Interruzioni

G. Lettieri

2 Aprile 2022

Il processore che abbiamo visto fino ad ora esegue un flusso di istruzioni dettato da un unico programma: ogni istruzione del programma ordina al processore sia quali operazioni deve compiere, sia qual è l'istruzione successiva. In particolare, ogni routine del programma va in esecuzione solo perché è stata esplicitamente invocata da una istruzione precedente nel flusso sequenziale, che le ha trasferito "volontariamente" il controllo. Vogliamo ora aggiungere un meccanismo nuovo: il programmatore può associare l'esecuzione di una routine al verificarsi di un *evento*. Più in generale, il sistema definisce un insieme (finito) di eventi e_1, e_2, ..., e_n e permette al programmatore di associarvi delle routine, chiamiamole r_1, r_2, ..., r_n, con r_i associata all'evento e_i, per $1 \leq i \leq n$. Per programmare il sistema il programmatore deve ora scrivere un programma principale, chiamiamolo p, ma può anche scrivere le routine r_1, ..., r_n e associarle agli eventi e_1, ..., e_n (o un sottoinsieme di essi, in base alle proprie necessità) Il processore partirà eseguendo p e obbedendo alle istruzioni di p, in sequenza, esattamente come prima. Mentre esegue p, però, il processore controllorà se si è verificato uno degli eventi previsti e, in tal caso, salterà automaticamente alla routine associata, *interrompendo* il programma p. L'idea è che l'interruzione sia comunque momentanea e che, al termine della routine, l'esecuzione ritorni dal programma p, dal punto in cui era stato interrotto.

Per capire perché potrebbe servirci un meccanismo del genere, e a che tipo di eventi potremmo essere interessati, ispiriamoci ad un esempio che fece uno dei primi ricercatori che si occupò di studiare le interruzioni—Edsger Dijkstra[1] Immaginiamo di dover scrivere un programma che deve calcolare i valori di una funzione $f(x)$ per vari valori di x (il tipico problema per cui i computer sono stati inventati), in modo da stampare una tabella dei valori della funzione:

x	$\sin(x)$
0.1000	0.0998
0.1001	0.0999
0.1002	0.1000
...	

Per guadagnare tempo, ci piacerebbe che, mentre è in corso la stampa dell'ultimo valore calcolato, per esempio per $x = 0.1001$, il nostro programma

[1] L'esempio originale si trova qui: `https://www.cs.utexas.edu/users/EWD/transcriptions/EWD13xx/EWD1303.html`.

potesse andare avanti a calcolare il successivo valore, per $x = 0.1002$. In questo modo stampa e calcoli procederebbero in parallelo. Come possiamo fare? Supponiamo di avere un dispositivo di uscita (una stampante) con i due classici registri TBR (Transmit Buffer Register) e un registro di stato STS. Per stampare un carattere ne dobbiamo scrivere il codice ASCII nel TBR. La stampante impiega un certo tempo per stampare il carattere, e per tutto quel tempo non può accettare un nuovo carattere in TRB. Un bit READY del registro di stato ci dice quando la stampante è pronta a ricevere un nuovo carattere. Il nostro programma dovrebbe essere organizzato così: quando ha finito di calcolare un valore di $f(x)$, prepara la riga da stampare in un buffer in memoria, aspetta che la stampante sia pronta (legge ripetutamente STS) e poi scrive il primo carattere. A questo punto, invece di aspettare che la stampante sia pronta a ricevere il prossimo carattere, prosegue con il calcolo del secondo valore. Qui viene la difficoltà: la routine che esegue i calcoli, ogni tanto, deve leggere STS e, se trova la stampante pronta, deve scrivere in TBR il prossimo carattere preso dal buffer, poi proseguire con i calcoli. Ogni quanto bisogna inserire le istruzioni che controllano STS? Se lo si fa troppo spesso si rallenta il calcolo del prossimo valore, se lo si fa troppo raramente si rallenta la stampa. Anche se troviamo una frequenza ideale, potrebbe non esserci una soluzione ottima: che succede se nella routine che fa i calcoli c'è un lungo ciclo con un corpo di poche istruzioni?

```
// controllo STS?
for (int i = 0; i < 1000000; i++) {
    // controllo STS?
    v[i]++;
}
// controllo STS?
```

