Un processore z64 controlla un sistema per la digitalizzazione e l'archiviazione di documenti ad alta affidabilità. Il sistema è utilizzato contemporaneamente da più utenti che trascrivono mediante una tastiera alcuni documenti cartacei.
Questo sistema è quindi composto da un insieme di *n* periferiche `TASTIERA`, attraverso le quali gli utenti possono trascrivere i documenti, e da un sistema di archiviazione con replica di backup. Gli utenti hanno anche a disposizione una periferica `SCHERMO` ciascuno, per controllare la correttezza del documento che stanno digitalizzando.
Alla pressione di un tasto, il carattere corrispondente viene memorizzato all'interno di un buffer interno a ciascuna periferica `TASTIERA` (fa eccezione il tasto *backspace* che cancella l'ultimo carattere inserito). La periferica ha anche a disposizione un contatore, interrogabile dal processore, che permette di conoscere quanti caratteri sono stati digitati dall'ultima acquisizione.
Lo z64 interroga le periferiche `TASTIERA` in maniera ciclica. Ogni volta che l'interrogazione comunica allo z64 che almeno un carattere è disponibile nel buffer interno, lo z64 attiva una specifica procedura per acquisire in maniera asincrona, uno alla volta, tutti i caratteri presenti nel buffer.
Questi caratteri vengono memorizzati all'interno di un vettore in memoria, di dimensione fissa, ciascuno associato ad ogni utente del sistema. Ogni carattere viene anche trasmesso alla periferica `SCHERMO`, in maniera sincrona.
Quando un buffer di memoria viene riempito, lo z64 programma il DMAC di sistema per trasferire tutto il contenuto del buffer su una periferica `DISCO` attestata sul DMAC. Al termine di questo trasferimento, i nuovi caratteri digitati dall'utente verranno memorizzati su un nuovo buffer. Il buffer precedente, prima o poi, potrà essere utilizzato nuovamente. La scrittura dei buffer su `DISCO` è sequenziale.
Il sistema è anche equipaggiato di un `TIMER`. Allo scadere di un intervallo di tempo precedentemente programmato, lo z64, per garantire l'alta affidabilità, programma una periferica `NASTRO` che governa un nastro magnetico per effettuare un backup di uno dei buffer dati già memorizzati precedentemente su `DISCO`. La periferica lavora in maniera sincrona e, specificando la posizione sul nastro (una quadword) in cui scrivere un carattere ed il carattere stesso, effettua la scrittura. È compito del software garantire che i vari buffer non si sovrappongano. Qualora all'attivazione di `TIMER` sia presente almeno un buffer completamente pieno di cui effettuare il backup, il sistema riprogramma `TIMER` con un intervallo di attivazione dimezzato. Altrimenti, l'intervallo di attivazione viene raddoppiato. Questa procedura tiene conto di due soglie, MAX e MIN, che non possono mai essere superate.
Data la natura meno critica di quest'ultimo task del sistema, la scrittura su nastro deve poter essere interrotta dall'acquisizione di caratteri.
Si progettino interfacce, driver, e codice del sistema. Per semplicità, si può assumere che `DISCO` e `NASTRO` abbiano capacità illimitata, così come la dimensione dei buffer interni alle tastiere.

Interfacce

TASTIERA

- Periferica di input
- Registro di interfaccia per comunicare quanti tasti sono presenti nel buffer interno
- Questo numero può cambiare anche durante l'acquisizione dei caratteri (l'utente può continuare a scrivere)
- Registro di interfaccia per comunicare un carattere

SCHERMO

- Periferica di output sincrona
- Registro di interfaccia per scrivere un carattere

