1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 If a page table entry can not be found in the TLB, then a page fault has occurred.

False, TLB acts as a cache for the page table, so an item can be valid in page table but not stored in TLB. Page fault occurs either when a page cannot be found in the page table or it has an invalid bit.

1.2 The local miss rate of one level of a cache is always greater than or equal to the global miss rate of that cache.

True, local miss rate cannot be lower than global miss rate. Even in the case that all other cache levels have 100% miss rate the local miss rate will be equal to the global miss rate.

1.3 SIMD is a form of instruction-level parallelism.

False, instruction-level parallelism deals with performing multiple instructions in parallel, i.e. pipelining. SIMD is a form of data parallelism with a single instruction performing operation on multiple streams of data.

2 VM Access Patterns

2.1 A processor has 16-bit addresses, 256 byte pages, and an 8-entry fully associative TLB with LRU replacement (the LRU field is 3 bits and encodes the order in which pages were accessed, 0 being the most recent). At some time instant, the TLB for the current process is the initial state given in the table below. Assume that all current page table entries are in the initial TLB. Assume also that all pages can be read from and written to. Fill in the final state of the TLB according to the access pattern below.
Free Physical Pages 0x17, 0x18, 0x19

Access Pattern

1. 0x11f0 (Read) 4. 0x2332 (Write)
2. 0x1301 (Write) 5. 0x20ff (Read)
3. 0x20ae (Write) 6. 0x3415 (Write)

Initial TLB

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
<th>Dirty</th>
<th>LRU</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>0x11</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0x00</td>
<td>0x00</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>0x10</td>
<td>0x13</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0x20</td>
<td>0x12</td>
<td>1</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>0x00</td>
<td>0x00</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>0x11</td>
<td>0x14</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>0xac</td>
<td>0x15</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0xff</td>
<td>0xff</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

Final TLB

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
<th>Dirty</th>
<th>LRU</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>0x11</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>0x13</td>
<td>0x17</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0x10</td>
<td>0x13</td>
<td>1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>0x20</td>
<td>0x12</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0x23</td>
<td>0x18</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0x11</td>
<td>0x14</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>0xac</td>
<td>0x15</td>
<td>1</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>0x34</td>
<td>0x19</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

1. 0x11f0 (Read): hit, LRUs: 1, 7, 2, 5, 7, 0, 3, 4
2. 0x1301 (Write): miss, map VPN 0x13 to PPN 0x17, valid and dirty, LRUs: 2, 0, 3, 6, 7, 1, 4, 5
3. 0x20ae (Write): hit, dirty, LRUs: 3, 1, 4, 0, 7, 2, 5, 6
4. 0x2332 (Write): miss, map VPN 0x23 to PPN 0x18, valid and dirty, LRUs: 4, 2, 5, 1, 0, 3, 6, 7
5. 0x20ff (Read): hit, LRUs: 4, 2, 5, 0, 1, 3, 6, 7
6. 0x3415 (Write): miss and replace last entry, map VPN 0x34 to 0x19, dirty, LRUs, 5, 3, 6, 1, 2, 4, 7, 0
3 AMAT

Recall that AMAT stands for Average Memory Access Time. The main formula for it is:

\[ \text{AMAT} = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty} \]

In a multi-level cache, there are two types of miss rates that we consider for each level. **Global**: Calculated as the number of accesses that missed at that level divided by the total number of accesses to the cache system. **Local**: Calculated as the number of accesses that missed at that level divided by the total number of accesses to that cache level.

