Progettazione di Periferiche e Programmazione di Driver

SAPIENZA
UNIVERSITÀ DI ROMA

Alessandro Pellegrini

Architettura dei Calcolatori Elettronici
Sapienza, Università di Roma

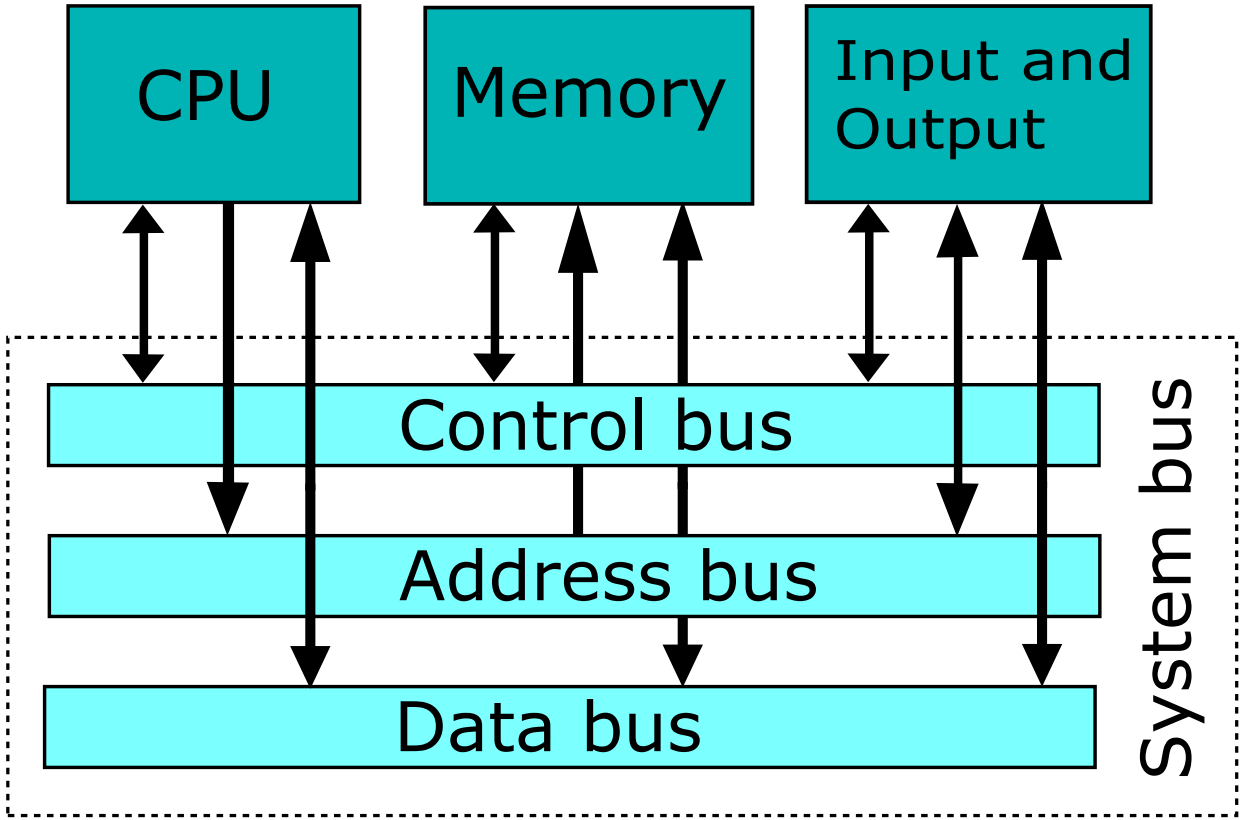
A.A. 2018/2019

Classe 7: Istruzioni di I/O

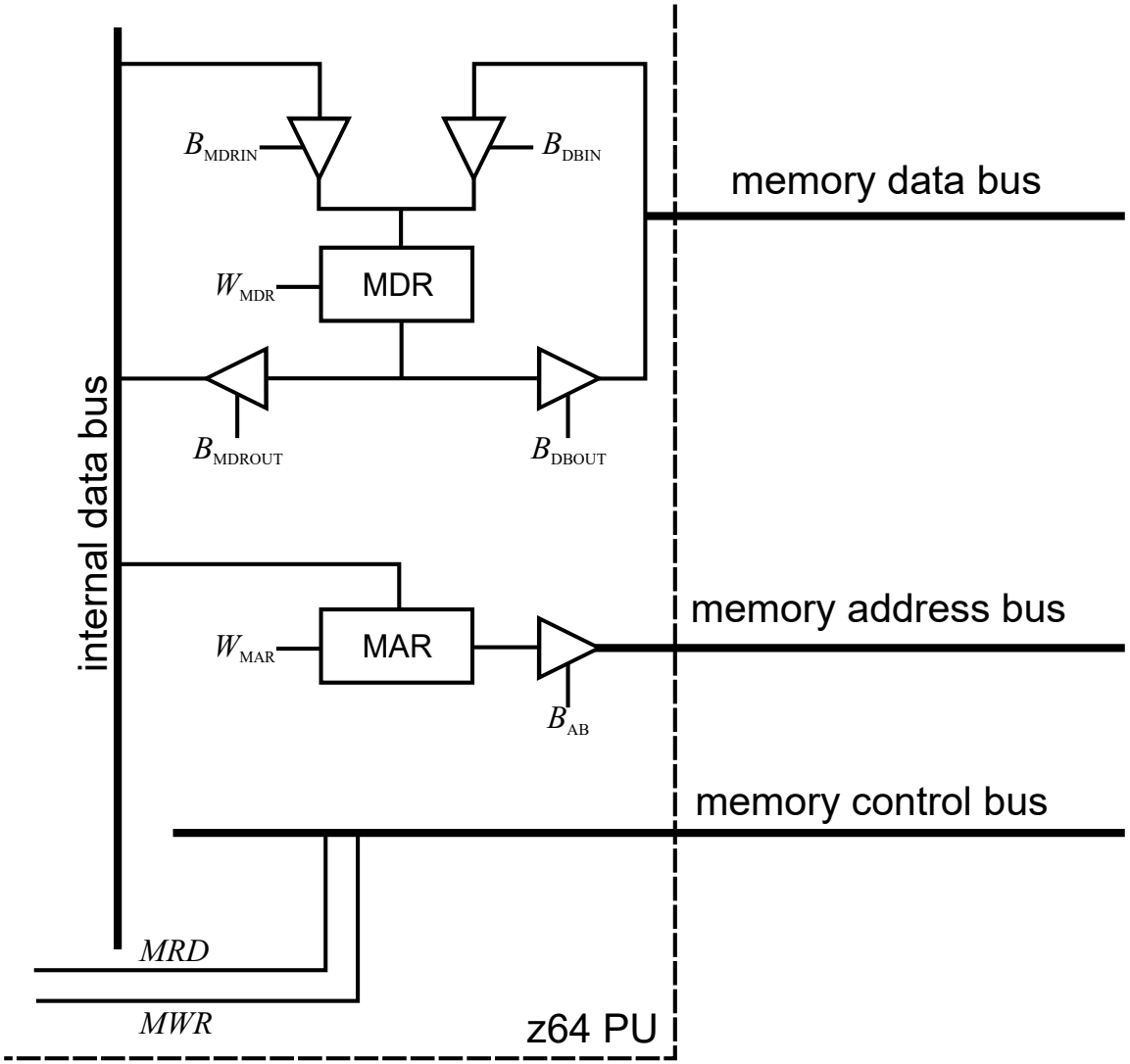
Tipo	Mnemonico	Operandi	C S Z O P	Commento
0	`inX`	%dx, RAX	- - - - -	Copia il dato dal buffer del device il cui indirizzo è in %dx in RAX
1	`outX`	RAX, %dx	- - - - -	Copia il dato da RAX nel buffer del device il cui indirizzo è in %dx
2	`insX`	%dx	- - - - -	Esegue una lettura di una stringa di dati
4	`outsX`	%dx	- - - - -	Esegue una scrittura di una stringa di dati

