

⚰

Assembly

# Porzioni di blocchi usati dal prof

## Modalità di passaggio di variabili

💼 esempio

section text

Stringa1 db 'Ottimo lavoro', 0;

### Passaggio per indirizzo (by reference)

mov ax, Stringa1 ; Passo ad ax l'offset di Stringa1, cioè Stringa10

Questo ti permette di scorrere la stringa, perché hai lʼindirizzo di partenza, fondamentale quando si

usa lodsb ,

stosb

### Passaggio per valore (by value)

mov al, Stringa1 ; al = 'O'

mov ah, Strigna1 + 3 ; al = 'i'

⚠ Va bene solo se vuoi prendere il primo carattere della stringa (es: 'O' )

✅ Utile quando devi leggere un solo carattere

## Stringhe

### Stringhe Pascal vs Stringhe alla C

SPascal: db 4,'Ciao' ; 4 è la lunghezza della stringa SC: db 'Ciao', 0 ; 0 è il terminatore stringa

**lodsb**

Legge da

**/**

→ carica in

**lodsw**



SI

. Dopo lʼoperazione, SI viene aggiornato automaticamente

(incrementato o decrementato a seconda della direzione)

AL/AX

Stampa:

lodsb ; Carico S1{i} in AL cmp al, 0

je fineStampa

int 10h ; interrupt di Stampa jmp Stampa

**stosb**

Scrive



DI

**/**

in

**stosw**

AL/AX

. Dopo lʼoperazione, DI viene aggiornato

sub al, ah

. ; altre operazioni

; Carica AL (Vett{i}/'-'/'+') in Dest{i}

stosb ; compreso tra 'A' e 'Z' quindi memorizzo in Dest inc bx ; incremento bx, così al prossimo giro punto ad i++ jmp Ciclo

⚠ Il comportamento di



SI

e

(se si incrementano o decrementano) dipende dal flag di direzione



DF :

(Clear Direction Flag) \* SI e DI aumentano (Set Direction Flag) \* SI e DI diminuiscono

SECTION data

src db 'Copia questa stringa', 0

dest db 50 dup('$') ; destinazione inizializzata

SECTION text

cld ; SI e DI aumentano dopo ogni istruzione mov si, src ; punta allʼinizio della sorgente

mov di, dest ; punta alla destinazione

.loop:

lodsb

cmp al, 0 je .fine

stosb

jmp .loop

; AL \* SI, poi SI++

; fine stringa?

; DI \* AL, poi DI++



DI



CLD



STD

💼 Esempio pratico: copia di una stinga

.fine:

### Calcolare la lunghezza di una stringa scritta in C

SECTION data

S1: db 'God bless the Queen',0 SECTION text

mov ax, S1 push ax

call LunghezzaStringa

;....

LunghezzaStringa: push bp

mov si, [bp+4 ; si  S1

; Uso solo il registro ax: ah=contatore, al=lettura carattere xor ah, ah ; pulisco il contatore

CalcoloLunghezza: lodsb ; al ← si{i} cmp al, '0'

je FineCalcoloLunghezza inc ah ; ah++

jmp CalcoloLunghezza

FineCalcoloLunghezza: pop bp

ret

**Inserire una stringa al contrario**

Ho due modi per farlo:



→ modifico DF (Direction Flag)

→ tramite accesso manuale

; MODIFICO LE DIRECTION FLAG

; seconda metà di S1 al contrario mov si, [bp+8 ; Ripristino si=S1

mov cl, dl ; cl  LunghezzaStringa(S1)

add si, cx ; SI \* fine della stringa (subito DOPO lʼultimo carattere) dec si ; SI \* punta prima del terminatore di S1

shr cl, 1 ; cl  LunghezzaStringa(S1) / 2

jnc Ciclo3 ; Se lunghezza pari (no CF), salta lʼinc

inc cl ; (si CF), aumento di 1 per copiare un carattere in più

; Carico in S3 la 2° metà di S1 al contrario Ciclo3:

std ; Setta DF1 \* direzione inversa per le string instructions

lodsb ; AL \* SI, SI-- (grazie a DF1)

cld ; Resetto DF a 0 per le istruzioni normali (sicurezza) stosb ; DI \* AL, DI++

loop Ciclo3 ; CL--, se  0 salta a Ciclo3

; TRAMITE ACCESSO MANUALE

; prima metà di S2 al contrario

mov bx, [bp+6 ; Ripristino bx  S2

mov cl, dh ; Ripristino cl  LunghezzaStringa(S2) shr cl, 1 ; cl  LunghezzaStringa(S2) / 2

add bx, cx ; BX \* fine della prima metà (inizio seconda metà) dec bx ; ora BX punta allʼultimo carattere della prima metà

