Comunicazione con i dispositivi di ingresso ed uscita

In questo Capitolo si descriveranno le modalità di interazione tra il processore e i dispositivi di ingresso e uscita. In questa esposizione per motivi pedagogici inizieremo prima ad ipotizzare, come avveniva nei primi sistemi di elaborazione e poi nei sistemi a microprocessore, che le attività di interazione vengono gestite direttamente dal processore. Ciò permetterà al lettore di capire le basi con cui poi si sono evolute le architetture dei sistemi di elaborazione, in cui le interazioni con il mondo esterno sono state in parte o completamente delegate a processori progettati ad-hoc mano a mano sempre più sofisticati.

Quindi di seguito si daranno solo le indicazioni di massima del funzionamento dei dispositivi di ingresso ed uscita, dedicando successivamente un capitolo ad hoc per approfondire come funzionano alcuni dispositivi di ingresso/uscita e l'evoluzione delle interconnessioni tra il processore e tali dispositivi, mentre ora si approfondiranno le loro modalità di interazione con il processore. Ciò permetterà al lettore di prendere dimestichezza con la nomenclatura sottostante e di capire i motivi fondamentali che hanno portato alle differenti soluzioni tenendo in conto dei costi, delle prestazioni e del livello di integrazione circuitale sui singoli chip.

Le modalità di interazione del processore con le periferiche dipendono da molti fattori quali: la velocità di funzionamento di ogni singolo componente, l'urgenza o meno con cui è necessario acquisire/trasmettere informazioni e il livello di efficienza con cui si desidera funzioni il processore.

Da quando sono nati i processori hanno avuto sempre una velocità di lavoro (vogliamo dire di processamento) di alcuni ordini superiore a quello dei dispositivi di ingresso ed uscita.

Tipo	Codice	Velocità di scambio
Dischi Magnetici	Byte	Fino a 300 Mcar/sec
Nastri Magnetici	Byte	Fino a 30 Mcar/sec
Stampante Seriale	Byte	200 – 1200 car/sec
Stampante Parallela	Byte	1K – 100K car/sec
Terminali CRT	Byte	300 – 19,2K car/sec
Convertitori analogico-digitali	Parola 8-16 bit	10-10M parole/sec
USB 1.0	Byte	1,5Mcar/sec
USB 2.0	Byte	60Mcar/sec

Tabella 1: velocità tipiche dei dispositivi di Ingresso/Uscita

In Tabella 1 si può notare che le velocità di trasferimento dei dati delle periferiche sono dai 3 agli 8 ordine di grandezze minori rispetto ai tempi di elaborazione dei processori e questo crea dei problemi nelle modalità di trasferimento dei dati.
Il trasferimento dei dati da e verso le periferiche dovrebbe essere il più veloce possibile, questo è stato formalmente dimostrato da Amdahl con la sua legge che mette in relazione lo speedup (S) con la velocità dei processori e quelle dei dispositivi di input e output. In particolare Amdahl ha notato che mentre le velocità dei processori migliorano notevolmente nel tempo, mentre quelle dei dispositivi di ingresso e uscita molto lentamente, pertanto dato un programma il cui tempo di esecuzione può essere suddiviso in due parti, di cui la prima f è la frazione di tempo di esecuzione sul processore e il rimanente in interazione con le periferiche, nell'ipotesi di utilizzare un processore K volte più veloce del precedente, lo speedup è dato da:

$$S = \frac{1}{(1-f)+\frac{f}{K}}$$

Pertanto il collo di bottiglia prestazionale risulta l'interazione con i dispositivi di ingreso ed uscita, infatti anche nell'ipotesi di avere un fattore K che tende

all'infinito, il valore dello speedup è limitato dalla frazione di tempo di interazione con il mondo esterno.

p.e. 10% x I/O & 90% x CPU (f=0,9)
se K=10 => S prossimo a 5
se K=1000 => S prossimo a 10

Le interazioni tra processore e dispositivi di ingresso/uscita possono avvenire secondo modalità che hanno delle forti similitudini con le interazioni tra gli esseri umani e tra di essi con l'ambiente esterno.

Prendendo a riferimento i seguenti esempi quotidiani di interazione tra due esseri umani (A, B e C):

- appuntamento tra un ragazzo (A) e un altro (B);
- ricevimento programmato di un professore (A) per gli studenti (B e C);
- colloquio tramite telefonata, B chiama A;
- ricevimento di una lettera, A riceve da B;

si può notare che l'attività di interazione può nascere dalle necessità di uno o più attori. Nel primo esempio l'appuntamento è fissato da entrambi gli attori, mentre nel secondo esempio l'attore dell'interazione è il professore, che prevede una finestra temporale in cui ricevere gli studenti nel suo studio, nel terzo e nel quarto caso, invece, l'attore B ha intenzione di interagire con l'A indipendentemente da quanto stia facendo quest'ultimo.

Per un essere umano quindi possiamo dire che sostanzialmente un'interazione può essere classificata come:

- programmata
- su richiesta esterna

In particolar modo le modalità di interazione di tipo programmato possono essere di due tipi:

- modalità busy waiting, quando un attore (p.e. A) è in attesa che un altro attore sia disponibile e nel frattempo l'attore A non fa nulla (come nel caso in cui uno dei due amici arriva puntuale ad un appuntamento e l'altro arriva in ritardo);

- modalità polling, quando l'attore A è in attesa che uno dei possibili interagenti sia disponibile all'interazione (come nel caso in cui il professore si rende disponibile per il ricevimento studenti e può arrivare nessuno, uno o più studenti della sua classe e rimane nella sua stanza).

Mentre quella su richiesta esterna, può essere classificata come:

- interruzione, quando l'attore B richiama l'attenzione dell'attore A indipendentemente dalla sua volontà (p.e. tramite una chiamata al telefono, il recapito di una email)

Così anche nei sistemi di elaborazione le interazioni tra un processore e di un dispositivo di I/O possono essere di tipo:

- *I/O programmato*
- *su richiesta esterna*

In generale i dispositivi di ingresso/uscita sono delle "entità" (sistemi digitali complessi) costituiti da almeno un Sottosistema di Calcolo (SCA) e da un Sottosistema di Controllo (SCO).
Quindi il trasferimento dati tra la CPU ed un dispositivo esterno deve essere effettuato in modo coordinato tra i loro SCO. Ovvero il SCO del dispositivo esterno deve essere in grado di reagire quando il SCO della CPU vuole effettuare un trasferimento e viceversa; ed inoltre il trasferimento può essere effettuato solo quando entrambi i SCO lo consentono. Questo coordinamento deve avvenire seguendo un insieme di regole prestabilite (***protocollo***) che definisce le possibili sequenze di interazione. Se il trasferimento è coordinato unicamente dal SCO del processore il trasferimento si dice ***sincrono*** (sincrono con la velocità di funzionamento del SCO del processore), altrimenti ***asincrono***.

Nel primo caso il dispositivo indirizzato (cioè quello con cui la CPU necessita di interagire) deve leggere e/o scrivere il dato nell'intervallo di tempo previsto dal SCO del processore per la lettura e/o la scrittura e quindi sincrono con la velocità di funzionamento del processore, come nel caso di interazione processore-memoria di lavoro (RAM statica). Invece, se la velocità di lettura/scrittura del dispositivo è inferiore a quella richiesta è necessario "rallentare" le attività del SCO di quest'ultimo, tale modalità si denota come "***busy_waiting***", e in tal caso il trasferimento è asincrono.

Ci sono due modalità per realizzare la tecnica di busy_waiting: la prima hardware o firmware e la seconda software.

La soluzione hardware o firmware è la istanziazione dell'interazione di due sistemi digitali complessi, dove il primo sistema digitale è il processore e il secondo è la periferica, quindi si prevede l'uso di un segnale di condizione $\overline{\text{WAIT}}$ che viene usato dal SCO del processore per individuare se la periferica ha completato o meno il trasferimento richiesto.
E' da notare che nel seguito tutti i segnali di condizione che dal mondo esterno vanno verso il processore sono variabili booleane attive basse (cioè lavorano in logica negativa, quando sono bassi equivalgono al vero booleano). Ciò viene fatto per semplificare la connessione tra le periferiche e il processore utilizzando la connessione wired or che lavora in logica negativa

Mentre la soluzione software prevede, come vedremo, l'utilizzazione di istruzioni software ad hoc necessarie per implementare il protocollo di comunicazione.

Notare come la modalità di interazione busy_waiting implementata ad hardware o firmware può essere utilizzata anche per effettuare il trasferimento dati da memorie con velocità di funzionamento inferiore a quella dei processori che vi accedono. Questo tipo di trasferimento si dice *semi-sincrono* in quanto il trasferimento dati è effettuato come variante del *sincrono*, salvaguardandone il principio di funzionamento. Mentre la modalità implementata a software che utilizza quindi un protocollo di comunicazione viene detta *asincrona*.

Oltre all'interazione busy_waiting implementata a software anche le altre due modalità di cui si è parlato (*polling* e *interruzione*) sono interazioni di tipo *asincrono*.

ISTRUZIONI DI INPUT/OUTPUT

Si ricorda che il set di istruzioni che si prevedono per l'interazione con le periferiche sono inX, outX, insX e outX. Si vuole inoltre ricordare che per la gestione delle interruzioni ci sono anche altre due istruzioni che, come vedremo, servono per modificare il flip/flop per la gestione delle interruzioni, flip/flop interno al processore e quindi non visibile dalle periferiche (le istruzioni : ***seti*** e ***clri***, altro non sono che una istanziazione della manipolazione di un flip/flop del registro di stato, SR).

Alcune delle istruzioni precedenti sono tipiche dell'ISA dell'X86, quali ***inX***, ***outX***, ***insX*** e ***outsX***, mentre le altre sono state introdotte per consentire al lettore di prendere confidenza con le funzionalità che nelle attuali architetture di sistemi di

elaborazione vengono normalmente delegate ai processori di input ed output, come faremo vedere successivamente.

Nel proseguo di questo capitolo faremo vedere prima come implementare le istruzioni ***inX*** e ***outX*** in modalità busy waiting implementata a firmware per poi far vedere come utilizzare le altre istruzioni per poter eseguire i protocolli prima accennati.

Per comodità del lettore si riportano la sintassi e la semantica delle prime due istruzioni che andremo a discutere : la outX e la inX.

outX %?a?, %dx

sposta nel registro di interfaccia il cui indirizzo è memorizzato nel registro %dx quanto è memorizzato nel registro accumulatore (si preleveranno 8, 16, 32 o 64 bit, secondo il valore di X. Specificato nell'istruzione stessa, e quindi i dati potranno essere prelevati da %al, %ax, %eax oppure da %rax).
Da notare che per identificare un registro esterno si utilizzano 16 bit, questo permette uno spazio di indirizzamento di 2^{16} (64K locazioni differenti) che è comunque un valore molto elevato considerando il numero delle periferiche presenti in un sistema di elaborazione.

inX %dx, %?a?

sposta nell'accumulatore quanto memorizzato nel registro interfaccia della periferica il cui indirizzo è memorizzato nel registro %dx (si preleveranno 8, 16, 32 o 64 bit, secondo il valore di X, specificato nell'istruzione stessa, e quindi i dati potranno essere memorizzati in %al, %ax, %eax oppure in %rax).
Da notare, anche in questo caso, che per identificare un registro esterno si utilizzano 16 bit, questo permette uno spazio di indirizzamento di 2^{16} (64K locazioni differenti) che è comunque un valore molto elevato considerando il numero delle periferiche presenti in un sistema di elaborazione

Interazione busy waiting implementata a firmware

Come detto precedentemente il processore è un sistema digitale complesso costituito da un SCA o data path e da un SCO, che interpreta le istruzioni codificate in linguaggio macchina. Inoltre anche i dispositivi di ingresso ed uscita sono dei sistemi digitali complessi, costituiti quindi da un SCA e da un SCO. Se entrambi i

sistemi digitali complessi partecipanti alla comunicazione o scambio dati (processore e dispositivo di I/O) avessero lo stesso clock allora l'interazione potrebbe avvenire in modo *sincrono*, in quanto trasferire un dato tra due componenti appartenenti allo stesso SCA o a due SCA distinte avverrebbe alla stessa velocità. Invece, come avviene normalmente, la velocità di funzionamento della periferica è inferiore a quella del processore, pertanto una modalità di interazione quella in cui i due SCO eseguono un protocollo di sincronizzazione (protocollo di handshaking) e che di seguito viene riportato brevemente per fini didattici. Come già detto, questo tipo di trasferimento si dice *semi-sincrono* in quanto il trasferimento dati è effettuato come variante del *sincrono*, salvaguardandone il principio di funzionamento.

Ovviamente nel caso in cui il produttore o il consumatore fosse molto più veloce del partner il primo è costretto ad aspettare il completamento delle attività dell'altro prima di poter riprendere a funzionare alla propria velocità.

Lo schema di interazione precedente tra due sistemi digitali complessi adottanti il protocollo di handshaking, nell'ipotesi che essi siano rispettivamente un processore ed un dispositivo di I/O, può essere ridisegnato come nelle figure successive nelle ipotesi, rispettivamente, che il processore sia il consumatore o produttore del dato.

In particolare in figura 1 è rappresentato lo schema delle interfacce per consentire lo scambio dei dati da una periferica verso il processore, mentre in figura 3 dal processore verso la periferica. Inoltre nelle figure 2 e 4 sono riportate le temporizzazioni dei segnali per consentire lo scambio delle informazioni.

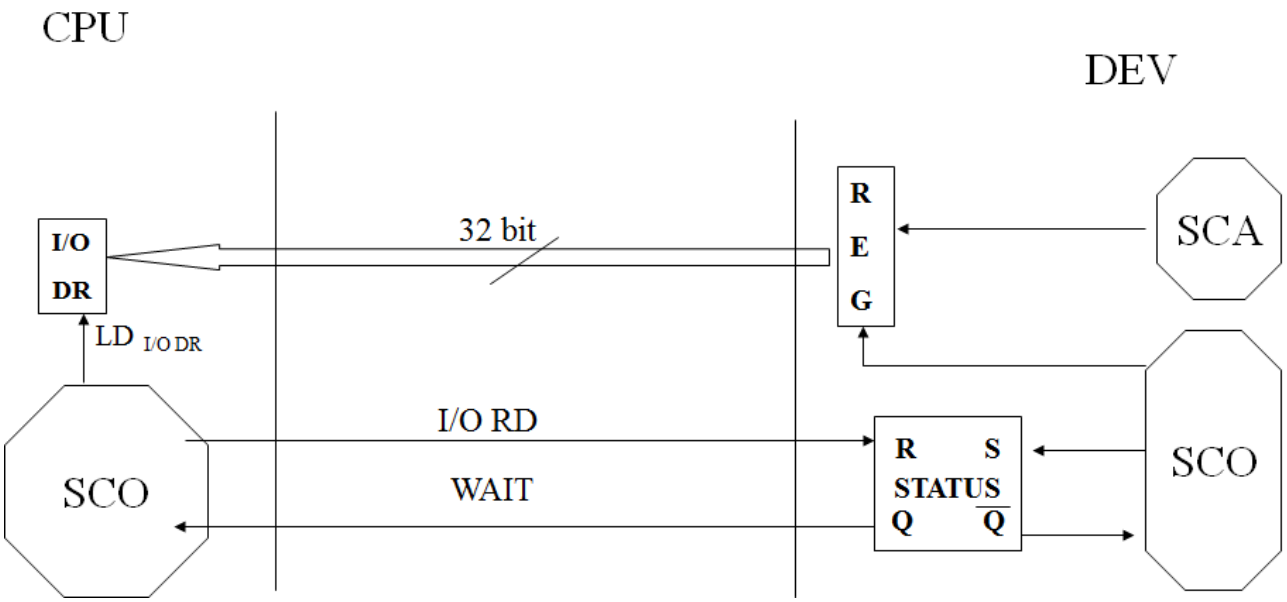


Figura 1: interfaccia di periferica di input

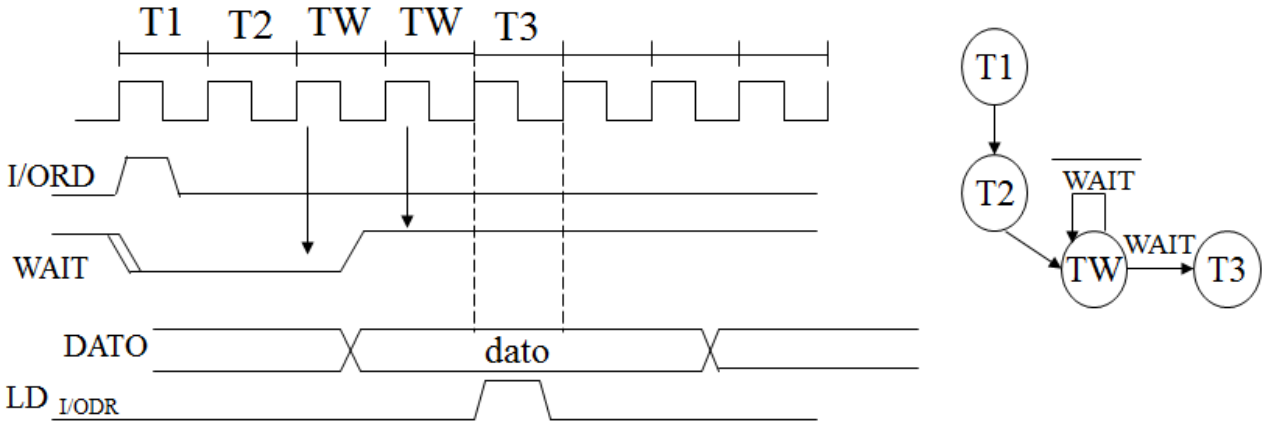


Figura 2: temporizzazioni

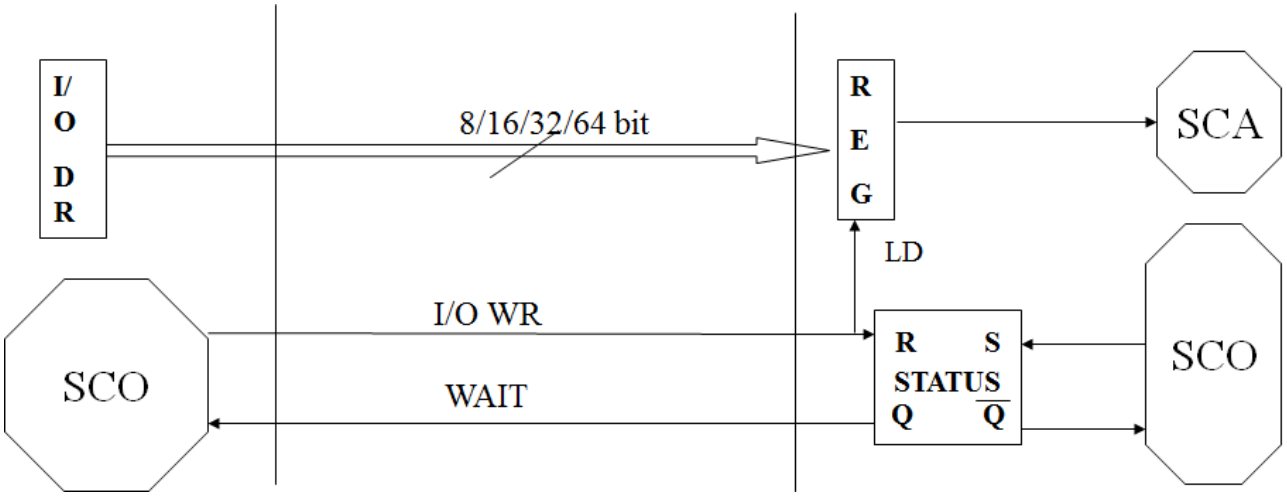


Figura 3: interfaccia di periferica di output

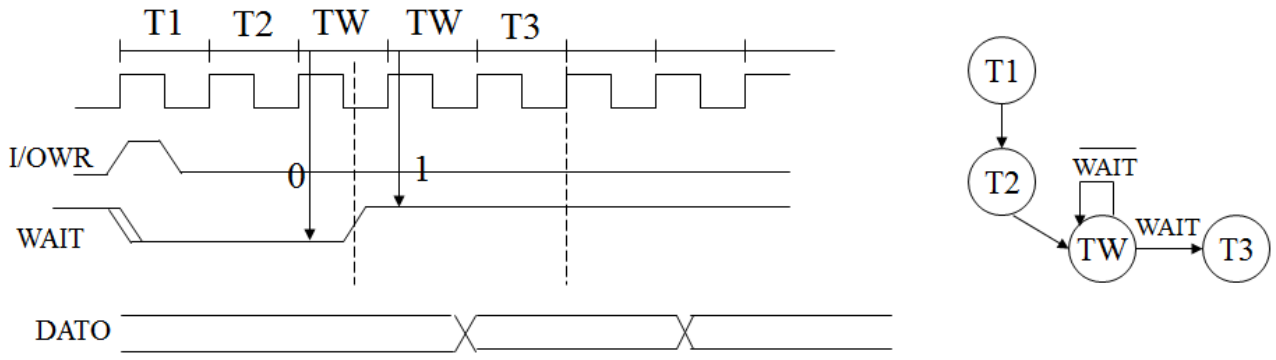


Figura 4: temporizzazioni

Dalle figura 1 si può vedere che il processore per acquisire un dato dalla periferica necessita di un registro di interfaccia, così come la periferica per spedirlo. Inoltre si utilizza un flip/flop di handshaking (STATUS) che consente al processore di richiedere il dato, il processore rimarrà in attesa fino a che il dato non sarà prodotto e stabile sul registro di interfaccia della periferica, tale abilitazione è comandata dal segnale di WAIT.

Stesso tipo di interazione si presenta nel caso in cui è il processore che produce il dato verso la periferica, figura 3. Anche in questo caso si fa uso di registri di interfaccia ed inoltre si utilizza il segnale di WAIT per completare l'handhaking. In particolare una volta che il processore scrive il dato nel registro di interfaccia della periferica resetta anche il flip/flop STATUS che rimane in tale stato fino a che il SCO della periferica non lo setti e questo potrà avvenire solo quando la periferica avrà consumato il dato.

I due schemi architetturali di figura 1 e figura 3 fanno riferimento ad un caso molto particolare, quello in cui il processore ha un'unica periferica, questo in generale non è vero, dato che gli elaboratori hanno molti dispositivi sia di ingresso che di uscita. Dai due disegni si può vedere che in entrambi i casi le periferiche di I/O presentano un registro di interfaccia, che nel disegno è stato dimensionato a 64 bit, ma che può essere di un qualunque numero. Ora ricordando le modalità di interazione tra il processore e la memoria ci si può ricordare che la memoria può essere vista come un insieme di registri indirizzabili individualmente dal processore, non a caso ogni qual volta il processore legge o scrive un dato in memoria deve indicarne il relativo indirizzo, e quindi la memoria è connessa al processore tramite un bus costituito da: un bus dati, un bus indirizzi e da un bus di controllo. Per i dispositivi di I/O si utilizza la stessa modalità di interazione, come schematizzato nelle figure successive, in cui ogni registro di interfaccia delle periferiche è identificato con un indirizzo. E' da ricordare che ogni registro, qualunque sia la sua dimensione (cioè numeri di bit che può memorizzare) è un insieme di flip/flop, come la memoria statica. Inoltre, come vedremo, per identificare un registro o un singolo flip/flop (caso particolare di registro ad un bit), si potranno utilizzare indirizzi utilizzati per identificare altri registri o flip/flop, purché nell'identificazione si utilizzino segnali di controllo di tipo diverso.

A questo punto nasce spontanea la domanda se per interconnettere i dispositivi di ingresso/uscita si possa utilizzare lo stesso bus già impiegato per la memoria oppure utilizzarne uno ad hoc. Entrambe le soluzioni sono fattibili, di seguito ne discuteremo i vantaggi e gli svantaggi.

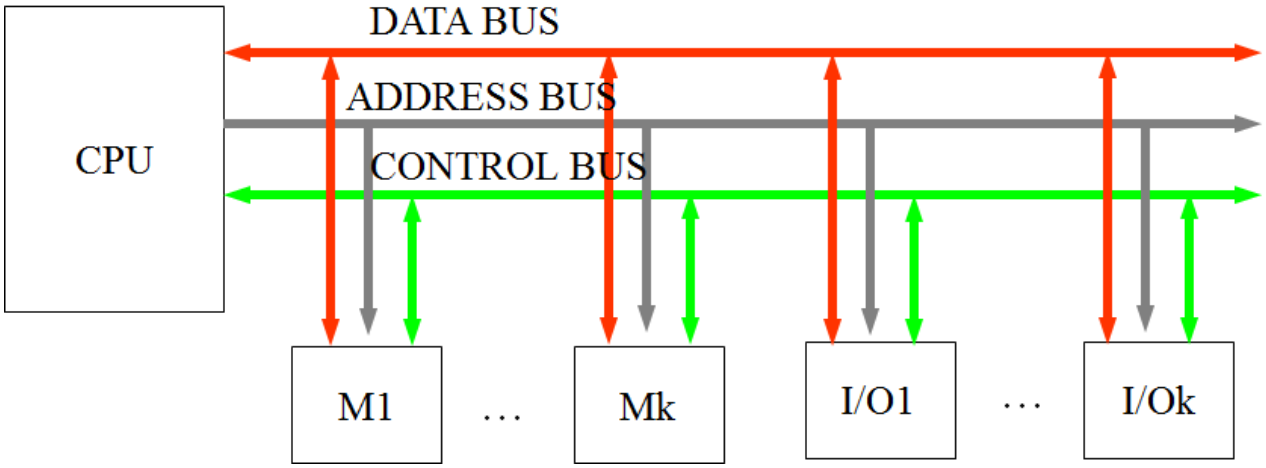


Figura 5: architettura di interconnessione con un unico bus

In particolare nel primo caso, soluzione presentata in Figura 5 si ha un risparmio del numero di linee e di pin del processore, ma ci sono problemi di rallentamento dell'interazione tra il processore e la memoria, normalmente più veloce dei

dispositivi di ingresso/uscita. Ciò è dovuto alla necessità di collegare sugli stessi bus sia le periferiche che le memorie e quindi i bus devono essere realizzati con linee fisiche più lunghe di quelle strettamente necessarie per collegare la memoria esterna con il processore. Ciò comporta un allungamento dei tempi di propagazione del segnale lungo i fili con conseguente allungamento dei tempi di trasmissione dei dati, inoltre si potrebbe presentare il cosiddetto data skew, ovvero disallineamento dei bit trasmessi su linee parallele, ciò è dovuto al fatto che la trasmissione dei bit viene effettuata sul fili differenti e quindi non perfettamente identici nella velocità di trasmissione del segnale, differenza che potrebbe essere significativa qualora i fili sono di dimensione ragguardevole (non a caso i bus vengono realizzati con materiai pregiati). Infine fili lunghi, per poter essere alloggiati in chassis di dimensioni limitate, necessitano di essere piegati o comunque incurvati, ciò comporta che nel caso in cui passino informazioni ad alta frequenza questi fili si possano trasformare in antenne con conseguente formazione di interferenza elettromagnetica che potrebbe inficiare l'affidabilità nella trasmissione dei dati e del funzionamento dei dispositivi posti nelle vicinanze.

Nel caso di utilizzazione di un unico bus c'è poi la necessità di identificare se i dati siano relativi alla memoria o ai dispositivi di ingresso/uscita. Una possibilità è quella di utilizzare un solo set di segnali di controllo per l'interazione con entrambi i tipi di dispositivi, p.e. MR e MW, in tal caso necessariamente si ha una condivisione dello spazio di indirizzamento tra la memoria e i dispositivi di I/O con conseguente diminuzione dello spazio di indirizzamento verso la memoria. Tale soluzione è conosciuta in letteratura come I/O mappato in memoria (memory mapped I/O) ed utilizzata in diverse architetture.

Alternativa al memory mapped I/O è l'utilizzazione di uno o più segnali di controllo aggiuntivi per indicare che l'interazione è diretta verso la memoria o verso un dispositivo di I/O. Per esempio sarebbe sufficiente affiancare ai segnali RD e WR un segnale di controllo che indichi se tale interazione è verso la memoria o meno. In modo alternativo si potrebbero utilizzare due insiemi di segnali di controllo, uno dedicato all'interazione con la memoria (MR e MW) ed uno dedicato all'interazione con i dispositivi di I/O (I/OR e I/OW). Per ridurre il numero di segnali di controllo si potrebbe pensare di utilizzare un segnale di controllo **I/O** che indichi se l'interazione che si vuole fare è verso una periferica o verso la memoria e due segnale per indicare se si vuole fare una lettura (**RD**) o una scrittura (**WR**). In tal caso gli spazi di indirizzamento sarebbero differenti e non ci sarebbero limitazioni sulle dimensioni della memoria. In questo libro faremo utilizzeremo l'ultima possibilità.

