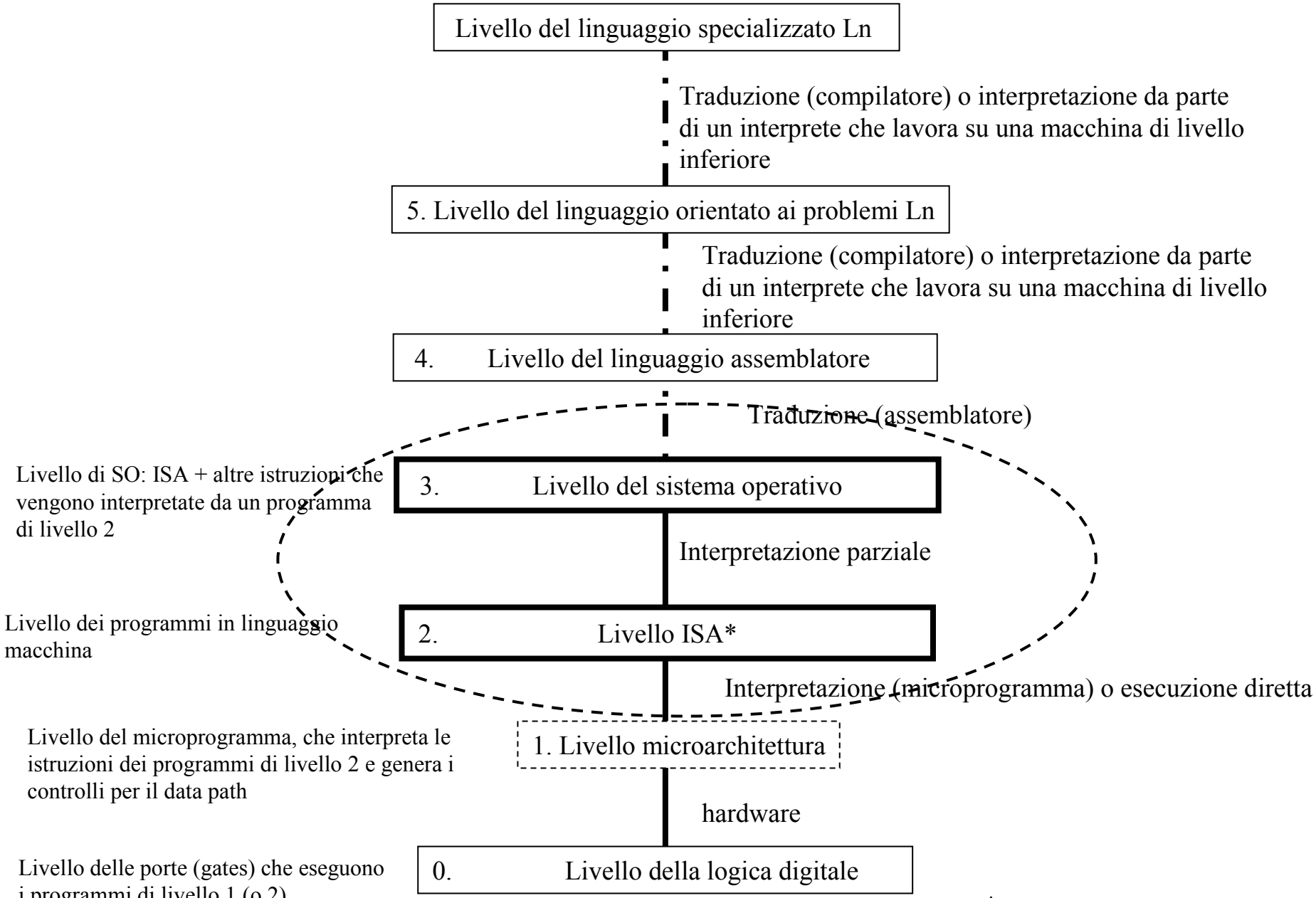
# Calcolatori Elettronici B
# a.a. 2004/2005

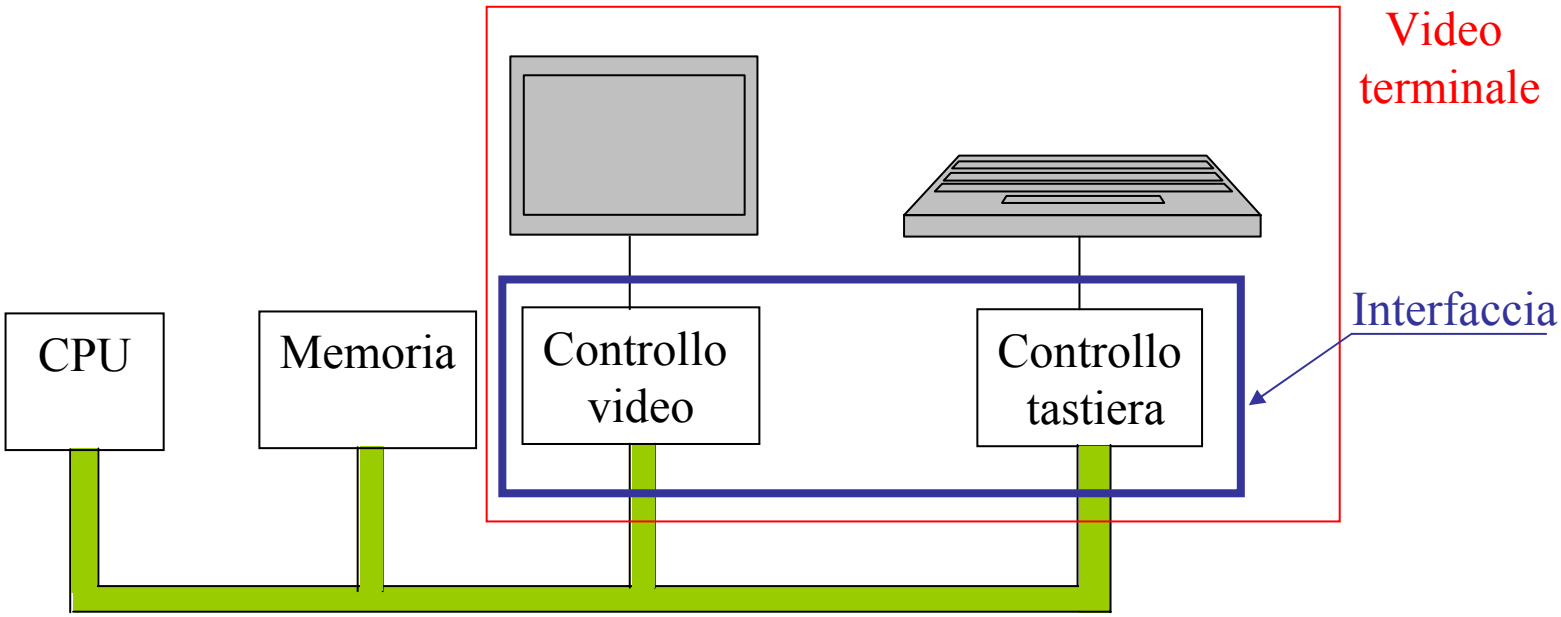
## GESTIONE DELL'INPUT/OUTPUT

*Massimiliano Giacomin*




*ISA: Instruction Set Architecture

# Una struttura a bus singolo



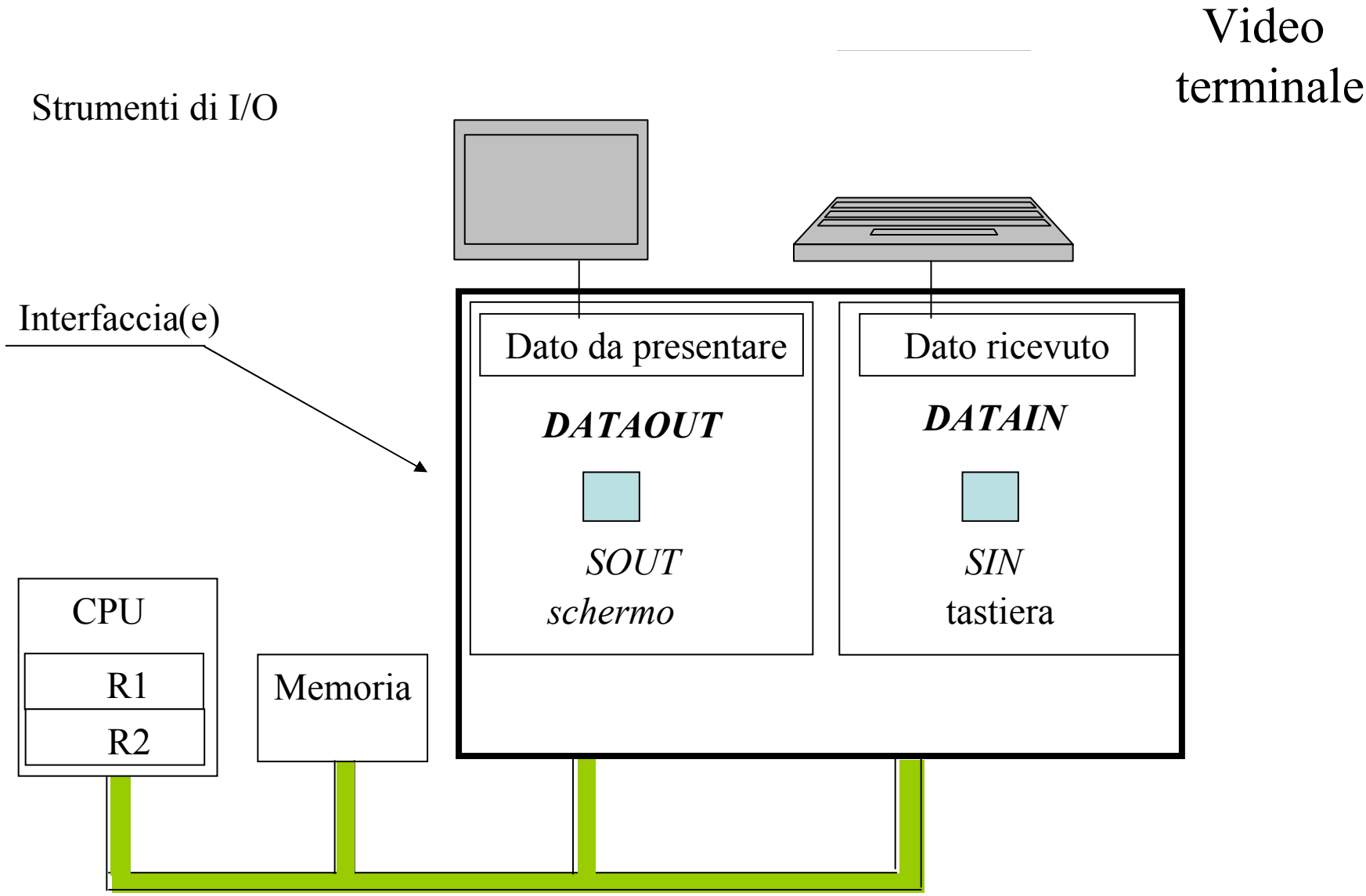

**Bus:** fascio di conduttori elettrici che connettono diversi apparecchi.

I bus (esterni ) sono specificati in modo da permettere il collegamento di CPU, memorie e apparecchi di I/O costruiti da produttori diversi

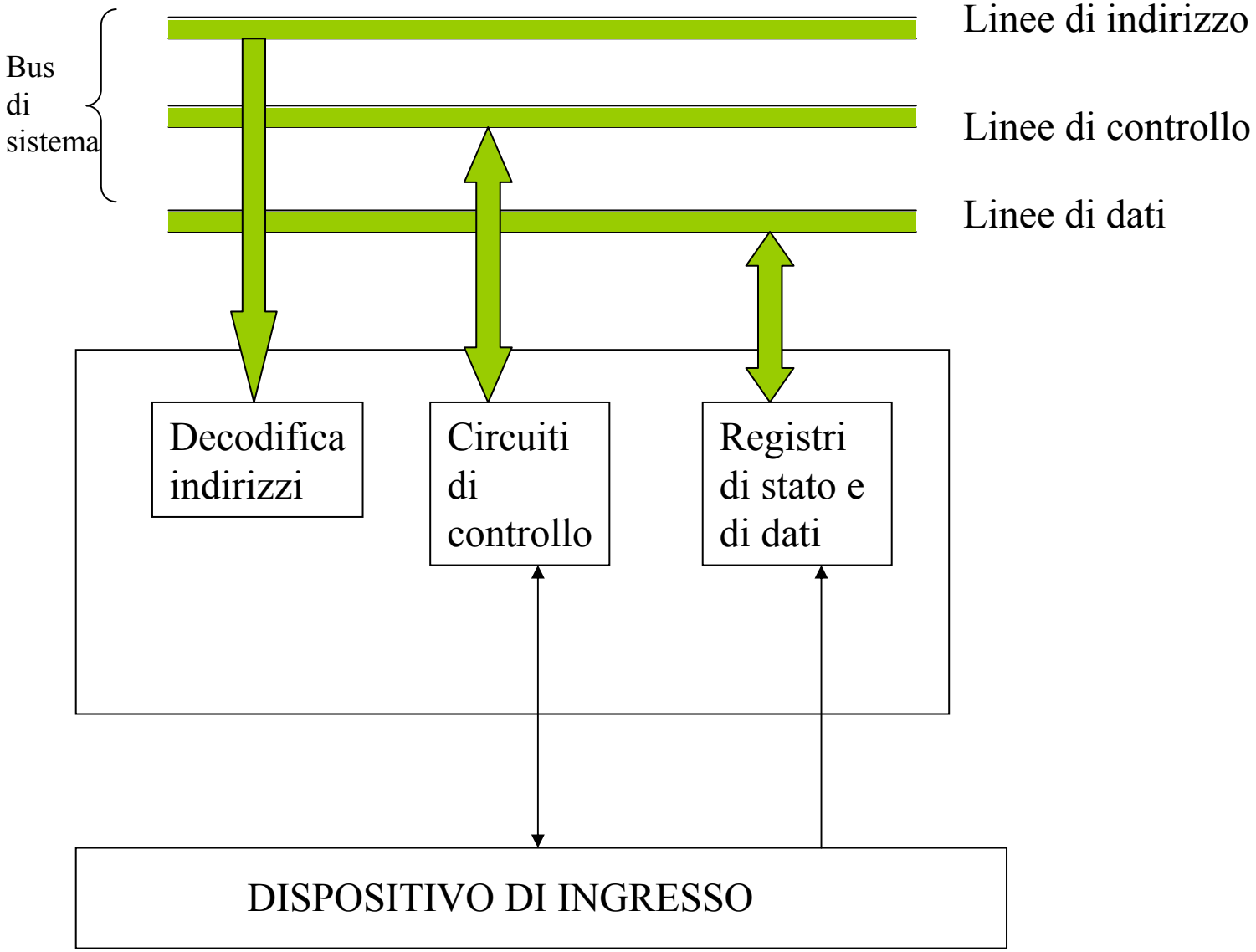
**3 gruppi di linee**:

- dati
- indirizzi
- segnali di controllo

# Strumenti di I/O e interfacce



# Dispositivo di ingresso e interfaccia



Il processore è in grado di leggere e scrivere i registri dei dati e di stato dei dispositivi; il circuito di controllo è in grado di gestire l'interazione con il dispositivo, ad esempio acquisendo il dato dal dispositivo e ponendolo in un registro DATAIN.

