<table>
<thead>
<tr>
<th></th>
<th>Lecture Overview</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Review of Caches</td>
</tr>
<tr>
<td>2.</td>
<td>Example Problems</td>
</tr>
</tbody>
</table>
Generic Cache Organization

- \( B = 2^b \) bytes per cache block
- \( E \) lines per set
- \( S = 2^s \) sets

Set 0:
- Valid
- Tag
- 0 1 \( \ldots \) \( B-1 \)

Set 1:
- Valid
- Tag
- 0 1 \( \ldots \) \( B-1 \)

Set \( S-1 \):
- Valid
- Tag
- 0 1 \( \ldots \) \( B-1 \)

Cache size: \( C = B \times E \times S \) data bytes
Looking up a memory address

- Set 0:
  - Valid
  - Tag
  - Cache block
- Set 1:
  - Valid
  - Tag
  - Cache block
- 
- Fully Associative
Cache Lookup

Three steps while determining whether a request is a hit or a miss:

- **Set selection**: Select the set where the address resides.
- **Line matching**: Select the cache line within the set.
- **Word extraction**: Extract the requested word from the right offset.
Lookup algorithm:
(cache receives address)
[use index to identify frame]

if frame is valid
  if frame's tag matches address' tag
    HIT
  else
    MISS
else
  MISS

Valid

<table>
<thead>
<tr>
<th>tag</th>
<th>data blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Types of Misses

- **Compulsory or cold misses**: Cache is empty to start with and will miss.

- **Conflict misses**: Cache has space but because objects map to the same cache block they keep missing.

- **Capacity misses**: Cache does not have space because size of the working set exceeds the size of the cache.
Cache Replacement Policies

- Which block to replace or evict to make space for new blocks?
  - **Random Replacement Policy**: chooses a random victim block.
  - **Least Recently Used (LRU) Policy**: chooses the block that was last accessed furthest in the past.
  - **Least Frequently Used (LFU) Policy**: chooses the block that was least frequently accessed in the past.
Implementing writes

1. write through
   change data in the cache, and send the write to main memory

   slow 😞, but very little circuitry 😊
2 write back

- at first, change data in the cache
- write to memory only when necessary

dirty bit is set on a write, to identify blocks to be written back to memory

when a program completes, all dirty blocks must be written to memory...
2. write back (continued)

- faster 😊
  multiple stores to the same location result in only 1 main memory access
- more circuitry 😞
  - must maintain the dirty bit
  - *dirty miss*: a miss caused by a read or write to a block not in the cache, but the required block frame has its dirty bit set. So, there is a write of the dirty block, followed by a read of the requested block.
Writing during cache miss: (Two approaches)

- **Write Alloc**: Load block in cache and update word (often used along with Write back)
- **Write No-Alloc (a.k.a. Write around)**: Just update memory (often used along with Write through)
We can send memory accesses to the 2 caches independently...

😊 (increased parallelism)
Suppose we have a system with the following properties:

- The memory is byte addressable.
- Memory accesses are to 1-byte words (not to 4-byte words).
- Addresses are 12 bits wide.
- The cache is two-way set associative ($E = 2$), with a 4-byte block size ($B = 4$) and four sets ($S = 4$).

The contents of the cache are as follows, with all addresses, tags, and values given in hexadecimal notation:

<table>
<thead>
<tr>
<th>Set index</th>
<th>Tag</th>
<th>Valid</th>
<th>Byte 0</th>
<th>Byte 1</th>
<th>Byte 2</th>
<th>Byte 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>1</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>83</td>
<td>1</td>
<td>FE</td>
<td>97</td>
<td>CC</td>
<td>D0</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>83</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>00</td>
<td>1</td>
<td>48</td>
<td>49</td>
<td>4A</td>
<td>4B</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>FF</td>
<td>1</td>
<td>9A</td>
<td>C0</td>
<td>03</td>
<td>FF</td>
</tr>
<tr>
<td></td>
<td>00</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A. The following diagram shows the format of an address (one bit per box). Indicate (by labeling the diagram) the fields that would be used to determine the following:

- \( CO \)  The cache block offset
- \( CI \)  The cache set index
- \( CT \)  The cache tag

```
11 10 9 8 7 6 5 4 3 2 1 0
```

B. For each of the following memory accesses indicate if it will be a cache hit or miss when carried out in sequence as listed. Also give the value of a read if it can be inferred from the information in the cache.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address</th>
<th>Hit?</th>
<th>Read value (or unknown)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>0x834</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td>0x836</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read</td>
<td>0xFFD</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Suppose we have a system with the following properties:

- The memory is byte addressable.
- Memory accesses are to 1-byte words (not to 4-byte words).
- Addresses are 13 bits wide.
- The cache is four-way set associative \((E = 4)\), with a 4-byte block size \((B = 4)\) and eight sets \((S = 8)\).