Nel caso di utilizzazione di due set di bus distinti, di cui uno dedicato all'interazione con la memoria e il secondo con i dispositivi di I/O, come schematizzato in Figura 6, si ha il vantaggio di poter trasferire i dati verso la memoria alla massima velocità possibile, date le ridotte dimensioni dei collegamenti verso la memoria, pagando naturalmente lo scotto di avere più linee di trasmissione e più pin del processore.

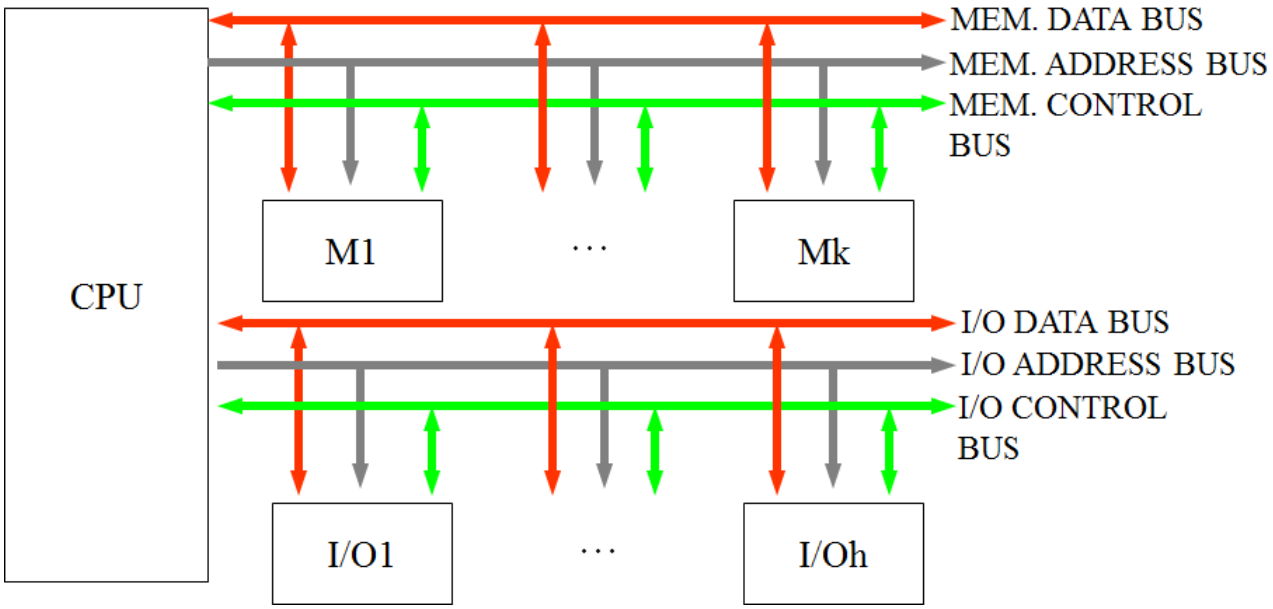


Figura 6: architettura di interconnessione a due bus

Nel proseguo di questo capitolo si utilizzerà questa soluzione, anche se poi una volta sedimentate le basi di conoscenza, si faranno vedere le evoluzioni successive delle modalità di interazione tra il processore e la memoria e le periferiche fino alle soluzioni attualmente utilizzate, che naturalmente potranno a loro volta evolvere verso altre soluzioni.

Nel caso di architettura a due bus, l'architettura interna del processore dovrà essere modificata, in quanto oltre ai registri MAR e MDR, necessari per interfacciare il processore con la memoria è necessario prevedere due registri (I/OAR e I/ODR) necessari per l'interfacciamento con le periferiche di Ingresso e Uscita.

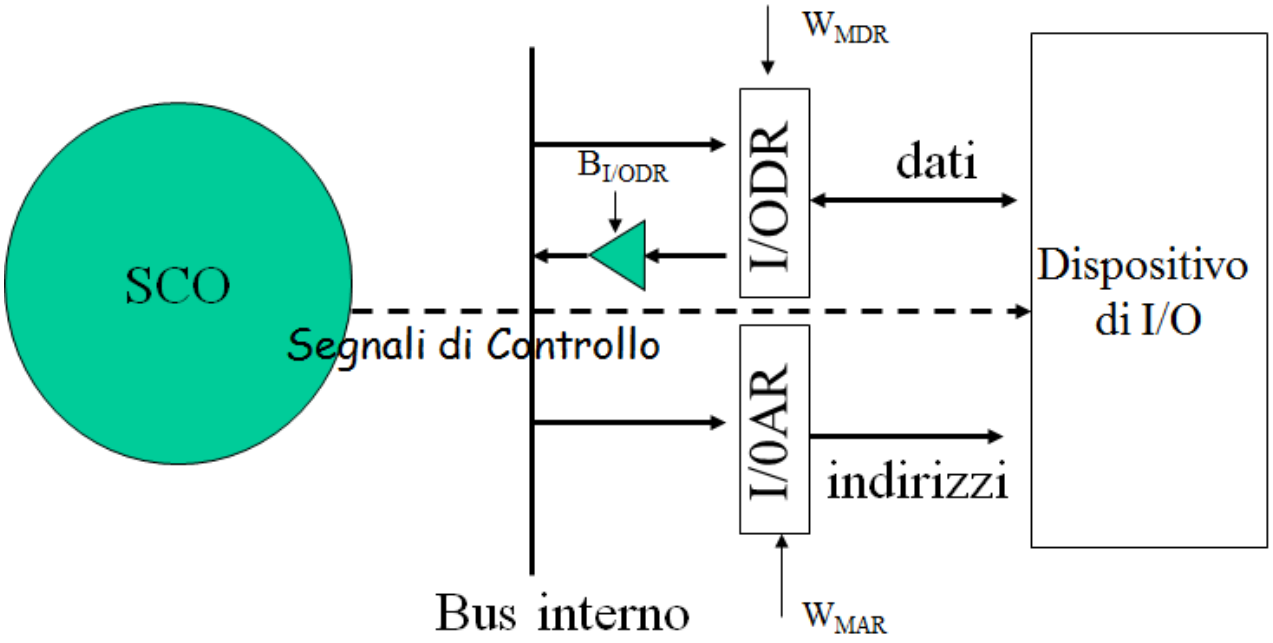


Figura 7: Interfaccia processore verso I/0

Mentre le periferiche avranno una struttura come quella indicata in Figura 8. Per semplicità si prevede che ci sia un solo registro di interfaccia, inoltre si ipotizza che ci sia un decoder che identifica se sull'I/OAB (I/O Address Bus) ci sia indicato o meno l'indirizzo della periferica. Si vuole notare che normalmente il circuito combinatorio riconoscitore di indirizzo presenta molte meno porte logiche di quelle di un decoder , infatti nel caso si volesse utilizzare un decoder standard a 16 bit di ingresso sarebbe necessario un circuito che possa generare 2^{16} uscite, numero davvero eccessivo per questi fini.

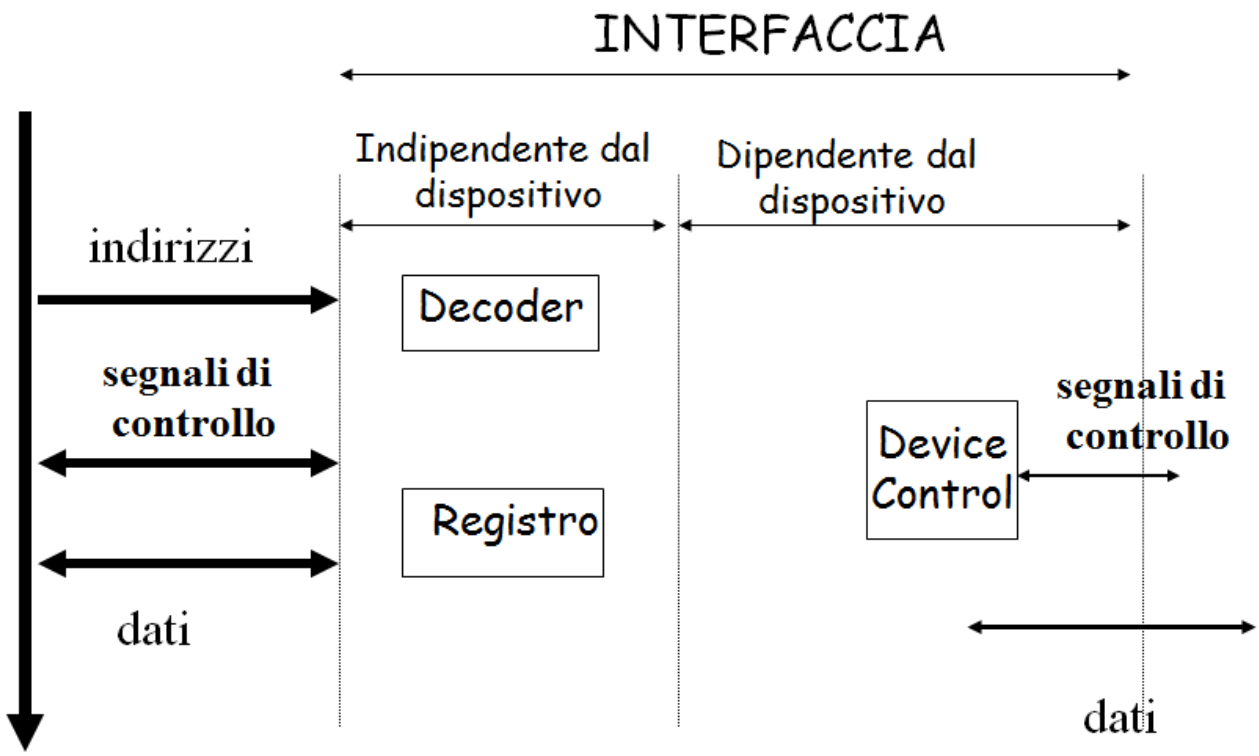


Figura 8: Interfaccia periferiche

Nelle figure 9 e 11 viene presentata l'architettura dell'interfaccia di una semplice periferica di Output e di una periferica di Input, con le relative temporizzazioni. In cui è evidenziato che il protocollo è basato sul riconoscimento e generazione del segnale di controllo WAIT (negato, attivo basso).

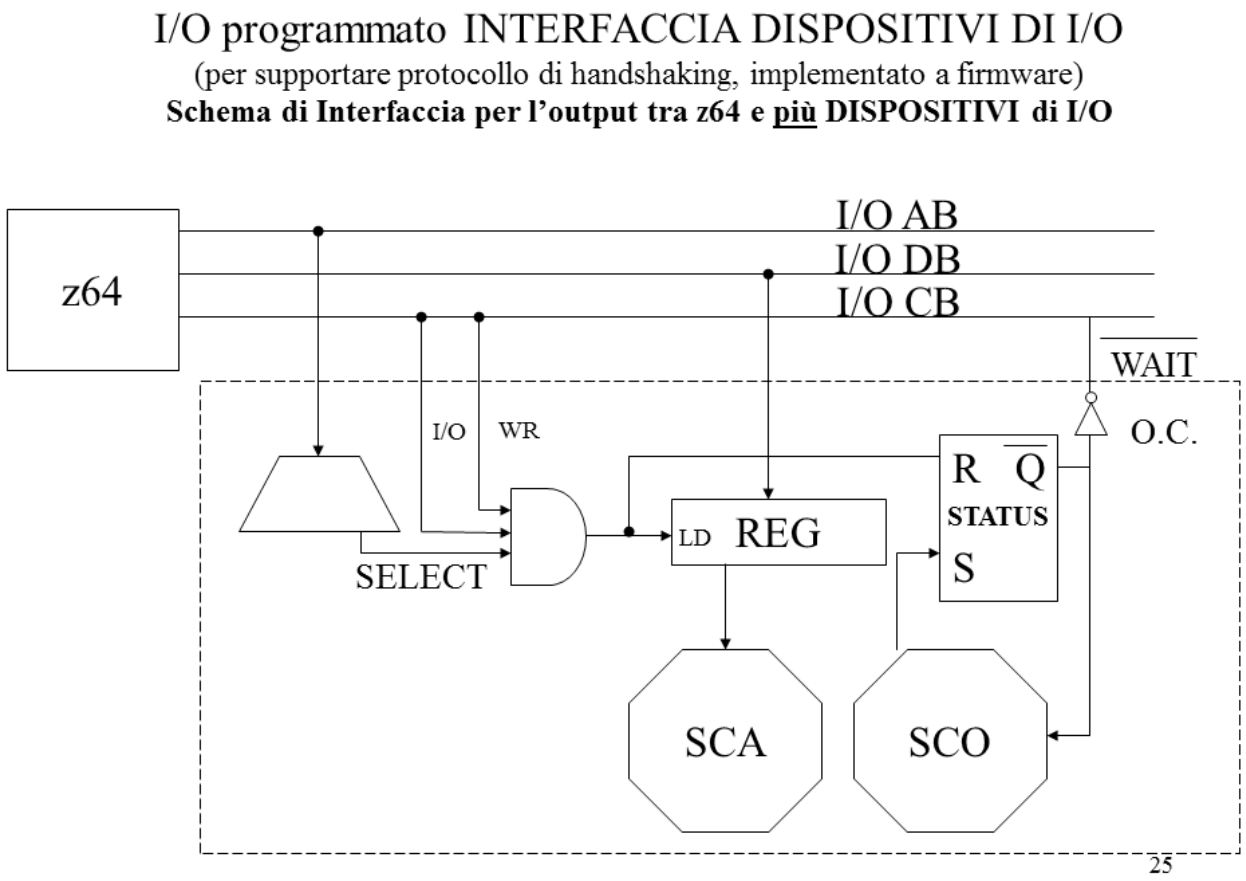


Figura 9: Interfaccia periferica di output

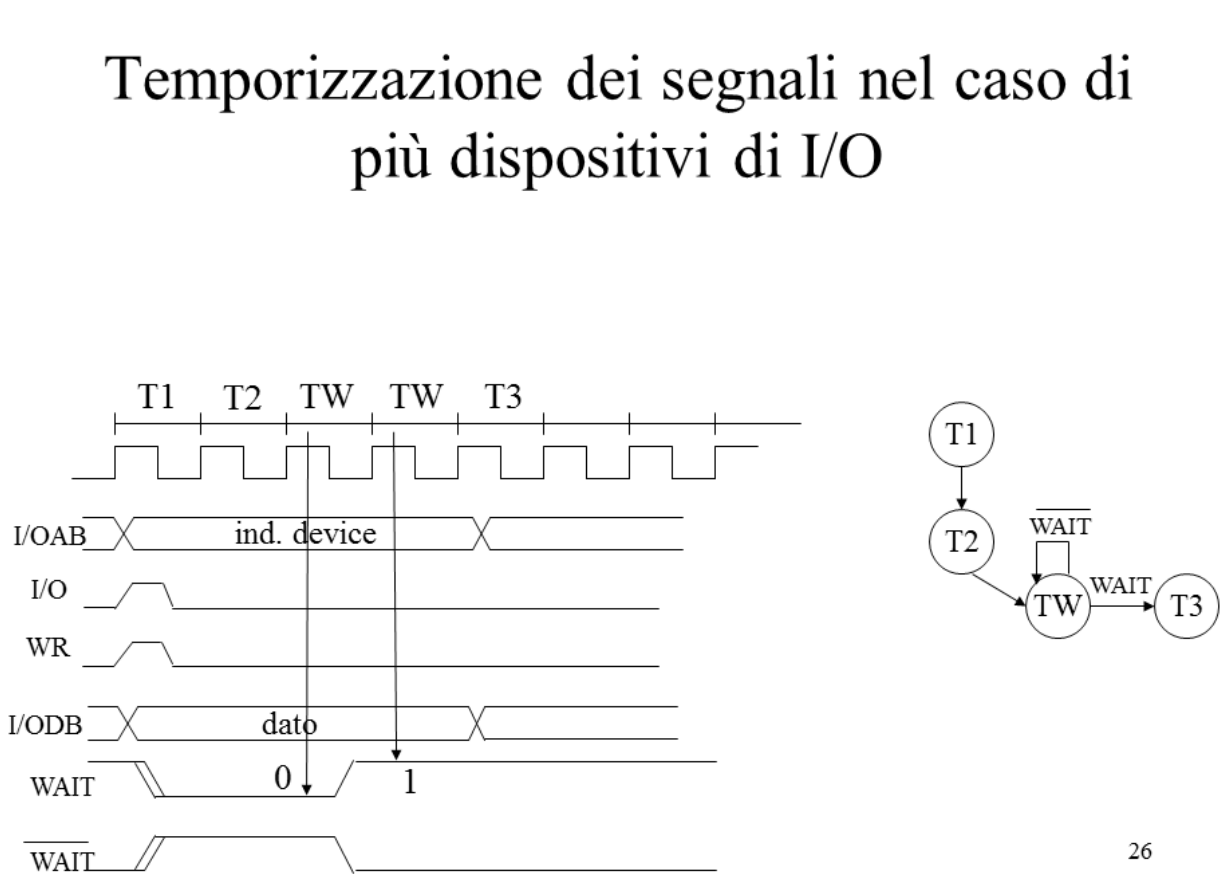


Figura 10: Ciclo di scrittura in periferica

Da notare che nel disegno della Figura 9 la identificazione del dispositivo da indirizzare è effettuata tramite un decoder; in generale questa è una soluzione molto costosa dal punto di vista circuitale, ma la si adotta per la semplicità di comprensione da parte del lettore.

In Figura 10 è rappresentato il ciclo di scrittura in periferica del processore. Durante il primo periodo di clock del SCO, l'indirizzo della periferica su cui bisogna scrivere il dato viene messo sul bus indirizzi delle periferiche (I/OAB), il dato da scrivere viene posto sull'I/ODB e vengono generati i segnali di comando I/O e WR (in questo caso sarebbe sufficiente un solo segnale di comando, e ne mettono due per rendere la soluzione omogenea con quanto si farà nel caso di uso di un solo bus, che è la soluzione attuale dei processori x86 based).

All'inizio del periodo successivo il SCO verifica se la variabile di condizione $\overline{\text{WAIT}}$ è alta o bassa; nel caso di $\overline{\text{WAIT}}$ basso il SCO non effettua alcuna operazione ed attende che questa variabile divenga alta (quando è alta il dispositivo

esterno ha letto dall'I/ODB il dato). Quando verifica che è alta il SCO del processore entra in un ciclo macchina successivo in quanto è certo che la periferica ha letto dal registro interfaccia il dato appena scritto e quindi, eventualmente, il processore potrebbe riutilizzare tale registro per un'altra scrittura.

In figura 11 invece è rappresentata l'interfaccia di una periferica di input e in figura 12 è rappresentata la temporizzazione delle interazioni.

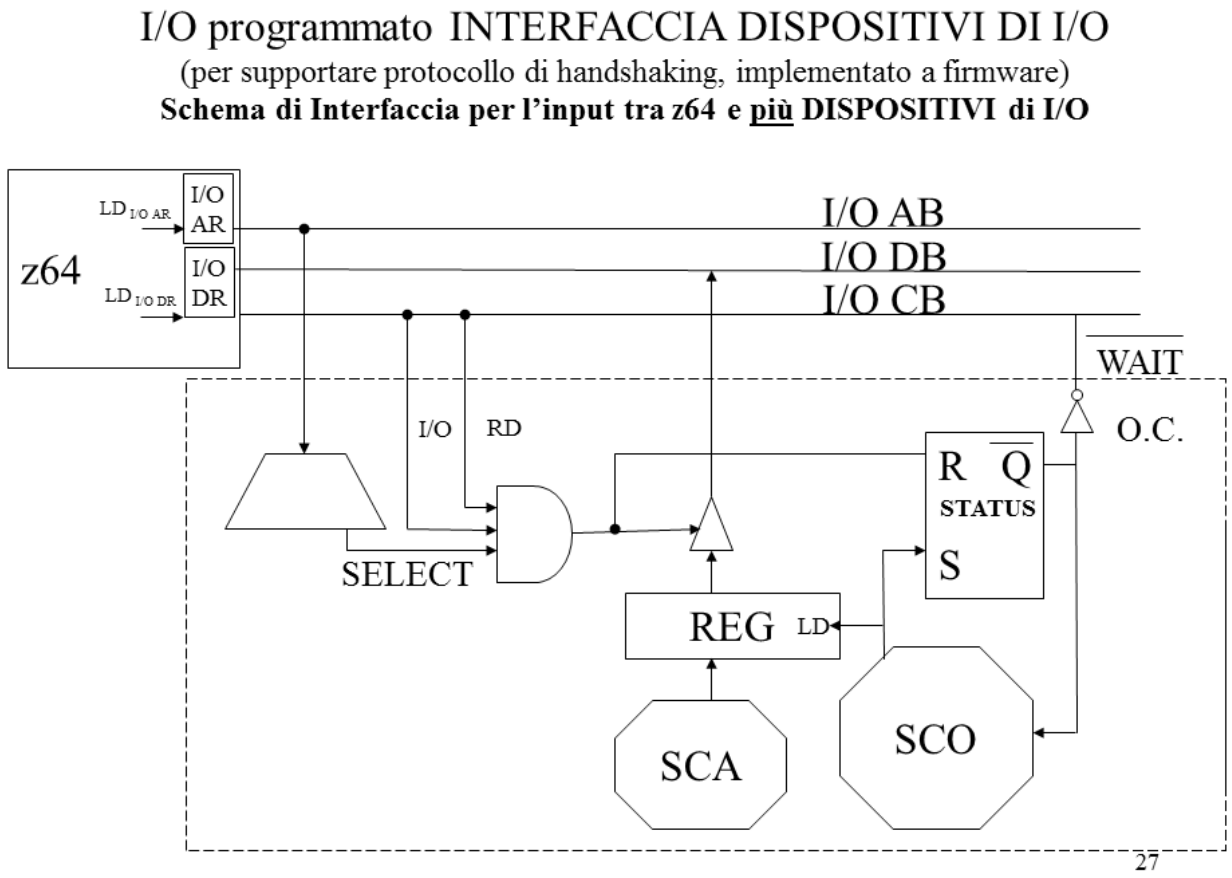


Figura 11: Interfaccia periferica di input

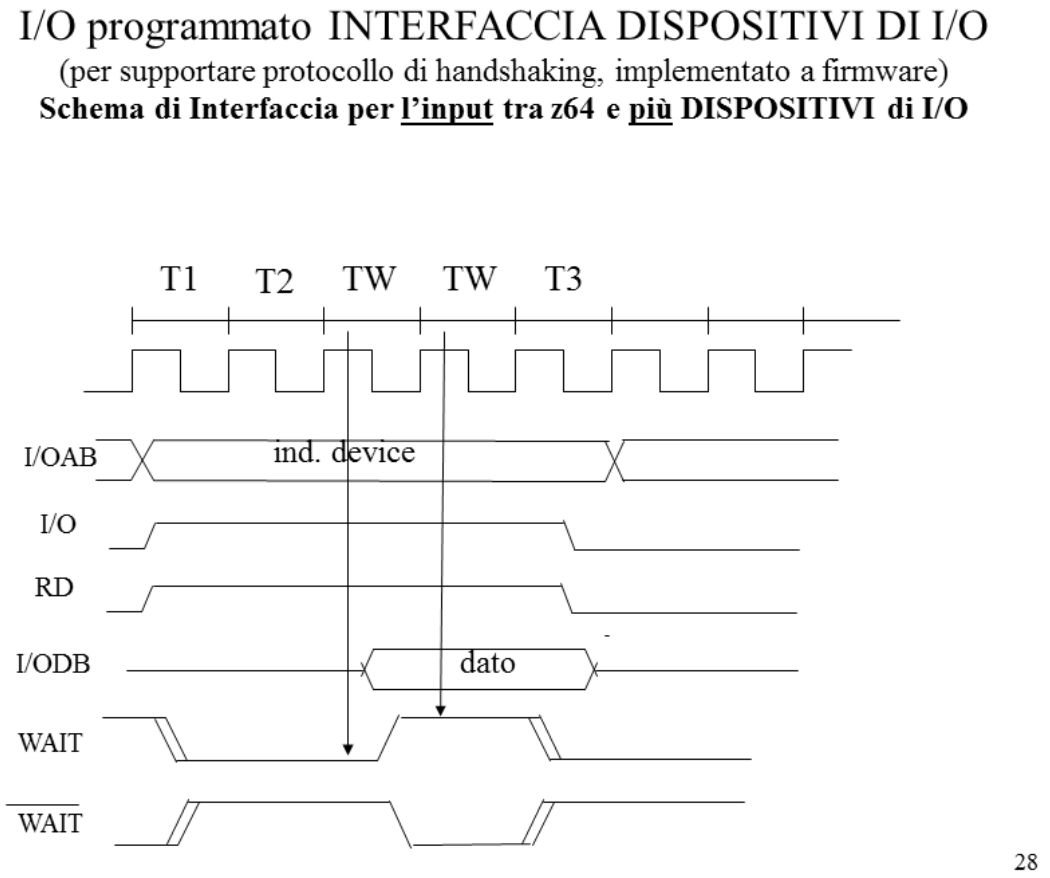


Figura 12: Ciclo di lettura da periferica

In figura 12 è rappresentato il ciclo di lettura da periferica del processore, schematizzato dal microprogramma indicato in figura 11. Durante il primo periodo di clock il SCO del processore genera i segnali di comando I/O e RD e abilita la scrittura dell'indirizzo del registro della periferica sull'address bus (I/OAB) da cui bisogna leggere il dato, notare che tale attività contemporaneamente resetta il F/F Status-1. Nel periodo successivo il SCO verifica se la variabile di condizione $\overline{\text{WAIT}}$ è alta o bassa; nel caso di $\overline{\text{WAIT}}$ bassa il SCO non effettua alcuna operazione ed attende che questa variabile divenga alta (da ricordare che quando è alta il dispositivo esterno ha scritto sul registro di interfaccia il dato richiesto). Quando verifica che la variabile è alta il SCO esce dallo stato di TW e nel periodo successivo memorizza nel registro tampone I/ODR il valore presente sul registro di interfaccia della periferica. Nei successivi periodi di clock il dato letto può essere trasferito e/o manipolato all'interno del SCA del processore. In figura 12 i periodi in cui il SCO è in attesa che il segnale $\overline{\text{WAIT}}$ diventi alto sono indicati con TW (W da

wait). Da notare che così come implementata la macchina a stati finiti, il SCO del processore entrerà almeno una volta nello stato TW.

Lato negativo dell'interazione semi-sincrona è che il SCO del processore potrebbe rimanere nello stato di TW, sia nel caso di lettura che nel caso di scrittura, per un periodo di tempo illimitato, con conseguente inutilizzazione di tale risorsa. Per esempio se il periodo del clock processore fosse di 1 nsec. (10^{-9} sec.) e la velocità di produzione/consumo dei dati della periferica è 1 msec (10^{-3} sec.), valore inferiore ai tempi di funzionamento dei dischi, allora il processore, per ogni interazione, "passa" nello stato di TW per un numero di volte dell'ordine di 10^6 .

Un modo alternativo per evitare ciò è di utilizzare istruzioni assembly di salto condizionato alla condizione dell'uscita del flip/flop STATUS, uscita che potrebbe essere letta tramite una istruzione di input.

Di seguito si farà vedere come implemetare a software il protocollo di handshaking con un'unica periferica in modalità busy waiting e poi si farà vedere come poter interagire con più periferiche in modalità polling.

Inoltre è necessario trovare un modo per poter avvertire la periferica che allertarla che il processore intende interagire con essa, in questo caso si utilizzerà una istruzione di out che avrà come indirizzo destinatario quello del flip/flop di handshaking STATUS.

Interazione busy waiting implementata a software

Per eseguire il protocollo di interazione busy_waiting implementato a software è necessario eliminare gli effetti del segnale di condizione WAIT (negato) del processore. A tal fine è sufficiente mettere a massa (a zero logico) il collegamento (pin) del processore relativo a tale segnale. Notare che in questo caso durante l'esecuzione di una istruzione di IN o di OUT il processore passa una volta solo nello stato WAIT (vedere lucidi precedenti).

Come detto il salto condizionato dipende dall'uscita del valore di un flip/flop delle interfacce delle periferiche, a tal fine si utilizzerà lo stesso flip/flop utilizzato precedentemente per la generazione del segnale di WAIT, che a questo punto, essendo il pin di ingresso corrispondente del processore a massa non serve più

funzionalmente. Quindi il flip/flop di handshaking sarà il F/F STATUS, che potrà essere comandato e letto sia dal processore che dal SCO della periferica.

