Caches & Memory

Hakim Weatherspoon
CS 3410
Computer Science
Cornell University

[Weatherspoon, Bala, Bracy, McKee, and Sirer]
Programs 101

Load/Store Architectures:
• Read data from memory (put in registers)
• Manipulate it
• Store it back to memory

C Code

```c
int main (int argc, char* argv[ ]) {
    int i;
    int m = n;
    int sum = 0;
    for (i = 1; i <= m; i++) {
        sum += i;
    }
    printf (“...”, n, sum);
}
```

RISC-V Assembly

```assembly
main:   addi    sp,sp,-48
sw      x1,44(sp)
sw      fp,40(sp)
move    fp,sp
sw      x10,-36(fp)
sw      x11,-40(fp)
la      x15,n
lw      x15,0(x15)
sw      x15,-28(fp)
sw      x0,-24(fp)
li      x15,1
sw      x15,-20(fp)
L2:     lw      x14,-20(fp)
lw      x15,-28(fp)
blt     x15,x14,L3
...    
```

Instructions that read from or write to memory…
Load/Store Architectures:
- Read data from memory (put in registers)
- Manipulate it
- Store it back to memory

```c
int main (int argc, char* argv[ ]) {
    int i;
    int m = n;
    int sum = 0;
    for (i = 1; i <= m; i++) {
        sum += i;
    }
    printf ("...", n, sum);
}
```

**Instructions that read from or write to memory...**
1 Cycle Per Stage: the Biggest Lie (So Far)

Code Stored in Memory
(also, data and stack)

Instruction Fetch

Instruction Decode

Instruction Execution

Instruction Write-Back
What's the problem?

CPU

Main Memory
+ big
– slow
– far away
The Need for Speed

CPU Pipeline
The Need for Speed

CPU Pipeline

Instruction speeds:
- **add, sub, shift**: 1 cycle
- **mult**: 3 cycles
- **load/store**: 100 cycles
  - off-chip 50(-70)ns
  - 2(-3) GHz processor → 0.5 ns clock
The Need for Speed

CPU Pipeline
What’s the solution?

Caches!

Level 1
Insn $ $

Level 1
Data $

Level 2 $ $

Intel Pentium 3, 1999
Aside

• Go back to 04-state and 05-memory and look at how registers, SRAM and DRAM are built.
What’s the solution?

Caches!

What lucky data gets to go here?

Intel Pentium 3, 1999
Locality Locality Locality

If you ask for something, you’re likely to ask for:

• the same thing again soon
  → Temporal Locality

• something near that thing, soon
  → Spatial Locality

```cpp
int total = 0;
for (int i = 0; i < n; i++)
    total += a[i];
return total;
```
Clicker Questions

This highlights the temporal and spatial locality of data.

Q1: Which line of code exhibits good temporal locality?

```
1  total = 0;
2  for (i = 0; i < n; i++) {
3      n--;
4      total += a[i];
5  return total;
```
Clicker Questions

This highlights the *temporal* and *spatial* locality of data.

Q1: Which line of code exhibits good *temporal* locality?

```java
1  total = 0;
2  for (i = 0; i < n; i++) {
3    n--;  
4  total += a[i];
5  return total;
```

A) 1
B) 2
C) 3
D) 4
E) 5
Clicker Questions

This highlights the temporal and spatial locality of data.

Q1: Which line of code exhibits good temporal locality?

```
1  total = 0;
2  for (i = 0; i < n; i++) {
3      n--;
4  total += a[i];
5  return total;
```

A) 1  
B) 2  
C) 3  
D) 4  
E) 5

Q2: Which line of code exhibits good spatial locality with the line after it?

A) 1  
B) 2  
C) 3  
D) 4  
E) 5
Clicker Questions

This highlights the **temporal** and **spatial** locality of **data**.

Q1: Which line of code exhibits good **temporal** locality?

A) 1  
B) 2  
C) 3  
D) 4  
E) 5

Q2: Which line of code exhibits good **spatial** locality with the line after it?

```c
1 total = 0;
2 for (i = 0; i < n; i++) {  
3 n--;  
4 total += a[i];  
5 return total;
```
Your life is full of Locality

Last Called
Speed Dial
Favorites
Contacts
Google/Facebook/email
Your life is full of Locality
The Memory Hierarchy

- **Registers**: 1 cycle, 128 bytes
- **L1 Caches**: 4 cycles, 64 KB
- **L2 Cache**: 12 cycles, 256 KB
- **L3 Cache**: 36 cycles, 2-20 MB
- **Main Memory**: 50-70 ns, 512 MB – 4 GB
- **Disk**: 5-20 ms, 16GB – 4 TB

Small, Fast → Big, Slow

Intel Haswell Processor, 2013
Some Terminology

Cache hit

- data is in the Cache
- \( t_{hit} \): time it takes to access the cache
- Hit rate (%hit): \( \# \text{ cache hits} / \# \text{ cache accesses} \)

Cache miss

- data is not in the Cache
- \( t_{miss} \): time it takes to get the data from below the $\$
- Miss rate (%miss): \( \# \text{ cache misses} / \# \text{ cache accesses} \)

Cacheline or cacheblock or simply line or block

- Minimum unit of info that is present/or not in the cache
The Memory Hierarchy

- ** Registers**: 1 cycle, 128 bytes
- **L1 Caches**: 4 cycles, 64 KB
- **L2 Cache**: 12 cycles, 256 KB
- **L3 Cache**: 36 cycles, 2-20 MB
- **Main Memory**: 50-70 ns, 512 MB – 4 GB
- **Disk**: 5-20 ms, 16 GB – 4 TB

Average access time:

\[
t_{\text{avg}} = t_{\text{hit}} + \%_{\text{miss}} \times t_{\text{miss}}
\]

\[
= 4 + 5\% \times 100
\]

\[
= 9 \text{ cycles}
\]
Single Core Memory Hierarchy
Multi-Core Memory Hierarchy

ON CHIP

Processor

Regs

I$  D$

L2

Processor

Regs

I$  D$

L2

Processor

Regs

I$  D$

L2

Processor

Regs

I$  D$

L2

Main Memory

L3

Disk

Disk
### Memory Hierarchy by the Numbers

**CPU clock rates ~0.33ns – 2ns (3GHz-500MHz)**

<table>
<thead>
<tr>
<th>Memory technology</th>
<th>Transistor count*</th>
<th>Access time</th>
<th>Access time in cycles</th>
<th>$ per GIB in 2012</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM (on chip)</td>
<td>6-8 transistors</td>
<td>0.5-2.5 ns</td>
<td>1-3 cycles</td>
<td>$4k</td>
<td>256 KB</td>
</tr>
<tr>
<td>SRAM (off chip)</td>
<td></td>
<td>1.5-30 ns</td>
<td>5-15 cycles</td>
<td>$4k</td>
<td>32 MB</td>
</tr>
<tr>
<td>DRAM</td>
<td>1 transistor</td>
<td>50-70 ns</td>
<td>150-200 cycles</td>
<td>$10-$20</td>
<td>8 GB</td>
</tr>
<tr>
<td>(needs refresh)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSD (Flash)</td>
<td></td>
<td>5k-50k ns</td>
<td>Tens of thousands</td>
<td>$0.75-$1</td>
<td>512 GB</td>
</tr>
<tr>
<td>Disk</td>
<td></td>
<td>5M-20M ns</td>
<td>Millions</td>
<td>$0.05-$0.1</td>
<td>4 TB</td>
</tr>
</tbody>
</table>

*Registers, D-Flip Flops: 10-100’s of registers*
Basic Cache Design

Direct Mapped Caches
16 Byte Memory

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>A</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>0010</td>
<td>C</td>
</tr>
<tr>
<td>0011</td>
<td>D</td>
</tr>
<tr>
<td>0100</td>
<td>E</td>
</tr>
<tr>
<td>0101</td>
<td>F</td>
</tr>
<tr>
<td>0110</td>
<td>G</td>
</tr>
<tr>
<td>0111</td>
<td>H</td>
</tr>
<tr>
<td>1000</td>
<td>J</td>
</tr>
<tr>
<td>1001</td>
<td>K</td>
</tr>
<tr>
<td>1010</td>
<td>L</td>
</tr>
<tr>
<td>1011</td>
<td>M</td>
</tr>
<tr>
<td>1100</td>
<td>N</td>
</tr>
<tr>
<td>1101</td>
<td>O</td>
</tr>
<tr>
<td>1110</td>
<td>P</td>
</tr>
<tr>
<td>1111</td>
<td>Q</td>
</tr>
</tbody>
</table>

- Byte-addressable memory
- 4 address bits $\rightarrow$ 16 bytes total
- $b$ addr bits $\rightarrow 2^b$ bytes in memory

load 1100 $\rightarrow$ r1
4-Byte, Direct Mapped Cache

**Direct mapped:**

- Each address maps to 1 cache block
- 4 entries → 2 index bits ($2^n \rightarrow n$ bits)

**Index with LSB:**

- Supports spatial locality
Analogy to a Spice Rack

Spice Rack (Cache)

index  spice
A   B   C   D   E   F   ...
Z

Spice Wall (Memory)

• Compared to your spice wall
  • Smaller
  • Faster
  • More costly (per oz.)

www.bedbathandbeyond.com
Analogy to a Spice Rack

- How do you know what’s in the jar?
- Need labels

**Tag** = Ultra-minimalist label
### 4-Byte, Direct Mapped Cache

**Tag:** minimalist label/address

**address = tag + index**

<table>
<thead>
<tr>
<th>index</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>A</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>C</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>D</td>
</tr>
</tbody>
</table>

**Tag**

```
tag | index
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>XXXX</td>
</tr>
</tbody>
</table>
```

**MEMORY**

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>A</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>0010</td>
<td>C</td>
</tr>
<tr>
<td>0011</td>
<td>D</td>
</tr>
<tr>
<td>0100</td>
<td>E</td>
</tr>
<tr>
<td>0101</td>
<td>F</td>
</tr>
<tr>
<td>0110</td>
<td>G</td>
</tr>
<tr>
<td>0111</td>
<td>H</td>
</tr>
<tr>
<td>1000</td>
<td>J</td>
</tr>
<tr>
<td>1001</td>
<td>K</td>
</tr>
<tr>
<td>1010</td>
<td>L</td>
</tr>
<tr>
<td>1011</td>
<td>M</td>
</tr>
<tr>
<td>1100</td>
<td>N</td>
</tr>
<tr>
<td>1101</td>
<td>O</td>
</tr>
<tr>
<td>1110</td>
<td>P</td>
</tr>
<tr>
<td>1111</td>
<td>Q</td>
</tr>
</tbody>
</table>
4-Byte, Direct Mapped Cache