TIMER

- Identico al timer visto a lezione

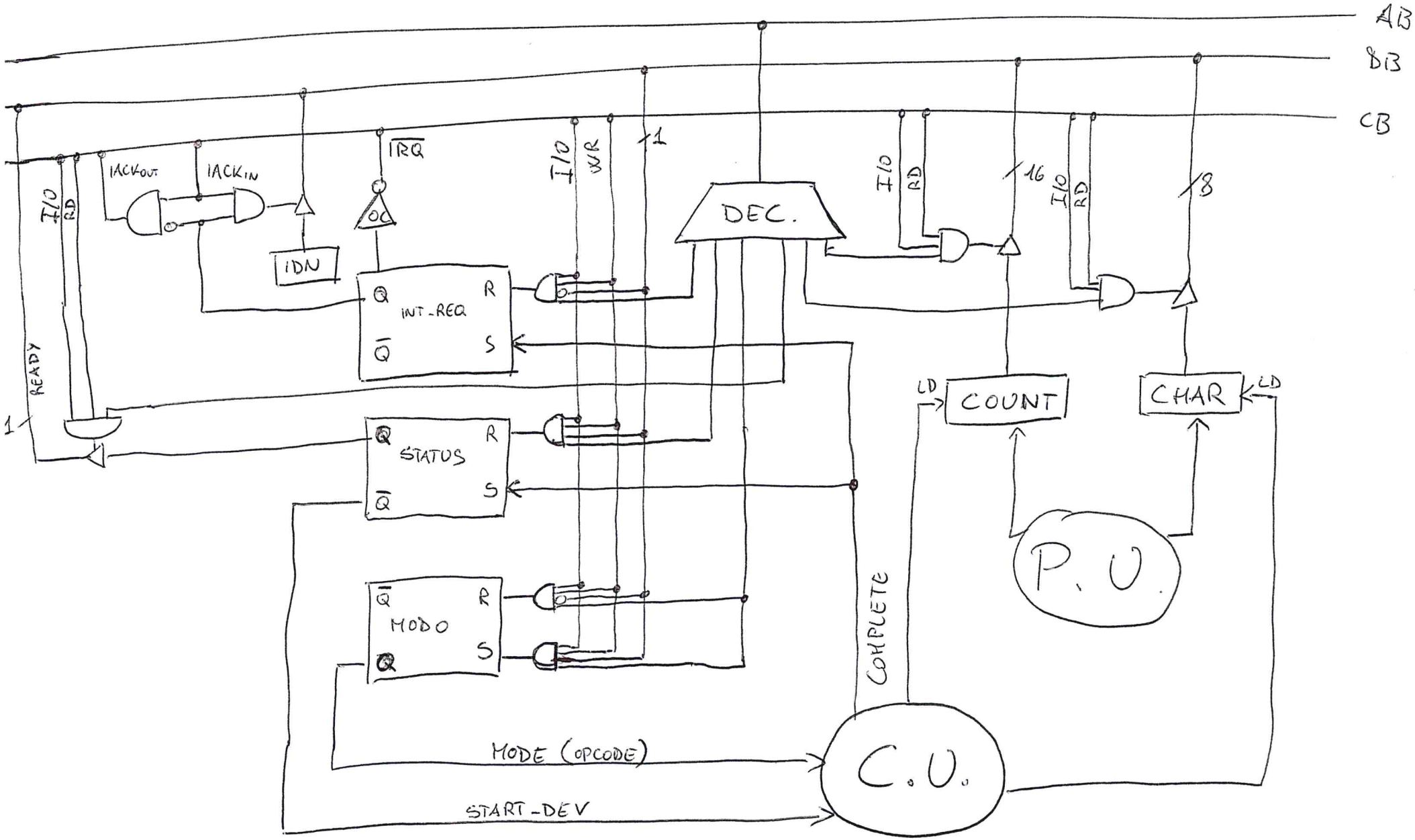
NASTRO

- Periferica di output asincrona
- Due registri: dato e posizione sul nastro

