

# Calcolatori Elettronici

## Parte VII: Il linguaggio assemblativo 8088 (basato su materiale di M. Di Felice)

Prof. Riccardo Torlone  
Università Roma Tre

# Linguaggi assemblativi

## Il linguaggio assemblativo (assembly)

- Rappresentazione simbolica dell'insieme di istruzioni macchina di un'architettura.
- Associa ai dati nomi simbolici che identificano le corrispondenti posizioni in memoria
- Nasconde i dettagli relativi alle singole istruzioni (codici operativi, formati, ecc.), non quelli relativi all'architettura della macchina
- Fornisce un set di istruzioni (direttive) che facilitano la traduzione in linguaggio macchina

# Perché imparare un linguaggio assemblativo?

- Per scrivere routine di sistema operativo? **NO**
  - Si scrive in C (o in C++)
- Per scrivere codice ottimizzato? **Solo in parte**
  - Impossibile battere i compilatori
  - Solo localmente abbiamo qualche chance
  - Regola 90%-10%
- Per conoscere meglio il calcolatore? **SI!**
  - Per imparare l'assembler bisogna conoscere l'architettura
  - Il debug dell'assembler ci fa comprendere come funziona l'architettura

# Assembler "embedded"

```
#include <stdio.h>

void main(void) {
    static int x = 3;

    asm{
        MOV %EAX, x
        ADD %EAX, x
        ADD %EAX, x
        NOP
        MOV x, %EAX
    }

    printf("%2u",x);
    return;
}
```

# Assemblatore e Tracer

## Assemblatore

- Programma che riceve in ingresso un programma in linguaggio assemblativo e genera un programma in linguaggio macchina (binario) pronto per essere eseguito dall'hardware.

## Tracer (interprete)

- Simulatore dell'esecuzione di un programma scritto in un linguaggio assemblativo
- Consente di procedere "passo-passo"
- Debugger per l'Assembler

# Assembly e linguaggio macchina

```
int void () {  
    int i;  
    int cont=0;  
    for (i=0; i<100; i++)  
        cont=cont+i;  
    return 1;  
}
```

Programma scritto in un  
Linguaggio Ad Alto Livello (C)

```
.....  
00 00 00 00 00 00 00 00 1b 00 00 00 01 00 00 00  
06 00 00 00 00 00 00 00 34 00 00 00 46 00 00 00  
00 00 00 00 00 00 00 04 00 00 00 00 00 00 00 00  
21 00 00 00 01 00 00 00 03 00 00 00 00 00 00 00  
7c 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
04 00 00 00 00 00 00 00 27 00 00 00 08 00 00 00  
03 00 00 00 00 00 00 00 7c 00 00 00 00 00 00 00  
00 00 00 00 00 00 00 04 00 00 00 00 00 00 00 00  
2c 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00  
7c 00 00 00 2d 00 00 00 00 00 00 00 00 00 00 00  
01 00 00 00 00 00 00 00 35 00 00 00 01 00 00 00  
00 00 00 00 00 00 00 00 a9 00 00 00 00 00 00 00  
00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00  
11 00 00 00 03 00 00 00 00 00 00 00 00 00 00 00  
a9 00 00 00 45 00 00 00 00 00 00 00 00 00 00 00  
.....
```

Programma in Linguaggio  
Macchina

compilazione

main:

```
pushl %ebp  
movl %esp, %ebp  
subl $24, %esp  
andl $-16, %esp  
movl $0, %eax  
addl $15, %eax  
addl $15, %eax  
shrl $4, %eax  
sall $4, %eax  
subl %eax, %esp  
movl $0, -4(%ebp)  
movl $0, -8(%ebp)  
jmp .L2  
.L3:  
    movl -8(%ebp), %eax  
    leal -4(%ebp), %edx  
    addl %eax, (%edx)  
    leal -8(%ebp), %eax  
    incl (%eax)  
.L2:  
    cmpl $99, -8(%ebp)  
    jle .L3  
    movl $0, %eax  
    leave  
    ret
```

Programma in Linguaggio  
Assembly (386/NASM)

assemblaggio

# Il Tracer

```

_EXIT T = 1      ! 1
_WRITE  = 4      ! 2
_STDOUT = 1      ! 3
.SECT .TEXT       ! 4
start:           ! 5
    MOV CX,de-hw ! 6
    PUSH CX       ! 7
    PUSH hw        ! 8
    PUSH _STDOUT   ! 9
    PUSH _WRITE    !10
    SYS            !11
    ADD SP, 8      !12
    SUB CX,AX     !13
    PUSH CX        !14
    PUSH _EXIT    !15
    SYS            !16
.SECT .DATA       !17
hw:              !18
.ASCII "Hello World\n" !19
de: .BYTE 0       !20

```

CS: 00 DS=SS=ES: 002	AH:00 AL:0c AX: 12	BH:00 BL:00 BX: 0	CH:00 CL:0c CX: 12	DH:00 DL:00 DX: 0	SP: 7fd8 SF O D S Z C =>0004	BP: 0000 CC - > p - - 0001 =>	SI: 0000 IP:000c:PC 0000	DI: 0000 start + 7 000c	MOV CX,de-hw ! 6	PUSH CX ! 7	PUSH HW ! 8	PUSH _STDOUT ! 9	PUSH _WRITE ! 10	SYS ! 11	ADD SP,8 ! 12	SUB CX,AX ! 13	PUSH CX ! 14
									E	I							
hw																	
■																	
hw + 0 = 0000: 48 65 6c 6c 6f 20 57 6f Hello World 25928																	

(a)

(b)

- (a) Un programma in linguaggio assemblativo  
(b) Il tracer in esecuzione sul programma

# Assembler 8088 / IA32 / x86

- x86: famiglia di ISA della famiglia Intel. Varie estensioni:
  - 8086 → ... → 80386 → ... → Pentium IV → ... → Core i7
- x86-32 (IA32): linguaggio macchina dei processori x86 a 32 bit
- Versione moderna: x86-64 estensione a 64 bit dell'x86
- Faremo riferimento ad una delle prime versioni: **8088**
  - versione semplificata di un microprocessore Intel moderno
  - il relativo codice assembler può essere eseguito anche su i microprocessori correnti
- Caratteristiche principali
  - Architettura a 16 bit
  - Address bus: 20 bit (1 MB di RAM)
  - Data bus: 8 bit
  - L'unità minima indirizzabile: 1 byte
  - Range di indirizzi: [00000:FFFFF]

# I registri generali

General registers			
AX	AH	AL	
BX	BH	BL	
CX	CH	CL	
DX	DH	DL	
	15	8	7
			0

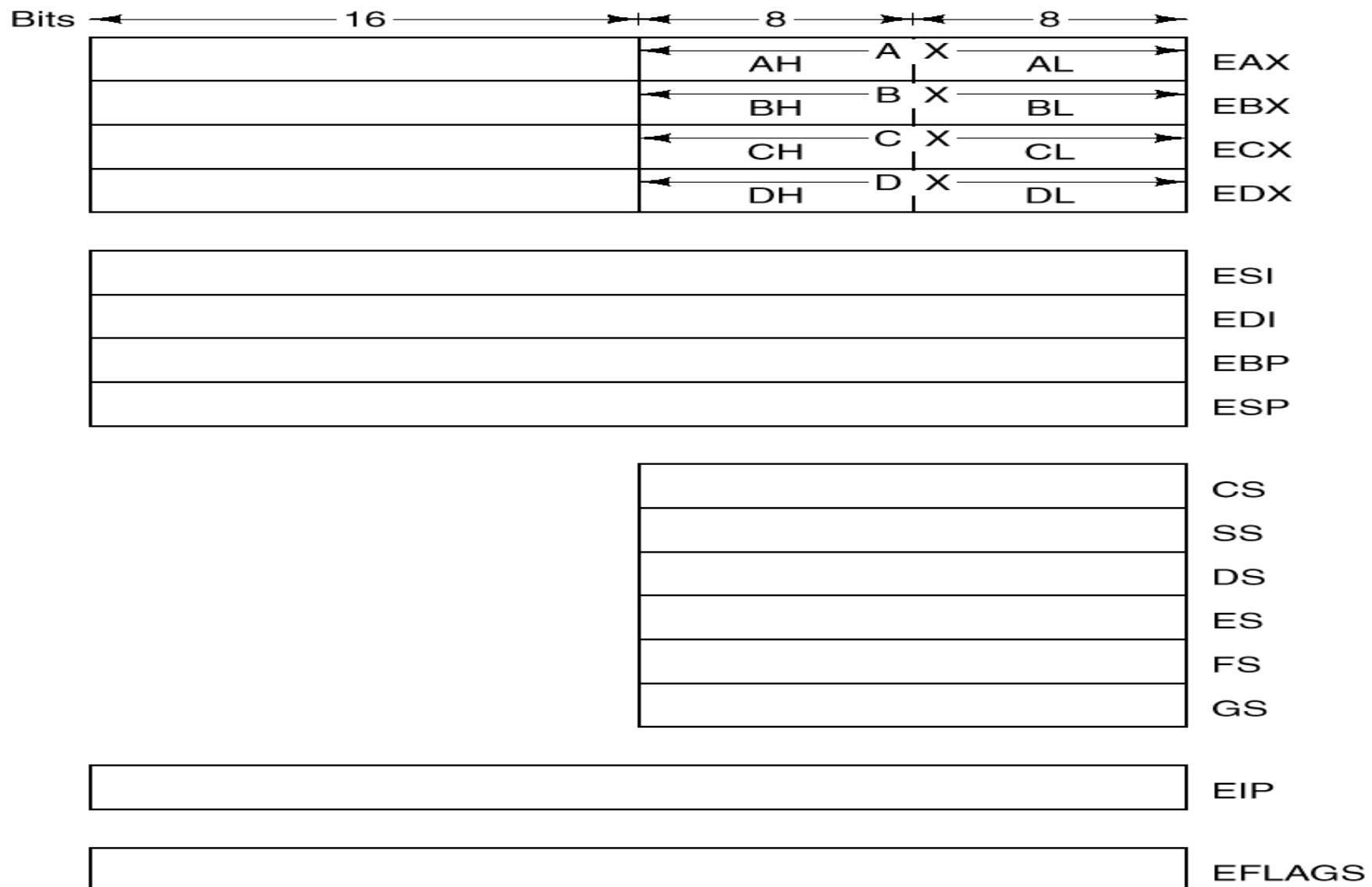
Segment registers			
CS	Code segment		
DS	Data segment		
SS	Stack segment		
ES	Extra segment		
	15		0

Pointer and index			
SP	Stack pointer		
BP	Base pointer		
SI	Source index		
DI	Destination index		
	15		0

Condition codes							
SF	ODI	TSZ	A	P	C	CC	
	15	Status flags			0		
Instruction pointer							IP
Program counter							PC
	15						0

Disponibili 14 registri, suddivisi in 4 gruppi funzionali

# Confronto con i registri del Core i7



# I registri di uso generale

- AX: registro accumulatore, usato per memorizzare il risultato dell'elaborazione e come destinazione di molte istruzioni (a volte implicitamente)

Esempio: ADD AX,20

- BX: registro base, usato come accumulatore o come puntatore alla memoria

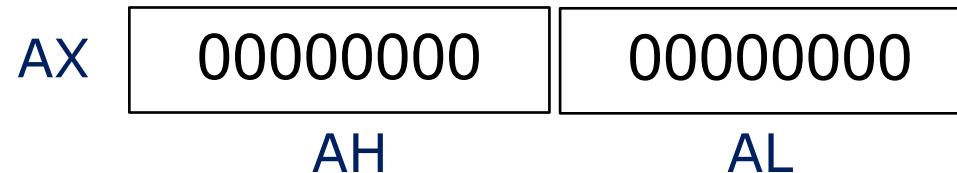
Esempio: MOV AX,(BX)

- CX: registro contatore, usato come contatore dei cicli
- DX: registro dati, usato insieme ad AX per contenere le istruzioni lunghe due parole (32 bit)

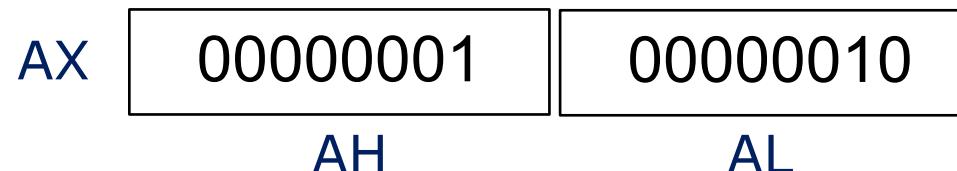
- DX:AX

# Registri a 8 e a 16 bit

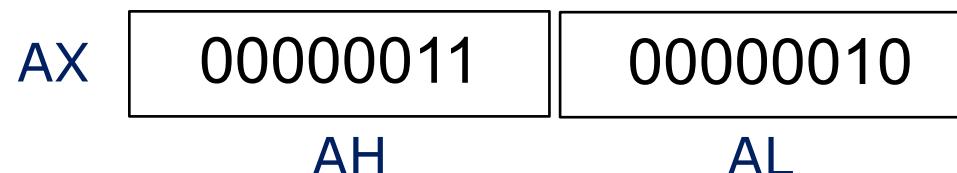
Tutti i registri possono essere visti come coppie di registri di 8 bit accessibili autonomamente (esempio: AX=AH:AL)



MOVE AX,258



ADD AH,AL



AX=770

# I registri puntatore ed indice

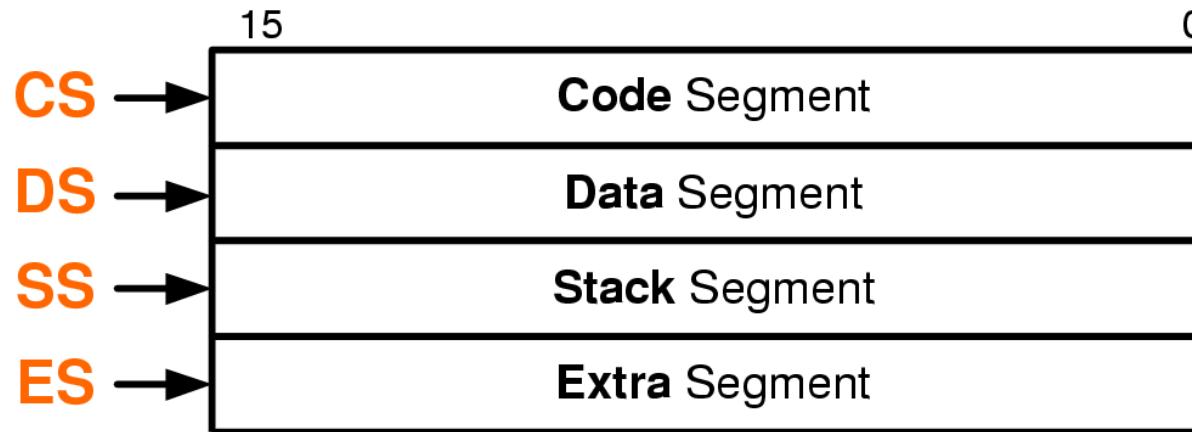
- **SP**: registro puntatore alla cima dello stack.
  - Viene modificato automaticamente dalle operazioni sullo stack (PUSH, POP).
- **BP**: registro puntatore base dello stack.
  - Punta alla base del frame (record di attivazione) assegnato alla procedura corrente
- **SI**: registro indice sorgente
  - usato in combinazione con BP per riferirsi a dati sullo stack o con BX per localizzare dati in memoria.
- **DI**: registro indice destinazione
  - usato come SI

# Registro di stato

- Il registro di stato (flag) è un insieme di registri da 1 bit.
- I bit sono impostati da istruzioni aritmetiche:
  - **Z** - il risultato è zero
  - **S** - il risultato è negativo (bit di segno)
  - **O** - il risultato ha causato un overflow
  - **C** - il risultato ha generato un riporto
  - **A** - riporto ausiliario (oltre il bit 3)
  - **P** - parità del risultato
- Gli altri bit del registro controllano alcuni aspetti dell'attività del processore
  - **I** = attiva gli interrupt
  - **T** = abilita il tracing
  - **D** = operazioni su stringhe
- Non tutti i bit sono utilizzati

# Segmenti e registri di segmento

- Lo spazio di memoria indirizzabile dalla CPU è suddiviso in segmenti logici. Ogni segmento è costituito da 65.536 byte consecutivi.
- Quattro registri di segmento puntano ai quattro segmenti correntemente attivi.



