Progetto

Le modalità di interazione di tipo programmato possono essere di due tipi:
- **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);
- **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.

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**.

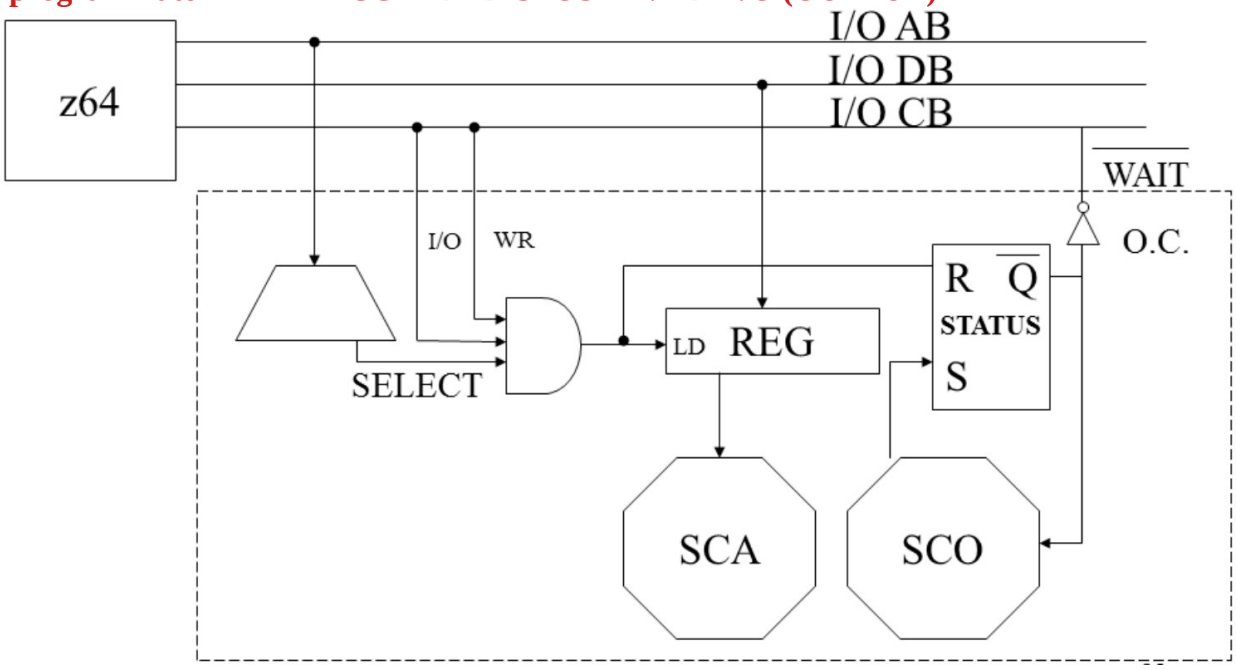
Nel primo caso il dispositivo indirizzato (cioè quello con cui la CPU necessita di interagire) deve leggere e/o scrivere il dato nell'intervallo di tempo previsto dal SCO del processore per la lettura e/o la scrittura e quindi sincrono con la velocità di funzionamento del processore, come nel caso di 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 **WAIT** che viene usato dal SCO del processore per individuare se la periferica ha completato o meno il trasferimento richiesto.
Mentre la soluzione **software** prevede, come vedremo, l'utilizzazione di istruzioni software ad hoc necessarie per implementare il protocollo di comunicazione.
Le istruzioni : ***seti*** e ***clri***, altro non sono che una istanziazione della manipolazione di un flip/flop del registro di stato, SR).

Interazione busy waiting implementata a firmware

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.
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.
Nel caso di utilizzazione di un unico bus c'è 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**)
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.

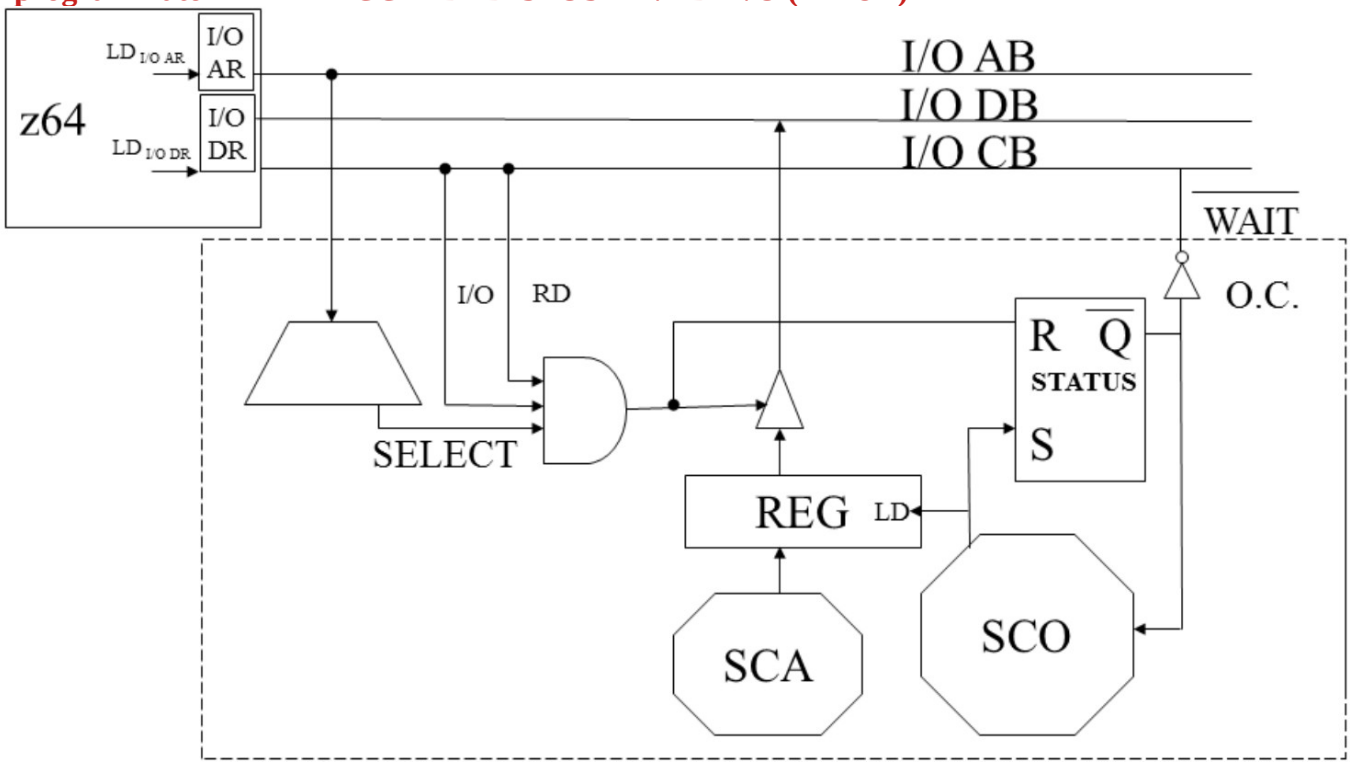
Per esempio sarebbe sufficiente affiancare ai segnali RD e WR un segnale di controllo che indichi se tale interazione è verso la memoria o meno. In modo alternativo si potrebbero utilizzare due insiemi di segnali di controllo, uno dedicato all'interazione con la memoria (MR e MW) ed uno dedicato all'interazione con i dispositivi di I/O (I/OR e I/OW).
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.

I/O programmato INTERFACCIA DI DISPOSITIVI DI I/O (OUTPUT)



Rappresenta 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 .
All'inizio del periodo successivo il SCO verifica se la variabile di condizione WAIT è alta o bassa; nel caso di 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.

I/O programmato INTERFACCIA DI DISPOSITIVI DI I/O (INPUT)

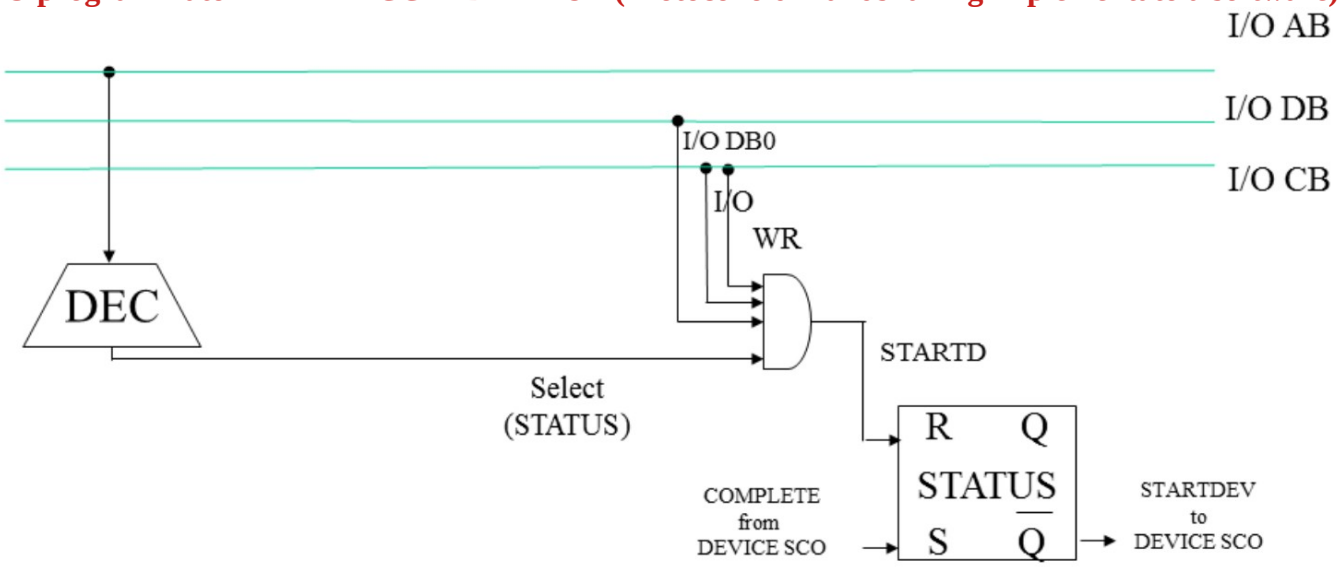


Rappresenta il ciclo di lettura da periferica del processore.
Durante il primo periodo di clock il SCO del processore genera i segnali di comando I/O e RD e abilita la scrittura dell'indirizzo del registro della periferica sull'address bus (I/OAB) da cui bisogna leggere il dato, notare che tale attività contemporaneamente resetta il F/F Status-1. Nel periodo successivo il SCO verifica se la variabile di condizione WAIT è alta o bassa; nel caso di 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.
I periodi in cui il SCO è in attesa che il segnale WAIT diventi alto sono indicati con TW (Time Wait).
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.

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.
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.

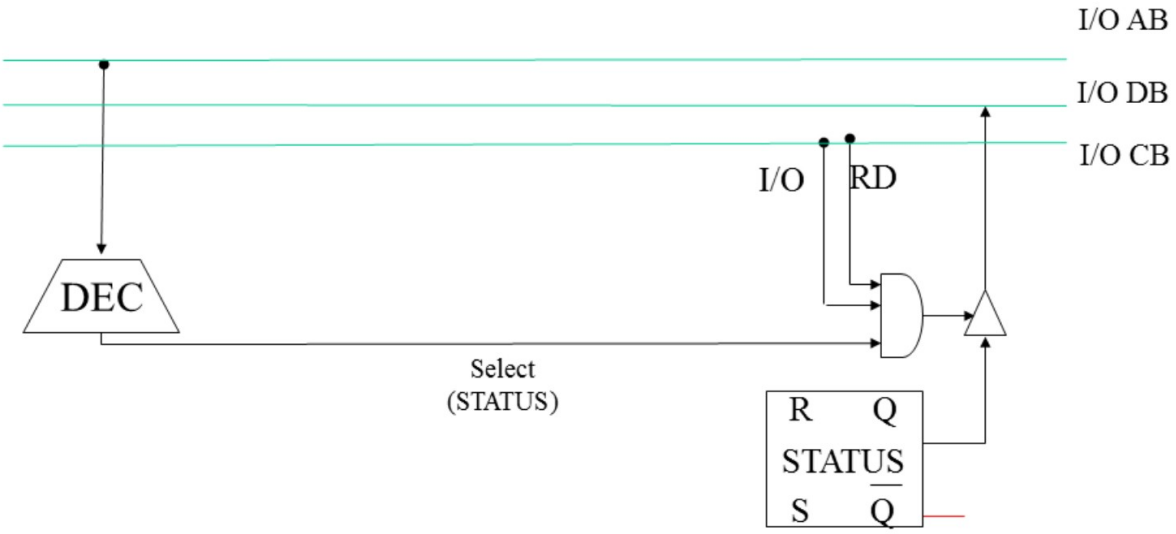
I/O programmato INTERFACCIA DI INPUT (Protocollo di handshaking implementato a software)



Hardware necessario per avvertire la periferica

Per avvertire la periferica che il processore vuole interagire con essa si resetta il flip/ flop STATUS. 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.

