Review: Direct-Mapped Cache

- Cache Location 0 can be occupied by data from:
  - Memory location 0, 4, 8, ...
  - 4 blocks => any memory location that is multiple of 4

Direct-Mapped Cache Terminology

- All fields are read as unsigned integers.
- **Index**: specifies the cache index (which "row" of the cache we should look in)
- **Offset**: once we've found correct block, specifies which byte within the block we want -- i.e., which "column"
- **Tag**: the remaining bits after offset and index are determined; these are used to distinguish between all the memory addresses that map to the same location

Direct-Mapped Cache Example (1/3)

- Suppose we have a 16KB of data in a direct-mapped cache with 4 word blocks
- Determine the size of the tag, index and offset fields if we're using a 32-bit architecture
- **Offset**
  - need to specify correct byte within a block
  - block contains 4 words
    - 16 bytes
    - 2^4 bytes
  - need **4 bits** to specify correct byte

Direct-Mapped Cache Example (2/3)

- **Index**: (~index into an “array of blocks”)  
  - need to specify correct row in cache
  - cache contains 16 KB = 2^14 bytes
  - block contains 2^4 bytes (4 words)
  - # blocks/cache
    - 2^14 bytes/block
    - 2^2 bytes/cache
    - 2^10 blocks/cache
  - need **10 bits** to specify this many rows
Direct-Mapped Cache Example (3/3)

• Tag: use remaining bits as tag
  • tag length = addr length - offset - index
    = 32 - 4 - 10 bits
    = 18 bits
  • so tag is leftmost 18 bits of memory address
• Why not full 32 bit address as tag?
  • All bytes within block need same address (4b)
  • Index must be same for every address within a block, so its redundant in tag check, thus can leave off to save memory (10 bits in this example)

Caching Terminology

• When we try to read memory, 3 things can happen:
  1. cache bit:
     cache block is valid and contains proper address, so read desired word
  2. cache miss:
     nothing in cache in appropriate block, so fetch from memory
  3. cache miss, block replacement:
     wrong data is in cache at appropriate block, so discard it and fetch desired data from memory (cache always copy)

Accessing data in a direct mapped cache

• 4 Addresses:
  • 0x00000014 , 0x0000001C , 0x00000034 , 0x00008014

Memory

Address (hex) Value of Word

00000000 00000014  d  
00000000 00000018  e  
00000000 0000001C  b  
00000000 00008014  f  
00000030 00000034  g  
00000030 00000038  o  
00000030 0000003C  r  
00000030 00008014  h  
00000810 00000038  t  
00000810 0000003C  u  
00000810 00008014  i  
00000810 00008014  j  
00000810 00008014  k  
00000810 00008014  l  
00000810 00008014  m  

Valid

Index Tag 0x0-3 0x4-7 0x8-b 0xc-f
0    1    2    3    4    5    6    7

...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...

...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...

...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...

...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...

...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...

...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...

...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...

...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...

...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...

...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...

...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...

...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...

...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...

...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...

...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...

...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...

...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...

16 KB Direct Mapped Cache, 16B blocks

• Valid bit: determines whether anything is stored in that row (when computer initially turned on, all entries invalid)

 area (cache size, B) = height (# of blocks) * width (size of one block, B/block)

height (# of blocks)

area (cache size, B)

width (size of one block, B/block)
1. Read 0x00000014

<table>
<thead>
<tr>
<th>Valid</th>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x0</td>
<td>0x0-3</td>
<td>0x0-3</td>
</tr>
</tbody>
</table>

So we read block 1 (0000000001)

<table>
<thead>
<tr>
<th>Valid</th>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x1</td>
<td>0x4-7</td>
<td>0x4-7</td>
</tr>
</tbody>
</table>

No valid data

<table>
<thead>
<tr>
<th>Valid</th>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x2</td>
<td>0x8-b</td>
<td>0x8-b</td>
</tr>
</tbody>
</table>

So load that data into cache, setting tag, valid

<table>
<thead>
<tr>
<th>Valid</th>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x3</td>
<td>0xc-f</td>
<td>0xc-f</td>
</tr>
</tbody>
</table>

Read from cache at offset, return word b

<table>
<thead>
<tr>
<th>Valid</th>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x4</td>
<td>0x0-3</td>
<td>0x0-3</td>
</tr>
</tbody>
</table>

2. Read 0x0000001C = 0...00 0..001 1100

<table>
<thead>
<tr>
<th>Valid</th>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x5</td>
<td>0x4-7</td>
<td>0x4-7</td>
</tr>
</tbody>
</table>

So load that data into cache, setting tag, valid

<table>
<thead>
<tr>
<th>Valid</th>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x6</td>
<td>0x8-b</td>
<td>0x8-b</td>
</tr>
</tbody>
</table>

