The Hardware/Software Interface
CSE351 Spring 2015

Lecture 17

Instructor:
Katelin Bailey

Teaching Assistants:
Roadmap

C:

car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);

Java:

Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();

Assembly language:

get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret

Machine code:

0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111

OS:

Windows 8

Mac

Computer system:

Intel Core i5

Memory, data, & addressing
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
Let’s set up a running example
This is memory:

<table>
<thead>
<tr>
<th>Binary</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>-3</td>
</tr>
<tr>
<td>00001</td>
<td>7</td>
</tr>
<tr>
<td>00010</td>
<td>12</td>
</tr>
<tr>
<td>00011</td>
<td>??</td>
</tr>
<tr>
<td>00100</td>
<td>??</td>
</tr>
<tr>
<td>00101</td>
<td>??</td>
</tr>
<tr>
<td>00110</td>
<td>??</td>
</tr>
<tr>
<td>00111</td>
<td>??</td>
</tr>
<tr>
<td>01000</td>
<td>??</td>
</tr>
<tr>
<td>01001</td>
<td>??</td>
</tr>
<tr>
<td>01010</td>
<td>-8</td>
</tr>
<tr>
<td>01011</td>
<td>??</td>
</tr>
<tr>
<td>01100</td>
<td>??</td>
</tr>
<tr>
<td>01101</td>
<td>??</td>
</tr>
<tr>
<td>01110</td>
<td>??</td>
</tr>
<tr>
<td>01111</td>
<td>??</td>
</tr>
<tr>
<td>10000</td>
<td>??</td>
</tr>
<tr>
<td>10001</td>
<td>??</td>
</tr>
<tr>
<td>10010</td>
<td>??</td>
</tr>
<tr>
<td>10011</td>
<td>??</td>
</tr>
<tr>
<td>10100</td>
<td>??</td>
</tr>
<tr>
<td>10101</td>
<td>??</td>
</tr>
<tr>
<td>10110</td>
<td>??</td>
</tr>
<tr>
<td>10111</td>
<td>??</td>
</tr>
<tr>
<td>11000</td>
<td>??</td>
</tr>
<tr>
<td>11001</td>
<td>??</td>
</tr>
<tr>
<td>11010</td>
<td>??</td>
</tr>
<tr>
<td>11011</td>
<td>??</td>
</tr>
<tr>
<td>11100</td>
<td>??</td>
</tr>
<tr>
<td>11101</td>
<td>??</td>
</tr>
<tr>
<td>11110</td>
<td>??</td>
</tr>
<tr>
<td>11111</td>
<td>??</td>
</tr>
</tbody>
</table>
We make these accesses:

- Read 0x00000
- Read 0x01010
- Read 0x00001
- Read 0x00010
We make these accesses:

- Read 0x00000
- Read 0x01010
- Read 0x00001
- Read 0x00010

```
<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>??</td>
</tr>
</tbody>
</table>
```

- Let’s assume a block size of one byte
  - If our cache is only one item (each of block size 1) we never really get much use out of it
  - We always have to toss something out

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>-3</td>
</tr>
<tr>
<td>00001</td>
<td>7</td>
</tr>
<tr>
<td>00010</td>
<td>12</td>
</tr>
<tr>
<td>00011</td>
<td>??</td>
</tr>
<tr>
<td>00100</td>
<td>??</td>
</tr>
<tr>
<td>00101</td>
<td>??</td>
</tr>
<tr>
<td>00110</td>
<td>??</td>
</tr>
<tr>
<td>00111</td>
<td>??</td>
</tr>
<tr>
<td>01000</td>
<td>??</td>
</tr>
<tr>
<td>01001</td>
<td>??</td>
</tr>
<tr>
<td>01010</td>
<td>-8</td>
</tr>
<tr>
<td>01011</td>
<td>??</td>
</tr>
<tr>
<td>01100</td>
<td>??</td>
</tr>
<tr>
<td>01101</td>
<td>??</td>
</tr>
<tr>
<td>01110</td>
<td>??</td>
</tr>
<tr>
<td>01111</td>
<td>??</td>
</tr>
<tr>
<td>10000</td>
<td>??</td>
</tr>
<tr>
<td>10001</td>
<td>??</td>
</tr>
<tr>
<td>10010</td>
<td>??</td>
</tr>
<tr>
<td>10011</td>
<td>??</td>
</tr>
<tr>
<td>10100</td>
<td>??</td>
</tr>
<tr>
<td>10101</td>
<td>??</td>
</tr>
<tr>
<td>10110</td>
<td>??</td>
</tr>
<tr>
<td>10111</td>
<td>??</td>
</tr>
<tr>
<td>11000</td>
<td>??</td>
</tr>
<tr>
<td>11001</td>
<td>??</td>
</tr>
<tr>
<td>11010</td>
<td>??</td>
</tr>
<tr>
<td>11011</td>
<td>??</td>
</tr>
<tr>
<td>11100</td>
<td>??</td>
</tr>
<tr>
<td>11101</td>
<td>??</td>
</tr>
<tr>
<td>11110</td>
<td>??</td>
</tr>
<tr>
<td>11111</td>
<td>??</td>
</tr>
</tbody>
</table>
Basic Direct-Mapped Cache