Attenzione: anche se il processore è a 64 bit, `X` non può identificare una quadword (inq, outq, insq, outsq non sono istruzioni valide)

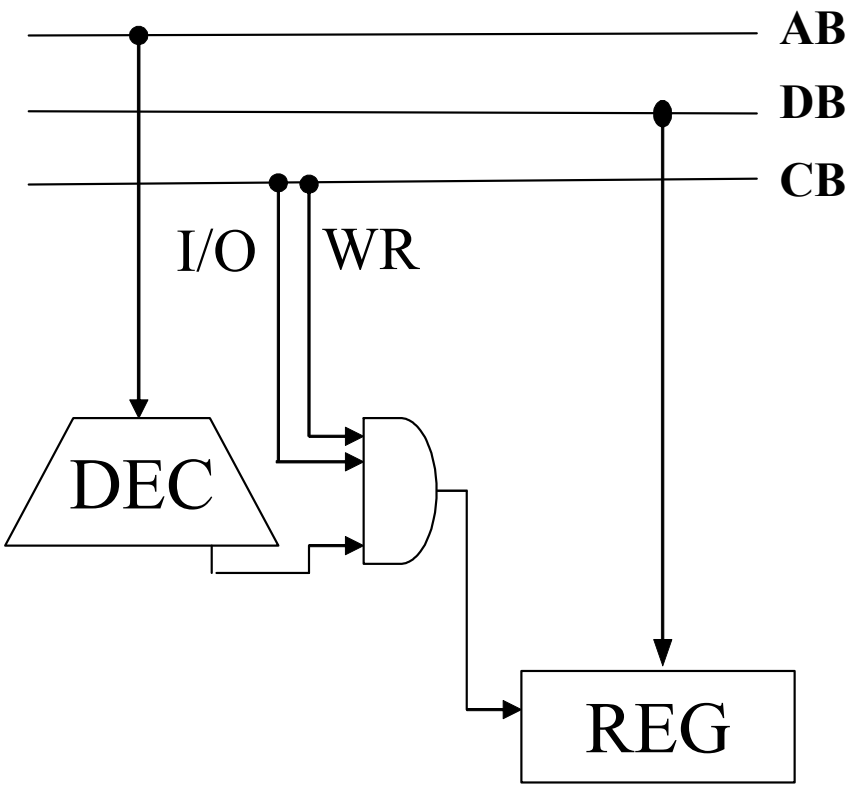
Trasferire dei dati: il BUS

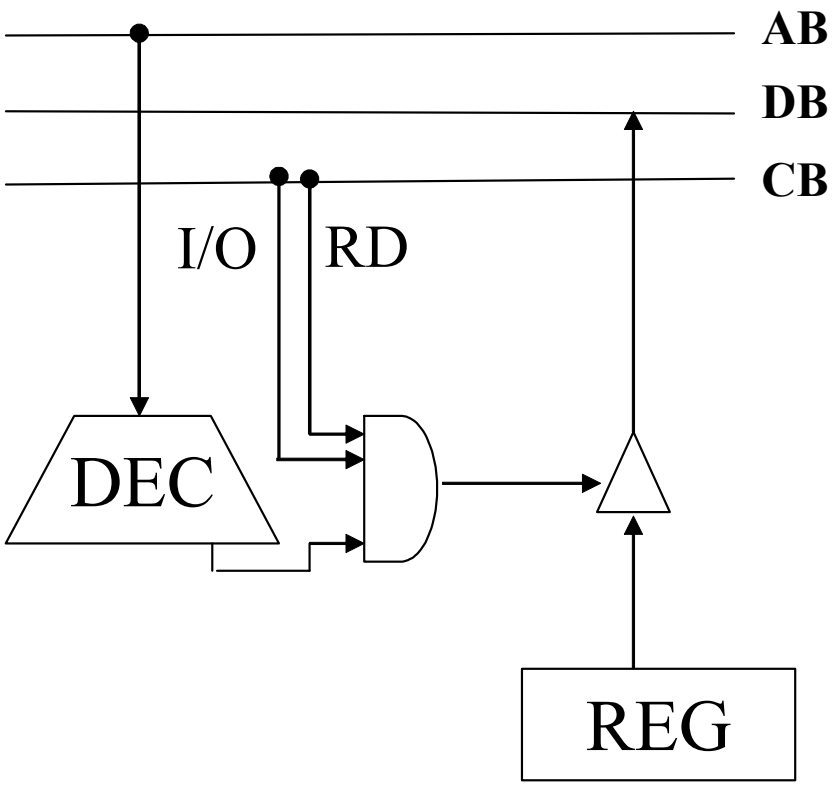


z64: Interazione con la memoria



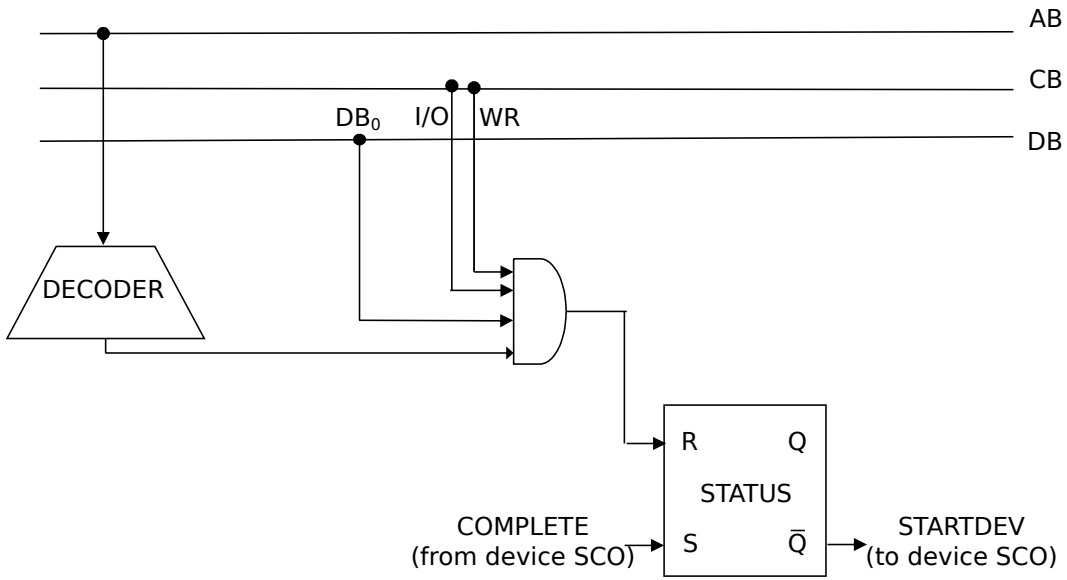
in e out in azione





Interazione con le periferiche

- Si può interagire con le periferiche in due modi
  - *Modo sincrono*: il codice del programma principale si mette in attesa della periferica
  - *Modo asincrono*: la periferica informa il sistema che una qualche operazione è stata completata
- In entrambi i casi, la periferica deve poter memorizzare il suo stato



