Architettura degli Elaboratori, a.a. 2006-07

Ingresso-Uscita e Interruzioni

Queste note sostituiscono, in maniera più organica, alcune parti delle Dispense che riguardano principalmente il sottosistema di I/O e il trattamento delle interruzioni, e/o servono da guida per l'utilizzazione del materiale didattico considerato.

Le parti delle Dispense sostituite sono:

- Cap. VI, sez. 3.2,
- alcune parti del Cap. VI, sez. 4,
- Cap. IX.

Le caratteristiche del sottosistema di I/O e delle interruzioni sono introdotte nel Cap. VI, sez. 3.1, 3.3 e nel Cap. VII, sez. 1.2.

Distinguiamo due aspetti riguardanti l'architettura di tale sottosistema:

1) *trasferimento di dati* tra Unità Centrale (CPU, M) e unità di I/O,
2) comunicazione di *eventi* asincroni da unità di I/O a Processore mediante il meccanismo delle *interruzioni.*

Questi due aspetti sono trattati nelle sezioni seguenti.

1. Ingresso-uscita: trasferimento dati

Il trasferimento dati tra Unità Centrale e unità di I/O avviene quando un programma, o processo, PROC, in esecuzione sulla CPU vuole inviare dati a, o ricevere dati da, una unità di I/O. PROC può essere il Driver dell'unità di I/O oppure un altro programma o processo (applicazione, modulo di sistema operativo, ecc).

I dati trasferiti possono essere di tipo elementare, implementati come parole o byte, o, più frequentemente, come *blocchi* di dati.

I modelli che vedremo, per realizzare la cooperazione tra Unità Centrale e I/O, hanno impatto sia sul livello firmware che sul livello assembler, e su di esse sono basate le funzionalità del nucleo di sistema operativo che riguardano la virtualizzazione e la gestione dell'I/O.

Distinguiamo due modelli:

- **Direct Memory Access (***DMA*): il trasferimento di dati tra I/O e memoria principale, e viceversa, avviene indipendentemente attraverso un canale distinto, utilizzando uno o più *Bus DMA*. Questi trasferimenti, per definizione, non interessano il processore, ma avvengono *in parallelo all'elaborazione in corso sul processore*;
- **Memory Mapped I/O** (*MM-I/O*): poiché ad ogni unità di I/O è associata una certa capacità di memoria locale, questa è *indirizzabile direttamente dal processore* P: il trasferimento dati tra P e I/O avviene dunque attraverso letture e scritture da parte di P nelle memorie locali di I/O. In altre parole, un programma in esecuzione su P *vede le unità di I/O come memoria.*

Nel modello DMA, M è *condivisa* tra P e le unità di I/O operanti in DMA.

Nel modello MM-I/O, ogni memoria locale di unità di I/O (MI/O) è *condivisa* tra P e tale unità.

I due modelli possono essere utilmente combinati, dando luogo ad un modello in cui ogni trasferimento di I/O è sempre effettuato come accesso in una memoria.

1.1. Tecnologia delle unità di I/O

Nel caso più tradizionale, le unità di I/O sono specializzate *a firmware* nei confronti dell'elaborazione delle operazioni esterne ad esse affidate. Tali unità contengono spesso una memoria di capacità significativa, ad esempio per bufferizzare uno o più blocchi di dati ricevuti dal dispositivo o da trasmettere all'Unità Centrale.

Allo stato attuale della tecnologia, e come chiara tendenza, le unità di I/O sono realizzate mediante veri e propri *processori general-purpose* standard, ognuno con la propria CP, memoria locale (anche di notevole capacità) e interfacce verso il dispositivo. In questo caso, le operazioni esterne sono interpretate a livello assembler. Questo permette una maggiore flessibilità e modificabilità, ed offre la possibilità di eseguire le funzionalità del Driver direttamente da parte dell'unità di I/O, rendendo di fatto indistinguibile la differenza tra unità e Driver, ed anzi rendendo possibile introdurre ottimizzazioni nell'implementazione delle funzionalità offerte alle applicazioni.

Da questo punto di vista, concettualmente un qualunque elaboratore è un multiprocessor a memoria condivisa, dove i processori sono la (le) CPU e le unità di I/O, e la memoria condivisa è implementata con la tecnica del DMA e/o MM-I/O.

