uMPS processor architecture

- The uMPS architecture

uMPS memory management

- uMPS physical memory address format
  - Physical Frame Number and Offset

```
PFN            Offset
  31 ........  12 11 ........  0
```
**uMPS memory management**

- **uMPS physical memory map**
  - Kernel and User modes

![Physical Memory Map Diagram](image1)

- **ROM and Device Registers Area**

![ROM Area Diagram](image2)

- **ROM reserved frame (first RAM frame)**

![ROM Frame Diagram](image3)

- **uMPS virtual memory address format**

  - Segment Number, Virtual Page Number and Offset
  - ASID (Address Space Identifier): 0..63 (0 for Kernel)

![Virtual Memory Address Format](image4)
**uMPS memory management**

- **Mapping virtual memory to physical memory**

<table>
<thead>
<tr>
<th>SEG NO</th>
<th>VPN</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 29</td>
<td>......</td>
<td>1211</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PFN</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>......</td>
</tr>
</tbody>
</table>

---

**uMPS memory management**

- **uMPS virtual memory map**

**Kernel and User modes**

<table>
<thead>
<tr>
<th>SEGNO</th>
<th>Start address</th>
<th>End address</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0x00000000</td>
<td>0x00000000</td>
</tr>
<tr>
<td>01</td>
<td>0x00000000</td>
<td>0x00000000</td>
</tr>
<tr>
<td>10</td>
<td>0x00000000</td>
<td>0x00000000</td>
</tr>
<tr>
<td>11</td>
<td>0x00000000</td>
<td>0x00000000</td>
</tr>
<tr>
<td>12</td>
<td>0x00000000</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

---

**uMPS memory management**

- **uMPS virtual memory segment description**

<table>
<thead>
<tr>
<th>ksegOS (SEGNO 00 and 01: 2 GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>is protected by User mode access:</td>
</tr>
<tr>
<td>completely (2 GB) when virtual memory is on (that is, when Status.VMc = 1)</td>
</tr>
<tr>
<td>partially (0.5 GB) when virtual memory is off (that is, when Status.VMc = 0)</td>
</tr>
<tr>
<td>holds ROM, device registers (the first 0.5 GB) and will hold kernel text, data, stack areas (the remaining 1.5 GB)</td>
</tr>
</tbody>
</table>

---

**uMPS memory management**

- **uMPS virtual memory segment description (cont’d)**

<table>
<thead>
<tr>
<th>kUseg2 (SEGNO 10: 1 GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>may be accessed in Kernel mode and User mode</td>
</tr>
<tr>
<td>will hold user mode process text, data, stack areas</td>
</tr>
<tr>
<td>user processes will be identified using ASIDs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>kUseg3 (SEGNO 11: 1 GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>may be accessed in Kernel mode and User mode</td>
</tr>
<tr>
<td>is typically used for data sharing among processes</td>
</tr>
<tr>
<td>here, too, user processes will be identified using ASIDs</td>
</tr>
</tbody>
</table>
uMPS memory management

- **uMPS virtual memory management scheme**
  - Problem: each process running in a virtual memory space requires the management of the list of the page frames it employs, and the mapping of virtual page addresses to these page frames (Offsets are the same)
  - Solution: to use Page Tables (PgTbl), one for each segment
  - Problem 1: PgTbls could become very large (1 GB RAM = 256K pages)
  - Problem 2: Kernel and ROM handlers will need to access these tables
  - Solution: to use a Segment Table: put the PgTbl addresses into ROM reserved frame, and PgTbls themselves somewhere else

- **PgTbl format:**
  - PgTbl Entry n
  - PgTbl Entry 1
  - PgTbl Entry 2

- **PgTbl format (cont’d):**
  - MagicNO: Magic number = 0x2A
  - EntryCNT: number of entries
  - PTE (Page Table Entry): holds the virtual -> physical address translation rule for one (ASID, SEGNO, VPN) to one PFN

- **Which format to use?**
- **Who does the job of translating the addresses?**
uMPS memory management

