Ingresso/uscita (note)
AE, Corso A e B, Anno Accademico 2020–21

M. Danelutto

10 dicembre 2020

Indice

Capitolo 1

Ingresso uscita

In questo capitolo delle note vediamo tre cose in particolare, che sul libro di testo sono solo accennate oppure non sono neppure trattate: memory mapped I/O, DMA e trattamento delle interruzioni. Tutte e tre hanno un ruolo fondamentale nella gestione dell'ingresso/uscita.

1.1 Memory mapped I/O

Per memory mapped I/O si intende la possibilità di accedere ed interagire con i dispositivi di ingresso uscita utilizzando le istruzioni messe a disposizione dal linguaggio assembler per interagire con la memoria, ovvero le istruzioni di load e store.

L'astrazione che utilizziamo per modellare un dispositivo di ingresso uscita è quella dell'unità firmware, ovvero un dispositivo hardware che quando viene acceso comincia ad eseguire un ciclo infinito nel quale attende che gli venga ordinato di eseguire un comando, lo esegue, ne riporta i risultati e ricomincia la prossima iterazione:

```
dispositivoI/O:
  while(true) {
    read(comando, parametri);
    results = exec(comando, parametri);
    writeback(results);
  }
```

Consideriamo una dispositivo di input qual'è la tastiera. Il tipico comando che gli può essere impartito è quello di lettura di in carattere. L'esecuzione del comando comporta l'attesa dello schiacciamento di un tasto da parte dell'utilizzatore della tastiera. Il risultato da restituire, quando il tasto sia stato effettivamente premuto, è il codice del tasto che è stato schiacciato. Supponiamo per il momento che questa sia l'unica operazione che possa essere richiesta al dispositivo tastiera.

Il dispositivo disporrà tipicamente di alcuni registri che permettono di memorizzare comandi, parametri e risultati delle operazioni che sono in grado di eseguire. Nel caso della tastiera ci possiamo immaginare di avere:

- un registro da 1 bit che sia interpretato come ordine di lettura di un tasto: se il bit è a 0, non è richiesta alcuna lettura, se il bit è a 1, deve essere letto un carattere;
- un registro lungo abbastanza per contenere il numero di bit utilizzati per rappresentare uno qualunque dei tasti che possono essere premuti. Su una tastiera tradizionale, il codice relativo ad un tasto potrebbe corrispondere alle sue coordinate riga/colonna. Di norma ci sono 6 righe di tasti (compresi i tasti Fn) e una quindicina di colonne diverse (spesso non perfettamente verticali). Per rappresentare un tasto serviranno quindi ca 21 bit. Immaginiamo che il tasto possa dunque essere rappresentato su una configurazione da 32 bit, contando anche che servono altri bit per indicare se il tasto è stato premuto insieme a uno shift, control, alt, meta, fn ...

Con queste ipotesi, il ciclo di funzionamento del nostro dispositivo tastiera potrebbe essere:

```
while(true) {
  if(ordinedilettura == 1) {
     ... attendi pressione tasto ...;
     codice = codice(tastoPremuto);
     0 -> ordinedilettura;
  }
}
```

La tecnica del memory mapped I/O permette di utilizzare le normali istruzioni di load e store (LDR e STR nel caso di ARM) per andare a scrivere e leggere i registri interni dell'unità. In pratica, supponiamo di avere a disposizione degli indirizzi codificati in modo da essere dirottati sull'unità di I/O invece che nella memoria. Per esempio, immaginiamo che i registri della nostra tastiera corrispondano agli indirizzi 1024 e 1025.

Un programma che voglia leggere un dato dalla tastiera dovrebbe quindi eseguire un codice tipo:

```
1       mov r0, #1024   @ indirizzo base del dispositivo
2       mov r1, #1      @ costante 1
3       str r1, [r0]    @ ordina la lettura di un tasto
4 wait: ldr r1, [r0]    @ controlla termina lettura
5       cmp r1, #1
6       beq wait        @ e in caso attendi
7       ldr r0, [r0, #1]@ carica codice del tasto letto
8       mov pc, lr      @ ritorno al chiamante
```

ATTENZIONE: questo modo di interagire con l'ingresso uscita è assolutamente inefficace: vengono eseguite istruzioni a vuoto (ciclo "wait") in attesa

