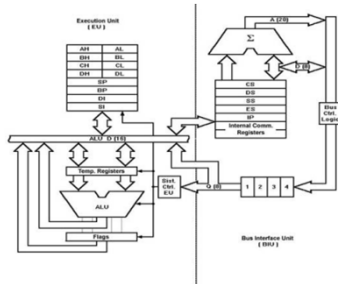


# 8086 CPU :: INSTRUCTION & DATA FLOW

## תקציר

זוהי מצגת נסיונית בנושא אסמבלר 8086, בעקבות תובנות של נסיונות לימוד של מי ש"איננו בעשירון העליון"



## מטרותיה :

- "סיור מודרך" על דוגמה איך משתמשים בטורבו-דבאגר (TD) (בכוונת מכוון בנוסח "ואיך זיל גמור")
- נסיון להדגים ישירות על המכונה תהליך JMP ("לחימום"), ותהליך CALL עם העברת פרמטרים
- לייצר "רב שיח" בין מורים/עוזרי הוראה לגבי השפעה של טכניקה זו או אחרת:
  - על קליטה החומר
  - על המוטיבציה לנצל כלים כדי להפוך לאוטו-דידקטים ("המטרה ארוכת הטווח בהא הידיעה")

```

[1]-CPU 00106
cs:0000 BED8 + mov ds, ax
cs:0000 B10000 + mov bx, offset ol
cs:0000 32E4 + xor ah, ah
cs:0000 B0C000 + mov ax, 0
if loop00 loop0
cs:0000 B007 + mov al, [bx]
cs:000F B801 + test al, 1 ; nz == LSB
cs:0011 B801 + jc next00
cs:0013 1004 + inc ah
if loop00mc next00
cs:0015 45 + inc bx
cs:0016 E2F5 + loop loop0
cs:001B B0C430 + add ah, 30h ; wake it
ds:0000 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
ds:0008 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16
ds:0010 1D 42 45 52 59 20 77 65 F8ERS uc
ds:001B 72 65 20 20 4 56 4E rc EVEN
    
```

JEEZ = Jump on Equal/Zero	01110100	dip	
JLENGE = Jump on Less/Not Greater or Equal	01111100	dip	
JLENGE = Jump on Less or Equal/Not Greater	01111110	dip	(-128 to +127)
JBNAGE = Jump on Below/Not Above or Equal	01110010	dip	

כתב: יגאל שפירא (עוזר הוראת אסמבלר - גבהים) 5 מרץ 15

## מבוא

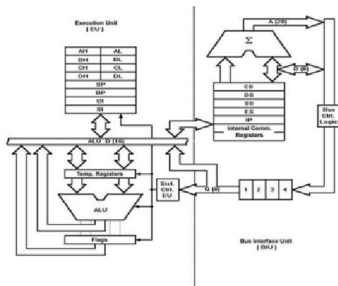
1. רבי עקיבא אמר: "ואידך זיל גמור".  
תרגום לעברית: "והשאר לך ולמד" (בעצמך!)
2. "לא נכשלתי 10000 פעם, אבל  
מצאתי 10000 שיטות שלא עובדות" - (אדיסון  
אחרי שמצא את החומר הנכון לנורה החשמלית)

לפני שבוע, במסגרת מפגש עוזרי הוראה, דיברנו על "התרגלו להאכלה בכפית", כשנשאלתי כיצד אני מנסה להשפיע, ענית: "כמו עם הנכד". הוא שואל משהו, אני עונה "ומה אתה חושב?" במחשבה שניה אחרי המפגש, הכרזתי על הטכניקה הזו כטעות (עוד אחת מיני 10000 בדרך למצוא את שיטת הלימוד המתאימה לי ולהם). הסיבה: אצל הנכד הזאטוט אני בחזקת סמכות עליונה. כשאני שואל – הוא מתאמץ לחשוב בכיתה – מי אני בשבילם? אפילו לא מורה, רק עוזר. לא נותן ציון. אשפיע על העתיד? מי מתעניין שם בעתיד? פלא שהתשובה היא "נו תגיד כבר ודי"

לפני יומיים קיבלתי את התובנות משיעור למתחילים בחאן-אקדמי. (גם ניסיתי אישית – מרשים עד מדהים !!) התובנה שלי: הנה הם מסוגלים ללמוד לבד, ומקבלים בתמורה את מה שחסר בשיעורי האסמבלר שלנו: סיפוק מידי ועכשיו. (גם אנחנו המבוגרים אוהבים לצייר. וכאן גם אפשר לעשות PAUSE באמצע שג'סיקה מדברת, לשחק בציור בשביל הכף, ולהמשיך שוב..). במחשבה שניה: ג'סיקה לא זרקה אותם למים ונתנה להם לחפש לבד (\*). היא כן מתחילה בהאכלה בכפית קטנה, מראה להם שזה טעים, ומראה שבמרחק 10 סמ' למטה יש עוד "GOODIES" – מעודדת לנסות.

ולכן אני מרים "בלון ניסוי": הדגמה "הכי קרוב להדגמה חיה", שניתן גם ללמוד עצמית

ברור שהנ"ל רחוק ממלוטש (וגם אם ילוטש – אין לי וודאות ששיג את המטרות הנכספות). אבל זה יכול לשפר את הרב-שיח לכן כל תגובה (או התנסות ותגובה) של מורה/עוזר תבורך, ואם מישהו טרם העביר את השיעור וירצה לנסות על תלמידים – יבורך כפליים

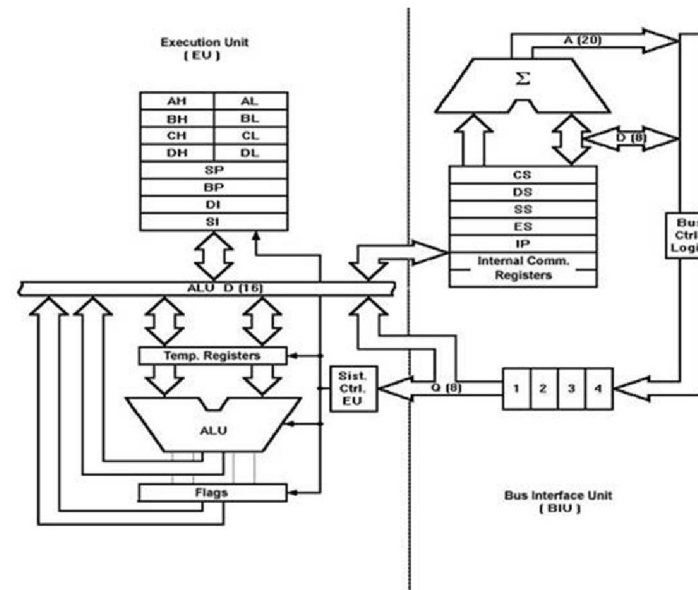
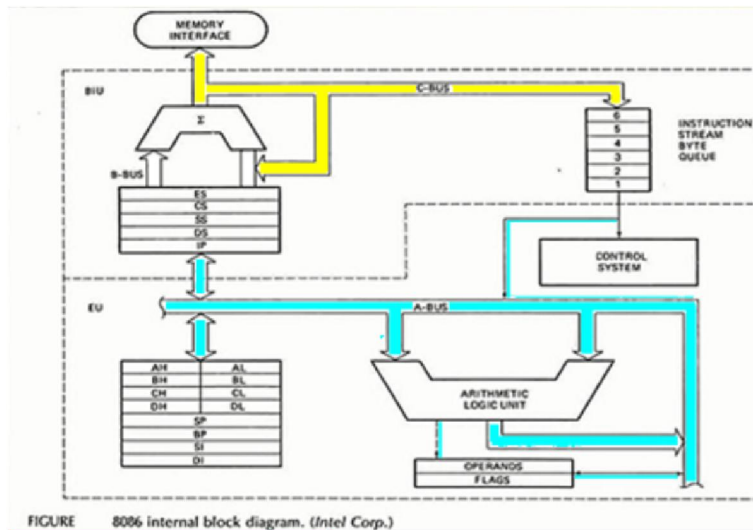


- (\* סוגטה מיטרה כן ניסה את שיטת "זרוק למים" – וזה הצליח (לפחות בתרבות ההודית) מצרף כאן 3 לינקים מתוך שלל מצגות שלו (הראשונה לטעמי היא החשובה יותר בעיניי תובנות של "מה אני רוצים להשיג")
1. [https://www.ted.com/talks/sugata\\_mitra\\_build\\_a\\_school\\_in\\_the\\_cloud?language=he](https://www.ted.com/talks/sugata_mitra_build_a_school_in_the_cloud?language=he)
  - 2.
  - 3.

# 8086 CPU :: INSTRUCTION & DATA FLOW

להלן שתי דיאגרמות דומות שמנסות לתאר את החיבורים בתוך אותו המעבד

**ומדוע זה צריך לעניין ?** ( את התוכניתנים ואנשי הסייבר ... )



**התשובה : בערך מאותה הסיבה שנהג צריך להבין מה קורה כאשר :**

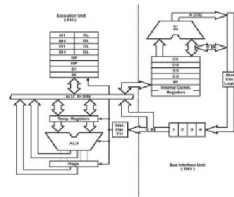
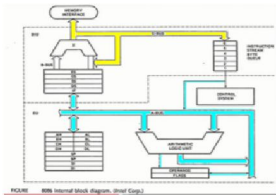
- \* בולמים פתאומית
- \* נדלקת נורת אזהרה
- \* לא מבינים מה קורה במצב גיר ידני ("קלאסי", או טיפטרונקס)
- \* נשארים עם בלם יד משוך

**ההסבר לתשובה: בחלקם יכולים לשפר ביצועים, בכולם עלולים להזיק למכונת (ולנוסעים)**

# 8086 CPU :: INSTRUCTION & DATA FLOW

בהמשך מצגת זו : תיאור עם אנימציות של פעילויות שונות

**לכן : הפעל במוד מצגת !! (אחרת סדר הפעולות לא נראה כלל)**



במקרה של מצגת PPS (שרצה ללא מגע יד אדם):

**תוכל לעצור ולגלגל אחורנית ע"י עכבר**  
(כדי להתעכב או לחזור על קטע)

# 8086 CPU :: INSTRUCTION & DATA FLOW

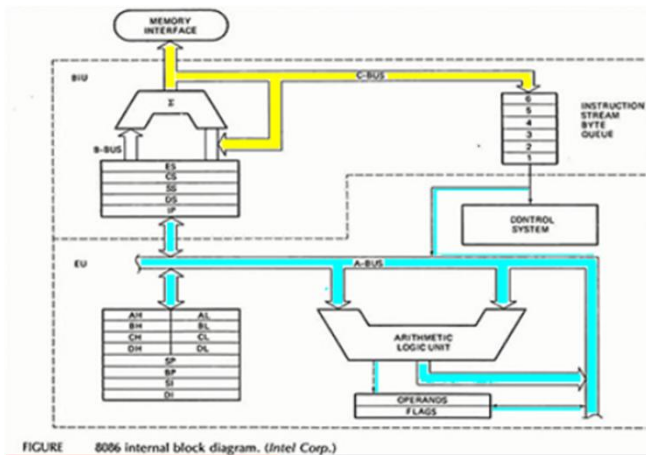
## מטרות מרכזיות של הצגת זרימת הפקודות והנתונים

- "להכניס טוב לראש" שמחשב לא חושב (רק פותח וסוגר ברזים לפי תכנית - כמו מכונת כביסה !)