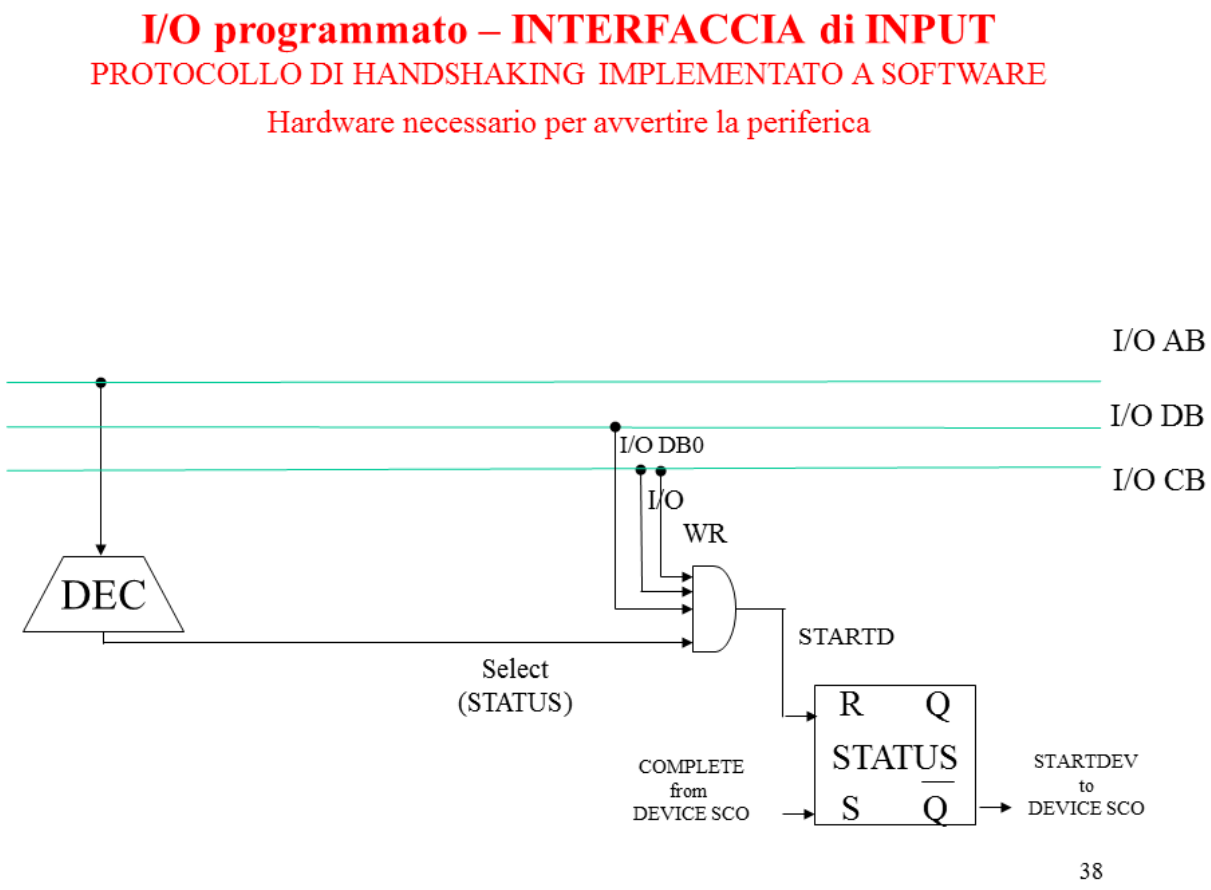


Figura 13-a

Per avvertire la periferica che il processore vuole interagire con essa si resetta il flip/flop STATUS (vedere Figura 13-a). A tal fine è necessario eseguire una istruzione di out dove è necessario identificare l'indirizzo del flip/flop e nell'accumulatore, nel bit meno significativo, sia memorizzato il valore 1. All'atto dell'esecuzione dell'istruzione out vengono posti ad uno sia il segnale di controllo I/O che il segnale di controllo WR.

Invece, per verificare che la periferica sia pronta per l'interazione con il processore, il processore deve leggere l'uscita del flip/flop STATUS (vedere Figura 13-b)

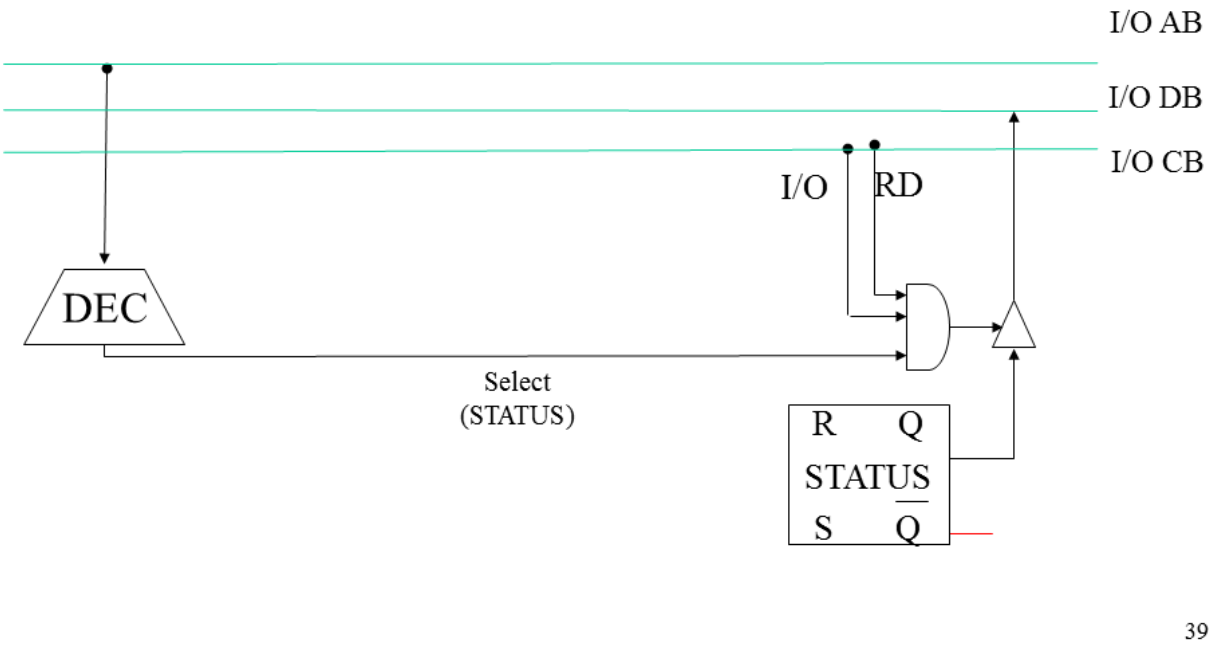


Figura 13-b

Per poter verificare che la periferica è pronta o meno il processore effettua una **in** dell'uscita del flip/flop STATUS e poi verifica se è pari ad 1 o meno (vedere figura 13-b).

L'interfaccia dei dispositivi di ingresso interagenti con il processore in modalità busy waiting implementata a software può essere schematizzata come in figura 14.

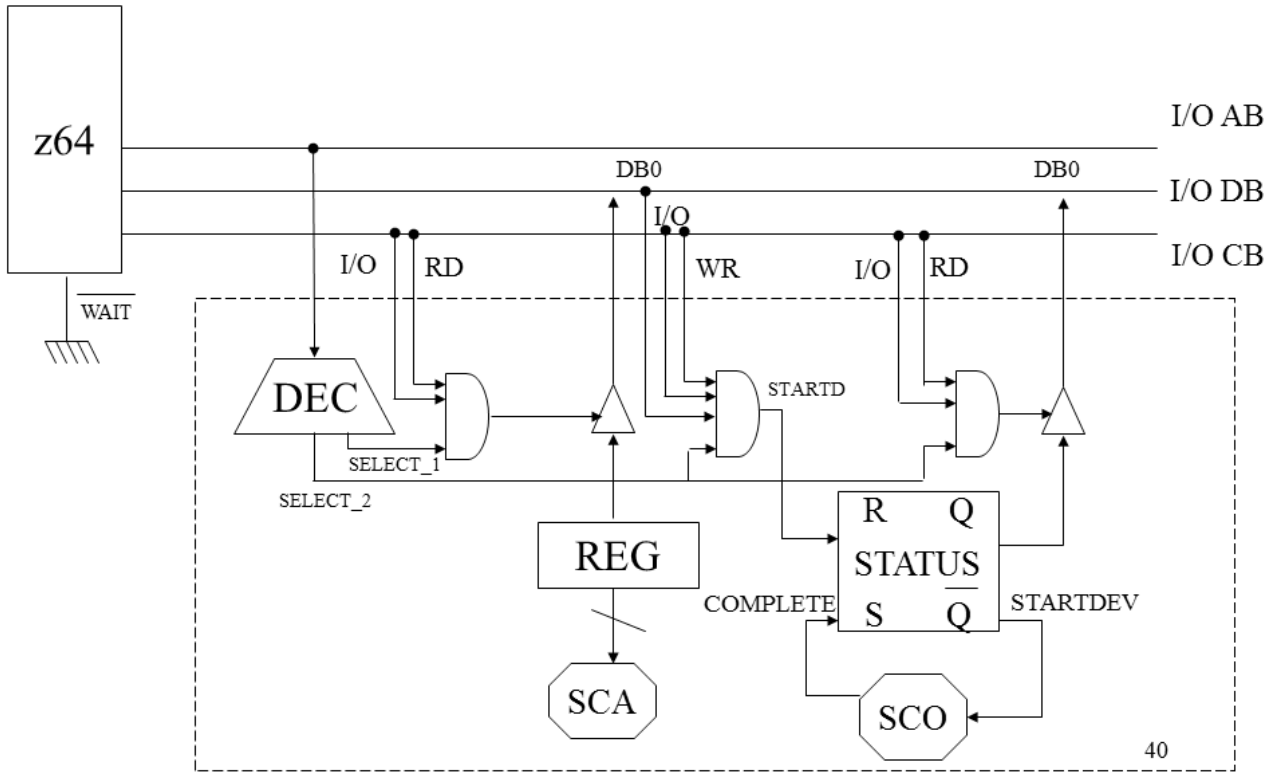


Figura 14: I/O programmato – INTERFACCIA di INPUT

Di seguito è presentato prima il protocollo di interazione e poi il relativo segmento programma assembly che permette l'acquisizione di un dato da una periferica, utilizzando un protocollo di handshaking implementato a software:

I/O programmato – INTERFACCIA di INPUT

PROTOCOLLO DI BUSY WAITING IMPLEMENTATO A SOFTWARE

1. Il processore avverte il dispositivo che vuole un dato da lui (resettando il flip/flop STATUS).
2. Il processore verifica che il dispositivo abbia prodotto il dato, ciò lo verifica testando il flip/flop STATUS.
3. Se il valore di STATUS è 0 il processore deve attendere e pertanto ritorna al punto 2.
4. Se il valore di STATUS è 1 il processore esegue una istruzione di INPUT (seleziona il dispositivo ed invia il segnale di controllo IO RD per trasferire il dato presente in REG all'interno di uno dei registri del processore).

Il segmento di codice che si presenta è relativo al caso in cui l'indirizzo della periferica venga memorizzato in %dx, in tal caso è necessario prevedere un'istruzione di caricamento.

Porzione di programma assembly per lettura di un dato in busy waiting

```
                MOVW $FF_STATUS, %dx
                MOVB $1, %al
                OUTB %al, %dx          # avverti la periferica
Aspetta:        INB %dx, %al
                BTB $0, %al            # copia il bit n.0 del reg.
                                       # A nel carry
                JNC Aspetta            # attendi per. sia pronta
                MOVW $DEVICE_IN, %dx
                INL %dx, %eax          # input da periferica
```

Da ricordare che l'assemblatore per calcolare l'eventuale indirizzo dell'istruzione da eseguire in caso di salto, identificata dall'etichetta (aspetta) a sinistra dell'istruzione stessa, converte lo spiazzamento associato all'identificativo (aspetta) a destra del codice mnemonico dell'istruzione come differenza tra il valore del contenuto in rip dopo il fetch e l'indirizzo assegnato precedentemente dall'assemblatore all'etichetta a sinistra dell'istruzione.

L'interfaccia dei dispositivi di output interagenti con il processore in modalità busy waiting implementata a software può essere schematizzata come in figura 15

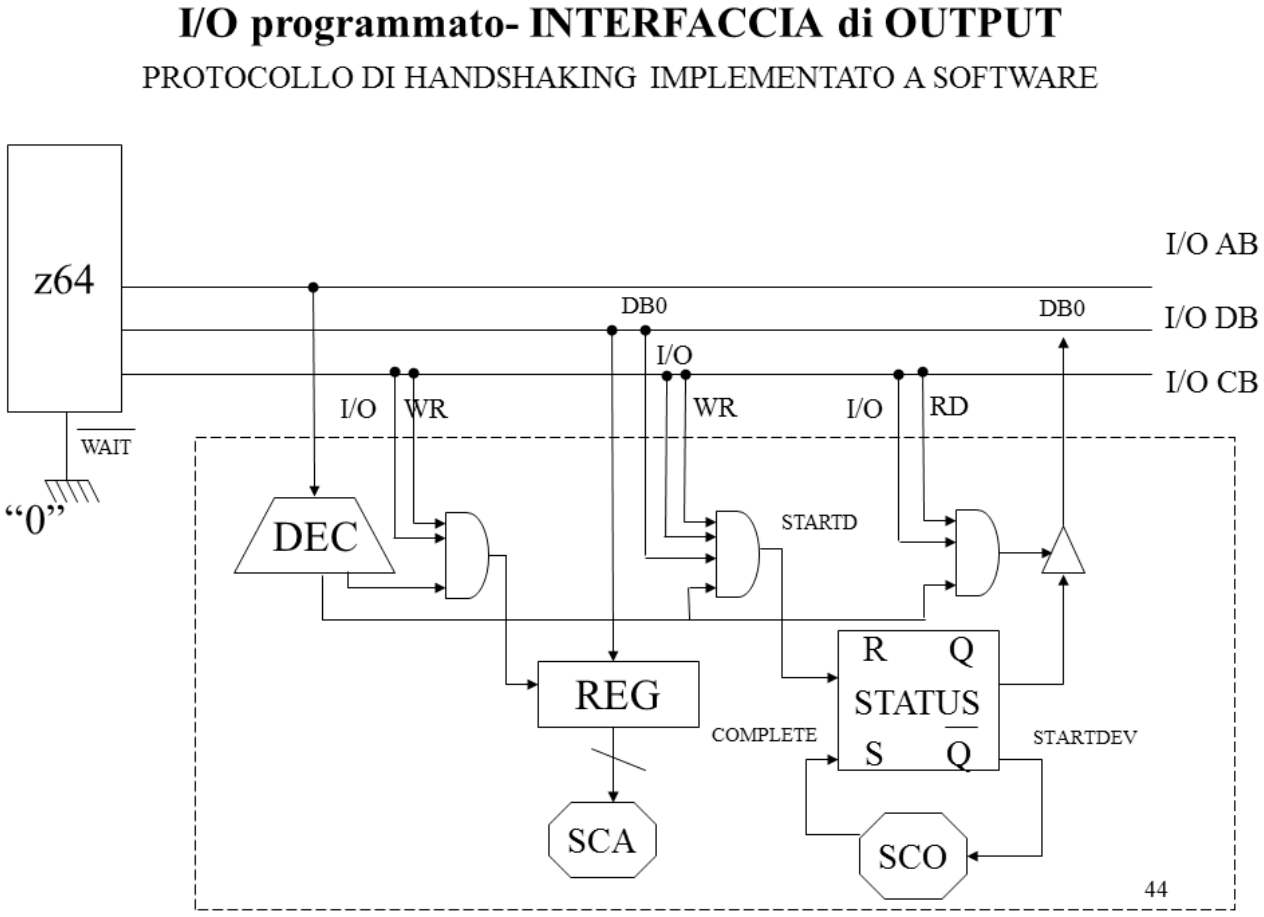


Figura 15: I/O programmato – INTERFACCIA di OUTPUT

Protocollo di Output

1. Il processore esegue una istruzione di OUTPUT e trasferisce il contenuto di un registro nel registro di interfaccia del dispositivo (mediante il segnale di controllo I/OWR).
2. Il processore avverte il dispositivo che vuole un dato da lui (resettando il flip/flop STATUS).
3. Il processore verifica che il dispositivo abbia prodotto il dato, ciò lo verifica testando il flip/flop STATUS.
4. Se il valore di STATUS è 0 il processore deve attendere e pertanto ritorna al punto 2.
5. Se il valore di STATUS è 1 il processore esce dallo stato di attesa e può proseguire le sue attività

Protocollo di Output

HP: la periferica è ad uso esclusivo del processore e il processore attende in busy waiting il consumo del dato che ha trasmesso alla periferica

1. il processore esegue una istruzione di OUTPUT e trasferisce il contenuto di un registro nel registro di interfaccia del dispositivo (mediante una istruzione di output che genera i segnali di controllo I/O e WR, mette l'indirizzo della porta sull'AB e il valore da scrivere sul DB).
2. Il processore avverte il dispositivo che gli ha trasferito un dato. Ciò viene fatto resettando il flip-flop STATUS e il flip/flop rimane in tale stato per tutta la durata delle operazioni di consumo del dato da parte del dispositivo. Il reset del flip/flop viene effettuato mediante una istruzione di output che genera i segnali di controllo I/O e WR e mette l'indirizzo della porta sull'AB, notare che sull'AB c'è un valore non specificato, ma ciò non comporta alcun problema dato che in questa fase non si usa il valore del DB. Quando il dato è stato letto dallo SCO della periferica dal registro di interfaccia (REG), il dispositivo genera il segnale COMPLETE, settando il flip/flop STATUS.
3. Nel frattempo il processore, in attesa, può esaminare lo stato del flip/flop campionandone l'uscita, ciò viene fatto leggendone l'uscita (mediante una istruzione di input che genera i segnali di controllo I/O, RD e mette l'indirizzo della porta sull'address bus) e verificandone, con altre istruzioni che non interessano la periferica, il valore tramite uno shift e un controllo del valore del carry (JC).
4. Se CARRY= 0 il processore deve attendere ed eventualmente tornare al punto 3.
5. Se CARRY= 1 il processore può eseguire un'altra istruzione.

Pertanto per interagire con la periferica, rispettando il protocollo, è necessario un segmento di programma del tipo:

Porzione di programma assembly handshaking per un dato

```
                MOVL $dato, %eax
                MOVW $DeviceOUT, %dx
                OUTL %eax, %dx              # out

                MOVW $FF_STATUS, %dx
                MOVB $1, %al
                OUTB %al, %dx               # start

Aspetta:        INB %dx, %al
                BTB $0, %al         # copia il bit n.0 del reg.
                                    # AL nel carry
                JNC Aspetta                 # jnr
```


Acquisizione di N word da una periferica in busy waiting

Di seguito viene presentato un sottoprogramma assembly che utilizzando l'handshaking, implementato a software, acquisisce N word da una periferica

I/O programmato
MODALITA' BUSY WAITING

```
…
movl $0, %ecx                    # contatore dei dati inizializzato a 0
movq $DATI, %rdi                 # inizializzazione della locazione di memoria
                                 #          dove memorizzare i dati acquisiti
.loop:
movw $AD_STATUS, %dx
movb $1, %al
outb %al, %dx                    # avvia la periferica a produrre dati
.bw:
inb %dx, %al
btb $0, %al
Jnc .bw                          # attendo che la periferica sia pronta
movw $AD_REG, %dx                # inizializzazione del registro di porta di I/O
inw %dx, %ax                     # prelevo il dato dalla periferica…
movw %ax, (%rdi, %rcx, 2)        # …lo copio in memoria (nel vettore)
addl $1, %ecx                    # incremento il contatore
cmpl $100, %ecx
jnz .loop
hlt
```

49

Come si può vedere dal codice dell'esempio precedente per acquisire 100 dati il processore deve attendere 100 volte che la periferica produca i dati e se la periferica, per esempio, ha una velocità di produzione dei dati 5 ordini di grandezza inferiore a quella del processore (p.e. processore che può eseguire 10^8 istruzioni al secondo e periferica che ne produce 10^3 al secondo), il processore sarebbe inutilizzato per attività "produttive" per tutto il tempo di produzione dei dati. Per ovviare a ciò, nell'ipotesi che la periferica non fosse pronta a spedire il dato si potrebbe pensare di far fare delle altre attività al processore, per esempio cercare di ottenere dati da altre periferiche. Ed è proprio questa esigenza che ha portato alcuni progettisti di acquisire dati dalle periferiche in modalità ***polling***.

Il termine polling viene dal fatto che come dice la periferica il processore interroga ciclicamente le periferiche da cui può prendere dei dati.

Di seguito viene presentato un programma assembly che permette l'acquisizione di N dati dalla prima delle periferiche pronta a trasferire i dati.

Esempio relativo all'acquisizione dati da 2 periferiche

```
    movl $0, %ecx              # contatore dei dati inizializzato a 0
    movb $1, %al               # valore '1' per avviare le periferiche
    movw $AD1, %dx
    outb %al, %dx              # Avvia AD1
    movw $AD2, %dx
    outb %al, %dx              # Avvia AD2

  .poll:
    movw $AD1, %dx
    inb %dx, %ax               # leggo il valore di STATUS da AD1
    btb $0, %ax
    jc .acquisisci_dati1       # se è pronta acquisisco dati
    movw $AD2, %dx
    inb %dx, %ax               # leggo il valore di STATUS da AD2
    btb $0, %ax
    jc .acquisisci_dati2       # se è pronta acquisisco dati
    jmp .poll                  # proseguo con il ciclo di polling

  .acquisisci_dati1:
    movw $AD1_REG, %dx
    call acquisisci            # chiama la routine di acquisizione
    movb $1, %al       # valore '1' per avviare le periferiche
    movw $AD1, %dx
    outb %al, %dx              # riavvia AD1
    jmp .poll                  # proseguo con il ciclo di polling

  .acquisisci_dati2:
    movw $AD2_REG, %dx
    call acquisisci            # chiama la routine di acquisizione
    movb $1, %al       # valore '1' per avviare le periferiche
    movw $AD2, %dx
    outb %al, %dx              # riavvia AD2
    jmp .poll                  # proseguo con il ciclo di polling

acquisisci:
    inw %dx, %ax               # prelievo del dato e…
```

```
  movw %ax, vettore(, %ecx, 2)# … suo trasferimento in memoria
  addl $1, %ecx              # incremento del contatore
  cmpl $100, %ecx            # controllo di uscita dal ciclo di polling
  jnz .return
      hlt
.return:
  ret
```

Comunque anche in questo caso il processore deve sempre verificare che la periferica sia pronta o meno, andando a testare il flip/flop di STATUS, con ovvia perdita di tempo.

Inoltre questa modalità di interazione non garantisce la possibilità di gestire in maniera efficiente ed affidabile eventuali esigenze di interazione di periferiche non previste nel polling che si sta effettuando. Per esempio un programma sta acquisendo dati da 4 periferiche, ma un'altra avrebbe la necessità di informare il processore che l'alimentazione elettrica si sta esaurendo, per esempio per esaurimento della batteria, e che è necessario sospendere in modo sicuro il processamento.

Un modo alternativo per interagire con le periferiche e nel contempo risparmiare tempo in inutili attese da parte del processore è di far avvertire il processore direttamente dalla periferica che il dato o è stato consumato o è stato prodotto, a seconda se è la periferica è di uscita o di ingresso o che è sorto un evento che è necessario gestire. Tale modalità di interazione è denominata **interrupt** e nel prossimo paragrafo la presenteremo nei suoi dettagli, così come implementata nel processore che stiamo progettando.

Però prima di descrivere tale tipo interazione è opportuno sottolineare che laddove il processore necessita di ricevere/trasmettere un insieme di dati da/a una periferica da memorizzare/prelevare sequenzialmente in/dalla memoria, quali una stringa o un file, per poter continuare la sua evoluzione, allora dovrebbe in prima approssimazione attendere in modalità busy waiting che si completi tale trasferimento come visto precedentemente. Nell'x86 e quindi anche nel processore che stiamo progettando sono a tal fine previste due istruzioni (***InsX*** e ***OutsX***) che

permettono proprio tale tipo di trasferimenti. Naturalmente tali attività sono bloccanti per il programma e quindi, per quello che abbiamo visto fino ad ora su come funziona il sistema di elaborazione, anche il processore sarà bloccato, con evidente spreco di tale risorsa. Per evitare ciò sono stati introdotti accorgimenti quali i canali di comunicazione e i S.O., ma al momento per consentire una acquisizione graduale delle conoscenze continueremo ad ipotizzare l'assenza delle funzionalità dei S.O.

InsX (Input from Port to String)

Trasferisce un numero di dati (il cui formato è specificato dalla X, che può essere B, W, L o Q) pari al valore memorizzato nel registro rcx dalla porta di ingresso identificata dal contenuto del registro dx nelle locazioni di memoria il cui indirizzo iniziale è memorizzato nel registro rdi.

OutsX (Output from String to Port)

Trasferisci un numero di dati (il cui formato è specificato dalla X, che può essere B, W, L o Q) pari al valore memorizzato nel registro rcx dalle locazioni di memoria il cui indirizzo iniziale è memorizzato nel registro rdi alla porta di uscita identificata dal contenuto del registro dx.

Notare che mentre i registri relativi agli indirizzi di memoria e del numero di dati da trasferire sono memorizzati in registri da 64 bit, l'identificativo (indirizzo) della porta di ingresso o di uscita è memorizzato in un registro di 16 bit. Tale scelta deriva dal fatto che si è ritenuto più che sufficiente avere a disposizione un numero porte di ingresso ed altrettante di uscita pari a 64K per connettere tutti i dispositivi di ingresso ed uscita.

Per implementare tali istruzioni è necessario che il microprogramma, relativo al codice operativo di tali istruzioni, emuli l'esecuzione dei frammenti di programma assembly come quello visto acquisisce per l'acquisizione di N word da una periferica, dove vengono trasferiti in busy waiting N dati da una periferica verso la memoria. Naturalmente il firmware, durante l'esecuzione del microprogramma non dovrà fare il fetch di tutte le istruzioni precedenti, ma di una sola (si ricorda che il fetch per identificare il tipo di istruzione da eseguire è già stato fatto), dovendo comunque implementarne la logica operativa. In particolare, per esempio, nel caso di acquisizione di 100 Byte (dato memorizzato nel registro rcx) dalla porta di

ingresso 1111 (memorizzato nel registro dx) a partire dalla locazione di memoria (memorizzato nel registro rdi), il microprogramma dovrà:

- Avvertire la periferica che desidera un dato
- Attendere che il dato sia pronto
- Acquisire il dato (equivalente a IN)
- Mettere il dato nella locazione di memoria il cui indirizzo è dato dal valore del registro rdi (equivalente a MOV)
- Incrementare di 1 (essendo il dato un Byte) il valore del registro rdi (equivalente a ADD)
- Decrementare il contenuto del registro rcs (equivalente a SUB)
- Verificare se il contenuto del registro rcs è pari a zero
- Se pari a zero effettuare il fetch dell'istruzione successiva, se pari ad uno ritorna alla prima attività (avvertire la periferica che desidera un dato)

Mentre nel segmento di programma scritto dal programmatore l'acquisizione in busy waiting di N dati effettivamente va ad alterare il contenuto dei flip/flop dello Status Register interno al procesore, dato che è necessario verificare se il registro contatore sia arrivato a zero, l'acquisizione in busy waiting implementata tramite l'esecuzione dell'istruzione INS sfrutta l'opportunità di disabilitare l'aggiornamento del registro FLAGS tramite l'apposito segnale di controllo W_{FLAGS}.

3.4 GESTIONE DEGLI INTERRUPT

Scopo di una interruzione è quello di permettere ad un dispositivo esterno di far sospendere l'esecuzione del programma in esecuzione e di indurre la CPU, se predisposta, ad iniziare un programma di servizio relativo alla richiesta stessa.

L'interruzione di un processore generato da una periferica è simile alla ricezione di una telefonata da parte di una persona.

