

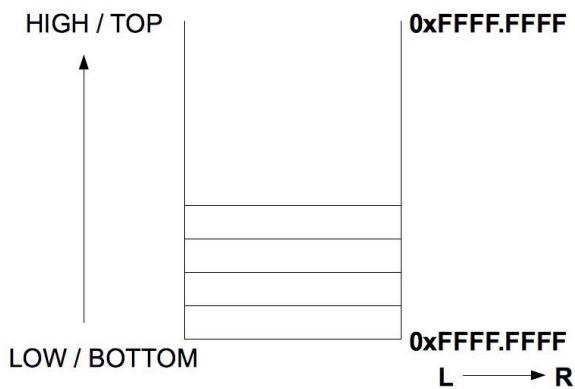
Capitolo 1

Introduction

Introduzione

Convenzioni:

- Le WORD che saranno definite sono in corsivo.
- I Registri, i Campi e le Istruzioni sono marcate in **grassetto**.
- Il campo “ F ” del registro “ R ” è definito come “ R.F ”.
- I bit di un’unità di memorizzazione sono numerati da destra a sinistra partendo da 0.
- L’i-esimo bit di un’unità di memorizzazione N sarà N [i].
- Gli indirizzi di memoria e i codici delle operazioni sono in HEX (esadecimale) e visualizzate in formato BIG-Endian, ovvero il primo bit di una WORD è quello più significativo.
- Seguire il disegno per interpretare i diagrammi che illustrano la memoria:



Capitolo 2

System Structure and Overview

Strutture di Sistema e Panoramica

uMPS contiene:

- una CPU
- un coprocessore, **CP0**, incorporato nella CPU
- una ROM e una RAM. La ROM contiene le routines / funzioni necessarie per il processo di Boot e per la gestione delle eccezioni.
- dei dispositivi periferici: 8 di ogni tipo (8 x disco, 8 x terminale x 2 (input / output), 8 x stampante, 8 x nastro)
- un BUS che connette tutti.

La CPU di uMPS implementa un accurata simulazione di un processore MIPS R2/3000 RISC e fornisce:

- Delle istruzioni di tipo RISC basate solo su paradigmi di tipo LOAD e STORE (Carica e Memorizza)
- Delle WORD a 32 bit sia per le istruzioni che per i registri. Tutti gli Indirizzi Fisici sono composti da 32 bit ovvero 1 WORD.

Lo spazio di indirizzamento fisico è di $2^{32} = 4 \text{ GB}$; ogni singolo byte (= 8 bit) ha il suo indirizzo.

doubleword = 64 bit, *halfword* = 16 bit

- 32 registri generali (**GPR**) rappresentati da **\$0 \$31**:

- Il registro **\$0** è impostato al valore 0. Questo registro ignora il caricamento di valori e ritorna sempre 0 in lettura e memorizzazione.
- I Registri **\$1 ... &31** supportano sia il caricamento (load) che la memorizzazione (store). In aggiunta al numero che li contraddistingue, ogni registro ha una connotazione mnemonica per individuarlo al meglio.

10 di questi registri sono per computazioni generali mentre il resto sono riservati a vari scopi.

Il più importante è il registro **\$28** chiamato **\$SP** e usato come stack pointer.

I registri **\$26** e **\$27**, chiamati anche **\$k0** e **\$k1** sono riservati ad uso esclusivo del kernel.

- 2 registri speciali, **HI** e **LO**, che sono utilizzati per mantenere i risultati delle operazioni di moltiplicazione e divisione.
- Un program counter, **PC**, adibito all'indirizzamento delle istruzioni (dice quale sarà la successiva istruzione da eseguire)

Il coprocessore, **CP0**, incorporato nella CPU fornisce:

- Supporto per 2 modalità d'uso della CPU: Kernel-Mode e User-Mode.
- Supporto per la gestione delle eccezioni (vedi capitolo 3)
- Traduzione di tutti gli indirizzi virtuali in fisici (vedi capitolo 4)
- e 2 possibili gestioni della memoria: con memoria virtuale attiva (**Status.VMc = 1**) o memoria virtuale disattiva (**Status.VMc = 0**)

CP0 implementa 8 registri di controllo:

5 (**Index**, **Random**, **EntryHI**, **EntryLO** e **BadVAddr**) sono usati per tradurre gli indirizzi virtuali in fisici. 2, **Cause** e **EPC**, sono usati dai meccanismi di gestione delle exception/interrupt per indicare quale tipo di eccezione o interruzione è avvenuta.

Infine, 1 registro **Status** leggibile e scrivibile che controlla l'uso del coprocessore, la modalità d'uso della CPU (Kernel-Mode e User-Mode), la modalità di traduzione degli indirizzi e il mascheramento dei bit di interrupt.

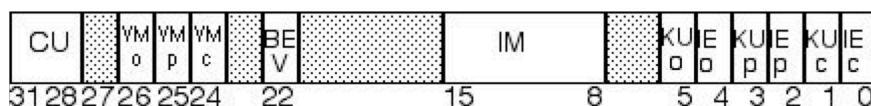


Figure 2.1: Status Register

Tutti i campi del registro Status sono leggibili e scrivibili. In particolare:

- **IEc**: bit 0 - E' il bit di controllo di tutti gli interrupt provenienti dai dispositivi esterni. Se settato a 0, indipendentemente da com'è settato lo **Status.IM**, tutti gli interrupt esterni sono disabilitati. Se settato a 1, tutti gli interrupt esterni accettati vengono controllati dallo **Status.IM**.
- **KUc**: bit 1 - E' il bit di controllo della modalità d'uso della CPU (Kernel-Mode e User-Mode). Se settato a 0 (**Status.KUc = 0**) il processore sarà in Kernel-Mode, altrimenti, **Status.KUc = 1**, il processore sarà in User-Mode.
- **IEp** e **KUp**: bit 2 e 3 - contengono i valori precedenti di **Status.IEc** e di **Status.KUc**. (servono quando si deve ripristinare un processo dopo un interrupt o un'eccezione)
- **IEo** e **KUo**: bit 4 e 5 - contengono i valori precedenti di **Status.IEp** e di **Status.KUp**. ("o" == "old") Questi 6 bit, **IEc**, **KUc**, **IEp**, **KUp**, **IEo**, **KUo** agiscono come 3 slot di profondità del **KU/IE** stack. Ogniqualvolta un'eccezione viene sollevata, viene fatta un PUSH dello stack e ogniqualvolta un'esecuzione interrotta riprende, viene fatto un POP allo stack. Vedi il capitolo 3.2 per maggiori dettagli esplicativi.
- **IM**: bit 8-15 - E' la maschera degli Interrupt. Sono gli 8 bit che abilitano / disabilitano gli interrupt esterni accettati. Quando un dispositivo (device) solleva un'interruzione sulla sua i-esima linea, il processore accetta l'interrupt solo se il corrispondente **Status.IM[i]** è attivo.
- **BEV**: bit 22 - Il Bootstrap Exception Vector: questo bit determina l'indirizzo iniziale degli Exception Vectors.
- **VMc**: bit 24 - Indica la modalità di gestione della memoria: Virtual Memory OFF: **Status.VMc = 0** Virtual Memory ON: **Status.VMc = 1**.
- **VMp**: bit 25 - Contiene il precedente valore di **Status.VMc**
- **VMo**: bit 26 - Contiene il precedente valore di **Status.VMp** ("o" == "old") Questi 3 bit: **VMc**, **VMp** e **VMo** agiscono come 3 slot del **VM** stack. Ogniqualvolta un'eccezione viene sollevata, viene fatta un PUSH dello stack e ogniqualvolta un'esecuzione interrotta riprende, viene fatto un POP allo stack. Vedi il capitolo 3.2 per maggiori dettagli esplicativi.

- **CU**: bit 28-31 - è un campo di 4 bit adibito al controllo dell'uso del coprocessore. I bit sono numerati da 0 a 3; Settando lo **Status.CU[i]** a 1 si premette l'uso dell'i-esimo coprocessore. uMPS implementa il **CP0** in modo tale che solo lo **Status.CU[0]** sia scrivibile. Gli altri 3 bit sono solamente leggibili e permanentemente settati a 0. Tentare di usare un coprocessore senza il corrispondente bit di controllo settato a 1 causerà il sollevamento della Coprocessor Unusable exception. In particolare un processo non attendibile potrebbe provenire dal **CP0** acceduto settando **Status.CU[0] = 0**. **CP0** è sempre accessibile / usabile quando si è in Kernel-Mode (**Status.KUc=0**) indipendentemente dal valore di **Status.CU[0]**.
 Punto importante: **CP1** (il coprocessore per le istruzioni floating point / virgola mobile) non è implementato, quindi l'esecuzione di istruzioni floating point generano una Coprocessor Unusable Exception.

2.0.1 Stato del Sistema al tempo di Boot e Reset

Quando uMPS è acceso per la prima volta, o resettato (vedi capitolo 8), il Coprocessore 0 (**CP0**) è abilitato (**Status.CU[0] = 1**), **VM** è OFF (**Status.VMp = 0** e **Status.VMc = 0**), gli interrupt sono disabilitati (**Status.IEc = 0**), il Bootstrapt Exception Vector è ON (**Status.BEV = 1**) e l'User-Mode è OFF (**Status.KUc = 0** quindi si è in Kernel-Mode).

Esempio di registro **Status** al tempo di BOOT / RESET: Status = 0x1040.0000

Il **PC** è settato al Bootstrap ROM code (0x1FC0.0000 - vedi Capitolo 4.1) e lo **\$SP** è settato a 0x0000.0000. Vedi Capitol 6.1 per una descrizione delle azioni del Bootstrap ROM code.

Capitolo 3: Gestione delle Eccezioni

Un eccezione è un interruzione dell'esecuzione corrente causata da un errore o altro.

Esistono 4 categorie di eccezioni:

1. Program Traps (**PgmTrap**):
 - a. Indirizzo Errato
 - b. Bus Errato
 - c. Esecuzione di Funzioni Riservate al Kernel Mode
 - d. Inusabilità del CP0
 - e. Overflow aritmetico
2. SYSCALL e Breakpoint (**SYS/Bp**):
 - a. System Call
 - b. Breakpoint
3. TLB Management (**TLB**):
 - a. Modifiche al TLB
 - b. TLB invalido
 - c. PTE-MISS
 - d. Eccezione sulla errata Page Table (Bad-PgTable)
4. Interrupts (**Ints**)
 - a. Interrupt Exception di un dispositivo (Lettura, Scrittura)
 - b. Interrupt Exception SW

3.1 Tipi di Exception:

3.1.1: Program Traps - PgmTrap:

- Address Error (AdEL e AdES): questa eccezione è sollevata quando:
 - Un'istruzione load/store **fetch** una word non allineata su una word **boundary**
 - Una istruzione load/store di una mezzaword non è allineata su una mezzaword **boundary**
 - Un accesso user-mode su di un indirizzo al di sotto di 0x2000.0000 quando lo Status.VMc=0.
 - Un accesso user-mode su di un indirizzo al di sotto di 0x8000.0000 quando lo Status.VMc=1.
- BUS error (IBE e DBE): questa eccezione viene sollevata quando si tenta un accesso a una locazione inesistente della memoria fisica, o quando un **attempt** è fatto per scrivere nella ROM..
- Reserved Instruction (RI): questa eccezione viene sollevata quando un'istruzione è mal-formata, non riconoscibile, o se privilegiata ed eseguita in user-mode.
- Coprocessor Unusable (CpU): questa eccezione viene sollevata quando un'istruzione richiede di usare o accedere a un coprocessore disinstallato o in quel momento non disponibile.
Tutti i registri di controllo di UMPS sono parti del CP0, quindi, accedere ad essi quando lo Status.KUc=1 e lo Status.CU=0 solleva questa eccezione.
CP0 è sempre avviabile in Kernel-Mode (Status.KUc=0)
- Overflow aritmetico (Ov): questa eccezione viene sollevata in seguito all'esecuzione di una ADD o a una SUB che sono andati in overflow in complemento a 2.

3.1.2 SYSCALL/Breakpoint (SYS/Bp)

Queste eccezioni vengono sollevate ogniqualvolta vengono eseguite istruzioni BREAK o SYSCALL. Queste eccezioni sono usate dai processi per chiedere al sistema operativo dei servizi.

3.1.3 TLB Management

I dettagli su quando queste eccezioni vengono sollevate si trovano al Capitolo 4.

- TLB Modification (Mod): Questa eccezione è sollevata quando una scrittura richiesta "*matching entry is found, the entry is marked valid, but not dirty/writable*".
- TLB Invalid (TLBL & TLBS)
- Bad-PgTbl (BdPT): Questa eccezione viene sollevata durante un TLB-Refill e il Gestore della ROM-TLB-Refill determina che:
 - l'indirizzo di PTE è minore di 0x2000.0000.
 - l'indirizzo di PTE non è *word-aligned*
 - Il Magic Number di PTE non è 0x2A
 - (l'indirizzo di PTE + 4 + 8 * Contatore Interno di PTE) < RAMTOP.

Vedere 4.3.4 per maggiori informazioni su TLB-Refill.

-PTE-MISS (PTMs):

Questa eccezione viene sollevata durante un TLB-Refill e il Gestore della ROM-TLB-Refill non trova il "matching" entry desiderata mentre ricercava linearmente nel PgTbl.

Vedere 4.3.4 per maggiori informazioni su TLB-Refill.

3.1.4: Interruzioni (Ints)

Un'interruzione è un'eccezione tipicamente sollevata da un dispositivo esterno al processore. uMPS fornisce 8 linee di interrupt monitorate ed ognuna supporta un certo numero di dispositivi connessi ad essa.