- **Enter the TLB (Translation Lookaside Buffer):**
  - is located in the uMPS main CPU
  - performs the virtual address translation
  - caches the most recent translations
  - has a finite number of entries (TLBSIZE: 4-64 entries)
  - entry #0 is protected by accidental overwriting
  - somebody has to (re)fill it

uMPS memory management

- **PTE and TLB entry format:**

```
<table>
<thead>
<tr>
<th>SEGP</th>
<th>VPN</th>
<th>ASID</th>
<th>UNUSED</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>29</td>
<td>......</td>
<td>12</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>PFN</th>
<th>N</th>
<th>D</th>
<th>V</th>
<th>G</th>
<th>Unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>......</td>
<td>12</td>
<td>7</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>
```

uMPS memory management

- **EntryLo flags explained:**
  - N (Non-cacheable) bit: unused
  - D (Dirty) bit: trying to write to a virtual memory address with D bit = 0 raises a TLB-Modification (Mod) exception (allows to define read-only virtual memory areas and memory protection schemes)
  - V (Valid) bit: trying to read at or write to a virtual memory address with V bit = 0 raises a TLB-Invalid (TLB&TLBS) exception (allows to build memory paging schemes)
  - G (Global) bit: if G bit = 1, the TLB entry will match for any ASID with the same VPN (allows to define memory sharing schemes)

uMPS memory management

- **CP0 registers used for virtual memory addressing:**
  - BadVAddr: employed in exception management
  - EntryHi: employed for defining the ASID of the current process and in TLB refilling operations (same format as TLB EntryHi)
  - EntryLo: employed in TLB refilling operations (same format as TLB EntryLo)
  - Index: employed in TLB refilling operations
  - Random: employed in TLB refilling operations
Putting all together

A process running with virtual memory on is identified by \texttt{CP0.EntryHi.ASID}, its current status is described in \texttt{CP0.Status}.

For each access in virtual memory, the CPU:
- checks if the process has been granted the access to the \texttt{SEGNO} requested (that is, if the \texttt{KUc} bit in \texttt{CP0.Status} enables it to access);
- if access is denied, an exception is raised:
  - Address Error (\texttt{AdEL} or \texttt{AdES}) (also raised if the address is ill-formed);
- if access is granted...

Putting all together (cont'd)

(if access is granted) ... the CPU scans the TLB looking for a match for the \texttt{(ASID, SEGNO, VPN)} requested.

If a match is found and it is valid, the \texttt{PFN} is paired with the \texttt{Offset} to form the physical address, and physical memory is finally accessed (raising a \texttt{IBE} or \texttt{DBE} exception if something goes wrong).

If a match is found but it is not valid, an exception is raised:
- TLB-Modification (\texttt{Mod}) on writing with \texttt{D} bit = 0
- TLB-Invalid (\texttt{TLBL} or \texttt{TLBS}) on \texttt{V} bit = 0

But if the match is not found? A \texttt{TLB-Refill} exception is raised and the ROM TLB-refill handler kicks in...

Putting all together (final)

The ROM TLB-refill handler looks at the Segment Table, finds the \texttt{PgTbl} and scans it, looking for the first \texttt{PTE} matching the request.

If a match is found: the ROM handler refills the TLB with the \texttt{PTE} needed, and returns from the exception; the CPU starts re-executing the access to the virtual memory again, as if the exception never happened; this time, the match in the TLB will be found, and the CPU will continue as required.

If a match is not found, it could be for two reasons:
- the Segment Table or the \texttt{PgTbl} is corrupted somehow: the ROM handler "raises" a Bad-PgTbl (\texttt{BdPT}) exception.
- the \texttt{PgTbl} does not contain any match: the ROM handler "raises" a PTE-MISS (\texttt{PTMs}) exception.

