

Programmazione Assembler

parte 2

Giovanni Stea

Reti Logiche

Corso di Laurea in Ingegneria Informatica, Universita' di Pisa

a.a. 2020/21

Testo di riferimento

- Paolo Corsini, Il calcolatore didattico C86.32, ETS, Pisa
 - Necessario averlo con voi durante le lezioni
 - Ci sono integrazioni sulla mia pagina web
 - http://docenti.ing.unipi.it/~a080368/Teaching/RetiLogiche/index_RL.html

Programmazione e assemblaggio

- Programma assemblato

```
000d E8970000
0012 3C2D
0014 7406
0016 3C2B
0018 7402
001a EBF1
001c E8940000
0021 A2020000
```



- Codice Assembler

```
in_sgn: CALL inchar
        CMP $'-' , %AL
        JE ok
        CMP $'+' , %AL
        JE ok
        JMP in_sgn
        CALL outchar
        MOV %AL, segno
```

XEP + disp

Assemblatore

- Mer
- gio

- GAS (GnuAssembler)

- www.delorie.com

- Scaricabile dalla mia pagina

- Il codice va scompattato nella cartella C:\amb_GAS
 - Cliccare installa.bat
 - Seguire le istruzioni alla lettera, non cambiare le cartelle
 - Altrimenti vi risolvete eventuali problemi da soli

- I programmi vanno scritti nella cartella C:\WORK

- Ci sono una serie di esempi
 - Vedremo poi i dettagli

Windows

Programma Assembler

- Sezione dati
 - Dichiarazione di variabili
 - Le variabili sono nomi simbolici per indirizzi di memoria che contengono i dati del programma
- Sezione codice
 - Istruzioni
- In un programma ci sono
 - **Direttive** (servono per l'assemblaggio e la dichiarazione di variabili)
 - **Istruzioni**

Note sintattiche

keyword , register
etc. chutte

MAUSOLI MINUSCOLI

- Ogni direttiva/istruzione **occupa una riga**, terminata da CR
 - Anche l'ultima riga deve essere terminata da CR

• GLOBAL main

. DATA
...

• EXIT

main

-Main
-Main

NOP

RE

Sottopref

-Cholin

Parole chiave

definiti dall' Ufficio

case-insensitive
registri

Case-sensitive

Esempio: conto dei bit a 1 in un long

```
# conteggio bit a 1 in un long
```

```
.GLOBAL _main
.DATA
dato:    .LONG 0x0F0F0101
conteggio: .BYTE 0x00

.TEXT
_main:    NOP
          MOVB $0x00, %CL
          MOVL dato, %EAX
comp:     CMPL $0x00, %EAX
          JE fine
          SHRL %EAX
          ADCB $0x00, %CL
          JMP comp
fine:     MOVB %CL, conteggio
          RET
```



```
0x00000100
0x00000101
0x00000102
0x00000103
0x00000104
...
0x00000200  MOVB $0x00, %CL
0x00000202  MOVL 0x00000100, %EAX
0x00000207  CMPL $0x000000, %EAX
0x0000020A  JE %EIP+$0x07
0x0000020C  SHRL %EAX
0x0000020E  ADCB $0x00, %CL
0x00000211  JMP %EIP-$0x0C
0x00000213  MOVB %CL, 0x00000104
0x00000218  ...
```

Note sintattiche

- **una riga di codice Assembler** è fatta in questo modo:

nome: KEYWORD operandi #commento [\CR]

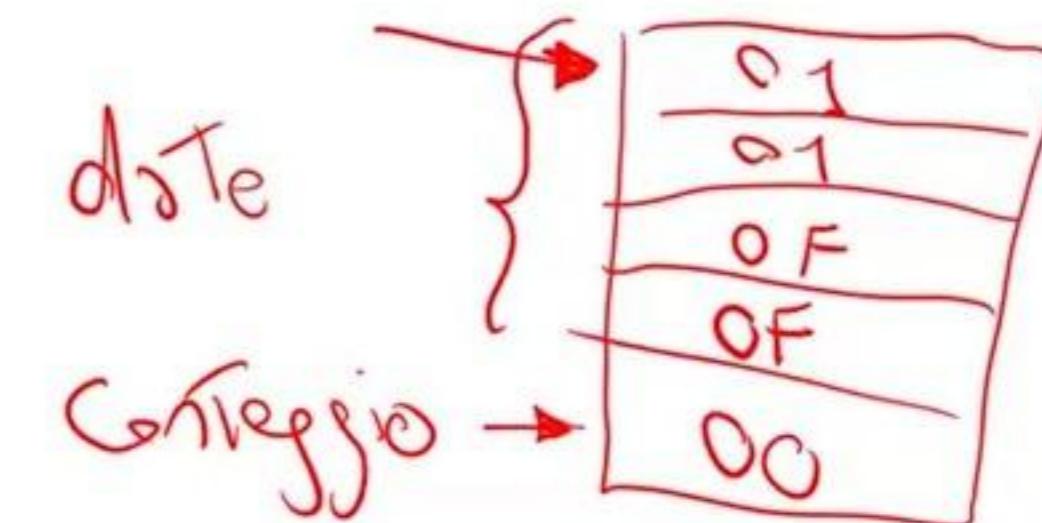
- Tutto puo' mancare, tranne il CR

Uno sguardo al codice

conteggio bit a 1 in un long

```
→ .GLOBAL _main  
→ .DATA  
- dato: .LONG 0x0F0F0101  
- conteggio: .BYTE 0x00
```

```
→ .TEXT →  
→ _main:  
  
definition →  
comp: →  
    NOP  
    MOVD $0x00, %CL  
    MOVI dato, %EAX  
    CMPD $0x00, %EAX  
    JE fine  
    SHRD %EAX  
    ADCD $0x00, %CL  
    JMP comp  
fine: →  
    MOVD %CL, conteggio  
    RET
```