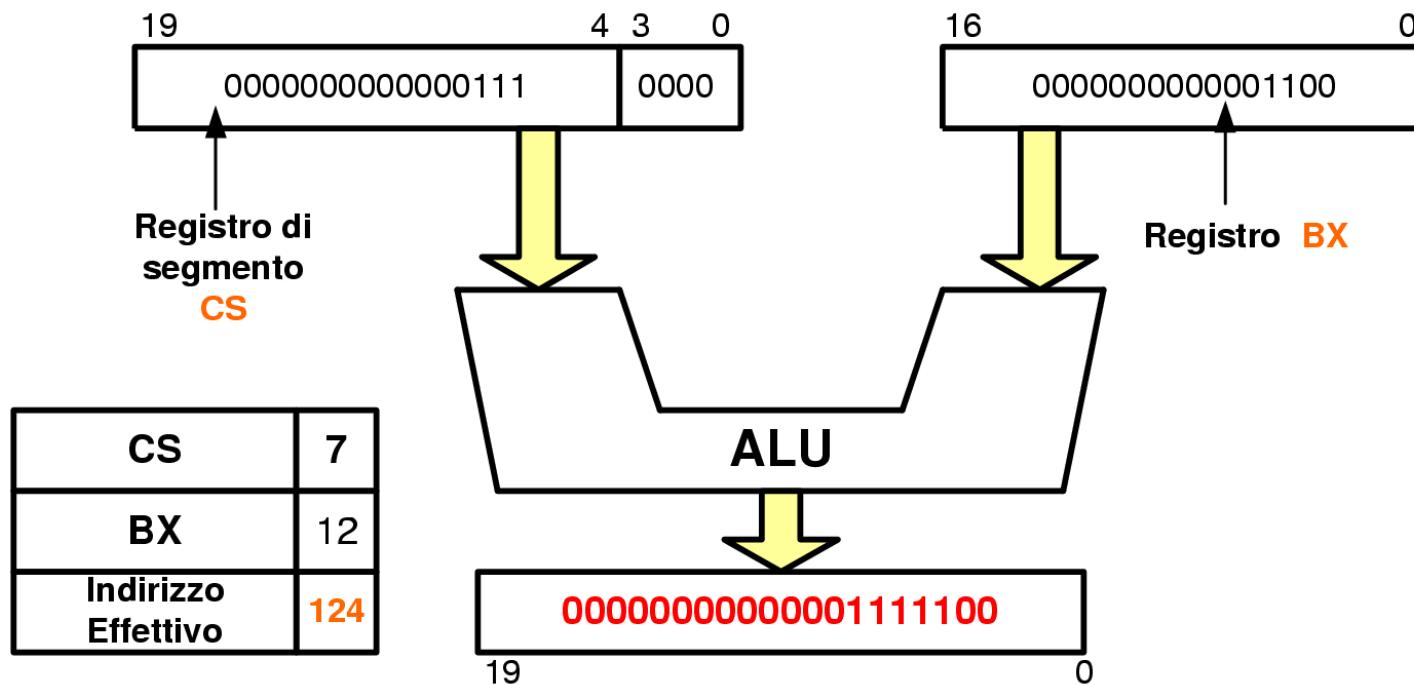
- **CS** punta al segmento contenente le istruzioni da eseguire
- **DS** punta al segmento contenente le variabili del programma
- **SS** punta al segmento contenente lo stack corrente
- **ES** punta al segmento extra, usato tipicamente per i dati

# Segmentazione della memoria

- Lo spazio di memoria viene visto come un gruppo di segmenti
- Ogni segmento:
  - Costituisce un'unità di memoria indipendente
  - È formata da locazioni contigue di memoria
  - Ha un limite massimo di 64KB
  - Inizia ad un indirizzo di memoria multiplo di 16
- Ogni riferimento alla memoria richiede l'intervento di un registro di segmento per la costruzione di un indirizzo fisico:  
Indirizzo Effettivo: <segmento::offset>

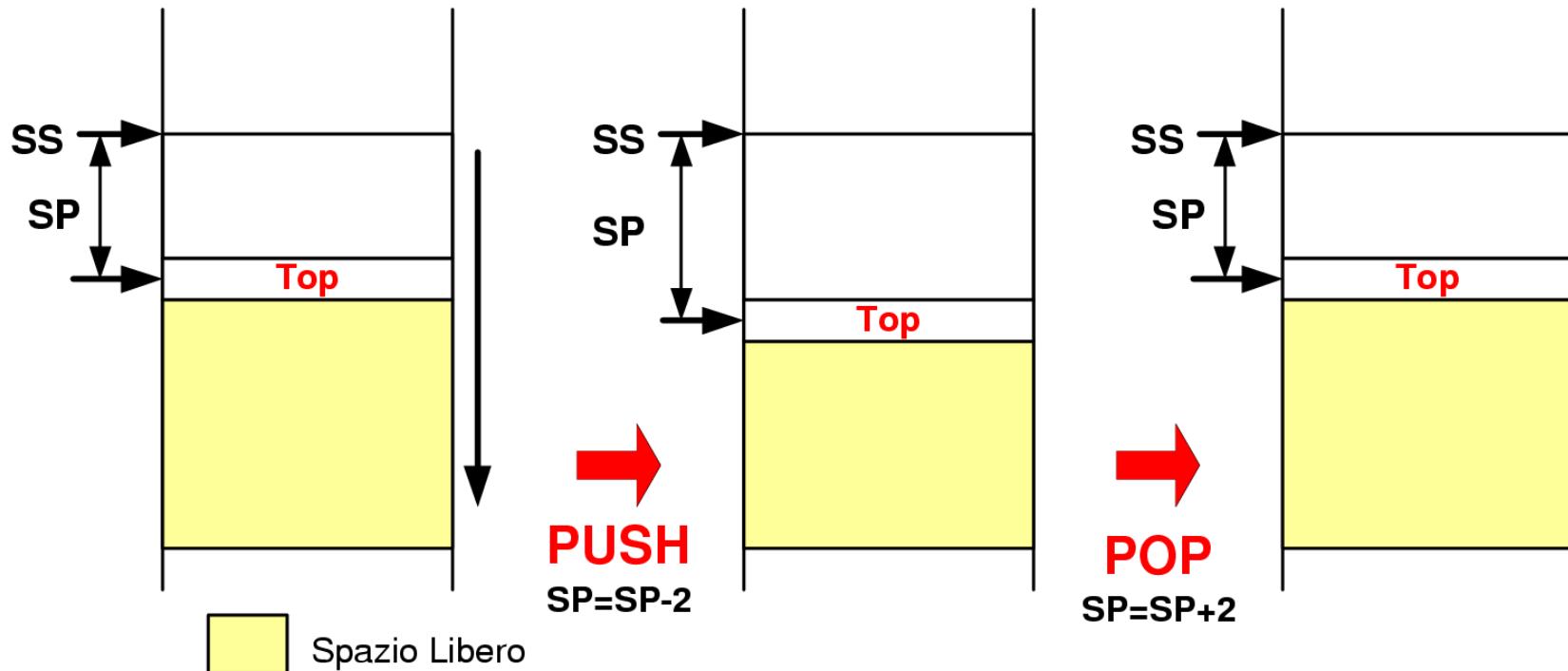
# Costruzione indirizzo fisico

1. Si considera il registro di segmento corrispondente
2. Si aggiungono 4 zero a destra ( $\times 16$ )
  - indirizzo a 20 bit
3. Si somma l'offset all'indirizzo da 20 bit



# Segmento per la gestione dello stack

- Il segmento di stack è costituito da parole di 2 byte
- Lo stack cresce andando dagli indirizzi alti a quelli bassi
- SS punta all'indirizzo di partenza dello stack
- SP punta alla locazione in cima allo stack



# Indirizzamento immediato e a registro

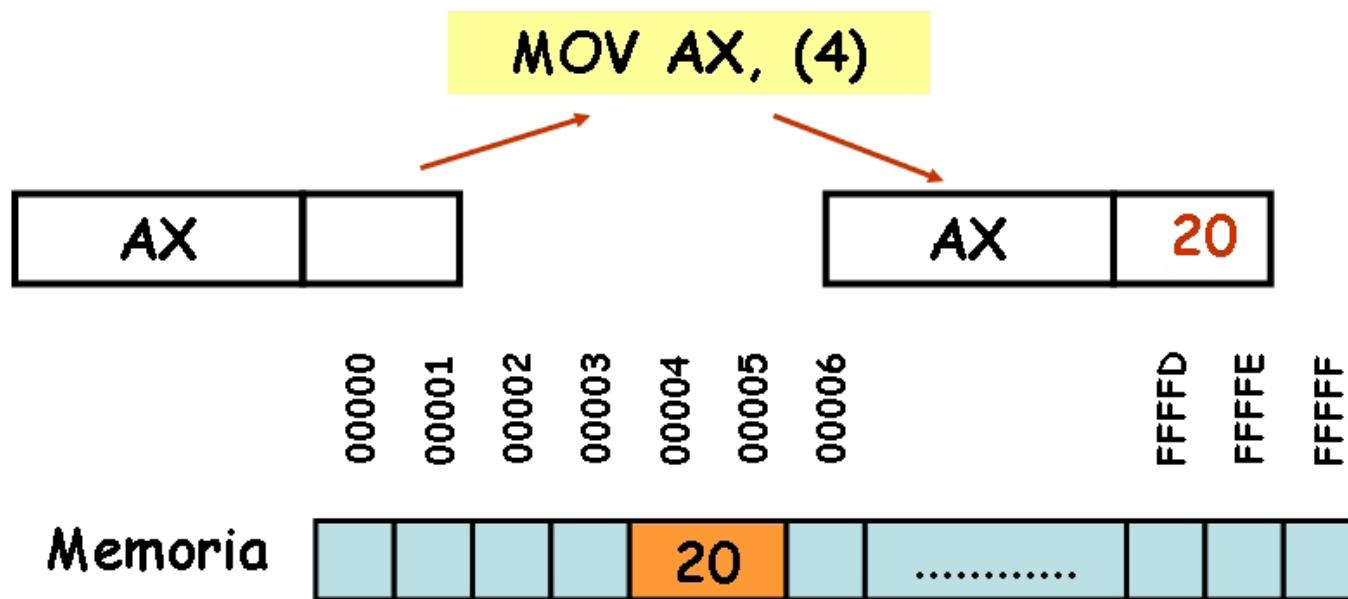
- Indirizzamento a registro
  - L'operando si trova nei registri, e non è necessario accedere alla memoria

Esempio: CX=5 → MOV AX,CX → **AX=5**
- Indirizzamento immediato
  - L'operando è contenuto nell'istruzione.
  - Il dato può essere una costante di 8 o 16 bit

