1 Input – Output

Ogni computer deve essere in grado di collegarsi con ciò che ha intorno. Ciò avviene attraverso i dispositivi di I/O.
In questo capitolo si tratta in generale del funzionamento dei dispositivi di I/O, delle diverse modalità con cui l'I/O è organizzato e di cosa del fare il software per controllare e gestire l'Input/Output.

1.1 Cenni all'organizzazione hardware di memoria ed I/O

1.1.1 Decodifica degli indirizzi

Come già visto precedentemente ogni locazione di memoria ha il suo indirizzo ed ogni indirizzo corrisponde ad una sola locazione di memoria. Precedentemente non avevamo indagato su come la memoria sia organizzata fisicamente, considerandola solamente un insieme di molte locazioni contigue.
La memoria è divisa fra diversi circuiti integrati, dato che un singolo chip non è abbastanza capiente da contenere tutta la memoria principale che serve. Vediamo come è possibile considerare la memoria contigua anche se risiede fisicamente su circuiti diversi.
Il meccanismo è semplice ed è illustrato nella Figura 1. L'indirizzo che viene emesso dalla CPU sull'address bus va in parte al chip di memoria ed in parte ad un altro circuito. Infatti la parte più alta dell'indirizzo viene mandata ad un "decodificatore" che decide se l'indirizzo emesso fa parte del chip o meno.
Il decodificatore d'indirizzo riceve in ingresso un indirizzo ed alza la sua linea di uscita se la locazione che risponde a quell'indirizzo è all'interno del circuito che esso controlla, altrimenti abbassa la stessa linea. La linea di uscita del decodificatore va nell'ingresso "chip select" del circuito (l'ingresso chip select è detto anche "chip enable").
Se un circuito integrato ha CS alto è collegato ai bus, se invece CS è basso il chip è come se non esistesse (alta impedenza), è disabilitato.

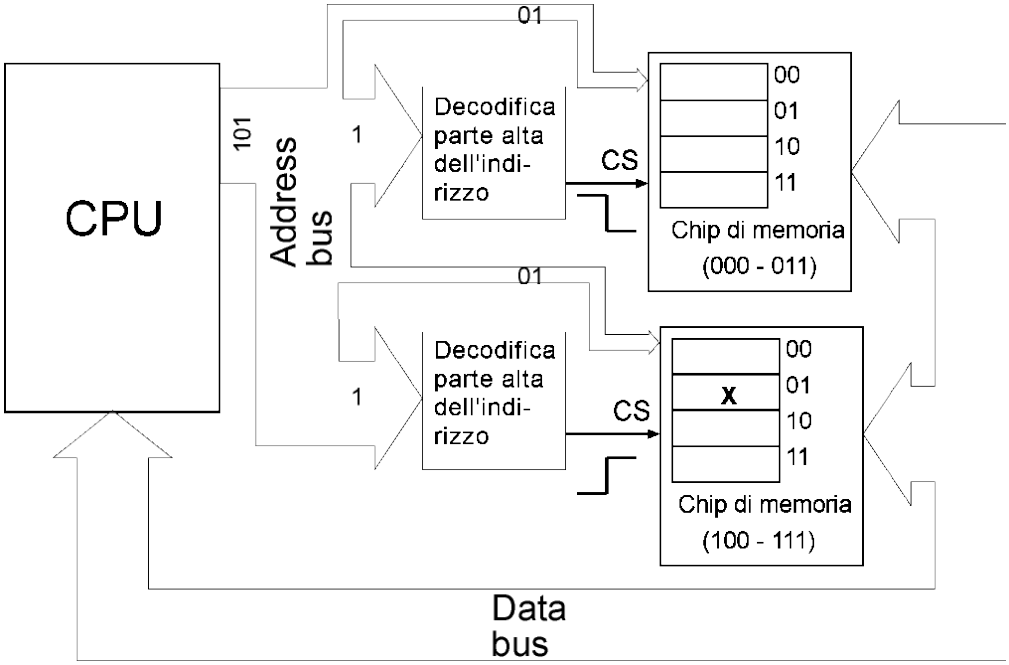


Figura 1: decodifica degli indirizzi

La parte bassa dell'indirizzo viene invece mandata fino al chip di memoria, ed è responsabile della selezione di una delle tante locazioni comprese in quel chip.
La Figura 1 è generale nel disegno ma particolare nell'esempio mostrato. Contiene infatti un esempio numerico semplicissimo. La memoria dell'esempio è grande la bellezza di 8 byte :-(, che corrispondono ad un indirizzo di 3 bit. Il chip più in alto risponde agli indirizzi da 000b a 011b, mentre quello in basso risponde da 100b a 111b. Di conseguenza se la CPU emette sull'address bus il numero 101 il chip in alto deve essere disabilitato, l'altro abilitato. Perciò il decoder in alto dà un CS basso, il contrario per l'altro. La parte bassa dell'indirizzo (01) viene portata ad entrambi i chip. Ma il chip in alto è "spento", per cui non è neppure in grado di leggere quello 01, mentre l'altro è attivo sul bus. Quindi lo 01 seleziona la locazione 01 all'interno del chip in basso, cioè quella contrassegnata con una X nel disegno. Essa perciò corrisponde all'indirizzo 101 della memoria "globale".
Naturalmente gli schemi di decodifica di un computer vero avranno molti più bit in gioco, dato che la memoria è molto più di 8 byte, ma saranno basati sullo stesso principio. Saranno quindi solo più complicati, ma non più "difficili". Non indagheremo ulteriormente su come può essere fatto un decodificatore di indirizzi.

In sintesi: dato un indirizzo sull'address bus l'hardware provvede automaticamente, tramite sistemi di decodifica degli indirizzi, a far funzionare i circuiti giusti ed a disabilitare quelli non coinvolti nell'operazione.

1.1.2 Circuiti integrati di I/O

Precedentemente avevamo visto come ogni come la CPU possa comunicare solo con dispositivi che si comportano in qualche modo come una memoria. Quindi OGNI dispositivo, per comunicare con una CPU, deve poter leggere un indirizzo sull'address bus e saper trasferire sul data bus le informazioni che la CPU richiede, effettuando una lettura od una scrittura, in base a ciò che la CPU comunica con le linee del control "bus".
Questo significa che non ci devono essere differenze elettriche sostanziali fra una normale memoria ed un circuito di I/O (si legga ai-o), oppure che si deve interporre un particolare circuito di "**interfaccia**", che da un lato si comporta come una memoria e può comunicare con la CPU e dall'altro lato comunica con il dispositivo con le tecniche che sono più comode.
Chiameremo "circuito di I/O" questa interfaccia fra CPU e dispositivi esterni.
Dentro un circuito di I/O possono essere presenti diversi registri. La CPU "vede" i registri di un dispositivo di I/O ad indirizzi specifici, stabiliti per via hardware da un sistema di decodifica degli indirizzi. Questo circuito abilita il circuito di I/O nello stesso modo con cui abilita un qualsiasi chip di memoria.
All'interno del circuito di I/O i registri sono selezionabili sequenzialmente, per cui la CPU li può vedere ad un primo indirizzo e a quelli successivi. Il primo degli indirizzi a cui è visibile un chip di I/O è detto **indirizzo "base"**.
I vari registri del chip saranno perciò all'indirizzo BASE, agli indirizzi BASE + 1, BASE + 2, .. e così via. Non tutti i registri di un circuito di I/O possono avere accesso diretto al data bus, quindi, a differenza delle memorie, un circuito di I/O può possedere risorse "inaccessibili" da parte della CPU.
I circuiti di I/O possono svolgere funzioni anche molto complicate, in alcuni casi sono vere e proprie CPU programmabili che svolgono un compito complesso scaricando la CPU dall'onere del controllo di dispositivi. Quando un circuito di I/O ha capacità di elaborazione autonoma viene detto "processore di I/O", o anche "controller" o dispositivo "intelligente", anche se senz'altro non è più intelligente del mostro del dottor Frankenstein @ :-).
Quando il circuito di I/O è semplice e non fa altro che costituire un "ponte" fra CPU e dispositivi esterni semplici il nome più comune che viene usato è "**port di I/O**". I dati che vengono scambiati fra il dispositivo esterno e la CPU passeranno per i registri del port di I/O, per poi finire nella memoria principale del computer.
Dunque memoria ed I/O si dividono gli indirizzi: si dice che lo "spazio di indirizzi" è condiviso fra memoria ed I/O. Il progettista dell'hardware del computer decide di riservare, nello spazio di indirizzi della memoria, alcuni indirizzi ai dispositivi di I/O. Il progettista del software dovrà conoscere quegli indirizzi e sapere che non sono destinati a normali operazioni di memoria, ma che scrivendo e leggendo in quegli indirizzi si interverrà in qualche modo nei dispositivi collegati al computer.
La sintesi di quanto detto è presentata nel seguente disegno:

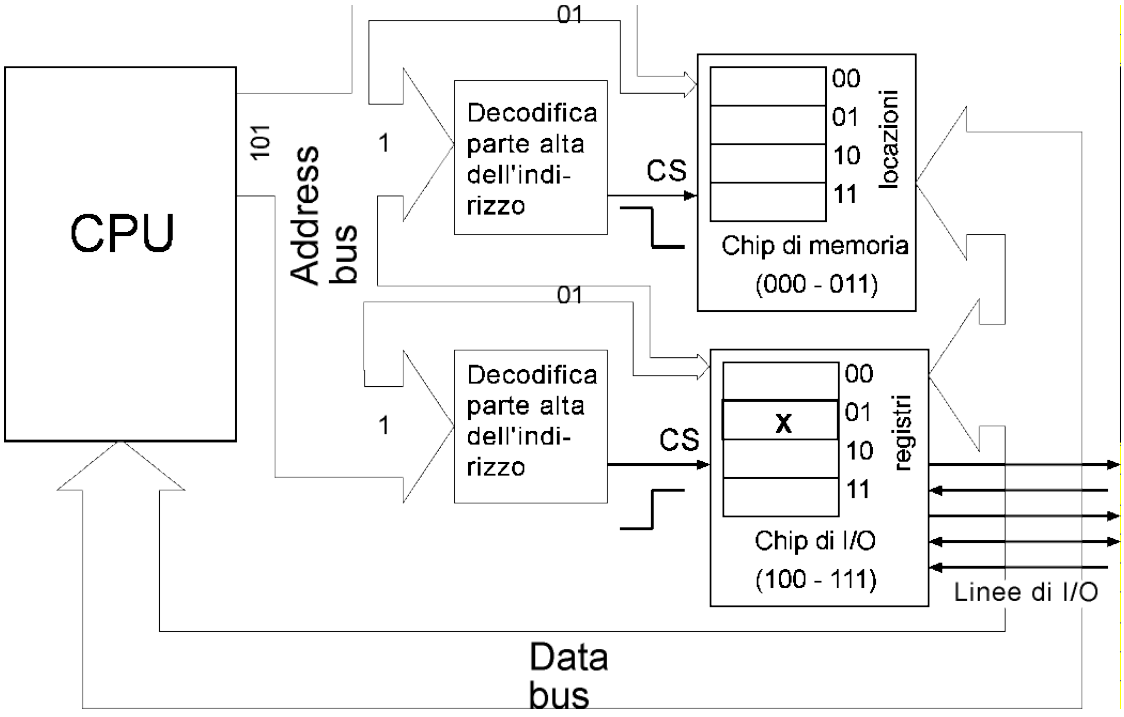
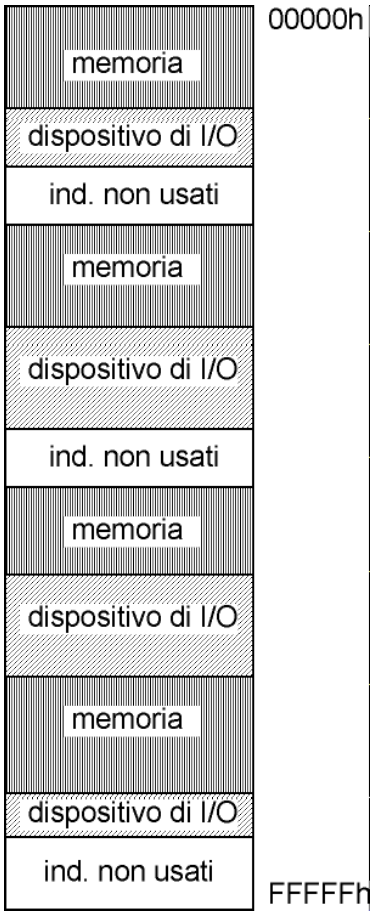


Figura 2: "Memoria ed I/O nello stesso spazio di indirizzi"

La Figura 2 è del tutto simile alla precedente, solo che il chip in basso non è una memoria ma un circuito di I/O, per cui non comunica solo con la CPU, ma anche con altri dispositivi, attraverso le sue linee di I/O.
Come si vede dal disegno le linee di I/O possono essere sempre in lettura, sempre in scrittura ed anche bidirezionali. Scrivendo nei registri del dispositivo si potrà cambiare lo stato delle linee di I/O esterne, come indica la crocetta in Figura 2 che significa che quel port è indirizzato.
Si noti che in Figura 2 il chip di memoria e quello di I/O hanno indirizzi DIVERSI. In Figura 2 l'indirizzo BASE del circuito di I/O è 100b.