Read from cache at offset, return word b

<table>
<thead>
<tr>
<th>Valid</th>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x7</td>
<td>0xc-f</td>
<td>0xc-f</td>
</tr>
</tbody>
</table>
### Index is Valid

- Tag field: 0x0-3, 0x4-7, 0x8-b, 0xc-f
- Index field: 0
- Offset: 1100

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a b c d</td>
<td></td>
</tr>
</tbody>
</table>

### Index valid, Tag Matches

- Tag field: 0x0-3, 0x4-7, 0x8-b, 0xc-f
- Index field: 0
- Offset: 1100

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a b c d</td>
<td></td>
</tr>
</tbody>
</table>

### Index Valid, Tag Matches, return d

- Tag field: 0x0-3, 0x4-7, 0x8-b, 0xc-f
- Index field: 0
- Offset: 1100

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a b c d</td>
<td></td>
</tr>
</tbody>
</table>

### 3. Read 0x00000034 = 0...00 0..011 0100

- Tag field: 0x0-3, 0x4-7, 0x8-b, 0xc-f
- Index field: 0
- Offset: 0100

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a b c d</td>
<td></td>
</tr>
</tbody>
</table>

### So read block 3

- Tag field: 0x0-3, 0x4-7, 0x8-b, 0xc-f
- Index field: 0
- Offset: 0100

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a b c d</td>
<td></td>
</tr>
</tbody>
</table>

### No valid data

- Tag field: 0x0-3, 0x4-7, 0x8-b, 0xc-f
- Index field: 0
- Offset: 0100

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a b c d</td>
<td></td>
</tr>
</tbody>
</table>
Load that cache block, return word f

4. Read 0x00008014 = 0…10 0..001 0100

Valid
Tag field
0x0-3 0x4-7 0x8-b 0xc-f
0 1 2 3
4 5 6 7
... ...
1022 1023

Index
0 1 2 3
4 5 6 7
... ...
1022 1023

So read Cache Block 1, Data is Valid

Cache Block 1 Tag does not match (0 != 2)

Valid
Tag field
0x0-3 0x4-7 0x8-b 0xc-f
0 1 2 3
4 5 6 7
... ...
1022 1023

Index
0 1 2 3
4 5 6 7
... ...
1022 1023

Miss, so replace block 1 with new data & tag

And return word j

Valid
Tag field
0x0-3 0x4-7 0x8-b 0xc-f
0 1 2 3
4 5 6 7
... ...
1022 1023

Index
0 1 2 3
4 5 6 7
... ...
1022 1023
Peer Instruction #1

- Chose from: Cache: Hit, Miss, Miss w. replace
  Values returned: a, b, c, d, e, ..., k, l

- Read address 0x00000030?
  000000000000000000 0000000011 0000

- Read address 0x0000001c?
  000000000000000000 0000000001 1100

Values returned: a ,b, c, d, e, ..., k, l

- Read address 0x00000000?
  000000000000000000 0000000000 0000

Pre-Exam Exercise #2

We are now going to stop for ~5 minutes. During this time, your goal is to (by yourself) come up with a potential exam exercise covering the topic of Digital Logic, State Machines, or CPU Design. Make it as much like a real exam question as possible.

After this five minutes, you will explain your question to a small group and work through how you would go about solving it. I’ll call on some random samples for the full class.

Big Endian vs. Little Endian

Big-endian and little-endian derive from Jonathan Swift’s Gulliver's Travels in which the Big Endians were a political faction that broke their eggs at the large end (”the primitive way”) and rebelled against the Lilliputian King who required his subjects (the Little Endians) to break their eggs at the small end.

- The order in which BYTES are stored in memory.
  - Bits always stored as usual. (E.g., 0x2C = 0b 1100 0010)

Consider the number 1025 as we normally write it:

BYTE3 BYTE2 BYTE1 BYTE0
00000000 00000000 00000010 00000001

Big Endian | Little Endian
---|---
0 | ADDRESS ADDRESS ADDRESS ADDRESS
1 | ADDRESS ADDRESS ADDRESS ADDRESS
2 | ADDRESS ADDRESS ADDRESS ADDRESS
3 | ADDRESS ADDRESS ADDRESS ADDRESS
4 | ADDRESS ADDRESS ADDRESS ADDRESS
5 | ADDRESS ADDRESS ADDRESS ADDRESS
6 | ADDRESS ADDRESS ADDRESS ADDRESS
7 | ADDRESS ADDRESS ADDRESS ADDRESS

Memorized this table yet?