Esempio: MOV AX,5 → **AX=5**

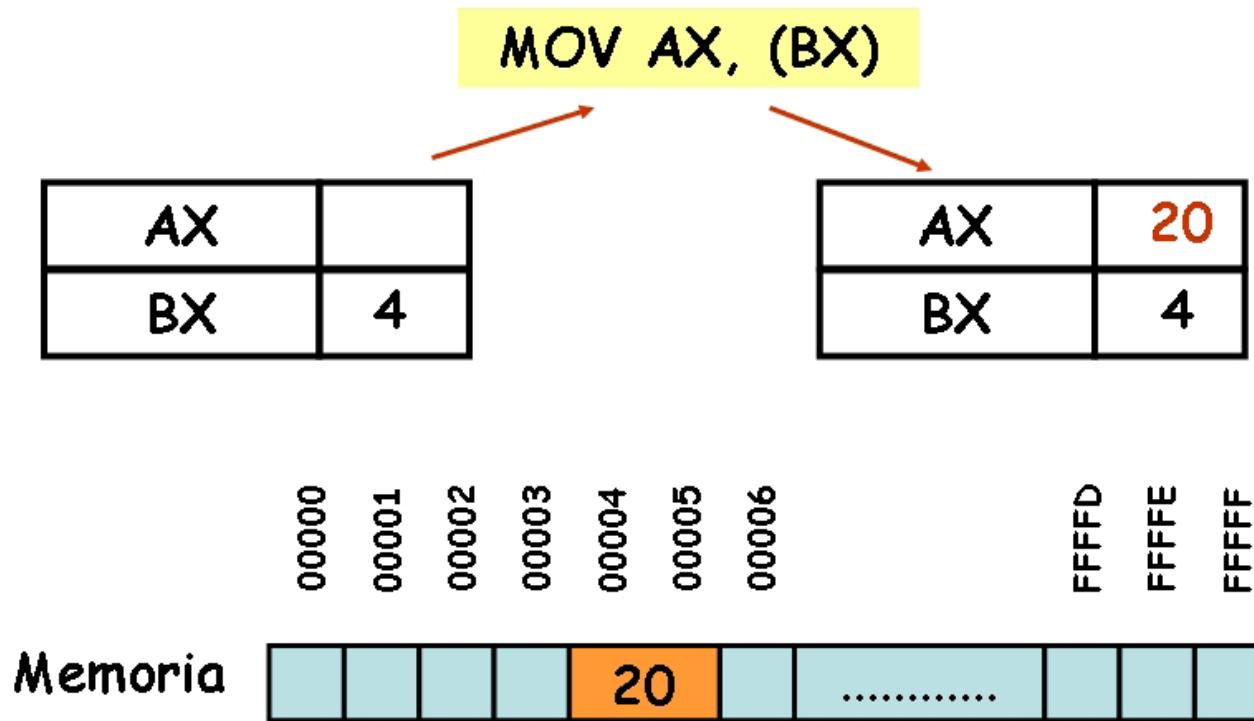
# Indirizzamento diretto

L'istruzione contiene l'indirizzo dei dati nell'operando stesso



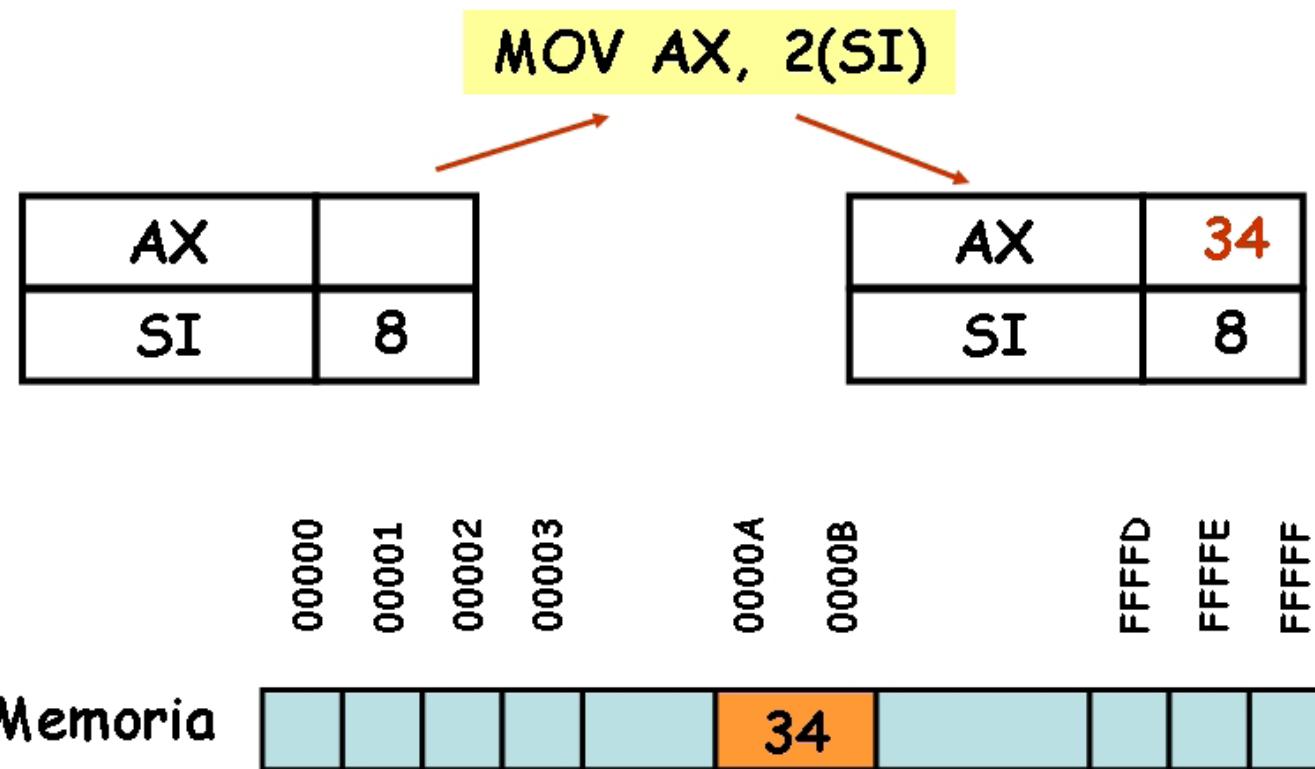
# Indirizzamento indiretto a registro

L'indirizzo dell'operando è memorizzato in uno dei registri BX, SI o DI



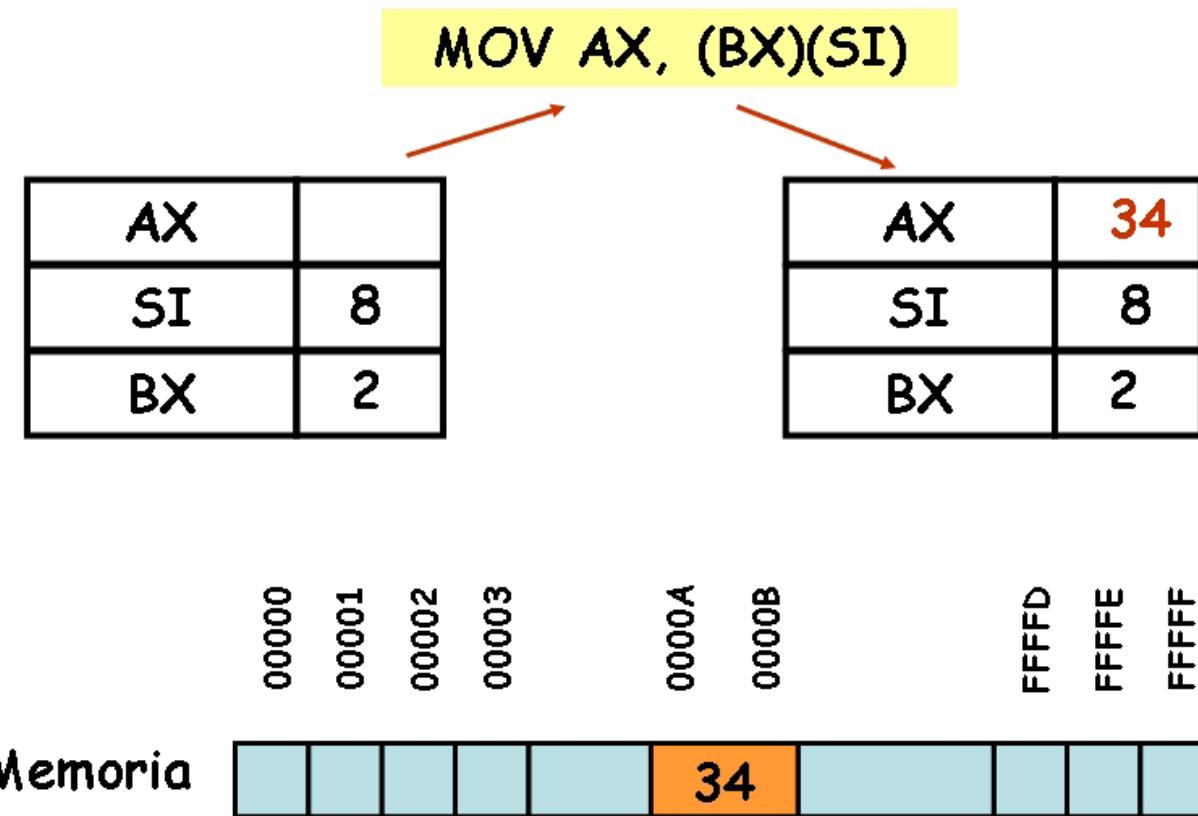
# Indirizzamento indiretto a registro con spiazzamento

L'indirizzo si ottiene dalla somma di uno dei registri BX, SI o DI ed una costante



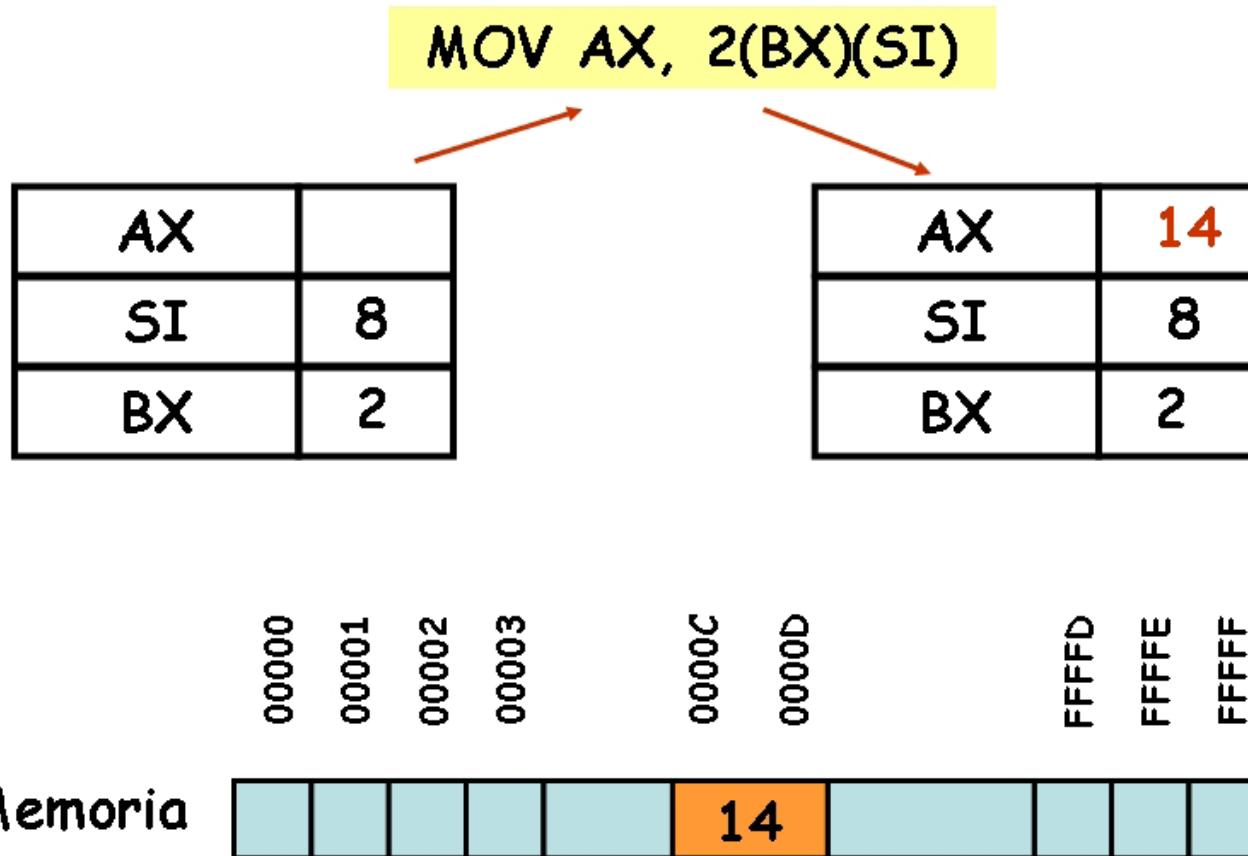
# Indirizzamento a registro indice

L'indirizzo si ottiene dalla somma dei registri SI o DI e BX



# Indirizzamento a registro indice con spiazzamento

L'indirizzo si ottiene dalla somma dei registri SI o DI, BX ed una costante.



# Modalità di indirizzamento

Mode	Operand	Examples
<b>Register addressing</b> Byte register Word register	Byte register Word register	AH, AL, BH, BL, CH, CL, DH, DL AX, BX, CX, DX, SP, BP, SI, DI
<b>Data segment addressing</b> Direct address Register indirect Register displacement Register with index Register index displacement	Address follows opcode Address in register Address is register+displ. Address is BX + SI/DI BX + SI DI + displacement	(#) (SI), (DI), (BX) #(SI), #(DI), #(BX) (BX)(SI), (BX)(DI) #(BX)(SI), #(BX)(DI)
<b>Stack segment address</b> Base Pointer indirect Base pointer displacement Base Pointer with index Base pointer index displ.	Address in register Address is BP + displ. Address is BP + SI/DI BP+SI/DI + displacement	(BP) #(BP) (BP)(SI), (BP)(DI) #(BP)(SI), #(BP)(DI)
<b>Immediate data</b> Immediate byte/word	Data part of instruction	#
<b>Implied address</b> Push/pop instruction Load/store flags Translate XLAT Repeated string instructions In / out instructions Convert byte, word	Address indirect ( SP ) status flag register AL, BX (SI), (DI), (CX) AX, AL AL, AX, DX	PUSH, POP, PUSHF, POPF LAHF, STC, CLC, CMC XLAT MOVS, CMPS, SCAS IN #, OUT # CBW,CWD

#=operando immediato

# L'assemblatore as88

Disponibile presso:

- CD-ROM allegato al libro di testo del corso
- <ftp://ftp.cs.vu.nl/pub/evert/>
- Sito Web del corso

Il tool comprende:

- Programma assemblatore (as88)
  - Utilizzo Generale: as88 Nomeprogetto(.s)
- Emulatore-Interprete dell'architettura 8088 (s88)
  - Utilizzo Generale: s88 Nomeprogetto
- Programma tracer per il debugging (t88)
  - Utilizzo Generale: t88 Nomeprogetto(.\\$)

# Direttive del compilatore

- Ogni programma assembly è strutturato in 3 sezioni:
  1. sezione TESTO (direttiva: .SECT .TEXT): contiene le istruzioni del programma
  2. sezione DATI (direttiva: .SECT .DATA): alloca spazio nel segmento DATI per i dati (inizializzati)
  3. sezione BSS (direttiva: .SECT .BSS): alloca spazio nel segmento DATI per i dati (non inizializzati)
- E' possibile definire etichette di due tipi:
  - **globali:** identificatori alfanumerici seguiti dal simbolo ":" (possono occupare una intera riga)
  - **locali:** utilizzabili solo nel segmento TESTO, costituite da una sola cifra seguita dal simbolo ":".

# Vincoli sulle etichette

- Le etichette globali DEVONO essere univoche
  - Es: .SECT .DATA hw: .ASCII "Hello"
- Le etichette locali possono occorrere più volte
  - Es. JMP 1f
  - Salto verso la prossima etichetta denominata "1"
- E' possibile attribuire nomi simbolici alle costanti mediante la sintassi: identificatore=espressione
  - Es. BLOCKSIZE=1024
- I valori numerici possono essere:
  - ottali (cominciano per zero),
  - decimali,
  - esadecimali (cominciano per 0x)
- I commenti iniziano con il carattere "!"

# Direttive del compilatore

<b>Instruction</b>	<b>Description</b>
.SECT .TEXT	Assemble the following lines in the TEXT section
.SECT .DATA	Assemble the following lines in the DATA section
.SECT .BSS	Assemble the following lines in the BSS section
.BYTE	Assemble the arguments as a sequence of bytes
.WORD	Assemble the arguments as a sequence of words
.LONG	Assemble the arguments as a sequence of longs
.ASCII "str"	Store str as ascii an string without a trailing zero byte
.ASCIZ "str"	Store str as ascii an string with a trailing zero byte
.SPACE n	Advance the location counter n positions
.ALIGN n	Advance the location counter up to an n-byte boundary
.EXTERN	Identifier is an external name

# The Tracer (debugger)

Processor with registers	Stack	Program text Source file
Subroutine call stack		Error output field Input field Output field
Interpreter commands		
Values of global variables Data segment		

Il tracer consente di effettuare l'esecuzione step-by-step del programma e di monitorare lo stato di registri/memoria

# Uso dei registri con il tracer

```
start:          ! 3
    MOV AX,258   ! 4
    ADDB AH,AL   ! 5
    MOV CX,(times) ! 6
    MOV BX,muldat ! 7
    MOV AX,(BX)   ! 8
    llp: MUL 2(BX) ! 9
    LOOP llp      ! 10
.SECT .DATA      ! 11
times: .WORD 10    ! 12
muldat :.WORD 625,2 ! 13
```

(a)

```
CS: 00 DS=SS=ES002
AH:03 AL:02 AX: 770
BH:00 BL:02 BX: 2
CH:00 CL:0a CX: 10
DH:00 DL:00 DX: 0
SP: 7fe0 SF O D S Z C
BP: 0000 CC - > p -
SI: 0000 IP:0009:PC
DI: 0000 start + 4
```

(b)

```
CS: 00 DS=SS=ES002
AH:38 AL:80 AX: 14464
BH:00 BL:02 BX: 2
CH:00 CL:04 CX: 4
DH:00 DL:01 DX: 1
SP: 7fe0 SF O D S Z C
BP: 0000 CC v > p - c
SI: 0000 IP:0011:PC
DI: 0000 start + 7
```

(c)

(a) Parte del programma

(b) I registri dopo l'esecuzione di 7 righe

(c) I registri dopo l'esecuzione di 6 iterazioni del ciclo

# The ACK-Based Assembler, as88

Escape symbol	Description
\n	New line (line feed)
\t	Tab
\\\	Backslash
\b	Back space
\f	Form feed
\r	Carriage return
\"	Double quote

Valori di escape consentiti nell'as88.