|| devastante incomprensibilità

C++

MOV dato, %BL

2 passate

nomes1: /CR
nomes2: ~~CMP~~ - -

JMP nomes1

JMP nomes2

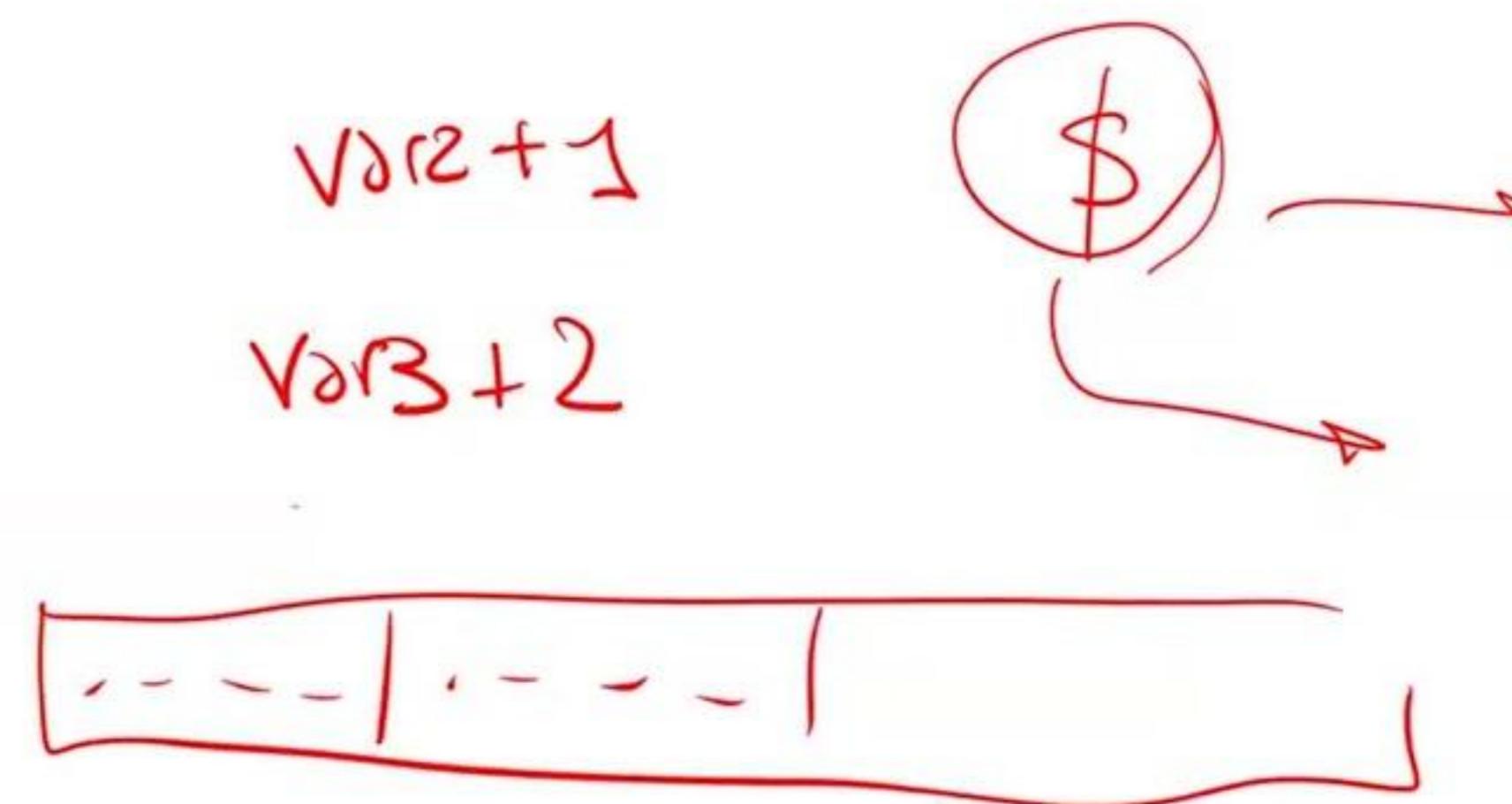
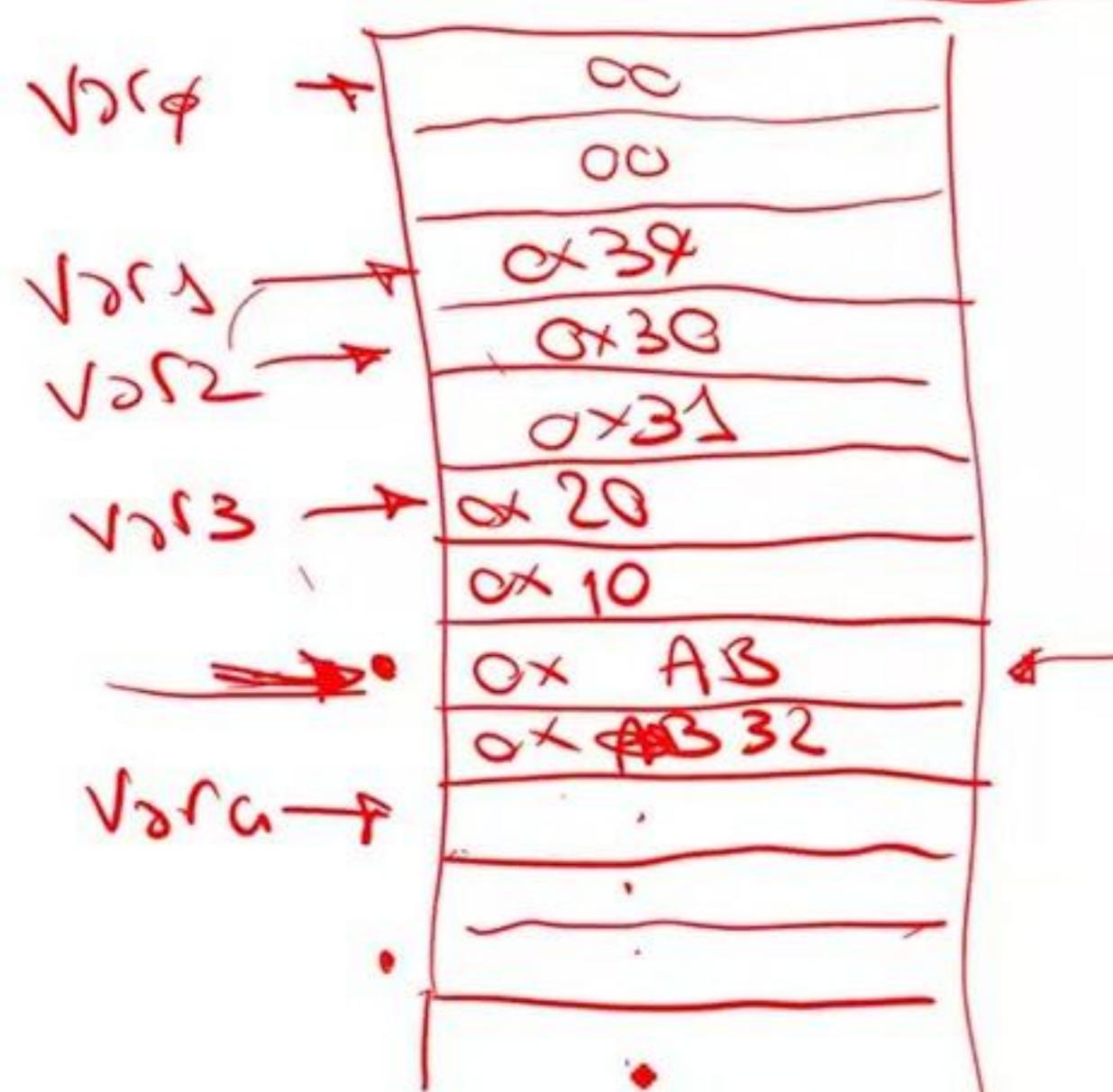
Direttive

.KEYWORD operandi [\CR]

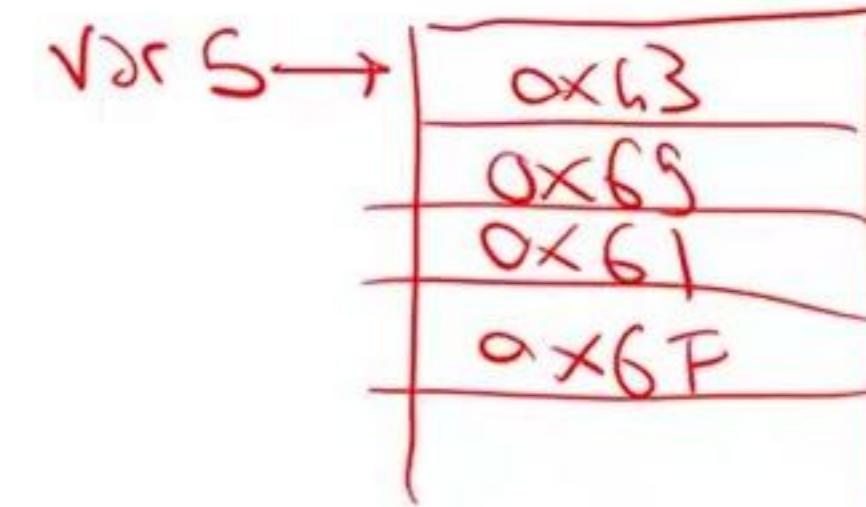
- **Dichiarazione di variabili**
- .BYTE: riserva 1 byte
- .WORD: riserva 2 byte
- .LONG: riserva 4 byte

Dichiarazioni di variabili

```
var0:      .WORD                 # scalare, 2 byte, valore 0x0000
var1:      .BYTE 0x30          # scalare, 1 byte, valore 0x30
var2:      .BYTE 0x30, 0x31, .. # vett., 2 componenti da 1 byte
var3:      .WORD 0x1020, 0x32AB # vett., 2 componenti da 2 byte
var4:      .LONG var3+2        # scalare, 4 byte
```



Dichiarazioni di variabile (cont.)



.FILL numero, dim, espressione

- Dichiara numero variabili di lunghezza dim e le inizializza ad espressione (default 0)
- dim può essere 1 (byte), 2 (word), 4 (long)

1bit → 8bit φ

- Codifiche ASCII

var5:.BYTE 'C', 'i', 'a', 'o' # vett., 4 comp. da 1 byte

var5:.BYTE 0x43, 0x69, 0x61, 0x6F

- Vettori di byte inizializzati con codifiche ASCII

• var6: .ASCII "messaggio" # vett., 9 comp. da 1 byte

var7: .ASCIZ "messaggio" # vett., 10 comp. da 1 byte
termina con \0 (NUL)

Dichiarazioni di variabili (cont.)

\n \t

- Per indicare i caratteri speciali (ritorno carrello, tabulazione, etc.) si usano le stesse sequenze di escape che si usano in C++.
- **Variabili dichiarate una di seguito all'altra nella sezione dati sono consecutive in memoria**
- se si usano variabili .BYTE, .WORD, .LONG, inserire sempre un'inizializzazione esplicita
- se non si vuole inizializzare una variabile, usare .FILL.

Altre direttive

.INCLUDE "path"

- Include un file sorgente nel presente file.
- L'assemblatore assembra un file unico contenente il codice di entrambi. Si mette, in genere, **in cima o in fondo**.

.SET nome, espressione

- Creare **costanti simboliche**. Tali costanti hanno il nome nome ed il valore espressione. Ad esempio, posso scrivere:

.SET dimensione, 4

costanti
indirizzi immidi

.SET n_iter, (100*dimensione)

...

