Progettazione di Periferiche e Programmazione di Driver

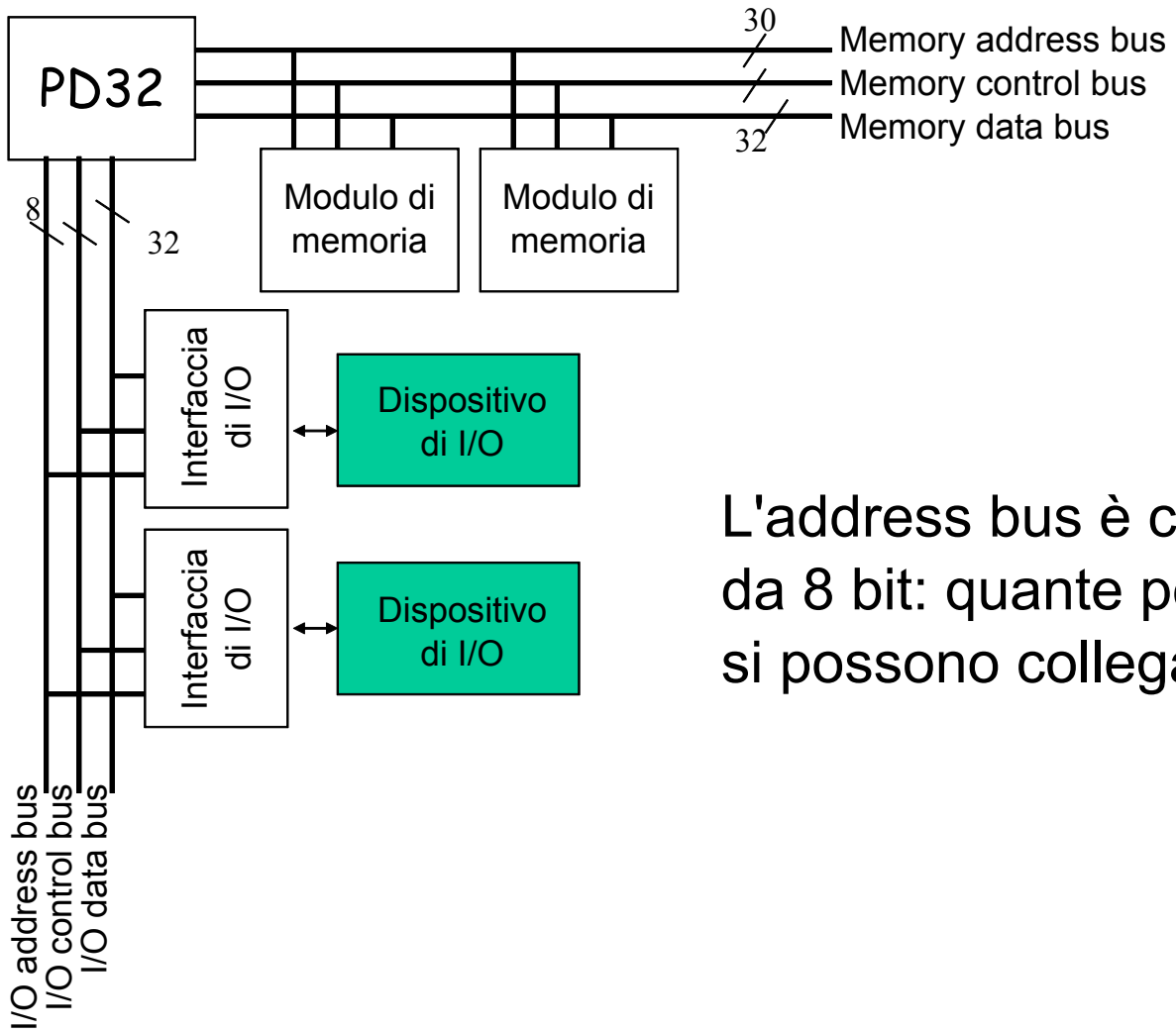
SAPIENZA
UNIVERSITÀ DI ROMA

Alessandro Pellegrini

Calcolatori Elettronici
Sapienza, Università di Roma

A.A. 2012/2013

Bus I/O del PD32



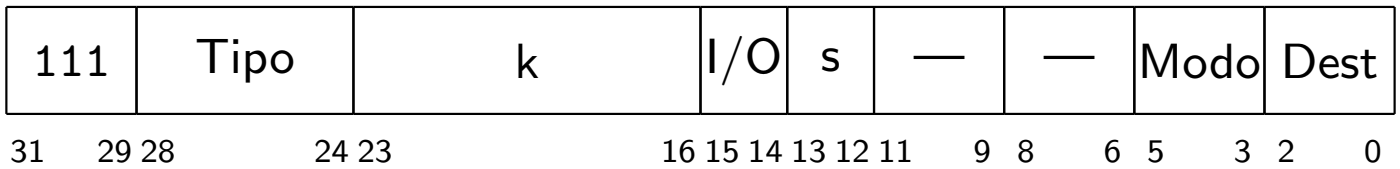
L'address bus è composto da 8 bit: quante periferiche si possono collegare al più?

Classe 7: Istruzioni di I/O

Tipo	Codice	Operandi	C N Z V P I	Commento
0	INs	dev, R	- - - - - -	Copia il dato dal buffer del device dev in R
1	OUTs	dev, R	- - - - - -	Copia il dato da (R) nel buffer del device dev
2	START	dev	- - - - - -	Azzera il flip-flop READY del dispositivo e avvia l'operazione di I/O
4	JR	dev, R	- - - - - -	Se READY=1, va all'indir. R
5	JNR	dev, R	- - - - - -	Se READY=0, va all'indir. R

Formato Istruzioni I/O

- Per l'operando dev sono ammessi solo due modi di indirizzamento: diretto con registro ed assoluto. Per la codifica di questo campo sono usati i campi I/O e k.
- Il campo I/O può assumere solo due valori:
  - `01`: indica che il contenuto di k è l'indirizzo del device
  - `10`: indica che l'indirizzo del device è contenuto nel registro generale specificato dai primi 3 bit del campo k
- Poiché i campi modo sorgente e sorgente sono inutilizzati, sia la sorgente (in caso di `OUT`) che la destinazione (in caso di `IN`) viene specificata nei campi modo destinazione e destinazione.

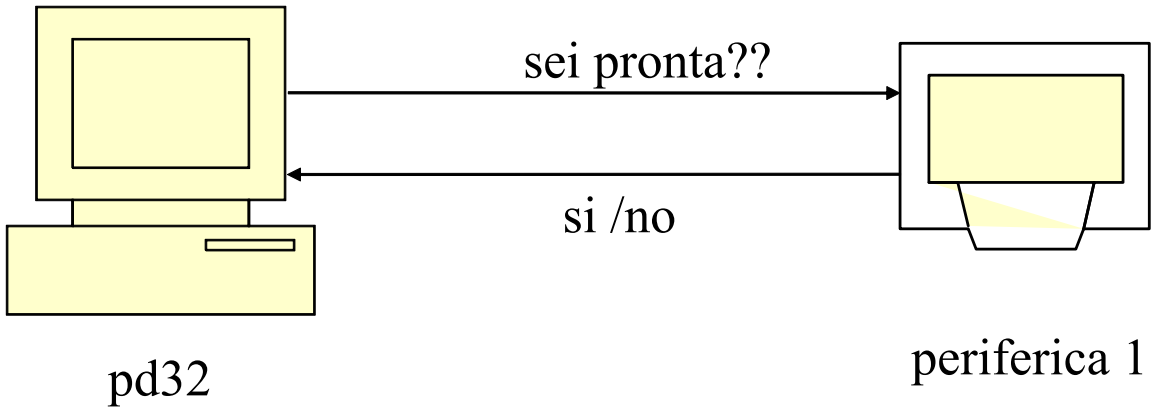


Interazione con le periferiche

- Si può interagire con le periferiche in due modi
  - *Modo sincrono*: il codice del programma principale si mette in attesa della periferica
  - *Modo asincrono*: la periferica informa il sistema che una qualche operazione è stata completata

Interazione Sincrona

- Nell'interazione sincrona, il software si preoccupa di testare direttamente lo stato di una periferica.
- L'architettura sottostante deve dare la possibilità al processore di conoscere ad ogni istante lo stato della periferica
- Le tecniche più utilizzate per interagire in modo sincrono con le periferiche sono:
  - *Busy Waiting*
  - *Polling*



Busy Waiting

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

Busy Waiting

```
Loop:  salta a Loop se la periferica non è pronta
```

- Il processore continua ad eseguire questo controllo restando in attesa attiva
- Il consumo di CPU resta al 100% fintanto che la periferica non diventa pronta

Busy Waiting

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

Busy Waiting

```
Loop:  salta a Loop se la periferica non è pronta
```

- Il processore continua ad eseguire questo controllo restando in attesa attiva
- Il consumo di CPU resta al 100% fintanto che la periferica non diventa pronta
- Nel PD32 si utilizza l'istruzione `JNR` (Jump Not Ready) per conoscere lo stato della periferica:

```
Loop:  JNR Device, Loop
```

Polling

- Il Polling è un'operazione simile al Busy Waiting, che coinvolge però più di una periferica connessa all'elaboratore
- La verifica viene svolta in maniera circolare su tutte le periferiche interessate

Polling

- Il Polling è un'operazione simile al Busy Waiting, che coinvolge però più di una periferica connessa all'elaboratore
- La verifica viene svolta in maniera circolare su tutte le periferiche interessate

```
1 poll:
2     jr D1, Op_Dev_1
3     jr D2, Op_Dev_2
4     jr D3, Op_Dev_3
5     jmp poll
```

- Il software interroga ciclicamente le periferiche per sapere se qualcuna è pronta ad interagire

