Assembler Intel 80x86: Set delle istruzioni

Calcolatori Elettronici B

a.a. 2004/2005

Massimiliano Giacomin

Classi di istruzioni

- Il set di istruzioni dell'Assembler Intel 80x86 può essere suddiviso nelle seguenti classi:
  - Istruzioni di *trasferimento*: mov, push, pop, …
  - Istruzioni *aritmetiche*: add, sub, cmp, mul, div, …
  - Istruzioni *logiche*: and, or, xor, not, …
  - Istruzioni di *manipolazione di bit*: shl, shr, …
  - Istruzioni di *controllo*: jmp, call, ret, jg, jge, loop, …
  - Istruzioni di *input/output*: in, out, …
  - Istruzioni di *manipolazione di stringhe*: movs, stos, lods, …
  - Istruzioni di *controllo dello stato della macchina*: hlt, wait, …
- Nel seguito vedremo alcuni esempi di istruzioni (nel caso dell'8086) per ciascuna classe

Istruzioni di trasferimento

- MOV
- XCHG
- LEA
- LDS, LES
- PUSH, POP

NB: nessuna istruzione altera i flag!

Istruzioni di trasferimento: mov

mov *destinazione*, *sorgente*

L'istruzione mov copia il contenuto di *sorgente* in *destinazione*

Sorgente	Destinazione
registro*	registro
immediato	registro
immediato	memoria
memoria	registro
registro	memoria
registro/memoria	registro di segmento (eccetto CS)
registro di segmento	registro/memoria

*il termine "registro" indica qui un registro general purpose

Esempio di uso di mov

Uso dell'istruzione mov per lo scambio del contenuto di due variabili di tipo byte definite come opr1 e opr2

```
mov al, opr1
mov bl, opr2
mov opr1, bl
mov opr2, al
```

Istruzioni di trasferimento: xchg

xchg *operando1*, *operando2*

dove xchg sta per “exchange”, cioè questa istruzione scambia il contenuto di *operando1* con il contenuto di *operando2,* almeno uno degli operandi deve essere un **registro** (general purpose) e, ovviamente, nessun operando può essere immediato

- Esempio: scambio del contenuto delle variabili opr1 e opr2

```
mov     al, opr1
xchg    al, opr2
mov     opr1, al
```

Istruzioni di trasferimento: lea

lea *registro*, *indirizzo*

- lea sta per "load effective address"
- con *registro* si intende un registro general-purpose
- con *indirizzo* si intende un qualsiasi modo di indirizzamento alla memoria (tutti tranne *immediate* e *register*)
- Esempio:

```
lea ax, 1000h[bx][si]
```

  carica in ax ***l'indirizzo effettivo*** della locazione di memoria puntata da 1000h[bx][si], ovvero l'**indirizzo** ottenuto sommando 1000h con il contenuto di bx e con il contenuto di si

- Esempio: istruzioni equivalenti (ma con codifica diversa!)

```
lea DX, VAR                ; indirizz. diretto
mov DX, OFFSET VAR         ; mov con indirizz. immediato
```

Istruzioni di trasferimento: lds e les

lds *registro, memoria*

- lds sta per "load data segment", permette di caricare simultaneamente in DS un indirizzo di segmento e in *registro* un indirizzo effettivo
- per *registro* si intende un registro general purpose
- con *memoria* si intende una locazione di memoria costituita da una doppia parola (ovvero una variabile dichiarata con DD) che contiene un indirizzo completo (*segmento:offset*)
- Esempio:

```
lds     BX, d_word     ; d_word di tipo double!
```

  carica in BX i primi due byte di d_word e in DS i due byte successivi di d_word

- les è simile eccetto che l'indirizzo di segmento viene caricato in ES

Istruzioni di trasferimento: push e pop

push ***operando***
pop ***operando***

- *operando* è sempre di tipo Word;
  può essere un **registro** general purpose, di segmento (eccetto l'istruzione pop CS che è illegale), una locazione di memoria (quindi una variabile) e un immediato (nel solo caso di push)
- Push e pop cambiano automaticamente il valore dello stack pointer SP; stack cresce verso indirizzi più bassi
- SS:SP contiene sempre l'indirizzo del valore presente sulla cima dello stack, ovvero quello inserito per ultimo

- PUSH: SP=SP-2
  (SP)←operando

  POP: (SP) → operando
  SP = SP+2

Istruzioni aritmetiche

- ADD e SUB
- ADC e SBB
- MUL e IMUL
- DIV e IDIV
- INC, DEC, NEG, CMP

NB: queste istruzioni modificano i flag nel modo consueto

Istruzioni aritmetiche: add e sub

add ***destination, source***

- Aggiunge il contenuto di *source* al contenuto di *destination* lasciando il risultato in *destination*

sub ***destination, source***

- Sottrae il contenuto di *source* dal contenuto di *destination* lasciando il risultato in *destination*
- Al solito, operando destinazione non può essere immediato e non è possibile avere entrambi gli operandi in memoria;
- I flag *sign*, *zero*, *parity*, *carry* sono modificati nel modo ovvio; il flag *overflow* è modificato secondo le regole del complemento a 2 (ovvero, considerando l'operazione come signed).
- add e sub possono gestire operandi di un byte o una parola; per estendere la precisione delle operazioni si usano **adc** e **sbb**

Istruzioni aritmetiche: adc e sbb

adc *destination, source*

- Aggiunge il contenuto di *source* al contenuto di *destination* mettendo il risultato in *destination,* tenendo conto di un riporto precedente (nel carry flag):
  (destination) ← (source) + (destination) + (CF)

sbb *destination, source*

- Sottrae il contenuto di *source* dal contenuto di *destination* lasciando il risultato in *destination,* tenendo conto di un prestito precedente (nel carry flag):
  (destination) ← (destination) - (source) - (CF)

- **Esempio**: usiamo addizioni a 16 bit per ottenere una somma a 32 bit. Siano op1, op2 e sum variabili dichiarate con DW (parola), ciascuna contenenti due parole successive

```
mov  ax, op1      ; somma le parole meno significative
add  ax, op2      ; di op1 e op2, il risultato
mov  sum, ax      ; viene copiato nella parola meno significativa di sum
mov  ax, op1+2    ; somma (con carry) le parole più significative
adc  ax, op2+2    ; di op1 e op2, il risultato
mov  sum+2, ax    ; viene copiato nella parola più significativa di sum
```

