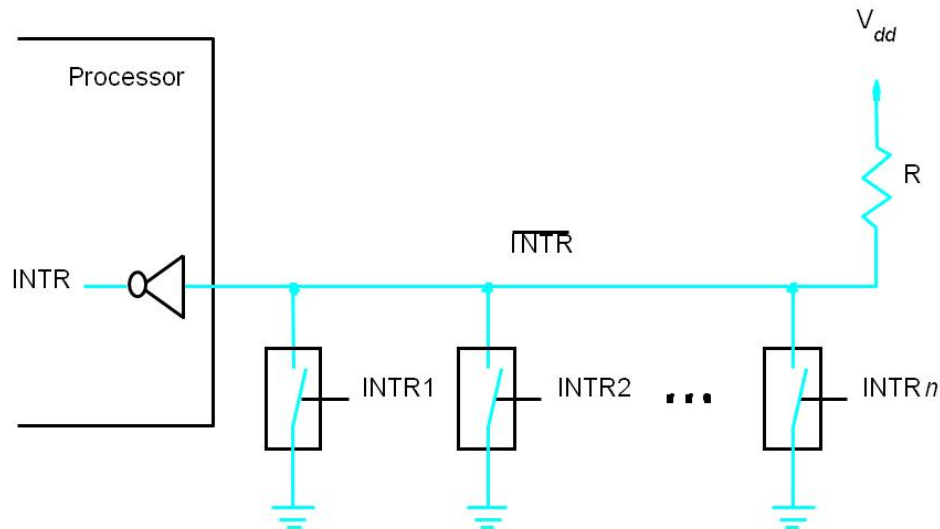


# Identificazione dei dispositivi (1/2)



- Se ci sono più dispositivi, il processore deve essere in grado di identificare il dispositivo che ha generato l'interruzione, poiché probabilmente diverse azioni dovranno essere intraprese a seconda del particolare dispositivo
- I dispositivi hanno una linea comune attraverso la quale segnalano richieste di interruzioni (INT)
- Quando INT è alto si pone il problema di identificare da quale dispositivo è partita la richiesta

## *Identificazione dei dispositivi (2/2)*

---

- INT potrebbe alzarsi anche in seguito a richieste “contemporanee” di due o più dispositivi
- Esistono diverse soluzioni a questo problema
- Tutte le soluzioni impiegano un misto di hardware e di software
- Tutte le soluzioni dipendono fortemente sia dall’architettura del sistema che da quella del processore

## *La soluzione a registri di stato*

---

- Una possibile soluzione consiste nel dotare ogni dispositivo di un registro di stato
- Quando un dispositivo richiede un'interruzione, inizializza un bit nel registro di stato, il bit di richiesta di interrupt (Interrupt Request, IRQ)
- La procedura di servizio inizia interrogando tutti i dispositivi in un certo ordine e, non appena trova un bit alto, fa partire la corrispondente routine di interrupt
- Questa interrogazione ciclica (polling) è semplice da realizzare, ma ha lo svantaggio di richiedere un certo tempo per interrogare anche i dispositivi che non hanno invocato alcun servizio

# *Gli interrupt vettorizzati*

---

- Un approccio alternativo consiste nel prevedere che sia il dispositivo stesso a fornire un proprio identificativo all'atto di una richiesta
- L'identificativo serve proprio a calcolare l'indirizzo della routine di interruzione che deve essere invocata, rendendo il meccanismo molto efficiente
- Il numero di bit utilizzati pone il limite massimo sul numero di diversi dispositivi che possono essere riconosciuti

# Enabling and Disabling Interrupts

---

- Must processor always respond *immediately* to interrupt requests from I/O devices?
- Some tasks cannot tolerate *interrupt latency* and must be completed without interruption
- Need ways to enable and disable interrupts, both in processor and in device interfaces
- Provides flexibility to programmers
- Use control bits in processor and I/O registers

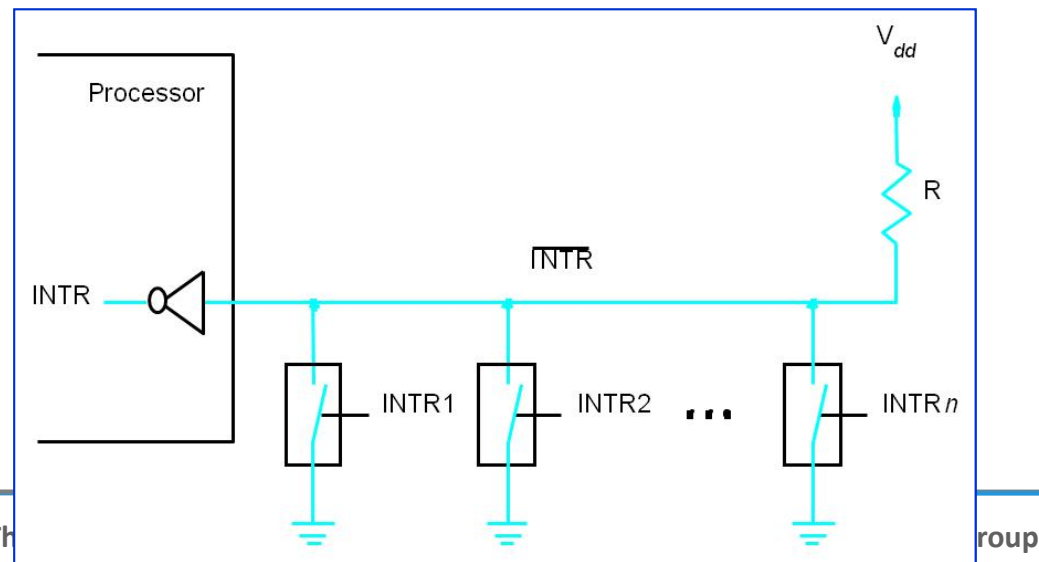
# *Event Sequence for an Interrupt*

---

- Processor status (PS) register has IE bit
- Program sets IE to 1 to enable interrupts
- When an interrupt is recognized, processor saves program counter and status register
- IE bit cleared to 0 so that same or other signal does not cause further interruptions
- After acknowledging and servicing interrupt, restore saved state, which sets IE to 1 again