## "להרגיש" מגבלות כגון :

- מדוע אין יכולת ישירה לעשות את כל הפעולות עם כל הרגיסטרים
- מדוע יש רק כפל של 8 ביט ב 8, של 16 ב 16 - אבל לא שניהם

- להבין דרך המגבלות הנ"ל את הודעות השגיאה של האסמבלר (TASM)  
( כי כשאני רואה את המכונה לנגד עיני ,  
הבנת הודעת השגיאה מקצרת את "זמן תיסכול" !! )



# 8086 CPU :: INSTRUCTION & DATA FLOW

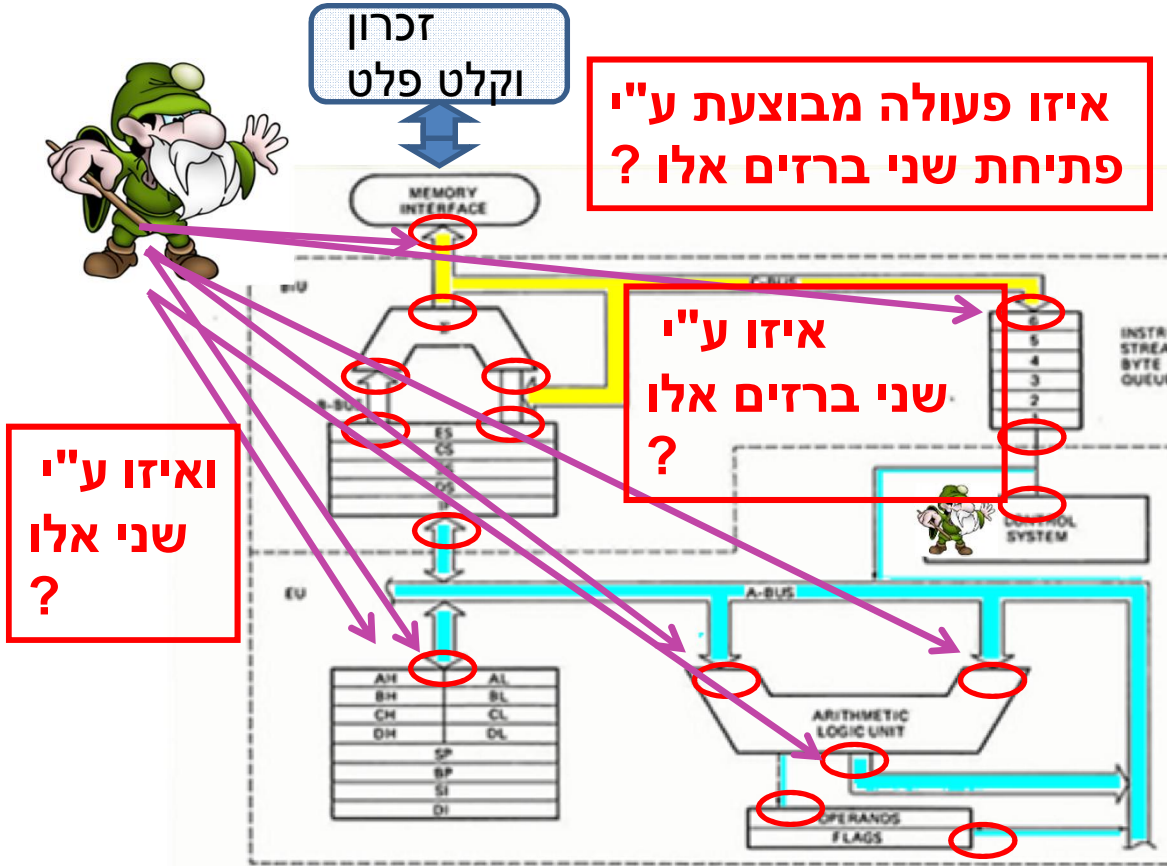


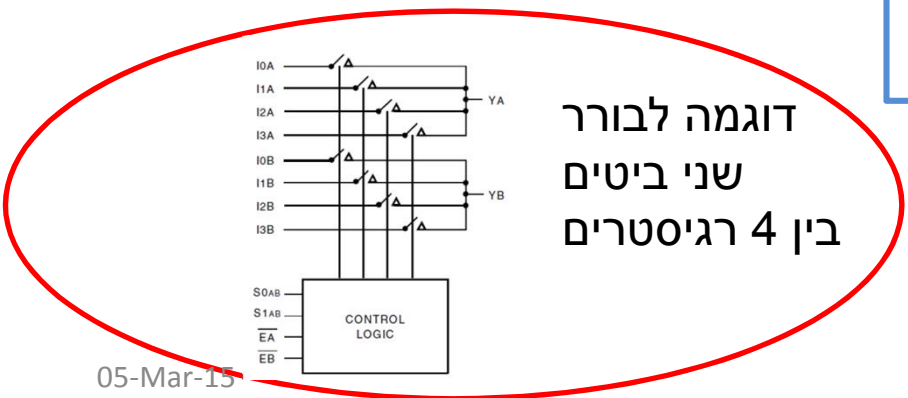
FIGURE 8086 internal block diagram. (Intel Corp.)

- **בפקודת-מכונה יש סדרת פעולות :**
  - הבאת הפקודה מהזכרון
  - פענוח
  - הצבת כתובות, [הבאת עוד בתים]
  - הבאת משתנים, שליחה ל ALU,
  - שמירת תוצאות [רגיסטר, זכרון]
  - קביעת מקום הפקודה הבאה

- **הסדרה הנ"ל היא למעשה:**
  - פתיחה וסגירה של "ברזים",
  - המידע רץ ב "אוטוסטרדות"
  - יש מרכזיה קטנה ליד כל אלמנט חומרה
  - יש "גמד קטן" שמבצע את הסדרה של פתיחת וסגירת השסתומים לפי סדר

- **ה"גמד" בעצמו הוא:**
  - "תת-מיקרו-פרוססור"
  - יש לו הוראות נוסח "ספר - בישול" (צרוב ללא יכולת שינוי),
  - שמושפע ע"י החומרה הבאה:
    - רגיסטר הפקודה
    - רגיסטר דגלים
    - קוי פסיקה מחומרות נוספות

זה מקום מושבו (יחידת הבקרה)



דוגמה לבורר שני ביטים בין 4 רגיסטרים

- (אנחנו המשתמשים טוענים קוד, שרץ במכונה)
- "ספר הבישול" של הבקר נקרא מיקרו-קוד, הוא שולט ישירות על כל המתגים שבמעבד

## 8086 JMP COMMAND

### מטרות מרכזיות של הצגת תהליך JMP

- "להכניס טוב לראש" שמחשב לא חושב ולא מנחש (אלא עושה **רק** את הפקודה הכתובה)

### "להרגיש" מגבלות והשלכות כגון:

- מגבלות טווח קפיצה
- השלכות אפשריות למקרי "התחכמות" של שינוי קוד תוך כדי ריצה (לא רק תחרויות ווירוסים, יש למשל נישה של אפליקציות של אבטחה מסיבות מסחריות – האמינות התפעולית מאד חשובה שם)

## 8086 JMP ZERO COMMAND - 2

### נקח למשל את הפקודה JZ (קפוץ לכתובת רק אם דגל Z מורם)

```

CPU 80486
cs:0003 8ED8      ♦ mov ds, ax
cs:0005 BB0000    ♦ mov bx , offset v1
cs:0008 32E4      ♦ xor ah,ah
cs:000A B90800    ♦ mov cx,8
#if loop0#loop8
cs:000D 8A07      ♦ mov AL , [bx]
cs:000F A801      ♦ test aL,1 ; nz == LSB
cs:0011 7402      ♦ jz next8
cs:0013 F404      ♦ inc ah
#if loop0#next8
cs:0015 43        ♦ inc bx
cs:0016 E2F5      ♦ loop loop8
cs:0018 80C430    ♦ add AH, 30h ; make it

```

בתכנית הזו לדוגמה - JZ בכתובת 11h .  
 הפקודה היא " JZ 02 " , לאן? : לכתובת 15h

#### מדוע:

רגיסטר כתובת הפקודה IP סיים לקרוא את הפקודה בשורה 11 ו-12 ,

IP יקדם עצמו אוטומטית ל 13h - לפקודה העוקבת. אבל אם Z=1 , במקום לבצע אותה ,

הוא צריך **לשנות את IP** ל- 15h !

ומה אומרת הפקודה למכונה (בשפת מכונה - מתורגם להפעלת מתגים בסיליקון ) ?

" אם Z=1 , אז תוסיף ל IP 2 לפני שתקרא את הפקודה הבאה "

JE/JZ = Jump on Equal/Zero	01110100	disp
JL/JNGE = Jump on Less/Not Greater or Equal	01111100	disp
JLE/JNG = Jump on Less or Equal/Not Greater	01111110	disp
JB/JNAE = Jump on Below/Not Above or Equal	01110010	disp

(-128 to +127)

שהשני מכיל את disp (displacement) שהוא בתחום -128 to +127

ומה כתוב בספרות על מבנה פקודת JZ ?

- שהיא בת שני בתים ,
- שהראשון הוא 74h



## 8086 JMP COMMAND - 3

"בונוס" - לא חיוני : היכן מתבצע הסיכום ?

- (לא מאד חשוב למתכנת – לכל היותר הוא רק צריך לדעת אורך פקודה ומספר שעונים לביצוע)
- (כן חשוב למתכנן החומרה – הוא מתחרה על נתח שוק, מנסה לעשות זול – אמין – זריז - גמיש)

ב 8086 ? לא יודע, אבל מתחשק לי לנחש:

- אחרי שהבאנו את הבית הראשון (74h), ההסט **DISP** מחכה בכניסה למפענח הכתובות, ו- IP כבר הועלה ל"כתובת המחדל" (של הפקודה הבאה אם Jump NOT taken)
- אם מבוצע סיכום, יש שתי אפשרויות

**סיכום במסכם הכתובות**

• או שמועבר מכניסת מפענח הפקודה

• או שהוכנס "ישירות מהצד" בצד ימין (שם יש פס נתונים ברוחב 8 ביט) ?

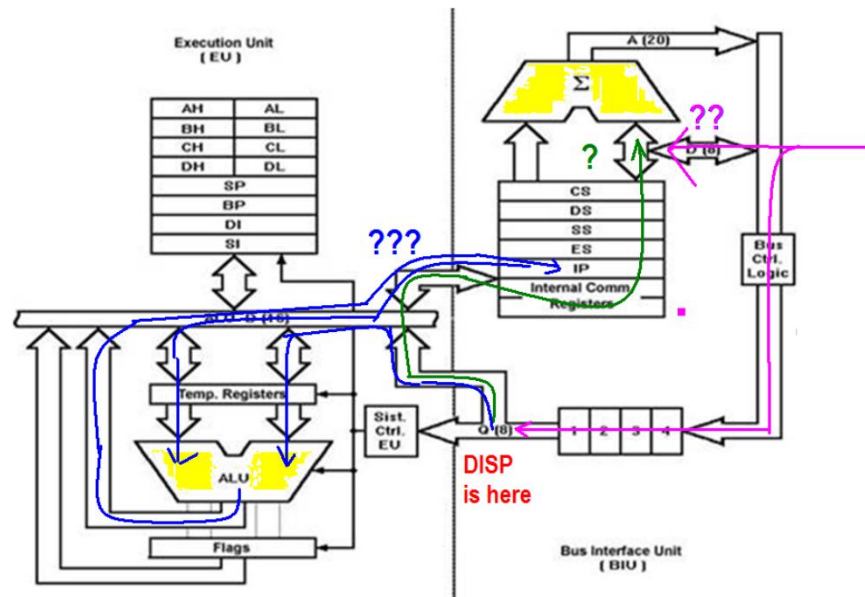
• או סיכום ב ALU ??

מנחש שלא

(כי זה לוקח יותר שעונים:

- להעביר IP של 16 ביט
- ולפחות 17 בדרך חזרה)

(אבל אלו שיקולים של מתכנן השבב)



## 8086 JMP COMMAND

- מגבלות והשלכות :

- מגבלות טווח קפיצה
- השלכות אפשריות למקרי "התחכמות" של שינוי קוד תוך כדי ריצה (לא רק תחרויות ווירוסים, יש למשל נישה של אפליקציות של אבטחה מסיבות מסחריות – האמינות התפעולית מאד חשובה שם)

# 8086 CALL PROCESS - 1

## מטרות מרכזיות של הצגת תהליך CALL

- אימון בשימוש ב TD – "לראות ל CPU את הלבן בעיניים"
- הבנת התהליך - לצורך "צמצום זמן תיסכול":
  - מניעת "טעויות מתכנת"
  - הכרת מנגנון ה CPU (החומרה שמאחורי):
    - משפר הבנה של הודעות שגיאה,
    - מאפשר יישום טכניקות לבדיקה טובה של הקוד בזמן הפיתוח:
    - הבנת הקוד עוזרת לתכנון סט בדיקות יעיל
    - דיבוג של קוד טרי קל יותר לפני שנשכח מה כתבנו ולמה
- להוביל להבנת מגבלות והשלכות כגון:
  - מגבלות טווח קפיצה
  - "זליגת זכרון", התברברות, ונזקיהן
  - חולשות (אבטחה)

## 8086 CALL PROCESS - 2

### הצגת דוגמה פרטנית צעד צעד של פניה לפונקציה, עם שליחה והחזרת מידע דרך המחסנית

- הדגמה עם תכנית פשוטה, והרצה צעד צעד בעזרת TD (התכנית נטו מתחילה בכתובת IP = 07, SP התחלתי בכתובת 28h (DS==SS))

תובנה צדדית – "לקחי תפעול": שינוי stack segment ועצירה תוך כדי תצוגת דה-באג משבשת קלות.