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>V</th>
<th>Bytes 0–3</th>
<th>Index</th>
<th>Tag</th>
<th>V</th>
<th>Bytes 0–3</th>
<th>Index</th>
<th>Tag</th>
<th>V</th>
<th>Bytes 0–3</th>
<th>Index</th>
<th>Tag</th>
<th>V</th>
<th>Bytes 0–3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F0</td>
<td>1</td>
<td>ED 32 0A A2</td>
<td>8</td>
<td>8A</td>
<td>1</td>
<td>BF 80 1D FC</td>
<td>14</td>
<td>14</td>
<td>E</td>
<td>09 86 2A</td>
<td>BC</td>
<td>0</td>
<td>25 44 6F 1A</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>BC</td>
<td>0</td>
<td>03 3E CD 38</td>
<td>9</td>
<td>A0</td>
<td>0</td>
<td>16 7B ED 5A</td>
<td>1E</td>
<td>1E</td>
<td>B</td>
<td>4E 4C DF 18</td>
<td>E4</td>
<td>1</td>
<td>FB B7 12 02</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>BC</td>
<td>1</td>
<td>54 9E 1E FA</td>
<td>10</td>
<td>B6</td>
<td>1</td>
<td>DC 81 B2 14</td>
<td>00</td>
<td>0</td>
<td>C</td>
<td>B6 1F 7B 44</td>
<td>74</td>
<td>0</td>
<td>10 F5 B8 2E</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>BE</td>
<td>0</td>
<td>2E 7E 3D A8</td>
<td>11</td>
<td>C0</td>
<td>1</td>
<td>27 95 A4 74</td>
<td>00</td>
<td>0</td>
<td>C</td>
<td>07 11 6B D8</td>
<td>BC</td>
<td>0</td>
<td>C7 B7 AF C2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>7E</td>
<td>1</td>
<td>32 21 1C 2C</td>
<td>12</td>
<td>8A</td>
<td>1</td>
<td>22 C2 DC 34</td>
<td>00</td>
<td>0</td>
<td>C</td>
<td>BA DD 37 D8</td>
<td>DC</td>
<td>0</td>
<td>E7 A2 39 BA</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>98</td>
<td>0</td>
<td>A9 76 2B EE</td>
<td>13</td>
<td>54</td>
<td>0</td>
<td>BC 91 D5 92</td>
<td>98</td>
<td>1</td>
<td>C</td>
<td>80 BA 9B F6</td>
<td>BC</td>
<td>1</td>
<td>48 16 81 0A</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>38</td>
<td>0</td>
<td>5D 4D F7 DA</td>
<td>14</td>
<td>8C</td>
<td>1</td>
<td>69 C2 8C 74</td>
<td>8A</td>
<td>1</td>
<td>C</td>
<td>A8 CE 7F DA</td>
<td>38</td>
<td>1</td>
<td>FA 93 EB 48</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8A</td>
<td>1</td>
<td>04 2A 32 6A</td>
<td>15</td>
<td>9E</td>
<td>0</td>
<td>B1 86 56 0E</td>
<td>CC</td>
<td>1</td>
<td>C</td>
<td>96 30 47 F2</td>
<td>BC</td>
<td>1</td>
<td>F8 1D 42 30</td>
<td></td>
</tr>
</tbody>
</table>

**Analyze memory references 0x071A and 0x16E8**
Strided Access Patterns

```c
int i, j, sum = 0;
for (i=0; i<16; i++)
    for (j=0; j<16; j++)
        sum += a[i][j]

What if: sum += a[j][i] ?
```
Memory Mountain from the CSAPP textbook