PROBLEMA

- Come far sì che un indirizzo venga riconosciuto dal dispositivo e non dalla memoria? [ovvero: spazio indirizzamento memoria vs. I/O]

Due soluzioni:

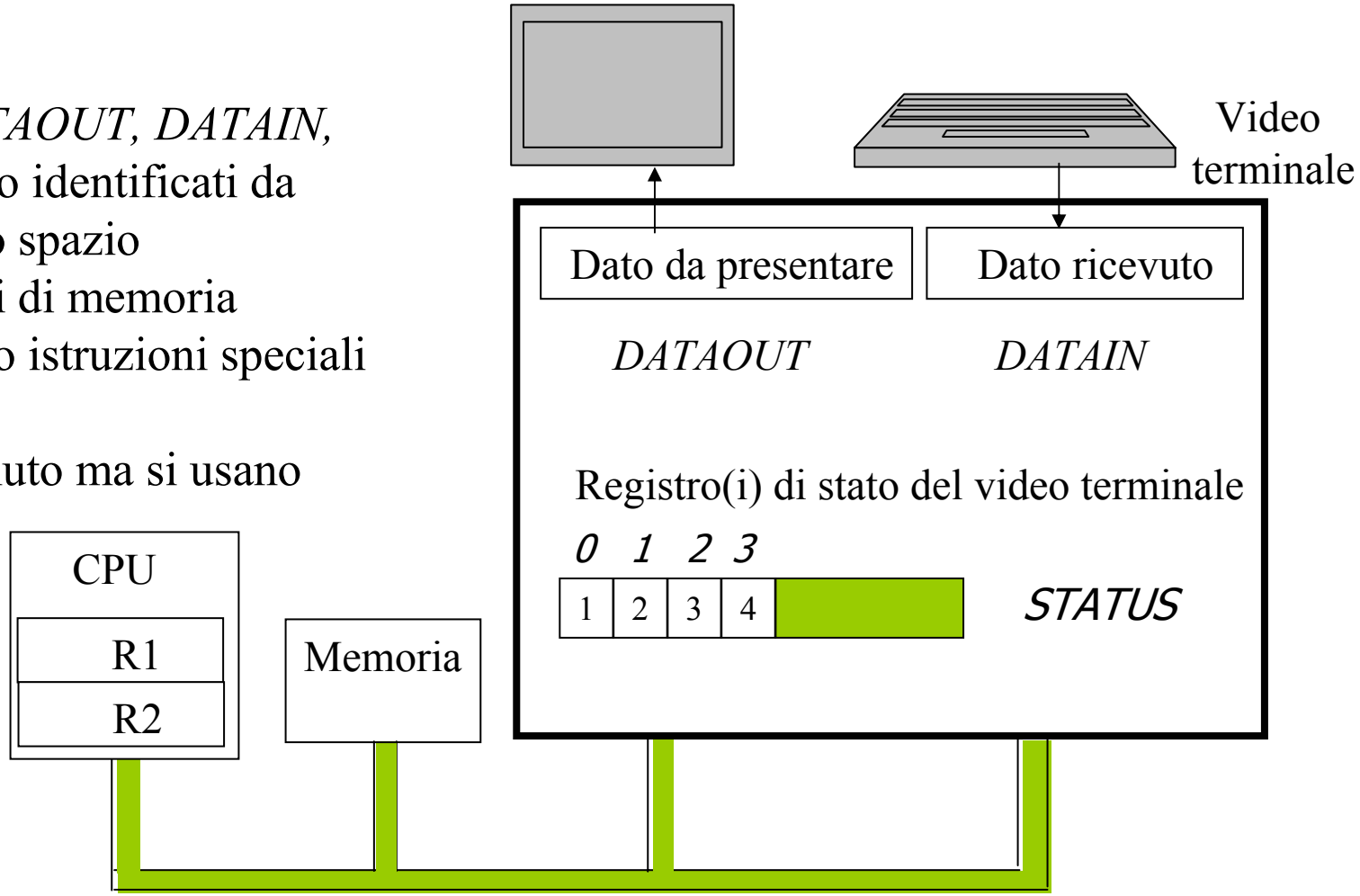
1. **Memory mapped**:
   - I registri dei buffer hanno indirizzi che appartengono allo spazio indirizzi di memoria.
   - I registri del buffer della periferica sono indirizzati come se fossero locazioni di memoria: uso le istruzioni per il normale trasferimento tra registri e memoria per trasferire dati dai buffer delle periferiche ai registri di CPU (e memoria centrale) o viceversa.
   - La memoria "ignora" indirizzi che non corrispondono a proprie locazioni
2. **Istruzioni speciali di I/O**:
   - Generano opportuni segnali sulle linee di controllo
     $\Rightarrow$ spazio di indirizzi di I/O distinto dallo spazio indirizzi di memoria

# Gestire l'I/O : approccio memory mapped

Come si leggono e scrivono sul video caratteri quando gli interrupt siano non abilitati e le istruzioni sono 'memory mapped '

I registri *DATAOUT, DATAIN, STATUS* sono identificati da indirizzi nello spazio degli indirizzi di memoria

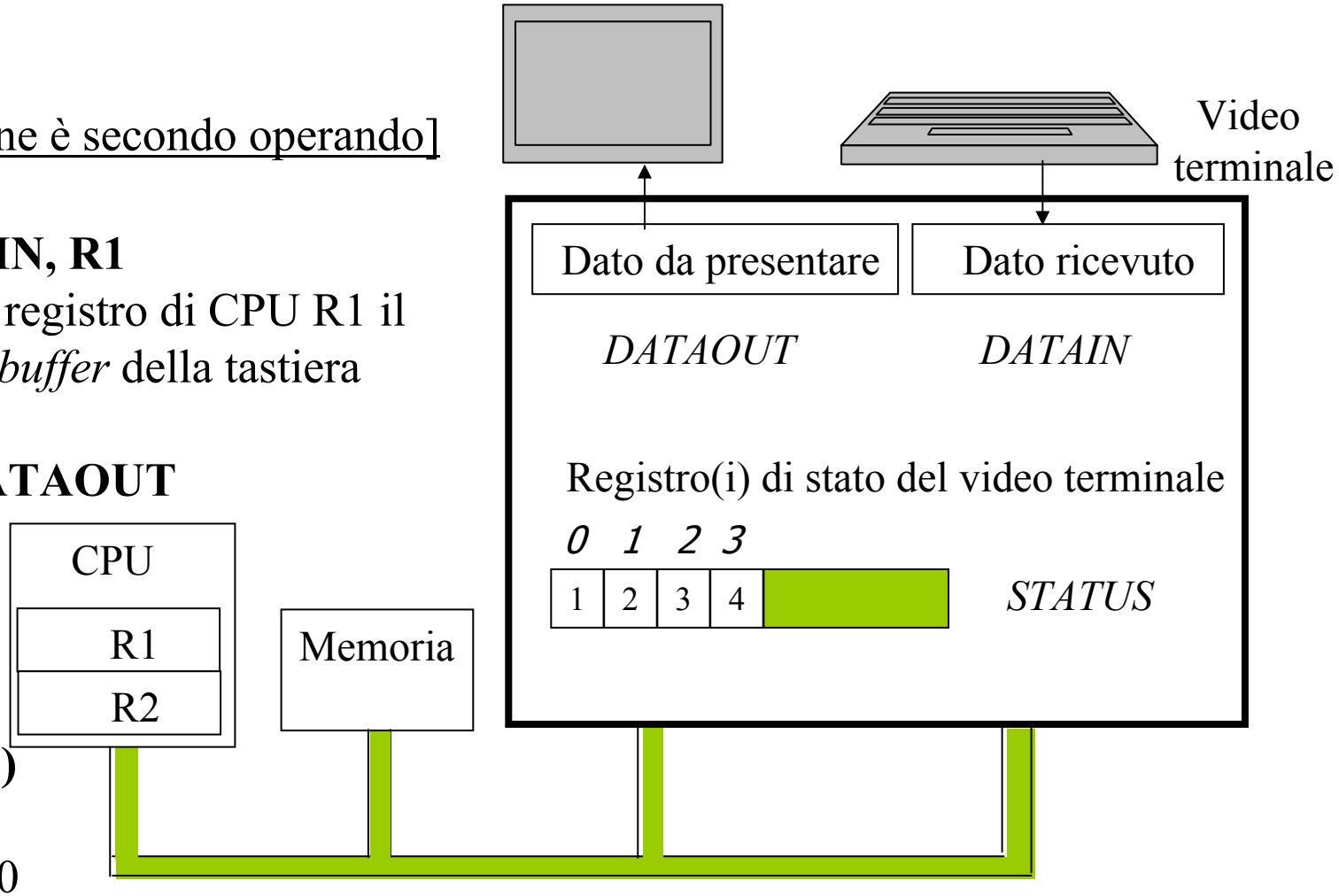
Non occorrono istruzioni speciali per accedere al loro contenuto ma si usano istruzioni come Load, Store, Move



# Gestire l'I/O : approccio memory mapped

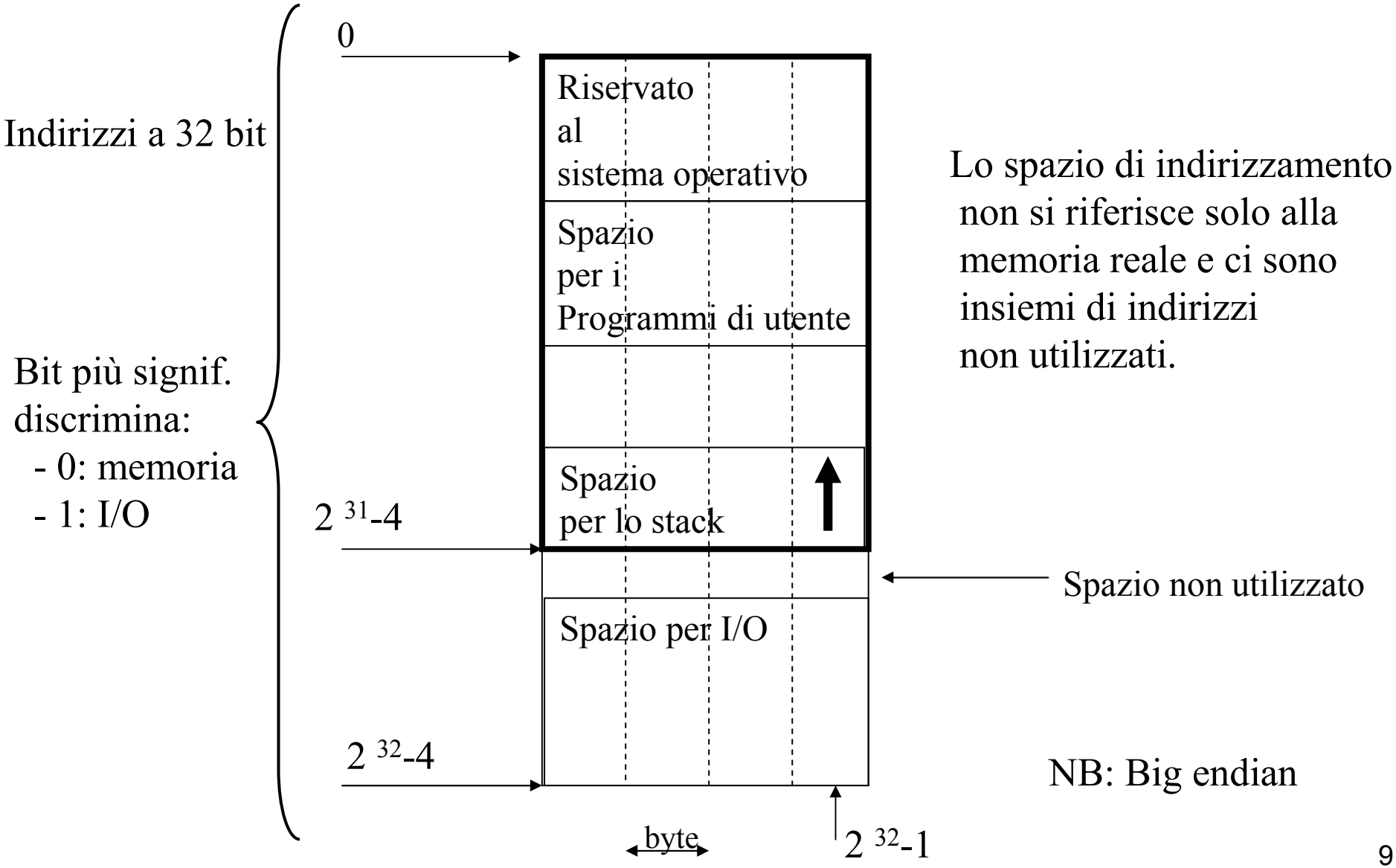
**Es.** [destinazione è secondo operando]

- **Move DATAIN, R1**
  trasferisce nel registro di CPU R1 il contenuto del *buffer* della tastiera *DATAIN*
- **Move R1, DATAOUT**
  trasferisce il contenuto di R1 in DATAOUT.
- **Move R0,(R1)**
  trasferisce il contenuto di R0 nella cella di memoria il cui indirizzo è contenuto in R1.



