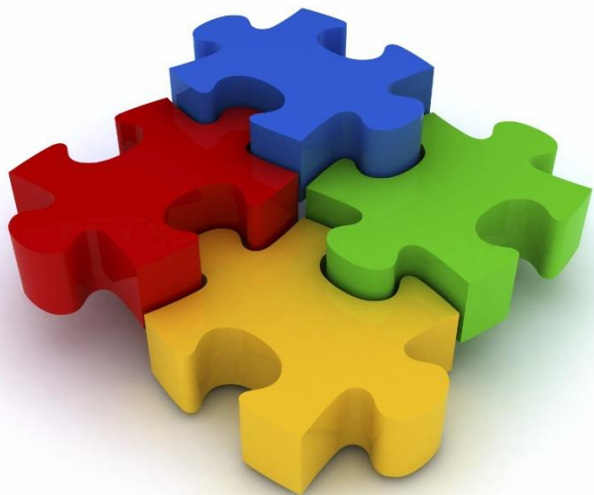


# פרק 9

## מחסנית ופרוצדורות מבוא ונושאים בסיסיים

# תוכנית מודולרית



▶ תוכנית שבנויה מחלקי קוד  
נפרדים המשולבים זה בזה

▶ מאפיינים:

- נקודת כניסה אחת
- נקודת יציאה אחת
- מבצעים פעולה מוגדרת
- נקראים פרוצדורות

# יתרונות כתיבת קוד מודולרי

## קוד קצר יותר ▶

- לא צריך לחזור על חלקי קוד
- שאלה למחשבה: האם אפשר לחזור על קטעי קוד ע"י פקודת  
?jmp

OpenComputer:

    jmp       ReadPassword

    ...

    ; Code for signing into computer

OpenEmail:

    jmp       ReadPassword

    ...

    ; Code for signing into email

ReadPassword:

    ...

    ; Code for reading password from user

# יתרונות כתיבת קוד מודולרי - המשך

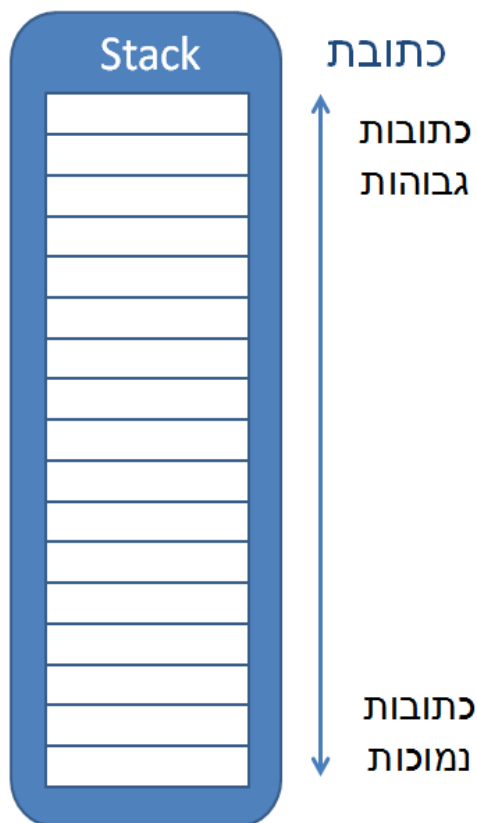
---

- ▶ חלוקת הקוד לחלקים מקלה על מציאה ותיקון באגים
- ▶ קוד קצר ופשוט לקריאה:
- ▶ Call      ReadPassword
- ▶ יכולת לשתף קוד
  - בין תוכניות
  - בין מתכנתים

# חסרונות קוד מודולרי

- ▶ שיתוף קוד הוא פתח לבעיות
  - באגים
  - בעיות אבטחה
- ▶ יותר מסובך לכתוב קוד מודולרי
- ▶ דורש יותר משאבי מחשב
  - זיכרון
  - פקודות נוספות (כניסה ויציאה מפרוצדורה)

# המחסנית Stack



▶ סגמנט בזיכרון  
▶ משמשת לאחסון לזמן קצר של:

- משתנים
- קבועים
- כתובות

▶ כלי חשוב בעבודה עם פרוצדורות

# הגדרת מחסנית בתוכנית

---

▶ stack    NumOfBytes

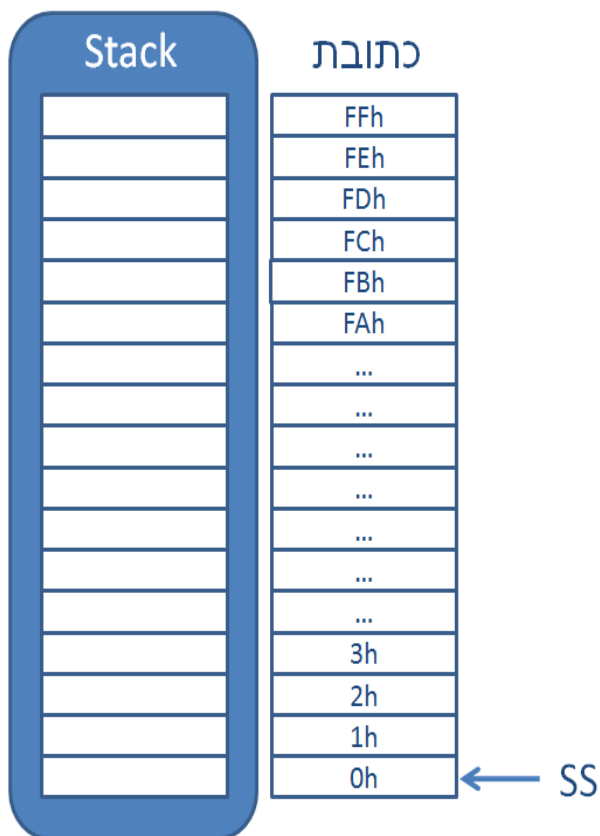
▶ לדוגמה:

▶ stack    100h

▶ בדוגמה, כמה בתים יש במחסנית?