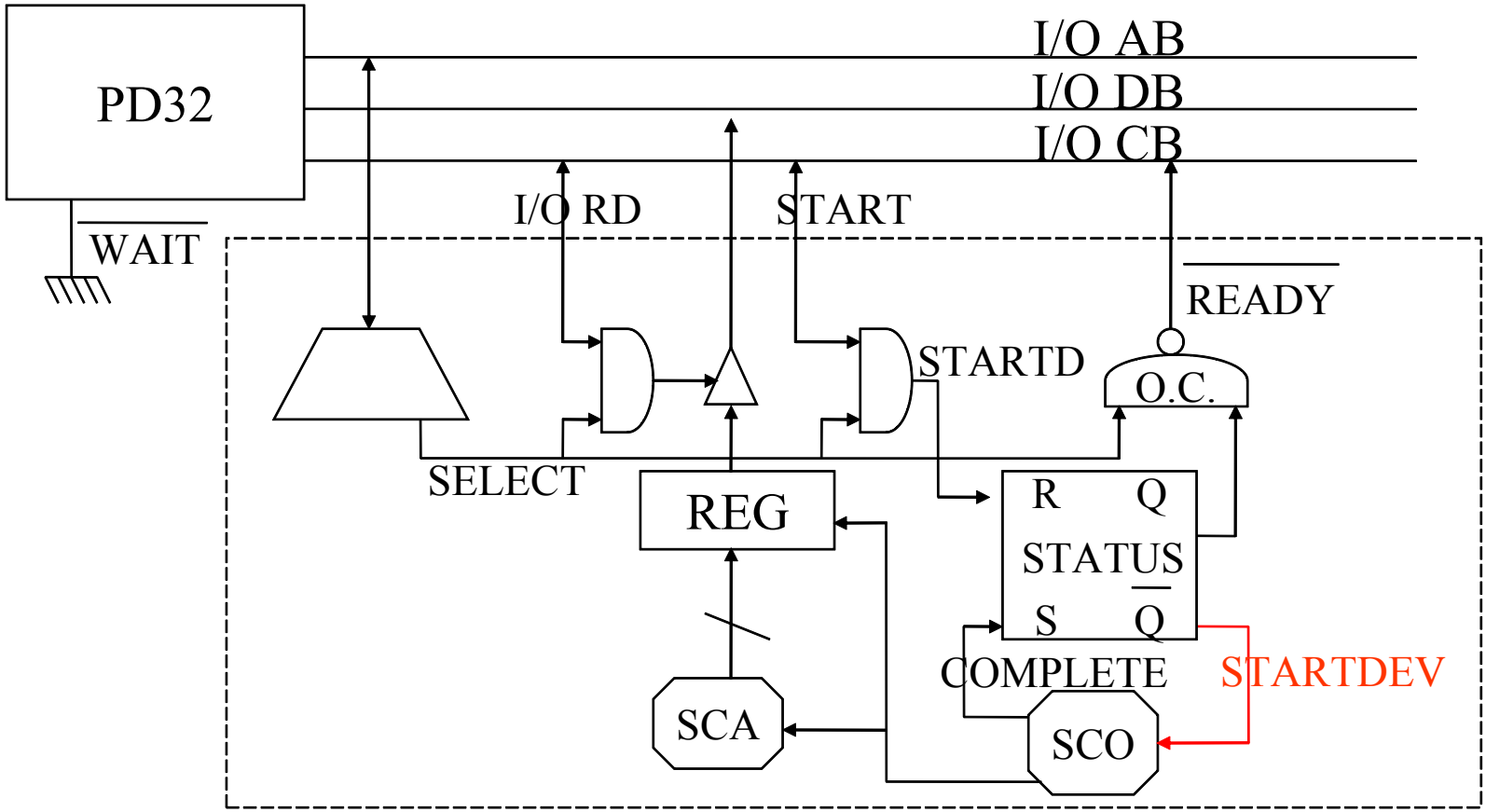
Progettazione dell'interfacciamento con le periferiche

- L'*interfaccia hardware* di una periferica consente di connettere ad una determinata architettura periferiche anche estremamente differenti tra loro
- Le interconnessioni ed i componenti dell'interfaccia hardware devono supportare la *semantica* della periferica

Progettazione dell'interfacciamento con le periferiche

- L'*interfaccia hardware* di una periferica consente di connettere ad una determinata architettura periferiche anche estremamente differenti tra loro
- Le interconnessioni ed i componenti dell'interfaccia hardware devono supportare la *semantica* della periferica
- Nel PD32, in generale, vorremo:
  - leggere dalla periferica
  - scrivere sulla periferica
  - selezionare una particolare periferica tra quelle connesse al bus
  - interrogare la periferica per sapere se ha completato la sua unità di lavoro
  - avviare la periferica

Interfaccia di Input



Interfaccia di Input: Software

I/O programmato: l'handshaking tra la periferica ed il processore deve essere implementato a mano.

```
1 (Loop1: jnr DeviceIN, Loop1)
2         start DeviceIN
3 Loop2: jnr DeviceIN, Loop2
4         inb DeviceIN, R0
```

1. Aspetto finché la periferica DeviceIN non è disponibile (nel caso in cui possa interagire con altri utilizzatori)
2. Avvio la periferica così che possa produrre informazioni
3. Aspetto che la periferica completi la sua unità di lavoro
4. Acquisisco da periferica il risultato dell'operazione

Interfaccia di Input: Software

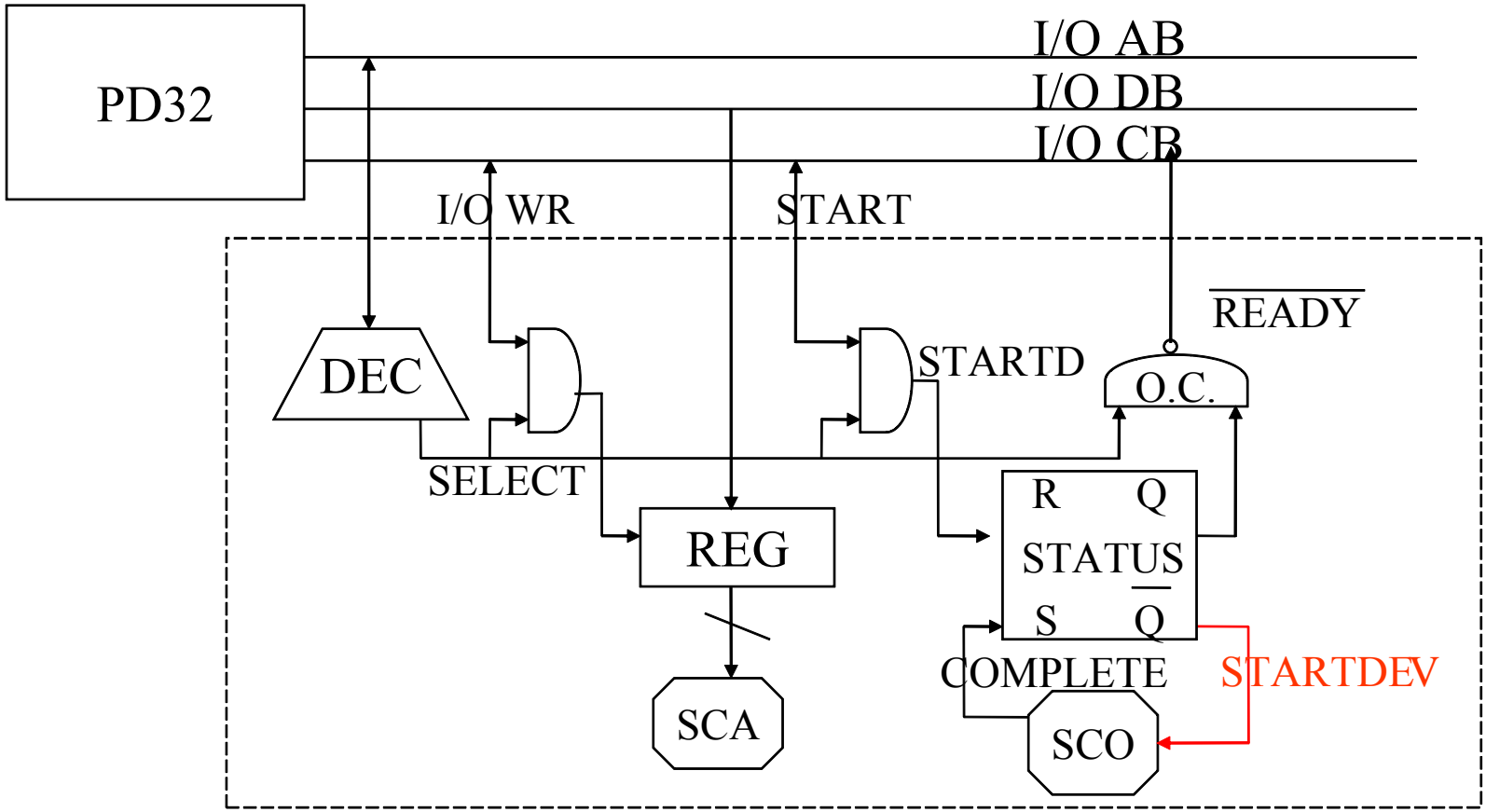
I/O programmato: l'handshaking tra la periferica ed il processore deve essere implementato a mano.

```
1 (Loop1: jnr DeviceIN, Loop1)
2         start DeviceIN
3 Loop2: jnr DeviceIN, Loop2
4         inb DeviceIN, R0
```

1. Aspetto finché la periferica DeviceIN non è disponibile (nel caso in cui possa interagire con altri utilizzatori)
2. Avvio la periferica così che possa produrre informazioni
3. Aspetto che la periferica completi la sua unità di lavoro
4. Acquisisco da periferica il risultato dell'operazione

Implementiamo busy waiting o polling?

Interfaccia di Output



Interfaccia di Output: Software

I/O programmato: l'handshaking tra la periferica ed il processore deve essere implementato a mano.

```
1 (Loop1: jnr DeviceOUT, Loop1)
2         outb R0, DeviceOUT
3         start DeviceOUT
4 Loop2: jnr DeviceOUT, Loop2
```

1. Aspetto finché la periferica DeviceOUT non è disponibile (nel caso in cui possa interagire con altri utilizzatori)
2. Scrivo nel registro di interfaccia con la periferica il dato che voglio produrre in output
3. Avvio la periferica, per avvertirla che ha un nuovo dato da processare
4. Attendo che la periferica finisca di produrre in output il dato

Esercizio Busy Waiting

Una periferica `AD1` produce dati di dimensione word come input per il processore PD32. Scrivere il codice di una subroutine `IN_AD1` che accetti come parametri nel registro `R1` il numero di dati (word) da leggere dalla periferica `AD1`, ed in `R2` l'indirizzo di memoria da cui il processore PD32 dovrà incominciare a scrivere i dati così acquisiti da periferica. Scrivere inoltre il programma che invochi la funzione `IN_AD1` chiedendo di acquisire 100 word dalla periferica `AD1` e di memorizzarli in un vettore posto a partire dall'indirizzo 1200h.

Esercizio Busy Waiting: Soluzione

```
1  ORG 400h
2  CODE
3      movl #100, R1 ; Numero di dati da acquisire
4      movl #1200h, R2 ; Indirizzo iniziale del vettore
5      jsr IN_AD1
6      halt
7
8  IN_AD1:
9      push R0 ; Salvo i registri che utilizzero' nello stack
10     push R1
11     push R2
12   in1: jnr AD1, in1 ; Non so se AD1 e' gia' al lavoro
13   in2: start AD1 ; Chiedo di produrre un dato da acquisire...
14   in3: jnr AD1, in3 ; ...e aspetto che la periferica finisca
15
```

Esercizio Busy Waiting: Soluzione (2)

```
16      inw AD1, R0 ; Muovo il dato dalla periferica alla CPU...
17      movw R0, (R2)+ ; ...e lo salvo nell'array
18      subl #1, R1
19      jnz in2 ; Se devo acquisire altri dati, procedo
20      pop R2 ; Distruggo la mia finestra di stack
21      pop R1
22      pop R0
23      ret ; torno alla funzione chiamante
24  END
```

Esercizio Polling

Quattro periferihe `AD1`, `AD2`, `AD3`, `AD4` producono dati di dimensione word come input per il processore PD32. Scrivere il codice di una subroutine `IN_AD` che accetti come parametri nel registro `R1` il numero di dati (word) da leggere dalle periferiche, ed in `R2` l'indirizzo di memoria da cui il processore PD32 dovrà incominciare a scrivere i dati così acquisiti. Scrivere inoltre il programma che invochi la funzione `IN_AD` chiedendo di acquisire 100 word dalle periferiche e di memorizzarli in un vettore posto a partire dall'indirizzo 1200h. ATTENZIONE: i 100 dati possono essere letti non necessariamente rispettando l'ordine delle periferiche (ad esempio 10 da `AD1`, 23 da `AD2`, ...)