1.1.3 I/O isolato



Usare lo stesso spazio di indirizzi per memoria ed I/O può costituire un problema. Infatti se non c'è differenza elettrica nelle operazioni in memoria od in I/O, dal punto di vista funzionale la differenza c'è, eccome!
La memoria RAM è sempre tanta: più ce n'è meglio è. Oggi si usano centinaia di MByte, in qualche caso anche alcuni GByte.
Al contrario i dispositivi di I/O occupano sempre pochi indirizzi. Come avremo modo di vedere studiando alcuni specifici circuiti, i produttori spesso ricorrono a trucchi stranissimi per limitare il numero degli indirizzi dei circuiti di I/O. Gli indirizzi di solito sono quattro e nel caso dei circuiti più complicati quasi mai superano i 16. Tutti i dispositivi di I/O di un computer, messi insieme, non superano pochi kByte di indirizzi di I/O. Le esigenze dei due tipi di circuiti e del software che li usa sono perciò molto diverse.
Nella Figura 3, del tutto ipotetica, si può vedere come potrebbe essere organizzato lo spazio di indirizzi di un computer.
Si può notare che la presenza dei dispositivi "frammenta" la memoria in tanti pezzi più piccoli cui non si può fare accesso in blocco. Ricordiamo infatti che lo scrivere all'indirizzo di un dispositivo pensando che sia una normale locazione di memoria può far dei danni all'hardware e comunque non fa funzionare il programma che usa quegli indirizzi. Così se c'è bisogno di un blocco di memoria contiguo in cui c'è in mezzo un dispositivo, esso non si potrà allocare. Il software, in particolare il Sistema Operativo che deve decidere come allocare la memoria, deve tener traccia degli indirizzi di tutti i dispositivi e deve fare in modo che quegli indirizzi non siano allocati come memoria normale.

Figura 3: frammentazione della memoria per effetto dei circuiti di I/O

Per semplificare la vita del software, ed anche per altre ragioni su cui ora sorvoliamo, una soluzione, usata anche nell'8086, è inventare due spazi di indirizzi separati per la memoria ordinaria e l'I/O. Si può pensare cioè di avere un altro "tipo" di indirizzi che servano solo per l'I/O. La situazione diverrebbe quindi quella prospettata dalla Figura 4.

!!!! da mettere !!!!

Figura 4: separazione degli spazi di indirizzi di memoria e I/O[1]

Questo sembrerebbe implicare il "raddoppio" dei bus, cioè aggiungere al di indirizzi per la memoria un altro bus speciale per l'I/O. Questa soluzione è costosa, anche se non è del tutto da scartare e viene anche usata in computer specializzati.
Per evitare il costo del raddoppio del bus degli indirizzi si usa un meccanismo hardware molto più semplice di quanto si pensi a prima vista, che è illustrato nella figura seguente:

1Il disegno è del tutto "fuori scala", dato che i dispositivi di I/O prendono solo pochi Byte. Si noti anche che in fondo allo spazio degli indirizzi di memoria ci può essere una zona di indirizzi non usati. Infatti in un computer non tutti gli indirizzi possibili sono presenti fisicamente, per esempio un 386, che può indirizzare 4 GByte non è detto che abbia tutta quella memoria RAM. I numeri di indirizzo indicati nel disegno sono relativi ad un 8086

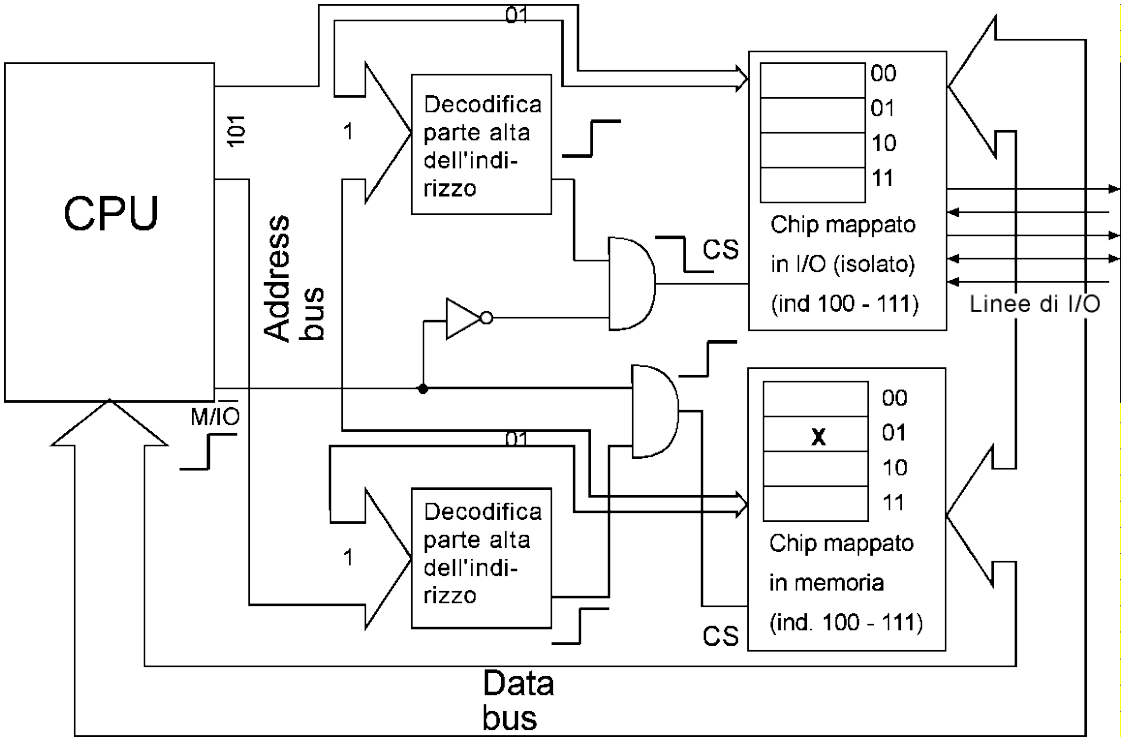


Figura 5: Lo stesso indirizzo per due chip diversi?

Si noti che in questo disegno il chip di memoria e quello di I/O hanno indirizzi UGUALI.
Dunque lo stesso indirizzo sull'address bus corrisponde a due dispositivi diversi? Dato che questo non può essere, come risponde solo il dispositivo giusto, mentre l'altro rimane "spento"?

Per risolvere il problema si aggiunge una nuova linea del control "bus", indicata come M/$\overline{IO}$ (Memory or I/O negato, è il nome di un piedino dell'8086). Questa linea indica il "tipo" di 'indirizzo che è stato emesso sull'address bus. Se la linea è alta è un indirizzo di memoria, se è bassa è di I/0.
Di conseguenza due AND ed un NOT bastano per attivare il circuito giusto. Nel disegno i

circuiti di I/O saranno attivati solo se M/$\overline{IO}$ è basso, il contrario per quelli di memoria.

Ma come fa la CPU a decidere se l'indirizzo da usare è di memoria o di I/O, cioè ad alzare od abbassare la linea M/$\overline{IO}$?
La risposta è semplice: ci sono istruzioni per la memoria e istruzioni per l'I/O. Mentre le istruzioni normali, come quelle che abbiamo visto finora, operano sempre e solo nello spazio di indirizzi della memoria, alcune istruzioni speciali operano solo su indirizzi di I/O.
Con termine bruttino, ma comodo, di dice che un circuito di I/O può essere "**mappato in memoria**" (memory mapped I/O) quando è visibile ad indirizzi di memoria ordinaria, mentre è "mappato in I/O" od è in "**I/O isolato**" (isolated I/O) quando è visibile ad indirizzi dello spazio separato dell'I/O. Nelle CPU che hanno uno spazio di I/O isolato la maggior parte dei dispositivi viene mappata in I/O.
Naturalmente nulla impedisce al progettista dell'hardware di decidere che alcuni dispositivi di I/O dovranno essere mappati in memoria e non in I/O isolato. Questo succede anche nei computer esistenti, per esempio nei PC la memoria video, che è parte della scheda video e quindi a rigor di logica è un insieme di port di I/O, viene mappata in memoria.

1.1.4 I/O isolato nell'8086: istruzioni IN e OUT

La strada della separazione fra indirizzi di memoria ed indirizzi di I/O è stata seguita anche con le CPU della famiglia X86, che dispongono di due istruzioni per operare in I/O isolato: la IN a la OUT.
Le istruzioni 8086 che usano indirizzi di I/O sono solo la IN e la OUT. Tutte le altre istruzioni dell'8086 utilizzano indirizzi della memoria ordinaria. In pratica quando un 8086 deve eseguire un'istruzione IN o OUT metterà a zero la linea di

controllo M/$\overline{IO}$, mentre la terrà a 1 per tutte le altre istruzioni, che devono accedere alla memoria ordinaria.
Lo spazio di indirizzi di I/O è di soli 64 kByte, perciò l'indirizzo che si deve usare nelle IN e nelle OUT è di soli 16 bit e non è segmentato. Lo spazio di 64 kByte per dispositivi è comunque molto ampio, tanto è vero che le CPU X86 più recenti usano ancora indirizzi di I/O a 16 bit, anche se il loro indirizzo di memoria ne ha ora 32 (4 GByte).
Le istruzioni IN e OUT hanno diversi limiti, anche nelle CPU successive all'8086.

Istruzioni di Input e Output

La forma più tipica delle istruzioni di I/O dell'8086 è la seguente:

```
IN AL , DX     ; input data from I/O port
OUT DX , AL; output data to I/O port
oppure
IN AL , <Indirizzo immediato minore di 256>
OUT <Indirizzo immediato minore di 256> , AL
```

Il dato di queste istruzioni deve essere sempre messo in AL, mentre l'indirizzo va in DX se si vuole usare l'indirizzamento indiretto. Si può usare anche un indirizzamento immediato, ma il valore dell'indirizzo di I/O deve essere minore di 256 (numero di 8 bit).

IN (**in**put) è l'istruzione di ingresso: il dato viene trasferito dal port di I/O il cui indirizzo è specificato in DX al registro AL.
OUT (**out**put) è l'istruzione di uscita: il dato contenuto in AL viene copiato al port di indirizzo indicato da DX.

Si noti che in queste istruzioni gli unici registri che si possono usare sono AL e DX! DX contiene l'indirizzo a 16 bit non segmentato e AL il dato da leggere o scrivere.
E' anche possibile fare trasferimenti di più di un byte, utilizzando un registro dati di 16 o 32 bit, in questo caso la CPU accede in sequenza a 2 o 4 port consecutivi durante la stessa istruzione:

```
IN AX , DX ; legge in AL il contenuto del port di indirizzo DX
           ; ed in AH il contenuto del port di indirizzo DX + 1
OUT DX , AX; scrive agli indirizzi DX e DX + 1

IN EAX , DX; solo 386 e>
           ; analoga alla IN precedente, ma con 4 byte trasferiti
OUT DX , EAX   ; anche questa a 4 byte
```

Esempi:

```
IN AL, 35    ; è possibile perché 35 < 256
OUT 135, AX  ; è possibile perché 135 < 256
IN AX, 35    ; accede al port 35 ed al 36, il contenuto va in AX
OUT 35, EAX  ; giusto se la CPU è >= 386; scrive EAX nei port 35, 36 37 e 38
IN EAX, DX   ;
IN BL, DX    ; SBAGLIATO, BL non può essere la destinazione
OUT SI, AL   ; SBAGLIATO, SI non può contenere l'indirizzo
OUT 1350, AX ; SBAGLIATO, perché 1350 > 256
```

I/O di stringa

Nelle CPU della famiglia X86 dall'80186 in avanti è stata introdotta la forma "di stringa" delle istruzioni di I/O. Queste istruzioni trasferiscono dati fra memoria ed I/O, aumentando automaticamente l'indirizzo in cui si lavora.

```
INSB        ; !! 186 > !!
INSW        ; !! 186 > !!
INSD        ; !! 386 > !!
```
; IN String of **B**ytes, **W**ord or **D**ouble word

```
OUTSB       ; !! 186 > !!
OUTSW       ; !! 186 > !!
OUTSD       ; !! 386 > !!
```
; OUT String of **B**ytes

Funzionamento:
```
INSB ; ES:DI <- (contenuto del port di I/O di indirizzo DX)
     ; DI <- DI + 1, se DF = 0 | DI <- DI – 1, se DF = 1
```

INSB legge un byte dal port di indirizzo DX, mappato in I/O, e lo scrive in ES:DI, poi incrementa o decrementa SI (NON modifica DX!). SI viene incrementato se il direction flag = 0 (DF=0) decrementato altrimenti).
```
OUTSB   ; ES:DI <- (contenuto del port di I/O di indirizzo DX)
        ; DI <- DI + 1, se DF = 0 | DI <- DI – 1, se DF = 1
```

OUTSB carica un byte da [DS:SI] (o da [DS:ESI] per 386>) e lo scrive al port di I/O specificato in DX, poi "sposta" il valore di DI.

INSW, OUTSW, INSD e OUTSD fanno lo stesso con parallelismo maggiore.

Dal 386 in poi si può usare come puntatore alla memoria il registro EDI.
Come le altre istruzioni di stringa, trattate precedentemente, queste ammettono il prefisso REP per la loro ripetizione automatica. In questo caso operano su blocchi di port, in input od in output, della dimensione di CX Byte.
Queste istruzioni hanno, dal 386 in poi, anche una forma con due operandi, esempio:

```
INSB <registro o locazione di memoria da 8 bit>, DX
INSW <registro o locazione di memoria da 16 bit>, DX
INSD <registro o locazione di memoria da 32 bit>, DX

OUTSB DX, <registro o locazione di memoria da 8 bit>
OUTSB DX, <registro o locazione di memoria da 16 bit>
OUTSD DX, <registro o locazione di memoria da 32 bit>
```

I/O digitali

Gli I/O digitali sono il tipo di Input - Output più semplice che si possa immaginare, ed anche il più comune. Si tratta dell'ingresso o dell'uscita di un singolo bit, che indica lo stato attuale di un sistema "binario".
La seguente tabella mostra alcuni esempi di dispositivi che generano un input digitale.