Le linee di Interrupt sono numerate da 0 a 7.

Più basso è il numero della linea di interrupt più alta è la precedenza di servizio per il dispositivo connesso a quella linea. Solo 6 linee di interrupt sono disponibili per i dispositivi esterni.

Le linee di interrupt 0 e 1 sono riservate per gli interrupt software.

I SW interrupt, al contrario di quelli generati dai dispositivi esterni che vengono sollevati da eventi esterni (es.: completamento di un'operazione esterna), sono sollevati da eventi SW (es.: la scrittura di 1 in **Cause.IP[0]** o **Cause.IP[1]**).

3.2: Azioni del Processore in seguito a un Eccezione.

Ogniqualvolta avviene un'eccezione, ci sono un certo numero di azioni che il processore uMPS compie. Queste azioni vengono fornite automaticamente e sono:

- 1) **CP0** mette in **EPC** (Exception PC) il valore del **PC** corrente.
- 2) Viene messo in **Cause.ExcCode** il codice dell'eccezione avvenuta.
- 3) I registri **VM**, **KU/IE** del registro **Status** di **CP0** vengono pushati nel seguente modo:

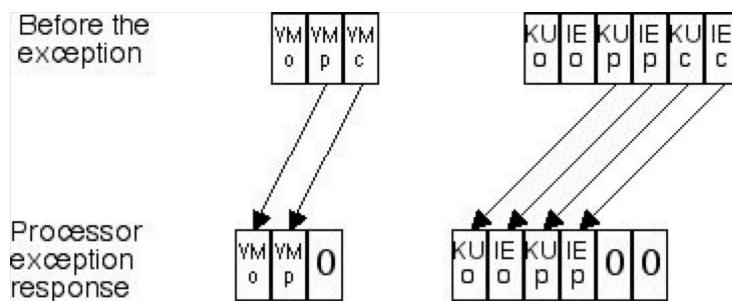


Figure 3.1: **VM** and **KU/IE** Stack Push

Quindi il processore entra SEMPRE in un Gestore di Eccezioni (Exception Handler) in Kernel-Mode, con Virtual Memory disattivata (**Status.VMc=0**) e tutti gli interrupt disabilitati

In aggiunta, il processore potrà incontrare anche:

- Address Error exceptions:

Carica nel registro **BadVAddr** di **CP0** l'indirizzo colpevole. Questo registro può essere usato solo quando lo **Status.VMc = 0**, ovvero quando la Virtual Memory è OFF

- Interrupt exceptions:

Aggiorna il campo **Cause.IP** in modo da mostrare su quali linee di interrupt è pendente/in attesa. Nel caso di un interrupt sw, questa operazione era già stata fatta dal momento che era questa la causa del software interrupt.

- Coprocessor Unusable exceptions:

Inserisce il numero appropriato del coprocessore nel campo **Cause.CE**.

- TLB-Modificato, TLB-Invalid, PTE-MISS, and Bad-PgTbl exception

(queste eccezioni possono essere sollevate solo se la Virtual Memory è attiva, ovvero **Status.VMc=1**):

- Viene caricato in **BadVAddr** di **CP0** il valore dell'indirizzo virtuale che ha fallito la traslazione.
- Viene caricato in **EntryHi.SEGNO** e in **EntryHi.VPN** con il **SEGNO** e il **VPN** dell'indirizzo virtuale che ha fallito la traslazione.

Infine, in **PC** viene caricato l'indirizzo di uno dei 2 ROM-based exception handlers. Uno, nella Bootstrap ROM code (0x1FC0.0180) è usato ogniqualvolta lo **Status.BEV=1**.

L'altro, identificato dal codice d'esecuzione (0x0000.0080) è usato ogniqualvolta **Status.BEV=0**.

Questo permette ai Gestori delle Eccezioni di essere usati durante il bootstrap del Sistema Operativo, **Status.BEV=1**, e durante la regolare esecuzione del processore, **Status.BEV=0**.

In sintesi, quando un'eccezione viene sollevata, il processore fornisce un numero di step automatici. Questi includono un'operazione di PUSH su **KU/IE** e **VM** stacks, il salvataggio del **PC** corrente, il settaggio del codice dell'eccezione nel registro Cause, possibilmente settando qualche altro registro di **CP0** (ad esempio **BadVAddr**) e infine carica in **PC** uno dei 2 indirizzi che dipendono dal settaggio di **Status.BEV**.

Ciò che accade poi è gestito dalla ROM exception handler il cui indirizzo è riposto in **PC**. Il lavoro del ROM Exception Handler è facilitato dal passare la gestione dell'eccezione al S.O. A tal fine, il ROM Exception handler salverà automaticamente il corrente stato del processore - memorizzando il contenuto dei suoi registri in una locazione di memoria data - e caricando poi il nuovo stato del processore - caricando nei registri del processore i nuovi valori salvati nella locazione di memoria fornita.

3.2.1 Stato del Processore

Lo stato del processore è definito come 35 blocchi di WORD che contengono i seguenti registri:

- 1 WORD per il registro **IEntryHI** del **CP0**. Questo registro contiene il corrente **ASID** (**EntryHI.ASID**).
- 1 WORD per il registro **Cause** del **CP0**.
- 1 WORD per il registro **Status** del **CP0**.
- 1 WORD per il registro **PC** (New Area) o per **EPC** (Old Area) - Lo slot **PC/EPC** nello stack.
- 29 WORD per i registri generali (**GPR**) di **CP0**. I registri **\$0**, **\$k0** e **\$k1** sono esclusi.
- 2 WORD per i registri **HI** e **LO** di **CP0**.

Non esistono istruzioni non-interrompibili che caricano o salvano lo stato del processore, il ROM Exception Handler salva e carica lo stato del processore automaticamente, disabilitando gli interrupt e poi, registro-dopo-registro, prima salva il corrente stato del processore - i 35 registri definiti sopra - poi carica negli stessi 35 registri i nuovi valori.

*Punto Importante: Il corrente stato del processore (ovvero il corrente contenuto dei 35 registri definiti sopra) del ROM Exception Handler è lo stesso stato in cui il processore era al tempo in cui l'eccezione è stata sollevata, salvo che il push degli stack **KU/IE** e **VM**, il **PC** al tempo dell'eccezione è stato memorizzato in **EPC** e **Cause.ExcCode** è stato aggiornato in modo appropriato.*

Quando il ROM Exception Handler salva lo stato del processore, l'**EPC** viene salvato nello stack allo slot **PC/ECP**. Quando il ROM Exception Handler carica un nuovo stato del processore, il contenuto del **PC/EPC** slot viene caricato nuovamente in **PC**.

3.2.2. Old Area e New Area dello Stato del Processore

Il ROM Exception handler ha bisogno di una locazione di memoria in cui salvare lo stato corrente del processore e dell'indirizzo della locazione di memoria da cui prelevare il nuovo stato del processore da caricare.

Il primo frame della RAM fisica (posizionata al 0x2000.0000), chiamato il ROM Reserved Frame, è riservato proprio a questo scopo, salvarvi dentro lo stato corrente del processore al momento dell'eccezione.

Il ROM Reserved Frame è usato solo per salvare la tabella dei segmenti del processo e fornire dello spazio nello stack per l'esecuzione del ROM Exception handler e del ROM-TLB-Refill handler.

Vedere capitolo 4.3 per maggiori dettagli circa la tabella dei segmenti salvata nella ROM Reserved Frame.

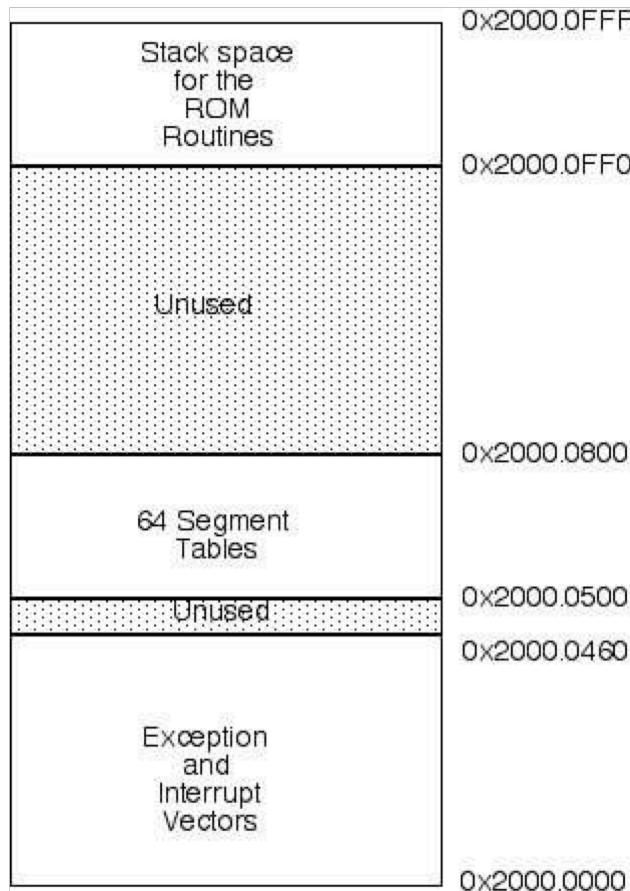


Figure 3.2: ROM Reserved Frame

La posizione in cui il ROM Exception handler salva il corrente stato del processore dipende da che tipo di eccezione è avvenuta. Il corrente stato del processore viene salvato nella Ints Old Area o nella TLB Old Area o nella SYS/Bp Old Area o nella PgmTrap Old Area.

I registri del processore sono poi caricati dalla corrispondente New Area dal ROM Exception handler.

STACK
SPACE
per le
ROM Routines

SYSCALL/BREAK New Area	0x2000.03D4
SYSCALL/BREAK Old Area	0x2000.0348
Program Trap New Area	0x2000.02BC
Program Trap Old Area	0x2000.0230
TLB Management New Area	0x2000.01A4
TLB Management Old Area	0x2000.0118
Interrupt New Area	0x2000.008C
Interrupt Old Area	0x2000.0000

Figure 3.3: Old and New State Areas

3.3 Il Registro Cause di CP0

Cause è un registro di **CP0** che contiene informazioni circa la corrente eccezione avvenuta e/o gli interrupt pendenti (in attesa) provocati dai dispositivi.

Come descritto sotto, **Cause** è settato dall'HW al tempo in cui l'eccezione viene sollevata ed è salvato come parte del corrente stato del processore nell'appropriata Old Area nella ROM Reserved Frame.

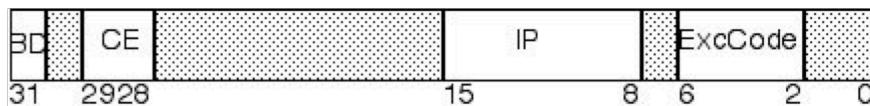


Figure 3.4: The **Cause CP0Register**

I campi del registro **Cause** sono tutti leggibili ma non scrivibili, ad eccezione del **Cause.IP[0]** e del **Cause.IP[1]**.

Il registro **Cause** è composto da:

- **ExcCode** (bit 2-6): è un campo di 5 bit che fornisce il codice dell'eccezione avvenuta.
- **IP** (bits 8-15): è un campo di 8 bit indicante su quale linea di interrupt, gli interrupt provenienti dai dispositivi sono pendenti (= in attesa).

Se un interrupt è pendente sulla linea di interrupt *i*, allora **Cause.IP[i]** sarà settato a 1.

Punto Importante: Più linee di interrupt possono essere attive allo stesso tempo. Inoltre, più dispositivi sulla stessa linea di interrupt possono richiedere servizi.

Cause.IP è sempre aggiornato, risponde immediatamente agli eventi causati dai dispositivi esterni.

I primi 2 bit di **Cause.IP** sono i soli a poter essere scritti oltre che letti in Cause. Un interrupt software può essere impostando a 1 uno di questi 2 bit. (**Cause.IP[0] = 1** o **Cause.IP[1] = 1**)

Vedere Capitolo 6.3.2 per una descrizione su come scrivere nel registro Cause e vedere Capitolo 5.2.3 per la procedura per il riconoscimento dell' Interrupt Software.

- **CE** (bits 28-29): è un campo di 2 bit che indicano quale coprocessore era stato acceduto illegalmente quando viene sollevata una Coprocessor Unusable Exception.
- **BD** (bit 31): Questo singolo bit indica l'ultima eccezione sollevata e avvenuta in un Branch Delay slot.

uMPS riproduce fedelmente la simulazione di un R2/3000 come meglio possibile.

Come descritto al Capitolo 7.3 il codice di uMPS è compilato usando lo standard MIPS R2/3000 cross compiler. Questo compilatore organizza il risultante codice macchina per un reale processore MIPS R2/3000 in cui include una presa di coscienza dei *ritardi di caricamento* e dei *branch delay slot*.

I ritardi di caricamento e i brach delay slot sono convenzioni/tecniche usate dai veloci processori RISC per prevenire il rallentamento delle pipeline o gli stalli. (Vedi [2] per maggiori informazioni).

In uMPS viene simulato un processore in cui non ci sono né fasi di pipeline né sovrapposizioni di istruzioni eseguite.

Anche se i ritardi di caricamento e i branch delay slot sono presenti - dal compilatore - essi sono correttamente gestiti. Quindi, il **BD** bit può essere ignorato in modo sicuro.