Busy Waiting

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

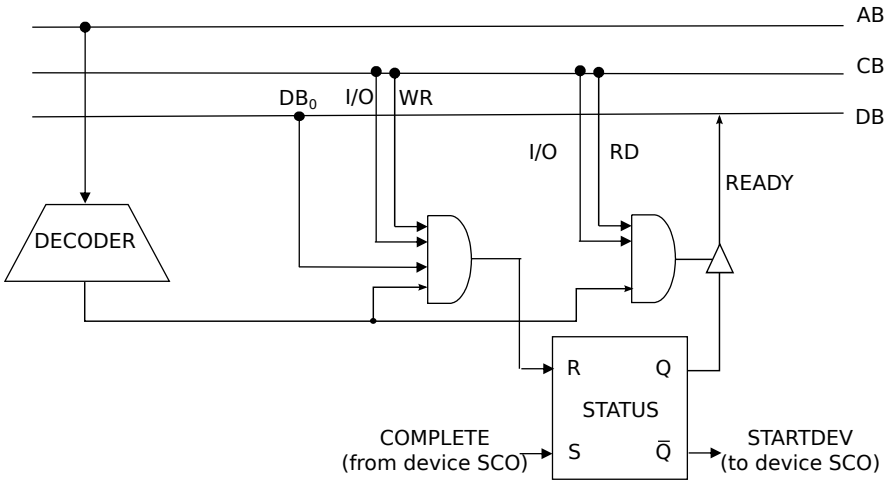
Busy Waiting

```
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

```
1 # Avvia la periferica
2     movw $STATUS, %dx
3     movb $1, %al
4     outb %al, %dx
5 # Cicla in attesa che essa sia pronta
6 .bw:
7     inb %dx, %al
8     btb $0, %al
9     jnc .bw
```



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      movw $STATUS_DEV1, %dx
3      inb %dx, %al
4      btb $0, %al
5      jc .dev1
6      movw $STATUS_DEV2, %dx
7      inb %dx, %al
8      btb $0, %al
9      jc .dev2
10     # ...
11     jmp .poll
```

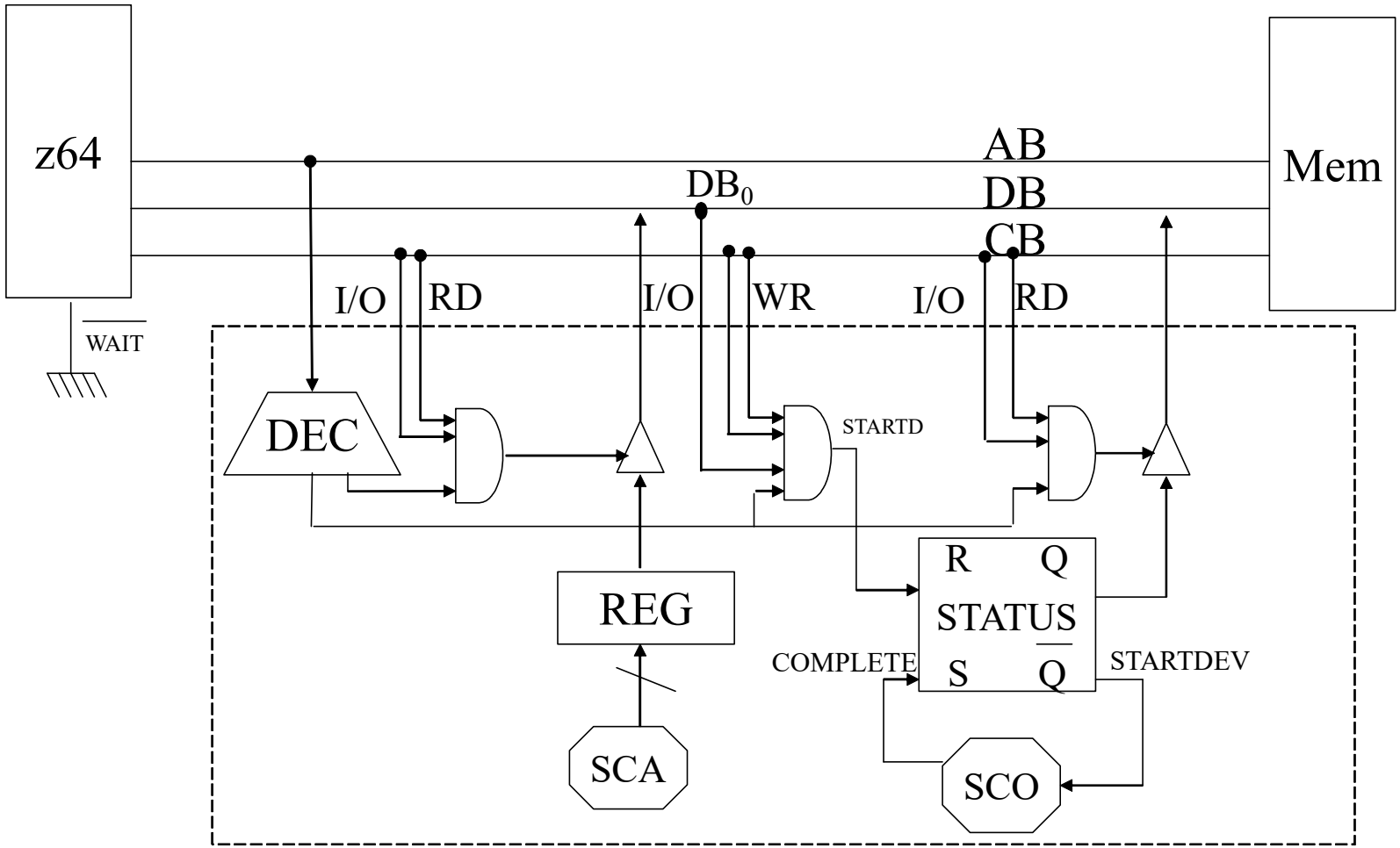
Progettazione dell'interfacciamento con le periferiche

- 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
- Nello z64, 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