# Esempio di spazio di indirizzamento

con I/O memory mapped



# Interfaccia I/O: alcuni dettagli

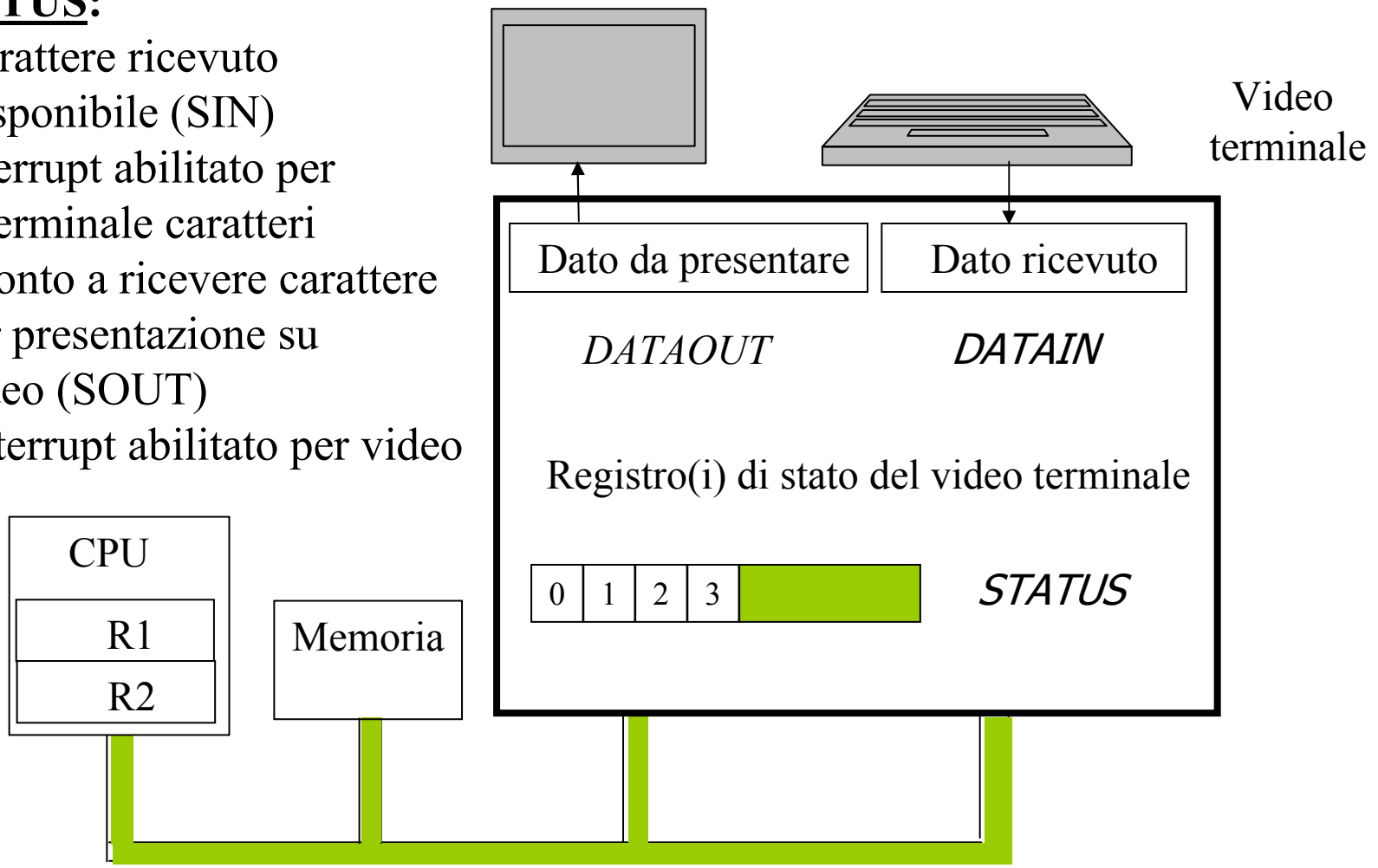
**STATUS:**

**0:** carattere ricevuto disponibile (SIN)

**1**:interrupt abilitato per il terminale caratteri

**2**: pronto a ricevere carattere per presentazione su video (SOUT)

**3:** interrupt abilitato per video



# Interfaccia I/O: alcuni dettagli (continua)

Funzionamento

SIN/SOUT segnalano quando il dispositivo:

- ha prodotto un dato nel registro DATAIN disponibile per l'input (SIN)
  [dispositivo di input: tastiera]
- è in grado di ricevere in DATAOUT un dato disponibile per output (SOUT)
  [dispositivo di output: terminale video]

Nel momento in cui CPU referenzia i registri dati per effettuare il trasferimento, i flag di controllo SIN o SOUT sono automaticamente disabilitati:

- lettura DATAIN: circuito di controllo pone SIN=0 e procede ad "aspettare" un nuovo dato dalla tastiera
- scrittura DATAOUT: circuito di controllo pone SOUT=0 e procede ad effettuare l'output al dispositivo video

Modalità di gestione e sincronizzazione con i dispositivi

- **A controllo di programma (o "programmato")**
- **Interrupt**
- **Accesso Diretto alla Memoria** (Direct Memory Access, DMA)

# I/O controllato da programma (1)

```
READWAIT        Branch to READWAIT if  SIN=0
                Input from DATAIN  to R1

WRITEWAIT       Branch to WRITEWAIT if  SOUT=0
                Output from R1  to  DATAOUT
```

**Architettura  memory mapped**

Assembler Motorola 6800 like [branch realizzato da due istruzioni dove #0 indica il bit *b0* di un registro IOSTATUS che corrisponde a SIN e #2 indica il bit *b2* che corrisponde a SOUT.

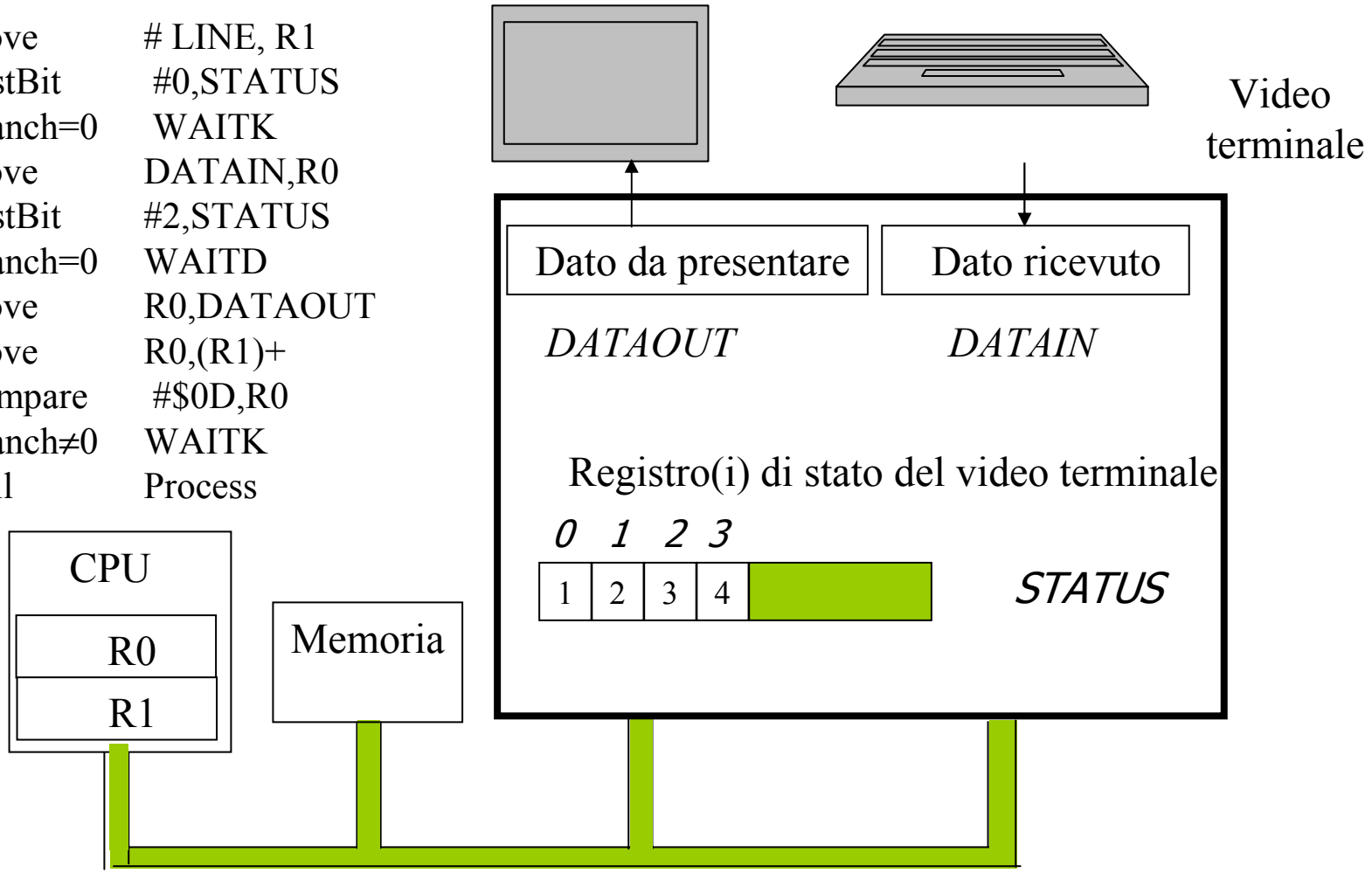
```
READWAIT        Testbit         #0,IOSTATUS
                Branch=0        READWAIT
                Move            DATAIN,R1

WRITEWAIT       Testbit         #2,IOSTATUS
                Branch=0        WRITEWAIT
                Move            R1,DATAOUT
```

# I/O controllato da programma (2)

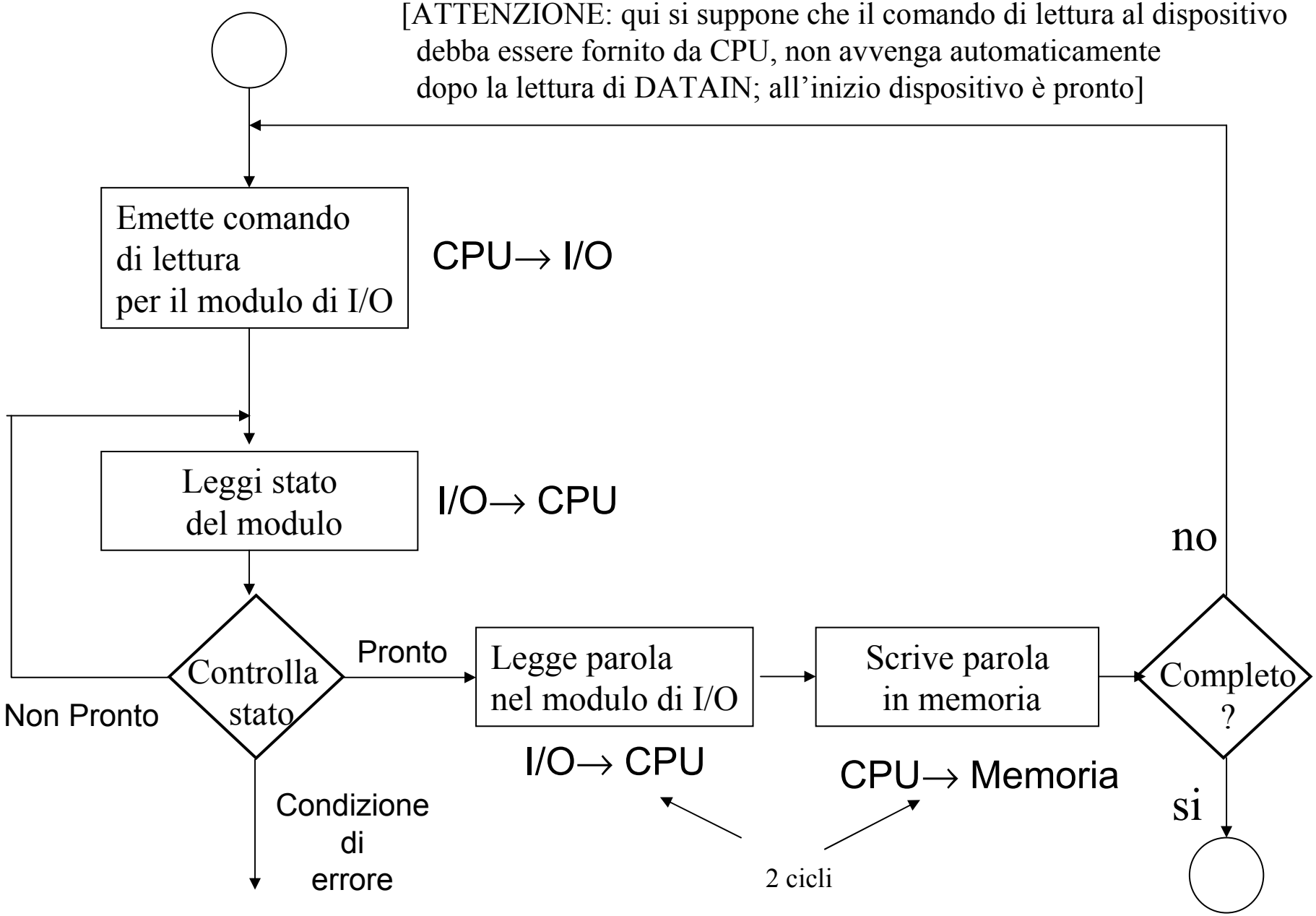
Programma che legge una linea di caratteri [termina con il carattere di ritorno a capo #$0D] e la presenta sul video (*echoback*) nell'ipotesi che gli interrupt siano non abilitati

```
        Move        # LINE, R1
WAITK   TestBit     #0,STATUS
        Branch=0    WAITK
        Move        DATAIN,R0
WAITD   TestBit     #2,STATUS
        Branch=0    WAITD
        Move        R0,DATAOUT
        Move        R0,(R1)+
        Compare     #$0D,R0
        Branch≠0    WAITK
        Call        Process
```