Inserendo il controllo di STS solo fuori dal ciclo si rischia di far passare troppo tempo tra un controllo e il successivo, ma se lo si inserisce dentro il ciclo si rischia di farlo inutilmente troppo spesso. Che succede poi se il programma che fa i calcoli usa una funzione di libreria già scritta, magari da qualcun altro? Questa sicuramente non conterrà già i controlli di STS, e dovremmo crearne una versione modificata. A questi problemi aggiungiamo anche il fatto che il programma e le librerie diventano presto illegibili se dobbiamo inserire i controlli di STS in mezzo a funzioni che fanno tutt'altro.

1 Interruzioni (singola sorgente)

Per risolvere questi problemi modifichiamo leggermente l'hardware. L'idea di base (da perfezionare) è di portare il bit READY di STS direttamente in ingresso alla CPU (Figura 1), quindi di modificare il microprogramma della CPU in modo che, dopo ogni fase di esecuzione, controlli lo stato del nuovo ingresso e, se lo trova attivo, carichi un nuovo valore in **rip**. L'effetto è di avere inserito automaticamente un salto nel programma nel momento in cui la stampante diventa pronta. Visto che il controllo di READY viene eseguito dopo ogni

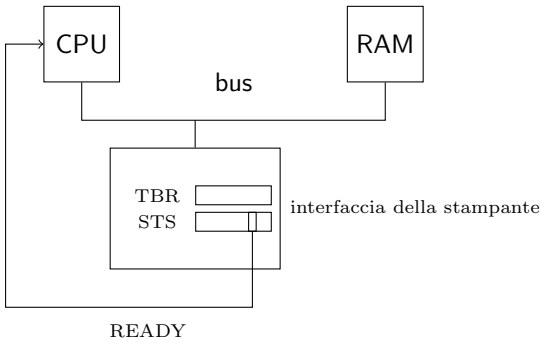

Figura 1: Idea di base del meccanismo delle interruzioni.

istruzione, non rischiamo di farlo troppo raramente (almeno rispetto ai tempi di una stampante); visto che abbiamo realizzato il controllo con un test in hardware di un piedino di ingresso, non paghiamo il costo di dover esegure le istruzioni che leggono STS e controllano READY; visto che il controllo è fatto automaticamente dalla CPU, il programma principale non deve più contenere queste istruzioni "estranee" che ne compromettono la leggibilità. Il sistema è ora in grado di riconoscere autonomamente il verificarsi dell'evento e che corrisponde a "la stampante è pronta a ricevere il prossimo carattere". Il programmatore deve solo scrivere una routine r che invia il prossimo carattere da stampare e associarla ad e. Il modo in cui avviene questa associazione, nel caso in cui ci sia un unico evento possibile, può essere molto semplice: il processore può caricare un indirizzo costante in **`rip`** quando riconosce l'evento, e il programmatore può semplicemente caricare la sua routine r a quell'indirizzo. Il programmatore poi scrive il programma principale p che ciclicamente calcola un nuovo valore, prepara la riga da stampare nel buffer e passa a calcolare il valore successivo. Periodicamente, il programma p verrà *interrotto* per eseguire la routine r che stampa il prossimo carattere preso dal buffer.