# Handling Multiple Devices

- Which device is requesting service?
- How is appropriate service routine executed?
- Should interrupt nesting be permitted?
- How are two simultaneous requests handled?
- For 1<sup>st</sup> question, poll device status registers, checking if *IRQ* bit for each device is set
- For 2<sup>nd</sup> question, call device-specific routine for first set *IRQ* bit that is encountered



# Vectored Interrupts

---

- **Vectored interrupts** reduce service latency; no instructions executed to poll many devices
- Let requesting device identify itself directly with a signal or a binary code
- Processor uses info to find address of correct routine in an *interrupt-vector table*
- Table lookup is performed by hardware
- Vector table is located at fixed address, but routines can be located anywhere in memory



# Interrupt Nesting

---

- Service routines usually execute to completion
- To reduce latency, allow **interrupt nesting** by having service routines set IE bit to 1
- Acknowledge the current interrupt request *before* setting IE bit to prevent infinite loop
- For more control, use different priority levels
- Current level held in processor status register
- Accept requests only from higher-level devices

# *Simultaneous Requests*

---

- Two or more devices request at the same time
- Arbitration or priority resolution is required
- With software polling of I/O status registers, service order determined by polling order
- With vectored interrupts, hardware must select only one device to identify itself
- Use arbitration circuits that enforce desired priority or fairness across different devices

# Controlling I/O Device Behavior

---

- Processor IE bit setting affects all devices
- Desirable to have finer control with IE bit for each I/O device in its control register
- Such a control register also enables selecting the desired mode of operation for the device
- Access register with Load/Store instructions
- For example interfaces, setting KIE or DIE to 1 enables interrupts from keyboard or display

# Processor Control Registers

---

- In addition to a processor status (PS) register, other control registers are often present
- IPS register is where PS is automatically saved when an interrupt request is recognized
- IENABLE has one bit per device to control if requests from that source are recognized
- IPENDING has one bit per device to indicate if interrupt request has not yet been serviced



# Accessing Control Registers

---

- Use special Move instructions that transfer values to and from general-purpose registers
- Transfer pending interrupt requests to R4:  
MoveControl R4, IPENDING
- Transfer current processor IE setting to R2:  
MoveControl R2, PS
- Transfer desired bit pattern in R3 to IENABLE:  
MoveControl IENABLE, R3

# Examples of Interrupt Programs

---

- Use keyboard interrupts to read characters, but polling within service routine for display
- Illustrate initialization for interrupt programs, including data variables and control registers
- Show saving of registers in service routine
- Consider RISC-style and CISC-style programs
- We assume that predetermined location *ILOC* is address of 1<sup>st</sup> instruction in service routine

## Interrupt-service routine

ILOC:	Subtract Store Store Load LoadByte StoreByte Add Store	SP, SP, #8 R2, 4(SP) R3, (SP) R2, PNTR R3, KBD_DATA R3, (R2) R2, R2, #1 R2, PNTR	Save registers.  Load address pointer. Read character from keyboard. Write the character into memory and increment the pointer. Update the pointer in memory.
ECHO:	LoadByte And Branch_if_[R2]=0 StoreByte Move Branch_if_[R3]≠[R2] Move Store Clear StoreByte	R2, DISP_STATUS R2, R2, #4 ECHO R3, DISP_DATA R2, #CR RTRN R2, #1 R2, EOL R2 R2, KBD_CONT	Wait for display to become ready.  Display the character just read. ASCII code for Carriage Return. Return if not CR.  Indicate end of line. Disable interrupts in the keyboard interface.
RTRN:	Load Load Add Return-from-interrupt	R3, (SP) R2, 4(SP) SP, SP, #8	Restore registers.



---

## Main program

START:	Move	R2, #LINE	
	Store	R2, PNTR	Initialize buffer pointer.
	Clear	R2	
	Store	R2, EOL	Clear end-of-line indicator.
	Move	R2, #2	Enable interrupts in
	StoreByte	R2, KBD_CONT	the keyboard interface.
	MoveControl	R2, IENABLE	
	Or	R2, R2, #2	Enable keyboard interrupts in
	MoveControl	IENABLE, R2	the processor control register.
	MoveControl	R2, PS	
	Or	R2, R2, #1	
	MoveControl	PS, R2	Set interrupt-enable bit in PS.
	next instruction		



## Interrupt-service routine

ILOC:	Move	– (SP), R2	Save register.
	Move	R2, PNTR	Load address pointer.
	MoveByte	(R2), KBD_DATA	Write the character into memory
	Add	PNTR, #1	and increment the pointer.
ECHO:	TestBit	DISP_STATUS, #2	Wait for the display to become ready.
	Branch=0	ECHO	
	MoveByte	DISP_DATA, (R2)	Display the character just read.
	CompareByte	(R2), #CR	Check if the character just read is CR.
	Branch≠0	RTRN	Return if not CR.
	Move	EOL, #1	Indicate end of line.
	ClearBit	KBD_CONT, #1	Disable interrupts in keyboard interface.
RTRN:	Move	R2, (SP)+	Restore register.
	Return-from-interrupt		

## Main program

START:	Move	PNTR, #LINE	Initialize buffer pointer.
	Clear	EOL	Clear end-of-line indicator.
	SetBit	KBD_CONT, #1	Enable interrupts in keyboard interface.
	Move	R2, #2	Enable keyboard interrupts in
	MoveControl	IENABLE, R2	the processor control register.
	MoveControl	R2, PS	
	Or	R2, #1	
	MoveControl	PS, R2	Set interrupt-enable bit in PS.
	next instruction		



# Multiple Interrupt Sources

---

- To use interrupts for both keyboard & display, call subroutines from ILOC service routine
- Service routine reads IPENDING register
- Checks which device bit(s) is (are) set to determine which subroutine(s) to call
- Service routine must save/restore Link register
- Also need separate pointer variable to indicate output character for next display interrupt

