Purpose of Cache

CPU

Bridge CPU-Memory speed gap

Memory access ≈ 30 to 100 cycles
Cache access ≈ 1 to 2 cycles
Block size = line size = 8 to 128 Bytes

Physical Address

Penalty?

Cache location

Must locate the data in the cache and get it in one cycle!
Use some variation of Content Addressable Memory (CAM)

Tag

Block Data

Copyright Koren UMass 2016
Fully Associative Cache

Place block anywhere in cache
Most flexible
Most expensive

Tag memory
Valid bits
Cache data memory

Main Memory

MM block 0
MM block 1
MM block 2
MM block 119
MM block 421

Cache block 255
Cache block 2
Cache block 0

Valid bits
13
3

Offset
Tag

Main memory address

Cache hit

Tag

Argument register

Match bit

Valid bit

Data

Cache block 0
Cache block 2
Cache block 255

Main memory address

Tag
Offset

Selector

To CPU

One cache line, 8 bytes

One cache line, 8 bytes

Fully Associative Cache Operation

TAG=119

ECE568/Koren Part.12.3

ECE568/Koren Part.12.4
Simplest Cache: Direct Mapped

Memory Address | Memory
--- | ---
0000 | 
0001 | 
0010 | 
0011 | 
0100 | 
0101 | 
0110 | 
0111 | 
1000 | 
1001 | 
1010 | 
1011 | 
1100 | 
1101 | 
1110 | 
1111 | 

Cache Index
- Location 0 in cache can be occupied by data from memory location 0, 4, 8, ...
- In general: any memory location whose 2 LSBs are 00
- Address<1:0> => cache index
- Which one should we place in cache?
- How can we tell which one is in cache?

Block size = Byte

Direct Mapped Cache

Input Address

Tag | Index
--- | ---

Tag Decoder

Data Decoder

Valid Bit

Comparator

MUX

Hit

Data

2^k

2t cache line size (bytes)
Direct-Mapped Cache (Block size = 8 Bytes)

<table>
<thead>
<tr>
<th>Main memory numbers</th>
<th>Main memory block numbers</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 256 512</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 257 513</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2 258 514</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>255 511 767</td>
<td>30</td>
<td>30</td>
</tr>
</tbody>
</table>

Cache address 8 3

Main memory address 5 8 3

Tag Index Offset

Direct-Mapped Cache Operation

Only 1 comparator

8-256 decoder

Index=0000 0010

Cache memory

Selector 6

Page 4
Compromise: n-way Set Associative Cache

- n entries for each Cache Index
  - A block can be placed in one out of n frames (n power of 2)
  - Direct mapping is 1-way set associative
  - Fully associative is j-way set associative (j is no. of block frames in cache)

Example: Two-way set associative cache

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
Disadvantage of Set Associative Cache

- N-way Set Associative Cache v. Direct Mapped Cache:
  - N comparators vs. 1
  - Extra MUX delay for the data
  - Data comes AFTER Hit/Miss

- In a direct mapped cache, Cache Block is available BEFORE Hit/Miss:
  - Possible to assume a hit and continue. Recover later if miss.

Q1: Where can a block be placed in the upper level?

- Block #12 placed in a cache with 8 block frames:
  - Fully associative, direct mapped, 2-way set associative
  - SetAssoc. Mapping = (Block_Number) Modulo (Number_of_Sets)
Q2: How is a block found if it is in the upper level?

- Tag for each block
- Increasing associativity shrinks index —— expands tag ——

<table>
<thead>
<tr>
<th>Block Address</th>
<th>Block Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>Index</td>
</tr>
</tbody>
</table>

Set No.

Q3: Which block should be replaced on a miss?

- Easy for Direct Mapped
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used) only approximated

<table>
<thead>
<tr>
<th>Assoc:</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>LRU</td>
<td>Ran</td>
<td>LRU</td>
</tr>
<tr>
<td>16 KB</td>
<td>5.2%</td>
<td>5.7%</td>
<td>4.7%</td>
</tr>
<tr>
<td>64 KB</td>
<td>1.9%</td>
<td>2.0%</td>
<td>1.5%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15%</td>
<td>1.17%</td>
<td>1.13%</td>
</tr>
</tbody>
</table>
Q4: What happens on a write? (Hit)

- Write in cache - should modify only part of block
- **Write through** - The information is written to both the block in the cache and to the block in the memory
- **Write back** - The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced

Pros and Cons of each

- WT: Can always discard cached data - up-to-date data is in memory, while WB: Can't discard it - may have to write it back
- WT: need only a valid bit, while WB: need valid and dirty bits
- WT: Read misses (in cache) do not result in writes (in memory); Memory (or other processors) always have latest data; Simpler management of cache
- WB: Much lower bandwidth, since data often overwritten multiple times; Better tolerance to long-latency memory

Write Buffer for Write Through

- **A Write Buffer is added between the Cache and Memory**
  - Processor: writes data into the cache and the write buffer and may continue w/o stalls
  - Memory controller: write contents of the buffer to memory
- **Write buffer is just a FIFO:**
  - Typical number of entries: 4
  - Works fine if: Store frequency \( < 1 / DRAM\text{\_write\_cycle} \)
What happens on write-miss

♦ **Write allocate**: allocate new cache line in cache
  - Usually means that you have to do a “read miss” to fill in rest of the cache-line!

♦ **Write non-allocate** (or “write-around”):
  - Simply send write data through to underlying memory/(level 2 cache) - don’t allocate new cache line

Three levels of Hierarchy

♦ **CPU** generates virtual rather than physical addresses
  - VA (disk) -> PA (memory) -> cache
  - VA to PA translation commonly done by TLB (*Translation Lookaside Buffer*)
  - Really just a cache on the page table mappings
  - TLB access time comparable to cache access time (much less than main memory access time)
Translation Look-Aside Buffers

Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped.

TLBs are usually small, typically not more than 128 - 256 entries even on high end machines. This permits fully associative lookup on these machines. Most mid-range machines use small n-way set associative organizations.

Translation with a TLB

TLB in "IBM 3033"
2-way set-assoc.
128 entries = 64 * 2
Page size = 4K
Memory size = 32M
= 2^25
Virtual Address and a Cache

It takes an extra TLB access to translate VA to PA making cache access expensive.

- **Solution 1:** Access cache with VA
  - **Synonym / alias problem:** two different VAs map to same PA
    => two different cache entries holding data for the same physical address
  - For update: must update all cache entries with same PA or memory becomes inconsistent
    determining this requires significant hardware
  - Two processes map the same VA to two different PAs with only one entry in cache - solution: flush cache upon context switch

Solution 2: Overlapped Cache & TLB Access

The page offset bits are available (will not change); Use high order bits of VA for TLB access while low order bits index the cache.

IF (V=1) AND (cache tag = PPN) then deliver data to CPU
**Drawback of Overlapped TLB Access**

- Small cache as it is limited by page size
- Increase page size, or
- Use n-way set associative cache
  
  \[ \text{Cache size} = n \times \text{Page size} \]

- Example: use 2-way set associative cache for an 8K byte total cache size

\[ \text{Cache size} = 2 \times 4K = 8K \text{ Bytes} \]

---

**Cache & TLB (modified) in IBM 3033**

- Page = 4KByte
- Cache (1 way)= 256 x 16 = 4KB
- 16-way set-associative
- Cache=64KByte
- Block=16 Bytes
- Memory=32MB = 2^25

---