I 15 codici che è possibile trovare in Cause.ExcCode sono:

Number	Code	Description
0	<i>Int</i>	External Device Interrupt
1	<i>Mod</i>	TLB-Modification Exception
2	<i>TLBL</i>	TLB Invalid Exception: on a Load instr. or instruction fetch
3	<i>TLBS</i>	TLB Invalid Exception: on a Store instr.
4	<i>AdEL</i>	Address Error Exception: on a Load or instruction fetch
5	<i>AdES</i>	Address Error Exception: on a Store instr.
6	<i>IBE</i>	Bus Error Exception: on an instruction fetch
7	<i>DBE</i>	Bus Error Exception: on a Load/Store data access
8	<i>Sys</i>	Syscall Exception
9	<i>Bp</i>	Breakpoint Exception
10	<i>RI</i>	Reserved Instruction Exception
11	<i>CpU</i>	Coprocessor Unusable Exception
12	<i>OV</i>	Arithmetic Overflow Exception
13	<i>BdPT</i>	Bad Page Table
14	<i>PTMs</i>	Page Table Miss

3.4 La Verità circa la ROM:

Come sarà più ampiamente descritto al Capitolo 4.5, la traduzione degli indirizzi virtuali, e la gestione degli eventi TLB-Refill in particolare, è trattata, in modo cooperativo, dall'HW fisico e dal ROM-TLB-Refill handler. L'HW fisico comprende solo il valore del Cause.ExcCode, un valore tra 0 e 12. Come descritto nel Capitolo 4.5, è il ROM-TLB-Refill handler che altera il valore del Cause.ExcCode, nella TLB Old Area da entrambe le TLBL o le TLBS, il codice impostato dall'HW fisico, a entrambe le BdPT o PTMs come indicato.

CAPITOLO 4 Gestione della Memoria

L'uMPS supporta la memoria virtuale, ci sono 2 tipi di memoria: quella fisica e quella virtuale. Partiamo con quella fisica.

4.1 Memoria Fisica

Lo spazio di indirizzamento fisico è diviso in spazi uguali (frame) di 4 KB ciascuno. Quindi un indirizzo fisico ha 2 componenti; un 20-bit *Physical Frame Number* o **PNF**, e un 12-bit di **Offset** (spiazzamento) dentro al frame.

L'Indirizzo fisico ha il seguente formato:

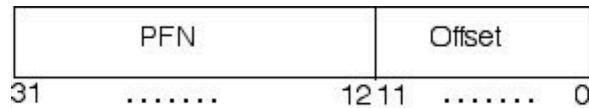


Figure 4.1: Physical Address Format

Questo significa che uMPS può avere fino a 2^{20} (o circa 1 M) di frame di memoria.

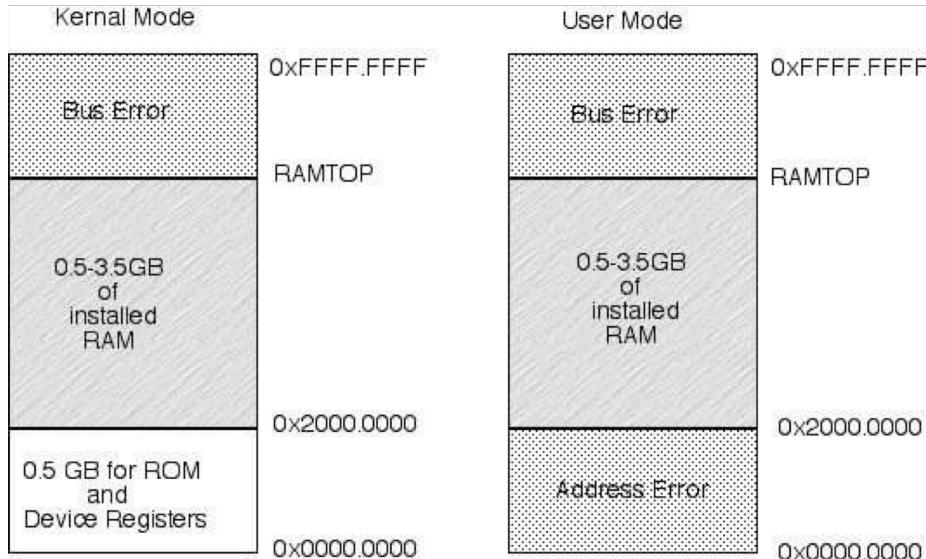


Figure 4.2: The Physical Address Space

Lo spazio di indirizzamento fisico è lo stesso sia sotto Kernel-Mode che sotto User-Mode eccetto che per 1 differenza. Come indicato dalla figura 4.2, i primi 2^{17} (circa 128 K) frame, che ammontano ai primi 0,5 GB di memoria, possono solo essere acceduti in Kernel-Mode.

La CPU deve essere in Kernel-Mode per poter leggere o scrivere ogni indirizzo di questo range (= intervallo).

Quando si tenta di accedere a uno degli indirizzi di questo intervallo in User-Mode, si provoca il sollevamento della Address Error exception.

La RAM fisica installata parte dall'indirizzo 0x2000.0000 e continua fino a RAMTOP. Quest'area conterrà:

- Il codice del sistema operativo (text), le variabili e le strutture globali (data) e lo/gli stack(s).
- I processi utente text, data e stacks.
- La ROM Reserved Frame. Come descritto al Capitolo 3.2.2, il Codice della ROM ha bisogno di una memoria in cui essere scritto. I primi 4 KB (il primo frame) della RAM fisica sono riservati per questo scopo.

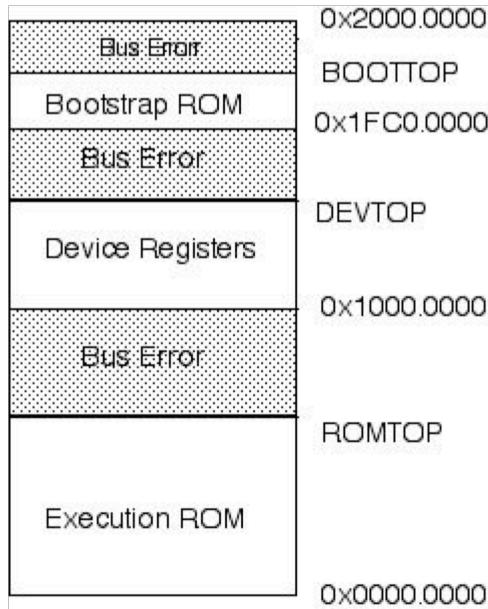


Figure 4.3: ROM Areas and Device Drivers

- Il primo 0,5 GB dello spazio di indirizzamento fisico, come illustrato dalla figura 4.3. è riservato per:
- L'Esecuzione del codice ROM. Questo segmento di codice read-only parte da 0x0000.0000 e arriva fino a ROMTOP
 - I Registri dei dispositivi. Quest'area leggibile/scrivibile comincia da 0x1000.0000 e si estende fino a DEVTOP.
 - Il codice di Bootstrap. Questo segmento di codice read-only parte da 0x1FC0.0000 e va fino a BOOTTOP.

Ogni tentativo di accesso a un'area indefinita della memoria (ROMTOP – 0x1000.0000, DEVTOP – 0x1FC0.0000, BOOTTOP – 0x2000.0000 e RAMTOP – 0xFFFF.FFFF) genererà un Bus Error exception.

Capitolo 5: Device Interface

uMPS supporta 4 diverse classi di dispositivi esterni: dischi, nastri, stampanti e terminali. Inoltre, uMPS può supportare fino a 8 istanze per ognuno dei 4 tipi (8x dischi, 8x nastri, 8x stampanti e 8x terminali).

Ogni singolo dispositivo è controllato da un *Controller*. I Controller scambiano informazioni con il processore tramite i *device register* che sono speciali locazioni di memoria.

Un *device register* è un blocco consecutivo di 4 WORD di memoria. Scrivendo e leggendo lo specifico campo del device register dato, il processore può eseguire comandi, testare lo Status del dispositivo e rispondere.

uMPS implementa un full-handshake interrupt-driven protocol. Specificatamente:

1. Viene comunicata l'operazione da eseguire sul dispositivo i scrivendo il codice del comando che esegue l'operazione nel device register del dispositivo i.
2. Il Controller del dispositivo i risponde avviando l'operazione e modificando un campo status del device register i.
3. Quando l'operazione viene completata, il Controller del dispositivo i modificherà di nuovo qualche campo del device register i; includendo il campo Status. Inoltre, il Controller del dispositivo i genererà un interrupt exception asserendo (setta a 1) l'appropriata linea di interrupt. L'interrupt generato informa il processore che l'operazione richiesta si è conclusa con successo e che il dispositivo richiede ora la sua attenzione.
4. L'interrupt lanciato viene riconosciuto scrivendo il suo codice di riconoscimento nel device register del dispositivo i.
5. Il controller del dispositivo i disasserirà (setta a 0) la linea dei interrupt e il protocollo può ripartire.

I device register sono posizionati nella parte bassa della memoria dall'indirizzo 0x1000.0000. Come spiegato nel Capitolo 4, indipendentemente dallo **Status.VMc**, tutti gli indirizzi compresi tra 0x1000.0000 e il DEVTOP sono interpretati come indirizzi fisici. Inoltre, il device register può solo essere acceduto quando lo **Status.KUc=0**.

La seguente tabella descrive le corrispondenze tra i tipi di dispositivo e le linee di interrupt.

Interrupt Line #	Device Class
2	Bus (Interval Timer)
3	Disk Devices
4	Tape Devices
5	Network (Ethernet) Devices
6	Printer Devices
7	Terminal Devices

5.1 Device Register

Tutti i dispositivi, eccetto il bus, condividono la stessa Device Register Structure.

Field #	Address	Field Name
0	(base) + 0x0	STATUS
1	(base) + 0x4	COMMAND
2	(base) + 0x8	DATA0
3	(base) + 0xc	DATA1

Device Register Structure

Mentre ogni classe di dispositivi ha un uso specifico e un formato per ognuno di questi campi, tutte le classi dei dispositivi, eccetto i terminali, usano:

- **COMMAND**: per permettere al comando di essere rilasciato al Controller del dispositivo.
- **STATUS**: Contiene lo Status del dispositivo che il suo Controller comunicherà al processore
- **DATA0 & DATA1**: per passare dei parametri in più al Controller del Dispositivo o per passare dati dal Controller del Dispositivo.

Tutti i 40 device register in uMPS sono posizionati nella parte bassa della memoria a partire dall'indirizzo 0x1000.0000.

Quest'area include altre 3 strutture dati:

- *Bus Register Area*: contiene le informazioni sullo stato del sistema e il “Bus device register”
- *Installed Devices Bit Maps*: che indica quali dispositivi sono attualmente installati e dove si trovano
- *Interrupting Device Bit Map*: che indica quale dispositivo ha un interrupt pendente (in attesa)

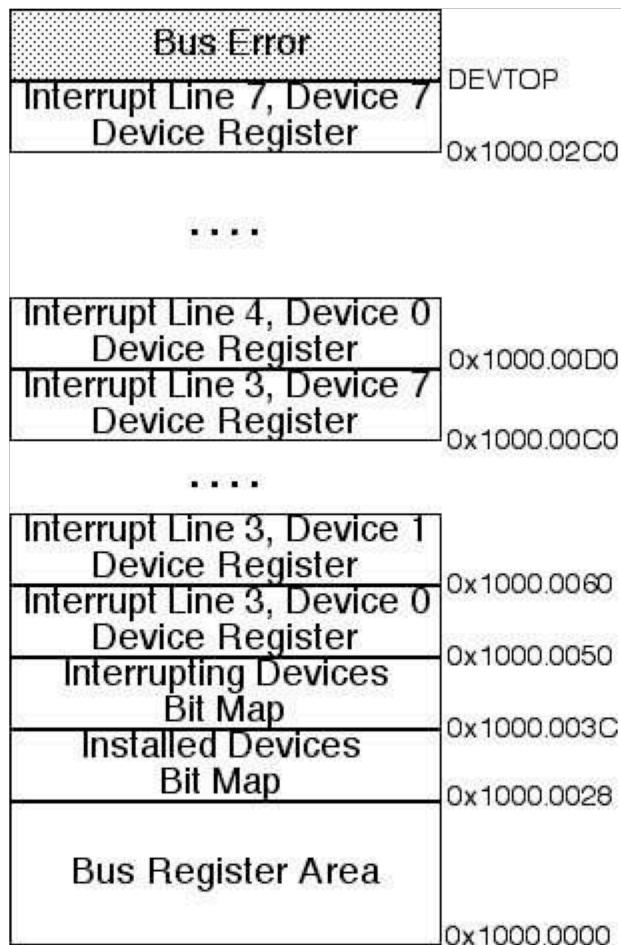


Figure 5.1: Device Registers Area

Data una linea di interrupt (IntLineNo) e un numero di dispositivo (DevNo) uno può calcolare l'indirizzo iniziale del device register del dispositivo nel seguente modo:

```
devAddrBase = 0x1000.0050 + (( IntLineNo - 3 ) * 0x80 ) + ( DevNo * 0x10 )
```

5.2 Il Bus Device e la Device Bit Maps

Il bus agisce come un'interfaccia fra il processore e la RAM, la ROM e tutti i dispositivi esterni. In particolare il bus fornisce le seguenti operazioni:

1. Gestione del **TOD** (Time Of Day) e dell' Interval Timer
2. Gestione delle linee di interrupt, i dispositivi attaccati a ogni linea e i device register
3. Deposito delle informazioni base del sistema

5.2.1 Bus Register Area

Il Bus Register Area è un'area di 10 WORD di memoria che contengono:

Physical Address	Field Name
0x1000.0000	RAM Base Physical Address
0x1000.0004	Installed RAM Size
0x1000.0008	Exec. ROM Base Physical Address
0x1000.000c	Installed Exec. ROM Size
0x1000.0010	Bootstrap ROM Base Physical Address
0x1000.0014	Installed Bootstrap ROM Size
0x1000.0018	Time of Day Clock - High
0x1000.001c	Time of Day Clock - Low
0x1000.0020	Interval Timer
0x1000.0024	Time Scale

Le prime 6 WORD di memoria/campi sono read-only (solamente leggibili) e settate al tempo di boot/reset.