Un modo per concettualizzare ciò che avviene, dal punto di vista della stampante, è che questa ha bisogno di "attenzione" da parte del software quando ha finito di stampare un carattere, perché il software le deve dire cosa stampare dopo. Inoltra dunque una richiesta di attenzione, o di *servizio*, alla CPU, settando il bit READY. La CPU reagisce a questa richiesta interrompendo il programma principale (per questo la richiesta della stampante viene anche detta *richiesta di interruzione*) e cedendo forzatamente il controllo ad una *routine di servizio* per la stampante. Quando questa routine scrive il prossimo carattere in TBR, la stampante (la sua interfaccia) pone READY a zero, e questo "rimuove" la richiesta di servizio dalla CPU, a significare che la stampante ha ricevuto il servizio che aveva richiesto. La scrittura in TBR, dunque, funge da *risposta* alla richiesta di interruzione.

Questa è l'idea di base, che però richiede di essere perfezionata perché possa

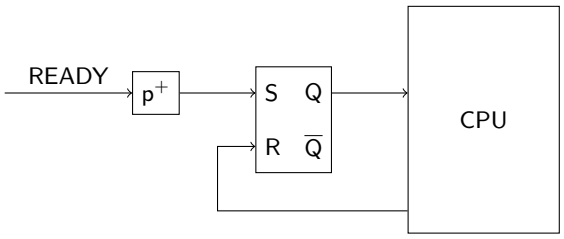

Figura 2: Rete per il riconoscimento di una nuova richiesta di interruzione.

funzionare effettivamente.

- A livello hardware, dobbiamo intanto assicurarci che la CPU non veda due volte la stessa richiesta. Per come abbiamo descritto il meccanismo, infatti, quello che accade è che, non appena la CPU vede READY 1, salta alla prima istruzione della routine di servizio. Se questa istruzione non è esattamente quella che scrive in TBR, alla fine della sua esecuzione READY sarà ancora a 1 e, dunque, la CPU salterà nuovamente alla prima istruzione della routine di servizio, ciò quella appena eseguita, e così all'infinito. Questo problema può essere risolto in hardware con una rete come in Figura 2, in cui la richiesta viene trasformata in un inpulso che setta un latch S/R. Il microprogramma della CPU provvederà a resettare il latch dopo aver visto la richiesta, in modo che il latch sia di nuovo attivo solo se arriva una *nuova* richiesta di interruzione.

- Se vogliamo che la routine di servizio possa tornare al programma principale nel punto in cui era stato interrotto, la CPU non può limitarsi a sovrascrivere **`rip`** quando accetta la richiesta, ma deve prima scrivere il precedente valore da qualche parte. Una soluzione è di scriverlo sullo stack, in modo che la routine di servizio possa terminare con una **`ret`** (o, come vedremo, con una istruzione simile); questa è la soluzione adottata nei procesori Intel.

- Anche a livello software vanno risolti molti problemi, per poter scrivere correttamente il programma principale e la routine di servizio. In realtà, i problemi sono così tanti che Dijkstra, non a torto, ha scritto che questo semplice meccanismo aprì un "vaso di Pandora". Per il momento vi accenniamo soltanto—avremo modo di ritornarci.

Risolti questi problemi, possiamo pensare di raffinare ed estendere il meccanismo in vari modi. Il problema principale, dal punto di vista del software, è l'imprevedibilità del momento in cui può essere accettata una richiesta, con conseguente salto alla routine di servizio. Dal punto di vista del programmatore è utile poter temporaneamente "disabilitare" le richieste di interruzione, fare cioé in modo che la CPU le ignori durante l'esecuzione di certe porzioni di codice. La soluzione tipicamente adottata, anche nei processori Intel, è che la CPU abbia

un flag di *interrupt enabled* che possa essere settato e resettato via software, e che la CPU ascolti le richieste di interruzione solo se questo flag è settato. Nel processore intel questo è il flag IF (Interrupt Flag, bit 9) del registro dei flag e le istruzioni `cli` e `sti` possono essere usate per resettarlo e settarlo, rispettivamente. Le richieste che dovessero arrivare mentre IF è zero verrano servite non appena IF tornerà a 1.