Esercizio Polling: Soluzione

```
1  ORG 400h
2  CODE
3      movl #100, R1 ; Numero di dati da acquisire
4      movl #1200h, R2 ; Indirizzo iniziale del vettore
5      jsr IN_AD
6      halt
7  IN_AD:
8      push R0 ; Salvo i registri che utilizzero'
9      push R1
10     push R2
11   poll1: jr AD1, IN_1 ; attende che AD1 sia pronta
12   poll2: jr AD2, IN_2
13   poll3: jr AD3, IN_3
14   poll4: jr AD4, IN_4
15     jmp poll1
```

Esercizio Polling: Soluzione (2)

```
16   IN_i: ; Blocco di codice per la periferica i-sima
17     inw AD1, R0 ; Muovo il dato dalla periferica alla CPU...
18     movw R0, (R2)+ ; ...e lo salvo nell'array
19     subl #1, R1
20     jz finished ; Se non devo acquisire altro, termino
21     start AD1 ; Riavvio la periferica per farle produrre altri dati
22     jmp poll_i+1 ; riparto dalla periferica successiva
23     ; ...
24   finished:
25     pop R2 ; Distruggo la mia finestra di stack
26     pop R1
27     pop R0
28     ret ; torno alla funzione chiamante
29 END
```

Esecuzione Asincrona: le interruzioni

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

Le interruzioni: problematiche da affrontare

Problemi:

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

Le interruzioni: problematiche da affrontare

Problemi:

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

Soluzioni:

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

Passi per la gestione di un'interruzione

Per poter gestire correttamente un'interruzione, è *sempre* necessario seguire i seguenti passi:

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

Cambio di contesto

- Il *contesto* d'esecuzione di un processo è costituito da:
  - Registro `PC`: contiene l'indirizzo dell'istruzione dalla quale si dovrà riprendere l'esecuzione una volta terminata la gestione dell'interrupt
  - Registro `SR`: alcuni bit di condizione potrebbero non essere ancora stati controllati dal processo
  - Registri `TEMP1` e `TEMP2`, per supportare la ripresa dell'esecuzione di operazioni logico aritmetiche. Se le interruzioni vengono gestite unicamente prima della fase di fetch, non è necessario memorizzare i valori di questi registri (essi vengono infatti scritti unicamente dalle microoperazioni associate ad un'istruzione di tipo logico-aritmetico)
- Quando viene generata un'interruzione avviene una commutazione dal contesto del processo interrotto a quello del driver
- Analogamente il contesto del processo interrotto deve essere ripristinato una volta conclusa l'esecuzione del driver

jsr e registro SR

- Nel caso di un'istruzione `jsr` lo stato del processo non viene (banalmente!) memorizzato
- È compito del programmatore, dunque, memorizzare manualmente il valore del registro SR se esso deve essere utilizzato al ritorno dalla routine

sbagliato:

```
1 cmpl #0, R0
2 jsr subroutine
3 jz label
```

corretto:

```
1 cmpl #0, R0
2 pushsr
3 jsr subroutine
4 popsr
5 jz label
```

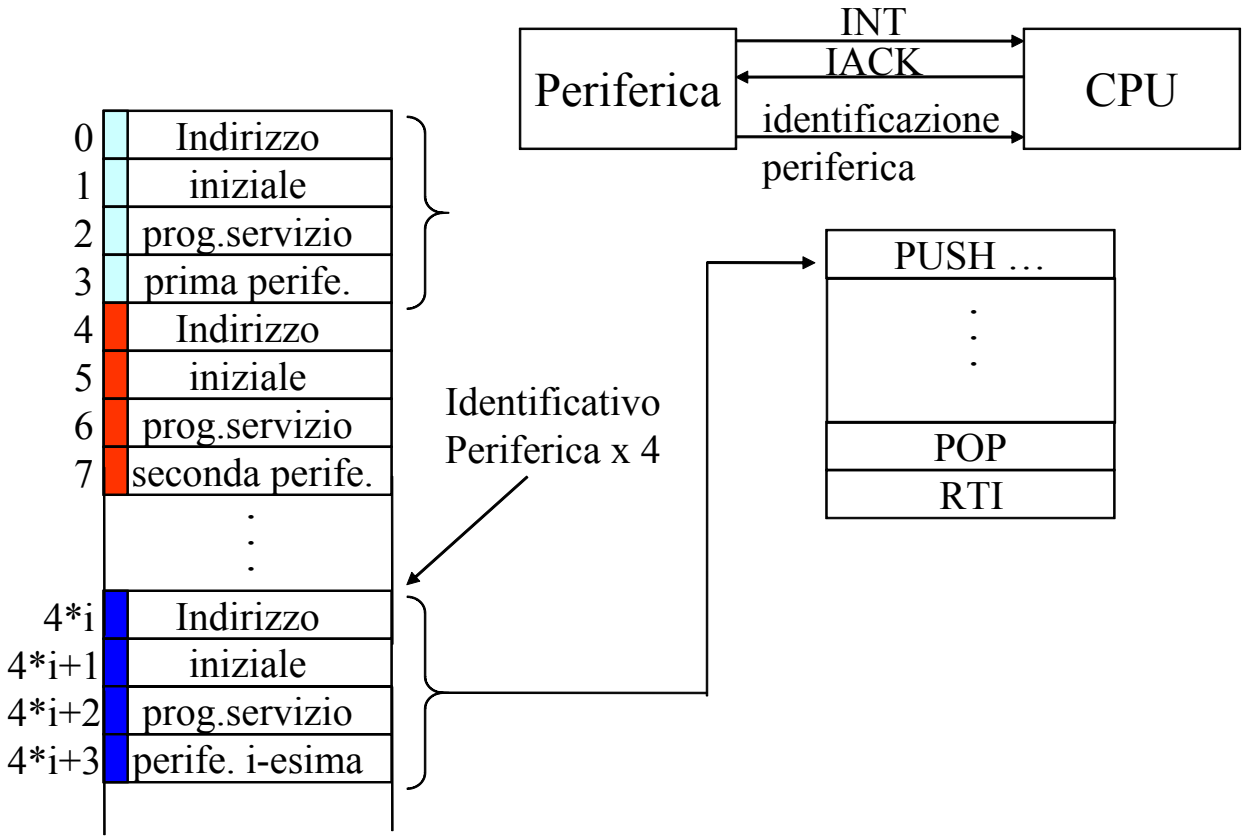
- Il programmatore può scegliere di eseguire le `pushsr`/`popsr` anche all'interno della funzione, se lo ritiene più performante

Cambio di contesto (2)

- È necessario assicurarsi che non si verifichino altre interruzioni durante le operazioni di cambio di contesto
  - Potrebbero altrimenti verificarsi delle incongruenze tra lo stato originale del processo e quello ripristinato
- Al termine dell'esecuzione di un'istruzione, il segnale `IRQ` assume il valore 1 e il flip-flop `I` viene impostato a 0 via firmware
- Inoltre il PD32 provvede a salvare nello stack i registri `SR` e `PC`
- Infine, in `PC` viene caricato l'indirizzo della routine di servizio (driver) della periferica che ha generato la richiesta di interruzione.

Identificazione del driver (IVT)

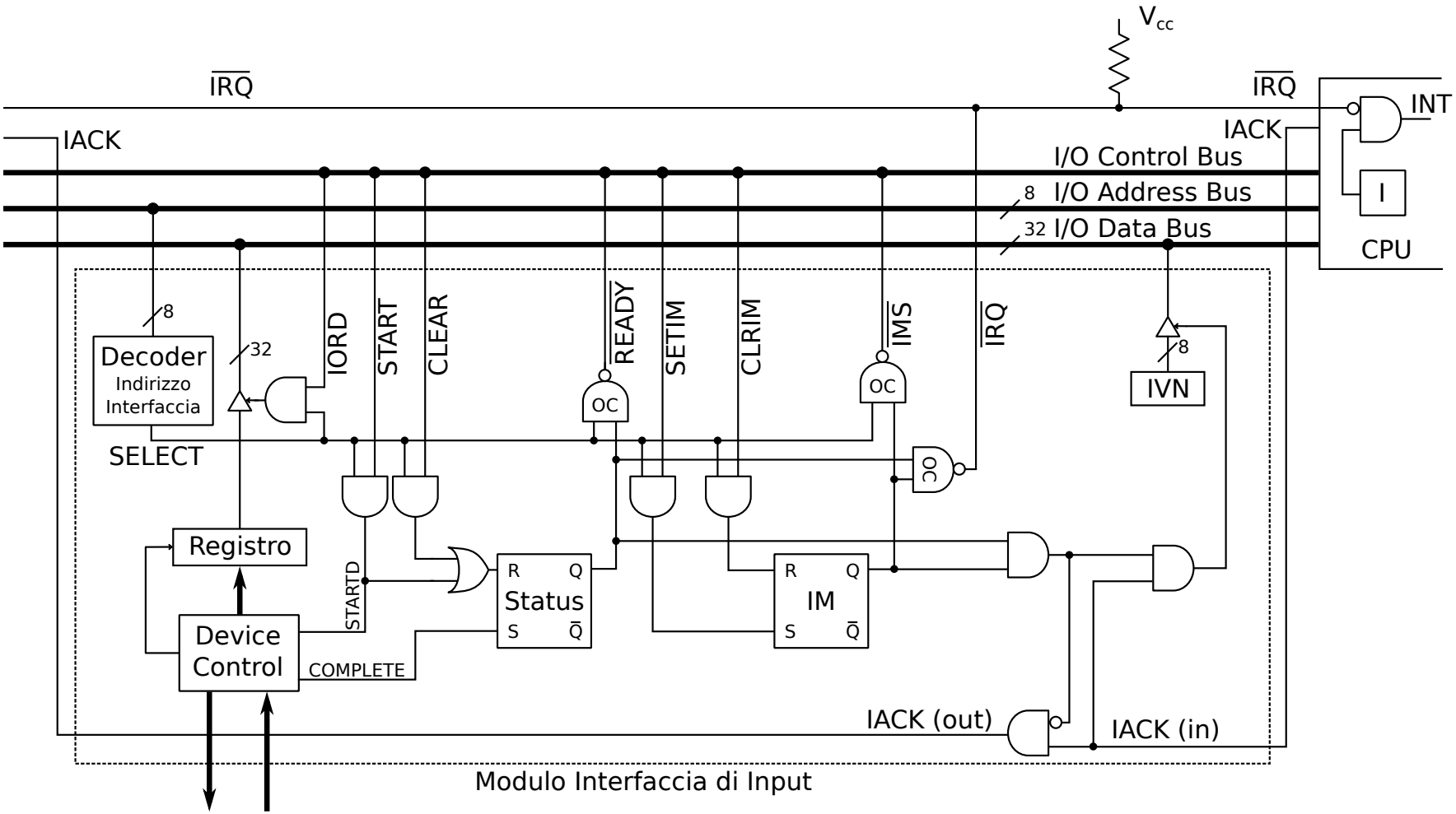
- L'identificazione del driver da attivare in risposta all'interruzione si basa su un numero univoco (IVN, Interrupt Vector Number) comunicato dalla periferica al processore, che identifica un elemento all'interno dell'Interrupt Vector Table (IVT)