# Comandi del tracer (1)

Address	Command	Example	Description
			Execute one instruction
#	, !, X	24	Execute # instructions
/T+#	g , !,	/start+5g	Run until line # after label T
/T+#	b	/start+5b	Put breakpoint on line # after label T
/T+#	c	/start+5c	Remove breakpoint on line # after label T
#	g	108g	Execute program until line #
	g	g	Execute program until current line again
	b	b	Put breakpoint on current line
	c	c	Remove breakpoint on current line

E' possibile interagire con il tracer:

- in modalità batch (fornendo in input un file con i comandi del tracer)
- in modalità interattiva (inserendo comandi da tastiera seguiti dal tasto INVIO)

# Comandi del tracer (2)

<b>Address</b>	<b>Command</b>	<b>Example</b>	<b>Description</b>
	n	n	Execute program until next line
	r	r	Execute until breakpoint or end
	=	=	Run program until same subroutine level
	-	-	Run until subroutine level minus 1
	+	+	Run until subroutine level plus 1
/D+#		/buf+6	Display data segment on label+#
/D+#	d , !	/buf+6d	Display data segment on label+#
	R , CTRL L	R	Refresh windows
	q	q	Stop tracing, back to command shell

# Chiamate di sistema

- Le **chiamate di sistema** consentono di utilizzare le procedure fornite dal sistema operativo.
- Le routine di sistema possono essere attivate con la sequenza di chiamata standard:
  - Si impilano gli argomenti sullo stack
  - Si impila il numero di chiamata
  - Si esegue l'istruzione SYS
- I risultati sono restituiti nel registro AX o nella combinazione di registri AX:DX (se il risultato è di tipo long)
- Gli argomenti sullo stack devono essere rimossi dalla funzione chiamante

# Chiamate di sistema in as88 (1)

L'interprete 8088 supporta 12 chiamate di sistema.

- **\_OPEN**: Apre il file *name* in lettura-scrittura

Identificativo chiamata: **5**

Argomenti: \*name, 0=lettura/1=scrittura/2=lettura-scrittura;

Valore Ritorno: un descrittore di file (fd)

- **\_CREAT**: Crea un nuovo file di nome *name*

Identificativo chiamata: **8**

Argomenti: \*name, \*mode = permessi UNIX;

Valore Ritorno: un descrittore di file (fd)

- **\_READ**: Legge *n* byte da un file con descrittore *fd* trasferendoli nel buffer *buf*

Identificativo chiamata: **3**

Argomenti: fd, buf, n;

Valore Ritorno: numero di byte letti correttamente

## Chiamate di sistema in as88 (2)

- **\_WRITE**: Scrive *n* byte sul file con descrittore *fd* prelevandoli dal buffer *buf*  
Identificativo chiamata: **4**  
Argomenti: *fd*, *buf*, *n*;  
Valore Ritorno: numero di byte scritti correttamente
- **\_CLOSE**: Chiude un file precedentemente aperto  
Identificativo chiamata: **6**  
Argomenti: *fd* (descrittore di file)  
Valore Ritorno: 0 se l'operazione ha successo
- **\_LSEEK**: Sposta il puntatore del file con descrittore *fd* di *offset* bytes  
Identificativo chiamata: **19**  
Argomenti: *fd*, *offset*, 0/1/2;  
Valore Ritorno: nuova posizione all'interno del file
- **\_EXIT**: Interrompe un processo  
Identificativo chiamata: **1;**  
Argomenti: 0=successo/1=errore;

## Chiamate di sistema in as88 (3)

- **\_GETCHAR**: Legge un carattere dallo standard input  
Identificativo chiamata: **117**  
Valore ritorno: il carattere letto e posto in AL
- **\_PUTCHAR**: Scrive un carattere sullo standard output  
Identificativo chiamata: **122**  
Argomenti: carattere da scrivere
- **\_PRINTF**: Stampa una stringa formattata sullo standard output  
Identificativo chiamata: **127**  
Argomenti: stringa di formato, argomenti
- **\_SSCANF**: Legge gli argomenti dal buffer *buf*  
Identificativo chiamata: **125**  
Argomenti: *buf*, stringa di formato, argomenti
- **\_SPRINTF**: Stampa una stringa formattata sul buffer *buf*  
Identificativo chiamata: **121**  
Argomenti: *buf*, stringa di formato, argomenti

# Primo esempio

```
!calcolo di (a+3)*b
```

```
    _EXIT = 1
```

```
.SECT .TEXT
```

```
start:
```

```
    MOV AX,(a)
```

```
    ADD AX,3
```

```
    MUL (b)
```

```
    PUSH 0
```

```
    PUSH _EXIT
```

```
    SYS
```

```
.SECT .DATA
```

```
a:    .WORD 5
```

```
b:    .WORD 3
```

```
.SECT .BSS
```

# Istruzione di copia e trasferimento

MOV(B): Trasferisce un byte (MOVB) o una word (MOV) da una sorgente ad una destinazione senza alterare il contenuto della sorgente

Indirizzamento effettivo: un qualunque indirizzamento tra quelli visti

- Indirizzamento:

- registro  $\leftarrow$  indirizzo effettivo (Es. MOV AX,(200))
- indirizzo effettivo  $\leftarrow$  registro (Es. MOV (BX), AX)
- indirizzo effettivo  $\leftarrow$  dato immediato (Es. MOV AX,100)

- Vincoli:

- Non è possibile caricare un valore immediato in un registro segmento
- Il registro CS non è utilizzabile come destinazione di un'istruzione MOV.

# Operazioni sullo stack

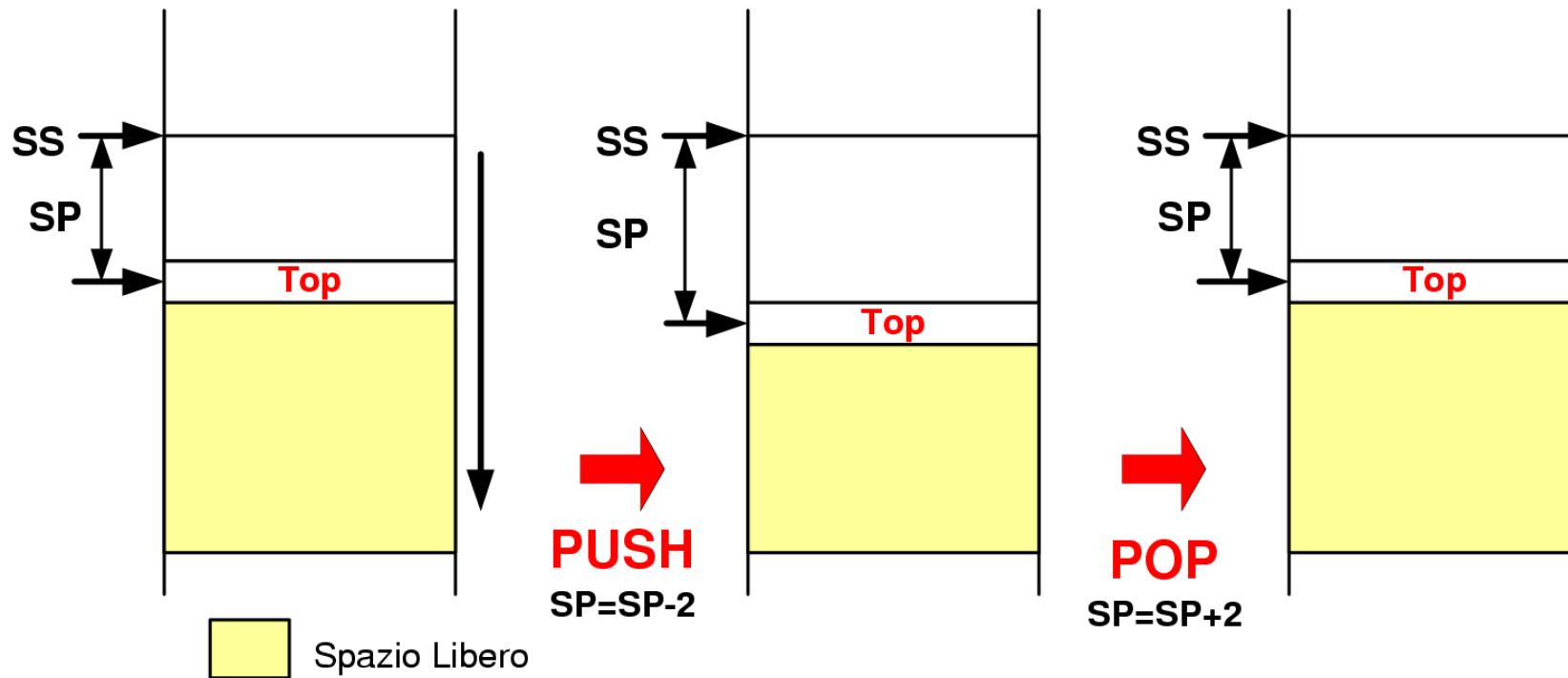
PUSH e POP aggiungono/rimuovono un elemento dalla cima dello stack selezionato da SS:SP

- Le operazioni sullo stack modificano il valore di SP:
  - Indirizzamento a stack implicito
- Operandi validi:
  - PUSH: operando immediato o indirizzo effettivo (es. PUSH 30 oppure PUSH AX)
  - POP: indirizzo effettivo (es POP AX)

Le operazioni PUSHF e POPF trasferiscono il contenuto del registro flag nella cima dello stack e viceversa.

# Operazioni di PUSH e POP

- L'operazione di PUSH decrementa SP di 2 byte
- L'operazione di POP incrementa SP di 2 byte



# Addizione

ADD(B): somma l'operando sorgente all'operando destinazione e memorizza il risultato nell'operando destinazione

- Indirizzamento:
  - registro  $\leftarrow$  indirizzo effettivo (Es. ADD AX,(200))
  - indirizzo effettivo  $\leftarrow$  registro (Es. ADD (BX), AX)
  - indirizzo effettivo  $\leftarrow$  dato immediato (Es. ADD AX,100)
- L'istruzione ADD modifica i bit del registro di flag

ADC comprende nella somma il flag del riporto.

# Sottrazione

SUB(B): sottrae l'operando sorgente all'operando destinazione e memorizza il risultato nell'operando destinazione.

- Indirizzamento:
  - registro  $\leftarrow$  indirizzo effettivo (Es. SUB AX,(200))
  - indirizzo effettivo  $\leftarrow$  registro (Es. SUB (BX), AX)
  - indirizzo effettivo  $\leftarrow$  dato immediato (Es. SUB AX,100)
- L'istruzione SUB modifica i bit del registro di flag

SBB comprende nella sottrazione il flag del riporto.

# Moltiplicazione

(I)MUL(B): moltiplica due operandi con/senza segno

- E' un'operazione unaria: MUL *source*
  - Il primo operando (implicito) è il registro accumulatore (AL per moltiplicazione tra byte, AX per word).
  - Il secondo operando è specificato da *source* e può essere un qualsiasi indirizzo effettivo.
  - Il risultato è posto in AX se si moltiplicano byte, in AX:DX se si moltiplicano word

IMUL effettua la moltiplicazione con segno

# Divisione

(I)DIV(B): divide due operandi con/senza segno.

- E' un'operazione unaria: DIV *source*
- Il divisore è specificato da *source* e può essere un qualsiasi indirizzo effettivo.
- Se *source* ha dimensioni di 1 byte:
  - Il dividendo (implicito) è AX
  - Il risultato della divisione è in AL, il resto in AH
- Se *source* ha dimensioni di 1 word:
  - Il dividendo (implicito) è DX:AX
  - Il risultato della divisione è in AX, il resto in DX

IDIV effettua la divisione con segno.

# Riepilogo istruzioni di movimento e aritmetiche

Mnemonic	Description	Operands	Status flags			
			O	S	Z	C
MOV(B)	Move word, byte	$r \leftarrow e, e \leftarrow r, e \leftarrow \#$	-	-	-	-
XCHG(B)	Exchange word	$r \leftrightarrow e$	-	-	-	-
LEA	Load effective address	$r \leftarrow \#e$	-	-	-	-
PUSH	Push onto stack	$e, \#$	-	-	-	-
POP	Pop from stack	$e$	-	-	-	-
PUSHF	Push flags	-	-	-	-	-
POPF	Pop flags	-	-	-	-	-
XLAT	Translate AL	-	-	-	-	-
ADD(B)	Add word	$r \leftarrow e, e \leftarrow r, e \leftarrow \#$	*	*	*	*
ADC(B)	Add word with carry	$r \leftarrow e, e \leftarrow r, e \leftarrow \#$	*	*	*	*
SUB(B)	Subtract word	$r \leftarrow e, e \leftarrow r, e \leftarrow \#$	*	*	*	*
SBB(B)	Subtract word with borrow	$r \leftarrow e, e \leftarrow r, e \leftarrow \#$	*	*	*	*
IMUL(B)	Multiply signed	$e$	*	U	U	*
MUL(B)	Multiply unsigned	$e$	*	U	U	*
IDIV(B)	Divide signed	$e$	U	U	U	U
DIV(B)	Divide unsigned	$e$	U	U	U	U

**e=indirizzo effettivo, r=registro, #=operando immediato**

# Operazioni logiche, su bit e di scorrimento

Principali Operazioni logiche: NEG(B), NOT(B), INC(B), DEC(B)

- L'operando è un indirizzo effettivo

Principali Operazioni su bit: AND(B), OR(B), XOR(B)

- registro  $\leftarrow$  indirizzo effettivo (Es. AND AX,(200))
- indirizzo effettivo  $\leftarrow$  registro (Es. AND (BX), AX)
- indirizzo effettivo  $\leftarrow$  dato immediato (Es. AND AX,1)

Principali Operazioni di scorrimento: SHR(B), SHL(B), ROL(B), ROR(B)

- L'operando è contenuto nel registro CL
- La destinazione è un indirizzo effettivo

# Operazioni logiche, su bit e scorrimento

Mnemonic	Description	Operands	Status flags			
			O	S	Z	C
CBW	Sign extend byte-word	-	-	-	-	-
CWD	Sign extend word-double	-	-	-	-	-
NEG(B)	Negate binary	e	*	*	*	*
NOT(B)	Logical complement	e	-	-	-	-
INC(B)	Increment destination	e	*	*	*	-
DEC(B)	Decrement destination	e	*	*	*	-
AND(B)	Logical and	$e \leftarrow r, r \leftarrow e, e \leftarrow \#$	0	*	*	0
OR(B)	Logical or	$e \leftarrow r, r \leftarrow e, e \leftarrow \#$	0	*	*	0
XOR(B)	Logical exclusive or	$e \leftarrow r, r \leftarrow e, e \leftarrow \#$	0	*	*	0
SHR(B)	Logical shift right	$e \leftarrow 1, e \leftarrow CL$	*	*	*	*
SAR(B)	Arithmetic shift right	$e \leftarrow 1, e \leftarrow CL$	*	*	*	*
SAL(B) (=SHL(B))	shift left	$e \leftarrow 1, e \leftarrow CL$	*	*	*	*
ROL(B)	Rotate left	$e \leftarrow 1, e \leftarrow CL$	*	-	-	*
ROR(B)	Rotate right	$e \leftarrow 1, e \leftarrow CL$	*	-	-	*
RCL(B)	Rotate left with carry	$e \leftarrow 1, e \leftarrow CL$	*	-	-	*
RCR(B)	Rotate right with carry	$e \leftarrow 1, e \leftarrow CL$	*	-	-	*

# Salti incondizionati

JMP: trasferisce il controllo all'istruzione specificata dall'operando in maniera incondizionata.