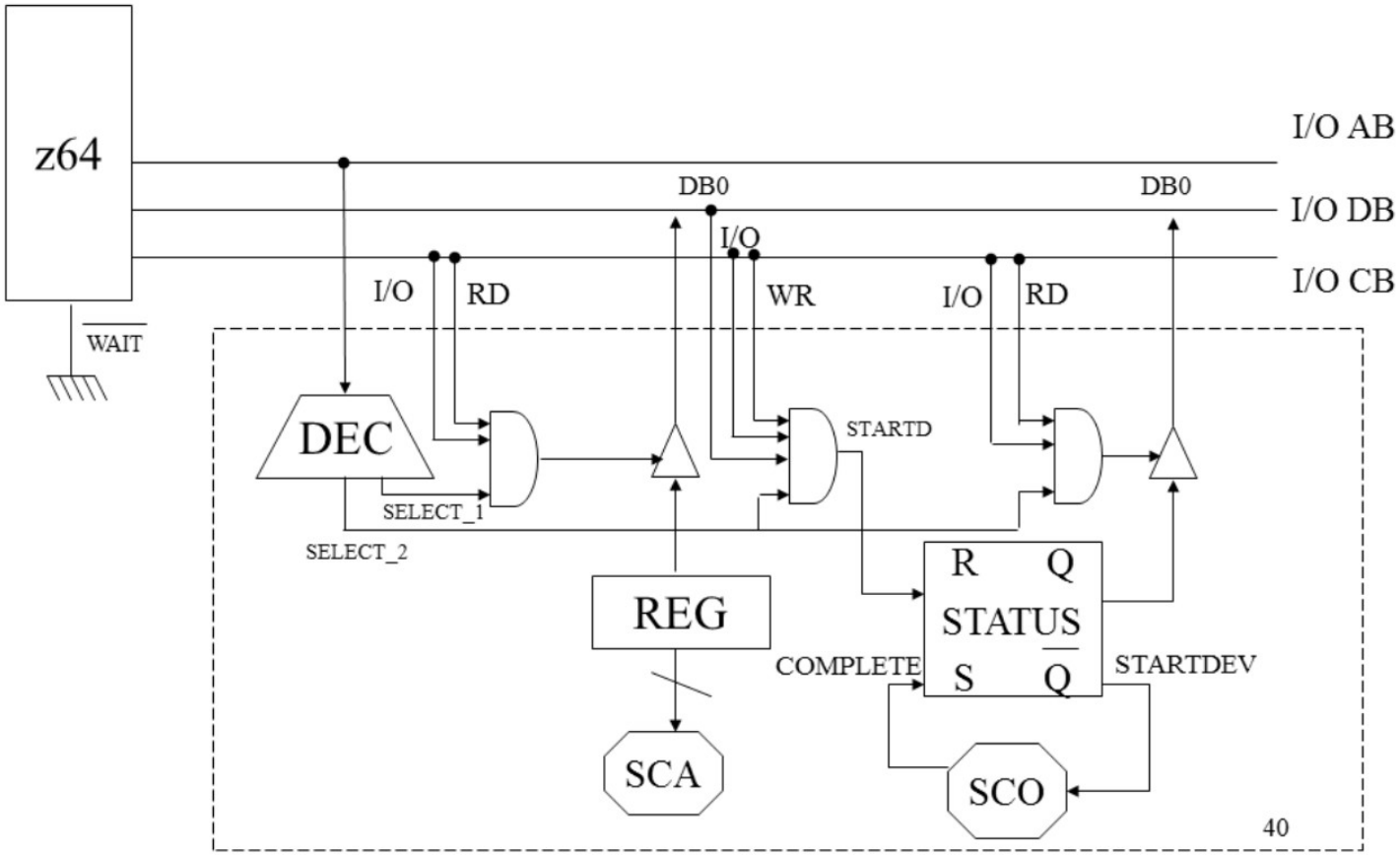
I/O programmato INTERFACCIA DI INPUT (Protocollo di handshaking implementato a software)



Hardware necessario per verificare se la periferica è pronta

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.

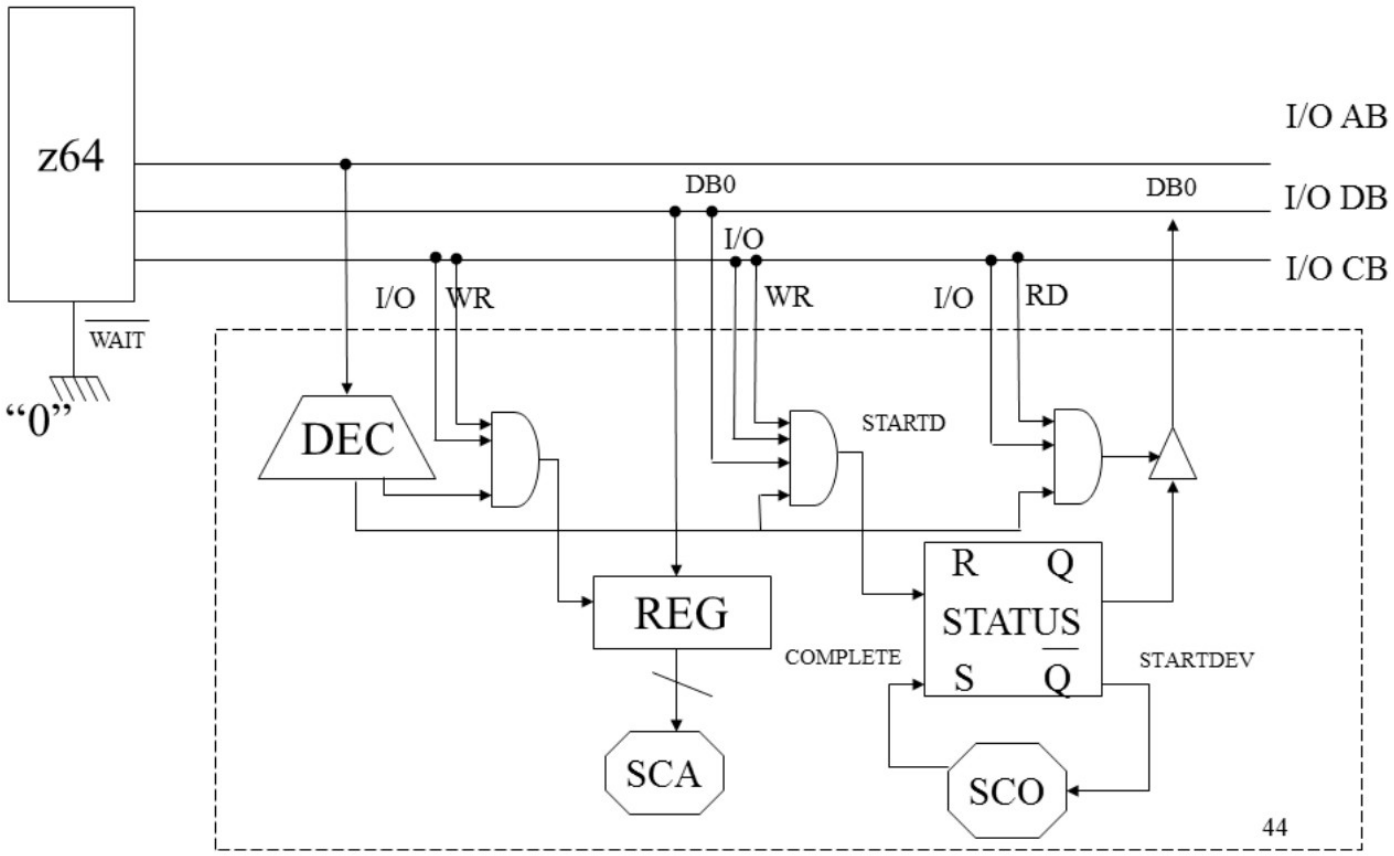
I/O programmato – INTERFACCIA di INPUT Schematizzata (Protocollo di hanshaking implementato a software)



Porzione d 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 che la periferica sia pronta
            MOVW $DEVICE_IN, %dx
            INL %dx, %eax          #Input da periferica
```

I/O programmato – INTERFACCIA di OUTPUT Schematizzata (Protocollo di hanshaking implementato a software)



Porzione di programma assebly 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. A nel carry
                JNC Aspetta                     #Attendi che la periferica sia pronta
```

I/O Programmato MODAITA' BUSY WAITING

```
            …..
            movl $0, %ecx                   # contatore dei dati inizializzato a 0
            movq $DATI, %rdi                # inizializzazione della locazione di memoria

.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 memoria
            movw %ax, (%rdi, %rcx, 2)       # …. lo copio in memoria (nel vettore)
            addl $1, %ecx                   # incremento il contatore
            cmpl $100, %ecx
            jnz .loop
            hlt
```