Gestione di un'interruzione

```
SR[I] = 0; R7 → TEMP1
4 → ALU; ALU_OUT[SUB] → R7
R7 → MAR
PC → MDR
MDR → (MAR)
R7 → TEMP1
4 → ALU; ALU_OUT[SUB] → R7
SR → MDR
R7 → MAR
MDR → (MAR)
IACK IN
IACK IN; IVN → IO D/R
IO D/R → TEMP2
SHIFTER_OUT[SX, 2] → MAR ; soltanto 256 driver differenti!
(MAR) → MDR
MDR → PC
```

Il supporto hardware alle interruzioni con priorità



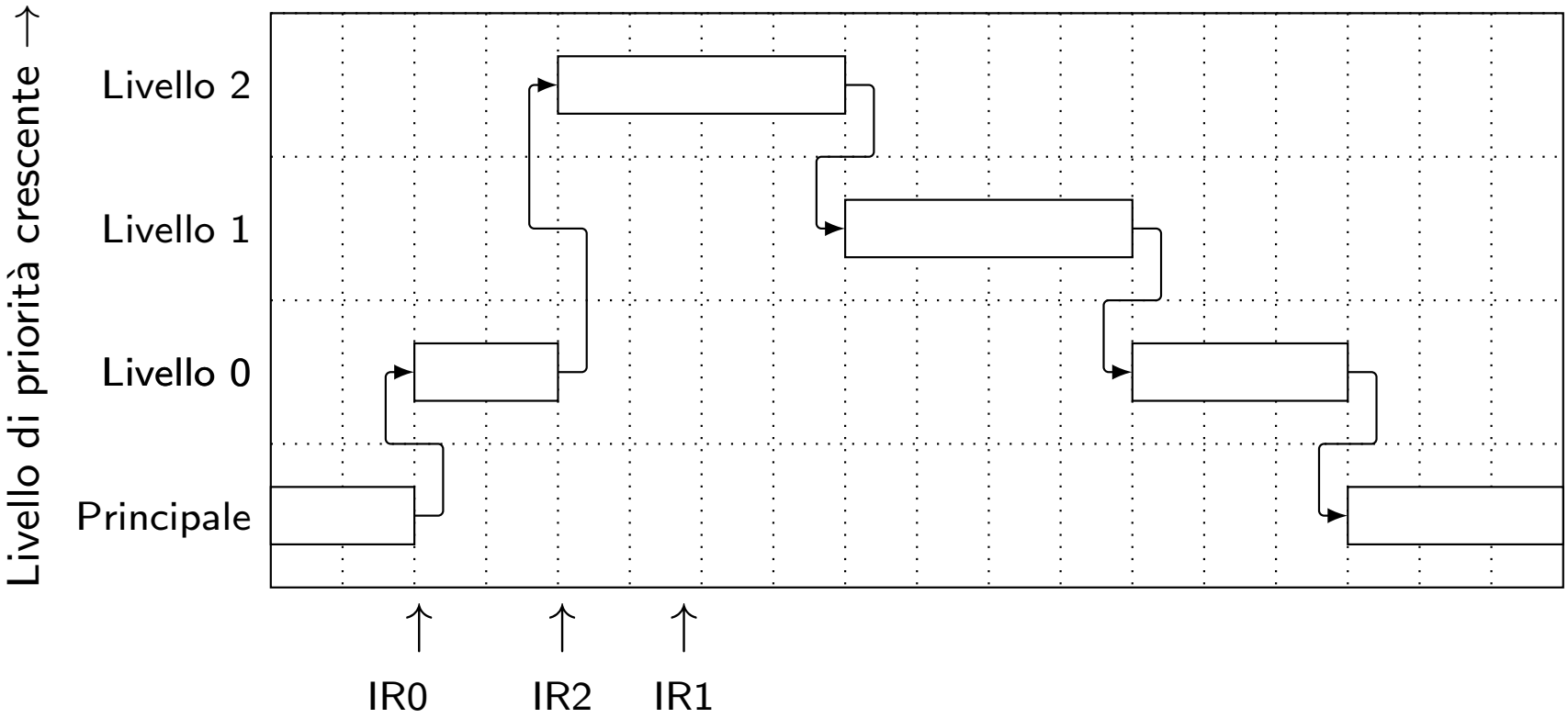
Ritorno da un'interruzione

```
R7 → MAR
(MAR) → MDR
MDR → SR ; viene ripristinato il flag I!
R7 → TEMP1
4 → ALU; ALU_OUT[ADD] → R7
R7 → MAR
(MAR) → MDR
MDR → PC
R7 → TEMP1
4 → ALU; ALU_OUT[ADD] → R7
```

Priorità nella gestione delle interruzioni

- Introdurre una gerarchia per la gestione delle interruzioni consiste essenzialmente nel definire dei meccanismi per:
  - Stabilire quale dispositivo debba essere servito per primo nel caso di richieste contemporanee
  - Consentire che il servizio di una interruzione possa essere a sua volta interrotto da dispositivi con priorità maggiore
- Tali meccanismi possono essere implementati via hardware (vedi controllore interruzione a priorità) o, nel caso in cui non via un supporto hardware dedicato, via software

Priorità nella gestione delle interruzioni (2)



Classe 7: Istruzioni di I/O

Tipo	Codice	Operandi	C N Z V P I	Commento
3	CLEAR	dev	- - - - - -	Azzera il flip-flop READY
6	SETIM	dev	- - - - - -	Abilita il dispositivo dev ad inviare interruzioni ($1 \rightarrow$ IMS)
7	CLRIM	dev	- - - - - -	Impedisce al dispositivo dev di inviare interruzioni ($0 \rightarrow$ IMS)
8	JIM	dev, R	- - - - - -	Se IMS $= 1$, va all'indir. R
9	JNIM	dev, R	- - - - - -	Se IMS $= 0$, va all'indir. R

Gestione della gerarchia delle priorità

1) Stabilire quale dispositivo debba essere servito per primo nel caso di richieste contemporanee

- Soluzione *hardware*: si utilizza il segnale di `IACK` propagato in *daisy-chain* per il riconoscimento dell'origine dell'interruzione. Così facendo si introduce una priorità che è funzione della "distanza" dal processore (la periferica più vicina ha priorità massima)

- Soluzione *software*: le periferiche vengono interrogate tramite polling. L'ordine di interrogazione definisce la priorità nella gestione delle interruzioni

Gestione della gerarchia delle priorità

2) Consentire che la gestione di una interruzione possa essere a sua volta interrotto da dispositivi con priorità maggiore

- Soluzione *hardware*: è legata all'hardware che abbiamo già visto

- Soluzione *software*:
  1. Ogni routine di servizio che prevede di essere interrotta (di priorità non massima) deve rendere il processore nuovamente interrompibile (`SETI`)
  2. Prima della `SETI` si devono mascherare i flip-flop `IM` delle periferiche a priorità minore
  3. Lo stato di interrompibilità, (valori dei registri `IM`) fa parte del contesto del programma e va ripristinato prima della `RTI`
  4. Prima della `SETI` si deve rimuovere la causa dell'interruzione, per evitare lo stack overflow (utilizzando `CLRIM`, oppure resettando il flip flop di status con `START` o `CLEAR`). Tuttavia se la stessa periferica genera un altro interrupt, si possono generare conflitti sui dati e sul codice!

Gestione della gerarchia delle priorità (2)

5. Il ripristino dello stato del processo deve avvenire con il processore non interrompibile, per evitare le incongruenze che potrebbero sorgere a causa di una commutazione incompleta.

Ad esempio, si consideri il driver periferica di priorità "media" (interrompibile solo da device con priorità "alta" e non da device con priorità "bassa"):

```
1 ...; codice del driver per device
       con priorita' media
2 seti; il processore e'
      interrompibile
3 ...; fine codice, inizio
      ripristino contesto
4 setim dev_low_priority;
5 pop ...; altre op. ripristino
      contesto
6 pop ...;
7 rti
```

Se arriva un'interruzione da parte del device a bassa priorità, questa viene subito servita ed è violata la gerarchia di priorità!

Gestione della gerarchia delle priorità: esempio

Tre rilevatori di movimento sono connessi al PD32. Quando viene rilevato un movimento, i sensori generano una richiesta di interruzione. Si vuole servire le interruzioni provenienti dai sensori con la seguente priorità:

- SENSORE0 ha priorità minore di SENSORE1
- SENSORE1 ha priorità minore di SENSORE2
- SENSORE2 ha la priorità massima nel sistema

Gestione della gerarchia delle priorità: esempio (2)

Driver SENSORE0:

```
1 setim SENSORE2
2 setim SENSORE1
3 clrim SENSORE0
4 seti
5 ...
6 clri
7 setim SENSORE0
8 rti
```

Driver SENSORE1:

```
1 setim SENSORE2
2 clrim SENSORE1
3 clrim SENSORE0
4 seti
5 ...
6 clri
7 setim SENSORE0
8 setim SENSORE1
9 rti
```

Driver SENSORE2:

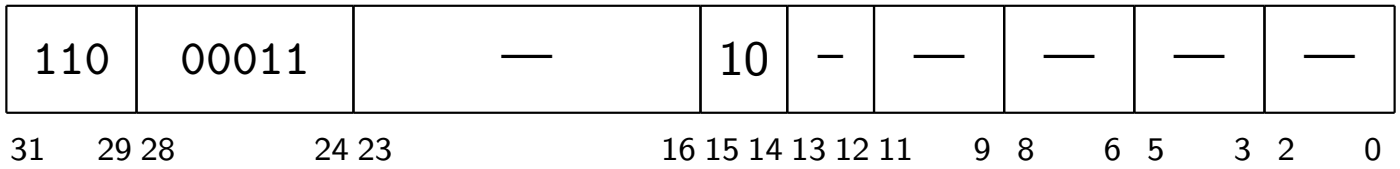
```
1 ...
2 rti
```

Interruzioni: riassunto

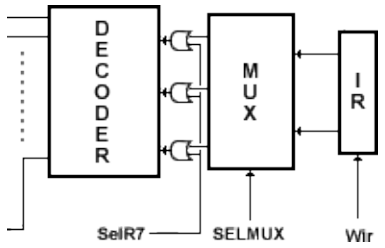
- La periferica attiva il segnale di interruzione sul Control Bus di I/O
- Prima della fase di fetch il PD32 controlla se ci sono richieste di interruzione in attesa
- L'IVN della periferica che ha generato l'interruzione viene letto e moltiplicato per 4, per leggere all'interno dell'IVT l'indirizzo in cui è memorizzata la routine di servizio (driver)
- All'attivazione del driver viene fatto un cambio di contesto, salvando `SR` e `PC`. Inoltre il microcodice imposta a 0 il flip flop `I` (equivalente all'esecuzione dell'istruzione `CLRI`) inibendo la verifica della presenza di eventuali nuove interruzioni
- Al momento dell'esecuzione di `RTI` al termine della routine vengono automaticamente ripristinati `PC` e `SR`. Inoltre il microcodice imposta a 1 il flip flop `I` (equivalente all'esecuzione dell'istruzione `SETI`) permettendo la verifica di presenza di nuove interruzioni

