Virtual Memory Example

(1) We are first asked to label the different parts of a virtual address. This is not something that we ask you to do for the hell of it. You will need to use this information for the later parts of the question.

(2) The whole idea behind caches is that we come up with imaginary borders in memory. We cut it up into cache *lines*. When we access a memory location, we load the entire line that contains it into the cache. If our programs are written with good spatial locality, then we will get hits on further memory accesses.

(3) Part 1: Each byte in memory has its own address. Given some virtual address, we should be able to narrow down that exact byte that we want. The CO (cache offset) bits are used to index into a specific line once we have loaded it. We know that each cache line is 4 bytes long, so we need 2 bits to index them.

(4) Each line goes into its own set. This figure shows 8 sets, so we need 3 bits to index them. We now know the number of bits taken up by each part of the virtual address: `TTTT TTTT III OO`

Part 2:

- Physical Address: `0x0d74` → `0000 1101 0111 0100`

  `0x0d74` → `0000 1101 0111 0100`→ CT of `0110 1011`, CI of `101`, CO of `00`

  `0x0d74` → `0000 1101 0111 0100`→ CT of `0x6b`, CI of `0x5`, CO of `0x0`

  → valid bit is OFF, the data is dirty.

- Physical Address: `0x0aee` → `0000 1010 1110 1110`

  `0x0aee` → `0000 1010 1110 1110`→ CT of `0101 0111`, CI of `011`, CO of `10`

  `0x0aee` → `0000 1010 1110 1110`→ CT of `0x57`, CI of `0x3`, CO of `0x2`

  → valid bit is ON, data is valid. The byte stored at this memory address is `0xFF`.

(5) Part 3: Set 7 can hold up to 4 valid lines, but here we only see 2. We can use the tag bits to get the address ranges:

One valid tag is (`0x84`) : `0x84` → `1000 0100 111 00 - 1000 0100 111 11` → `0x109c-0x109f`.

One valid tag is (`0x91`) : `0x91` → `1001 0001 111 00 - 1001 0001 111 11` → `0x123c-0x123f`.

(6) Part 4: `0x1080-0x109F` → `1 0000 1000 0000 - 1 0000 1001 1111`: set indices go from 0 to 7. offsets go from 0 to 3. Tag stays fixed as `0x1000 0100` → `0x84`. We can look for this tag and see how many sets contain it with valid data.

We see the tag `0x84` in 4 out of our 8 sets: $4/8 = 50$ percent

Example 1: Pixel Scan

A bitmap image is composed of pixels. Each pixel in the image is represented as four values: three for the primary colors(red, green and blue - RGB) and one for the transparency information defined as an alpha channel. In this problem, you will compare the performance of direct mapped and 4-way associative caches for a square bitmap image initialization. Both caches have a size of 128 bytes. The direct-mapped cache has 8-byte blocks while the 4-way associative cache has 4-byte blocks. You are given the definitions:

```
typedef struct {
   unsigned char r;
   unsigned char g;
   unsigned char b;
   unsigned char a;
} pixel_t;

pixel_t pixel[16][16];
register int i, j;
```

Also assume that

- `sizeof(unsigned char) == 1`
- `pixel` begins at memory address 0
- Both caches are initially empty
- The array is stored in row-major order
- Variables `i`,`j` are stored in registers and any access to these variables does not cause a cache miss

1. What fraction of the writes in the following code will result in misses?

```
for (i = 0; i < 16; i ++){
   for (j = 0; j < 16; j ++){
      pixel[i][j].r = 0;
      pixel[i][j].g = 0;
      pixel[i][j].b = 0;
      pixel[i][j].a = 0;
   }
}
```

Miss rate for writes to pixel in a direct mapped cache: 12.5%

The terms line and block are used interchangeably.

Each cache line is 8 bytes, so it stores 2 pixels. We perform a stride-1 (sequential scan) through the matrix, performing one write to each of the `r`, `g`, `b`, and `a` components of each pixel. The important thing to note here is that when we write to the `r` component of the first pixel in a line, we get a cache miss. However, writes to the `g`, `b`, and `a` components are hits. Furthermore, since we load two pixels with each cache line, we have already loaded the next pixel. That means the `r`, `g`, `b`, and `a` accesses will all be cache hits.

For example, consider what happens when we access `pixel[0][0].r`. We get a cache miss and load `pixel[0][0]` and `pixel[0][1]` into the cache. We will get cache hits when we try to access `pixel[0][0].g`, `pixel[0][0].b`, `pixel[0][0].a`, `pixel[0][1].r`, `pixel[0][1].g`, `pixel[0][1].b`, or `pixel[0][1].a`.

We perform 8 writes with only 1 miss. Our miss rate is 1/8 = 12.5 percent.

Miss rate for writes to pixel in a 4-way associative cache: 25%

The fact that the cache is 4-way associative does not affect us since we are executing a sequential scan access pattern. The block size does affect us, though. Since it is only 4 bytes, we load only one pixel into our cache on each miss (as opposed to the two we were able to load with 8 byte lines). This means that we will get a cache miss when accessing the `r` component of a pixel. We will get hits accessing the `g`, `b`, and `a` components.

For example, consider what happens when we access `pixel[0][0].r`. We get a cache miss and load `pixel[0][0]` into the cache. We will get cache hits when we try to access `pixel[0][0].g`, `pixel[0][0].b`, or `pixel[0][0].a`.

We perform 4 writes with only 1 miss. Our miss rate is $1/4 = 25$ percent.

2. What fraction of the writes in the following code will result in a miss in the direct mapped cache?

```
for (i = 0; i < 16; i ++){
   for (j = 0; j < 16; j ++){
      pixel[j][i].r = 0;
      pixel[j][i].g = 0;
      pixel[j][i].b = 0;
      pixel[j][i].a = 0;
   }
}
```

Miss rate for writes to pixel: 25%

This time, we perform a column-wise access. We are still able to load two pixels (a full 8 bytes) into the cache on a miss. The problem is that the second pixel will be evicted before we every access it. For example, consider what happens when we access the `r` component of `pixel[0][0]`. We get a cache miss and load `pixel[0][0]` and `pixel[0][1]` into the cache. The cache itself though, is only 128 bytes long. This means that the second set of 128 bytes will get mapped over the first set of 128 bytes in the cache. 128 bytes is 32 pixels and, in the case of this matrix, two rows. This means that when we access `pixel[2][0]`, we will load `pixel[2][0]` and `pixel[2][1]` into the cache, evicting the line containing `pixel[0][0]` and `pixel[0][1]`. By the time we finish with column 0 and move up to access `pixel[0][1]`, it has already been evicted from the cache.

The writes to `pixel[0][0].g`, `pixel[0][0].b`, and `pixel[0][0].a` are cache hits. We get 4 writes per 1 cache miss. Our miss rate is $1/4 = 25$ percent.

3. Using code in part C, what fraction of the writes will result in a miss in the 4-way associative cache?

Miss rate for writes to pixel: 25%

Once again, the block size determines the answer to this question. We miss trying to access `pixel[0][0].r`, but then `pixel[0][0]` is loaded into the cache. The writes to `pixel[0][0].g`, `pixel[0][0].b`, and `pixel[0][0].a` are cache hits. We never touch this pixel again, and we never loaded any other pixels with this cache line. We get 4 writes per 1 cache miss. Our miss rate is $1/4 = 25$ percent.