Istruzioni aritmetiche: inc, dec, neg, cmp

inc ***operando***
dec ***operando***
neg ***operando***
cmp ***operando1, operando2***

- inc aggiunge 1 all'operando, dec sottrae 1 dall'operando e neg nega l'operando
- Per inc, dec, neg non è ovviamente consentito il modo immediato, ma tutti gli altri modi di indirizzamento sono permessi
- inc e dec sono usati per conteggiare e indicizzare (p.es. nei cicli); ma non modificano il flag CF…
- cmp opera come sub tranne che il risultato non viene memorizzato da nessuna parte. Modifica il registro flags ed è normalmente usato prima delle istruzioni di salto condizionato.
- NB: operando come sub, nell'istruzione cmp operando1 non può essere immediato, per il resto valgono tutti i modi di indirizzamento (tranne memoria-memoria)

Istruzioni aritmetiche: mul, imul

mul ***operando***
imul ***operando***

- se *operando* è un byte: moltiplica il contenuto del registro al con il contenuto di *operando* e salva il risultato in ax (2 byte)
- se *operando* è una parola: moltiplica il contenuto del registro ax con il contenuto di *operando* e salva il risultato in dx (high order bytes) e ax (low order bytes)
- Notare che un overflow non è possibile
- *operando* può essere un registro o una locazione di memoria, ma non un immediato
- imul opera nello stesso modo, ma gli operandi e il risultato sono *con segno* (notare che in questo caso è necessario distinguere tra operazioni con segno e senza segno, perché le regole sono diverse)

Istruzioni aritmetiche: div, idiv

div ***operando***
idiv ***operando***

- se *operando* (il divisore) è un byte: divide il contenuto del registro ax per il contenuto di *operando* e mette il quoziente in al e il resto in ah
- se *operando* (il divisore) è una parola: divide il contenuto di dx:ax per il contenuto di *operando* e mette il quoziente in ax e il resto in dx
- *operando* può essere un registro o una locazione di memoria, ma non un immediato
- Poiché il dividendo è due volte più lungo dello spazio per il quoziente, è possibile un overflow (si pensi a divisione per 1) ma OF non dà nessuna informazione a riguardo
- idiv opera nello stesso modo, ma con segno (in particolare il resto assume lo stesso segno del dividendo)

Istruzioni logiche

Istruzioni logiche: not, and, or, xor, test

not	***operando***
and	***destinazione, sorgente***
or	***destinazione, sorgente***
xor	***destinazione, sorgente***
test	***operando1, operando2***

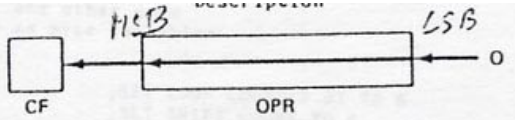
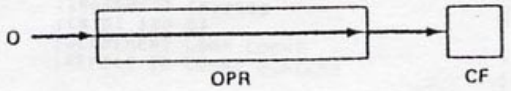
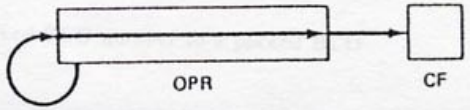
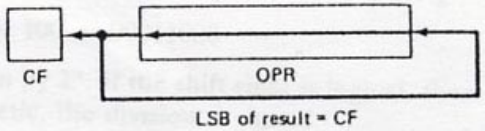
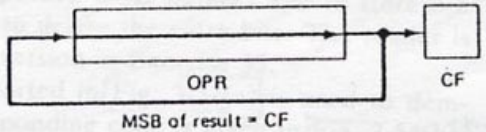
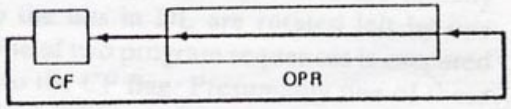
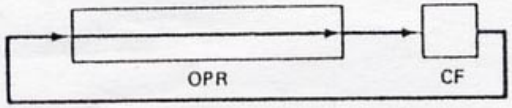
- Sono operazioni bit a bit sugli operandi.
- *operando,* nell'istruzione not, non può essere un immediato. Esso funge sia da sorgente che da destinazione.
- In and, or e xor, destinazione non può essere un immediato e non è consentito avere entrambi gli operandi in memoria. Il risultato dell'operazione viene salvato nel primo operando.
- L'istruzione test esegue la stessa operazione di and ma non salva il risultato da nessuna parte, modifica solo il registro flags (e quindi viene in genere usata prima di un salto condizionato)

Istruzioni di manipolazione dei bit

Istruzioni di manipolazione di bit: shl, shr

shl ***operando, contatore***
shr ***operando, contatore***

- shl (shift logical left) effettua lo scorrimento a sinistra di *operando* di tanti bit quando vale *contatore* (equivale a una moltiplicazione per una potenza di 2)
- l'ultimo bit che "esce" da *operando* viene posto nel carry flag
- shr (shift logical right) effettua lo scorrimento a destra di *operando* di tanti bit quando vale *contatore* (divisione per potenza di 2)
- l'ultimo bit che "esce" da *operando* viene posto nel carry flag
- *contatore* deve valere 1 oppure essere in CL
- Altre istruzioni di manipolazione di bit sono: sal (shift arithmetic left), sar (shift arithmetic right), rol (rotate left), ror (rotate right), rcl (rotate left through carry), rcr (rotate right through carry)

Name	Mnemonic and format	Description
Shift logical left	SHL OPR,CNT	
Shift arithmetic left	SAL OPR,CNT	Same as SHL
Shift logical right	SHR OPR,CNT	
Shift arithmetic right	SAR OPR,CNT	
Rotate left	ROL OPR,CNT	
Rotate right	ROR OPR,CNT	
Rotate left through carry	RCL OPR,CNT	
Rotate right through carry	RCR OPR,CNT	

*Number of bit positions shifted is determined by CNT

Flags: CF flag set as indicated. PF, SF, and ZF flags are set by shift instructions but left unchanged by the rotate instructions. OF flag is meaningful only if count is 1. AF flag is affected by shift instruction, but has no meaning.

Addressing modes: OPR can have any mode except immediate; CNT must be 1 or CL.