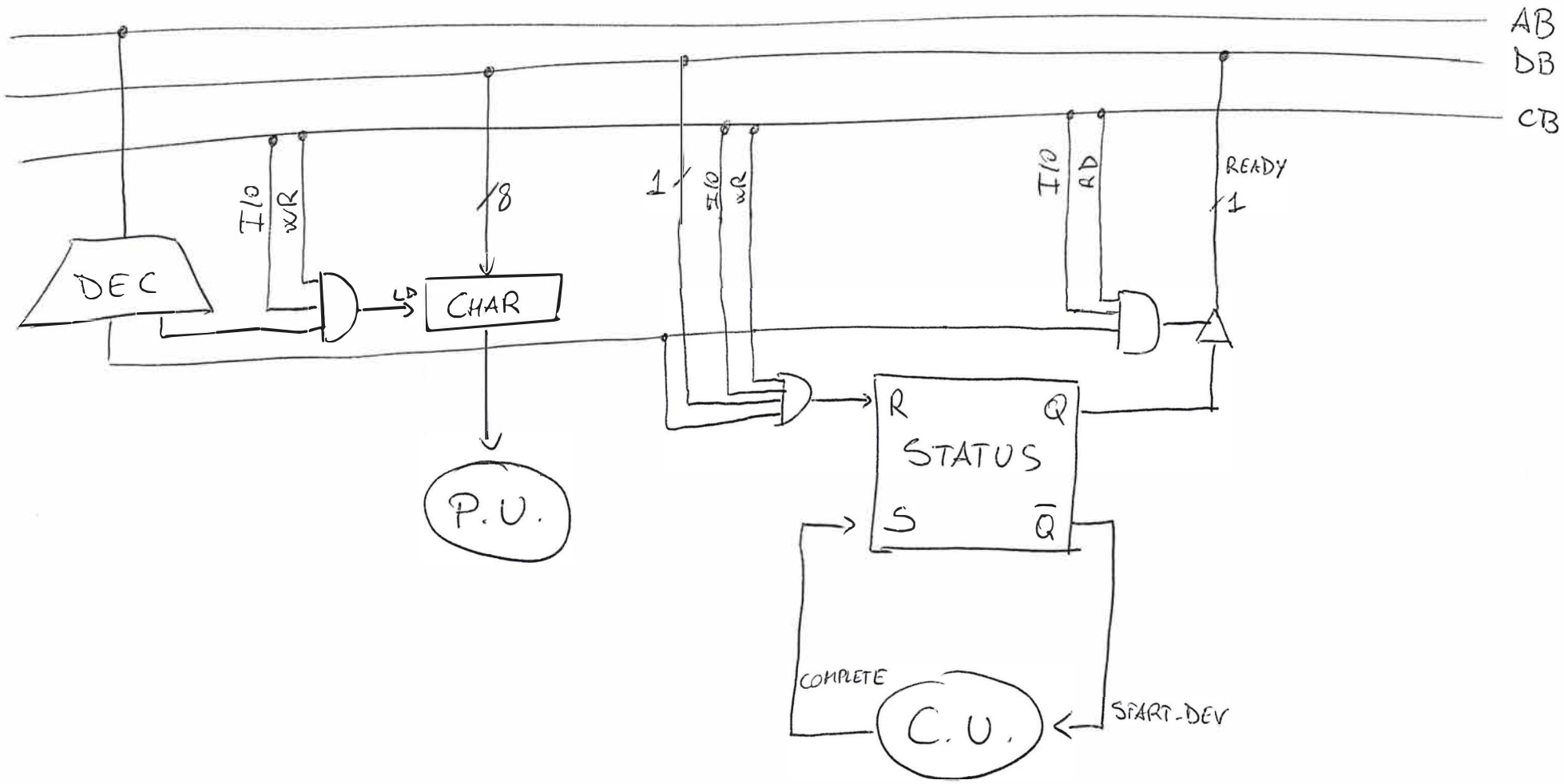
DISCO

- È attestato sul DMAC, pertanto non ci interessa progettarne l'interfaccia

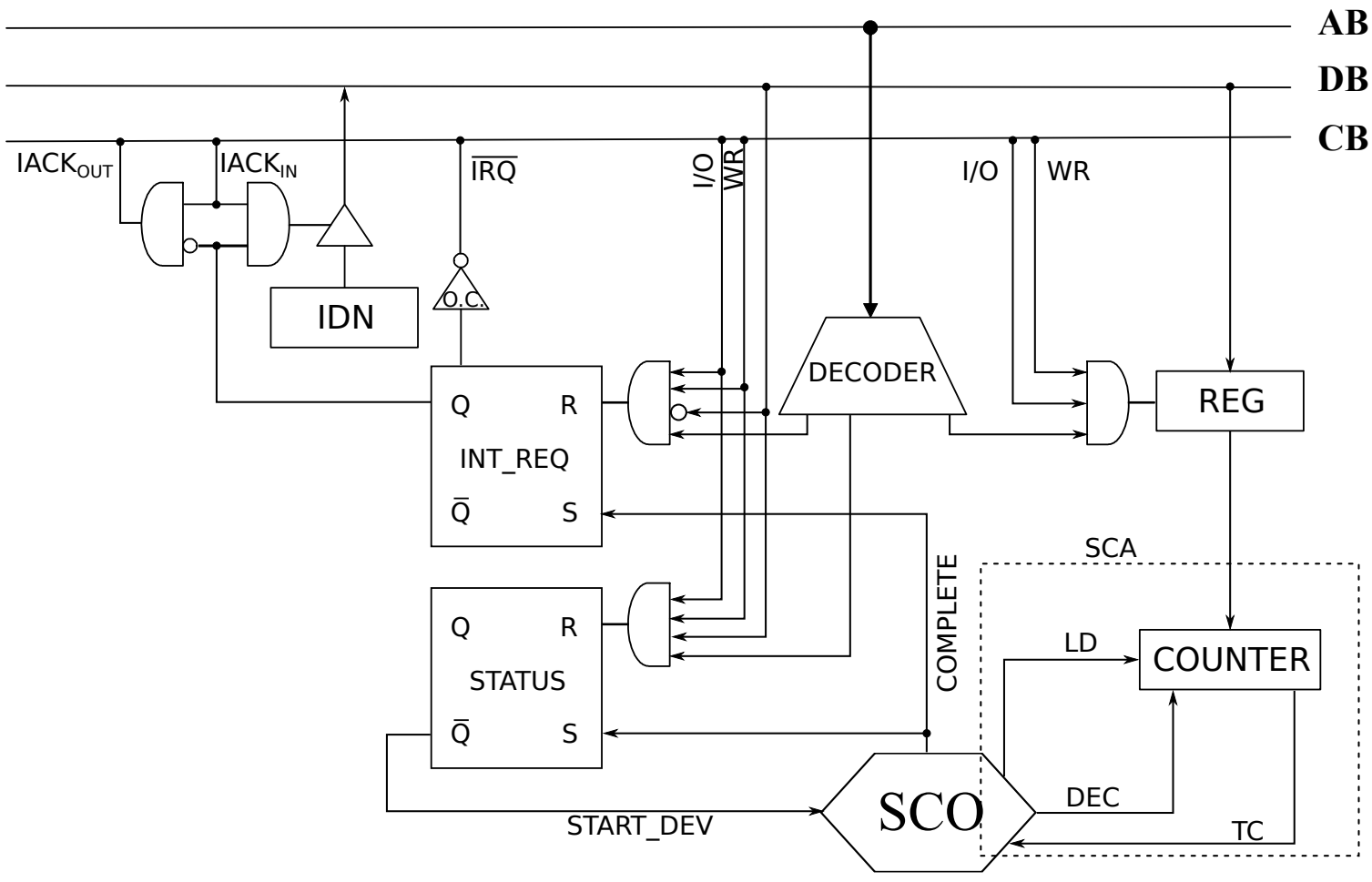
TASTIERA


AB
DB
CB
I/O
RD
IACKOUT
IACKIN
IDN
IRQ
OC
I/O
WR
1
DEC.
I/O
RD
16
I/O
RD
8
INT-REQ
R
Q
S
READY
1
STATUS
R
Q
S
LD
COUNT
CHAR
LD
P.U.
MODO
R
Q
S
COMPLETE
MODE (OPCODE)
START-DEV
C.U.

