La Gestione Software dei Dispositivi di I/O

Appunti a cura di

Proff. Danilo Bruschi & Emilia Rosti

Anno Accademico 96/97

1 Premessa

Questi appunti contengono parte del materiale trattato nell'ultima parte del corso di Architettura degli Elaboratori I tenuto dai Proff. D. Bruschi ed E. Rosti nell'anno accademico 96/97, presso il Dipartimento di Scienze dell'Informazione di Milano.

L'obiettivo è quello di integrare il capitolo 8 del libro di testo per quanto riguarda la problematica inerente la gestione dei dispositivi di I/O. Prerequisito essenziale per la comprensione di questi appunti è quindi lo studio degli argomenti riportati nel capitolo 8 del libro di testo.

La comprensione degli argomenti trattati inoltre è strettamente legata ad una fase di implementazione di codice e testing, vengono quindi fornite tutte le indicazioni per poter sperimentare, attraverso il simulatore SPIM, le varie tecniche illustrate. Lo studente è quindi invitato a svolgere tale tipo di sperimentazione, che ricordiamo può essere argomento d'esame.

2 Introduzione

Un dispositivo di I/O è costituito dal punto di vista fisico da due componenti:

1. il dispositivo fisico effettivo (e.g., stampante, unità a dischi, mouse, tastiera, terminale video) e
2. una parte di "circuiteria" che gestisce tutte le operazioni che il dispositivo è in grado di svolgere; tale componente è nota come *device controller*.

Un device controller è in grado di gestire più periferiche, per semplicità però assumiamo che la corrispondenza tra device controller e periferica sia uno a uno. Il device controller è collegato attraverso il bus con i principali elementi del sistema (CPU e Memoria).

Ogni operazione di I/O eseguita all'interno di un programma usando le istruzioni `read,write`, richiede quindi l'attivazione di un certo numero di dispositivi (CPU, Memoria, Bus, Device Controller).

In particolare, un'istruzione di I/O espressa in un linguaggio ad alto livello (`read, write`), deve essere "decodificata" in una serie di comandi per il controller di I/O, che governa il dispostivo a cui l'operazione è indirizzata. Tale decodifica viene effettuata in via preliminare dal *compilatore*, che provvede a tradurre l'istruzione in questione con una chiamata al sistema operativo (syscall), che in fase di esecuzione (run-time) provvede a richiamare uno dei moduli del *Sistema Operativo* che si occupano della gestione dell'I/O, generalmente denominati *device driver*. Le funzioni che un device controller deve svolgere sono molteplici; noi analizzeremo solo la gestione delle operazioni di I/O.

In particolare ci soffermeremo a descrivere i principali meccanismi adottati per la comunicazione tra "CPU" e dispositivi periferici e conseguentemente le principali tecniche usate per l'esecuzione

delle principali operazioni di I/O. Riteniamo che al termine di questo ciclo di lezioni ogni studente possa autonomamente realizzare dei semplici device driver per il simulatore SPIM.

3 Come inviare comandi ad un device controller

La CPU comunica con i controller dei vari dispostitivi di I/O attraverso il bus. È sul bus che la CPU "carica" le richieste per l'effettuazione di operazioni di I/O ed è sempre attraverso il bus che i dati vengono trasferiti da periferica a memoria centrale o viceversa. In generale l'esecuzione di un'operazione di I/O può essere logicamente suddivisa in due fasi:

1. una fase in cui la CPU richiede alla periferica, più precisamente al controller della periferica, l'esecuzione di un'operazione di `read` o `write`;
2. una fase in cui i dati coinvolti nell'operazione devono essere trasferiti o dalla periferica alla memoria centrale, nel caso di operazione di read su periferica, o dalla memoria centrale alla periferica nel caso di operazione di write su periferica.

Entrambe le fasi possono essere implementate usando tecniche diverse. Inizialmente ci occuperemo della prima fase.

Prima di procedere con la descrizione dell'implementazione della prima fase richiamiamo alcuni elementi sul funzionamento di un device controller. Come già detto, il device controller è un dispositivo che controlla e gestisce le periferiche di I/O. Per poter svolgere tale funzione usa una serie di registri in cui memorizza lo stato della periferica (idle, busy, down), il comando che la periferica sta eseguendo, i dati acquisiti o da trasmettere alla periferica. Il device controller può essere assimilato ad un processore con ridotte potenzialità, che esegue le operazioni cui è preposto previo *esplicito* comando della CPU e che quando non opera è in stato idle. Per richiedere l'esecuzione di un'operazione di I/O ad un controller, la CPU deve predisporre il contenuto dei registri del controller a valori predeterminati, e successivamente avviare il controller stesso. L'operazione di predisposizione del controller può essere svolta in due modi diversi:

- modalità *memory-mapped*,
- attraverso l'uso di istruzioni di I/O dedicate.

3.1 Memory-mapped I/O

Quando viene adottata questa modalità, i registri dei vari controller presenti nel sistema vengono considerati come locazioni di memoria centrale, pur essendo fisicamente localizzati all'interno dei device controller. Questo implica che a tali registri vengano assegnati indirizzi a 32 bit (oppure a 16 o 64, a seconda del tipo di architettura usata) e che per caricare dei valori in tali registri è sufficiente eseguire l'istruzione `sw rt,address` mentre per leggere i valori in esso contenuti useremo l'istruzione `lw rt,address`. Come abbiamo già avuto modo di studiare, l'esecuzione di una `sw`

richiede che la CPU carichi sul bus il tipo di operazione (`write`), l'indirizzo della locazione interessata all'operazione e i dati ad essa indirizzati. Sul bus possono quindi transitare sia operazioni indirizzate alla memoria centrale che ai controller di periferica. È allora necessario provvedere tali dispositivi di un meccanismo che consenta loro di riconoscere le transazioni ad essi indirizzate. Più precisamente, tutti i dispositivi sul bus "ascoltano" tutti i segnali in transito su di esso e si attivano nel momento in cui riconoscono sul "bus indirizzi" l'indirizzo di una propria locazione di memoria. Discorso analogo vale per l'istruzione `lw`.

NOTA Gli indirizzi riservati ai registri dei controller fanno in genere riferimento alla porzione di memoria riservata al sistema operativo e non accessibile ai programmi utente. Questo significa che i programmi utente non possono accedere direttamente alle locazioni in questione ma devono usare come tramite il sistema operativo.

3.2 Istruzioni dedicate

Un altro modo per consentire alla CPU di poter accedere ai registri dei controller è quello di inserire nell'instruction set istruzioni dedicate alla gestione dell'I/O. Queste istruzioni fanno riferimento in modo esplicito in qualche modo al dispositivo interessato dall'operazione di I/O. Questa informazione verrà sfruttata dalla CPU per caricare sul bus, durante la fase di esecuzione, il codice della periferica interessata e quindi per escludere tutti i dispositivi non interessati alla transazione.