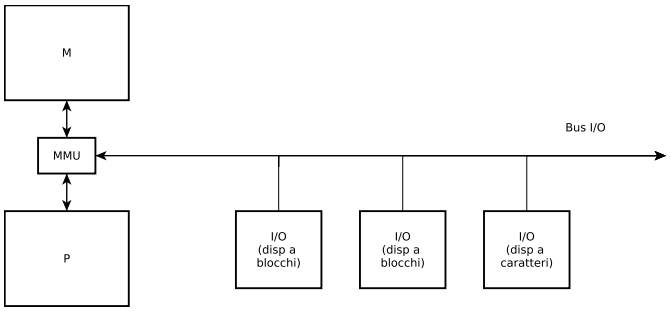

Figura 1.1: Bus I/O

che l'utente schiacci un tasto. Un ritardo di un paio di secondi da parte dell'utente a schiacciare il tasto dopo che è stata fatta partire la routine di lettura implica, su una macchina con un clock da 1GHz, l'esecuzione di qualcosa come 2sec/3nsec istruzioni in attesa che venga letto il carattere! (vedi Sez. 1.3)

Ma come facciamo a fare in modo che le load e le store del codice abbiano effetto sul dispositivo e non siano interpretate come normali load e store sul sottosistema di memoria?

Il processore riconosce alcuni indirizzi come appartenenti allo spazio di I/O (è questo il vero e proprio concetto di memory mapped I/O) e, tramite la MMU, redirige le richieste relative a questi indirizzi sul bus che collega processore e dispositivi di I/O (vedi Fig. 1.1). Tutti i dispositivi di I/O vedono passare le operazioni, ma ognuno intercetta solo quelle che fanno riferimento agli indirizzi che gli sono stati assegnati. Per esempio, il nostro dispositivo tastiera controllerà quali sono gli indirizzi indicati nelle load e store che passano sul bus e tratterrà per se, ed eseguirà opportunamente, solo quelli relativi all'indirizzo 1024 e 1025. Lo schema semplificato che implementa questa soluzione è quello riportato nella Fig. e9.1 del libro[1].

Lo schema della Fig. e9.1 in realtà fa riferimento a un modello molto semplice in cui l'unità di ingresso uscita accede esclusivamente un registro in ingresso e di conseguenza produce un dato in uscita. Il dispositivo di I/O in oggetto riceverà tutti i dati relativi alle operazioni di lettura scrittura effettuate nello spazio di indirizzamento di ingresso uscita.

Per come lo abbiamo descritto, il memory mapped I/O fa in realtà corrispondere un certo insieme di indirizzi a indirizzi di una memoria interna alla unità di ingresso uscita. Vediamo come questo funziona con un semplice esempio.

Immaginiamo che gli indirizzi dedicati all'I/O siano gli indirizzi che vanno da 1024 a 2048 e che gli indirizzi 1024 a 1025 debbano corrispondere a 2 locazioni di memoria di un certo dispositivo di I/O (la nostra tastiera). Come

[1] Il capitolo 9 del libro, che parla dell'ingresso uscita, può essere scaricato dal sito web dell'editore in forma PDF

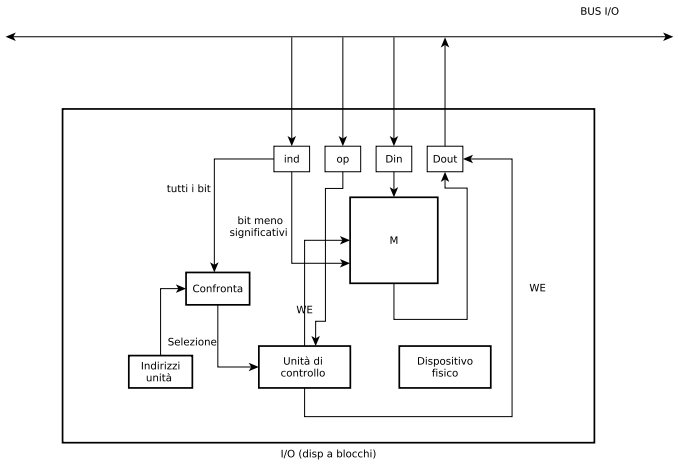

Figura 1.2: Unità di I/O

prima cosa, le operazioni di load e store eseguite dal processore verranno dirottate verso

- il sottosistema di memoria se l'AND dei bit IND[31:10] è 0 oppure l'OR dei bit IND[31:11] è 1
- il sottosistema di I/O nessuna delle due condizioni precedenti è vera.