Una porzione di codice che non vogliamo venga interrotta è, tipicamente, la routine di servizio stessa, cosa che invece può accadere non appena la routine esegue l'istruzione di risposta alla richiesta di servizio (la scrittura in TBR, nel nostro esempio). La situazione è talmente tipica che il processore Intel può (su richiesta del programmatore) resettare automaticamente il flag IF ogni volta che accetta una richiesta. Prima di farlo, il processore salva in pila il vecchio valore del registro dei flag, oltre al vecchio valore di **`rip`**. Il processore fornisce poi l'istruzione **`iretq`** che preleva dalla pila sia **`rip`** che i flag. Le routine di servizio devono terminare con **`iretq`**, invece che con **`ret`**, in modo da estrarre dalla pila entrambi i valori (e altri che vedremo) e riportare IF a 1.

2 Interruzioni (più sorgenti)

Un altro modo di estendere il meccanismo è quello di prevedere più sorgenti di interruzione, in modo che il sistema possa riconoscere un certo numero di eventi distinti e il programmatore possa associare una routine di servizio diversa a ciascuno di questi eventi, come avevamo detto all'inizio. Delle periferiche che abbiamo studiato, tre sono in grado di generare richieste di interruzione:

- la tastiera, quando RBR è pieno o TBR si svuota dopo una scrittura;
- il timer, quando il contatore 0, opportunamente programmato, termina il conteggio;
- l'hard disk, quando il buffer interno è pieno dopo una operazione di lettura o vuoto dopo una operazione di scrittura.

La tastiera e l'hard disk generano richieste di interruzione solo se queste sono state abilitate *nell'interfaccia*. Questa è un'altra soluzione molto tipica, che permette al programmatore di non dover gestire richieste di interruzione da parte di interfacce che non sta usando. Le interfacce prevedono dunque uno o più registri di *controllo*, con varie funzioni tra cui abilitare o disabilitare l'interfaccia a generare richieste di interruzione (consultare gli esempi di I/O per i dettagli).

Per supportare più sorgenti di richieste di interruzione dobbiamo rispondere a tre domande:

- come inoltrare le richieste alla CPU;
- se e come associare una routine di servizio diversa ad ogni sorgente;
- cosa fare in caso di richieste che arrivano mentre ancora è in corso il servizio di una richiesta precedente da parte di un'altra sorgente.

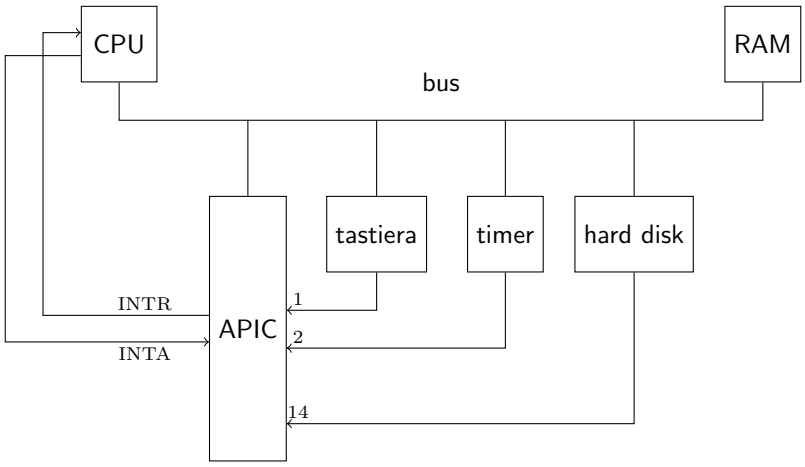

Figura 3: Controllore delle interruzioni.