Due tipi di salto:

- Salto Corto: la destinazione si trova nel segmento di codice corrente (cui fa riferimento il registro CS)
- Salto Lungo: l'istruzione modifica il contenuto del registro CS

Esempio:

JMP label

ADD AX,BX

label:

AND AX,BX

N.B. La prossima istruzione ad essere eseguita è la AND

# Confronti

CMP(B): Effettua una sottrazione fra due operandi senza modificare nessuno dei due operandi

Esempio: CMP *operando1 operando2*

- Il risultato della sottrazione viene scartato.
- I bit del registro di flag vengono modificati.

# Salti condizionati

Jxx: istruzioni di salto in base ai valori del registro di flag

- Le istruzioni di salto condizionato controllano se una certa condizione è verificata.
- La condizione è specificata dal valore dei registri di flag.
- Azioni:
  - Se la condizione è verificata, il controllo passa all'istruzione il cui indirizzo è specificato come operando
  - Se la condizione non è verificata, l'esecuzione prosegue con l'istruzione successiva
- Vincoli:
  - Jxx consente salti di lunghezza massima pari a 128 byte

# Salti condizionati

<b>Instruction</b>	<b>Description</b>	<b>When to jump</b>
JNA, JBE	Below or equal	CF=1 or ZF=1
JNB, JAE, JNC	Not below	CF=0
JE, JZ	Zero, equal	ZF=1
JNLE, JG	Greater than	SF=OF and ZF=0
JGE, JNL	Greater equal	SF=OF
JO	Overflow	OF=1
JS	Sign negative	SF=1
JCXZ	CX is zero	CX=0
JB, JNAE, JC	Below	CF=1
JNBE, JA	Above	CF=0&ZF=0
JNE, JNZ	Nonzero, nonequal	ZF=0
JL, JNGE	Less than	SF≠OF
JLE, JNG	Less or equal	SF≠OF or ZF=1
JNO	Nonoverflow	OF=0
JNS	Nonnegative	SF=0

# Registro di stato

- Il registro di stato (flag) è un insieme di registri da 1 bit.
- I bit sono impostati da istruzioni aritmetiche:
  - **Z** - il risultato è zero
  - **S** - il risultato è negativo (bit di segno)
  - **O** - il risultato ha causato un overflow
  - **C** - il risultato ha generato un riporto
  - **A** - riporto ausiliario (oltre il bit 3)
  - **P** - parità del risultato
- Gli altri bit del registro controllano alcuni aspetti dell'attività del processore
  - **I** = attiva gli interrupt
  - **T** = abilita il tracing
  - **D** = operazioni su stringhe
- Non tutti i bit sono utilizzati

# Implementazione di istruzioni condizionali e cicli

If (a> b) then

    a=b;

else

    b=a;

while (a<1000)

    ...

    ...

end while;

CMP AX,BX

JLE else\_label

MOV AX,BX

JMP end\_label

else\_label:

MOV BX,AX

end\_label:

whileSum:

CMP AX,1000

JGE end\_while

...

JMP whileSum

end\_while:

# Cicli

L'istruzione LOOP consente di implementare esplicitamente cicli

- Il registro CX deve essere inizializzato con il numero di cicli dell'istruzione LOOP
- `LOOP statementLabel`
  - Il valore di CX viene decrementato
  - Se CX vale zero, l'esecuzione continua con l'istruzione successiva all'istruzione LOOP
  - Se CX è diverso da zero, allora viene eseguito un salto all'etichetta *statementLabel*

# Esempi di uso di loop e sue varianti

```
for (i=0; i<5; i++) {                                MOV AX,(a)  
    a=a+3;                                         MOV CX,5  
}                                                 repeat:  
                                                ADD AX,3  
                                                LOOP repeat
```

- LOOPE *statementLabel*
  - Decrementa CX e cicla (saltando all'etichetta *statementLabel*) se CX≠0 e il flag ZF=1
- LOOPNE *statementLabel*
  - Decrementa CX e cicla (saltando all'etichetta *statementLabel*) se CX≠0 e il flag ZF=0.

# Altre istruzioni

Mnemonic	Description	Operands	Status flags			
			O	S	Z	C
TEST(B)	Test operands	e ↔ r, e ↔ #	0	*	*	0
CMP(B)	Compare operands	e ↔ r, e ↔ #	*	*	*	*
STD	Set direction flag (↓)	-	-	-	-	-
CLD	Clear direction flag (↑)	-	-	-	-	-
STC	Set carry flag	-	-	-	-	1
CLC	Clear carry flag	-	-	-	-	0
CMC	Complement carry	-	-	-	-	*
LOOP	Jump back if decremented CX ≥ 0	label	-	-	-	-
LOOPZ LOOPE	Back if Z=1 and DEC(CX)≥0	label	-	-	-	-
LOOPNZ LOOPNE	Back if Z=0 and DEC(CX)≥0	label	-	-	-	-
REP REPZ REPNZ	Repeat string instruction	string instruction	-	-	-	-
MOVS(B)	Move word string	-	-	-	-	-
LODS(B)	Load word string	-	-	-	-	-
STOS(B)	Store word string	-	-	-	-	-
SCAS(B)	Scan word string	-	*	*	*	*
CMPS(B)	Compare word string	-	*	*	*	*
JCC	Jump according conditions	label	-	-	-	-
JMP	Jump to label	e, label	-	-	-	-
CALL	Jump to subroutine	e, label	-	-	-	-
RET	Return from subroutine	-, #	-	-	-	-
SYS	System call trap	-	-	-	-	-

## Esercizio II

Scrivere un programma in linguaggio assemblativo 8088 che, preso un intero n in memoria, calcola la somma dei primi n interi.

Il risultato deve essere stampato sullo standard output (video).

# Soluzione Esercizio

!Somma dei primi n numeri

\_EXIT = 1  
\_PRINTF = 127

.SECT .TEXT

start:

```
    MOV     AX,0
    MOV     CX,(number)
1:   ADD     AX,CX
    LOOP    1b
    MOV     (result), AX
    PUSH    (result)
    PUSH    (number)
    PUSH    format
    PUSH    _PRINTF
    SYS
    MOV     SP,BP
    PUSH    0
    PUSH    _EXIT
    SYS
```

.SECT .DATA

number: .WORD 5

result: .WORD 1 !

format: .ASCII "La somma dei primi %d interi e' %d"

## Esercizio IV

Scrivere un programma in linguaggio assemblativo 8088 che calcola la somma degli elementi di un vettore vec memorizzato in memoria principale.

Il risultato deve essere stampato sullo standard output (video).

# Soluzione Esercizio

! Stampa la somma di un vettore di interi

```
_EXIT      = 1  
_PRINTF   = 127
```

.SECT .TEXT

start:

```
    MOV CX,end-vec  
    SHR CX,1          ! In CX va la lunghezza del vettore  
    MOV BX,vec  
    MOV SI,0  
    MOV AX,0  
1:   ADD AX,(BX)(SI)  
    ADD SI,2  
    LOOP 1b  
    PUSH AX  
    PUSH format  
    PUSH _PRINTF  
    SYS  
    MOV SP,BP  
    PUSH 0  
    PUSH _EXIT  
    SYS
```

.SECT .DATA

vec: .WORD 3,4,7,11,3

end: .SPACE 1

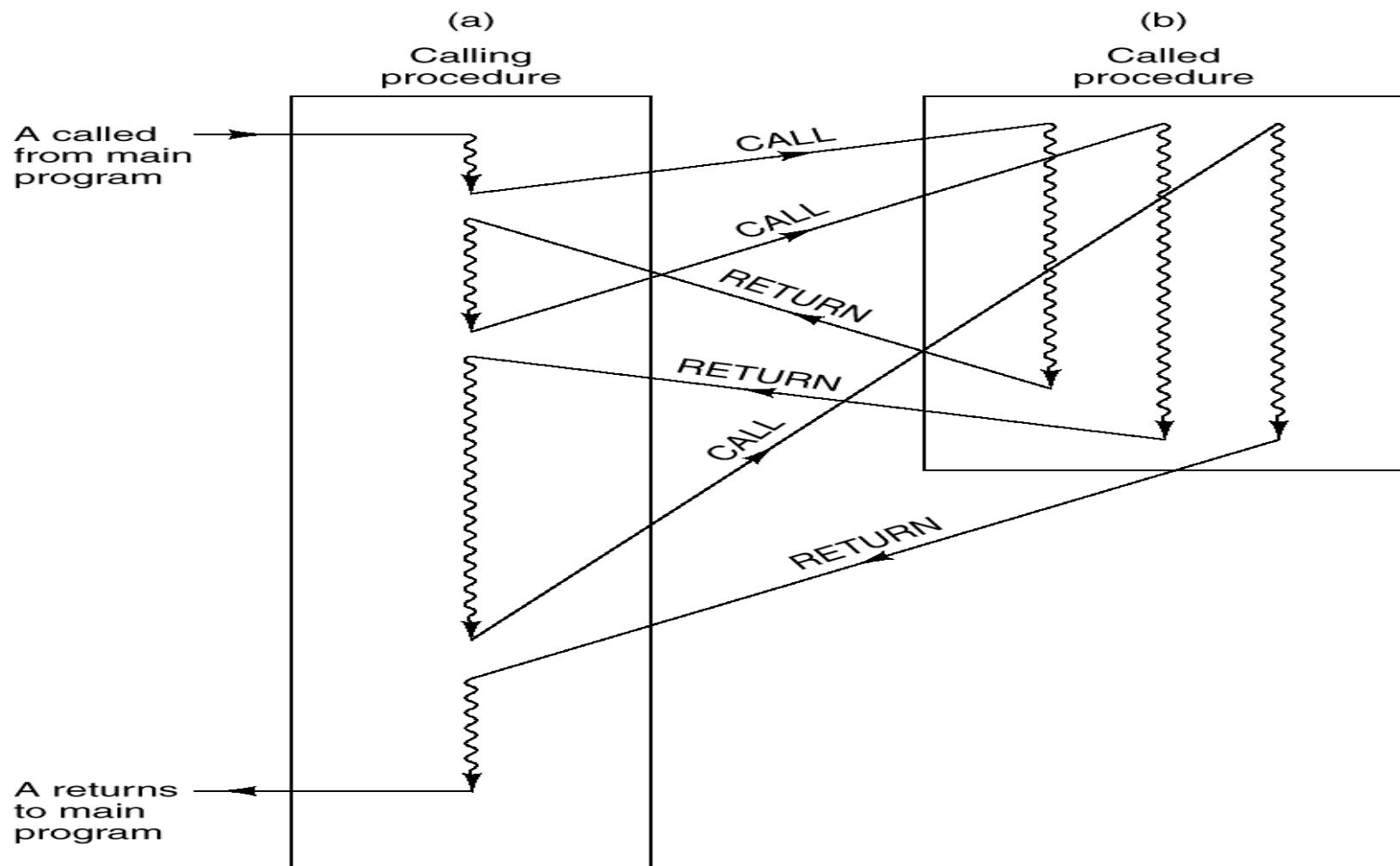
format: .ASCII "La somma degli elementi del vettore e' %d"

# Chiamate di procedura (subroutine)

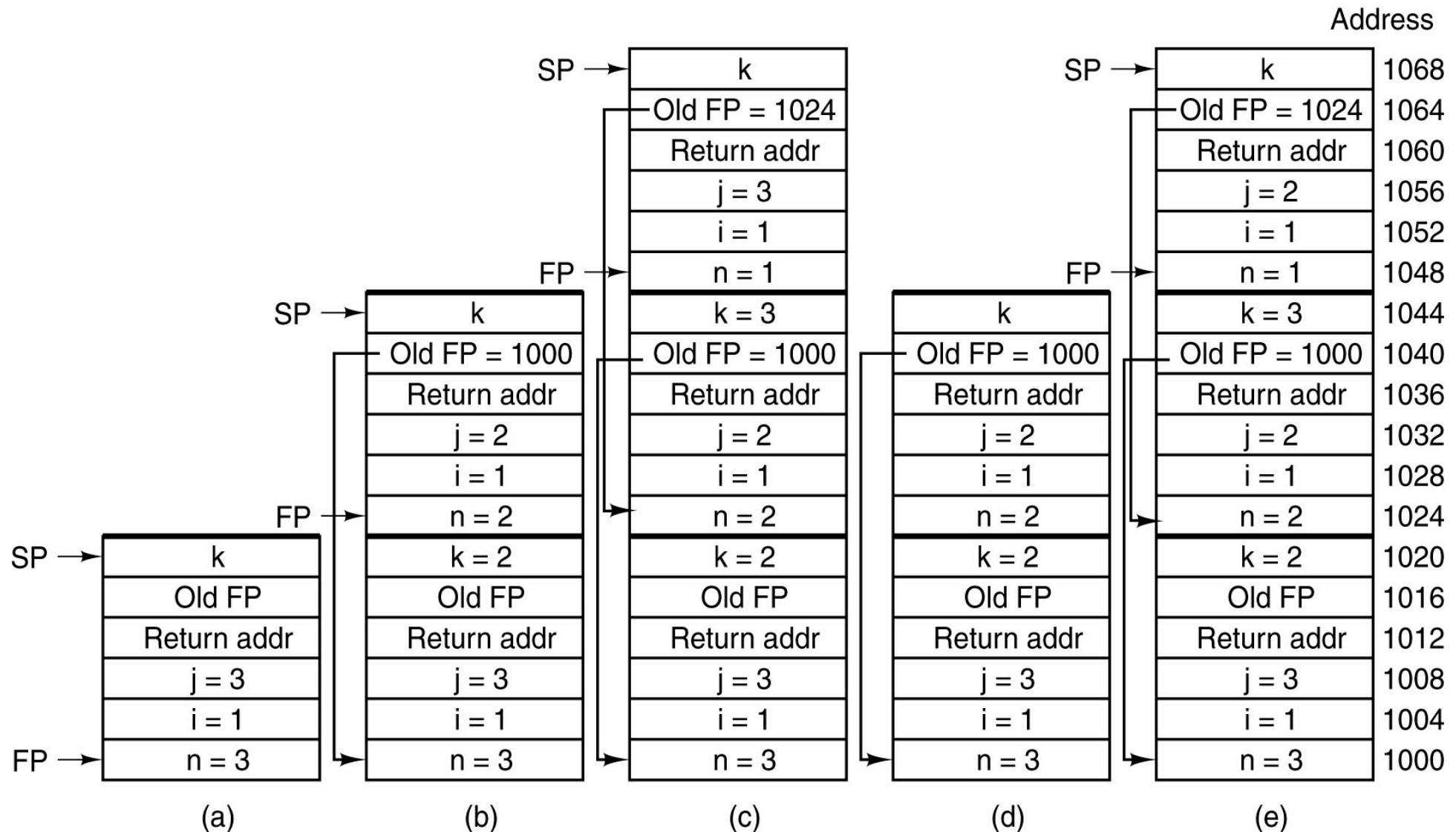
Un programma in linguaggio assemblativo può essere suddiviso in sottoprogrammi detti subroutine.