Formato istruzione RTI

L'istruzione RTI ha il seguente formato:



Nell'IR non è presente pertanto il codice identificativo del registro R7 necessario per il ripristino dello stato. Per questo motivo, l'hardware associato al selettore di registro presenta il segnale SelR7:



Interrupt nel mondo reale

- La gestione degli interrupt è un punto critico per le architetture, ed è uno dei punti critici di interconnessione tra hardware e software
  - La IVT, nei moderni sistemi, viene popolata dal Sistema Operativo in funzione dell'architettura sottostante
  - Le informazioni scritte dal sistema operativo devono essere coerenti con il formato interpretabile dal microcodice del processore!
- Praticamente tutti i sistemi operativi (Unix, Mac OS X, Microsoft Windows) dividono la gestione degli interrupt in due parti:
  - *First-Level Interrupt Handler*, o *top half*
  - *Second-Level Interrupt Handler*, o *bottom half*

Interrupt nel mondo reale (2)

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

- Una *bottom half* è molto più simile ad un normale processo
  - Viene mandata in esecuzione (dal Sistema Operativo) non appena c'è disponibilità di tempo di CPU
  - L'esecuzione dei compiti assegnati alla bottom half può avere una durata non minimale

Esercizio sulle Interruzioni: Produttore/Consumatore

Due periferiche, una di input ed una di output, vogliono scambiare dati (in formato byte) tra di esse. Per supportare lo scambio, viene utilizzato un buffer (di dimensione 1 byte) in memoria di lavoro RAM. La periferica PRODUTTORE (periferica di input) genera un dato che deve essere scritto all'interno del buffer tampone. La periferica CONSUMATORE (di output) leggerà dal buffer il dato prodotto e lo processerà. È necessario impedire alla periferica PRODUTTORE di generare un nuovo dato fintanto che quello contenuto all'interno del buffer tampone non sia stato correttamente processato dalla periferica CONSUMATORE. Analogamente, la periferica CONSUMATORE non può processare il dato contenuto all'interno del buffer tampone prima che un nuovo dato sia stato generato dalla periferica PRODUTTORE.

Produttore/Consumatore

```
1  ; Nel simulatore e' necessario installare due periferiche:
2  ; INPUT: I/O=I, ind=30, IVN=2
3  ; OUTPUT:I/O=O, ind=56, IVN=7
4  org 400h
5      input equ 30 ;indirizzo periferica input
6      output equ 56 ;indiriirizzo periferica output
7      flag db 0 ; flag=0 buffer vuoto, flag=1 buffer pieno
8      buffer equ 500h ;indirizzo buffer di scambio
9
10 code
11     seti
12     setim input
13     setim output
14     start input
15     start output
16     halt
17
18
```

Produttore/Consumatore (2)

```
19 driver 2, 600h ;Il driver della periferica con IVN=2 (input)
20   pinput: push R0 ;salva contenuto di R0
21     movb flag, R0 ;carica flag in R0
22     cmpb #1, R0 ;controlla se buffer pieno
23     jz inibisci ;se pieno pongo in attesa la periferica di input
24   accetta: ;altrimenti invia dato
25     movb #1, flag ;pongo il flag=1 (buffer pieno)
26     inb input, buffer ;carico dato nel buffer
27     start input ;abilito device a generare un nuovo dato
28     jnim output,sbloccato ;se periferica di output e' in attesa la sblocco
29     jmp fineinp ;termina
30   inibisci: clrim input ;pone perif. input in attesa (buffer pieno)
31   fineinp: pop R0 ;fine driver
32     rti
33   sbloccato:setim output ;sblocco periferica output
34     jmp fineinp
35
36
```

Produttore/Consumatore (3)

```
37 driver 7,700h ;Il driver della periferica di IVN=7 (output)
38   poutput:push R0 ;salva contenuto di R0
39     movb flag, R0 ;carica flag in R0
40     cmpb #0, R0 ;il buffer e' vuoto?
41     jz blocca ;se vuoto pongo in attesa la periferica di output
42   consuma: outb buffer, output ;altrimenti invio dato
43     movb #0, flag ;pongo flag=0 (buffer vuoto)
44     start output ;abilito perif. output a consumare il dato
45     jnim input, sblocca ;se perif. di input 'in attesa' la sblocco
46     jmp fineoutp ;termina
47   blocca: clrim output ;blocco perif. output (buffer vuoto)
48   fineoutp: pop R0 ;termina driver
49     rti
50   sblocca:setim input ;sblocco input
51     jmp fineoutp
52 end
```

Costo aggiuntivo di un'operazione di I/O gestita tramite interruzioni

- Supponiamo di avere un processore a 2GHz e un hard disk che trasferisce dati in blocchi da 4 longword con un throughput massimo di 16 MiB/s.
- Si ipotizzi che il costo aggiuntivo per ogni trasferimento, tenendo conto delle interruzioni, è pari a 500 cicli di clock.
- Si trovi la frazione di tempo di processore utilizzato nel caso in cui l'hard disk stia trasferendo dati per il 5% del suo tempo

Costo aggiuntivo di un'operazione di I/O gestita tramite interruzioni

- Supponiamo di avere un processore a 2GHz e un hard disk che trasferisce dati in blocchi da 4 longword con un throughput massimo di 16 MiB/s.
- Si ipotizzi che il costo aggiuntivo per ogni trasferimento, tenendo conto delle interruzioni, è pari a 500 cicli di clock.
- Si trovi la frazione di tempo di processore utilizzato nel caso in cui l'hard disk stia trasferendo dati per il 5% del suo tempo

$$\text{Frequenza di interrogazione: } \frac{16 MiB/s}{16 byte/accesso} = 1M = 10^6 accessi/sec$$

$$\text{Cicli per secondo spesi: } 500 \cdot 10^6$$

$$\text{\% processore utilizzato: } \frac{500 \cdot 10^6}{2000 \cdot 10^6} = 25\%$$

$$\text{Frazione di tempo di CPU: } 25\% \cdot 5\% = 1,25\%$$

Costo aggiuntivo di un'operazione di I/O gestita tramite interruzioni

- La gestione dell'I/O tramite interrupt solleva il processore dal peso di attendere il verificarsi degli eventi di I/O
- L'utilizzo degli interrupt aggiunge comunque un certo costo aggiuntivo (overhead) all'esecuzione
- Questo costo aggiuntivo può diventare intollerabile se i dispositivi con cui si interagisce hanno a disposizione una elevata larghezza di banda

Esercizio sulle interruzioni: Monitoraggio Stanza

Una stanza è monitorata da quattro sensori di temperatura, che sono pilotati da un processore PD32. Quest'ultimo controlla costantemente che il valor medio della temperatura rilevata nella stanza sia compreso nell'intervallo $[tMin, tMax]$. Nel caso in cui la temperatura non cada all'interno di questo intervallo, il microprocessore invia un segnale di allarme ad un'apposita periferica (ALLARME). Il segnale d'allarme utilizzato è il valore 1 codificato con 8 bit. Se la temperatura ritorna all'interno dell'intervallo $[tMin, tMax]$, la CPU trasmette alla periferica il valore 0.
I sensori restituiscono la temperatura misurata come un intero a 16 bit, utilizzando i decimi di grado Celsius come unità di misura.
Scrivere il codice assembly per il controllo dei sensori di temperatura e della periferica ALLARME, utilizzando il meccanismo delle interruzioni vettorizzate.

Monitoraggio Stanza: scelte di progetto

- Le misure di temperatura dei quattro sensori vengono memorizzate all'interno di un vettore di quattro elementi
- All'avvio del sistema, le quattro misure vengono forzate al centro dell'intervallo $[tMin, tMax]$
- Il sensore è una periferica di input che fornisce un solo registro di sola lettura che contiene il valore misurato
- Se la temperatura è negativa, il sensore restituisce comunque il valore 0
- ALLARME è una periferica di output, che attiva/disattiva una sirena lampeggiante. Un Flip/Flop collegato al primo bit del bus dati accende/spegne l'allarme quando viene settato/resettato.
- ALLARME è una periferica *passiva*: non ha Flip/Flop di status, né di IM. L'istruzione `JR`, pertanto, fallisce sempre

Monitoraggio Stanza

```
1  org 400h
2      sensore1 equ 0h ; indirizzo sensore1 (IVN = 1)
3      sensore2 equ 1h ; indirizzo sensore2 (IVN = 2)
4      sensore3 equ 2h ; indirizzo sensore3 (IVN = 3)
5      sensore4 equ 3h ; indirizzo sensore4 (IVN = 4)
6      allarme equ 4h ; indiririzzo periferica allarme (IVN = 5)
7      tMin equ 200 ; tMin espresso in decimi di gradi Celsius
8      tMax equ 300 ; tMax espresso in decimi di gradi Celsius
9      temperature equ 2000h ; vettore contenente le (4) temperature misurate
10 code
11   main:
12     xorl R0, R0 ; nuova media
13     xorl R1, R1 ; vecchia media
14     jsr INIT
15     setim sensore1 ; abilita i sensori a inviare interruzioni
16     setim sensore2
17     setim sensore3
18     setim sensore4
```

Monitoraggio Stanza (2)

```
19    seti ; abilita il processore a ricevere interruzioni
20    start sensore1 ; avvia l'acquisizione dei dati dai sensori di temp.
21    start sensore2
22    start sensore3
23    start sensore4
24  loop:
25    movw R0, R1 ; vecchia media = nuova media
26    jsr MEDIA ; calcola la media delle misure correnti
27    cmpw R0, R1
28    jz loop ; se la media non e' cambiata, non faccio nulla
29    movb #1, R1 ; R1 = 1 --> allarme acceso
30    cmpw #tMax, R0
31    jnc set ; tMax <= R0
32    cmpw #tMin, R0;
33    jc set; tMin > R0
34    xorb R1, R1; R1 = 0 --> allarme spento
35  set: outb R1, allarme; accende o spegne l'allarme
36    jmp loop
```

Monitoraggio Stanza (3)

```
37 ; La funzione INIT inizializza le strutture dati utilizzate dal programma
38 INIT:
39     push R0
40     push R1
41     movw #tMin, R0 ; calcola il centro dell'intervallo [tMin,tMax]
42     addw #tMax, R0
43     lsrw #1, R0
44     movw #temperature, R1
45     movw R0, (R1) ; aggiorna i buffer dei 4
46     movw R0, 2(R1) ; sensori con il valor medio
47     movw R0, 4(R1) ; dell'intervallo
48     movw R0, 6(R1)
49     pop R1
50     pop R0
51     ret
52 ; DRIVER SENSORI
53 driver 1, 1600h
54     push R1 ; Salvo il contesto del programma chiamante
```

Monitoraggio Stanza (4)

```
55      movl #sensore1, R1
56      jsr GET
57      pop R1 ; ripristino il contesto del programma chiamante
58      rti
59  driver 2, 1650h
60      push R1
61      movl #sensore2, R1
62      jsr GET
63      pop R1
64      rti
65  driver 3, 1700h
66      push R1
67      movl #sensore3, R1
68      jsr GET
69      pop R1
70      rti
71  driver 4, 1750h
72      push R1
```