The ROM strikes again

To refill the TLB, the ROM TLB-refill handler uses the following \texttt{CP0} registers:
- \texttt{EntryHi}
- \texttt{EntryLo}
- \texttt{Index}
- \texttt{Random}

And may use some special \texttt{CP0} instructions:
- \texttt{TLBWI} (TLB Write Indexed)
- \texttt{TLBWR} (TLB Write Random)
- \texttt{TLBP} (TLB Probe)
- \texttt{TLBR} (TLB Read)
- \texttt{TLBCLR} (TLB Clear)
uMPS memory management

> CP0 registers for VM management:

<table>
<thead>
<tr>
<th>SEGMD</th>
<th>VPN</th>
<th>ASID</th>
<th>Unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>29</td>
<td>......</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EntryHi</th>
</tr>
</thead>
<tbody>
<tr>
<td>PFN</td>
</tr>
<tr>
<td>N</td>
</tr>
<tr>
<td>D</td>
</tr>
<tr>
<td>V</td>
</tr>
<tr>
<td>Unused</td>
</tr>
<tr>
<td>N</td>
</tr>
<tr>
<td>D</td>
</tr>
<tr>
<td>V</td>
</tr>
<tr>
<td>Unused</td>
</tr>
</tbody>
</table>

uMPS memory management

> CP0 registers for VM management (cont’d):

- **Index:**
  - P (Probe failure) bit: becomes 0 if a TLBP is successful, 1 otherwise
  - TLB Index:
    - tells which TLB entry matches in a TLBP
    - sets which TLB entry gets read (in a TLBR) or written (in a TLBWI)
- **Random:** cycles its TLB Index field in the range [1..TLBSize-1] (one step forward each clock cycle)

uMPS memory management

> The ROM TLB-Refill algorithm, part 1

- The ROM TLB-refill handler:
  - looks up the PgTbl position in the segment table, looking at CP0.EntryHi
  - accesses the PgTbl performing some basic checks (address alignment, magic number, size vs. RAMTOP)
  - if something is not ok, then the PgTbl is not good: the handler saves the “Old” processor state, sets Cause.ExcCode in the “Old” area to indicate a Bad-PgTbl (BdPT) exception, and loads a “New” processor state to handle the exception
  - if all is ok, performs a linear scan of the PgTbl looking for a match…
The ROM TLB-Refill algorithm, part 2

- if, scanning the PgTbl, a match is found:
  - the matching PTE gets loaded (a TLBWR is performed)
  - a RFE is executed, and the processor restarts execution
- if a match is not found: the handler saves the “Old” processor state in the appropriate area, sets Cause.ExcCode in the “Old” area to indicate a PTE-MISS (PTMs) exception, and loads a “New” processor state to handle the exception

Some interesting questions:

- How does the ROM detect a TLB-Refill exception?
- Could the ROM TLB-refilling algorithm be made smarter?
- Who fills the Segment Tables and the PgTbls?
- Why the kernel should start with virtual memory off?
- Could the kernel be started with virtual memory on?
- Are other memory management schemes possible?
- If so, how to implement it?
- Why the TLB size can be made smaller or larger?
- Why TLB entry #0 is protected from TLBWR?
- Will Kayaproject phase2 require the virtual memory management?

Some interesting answers (1 of 3):

- How does the ROM detect a TLB-Refill exception?
  - Because the CPU jumps to a different address if a TLB-Refill exception is raised:
    - 0x1FC0.0100 if Status.BEV is set
    - 0x0000.0000 if Status.BEV is not set
- Could the ROM TLB-refilling algorithm be made smarter?
  - yes, by defining different specifications (e.g., using an ordered list for the PgTblds)
- Who fills the Segment Tables and the PgTblds?
  - the kernel itself (or some part of it, e.g., a specialized VM process)

Some interesting answers (2 of 3):

- Why the kernel should start with virtual memory off?
  - because it’s easier to manage the VM by starting with it off
- Could the kernel be started with virtual memory on?
  - yes (in MPS, it was the only way); in that case, the Bootstrap ROM needs to be more sophisticated
- Are other memory management schemes possible?
  - yes: they could be simpler or more complex, and they could be more or less efficient, depending on hardware/software interaction
- If other memory management schemes are possible, how to implement them?
  - by changing the ROMs
uMPS memory management

Some interesting answers (3 of 3):

- Why the TLB size can be made smaller or larger?
  - to allow testing for more frequent or infrequent TLB-Refill exceptions, that is, to allow testing the performance of different memory management schemes
- Why TLB entry #0 is protected from TLBWR?
  - To have a place where to put a “safe” TLB entry (eg. if kernel starts with VM on, such a TLB entry is quite useful)
- Will Kaya project phase2 require virtual memory management?
  - NO