Istruzioni di controllo

Istruzioni di controllo: salti condizionati

j*cnd* *etichetta*

- dove *etichetta* è indirizzata da uno scostamento con segno di 8 bit (da – 128 a + 127) relativo all'indirizzo della istruzione successiva (puntata da IP) e *cnd* include una, due o tre lettere che indicano la condizione
- Il salto è solamente intrasegment diretto "short"
- L'istruzione cmp viene in genere usata prima di un'istruzione di salto condizionato: essa modifica il registro flags a seconda della relazione esistente fra due operandi e il branch viene preso a seconda del valore dei flag
- Alcuni esempi: jz (salta se zero), jnz (salta se non zero), jl (salta se minore), jle (salta se minore o uguale), jnl (salta se non minore), jnle (salta se non minore o uguale), jg (salta se maggiore), jge (salta se maggiore o uguale)

Salti condizionati: esempio

```
                mov     al, x       ; poni il contenuto di x in al
                mov     bl, y       ; poni il contenuto di y in bl
                mul     bl          ; ax = al*bl
                cmp     ax, 50      ; confronta ax con 50
                jg      maggiore            ; se ax > 50 salta a “maggiore”
                …
maggiore:       …
```

NB: si vede come in assembler non occorra esprimere direttamente lo scostamento: basta riferirsi ad una etichetta (cfr. poi i modi di indirizzamento in caso di salto).

CMP op1, op2

JZ [JE]	op1 = op2	ZF = 1	
JNZ [JNE]	op1 ≠ op2	ZF = 0	
JB [JNAE o JC]	op1 < op2	CF = 1	senza segno
JNB [JAE o JNC]	op1 ≥ op2	CF = 0	
JBE [JNA]	op1 ≤ op2	(CF ∨ ZF) = 1	
JNBE [JA]	op1 > op2	(CF ∨ ZF) = 0	
JL [JNGE]	op1 < op2	(SF XOR OF) = 1	con segno
JNL [JGE]	op1 ≥ op2	(SF XOR OF) = 0	
JLE [JNG]	op1 ≤ op2	(SF XOR OF) ∨ ZF = 1	
JNLE [JG]	op1 > op2	(SF XOR OF) ∨ ZF = 0	

ALTRE ISTRUZIONI: JS, JNS, JO, JNO, JP, JNP

saltano sulla base del flag di segno, overflow e parità rispettivamente.

Istruzioni di controllo: salti incondizionati

jmp ***operando***

- l'istruzione jmp, tradotta in linguaggio macchina, può avere diversi formati (si vedano i modi di indirizzamento in caso di salto)
- Esempi:

```
jmp     skip            ; dove skip è un'etichetta
jmp     table[si]       ; salta all'indirizzo contenuto nella
                        ; locazione di memoria indicata
                        ; con table[si]
```

Modalità di indirizzamento nei salti

1. **Diretto (Intrasegment e Intersegment)**: *Etichetta ± espressione costante.*

   Esempi:

```
jmp Label              ; decide l'assemblatore il tipo di salto
jmp SHORT Label        ; forza un intrasegment +displacement a 8 bit
jmp NEAR PTR Label     ; forza un intrasegment near
jmp FAR PTR Label      ; forza un intersegment
```

2. **Indiretto (Intrasegment e Intersegment):** *Indirizzamento ai dati*

   Se il dato è word è intrasegment, se è double word è intersegment

   Esempi:

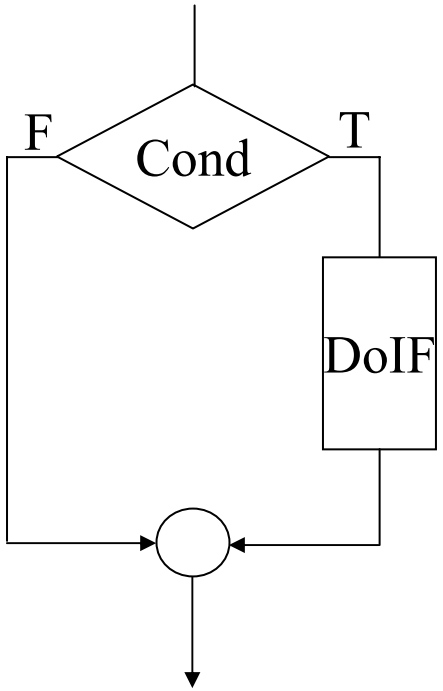
```
jmp BX          ; intrasegment,  salta all'indirizzo contenuto in BX
jmp Table[SI]   ; salta all'indirizzo contenuto nella locazione di
                ; memoria indicata con Table[SI]
                ; Intrasegment se Table è Word, Intersegment se double
```

Implementazione in assembler delle strutture di controllo più comuni

- IF…THEN
- IF…THEN…ELSE
- CASE
- Ciclo WHILE
- Ciclo FOR

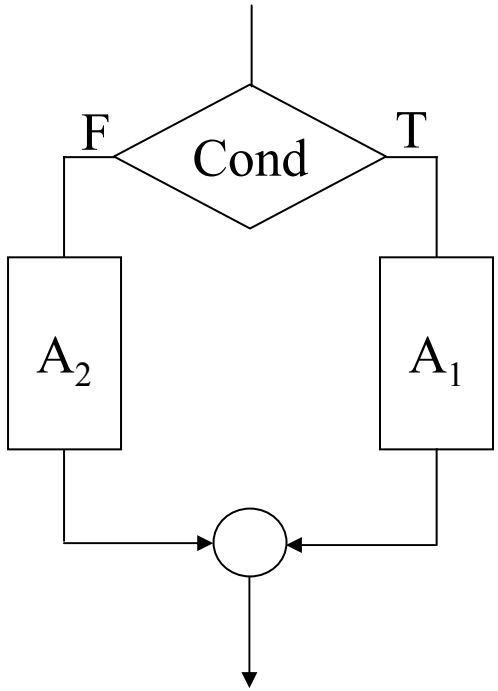
IF … then

```
; IF ( ((X > Y) && (Z < T)) || (A != B) )   C = D;
; implementato come:

; IF (A != B) Goto DoIF;
            mov     ax, A
            cmp     ax, B
            jne     DoIF
; IF !(X >Y) Goto EndIF;
            mov     ax, X
            cmp     ax, Y
            jng     EndIF
; IF !(Z <T) Goto EndIF;
            mov     ax, Z
            cmp     ax, T
            jnl     EndIF
DoIF:       mov     ax, D
            mov     C, ax
EndIF:
```



IF … then … else

```
;if (a == b) c = d else b = b+1
;implementato come

            mov     ax, a
            cmp     ax, b
            jne     Else
            mov     ax, d       ; A1
            mov     c, ax       ; A1
            jmp     EndIF
Else:       inc     b           ; A2
EndIF:              …
```



