Cache addressing

- How do you know if something is in the cache? (Q1)
- If it is in the cache, how to find it? (Q2)

- Traditional Memory
  - Given an address, provide the data (has address decoder)

- Associative Memory
  - AKA “Content Addressable Memory”
  - Each line contains the address (or part of it) and the data
**Cache Organization**

- **Fully-associative**: any memory location can be stored anywhere in the cache
  - Cache location and memory address are unrelated

- **Direct-mapped**: each memory location maps onto exactly one cache entry
  - Some of the memory address bit are used to index the cache

- **N-way set-associative**: each memory location can go into one of N sets

**Direct Mapped Cache**

- Simplest mapping is a **direct mapped cache**
- Each memory address is associated with one possible block within the cache
  - Therefore, we only need to look in a single location in the cache for the data if it exists in the cache
Direct mapped cache (assume 1 byte/Block)

- Cache Block 0 can be occupied by data from
  - Memory blocks 0, 4, 8, 12
- Cache Block 1 can be occupied by data from
  - Memory blocks 1, 5, 9, 13
- Cache Block 2 can be occupied by data from
  - Memory blocks 2, 6, 10, 14
- Cache Block 3 can be occupied by data from
  - Memory blocks 3, 7, 11, 15

Direct Mapped Cache – Index and Tag

- **index** determines block in cache
- index = (address) mod (# blocks)
- The number of cache blocks is power of 2 ⇒ cache index is the lower n bits of memory address
  - n = log₂(# blocks)
**Direct Mapped w/Tag**

- **Memory**: Shows memory blocks and cache blocks.
- **Tag**: Determines which memory block occupies cache block.
- **Tag = most significant bits of address**
- **Hit**: Cache tag field = tag bits of address
- **Miss**: Tag field ≠ tag bits of address

**Finding Item within Block**

- In reality, a cache block consists of a number of bytes/words to
  1. Increase cache hit due to locality property and
  2. Reduce the cache miss time

- Mapping: Memory block \( i \) is mapped to cache block with index \( i \mod k \), where \( k \) is the number of blocks in the cache.

- Given an address of item, index tells which block of cache to look in.

- Then, how to find requested item within the cache block?

- Or, equivalently, “What is the byte offset of the item within the cache block?”
Selecting part of a block

- If block size > 1, rightmost bits of index are really the **offset** within the indexed block

<table>
<thead>
<tr>
<th>TAG</th>
<th>INDEX</th>
<th>OFFSET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag to check if have correct block</td>
<td>Index to select a block in cache</td>
<td>Byte offset</td>
</tr>
</tbody>
</table>

- Example: Block size of 8 bytes; select 2\(^{nd}\) word

<table>
<thead>
<tr>
<th>Memory address</th>
<th>Cache Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 01 100</td>
<td>0 1 2 3</td>
</tr>
</tbody>
</table>

Accessing data in a direct mapped cache

- Three types of events:
  - **cache hit**: cache block is valid and contains proper address, so read desired word
  - **cache miss**: nothing in cache in appropriate block, so fetch from memory
  - **cache miss, block replacement**: wrong data is in cache at appropriate block, so discard it and fetch desired data from memory

- **Cache Access Procedure**: (1) Use Index bits to select cache block (2) If valid bit is 1, compare the tag bits of the address with the cache block tag bits (3) If they match, use the offset to read out the word/byte
Data valid, tag OK, so read offset return word d

Valid Tag 0x0-3 0x4-7 0x8-b 0xc-f

Index

0 1 2 3 4 5 6 7

0 0 0 0 0 0 0 0
1 0 1 a b c d
2 0 0 0 0 0 0 0
3 0 0 0 0 0 0 0
4 0 0 0 0 0 0 0
5 0 0 0 0 0 0 0
6 0 0 0 0 0 0 0
7 0 0 0 0 0 0 0

... ... ...

1022 0 0 0 0 0 0 0
1023 0 0 0 0 0 0 0

An Example Cache: DecStation 3100

- Commercial Workstation: ~1985
- MIPS R2000 Processor
- Separate instruction and data caches:
  - direct mapped
  - 64K Bytes (16K words) each
  - Block Size: 1 Word (Low Spatial Locality)

Solution:
Increase block size – 2nd example
DecStation 3100 Cache (Block size 1 word)

Address (showing bit positions)

If miss, cache controller stalls the processor, loads data from main memory

64KB Cache with 4-word (16-byte) blocks
**Fully Associative Cache**

- Fully associative cache allows any memory block to be stored in any cache block.
- Each cache block has a unique tag.

**Diagram:**
- Memory block addresses are shown.
- Tag and offset extraction is illustrated.

**Two-way Set Associative Cache**

- Two direct-mapped caches operate in parallel.
- Cache Index selects a "set" from the cache (set includes 2 blocks).
- The two tags in the set are compared in parallel.
- Data is selected based on the tag result.