Così come l'arrivo di una telefonata può essere schematizzata in queste fasi

- il richiedente chiama il ricevente (squillo del telefono);
- il ricevente alza la cornetta;
- il ricevente avverte il richiedente che vuol sapere (con "pronto") chi è l'interlocutore;
- il richiedente si identifica
- inizia il colloquio

così l'arrivo di una interruzione può essere schematizzata similmente:

- generazione della richiesta di interruzione e suo arrivo al processore (IRQ, attivo basso)
- il processore si avvede della presenza della richiesta
- il processore avverte il richiedente che ha capito che c'è stata una richiesta di interruzione (INTA, INTerrupt Acknowledgment)
- il richiedente si identifica
- il processore basandosi sull'identificazione del chiamante esegue il programma/subroutine (DRIVER) relativo all'interruzione.

Notare che l'identificativo del programma di servizio da eseguire viene identificato dalla periferica e come nelle interazioni umane al momento dell'identificazione del chiamante il ricevente risponderà in modalità differente in funzione dell'identificativo dell'interlocutore stesso.

Naturalmente così come una persona per poter ricevere una telefonata deve avere il telefono connesso alla rete o acceso ed inoltre la telefonata non dovrebbe interferire con le attività in corso (per esempio il ricevente stata preparando la cena e quindi dopo la telefonata potrà completare le relative attività connesse alla preparazione della cena), così anche per poter gestire una interruzione è necessario:

- che la CPU sia disponibile ad essere interrotta (interruzioni abilitate);

- che l'interruzione del programma corrente non comporti un'interferenza sulla correttezza della sua esecuzione.

Di seguito faremo vedere come lo z64 gestisce l'abilitazione delle interruzioni e garantisca che l'esecuzione di un programma di servizio non interferisca con quello corrente e quale tecnica si utilizza per identificare la sorgente dell'interruzione e quindi il programma (subroutine/driver da attivare).

Si ipotizzerà che la richiesta di interruzione avvenga tramite il segnale IRQ (negato) e che il processore avverte il richiedente che ha capito che c'è stata una richiesta di interruzione con il segnale di controllo INTA.

Anche in questo caso, essendo il segnale di controllo IRQ (attivo basso) generabile da più dispositivi esterni, è opportuno connettere le richieste di interruzione verso il processore in modalità wired OR, come schematizzato in Figura 16. In questo modo ovviamente si ha un risparmio nel numero di porte logiche necessarie per la

propagazione delle richieste di interruzione e inoltre la soluzione è scalabile e facilmente implementabile dal punto di vista elettronico.

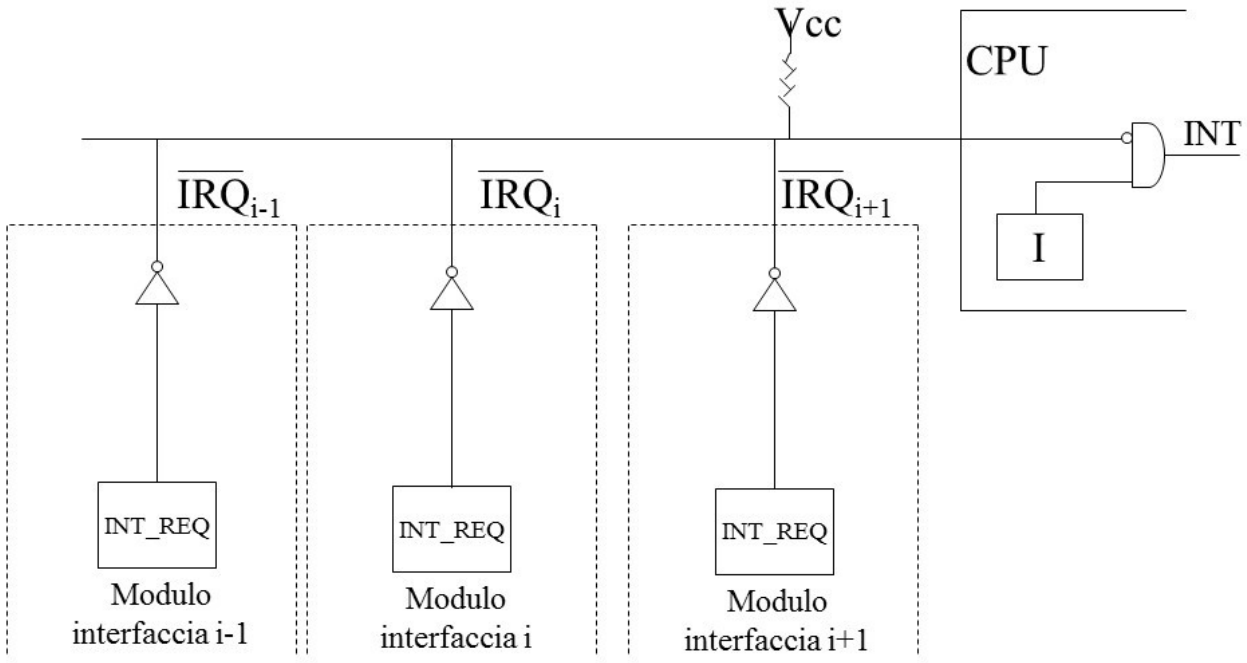


Figura 16: connessione in wired or delle richieste di interruzione

Abilitazione/disabilitazione delle interruzioni

Per garantire la corretta esecuzione di un segmento di programma o per garantirne la sua esecuzione in un fissato intervallo di tempo alcune volte è necessario che la sua esecuzione da parte della CPU venga fatta in modo non interrompibile. Per memorizzare l'informazione che le interruzioni siano o meno abilitate si fa uso di un flip-flop (denominato *I* e contenuto nel registro SR). Il contenuto di questo flip-flop può essere manipolato dalle istruzioni assembly ***clrI*** e ***setI***.

Verifica richiesta delle interruzioni

La richiesta di un'interruzione avviene in modo asincrono rispetto alle attività del processore e quindi del suo SCO. Le attività del SCO sono scandite da un clock

e si è visto nel Paragrafo precedente che il controllo del SCO una volta eseguito un microprogramma relativo ad una istruzione macchina ritorna al microprogramma relativo alla fase di fetch (l'insieme fetch ed esecuzione di una istruzione viene anche detto *ciclo istruzione*). Per non complicare ulteriormente l'organizzazione del SCO, se le attività correnti sono interrompibili, la verifica della presenza della richiesta di un'interruzione viene fatta alla fine di ogni ciclo istruzione, questo perché in questo modo è necessario salvare solo lo stato del programma in esecuzione (corrente) codificato in linguaggio macchina, altrimenti sarebbe stato necessario salvare quello del microprogramma.

Salvataggio dello stato

Prima di eseguire il programma di servizio la CPU deve mettersi nelle condizioni di poter riprendere le attività interrotte, una volta eseguito il programma di servizio. Questo modo di operare è simile a quello visto in precedenza nella gestione dei sottoprogrammi, quindi prima che la CPU possa iniziare l'esecuzione del programma di servizio il SCO deve salvare il contenuto del RIP e il contenuto dello SR in una zona di memoria predefinita. Poiché si prevede che la gestione di una interruzione possa essere a sua volta interrotta da un'altra interruzione è necessario prevedere che la gestione della memoria usata per salvare lo stato della CPU venga fatta come uno stack. Il salvataggio del contenuto del PC e del SR è effettuato dal SCO nelle locazioni di memoria in cima allo stack. Per motivi di efficienza il puntatore dello stack deve essere memorizzato in un registro interno della CPU. Il registro RSP viene specializzato per questa finalità.

E' da notare che il contenuto del ***rip*** deve essere salvato necessariamente dal SCO, in quanto per poter attivare il programma di servizio richiesto dall'interruzione è necessario caricare l'indirizzo della sua prima istruzione nel ***rip***, e quindi se non la si salva precedentemente questa informazione viene persa. Il contenuto dello SR potrebbe anche essere salvato via software, ma poiché, in generale, l'esecuzione di un programma di servizio comporta la modifica di questo registro tanto vale salvarlo via SCO guadagnandone in velocità. Eventualmente potrebbero essere salvati anche i contenuti di tutti i registri interni visibili dall'esterno via SCO (gestione salvataggio via *hardware*), ma in generale i programmi di servizio ne usano solo un sottoinsieme, quindi conviene farlo effettuare di volta in volta dai programmi di servizio (gestione salvataggio via *software*, tramite l'esecuzione di istruzioni di tipo PUSH).

Identificazione sorgente dell'interruzione

L'identificazione del dispositivo esterno che ha effettuato la richiesta dell'interruzione può essere fatta via software o via hardware. Nello z64 si è optato per il secondo tipo di identificazione.

Una volta riconosciuta la presenza di un'interruzione il SCO genera un segnale di controllo, *Interrupt Acknowledgement* (INTA), per avvertire il dispositivo esterno che è in grado di ricevere sull'I/O Data Bus (I/ODB) l'identificazione del driver da attivare. Notare che ci potrebbero essere più dispositivi che hanno fatto una richiesta di interruzione, ciò può verificarsi se più dispositivi effettuano una richiesta di interruzione durante il periodo di esecuzione di una istruzione (da ricordare che la presenza di richieste di interruzione viene effettuata alla fine dell'ultimo ciclo macchina di ogni istruzione), ovvero durante tutto il periodo di tempo in cui le interruzioni sono disabilitate (che corrisponde allo stato zero del flip/flop I). Quindi ci potrebbe essere un'interferenza in scrittura sul bus dell'identificazione del driver da attivare, per evitare tale problema c'è la necessità di un meccanismo che serializzi le identificazioni dei driver da eseguire. Ci sono diverse soluzioni basate su dispositivi esterni che permettono ciò, prima si farà vedere la soluzione più banale, ma che non garantisce la fairness e potrebbe creare problemi di starvation, dopodiché si farà vedere una soluzione che permette di implementare politiche basate sulla priorità delle periferiche.

L'identificativo del driver da attivare, memorizzato nella periferica, è utilizzato per indirizzare il programma di servizio. Per motivi di sicurezza e di efficienza le periferiche non conoscono l'indirizzo dei driver che possono richiedere di attivare, ma solo il loro identificativo. Considerando il numero limitato di driver che possono essere attivati, normalmente si utilizzano 8 bit per l'identificazione dei driver. Questi 8 bit sono utilizzati per identificare l'indirizzo iniziale del driver da eseguire. Ciò viene effettuato tramite l'utilizzazione dell'Interrupt Descriptor Table (IDT), in cui sono memorizzati gli indirizzi iniziali dei driver, in particolare ogni identificativo, identificato come Interrupt Vector Number (IVN), permette di accedere alla locazione dell'IDT dove prelevare l'indirizzo iniziale del codice del driver.

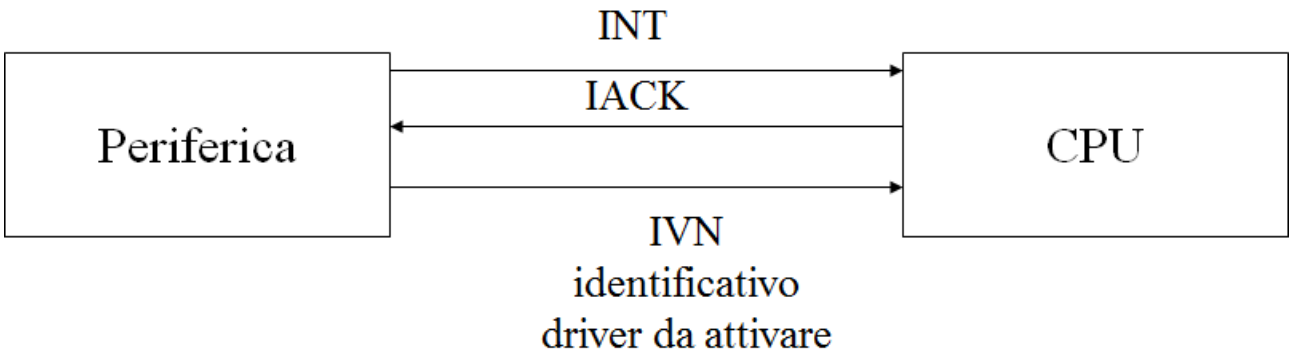


Figura 17: connessioni periferica-processore per gestire richieste di interruzione

L'IDT è memorizzato a partire dalla locazione di memoria $(00000000)_H$, tale vettore è costituito di 8 byte x N, dove N è il numero di driver differenti, gli 8 byte servono per identificare l'indirizzo della memoria da cui inizia il driver, pertanto al componente *i-esimo* dell'IDT è associato l'indirizzo iniziale del programma di servizio relativo al driver *i*. Per identificare la posizione del vettore in cui è memorizzato l'indirizzo del programma di servizio si moltiplica per otto l'identificatore del dispositivo esterno. Nell'implementazione del processore z***64*** il numero massimo di programmi di servizio è pari a $2^8 = 256$.
E' compito del programmatore di sistema associare ad ogni locazione del vettore delle interruzioni l'indirizzo del relativo programma di servizio.
Naturalmente si poteva scegliere un qualunque numero di programmi di servizio da attivare, date le finalità didattiche del processore si è ritenuto 256 un numero adeguato.

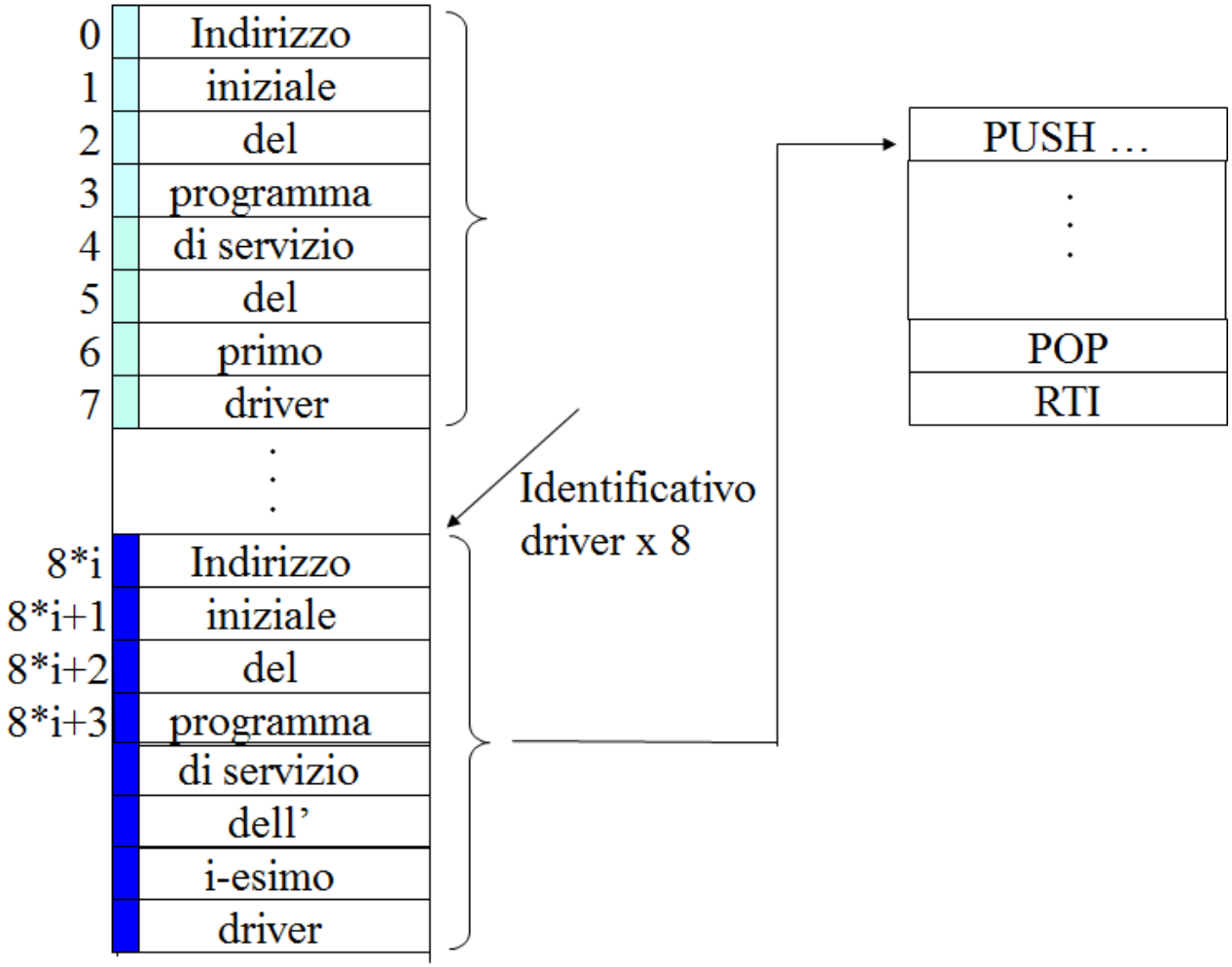


Figura 18: Tabella degli identificatori indirizzi dei driver - IDT

Esecuzione del programma di servizio

Per garantire che l'interruzione dell'esecuzione del programma corrente non comporti un'interferenza sulla sua evoluzione è necessario che, oltre alle informazioni salvate dal SCO (contenuto del ***rip*** e ***rlags***), nello stack vengano anche memorizzati i contenuti di quei registri che il programma di servizio modificherà. Questo salvataggio (tramite l'esecuzione di una serie di istruzioni ***PUSH***) deve essere effettuato dal programma di servizio prima di iniziare ad eseguire quelle istruzioni che possono modificarne il contenuto, quindi è buona norma mettere all'inizio del programma di servizio le istruzioni di salvataggio del contenuto dei registri. Dopodiché si potranno eseguire le attività connesse alla richiesta di servizio.

Ripristino dello stato e ripresa dell'ultimo programma interrotto

Le ultime attività del programma di servizio debbono essere quelle di ripristino dello stato del processo interrotto salvato via software (tramite una serie di ***POP*** i cui operandi saranno in ordine inverso di quello delle corrispondenti ***PUSH***) e quindi si dovrà ristabilire il contenuto del ***rflags*** e del ***rip***. Questo viene fatto con l'esecuzione dell'istruzione RTI, che dovrà quindi essere l'ultima del programma di servizio.

Ciclo istruzione includente la gestione delle interruzioni

Per verificare la presenza di una interruzione il SCO del processore deve essere modificato per in modo tale che alla fine dell'esecuzione di ogni istruzione il SCO va a verificare se è presente o meno la richiesta di una interruzione. Naturalmente questo lo dovrà fare solo se il SCO è stato abilitato a verificarne la presenza, ciò viene fatto programmando opportunamente il flip/flop I. Quindi il ciclo istruzione visto precedentemente, includente solo cicli macchina, deve essere modificato come specificato in figura

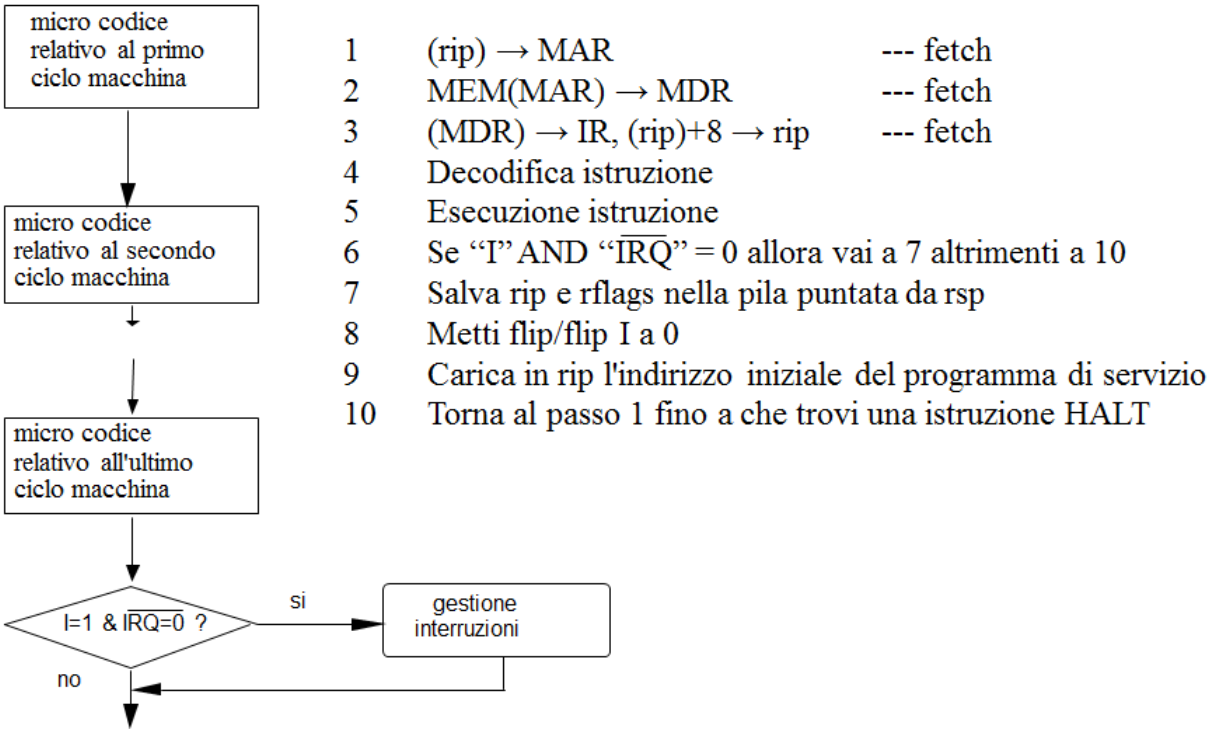


Figura 19: Ciclo istruzione

Da notare che il flip/flop I viene posto a 0 dallo SCO del processore nel microprogramma relativo all'attivazione del driver in quanto altrimenti appena inizia l'esecuzione del driver il SCO si trova ancora la richiesta delle interruzioni attiva. Per evitare che terminata l'esecuzione del driver ci sia ancora la richiesta di interruzione pendente è opportuno, all'interno del driver, di resettare/settare il flip/flop INT_REQ della periferica che ha effettuato l'interruzione, ciò verrà fatto tramite l'esecuzione di una istruzione di ***out.*** Sarà poi compito del programmatore del DRIVER prevedere o meno che il driver possa o meno essere interrotto da altre interruzioni, a tal fine il programmatore potrà usare le istruzioni ***SetI*** e ***ClrI*** per settare e resettare opportunamente tale flip/flop.
Comunque alla fine dell'esecuzione del driver, e cioè con l'esecuzione della RTI con cui termina il software del driver, verrà ripristinato nello rflags il vecchio contenuto memorizzato nello STACK e pertanto il flip/flop I ritornerà sicuramente abilitato.

Ricapitolando le istruzioni introdotte per gestire l'interruzione sono:

- ***SetI***
- ***ClrI***

SetI

Pone ad 1 il flip/flop IF del registro di stato per l'abilitazione della ricezione delle interruzioni.

ClearI
Pone a 0 il flip/flop IF del registro di stato per l'abilitazione della ricezione delle interruzioni

Naturalmente per l'interazione si potranno eseguire anche altre istruzioni tipo: ***In***, ***Out***, ***Ins*** e ***Outs***. E' inoltre da notare che è sempre opportuno avvertire la periferica con cui il processore desidera interagire, ovvero che ha terminato la sua programmazione. Ciò verrà fatto utilizzando un flip/flop di STATUS che avvertirà la periferica dell'interazione con il processore (vedere più avanti).

Politiche di gestione delle richieste di interruzione

Come detto precedentemente, al momento del riconoscimento della presenza di una richiesta di interruzione il SCO del processore azzera il flip/flop I per evitare che alla esecuzione della prima istruzione del driver attivato si ritrovi a dover rieseguire il microprogramma relativo al riconoscimento di una interruzione (si ricorda che tale microprogramma viene attivato dall'and tra il segnale IRQ (attivo basso) e l'uscita del flip/flop I. Il flip/flop I poi viene rimesso di nuovo rimesso ad uno dal microprogramma relativo all'esecuzione dell'istruzione RTI. In questo modo si garantisce che il driver relativo all'esecuzione del driver richiesto dall'interruzione non venga interrotto. In questo modo, però non si permetterebbe ad alcun altro dispositivo esterno di far eseguire il proprio driver durante l'esecuzione di un altro driver. Per questo motivo è possibile permettere al programmatore dei driver, tramite l'esecuzione di una istruzione SETI, di mettere ad uno il flip/flop di interruzione. In tal caso, durante l'esecuzione di un driver, all'atto dell'arrivo di un interrupt questo potrebbe essere gestito dal processore. Naturalmente si dovrebbero rieffettuare tutte le attività associate al riconoscimento di interruzione, quali il salvataggio nello stack di sistema del valore del RIP e dello status register. Comunque è da notare che un programmatore potrebbe desiderare di non far interrompere il proprio driver da un qualunque altro dispositivo, ma solo da alcuni.

Inoltre è da notare che durante l'esecuzione di una istruzione, dato che la condizione di presenza di una interruzione viene effettuata alla fine dell'esecuzione del ciclo istruzione stessa, si potrebbero verificare più richieste di interrupt. In tal caso è necessario decidere quale politica utilizzare per scegliere tra le richieste pending.

La soluzione più semplice che permette di risolvere il secondo problema, ma non il primo, è quello di assegnare una priorità statica ai dispositivi, utilizzando la proprietà che la velocità di trasmissione dei segnali elettronici è finita. In tal caso il segnale di riconoscimento della richiesta di interrupt (INTA) viene trasmesso in serie a tutti i dispositivi e quindi si ottiene una priorità in funzione della distanza fisica dal processore, ovvero della lunghezza della linea di comunicazione in cui viene tramesso il segnale INTA. Tale modalità viene chiamata DAISY-CHAIN e come schematizzato in figura 20: il segnale di INTA entra nell'interfaccia di un dispositivo e se questa non ha generato la richiesta di interruzione tale segnale viene propagato alla periferica fisicamente adiacente.

Come resettare flip/flop INT_REQ

Come detto, una volta attivato il driver è necessario, all'interno del driver stesso, annullare la richiesta dell'interruzione stessa. Poiché la richiesta di interruzione è memorizzata dalla periferica nel flip/flop INT_REQ il processore deve essere in grado di resettare tale flip/flop. Lo potrà fare con una istruzione di out. A tal fine sarà necessario aver scritto prima nel registro %dx l'indirizzo del flip/flop INT_REQ e nell'accumulatore il valore 1. All'atto dell'esecuzione dell'istruzione out verrà posto ad 1 il segnale di controllo I/O, e il segnale di controllo WR, mentre il bit meno significativo del Data Bus sarà pari ad 1 e sull'I/OAB ci sarà l'indirizzo di INT_REQ della periferica con cui il processore sta interagendo. Come si vede dalla Figura 20 il flip/flop di INT_REQ viene resettato dal segnale che viene generato dall'uscita della porta logica AND il cui ingresso è dato dal segnale di I/O, dal segnale WR, da DB0 e dal "select" generato a fronte della presenza dell'indirizzo della periferica presente sull'I/OAB. In questo modo si annulla la presenza dell'interruzione ormai riconosciuta dal processore.