## — Interrupt handler

ILOC:	Subtract	SP, SP, #12	Save registers.
	Store	LINK_reg, 8(SP)	
	Store	R2, 4(SP)	
	Store	R3, (SP)	
	MoveControl	R2, IPENDING	Check contents of IPENDING.
	And	R3, R2, #4	Check if display raised the request.
	Branch_if_[R3]=0	TESTKBD	If not, check if keyboard.
	Call	DISR	Call the display ISR.
TESTKBD:	And	R3, R2, #2	Check if keyboard raised the request.
	Branch_if_[R3]=0	NEXT	If not, then check next device.
	Call	KISR	Call the keyboard ISR.
NEXT:	...		Check for other interrupts.
	Load	R3, (SP)	Restore registers.
	Load	R2, 4(SP)	
	Load	LINK_reg, 8(SP)	
	Add	SP, SP, #12	
	Return-from-interrupt		



# Exceptions

---

- An **exception** is any interruption of execution
- This includes interrupts for I/O transfers
- But there are also other types of exceptions
- *Recovery from errors*: detect division by zero, or instruction with an invalid OP code
- *Debugging*: use of trace mode & breakpoints
- *Operating system*: software interrupt to enter
- The last two cases are discussed in Chapter 4

# Recovery from Errors

---

- After saving state, service routine is executed
- Routine can attempt to recover (if possible) or inform user, perhaps ending execution
- With I/O interrupt, instruction being executed at the time of request is allowed to complete
- If the instruction is the *cause* of the exception, service routine must be executed immediately
- Thus, return address may need adjustment

# Concluding Remarks

---

- Two basic I/O-handling approaches:  
*program-controlled* and *interrupt-based*
- 1<sup>st</sup> approach has direct control of I/O transfers
- Drawback: wait loop to poll flag in status reg.
- 2<sup>nd</sup> approach suspends program when needed to service I/O interrupt with separate routine
- Until then, processor performs useful tasks
- Exceptions cover all interrupts including I/O



## Interrupt management: ARM



# *Gestione interruzioni nel processore ARM*

---

- Il processore ARM ha un'architettura orientata al controllo processo e alla realizzazione di sistemi embedded.
- Tale peculiarità, e la necessità di fornire elevati livelli prestazionali a costi ridotti, fa sì che l'architettura, di tipo RISC, si presenta scarsamente ortogonale. In essa si ricorre spesso a soluzioni particolari per realizzare figure di programmazione a livello di linguaggio macchina volte a concentrare in un'unica istruzione, più operazioni.

# Processore ARM

---

- Il processore ARM ricorre al suo interno ad una pipe a 3 stadi in cui sono eseguite, simultaneamente, le operazioni di fetch, decode e esecuzione.
- Il PC punta sempre all'istruzione che deve essere prelevata dalla memoria (fetch).
- Essendo un'istruzione codificata completamente su 4 byte (non sono ammesse istruzioni a lunghezza variabile) ad ogni ciclo di v.N si ha che nella pipe sono presenti le seguenti tre istruzioni:

Fetch  
(PC)

Decode  
(PC-4)

Execute  
(PC-8)

# Il processore ARM gestisce i seguenti sette differenti tipi di eccezioni

Tipo di eccezione	Stato	Indirizzo vettore	Priorità	descrizione
Reset	Svc	\$0	1	Evento Hw asincrono. Occorre quando il pin di Reset del processore è attivato. Tipicamente è attivato all'atto dell'accensione o se si vuole attivare un software reset.
Undefined instruction	Undef	\$4	6	Evento Sw sincrono. Occorre se la CPU (o uno dei coprocessori ad essa associati) non riconosce valido il codice operativo corrente
Software Interrupt (SWI)	Svc	\$8	6	Evento Sw sincrono. E' generato mediante l'esecuzione di una istruzione, SWI che codifica in essa un parametro. Consente di passare allo stato Svc. Usata dai S.Op. per gestire le system call e dai monitor per eseguire funzioni.
Prefetch Abort	Abort	\$C	5	Evento Sw sincrono. Occorre se si tenta di eseguire un'istruzione che è stata prelevata da un indirizzo illecito relativamente allo stato corrente dell'elaborazione.
Data Abort	Abort	\$10	2	Evento Sw sincrono. Occorre se si tenta di eseguire un'operazione di load/store ad un indirizzo illecito relativamente allo stato corrente dell'elaborazione
Reserved	-	\$14	-	Riservato per future implementazione
Interrupt Request (IRQ)	Irq	\$18	4	Evento Hw asincrono. Occorre quando il pin IRQ del processore è attivato (basso) e il bit I del registro di stato CPSR è basso
Fast Interrupt Request (FIQ)	Fiq	\$1C	3	Evento Hw asincrono. Occorre quando il pin FIQ del processore è attivato (basso) e il bit F del registro di stato CPSR è basso

# Attivazione delle ISR

---

- Il processore ARM impiega per la gestione delle interruzioni uno schema semplice basato su di una forma di vettorizzazione utilizzata anche nel processore Intel della famiglia 8080 e nella famiglia Motorola 6800. L'indirizzo della ISR associata all'evento interrompente è vettorizzato in locazioni di memoria staticamente associate allo stesso (gli indirizzi di tali locazioni per l'ARM sono riportate nella terza colonna della precedente tabella). In tali indirizzi taluni processori pongono direttamente il vettore che punta all'indirizzo assoluto della ISR da eseguire (pseudo vettori), altri un codice operativo, tipicamente un codice di salto al codice dell'ISR interessata. Il processore ARM opera in tal'ultimo modo e nei vettori è posto, in genere, un codice di salto (BL) o di load PC.
- Quando arriva un'eccezione il pc può essere stato aggiornato o meno a seconda dello stato della pipe