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

.poll:
                movw $AD1, %dx
                inb %dx, %ax                    # leggo il valore di STATUS da AD1
                btb $0, %ax
                jc .acquisisci_dati1            # se è pronta acquisisco i dati
                movw $AD2, %dx
                inb %dx, %ax                    # leggo il valore di STATUS da AD2
                jc .acquisisci_dati2            # se è pronta acquisisco i 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)      # … lo trasferisco in memoria
                addl $1, %ecx                   # cotrollo di uscita dal ciclo di polling
                jnz .return
                hlt
.return:
                ret
```

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.

GESTIONE DEGLI INTERRUPT

L'arrivo di una interruzione può essere schematizzata nel seguente modo:
- 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.

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.

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 il controllo del SCO una volta eseguito un microprogramma relativo ad una istruzione macchina ritorna al microprogramma relativo alla fase di fetch.
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.

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.

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.
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.

Esecuzione del programma di servizio

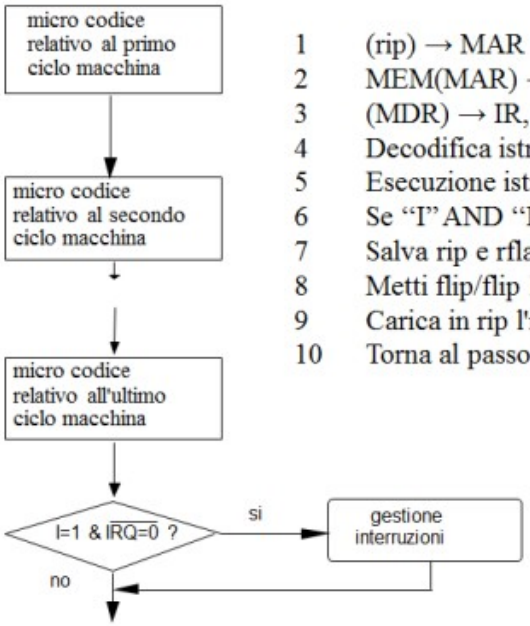
Per garantire che l'interruzione dell'esecuzione del programma corrente non comporti un'interferenza sulla sua evoluzione è necessario che, oltre alle informazioni salvate dal SCO (contenuto del rip e rflags), nello stack vengano anche memorizzati i contenuti di quei registri che il programma di servizio 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:



1 (rip) → MAR --- fetch
2 MEM(MAR) → MDR --- fetch
3 (MDR) → IR, (rip)+8 → rip --- fetch
4 Decodifica istruzione
5 Esecuzione istruzione
6 Se "I" AND "$\overline{IRQ}$" = 0 allora vai a 7 altrimenti a 10
7 Salva rip e rflags nella pila puntata da rsp
8 Metti flip/flip I a 0
9 Carica in rip l'indirizzo iniziale del programma di servizio
10 Torna al passo 1 fino a che trovi una istruzione HALT

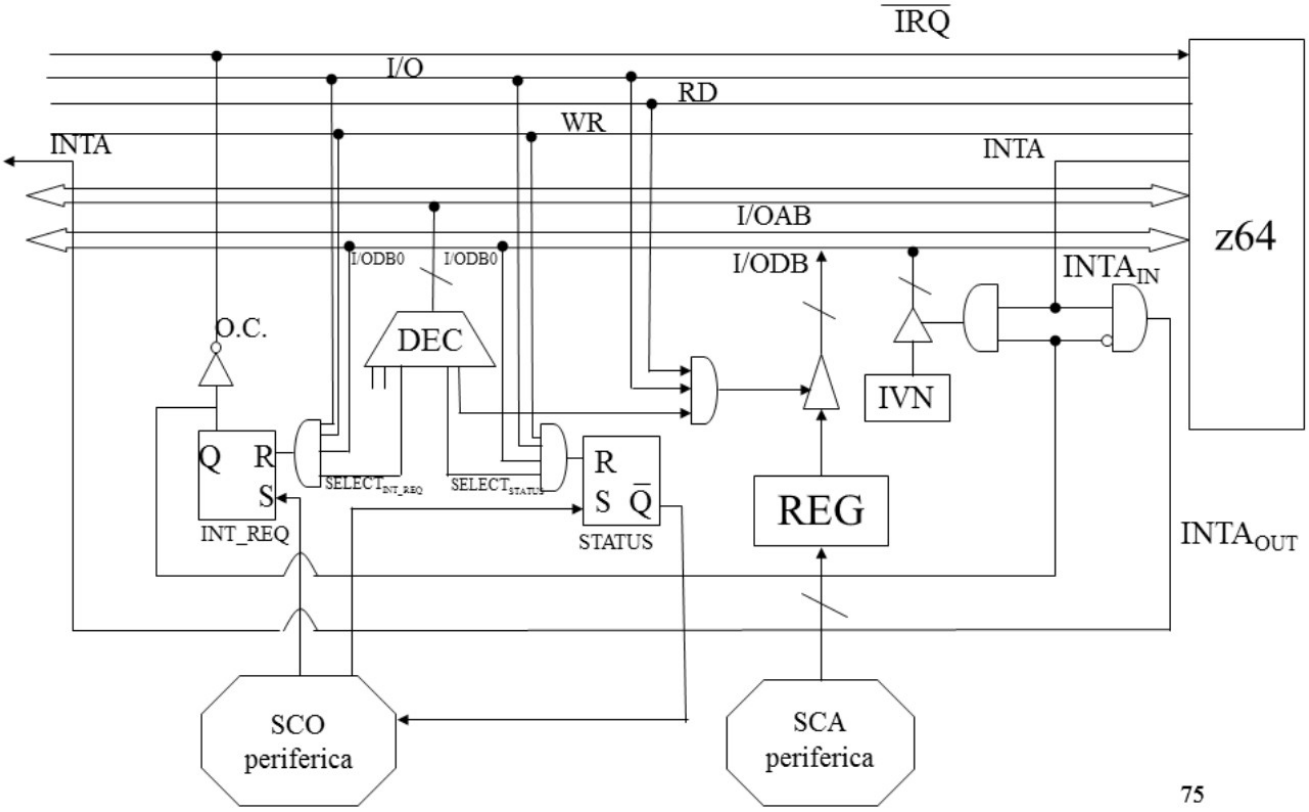
Da notare che il flip/flop I viene posto a 0 dallo SCO del processore nel microprogramma relativo all'attivazione del driver in quanto altrimenti appena inizia l'esecuzione del driver il SCO si trova ancora la richiesta delle interruzioni attiva.
Per evitare che terminata l'esecuzione del driver ci sia 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.
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

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.

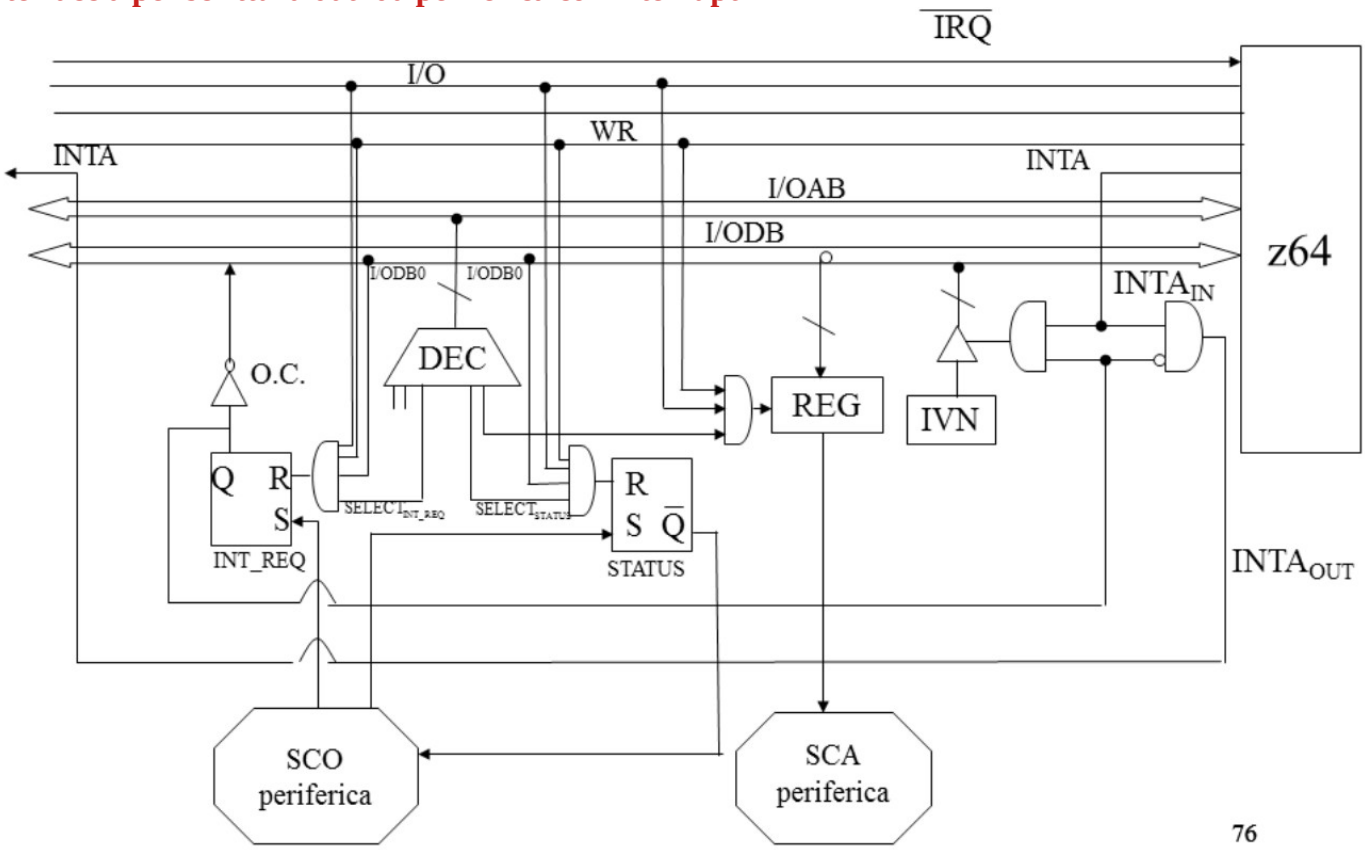
Come resettare flip/flop INT_REQ

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.

Interfaccia per lettura dati da periferica con interrupt



Interfaccia per scrittura dati da periferica con interrupt



ESEMPIO

```
.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
```

Operazioni di I/O gestite da canale

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 comeuna periferica. Il DMAC potrebbe nascondere le periferiche al processore oppure potrebbe utilizzare lo stesso bus di I/O 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.
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, 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: MBR e MBG).

Gestione delle richieste di accesso ai bus

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 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 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 MBR).

Architettura del SCA del DMAC

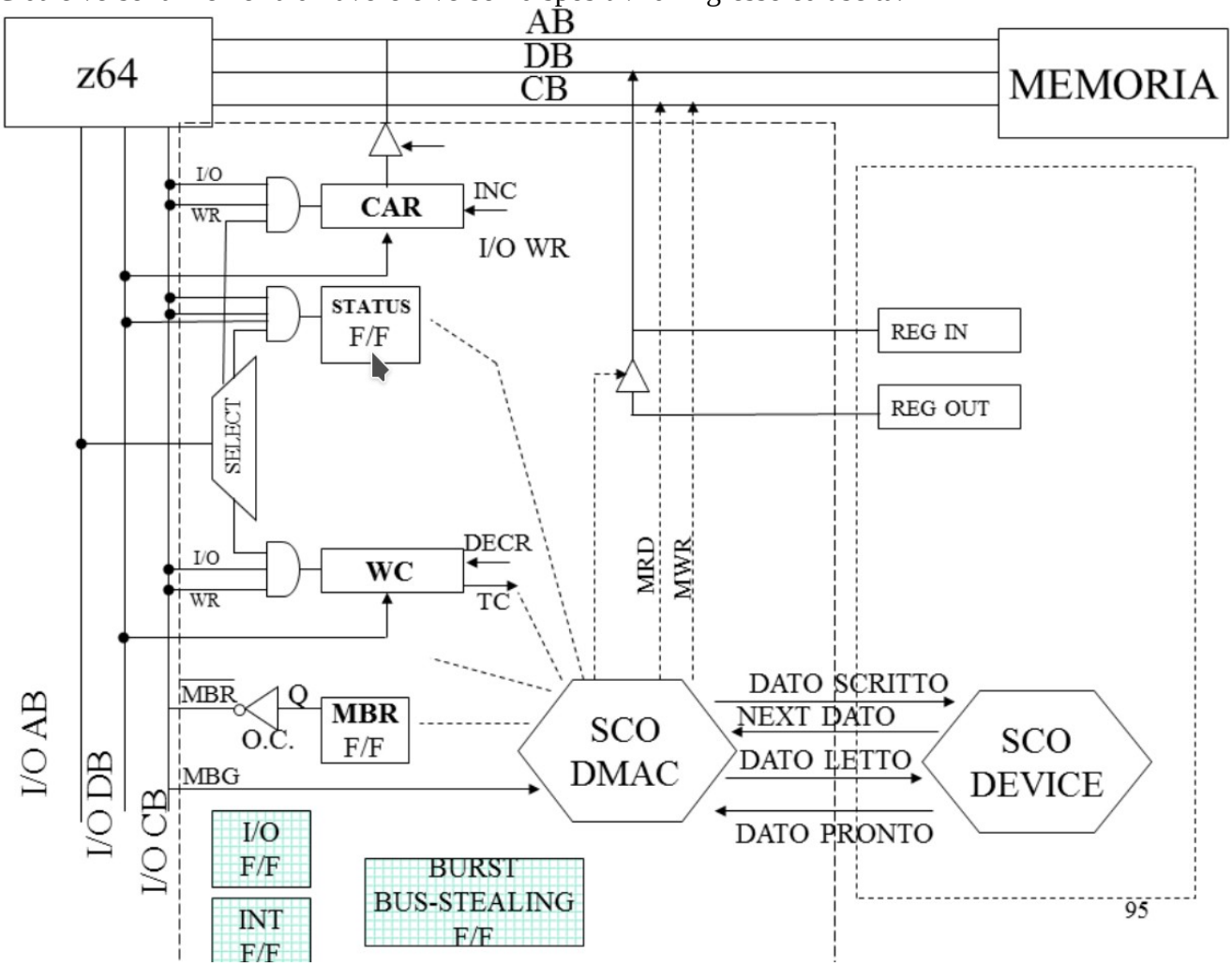
Le funzioni del DMAC sono solo quelle di trasferire dati in modo efficiente al posto del processore, pertanto tutte le informazioni necessarie al trasferimento, quali:

- 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,

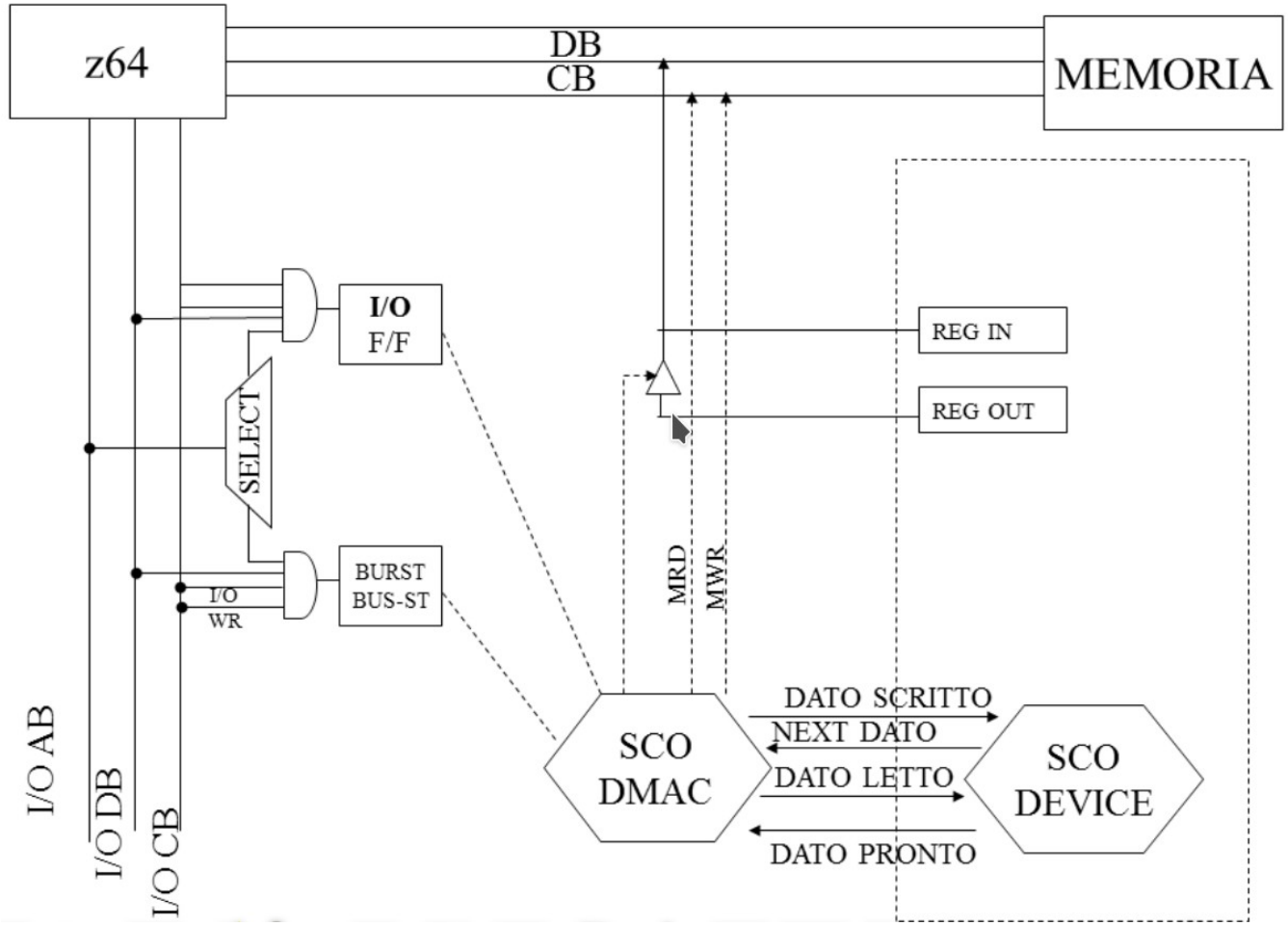
sono trasferite dal processore verso il SCA del DMAC. Il processore vede il DMAC come una periferica, pertanto potrà eseguire una serie di istruzioni di output per trasferire tali informazioni, che saranno utilizzate dal SCO del DMAC per trasferire i dati secondo le direttive del processore.

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.
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.
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.

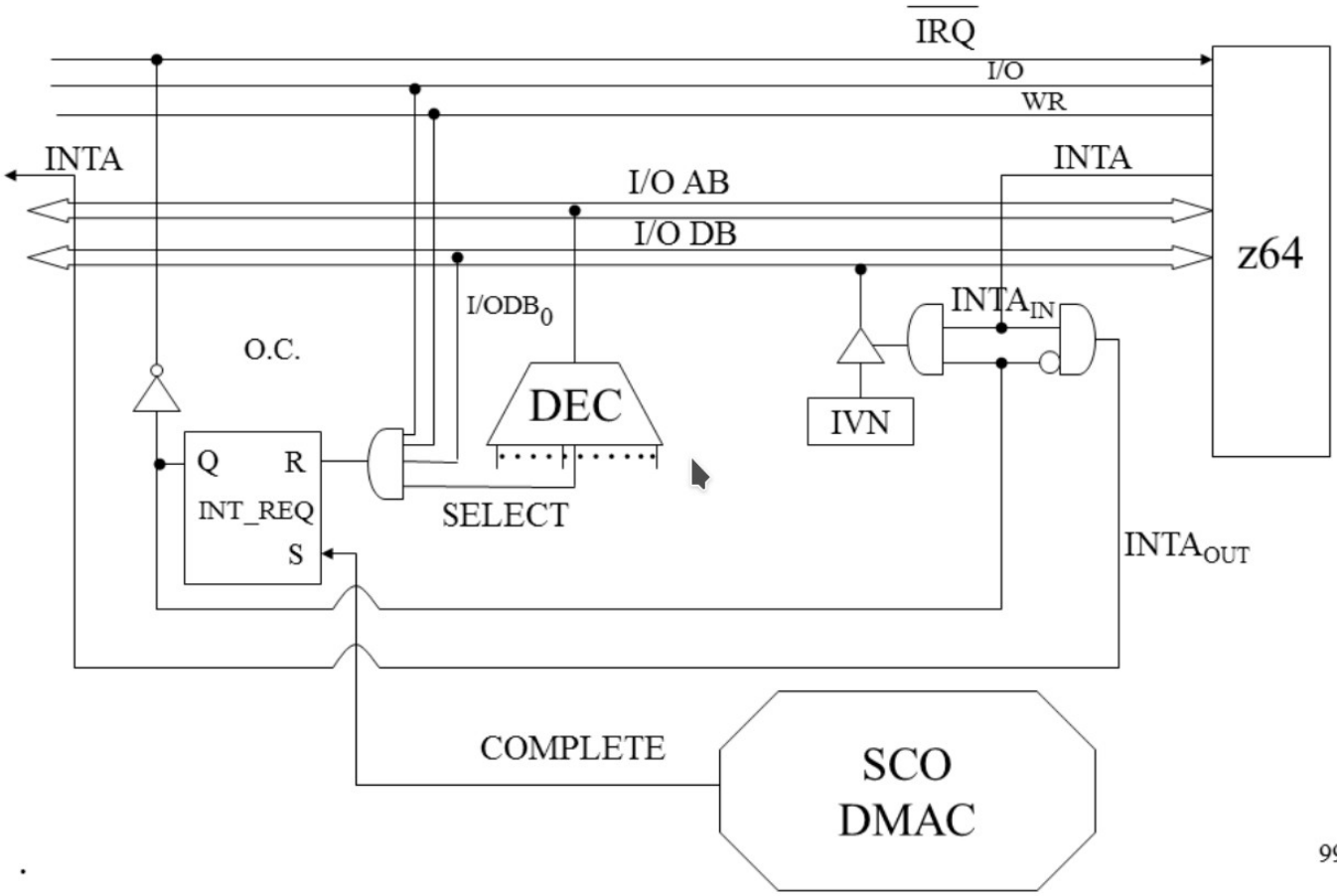
Viene presentata di seguito la parte dell'interfaccia che include il CAR, il WC, il flip/flop necessario per la richiesta del rilascio del bus (MBR) e il flip/flop di stato, necessario per permettere al processore di avvertire il SCO del DMAC che sono stati programmati i registri di interfaccia. Inoltre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



Viene presentata la parte relativa alla gestione delle interruzioni. Viene presentata la parte dell'interfaccia che include i flip/flop chen 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.



Viene presentata la parte relativa alla gestione delle interruzioni.



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.

ESEMPIO Inizializzazione DMAC