One last tweak: valid bit
Simulation #1 of a 4-byte, DM Cache

- **Load:** 1100
- **Miss:**

**MEMORY**

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>A</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>0010</td>
<td>C</td>
</tr>
<tr>
<td>0011</td>
<td>D</td>
</tr>
<tr>
<td>0100</td>
<td>E</td>
</tr>
<tr>
<td>0101</td>
<td>F</td>
</tr>
<tr>
<td>0110</td>
<td>G</td>
</tr>
<tr>
<td>0111</td>
<td>H</td>
</tr>
<tr>
<td>1000</td>
<td>J</td>
</tr>
<tr>
<td>1001</td>
<td>K</td>
</tr>
<tr>
<td>1010</td>
<td>L</td>
</tr>
<tr>
<td>1011</td>
<td>M</td>
</tr>
<tr>
<td>1100</td>
<td>N</td>
</tr>
<tr>
<td>1101</td>
<td>O</td>
</tr>
<tr>
<td>1110</td>
<td>P</td>
</tr>
<tr>
<td>1111</td>
<td>Q</td>
</tr>
</tbody>
</table>

**CACHE**

<table>
<thead>
<tr>
<th>index</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>011</td>
<td>X</td>
</tr>
<tr>
<td>01</td>
<td>011</td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td>011</td>
<td>X</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>X</td>
</tr>
</tbody>
</table>

** Lookup:**
- Index into $\$
- Check tag
- Check valid bit
Simulation #1 of a 4-byte, DM Cache

tag|index
xxxx

load 1100 Miss

Lookup:
• Index into $
• Check tag
• Check valid bit

33
Simulation #1 of a 4-byte, DM Cache

**CACHE**

<table>
<thead>
<tr>
<th>index</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>N</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>X</td>
</tr>
</tbody>
</table>

**MEMORY**

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>A</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>0010</td>
<td>C</td>
</tr>
<tr>
<td>0011</td>
<td>D</td>
</tr>
<tr>
<td>0100</td>
<td>E</td>
</tr>
<tr>
<td>0101</td>
<td>F</td>
</tr>
<tr>
<td>0110</td>
<td>G</td>
</tr>
<tr>
<td>0111</td>
<td>H</td>
</tr>
<tr>
<td>1000</td>
<td>J</td>
</tr>
<tr>
<td>1001</td>
<td>K</td>
</tr>
<tr>
<td>1010</td>
<td>L</td>
</tr>
<tr>
<td>1011</td>
<td>M</td>
</tr>
<tr>
<td>1100</td>
<td>N</td>
</tr>
<tr>
<td>1101</td>
<td>O</td>
</tr>
<tr>
<td>1110</td>
<td>P</td>
</tr>
<tr>
<td>1111</td>
<td>Q</td>
</tr>
</tbody>
</table>

**Lookup:**
- **Index** into $\$
- Check **tag**
- Check **valid bit**

Awesome!
Clicker Questions

Processor A is a 1 GHz processor (1 cycle = 1 ns). There is an L1 cache with a 1 cycle access time and a 50% hit rate. There is an L2 cache with a 10 cycle access time and a 90% hit rate. It takes 100 ns to access memory.

Q1: What is the average memory access time of Processor A in ns?
A: 5   B: 7   C: 9   D: 11   E: 13

Proc. B attempts to improve upon Proc. A by making the L1 cache larger. The new hit rate is 90%. To maintain a 1 cycle access time, Proc. B runs at 500 MHz (1 cycle = 2 ns).

Q2: What is the average memory access time of Processor B in ns?
A: 5   B: 7   C: 9   D: 11   E: 13

Q3: Which processor should you buy?
A: Processor A   B: Processor B   C: They are equally good.
Clicker Answers

• Q1:
  \[1 + 50\% \times (10 + 10\% \times 100) =\]
  \[1 + 5 + 5 = 11 \text{ ns}\]

• Q2:
  \[2 + 10\% \times (20 + 10\% \times 100) =\]
  \[2 + 2 + 1 = 5 \text{ ns}\]

**Processor A.** Although memory access times are over 2x faster on Processor B, the frequency of Processor B is 2x slower. Since most workloads are not more than 50% memory instructions, Processor A is still likely to be the faster processor for most workloads. (If I did have workloads that were over 50% memory instructions, Processor B might be the better choice.)
Block Diagram
4-entry, direct mapped Cache

Great!
Are we done?
Simulation #2: 4-byte, DM Cache

Clicker: A) Hit  B) Miss

Load 1100
Load 1101
Load 0100
Load 1100

Lookup:  
- Index into $ 
- Check tag 
- Check valid bit

Index into $
Simulation #2: 4-byte, DM Cache

**Cache:***

<table>
<thead>
<tr>
<th>index</th>
<th>V</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>11</td>
<td>N</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>11</td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>11</td>
<td>X</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

**Memory:**

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>A</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>0010</td>
<td>C</td>
</tr>
<tr>
<td>0011</td>
<td>D</td>
</tr>
<tr>
<td>0100</td>
<td>E</td>
</tr>
<tr>
<td>0101</td>
<td>F</td>
</tr>
<tr>
<td>0110</td>
<td>G</td>
</tr>
<tr>
<td>0111</td>
<td>H</td>
</tr>
<tr>
<td>1000</td>
<td>J</td>
</tr>
<tr>
<td>1001</td>
<td>K</td>
</tr>
<tr>
<td>1010</td>
<td>L</td>
</tr>
<tr>
<td>1011</td>
<td>M</td>
</tr>
<tr>
<td>1100</td>
<td>N</td>
</tr>
<tr>
<td>1101</td>
<td>O</td>
</tr>
<tr>
<td>1110</td>
<td>P</td>
</tr>
<tr>
<td>1111</td>
<td>Q</td>
</tr>
</tbody>
</table>

**Lookup:**

- **Index into $**
- **Check tag**
- **Check valid bit**

**Load Actions:**

- **load 1100 Miss**
- **load 1101**
- **load 0100**
- **load 1100**
Simulation #2: 4-byte, DM Cache

Clicker: A) Hit B) Miss

│ index │ V  │ tag │ data │
│-------|----|-----|------|
│ 00    │ 1  | 11  | N    |
│ 01    │ 0  | 11  | X    |
│ 10    │ 0  | 11  | X    |
│ 11    │ 0  | 11  | X    |

Load 1100: Miss
Load 1101: Miss
Load 0100
Load 1100

Lookup:
- Index into $
- Check tag
- Check valid bit

Memory:

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>A</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>0010</td>
<td>C</td>
</tr>
<tr>
<td>0011</td>
<td>D</td>
</tr>
<tr>
<td>0100</td>
<td>E</td>
</tr>
<tr>
<td>0101</td>
<td>F</td>
</tr>
<tr>
<td>0110</td>
<td>G</td>
</tr>
<tr>
<td>0111</td>
<td>H</td>
</tr>
<tr>
<td>1000</td>
<td>J</td>
</tr>
<tr>
<td>1001</td>
<td>K</td>
</tr>
<tr>
<td>1010</td>
<td>L</td>
</tr>
<tr>
<td>1011</td>
<td>M</td>
</tr>
<tr>
<td>1100</td>
<td>N</td>
</tr>
<tr>
<td>1101</td>
<td>O</td>
</tr>
<tr>
<td>1110</td>
<td>P</td>
</tr>
<tr>
<td>1111</td>
<td>Q</td>
</tr>
</tbody>
</table>
Simulation #2: 4-byte, DM Cache

**Cache Lookup:**
- Index into $\$
- Check tag
- Check valid bit

**Load Addresses:**
- 1100: Miss
- 1101: Miss
- 0100
- 1100

**Memory Table:**

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>A</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>0010</td>
<td>C</td>
</tr>
<tr>
<td>0011</td>
<td>D</td>
</tr>
<tr>
<td>0100</td>
<td>E</td>
</tr>
<tr>
<td>0101</td>
<td>F</td>
</tr>
<tr>
<td>0110</td>
<td>G</td>
</tr>
<tr>
<td>0111</td>
<td>H</td>
</tr>
<tr>
<td>1000</td>
<td>J</td>
</tr>
<tr>
<td>1001</td>
<td>K</td>
</tr>
<tr>
<td>1010</td>
<td>L</td>
</tr>
<tr>
<td>1011</td>
<td>M</td>
</tr>
<tr>
<td>1100</td>
<td>N</td>
</tr>
<tr>
<td>1101</td>
<td>O</td>
</tr>
<tr>
<td>1110</td>
<td>P</td>
</tr>
<tr>
<td>1111</td>
<td>Q</td>
</tr>
</tbody>
</table>
Simulation #2: 4-byte, DM Cache

Clicker: A) Hit B) Miss

Load: 1100
Miss
Load: 1101
Miss
Load: 0100
Miss
Load: 1100

Lookup:
- Index into $
- Check tag
- Check valid bit

CaCHe

<table>
<thead>
<tr>
<th>index</th>
<th>V</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>11</td>
<td>N</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>11</td>
<td>O</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>11</td>
<td>X</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

MEMORY

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>A</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>0010</td>
<td>C</td>
</tr>
<tr>
<td>0011</td>
<td>D</td>
</tr>
<tr>
<td>0100</td>
<td>E</td>
</tr>
<tr>
<td>0101</td>
<td>F</td>
</tr>
<tr>
<td>0110</td>
<td>G</td>
</tr>
<tr>
<td>0111</td>
<td>H</td>
</tr>
<tr>
<td>1000</td>
<td>J</td>
</tr>
<tr>
<td>1001</td>
<td>K</td>
</tr>
<tr>
<td>1010</td>
<td>L</td>
</tr>
<tr>
<td>1011</td>
<td>M</td>
</tr>
<tr>
<td>1100</td>
<td>N</td>
</tr>
<tr>
<td>1101</td>
<td>O</td>
</tr>
<tr>
<td>1110</td>
<td>P</td>
</tr>
<tr>
<td>1111</td>
<td>Q</td>
</tr>
</tbody>
</table>
Simulation #2: 4-byte, DM Cache