העקיפה: החל את התכנית עם 3 צעדים ראשונים בסדר הבא:

```
mov ax, @data
```

```
mov SS, ax ; ss 1st
```

```
mov DS, ax ;You completed this one after pressing F7 3 times ,
```

```
; Now you are ready to start display CPU pane {by <alt-V> , <C> } {or by mouse}
```

```
; 1st "real" code starts here (IP=7 , The 1st 3 op-codes are "overhead" )
```

```
; ===== B. CODE HERE ===== (ans = max_byte (v1,V2] "התכנית הראשית: בשפה גבוהה")
; sum v1-3 ByVal:
push [ANS] ; -- MAin must read result by POP ()
push [word v1] ; --pushed 1 extra byte (ignored in sub) - Stack release inside the sub
push [word v2] ; ----pushed 1 extra byte (ignored in sub) - Stack release inside the sub
call Max_Byte ; returns unsigned bigger byte
pop [ANS] ; two bytes , msByte ignored
; =====
```

כדי ללמוד מזה לנצל את TD למטרתו (כדה-באגר),  
מומלץ לצפות קודם בסרטון שמראה את ההפעלה ברצף, ואז לעבור דרך השקפים  
הקובץ הוא : [Call\\_example\\_View\\_by\\_TD\\_session.avi](#)

## 8086 CALL PROCESS - 3

mov ax, @data (עצה: אם תשנה את SS בתחילת תכנית – עשה זאת בסדר שמופיע פה) תחילת התכנית:

mov SS, ax ; ss 1<sup>st</sup>

mov DS, ax ; You completed this one after pressing F7 3 times ,  
; Now you are ready to start display CPU pane {by <alt-V> , <C> } {or by mouse}

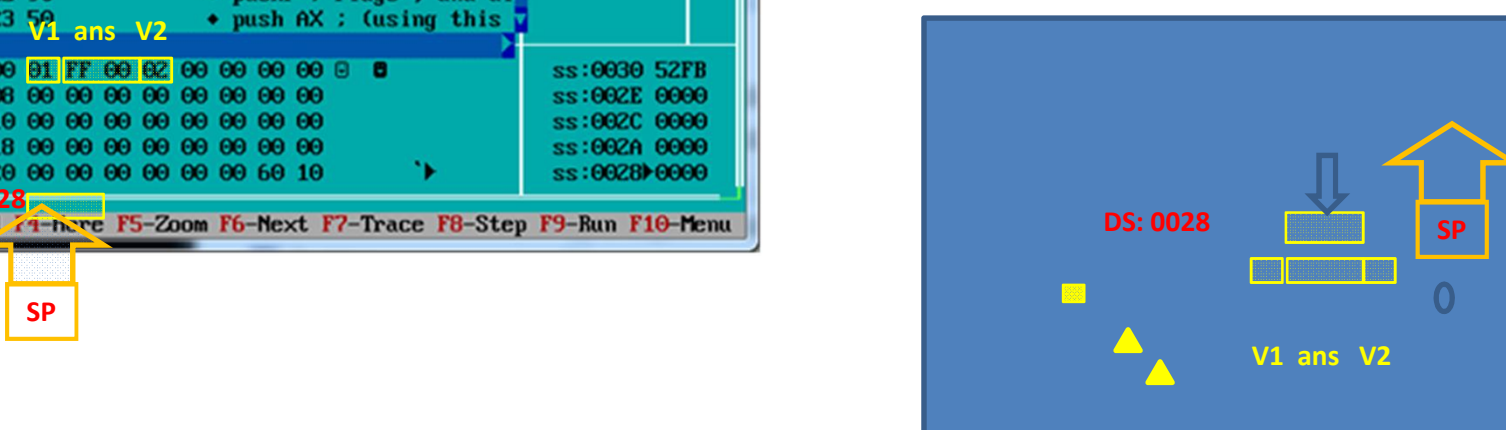
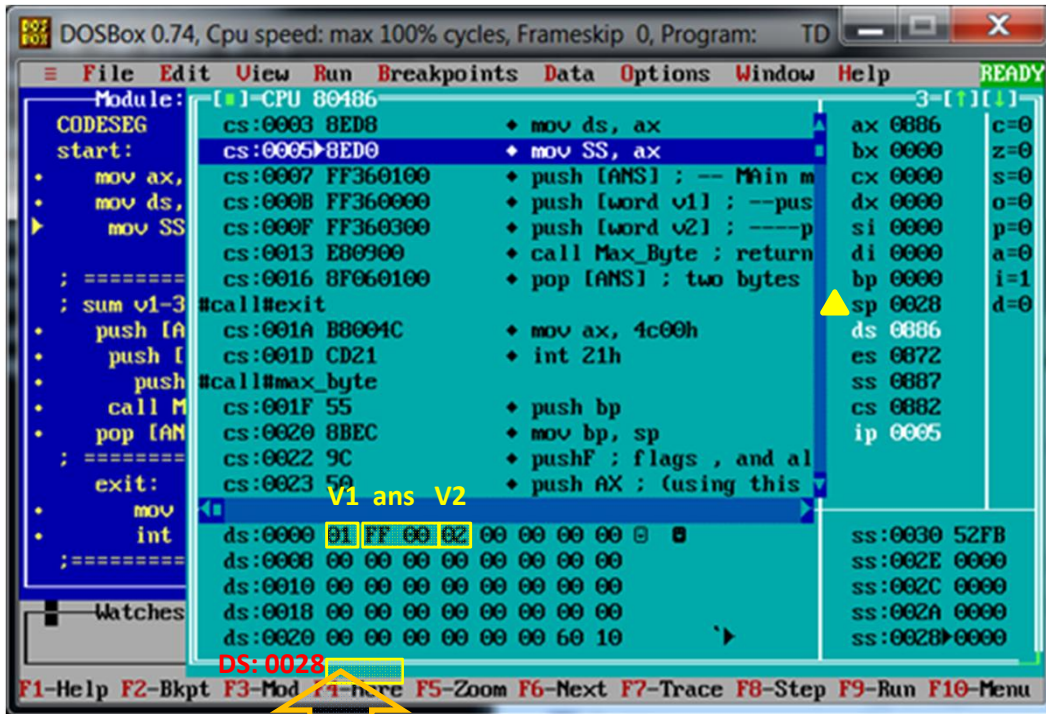
```
; ===== B. CODE HERE ===== (ans = max_byte (v1,v2) "התכנית הראשית: בשפה גבוהה")
; sum v1-3 ByVal:
  push [ANS] ; -- MAin must read result by POP ()
  push [word v1] ; --pushed 1 extra byte (ignored in sub) - Stack release inside the sub
  push [word v2] ; ----pushed 1 extra byte (ignored in sub) - Stack release inside the sub
  call Max_Byte ; returns unsigned bigger byte
  pop [ANS] ; two bytes , msByte ignored
; =====
```

```
; ===== MAX_BYTE ===== הפונקציה:
proc Max_Byte ; UNSIGNED (ByVal A db ,ByVal B db ,ByVal C db )
  push bp
  mov bp, sp
  pushF ; flags , and all registers that will be changed are pushed here
  push AX ; (using this reg , RESTORE LATER)
  ;===== Procedure "MEAT": sum 3 params from stack , answer must be pooped by caller
  MOV AL , [byte BP+ 4] ; "pre-pushed" C (v2)
  CMP AL, [byte BP+6] ; (AX-B)== (v2-v1)
  JNB C_gt_B ; C not below leave as is
  MOV AL, [byte BP+6] ; (AX=B)== (v1)
  C_gt_B:
  mov [byte BP+8] , AL
  ;===== end Procedure "MEAT": =====
  pop AX
  popF
  pop bp
  ret 4 ; leaves ans to be popped by CALLER!!
endp Max_Byte
```

# 8086 CALL PROCESS - 4

## תחילת התכנית:

- הפקודה המוארת היא זו שטרם התבצעה
- התוצאות הן של הפקודה הקודמת
- נתרכז ב DATA שהוכנסה (מסומנת בצהוב)
- SP מוצב בכתובת 28h
- (נתרגל שהמידע בכתובת 27h במחסנית היא ה"שאריות" של תכנית ה TD – היא "תטייל" עם תזוזת ה STACK שלנו)

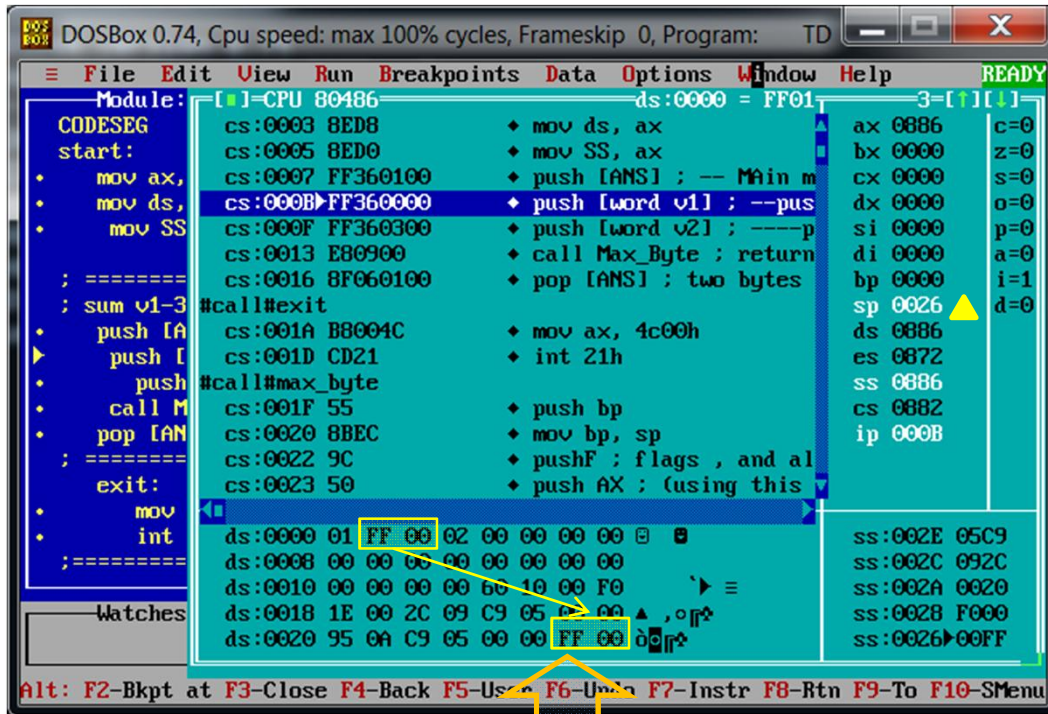


# 8086 CALL PROCESS - 5

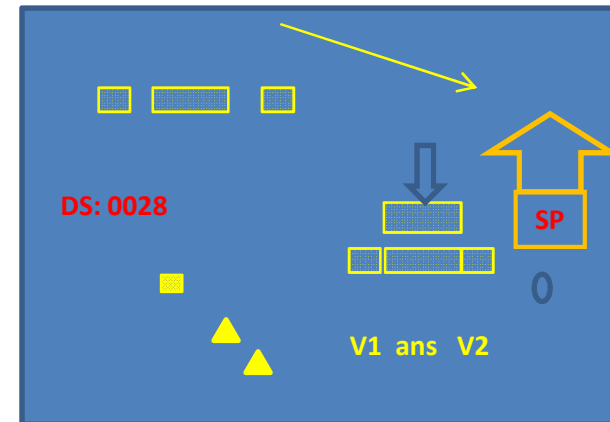
אחרי דחיפת המשתנה הראשון למחסנית:

החץ מראה מי הועתק לאן (ע"י PUSH)  
 SP הוזז למטה בשניים

(אנחנו רואים "הופעת" 18 בתים במחסנית מתחת לכתובת אליה דחפנו מילה - זה מידע לא מעניין של TD - בתנאי שלא ידרוס את הנתונים שלנו. אנו סופרים 17 בתים בשימוש (TD