Monitoraggio Stanza (5)

```
73      movl #sensore4, R1
74      jsr GET
75      pop R1
76      rti
77
78  ; La funzione GET recupera da un sensore la temperatura misurata
79  ; Accetta in R1 l'indirizzo della periferica da cui leggere la temperatua
80  ; Memorizza in temperature[sensore] la temperatura misurata
81  GET:
82      push R0 ; salvo il contesto del programma chiamante
83      push R1
84      movw #temperature, R0
85      asll #2, R1 ; R1 = device * 4
86      addw R1, R0 ; R0 = temperature + device * 4
87      asrl #2, R1 ; ripristina R1 all'address del device
88      inw R1, (R0); preleva il valore e lo mette in RAM
89      start R1 ; avvia nuova acquisizione
90      pop R1 ; ripristino il contesto del programma chiamante
```

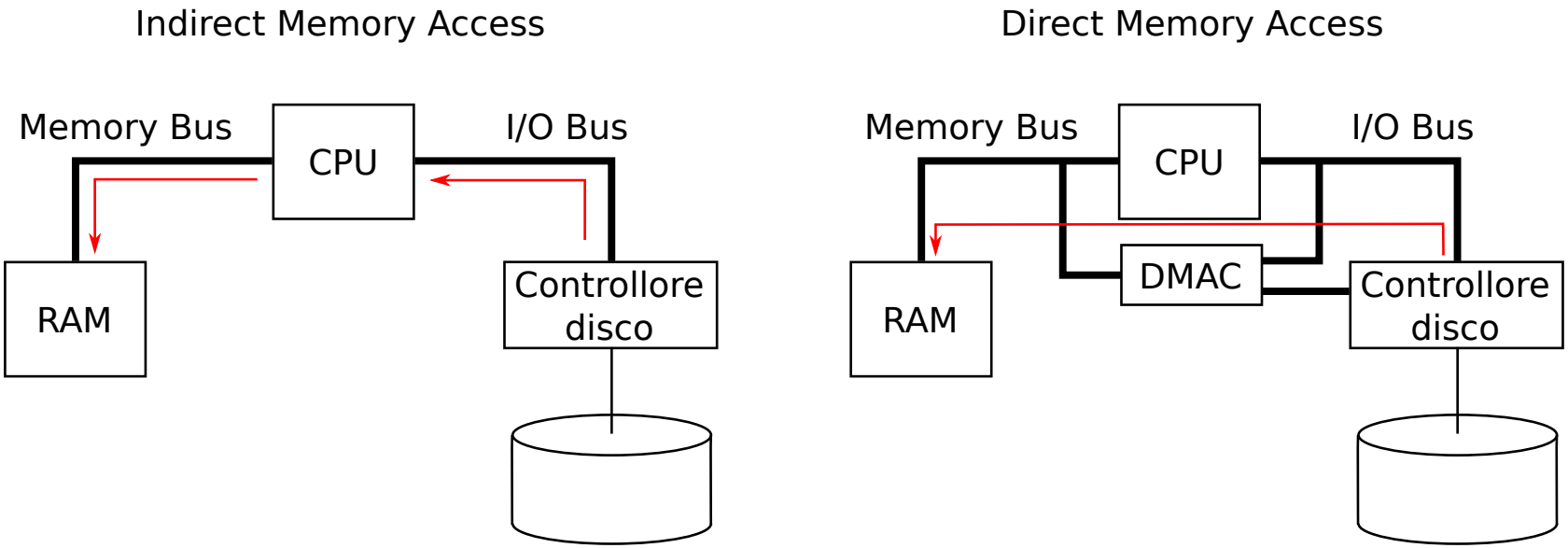
Monitoraggio Stanza (6)

```
91      pop R0
92      ret
93  ; Calcola la media secondo le temperature contenute nel vettore temperature
94  ; Restituisce in R0 la media calcolata
95  MEDIA:
96      push R1
97      movw #temperature, R1
98      xorw R0, R0
99      addw (R1), R0
100     addw 2(R1), R0
101     addw 4(R1), R0
102     addw 6(R1), R0
103     lsrw #2, R0
104     pop R1
105     ret
106 end
```

Direct Memory Access Controller (DMAC)

- La maggior parte delle interazioni tra un dispositivo di I/O e il processore avviene per trasferire dati (file).
- È un'operazione che non richiede capacità elaborative particolari: perché scomodare la CPU?!
- Ci si può appoggiare a dei *canali* che mettono in comunicazione diretta (e controllata) i dispositivi con la memoria
- Questa tecnica (chiamata *Direct Memory Access*, DMA) si basa su un controllore di canale (DMA Controller, DMAC) che gestisce la comunicazione tra periferica e memoria

Direct Memory Access Controller (DMAC)



Direct Memory Access Controller (DMAC)

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

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

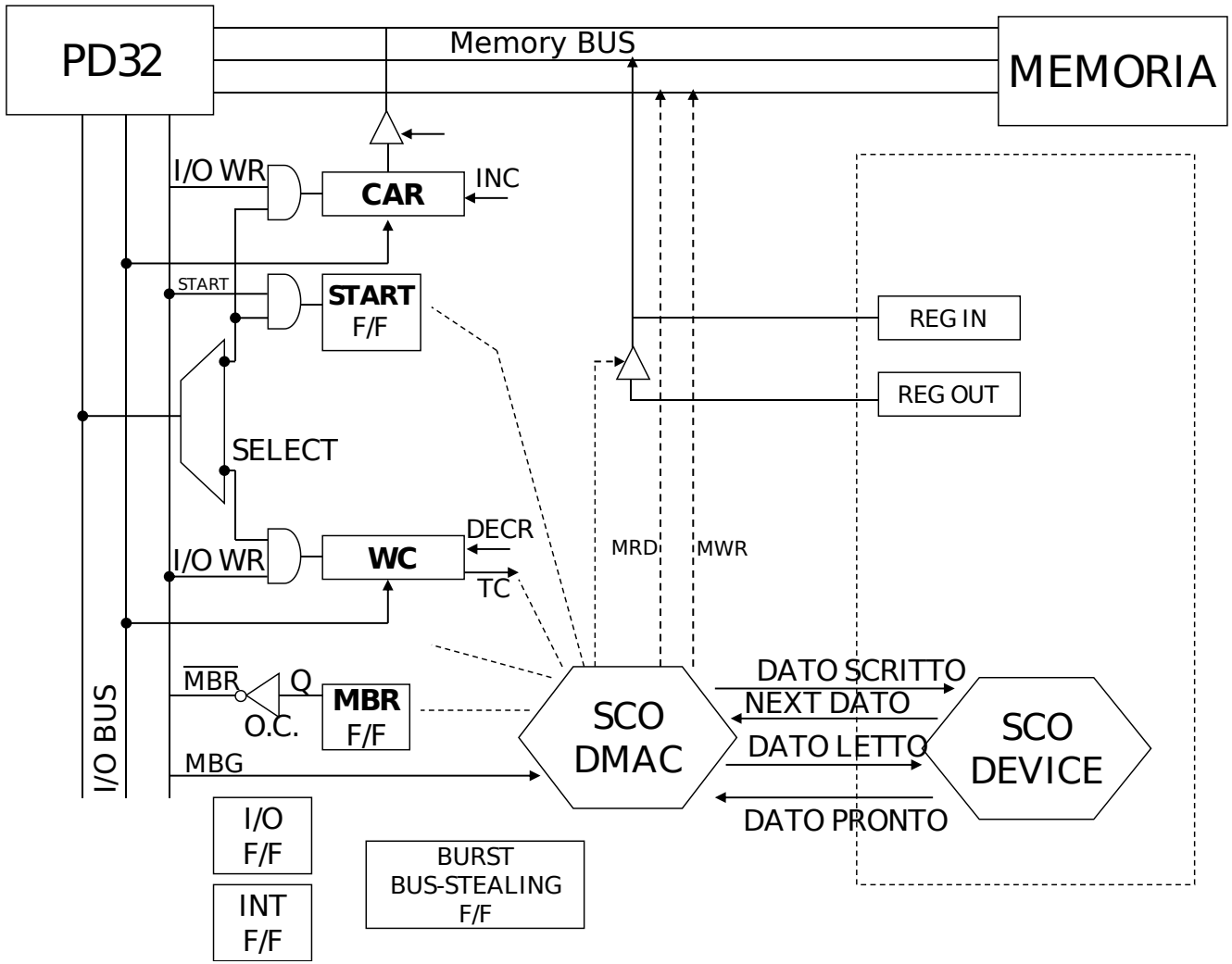
Direct Memory Access Controller (DMAC)

- L'utilizzo del DMAC permette al processore di proseguire nella sua attività elaborativa durante il trasferimento dei dati
- Il DMAC può essere visto quindi come una periferica speciale che è connessia sia all'I/O Bus che al Memory Bus
- Cosa avviene se, durante il trasferimento dati, il processore ha la necessità di accedere a dati in memoria?

Direct Memory Access Controller (DMAC)

- L'utilizzo del DMAC permette al processore di proseguire nella sua attività elaborativa durante il trasferimento dei dati
- Il DMAC può essere visto quindi come una periferica speciale che è connessia sia all'I/O Bus che al Memory Bus
- Cosa avviene se, durante il trasferimento dati, il processore ha la necessità di accedere a dati in memoria?
  - Il processore deve essere a conoscenza della presenza del DMAC
  - È necessario prevedere un protocollo di comunicazione CPU–DMAC che garantisca l'accesso esclusivo al Memory Bus
  - Si basa sull'utilizzo di due segnali di interfaccia, $\overline{\texttt{MBR}}$ (Memory Bus Request) e `MBG` (Memory Bus Grant)

Direct Memory Access Controller (DMAC)