Sensore	Caratteristiche
sensore di prossimità, microswitch	Indica la "vicinanza" di un oggetto al sensore. Usato come finecorsa.
sensore di livello	Indica il superamento di una soglia nel livello del materiale contenuto in un serbatoio

Tabella 1: alcuni dispositivi per ingressi digitali

Un output digitale è collegato ad un attuatore digitale, un dispositivo del tipo "acceso o spento" che interviene sul mondo esterno. Nella tabella alcuni esempi:

Attuatore	Caratteristiche
elettrovalvola ON/OFF	Valvola per fluidi accesa o spenta.
Relais	Commutatore che, comandato dal computer, può accendere o spegnere carichi elettrici considerevoli, come p.es. un resistore scaldante
motore elettrico ON/OFF	Attuatore digitale di posizione. Movimenta un oggetto.

Tabella 2: alcuni dispositivi per uscite digitali

Alcuni I/O digitali, peraltro più rari, possono avere parallelismo maggiore di uno. Come esempio si può considerare il motore elettrico passo - passo (step motor), che permette la utilizzazione diretta di numeri binari che, trasmessi al motore, lo fanno muovere del numero di "passi" indicato. In questo caso più di una linea elettrica trasporta la stessa informazione.

Per collegarsi con sensori ed attuatori digitali i computer fanno uso di circuiti detti "port" di I/O digitale.
Molto spesso i port di I/O sono registri di 8 bit nei quali ciascuno dei bit è collegato ad un sensore o ad un attuatore.
In molti casi i port di I/O possono essere utilizzati sempre solo in ingresso o solo in uscita, per limitazioni dovute alla loro elettronica. Peraltro esistono port di I/O "programmabili", che possono essere usati ingresso oppure in uscita sotto controllo del software. In casi rari, come quello illustrato nella , ogni singolo bit del port può essere programmato in Input o in Output.
Di solito invece tutti gli 8 bit di ogni port possono essere o in ingresso o in uscita (come accade p.es. con il circuito integrato 8255, che vedremo in seguito).
Di solito per lo stato di un I/O digitale basta fare l'operazione di I/O e non è necessario sapere il momento in cui il dato è valido. In questo caso si dice che l'operazione non è "sincronizzata".
I dispositivi tipici di I/O digitale presentano il loro stato sulle linee di I/O corrispondenti e non hanno bisogno di altro. Quando il software ha bisogno di sapere in che condizione è un sensore di livello non farà altro che leggere il corrispondente port di I/O.
In diversi casi, spesso quando l'I/O è su diverse linee contemporaneamente ("parallelo"), i dispositivi di I/O devono poter comunicare che il dato che stanno fornendo è valido.
Il trasmittente ed il ricevente devono perciò "sincronizzarsi", in modo che la comunicazione sia efficace.

Sincronizzazione delle comunicazioni

..
Il meccanismo che viene usato per i due dispositivi viene detto "handshake", che significa stretta di mano. Il termine è significativo, perché dice che i due dispositivi si "mettono d'accordo" sul momento in cui deve avvenire lo scambio dei dati; quindi si "stringono la mano", come fanno le persone educate una volta che stabiliscono un accordo.
L'**handshake** è dunque il meccanismo attraverso il quale si possono sincronizzare due dispositivi, ossia il modo in cui essi possono stabilire il momento giusto per realizzare uno scambio di informazioni coordinato.

Il modo più semplice per fare un handshake è usare due linee elettriche, che chiameremo "**strobe**" e "**acknowledge**".
Il dispositivo che spedisce l'informazione è quello che sa quando essa è valida; esso deve comunicare all'altro che ci sono novità.
Per comunicare che un dato è pronto lo scrivente usa la linea "strobe", termine che "è un po' intraducibile", ma che in genere indica un segnale che stabilisce l'istante preciso in cui qualcosa ha luogo.
In questo caso lo strobe identifica il momento in cui i segnali sulle altre linee possono essere considerati validi. Perciò la linea di **strobe** significa "pronto" e viene modificata da chi scrive, per comunicare che c'è un nuovo dato disponibile.
Certe volte il segnale di strobe viene detto "request".
Tipicamente lo strobe è un segnale normalmente alto che viene abbassato quando è attivo. Il ricevente saprà che c'è un dato pronto quando vedrà abbassarsi la linea strobe. Allora provvederà a leggere il dato e a memorizzarlo od usarlo, poi risponderà utilizzando un'altra linea, detta "acknowledge".
La parola acknowledge significa "conferma" e dà bene l'idea su cosa sia.
La linea **acknowledge** significa "OK, ho letto" e dà l'autorizzazione a chi trasmette a riprendere la trasmissione. Infatti il trasmittente, quando vede abbassarsi acknowledge, sa che l'ultimo dato è stato letto, quindi può cominciare a spedire il prossimo.
Lo scambio di informazioni avviene quindi secondo la seguente sequenza:
1. Trasmettitore: scrive il dato sui fili corrispondenti
2. Trasmettitore: abbassa il livello della linea strobe (request)
3. Ricevitore: legge il dato dalle linee che lo trasportano e lo memorizza all'interno di un suo registro
4. Ricevitore: abbassa la linea di acknowledge

Esistono altri tipi di handshake, più completi e complessi, che possono sincronizzare sistemi che scambiano informazioni in entrambe le direzioni ed anche più di due sistemi che usano le stesse linee per i dati.
Comunque sia il funzionamento di principio degli schemi complicati parte sempre dal meccanismo Strobe – Acknowledge.

Vantaggio: il dispositivo esterno può funzionare alla sua velocità, mentre il chip di I/O comunica con la CPU come se fosse una memoria.

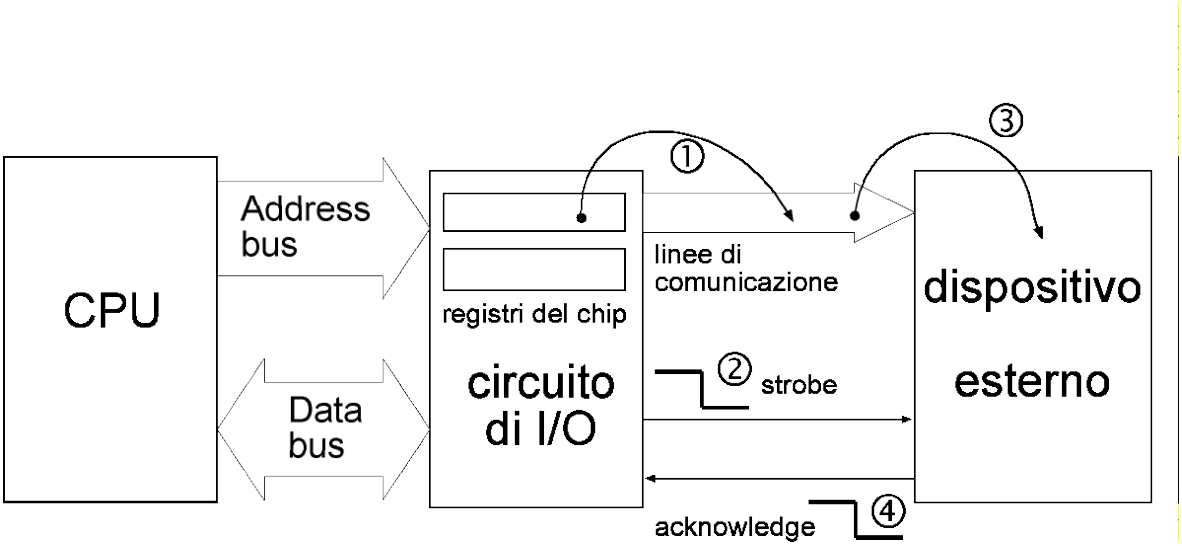


Figura 6: Trasferimento unidirezionale con handshake

Per un buon esempio pratico di come funziona un trasferimento con strobe e acknowledge si veda nel prosieguo la trattazione sul circuito integrato 8255.

Ingressi da I/O digitali
A meno che non si sia sicuri del funzionamento dell'hardware che si controlla è meglio fare sempre un mascheramento dei bit che non servono, prima di fare un confronto. In questo modo i bit che non servono hanno sempre un valore ben determinato ed il confronto funziona senz'altro. Infatti l'hardware di I/O potrebbe essere fatto in modo tale che i bit "non collegati" possano dare ingressi variabili e non prevedibili (basta che le relative tensioni siano "flottanti").

I/O analogici: convertitori A/D - D/A
Molti dei sensori ed attuatori digitali già descritti possono avere

Controller e processori di I/O
Controller della tastiera, dell'hard disk

Coprocessori
schede grafiche accelerate

Misurazione del tempo

Molto spesso nei programmi che fanno uso di dispositivi di I/O è necessario misurare intervalli di tempo.
In questo caso si hanno due alternative:
- il computer che si sta programmando dispone di un Sistema Operativo che ha una funzione che dà, in qualche forma e unità di misura, il tempo intercorso a partire da un dato evento.
- il computer non ha un sistema operativo, per cui ci si può affidare solo all'hardware. Questo può succedere quando si programmano i microcontrollori.

In alcuni casi può essere necessario scrivere programmi che usino direttamente l'hardware anche quando il S.O. mette a disposizione funzioni di più alto livello per la misura del tempo.
Il caso più semplice di misurazione del tempo è quando bisogna fare in modo che il programma attenda per intervalli di tempo precisi.
La prima tecnica che viene in mente è far perdere tempo al programma in un loop "inutile", che non ha altro senso che sprecare del tempo.

Loop di conteggi

La tecnica del loop di conteggi (loop delay = ritardo a loop) è semplice ma ha grandi limitazioni. Con un semplice loop di conteggi si può realizzare un ritardo:

```
    MOV CX, 6000
AttendoUnPo:
    LOOP AttendoUnPo
```

Questo loop realizza un ritardo, che può essere variato cambiando ciò che si scrive in CX inizialmente.
Il primo problema che si pone è che questo loop potrebbe essere percorso in troppo poco tempo: per fare 6000 cicli una CPU moderna può metterci davvero poco. Per cui si può fare in modo che l'esecuzione di un singolo loop prenda più tempo:

```
    MOV CX, 6000
AttendoUnPoDiPiu:
    MUL 0       ; seimila moltiplicazioni
    LOOP AttendoUnPoDiPiu
```

Se anche questo non basta si possono fare due loop uno dentro all'altro (notare che nel codice seguente si usa la LOOP per entrambi i cicli, con un semplice trucco):

```
    MOV CX, 6000
LoopRitardo:
    ; salvo il CX "esterno" perché devo usarne un altro "interno":
    PUSH CX
    MOV CX, Parametro    ; carico in CX un numero fisso, chiamato Parametro
GiraParametroVolte:
    MUL 0
    LOOP GiraParametroVolte
    ; riprendo il CX del loop esterno
    POP CX
    LOOP LoopRitardo
```

Una versione più completa potrebbe essere la seguente:

```
Ritardo PROC
    ; procedura che effettua un ritardo lungo a piacere.
    ; in BX viene passato il numero di centesimi di secondo
    ; della durata del ritardo
AltroCentesimo:
    ; si esegue un loop "nullo" per il numero di volte giusto
    ; per farlo durare un centesimo di secondo (NumeroMagico)
    MOV CX, NumeroMagico
GiraPerUnCentesimo: LOOP GiraPerUnCentesimo
    ; "loop fatta a mano" per evitare di usare lo stack:
    DEC BX
    JNZ AltroCentesimo
    RET
Ritardo ENDP
..
```

```
    ; per usarla:
    MOV BX, 300  ; ritardo di tre secondi
    CALL Ritardo
```

Il codice precedente può essere una buona base per un ritardo da realizzare velocemente e senza pensarci troppo.
Ma nella maggior parte dei casi un ritardo a loop di conteggio è del tutto inadeguato. Se si usa lo stesso software su computer diversi essi possono avere CPU diverse, che eseguono le istruzioni a velocità diverse. In questo caso il loop di conteggi andrebbe in qualche modo "tarato". Per esempio si potrebbe leggere l'orologio di sistema, fare il loop di ritardo, rileggere l'orologio di sistema per determinare quanto tempo è intercorso e cambiare il valore di "Parametro" nel codice precedente. In questo modo il loop di conteggi si adatta alla velocità della CPU utilizzata.
Anche adottando la "taratura" dei loop il programma può risultare inadeguato.
Infatti il loop di conteggi non è affidabile quando più di un programma gira "contemporaneamente" sulla stessa CPU, come avviene in tutti i Sistemi Operativi multitasking (vedi oltre in questo volume). In questo caso il programma in attesa su un loop di conteggio potrebbe essere sospeso ed al suo posto la CPU potrebbe mettersi ad eseguire un altro programma.
Dato che durante lo stesso loop, non si sa quante volte il programma verrà sospeso, il loop verrà eseguito ogni volta in tempi diversi.
Dunque la tecnica del loop di conteggi può andar bene solo quando la CPU è dedicata ad un solo programma, come nel caso del Sistema Operativo MSDOS o dei sistemi embedded senza Sistema Operativo.