MOV \$n_iter, %CX #op. immediato, ci vuole '\$'

Esempio

- Calcolo memoria occupata

```
dato1:    .FILL 1024, 4
dato2:    .FILL 100, 2
...
datoN:    .FILL 350, 2
→ foo:     .BYTE 1
           .SET occupazione, (foo-dato1) # memoria occupata
...
           MOV $occupazione, %ECX
```

Costanti numeriche

Mov \$128, %AL
Mov \$+128, %AL

- naturali: non hanno segno, e vengono convertite nella loro rappresentazione in base 2.

Mov \$10, %EAX
Mov \$0xAB,

- intero: hanno un **segno** davanti (+ o -), e vengono convertite nella loro rappresentazione in C2 sul numero di bit opportuno.

- I numeri possono essere scritti **in base 2, 8, 10, 16**.

- in base 2 devono essere preceduti da 0b,
- in base 8 devono cominciare per 0,
- in base 10 **non** devono cominciare per 0,
- in base 16 devono essere preceduti da 0x.

Mov \$32, %AL

Mov \$032, %AL

- Quando non sono della dimensione giusta, vengono

- Troncate se troppo lunghe, ed in genere l'assemblatore ve lo dice
- Estese, e normalmente l'assemblatore non ve lo dice

ob 01101011

b8
0 000
1 001
2 010
3 011
:
16

Controllo di flusso

- I costrutti di controllo di flusso tipici dei linguaggi ad alto livello devono essere implementati in termini di **istruzioni di salto**

- if...then...else
- for...
- while...
- do...while



JMP
JMP

CALL
RET

- Conviene continuare a ragionare in termini di costrutti ad alto livello, e tradurre in Assembler
 - Si scrivono programmi piu' chiari e meglio verificabili

If...then...else

JL

- if (%AX < variabile) // naturali
 { ist1; ...; istN; }
else
 { istN+1; ...; istN+M; }
ist_nuova;

• Invertire ramo then e ramo else

CMP variabile, %AX
JB ramothen
ramoelse: istN+1
...
istN+M
→ JMP segue
ramothen: ist1
{ ...
istN
segue: ist_nuova

• Invertire la condizione

CMP variabile, %AX
JAE ramoelse
ramothen: ist1
...
istN
• JMP segue
ramoelse: istN+1
{ ...
istN+M
segue: ist_nuova

Ciclo for

```
for (int i=0; i<var; i++)  
{ ist1; ...; istN }
```

while

// "var" variabile o costante

ciclo:

```
    MOV $0, %CX      ECX      CL  
    CMP var, %CX  
    JE fuori  
    ist1  
    ...  
    istN  
    INC %CX          // NON ADD $1, %CX (sta su 3 byte)  
    JMP ciclo
```

fuori: ...

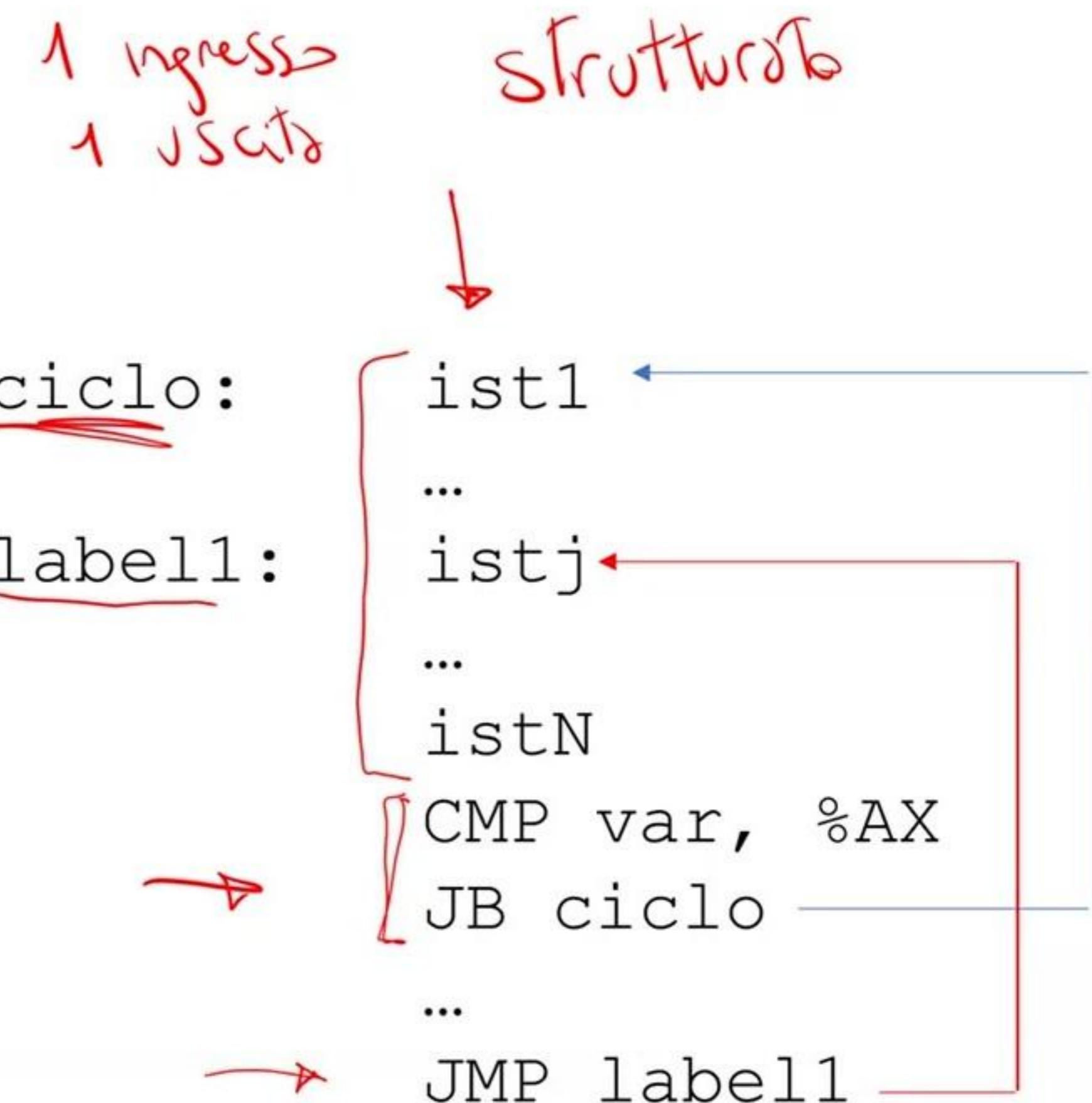
Ciclo do...while

```
{ do  
  { ist1; ...; istN }  
} while (AX<var);
```

```
ciclo:  { ist1  
          ...  
          istN  
        .  
        [ CMP var, %AX  
          JB ciclo ]
```

Spaghetti-like coding

- In Assembler si puo' saltare nel mezzo di un ciclo
 - JMP
 - ~~goto~~
- In C++ no
 - Per ottimi motivi
 - Se il controllo di flusso sembra un piatto di spaghetti, il programma e' **inverificabile**
- Linguaggi **strutturati** come C e Pascal pensati apposta per evitare questo stile di programmazione
 - **Unico** punto di ingresso, **unico** punto di uscita
- In Assembler, e' **il programmatore** che ci deve pensare (e lo **deve fare**)



Istruzione **LOOP**

- Decrementa **ECX** e salta se ECX!=0

→ MOV \$5, %ECX
ciclo: ist1
 ...
 istN;
 LOOP ciclo .