DMAC: Inizializzazione

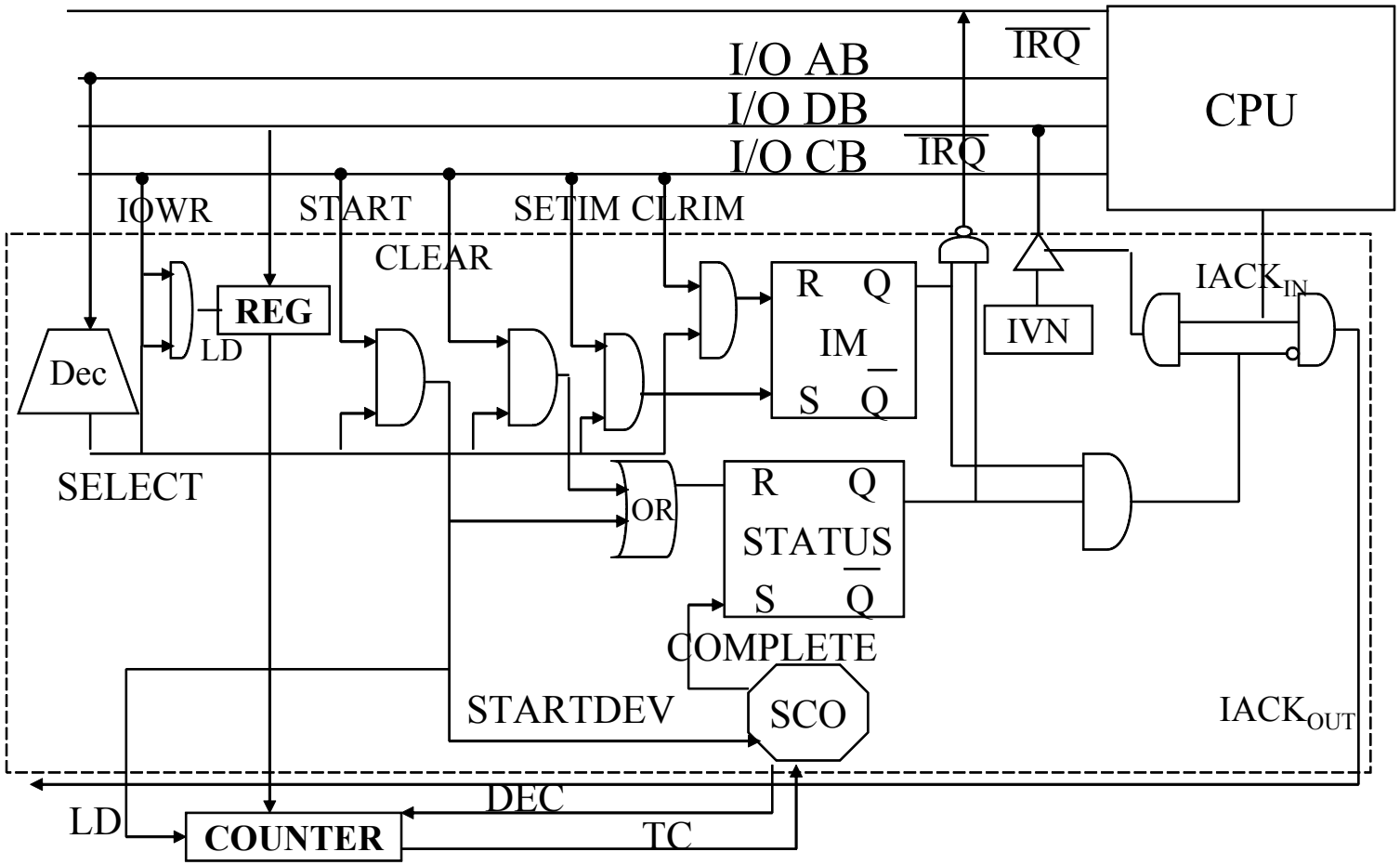
```
1 movl #100, R0 ; carica in R0 il numero di word da leggere
2 outl R0, WCOUNTER ; e passa il valore al WC (nel DMAC)
3 movl #2000, R0 ; carica in R0 l'indirizzo da cui leggere
4 outl R0, CAREGISTER ; e passa il valore al CAR (nel DMAC)
5 movl #1, R0
6 outl R0, DMACI/O ; programma il DMAC per la lettura
7 movl #0, R0
8 outl R0, DMACB-ST ; seleziona la modalita' (bus-stealing/burst)
9 start DMAC ; avvia trasferimento
```

Esercitazione sul DMAC

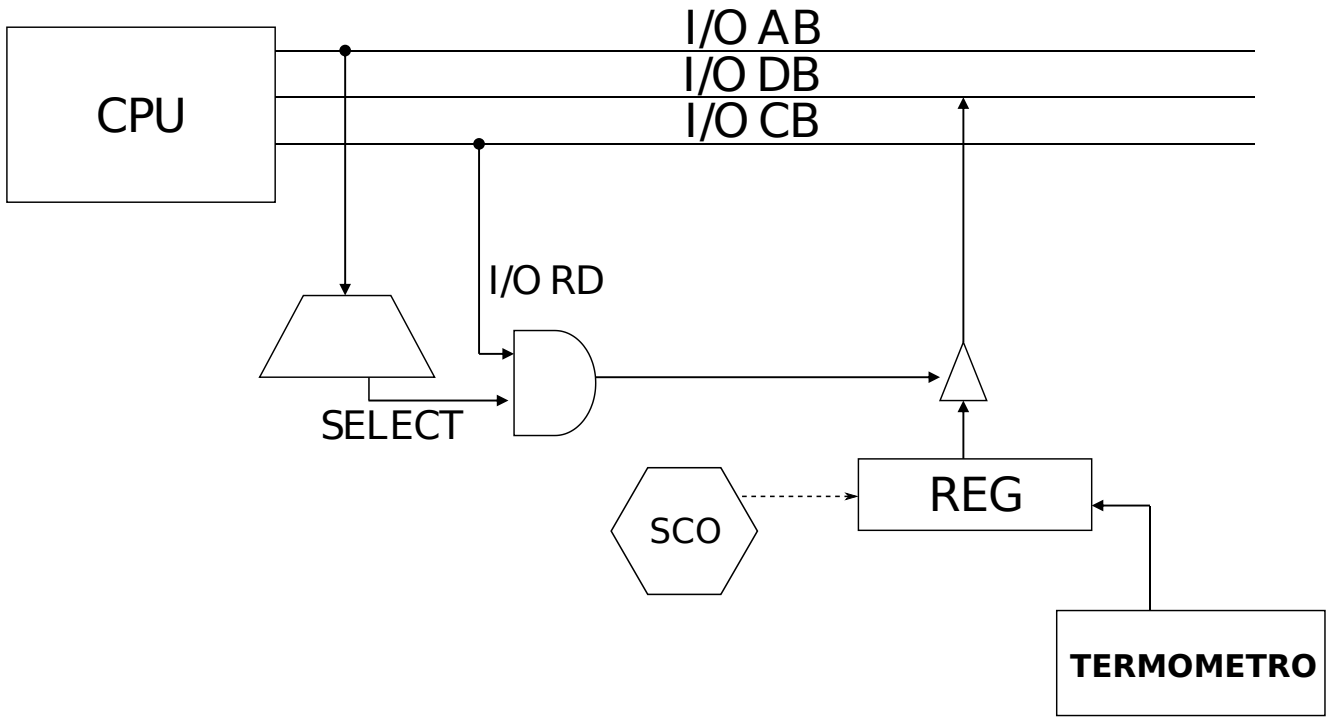
Dall'appello del 02/06/2000

Sia TIMER una periferica del processore PD32 programmata dallo stesso per richiedere un'interruzione ogni 10 millisecondi. Il servizio associato all'interruzione è il seguente: il processore deve controllare se il valore registrato nel registro d'interfaccia della periferica DEV_TEMPERATURA è maggiore di 40 gradi (la temperatura è espressa in binario utilizzando 8 bit). In caso positivo il processore disabilita la generazione di interruzioni da parte di TIMER e programma un DMAC per inviare un messaggio di allarme ad un MONITOR interfacciato al DMAC. Il messaggio è di 512 longword ed è memorizzato in un buffer di memoria di indirizzo iniziale BBBBh.
Al termine del trasferimento il DMAC avverte il processore dell'avvenuto trasferimento tramite interruzione ed il processore riattiva TIMER.
Progettare le interfacce di TIMER, DEV_TEMPERATURA e DMAC. Inoltre, implementare il software per attivare TIMER, programmare il DMAC, gestire l'interruzione di TIMER e quella del DMAC. Nella soluzione si supponga che la gestione del servizio associato alle interruzioni sia non interrompibile.
Progettare, in aggiunta, lo SCA della periferica TIMER.