CASE

- Frammento di programma C:

```
switch (k) {
        case 0: f = i + j; break;
        case 1: f = i - j; break;
        case 2: f = i + 1; break;
}
```

- **In assembler:**
  Nel segmento dati dichiariamo le seguenti variabili:

```
i          DW    ?
j          DW    ?
f          DW    ?
k          DW    ?
JmpTbl     DW    Istr1, Istr2, Istr3
```

  E assumiamo di aver acquisito i valori delle variabili i, j, k

CASE in Assembler

```
                …
                mov     bx, k
                shl     bx, 1           ; moltiplica bx per 2
                jmp     JmpTbl[bx]      ; salta alla locazione di
                                        ; memoria corrispondente
                                        ; al valore di k
istr1:          mov     ax, i
                add     ax, j           ; esegue i + j
                mov     f, ax           ; e copia il risultato in f
                jmp     endcase
istr2:          mov     ax, i
                sub     ax, j           ; esegue i - j
                mov     f, ax           ; e copia il risultato in f
                jmp     endcase
istr3:          mov     ax, i
                inc     ax              ; esegue i + 1
                mov     f, ax           ; e copia il risultato in f
endcase:        …
```

NB: vedi `case.asm`

Ciclo While

Il codice C

```
I = 0;
while (I<100) I = I + 1;
```

In Assembler diventa:

```
        mov     I, 0
While:  cmp     I, 100
        jge     WhileDone       ; se I >= 100 esce
        inc     I
        jmp     While
WhileDone:
```

Ciclo For

Il codice C

for (i=inizio;i<=fine;i++) {*corpo del for*}

In Assembler diventa:

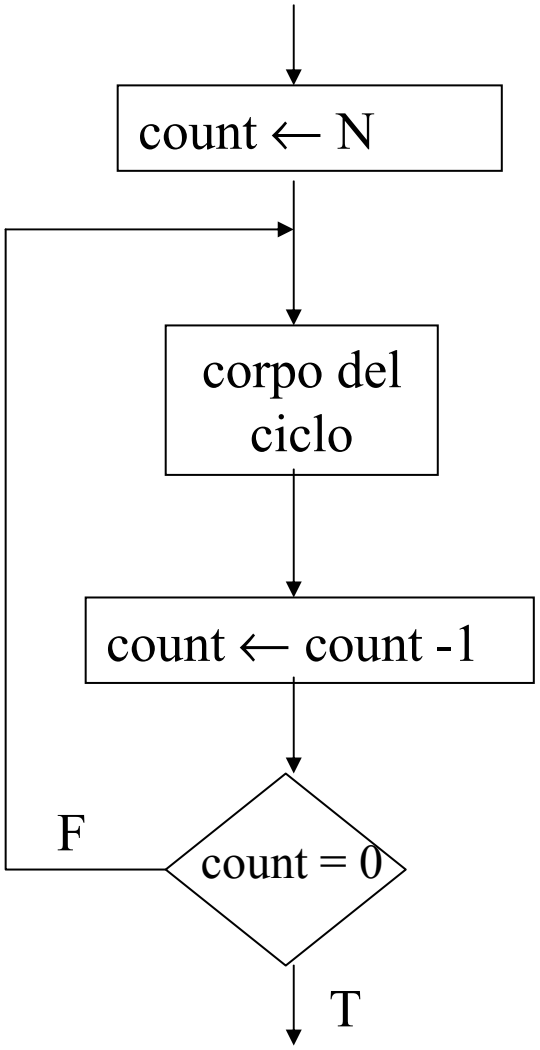
```
            mov     i, inizio
For:        mov     ax, i
            cmp     ax, fine
            jg      EndFor
            …       ; istruzioni nel corpo del for
            inc     i
            jmp     For
EndFor:
```

Istruzioni di controllo: loop, loopz, loopnz, loope, loopne

			Condizione controllata
loop		*etichetta*	cx ≠ 0
loopz	(o loope)	*etichetta*	ZF = 1 e cx ≠ 0
loopnz	(o loopne)	*etichetta*	ZF = 0 e cx ≠ 0

- Tutte le istruzioni decrementano di un'unità il contenuto del registro cx
- Se la condizione è soddisfatta, viene aggiornato il contenuto di IP tenendo conto dell'indirizzo in *etichetta*
- Lo scostamento rispetto all'IP (che già contiene l'indirizzo della istruzione successiva al loop) può essere fra –128 e +127 byte
- Il decremento di cx non modifica i flag!

Istruzioni di controllo: loop




```
        mov     cx, N
Begin:
        …       ; corpo del ciclo
        loop    Begin
```

L'istruzione loop richiede l'utilizzo del registro cx il cui contenuto viene decrementato automaticamente dall'istruzione loop

loop controlla quando il contenuto di cx diventa zero

Esempio di uso di loop

- Scriviamo la porzione di codice che somma i numeri contenuti in array e pone il risultato in total. Sia m il numero di elementi di array
- Siano m, array e total variabili dichiarate come word (cioè con direttiva DW)

```
        mov     cx, m       ; pone il numero degli elementi in cx
        mov     ax, 0       ; azzera registro ax
        mov     si, ax      ; azzera registro si
Ciclo:  add     ax, array[si]       ; aggiungi il prossimo elemento
                                    ; al contenuto di ax
        add     si, 2       ; incrementa l'indice dell'array
        loop    Ciclo       ; ripete il ciclo finchè cx è ≠ 0
        mov     total, ax
```

Istruzione jcxz

- Se l'array contiene 0 elementi (cx=0) si entra comunque nel loop eseguendolo 2^{16} volte
- L'istruzione jcxz salta all'etichetta se cx è nullo:

```
            mov     cx, m       ; pone il numero degli elementi in cx
            jcxz    ArrayVuoto  ; esce subito se l'array è vuoto
            mov     ax, 0       ; azzera registro ax
            mov     si, ax      ; azzera registro si
Ciclo:      add     ax, array[si]   ; aggiungi il prossimo elemento
                                    ; al contenuto di ax
            add     si, 2       ; incrementa l'indice dell'array
            loop    Ciclo       ; ripete il ciclo finchè cx è ≠ 0
            mov     total, ax
```