- Read 0x00000
- Read 0x01010
- Read 0x00001
- Read 0x00010

- Sticking with a one-byte block size
- With our cache able to store two things at once...
  - address % size (e.g. 01010 % 2) is the index
Basic Direct-Mapped Cache

- Read 0x00000 - miss
- Read 0x01010
- Read 0x00001
- Read 0x00010

- Sticking with a one-byte block size
- With our cache able to store two things at once...
  - address % size (e.g. 01010 % 2) is the index
Basic Direct-Mapped Cache

- Read 0x00000 - miss
- Read 0x01010 - evict
- Read 0x00001
- Read 0x00010

- Sticking with a one-byte block size
- With our cache able to store two things at once...
  - address % size (e.g. 01010 % 2) is the index
Basic Direct-Mapped Cache

- Read 0x00000 - miss
- Read 0x01010 - evict
- Read 0x00001 - miss
- Read 0x00010

- Sticking with a one-byte block size
- With our cache able to store two things at once…
  - address % size (e.g. 01010 % 2) is the index
Basic Direct-Mapped Cache

- Read 0x00000 - miss
- Read 0x01010 - evict
- Read 0x00001 - miss
- Read 0x00010 - evict

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>

- Sticking with a one-byte block size
- With our cache able to store two things at once...
  - address % size (e.g. 01010 % 2) is the index
Basic Direct-Mapped Cache

- Read 0x00000 - miss
- Read 0x00010 - evict
- Read 0x00001 - miss
- Read 0x00010

- Sticking with a one-byte block size
- With our cache able to store two things at once…
  - address % size (e.g. 01010 % 2) is the index
- But wait! how did we know that this wasn’t the right value already in the cache?
Basic Direct-Mapped Cache

- Read 0x00000 - miss
- Read 0x01010 - evict
- Read 0x00001 - miss
- Read 0x00010

- Sticking with a one-byte block size
- With our cache able to store two things at once...
  - address % size (e.g. 01010 % 2) is the index
- But wait! how did we know that this wasn’t the right value already in the cache?
  - We actually store the rest of the address to check against when we’re reading
What about larger block sizes?

- Read 0x00000
- Read 0x01010
- Read 0x00001
- Read 0x00010

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>XX</td>
<td>??</td>
</tr>
<tr>
<td>1</td>
<td>XX</td>
<td>??</td>
</tr>
</tbody>
</table>

With two-byte block size

- address % block size is the block offset

With our cache able to store two things at once...

- address % block size % # cache lines is the index

For 01011

- Block offset is 1
- Cache index is also 1
- Tag is the leftover bits: 010
What about larger block sizes?

- Read 0x00000 - miss
- Read 0x01010
- Read 0x00001
- Read 0x00010

With two-byte block size
  - address % block size is the block offset

With our cache able to store two things at once…
  - address % block size % # cache lines is the index

For 01011
  - Block offset is 1
  - Cache index is also 1
  - Tag is the leftover bits: 010
What about larger block sizes?

- Read 0x00000 - miss
- Read 0x01010 - miss
- Read 0x00001
- Read 0x00010

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>010</td>
<td>??</td>
</tr>
</tbody>
</table>

- With two-byte block size
  - address % block size is the block offset
- With our cache able to store two things at once...
  - address % block size % # cache lines is the index
- For 01011
  - Block offset is 1
  - Cache index is also 1
  - Tag is the leftover bits: 010

Index | Value
---|---
00000 | -3
00001 | 7
00010 | 12
00011 | ??
00100 | ??
00101 | ??
00110 | ??
00111 | ??
01000 | ??
01001 | ??
01010 | -8
01011 | ??
01100 | ??
01101 | ??
01110 | ??
01111 | ??
10000 | ??
10001 | ??
10010 | ??
10011 | ??
10100 | ??
10101 | ??
10110 | ??
10111 | ??
11000 | ??
11001 | ??
11010 | ??
11011 | ??
11100 | ??
11101 | ??
11110 | ??
11111 | ??
What about larger block sizes?