# רגיסטרים של המחסנית



ss – stack segment ▶

- מצביע על הכתובת הנמוכה ביותר במחסנית

sp – stack pointer ▶

- מצביע על האופסט בתוך סגמנט המחסנית

- בתחילת התוכנית ערכו שווה לגודל המחסנית

▶ בעזרת הצירוף  $SS:sp$  אפשר להגיע לכל כתובת במחסנית



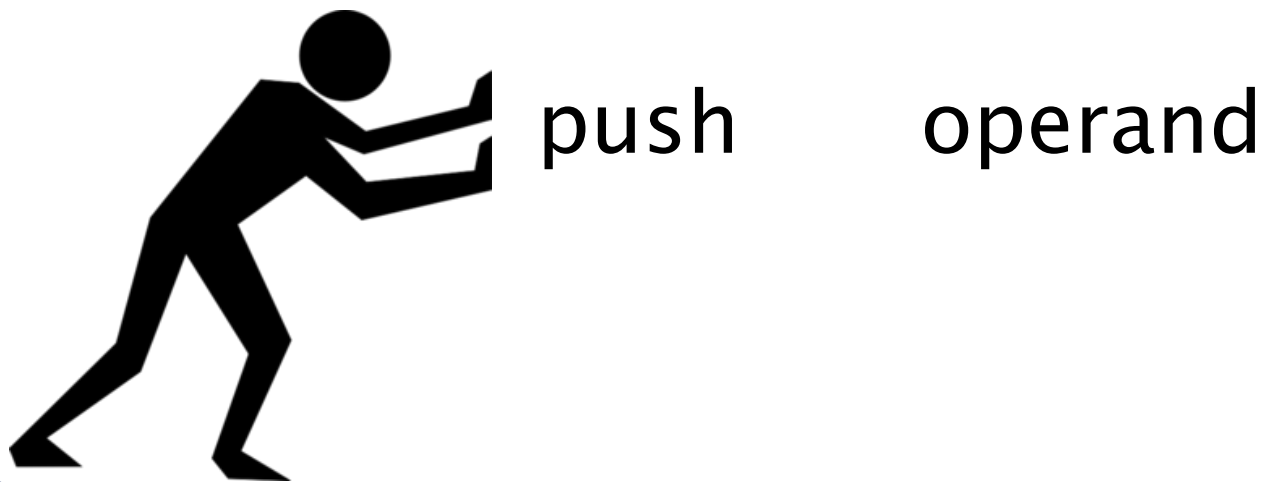
# הכנסה והוצאה של נתונים מהמחסנית

---

- ▶ המחסנית עובדת בשיטת Last In First Out - LIFO
  - הערך שנכנס אחרון הוא הראשון לצאת
- ▶ לפני הכנסת ערך למחסנית ערכו של sp יורד
- ▶ לאחר הוצאת ערך מהמחסנית ערכו של sp עולה
- ▶ פקודת `push` מכניסה ערך למחסנית
- ▶ פקודת `pop` מוציאה ערך מהמחסנית

# פקודת push

- ▶ הכנסה של ערך למחסנית
- ▶ מה יבוצע?
  - ערכו של SP יירד בשתיים:  $sp = sp - 2$ .
  - ערכו של האופרנד יועתק למחסנית, לכתובת  $ss:sp$ .



