Lecture Topics

- Today: The Memory Hierarchy (P&H 5.1-5.11)
- Next: continued

Announcements

- Calendar revised
- Milestone #6 (due 3/16)
- Milestone #7 (due 3/23)
- Milestone #8 (due 4/6)
Locality of Reference

- Observation: a program references only a small portion of its address space at any particular moment in time

- We can exploit locality of reference: keep the currently used instructions and data objects at the top of the memory hierarchy and keep everything else at the bottom of the hierarchy; move items within the hierarchy as needed

Locality of Reference

- Temporal locality (locality in time): if an item is referenced, it is likely to be referenced again in the near future

- Spatial locality (locality in space): if an item is referenced, nearby items are likely to be referenced in the near future

- Both instructions and data exhibit both types of locality of reference
The Memory Hierarchy

- There is limited storage in the processor (registers), so there must be additional memory available.

- Ideal scenario: unlimited amount of fast memory to hold instructions and data.

- Not practical, so a memory hierarchy is used to provide the illusion of a large amount of fast memory.

---

The Memory Hierarchy

- Top:
  - fast access
  - small capacity

- Bottom:
  - slow access
  - large capacity

- Illusion:
  - speed of level 1
  - capacity of level n
The Memory Hierarchy

- Each pair of adjacent levels can be viewed as an upper level (closer to processor) and lower level
  - Upper: faster, smaller
  - Lower: slower, larger
  - Data transferred: block (or line)
Cache Basics

- Level between processor and primary storage (RAM) – introduced because of gap between speed of registers and RAM

- As gap has increased, additional levels introduced:
  - Level 1 (instruction cache and data cache)
  - Level 2
  - Level 3

Read Operation

Read: copy data from RAM to CPU

Check cache first – if desired item is already present in the cache, simply copy item from cache to CPU

If desired item is not already present in the cache, copy a block (item and its neighbors) from RAM to the cache and copy the item to the CPU
Write Operation

Write: copy data from CPU to RAM

Check cache first – if desired item is already present in the cache, simply copy item from CPU to cache

If desired item is not already present in the cache, copy item (and its neighbors) from RAM to the cache and copy the item from the CPU

Write Policies

After a store instruction, cache and RAM are inconsistent: contents of block in cache and RAM are different

Two strategies:
• Write through
• Write back
Direct Mapped Cache

- With direct mapping, each RAM slot maps to a specific cache slot.
- Since RAM is much larger than cache, multiple RAM slots map to the same cache slot.
- Mapping formula:
  \[(\text{address}) \mod (\text{number of cache slots})\]

Example

- Addresses are 32 bits.
- Cache characteristics:
  - Direct mapped
  - 1 word (4 bytes) per block
  - 1024 slots
- Address subdivided into 3 fields:
  - 20 + 10 + 2
To exploit spatial locality, a cache slot must hold more than one item (one word).

- Block size is always a multiple of 2 (use least significant bits of address to identify specific byte within block).
- Typical block sizes are 32 to 256 bytes.
Example: Intrinsity FastMATH

- Embedded processor based on MIPS

- Cache characteristics:
  - 256 slots
  - 16 words (64 bytes) per block

- Miss rates for benchmark:
  - instruction cache 0.4%
  - data cache 11.4%
  - Effective combined miss rate: 3.2%
Direct Mapped Cache Example

- Address (32 bits) viewed as three fields:
  - Byte offset: 8 bits to identify byte within block
  - Line: 4 bits to identify cache line
  - Tag: 20 bits (remaining bits)

- Example: FFF7C408

Example (2)

- How many lines in the cache?
  \[ 2^4 = 16 \text{ lines} \]
- How many bytes in one block?
  \[ 2^8 = 256 \text{ bytes} \]
- How many control bits in one line?
  \[ V + M + \text{Tag} = 1 + 1 + 20 = 22 \text{ bits} \]
- How many total bits in one line?
  \[ \text{control + data} = 22 + 2048 = 2070 \text{ bits} \]
Example (3)

<table>
<thead>
<tr>
<th>I</th>
<th>V</th>
<th>M</th>
<th>Tag</th>
<th>I</th>
<th>V</th>
<th>M</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>FF61</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>004A</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0014</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>0028</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>003A</td>
<td>A</td>
<td>1</td>
<td>0</td>
<td>0028</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>FF53</td>
<td>B</td>
<td>1</td>
<td>1</td>
<td>FFFC</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>FFFC</td>
<td>C</td>
<td>0</td>
<td>1</td>
<td>00EA</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0014</td>
<td>D</td>
<td>1</td>
<td>0</td>
<td>0028</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0014</td>
<td>E</td>
<td>1</td>
<td>1</td>
<td>003A</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0014</td>
<td>F</td>
<td>1</td>
<td>1</td>
<td>003A</td>
</tr>
</tbody>
</table>