```
movw $WC. %dx                 # inizializza WC a 100
movw $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
```

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.

Uscite del SCO (segnali di controllo)

Mbi (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 (I/O)

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.

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.

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.

Ingressi del SCO (variabili di condizione)

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.

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.

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.

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).

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 laCPU 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) → IR;         /* trasferimento dell'istruzione da eseguire sull'IR */
RIP +8 → RIP;       /* predisposizione esecuzione istruzione successiva */
```

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).

Memory Endianness—Ordine dei Byte

• L'ordine dei byte descrive la modalità utilizzata dai calcolatori per immagazzinare in memoria dati di dimensione superiore al byte
• La differenza dei sistemi è data dall'ordine con il quale i byte costituenti il dato da immagazzinare vengono memorizzati:
◦ **litte-endian**: memorizzazione che inizia dal byte meno significativo per finire col più significativo (usato dai processori Intel)
◦ **big-endian**: memorizzazione che inizia dal byte più significativo per finire col meno significativo (Network-byte order)
◦ **middle-endian**: ordine dei byte né crescente né decrescente (es: 3412, 2143)
• La differenza si rispecchia nel network-byte order vs host order

Istruzioni di controllo hardware

Tipo	Mnemonico	Operandi	O S Z P C	Descrizione
1	`hlt`	–	- - - - -	Mette la CPU in modalità di basso consumo energetico, finché non viene ricevuta l'interruzione successiva
2	`nop`	–	- - - - -	Nessuna operazione
3	`int`	–	- - - - -	Chiama esplicitamente un gestore di interruzioni

Istruzioni di movimento dei dati

Tipo	Mnemonico	Operandi	O S Z P C	Descrizione
0	`mov`	B, E	- - - - -	Fa una copia di `B` in `E`
1	`movsX`	E, G	- - - - -	Fa una copia di `E` in `G` con estensione del segno
2	`movzX`	E, G	- - - - -	Fa una copia di `E` in `G` con estensione dello zero
3	`lea`	E, G	- - - - -	Valuta la modalità di indirizzamento, salva il risultato in `G`
4	`push`	E	- - - - -	Copia il contenuto di `E` sulla cima dello stack
5	`pop`	E	- - - - -	Copia il contenuto della cima dello stack in `E`
6	`pushf`	–	- - - - -	Copia sulla cima dello stack il registro FLAGS
7	`popf`	–	- - - - -	Copia nel registro FLAGS il contenuto della cima dello stack
8	`movs`	–	- - - - -	Esegue una copia memoria-memoria
9	`stos`	–	- - - - -	Imposta una regione di memoria ad un dato valore

Istruzioni logico/aritmetiche

Tipo	Mnemonico	Operandi	O S Z P C	Descrizione
0	add	B, E	⇕ ⇕ ⇕ ⇕ ⇕	Memorizza in E il risultato di E + B
1	sub	B, E	⇕ ⇕ ⇕ ⇕ ⇕	Memorizza in E il risultato di E - B
2	adc	B, E	⇕ ⇕ ⇕ ⇕ ⇕	Memorizza in D il risultato di E + B + CF
3	sbb	B, E	⇕ ⇕ ⇕ ⇕ ⇕	Memorizza in D il risultato di E - (B + neg(CF))
4	cmp	B, E	⇕ ⇕ ⇕ ⇕ ⇕	Confronta i valori di B ed E calcolando E - B, il risultato viene poi scartato
4	test	B, E	⇕ ⇕ ⇕ ⇕ ⇕	Calcola l'and logico bit a bit di B ed E, il risultato viene poi scartato
5	neg	E	⇕ ⇕ ⇕ ⇕ ⇕	Rimpiazza il valore di E con il suo complemento a 2
6	and	B, E	0 ⇕ ⇕ ⇕ 0	Memorizza in E il risultato dell'and bit a bit tra B ed E
7	or	B, E	0 ⇕ ⇕ ⇕ 0	Memorizza in E il risultato dell'or bit a bit tra B ed E
8	xor	B, E	0 ⇕ ⇕ ⇕ 0	Memorizza in E il risultato dello xor bit a bit tra B edE
9	not	E	0 ⇕ ⇕ ⇕ 0	Rimpiazza il valore di E con il suo complemento a uno
10	bt	K, E	- - - - ⇕	Imposta CF al valore del K-simo bit di E (bit testing)

Istruzioni di rotazione e shift

Tipo	Mnemonico	Operandi	O S Z P C	Descrizione
0	sal	K, G	⇕ ⇕ ⇕ ⇕ ⇕	Moltiplica per 2, K volte
1	sal	G	⇕ ⇕ ⇕ ⇕ ⇕	Moltiplica per 2, RCX volte
0	shl	K, G	⇕ ⇕ ⇕ ⇕ ⇕	Moltiplica per 2, K volte
1	shl	G	⇕ ⇕ ⇕ ⇕ ⇕	Moltiplica per 2, RCX volte
2	sar	K, G	⇕ ⇕ ⇕ ⇕ ⇕	Dividi (con segno) per 2, K volte
3	sar	G	⇕ ⇕ ⇕ ⇕ ⇕	Dividi (con segno) per 2, RCX volte
4	shr	K, G	⇕ ⇕ ⇕ ⇕ ⇕	Dividi (senza segno) per 2, K volte
5	shr	G	⇕ ⇕ ⇕ ⇕ ⇕	Dividi (senza segno) per 2, RCX volte
6	rcl	K, G	⇕ - - - ⇕	Ruota a sinistra, K volte
7	rcl	G	⇕ - - - ⇕	Ruota a sinistra, RCX volte
8	rcr	K, G	⇕ - - - ⇕	Ruota a destra, K volte
9	rcr	G	⇕ - - - ⇕	Ruota a destra, RCX volte
10	rol	K, G	⇕ - - - ⇕	Ruota a sinistra, K volte
11	rol	G	⇕ - - - ⇕	Ruota a sinistra, RCX volte
12	ror	K, G	⇕ - - - ⇕	Ruota a destra, K volte
13	ror	G	⇕ - - - ⇕	Ruota a destra, RCX volte

Manipolazione dei bit di FLAGS

Tipo	Mnemonico	Operandi	O	S	Z	P	C	Descrizione
0	clc	–	-	-	-	-	0	Resetta CF
1	clp†	–	-	-	-	0	-	Resetta PF
2	clz†	–	-	-	0	-	-	Resetta ZF
3	cls†	–	-	0	-	-	-	Resetta SF
4	cli	–	-	-	-	-	-	Resetta IF
5	cld	–	-	-	-	-	-	Resetta DF
6	clo†	–	0	-	-	-	-	Resetta OF
7	stc	–	-	-	-	-	1	Imposta CF
8	stp†	–	-	-	-	1	-	Imposta PF
9	stz†	–	-	-	1	-	-	Imposta ZF
10	sts†	–	-	1	-	-	-	Imposta SF
11	sti	–	-	-	-	-	-	Imposta IF
12	std	–	-	-	-	-	-	Imposta DF
13	sto†	–	1	-	-	-	-	Imposta OF

Controllo del flusso di programma

Tipo	Mnemonico	Operandi	O	S	Z	P	C	Descrizione
0	jmp	M	-	-	-	-	-	Esegue un salto relativo
1	jmp	*G	-	-	-	-	-	Esegui un salto assoluto
2	call	M	-	-	-	-	-	Esegue una chiamata a subroutine relativa
3	call	*G	-	-	-	-	-	Esegue una chiamata a subroutine assoluta
4	ret	–	-	-	-	-	-	Ritorna da una subroutine
5	iret	–	⇕	⇕	⇕	⇕	⇕	Ritorna dal gestore di una interruzione