4 I dispositivi memory-mapped di SPIM

Il simulatore SPIM mette a disposizione due periferiche, una di input e una di output, gestite con modalità memory-mapped. Tali dispositivi possono essere utilizzati solo usando la versione del simulatore che opera in ambiente UNIX vale a dire `spim` o `xspim`. Per poter usare queste periferiche ricordarsi di dare l'opzione `-mapped_io` nella riga di comando `xspim`.

4.1 Il dispositivo di input

Il dispositivo di input messo a disposizione da SPIM simula una tastiera ed è denominato *Receiver*. Il controller del receiver è fornito di 2 registri: un registro di *controllo* (*Receiver Control Register*) e un registro *dati* (*Receiver Data Register*). Siccome tale dispositivo viene gestito in modalità memory-mapped, a ciascuno dei suddetti registri è associato un indirizzo che dovrà essere utilizzato per accedere alle informazioni in essi contenute. In particolare, al registro di controllo è assegnato l'indirizzo `0xffff0000`, mentre l'indirizzo assegnato al registro dati è `0xffff0004`.

Il byte meno significativo del registro dati (*Receiver Data Register*) contiene l'ultimo carattere digitato sulla tastiera. Tutti gli altri bit contengono il valore 0. Il contenuto di questo registro *può solo essere letto dalla CPU* e il suo contenuto viene modificato dal controller ogni volta che si digita un nuovo carattere sulla tastiera. Lo studente si soffermi su quest'ultima affermazione. È infatti la prima volta, all'interno del corso di architetture, che viene incontrato un dispositivo che "opera

autonomamente" rispetto alla CPU, cioè che esegue delle operazioni, in questo caso modifica del contenuto di registro, senza coinvolgere in alcun modo la CPU.

Nel registro di controllo (*Receiver Control Register*) siamo per ora interessati al solo bit di posizione 0. Tale bit è denominato *Ready Bit*. Il Ready Bit vale inizialmente 0 e viene modificato da 0 a 1 dal *controller* del Receiver quando viene digitato un carattere, che a sua volta viene depositato nel registro dati. Il Ready Bit viene modificato da 1 a 0 quando un dato viene prelevato dal Receiver Data Register. Anche questo bit non può essere modificato dalla CPU ma esclusivamente dal controller della periferica. La CPU deve limitarsi a leggerne il contenuto.

4.2 Il dispositivo di output

SPIM mette a disposizione dell'utente anche un dispositivo di output che simula il comportamento di un terminale video, tale dispositivo è denominato *Transmitter*. Anche il controller del transmitter contiene 2 registri: un registro di *controllo* (*Transmitter Control*) e un registro *dati* (*Transmitter Data Register*). Gli indirizzi assegnati a questi registri sono rispettivamente `0xffff0008` e `0xffff000c`.

Nel registro di controllo siamo per ora interessati al solo bit di posizione 0. Tale bit è denominato *Ready Bit*. Questo bit vale 1 quando il dispositivo gestito dal controller è pronto a ricevere un carattere da visualizzare, vale 0 quando il dispositivo sta visualizzando il carattere precedentemente caricato. Anche questo bit non può essere modificato dalla CPU ma esclusivamente dal controller della periferica. La CPU deve limitarsi a leggerne il contenuto.

Consideriamo ora il registro dati. Quando il ready bit vale 1 e attraverso un'istruzione `sw` viene memorizzato un valore in questo registro, il contenuto del byte meno significativo del registro viene visualizzato. Inoltre, durante la fase di visualizzazione il controller provvede a porre il ready bit del registro di controllo al valore 0. Tale bit verrà posto a 1 non appena sarà terminata l'operazione di output, la cui durata è non trascurabile, dato che SPIM simula il ritardo generato dall'uso di periferiche. Eventuali operazioni di store effettuate sul registro dati mentre il ready bit vale 0 vengono completamente ignorate dal controller, cioè non vi sarà alcuna fase di visualizzazione successiva all'istruzione store. È quindi estremamente importante che le scritture di dati sul transmitter data register vengano effettuate solo quando il ready bit vale 1.

5 I/O a controllo da programma

Nelle precedenti sezioni ci siamo soffermati sulle modalità di comunicazione tra CPU e controller di I/O ed abbiamo illustrato come le stesse sono state implementate nel simulatore SPIM.

Scopo di questa sezione e delle successive è quello di descrivere le principali tecniche usate dai device driver, per controllare l'operato dei device controller e per gestire lo scambio dei dati tra controller e memoria centrale.

La prima tecnica che illustriamo è la più intuitiva ed è nota con il termine di *gestione a controllo da programma*. Questa tecnica prevede il completo coinvolgimento della CPU nella gestione e nel controllo dell'operazione richiesta. Vale a dire che dopo aver predisposto il controller all'esecuzione dell'operazione desiderata, la CPU interroga continuamente lo stesso per verificare che l'operazione richiesta sia stata eseguita e, ad operazione ultimata, provvederà a trasferire il dato interessato (nel caso di operazione di input) o ad eseguire una nuova istruzione. Si usa solitamente in microprocessori low end, quali quelli impiegati in sistemi embedded, le cui attività sono limitate al controllo di un ristretto numero di dispositivi, o in sistemi real time in cui i vincoli sui tempi di risposta impongono un controllo diretto delle periferiche da parte della CPU.

Vediamo più in dettaglio lo schema di un'operazione di ingresso effettuata da un programma in cui viene adottata la tecnica di controllo da programma per la gestione delle operazioni di input/output. In questo esempio ci rifacciamo in parte ai dispositivi I/O di SPIM.

```
BEGIN
a. Predisponi i registri del controller ad eseguire
   un'operazione di lettura;
b. WHILE ready-bit = 0 DO;
c. Carica i dati acquisiti;
END
```

Con la tecnica di gestione dell' I/O a controllo da programma la CPU è quindi coinvolta direttamente durante l'esecuzione dell'intera operazione di input a svolgere il ruolo di "controllore". Infatti, tramite l'istruzione (b) la CPU "segue" l'intera operazione ed è in grado di verificare immediatamente quando la stessa termina (ready-bit = 1) e può quindi procedere al trasferimento dei dati. Il ciclo svolto dalla CPU, in attesa che il ready-bit assuma un determinato valore è detto ciclo di **busy waiting** o anche **spin lock**.

