Cache Organization

- Small amount of fast memory
- Sits between normal main memory and CPU
- May be located on CPU chip or module

Cache operation - overview

- CPU requests contents of memory location
- Check cache for this data
  - Cache includes tags to identify which block of main memory is in each cache slot (line)
  - If present, get from cache (fast)
  - Ideally most heavily used memory words stored in cache.
- If not present, read required block from main memory to cache
  - Locality principle: when a word is referenced, words around it brought into cache, too [a cache line (memory block) brought in]
- Then deliver from cache to CPU
- Timing
  - Hit ratio \( h \): fraction of memory word references satisfied by cache (also, miss ratio = 1 − \( h \))
  - Mean memory word access time: \( hc + (1 − h) m \) where \( c \) is cache access time, \( h \) is hit ratio, \( (1 − h) \) is percentage of references that have to go to memory, and \( m \) is memory access time

Cache Design

- Size
- Mapping Function
- Replacement Algorithm
- Write Policy
- Block Size
- Number of Caches
Cache Size Does Matter

• Cost
  – More cache is expensive

• Speed
  – More cache is faster (up to a point)
  – Checking cache for data takes time

Typical Cache Organization

Mapping Function Example

• Byte-addressable memory
• Cache of 64KBytes
• Cache line size of 4 bytes
  – i.e. cache is 16k ($2^{14}$) lines of 4 bytes
• 16MBytes main memory
• 24 bit address
  – ($2^{24}=16M$)

Direct Mapping

• Each block of main memory maps to only one cache line
  – i.e. if a block is in cache, it must be in one specific place

• Address is in two parts
• Least Significant $w$ bits identify unique word (addressable unit)
• Most Significant $s$ bits specify one memory block
• The most significant bits are split into a cache line field $r$ and a tag of $s-r$
### Direct Mapping Address Structure

<table>
<thead>
<tr>
<th>Tag</th>
<th>Line</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>14</td>
<td>2</td>
</tr>
</tbody>
</table>

- 24 bit address
- 2 bit word identifier (4 byte block)
- 22 bit block identifier
  - 8 bit tag (=22-14)
  - 14 bit line
- No two blocks in the same line have the same Tag field
- Check contents of cache by finding line and checking Tag
- e.g. (in hexadecimal)

#### Address
FFFFFC

#### Tag Line Word

<table>
<thead>
<tr>
<th>8</th>
<th>14</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1111 1111</td>
<td>11111111 1111100</td>
</tr>
</tbody>
</table>

---

### Direct Mapping: Cache Line “Table”

<table>
<thead>
<tr>
<th>Cache line</th>
<th>Main Memory blocks held</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0, m, 2m, 3m..2^s-m</td>
</tr>
<tr>
<td>1</td>
<td>1,m+1, 2m+1..2^s-m+1</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>m-1</td>
<td>m-1, 2m-1,3m-1..2^s-1</td>
</tr>
</tbody>
</table>

Our example:  

- \( m = 2^{14} \)
- \( s = 22 \)

---

### Direct Mapping Cache Organization
**Direct Mapping pros & cons**

- Simple
- Inexpensive
- Fixed location for given block
  - If a program accesses 2 blocks that map to the same line repeatedly, cache misses are very high

**Associative Mapping: Address Structure**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>22 bits</td>
<td>2 bits</td>
</tr>
</tbody>
</table>

- 22 bit tag stored with each 32 bit block of data
- Compare tag field with tag entry in cache to check for hit
- Least significant 2 bits of address identify which 8 bit byte is required from 32 bit data block
- e.g. (in hexadecimal)
  
  **Address**  
  `FFFFFC`

**Associative Mapping: Address Structure**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>22 bits</td>
<td>2 bits</td>
</tr>
</tbody>
</table>

- 22 bit tag stored with each 32 bit block of data
- Compare tag field with tag entry in cache to check for hit
- Least significant 2 bits of address identify which 8 bit byte is required from 32 bit data block
- e.g. (in hexadecimal)
  
  **Address**  
  `FFFFFC`  
  **Tag**  
  `3FFFFF`  
  **Word**  
  `0`

**Fully) Associative Mapping**

- A main memory block can load into any line of cache
- Memory address is interpreted as tag and word
- Tag uniquely identifies block of memory
- Every line's tag is examined for a match
- Cache searching gets expensive
Associative Cache Organization

- Cache is divided into a number of sets
- Each set contains a number of lines
- A given memory block maps to any line in a given set
  - e.g. Block B can be in any line of set i
- e.g. 2 lines per set
  - 2-way associative mapping
  - A given block can be in one of 2 lines in only one set

Set Associative Mapping Example

- 13-bit set number
- Block number in main memory is modulo $2^{13}$
- $0x000000$, $0x002000$, $0x004000$, $0x006000$, … map to same set

2-Way Set Associative Mapping Address Structure

- Use set field to determine cache set to look in
- Compare tag field to see if we have a hit
- e.g. (in hexadecimal)
  - Address
  - FFFFFC
  - 1111 1111 1111 1111 1111 1111 1110
2-Way Set Associative Mapping Address Structure

<table>
<thead>
<tr>
<th>Tag</th>
<th>Set</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 bit</td>
<td>13 bit</td>
<td>2 bit</td>
</tr>
</tbody>
</table>

- Use set field to determine cache set to look in
- Compare tag field to see if we have a hit
- E.g. (in hexadecimal)

<table>
<thead>
<tr>
<th>Address</th>
<th>Tag</th>
<th>Set</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFFFFC</td>
<td>1FF</td>
<td>1FF</td>
<td>0</td>
</tr>
</tbody>
</table>

K-Way Set Associative Cache Organization

Replacement Algorithms

- Direct Mapping
  - No choice
  - Each block only maps to one line
  - Replace that line
- Associative and Set-Associative
  - Hardware implemented algorithm (speed)
  - Least Recently Used (LRU)
  - E.g. in 2-way set associative
    - Which of the 2 blocks is LRU?
  - First In First Out (FIFO)
    - Replace block that has been in cache longest
  - Least Frequently Used
    - Replace block which has had fewest hits
  - Random

Write Policy Importance

- Must not overwrite a cache block without updating main memory
- Multiple CPUs may have individual caches
  - Cache coherency
- I/O may address main memory directly
Write through

- All writes go to main memory as well as cache
- Multiple CPUs can monitor main memory traffic to keep local (to CPU) cache up to date
- Lots of traffic
- Slows down writes

Write back

- Updates initially made in cache only
- Update bit for cache slot is set when update occurs
- If block is to be replaced, write to main memory only if update bit is set
- Other caches get out of sync
- I/O must access main memory through cache
- Note: ~15% of memory references are writes

Number of Caches

- **L1 cache**
  - On-chip cache
  - Unified vs split
    - Unified: One cache that holds both instructions and data
    - Split: Two separate caches, one for data, one for instructions allowing for parallel access
- **L2 cache**
  - External (to processor & main memory) cache
- **L3 cache**
  - On bus interconnect switch of multi-processor system
  - Other locations