Fondamenti di Informatica II
(AG 0201)

prof. G. Pozzi

12 giugno 2001

Linguaggio AS11X

- Cancellazione di un elemento in un array
- Ricerca di una data sequenza in un file
- Somma e sotrazione tra due interi di tipo long
- Individuazione dell'operato di programma

Cancellazione di un elemento in un array

Si scriva una subroutine (non e' necessario scrivere il codice del programma chiamante) in linguaggio AS11X che si comporti nel seguente modo:

Sia dato nel programma chiamante un vettore VETT, di NEL elementi, ordinato in modo crescente: ogni elemento occupi una parola. Sullo stack, il programma chiamante passa alla subroutine un generico elemento EL: la subroutine cerca tale elemento all'interno di VETT. Se la subroutine trova l'elemento EL passatole, lo elimina da VETT, decrementando di uno NEL e spostando avanti tutti gli elementi di VETT successivi all'elemento trovato, e restituisce al chiamante il valore 1. Se l'elemento EL non e' presente in VETT, la subroutine restituisce al chiamante il valore 0.

E' richiesto l'utilizzo di un algoritmo ottimizzato, in grado cioe' di determinare se un elemento non possa essere presente in VETT senza effettuare il confronto con tutti gli NEL elementi presenti.

E' richiesto di commentare adeguatamente il codice scritto. Per semplicita' si ipotizzi il chiamante del seguente tipo:

```
PILA: .RES 1000
NEL:  .INT xxx
VETT: .RES xxx
```

Soluzione

```
PILA:       .RES 1000              ; pila
NEL:        .INT xxx               ; numero di elementi in VETT
VETT:       .RES xxx               ; VETT: ogni elem. occupa
                                   ; 1 parola


; . . . .
; MAIN: programma chiamante
;

; . . . .
START       MOV #PILA, SP          ; SP punta inizio pila
; . . . .
            MOV #0, (SP)+          ; spazio per il risultato
            MOV EL, (SP)+          ; passo alla sbr. il parametro
            JSR ELIMINA            ; chiamo la sbr
; . . . .
            EXIT$                  ; termine del main


; . . . .
; Subroutine chiamata
;

ELIMINA:                           ; sbr ELIMINA
            MOV R0,(SP)+           ; prologo: salvo R0
            MOV SP, R0             ; RO puntatore locale
```

```
                                ; allo stack

                                ; salvo altri registri
            MOV R1, (SP)+       ; salvo R1
            MOV R2, (SP)+       ; salvo R2
            MOV R3, (SP)+       ; salvo R3

                                ; inizializzo alcuni valori
            MOV #VETT, R1       ; R1 punta a VETT[0]
            MOV #0, R2          ; R2 e' il numero di elem.
                                ; processati. R2 = 0

ANCORA:     CMP R2, NEL         ; termine scansione?
            BEQ FINE0           ; ho scandito tutti gli elem.
                                ; non ho trovato l'elemento
                                ; ricercato ==> restituisco 0
            INC R2              ; numero elem. confrontati

            CMP -3(R0), (R1)+   ; confronto VETT[i] con il
                                ; param. sullo stack
            BGT FINE0           ; VETT e' ordinato: se VETT[i]
                                ; e' magg. di param ==> param
                                ; non potra' essere in VETT
            BNE ANCORA          ; se non ho trovato param,
                                ; continuo la ricerca

                                ' ho trovato in VETT i param.
            MOV R1, R3          ; R3 punta come R1 a VETT[i+1]
            DEC R1              ; R1 punta a VETT[i]
COPIA:      CMP R2, NEL         ; ho copiato tutti gli elem.?
            BEQ DECREM          ; si, devo decrementare NEL

                                ; no, devo continuare la copia
                                ; in VETT
            INC R2              ; R2 e' il numero di elem.
                                ; processati
            MOV (R3)+,(R1)+     ; Copio VETT[i+1] in VETT[i]
            BR COPIA            ; continuo la copia

DECREM:     MOV #1, -4(R0)      ; restituisco 1 al chiamante
            DEC NEL             ; decremento NEL
            BR FINE1

FINE0:      MOV #0, -4(R0)      ; restituisco 0 al chiamante

FINE1:      MOV -(SP), R3       ; ripristino R3
            MOV -(SP), R2       ; ripristino R2
            MOV -(SP), R1       ; ripristino R1
            MOV -(SP), R0       ; ripristino R0
            RTS
```

Ricerca di una data sequenza in un file

Scrivere un progamma AS11X che dato un file in ingresso "FILE_DATI" verifichi se una data sequenza di caratteri e' presente all'interno di un file. Per ogni volta in cui la sequenza viene trovata, il programma genera un messaggio sul video "SEQUENZA TROVATA".

Si deve tener conto che se la stringa da leggere e' di lunghezza superiore alla lunghezza del file, il programma visualizza il messaggio "FINE FILE".

Si noti che la stringa da cercare non e' richiesta a video ma e' inclusa nel programma (come etichetta CERCA_S): la lunghezza di CERCA_S deve essere calcolata dal programma stesso. Per semplicita' si considerino le dichiarazioni di variabili indicate nel seguito.

Si noti che, nel caso in cui la stringa da cercare sia ad esempio ZZ e nel file sia presente la sequenza ZZZ, la stringa verra' indicata come presente due volte.

E' obbligatorio commentare in modo adeguato il codice scritto.

Soluzione

```
CERCA_S:          .INT 02                       ; sequenza da cercare
                  .INT 23                       ; lunghezza max. 100
                  .INT 99
                  .INT 35
                  .INT 87                       ; ultimo carattere
BUF:              .RES 100                      ; buffer per confronto
NOME:             .STRING/FILE_DATI/            ; nome del file in ingresso
                  .INT 0
FDIN:             .RES 1                        ; descrittore del file
LEN:              .RES 1                        ; lunghezza utile di BUF
TROVATO:          .STRING/TROVATA LA SEQUENZA/  ; messaggio di stringa
                  .INT 0                        ; trovata
ERR_LETTURA:      .STRING/FINE FILE/            ; messaggio di improvvisa
                  .INT 0                        ; fine del file

BUF2:             .RES 0                        ; buffer per lettura
RIS:              .RES 1                        ; esito READ$
PILA:             .RES 1000                     ; pila

MAIN:             MOV #BUF, R0                  ; R0 punta a BUF
                  SUB #CERCA_S, R0              ; R0 = numero di elementi
                                                ; da cercare, calcolato
                                                ; come distanza tra BUF e
                                                ; CERCA_S
                  MOV R0, LEN                   ; LEN = numero di elementi
                                                ; da cercare
                  OPEN$ #NOMEIN, FDIN           ; apertura del file
                  READ$ FDIN, #BUF, LEN, RIS    ; lettura delle prime LEN
                                                ; parole
                  TST RIS                       ; c'e' stato errore?
                  BEQ READ_ERR                  ; fine file

ANCORA:           MOV #CERCA_S, R1              ; R1 scandisce CERCA_S
                  MOV #BUF, R2                  ; R2 scandisce BUF
```

```
                MOV #1, R0                      ; contatore

CONFR:          CMP (R1)+, (R2)+                ; R1 e R2 puntano a due
                                                ; parole identiche? Cioe':
                                                ; CERCA_S[R1] = BUF[R2] ?
                BNE LEGGI_ANCORA
                CMP R1, LEN                     ; scandita tutta CERCA_S?
                BEQ TROVATO
                INC R0                          ; confronto la parola seg.
                BR CONFR

LEGGI_ANCORA:   MOV #0, R0                      ; sposta indietro di 1 pos-
                                                ; izione tutte le parole di
                                                ; BUF
                MOV #BUF, R1                    ; R1 punta a BUF[0]
                MOV R1, R2
                INC R2                          ; R2 punta a BUF[1]
SPOSTA:         MOV (R2)+, (R1)+                ; BUF[i] = BUF[i+1]
                INC R0                          ; R0 = n. parole spostate
                CMP R0, LEN                     ; ho ricopiato tutto BUF?
                BNE SPOSTA
                DEC R1                          ; R1 punta all'ultima pos.
                                                ; utile di BUF, dove devo
                                                ; mettere la parola che
                                                ; leggero'

                READ$ FDIN, #BUF2, #1, RIS      ; leggo una nuova parola
                TST RIS
                BEQ FINE
                MOV BUF2, (R1)                  ; e la metto in ultima
                                                ; posizione in BUF
                BR CONFR                        ; riparte il confronto

TROVATO:        WRITE 1, #TROVATO, #<ERR_LETTURA - TROVATO>, RIS
                                                ; ho trovato una stringa
                BR LEGGI_ANCORA                 ; riparte con il confronto

READ_ERR:       WRITE 1, #ERR_LETTURA, #<BUF2 - ERR_LETTURA>, RIS
                                                ; errore di improvvisa fine
                                                ; file

FINE:           CLOSE$ FDIN                     ; chiusura del file
                EXIT$                           ; termine del programma
                .END                            ; fine di MAIN
```

Somma e sottrazione
tra due interi di tipo long

Utilizzando il linguaggio AS11X, e' richiesta l'implementazione di due macro LONG_SUM e LONG_DIFF che utilizzano tre parametri IND_1, IND_2, IND_RIS corrispondenti a tre indirizzi di memoria distinti ed effettuino le seguenti operazioni:

LONG_SUM effettua la somma tra due interi long (che occupano cioe' due parole consecutive ciascuno ed i cui indirizzi sono IND_1 e IND_2) e scrive il risultato in due parole consecutive (il cui indirizzo e' IND_RIS).

LONG_DIFF effettua la differenza tra due interi long (che occupano cioe' due parole consecutive ciascuno ed i cui indirizzi sono IND_1 e IND_2) e scrive il risultato in due parole consecutive (il cui indirizzo e' IND_RIS).

E' necessario gestire i possibili casi di carry ed overflow. E' obbligatorio commentare in modo adeguato il codice scritto.

Soluzione

```
; Si ipotizza che i numeri interi positivi siano memorizzati come:
;   31 ..... 16 15 ..... 0
;   +---------+ +--------+
;   !HIGH-BYTE! !LOW-BYTE!
;   +---------+ +--------+
;
.MACRO LONG_SUM, IND_1, IND_2, IND_RIS

      MOV R0, (SP)+     ; Salvo i registri
      MOV R1, (SP)+     ;
      MOV R2, (SP)+     ;
      MOV R3, (SP)+     ;
      MOV R4, (SP)+     ;
      MOV R5, (SP)+     ;

      MOV #IND_1, R0    ; R0 punta a high-byte del primo operando
      MOV R0, R1
      INC R1            ; R1 punta a low-byte del primo operando
      MOV #IND_2, R2    ; R2 punta a high-byte del secondo operando
      MOV R2, R3
      INC R3            ; R3 punta a low-byte del secondo operando
      MOV #IND_RIS, R4  ; R4 punta a high-byte del risultato
      MOV R4, R5
      INC R5            ; R5 punta a low-byte del risultato

      MOV (R2), (R4)    ; Copio (IND_2) in (IND_RIS)
      MOV (R3), (R5)    ; Copio (IND_2+1) in (IND_RIS+1)

      ADD (R1), (R5)    ; (IND_RIS+1) = (IND_1+1) + (IND_2+1)

      UBLT NO_RIP       ; Se non c'e' carry, non effettuo il riporto
      ADD #1, (R4)      ; Riporto su high-byte del risultato
NO_RIP:ADD (R0), (R4)   ; (IND_RIS) = (IND_1) + (IND_2)

      MOV -(SP), R5     ; Ripristino i registri
      MOV -(SP), R4     ;
      MOV -(SP), R3     ;
```

```
      MOV -(SP), R2     ;
      MOV -(SP), R1     ;
      MOV -(SP), R0     ;
.MEND
;
; I bit di carry e overflow rimangono automaticamente settati in modo
; corretto dopo le operazioni su high-byte.
;
; Termine della macro LONG_SUM
;




.MACRO LONG_DIF, IND_1, IND_2, IND_RIS

      MOV R0, (SP)+     ; Salvo i registri
      MOV R1, (SP)+     ;
      MOV R2, (SP)+     ;
      MOV R3, (SP)+     ;
      MOV R4, (SP)+     ;
      MOV R5, (SP)+     ;

      MOV #IND_1, R0    ; R0 punta a high-byte del primo operando
      MOV R0, R1
      INC R1            ; R1 punta a low-byte del primo operando
      MOV #IND_2, R2    ; R2 punta a high-byte del secondo operando
      MOV R2, R3
      INC R3            ; R3 punta a low-byte del secondo operando
      MOV #IND_RIS, R4  ; R4 punta a high-byte del risultato
      MOV R4, R5
      INC R5            ; R5 punta a low-byte del risultato

      MOV (R2), (R4)    ; Copio (IND_2) in (IND_RIS)
      MOV (R3), (R5)    ; Copio (IND_2+1) in (IND_RIS+1)

      SUB (R1), (R5)    ; (IND_RIS+1) = (IND_1+1) - (IND_2+1)

      UBLT NO_RIP       ; Se non c'e' carry, non effettuo il riporto
      SUB #1, (R4)      ; Riporto su high-byte del risultato
NO_RIP:SUB (R0), (R4)   ; (IND_RIS) = (IND_1) - (IND_2)


      MOV -(SP), R5     ; Ripristino i registri
      MOV -(SP), R4     ;
      MOV -(SP), R3     ;
      MOV -(SP), R2     ;
      MOV -(SP), R1     ;
      MOV -(SP), R0     ;
.MEND
;
; I bit di carry e overflow rimangono settati in modo
; corretto dopo le operazioni su high-byte.
;
; Termine della macro LONG_DIF
;
```

Individuazione dell'operato di un programma

Si descriva l'operato del seguente programma assembler AS11/X e si commenti ogni singola istruzione, indicando il contenuto finale di VETT al termine dell'esecuzione del programma.

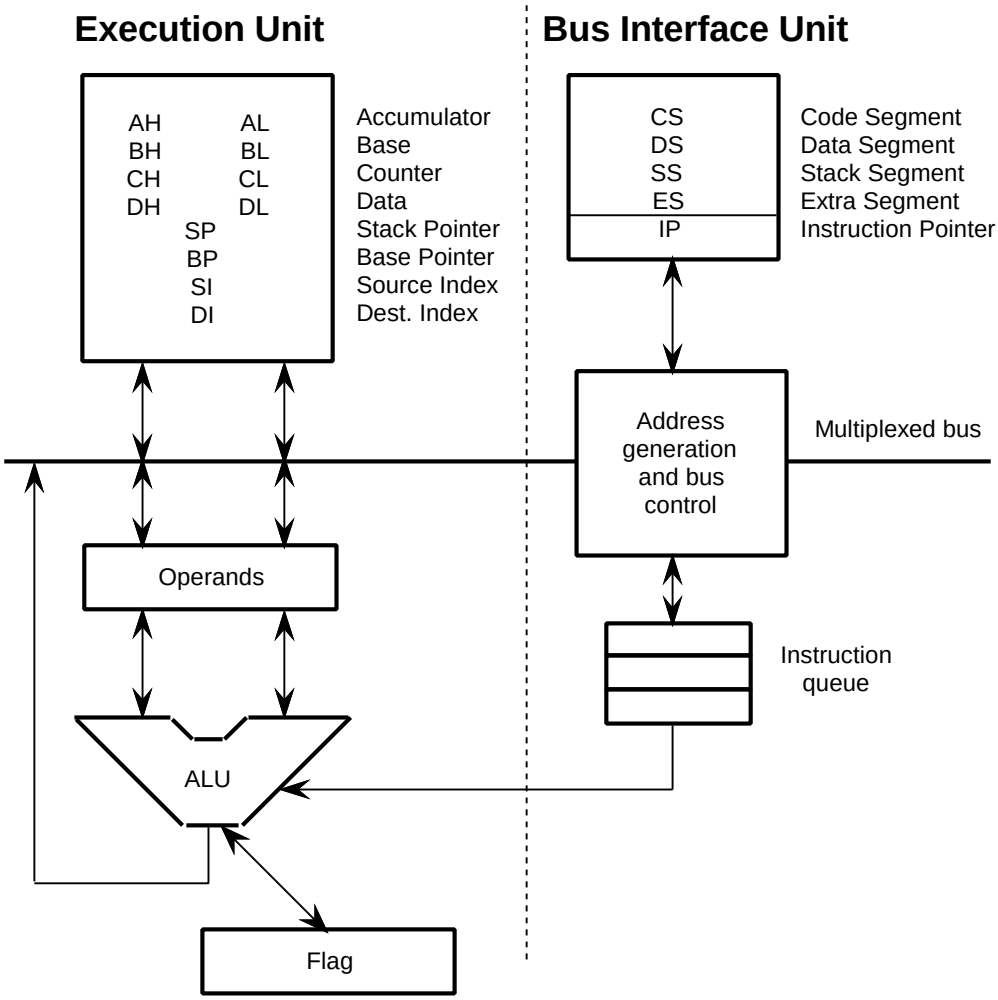
Soluzione

```
; Ordinamento crescente di un vettore con bubble-sort
;
VETT: .INT 6                  ; Inizio del vettore VETT
      .INT 4
      .INT 7
      .INT 3
      .INT 2                  ; Termine del vettore VETT
LUNG: .INT 5                  ; Lunghezza del vettore VETT
TEMP: .RES 1
;
;
START:MOV #VETT, R0           ; R0 punta al primo elem.
      MOV R0, R1
      INC R1                  ; R1 punta al secondo elem.
L1:   CMP (R0), (R1)          ; Inizio confronti
      BLT L3                  ; Sono gia' ordinati
L2:   MOV (R0), TEMP          ; Devo ordinarli: li scambio.
      MOV (R1), (R0)
      MOV TEMP, (R1)
L3:   INC R1                  ; R1 punta all'elemento successivo
      CMP #LUNG, R1           ; R1 sconfina oltre VETT?
      BEQ L4
      BR L1
L4:   INC R0                  ; R0 punta ad un nuovo elemento
      CMP #<LUNG-1>, R0       ; Se R0 sconfina oltre VETT, termino.
      BEQ FINE
      MOV R0, R1
      INC R1                  ; Riparto per il confronto
      BR L1
FINE  EXIT$
      .END START
```