Misurazione del tempo con contatori hardware

Visto il discorso fatto precedentemente si può concludere che per avere ritardi accurati bisogna far uso di "orologi" hardware.
Tutti i computer hanno una forma di orologio interno. Esso è il segnale di clock, un segnale ad onda quadra che ha frequenza costante.
Conoscendo il valore della frequenza di clock e potendo contare il numero di cicli di clock che sono trascorsi si ha una misura di intervalli di tempo. Questa misura di solito è piuttosto accurata, essendo basata su oscillatori al quarzo di buona stabilità.
Quindi per misurare il tempo basta avere un contatore hardware. Molti computer, PC compresi, hanno un clock particolare a frequenza relativamente bassa, al massimo dell'ordine dei MHz, realizzato appositamente per alimentare uno o più contatori, che servono a misurare il tempo per tutto il sistema.

```
Passiamo subito ad un esempio e vediamo come si può fare la misura di un intervallo di
tempo usando i servizi del DOS.
Il DOS rende disponibile il servizio AH = 2Ch, che dà in uscita il tempo intercorso fra
la mezzanotte e l'istante in cui si esegue la INT 21h di chiamata.
I parametri di uscita del servizio sono CH = ora del giorno, CL = minuto, DH = secondo,
DL = centesimo di secondo
Per misurare intervalli di tempo bisognerà "congelare" in memoria il valore del tempo
nell'istante iniziale dell'intervallo, poi sottrarre il valore "congelato" a quello de-
terminato nell'istante finale.

Per esempio:

    ..
    ; tempo del via:
    CentesimiVia DB (?)
    SecondiVia DB (?)
    MinutiVia DB (?)
    OreVia DB (?)
    ..

    ; Partenza! (inizio dell'intervallo) ricorda il momento del via:
    ; legge il tempo:
    MOV AH, 2Ch
    INT 21h
    ; ora in DL ho i centesimi, in DH i secondi, in CL i minuti,
    ; in CH le ore

    ; memorizzazione:
    MOV word PTR [CentesimiVia], DX
    ; !! occhio al trucco, la memorizzazione con word PTR
    ; !! scrive in un colpo solo sia CentesimiVia che SecondiVia!
    MOV word PTR [MinutiVia], CX
    ; !! anche qui lo stesso trucco

    ..

    ; il tempo trascorre fino a quando, in questo punto del programma,
    ; si determina l'evento che chiude l'intervallo di tempo.
```

```
    ; Arrivo! (fine dell'intervallo): segue la determinazione del tempo
    ; che è passato leggo il tempo:
    MOV AH, 2Ch
    INT 21h
    ; Per determinare il tempo passato chiamo un procedura:
    CALL DeltaTempo

    ..

DeltaTempo PROC
;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
; Scritta da: MONTI
; Data ultima modifica: XX.XX.XXXX
;   ------------------------------------------------------------------------
;   Descrizione:
;     Calcola la differenza di tempo fra un istante iniziale passato
;       attraverso la memoria ed un istante finale passato attraverso i
;       registri
;   ------------------------------------------------------------------------
;   Parametri passati attraverso:
;     Ingressi: registri e memoria
;     Uscite  : registri
;
;     Ingressi: ----------------------------
;       [CentesimiVia], [SecondiVia], [MinutiVia], [OreVia] tutti Byte
;       DL centesimi istante corrente, DH secondi istante corrente
;       CL minuti istante corrente, CH ore istante corrente
;     Uscite: ------------------------------
;       DL centesimi di tempo passato, DH secondi di tempo passato
;       CL minuti di tempo passato, CH ore di tempo passato
;     Registri modificati: -----------------
;       nessuno
;
;     Effetti collaterali sulla memoria: ---
;       nessuno
;   ------------------------------------------------------------------------
;   Note sull'uso:
;   !! questa procedura non gestisce intervalli di tempo maggiori
;   !! di 24 ore e ha problemi nella misura di intervalli
;   !! di tempo che stanno a cavallo della mezzanotte
;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    ; calcolo dei centesimi dell'intervallo, si deve risolvere un problema
    ; di "riporti":
    SUB DL, [CentesimiVia]
    ; ora in DL avrei il numero di centesimi di tempo trascorsi,
    ; però, se nel frattempo il numero di secondo è cambiato, DL ora è
    ; negativo, in questo caso devo aggiungere 100 ai centesimi per
    ; ottenere il numero giusto, ma devo anche andare a "prestito"
    ; di un secondo dal numero di secondi, che otterrò con la sottrazione
    ; successiva. Per avere questo prestito setto il carry:
    JNS NonAggiungere100
    ADD DL, 100
    STC     ; preparo il carry per la successiva SBB
NonAggiungere100:
    ; qui il valore dei centesimi è a posto, già nel parametro di uscita
    ; calcolo dei secondi, con il prestito dai centesimi:
    SSB DH, [SecondiVia]
    ; ^ Subtract with Borrow (prestito), sottrae anche il bit di carry
    JNS NonAggiungereUnMinuto
    ADD DH, 60
    STC
NonAggiungereUnMinuto:
    ; calcolo dei minuti, con il prestito dai secondi:
    SSB CL, [MinutiVia]
    JNS NonAggiungereUnOra
    ADD CL, 60
    STC
NonAggiungereUnOra:
    ; calcolo delle ore, con il prestito dai minuti:
    SSB CH, [OreVia]
    JNS NonAggiungere24
    ADD CH, 24
NonAggiungere24:
```

```
    ; qui abbiamo finito:
    ; in CH ci sono le ore, in CL i minuti,
    ; in DH i secondi, in DL i centesimi
    RET
DeltaTempo ENDP
```

La procedura DeltaTempo è inclusa nel file cronom.asm nel CDROM allegato.
Questo esempio fa vedere che è complicato usare il sistema sessagesimale per la misura dei tempi.
Un'alternativa praticabile è quella di rappresentare semplicemente il tempo con un numero di conteggi, enumerando i "tick" di un orologio.
In questo modo non si hanno quattro "riporti", per centesimi, minuti, secondi e ore, come nel programma precedente, ma eventualmente solo uno, quando il contatore esaurisce la sua capienza.
Naturalmente l'orologio di cui si vogliono contare i "tick" deve essere presente nell'hardware del computer, ma questo non è mai un problema, dato che abbiamo visto come in ogni computer ci siano molti clock.

Come esempio vediamo un programma che realizza un ritardo utilizzando un altro servizio disponibile nel PC, che misura il tempo in "tick", cioè il servizio 00h dell'INT 1Ah (read system timer counter).
Questo servizio è basato su un clock speciale e su un contatore hardware, il cui funzionamento avremo modo di illustrare dettagliatamente. Per effetto di questo contatore, un numero di 32 bit in memoria viene incrementato per 18,2 volte al secondo (ogni 55 ms).
L'INT 1Ah è una procedura contenuta nella BIOS ROM (vedi), cioè nella memoria non volatile dei PC, che va a leggere il numero da 32 bit che conta il tempo.
Se si chiama un INT 1Ah passando come parametro il numero 0 in AH si ottiene, al ritorno della "procedura", un numero di 32 bit che rappresenta il numero di intervalli di tempo di 55 ms passati dalla mezzanotte. In CX c'è la parte alta del numero, in DX la parte alta. Inoltre in AL viene passato un flag che indica, se zero, che è passata la mezzanotte dall'ultima volta che è stato chiamato il servizio.
Dunque un programma come il seguente fa un ritardo di mezzo secondo:

```
..
    ; (numero di tick del ritardo) = (numero di millisecondi del ritardo) / 55
    ; metto in un numero di 32 bit il numero di tick del ritardo:
    MOV [TempoFinaleLow],  9 ; 9 tick = (500 ms/ 55 ms)
    MOV [TempoFinaleHigh], 0

    MOV AH, 00h      ; servizio lettura system clock
    INT 1Ah
    ; in CX-DX c'è il numero di tick del system timer attuale
    ; sommo a 32 bit il ritardo al tempo attuale, ottenendo il tempo finale:
    ADD [TempoFinaleLow],  DX
    ADC [TempoFinaleHigh], CX

RitardoConServizio1A:
; leggo ancora il clock fino a che non si raggiunge il tempo finale:
    MOV AH, 00h
    INT 21h
    CMP AL, 0
    JNE FineRitardo ; uscita di emergenza se è passata la mezzanotte
    ; confronto a 32 bit:
    CMP CX, [TempoFinaleHigh] ; confronto delle parti alte
    JA FineRitardo
    JB RitardoConServizio1A
    ; qui solo se le parti alte sono uguali
    CMP DX, [TempoFinaleLow] ; confronto delle parti basse
    JB RitardoConServizio1A
FineRitardo:

    ; !! ATTENZIONE: nell'istante esatto della mezzanotte il programma
    ; !!             ESCE COMUNQUE (si potrebbe migliorare!) !!
..
```

Questo esempio è nel programma tckRitar.ASM nel CDROM allegato.
Se si considera che i ritardi siano minori di 11918 s (11918 = 655535/55), cioè 3,3 h, si possono considerare solo le parti basse del contatore a 32 bit, a patto naturalmente di considerare dei "riporti" simili a quelli utilizzati nel primo esempio.

Architettura di un counter – timer programmabile
Un counter - timer programmabile è un contatore hardware che dispone di un particolare registro modificando il quale il software può farne cambiare la modalità di funzionamento.

Scrivendo l'opportuno codice in quel registro il contatore può contare all'avanti o all'indietro, contare una volta solo fino al massimo e poi fermarsi oppure ricominciare a contare daccapo ogni volta che giunge al massimo, cambiare il valore del massimo e funzionare in molti altri modi diversi.
Un counter – timer per microprocessore è un dispositivo di I/O che si comporta come una memoria, per cui deve possedere un'interfaccia per le linee del data bus ed alcuni piedini da collegare all'address bus, per la selezione dei suoi registri interni tramite indirizzi.

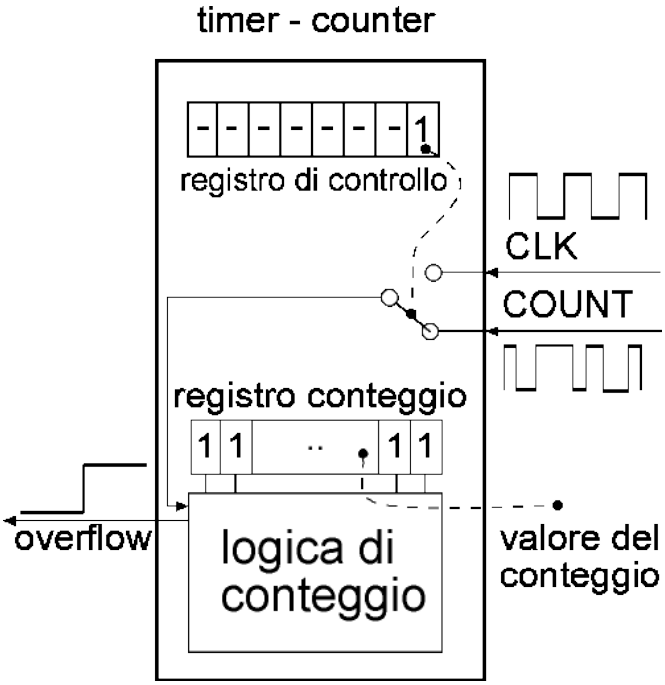


Figura 7: counter o timer

La Figura 7 mostra un circuito di I/O che può essere programmato per essere un timer oppure un counter.
Il bit meno significativo del suo registro di stato stabilisce la "sorgente" degli eventi da conteggiare.
Se in quel bit c'è un 1 si conta il numero degli eventi dall'ingresso "count", che è collegato ad un segnale irregolare.
Dunque il circuito funziona da contatore.
Se invece il bit meno significativo contiene 0 vengono conteggiate le transizioni su un segnale ad onda quadra di frequenza fissa, proveniente dall'ingresso "clock", ed il valore del conteggio è proporzionale al tempo trascorso.
Naturalmente il valore del conteggio ha un numero finito di bit, per cui oltre un valore si va in overflow.
Quando il conteggio va in overflow viene alzata una linea di uscita del circuito, indicata come "overflow" in Figura 7. In questo modo il timer è in grado di influenzare il comportamento di altri componenti hardware del computer.

Nel caso del real time clock del PC il piedino di overflow del contatore è usato come linea di interrupt, per cui ad ogni overflow del contatore parte una ISR che, contando il numero interrupt, tiene traccia del tempo trascorso.
Spesso i timer hardware contengono due registri, uno per memorizzare il conteggio, l'altro per contenere il numero di conteggi da fare prima di andare in overflow.
Molti timer – counter funzionano decrementando il valore del conteggio e segnalando la fine del conteggio sul valore di zero.

Gli eventi da conteggiare possono essere molto diversi come per esempio il passaggio dei pezzi prodotti da una macchina di fronte ad un sensore di prossimità, oppure il passaggio di finestre ottiche in una ghiera collegata ad un rullo, che serve a misurare la posizione dalla pallina di un mouse.

I microcontrollori possiedono molto spesso uno o più counter – timer al loro interno.
Un esempio di contatore commerciale (8253) è trattato nel capitolo sui circuiti di I/O.

1.2 L'Input/Output ed il software

Ora che abbiamo visto, sia pur a grandi linee, qualcosa sull'hardware di I/O, vediamo come il software interagisce con l'hardware.
Per gestire l'Input/Output in software si usano tre tecniche molto diverse:
1. Input/Output a controllo di programma o in "**polling**"
2. Input/Output a interruzione di programma o in "**interrupt**"
3. Input/Output con accesso diretto in memoria o in **DMA**

Le tre tecniche sono elencate in ordine di frequenza di uso nelle applicazioni.
Nei prossimi paragrafi se ne darà una illustrazione di massima, per passare in seguito ad una disamina in dettaglio.

1.2.1 Input/Output a controllo di programma (polling)

Il modo più comune di gestire l'I/O è il polling od "I/O a controllo di programma". Come il nome suggerisce, in questo caso il solo responsabile dell'andamento delle operazioni è il software.
Questa tecnica prevede che il programma controlli il funzionamento dei dispositivi andando regolarmente a leggerne lo stato dai relativi port di I/O.
In questo caso il programma si incarica di verificare lo stato del dispositivo con cui la CPU deve scambiare dati; il dispositivo non "prende l'iniziativa" in alcun modo.
Il termine "polling" è usato in questo contesto in modo appropriato; infatti "poll" in Inglese significa ,"sondaggio" (opinion poll = sondaggio d'opinione, exit poll = sondaggio all'uscita delle urne). Con questa tecnica si procede a "sondare" regolarmente il dispositivo per conoscerne lo stato. Il dispositivo non ha alcun ruolo attivo.