RAMTOP è calcolata aggiungendo all'indirizzo fisico base della RAM la sua dimensione totale. ROMTOP e BOOTTOP sono calcolati nello stesso modo.

Le altre 3 WORD di memoria/campi sono:

1. **Time Scale:** Un campo read-only settato dal sistema al tempo di boot/reset e che indica il numero di clock ticks che si verificano in microsecondi. Come descritto al Capitolo 8 uno può modificare (aumentando o diminuendo) la velocità del processore. Quando la velocità del processore è settata a 1 MHz, il Time Scale è settato a 1. Questo campo è usato per aiutare a produrre tempi di computazione (calcolo) accurati.
2. **TOD (Time Of Day clock):** Questo registro read-only di 64 bit (diviso in high word e low word - parte alta e parte bassa della word) è settato da uMPS a 0 al tempo di boot/reset. Esso è incrementato di 1 dopo ogni ciclo del processore; ovvero dopo ogni clock tick. Ogni istruzione macchina di uMPS viene eseguita in 1 ciclo di clock, o in un clock tick.
3. **Interval Timer:** è una WORD leggibile e scrivibile che viene decrementata di 1 dopo ogni ciclo di clock, dopo ogni clock tick e settata da uMPS a 0xFFFF.FFFF al tempo di boot/reset. L'Interval Timer genererà un interrupt sulla linea di interrupt 2 ogniqualvolta transiterà da 0x0000.0000 a 0xFFFF.FFFF. Questo è il solo dispositivo attaccato alla linea 2, quindi ogni interruzione su questa linea può essere associata all'Interval Timer. Qualsiasi valore può essere caricato nell'Interval Timer e l'interrupt sarà generato solo al momento della transazione da 0x0000.0000 a 0xFFFF.FFFF. Attenzione a non caricare inavvertitamente nell'Interval Timer un valore troppo grande; per esempio un intero negativo piccolo. L'Interrupt prodotto dall'Interval Timer è riconosciuto dalla scrittura di un nuovo valore dentro il registro dell'Interval Timer.

5.2.2 Bit Map dei dispositivi installati (device installed)

Questa area di memoria è formata da 5 WORD che indicano quali dispositivi sono attaccati e a quali linee di interrupt. 1 WORD viene riservata per visualizzare i dispositivi attaccati dalla linea 3 alla 7.

Word #	Physical Address	Field Name
0	0x1000.0028	Interrupt Line 3 Installed Devices Bit Map
1	0x1000.002C	Interrupt Line 4 Installed Devices Bit Map
2	0x1000.0030	Interrupt Line 5 Installed Devices Bit Map
3	0x1000.0034	Interrupt Line 6 Installed Devices Bit Map
4	0x1000.0038	Interrupt Line 7 Installed Devices Bit Map

Ogni Device Bit Map WORD installata ha il seguente formato:

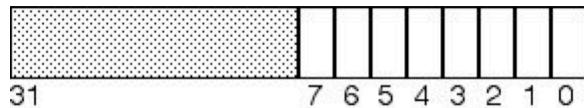


Figure 5.2: Installed Devices Bit Map

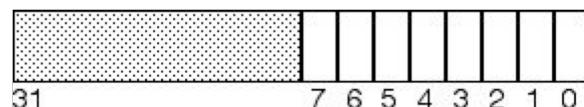
Se il bit i della WORD j è settato a 1, allora c'è un dispositivo (=device) con il device numer (numero di dispositivo) i attaccato alla linea di interrupt $j + 3$. Queste WORD sono settate da uMPS al tempo di boot/reset e non cambiano.

5.2.3 Bit Map dei dispositivi che hanno generato interruzioni (Interrupting Devices)

Questa è un'area di 5 WORD che indica quali dispositivi hanno interrupt pendenti. 1 WORD è riservata a indicare quale dispositivi hanno interrupt pendenti sulle linee 3-7.

Word #	Physical Address	Field Name
0	0x1000.003C	Interrupt Line 3 Interrupting Devices Bit Map
1	0x1000.0040	Interrupt Line 4 Interrupting Devices Bit Map
2	0x1000.0044	Interrupt Line 5 Interrupting Devices Bit Map
3	0x1000.0048	Interrupt Line 6 Interrupting Devices Bit Map
4	0x1000.004C	Interrupt Line 7 Interrupting Devices Bit Map

Le WORD della Interrupting Device Bit Map hanno lo stesso formato delle WORD della Installed Device Bit Map:



Quando il bit i della WORD j è settato a 1 allora il dispositivo i attaccato alla linea di interrupt $j + 3$ ha interrupt pendenti.

Un bit di un interrupt pendente è attivato automaticamente dall'HW ogniqualvolta un Controller di un dispositivo asserisce la linea di interrupt su chi esso è attaccato.

L'interrupt rimarrà pendente, cioè attivo, fino a che l'interrupt è riconosciuto.

Gli interrupt per i dispositivi esterni sono riconosciuti scrivendo il codice di riconoscimento nell'appropriato device register del dispositivo.

Ogniqualvolta che un dispositivo ha un interrupt pendente (in attesa) sulla linea di interrupt *i*, oltre ad attivare (porre a 1) il o i bit(s) dell'interrupt pendente nella *i* – 3 WORD dell' Interrupting Devices Bit Map, anche **Cause.IP[i]** sarà attivato (posto a 1). **Cause.IP [i]** sarà disattivato (Posto a 0) solo quando nessun dispositivo sarà più attaccato alla linea *i* con interrupt pendenti.

I bits degli interrupt pendenti, entrambi presenti in **Cause.IP** e nella Interrupting Device Bit Map, si attivano automaticamente in risposta all'asserzione (porre a 1) che i Controller dei dispositivi fanno sulla linea di interrupt. I Flag mascherati dall'interrupt, **Status.IEc** e **Status.IM** sono usati per determinare se un interrupt pendente genera un interrupt exception o no.

Un'interrupt pendente sulla linea di interrupt *i* genererà un interrupt exception se entrambi i flags **Status.IEc** e **Status.IM** sono a 1.

Non ci sono Interrupting Device Bit Maps per le linee di interrupt 0,1,2. **Cause.IP[2]=1** indica che c'è un interrupt pendente lanciato dall'Interval Timer, mentre **Cause.IP[0]=1** o **Cause.IP[1]=1** indica un Interrupt Software pendente. Come discusso prima, un interrupt lanciato dall'Interval Timer viene riconosciuto scrivendo un nuovo valore nel registro dell'Interval Timer. Un Software Interrupt viene riconosciuto settando direttamente il registro Cause.IP[0] = 0 o Cause.IP[1] = 0.

Punto Importante:

Possono esserci attive più di 1 linea di interrupt nello stesso tempo. Inoltre, più di 1 dispositivo, connessi alla medesima linea di interrupt, possono richiedere servizi.

Cause.IP e la Interrupting Devices Bit Map vengono sempre aggiornate e rispondono immediatamente agli eventi provocati dai dispositivi esterni.

5.3 Dispositivo Disco

uMPS supporta fino a 8 Hard Disk DMA (Direct Memory Access) leggibili e scrivibili.

Tutti i dischi di uMPS hanno blocchi di memorizzazione grandi quanto i frame, ovvero 4 KB.

Il campo DATA1 dei ogni disco installato è solamente leggibile e descrive le caratteristiche fisiche della geometria del dispositivo.

MAXCYL	MAXHEAD	MAXSECT
31	1615	87

Figure 5.3: Disk Device **DATA1** Field

I dischi di uMPS possono avere fino a 65536 cilindri/tracce, indirizzati da 0...**MAXCYL-1**; 256 testine (o superfici delle tracce), indirizzati da 0...**MAXHEAD-1**; e 256 settori/tracce indirizzati da 0...**MAXSECT-1**. Ogni blocco del disco (4 KB), o settore, può essere indirizzato dalle sue specifiche coordinate: Cilindro, Testina, Settore.

Il campo Status del registro del dispositivo disco è solamente read-only (leggibile) e conterrà 1 dei seguenti codici di stato:

Code	Status	Possible Reason for Code
0	Device Not Installed	Device not installed
1	Device Ready	Device waiting for a command
2	Illegal Operation Code Error	Device presented unknown command
3	Device Busy	Device executing a command
4	Seek Error	Illegal parameter/hardware failure
5	Read Error	Illegal parameter/hardware failure
6	Write Error	Illegal parameter/hardware failure
7	DMA Transfer Error	Illegal physical address/hardware failure

Codici di Stato.

Gli Status Code 1, 2, 4, 5, 6 e 7 sono codici di completamento. Un parametro illegale può essere un valore fuori dai limiti (es. Un numero di cilindro fuori dall'intervallo 0...MAXCYL-1), o un indirizzo fisico non esistente per un trasferimento mediante DMA.

Il campo DATA0 del registro del disco è leggibile e scrivibile e viene usato per indicare l'indirizzo di partenza di una scrittura o lettura di un'operazione di tipo DMA.

Dal momento che la memoria è indirizzata dal basso verso l'alto, questo indirizzo è l'indirizzo fisico di 4 KB più basso che può essere trasferito.

Il campo COMMAND del registro di un disco è leggibile e scrivibile ed è usato per dare comandi al disco.

CODE	COMMAND	OPERATION
0	RESET	Resetta il dispositivo e muove il boom al cilindro 0
1	ACK	Riconoscimento di un interrupt pendente
2	SEEKCYL	Ricerca il cilindro di numero CYLNUM
3	READBLK	Legge il blocco situato a (HEADNUM , SECTNUM) nel cilindro corrente e lo copia nella RAM partendo dall'indirizzo presente in DATA0
4	WRITEBLK	Copia 4 KB della RAM, partendo dall'indirizzo presente in DATA0, dentro il blocco situato a (HEADNUM , SECTNUM) nel cilindro corrente.

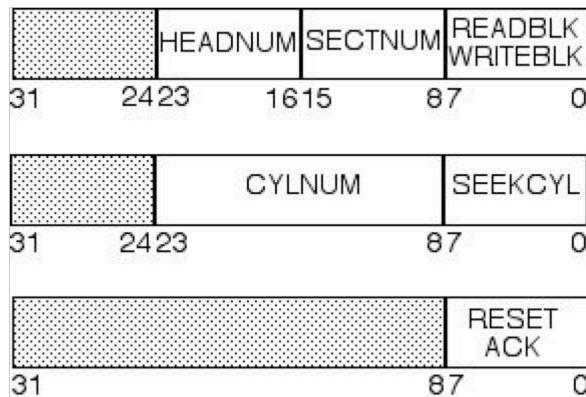


Figure 5.4: Disk Device **COMMAND** Field

Il formato del campo COMMAND, come illustrato nella figura 5.4, differisce a seconda di quale comando deve essere rilasciato:

Un'operazione del disco inizia caricando il valore appropriato nel campo COMMAND. Per tutta la durata dell'operazione, lo status del dispositivo sarà "Device Busy".

Al completamento dell'operazione, un interrupt viene sollevato e il codice di stato appropriato viene settato; "Device Ready" è il codice di stato nel caso sia andato tutto bene, o uno dei sette dei codici di errore altrimenti. L'interrupt è poi riconosciuto mediante l'emissione di un ACK o di un RESET command.

Le prestazioni del disco, visto che sia la lettura che la scrittura sono operazioni basate su DMA, dipendono fortemente dalla velocità di clock del sistema. Mentre una read/write può arrivare a estendersi a MB/sec in magnitudo, le operazioni hardware del disco rimangono in intervalli di millisecondi.

5.4 Tape Devices (Nastri)

uMPS supporta fino a 8 nastri rimovibili, con DMA, solamente leggibili.

Tutti i dispositivi nastro di uMPS hanno blocchi di 4 KB. Ogni campo DATA1 del registro dei nastri installati è read-only e descrive il corrente marcatore sotto la testa del nastro quando il dispositivo è inattivo.

Code	Marker	Meaning
0	EOT	End of Tape
1	EOF	End of File
2	EOB	End of Block
3	TS	Tape Start

Un nastro parte con un marcatore **TS** e termina con un marcatore **EOT**. Può essere visto come una collezione di blocchi, delimitato dai marcatori **EOB**, che si dividono in file, delimitati da marcatori **EOF**.

Un **EOF** agisce come un **EOB** per l'ultimo blocco del file e un **EOT** agisce come un **EOF** (e quindi anche uno **EOB**) per l'ultimo file del nastro.

Quando nessun nastro è caricato nel lettore di nastri, il campo **DATA1** conterrà il marcatore **EOT**, e il campo **STATUS** conterrà il codice di stato 1, ovvero "Device Ready".

Dal momento che non ci sono nastri presenti, il campo **COMMAND** non accetterà alcun comando. Solamente quando un nastro verrà caricato nel lettore, allora il dispositivo si attiverà e comincerà ad accettare comandi. Quando un nastro è caricato, il lettore lo riavvolge fino al suo marcatore **TS**.

Il campo **STATUS** del registro del lettore di nastri è read-only e potrà contenere 1 dei seguenti codici di stato:

Code	Status	Possible Reason for Code
0	Device Not Installed	Device not installed
1	Device Ready	Device waiting for a command
2	Illegal Operation Code Error	Device presented unknown command
3	Device Busy	Device executing a command
4	Skip Error	Illegal command/hardware failure
5	Read Error	Illegal command/hardware failure
6	Back 1 Block Error	Illegal command/hardware failure
7	DMA Transfer Error	Illegal physical address/hardware failure

Gli Status Code 1, 2, 4, 5, 6 e 7 sono codici di completamento. Un parametro illegale può essere la richiesta di leggere al di là dell' **EOT** o un indirizzo non esistente per un trasferimento con DMA.