2.1 Controllore delle interruzioni

Per rispondere alla prima domanda, teniamo presente che il meccanismo che abbiamo introdotto non ha cambiato la natura fondamentalmente sequenziale della CPU: la CPU, dal suo punto di vista, continua ad avere un unico flusso di controllo e ad ogni istante esegue un unico programma: il programma principale, oppure una routine di servizio[2] Anche se abbiamo più sorgenti di richieste di interruzione e più routine di servizio, la nostra CPU potrà comunque eseguirne solo una alla volta. Ha allora senso che la CPU abbia un unico piedino su cui riceve richieste di interruzione, e un qualche componente esterno si preoccupi di raccogliere tutte le richieste provenienti dalle varie sorgenti, ordinarle in qualche modo e inoltrarle una alla volta alla CPU[3] Questo componente è detto *controllore delle interruzioni*. Lo schema del sistema è ora quello di Figura 3. In particolare, facciamo riferimento al controllore APIC (Advanced Programmable Interrupt Controller) che si trova nei sistemi basati su chipset Intel e anche, emulato, nella nostra macchina QEMU. L'APIC possiede 24 piedini di ingresso, numerati da 0 a 23, a cui possono essere collegate le linee di richiesta di interruzione di altrettante interfacce. In figura vediamo come sono collegate le periferiche della macchina QEMU. Il controllore permette la gestione *vettorizzata* delle richieste di interruzione. Con questo termine ci si riferisce al fatto che

[2]La CPU, in realtà, non è neanche consapevole della distinzione: per lei, la gestione della richiesta di interruzione termina subito dopo aver effettuato il salto, dopo di che riprende il normale ciclo di prelievo, decodifica, esecuzione, a partire dal nuovo **`rip`**. In particolare, la CPU non è "in attesa" che venga eseguita una **`iretq`** dopo l'accettazione di una richiesta di interruzione, come non è in attesa che venga eseguita una **`ret`** dopo una **`call`**.

[3]Anche se la CPU ha più di un piedino di ingresso per le richieste di interruzione, serve comunque una circuiteria, interna alla CPU stessa ma concettualmente distinta, che le raccolga e ne lasci passare una sola per volta. Vedremo che, in effetti, la CPU Intel, come molte altre, ha anche un altro piedino da cui riceve richieste di interruzione "non mascherabili": queste hanno precedenza su tutte le altre.

ad ogni sorgente di interruzione è associato un *tipo* numerico, che viene passato alla CPU insieme alla richiesta. La CPU utilizza questo tipo per consultare una tabella che associa i tipi agli indirizzi delle routine di servizio. Nei processori Intel la tabella si chiama Interrupt Descriptor Table (IDT) e il programmatore la può allocare ovunque in memoria, quindi caricarne l'indirizzo nel registro IDTR del processore. In questo modo è possibile associare una routine di servizio diversa ad ogni sorgente: basta scrivere l'indirizzo della routine nell'entrata della IDT corrispondente al tipo da associare. In assenza della vettorizzazione, ad ogni richiesta andrebbe in esecuzione sempre la stessa routine, che a quel punto dovrebbe capire via software quale interfaccia ha richiesto il servizio (per esempio, consultando i registri di stato di ogni interfaccia).

Nell'APIC l'associazione tra sorgente e tipo è configurabile dal programmatore. Il controllore possiede infatti un diverso registro interno, accessibile via software, per ognuno dei piedini di ingresso. Per ogni piedino è possibile specificare il tipo, su 16 bit, e altre informazioni legate a come l'APIC dovrà riconoscere le richieste in ingresso. In particolare, per ogni piedino è possibile decidere:

- se il riconoscimento di una nuova richiesta deve avvenire sul fronte (segnale in ingresso che cambia livello) o sul livello (vedere più avanti);
- se il segnale di ingresso deve essere considerato attivo quando è alto oppure quando è basso;
- se le richieste devono essere ignorate o meno (mascherate).