Vediamo un esempio di I/O in polling. Supponiamo di voler controllare se un carrello in movimento è giunto al fondo della sua corsa.
Per conoscere la posizione del carrello ci serve un bit di input digitale: supponiamo che il bit di peso 6 del port di indirizzo 1289, mappato in I/O isolato, sia collegato ad un microswitch. Quando il microswitch dà il valore 1, indica che il carrello è giunto a finecorsa, mentre se esso vale 0 il carrello è ancora lontano dal microswitch.
Per il controllo del motore del carrello ci serve un bit di output digitale: supponiamo che il bit di peso 1 del port 124, mappato in I/O isolato, sia collegato al motore, scrivendo un 1 in quel bit il motore si accende, se invece si scrive 0 il motore si spegne.

Il programma che realizza l'approccio del carrello al suo finecorsa è il seguente:
Esempio:

```
..
PortInput EQU 1289
PortOutput EQU 124
MotoreON EQU 10b
MotoreOFF EQU 0
MascheraANDmicroswitch EQU 01000000b
..
    ; accendo il motore:
    MOV AL, MotoreON
    OUT PortOutput, AL
    ;   ^ PortOutput < 256 => l'indirizzamento immediato è possibile
    ; attendo che il carrello giunga a finecorsa:
    MOV DX, PortInput
    ; ^ PortInput > 256 => l'indirizzamento deve essere indiretto tramite DX
LoopDiPolling:
    ; leggo lo stato del microswitch:
    IN AL, DX
    ; il carrello è a finecorsa se il bit 6 è 1:
    TEST AL, MascheraANDmicroswitch   ; verifico il bit 6
    JZ LoopDiPolling     ; se non è arrivato continuo ad aspettarlo
    ; è arrivato => spengo il motore:
    MOV AL, MotoreOFF
    OUT PortOutput, AL
..
```

Il programma tiene sotto controllo il bit di ingresso, "interrogando" continuamente il port relativo, non appena rileva che il carrello è giunto a destinazione esce dal loop di polling e spegne il motore.
Da questo esempio possiamo trarre delle conclusioni generali.
Durante il programma è stato necessario tenere sotto controllo lo stato del sistema per decidere in quale momento intervenire su di esso. Per sapere lo stato del sistema abbiamo continuamente letto il suo valore in un "loop di polling", poi siamo intervenuti con un output quando l'evento si è verificato.
Quando si esegue l'I/O sotto il controllo del programma (polling) si fa sempre così: c'è un loop nel quale si attende che accada un evento, quando l'evento accade il programma interviene.
La gran parte dei circuiti di I/O possiede un registro particolare (byte di stato) che segnala lo stato del dispositivo collegato. Leggendo continuamente questo byte di stato il software identifica gli istanti in cui il dispositivo ha bisogno di intervento, per esempio quando esso ha un byte pronto, oppure quando è pronto per accettare un dato.
Determinato in questo modo l'istante dell'intervento, il programma effettua l'operazione di I/O richiesta dal dispositivo.

Esempio di polling
Presentiamo ora un esempio di polling un po' sofisticato: un traguardo ottico "multiplo".

Otto macchine producono 8 pezzi diversi che vengono movimentati con tappeti a rulli. Alla fine del loro percorso i tappeti convergono tutti su di uno. Supponiamo di avere 8 sensori a traguardo ottico, che sono collegati sugli 8 bit di un port di I/O. I sensori danno un 1 quando c'è qualcosa davanti, 0 quando non c'è nulla. Un pezzo che entra nella parte "comune" del percorso "oscura" il relativo sensore. L'obbiettivo del nostro programma è scrivere in un vettore di byte l'ordine di arrivo dei pezzi, in modo da poterli smistare successivamente.

```
PosizioneDiArrivo DB 8 DUP ? ; vogliamo scrivere il numero del tappeto
                             ; del primo arrivato all'indirizzo PosizioneDiArrivo
                             ; il tappeto del secondo a PosizioneDiArrivo + 1
                             ; e così via.
..

MOV CX, 1    ; CX serve per aspettare tutti i pezzi,
             ; va da 1 a 8 ed è anche la posizione di arrivo
; attendo che un bit qualsiasi diventi 1:
PollTraguardo:
    MOV AL, [Traguardo] ; dispositivo mappato in memoria all'indirizzo "Traguardo"
    CMP AL, 0
    JZ PollTraguardo ; se è zero vuol dire che non è arrivato nessuno
    ;  se arrivo qui vuol dire che è arrivato un pezzo, devo
    ; individuare qual è e segnarlo nel vettore:
    MOV BX, 0    ; in BX metterò il numero della macchina
                 ; da cui arriva il pezzo (da 0 a 7)
    ; sposto nel carry un bit alla volta, se il carry diventa 1
    ; concludo che il pezzo arrivato era quello

    ; faccio un loop da 0 a 7 completo, così posso segnare anche i pezzi
    ; che arrivano contemporaneamente
NonAncoraFinito:
    SHR AL, 1
    JNC NonLoSegno
    ; Trovato! In BX ho il numero della macchina da cui usciva
    ; il pezzo che è arrivato (da 0 a 7)
    ; segno nel vettore PosizioneDiArrivo, al posto che compete alla
    ; macchina (BX), l'ordine di arrivo:
    MOV [PosizioneDiArrivo + BX], CX  ; scrivo l'ordine di arrivo (CX), nel posto
                                      ; che compete alla macchina trovata
    ; PER SEMPLICITA': se due bit scattano a 1 contemporaneamente do la priorità
    ; al primo che guardo!

    INC CX       ; aumento l'ordine di arrivo
NonLoSegno:
    ; continuo a controllare fino alla fine di AL, perché più di un bit
    ; potrebbe essere a 1:
    INC BX           ; aumento il numero di macchina
    CMP BX, 7
    JNE NonAncoraFinito ; cerco se ci sono altri bit a 1 in AL
   ; Ho controllato tutti i bit di AL. Devo aspettare altri pezzi?
   ; Controllo che siano arrivati tutti, se no passo ad aspettare il prossimo arrivo:
   CMP CX, 8
   JB PollTraguardo  ; quando torno su aspetto ancora altri arrivi
   ; se sono arrivati tutti e 8, esco da questa parte del programma
Finito:
    ..
    .. ; arrivati tutti i pezzi, il programma continua
```

Se si prova questo programma sulla macchina NON FUNZIONA. Il problema è che ogni pezzo rimane davanti al sensore per un certo tempo. Mentre il primo pezzo è ancora davanti al sensore, e perciò sta continuando a dare un 1 nel suo bit, un secondo pezzo potrebbe arrivare davanti al suo sensore e dare un 1 a sua volta. Il byte Traguardo avrà dunque due bit a 1 ma uno solo sarà da rilevare, perché l'altro è già stato rilevato.
Vediamo come risolvere questo problema:
Dobbiamo ricordarci di quali bit erano già stati rilevati in precedenza e far scattare il nostro test solo quando vanno a 1 bit che non erano mai stati a 1.
Per ricordarci, prendiamoci un po' di memoria:

```
NonAncoraArrivato DB ?   ; in NonAncoraArrivato mettiamo inizialmente tutti 1,
                         ; quando quel pezzo arriva metteremo uno 0 nel
                         ; bit che riguarda quel pezzo
..
```

```
    MOV [NonAncoraArrivato], 0FFh     ; all'inizio tutti 1
   ; facciamo il nostro loop di polling, guardando se ci sono degli 1 nei bit
   ; che corrispondono a pezzi non ancora arrivati:
PollTraguardo:
    MOV AL, [Traguardo]
    AND AL, [NonAncoraArrivato]       ; il codice che seguirà mette a zero i bit
                                      ; dei pezzi già arrivati in NonAncoraArrivato.
                                      ; quindi questo mascheramento lascia 0 in AL
                                      ; se non c'è neanche un nuovo arrivato
    JE PollTraguardo ;  se è zero vuol dire che non è arrivato nessun pezzo che non era
    ; già arrivato
    ; se arrivo qui vuol dire che è arrivato almeno un pezzo nuovo il cui bit è a 1 in AL

    ; faccio "un buco" in NonAncoraArrivato in modo che il valore di questo bit non venga
    ; considerato nei prossimi poll:
    XOR [NonAncoraArrivato], AL       ; mette a zero i bit di NonAncoraArrivato che
                                      ; corrispondono ai pezzi arrivati ora
    ; in AL sono a 1 i bit dei sensori appena arrivati,
    ; devo cercare quelli a uno e mettere a posto il vettore PosizioneDiArrivo,
    ; perciò l'ultima parte è identica a prima:
NonAncoraFinito:
    .. ; da qui è identico a prima
```

Il loop su PollTraguardo è pericoloso perché non ha altre uscite se non l'arrivo di un nuovo pezzo. Se non arrivano nuovi pezzi o se c'è un guasto nei sensori il programma rimane per sempre al suo interno. In un'applicazione "completa" questo loop sarebbe da "irrobustire" prevedendo una diversa via d'uscita, per esempio un timeout, che esce quando, passato un certo tempo massimo, non si vede arrivare nulla.
Quest'ultimo programma è un esempio di quanto potente possa essere programmare in Assembly con le maschere. E' opportuno analizzarlo in dettaglio, fino a capirlo bene, provando ad eseguirlo, a mente e sulla carta, passo per passo.

1.2.2 Input/Output a interruzione di programma (interrupt)

Non sempre la modalità di I/O a controllo di programma è adeguata ai problemi che si pongono al progettista.
Può infatti presentarsi il caso in cui la CPU non riesce ad interrogare i dispositivi con la frequenza e la regolarità che il problema impone.
In questi casi, per soddisfare in fretta le esigenze dei dispositivi di I/O, è indispensabile fare in modo che la CPU smetta di eseguire il programma corrente ed esegua un altro programma. Questo secondo programma provvede a fare l'I/O richiesto dal dispositivo.

Vediamo con un esempio un caso in cui l'I/O con polling non è adeguato al problema da risolvere.
Torniamo al carrello che si avvicina al finecorsa, esempio del paragrafo precedente. Supponiamo che il nostro programma, oltre a controllare il carrello, debba eseguire una procedura "pesante" dal punto di vista computazionale, come potrebbe essere, per esempio, l'aggiornamento della grafica dell'applicazione. Questa procedura, che chiameremo CiMette2secondi, richiede molto tempo per la sua esecuzione; il codice del programma si modifica in questo modo:

```
..
LoopDiPolling:
    CALL CiMette2secondi
    ; leggo lo stato del microswitch:
    IN AL, DX                         ;           [1]
    ; il carrello è a finecorsa se il bit 6 è 1:
    TEST AL, MascheraANDmicroswitch   ; verifico il bit 6
    JZ LoopDiPolling     ; se non è arrivato continuo ad aspettarlo
..
```

Se il carrello tocca il microswitch un attimo dopo l'istruzione [1] esso prosegue la sua corsa fino a che non si passa dalla stessa istruzione al giro successivo della loop. Questo non creava nessun problema nel programma iniziale, perché il tempo con cui si percorreva un ciclo del LoopDiPolling era irrisorio.
In questo programma le cose cambiano, perché la presenza di CALL CiMette2secondi può far si che, quando la CPU si accorge, al prossimo giro, che il microswitch è stato premuto, sia troppo tardi ed il carrello abbia proseguito la sua folle corsa, eventualmente distruggendo lo switch ed i suoi dintorni.
Per ovviare a questo problema si impone una nuova modalità di I/O nella quale sia il microswitch a segnalare che il suo stato è cambiato e la CPU possa interrompere temporaneamente l'esecuzione della procedura CiMette2secondi, per poter spegnere subito il motore.

Un **interrupt** consiste nell'esecuzione immediata, su richiesta da parte dell'hardware, di una procedura.

Quando riceve una richiesta d'interruzione la CPU interrompe ! :-) temporaneamente il programma che stava eseguendo e lancia una procedura, che esaudisce la richiesta da parte del dispositivo. Al ritorno di questa procedura la CPU riprende regolarmente l'esecuzione del programma che era stato interrotto.
La procedura che viene eseguita in conseguenza ad una richiesta d'interruzione viene detta "procedura di risposta all'interruzione", in Inglese "Interrupt Service Routine" (**ISR**), o anche Interrupt Handling Routine (IHR) o "interrupt handler" ("manipolatore" di interrupt ..).
Usando l'interrupt la CPU, invece di sprecare il tempo attendendo che succeda qualcosa, può procedere a fare altro, ed essere interrotta solo quando c'è la necessità di Input/Output.
Per dirla con Peter Norton: "(With polling the CPU) wastes its time looking for work - (With interrupt) when there is something to be done, the work comes looking for the processor."