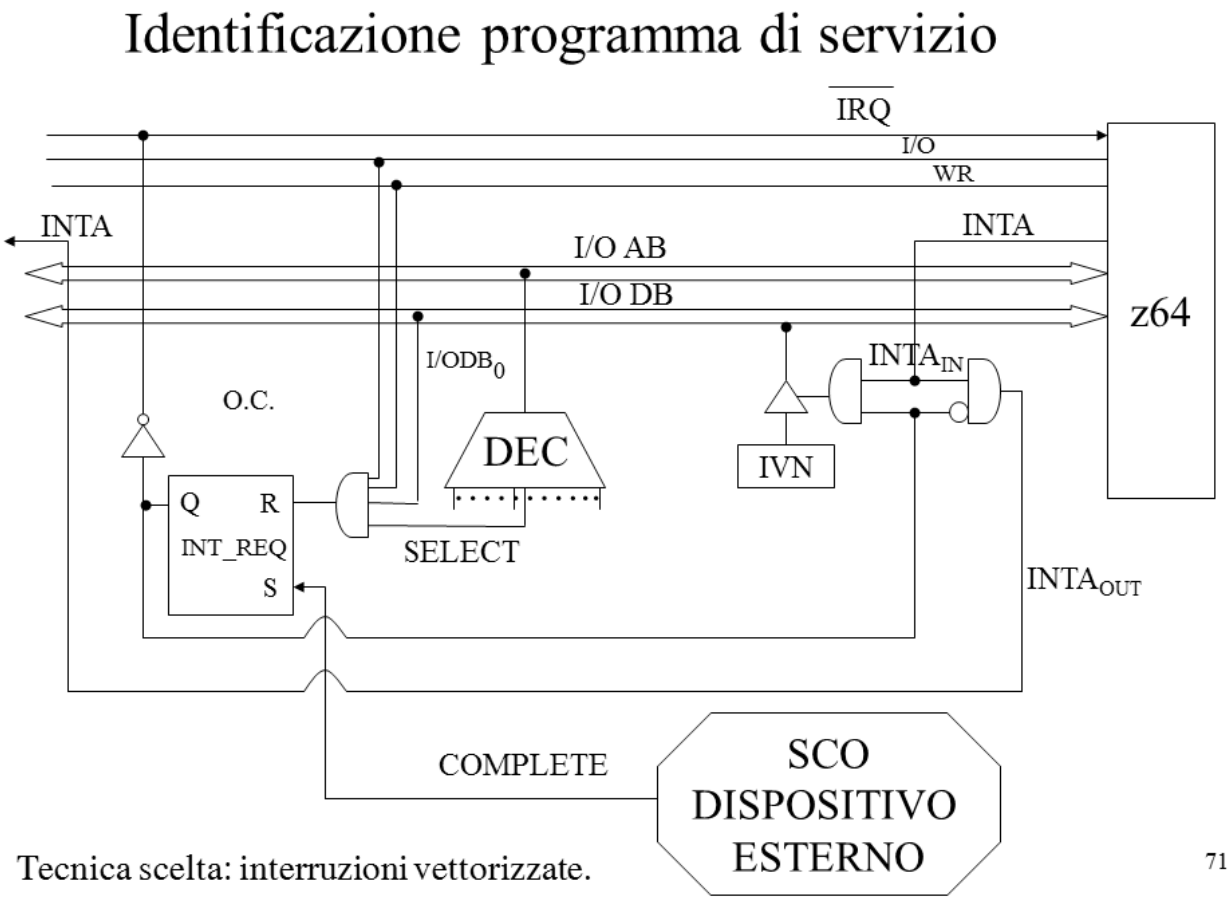


Figura 20: Propagazione del segnale di INTA (in modalità daisy-chain) e resettaggio flip(flop INT_REQ

A questo punto si può completare la progettazione delle interfacce delle periferiche che interagiscono con il processore tramite interrupt.
A tal fine si può fare riferimento alle interfacce delle periferiche che interagiscono in modalità bysy waiting e ricordiamo che c'era bisogno;

- di un flilp/flop che avverta la periferica che il processore è intenzionato a interagire con esse: nel caso di periferica di output che il processore ha scritto il dato nel registro di interfaccia e per le periferiche di input che il processore desira ricevere dati da tale registro di interfaccia
- di un registro di interfaccia per lo scambio dei dati, registro che può essere di 8, 16, 32 o 64 bit.

In figura 21 è inserito il flip/flop di STATUS, simile a quello utilizzato per l'interazione busy waiting, con l'avvertenza che questa volta serve solo al processore per avvertire la periferica ad attivarsi per produrre o consumare un dato.

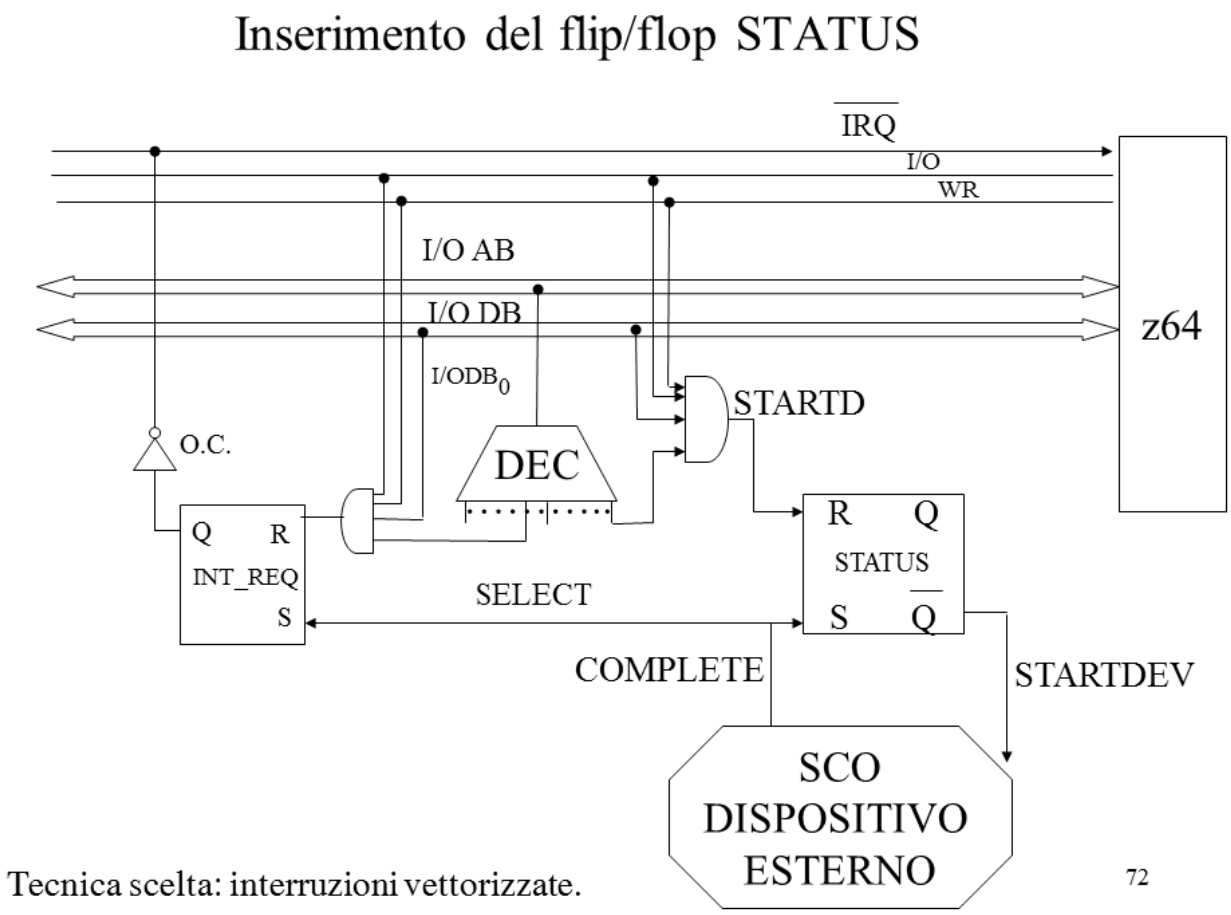


Figura 21: Inserimento del flip/flop di STATUS

Per completare le interfacce standard è necessario inserire solo i registri di interfaccia, come presentato in figura 22 per la periferica di input e in figura 23 per quella di output.

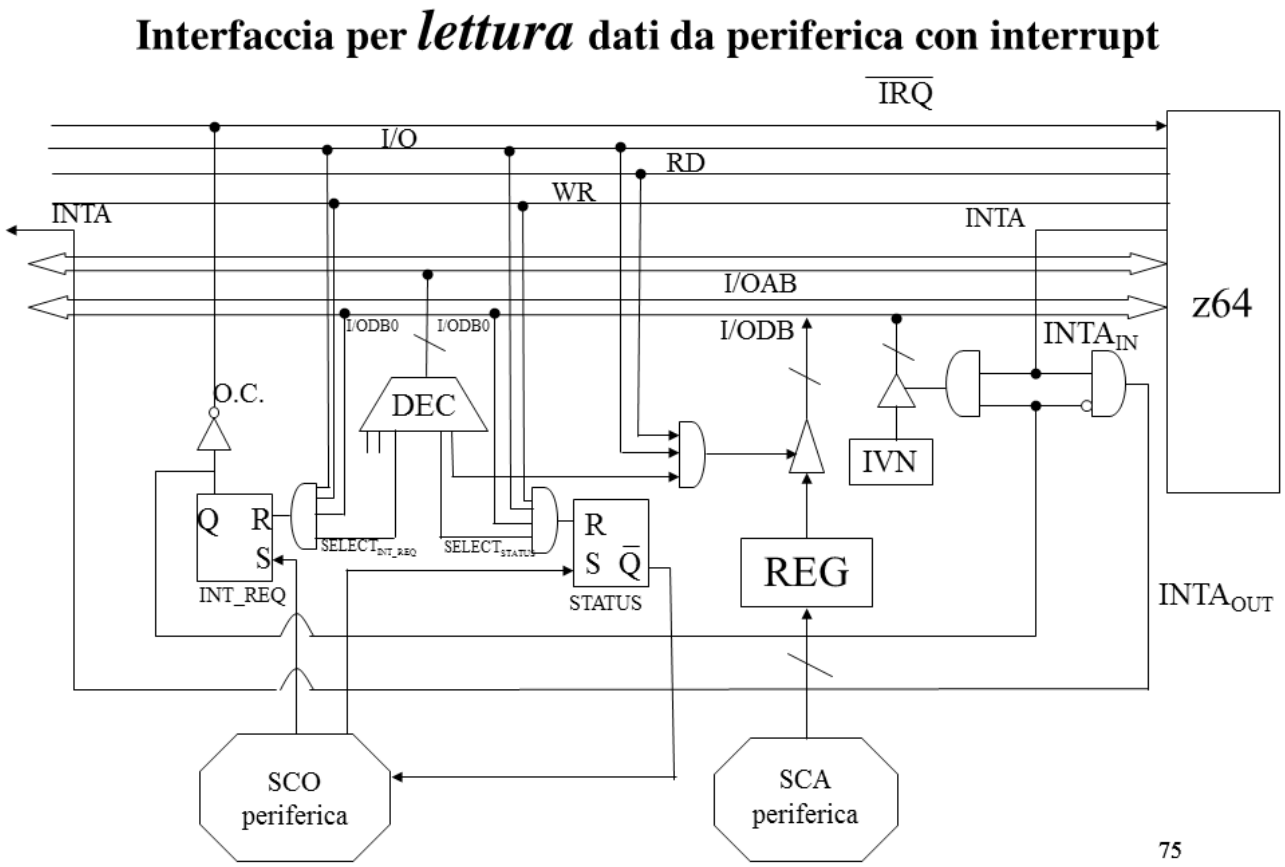


Figura 22: Interfaccia di periferica standard di input

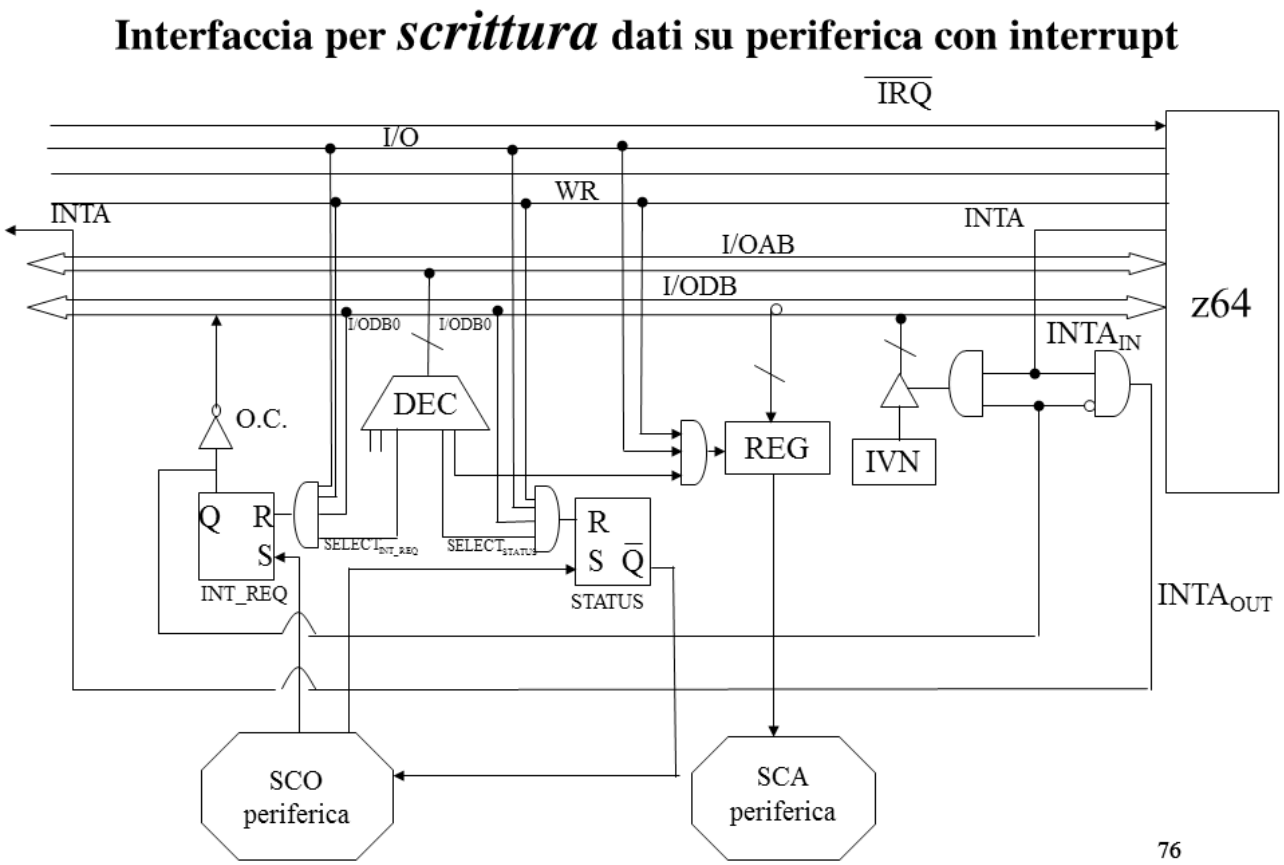


Figura 23: Interfaccia di periferica standard di output

Di seguito si presenta un driver per l'acquisizione di 100 dati (word) da una periferica, si ipotizza che i dati sono memorizzati a partire dalla locazione di memoria "dati" e che l'identificativo della periferica sia "AD1". Notare come a differenza dell'acquisizione in busy waiting il processore non sia mai in attesa attiva della produzione del dato della periferica, ma si attivi solo per acquisirne i dati. Per semplicità in questo esempio si è ipotizzato che il processore stia solo acquisendo dati dalla periferica AD1

DA RISCRIVERE SENZA START E CLEAR

```
.org 0x800          # Memorizza il programma dopo l'IDT
  .equ dati, 0xAAAA
  .equ AD, 0xAA
```

```
   .equ STATUS_AD, 0xAB
   .equ INT_REQ_AD, 0xAC
.text                  # Identifica l'inizio del programma
   xorl %rcx, %rcx         # Resetta il contatore dei dati acquisiti
   movq $dati, %rdi              # Imposta il 'registro destinazione' con la base del
vettore
   movw $STATUS_AD, %dx
   movb $1, %al
   outb %al, %dx           # Avvia la periferica AD
   sti                  # Abilita lo z64 per ricevere interruzioni
.loop:
   hlt                  # pone ciclicamente lo z64 in attesa di un interrupt
   jmp .loop

.driver 1                     # '1' è l'IDN della periferica
   movw $AD, %dx        # AD è l'indirizzo del registro di porta di I/O
   inw %dx, %ax            # acquisisce una word dalla periferica AD
   movw %ax, (%rdi, %rcx, 2)  # copia il dato acquisito nel vettore in memoria
       movw $INT_REQ_AD, %dx # INT_REQ_AD è l'indirizzo del F/F di
interruzione
   movb $0, %al            # Scrivere '0' su INT_REQ_AD cancella la causa di
interruzione…
   outb %al, %dx           # elimina la richiesta di interruzione
   addl $1, %rcx           # incrementa il contatore
   cmpq $100, %rcx         # verifica se sono state fatte 100 acquisizioni
   jz .exit                # in caso affermativo, esce direttamente dal driver
   movb $1, %al            # altrimenti riavvia periferica
    movw $STATUS_AD, %dx # STATUS_AD è l'indirizzo del F/F di stato della
periferica
   outb %al, %dx           # riavvia la periferica
  .exit:
  iret                  # ritorno da interruzione (all'istruzione dopo hlt)
.end
```

Gestione delle interruzioni

La soluzione daisy_chain non consente ad un programmatore di permettere l'interruzione dell'esecuzione solo ad un certo insieme di periferiche. Un modo per consentire ciò è ancora quello di assegnare staticamente delle priorità alle periferiche in base alla allocazione fisica, ma di utilizzare un dispositivo esterno al

processore, il ***controllore delle interruzioni***, con il compito di filtrare le richieste di interruzione e propagare il segnale di IACK solo alle periferiche che hanno una priorità maggiore o uguale a quella della periferica in cui è in esecuzione il driver. In figura 24 e in figura 25 sono rappresentate, rispettivamente, una possibile interconnessione di periferiche con il processore organizzate a L+1 livelli di priorità (le priorità sono inverse al valore del livello, i.e il livello zero è quello con priorità maggiore) e l'architettura interna del controllore delle interruzioni nel caso di 4 livelli di priorità. Come si vede dalla Figura 24 le periferiche di ogni livello di priorità sono a loro volta connesse in daisy_chain (naturalmente si potrebbe avere una sola periferica per livello) e le loro richieste di interruzione sono acquisite dal controllore, che come si può vedere nella figura successiva, invia il segnale di INTA, generato dallo SCO del processore, solo al livello di priorità più elevato (che equivale all'identificativo inferiore) delle periferiche che hanno effettuato la richiesta di interruzione, e tra quelle con lo stesso livello di priorità verrà servita prima quella fisicamente più vicina al controllore, dato che in ogni livello c'è una connessione daisy_chain per la propagazione dell'INTA. Nel caso in cui si usasse tale dispositivo il programmatore, per permettere le abilitazioni delle interruzioni durante l'esecuzione del suo driver dovrebbe, come nel caso precedente in cui non si usava il controllore, dovrebbe settare a programma il flip/flop I. In questo caso, permetterebbe l'esecuzione solo di altri driver con priorità uguale o superiore a quella corrente.

Entrambe le soluzioni indicate, essendo basate su priorità statiche possono causare problemi di fairness, per evitare ciò è necessario utilizzare soluzioni in cui la priorità delle richieste possa variare dinamicamente nel tempo, ovvero permettere al processore di mascherare richieste di interruzioni di periferiche che impediscano ad altre periferiche di far eseguire i propri driver. Lo studio di politiche di scheduling di richieste di attivazione di driver che evitino starvation è proprio di corsi tipo Sistemi Operativi, comunque brevemente si vuole far vedere una soluzione in cui il processore possa filtrare dinamicamente delle richieste di interruzione. A tal fine si fa riferimento alla Figura 24 dove le richieste di interruzioni che arrivano al controllore sono filtrate da delle porte NOR e da dei flip/flop (MFFi). L'uscita di questi flip/flop va in NOR con la richiesta delle interruzioni (attiva bassa), quindi se programmati ad 1 generano in uscita del NOR un segnale alto e quindi si filtra l'eventuale richiesta di interruzione. Naturalmente questi flip/flop devono essere visibili al processore per essere programmati, in tal caso ogni singolo flip/flop potrebbe essere programmato tramite l'esecuzione di una out.

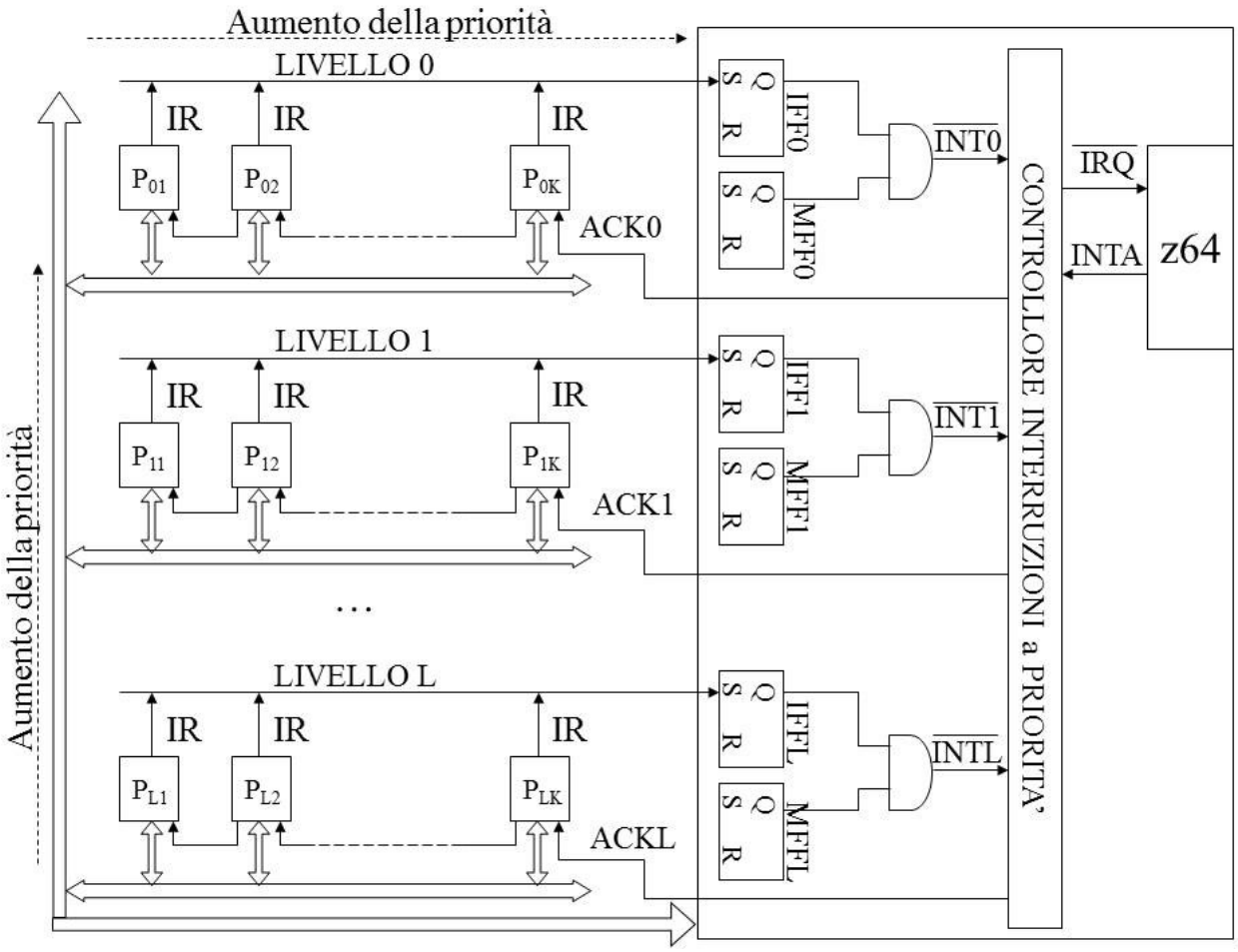


Figura 24: Gestione interruzioni con priorità

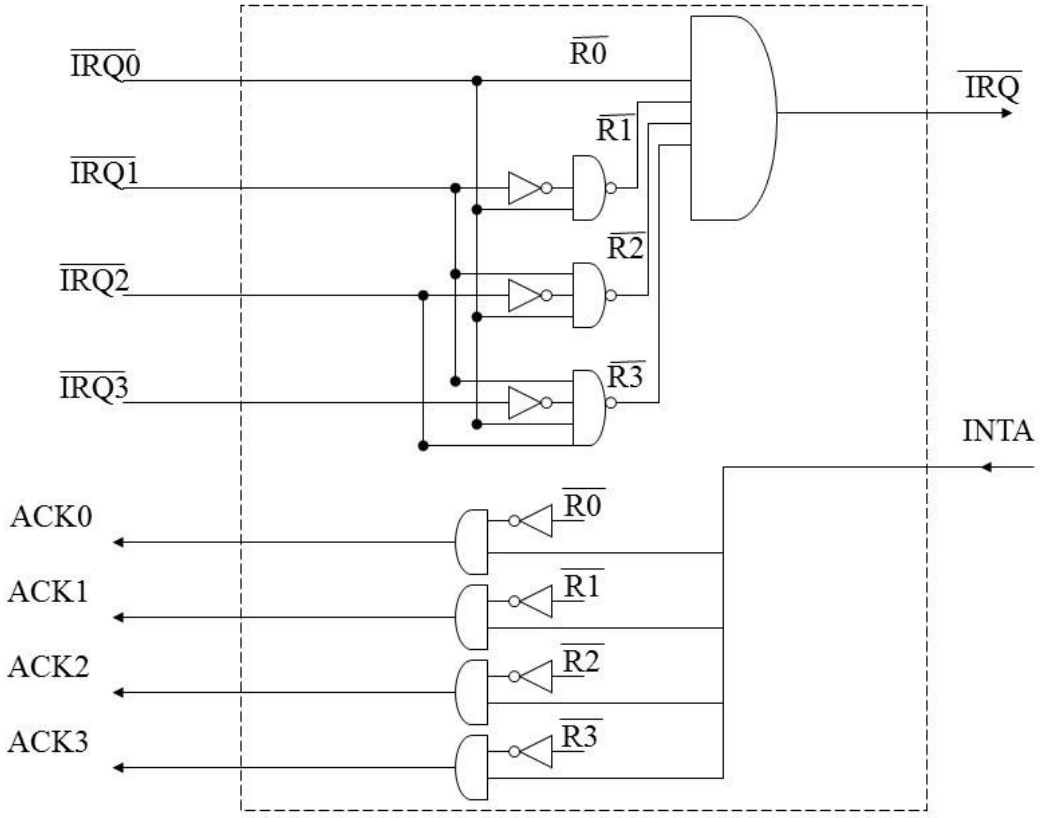


Figura 25: Architettura del controllore delle priorità

Oltre che dalle periferiche la richiesta di interrompere le attività in esecuzione per eseguirne delle altre può venire dall'interno del processore, come ad esempio per l'esecuzione di una istruzione che causa un overflow o per gestire l'esecuzione di istruzioni, quali le system call, che esplicitamente desiderano interrompere il normale funzionamento per attivare un driver particolare. La gestione di tali interruzioni avviene nella stessa modalità con cui vengono gestite le interruzioni esterne. In questo Capitolo siamo interessati ad analizzare come il processore interagisce con il mondo esterno e quindi si rimanda ad altro capitolo la discussione sulla natura e sull'utilità di questi ultimi tipi di interruzione. Notare che nella letteratura scientifica si chiamano ***interruzione*** le richieste generate dall'esterno ed ***eccezioni*** o ***trap*** quelle generate internamente nel processore.

Operazioni di I/O gestite da canale

Il trasferimento di un blocco di dati tra due periferiche o tra due moduli di memoria o tra una periferica ed un modulo di memoria può essere effettuato sotto il controllo del processore, tramite l'esecuzione di un programma o di una singola istruzione (***insX*** o ***outX***). In questo caso il trasferimento di ogni singolo dato (byte, parola, doppia parola o quadrupla parola) può essere fatto utilizzando un registro interno del processore per memorizzare temporaneamente il dato che deve essere trasferito. Ciò comporta che il trasferimento di ogni singolo dato necessita che il processore prelevi il dato dalla sorgente, scrivi il dato nel dispositivo destinatario, incrementi o decrementi il contenuto di un registro (per identificare la locazione di memoria dove memorizzare il dato), decrementi un contatore (necessario per tenere in conto del numero di trasferimenti effettuati) e infine verifichi se si è completato il trasferimento del blocco di dati. Considerando che l'esecuzione di ogni singola istruzione necessita di un ciclo macchina per il fetch e di eventuali altri cicli macchina per leggere/scrivere dati e/o operandi, si comprende come questa tecnica sia relativamente lenta, sia nel caso che si usi la tecnica busy waiting che usando le interruzioni.

Per incrementare la velocità di trasferimento dei dati si può pensare di utilizzare dei dispositivi progettati ad hoc per il trasferimento dei dati. Ciò è relativamente facile in quanto i programmi assembly per i trasferimenti dati, visti precedentemente utilizzanti la modalità di busy waitnig o le interruzioni, possono essere scritti in modo parametrico (dove i parametri sono almeno: identificazione sorgente, identificazione destinazione, lunghezza del blocco da trasferire – i casi con più parametri verranno analizzati successivamente), Quindi è sufficiente progettare un dispositivo in grado di utilizzare tali parametri per il trasferimento dei dati. Tale dispositivo è denominato DMAC da Direct Memory Access Controller, per la sua capacità di accedere direttamente alla memoria di lavoro. Questo dispositivo per poter effettuare il trasferimento dei dati deve emulare il comportamento della CPU e lo farà eseguendo un microprogramma, a livello quindi firmware, piuttosto che eseguendo un programma assembly, come visto per le precedenti modalità di trasferimento dati.