Vediamo ora come il precedente pseudo-codice può essere trascritto nel caso in cui la periferica da gestire sia il Receiver del simulatore SPIM. In questo caso l'istruzione (a) non ha alcun riscontro poiché la gestione delle periferiche è stata estremamente semplificata da SPIM.

```
        .text
        .globl  main
main:   li   $t0,0xffff0000  # Carica in $t0 l'indirizzo del Rec. Cont. Reg.
        li   $t2,0xffff0004  # carica in $t1 l'indirizzo del Receiver Data Reg.
ciclo:  lw   $t1,0($t0)      #
        rem  $t1,$t1,2       # Verifica se ready-bit = 0
        beqz $t1,ciclo
        lb   $a0,0($t2)      # Carica in $a0 il carattere acquisito
```

Esercizio 1. Aggiungere al precedente esempio le istruzioni necessarie per visualizzare sul Transmitter il dato caricato in $a0$.

Esercizio 2. Scrivere un programma per SPIM che accetti in input stringhe lunghe al più 20 caratteri, e provveda alla stampa delle stesse.

Esercizio 3. Valutare il tempo medio che la CPU spende durante il ciclo di busy waiting nella gestione di un'operazione di I/O.

6 Polling

Il ricorso al busy-waiting per monitorare l'attività di una periferica è molto intuitivo anche se notevolmente dispendioso per la CPU, che spreca buona parte del proprio tempo proprio nel ciclo di busy-waiting. Si consideri a tal proposito il seguente esempio.

Esempio 1. Si consideri un sistema in cui il processore opera con un clock a 50 MHz. Tale sistema dispone di tre unità periferiche tutte gestite con la tecnica di controllo da programma. Tali periferiche sono una tastiera che opera a 0.01KB/sec, un floppy disk che opera a 50KB/sec ed un disco magnetico che lavora a 2000KB/sec. Si determini la percentuale del tempo in cui la CPU viene effettivamente usata durante il trasferimento di 4 byte da ciascuno dei suddetti dispositivi. Si supponga che le operazioni di trasferimento dati che vedono coinvolta la CPU richiedano complessivamente 20 cicli di clock per ogni byte.

Soluzione. Calcoliamo per prima cosa il tempo richiesto da ciascuna delle periferiche per eseguire l'operazione richiesta.

Per acquisire 4 byte la tastiera impiega:

$$\frac{4\text{byte}}{0.01\text{KB/sec}} = \frac{4\text{byte}}{10^{-2}2^{10}\text{byte/sec}} = \frac{4 \times 10^2\text{secondi}}{2^{10}} = 0.39 \text{ secondi.}$$

In questo periodo, 0.39 secondi, il processore potrebbe sfruttare

$$50 \times 10^6 \times 0.39 = 19,5 \times 10^6\text{cicli.}$$

Essendo impegnata però a controllare l'I/O dalla tastiera, la CPU ne usa solo 80 (20 cicli per ciascuno di 4 byte), quindi la percentuale di utilizzo della CPU durante il trasferimento di dati da tastiera a memoria è

$$\frac{80 \times 100}{19.5 \times 10^6} = 0.00041\%.$$

Lo studente completi da solo l'esercizio.

Come risulta dall'esempio, la CPU svolge lavoro utile (trasferendo i caratteri) solo per una frazione piccolissima del tempo in cui è attiva (eseguendo busy-waiting per monitorare la periferica). Il precedente esempio sottolinea un aspetto estremamente importante della tecnica di gestione a controllo da programma: tale tecnica non deve essere utilizzata per la gestione di periferiche lente

poiché forza la CPU ad operare alla stessa velocità delle periferiche e di conseguenza la potenza del processore viene "sprecata". Per contro, con periferiche veloci come i dischi, la tecnica a controllo da programma può essere adottata, riducendo notevolmente lo "spreco" del tempo di CPU, ma vedremo successivamente che anche in questo caso è conveniente adottare tecniche più specifiche.

Per porre rimedio all'inefficienza della gestione dell'I/O a controllo da programma, nel caso di periferiche lente, è stata introdotta la tecnica di *polling*. Il polling non può comunque essere considerato una nuova tecnica di gestione dell'I/O ma è semplicemente un raffinamento della tecnica di controllo da programma. Il polling sfrutta la seguente idea. Durante un ciclo di busy-waiting, invece di limitare la CPU al controllo di una sola periferica, estendiamo il controllo della CPU a tutte o parte delle periferiche di I/O gestite a controllo da programma, presenti nel sistema. Quando si individua una periferica che necessita di qualche intervento da parte del processore si provvede a soddisfarne le richieste e si riprende il ciclo di polling. Vediamo più in dettaglio lo schema di un'operazione di ingresso effettuata da un programma in cui viene adottata la tecnica di polling per la gestione delle operazioni di input/output. In questo esempio ci rifacciamo in parte ai dispositivi I/O di SPIM. A titolo esemplificativo assumiamo che le periferiche "testate" durante il ciclo di polling siano numerate con $1, \ldots, n$.

```
Leggi dato dalla periferica x;
BEGIN
a. Predisponi i registri del controller ad eseguire
   un'operazione di lettura;
b. IF ready-bit(periferica_1) = 1 THEN servi periferica_1;
   IF ready-bit(periferica_2) = 1 THEN servi periferica_2;
                    .
                    .
                    .
   IF ready-bit(periferica_x) = 1 THEN
                  servi periferica x e poi esci dal ciclo;
   IF ready-bit(periferica_x+1) = 1 THEN servi periferica_x+1;
                    .
   IF ready-bit(periferica_n) = 1 THEN servi periferica_n;
   GO TO b
END
```

Con il polling si cerca quindi di fare in modo che la CPU possa effettuare più lavoro utile. Lo schema da noi illustrato presenta alcuni problemi quali il privilegio delle periferiche con identificativo più basso, che verranno testate più frequentemente delle altre. A questo e ad altri problemi si può ovviare implementando schemi di servizio più equi (fair). In generale comunque, anche se il polling migliora lo sfruttamento di un processore rispetto alla tecnica a controllo da programma, i miglioramenti riscontrati sono molto lievi e consentono di concludere che il polling pur rimanendo una tecnica valida per la gestione di alcuni tipi di periferiche non consente di risolvere i problemi già esaminati nel caso del controllo da programma. Per la soluzione di questi problemi è stato necessario introdurre una tecnica innovativa basata sull'uso degli interrupt. I paragrafi successivi sono dedicati alla descrizione di tale tecnica.