**Memory**

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>A</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>0010</td>
<td>C</td>
</tr>
<tr>
<td>0011</td>
<td>D</td>
</tr>
<tr>
<td>0100</td>
<td>E</td>
</tr>
<tr>
<td>0101</td>
<td>F</td>
</tr>
<tr>
<td>0110</td>
<td>G</td>
</tr>
<tr>
<td>0111</td>
<td>H</td>
</tr>
<tr>
<td>1000</td>
<td>J</td>
</tr>
<tr>
<td>1001</td>
<td>K</td>
</tr>
<tr>
<td>1010</td>
<td>L</td>
</tr>
<tr>
<td>1011</td>
<td>M</td>
</tr>
<tr>
<td>1100</td>
<td>N</td>
</tr>
<tr>
<td>1101</td>
<td>O</td>
</tr>
<tr>
<td>1110</td>
<td>P</td>
</tr>
<tr>
<td>1111</td>
<td>Q</td>
</tr>
</tbody>
</table>

**Cache**

<table>
<thead>
<tr>
<th>index</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>E</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>O</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>X</td>
</tr>
</tbody>
</table>

**Lookup:**
- Index into $\$
- Check tag
- Check valid bit

**Example:**
- load 1100 Miss
- load 1101 Miss
- load 0100 Miss
- load 1100
Simulation #2: 4-byte, DM Cache

Clicker: A) Hit  
B) Miss

**CACHE**

<table>
<thead>
<tr>
<th>index</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>E</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>O</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>X</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td>X</td>
</tr>
</tbody>
</table>

**MEMORY**

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>A</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>0010</td>
<td>C</td>
</tr>
<tr>
<td>0011</td>
<td>D</td>
</tr>
<tr>
<td>0100</td>
<td>E</td>
</tr>
<tr>
<td>0101</td>
<td>F</td>
</tr>
<tr>
<td>0110</td>
<td>G</td>
</tr>
<tr>
<td>0111</td>
<td>H</td>
</tr>
<tr>
<td>1000</td>
<td>J</td>
</tr>
<tr>
<td>1001</td>
<td>K</td>
</tr>
<tr>
<td>1010</td>
<td>L</td>
</tr>
<tr>
<td>1011</td>
<td>M</td>
</tr>
<tr>
<td>1100</td>
<td>N</td>
</tr>
<tr>
<td>1101</td>
<td>O</td>
</tr>
<tr>
<td>1110</td>
<td>P</td>
</tr>
<tr>
<td>1111</td>
<td>Q</td>
</tr>
</tbody>
</table>

**Simulation #2:**

4-byte, DM Cache

**Lookup:**
- Index into $V$
- Check tag
- Check valid bit

- load 1100  Miss
- load 1101  Miss
- load 0100  Miss
- load 1100  Miss
Simulation #2: 4-byte, DM Cache

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>N</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>O</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>X</td>
</tr>
</tbody>
</table>

load 1100  Miss cold
load 1101  Miss cold
load 0100  Miss cold
load 1100  Miss

Disappointed!
Reducing Cold Misses by Increasing Block Size

- Leveraging Spatial Locality
## Increasing Block Size

### CACHE

<table>
<thead>
<tr>
<th>offset</th>
<th>index</th>
<th>V</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>0</td>
<td>x</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>0</td>
<td>x</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0</td>
<td>x</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>0</td>
<td>x</td>
<td>G</td>
</tr>
</tbody>
</table>

### MEMORY

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>A</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>0010</td>
<td>C</td>
</tr>
<tr>
<td>0011</td>
<td>D</td>
</tr>
<tr>
<td>0100</td>
<td>E</td>
</tr>
<tr>
<td>0101</td>
<td>F</td>
</tr>
<tr>
<td>0110</td>
<td>G</td>
</tr>
<tr>
<td>0111</td>
<td>H</td>
</tr>
<tr>
<td>1000</td>
<td>J</td>
</tr>
<tr>
<td>1001</td>
<td>K</td>
</tr>
<tr>
<td>1010</td>
<td>L</td>
</tr>
<tr>
<td>1011</td>
<td>M</td>
</tr>
<tr>
<td>1100</td>
<td>N</td>
</tr>
<tr>
<td>1101</td>
<td>O</td>
</tr>
<tr>
<td>1110</td>
<td>P</td>
</tr>
<tr>
<td>1111</td>
<td>Q</td>
</tr>
</tbody>
</table>

- **Block Size:** 2 bytes
- **Block Offset:** least significant bits indicate where you live in the block
- **Which bits are the index? tag?**
Simulation #3: 8-byte, DM Cache

**MEMORY**

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>A</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>0010</td>
<td>C</td>
</tr>
<tr>
<td>0011</td>
<td>D</td>
</tr>
<tr>
<td>0100</td>
<td>E</td>
</tr>
<tr>
<td>0101</td>
<td>F</td>
</tr>
<tr>
<td>0110</td>
<td>G</td>
</tr>
<tr>
<td>0111</td>
<td>H</td>
</tr>
<tr>
<td>1000</td>
<td>J</td>
</tr>
<tr>
<td>1001</td>
<td>K</td>
</tr>
<tr>
<td>1010</td>
<td>L</td>
</tr>
<tr>
<td>1011</td>
<td>M</td>
</tr>
<tr>
<td>1100</td>
<td>N</td>
</tr>
<tr>
<td>1101</td>
<td>O</td>
</tr>
<tr>
<td>1110</td>
<td>P</td>
</tr>
<tr>
<td>1111</td>
<td>Q</td>
</tr>
</tbody>
</table>

**CACHE**

<table>
<thead>
<tr>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Lookup:
- Index into $\$
- Check tag
- Check valid bit

load 1100  Miss
load 1101
load 0100
load 1100
Simulation #3: 8-byte, DM Cache

**CACHE**

<table>
<thead>
<tr>
<th>index</th>
<th>tag</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>N</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>X</td>
</tr>
</tbody>
</table>

**Lookup:**
- **Index into $**
- **Check tag**
- **Check valid bit**

**MEMORY**

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>A</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>0010</td>
<td>C</td>
</tr>
<tr>
<td>0011</td>
<td>D</td>
</tr>
<tr>
<td>0100</td>
<td>E</td>
</tr>
<tr>
<td>0101</td>
<td>F</td>
</tr>
<tr>
<td>0110</td>
<td>G</td>
</tr>
<tr>
<td>0111</td>
<td>H</td>
</tr>
<tr>
<td>1000</td>
<td>J</td>
</tr>
<tr>
<td>1001</td>
<td>K</td>
</tr>
<tr>
<td>1010</td>
<td>L</td>
</tr>
<tr>
<td>1011</td>
<td>M</td>
</tr>
<tr>
<td>1100</td>
<td>N</td>
</tr>
<tr>
<td>1101</td>
<td>O</td>
</tr>
<tr>
<td>1110</td>
<td>P</td>
</tr>
<tr>
<td>1111</td>
<td>Q</td>
</tr>
</tbody>
</table>

**Examples:**
- load 1100 Miss
- load 1101
- load 0100
- load 1100
Simulation #3: 8-byte, DM Cache

**CACHE**

<table>
<thead>
<tr>
<th>index</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0 x</td>
<td>X</td>
</tr>
<tr>
<td>01</td>
<td>0 x</td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td>1 1</td>
<td>N</td>
</tr>
<tr>
<td>11</td>
<td>0 x</td>
<td>X</td>
</tr>
</tbody>
</table>

**MEMORY**

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>A</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>0010</td>
<td>C</td>
</tr>
<tr>
<td>0011</td>
<td>D</td>
</tr>
<tr>
<td>0100</td>
<td>E</td>
</tr>
<tr>
<td>0101</td>
<td>F</td>
</tr>
<tr>
<td>0110</td>
<td>G</td>
</tr>
<tr>
<td>0111</td>
<td>H</td>
</tr>
<tr>
<td>1000</td>
<td>J</td>
</tr>
<tr>
<td>1001</td>
<td>K</td>
</tr>
<tr>
<td>1010</td>
<td>L</td>
</tr>
<tr>
<td>1011</td>
<td>M</td>
</tr>
<tr>
<td>1100</td>
<td>N</td>
</tr>
<tr>
<td>1101</td>
<td>O</td>
</tr>
<tr>
<td>1110</td>
<td>P</td>
</tr>
<tr>
<td>1111</td>
<td>Q</td>
</tr>
</tbody>
</table>

**Lookup:**
- Index into $index$
- Check tag
- Check valid bit

- **Miss**: load 1100
- **Hit!**: load 1101, load 0100, load 1100
Simulation #3: 8-byte, DM Cache

**CACHE**

<table>
<thead>
<tr>
<th>index</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>N</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>X</td>
</tr>
</tbody>
</table>

**MEMORY**

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>A</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>0010</td>
<td>C</td>
</tr>
<tr>
<td>0011</td>
<td>D</td>
</tr>
<tr>
<td>0100</td>
<td>E</td>
</tr>
<tr>
<td>0101</td>
<td>F</td>
</tr>
<tr>
<td>0110</td>
<td>G</td>
</tr>
<tr>
<td>0111</td>
<td>H</td>
</tr>
<tr>
<td>1000</td>
<td>J</td>
</tr>
<tr>
<td>1001</td>
<td>K</td>
</tr>
<tr>
<td>1010</td>
<td>L</td>
</tr>
<tr>
<td>1011</td>
<td>M</td>
</tr>
<tr>
<td>1100</td>
<td>N</td>
</tr>
<tr>
<td>1101</td>
<td>O</td>
</tr>
<tr>
<td>1110</td>
<td>P</td>
</tr>
<tr>
<td>1111</td>
<td>Q</td>
</tr>
</tbody>
</table>

**Lookup:**
- Index into $\$
- Check tag
- Check valid bit

**Miss**
- load 1100
- load 1101
- load 0100
- load 1100

**Hit!**
- load 1100
Simulation #3: 8-byte, DM Cache

**Lookup:**
- Index into $x$
- Check tag
- Check valid bit

**CACHE**

<table>
<thead>
<tr>
<th>index</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>E</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>X</td>
</tr>
</tbody>
</table>

**MEMORY**

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>A</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>0010</td>
<td>C</td>
</tr>
<tr>
<td>0011</td>
<td>D</td>
</tr>
<tr>
<td>0100</td>
<td>E</td>
</tr>
<tr>
<td>0101</td>
<td>F</td>
</tr>
<tr>
<td>0110</td>
<td>G</td>
</tr>
<tr>
<td>0111</td>
<td>H</td>
</tr>
<tr>
<td>1000</td>
<td>J</td>
</tr>
<tr>
<td>1001</td>
<td>K</td>
</tr>
<tr>
<td>1010</td>
<td>L</td>
</tr>
<tr>
<td>1011</td>
<td>M</td>
</tr>
<tr>
<td>1100</td>
<td>N</td>
</tr>
<tr>
<td>1101</td>
<td>O</td>
</tr>
<tr>
<td>1110</td>
<td>P</td>
</tr>
<tr>
<td>1111</td>
<td>Q</td>
</tr>
</tbody>
</table>