- Vantaggi:
  - Consentono di suddividere il codice in blocchi funzionali
  - Consentono di riutilizzare codice
- Caratteristiche:
  - Possono ricevere parametri in ingresso
  - Possono disporre di variabili locali
  - Possono restituire un valore di ritorno.

# Chiamata di procedura e indirizzo di ritorno



# Struttura tipica Stack nell'invocazione di subroutine



# Invocazione di subroutine in x86

L'istruzione CALL trasferisce il controllo dal programma chiamante alla procedura chiamata

- Sintassi: *CALL Nome Funzione*
  - Salva l'indirizzo di ritorno sulla cima dello stack
  - Passa il controllo alla procedura chiamata
- Esistono invocazioni:
  - *ravvicinate* (all'interno dello stesso segmento di codice)
  - *a distanza* (tra segmenti diversi, occorre salvare CS ed indirizzo di ritorno)

# Ritorno da subroutine

L'istruzione RET trasferisce il controllo dalla procedura chiamata alla procedura chiamante

- Sintassi: RET [No argomenti]
  - Legge dalla cima dello stack l'indirizzo di ritorno salvato dalla precedente CALL.
  - Restituisce il controllo alla procedura chiamante
- N.B. Al momento dell'esecuzione della RET, la cima dello stack deve contenere l'indirizzo di ritorno

## Invocazione semplice

Esempio di subroutine con CALL e RET che non usa variabili locali e parametri

MOV AX,BX

CALL esempio ;Chiamo la subroutine

....

esempio:

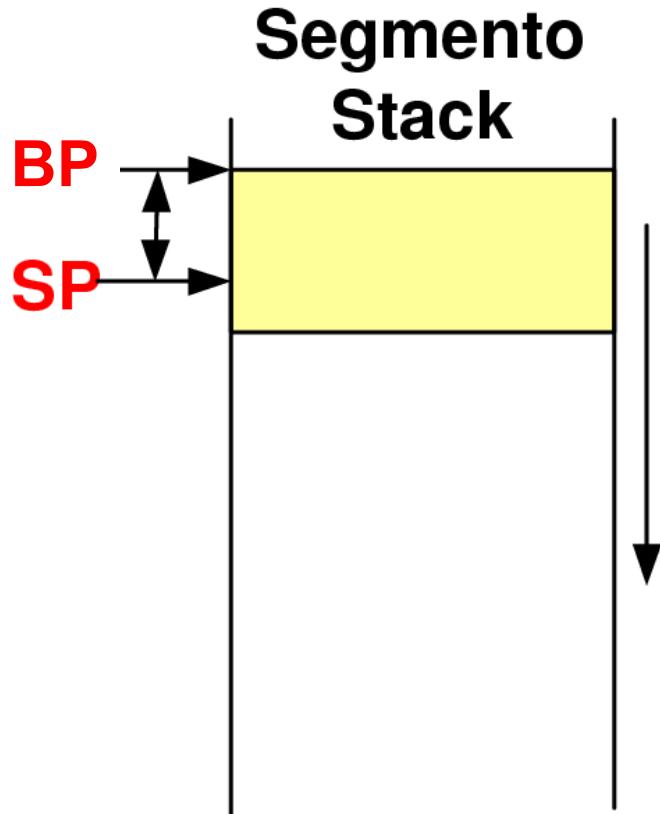
MOV BX,2

....

RET ;Restituisco il controllo al chiamante

# Invocazione con argomenti (1)

## PASSO 1: Invocazione della funzione



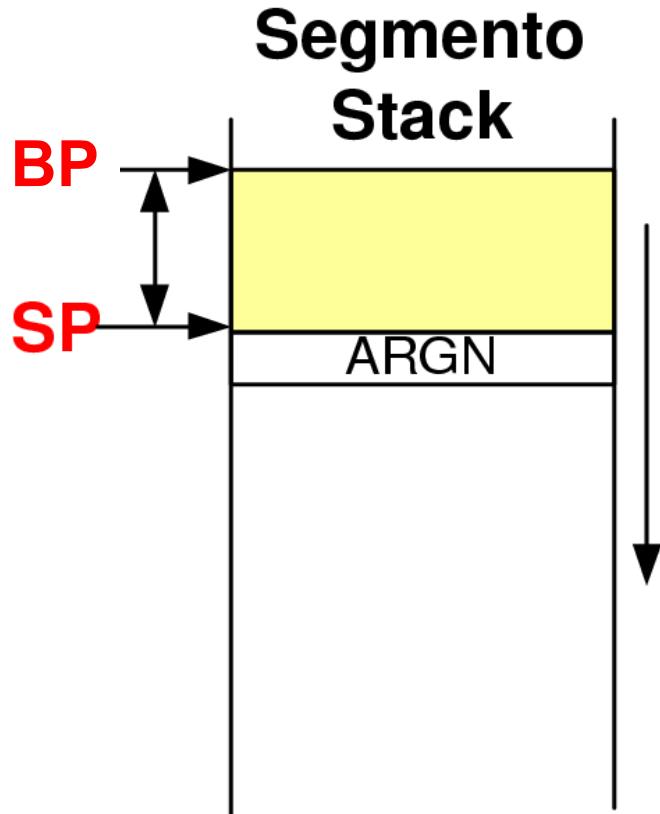
Linguaggio ad Alto Livello:  
fun (arg1, arg2, .... argN)

In Linguaggio Assembly:

```
PUSH ARGN  
...  
PUSH ARG2  
PUSH ARG1  
CALL fun
```

# Invocazione con argomenti (2)

## PASSO 1: Invocazione della funzione

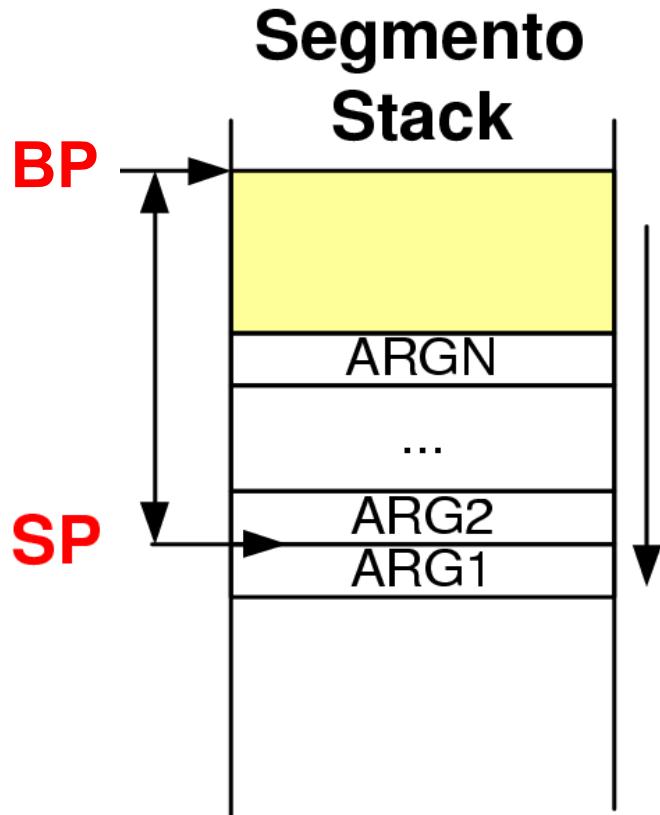


Linguaggio ad Alto Livello:  
fun (arg1, arg2, .... argN)

In Linguaggio Assembly:  
**PUSH ARGN**  
...  
**PUSH ARG2**  
**PUSH ARG1**  
**CALL fun**

# Invocazione con argomenti (3)

## PASSO 1: Invocazione della funzione

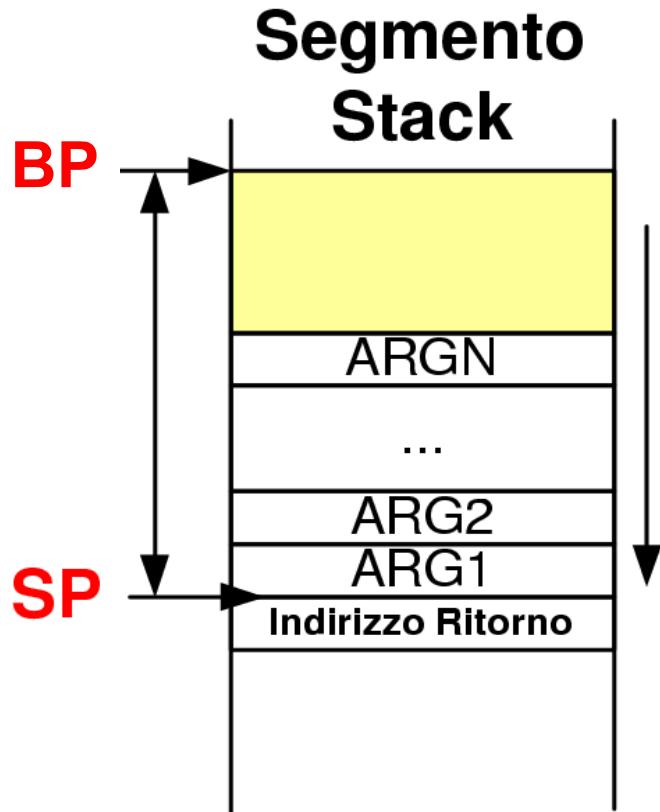


Linguaggio ad Alto Livello:  
fun (arg1, arg2, .... argN)

In Linguaggio Assembly:  
**PUSH ARGN**  
...  
**PUSH ARG2**  
**PUSH ARG1**  
**CALL fun**

# Invocazione con argomenti (4)

## PASSO 1: Invocazione della funzione



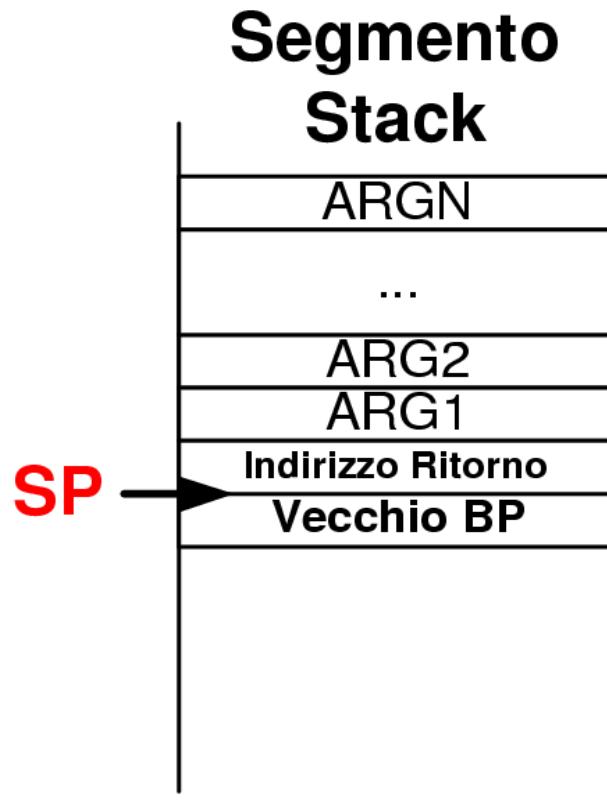
Linguaggio ad Alto Livello:  
fun (arg1, arg2, .... argN)

In Linguaggio Assembly:

```
PUSH ARGN  
...  
PUSH ARG2  
PUSH ARG1  
CALL fun
```

# Invocazione con argomenti (5)

## PASSO 2: Prologo della funzione chiamata

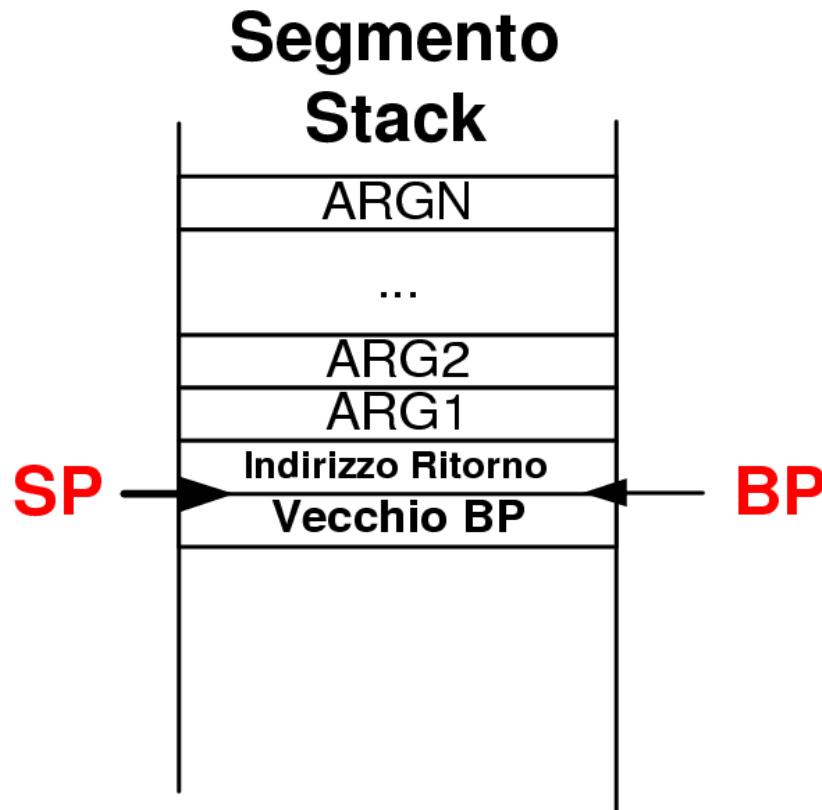


In Linguaggio Assembly:

**PUSH BP  
MOV BP,SP  
SUB SP, xx**

# Invocazione con argomenti (6)

## PASSO 2: Prologo della funzione chiamata



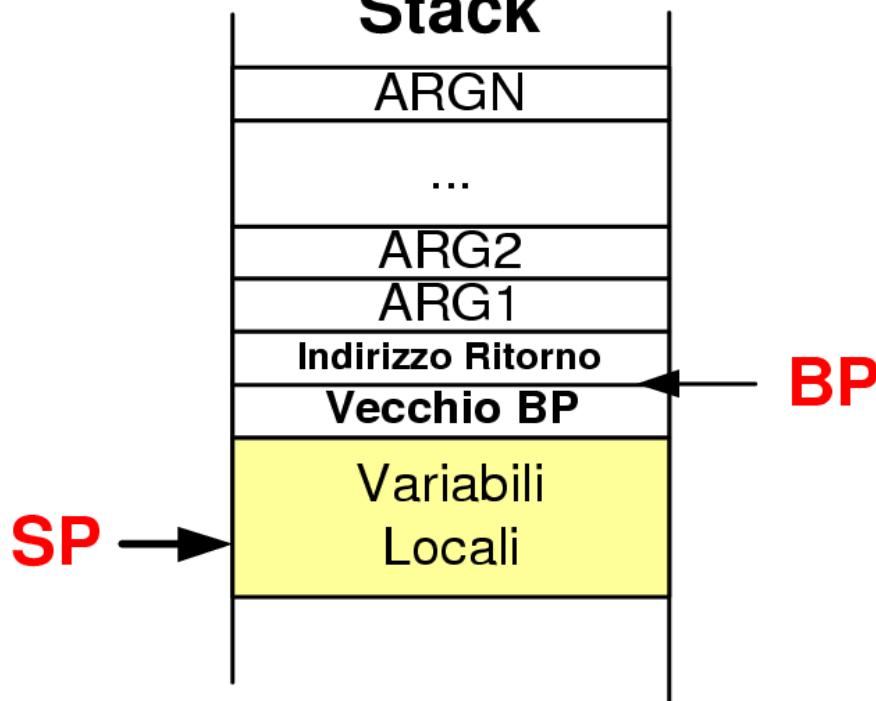
In Linguaggio Assembly:

**PUSH BP**  
**MOV BP,SP**  
**SUB SP, xx**

# Invocazione con argomenti (7)

## PASSO 2: Prologo della funzione chiamata

### Segmento Stack



In Linguaggio Assembly:

**PUSH BP**  
**MOV BP,SP**  
**SUB SP, xx**

# Stack durante l'esecuzione di una subroutine

BP+8	...
BP+6	Argument 2
BP+4	Argument 1
BP+2	Return address
BP	Old BP
BP-2	Local variable 1
BP-4	Local variable 2
BP-6	Local variable 3
BP-8	Temporary result

← BP      ← SP

# Invocazione di subroutine con argomenti

- La funzione chiamante deve:
  - Impilare sullo stack gli argomenti della funzione in ordine inverso (dall'ultimo al primo)
  - Trasferire il controllo con l'istruzione CALL
- La funzione chiamata deve:
  - Salvare sullo stack il valore corrente del registro BP
  - Sovrascrivere BP con il contenuto corrente di SP
  - Allocare le variabili locali sullo stack
- Al termine della funzione occorre:
  - Sovrascrivere SP con il contenuto di BP (svuota il record di attivazione)
  - Effettuare una POP dallo stack su BP
  - Eseguire l'istruzione RET
  - Rimuovere gli argomenti dallo stack

# Chiamate di procedura: esempio

Esempio di subroutine che calcola a + b + c

```
PUSH (c)  
PUSH (b)  
PUSH (a)  
CALL subroutine  
ADD SP,6  
  
....
```

subroutine:

```
PUSH BP  
MOV BP, SP  
MOV AX,4(BP)  
MOV BX,6(BP)  
MOV DX,8(BP)  
ADD AX,BX  
ADD AX,DX  
MOVE SP, BP !Inutile in questo caso. Serve se ho allocato variabili locali.  
POP BP  
RET
```

situazione stack →

SP →

BP ←

c	BP+8	
b	BP+6	
a	BP+4	
Indirizzo di ritorno		
vecchio BP		

# Chiamate di sistema

- Le chiamate di sistema consentono di utilizzare le procedure fornite dal sistema operativo
- Le routine di sistema possono essere attivate con la sequenza di chiamata standard:
  - Si copiano gli argomenti sullo stack
  - Si impila il numero di chiamata
  - Si esegue l'istruzione SYS
- I risultati sono restituiti nel registro AX o nella combinazione di registri AX:DX (se il risultato è di tipo long)
- Gli argomenti sullo stack devono essere rimossi dalla funzione chiamante

# System Calls and System Subroutines

Nr	Name	Arguments	Return value	Description
5	_OPEN	*name, 0/1/2	file descriptor	Open file
8	_CREAT	*name, *mode	file descriptor	Create file
3	_READ	fd, buf, nbytes	# bytes	Read nbytes in buffer buf
4	_WRITE	fd, buf, nbytes	# bytes	Write nbytes from buffer buf
6	_CLOSE	fd	0 on success	close file with fd
19	_LSEEK	fd, offset(long), 0/1/2	position (long)	Move file pointer
1	_EXIT	status		Close files Stop process
117	_GETCHAR		read character	Read character from std input
122	_PUTCHAR	char	write byte	Write character to std output
127	_PRINTF	*format, arg		Print formatted on std output
121	_SPRINTF	buf, *format, arg		Print formatted in buffer buf
125	_SSCANF	buf, *format, arg		Read arguments from buffer buf

## Esercizio

Scrivere un programma in linguaggio assemblativo 8088 che, preso un numero *a* in memoria, calcola il quadrato del numero facendo uso di una subroutine “square” che ha come unico argomento il numero *a*.

Il risultato deve essere stampato sullo standard output (video).

# Soluzione Esercizio III

! Calcola la il quadrato di un numero con la subroutine "square"

```
_EXIT      = 1
_PRINTF    = 127
.SECT .TEXT
start:
    PUSH (a)
    CALL square
    MOV SP,BP
    PUSH AX
    PUSH pfmt
    PUSH _PRINTF
    SYS
    MOV SP,BP
    PUSH 0
    PUSH _EXIT
    SYS

square:
    PUSH BP
    MOV BP,SP
    MOV AX,4(BP)
    MOV BX,AX
    MUL BX
    POP BP
    RET

.SECT .DATA
pfmt: .ASCIZ "Il quadrato e' %d!\n"
a:      .WORD 3
```

## Esercizio V

Scrivere un programma in linguaggio assemblativo 8088 che calcola la somma degli elementi di un vettore `vec` memorizzato in memoria principale, facendo uso di una subroutine "vecsum" che ha come argomento la dimensione del vettore e il vettore.

Il risultato deve essere stampato sullo standard output (video).

# Soluzione Esercizio V

! Stampa la somma di un array di interi mediante una subroutine "vecsum"

```
_EXIT          = 1
_PRINTF        = 127

.SECT .TEXT
vecpstrt:
    PUSH vec
    MOV CX,end-vec
    SHR CX,1
    PUSH CX
    CALL vecsum
    MOV SP,BP
    PUSH AX
    PUSH format
    PUSH _PRINTF
    SYS
    MOV SP,BP
    PUSH 0
    PUSH _EXIT
    SYS

vecsum:
    PUSH BP
    MOV BP,SP
    MOV CX,4(BP)
    MOV BX,6(BP)
    MOV SI,0
    MOV AX,0
    1:   ADD AX,(BX)(SI)
        ADD SI,2
        LOOP 1b
    MOV SP,BP
    POP BP
    RET

.SECT .DATA
vec:  .WORD 3,4,7,11,3
end:  .SPACE 1
format: .ASCII "La somma della stringa e' %d"
.SECT .BSS
```

# Operazioni su array e stringhe: MOVSB

L'istruzione MOVSB sposta un byte dalla posizione indirizzata dal registro SI alla posizione indirizzata dal registro DI.

- Sintassi: MOVSB [No Operandi]
  - L'indirizzo del byte sorgente deve trovarsi in SI
  - L'indirizzo del byte destinazione deve trovarsi in DI
- Al termine dell'operazione, i registri SI/DI vengono incrementati o decrementati a seconda del valore corrente del bit di direzione nel registro di flag:
  - CLD: con questa istruzione, i registri SI e DI vengono incrementati (scorrimento in avanti)
  - STD: con questa istruzione, i registri SI e DI vengono decrementati (scorrimento all'indietro)
- MOVSB per operazioni su byte (8bit), MOVSW per operazioni su parole (16 bit).

# Operazioni su array e stringhe: REP

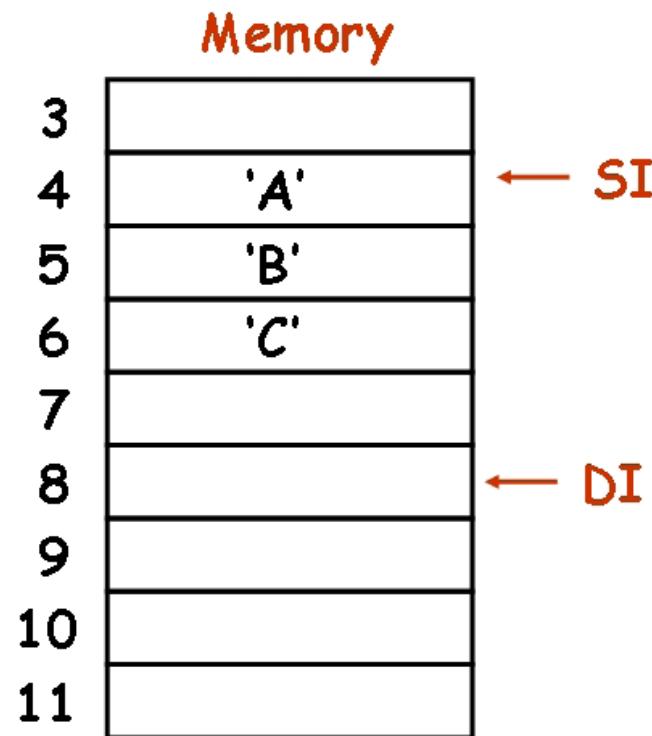
L'istruzione REP MOVSB itera l'esecuzione dell'istruzione MOVSB un numero di volte pari al valore corrente del registro CX.

- Sintassi: REP MOVSB

- I registri SI e DI sono inizializzati con l'indirizzo della stringa sorgente e della stringa destinazione.
- CX deve essere inizializzato con la lunghezza della stringa.
- L'istruzione REP MOVSB ripete l'esecuzione di MOVSB finché CX=0.
- In ogni iterazione, il valore dei registri SI e DI viene incrementato/decrementato.

# Esempio (1)

```
CLD  
MOV SI,4  
MOV DI,8  
MOV CX,3  
REP MOVSB
```



SI

4

DI

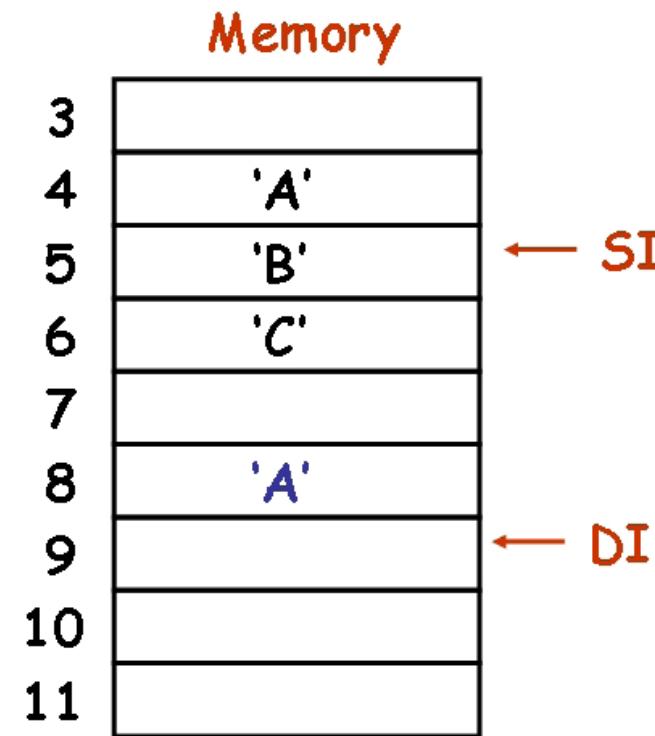
8

CX

3

## Esempio (2)

```
CLD  
MOV SI,4  
MOV DI,8  
MOV CX,3  
REP MOVSB
```



SI

5

DI

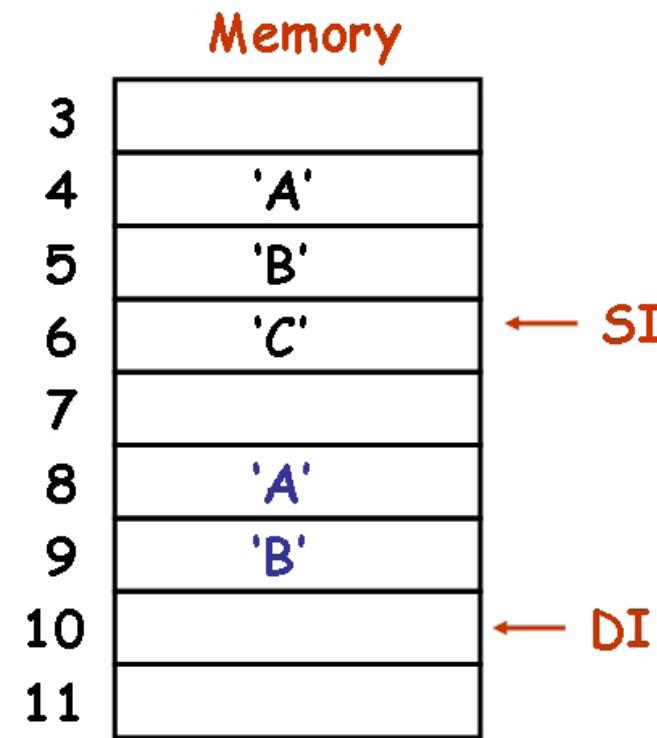
9

CX

2

## Esempio (3)

```
CLD  
MOV SI,4  
MOV DI,8  
MOV CX,3  
REP MOVS
```



SI

6

DI

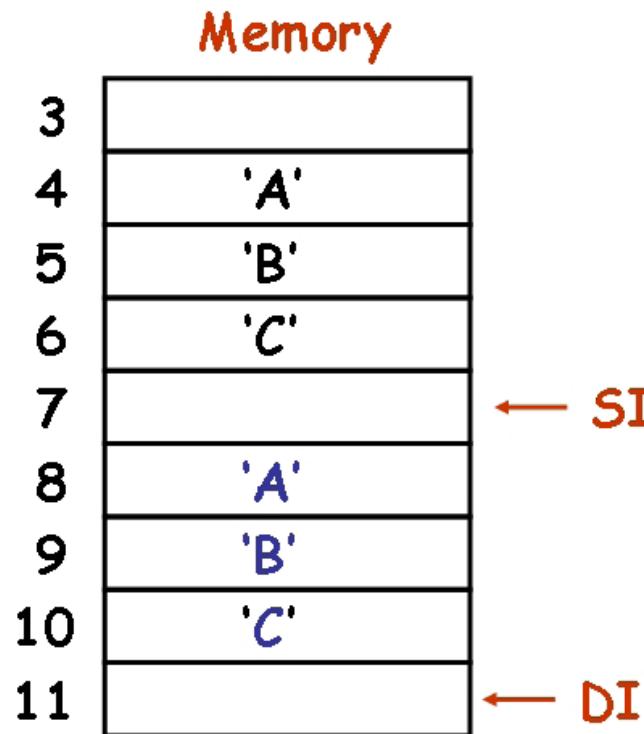
10

CX

1

## Esempio (4)

```
CLD  
MOV SI,4  
MOV DI,8  
MOV CX,3  
REP MOVSB
```



SI

7

DI

11

CX

0

# Operazioni su array e stringhe: SCASB

L'istruzione SCASB confronta il byte indirizzato dal registro DI con il contenuto del registro AL.