SP

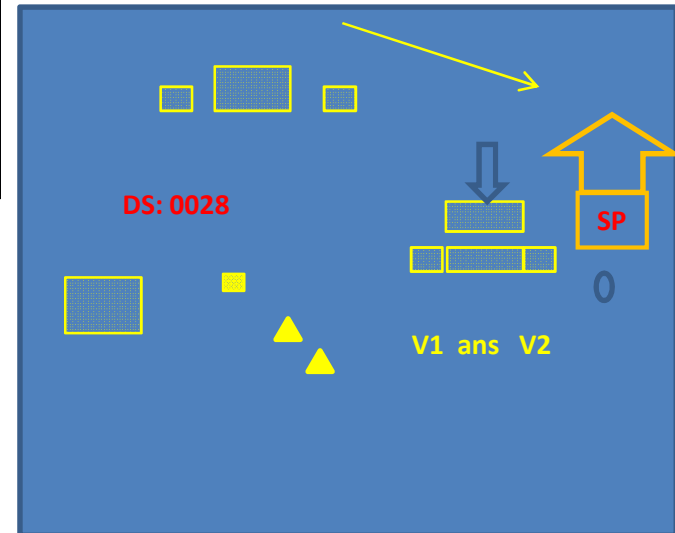
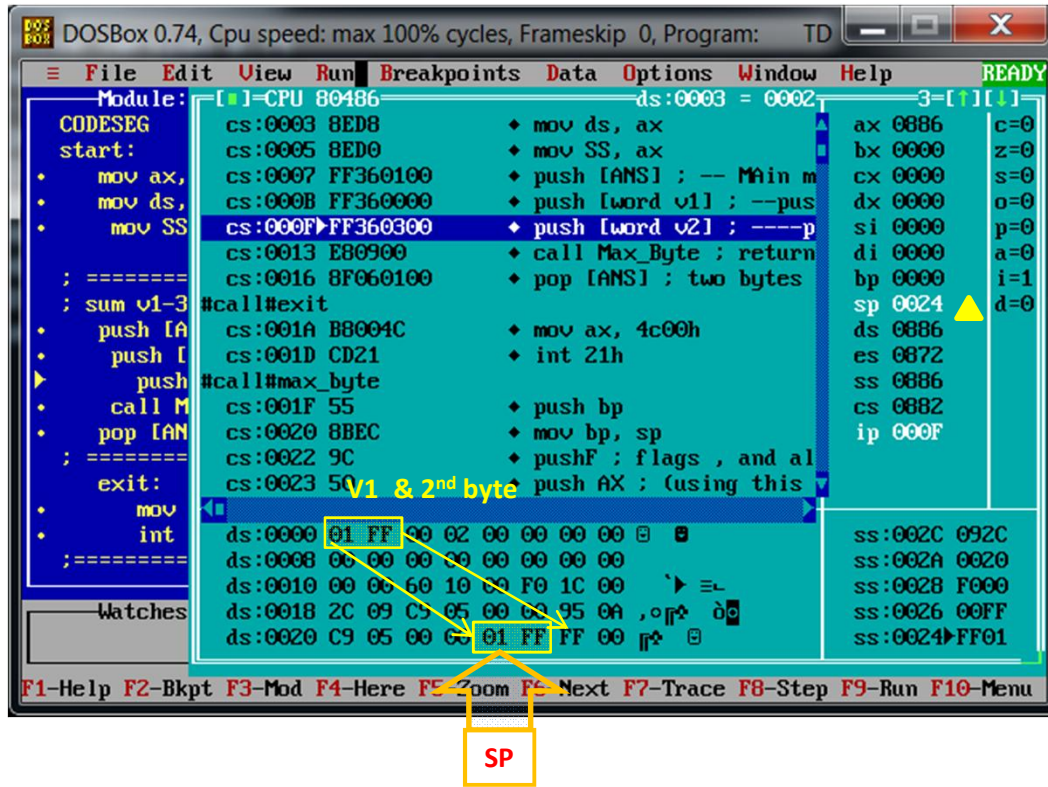


# 8086 CALL PROCESS - 6

אחרי דחיפת המשתנה השני למחסנית:

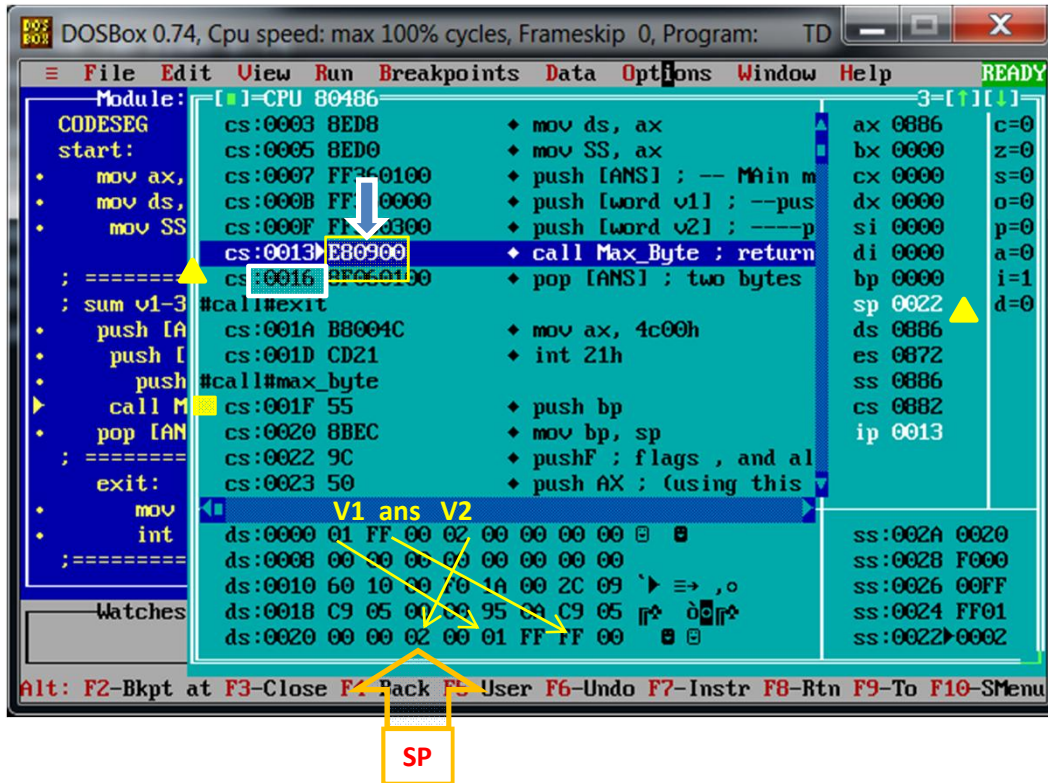
- החץ מראה מי הועתק לאן (ע"י PUSH)
- SP הוזז למטה בשניים

(אנחנו רואים "הופעת" 18 בתים במחסנית מתחת לכתובת אליה דחפנו מילה - זה מידע לא מעניין של TD - בתנאי שלא ידרוס את הנתונים שלנו. אנו סופרים 17 בתים בשימוש (TD





# 8086 CALL PROCESS - 7

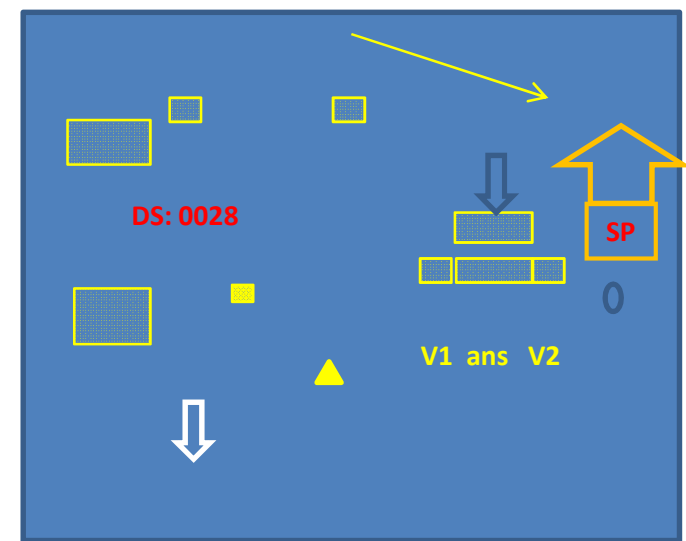


אחרי דחיפת המשתנה השלישי למחסנית: (ורגע לפני ביצוע CALL)

החץ מראה מי הועתק לאן (ע"י PUSH 3 המילים הועברו, אנחנו טרם קפיצה) נשים לב לכתובת שתהיה לאחר ביצוע CALL:

IP של הפקודה העוקבת היא 16h IP הנדרש כדי לקפוץ לפונקציה הוא 1Fh כדי לקפוץ - יש להוסיף 9 ל IP (זה מה ש"אומר" ה BYTE השני של CALL, שנמצא בכתובת 14h)

הכתובת העוקבת היא זו אליה נצטרך לקפוץ חזרה מהפונקציה. היא תידחף עכשיו למחסנית תוך כדי הקפיצה



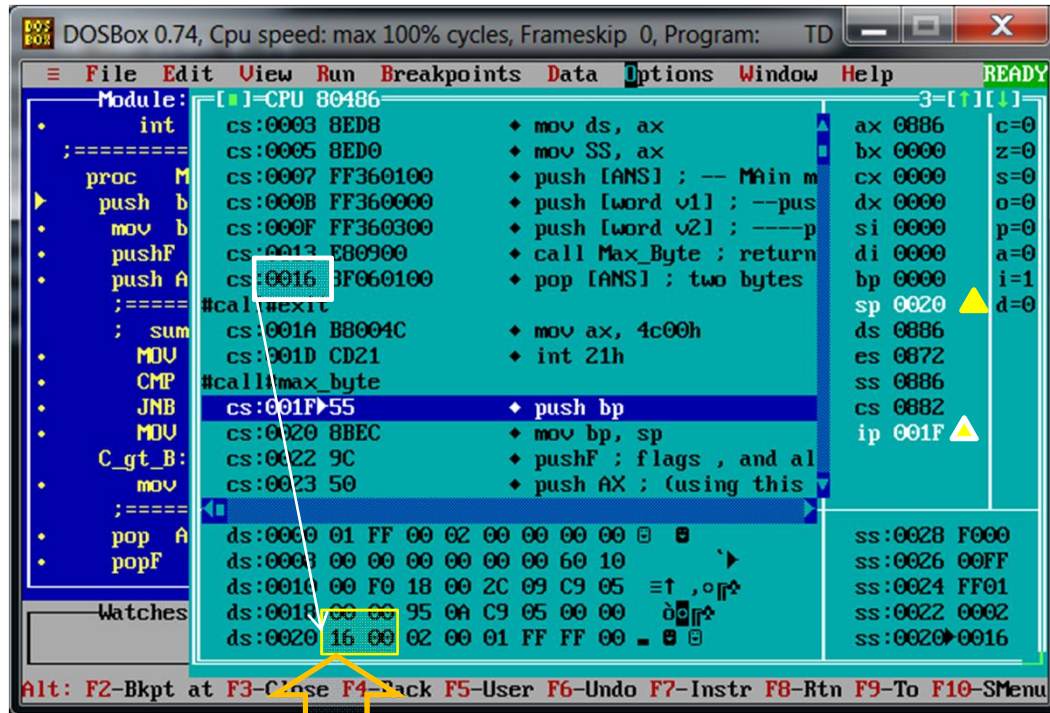
# 8086 CALL PROCESS - 8

אחרי הקריאה לפונקציה (קפיצה + דחיפה) :

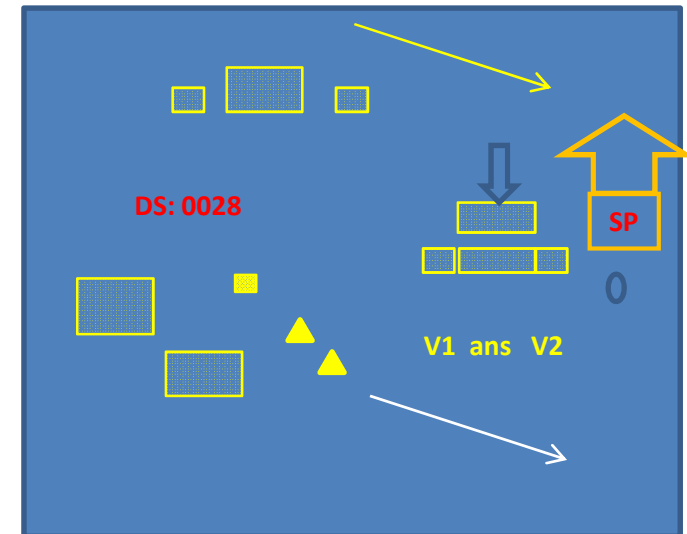
- החץ מראה מי הועתק לאן ( PUSH מובנה ב-CALL )
- SP הוזז למטה בשניים ודחף את כתובת החזרה
- בתוך הפונקציה נרצה להצביע על המשתנים

שנדחפו:

- (אפשר היה ע"י POP , אבל כתובת החזרה בדרך . והיכן נאחסן אם יש לא 2 אלא 200?)
- אז נקצה את ה Base Pointer ( BP ) , ונעניק לו את כתובת ה SP הנוכחי.
- אבל נרצה להיות נחמדים לתכנית שפנתה אלינו לעזרה, ולא להשמיד לה את תכולת הרגיסטרים , לכן נשמור אותם במחסנית לפני שנשנה את ערכם.