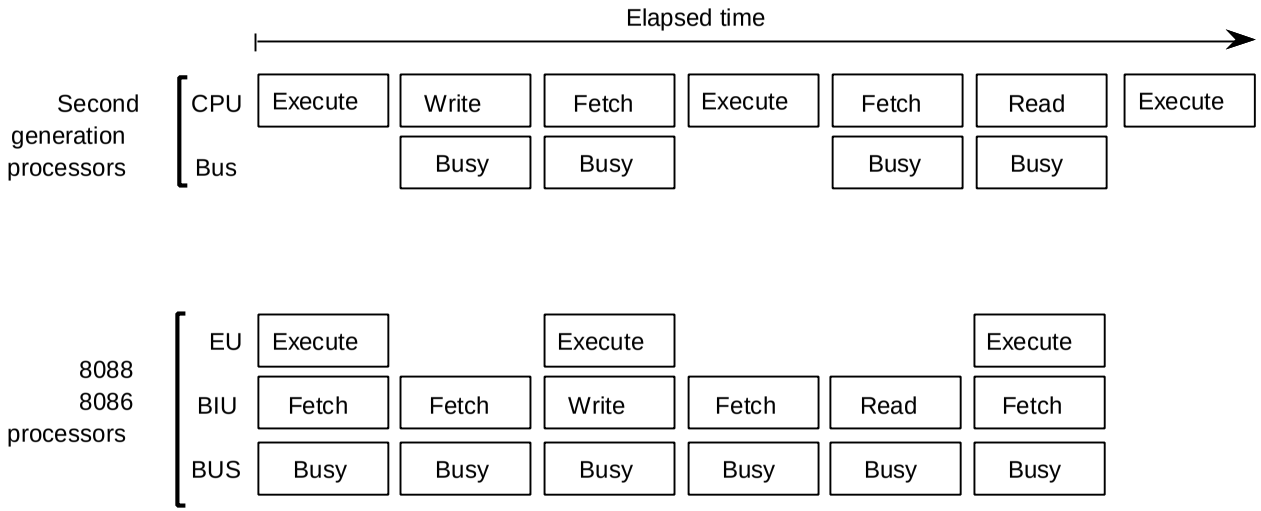
Linguaggio Assembler 8088

- Microprocessori della famiglia 8088
- Organizzazione della memoria
- I registri
- I modi di indirizzamento
- Il set delle istruzioni
    - Istruzioni di trasferimento dei dati
    - Istruzioni di trasferimento degli indirizzi
    - Istruzioni di trasferimento sui flag
    - Istruzioni aritmetiche
    - Istruzioni logiche
    - Istruzioni di scorrimento
    - Istruzioni su stringhe
    - Istruzioni di tarsferimento di controllo
    - Istruzioni di interruzione
    - Istruzione di controllo del processore
    - Istruzione di sincronizzazione esterna
    - Istruzioni di nessuna operazione
- Elenco completo dei codici mnemonici
- Gli interrupt
    - Interrupt 01
    - Interrupt 02
    - Interrupt 05
    - Interrupt 08
    - Interrupt 09
    - Interrupt 0A
    - Interrupt 13
    - Interrupt 16
    - Interrupt 14
    - Interrupt 15
    - Interrupt 17
    - Interrupt 19
    - Interrupt 1A
    - Interrupt 20
    - Interrupt 21

Microprocessori della famiglia 8088



Execution Unit and Bus Interface Unit



1st instruction (already fetched): execute and write results
2nd instruction: execute only
3rd instruction: read operand and execute
4th instruction: undefined

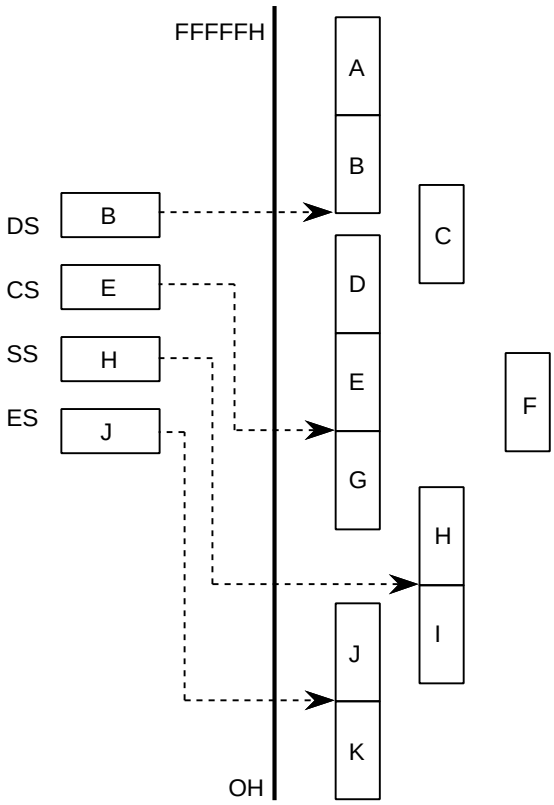
Overlapped Instruction Fetch and Execution

Organizzazione della memoria

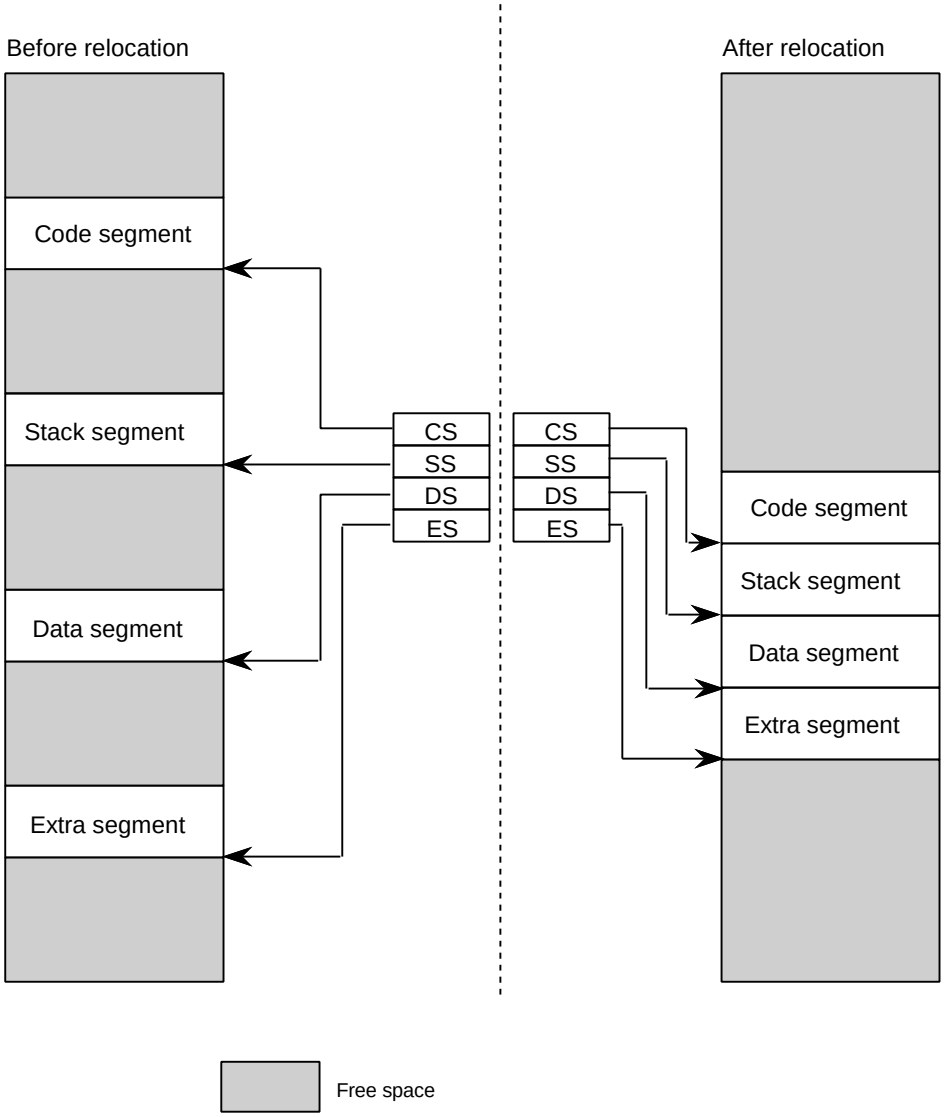
100000H - oltre	memoria estesa PC/AT	
E0000H - FFFFFH	riservata per ROM BIOS	
C0000H - DFFFFH	riservata per ROM installabili	
A0000H - BFFFFH	buffer video	
. . . . . - 09FFFH	porzione transitoria del DOS	RAM di sistema
	area transitoria del programma (progr. e dati dell'utente)	RAM di sistema
	porzione residente del DOS	RAM di sistema
00500H -	area dati per ROM BIOS e BASIC	RAM di sistema
00400H - 004FFH	area dati per ROM BIOS	RAM di sistema
00000H - 003FFH	vettore di interrupt	RAM di sistema

I/O Processor Block Diagram

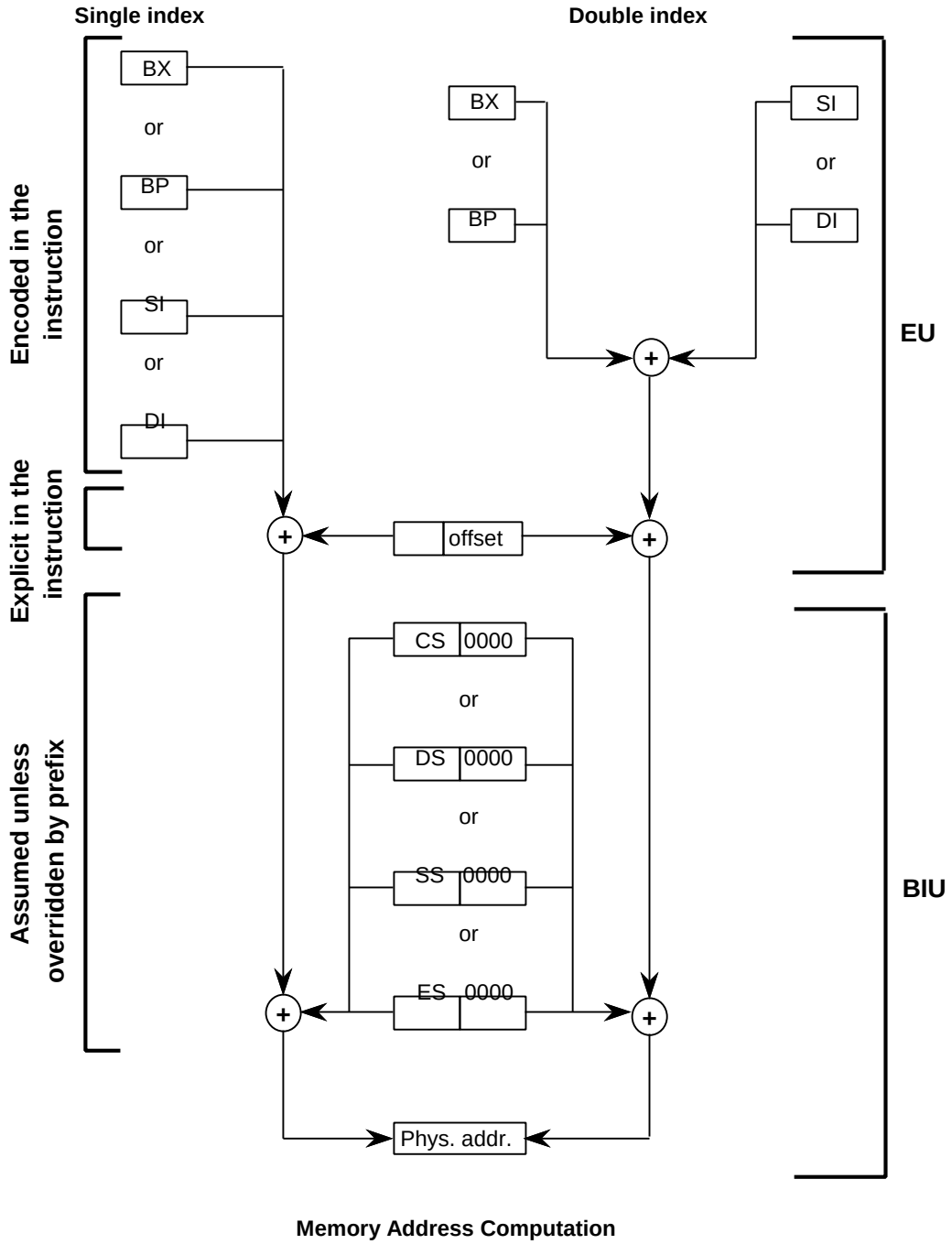
I registri



Currently Addressable Segments



Dynamic code relocation



Memory Address Computation

I modi di indirizzamento

Nome	Indirizzo fisico	Esempi	Descrizione
immediato	incluso nell'istruzione	mov ax, 1234h	metti in AX il valore 1234 esadecimale
diretto	incluso nell'istruzione	mov ax, [1234h]	metti in AX il valore della locazione 1234h, nel segmento dato da DS AX = [DS:1234]

registro indiretto	contenuto in BX, SI, DI o BP	mov ax,[bx]
		mov ax,[si]
		mov ax,[di]
		mov ax,[bp]
			metti in AX il valore della cella il cui offset e' contenuto in BX/SI/DI/BP. Il segmento e' DS per BX, SI: e' ES per DI: e' SS per BP. Cioe':
			AX = [DS:BX]
			AX = [DS:SI]
			AX = [ES:DI]
			AX = [SS:BP]

da base	somma dello scostamento e del valore in BX o BP	mov ax,[bx+2]
		mov ax,2[bx]
		mov ax,[bp+2]
		mov ax,2[bp]
			metti in AX il valore della cella che sta 2 byte oltre l'offset dato da BX/BP. Il segmento e' DS per BX: e' SS per BP.
			Cioe':
			AX = [DS:BX+2]
			AX = [SS:BP+2]

indicizzato	somma dello scostamento e del valore in SI o DI	mov ax,[si+2]
		mov ax,2[si]
		mov ax,[di+2]
		mov ax,2[di]
			metti in AX il valore della cella che sta 2 byte oltre l'offset dato da SI. Il segmento e' DS.
			Cioe':
			AX = [DS:SI+2]
			AX = [ES:DI+2]

indicizzato di base	somma dello scostamento, del valore in SI o DI e del valore in BX o BP	mov ax,[bx+si+2]
		mov ax,2[bx+si]
		mov ax,2[bx][si]
		mov ax,[bx+di+2]
		mov ax,2[bx+di]
		mov ax,2[bx][di]
		mov ax,[bp+si+2]
		mov ax,2[bp+si]
		mov ax,2[bp][si]
		mov ax,[bp+di+2]
		mov ax,2[bp+di]
		mov ax,2[bp][di]
			metti in AX il valore che sta 2 byte oltre l'offset dato dalla somma di SI e di BX/BP. Il segmento e' DS se si usa BX; e' SS se si usa BP.
			Cioe':
			AX = [DS:BX+SI+2]
			AX = [DS:BX+DI+2]
			AX = [SS:BP+SI+2]
			AX = [SS:BP+DI+2]
			(utile per puntare ad un elemento di un array posto sullo stack)

indirizzamento stringa	stringa di origine in modo indiretto con SI. stringa destinazione in modo indiretto con DI	movsb	copia la stringa partendo da [DS:SI] a [ES:DI]

Il set delle istruzioni

Istruzioni: si dividono in istruzioni per trasferimento di dati, trasferimento di indirizzi, di trasferimenti sui flag, aritmetiche (somma, sottrazione, moltiplicazione, divisione), logiche, di scorrimento, su stringhe, trasferimento controllo, di interruzione, controllo processore, di sincronizzazione esterna, di nessuna operazione. Alcune di esse sono:

Istruzioni di trasferimento dei dati:

MOV dest, src: sposta un byte o una parola da src a dest;

```
Es:   MOV AX,BX          ; copio in AX il valore di BX
                         ; (cicli 2)
      MOV AX,FFH         ; copio in AX il numero FF = 255
                         ; (cicli 10)
      MOV AX,[E00H]      ; copio in AX il valore della cella
                         ; di indirizzo E00h del segm. DS
                         ; (cicli 10). Copio in AX 16 bit,
                         ; quindi due byte (E00 e' low-byte,
                         ; E01 e' high byte.
```

PUSH src: sposta la parola data da src sullo stack, e decrementa di 2 SP;

```
Es:   PUSH AX            ; metto AX sullo stack
```

POP dest: richiama una parola dallo stack, la mette in dest e incrementa di 2 SP;

```
Es:   POP AX             ; richiamo AX dallo stack
```

XCHG dest, src: scambia tra loro il byte/parola di dest con il byte/parola di src;

```
Es:   XCHG AX, BX        ; scambio AX con BX
```

XLAT (table): sostituisce il byte in AL con un byte preso da una tabella di 256 byte, definita dall'utente e memorizzata in table. E' usato ad es. per tradurre da ASCII a EBCDIC;

```
Es:   MOV AL,byte_to_xlate   ; byte da tradurre
      LEA BX,table           ; inizio della tabella
      XLAT table             ; traduce AL nella nuova tabella
      XLAT             ; identico al precedente
```

IN acc, port: trasferisce nell'accumulatore un byte/parola (in AL/AX rispettivamente) dalla porta port;

```
Es:   IN AL,0FFEAH       ; copia in AL un byte dalla porta
                         ; di indirizzo 0FFEAh
```

Es: IN AX,0FFEAH ; copia in AX una parola dalla porta
; di indirizzo 0FFEAh

OUT port, acc: trasferisce nella porta port il valore dell'accumulatore di un byte/parola (in AL/AX rispettivamente).

Es: OUT 0FFEAH,AL ; copia un byte sulla porta
; di indirizzo 0FFEAh da AL

Es: OUT 0FFEAH,AX ; copia una parola sulla porta
; di indirizzo 0FFEAh da AX

Istruzioni di trasferimento di indirizzi:

LEA dest, src: trasferisce l'offset di src in dest. src deve essere un operando di memoria, dest deve essere un registro a 16 bit;

Es: LEA BX,data_label ; carica in BX l'indirizzo
; di data_label in DS

LEA BX,CS:code_ms ; carica in BX l'indirizzo
; di code_ms in CS

LDS dest, src: trasferisce una variabile puntatore a 32 bit (16 bit per offset + 16 bit per segmento: deve essere un operando di memoria) all'operando destinazione ed al registro DS;

Es: LDS BX,label ; carica in BX l'indirizzo
; di label. Carica in DS il segmento
; di label

LES dest, src: trasferisce una variabile puntatore a 32 bit (16 bit per offset + 16 bit per segmento: deve essere un operando di memoria) all'operando destinazione ed al registro ES;

Es: LES BX,label ; carica in BX l'indirizzo
; di label. Carica in ES il segmento
; di label

Istruzioni di trasferimento sui flag:

LAHF: carica in AH i flag (bit 7,6,4,2,0);

SAHF: trasferisce i flag (bit 7,6,4,2,0) da AH;

PUSHF: fa PUSH di tutti i flag (1 parola);

POPF: fa POP di tutti i flag (1 parola).