Per il dialogo con il processore supponiamo che ci siano due collegamenti: uno, INTR (INTerrupt Request), dall'APIC alla CPU, tramite il quale l'APIC inoltra una richiesta di interruzione; un altro, INTA (INTerrupt Acknowledge), dalla CPU all'APIC, usato per un protocollo di handshake:

1. quando l'APIC vuole inviare una richiesta di interruzione, attiva INTR e aspetta che sia attivo INTA;
2. la CPU controlla lo stato di INTR dopo l'esecuzione di ogni istruzione, se IF è 1; quando lo trova attivo, attiva a sua volta INTA;
3. quando l'APIC vede INTA attivo, passa il tipo dell'interruzione al processore sul bus e disattiva INTR[4]
4. quando il processore vede INTR disattivo, legge il tipo e disattiva INTA, chiudendo l'handshake.

[4]Restiamo sul vago su come il tipo viene passato, in quanto il vero APIC è in realtà scomposto in due parti: un I/O APIC sulla scheda madre e un Local APIC all'interno della CPU. Il dialogo, compreso lo scambio del tipo, avviene tra i due APIC su un bus dedicato. Questa organizzazione è pensata per i sistemi con più CPU, in modo da avere, per esempio, un unico I/O APIC che dialoga con tanti Local APIC, uno per CPU. Nel nostro caso possiamo semplificare la discussione, perché ci limitiamo al caso con una sola CPU.

L'APIC possiede altri tre registri interni che ci servono per capire in dettaglio il suo funzionamento: IRR (Interrupt Request Register), ISR (In Service Register) e EOI (End Of Interrupt). I registri IRR e ISR sono entrambi da 256 bit, un bit per ogni possibile tipo. Supponiamo per il momento che una sola interfaccia sia abilitata a inviare richieste di interruzioni, e chiamiamo i il piedino di ingresso dell'APIC a cui tale interfaccia è collegata. Supponiamo anche che il piedino i sia impostato per il riconoscimento sul fronte, e il tipo associato sia t. L'APIC svolge continuamente le seguenti azioni:

1. ad ogni fronte su i, se il bit t di IRR non è già attivo, lo attiva
2. se il bit t di IRR è attivo e il bit t di ISR non lo è, l'APIC inizia l'handshake con la CPU;
3. quando l'handshake arriva al punto 3 (la CPU ha accettato la richiesta), invia il tipo t e "sposta" il bit attivo da IRR in ISR (disattiva t in IRR e lo attiva in ISR);

Lo scopo di ISR è di tenere traccia di quali richieste sono attualmente in servizio sulla CPU. L'APIC assume che la CPU, nel momento in cui accetta la sua richiesta, passi ad eseguire la routine di servizio corrispondente, e per questo setta il bit t di ISR a 1. L'APIC non inoltrerà altre richieste dello stesso tipo fino a quando si troverà in questo stato (si veda il punto 2 qui sopra: l'handshake parte solo se il bit t di ISR è a zero). Un eventuale altro fronte sul piedino i verrà comunque registrato in IRR (punto 1). In pratica è come se i due bit t in IRR e ISR realizzassero una coda di due richieste: una in servizio e l'altra in attesa. Una terza richiesta che dovesse arrivare in questo stato, però, verrebbe persa. È compito del software far sapere all'APIC che la routine di servizio è terminata, scrivendo un valore qualsiasi nel registro EOI. In riposta a questa scrittura l'APIC azzera il bit t in ISR e, se il bit t di IRR è 1 (c'è un'altra richiesta pendente arrivata nel frattempo) si riporta al punto 2, altrimenti al punto 1.

Nel caso di riconoscimento sul livello l'APIC si comporta diversamente solo per quanto riguarda il riconoscimento di una nuova richiesta sul piedino: se il bit t di IRR è a zero e c'è un fronte su i, l'APIC registra una nuova richiesta, come nel punto 1 qui sopra; da questo punto in poi, però, l'APIC smette di osservare il piedino i; lo osserverà di nuovo solo all'arrivo del prossimo EOI: se in quel momento lo ritrova ancora (o di nuovo) attivo, registra una nuova richiesta. L'idea è che normalmente la periferica dovrebbe aver disattivato il piedino i, una volta che la routine di servizio ha effettuato la risposta alla richiesta (si pensi all'esempio della stampante: la richiesta resta attiva fino a quando la routine non scrive in TBR) e il software dovrebbe inviare l'EOI *dopo* che la routine ha effettuato questa azione. Dunque, se la richiesta è ancora attiva, deve trattarsi di una nuova richiesta.

