracy**: don’t evict useful data
Write Issues

- So far we have looked at reading from cache (loads)
- What about writing into cache (stores)?

- Several new issues
  - Tag/data access
  - Write-through vs. write-back
  - Write-allocate vs. write-not-allocate
Tag/Data Access

- **Reads**: read tag and data in parallel
  - Tag mis-match → data is garbage (OK)
- **Writes**: read tag, write data in parallel?
  - Tag mis-match → clobbered data (oops)
  - For SA cache, which way is written?

- **Writes are a pipelined 2 cycle process**
  - Cycle 1: match tag
  - Cycle 2: write to matching way

- Diagram showing the process of tag/data access.
Tag/Data Access

- Cycle 1: check tag
  - Hit? Write data next cycle
  - Miss? We’ll get to this in a few slides ...
Tag/Data Access

- Cycle 2: write data
Write-Through vs. Write-Back

• When to propagate new value to (lower level) memory?
  • **Write-through**: immediately
    + Conceptually simpler
    + Uniform latency on misses
    − Requires additional bus bandwidth
  • **Write-back**: when block is replaced
    • Requires additional “dirty” bit per block
    + Minimal bus bandwidth
      • Only write back dirty blocks
    − Non-uniform miss latency
      • Clean miss: one transaction with lower level (fill)
      • Dirty miss: two transactions (writeback & fill)
Write-allocate vs. Write-non-allocate

- What to do on a write miss?
  - **Write-allocate**: read block from lower level, write value into it
    + Decreases read misses
    - Requires additional bandwidth
  - Use with write-back
  - **Write-non-allocate**: just write to next level
    - Potentially more read misses
    + Uses less bandwidth
  - Use with write-through
Write Buffer

- **Write buffer**: between cache and memory
  - Write-through cache? Helps with store misses
    + Write to buffer to avoid waiting for memory
  - Store misses become store hits
  - Write-back cache? Helps with dirty misses
    + Allows you to do read (important part) first
      1. Write dirty block to buffer
      2. Read new block from memory to cache
      3. Write buffer contents to memory
Typical Processor Cache Hierarchy

- First level caches: optimized for $t_{hit}$ and parallel access
  - Insns and data in separate caches ($I$, $D$)
  - Capacity: 8–64KB, block size: 16–64B, associativity: 1–4
  - Other: write-through or write-back
  - $t_{hit}$: 1–4 cycles
- Second level cache ($L2$): optimized for $\%_{miss}$
  - Insns and data in one cache for better utilization
  - Capacity: 128KB–1MB, block size: 64–256B, associativity: 4–16
  - Other: write-back
  - $t_{hit}$: 10–20 cycles
- Third level caches ($L3$): also optimized for $\%_{miss}$
  - Capacity: 1–8MB
  - $t_{hit}$: 30 cycles
Performance Calculation Example

- **Parameters**
  - Reference stream: 20% stores, 80% loads
  - L1 D$: \( t_{hit} = 1 \text{ns}, \ %_{miss} = 5\%, \ \) write-through + write-buffer
  - L2: \( t_{hit} = 10 \text{ns}, \ %_{miss} = 20\%, \ \) write-back, 50% dirty blocks
  - Main memory: \( t_{hit} = 50 \text{ns}, \ %_{miss} = 0\% \)
- **What is \( t_{avgL1D} \) without an L2?**
  - Write-through+write-buffer means all stores effectively hit
  - \( t_{missL1D} = t_{hitM} \)
  - \( t_{avgL1D} = t_{hitL1D} + %_{loads} \times %_{missL1D} \times t_{hitM} = 1\text{ns} + (0.8 \times 0.05 \times 50\text{ns}) = 3\text{ns} \)
- **What is \( t_{avgD} \) with an L2?**
  - \( t_{missL1D} = t_{avgL2} \)
  - Write-back (no buffer) means dirty misses cost double
  - \( t_{avgL2} = t_{hitL2} + (1 + %_{dirty}) \times %_{missL2} \times t_{hitM} = 10\text{ns} + (1.5 \times 0.2 \times 50\text{ns}) = 25\text{ns} \)
  - \( t_{avgL1D} = t_{hitL1D} + %_{loads} \times %_{missL1D} \times t_{avgL2} = 1\text{ns} + (0.8 \times 0.05 \times 25\text{ns}) = 2\text{ns} \)
Summary

• Average access time of a memory component
  • $t_{avg} = t_{hit} + \%_{miss} \cdot t_{miss}$
  • Hard to get low $t_{hit}$ and $\%_{miss}$ in one structure $\rightarrow$ hierarchy

• Memory hierarchy
  • Cache (SRAM) $\rightarrow$ memory (DRAM) $\rightarrow$ swap (Disk)
  • Smaller, faster, more expensive $\rightarrow$ bigger, slower, cheaper

• SRAM
  • Analog technology for implementing big storage arrays
  • Cross-coupled inverters + bitlines + wordlines
  • Delay $\sim \sqrt{\#\text{bits} \cdot \#\text{ports}}$
Summary, cont’d

• Cache ABCs
  • Capacity, associativity, block size
  • 3C miss model: compulsory, capacity, conflict

• Some optimizations
  • Victim buffer for conflict misses
  • Prefetching for capacity, compulsory misses

• Write issues
  • Pipelined tag/data access
  • Write-back vs. write-through/write-allocate vs. write-no-allocate
  • Write buffer

Next Course Unit: Main Memory