**Example:**
- load 1100  Miss
- load 1101  Hit!
- load 0100  Miss
- load 1100  Miss
Simulation #3: 8-byte, DM Cache

**CACHE**

<table>
<thead>
<tr>
<th>V</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>E</td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td>X</td>
</tr>
</tbody>
</table>

**MEMORY**

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>A</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>0010</td>
<td>C</td>
</tr>
<tr>
<td>0011</td>
<td>D</td>
</tr>
<tr>
<td>0100</td>
<td>E</td>
</tr>
<tr>
<td>0101</td>
<td>F</td>
</tr>
<tr>
<td>0110</td>
<td>G</td>
</tr>
<tr>
<td>0111</td>
<td>H</td>
</tr>
<tr>
<td>1000</td>
<td>J</td>
</tr>
<tr>
<td>1001</td>
<td>K</td>
</tr>
<tr>
<td>1010</td>
<td>L</td>
</tr>
<tr>
<td>1011</td>
<td>M</td>
</tr>
<tr>
<td>1100</td>
<td>N</td>
</tr>
<tr>
<td>1101</td>
<td>O</td>
</tr>
<tr>
<td>1110</td>
<td>P</td>
</tr>
<tr>
<td>1111</td>
<td>Q</td>
</tr>
</tbody>
</table>

**Lookup:**
- Index into $index$
- Check tag
- Check valid bit

- load 1100: Miss
- load 1101: Hit!
- load 0100: Miss
- load 1100: Miss

- Miss
- Hit!
Simulation #3: 8-byte, DM Cache

<table>
<thead>
<tr>
<th>index</th>
<th>V</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>x</td>
<td>X</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>x</td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>E</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>x</td>
<td>X</td>
</tr>
</tbody>
</table>

load 1100  Miss cold  1 hit, 3 misses
load 1101  Hit!
load 0100  Miss cold  3 bytes don’t fit in a 4 entry cache?
load 1100  Miss conflict

MEMORY

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>A</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>0010</td>
<td>C</td>
</tr>
<tr>
<td>0011</td>
<td>D</td>
</tr>
<tr>
<td>0100</td>
<td>E</td>
</tr>
<tr>
<td>0101</td>
<td>F</td>
</tr>
<tr>
<td>0110</td>
<td>G</td>
</tr>
<tr>
<td>0111</td>
<td>H</td>
</tr>
<tr>
<td>1000</td>
<td>J</td>
</tr>
<tr>
<td>1001</td>
<td>K</td>
</tr>
<tr>
<td>1010</td>
<td>L</td>
</tr>
<tr>
<td>1011</td>
<td>M</td>
</tr>
<tr>
<td>1100</td>
<td>N</td>
</tr>
<tr>
<td>1101</td>
<td>O</td>
</tr>
<tr>
<td>1110</td>
<td>P</td>
</tr>
<tr>
<td>1111</td>
<td>Q</td>
</tr>
</tbody>
</table>
1 Cycle Per Stage: the Biggest Lie (So Far)

Code Stored in Memory (also, data and stack)
Removing Conflict Misses with Fully-Associative Caches
8 byte, fully-associative Cache

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>A</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>0010</td>
<td>C</td>
</tr>
<tr>
<td>0011</td>
<td>D</td>
</tr>
<tr>
<td>0100</td>
<td>E</td>
</tr>
<tr>
<td>0101</td>
<td>F</td>
</tr>
<tr>
<td>0110</td>
<td>G</td>
</tr>
<tr>
<td>0111</td>
<td>H</td>
</tr>
<tr>
<td>1000</td>
<td>J</td>
</tr>
<tr>
<td>1001</td>
<td>K</td>
</tr>
<tr>
<td>1010</td>
<td>L</td>
</tr>
<tr>
<td>1011</td>
<td>M</td>
</tr>
<tr>
<td>1100</td>
<td>N</td>
</tr>
<tr>
<td>1101</td>
<td>O</td>
</tr>
<tr>
<td>1110</td>
<td>P</td>
</tr>
<tr>
<td>1111</td>
<td>Q</td>
</tr>
</tbody>
</table>

What should the offset be?

What should the index be?

What should the tag be?

Clicker:

A) xxxx
B) xxxx
C) xxxx
D) xxxx
E) None
Simulation #4: 8-byte, FA Cache

### CACHE

<table>
<thead>
<tr>
<th>V</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>xxx</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>xxx</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>xxx</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>xxx</td>
<td>X</td>
</tr>
</tbody>
</table>

#### MEMORY

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>A</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>0010</td>
<td>C</td>
</tr>
<tr>
<td>0011</td>
<td>D</td>
</tr>
<tr>
<td>0100</td>
<td>E</td>
</tr>
<tr>
<td>0101</td>
<td>F</td>
</tr>
<tr>
<td>0110</td>
<td>G</td>
</tr>
<tr>
<td>0111</td>
<td>H</td>
</tr>
<tr>
<td>1000</td>
<td>J</td>
</tr>
<tr>
<td>1001</td>
<td>K</td>
</tr>
<tr>
<td>1010</td>
<td>L</td>
</tr>
<tr>
<td>1011</td>
<td>M</td>
</tr>
<tr>
<td>1100</td>
<td>N</td>
</tr>
<tr>
<td>1101</td>
<td>O</td>
</tr>
<tr>
<td>1110</td>
<td>P</td>
</tr>
<tr>
<td>1111</td>
<td>Q</td>
</tr>
</tbody>
</table>

**Lookup:**
- **Index** into $\$
- Check **tags**
- Check **valid bits**

**LRU Pointer**

**Load:**
- 1100: Miss
- 1101
- 0100
- 1100

**LRU Pointer**
Simulation #4: 8-byte, FA Cache

lookup:
- Index into 
- Check tags
- Check valid bits

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>A</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>0010</td>
<td>C</td>
</tr>
<tr>
<td>0011</td>
<td>D</td>
</tr>
<tr>
<td>0100</td>
<td>E</td>
</tr>
<tr>
<td>0101</td>
<td>F</td>
</tr>
<tr>
<td>0110</td>
<td>G</td>
</tr>
<tr>
<td>0111</td>
<td>H</td>
</tr>
<tr>
<td>1000</td>
<td>J</td>
</tr>
<tr>
<td>1001</td>
<td>K</td>
</tr>
<tr>
<td>1010</td>
<td>L</td>
</tr>
<tr>
<td>1011</td>
<td>M</td>
</tr>
<tr>
<td>1100</td>
<td>N</td>
</tr>
<tr>
<td>1101</td>
<td>O</td>
</tr>
<tr>
<td>1110</td>
<td>P</td>
</tr>
<tr>
<td>1111</td>
<td>Q</td>
</tr>
</tbody>
</table>

load 1100 Miss
load 1101 Hit!
load 0100
load 1100
Simulation #4: 8-byte, FA Cache

**CACHE**

<table>
<thead>
<tr>
<th>V</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>110</td>
<td>N</td>
</tr>
<tr>
<td>0</td>
<td>xxx</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>xxx</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>xxx</td>
<td>X</td>
</tr>
</tbody>
</table>

**MEMORY**

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>A</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>0010</td>
<td>C</td>
</tr>
<tr>
<td>0011</td>
<td>D</td>
</tr>
<tr>
<td>0100</td>
<td>E</td>
</tr>
<tr>
<td>0101</td>
<td>F</td>
</tr>
<tr>
<td>0110</td>
<td>G</td>
</tr>
<tr>
<td>0111</td>
<td>H</td>
</tr>
<tr>
<td>1000</td>
<td>J</td>
</tr>
<tr>
<td>1001</td>
<td>K</td>
</tr>
<tr>
<td>1010</td>
<td>L</td>
</tr>
<tr>
<td>1011</td>
<td>M</td>
</tr>
<tr>
<td>1100</td>
<td>N</td>
</tr>
<tr>
<td>1101</td>
<td>O</td>
</tr>
<tr>
<td>1110</td>
<td>P</td>
</tr>
<tr>
<td>1111</td>
<td>Q</td>
</tr>
</tbody>
</table>

**Load Operations:**
- load 1100 Miss
- load 1101 Hit!
- load 0100 Miss
- load 1100

**Lookup Algorithm:**
- Index into $\text{MEMORY}$
- Check tags
- Check valid bits

**LRU Pointer**
Simulation #4: 8-byte, FA Cache

Load

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>A</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>0010</td>
<td>C</td>
</tr>
<tr>
<td>0011</td>
<td>D</td>
</tr>
<tr>
<td>0100</td>
<td>E</td>
</tr>
<tr>
<td>0101</td>
<td>F</td>
</tr>
<tr>
<td>0110</td>
<td>G</td>
</tr>
<tr>
<td>0111</td>
<td>H</td>
</tr>
<tr>
<td>1000</td>
<td>J</td>
</tr>
<tr>
<td>1001</td>
<td>K</td>
</tr>
<tr>
<td>1010</td>
<td>L</td>
</tr>
<tr>
<td>1011</td>
<td>M</td>
</tr>
<tr>
<td>1100</td>
<td>N</td>
</tr>
<tr>
<td>1101</td>
<td>O</td>
</tr>
<tr>
<td>1110</td>
<td>P</td>
</tr>
<tr>
<td>1111</td>
<td>Q</td>
</tr>
</tbody>
</table>

**Cache**

Load 1100 Miss
Load 1101 Hit!
Load 0100 Miss
Load 1100 Hit!

**Lookup:**
- Index into $\$
- Check tags
- Check valid bits

**LRU Pointer**
Pros and Cons of Full Associativity

+ No more conflicts!
+ Excellent utilization!

But either:

Parallel Reads
– lots of reading!