# Schema tipo per la lettura di dati

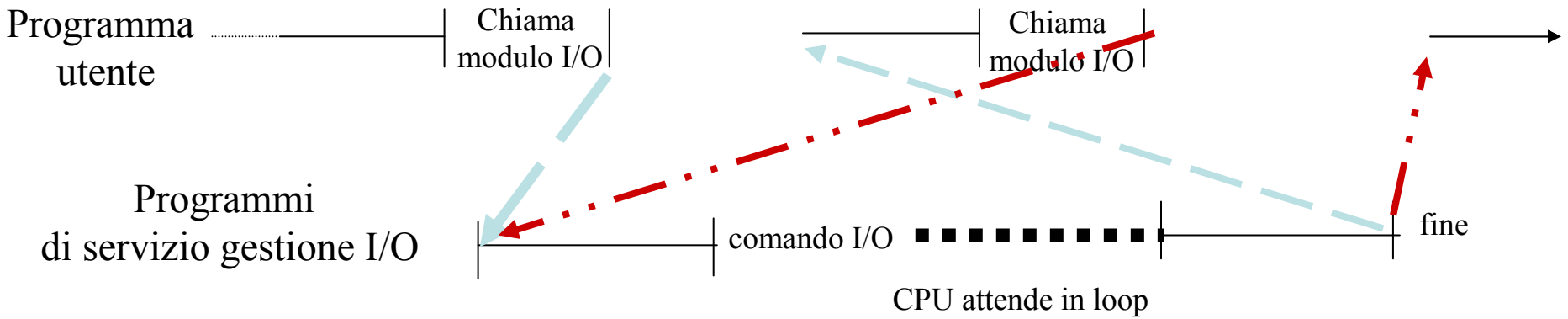
[ATTENZIONE: qui si suppone che il comando di lettura al dispositivo debba essere fornito da CPU, non avvenga automaticamente dopo la lettura di DATAIN; all'inizio dispositivo è pronto]



# I/O Programmato: flusso del programma

## Esempio con modulo di programma che gestisce I/O

C'è corrispondenza fra istruzioni di I/O nel programma ed i comandi di I/O inviati al modulo



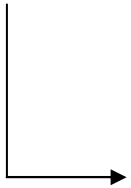

La frecce indicano diramazioni nella sequenza di istruzioni da eseguire

# I/O con interruzioni (interrupt)

- I/O a controllo di programma: cicli di attesa del processore

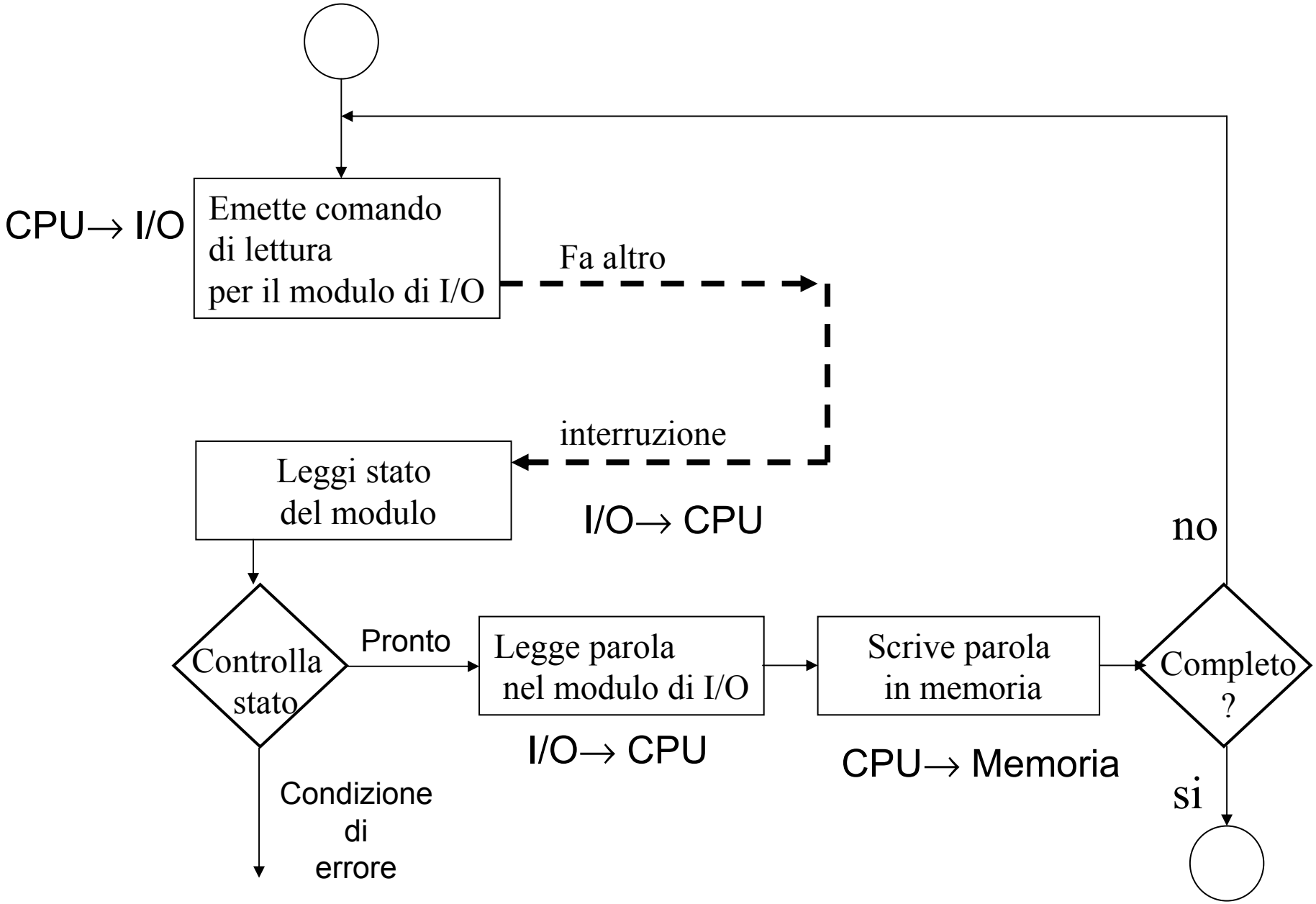
  $\Rightarrow$ tempo sprecato: il processore potrebbe invece fare del lavoro “utile”

IDEA: il processore non controlla il dispositivo: procede con del lavoro “utile”
e viene avvisato dal dispositivo tramite un interrupt



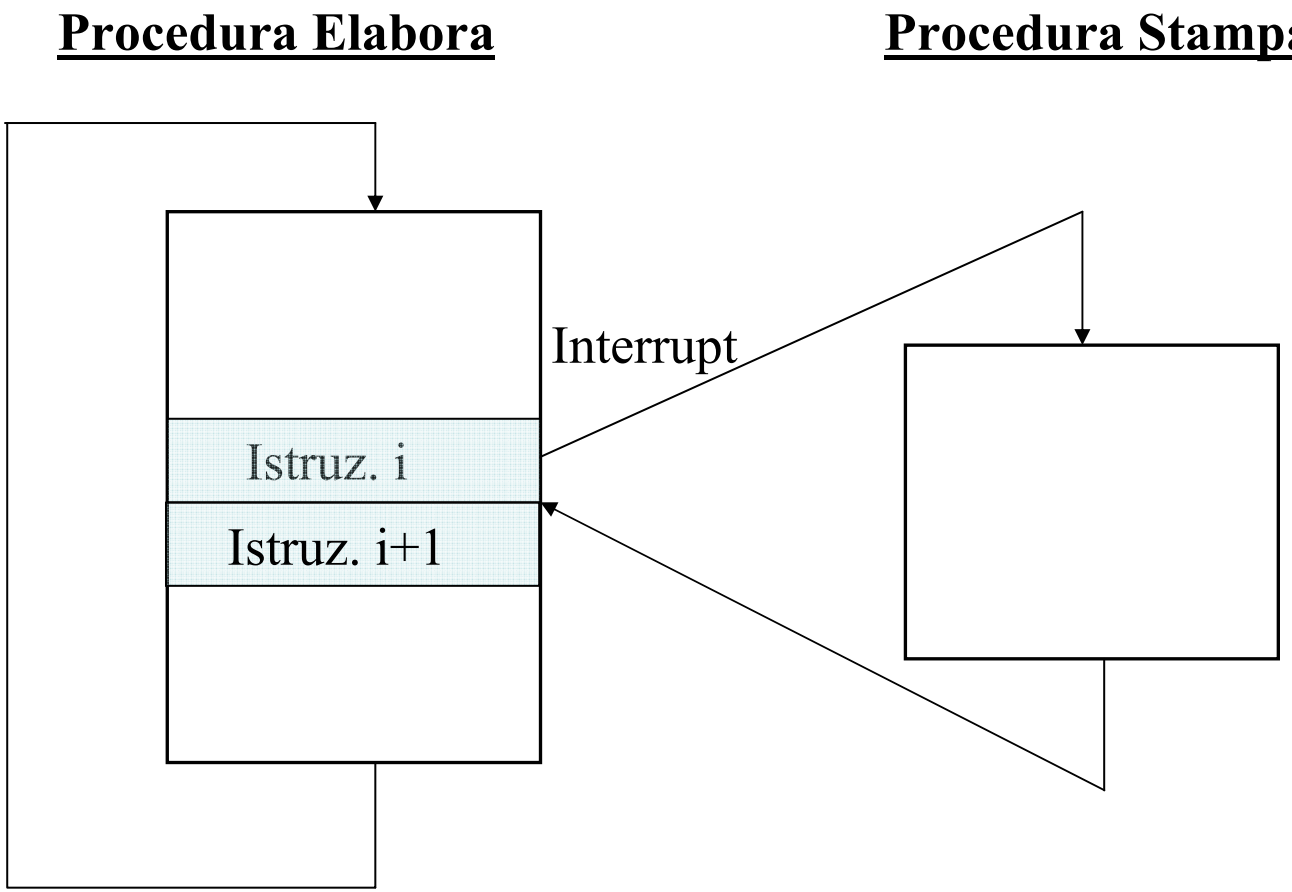
Richiede linea di controllo apposita nel bus:
INTR [Interrupt Request]

**Vediamo come si modifica l'ultimo schema…**



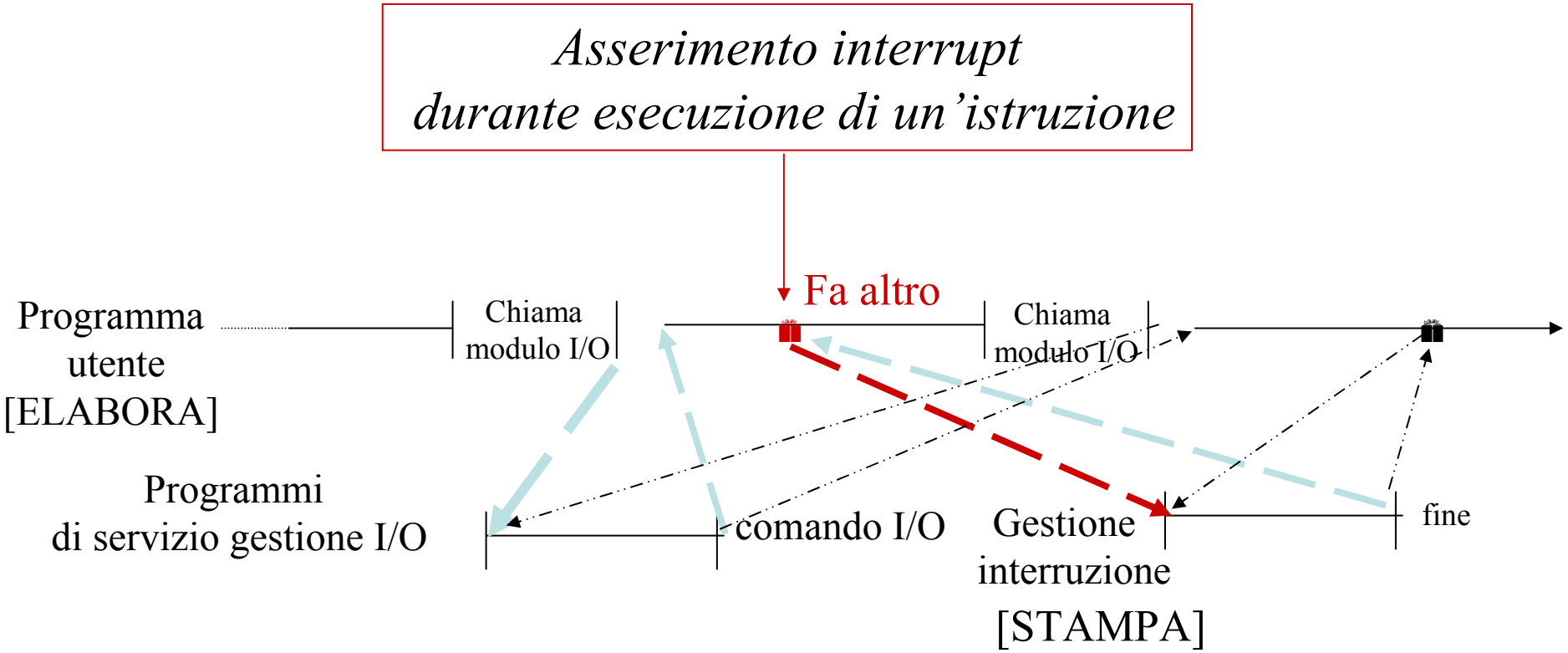
**UN ESEMPIO**: stampa di righe prodotte da una procedura ELABORA