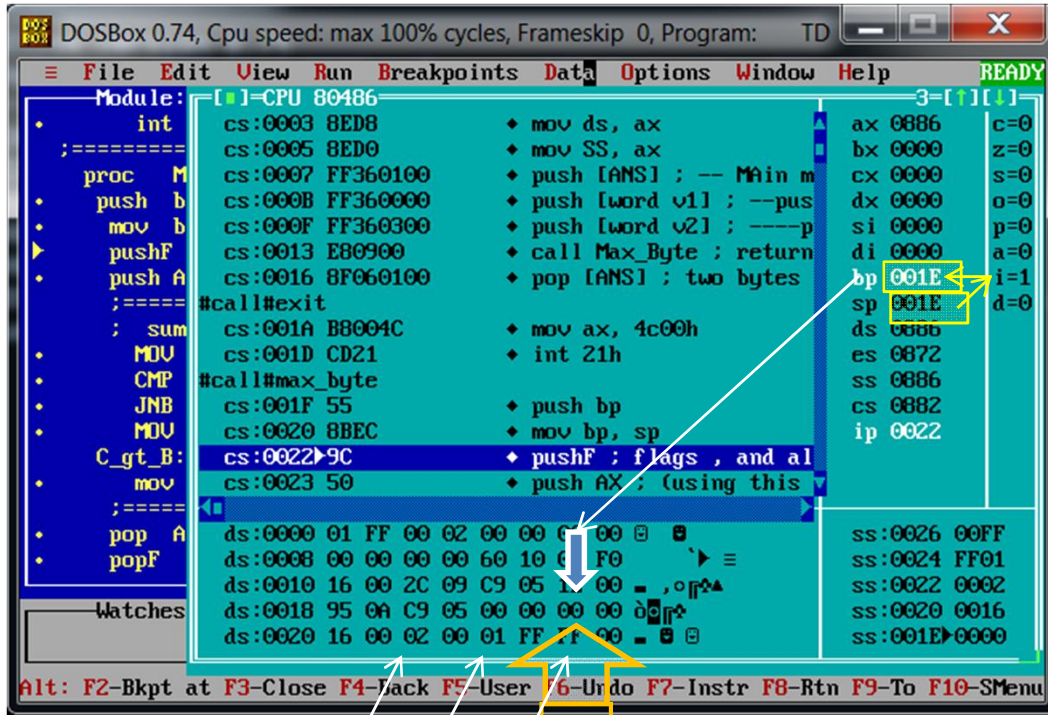
SP





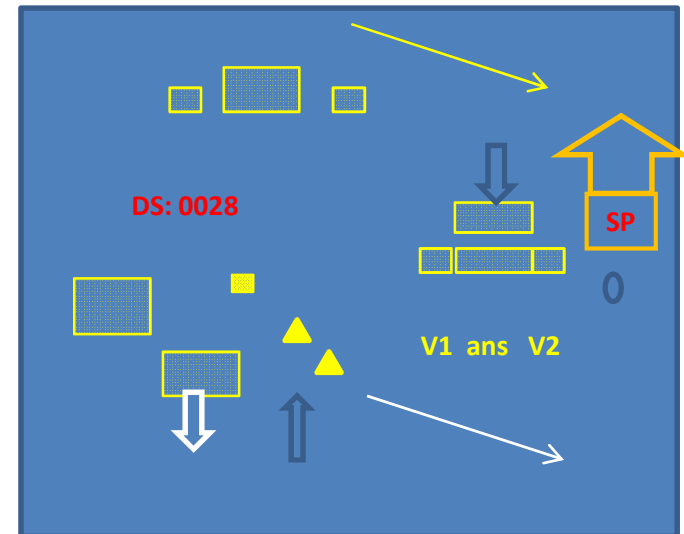
# 8086 CALL PROCESS - 10

אחרי העתקת ל BP ל SP :



[BP+4] [BP+6] [BP+8]  
 == V2 == V1 == ans

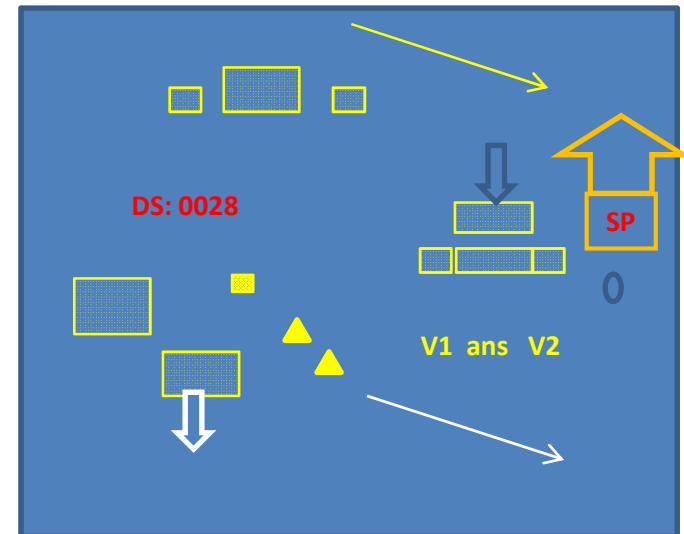
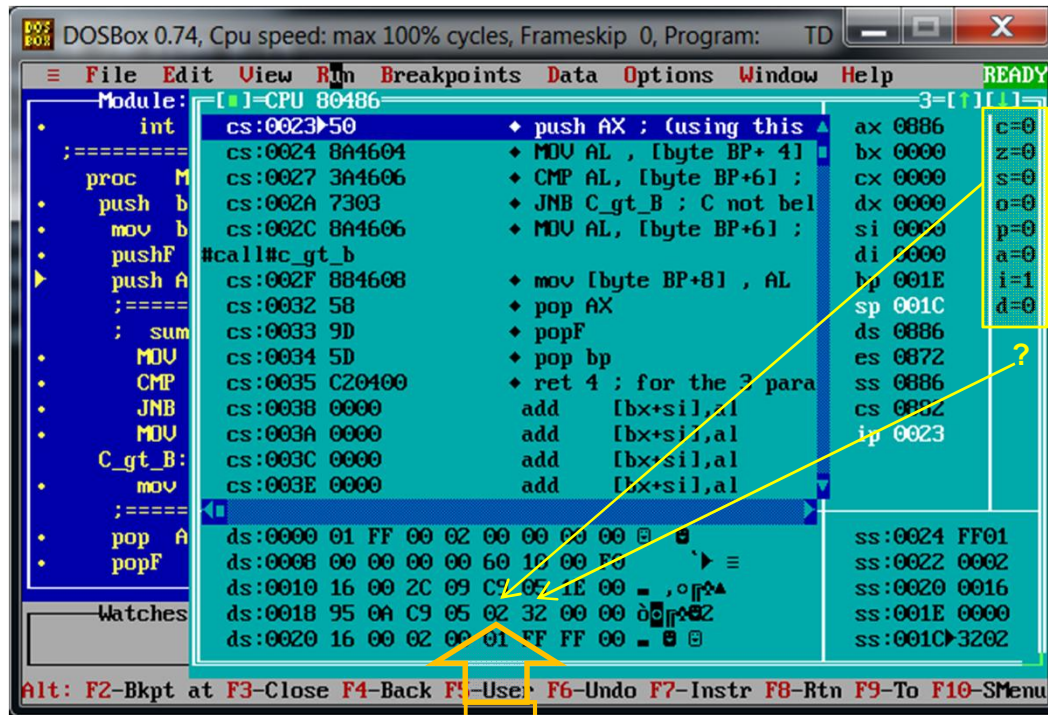
- עכשיו דחיפת דברים נוספים למחסנית לא תשנה את המרחק בין BP לכתובת הארגומנטים שמתקשרים עם התכנית שקראה לפונקציה (" לקוח ")
- (כולל שימוש למשתנים מקומיים , לקריאה לפונקציות אחרות, ( ... )
- (הפקודה הבאה : שמירת ערך מקורי של הדגלים, לפני שהפונקציה תעשה פעולות שישנו אותו, כך שנוכל לשחזר ביציאה).



# 8086 CALL PROCESS - 11

(אחרי העתקת 8 ביטים של דגלים למחסנית):

- הדגלים בעלי ערך 2, נשמר במחסנית
- למחסנית נכנס גם בית נוסף ( PUSH זה 16 ביט ) - לא רלוונטי (ואולי כן?? אם הייתי האקר הייתי חוקר ע"י ניסויי נגיעה בו. אבל תכנתת "רגיל" - לא כדאי לחשוב על זה בכלל)

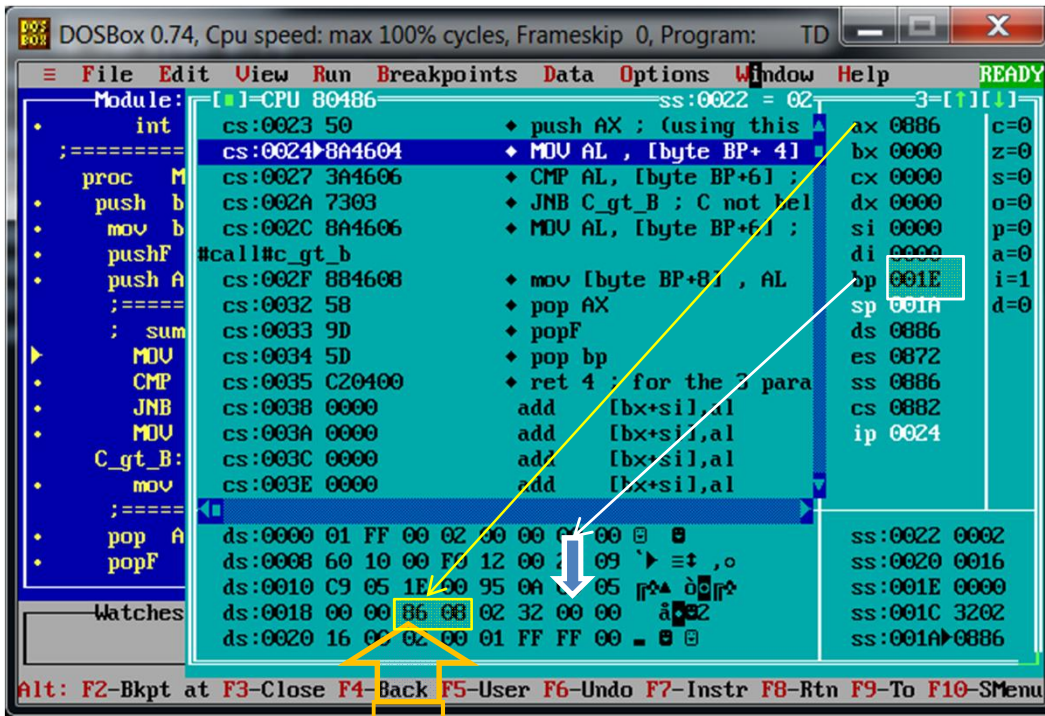


# 8086 CALL PROCESS - 12

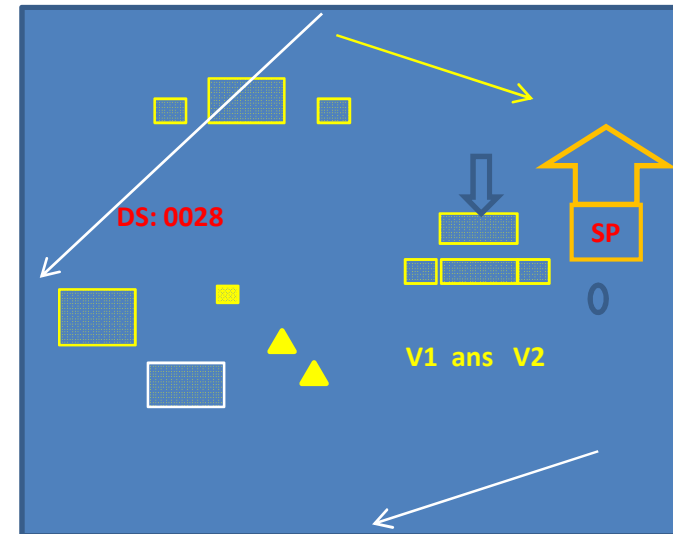
• עכשיו שימרנו גם את הערך המקורי של AX

• בצעד הבא נשלוף את המשתנה השני מהמחסנית השליפה כקריאת זכרון מ [BP+4] (במקור: V2)