- ECX va inizializzato al **numero di iterazioni desiderate**
 - E **non va toccato** durante il ciclo
 - Altrimenti si scrivono programmi incomprensibili
 - Utile per tradurre cicli **for** (numero di iterazioni noto)

Istruzione LOOP

- Ciclo for con variabile di conteggio i
 - Descendente, oppure ascendente, ma dove non uso i

```
for (int i=var; i>0; i--)  
{ist1;  
 //op. che usa i  
 ...;  
 istN;}
```



```
MOV var, %ECX  
ciclo: ist1  
...  
# ECX usato per i  
istN  
LOOP ciclo
```



```
for (int i=0; i<var; i++)  
{ist1;  
 //op. che usa i  
 ...;  
 istN;}
```



```
→ MOV var, %ECX  
→ MOV $0, %EBX  
ciclo: ist1  
...  
# EBX usato per i  
istN  
LOOP ciclo  
→ INC %EBX  
→ CMP var, %EBX  
→ JE cicl
```

LOOP condizionali

Loop

- LOOPE, LOOPNE (LOOPZ, LOOPNZ)
- • Sensate soltanto **dopo una CMP**
- Il salto avviene se:
 1. La condizione e' vera
 2. Dopo il decremento, ECX!=0
- Si scrive in ECX il numero **massimo** di iterazioni
 - Il ciclo puo' terminare prima, se la condizione diventa falsa

ciclo:

```
MOV $10, %ECX  
[ ist1  
...  
istN  
CMP <src>, <dest>  
LOOPcond ciclo
```

LoopE ciclo



Controllo di flusso

- LOOP, LOOPcond non indispensabili
 - Possono essere sostituiti da CMP+Jcon
 - Le trovate in appendice ai miei appunti

- Istruzioni per la manipolazione di **stringhe**
 - Hanno prefissi di ripetizione (quindi implementano cicli)
 - Servono per accessi sequenziali in memoria
 - Le vedremo piu' avanti

Sottoprogrammi e passaggio dei parametri

- **CALL**
 - Non prevede passaggio di parametri al sottoprogramma chiamato
- **RET**
 - Non prevede un valore di ritorno per il chiamante
- E' necessario stabilire delle **convenzioni** e rispettarle
 1. Usare locazioni di memoria condivise
 2. Usare registri
- In Assembler non esiste il concetto di **variabile locale** ad un sottoprogramma
 - La memoria principale e' indirizzabile da qualunque sottoprogramma

Sottoprogrammi e passaggio dei parametri

- Chi scrive un sottoprogramma **deve** specificare I parametri di ingresso e di uscita del sottoprogramma
 - **Commenti** nel codice

```
MOV ..., %AX          # preparazione dei parametri
MOV ..., %EBX         # per la chiamata di sottoprogramma
CALL sottoprg
MOV %CX, var          # utilizzo del valore di ritorno

# sottoprogramma "sottoprg", [descrizione]
# ingresso:  %AX, [descrizione]           locazioni*
#              %EBX, [descrizione]
# uscita:    %CX,  [descrizione]

sottoprg: ...
...
MOV $..., %CX # preparazione del valore di ritorno
RET
```

Uso dei registri ed effetti collaterali

• CALL sotoprg
..
..

- I registri che non contengono valori di ritorno **non devono essere sporcati** da un sottoprogramma

- Chi chiama il sottoprogramma deve trovarci gli stessi valori che ci ha lasciato
- I registri che un sottoprogramma utilizza **devono essere salvati in pila** e ripristinati alla fine

sotoprg:

```
PUSH ...1,AX          # registro1 usato dal sottoprogramma
PUSH ...2,BX          # registro2 usato dal sottoprogramma
...
MOV $..., 3,CX        # preparazione del valore di ritorno
POP ...2,BX          # registro2 usato dal sottoprogramma
POP ...1,AX          # registro1 usato dal sottoprogramma
RET
```

A red curly brace on the left side of the assembly code groups the first four lines (PUSH, ..., MOV) as "what goes in". A red circle highlights the CX register in the MOV instruction. Red X's are placed over the values in the comments to indicate they are no longer valid.

Salvare i registri in pila

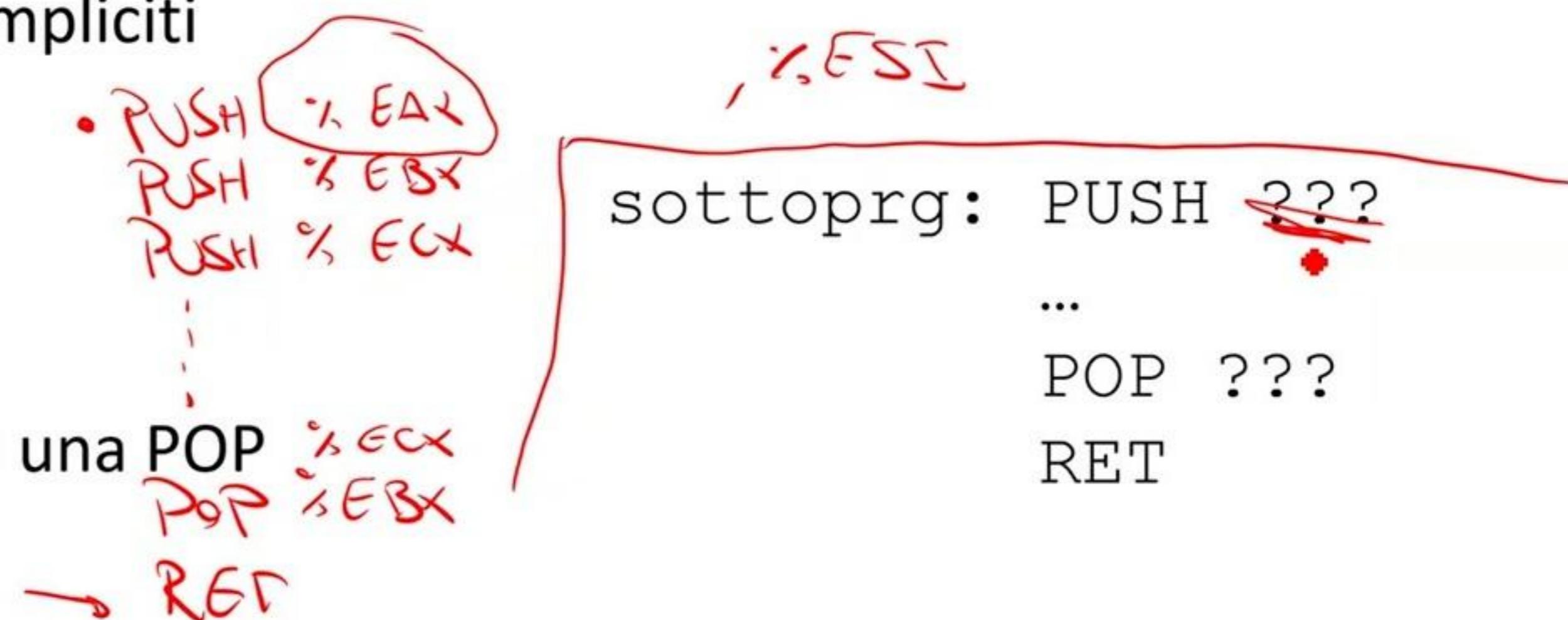
- Tutti quelli che uso **e** non servono a contenere valori di ritorno
- Attenzione: non sempre si vedono scorrendo il codice

- MUL e DIV hanno operandi impliciti
- Spesso sporcano (E)DX

~~→~~

- La pila va **lasciata in ordine**

- Per ogni PUSH ci deve essere una POP
- Nell'ordine inverso



- Altrimenti quando la RET pesca l'indirizzo di ritorno dalla pila ci trova valori casuali, e il programma si inchioda

Il sottoprogramma principale

- Il main va in esecuzione come **sottoprogramma**

- Deve terminare con una RET
- Chi lo chiama si aspetta un valore di ritorno dentro %EAX

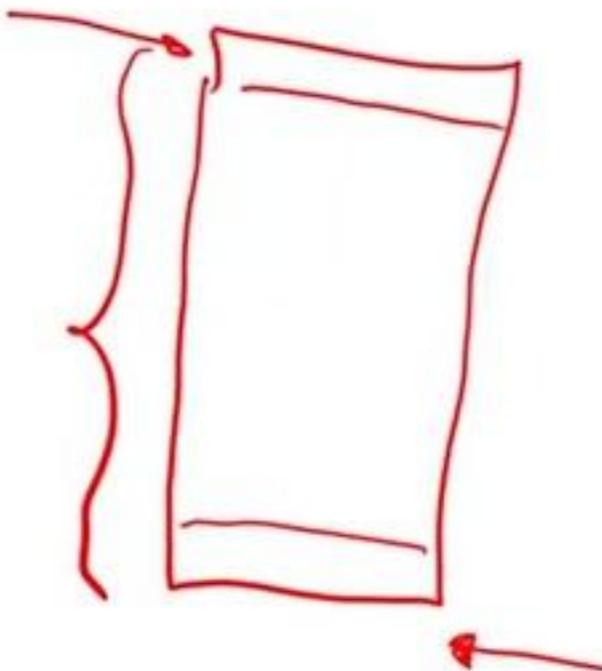
- 0: tutto ok
- Altro valore: codice di errore

- Prassi usata spesso:


- Non e' un obbligo per questo esame

_main: NOP
 ...
 XOR %EAX, %EAX
 RET

Dichiarazione dello stack



- Lo stack esiste e funziona correttamente se
 - a) qualcuno lo dichiara
 - b) Si inizializza %ESP
- "dichiarare" lo stack -> **riservare abbastanza memoria**
- %ESP va inizializzato alla **cella successiva al fondo dello stack**

```
.DATA
...
mystack:    * .FILL 1024,4          #dichiarazione stack
.SET         initial_esp, (mystack + 1024*4)

.TEXT
_main: _NOP
      MOV $initial_esp, %ESP      #inizializzazione stack
```