; Carico in S3 la 1° metà di S2 al contrario Ciclo4:

mov al, [bx] ; AL \* carattere corrente stosb ; DI \* AL, DI++

dec bx ; BX--

loop Ciclo4 ; CL--, salta a Ciclo4 se  0

**Stampare una stringa**

StampaStringa: push bp

mov bp, sp

mov si, [bp+4 ; si  S3 mov ah, 0eh

mov bx, 0

Stampa: lodsb

cmp al, 0

je FineStringa int 10h

jmp Stampa

FineStringa: pop bp ret

## Numeri

### Sommare un numero i-esimo di in un array

Equivale a in C.

al + Vett[bx]

Vett: db 3, 5, 9, 12, 23, 21, 5, 6, 3, -1 ; in section text

Indici: db 2, 0, 10, 3, 1, 22, 1, 7, 8, 4

....

xor bh, bh

mov si, Indici

mov bl, [si] ; al = valore dell'indice corrente add al, Vett + bx]

### Leggere da input numeri a due o più cifre

💡 In Assembly DOS, l'input da tastiera è sempre ASCII, quindi per ottenere il numero vero serve sempre **sub 30h** .

cmp al,'0'

jb Ripeti ; al < '0' torno a Ripeti e chiedo un'altra cmp al, '9'

ja Ripeti ; al > '9' torno a Ripeti e chiedo un'altra mov bl, al

sub bl, 30h ; converto il codice ascii del numero mov ax, Numero]

mov dx, 10

; se Numero è DD \* risultato in DX:AX

; se Numero è DW \* risultato in AX mul dx

add ax, bx ; vecchio numero per 10 + nuovo numero (in BX) mov Numero], ax

**Stampare a monitor 1/2/3 cifre ( 8 bit )**

mov al, dl ; al = valore output ( 1/2/3 cifre, 8bit ) xor ah, ah

mov bl, 100

div bl ; AL  RIS DIV; AH  RESTO

mov ch, ah cmp al, 0

je duecifre ; se al  0 \* ho solo due cifre

; altrimenti stampo le centinaia add al, 30h

mov ah, 0eh mov bx, 0

int 10h

; scrivo le decine anche se è uno 0, ho già scritto le centinaia decine:

mov al, ch xor ah, ah mov bl, 10

div bl ; al = quoziente, ah = resto mov ch, ah

; Stampo le decine add al, 30h

mov ah, 0eh mov bx, 0

int 10h

jmp unita duecifre:

mov al, ch xor ah, ah mov bl, 10 div bl

mov ch, ah cmp al, 0

je unita ; se al  0 \* ho solo una cifra

; altrimenti stmapo le decine add al, 30h

mov ah, 0eh mov bx, 0

int 10h unita:

mov al, ch add al, 30h

mov ah, 0eh mov bx, 0

int 10h

## Stack

⚠ Stack deve restare allineato su 2 byte (8086 non gestisce bene sempre caricare un registro da 16 bit

push

### 📦 Chiamata ad una funzione

dispari), quindi devo

xor ah, ah mov al, N

push ax ; ax  00XXh dove XXh \* N mov ax, Vett

push ax

call ModificaStringa

### 🔁 Gestione di più funzioni: Stack Frames separati

Ogni crea una nuova "finestra" sullo stack, con:



call

il proprio il proprio



bp

di ritorno



ip

 i propri parametri a partire da

[bp+4

### 🧹 Pulizia Stack post call

call ModificaStringa ; ....