Istruzioni aritmetiche (somma, sottrazione, moltiplicazione, divisione):

ADD dest, src: incrementa dest (byte/parola) del valore di src (byte/parola);

Es: ADD AX,23H ; AX = AX + 23h

ADDC dest, src: incrementa dest (byte/parola) del valore di src (byte/parola). Se Carry = 1, aggiunge 1 alla somma: utile per operazioni su numeri con piu' di 16 bit;

INC dest: incrementa di 1 dest (byte/parola);

```
Es:   INC AX              ; AX = AX + 1h
```

AAA: trasforma il contenuto di AL in un numero decimale unpacked valido. L'high byte di AL e' forzato a 0;

```
Es:   AAA                 ; senza operandi
```

DAA: dopo una somma di due numeri decimali packed, corregge il risultato packed che e' in AL;

```
Es:   DAA                 ; senza operandi
```

SUB dest,src: decrementa dest (byte/parola) del valore di src (byte/parola);

```
Es:   SUB AX,BX           ; AX = AX - BX
      SUB AX,12H          ; AX = AX - 12h
```

SBB dest,src: decrementa dest (byte/parola) del valore di src (byte/parola). Se Carry = 1, sottrae ulteriormente 1: utile per operazioni su numeri con piu' di 16 bit;

DEC dest: decrementa di 1 dest (byte/parola);

```
Es:   DEC AX              ; AX = AX - 1h
```

NEG dest: effettua il complemento a due di dest (byte/parola). Non funziona con -128 e -32768, lasciandoli inalterati;

```
Es:   NEG AX              ; AX = -AX
```

CMP dest,src: confronta dest (byte/parola) con src (byte/parola). In realta' setta i flag come se descrementasse descr del valore di src, lasciandoli pero' dest e src inalterati;

```
Es:   CMP AX,BX           ; RisX = AX - BX
```

AAS: dopo una sottrazione tra unpacked, corregge il risultato unpacked che e' in AL;

```
Es:   AAS                 ; senza operandi
```

DAS: dopo una sottrazione tra packed, corregge il risultato packed che e' in AL;

```
Es:   DAS                 ; senza operandi
```

MUL src: effetua la moltiplicazione senza segno di AL per src (byte): il risultato e' in AX. Se src e' una parola, effettua la moltiplicazione senza segno di AX: il risultato e' in DX ed AX;

```
Es:   MUL BL              ; AX = AL * BL
      MUL CX              ; DX|AX = AX * CX
```

IMUL src: effetua la moltiplicazione con segno di AL per src (byte): il risultato e' in AX. Se src e' una parola, effettua la moltiplicazione con segno di AX: il risultato e' in DX ed AX;

```
Es:   IMUL BL          ; AX = AL * BL
      IMUL CX          ; DX|AX = AX * CX
```

AAM: dopo una moltiplicazione tra due numeri unpacked, trasforma il risultato in un numero decimale unpacked valido, restituito in AH ed AL;

```
Es:   AAM              ; senza operandi
```

DIV src: effetua la divisione senza segno di AX su src (byte): il risultato e' in AL, il resto in AH. Se src e' una parola, effettua la divisione senza segno di AX e DX (doppia lunghezza, 64 bit) su src: il risultato e' in AX, il resto in DX. Si puo' avere interrupt 0 (divide by zero) o per supero capacita' o per divisione per 0. Tronca a intero i quozienti;

```
Es:   DIV BL           ; AL = AX / BL
                       ; AH = resto
      DIV CX           ; AX = DX|AX / CX
                       ; DX = resto
```

IDIV src: effetua la divisione con segno di AX su src (byte): il risultato e' in AL, il resto in AH. Se src e' una parola, effettua la divisione con segno di AX e DX (doppia lungehzza, 64 bit) su src: il risultato e' in AX, il resto in DX. Si puo' avere interrupt 0 (divide by zero) o per supero capacita' o per divisione per 0. Tronca a intero i quozienti;

```
Es:   IDIV BL          ; AL = AX / BL
                       ; AH = resto
      IDIV CX          ; AX = DX|AX / CX
                       ; DX = resto
```

AAD: modifica il numeratore in AL prima di dividere due operandi decimali unpacked, cosi' che il quoziente prodotto dalla divisione sia un numero decimale unpacked valido;

```
Es:   AAD              ; senza operandi
```

CBW: converte un byte dato in AL in una parola in AX. Mantiene il segno!!;

```
Es:   CBW              ; senza operandi
```

CWD: converte una parola data in AX in una parola doppia in AX e DX. Mantiene il segno!!;

```
Es:   CWD              ; senza operandi
```

Istruzioni logiche:

NOT dest; AND dest,src; OR dest,src; XOR dest,src: operatori logici bit-oriented;

```
Es:     NOT AL                  ; se AL = 01101100,
                                ; ==> AL = 10010011
        AND AL,BL       ; se AL = 01101100
                                ; e   BL = 11000101
                                ;==>AL = 01000100
```

TEST dest,src: effetua l'and logico bit a bit tra dest (byte/parola) e src (byte/parola), lasciando inalterati sia dest che src. Se dopo TEST c'e' JNZ, il salto viene effettuato se c'e' almeno un 1 corrispondente;

```
Es:     TEST AL,BL      ; setta i flag come AND ma NON
                        ; altera AL.
```

Istruzioni di scorrimento:

Shift aritmetici per moltiplicare/dividere, logici per isolare sequenze di bit. CL deve essere usato se il numero di shift e' variabile o diverso da 1.

SHL/SAL dest,count: dest (byte/parola) e' spostato a sin. di count bit. I bit a destra sono forzati a 0;

```
Es:     SHL AL,1                ; sposta a sinistra di 1 bit
```

SHR dest,count: dest (byte/parola) e' spostato a dx. di count bit. I bit a sin. sono forzati a 0. Puo' cambiare il segno!!;

```
Es:     SHR AL,1                ; sposta a destra di 1 bit
```

SAR dest,count: dest (byte/parola) e' spostato a dx. di count bit. I bit a sin. sono forzati a 0. Mantiene il segno!!;

```
Es:     SAR AL,1                ; sposta a destra di 1 bit
```

ROL dest,count: dest (byte/parola) e' ruotato a sin. di count bit. Puo' cambiare segno!!;

```
Es:     ROL BX,1                ; ruota a sinistra di 1 bit
```

ROR dest,count: dest (byte/parola) e' ruotato a dx. di count bit. Puo' cambiare segno!!;

```
Es:     ROR BX,1                ; ruota a destra di 1 bit
```

RCL dest,count: dest (byte/parola) e' ruotato a sin. di count bit includendo il carry. Carry e' considerato come il bit di ordine piu' basso;

```
Es:     RCL BX,1                ; ruota a sinistra di 1 bit
```

RCR dest,count: dest (byte/parola) e' ruotato a dx. di count bit includendo il carry. Carry e' considerato come il bit di ordine piu' basso.

```
Es:     RCL BX,1                ; ruota a destra di 1 bit
```

Istruzioni su stringhe: per le operazioni sulle stringhe vengono utilizzati i seguenti registri: SI (indice/offset per stringa sorgente, assunta in DS); DI (indice/offset per la stringa destinazione, assunta in ES); CX (contatore); AL/AX (valore di scansione. Destinazione per LODS, sorgente per STOS); DF (direzione, 0

autoincrementa SI e DI, 1 autodecrementa SI e DI); ZF (terminatore di scansione/confronto);

REP/REPE/REPZ/REPNE/REPNZ: ripeti.....finche' non sei in fine alla stringa (CX = 0). Ad ogni istruzione decrementa (incrementa) CX di 1 a seconda del flag di direzione;

MOVS dest_string,src_string: trasferisce un byte/parola da src_string (in SI) a dest_string (in DI). SI e DI puntano poi all'elemento successivo;

```
Es:   LDS SI,src_string          ; carica in SI l'indirizzo dell'inizio
                                 ; della stringa sorgente ed in DS il
                                 ; segmento dove c'e' src_string
      LES DI,dest_string         ; carica in DI l'indirizzo dell'inizio
                                 ; della stringa destinazione ed in
      DS
                                 ; il segmento dove c'e' dest_string
      MOV CX,20          ; CX = lunghezza stringa da copiare
      MOVS DI,SI         ; copia un elemento della stringa.
                                 ; Incrementa/decrementa SI e DI.
```

MOVSB, MOVSW: muove una stringa di byte/parole;

```
Es:   REP MOVSB          ; copia una stringa di byte.
                                 ; Continua a copiare finche' non si
                                 ; arriva a CX = 0.
      REP MOVSW                  ; copia una stringa di parole.
                                 ; Continua a copiare finche' non si
                                 ; arriva a CX = 0.
```

CMPS dest_string,src_string: sottrae a dest (byte/parola, indirizzati da DI) il valore di src_string (byte/parola, indirizzati da SI). Non modifica dest_string e src_string: SI e DI puntano poi all'elemento successivo;

SCAS dest_string: sottrae all'elemento (byte/parola) della stringa destinazione indirizzato da DI il valore di AL (byte) o AX (parola). Puo' essere utile per cercare un valore all'interno di una stringa;

LODS src_string: trasferisce il byte/parola come elemento di una stringa indirizzata da SI, in AL/AX;

```
Es:   LEA SI,src_string
      STOS SI                    ; copia in AL (AX) un byte
      (parola)
                                 ; della stringa src_string. SI e'
                                 ; automaticamente incrementato
```

STOS dest_string: trasferisce AL/AX in un byte/parola come elemento di una stringa indirizzata da DI. Utile, ad es., per inizializzare una stringa;

```
Es:   LEA DI,src_string
      LODS DI                ; copia da AL (AX) un byte
      (parola)
                             ; nella stringa dest_string. DI e'
                             ; automaticamente incrementato
```

Istruzioni di trasferimento di controllo:

CALL proc_name: salto incondizionato ad una procedura proc_name;

```
Es:   CALL pippo             ; chiama la procedura pippo
```

RET: ritorna dalla procedura al chiamante;

```
Es:   RET                    ; ritorna al chiamante
```

JMP target: salta incondizionatamente ad una etichetta target. Puo' essere SHORT JMP (-128÷+127), NEAR JMP (-327688÷+32767);

LOOP/LOOPE/LOOPZ short_label: decrementa CX e se diverso da 0 salta a short_label;

LOOPNE/LOOPNZ short_label: decrementa CX e se uguale a 0 salta a short_label;

REP/REPE/REPZ/REPNE/REPNZ: ripeti.....finche' non sei in fine alla stringa (CX = 0) (vedi sopra). Automaticamente decrementa di 1 CX;

JCXZ short_label: salta se CX e' zero;

Istruzioni di interruzione:

INT int_type: attiva la procedura di interrupt data da int_type;

```
Es:   INT xxH                ; chiama l'interrupt xxh
```

INTO: genera un interrupt software se c'e' overflow;

```
Es:   INTO                   ; chiama l'interrupt di overflow
```

IRET: trasferisce il controllo a chi ha chiamato una procedura di interrupt.

```
Es:   IRET                   ; ritorna da interrupt
```

Istruzioni di controllo del processore:

CLC: azzera il flag di carry;

```
Es:   CLC                    ; senza operandi
```

CMC: complementa il flag di carry;

STC: mette a 1 il flag di carry;

CLD: azzera il flag di direzione;

STD: mette a 1 il flag di direzione;

CLI: azzera il flag di abilitazione interrupt. Interrupt mascherabili sono disabilitati;

STI: mette a 1 il flag di abilitazione interrupt;

Istruzioni di sincronizzazione esterna:

HLT: ferma il processore;

```
Es:   HLT                    ; ferma la CPU
```

WAIT: attende finche' la linea TEST non e' attiva;

```
Es:     WAIT                ; la CPU aspetta che TEST sia attiva
```

ESC ext_opcode, src: un processore esterno puo' ottenenere un opcode dalla CPU;

```
Es:     ESC 20,AL           ; un processore esterno riceve un
                            ; codice operativo a 6 bit
```

LOCK: blocca il bus per 1 ciclo.

```
Es:     LOCK                ; senza operandi
```

Istruzioni di nessuna operazione:

NOP: non fa nulla, per 1 ciclo.

Elenco completo dei codici mnemonici

Mnem.	Nome completo	Operazione
AAA	ASCII Adjust after Addition	Aggiustamento ASCII dopo somma
AAD	ASCII Adjust after Division	Aggiustamento ASCII dopo divisione
AAM	ASCII Adjust after Multiplication	Aggiustamento ASCII dopo prodotto
AAS	ASCII Adjust after Subtraction	Aggiustamento ASCII dopo sottrazione
ADC	ADd with Carry	Somma col valore del carry
ADD	ADD	Somma
AND	AND	AND logico dei bit
CALL	CALL	Chiamata a subroutine
CBW	Convert Byte to Word	Converti un byte in una parola
CLC	CLear Carry flag	Azzera il flag di Carry
CLD	CLear Direction flag	Azzera il flag di Direzione
CLI	CLear Interrupt flag	Azzera il flag Interrupt
CMC	CoMplement Carry flag	Complementa il flag di Carry
CMP	CoMPare	Confronta
CMPS	CoMPare String	Confronta le stringhe
CMPSB	CoMPare String - Byte	Confronta le stringhe - byte
CMPSW	CoMPare String - Word	Confronta le stringhe - parole
CWD	Convert Word to Doubleword	Conversione da parola a parola doppia
DAA	Decimal Adjust after Addition	Aggiustamento decimale dopo somma
DAS	Decimal Adjust after Subtraction	Aggiustamento decimale dopo sottrazione
DEC	DECrement	Decrementa di 1
DIV	unsigned DIVide	Divisione senza segno
ESC	ESCape	Cede controllo ad altra CPU
HLT	HaLT	Ferma l'esecuzione
IDIV	Integer DIVide	Divisione intera
IMUL	Integer MULtiply	Moltiplicazione intera
IN	INput from I/O port	Input da porta di I/O
INC	INCrement	Incementa di 1
INT	INTerrupt	Chiamata di interrupt
INTO	INTerrupt on Overflow	Interrupt se overflow
IRET	Interrupt RETurn	Ritorno da interruzione
JA	Jump if Above	Salta se maggiore
JAE	Jump if Above or Equal	Salta se maggiore o uguale
JB	Jump if Below	Salta se inferiore
JBE	Jump if Below or Equal	Salta se inferiore o uguale
JC	Jump if Carry	Salta se Carry = 1
JCXZ	Jump if CX is Zero	Salta se CX = 0
JE	Jump if Equal	Salta se eguale
JG	Jump if Greater than	Salta se maggiore
JGE	Jump if Greater Equal	Salta se maggiore o uguale
JL	Jump if Less	Salta se inferiore
JLE	Jump if Less Equal	Salta se inferiore o uguale

JMP	JuMP	Salto incodizionato
JNA	Jump if Not Above	Salta se non maggiore
JNAE	Jump if Not Above or Equal	Salta se non maggiore o uguale
JNB	Jump i Not Below	Salta se non inferiore
JNBE	Jump if Not Below or Equal	Salta se non inferiore o uguale
JNC	Jump if Not Carry	Salta se Carry = 0
JNE	Jump if Not Equal	Salta se non uguale
JNG	Jump if Not Greater than	Salta se non maggiore
JNGE	Jump if Not Greater than or Equal	Salta se non maggiore o uguale
JNL	Jump if Not Less than	Salta se non minore
JNLE	Jump if Nor Less than or Equal	Salta se non minore o uguale
JNO	Jump if Not Overflow	Salta se Overflow = 0
JNP	Jump if Not Parity	Salta se flag Parita' = 0
JNS	Jump if Not Sign	Salta se flag Segno = 0
JNZ	Jump if Not Zero	Salta se flag Zero = 0
JO	Jump on Overflow	Salta se Overflow = 1
JP	Jump on Parity	Salta se flag Parita' = 1
JPE	Jump if Parity Even	Salta se Parita' pari
JPO	Jump if Parity Odd	Salta se Parita' dispari
JS	Jump if Sign	Salta se flag Segno = 1
JZ	Jump if Zero	Salta se flag Zero = 1
LAHF	Load AH with Flags	Carica i Flag in AH
LDS	Load pointer using DS	Carica DS nel puntatore
LEA	Load Effective Address	Carica l'indirizzo effettivo
LES	Load pointer using ES	Carica ES nel puntatore
LOCK	LOCK bus	Blocca il bus per 1 cilco di istruzioni
LODS	LOaD String	Carica la stringa
LODSB	LOaD String - Byte	Carica la stringa - byte
LODSW	LOaD String - Word	Carica la stringa - parola
LOOP	LOOP	Salta all'etichetta
LOOPE	LOOP while Equal	Torna in ciclo finche' uguale
LOOPNE	LOOP while Not Equal	Torna in ciclo finche' non uguale
LOOPNZ	LOOP while Not Zero	Torna in ciclo finche' non zero
LOOPZ	LOOP while Zero	Torna in ciclo finche' zerp
MOV	MOVe data	Carica dati
MOVS	MOVe String	Carica la stringa
MOVSB	MOVe String - Byte	Carica la stringa - byte
MOVSW	MOVe String - Word	Carica la stringa - parola
MUL	MULtiply	Moltiplica
NEG	NEGate	Inverte di segno
NOP	No OPeration	Nessuna operazione
NOT	NOT	Operatore logico
OR	OR	Operatore logico
OUT	OUTput to I/O port	Uscita su porta I/O
POP	POP	Scarica dallo stack
POPF	POP Flag	Scarica dallo stack i flag
PUSH	PUSH	Carica sullo stack
PUSHF	PUSH Flag	Carica sullo stack i flag

RCL	Rotate through Carry Left	Ruota a sinistra tramite il Carry
RCR	Rotate through Carry Right	Ruota a destra tramite il Carry
REP	REPeat	Ripeti
REPE	REPeat while Equal	Ripeti finche' uguale
REPNE	REPeat while Not Equal	Ripeti finche' diverso
REPNZ	REPeat while Not Zero	Ripeti' finche' diverso da 0
REPZ	REPeat while Zero	Ripeti finche' uguale a 0
RET	RETurn	Rientra dalla subroutine
ROL	ROtate Left	Ruota a sinistra
ROR	ROtate Right	Ruota a destra
SAHF	Store AH into Flags	Salva AH nei flag
SAL	Shift Arithmetic Left	Scorrimento aritmetico a sinistra
SAR	Shift Arithmetic Right	Scorrimento aritmetico a destra
SBB	SuBtract with Borrow	Sottrai con riporto
SCAS	SCAn String	Scansione della stringa
SCASB	SCAn String - Byte	Scansione della stringa - byte
SCASW	SCAn String - Word	Scansione della stringa - parola
SHL	SHift Left	Spostamento a sinistra
SHR	SHift Right	Spostamento a destra
STC	SeT Carry flag	Poni Carry = 1
STD	SeT Direction flag	Poni Direction = 1
STI	SeT Interrupt flag	Poni Interrupt = 1
STOS	STOre String	Memorizza stringa
STOSB	STOre String - byte	Memorizza stringa - byte
STOSW	STOre String - word	Memorizza stringa - parola
SUB	SUBtract	Sottrai
TEST	TEST	AND bit a bit dei due operandi
WAIT	WAIT	Attesa che il pin TEST venga negato
XCHG	eXCHanGe	Scambia
XLAT	transLATe	Converti
XOR	eXclusive OR	Operatore logico

Gli interrupt

Gli interrupt si dividono in: interrupt del microprocessore, interrupt hardware, interrupt software, interrupt del DOS, interrupt del Basic, interrupt di uso generale.