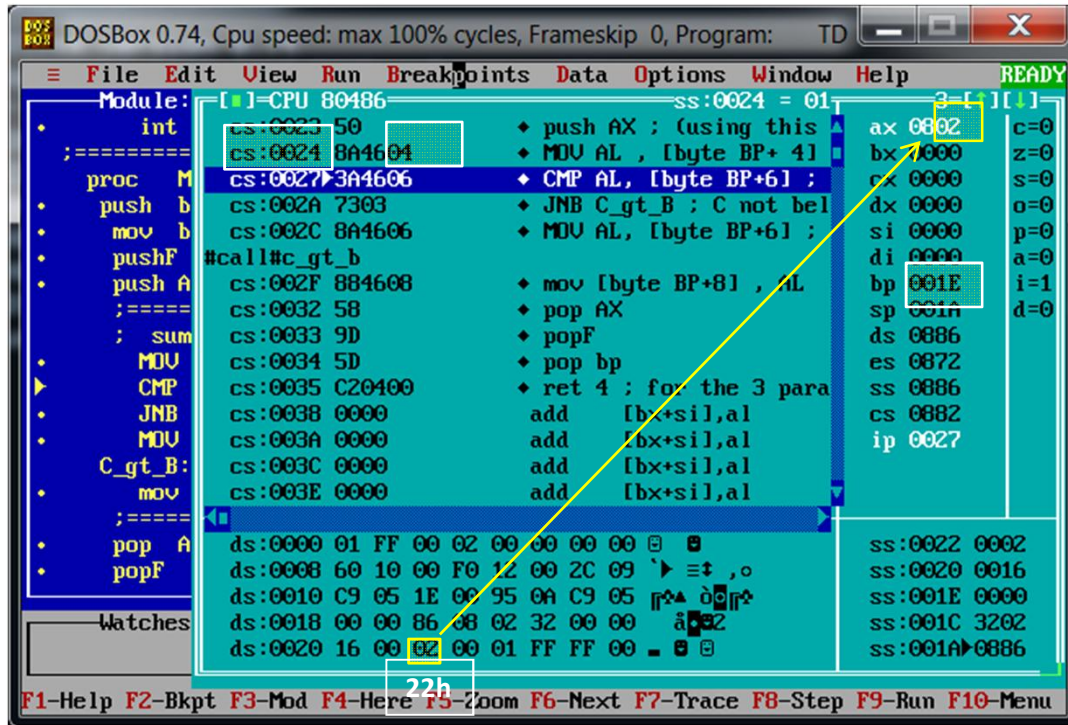
ואז נשווה למשתנה הראשון מהמחסנית הפנייה כקריאת זכרון מ [BP+6] (במקור: V1)



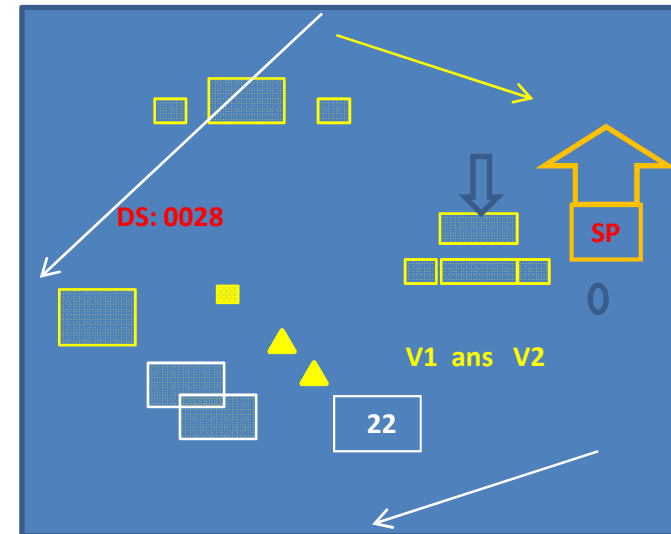
SP



# 8086 CALL PROCESS - 13



- אחרי משיכת המשתנה השני מהמחסנית :
- החץ מראה מי הועתק לאן (ע"י `MOV AL, [bp+4]`)
- הבית השלישי בפקודת `MOV` (כתובת 26h) מעיד על כך שהאופסט הוא 4







# 8086 CALL PROCESS - 15

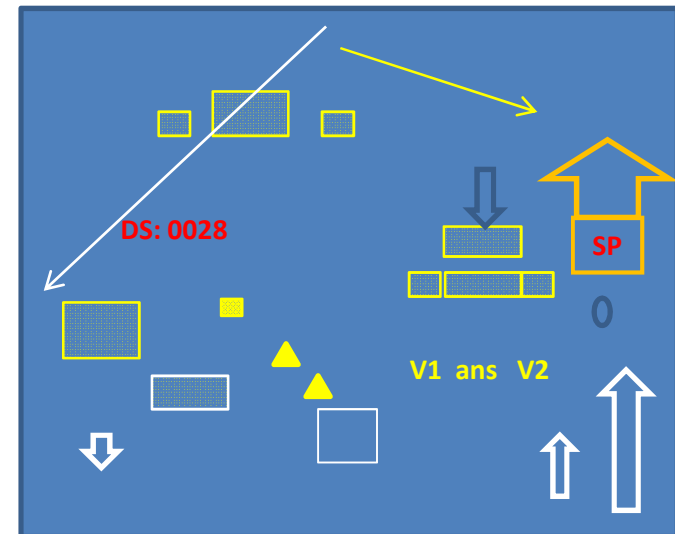
- אכן הקפיצה בוצעה (דילגנו על פקודת MOV אחת)
- עומד להחזיר את AL למחסנית,

```

Module: [I]-CPU 80486 ss:0026 = FF 3-[[[[]]]
cs:0023 50      * push AX ; (using this
cs:0024 8A4604  * MOV AL , [byte BP+ 4]
cs:0027 3A4606  * CMP AL, [byte BP+6] ;
cs:002A 7303    * JNB C_gt_B : C not bel
cs:002C 8A4606  * MOV AL, [byte BP+6] ;
cs:002F 8B4608  * mov [byte BP+8] , AL
cs:0032 58      * pop AX
cs:0033 9D      * popF
cs:0034 5D      * pop bp
cs:0035 C20400  * ret 4 ; for the 3 para
cs:0038 0000    add [bx+si],al
cs:003A 0000    add [bx+si],al
cs:003C 0000    add [bx+si],al
cs:003E 0000    add [bx+si],al
ds:0000 01 FF 00 02 00 00 00 00
ds:0008 60 10 00 F0 12 00 2C 09
ds:0010 C9 05 1E 00 95 0A C9 05
ds:0018 00 00 B6 08 02 32 00 00
ds:0020 16 00 02 00 01 FF FF 00
ax 0002
bx 0000
cx 0000
dx 0000
si 0000
di 0000
bp 001E
sp 001A
ds 0006
es 0072
ss 0086
cs 0082
ip 002F

```

למקום שהכנו מראש ע"י ה PUSH הראשון  
(זה שבאחריות הקורא למשוך ע"י POP )

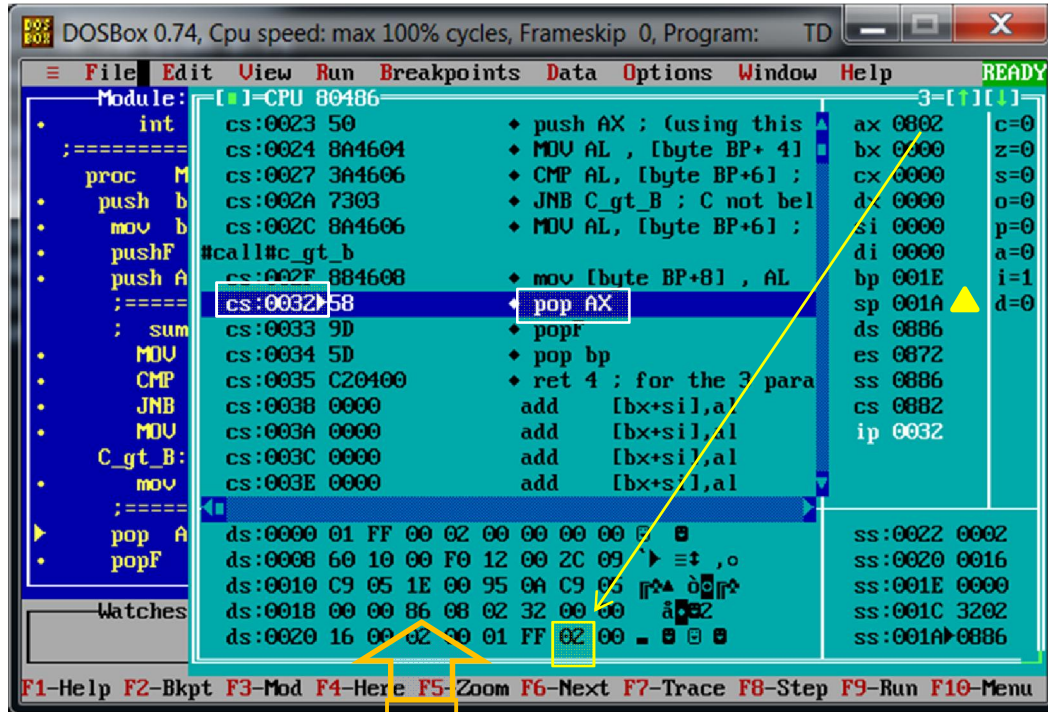


# 8086 CALL PROCESS - 16

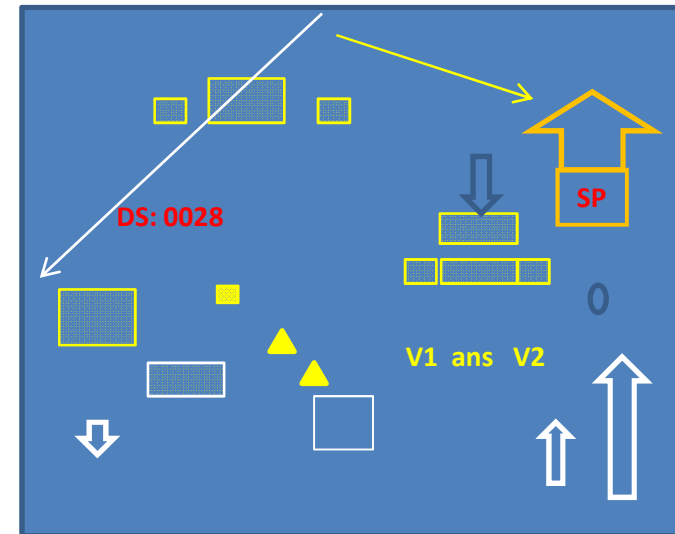
- ואכן AL (התוצאה) הועבר למחסנית,

מיקומו בתא הראשון שנדחף ע"י התכנית הקוראת

- (ואנו עומדים להחזיר את AX המקורי מהמחסנית)



SP

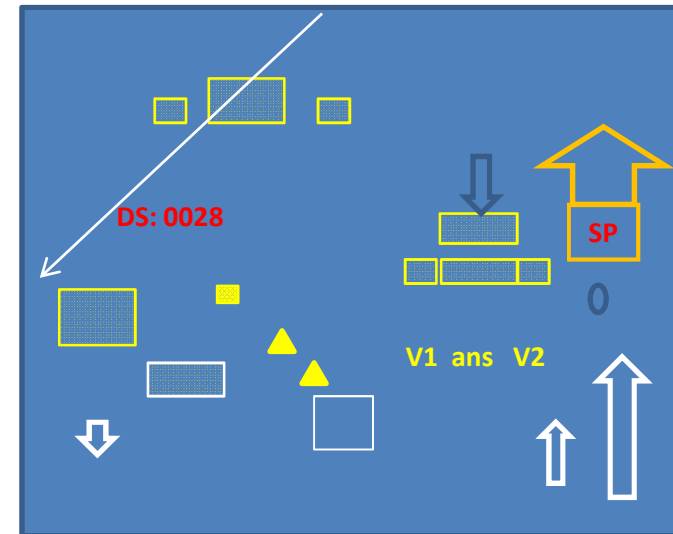
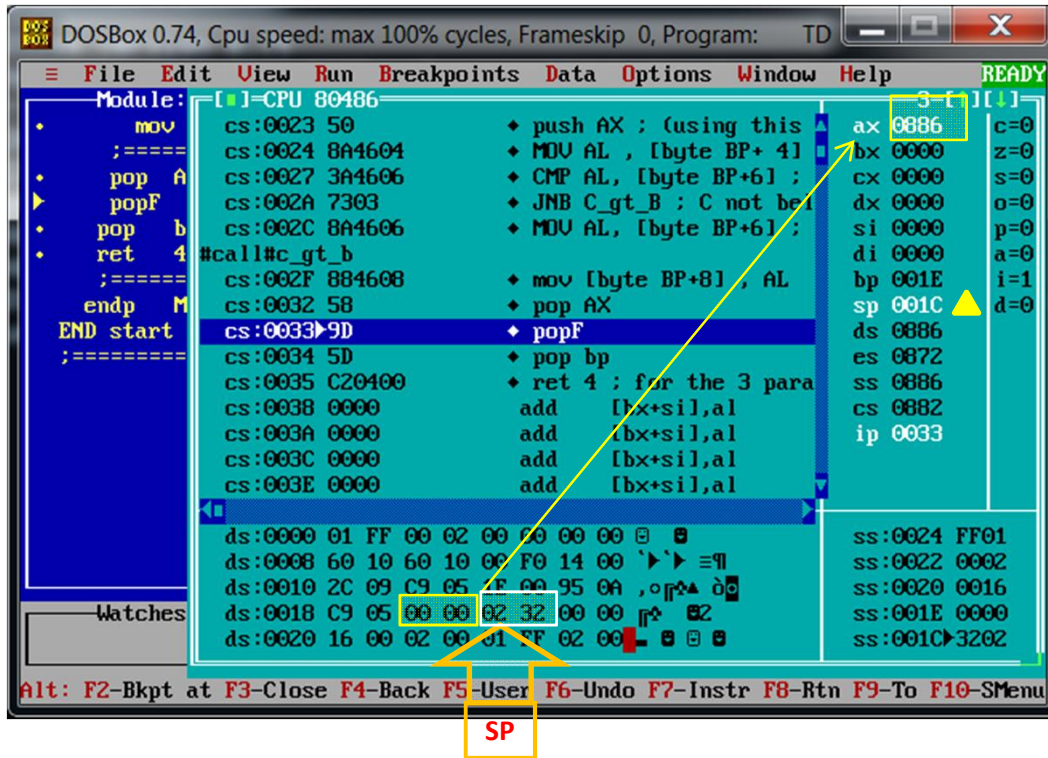