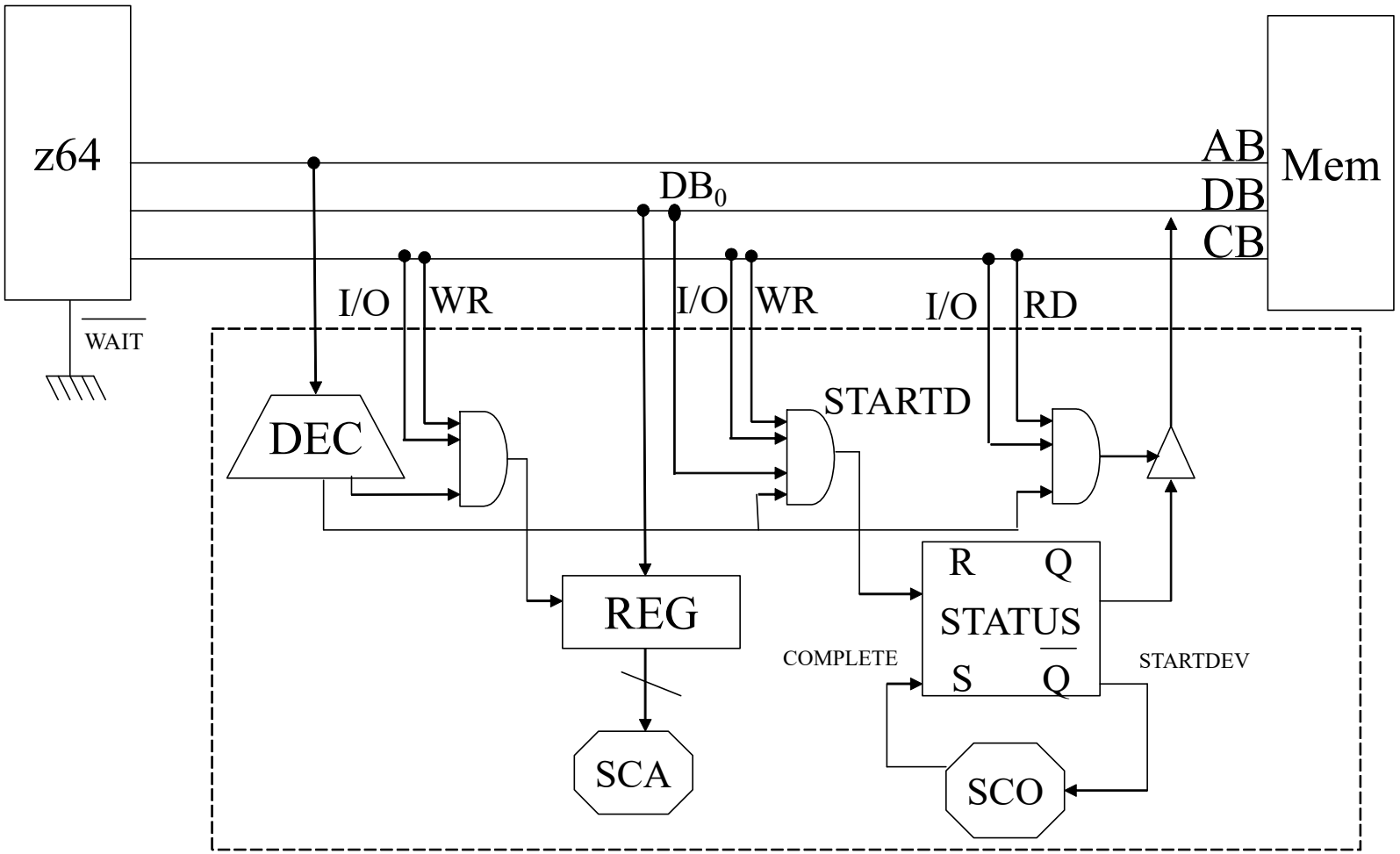
I/O programmato: handshaking manuale.

```
1     movw $status, %dx
2  .loop1:
3     inb %dx, %al
4     btb $0, %al
5     jnc .loop1
6     movb $1, %al
7     outb %al, %dx
8  .loop2:
9     inb %dx, %al
10    btb $0, %al
11    jnc .loop2
12    movw $device_reg, %dx
13    inl %dx, %eax
```

1. Aspetto che la periferica sia disponibile
2. Avvio la periferica così che possa produrre informazioni
3. Aspetto che la periferica completi la sua unità di lavoro
4. Acquisisco dalla periferica il risultato dell'operazione

- Il primo ciclo di busy waiting può non essere necessario!

Interfaccia di Output



Interfaccia di Output: Software

I/O programmato: handshaking manuale.

```
1     movw $status, %dx
2  .loop1:
3     inb %dx, %al
4     btb $0, %al
5     jnc .loop1
6     movw $device_reg, %dx
7     movl $DATO, %eax
8     outl %eax, %dx
9     movw $status, %dx
10    movb $1, %al
11    outb %al, %dx
12 .loop2:
13    inb %dx, %al
14    btb $0, %al
15    jnc .loop2
```

1. Aspetto che la periferica sia disponibile
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 z64. Scrivere il codice di una subroutine `in_AD1` (secondo le convenzioni di chiamata System V ABI) che accetti come parametri il numero di dati (word) da leggere dalla periferica `AD1`, e l'indirizzo di memoria da cui il processore z64 dovrà incominciare a scrivere i dati così acquisiti da periferica. Scrivere inoltre il programma che invoca la funzione `in_AD1` chiedendo di acquisire 100 word dalla periferica `AD1` e di memorizzarli in un vettore posto a partire dall'indirizzo 0x1200. La dimensione massima del vettore è 400 word.

Esercizio Busy Waiting: Soluzione I

```
1  .org 0x1200
2  .data
3      .equ AD1_status, 0x0000 # indirizzo di STATUS
4      .equ AD1_reg, 0x0001 # indirizzo del registro
5      vettore: .fill 400, 2
6  .org 0x800
7  .text
8      movq $100, %rdi
9      movq $vettore, %rsi
10     call in_AD1
11     hlt
12 in_AD1:
13     push %rbx # RBX e' callee-save
14     xorq %rbx, %rbx # Indice del vettore
15     movw $AD1_status, %dx
16   .bw1:
17     inb %dx, %al # Non so se AD1 e' gia' attiva
```

Esercizio Busy Waiting: Soluzione II

```
18    btb $0, %al
19    jnc .bw1
20  .acquisizione:
21    movb $1, %al
22    outb %al, %dx # Chiedo ad AD1 di produrre un nuovo dato
23  .bw2:
24    inb %dx, %al # Attendo la produzione del dato
25    btb $0, %al
26    jnc .bw2
27    movw $AD1_reg, %dx # Recupero il dato
28    inw %dx, %ax
29    movw %ax, (%rsi, %rbx, 2) # Salva nel vettore
30    addq $1, %rbx # Verifica se siamo alla fine
31    cmpq %rbx, %rdi
32    jnz .acquisizione
33    pop %rbx; ret
```

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:

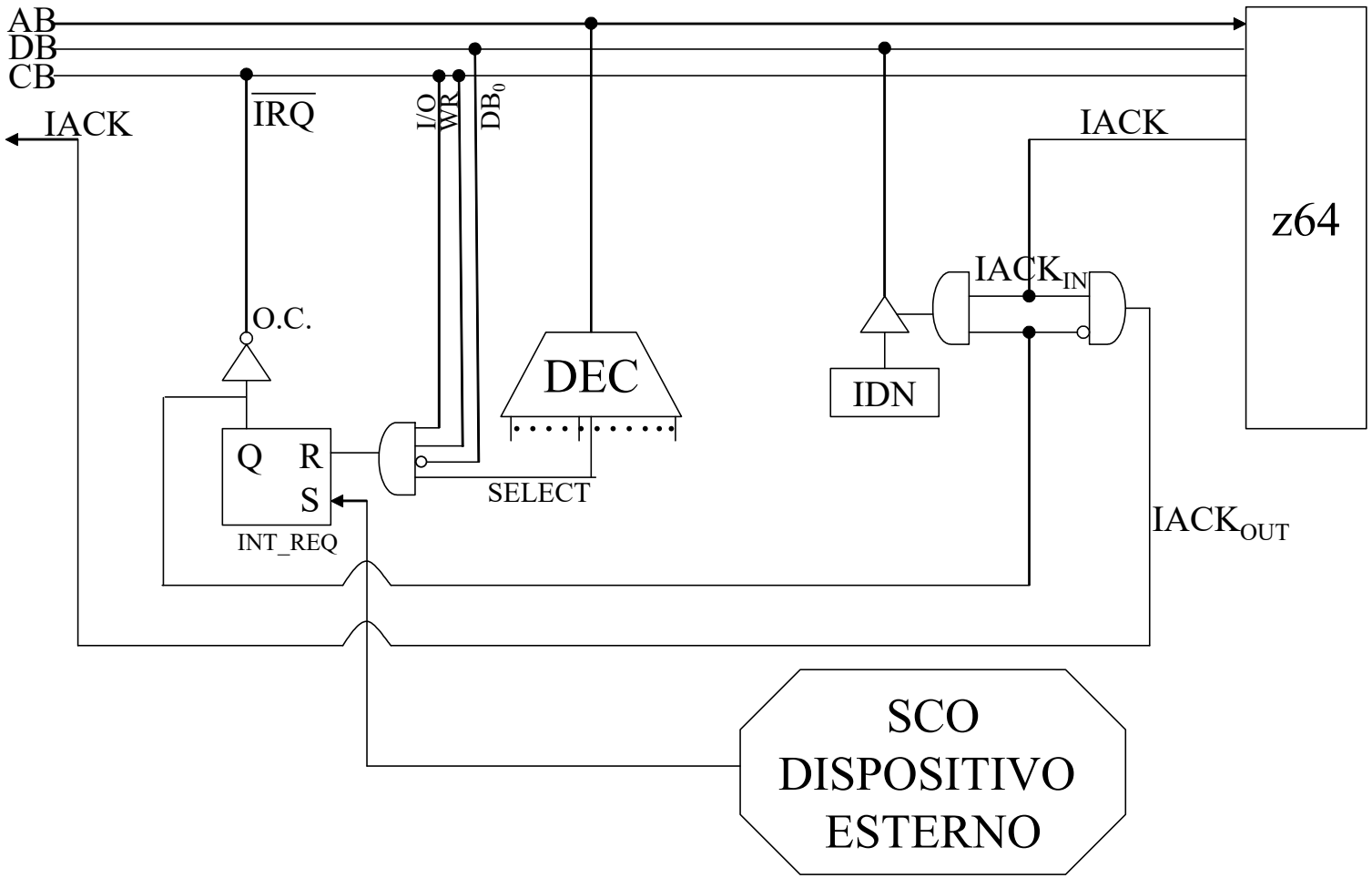
1. *Salvataggio del contesto d'esecuzione*
2. *Identificazione dell'origine dell'interruzione*
3. *Definizione della gerarchia di priorità*

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)

Interrupt Request e Interrupt Acknowledge



Cambio di contesto

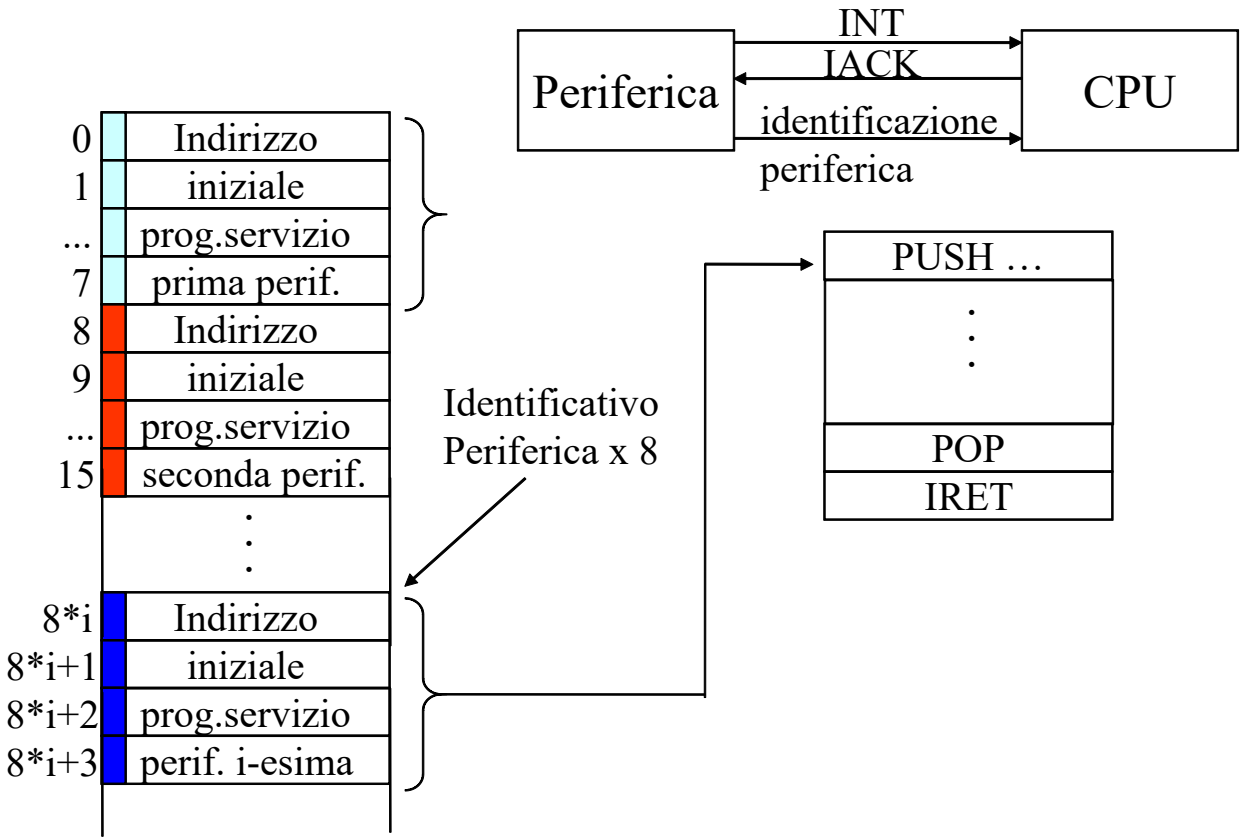
- Il *contesto* d'esecuzione di un processo è costituito da:
  - Registro `RIP`: contiene l'indirizzo dell'istruzione dalla quale si dovrà riprendere l'esecuzione una volta terminata la gestione dell'interrupt
  - Registro `FLAGS`: alcuni bit di condizione potrebbero non essere ancora stati controllati dal processo
  - Altri registri (p. es., `TEMP1`, `TEMP2`), per supportare la ripresa dell'esecuzione di operazioni logico/aritmetiche. La gestione al termine del ciclo istruzione previene la necessità di salvarne il contenuto
- Quando viene generata un'interruzione avviene una commutazione dal contesto del processo interrotto a quello del driver
- Analogamente il contesto del processo interrotto deve essere ripristinato una volta conclusa l'esecuzione del driver

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 lo z64 provvede a salvare nello stack i registri `FLAGS` e `RIP`
- Infine, in `RIP` viene caricato l'indirizzo della routine di servizio (driver) della periferica che ha generato la richiesta di interruzione.