# Eccezioni e banked registers

---

- L'attivazione di una eccezione cambia la modalità di esecuzione
- Di conseguenza, ogni *handler* di eccezione ha accesso ad un particolare gruppo di banked registers:
  - r13 o *Stack Pointer* (indicato con `SP_<mode>` )
  - r14 o *Link Register* ( `LR_<mode>` )
  - il *Saved Program SR* ( `SPSR_<mode>` )
  - solo nel caso FIQ, r8 - r12 ( `r8_FIQ - r12_FIQ` )

# *Risposta ad un'eccezione*

---

- salvataggio del CPSR nell'SPSR della modalità nella quale verrà servita l'eccezione
- settaggio di CPSR (cambio modalità ed eventuale mascheramento di ulteriori interrupt)
- salvataggio dell'indirizzo di ritorno nel LR\_<mode>
- aggiornamento del Program Counter

# Ritorno da un'eccezione

---

- ripristino di CPSR da SPSR\_<mode>
- ripristino del program counter da LR\_<mode>

Queste due operazioni devono essere necessariamente effettuate in maniera *atomica*

(si usa il flag *S* o il qualificatore  $\wedge$  in modalità privilegiata per segnalare questo meccanismo)

# *Calcolo dell'indirizzo di ritorno (1)*

---

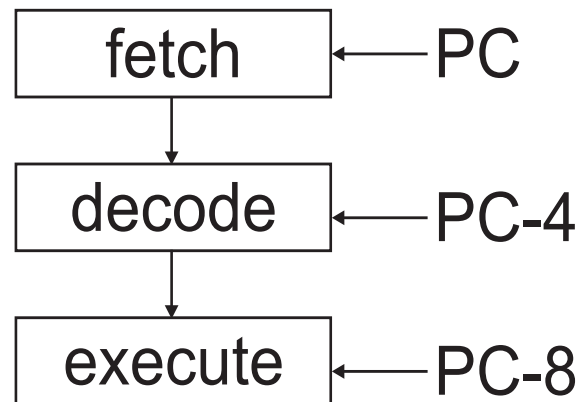
- L'indirizzo di ritorno da un'eccezione dipende dal tipo di eccezione servita
- Per comprendere tali differenze, occorre avere presente la pipeline usata da ARM



## Calcolo dell'indirizzo di ritorno (2)

---

- ARM usa una pipeline a tre stadi (fetch, decode, execute). In ogni stadio è presente un'istruzione
- il PC punta all'istruzione presente nello stadio di fetch, posta due locazioni di memoria dopo l'istruzione correntemente eseguita



## Calcolo dell'indirizzo di ritorno (3)

---

- Eccezioni SWI e “Istruzione non-definita”
  - l'eccezione è generata dall'istruzione durante la decodifica, quindi il PC corrente coincide con il corretto indirizzo di ritorno
- L'ultima istruzione dell'exception handler sarà pertanto:

**MOVS pc, lr**

## Calcolo dell'indirizzo di ritorno (4)

---

- Eccezioni FIQ e IRQ
  - ARM, come molti altri processori, controlla la presenza di richieste di interrupt **dopo** aver eseguito l'istruzione corrente, ovvero dopo aver aggiornato il PC alla successiva locazione di memoria.
- Perché assuma il corretto valore di ritorno, il PC dovrà dunque essere decrementato di 4 al momento del ritorno dall'exception handler:

**SUBS pc, lr, #4**

## Calcolo dell'indirizzo di ritorno (5)

---

- Eccezione “Prefetch Abort”
  - ARM controlla la validità dell'istruzione caricata non nello stadio di fetch, ma durante l'esecuzione. Dunque è in questo stadio che l'eventuale eccezione è sollevata.
- Come nei casi FIQ e IRQ il corretto valore di ritorno sarà dato da  
**SUBS pc, lr, #4**

## Calcolo dell'indirizzo di ritorno (6)

---

- Eccezione “Data Abort”
  - Una istruzione di Load o Store ha per effetto l'accesso in memoria *dopo* lo stadio di execute, quando il PC è stato ulteriormente aggiornato
- il PC va dunque decrementato di 8