; Al ritorno lo stack ha 6B in memoria add sp, 6 ; Pulisco lo stack

🗃 **FILE**

### Lettura da un file binario

SECTION .data

filename1 db 'dat1', 0 ; nome file terminato da 0 handle1 dw 0 ; dove salveremo lʼhandle

Vett1 resw 100 ; spazio per 100 word  200 byte N db 0 ; numero di elementi letti (word)

SECTION .text

;  APRI FILE "dat1" ===

mov ah, 3Dh ; funzione open file mov al, 0 ; modalità read-only

mov dx, filename1 ; DS:DX  nome file int 21h

jc errore ; se CF1 \* errore

mov [handle1, ax ; AX  file handle

;  LEGGI FILE IN Vett1 

mov ah, 3Fh ; funzione read mov bx, [handle1 ; handle

mov cx, 200 ; byte da leggere (es: 100 word  200 byte) mov dx, Vett1 ; destinazione

int 21h jc errore

; AX  numero di byte letti \* AX/2  numero di word mov bl, 2

xor ah, ah

div bl ; AL  AX/2 mov N, al ; salvo N

;  CHIUDI FILE  mov ah, 3Eh

mov bx, [handle1 int 21h

jc errore

jmp continua errore:

; gestisci lʼerrore come vuoi

jmp $

continua:

; qui puoi continuare con il resto del programma

# Template logico

## Segmenti

SECTION data

; Dichiarazione delle variabili statiche (es. stringhe,

; vettori, costanti)

SECTION text

; Codice eseguibile

1. **Main program - Schema Standard**

Uso mov es, ax quando:

→ uso stosb , lodsb

→ la stinga di destinazione ( Dest ) è gestita tramite



di

SECTION text

..start:

; Carico i dati mov ax, data mov ds, ax

; se accedo a dati esterni o lavoro su `dest`, `extra`, ecc. mov es, ax

; Inizializzazione registri e contatori (xor cl, cl / xor ch, ch)

; Eventuale ciclo di input: CicloInput:

; Interrupt di lettura schermo mov ah, 00h

int 16h

cmp al, 1Bh ; Gestione ESC je Fine

; Operazioni

; push dei parametri

; call funzione

; add sp, X (pulizia) con X  numero valori nello stack

jmp CicloInput

Fine:

; Eventuale zero-terminazione

; Eventuale stampa stringa mov ax, 4c00h

int 21h

## Operazioni ricorrenti

🧪 **Lettura tastiera**

mov ah, 00h int 16h

cmp al, 1bh ; Carattere ESC je Fine

🎯 **Controllo carattere**

;posso Confrontare cmp al, 'S' ; char

cmp al, 50h ; hex cmp al, 80 ; dec je EtichettaS

cmp al, 'N' je EtichettaN

; Altrimenti salto (input non valido) jmp Ciclo

🔄 **Controllo pari/dispari (per input [ - ] )**

sub al, 30h ; da ASCII a cifra (es. '3' \* 3) test al, 00000001b

je Pari

; altrimenti dispari

✏ **Aggiunta caratteri a stringa (modificata)**

add dl, cl ; input{i} += pos

add si, ax ; Sposto di {i} posizioni in avanti nella Stringa mov [si], dl ; carico il valore nuovo del char nella Stringa

📦 **Passaggio parametri via stack**

**ʼ0ʼ**

**ʼ9ʼ**

|  |  |
| --- | --- |
| Offset | Contenuto |
| [bp+2 | IP (ritorno) |

⚠ Usare parametri tramite

mov bp, sp

per accedere ai

[bp+X

|  |  |
| --- | --- |
| Offset | Contenuto |
| [bp+4 | parametro 1 |
| [bp+6 | parametro 2 |
| [bp+8 | parametro 3 |

### 🔢 Convertire un numero binario in decimale per visualizzarlo



Se devi stampare un numero letto da file, ad esempio 45 , devi trasformarlo in caratteri ASCII: e



'4'

'5' .

Strategia utilizzata: Divisione ripetuta per 10. Usi per ottenere cifra a cifra.