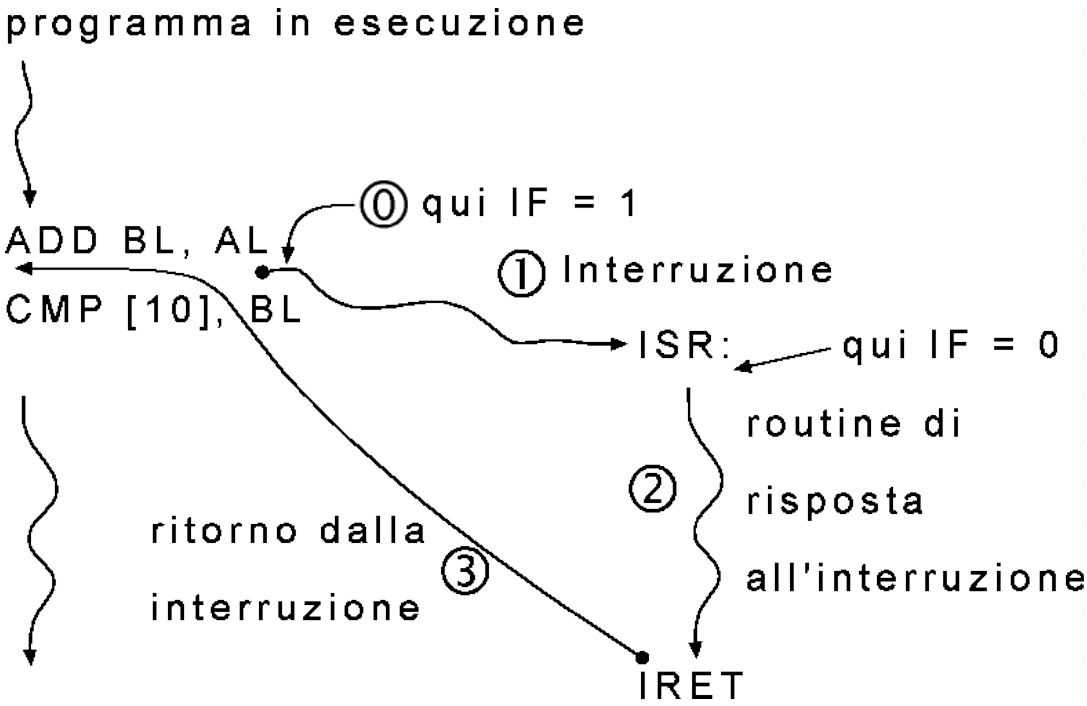


Figura 8: interruzione del programma e salto alla ISR

Funzionalmente la ISR non è diversa da una procedura "ordinaria", già descritta precedentemente; l'unica, fondamentale, differenza è che, mentre la procedura viene chiamata esplicitamente dal software, attraverso un'istruzione CALL, una ISR viene chiamata dall'hardware e può scattare in OGNI istante. Almeno nelle CPU X86 ci sono inoltre differenze hardware nelle sequenze di esecuzione delle procedure e degli interrupt.
Una ISR deve essere una piccola procedura "mordi e fuggi" che fa il suo lavoro rapidamente e ritorna per restituire il controllo al programma che ha interrotto; per ragioni che diverranno chiare più avanti le ISR che impiegano molto tempo ad eseguire sono "pericolose".
Come indicato anche dalla Figura 8, la CPU deve avere un'istruzione che indichi la conclusione della ISR. In Figura 8 (punto (2)) è indicato il mnemonico delle CPU X86, IRET (**I**nterrupt **Ret**urn). Le ragioni per cui l'istruzione IRET è diversa da una normale RET saranno chiare più avanti.

Dato che l'interrupt viene iniziato dal dispositivo, esso deve essere supportato dall'hardware: deve esistere un "**sistema d'interruzione**"[1], che porti alla CPU la richiesta di intervento dei dispositivi hardware.

A questo scopo si deve introdurre una nuova linea del control "bus", che chiameremo "**Int**errupt **R**equest" (Richiesta d'interruzione") (**INTR**).
La linea Interrupt Request è, lo dice il nome, la richiesta d'interruzione; per questo viene controllata dal dispositivo esterno.
Il sistema d'interruzione può essere disabilitato; infatti la CPU possiede un "flag di interrupt" (interrupt flag = IF), che ne permette la totale esclusione. Se IF = 0 le richieste d'interruzione vengono completamente ignorate, se invece IF = 1 il sistema d'interruzione è abilitato ed esegue le ISR quando ne viene fatto richiesta.

1 Il sistema d'interruzione è presente in quasi tutte le CPU odierne, solo i microcontrollori più a buon mercato non lo hanno.

Quando la linea INTR, controllata dal dispositivo, si alza, la CPU esegue una sequenza automatica che deve saltare al codice della ISR (Figura 8, punto (1)). Ciò naturalmente a patto che IF = 1 (Figura 8, punto (0)), altrimenti prosegue.

Una cosa molto importante da notare è che il programma interrotto deve poter riprendere come se nulla fosse successo. Innanzitutto deve tornare all'istruzione successiva a quella ove è avvenuta l'interruzione (punto (3) di Figura 8).
Per ricordarne l'indirizzo bisogna che la CPU lo salvi nello stack prima del salto. Si noti che questo deve essere fatto necessariamente dalla CPU, perché il programmatore non può sapere quando scatta un interrupt.
Dunque la sequenza di interruzione deve fare ciò che fa ogni chiamata a procedura: salvare nello stack il contenuto corrente do Program Counter, in modo che la successiva istruzione di ritorno possa tornare all'istruzione giusta leggendone l'indirizzo nello stack.
La più semplice sequenza delle operazioni che vengono eseguite durante un interrupt è dunque la seguente:

1. **Dispositivo**. Per richiedere servizio da parte della CPU alza il segnale di richiesta d'interruzione (INTR)
2. **CPU**. Completa la fase di execute dell'istruzione corrente
3. **CPU**. Verifica se c'è una richiesta di interruzione su INTR. Se IF = 0 oppure INTR = 0 passa al fetch dell'istruzione successiva del programma correntemente in esecuzione, non eseguendo alcuna ISR
4. **CPU**. Se IF = 1 e (and) INTR = 1 salva nello stack l'indirizzo di ritorno
5. **CPU**. Salta alla procedura di risposta all'interruzione, in uno dei modi che descriveremo in seguito.

Si noti che la sequenza appena indicata è molto semplificata; nel prossimo capitolo esamineremo la sequenza completa per una CPU 8086, che terrà conto di diverse altre operazioni.

Per proseguire nella sequenza di interrupt la CPU deve stabilire:
- A quale indirizzo saltare (l'indirizzo della ISR)
- Qual è il dispositivo che sta richiedendo l'interruzione

Stabilire l'indirizzo della ISR

Riguardo al salto all'indirizzo della ISR i progettisti delle varie famiglie di CPU hanno dato prova di fantasia. Molte e molto diverse sono le soluzioni scelte per questo problema; discutiamo in questa sede le principali:
- Indirizzo fisso: la ISR sta sempre alla stessa locazione di memoria.
All'arrivo dell'interrupt il flusso dell'esecuzione salta ad una specifica locazione di memoria, fissa e stabilita dal costruttore della CPU. Questa soluzione ha il pregio di essere semplice, per cui è ancora molto usata nel campo dei microcontrollori, che devono costare poco.
In questo caso quando si compila una ISR bisogna fare in modo che essa venga allocata proprio all'indirizzo previsto dal costruttore[1].
- Salto indiretto: la CPU trova l'indirizzo della ISR in memoria, in una locazione fissa
Questa soluzione, non molto più complicata della precedente, è un po' più flessibile, dato che la ISR può essere allocata in qualsiasi punto della memoria, a patto di scriverne l'indirizzo nella locazione fissa, stabilita dai progettisti della CPU, in cui la CPU stessa lo va a cercare.
- **Salti vettorizzati** (vectored interrupt)
Le CPU con sistema di interruzione vettorizzato sono in grado di ricevere l'interrupt da molti dispositivi ed anche di capire quale di essi sta interrompendo in un certo istante. Questi sistemi d'interruzione sono in grado di saltare direttamente ad una di tante ISR, cioè proprio a quella del dispositivo che sta interrompendo.
L'interrupt vettorizzato verrà trattato in dettaglio nel capitolo successivo che illustra il sistema di interrupt vettorizzato dell'8086.

1 con TASM e MASM si dovrebbe usare la direttiva AT, che qui NON verrà illustrata.

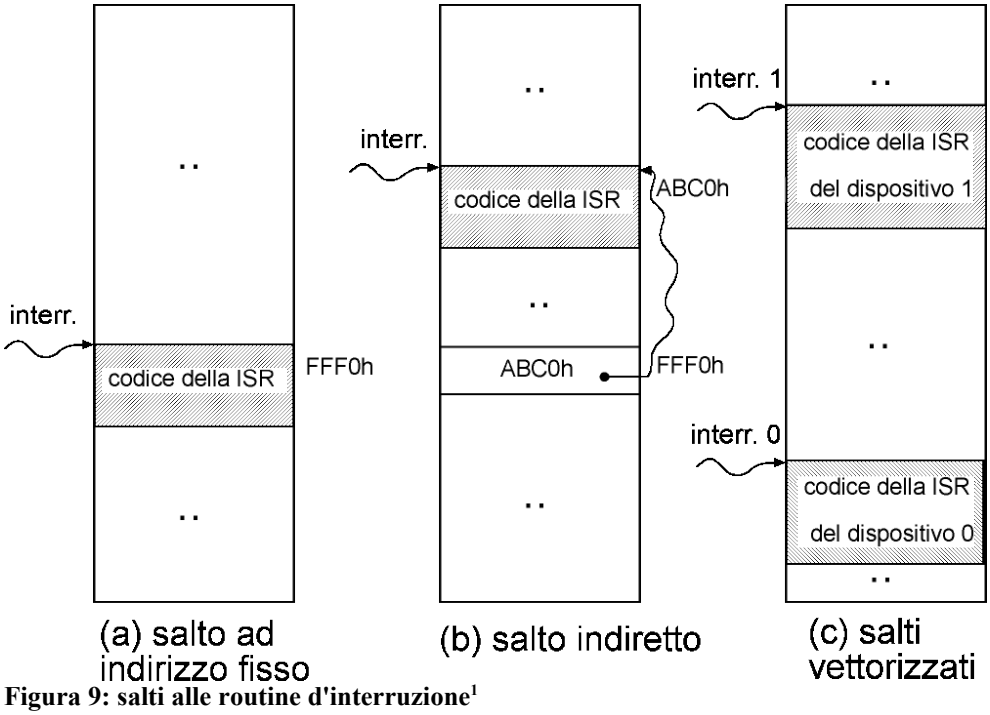


Figura 9: salti alle routine d'interruzione[1]

Interrupt da molti dispositivi

Quando più di un dispositivo può lanciare una richiesta d'interruzione si pongono due ordini di problemi:
- Individuazione del dispositivo

Dato che i dispositivi sono molti, si deve capire quale è quello che richiede l'interrupt.

- Priorità degli interrupt

Se due o più dispositivi richiedono un interrupt contemporaneamente si deve decidere quale di essi andrà servito per primo.

Individuare il dispositivo che interrompe

Per capire quale dispositivo sta richiedendo un'interruzione ci sono tre alternative:
- Individuazione via software ("interrupt polling")
- Individuazione con diversi piedini di interrupt
- Identificazione via hardware da parte del dispositivo (interrupt **vettorizzato**)

Individuazione del dispositivo via software

La situazione in cui è necessario un riconoscimento via software del dispositivo è quella illustrata in Figura 10.

Essa mostra tre dispositivi, non necessariamente identici, che possono alzare un segnale di interrupt sulla stessa linea (INTR) della CPU.

Ogni dispositivo in grado di lanciare un'interruzione deve avere un registro interno che ne indica lo stato (in genere viene detto "status byte"). Un bit all'interno di questo registro indica in ogni istante se il dispositivo sta effettivamente richiedendo un interrupt.

Andando a vedere, uno alla volta, i valori di questi bit si può capire quale sia il dispositivo che interrompe.

Scriviamo una ISR di esempio, nella quale identifichiamo uno di tre possibili dispositivi che hanno lanciato l'interrupt .

1 gli indirizzi sono di fantasia

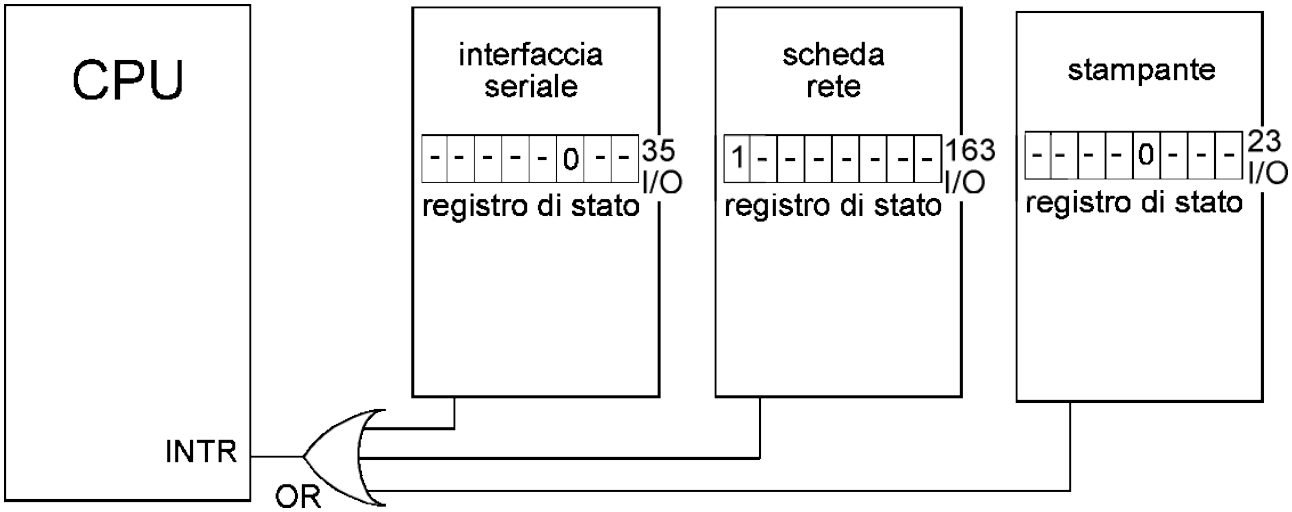


Figura 10: più dispositivi sulla stessa linea d'interruzione ("daisy chain")

Nella figura sono illustrati tre dispositivi, uno solo dei quali (scheda rete) sta lanciando un interrupt, la linea INTR della CPU viene alzata se almeno una delle linee d'interruzione dei dispositivi è alta.

```
..
; indirizzi degli "status byte" dei dispositivi:
; (i numeri non sono indicativi! (ma sono < 256))
StatusByteRete EQU 163   ; indirizzo del byte di stato della scheda di rete
StatusByteStamp EQU 23   ; porta stampante
StatusByteSeriale EQU 35 ; porta seriale

; Maschere che identificano qual è il bit di interrupt dentro lo status byte
; (quando il bit vale 1 quel dispositivo sta lanciando l'interrupt => le
; maschere sono "da AND":
MaskINTbitRete EQU 10000000b ; maschera per l'interrupt della scheda di rete
MaskINTbitStamp EQU 00001000b
MaskINTbitSeriale EQU 00000100b
..
ISRqualeDeiTre:
; se entro nella ISR significa che c'è almeno una linea di interrupt ON
    PUSH AX  ; salvo AX perché userò AL
    ; leggo il byte di stato della scheda di rete:
    IN AL, StatusByteRete
    TEST AL, MaskINTbitRete
    JNZ EraLaRete; se il bit vale 1 => la scheda di rete ha interrotto
    ; leggo il byte di stato del chip di I/O della seriale:
    IN AL, StatusByteSeriale
    TEST AL, MaskINTbitSeriale
    JNZ EraSeriale   ; se il bit vale 1 => la seriale ha interrotto

    ; se arrivo qui è la stampante che ha interrotto
    ..
    ; qui è il codice della ISR per la stampante
    ..
    JMP Esci
EraLaRete:
    ..
    ; qui è il codice della ISR per la rete
    ..
    JMP Esci
    ..
EraSeriale:
    ..
    ; qui è il codice della ISR per la stampante
    ..
Esci:
    POP AX   ; ripristino AX del programma interrotto
    IRET
```

Analizzando questo programma vediamo che esso risolve anche il problema della priorità fra i dispositivi.
L'ordine con cui il programma della ISR interroga i dispositivi stabilisce la priorità degli interrupt.