Interrupt del microprocessore: sono anche detti interrupt logici, e sono generati direttamente dal microprocessore o da un altro processore (8087). Interrupt 00h÷04h.

Interrupt hardware: sono generati dall'hardware. Interrupt 08÷0Fh.

Interrupt software: incorporati nel PC, fanno parte dei programmi ROM del BIOS (Basic Input-Output System). Interrupt 10h÷1Fh e 40h÷5Fh.

Interrupt del DOS: sono disponibili ogni qual volta venga usato il DOS. Interrupt 20h÷3Fh.

Interrupt Basic: sono generati dall'interprete Basic. Interrupt 80h÷F0h.

Interrupt di uso generale: sono disponibili per l'uso generale da parte dei programmi dell'utente. Interrupt 60h÷66h.

Alcuni interrupt riguardano:

Int.	Impiego
0h	Generato dalla CPU quando si tenta una divisione per 0
1h	Inpiegato per esecuzione passo-passo (DEBUG in modo trace)
2h	NMI
3h	Impiegato per fissare punti di arresto nell'esecuzione del programma DEBUG
4h	Generato quando si ha overflow aritmetico
5h	Attiva routine ROM BIOS di copia dello schermo su carta (printscreen)
8h	Generato ad ogni impulso dell'orologia hardware
9h	Generato da una attivita' di tastiera
Eh	Segnala richiesta di attenzione da parte del disco (es. termine operazione)
Fh	Usato per controllo stampante
10h	Richiamo di servizi video del ROM BIOS
11h	Richiamo servizio elenco dispositivi del ROM BIOS
12h	Richiamo servizio quantita' di memoria del ROM BIOS
13h	Richiamo servizi disco del ROM BIOS
14h	Richiamo servizi comunicazione del ROM BIOS
15h	Richiamo servizi di sistema del ROM BIOS
16h	Richiamo servizi standard per la tastiera del ROM BIOS
17h	Richiamo servizi per la stampante del ROM BIOS
18h	Attiva il Basic della ROM
19h	Richiama le routine di bootstrap del ROM BIOS
1Ah	Richiamo servizi data e ora del ROM BIOS
1Bh	Interrupt del ROM BIOS su pressione ctrl-break
1Ch	Interrupt generato ad ogni impulso di clock
1Dh	Punta alla tabella parametri di controllo del video
1Eh	Punta alla tabella parametri dei dischetti
1Fh	Punta ai caratteri grafici del modo video CGA
20h	Richiama servizio completamento programma del DOS

21h	Richiama tutte le funzioni del DOS
22h	Indirizzo della routine di termine del programma
23h	Indirizzo della routine DOS di gestione di un Break da tastiera
24h	Indirizzo della routine DOS di gestione errori critici (Abort, Retry, Ignore)
25h	Richiama servizio di lettura assoluta su disco del DOS
26h	Richiama servizio di scrittura assoluta su disco del DOS
27h	Termine del programma ma lo mantiene in memoria sotto DOS
28h	Interrupt stato di attesa (quando il DOS aspetta con int 21h, passa da qui)
2Fh	Interrupt Multiplex del DOS
41h	Punta alla tabella parametri del disco fisso
43h	Punta ai caratteri grafici del video (EGA, PS/2)
67h	Attiva gestione memoria espansa LIM

Interrupt 10h: racchiude tutti i servizi per lo schermo del ROM BIOS. Vengono chiamati mettendo in AH il valore del servizio desiderato, e poi chiamando l'interrupt.

Servizio 00h: determina il modo video. Il parametro AL contiene il modo: se si aggiunge 80h, non viene cancellato lo schermo. (Nella tabella 00 e 01 sono del tutto identici, cosi' anche 02 e 03.. La differenza e' che alcuni usano monocromatico some emulaz. colore)

Modo	**Tipo**	**Risoluz.**	**Colori**	**Sottosistema video**
00h, 01h	Testo	40x25	16	CGA, EGA, MCGA, VGA
02h, 03h	Testo	80x25	16	CGA, EGA, MCGA, VGA
04h, 05h	Grafica	320x200	4	CGA, EGA, MCGA, VGA
06h	Grafica	640x200	2	CGA, EGA, MCGA, VGA
07h	Testo	80x25	Mono	MDA, EGA, VGA
08h, 09h, 0Ah				solo su PC jr
0Bh, 0Ch				uso interno BIOS EGA
0Dh	Grafica	320x200	16	EGA, VGA
0Eh	Grafica	640x350	Mono	EGA, VGA
10h	Grafica	640x350	16	EGA, VGA
11h	Grafica	640x480	2	MCGA, VGA
12h	Grafica	640x480	16	VGA
13h	Grafica	320x200	256	MCGA, VGA

Servizio 02h: fissa la posizione del cursore data dai parametri DH (numero di riga), DL (numero di colonna), BH (numero di pagina). Il punto 0,0 e' nell'angolo alto a sinistra;

Servizio 05h: determinazione pagina video attiva: il parametro AL contiene la pagina video da rendere attiva (di solito 0, sono al max. 7);

Servizio 08h: lettura di un carattere presente sullo schermo in corrispondenza del cursore e dei suoi attributi. Il parametro BH indica il numero di pagina video

attiva: AL restituisce il codice ASCII del carattere, AH i suoi attributi. Gli attributi sono:

bit 7 = blinking del carattere o intensita' colore sfondo;

bit 6 = componente colore R pe sfondo;

bit 5 = componente colore G per sfondo;

bit 4 = componente colore B per sfondo;

bit 3 = intensita' colore primo piano;

bit 2 = componente colore R per primo piano;

bit 1 = componente colore G per primo piano;

bit 0 = componente colore B per primo piano;

Servizio 09h: scritture di un carattere in corrispondenza del cursore e dei suoi attributi. Il parametro AL contiene il codice ASCII del carattere, BL l'attributo, BH il numero di pagina attiva, CX numero di ripetizioni del carattere.

Servizio 0Ah: scrittura di un carattere in corrispondenza del cursore. Il parametro AL contiene il codice ASCII del carattere, BL l'attributo (per modo grafico), BH il numero di pagina attiva, CX numero di ripetizioni del carattere.

Servizio 15h: restituisce il modo video corrente: AL restitusice il modo video, AH restituisce il numero di caratteri per riga.

Interrupt 13h: racchiude tutti i servizi per i dischi del ROM BIOS. Vengono chiamati mettendo in AH il valore del servizio desiderato, e poi chiamando l'interrupt.

Servizio 02h: lettura di uno o piu' settori di un disco. I parametri sono:

DL = numero del drive;

DH = numero della testina;

CH = numero del cilindro (per hard-disk)

CL = numero di settore (per hard-disk)

AL = numero di settori che si vogliono leggere

ES:BX = indirizzo del buffer

I risultati sono:

CF = 0, nessun errore e AH = 0

CF = 1, AH indica l'errore;

Servizio 03h: scrittura di uno o piu' settori di un disco. I parametri ed i risultati sono identici a quelli del servizio 02h;

Interrupt 16h: racchiude tutti i servizi per la tastiera del ROM BIOS. Vengono chiamati mettendo in AH il valore del servizio desiderato, e poi chiamando l'interrupt.

Servizio 00h: lettura di un carattere da tastiera: rimane in attesa finche' non e' stato premuto un tasto. I risultati sono:

AL = codice ASCII del tasto premuto. Se e' stato premuto un tasto speciale,

AL = 0 ed AH = identificatore tasto speciale;

Servizio 01h: determina se e' pronto un carattere. I risultati sono;

ZF = 1, non ci sono dati pronti;

ZF = 0, ci sono dati pronti e sono restituiti in AL e AH (come servizio 00);

Servizio 02h: determina lo stato di shift. I risultati sono:

AL = bits stato della tastiera;

bit 7 = insert on

bit 6 = caps lock on

bit 5 = num lock on

bit 4 = scroll lock on

bit 3 = alt premuto

bit 2 = ctrl premuto

bit 1 = shift sinistro premuto

bit 0 = shift destro premuto

Servizio 12h: determina lo stato di shift su tastiera estesa. I risultati sono:

AH = bits stato della tastiera estesa;

bit 7 = sys req

bit 6 = caps lock on

bit 5 = num lock on

bit 4 = scroll lock on

bit 3 = alt destro premuto

bit 2 = ctrl destro premuto

bit 1 = alt sinistro premuto

bit 0 = ctrl sinistro premuto

AL come servizio 02.

Interrupt 14h: comunicazione seriale asincrona RS232 del ROM BIOS. Consentono di: inizializzare la porta seriale, emettere un singolo carattere, ricevere un singolo carattere, determinare lo stato della porta seriale, inizializzare la porta seriale estesa (PS/2).

Interrupt 15h: avvia motore registratore a cassette, spegne il motore, legge uno o piu' blocchi dati da cassetta, scrive uno o piu' blocchi su cassetta, effettua trasferimento dati in modo protetto (solo AT), determina dimensione memoria estesa, commuta in modo protetto, termine del progamma (servizio AH = 82h, CX = codice ritorno).

Interrupt 17h: trasmette un byte alla stampante, inizializza la stampante, determina lo stato della stampante.

Interrupt 19h: attiva la routine di bootstrap (come ctrl-alt-del).

Interrupt 1Ah: servizi ora e data (legge o scrive data, ora, allarme - per allarme ci vuole TSR)

Interrupt 20h: termina il programma. Non chiude tutti i file aperti con int 21h (funzioni 06h o 16h): bisogna allora usare int 21h funzione 10h (chiude tutti i file aperti). Al suo posto e' meglio usare l'int 21h, funzione 4Ch.

Interrupt 21h: funzioni generali del DOS. Vengono chiamate mettendo in AH il valore della funzione desiderata, e poi chiamando l'interrupt.

Alcune funzioni richiedono l'indirizzo di una stringa che termina con il byte nullo: cioe' e' fatto con la direttiva ASCIIZ (ho cosi' una stringa + chr(0)).

Handle (descrittore del file): e' un numero a 16 bit. Il numero di handle a disposizione e' fissato nel config.sys con files = xxx (max. 255).

Funzione 00h: termina l'esecuzione del programma. Solo per versione 1 del DOS.

Funzione 01h: legge un carattere con eco dal dispositivo standard di input e lo restituisce (quando disponibile) in AL. La funzione aspetta un carattere, ne fa l'eco, puo' essere interrotta con ctrl-c (int 23h).

AL = codice ASCII del tasto premuto. Se e' stato premuto un tasto speciale, AL = 0.

Funzione 02h: scrive un carattere il cui codice ASCII e' in DL.

Funzione 05h: invia il carattere in DL al dispositivo standard di stampa.

Funzione 09h: scrive una stringa terminata dal carattere '$' sul dispositivo standard di output. DS:DX fornisce l'indirizzo di inizio della stringa.

Funzione 0Ah: lettura di una stringa da tastiera e memorizzazione in un buffer. La coppia DS:DX punta al buffer: il primo byte indica la lunghezza del buffer (max. lunghezza della stringa); il secondo byte indica la lunghezza utile della stringa (quanti caratteri effettivamente ci sono); dal terzo byte inizia la stringa. L'ultimo carattere (non contato nella lunghezza utile della stringa) e' 0Dh.

Funzione 0Bh: riporta se e' pronto l'input da tastiera. I risultati sono:

AL = FFh (c'e' input);

AL = 00h (non c'e' input)

Funzione 0Eh: sceglie l'unita' disco attuale. L'unita' disco e' specificata in DL. DL = 0 indica il lettore A, DL = 01h indica il lettore B (se presente). Il primo disco fisso ha sempre DL = 2. AL indica il numero totale di unita' a disposizione.

Funzione 19h; riporta in AL il lettore correntemente selezionato (AL = 0, indica lettore A, AL = 1 indica lettore B....);

Funzione 25h: imposta un vettore di int. DS:DX = indirizzo segmentato della routine di interrupt, AL = numero di interrupt.

Funzione 2Ah: acquisisce la data;

Funzione 2Bh: scrive la data;

Funzione 2Ch: acquisisce l'ora;

Funzione 2Dh: imposta l'ora;

Funzione 4Ch: termina l'esecuzione del programma. Per versioni successive del DOS.

I parametri - che verranno poi restituiti al chiamante - sono:

AL = 00h (conclusione volontaria normale);

AL = 01h (chiusura dal DOS per break da tastiera);

AL = 02h (chiusura dal DOS per errore critico);

AL = 03h (conclusione volontaria con funzione TSR);

Funzione 4Dh: acquisice il codice di ritorno di un programma dopo il termine della sua esecuzione. I risultati sono:

AL = codice di ritorno del programma, leggibile con ERRORLEVEL;

Funzione 2Fh: acquisice l'indirizzo dell'area DTA, mettendolo in ES:DX.

Funzione 30h: acquisice il numero di versione DOS. Restituisce in AL la versione (es. AL = 03h) ed in AH la sottoversione (es. AH = 10h), per vers. 3.10

Funzione 31h: terminate and stay resident (TSR). Il programma termina, ma rimane residente in memoria: non rilascia cioe' la mem. occupata. E' necessario verificare prima che:

La versione DOS sia 2.0 o successiva (funzione 31h non funziona con vers. inferiori a 2.0);

Chiamare la funzione 49h per liberare la memoria assegnata al blocco ambiente di programma;

Determinare la quantita' di mem. da riservare per il programma residente (con int 21h, funzione 48h);

Chiamare la funzione 31h.

Funzione 39h: crea una directory. DS:DX punta ad ASCIIZ con nome directory da creare. I valori restituiti:

Carry = 1, AX = 03h se percorso non trovato;

Carry = 1, AX = 05h se accesso vietato;

Funzione 3Ah: rimuovi una directory. DS:DX punta ad ASCIIZ con nome directory da cancellare. I valori restituiti:

Carry = 1, AX = 03h se percorso non trovato;

Carry = 1, AX = 05h se accesso vietato;

Carry = 1, AX = 10h se tento di cancellare direttorio corrente;

Funzione 3Bh: cambia directory corrente. DS:DX punta ad ASCIIZ con nome directory dove posizionarsi. I valori restituiti:

Carry = 1, AX = 03h se percorso non trovato;

Funzione 3Ch: crea un file. Se il file esiste, tronca la lunghezza a 0; se non esiste lo crea ex-novo. DS:DX punta ad ASCIIZ con nome file da creare.

CX contiene gli attributi del file: bit 5 = archivio, bit 4 = sottodir, bit 3 = etichetta, bit 2 = sistema, bit 1 = nascosto, bit 0 = solo lettura;

I valori restituiti:

se eseguita con successo, Carry = 0, AX = handle del file;

se fallisce, Carry = 1, AX = 03h se percorso non trovato, AX = 04h se nessun handle e' disponibile, AX = 05h se accesso vietato;

Funzione 3Dh: apre un file. Se il file esiste, tronca la lunghezza a 0; se non esiste lo crea ex-novo. DS:DX punta ad ASCIIZ con nome file da creare.
AL contiene codici di accesso al file: AL = 0 (accesso solo lettura), AL = 1 (accesso solo scrittura), AL = 2 (accesso scrittura e lettura).
I valori restituiti:
se eseguita con successo, Carry = 0, AX = handle del file;
se fallisce, Carry = 1, AX = 02h se file non trovato, AX = 03h se percorso non trovato, AX = 04h se nessun handle e' disponibile, AX = 05h se accesso vietato, AX = 0Ch se codice di accesso non valido;

Funzione 3Eh: chiude un file il cui handle e' in BX. Scrive ed aggiorna i buffer: I valori restituiti:
se eseguita con successo, Carry = 0;
se fallisce, Carry = 1, AX = 06h handle non trovato;

Funzione 3Fh: legge da un file il cui handle e' dato da BX. I paramateri sono:
CX = numero di byte da leggere;
DS:DX punta al buffer dove scrivere i dati;
I valori restituiti:
se eseguita con successo, Carry = 0, AX = numero caratteri letti;
se fallisce, Carry = 1, AX = 00h se tenta di leggere oltre fine del file, AX = 05h se accesso vietato, AX = 06h handle non valido;

Funzione 40h: scrive su un file il cui handle e' dato da BX. I paramateri sono:
CX = numero di byte da scrivere;
DS:DX punta al buffer dove leggere i dati da scrivere;
I valori restituiti:
se Carry = 0, AX = CX (numero caratteri scritti), eseguita con successo;
se Carry = 0, AX < CX (numero caratteri scritti), scrittura non completa per mancanza di spazio sul volume;
se Carry = 1, AX = 05h se accesso vietato, AX = 06h handle non valido;

Funzione 41h: cancella un file. DS:DX punta ad ASCIIZ con nome file da cancellare.
I valori restituiti:
se fallisce AX = 02h se file non trovato, AX = 03h se precorso non trovato, AX = 05h se accesso vietato;

Funzione 42h: sposta il puntatore alla posizione corrente di un file;

Funzione 43h: acquisice/imposta gli attributi del file;

Funzione 47h: acquisice la directory corrente;

Funzione 48h: assegna un blocco di memoria (di 16 byte);

Funzione 49h: libera un blocco di memoria;

Funzione 4Bh: carica ed esegue un programma (exec);

Funzione 56h: cambia il nome ad un file;

Funzione 57h: acquisisce/imposta la data e l'ora di un file;

Funzione 62h: restituisce il segmento del PSP in BX (BX = segm di PSP). Funziona solo dalla versione DOS 3.0 in poi.