# 8086 CALL PROCESS - 17

- והחזיר את AX המקורי מהמחסנית למחסנית:

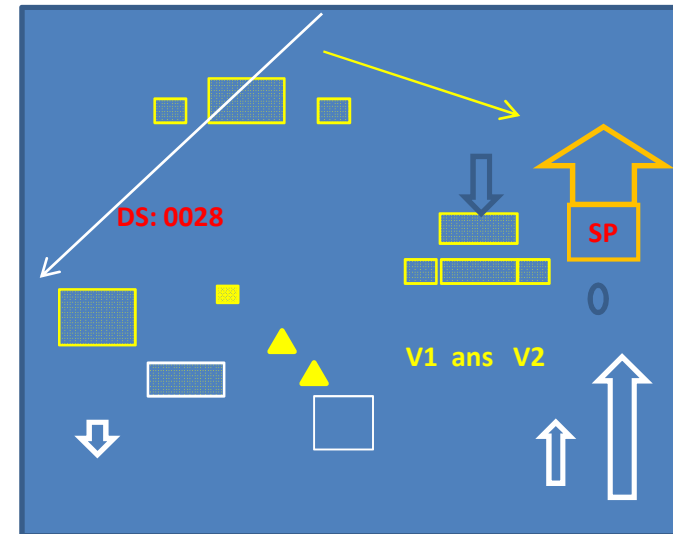
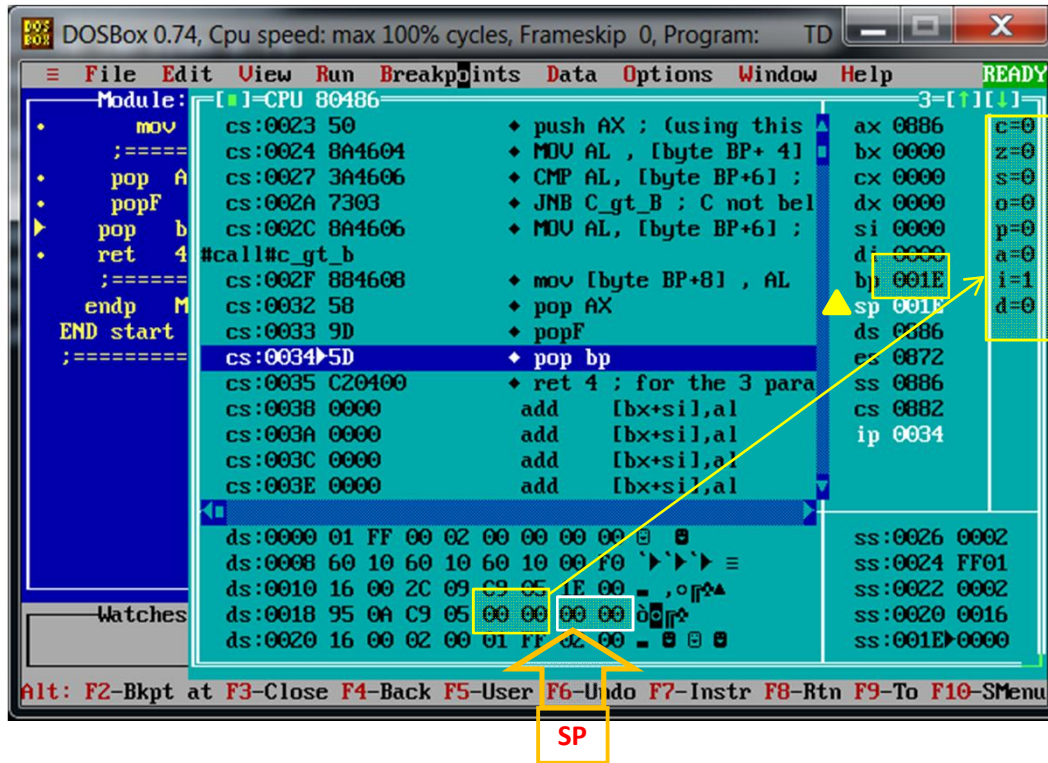
(לא להתבלבל – הנתון נמחק מהמחסנית רק משום ש TD החליט מיד אחרי הביצוע שהתא פנוי והוא יכול "לפלוש לדירה הריקה" כדי לבצע את "עבודות השירות")

- ועומד לשחזר את הדגלים המקוריים (הפקודה הבאה: popF) (במקרה ערכם 02)

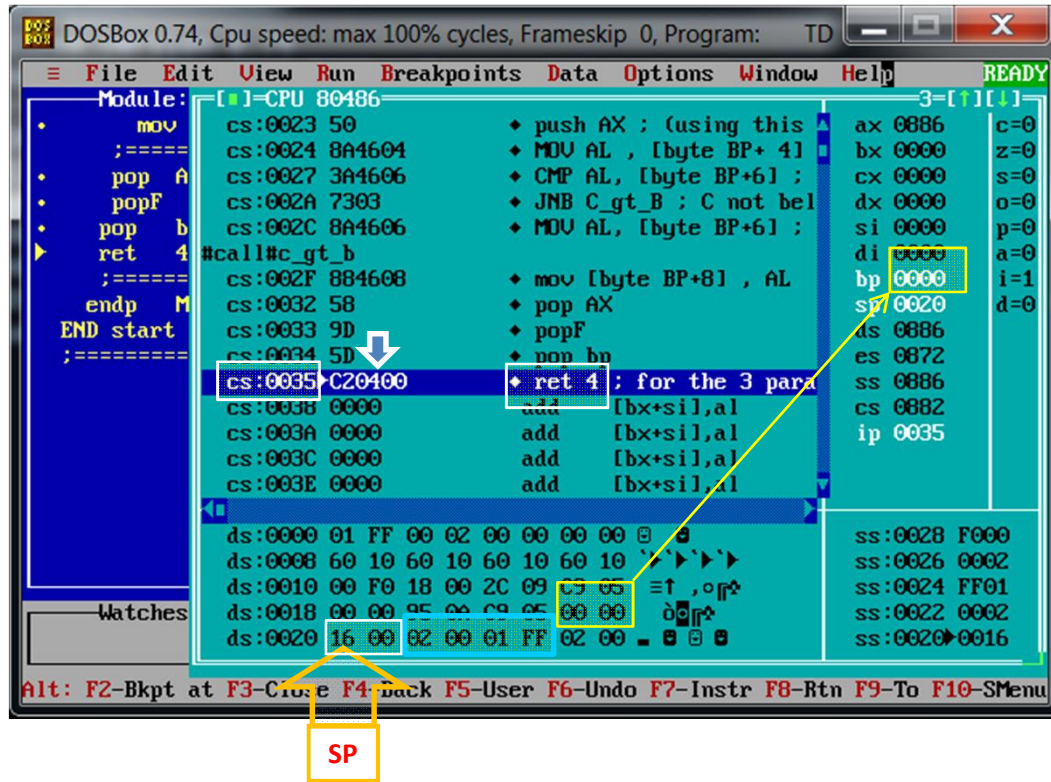


# 8086 CALL PROCESS - 18

- והדגלים שוחררו גם,
- (וגם הפעם TD השתלט מיד על השטח המשוחרר (ודרס)
- ועומד לשחרר את BP מהמחסנית: (ערכו עדין 1Eh)



# 8086 CALL PROCESS - 20



• אכן גם BP שוחזר,

• והגענו לרגע הגדול:

• SP מצביע על כתובת הקפיצה חזרה

• שלש מילים נדחפו למחסנית מחוץ לפונקציה.

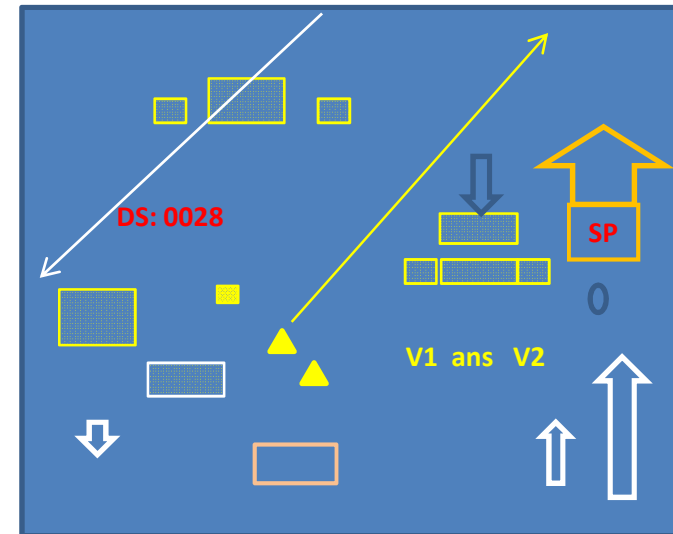
וקשה לעשות PUSH מתוך הפונקציה

• המתכננים הלכו לקראתנו ומימשו "קיצור דרך"

**RET+4 יתורגם לשפת מכונה שפרושה:**

**" .. והקפץ את 4 SP בתים נוספים "**

( ואנו רואים את זה בבית השני של הפקודה בכתובת 0036h )



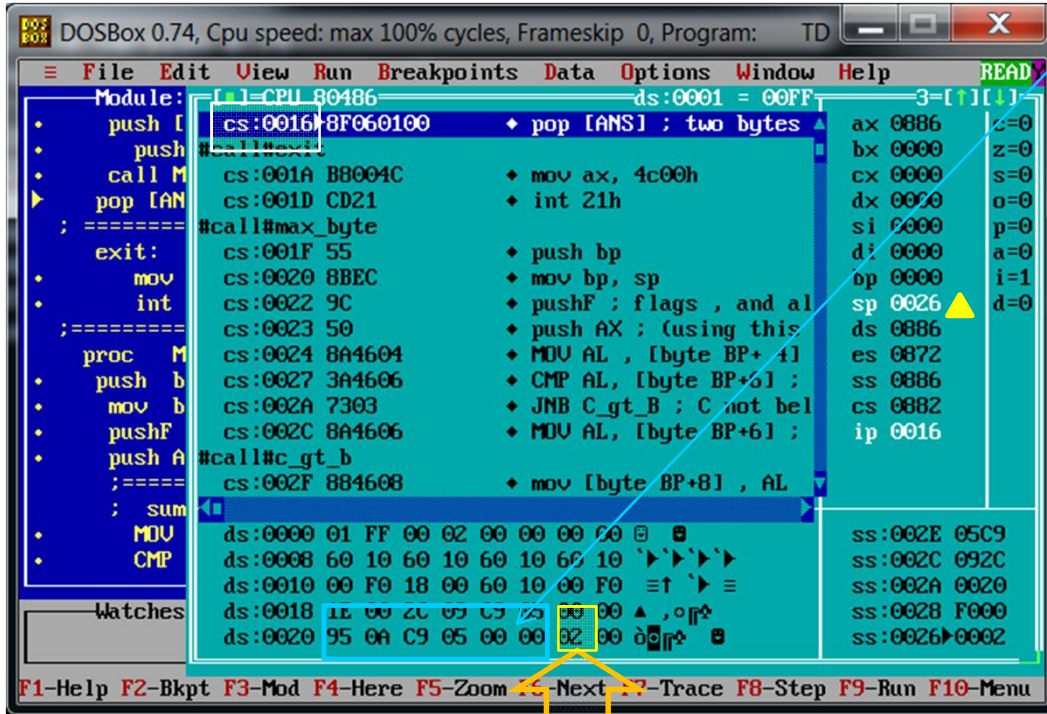
# 8086 CALL PROCESS - 21

## אכן הקפיצה חזרה מהפונקציה הושלמה

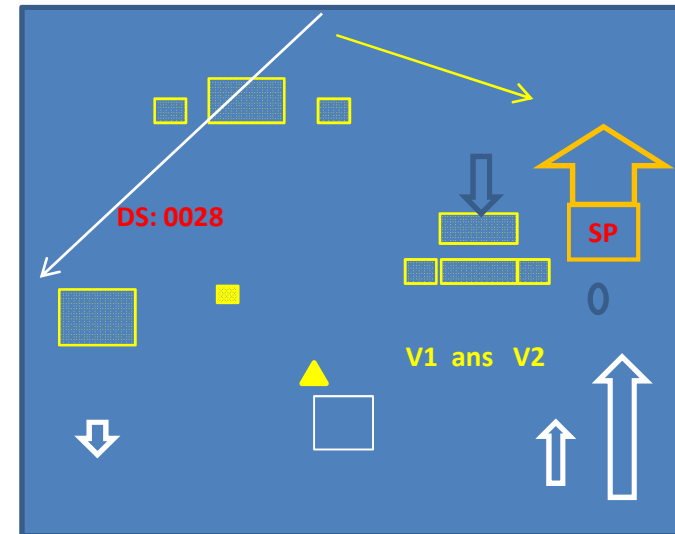
ו 6 הבתים ששחררו מהמחסנית נדרסו ע"י TD  
 תוך כדי מתן "שרות מסירת דין וחשבון למפעיל"

