Caches II
Recap: You Are Here!

**Software**

- **Parallel Requests**
  Assigned to computer
  e.g., Search “Katz”

- **Parallel Threads**
  Assigned to core
  e.g., Lookup, Ads

- **Parallel Instructions**
  >1 instruction @ one time
  e.g., 5 pipelined instructions

- **Parallel Data**
  >1 data item @ one time
  e.g., Add of 4 pairs of words

- **Hardware descriptions**
  All gates @ one time

- **Programming Languages**

---

**Hardware**

- **Warehouse Scale Computer**
- **Harness Parallelism & Achieve High Performance**

**Today’s Lecture**

- Smart Phone
- Main Memory
- Instruction Unit(s)
  - $A_0 + B_0$
  - $A_1 + B_1$
  - $A_2 + B_2$
  - $A_3 + B_3$
- Functional Unit(s)
- Logic Gates
Recap: Processor Address Fields used by Cache Controller

- **Block Offset**: Byte address within block
- **Set Index**: Selects which set
- **Tag**: Remaining portion of processor address

- Size of Index = \( \log_2 \) (number of sets)
- Size of Tag = Address size – Size of Index – \( \log_2 \) (number of bytes/block)
Recap: What is limit to number of sets?

• For a given total number of blocks, we can save more comparators if have more than 2 sets

• Limit: As Many Sets as Cache Blocks => only one block per set – only needs one comparator!

• Called “Direct-Mapped” Design
Recap: Cache Names for Each Organization

- “Fully Associative”: Block can go anywhere
  - First design in lecture
  - Note: No Index field, but 1 comparator/block
- “Direct Mapped”: Block goes one place
  - Note: Only 1 comparator
  - Number of sets = number of blocks
- “N-way Set Associative”: N places for a block
  - Number of sets = number of blocks / N
  - N comparators
  - *Fully Associative*: $N = \text{number of blocks}$
  - *Direct Mapped*: $N = 1$
Memory Block-addressing example

<table>
<thead>
<tr>
<th>address</th>
<th>8</th>
<th>address</th>
<th>8</th>
<th>address</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td></td>
<td>00000</td>
<td></td>
<td>00000</td>
<td></td>
</tr>
<tr>
<td>00001</td>
<td></td>
<td>00001</td>
<td></td>
<td>00001</td>
<td></td>
</tr>
<tr>
<td>00010</td>
<td></td>
<td>00010</td>
<td></td>
<td>00010</td>
<td></td>
</tr>
<tr>
<td>00011</td>
<td></td>
<td>00011</td>
<td></td>
<td>00011</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td></td>
<td>00100</td>
<td></td>
<td>00100</td>
<td></td>
</tr>
<tr>
<td>00101</td>
<td></td>
<td>00101</td>
<td></td>
<td>00101</td>
<td></td>
</tr>
<tr>
<td>00110</td>
<td></td>
<td>00110</td>
<td></td>
<td>00110</td>
<td></td>
</tr>
<tr>
<td>00111</td>
<td></td>
<td>00111</td>
<td></td>
<td>00111</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td></td>
<td>01000</td>
<td></td>
<td>01000</td>
<td></td>
</tr>
<tr>
<td>01001</td>
<td></td>
<td>01001</td>
<td></td>
<td>01001</td>
<td></td>
</tr>
<tr>
<td>01010</td>
<td></td>
<td>01010</td>
<td></td>
<td>01010</td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td></td>
<td>01011</td>
<td></td>
<td>01011</td>
<td></td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td>01100</td>
<td></td>
<td>01100</td>
<td></td>
</tr>
<tr>
<td>01101</td>
<td></td>
<td>01101</td>
<td></td>
<td>01101</td>
<td></td>
</tr>
<tr>
<td>01110</td>
<td></td>
<td>01110</td>
<td></td>
<td>01110</td>
<td></td>
</tr>
<tr>
<td>01111</td>
<td></td>
<td>01111</td>
<td></td>
<td>01111</td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td></td>
<td>10000</td>
<td></td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td>10001</td>
<td></td>
<td>10001</td>
<td></td>
<td>10001</td>
<td></td>
</tr>
<tr>
<td>10010</td>
<td></td>
<td>10010</td>
<td></td>
<td>10010</td>
<td></td>
</tr>
<tr>
<td>10011</td>
<td></td>
<td>10011</td>
<td></td>
<td>10011</td>
<td></td>
</tr>
<tr>
<td>10100</td>
<td></td>
<td>10100</td>
<td></td>
<td>10100</td>
<td></td>
</tr>
<tr>
<td>10101</td>
<td></td>
<td>10101</td>
<td></td>
<td>10101</td>
<td></td>
</tr>
<tr>
<td>10110</td>
<td></td>
<td>10110</td>
<td></td>
<td>10110</td>
<td></td>
</tr>
<tr>
<td>10111</td>
<td></td>
<td>10111</td>
<td></td>
<td>10111</td>
<td></td>
</tr>
<tr>
<td>11000</td>
<td></td>
<td>11000</td>
<td></td>
<td>11000</td>
<td></td>
</tr>
<tr>
<td>11001</td>
<td></td>
<td>11001</td>
<td></td>
<td>11001</td>
<td></td>
</tr>
<tr>
<td>11010</td>
<td></td>
<td>11010</td>
<td></td>
<td>11010</td>
<td></td>
</tr>
<tr>
<td>11011</td>
<td></td>
<td>11011</td>
<td></td>
<td>11011</td>
<td></td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td>11100</td>
<td></td>
<td>11100</td>
<td></td>
</tr>
<tr>
<td>11101</td>
<td></td>
<td>11101</td>
<td></td>
<td>11101</td>
<td></td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td>11110</td>
<td></td>
<td>11110</td>
<td></td>
</tr>
<tr>
<td>11111</td>
<td></td>
<td>11111</td>
<td></td>
<td>11111</td>
<td></td>
</tr>
</tbody>
</table>