DIV

; Input: AL contiene il numero da stampare (es: 45)

; Output: stampa il numero su schermo

ConvertiEStampa: mov ah, 0

mov cx, 0 ; Conta cifre

push ax ; Salva il numero

; --- Divisione per 10 fino a zero --- LoopDiv:

pop ax

mov ah, 0

mov bl, 10

div bl ; AL / 10 \* AL=quoziente, AH=resto add ah, '0' ; Converte cifra in ASCII

push ax ; Salva quoziente per dopo push ah ; Salva cifra

inc cx

cmp al, 0 jne LoopDiv

; --- Stampa cifre dallo stack --- Stampa:

pop ax

mov dl, al

mov ah, 4c00h int 21h

loop Stampa ret

⚠ Se il numero è in un registro di 16 bit (💼 AX ), va adattato dividendo



AX

### 📁 Leggere due vettori da un file binario

Ho bisogno di questi interrupt:

 INT 21h - 3Dh : apre un file

 INT 21h - 3Fh : legge da file

e gestendo a due Byte

 INT 21h - 3Eh : chiude il file SECTION data

FileName: db 'vettori.bin', 0 ; Nome del file binario (ASCIIZ) Handle: dw ? ; File handle

Buffer: resb 512 ; Buffer per leggere i dati

Vett1: resb 100 ; Primo vettore

Vett2: resb 100 ; Secondo vettore

SECTION text

..start:

mov ax, data mov ds, ax

; --- Apri il file --- mov dx, FileName

mov al, 0 ; 0  read-only mov ah, 3Dh

int 21h

jc Errore ; Se c'è errore, salta

mov Handle], ax ; Salva handle in memoria

; --- Leggi 200 byte nel buffer --- mov bx, Handle] ; File handle

mov cx, 200 ; Byte da leggere mov dx, Buffer ; Dove leggere mov ah, 3Fh

int 21h jc Errore

; --- Copia in Vett1 e Vett2 --- mov si, Buffer

mov di, Vett1 mov cx, 100

rep movsb ; Ripeto l'operazione finché CX ! 0

mov di, Vett2 mov cx, 100 rep movsb

; --- Chiudi il file --- mov bx, Handle]

mov ah, 3Eh int 21h

; (Prosegui elaborazione...) Errore:

mov ax, 4C00h int 21h

### 🧩 Funzioni Classiche

ScriviStringa: push bp

mov bp, sp

mov si, [bp+4 mov ah, 0eh

Stampa: lodsb

cmp al, 0

je fineStampa int 10h

jmp Stampa fineStampa:

pop bp ret

📚 **Teoria**

## 🛂 Passaggio per riferimento vs per valore

Stringa db 'A'

mov ax, Stringa ; AX  offset (es. 1050h) mov al, Stringa] ; AL  'A' (41h)

### Cosa contiene il parametro

 Se passo per riferimento, dentro il registro trovo un indirizzo.

 Se passo per valore, dentro il registro trovo

un dato.

## Impatto su array e stringhe

 Per lavorare su array/stringhe → si usa quasi sempre per riferimento (così puoi scorrere i caratteri con SI , DI , ecc.).

 Se ti serve solo un numero singolo (es. sommare un elemento) → puoi passarlo per valore.

## 📌 8086 – Modi di indirizzamento memoria

🔹 Forme possibili

 Solo registro: BX , BP , SI ,



DI

 Combinazione base + indice: BX+SI , BX+DI , BP+SI ,  Con spiazzamento (costante immediata):

BP+DI

BX+4 , BP+SI+10 , DI+100 , ecc.

🔹 Registri ammessi

* Base: BX ,



BP

* Indice: SI ,



DI

✅ **Esempi validi**

mov al, [bx] ; AL  byte in memoria allʼindirizzo DS:BX

mov ax, [bp+4 ; AX  word in SS:BP+4 (tipico per parametri su stack) mov dl, [si+10 ; DL  elemento di un array a offset 10

mov cx, [bx+di] ; CX  word in DS:BX+DI

mov ax, [bp+si+2 ; AX  parametro da stack frame

❌ **Esempi non validi**

mov al, [cx] ; CX non può fare da base/indice mov ax, [dx] ; DX idem

mov bx, [ax+10 ; AX non ammesso come base

👉 **Regola pratica:**

Se vuoi scorrere un array, usa SI/DI (tipici per stringhe/vettori). Se vuoi puntare allo stack, usa BP.

