

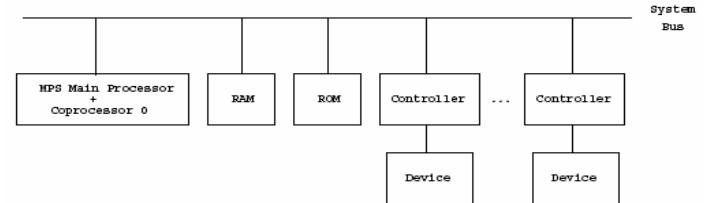
Laboratorio di Sistemi Operativi
Anno Accademico 2005-2006

uMPS Introduction
Part 2
Mauro Morsiani

Copyright © 2006 Mauro Morsiani
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license can be found at: <http://www.gnu.org/licenses/fdl.html>

uMPS processor architecture

? The uMPS architecture



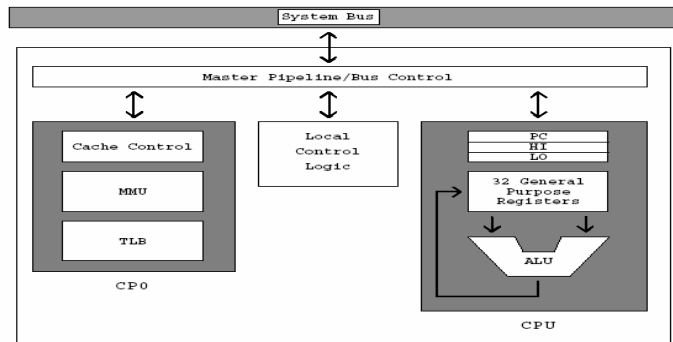
© 2006 Mauro Morsiani

2

uMPS processor architecture

? The MIPS processor architecture

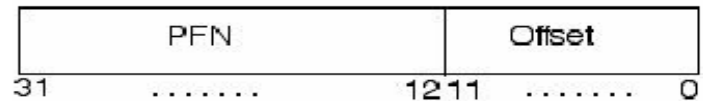
MIPS R2/3000 Architecture



uMPS memory management

? uMPS physical memory address format

? Physical Frame Number and Offset



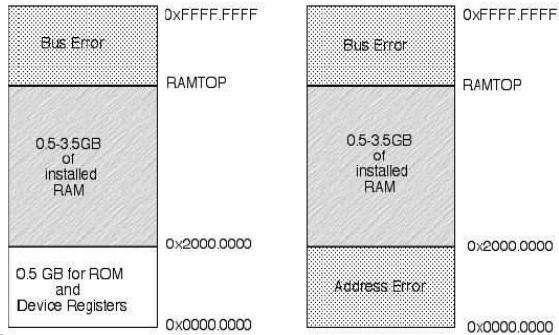
© 2006 Mauro Morsiani

4

uMPS memory management

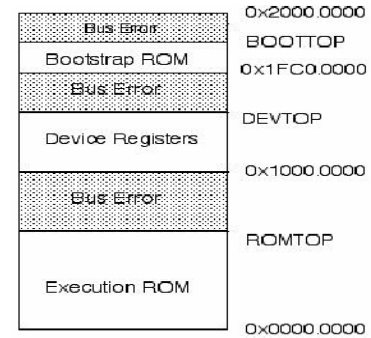
? uMPS physical memory map

? Kernel and User modes



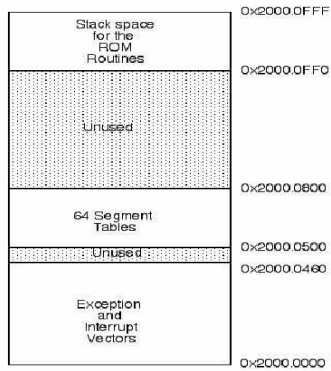
uMPS memory management

? ROM and Device Registers Area



uMPS memory management

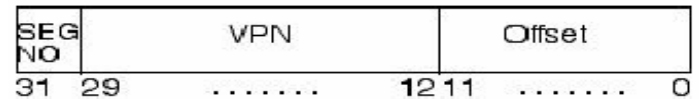
? ROM reserved frame (first RAM frame)



uMPS memory management

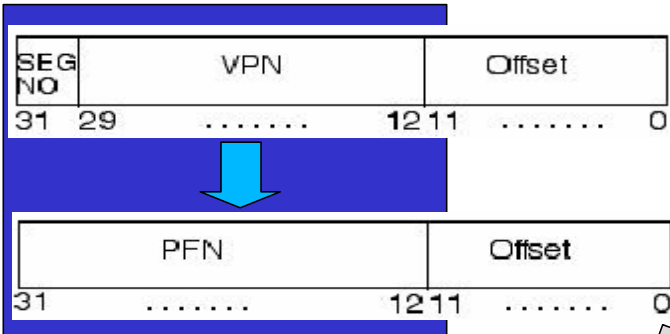
? uMPS virtual memory address format

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



uMPS memory management

? Mapping virtual memory to physical memory



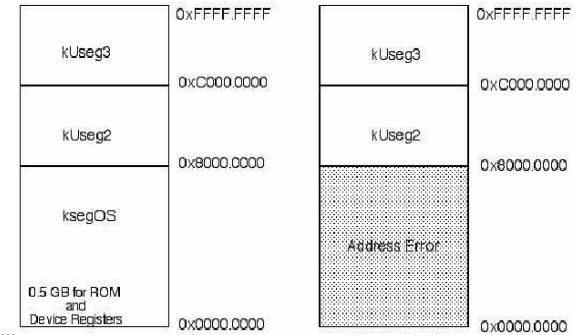
© 2006 Mauro Morsiani

9

uMPS memory management

? uMPS virtual memory map

? Kernel and User modes



© 2006 Mauro Morsiani

10

uMPS memory management

? uMPS virtual memory segment description

- ? *ksegOS* (**SEGNO 00** and **01**: 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)