7 La comunicazione tra S.O. e programma utente

Il sistema operativo è quello strato di software che si frappone tra la "bare machine" e i programmi utente. Tra i vari compiti del sistema operativo vi è quello di fornire all'utente una serie di routine che facilitano la gestione delle risorse del calcolatore e provvedono ad intervenire al verificarsi di eventi anomali, quali il guasto di un dispositivo, che si possono verificare durante l'esecuzione di un programma. Nella stesura di queste routine vengono solitamente usate particolari istruzioni dell'instruction set architecture dette *istruzioni privilegiate*, che possono essere eseguite *solo* quando il processore sta operando sotto il controllo del sistema operativo. Affinché ciò sia possibile, esiste all'interno del processore un bit, detto *bit di modo*, che consente al processore di verificare in ogni istante il tipo di istruzioni che gli è consentito eseguire. Ad esempio tale bit varrà 0 quando il processore non potrà eseguire le istruzioni previlegiate; in questo caso diciamo anche che il processore è in *modo utente*. Varrà 1 quando il processore oltre alle suddette istruzioni può eseguire tutte le istruzioni dell'instruction set, in questo caso diremo che il processore è in *kernel mode*. Durante l'esecuzione di un programma utente tale bit varrà inizialmente 0 e verrà posto a 1 ad ogni intervento da parte del sistema operativo.

Ma quando interviene il sistema operativo durante l'esecuzione di un programma utente? L'intervento del sistema operativo durante l'esecuzione di un programma può avvenire per uno dei seguenti motivi:

1. perché esplicitamente richiamato dall'utente;
2. per cercare di porre rimedio ad un errore generatosi durante l'esecuzione del programma utente;
3. per gestire richieste d'intervento da parte di dispositivi esterni al processore, tipicamente dispositivi di I/O.

1. Abbiamo già incontrato un modo con cui l'utente può esplicitamente richiamare il sistema operativo: le *system call* indicate da altri autori anche come *supervisor call* o *trap*.

2. Durante l'esecuzione di un programma possono verificarsi degli eventi anomali quali un overflow aritmetico, un'istruzione load/store con indirizzo errato, la chiamata di una system call con codice errato e così via. Tali eventi, detti anche eventi interni perché generati all'interno del processore, vengono solitamente chiamati *exception (eccezioni)*. In questo caso il sistema operativo deve intervenire per cercare in qualche modo di porre un rimedio.

3. Come abbiamo già visto, la gestione dei dispositivi di I/O è svolta dal sistema operativo attraverso i device driver. Ogni volta che un'operazione deve essere svolta su ciascuno dei suddetti dispositivi, il corrispondente device driver deve essere mandato in esecuzione. I dispositivi in questione comunicano al processore la necessità di essere serviti attraverso opportuni segnali detti *interrupt*. Quando l'esecuzione di un programma utente viene interrotta da uno degli eventi sopra citati, il programma deve essere momentaneamente sospeso, per fare in modo che le opportune routine di sistema operativo possano essere eseguite. Al termine dell'esecuzione di queste routine però, l'esecuzione del programma utente deve poter riprendere da dove era stata sospesa, è quindi neces-

sario, prima di procedere con l'esecuzione della routine di s.o., prendere i necessari provvedimenti che consentano di poter riprendere l'esecuzione del programma interrotto.

Tali provvedimenti variano leggermente a seconda che ci si trovi di fronte ad una syscall, ad un'eccezione o ad un interrupt. Rifacendoci al simulatore SPIM illustreremo nelle seguenti sezioni alcuni esempi di routine per la gestione dei suddetti eventi.

7.1 La gestione delle eccezioni in SPIM

Abbiamo già visto che un'eccezione è un evento anomalo che si verifica durante l'esecuzione di un programma utente. Le eccezioni che possiamo gestire con il simulatore SPIM sono un sottinsieme di tutte quelle effettivamente gestite da un processore MIPS e sono riportate nella seguente tabella:

0	Interrupt esterno
4	Indirizzo errato in una load
5	Indirizzo errato in una store
6	Errore sul bus durante il caricamento di un'istruzione
7	Errore sul bus in fase di trasferimento dati
8	Eccezione generata da syscall
9	Eccezione generata da breakpoint
10	Eccezione generata da istruzione riservata
12	Overflow aritmetico

Quando si verifica un'eccezione è necessario sospendere l'esecuzione del programma e provvedere a salvare un certo insieme di informazioni che consenta di riprendere in un istante successivo la normale esecuzione del programma che ha generato l'eccezione stessa.

Più precisamente, le azioni da eseguire al momento in cui si verifica un'eccezione sono:

1. salvare tutte le informazioni, che possono essere utilizzate dal sistema operativo per cercare di "riparare" il guasto, e consentire, in un tempo successivo, la ripresa dell'esecuzione del programma;
2. procedere con l'esecuzione della routine di sistema operativo per la gestione dell'eccezione, detta anche *routine di risposta*;
3. al termine del passo (2), o porre termine al programma interrotto o riprenderne l'esecuzione dal punto in cui era stata sospesa.

Lo studente è invitato a consultare il libro (sez. 5.6) per apprendere la tecnica utilizzata dai progettitisti di MIPS per individuare e segnalare il verificarsi di un'eccezione durante l'esecuzione di un programma.

Le operazioni svolte durante la fase (1), detta anche fase di salvataggio del contesto, variano da architettura ad architettura. Vediamo il caso del MIPS. In MIPS un insieme di registri della

CPU, denominato *coprocessore 0*, viene dedicato alla memorizzazione delle informazioni necessarie alla gestione delle eccezioni e, vedremo successivamente, anche degli interrupt. In particolare tale coprocessore contiene una serie di registri ai quali è possibile accedere attraverso le istruzioni `lwc0,swc0,mfc0,mtc0`. Tra questi registri quelli che possono essere manipolati direttamente attraverso SPIM sono:

- $8 (*bad vaddr register*) contiene l'indirizzo di memoria che ha provocato l'eccezione;
- $12 (*status register*) contiene una serie di informazioni necessarie alla gestione degli interrupt;
- $13 (*cause register*) contiene il codice dell'eccezione verificatasi;
- $14 (*Exception Program Counter*) contiene l'indirizzo dell'istruzione che ha provocato l'eccezione.

Al verificarsi di un'eccezione, il controllo *via microprogramma* provvede ad eseguire le seguenti operazioni per salvare nei registri suddetti le relative informazioni:

1. salvataggio in EPC del PC corrente -4;
2. salvataggio dello status register;
3. caricamento nei bit 2–5 di $13 del codice dell'eccezione verificatasi;
4. caricamento nel PC del valore `0x80000080`.

Sottolineiamo la necessaria e stretta sequenzialità con cui devono essere eseguite le operazioni (1) e (4). Il caricamento immediato nel PC dell'indirizzo della routine di risposta implicherebbe infatti la perdita del valore presente nel PC al verificarsi dell'eccezione e di conseguenza l'impossibilità di riprendere successivamente l'esecuzione del programma interrotto.

Quindi dopo aver provveduto al parziale salvataggio del contesto del programma fino a quel punto in esecuzione, il controller passa ad eseguire il programma presente in memoria a partire dalla locazione 0x80000080. Questo programma è genericamente chiamato *exception handler* e risiede nell'area di kernel cioè l'area di memoria riservata al sistema operativo.