- Read 0x00000 - miss
- Read 0x01010 - miss
- Read 0x00001 - hit!
- Read 0x00010

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>010</td>
<td>??</td>
</tr>
</tbody>
</table>

With two-byte block size
- address % block size is the block offset

With our cache able to store two things at once...
- address % block size % # cache lines is the index

For 01011
- Block offset is 1
- Cache index is also 1
- Tag is the leftover bits: 010
**What about larger block sizes?**

- **Read 0x00000** - miss
- **Read 0x01010** - miss
- **Read 0x00001** - hit!
- **Read 0x00010** - evict

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>000</td>
<td>??</td>
</tr>
</tbody>
</table>

- **With two-byte block size**
  - address \% block size is the block offset
- **With our cache able to store two things at once...**
  - address \% block size \% \# cache lines is the index
- **For 01011**
  - Block offset is 1
  - Cache index is also 1
  - Tag is the leftover bits: 010

---

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>-3</td>
</tr>
<tr>
<td>00001</td>
<td>7</td>
</tr>
<tr>
<td>00010</td>
<td>12</td>
</tr>
<tr>
<td>00011</td>
<td>??</td>
</tr>
<tr>
<td>00100</td>
<td>??</td>
</tr>
<tr>
<td>00101</td>
<td>??</td>
</tr>
<tr>
<td>00110</td>
<td>??</td>
</tr>
<tr>
<td>00111</td>
<td>??</td>
</tr>
<tr>
<td>01000</td>
<td>??</td>
</tr>
<tr>
<td>01001</td>
<td>??</td>
</tr>
<tr>
<td>01010</td>
<td>-8</td>
</tr>
<tr>
<td>01011</td>
<td>??</td>
</tr>
<tr>
<td>01100</td>
<td>??</td>
</tr>
<tr>
<td>01101</td>
<td>??</td>
</tr>
<tr>
<td>01110</td>
<td>??</td>
</tr>
<tr>
<td>01111</td>
<td>??</td>
</tr>
<tr>
<td>10000</td>
<td>??</td>
</tr>
<tr>
<td>10001</td>
<td>??</td>
</tr>
<tr>
<td>10010</td>
<td>??</td>
</tr>
<tr>
<td>10011</td>
<td>??</td>
</tr>
<tr>
<td>10100</td>
<td>??</td>
</tr>
<tr>
<td>10101</td>
<td>??</td>
</tr>
<tr>
<td>10110</td>
<td>??</td>
</tr>
<tr>
<td>10111</td>
<td>??</td>
</tr>
<tr>
<td>11000</td>
<td>??</td>
</tr>
<tr>
<td>11001</td>
<td>??</td>
</tr>
<tr>
<td>11010</td>
<td>??</td>
</tr>
<tr>
<td>11011</td>
<td>??</td>
</tr>
<tr>
<td>11100</td>
<td>??</td>
</tr>
<tr>
<td>11101</td>
<td>??</td>
</tr>
<tr>
<td>11110</td>
<td>??</td>
</tr>
<tr>
<td>11111</td>
<td>??</td>
</tr>
</tbody>
</table>
What about multiple blocks?

- Read 0x00000 - miss
- Read 0x01010 - miss
- Read 0x00001 - hit!
- Read 0x00010 - evict

Things are still getting evicted in our cache though...
- What if we could store an extra tag and value per set?
What about multiple blocks?

- Read 0x00000
- Read 0x01010
- Read 0x00001
- Read 0x00010

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag1</th>
<th>Value1</th>
<th>Tag2</th>
<th>Value2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>XX</td>
<td>??</td>
<td>XX</td>
<td>??</td>
</tr>
<tr>
<td>1</td>
<td>XX</td>
<td>??</td>
<td>XX</td>
<td>??</td>
</tr>
</tbody>
</table>

- **Same address split up:**
- **For 01011**
  - Block offset is 1
  - Cache index is also 1
  - Tag is the leftover bits: 010
- **We just have to check all the slots in the row**
What about multiple blocks?