Controllo condizionale del flusso

Tipo	Mnemonico	Operandi	O	S	Z	P	C	Descrizione
0	jc	M	-	-	-	-	-	Salta a M se CF è impostato
1	jp	M	-	-	-	-	-	Salta a M se PF è impostato
2	jz	M	-	-	-	-	-	Salta a M se ZF è impostato
3	js	M	-	-	-	-	-	Salta a M se SF è impostato
4	jo	M	-	-	-	-	-	Salta a M se OF è impostato
5	jnc	M	-	-	-	-	-	Salta a M se CF non è impostato
6	jnp	M	-	-	-	-	-	Salta a M se PF non è impostato
7	jnz	M	-	-	-	-	-	Salta a M se ZF non è impostato
8	jns	M	-	-	-	-	-	Salta a M se SF non è impostato
9	jno	M	-	-	-	-	-	Salta a M se OF non è impostato

La fase di Fetch

• Ogni operazione incomincia con la fase di fetch
• Le microoperazioni associate alla fase di fetch sono:
  ◦ MAR ← RIP
  ◦ MDR ← (MAR); RIP ← RIP + 8
  ◦ IR ← MDR
• In questo modo, l'istruzione successiva viene caricata nel registro IR (così da poterla interpretare ed eseguire) e il valore di RIP viene incrementato (così da puntare alla prossima istruzione/dato)

Istruzioni di movimento dati

• Le microoperazioni associate al movimento dati dipendono dalla modalità di indirizzamento utilizzato
• Accedere in memoria utilizzando la modalità di indirizzamento dello z64 è un'attività costosa
• movq %rax, %rcx:
  ◦ MAR ← RIP
  ◦ MDR ← (MAR); RIP ← RIP + 8
  ◦ IR ← MDR
  ◦ TEMP2 ← RAX
  ◦ RCX ← TEMP2

Istruzioni Aritmetiche e Logiche

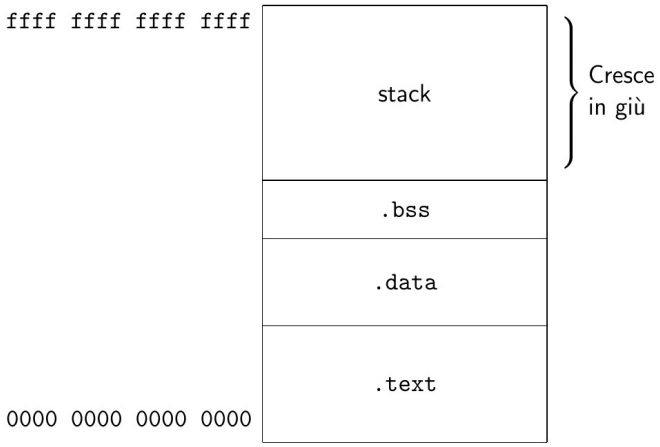
• L'esecuzione di una determinata operazione aritmetica o logica dipende dall'opcode passato alla ALU
• addw %ax, %cx:
  ◦ MAR ← RIP
  ◦ MDR ← (MAR); RIP ← RIP + 8
  ◦ IR ← MDR
  ◦ TEMP1 ← AX
  ◦ TEMP2 ← CX
  ◦ CX ← ALU OUT[ADD]

Istruzioni di salto condizionale

• jz displacement:
  ◦ MAR ← RIP
  ◦ MDR ← (MAR); RIP ← RIP + 8
  ◦ IR ← MDR
  ◦ IF FLAGS[ZF] == 1 THEN
  ◦ TEMP1 ← RIP
  ◦ TEMP2 ← IR[0:31]
  ◦ RIP ← ALU OUT[ADD]
  ◦ ENDIF

ORDINE DATI ASSEMBLY

b (byte) → w(word)→ l(longword) → q(quadword)



Scheletro di un programma assembly

.org [INDIRIZZO CARICAMENTO}

```
.data
        #Dichiarazione costanti e variabili globali
.text
        #Corpo del programma
        hlt     # Per arrestare l'esecuzione
```

Direttive Assembly

• **Labels**: mnemonico testuale definito dal programmatore ed associato all'indirizzo di ciò che la segue immediatamente

• **Location Counter**: identificato da ., viene valutato con il valore dell'indirizzo corrente.

- Può essere impostato esplicitamente per far "saltare" la generazione di indirizzi.
- Può essere usato per calcolare le dimensioni di strutture dati:

```
msg:
        .ascii "Hello, world!\\n"
        len = . - msg
```

• **.org address, fill**: metodo alternativo di impostare il location counter, impostando i byte a fill

• **.equ symbol, expression**: definisce una costante (non occupa memoria al momento della dichiarazione)

- Metodo alternativo: symbol = expression
- Lo stesso simbolo può essere ridefinito in più parti del codice
- Non si può usare il simbolo prima della sua definizione (one pass scan)

• **.byte expressions**: riserva memoria (di dimensione byte) per expressions:

```
var:    .byte 0
array:  .byte 0, 1, 2, 3, 4, 5
```

• **.word expressions**: riserva memoria (di dimensione word) per expressions

• **.long expressions**: riserva memoria (di dimensione longword) per expressions

• **.quad expressions**: riserva memoria (di dimensione quadword) per expressions

• **.ascii "string"**: riserva memoria per un vettore di caratteri e imposta il valore a string

• **.fill repeat, size, value**: riserva una regione di memoria composta da repeat celle di dimensione size impostate a value

- size e value sono opzionali (default: size = 1, value = 0).

• **.text**: tutto ciò che compare da qui in poi va nella sezione testo

• **.data**: tutto ciò che compare da qui in poi va nella sezione data

• **.comm symbol, length**: dichiara un'area di memoria con nome (symbol) di dimensione length nella sezione bss

• **.driver idn/.handler idn**: identifica l'inizio della routine di servizio associato al codice idn

Esempio di programma assebly

```
.org 0x800
.data
message:        .ascii   “Hello World!”
counter:        .byte 0
.text
main:
                movq $message, %rax
        .repeat:
                cmpb $0, (%rax)
                jz .end
                addb $1, counter
                addq $1, %rax
                jmp .repeat
        .end:
                hlt
```

If-Then-Else

```
int x = 1;
int val;

if (x == 2) {
        val = 2;
} else if (x == 1) {
        val = 1:
} else {
        val = 0;
}
```

Corrispondente Assembly

```
.org 0x800
.data
        x: .byte 1
        val: .byte 0
.text
        movb x, %al
        cmpb $2, %al            # Test prima condizione
        jnz .elseif
        movb $2, val            # blocco A
        jmp .endif              # Altrimenti andrei all’istruzione successiva
.elseif:
        cmpb $1, %al            # Test seconda condizione
        jnz .else
        movb $1, val            # blocco B
        jmp .endif
else:
        movb $0, val
.endif:
        hlt
```

Le variabili booleane?

• Il tipo booleano non esiste realmente nei processori, ma si utilizzano degli interi (tipicamente dei byte), e per convenzione si assume:

- false = 0
- true = !false

Esempio

```
boolean var = true;

if (var) {
        < blocco A >
} else {
        < blocco B >
}
```

Corrispettivo Assembly

```
.org 0x800
.data
        var: .byte 1                    # considerato come 'true'
.text
        cmpb $0, var
        jz .elsebranch
        nop                             # blocco A
        jmp .endif
.elsebranch:
        nop                             # blocco B
.endif: hlt
```

Cicli while

Un ciclo while ha due forme, a seconda di dove si effettua il controllo sulla condizione:

```
while (<condizione>) {
        <codice>
}
```

```
.test:
cmpb ….
jnz .skip
# <codice>
jmp .test
skip:
```

```
do {
<codice>
} while (<condizione>);
```

```
begin: # <codice>
cmpb ….
jz .begin
```

Cicli for

```
                                        movq $0, %rcx
int i;                                  .test:
for (i = 0; i < 3; i++) {               cmpq $3. %rcx
        <codice>                        jz .end
}                                       # <codice>
                                        addq $1, %rcx
                                        jmp .test
                                        .end:
```

Operazione bit a bit: forzatura

Per forzare dei bit ad un valore specifico, si usano ancora delle maschere di bit:
• Per forzare un bit a 1, si utilizza l'istruzione OR
• Per forzare un bit a 0, si utilizza l'istruzione AND
• Per invertire un bit, di utilizza l'istruzione XOR
Per forzare a 1 l'ultimo bit in R0:

```
orl $0x80000000, %eax
```

Per forzare a 0 l'ultimo bit in R0:

```
andl $0x7FFFFFFF, %eax
```

Per invertire l'ultimo bit in R0:

```
xorl $0x80000000, %eax
```

Operazione bit a bit: reset di un registro

L'istruzione xor permette di invertire un bit particolare in un registro
Ciò vale perché:
- $0 \oplus 1 = 1$
- $1 \oplus 1 = 0$

Questa stessa tecnica può essere utilizzata per azzerare un registro:

```
movq $0, %rax        è uguale a      xorq %rax, %rax       ma la seconda è più efficiente
```

Operazione bit a bit: estrazione

Supponiamo di avere un numero a 32 bit e di voler sapere qual è il valore dei tre bit meno significativi
Si può costruire una maschera di bit del tipo 00...00111, in cui gli ultimi tre bit sono impostati a 1
La maschera di bit 00...00111 corrisponde al valore decimale 7 e al valore esadecimale 0x7
Si può quindi eseguire un AND tra il dato e la maschera di bit

```
testl $7, %eax
```

Manipolazioni di vettori

Iterare su degli array è una delle operazioni più comuni, questa operazione può essere fatta in più modi
- **Modo** 1: si carica in un registro l'indirizzo del primo elemento e si incrementa manualmente il puntatore per scandire uno alla volta gli elementi. La fine del vettore viene individuata con un confronto con il contenuto di un secondo regsitro (numero di elementi) che viene decrementato

```
        movq $array, %rax          # array: indirizzo primo elemento del vettore
        movq $num, %rcx            # num: numero di elementi
.loop:
        movq (%rax), %rdx          # carica un dato
        # <processa i dati>
        addq $8, %rax              # Va all'elemento successivo
        subq $1, %rcx              # decrementa il contatore degli elementi
        jnz .loop
```

◦ **Modo** 2: come il modo 1, ma il primo elemento fuori dal vettore viene individuato dal suo indirizzo

```
.org 0x800
.data
        array: .long 1,2,3,4,5,6,7,8,9,10
        endarr: .long 0xdeadc0de
.text
        movq $array, %rax                    # array: indirizzo primo elemento del vettore
        movq $endarr, %rbx                   # endarr: primo indirizzo fuori dal vettore
.loop:
        addq (%rax), %rdx                    # processa un dato
        addq $4, %rax                        # vai all'elemento successivo
        cmpq %rax, %rbx
        jnz .loop
        hlt
```

Manipolazioni di vettori

Se non si ha a disposizione l'indirizzo del primo elemento fuori del vettore, ma si ha a disposizione il numero di elementi, si può calcolare l'indirizzo in questo modo:

```
        movq $array, %rax                    # array: indirizzo primo elemento del vettore
        movq $num, %rcx                      # num: numero di elementi
        shlq $3, %rcx # left shift per trasformare il numero in una taglia
                                             # Ogni elemento ha dimensione 8 byte ed uno shift
                                             # di  tre posizioni corrisponde a moltiplicare per 8
                                             # %rcx contiene quindi la dimensione (in byte)
                                             # dell'array. Sommando il valore dell'indirizzo di
                                             # base si ottiene il primo indirizzo fuori dall'array
        addq %rax, %rcx
```

e si può poi iterare utilizzando il modo 2

◦ **Modo** 3: in caso di tipi primitivi, si utilizza il registro indice per tenere traccia dell'elemento corrente

```
        xorq %rcx, %rcx                      # %rcx viene usato come indice
        movq $array, %rax                    # %rax viene usato come base
.loop:
        movq (%rax, %rcx, 8), %rbx           # sposta i dati dove serve
        # <processa i dati>
        addq $1, %rcx
        cmpq $num, %rcx
        jnz .loop
```

Modo 3 senza utilizzare la base:

```
        xorq %rcx, %rcx                      # %rcx viene usato come indice
.loop:
        movq array(, %rcx, 8), %rbx          # sposta i dati dove serve
        # <processa i dati>
        addq $1, %rcx
        cmpq $num, %rcx
        jnz .loop
```

Se dobbiamo saltare degli elementi in un vettore, si può decrementare il contatore degli elementi ancora da controllare. Nel controllo di terminazione del ciclo dobbiamo però accertarci che il puntatore non sia andato oltre l'ultimo elemento!

```
        movq $array, %rax
        movq $num, %rcx
.loop:
        addq (%rax), %rdx
        addq $16, %rax          # Salto un elemento!
        subq $2, %rcx           # Considero solo gli elmenti in posizione dispari
        js .skip                # %rcx puo' diventare negativo senza passare per 0!
        jnz .loop
.skip:
```

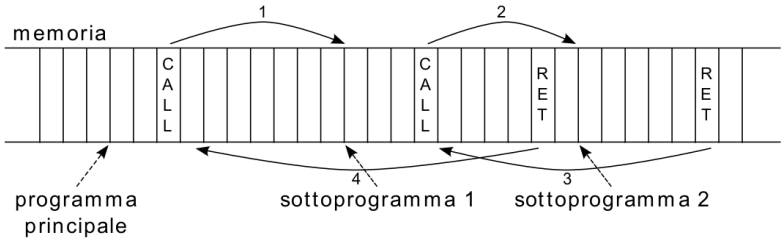
Moltiplicazione: soluzione naif

```
        .org 0x800
        .data
                op1: .long 6
                op2: .long 3
        .text
                xorl %eax, %eax         # Accumulatore del risultato
                movl op2, %ecx          # Registro utilizzato per la terminazione
        .ciclo:
                addl op1, %eax          # Un gran numero di accessi in memoria!
                jc .overflow            # Se incorriamo in overflow...
                subl $1, %ecx
                cmpl $0, %ecx
                jz .fine
                jmp .ciclo
        .overflow:
                movl $-1, %eax          # ...impostiamo il risultato a -1
        fine:
                hlt
```

Differenza tra jmp e call

A differenza della jmp, il microcodice della call memorizza l'indirizzo dell'istruzione successiva (indirizzo di ritorno) prima di aggiornare il valore contenuto nel registro RIP
L'unico posto in cui può memorizzare quest'informazione è la memoria, quest'area di memoria deve essere organizzata in modo tale da gestire correttamente lo scenario in cui un sottoprogramma chiama un altro sottoprogramma (subroutine annidate).