Example (4)

- Index – line number (not stored)
- Valid bit (V) – initially 0, set to 1 when that entry in the cache is in use
- Modified bit (M) – set to 1 when at least one byte in the block has been modified by a "write" operation (aka dirty bit)
- Tag bits – compared to tag bits from address
- Block – 256 bytes (not shown)
Example (5)

- Consider the cache entry at index 4:
  
  \[
  \begin{array}{llll}
  & 1 & 1 & \text{FFF7C} \\
  \end{array}
  \]

  - What are the addresses of the first and last bytes in that cache entry?
    
    - first byte: \textbf{FFF7C400}
    - last byte: \textbf{FFF7C4FF}

  - Has the contents of that cache block been modified?
    
    Yes, \( M = 1 \)

Example (6)

- Request to read from address \textbf{00028A14}

  Line in address is A, so check cache line at index A:

  \[
  \begin{array}{llll}
  & 1 & 0 & \text{00028} \\
  \end{array}
  \]

  Hit: \( V = 1 \) and tag in cache line matches tag in address

  Transfer 4 bytes (14, 15, 16, 17) from cache block to CPU
Example (7)

- Request to read from address **0007260C**
  
  Line in address is 6, so check index 6:
  
  \[
  \begin{array}{ccc}
  0 & 0 & 00014 \\
  \end{array}
  \]
  
  Miss: \( V = 0 \)
  
  Transfer 256 bytes from RAM to cache
  
  Set V bit to 1
  
  Set M bit to 0
  
  Set tag to 00072
  
  Transfer 4 bytes (0C, 0D, 0E, 0F) from cache to CPU

Example (8)

- Request to write to address **0003AED8**
  
  Line in address is E, so check index E:
  
  \[
  \begin{array}{ccc}
  1 & 1 & 0003A \\
  \end{array}
  \]
  
  Hit: \( V = 1 \) and tag in cache line matches tag in address
  
  Transfer 4 bytes from CPU to cache (D8, D9, DA, DB)
  
  Set M bit to 1
  
  Note: some of the 256 bytes in the cache block are no longer the same as the corresponding bytes in RAM (copy block to RAM later)
Example (9)

- Request to write to address \texttt{0003A344}

  Line in address is 3, so check index 3:

  \begin{center}
  \begin{tabular}{c c}
    0 & 1 \\
    FF593
  \end{tabular}
  \end{center}

  Miss: \( V = 0 \)

  Transfer 256 bytes from RAM to cache

  Set \( V \) bit to 1

  Set \( M \) bit to 1

  Set tag to \texttt{0003A}

  Transfer 4 bytes (44, 45, 46, 47) from CPU to cache

Example (10)

- Request to read from address \texttt{002C5934}

  Line in address is 9, so check index 9:

  \begin{center}
  \begin{tabular}{c c}
    1 & 0 \\
    00028
  \end{tabular}
  \end{center}

  Miss: \( V = 1 \), but tags don’t match

  Transfer 256 bytes from RAM to cache

  Set \( V \) bit to 1

  Set \( M \) bit to 0

  Set tag to \texttt{002C5}

  Transfer 4 bytes (34, 35, 36, 37) from cache to CPU
Example (11)

- Request to read from address 002D1F98

  Line in address is F, so check index F:

  1 1 0003A

  Miss: $V = 1$, but tags don’t match

  Transfer 256 bytes from cache to RAM (write back)
  Transfer 256 bytes from RAM to cache
  Set $V$ bit to 1
  Set $M$ bit to 0
  Set tag to 002D1
  Transfer 4 bytes (98, 99, 9A, 9B) from cache to CPU

Fully Associative Cache

- Direct mapped cache is easy to implement, but is inflexible: a block from RAM can only be stored in one slot in the cache

- Fully associative cache allows a block to be placed in any cache slot
  - benefit: more flexible placement
  - cost: more expensive to implement (must check all tags simultaneously)
Fully Associative Cache Example

- Address (32 bits) viewed as two fields:
  - Byte offset: 8 bits to identify byte within block
  - Tag: 24 bits (remaining bits)

- Example: FFF7C408
  
  \[
  \begin{array}{c}
  11111111111101111100010000001000
  
  \end{array}
  \]

Example (2)

- Assume 16 lines in the cache
- How many bytes in one block?
  \[2^8 = 256\text{ bytes}\]
- How many control bits in one line?
  \[V + M + \text{Tag} = 1 + 1 + 24 = 26\text{ bits}\]
- How many total bits in one line?
  \[\text{control + data} = 26 + 2048 = 2074\text{ bits}\]
### Example (3)