### 3.1 An L2$, out of 100 total accesses to the cache system, missed 20 times. What is the global miss rate of L2$?

\[
\frac{20}{100} = 20\%
\]

### 3.2 If L1$ had a miss rate of 50%, what is the local miss rate of L2$?

\[
\frac{20}{50\% + 100} = \frac{20}{60\%} = 40\%. \text{ We know that L2$ is accessed when L1$ misses, so if L1$ misses 50\% of the time, that means we access L2$ 50 times.}
\]

Suppose your system consists of:

1. An L1$ that has a hit time of 2 cycles and has a local miss rate of 20%
2. An L2$ that has a hit time of 15 cycles and has a global miss rate of 5%
3. Main memory where accesses take 100 cycles

### 3.3 What is the local miss rate of L2$?

\[
\text{L2$ Local miss rate} = \frac{\text{Global Miss Rate}}{\text{L1$ Miss Rate}} = \frac{5\%}{20\%} = 0.25 = 25\%
\]

### 3.4 What is the AMAT of the system?

\[
\text{AMAT} = 2 + 20\% \times 15 + 5\% \times 100 = 10 \text{ cycles (using global miss rates)}
\]

Alternatively, \( \text{AMAT} = 2 + 20\% \times (15 + 25\% \times 100) = 10 \text{ cycles} \)

### 3.5 Suppose we want to reduce the AMAT of the system to 8 cycles or lower by adding in a L3$ . If the L3$ has a local miss rate of 30\%, what is the largest hit time that the L3$ can have?

Let \( H = \text{hit time of the cache. Using the AMAT equation, we can write:}

\[
2 + 20\% \times (15 + 25\% \times (H + 30\% \times 100)) \leq 8
\]

Solving for \( H \), we find that \( H \leq 30 \). So the largest hit time is 30 cycles.

4 Flynn’s Taxonomy

### 4.1 Explain SISD and give an example if available.

**Single Instruction Single Data**: each instruction is executed in order, acting on a single stream of data. For example, traditional computer programs.

### 4.2 Explain SIMD and give an example if available.
Single Instruction Multiple Data; each instruction is executed in order, acting on multiple streams of data. For example, the SSE Intrinsics.

**4.3 Explain MISD and give an example if available.**

Multiple Instruction Single Data; multiple instructions are executed simultaneously, acting on a single stream of data. There are no good modern examples.

**4.4 Explain MIMD and give an example if available.**

Multiple Instruction Multiple Data; multiple instructions are executed simultaneously, acting on multiple streams of data. For example, map reduce or multithreaded programs.

## 5 Data-Level Parallelism

The idea central to data level parallelism is vectorized calculation: applying operations to multiple items (which are part of a single vector) at the same time.

Some machines with x86 architectures have special, wider registers, that can hold 128, 256, or even 512 bits. Intel intrinsics (Intel proprietary technology) allow us to use these wider registers to harness the power of DLP in C code.

Below is a small selection of the available Intel intrinsic instructions. All of them perform operations using 128-bit registers. The type \_\_m128i is used when these registers hold 4 ints, 8 shorts or 16 chars; \_\_m128d is used for 2 double precision floats, and \_\_m128 is used for 4 single precision floats. Where you see “epiXX”, epi stands for extended packed integer, and XX is the number of bits in the integer. “epi32” for example indicates that we are treating the 128-bit register as a pack of 4 32-bit integers.

- **\_\_m128i \_mm_set1_epi32(int i):**
  Set the four signed 32-bit integers within the vector to i.

- **\_\_m128i \_mm_loadu_si128( \_\_m128i *p):**
  Load the 4 successive ints pointed to by p into a 128-bit vector.

- **\_\_m128i \_mm_mullo_epi32(\_\_m128i a, \_\_m128i b):**
  Return vector \((a_0 \cdot b_0, a_1 \cdot b_1, a_2 \cdot b_2, a_3 \cdot b_3)\).

- **\_\_m128i \_mm_add_epi32(\_\_m128i a, \_\_m128i b):**
  Return vector \((a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)\).

- **\textbf{void} \_mm_storeu_si128( \_\_m128i *p, \_\_m128i a):**
  Store 128-bit vector a at pointer p.

- **\_\_m128i \_mm_and_si128(\_\_m128i a, \_\_m128i b):**
  Perform a bitwise AND of 128 bits in a and b, and return the result.
• \texttt{\_m128i \_mm\_cmpeq\_epi32(\_m128i a, \_m128i b)}:

The \textit{i}th element of the return vector will be set to \texttt{0xffffffff} if the \textit{i}th elements of a and b are equal, otherwise it'll be set to 0.