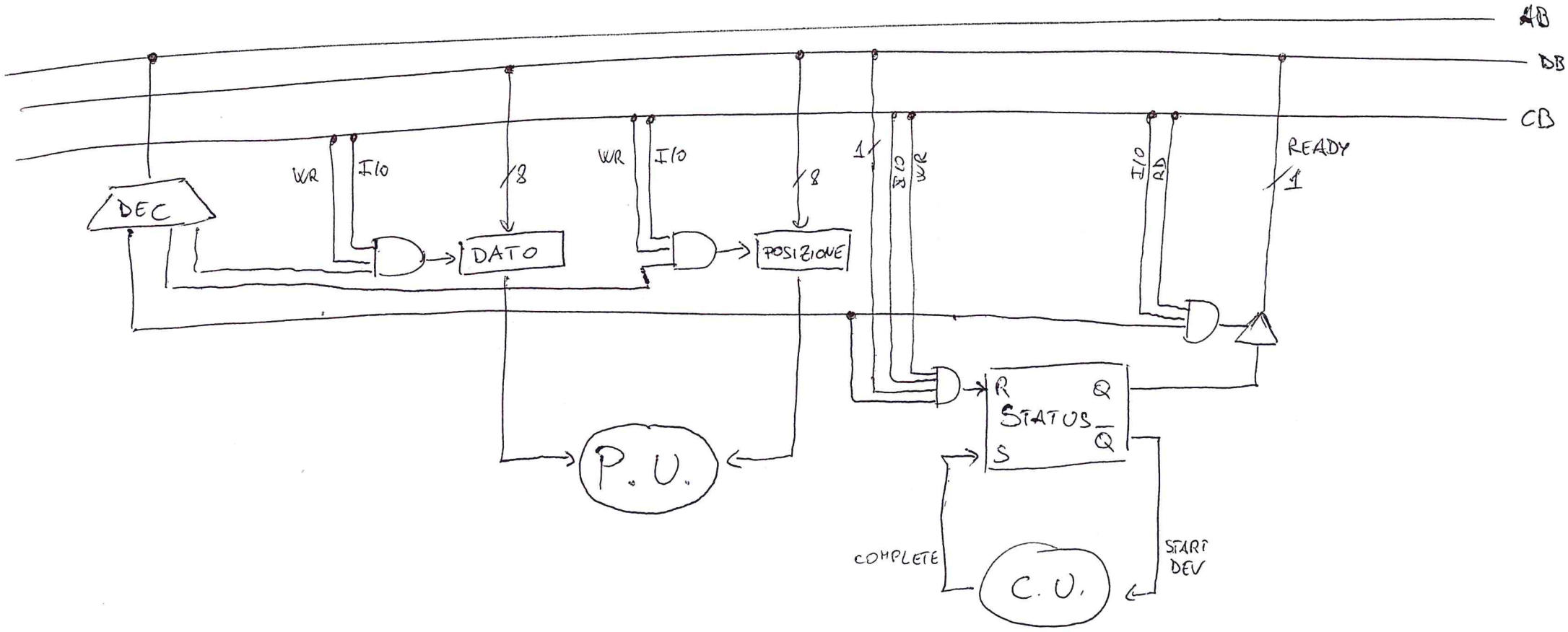
SCHERMO


AB
DB
CB
DEC
I/O
WR
8
CHAR
P.U.
1
I/O
WR
I/O
RD
READY
1
R
Q
STATUS
S
$\bar{Q}$
COMPLETE
C.U.
START-DEV

TIMER


AB
DB
CB
IACK$_{OUT}$
IACK$_{IN}$
$\overline{IRQ}$
I/O
WR
I/O
WR
IDN
O.C.
DECODER
REG
Q
R
INT_REQ
$\overline{Q}$
S
Q
R
STATUS
$\overline{Q}$
S
COMPLETE
SCA
LD
COUNTER
SCO
DEC
TC
START_DEV

NASTRO


AB
DB
CB
DEC
WR
IO
8
DATO
WR
IO
8
POSIZIONE
1
IO
WR
IO
RD
READY
1
R
Q
STATUS
S
Q
P. U.
COMPLETE
C. U.
START
DEV

Software

TASTIERA

- Un ciclo di polling sulle periferiche tastiera per scoprire se è presente almeno un carattere
- Bitmap per sapere per quale tastiera non è in corso un ciclo di acquisizione
- Driver a interruzioni per acquisire un carattere alla volta, finché non arriva a zero il registro associato al numero di caratteri presenti nel buffer
- Flip/flop di modo per discriminare l'operazione richiesta alla tastiera quando questa viene avviata a software
- Interfaccia funzionante sia a busy waiting che a interruzioni
- Assunzione: un backspace, se possibile, viene gestito nel buffer interno. Altrimenti, viene inviato il codice ASCII associato
- Un vettore "buffers" viene utilizzato per associare a ciascuna tastiera il buffer (di taglia fissa) in cui scaricare i caratteri
- Un vettore "filled" viene utilizzato per tenere traccia di quanti caratteri sono stati scaricati da ciascuna tastiera
- Al riempimento, un buffer libero viene recuperato dal vettore "buffer pool" che contiene un numero doppio di buffer. Una bitmap "buffer pool used" permette di sapere quali buffer sono attualmente liberi.

NASTRO

- Driver interrompibile
- Utilizza due variabili d'appoggio: una per sapere quale buffer sta scaricando su nastro, uno per sapere a quale carattere si è arrivati. Il valore -1 associato al buffer indica che ha completato tutte le proprie operazioni.