- STAMPA: procedura che invia alla stampante 1 riga alla volta
- ELABORA: continua ad elaborare righe riempiendo un buffer
  (resta in attesa se il buffer è pieno)



FLUSSO DEL PROGRAMMA

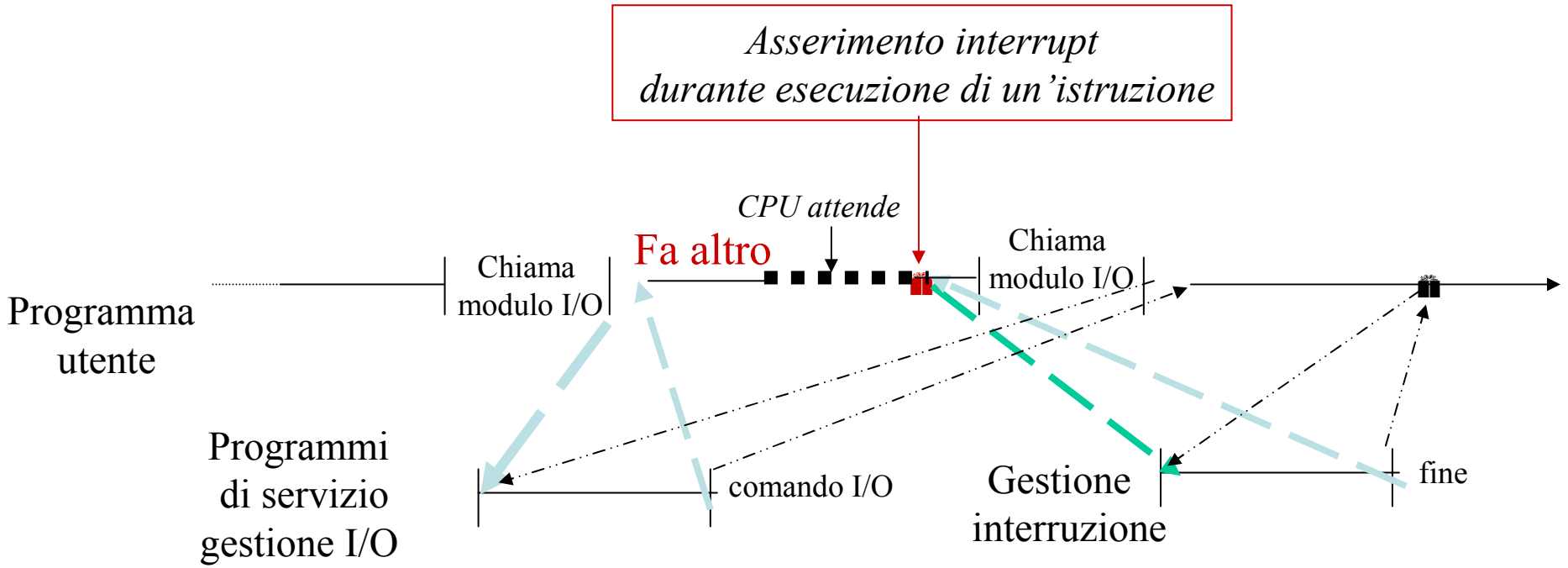
a) Con attesa **breve** per l'operazione di I/O
[ELABORA impiega più tempo per generare righe di quanto serve per stamparle]




[Le frecce indicano diramazioni nella sequenza di istruzioni da eseguire]

FLUSSO DEL PROGRAMMA

a) Con attesa **lunga** per l'operazione di I/O
[ELABORA impiega meno tempo per generare righe di quanto serve per stamparle]




[Le frecce indicano diramazioni nella sequenza di istruzioni da eseguire]

GESTIONE DELL'INTERRUPT

- Il dispositivo richiede interruzione con segnale di controllo su bus *interrupt request* [INTR]
- Il processore risponde con *segnale di riscontro* INTA [Interrupt Acknowledge]
- Ricevuto questo segnale, il dispositivo toglie il segnale INTR

A questo punto: il processore deve passare il controllo alla *procedura di servizio dell'interrupt [Interrupt Service Routine ISR]*

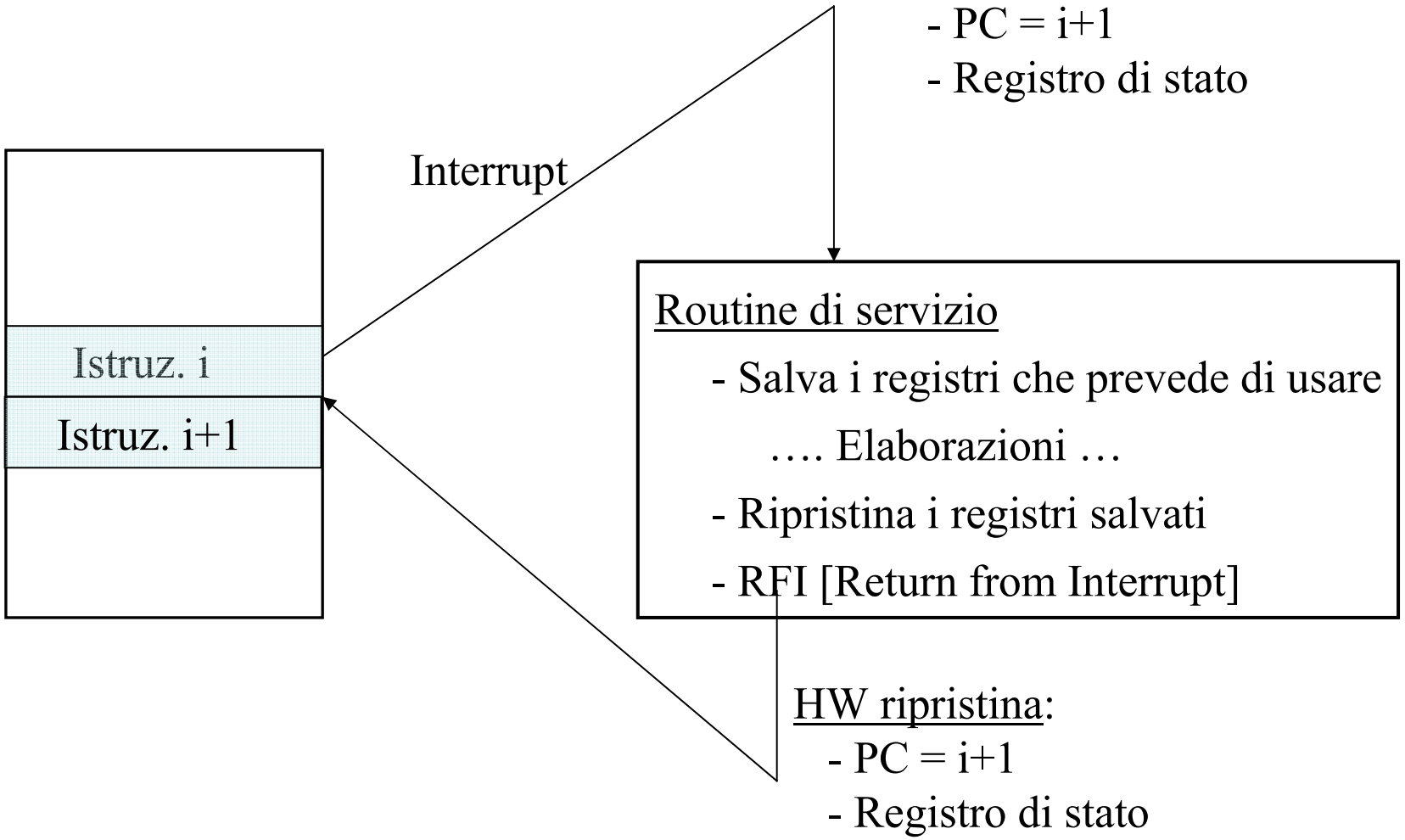
Terminata la procedura: il processore restituisce il controllo alla procedura originaria [istruzione i+1]

**NB**: differenza con procedura: il programma interrotto non "sa" cosa viene modificato – cambia il concetto di "responsabilità",
es. registri temporanei devono essere ripristinati!

⇒ HW deve salvare almeno:
- Program Counter [modificato cedendo il controllo]
- **Registro di stato** del processore [**PS**]

Di solito, al resto pensa la routine di servizio!

**QUINDI:**


HW salva:
- PC = i+1
- Registro di stato
Interrupt
Istruz. i
Istruz. i+1
Routine di servizio
- Salva i registri che prevede di usare
…. Elaborazioni …
- Ripristina i registri salvati
- RFI [Return from Interrupt]
HW ripristina:
- PC = i+1
- Registro di stato

## INTERRUPT: i problemi da gestire

- Come gestire il fatto che il dispositivo continua a richiedere interrupt finchè il processore non gli risponde?
  [problema delle continue interruzioni che generano ciclo infinito]
- Come identificare il dispositivo?
- Come gestire le richieste di interrupt simultanee?
- Una procedura di servizio dell'interrupt può essere a sua volta interrotta?
  Se sì, come gestire le interruzioni annidate?

**Abilitazione e disabilitazione degli interrupt**

Scenario tipico per risolvere il problema

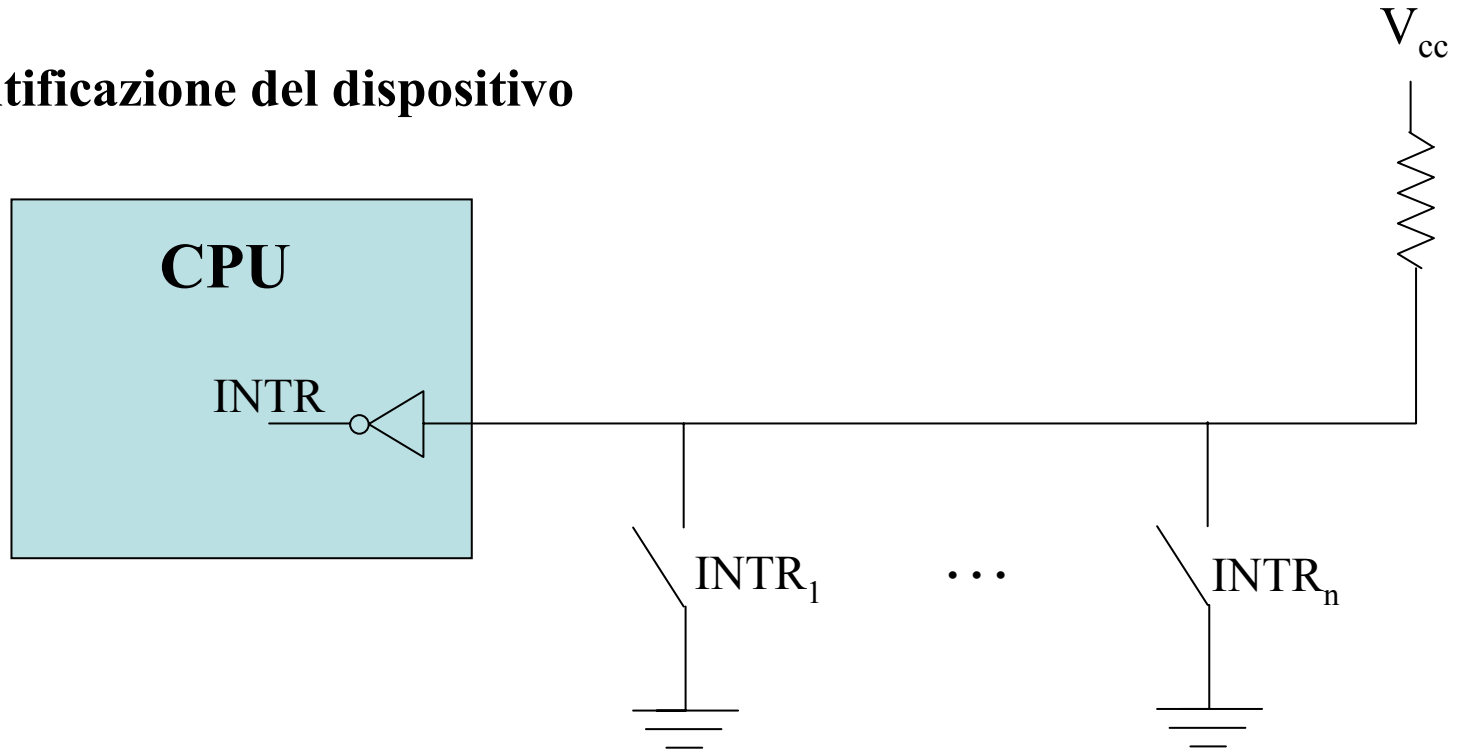
1. Dispositivo attiva INTR
2. Processore interrompe il programma
3. Gli interrupt vengono disabilitati: due metodi [principali] diversi di farlo
   - processore ignora successivi INTR fino alla fine della prima istruzione ISR, che è sempre istruzione di disabilitazione interrupt
     [verranno riabilitati da un'istruzione apposita alla fine della procedura]
   - processore, dopo aver salvato PS, disabilita interrupt ponendo a 1 flag in PS
     [l'istruzione di ritorno da interrupt automaticamente riabilita interrupt ripristinando PS]
4. Si informa dispositivo via INTA che interrupt è accettato
5. Esecuzione procedura ISR
6. Ripristino esecuzione programma interrotto con riabilitazione interrupt

NB: deve anche essere possibile disabilitare interrupt del singolo dispositivo

Se un dispositivo è inattivo, non deve interrompere processore, anche se è pronto ad eseguire un trasferimento

⇒ Come visto, è sempre presente un bit opportuno in un registro dell'interfaccia del dispositivo