Dichiarazione dello stack

- Quanto va fatto grande lo stack?
 - E' un problema del programmatore
- Nel **nostro ambiente** (ma non nell'Assembler in generale) possiamo **omettere la dichiarazione** dello stack
 - Ci pensa l'ambiente, secondo regole sue
- Terzo modo di passare parametri: usare la **pila**
 - Lo fanno i compilatori
 - • Lo vedrete a Calcolatori Elettronici, quando farete la compilazione
 - Difficile da fare a mano, quindi sconsigliato

Ingresso/uscita di caratteri e stringhe

CALL inchar
CALL outchar

- includere il file utility con l'apposita direttiva
 - .INCLUDE "C:/amb_GAS/utility"
- Alcuni sottoprogrammi di I/O
 - inchar: mette in AL (par. uscita) la codifica ASCII del tasto premuto
 - Non fa l'eco sul video. Se voglio che il carattere digitato appaia sul video, devo usare:
 - outchar: mette sul video la codifica ASCII contenuta in AL (parametro ingresso)
 - newline: serve per andare a capo
 - Stampa i caratteri 0x0D (Carriage Return) e 0x0A (Line Feed). Servono **entrambi**.
 - pauseN: mette in pausa il programma e stampa sul video.
Checkpoint number N. Press any key to continue
 - N deve essere una cifra decimale

pause

Ingresso/uscita e sottoprogrammi di utilità

- In Assembler non esistono istruzioni di ingresso/uscita

- Solo IN e OUT su interfacce

privilegiate

- Ma non le possiamo usare (sono istruzioni protette, non utilizzabili in modalita' utente)

~~IN~~ eccezione ~~OUT~~ Protezione

8° x 25

- I/O tastiera/video: si usano servizi del Sistema (DOS)

- Sottoprogrammi scritti da altri, che girano in modalita' sistema, e possono usare le IN/OUT

- Molto primitivi: ingresso e uscita di un carattere

- Sopra questi, costruiti altri servizi che consentono di fare cose di livello un po' piu' alto (ma non troppo)

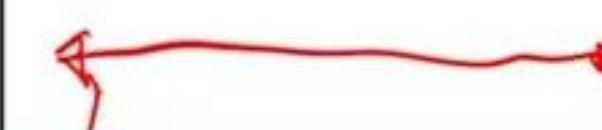
IN ← Tastiera

OUT → video

ASCII TABLE

7 bit

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]	48	30	0110000	60	0	96	60	1100000	140	`
1	1	1	1	[START OF HEADING]	49	31	0110001	61	1	97	61	1000001	141	a
2	2	10	2	[START OF TEXT]	50	32	0110010	62	2	98	62	1100010	142	b
3	3	11	3	[END OF TEXT]	51	33	0110011	63	3	99	63	1100011	143	c
4	4	100	4	[END OF TRANSMISSION]	52	34	0110100	64	4	100	64	1100100	144	d
5	5	101	5	[ENQUIRY]	53	35	0110101	65	5	101	65	1100101	145	e
6	6	110	6	[ACKNOWLEDGE]	54	36	0110110	66	6	102	66	1100110	146	f
7	7	111	7	[BELL]	55	37	0110111	67	7	103	67	1100111	147	g
8	8	1000	10	→ [BACKSPACE] ←	56	38	0111000	70	8	104	68	1101000	150	h
9	9	1001	11	[HORIZONTAL TAB]	57	39	0111001	71	9	105	69	1101001	151	i
10	A	1010	12	→ [LINE FEED]	58	3A	0111010	72	.	106	6A	1101010	152	j
11	B	1011	13	[VERTICAL TAB]	59	3B	0111011	73	;	107	6B	1101011	153	k
12	C	1100	14	[FORM FEED]	60	3C	0111100	74	<	108	6C	1101100	154	l
13	D	1101	15	→ [CARRIAGE RETURN]	61	3D	0111101	75	=	109	6D	1101101	155	m
14	E	1110	16	[SHIFT OUT]	62	3E	0111110	76	>	110	6E	1101110	156	n
15	F	1111	17	[SHIFT IN]	63	3F	0111111	77	?	111	6F	1101111	157	o
16	10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	@	112	70	1110000	160	p
17	11	10001	21	[DEVICE CONTROL 1]	65	41	1000001	101	→ A ←	113	71	1110001	161	q
18	12	10010	22	[DEVICE CONTROL 2]	66	42	1000010	102	B	114	72	1110010	162	r
19	13	10011	23	[DEVICE CONTROL 3]	67	43	1000011	103	C	115	73	1110011	163	s
20	14	10100	24	[DEVICE CONTROL 4]	68	44	1000100	104	D	116	74	1110100	164	t
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]	69	45	1000101	105	E	117	75	1110101	165	u
22	16	10110	26	[SYNCHRONOUS IDLE]	70	46	1000110	106	F	118	76	1110110	166	v
23	17	10111	27	[ENG OF TRANS. BLOCK]	71	47	1000111	107	G	119	77	1110111	167	w
24	18	11000	30	[CANCEL]	72	48	1001000	110	H	120	78	1111000	170	x
25	19	11001	31	[END OF MEDIUM]	73	49	1001001	111	I	121	79	1111001	171	y
26	1A	11010	32	[SUBSTITUTE]	74	4A	1001010	112	J	122	7A	1111010	172	z
27	1B	11011	33	[ESCAPE]	75	4B	1001011	113	K	123	7B	1111011	173	{
28	1C	11100	34	[FILE SEPARATOR]	76	4C	1001100	114	L	124	7C	1111100	174	
29	1D	11101	35	[GROUP SEPARATOR]	77	4D	1001101	115	M	125	7D	1111101	175	}
30	1E	11110	36	[RECORD SEPARATOR]	78	4E	1001110	116	N	126	7E	1111110	176	~
31	1F	11111	37	[UNIT SEPARATOR]	79	4F	1001111	117	O	127	7F	1111111	177	[DEL]
32	20	100000	40	[SPACE]	80	50	1010000	120	P					
33	21	100001	41	!	81	51	1010001	121	Q					
34	22	100010	42	"	82	52	1010010	122	R					
35	23	100011	43	#	83	53	1010011	123	S					
36	24	100100	44	\$	84	54	1010100	124	T					
37	25	100101	45	%	85	55	1010101	125	U					
38	26	100110	46	&	86	56	1010110	126	V					
39	27	100111	47	'	87	57	1010111	127	W					
40	28	101000	50	(88	58	1011000	130	X					
41	29	101001	51)	89	59	1011001	131	Y					
42	2A	101010	52	*	90	5A	1011010	132	Z					
43	2B	101011	53	+	91	5B	1011011	133	[
44	2C	101100	54	,	92	5C	1011100	134	\					
45	2D	101101	55	-	93	5D	1011101	135]					
46	2E	101110	56	.	94	5E	1011110	136	^					
47	2F	101111	57	/	95	5F	1011111	137	-					



Cifre: 0x30 ('0') - 0x39 ('9')

Maiuscole: 0x41 ('A') - 0x5A ('Z')

Minuscole: 0x61 ('a') - 0x7A ('z')

minuscole <-> maiuscole

set/reset bit n. 5

caratteri speciali:

ritorno carrello 0xD,

avanzamento linea è 0xA

backspace 0x08

I/O tastiera e video

- Entrano (dalla tastiera) ed escono (sul video) **codifiche ASCII di singoli caratteri**
- Non esiste I/O di variabili **tipate**
- “stampa il numero 32”
 - Stampa prima il carattere ASCII **0x33** ('3')
 - E poi il carattere ASCII **0x32** ('2')
- Per fare ingresso **da tastiera** di un numero naturale a 2 cifre in base 10
 - Leggo e memorizzo **due codifiche ASCII** c_1, c_0 [-----]
 - Calcolo le singole cifre decimali a_1, a_0 a partire dalle codifiche c_1, c_0
 - Ricostruisco il numero digitato come $10 \cdot a_1 + a_0$

Ingresso/uscita di caratteri e stringhe

CALL inchar
CALL outchar

- includere il file utility con l'apposita direttiva

.INCLUDE "C:/amb_GAS/utility"

- Alcuni sottoprogrammi di I/O

- inchar: mette in AL (par. uscita) la codifica ASCII del tasto premuto
 - Non fa l'eco sul video. Se voglio che il carattere digitato appaia sul video, devo usare:
- outchar: mette sul video la codifica ASCII contenuta in AL (parametro ingresso)
- newline: serve per andare a capo
 - Stampa i caratteri 0x0D (Carriage Return) e 0x0A (Line Feed). Servono **entrambi**.
- pauseN: mette in pausa il programma e stampa sul video.