Identificazione del driver (IDT)

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



Gestione di un'interruzione

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

Ritorno da un'interruzione

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

Interrupt nel mondo reale

- La gestione degli interrupt è un punto critico per le architetture, ed è uno dei punti critici di interconnessione tra hardware e software
  - La IDT, 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 memorizza informazioni critiche disponibili soltanto al momento dell'interrupt e schedula l'esecuzione di una bottom half non appena possibile
  - L'esecuzione di una top half blocca temporaneamente l'esecuzione di tutti gli altri processi del sistema: si cerca di ridurre al minimo il tempo d'esecuzione di una top half

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

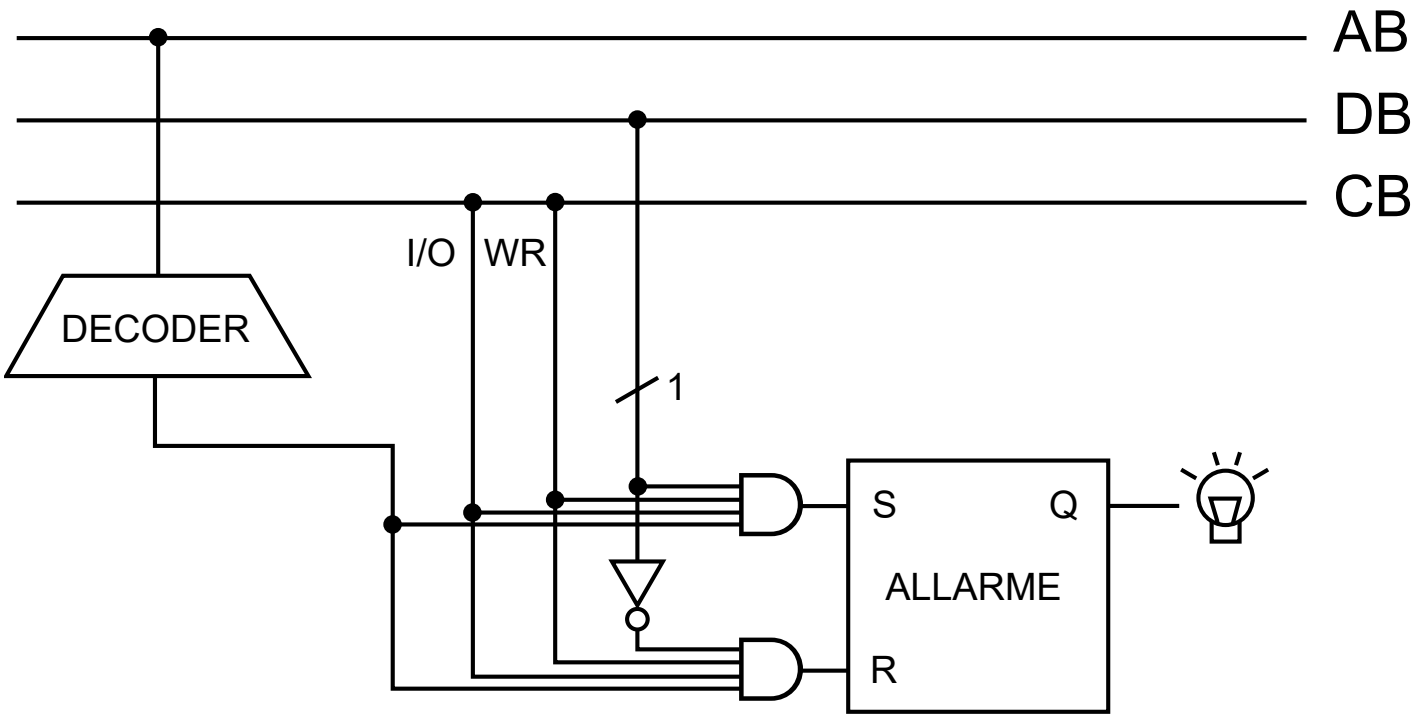
Esercizio sulle interruzioni: Monitoraggio Stanza

Una stanza è monitorata da quattro sensori di temperatura, che sono pilotati da un processore z64. 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 bit meno significativo del bus dati accende/spegne l'allarme quando viene settato/resettato.
- ALLARME è una periferica *passiva*: non ha Flip/Flop di status

Interfaccia ALLARME



Monitoraggio Stanza I

```
1  .org 0x800
2  .data
3      .equ sensore1_reg, 0x0
4      .equ sensore2_reg, 0x1
5      .equ sensore3_reg, 0x2
6      .equ sensore4_reg, 0x3
7      .equ sensore1_status, 0x4
8      .equ sensore2_status, 0x5
9      .equ sensore3_status, 0x6
10     .equ sensore4_status, 0x7
11     .equ sensore1_irq, 0x8
12     .equ sensore2_irq, 0x9
13     .equ sensore3_irq, 0xa
14     .equ sensore4_irq, 0xb
15     .equ allarme, 0xc
16     .equ tMin, 200 # tMin espresso in decimi di gradi Celsius
17     .equ tMax, 300 # tMax espresso in decimi di gradi Celsius
```

Monitoraggio Stanza II

```
18      temperature: .word 250, 250, 250, 250 # vettore contenente le 4
            temperature misurate
19  .text
20      xorw %r8w, %r8w # nuova media
21      xorw %r9w, %r9w # vecchia media
22      movb $1, %al
23      movw $sensore1_status, %dx # Avvia i sensori
24      outb %al, %dx
25      movw $sensore2_status, %dx
26      outb %al, %dx
27      movw $sensore3_status, %dx
28      outb %al, %dx
29      movw $sensore4_status, %dx
30      outb %al, %dx
31      sti
32    .loop:
33      movw %r8w, %r9w # vecchia media = nuova media
34      call media # calcola la media delle misure correnti
35      movw %ax, %r8w
36      cmpw %r8w, %r9w
```

Monitoraggio Stanza III

```
37      jz .loop # se la media non e' cambiata, non faccio nulla
38      movb $1, %al # rdx = 1 --> allarme acceso
39      cmpw $tMax, %r8w
40      jnc .set # tMax <= %r8
41      cmpw $tMin, %r8w;
42      jc .set # tMin > %r8
43      xorb %al, %al # rdx = 0 --> allarme spento
44    .set:
45      movw $allarme, %dx
46      outb %al, %dx # accende o spegne l'allarme
47      jmp loop
48
49  .driver 1
50      pushw %dx # Il programma principale usa dx!
51      movq $sensore1_reg, %rdi
52      movq $sensore1_status, %rsi
53      movq $sensore1_irq, %rdx
54      call get
55      popw %dx
56      iret
```

Monitoraggio Stanza IV

```
57 .driver 2
58     pushw %dx
59     movq $sensore2_reg, %rdi
60     movq $sensore2_status, %rsi
61     movq $sensore2_irq, %rdx
62     call get
63     popw %dx
64     iret
65 .driver 3
66     pushw %dx
67     movq $sensore3_reg, %rdi
68     movq $sensore3_status, %rsi
69     movq $sensore3_irq, %rdx
70     call get
71     popw %dx
72     iret
73 .driver 4
74     pushw %dx
75     movq $sensore4_reg, %rdi
76     movq $sensore4_status, %rsi
```