Il campo **DATA0** del registro del disco è leggibile e scrivibile e viene usato per indicare l'indirizzo di partenza in un'operazione di lettura con DMA.

Dal momento che la memoria è indirizzata dal basso verso l'alto, questo indirizzo è l'indirizzo fisico di 4 KB più basso che può essere trasferito.

Il campo **COMMAND** del registro di un disco è leggibile e scrivibile ed è usato per dare comandi al lettore di nastri:

CODE	COMMAND	OPERATION
0	RESET	Resetta il dispositivo e riavvolge il nastro fino al marcatore TS
1	ACK	Riconoscimento di un interrupt pendente
2	SKIPBLK	Manda avanti il nastro fino a un EOB o un EOT
3	READBLK	Legge il blocco corrente fino al successivo EOB o EOT e lo copia nella RAM partendo dall'indirizzo presente in DATA0
4	BACKBLK	Riavvolge il nastro all' EOB o EOT precedente

Un operazione del lettore di nastri comincia col caricare il giusto valore nel campo **COMMAND**. Per tutta la durata dell'operazione lo **STATUS** del dispositivo è "Device Busy".

Al completamento dell'operazione, un interrupt viene sollevato e il codice di stato appropriato viene settato; "Device Ready" è il codice di stato nel caso sia andato tutto bene, o uno dei sette dei codici di errore altrimenti. L'interrupt è poi riconosciuto mediante l'emissione di un ACK o di un RESET command.

Le prestazioni del lettore di nastri, siccome le operazioni di lettura son basate su DMA, dipendono fortemente dalla velocità di clock del sistema. La lettura di un nastro può variare in un range da 2 MB/sec quando la velocità di clock del processore è settata a 1 MHz, a più di 4 MB/sec quando la velocità di clock del processore supera i 99 MHz.

5.5. Network (Ethernet) Devices

Documentato nell'interfaccia network...

5.6 Printer Device (Stampanti)

uMPS supporta fino a 8 stampanti parallele, ognuna con una capacità di trasmissione di 1 singolo carattere da 8 bit.

Il campo **DATA0** per le stampanti è leggibile/scrivibile ed è usato per contenere il carattere da trasmettere alla stampante. Il carattere viene messo nel byte più basso del campo **DATA0**.

Il campo **DATA1** per le stampanti non viene utilizzato.

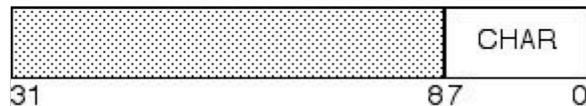


Figure 5.5: Printer Device **DATA0** Field

Il campo **STATUS** del registro della stampante è read-only e conterrà uno dei seguenti codici di stato:

Code	Status	Possible Reason for Code
0	Device Not Installed	Device not installed
1	Device Ready	Device waiting for a command
2	Illegal Operation Code Error	Device presented unknown command
3	Device Busy	Device executing a command
4	Print Error	Error during character transmission

Gli Status Code 1, 2 e 4 sono codici di completamento.

Il campo **COMMAND** del registro della stampante è leggibile/scrivibile e viene usato per dare comandi alla stampante.

CODE	COMMAND	OPERATION
0	RESET	Resetta il dispositivo
1	ACK	Riconoscimento di un interrupt pendente
2	PRINTCHR	Trasmette il carattere in DATA0 sulla linea di comando

Una stampante si avvia caricando il valore appropriato nel campo **COMMAND**.

Per tutta la durata dell'operazione, lo stato della stampante rimane "Device Busy".

Al completamento dell'operazione un interrupt viene sollevato e un appropriato status code viene settato nel campo STATUS: "Device Ready" se tutto è filato liscio, altrimenti un codice di errore.

L'interrupt viene riconosciuto rilasciando un ACK o un RESET command.

La stampante ha un throughput massimo di 125 KB/sec.

5.7 Terminal Devices (Terminali)

uMPS supporta fino a 8 terminali, ognuno dei quali avente una capacità di ricezione e di trasmissione di un singolo carattere da 8 bit.

Ogni terminale contiene 2 sotto-dispositivi: un trasmettitore e un ricevente.

Questi 2 dispositivi operano indipendentemente e concorrentemente. Per supportare questi 2 sotto-dispositivi, il registro del dispositivo terminale è stato ridefinito così:

Field #	Address	Field Name
0	(base) + 0x0	RECV_STATUS
1	(base) + 0x4	RECV_COMMAND
2	(base) + 0x8	TRANSM_STATUS
3	(base) + 0xc	TRANSM_COMMAND

Il TRANSM_STATUS e il RECV_STATUS (campi 0 e 2) sono read-only e hanno il seguente formato:

	TRANS'D CHAR	TRANS STATUS
31	1615	87 0
	RECV'D CHAR	RECV STATUS
31	1615	87 0

Figure 5.6: Terminal Device **TRANSM_STATUS** and **RECV_STATUS** Fields

Lo status byte ha i seguenti significati:

Code	RECV_STATUS	TRANSM_STATUS
0	Device Not Installed	Device not installed
1	Device Ready	Device Ready
2	Illegal Operation Code Error	Illegal Operation Code Error
3	Device Busy	Device Bust
4	Receive Error	Transmission Error
5	Character Received	Character Transmitted

I significati degli status code da 0 a 4 sono uguali a quelli di un lettore di nastri.

Inoltre:

- Il Character Received code (5) è settato quando un carattere è stato correttamente ricevuto dal terminale ed è stato posizionato in **RECV_STATUS.RECV'D-CHAR**.
- Il Character Trasmitted code (5) è settato quando un carattere è stato correttamente trasmesso al terminale ed è stato posizionato in **TRANSM_STATUS.TRANS'D-CHAR**.
- Il Device Ready Code (1) è settato in risposta al comando **ACK** o al **RESET**

I campi **TRANSM_COMMAND** e **RECV_COMMAND** sono leggibili e scrivibili e vengono usati per dare comandi al terminale.

CODE	TRANSM COMMAND	RECV COMMAND	OPERATION
0	RESET	RESET	Resetta il trasmettitore o il ricevitore
1	ACK	ACK	Riconoscimento di un interrupt pendente
2	TRASMITCHAR	RECEIVECHAR	Trasmette o Riceve il carattere a linea di comando

I campi **TRASM_COMMAND** e **RECV_COMMAND** hanno il seguente formato:

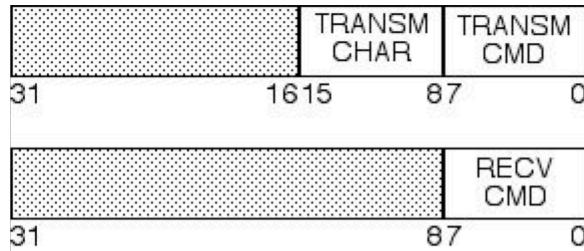


Figure 5.7: Terminal Device **TRANSM_COMMAND** and **RECV_COMMAND** Fields

RECV_COMMAND.RECV-CMD è semplicemente il comando. Il campo **TRANSM_COMMAND** ha 2 parti; il comando in se stesso (**TRANSM_COMMAND.TRANSM-CMD**) e il carattere da trasmettere (**TRANSM_COMMAND.TRANSM-CHAR**).

Un carattere viene ricevuto e, dopo che il ricevente ha rilasciato il comando **RECEIVECHAR**, posizionato dentro il **RECV_STATUS.RECV'D-CHAR**.

L'operazione di un dispositivo terminale è molto più complessa rispetto a quelle degli altri dispositivi, perchè esso ha 2 sotto-dispositivi che condividono la stessa interfaccia del dispositivo.

Quando un terminale genera un interrupt, il gestore degli interrupt del dispositivo terminale, dopo aver determinato quale terminale ha generato l'interrupt, deve inoltre determinare se l'interrupt è corrispondente a una ricezione di un carattere, o a una trasmissione di un carattere o a entrambi. Cioè 2 interrupt pendenti contemporaneamente.

Se ci sono 2 interruzioni pendenti contemporaneamente, entrambi devono essere riconosciuti al fine di disattivare (porre a 0) l'appropriato bit dell'interrupt pendente nella linea di interrupt 7 della Interrupting Devices Bit Map.

Per rendere possibile il determinare quale sotto-dispositivo ha interrupt pendenti, ci sono 2 condizioni "ready" per i sotto-dispositivi; Device Ready e Character Received/Trasmitted.

Mentre gli altri dispositivi usano lo stato "Device Ready" per segnalare che l'operazione è andata a buon fine, questo diventa insufficiente per i dispositivi terminale.

Per i dispositivi terminale si rende necessario distinguere tra lo stato di successo di un'operazione se l'interrupt non è stato ancora riconosciuto, Character Received/Trasmitted, e un comando il cui completamento è stato riconosciuto, Device Ready.

Un'operazione di un terminale comincia caricando l'appropriato valore/valori nei campi

TRANSM_COMMAND o **RECV_COMMAND**. Per tutta la durata dell'operazione lo **STATUS** dei sotto-dispositivi è "Device Busy". Al momento del completamento dell'operazione un interrupt viene sollevato e un appropriato status code viene settato in **TRANSM_STATUS** o **RECV_STATUS** rispettivamente; "Character Trasmitted / Received" se tutto è andato a buon fine, un error code se è successo qualcosa. L'interrupt viene riconosciuto rilasciando un comand ACK o un RESET al quale il sotto-dispositivo risponderà settando nel rispettivo campo **STATUS** il valore di "Device Ready".

Il Throughput massimo per un terminale è di 12.5 KB/sec per entrambi: Carattere trasmesso e carattere ricevuto.

Capitolo 6

Summary of ROM Services and Library Function: Sommario dei Servizi della ROM e Funzioni di Libreria

Come descritto nei capitoli 3 e 4, la ROM fornisce i servizi vitali per il sistema.

In particolare:

- Un sistema addizionale di inizializzazione del sistema al tempo di BOOT o di RESET

- Il ROM-Excpt Handler trasferisce la gestione delle eccezioni al SO salvando prima lo stato del processore al tempo dell'eccezione e caricando poi il nuovo stato del processore per eseguire l'effettiva gestione dell'eccezione. Il codice del ROM-Excpt Handler può essere trovato nel file EXEC.S
- IL ROM-TLB-Refill Handler trova l'indirizzo della PgTbl (PageTable) necessaria. Poi convalida la PgTbl indicata e, se valida, ricerca il corrispondente PTE.
Se la ricerca termina con una corrispondenza, la corrispondenza viene copiata dentro al TLB e il controllo ritorna al flusso di esecuzione interrotto, altrimenti, come per il ROM-Excpt handler, il ROM-TLB-Refill handler trasferisce la gestione della Bad-PgTbl o della PTE-MISS exception al livello superiore. Il codice della ROM-TLB-Refill Handler può essere trovato nel file EXEC.S

Il file EXEC.S con tutti gli altri files menzionati in questo capitolo (COREBOOT.S, TAPEBOOT.S, DISKBOOT.S, CRTOS.S e LIBUMPS.S) sono parte della distribuzione di uMPS.

La directory di installazione per questi file è: /USR/LOCAL/UMPS/SUPPORT/

6.1 Bootstrap ROM Functionality: Il funzionamento del BOOT della ROM

Ogniqualvolta uMPS si avvia o si resetta (Vedi capitolo 8), lo **Status** viene settato in modo tale da abilitare la **CP0 (Status.CU[0]=1)**, la VM viene disabilitata (**Status.VMc = 0**), le interruzioni vengono disabilitate (**Status.IEc=0**), il Bootstrap Exception Vector viene abilitato (**Status.BEV = 1**) e il sistema passa in Kernel-Mode (**Status.KUc = 0**). Es. **Status** = 0x1040.0000. Il **PC** viene settato al bootstrap Rom code (cioè a 0x1FC0.0000 , vedi capitolo 4.1) e **\$SP** viene settato a 0x0000.0000.

Il Bootstrap ROM, il cui codice si trova nel file COREBOOT.S o nel TAPEBOOT.S, prima disattiva il Bootstrap Exception Vector bit (**Status.BEV = 0**) e poi carica il S.O.

Se il S.O. deve essere presentato su nastro/tape (TAPEBOOT.S), il S.O. è letto dal dispositivo TAPE0 e caricato nella RAM. Se il S.O. deve essere presentato a uMPS in core (COREBOOT.S) non c'è niente da leggere o caricare, dato che il S.O. verrà già messo nella RAM prima dell'esecuzione del Boostrap ROM code. Questa seconda opzione non è comunque troppo realistica per lo sviluppo del S.O. a livello studentesco.

Infine il Bootstrap ROM code setta il **PC** all'indirizzo salvato in 0x2000.1004, l'indirizzo di `_start()`. La funzione `_start()`, il cui codice si trova nel file CRTSO.S, setta **\$SP** a RAMTOP e chiama `main()`. Se `main` ritorna, `_start()` termina chiamando **HALT**.

6.2 New ROM Service/Instructions: Le nuove istruzioni/servizi della ROM

In Aggiunta, il ROM code estende le istruzioni intere del MIPS R2/3000 con i seguenti servizi/istruzioni:

- **LSDT**: Carica automaticamente lo stato del processore con lo stato posizionato nella locazione di memoria fisica forniti. Questo servizio/istruzione richiede al processore di essere in Kernel-Mode, altrimenti solleva una Breakpoint Exception.