# פקודת push- המשך

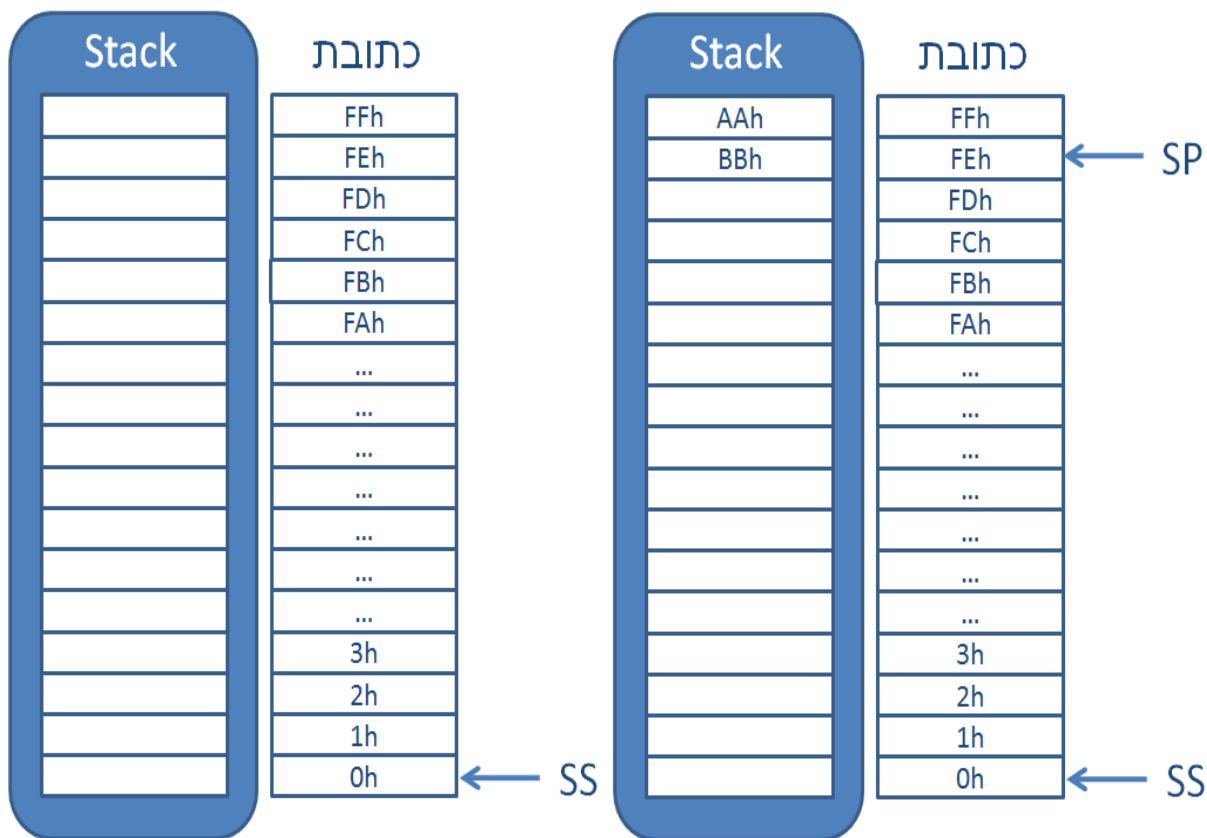
- ▶ אפשר לעשות push רק לערכים בגודל word
  - לדוגמה, הפקודה push al אינה חוקית

```
push 5h  
push ax  
push ds  
push var
```

# פקודת push- המשך

מה יהיה מצב המחסנית אחרי הפקודות הבאות?

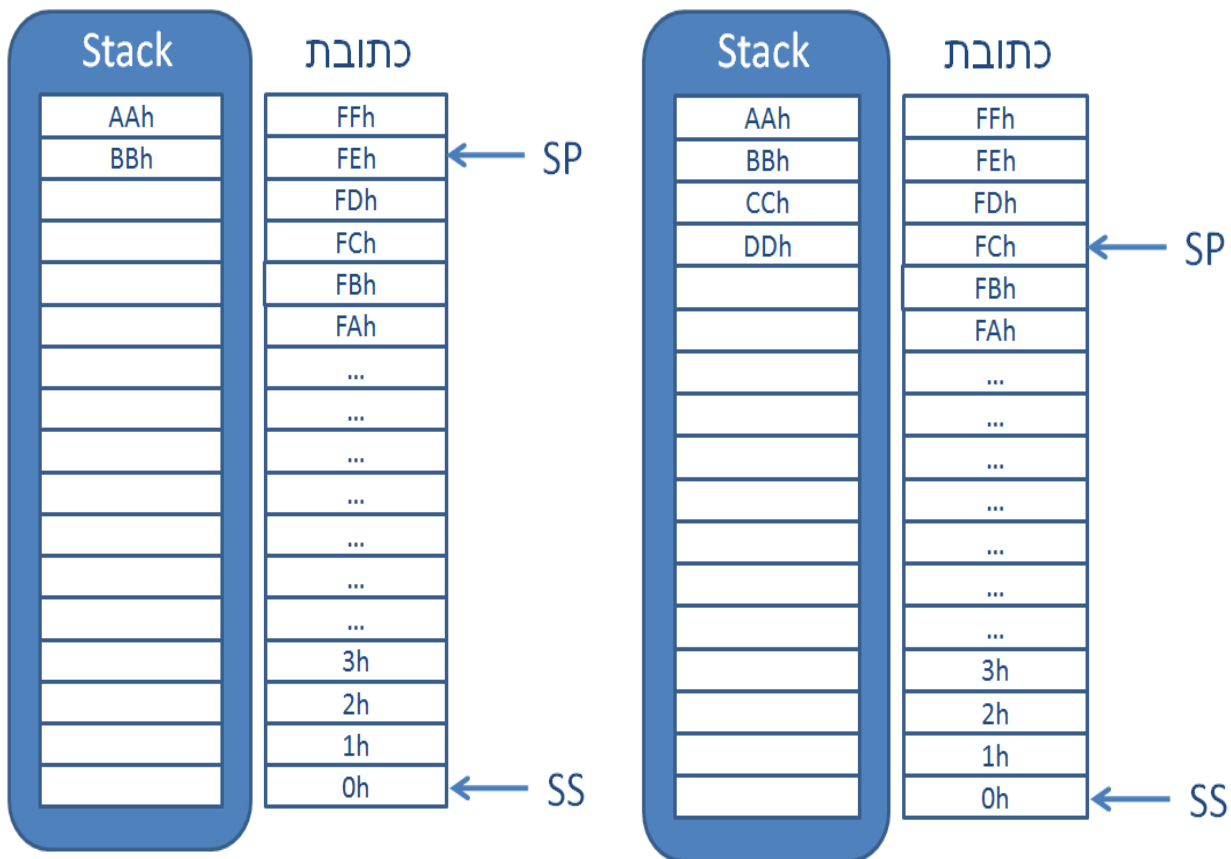
```
mov ax, 0AABBh
push ax
```



# פקודת push - המשך

push

0CCDDh



# פקודת pop



- ▶ הפקודה ההפוכה ל-push
- ▶ הוצאה של מילה מהמחסנית והעתקה שלה לאופרנד
- ▶ מה יבוצע?
  - מילה מראש המחסנית תועתק לאופרנד
  - ערכו של sp יעלה ב-2