Monitoraggio Stanza V

```
77      movq $sensore4_irq, %rdx
78      call get
79      popw %dx
80      iret
81
82
83  # La subroutine get recupera da un sensore la temperatura misurata
84  # Memorizza in temperature[sensore] la temperatura misurata
85  # Riavvia l'acquisizione di una nuova misura
86  # %rdi: *_reg
87  # %rsi: *_status
88  # %rdx: *_irq
89  get:
90      push %rax # Viene usato dal programma principale
91      push %rdx # Ci serve per accedere al dispositivo
92      movw %di, %dx
93      inw %dx, %ax
94      movw %ax, temperature(, %rdi, 2)
95      movw %si, %dx
96      movb $1, %al
```

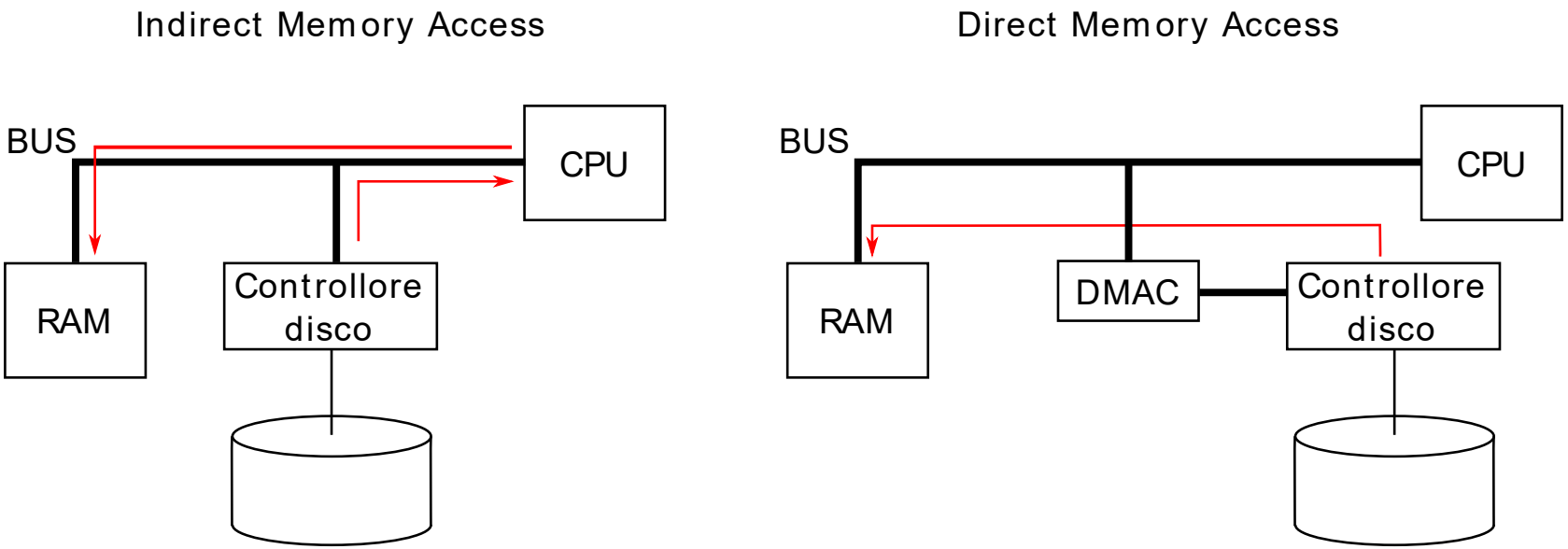
Monitoraggio Stanza VI

```
97      outb %al, %dx # Riavvia l'interfaccia
98      pop %rdx # Ripristina l'indirizzo di *_irq
99      movb $0, %al # Cancella la causa di interruzione
100     outb %al, %dx
101     pop %rax
102     ret
103
104 # Calcola la media secondo le temperature contenute nel vettore temperature
105 # Restituisce in %ax la media calcolata
106 media:
107     movq $temperature, %rdx
108     xorw %ax, %ax
109     addw (%rdx), %ax
110     addw 2(%rdx), %ax
111     addw 4(%rdx), %ax
112     addw 6(%rdx), %ax
113     shrw $2, %ax
114     ret
```

Direct Memory Access Controller (DMAC)

- Molte 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`)
- 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)

DMAC dispositivo-memoria

- Esistono delle istruzioni ottimizzate per programmare il DMAC di sistema:
  - `insX`
  - `outsX`
- Sono istruzioni di tipo *stringa*, pertanto:
  - RCX contiene il numero di blocchi di dati da leggere/scrivere
  - RDI contiene l'indirizzo destinazione (per la `insX`)
  - RSI contiene l'indirizzo sorgente (per la `outsX`)
  - DF indica la direzione

`insX`: leggi 10 byte da DEV

```
1 movq $10, %rcx
2 movq $dest, %rdi
3 movq $dev_mem, %dx
4 cld
5 insb
```

`insX`: scrivi 10 byte su DEV

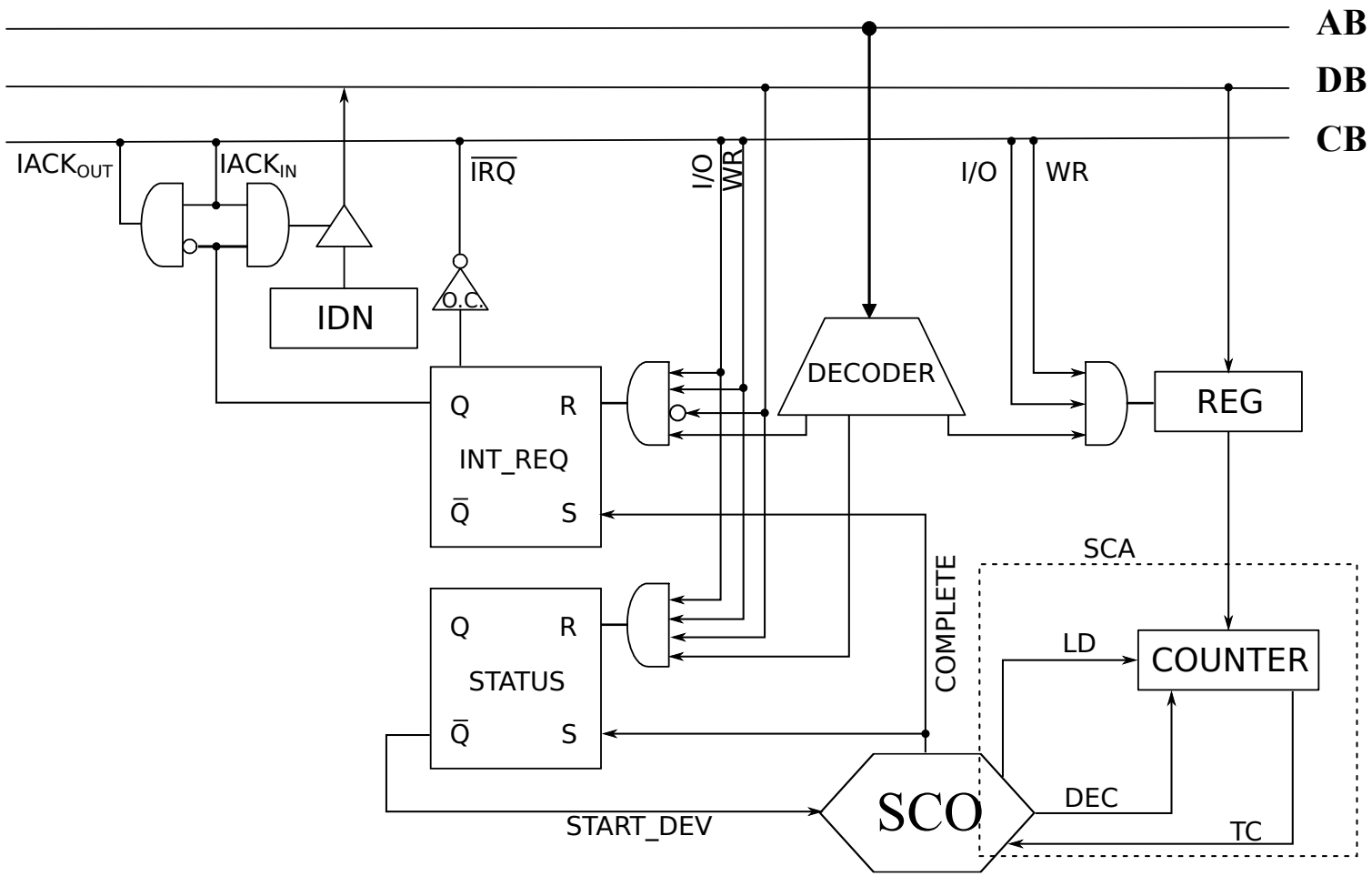
```
1 movq $10, %rcx
2 movq $dest, %rsi
3 movq $dev_mem, %dx
4 cld
5 outsb
```