- **FORK**: Carica lo stato del processore con lo stato posizionato nella locazione di memoria fisica forniti. Questa istruzione NON E' COMPLETAMENTE ATOMICA (cioè non si svolge come 1 singola istruzione); solo il caricamento dell' **EntryHi**, del **Cause**, dello **Status** e del **PC** sono eseguiti atomicamente. Inoltre, lo stato del processore non è caricato completamente dalla locazione di memoria fisica fornita, infatti **EntryHi**, **Status** e **PC** vengono caricati da parametri addizionali forniti.

In aggiunta, i registri a0, a1 e a2 non sono caricati dalla memoria in quanto facenti già parte del nuovo stato del processore.

Infine, anche v0 non viene caricato dalla memoria e il suo valore, nel nuovo stato del processore, è indefinito.

Questo servizio/istruzione richiede al processore di essere in Kernel-Mode, altrimenti solleva una Breakpoint Exception.

- **PANIC:** Mostra il testo “Kernel Panic” sul terminale 0 e mette il processore in un loop infinito. Questo servizio/istruzione richiede al processore di essere in Kernel-Mode, altrimenti solleva una Breakpoint Exception.
- **HALT:** Mostra il testo “System Halted” sul terminale 0 e mette il processore in un loop infinito. Questo servizio/istruzione richiede al processore di essere in Kernel-Mode, altrimenti solleva una Breakpoint Exception.

6.2.1 ROM Actions Upon Loading a New Processor State: Azioni della ROM quando viene caricato un nuovo stato del processore

Questo è il lavoro del ROM-Excpt handler che carica il nuovo stato del processore; o come parte di un “passaggio all’insù” di una gestione dell’eccezione (il caricamento del nuovo stato del processore dall’appropriata New Area) o per una **LDST**. Ogniqualvolta il ROM-Excpt handler carica uno stato del processore, un’operazione di POP, come illustrato nella figura 6.1, viene eseguita sui **KU/IE** e **VM** stacks/pile.

Ci sono 2 operazioni di POP che agiscono come complemento delle operazioni di PUSH che vengono eseguite quando un’eccezione viene sollevata.

Notare come il “vecchio” valore nel 2 stacks rimane inalterato. (Vedi Capitolo 3.2)

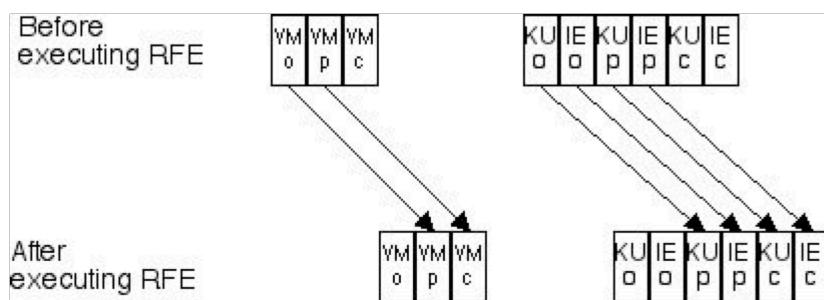


Figure 6.1: **VM** and **KU/IE** Stack Pop

Come quando un ROM-Excpt handler salva lo stato del processore, il caricamento dello stato del processore viene eseguito atomicamente.

Siccome non ci sono singole istruzioni assembly di uMPS che permettano il caricamento atomico dello stato del processore, il ROM-Excpt handler carica il nuovo stato del processore registro dopo registro con gli interrupt disabilitati.

Per finire, il caricamento del PC viene eseguito usando l’istruzione assembly di uMPS **RFE** (**R**eturn **F**rom **E**xception) che in aggiunta al caricamento del nuovo valore di **PC**, esegue anche un’operazione di POP sugli stacks/pile **KU/IE** e **VM**.

6.3 Accessing Machine Registers and Assembler Instructions in C: Accesso ai Registri Macchina e Istruzioni dell’Assemblatore in C

Nel processo di scrittura di un Sistema Operativo, si potrebbe aver bisogno di accedere ai vari registri della **CP0** (es. **Status**) e emettere particolari istruzioni-assemblatore per **CP0** (es. **TLB-CLR**).

Al fine di evitare la necessità di programmare in uMPS assembler, viene fornita una libreria C, *libumps*, per accedere ai registri di **CP0** e ai servizi/istruzioni della ROM-based estesa. Questa libreria è implementata nel file LIBUMPS.S ed è descritta dall’interfaccia LIBUMPS.E.

La libreria libumps definisce anche una routine/funzione, **STST**, che invece di fornire il contenuto di un singolo registro, salva il corrente stato del processore (vedi Sezione 3.2.1) nella locazione di memoria fisica fornитagli. **STST**, che NON E’ ATOMICA, non salva il corrente contenuto di **PC**. Viene scritto 0 nello stato salvato al posto di **PC**.

6.3.1 Accessing CP0 Instructions in C: Istruzioni di Accesso al CP0 in C

Tutte e 5 le istruzioni su CP0 sono invocabili, usando la libreria libumps, come normali funzioni void senza parametri. Le semantiche di queste chiamate sono descritte nella sezione 4.4. I comandi (TLBWI, TLBWR e TLBCLR) modificano il TLB, mentre i comandi Read e Probe modificano i registri **EntryHi**, **EntryLo** e l'**Index** CP0.

C usage	CP0 Instruction
void TLBWR()	TLB-Write-Random
void TLBWI()	TLB-Write-Index
void TLBR()	TLB-Read
void TLBP()	TLB-Probe
void TLBCLR()	TLB-Clear

Nota che il TLBR ha potenzialmente effetti pericolosi di alterazione del valore dell' **EntryHi.ASID**. Tutte e 5 queste istruzioni richiedono al processore o di essere in kernel-mode o, se in user-mode, di avere lo Status.CU[0] = 1 altrimenti solleverà una Coprocessor Unusable exception.

6.3.2 Accessing CP0 Registers in C: Accesso ai registri di CP0 in C

CP0 implementa 8 registri di controllo. 5 di questi registri sono leggibili/scrivibili, mentre gli altri 3 sono solamente leggibili.

Tutti gli 8 registri possono essere letti dalla libreria libumps come funzioni intere senza segno e senza parametri. In ogni caso il contenuto del registro specifico di CP0 è ritornato al chiamante come un unsigned int.

La funzione STST è differente dalle altre perchè è una funzione di tipo void e accetta un solo parametro, ovvero il puntatore allo stato del processore.

C usage	CP0 Register
unsigned int getINDEX()	Index
unsigned int getENTRYHI()	EntryHi
unsigned int getENTRYLO()	EntryLo
unsigned int getStatus()	Status
unsigned int getCause()	Cause
unsigned int getRandom()	Random
unsigned int getEPC()	EPC
unsigned int getBADVADDR()	BadVAddr
void STST(state_t *statep)	STST

I 5 registri scrivibili possono essere modificati, con la libreria libumps, attraverso funzioni tipo unsigned int che accettano come singolo parametro un insigned int.

Il parametro è il valore da caricare nel registro e il valore di ritorno è il valore nel registro dopo l'operazione di caricamento.

C usage	CP0 Register
unsigned int setINDEX(unsigned int)	Index
unsigned int setENTRYHI(unsigned int)	EntryHi
unsigned int setENTRYLO(unsigned int)	EntryLo
unsigned int setSTATUS(unsigned int)	Status
unsigned int setCAUSE(unsigned int)	Cause

Notare che **setENTRYHI** ha potenzialmente effetti pericolosi di alterazione del valore dell'**EntryHi.ASID**.

Tutti e 14 le istruzioni richiedono o che il processore sia in Kernel-Mode o, se in User-Mode, di avere lo **Status.CU[0]=1** altrimenti verrebbe sollevata una Coprocessor Unusable exception.

6.3.3 Accessing ROM-Implemented Services/Instructions in C: Accesso ai Servizi/Istruzioni implementati nella ROM

Tutti i servizi/istruzioni della ROM possono essere invocate tramite la libreria lipumps. Le semantiche di queste chiamate sono descritte qui sotto:

C usage	ROM Service/Instr.
void LDST(state_t *statep)	LDST
void FORK(unsigned int entryhi, unsigned int status, unsigned int pc, state_t *statep)	FORK
void PANIC()	PANIC
void HALT()	HALT

Tutti questi comandi richiedono che il processore sia in Kernel-Mode altrimenti solleveranno una Breakpoint exception.

Breakpoint Exception on Illegal ROM Service/Instruction: Breakpoint Exception in seguito a Servizi/Istruzioni illegali della ROM

I 4 servizi/istruzioni della ROM sono implementati usando una Breakpoint Exception; il codice assembly nella libumps contiene l'istruzione assembly **BREAK** che forza l'attivazione del meccanismo di gestione dell'eccezione. (Al registro **EPC** è assegnato il valore corrente del **PC**, a **Cause.ExcCode** è assegnato il codice indicante una Breakpoint exception (ovvero 9), gli stacks **KE/IE** e **VM** vengono “pushati”, e il ROM-Excpt handler viene invocato.) Se il ROM-Excpt handler non riconosce il codice in a0 o se Status.Kuc=1, la gestione dell'eccezione Breakpoint è “passata all'insù” nel modo usuale.

Quindi un tentativo di eseguire una LDST in User-Mode non causa la più intuitiva Reserved Instruction exception (LDST NON è un'istruzione dell'assemblatore uMPS), ma viene visto come la richiesta di un servizio/istruzione non riconosciuta dalla ROM che viene passata al livello superiore.

Capitolo 7

Programming and Compiling for uMPS

Programmare e Compilare per uMPS

Programmare per uMPS è facilitato dal Software Development Kit (SDK) che contiene:

- MISPEL-LINUX-GCC: un compilatore C; il gcc del MIPS R2/3000 cross-compiler
- MIPSEL-LINUX-AS: un assemblatore; il gcc del MIPS R2/3000 cross-assembler
- MIPSEL-LINUX-LD: un linker; il gcc del MIPS R2/3000 cross-linker
- UMPS-MKDEV; un utilità di creazione di dispositivi. Questa utility viene usata per creare i dispositivi disco di uMPS e per creare e caricare file su uMPS tape. Vedi Capitolo 8.8 per la descrizione di questa utility.
- UMPS-ELF2UMPS; un utility per la conversione in object file. Il compilatore genera file oggetto ELF. Gli ELF object files devono essere convertiti in uno dei 3 formati-oggetto riconosciuti da uMPS.
- UMPS-OBJDUMP e MIPSEL-LINUX-OBJDUMP; utility per l'analisi degli object file. MIPSEL-LINUX-OBJDUMP analizza l'ELF object files mentre UMPS-OBJDUMP è usato per analizzare i files oggetto che sono stati processati con l' UMPS-ELF2UMPS utility.

Usando l'SDK si può produrre codice per:

- Il Kernel/OS, es. Kaya
- Le 2 ROM exception handlers; ROM-Excpt handler, ROM-TLB-Refill handler e Bootstrap ROM routines/funzioni.
- Il programma utente (U-proc's) che il SO (es. Kaya) eseguirà

Inoltre, uno può programmare o in linguaggio C o in linguaggio uMPS assembler. Questa guida copre Kernel/OS e U-proc programmati usando C.

7.1 A Word About Endian-ness : Una parola su Endian-ness

A differenza della maggioranza delle architetture CPU, il MIPS R2/3000 supporta sia il big-endian che il little-endian, non simultaneamente, ma scelto da un pin settabile.

Nel caso di una HALFWORD (16 bit), il numero esadecimale 0x0123 verrà immagazzinato come:

Little endian		Big endian	
+----+----+		+----+----+	
0x23 0x01		0x01 0x23	
+----+----+		+----+----+	
byte: 0 1		0 1	

Nel caso di una WORD (32 bit), il numero esadecimale 0x01234567 verrà immagazzinato come:

Little endian				Big endian			
+----+----+----+----+		+----+----+----+----+		+----+----+----+----+		+----+----+----+----+	
0x67 0x45 0x23 0x01		0x01 0x23 0x45 0x67		+----+----+----+----+		+----+----+----+----+	
+----+----+----+----+		+----+----+----+----+		+----+----+----+----+		+----+----+----+----+	
byte: 0 1 2 3		0	1	2	3		

(Negli esempi il valore in **grassetto** è il byte più significativo)

Similarmente, uMPS supporta sia processi big-endian che little-endian; l'endian-ness di uMPS è quella della macchina su cui viene eseguito.

Come descritto nel Capitolo 8, indipendentemente dall'endian-ness della macchina ospite, la trace window's hexadecimal output è sempre mostrata nel formato big-endian mentre il window's ASCII output è sempre mostrato nel formato little-endian.

MIPSEL-LINUX-GCC, MIPSEL-LINUX-AS, MIPSEL-LINUX-LD e MIPSEL-LINUX-OBJDUMP sono nella versione little-endian; per essere eseguiti su macchine ospiti di tipo little-endian come i386.

Esiste un equivalente set di strumenti SDK eseguibili su macchine big-endian. Questi sono i loro nomi: MIPS-LINUX-GCC, MIPS-LINUX-AS, MIPS-LINUX-LD e MIPS-LINUX-OBJDUMP.

7.2 C Language Software Development: Software di sviluppo in linguaggio C

Programmando in C non è facile supportare l'incapsulazione e la protezione dei moduli e degli ADT. Il Capitolo 7.4 prova a dare una strategia per implementare l'incapsulazione in C.

Mentre l'ISO Standard per il C (C99) permetteva alle dichiarazioni di variabili e statements di essere liberamente mischiati e alla prima espressione in un ciclo for di essere una dichiarazione, queste aggiunte sintattiche non sono supportate dal compilatore di uMPS.