Esempio di uso di loopne

- Scriviamo la porzione di codice che permette di cercare un carattere ‘spazio’ in una stringa (ascii_str) di L caratteri
- Il ciclo termina quando viene trovato lo spazio oppure la stringa è finita

```
        mov     cx, L       ; mette in cx il numero dei caratteri
        mov     si, -1      ; inizializza l'indice si
        mov     al, 20h     ; pone in al il codice ascii dello spazio
Next:   inc     si          ; incrementa l'indice
        cmp     al, ascii_str[si];    confronta il carattere in al con il
                            ; carattere della stringa nella posizione si
        loopne  Next        ; continua se non è lo spazio e se la
                            ; stringa non è finita
```

- Usiamo una porzione di codice simile per scrivere un programma che prende una stringa e la copia in un’altra stringa togliendo gli spazi (vedi `stringhe.asm`)

Definizione e gestione di Procedure e Macro

Procedure:
direttiva proc, Istruzioni di controllo call e ret

- Definizione di procedura:

```
nome_proc    proc  [near|far]
             ; corpo procedura
             ret
nome_proc    endp
```

- L'istruzione

```
call    nome_proc
```

  effettua la chiamata di procedura

- L'istruzione **ret** effettua il ritorno dalla procedura

- Esistono due tipi di **call** e **ret** a seconda che la procedura sia NEAR o FAR

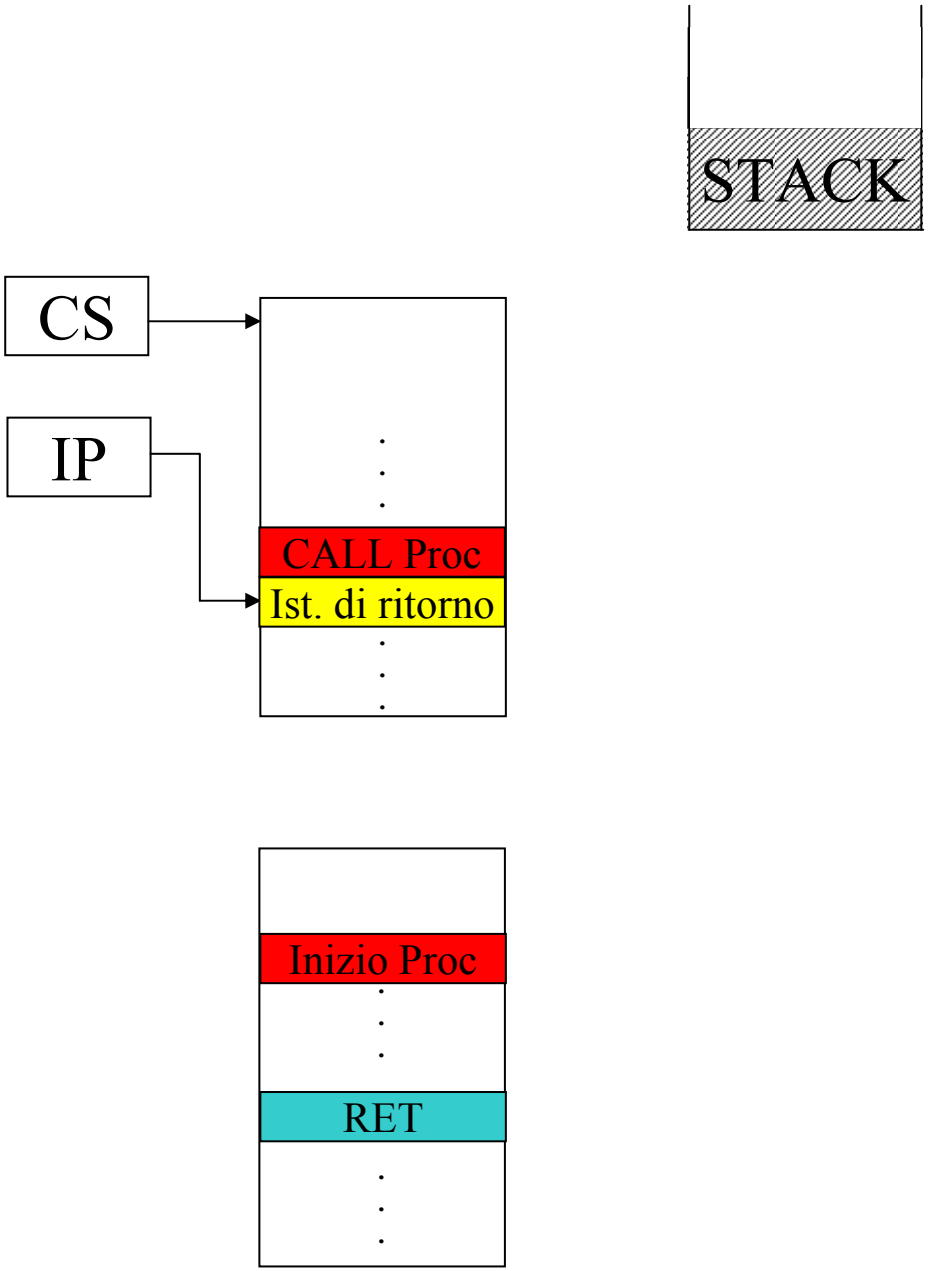
Near vs. Far

- near e far: attributi di distanza
- **near** corrisponde a un riferimento a un'etichetta o a una procedura all'interno dello stesso segmento
- **far** corrisponde a un riferimento a un etichetta o a una procedura fuori dal segmento
- Con i riferimenti near cambia solo l'IP
- Con i riferimenti far cambia sia IP che CS