1.2. Modello DMA

Questo modello viene adottato per unità di I/O che effettuano trasferimenti a blocchi in modo indipendente da, e quindi in parallelo a, l'elaborazione del processore centrale. Ad esempio, una semplice unità disco può ricevere richieste così espresse

- lettura, indirizzo base del blocco su disco da cui leggere, indirizzo base in M del blocco in cui scrivere,
- scrittura, indirizzo base del blocco su disco in cui scrivere, indirizzo base in M del blocco da cui leggere.

Nel caso di unità di I/O realizzate con processori general-purpose, gli indirizzi generati dalle unità di I/O sono *logici* e vengono tradotti in indirizzi fisici da una propria MMU.

Nel caso di unità specializzate a firmware, in assenza di una MMU gli indirizzi generati devono essere direttamente quelli *fisici*. Normalmente, tali indirizzi saranno comunicati da programmi in esecuzione sulla CPU, che, allo scopo, provvederanno alla traduzione da logico a fisico.

L'arbitraggio del Bus DMA è indipendente da quello del Bus di I/O. Spesso, per aumentare la banda, vengono adottate soluzioni strutturate ad albero, come nel caso dello standard PCI.

La memoria principale M si trova ora ad avere due interfacce: una dedicata con MMU, l'altra con il Bus DMA. Il microprogramma di M è nondeterministico in quanto deve *arbitrare le situazioni di conflitto*, per la presenza contemporanea di messaggi sulla due interfacce d'ingresso, dando priorità ad una di esse (tipicamente, al Bus DMA), oppure stabilendo una disciplina di priorità variabile.

1.3. Modello Memory Mapped I/O

Come detto, a differenza del modello DMA (che ha come scopo quello di implementare trasferimenti diretti tra M e I/O, "scavalcando" il processore), il modello MM-I/O ha come scopo quello di implementare efficientemente trasferimenti tra processore e unità di I/O.

In molte architetture che usano questo modello, a livello assembler *non esistono esplicite istruzioni di I/O*, essendo sufficienti le istruzioni di **LOAD** e **STORE**: poiché gli indirizzi generati dal processore sono *logici*, il processore ha visibilità di uno spazio uniforme di memoria, *senza* distinzione tra M e {MI/O}*; è solo la MMU che, mediante l'indirizzo fisico ottenuto dalla traduzione dell'indirizzo logico, ha visibilità di tale distinzione.*

Questo modello è adottato anche dalla **macchina assembler Risc del Cap. V**.

Ogni unità di I/O, accedendo solo sulla rispettiva memoria locale, potrà operarvi con indirizzi logici (processori veri e propri) o fisici (unità specializzate a firmware): questa distinzione è del tutto invisibile ai programmi con interagiscono con l'unità.

Si consideri il caso di un sistema con una memoria di capacità massima *complessiva* 1G parole e con un numero massimo di 32 unità di I/O operanti in MM-I/O, ognuna con una memoria locale (MI/O) di capacità massima di 8 Mbyte. La memori principale M ha quindi capacità massima di 844M parole. Complessivamente l'elaboratore è *come se* contenesse una *memoria modulare*, di capacità complessiva 1G parole, con organizzazione *sequenziale* [1], suddivisa in 128 moduli ognuno di 10M parole, dei quali i primi 32 coincidenti con MI/O_0, ..., MI/O_{31} e gli altri 96 corrispondenti a blocchi di 10M parole di M. Si noti che, agli effetti di M, questa è una astrazione di comodo: nella realtà M potrà non essere modulare, o comunque non avere un numero di moduli così grande. I 7 bit più significativi dell'indirizzo *fisico* rappresentano *l'identificatore del modulo*, e permettono ad MMU di distinguere il modulo a cui instradare la richiesta del processore, mentre i restanti 23 bit rappresentano *l'indirizzo all'interno di quel modulo*.

È importante notare come, con l'architettura Memory Mapped I/O, *qualunque* comunicazione tra processore e unità di I/O avviene mediante istruzioni di Load e Store. In altri termini, agli effetti del processore, qualunque informazione debba essere scambiata con una unità di I/O è *come se* corrispondesse ad una locazione di memoria:

1. in certi casi tale locazione esisterà realmente, ed allora verranno effettivamente eseguite letture/scritture in essa,
2. in altri sarà una astrazione cui il processore fa riferimento, ed allora sarà l'unità di I/OP ad interpretare opportunamente le richieste di lettura/scrittura per eseguire, in realtà, funzionalità diverse.