Quando l'unità che gestisce la tastiera vede passare un indirizzo, controlla che sia compreso nell'intervallo $[1024, 1025]$, ovvero che valga IND[10]=1 e OR$\{$IND[31:11], IND[9:0]$\}$=0 oppure che IND[10]=1, IND[0]=1 e OR$\{$IND[31:11], IND[9:1]$\}$=0. Se questa condizione è vera, utilizza l'ultimo bit come indirizzo della propria memoria interna di due posizioni ed esegue l'operazione richiesta dal processore (LDR o STR). Diversamente ignora l'operazione (la Fig. 1.2 illustra un possibile schema di implementazione della unità che controlla la tastiera interagendo col bus di I/O).

La tecnica del Memory Mapped I/O richiede una fase di negoziazione fra dispositivo e sistema operativo (codice che di fatto si occupa delle interazioni con i dispositivi di ingresso uscita). Gli indirizzi riservati all'I/O sono definiti dal costruttore del processore. Come questi vengano mappati sugli indirizzi riconosciuti dai dispositivi come propri è normalmente oggetto di negoziazione durante l'inizializzazione dei dispositivi "plug-and-play".

E' da notare che gli indirizzi utilizzati per indirizzare i registri dei dispositivi di ingresso uscita potrebbero non essere soggetti al normale processo di traduzione da indirizzi logici a indirizzi fisici nella MMU o che la MMU stessa potrebbe implementare una traduzione dall'indirizzo riservato dal costruttore del processore a quello invece realmente utilizzato nel dispositivo.

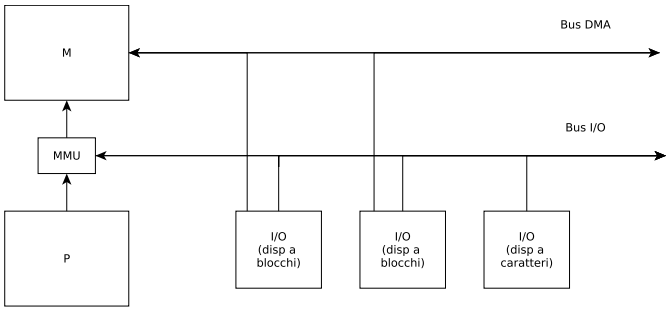

Figura 1.3: Sistema con periferiche in DMA

1.2 DMA

DMA sta per *direct memory access* ed è una tecnica che si usa per implementare i dispositivi di ingresso uscita in modo da permettergli un accesso controllato alla memoria centrale.

I dispositivi di I/O, per come li abbiamo visti nella Sez. 1.1 dovrebbero lavorare esclusivamente sulle loro memorie interne anche quando devono trattare operazioni che coinvolgono moli di dati relativamente grosse. Ad esempio, una unità di controllo di un disco lavora normalmente con operazioni su *blocchi*, ovvero mette a disposizioni operazioni elementari di lettura e scrittura di un blocco del disco, di dimensioni dell'ordine del Kbyte. In questo caso, una ipotetica operazione di lettura richiederebbe che il processore trasferisse il blocco dalla memoria del dispositivo alla memoria centrale mediante una serie di load. Questo è chiaramente un problema, perché può impegnare il processore per un tempo relativamente lungo.

La tecnica del DMA consente ad un dispositivo di accedere direttamente alla memoria centrale del sistema. In pratica (vedi Fig. 1.3):

- il dispositivo è collegato alla memoria centrale mediante un *BUS DMA* e può effettuare letture e scritture in memoria quando ha ottenuto la possibilità di utilizzare il bus;

- il bus DMA è condiviso fra tutte le periferiche in grado di lavorare in DMA;

- il processore passa alle periferiche, quando ordina l'esecuzione di una certa operazione, gli eventuali indirizzi necessari a completare l'operazione direttamente in memoria centrale (questo avviene tramite il bus di I/O);

- la memoria diventa un dispositivo che deve essere in grado di soddisfare le richieste di lettura e scrittura che arrivano da più soggetti (processore, via interfaccia di memoria standard, e periferiche, via bus DMA) e deve pertanto essere dotato di una unità controllo più sofisticata.