Inoltre per emulare il comportamento della CPU il DMAC necessariamente deve generare gli stessi segnali di controllo e di indirizzamento della CPU, in tal modo i dispositivi di I/O e/o i moduli di memoria si comporteranno come se il trasferimento avvenisse sotto il controllo della CPU. Per questo motivo il DMAC deve vedere sia la memoria che il dispositivo, di ingresso o di uscita, interessato al trasferimento. Inoltre il processore deve poter programmare il DMAC trasferendogli i parametri necessari per il trasferimento dei dati. Pertanto il DMAC è visto dal processore come

una periferica. Il DMAC potrebbe nascondere le periferiche al processore. Come schematizzato in figura 26 oppure potrebbe utilizzare lo stesso bus di I/O del processore per interagire con le periferiche come schematizzato in Figura 27. Di seguito si ipotizza che è previsto che il DMAC si interponga tra il processore e le periferiche (figura 26).

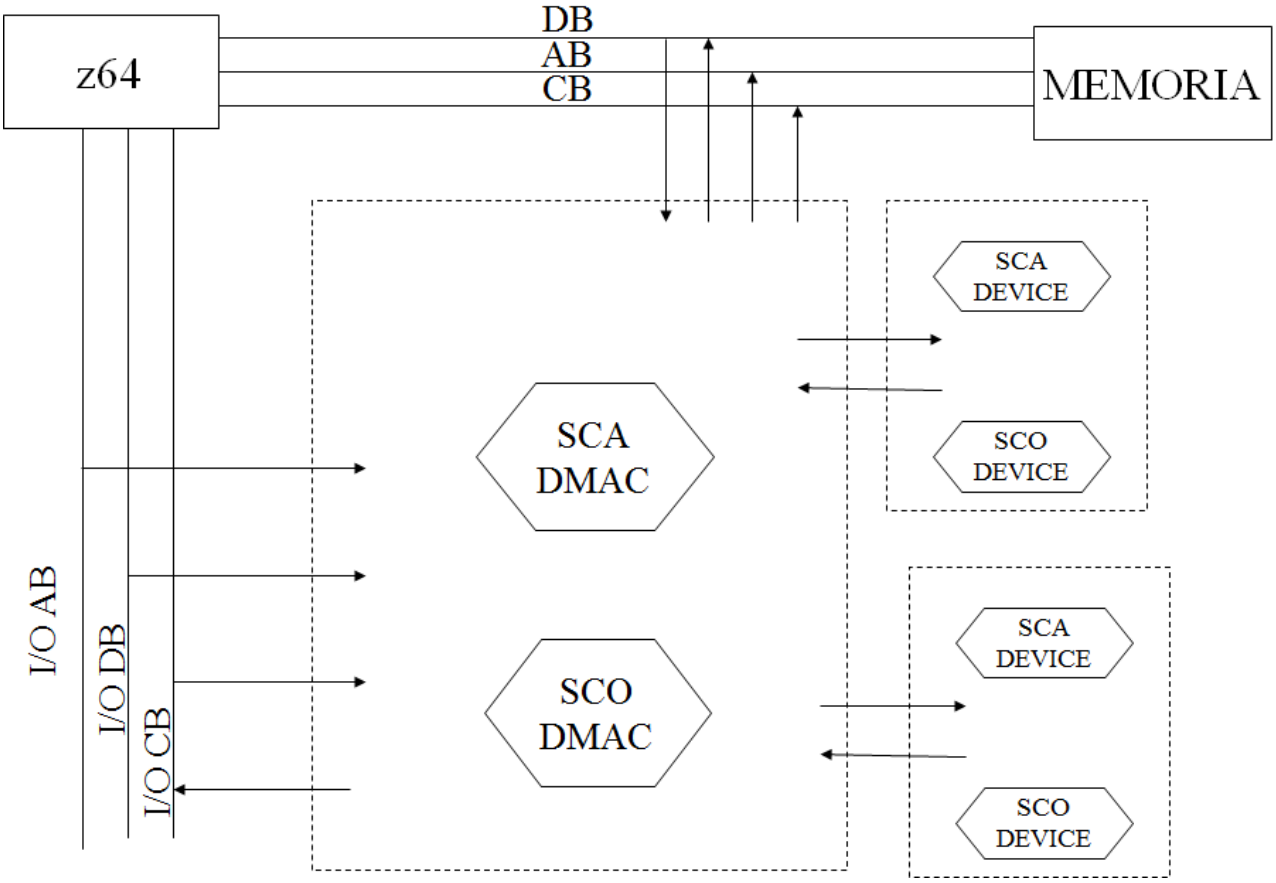


Figura 26: Architettura in cui il DMAC si interpone tra processore e periferiche

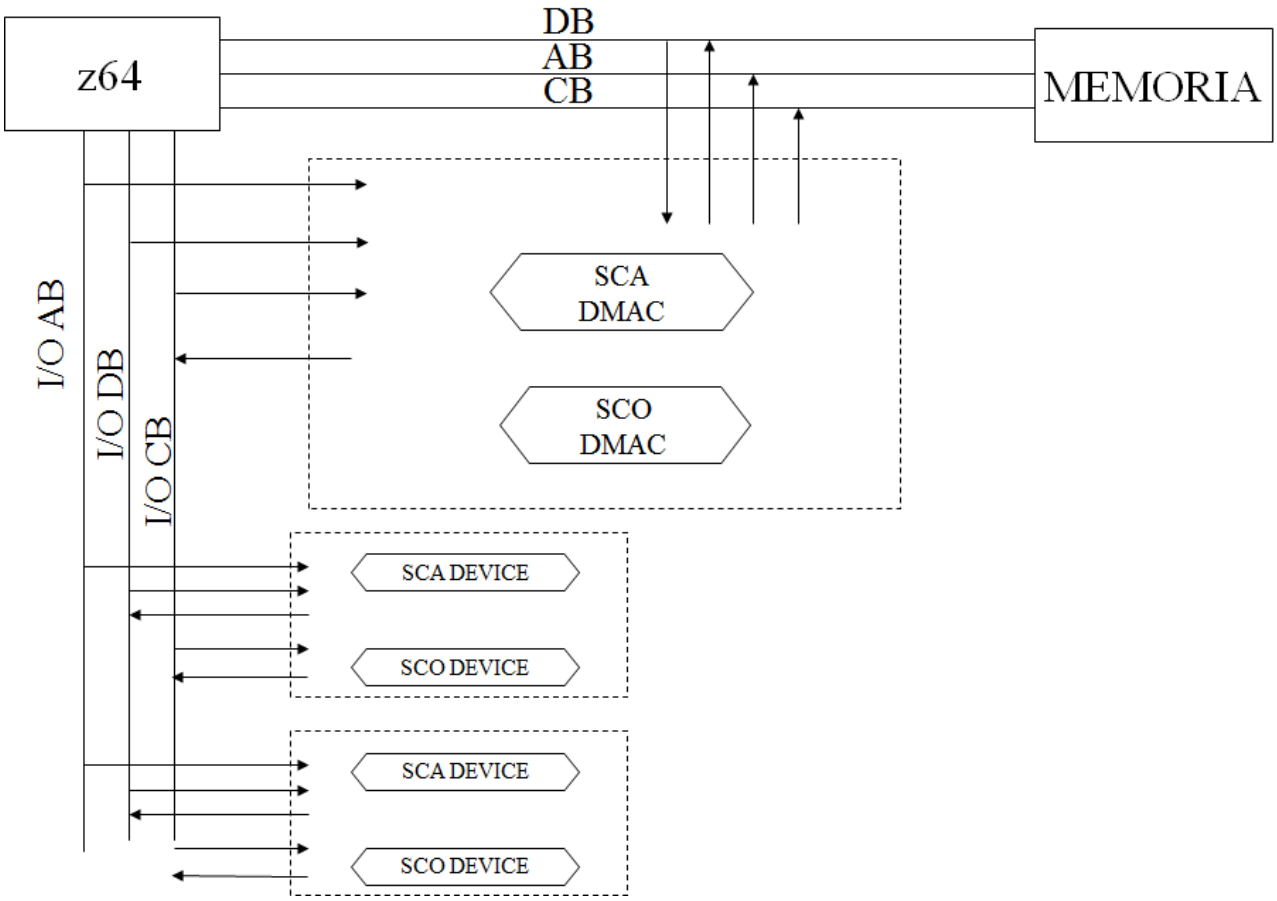


Figura 27: Architettura in cui il DMAC utilizza lo stesso bus del processore per interagire con le periferiche

Naturalmente ci potrebbe essere una soluzione intermedia in cui il DMAC si interpone per un sottoinsieme delle periferiche, mentre le altre sono visibili anche dal processore, come schematizzato in figura 28.

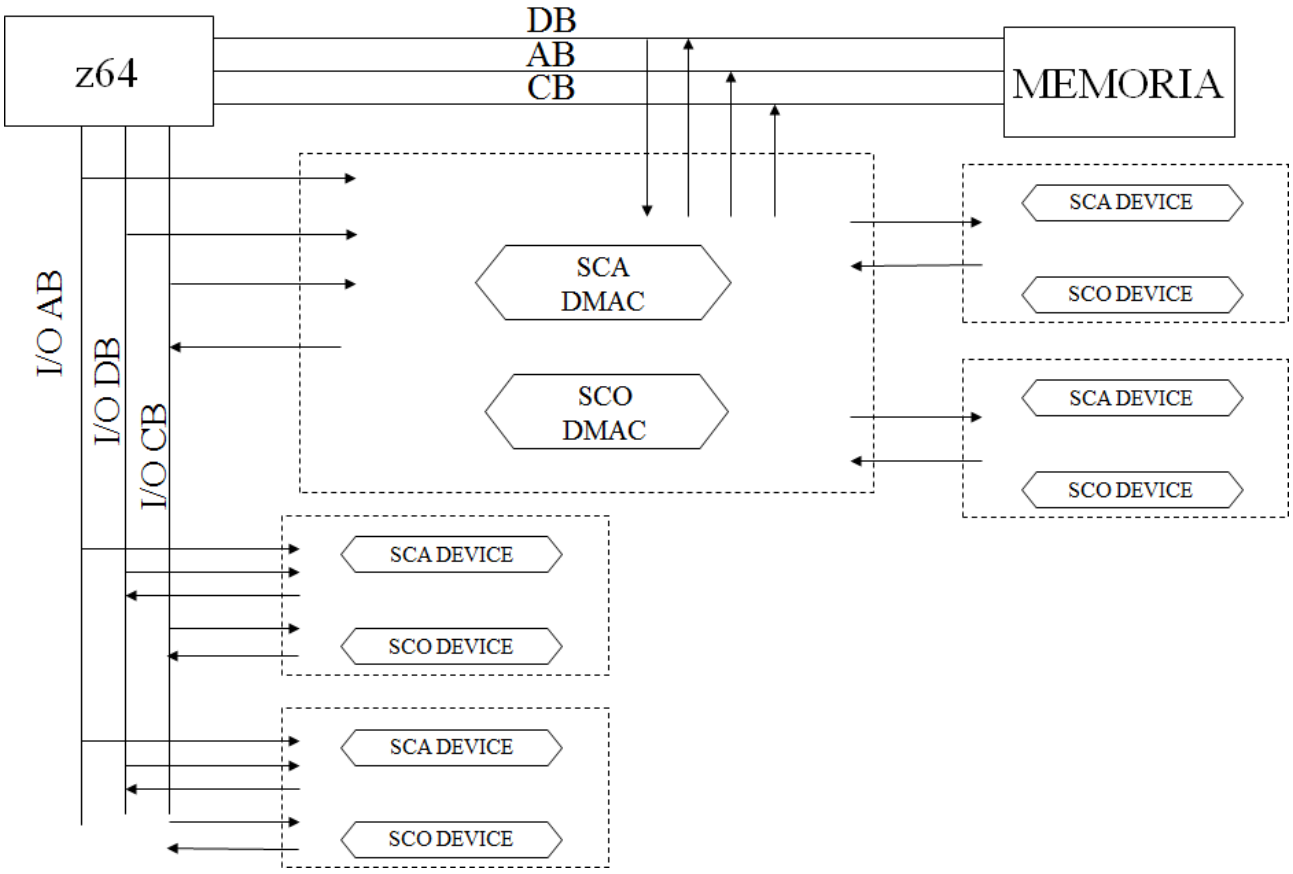


Figura 28: Soluzione ibrida

In tutte e tre le soluzioni architetturali vi è una concorrenza nell'utilizzazione dei bus tra la CPU e il DMAC. Affinché il DMAC possa utilizzare correttamente queste risorse è necessario che la CPU non interferisca elettricamente su di esse, e viceversa. Poiché la richiesta di utilizzare i bus (e quindi che essi vengano rilasciati dalla CPU) può sorgere indipendentemente dalle attività del processore (ovvero in modo asincrono) è necessario utilizzare un segnale di controllo verso il processore (*Memory Bus Request*, $\overline{\text{MBR}}$) per avvertire il suo SCO della presenza di questo evento. Una volta riconosciuta tale richiesta il SCO del processore avverte il SCO del DMAC dell'avvenuto riconoscimento della richiesta (*Memory Bus Grent*, MBG) e contemporaneamente mette in alta impedenza le uscite de processore verso i suddetti bus ad eccezione dei segnali di controllo in ingresso ed in uscita per la gestione della richiesta di rilascio del bus: $\overline{\text{MBR}}$ e MBG).

Successivamente, dopo aver introdotto i metodi di cooperazione tra SCO del processore e SCO del DMAC per l'utilizzo dei bus, verrà descritto come il SCO dello z64 gestisce le richieste di interruzione e le richieste di utilizzare i bus.

Gestione delle richieste di accesso ai bus

Abbiamo visto precedentemente che la richiesta di accesso ai bus e, quindi, del loro rilascio da parte della CPU, può sorgere in modo asincrono rispetto alle attività del processore e che questa richiesta viene trasmessa alla CPU tramite il segnale di controllo $\overline{MBR}$ (Memory Bus Request), attivo basso per poter essere utilizzato in wired_or. A fronte di tale richiesta il processore deve avvertire il richiedente che la sua richiesta è stata accettata, indicheremo con MBG (Memory Bus Grant) il relativo segnale di controllo. A differenza della gestione dell'interruzione, in caso di richiesta di rilascio del bus il processore non deve eseguir alcun programma di servizio, ma deve solo mettere in alta impedenza i propri ingressi e le proprie uscite verso i bus esterni ad eccezione di $\overline{\text{MBR}}$ e MBG. La verifica della presenza della richiesta di rilascio del bus quindi non deve necessariamente essere effettuata alla fine del *ciclo istruzione* ma è possibile verificarla alla fine di ogni singolo *ciclo macchina*. Una volta riconosciuta tale richiesta il SCO del PD32 mette ad alta impedenza le uscite del PD32 verso i suddetti bus. Il richiedente viene avvertito del rilascio dei bus tramite il segnale di controllo specifico MBG. Il richiedente avvertirà il SCO del rilascio dei bus condivisi annullando la richiesta (ovvero mettendo alto il segnale $\overline{MBR}$). La scelta di verificare la presenza o meno di una richiesta di rilascio del bus ad ogni ciclo macchina nasce anche dal fatto che il DMAC è un dispositivo normalmente asservito alle necessità del processore, il quale lo programma per trasferire informazioni/dati in modo molto più veloce di quanto lo possa fare lui direttamente, quindi prima il DMAC finisce di trasferire i dati meglio è per processo che utilizza i dati.

E' da notare che se il DMAC, una volta ottenuto il controllo dei bus li utilizza fino a completo trasferimento dei dati il processore sarà inutilizzato per tutto il tempo di trasferimento dei dati e se il trasferimento impegna una periferica lenta questo tempo di trasferimento può divenire significativo per il processore, seppure stia utilizzando un dispositivo più veloce di lui nella gestione del trasferimento. Tale modalità di trasferimento, detta a BURST, è comunque utile nel caso in cui il processore per poter continuare nelle sue attività ha assoluta necessità dei dati che si devono trasferire, come vedremo successivamente nel caso di cache miss quando si parlerà delle memorie cache. Una soluzione alternativa è quella in cui il DMAC richiede e rilascia il bus per il trasferimento di un solo dato alla volta. In tal modo ci potrebbe essere un'alternanza tra il processore e il DMAC nell'uso del bus e la durata di assenza del controllo del processore è solo quella relativa al tempo di trasferimento di un dato quando il DMAC detiene il controllo dei bus, ma poiché è buona norma per il DMAC richiedere il bus solo quando il dato è pronto per il suo trasferimento (quindi indipendentemente dalla velocità della periferica a produrre/consumare il

dato), il tempo di mancanza di controllo del bus da parte del processore sarebbe solo pari al tempo di trasferimento del dato. Con questa modalità di richiesta dei bus il processore potrebbe ottenere il bus per almeno un ciclo macchina e quindi l'esecuzione del programma potrebbe comunque avanzare anche durante le attività di trasferimento del DMAC, anzi nel caso in cui le attività interne del processore fossero dello stesso ordine del tempo di trasferimento di un dato sotto il controllo del DMAC il programma in esecuzione sul processore avrebbe alcun ritardo per essere eseguito. Tale modalità di richiesta del bus viene denominata BUS STEALING.

Di seguito vengono presentati il diagramma di flusso del microprogramma del processore per l'esecuzione delle istruzioni nel caso in cui è possibile prevedere richieste di interruzione e di rilascio del bus. Successivamente dopo aver presentato il SCA del DMAC verrà presentato il diagramma di flusso ad alto livello dello SCO del DMAC nel caso di richiesta del bus in modalità BURST e in modalità BUS-STEALING

Diagramma di flusso del microprogramma della CPU in cui è prevista la possibilità di gestire le interruzioni e le richieste di rilascio del bus

In figura 29 viene ripresentato il diagramma di flusso del microprogramma relativo ad una generica istruzione macchina in cui è prevista anche la gestione delle richieste di interruzione e della richiesta del rilascio dei bus. Da notare che mentre la verifica della richiesta delle interruzioni viene fatta a fine ciclo istruzione, quella del rilascio del bus viene effettuata alla fine di ogni ciclo macchina.

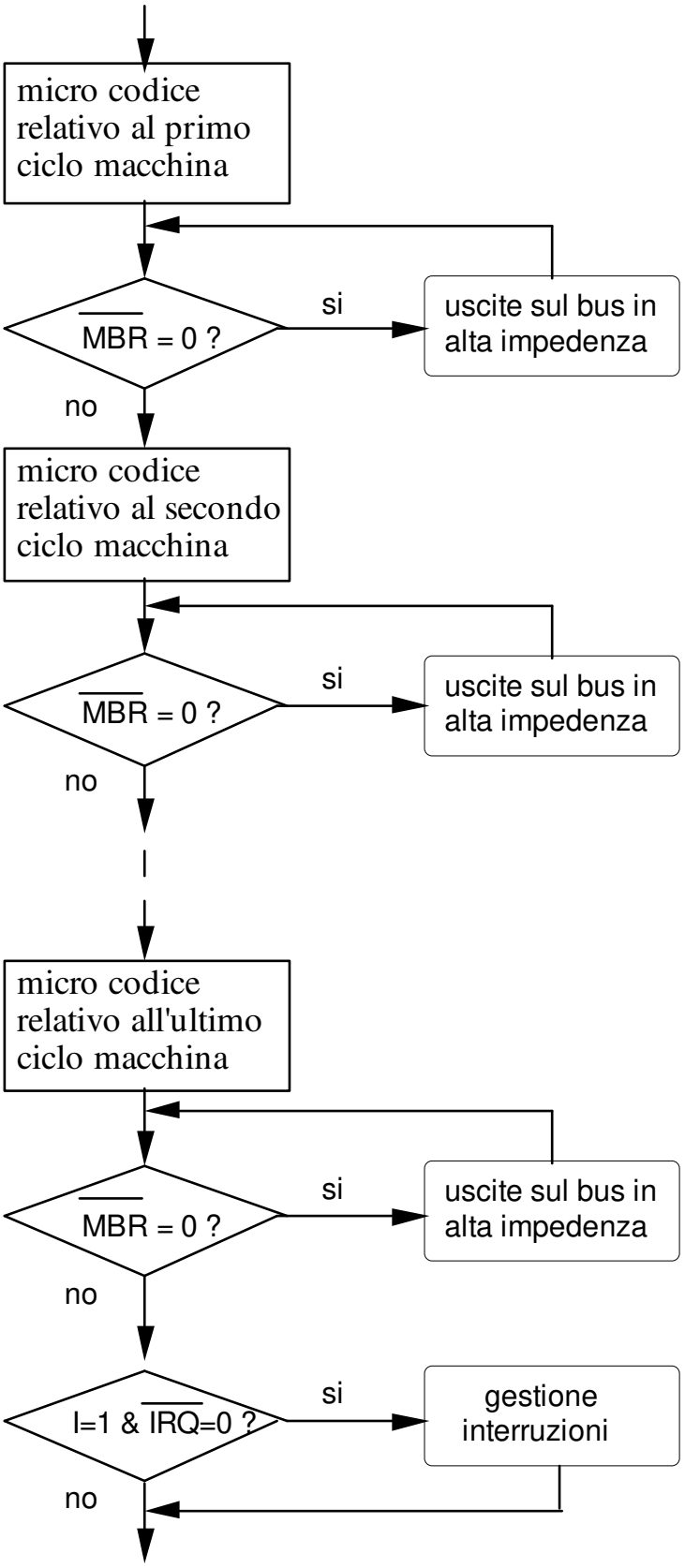


Figura 29: Modifica del ciclo istruzione per gestire la richiesta del rilascio del bus

Architettura del SCA del DMAC

Come detto precedentemente il DMAC è un processore dedicato al trasferimento dati tra memorie o tra una memoria ed una o più periferiche, come primo esempio, solo per semplicità didattica, si presenterà il SCA di un DMAC monocanale, bidirezionale, in cui la periferica per leggere o scrivere i dati di un unico formato (p.e. quadword), presenta una semplice interfaccia costituita da un registro bidirezionale, da un flip/flop di handshaking e da uno SCO in grado di gestire un protocollo asincrono. Per semplicità di esposizione non si presenterà il dettaglio dello SCA e dello SCO della periferica facendo vedere solo l'interfaccia connessa come in figura 30 al DMAC. Inoltre si prevede che il DMAC possa acquisire il bus in modalità BURST o in quella BUS-STEALING. Una volta acquisiti gli elementi di base si presenterà un DMAC quadricanale e poi, una volta introdotte le memorie cache, un dispositivo in grado di traferire dati in modalità DMA tra una memoria di lavoro ed una cache.

Le funzioni del DMAC sono solo quelle di trasferire dati in modo efficiente al posto del processore, pertanto tutte le informazioni necessarie al trasferimento, quali:

- direzione del trasferimento (i.e. dalla memoria verso la periferica o dalla periferica verso la memoria),
- modalità di acquisizione del bus (BURST o BUS-STEALING),
- indirizzo iniziale di memoria da cui leggere/scrivere i dati e
- quantità di quadword da trasferire,

sono trasferite dal processore verso il SCA del DMAC. Il processore, fare riferimento alla figura 26, vede il DMAC come una periferica, pertanto potrà eseguire una serie di istruzioni di output per trasferire tali informazioni, che saranno utilizzate dal SCO del DMAC per trasferire i dati secondo le direttive del processore. Una volta programmato il DMAC il processore potrà avvertire il SCO del DMAC dell'avvenuta programmazione tramite una semplice istruzione di START. Tale istruzione, settando opportunamente il flip/flop di stato del DMAC, consentirà al SCO del DMAC di iniziare il trasferimento dei dati. Naturalmente il SCO del DMAC per poter interagire con la memoria dovrà generare i segnali di controllo tipici per la lettura/scrittura dei dati dalla memoria e i segnali di controllo verso la periferica per il trasferimento asincrono dei dati. Una volta completato il trasferimento il DMAC dovrà avvertire il processore dell'avvenuto trasferimento, ciò può essere effettuato generando una interruzione e presentando un IVN relativo al driver che avvertirà il processore dell'avvenuto trasferimento. In realtà sarebbe più corretto parlare di S.O. invece di processore, ma dato il carattere introduttivo del

libro che può essere utilizzato senza alcuna conoscenza del funzionamento dei S.O., si utilizza tale termine anche se non completamente appropriato.

Nel DMAC appena presentato si è fatto riferimento ad un trasferimento di dati di tipo quadword. Se, invece, si volesse indicare il formato del dato (Byte, Word, Longword, Quadword) sarebbe sufficiente introdurre un altro registro di interfaccia nel DMAC in cui durante la fase di programmazione il programma che vuole effettuare il trasferimento possa indicare il formato dei dati (a tal fine sarebbe sufficiente utilizzare un registro a due bit, dato che con due bit si possono codificare 4 possibili configurazioni diverse). Registro che poi deve essere utilizzato dal SCO del DMAC per incrementare, decrementare adeguatamente il WC e il CAR ed inoltre trasferire dati da o verso la memoria nel formato richiesto (naturalmente questo avrà delle ripercussioni sulle modalità di generazione dei segnali di controllo Mbi verso i moduli della memoria).

Nelle figure successive viene indicato come è organizzato il SCA di un DMAC programmabile dallo z64, si presentano più figure per motivi di spazio, l'architettura complessiva è data dall'unione degli schemi relativi.

In particolar modo nella figura 30 è presentata la parte dell'interfaccia che include il CAR, il WC, il flip/flop necessario per la richiesta del rilascio del bus (MBR) e il flip/flop di stato, necessario per permettere al processore di avvertire il SCO del DMAC che sono stati programmati i registri di interfaccia. Inoltre sono schematizzate le connessioni del DMAC da e verso la memoria di lavoro e verso i dispositivi di ingresso ed uscita.

Invece nella figura 31 è presentata la parte dell'interfaccia che include i flip/flop che sono utilizzati dal processore per programmare se il DMAC deve lavorare in modalità burst o bus_stealing, e le deve prelevare/spedire dati dalla/alla periferica.

Infine nella figura 32 è presentata la parte relativa alla gestione delle interruzioni.