Se vuoi un “base index genericoˮ, usa BX.

## ⛓ String Istruction

### Registri fondamentali

|  |  |
| --- | --- |
| Registro | Uso principale |
| SI | Source Index → punta allʼinizio del  vettore sorgente |
| DI | Destination Index → punta  allʼinizio del vettore destinazione |
| CX | Contatore → numero di elementi da processare |
| AL / AX | Registro che contiene il dato da leggere o scrivere (byte o word) |
| DS / ES | Segmenti → DS:SI sorgente,  ES:DI destinazione |
| DF flag | Flag che permette di  incrementare/decrementare  SI / DI |

### Istruzioni Principali

|  |  |  |
| --- | --- | --- |
| Istruzione | Funzione | Commento |
| MOVSB | Copia un byte da DS:SI a  ES:DI | SI e DI increment decrementati se |
| MOVSW | Copia una word (2 byte) | SI/DI incrementat decrementati) |
| LODSB | Carica un byte  da DS:SI in AL | SI  incrementato/dec |
| LODSW | Carica una word da  DS:SI in AX | SI  incrementato/dec |
| STOSB | Scrive AL in  ES:DI | DI  incrementato/dec |
| STOSW | Scrive AX in  ES:DI | DI  incrementato/dec |
| SCASB | Confronta AL con ES:DI | Aggiorna flag, DI incrementato/dec |
| SCASW | Confronta AX con ES:DI | Aggiorna flag, DI incrementato/dec |





|  |  |
| --- | --- |
| Istruzione | Effetto |
| CLD | Clear DF \* SI/DI incrementano (forward) |
| STD | Set DF \* SI/DI decrementano (backward) |

|  |  |  |
| --- | --- | --- |
| Istruzione | Funzione | Commento |
| REP / REPE /  REPNE | Ripete lʼistruzione  precedente CX  volte, o finché condizione |  |

## 🔡Stringhe



📘 **Stringa C**

È un array di char terminato da un carattere speciale NUL ( 0x00 ).

💼 "ciao" → memoria = 63h 69h 61h 6Fh

00h .

Non serve sapere la lunghezza a priori: la scansione termina sul byte 00h .

Max Lunghezza: lʼunico limite è la memoria



📘 **Stringa Pascal**

Il primo byte contiene la lunghezza

della stringa.

💼 "ciao" → memoria = 04h 63h 69h 61h 6Fh .

Non cʼè terminatore, la scansione si basa sul primo byte.

Max Lunghezza: 255, cioè val max di 1 Byte

### Cicli con le String Instruction

CLD

CicloC:

lodsb ; AL  SI, SI++ cmp al, 0

je FineC ; trovato terminatore

; ... usa AL ...

jmp CicloC FineC:

mov si, Stringa+1 ; salto il byte di lunghezza mov cl, Stringa] ; lunghezza

xor ch, ch CLD

CicloP:

lodsb

; ... usa AL ...

loop CicloP

Per scorrere devo controllare ogni volta se Ho già la lunghezza in Stringa] , la metto in e

0 , non uso CX come contatore, ma il



AL 



CX

uso

per decrementare CX . So già

terminatore 00h . quanti caratteri ci sono



loop / rep

### Leggere La Lunghezza Della Stringa

xor cx, cx CicloLenC:

lodsb

cmp al, 0

je FineLenC inc cx

mov cl, Stringa]

La lunghezza è direttamente nel primo byte-

jmp CicloLenC FineLenC:

Devo contare manualmente i caratteri fino al terminatore 00h .

### Inserire un carattere

In C Devo:

 Spostare a destra i caratteri successivi



00h

In Pascal Devo:

 Aumentare il valore del primo byte

(incluso

finale).

(lunghezza).

Inserire il carattere nuovo.

Spostare a destra i caratteri successivi.

Lasciare il



00h

in fondo.

Inserire il nuovo carattere.

Esempio (aggiungere

in "ciao" ):

Esempio (aggiungere

in "ciao" ):



!

Prima: 63h 69h 61h 6Fh 00h . Dopo:



!

63h 69h 61h 6Fh 21h 00h

Prima: 04h 63h 69h 61h 6Fh .