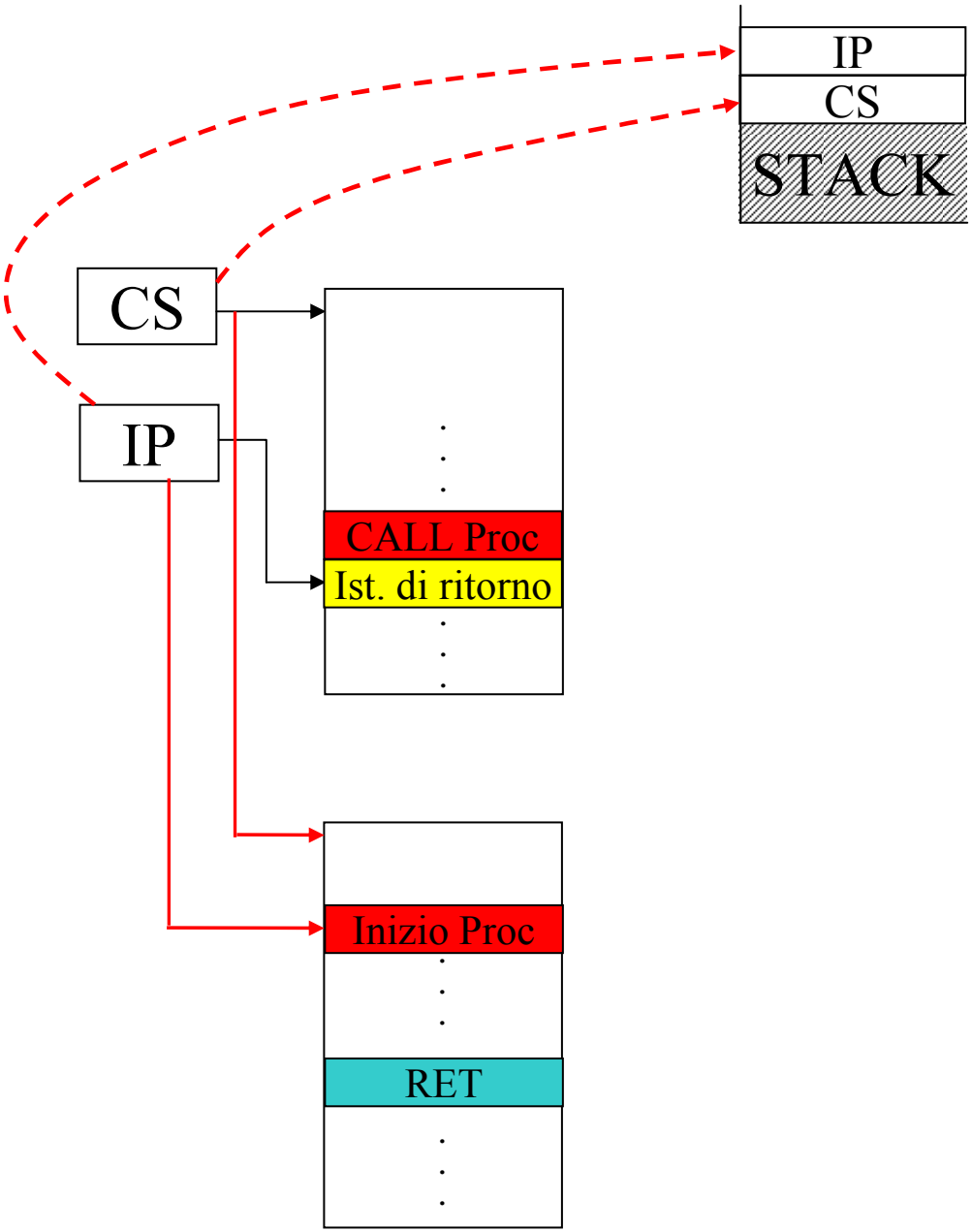
Procedure di tipo near e far

- La direttiva proc ha un operando opzionale che può essere near o far
- La procedura è near di default, se l'operando viene omesso
- Procedure di tipo near possono essere chiamate soltanto dallo stesso segmento di codice, procedure di tipo far possono essere chiamate anche da un altro segmento
- L'assemblatore traduce in modo diverso le istruzioni call e ret (di tipo near o far) a seconda del tipo associato al simbolo che costituisce il nome della procedura.
- Nel caso near è sufficiente salvare (call) e ripristinare (ret) soltanto il registro IP, nel caso FAR è necessario salvare e ripristinare sia CS che IP