pop operand

# פקודת קס- המשך

▶ האופרנד חייב להיות בגודל מילה

- `pop al ;illegal`

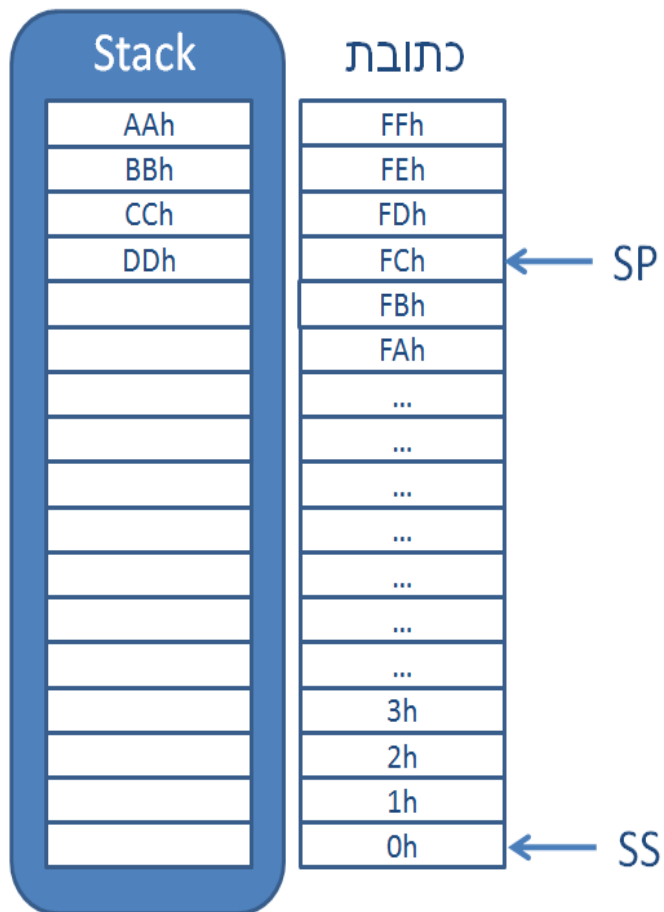
▶ אסור לאופרנד להיות קבוע

- `pop 5h ;illegal`

```
pop ax
pop cs
pop var
```



# פקודת סקס- המשך



▶ נתונה מחסנית. מה יקרה לאחר הפקודה הבאה?

pop bx

ערכו של ax יהיה 0CCDDh, ערכו של sp יעלה בשתיים ויהיה שווה 0FEh

האם פקודת סקס מוחקת את הערכים מהמחסנית?

# מחסנית- תרגילים

- ▶ הגדירו מחסנית בגדלים שונים. 20h, 10h. ראו כיצד משתנה ערכו של sp עם תחילת ההרצה.
- ▶ הכניסו את הערך 1234h לתוך ax. בצעו push של ax. איך השתנה sp?  
הסתכלו על הזיכרון שבמחסנית ומיצאו את הערך שדחפתם.
- ▶ בצעו pop למחסנית לתוך ax. איך השתנה sp? הסתכלו על הזיכרון שבמחסנית- האם הערך 1234h נמחק?
- ▶ בצעו push לערך 5678h. האם עכשיו נמחק הערך 1234h?
- ▶ העתיקו את ax לתוך ax בעזרת המחסנית ללא שימוש בפקודת mov.

- ▶ מה הופך "סתם" קטע קוד לפרוצדורה?
  - קוראים לפרוצדורה באמצעות הפקודה call.
  - אפשר להעביר פרמטרים לפרוצדורה.
  - לפרוצדורה יש מנגנונים להחזרת תוצאות העיבוד.
  - פרוצדורה יכולה ליצור משתנים מקומיים (שמשמשים את הפרוצדורה בלבד) ולהיפטר מהם לפני החזרה לתוכנית הראשית.

# מבנה של פרוצדורה

- ▶ פרוצדורה מוגדרת תמיד:
  - או בתחילת סגמנט הקוד
  - או בסוף סגמנט הקוד
- ▶ להלן שלד של פרוצדורה:

```
proc ProcedureName
    ... ; Code for something that the procedure does
    ret ; Return to the code that called the procedure
endp ProcedureName
```

# “SecretProc” הפרוצדורה

```

IDEAL
MODEL small
Stack 100h
DATASEG
digit      db 10      dup(1)

CODESEG

proc      SecretProc
          xor        al, al
          mov        cx, 10

SecretLoop:
          mov        [bx], al
          inc        bx
          loop       SecretLoop
          ret

endp      SecretProc
    
```

- ▶ מה מבצעת הפרוצדורה הבאה?
- ▶ העתיקו, הריצו וצפו בשינויים בערכו של ip במהלך הריצה
- ▶ תרגיל מחקר: מה עושות פקודות call, ret?

```

start:
          mov        ax, @data
          mov        ds, ax
          mov        bx, offset digit
          call       SecretProc

exit:
          mov        ax, 4C00h
          int        21h

END start
    
```