Poniamo infatti che tutte e tre le richieste di interrupt siano abilitate. Allo scattare della ISRqualeDeiTre il primo interrupt che viene verificato viene anche servito, per cui l'interrupt più che ha maggiore priorità è quello della scheda di rete, mentre gli altri due rimangono in attesa.
Servito l'interrupt della rete la relativa scheda non interrompe più, perchè la ISR ha letto i dati che la scheda doveva comunicare ed essa non ha più l'esigenza di traferire informazioni. Però, appena finita la ISR, l'interrupt scatta ancora, lanciato dagli altri due dispositivi che hanno ancora bisogno di essere serviti.
Allora ISRqualeDeiTre partirà un'altra volta e questa volta servirà il secondo dispositivo che viene controllato, cioè la porta seriale.
La volta successiva sarà il turno della porta della stampante; l'ordine di priorità realizzato dal precedente programma è perciò: rete, seriale, stampante.
Dato che questa tecnica prevede l'interrogazione sequenziale dei dispositivi viene anche detta "interrupt polling", anche se il nome è un po' ambiguo, dato che non è una tecnica di I/O in polling, ma in interrupt!

Individuazione del dispositivo con diversi piedini di interrupt
Alcuni tipi di CPU, in particolare i microcontrollori, hanno più di un piedino di interrupt.
Collegando dispositivi diversi a queste linee è possibile riconoscere il dispositivo che interrompe e saltare direttamente alla procedura di risposta all'interruzione del dispositivo giusto
Per esempio la CPU salterà ad un indirizzo se l'interrupt arriva dal piedino INTR1, e ad altri indirizzo se arriva dagli altri piedini.

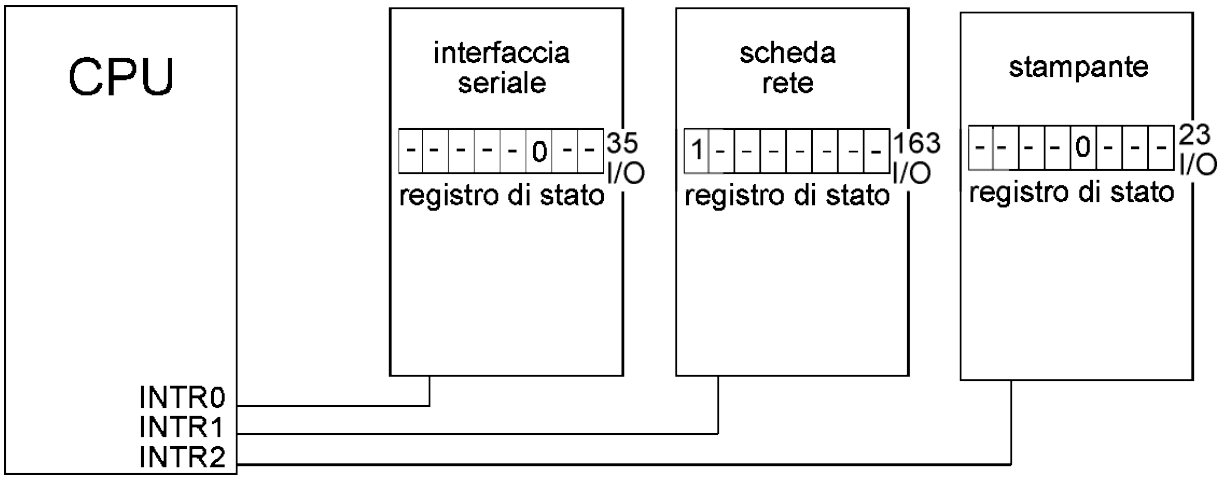


Figura 11: più dispositivi su diverse linee d'interruzione
Il software d'interruzione per una CPU del genere comprenderà istruzioni e flag particolari, in modo che il programmatore possa capire su quale piedino giunge la richiesta d'interruzione.
Le procedure di risposta saranno separate, ciascuna si esse sarà evocata quando scatta l'interrupt nel relativo piedino.
Proponiamo un esempio "stile X86" (avvertendo però che in verità gli X86 hanno UN solo piedino di interrupt!):

```
..
StatusByteDisp1 EQU 163  ; i numeri non sono indicativi!
StatusByteDisp2 EQU 23
StatusByteDisp3 EQU 35

..
ISRdisp1:
; se entro in questa IRS significa che c'è stato un interrupt dal disp.n.1
    ..
    ; qui è il codice della ISR per il dispositivo 1
    ..
    IRET
ISRdisp2:
; se entro in questa ISR significa che c'è stato un interrupt dal disp.n.2
    ..
    ; qui è il codice della ISR per il dispositivo 2
    ..
    IRET
ISRdisp3:
; se entro in questa ISR significa che c'è stato un interrupt dal disp.n.3
    ..
    ; qui è il codice della ISR per il dispositivo 3
    ..
    IRET
```

Se più di un interrupt viene lanciato contemporaneamente la decisione su qual è il più importante deve essere presa dall'hardware, in questo caso dall'unità di controllo della CPU, che deve anche lanciare la relativa ISR.
Gli altri dispositivi rimarranno in attesa e verranno serviti quando la ISR del più importante è terminata.
L'hardware dovrà mettere a disposizione funzioni che realizzano la priorità ed, eventualmente, la rendono "programmabile".

Individuazione **vettorizzata** del dispositivo
Nei sistemi con interrupt vettorizzato il dispositivo spedisce alla CPU un "codice" tramite il quale esso si identifica. La CPU, raccogliendo questo codice, può saltare direttamente alla ISR del dispositivo giusto.
Come si può intuire da quanto detto l'interrupt vettorizzato permette di risolvere "contemporaneamente" il problema dell'identificazione del dispositivo e quello dell'indirizzo della ISR.
Il mezzo che permette al dispositivo di passare il suo "codice" alla CPU è, e non potrebbe essere altrimenti, il data bus. I dettagli nel prossimo capitolo.

Il problema del riconoscimento dell'interrupt da parte del dispositivo

Quando un dispositivo lancia un'interruzione continua a tenere alta la sua linea di interrupt fino a che la CPU, o chi per essa, non gli comunica che l'interrupt è stato servito.
Ricevuta questa comunicazione abbassa la sua linea di richiesta, così che il suo interrupt non venga servito di nuovo, anche se non ce n'è più bisogno.
La CPU può comunicare il "riconoscimento" dell'interrupt ("acknowledge"[1]) in molti modi, tutto dipende da come è fatta l'elettronica del dispositivo e quella della CPU.

Alcuni dei modi più tipici sono i seguenti:
- Lettura o scrittura nel port di I/O interessato
  L'interrupt viene lanciato per richiedere l'intervento della CPU in un'operazione di I/O.
  Quando la CPU esegue l'operazione di I/O richiesta non c'è più bisogno di interrupt, per cui l'elettronica del circuito di I/O del dispositivo abbassa automaticamente la sua richiesta.
  Per esempio il circuito che controlla la porta seriale di un PC (UART 8250) lancia un interrupt quando ha un dato pronto per la lettura da parte della CPU e lo abbassa automaticamente non appena la CPU effettua la lettura.
  Lo stesso avviene di solito per i counter – timer, che abbassano la loro richiesta d'interruzione quando si legge il valore del loro conteggio.
- Sequenza di "End Of Interrupt"
  In altri casi non basta leggere il dato che interessa perché il dispositivo abbassi la sua richiesta di interruzione, ma è necessario scrivere un codice od una sequenza di codici in un altro registro del circuito di I/O.
  In questo caso bisogna studiare attentamente il databook del dispositivo essere sicuri che la richiesta d'interruzione venga abbassata.
  Per esempio si può considerare l'interrupt controller 8259, che richiede la scrittura del numero 20h in uno dei suoi registri per poter riprendere a funzionare regolarmente dopo il lancio di un'interruzione.
- Linea elettrica al circuito di I/O
  Il chip di I/O della periferica che richiede l'interruzione potrebbe avere uno specifico piedino che, quando viene alzato dalla CPU, fa abbassare la richiesta d'interruzione. Questo piedino potrebbe essere collegato ad un output digitale della CPU (questa tecnica viene utilizzata nei microcontrollori). Durante la ISR la CPU alza quell'output e la richiesta di interruzione viene abbassata.

Salvataggio del contesto

Quando una procedura di risposta viene lanciata non si sa in quale punto venga interrotto il programma principale, per cui non si sa neppure esso sta utilizzando o meno alcuni dei registri.
Per questo all'interno di una ISR non si può far altro che salvare tutti i registri che si usano e ripristinarli prima della IRET, in modo da restituire intatti tutti i registri al programma che ha interrotto.

Chiamiamo "**contesto**" (context) di una procedura l'insieme di tutti i registri che essa usa.

Tutte le ISR necessitano di un salvataggio integrale del loro contesto al momento della loro esecuzione e del ripristino integrale del contesto prima del loro ritorno.

Tempo di latenza dell'interrupt

Il modo più rapido e preciso per misurare il tempo di risposta di una ISR è accendere un bit di un port di I/O non usato all'inizio della procedura e spegnerlo alla fine. Con un oscilloscopio si potrà misurare il tempo in cui quel bit rimane ON, quello è la durata della ISR.
Un altro modo è affidarsi ai contatori inclusi nell'hardware del sistema, salvandone il valore all'inizio della ISR e facendo la differenza con il valore che assume il contatore alla fine.

1 il termine "interrupt acknowledge" ha anche un altro significato nel "mondo" X86, come vedremo nel prossimo capitolo.

Interrupt non mascherabile

I microprocessori possiedono piedini speciali per gli "**interrupt non mascherabili**".

Un interrupt non mascherabile viene eseguito comunque, anche se l'interrupt flag è a zero.

Questo tipo di interrupt deve essere usato solo per casi molto particolari come eventi speciali e drammatici (p.es. rilevazione di un malfunzionamento nella memoria o nell'hardware, condizioni di errore irrecuperabili).
Solitamente la ISR di un interrupt non mascherabile non si conclude con una IRET ma con lo spegnimento od il "reset" del sistema.

1.2.3 Input/Output con accesso diretto in memoria (**DMA**)

Quando la quantità di dati di I/O da trasferire è grande rispetto alle capacità della CPU, l'I/O interruzione può divenire inadeguato.
Per chiarire la cosa poniamo subito un esempio. Supponiamo di dover leggere di dati provenienti da una porta seriale RS232 (vedi nel capitolo "Architettura del PC"), che riceve alla velocità di 9600 bit/s. A questa velocità verrano ricevuti circa 1000 caratteri di 8 bit al secondo.
Considerando che il tempo di latenza di un interrupt è dell'ordine di un microsecondo, il processo di interruzione impiega circa un millesimo del tempo che la CPU può dedicare ad altri compiti.
E' perciò del tutto accettabile che la porta seriale lanci un interrupt per ogni carattere che riceve. Ciò è vero anche se spingiamo la porta seriale al massimo della sua velocità, che è di 115 kbit/s. In questo caso il rapporto fra tempo dedicato all'interrupt e tempo "libero" diviene superiore all'1%, ed ancora accettabile.
Esistono dunque dei dispositivi per i quali l'acceso ad interruzione è sempre adeguato.
Considerando che esistono dispositivi più veloci, come per esempio una scheda di rete che produce un nuovo carattere ogni microsecondo, ci rendiamo conto che esistono dispositivi in cui il periodo fra la ricezione di un carattere ed il successivo è confrontabile con il tempo di latenza dell'interrupt.

Quando ciò accade si può pensare di usare l'interrupt in due modi:
1. Ad ogni carattere ricevuto viene lanciato un interrupt.
In questo caso al ISR deve eseguita ad ogni carattere ed ogni volta è necessario salvare e ripristinare indirizzo di ritorno e contesto del programma interrotto. Questo fa perdere tempo e fa diminuire l'efficienza del computer.
2. Viene lanciato un solo interrupt per un intero "blocco" di dati, grande centinaia o migliaia di Byte.
In questo caso la ISR è necessariamente "lunga", dato che deve trattare molti dati.
Ricordando che le ISR eseguono a interrupt disabilitati, ciò significa che il sistema di interruzione viene disabilitato per un lungo tempo. Questa cosa è da sconsigliare, così come è da sconsigliare l'abilitazione degli interrupt durante le ISR, come sarà più chiaro successivamente.