Le informazioni salvate attraverso queste operazioni, se possono essere sufficienti per l'individuazione della anomalia che ha causato l'eccezione, non sono però sufficienti per garantire la ripresa del programma interrotto. Nel caso di MIPS, si assume che sia la routine di risposta a salvare le informaizoni mancanti. In altre architetture invece il salvataggio del contesto del programma interrotto viene effettuato completamente via hw.

Con il simulatore `spim` o `xspim` è possibile scrivere del codice in linguaggio assembly che viene caricato nell'area di kernel e quindi scrivere una versione personalizzata di un exception handler. Per sfruttare questa potenzialità del simulatore è necessario usare le seguenti direttive per l'assemblatore, `.ktext <indirizzo>` e `.kdata <indirizzo>`. La prima specifica al simulatore di inserire il testo che segue nell'area di kernel a partire dall'indirizzo specificato; la seconda invece deve precedere la dichiarazione di dati che si vogliono memorizzare nell'area kernel. Lo studente che usa il

simulatore SPIM in ambiente MS-DOS potrà esercitarsi alla stesura di codice kernel riscrivendo il contenuto del file `traphand.spim`.

Di seguito riportiamo un breve esempio di exception handler per una gestione personalizzata di un'eccezione di oveflow aritmetico.

```
      .ktext  0x80000080
      sw  $a0,save0      #salva il contenuto dei registri di CPU che
      sw  $v0,save1      #devono essere usati successivamente

      mfc0 $k0,$13       #carica in $k0 il registro con il codice
                         #dell'eccezione
      srl  $a0,$k0,2     #mette in $a0 il codice dell'eccezione
      li   $k1,12        #in $k1 il codice dell'eccezione di overflow
      andi $a0, $a0, 12  #considera solo i bit relativi al codice dell'eccezione
      bne  $a0,$k1,fine  #ignora interrupt o eccezioni diversi da overflow

      la   $a0,msg       #visualizza messaggio d'errore
      li   $v0,4
      syscall
      mfc0 $a0,$14       #carica l'indirizzo dell'istruzione che ha causato
                         #l'overflow in $a0 e visualizzalo
      li   $v0,1
      syscall

      lw   $v0,sys       #data la segnalazione di errore il programma viene
      sw   $v0,0($a0)    #forzato alla terminazione, non e' quindi necessario
      li   $v0,10        #provvedere al ripristino del suo stato al rientro
      jr   $a0           #dalla routine di gestione interrupt

fine: lw  $a0,save0      #ripristina il contenuto dei registri di CPU
      lw  $v0,save1      #utilizzati
      mfc0 $k1,$14       #in $k1 l'indirizzo di EPC
      addi $k1,$k1,4

      rfe
      jr   $k1           #rientra  nel programma interrotto a partire
                         #dall'istruzione successiva a quella che
                         #ha generato l'eccezione
      .kdata
save0: .word 0
save1: .word 0
sys:   .word 0x0000000c
msg:   .asciiz "rilevato overflow durante l'esecuzione dell'istruzione:"
```

Alcune considerazioni sulla suddetta routine sono d'obbligo.

(1) Prima di procedere con l'esecuzione di una routine di risposta, è necessario, se si vuole riprendere successivamente l'esecuzione del programma interrotto, provvedere al salvataggio dei registri che la routine di risposta deve utilizzare, terminare cioè il salvataggio del contesto del programma interrotto. Poiché non è possibile stabilire a priori quando verrà eseguita una routine di risposta ad un'eccezione, non è possibile stabilire delle convenzioni generali per il salvataggio dei registri. L'unica politica consigliata è la seguente: la routine stessa provveda al salvataggio di tutti i registri che utilizza nel corso della sua esecuzione e provveda al ripristino degli stessi al termine dell'esecuzione stessa (convenzione *callee saved*). Inoltre, non è opportuno utilizzare lo stack per il salvataggio dei registri poiché l'eccezione generata potrebbe anche essere dovuta ad un'errata gestione dello stack da parte dell'utente e quindi si rischierebbe di compromettere anche l'esecuzione della routine di risposta. Nell'esempio precedente si è preferito memorizzare i registri ($a0, $v0) in aree di memoria riservate al kernel.

(2) La tecnica usata nell'esempio per il salvataggio dei registri presenta un inconveniente, che illustreremo con un esempio. Si supponga che durante l'esecuzione di una routine di risposta a un'eccezione si verifichi un'altra eccezione. Verrebbe allora richiamata di nuovo la routine di risposta che nella fase di salvataggio di contesto andrebbe a sostituire il contesto del programma utente che per primo ha generato l'eccezione, eliminando in questo modo la possibilità di rientro al programma stesso. Per garantire la corretta esecuzione di tale procedura, è quindi necessario che durante l'esecuzione della stessa non si verifichi mai alcuna eccezione. La routine descritta nell'esempio precedente è cioè corretta solo se viene di volta in volta *completamente* eseguita e non è possibile avere più copie attive contemporaneamente della stessa routine. In questo caso si dice anche che la procedura è *non rientrante*. Per contro si dice che una procedura è *rientrante* quando è possibile avere più copie della stessa contemporaneamente attive, si pensi ad esempio alle procedure ricorsive. In generale, affinché una procedura sia rientrante, è necessario che siano verificate le seguenti condizioni:

- il testo della procedura deve rimanere invariato durante l'esecuzione dello stesso;
- deve esistere un'organizzazione dei dati che evita la sovrapposizione dei dati relativi a più copie attive della procedura; una soluzione spesso adottata per la soluzione di questo caso è quella di utilizzare un'organizzazione a stack.

8 Trap

Come abbiamo già avuto modo di dire, i trap sono il meccanismo con cui l'utente richiede esplicitamente l'intervento del sistema operativo all'interno di un suo programma. Contrariamente alle eccezioni, i trap sono esplicitamente richiesti dal programmatore, che può quindi provvedere, prima della chiamata della syscall, a fornire al sistema operativo le informazioni necessarie all'espletamento della funzione richiesta. La syscall deve comunque essere eseguita in modalità kernel. Prima di procedere con la sua esecuzione è quindi necessario provvedere a salvare tutte le informazioni necessarie per riprendere successivamente l'esecuzione del programma che ha chiesto l'intervento del sistema operativo. Non devono essere però salvate le informazioni necessarie all'individuazione dell'evento accaduto.

Nel caso particolare da noi considerato, nel caso di una system call rispetto ad un'eccezione non verranno svolte le seguenti azioni:

-caricamento nei bit 2–5 di $13 del codice dell'eccezione verificatasi,

-caricamento del registro $8 del coprocessore.