Big endian and little-endian derive from Jonathan Swift's *Gulliver's Travels* in which the Big Endians were a political faction that broke their eggs at the large end ("the primitive way") and rebelled against the Lilliputian King who required his subjects (the Little Endians) to break their eggs at the small end.

- The order in which BYTES are stored in memory.
  - Bits always stored as usual. (E.g., 0x2C = 0b 1100 0010)

Consider the number 1025 as we normally write it:

<table>
<thead>
<tr>
<th>BYTE3</th>
<th>BYTE2</th>
<th>BYTE1</th>
<th>BYTE0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
</tbody>
</table>

Blah Blah Cache size 16KB Blah Blah

2^16 blocks Blah Blah how many bits?

- Answer! 2^Y means...

<table>
<thead>
<tr>
<th>X=0</th>
<th>X=1</th>
<th>X=2</th>
<th>X=3</th>
<th>X=4</th>
<th>X=5</th>
<th>X=6</th>
<th>X=7</th>
<th>X=8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y=0</td>
<td>Y=1</td>
<td>Y=1</td>
<td>Y=2</td>
<td>Y=3</td>
<td>Y=4</td>
<td>Y=6</td>
<td>Y=7</td>
<td>Y=8</td>
</tr>
</tbody>
</table>

0x00000030 a hit
Index = 3, Tag matches,
Offset = 0, value = e

0x0000001c a miss
Index = 1, Tag mismatch,
so replace from memory,
Offset = 0xc, value = d

Since reads, values must = memory values whether or not cached:

- 0x00000030 = e
- 0x0000001c = d

Answers

- 0x00000030 a hit
  - Memory Address Value of Word
    - 0x00000030 e
    - 0x0000001c d

- 0x0000001c a miss
  - Memory Address Value of Word
    - 0x00000030 e
    - 0x0000001c d

Pre-Exam Exercise #2

We are now going to stop for ~5 minutes. During this time, your goal is to (by yourself) come up with a potential exam exercise covering the topic of Digital Logic, State Machines, or CPU Design. Make it as much like a real exam question as possible.

After this five minutes, you will explain your question to a small group and work through how you would go about solving it. I’ll call on some random samples for the full class.

Memorized this table yet?

Big endian and little-endian derive from Jonathan Swift's *Gulliver's Travels* in which the Big Endians were a political faction that broke their eggs at the large end ("the primitive way") and rebelled against the Lilliputian King who required his subjects (the Little Endians) to break their eggs at the small end.

- The order in which BYTES are stored in memory.
  - Bits always stored as usual. (E.g., 0x2C = 0b 1100 0010)

Consider the number 1025 as we normally write it:

<table>
<thead>
<tr>
<th>BYTE3</th>
<th>BYTE2</th>
<th>BYTE1</th>
<th>BYTE0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
</tbody>
</table>

Big Endian | Little Endian
---|---
0 | ADDRESS ADDRESS ADDRESS ADDRESS
1 | ADDRESS ADDRESS ADDRESS ADDRESS
2 | ADDRESS ADDRESS ADDRESS ADDRESS
3 | ADDRESS ADDRESS ADDRESS ADDRESS
4 | ADDRESS ADDRESS ADDRESS ADDRESS
5 | ADDRESS ADDRESS ADDRESS ADDRESS
6 | ADDRESS ADDRESS ADDRESS ADDRESS
7 | ADDRESS ADDRESS ADDRESS ADDRESS

Memorized this table yet?

Blah Blah Cache size 16KB Blah Blah

2^16 blocks Blah Blah how many bits?

- Answer! 2^Y means...

<table>
<thead>
<tr>
<th>X=0</th>
<th>X=1</th>
<th>X=2</th>
<th>X=3</th>
<th>X=4</th>
<th>X=5</th>
<th>X=6</th>
<th>X=7</th>
<th>X=8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y=0</td>
<td>Y=1</td>
<td>Y=1</td>
<td>Y=2</td>
<td>Y=3</td>
<td>Y=4</td>
<td>Y=6</td>
<td>Y=7</td>
<td>Y=8</td>
</tr>
</tbody>
</table>

0x00000030 a hit
Index = 3, Tag matches,
Offset = 0, value = e

0x0000001c a miss
Index = 1, Tag mismatch,
so replace from memory,
Offset = 0xc, value = d

Since reads, values must = memory values whether or not cached:

- 0x00000030 = e
- 0x0000001c = d

Answers

- 0x00000030 a hit
  - Memory Address Value of Word
    - 0x00000030 e
    - 0x0000001c d

- 0x0000001c a miss
  - Memory Address Value of Word
    - 0x00000030 e
    - 0x0000001c d

Pre-Exam Exercise #2

We are now going to stop for ~5 minutes. During this time, your goal is to (by yourself) come up with a potential exam exercise covering the topic of Digital Logic, State Machines, or CPU Design. Make it as much like a real exam question as possible.

