In the following diagram, each blank box in the CPU Cache represents 8 bits (1 byte) of data. Our memory is byte-addressed, meaning that there is one address for each byte. Compare this to word-addressed, which means that there is one address for each word.

<table>
<thead>
<tr>
<th>CPU Cache</th>
<th>Index Number</th>
<th>Offset</th>
<th>Tag bits</th>
<th>Index bits</th>
<th>Offset bits</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3 2 1 0</td>
<td></td>
<td>29</td>
<td>1</td>
<td>2</td>
<td>32</td>
</tr>
</tbody>
</table>

Index bits = \( \log_2 \) (Number of index rows)  
Offset bits = \( \log_2 \) (Number of offsets columns)

1. Direct mapped caches
   1. How many bytes of data can our cache hold? **8 bytes**  
      How many words? **2 words**
   2. Fill in the “Tag bits, Index bits, Offset bits” with the correct T:I:O breakdown according to the diagram.
   3. Let’s say we have a 8192KiB cache with an 128B block size, what is the tag, index, and offset of 0xFEEDF00D?

<table>
<thead>
<tr>
<th>Address</th>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>FE ED F0 OD</td>
<td>1111 1110 1110 1101 1111 0000 0000 1101</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

   Tag: **11111101 (0x1FD)**  
   Index: **110110111100000 (0xDBE0)**  
   Offset: **0001101 (0x0D)**

4. Fill in the table below. Assume we have a write-through cache, so the number of bits per row includes only the cache data, the tag, and the valid bit.

<table>
<thead>
<tr>
<th>Address size (bits)</th>
<th>Cache size</th>
<th>Block size</th>
<th>Tag bits</th>
<th>Index bits</th>
<th>Offset bits</th>
<th>Bits per row</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>4KiB</td>
<td>4B</td>
<td>4</td>
<td>10</td>
<td>2</td>
<td>32+4+1</td>
</tr>
<tr>
<td>32</td>
<td>32KiB</td>
<td>16B</td>
<td>17</td>
<td>11</td>
<td>4</td>
<td>128+17+1</td>
</tr>
<tr>
<td>32</td>
<td>64KiB</td>
<td>16B</td>
<td>16</td>
<td>12</td>
<td>4</td>
<td>128+16+1</td>
</tr>
<tr>
<td>64</td>
<td>2048KiB</td>
<td>128B</td>
<td>43</td>
<td>14</td>
<td>7</td>
<td>1068</td>
</tr>
</tbody>
</table>

2. Cache hits and misses

Assume we have the following byte-addressed cache. Of the 32 bits in each address, which bits do we use to find the row of the cache to use? **We use the 4th and 5th least significant bit since the offset is 3 bits**

Classify each of the following byte memory accesses as a cache hit (H), cache miss (M), or cache miss with replacement (R).
### 3. 3C’s of Caches

3 types of cache misses:

1. **Compulsory**: Miss to an address not seen before. Reduce compulsory misses by having a longer cache line, which brings in locations before we ask for them.

2. **Conflict**: Increasing the associativity or improving the replacement policy would remove the miss.

3. **Capacity**: The only way to remove the miss is to increase the cache capacity.

Classify each M and R above as one of the 3 misses above.

### 4. Analyzing C Code

```c
#define NUM_INTS 8192
int A[NUM_INTS]; /* A lives at 0x10000 */
int i, total = 0;
for (i = 0; i < NUM_INTS; i += 128) { A[i] = i; } /* Line 1 */
for (i = 0; i < NUM_INTS; i += 128) { total += A[i]; } /* Line 2 */
```

Let’s say you have a byte-addressed computer with a total memory of 1MiB. It features a 16KiB CPU cache with 1KiB blocks.

1. How many bits make up a memory address on this computer? **20**

2. What is the T:I:O breakdown? tag bits: **6** index bits: **4** offset bits: **10**

3. Calculate the cache hit rate for the line marked Line 1: **50%**

   The integer accesses are 4*128=512 bytes apart, which means there are 2 accesses per block. The first accesses in each block is a cache miss, but the second is a hit because A[i] and A[i+128] are in the same cache block.

4. Calculate the cache hit rate for the line marked Line 2: **50%**

   The size of A is 8192*4 = 2¹³ bytes. This is exactly twice the size of our cache. At the end of line 1, we have the second half of A inside the cache, while in line 2 we start accesses from the beginning of the array. Thus we cannot reuse any of the content of A and we get the same hit rate as before. Note that we do not have to consider cache hits for total, since the compiler will probably leave it in a register.
5. Average Memory Access Time

AMAT is the average (expected) time it takes for memory access. It can be calculated using the formula:

\[ \text{AMAT} = \text{hit\_time} + \text{miss\_rate} \times \text{miss\_penalty} \]

Remember that the miss penalty is the additional time it takes for memory access in the event of a cache miss. Therefore, a cache miss takes \((\text{hit\_time} + \text{miss\_penalty})\) time.

1. Suppose that you have a cache system with the following properties. What is the AMAT?
   a) L1$ hits in 1 cycle (local miss rate 25%)
   b) L2$ hits in 10 cycles (local miss rate 40%)
   c) L3$ hits in 50 cycles (global miss rate 6%)
   d) Main memory hits in 100 cycles (always hits)

The AMAT is \(1 + 0.25*(10 + 0.4*(50)) + 0.06*100 = 14.5\) cycles.

Alternatively, we can calculate the global hit rates for each hierarchy:

- L1$: 0.75
- L2$: 0.25*0.6 = 0.15
- L3$: 0.94 – (0.75+0.15) = 0.04
- Main Memory: 1 – 0.75 – 0.15 – 0.04 = 0.06

And the following hit times:

- L1$: 1 cycle
- L2$: 1+10 = 11 cycles
- L3$: 1 + 10 + 50 = 61 cycles
- Main Memory: 1 + 10 + 50 + 100 = 161 cycles

Then, AMAT = 0.75*1 + 0.15*11 + 0.04*61 + 0.06 * 161 = 14.5 cycles.