Serial Reads
– lots of waiting

\[ t_{\text{avg}} = t_{\text{hit}} + \%_{\text{miss}} \times t_{\text{miss}} \]

\[ = 4 + 5\% \times 100 \]
\[ = 9 \text{ cycles} \]

\[ = 6 + 3\% \times 100 \]
\[ = 9 \text{ cycles} \]
## Pros & Cons

<table>
<thead>
<tr>
<th></th>
<th>Direct Mapped</th>
<th>Fully Associative</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tag Size</strong></td>
<td>Smaller</td>
<td>Larger</td>
</tr>
<tr>
<td><strong>SRAM Overhead</strong></td>
<td>Less</td>
<td>More</td>
</tr>
<tr>
<td><strong>Controller Logic</strong></td>
<td>Less</td>
<td>More</td>
</tr>
<tr>
<td><strong>Speed</strong></td>
<td>Faster</td>
<td>Slower</td>
</tr>
<tr>
<td><strong>Price</strong></td>
<td>Less</td>
<td>More</td>
</tr>
<tr>
<td><strong>Scalability</strong></td>
<td>Very</td>
<td>Not Very</td>
</tr>
<tr>
<td># of conflict misses</td>
<td>Lots</td>
<td>Zero</td>
</tr>
<tr>
<td><strong>Hit Rate</strong></td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td><strong>Pathological Cases</strong></td>
<td>Common</td>
<td>?</td>
</tr>
</tbody>
</table>
Reducing Conflict Misses with Set-Associative Caches

Not too conflict-y. Not too slow.

… Just Right!
8 byte, 2-way set associative Cache

What should the **offset** be?
What should the **index** be?
What should the **tag** be?
Clicker Question

5 bit address
2 byte block size
24 byte, 3-Way Set Associative CACHE

How many tag bits?

A) 0
B) 1
C) 2
D) 3
E) 4
Clicker Question

5 bit address
2 byte block size
24 byte, 3-Way Set Associative CACHE

<table>
<thead>
<tr>
<th>index</th>
<th>V</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>?</td>
<td>X</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>?</td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>?</td>
<td>X</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>?</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index</th>
<th>V</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>?</td>
<td>X'</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>?</td>
<td>X'</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>?</td>
<td>X'</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>?</td>
<td>X'</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index</th>
<th>V</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>?</td>
<td>X''</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>?</td>
<td>X''</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>?</td>
<td>X''</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>?</td>
<td>X''</td>
</tr>
</tbody>
</table>

A) 0
B) 1
C) 2
D) 3
E) 4

How many tag bits?
8 byte, 2-way set associative Cache

MEMORY

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>A</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>0010</td>
<td>C</td>
</tr>
<tr>
<td>0011</td>
<td>D</td>
</tr>
<tr>
<td>0100</td>
<td>E</td>
</tr>
<tr>
<td>0101</td>
<td>F</td>
</tr>
<tr>
<td>0110</td>
<td>G</td>
</tr>
<tr>
<td>0111</td>
<td>H</td>
</tr>
<tr>
<td>1000</td>
<td>J</td>
</tr>
<tr>
<td>1001</td>
<td>K</td>
</tr>
<tr>
<td>1010</td>
<td>L</td>
</tr>
<tr>
<td>1011</td>
<td>M</td>
</tr>
<tr>
<td>1100</td>
<td>N</td>
</tr>
<tr>
<td>1101</td>
<td>O</td>
</tr>
<tr>
<td>1110</td>
<td>P</td>
</tr>
<tr>
<td>1111</td>
<td>Q</td>
</tr>
</tbody>
</table>

CACHE

<table>
<thead>
<tr>
<th>index</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>xx</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>X</td>
</tr>
</tbody>
</table>

Lookup:
- Index into $V$
- Check tag
- Check valid bit

LRU Pointer

Load 1100: Miss
Load 1101
Load 0100
Load 1100
8 byte, 2-way set associative Cache

**CACHE**

<table>
<thead>
<tr>
<th>index</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>N</td>
</tr>
<tr>
<td>0</td>
<td>xx</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>X</td>
</tr>
</tbody>
</table>

**MEMORY**

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>A</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>0010</td>
<td>C</td>
</tr>
<tr>
<td>0011</td>
<td>D</td>
</tr>
<tr>
<td>0100</td>
<td>E</td>
</tr>
<tr>
<td>0101</td>
<td>F</td>
</tr>
<tr>
<td>0110</td>
<td>G</td>
</tr>
<tr>
<td>0111</td>
<td>H</td>
</tr>
<tr>
<td>1000</td>
<td>J</td>
</tr>
<tr>
<td>1001</td>
<td>K</td>
</tr>
<tr>
<td>1010</td>
<td>L</td>
</tr>
<tr>
<td>1011</td>
<td>M</td>
</tr>
<tr>
<td>1100</td>
<td>N</td>
</tr>
<tr>
<td>1101</td>
<td>O</td>
</tr>
<tr>
<td>1110</td>
<td>P</td>
</tr>
<tr>
<td>1111</td>
<td>Q</td>
</tr>
</tbody>
</table>

Load 1100
Load 1101
Load 0100
Load 1100

**Lookup:**
- Index into $
- Check tag
- Check valid bit

Miss
Hit!
8 byte, 2-way set associative Cache

Lookup:
- Index into $
- Check tag
- Check valid bit

LRU Pointer

load 1100 Miss
load 1101 Hit!
load 0100 Miss
load 1100
8 byte, 2-way set associative Cache

**Cache Entry:**

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>N</td>
</tr>
<tr>
<td>0</td>
<td>xx</td>
<td>X</td>
</tr>
</tbody>
</table>

**Memory:**

<table>
<thead>
<tr>
<th>Addr</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>A</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>0010</td>
<td>C</td>
</tr>
<tr>
<td>0011</td>
<td>D</td>
</tr>
<tr>
<td>0100</td>
<td>E</td>
</tr>
<tr>
<td>0101</td>
<td>F</td>
</tr>
<tr>
<td>0110</td>
<td>G</td>
</tr>
<tr>
<td>0111</td>
<td>H</td>
</tr>
<tr>
<td>1000</td>
<td>I</td>
</tr>
<tr>
<td>1001</td>
<td>J</td>
</tr>
<tr>
<td>1010</td>
<td>K</td>
</tr>
<tr>
<td>1011</td>
<td>L</td>
</tr>
<tr>
<td>1100</td>
<td>M</td>
</tr>
<tr>
<td>1101</td>
<td>N</td>
</tr>
<tr>
<td>1110</td>
<td>O</td>
</tr>
<tr>
<td>1111</td>
<td>P</td>
</tr>
</tbody>
</table>

**Lookup:**

- Index into $V$
- Check tag
- Check valid bit

**LRU Pointer**
Eviction Policies

Which cache line should be evicted from the cache to make room for a new line?

- Direct-mapped: no choice, must evict line selected by index
- Associative caches
  - Random: select one of the lines at random
  - Round-Robin: similar to random
  - FIFO: replace oldest line
  - LRU: replace line that has not been used in the longest time
Misses: the Three C’s

❄️ Cold (compulsory) Miss: never seen this address before

☮️ Conflict Miss: cache associativity is too low

truck Capacity Miss: cache is too small
Block Size Tradeoffs

• For a given total cache size,
  Larger block sizes mean….
  • fewer lines
  • so fewer tags, less overhead
  • and fewer cold misses (within-block “prefetching”)

• But also…
  • fewer blocks available (for scattered accesses!)
  • so more conflicts
  • can decrease performance if working set can’t fit in $\$$
  • and larger miss penalty (time to fetch block)
Miss Rate vs. Associativity

The graph shows the relationship between miss rate and associativity for different cache sizes:
- 1 KiB
- 2 KiB
- 4 KiB
- 8 KiB
- 16 KiB
- 32 KiB
- 64 KiB
- 128 KiB

The graph indicates that as associativity increases from one-way to eight-way, the miss rate decreases, especially for larger cache sizes.
Clicker Question

What does NOT happen when you increase the associativity of the cache?

A) Conflict misses decrease
B) Tag overhead decreases
C) Hit time increases
D) Cache stays the same size
Clicker Question

What does **NOT** happen when you increase the associativity of the cache?

A) Conflict misses decrease
B) Tag overhead decreases
C) Hit time increases
D) Cache stays the same size
ABCs of Caches

\[ t_{avg} = t_{hit} + \%_{miss} * t_{miss} \]

+ Associativity:
  ↓ conflict misses 😊
  ↑ hit time 😞

+ Block Size:
  ↓ cold misses 😊
  ↑ conflict misses 😞

+ Capacity:
  ↓ capacity misses 😊
  ↑ hit time 😞
Which caches get what properties?

- **L1 Caches**
  - Fast
  - More Associative
- **L2 Cache**
  - Big
  - Bigger Block Sizes
- **L3 Cache**
  - Larger Capacity

\[ t_{avg} = t_{hit} + \%_{miss} \times t_{miss} \]

*Design with miss rate in mind*

*Design with speed in mind*
Roadmap

- **Things we have covered:**
  - The Need for Speed
  - Locality to the Rescue!
  - Calculating average memory access time
  - $ Misses: Cold, Conflict, Capacity
  - $ Characteristics: Associativity, Block Size, Capacity

- **Things we will now cover:**
  - Cache Figures
  - Cache Performance Examples
  - Writes
3-Way Set Associative Cache (Reading)

- Tag
- Index
- Offset

hit?

line select

word select

data

32 bits

64 bytes
How Big is the Cache?

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
</table>

$n$ bit index, $m$ bit offset, **N-way Set Associative**

Question: How big is cache?

- **Data only?**
  
  (what we usually mean when we ask “how big” is the cache)

- **Data + overhead?**
How Big is the Cache?

- $n$ bit index, $m$ bit offset, **N-way Set Associative**
- Question: How big is cache?
- **How big is the cache (Data only)?**

Cache of size $2^n$ sets
Block size of $2^m$ bytes, $N$-way set associative
Cache Size: $2^m$ bytes-per-block x ($2^n$ sets x $N$-way-per-set)
  
  $= N \times 2^{n+m}$ bytes
How Big is the Cache?

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
</table>

$n$ bit index, $m$ bit offset, **N-way Set Associative**

Question: How big is cache?