9 Interrupt

L'interrupt è un ulteriore meccanismo che consente di interrompere l'esecuzione di un programma utente al fine di eseguire una routine di sistema operativo. Nati per eliminare la necessità del polling software delle periferiche, sono ormai definiti nel modo più generale come "eventi (escudendo i branch) che alterano il flusso di esecuzione". Abitualmente, e nella maggior parte dei sistemi, gli interrupt sono legati a dispositivi esterni al processore[1] e servono per segnalare allo stesso il verificarsi di eventi che il processore non ha modo di controllare direttamente, cioè eventi esterni.

In generale, un processore è fornito di una o più linee per la segnalazione di interrupt (*interrupt request*). Un dispostitivo richiede un interrupt asserendo una di queste linee e mantenendo il segnale fintantoché non riceve un segnale di *interrupt acknowledge*. A sua volta, l'interrupt viene "sentito" dal controller che in genere prima del fetch di un'istruzione provvede a verificare se sulle linee di interrupt sia presente qualche segnale ed in questo caso avvia una procedura di context switch e chiamata a sistema operativo simile a quelle che abbiamo già visto nel caso di trap ed eccezioni.

Vanno però fatte le necessarie distinzioni. Contrariamente ad eccezioni e trap, l'interrupt non viene causato dall'istruzione attualmente in esecuzione, ma da un evento esterno generalmente scorrelato all'istruzione stessa. In particolare si dice che un segnale di interrupt è *asincrono* rispetto alle attività correnti della CPU. Per contro, trap ed eccezioni sono eventi *sincroni*.

L'asincronicità del segnale di interrupt rende impossibile sapere a priori quando verificarne l'eventuale presenza, se prima della fase di fetch dell'istruzione, subito dopo o durante l'esecuzione dell'istruzione. La routine di risposta all'interrupt, qualunque sia il momento scelto per sentire se ci sono interrupt pendenti, è sempre una routine di sistema operativo ed è quindi necessario operare un cambio di contesto da programma utente a sistema operativo prima della sua esecuzione. Nel caso particolare di MIPS, gli interrupt, una volta "sentiti", vengono gestiti come le eccezioni o i trap (ad eccezione dei dettagli sopra descritti). In particolare viene eseguita, come interrupt handler, la routine di sistema che si trova all'indirizzo `0x80000080`, cioè la stessa routine usata per i trap e la gestione delle eccezioni.

A questo punto va però fatta la seguente considerazione. Abbiamo già avuto modo di sottolineare come la routine di risposta alle eccezioni sia non rientrante. Nel caso delle eccezioni il problema è stato risolto facendo in modo che la routine di risposta ad un'eccezione non generasse a sua volta un'eccezione e quindi la routine di risposta potesse terminare prima di poter essere rieseguita.

[1] Alcuni autori chiamano interrupt qualunque tipo di interruzione, interna o esterna che sia, non distinguendo tra trap, eccezioni e interrupt come fatto qui.

Tutto ciò era possibile poiché sono note le situazioni che provocano il verificarsi di un'eccezione. Non è però quasi mai possibile individuare a priori il verificarsi di un interrupt poiché il dispositivo che lo genera è scorrelato dall'attività che la CPU esegue. Può quindi verificarsi un interrupt durante l'esecuzione di una routine di risposta ad un'eccezione, che comprometterebbe il funzionamento dell'intero meccanismo di gestione degli eventi eccezionali. Quando si verifica un interrupt durante l'esecuzione della routine di risposta ad un'eccezione o ad un interrupt precedente, si può procedere in due modi. In un caso, l'interrupt più recente, non viene servito fino a che l'esecuzione della routine di risposta all'interrupt corrente non è terminata. La coda degli interrupt pendenti è gestita in modo First In First Out (FIFO) e si parla di interrupt "accodati". Alternativamente, l'interrupt più recente interrompe l'esecuzione della routine di risposta all'interrupt corrente e si passa ad eseguire la risposta al nuovo interrupt (il più recente). La coda degli interrupt è gestita in modo Last In First Out (LIFO) e gli interrupt si dicono "annidati". La scelta tra i due casi dipende dalla natura dell'interrupt. Con interrupt esterni, entrambe le gestioni sono possibili e sicure, cioè garantiscono la corretta terminazione della risposta all'interrupt corrente prima che venga eseguita la routine di risposta al nuovo interrupt. Se l'interruzione nuova è di origine interna, verificatasi nell'esecuzione di una istruzione, la risposta alla nuova interruzione deve essere annidata. In questo caso l'interrupt handler corrente non può completare l'esecuzione in modo corretto se prima non viene elaborato il nuovo interrupt.

Bisogna quindi individuare dei meccanismi che consentano, in alcuni casi, di gestire gli interrupt accodati e in altri di gestire gli interrupt annidati. Nel caso del MIPS, ad esempio, deve essere possibile accodare il nuovo interrupt fino al termine dell'esecuzione della routine di risposta ad un'eccezione o a un trap o a un interrupt precedente. Per evitare la gestione degli interrupt annidati, la maggior parte dei processori disabilitano gli interrupt esterni e segnalano un errore o fermano l'esecuzione all'arrivo di quelli interni.

Esistono due meccanismi per la realizzazione di tale obiettivo: *interrupt-enable mask* o *maschere di interrupt* (questo è il meccanismo adottato in MIPS) e *interrupt-priority systems*.

Nel primo caso ad ogni possibile interrupt viene associato un *livello*. La maschera di interrupt è una sequenza di bit in cui ogni bit corrisponde ad un livello di interrupt. Gli interrupt di un certo livello possono essere serviti solo se il corrispondente bit della maschera vale 1. La maschera di interrupt fa parte delle informazioni che caratterizzano il contesto di un programma utente e viene quindi salvata prima di procedere con l'esecuzione di una routine di sistema operativo.

Nel secondo caso ad ogni tipo di interrupt viene associata una priorità, una priorità è anche associata ai vari stati del processore. Un interrupt viene servito nel momento in cui ha una priorità maggiore di quella associata allo stato attuale del processore.

Per gestire le interruzioni esterne SPIM mette a disposizione del programmatore le seguenti informazioni:

1. *exception code* del cause register, vale 0 nel caso di interrupt esterno;
2. *interrupt mask*, memorizzata nei bit 8–15 dello status register. Sono infatti previsti 8 diversi livelli di interrupt (5 interrupt hw e 3 sw), dei quali però SPIM gestisce solo quelli di livello 0 e 1. Il bit 8 della maschera di interrupt è relativo all'interrupt sw di livello 0, il bit 11 a quello

hw di livello 0 e così via. Un bit a 1 nella maschera di interrupt significa che gli interrupt a quel livello sono abilitati.