Interrupt 22h: indica dove viene trasferito il controllo del computer quando termina l'esecuzione di un programma. Pericoloso modificare.

Interrupt 23h: indirizzo del gestore di ctrl-c. Indica dove viene trasferito il controllo del computer quando viene premuto il tasto di interruzione esecuzione. Generalmente dopo un ctrl-C si puo':

- riprendere l'esecuzione dove e' stata interrotta (lo si fa con iret) come se non fosse successo nulla;
- tornare al DOS, ponendo termine all'esecuzione del programma;
- intraprendere qualche azione particolare, come ad es. cancellare i files temporanei utilizzati. Attenzione: non e' noto lo stato dello stack!

Il Program Segment Prefix (PSP)

Program Segment Prefix: e' il prefisso del segmento del programma (pag. 316 libro Norton). E' caricato in memoria in un'area di 256 byte. Ogni programma DOS ne ha uno, qualunque sia il linguaggio. Se uso ASM, posso e devo utilizzare le informazioni del PSP: con l'utilizzo di linguaggi ad alto livello ne posso fare a meno.
Il command.com calcola l'indirizzo piu' basso dove caricare il programma: tale indirizzo e' l'inizio del segmento programma. A partire da tale indirizzo vengono posti i 256 byte del PSP: subito dopo c'e' il programma, a cui viene ceduto il controllo. All'inizio dell'esecuzione, DS ed ES puntano a PSP. Nel PSP ci sono informazioni **inutili**, in quanto ereditate dal CP/M o da vecchie versioni del DOS. Le informazioni utili sono ai seguenti offset:

- 000Ah (4 byte): indirizzo int 22h (termine programma);
- 000Eh (4 byte): indirizzo int 23h (ctrl-c);
- 0012h (4 byte): indirizzo int 24h (errore critico). Sono usati dal DOS per ripristinare i valori corretti - qualora i valori correnti siano stati modificati dal programma in esecuzione;
- 0080h (1 byte): lunghezza del command tail;
- 0081h (lunghezza variabile): coda del comando (command tail):

Un semplice programma ASM

Si sviluppi un programma assembler che scriva un messaggio sul video, effettui la lettura da tastiera di una stringa e riscriva sul video la stringa letta.

```
;
; Compilare con:        MASM PROG.ASM
; Linkare con:          LINK PROG
; Rendere file com con: EXE2BIN PROG.EXE PROG.COM
; Cancellare l'exe con: DEL PROG.EXE
; Eseguire con:         PROG
;
CODE    SEGMENT
        ASSUME      CS:CODE,DS:CODE
        PAGE        60,132              ; 60 righe, 132 colonne per il
                                        ; listato assembler
        ORG         100H                ; 256 bytes iniziali. I file
                                        ; .com iniziano SEMPRE a $100,
                                        ; perche' prima c'e' il PSP
INIZIO:                                 ; Inizio esecuzione
; Visualizzazione testo di saluto
        MOV         AH,09H              ; Setta registro AH
        LEA         DX,TESTO            ; Indirizza il testo
;       MOV         DX, OFFSET TESTO    ; Del tutto equivalente a LEA
        INT         21H                 ; Chiamata all'emissione
; Immissione
ING:    MOV         AH,0AH              ; Setta registro AH
        LEA         DX,RIGA             ; Indirizzo di immissione
        INT         21H                 ; Chiamata all'immissione
; Emissione
USC:    MOV         RIGA,0DH            ; CR dopo RIGA
        MOV         RIGA+1,0AH          ; LF dopo RIGA+1
        MOV         AH,09H              ; Setta registro AH
        LEA         DX,RIGA             ; Chiamata all'emissione
        INT         21H
; Termina il programma dinamicamente
FINE:   MOV         AH,0
        INT         21H
; Dati
RIGA    DB          80,0,81 DUP("$")
TESTO   DB          "Per favore impostare i caratteri",0DH,0AH,"$"
CODE    ENDS
        END         INIZIO
```

Stack e Passaggio dei Parametri

Il passaggio dei parametri da un programma chiamante ad una procedura avviene attraverso lo stack. Nel chiamante deve esserci il segmento dello stack: e' obbligatorio. Se ad es. il chiamante deve passare tre parametri alla procedura che proveddera' a sommarli e a restituire in AX il risultato, si avra':

```
        . . . . . .                     ; Programma chiamante
        MOV         AX, 10h             ; Inizializz. reg. e variabili
        MOV         ARG2, 100h
        MOV         CX, 5h
        PUSH        AX                  ; Terzo parametro
        PUSH        ARG2                ; Secondo parametro
        PUSH        CX                  ; Primo parametro
        CALL        ADDUP               ; Chiamo la procedura
        ADD         SP, 6               ; Ripristino lo stack, togliendo
                                        ; i valori messi come parametri per
                                        ; ADDUP. E' equivalente a fare
                                        ; 3 POP di seguito.
        . . . . . .                     ; Resto del chiamante

ADDUP   PROC        NEAR                ; Inizio codice procedura ADDUP
                                        ; NEAR = il codice della proc.
                                        ; e' nello stesso segmento di
                                        ; quello del chiamante.
        PUSH        BP                  ; ADDUP salva il reg. BP, cosi' che
                                        ; lo possa ripristinare al termine
                                        ; per il chiamante
        MOV         BP, SP              ; BP e' la base dello stack per
                                        ; ADDUP
        MOV         AX, [BP+4]          ; AX carica primo parametro (CX)
        ADD         AX, [BP+6]          ; AX = AX + secondo parametro (ARG2)
        ADD         AX, [BP+8]          ; AX = AX + terzo parametro (AX)
                                        ; In AX ho adesso il risultato
        POP         BP                  ; Ripristino il valore di BP, salvato
                                        ; all'inizio di ADDUP
        RET                             ; Termine della procedura. Il
                                        ; risultato e' in AX
ADDUP   ENDP                            ; Dice all'assemblatore che il
                                        ; codice per la procedura e' terminato
```

Dopo la PUSH CX del chiamante, lo stack e' in fig. **.a. Con la CALL automaticamente sullo stack viene posto l'indirizzo di ritorno (RA = Return Address). Poiche' la procedura e' NEAR, il RA e' solo l'offset (il segmento e' lo stesso): quindi occupa solo 2 byte. Se ADDUP fosse stata far, il RA sarrebe stato lungo 4 byte, essendo composto dall'intero indirizzo segmentato di ritorno (2 byte per il segmento e 2 per l'offset).

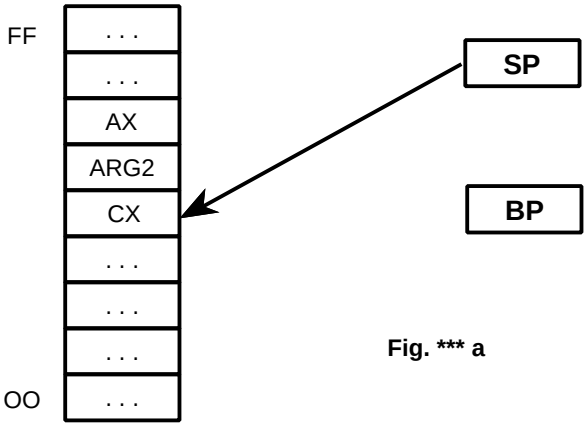


Fig. * a**

Dopo la PUSH BP della procedura, lo stack e' in fig. **.b. Con il RET tolgo automaticamente dallo stack l'indirizzo di ritorno (RA), mettendolo in IP.

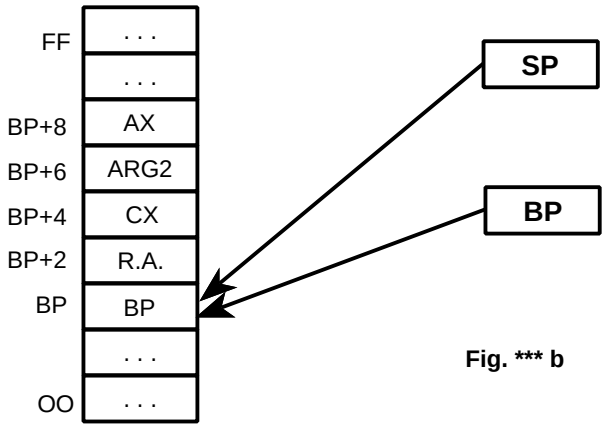


Fig. * b**

Il chiamante con l'istruzione ADD SP, 6 ripristina poi lo stack cosi' come esso era prima che vi mettesse i tre parametri per ADDUP. L'istruzione ADD SP, 6 equivale del tutto ad una serie di tre POP consecutive.

Chiamate a Procedure ASM da C e Pascal

I programmi ad alto livello (C e Pascal, ad es.) possono chiamare procedure scritte in assembler. Essi hanno un metodo standard (convenzione) per passare i parametri: tale convenzione e' tuttavia caratteristica del linguaggio, e non uguale per tutti i linguaggi. Ad es. il Pascal utilizza il modello MEDIUM, in cui ho un solo segmento per i dati e piu' segmenti per il codice: di conseguenza il RA sara' lungo 4 byte (2 per l'offset e 2 per il segmento, in quanto l'indirizzo dovra' essere segmentato). Il Pascal mette i parametri sullo stack nell'ordine argomento1, argomento2, argomento3... Sempre il Pascal affida alla procedura il compito di togliere dallo stack i parametri che le sono stati passati. Di conseguenza si avra' per il programma Pascal chiamante (che calcola 3 x 2^5 = 3 x 32 = 96):

```
program Asmtest(input, output);
function Power2(a:integer; b:integer): integer; extern;
begin
   writeln('3 times 2 to the power of 5 is ', Power2(3,5));
end.
```

e per la procedura chiamata:

```
.MODEL medium                       ; Esempio tratto da MASM5.1
.CODE
        PUBLIC    Power2
Power2  PROC
        push      bp                ; Entry sequence - save old BP
        mov       bp, sp            ; Set stack framepointer
        mov       ax, [bp+8]        ; Load Arg1 into AX
        mov       cx, [bp+6]        ; Load Arg2 into CX
        shl       ax, cl            ; AX = AX * (2 to power of CX)
                                    ; Leave return value in AX
        pop       bp                ; Restore old framepointer
        ret       4                 ; Exit, and restore 4 bytes of args
Power2  ENDP
        END
```

Il C utilizza il modello SMALL (puo' usare pero' anche gli altri), in cui ho un solo segmento per i dati ed un solo segmento per il codice: di conseguenza il R.A. sara' lungo 2 bye per l'offset (non e' necessario il segmento). Il C antepone il carattere '_' a tutti i nomi delle procedure: il C mette i parametri sullo stack nell'ordine argomento3, argomento2, argomento1... Il C inoltre affida al chiamante il compito di togliere dallo stack i parametri passati alla procedura: tale operazione e' quindi effettuata dal codice C. Di conseguenza si avra' per il programma C chiamante (che calcola 3 x 2^5 = 3 x 32 = 96):

```
extern int Power2(int,int);

main()
{
   printf("3 times 2 to the power of 5 is %d\n", Power2(3,5));
   exit(0);
}
```

e per la procedura chiamata:

```
.MODEL small
.CODE
        PUBLIC    _Power2           ; E' anteposto '_' al nome della proc.
_Power2 PROC
        push      bp                ; Entry sequence - save old BP
        mov       bp, sp            ; Set stack framepointer
        mov       ax, [bp+4]        ; Load Arg1 into AX
        mov       cx, [bp+6]        ; Load Arg2 into CX
        shl       ax, cl            ; AX = AX * (2 to power of CX)
                                    ; Leave return value in AX
        pop       bp                ; Restore old framepointer
        ret                         ; Exit sequence
_Power2 ENDP
        END
```

Le Strutture

Le stutture: e' possibile definire strutture, che sono del tutto analoghe alle struct del C ovvero ai record del Pascal. Ad es.

```
ALUNNO  STRUC                       ; Inizio definizione struttura
        NOME      DB  15 DUP('?')   ; Nome e' lungo 15 caratteri
        COGNOME   DB  15 DUP('?')   ; Cognome e' lungo 15 caratteri
        MATRICOLA DB   6 DUP('?')   ; Matricola e' lungo 6 caratteri
        ANNO      DB   1 DUP('?')   ; Anno e' lungo 1 carattere
ALUNNO  ENDS                        ; Fine della struttura
```

Per instanziare degli alunni faro':

```
STUD1   DB 37 DUP ('?')             ; Ho uno studente
STUD2   DB 37 DUP ('?')             ; Ho un altro studente
```

Posso poi accedere direttamente ai campi del record con:

```
        MOV       SI, 00H           ; SI = 0
        LEA       BX, STUD1         ; BX punta a STUD1
        MOV       AX, [BX].ANNO     ; Metto in AX l'anno
        MOV       AX, [BX].MATRICOLA[SI]
                                    ; Metto in AX il primo carattere
                                    ; della matricola
```

Tale accesso ai campi e' del tutto simile a quello caratteristico di C e Pascal (notazione con il punto o 'dot notation').

Le Macro

Le macro: sono del tutto analoghe a quelle di AS11. Esse vengono espanse alla prima analisi del codice effettuata da parte dell'assemblatore. Si ricorda che quando l'assemblatore incontra una chiamata ad una macro, il codice della macro viene ricopiato nel punto esatto dove la macro e' stata chiamata: le macro sono differenti dalle procedure. Ad esempio, la macro:

```
LEGGI   MACRO     CAMPO, LUGHEZZA   ; LEGGI e' il nome della macro.
        MOV       AH, 3FH
        LEA       DX, CAMPO
        MOV       CX, LUNGHEZZA
        MOV       BX, 01H
        INT       21H
        ENDM                        ; Termine della macro
```

che viene chiamata nel seguente modo:

```
        . . . . .
        LEGGI     BYTE1, 20H        ; Chiamata alla macro
        . . . . .
```

viene tradotta come:

```
        MOV       AH, 3FH
        LEA       DX, BYTE1
        MOV       CX, 20H
        MOV       BX, 01H
        INT       21H
```

Direttive all'Assemblatore

Direttive all'assemblatore: sono dei comandi inseriti all'interno del codice sorgente, non generano codice vero e proprio ma vengono utilizzati per fornire delle istruzioni all'assemblatore. Alcuni di esse sono:

.8086: assembla le istruzioni utilizzando solo l'instruction set dell'8086;

.8087: assembla includendo le istruzioni dell'8087;

.286: assembla includendo le istruzioni dell'80286;

.MODEL: consente di utilizzare il modello SMALL, MEDIUM, LARGE, HUGE. Posso avere un segmento dati ed un segmento codice, oppure piu' dati e piu' codice;

nome = espressione: nome contiene il valore numerico dell'espressione (ad es. VAL = 32H, MOV AX, VAL equivale a mettere 32H nel registro AX);

ASSUME Segment_reg:nome_seg tutti i nomi contenuti in nome_seg vengono indirizzati usando Segment_reg;

DB, DW, DD, DQ, DT: riserva un byte, una parola (16 bit), una doppia parola (32 bit), una parola quadrupla (64 bit), una parola decupla (80 bit);

ENDP: e' il termine di una procedura;

ENDS: e' il termine di un segmento;

nome GROUP nome1,nome2,nome3: raggruppa in un unico segmento nome i vari segmenti nome1, nome2, nome3. Dovro' poi fare ASSUME RegS:nome;

INCLUDE nome_file: include nella posizione corrente il contenuto del file nome_file;

ORG: setta il livello di indirizzamento al valore dichiarato. Di solito si usa ORG 100H per i progammi com (prima c'e' il PSP);

PARA: l'etichetta e' spostata al primo paragrafo successivo libero (ogni paragrafo ha la dimensione di 16 byte). Es: MY_CODE SEGMENT PARA 'CODE';

PAGE: l'etichetta e' spostata alla prima pagina successiva libera (ogni pagina ha la dimensione di 256 byte). Es: MY_CODE SEGMENT PAGE 'CODE ';

nome PROC tipo: inizia la definizione di una procedura chiamata nome. Tipo e' near (se la procedura sta nello stesso segmento codice del chiamante), oppure far (nell'altro caso);

PUBLIC nome: nome e' disponibile ad altri programmi tramite il linker. Il linre cioe' cerca di risolvere simboli utilizzando anche nome;

nome SEGMENT: assegna il nome al segmento in uso;

PAGE lung, larg: determina lunghezza e larghezza della pagina del listato assembler;

TITLE testo: definisce testo come il titolo di una pagina del listato.

L'utilizzo di Piu' Moduli

E' possibile (e richiesto per l'esame) dividere il codice in piu' moduli, ognuno compilabile separatamente: anche i dati possono essere presenti in piu' moduli. La direttiva che consente all'assemblatore di utilizzare etichette presenti in altri moduli e' EXTRN: essa dice che l'etichetta che segue e' esterna, cioe' sta in un altro modulo. Inoltre, affinche' l'etichetta sia visibile dall'esterno, dovra' essere dichiarata PUBLIC. L'etichetta puo' essere sia un dato (nel DS) che una istruzione (nel CS).