Generalizzando i concetti appena enunciati, si può dire che le periferiche di un computer possono essere divise in due categorie: "**periferiche a caratteri**" e "**periferiche a blocchi**".
Le periferiche a caratteri comunicano "un numero alla volta", mentre quelle a blocchi hanno l'esigenza di comunicare a tale velocità che debbono trasferire molti numeri "insieme", in blocchi di dimensioni considerevoli.
Esempi di periferiche a caratteri possono essere la tastiera od il modem, mentre l'hard disk o lo scanner grafico sono tipiche periferiche a blocchi.
Per le periferiche a caratteri di solito basta l'interrupt, mentre per le periferiche a blocchi potrebbe essere necessario utilizzare una modalità di I/O più efficiente dell'interrupt.
Una tecnica di I/O più efficiente dell'interrupt è il DMA.
Il termine **DMA** significa **D**irect **M**emory **A**ccess, cioè "accesso diretto in memoria".
Per poter usare una tecnica DMA il dispositivo, oppure un altro circuito intermediario, deve essere in grado di prendere il controllo dell'address bus al posto della CPU e di scrivere i suoi dati "direttamente in memoria", senza aiuto da parte della CPU.
Sin dal volume precedente avevamo affermato con forza che la CPU deve essere l'unico dispositivo del computer che stabilisce quali sono gli indirizzi della memoria su cui si lavora. Per questo abbiamo sempre disegnato l'address bus con una sola freccia che parte dalla CPU.
La tecnica del DMA costituisce un'eccezione a questa regola, perché in questo caso è il dispositivo di I/O che stabilisce gli indirizzi dei trasferimenti in memoria.
Nella pratica non è il dispositivo in prima persona che ha accesso diretto alla memoria, ma piuttosto un dispositivo dedicato, detto **DMA controller**.
Il DMA controller fa da intermediario fra la CPU e la periferica, rispettando tutte le specifiche elettriche e temporali che permettono di prendere il controllo dell'address bus del microprocessore.
Il DMA è una tecnica complicata dal punto di vista dell'hardware ed è realizzata in modo diverso in ogni tipo di microprocessore, per questo i DMA controller vengono progettati per lavorare con specifiche famiglie di microprocessori.

Dunque abbiamo visto che in un computer con DMA ci sono due unità che possono controllare l'address bus: la CPU ed in DMA controller. Perché tutto funzioni è necessario che una sola di queste unità sia attiva in un determinato istante, cioè che la CPU "sparisca" dall'address bus nel momento in cui il DMA controller ne prede il controllo, e viceversa. Nella Figura 12 è illustrata una semplice sequenza di scrittura DMA.

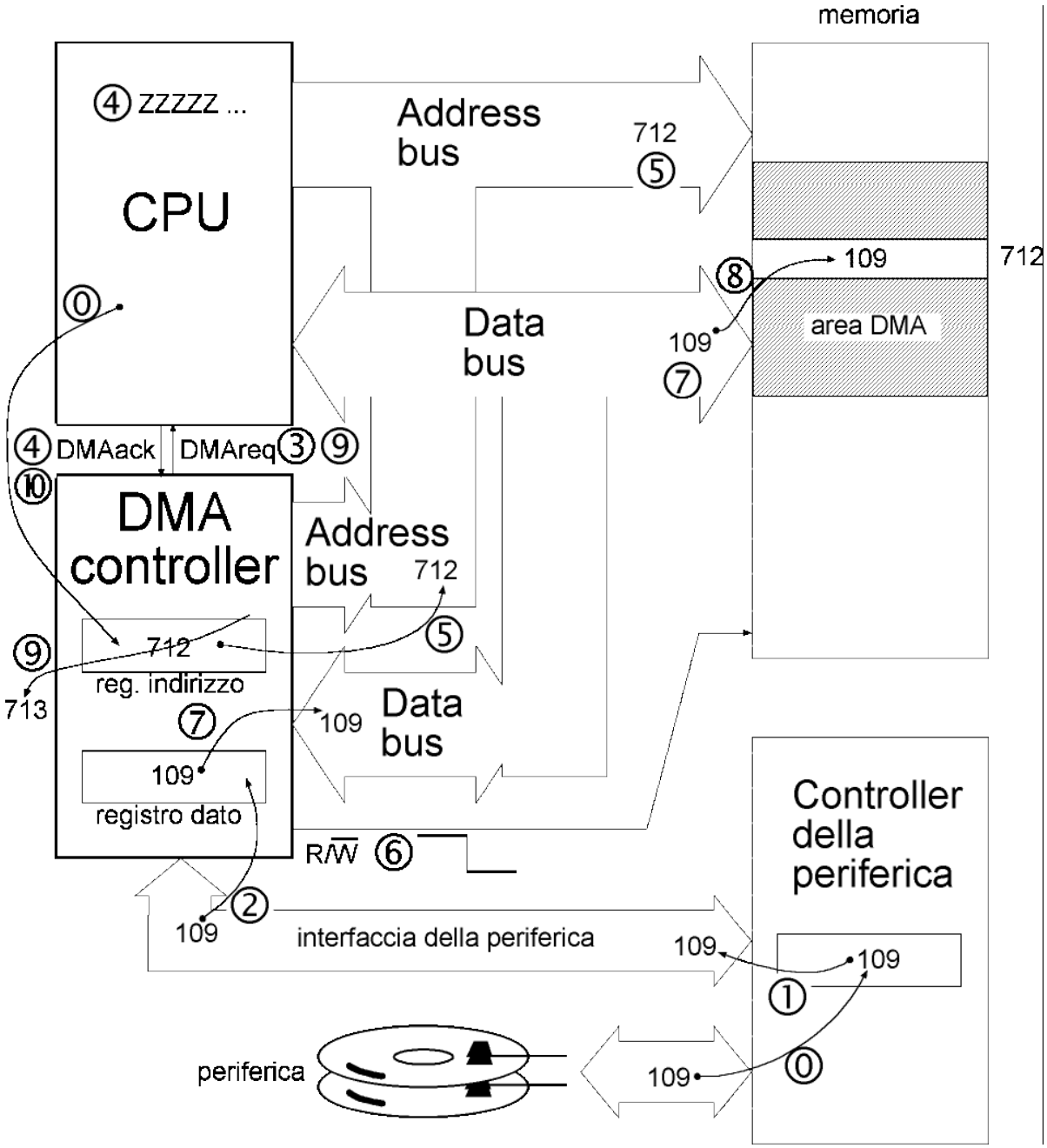


Figura 12: sequenza di DMA

(0) In normali condizioni operative il DMA controller non entra in gioco ed è "distaccato" dall'address bus (in alta impedenza). Prima dell'inizio di una sequenza di DMA la CPU scrive in un registro del DMA controller l'indirizzo di memoria nel quale deve avvenire la scrittura (712 in Figura 12). Questo registro è stato indicato come "registro indirizzo" nella Figura 12.

(1) La periferica ha un dato pronto, che deve essere acquisito nella memoria del computer. Il dato è letto dalla periferica (es. hard disk) ed è presente in un registro del suo controller.

(2) DMA controller: acquisisce dal controller della periferica il valore da trasferire in memoria, attraverso un'apposita interfaccia che qui non specifichiamo ulteriormente.

(3) DMA controller: richiede alla CPU un accesso diretto alla memoria alzando la linea DMAreq (DMA request. Richiesta di DMA)

(4) CPU: accorda al DMA controller l'accesso in DMA, alzando la linea DMAack (DMA acknowledge, autorizzazione al DMA). In questo momento la CPU mette entrambi i suoi bus in stato di alta impedenza, poi si "addormenta", metten-

dosi in una condizione detta di "**idle**" ("ozioso", "nullafacente") nella quale non fa nulla. In tutto il periodo in cui la CPU è in stato di idle essa non esegue nessuna istruzione e non partecipa in alcun modo a ciò che avviene.
(5) DMA controller: prende il controllo dell'address bus e vi scrive il valore dell'indirizzo contenuto nel suo registro d'indirizzo (712 in Figura 12). Ora avviene una normale scrittura in memoria, con la differenza che è controllata dal DMA controller e non dalla CPU.
(6) DMA controller: abbassa la linea R/W per indicare che l'operazione è una scrittura
(7) DMA controller: scrive sul data bus il valore da copiare in memoria (109 in Figura 12)
(8) Memoria: memorizza all'indirizzo indicato il contenuto del data bus
A questo punto la scrittura in DMA si è conclusa, si deve ripristinare il funzionamento normale del computer.
(9) DMA controller: abbassa la linea DMAreq e si pone in alta impedenza sui bus. E' possibile, ma non obbligatorio, che il DMA controller aggiorni il valore del suo registro indirizzo, per essere pronto a scrivere alla locazione successiva alla prossima sequenza di DMA (in Figura 12, 712 diviene 713).
(10) CPU: riprende il controllo dei bus, abbassa la linea DMAack, esce dallo stato di idle, ricominciando ad eseguire le istruzioni del suo programma.
Una sequenza di lettura sarà simile a questa e non la dettagliamo.
Una cosa da ben rimarcare nella sequenza appena descritta è che la CPU durante la scrittura in DMA è inattiva.
E' naturale domandarsi qual è la convenienza nel far eseguire il DMA, visto che durante il DMA la CPU non fa nulla.
La differenza è che, a differenza dell'interrupt, con il DMA non è necessario alcun salvataggio, né ripristino del contesto.
Come abbiamo già visto ad ogni ISR vengono salvati, come minimo:
- l'indirizzo di ritorno
- il valore di tutti i flag
- il valore di tutti i registri che verranno usati durante la ISR
Queste operazioni possono richiedere anche alcune centinaia di cicli di macchina e non sono mai effettuate durante i trasferimenti in DMA.
Infatti durante un DMA lo stato di esecuzione della CPU viene "congelato" e ripreso tale e quale quando il DMA è concluso. Non c'è bisogno di salvare i registri, i flag od il program counter semplicemente perché essi non cambiano durante il DMA.
Per questo se le richieste di I/O sono molto frequenti si risparmiano molti salvataggi e ripristini di contesto ed il computer diviene più efficiente. Si può anche notare che le operazioni di una sequenza di DMA sono tutte eseguite in hardware e perciò con velocità molto maggiore.
Questa maggiore efficienza si paga con una notevole complicazione dell'hardware, per cui il DMA è più costoso della altre modalità di I/O e viene usato solo quando è indispensabile.
Vista dalla parte della CPU il DMA è solo un periodo di tempo nel quale essa di "addormenta" temporaneamente. Al suo risveglio la CPU continua a fare quello che faceva prima.
Durante il sonno però il contenuto dell'area di memoria interessata dal DMA è cambiato, senza che la CPU se ne potesse render conto. La CPU potrà andare a leggere in contenuto di quell'area di memoria quando ne avrà bisogno, in un secondo tempo e "con comodo".

La sequenza descritta illustra un tipo di DMA detto "**DMA byte mode**", nel quale avviene una richiesta di DMA per ciascuna scrittura in memoria che è necessaria.
Quando i dispositivi sono molto veloci il DMA byte mode potrebbe non bastare.
Se si pensa che un hard disk di tipo SCSI può trasferire facilmente 40 MByte/s e che certi computer hanno bus con velocità dell'ordine del GByte/s ci si rende conto che in questi casi bisogna utilizzare una modalità di DMA che trasferisca più di un carattere con una singola richiesta.
Il "**DMA burst mode**" (burst = raffica) trasferisce un blocco di dati per ogni sequenza di DMA, ed è implementato nei bus dei moderni computer (es. bus PCI nei PC).
Perché il DMA burst mode sia possibile il DMA controller deve conoscere più informazioni di quelle necessarie nel caso "byte mode".
Siccome il DMA controller deve mantenere il controllo dei bus per tutto il tempo del trasferimento di un intero blocco di dati esso dovrà conoscere non solo l'indirizzo iniziale dell'area di memoria del DMA, ma anche la lunghezza del blocco da trasferire.

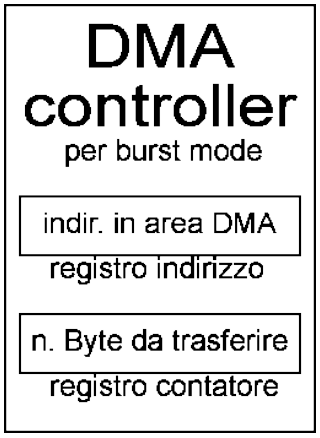


Figura 13: DMA burst mode

Si può immaginare che nel DMA controller sia presente un secondo registro, indicato come "registro contatore" in Figura 13, che contiene il numero di Byte da trasferire.
In questo registro la CPU scrive, prima dell'inizio della sequenza DMA, il valore della lunghezza del blocco.
Durante la sequenza di DMA il controller incrementa ad ogni Byte trasferito il valore del registro di indirizzo e decrementa il registro contatore, terminando la sequenza di DMA quando il registro contatore assume il valore zero.

Idle attivo

Un altro punto sul quale i produttori di CPU hanno lavorato per migliorare l'efficienza del DMA è lo stato di idle.
Nelle CPU moderne durante un DMA la CPU non sta ferma ma esegue tutte le istruzioni che ha nella cache interna e che non richiedono accesso alla memoria principale.
Dato che le cache interne, per le istruzioni e per i dati, delle CPU odierne sono molto vaste, può accadere che si riesca a proseguire nell'esecuzione delle istruzioni quasi come se il DMA non avesse luogo.

1.2.4 Bus Mastering

Modalità di I/O simile al DMA, ma più sofisticato.
In questo caso il controller del dispositivo non solo ha accesso diretto alla memoria, ma ha anche capacità di elaborazione e si sostituisce completamente alla CPU, sia pur in modo temporaneo.
Oltre che per l'I/O il bus mastering è anche una delle tecniche che vengono usate per realizzare sistemi multiprocessore (vedi in seguito). Attraverso un bus di espansione che supporti questa tecnologia si può condividere fra diverse CPU una memoria comune. [1]
Questo accadere, per esempio, in certi sistemi multiprocessore basati sul bus PCI.

1 PCI è il bus usato per molti anni nei PC con CPU da 32 bit in su. Sono PCI i connettori di espansione bianchi sulla scheda madre.