Ad esempio, supponiamo che un Driver debba richiedere all'unità di I/O associata le seguenti operazioni esterne:

---

[1] Vedi "Note sull'implementazione di componenti logici memoria".

1. provocare il trasferimento di un blocco di dati dal dispositivo alla memoria interna dell'unità,
2. provocare il trasferimento di un blocco di dati dalla memoria interna all'unità al dispositivo,
3. leggere un dato dalla memoria interna dell'unità,
4. scrivere un dato nella memoria interna all'unità. [2]

I casi 3, 4 corrispondono in maniera naturale ad istruzioni di Load e Store rispettivamente.

Per implementare i casi 1, 2 si può pensare di implementare l'unità di I/O in modo che *una specifica locazione* della propria memoria, ad esempio quella di indirizzo fisico zero (internamente al modulo di memoria allocato all'unità), sia *fittizia*: ricevendo una richiesta di scrittura a tale indirizzo, l'unità interpreta il valore del dato come un "codice di operazione esterna", che non potrà non venire realmente scritto nella locazione di indirizzo zero, bensì usato per acquisire informazioni sulla richiesta del Driver. Nel nostro caso, ad esempio, a seconda che il dato "da scrivere" sia un numero pari o dispari, il significato della richiesta del processore è di eseguire l'operazione esterna 1 oppure la 2, rispettivamente. È dunque sufficiente che il Driver esegua istruzioni di *Store* con indirizzo e dato sorgente opportuno.

2. Trattamento delle interruzioni

Rispetto alla trattazione del Cap. VII, sez. 4, vengono fatte le seguenti precisazioni.

2.1. Trattamento ai vari livelli

A livello firmware le interruzioni hanno il significato di segnali di richiesta all'arbitro del Bus di I/O di inviare un messaggio di I/O su tale bus. Ai livelli superiori, il significato di una interruzione è di segnalare, attraverso il messaggio di I/O, un **evento**. Una procedura del programma in esecuzione, che chiameremo **Handler** dell'interruzione (o dell'evento), provvede a compiere le azioni necessarie per trattare l'evento stesso, eventualmente chiedendo la collaborazione di altri programmi o processi del sistema.

Ad ogni tipo (classe) di evento che le unità di I/O possono comunicare (in generale, una stessa unità potrà comunicare più eventi distinti) corrisponde un Handler distinto.

Ad esempio, si consideri una unità di I/O che contiene una coda di al più 1K parole, ottenute dal dispositivo associato. L'unità segnala il seguente evento: "ho almeno una parola in coda". Alla ricezione dell'interruzione che segnale tale evento, viene eseguito un Handler così definito: "estrarre la prima parola in coda e scriverla in una locazione della memoria virtuale di indirizzo noto".

Nella sez. 4.1 del Cap. VII, il concetto espresso dalla **figura 10** è quello fondamentale per comprendere che il trattamento dell'interruzione, che prevede due fasi:

- una prima **fase firmware**, nella quale il processore accetta l'interruzione, attende il messaggio di I/O e, usando tale messaggio come parametro, chiama una procedura indicata con *Routine di Trattamento Interruzione*. Il messaggio di I/O (ad esempio, due parole) conterrà *l'identificatore del tipo di evento* ed un dato elementare; almeno la prima parola è indispensabile.

[2] Si ricordi che un processo (e quindi il processore) genera indirizzi logici. La dizione "memoria interna all'unità di I/O" non è, quindi, formalmente corretta; si tratta solo di una abbreviazione per "**parte della memoria virtuale che risulta allocata nello spazio fisico di I/O**".

- una seconda **fase assembler**, nella quale viene eseguita la Routine di Trattamento Interruzione e la procedura Handler dell'evento comunicato con l'interruzione. Nella figura 10 della Dispensa, i blocchi indicati con *Driver_1, …, Driver_k* vanno ridenominati *Handler_1, …, Handler_k.*

La Routine di Trattamento Interruzione serve semplicemente ad interfacciare la fase firmware agli Handler, *svincolando la progettazione del processore della conoscenza a priori degli Handler stessi*. Ad esempio, tale Routine può essere:

LOAD Rtab_int, Revent, Rhandler

CALL Rhandler, Rret_handler

GOTO Rret_prog

dove il registro di indirizzo *Rtab* contiene l'indirizzo base di una Tabella di Puntatori ai Codici degli Handler e quello di indirizzo *Revent* contiene il codice dell'evento ricevuto con il messaggio di I/O.

