uMPS processor architecture

The uMPS architecture

The MIPS processor architecture

uMPS memory management

uMPS physical memory address format

Physical Frame Number and Offset

31 ......... 1211 ......... 0
uMPS memory management

**uMPS physical memory map**

Kernel and User modes

- Self Error
- RAMTOP
- Boot Error
- RAMTOP
- Bus Error
- RAMTOP
- 0.5 GB for ROM and Device Registers
- 0x0000.0000
- 0x2000.0000
- 0x0000.2000
- 0x0000.4000
- 0x0000.8000
- 0x0000.0000
- 0.5 GB for Installed RAM
- 0x0000.0000

**uMPS virtual memory address format**

Segment Number, Virtual Page Number and Offset

ASID (Address Space IDentifier): 0..63 (0 for Kernel)

<table>
<thead>
<tr>
<th>SEG NO</th>
<th>VPN</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 29</td>
<td>. . .</td>
<td>12 11</td>
</tr>
</tbody>
</table>
uMPS memory management

Mapping virtual memory to physical memory

 SEGNO   VPN      Offset
 31  29 ... 1211 ... 0

 PFN      Offset
 31 ... 1211 ... 0

© 2007 Mauro Morsiani

uMPS memory management

uMPS virtual memory map

Kernel and User modes

kUseg2 (SEGNO 00: 2 GB)
- is protected by User mode access:
  - completely (2 GB) when virtual memory is on (that is, when `Status.VMc = 1`)
  - partially (0.5 GB) when virtual memory is off (that is, when `Status.VMc = 0`)
- holds ROM, device registers (the first 0.5 GB) and will hold kernel text, data, stack areas (the remaining 1.5 GB)

kUseg3 (SEGNO 01: 2 GB)
- may be accessed in Kernel mode and User mode
- will hold user mode process text, data, stack areas
- user processes will be identified using ASIDs

© 2007 Mauro Morsiani

uMPS memory management

uMPS virtual memory segment description

kUseg2 (SEGNO 00: 1 GB)
- may be accessed in Kernel mode and User mode
- will hold user mode process text, data, stack areas
- user processes will be identified using ASIDs

kUseg3 (SEGNO 01: 1 GB)
- may be accessed in Kernel mode and User mode
- is typically used for data sharing among processes
- here, too, user processes will be identified using ASIDs

© 2007 Mauro Morsiani
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.

**Segment Table format:**

All PgTbl addresses are physical memory addresses:

```
<table>
<thead>
<tr>
<th>ksegDS</th>
<th>kUseg2</th>
<th>kUseg3</th>
<th>0x0000,0800</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASID 63: PgTbl addr</td>
<td>PgTbl addr</td>
<td>PgTbl addr</td>
<td>PgTbl addr</td>
</tr>
<tr>
<td>ASID 62: PgTbl addr</td>
<td>PgTbl addr</td>
<td>PgTbl addr</td>
<td>PgTbl addr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x2000,0000</td>
<td>PgTbl addr</td>
<td>PgTbl addr</td>
<td>PgTbl addr</td>
</tr>
<tr>
<td>ASID 2: PgTbl addr</td>
<td>PgTbl addr</td>
<td>PgTbl addr</td>
<td>PgTbl addr</td>
</tr>
<tr>
<td>ASID 1: PgTbl addr</td>
<td>PgTbl addr</td>
<td>PgTbl addr</td>
<td>PgTbl addr</td>
</tr>
<tr>
<td>ASID 0: PgTbl addr</td>
<td>PgTbl addr</td>
<td>PgTbl addr</td>
<td>PgTbl addr</td>
</tr>
</tbody>
</table>
```

**PgTbl format:**

- **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:

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 (TLBL & 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 CP0.EntryHi.ASID; its current status is described in CP0.Status.
For each access in virtual memory, the CPU:
- checks if the process has been granted the access to the SEGNO requested (that is, if the KUc bit in CP0.Status enables it to access)
- if access is denied, an exception is raised: Address Error (AdEL or 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 (ASID, SEGNO, VPN) requested
- if a match is found and it is valid, the PFN is paired with the Offset to form the physical address, and physical memory is finally accessed (raising a IBE or DBE exception if something goes wrong)
- if a match is found but it is not valid, an exception is raised: TLB-Modification (Mod) on writing with D bit = 0
  TLB-Invalid (TLBL or TLBS) on V bit = 0
  but if the match is not found? A 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 PgTbl and scans it, looking for the first PTE matching the request
- if a match is found: the ROM handler refills the TLB with the 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 PgTbl is corrupted somehow: the ROM handler “raises” a Bad-PgTbl (BdPT) exception
  - the PgTbl does not contain any match: the ROM handler “raises” a PTE-MISS (PTMs) exception

The ROM strikes again
To refill the TLB, the ROM TLB-refill handler uses the following CP0 registers:
- EntryHi
- EntryLo
- Index
- Random
and may use some special CP0 instructions:
- TLBW I (TLB Write Indexed)
- TLBW R (TLB Write Random)
- TLBP (TLB Probe)
- TLBR (TLB Read)
- TLBCLR (TLB Clear)
uMPS memory management

CP0 registers for VM management:

EntryHi

<table>
<thead>
<tr>
<th>SEG NO</th>
<th>VPN</th>
<th>ASID</th>
<th>Unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>29</td>
<td>12</td>
<td>11</td>
</tr>
</tbody>
</table>

EntryLo

<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></td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

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)

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 (BadPT) 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…
if, scanning the \texttt{PgTbl}, a match is found:

- the matching \texttt{PTE} gets loaded (a \texttt{TLBWR} is performed)
- a \texttt{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 \texttt{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 \texttt{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 AMIKaya project phase2 require the virtual memory management?

**Some interesting answers:**

- 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 \texttt{Status.BEV} is set
    - 0x0000.0000 if \texttt{Status.BEV} is not set
  - Could the ROM TLB-refilling algorithm be made smarter?
    - yes, by defining different specifications (eg. using an ordered list for the \texttt{PgTbls})
  - Who fills the Segment Tables and the \texttt{PgTbls}?
    - the kernel itself (or some part of it, eg. a specialized VM process)

- 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
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 (e.g., if kernel starts with VM on, such a TLB entry is quite useful)

- Will AMIKaya project phase2 require virtual memory management?
  - NO