Con queste assunzioni, un'operazione di lettura dal disco su un dispositivo che supporta sia Memory Mapped I/O che DMA avviene tramite i seguenti passi:

1. il sistema operativo (programma in esecuzione sul processore), invia alla periferica utilizzando il memory mapped I/O l'ordine di esecuzione della lettura che contiene:
   - la specifica che si tratta di una operazione di *lettura*;
   - l'*indirizzo del blocco* di disco interessato
   - l'*indirizzo in memoria* del buffer da utilizzare per i dati letti da disco;
2. il dispositivo legge i dati dal disco in un proprio buffer interno. Questa operazione deve avvenire in un buffer interno per ragioni di temporizzazione: attendere l'accesso al bus DMA e conseguentemente alla memoria centrale potrebbe risultare troppo lungo (o in ritardo) rispetto ai tempi dettati dal trasferimento dei dati dal dispositivo, una volta arrivati al blocco desiderato;
3. infine, il dispositivo acquisisce, interagendo con l'arbitro del bus, il controllo del bus DMA e avvia il trasferimento del blocco letto nella posizione di memoria il cui indirizzo base è stato passato dal sistema operativo fra i parametri dell'operazione di lettura.

1.2.1 Dispositivi DMA

I dispositivi che normalmente sono dotati di DMA sono quelli cosiddetti *a blocchi*, ovvero quelli che operano su blocchi di dato. Tipicamente, si tratta di dispositivi di memorizzazione di massa (dischi a stato solido e non), di rete (interfacce di rete Ethernet), interfacce video, etc. Quelli che non sono dotati di DMA sono invece quelli cosiddetti *a caratteri* che comprendono dispositivi come tastiere, mouse, tavolette grafiche, etc.

1.3 Interruzioni

Un'interruzione è un evento asincrono[2] rispetto all'esecuzione del programma sul processore. Nel caso più generale, le interruzioni o eccezioni vengono utilizzate per scopi diversi:

- per gestire le operazioni di ingresso uscita; tenendo conto del fatto che il completamento di una operazione di I/O richiede tempi molto lunghi

[2] in caso di interruzioni legate ad errori che si possono verificare nel processore, sono in realtà eventi sincroni, diretta conseguenza di azioni intraprese durante l'esecuzione di istruzioni assembler da parte del processore. In questo caso sarebbe più corretto parlare di "eccezioni". Nel caso di architetture ARM il termine eccezione viene spesso utilizzato per tutti i casi, ovvero per le interruzioni (asincrone) e per le eccezioni vere e proprio (sincrone).

rispetto alla scala di tempi richiesta dal processore per eseguire le istruzioni assembler, si utilizzano le interruzioni per segnalare il completamento di una operazione di I/O, ordinata dal processore utilizzando memory mapped I/O. In questo modo, il processore è svincolato dalla necessità di attendere esplicitamente il completamento delle operazioni di ingresso uscita e può utilizzare il tempo speso dall'unità per completare l'operazione richiesta per svolgere altri compiti;

- per gestire situazioni di errore conseguenti ad azioni specifiche eseguite dal programma (fetch di una istruzione il cui codice non è riconosciuto come uno dei codici delle istruzioni assembler implementate dal processore, fault di pagina, divisione per 0).

- per eseguire chiamate di sistema, ovvero per chiamare parti di codice assembler con diritti diversi da quelli normalmente assegnati agli utenti (in generale con minimi livelli di privilegio);

Le interruzioni sono generate da dispositivi esterni al processore, quali dispositivi di I/O, ma anche dal sottosistema di memoria, per esempio in caso la memoria voglia segnalare un fault di pagina. Le eccezioni (interruzioni sincrone) sono invece quelle generate direttamente dal processore in caso di errori di vario genere.

Le interruzioni vengono *sentite* dal processore alla fine di ognuna delle iterazioni del ciclo *fetch-decode-execute*, come già accennato nelle parti precedenti del corso. Il processore, come tutte le unità firmware, implementa un ciclo simile al seguente:

```
while(true) {
  IR = fetch(PC);
  decode(IR);
  execute(IR);
  update(PC);
  if(interrupt) {
    interrupt_management();
  }
}
```

Ad ogni iterazione, in assenza di interruzioni, il processore preleva, decodifica ed esegue una singola istruzione assembler[3].