Checkpoint number N. Press any key to continue

- N deve essere una cifra decimale

pause

Ingresso/uscita di caratteri e stringhe (cont.)

- **Inline**

- Consente di portare una stringa di max 80 caratteri in un buffer di memoria, digitando da tastiera con eco su video
- Parametri di ingresso
 - EBX: indirizzo di memoria del buffer
 - CX: numero di caratteri da leggere (max 80, una riga)
CX - 2
 - ES
- La lettura da tastiera termina dopo 78 caratteri o se premete "Invio"
 - vengono **aggiunti** in coda al buffer i caratteri LF (ASCII 10, 0x0A) e CR (ASCII 13, 0x0D)
- Il sottoprogramma interpreta un carattere di backspace (ASCII 8) come l'ordine di cancellare dal buffer e dal video l'ultimo carattere digitato

Ingresso/uscita di caratteri e stringhe (cont.)

- outline

- Parametri di ingresso

- EBX: indirizzo di memoria del buffer

- stampa a video max 80 caratteri. Si ferma prima se trova un carattere di ritorno carrello (0x0D), stampando anche i caratteri necessari ad andare a capo.

inline

LF CR

MOV \$-1;[CX]

CALL offset

.....

65535

- outmess

- Parametri di ingresso

- EBX: indirizzo di memoria del buffer

- CX: numero di caratteri da stampare a video

Ingresso/uscita di numeri esadecimali

- **inbyte, inword, inlong:**

- prelevano da tastiera (con echo a video) 2, 4, o 8 caratteri dalla tastiera.
- Interpretano tale sequenza di caratteri come un numero esadecimale a 2, 4, o 8 cifre
- mettono in AL, AX, EAX il numero esadecimale digitato.
- Ignorano qualunque altro carattere che viene premuto.

- **outbyte, outword, outlong:**

- stampano sul video, rispettivamente, 2, 4, o 8 caratteri, corrispondenti a cifre esadecimali, ottenute interpretando il contenuto di AL, AX, EAX come un numero naturale.

003A 3A



Ingresso/uscita di numeri decimali

3

- indecimal_byte, indecimal_word, indecimal_long:

- prelevano da tastiera (con eco a video) fino a 3, 5, o 10 cifre decimali.
- Interpretano tale sequenza di caratteri come un numero decimale
- lo mettono in AL, AX, EAX.
- Ignorano qualunque altro carattere che viene premuto.
- Se il numero decimale digitato e' troppo grande viene troncato
- Si puo' premere invio per dare in ingresso meno di 3, 5, 10 cifre

982
0.255

- outdecimal_byte, outdecimal_word, outdecimal_long:

- stampano sul video il contenuto di AL, AX, EAX, interpretato come un numero naturale, sul numero di cifre strettamente necessario.

b₁₀

31

Istruzioni che manipolano le stringhe

- In Assembler **non esistono tipi di dati né strutture dati.**

- Esistono soltanto byte, word e long.

- L' Assembler supporta però il concetto di **vettore**

- **dichiarare vettori** di variabili di una certa dimensione;
 - indirizzamento con **displacement + registri base/indice**

- **Istruzioni stringa**

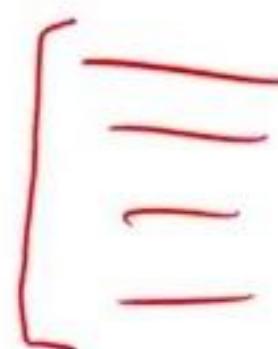
efficiente

- **stringa** \Leftrightarrow vettore

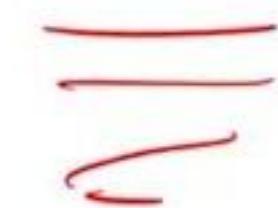
(no relation to «char»)

- Servono a copiare **interi buffer di memoria**

- Usano i registri **%ESI**, **%EDI** come puntatori (S: source, D: destination)



$$\begin{array}{c} \rightarrow a = b \\ | \\ 200 \end{array}$$



Esempio: copia di un vettore

- Ci vogliono un po' di istruzioni

```
vett_sorg: .FILL 1000, 4  
vett_dest: .FILL 1000, 4
```

```
[...]  
MOV $1000, %ECX  
LEA vett_sorg, %ESI  
LEA vett_dest, %EDI  
MOV (%ESI), %EAX  
MOV %EAX, (%EDI)  
[ ADD $4, %ESI  
  ADD $4, %EDI  
  LOOP ciclo]
```

ciclo:

```
ciclo: MOV vett_sorg(,%ECX,4), %EAX  
        MOV %EAX, vett_dest(,%ECX,4)  
        → LOOP ciclo  
  
→ MOV $1000, %ECX  
→ LEA vett_sorg, %ESI  
→ LEA vett_dest, %EDI  
→ REP MOVSL
```

MOVE DATA FROM STRING TO STRING (with REPEAT)

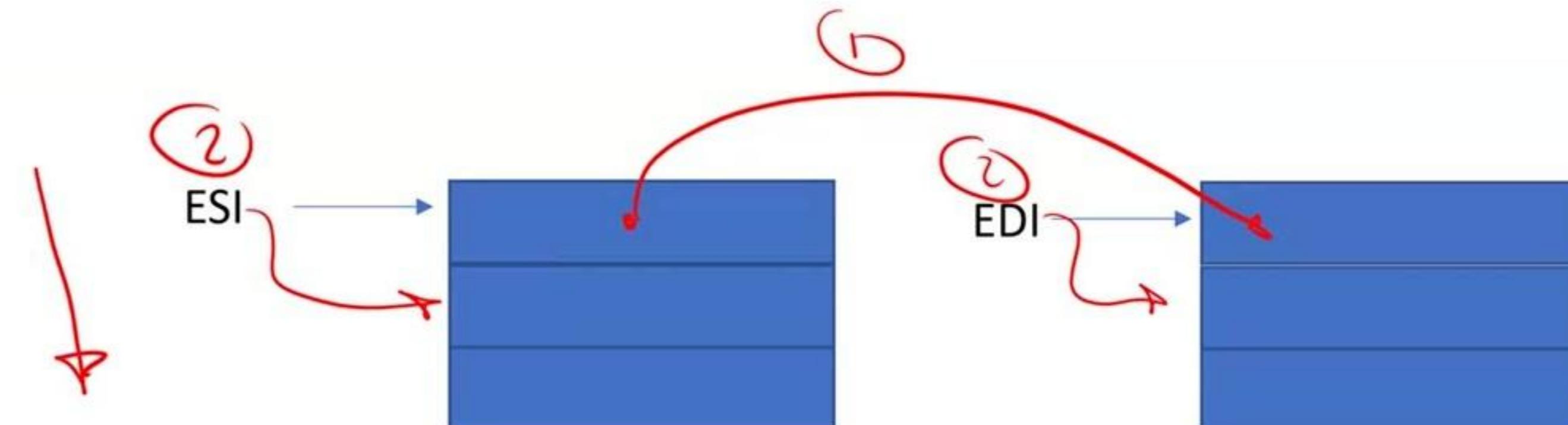
- **FORMATO:**

~~MOVSSuf~~ ESI EDI 1, 2, 5
~~REP~~ MOVSSuf ← ECX

- **AZIONE:** Copia il numero di byte specificato dal suffisso *suf* dall'indirizzo di memoria puntato da ~~ESI~~ all'indirizzo di memoria puntato da ~~EDI~~. Se DF=0, somma ad ESI e ad EDI il numero di byte specificato dal suffisso. Se DF=1, sottrae da ESI e da EDI il numero di byte specificato dal suffisso.
- Se viene premesso il prefisso REP, allora le azioni indicate sopra vengono ripetute per il numero di volte specificato in ECX, che viene decrementato fino a zero.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

MOVE DATA FROM STRING TO STRING (with REPEAT)

- ~~MOVSSuf~~
~~B~~



- REP MOVSSuf



- Direzione della copia: Avanti (DF=0) o Indietro (DF=1)

Direction flag

STD
CLD

- E' un altro bit nel registro dei flag
- STD (**set** direction flag): imposta DF a 1 (quindi abilita la copia "all'indietro")
- CLD (**clear** direction flag): imposta DF a 0 (quindi abilita la copia "in avanti")
- Nel codice scritto prima, quindi, e' necessario scrivere:
 - CLD
 - REP MOVSL
- Perche' **DF** ha il valore che gli ha dato l'ultima istruzione che lo ha toccato (eseguita da chiunque, non necessariamente da noi)

Altre istruzioni stringa

LOAD

• **LODSsuf:**

ESI
mem → *AL*

- copia in **AL**, **AX**, oppure **EAX** (a seconda del suffisso) il contenuto della memoria all'indirizzo puntato da **ESI**.
- A seconda del valore del flag **DF**, incrementa o decrementa di **1, 2, 4 ESI**.