Si osservi che *tutte le procedure della fase assembler (Routine di Trattamento Interruzione, insieme degli Handler) fanno parte della Memoria Virtuale di ogni programma, assieme ai relativi dati* (Tabella di Puntatori ai Codici degli Handler, altre strutture dati degli Handler). Esse potranno essere presenti in memoria principale in singola copia o replicate, a seconda della specifica politica di gestione dello spazio fisico.

Gli Handler sono procedure che sono realizzate in modo diverso a seconda del modello di sistema operativo e di gestione dell'I/O. Ad esempio, possono coincidere con i Driver (se questi sono appunto implementati come procedure), oppure essere interfacciare i Driver stessi o altri processi. *Il modello è sufficientemente generale da adattare l'architettura firmware-assembler a qualsiasi sistema operativo e di gestione dell'I/O ai livelli superiori.*

Nota: non considerare la parte da pag. VII.38, ultime cinque righe, a pag. VII.39, prime 13 righe.

2.2. Fase firmware

La fase firmware è esposta nel Cap. VII, sez. 4.2. Senza usare registri A, B in uscita alla memoria RG del processore (Cap. VII, sez. 2.2.2), il microprogramma diviene:

tratt_int. reset INT, set ACKINT, **int1**

int1. (RDYIN, or(ESITO), = 0-) nop, int1; (= 11) ESITO → N[29 .. 31], reset RDYIN, tratt_ecc ;

(= 10) reset RDYIN, set ACKIN, DATAIN → RG[61], int2

{ricevuto il nome dell'unità di I/O, questo viene passato alla Routine di Trattamento Interruzione via RG[61]}

int2. (RDYIN, or(ESITO), = 0-) nop, int2; (= 11) ESITO → N[29 .. 31], reset RDYIN, tratt_ecc ;

(= 10) reset RDYIN, set ACKIN, DATAIN → RG[62], IC → RG[63], RG[60] → IC, ch0

{ricevuta la seconda parola del messaggio di I/O, questa viene passata alla Routine di Trattamento Interruzione via RG[62], viene salvato l'indirizzo di ritorno dalla Routine, ed effettuato il salto alla Routine stessa}

Si osservi che

1) *la semantica di tutte le istruzioni assembler* include, oltre alla "semantica normale", la "semantica in seguito ad interruzioni" secondo la quale una qualunque istruzione deve anche essere capace di effettuare una *chiamata di procedura*. Questo vale anche per la "semantica in seguito ad eccezioni";

2) per le informazioni che la fase firmware passa alla fase assembler (parole del messaggio di I/O) e che usa per collegarsi a tale fase (indirizzo della Routine Trattamento Interruzione, indirizzo di ritorno) non è necessario utilizzare registri generali dedicati. Per risparmiare sul loro numero, si può sempre ricorrere ad un pacchetto di locazioni della Memoria Virtuale, al prezzo di un maggior tempo di elaborazione per gli accessi alla memoria. Allo scopo, è necessario conoscere l'indirizzo base di tale pacchetto. Tipicamente, tutte le *informazioni di utilità* del processore (incluse molte informazioni di nucleo) sono riunite nel *PCB* del processo:

   - immagine di RG, immagine di IC,
   - informazioni per il trattamento delle interruzioni,
   - Tabella dei Puntatori agli Handler delle interruzioni,
   - informazioni per il trattamento delle eccezioni,
   - Tabella dei Puntatori agli Handler delle eccezioni,
   - puntatore alla Tabella di Rilocazione,
   - stack per procedure annidate,
   - ecc,

   ognuna in una posizione nota a priori. Per puntare alla base del PCB è utilizzato un registro generale (*Rpcb*).

   Nel microprogramma della fase firmware del trattamento interruzioni, utilizzando *Rpcb* vengono effettuati accessi i necessari memoria a determinate parole del PCB.

3. Istruzioni per la gestione delle interruzioni

Nel Cap. V, sez. 2.6 sono introdotte, nel linguaggio assembler Risc, alcune "istruzioni speciali", tra le quali quelle per far sì che il programma in esecuzione possa, durante certe sequenze di istruzioni, ignorare la presenza di alcune o tutte le interruzioni.