Come prima del C99 ISO Standard, tutte le variabili usate in una funzione devono essere dichiarate all'inizio della funzione.

Le funzioni a run-time della libreria C non sono disponibili.

Queste includono I/O Statements (es. printf da stdio.h), chiamate per l'allocazione di memoria (es. malloc) e metodi per la manipolazione di file.

In generale, qualsiasi metodo della libreria C che si interfaccia con il sistema operativo non è supportato; uMPS non è un sistema operativo che supporta queste chiamate, pertanto, se le si vuole utilizzare, bisognerà implementarle.

La libreria libumps, descritta al Capitolo 6.3, è la sola libreria supportata e disponibile.

Programmare in uMPS richiede un numero di convenzione per la struttura dei programmi e dei registri usati che devono essere seguite.

Molte di queste sono automaticamente fornite dal compilatore, ciò nonostante ce ne sono alcune che devono essere esplicitamente seguite.

- Il linker di uMPS richiede una piccola funzione: `_start()`. Questa funzione è l'entry point del programma da linkare. Tipicamente `_start()` inizializzerà qualche registro e chiamerà `main()`. Dopo che il `main()` ha concluso, il controllo ritorna a `_start()` il quale eseguirà qualche servizio di terminazione appropriato. 2 funzioni simili, scritte in assembler, sono fornite:
 - `CRTS0.O`: Questo file viene usato quando si collegano insieme i files per il kernel/OS. La versione di `_start()` in questo file assume che il programma (cioè il kernel) sia caricato nella RAM partendo da 0x2000.1000. Diversi registri sono inizializzati includendo lo stack pointer (`$SP`) che viene inizializzato a RAMTOP. (Gli stack in uMPS crescono "verso il basso" dal valore più alto della memoria a quello più basso. Quando il `main()` ritorna, `_start()` invoca l'istruzione **HALT** della ROM.)
 - `CRTI.O`: Questo file viene usato quando si devono linkare assieme più files per comporre il programma utente (U-proc's). La versione di `_start()` in questo file assume che il programma utente (U-proc's) parta dall'indirizzo (virtuale) 0x8000.0000. Diversi registri vengono inizializzati, ma non lo Stack Pointer (`$SP`), il quale sarà settato tipicamente alla fine del kUseg2. Quando `main()` ritorna, `_start()` carica `a0` con un valore significativo (es. 18) e invoca il servizio/istruzione **SYSCALL** della ROM.
- Il registro *Global Pointer (\$GP)*, richiede di puntare nel mezzo della struttura dati chiamata la *Global Offset Table (GOT)*. Il compilatore, generando la GOT e codice che li usa entrambi (`$GP` e GOT) posizionati nella sezione dati di un programma), può incrementare l'efficienza del linking e la velocità di esecuzione del codice risultante. `$GP` pertanto necessita di essere ricomputato con la chiamata di procedura.
Il registro generale `t9`, che per convenzione mantiene l'indirizzo iniziale di una procedura, è usato per questo scopo. Mentre il codice viene prodotto automaticamente dal compilatore, il programmatore del S.O. deve inizializzare il `t9` ogniqualvolta il valore di `PC` viene settato/inizializzato a una funzione. Pertanto, ogniqualvolta uno assegna un valore a `PC`, deve assegnare lo stesso valore anche al `t9` (ovvero `s_t9` definito in `TYPES.H`)

- Data la natura carica/salva dell'architetture uMPS e di MIPS R2/3000 su cui si basa, il codice generato dal cross-compiler può essere poco somigliante al codice sorgente. Questo capita specialmente quando uno attiva le ottimizzazioni del compilatore; cosa che non dovrebbe mai fare quando si programma in uMPS. Ciò nonostante, anche senza le ottimizzazioni abilitate, il compilatore si adopera per mantenere le variabili più utilizzate nei registri. Questo comportamento può presentare problemi, specialmente quando la locazione di memoria di una variabile è parte di un device register (o qualsiasi altra locazione hardware dipendente). Il compilatore può, in questo caso, muovere la variabile nel registro per velocizzare il codice. Qualsiasi modifica alla variabile originale sarà invisibile, in quanto ogni successivo riferimento alla variabile originale verrà sostituito con un riferimento a un registro – che non è stato aggiornato.
Per impedire questo comportamento anomalo, tutti gli accessi a locazioni hardware definite andrebbero fatto tramite puntatori “caching”; il valore del puntatore in un registro non sarebbe affatto dal comportamento anomalo. Se l'oggetto puntato dal puntatore può essere cambiato/aggiornato dall'hardware, il valore stesso del puntatore rimarrà costante.
E' probabilmente una buona idea fare un uso liberale di un modificatore/keyword `volatile`. Ogni variabile dichiarata come `volatile` non usa la cache in un registro per incrementare le prestazioni del codice. E' raccomandabile che tutte le variabili e le strutture importanti siano dichiarate come `volatile`. Questo dovrebbe includere tutte le strutture dati del kernel e della VM-I/O level. (Es. Semafori, PgTbl's, le strutture che descrivono lo swap pool, etc..)

7.3 The Compiling Process: Il Processo di Compilazione

Il cross-compile e il cross-linker generano codice nel formato ELF (Executable and Linking Format). Il formato ELF permette un'efficiente compilazione ed esecuzione da parte di un S.O che senza sarebbe troppo complesso.

Usando il formato ELF non è necessariamente complicato sviluppare un SO dal momento che non ci sono programmi da caricare o librerie disponibili prima di essere scritte.

Quindi uMPS usa 3 differenti formati di file oggetto:

- **.aout**: Basata sul predecessore del formato ELF, a.out, questo formato oggetto è usato per i programmi utente (U-proc's)
- **.core**: Una semplice variante del formato .aout usato per il formato oggetto del kernel/OS.
- **.rom**: Altra variante del formato .aout che viene usato come formato oggetto per i ROM exception handlers. Il formato .rom è utilizzato per il file object e non per i programmi eseguibili.

L'utility di conversione dei file object, `UMPS-ELF2UMPS` esegue la conversione necessaria di un file ELF o di un programma eseguibile nel suo equivalente .aout, .core o .rom object file/programma eseguibile.

7.3.1 The .aout Format

7.3.2 The .core Format

7.3.3 The .rom Format

7.3.4 Using the Compiler, Linker e Assembler: Usare il Compilatore, il Linker e l'Assemblatore

Il Compiler, l'assembler e il linker sono quegli strumenti di sviluppo “fuori dalla scatola”.

In quanto tali essi accattano una vasta gamma di argomenti/parametri a linea di comando.

Mentre un linker non richiede particolari flags(variabile binaria da porre a 0 o a 1), esso richiede un *linker script*. 2 Linker Script vengono forniti:

1) per produrre un file oggetto (`ELF32LTSMIP.H.UMPSAOUT.H`) che sarà eventualmente convertito nel formato .aout.

2) per produrre un file oggetto (`ELF32LTSMIP.H.UMPSCORE.X`) che sarà eventualmente convertito nel formato .core.

Per i curiosi; questo è il modo, usando lo stesso compilatore, con cui uno può generare un object file per il Kernel/OS attraverso un indirizzo profilo e files object per i U-proc's programs con un differente indirizzo profilo. (???)

Per chi sceglie di scrivere codice in uMPS assembler, es (ri)scrivendo una routine della ROM Bootstrap o alterando la `_start()` in `CRTI.S`, è necessario usare il `-KPIC` assembler flag. Questo flag forza la generazione di codice con posizione indipendente da parte dell'assemblatore.

7.3.5 Using The UMPS-ELF2UMPS Object File Conversion Utility: Usare l'utility di conversione UMPS-ELF2UMPS

L'utility `UMPS-ELF2UMPS` viene usata per convertire i file eseguibili e i file oggetto in formato ELF prodotti dall'SDK (gcc cross-platform development tool) in file formattati e richiesti da uMPS (.aout, .core, .rom).

```
UMPS-ELF2UMPS [-V] [-M] {-K | -B | -A} < file >
```

dove

- **file**: è il file eseguibile o il file oggetto dal convertire
- `-V`: flag opzionale che indica la volontà di avere un output verboso(cioè che mostra ogni singola operazione compiuta) durante la conversione.
- `-M`: flag opzionale che indica la generazione del file contenente la tabella dei simboli `.stab` associata a `<file>`.
- `-K`: flag che produce un file formattato `.core`. Questo flag può essere usato solamente con i file eseguibili. Lo `.stab` file viene automaticamente prodotto con questa opzione.
- `-B`: flag che produce un file formattato `.rom`. Questo flag può essere usato solamente con i file oggetto che non contengono rilocazioni.
- `-A`: flag che produce un file formattato `.aout`. Questo flag può essere usato solamente con i file eseguibili.

Una conversione avvenuta con successo produce un file chiamato **file.core.umps**, **file.rom.umps** o **file.aout.umps** di conseguenza.

Uno `.stab` file è un file di testo contentente una linea dell'uMPS specific header e il contenuto della tabella dei simboli del file, in formato ELF, dato in input.

Viene usato da uMPS per simulare la mappatura delle locazioni di `.text` e `.data` ai loro simbolici, cioè Kernel/OS codice sorgente, nomi. Quindi la generazione automatica del file `.stab` avviene ogniqualvolta un `.core` file viene prodotto. Visto che i `.stab` file sono semplici file di testo, è possibile modificarli anche attraverso un semplicissimo editor di testo.

Oltre alla sua utilità nel rintracciare gli errori nel programma `UMPS-ELF2UMPS` (che speriamo non esistano più), il `-V` flag è di interesse generale perché mostra quali sezioni dell'ELF sono state trovate e prodotte e la conseguente header data per `.core` e `.aout` files. Per i `.rom`, il `-V` flag mostra solamente la dimensione della ROM ottenuta durante la conversione del file.

7.3.6 Using The UMPS-OBJDUMP Object File Analysis Utility: Usare l' UMPS-OBJDUMP utility per l'analisi dei file oggetto

L'Utility UMPS-OBJDUMP viene usata per analizzare i file oggetto creati da UMPS-ELF2UMPS. Questa utility fornisce le stesse funzioni del MISPEL-LINUX-OBJDUMP (o MIPS-LINUX-OBJDUMP) che è incluso nel SDK. UMPS-OBJDUMP viene usato per analizzare i file oggetto .core, .rom e .aout mentre il MIPSEL-LINUX-OBJDUMP viene usato per analizzare i file oggetto formattati in ELF.

```
UMPS-OBGDUMP [-H] [-D] [-X] [-B] [-A] < file.mps >
```

dove

- **file.mps**: è il file_obj.core, file_obj.rom o file_obj.aout che si vuole analizzare
- -H: flag opzionale per mostrare l'header del programma .aout, se presente
- -D: flag opzionale che disassembلا e mostra la **.text** area nel **file.mps**.
- -X: flag opzionale che produce una word esadecimale completa formattata in little-endian partendo dal **file.mps**. I blocchi con soli 0 vengono saltati e marcati con *asterischi*. L'output apparirà identico indipendentemente da come si trova il **file.mps** (little-endian o big-endian)
- -B: flag opzionale che produce un byte dump del file.mps. I blocchi con soli 0 vengono saltati e marcati con *asterischi*. Diversamente dalla -X flag, il formato endian-ness per il suo output dipende dal formato dell'endian-ness del **file.mps**. Se il file.mps è big-endian, allora anche l'output prodotto con -B sarà big-endian.
- -A: flag che esegue tutte le operazioni descritte sopra.

L'output uscente da UMPS-OBJDUMP è diretto all'stdout.

7.4 Encapsulation Strategy for C Programming: Strategia di Incapsulazione per la programmazione in C.

Si prevede che il tuo sistema operativo sarà implementato in C (e non in C++ o Java).

Mentre C non è un linguaggio orientato agli oggetti, sei fortemente incoraggiato a dividere il tuo codice in moduli e cercare di trarne vantaggio, al meglio possibile, dell'incapsulazione.

Sei fortemente incoraggiato a creare i+1 (o anche i+2) sottodirectory nella tua home. I di queste directory dovranno contenere il codice (" *.c " files) per ognuna delle fasi (phase) che dovrai implementare, una cartella (/H) per i *.h (header files) e una cartella opzionale (/E) per gli external file, i *.e (extern files).

Invece di mettere tutti i file.e in una sola directory, tu puoi, opzionalmente, mettere ogni ".e" nella sua corrispondente cartella della sua determinata phase.

La distribuzione uMPS contiene 2 files che definiscono alcune costanti relative all'hardware, CONST.H e tipi relativi all'hardware, TYPES.H. Questi file ti saranno molto utili.

Copia essi dentro la cartella degli header (/H) del tuo account e integrali se necessario.

7.4.1 Module Encapsulation in C

Sei incoraggiato ad adottare il seguente set di convenzioni per programmare in C. Queste convenzioni sono state elaborate in modo per fornire ai programmatore qualche beneficio delle classi e dell'incapsulazione.

Per esempio considera un file (o un modulo) che contiene tutte le funzioni relativi a uno specifico e ben definito scopo. Questo file contiene:

- “public” functions: funzioni che il programmatore vuole che siano visibili anche agli utenti esterni al modulo.
- “private” functions: funzioni di appoggio per altre funzioni; funzioni che il programmatore NON vuole che siano visibili agli utenti esterni al modulo.