Subroutine annidate



Gestione dello stack

Lo stack è composto da quadword (non si può eseguire una push di un singolo byte)
La cima dello stack è individuata dall'indirizzo memorizzato in un registro specifico chiamato SP (Stack Pointer)
Modificare il valore di SP coincide con il perdere il riferimento alla cima dello stack, e quindi a tutto il suo contenuto.
Lo **stack** "**cresce**" se il valore contenuto in **SP diminuisce**, "**decresce**" se il valore contenuto in SP **cresce.**
Lo stack è posto in fondo alla memoria e cresce "all'indietro".

La finestra di stack

Se lo stack cresce in punti diversi del codice, gli spiazzamenti cambiano
La definizione della finestra di stack usa il base pointer RBP, il base pointer non cambia durante l'esecuzione di una subroutine.
Gli spiazzamenti (negativi!) non cambiano anche se lo stack cresce

```
pushq %rbp                  # Salva il frame pointer corrente
movq %rsp, %rbp             # Crea un nuovo frame, sull cima dello stack
subq $20, %rsp              # Alloca 20 byte per le variabili locali su stack
movl %eax, -4(%rbp)         # Salva %eax nella prima variabile locale
movl -8(%rbp), %ebx         # Carica %ebx dalla seconda variabile locale
```

Al termine della funzione, si esegue l'istruzione ret, questa recupera dallo stack l'indirizzo di ritorno
La finestra di stack va quindi "distrutta" prima di eseguire ret:

- Lo stack pointer deve puntare all'indirizzo di ritorno
- Il valore precedente di RBP va recuperato dallo stack

```
addq $20, %rsp              # Invalida lo spazio usato dalle variabili
popq %rbp                   # locali
                            # Rimette a posto RBP precedente
```

Convenzioni di chiamata

Affinché una subroutine chiamante possa correttamente dialogare con la subroutine chiamata, occorre mettersi d'accordo su come passare i parametri ed il valore di ritorno.
Le convenzioni principali permettono di passare i parametri tramite:

- Lo stack
- I registri

Generalmente il valore di ritorno viene passato in un registro perché la finestra di stack viene distrutta al termine della subroutine.
Se la subroutine chiamante vuole conservare il valore nel registro, deve memorizzarlo nello stack prima di eseguire la call.
I primi sei parametri di una subroutine vengono passati tramite registri:

- RDI, RSI, RDX, RCX, R8, R9
- R10 viene usato al posto di RCX per le system call

Se una subroutine accetta più di 6 parametri, si utilizza lo stack per quelli aggiuntivi.
I parametri vengono inseriti sullo stack in ordine inverso rispetto alla segnatura della funzione:

- La pulizia dello stack è a carico del chiamante (caller cleanup)

Il valore di ritorno viene memorizzato all'interno di RAX.
Alcuni registri devono essere salvati dalla funzione chiamata (callee-save):

- RBP, RBX, R12–R15

Istruzioni su stringhe

Lo z64 offre istruzioni che possono eseguire istruzioni su stringhe di dati:

- Copia da memoria a memoria di dati di dimensione arbitraria
- Imposta aree di memoria di dimensioni arbitrarie ad un valore prestabilito

Vari registri sono coinvolti da queste istruzioni:

- RCX: contatore del numero di operazioni elementari da eseguire
- RSI: indirizzo sorgente (per il movimento)
- RDI: indirizzo destinazione
- RAX: valore cui impostare la memoria (per l'impostazione)

Il direction flag (DF) identifica la direzione dell'operazione:

- DF = 0: l'operazione di copia si svolge in avanti
- DF = 1: l'operazione di copia si svolge all'indietro

Istruzioni su stringhe

movs: move data from string to string

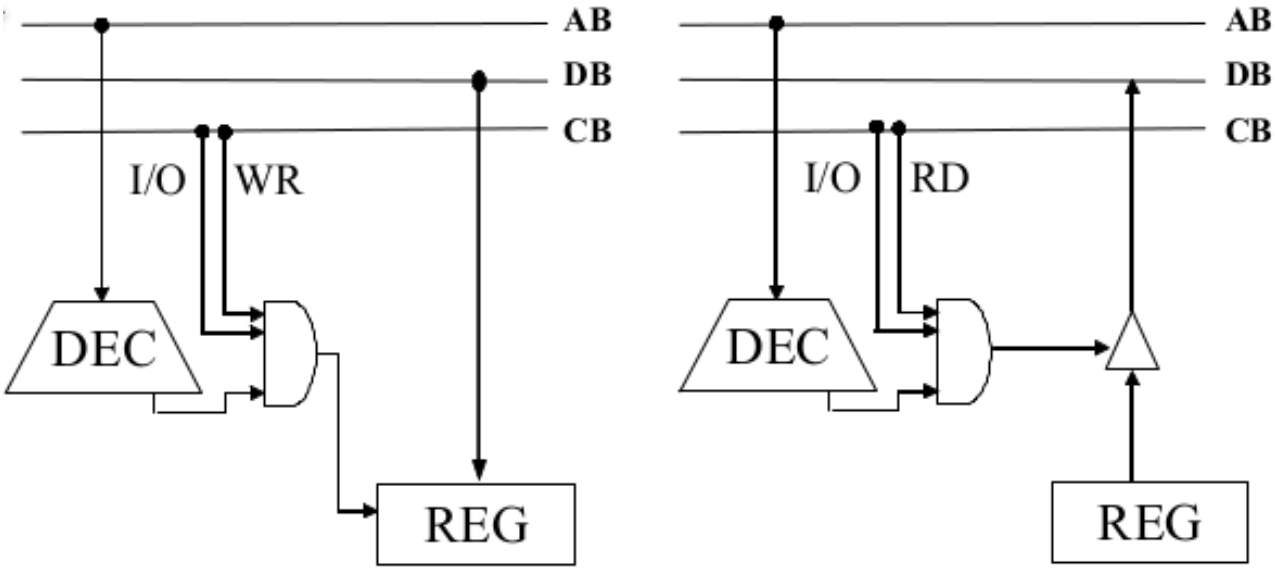
```
movq $S, %rsi
movq $D, %rdi
movq $size/8, %rcx
cld
movsq
```

stos: store string

```
movq $0x0, %rax
movq $D, %rdi
movq $size/8, %rcx
cld
stosq
```

Il microcodice dello z64 incrementa/decrementa i valori di RDI e RSI in funzione di DF.
RCX viene sempre decrementato. Se RCX è diverso da zero, RIP viene decrementato di 8.

In e out in azione

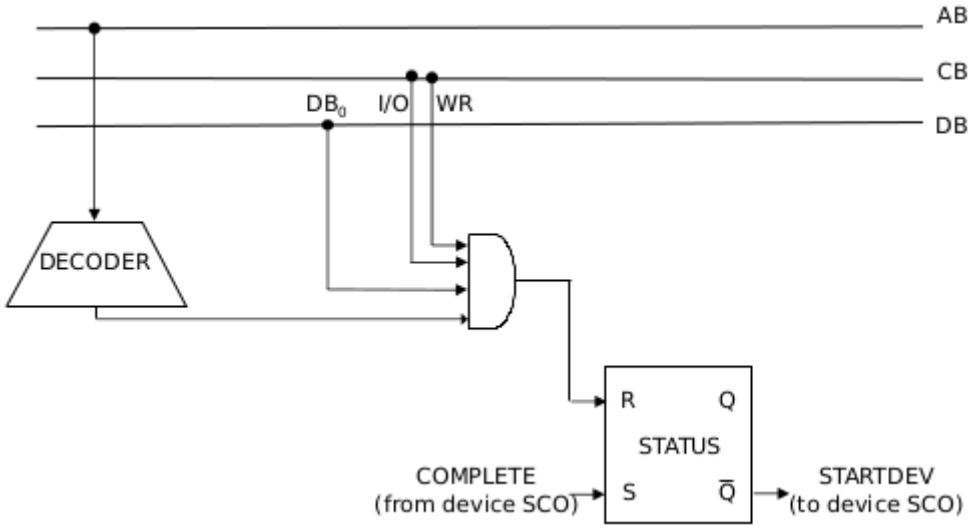


Interazione con le periferiche

Si può interagire con le periferiche in due modi:

- **Modo sincrono**: il codice del programma principale si mette in attesa della periferica
- **Modo asincrono**: la periferica informa il sistema che una qualche operazione è stata completata

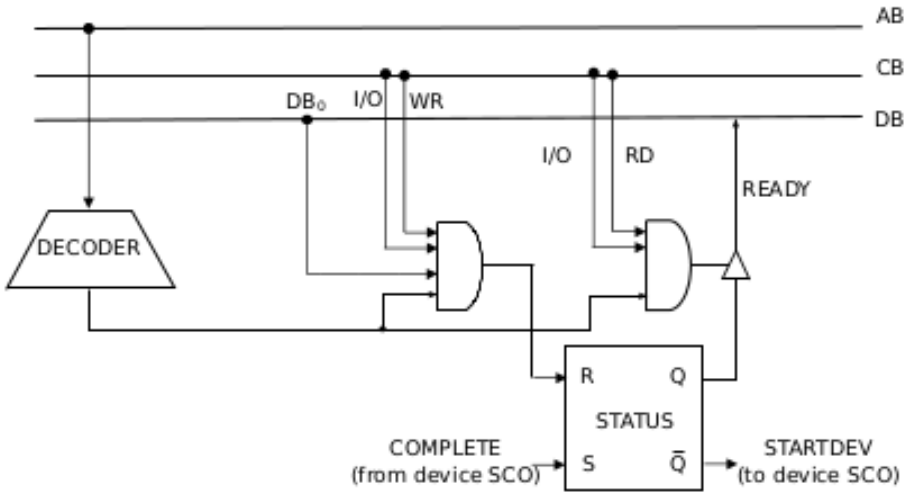
In entrambi i casi, la periferica deve poter memorizzare il suo stato.