TIMER

- Driver interrompibile
- Se NASTRO non ha ancora completato la sua operazione, si riprogramma senza modificare la sua configurazione
- Controlla tutti i buffer della tastiera, ne identifica uno pieno ed avvia il funzionamento di NASTRO
- Se ha programmato NASTRO, dimezza il tempo di risveglio
- Se non ha programmato NASTRO, raddoppia il tempo di risveglio

```
.org 0x800

        .data

        .equ T_MODO_BW, 0
        .equ T_MODO_INT, 1

        .equ disco, 0x02

        .equ timer_status, 0x03
        .equ timer_irq, 0x04
        .equ timer_reg, 0x05

        .equ nastro_status, 0x06
        .equ nastro_dato, 0x08
        .equ nastro_posizione, 0x09

        # Si assume che le tastiere connesse al sistema siano soltanto 4.
        # Il codice di funzionamento del sistema è però indipendente dal numero.
        # Per aggiungere una nuova periferica tastiera è sufficiente popolare
        # un elemento in più dei vettori ed aumentare la taglia

        .equ TASTIERE, 4
        TAST_status:    .word   0x0a, 0x0f, 0x14, 0x19
        TAST_irq:       .word   0x0b, 0x10, 0x15, 0x1a
        TAST_modo:      .word   0x0c, 0x11, 0x16, 0x1b
        TAST_count:     .word   0x0d, 0x12, 0x17, 0x1c
        TAST_char:      .word   0x0e, 0x13, 0x18, 0x1d

        SCHERMO_status:.word    0x1e, 0x20, 0x22, 0x24
        SCHERMO_char:   .word   0x1f, 0x21, 0x23, 0x25

        # Maschera di bit per indicare se una tastiera è coinvolta o meno
        # in una operazione di trasferimento dati verso memoria
        .comm TAST_busy, (TASTIERE / 8 + 1)

        # Vettore di puntatori: ogni elemento punta al buffer associato alla
        # tastiera
        .comm TAST_buff, (TASTIERE * 8)

        # Vettore di interi: ogni elemento dice quanti caratteri sono stati
        # scaricati nel buffer corrispondente
        .comm TAST_counter, (TASTIERE * 4)

        .equ BUFF_SIZE,1024

        # Buffer per scaricare le sequenze di caratteri e relativa bitmap
        .comm buffers, (BUFF_SIZE * TASTIERE * 2)
        .comm buffer_bitmap, (TASTIERE * 2 / 8 + 1)
```

```
# Bitmap per indicare che il buffer associato deve essere processato
# da NASTRO
.comm buffer_bitmap_nastro, (TASTIERE * 2 / 8 + 1)

.equ NUM_BUFFERS, (TASTIERE * 2)




# Variabili utilizzate dal driver del TIMER
last_checked_buffer:    .long 0
.equ MAX, 16
.equ MIN, 2
current_interval:       .byte   MIN

.text
                # Programma il timer
                movb current_interval, %al
                movw $timer_reg, %dx
                outb %al, %dx

                movb $1, %al
                movw $timer_status, %dx
                outb %al, %dx

                # Ciclo di polling verso le tastiere. Qui si sfrutta il fatto che
                # i F/F di status sono associati agli indirizzi bassi
         .init:
                sti
                xorq %rsi, %rsi # %rsi identifica la tastiera da interrogare
                cli
        .loop:
                call verifica_tastiera
                addq $1, %rsi
                cmpq $TASTIERE, %rsi
                jz .init
                jmp .loop
```

```
# Questa funzione verifica il valore dell'i-simo bit di una bitmap.
# In %rsi si passa la posizione, in %rdi si passa la bitmap. Viene
# restituito "true" se il bit è impostato
check_bit_at:
                # Calcola il blocco in cui compare il bit associato, effettuando
                # una divisione per 8 (bitmap di byte)
                movq %rsi, %rdx
                shrq $3, %rdx

                # Calcola il bit da testare. Il bit è il resto della divisione
                # precedente. Poiché 8 è potenza di 2, vale la relazione:
                # a % b == a & (b - 1)
                # Costruisce anche la maschera da usare per fare il test
                movq %rsi, %rcx
                andq $7, %rcx
                movb $1, %r8b
                shlb %r8b

                # Carica il blocco di interesse ed effettua il mascheramento
                movb (%rdi, %rdx, 1), %al
                andb %r8b, %al

                ret


# Questa funzione imposta l'i-simo bit di una bitmap.
# In %rsi si passa la posizione, in %rdi si passa la bitmap.
set_bit_at:
                # Calcola il blocco in cui compare il bit associato, effettuando
                # una divisione per 8 (bitmap di byte)
                movq %rsi, %rdx
                shrq $3, %rdx

                # Calcola il bit da testare. Il bit è il resto della divisione
                # precedente. Poiché 8 è potenza di 2, vale la relazione:
                # a % b == a & (b - 1)
                # Costruisce anche la maschera da usare per fare il test
                movq %rsi, %rcx
                andq $7, %rcx
                movb $1, %r8b
                shlb %r8b

                # Imposta il bit nel blocco di interesse
                orb %r8b, (%rdi, %rdx, 1)

                ret
```

```
# Questa funzione cancella l'i-simo bit di una bitmap.
# In %rsi si passa la posizione, in %rdi si passa la bitmap.
clear_bit_at:
                # Calcola il blocco in cui compare il bit associato, effettuando