Quando si verifica un'interruzione, il suo trattamento prevede una sequenza di passi standard:

- l'istruzione in esecuzione viene completata e lo stato del processore viene aggiornato di conseguenza (per esempio, viene calcolato il PC corrispondente alla prossima istruzione da eseguire e, se necessario, viene effettuato il writeback dei risultati nel file dei registri o nella memoria dati);

- si salva parte dello stato del processore (tipicamente almeno PC, LR e SP) e si procede ad eseguire una parte di codice assembler che

[3]per non complicare la spiegazione stiamo assumendo di lavorare con un processore single o multi-cycle, non pipeline

  - dipende dal tipo di interruzione che si è verificata, e
  - contiene l'intero codice necessario a gestire quel tipo di interruzione

  L'esecuzione del codice di trattamento avviene in uno *stato* particolare, a seconda del tipo di interruzione, in generale con privilegi diversi (maggiori) rispetto ai privilegi disponibili in nello stato "user" (stato in cui un normale utente del processore esegue i propri programmi);

- al termine, si ripristina lo stato del processore e, al prossimo ciclo **while(true)** si procede ad eseguire la "prossima" istruzione del programma che era stato interrotto, quella corrispondente al PC calcolato quando ci siano accorti dell'interruzione, prima di saltare alla routine di trattamento.

Se volessimo essere più precisi, il ciclo fetch-decode-execute presentato nel listato precedente andrebbe presentato in modo leggermente diverso, ovvero:

```
1  while(true) {
2    try {
3      IR = fetch(PC);
4      decode(IR);
5      execute(IR);
6      update(PC);
7    } catch (exception e) {
8      exception_management();
9    }
10   if(interrupt) {
11     interrupt_management();
12   }
13 }
```

Il listato evidenzia come, in presenza di eccezioni, venga immediatamente eseguito un handler. Se si è verificato un errore durante la execute (o la fetch), viene immediatamente invocato l'handler dell'eccezione, senza di fatto aggiornare il PC. Quando l'handler risolve il problema ritorna all'esecuzione del codice originale ripetendo l'istruzione che ha causato l'eccezione, visto che il PC non è stato ancora aggiornato.[4] ARM riconosce un certo numero di stati diversi, riassunti nella Fig. 1.4. Gli stati Fast Interrupt e Interrupt sono gli stati che vengono utilizzati per il trattamento delle interruzioni generate dai dispositivi di ingresso uscita. Gli stati Abort e Undefined sono utilizzati per trattare le eccezioni generate da accessi illegali in memoria (per fetch o ldr/str) e per quelle generate da tentativo di esecuzione di una istruzione (assembler) illegale. Lo stato Supervisor viene utilizzato per l'esecuzione delle chiamate di sistema (istruzioni SVC), che sono anche loro gestite come "interruzioni software". Ciascuno di questi

[4] Questo meccanismo permette, per esempio, il trattamento del fault di pagina. La MMU genera un'eccezione, l'handler provoca l'esecuzione della parte di sistema operativo che si occupa del page fault. Quando il trattamento del page fault da parte del sistema operativo è terminato, il programma che aveva generato il fault viene fatto ripartire e riparte esattamente dal punto in cui si era fermato, cioè ordinando lo stesso accesso in memoria che aveva causato il fault. Questa volta l'accesso avverrà senza problemi visto che il sistema operativo ha appena effettuato i passi necessari a portare in memoria centrale la pagina mancante.

Mode	Privileged	Purpose
User	No	Normal operating mode for most programs (tasks)
Fast Interrupt (FIQ)	Yes	Used to handle a high-priority (fast) interrupt
Interrupt (IRQ)	Yes	Used to handle a low-priority (normal) interrupt
Supervisor	Yes	Used when the processor is reset, and to handle the software interrupt instruction `swi`
Abort	Yes	Used to handle memory access violations
Undefined	Yes	Used to handle undefined or unimplemented instructions
System	Yes	Uses the same registers as User mode

Figura 1.4: Stati del processore ARM

User	System	Fast Interrupt	Interrupt	Supervisor	Abort	Undefined
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	**R8_fiq**	R8	R8	R8	R8
R9	R9	**R9_fiq**	R9	R9	R9	R9
R10	R10	**R10_fiq**	R10	R10	R10	R10
R11	R11	**R11_fiq**	R11	R11	R11	R11
R12	R12	**R12_fiq**	R12	R12	R12	R12
R13 (SP)	R13 (SP)	**R13_fiq**	**R13_irq**	**R13_svc**	**R13_abt**	**R13_und**
R14 (LR)	R14 (LR)	**R14_fiq**	**R14_irq**	**R14_svc**	**R14_abt**	**R14_und**
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)
Program Status Registers						
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_fiq	**SPSR_irq**	**SPSR_svc**	**SPSR_abt**	**SPSR_und**