<table>
<thead>
<tr>
<th>I</th>
<th>V</th>
<th>M</th>
<th>Tag</th>
<th>I</th>
<th>V</th>
<th>M</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>FF7814</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>0004A7</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>000146</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>000286</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0003AF</td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>000289</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0003AC</td>
<td>B</td>
<td>0</td>
<td>1</td>
<td>FFF7C0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>FF781C</td>
<td>C</td>
<td>0</td>
<td>0</td>
<td>000146</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>000142</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>000287</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>000000</td>
<td>E</td>
<td>0</td>
<td>0</td>
<td>000000</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>000000</td>
<td>F</td>
<td>0</td>
<td>0</td>
<td>000000</td>
</tr>
</tbody>
</table>

### Example (4)

- **Index** – not derived from address
- **Valid bit (V)** – initially 0, set to 1 when that entry in the cache is in use
- **Modified bit (M)** – set to 1 when at least one byte in the block has been modified by a "write" operation (aka dirty bit)
- **Tag bits** – compared to tag bits from address
- **Block** – 256 bytes (not shown)
Example (5)

- Consider the cache entry at index 0:
  \[
  1 \quad 0 \quad \text{FF7814}
  \]
  - What are the addresses of the first and last bytes in that cache entry?
    - first byte: \text{FF781400}
    - last byte: \text{FF7814FF}
  - Has the contents of that cache block been modified?
    - No, \( M = 0 \)

Example (6)

- Consider a request to read from address \text{0003AC64}
  
  Search all cache lines simultaneously, found at 3:
  \[
  1 \quad 0 \quad \text{0003AC}
  \]
  - Hit: \( V = 1 \) and tag in cache line matches tag in address
    - Transfer 4 bytes (64, 65, 66, 67) from cache block to CPU
Example (7)

- Request to read from address 0034560C

Search all cache lines simultaneously, not found (miss)

Transfer 256 bytes from RAM to cache
Set V bit to 1
Set M bit to 0
Set tag to 003456
Transfer 4 bytes (0C, 0D, 0E, 0F) from cache to CPU

Note: where should the line be placed in the cache?

Set Associative Cache

- Fully associative cache is expensive to implement, only practical for small caches

- Set associative cache uses direct mapping to identify a set; block can be placed anywhere in the set
  - 2-way set associative (2 entries per slot)
  - 4-way set associative (4 entries per slot)
  - 8-way set associative (8 entries per slot)
Example

- 4-way set associative
  - 256 cache slots
  - each slot has 4 separate elements
  - each element has V bit, tag and data block

- Added cost (compared to direct mapped):
  - 4 comparators
  - 4-to-1 MUX
Comparison

Fully associative:
block 12 can go anywhere

Direct mapped:
block 12 can go only into block 4
(12 % 4 = 0)

Set associative:
block 12 can go anywhere in set 0
(12 % 3 = 0)

One-way set associative
(direct mapped)

<table>
<thead>
<tr>
<th>Block</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

Two-way set associative

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

Four-way set associative

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

Eight-way set associative (fully 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>
</tr>
</thead>
</table>
Summary: Four Questions

1) Where can a block be placed in the upper level? (block placement)

2) How is a block found if it is in the upper level? (block identification)

3) Which block should be replaced on a miss? (block replacement)

4) What happens on a write? (write strategy)

Block Placement

- Direct mapped – exactly one location where a given block can be placed in the upper level
- Fully associative – block can be placed anywhere in the upper level
- Set associative – block can only be placed in a restricted set of locations (block mapped onto a set, and then the block is placed anywhere in that set)
Block Identification

- Valid bit used to indicate whether an entry contains valid data or not
- Tag bits from address compared to tag bits in entry
- All possible locations searched at once:
  - direct mapped – only one location possible
  - fully associative – all locations possible
  - set associative – subset of locations possible

Block Replacement

- Hardware must choose victim (block to be evicted)
- Direct mapped: only one choice
- Full associative and set associative: choose one of multiple entries
  - Random
  - LRU (least recently used)
  - FIFO (first in, first out)
Write Strategy

- Store operations modify data objects (about 10% of instructions in a typical program)

- Write back – block in lower level updated when block is evicted from upper level
  - block may be updated multiple times
  - modified bit used to flag modified blocks

Write Strategy

- Write through – block in upper level and in lower level updated together

- When using "write through", there are two alternatives for write misses
  - write allocate: the block is copied from RAM into the cache, updated, then copied back to RAM
  - write around: the block is just updated in RAM (it is not copied from RAM to cache)