• **How big is the cache (Data + overhead)?**

Cache of size $2^n$ sets
Block size of $2^m$ bytes, $N$-way set associative
Tag field: $32 - (n + m)$, Valid bit: 1
SRAM Size: $2^n$ sets $\times N$-way-per-set $\times$ (block size $+$ tag size $+$ valid bit size)

$$= 2^n \times N\text{-way} \times (2^m \text{ bytes} \times 8 \text{ bits-per-byte} + (32 - n - m) + 1)$$
Performance Calculation with $ Hierarchy

- **Parameters**
  \[ t_{avg} = t_{hit} + \%_{miss} \times t_{miss} \]
- Reference stream: all loads
- D$: \( t_{hit} = 1\text{ns}, \%_{miss} = 5\% \)
- L2: \( t_{hit} = 10\text{ns}, \%_{miss} = 20\% \) (local miss rate)
- Main memory: \( t_{hit} = 50\text{ns} \)

- **What is** \( t_{avgD$} \) **without an L2?**
  \[ t_{missD$} = \]
  \[ t_{avgD$} = \]

- **What is** \( t_{avgD$} \) **with an L2?**
  \[ t_{missD$} = \]
  \[ t_{avgL2} = \]
  \[ t_{avgD$} = \]
Performance Calculation with $ Hierarchy

- **Parameters**
  - \( t_{avg} = t_{hit} + \%_{miss} \times t_{miss} \)
  - Reference stream: all loads
  - D$: \( t_{hit} = 1\text{ns}, \%_{miss} = 5\% \)
  - L2: \( t_{hit} = 10\text{ns}, \%_{miss} = 20\% \) (local miss rate)
  - Main memory: \( t_{hit} = 50\text{ns} \)

- **What is \( t_{avgD$} \) without an L2?**
  - \( t_{missD$} = t_{hitM} \)
  - \( t_{avgD$} = t_{hitD$} + \%_{missD$} \times t_{hitM} = 1\text{ns} + (0.05 \times 50\text{ns}) = 3.5\text{ns} \)

- **What is \( t_{avgD$} \) with an L2?**
  - \( t_{missD$} = t_{avgL2} \)
  - \( t_{avgL2} = t_{hitL2} + \%_{missL2} \times t_{hitM} = 10\text{ns} + (0.2 \times 50\text{ns}) = 20\text{ns} \)
  - \( t_{avgD$} = t_{hitD$} + \%_{missD$} \times t_{avgL2} = 1\text{ns} + (0.05 \times 20\text{ns}) = 2\text{ns} \)
Performance Summary

Average memory access time (AMAT) depends on:
- cache architecture and size
- Hit and miss rates
- Access times and miss penalty

Cache design a very complex problem:
- Cache size, block size (aka line size)
- Number of ways of set-associativity ($1, N, \infty$)
- Eviction policy
- Number of levels of caching, parameters for each
- Separate I-cache from D-cache, or Unified cache
- Prefetching policies / instructions
- Write policy
Takeaway

Direct Mapped → fast, but low hit rate
Fully Associative → higher hit cost, higher hit rate
Set Associative → middleground

Line size matters. Larger cache lines can increase performance due to prefetching. BUT, can also decrease performance is working set size cannot fit in cache.

Cache performance is measured by the average memory access time (AMAT), which depends cache architecture and size, but also the access time for hit, miss penalty, hit rate.
What about Stores?

We want to write to the cache.

If the data is not in the cache?
  Bring it in.  (Write allocate policy)

Should we also update memory?
  • Yes: write-through policy
  • No: write-back policy
Write-Through Cache

16 byte, byte-addressed memory
4 byte, fully-associative cache: 2-byte blocks, write-allocate
4 bit addresses:
  3 bit tag, 1 bit offset

Instructions:
LB x1 ← M[ 1 ]
LB x2 ← M[ 7 ]
SB x2 → M[ 0 ]
SB x1 → M[ 5 ]
LB x2 ← M[ 10 ]
SB x1 → M[ 5 ]
SB x1 → M[ 10 ]

Register File
x0
x1
x2
x3

Cache
Misses: 0
Hits: 0
Reads: 0
 Writes: 0

Memory

0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

0
78
29
120
123
71
150
162
173
18
21
33
28
19
200
210
225
Write-Through (REF 1)

Instructions:
- LB x1 ← M[ 1 ]
- LB x2 ← M[ 7 ]
- SB x2 → M[ 0 ]
- SB x1 → M[ 5 ]
- LB x2 ← M[ 10 ]
- SB x1 → M[ 5 ]
- SB x1 → M[ 10 ]

Register File
- x0
- x1
- x2
- x3

Cache
- Misses: 0
- Hits: 0
- Reads: 0
- Writes: 0

Memory
Instructions:
LB  x1 ← M[  1  ]
LB  x2 ← M[  7  ]
SB  x2 → M[  0  ]
SB  x1 → M[  5  ]
LB  x2 ← M[ 10 ]
SB  x1 → M[  5  ]
SB  x1 → M[ 10 ]

Cache
Misses:  1
Hits:       0
Reads:  2
Writes:  0

Memory
Write-Through (REF 2)

Instructions:
LB x1 ← M[ 1 ]
LB x2 ← M[ 7 ]
SB x2 → M[ 0 ]
SB x1 → M[ 5 ]
LB x2 ← M[ 10 ]
SB x1 → M[ 5 ]
SB x1 → M[ 10 ]

Misses: 1
Hits: 0
Reads: 2
Writes: 0

Memory

Register File

Cache
Instructions:
LB x1 ← M[ 1 ] M
LB x2 ← M[ 7 ] M
SB x2 → M[ 0 ]
SB x1 → M[ 5 ]
LB x2 ← M[ 10 ]
SB x1 → M[ 5 ]
SB x1 → M[ 10 ]

Memory
0 78
1 29
2 120
3 123
4 71
5 150
6 162
7 173
8 18
9 21
10 33
11 28
12 19
13 200
14 210
15 225

Register File
x0
x1 29
x2 173
x3

Cache
Misses: 2
Hits: 0
Reads: 4
Writes: 0

Instructions:
LB  x1 ← M[ 1 ]  M
LB  x2 ← M[ 7 ]  M
SB  x2 → M[ 0 ]
SB  x1 → M[ 5 ]
LB  x2 ← M[ 10 ]
SB  x1 → M[ 5 ]
SB  x1 → M[ 10 ]

Memory
0 78
1 29
2 120
3 123
4 71
5 150
6 162
7 173
8 18
9 21
10 33
11 28
12 19
13 200
14 210
15 225

Register File
x0
x1 29
x2 173
x3
Write-Through (REF 3)

Instructions:
LB x1 ← M[1] M
LB x2 ← M[7] M
SB x2 → M[0]
SB x1 → M[5]
LB x2 ← M[10]
SB x1 → M[5]
SB x1 → M[10]

CLICKER:
(A) HIT
(B) MISS

Memory
0  78
1  29
2  120
3  123
4  71
5  150
6  162
7  173
8  18
9  21
10 33
11 28
12 19
13 200
14 210
15 225

Register File
x0  29
x1  173
x2
x3

Cache
Misses: 2
Hits: 0
Reads: 4
Writes: 0
Write-Through (REF 3)

Instructions:
LB  x1 ← M[ 1 ]  M
LB  x2 ← M[ 7 ]  M
SB  x2 → M[ 0 ]  Hit
SB  x1 → M[ 5 ]
LB  x2 ← M[ 10 ]
SB  x1 → M[ 5 ]
SB  x1 → M[ 10 ]

Memory

Register File

Cache

Misses: 2
Hits: 1
Reads: 4
 Writes: 1
Write-Through (REF 4)

Instructions:
LB  x1 ← M[ 1 ]  M
LB  x2 ← M[ 7 ]  M
SB  x2 → M[ 0 ]  Hit
SB  x1 → M[ 5 ]  M
LB  x2 ← M[ 10 ]
SB  x1 → M[ 5 ]
SB  x1 → M[ 10 ]

Memory

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>173</td>
</tr>
<tr>
<td>1</td>
<td>29</td>
</tr>
<tr>
<td>2</td>
<td>120</td>
</tr>
<tr>
<td>3</td>
<td>123</td>
</tr>
<tr>
<td>4</td>
<td>71</td>
</tr>
<tr>
<td>5</td>
<td>150</td>
</tr>
<tr>
<td>6</td>
<td>162</td>
</tr>
<tr>
<td>7</td>
<td>173</td>
</tr>
<tr>
<td>8</td>
<td>173</td>
</tr>
<tr>
<td>9</td>
<td>18</td>
</tr>
<tr>
<td>10</td>
<td>21</td>
</tr>
<tr>
<td>11</td>
<td>33</td>
</tr>
<tr>
<td>12</td>
<td>28</td>
</tr>
<tr>
<td>13</td>
<td>19</td>
</tr>
<tr>
<td>14</td>
<td>200</td>
</tr>
<tr>
<td>15</td>
<td>210</td>
</tr>
<tr>
<td>16</td>
<td>225</td>
</tr>
</tbody>
</table>

Iru V  tag  data

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 000 173</td>
</tr>
<tr>
<td>1</td>
<td>1 010 71 150</td>
</tr>
</tbody>
</table>

Register File

x0
x1  29
x2  173
x3

Cache

Misses: 2
Hits: 1
Reads: 4
Writes: 1
Write-Through (REF 4)

Instructions:
LB x1 ← M[ 1 ]  M
LB x2 ← M[ 7 ]  M
SB x2 → M[ 0 ]  Hit
SB x1 → M[ 5 ]  M
LB x2 ← M[ 10 ]
SB x1 → M[ 5 ]
SB x1 → M[ 10 ]

Memory

0 1 120 123
1 29
2 173
3 120
4 123
5 71
6 29
7 162
8 173
9 18
10 21
11 33
12 28
13 19
14 200
15 210 225

Register File

x0
x1 29
x2 173
x3

Cache

Iru V tag data

1 1 000 173
0 1 010 29
0 1 010 29

Misses: 3
Hits: 1
Reads: 6
Writes: 2
Write-Through (REF 4)

CLICKER:
(A) HIT
(B) MISS

Instructions:
LB x1 ← M[ 1 ]  M
LB x2 ← M[ 7 ]  M
SB x2 → M[ 0 ]  Hit
SB x1 → M[ 5 ]  M
LB x2 ← M[ 10 ]
SB x1 → M[ 5 ]
SB x1 → M[ 10 ]

Misses:  3
Hits:       1
Reads: 6
 Writes: 2
Write-Through (REF 5)

Instructions:
- LB x1 ← M[ 1 ]
- LB x2 ← M[ 7 ]
- SB x2 → M[ 0 ] → Hit
- SB x1 → M[ 5 ]
- LB x2 ← M[ 10 ]
- SB x1 → M[ 5 ]
- SB x1 → M[ 10 ]

Memory:

Register File:
- x0
- x1: 29
- x2: 33
- x3

Cache:

Misses: 4
Hits: 1
Reads: 8
Writes: 2
Write-Through (REF 6)

Instructions:
- LB x1 ← M[ 1 ] M
- LB x2 ← M[ 7 ] M
- SB x2 → M[ 0 ] Hit
- SB x1 → M[ 5 ] M
- LB x2 ← M[ 10 ] M
- SB x1 → M[ 5 ]
- SB x1 → M[ 10 ]