## Identificazione del dispositivo




$INTR = INTR_1 + INTR_2 + \ldots INTR_n$

**<u>Metodo 1</u>: Polling (Interrogazione)**

- ISR controlla i bit IRQ in tutte le interfacce dei dispositivi
- Il primo che ha IRQ=1 viene servito

⇒ Facile da realizzare, ma spreca tempo

**<u>Metodo 2</u>: Interrupt vettorizzati**

- dispositivo: INTR
- CPU: INTA [segnala che è pronto a trattare l'interrupt]
- dispositivo: invia un codice di identificazione attraverso linee dati del bus (vettore di identificazione) e interrompe INTR
- CPU accede ad una tabella con corrispondenza codice → indirizzo $ISR_{dispositivo}$

**NB**: finchè INTA non è attivato, il dispositivo non ha il diritto di accedere al bus
[quando arriva INTR, può esserci istruzione in fase di esecuzione, oppure gli interrupt potrebbero essere disabilitati durante una routine di servizio]

**NB2**: sono anche possibili schemi misti
[gruppi di dispositivi riconosciuti con vettori, all'interno di un gruppo uso del polling]

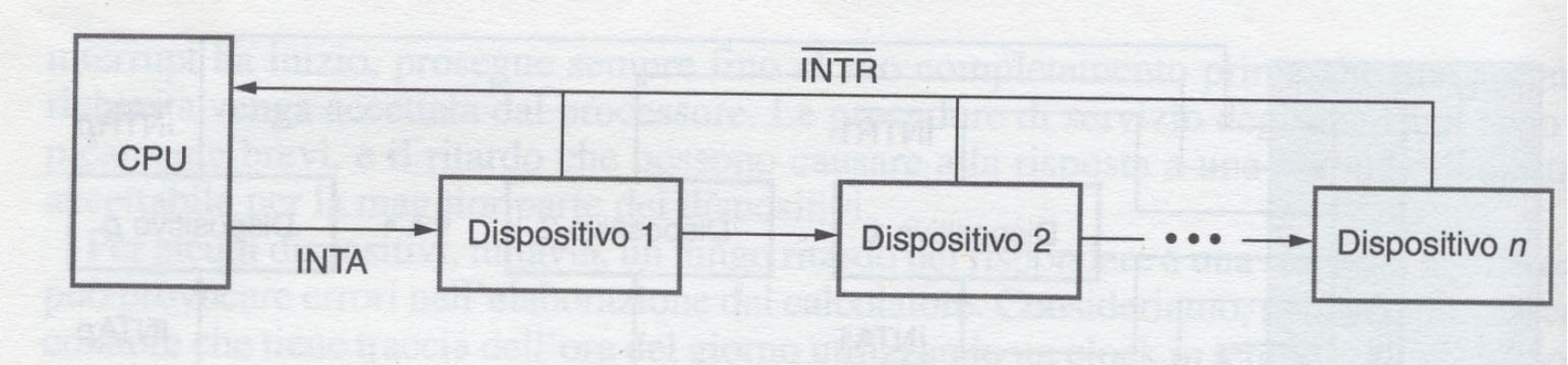
## Interruzioni simultanee da parte di più dispositivi

Caso in cui si usa polling per identificare i dispositivi

Ordine di interrogazione dei dispositivi determina la priorità

Caso con interrupt vettorizzato

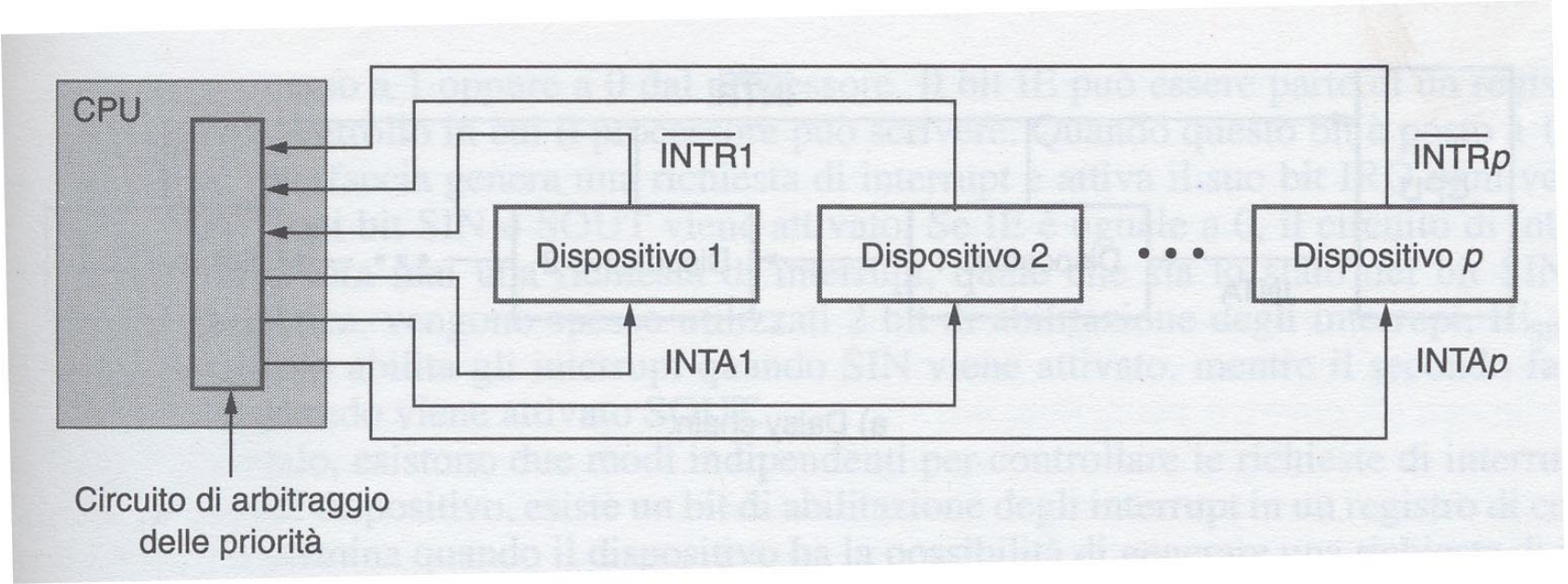
1) **Schema daisy chain**




- Dispositivo non ha richiesto interrupt: propaga INTA ai successivi
- Dispositivo ha richiesto interrupt: blocca propagazione INTA e risponde ponendo nel bus il codice identificativo

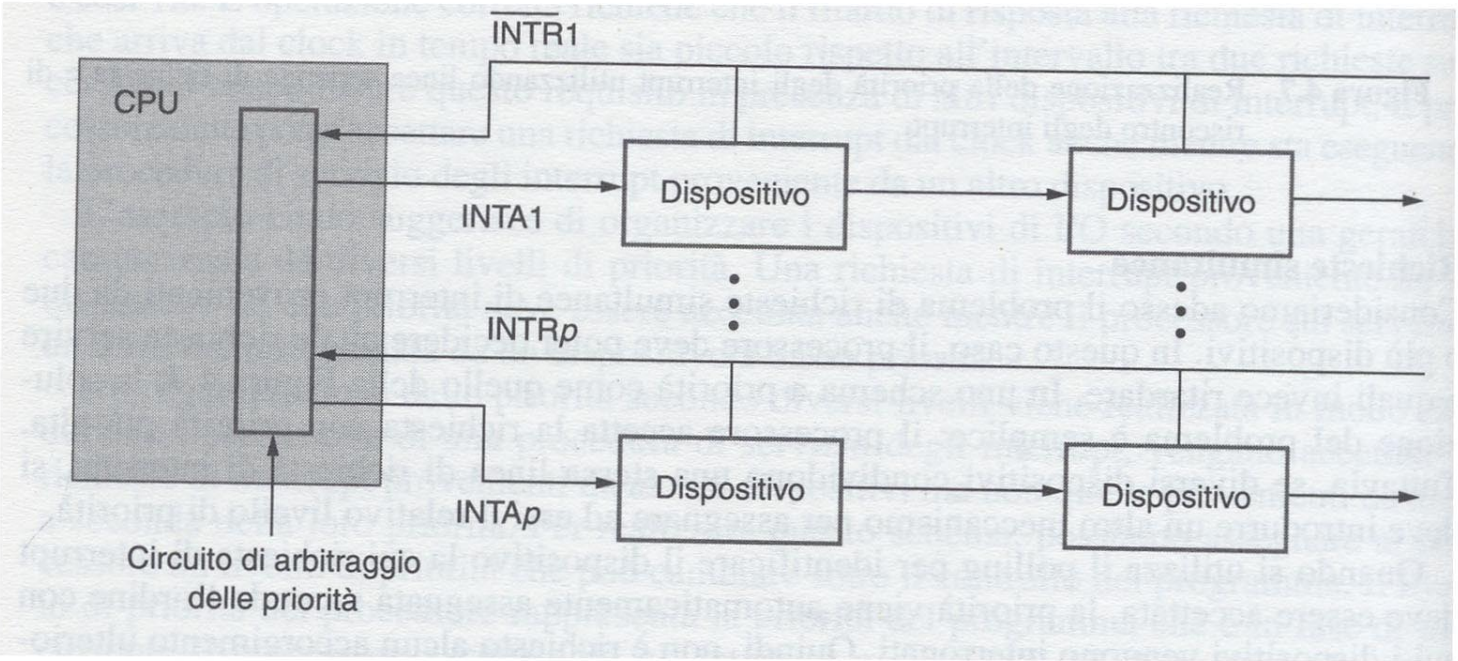
⇒ Il più vicino ha priorità maggiore

## 2) **Schema con priorità**



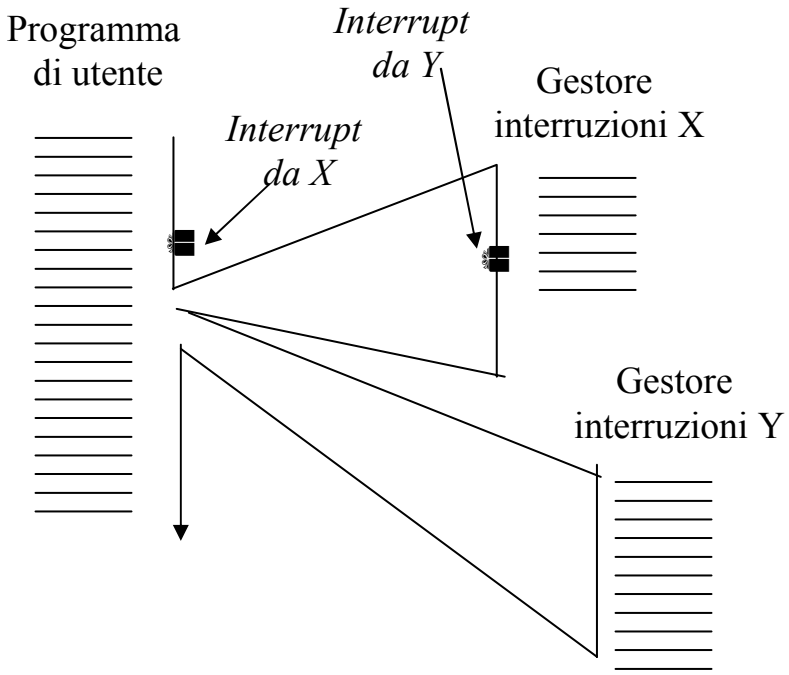

- Ogni linea ha una diversa priorità
- Servita solo la linea a priorità più alta (le altre rimangono pendenti)

3) **Schema misto**




- Servita la linea a priorità più alta e, tra i suoi dispositivi, il più vicino tra quelli che richiedono il servizio
- Possibilità per un dispositivo di essere collegato a diversi livelli di priorità: $INTR_i$ compatibile con l'urgenza del servizio richiesto

# Interruzioni annidate




Uno schema semplificato è quello visto in precedenza:

nel servire interruzione, vengono disabilitate le interruzioni:

priorità, al più, gestita nel caso di richieste simultanee

PROBLEMA: Le richieste prioritarie possono ragionevolmente dover interrompere le altre (es. orologio)

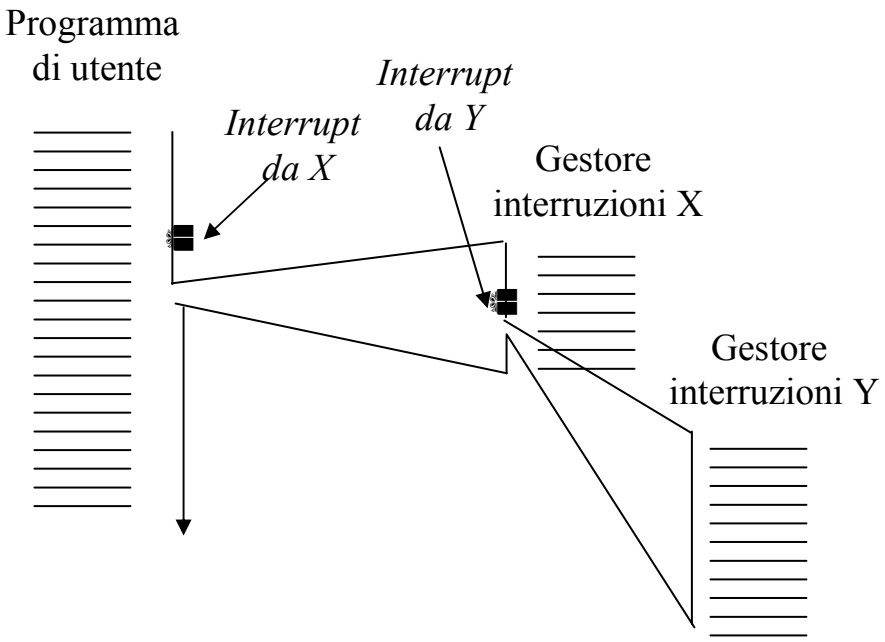
## Interruzioni annidate (2)

Si usa uno schema con più linee di priorità [cfr. lucidi precedenti]
e si consente ad interruzioni con priorità più alta di interrompere processore se sta servendo quelle a priorità più bassa.