- Sintassi: SCASB [No Operandi]
  - L'indirizzo del primo byte deve trovarsi in DI.
  - Il secondo byte deve trovarsi in AL.
  - Al termine dell'operazione, il registro DI viene incrementato o decrementato a seconda del valore corrente del bit di direzione nel registro di flag.
  - Il registro AL non viene alterato, ma i bit del registro di flag vengono modificati
- SCASB per operazioni su byte (8bit), SCASW per operazioni su parole (16 bit, confronto con registro AX).

# Operazioni su array e stringhe: CMPSB

L'istruzione CMPSB confronta due byte, indirizzati rispettivamente dal registro SI e dal registro DI.

- Sintassi: CMPSB [No Operandi]
  - L'indirizzo del primo byte deve trovarsi in SI.
  - L'indirizzo del secondo byte deve trovarsi in DI.
  - Al termine dell'operazione, i registri SI/DI vengono incrementati o decrementati a seconda del valore corrente del bit di direzione nel registro di flag.
- CMPSB per confronto tra byte (8bit), CMPSW per confronto tra parole (16 bit).

# Operazioni su array e stringhe: REP, REPE, REPNE

- REP {MOVSB|MOVSW}
  - Ripete l'istruzione fino a fine stringa (CX=0).
- REPE (o REPZ) {SCASB|SCASW|CMPSB|CMPSW}
  - Ripete l'istruzione fino a fine stringa (CX=0), oppure fino a quando il confronto fallisce (ZF=0).
- REPNE (o REPNZ) {SCASB|SCASW|CMPSB|CMPSW}
  - Ripete l'istruzione fino a fine stringa (CX=0), oppure fino a quando il confronto ha successo (ZF=1).

## Esercizio VI

Con riferimento al programma assemblativo 8088 che segue,  
indicare cosa fa e il valore stampato.

# Esercizio VI

```
_EXIT = 1
_PRINTF      = 127

.SECT .TEXT
start:
    MOV CX,num-vec
    SHR CX,1
    MOV BX,vec
    MOV SI,0
    MOV AX,(num)
1:     CMP AX,(BX)(SI)
    JE 2f
    ADD SI,2
    LOOP 1b
    MOV DX,0
    JMP 3f
2:     MOV DX,1
3:     PUSH DX
    PUSH format
    PUSH _PRINTF
    SYS
    MOV SP,BP
    PUSH 0
    PUSH _EXIT
    SYS

.SECT .DATA
vec: .WORD 3,4,7,11,3
num: .WORD 5
format: .ASCII "%d"
```

# Esercizio VIbis

! Equivalente al VI utilizzando pero' l'istruzione SCASW insieme alla REPNE

```
_EXIT      = 1
_PRINTF    = 127
```

```
.SECT .TEXT
```

```
start:
```

```
    MOV CX,num-vec
```

```
    SHR CX,1
```

```
    MOV AX,(num)
```

```
    MOV DI, vec
```

```
    CLD
```

**REPNE SCASW**

```
    JE 1f
```

```
    MOV DX,0
```

```
    JMP 2f
```

```
1:   MOV DX,1
```

```
2:   PUSH DX
```

```
    PUSH format
```

```
    PUSH _PRINTF
```

```
    SYS
```

```
    MOV SP,BP
```

```
    PUSH 0
```

```
    PUSH _EXIT
```

```
    SYS
```

```
.SECT .DATA
```

```
vec:  .WORD 3,4,7,11,3
```

```
num:  .WORD 11
```

```
format: .ASCII "%d"
```

## Esercizio VII

Scrivere un programma in linguaggio assemblativo 8088 che verifica se due vettori di interi memorizzati in memoria principale sono identici.

# Esercizio VII

```
_EXIT      = 1
_PRINTF    = 127
```

```
.SECT .TEXT
```

```
inizio:
```

```
    MOV CX,end1-vec1
    SHR CX,1
    MOV AX,end2-vec2
    SHR AX,1
    CMP AX,CX
    JNE 1f
    MOV SI,vec1
    MOV DI,vec2
    CLD
    REPE CMPSW
    JNE 1f
```

```
    PUSH uguali
    JMP 2f
```

```
1:    PUSH diversi
```

```
2:    PUSH _PRINTF
```

```
    SYS
    MOV SP,BP
    PUSH 0
    PUSH _EXIT
    SYS
```

```
.SECT .DATA
```

```
vec1: .WORD 3,4,7,11,3
```

```
end1: .SPACE 1
```

```
vec2: .WORD 3,4,7,11,3
```

```
end2: .SPACE 1
```

```
uguali: .ASCII "Uguali!\0"
```

```
diversi: .ASCII "Diversi!\0"
```

# Operazioni su array e stringhe: LODSB

L'istruzione LODSB trasferisce il contenuto del byte di memoria indirizzato dal registro DI nel registro AL.

- Sintassi: LODSB [No Operandi]
  - L'indirizzo di memoria del dato da trasferire deve trovarsi in DI.
  - La destinazione è il registro AL.
  - Al termine dell'operazione, il registro DI viene incrementato o decrementato a seconda del valore corrente del bit di direzione nel registro di flag.
- LODSB per operazioni su byte (8bit), LODSW per operazioni su parole (16 bit, destinazione registro AX).

# Operazioni su array e stringhe: STOSB

L'istruzione STOSB trasferisce il contenuto del registro AL nel byte di memoria indirizzato dal registro DI.

- Sintassi: STOSB [No Operandi]
  - Il dato da trasferire deve trovarsi in AL.
  - L'indirizzo di memoria della destinazione deve trovarsi in DI.
  - Al termine dell'operazione, il registro DI viene incrementato o decrementato a seconda del valore corrente del bit di direzione nel registro di flag.
- STOSB per operazioni su byte (8bit), STOSW per operazioni su parole (16 bit).

# Operazioni su array e stringhe: REP

- REP {LODSB|LODSW|STOSB|STOSW}
  - Ripete l'istruzione fino a fine stringa (CX=0).

## Esercizio VIII

Scrivere un programma in linguaggio assemblativo 8088 che, dato un numero memorizzato in memoria principale, calcola il fattoriale del numero ( $n! = n \times (n-1) \times \dots \times 1$ ) e lo stampa.

# Esercizio VIII

!Calcolo del fattoriale

```
_EXIT      = 1
_PRINTF = 127

.SECT .TEXT
start:
    MOV      AX,(number)
    CMP      AX,1
    JG       1f
    MOV      AX,1
    JMP      3f
1:   MOV      CX,AX
    DEC      CX
2:   IMUL     CX
    LOOP     2b
3:   MOV      (result), AX
    PUSH     (result)
    PUSH     (number)
    PUSH     fmt
    PUSH     _PRINTF
    SYS
    MOV      SP,BP
    PUSH     0
    PUSH     _EXIT
    SYS

.SECT .DATA
number: .WORD      5
result: .WORD     1
fmt:      .ASCII "il fattoriale di %d e' %d\0"
```

## Esercizio X

Scrivere un versione ricorsiva del programma del calcolo del fattoriale.

# Esercizio X: Possibile soluzione

!Calcolo del fattoriale: versione ricorsiva

\_EXIT = 1  
\_PRINTF = 127

.SECT .TEXT

start:

```
PUSH (number)  
  
CALL fatt  
POP CX  
MOV SP,BP  
PUSH CX  
PUSH (number)  
PUSH fmt  
PUSH _PRINTF  
SYS  
  
MOV SP,BP  
PUSH 0  
PUSH _EXIT  
SYS
```

fatt:

```
PUSH BP  
MOV BP,SP  
MOV CX,4(BP)  
CMP CX,1  
JG 1f  
MOV 4(BP),1  
JMP 2f
```

1:

```
DEC CX  
PUSH CX  
CALL fatt  
POP CX  
MOV AX,4(BP)  
IMUL CX  
MOV 4(BP),AX
```

2:

```
MOV BP,SP  
POP BP  
RET
```

.SECT .DATA

number: .WORD 3

fmt: .ASCII "il fattoriale di %d e' %d\n"

## Esercizio IX

Scrivere un programma in linguaggio assemblativo 8088 che dato un numero  $n$  memorizzato in memoria principale, verifica se è un numero primo.

Consiglio: utilizzare l'istruzione DIV che divide l'argomento per il contenuto di AX mettendo il risultato in AX e il resto in DX

# Esercizio IX

```
_EXIT      = 1
_PRINTF    = 127

.SECT .TEXT
start:
        MOV      BX,(n)
        MOV      CX,BX
1:      DEC      CX
        CMP      CX,1
        JLE      3f
        MOV      AX,BX
        MOV      DX,0
        DIV      CX
        CMP      DX,0
        JE       2f
        JMP      1b
2:      MOV      BX, nonprimo
        JMP      4f
3:      MOV      BX, primo
4:      PUSH     (n)
        PUSH     BX
        PUSH     _PRINTF
        SYS
        MOV      SP,BP
        PUSH     0
        PUSH     _EXIT
        SYS

.SECT .DATA
n:      .WORD     49
primo: .ASCII "%d e' un numero primo\0"
nonprimo: .ASCII "%d non e' un numero primo\0"
```

# Esercizio IXbis: altra possibile soluzione

```
_EXIT      = 1
_PRINTF = 127

.SECT .TEXT
start:
        MOV      BX,(n)
        MOV      CX,BX
        DEC      CX
1:       CMP      CX,1
        JLE      3f
        MOV      AX,BX
        MOV      DX,0
        IDIV     CX
        CMP      DX,0
        LOOPNZ  1b
2:       MOV      BX, nonprimo
        JMP      4f
3:       MOV      BX, primo
4:       PUSH     (n)
        PUSH     BX
        PUSH     _PRINTF
        SYS
        MOV      SP,BP
        PUSH     0
        PUSH     _EXIT
        SYS

.SECT .DATA
n:        .WORD    49
primo:   .ASCII  "%d e' un numero primo\0"
nonprimo: .ASCII  "%d non e' un numero primo\0"
```

## Esercizio XI

Scrivere un programma in linguaggio assemblativo 8088 che calcola il del prodotto scalare di due vettori (somma dei prodotti degli elementi omologhi). Il programma deve essere dotato di una subroutine **prodvec** avente quattro parametri:

- **vec1** (indirizzo del primo vettore)
- **vec2** (indirizzo del secondo vettore)
- **dimensione** (dimensione dei vettori – si assume che siano della stessa lughezza)
- **risultato** (parametro di output che al termine dell'esecuzione della subroutine memorizza il risultato del prodotto).

# Esercizio XI: possibile soluzione

```
_EXIT      = 1
_PRINTF    = 127
.SECT .TEXT
start:
    MOV CX,vec2-vec1
    SHR CX,1
    PUSH 0      !quarto parametro inz. a zero
    PUSH CX     !terzo parametro
    PUSH vec2   !secondo parametro
    PUSH vec1   !primo parametro
    CALL prodvec
    ADD SP,6    !tolgo i primi tre parametri
    POP AX
    MOV SP,BP
    PUSH AX
    PUSH fmt
    PUSH _PRINTF
    SYS
    MOV SP,BP
    PUSH 0
    PUSH _EXIT
    SYS
```

prodvec:

```
PUSH BP
MOV BP,SP
MOV CX,8(BP)
MOV SI,0
PUSH 0      !variabile locale inz. a zero
1:          MOV BX,4(BP)
            MOV AX,(BX)(SI)
            MOV BX,6(BP)
            MUL (BX)(SI)
            ADD -2(BP),AX
            ADD SI,2
            LOOP 1b
            POP 10(BP) !salvo la var. nel 4
argomento
POP BP
RET
```

```
.SECT .DATA
vec1: .WORD 3,4,7,11,3
vec2: .WORD 2,6,3,1,0
fmt: .ASCII "Il prodotto dei due vettori e': %d!\0"!
```

## Esercizio XII

Scrivere una subroutine PAL in assembler 8088 che, dato una stringa (vettore di caratteri) S memorizzata in memoria principale, stampa restituisce 1 se la stringa S è palindroma (è uguale leggendola nei due versi; per esempio la stringa "anna" è palindroma) e 0 altrimenti.

La subroutine PAL ha come parametri:

- L'indirizzo della stringa da verificare e
- la lunghezza della stringa

# Soluzione Esercizio XII

```
! Verifica di stringhe palindrome

_EXIT      = 1
_PRINTF    = 127
.SECT .TEXT
start:
    PUSH ends-str !secondo parametro
    PUSH str !primo parametro
    CALL pal
    MOV SP,BP
    PUSH AX
    PUSH fmt
    PUSH _PRINTF
    SYS
    MOV SP,BP
    PUSH 0
    PUSH _EXIT
    SYS

pal:
    PUSH BP
    MOV BP,SP
    MOV BX,4(BP)
    MOV CX,6(BP)

    MOV SI,CX
    DEC SI
    MOV DI,str2
    MOVB AL,(BX)(SI)
    STOSB
    DEC SI
    LOOP 1b
    MOV SI,4(BP)
    MOV DI,str2
    MOV CX,6(BP)
    REPE CMPSB
    JE 2f
    MOV AX,0
    JMP 3f
2:   MOV AX,1
3:   POP BP
    RET

.SECT .DATA
str: .ASCII "ingegni"
ends: .SPACE 1
fmt: .ASCII "%d"
endf: .SPACE 1
str2: .ASCII "."

```

## Esercizio XIII

Scrivere un programma in linguaggio assemblativo 8088 che trova il più grande degli elementi di un vettore vec memorizzato in memoria principale. Si assuma che il vettore abbia almeno un elemento.

Il risultato deve essere stampato sullo standard output (video).

# Soluzione Esercizio XIII

! Trova il piu' grande tra gli elementi di un vettore di interi

```
_EXIT      = 1
_PRINTF   = 127
```

```
.SECT .TEXT
```

```
start:
```

```
    MOV CX,end-vec
```

```
    SHR CX,1 !in CX va la dimensione di vec
```

```
    MOV BX,vec !il registro base punta al primo elemento di vec
```

```
    MOV AX,(vec) !inizializzo AX con il primo elemento di vec
```

```
1:     CMP AX,(BX)(SI)
```

```
        JGE 2f
```

```
        MOV AX,(BX)(SI)
```

```
2:     ADD SI,2
```

```
     LOOP 1b
```

```
     PUSH AX
```

```
     PUSH format
```

```
     PUSH _PRINTF
```

```
     SYS
```

```
     MOV SP,BP
```

```
     PUSH 0
```

```
     PUSH _EXIT
```

```
     SYS
```

```
.SECT .DATA
```

```
vec:  .WORD 3,-4,7,11,34,-4,22,0,5
```

```
end:  .SPACE 1
```

```
format: .ASCII "Il piu' grande tra elementi del vettore e' %d"
```