Busy Waiting

Il Busy Waiting (attesa attiva) si basa su un ciclo in cui il processore chiede ripetutamente alla periferica se è pronta.

```
        # Avvia la periferica
        movw $STATUS, %dx
        movb $1, %al
        outb %al, %dx
        # Cicla in attesa che essa sia pronta
.bw:
        inb %dx, %al
        btb $0, %al
        jnc .bw
```

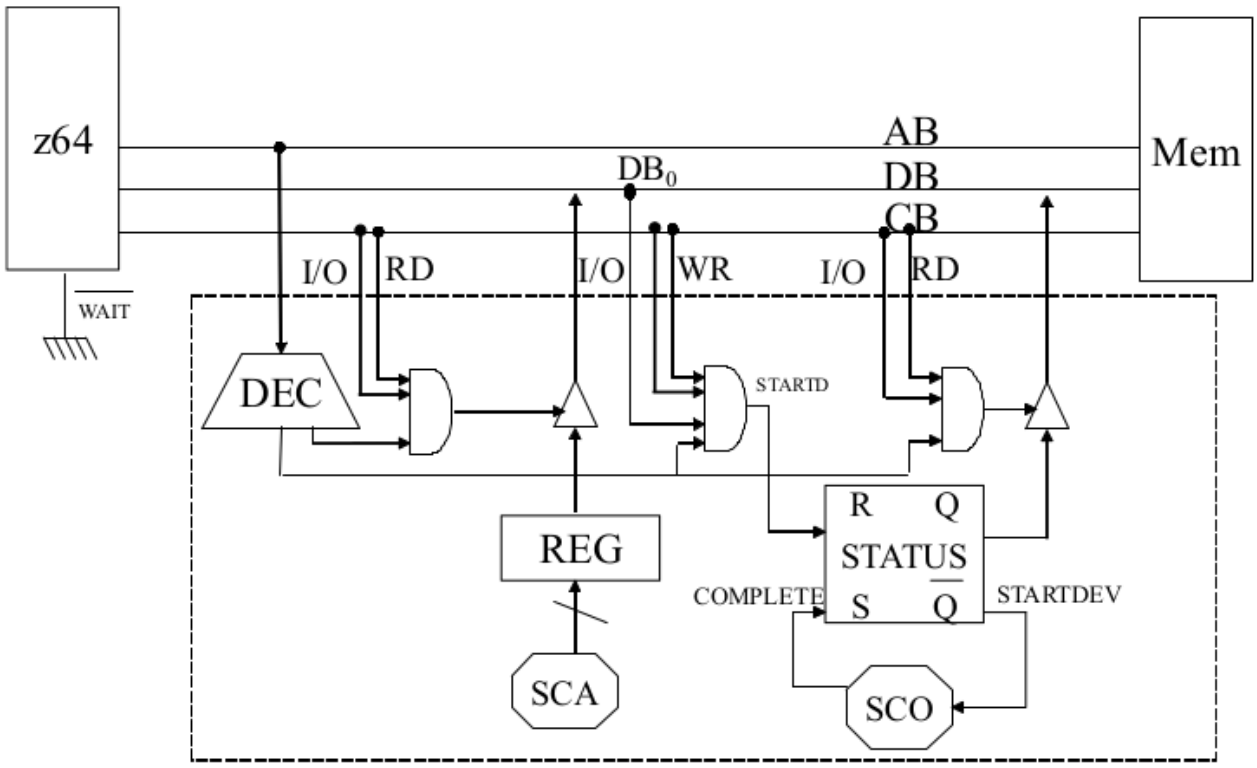


Polling

Il Polling è un’operazione simile al Busy Waiting, che coinvolge però più di una periferica connessa all’elaboratore.

```
.poll:
        movw $STATUS_DEV1, %dx
        inb %dx, %al
        btb $0, %al
        jc .dev1
        movw $STATUS_DEV2, %dx
        inb %dx, %al
        btb $0, %al
        jc .dev2
        # ...
        jmp .poll
```

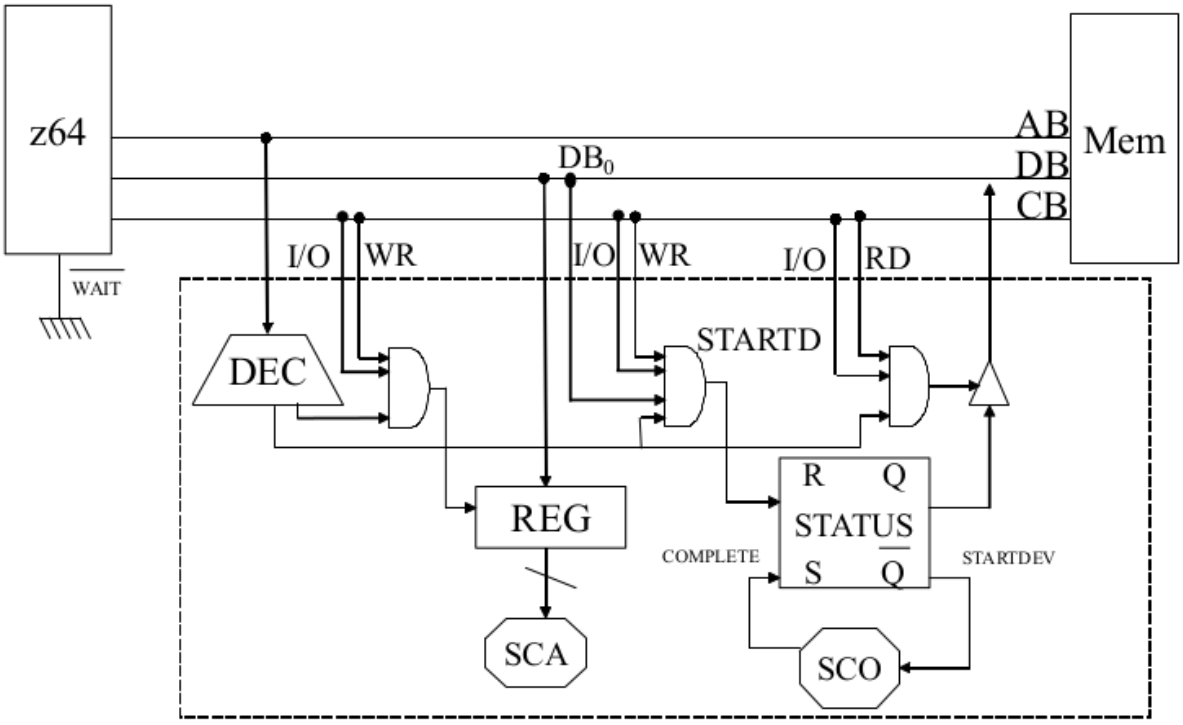
Interfaccia di Input



Relativo codie Assembly: I/O programmato (handshaking manuale)

```
        movw $status, %dx
.loop1:
        inb %dx, %al                    # 1. Aspetto che la periferica sia disponibile
        btb $0, %al
        jnc .loop1
        movb $1, %al                    # 2. Avvio la periferica così che possa produrre
        outb %al, %dx                   # informazioni
.loop2:                                 # 3. Aspetto che la periferica completi la sua unità di lavoro
        inb %dx, %al                    # 4. Acquisisco dalla periferica il risultato dell'operazione
        btb $0, %al
        jnc .loop2                      # Il primo ciclo di busy waiting può non essere necessario!
        movw $device_reg, %dx
        inl %dx, %eax
```

Interfaccia di Output



Relativo codie Assembly: I/O programmato (handshaking manuale)

```
        movw $status, %dx
.loop1:
        inb %dx, %al            # 1. Aspetto che la periferica sia disponibile
        btb $0, %al
        jnc .loop1              # 2. Scrivo nel registro di interfaccia con la periferica il
        movw $device_reg, %dx   # dato che voglio produrre in output
        movl $DATO, %eax
        outl %eax, %dx
        movw $status, %dx
        movb $1, %al            # 3. Avvio la periferica, per avvertirla che ha un nuovo
        outb %al, %dx           # dato da processare
.loop2:
        inb %dx, %al            # 4. Attendo che la periferica finisca di produrre in
        btb $0, %al             # output il dato
        jnc .loop2
```

Esecuzione Asincrona: le interruzioni

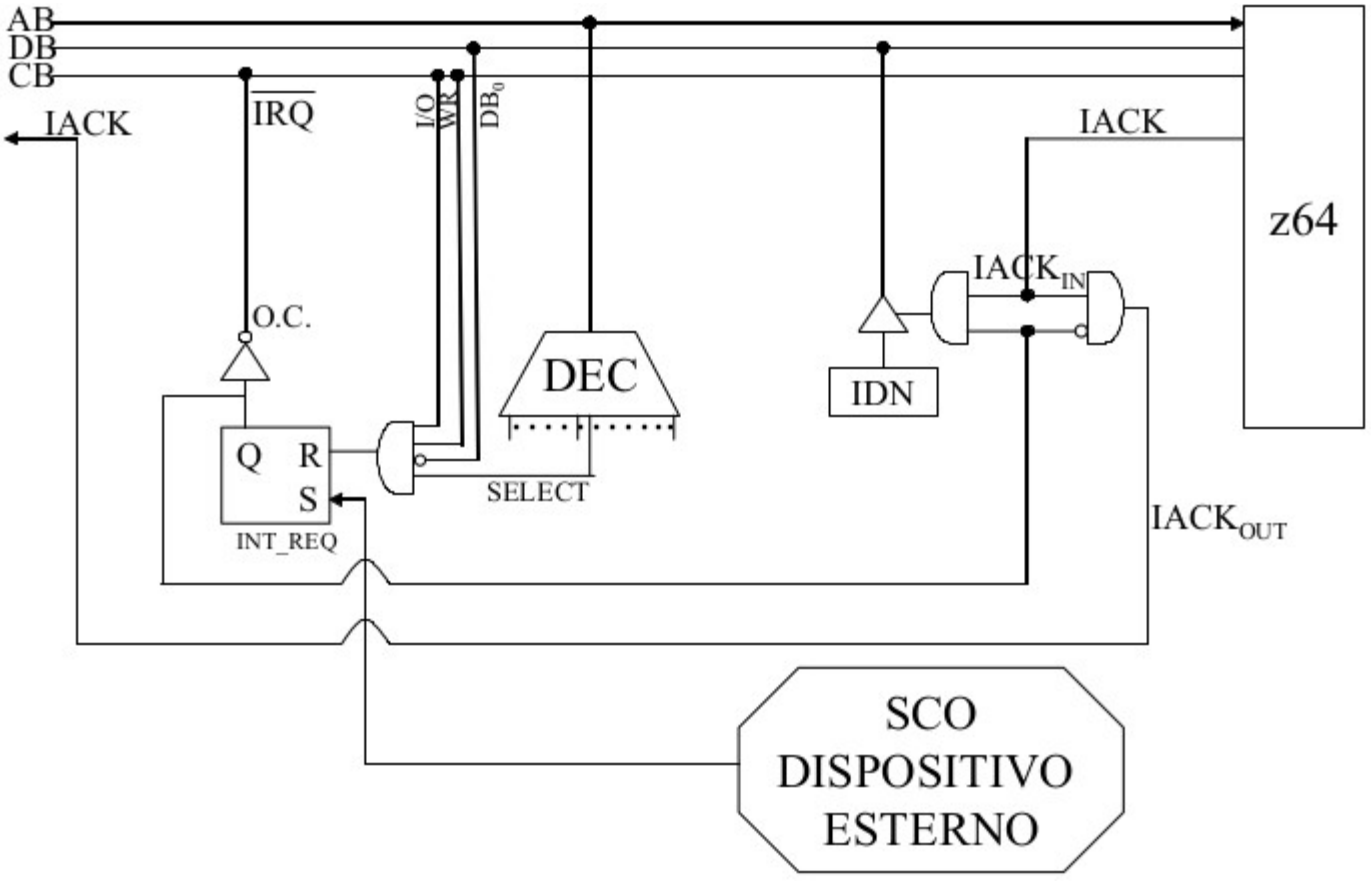
Nell'esecuzione asincrona, il processore programma la periferica, la avvia, e prosegue nella sua normale esecuzione.
L'intervallo di tempo in cui la periferica porta a termine la sua unità di lavoro può essere utilizzata dal processore per svolgere altri compiti
La periferica porta a termine la sua unità di lavoro ed al termine informerà il processore, interrompendone il normale flusso d'esecuzione.

Problemi:
1. Quando si verifica un'interruzione, occorre evitare che si verifichino interferenze indesiderate con il programma in esecuzione.
2. Una CPU può dialogare con diverse periferiche, ciascuna delle quali deve essere gestita tramite routine specifiche (driver)
3. Si debbono poter gestire richieste concorrenti di interruzione, oppure richieste di interruzione che giungono mentre è già in esecuzione un driver in risposta ad un'altra interruzione.

Soluzioni:
1. Salvataggio del contesto d'esecuzione
2. Identificazione dell'origine dell'interruzione
3. Definizione della gerarchia di priorità

Passi per la gestione di un'interruzione
1. Salvataggio dello stato del processo in esecuzione
2. Identificazione del programma di servizio relativo alla periferica che ha generato l'interruzione (driver)
3. Esecuzione del programma di servizio
4. Ripresa delle attività lasciate in sospeso (ripristino dello stato del processore precedente)

Interrupt Request e Interrupt Acknowledge



Cambio di contesto

Il contesto d'esecuzione di un processo è costituito da:

- **Registro RIP**: contiene l'indirizzo dell'istruzione dalla quale si dovrà riprendere l'esecuzione una volta terminata la gestione dell'interrupt.
- **Registro FLAGS**: alcuni bit di condizione potrebbero non essere ancora stati controllati dal processo.
- Altri registri (p. es., TEMP1, TEMP2), per supportare la ripresa dell'esecuzione di operazioni.

logico/aritmetiche. La gestione al termine del ciclo istruzione previene la necessità di salvarne il contenuto

Quando viene generata un'interruzione avviene una commutazione dal contesto del processo interrotto a quello del driver.

Analogamente il contesto del processo interrotto deve essere ripristinato una volta conclusa l'esecuzione del driver

È necessario assicurarsi che non si verifichino altre interruzioni durante le operazioni di cambio di contesto

- Potrebbero altrimenti verificarsi delle incongruenze tra lo stato originale del processo e quello ripristinato

Al termine dell'esecuzione di un'istruzione, il segnale **IRQ** assume il valore 1 e il flip-flop I viene impostato a 0 via firmware, inoltre lo z64 provvede a salvare nello stack i registri **FLAGS** e **RIP**.

Infine, in RIP viene caricato l'indirizzo della routine di servizio (driver) della periferica che ha generato la richiesta di interruzione.