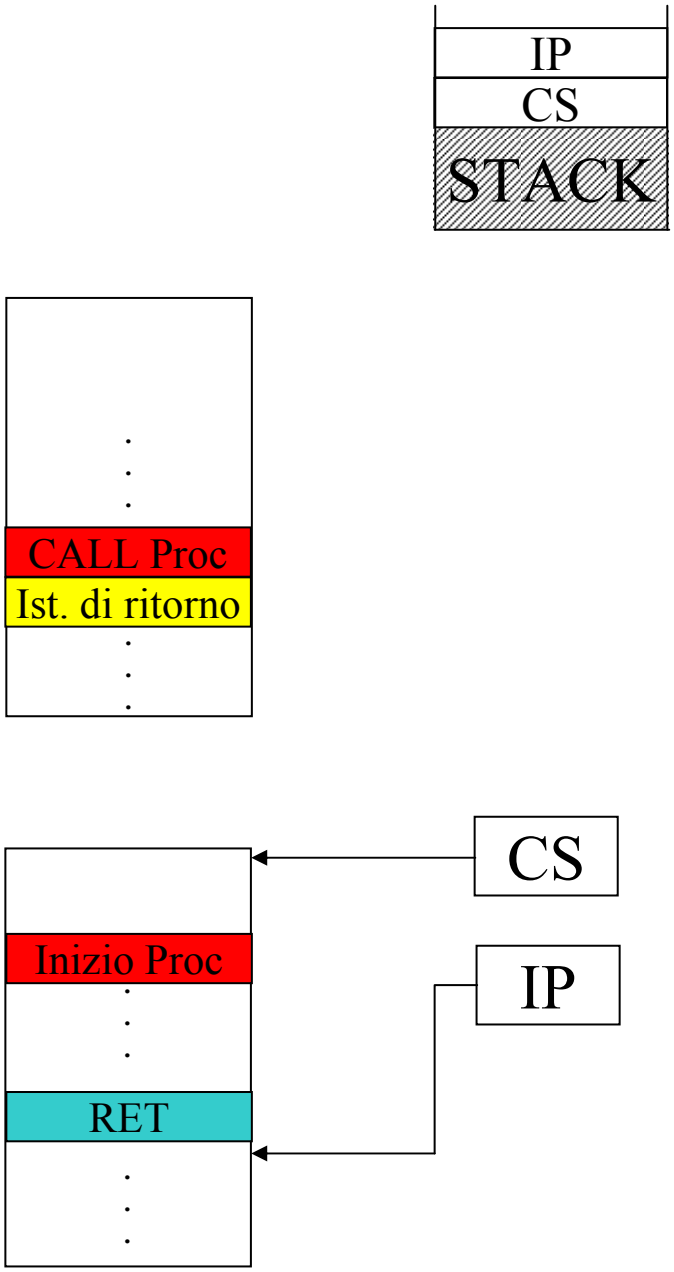
Call di tipo FAR



Call di
tipo FAR

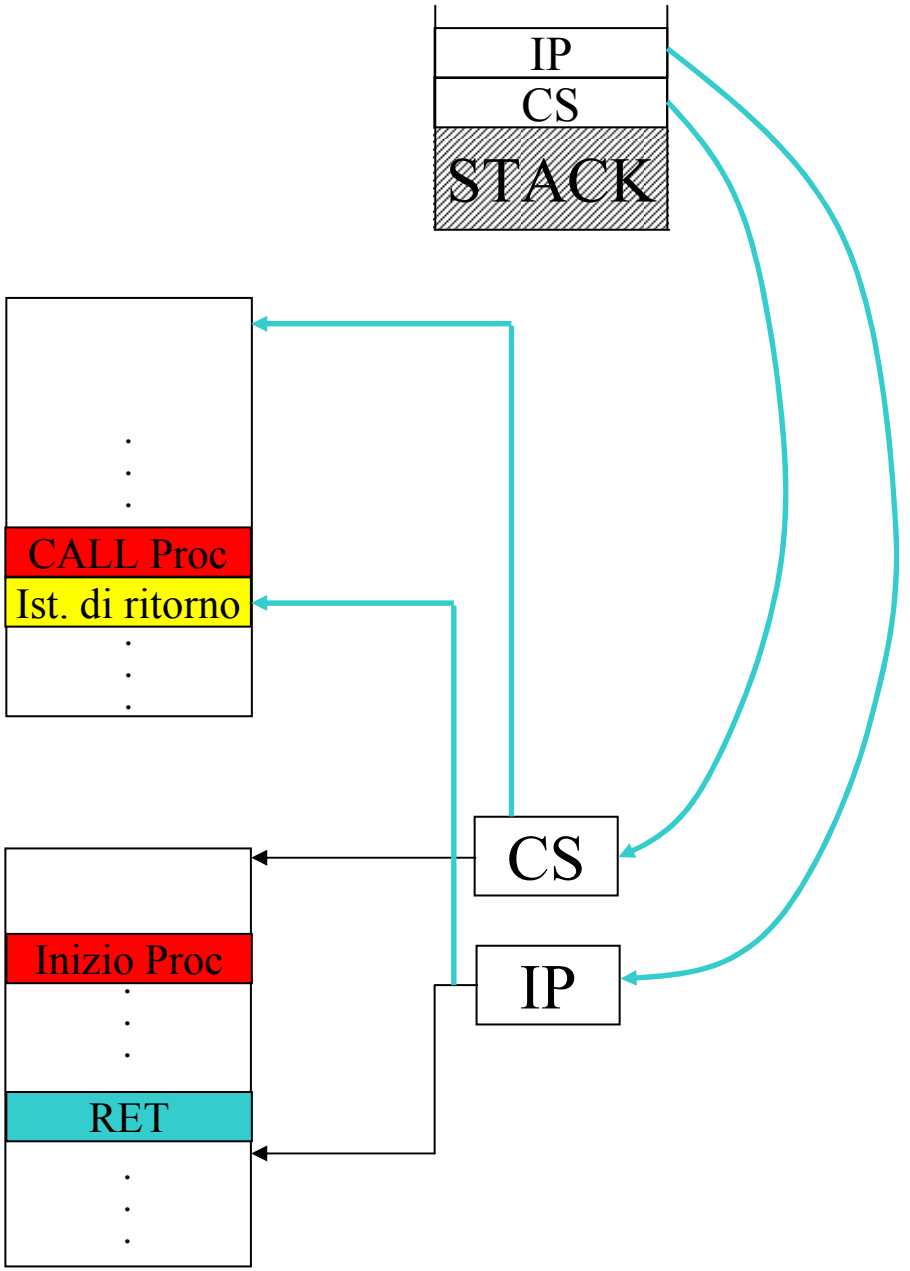


Call di
tipo FAR



Ret di
tipo FAR

Call di
tipo FAR



Ret di
tipo FAR

Uso di procedure e salvataggio dello stato della macchina

- Si usano le istruzioni push e pop per preservare i valori dei registri se questi vengono utilizzati all'interno di una procedura

- Esempio: chiamata di procedura all'interno di un loop

```
Loop0:      mov cx, 10
            call  Procedure1
            …
            loop Loop0          ; usa cx come contatore
```

```
Procedure1  proc near
            mov cx, 40
PSLoop:     …
            loop PSLoop         ; usa cx come contatore
            ret
Procedure1  endp
```

Salvataggio registri

- Primo caso: salvataggio nella procedura chiamata

```
Procedure1      proc near
                push cx             ; salva valore di cx per chiamante
                mov cx, 40
PSLoop:         …
                loop PSLoop         ; usa cx come contatore
                pop cx              ; ripristina valore di cx per chiamante
                ret
Procedure1      endp
```

- Secondo caso: salvataggio nella procedura chiamante

```
Loop0:          mov cx, 10
                push cx             ; salva cx prima di fare la call
                call  Procedure1
                …
                pop cx              ; ripristina cx per loop
                loop Loop0          ; usa cx come contatore
```

Pregi e difetti dei due casi

- Nel primo caso:
  - le istruzioni push e pop vanno in un solo punto, cioè all'interno della procedura, mentre mettendole nel chiamante bisogna averle prima e dopo ogni call
- Nel secondo caso:
  - Vengono preservati solo i registri necessari, ovvero nel chiamante si preservano solo i registri che si ritiene di dover salvare, mentre scrivendo una procedura si potrebbe non sapere che non c'è bisogno di salvare certi registri

Passaggio parametri: valori nei registri

- Per passare dati di ridotte dimensioni a una procedura si usano in genere i registri
- Esempio:

  un byte: in al

  una parola: in ax

  una doppia parola: in dx:ax
- Una tacita convenzione per l'uso dei registri nel passaggio dei parametri sembra essere il seguente ordine: ax, dx, si, di, bx, cx

Esempio: procedura PutChar

```
PutChar     PROC NEAR
            MOV AH, 2       ; codice funzione per display char
            INT 21H         ; interrupt del DOS
            RET
PutChar     ENDP
```

- Questa funzione si aspetta il codice di un carattere ASCII nel registro DL. Quindi nel programma chiamante si avrà, ad esempio:

```
MOV     DL, ' '    ; copia il DL il carattere spazio
CALL    PutChar
```

Passaggio parametri: indirizzi nei registri

- Un altro modo per passare parametri è passare un puntatore a un certo dato in memoria (passaggio per riferimento)
- Si passa quindi un indirizzo che può essere solo un offset oppure un indirizzo di segmento e un offset
- Nel primo caso è sufficiente un registro a 16 bit: si usano generalmente si, di, bx
- Nel secondo caso occorrono 32 bit, e quindi si può usare la combinazione dei registri dx:ax come un normale passaggio di doppia parola
- Per indirizzi segmentati si possono anche usare: ds:bx, ds:si, ds:di, es:bx, es:si, es:di