STORE

• **STOSsuf:**

AL → *mem*
EDI

- copia il registro **AL**, **AX**, oppure **EAX** (a seconda del suffisso) in memoria all'indirizzo puntato da **EDI**.
- A seconda del valore del flag **DF**, incrementa o decrementa di **1, 2, 4 EDI**.

- ||*
- I registri usati come puntatore implicito sono **differenti**: **ESI** come sorgente, **EDI** come destinatario

Esempi di codice che usa istruzioni stringa

- Copiare un vettore da una parte ad un'altra della memoria,
~~eseguendo la stessa operazione~~
su tutti i suoi elementi

X → MOV \$1000, %CX
LEA buffer_src, %ESI
LEA buffer_dst, %EDI
CLD
ciclo: • LODSL +u %ESI •
modifica %EAX
• STOSL +u %EAX
LOOP ciclo

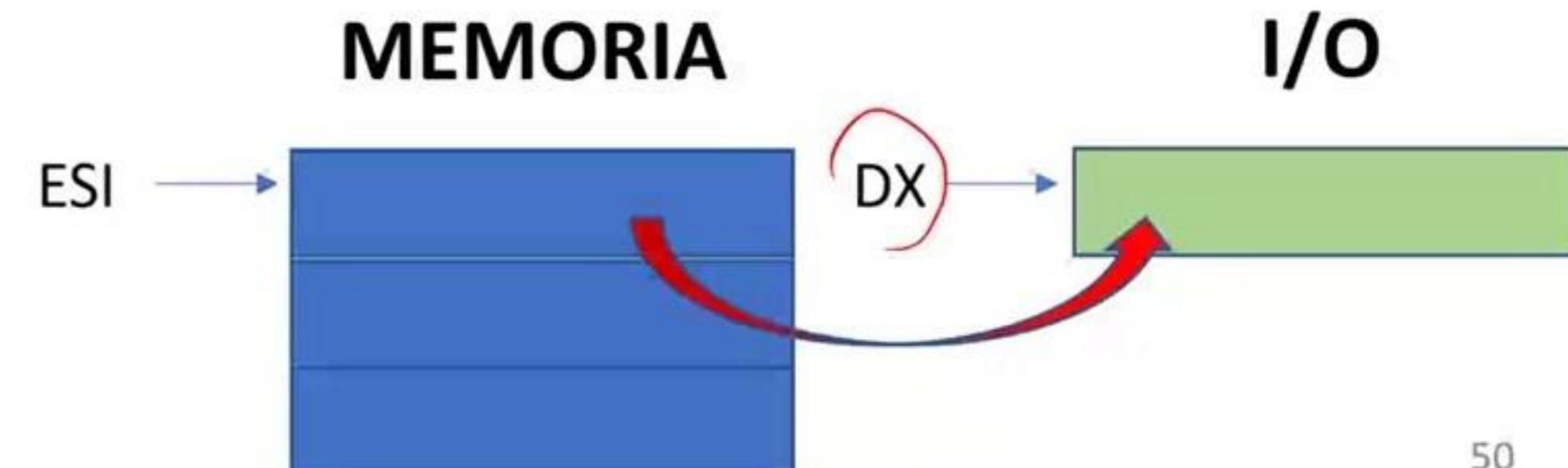
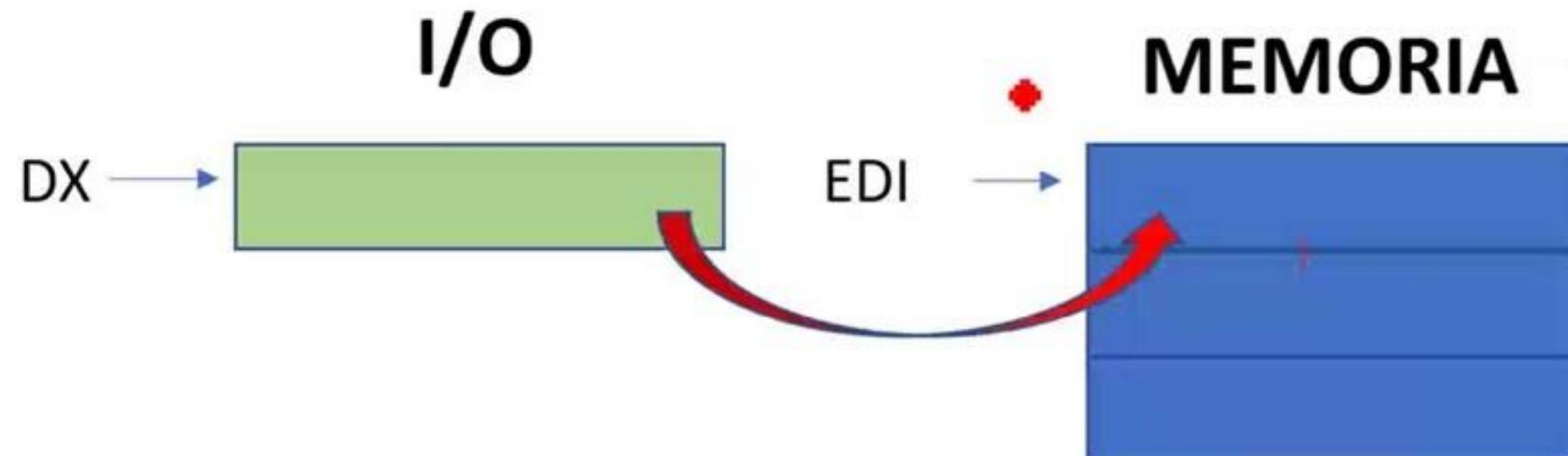
- Riempire un buffer di memoria con lo stesso valore (e.g., zero)

→ MOV \$1000, %ECX
→ LEA buffer, %EDI
→ XOR %EAX, %EAX
→ CLD
REP STOSL
EAX → (%EDI)

Istruzioni stringa per l'I/O

(non le potevo usare)

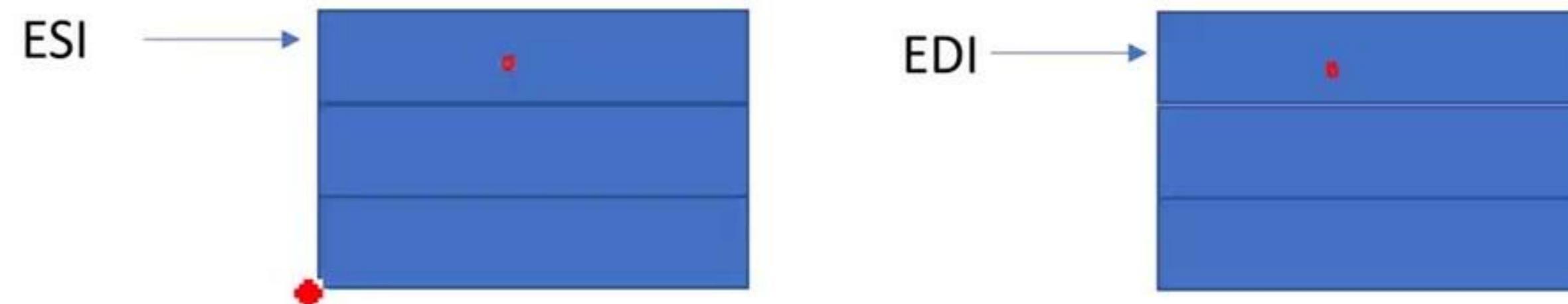
- **INSSsuf:** fa ingresso di uno, due, quattro byte dalla porta di I/O il cui offset è contenuto in **DX**. L'operando viene inserito in memoria a partire dall'indirizzo di memoria contenuto in **EDI**. A seconda del valore del flag **DF**, incrementa o decrementa di 1, 2, 4 il contenuto di **EDI**.
- **OUTSsuf:** copia uno, due, quattro byte, contenuti in memoria a partire dall'indirizzo di memoria contenuto in **ESI**, alla porta di I/O il cui offset è contenuto in **DX**. A seconda del valore del flag **DF**, incrementa o decrementa di 1, 2, 4 il contenuto di **ESI**.



COMPARE STRINGS

- **CMPSSuf:** confronta il contenuto delle locazioni (doppiie locazioni, quadruple locazioni) indirizzate da **ESI** (sorgente) ed **EDI** (destinatario). A seconda del valore del flag DF, incrementa o decrementa di 1, 2, 4 il contenuto di **ESI**, **EDI**.