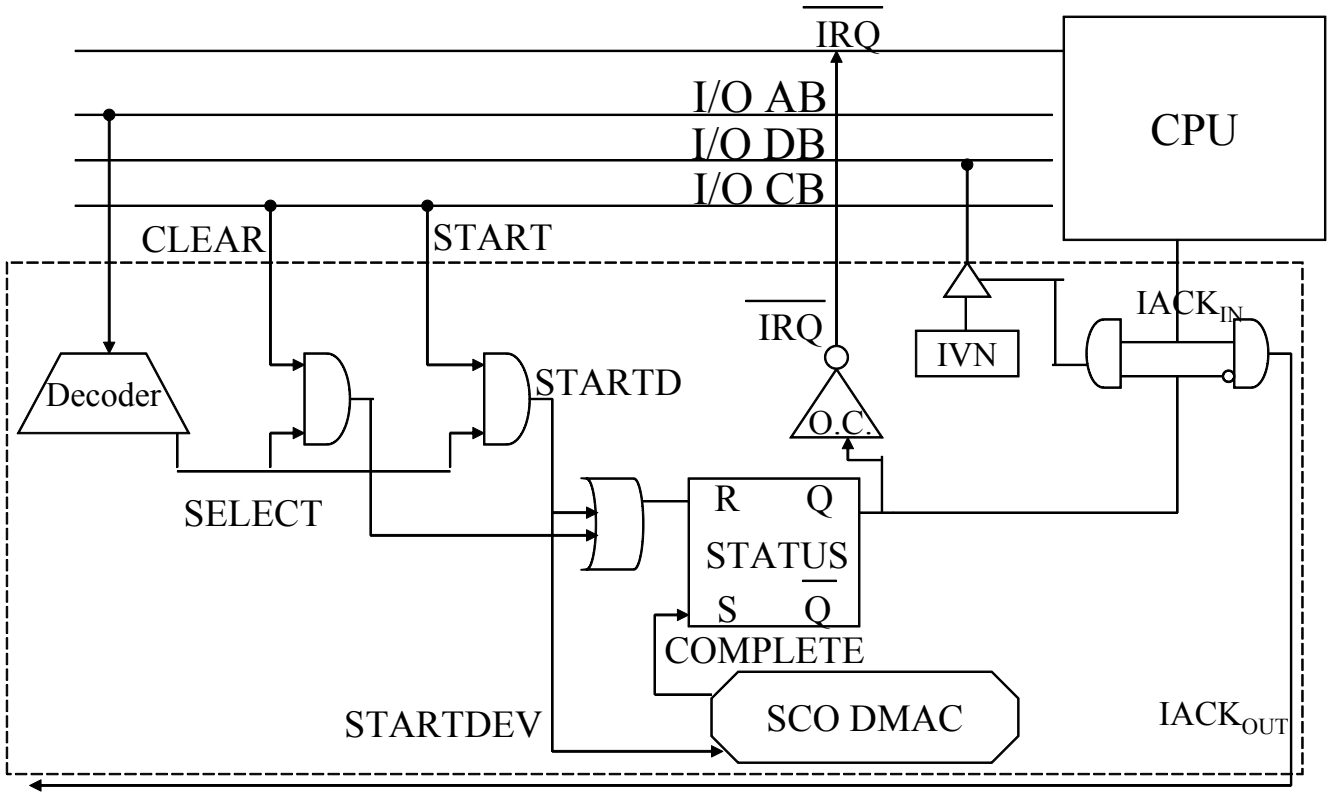
Interfaccia TIMER



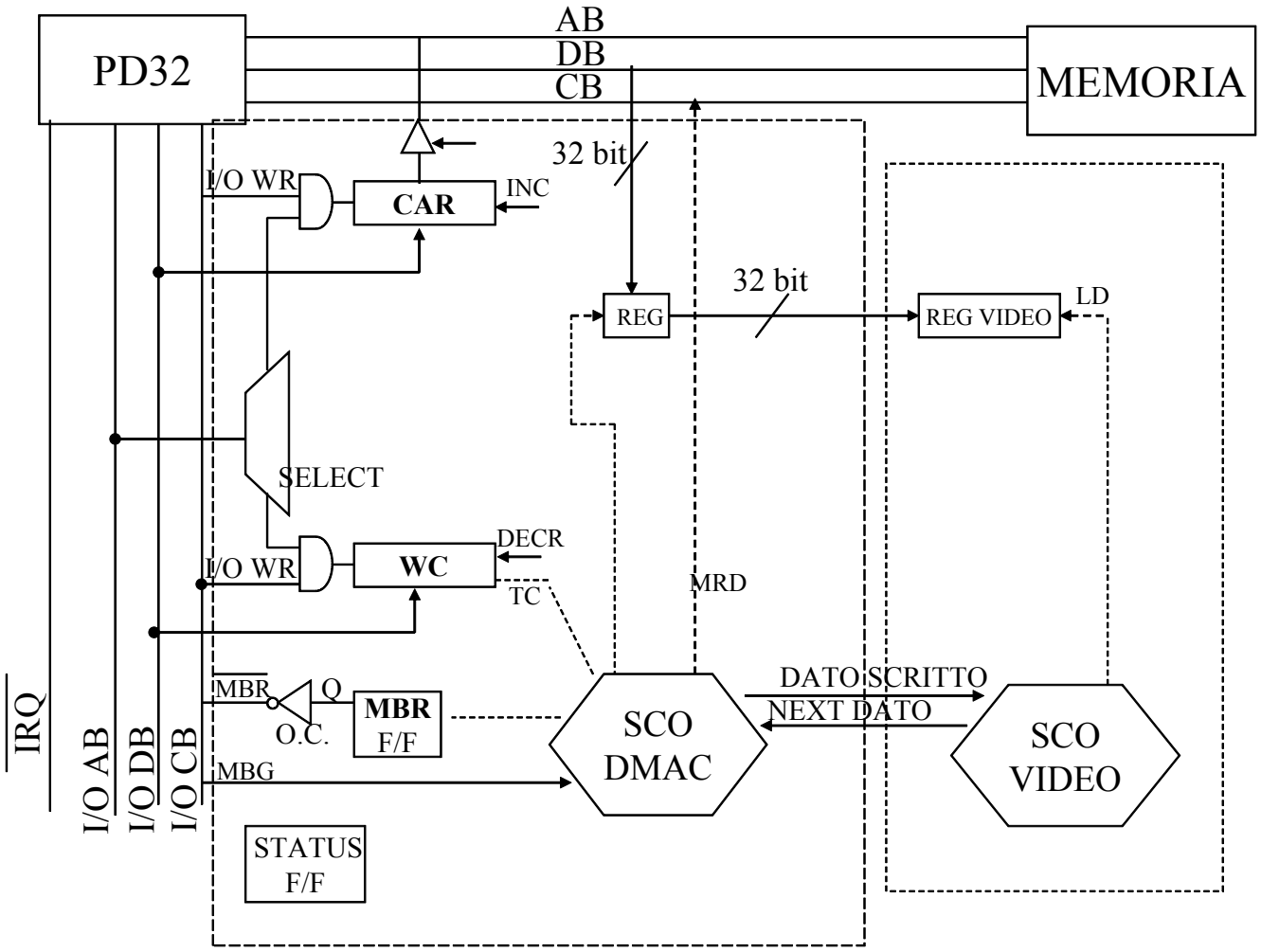
Interfaccia DEV_TEMPERATURA



Interfaccia DMAC per Interruzioni



Interfaccia DMAC–MONITOR



Inizializzazione di TIMER

```
1  org 400h
2      intervallo equ 100; l'intervallo (in msec) tra due interruzioni
3  code
4      outb #intervallo, TIMER ; configura e avvia TIMER
5      start TIMER
6      halt
7  driver 1, 500h ; TIMER
8      push R0
9      inb DEV\_TEMP, R0
10     cmpb #40, R0; R0 > #40 ? Si' se N=V
11     jn nset
12     jv minore
13   maggiore: ; devo disabilitare DEVICE e trasferire il video
14     clrim TIMER
15     outl #512, WC ; utilizzo il DMAC per il trasferimento
16     outl #BBBBh, CAR
```

Inizializzazione di TIMER (2)

```
17      start DMAC
18      jmp minore
19    nset:
20      jv maggiore
21    minore: ; non devo fare niente
22      pop R0
23      start TIMER
24      rti
25
26  driver 2, 600h ; DMAC
27      setim TIMER
28      clear DMAC
29      rti
```

Esercizio sulle interruzioni: Scheduler Round-Robin

- Lo *scheduler* (da *to schedule*, pianificare) è un programma che, dato un insieme di richieste di accesso ad una risorsa, stabilisce un ordinamento temporale per l'esecuzione di tali richieste
- Lo scheduler della CPU è un componente del sistema operativo che consente di eseguire più processi concorrentemente
- Lo scheduling basato su interruzioni si appoggia ad un timer che, ad intervalli regolari, invia un interruzione per comunicare che il processo attualmente in esecuzione ha già consumato il suo *quanto* di tempo
- Lo scheduler Round-Robin (*carosello*) assegna in maniera circolare un quanto di tempo a ciascun processo nel sistema

Scheduler Round-Robin: scelte di progetto

- Il PD32 programma una periferica TIMER per generare una richiesta di interruzioni ad intervalli regolati
- Quando il driver della periferica TIMER prende il controllo, viene salvato il contesto completo del processo in esecuzione
- Viene selezionato il processo successivo da mandare in esecuzione
- Il contesto del programma da mandare in esecuzione viene ripristinato
- L'esecuzione dell'istruzione `RTI` deve restituire il controllo al nuovo processo, non al precedente

Scheduler Round-Robin

```
1  org 400h
2      ; Variabili per memorizzare il contesto di processo
3      ; PC SR R0 R1 R2 R3 R4 R5 R6
4      proc1 dl 0, 20h, 0, 0, 0, 0, 0, 0, 0
5      proc2 dl 0, 20h, 0, 0, 0, 0, 0, 0, 0
6      proc3 dl 0, 20h, 0, 0, 0, 0, 0, 0, 0
7
8      ; Descrizione dei processi nel sistema
9      proc equ 3 ; Numero di processi totale
10     curr dl 0 ; Processo attualmente in esecuzione
11
12     ; Configurazione della periferica
13     timer equ 0h ; indirizzo del timer
14     quantum equ 1000; quanto di tempo assegnato al singolo processo
15
16     ; Dati processo 1
17     array1 dl 1, 2, 3, 4
18     array1dim equ 4
```

Scheduler Round-Robin (2)

```
19      ; Dati processo 2
20      num2 equ 1024
21
22      ; Dati processo 3
23      num3 equ 2048
24
25  code
26    main:
27      ; Inizializza il PC per ciascun processo
28      movl #PROG1, proc1
29      movl #PROG2, proc2
30      movl #PROG3, proc3
31
32      ; Configura il timer di sistema per generare interruzioni
33      setim timer
34      outl #quantum, timer
35
36
```

Scheduler Round-Robin (3)

```
37      ; Avvia il primo programma
38      xorl R0, R0 ; il processo 0 deve vedere tutti i registri azzerati
39      push #0
40      popsr
41
42      seti
43      start timer ; Avvia il timer
44
45      push proc1 ; Da' il controllo al processo 0
46      ret
47
48
49
50  ; Programma 1: scandisce gli elementi di un vettore e ne calcola la somma
51  PROG1:
52      movl #array1, R0
53      movl #array1dim, R1
54      xorl R2, R2
```

Scheduler Round-Robin (4)

```
55    cicloProg1:
56      cmpl #0, R1 ; Se arrivo alla fine, riparto dall'inizio!
57      jz PROG1
58      addl (R0)+, R2
59      subl #1, R1
60      jmp cicloProg1
61      halt
62
63  ; Programma 2: incrementa un contatore fino a raggiungere il valore num2
64  PROG2:
65      xorl R2, R2
66    cicloProg2:
67      addl #1, R2
68      cmpl #num2, R2
69      jz PROG2
70      jmp cicloProg2
71      halt
72
```

Scheduler Round-Robin (5)

```
73 ; Programma 3: imposta un contatore a num3 e lo decrementa fino a
       raggiungere 0
74 PROG3:
75     movl #num3, R4
76   cicloProg3:
77     subl #1, R4
78     cmpl #0, R4
79     jnz cicloProg3
80     jmp PROG3
81     halt
82
83 ; Driver del timer: e' qui che viene schedulato il processo successivo
84 driver 0, 1000h
85     ; Salva il contesto del processo corrente
86     ; L'interruzione ha scritto nello stack PC ed SR
87
88     push R0
89     push R1
```

Scheduler Round-Robin (6)

```
90      push R2
91
92      movl curr, R1
93      movl #36, R2 ; ogni vettore proc e' grande 36 byte
94      jsr MULTIPLY
95      addl #proc1, R0 ; R0 contiene l'indirizzo del primo elemento del
            contesto di curr
96
97      pop R2
98      pop R1
99
100     movl 4(R7), 4(R0) ; Salvo SR
101     movl 8(R7), (R0) ; Salvo PC
102     movl R1, 12(R0) ; Salvo R1
103     movl R2, 16(R0) ; Salvo R2
104     movl R3, 20(R0) ; Salvo R3
105     movl R4, 24(R0) ; Salvo R4
106     movl R5, 28(R0) ; Salvo R5
```

Scheduler Round-Robin (7)

```
107     movl R6, 32(R0) ; Salvo R6
108     movl R0, R1
109     pop R0
110     movl R0, 8(R1); Salvo R0
111
112     ; Seleziona il processo successivo da mandare in esecuzione
113     movl curr, R1
114     addl #1, R1
115     cmpl #proc, R1
116     jnz successivo
117     xorl R1, R1
118   successivo:
119     movl R1, curr
120
121     ; Ripristina il contesto del processo originale
122     ; PC ed SR vanno messi nello stack, saranno ripristinati da rti
123     movl #36, R2 ; ogni vettore proc e' grande 36 byte
124     jsr MULTIPLY
```

Scheduler Round-Robin (8)

```
125     addl #proc1, R0 ; R0 contiene l'indirizzo del primo elemento del
            contesto di curr
126
127     movl (R0), 4(R7)
128     movl 4(R0), (R7)
129     push 8(R0)
130     movl 12(R0), R1
131     movl 16(R0), R2
132     movl 20(R0), R3
133     movl 24(R0), R4
134     movl 28(R0), R5
135     movl 32(R0), R6
136     pop R0
137     start timer
138     rti
139
140
141
```

Scheduler Round-Robin (9)

```
142 ; Subroutine per la moltiplicazione
143 ; Parametri in R1 e R2
144 ; Valore di ritorno in R0
145 MULTIPLY:
146     xorl R0, R0
147   loopMult:
148     cmpl #0, R1
149     jz donemult
150     addl R2, R0
151     jc donemult
152     subl #1, R1
153     jmp loopMult
154   donemult:
155     ret
156 end
```