                # una divisione per 8 (bitmap di byte)
                movq %rsi, %rdx
                shrq $3, %rdx

                # Calcola il bit da testare. Il bit è il resto della divisione
                # precedente. Poiché 8 è potenza di 2, vale la relazione:
                # a % b == a & (b - 1)
                # Costruisce anche la maschera da usare per fare il test
                movq %rsi, %rcx
                andq $7, %rcx
                movb $1, %r8b
                shlb %r8b
                negb %r8b

                # Carica il blocco di interesse ed imposta il bit
                andb %r8b, (%rdi, %rdx, 1)

                ret



# Questa funzione trova un buffer libero in buffers, controllando
# la bitmap in buffer_bitmap e convertendo l'indice in spiazzamento
# rispetto a buffers.
get_free_buffer:
                push %rsi
                movq $buffer_bitmap, %rdi
        .gfb_init:
                xorq %rsi, %rsi
        .gfb_loop:
                call check_bit_at
                cmpq $0, %rax
                jnz .gfb_skip

                # Marca il buffer come usato
                movq %rax, %rsi
                call set_bit_at

                # La funzione moltiplica è identica a quella vista a lezione
                movq $BUFF_SIZE, %rsi
                movq %rax, %rdi
                call moltiplica

                pop %rsi
                ret
```

```
        .gfb_skip:
                addq $1, %rsi
                cmpq $NUM_BUFFERS, %rsi
                jz .gfb_init
                jmp .gfb_loop


# Chiedi alla tastiera quanti caratteri sono disponibili.
# In %rsi il codice della tastiera. Restituisce il numero di caratteri
caratteri_disponibili:
                # - Imposta il modo a MODO_BW
                # - Avvia la periferica
                # - Aspetta che sia pronta
                # - Leggi il registro COUNT
                # - Se è zero, ritorna al chiamante

                movb $MODO_BW, %al
                movw TAST_modo(, %rsi, 2), %dx
                outb %al, %dx
                movb $1, %al
                movw TAST_status(, %rsi, 2), %dx
                outb %al, %dx
        .bw_vt:
                inb %dx, %al
                btb $0, %al
                jnc .bw_vt

                movw TAST_count(, %rsi, 2), %dx
                inw %dx, %ax
                ret


# Se la tastiera identificata dal codice passato in %rsi
# non è soggetta ad operazione di acquisizione dati, verifica se
# almeno un carattere è stato immesso. In caso positivo, avvia la
# procedura di acquisizione caratteri.
verifica_tastiera:

                # Verifica se la tastiera è occupata per un'operazione di copia.
                # Se la funzione restituisce "true", ritorna al chiamante.
                movq $TAST_busy, %rdi
                call check_bit_at
                testq %rax, %rax
                jnz .out

                call caratteri_disponibili
                cmpw $0, %ax
                jz .out
```

```
                # Avvia il processo di acquisizione:
                # - Trova un buffer libero
                # - Assegnalo alla tastiera
                # - Azzera il contatore di caratteri associato
                # - Programma l'acquisizione dei caratteri
                # - Marca la tastiera come "occupata"

                call get_free_buffer
                movq %rax, TAST_buff(, %rsi, 2)
                movq $0, TAST_count(, %rsi, 2)

                movb $T_MODO_IRQ, %al
                movw TAST_modo(, %rsi, 2), %dx
                outb %al, %dx

                movb $1, %al
                movw TAST_status(, %rsi, 2), %dx
                outb %al, %dx

                movq $TAST_busy, %rdi
                call set_bit_at
        .out:
                ret

# Recupera un carattere da una tastiera e lo salva nel buffer corrispondente.
# Inoltre, lo scrive sull'i-simo schermo in modalità sincrona.
# Restituisce il numero di caratteri nel buffer corrispondente.
recupera_carattere:
                movw TAST_char(, %rsi, 2), %dx
                inb %dx, %al
                # Controlla se è un backspace
                cmpb $0x8, %al
                jnz .rc_store

                subl $1, TAST_counter(, %rsi, 2) # TODO: non si controlla se è
                                                 # il primo carattere
                jmp .rc_out
        .rc_store:
                movq TAST_buff(, %rsi, 2), %rdi
                movq TAST_counter(, %rsi, 2), %rcx
                movb %al, (%rdi, %rcx, 1)
                addl $1, TAST_counter(, %rsi, 2)

                movw SCHERMO_char(, %rsi, 2), %dx
                outb %al, %dx

                movw SCHERMO_status(, %rsi, 2), %dx
                movb $1, %al
                outb %al, %dx
```

```
        .rc_bw:
                inb %dx, %al
                btb $0, %al
                jnc .rc_bw
        .rc_out:
                movl TAST_counter(, %rsi, 2), %eax
                ret


# Effettua la copia su disco.
programma_backup:
                push %rsi

                # Utilizza il DMAC di sistema