**SUBS pc, lr, #8**

# Installazione di un Exception Handler (1)

---

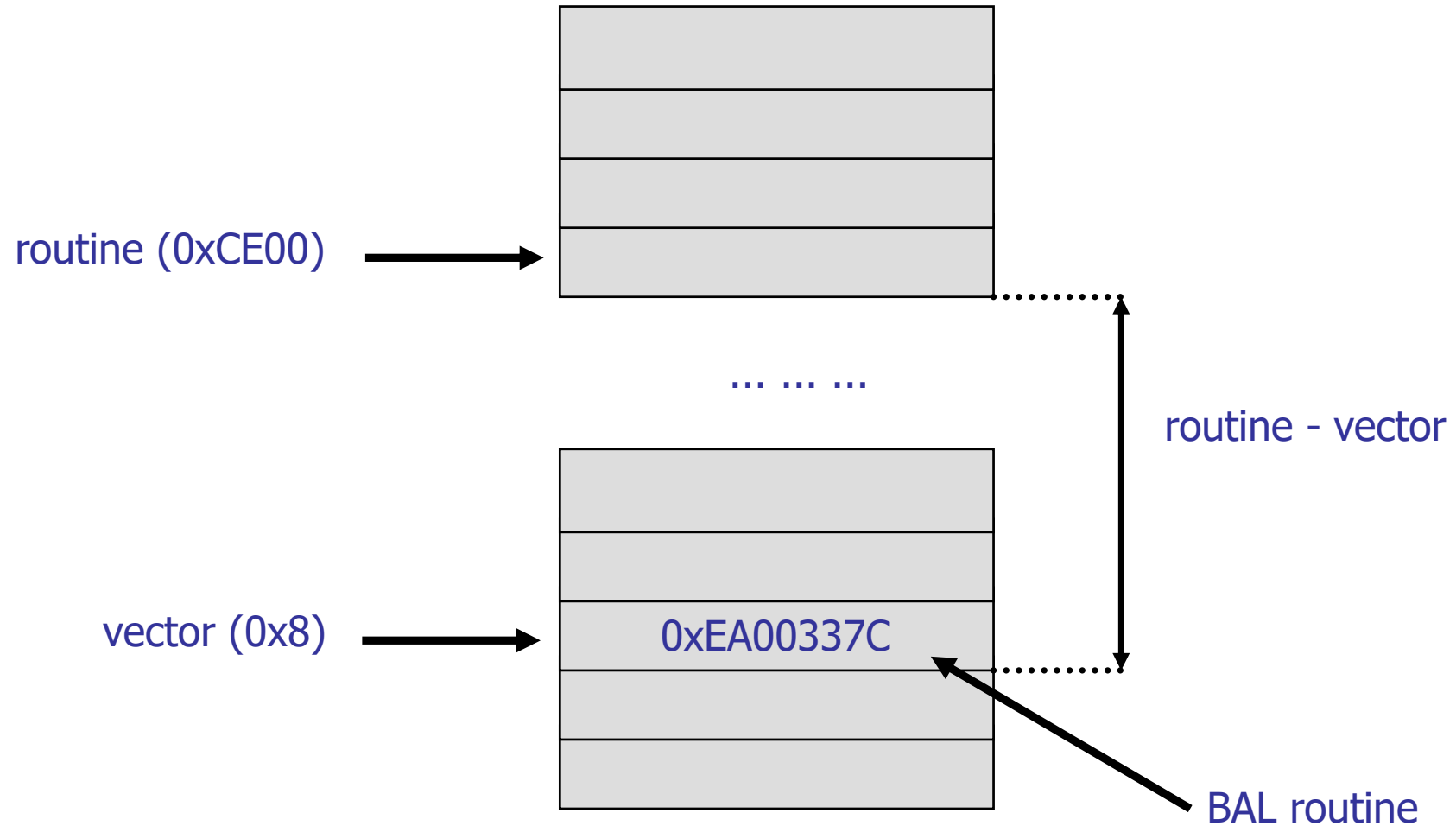
- Normalmente, nella *vector table* sono presenti per ogni vettore delle istruzioni di salto (FIQ può fare eccezione per eseguire direttamente il codice)
- Per installare un handler occorre:
  - ottenere l'indirizzo dell'exception handler
  - sottrargli l'indirizzo del corrispondente vettore
  - sottrarre #8 (per tenere in conto la pipeline)
  - shiftare a destra il risultato di 2 bit (per indirizzare le word)
  - controllare che il risultato sia rappresentabile su 24 bit
  - farne la OR logica con 0xEA000000 per avere una BAL

## Installazione di un Exception Handler (2)

---

```
unsigned Install_Handler (unsigned routine, unsigned
    *vector)
{ unsigned vec, oldvec;
  vec = ((routine - (unsigned)vector - 0x8)>>2);
  if (vec & 0xff000000)
  { printf ("installazione dell'handler fallita...");
    exit (0);
  }
  vec = 0xea000000 | vec;
  oldvec = *vector;
  *vector = vec;
  return (oldvec);
}
```