3. *interrupt enable bit*, questo bit è il bit 0 dello status register. Quando vale 0 tutti gli interrupt sono disabilitati, quando vale 1 invece sono abilitati tutti gli interrupt previsti dalla maschera di interrupt.

4. *pending interrupt*, è una maschera presente nei bit 8–15 del cause register per la gestione degli interrupt di livello hw. Il bit 8 di questa maschera vale 1 quando si è verificato un interrupt di livello 0 che non è ancora stato servito, il bit 9 vale 1 quando si è verificato un interrupt di livello 1 che non è ancora stato servito, e così via.

Al verificarsi di un interrupt le azioni intraprese da SPIM sono:

1. carica in EPC il valore PC−4;
2. salva il valore dello Status Register;
3. disabilita tutti gli interrupt, cioè pone a 0 il bit 0 dello status register;
4. carica nei bit 2–5 del registro $13 del coprocessore 0 il codice di interrupt esterno;
5. carica nella pending interrupt mask il codice associato all'interrupt verificatosi;
6. carica in PC il valore 0x80000080.

Al rientro dalla routine di risposta all'interrupt è possibile ripristinare il valore dello status register salvato al passo (2) utilizzando l'istruzione `rfe` (return from exception).

10 I/O interrupt driven

Dopo aver introdotto il meccanismo dell'interrupt, il prossimo passo da fare è quello di comprendere come questo meccanismo possa essere usato per rendere più efficienti le operazioni di input/output.

Abbiamo infatti avuto modo di constatare precedentemente come la gestione dei dispositivi di I/O attraverso le tecniche a controllo da programma e di polling provoca:

1. nel caso di periferiche lente (ad es. tastiere, stampanti, mouse) un eccessivo spreco di tempo di CPU, che viene forzata ad operare alla stessa velocità delle periferiche stesse spendendo la stragrande maggioranza dei suoi cicli di clock in busy waiting;
2. nel caso di periferiche veloci invece, la percentuale di utilizzo della CPU è decisamente significativa ma il lavoro svolto dalla CPU è esclusivamente limitato al trasferimento di dati dal controller alla memoria centrale, un tipo di attività molto al di sotto delle potenzialità di una CPU.

Attraverso gli interrupt vedremo che è possibile ovviare a questi problemi ed introdurre delle tecniche per la gestione dell'I/O che ne consentano il superamento.

L'idea di base è la seguente. Nelle tecniche per la gestione dell'I/O sinora viste, la fase più costosa, dal punto di vista del tempo di CPU sprecato, era il ciclo di busy waiting, utilizzato dalla CPU per individuare l'esatto istante in cui il controller termina di eseguire l'operazione che gli era stata richiesta. Un modo quindi per migliorare la gestione delle operazioni di I/O è quello di eliminare dalle stesse il ciclo di busy waiting. Per eliminare il ciclo di busy waiting è però necessario individuare un meccanismo alternativo che consenta alla CPU di poter intervenire quando il controller ha terminato l'operazione richiesta. Questo meccanismo è l'interrupt esterno, che viene usato nel seguente modo.

Inizialmente la **CPU** predispone il controller ad eseguire il comando richiesto da un programma utente (`read, write`), dopodiché si disinteressa completamente dello svolgimento dell'operazione da parte del controller stesso.

A sua volta il **controller**, ricevuto il comando, provvede ad eseguirlo, e ad operazione ultimata invia un segnale di interrupt alla CPU, che attraverso la routine di gestione interrupt provvederà a richiamare il driver per gestire l'evento di I/O comunicatole. Lo schema appena descritto è quello adottato per la gestione dell'I/O *interrupt driven*.

Tra il momento in cui termina l'invio del comando al controller e la ricezione dell'interrupt inviatole dal controller stesso, la CPU è assolutamente svincolata dall'operazione di I/O e può quindi dedicarsi ad altre attività, tipicamente l'esecuzione di un altro programma.

Osservazione Si noti che, adottando lo schema appena descritto, nello stesso istante di tempo possono essere in esecuzione all'interno di un calcolatore più attività. Infatti il controller sta eseguendo un I/O per un programma e la CPU può a sua volta eseguire un programma diverso.

10.1 Interrupt Driven I/O con SPIM

Vediamo quali sono le informazioni che ci vengono messe a disposizione da SPIM per sperimentare la tecnica di gestione dell'I/O interrupt driven. Una forma di gestione di I/O interrupt driven può essere realizzata in SPIM utilizzando, oltre alle informazioni per la gestione degli interrupt citate nella sezione precedente, i bit di posizione 1 dei registri di controllo dei dispositivi receiver e transmitter. Più precisamente entrambi questi bit, denominati *interrupt enable bit*, possono essere sia letti che scritti dalla CPU ed in particolare

- il bit 1 del receiver vale inizialmente 0. Se viene posto a 1, il receiver invia alla CPU un interrupt di *livello* 0 ogni volta che il suo ready bit va a 1 cioè quando il buffer del dispositivo contiene un nuovo carattere da trasferire;
- il bit 1 del transmitter vale inizialmente 1. Se viene posto a 1, il transmitter invia alla CPU un interrupt di *livello* 1 ogni volta che il suo ready bit va a 1, cioè quando il dispositivo è pronto a ricevere un carattere da visualizzare.

Vale la pena ricordare che, ovviamente, gli interrupt che i suddetti dispositivi di I/O inviano alla CPU potranno essere "sentiti" dalla stessa solo se alla maschera di interrupt ed al bit di interrupt enable dello status register è stato assegnato l'opportuno valore.

Vediamo ora com'è possibile realizzare la routine dell'interrupt driven I/O su SPIM, in particolare vediamo come può essere risolto il seguente esercizio.

Esercizio. Si costruisca una routine che, data una stringa di al più 20 byte, provvede a stamparla via transmitter adottando un meccanismo di interrupt driven I/O. In particolare, per rendere il più efficiente possibile il suddetto meccanismo, si proceda come segue. La routine trasmette la stringa attraverso un buffer al gestore di interrupt, che provvederà ad inviare la stringa da stampare, carattere per carattere, al transmitter.