Figura 1.5: Registri duplicati (per modo operativo) in ARM

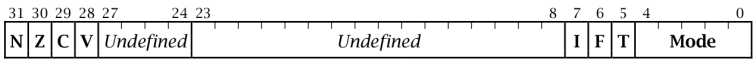

Figura 1.6: Contenuto del registro CPSR

stati dispone di alcuni registri duplicati. La Fig. 1.5 indica quali registri sono duplicati nei vari "modi" (stati) dei processori ARM. Vengono sempre messi a disposizione, nei vari modi che servono il trattamento delle interruzioni, registri duplicati per LR e SP. Questo significa che non occorre preoccuparsi di salvare, quando si passa ad uno di questi modi operativi, nessuno dei due registri. Questi registri vengono ripristinati quando si ritorna dal trattamento dell'interruzione che provoca il cambiamento di stato. Inoltre, tutti i modi mettono a disposizione un registro SPSR (Saved Program Status Register) che mantiene lo stato del CPSR al momento del passaggio di stato in modo che tale registro possa essere ripristinato al rientro dal trattamento dell'interruzione. Il modo Fast Interrupt mette a disposizione una copia anche dei registri general purpose da R8 a R12. Questo modo è stato previsto per quelle interruzioni che richiedono un trattamento molto breve e che necessita di pochi registri general purpose. Nel caso ne bastino 5, possiamo utilizzare durante il trattamento di una fast interrupt i registri R8–R12 senza bisogno di salvarli sullo stack: nel modo fast interrupt infatti usiamo le copie (grige nella figura) e non i registri originali.

In una gran parte dei casi, il codice eseguito per trattare un'interruzione è eseguito *ad interruzioni disabilitate*, ovvero facendo in modo che ulteriori interruzioni non possano essere prese in considerazione se non dopo che il trattamento dell'interruzione corrente sia effettivamente terminato.

Questo effetto si ottiene, in ARM, intervenendo sul registro CPSR (Current Program Status Register) che contiene due bit (I ed F) che, se messi a 1, rispettivamente mascherano (fanno in modo che non vengono trattate) le interruzioni normali e quelle "fast" (vedi oltre). Ci sono anche una serie di casi in cui un'interruzione può essere a sua volta interrotta da una interruzione il cui trattamento sia più urgente e semplicemente possa essere completato in modo molto veloce senza disturbare il trattamento dell'interruzione corrente. In questo caso occorre includere nel codice per il trattamento delle interruzioni "interrompibili" tecniche di programmazione che assicurano che non vengono persi (sovrascritti) dati in caso di interruzioni di interruzioni[5].

Le interruzioni vengono segnalate al processore utilizzando appositi pin del processore che portano all'unità controllo 1 o più bit che vengono utilizzati, quando sono messi a 1, per segnalare tipi diversi di interruzioni. Arm per esempio, ha a disposizione, fra gli altri, due segnali diversi per segnalare un'interruzione "normale" (bit IRQ) oppure un'interruzione "fast" (bit FIQ). Le inter-

[5] un po' come quando si programmano routing ricorsive e si utilizza lo stack invece che i registri per il passaggio dei parametri per fare in modo che siano supportate le chiamate ricorsive alla funzione

0	reset
4	undef instruction
8	sw interrupt (SVC)
12	abort (fecth da indirizzo illegale)
16	abort (ldr/str indirizzo illegale)
20	reserved
24	IRQ
28	FIQ

Figura 1.7: Tipo di interruzioni ARM

ruzioni di tipo fast vengono utilizzate per segnalare eventi che possono essere trattati molto rapidamente. Quelle di tipo normale richiedono tempi di trattamento più lunghi. Normalmente, a ciascuno dei due tipi di interruzioni possono corrispondere interruzioni generate da dispositivi diversi. Per esempio, dischi e schede di rete genereranno entrambe un'interruzione IRQ. Hardware dedicato provvederà a presentare al processore un unico segnale (da un bit) IRQ risultato dell'OR di tutti gli IRQ delle diverse unità di controllo delle periferiche e messo in AND con la negazione del bit `I` del registro CPSR. Il motivo dell'interruzione è di norma posto in un indirizzo particolare di memoria, noto alla routine che gestisce le interruzioni, che lo può quindi controllare per capire quale dei dispositivi ha veramente messo IRQ a 1.