# Installazione di un Exception Handler (3)

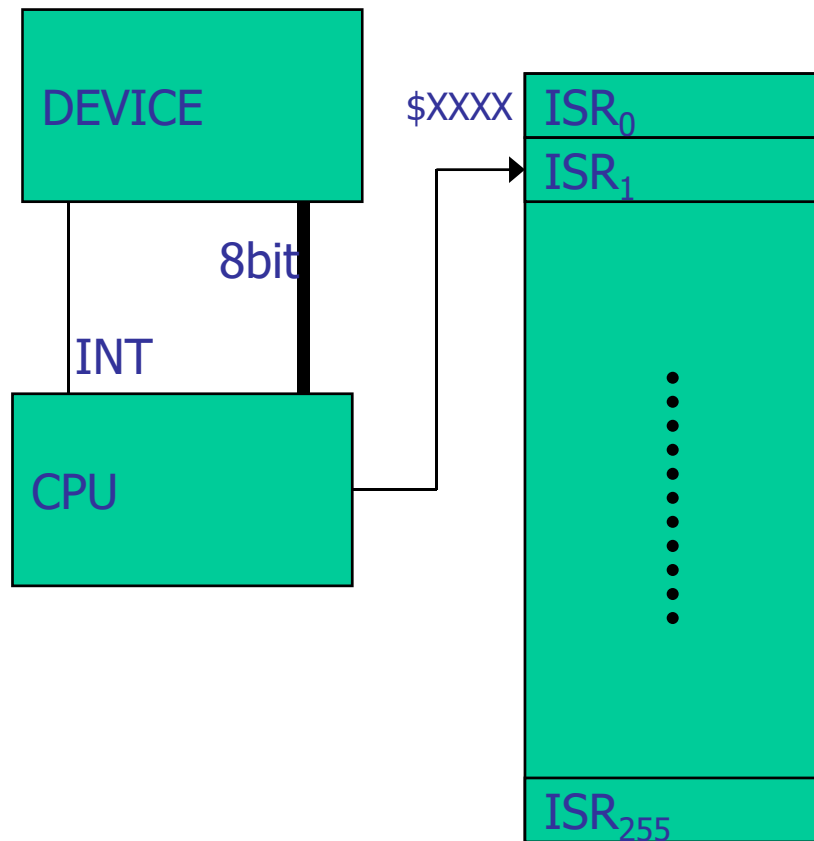






## Interrupt management: Coldfire

# La soluzione del M68000

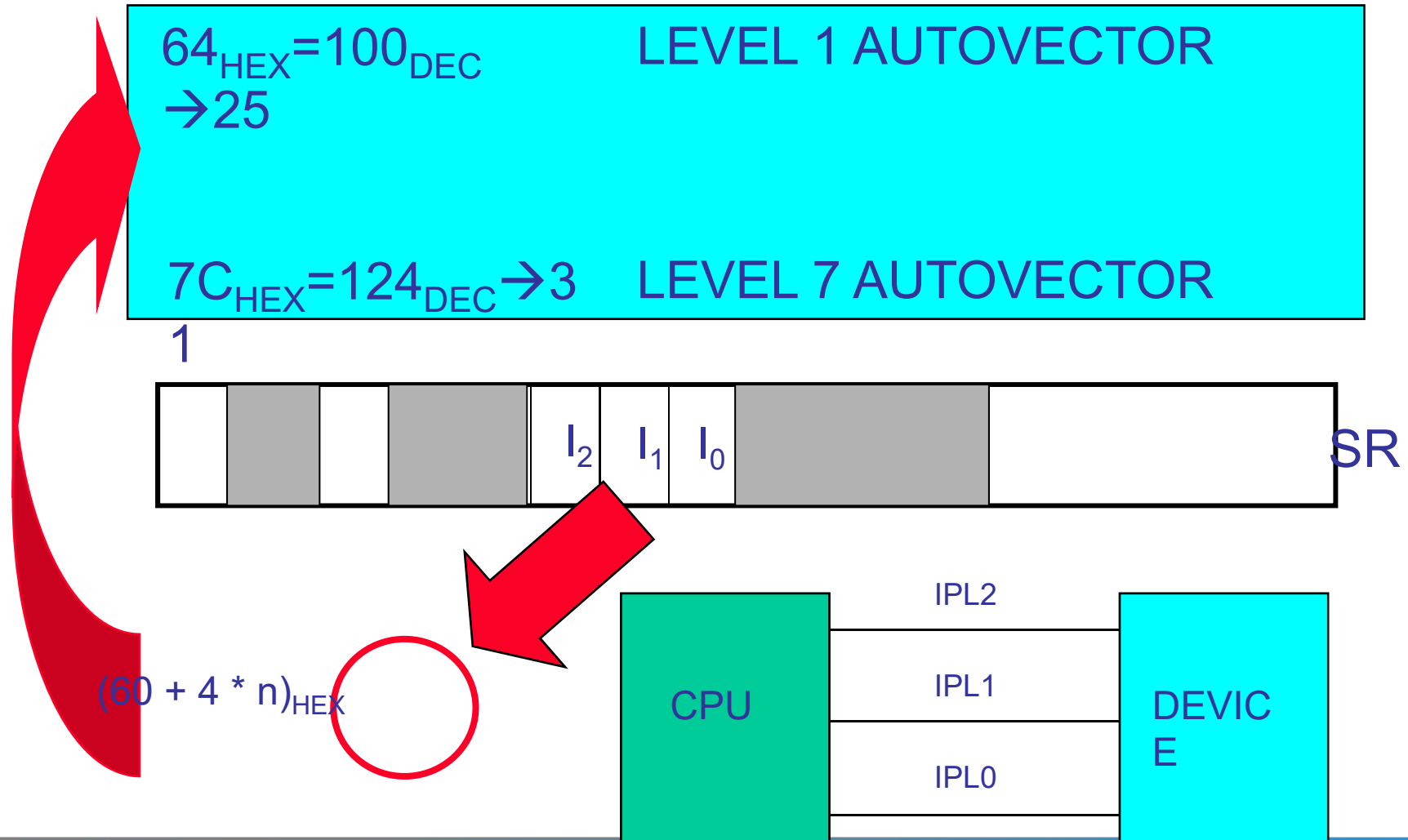


- Il processore M68000 utilizza il meccanismo degli interrupt vettorizzati
- In memoria sono presenti 256 locazioni consecutive dette *vettori di interrupt*
- Ciascuna di queste locazioni contiene l'indirizzo di una ISR
- Quando un dispositivo richiede un'interrupt, invia al processore un numero di 8 bit che rappresenta il *vettore di interrupt* da utilizzare

# Exception Vector Table

0	RESET (SSP)
1	RESET (PC)
16-23	UNASSIGNED, RESERVED
25	LEVEL 1 AUTOVECTOR
31	LEVEL 7 AUTOVECTOR
32-47	TRAP #0-15 INSTRUCTIONS
64-255	USER DEVICE INTERRUPTS

# Servizio mediante autovettore



# *Classificazione delle eccezioni nel MC68000*

---

- gruppo 0: reset, bus error, address error
- gruppo 1: trace, interrupt, privilege violation, illegal instruction
- gruppo 2: TRAP, TRAPV, CHK, divisione per zero