# פקודת call

```

DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TD
File Edit View Run Breakpoints Data Options Window Help READY
[ ]-CPU 80486 2-[1111]
#baseproc#start: mov ax, 0data          ax 087B c=0
cs:000B B87B08   mov  ax,087B          bx 087B z=0
#baseproc#22:   mov ds, ax             cx 0000 s=0
cs:000E BED8    mov  ds,ax           dx 0000 o=0
#baseproc#24:   mov bx, ds             si 0000 p=0
cs:0010 8CDB    mov  bx,ds           di 0000 a=0
#baseproc#25:   call ZeroMemory       bp 0000 i=1
cs:0012E8EBFF   call #baseproc#zeromemory sp 0100 d=0
#baseproc#exit: mov ax, 4c00h          ds 087B
cs:0015 B8004C   mov  ax,4C00         es 0869
#baseproc#28:   int 21h               ss 0882
cs:0018 CD21    int  21              cs 0879
cs:001A 0000    add  [bx+si],al      ip 0012
cs:001C 0000    add  [bx+si],al
cs:001E 0000    add  [bx+si],al

es:0000 CD 20 7D 9D 00 EA FF FF = }¥ ¨
es:0008 AD DE 32 0B C3 05 6B 07 i 26 |k•
es:0010 14 03 28 06 14 03 92 01 90 (ff)ff
es:0018 01 01 01 00 02 04 FF FF 000 0+
es:0020 FF FF FF FF FF FF FF FF

ss:010052FB
ss:00FE 0000
ss:00FC 05C3
ss:00FA 0A95
ss:00F8 0000

F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
  
```

- ▶ נחקור מה מבצעת פקודת call
- ▶ בתוכנית הקודמת, נשנה את שם הפרוצדורה מ-SecretProc ל-ZeroMemory ונבצע קריאה:
- call ZeroMemory
- ▶ לפני הקריאה ip=12h

# פקודת call-המשך

```

DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TD
File Edit View Run Breakpoints Data Options Window Help
[ ]-CPU 80486
#baseproc#zeromemory: xor al, al
cs:0000 32C0 xor al,al
#baseproc#12: mov cx, 100
cs:0002 B96400 mov cx,0064
#baseproc#zeroloop: mov [bx], al
cs:0005 8807 mov [bx],al
#baseproc#15: inc bx
cs:0007 43 inc bx
#baseproc#16: loop ZeroLoop
cs:0008 E2FB loop #baseproc#zeroloop (0005)
#baseproc#17: ret
cs:000A C3 ret
#baseproc#start: mov ax, @data
cs:000B B87B08 mov ax,087B
#baseproc#22: mov ds, ax
es:0000 CD 20 7D 9D 00 EA FF FF = }¥ Ω
es:0008 AD DE 32 0B C3 05 6B 07 ; |Z|*k•
es:0010 14 03 28 08 14 03 92 01 ¶ (|)ffff
es:0018 01 01 01 00 02 04 FF ff ff ff
es:0020 FF FF FF FF FF FF FF FF
ax 087B c=0
bx 087B z=0
cx 0000 s=0
dx 0000 o=0
si 0000 p=0
di 0000 a=0
bp 0000 i=1
sp 00FE d=0
ds 087B
es 0869
ss 0882
cs 0879
ip 0000
ss:0100 52FB
ss:00FE 0015
ss:00FC 0000
ss:00FA 05C3
ss:00F8 0A95
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

```

- ▶ מיד לאחר הקריאה לפרוצדורה השתנה ערכו של ip
- מדוע כעת  $ip=0000$ ?
- ▶ איזה עוד רגיסטר השתנה?
- איזה ערך נמצא כרגע במקום  $ss:sp$ ?



# פקודת ret

► מצב המעבד לפני ואחרי הרצת פקודת ה-ret:

The image displays two screenshots of the DOSBox 0.74 interface, showing the execution of assembly code and the state of registers before and after the execution of the `ret` instruction.

**Top Screenshot (Before `ret`):**

- Assembly code:
 

```
cs:FFFF 0032  add [bp+sil,dh
cs:0001 C0B9640088  sar byte ptr [bx+di+0064],88
cs:0006 07      pop  es
#baseproc#15:  inc  bx
                inc  bx
                zeroLoop
                loop #baseproc#zeroloop (0005)
```
- Register state:
 

ax	0000	c=0
bx	00DF	z=0
cx	0000	s=0
dx	0000	o=0
si	0000	p=0
di	0000	a=0
bp	0000	i=1
sp	00FE	d=0

**Bottom Screenshot (After `ret`):**

- Assembly code:
 

```
#baseproc#22:  mov  ds,ax
cs:000E 8ED8      mov  ds,ax
#baseproc#24:  mov  bx,ds
cs:0010 8CDB      mov  bx,ds
#baseproc#25:  call ZeroMemory
cs:0012 E8EBFF      call #baseproc#zeromemory
#baseproc#exit: mov  ax,4c00h
cs:0015 B8004C      mov  ax,4000
#baseproc#28:  int  21h
cs:0018 CD21      int  21
cs:001A 0000      add  [bx+sil],al
cs:001C 0000      add  [bx+sil],al
```
- Register state:
 

ax	0000	c=0
bx	00DF	z=0
cx	0000	s=0
dx	0000	o=0
si	0000	p=0
di	0000	a=0
bp	0000	i=1
sp	0100	d=0
ds	087B	
es	0869	
ss	0882	
cs	0879	
ip	0015	

The `ret` instruction is highlighted in blue in both screenshots, indicating its execution. The register state changes significantly, particularly the stack pointer (`sp`) and the instruction pointer (`ip`).