Register File
- x0
- x1
- x2
- x3

Cache
- Misses: 4
- Hits: 1
- Reads: 8
- Writes: 2

Memory
- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15

lru V tag data
- 0 1 101 33
- 1 1 010 71
- 1 1 010 29
Write-Through (REF 6)

Instructions:
LB  x1 ← M[ 1 ]  M
LB  x2 ← M[ 7 ]  M
SB  x2 → M[ 0 ]  Hit
SB  x1 → M[ 5 ]  M
LB  x2 ← M[ 10 ]  M
SB  x1 → M[ 5 ]  Hit
SB  x1 → M[ 10 ]

Memory

Instructions:  
LB  x1 ← M[ 1 ]  M
LB  x2 ← M[ 7 ]  M
SB  x2 → M[ 0 ]  Hit
SB  x1 → M[ 5 ]  M
LB  x2 ← M[ 10 ]  M
SB  x1 → M[ 5 ]  Hit
SB  x1 → M[ 10 ]  

Cache

Misses:  4
Hits:       2
Reads: 8
Writes: 3

Register File

x0
x1  29
x2
x3
Write-Through (REF 7)

Instructions:
LB  x1 ← M[ 1 ]  M
LB  x2 ← M[ 7 ]  M
SB  x2 → M[ 0 ]  Hit
SB  x1 → M[ 5 ]  M
LB  x2 ← M[ 10 ]  M
SB  x1 → M[ 5 ]  Hit
SB  x1 → M[ 10 ]

Memory

Hit

Register File

Cache

Misses: 4
Hits: 2
Reads: 8
Writes: 3
Write-Through (REF 7)

Instructions:
- LB x1 ← M[ 1 ]
- LB x2 ← M[ 7 ]
- SB x2 → M[ 0 ]
- SB x1 → M[ 5 ]
- LB x2 ← M[ 10 ]
- SB x1 → M[ 5 ]
- SB x1 → M[ 10 ]

Cache:
- Misses: 4
- Hits: 3
- Reads: 8
- Writes: 3
Summary: Write Through

Write-through policy with write allocate

• Cache miss: read entire block from memory
• Write: write only updated item to memory
• Eviction: no need to write to memory
Next Goal: Write-Through vs. Write-Back

What if we DON’T to write stores immediately to memory?

Keep the current copy in cache, and update memory when data is evicted (write-back policy)
Write-back all evicted lines?
No, only written-to blocks
Write-Back Meta-Data (Valid, Dirty Bits)

<table>
<thead>
<tr>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Byte 1</th>
<th>Byte 2</th>
<th>...</th>
<th>Byte N</th>
</tr>
</thead>
</table>

- V = 1 means the line has valid data
- D = 1 means the bytes are newer than main memory
- When allocating line:
  - Set V = 1, D = 0, fill in Tag and Data
- When writing line:
  - Set D = 1
- When evicting line:
  - If D = 0: just set V = 0
  - If D = 1: write-back Data, then set D = 0, V = 0
Write-back Example

• Example: How does a write-back cache work?
• Assume write-allocate
Handling Stores (Write-Back)

16 byte, byte-addressed memory
4 byte, fully-associative cache:
2-byte blocks, write-allocate
4 bit addresses:
3 bit tag, 1 bit offset

Instructions:
LB x1 ← M[ 1 ]
LB x2 ← M[ 7 ]
SB x2 → M[ 0 ]
SB x1 → M[ 5 ]
LB x2 ← M[ 10 ]
SB x1 → M[ 5 ]
SB x1 → M[ 10 ]

Memory

Misses: 0
Hits: 0
Reads: 0
Writes: 0

16 byte, byte-addressed memory
4 byte, fully-associative cache:
2-byte blocks, write-allocate
4 bit addresses:
3 bit tag, 1 bit offset
Write-Back (REF 1)

Instructions:

- LB x1 ← M[ 1 ]
- LB x2 ← M[ 7 ]
- SB x2 → M[ 0 ]
- SB x1 → M[ 5 ]
- LB x2 ← M[ 10 ]
- SB x1 → M[ 5 ]
- SB x1 → M[ 10 ]

Cache

- Misses: 0
- Hits: 0
- Reads: 0
- Writes: 0

Register File

Memory

- 0: 78
- 1: 29
- 2: 120
- 3: 123
- 4: 71
- 5: 150
- 6: 162
- 7: 173
- 8: 18
- 9: 21
- 10: 33
- 11: 28
- 12: 19
- 13: 200
- 14: 210
- 15: 225
Write-Back (REF 1)

Instructions:
- LB x1 ← M[1]
- LB x2 ← M[7]
- SB x2 → M[0]
- SB x1 → M[5]
- LB x2 ← M[10]
- SB x1 → M[5]
- SB x1 → M[10]

Cache

Misses: 1
Hits: 0
Reads: 2
Writes: 0

Register File:
- x0
- x1: 29
- x2
- x3

Memory:
- 0: 78
- 1: 29
- 2: 120
- 3: 123
- 4: 71
- 5: 150
- 6: 162
- 7: 173
- 8: 18
- 9: 21
- 10: 33
- 11: 28
- 12: 19
- 13: 200
- 14: 210
- 15: 225
Write-Back (REF 2)

Instructions:
LB x1 ← M[ 1 ]
LB x2 ← M[ 7 ]
SB x2 → M[ 0 ]
SB x1 → M[ 5 ]
LB x2 ← M[ 10 ]
SB x1 → M[ 5 ]
SB x1 → M[ 10 ]

Memory

0 78
1 29
2 120
3 123
4 71
5 150
6 162
7 173
8 18
9 21
10 33
11 28
12 19
13 200
14 210
15 225

Cache

Misses: 1
Hits: 0
Reads: 2
Writes: 0

Register File

x0
x1 29
x2
x3

Iru  V  d  tag  data
0 1 0 000 78
1 0 0 00 29

Write-Back (REF 2)
Write-Back (REF 2)

Instructions:
LB  x1 ← M[ 1 ]
LB  x2 ← M[ 7 ]
SB  x2 → M[ 0 ]
SB  x1 → M[ 5 ]
LB  x2 ← M[ 10 ]
SB  x1 → M[ 5 ]
SB  x1 → M[ 10 ]

CacheRegister File

Memory

Register File

Cache

Misses: 2
Hits: 0
Reads: 4
Writes: 0

Iru  V  d  tag  data
1  1  0  000  78
1  1  0  011  162
0  1  0  011  162
0  1  0  000  29

Write-Back (REF 2)
Write-Back (REF 3)

Instructions:
LB x1 ← M[ 1 ]
LB x2 ← M[ 7 ]
SB x2 → M[ 0 ]
SB x1 → M[ 5 ]
LB x2 ← M[ 10 ]
SB x1 → M[ 5 ]
SB x1 → M[ 10 ]

CacheRegister File

Memory

Register File

<table>
<thead>
<tr>
<th>x0</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>29</td>
<td>173</td>
<td></td>
</tr>
</tbody>
</table>

Cache

<table>
<thead>
<tr>
<th>Iru</th>
<th>V</th>
<th>d</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>78</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>011</td>
<td>162</td>
</tr>
</tbody>
</table>

Misses: 2
Hits: 0
Reads: 4
Writes: 0
Write-Back (REF 3)

Instructions:
LB  x1 ← M[ 1 ]
LB  x2 ← M[ 7 ]
SB  x2 → M[ 0 ] Hit
SB  x1 → M[ 5 ]
LB  x2 ← M[ 10 ]
SB  x1 → M[ 5 ]
SB  x1 → M[ 10 ]

Cache

Register File

Memory

Misses: 2
Hits: 1
Reads: 4
Writes: 1
Write-Back (REF 4)

Instructions:
LB  x1 ← M[ 1 ]  M
LB  x2 ← M[ 7 ]  M
SB  x2 → M[ 0 ]  Hit
SB  x1 → M[ 5 ]
LB  x2 ← M[ 10 ]
SB  x1 → M[ 5 ]
SB  x1 → M[ 10 ]

Memory

Cache

Register File

Misses: 2
Hits: 1
Reads: 4
 Writes: 0
Write-Back (REF 4)

Instructions:
LB x1 ← M[ 1 ]
LB x2 ← M[ 7 ]
SB x2 → M[ 0 ] Hit
SB x1 → M[ 5 ]
LB x2 ← M[ 10 ]
SB x1 → M[ 5 ]
SB x1 → M[ 10 ]

CacheRegister File

Register File

x0
x1 29
x2 173
x3

Memory

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>78</td>
<td>29</td>
<td>120</td>
<td>123</td>
<td>71</td>
<td>150</td>
<td>162</td>
<td>173</td>
<td>18</td>
<td>21</td>
<td>33</td>
<td>28</td>
<td>19</td>
<td>200</td>
<td>210</td>
</tr>
</tbody>
</table>

Misses: 3
Hits: 1
Reads: 6
Writes: 0
Write-Back (REF 4)

Instructions:
LB  x1 ← M[ 1 ]  M
LB  x2 ← M[ 7 ]  M
SB  x2 → M[ 0 ] Hit
SB  x1 → M[ 5 ]  M
LB  x2 ← M[10]  M
SB  x1 → M[ 5 ]
SB  x1 → M[10]

Cache

Register File

Memory

Misses: 3
Hits: 1
Reads: 6
Writes: 0

Write-Back (REF 4)
Write-Back (REF 5)

Instructions:
LB  x1 ← M[ 1 ] M
LB  x2 ← M[ 7 ] M
SB  x2 → M[ 0 ] Hit
SB  x1 → M[ 5 ] M
LB  x2 ← M[ 10 ]
SB  x1 → M[ 5 ]
SB  x1 → M[ 10 ]

Cache

Misses: 3
Hits: 1
Reads: 6
Writes: 0

Register File

Memory

x0
x1  29
x2  173
x3

Iru  V  d  tag  data
1  1  1  000  173
0  1  1  010  71  29
Write-Back (REF 5)

Eviction, WB dirty block

Instructions:
LB x1 ← M[ 1 ] M
LB x2 ← M[ 7 ] M
SB x2 → M[ 0 ] Hit
SB x1 → M[ 5 ] M
LB x2 ← M[ 10 ]
SB x1 → M[ 5 ]
SB x1 → M[ 10 ]

Misses: 3
Hits: 1
Reads: 6
Writes: 2
Instructions:

LB x1 ← M[ 1 ]
LB x2 ← M[ 7 ]
SB x2 → M[ 0 ]
SB x1 → M[ 5 ]
LB x2 ← M[ 10 ]
SB x1 → M[ 5 ]
SB x1 → M[ 10 ]

Misses: 4
Hits: 1
Reads: 8
Writes: 2
Instructions:
LB x1 ← M[ 1 ]
LB x2 ← M[ 7 ]
SB x2 → M[ 0 ]
SB x1 → M[ 5 ]
LB x2 ← M[ 10 ]
SB x1 → M[ 5 ]
SB x1 → M[ 10 ]

(CLICKER: (A) HIT (B) MISS)

Cache
Misses: 4
Hits: 1
Reads: 8
Writes: 2

Register File
x0 29
x1 33
x2
x3

Memory
0 78
1 29
2 120
3 123
4 71
5 150
6 162
7 173
8 18
9 21
10 33
11 28
12 19
13 200
14 210
15 225
Write-Back (REF 6)

Instructions:
LB x1 ← M[1] M
LB x2 ← M[7] M
SB x2 → M[0] Hit
SB x1 → M[5] M
LB x2 ← M[10] M
SB x1 → M[5] Hit
SB x1 → M[10]

Register File
x0
x1 29
x2
x3

Cache
Misses: 4
Hits: 2
Reads: 8
Writes: 2

Memory
0 78
1 29
2 120
3 123
4 71
5 150
6 162
7 173
8 18
9 21
10 33
11 28
12 19
13 200
14 210
15 225
### Write-Back (REF 7)

#### Instructions:
- LB $x1 \leftarrow M[1]$  
- LB $x2 \leftarrow M[7]$  
- SB $x2 \rightarrow M[0]$  
- SB $x1 \rightarrow M[5]$  
- LB $x2 \leftarrow M[10]$  
- SB $x1 \rightarrow M[5]$  
- SB $x1 \rightarrow M[10]$  

#### Cache

<table>
<thead>
<tr>
<th>Iru</th>
<th>V</th>
<th>d</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>101</td>
<td>33</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>010</td>
<td>71</td>
</tr>
</tbody>
</table>

- Misses: 4
- Hits: 2
- Reads: 8
- Writes: 2
- Memory:
  - 0: 78
  - 1: 29
  - 2: 120
  - 3: 123
  - 4: 71
  - 5: 150
  - 6: 162
  - 7: 173
  - 8: 18
  - 9: 21
  - 10: 33
  - 11: 28
  - 12: 19
  - 13: 200
  - 14: 210
  - 15: 225
Write-Back (REF 7)

Instructions:
LB  x1  ←     M[ 1 ]  M
LB  x2  ←     M[ 7 ]  M
SB  x2  →     M[ 0 ]  Hit
SB  x1  →     M[ 5 ]  M
LB  x2  ←     M[ 10 ]  M
SB  x1  →     M[ 5 ]  Hit
SB  x1  →     M[ 10 ]  Hit

Cache Register File

x0
x1
x2
x3

Misses: 4
Hits: 3
Reads: 8
Writes: 2

Cache

Memory

0 78
1 29
2 120
3 123
4 71
5 150
6 162
7 173
8 18
9 21
10 33
11 28
12 19
13 200
14 210
15 225
Write-Back (REF 8,9)

Cheap subsequent updates!

Memory

<table>
<thead>
<tr>
<th>Iru</th>
<th>V</th>
<th>d</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>101</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>010</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>29</td>
<td></td>
</tr>
</tbody>
</table>

Register File

- x0
- x1 29
- x2 33
- x3

Cache

- Misses: 4
- Hits: 3
- Reads: 8
- Writes: 2

Instructions:

... 
SB $1 \rightarrow M[5]$
LB $2 \leftarrow M[10]$
SB $1 \rightarrow M[5]$
SB $1 \rightarrow M[10]$
SB $1 \rightarrow M[5]$
SB $1 \rightarrow M[10]$
Write-Back (REF 8,9)

Instructions:
...
SB $1 \to M[5]\hspace{1cm} M
LB $2 \leftarrow M[10]\hspace{1cm} M
SB $1 \to M[5]\hspace{1cm} Hit
SB $1 \to M[10]\hspace{1cm} Hit
SB $1 \to M[5]\hspace{1cm} Hit
SB $1 \to M[10]\hspace{1cm} Hit

Cache

Misses: 4
Hits: 3
Reads: 8
Writes: 2

Memory

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

0
29
120
123
71
150
162
173
18
21
33
28
19
200
210
225

Register File

x0
x1
x2
x3
How Many Memory References?

Write-back performance

• How many reads?
  • Each miss (read or write) reads a block from mem
  • 4 misses $\rightarrow$ 8 mem reads

• How many writes?
  • *Some* evictions write a block to mem
  • 1 dirty eviction $\rightarrow$ 2 mem writes
  • (+ 2 dirty evictions later $\rightarrow$ +4 mem writes)
Write-back vs. Write-through Example

Assume: large associative cache, 16-byte lines
N 4-byte words

for (i=1; i<n; i++)
    A[0] += A[i];

for (i=0; i<n; i++)
    B[i] = A[i]

Write-thru: n reads (n/4 cache lines)
            n writes
Write-back: n reads (n/4 cache lines)
            4 writes (one cache line)

Write-thru: 2 x n reads (2 x n/4 cache lines)
            n writes
Write-back: 2 x n reads (2 x n/4 cache lines)
            n writes (n/4 cache lines)
So is write back just better?

Short Answer: Yes (fewer writes is a good thing)

Long Answer: It’s complicated.

• Evictions require entire line be written back to memory (vs. just the data that was written)
• Write-back can lead to incoherent caches on multi-core processors (later lecture)
Optimization: Write Buffering

• Q: Writes to main memory are slow!
• A: Use a write-back buffer
  • A small queue holding dirty lines
  • Add to end upon eviction
  • Remove from front upon completion
• Q: When does it help?
  • A: short bursts of writes (but not sustained writes)
  • A: fast eviction reduces miss penalty
Write-through vs. Write-back

- Write-through is slower
  - But simpler (memory always consistent)

- Write-back is almost always faster
  - write-back buffer hides large eviction cost
  - But what about multiple cores with separate caches but sharing memory?

- **Write-back requires a cache coherency protocol**
  - Inconsistent views of memory
  - Need to “snoop” in each other’s caches
  - Extremely complex protocols, very hard to get right
Cache-coherency

Q: Multiple readers and writers?

A: Potentially inconsistent views of memory

Cache coherency protocol

- May need to **snoop** on other CPU’s cache activity
- **Invalidate** cache line when other CPU writes
- **Flush** write-back caches before other CPU reads
- Or the reverse: Before writing/reading…
- Extremely complex protocols, very hard to get right
Takeaway

• Write-through policy with write allocate
  • Cache miss: read entire block from memory
  • Write: write only updated item to memory
  • Eviction: no need to write to memory
  • Slower, but cleaner

• Write-back policy with write allocate
  • Cache miss: read entire block from memory
  - **But may need to write dirty cacheline first**
  • Write: nothing to memory
  • Eviction: have to write to memory *entire cacheline*
    because don’t know what is dirty (only 1 dirty bit)
  • Faster, but more complicated, especially with multicore
Cache Conscious Programming

// H = 6, W = 10
int A[H][W];
for(x=0; x < W; x++)
    for(y=0; y < H; y++)
        sum += A[y][x];
Cache Conscious Programming

// H = 6, W = 10
int A[H][W];
for(x=0; x < W; x++)
    for(y=0; y < H; y++)
        sum += A[y][x];

Every access a cache miss!
(unless *entire* matrix fits in cache)
Cache Conscious Programming

```c
// H = 6, W = 10
int A[H][W];
for(x=0; x < H; x++)
    for(y=0; y < W; y++)
        sum += A[x][y];
```

- Block size = 4 → 75% hit rate
- Block size = 8 → 87.5% hit rate
- Block size = 16 → 93.75% hit rate
- And you can easily prefetch to warm the cache
Clicker Question

Choose the best block size for your cache among the choices given. Assume that integers and pointers are all 4 bytes each and that the scores array is 4-byte aligned.

(a) 1 byte  (b) 4 bytes  (c) 8 bytes  (d) 16 bytes  (e) 32 bytes

```c
int scores[NUM STUDENTS] = 0;
int sum = 0;
for (i = 0; i < NUM STUDENTS; i++) {
    sum += scores[i];
}
```
Clicker Question

Choose the best block size for your cache among the choices given. Assume that integers and pointers are all 4 bytes each and that the scores array is 4-byte aligned.

(a) 1 byte (b) 4 bytes (c) 8 bytes (d) 16 bytes (e) 32 bytes

```c
int scores[NUM STUDENTS] = 0;
int sum = 0;
for (i = 0; i < NUM STUDENTS; i++) {
    sum += scores[i];
}
```
Clicker Question

Choose the best block size for your cache among the choices given. Assume integers and pointers are 4 bytes.

(a) 1 byte (b) 4 bytes (c) 8 bytes (d) 16 bytes (e) 32 bytes

```c
typedef struct item_t {
    int value;
    struct item_t *next;
    char *name;
} item_t;

int sum = 0;
item_t *curr = list_head;
while (curr != NULL) {
    sum += curr->value;
    curr = curr->next;
}
```
Clicker Question
Choose the best block size for your cache among the choices given. Assume integers and pointers are 4 bytes.
(a) 1 byte (b) 4 bytes (c) 8 bytes (d) 16 bytes (e) 32 bytes

```c
typedef struct item_t {
    int value;
    struct item_t *next;
    char *name;
} item_t;

int sum = 0;
item_t *curr = list_head;
while (curr != NULL) {
    sum += curr->value;
    curr = curr->next;
}
```
A Real Example

- Dual 32K L1 Instruction caches
  - 8-way set associative
  - 64 sets
  - 64 byte line size
- Dual 32K L1 Data caches
  - Same as above
- Single 6M L2 Unified cache
  - 24-way set associative (!!!)
  - 4096 sets
  - 64 byte line size
- 4GB Main memory
- 1TB Disk

Dual-core 3.16GHz Intel
Summary

• **Memory performance matters!**
  • often more than CPU performance
  • … because it is the bottleneck, and not improving much
  • … because most programs move a LOT of data

• **Design space is huge**
  • Gambling against program behavior
  • Cuts across all layers:
    users → programs → os → hardware

• **NEXT: Multi-core processors are complicated**
  • Inconsistent views of memory
  • Extremely complex protocols, very hard to get right