Soluzione.

```
        .data
msg1:   .asciiz "interrupt"
        .text
        .globl kbuf
        .globl main
main:   li $t4,0x0000ff01
        mtc0 $t4,$12      # abilita interrupt
        li   $t4,1000
        la $t0,msg1       # inizio trasf. stringa a buffer sistema
        la $t1,kbuf
trans:  lb $t2,0($t0)
        sb $t2,0($t1)
        addi $t0,1
        addi $t1,1
        bnez $t2,trans    # fine trasferimento stringa

        li $t0,0xffff0000 # inizio fase per porre a 1 il
        lw $t1,8($t0)     # bit di enable sul transmitter
        addi $t1,$t1,2
        sw  $t1,8($t0)    # enable bit del transmitter settato a 1

loop:   addi $t4,$t4,-1   # ciclo controllo
        bgez $t4,loop
        li   $v0,10
        syscall

       .ktext  0x80000080
       sw  $a0,save0      # salva il contenuto dei registri di CPU
       sw  $v0,save1      # che devono essere usati successivamente
       sw  $t0,save2
       sw  $t1,save3
```

```
      mfc0 $a0,$13       # carica in $a0 cause register
      li   $k1,0x200     # in $k1 codice interrupt output
      andi $a0, $a0, 0x200 # considera solo i bit del codice di eccezione
      bne  $a0,$k1,fine  # ignora interrupt diversi da livello 1

      la  $a0,kbuf       # carica prossimo carattere da visualizzare
      lw  $t0,indice
      add $a0,$a0,$t0
      lb  $v0,0($a0)
      beqz $v0,eout      # se 0x00 hai finito

c1:   lw $t1,0xffff0008  # ciclo di attesa
      rem $t1,$t1,2
      beqz $t1,c1
      sw   $v0,0xffff000c # invia carattere in stampa
      addi $t0,$t0,1
      sw   $t0,indice
      b    fine           # vai a fine interrupt handler

eout: lw   $t1,0xffff0008 # la stampa del buffer e' terminata
      addi $t1,$t1,-2     # riazzera enable bit e vai a fine
      sw   $t1,0xffff0008
      sw   $0,indice

fine: lw  $a0,save0      # ripristina il contenuto dei registri di CPU
      lw  $v0,save1      # utilizzati
      lw  $t0,save2
      lw  $t1,save3
      mfc0 $k1,$14       # in $k1 l'indirizzo di EPC

      rfe                # ripristina status register
      jr   $k1           # rientra  nel programma interrotto
       .kdata
save0: .word 0
save1: .word 0
save2: .word 0
save3: .word 0
indice: .word 0
kbuf:  .space 20
```

Alcune osservazioni riguardanti l'esercizio precedente. La prima è in relazione al ciclo di attesa presente nell'interrupt handler ed identificato dalla label `c1`. Tale ciclo non è in via di principio necessario. Difatti durante tale ciclo viene continuamente testato il ready bit in attesa che diventi 1; per contro, quando l'interrupt handler intraprende l'esecuzione del ciclo in questione, significa che è stato rilevato un interrupt di livello 1 e quindi il ready bit dovrebbe già trovarsi a uno. A causa però di alcuni problemi che si possono verificare con l'uso del simulatore SPIM, è sempre

consigliabile introdurre questo ciclo quando si gestisce il transmitter.

La seconda osservazione riguarda il ciclo presente nel programma principale e caratterizzato dalla label `loop`. Dopo aver "scaricato" al sistema operativo la gestione dell'operazione di I/O, il programma main in genere viene sospeso in attesa che l'operazione termini. Nel frattempo la CPU può essere dedicata allo svolgimento di altri programmi. Noi non abbiamo ancora gli strumenti per poter sperimentare questo fenomeno, è però possibile sperimentare la sovrapposizione tra i tempi di esecuzione del programma main e dell'operazione da lui richiesta. Il modo più semplice è proprio costituito dal loop in questione. In particolare lo studente è invitato a effettuare i seguenti esperimenti:

1. modificare il valore iniziale di $t4 e valutare come lo stesso influenza il comportamento del dispositivo di output; si noti che questo fenomeno testimonia direttamente che, mentre viene fatto l'output da parte del controller, il processore sta eseguendo qualcosa di diverso;
2. al ciclo loop si sostuisca una routine per il calcolo del fattoriale di un numero, si modifichi poi il codice del handler in modo tale che terminata l'operazione di output venga stampato il risultato calcolato fino a quel momento e forzata la terminazione del main.

L'esempio appena illustrato può essere usato come base per la realizzazione egli esercizi 8.17 e 8.18 del libro che lo studente è invitato a svolgere, al fine di verificare la propria comprensione degli argomenti sinora trattati.

11 Direct Memory Access

La gestione dell'I/O interrupt driven evita al processore di dover attendere che la periferica abbia terminato l'operazione, ma non solleva il processore dalla responsabilità del trasferimento dati. Per periferiche veloci abbiamo però visto che questa attività è preponderante rispetto al tempo speso in busy waiting. In particolare nel caso di periferiche veloci, durante l'esecuzione di un'operazione di I/O, il processore spende la stragrande maggioranza del suo tempo a trasferire dati. Per evitare l'intervento della CPU anche durante questa fase è stato introdotto il DMA (*direct memory access*). Il DMA è un processore specializzato nel trasferimento dati tra dispositivi di I/O e memoria centrale. In genere il DMA è supportato dai controller dei dispositivi, si parla infatti di DMA controller, o dai controller di bus in funzione dell'architettura del calcolatore in questione. Vediamo come viene eseguita un'operazione di lettura da disco sotto il controllo del DMA.

1. a fronte di una richiesta di I/O da parte di un programma utente, la CPU, attraverso il device driver, provvede a caricare nel DMA controller coinvolto nell'operazione le seguenti informazioni: *tipo di operazione richiesta*, *indirizzo della prima locazione* della zona di memoria centrale su cui scrivere i dati coinvolti nell'operazione, *il numero* di byte da leggere; dopodiché si svincola completamente dall'esecuzione dell'operazione di I/O;
2. il controller avvia l'operazione richiesta e acquisisce il primo blocco di dati dal disco, che memorizza in un suo buffer interno; terminata l'acquisizione verifica la correttezza dei dati letti e se non vi sono errori prosegue con il passo successivo, altrimenti ripete il passo (2);

3. nel caso in cui siano stati richiesti più blocchi da disco e fintantoché il controller ha posizioni disponibili all'interno del suo buffer, vengono continuamente acquisiti dati dal disco. Contemporaneamente il controller, che è arbitro di bus, dà inizio ad una fase di bus arbitration per acquisire l'accesso al bus. Ottenuto l'accesso al bus, trasferisce i dati dal proprio buffer alla memoria centrale a partire dall'indirizzo che gli è stato indicato al passo (1);

4. dopo aver effettuato il trasferimento di **tutti** i dati richiesti, il DMA segnala il completamento dell'operazione via interrupt.

Si noti in questo caso l'estrema importanza del buffer del DMA. E' infatti grazie alla presenza del buffer che il DMA può risolvere situazioni critiche quali l'arrivo di dati dal drive e l'impossibilità di accedere al bus.

Esercizio. Il DMA per effettuare i propri trasferimenti dati contende alla CPU il bus per l'accesso alla memoria centrale, costringendo quindi la CPU a rimanere inattiva durante questi trasferimenti. Come si può ovviare a tale situazione?