I'm sorry, we can't approve this permit. Your land isn't zoned for giant-money-bin construction.

Also, you're a duck.

https://what-if.xkcd.com/111/
Making memory accesses fast!

- Cache basics
- Principle of locality
- Memory hierarchies
- Cache organization
  - Direct-mapped (sets; index + tag)
  - Associativity (ways)
  - Replacement policy
  - Handling writes
- Program optimizations that consider caches
Associativity

- What if we could store data in any place in the cache?
  - More complicated hardware = more power consumed, slower
- So we combine the two ideas:
  - Each address maps to exactly one set
  - Each set can store block in more than one way

<table>
<thead>
<tr>
<th>1-way: 8 sets, 1 block each</th>
<th>2-way: 4 sets, 2 blocks each</th>
<th>4-way: 2 sets, 4 blocks each</th>
<th>8-way: 1 set, 8 blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="#">Diagram</a></td>
<td><a href="#">Diagram</a></td>
<td><a href="#">Diagram</a></td>
<td><a href="#">Diagram</a></td>
</tr>
</tbody>
</table>

- direct mapped
- fully associative
Cache Organization (3)

- **Associativity** ($E$): # of ways for each set
  - Such a cache is called an “$E$-way set associative cache”
  - We now index into cache sets, of which there are $C/K/E$
  - Use lowest $\log_2(C/K/E) = s$ bits of block address
    - Direct-mapped: $E = 1$, so $s = \log_2(C/K)$ as we saw previously
    - Fully associative: $E = C/K$, so $s = 0$ bits

---

Note: The textbook uses “b” for offset bits

<table>
<thead>
<tr>
<th>Used for tag comparison</th>
<th>Selects the set</th>
<th>Selects the byte from block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag ($t$)</td>
<td>Index ($s$)</td>
<td>Offset ($k$)</td>
</tr>
</tbody>
</table>

Decreasing associativity: Direct mapped (only one way)  
Increasing associativity: Fully associative (only one set)
Example Placement

- Where would data from address 0x1833 be placed?
  - Binary: \(0b \ 0001 \ 1000 \ 0011 \ 0011\)

\[ t = m - s - k \]
\[ s = \log_2(C/K/E) \]
\[ k = \log_2(K) \]

\( m \)-bit address:

<table>
<thead>
<tr>
<th>Tag (t)</th>
<th>Index (s)</th>
<th>Offset (k)</th>
</tr>
</thead>
</table>

- Set Tag Data
  - Direct-mapped
    - Set 0
      - Tag
      - Data
    - Set 1
      - Tag
      - Data
    - Set 2
      - Tag
      - Data
    - Set 3
      - Tag
      - Data
    - Set 4
      - Tag
      - Data
    - Set 5
      - Tag
      - Data
    - Set 6
      - Tag
      - Data
    - Set 7
      - Tag
      - Data
  - 2-way set associative
    - Set 0
      - Tag
      - Data
    - Set 1
      - Tag
      - Data
    - Set 2
      - Tag
      - Data
    - Set 3
      - Tag
      - Data
  - 4-way set associative
    - Set 0
      - Tag
      - Data
    - Set 1
      - Tag
      - Data
    - Set 0
      - Tag
      - Data
# Block Replacement

- Any empty block in the correct set may be used to store block.
- If there are no empty blocks, which one should we replace?
  - No choice for direct-mapped caches.
  - Caches typically use something close to **least recently used (LRU)** (hardware usually implements “*not most recently used*”)
Peer Instruction Question

- We have a cache of size 2 KiB with block size of 128 B. If our cache has 2 sets, what is its associativity?
  A. 2
  B. 4
  C. 8
  D. 16
  E. We’re lost...

- If addresses are 16 bits wide, how wide is the Tag field?
General Cache Organization \((S, E, K)\)

- \(E = \) blocks/lines per set
- \(S = \) # sets = \(2^s\)
- \(K = \) bytes per block

**Cache size:**
\[ C = K \times E \times S \] data bytes (doesn’t include V or Tag)
# Notation Review

- We just introduced a lot of new variable names!
  - Please be mindful of block size notation when you look at past exam questions or are watching videos

<table>
<thead>
<tr>
<th>Variable</th>
<th>This Quarter</th>
<th>Formulas</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block size</td>
<td>$K$ (in book)</td>
<td>$M = 2^m \leftrightarrow m = \log_2 M$</td>
</tr>
<tr>
<td>Cache size</td>
<td>$C$</td>
<td>$S = 2^s \leftrightarrow s = \log_2 S$</td>
</tr>
<tr>
<td>Associativity</td>
<td>$E$</td>
<td>$K = 2^k \leftrightarrow k = \log_2 K$</td>
</tr>
<tr>
<td>Number of Sets</td>
<td>$S$</td>
<td>$C = K \times E \times S$</td>
</tr>
<tr>
<td>Address space</td>
<td>$M$</td>
<td>$s = \log_2(C/K/E)$</td>
</tr>
<tr>
<td>Address width</td>
<td>$m$</td>
<td>$m = t + s + k$</td>
</tr>
<tr>
<td>Tag field width</td>
<td>$t$</td>
<td></td>
</tr>
<tr>
<td>Index field width</td>
<td>$s$</td>
<td></td>
</tr>
<tr>
<td>Offset field width</td>
<td>$k$ (in book)</td>
<td></td>
</tr>
</tbody>
</table>
Cache Read

1) Locate set
2) Check if any line in set is valid and has matching tag: hit
3) Locate data starting at offset

$S = \# \text{ sets} = 2^s$

$E = \text{blocks/lines per set}$

Address of byte in memory:

- $t$ bits
- $s$ bits
- $k$ bits

- tag
- set index
- block offset

Data begins at this offset

valid bit

$K = \text{bytes per block}$
Example: Direct-Mapped Cache ($E = 1$)

Direct-mapped: One line per set
Block Size $K = 8$ B

$$S = 2^s \text{ sets}$$

Address of `int`:

| bits | 0...01 | 100 |

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

Direct-mapped: One line per set
Block Size \(K = 8\) B

Address of \texttt{int}:

```
0...01 100
```

Valid? + match?: yes = hit

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

Direct-mapped: One line per set
Block Size $K = 8$ B

Int (4 B) is here

No match? Then old line gets evicted and replaced

This is why we want alignment!
Example: Set-Associative Cache ($E = 2$)

2-way: Two lines per set
Block Size $K = 8$ B

Address of `short int`:

```
| bits | 0...01 | 100 |
```

find set
Example: Set-Associative Cache ($E = 2$)

2-way: Two lines per set
Block Size $K = 8$ B

Compare both

Valid? + Match: yes = hit

Block offset

Address of `short int`

bits

0...01

100
Example: Set-Associative Cache ($E = 2$)

2-way: Two lines per set
Block Size $K = 8$ B

Valid? + Match: yes = hit

Address of `short int`:

No match?
- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), ...
Types of Cache Misses: 3 C’s!

- **Compulsory (cold) 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, ... could miss every time
  - Direct-mapped caches have more conflict misses than $E$-way set-associative (where $E > 1$)

- **Capacity miss**
  - Occurs when the set of active cache blocks (the *working set*) is larger than the cache (just won’t fit, even if cache was *fully-associative*)
  - **Note:** *Fully-associative* only has Compulsory and Capacity misses
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 next level
  - **Write-back**: defer write to next level until line is evicted (replaced)
    - Must track which cache lines have been modified ("dirty bit")

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

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

In this example we are sort of ignoring block offsets. Here a block holds 2 bytes (16 bits, 4 hex digits).

Normally a block would be much bigger and thus there would be multiple items per block. While only one item in that block would be written at a time, the entire line would be brought into cache.
Write-back, write-allocate example

```
mov 0xFACE, F
```

Cache

Memory

```
G  0xBEEF  0
F  0xCAF
G  0xBEEF
```
Write-back, write-allocate example

mov 0xCACE, F

Step 1: Bring F into cache
Write-back, write-allocate example

\texttt{mov 0xFACE, F}

\begin{itemize}
  \item \textbf{Cache}:
    \begin{itemize}
      \item F: 0xFACE
      \item Dirty bit: 1
    \end{itemize}
    \begin{itemize}
      \item Dirty bit: \textcolor{red}{dirty bit}
    \end{itemize}

  \item \textbf{Memory}:
    \begin{itemize}
      \item F: 0xCAFE
      \item G: 0xBEEF
    \end{itemize}
\end{itemize}

\textbf{Step 2: Write 0xFACE to cache only and set dirty bit}
Write-back, write-allocate example

mov 0xFACE, F
mov 0xFEED, F

Write hit!
Write 0xFEED to cache only

Dirty bit
Write-back, write-allocate example

```
mov 0xFACE, F  
mov 0xFEED, F  
mov G, %rax
```

Cache

```
F 0xFEED 1
```

dirty bit

Memory

```
F 0xCAFE
G 0xBEEF
```
Write-back, write-allocate example

mov 0xFACE, F  
mov 0xFEED, F  
mov G, %rax

1. Write F back to memory since it is dirty
2. Bring G into the cache so we can copy it into %rax
Peer Instruction Question

Which of the following cache statements is FALSE?

A. We can reduce compulsory misses by decreasing our block size
B. We can reduce conflict misses by increasing associativity
C. A write-back cache will save time for code with good temporal locality on writes
D. A write-through cache will always match data with the memory hierarchy level below it
E. We’re lost...
Example Cache Parameters Problem

- 1 MiB address space, 125 cycles to go to memory.
  Fill in the following table:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Size</td>
<td>4 KiB</td>
</tr>
<tr>
<td>Block Size</td>
<td>16 B</td>
</tr>
<tr>
<td>Associativity</td>
<td>4-way</td>
</tr>
<tr>
<td>Hit Time</td>
<td>3 cycles</td>
</tr>
<tr>
<td>Miss Rate</td>
<td>20%</td>
</tr>
<tr>
<td>Write Policy</td>
<td>Write-through</td>
</tr>
<tr>
<td>Replacement Policy</td>
<td>LRU</td>
</tr>
<tr>
<td>Tag Bits</td>
<td>10</td>
</tr>
<tr>
<td>Index Bits</td>
<td>6</td>
</tr>
<tr>
<td>Offset Bits</td>
<td>4</td>
</tr>
<tr>
<td>AMAT</td>
<td>AMAT = 3 + 0.2 * 125 = 28</td>
</tr>
</tbody>
</table>
Example Code Analysis Problem

- Assuming the cache starts **cold** (all blocks invalid), calculate the **miss rate** for the following loop:
  - \( m = 20 \) bits, \( C = 4 \) KiB, \( K = 16 \) B, \( E = 4 \)

```c
#define AR_SIZE 2048
int int_ar[AR_SIZE], sum=0; // &int_ar=0x80000
for (int i=0; i<AR_SIZE; i++)
    sum += int_ar[i];
for (int j=AR_SIZE-1; j>=0; j--)
    sum += int_ar[i];
```
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
Where else is caching used?

- Caching is one of the biggest ideas in CSE, far beyond the hardware memory hierarchy

- Use a faster set of a subset of the data to exploit temporal and spatial locality
  - At the end of Summer, grab a few jackets from the bedroom closet and leave them near the door
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