Riepilogando: per utilizzare etichette di altri moduli, e' necessario dichiarale EXTRN ed informare l'assemblatore sul loro tipo. Per rendere utilizzabili da altri le proprie etichette, e' necessario dichiararle PUBLIC. Vediamo di seguito un esempio di modulo con dati propri che chiama una procedura di un altro modulo:

```
;
; File: SHOWREG3.ASM
;
; Compilare con:    MASM SHOWREG3.ASM;
; Richiede:         SHOWREG3.ASM
; Linkare con:      LINK SHOWREG3 SHOWREG4
;
;
; Programma di visualizzazione del contenuto del registro AX.
; Utilizza una procedura nel modulo SHOWREG4.ASM.
; Differisce da SHOWREG1.ASM perche' i dati sono nel chiamante
; SHOWREG2.ASM vede i dati come EXTRN.
;
; Giuseppe Pozzi, Politecnico di Milano - Sede staccata di Como -
; Como, 19 aprile 1994
;
;
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
MY_STACK   SEGMENT PARA STACK 'STACK'
ST_BEG     DB         100 DUP("$")
ST_END     DB         "$"
MY_STACK   ENDS
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -


; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
MY_DATA    SEGMENT PARA PUBLIC 'DATA'
TESTO      DB        "Registro "           ; Dati
REGNAME    DB        "AX = "
           DB        "$"
RH1        DB        00
RH2        DB        00
RL1        DB        00
RL2        DB        00
           DB        0Dh
           DB        0Ah
           DB        "$"
           PUBLIC TESTO,REGNAME,RH1,RH2,RL1,RL2
                                          ; Le etichette sono visibili
                                          ; dall'esterno
MY_DATA    ENDS
```

```
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -


; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
MY_CODE  SEGMENT PARA PUBLIC 'CODE'
         ASSUME    CS:MY_CODE

         EXTRN     STAMPA_REG:NEAR         ; Sara' definita altrove

INIZIO:                                    ; Inizio percorso
; Prologo standard al programma. I registri DS ed ES puntano al PSP
;
         PUSH      DS                      ; Salvo DS
         PUSH      AX                      ; Salvo AX
         MOV       AX, 00H                 ; AX = 00h

         MOV       AX, MY_DATA             ; AX = MY_DATA
         MOV       ES, AX                  ; ES = MY_DATA
         ASSUME    ES: MY_DATA             ;

         MOV       DS, AX                  ; DS = MY_DATA
         ASSUME    DS: MY_DATA             ;

         MOV       AX, MY_STACK            ; AX = MY_STACK
         MOV       SS, AX                  ; SS = MY_STACK
         ASSUME    SS: MY_STACK
         MOV       SP, OFFSET ST_END       ; SP punta alla fine dello stack

         MOV       AX, 01234H              ; Valore di AX da stampare
         PUSH      AX                      ; Metto sullo stack il valore
                                           ; che deve essere stampato
         CALL      STAMPA_REG              ; Chiamo la procedura
         ADD       SP, 02H                 ; Ripristino SP

         MOV       AX, 09876H              ; Valore di AX da stampare
         PUSH      AX                      ; Metto sullo stack il valore
                                           ; che deve essere stampato
         CALL      STAMPA_REG              ; Chiamo la procedura
         ADD       SP, 02H                 ; Ripristino SP

         MOV       AX, 0ABCDH              ; Valore di AX da stampare
         PUSH      AX                      ; Metto sullo stack il valore
                                           ; che deve essere stampato
         CALL      STAMPA_REG              ; Chiamo la procedura
         ADD       SP, 02H                 ; Ripristino SP

         MOV       AX, 0FEDCH              ; Valore di AX da stampare
         PUSH      AX                      ; Metto sullo stack il valore
                                           ; che deve essere stampato
         CALL      STAMPA_REG              ; Chiamo la procedura
         ADD       SP, 02H                 ; Ripristino SP

; Termine del programma dinamicamente
FINE:
         POP       AX
         POP       DS
         MOV       AH, 4CH                 ; AH = 4Ch (termine programma)
         MOV       AL, 0                   ; esito (per ERRORLEVEL)
         INT       21H                     ; Chiamata all'interrupt

MY_CODE  ENDS
         END       INIZIO
```

Un esempio di modulo che include una procedura ed utilizza dati del chiamante e':

```
;
; File: SHOWREG4.ASM
;
; Compilare con:    MASM SHOWREG4.ASM;
; Richiede:         SHOWREG3.ASM come programma chiamante
;
;
; Contiene la procedura STAMPA_REG per il chiamante SHOWREG3.ASM.
; Differisce da SHOWREG2.ASM perche' i dati TESTO, REGNAME.....
; sono EXTRN e sono dichiarati in SHOWREG3.ASM
;
; Giuseppe Pozzi, Politecnico di Milano - Sede staccata di Como -
; Como, 19 aprile 1994
;
;
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
MY_DATA    SEGMENT PARA PUBLIC 'DATA'
           EXTRN
TESTO:BYTE,REGNAME:BYTE,RH1:BYTE,RH2:BYTE,RL1:BYTE,RL2:BYTE
MY_DATA    ENDS
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

MY_CODE    SEGMENT PARA PUBLIC 'CODE'

;
; Procedure
;
         ASSUME    CS:MY_CODE              ; Gia' inizializzati
         ASSUME    DS:MY_DATA              ; dal chiamante

         PUBLIC    STAMPA_REG              ; STAMPA_REG e' visibile
                                           ; all'esterno

STAMPA_REG         PROC NEAR

         PUSH      BP                      ; Salvo BP
         MOV       BP, SP                  ; Fisso SP
         PUSH      AX                      ; Salvo AX
         MOV       AX, [BP+4]              ; Recupero dallo stack
                                           ; il primo parametro.
                                           ; BP - 2 e' AX salvato
                                           ; BP     e' il BP salvato
                                           ; BP + 2 e' Ret. Addr.
                                           ; BP + 4 e' parametro

; Visualizzazione testo iniziale
         MOV       AH, 09H                 ; AH = 9h (stampa stringa)
         LEA       DX, TESTO               ; Indirizza il testo
         INT       21H                     ; Chiamata all'interrupt

; Conversione
         MOV       AX, [BP+4]              ; Rileggo il parametro
         MOV       CL, 04H

         SHR       AH, CL                  ; AH = nibble alto di AH
         ADD       AH, 30H                 ; AH = c.ASCII nibble alto AH
         MOV       RH1, AH

         MOV       AX, [BP+4]              ; Rileggo il primo parametro
         AND       AH, 0FH                 ; AH = nibble basso di AL
         ADD       AH, 30H                 ; AH = c.ASCII nibble basso AH
         MOV       RH2, AH

         MOV       AX, [BP+4]              ; Rileggo il primo parametro
```

```
          MOV       AH, AL                  ; AH = AL
          SHR       AH, CL                  ; AH = nibble alto di AH
          ADD       AH, 30H                 ; AH = c.ASCII nibble alto AH
          MOV       RL1, AH

          MOV       AX, [BP+4]              ; Rileggo il primo parametro
          MOV       AH, AL                  ; AH = AL
          AND       AH, 0FH                 ; AH = nibble basso di AL
          ADD       AH, 30H                 ; AH = c.ASCII nibble basso AL
          MOV       RL2, AH

; Verifica dei risultati. Possono esserci caratteri A-F
          MOV       SI, 0h
          LEA       BX, RH1                 ; Se il cod.ASCII e' sup. a 39h,
ANCORA:   CMP       BYTE PTR [BX][SI], 39H  ; devo aggiungere 7h per avere
          JBE       NEXT                    ; il carattere A
          ADD       BYTE PTR [BX][SI], 07h
NEXT:     INC       SI                      ; Carattere successivo
          CMP       SI, 04H                 ; Sono 4 i caratteri da analizz.
          JNE       ANCORA

; Visualizzazione risultati
          MOV       AH, 09H                 ; AH = 09h (stampa stringa)
          LEA       DX, RH1                 ; inizio stringa risultati
          INT       21H                     ; Chiamata all'interrupt

; Termine della procedura
          POP       AX                      ; Ripristino AX salvato
          POP       BP                      ; Ripristino BP salvato
          RET
STAMPA_REG          ENDP                    ; Termine procedura


MY_CODE   ENDS
          END
```

Bibliografia

- Peter Norton, Richard Wilton, PS/2 & PC IBM - Guida del programmatore, Mondadori informatica, 1989, o in alternativa
- Julio Sanchez, Assembly language tools and techniques for the IBM microcomputers, Prentice Hall, Englewood Cliffs, New Jersey, 1990.

Esempi di progettini ASM 8088

Calcola CRC-16 di un file
Cerca file omonimi in due directory
Comando dir
Compressione/decompressione files: alla compress
Conversione di numero da una base ad un'altra
Dice se un numero e' primo (numero tra 0 e 32767) + radice se e' intera
Disegna rettangoli (recursivi) in VGA
Disegna rettangolo su schermo. Generaz. rettangoli gemelli
Disegna triangolo in VGA
Disegno righe sul video con aiuto del mouse, stampa con printscreen
FDTOOLS: visualizza spazio occupato da ogni subdir
Filefind
Filefind + lunghezza, data, ora modifica
Ingresso veloce in sub-dir (alla ncd)
Linee e punti disegnati sul video con mouse
rm alla Unix, con numero file/dir rimossi e spazio liberato
Stampa caratteri esadecimali
TSR per visualizzare i codici ASCII dei tasti premuti
Visual. info su coproc/porte/ spazio su disco/schede utente...
Visualizza elenco file presenti ed attributi (anche subdir)
Visualizza rettangolo: muove, rimpicciolisce, ingrandisce
Visualizza spazio libero su disco (n. settori totali, settori liberi...)
Visualizza dimensione, elenco blocchi occupati, data e ora modifica del file

TCP/IP

Definizioni

Commutazione di circuito. Connessione dedicata tra due punti. Banda garantita di 64 kbps.
Svantaggi: costo. Vantaggi: capacita' garantita.

Commutazione di pacchetto. Il traffico e' diviso in piccoli pezzi (pacchetti), di poche centinaia di byte. Svantaggi: ricezione frammentata. Vantaggi: piu' comunicazioni simultanee.

Categoria delle reti.

reti locali: Local Area Network, LAN. Velocita' tipiche 4Mbps - 2 Gbps;

reti metropolitane: Metropolitan Area Network, MAN. Velocita' tipiche 56 kbps - 100 Mbps;

reti geografiche: Wide Area Network, WAN. Velocita' tipiche 9.6 kbps - 45 Mbps.

Le reti locali hanno dimensione limitata.

Nessuna rete puo' servire tutti gli utenti.

Gli utenti desiderano l'interconnessione universale.

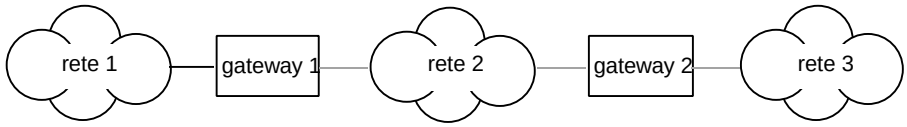
Internet: INTERconnected NETworks.

Tutte le macchine condividono un insieme universale di identificatori (nomi o indirizzi).

Architettura dell'internet.

E' necessaria la presenza di un sistema per consentire il transito dei pacchetti da una rete ad un'altra. Il sistema e' chiamato gateway IP o router. I router instradano i pacchetti in base alla rete di destinazione.

Per l'utente il sistema si deve presentare come una singola rete, come se tutte le reti fossero uguali. L'utente vuole raggiungere un certo calcolatore senza dover considerare come le reti siano collegate tra di loro.



Tre reti interconnesse da due gateway.

Gli Indirizzi dell'Internet

Un sistema di comunicazione e' detto universale se consente ad un host qualsiasi di comunicare con qualsiasi altro host.

Ad ogni host che sia collegato ad un internet TCP/IP viene associato un indirizzo numerico, unico, composto da 32 bit. Tutte le comunicazioni con quell'host devono fare riferimento a quel numero.

Ogni indirizzo e' composto da una coppia netid, hostid. Esistono tre classi primarie di reti, definite in base al numero di host collegati alla rete.

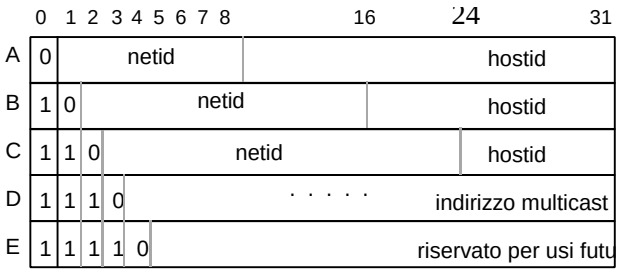
classe A: per reti che hanno piu' di 2^16 (65535) host;

classe B: per reti il cui numero di host e' tra 2^8 (256) e 2^16 (65536);

classe C: per reti il cui numero di host e' inferiore a 2^8 (256);

classe D: multicast

classe E: riservato per usi futuri



Le forme di indirizzi internet IP.

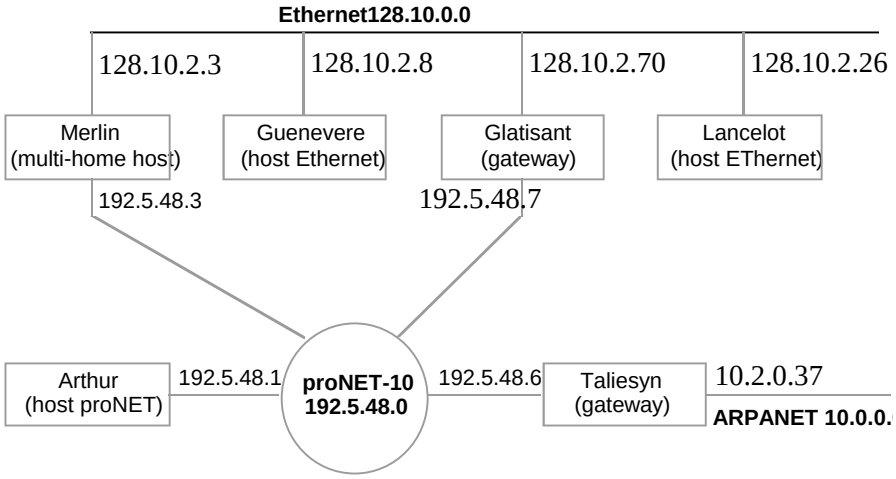
La notazione decimale. Gli indirizzi IP, composti da 32 bit, possono essere scritti in notazione decimale come ad es:

forma binaria: 10000011.10101111.00111001.00000001

forma decimale: 131.175.57.1

L'ordine dei byte nella rete. Lo standard per l'ordine dei byte nella rete specifica che gli interi sono trasmessi a partire dal byte piu' significativo.

Un esempio di internet composta da reti Ethernet, Token Ring e ARPANET: tale esempio era utilizzato alla Purdue University agli inizi degli anni 80.



Esempio di assegnazione di indirizzi IP per host e gateway in una rete Ethernet, Token Ring e ARPANET.

Consegna del Datagramma senza Connessione

Architettura e filosofia dell'internet: TCP/IP fornisce 3 insiemi di servizi. La relativa architettura a strati e' la seguente:

servizi di applicazione
servizio di trasporto affidabile
servizio di consegna del pacchetto senza connessione

Il sistema di consegna senza connessione. Il servizio IP e' detto:

- senza connessione, perche' ogni pacchetto e' trattato indipendentemente dagli altri;
- inaffidabile, perche' la consegna non e' garantita: inoltre possono esservi pacchetti persi, duplicati, ritardati o consegnati fuori sequenza;
- best-effort, perche' il sw compie ogni tentativo per consegnare i pacchetti correttamente.

Lo scopo di IP. IP fornisce tre definizioni importanti:

- formato esatto di tutti i dati;
- funzione di instradamento;
- insieme di regole che inglobano l'idea di consegna non affidabile.

Stratificazione del protocollo

L'esigenza di piu' protocolli. Viene utilizzata una famiglia di protocolli (o suite di protocolli).

La suddivisione in protocolli consente una migliore analisi di ogni protocollo. I vari protocolli gestiscono:

malfunzionamenti hw: un gateway o un host possono manifestare un crollo;

congestione della rete: una macchina congestionata puo' sopprimere l'ulteriore traffico;

ritardo o perdita di pacchetti: il sw deve essere informato di mancata consegna o deve adattarsi a lunghi ritardi;

alterazione dei dati: possibili errori di trasmissione;

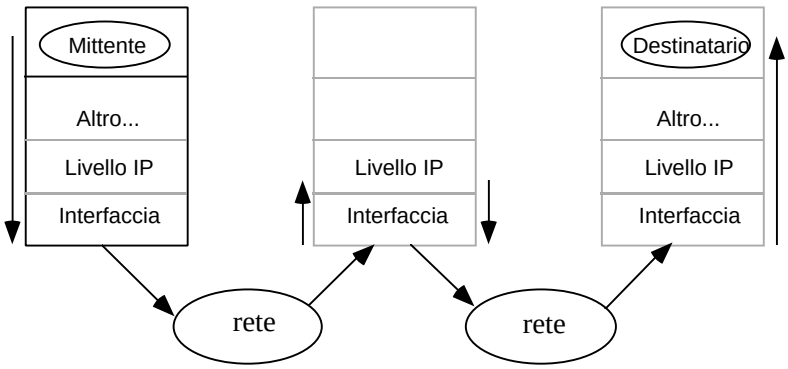
duplicazione dei dati o errori nella sequenza: ad es. prodotti da reti che offrono piu' percorsi.

Livelli concettuali del sw del protocollo. L'inizio di un msg da un applicativo ad un altro su una macchina differente implica il trasferire il msg verso il basso attraverso i livelli del protocollo nel mittente e - a rovescio - nel destinatario. I livelli concettuali del protocollo sono:

ad alto livello:

a livello IP:

a livello di interfaccia di rete:



Percorso di un messaggio che viaggia in internet dal mittente al ricevitore attraverso una macchina intermedia.

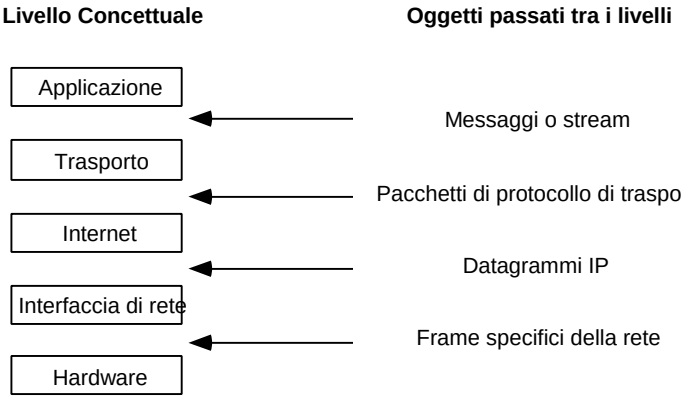
Il modello di stratificazione dell'internet TCP/IP. Sono previsti i seguenti livelli:

livello di applicazione: gli utenti chiamano i programmi applicativi che accedono ai servizi disponibili attraverso un internet TCP/IP;

livello di trasporto: comunicazione tra un programma applicativo ed un altro. Connessione *end-to-end*. Il sw di trasporto scompone il messaggio trasmesso in piu' pacchetti;

livello di internet: gestisce la comunicazione da una macchina ad un'altra. Il messaggio viene incapsulato in un datagramma. I datagrammi sono analizzati (verifica correttezza) e inoltrati o elaborati localmente;

livello di interfaccia di rete: accetta i datagrammi e li trasmette su una rete specifica; hardware.



I quattro livelli concettuali e la forma degli oggetti passati.

I confini importanti del modello TCP/IP.