Esercitazione sul DMAC

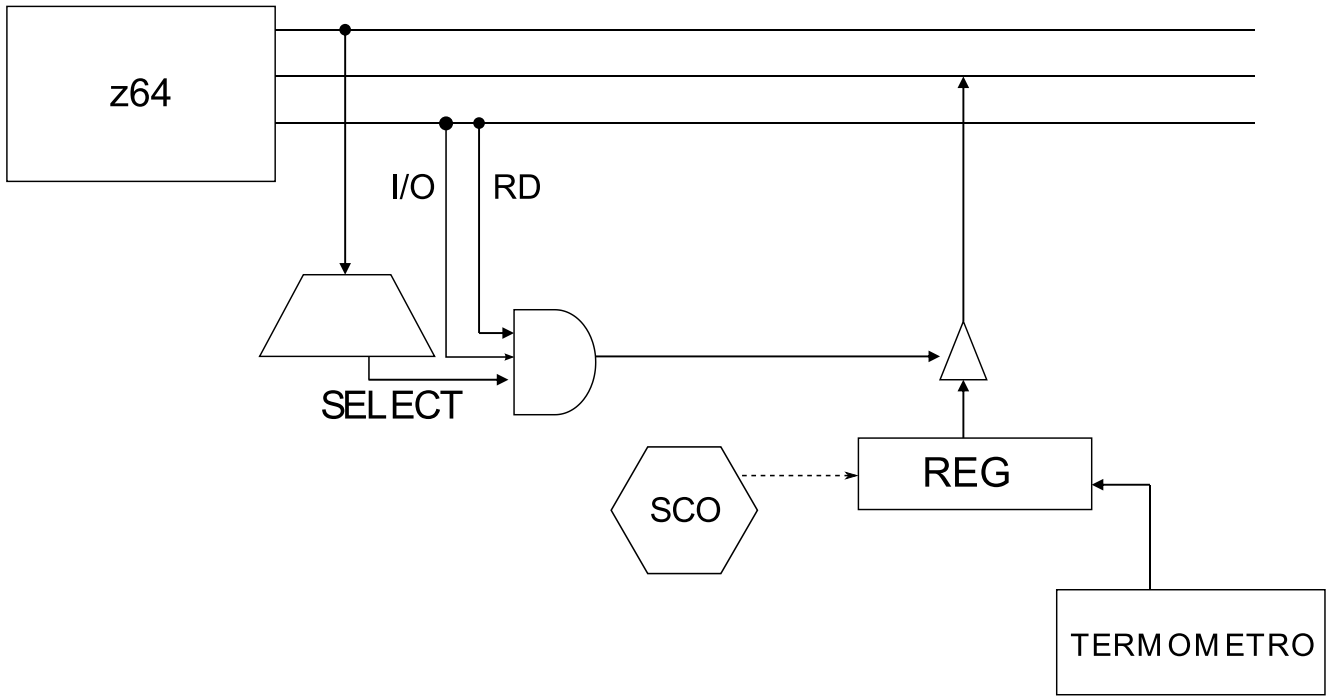
Dall'appello del 02/06/2000

Sia TIMER una periferica del processore z64 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 TEMPERATURA è maggiore di 40 gradi (la temperatura è espressa senza segno in binario utilizzando 8 bit). In caso positivo il processore programma un DMAC per inviare un messaggio di allarme ad un MONITOR interfacciato al DMAC. Il messaggio è di 512 byte ed è memorizzato in un buffer di memoria di indirizzo iniziale BBBBh.
Al termine del trasferimento attraverso il DMAC il processore riattiva TIMER.
Progettare le interfacce di TIMER e TEMPERATURA. Inoltre, implementare il software per attivare TIMER, programmare il DMAC e gestire l'interruzione di TIMER. Nella soluzione si supponga che la gestione del servizio associato alle interruzioni sia non interrompibile.

Interfaccia TIMER



Interfaccia DEV_TEMPERATURA



Software I

```
1  .org 0xBBBB
2  .data
3      messaggio: .fill 512, 1 # 512 byte
4      .equ intervallo, 10 # intervallo (in ms) tra due interruzioni
5      .equ TEMPERATURA_REG, 0x00
6      .equ TIMER_REG, 0x01
7      .equ TIMER_INT_REQ, 0x02
8      .equ TIMER_STATUS, 0x03
9      .equ VIDEO, 0x04
10
11 .org 0x800
12 .text
13     movw $TIMER_REG, %dx # Configura TIMER
14     movw $intervallo, %ax
15     outw %ax, %dx
16
```

Software II

```
17      movw $TIMER_STATUS, %dx # Avvia TIMER
18      movb $1, %al
19      outb %al, %dx
20      sti # Abilita la ricezione delle interruzioni
21      hlt
22
23  .driver 0 # Driver di TIMER
24      movw $TEMPERATURA_REG, %dx
25      inb %dx, %al
26      cmpb $40, %al # al > 40 se al - 40 > 0
27      js .minore
28      movw $VIDEO, %dx # Programma il DMAC di sistema
29      movq $messaggio, %rsi
30      movl $512/4, %ecx
31      cld
32      outsl # Copia 512 byte, una longword per volta, verso VIDEO
33
34    .minore:
```

Software III

```
35      # Cancella la causa di interruzione e riavvia TIMER
36      movw $TIMER_INT_REQ, %dx
37      movb $0, %al
38      outb %al, %dx
39      movb $1, %al
40      movw $TIMER_STATUS, %dx
41      outb %al, %dx
42      iret
```