Translation Lookaside Buffer

Virtual Memory would not be very effective if every memory address had to be translated by looking up the associated physical page in memory. The solution is to cache the recent translations in a Translation Lookaside Buffer (TLB).

Addressing The Cache

Since the program is generating virtual addresses, and the memory uses physical addresses, there are two solutions to the problem of addressing the cache:

- Physically addressed cache: Translate virtual address before cache reference
- Virtually addressed cache: Reference cache directly and translate only on a cache miss, which is when physical memory must be referenced

It’s a no brainer, right?
Translation Lookaside Buffer

The TLB is a small cache of the most recent virtual-physical mappings. By checking here first, temporal locality is exploited to speed virtual address translation.

<table>
<thead>
<tr>
<th>Block size</th>
<th>1-2 page-table entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hit Time</td>
<td>1/2-1 clock cycle</td>
</tr>
<tr>
<td>Miss penalty</td>
<td>10-30 clock cycles</td>
</tr>
<tr>
<td>Miss rate</td>
<td>0.01%-1%</td>
</tr>
<tr>
<td>Size</td>
<td>32-1024 entries</td>
</tr>
</tbody>
</table>

The strange name comes from the idea that, while the process to perform a virtual-to-physical translation is underway, the hardware checks to see if it has seen this translation problem recently.
Logic of Memory Reference for 3100

Virtual address → TLB access

- TLB miss interrupt
  - No → TLB Hit?
    - Yes → No → Write Access Bit=1
      - No → Cache Hit?
        - Yes → Deliver data to CPU
          - No → Cache miss stall
            - Yes → Write data into cache, update the tag, and put the data and the address into the write buffer.
      - Yes → Write ?
        - Yes → Write Protect Exception
          - No → Try to read data from cache
            - No → TLB Hit?
              - Yes → No → Write Access Bit=1
                - No → Cache Hit?
                  - Yes → Deliver data to CPU
                    - No → Write data into cache, update the tag, and put the data and the address into the write buffer.
  - Yes → No → Write ?
    - Yes → Write Protect Exception
      - No → Try to read data from cache
        - No → TLB Hit?
          - Yes → No → Write Access Bit=1
            - No → Cache Hit?
              - Yes → Deliver data to CPU
                - No → Write data into cache, update the tag, and put the data and the address into the write buffer.
  - Yes → No → Write ?
    - Yes → Write Protect Exception
      - No → Try to read data from cache
        - No → TLB Hit?
          - Yes → No → Write Access Bit=1
            - No → Cache Hit?
              - Yes → Deliver data to CPU
                - No → Write data into cache, update the tag, and put the data and the address into the write buffer.
TLB Miss Means Either ...

- The page is present ==> only a TLB entry must be created
- The page is not present (i.e. page table entry for the virtual address has 0 valid bit), a page fault exception is signaled
  - The exception flushed the instruction, put the PC in the exception program counter (EPC) and interrupted the processor.
  - The operating system, checking the cause, discovers a page fault was signaled, and knowing this is a time consuming operation, saves the state: GP and FP registers, Page Table Address, EPC & Cause.
  - What address is needed:
    - Instruction Page Fault, find address in EPC.
    - Data Page Fault, compute address from Inst.
  - OS then:
    - Finds disk address in page table entry.
    - Chooses victim to replace; writes back if dirty bit set.
    - Initiates read of disk block.

Protection Through Virtual Memory

A multi-user environment requires protection
Virtual address spaces are logically separate as long as they never reference the same physical page
Operating System sets page tables
  Two execution modes: user/supervisor
  Page table address must be supervisor readable
Sharing can be assisted with "write protection" or read/write bits
Context switching can be assisted when there is a TLB by extending the tag field of TLB entry with a process ID ... matches require both the address and the ID to match

TLB

<table>
<thead>
<tr>
<th>Tag</th>
<th>PhysPage</th>
<th>Proc ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>0001</td>
<td>id 001</td>
</tr>
<tr>
<td>1010</td>
<td>0020</td>
<td>id 020</td>
</tr>
<tr>
<td>1000</td>
<td>0101</td>
<td>id 991</td>
</tr>
<tr>
<td>0011</td>
<td>1001</td>
<td>id 104</td>
</tr>
</tbody>
</table>