# סיכום- פקודות call, ret

## ▶ פקודת call:

- מורידה את ערכו של  $sp$  בשתיים (יש מקרה שבו היא מורידה את ערכו של  $sp$  בארבע- בהמשך)
- מעתיקה לתוך הכתובת  $ss:sp$ , את הכתובת בזיכרון אליה יש לחזור בסיום ריצת הפרוצדורה
- מעתיקה לתוך  $ip$  את הכתובת של הפקודה הראשונה של הפרוצדורה
- מבצעת  $jump$  אל תחילת הפרוצדורה

## ▶ פקודת ret:

- קוראת מהכתובת  $ss:sp$ , את הכתובת בזיכרון אליה יש לחזור.
- מעלה את ערכו של  $sp$  בשתיים (במקרה זה- מקרים אחרים בהמשך).
- משנה את ה- $ip$  אל הכתובת שנקראה מ- $ss:sp$ , ובכך מחזירה את התוכנית לנקודה בה היא היתה לפני הקריאה לפרוצדורה.

# פרוצדורות near, far



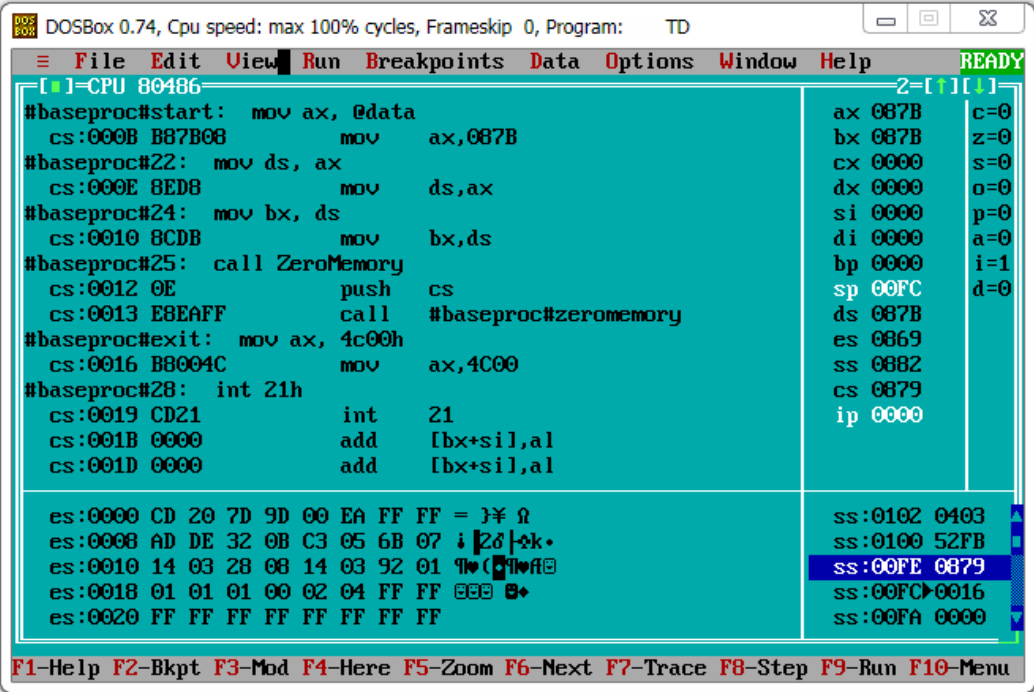
- ▶ פרוצדורה יכולה להיות או באותו סגמנט עם הקוד שקורא לה, או בסגמנט אחר
- ▶ פרוצדורה near - באותו סגמנט
- ▶ פרוצדורה far - בסגמנט אחר
- תוספת ההוראה far בכותרת הפרוצדורה
- למחסנית יידחף גם ה-`ip` וגם הסגמנט של כתובת החזרה
- ערכו של `sp` יירד ב-4 עם ביצוע `call`
- ערכו של `sp` יעלה ב-4 עם ביצוע `ret`
- ▶ סביר שנשתמש רק בפרוצדורות near

# פרוצדורת far

▶ התוכנית הועתקה עם תוספת של מילה אחת בלבד:

```
proc ZeroMemory far
```

▶ אילו שינויים חלו במהלך הריצה?



# שימוש במחסנית לשמירת מצב התוכנית

```

CODESEG
proc   Print10X
      mov     cx, 4      ; 4 'X' per line
PrintXLoop:
      mov     dl, 'X'
      mov     ah, 2h
      int     21h      ; Print 'X'
      loop   PrintXLoop
      ret
endp   Print10X

start:
      mov     ax, @data
      mov     ds, ax
      mov     cx, 3
    
```

▶ מתכנת רצה להדפיס  
למסך שלוש שורות,  
ארבעה Xים בכל שורה

▶ מה הבעיה בתוכנית  
הבאה?

```

Row:
      call    Print10X
      mov     dl, 0ah
      mov     ah, 2h
      int     21h      ; New line
      loop   Row
exit:
      mov     ax, 4c00h
      int     21h
END    start
    
```

## שימוש במחסנית לשמירת מצב התוכנית

```
CODESEG
proc   Print10X
      push   cx
      mov    cx, 4      ; 4 'X' per line
PrintXLoop:
      mov    dl, 'X'
      mov    ah, 2h
      int    21h      ; Print 'X'
      loop  PrintXLoop
      pop    cx
      ret
endp   Print10X

start:
      mov    ax, @data
      mov    ds, ax
      mov    cx, 3
```