Funzionamento (è un'estensione dello schema visto in precedenza)

- Registro PS (Process Status) mantiene in pochi bit la priorità corrente del processore.
- Il processore può essere interrotto soltanto da linee a priorità strettamente maggiore rispetto a priorità corrente.
- In tal caso, il processore salva PS e modifica PS portando la priorità pari a quella del dispositivo $\Rightarrow$ processore sarà interrotto solo da priorità maggiori
- Nel ritorno dalla procedura di servizio (ISR) PS è automaticamente ripristinato
  $\Rightarrow$ processore ritorna alla priorità originaria

# Interruzioni annidate (3)




PROBLEMA: come gestire il salvataggio del contesto [PC, PS e altri registri]?

⇒ HW e SW hanno la responsabilità verso il programma interrotto (che non ha "idea" di essere stato interrotto) di fargli ritrovare tutto com'era prima

## Interruzioni annidate (4)

UNA SOLUZIONE

Al verificarsi di un Interrupt:

- HW salva PC e PS nello stack e cede controllo a ISR
  alterando PC e PS (nuova priorità)
- ISR salva i registri che usa nello stack [responsabilità verso interrotto]
  …. elaborazione …
- Dopo aver eseguito il compito, ISR ripristina i registri prelevandoli da stack
- Istruzione "Return from Interrupt" ripristina via HW registri PC e PS

**NB**: cosa succede se ISR è interrotta in un qualsiasi momento?
Funziona perché IRS_nuova non cambia il contesto
(ha responsabilità verso ISR) e in particolare restituisce registri e stack originari!

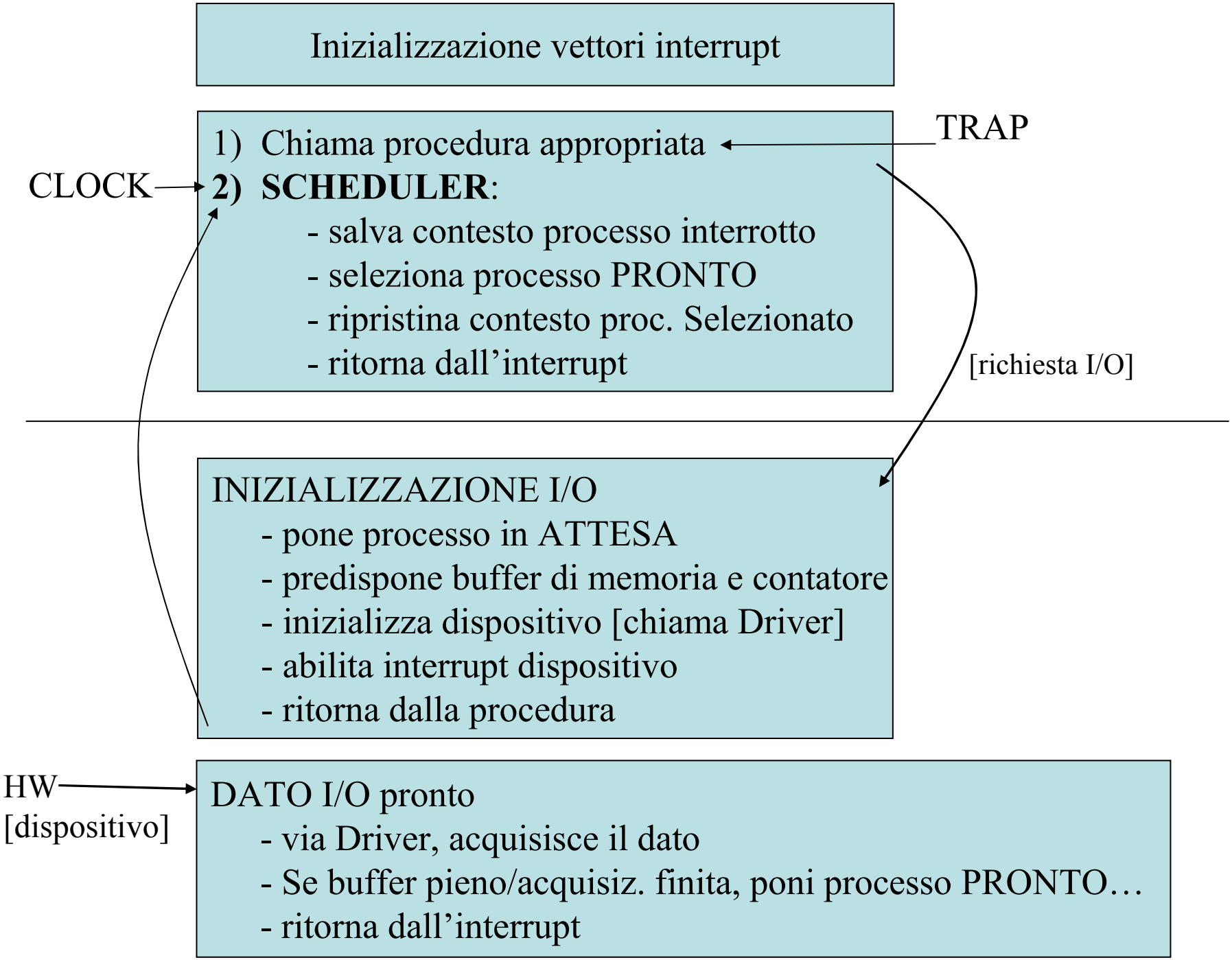
**NB2**: sono possibili varianti allo schema.
P.es. nel POWERPC (cfr. Hamacher, par. 4.3) il processore salva PC di ritorno
e PS in due registri speciali, disabilitando gli interrupt
⇒ ISR deve salvare questi registri nello stack prima di riabilitare
gli interrupt

# UTILIZZO DEGLI INTERRUPT NEI SISTEMI OPERATIVI

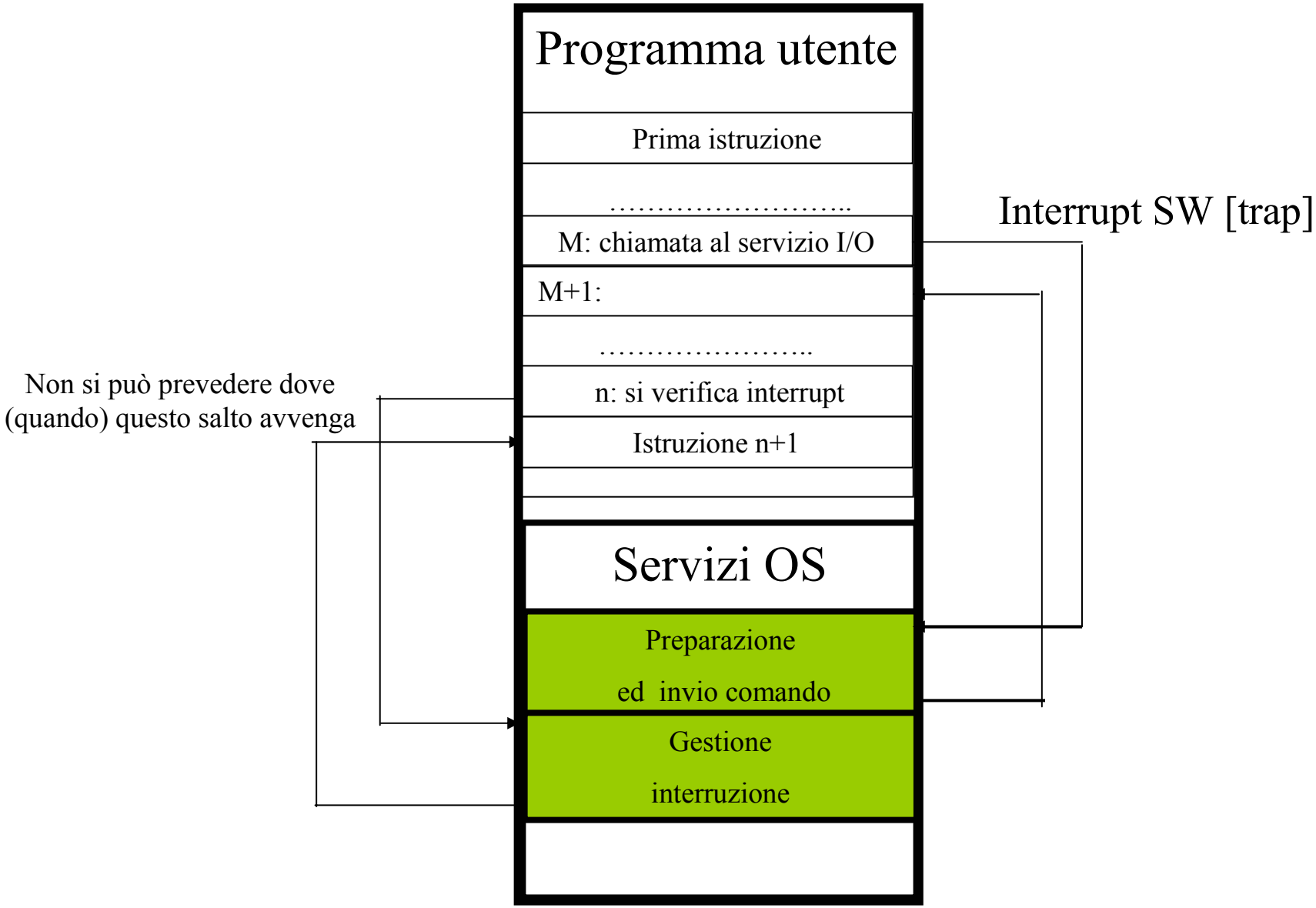
**[CENNI!]**

- Sono presenti molti processi = programma + contesto (info su stato corrente)
- E' presente un clock HW che genera un interrupt ogni periodo:
  partizione di tempo (suddivisione tra i processi)
- Due stati per il processore: utente vs. supervisore
  - Esecuzione processi utente: stato utente
  - Interrupt HW o Interrupt SW [Trap] cede controllo a S.O. e porta nello stato supervisore [NB: al solito PS aggiornato ed il vecchio PS salvato…]
- Il contesto di ogni processo (PS, PC, registri, registro base tabella delle pagine, ecc.) è mantenuto dal S.O. in un'apposita struttura SW
- Ogni processo è in stato ESECUZIONE | PRONTO | ATTESA

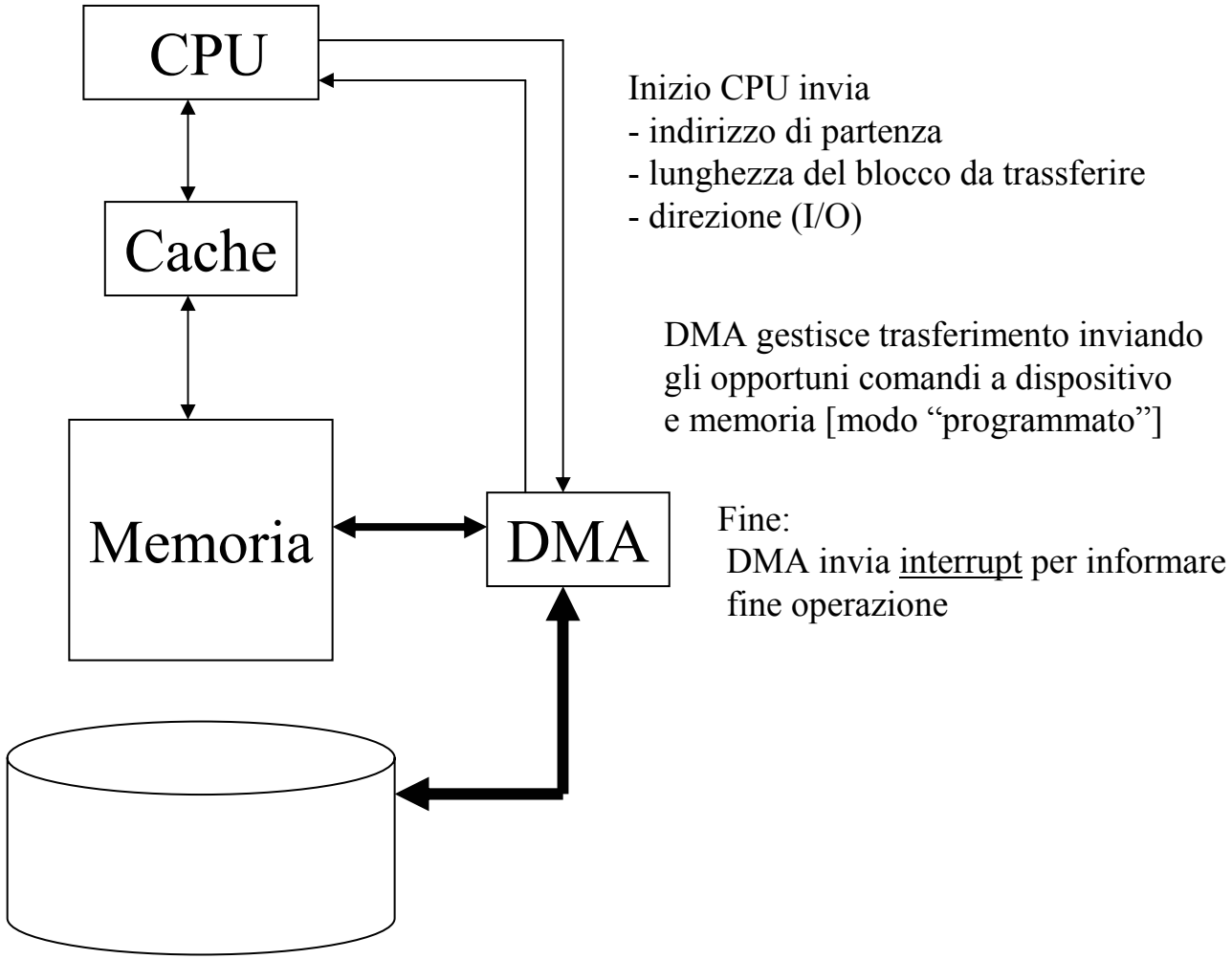
Vediamo una struttura ipotetica
[solo per dare l'idea!!! Ignorati molti dettagli]