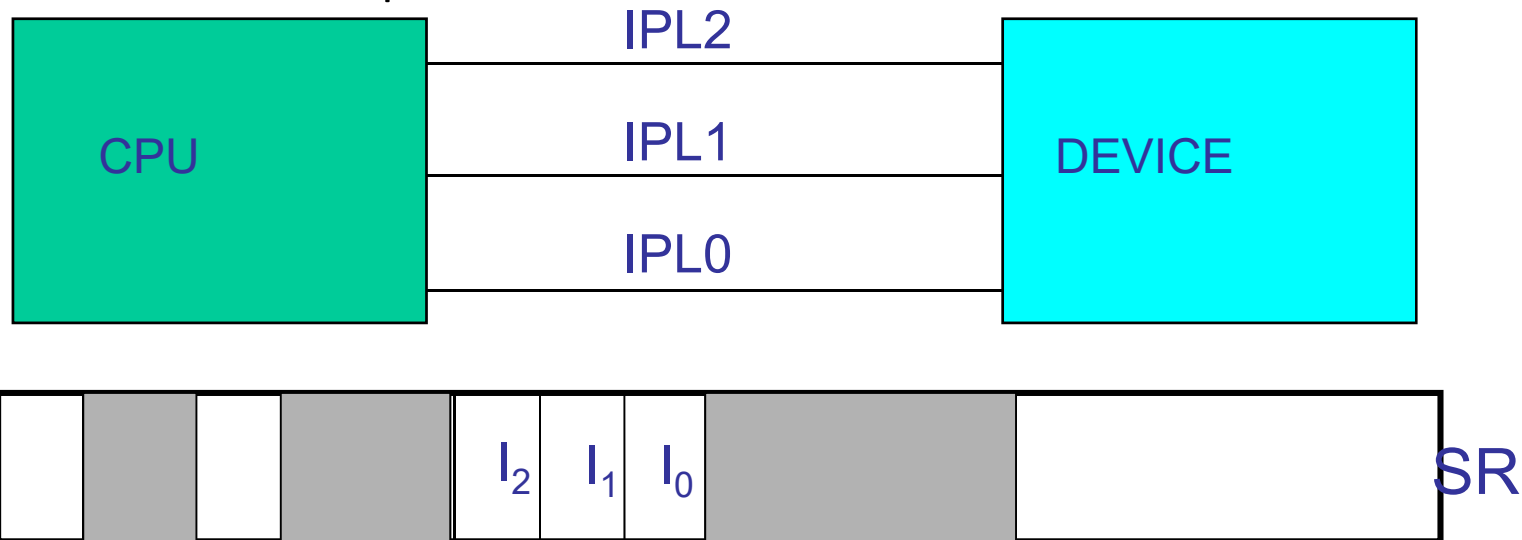
# Gestione delle priorità

## ➤ Problemi:

- Mascheramento
- Abilitazione

## ➤ Soluzione del 68K:

- Interrupt Priority Level
- Processor Priority Level
- Le interruzioni a priorità 7 non sono mascherabili



# *Il sistema delle eccezioni del processore MC68000*

---

- Le interruzioni sono abilitate se PI (Priority Interrupt) > livello maschera
- PPI process priority interrupt (in status word maschera)
- IPL interrupt priority level
- Interruzioni abilitate se  $IPL > PPI$ , inibite se  $IPL \leq PPI$

# Il sistema delle eccezioni del processore MC68000

---

- All'atto del servizio  $PPI=IPL$  che disabilita le interruzioni del livello corrente. Le interruzioni sono totalmente abilitate se  $PPI=0$ , mascherate se  $PPI=7$  eccetto le NMI (Non Maskerale Interrupt). I due estremi 0 e 7 rappresentano due condizioni particolari:
- $IPL=0$  rappresenta la base neutra su cui si poggiano le informazioni codificate dei segnali di interruzione. A tale valore corrisponde, quindi, la condizione di assenza di interruzioni. La presenza di interruzione si ha se  $IPL>0$ , cioè a partire da  $IPL=1$  e a finire a  $IPL=7$ . Il valore  $PPI=0$  della maschera abilita, pertanto, tutte le interruzioni.
- $IPL=7$  è considerato NMI (non maskerale interrupt). Tale richiesta di interruzione è sempre accettata dal processore, anche se  $PPI=7$ . Le NMI operano, infatti, in eccezione alla regola  $IPL>PPI$  valida per la gestione di interruzioni di livello generico.



# *Il sistema delle eccezioni del processore MC68000*

---

- I segnali di interruzione con IPL da 1 a 6 sono a livello. La permanenza sulle linee di interruzione di uno stato alto (o basso) è considerata, dal processore, interruzione. Il segnale IPL=7 di NMI è di tipo “edge triggered” e cioè sensibili alla variazione di fronte. Ciò per evitare che il segnale di interruzione corrente di IPL=7, con il suo permanere, venga reinterpretato dal processore come ulteriore richiesta di interruzione, stante il comportamento diverso del confronto fra maschera e segnali di richiesta che, per IPL=7, non segue la regola  $IPL > PPL$  ma  $IPL \leq PPL$ , trattandosi di NMI, interruzioni sempre abilitate e non mascherabili.
- L’essere “edge triggered” fa sì che la sorgente, per generare una nuova interruzione, deve effettuare una variazione di livello.

# *Il sistema delle eccezioni del processore MC68000*

---

- Le interruzioni esterne IPL sono asincrone rispetto all'esecuzione dei codici operativi di un programma, ma servono alla fine del codice corrente all'atto del loro arrivo. L'esecuzione dei codici operativi non è interrompibile. In caso contrario si richiederebbe, infatti, il salvataggio non del semplice contesto elaborativo ma del contesto legato allo stato delle microoperazioni (Datapath) e, pertanto, caratterizzato da un elevato numero di informazioni.
- Sono prese in considerazione, all'interno di una esecuzione, particolari tipi di interruzioni legate ad eventi che non richiedono il ripristino dell'elaborazione interrotta e quindi il salvataggio del contesto (ad esempio interruzioni generate da una caduta di tensione d'alimentazione).

# *Il sistema delle eccezioni del processore MC68000*

---

- Le TRAP (o software interrupt) sono interruzioni a tutti gli effetti. L'unica differenza è che essendo generate da programma, sono sincrone. Al ripetersi del programma occorrono sempre nello stesso istante di tempo. Ciò, ovviamente, non è vero per i segnali di interruzione esterni.

NIOS II	ARM	Commenti
<b>Routine di servizio delle interruzioni</b>		
ILOC: subi sp, sp, 16 stw r2, 12(sp) stw r3, 8(sp) stw r4, 4(sp) stw r5, (sp) movia r2, PNTR ldw r3, (r2) movia r4, KBD ldbio r5, (r4) stb r5, (r3) addi r3, r3, 1 stw r3, (r2)	ILOC STMFD R13!, {R2, R3}	Salva i registri
	LDR R2, PNTR	Puntatore all'indirizzo
	LDRB R3, KBD_DATA	Leggi un carattere da tastiera,
	STRB R3, [R2], #1	immagazzinalo in memoria e incrementa il puntatore, aggiornandolo in memoria
	STR R2, PNTR	
ECO: movia r2, DISP ldbio r3, 4(r2) andi r3, r3, 4 beq r3, r0, ECO stbio r5, (r2) addi r3, r0, 0x0D bne r5, r3, RTRN movi r3, 1 movia r5, EOL stw r3, (r5) stbio r0, 8(r4)	ECO LDR BR2, DISP_STATUS	Attendi che lo schermo sia pronto, controllandone il bit di stato DOUT
	TST R2, #4	
	BEQ ECO	
	STRB R3, DISP_DATA	Invia carattere allo schermo, controlla se sia CR e rientra se non lo è, altrimenti segnala la fine della linea
	CMP R3, #CR	
	BNE RTRN	
	MOV R2, #1	
	STR R2, EOL	
	MOV R2, #0	Disabilita le interruzioni nell'interfaccia di tastiera
	STRB R2, KBD_CONT	
RTRN: ldw r5, (sp) ldw r4, 4(sp) ldw r3, 8(sp) ldw r2, 12(sp) addi sp, sp, 16 subi ea, ea, 4 eret	RTRN LDMFD R13!, {R2, R3}	Ripristina i registri
	SUBS R15, R14, #4	Aggiusta indirizzo di rientro e rientra dall'interruzione

