PROGETTO

Quattro periferiche P1,P2,P3,P4 producono dati di dimensione word *come input* per il processore z64.

Scrivere il codice di una subroutine leggi che accetti come parametri (secondo le calling convention System V ABI) il numero di dati (byte) da leggere dalle periferiche e l'indirizzo di memoria da cui il processore z64 dovrà incominciare a scrivere i dati così acquisiti. Scrivere inoltre il programma che invochi la funzione leggi chiedendo di acquisire 100 word dalle periferiche e di memorizzarli in un vettore posto a partire dall'indirizzo 0x1200.

ATTENZIONE: i 100 dati possono essere letti non necessariamente rispettando l'ordine delle periferiche (ad esempio 10 da P1, 23 da P2, ecc.)

S v o l g i m e n t o

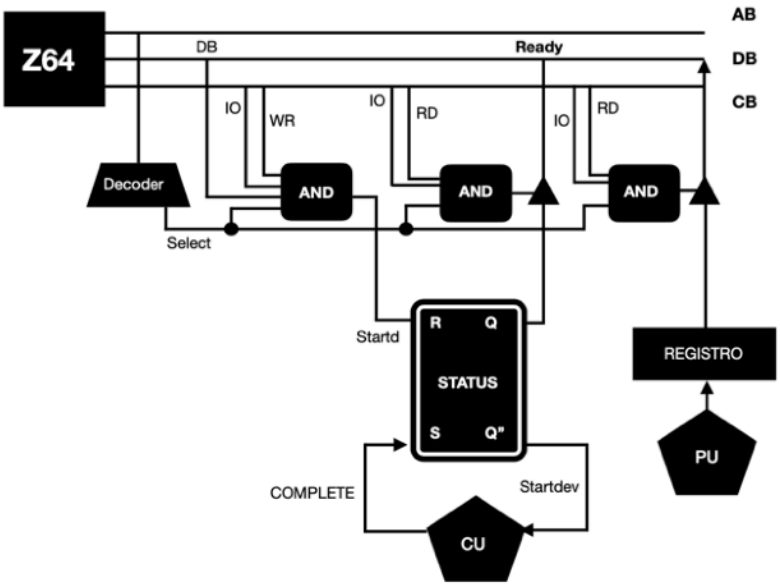
Per svolgere questo codice innanzitutto è opportuno capire il modo con cui i dispositivi interagiscono con la CPU. Siamo in presenza di un'interazione basata su "polling": è analogo al Busy waiting (attesa indefinita) ma a differenza di quest'ultimo il processore non aspetta più che il singolo device finisca di processare/produrre i dati.
Le varie interfacce dei vari dispositivi visti come catena circolare di polling sono di input: ciascun dispositivo produce dati di dimensione word come input per il processore.
Dalla periferica bisogna leggere un byte.

Ciascuna periferica sarà costruita nel seguente modo ——>
Essa è un'interfaccia di input con funzionamento Busy waiting.

0x0000: è importante capire che quello è un indirizzo di un elemento di memoria in un'interfaccia di dispositivo, *non è* un indirizzo di memoria.

Fatte queste premesse si riporta di seguito il codice completo.

La soluzione proposta non ha l'I/O Port parametrico.



```
.org 0x800
.data
        .equ $STATUS_DEV1, 0x0000
        .equ $STATUS_DEV2, 0x0001
        .equ $STATUS_DEV3, 0x0002
        .equ $STATUS_DEV4, 0x0003
        .equ $REG_STATUS_1, 0x0004
        .equ $REG_STATUS_2, 0x0005
        .equ $REG_STATUS_3, 0x0006
        .equ $REG_STATUS_4, 0x0007
        . = 0x1200
        vettore: .fill 100,2
.text
main:
        movq $100, %rdi
        movq $vettore, %rsi
        call leggi
        hlt
 leggi:
        push %rbx
        xorq %rbx, %rbx
   .poll:
        outb %al, $STATUS_DEV1 #chiedo al device 1 di produrre un nuovo dato
        inb $STATUS_DEV1, %al #leggo il valore del flip flop status (attendo che il dato venga prodotto)
        btb $0, %al #confronto il valore del flip flop status con 0 (BitTest)
        jc .dev1 #se il valore del flip flop è uguale ad 1 significa che è pronto il dato che la CPU deve consumare
        outb %al, $STATUS_DEV2
        inb $STATUS_DEV2, %al
        btb $0, %al
        jc .dev2
        outb %al, $STATUS_DEV3
        inb $STATUS_DEV3, %al
        btb $0, %al
        jc .dev3
        outb %al, $STATUS_DEV4
        inb $STATUS_DEV4, %al
        btb $0, %al
        jc .dev4
        jmp .poll
   .fine:
        pop %rbx # lo spazio del vettore è esaurito; Non rimane che distruggere lo stack frame
        hlt

   .dev1:
        inw $REG_STATUS_1, %ax #prelevo una word dal dispositivo
        movw %ax, (%rsi, %rbx, 2) #la inserisco nel vettore dove ogni cella sono due byte
        addq $1, %rbx #passo alla prossima cella
        cmpq %rbx, %rdi #confronto se abbiamo esaurito lo spazio del vettore
        #outb %al, $STATUS_DEV_1 #riavvio il dispositivo (non viene messo perché il riavvio viene effettuato nella
etichetta .poll
        jnz .poll #se lo spazio non è esaurito continuano ad acquisire
        jmp .fine #se lo spazio del vettore non è esaurito saltiamo ad una label
   .dev2:
        inw $REG_STATUS_2, %ax
        movw %ax, (%rsi, %rbx, 2)
        addq $1, %rbx
        cmpq %rbx, %rdi
        jnz .poll
        jmp .fine
   .dev3:
        inw $REG_STATUS_3, %ax
        movw %ax, (%rsi, %RBX, 2)
        addq $1, %rbx
        cmpq %rbx, %rdi
        jnz .poll
        jmp .fine
   .dev4:
        inw $REG_STATUS_4, %ax
        movw %ax, (%rsi, %RNX, 2)
        addq $1, %rbx
        cmpq %rbx, %rdi
        jnz .poll
        jmp .fine
```