                movq TAST_buff(, %rsi, 2), %rsi
                movq $BUFF_SIZE, %rcx
                movw $disco, %dx
                cld
                outsb

                pop %rsi

                # Indica che il buffer appena scaricato su disco deve anche
                # essere processato da NASTRO
                movq $buffer_bitmap_nastro, %rdi
                call set_bit_at

                # Indica la tastiera in questione come "non occupata"
                movq $TAST_busy, %rdi
                call clear_bit_at

                ret
```

```
# Codice comune dei driver
common_driver_code:
                pushq %rsi
                pushq %rdi
                pushq %rdx
                pushq %rax
                call recupera_carattere
                cmpl $BUFF_SIZE, %eax
                jnz .riprogramma_0

                call programma_backup
                jmp .leave_0
        .riprogramma_0:
                call caratteri_disponibili
                testb %al, %al
                jz .leave_1

                movb $1, %al
                movw TAST_status(, %rsi, 2), %dx
                outb %al, %dx

        .leave_0:
                # Cancella la causa di interruzione prima di ritornare
                movb $0, %al
                movw TAST_irq(, %rsi, 2), %dx
                outb %al, %dx

                pushq %rax
                pushq %rdx
                pushq %rdi
                pushq %rsi

                iret

        .leave_1:
                # Indica la tastiera in questione come "non occupata"
                movq $TAST_busy, %rdi
                call clear_bit_at
                jmp .leave_0
```

```
# Driver tastiere: un carattere è stato messo nel registro corrispondente.
# Recupera il carattere e verifica se ce ne sono altri
.driver 0
                movq $0, %rsi
                jmp common_driver_code

.driver 1
                movq $1, %rsi
                jmp common_driver_code

.driver 2
                movq $2, %rsi
                jmp common_driver_code

.driver 3
                movq $3, %rsi
                jmp common_driver_code

# Questa funzione scrive su NASTRO un carattere.
# In %sil il carattere, in %rdi l'indice del carattere nel buffer
scrivi_su_nastro:
                movb %sil, %al
                movw $nastro_dato, %dx
                outb %al, %dx

                # La cella del nastro su cui scrivere è:
                # BUFF_SIZE * numero_copie + posizione carattere
                push %rdi
                movq $BUFF_SIZE, %rdi
                movq nastro_copie, %rsi
                call moltiplica
                pop %rdi
                addq %rdi, %rax

                # moltiplica ci dà il risultato in %rax che va nel registro POSIZIONE
                movw $nastro_posizione, %dx
                outw %ax, %dx

                # Avvia la scrittura
                movb $1, %al
                movw $nastro_status, %dx
                outb %al, %dx

        .ssn_bw:
                inb %dx, %al
                btb $0, %al
                jnc .ssn_bw

                ret
```

```
# Driver di timer: controlla se un qualche buffer va scaricato.
# Riparte sempre dall'ultimo buffer controllato per evitare starvation.
# Driver interrompibile
.driver 4
                movb $0, %al
                movw $timer_irq, %dx
                outb %al, %dx
                sti

                cmpq $-1, nastro_buff
                jnz .timer_out

                # Cerca un buffer che NASTRO deve ancora copiare
                movl last_checked_buffer, %r9d

        .timer_loop:
                movzlq %r9d, %rsi
                movq $buffer_bitmap_nastro, %rdi
                call check_bit_at
                cmpq $1, %rax
                jz .found

                addl $1, %r9d
                cmpl last_checked_buffer, %r9d
                jnz .timer_loop

                # Nessun buffer trovato: raddoppio il tempo di attivazione.
                cmpb $MAX, current_interval
                jz .timer_out
                shlb $1, current_interval

                movb current_interval, %al
                movw $timer_reg, %dx
                outb %al, %dx

                jmp .timer_out

        .found:
                # Determina l'offset del buffer da cui copiare
                movq %rax, nastro_buff
                movq %rax, %rdi
                movq $BUFF_SIZE, %rsi
                call moltiplica

                # Effettua la copia del buffer in busy waiting
                xorq %rcx, %rcx
        .copy_loop:
                movb buffers(%rax, %rcx, 1), %sil
```

```
        movq $0, %rdi
        call scrivi_su_nastro
        addl $1, %rcx
        cmpl $BUFF_SIZE, %rcx
        jnz .copy_loop

        # Dimezza il tempo di attivazione di timer
        cmpb $MIN, current_interval
        jz .timer_out
        shrb $1, current_interval

        movb current_interval, %al
        movw $timer_reg, %dx
        outb %al, %dx

.timer_out:
        cli
        movb $1, %al
        movw $timer_status, %dx
        outb %al, %dx
        iret
```