**SP מצביע על כתובת הנתון שטרם נשלף (26h)**  
 (הערך המקורי של SP היה 28h כלומר: "עוד POP וגמרנו")

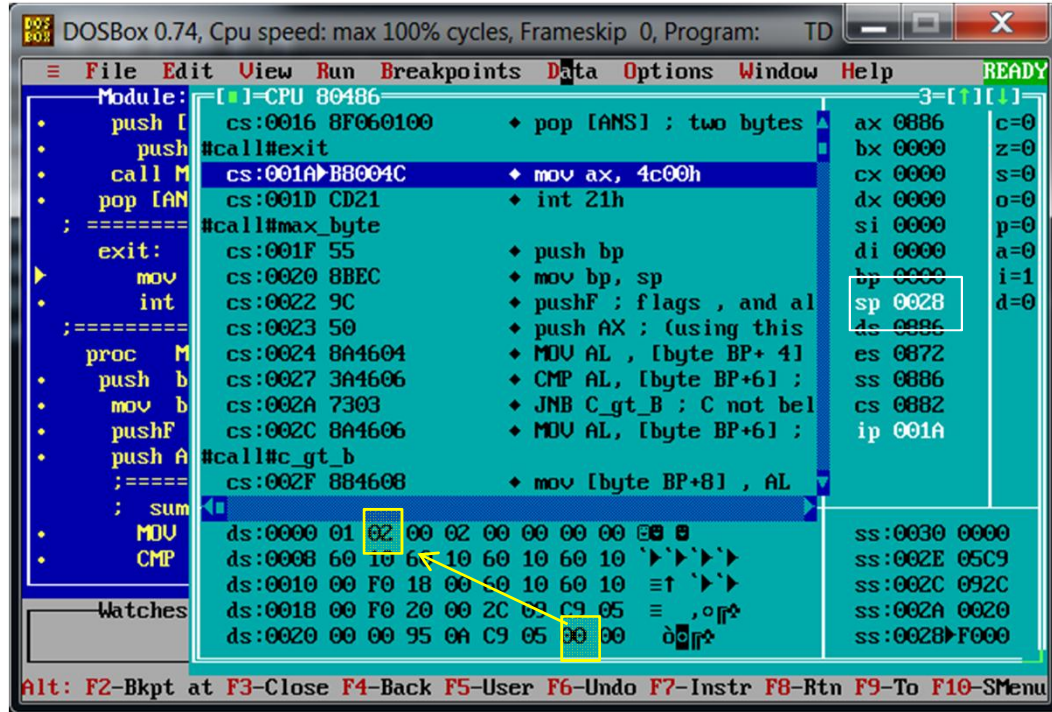
ואנו לפני ביצוע הפקודה האחרונה שעומדת להחזיר את  
 התשובה מהמחסנית לכתובת קבועה בזכרון:



SP



# 8086 CALL PROCESS - 22



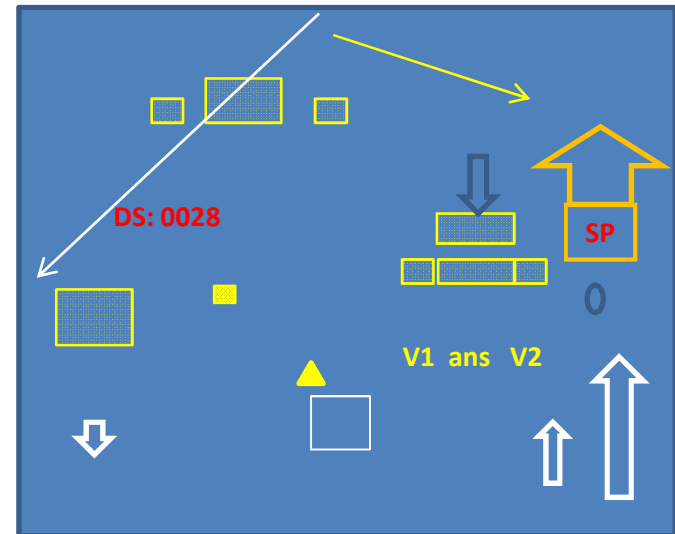
## השליפה האחרונה הסתיימה

התוצאה במקום  
(נכון ש TD דרס ... התרגלנו ל "אנומליה" הזו)

**SP חזר לערכו המקורי**  
(חשוב לוודא - או שתהיה "זליגת זכרון")

שאר הרגיסטרים חזרו לערכם המקורי  
(מדוע למי ולמה זה חשוב?)

והתכנית יכולה להמשיך בדרכה  
(במקרה הזה - ממשיכה לאן?)

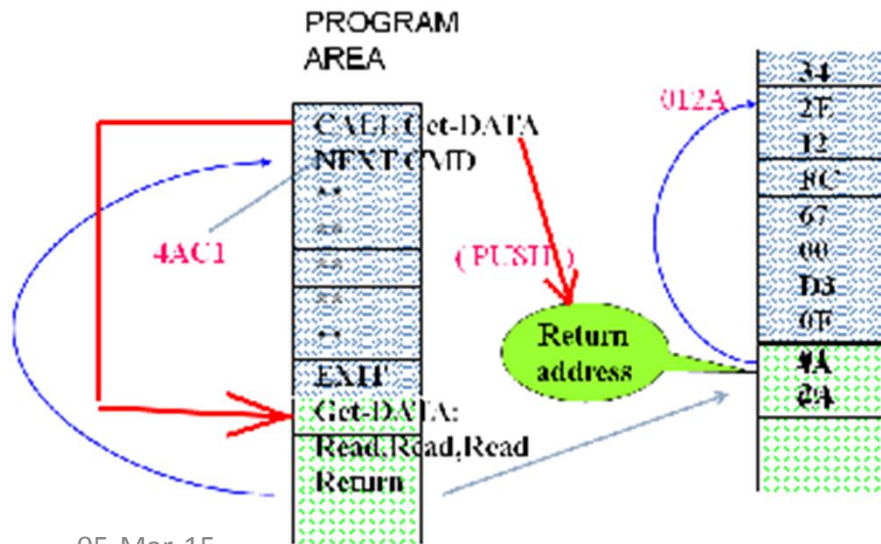


# 8086 CALL PROCESS - 23

- סיכום
- מבוא הזכיר לנו שהבנת מבנה המכונה משפיע על יכולתנו "ללחוץ על הכפתורים הנכונים"
- הודגם השימוש ב Turbo Debugger :
  - לצורך הבנת תהליכים
  - הרצה על מקרה קל (JMP)
  - הדגמה של מקרה מורכב יותר (CALL עם העברת פרמטרים במחסנית)
- הודגמו כמה "טכניקות וטריקים" לצורך עבודה יעילה עם ה TD
- ה"בונוס": הנ"ל מסייע לצורך:
  - אימות ובדיקת קוד (מהלך תוכנה, או שינוי ערכים באמצע)
  - דיבוג תקלות בזמן פיתוח
  - "הנדסה לאחור" (למשל - בתחרות. למשל - בעת תחזוקת קוד של עמית)

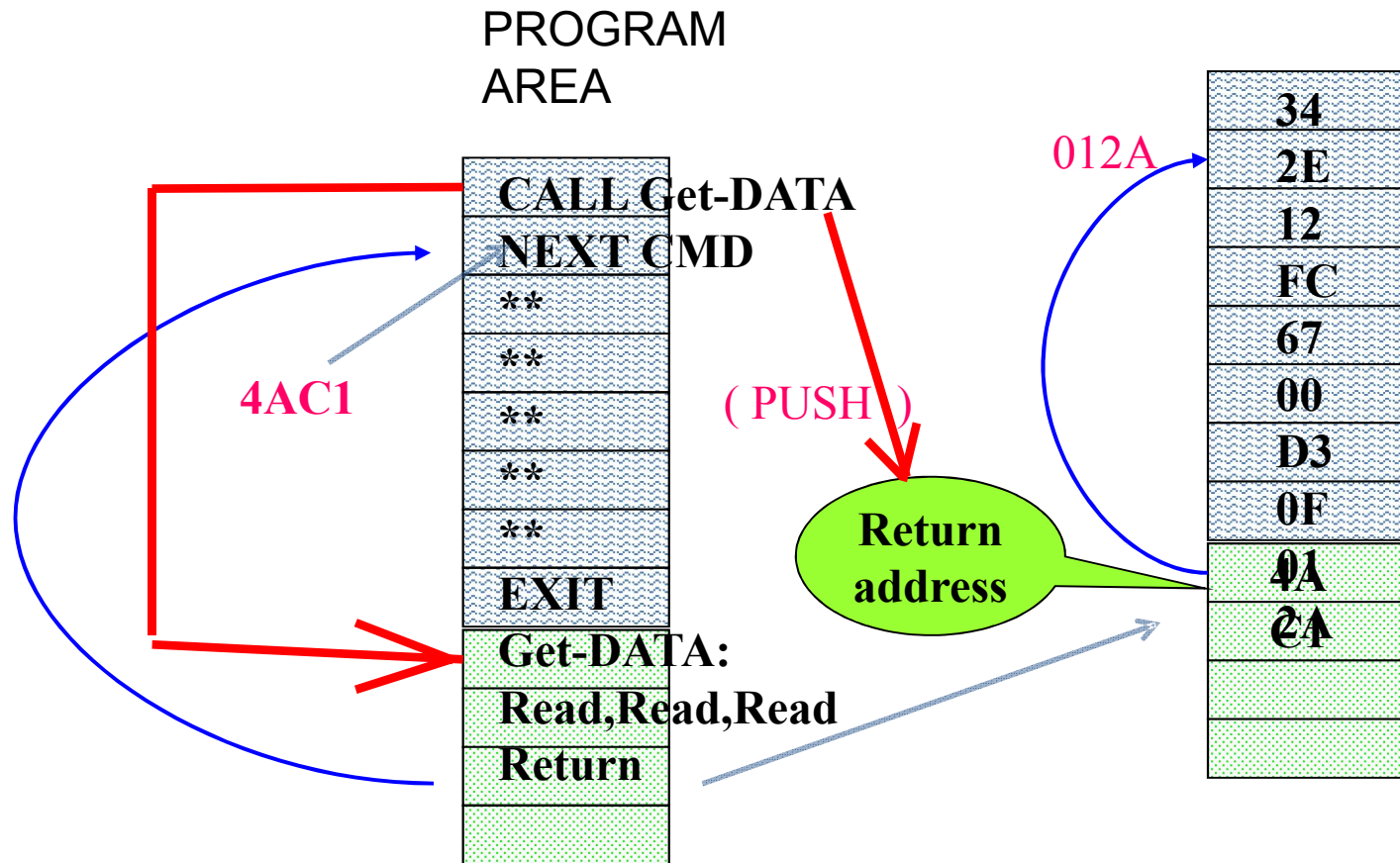
## הערות

- יש יתרון להדגמה על "הדבר האמיתי"
- החסרון: הטרחה (באמצעים הקיימים) - עצומה
- הטכניקה הקלה יותר היא שקפים שמדגימים:
  - אז אחרי שהשתכנענו: נוכל לעבור אליה





• מניעת טעויות ,



טמפלאיט ראשוני להצגה עם אנימציה בשקפים (הצורך: הדגמת תהליך תוכנה ב CPU קל ליצירה)

הארה: צורך הוא פתרון לצורך "בסיסי" מסויים. הצורך הזה קיים עד אשר הבעייה הראשונית נעלמה, או שגילינו פתרון טוב יותר ...

ADDR	main prog	REGISTERS	ADDR	Subroutine	ADDR	STACK
01		AX	101		201	
02		BX	102		202	
03		CX	103		203	
04		DX	104		204	
05			105		205	
06		SP	106		206	
07		BP	107		207	
08		Di	108		208	
09		Si	109		209	
10			110		210	
11		IP	111		211	
12			112		212	
13		Flags	113		213	
14			114		214	
15			115		215	
16			116		216	
17			117		217	
18			118		218	
19			119		219	
20			120		220	
21			121		221	
22			122		222	
23			123		223	
24			124		224	
25			125		225	
26			126		226	
27			127		227	
28			128		228	
29			129		229	
30			130		230	
31			131		231	
32			132		232	