Dopo: 05h 63h 69h 61h 6Fh 21h .

## Interrupt

### INT 10h - Video Services (BIOS)

Interrupt è usato per gestire l'output video (scrivere caratteri, cambiare colori, spostare il cursore...).



10h



Funzione **0eh** : Scrive un singolo carattere sullo schermo, come se fosse stato digitato.

⚙ Registri da impostare:

AH  0Eh → seleziona la funzione

AL  codice ASCII del carattere da stampare

BH  numero pagina (di solito 0 )

BL  [usato solo in modalità grafica]

📝 Note:

In modalità testo, colore predefinito (non personalizzabile qui)

Il cursore avanza automaticamente

mov ah, 0Eh mov al, 'A' mov bh, 0

int 10h

Funzione **09h** : Scrive lo stesso carattere più volte, con colore personalizzato, senza spostare il cursore.



### ⚙ Registri da impostare:

AH  09h

AL  carattere ASCII da stampare

BH  numero pagina (di solito 0 )

BL  attributo colore

CX  quante volte stampare il carattere

📝 **Note:**

Utile per cambiare colore di foreground / background

Il cursore non si muove

Colori: usa un byte BL codificato così:

Bit 7 \* Blink (1  lampeggia) Bit 6-4 \* Background (3 bit)

Bit 3-0 \* Foreground (4 bit)

🎨 Colori utili:

|  |  |
| --- | --- |
| Colore | Codice Foreground |
| Bianco | 07h |
| Grigio | 08h |
| Giallo | 0Eh |
| Bianco acc. | 0Fh |

|  |  |
| --- | --- |
| Colore | Codice Foreground |
| Nero | 00h |
| Blu | 01h |
| Verde | 02h |
| Rosso | 04h |

mov ah, 09h mov al, 'X' mov bh, 0

; colore: background blu, foreground giallo acceso + blink mov bl, 29h

mov cx, 1 int 10h

### INT 16h - Keyboard Services (BIOS)

Lʼinterrupt permette di interagire con la tastiera: leggere caratteri premuti o ottenere lo stato dei



16h

tasti speciali (Shift, Ctrl, ecc.).



Funzione **00h** : Legge un carattere dalla tastiera. Il programma si ferma in attesa del tasto (bloccante).

⚙ Registri da impostare:

AH  00h → seleziona la funzione Nessun input necessario

🔁 Valori ritornati:

AL  codice ASCII del tasto premuto

AH  scan code del tasto (utile per distinguere da dove arriva il carattere, es. tastierino numerico) (non cʼè bisogno per questo corso)

mov ah, 00h int 16h

|  |  |  |
| --- | --- | --- |
| Bit | Significato | Descrizione |
| 0 | Right Shift | Tasto Shift DX premuto |
| 1 | Left Shift | Tasto Shift SX premuto |
| 2 | Ctrl | Tasto Ctrl premuto |
| 3 | Alt | Tasto Alt premuto |
| 4 | Scroll Lock | Scroll Lock attivo |
| 5 | Num Lock | Num Lock attivo |
| 6 | Caps Lock | Caps Lock attivo |
| 7 | Insert | Tasto Insert attivo |

### INT 17h - Printer Services (BIOS)



Funzione **02h** : Legge lo stato attuale dei tasti speciali (Shift, Alt, Ctrl, Caps Lock, ecc.).

⚙ Registri da impostare:

AH  02h

🔁 Valore ritornato:

AL  stato dei tasti speciali (ogni bit indica un tasto attivo o meno)

🔍 Significato dei bit di AL (da meno significativo a più significativo):

mov ah, 02h int 16h

; AL ora contiene un valore bit a bit che rappresenta i tasti attivi

; Es: AL  01000010b \* Shift SX + Caps Lock attivi

Serve per stampanti parallele (molto vecchio, di solito mai usato a lezione).

### INT 1Ah - Time of Day (BIOS)

Serve per leggere ora e data dal Real-Time Clock (RTC) del BIOS. I valori ritornati sono in BCD (Binary-Coded Decimal).



Funzione **02h** : Legge lʼorario corrente (ora/minuti/secondi) dal BIOS.