Identificazione del driver (IDT)

L'identificazione del driver da attivare in risposta all'interruzione si basa su un numero univoco (IDN, Interrupt Descriptor Number) comunicato dalla periferica al processore, che identifica un elemento all'interno dell'Interrupt Descriptor Table (IDT),

Gestione di un'interruzione

```
FLAGS[I] ← 0; TEMP1 ← RSP
ALU ← 8; RSP ← ALU OUT[SUB]
MAR ← RSP
MDR ← RIP
(MAR) ← MDR
TEMP1 ← RSP
ALU ← 8; RSP ← ALU OUT[SUB]
MDR ← FLAGS
MAR ← RSP
(MAR) ← MDR
IACK IN
IACK IN; MDR ← IDN
TEMP2 ← MDR
MAR ← SHIFTER OUT[SX, 3]          # soltanto 256 driver differenti!
MDR ← (MAR)
RIP ← MDR
```

Ritorno da un’interruzione

MAR ← RSP
MDR ← (MAR)
FLAGS ← MDR
TEMP1 ← RSP
ALU ← 8; RSP ← ALU OUT[ADD]
MAR ← RSP
MDR ← (MAR)
RIP ← MDR
TEMP1 ← RSP
ALU ← 8; RSP ← ALU OUT[ADD]
FLAGS[I] ← 1

Interrupt nel mondo reale

Tutti i sistemi operativi dividono la gestione degli interrupt in due parti:

- First-Level Interrupt Handler, o **top half**
- Second-Level Interrupt Handler, o **bottom half**

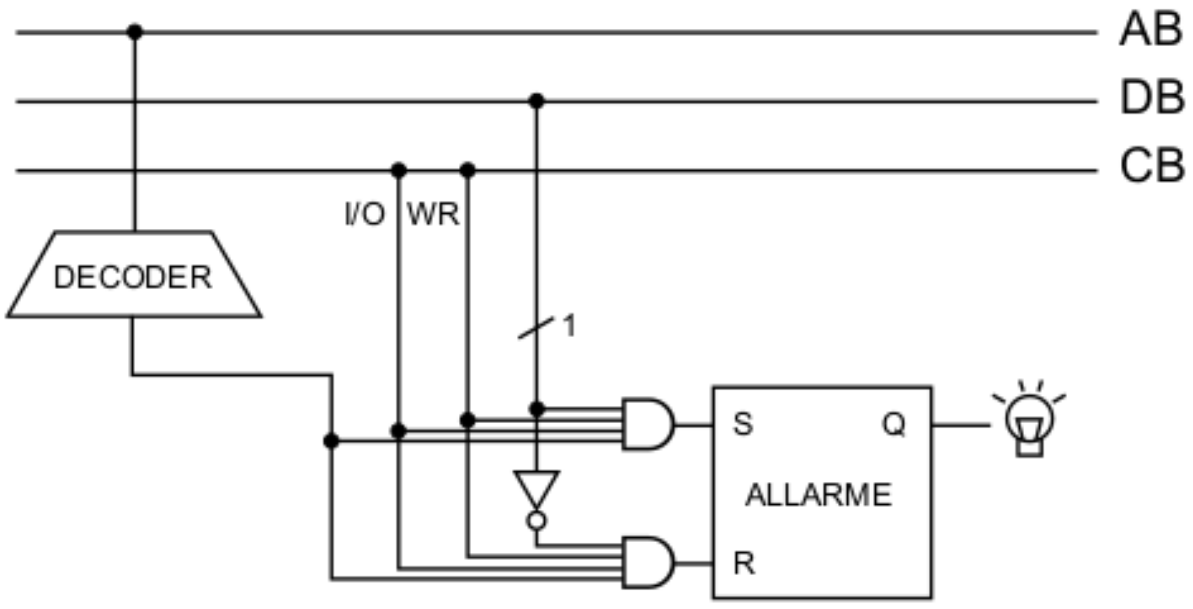
Una **top half** implementa una gestione minimale delle interruzioni:

- Viene effettuato un cambio di contesto (con mascheramento delle interruzioni)
- Il codice della top half viene caricato ed eseguito
- La top half serve velocemente la richiesta di interruzione, o memorizza informazioni critiche disponibili soltanto al momento dell’interrupt e schedula l’esecuzione di una bottom half non appena possibile
- L’esecuzione di una top half blocca temporaneamente l’esecuzione di tutti gli altri processi del sistema: si cerca di ridurre al minimo il tempo d’esecuzione di una top half

Una **bottom half** è molto più simile ad un normale processo

- Viene mandata in esecuzione (dal Sistema Operativo) non appena c’è disponibilità di tempo di CPU
- L’esecuzione dei compiti assegnati alla bottom half può avere una durata non minimale

Interfaccia ALLARME



Direct Memory Access Controller (DMAC)

Per effettuare il trasferimento di un file dalla memoria ad un dispositivo di Ingresso/Uscita o viceversa è necessario definire da processore:

- la direzione del trasferimento (verso o dalla memoria – IN/OUT)
- l'indirizzo iniziale della memoria (nel DMAC c'è un registro contatore CAR – Current Address Register
- il tipo di formato dei dati (B, W, L)
- la lunghezza del file (numero di dati) (nel DMAC c'è un registro contatore WC – Word Counter)
- l'identificativo della periferica di I/O interessata al trasferimento (se più di una periferica è presente)

Esistono delle istruzioni ottimizzate per programmare il DMAC di sistema:

- insX
- outsX

Sono istruzioni di tipo stringa, pertanto:

- RCX contiene il numero di blocchi di dati da leggere/scrivere
- RDI contiene l'indirizzo destinazione (per la insX)
- RSI contiene l'indirizzo sorgente (per la outsX)
- DF indica la direzione

insX: leggi 10 byte da DEV

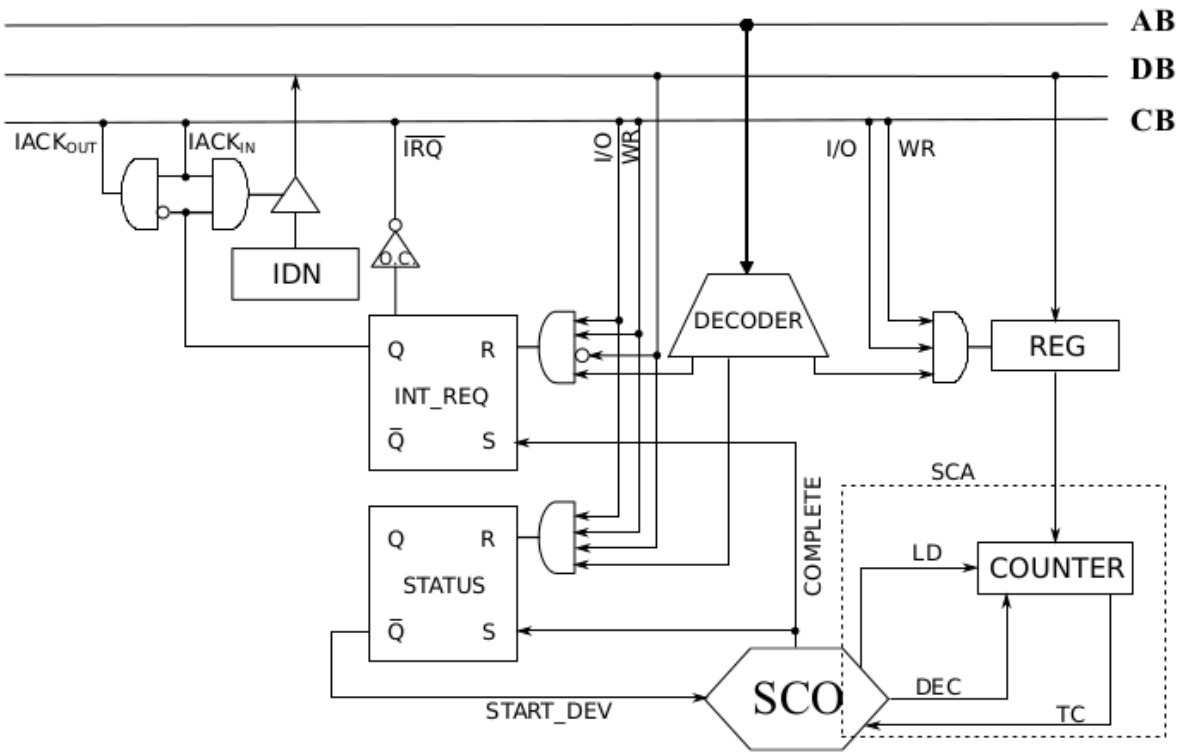
```
movq $10, %rcx
movq $dest, %rdi
movq $dev_mem, %dx
cld      movwdxmovbaloutbaldxmovbalmovwdxoutbaldx
                                        cld
insb
```

insX: scrivi 10 byte su DEV

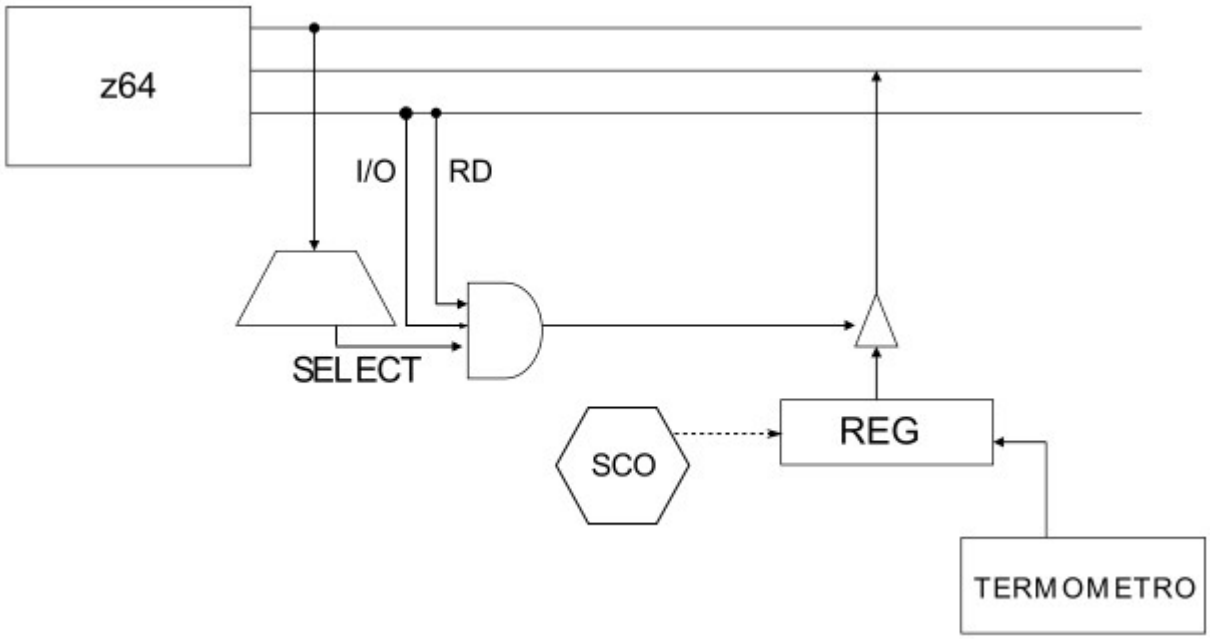
```
movq $10, %rcx
movq $dest, %rsi
movq $dev_mem, %dx


outsb
```

Interfaccia TIMER



Interfaccia DEV TEMPERATURA



Condizioni di Salto

Con segno

Test	CC	es.
==	e	je
!=	ne	jne
<	l	jl
≤	le	jle
>	g	jg
≥	ge	jge

Senza segno

Test	CC	es.
<	b	jb
≤	be	jbe
>	a	ja
≥	ae	jae

Relativo codice Assembly

```
        .org 0xBBBB
        .data
                messaggio: .fill 512, 1                 # 512 byte
                .equ intervallo, 10                     # intervallo (in ms) tra due interruzioni
                .equ TEMPERATURA_REG, 0x00
                .equ TIMER_REG, 0x01
                .equ TIMER_INT_REQ, 0x02
                .equ TIMER_STATUS, 0x03
                .equ VIDEO, 0x04

        .org 0x800
        .text
                movw $TIMER_REG, %dx                    # Configura TIMER
                movw $intervallo, %ax
                outw %ax, %dx
                movw $TIMER_STATUS, %dx                 # Avvia TIMER
                movb $1, %al
                outb %al, %dx
                sti                                     # Abilita la ricezione delle interruzioni
                hlt

        .driver 0                                       # Driver di TIMER
                movw $TEMPERATURA_REG, %dx
                inb %dx, %al
                cmpb $40, %al                           # al > 40 se al - 40 > 0
                js .minore
                movw $VIDEO, %dx                        # Programma il DMAC di sistema
                movq $messaggio, %rsi
                movl $512/4, %ecx
                cld
                outsl                                   # Copia 512 byte, una longword per volta,
                                                        # verso VIDEO
.minore:
                movw $TIMER_INT_REQ, %dx                # Cancella la causa di interruzione e riavvia
                movb $0, %al                            # TIMER
                outb %al, %dx
                movb $1, %al
                movw $TIMER_STATUS, %dx
                outb %al, %dx
                iret
```