- **Read 0x00000** - miss
- **Read 0x01010**
- **Read 0x00001**
- **Read 0x00010**

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag 1</th>
<th>Value 1</th>
<th>Tag2</th>
<th>Value2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>7</td>
<td>XX</td>
<td>??</td>
</tr>
<tr>
<td>1</td>
<td>XX</td>
<td>??</td>
<td>XX</td>
<td>??</td>
</tr>
</tbody>
</table>

- **Same address split up:**
- **For 01011**
  - **Block offset** is 1
  - **Cache index** is also 1
  - **Tag** is the leftover bits: 010
- **We just have to check all the slots in the row**
What about multiple blocks?

- Read 0x00000 - miss
- Read 0x01010 - miss
- Read 0x00001
- Read 0x00010

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag1</th>
<th>Value1</th>
<th>Tag2</th>
<th>Value2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>7</td>
<td>XX</td>
<td>??</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>010</td>
<td>??</td>
<td>XX</td>
<td>??</td>
</tr>
</tbody>
</table>

- Same address split up:
- For 01011
  - Block offset is 1
  - Cache index is also 1
  - Tag is the leftover bits: 010
- We just have to check all the slots in the row
What about multiple blocks?

- Read 0x00000 - miss
- Read 0x01010 - miss
- Read 0x00001 - hit!
- Read 0x00010

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Value</th>
<th>Tag2</th>
<th>Value2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>7</td>
<td>-3</td>
<td>??</td>
</tr>
<tr>
<td>1</td>
<td>010</td>
<td>??</td>
<td>-8</td>
<td>??</td>
</tr>
</tbody>
</table>

- Same address split up:
- For 01011
  - Block offset is 1
  - Cache index is also 1
  - Tag is the leftover bits: 010
- We just have to check all the slots in the row
What about multiple blocks?

• Read 0x00000 - miss
• Read 0x01010 - miss
• Read 0x00001 - hit!
• Read 0x00010 - miss (but no evict!)

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag1</th>
<th>Value1</th>
<th>Tag2</th>
<th>Value2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>7</td>
<td>XX</td>
<td>??</td>
</tr>
<tr>
<td>1</td>
<td>010</td>
<td>??</td>
<td>-8</td>
<td>000</td>
</tr>
</tbody>
</table>

• Same address split up:
• For 01011
  • Block offset is 1
  • Cache index is also 1
  • Tag is the leftover bits: 010
• We just have to check all the slots in the row
What about multiple blocks?

- That is an associative cache

- This example is
  - Has two **sets** (rows)
  - a **2-way** cache (has room for two blocks per set)
    - an n-way cache has n blocks per set
  - Has a **block size** of two bytes

<table>
<thead>
<tr>
<th></th>
<th>000</th>
<th>010</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>010</td>
</tr>
<tr>
<td>1</td>
<td>000</td>
<td>??</td>
</tr>
</tbody>
</table>
Associativity

- For a cache with room for $n$ blocks:
  - You can have $n$ sets (each one block): **direct mapped cache**
  - You can have 1 set (each with 8 blocks): **fully associative**, or 8 way cache
  - You can have something in between

1-way
8 sets, 1 block each

2-way
4 sets, 2 blocks each

4-way
2 sets, 4 blocks each

8-way
1 set, 8 blocks

direct mapped

fully associative
Example placement in set-associative caches

- Where would data from address 0x1833 be placed?
  - Block size is 16 bytes.
- 0x1833 in binary is 00...0110000 011 0011.
Block replacement

- Replace something, of course, but what?
  - Caches typically use something close to least recently used (LRU)
  - (hardware usually implements “not most recently used”)
General Cache Organization (S, E, B)

- **E** = $2^e$ lines per set (we say “E-way”)
- **S** = $2^s$ sets
- **B** = $2^b$ bytes of data per cache line (the data block)

**Cache size:**
$$S \times E \times B \text{ data bytes}$$
Cache Read

- $E = 2^e$ lines per set
- $S = 2^s$ sets
- Address of byte in memory:
  - $t$ bits
  - $s$ bits
  - $b$ bits

- Locate set
- Check if any line in set has matching tag
- Yes + line valid: hit
- Locate data starting at offset

$B = 2^b$ bytes of data per cache line (the data block)
Example: Direct-Mapped Cache (E = 1)