After this five minutes, you will explain your question to a small group and work through how you would go about solving it. I’ll call on some random samples for the full class.

Big Endian vs. Little Endian

Big-endian and little-endian derive from Jonathan Swift’s *Gulliver’s Travels* in which the Big Endians were a political faction that broke their eggs at the large end (“the primitive way”) and rebelled against the Lilliputian King who required his subjects (the Little Endians) to break their eggs at the small end.

- The order in which BYTES are stored in memory.
  - Bits always stored as usual. (E.g., 0x2C = 0b 1100 0010)

Consider the number 1025 as we normally write it:

<table>
<thead>
<tr>
<th>BYTE3</th>
<th>BYTE2</th>
<th>BYTE1</th>
<th>BYTE0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
</tbody>
</table>

Memorized this table yet?

Big endian and little-endian derive from Jonathan Swift's *Gulliver's Travels* in which the Big Endians were a political faction that broke their eggs at the large end ("the primitive way") and rebelled against the Lilliputian King who required his subjects (the Little Endians) to break their eggs at the small end.

- The order in which BYTES are stored in memory.
  - Bits always stored as usual. (E.g., 0x2C = 0b 1100 0010)

Consider the number 1025 as we normally write it:

<table>
<thead>
<tr>
<th>BYTE3</th>
<th>BYTE2</th>
<th>BYTE1</th>
<th>BYTE0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
</tbody>
</table>
How Much Information IS that?

- Print, film, magnetic, and optical storage media produced about 5 exabytes of new information in 2002. 92% of the new information stored on magnetic media, mostly in hard disks.
- Amt of new information stored on paper, film, magnetic, & optical media ~doubled in last 3 yrs
- Information flows through electronic channels -- telephone, radio, TV, and the Internet -- contained ~18 exabytes of new information in 2002, 3.5x more than is recorded in storage media. 98% of this total is the information sent & received in telephone calls - incl. voice & data on fixed lines & wireless.
- WWW ⇒ 170 Tb of information on its surface; in volume 17x the size of the Lib. of Congress print collections.
- Instant messaging ⇒ 5x10^9 msgs/day (750GB), 274 TB/yr.
- Email ⇒ ~400 PB of new information/year worldwide.

Block Size Tradeoff (1/3)

- Benefits of Larger Block Size
  - Spatial Locality: if we access a given word, we’re likely to access other nearby words soon
  - Very applicable with Stored-Program Concept: if we execute a given instruction, it’s likely that we’ll execute the next few as well
  - Works nicely in sequential array accesses too

Block Size Tradeoff (2/3)

- Drawbacks of Larger Block Size
  - Larger block size means larger miss penalty
  - on a miss, takes longer time to load a new block from next level
  - If block size is too big relative to cache size, then there are too few blocks
  - Result: miss rate goes up
  - In general, minimize Average Memory Access Time (AMAT)
    - Hit Time + Miss Penalty × Miss Rate

Block Size Tradeoff (3/3)

- Hit Time = time to find and retrieve data from current level cache
- Miss Penalty = average time to retrieve data on a current level miss (includes the possibility of misses on successive levels of memory hierarchy)
- Hit Rate = % of requests that are found in current level cache
- Miss Rate = 1 - Hit Rate

Extreme Example: One Big Block

- Cache Size = 4 bytes, Block Size = 4 bytes
  - Only ONE entry in the cache!
  - If item accessed, likely accessed again soon
  - But unlikely will be accessed again immediately!
  - The next access will likely to be a miss again
  - Continually loading data into the cache but discard data (force out) before use it again
  - Nightmare for cache designer: Ping Pong Effect

Block Size Tradeoff Conclusions

- Miss Penalty Exploits Spatial Locality
  - Fewer blocks: compromises temporal locality
  - Increased Miss Penalty & Miss Rate
Peer Instructions

1. All caches take advantage of spatial locality.
2. All caches take advantage of temporal locality.
3. On a read, the return value will depend on what is in the cache.

Peer Instruction Answer

1. All caches take advantage of spatial locality.
2. All caches take advantage of temporal locality.
3. On a read, the return value will depend on what is in the cache.

TRUE

FALSE

1. Block size = 1, no spatial!
2. That’s the idea of caches; We’ll need it again soon.
3. It better not! If it’s there, use it. Or, get from mem.

And in Conclusion...

- Mechanism for transparent movement of data among levels of a storage hierarchy
- set of address/value bindings
- address ⇒ index to set of candidates
- compare desired address with tag
- service hit or miss
  - load new block and binding on miss

Valid

Valid

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

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>