In particolare, quando si verifica un'interruzione il processore, una volta completata l'esecuzione dell'istruzione corrente ed aggiornato lo stato del processore esegue una serie ben precisa di passi[6]:

- salva il valore del program counter corrente (è l'indirizzo di ritorno dall'interruzione, di fatto) nella copia del registro LR disponibile in ognuno dei modi operativi
- salva il CPSR nel SPSR
- setta il modo (stato) del processore nella CPSR (vedi Fig. 1.8). Per accedere le diverse parti del registro di stato vengono utilizzate due istruzioni particolari: la MRS (move status to register) che permette di copiare il contenuto del CPSR in un registro generale (per esempio, `MRS R0, CPSR` lo copia nel registro generale R0) in modo da poter successivamente manipolarne il contenuto (testare o settare/azzerare particolari bit o configurazioni di bit) e l'istruzione duale MSR (move register to status) che scrive un registro generale nel CPSR[7] (per esempio, `MRS CPSR, R0` copia il

[6] l'elenco che segue è sempre riferito all'architettura ARM, architetture differenti potrebbero eseguire azioni leggermente diverse o utilizzare indirizzi diversi

[7] versioni diverse del processore hanno modalità leggermente diverse di utilizzo di queste due istruzioni, per esempio nel modo utilizzato per nominare i registri di stato (CPSR, APSR, ...). Queste istruzioni potrebbero essere disponibili solo in certi "modi", a seconda della classe di processori ARM utilizzati

Mode Bits		Processor Mode	Accessible Registers
Bin	Hex	(Abbreviation)	
10000	10	User (usr)	PC, R14–R0, CPSR
10001	11	Fast Interrupt (fiq)	PC, R14_fiq–R8_fiq, R7–R0, CPSR, SPSR_fiq
10010	12	Interrupt (irq)	PC, R14_irq, R13_irq, R12–R0, CPSR, SPSR_irq
10011	13	Supervisor (svc)	PC, R14_svc, R13_svc, R12–R0, CPSR, SPSR_svc
10111	17	Abort (abt)	PC, R14_abt, R13_abt, R12–R0, CPSR, SPSR_abt
11011	1B	Undefined (und)	PC, R14_und, R13_und, R12–R0, CPSR, SPSR_und
11111	1F	System (sys)	PC, R14–R0, CPSR

Figura 1.8: Modi ARM (rappresentazione nel CPSR)

contenuto di R0 nel CPSR). In certi casi, sono disponibili modi per indicare flag particolari del CPSR come destinazioni delle MRS. Per esempio, `MSR CPSR_F, #1` setta il bit di mascheramento delle fast interrupt.

- setta il bit T della CPSR a 0 in modo che il processore esegua istruzioni normali anziché Thumb
- a seconda dei casi, disabilita interruzioni normali e/o fast settando i bit F e I del registro CPSR
- salta ad eseguire il codice che si trova all'indirizzo `0x0000000` + tipo dell'interruzione, dove il tipo dell'interruzione è definito come in Tab. 1.7.

Dualmente, il ritorno dal trattamento interruzioni esegue le seguenti azioni:

- sposta il contenuto del registro LR in PC[8];
- copia il registro SPSR in CPSR, cosa che ripristina il modo di funzionamento precedente all'interruzione, e
- azzera i bit di mascheramento delle istruzioni, in caso fossero stati settati a 1 (questi sono i bit I e F della parola di stato in CPSR).

Vediamo, per chiarire i concetti, come avviene il trattamento di una sw interrupt (il tipo di interruzione (eccezione, in verità) causato dall'esecuzione di una istruzione SVC 0).

L'istruzione SVC è rappresentata in memoria da 8 bit con il codice corrispondente all'istruzione e 24 bit utilizzati per rappresentare il parametro (imm24):

```
[COND|1111|     24 bit immediate   ]
```

L'esecuzione della `SVC` comporta i passi che abbiamo dettagliato poco sopra, al termine dei quali si salta ad eseguire l'istruzione all'indirizzo 0x00000008. Questa istruzione dovrebbe saltare immediatamente un pezzetto di codice simile al seguente:

```
1 LDR R1, [LR, -4]          ; carica in R1 la svc
2 BIC R1, #0xff000000       ; cancella 8 bit piu' significativa
3 LDR R0, =svcvec           ; base vettore con indirizzi svc[i]
4 LDR PC, [R0, R1, LSL #2] ; salto alla routine svc[i]
```

[8] in alcuni casi dopo averci sommato un piccolo offset