Direct-mapped: One line per set
Assume: cache block size 8 bytes

S = 2^s sets

Address of int:

```
t bits 0...01 100
```

find set
Example: Direct-Mapped Cache (E = 1)

Direct-mapped: One line per set
Assume: cache block size 8 bytes

Address of int:

valid? + match?: yes = hit

block offset
Example: Direct-Mapped Cache (E = 1)

Direct-mapped: One line per set
Assume: cache block size 8 bytes

No match: old line is evicted and replaced
E-way Set-Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

Address of short int:

```
s t bits 0...01 100
```

```
find set
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```

```
0 1 2 3 4 5 6 7
```
E-way Set-Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

Address of short int:

compare both

valid? + match: yes = hit

block offset
E-way Set-Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

Valid? + match: yes = hit

No match:
- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), …
Types of Cache Misses

- **Cold (compulsory) miss**
  - Occurs on first access to a block

- **Conflict miss**
  - Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot
    - e.g., referencing blocks 0, 8, 0, 8, ... would miss every time
  - **direct-mapped cache**s have more conflict misses than **n-way set-associative** (where n is a power of 2 and n > 1)

- **Capacity miss**
  - Occurs when the set of active cache blocks (the working set) is larger than the cache (just won’t fit)
What about writes?

- Multiple copies of data exist:
  - L1, L2, possibly L3, main memory

What’s the problem with that?
What about writes?

- Multiple copies of data exist:
  - L1, L2, possibly L3, main memory

- What to do on a write-hit?
  - Write-through: write immediately to memory, all caches in between.
  - Write-back: defer write to memory until line is evicted (replaced)
    - Need a dirty bit to indicate if line is different from memory or not

- What to do on a write-miss?
  - Write-allocate: load into cache, update line in cache.
    - Good if more writes or reads to the location follow
  - No-write-allocate: just write immediately to memory.

- Typical caches:
  - Write-back + Write-allocate, usually
  - Write-through + No-write-allocate, occasionally
Write-back, write-allocate example

```
mov 0xFACE, T
```

**Cache**

```
U                        0xBEEF 0
```

dirty bit

**Memory**

```
T                      0xCAFE
U                      0xBEEF
```
Write-back, write-allocate example

mov 0xFACE, T  mov 0xFEED, T  mov U, %rax

Cache

<table>
<thead>
<tr>
<th>T</th>
<th>0xFEED</th>
</tr>
</thead>
</table>

Memory

<table>
<thead>
<tr>
<th>T</th>
<th>0xCAFE</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>0xBEEF</td>
</tr>
</tbody>
</table>
Write-back, write-allocate example

mov 0xFACE, T  mov 0xFEED, T  mov U, %rax

Cache

| U | 0xBEEF | 0 |

Memory

| T | 0xFEED |
| U | 0xBEEF |

dirty bit
Where else is caching used?
Software Caches are More Flexible

• **Examples**
  • File system buffer caches, web browser caches, etc.

• **Some design differences**
  • Almost always fully-associative
    • so, no placement restrictions
    • index structures like hash tables are common (for placement)
  • Often use complex replacement policies
    • misses are very expensive when disk or network involved
    • worth thousands of cycles to avoid them
  • Not necessarily constrained to single “block” transfers
    • may fetch or write-back in larger units, opportunistically
Optimizations for the Memory Hierarchy

• Write code that has locality!
  • Spatial: access data contiguously
  • Temporal: make sure access to the same data is not too far apart in time

• How can you achieve locality?
  • Proper choice of algorithm
  • Loop transformations
Cache-Friendly Code

- Programmer can optimize for cache performance
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
    - Blocking is a general technique
- All systems favor “cache-friendly code”
  - Getting absolute optimum performance is very platform specific
    - Cache sizes, line sizes, associativities, etc.
  - Can get most of the advantage with generic code
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality)
    - Focus on inner loop code
Intel Core i7 Cache Hierarchy

Processor package

Core 0

Regs

L1 d-cache

L1 i-cache

L2 unified cache

L3 unified cache (shared by all cores)

Core 3

Regs

L1 d-cache

L1 i-cache

L2 unified cache

L2 unified cache

L3 unified cache (shared by all cores)

L1 i-cache and d-cache: 32 KB, 8-way, Access: 4 cycles

L2 unified cache: 256 KB, 8-way, Access: 11 cycles

L3 unified cache: 8 MB, 16-way, Access: 30-40 cycles

Block size: 64 bytes for all caches.