CMP



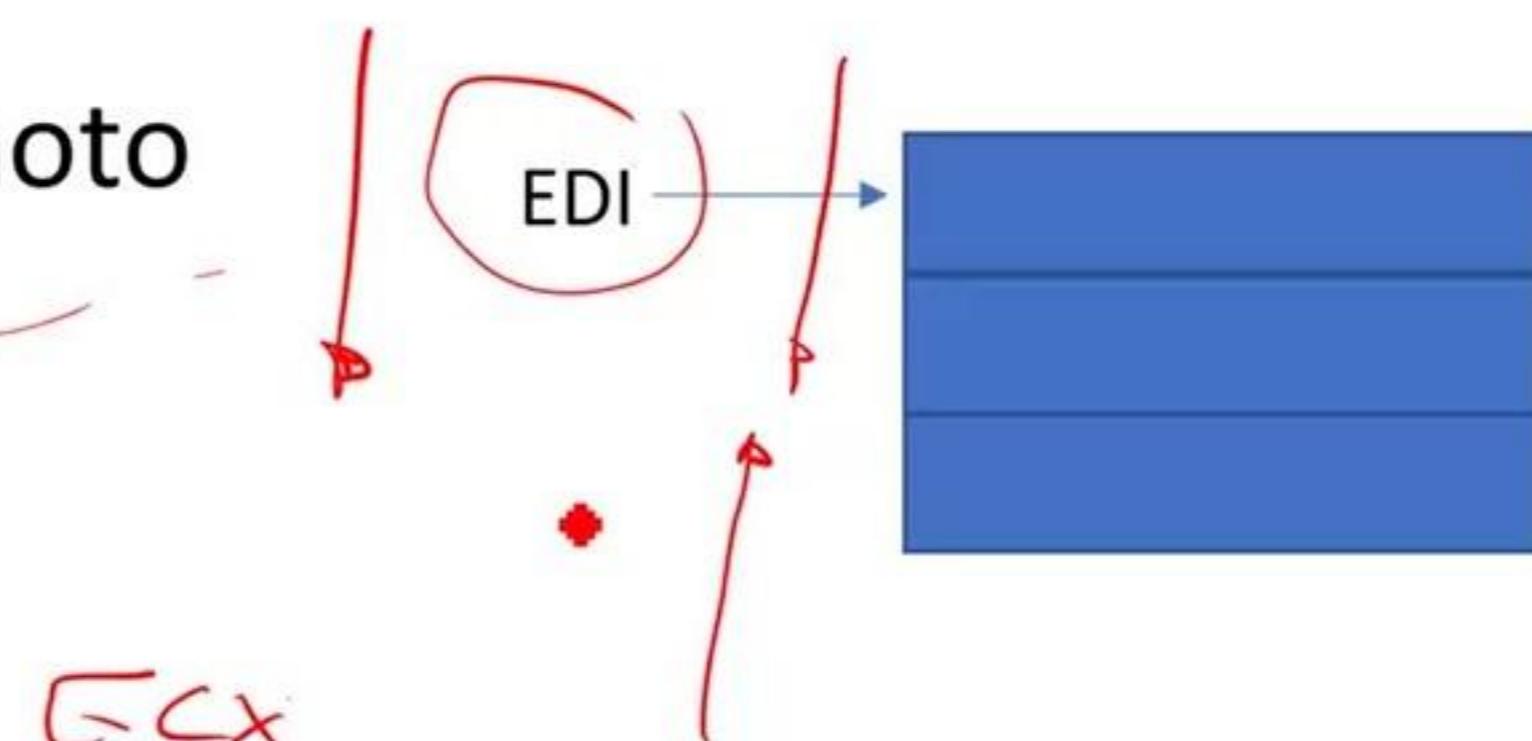
SCAN STRING

EAX

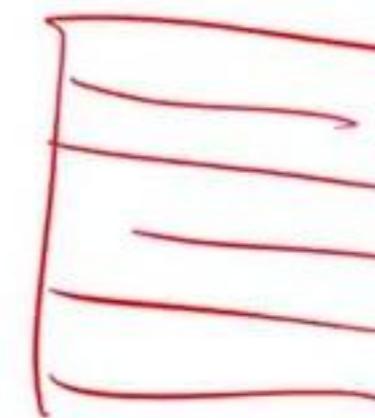
- **SCASsuf:** Confronta il contenuto del registro **AL**, **AX**, oppure **EAX** (a seconda del suffisso) con la locazione (doppia locazione, quadrupla locazione) di memoria indirizzata da **EDI**. L'algoritmo usato nel confronto è identico a quello della CMP. A seconda del valore del flag DF, incrementa o decrementa di **1**, **2**, **4** il contenuto di **EDI**.

Serve a trovare un elemento di valore noto
dentro a un vettore

- DF = 0: cerca la prima occorrenza
- DF = 1: cerca l'ultima occorrenza



Prefissi di ripetizione



A^x

REP LODS

- REP

- Puo' essere usato con MOVS, LODS, STOS, INS, OUTS.
- Utilizzo con LODS è del tutto privo di senso, in quanto si finisce a sovrascrivere N volte lo stesso registro.
- Si applica ad una istruzione (non ad un blocco di codice)

- ✖ • REPE / REPNE

REPE SCASL

- Puo' essere usato con CMPS, SCAS.
- Si applica ad una istruzione (non ad un blocco di codice)
- Si fanno al **massimo ECX ripetizioni**, finche' la condizione specificata e' vera

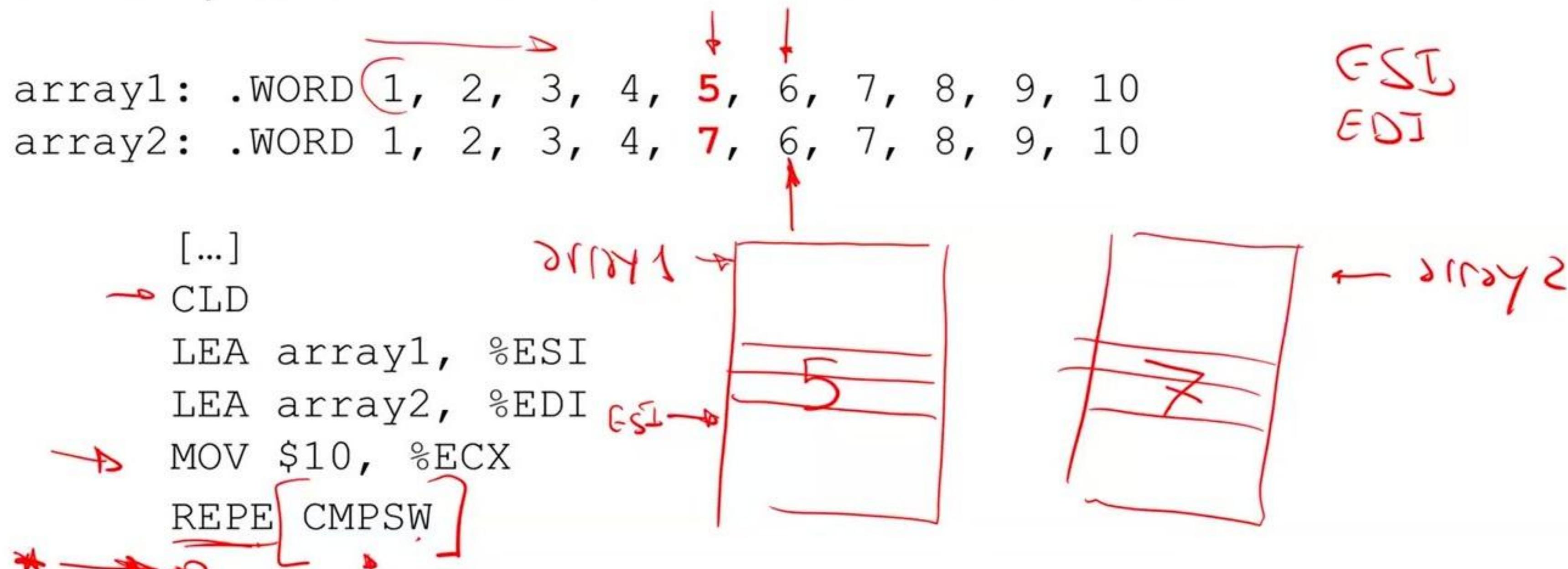
DP = Ø



Mov \$100,%ECX
REPNE CMPSL

Esempio

- Trovare il primo elemento differente tra due vettori



- Attenzione: all'uscita dall'istruzione %ESI, %EDI puntano all'elemento successivo (infatti la CMPS li ha incrementati)

Perche' due direzioni (DF=0, DF=1)?

- Trovare la prima / l'ultima occorrenza di un dato in un vettore
 - SCASSuf, ma devo poter impostare DF
- Copia di buffer parzialmente sovrapposti

dissjunti