2 LSBS are 0

3 LSBS are 0

2/10/16

8-Byte Block

Byte offset in block
Block number aliasing example

12-bit memory addresses, 16 Byte blocks

<table>
<thead>
<tr>
<th>Block #</th>
<th>Block # mod 8</th>
<th>Block # mod 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>82 010100100000</td>
<td>2 010100100000</td>
<td>0 010100100000</td>
</tr>
<tr>
<td>83 010100110000</td>
<td>3 010100110000</td>
<td>1 010100110000</td>
</tr>
<tr>
<td>84 010101000000</td>
<td>4 010101000000</td>
<td>0 010101000000</td>
</tr>
<tr>
<td>85 010101010000</td>
<td>5 010101010000</td>
<td>1 010101010000</td>
</tr>
<tr>
<td>86 010101100000</td>
<td>6 010101100000</td>
<td>0 010101100000</td>
</tr>
<tr>
<td>87 010101110000</td>
<td>7 010101110000</td>
<td>1 010101110000</td>
</tr>
<tr>
<td>88 010110000000</td>
<td>0 010110000000</td>
<td>0 010110000000</td>
</tr>
<tr>
<td>89 010110010000</td>
<td>1 010110010000</td>
<td>1 010110010000</td>
</tr>
<tr>
<td>90 010110100000</td>
<td>2 010110100000</td>
<td>0 010110100000</td>
</tr>
<tr>
<td>91 010110110000</td>
<td>3 010110110000</td>
<td>1 010110110000</td>
</tr>
</tbody>
</table>

3/10/16
Direct Mapped Cache Ex:
Mapping a 6-bit Memory Address

• In example, block size is 4 bytes (1 word)
• Memory and cache blocks always the same size, unit of transfer between memory and cache
• # Memory blocks >> # Cache blocks
  – 16 Memory blocks = 16 words = 64 bytes => 6 bits to address all bytes
  – 4 Cache blocks, 4 bytes (1 word) per block
  – 4 Memory blocks map to each cache block
• Memory block to cache block, aka *index*: middle two bits
• Which memory block is in a given cache block, aka *tag*: top two bits
Caching: A Simple First Example

Q: Where in the cache is the mem block?
Use next 2 low-order memory address bits – the index – to determine which cache block (i.e., modulo the number of blocks in the cache)

Q: Is the memory block in cache?
Compare the cache tag to the high-order 2 memory address bits to tell if the memory block is in the cache (provided valid bit is set)

Cache
Index Valid Tag Data
00
01
10
11

Main Memory

0000xx
0001xx
0010xx
0011xx
0100xx
0101xx
0110xx
0111xx
1000xx
1001xx
1010xx
1011xx
1100xx
1101xx
1110xx
1111xx

One word blocks
Two low order bits (xx) define the byte in the block (32b words)
One More Detail: Valid Bit

- When start a new program, cache does not have valid information for this program
- Need an indicator whether this tag entry is valid for this program
- Add a “valid bit” to the cache tag entry
  - 0 => cache miss, even if by chance, address = tag
  - 1 => cache hit, if processor address = tag
Direct-Mapped Cache Example

- One word blocks, cache size = 1K words (or 4KB)

```
Index | Valid | Tag | Data
0     |       |     |
1     |       |     |
2     |       |     |
.     |       |     |
1021  |       |     |
1022  |       |     |
1023  |
```