Esempio: procedura PutString

```
PutString   PROC
            MOV AH, 9       ; codice funzione per display stringa
            INT 21H         ; interrupt del DOS
            RET
PutString   ENDP
```

- Questa procedura si aspetta il puntatore alla stringa da visualizzare nella combinazione di registri DS:DX.
- Sia Stringa una stringa definita del segmento dati 'DATA', si avrà:

```
MOV AX, SEG DATA        ; necessari 2 trasferimenti per porre
MOV DS, AX;             ; in DS indirizzo del segmento dati
MOV DX, OFFSET Stringa ; pone offset in DX
CALL PutString          ; chiamata procedura con puntatore
                        ; alla stringa in DS:DX
```

NB: vedi `spazi.asm` per un esempio d'uso combinato delle procedure introdotte

Esempio: Passaggio per riferimento con modifica di una variabile

- Si assuma di disporre di una procedura che copia il contenuto della variabile, la cui locazione è specificata in BX, nella variabile la cui locazione è specificata in SI e che incrementa poi la variabile la cui locazione è in BX:

  J=I; I++

- Per chiamare la procedura con le variabili I e J si farà:

```
lea     bx, I              ; carica indirizzo (offset) di I in bx
lea     si, J              ; carica indirizzo (offset) di J in si
call    CopyAndInc         ; chiamata procedura
                           ; al ritorno I e J risultano modificate
```

Procedura CopyAndInc

```
CopyAndInc  Proc
            push    ax          ; preserva ax
            push    cx          ; preserva cx
            mov     ax, [bx]    ; fa una copia di I(*)
            mov     cx, ax      ; fa una seconda copia di I
            inc     ax          ; incrementa valore di I
            mov     [si], cx    ; copia vecchio valore di I in J
            mov     [bx],ax     ; copia nuovo valore di I
            pop     cx          ; ripristina cx
            pop     ax          ; ripristina ax
            ret
CopyAndInc  endp
```

(*) Per semplicità, nei commenti usiamo I e J anche se potrebbero essere altre variabili (dipende dai parametri passati)

Macro

- Una macro è una sequenza di istruzioni che l'assemblatore replica nel programma ogni volta che viene usata (esecuzione veloce ma codice ripetuto). La dichiarazione viene fatta con:

```
NomeMacro        macro [parametri]
                 …
                 endm
```

- Una procedura è invece una sezione di codice "a cui si salta" quando viene chiamata (overhead dovuto a call/ret ma la sezione di codice compare una sola volta nel programma)
- Le macro si usano di solito quando si vogliono migliorare le prestazioni in termini di tempo di esecuzione
- Si possono avere anche macro con parametri (vedi esempio)
- Le macro si dichiarano di solito vicino alle dichiarazioni di costanti (istruzioni EQU), ovvero dopo il segmento dati e prima del segmento codice

Esempio di macro

- Esempio: la procedura PutString come macro con parametro (vedi `spazi2.asm`)

```
PutString   MACRO text
            MOV DX, OFFSET text      ; pone offset in DX
            MOV AH, 9                ; codice funzione per display di una stringa
            INT 21H                  ; interrupt del DOS
            ENDM
```

- Chiamata della macro:

```
            MOV AX, SEG DATA  ; necessari 2 trasferimenti per porre
            MOV DS, AX;       ; in DS indirizzo del segmento dati
            ….
            PutString Stringa ; chiamata macro per visualizzare Stringa
```

Istruzioni di manipolazione di stringhe

Istruzioni di manipolazione di stringhe: movs e rep

- movs assume che DS:SI punti a una *stringa sorgente* ed ES:DI punti a una *stringa destinazione,* che deve trovarsi nell'extended segment
- copia un byte (o parola) della stringa sorgente nella stringa destinazione incrementando DI ed SI di 1 (se byte) o di 2 (se parola)
- Esempio: copia della stringa ascii_str1 in asci_str2

```
MOV     SI, OFFSET ascii_str1
MOV     DI, OFFSET ascii_str2
MOV     CX, LENGTH ascii_str1     ; pone la lunghezza della
                                  ; stringa in CX
REP     MOVS ES:ascii_str2, ascii_str1
```

- E' necessario specificare il registro ES nella variabile stringa destinazione
- Le variabili ascii_str2 e ascii_str1 servono solo per decidere se l'operazione è di tipo byte o word: l'istruzione è codificata in linguaggio macchina con un solo byte (operandi impliciti sono SI e DI)
- REP permette di ripetere l'esecuzione dell'istruzione MOVS usando CX come contatore. E' usata solo con istruzioni su stringhe

Istruzioni di controllo dello stato della macchina: hlt, nop

- L'istruzione hlt fa terminare l'attività del calcolatore
- Pone l'unità di controllo in attesa di un segnale esterno di tipo reset o un'interruzione esterna
- L'istruzione nop non causa alcuna azione
- nop può essere usata per la sincronizzazione con dispositivi esterni

Istruzioni di input/output

in	***ax/al, porta***
in	***ax/al, dx***
out	***porta, ax/al***
out	***dx, ax/al***

- in e out sono istruzioni speciali per effettuare trasferimenti da e verso i dispositivi
- perché due forme:
  - Sono supportate 65536 differenti porte di I/O (necessari 16 bit)
  - L'operando porta è però di tipo byte, quindi si possono indirizzare direttamente solo 256 porte
  - Per indirizzare tutte le 65536 porte occorre caricare l'indirizzo della porta desiderata nel registro dx e fare accesso alla porta in modo indiretto

Esempi di uso di in e out

- La porta 60h è quella del microcontrollore della tastiera

```
in      al, 60h
out     60h, al
```

- Si possono fare semplici programmi utilizzando l'istruzione out
- Ad esempio per settare i bit dei *led della tastiera* occorre inviare 2 byte sulla porta 60h:
  - Il primo che contiene il valore EDh (o 237)
  - Il secondo che contiene un valore compreso fra 0 e 7, che rappresenta la desiderata configurazione di accensione dei led ScrollLock, NumLock e CapsLock
- Si veda il programma `led.asm` che accende il led CapsLock e spegne i led NumLock e ScrollLock
- L'uso dell'istruzione in richiede la conoscenza di altri dettagli legati all'hardware della tastiera e alla configurazione del sistema