© 2006 Mauro Morsiani

11

uMPS memory management

? uMPS virtual memory segment description (cont'd)

- ? *kUseg2* (**SEGNO 10**: 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 11**: 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**

© 2006 Mauro Morsiani

12

uMPS memory management

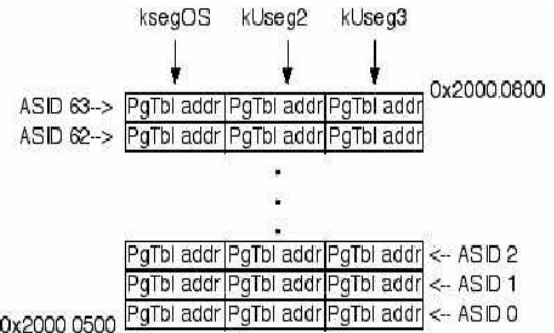
? uMPS virtual memory management scheme

- 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

uMPS memory management

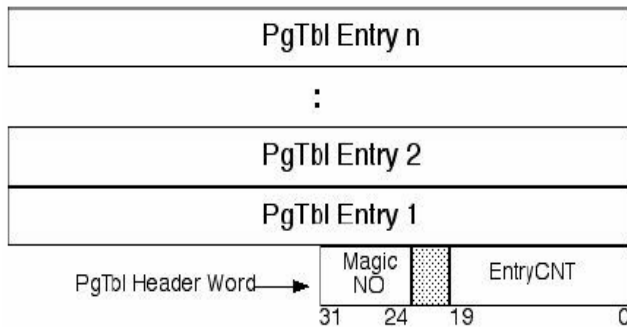
? Segment Table format:

- All **PgTbl** addresses are *physical memory addresses*



uMPS memory management

? PgTbl format:



uMPS memory management

? 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?

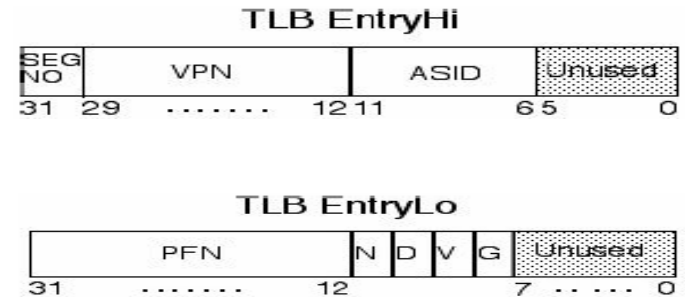
uMPS memory management

? 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:



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 (*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

uMPS memory management

? 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...

uMPS memory management

? 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...

uMPS memory management

? 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

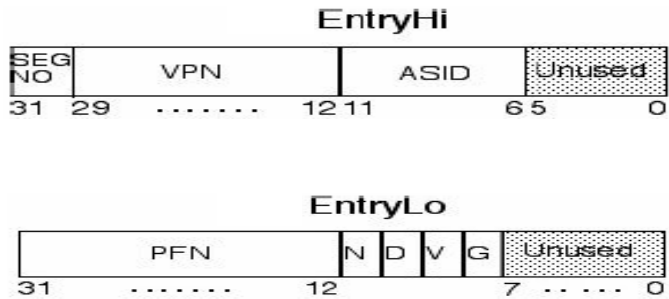
uMPS memory management

? 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:
 - ? **TLBWI** (*TLB Write Indexed*)
 - ? **TLBWR** (*TLB Write Random*)
 - ? **TLBP** (*TLB Probe*)
 - ? **TLBR** (*TLB Read*)
 - ? **TLBCLR** (*TLB Clear*)

uMPS memory management

? CP0 registers for VM management:

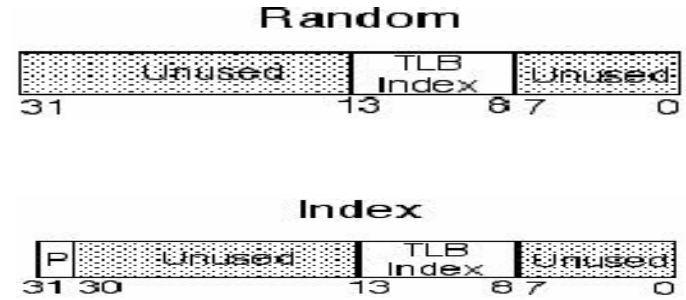


© 2006 Mauro Morsiani

25

uMPS memory management

? CP0 registers for VM management (cont'd):



© 2006 Mauro Morsiani

26

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)

© 2006 Mauro Morsiani

27

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...

© 2006 Mauro Morsiani

28

uMPS memory management

- ? **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

uMPS memory management

- ? **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 **PgTbIs**?
 - ? Why the kernel should start with virtual memory off?
 - ? Could the kernel be started with virtual memory on?
 - ? Is another memory management scheme 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 Kaya project phase2 require the virtual memory management?

uMPS 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 (eg. using an ordered list for the **PgTbIs**)
 - ? Who fills the Segment Tables and the **PgTbIs**?
 - ? the kernel itself (or some part of it, eg. a specialized VM process)

uMPS memory management

- ? **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*