Applicazione	sw esterno sl s.o.
Trasporto	sw interno al s.o.
Internet	**utilizza solo indirizzi IP**
Interfaccia di rete	**utilizza solo indirizzi fisici**
Hardware	

Svantaggi della stratificazione. La stratificazione consente al progettista di dividere il problema complesso in piu' sottoproblemi. Se la trasmissione e' nella stessa sottorete, al livello di trasporto vale la pena individuare la dim. dei pacchetti che riduca il numeri di pacchetti trasmessi (pacchetti grandi): se invece la trasmissione e' tra reti differenti, e' meglio individuare pacchetti di dimensioni piu' ridotte. Una stratificazione rigida impedisce di ottimizzare l'utilizzo della rete. E' pero' possibile utilizzare dei buffer....

User Datagram Protocol

Scopo: distinguere piu' destinazioni all'interno dello stesso host, consentendo a piu' programmi dello stesso host di ricevere/trasmettere datagrammi indipendentemente.

Identificazione della destinazione finale.

Ogni macchina contiene un insieme di punti destinazione, dette **porte del protocollo**.

Ogni porta e' individuata da un numero intero positivo unico.

Per un processo, l'accesso alla porta e' sincrono. Le porte sono bufferizzate.

Ciascun messaggio include porta di partenza e porta destinazione.

Lo UDP. Definisce il protocollo per l'invio di datagrammi tra programmi applicativi. Lo UDP utilizza il sottostante IP. Lo UDP:

non ordina i messaggi in arrivo, non fornisce feedback al mittente;

e' non affidabile, senza connessione. Distingue pero' piu' destinazioni nello stesso host.

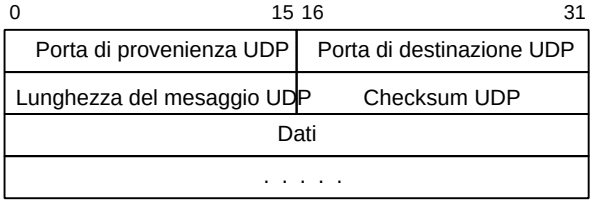
Formato dei messaggi UDP.

porta di provenienza: 16 bit;

porta di destinazione: 16 bit;

lunghezza in numero di byte: 16 bit, include porte di partenza e porte di destinazione. Valore minimo 8;

checksum (facoltativa): da non usare.



Formato dei campi per il datagramma UDP.

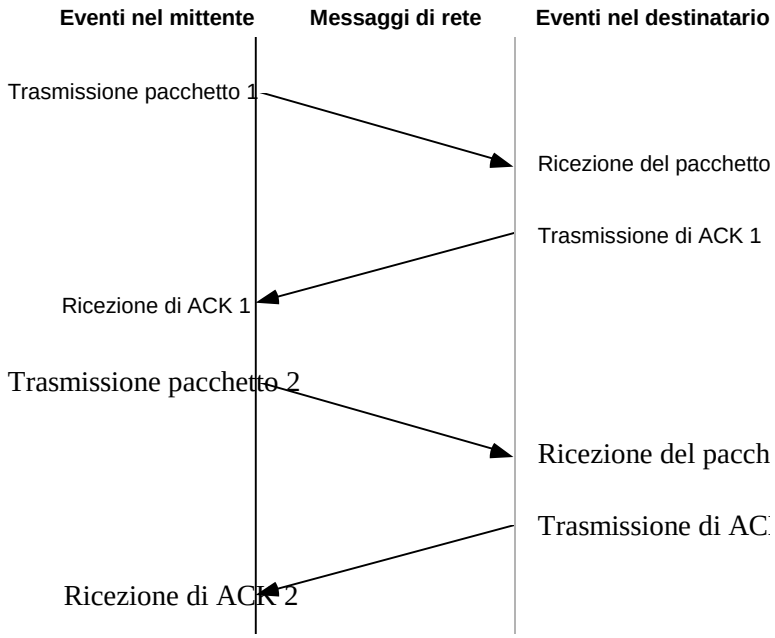
Il Servizio Affidabile di Trasporto di Stream (TCP)

Transmission Control Protocol (TCP).

Proprieta' del servizio di consegna affidabile.

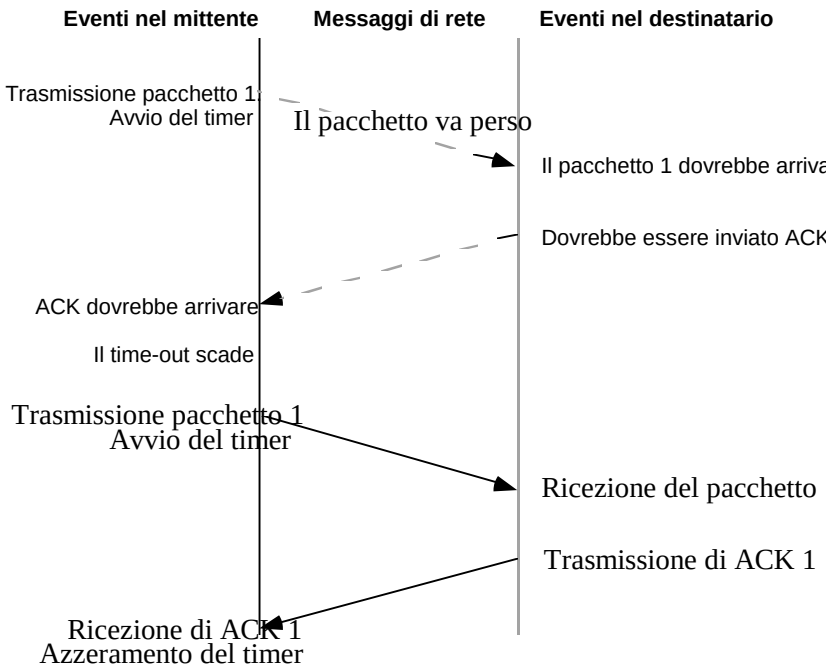
- Orientamento allo stream: i dati trasferiti sono immaginati come stream (sequenze) di byte;
- Connessione di circuito virtuale: il trasferimento avviene tra macchina chiamante e macchina chiamata, che stabiliscono una connessione utilizzando un circuito virtuale. In caso di failure della connessione, entrambe le macchine se ne accorgerebbero;
- Trasferimento bufferizzato: vengono raccolti dati prima di completare un datagramma. Esiste servizio di push per spedire un datagramma prima che sia completato (utile ad es. se si fa telnet);
- Stream non strutturato: non vengono onorati i dati strutturati (ad es. un record puo' essere trasmesso in datagrammi differenti);
- Connessione full-duplex: la connessione tra due host e' full-duplex.

Affidabilita'. E' una caratteristica imprescindibile. Poiche' la trasmissione e' non affidabile, si utilizza il *riscontro positivo di trasmissione*. Il destinatario deve mandare un messaggio di acknowledge quando ha ricevuto i dati.



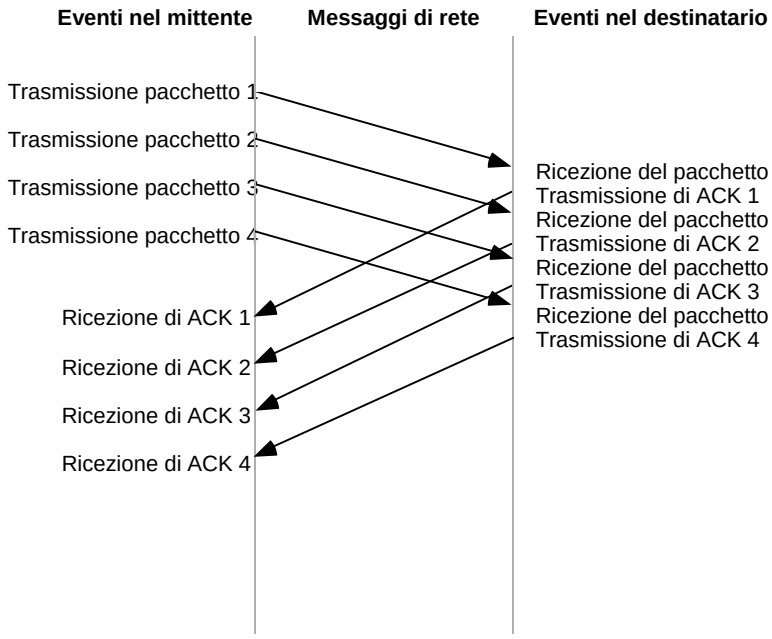
Protocollo con riscontro positivo di ricezione.

Il mittente ha un time-out entro il quale rimandare i dati se il loro ricevimento non e' stato confermato.



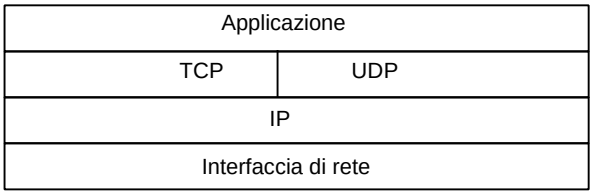
Trasmissione con perdita di pacchetto.

Le finestre scorrevoli. Per aumentare il throughput vengono spediti piu' pacchetti consecutivi.
Di ogni pacchetto deve essere confermato il ricevimento.



Trasmissione contemporanea di piu' pacchetti.

Il TCP. Specifica il formato dei dati e dei riscontri per ottenere un trasferimento affidabile.
Risiede SOPRA il protocollo IP.



Stratificazione tra i protocolli IP, UDP, TCP.

Porte, connessioni e punti terminali.

Impiega il **numero di connessione** (e non il numero di porta) per identificare la destinazione finale.

Ogni porta non corrisponde ad un singolo oggetto.

Ogni punto terminale (end-point) e' costituito da *host, porta*. Ogni connessione e' definita da due end-point.

Una porta puo' essere condivisa da piu' connessioni di una unica macchina.

Aperture. La open di un applicativo e' passiva se e' pronto per essere contattato. La open e' attiva se l'applicativo richiede la connessione.

La finestra scorrevole del TCP opera a livello di byte. Per ogni estremita' di connessione, il software del TCP ha due finestre scorrevoli aperte: una scorre lungo lo stream dei byte trasmessi, un'altra scorre mentre i dati vengono ricevuti.

Dimensione variabile della finestra di controllo. Ogni riscontro contiene un annuncio di finestra che indica il numero di byte che il ricevente e' disposto ad ottenere. Il vantaggio della dimensione variabile e' il **controllo del flusso**: il ricevente puo' annullare la trasmissione dicendosi pronto a ricevere pacchetti di lunghezza zero. Tale controllo e' indispensabile in quanto si hanno elaboratori, reti e gateway con prestazioni differenti. Si ha cosi' il controllo della congestione.

Il formato del segmento TCP.

Porta di provenienza: 16 bit;

Porta di destinazione: 16 bit;

Numero sequenziale: 32 bit. Indica la posizione, nello stream di byte del trasmettitore, dei dati nel segmento;

Numero riscontro: 32 bit. Indica il numero del byte che il ricevente si aspetta successivamente;

HLEN: 4 bit. Indica la lunghezza del segmento misurata in multipli di 32 bit. E' necessario perche' il campo Opzioni ha lunghezza variabile;

Riservato: 6 bit. Per usi futuri;

CODICE: 6 bit. Indica lo scopo ed il contenuto del segmento:

URG: il campo puntatore urgente e' valido;

ACK: il campo riscontro e' valido;

PSH: il segmento richiede un push;

RST: effettua il reset della connessione;

SYN: sincronizza i numeri di sequenza;

FIN: il trasmettitore ha raggiunto la fine della sua stream di byte;

Finestra: 16 bit. Indica la dim. del buffer per i dati;

Checksum: 16 bit;

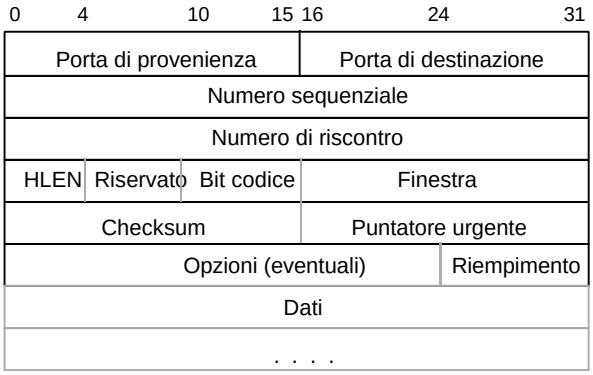
Puntatore urgente: 16 bit. Indica che il pacchetto deve essere processato con urgenza, ad es. perche' include il segnale di abort del programma - con conseguente

soppressione di invio di pacchetti residui -. Si dice che si ha trasmissione fuori banda. Se URG = 1, puntatore urgente specifica la posizione della fine dei dati urgenti nella finestra;

Opzioni (eventuali): 24 bit;

Riempimento: 8 bit;

Dati:



Il formato di un segmento TCP

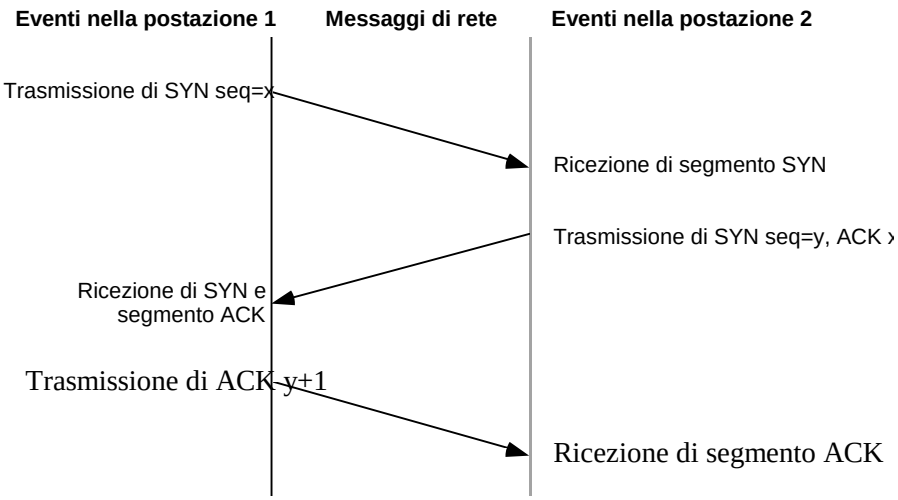
Numeri di sequenza iniziali. Scelti casualmente dal trasmettitore. Il ricevente risponde incrementando quel numero di sequenza.

Attivazione di una connessione TCP. Si ha un protocollo di handshake a tre vie.

Il bit di CODICE relativo a SYN e' attivato.

I dati fluiscono in entrambe le direzioni: non ci sono ne' master ne' salve.

Il protocollo e' a tre vie perche' la trasmissione puo' non essere affidabile.

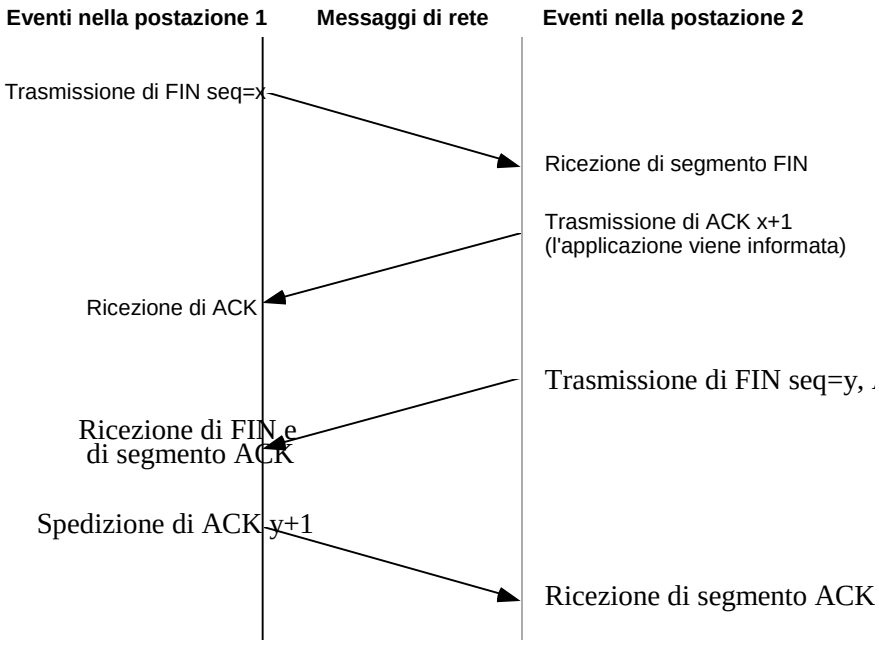


Sequenza di messaggi per handshake a tre vie: attivazione di connessione.

Reset di una connessione TCP.

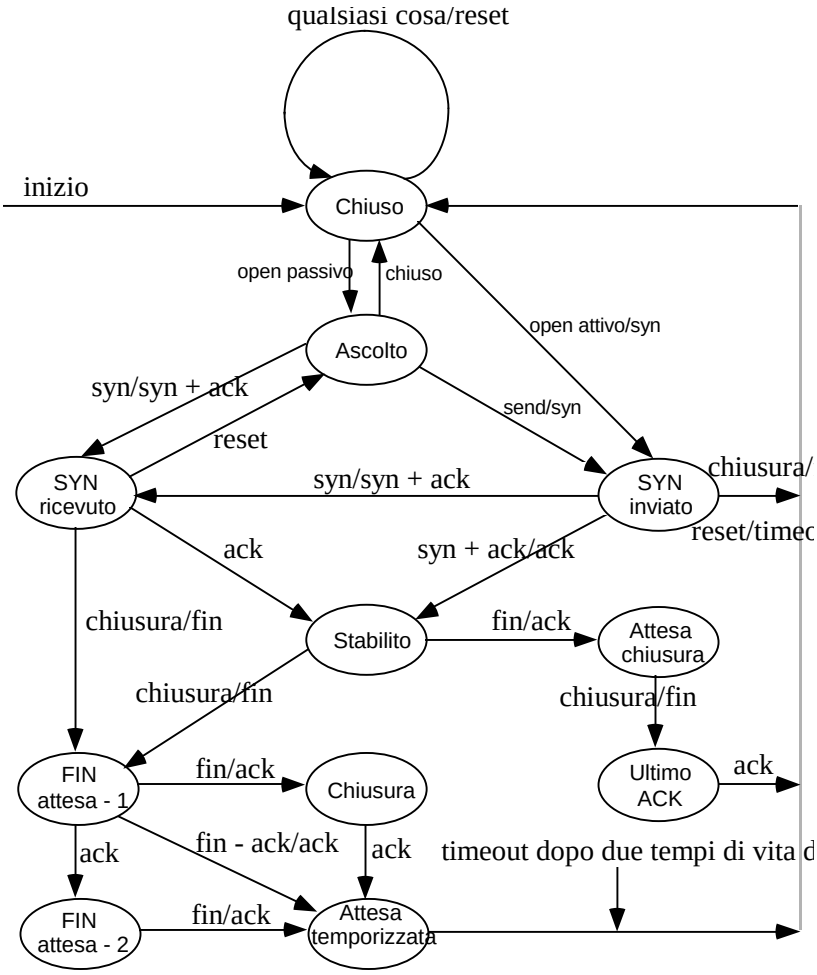
Chiusura di una connessione TCP.