נדאג לשמור את CX  
בכניסה לפרוצדורה  
ולשחזר את ערכו  
ביציאה

```
Row:
      call   Print10X
      mov    dl, 0ah
      mov    ah, 2h
      int    21h      ; New line
      loop  Row
exit:  mov    ax, 4c00h
      int    21h
END    start
```

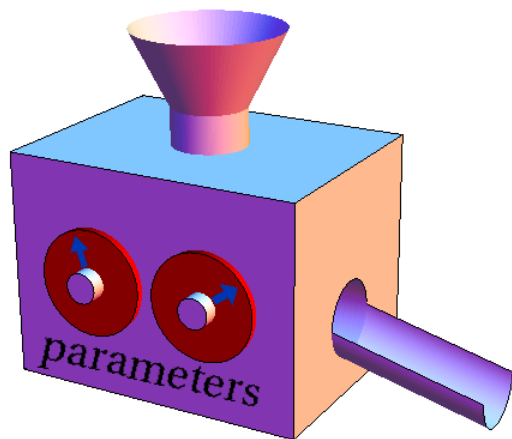
CODESEG

```
proc    ChangeRegistersValues  
        ; ???  
        mov     ax, 1  
        mov     bx, 2  
        mov     cx, 3  
        mov     dx, 4  
        ; ???  
        ret  
endp    ChangeRegistersValues
```

▶ הוסיפו שורות קוד  
לפרוצדורה (במקום ה-  
???) , כך שערכי  
הרגיסטרים ביציאה יהיו  
זהים לערכיהם בכניסה



# העברת פרמטרים לפרוצדורה



▶ לא תמיד נרצה שפרוצדורה תבצע את אותו הדבר

◦ תדפיס למסך 10 Xים

◦ תדפיס למסך 9 Xים

◦ האם צריך לכתוב פרוצדורה חדשה?

• ניתן להעביר את כמות ה-Xים להדפסה

כפרמטר

▶ שיטות להעברת פרמטרים לפרוצדורה:

◦ שימוש ברגיסטרים כלליים

◦ שימוש במשתנים שמוגדרים ב-DATASEG

◦ העברת הפרמטרים על גבי המחסנית

# שימוש ברגיסטרים כללים

```

proc ZeroMemory
  mov    cx, ax
  ; ax holds the number
  ; of bytes that should
  ; become zero
  xor    al, al
ZeroLoop:
  mov    [bx], al
  inc    bx
  loop   ZeroLoop
  ret
endp ZeroMemory

```

נשים ברגיסטרים כללים את כל הפרמטרים שהפרוצדורה מקבלת

פשוט לשימוש

כמות מוגבלת של רגיסטרים

אם הפרוצדורה תשנה את

הרגיסטרים ייתכן שזה ישנה את

פעולת הקוד שקרא לה

דוגמה: נשלח ל-ZeroMemory

את כמות הבתים לאיפוס

כפרמטר ע"י ax

# שימוש במשתנים ב-dataseg

```

proc ZeroMemory
    mov     cx, [NumOfZeroBytes]
    xor     al, al
ZeroLoop:
    mov     bx, [MemoryStart]
    mov     [bx], al
    inc     [MemoryStart]
loop ZeroLoop
ret
endp ZeroMemory
    
```

▶ נגדיר משתנים ב-  
dataseg ונעתיק לתוכם  
את הפרמטרים

- כבר אין בעיה של כמות פרמטרים
- מתכנת הפרוצדורה צריך לתאם עם מתכנת התוכנית שקוראת לה
- הקוד של הפרוצדורה קשה לשיתוף
- מה אם שתי פרוצדורות צריכות להשתמש באותם פרמטרים?

# העברת פרמטרים על המחסנית

▶ נשתמש במחסנית להעברת הפרמטרים

◦ מצריך פקודות נוספות וזיכרון נוסף

◦ אין מגבלה על כמות הפרמטרים

◦ הפרוצדורה לא צריכה להכיר את שמות

המשתנים בתוכנית שראה לה

◦ ניתן לשתף קוד בקלות

◦ השיטה המקובלת להעברת פרמטרים

▶ קיימות שתי שיטות להעברה:

◦ העברה לפי ערך - Pass by value

◦ העברה לפי ייחוס - Pass by reference

# Pass By Value



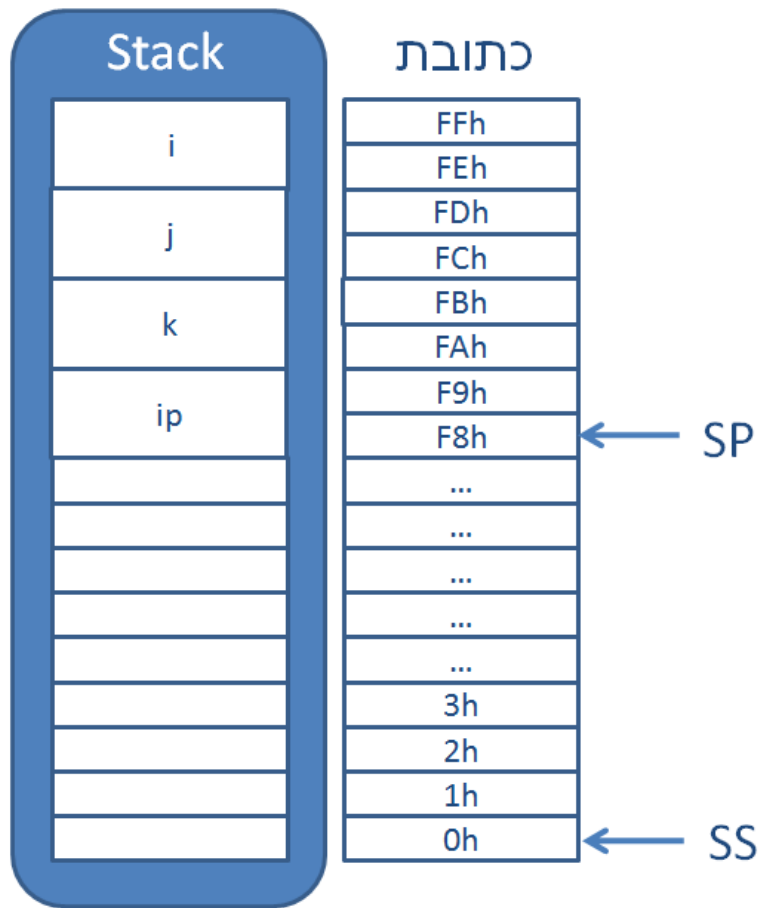
- ▶ בשיטה זו מועבר  
לפרוצדורה ערך הפרמטר  
לדוגמה:

```
push [var]
```

```
push 8
```

- ▶ על המחסנית נוצר העתק  
של הפרמטר  
לפרוצדורה אין גישה  
לפרמטר המקורי
- אינה יכולה לשנות את ערכו,  
רק את ההעתק

# Pass By Value - המשך



דוגמה: הפרוצדורה  
**SimpleProc**  
 $I+J-K$   
 קריאה לפרוצדורה:

```

push [i]
push [j]
push [k]
call SimpleProc
    
```

# Pass By Value - המשך

```

proc   SimpleProc
      pop   ReturnAddress
      pop   ax      ; k
      pop   bx      ; j
      sub   bx, ax   ; j-k
      pop   ax      ; i
      add   ax, bx   ; i+j-k
      push  ReturnAddress
      ret
endp   SimpleProc
    
```

- ▶ בתוך הפרוצדורה נבצע קסק למחסנית כדי להוציא את הפרמטרים
- ▶ ה-קסק הראשון יוציא את קי
  - זו אינה שיטה מקובלת, אנחנו משתמשים בה - זמנית- רק לצורך הבנת פעולת המחסנית
  - צריך לדחוף אותו חזרה למחסנית (מדוע?)
  - בדוגמה זו נשתמש במשתנה בשם ReturnAddress
  - ניתן להשתמש גם ברגיסטר





# Pass By Value - תרגילים

▶ צרו פרוצדורה שמקבלת פרמטר אחד בשיטת pass by value ומדפיסה למסך מספר 'X' לפי הנתון בפרמטר. דגשים: יש לבדוק קודם לכן שערך הפרמטר חיובי! בסיום הריצה יש לשחזר את ערכי הרגיסטרים שנעשה בהם שימוש. עזרה - הדפסת תו למסך:

```
mov    dl, 'X'
mov    ah, 2h
int    21h
```

▶ צרו פרוצדורה שמקבלת שני פרמטרים בשיטת pass by value ומדפיסה למסך 'A' אם הפרמטר הראשון גדול מהשני, 'B' אם הפרמטר השני גדול מהראשון ו-'C' אם הם שווים. בסיום הפרוצדורה יש לשחזר את הערכים המקוריים של הרגיסטרים.

▶ צרו תוכנית שמוגדרים בה ארבעה מספרים כקבועים בתחילת התוכנית, ומשתנים בשם max ו-min. צרו פרוצדורה שמקבלת כפרמטרים pass by value את ארבעת המספרים הקבועים ומכניסה למשתנה max את הערך המקסימלי מביניהם ולמשתנה min את הערך המינימלי מביניהם.

# Pass By Reference



- ▶ מעבירים לפרוצדורה את הכתובת של הפרמטר
  - מיד נראה כיצד
- ▶ על המחסנית לא נוצר העתק של הפרמטר
- ▶ לפרוצדורה יש גישה לכתובת בזיכרון שמכילה את הפרמטר
  - יכולה לשנות את ערכו



# Pass By Reference - המשך

▶ הפרוצדורה SimpleAdd מקבלת פרמטר בשיטת Pass by reference ומעלה את ערכו ב-2

▶ האם אפשר לבצע את אותה פעולה בשיטת Pass by value?

```

proc   SimpleAdd
      pop   ReturnAddress
      pop   bx       ; bx holds the
                    ; offset of "parameter"
      pop   es       ; es holds the
                    ; segment of "parameter"
      add   [byte ptr es:bx], 2
      push  ReturnAddress
      ret
endp   SimpleAdd
    
```

לא.  
 הפרוצדורה  
 היתה משנה  
 את ההעתק,  
 המקור היה  
 נשאר ללא  
 שינוי!

# Pass By Reference - תרגילים

- ▶ צרו פרוצדורה שמקבלת פרמטר בשיטת pass by reference ומעלה את ערכו ב-1.
- ▶ צרו פרוצדורה שמקבלת ארבעה פרמטרים בשיטת pass by reference ומאפסת אותם.
- ▶ צרו פרוצדורה שמקבלת שני פרמטרים בשיטת pass by reference, ומחליפה ביניהם
  - לדוגמה - לפני הפרוצדורה  $var1 = 4, var2 = 5$ . אחרי הפרוצדורה  $var1 = 5, var2 = 4$