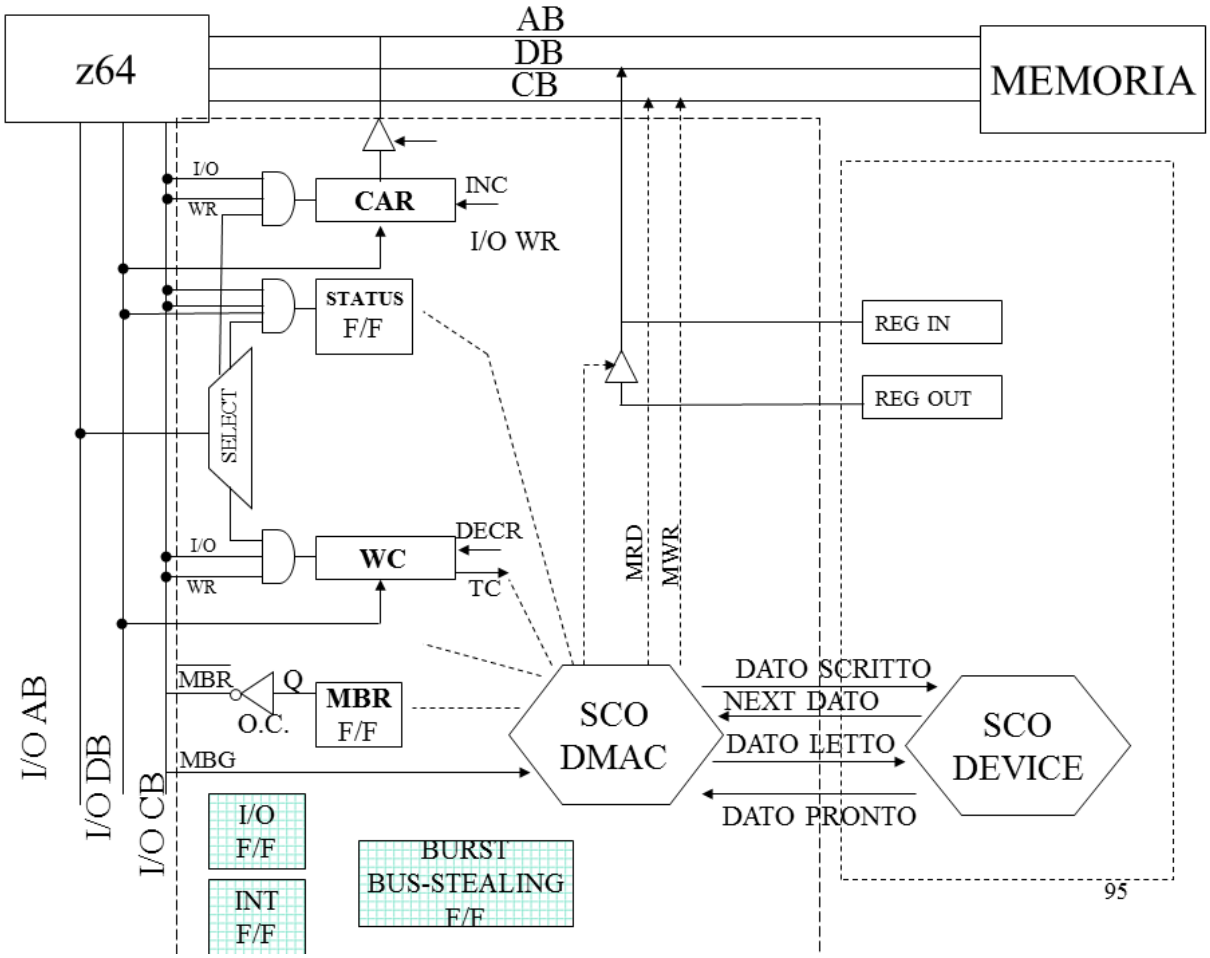


Figura 30: Architettura SCA del DMAC (I parte)

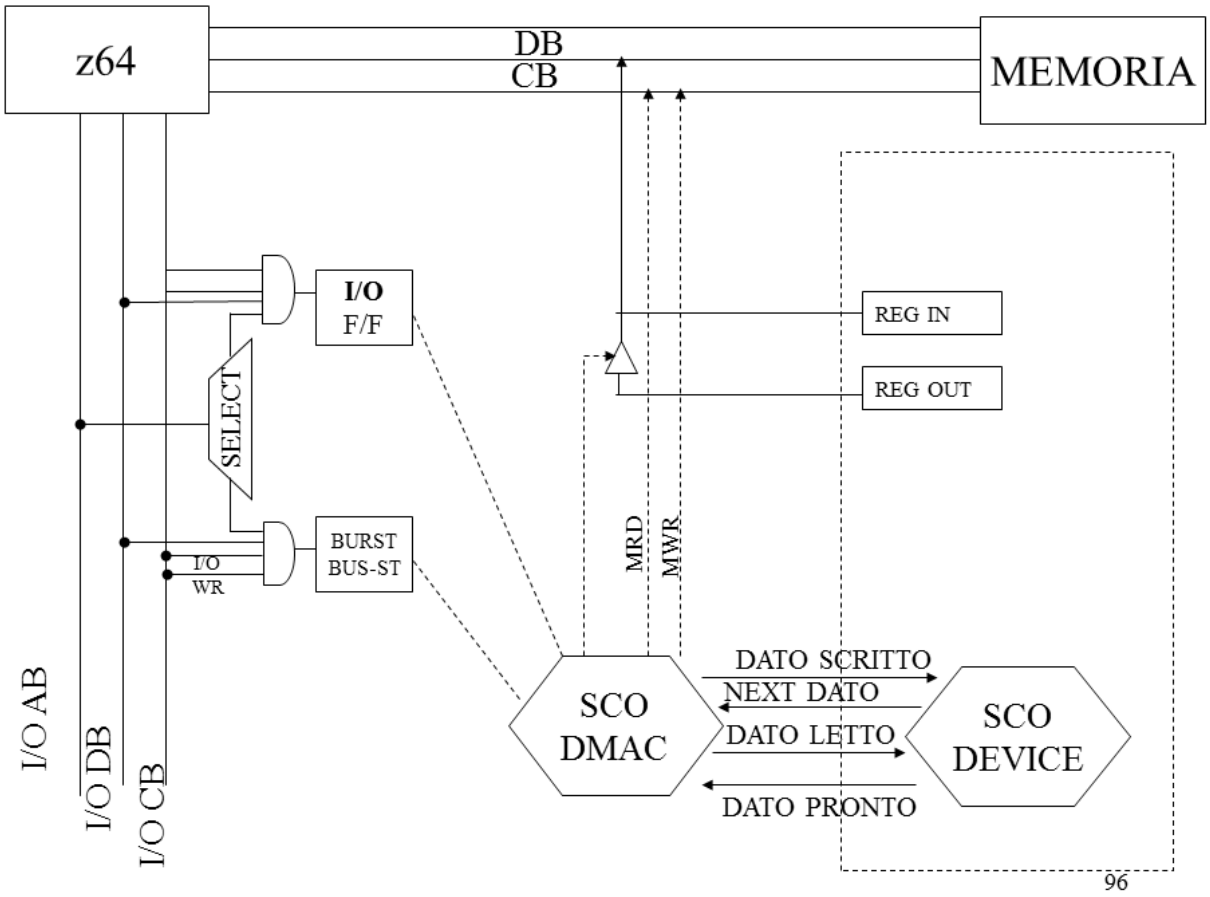


Figura 31: Architettura SCA del DMAC (II parte)

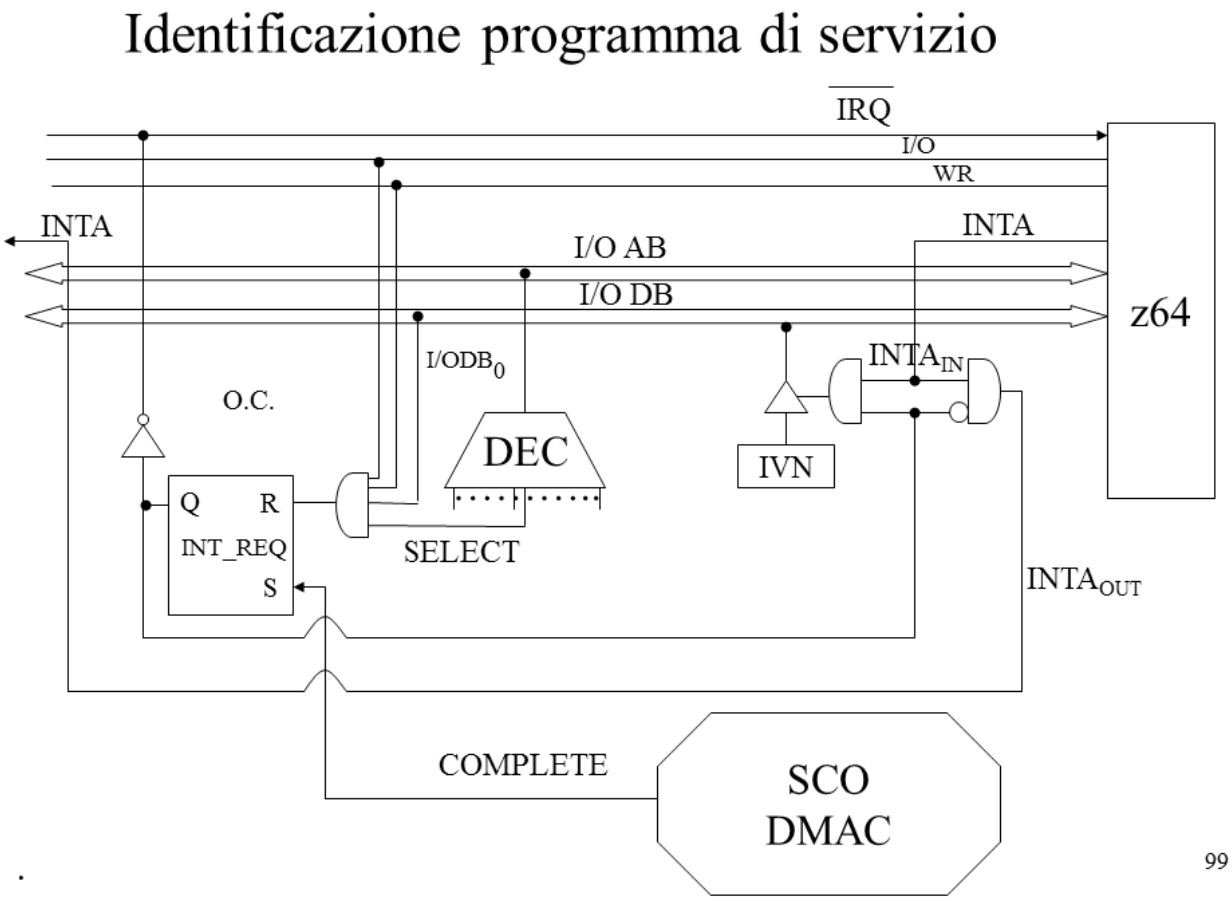


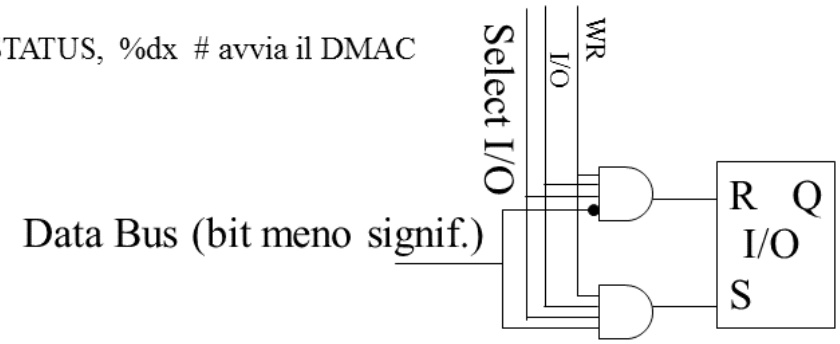
Figura 32: Architettura SCA del DMAC (III parte)

Come si può vedere dalle figure 30, 31 e 32 il DMAC è visto dal processore come una periferica standard con l'unica eccezione che il DMAC può richiedere al processore di mandare in alta impedenza le proprie uscite. Processore che naturalmente avrà tutti i vantaggi di far lavorare il DMAC per permettergli di trasferire quei dati che altrimenti il processore stesso avrebbe dovuto trasferire.

Per completezza in figura 33 si riporta un frammento di programma assembly per la programmazione di un DMAC che deve trasferire 100 dati dalla periferica di ingresso verso la memoria a partire dalla locazione di memoria 800aaa in modalità burst.

Esempio inizializzazione DMAC

```
movw $WC, %dx              # inizializza  WC a 100
movl $100, %eax
outl %eax, %dx
movw $CAR, %dx             # inizializza  CAR a 0x800aaa
movl $0x800aaa, %eax
outl %eax, %dx
movw $DMACIO, %dx          # Inizializza  il DMAC per la scrittura
movl $1, %eax
outl %eax, %dx
movw $DMACB-ST, %dx        # Inizializza  il DMAC per lavorare in burst
movl $0, %eax
outl %eax, %dx
movw $DMAC_STATUS, %dx  # avvia il DMAC
movl $1, %eax
outl %eax, %dx
```



98

Figura 34: Esempio di inizializzazione di un DMAC

Da notare inoltre che il DMAC così come progettato di fatto emula il comportamento del processore nell'eseguire le istruzioni ***insX*** e ***outX***, vantaggio della sua funzionalità è che mentre il DMAC effettua il trasferimento il processore potrebbe eseguire altri programmi ed eventualmente rientrare ad eseguire il programma che ha effettuato la richiesta di interazione con la periferica solo dopo che il trasferimento è stato completato. Inoltre nell'ipotesi che il programmatore volesse che il trasferimento dati sia effettuato il più velocemente possibile e non permettere al programma di eseguire altre istruzioni fino al completo trasferimento dei dati sarebbe opportuno programmare il DMAC in modalità BURST.

Al momento si è ipotizzato che il processore abbia due bus distinti e che lo SCO del processore fosse in grado di interagire con le periferiche. Questa modalità, seppure ancora presente nei microprocessori, con l'evoluzione delle tecnologie è stata abbandonata e le funzionalità di interazione con le periferiche sono state delegate a

dei processori dedicati alla comunicazione con i dispositivi di ingresso/uscita. Per processore I/O dedicato s'intende un sistema digitale complesso provvisto di un SCO e di un SCA in grado di comunicare sia con il processore principale (quello che interpreta le istruzioni dei programmi) che con le periferiche ed eventualmente con la memoria, come il DMAC appena progettato. Nei capitoli successivi faremo vedere come è possibile delegare completamente le interazioni con i dispositivi di ingresso/uscita ad uno o più processori di I/O dedicati.

Attività dello SCO del DMAC e del processore per trasferire dati in modalità Bus Stealing o Burst

Il DMAC per poter trasferire dati da e verso le periferiche deve poter accedere alla memoria di lavoro, per poter utilizzarla deve generare gli stessi segnali di controllo del processore, inoltre deve generare gli indirizzi delle locazioni i di memoria interessate. Quindi il processore, in caso di utilizzo dei bus da parte del DMAC, deve mettere in alta impedenza le proprie uscite per non creare corto circuiti e viceversa. Il DMAC prima di poter utilizzare i bus, mettendo in bassa impedenza le proprie uscite, deve essere abilitato dal processore per poter far ciò. A tal fine, come detto, invia una richiesta di rilascio dei bus al processore (MBR attivo basso) e alla ricezione della risposta positiva (MBG attivo alto) acquisice il bus. Per quanto riguarda il rilascio, come detto, ci sono due modalità: ***burst*** e ***bus_stealing***.
Naturalmente nella prima modalità il bus verrà rilasciato solo alla fine del trasferimento di tutti i dati, mentre nella seconda modalità verrà rilasciato al trasferimento del singolo dato. Nelle figure 35 e 36 sono rappresentate le modalità di interazione tra il processore e il DMAC delle due modalità. Mentre nelle figure successive è rappresentato in dettaglio il diagramma di flusso del SCO nella gestione dei trasferimenti burt e bus_stealing.

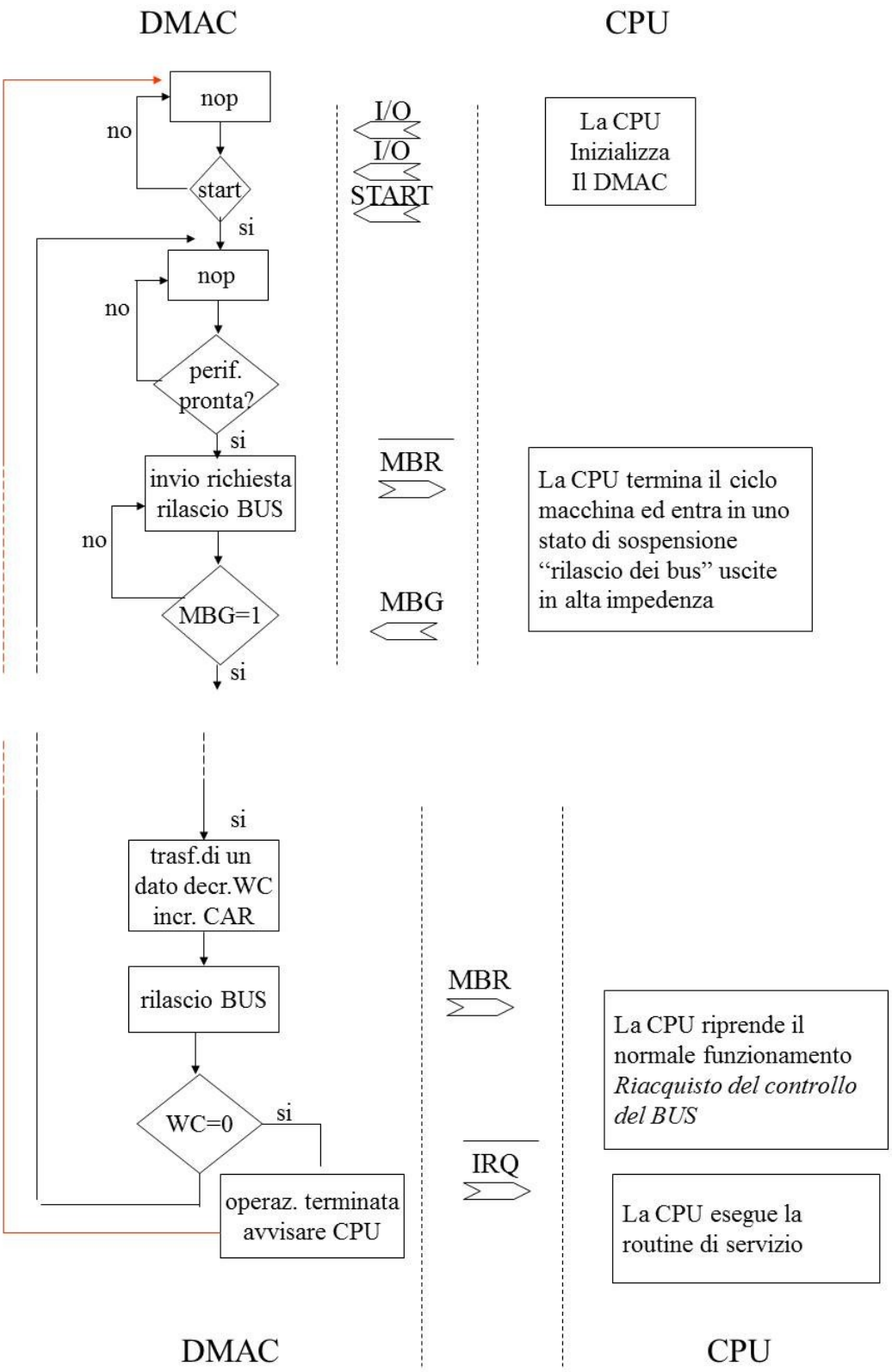

DMAC
CPU
nop
no
start
si
nop
no
perif. pronta?
si
invio richiesta rilascio BUS
no
MBG=1
si
I/O
I/O
START
MBR
MBG
La CPU Inizializza Il DMAC
La CPU termina il ciclo macchina ed entra in uno stato di sospensione "rilascio dei bus" uscite in alta impedenza
si
trasf.di un dato decr.WC incr. CAR
rilascio BUS
WC=0
si
operaz. terminata avvisare CPU
MBR
IRQ
La CPU riprende il normale funzionamento *Riacquisto del controllo del BUS*
La CPU esegue la routine di servizio
DMAC
CPU

Figura 35: Attività dello SCO del DMAC e del processore per trasferire dati in modalità Bus Stealing

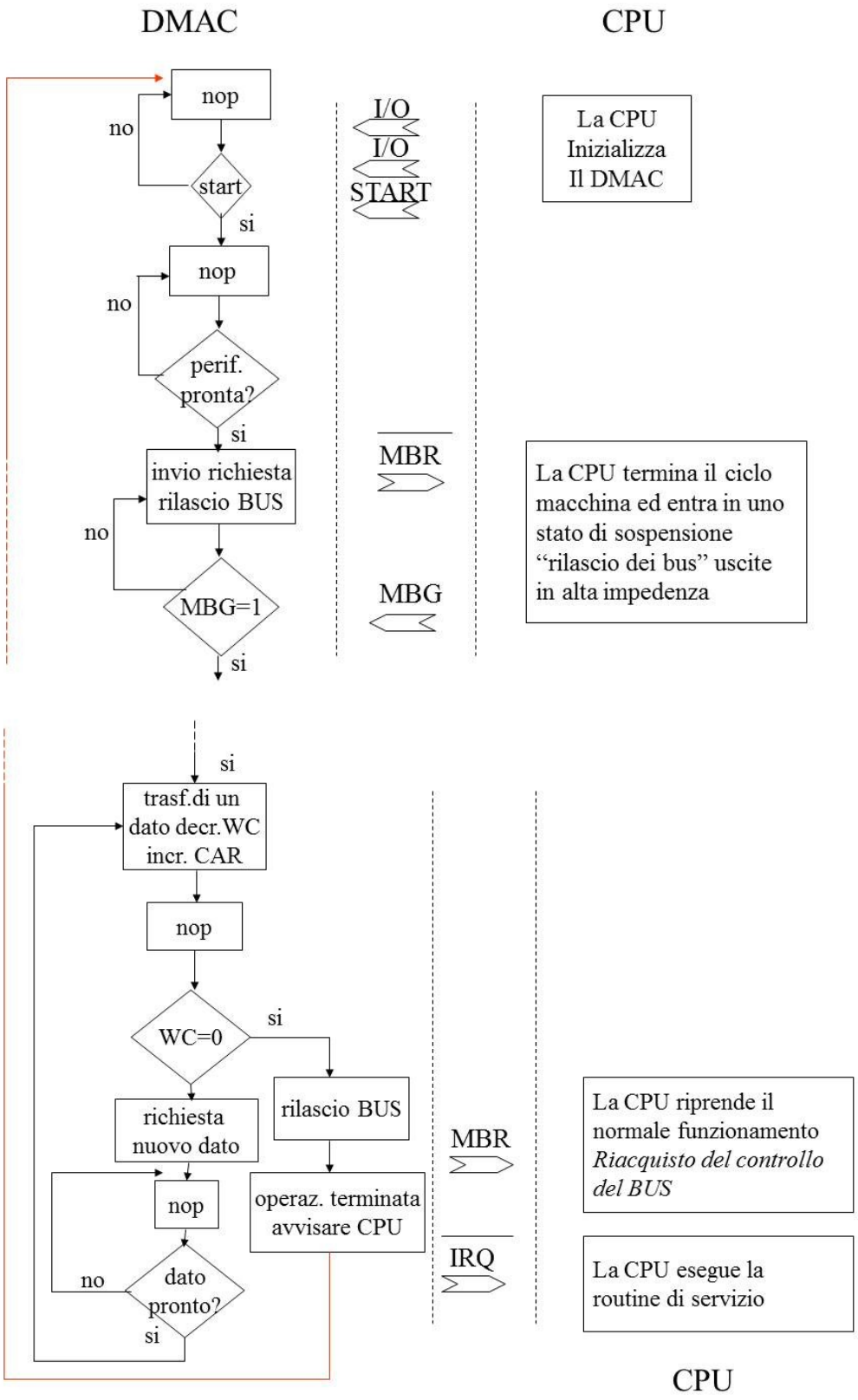
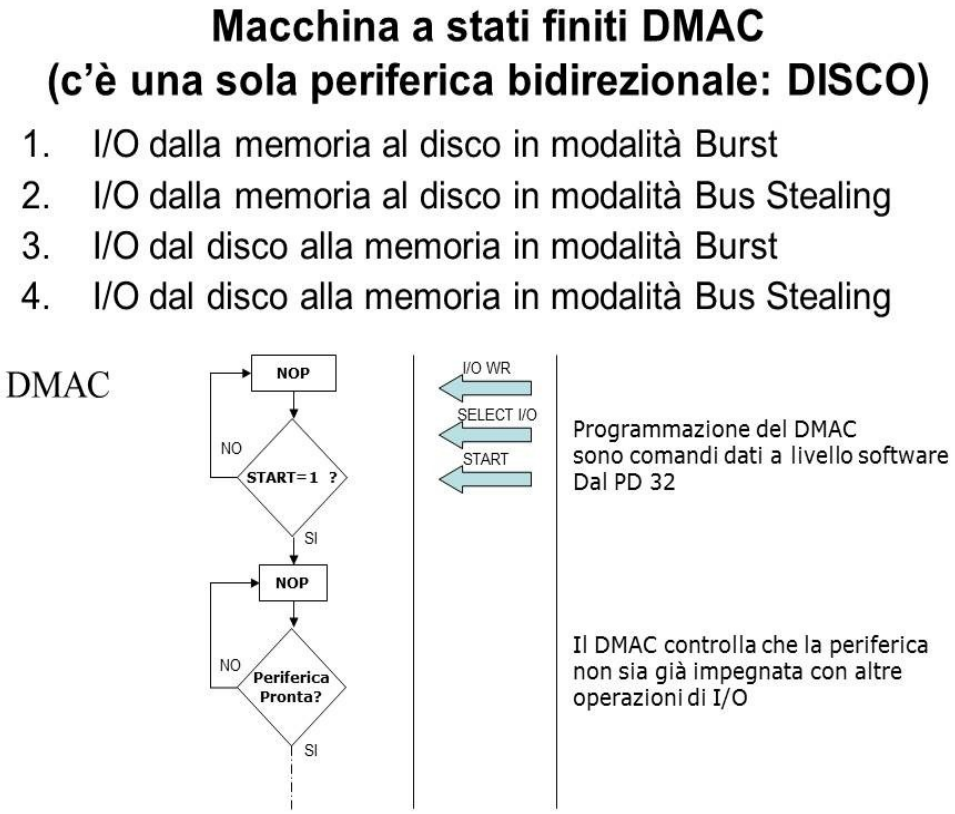
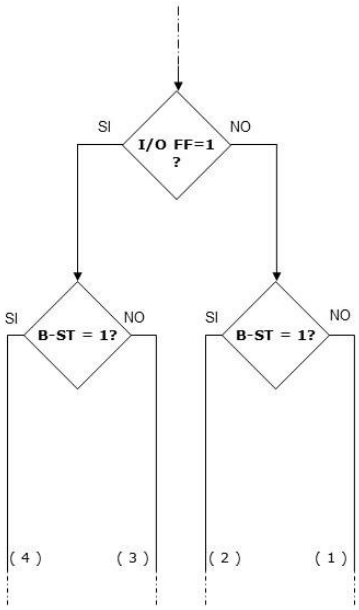


Figura 36: Attività dello SCO del DMAC e del processore per trasferire dati in modalità Bus Stealing



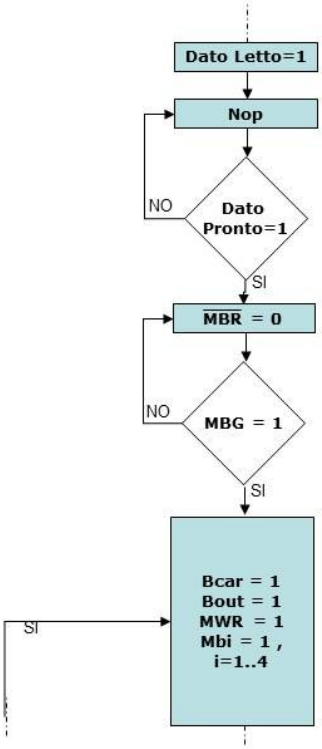


L'SCO del DMAC legge il valore del Flip Flop SR I/O:
-Se è 0 è un'interazione da disco verso la memoria
-Se è 1 è un'interazione dalla memoria al disco

L'SCO legge il valore del Flip Flop SR Burst Bus – St:
-Se è 0 si tratta di un uso del Bus di tipo Burst
-Se è 1 si tratta di un uso del Bus di tipo Burst Stealing

*si tratta di controlli implementati a livello Firmware

(1) Trasferimento da disco a memoria di tipo Burst 1 di 2



Lo SCO del DMAC invia allo SCO della periferica il segnale "Dato Letto"

Subito dopo lo SCO del DMAC si mette in ascolto del segnale "Dato Pronto", il quale verrà posto ad uno dalla periferica quando sarà pronto il dato da lei generato.

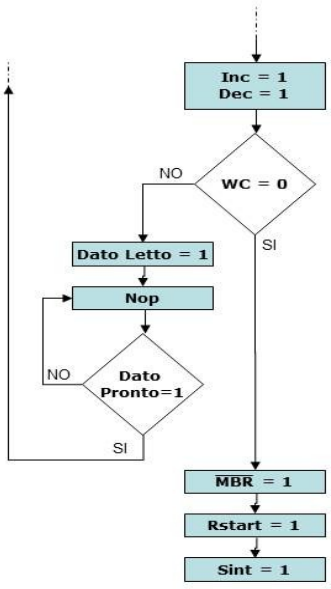
Viene richiesto il bus al Processore, da parte del DMAC, attraverso Il Memory Bus Request, funzionante in logica negata

Si attende che il PD32 abbia messo le sue uscite in alta impedenza, Il tutto avviene quando il segnale Memory Bus Grent, ascoltato dal DMAC, è posto ad uno dal PD32

Questa serie di segnali di controllo generati dal SCO del DMAC servono per scaricare i 30 bit dal CAR e i 32 del registo di out della periferica nella memoria.

Settare i quattro segnali Mbi, i=1,..,4 significa ipotizzare di scrivere simultaneamente 4 Byte allineati in memoria

(1) Trasferimento da disco a memoria di tipo Burst 2 di 2



Una volta effettuato il trasferimento viene incrementato di 4 il registro CAR e decrementato di 4 il registro WC

Si verifica se il World Counter è divenuto 0:
Se si il lavoro del DMAC è finito
Se no ci sono ancora dati da trasferire

Chiedo alla periferica di generare l'n-esimo dato

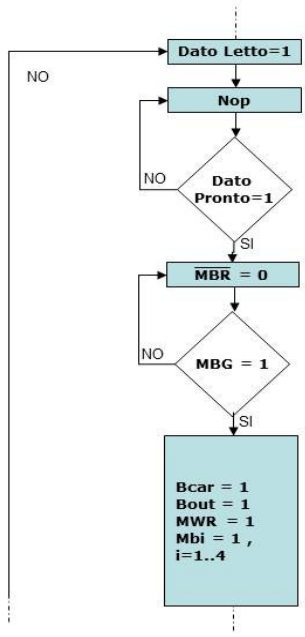
Lo SCO del DMAC si mette di nuovo in ascolto del segnale "Dato Pronto", il quale verrà posto ad uno dalla periferica quando sarà pronto il nuovo dato da lei generato.