Ad esempio, conviene che la fase assembler del trattamento interruzioni venga eseguita *a interruzioni disabilitate*, in modo da semplificare la sua progettazione e/o non provocare inconsistenze (ad esempio, gli Handler potrebbero condividere dei dati modificabili). La stessa situazione si ha per le procedure di trattamento delle *eccezioni*, ed *in generale per le funzionalità del nucleo del sistema operativo*.

Le istruzioni sono definite nella sez. 1.2 del Cap. VII:

- **Disabilita Interruzioni** (DI) : nessuna interruzione è accettata dal processore ;
- **Riabilita Interruzioni** (EI) : tutte le interruzioni abilitate prima di eseguire l'ultima DI sono nuovamente abilitate ;
- **Cambia Maschera Interruzioni** (MASKINT) : definita INTMASK come un array di bit tale che INTMASK[i] = 1 se l'interruzione da I/O_i è abilitata, l'istruzione consiste

nell'inviare un valore di INTMASK a UNINT. All'interno di questa, ogni segnale INT_i è messo in AND con INTMASK[i].

La loro implementazione è descritta sinteticamente nel Cap. VII, sez. 4.3.

Per ragioni legate alla progettazione di codice di nucleo, conviene adottare anche una soluzione più flessibile che permetta di **specificare in una qualsiasi istruzione anche la disabilitazione o abilitazione delle interruzioni**. Poiché la maggior parte delle istruzioni dell'assembler Risc del Cap. V hanno almeno i sei bit meno significativi non specificati, due di questi bit possono essere specificati per indicare, *opzionalmente,*

- la disabilitazione delle interruzioni, a partire dall'istruzione stessa,
- la riabilitazione di tutte le interruzioni, a partire dall'istruzione successiva.

Ad esempio:

ADD Ri, Rj, Rk, **DI**

LOAD Ri, Rj, Rk, **EI**

Questa opzione non solo permette di risparmiare le istruzioni DI, EI, ma soprattutto permette di disabilitare/riabilitare le interruzioni *contemporaneamente ad altre azioni*, risparmiando in complessità del programma e tempo di elaborazione.

Ad esempio, per concludere la fase assembler del trattamento interruzioni, la Routine Trattamento Interruzioni esegue l'istruzione "GOTO Rret_prog". La riabilitazione delle interruzioni non può quindi essere inserita "dopo" l'ultima istruzione di tale Routine; con il metodo introdotto si ha invece:

LOAD Rtab_int, Revent, Rhandler, **DI**

CALL Rhandler, Rret_handler

GOTO Rret_prog, **EI**

Si noti che anche l'Handler, essendo una procedura chiamata dalla Routine Trattamento Interruzione, viene eseguito a interruzioni disabilitate.

In generale, questa "facility" è utile in tutte le situazioni (e sono molto frequenti) in cui il codice da eseguire in modo *non interrompibile* è una procedura (o un annidamento di procedure).

Un altro esempio lo vedremo nella fase di commutazione di contesto (Cap. VI, sez. 2.3), nella quale è conveniente riabilitare le interruzioni durante l'ultima istruzione della fase stessa (START_PROCESS, Cap, V sez. 2.7).

Un'altra estensione utile è data dall'introduzione di una istruzione *per attendere, in attesa attiva, una determinata interruzione*:

WAITINT Ri/o, Revent, Ra, Rb

dove *Ri/o* contiene il nome di una unità di I/O; *Revent* il codice di un evento segnalabile da parte di tale unità; *Ra, Rb* sono i registri generali dove è memorizzato il messaggio di I/O. Il microprogramma dell'istruzione

1. provvede a porre ad uno, nella maschera delle interruzioni, il bit corrispondente all'interruzione attesa, e tutti gli altri bit a zero;

2. attende l’interruzione, invia l’ack all’interruzione, e attende il messaggio di I/O che verrà copiato nei registri Ra, Rb;
3. rimette la maschera delle interruzioni al valore precedente;
4. incrementa IC di uno.

Esempi di utilizzo si hanno nell’implementazione del nucleo per architetture multiprocessor. Un altro esempio di utilizzo si ha in sistemi specializzati (“embedded”), realizzati mediante CPU general-purpose, nei quali le comunicazioni tra le CPU possono essere effettuate direttamente via I/O ed interruzioni.