- “**public**” global variables: variabili definite al di fuori di qualsiasi funzione individuale presenti nel file/modulo e che il programmatore vuole che siano visibili e utilizzabili anche dagli utenti esterni al modulo in cui sono dichiarate. Basta includere il file per usarne le variabili globali.
- “**private**” global variables: variabili definite al di fuori di qualsiasi funzione individuale presenti nel file/modulo e che il programmatore NON vuole che siano visibili e utilizzabili anche dagli utenti esterni al modulo in cui sono dichiarate. Anche includendo i moduli in cui sono presenti questo tipo di variabili, esse non possono essere utilizzate.
- “**persistent**” local variables (**static**): Variabili definite all’interno della particolare funzione (e quindi “privata”) ma, come per le variabili globali, hanno un tempo di vita uguale a quello del programma.

Componenti privati; funzioni e variabili possono essere dichiarate usando la keyword **static**. Un oggetto **static** è visibile per intero all’interno del file in cui viene dichiarato, ma completamente invisibile all’esterno. Crea effettivamente delle funzioni e delle variabili private. Una variabile “persistent” può venire anche dichiarata usando la keyword **static**. Qualsiasi variabile dichiarata dentro una funzione la cui dichiarazione è preceduta dalla keyword static, diventa persistent conservando il suo valore tra le chiamate alla funzione. Le variabili **static**, o persistent, non sono allocate sullo stack (come le variabili automatiche) ma nella stessa sezione usata per l’allocazione delle variabili globali. E’ una sfortuna che la keyword **static** sia sovraccaricata in C. Per aiutare a differenziare i suoi 2 usi, potrebbe essere utile creare un alias di **static** a HIDDEN.

```
#define HIDDEN static
```

Ora, i componenti privati possono essere dichiarati come HIDDEN mentre i componenti persistenti possono essere dichiarati come static.

Per ogni file/modulo ci dovrebbero anche essere dei file di dichiarazioni esterne (*.e). Questi file devono contenere i prototipi di ogni funzione pubblica e variabile globale. Ogni prototipo deve essere preceduto dalla keyword **extern**. Qualsiasi altro modulo che fa uso di una funzione pubblica o variabile globale di un altro modulo, dovrà #include il corrispondente file.e di quel modulo.

Per esempio:

```
#include "../e/asl.e"
```

Infine, le strutture globali (cioè typedef) e le costanti devono essere definite in appropriati files “.h”; es. CONST.H e TYPES.H.

Capitolo 9 Debugging in uMPS

Come descritto nel Capitolo 7.2, scrivere codice per un S.O. richiede qualche speciale considerazione. Il debugging di un S.O., sfortunatamente, è addirittura più impegnativo. Secondo l’esperienza dell’autore, più di uno studente universitario, anche quando vengono forniti sofisticati strumenti di debugging, fate principalmente affidamento sull’output (es. cout o printf) per il debugging.

Esaminando l’output generato, lo studente può dedurre sia il flusso di esecuzione che lo stato del programma.

Questo può essere chiamato: “debugging dai side-effect”.

Quando debuggiamo un S.O. non ci sono supporti per l’output; almeno non finché l’autore del S.O. non scrive il supporto per essi.

Debuggare un S.O. è ulteriormente complicato per la sua interconnessione intrinseca; frustrante è il desiderio di fornire un’unità di testing. Uno non può testare uno scheduler senza il supporto per le tempistiche. Uno non può testare il servizio di tempistica senza il supporto per la gestione delle

interruzioni. Uno non può testare la gestione delle interruzioni senza il supporto per i semafori e uno scheduler.

La mancanza di uno strumento tradizionale per il debug, di un output e l'impossibilità di fare test sui moduli a causa di un S.O. interconnesso è per lo studente una sfida fare il debug.

E' importante partire pensando al debugging, non in termini di side-effect, ma in termini di corrente stato del programma. A differenza dei tradizionali progetti di programmazione universitari, dove è possibile testare tutti i possibili percorsi di controllo e tutti gli stati significativi del programma, qui ci sono troppi stati significativi di un programma durante l'esecuzione di un S.O. per poter fare un testing esaustivo; almeno entro i limiti di un progetto universitario.

Ciò nonostante, tramite il debugging, con particolare enfasi sullo stato del programma, invece che sui side-effect (effetti collaterali), uno può cominciare a guadagnare un certo grado di confidenza con la correttezza del S.O.

9.1 uMPS Debugging Strategies: Strategie di debug in uMPS

Il simulatore uMPS, da una prospettiva, può essere pensato come un sofisticato strumento/ambiente di debugging. Come descritto al Capitolo 8, esso fornisce 3 primi meccanismi di assistenza nel processo di debug; breakpoints, intervalli sospetti e tracciamento della memoria. La seguente è la descrizione di 2 strategie di debugging.

9.1.1 Using a Character Buffer to Mimic `printf`: Usare un buffer di caratteri per “mimare” la `printf`

E' possibile usare un buffer RAM come un flusso output; consentendo di utilizzare la “familiare” tecnica di debug.

Per fare questo uno dichiara un array globale di caratteri e invece che rilasciare l'output, uno muove un carattere o un valore significativo dentro al buffer. Le tracce (?) sono poi usate per mostrare il contenuto del buffer. Eseguire il S.O. mentre si monitora il contenuto di un buffer è isomorfico alla gestione di un tradizionale programma di monitoraggio e del flusso di output (??).

Scrivere sul buffer può essere fatto in modo cumulativo, come un flusso di output, o ogni linea di uscita può sovrascrivere il precedente.

In uMPS uno ha la possibilità di migliorare questo approccio mettendo il buffer in una lista sospetta e abilitando il simulatore a fermare il sospettato.

Ora ogniqualvolta un output viene ricevuto dal simulatore, esso si fermerà per l'esaminazione, tramite la finestra di traccia, dello stato delle variabili del S.O.

9.1.2 Implementing Debugging Functions: Implementare le funzioni di debug.

L'approccio soprastante, benchè utile, ha le sue limitazioni. Non ci sono funzioni itoa (intero to ascii), a meno che tu non le scriva di tuo pugno, così uno è obbligato, attraverso il buffer globale, a mostrare solo stringhe di caratteri.

Inoltre, in un programma che può essere interrotto prima di ogni suo output si possono esaminare soltanto le variabili globali dalla finestra che tiene traccia dell'esecuzione.

Un miglioramente a questo approccio consisterebbe nell'implementare o una funzione di debug, o una collezione di funzioni simili; es. debugA, debugB, debugC, Ogni funzione può essere definita in modo che prenda in input 4 parametri interi. Ora, invece di generare e stampare una stringa tipo “tu sei qui”, nel punto in cui si vuole esaminare lo stato del programma si chiamerà la funzione di debug.

In questo scenario il primo parametro è di solito un valore chiave (es. 10, 20, 42, ...) che identifica univocamente dove la chiamata di funzione si trovi nel programma. Gli altri 3 parametri possono essere usati per passare variabili locali alle funzioni, variabili globali, espressioni o ogni altro valore che sarà utile al debugger per capire lo stato del programma a quel punto dell'esecuzione del programma stesso.

Aggiungendo un breakpoint per ogni funzione di debug (e abilitando il simulatore a fermarsi sui breakpoints), il simulatore si fermerà in entrata ad esse.

Inoltre, i registri **a0**, **a1**, **a2** e **a3** conterranno i 4 parametri passati alla funzione di debug. I contenuti di questi registri sono sempre mostrati sulla Finestra Principale (Main Window) del simulatore uMPS eliminando la necessità di utilizzare la trace windows per mostrare le informazioni di stato del S.O. Inoltre, a differenza della piccola trace window che mostra sempre tutti gli intervalli di memoria tracciati, con una funzione di debug uno può scegliere quali variabili ispezionare su una chiamata chiamando quella base.

Verp è che uno è limitato a solamente 3 valori (a1, a2, a3), ma la trace window è ancora disponibile per visualizzare informazioni aggiuntive.

Usando una collezione di funzioni di debug si permette un maggiore grado di sofisticatezza del debugging.

???

Per esempio: debugA può essere usato per schedulare i problemi, debugB può essere usato per la gestione degli interrupt.

Uno non desidera andare oltre gli n breakpoints relativi alla schedulazione mentre cerca di raggiungere il breakpoint relativo alla gestione dell'interrupt; ma consentire solo il debugB breakpoint. Una collezione di funzioni di debug può aiutare nel seguente scenario: uno sospetta che la Ready Queue (coda dei processi Pronti) sia in qualche modo corrotta/danneggiata, ma solamente dopo il primo "warm" page fault.

Abilitando una funzione di debug, detto dallo scheduler, è inefficiente.

Ci saranno centinaia di scheduler breakpoints che avverranno prima di quello in questione.

Abilita invece una diversa funzione di debug nel pager. Quando quel breakpoint avviene, allora si abilita la funzione di debug nello scheduler.

Così si ha la possibilità di abilitare un breakpoint dove si verificano frequentemente...

???

9.2 Common Pitfalls to Watch Out For

Nonostante l'autore di un SO sembra generare i propri errori e quelli derivanti dal debug, un certo numero di errori avviene con regolarità.

La lista che seguirà presenta le maggiori difficoltà nella creazione di un SO.

Enumerandoli qui, speriamo di risparmiare a qualche fortunato autore di SO da lunghe e frustranti sessioni di debugging. Controllate il sito di uMPS per le possibili aggiunte a questa lista.

9.2.1 Error in Syntax: Errori di sintassi

Non c'è molto da fare per un errore logico, basta trovarlo e correggerlo.

Tuttavia, qualche volta la logica appare impeccabile, ma il codice non lavora ancora come ci si aspetta.

Questo può essere dovuto ad un errore di sintassi. Qualche struttura di un SO può essere complessa; un array di strutture, dove ogni struttura contiene a sua volta degli array di stati di processore, ognuno dei quali a sua volta contiene un array, l'array del PTE e altri dati, e tutti acceduti tramite un puntatore. Anche se la sintassi, usata per accedere a qualche valore in profondità della struttura, può compilare ed essere eseguita, essa potrebbe comunque non essere corretta.

E' raccomandato verificare la corretta sintassi delle funzioni di debug usate per mostrare i valori appropriati e situati nelle profondità delle strutture dati.

Anche i programmati più esperti possono commettere errori di sintassi quando si mischiano assieme strutture, array, strutture di array, array di strutture, notazioni col punto e puntatori.

9.2.2 Errors in Structure Initialization: Errori nell'inizializzazione delle strutture dati

Errare nell'inizializzare le strutture è un errore piuttosto comune. Molti programmati sono cresciuti usando ambienti dove non inizializzare una variabile voleva dire lasciare che il compilatore la inizializzasse a 0 di default.

Questa realtà è vera anche per uMPS; la .bss area per i files.core è esplicitamente incluso nel file .core e azzerato.

Mentre i valori iniziali per le strutture e le variabili della .bss sono a 0, molte di queste strutture vengono usate e riusate più e più volte. Il mantenimento dei processi bloccati da parte del Kernel ne sono un esempio canonico.

E' IMPORTANTE RICORDARSI DI INIZIALIZZARE TUTTI I CAMPI DI STRUTTURE SIMILI PRIMA DI RIUTILIZZARLE, ALTRIMENTI SI POTREBBERO AVERE VALORI INDESIDERATI.

9.2.3 Overlapping Stack Spaces and Other Program Components: Sovrapposizionamento di stack e altri componenti dei programmi.

I dati del SO per un U-proc's devono essere separati dai dati del SO per un altro U-proc's.

Questo è piuttosto facile rispetto a ogni spazio di indirizzamento virtuale U-proc's con la magica memoria virtuale.

Le strutture del SO che risiedono in ksegOS per ogni U-proc sono diversi tra loro.

Pertanto bisogna fare attenzione che le strutture dati del SO per ogni U-proc (che possono comprendere più aree dello stack in aggiunta alla PgTbl) siano abbastanza grandi e completamente disgiunte.

Data la difficile natura della sovrapposizione degli stack durante il debug, è raccomandabile che questo sia considerato ognqualvolta un SO si comporti in modo inaspettato e erroneamente.

9.2.4 Compiler Anomalies: Anomalie del Compilatore

Come spiegato nel Capitolo 7.2, il compilatore, anche se istruito a conservare il più possibile, riordinerà il codice e la cache delle variabili usate più di frequente.

Questo è particolarmente pericoloso quando si tratta di locazioni hardware definite – che per ragioni di sicurezza del compilatore dovrebbe essere sempre accessibile tramite puntatori.

Non esiste una via per determinare se il codice corretto è stato modificato in codice scorretto dal compilatore attraverso le funzioni di debug.

Specialmente quando il codice si esegue correttamente, facendo uso di funzioni di debug, e non si esegue più correttamente una volta tolte le funzioni di debug; è probabilmente il codice che il compilatore riordina o il caching delle variabili il problema.

Come uno può immaginare è frustrante per lo studente credere di aver completato con successo una phase i del loro OS project con le funzioni di debug e poi scoprire che senza non funziona più allo stesso modo.

Un compilatore non può riordinare gli statements dell'assemblatore che avvengono dopo la chiamata di funzione di prima o viceversa.

Inoltre qualsiasi registro / variabile nella cache deve essere ripristinato nella memoria prima della chiamata di funzione.

Una chiamata di funzione forza il compilatore, indipendentemente dalle ottimizzazioni che fornisce, a sincronizzare il codice generato con il codice sorgente originale.

Ci sono un certo numero di rimedi che uno può provare in questi casi:

- Fare niente. Le chiamate di funzione di debug aggiuntive rallentano semplicemente il SO, ma non hanno effetti sulla sua correttezza.
- Provare tutte le opzioni descritte nel Capitolo 7.2. Questo richiede l'uso di puntatori a locazioni hardware ben definite e l'uso della keyword `volatile` sulle strutture dati e variabili appropriate.