Il bit di CODICE relativo a FIN e' attivato.



Sequenza di messaggi per chiusura di connessione.

La macchina a stati finiti del TCP



Macchina a stati finiti del TCP.

Forzatura della consegna dei dati. Viene emesso un push, attivando il bit di PSH in Codice.

I numeri di porta riservati. Secondo una convenzione assai diffusa, vengono riservate le seguenti porte:

num.	nome		descrizione
0			riservata
1	TCPMUX		multiplexer TCP
5	RJE		remote job entry
7	ECHO	echo	eco
9	DISCARD	discard	scarto
11	USERS	systat	utenti attivi
13	DAYTIME	daytime	ora del giorno
15		netstat	stato della rete
17	QUOTE	quotd	citazione del giorno
19	CHARGEN	chargen	generatore di caratteri
20	FTP-DATA	ftp-data	dati FTP
21	FTP	ftp	FTP
23	TELNET	telnet	connessione di terminale
25	SMTP	smtp	simple mail transport protoc.
37	TIME	time	tempo
42	NAMESERVER	name	server di nomi dell'host
43	NICNAME	whois	chi e'
53	DOMAIN	nameserver	server di nomi del dominio
77		rje	qualunque servizio rje privato
79	FINGER	finger	
93	DCP		device control protocol
95	SUPDUP	supdup	protocollo supdup
101	HOSTNAME	hostnames	server di nomi di host NIC
102	ISO-TSAP	iso-tsap	ISO-TSAP
103	X400	x400	servizio di posta X.400
104	X400SND	x400-snd	invio posta x.400
111	SUNRPC	sunrpc	RPC di sun microsystem
113	AUTH	auth	servizio di autenticazione
117	UUCP-AUTH	uucp-auth	servizio di percorso UUCP
119	NNTP	nntp	protocollo trasferimento news
129	PWDGEN		protocollo di generatore di p.
139	NETBIOS-SSN		servizio di sessione NETBIOS
160-223			riservati

Prestazioni del TCP. Throughput di 8 Mbps tra due workstation collegate con Ethernet a 10 Mbps.

Il Modello di Interazione Client-Server

Definizioni.

Server. Qualsiasi programma che offre un servizio che possa essere raggiunto attraverso la rete. Sono programmi sequenziali ma devono gestire piu' richieste pervenute anche contemporaneamente. Spesso devono poter essere eseguite con il piu' alto privilegio.

Client. Qualsiasi programma che invia una richiesta ad un server ed attende la risposta.

Esempio: server della data e dell'ora.

Rappresentazione della data e dell'ora, come i secondi trascorsi da 0:00:00 del 1-1-1900 (per TCP/IP).

L'Interfaccia di Socket

Obiettivo: specificare l'interfaccia tra programmi applicativi ed il software del protocollo.

Mancanza di uno standard: come esempio, viene scelto il sistema operativo UNIX 4BSD.

Le primitive di I/O UNIX sono del tipo open/read/write/close: con tali primitive non e' molto agevole scrivere il codice che implementa un server. Ulteriore complessita' e' legata alla esigenza di avere software che gestisca differenti tipi di protocolli (ad es. TCP/IP, Xerox XNS).

socket (letteralmente incavo): e' una generalizzazione del meccanismo UNIX per accedere ai file, che fornisce un punto terminale per la comunicazione. Ogni socket e' identificato da un numero intero a 16 bit (short int). La applicazione puo' scegliere di fornire un indirizzo di destinazione ogni volta che utilizza un socket oppure legare un indirizzo ad un socket ed evitare di specificare ogni volta l'indirizzo di destinazione. Le operazioni di read e write utilizzano un descrittore univoco sia tra i socket sia tra i file.

Creazione di un socket. I socket vengono creati su richiesta: alla creazione NON hanno un indirizzo locale o una destinazione e quindi non e' specificato alcun indirizzo IP.

La famiglia dei protocolli family puo' indicare TCP/IP (AF_INET), Xerox (AF_UP), AppleTalk (AF_APPLETALK), UNIX (AF_UNIX).

Il tipo type specifica la comunicazione desiderata: servizio di consegna affidabile (SOCK_STREAM), datagramma senza connessione (SOCK_DGRAM), grezzo (SOCK_RAW).

Il protocollo protocol indica quale protocollo utilizzare se la famiglia specificata ne prevede piu' di uno: e' strettamente legato al tipo di famiglia indicato.

```
/* Create a new socket of type TYPE in domain DOMAIN, using
   protocol PROTOCOL.  If PROTOCOL is zero, one is chosen
   automatically.  Returns a file descriptor for the new socket,
   or -1 for errors.  */
int socket (int family, int type, int protocol);
```

La pipe crea simultaneamente i due punti per la comunicazione. E' costruita con la socketpair, che definisce due socket.

L'array di due interi sockvec include i due descrittori di socket.

```
/* Create two new sockets, of type TYPE in domain DOMAIN and using
   protocol PROTOCOL, which are connected to each other, and put file
   descriptors for them in FDS[0] and FDS[1].  If PROTOCOL is zero,
   one will be chosen automatically.  Returns 0 on success, -1
   for errors.  */
int socketpair (int family, int type, int protocol,
            int sockvec[2]);
```

Chiusura di un socket.

```
close(socket)
```

Specifica di un indirizzo locale. Poiche' i socket alla creazione non hanno associato un indirizzo locale, e' necessario specificarne uno tramite il comando bind. Tale comando associa (bind) ad un socket un indirizzo.

L'indirizzo locale e' specificato da una struct sockaddr. La struttura di sockaddr e' riportata di seguito in figura. Per i servizi TCP/IP, la famiglia di indirizzi richiede il valore numerico 2.

La lunghezza in byte dell'indirizzo e' data da addrlen.

```
/* Give the socket FD the local address ADDR (which is LEN bytes
   long).  */
int bind (int sockfd, const struct sockaddr *my_addr,
            int addrlen);
```

Formato della struct sockaddr.

Creazione di una connessione. Alla creazione un socket non ha un indirizzo associato: di conseguenza, non e' nemmeno possibile associargli una connessione esterna. Per associare ad un socket una connessione esterna, si utilizza il comando connect. Il funzionamento dipende dalla famiglia di protocolli specificati al momento della creazione del socket.

L'indirizzo del destinatario della connessione e' indicato dalla struct sockaddr, la cui lunghezza e' specificata da addrlen.

```
/* Open a connection on socket FD to peer at ADDR (which LEN bytes
   long). For connectionless socket types, just set the default
   address to send to and the only address from which to accept
   transmissions.  Return 0 on success, -1 for errors.  */
int connect (int sockfd, const struct sockaddr *serv_addr,
            int addrlen);
```

Invio dati attraverso un socket. Dopo aver stabilito una connessione, e' possibile inviare dati attraverso un socket utilizzando alcune funzioni. Richiedono l'utilizzo di socket connessi le funzioni send, write, writev perche' tra i parametri non hanno l'indirizzo di destinazione. Le rimanenti funzioni sono sendto, sendmsg.

Il buffer contenente il messaggio da inviare e' indicato da buff, mentre la lunghezza utile del buffer e' indicata da len.

```
/* Write N bytes of BUF to socket FD.  Returns the number sent or
   -1. */
int write (int sockfd, char *buff, size_t len);
```

Il buffer contenente il messaggio da inviare e' indicato da buff, mentre la lunghezza utile del buffer e' indicata da len.

I flag flag indicano alcune informazioni relative all'istradamento, come ad es. se il messaggio deve essere inviato fuori banda (messaggio urgente) o il messaggio deve essere istradato senza usare le tabelle di istradamento locali.

```
/* Send N bytes of BUF to socket FD.  Returns the number sent or
   -1. */
int send (int sockfd, char *buff, size_t len,
             unsigned int flags);
```

Rispetto a send, sendto consente di specificare l'indirizzo del destinatario: cio' e' necessario perche' utilizza un protocollo senza connessione.

```
/* Send N bytes of BUF on socket FD to peer at address ADDR (which
   is ADDR_LEN bytes long).  Returns the number sent, or -1 for
   errors. */
int sendto (int sockfd, char *buff, size_t len,
             unsigned int flags, const struct sockaddr *to,
             int tolen);
```

Del tutto analoga a sendto, sendmsg ha le informazioni necessarie specificate attraverso la struct msghdr.

```
/* Send a message described MESSAGE on socket FD.
   Returns the number of bytes sent, or -1 for errors.  */
extern int sendmsg (int fd, const struct msghdr *message,
                 unsigned int flags);
```

Ricezione dati attraverso un socket. Sono le funzioni duali rispetto a quelle utilizzate per l'invio di dati.

```
/* Read N bytes into BUF from socket FD.
   Returns the number read or -1 for errors.  */
int read (int sockfd, char *buff, size_t len);

/* Read N bytes into BUF from socket FD.
   Returns the number read or -1 for errors.  */
int recv (int sockfd, char *buff, size_t len,
             unsigned int flags);

/* Read N bytes into BUF through socket FD.
   If ADDR is not NULL, fill in *ADDR_LEN bytes of it with the
   address of the sender, and store the actual size of the address in
   *ADDR_LEN. Returns the number of bytes read or -1 for errors. */
int recvfrom (int sockfd, char *buff, size_t len,
```

```
                unsigned int flags, struct sockaddr *from,
                int *fromlen);

/* Receive a message as described by MESSAGE from socket FD.
   Returns the number of bytes read or -1 for errors.  */
extern int recvmsg (int fd, struct msghdr *message,
                    unsigned int flags);
```

Preparazione di un socket per connessioni in arrivo. La funzione listen consente ai server di preparare un socket per le connessioni in arrivo. La lunghezza della coda di richieste per il socket e' indicata da n.

```
/* Prepare to accept connections on socket FD.
   N connection requests will be queued before further requests are
   refused. Returns 0 on success, -1 for errors.  */
int listen (int sockfd, int n);
```

Accettazione di connessione. La funzione accept attende che arrivi una connessione ad un socket. L'indirizzo del socket con il quale si instaura la connessione e' data dalla struct sockaddr.

```
/* Await a connection on socket FD.
   When a connection arrives, open a new socket to communicate with
   it, set *ADDR (which is *ADDR_LEN bytes long) to the address of
   the connecting peer and *ADDR_LEN to the address's actual length,
   and return the new socket's descriptor, or -1 for errors.  */
int accept (int sockfd, const struct sockaddr *peer,
            int *paddrlen);
```

Indirizzi di socket remoti. Per conoscere l'indirizzo di destinazione a cui e' connesso un socket, un software puo' utilizzare le funzioni getpeername (per il socket remoto della comunicazione) e getsockname (per il socket locale).

Il socket sockfd cerca l'indirizzo del socket remoto connesso. L'indirizzo remoto e' specificato da peer, con struttura sockaddr.

```
/* Put the address of the peer connected to socket FD into *ADDR
   (which is *LEN bytes long), and its actual length into *LEN.  */
int getpeername (int sockfd, struct sockaddr *peer,
            int *paddrlen);
```

Il socket sockfd cerca il proprio indirizzo locale specificato da peer, con struct sockaddr.

```
/* Put the local address of FD into *ADDR and its length in *LEN.  */
int getsockname (int sockfd, struct sockaddr *addr,
            int *paddrlen);
```

Definzione delle opzioni. Le funzioni consentono di richiedere o definire le caratteristiche di un socket, accedendo al descrittore del socket stesso.

```
/* Put the current value for socket FD's option OPTNAME at protocol
   level LEVEL into OPTVAL (which is *OPTLEN bytes long), and set
   *OPTLEN to the value's actual length. Returns 0 on success, -1 for
   errors.  */
int getsockopt (int s, int level, int optname,
            void *optval, int *optlen);



/* Set socket FD's option OPTNAME at protocol level LEVEL
   to *OPTVAL (which is OPTLEN bytes long).
   Returns 0 on success, -1 for errors.  */
int setsockopt (int s, int level, int optname,
            const void *optval, int optlen);
```

Listati delle Principali Funzioni di socket.h (dal sistema operativo linux)

```
#ifndef _SYS_SOCKET_H
#define _SYS_SOCKET_H

#include <features.h>
#include <sys/types.h>
#include <linux/socket.h>

#ifdef _MIT_POSIX_THREADS
#include <pthread/mit/posix.h>
#endif

__BEGIN_DECLS

/* struct msghdr is not defined in linux 1.2.  This will allow sendmsg
   and recvmsg in libc 5.2.9 to compile under 1.2.x and shouldn't cause
   any problem for 1.3.x */
struct msghdr;

/* Create a new socket of type TYPE in domain DOMAIN, using
   protocol PROTOCOL.  If PROTOCOL is zero, one is chosen
   automatically.  Returns a file descriptor for the new socket,
   or -1 for errors.  */
int socket __P ((int __family, int __type, int __protocol));

/* Create two new sockets, of type TYPE in domain DOMAIN and using
   protocol PROTOCOL, which are connected to each other, and put file
   descriptors for them in FDS[0] and FDS[1].  If PROTOCOL is zero,
   one will be chosen automatically.  Returns 0 on success, -1
   for errors.  */
int socketpair __P ((int __family, int __type, int __protocol,
            int __sockvec[2]));

/* Give the socket FD the local address ADDR (which is LEN bytes
   long).  */
int bind __P ((int __sockfd, __const struct sockaddr *__my_addr,
```

```
            int __addrlen));

/* Open a connection on socket FD to peer at ADDR (which LEN bytes
   long). For connectionless socket types, just set the default
   address to send to and the only address from which to accept
   transmissions.  Return 0 on success, -1 for errors.  */
int connect __P ((int __sockfd, __const struct sockaddr *__serv_addr,
            int __addrlen));

/* Prepare to accept connections on socket FD.
   N connection requests will be queued before further requests are
   refused. Returns 0 on success, -1 for errors.  */
int listen __P ((int __sockfd, int __n));

/* Await a connection on socket FD.
   When a connection arrives, open a new socket to communicate with it,
   set *ADDR (which is *ADDR_LEN bytes long) to the address of the
   connecting peer and *ADDR_LEN to the address's actual length, and
   return the new socket's descriptor, or -1 for errors.  */
int accept __P ((int __sockfd, __const struct sockaddr *__peer,
            int *__paddrlen));

/* Put the current value for socket FD's option OPTNAME at protocol
   level LEVEL into OPTVAL (which is *OPTLEN bytes long), and set
   *OPTLEN to the value's actual length. Returns 0 on success, -1 for
   errors.  */
int getsockopt __P ((int __s, int __level, int __optname,
            void *__optval, int *__optlen));

/* Set socket FD's option OPTNAME at protocol level LEVEL
   to *OPTVAL (which is OPTLEN bytes long).
   Returns 0 on success, -1 for errors.  */
int setsockopt __P ((int __s, int __level, int __optname,
            __const void *__optval, int optlen));

/* Put the local address of FD into *ADDR and its length in *LEN.  */
int getsockname __P ((int __sockfd, struct sockaddr *__addr,
            int *__paddrlen));

/* Put the address of the peer connected to socket FD into *ADDR
   (which is *LEN bytes long), and its actual length into *LEN.  */
int getpeername __P ((int __sockfd, struct sockaddr *__peer,
            int *__paddrlen));

/* Send N bytes of BUF to socket FD.  Returns the number sent or -1. */
int send __P ((int __sockfd, __const void *__buff, size_t __len,
             unsigned int __flags));

/* Read N bytes into BUF from socket FD.
   Returns the number read or -1 for errors.  */
int recv __P ((int __sockfd, void *__buff, size_t __len,
             unsigned int __flags));

/* Send N bytes of BUF on socket FD to peer at address ADDR (which is
   ADDR_LEN bytes long).  Returns the number sent, or -1 for errors. */
int sendto __P ((int __sockfd, __const void *__buff, size_t __len,
             unsigned int __flags, __const struct sockaddr *__to,
             int __tolen));

/* Read N bytes into BUF through socket FD.
   If ADDR is not NULL, fill in *ADDR_LEN bytes of it with tha address
   of the sender, and store the actual size of the address in
   *ADDR_LEN. Returns the number of bytes read or -1 for errors. */
int recvfrom __P ((int __sockfd, void *__buff, size_t __len,
             unsigned int __flags, struct sockaddr *__from,
             int *__fromlen));
```

```
/* Send a message described MESSAGE on socket FD.
   Returns the number of bytes sent, or -1 for errors.  */
extern int sendmsg __P ((int __fd, __const struct msghdr *__message,
                         unsigned int __flags));

/* Receive a message as described by MESSAGE from socket FD.
   Returns the number of bytes read or -1 for errors.  */
extern int recvmsg __P ((int __fd, struct msghdr *__message,
                         unsigned int __flags));

/* Shut down all or part of the connection open on socket FD.
   HOW determines what to shut down:
      0 = No more receptions;
      1 = No more transmissions;
      2 = No more receptions or transmissions.
   Returns 0 on success, -1 for errors.  */
int shutdown __P ((int __sockfd, int __how));


/* belongs here or elsewhere? */
int rcmd __P ((char **__ahost, unsigned short __inport,
               __const char *__locuser, __const char *__remuser,
               __const char *__cmd, int *__fd2p));
int rresvport __P ((int *__port));
int ruserok __P ((__const char *__rhost, int __superuser,
               __const char *__ruser, __const char *__luser));
int rexec __P ((char **__ahost, int __inport, __const char *__user,
                __const char *__passwd, __const char *__cmd,
                int *__fd2p));

__END_DECLS

#endif /* _SYS_SOCKET_H */
```

Bibliografia

Comer, Internetworking con TCP/IP, Gruppo Editoriale Jackson, 1992.

Goldberg Lee, "The Internet in Space: Problems and Solutions", IEEE Computer, Febr. 1997, pag. 15 - 16