\textbf{Notice:} On this worksheet, we are using the \textit{unaligned} versions of the commands that interface with memory (i.e. \texttt{storeu/loadu} vs. \texttt{store/load}). This is because the \texttt{store/load} commands require that the address we are loading at is aligned at some byte boundary (and not necessarily just word-aligned), whereas the unaligned versions have no such requirements. For instance, \texttt{\_mm\_store\_si128} needs the address to be aligned on a 16-byte boundary (i.e. is a multiple of 16). There is extra work that needs to be done to achieve these alignment requirements, so for this class, we just use the unaligned variants.

5.1 You have an array of 32-bit integers and a 128-bit vector as follows:

\begin{verbatim}
int arr[8] = {1, 2, 3, 4, 5, 6, 7, 8};
_mm128i vector = _mm_loadu_si128((__m128i *) arr);
\end{verbatim}

For each of the following tasks, fill in the correct arguments for each SIMD instruction, and where necessary, fill in the appropriate SIMD function. Assume they happen independently, i.e. the results of Part (a) do not at all affect Part (b).

(a) Multiply \texttt{vector} by itself, and set \texttt{vector} to the result.

\begin{verbatim}
vector = _mm_mullo_epi32(vector, vector);
\end{verbatim}

(b) Add 1 to each of the first 4 elements of the \texttt{arr}, resulting in \texttt{arr} = {2, 3, 4, 5, 5, 6, 7, 8}

\begin{verbatim}
__m128i vector_ones = _mm_set1_epi32(1);
__m128i result = _mm_add_epi32(vector, vector_ones);
_mm_storeu_si128((__m128i *) arr, result);
\end{verbatim}

(c) Add the second half of the array to the first half of the array, resulting in \texttt{arr} = {1 + 5, 2 + 6, 3 + 7, 4 + 8, 5, 6, 7, 8} = {6, 8, 10, 12, 5, 6, 7, 8}

\begin{verbatim}
__m128i result = _mm_add_epi32(_mm_loadu_si128((__m128i *) (arr + 4)), vector);
_mm_storeu_si128((m128i*) arr, result);
\end{verbatim}

(d) Set every element of the array that is not equal to 5 to 0, resulting in \texttt{arr} = {0, 0, 0, 0, 5, 0, 0, 0}. Remember that the first half of the array has already been loaded into \texttt{vector}.

\begin{verbatim}
__m128i fives = _mm_set1_epi32(5);
__m128i mask = _mm_cmpeq_epi32(vector, fives);
__m128i result = _mm_and_si128(mask, vector);
_mm_storeu_si128((__m128i *) arr, result);
\end{verbatim}
vector = _mm_loadu_si128((__m128i *) (arr + 4));
mask = _mm_cmpeq_epi32(vector, fives);
result = _mm_and_si128(mask, vector);
_mm_storeu_si128((__m128i *) (arr + 4), result);

SIMD-ize the following function, which returns the product of all of the elements in an array. Things to think about: When iterating through a loop and grabbing elements 4 at a time, how should we update our index for the next iteration? What if our array has a length that isn’t a multiple of 4? Can we always SIMD-ize an entire array? What can we do to handle this tail case?

```c
static int product_naive(int n, int *a) {
    int product = 1;
    for (int i = 0; i < n; i++) {
        product *= a[i];
    }
    return product;
}
```

```c
static int product_vectorized(int n, int *a) {
    int result[4];
    __m128i prod_v = __mm_set1_epi32(1);
    for (int i = 0; i < n/4 * 4; i += 4) { // Vectorized loop
        prod_v = __mm_mullo_epi32(prod_v, __mm_loadu_si128((__m128i *) (a + i)));
    }
    _mm_storeu_si128((__m128i *) result, prod_v);
    for (int i = n/4 * 4; i < n; i++) { // Handle tail case
        result[0] *= a[i];
    }
    return result[0] * result[1] * result[2] * result[3];
}
```