Negli esempi vedremo quando conviene scegliere un riconoscimento sul fronte o uno sul livello.

2.2 Gestione annidata delle richieste di interruzione

Passiamo ora a considerare il caso generale, in cui le richieste possono arrivare da qualsiasi piedino, anche contemporaneamente. In generale si preferisce realizzare la *gestione annidata* delle richieste di interruzione, in cui una routine di servizio può essere a sua volta interrotta da un'altra routine di servizio. La cosa ha senso se la seconda routine ha una *priorità* maggiore rispetto alla prima (per esempio, le richieste che arrivano dal timer hanno in genere massima priorità, in quanto non è possibile ritardarle). Ovviamente l'annidamento può avvenire a qualsiasi livello, con la seconda routine che può essere interrotta da una terza, a maggiore priorità, e così via. Dal momento che tutti gli indirizzi a cui ritornare sono salvati sulla stessa pila, le routine di servizio devono poi terminare in ordine LIFO: l'ultima ritorna alla penultima e così via.

Affinchè l'annidamento sia possibile, la CPU non deve disabilitare le interruzioni mentre è in esecuzione una routine di servizio. Nel processore Intel questo può essere ottenuto facilmente, in quanto la disabilitazione delle interruzioni è facoltativa (basta settare un flag nelle entrate della IDT corrispondenti alle routine che si vuole eseguire in questo modo).

L'APIC può essere usato per aiutare nella gestione annidata delle interruzioni con priorità. L'APIC, infatti, interpreta il tipo associato ad un piedino come la priorità delle richieste provenienti da quel piedino. Più precisamente, i 4 bit più significativi del tipo rappresentano, per l'APIC, la sua *classe di priorità*, in ordine numericamente crescente (15 è dunque la classe di priorità massima). Quando una nuova richiesta viene registrata in IRR, l'APIC confronta la classe di priorità della richiesta più a sinistra in IRR (la richiesta pendente a priorità maggiore), sia p_r, con la classe di priorità della richiesta più a sinistra in ISR (la richiesta in servizio a priorità maggiore), sia p_s, e inizia l'handshake con la CPU se, e solo se, $p_r > p_s$. Quando la CPU accetta questa richiesta l'APIC si comporta come al solito, spostando il bit da IRR in ISR. Si noti che ora più bit di ISR possono essere a 1, riflettendo il fatto che più routine di servizio hanno iniziato la propria esecuzione e non sono ancora concluse. Si noti, inoltre, che il fatto che l'APIC non inoltra le richieste con $p_r \leq p_s$ comprende il caso particolare, che abbiamo discusso in precedenza, in cui arrivi una nuova richiesta dallo stesso piedino di una richiestra già accettata, perché in quel caso le classi di priorità sono uguali.

Quando il software scrive in EOI, l'APIC resetta il bit più a sinistra in ISR. In pratica, l'APIC assume che il software stia rispettando l'ordine LIFO di esecuzione delle routine di servizio, e dunque che la scrittura sia stata effettuata perché è terminata la routine a più alta priorità tra quelle attualmente attive. Dopo aver resettato il bit, l'APIC ricontrolla le classi di priorità in IRR e in ISR per vedere se è necessario inoltrare una nuova richiesta

Si noti che l'APIC, per decidere se inoltrare o no una richiesta, guarda esclusivamente le classi di priorità, ignorando i 4 bit meno significativi dei tipi. Al momento di inoltrare una nuvoa richiesta alla CPU, però, usa l'intero tipo per scegliere quale richiesta inviare tra quelle registrate in IRR.