**Question F7: Virtual Memory**  [10 pts]

Our system has the following setup:

- 24-bit virtual addresses and 512 KiB of RAM with 4 KiB pages
- A 4-entry TLB that is fully associative with LRU replacement
- A page table entry contains a valid bit and protection bits for read (R), write (W), execute (X)

(A) Compute the following values: [2 pt]

<table>
<thead>
<tr>
<th>Page offset width</th>
<th>PPN width</th>
<th>Entries in a page table</th>
<th>TLBT width</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>12</strong></td>
<td><strong>7</strong></td>
<td><strong>2^12</strong></td>
<td><strong>12</strong></td>
</tr>
</tbody>
</table>

Because TLB is fully associative, TLBT width matches VPN. There are $2^{\text{VPN width}}$ entries in PT.

(B) Briefly explain why we make the page size so much larger than a cache block size. [2 pt]

Take advantage of spatial locality and try to avoid page faults as much as possible.
Disk access is also super slow, so we want to pull a lot of data when we do access it.

(C) Fill in the following blanks with “A” for always, “S” for sometimes, and “N” for never if the following get updated during a page fault. [2 pt]

<table>
<thead>
<tr>
<th>Page table</th>
<th>Swap space</th>
<th>TLB</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A</strong></td>
<td><strong>S</strong></td>
<td><strong>A/N</strong></td>
<td><strong>S</strong></td>
</tr>
</tbody>
</table>

When the page is place in physical memory, the new PPN is written into the page table entry.
Swap space will get updated if a dirty page is kicked out of physical memory.

For this class, we say that the page fault handler updates the TLB because it is more efficient.
In reality not all do (OS does not have access to hardware-only TLB; instr gets restarted).
To update a PTE (in physical mem), you check the cache, so it gets updated on a cache miss.

(D) The TLB is in the state shown when the following code is executed. Which iteration (value of $i$) will cause the protection fault (segfault)? Assume sum is stored in a register.

Recall: the hex representations for TLBT/PPN are padded as necessary. [4 pt]

```c
long *p = 0x7F0000, sum = 0;
for (int i = 0; 1; i++) {
    if (i%2)
        *p = 0;
    else
        sum += *p;
    p++;
}
```

<table>
<thead>
<tr>
<th>TLBT</th>
<th>PPN</th>
<th>Valid</th>
<th>R</th>
<th>W</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7F0</td>
<td>0x31</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0x7F2</td>
<td>0x15</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0x004</td>
<td>0x1D</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0x7F1</td>
<td>0x2D</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Only the current page (VPN = TLBT = 0x7F0) has write access. Once we hit the next page (TLBT = 0x7F1), we will encounter a segfault once we try to write to the page. We are using pointer arithmetic to increment our pointer by 8 bytes at a time. One page holds $2^{12}/2^3 = 512$ longs, so we first access TLBT 0x7F1 when $i = 512$. However, the code is set up so that we only write on odd values of $i$, so the answer is $i = 513$. 