⚙ Registri da impostare:

AH  02h

🔁 Valori ritornati (in BCD): CH  ore (es: 21h  21) CL  minuti (es: 37h  37)

DH  secondi (es: 08h  8)

mov ah, 02h int 1Ah

; Ora in CH, minuti in CL, secondi in DH



Funzione **04h** : Legge la data corrente (giorno/mese/anno/secolo) dal BIOS.

⚙ Registri da impostare:

AH  04h

🔁 Valori ritornati (in BCD):

CH  secolo (es: 20h  2000) CL  anno (es: 24h  2024) DH  mese (es: 06h = giugno) DL  giorno (es: 12h  12)

mov ah, 04h int 1Ah

; CH  secolo, CL  anno, DH  mese, DL  giorno

### ❓ Cos'è il formato BCD?

Ogni cifra decimale è rappresentata da 4 bit (es: 25 diventa 0010 0101  25h)

; Convertire BCD in decimale

; Es: CH  25h → (2 \* 10 + 5)  25

mov al, ch

mov ah, al and ah, 0Fh

and al, 0F0h shr al, 4

mov bl, 10 mul bl

add al, ah

; parte unitaria

; parte decine

; AL  decine \* 10

; AL  valore decimale

### INT 21h - INPUT, OUTPUT & EXIT (DOS)

Lʼinterrupt più utilizzato.



Funzione **02h** : Scrive un singolo carattere sullo standard output (monitor).

⚙ Registri da impostare:

DL  codice ASCII del carattere da scrivere

mov ah, 02h

mov dl, 'A' ; carattere da visualizzare int 21h



Funzione **06h** : NellʼOutput funziona come funziona 02h (scrive un carattere), nellʼ Input, se

DL  FFh , legge un carattere dallo standard input (tastiera) senza bloccare. Se nessun carattere disponibile, lo zero flag (ZF)  1

⚙ Registri da impostare:

Per output: DL  carattere ASCII

Per input: DL  0FFh (chiede un carattere, ma non aspetta)

🔁 Valore ritornato:

Se input, il carattere letto è in AL

ZF  1 se nessun carattere disponibile

; Come si usa per Output mov ah, 06h

mov dl, 'A' int 21h

; Come si usa per input non bloccante mov ah, 06h

mov dl, 0FFh int 21h

jnz carattere\_presente

; qui ZF  1 \* nessun carattere

Funzione **09h** : Stampa a video una stringa terminata dal carattere $ . (non utilizzata da noi)



💼 EsempioStrigna db “Ciao Mondo!$ˮ



Funzione **40h** : Scrive dati su un file o dispositivo usando un handle.

⚙ Registri da impostare:

BX  file handle (es: 1  stdout)

CX  numero di byte da scrivere DX  offset buffer/stringa

🔁 Valore ritornato:

AX  numero di byte effettivamente scritti

mov ah, 40h

mov bx, 1 ; stdout 1=output mov cx, lunghezza

mov dx, offset buffer int 21h

; AX contiene il numero di byte scritti



Funzione **01h** : Legge un carattere dalla tastiera e lo stampa subito a schermo (echo). È bloccante, si aspetta che venga premuto un tasto,

🔁 Valore ritornato:

AL  codice ASCII del tasto premuto

mov ah, 01h int 21h

; AL ora contiene il carattere, e lo ha anche stampato



Funzione **3Fh** : Legge dati da un file o da uno stream come la tastiera (stdin).

⚙ Registri da impostare:

BX  file handle → 0 = standard input (tastiera)

CX  numero di byte da leggere

DX  offset del buffer in cui memorizzare i dati

🔁 Valore ritornato:

AX  numero di byte effettivamente letti

Nota: legge senza echo e può leggere anche da file o pipe.

mov ah, 3Fh

mov bx, 0 ; stdin

mov cx, 10 ; leggi massimo 10 byte mov dx, offset buffer

int 21h

; AX  numero di byte letti



Funzione **4Ch** : Termina il programma e restituisce un codice di uscita al DOS.

⚙ Registri da impostare:

AL  codice di uscita (0 \* successo)

mov ah, 4Ch

mov al, 00h ; codice di uscita int 21h