- Valid bit ensures something useful in cache for this index
- Compare Tag with upper part of Address to see if a Hit
- Read data from cache instead of memory if a Hit

What kind of locality are we taking advantage of?
Multiword-Block Direct-Mapped Cache

- Four words/block, cache size = 1K words

![Cache Diagram](Image)

Index

Data

Valid

Tag

Word offset

Byte offset

Hit

What kind of locality are we taking advantage of?
Handling Stores with Write-Through

• Store instructions write to memory, changing values
• Need to make sure cache and memory have same values on writes: 2 policies

1) Write-Through Policy: write cache and write through the cache to memory
   – Every write eventually gets to memory
   – Too slow, so include Write Buffer to allow processor to continue once data in Buffer
   – Buffer updates memory in parallel to processor
Write-Through Cache

- Write both values in cache and in memory
- Write buffer stops CPU from stalling if memory cannot keep up
- Write buffer may have multiple entries to absorb bursts of writes
- What if store misses in cache?
Handling Stores with Write-Back

2) Write-Back Policy: write only to cache and then write cache block back to memory when evict block from cache

- Writes collected in cache, only single write to memory per block
- Include bit to see if wrote to block or not, and then only write back if bit is set
  • Called “Dirty” bit (writing makes it “dirty”)
Write-Back Cache

- Store/cache hit, write data in cache *only* & set dirty bit
  - Memory has stale value
- Store/cache miss, read data from memory, then update and set dirty bit
  - “Write-allocate” policy
- On any miss, write back evicted block, only if dirty. Update cache with new block and clear dirty bit.
Write-Through vs. Write-Back

• Write-Through:
  – Simpler control logic
  – More predictable timing simplifies processor control logic
  – Easier to make reliable, since memory always has copy of data (big idea: Redundancy!)

• Write-Back
  – More complex control logic
  – More variable timing (0,1,2 memory accesses per cache access)
  – Usually reduces write traffic
  – Harder to make reliable, sometimes cache has only copy of data
Write Policy Choices

• Cache hit:
  – **write through**: writes both cache & memory on every access
    • Generally higher memory traffic but simpler pipeline & cache design
  – **write back**: writes cache only, memory `written only when dirty entry evicted`
    • A dirty bit per line reduces write-back traffic
    • Must handle 0, 1, or 2 accesses to memory for each load/store

• Cache miss:
  – **no write allocate**: only write to main memory
  – **write allocate** (aka fetch on write): fetch into cache

• Common combinations:
  – write through and no write allocate
  – write back with write allocate
Cache (*Performance*) Terms

- **Hit rate**: fraction of accesses that hit in the cache
- **Miss rate**: $1 - \text{Hit rate}$
- **Miss penalty**: time to replace a block from lower level in memory hierarchy to cache
- **Hit time**: time to access cache memory (including tag comparison)

- Abbreviation: “$” = cache (A Berkeley innovation!)
Average Memory Access Time (AMAT)

- Average Memory Access Time (AMAT) is the average time to access memory considering both hits and misses in the cache.

\[
\text{AMAT} = \text{Time for a hit} + \text{Miss rate} \times \text{Miss penalty}
\]
Clickers/Peer instruction

AMAT = Time for a hit + Miss rate x Miss penalty

Given a 200 psec clock, a miss penalty of 50 clock cycles, a miss rate of 0.02 misses per instruction and a cache hit time of 1 clock cycle, what is AMAT?

☐ A: ≤200 psec

☐ B: 400 psec

☐ C: 600 psec

☐ D: ≥ 800 psec
Example: Direct-Mapped Cache with 4 Single-Word Blocks, Worst-Case Reference String

- Consider the main memory address reference string of word numbers:

  0 4 0 4 0 4 0 4

Start with an empty cache - all blocks initially marked as not valid
Example: Direct-Mapped Cache with 4 Single-Word Blocks, Worst-Case Reference String

- Consider the main memory address reference string of word numbers: 0 4 0 4 0 4 0 4

Start with an empty cache - all blocks initially marked as not valid

- 8 requests, 8 misses
- Ping-pong effect due to conflict misses - two memory locations that map into the same cache block
Alternative Block Placement Schemes

- DM placement: mem block 12 in 8 block cache: only one cache block where mem block 12 can be found—(12 modulo 8) = 4
- SA placement: four sets x 2-ways (8 cache blocks), memory block 12 in set (12 mod 4) = 0; either element of the set
- FA placement: mem block 12 can appear in any cache blocks
Example: 4 Word 2-Way SA $ Same Reference String

• Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid

0 4 0 4 0 4 0 4
Example: 4-Word 2-Way SA $ 
Same Reference String

- Consider the main memory address reference string

Start with an empty cache - all blocks initially marked as not valid

Start with an empty cache - all blocks initially marked as not valid

- 8 requests, 2 misses

- Solves the ping-pong effect in a direct-mapped cache due to conflict misses since now two memory locations that map into the same cache set can co-exist!
Different Organizations of an Eight-Block Cache

Total size of $S$ in blocks is equal to number of sets $\times$ associativity. For fixed $S$ size and fixed block size, increasing associativity decreases number of sets while increasing number of elements per set. With eight blocks, an 8-way set-associative $S$ is same as a fully associative $S$. 
Range of Set-Associative Caches

• For a fixed-size cache and fixed block size, each increase by a factor of two in associativity doubles the number of blocks per set (i.e., the number or ways) and halves the number of sets – decreases the size of the index by 1 bit and increases the size of the tag by 1 bit
Range of Set-Associative Caches

- For a *fixed-size* cache and fixed block size, each increase by a factor of two in associativity doubles the number of blocks per set (i.e., the number or ways) and halves the number of sets – decreases the size of the index by 1 bit and increases the size of the tag by 1 bit
Total Cache Capacity =

Associativity \times \text{ # of sets} \times \text{ block}_size

Bytes = \text{ blocks/}set \times \text{ sets} \times \text{ Bytes/block}

C = N \times S \times B

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

address\_size = \text{ tag\_size} + \text{ index\_size} + \text{ offset\_size}

= \text{ tag\_size} + \log_2(S) + \log_2(B)
Clickers/Peer Instruction

• For a cache with constant total capacity, if we increase the number of ways by a factor of 2, which statement is false:

• A: The number of sets could be doubled
• B: The tag width could decrease
• C: The block size could stay the same
• D: The block size could be halved
• E: Tag width must increase
Total Cache Capacity =

\[ \text{Associativity} \times \# \text{ of sets} \times \text{block\_size} \]

\[ \text{Bytes} = \text{blocks/\textit{set}} \times \text{\textit{sets}} \times \text{Bytes/\textit{block}} \]

\[ C = N \times S \times B \]

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

\[ \text{address\_size} = \text{tag\_size} + \text{index\_size} + \text{offset\_size} \]

\[ = \text{tag\_size} + \log_2(S) + \log_2(B) \]

Clicker Question: C remains constant, S and/or B can change such that

\[ C = 2N \times (SB)' \Rightarrow (SB)' = SB/2 \]

\[ \text{Tag\_size} = \text{address\_size} - (\log_2(S') + \log_2(B')) = \text{address\_size} - \log_2(SB)' \]

\[ = \text{address\_size} - \log_2(SB/2) \]

\[ = \text{address\_size} - (\log_2(SB) - 1) \]
Costs of Set-Associative Caches

• N-way set-associative cache costs
  – N comparators (delay and area)
  – MUX delay (set selection) before data is available
  – Data available after set selection (and Hit/Miss decision).
    DM $: block is available before the Hit/Miss decision
    • In Set-Associative, not possible to just assume a hit and continue and
      recover later if it was a miss

• When miss occurs, which way’s block selected for replacement?
  – Least Recently Used (LRU): one that has been unused the
    longest (principle of temporal locality)
    • Must track when each way’s block was used relative to other blocks in
      the set
    • For 2-way SA $, one bit per set → set to 1 when a block is referenced;
      reset the other way’s bit (i.e., “last used”)
Cache Replacement Policies

• Random Replacement
  – Hardware randomly selects a cache entry

• Least-Recently Used
  – Hardware keeps track of access history
  – Replace the entry that has not been used for the longest time
  – For 2-way set-associative cache, need one bit for LRU replacement

• Example of a Simple “Pseudo” LRU Implementation
  – Assume 64 Fully Associative entries
  – Hardware replacement pointer points to one cache entry
  – Whenever access is made to the entry the pointer points to:
    • Move the pointer to the next entry
  – Otherwise: do not move the pointer
  – (example of “not-most-recently used” replacement policy)
Benefits of Set-Associative Caches

- Largest gains are in going from direct mapped to 2-way (20%+ reduction in miss rate)
Sources of Cache Misses (3 C’s)

• **Compulsory** (cold start, first reference):
  – 1\textsuperscript{st} access to a block, not a lot you can do about it.
    • If running billions of instructions, compulsory misses are insignificant

• **Capacity**:
  – Cache cannot contain all blocks accessed by the program
    • Misses that would not occur with infinite cache

• **Conflict** (collision):
  – Multiple memory locations mapped to same cache set
    • Misses that would not occur with ideal fully associative cache