Inizializzazione vettori interrupt
1) Chiama procedura appropriata
2) SCHEDULER:
- salva contesto processo interrotto
- seleziona processo PRONTO
- ripristina contesto proc. Selezionato
- ritorna dall'interrupt
TRAP
CLOCK
[richiesta I/O]
INIZIALIZZAZIONE I/O
- pone processo in ATTESA
- predispone buffer di memoria e contatore
- inizializza dispositivo [chiama Driver]
- abilita interrupt dispositivo
- ritorna dalla procedura
HW
[dispositivo]
DATO I/O pronto
- via Driver, acquisisce il dato
- Se buffer pieno/acquisiz. finita, poni processo PRONTO…
- ritorna dall'interrupt

# Dettaglio gestione I/O…



# DMA (Direct Memory Access)

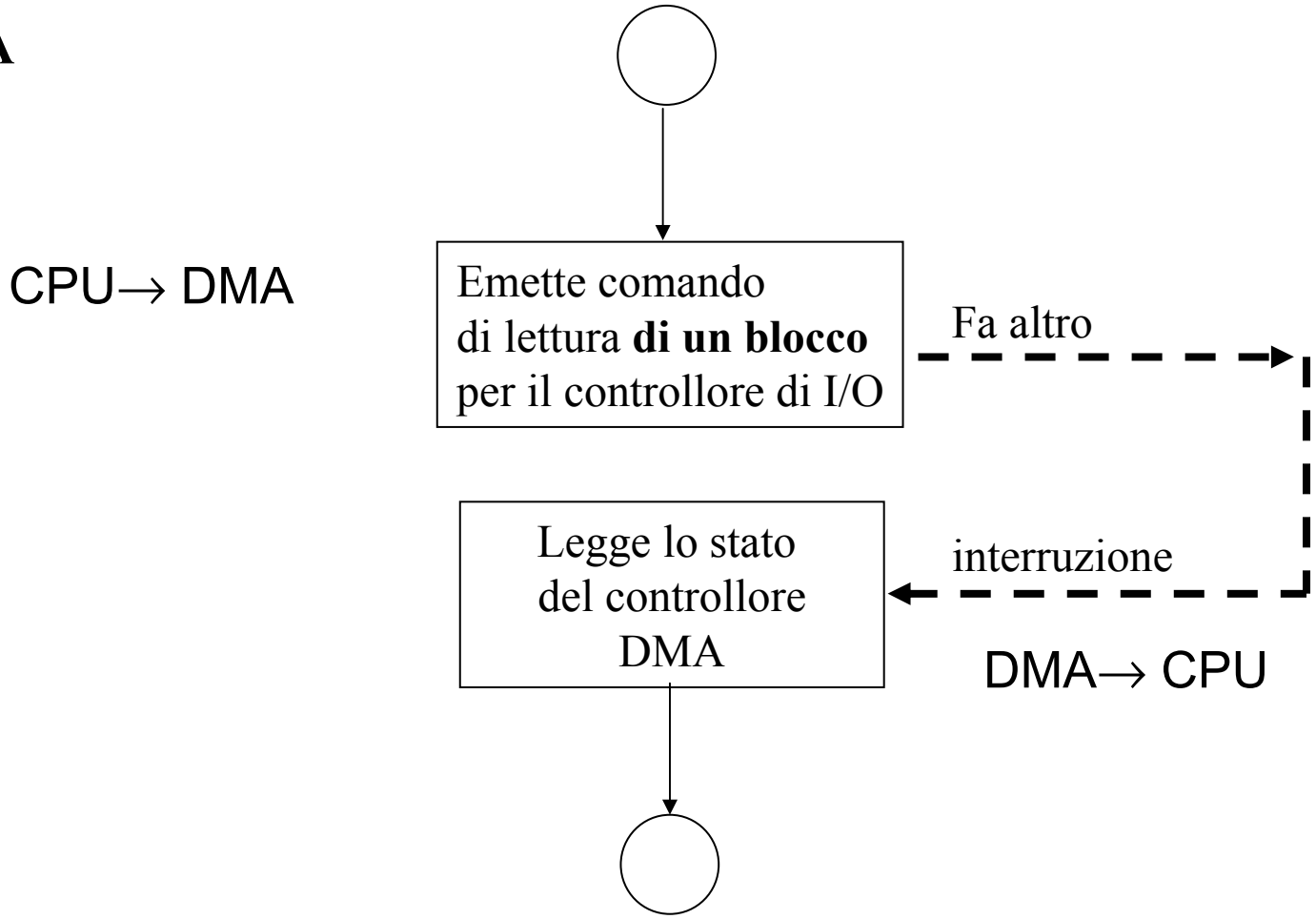



Il processore non è impegnato nell'acquisizione del singolo dato

[P.es. salvataggio in memoria, incremento indice, salvataggio e ripristino stato,…]

ma solo all'inizio e alla fine del trasferimento di un blocco!

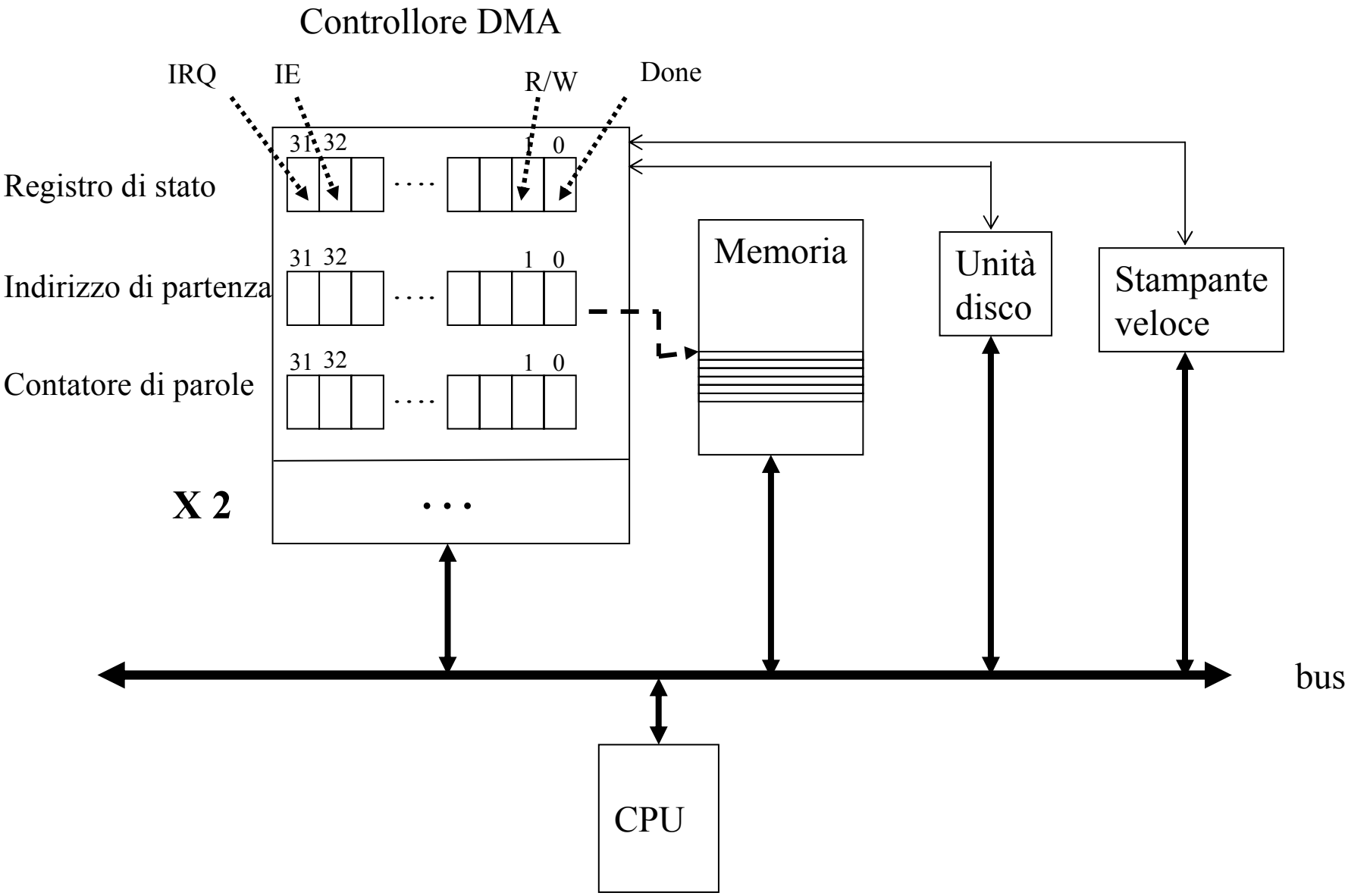
**I/O con DMA**




NB: processore dialoga con DMA come con un normale dispositivo I/O;
controllore DMA ha vari registri:

- stato e controllo [bit IRQ, IE, R/W, Done]
- indirizzo di partenza
- contatore di parole

**Un controllore DMA a due canali**: può controllare due dispositivi…




NB: si vede che può esserci conflitto sul bus [con processore o con vari DMA]

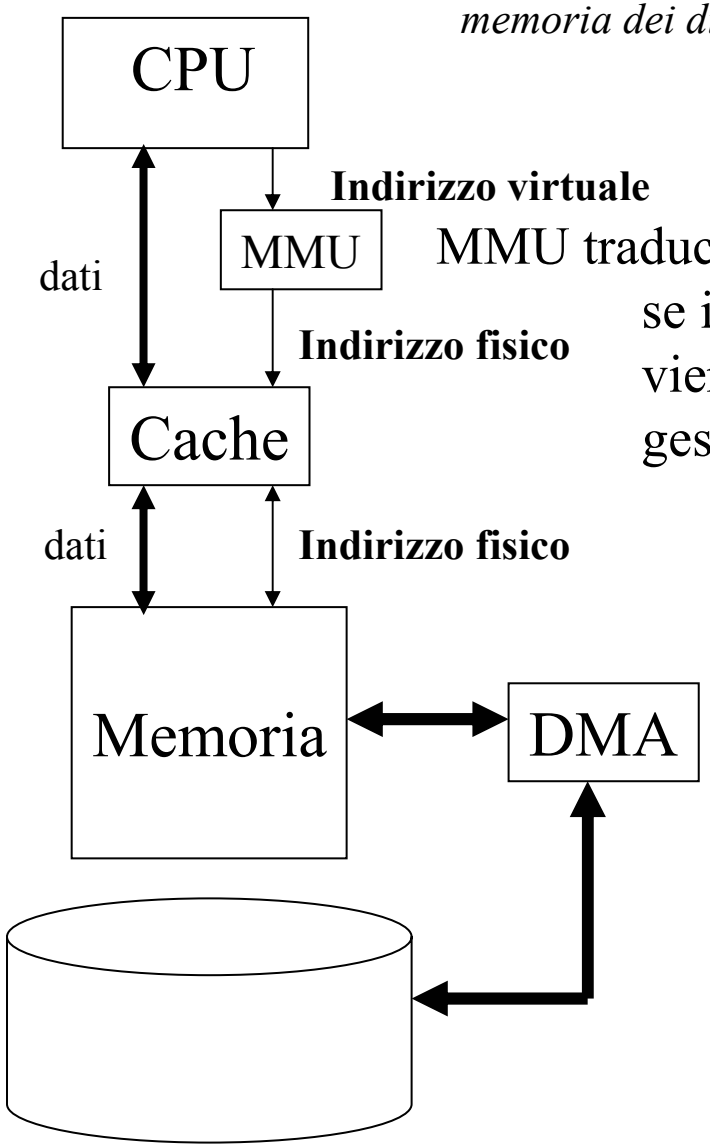
- Come vedremo, un opportuno meccanismo di arbitraggio assegna il controllo del bus ad uno dei dispositivi richiedenti.
- Tipicamente, DMA ha la priorità su processore

  [gestisce trasferimenti periferiche ad alta velocità]
- Due modalità:
  - cycle stealing: furto di cicli

    Accesso alternato DMA-Processore [che genera la maggior quantità di accessi in memoria]
  - blocco [DMA trasferisce un intero blocco dati senza interruzione]

# Richiamo: Uso in Sistemi a Memoria Virtuale

(*hardware e SO collaborano per gestire uno spazio degli indirizzi più grande di quello di memoria una velocità di trasferimento vicina a quella della cache e per garantire la non invasione di memoria dei differenti processi)*




MMU traduce indirizzi virtuali in reali:
se i dati sono in memoria, ok; se sono su disco, viene invocato il Sistema Operativo per gestire il trasferimento dal disco in Memoria

DMA gestisce il trasferimento di pagine: riceve comando e segnala fine transazione.

## ALCUNE NOTE SU I/O, MEMORIA VIRTUALE E SISTEMI OPERATIVI

- Come già visto, un processo che richieda un servizio di I/O
  [via polling, interrupt o DMA] lo fa attraverso il sistema operativo:
  richiesta $\Rightarrow$ viene posto in stato di attesa [controllo ad altro processo]
- In generale, Sistema Operativo "virtualizza" i dispositivi di I/O, gestendo tutti i dettagli di basso livello. Questo implica anche che i processi utente non hanno mai il controllo "diretto" delle periferiche:
  - con "memory-mapped I/O", il sistema operativo fa in modo che lo spazio di indirizzamento virtuale non sia mappato in indirizzi I/O
  - altrimenti, istruzioni I/O illegali in modo utente
- DMA trasferisce blocchi, ma le pagine fisiche non sono contigue!
  - DMA lavora con indirizzi fisici: trasferimento limitato ad una pagina fisica, dopodichè interrupt e controllo al S.O.
  - DMA lavora con indirizzi virtuali; il S.O. fornisce a DMA la parte della tabella delle pagine che serve per tradurre i suoi indirizzi in fisici
  - DMA "intelligente" che accetta trasferimenti a blocchi