**Diagram:**
- Functional blocks for cache data, cache index, and tag comparison are depicted.
4-way Set Associative Cache

- Allow block anywhere in a set
- Advantages:
  - Better hit rate
- Disadvantage:
  - More tag bits
  - More hardware
  - Higher access time

A Four-Way Set-Associative Cache, Block size = 4 bytes

Set Associative Cache - addressing

<table>
<thead>
<tr>
<th>TAG</th>
<th>INDEX/Set #</th>
<th>OFFSET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag to check if have correct block anywhere in set</td>
<td>Index to select a set in cache</td>
<td>Byte offset</td>
</tr>
</tbody>
</table>
Types of Cache Misses (for 3 organizations)

- **Compulsory (cold start):** location has never been accessed - first access to a block not in the cache
- **Capacity:** since the cache cannot contain all the blocks of a program, some blocks will be replaced and later retrieved
- **Conflict:** when too many blocks try to load into the same set, some blocks will be replaced and later retrieved

Cache Design Decisions

- For a given cache size
  - Block (Line) size
    - Number of Blocks (Lines)
  - How is the cache organized
  - Write policy
  - Replacement Strategy

- Increase cache size
  - More Blocks (Lines)
  - More lines == Higher hit rate
  - Slower Memory
  - As many as practical
**Block size**

- Miss rate goes down with block size (why?)
- Extreme Example: choose block size = cache size.
  - only one block in cache
- Temporal Locality says if an item is accessed, it is likely to be accessed again soon
  - But it is unlikely that it will be accessed again immediately!!!
  - The next access is likely to be a miss
    - Continually loading data into the cache but forced to discard them before they are used again
    - Worst nightmare of a cache designer: Ping Pong Effect

**Block Size and Miss Penalty**

- With increase in block size, the cost of a miss also increases
- Miss penalty: time to fetch the block from the next lower level of the hierarchy and load it into the cache
- With very large blocks, increase in miss penalty overwhelms decrease in miss rate
- Can minimize average access time if design memory system right
Miss Rate Versus Block Size

![Graph showing the relationship between miss rate and block size for different cache sizes.]

Block Size Tradeoff

Exploits Spatial Locality
Fewer blocks: compromises temporal locality

Increased Miss Penalty & Miss Rate

Average Access Time

Block Size
Writing to the Cache and Block Replacement

- Need to keep cache consistent with memory
  - Write to cache & memory simultaneously: "Write-through"
  - Or: Write to cache and mark as 'dirty'
    - Need to eventually copy back to memory: "Write-back"

- Need to make space in cache for a new entry

Which Line Should be 'Evicted' (Q3)

- Ideal?: Longest Time Till Next Access
- Least-recently used
  - Complicated
- Random selection
  - Simple
- Effect on hit rate is relatively small

Replacement Policy

- For direct-mapped cache - easy since only one block is replaced
- For fully-associative and set-associative cache - two strategies:
  - Random
  - Least-recently used (LRU)—replace the block that has not been accessed for a long time. (Principle of temporal locality)
Measuring Cache Performance

- CPU time = Execution cycles \times \text{clock cycle time} = 
  \text{Instruction\_Count} \times \text{CPI} \times \text{clock cycle}

- If cache miss: (Execution cycles + Memory stall cycles) \times \text{clock cycle time}

- Memory-stall cycles
  = Memory accesses \times \text{miss rate} \times \text{miss penalty}
  = \# \text{instructions} \times \text{misses/instruction} \times \text{miss penalty}

Example

**Question:** Cache miss penalty = 50 cycles and all instructions take 2.0 cycles without memory stalls. Assume cache miss rate of 2% and 1.33 (why?) memory references per instruction. What is the impact of cache?

**Answer:** CPU time = IC \times (\text{CPI} + \text{Memory stall cycles/instruction}) \times \text{cycle time} \tau

Performance including cache misses is

CPU time = IC \times (2.0 + (1.33 \times 0.02 \times 50)) \times \text{cycle time} = IC \times 3.33 \times \tau

For a perfect cache that never misses CPU time = IC \times 2.0 \times \tau

Hence, including the memory hierarchy stretches CPU time by 1.67

But, without memory hierarchy, the CPI would increase to 2.0 + 50 \times 1.33 or 68.5 – a factor of over 30 times longer
Summary: cache organizations

- **Direct-mapped**: a memory location maps onto exactly one cache entry
- **Fully-associative**: a memory location can be stored anywhere in cache
- **N-way set-associative**: each memory location can go into one of n sets

Block #12 placed in a cache with 8 block frames:

![Diagram of cache organizations](image_url)