Se il trasferimento è completato il DMAC rilascia il bus, il Processore si risveglia e continua il suo lavoro...

Resetta il Flip Flop SR "START"

Avvisa il PD32 di aver finito generando un Interrupt

(2) Trasferimento da disco a memoria di tipo Burst Stealing 1 di 2



Lo SCO del DMAC invia allo SCO della periferica il segnale "Dato Letto"

Subito dopo lo SCO del DMAC si mette in ascolto del segnale "Dato Pronto", il quale verrà posto ad uno dalla periferica quando sarà pronto il dato da lei generato.

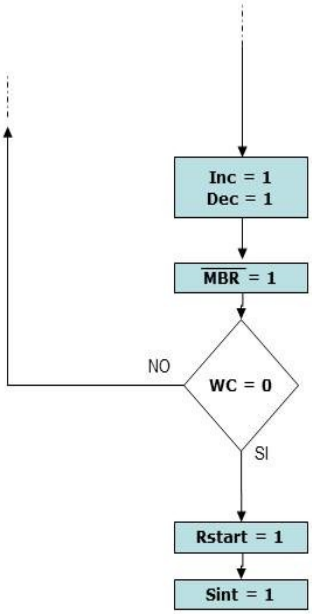
Viene richiesto il bus al Processore, da parte del DMAC, attraverso Il Memory Bus Request, funzionante in logica negata

Si attende che il PD32 abbia messo le sue uscite in alta impedenza, Il tutto avviene quando il segnale Memory Bus Grent, ascoltato dal DMAC, è posto ad uno dal PD32

Questa serie di segnali di controllo generati dal SCO del DMAC servono per scaricare i 30 bit dal CAR e i 32 del registo di out della periferica nella memoria.

Settare i quattro segnali Mbi, i=1,..,4 significa ipotizzare di scrivere simultaneamente 4 Byte allineati in memoria

(2) Trasferimento da disco a memoria di tipo Burst Stealing 2 di 2



Una volta effettuato il trasferimento viene incrementato di 4 il registro CAR e decrementato di 4 il registro WC

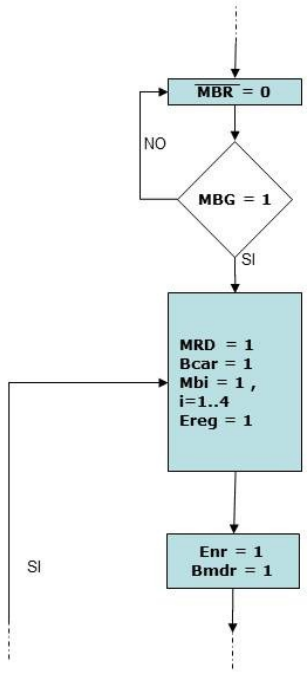
Un trasferimento è completato il DMAC rilascia il bus , il Processore si risveglia e continua il suo lavoro...

Si verifica se il World Counter è divenuto 0:
Se si il lavoro del DMAC è finito
Se no ci sono ancora dati da trasferire

Resetta il Flip Flop SR "START"

Avvisa il PD32 di aver finito generando un Interrupt

(3) Trasferimento da memoria a disco di tipo Burst 1 di 2



Viene richiesto il bus al Processore, da parte del DMAC, attraverso Il Memory Bus Request, funzionante in logica negata

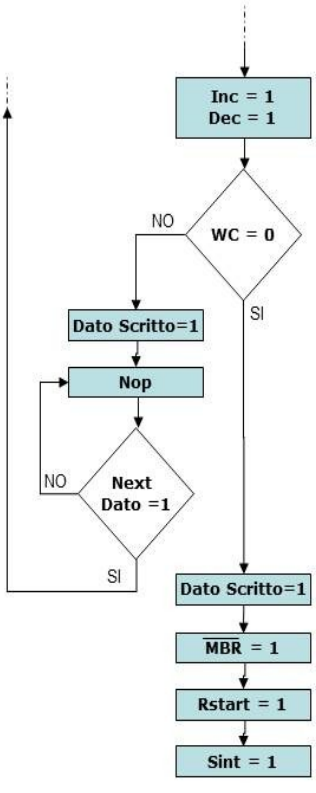
Si attende che il PD32 abbia messo le sue uscite in alta impedenza, Il tutto avviene quando il segnale Memory Bus Grent, ascoltato dal DMAC, è posto ad uno dal PD32

Questa serie di segnali di controllo generati dal SCO del DMAC servono per selezionare i dati indirizzati dal CAR e depositarli Nel REGin della periferica

Settare i quattro segnali Mbi, i=1,..,4 significa ipotizziamo di scrivere simultaneamente 4 Byte allineati in memoria

Trasferimento del dato dall'MDR al registro interno della periferica

(3) Trasferimento da memoria a disco di tipo Burst 2 di 2



Una volta effettuato il trasferimento viene incrementato di 4 il registro CAR e decrementato di 4 il registro WC

Si verifica se il World Counter è divenuto 0:
Se si il lavoro del DMAC è finito
Se no ci sono ancora dati da trasferire

Il DMAC avverte la periferica che il dato è stato scritto nel REGIN

Lo SCO del DMAC si mette in ascolto del segnale "Next Dato", il quale verrà posto ad uno dalla periferica quando sarà pronta a ricevere un nuovo dato.

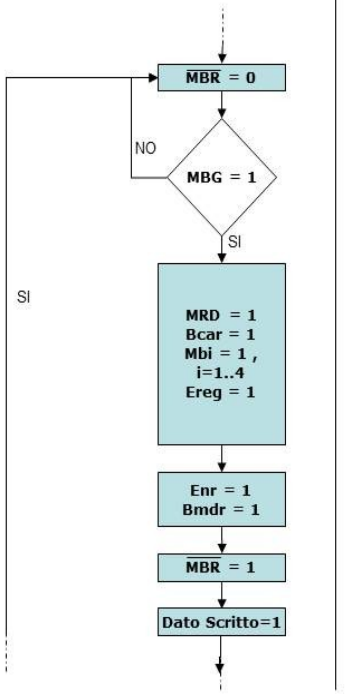
Il DMAC avverte la periferica che il dato è stato scritto nel REGIN

DMAC rilascia il bus, il Processore si risveglia e continua il lavoro...

Resetta il Flip Flop SR "START"

Avvisa il PD32 di aver finito generando un Interrupt

(4) Trasferimento da memoria a disco di tipo Burst Stealing 1 di 2



Viene richiesto il bus al Processore, da parte del DMAC, attraverso Il Memory Bus Request, funzionante in logica negata

Si attende che il PD32 abbia messo le sue uscite in alta impedenza, Il tutto avviene quando il segnale Memory Bus Grent, ascoltato dal DMAC, è posto ad uno dal PD32

Questa serie di segnali di controllo generati dal SCO del DMAC servono per selezionare i dati indirizzati dal CAR e depositarli Nel REGin della periferica

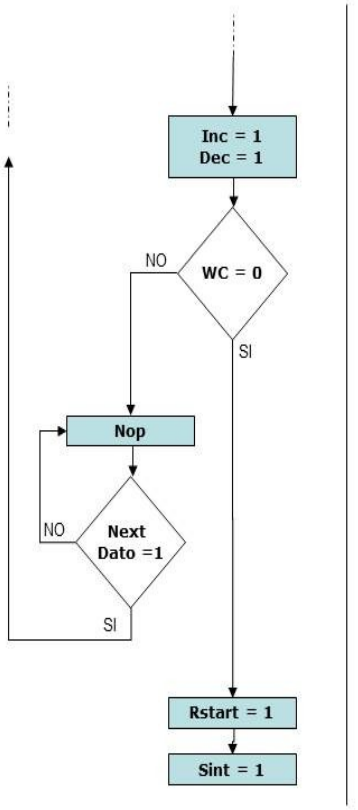
Settare i quattro segnali Mbi, i=1,..,4 significa ipotizzare di scrivere simultaneamente 4 Byte allineati in memoria

Trasferimento del dato dall'MDR al registro interno della periferica

Un trasferimento è completato il DMAC rilascia il bus , il Processore si risveglia e continua il suo lavoro...

Il DMAC avverte la periferica che il dato è stato scritto nel REGIN

(4) Trasferimento da memoria a disco di tipo Burst Stealing 2 di 2



Una volta effettuato il trasferimento viene incrementato di 4 il registro CAR e decrementato di 4 il registro WC

Si verifica se il World Counter è divenuto 0:
Se si il lavoro del DMAC è finito
Se no ci sono ancora dati da trasferire

Lo SCO del DMAC si mette in ascolto del segnale "Next Dato", il quale verrà posto ad uno dalla periferica quando sarà pronta a ricevere un nuovo dato, poi il DMAC andrà a chiedere di nuovo il Bus al Processore

Resetta il Flip Flop SR "START"

Avvisa il PD32 di aver finito generando un Interrupt

Figura 37: Flow chart del SCO del DMAC

DMAC MULTICANALE

Di seguito verrà presentata l'architettura di un DMAC per supportare il trasferimento dati su quattro canali differenti, ciò verrà fatto come semplice estensione del DMAC monocanale presentato fino ad ora. DMAC multicanali sono dispositivi costruiti dai produttori di chip quali l'8237 dell'Intel degli anni '80. Tali funzionalità poi nel tempo sono state incorporate in dispositivi più complessi, che mettevano e mettono in comunicazione i processori con i dispositivi di ingresso ed uscita, quali il North-Bridge e il South-Bridge sempre della Intel. Successivamente, dopo aver introdotto le memorie cache e i bus faremo vedere gli schemi di principio di tali tipi di componenti.

Il motivo di utilizzare più canali contemporaneamente nasce dal solito problema ingegneristico di ottimizzare le prestazioni, infatti le periferiche normalmente sono molto più lente dei dispositivi elettronici quali il DMAC e le memorie e quindi il poter interagire contemporaneamente con più dispositivi permette di utilizzare al meglio il DMAC e nel contempo ridurre il tempo complessivo del trasferimento dei dati da più periferiche. Infatti nel caso si volesse interagire con due periferiche nel caso di DMAC monocanale si dovrebbe prima interagire con una periferica e poi con l'altra, invece nel caso di DMAC multicanale tale interazione potrebbe essere effettuata in parallelo: mentre una periferica prepara/consuma un dato il DMAC potrebbe interagire con l'altra.

Come si vede in figura 38 per ogni canale si presenta lo stesso hardware del DMAC monocanale, di fatto cambia solo il SCO in quanto dovrà poter gestire contemporaneamente anche quattro trasferimenti. Naturalmente i segnali di controllo tra il processore e il DMAC rimarranno gli stessi, il SCO del DMAC per avvertire la CPU dell'avvenuto trasferimento di un file potrà utilizzare quattro IVN differenti per avvertire in modo appropriato il processore del completamento di uno dei possibili trasferimenti.

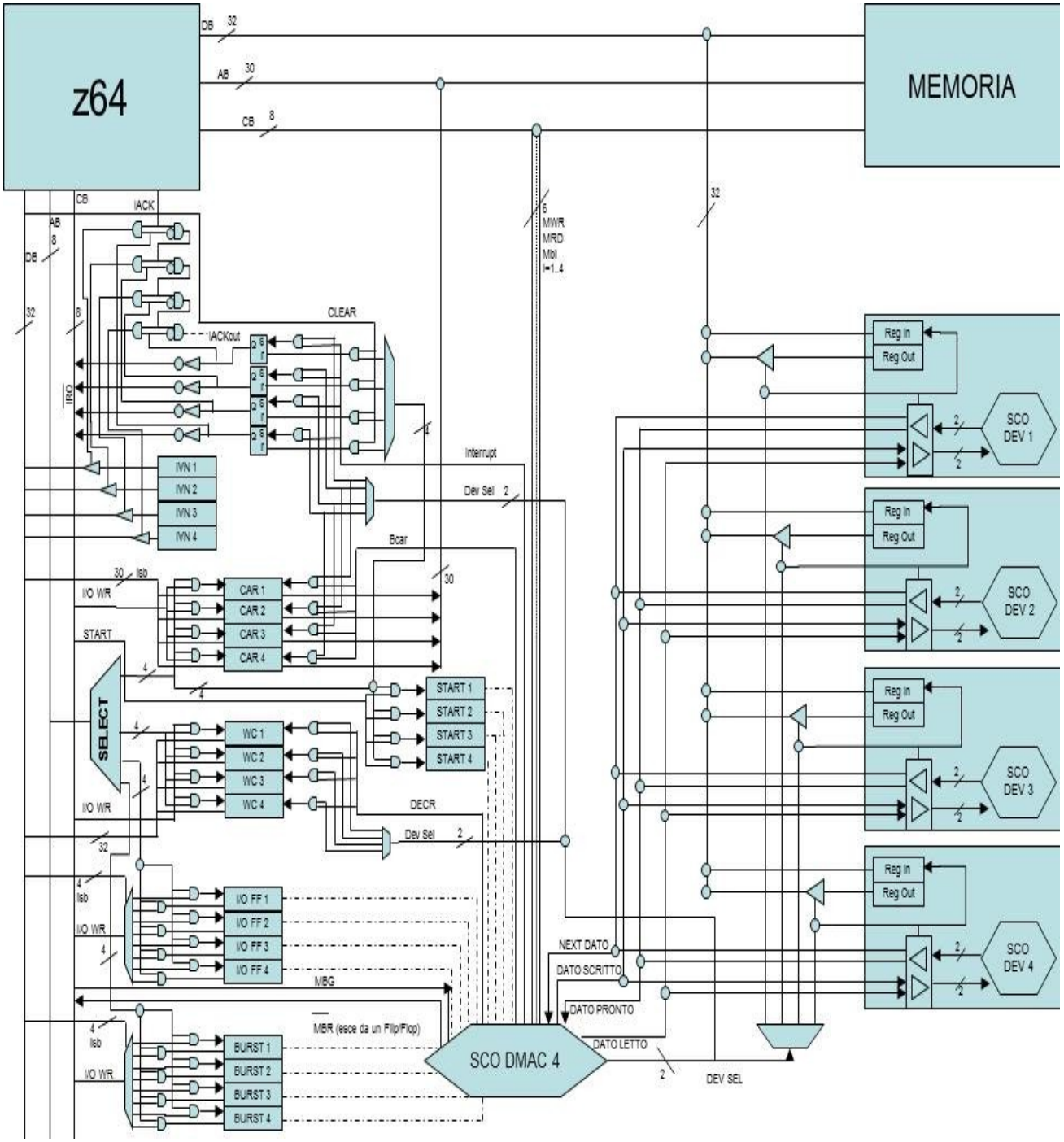


Figura 38: Architettura dello SCA di un DMAC multicanale

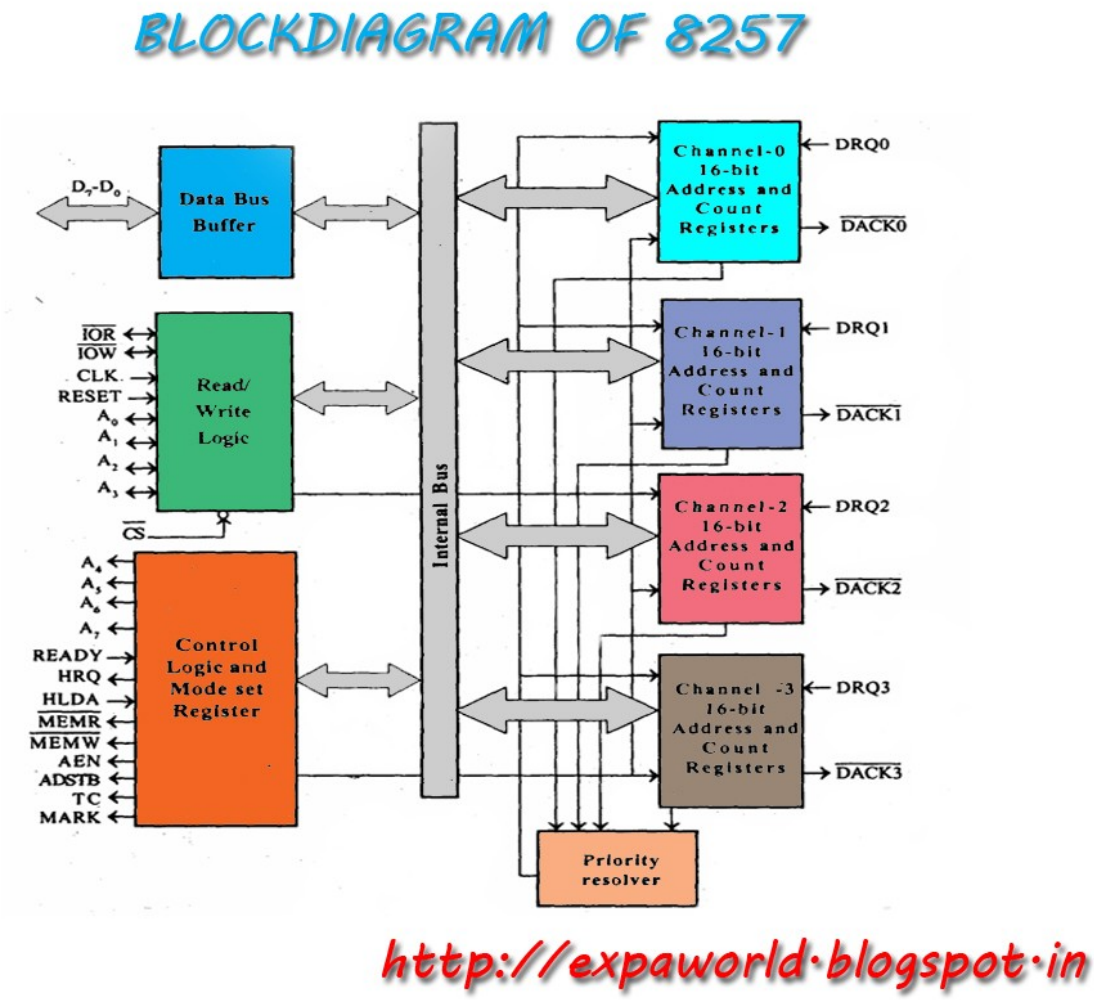


Figura 39: Architettura dell'8257

APPENDICE

Segnali di controllo dello z64 con 2 bus

Di seguito, per comodità del lettore, prima di elencare i cicli macchina (o cicli di bus) del processore e descriverne alcuni in dettaglio, vengono elencati integralmente i segnali di controllo in ingresso al processore (variabili di condizione del SCO del processore) e i segnali di controllo in uscita dal processore (generati dal SCO del processore) necessari per interagire con le unità esterne. Alcuni segnali di controllo sono stati introdotti nelle Sezioni precedenti, altri invece saranno descritti in dettaglio successivamente. Il riepilogo viene fatto separando i segnali di ingresso da quelli in uscita. I segnali di ingresso ed uscita corrispondenti sono disposti sulla stessa riga.

USCITE del SCO **- segnali di controllo -**	**INGRESSI del SCO** **- variabili di condizione -**
Mb_i (i= 0, 1, . . . , 8)	
MRD (Memory Read)	
MWR (Memory Write)	
I/O (I/O)	
RD (I/O Read)	
WR (I/O Write)	
INTA (Interrupt Acknowledgment)	$\overline{\text{IRQ}}$ (Interrupt Request)
MBG (Memory Bus Grant)	$\overline{\text{MBR}}$ (Memory Bus Request)
	$\overline{\text{WAIT}}$
	$\overline{\text{RESET}}$

Di seguito viene data una descrizione delle attività collegate ai segnali di controllo.

Mb_i (i= 0, 1, . . . , 8)

Segnale di controllo generato dal SCO all'atto dell'interpretazione di un'istruzione macchina di tipo trasferimento dati da/verso la memoria. Questo segnale di controllo abilita l'accesso al modulo di memoria i-esimo.

MRD (Memory Read)

Segnale di controllo generato dal SCO all'atto dell'interpretazione di un'istruzione macchina di tipo trasferimento dati da memoria. Questo segnale di controllo abilita la lettura del dato memorizzato nella cella (o nelle celle se si tratta di dati a due o quattro byte) di memoria specificata nell'AB e il suo trasferimento sul DB.

MWR (Memory Write)

Segnale di controllo generato dal SCO all'atto dell'interpretazione di un'istruzione macchina di tipo trasferimento dati a memoria. Questo segnale di controllo abilita la scrittura del dato sul DB nella cella (o nelle celle se si tratta di dati a due o quattro byte) di memoria specificata dall'AB.

I/O (Input/Output)

Segnale di controllo generato dal SCO all'atto dell'interpretazione di un'istruzione macchina di tipo trasferimento dati da o verso una porta di I/O. Segnale non necessariamente utile nel caso di presenza di due bus, ma fondamentale nel caso di uso di un solo bus

RD (I/O Read)

Segnale di controllo generato dal SCO all'atto dell'interpretazione di un'istruzione macchina di tipo trasferimento dati da una porta di ingresso. Questo segnale di controllo abilita la lettura del dato memorizzato nel registro della porta di ingresso specificata nell'I/OAB e il suo trasferimento sull'I/ODB.

WR (I/O Write)

Segnale di controllo generato dal SCO all'atto dell'interpretazione di un'istruzione macchina di tipo trasferimento dati a porta di uscita. Questo segnale di controllo abilita la scrittura del dato sull'I/ODB nel registro della porta di uscita specificata dall'I/OAB.

$\overline{IRQ}$ (Interrupt Request)

Segnale di controllo generato da un dispositivo esterno per sospendere l'attività corrente del processore e fargli eseguire un programma di servizio.

INTA (INTerrupt Acknowledgment)

Segnale di controllo generato dal SCO del processore per avvertire il dispositivo generante la richiesta dell'interruzione che ha sospeso la precedente attività e che attende l'identificazione del richiedente.

$\overline{\text{MBR}}$ (Memory Bus Request)

Segnale di controllo generato da un dispositivo esterno per richiedere al SCO del processore di mettere le proprie uscite sui bus esterni in alta impedenza. Questo segnale di controllo deve essere mantenuto fino a che il dispositivo esterno non ha finito di utilizzare i suddetti bus.

MBG (Memory Bus Grant)

Segnale di controllo generato dal SCO del processore per avvertire il dispositivo generante la richiesta di rilascio dei bus che ha effettivamente messo in alta impedenza le proprie uscite e quelle del SCA interno del processore verso i bus esterni.

$\overline{\text{WAIT}}$

Segnale di controllo generato da un dispositivo esterno per "rallentare" il SCO del processore nell'interazione con il dispositivo stesso. Questo segnale deve rimanere basso fino a che il dispositivo esterno non ha completato il trasferimento richiesto.

$\overline{\text{RESET}}$

Segnale di controllo generato da un dispositivo esterno per forzare il processore ad eseguire un programma da un indirizzo prefissato. All'atto della ricezione di questo segnale il SCO prende il contenuto informativo presente sull'I/ODB e lo pone sul PC, pone a zero i flip-flop del registro SR (quindi disabilita le interruzioni).

Passi elementari dell'interpretazione di una istruzione

Come visto l'interpretazione di ogni istruzione del processore comporta l'esecuzione di un microprogramma da parte del SCO. L'esecuzione del microprogramma relativa ad una istruzione viene detta *ciclo istruzione*. Ogni ciclo istruzione è composto da una o più fasi elementari che comporteranno l'attivazione di comandi (*micro operazioni*) relativi ad unità interne al processore (registri, ALU, shift register) e/o relative ad unità esterne allo stesso (memoria, unità periferiche). Le fasi che comportano interazione con le unità esterne vengono anche dette *cicli macchina.* A sua volta ogni ciclo macchina può essere costituito da uno o due *cicli*

di bus; per esempio la lettura di una parola memorizzata su due byte non allineati sullo stesso indirizzo di riga necessita di due accessi in memoria (cioè di due cicli di bus).

I cicli macchina possibili per il processore z64 sono:

- fetch dell'istruzione;
- lettura da memoria;
- scrittura in memoria;
- lettura da memoria tramite stack pointer;
- scrittura in memoria tramite stack pointer;
- lettura da periferica;
- scrittura in periferica;
- lettura di stringa da periferica;
- scrittura di stringa in periferica;
- riconoscimento di interruzione;
- riconoscimento di interruzione nello stato di halt;
- riconoscimento di $\overline{\text{RESET}}$;

Ogni ciclo istruzione è composto da almeno un ciclo macchina (uguale per tutti i cicli istruzioni) per l'estrazione dalla memoria dell'istruzione da eseguire (fetch dell'istruzione), da una o più attività interne ed eventualmente da altri cicli macchina se la semantica dell'istruzione comporta ulteriori interazioni con le unità esterne.

I cicli macchina sono costituiti da una sequenza di microistruzioni (o stati operativi) il cui numero dipende dal tipo di ciclo.

Di seguito si faranno vedere le attività dello z64 relativamente ad alcuni cicli macchina.

Ciclo di fetch

In questo ciclo si legge dalla memoria la prossima istruzione da eseguire e si predispone il Register Instruction Pointer (RIP) a puntare alla successiva istruzione. Poichè il RIP non è connesso al bus esterno degli indirizzi il contenuto del RIP è messo sul Memory Address Register (MAR), che ha compiti di interfaccia tra la

CPU e il bus degli indirizzi. L'istruzione da eseguire deve essere caricata sull'Instruction Register (IR) per essere interpretata dal SCO.

```
RIP  -> MAR;            /* trasferimento del contenuto del RIP sul
                           MAR */
(MAR) -> IR63-0;        /* trasferimento dell'istruzione da eseguire
                           sull'IR */
RIP + 8 -> RIP ;        /* predisposizione esecuzione istruzione
                           successiva */
```

Da notare che questo ciclo macchina può essere costituito da uno o due cicli di bus a secondo se il contenuto di RIP è multiplo o meno di otto (se non è multiplo di otto c'è un disallineamento nella memorizzazione del codice nella memoria).

Ciclo di riconoscimento di interruzione

Questo ciclo viene eseguito se le interruzioni sono abilitate ed esiste una richiesta di interruzione. La richiesta di interruzione viene verificata alla fine di ogni ciclo istruzione. Questo ciclo serve ad identificare l'indirizzo iniziale del sottoprogramma di servizio (driver) associato alla richiesta di interruzione. A tal fine il SCO si aspetta che un dispositivo esterno invii sull'I/ODB un byte che una volta moltiplicato per otto, fornisce l'indirizzo dell'elemento del vettore delle interruzioni contenente l'indirizzo iniziale del programma di servizio. Questo ciclo è seguito da due cicli di scrittura in memoria tramite stack pointer per il salvataggio del RIP e dello SR, dopodiché inizierà l'esecuzione del programma di servizio.

/**ciclo di riconoscimento di interrupt* */

```
1 -> INTA;              /*generazione segnale di
                           riconoscimento di interruzione */
I/ODB -> I/ODR;         /*prelievo identificatore dispositivo*/
I/ODR* 8 -> MAR;        /*individuazione della locazione del vettore
                           di interruzione in cui è memorizzato
                           l'indirizzo iniziale del programma di
                           servizio richiesto*/
(MAR) -> TEMP2;         /*caricamento dell'indirizzo iniziale del
                           programma di servizio in un registro
                           temporaneo*/
```

/*primo *ciclo di scrittura in memoria tramite stack pointer*
per salvare RIP del programma corrente */

```
RSP - 8 -> RSP;          /*decremento stack pointer*/
RSP -> MAR;
RIP -> MDR;
MDR -> (MAR);            /*memorizzazione del contenuto del RIP
                         relativo al programma corrente nello
                         stack*/
```

/* secondo *ciclo di scrittura in memoria tramite stack pointer* per salvare SR del programma corrente*/

```
RSP - 8 -> RSP;          /* decremento stack pointer*/
RSP -> MAR;
SR -> MDR;
MDR -> (MAR);            /*memorizzazione del contenuto del SR
                         relativo al programma corrente nello
                         stack*/
```

/**attività interne* per la disabilitazione delle interruzioni e la memorizzazione nel PC dell'indirizzo iniziale del programma di servizio*/

```
0 -> I;
TEMP2 -> PC;
```

/* *ciclo di fetch* per prelevare la prima istruzione del programma di servizio*/

<vedi ciclo di fetch>

Da notare che il ciclo di scrittura in memoria tramite stack pointer può essere costituito da uno o due cicli di bus a seconda se il contenuto di RSP è multiplo o meno di otto (se non è multiplo di otto c'è un disallineamento nella memorizzazione del dato).