**Figura A3.3** Programmi NIOS II e ARM per la lettura e visualizzazione di una linea di caratteri su interruzioni. (Segue)



NIOS II	ARM	Commenti
<b>Programma principale</b>		
START: movia r2, LINE		
movia r3, PNTR		
stw r2, (r3)	LDR R2, =LINE	Inizializza puntatore al buffer
movia r2, EOL	STR R2, PNTR	
stw r0, (r2)	MOV R2, #0	Azzerà indicatore di fine linea
	STR R2, EOL	
movia r2, KBD		
movi r3, 2	MOV R2, #2	Abilita le interruzioni nella
stbio r3,8(r2)	STRB R2, KBD_CONT	interfaccia della tastiera
rdctl r2, ienable		Abilita le sue interruzioni nel
ori r2, r2, 2		registro di controllo del
wrcctl ienable, r2		processore
rdctl r2, status		Abilita le interruzioni nel
ori r2, r2, 1	MOV R2, #&50	registro di stato del
wrcctl status, r2	MSR CPSR, R2	programma
...	...	Prossima istruzione

**Figura A3.3** (continua).

ColdFire	IA-32	Commenti
<b>Routine di servizio delle interruzioni</b>		
ILOC: MOVE.L A2, (A7) MOVE.L D0, (A7) MOVEA.L PNTR, A2  CLR.L D0  MOVE.B KBD_DATA, D0 MOVE.B D0, (A2)+ MOVE.L A2, PNTR  MOVEA.L #DISP_STATUS, A2	ILOC: PUSH EAX PUSH EBX MOV EBX, PNTR   MOV AL, KBD_DATA MOV [EBX], AL INC DWORD PTR PNTR     BT DISP_STATUS, 2 JNC ECO MOV DISP_DATA, AL CMP AL, CR JNE RTRN MOV DWORD PTR EOL, 1  BTR KBD_CONT, 1	Salva i registri  Inizializza il puntatore all'indirizzo  Azzera il registro dati destinato ai caratteri  Trasferisci il carattere al registro (ciò azzera KIN)  Immagazzina il carattere in memoria e aggiorna il puntatore in memoria  Indirizzo del registro di stato dello schermo  Attendi che lo schermo sia pronto  Invia il carattere allo schermo (ciò azzera DOUT)  Controlla se il carattere letto sia CR  Rientra se non lo è  Segnala la fine della linea  Indirizzo del registro di controllo della tastiera  Disabilita interruzioni nell'interfaccia della tastiera
ECO: BTST.B #2, (A2) BEQ ECO MOVE.B D0, DISP_DATA CMP.L #CR, D0 BNE RTRN ADDQ.L #1, EOL  MOVEA.L #KBD_CONT, A2 BCLR.B #1, (A2)	ECO: BT DISP_STATUS, 2 JNC ECO MOV DISP_DATA, AL CMP AL, CR JNE RTRN MOV DWORD PTR EOL, 1  BTR KBD_CONT, 1	Attendi che lo schermo sia pronto  Invia il carattere allo schermo (ciò azzera DOUT)  Controlla se il carattere letto sia CR  Rientra se non lo è  Segnala la fine della linea  Indirizzo del registro di controllo della tastiera  Disabilita interruzioni nell'interfaccia della tastiera
RTRN: MOVE.L (A7)+, D0 MOVEA.L (A7)+, A2 RTE	RTRN: POP EBX POP EAX IRET	Ripristina i registri  Rientra dall'interruzione

**Figura A3.4** Programmi ColdFire e IA-32 per la lettura e visualizzazione di una linea di caratteri su interruzioni. (Segue)



ColdFire	IA-32	Commenti
<b>Programma principale</b>		
START: MOVEA.L #LINE, A0	MOV DWORD PTR PNTR, OFFSET LINE	Inizializza puntatore al buffer
MOVE.L A0, PNTR		
CLR.L EOL	MOV DWORD PTR EOL, 0	Azzera indicatore di fine linea
MOVEA.L #KBD_CONT, A0		
BSET.B #1, (A0)	BTS KBD_CONT, 1	Abilita le interruzioni nell'interfaccia della tastiera
MOVE.W SR, D0		Carica lo stato del programma in D0
ANDI.L #\$F8FF, D0		Azzera la priorità nella maschera delle interruzioni
MOVE.W D0, SR	STI	Abilita le interruzioni nel processore
...	...	Prossima istruzione

**Figura A3.4** (continua).

**Tabella A3.1** Tipi di eccezioni e loro caratteristiche nei processori ARM.

<b>Tipo di eccezione</b>	<b>P</b>	<b>Modo</b>	<b>VA</b>	<b>Istruzione di rientro</b>
Interruzione di I/O ordinaria	4	IRQ	28	SUBS R15, R14, #4
Interruzione di I/O urgente	3	FIQ	24	SUBS R15, R14, #4
Interruzione software	–	SVC	8	MOVS R15, R14
Avvio/reset	1	SVC	0	–
Errore di accesso a operando	2	Abort	16	SUBS R15, R14, #8
Errore di accesso a istruzione	5	Abort	12	SUBS R15, R14, #4
Istruzione non definita	6	Undefined	4	MOVS R15, R14

Legenda:

- P Priorità (massima = 1);
- VA Indirizzo del vettore di eccezione;
- IRQ Interrupt request;
- FIQ Fast Interrupt Request;
- SVC Supervisor Call.





## Contact Info